Android 中高级面试必知必会

目录

Android 中高级面试必知必会.....1
第一章 Java 相关高频面试解析.....3
1. HashMap.....3
2. ArrayList.....25
3.LinkedList.....30
4.Hashset 源码分析.....35
5. 内存模型.....47
6. 垃圾回收算法（JVM）.....63
7、垃圾回收机制和调用 System.gc()的区别？.....64
8. 类加载过程.....90
9. 反射.....101
10. 多线程和线程池.....112
11.HTTP、HTTPS、TCP/IP、Socket 通信、三次握手四次挥手过程.....113
12.设计模式（六大基本原则、项目中常用的设计模式、手写单例等）.....120
13.断点续传.....134
14.Java 四大引用.....141
15.Java 的泛型.....146
16. final、finally、finalize 的区别.....157
17.接口、抽象类的区别.....163
第二章 Android d 面试题解析大全.....170
1.自定义 View.....170
. 1. 事件拦截分发.....174
. 2. 解决过的一些性能问题，在项目中的实际运用.....175
. 3. 性能优化工具.....175
. 4. 性能优化 （讲讲你自己项目中做过的性能优化）.....180
. 6. Http[s] 请求慢的解决办法 （ DNS 、 携问 带数据、直接访问 IP ）.....180
. 7. 缓存自己如何实现 （e LRUCache 原理 ）.....180
. 8. 图形图像相关：L OpenGL S ES 管线流程、L EGL 的认识、r Shader 相关.....180
. 9. SurfaceView 、 TextureView 、GLSurfaceView 区别及使用场景.....180
. 10. 动画、差值器、估值器（ Android d 中的 w View 画 动画和属性动画 - - 简书、Android 画 动画 介绍与使用）.....180
. 11. MVC 、 MVP 、 MVVM.....181
. 12. Handler、 、 ThreadLocal、 、 AsyncTask 、 IntentService 原理及应用.....181
. 13. Gradle （y Groovy 语法、e Gradle 插件开发基础）.....181
. 14. 热修复、插件化.....190
. 15. 组件化架构思路.....190
. 16. 系统打包流程.....213

17. Android 有哪些存储数据的方式。 214
. 18. SharedPrefrence 源码和问题点； 235
. 19. sqlite 相关 236
. 20. 如何判断一个 P APP 在前台还是后台？ 253
. 21. 混合开发 258
Android Framework 高频面试题总结 259
第三章：网络相关面试题 316
一、HTTP/HTTPS 316
二、 TCP/UDP 325
三、其它重要网络概念 328
四、常见网络流程机制 332
第四章：三方源码高频面试总结 332
1.Glide ：加载、缓存、LRU 算法 (如何自己设计一个大图加载框架)（LRUCache 原理） 332
2.LeakCanary 407
3.ARouter 421
4.RXJava （RxJava 的线程切换原理） 446
5.Retrofit （Retrofit 在 OkHttp 上做了哪些封装？动态代理和静态代理的区别，是怎么实现的） 462
6.OkHttp 481
第五章 Kotlin 相关 498
1.从原理分析 Kotlin 的延迟初始化: lateinit var 和 by lazy 498
2.From Java To Kotlin 504
3.怎么用 Kotlin 去提高生产力：Kotlin Tips 515
4.使用 Kotlin Reified 让泛型更简单安全 550
5.Kotlin 里的 Extension Functions 实现原理分析 559
6.Kotlin 系列之顶层函数和属性 562
7.Kotlin 兼容 Java 遇到的最大的“坑” 568
8.Kotlin 的协程 576
9.Kotlin 协程「挂起」的本质 588
10.到底什么是「非阻塞式」挂起？协程真的更轻量级吗？ 597
11.资源混淆是如何影响到 Kotlin 协程的 600
12.破解 Kotlin 协程 603
第六章 Flutter 相关 615
1. Dart 当中的 「..」表示什么意思？ 615
2. Dart 的作用域 615
3.Dart 是不是单线程模型？是如何运行的？ 616
4. Dart 是如何实现多任务并行的？ 617
5. 说一下 Dart 异步编程中的 Future 关键字？ 617
6. 说一下 Dart 异步编程中的 Stream 数据流？ 618
7.Stream 有哪两种订阅模式？分别是怎么调用的？ 618
8.await for 如何使用？ 619
9. 说一下 mixin 机制？ 619

Flutter.....620
1. 请简单介绍下 Flutter 框架，以及它的优缺点？.....620
2. 介绍下 Flutter 的理念架构.....621
3. 介绍下 FFlutter 的 FrameWork 层和 Engine 层，以及它们的作用.....622
4. 介绍下 Widget、State、Context 概念.....622
5. 简述 Widget 的 StatelessWidget 和 StatefulWidget 两种状态组件类.....623
6. StatefulWidget 的生命周期.....624
7. 简述 Widgets、RenderObjects 和 Elements 的关系.....625
8. 什么是状态管理，你了解哪些状态管理框架？.....626
9. 简述 Flutter 的绘制流程.....626
10. 简述 Flutter 的线程管理模型.....627
11. Flutter 是如何与原生 Android、iOS 进行通信的？.....628
12. 简述 Flutter 的热重载.....628

第一章 Java 相关高频面试解析

1. HashMap

（1）问：HashMap 有用过吗？您能给我说说他的主要用途吗？

答：有用过，我在平常工作中经常会用到 HashMap 这种数据结构，HashMap 是基于 Map 接口实现的一种键-值对<key,value>的存储结构，允许 null 值，同时非有序，非同步(即线程不安全)。HashMap 的底层实现是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分）。它存储和查找数据时，是根据键 key 的 hashCode 的值计算出具体的存储位置。HashMap 最多只允许一条记录的键 key 为 null，HashMap 增删改查等常规操作都有不错的执行效率，是 ArrayList 和 LinkedList 等数据结构的一种折中实现。

示例代码：

```
// 创建一个 HashMap，如果没有指定初始大小，默认底层 hash 表数组的大小为 16
HashMap<String, String> hashMap = new HashMap<String, String>();
// 往容器里面添加元素
hashMap.put("小明", "好帅");
hashMap.put("老王", "坑爹货");
hashMap.put("老铁", "没毛病");
hashMap.put("掘金", "好地方");
hashMap.put("王五", "别搞事");
```

```
// 获取 key 为小明的元素 好帅
String element = hashMap.get("小明");
// value : 好帅
System.out.println(element);
// 移除 key 为王五的元素
String removeElement = hashMap.remove("王五");
// value : 别搞事
System.out.println(removeElement);
// 修改 key 为小明的元素的值 value 为 其实有点丑
hashMap.replace("小明", "其实有点丑");
// {老铁=没毛病, 小明=其实有点丑, 老王=坑爹货, 掘金=好地方}
System.out.println(hashMap);
// 通过 put 方法也可以达到修改对应元素的值的效果
hashMap.put("小明", "其实还可以啦,开玩笑的");
// {老铁=没毛病, 小明=其实还可以啦,开玩笑的, 老王=坑爹货, 掘金=好地方}
System.out.println(hashMap);
// 判断 key 为老王的元素是否存在(捉奸老王)
boolean isExist = hashMap.containsKey("老王");
// true , 老王竟然来搞事
System.out.println(isExist);
// 判断是否有 value = "坑爹货" 的人
boolean isHasSomeOne = hashMap.containsValue("坑爹货");
// true 老王是坑爹货
System.out.println(isHasSomeOne);
// 查看这个容器里面还有几个家伙 value : 4
System.out.println(hashMap.size());
```

- HashMap 的底层实现是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分），核心组成元素有：

  int size;用于记录 HashMap 实际存储元素的个数；

  float loadFactor;负载因子（默认是 0.75，此属性后面详细解释）。

  int threshold;下一次扩容时的阈值，达到阈值便会触发扩容机制 resize（阈值 threshold = 容器容量 capacity * 负载因子 load factor）。也就是说，在容器定义好容量之后，负载因子越大，所能容纳的键值对元素个数就越多。

  Node<K,V>[] table; 底层数组，充当哈希表的作用，用于存储对应 hash 位置的元素 Node<K,V>，此数组长度总是 2 的 N 次幂。（具体原因后面详细解释）

示例代码：

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
.....

    /* ---------------- Fields -------------- */

    /**
     * 哈希表，在第一次使用到时进行初始化，重置大小是必要的操作，
     * 当分配容量时，长度总是 2 的 N 次幂。
     */
    transient Node<K,V>[] table;

    /**
     * 实际存储的 key - value 键值对 个数
     */
    transient int size;


    /**
     * 下一次扩容时的阈值
     * (阈值 threshold = 容器容量 capacity * 负载因子 load factor).
     * @serial
     */
    int threshold;

    /**
     * 哈希表的负载因子
     *
     * @serial
     */
    final float loadFactor;

.....}
```

- 其中 Node<K,V>[] table;哈希表存储的核心元素是 Node<K,V>,Node<K,V>包含：

  final int hash;元素的哈希值，决定元素存储在 Node<K,V>[] table;哈希表中的位置。由 final 修饰可知，当 hash 的值确定后，就不能再修改。

  final K key; 键，由 final 修饰可知，当 key 的值确定后，就不能再修改。

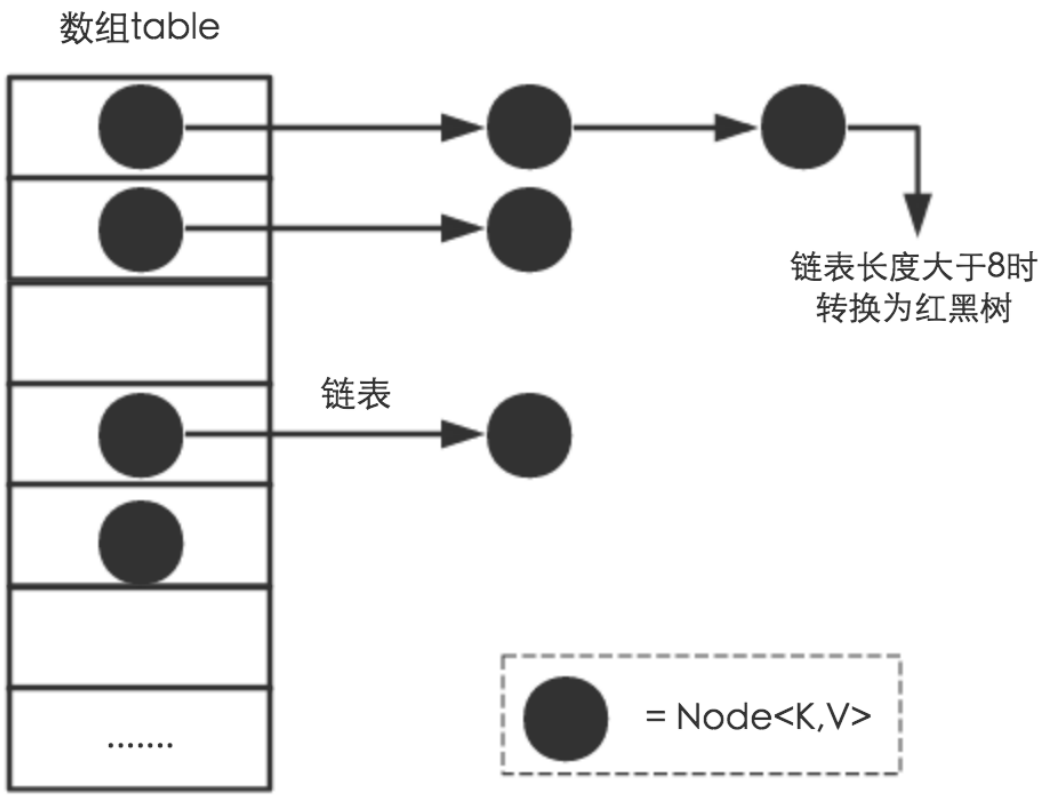
  V value; 值

  Node<K,V> next; 记录下一个元素结点(单链表结构，用于解决 hash 冲突)

示例代码：

```
    /**
     * 定义 HashMap 存储元素结点的底层实现
     */
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;//元素的哈希值 由 final 修饰可知，当 hash 的值确定后，就不能再修改
        final K key;// 键，由 final 修饰可知，当 key 的值确定后，就不能再修改
        V value; // 值
        Node<K,V> next; // 记录下一个元素结点(单链表结构，用于解决 hash 冲突)


        /**
         * Node 结点构造方法
         */
        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;//元素的哈希值
            this.key = key;// 键
            this.value = value; // 值
            this.next = next;// 记录下一个元素结点
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        /**
         * 为 Node 重写 hashCode 方法，值为：key 的 hashCode 异或 value 的 hashCode
         * 运算作用就是将 2 个 hashCode 的二进制中,同一位置相同的值为 0，不同的为 1。
         */
        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        /**
         * 修改某一元素的值
         */
        public final V setValue(V newValue) {
```

```
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        /**
         * 为Node 重写 equals 方法
         */
        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }
```



hashMap 内存结构图 - 图片来自于《美团点评技术团队文章》

2.问：您能说说 HashMap 常用操作的底层实现原理吗？如存储 put(K key, V value)，查找 get(Object key)，删除 remove(Object key)，修改 replace(K key, V value)等操作。

答：调用 put(K key, V value)操作添加 key-value 键值对时，进行了如下操作：

判断哈希表 Node<K,V>[] table 是否为空或者 null，是则执行 resize()方法进行扩容。

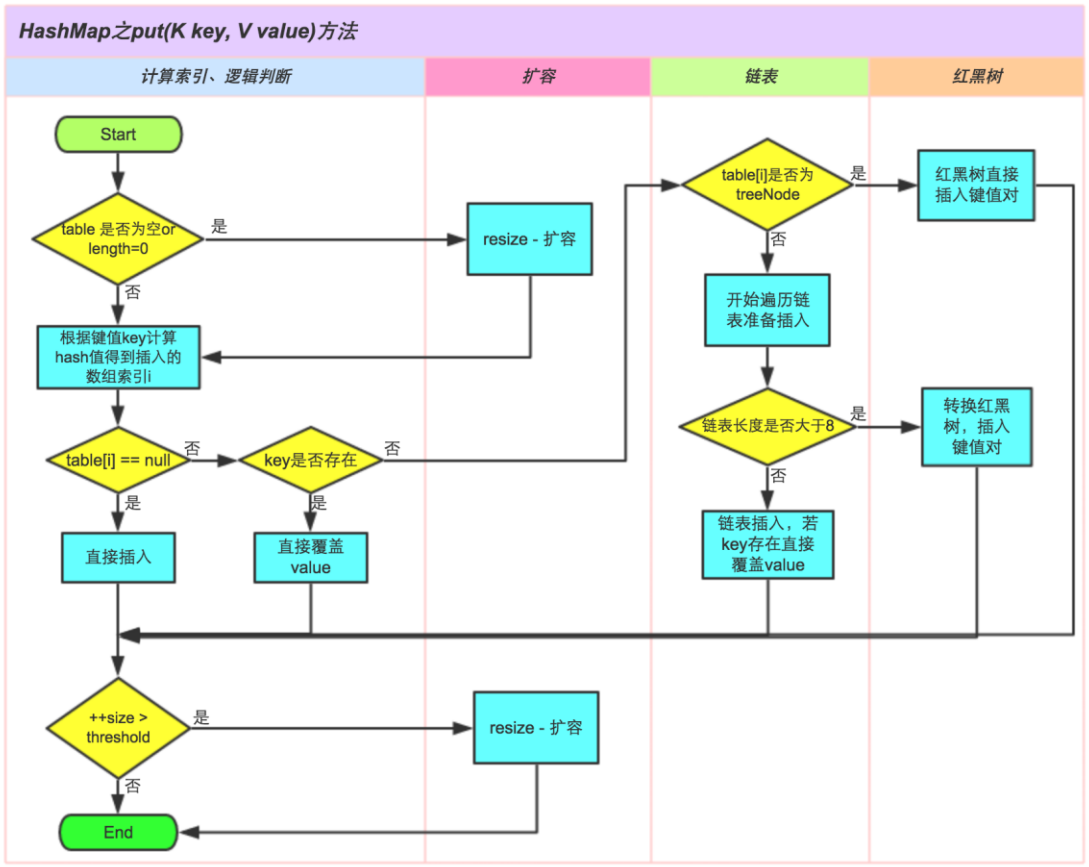
根据插入的键值 key 的 hash 值，通过(n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1（实际就是 hash 值 % hash 表长度） 计算出存储位置 table[i]。如果存储位置没有元素存放，则将新增结点存储在此位置 table[i]。

如果存储位置已经有键值对元素存在，则判断该位置元素的 hash 值和 key 值是否和当前操作元素一致，一致则证明是修改 value 操作，覆盖 value 即可。

当前存储位置即有元素，又不和当前操作元素一致，则证明此位置 table[i]已经发生了 hash 冲突，则通过判断头结点是否是 treeNode，如果是 treeNode 则证明此位置的结构是红黑树，已红黑树的方式新增结点。

如果不是红黑树，则证明是单链表，将新增结点插入至链表的最后位置，随后判断当前链表长度是否 大于等于 8，是则将当前存储位置的链表转化为红黑树。遍历过程中如果发现 key 已经存在，则直接覆盖 value。

插入成功后，判断当前存储键值对的数量 大于 阈值 threshold 是则扩容。



hashMap put 方法执行流程图- 图片来自于《美团点评技术团队文章》

示例代码：

```
/**
 * 添加 key-value 键值对
 *
 * @param key 键
 * @param value 值
 *@return 如果原本存在此 key，则返回旧的 value 值，如果是新增的 key-
 *         value，则返回 nulll
 */
public V put(K key, V value) {
    //实际调用 putVal 方法进行添加 key-value 键值对操作
    return putVal(hash(key), key, value, false, true);
}

/**
 * 根据 key 键 的 hashCode 通过 “扰动函数” 生成对应的 hash 值
 * 经过此操作后，使每一个 key 对应的 hash 值生成的更均匀，
 * 减少元素之间的碰撞几率（后面详细说明）
 */
```

```
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }


    /**
     * 添加 key-value 键值对的实际调用方法（重点）
     *
     * @param hash key 键的 hash 值
     * @param key 键
     * @param value 值
     * @param onlyIfAbsent 此值如果是 true, 则如果此 key 已存在 value，则不执
     * 行修改操作
     * @param evict 此值如果是 false，哈希表是在初始化模式
     * @return 返回原本的旧值, 如果是新增，则返回 null
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        // 用于记录当前的 hash 表
        Node<K,V>[] tab;
        // 用于记录当前的链表结点
        Node<K,V> p;
        // n 用于记录 hash 表的长度，i 用于记录当前操作索引 index
        int n, i;
        // 当前 hash 表为空
        if ((tab = table) == null || (n = tab.length) == 0)
            // 初始化 hash 表，并把初始化后的 hash 表长度值赋值给 n
            n = (tab = resize()).length;
        // 1）通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
        // 2）确定当前元素的存储位置，此运算等价于 当前元素的 hash 值 % hash 表的长度
        // 3）计算出的存储位置没有元素存在
        if ((p = tab[i = (n - 1) & hash]) == null)
            // 4) 则新建一个 Node 结点，在该位置存储此元素
            tab[i] = newNode(hash, key, value, null);
        else { // 当前存储位置已经有元素存在了(不考虑是修改的情况的话，就代表发生 hash 冲突了)
            // 用于存放新增结点
            Node<K,V> e;
            // 用于临时存在某个 key 值
            K k;
            // 1)如果当前位置已存在元素的 hash 值和新增元素的 hash 值相等
```

```
            // 2)并且 key 也相等，则证明是同一个 key 元素，想执行修改 value 操作
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;// 将当前结点引用赋值给 e
            else if (p instanceof TreeNode) // 如果当前结点是树结点
                // 则证明当前位置的链表已变成红黑树结构，则已红黑树结点结构新增元素
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {// 排除上述情况，则证明已发生 hash 冲突，并 hash 冲突位置现时的结构是单链表结构
                for (int binCount = 0; ; ++binCount) {
                    //遍历单链表，将新元素结点放置此链表的最后一位
                    if ((e = p.next) == null) {
                        // 将新元素结点放在此链表的最后一位
                        p.next = newNode(hash, key, value, null);
                        // 新增结点后，当前结点数量是否大于等于 阈值 8
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            // 大于等于 8 则将链表转换成红黑树
                            treeifyBin(tab, hash);
                        break;
                    }
                    // 如果链表中已经存在对应的 key，则覆盖 value
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            if (e != null) { // 已存在对应 key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null) //如果允许修改，则修改 value 为新值
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        // 当前存储键值对的数量 大于 阈值 是则扩容
        if (++size > threshold)
            // 重置 hash 大小,将旧 hash 表的数据逐一复制到新的 hash 表中(后面详细讲解)
```

```
            resize();
        afterNodeInsertion(evict);
        // 返回 null，则证明是新增操作，而不是修改操作
        return null;
    }
```

- 调用 get(Object key)操作根据键 key 查找对应的 key-value 键值对时，进行了如下操作：

  先调用 hash(key)方法计算出 key 的 hash 值

  根据查找的键值 key 的 hash 值，通过(n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1（实际就是 hash 值 % hash 表长度） 计算出存储位置 table[i]，判断存储位置是否有元素存在 。

  如果存储位置有元素存放，则首先比较头结点元素，如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等，并且 头结点的 key 本身 和要获取的 key 相等，则返回该位置的头结点。

  如果存储位置没有元素存放，则返回 null。

  如果存储位置有元素存放，但是头结点元素不是要查找的元素，则需要遍历该位置进行查找。

  先判断头结点是否是 treeNode，如果是 treeNode 则证明此位置的结构是红黑树，以红色树的方式遍历查找该结点，没有则返回 null。

  如果不是红黑树，则证明是单链表。遍历单链表，逐一比较链表结点，链表结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等，并且 链表结点的 key 本身 和要获取的 key 相等，则返回该结点，遍历结束仍未找到对应 key 的结点，则返回 null。

示例代码：

```
    /**
     * 返回指定 key 所映射的 value 值
     * 或者 返回 null 如果容器里不存在对应的 key
     *
     * 更确切地讲，如果此映射包含一个满足 (key==null ?
k==null :key.equals(k))
     * 的从 k 键到 v 值的映射关系，
     * 则此方法返回 v；否则返回 null。（最多只能有一个这样的映射关系。）
     *
     * 返回 null 值并不一定 表明该映射不包含该键的映射关系；
```

```
     * 也可能该映射将该键显示地映射为 null。可使用 containsKey 操作来区分这两种情况。
     *
     * @see #put(Object, Object)
     */
    public V get(Object key) {
        Node<K,V> e;
        // 1.先调用 hash(key)方法计算出 key 的 hash 值
        // 2.随后调用 getNode 方法获取对应 key 所映射的 value 值
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }


    /**
     * 获取哈希表结点的方法实现
     *
     * @param hash key 键的 hash 值
     * @param key 键
     * @return 返回对应的结点，如果结点不存在，则返回 null
     */
    final Node<K,V> getNode(int hash, Object key) {
        // 用于记录当前的 hash 表
        Node<K,V>[] tab;
        // first 用于记录对应 hash 位置的第一个结点，e 充当工作结点的作用
        Node<K,V> first, e;
        // n 用于记录 hash 表的长度
        int n;
        // 用于临时存放 Key
        K k;
        // 通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
        // 判断当前元素的存储位置是否有元素存在
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {//元素存在的情况
            // 如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等
            // 并且 头结点的 key 本身 和要获取的 key 相等
            if (first.hash == hash && // always check first node 总是检查头结点
                ((k = first.key) == key || (key != null && key.equals(k))))
                // 返回该位置的头结点
                return first;
            if ((e = first.next) != null) {// 头结点不相等
                if (first instanceof TreeNode) // 如果当前结点是树结点
                    // 则证明当前位置的链表已变成红黑树结构
                    // 通过红黑树结点的方式获取对应 key 结点
```

```
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {// 当前位置不是红黑树，则证明是单链表
                // 遍历单链表，逐一比较链表结点
                // 链表结点的key的hash值 和 要获取的key的hash值相
等
                // 并且 链表结点的key本身 和要获取的 key 相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 找到对应的结点则返回
                    return e;
            } while ((e = e.next) != null);
        }
    }
    // 通过上述查找均无找到，则返回null
    return null;
}
```

- 调用 remove(Object key)操作根据键 key 删除对应的 key-value 键值对时，进行了如下操作：

  先调用 hash(key)方法计算出 key 的 hash 值

  根据查找的键值 key 的 hash 值，通过(n - 1) & hash 当前元素的 hash 值 &hash 表长度 -1（实际就是 hash 值 %hash 表长度） 计算出存储位置 table[i]，判断存储位置是否有元素存在 。

  如果存储位置有元素存放，则首先比较头结点元素，如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等，并且 头结点的 key 本身 和要获取的 key 相等，则该位置的头结点即为要删除的结点，记录此结点至变量 node 中。

  如果存储位置没有元素存放，则没有找到对应要删除的结点，则返回 null。

  如果存储位置有元素存放，但是头结点元素不是要删除的元素，则需要遍历该位置进行查找。

  先判断头结点是否是 treeNode，如果是 treeNode 则证明此位置的结构是红黑树，以红色树的方式遍历查找并删除该结点，没有则返回 null。

  如果不是红黑树，则证明是单链表。遍历单链表，逐一比较链表结点，链表结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等，并且 链表结点的 key 本身 和要获取的 key 相等，则此为要删除的结点，记录此结点至变量 node 中，遍历结束仍未找到对应 key 的结点，则返回 null。

如果找到要删除的结点 node，则判断是否需要比较 value 也是否一致，如果 value 值一致或者不需要比较 value 值，则执行删除结点操作，删除操作根据不同的情况与结构进行不同的处理。

如果当前结点是树结点，则证明当前位置的链表已变成红黑树结构，通过红黑树结点的方式删除对应结点。

如果不是红黑树，则证明是单链表。如果要删除的是头结点，则当前存储位置 table[i]的头结点指向删除结点的下一个结点。

如果要删除的结点不是头结点，则将要删除的结点的后继结点 node.next 赋值给要删除结点的前驱结点的 next 域，即 p.next = node.next;。

1. HashMap 当前存储键值对的数量 -1，并返回删除结点。

示例代码：

```
    /**
     * 从此映射中移除指定键的映射关系（如果存在）。
     *
     * @param  key 其映射关系要从映射中移除的键
     * @return 与 key 关联的旧值；如果 key 没有任何映射关系，则返回 null。
     *         （返回 null 还可能表示该映射之前将 null 与 key 关联。）
     */
    public V remove(Object key) {
        Node<K,V> e;
        // 1.先调用 hash(key)方法计算出 key 的 hash 值
        // 2.随后调用 removeNode 方法删除对应 key 所映射的结点
        return (e = removeNode(hash(key), key, null, false, true)) == null ?
            null : e.value;
    }


    /**
     * 删除哈希表结点的方法实现
     *
     * @param hash 键的 hash 值
     * @param key 键
     * @param value 用于比较的 value 值，当 matchValue 是 true 时才有效，否则忽略
     * @param matchValue 如果是 true 只有当 value 相等时才会移除
     * @param movable 如果是 false 当执行移除操作时，不删除其他结点
     * @return 返回删除结点 node，不存在则返回 null
```

```
     */
    final Node<K,V> removeNode(int hash, Object key, Object value,
                               boolean matchValue, boolean movable) {
        // 用于记录当前的 hash 表
        Node<K,V>[] tab;
        // 用于记录当前的链表结点
        Node<K,V> p;
        // n 用于记录 hash 表的长度，index 用于记录当前操作索引 index
        int n, index;
        // 通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
        // 判断当前元素的存储位置是否有元素存在
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (p = tab[index = (n - 1) & hash]) != null) {// 元素存在的情
况
            // node 用于记录找到的结点，e 为工作结点
            Node<K,V> node = null, e;
            K k; V v;
            // 如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等
            // 并且 头结点的 key 本身 和要获取的 key 相等
            // 则证明此头结点就是要删除的结点
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                // 记录要删除的结点的引用地址至 node 中
                node = p;
            else if ((e = p.next) != null) {// 头结点不相等
                if (p instanceof TreeNode)// 如果当前结点是树结点
                    // 则证明当前位置的链表已变成红黑树结构
                    // 通过红黑树结点的方式获取对应 key 结点
                    // 记录要删除的结点的引用地址至 node 中
                    node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
                else {// 当前位置不是红黑树，则证明是单链表
                    do {
                        // 遍历单链表，逐一比较链表结点
                        // 链表结点的 key 的 hash 值 和 要获取的 key 的 hash
值相等
                        // 并且 链表结点的 key 本身 和要获取的 key 相等
                        if (e.hash == hash &&
                            ((k = e.key) == key ||
                             (key != null && key.equals(k)))) {
                            // 找到则记录要删除的结点的引用地址至 node 中，
中断遍历
                            node = e;
                            break;
                        }
```

```
                    p = e;
                } while ((e = e.next) != null);
            }
        }
        // 如果找到要删除的结点，则判断是否需要比较 value 也是否一致
        if (node != null && (!matchValue || (v = node.value) == value
||
                             (value != null && value.equals(v)))) {
            // value 值一致或者不需要比较 value 值，则执行删除结点操作
            if (node instanceof TreeNode) // 如果当前结点是树结点
                // 则证明当前位置的链表已变成红黑树结构
                // 通过红黑树结点的方式删除对应结点
                ((TreeNode<K,V>)node).removeTreeNode(this, tab,
movable);
            else if (node == p) // node 和 p 相等，则证明删除的是头结
点
                // 当前存储位置的头结点指向删除结点的下一个结点
                tab[index] = node.next;
            else // 删除的不是头结点
                // p 是删除结点 node 的前驱结点，p 的 next 改为记录要删
除结点 node 的后继结点
                p.next = node.next;
            ++modCount;
            // 当前存储键值对的数量 - 1
            --size;
            afterNodeRemoval(node);
            // 返回删除结点
            return node;
        }
    }
    // 不存在要删除的结点，则返回 null
    return null;
}
```

- 调用 replace(K key, V value)操作根据键 key 查找对应的 key-value 键值对，随后替换对应的值 value，进行了如下操作：

  先调用 hash(key)方法计算出 key 的 hash 值

  随后调用 getNode 方法获取对应 key 所映射的 value 值 。

  记录元素旧值，将新值赋值给元素，返回元素旧值，如果没有找到元素，则返回 null。

示例代码：

```
    /**
     * 替换指定 key 所映射的 value 值
     *
     * @param key 对应要替换 value 值元素的 key 键
     * @param value 要替换对应元素的新 value 值
     * @return 返回原本的旧值，如果没有找到 key 对应的元素，则返回 null
     * @since 1.8 JDK1.8 新增方法
     */
    public V replace(K key, V value) {
        Node<K,V> e;
        // 1.先调用 hash(key)方法计算出 key 的 hash 值
        // 2.随后调用 getNode 方法获取对应 key 所映射的 value 值
        if ((e = getNode(hash(key), key)) != null) {// 如果找到对应的元
素
            // 元素旧值
            V oldValue = e.value;
            // 将新值赋值给元素
            e.value = value;
            afterNodeAccess(e);
            // 返回元素旧值
            return oldValue;
        }
        // 没有找到元素，则返回 null
        return null;
    }
```

（3.）问 1：您上面说，存放一个元素时，先计算它的 hash 值确定它的存储位置，然后再把这个元素放到对应的位置上，那万一这个位置上面已经有元素存在呢，新增的这个元素怎么办？

问 2：hash 冲突（或者叫 hash 碰撞）是什么？为什么会出现这种现象，如何解决 hash 冲突？

答：hash 冲突： 当我们调用 put(K key, V value)操作添加 key-value 键值对，这个 key-value 键值对存放在的位置是通过扰动函数(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)计算键 key 的 hash 值。随后将 这个 hash 值 % 模上 哈希表 Node<K,V>[] table 的长度 得到具体的存放位置。所以 put(K key, V value)多个元素，是有可能计算出相同的存放位置。此现象就是 hash 冲突或者叫 hash 碰撞。

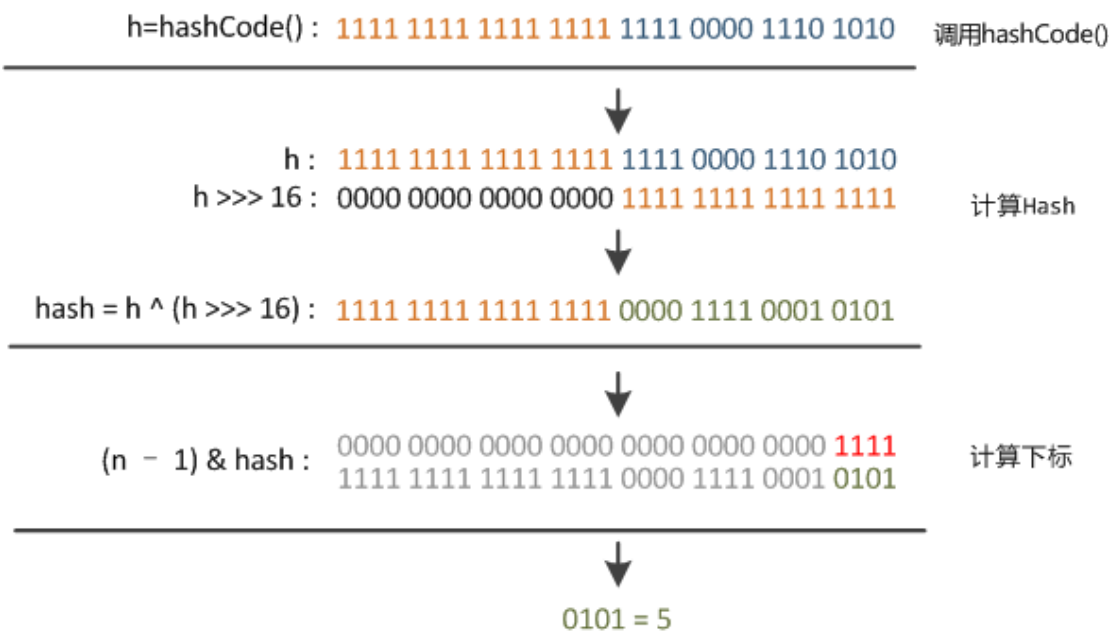
例子如下：
元素 A 的 hash 值 为 9，元素 B 的 hash 值 为 17。哈希表 Node<K,V>[] table 的长度为 8。则元素 A 的存放位置为 9 % 8 = 1，元素 B 的存放位置为 17 % 8 = 1。两个元素的存放位置均为 table[1]，发生了 hash 冲突。

hash 冲突的避免：既然会发生 hash 冲突，我们就应该想办法避免此现象的发生，解决这个问题最关键就是如果生成元素的 hash 值。Java 是使用“扰动函数”生成元素的 hash 值。

示例代码：

```
/**
 * JDK 7 的 hash 方法
 */
final int hash(int h) {

    h ^= k.hashCode();

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

/**
 * JDK 8 的 hash 方法
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Java7 做了 4 次 16 位右位移异或混合，Java 8 中这步已经简化了，只做一次 16 位右位移异或混合，而不是四次，但原理是不变的。例子如下：



扰动函数执行例子 - 图片来自于《知乎》

右位移 16 位，正好是 32bit 的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

上述扰动函数的解释参考自：JDK 源码中 HashMap 的 hash 方法原理是什么？

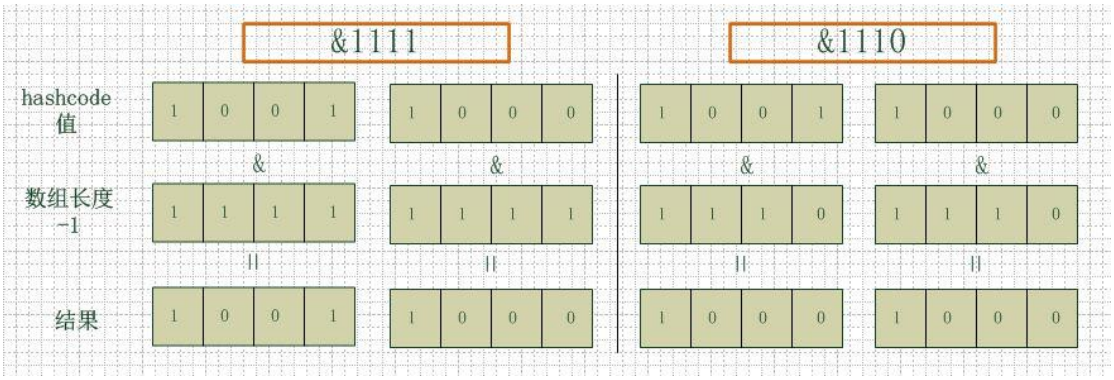
- hash 冲突解决：解决 hash 冲突的方法有很多，常见的有：开发定址法，再散列法，链地址法，公共溢出区法（详细说明请查看我的文章 JAVA 基础-自问自答学 hashCode 和 equals）。HashMap 是使用链地址法解决 hash 冲突的，当有冲突元素放进来时，会将此元素插入至此位置链表的最后一位，形成单链表。但是由于是单链表的缘故，每当通过 hash % length 找到该位置的元素时，均需要从头遍历链表，通过逐一比较 hash 值，找到对应元素。如果此位置元素过多，造成链表过长，遍历时间会大大增加，最坏情况下的时间复杂度为 O(N)，造成查找效率过低。所以当存在位置的链表长度 大于等于 8 时，HashMap 会将链表 转变为 红黑树，红黑树最坏情况下的时间复杂度为 O(logn)。以此提高查找效率。

4.问：HashMap 的容量为什么一定要是 2 的 n 次方？

答：因为调用 put(K key, V value)操作添加 key-value 键值对时，具体确定此元素的位置是通过 hash 值 % 模上 哈希表 Node<K,V>[] table 的长度 hash % length 计算的。但是"模"运算的消耗相对较大，通过位运算 h & (length-1)也可以得到取模后的存放位置，而位运算的运行效率高，但只有 length 的长度是 2 的 n 次方时，h & (length-1) 才等价于 h % length。

而且当数组长度为 2 的 n 次幂的时候，不同的 key 算出的 index 相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

例子：



hash & (length-1)运算过程.jpg

上图中，左边两组的数组长度是 16（2 的 4 次方），右边两组的数组长度是 15。两组的 hash 值均为 8 和 9。

当数组长度是 15 时，当它们和 1110 进行&与运算（相同为 1，不同为 0）时，计算的结果都是 1000，所以他们都会存放在相同的位置 table[8]中，这样就发生了 hash 冲突，那么查询时就要遍历链表，逐一比较 hash 值，降低了查询的效率。

同时，我们可以发现，当数组长度为 15 的时候，hash 值均会与 14（1110）进行&与运算，那么最后一位永远是 0，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率。

- 所以，HashMap 的容量是 2 的 n 次方，有利于提高计算元素存放位置时的效率，也降低了 hash 冲突的几率。因此，我们使用 HashMap 存储大量数据的时候，最好先预先指定容器的大小为 2 的 n 次方，即使我们不指定为 2 的 n 次方，HashMap 也会把容器的大小设置成最接近设置数的 2 的 n 次方，如，设置 HashMap 的大小为 7 ，则 HashMap 会将容器大小设置成最接近 7 的一个 2 的 n 次方数，此值为 8 。

上述回答参考自：深入理解 HashMap

示例代码：

```
/**
```

```
     * 返回一个比指定数 cap 大的，并且大小是 2 的 n 次方的数
     * Returns a power of two size for the given target capacity.
     */
    static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY :
n + 1;
    }
```

（5.）问：HashMap 的负载因子是什么，有什么作用？

答：负载因子表示哈希表空间的使用程度（或者说是哈希表空间的利用率）。

例子如下：
底层哈希表 `Node<K,V>[] table` 的容量大小 `capacity` 为 16，负载因子 `load factor` 为 0.75，则当存储的元素个数 `size = capacity 16 * load factor 0.75` 等于 12 时，则会触发 `HashMap` 的扩容机制，调用 `resize()` 方法进行扩容。

当负载因子越大，则 `HashMap` 的装载程度就越高。也就是能容纳更多的元素，元素多了，发生 `hash` 碰撞的几率就会加大，从而链表就会拉长，此时的查询效率就会降低。

当负载因子越小，则链表中的数据量就越稀疏，此时会对空间造成浪费，但是此时查询效率高。

我们可以在创建 `HashMap` 时根据实际需要适当地调整 `load factor` 的值；如果程序比较关心空间开销、内存比较紧张，可以适当地增加负载因子；如果程序比较关心时间开销，内存比较宽裕则可以适当的减少负载因子。通常情况下，默认负载因子 (0.75) 在时间和空间成本上寻求一种折衷，程序员无需改变负载因子的值。

因此，如果我们在初始化 `HashMap` 时，就预估知道需要装载 `key-value` 键值对的容量 `size`，我们可以通过 `size / load factor` 计算出我们需要初始化的容量大小 `initialCapacity`，这样就可以避免 `HashMap` 因为存放的元素达到阈值 `threshold` 而频繁调用 `resize()` 方法进行扩容。从而保证了较好的性能。

（6.）问：您能说说 HashMap 和 HashTable 的区别吗？

答：`HashMap` 和 `HashTable` 有如下区别：

1）容器整体结构：

HashMap 的 key 和 value 都允许为 null，HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理。

Hashtable 的 key 和 value 都不允许为 null。Hashtable 遇到 null，直接返回 NullPointerException。

2） 容量设定与扩容机制：

HashMap 默认初始化容量为 16，并且容器容量一定是 2 的 n 次方，扩容时，是以原容量 2 倍 的方式 进行扩容。

Hashtable 默认初始化容量为 11，扩容时，是以原容量 2 倍 再加 1 的方式进行扩容。即 int newCapacity = (oldCapacity << 1) + 1;。

3） 散列分布方式（计算存储位置）：

HashMap 是先将 key 键的 hashCode 经过扰动函数扰动后得到 hash 值，然后再利用 hash & (length - 1)的方式代替取模，得到元素的存储位置。

Hashtable 则是除留余数法进行计算存储位置的（因为其默认容量也不是 2 的 n 次方。所以也无法用位运算替代模运算），int index = (hash & 0x7FFFFFFF) % tab.length;。

由于 HashMap 的容器容量一定是 2 的 n 次方，所以能使用 hash & (length - 1)的方式代替取模的方式计算元素的位置提高运算效率，但 Hashtable 的容器容量不一定是 2 的 n 次方，所以不能使用此运算方式代替。

4）线程安全（最重要）：

HashMap 不是线程安全，如果想线程安全，可以通过调用 synchronizedMap(Map<K,V> m)使其线程安全。但是使用时的运行效率会下降，所以建议使用 ConcurrentHashMap 容器以此达到线程安全。

Hashtable 则是线程安全的，每个操作方法前都有 synchronized 修饰使其同步，但运行效率也不高，所以还是建议使用 ConcurrentHashMap 容器以此达到线程安全。

因此，Hashtable 是一个遗留容器，如果我们不需要线程同步，则建议使用 HashMap，如果需要线程同步，则建议使用 ConcurrentHashMap。

此处不再对 Hashtable 的源码进行逐一分析了，如果想深入了解的同学，可以参考此文章
Hashtable 源码剖析

（7.）问：您说 HashMap 不是线程安全的，那如果多线程下，它是如何处理的？并且什么情况下会发生线程不安全的情况？

答：HashMap 不是线程安全的，如果多个线程同时对同一个 HashMap 更改数据的话，会导致数据不一致或者数据污染。如果出现线程不安全的操作时，HashMap 会尽可能的抛出 ConcurrentModificationException 防止数据异常，当我们在对一个 HashMap 进行遍历时，在遍历期间，我们是不能对 HashMap 进行添加，删除等更改数据的操作的，否则也会抛出 ConcurrentModificationException 异常，此为 fail-fast（快速失败）机制。从源码上分析，我们在 put，remove 等更改 HashMap 数据时，都会导致 modCount 的改变，当 expectedModCount != modCount 时，则抛出 ConcurrentModificationException。如果想要线程安全，可以考虑使用 ConcurrentHashMap。

> 而且，在多线程下操作 HashMap，由于存在扩容机制，当 HashMap 调用 resize()进行自动扩容时，可能会导致死循环的发生。

由于时间关系，我暂不带着大家一起去分析 resize()方法导致死循环发生的现象造成原因了，迟点有空我会再补充上去，请见谅，大家可以参考如下文章：

Java 8 系列之重新认识 HashMap

谈谈 HashMap 线程不安全的体现

（8.）问：我们在使用 HashMap 时，选取什么对象作为 key 键比较好，为什么？

答：可变对象：指创建后自身状态能改变的对象。换句话说，可变对象是该对象在创建后它的哈希值可能被改变。

> 我们在使用 HashMap 时，最好选择不可变对象作为 key。例如 String，Integer 等不可变类型作为 key 是非常明智的。
>
> 如果 key 对象是可变的，那么 key 的哈希值就可能改变。在 HashMap 中可变对象作为 Key 会造成数据丢失。因为我们再进行 hash & (length - 1) 取模运算计算位置查找对应元素时，位置可能已经发生改变，导致数据丢失。

详细例子说明请参考：危险！在 HashMap 中将可变对象用作 Key

总结

> HashMap 是基于 Map 接口实现的一种键-值对<key,value>的存储结构，允许 null 值，同时非有序，非同步(即线程不安全)。HashMap 的底层实现是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分）。

HashMap 定位元素位置是通过键 key 经过扰动函数扰动后得到 hash 值，然后再通过 hash & (length - 1)代替取模的方式进行元素定位的。

HashMap 是使用链地址法解决 hash 冲突的，当有冲突元素放进来时，会将此元素插入至此位置链表的最后一位，形成单链表。当存在位置的链表长度 大于等于 8 时，HashMap 会将链表 转变为 红黑树，以此提高查找效率。

HashMap 的容量是 2 的 n 次方，有利于提高计算元素存放位置时的效率，也降低了 hash 冲突的几率。因此，我们使用 HashMap 存储大量数据的时候，最好先预先指定容器的大小为 2 的 n 次方，即使我们不指定为 2 的 n 次方，HashMap 也会把容器的大小设置成最接近设置数的 2 的 n 次方，如，设置 HashMap 的大小为 7 ，则 HashMap 会将容器大小设置成最接近 7 的一个 2 的 n 次方数，此值为 8 。

HashMap 的负载因子表示哈希表空间的使用程度（或者说是哈希表空间的利用率）。当负载因子越大，则 HashMap 的装载程度就越高。也就是能容纳更多的元素，元素多了，发生 hash 碰撞的几率就会加大，从而链表就会拉长，此时的查询效率就会降低。当负载因子越小，则链表中的数据量就越稀疏，此时会对空间造成浪费，但是此时查询效率高。

HashMap 不是线程安全的，Hashtable 则是线程安全的。但 Hashtable 是一个遗留容器，如果我们不需要线程同步，则建议使用 HashMap，如果需要线程同步，则建议使用 ConcurrentHashMap。

在多线程下操作 HashMap，由于存在扩容机制，当 HashMap 调用 resize()进行自动扩容时，可能会导致死循环的发生。

我们在使用 HashMap 时，最好选择不可变对象作为 key。例如 String，Integer 等不可变类型作为 key 是非常明智的。

2. ArrayList

定义

快速了解 ArrayList 究竟是什么的一个好方法就是看 JDK 源码中对 ArrayList 类的注释，大致翻译如下：

```
/**
 * 实现了 List 的接口的可调整大小的数组。实现了所有可选列表操作，并且允许所有类型的元素，
```

* 包括 null。除了实现了 List 接口，这个类还提供了去动态改变内部用于存储集合元素的数组尺寸
* 的方法。(这个类与 Vector 类大致相同,除了 ArrayList 是非线程安全外。) size，isEmpty，
* get，set，iterator，和 listIterator 方法均为常数时间复杂度。add 方法的摊还时间复杂度为
* 常数级别，这意味着，添加 n 个元素需要的时间为 O(n)。所有其他方法的时间复杂度都是线性级别的。
* 常数因子要比 LinkedList 低。
* 每个 ArrayList 实例都有一个 capacity。capacity 是用于存储 ArrayList 的元素的内部数组的大小。
* 它通常至少和 ArrayList 的大小一样大。当元素被添加到 ArrayList 时，它的 capacity 会自动增长。
* 在向一个 ArrayList 中添加大量元素前，可以使用 ensureCapacity 方法来增加 ArrayList 的容量。
* 使用这个方法来一次性地使 ArrayList 内部数组的尺寸增长到我们需要的大小提升性能。需要注意的
* 是，这个 ArrayList 实现是未经同步的。若在多线程环境下并发访问一个 ArrayList 实例，并且至少
* 一个线程对其作了结构型修改，那么必须在外部做同步。（结构性修改指的是任何添加或删除了一个或
* 多个元素的操作，以及显式改变内部数组尺寸的操作。set 操作不是结构性修改）在外部做同步通常通
* 过在一些自然地封装了 ArrayList 的对象上做同步来实现。如果不存在这样的对象，ArrayList 应
* 使用 Collections.synchronizedList 方法来包装。最好在创建时就这么做，以防止对 ArrayList
* 无意的未同步访问。（List list = Collections.synchronizedList(new ArrayList(...));）
* ArrayList 类的 iterator()方法以及 listIterator()方法返回的迭代器是 fail-fast 的：
* 在 iterator 被创建后的任何时候，若对 list 进行了结构性修改（以任何除了通过迭代器自己的
* remove 方法或 add 方法的方式），迭代器会抛出一个 ConcurrentModificationException 异常。
* 因此，在遇到并发修改时，迭代器马上抛出异常，而不是冒着以后可能在不确定的时间发生不确定行为
* 的风险继续。需要注意的是，迭代器的 fail-fast 行为是不能得到保证的，因为通常来说在未同步并发
* 修改面前无法做任何保证。fail-fast 迭代器会尽力抛出 ConcurrentModificationException 异常。
* 因此，编写正确性依赖于这个异常的程序是不对的：fail-fast 行为应该仅仅在检测 bugs 时被使用。
* ArrayList 类是 Java 集合框架中的一员。

*/

根据源码中的注释，我们了解了 ArrayList 用来组织一系列同类型的数据对象，支持对数据对象的顺序迭代与随机访问。我们还了解了 ArrayList 所支持的操作以及各项操作的时间复杂度。接下来我们来看看这个类实现了哪些接口。

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

我们可以看到，它实现了 4 个接口：List、RandomAccess、Cloneable、Serializable。
官方文档对 List 接口的说明如下：List 是一个有序的集合类型（也被称作序列）。使用 List 接口可以精确控制每个元素被插入的位置，并且可以通过元素在列表中的索引来访问它。列表允许重复的元素，并且在允许 null 元素的情况下也允许多个 null 元素。
List 接口定义了以下方法：

```
ListIterator<E> listIterator();void add(int i, E element);E remove(int i);E get(int i);E set(int i, E element);int indexOf(Object element);
```

我们可以看到，add、get 等方法都是我们在使用 ArrayList 时经常用到的。
在 ArrayList 的源码注释中提到了，ArrayList 使用 Object 数组来存储集合元素。我们来一起看下它的源码中定义的如下几个字段：

```
/** * 默认初始 capacity. */
private static final int DEFAULT_CAPACITY = 10;/** * 供空的 ArrayList 实例使用的空的数组实例 */
private static final Object[] EMPTY_ELEMENTDATA = {};/** * 供默认大小的空的 ArrayList 实例使用的空的数组实例。
     * 我们把它和 EMPTY_ELEMENTDATA 区分开来，一边指导当地一个元素被添加时把内部数组尺寸设为
     * 多少
*/
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};/** * 存放 ArrayList 中的元素的内部数组。
     * ArrayList 的 capacity 就是这个内部数组的大小。
     * 任何 elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的空 ArrayList 在第一个元素
     * 被添加进来时，其 capacity 都会被扩大至 DEFAULT_CAPACITYhe
*/
transient Object[] elementData; // non-private to simplify nested class access/** *ArrayList 所包含的元素数 */
private int size;
```

通过以上字段，我们验证了 ArrayList 内部确实使用一个 Object 数组来存储集合元素。

那么接下来我们看一下 ArrayList 都有哪些构造器，从而了解 ArrayList 的构造过程。

ArrayList 的构造器

首先我们来看一下我们平时经常使用的 ArrayList 的无参构造器的源码：

```
/** * Constructs an empty list with an initial capacity of ten. */public
ArrayList() {
  this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;}
```

我们可以看到，无参构造器仅仅是把 ArrayList 实例的 elementData 字段赋值为 DEFAULTCAPACITY_EMPTY_ELEMENTDATA。
接下来，我们再来看一下 ArrayList 的其他构造器：

```
/** * Constructs an empty list with the specified initial capacity.
     * * @param  initialCapacity  the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial
capacity
     *         is negative
*/
public ArrayList(int initialCapacity) {
  if (initialCapacity > 0) {
    this.elementData = new Object[initialCapacity];
  } else if (initialCapacity == 0) {
    this.elementData = EMPTY_ELEMENTDATA;
  } else {
    throw new IllegalArgumentException("Illegal Capacity: "+
       initialCapacity);
  }}
/** * Constructs a list containing the elements of the specified
     * collection, in the order they are returned by the collection's *
iterator.
     * * @param c the collection whose elements are to be placed into this
list
     * @throws NullPointerException if the specified collection is null
*/
public ArrayList(Collection<? extends E> c) {
  elementData = c.toArray();
  if ((size = elementData.length) != 0) {
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
    if (elementData.getClass() != Object[].class)
      elementData = Arrays.copyOf(elementData, size, Object[].class);
  } else {
```

```
    // replace with empty array.
    this.elementData = EMPTY_ELEMENTDATA;
  }}
```

通过源码我们可以看到，第一个构造器指定了 ArrayList 的初始 capacity，然后根据这个初始 capacity 创建一个相应大小的 Object 数组。若 initialCapacity 为 0，则将 elementData 赋值为 EMPTY_ELEMENTDATA；若 initialCapacity 为负数，则抛出一个 IllegalArgumentException 异常。

第二个构造器则指定一个 Collection 对象作为参数，从而构造一个含有指定集合对象元素的 ArrayList 对象。这个构造器首先把 elementData 实例域赋值为集合对象转为的数组，而后再判断传入的集合对象是否不含有任何元素，若是的话，则将 elementData 赋值为 EMPTY_ELEMENTDATA；若传入的集合对象至少包含一个元素，则进一步判断 c.toArray 方法是否正确返回了 Object 数组，若不是的话，则需要用 Arrays.copyOf 方法把 elementData 的元素类型改变为 Object。

现在，我们又了解了 ArrayList 实例的构建过程，那么接下来我们来通过 ArrayList 的 get、set 等方法的源码来进一步了解它的实现原理。

add 方法源码分析

```
/** * Appends the specified element to the end of this list.
     * * @param e element to be appended to this list
     * @return <tt>true</tt> (as specified by {@link Collection#add})
*/public boolean add(E e) {
  ensureCapacityInternal(size + 1);  // Increments modCount!!
  elementData[size++] = e;
  return true;}
```

我们可以看到，在 add 方法内部，首先调用了 ensureCapacityInternal(size+1)，这句的作用有两个：

- 保证当前 ArrayList 实例的 capacity 足够大；
- 增加 modCount，modCount 的作用是判断在迭代时是否对 ArrayList 进行了结构性修改。

然后通过将内部数组下一个索引处的元素设置为给定参数来完成了向 ArrayList 中添加元素，返回 true 表示添加成功。

get 方法源码分析

```
/** * Returns the element at the specified position in this list.
     * * @param  index index of the element to return
```

```
     * @return the element at the specified position in this list
     * @throws IndexOutOfBoundsException {@inheritDoc}
*/public E get(int index) {
  rangeCheck(index);
  return elementData(index);}
```

首先调用了 rangeCheck 方法来检查我们传入的 index 是否在合法范围内，然后调用了 elementData 方法，这个方法的源码如下：

```
E elementData(int index) {
  return (E) elementData[index];}
```

set 方法源码分析

```
/** * Replaces the element at the specified position in this list with
     * the specified element.
     * * @param index index of the element to replace
     * @param element element to be stored at the specified position
    * @return the element previously at the specified position
    * @throws IndexOutOfBoundsException {@inheritDoc}
*/
public E set(int index, E element) {
  rangeCheck(index);
  E oldValue = elementData(index);
  elementData[index] = element;
  return oldValue;}
```

我们可以看到，set 方法的实现也很简单，首先检查给定的索引是否在合法范围内，若在，则先把该索引处原来的元素存储在 oldValue 中，然后把新元素放到该索引处并返回 oldValue 即可。

3. LinkedList

定义

LinkedList 类源码中的注释如下：

/** * 实现了 List 接口的双向链表。实现了所有可选列表操作，并且可以存储所有类型的元素，包括 null。
 * 对 LinkedList 指定索引处的访问需要顺序遍历整个链表，直到到达指定元素。

* 注意 LinkedList 是非同步的。若多线程并发访问 LinkedList 对象，并且至少一个线程对其做
* 结构性修改，则必须在外部对它进行同步。这通常通过在一些自然封装了 LinkedList 的对象上
* 同步来实现。若不存在这样的对象，这个 list 应使用 Collections.synchronizedList 来包装。
* 这最好在创建时完成，以避免意外的非同步访问。
* LinkedList 类的 iterator()方法以及 listIterator()方法返回的迭代器是 fail-fast 的：
* 在 iterator 被创建后的任何时候，若对 list 进行了结构性修改（以任何除了通过迭代器自己的
* remove 方法或 add 方法的方式），迭代器会抛出一个 ConcurrentModificationException 异常。
* 因此，在遇到并发修改时，迭代器马上抛出异常，而不是冒着以后可能在不确定的时间发生不确定行为
* 的风险继续。需要注意的是，迭代器的 fail-fast 行为是不能得到保证的，因为通常来说在未同步并发
* 修改面前无法做任何保证。fail-fast 迭代器会尽力抛出 ConcurrentModificationException 异常。
* 因此，编写正确性依赖于这个异常的程序是不对的：fail-fast 行为应该仅仅在检测 bugs 时被使用。
* LinkedList 类是 Java 集合框架中的一员。
*/

LinkedList 是对链表这种数据结构的实现（对链表还不太熟悉的小伙伴可以参考深入理解数据结构之链表），当我们需要一种支持高效删除/添加元素的数据结构时，可以考虑使用链表。
总的来说，链表具有以下两个优点：

- 插入及删除操作的时间复杂度为 O(1)
- 可以动态改变大小

链表主要的缺点是：由于其链式存储的特性，链表不具备良好的空间局部性，也就是说，链表是一种缓存不友好的数据结构。

支持的操作

LinkedList 主要支持以下操作：

void addFirst(E element);void addLast(E element);E getFirst();E getLast();E removeFirst();E removeLast();boolean add(E e) //把元素 e 添加到链表末尾 void add(int index, E element) //在指定索引处添加元素

以上操作除了 add(int index, E element)外，时间复杂度均为 O(1)，而 add(int index, E element)的时间复杂度为 O(N)。

Node 类

在 LinkedList 类中我们能看到以下几个字段：

```
transient int size = 0;/** * 指向头结点  */transient Node<E> first;/** * 指向尾结点 */transient Node<E> last;
```

我们看到，LinkedList 只保存了头尾节点的引用作为其实例域，接下来我们看一下 LinkedList 的内部类 Node 的源码如下：

```
private static class Node<E> {
  E item;
  Node<E> next;
  Node<E> prev;
  Node(Node<E> prev, E element, Node<E> next) {
    this.item = element;
    this.next = next;
    this.prev = prev;
  }}
```

每个 Node 对象的 next 域指向它的下一个结点，prev 域指向它的上一个结点，item 为本结点所存储的数据对象。

addFirst 源码分析

```
/** * Inserts the specified element at the beginning of this list.
      * * @param e the element to add
*/public void addFirst(E e) {
  linkFirst(e);}
```

实际干活的是 linkFirst，它的源码如下：

```
/** * Links e as first element. */private void linkFirst(E e) {
  final Node<E> f = first;
  final Node<E> newNode = new Node<>(null, e, f);
  first = newNode;
  if (f == null)
    last = newNode;
  else
    f.prev = newNode;
```

```
  size++;
  modCount++;}
```

首先把头结点引用存于变量 f 中，而后创建一个新结点，这个新结点的数据为我们传入的参数 e，prev 指针为 null，next 指针为 f。然后把头结点指针指向新创建的结点 newNode。而后判断 f 是否为 null，若为 null，说明之前链表中没有结点，所以 last 也指向 newNode；若 f 不为 null，则把 f 的 prev 指针设为 newNode。最后还需要把 size 和 modCount 都加一，modCount 的作用与在 ArrayList 中的相同。

getFirst 方法源码分析

```
/** * Returns the first element in this list.
     * * @return the first element in this list
     * @throws NoSuchElementException if this list is empty
*/public E getFirst() {
  final Node<E> f = first;
  if (f == null)
    throw new NoSuchElementException();
  return f.item;}
```

这个方法的实现很简单，主需要直接返回 first 的 item 域（当 first 不为 null 时），若 first 为 null，则抛出 NoSuchElementException 异常。

removeFirst 方法源码分析

```
/** * Removes and returns the first element from this list.
     * * @return the first element from this list
     * @throws NoSuchElementException if this list is empty
*/public E removeFirst() {
  final Node<E> f = first;
  if (f == null)
    throw new NoSuchElementException();
  return unlinkFirst(f);}
```

unlinkFirst 方法的源码如下：

```
/** * Unlinks non-null first node f. */private E unlinkFirst(Node<E> f)
{
  // assert f == first && f != null;
  final E element = f.item;
  final Node<E> next = f.next;
  f.item = null;
```

```
  f.next = null; // help GC
  first = next;
  if (next == null)
    last = null;
  else
    next.prev = null;
  size--;
  modCount++;
  return element;}
```

add(int index, E e)方法源码分析

```
/** * Inserts the specified element at the specified position in this list.
     * Shifts the element currently at that position (if any) and any
     * subsequent elements to the right (adds one to their indices).
     * * @param index index at which the specified element is to be
inserted
     * @param element element to be inserted
     * @throws IndexOutOfBoundsException {@inheritDoc}
*/
public void add(int index, E element) {
  checkPositionIndex(index);
  if (index == size)
    linkLast(element);
  else
    linkBefore(element, node(index));}
```

这个方法中，首先调用 checkPositionIndex 方法检查给定 index 是否在合法范围内。然后若 index 等于 size，这说明要在链表尾插入元素，直接调用 linkLast 方法，这个方法的实现与之前介绍的 linkFirst 类似；若 index 小于 size，则调用 linkBefore 方法，在 index 处的 Node 前插入一个新 Node（node(index)会返回 index 处的 Node）。linkBefore 方法的源码如下：

```
/** * Inserts element e before non-null Node succ. */void linkBefore(E
e, Node<E> succ) {
  // assert succ != null;
  final Node<E> pred = succ.prev;
  final Node<E> newNode = new Node<>(pred, e, succ);
  succ.prev = newNode;
  if (pred == null)
    first = newNode;
  else
    pred.next = newNode;
  size++;
```

```
  modCount++;}
```

我们可以看到，在知道要在哪个结点前插入一个新结点时，插入操作是很容易的，时间复杂度也只有 O(1)。下面我们来看一下 node 方法是如何获取指定索引处的 Node 的：

```
/** * Returns the (non-null) Node at the specified element index. */
Node<E> node(int index) {
  // assert isElementIndex(index);
  if (index < (size >> 1)) {
    Node<E> x = first;
    for (int i = 0; i < index; i++)
      x = x.next;
    return x;
  } else {
    Node<E> x = last;
    for (int i = size - 1; i > index; i--)
      x = x.prev;
    return x;
  }}
```

首先判断 index 位于链表的前半部分还是后半部分，若是前半部分，则从头结点开始遍历，否则从尾结点开始遍历，这样可以提升效率。我们可以看到，这个方法的时间复杂度为 O(N)。

HashSet 是 Set 的一种实现方式，底层主要使用 HashMap 来确保元素不重复。

4. Hashset 源码分析

属性

```
    // 内部使用 HashMap

    private transient HashMap<E,Object> map;

    // 虚拟对象，用来作为 value 放到 map 中

    private static final Object PRESENT = new Object();
```

构造方法

```
public HashSet() {
    map = new HashMap<>();
}

public HashSet(Collection<? extends E> c) {
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
    addAll(c);
}
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}
public HashSet(int initialCapacity) {
    map = new HashMap<>(initialCapacity);
}
// 非 public，主要是给 LinkedHashSet 使用的
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

构造方法都是调用 HashMap 对应的构造方法。

最后一个构造方法有点特殊，它不是 public 的，意味着它只能被同一个包或者子类调用，这是 LinkedHashSet 专属的方法。

添加元素

直接调用 HashMap 的 put()方法，把元素本身作为 key，把 PRESENT 作为 value，也就是这个 map 中所有的 value 都是一样的。

```
public boolean add(E e) {

    return map.put(e, PRESENT)==null;

}
```

删除元素

直接调用 HashMap 的 remove()方法，注意 map 的 remove 返回是删除元素的 value，而 Set 的 remov 返回的是 boolean 类型。

这里要检查一下，如果是 null 的话说明没有该元素，如果不是 null 肯定等于 PRESENT。

```
public boolean remove(Object o) {

    return map.remove(o)==PRESENT;

}
```

查询元素

Set 没有 get()方法哦，因为 get 似乎没有意义，不像 List 那样可以按 index 获取元素。

这里只要一个检查元素是否存在的方法 contains()，直接调用 map 的 containsKey()方法。

```
public boolean contains(Object o) {

    return map.containsKey(o);

}
```

遍历元素

直接调用 map 的 keySet 的迭代器。

```
public Iterator<E> iterator() {
    return map.keySet().iterator();
}
```

全部源码

```
package java.util;
import java.io.InvalidObjectException;
import sun.misc.SharedSecrets;
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;
    // 内部元素存储在 HashMap 中
    private transient HashMap<E,Object> map;
    // 虚拟元素，用来存到 map 元素的 value 中的，没有实际意义
    private static final Object PRESENT = new Object();
    // 空构造方法
    public HashSet() {
        map = new HashMap<>();
    }
```

```
    // 把另一个集合的元素全都添加到当前 Set 中
    // 注意，这里初始化 map 的时候是计算了它的初始容量的
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }
    // 指定初始容量和装载因子
    public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }
    // 只指定初始容量
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }
    // LinkedHashSet 专用的方法
    // dummy 是没有实际意义的，只是为了跟上上面那个操持方法签名不同而已
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
    // 迭代器
    public Iterator<E> iterator() {
        return map.keySet().iterator();
```

```
    }
    // 元素个数
    public int size() {
        return map.size();
    }
    // 检查是否为空
    public boolean isEmpty() {
        return map.isEmpty();
    }
    // 检查是否包含某个元素
    public boolean contains(Object o) {
        return map.containsKey(o);
    }
    // 添加元素
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
    // 删除元素
    public boolean remove(Object o) {
.
.
        return map.remove(o)==PRESENT;
.
.
    }
```

```
    // 清空所有元素

    public void clear() {

        map.clear();

    }


    // 克隆方法

    @SuppressWarnings("unchecked")

    public Object clone() {

        try {

            HashSet<E> newSet = (HashSet<E>) super.clone();

            newSet.map = (HashMap<E, Object>) map.clone();

            return newSet;

        } catch (CloneNotSupportedException e) {
```

```
                throw new InternalError(e);
.
.
            }
.
.
        }
.
.


.
.
        // 序列化写出方法
.
.
        private void writeObject(java.io.ObjectOutputStream s)
.
.
            throws java.io.IOException {
.
.
            // 写出非 static 非 transient 属性
.
.
            s.defaultWriteObject();
.
.


.
.
            // 写出 map 的容量和装载因子
.
.
            s.writeInt(map.capacity());
.
.
            s.writeFloat(map.loadFactor());
.
.


.
.
            // 写出元素个数
.
```

```
        s.writeInt(map.size());


        // 遍历写出所有元素

        for (E e : map.keySet())

            s.writeObject(e);

    }


    // 序列化读入方法

    private void readObject(java.io.ObjectInputStream s)

        throws java.io.IOException, ClassNotFoundException {

        // 读入非 static 非 transient 属性

        s.defaultReadObject();


        // 读入容量，并检查不能小于 0

        int capacity = s.readInt();
```

```
        if (capacity < 0) {
            throw new InvalidObjectException("Illegal capacity: " +
                                             capacity);
        }

        // 读入装载因子，并检查不能小于等于 0 或者是 NaN(Not a Number)
        // java.lang.Float.NaN = 0.0f / 0.0f;
        float loadFactor = s.readFloat();
        if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
            throw new InvalidObjectException("Illegal load factor: " +
                                             loadFactor);
        }

        // 读入元素个数并检查不能小于 0
```

```
        int size = s.readInt();

        if (size < 0) {

            throw new InvalidObjectException("Illegal size: " +

                                             size);

        }

        // 根据元素个数重新设置容量

        // 这是为了保证 map 有足够的容量容纳所有元素，防止无意义的扩容

        capacity = (int) Math.min(size * Math.min(1 / loadFactor, 4.0f),

                HashMap.MAXIMUM_CAPACITY);



        // 再次检查某些东西，不重要的代码忽视掉

        SharedSecrets.getJavaOISAccess()

                     .checkArray(s, Map.Entry[].class,
HashMap.tableSizeFor(capacity));



        // 创建 map，检查是不是 LinkedHashSet 类型
```

```
        map = (((HashSet<?>)this) instanceof LinkedHashSet ?
               new LinkedHashMap<E,Object>(capacity, loadFactor) :
               new HashMap<E,Object>(capacity, loadFactor));


        // 读入所有元素，并放入 map 中
        for (int i=0; i<size; i++) {
            @SuppressWarnings("unchecked")
                E e = (E) s.readObject();
            map.put(e, PRESENT);
        }
    }

    // 可分割的迭代器，主要用于多线程并行迭代处理时使用
    public Spliterator<E> spliterator() {
        return new HashMap.KeySpliterator<E,Object>(map, 0, -1, 0, 0);
    }
}
```

总结

（1）HashSet 内部使用 HashMap 的 key 存储元素，以此来保证元素不重复；

（2）HashSet 是无序的，因为 HashMap 的 key 是无序的；

（3）HashSet 中允许有一个 null 元素，因为 HashMap 允许 key 为 null；

（4）HashSet 是非线程安全的；

（5）HashSet 是没有 get()方法的；

5. 内存模型

内存模型产生背景

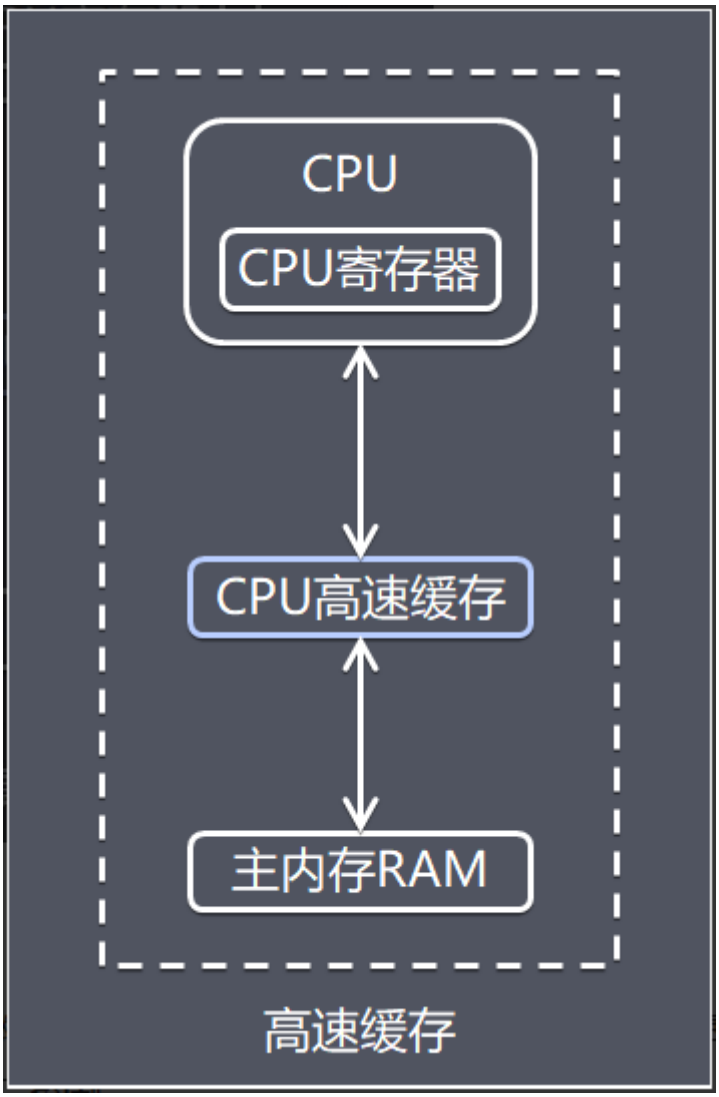
在介绍 Java 内存模型之前，我们先了解一下物理计算机中的并发问题，理解这些问题可以搞清楚内存模型产生的背景。物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机的解决方案对虚拟机的实现有相当的参考意义。

物理机的并发问题

- **硬件的效率问题**

计算机处理器处理绝大多数运行任务都不可能只靠处理器“计算”就能完成，处理器至少需要与**内存交互**，如读取运算数据、存储运算结果，这个 I/O 操作很难消除(无法仅靠寄存器完成所有运算任务)。

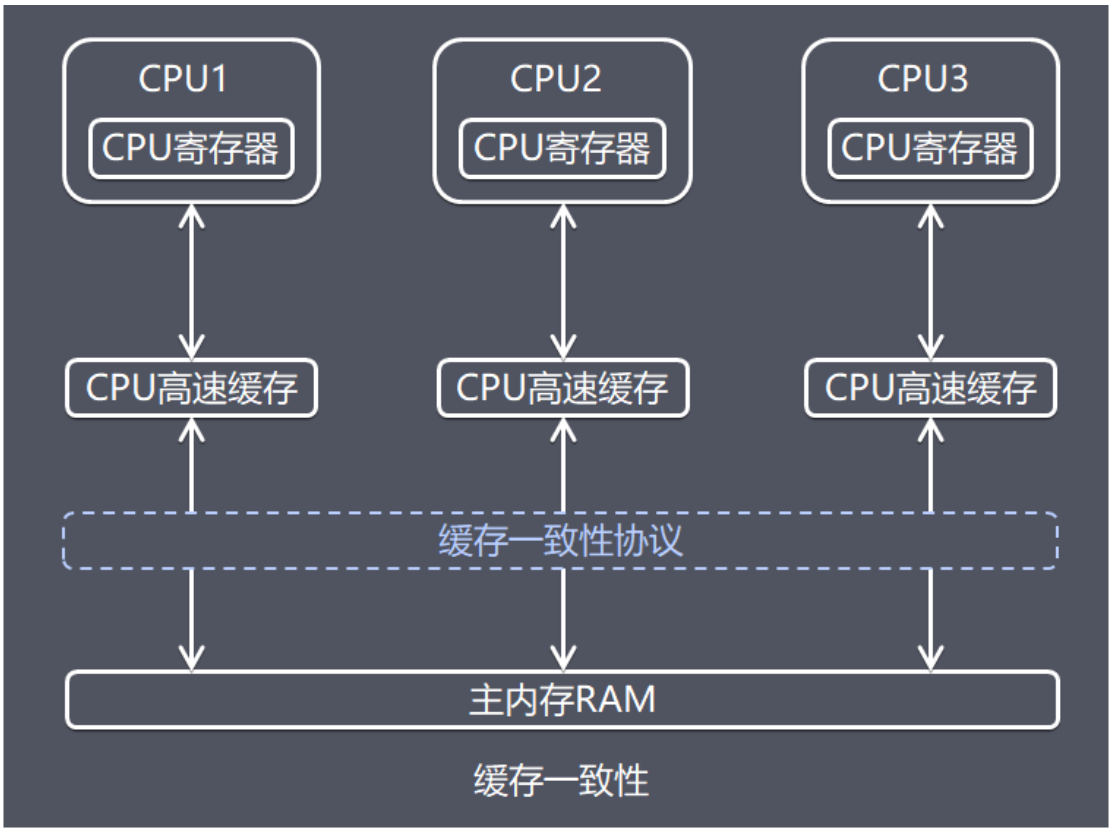
由于计算机的存储设备与处理器的运算速度有几个数量级的差距，为了避免处理器等待缓慢的内存读写操作完成，现代计算机系统通过加入一层读写速度尽可能接近处理器运算速度的高速缓存。缓存作为内存和处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速运行，当运算结束后再从缓存同步回内存之中。



- **缓存一致性问题**

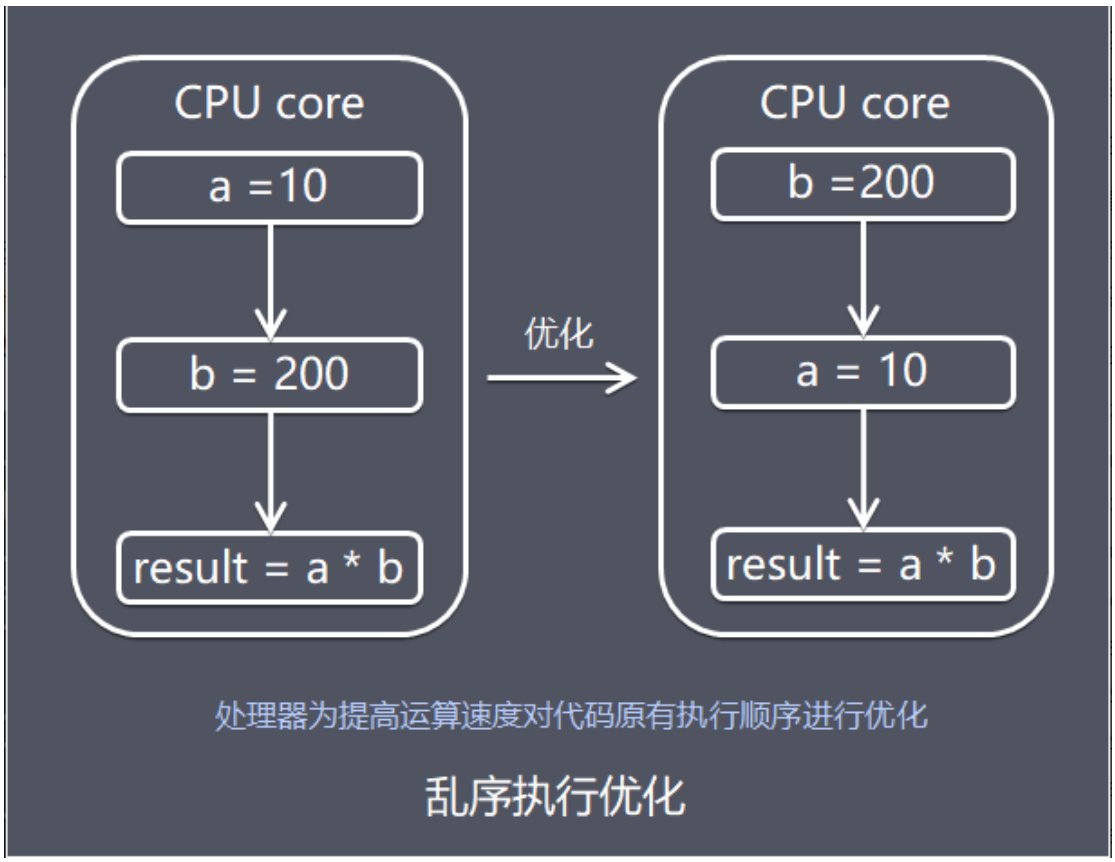
基于高速缓存的存储系统交互很好地解决了处理器与内存速度的矛盾，但是也为计算机系统带来更高的复杂度，因为引入了一个新问题：**缓存一致性。**

在多处理器的系统中(或者单处理器多核的系统)，每个处理器(每个核)都有自己的高速缓存，而它们有共享同一主内存(Main Memory)。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致。 为此，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议进行操作，来维护缓存的一致性。



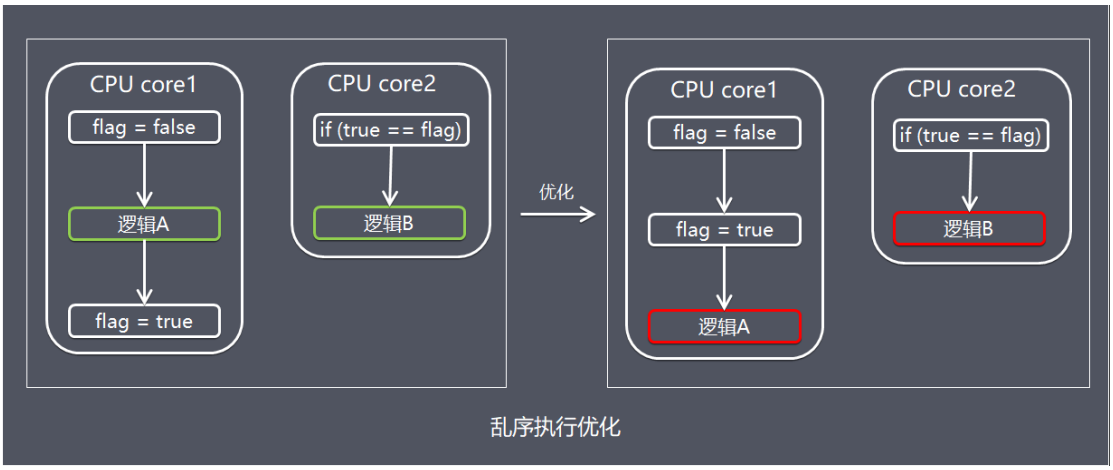
- **代码乱序执行优化问题**

为了使得处理器内部的运算单元尽量被充分利用，提高运算效率，处理器可能会对输入的代码进行乱序执行，处理器会在计算之后将乱序执行的结果重组，**乱序优化可以保证在单线程下该执行结果与顺序执行的结果是一致的**，但不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致。



乱序执行技术是处理器为提高运算速度而做出违背代码原有顺序的优化。在单核时代，处理器保证做出的优化不会导致执行结果远离预期目标，但在多核环境下却并非如此。

多核环境下，如果存在一个核的计算任务依赖另一个核 计的算任务的中间结果，而且对相关数据读写没做任何防护措施，那么其顺序性并不能靠代码的先后顺序来保证，处理器最终得出的结果和我们逻辑得到的结果可能会大不相同。



以上图为例进行说明：CPU 的 core2 中的逻辑 B 依赖 core1 中的逻辑 A 先执行

- 正常情况下，逻辑 A 执行完之后再执行逻辑 B。
- 在处理器乱序执行优化情况下，有可能导致 flag 提前被设置为 true，导致逻辑 B 先于逻辑 A 执行。

2 Java 内存模型的组成分析

内存模型概念

为了更好解决上面提到系列问题，内存模型被总结提出，我们可以把内存模型理解为**在特定操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象**。

不同架构的物理计算机可以有不一样的内存模型，Java 虚拟机也有自己的内存模型。Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model，简称 JMM）来**屏蔽掉各种硬件和操作系统的内存访问差异**，以实现**让 Java 程序在各种平台下都能达到一致的内存访问效果**，不必因为不同平台上的物理机的内存模型的差异，对各平台定制化开发程序。

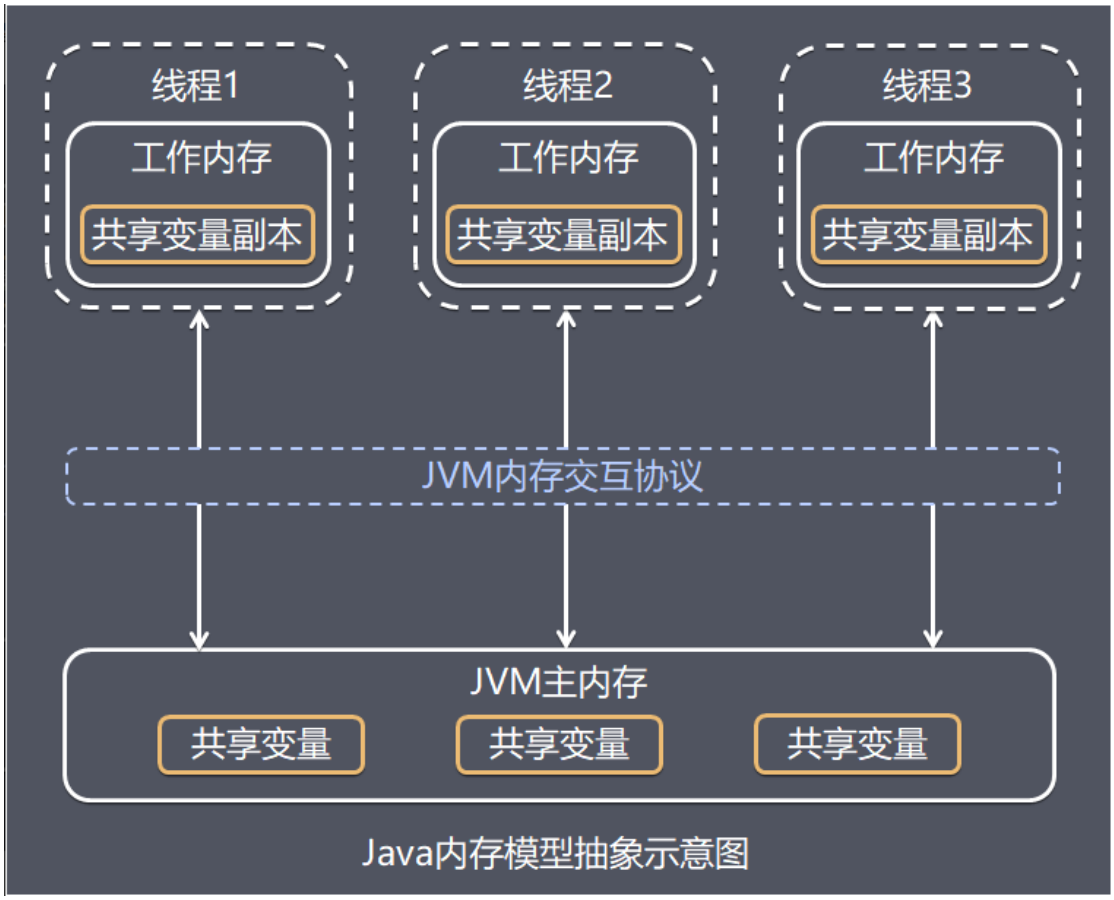
更具体一点说，Java 内存模型提出目标在于，**定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节**。此处的变量(Variables)与 Java 编程中所说的变量有所区别，它包括了实例字段、静态字段和构成数值对象的元素，但不包括局部变量与方法参数，因为后者是线程私有的。(如果局部变量是一个 reference 类型，它引用的对象在 Java 堆中可被各个线程共享，但是 reference 本身在 Java 栈的局部变量表中，它是线程私有的)。

Java 内存模型的组成

主内存 Java 内存模型规定了所有变量都存储在主内存(Main Memory)中（此处的主内存与介绍物理硬件的主内存名字一样，两者可以互相类比，但此处仅是虚拟机内存的一部分）。

工作内存 每条线程都有自己的工作内存(Working Memory，又称本地内存，可与前面介绍的处理器高速缓存类比)，线程的工作内存中保存了该线程使用到的变量的主内存中的共享变量的副本拷贝。**工作内存是 JMM 的一个抽象概念，并不真实存在**。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

Java 内存模型抽象示意图如下



JVM 内存操作的并发问题

结合前面介绍的物理机的处理器处理内存的问题，可以类比总结出 JVM 内存操作的问题，下面介绍的 Java 内存模型的执行处理将围绕解决这 2 个问题展开：

1 工作内存数据一致性 各个线程操作数据时会保存使用到的主内存中的共享变量副本，当多个线程的运算任务都涉及同一个共享变量时，将导致各自的的共享变量副本不一致，如果真的发生这种情况，数据同步回主内

存以谁的副本数据为准？ Java 内存模型主要通过一系列的数据同步协议、规则来保证数据的一致性，后面再详细介绍。

2 指令重排序优化 Java 中重排序通常是编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：**编译期重排序和运行期重排序**，分别对应编译时和运行时环境。同样的，指令重排序不是随意重排序，它需要满足以下两个条件：

- 1 在单线程环境下不能改变程序运行的结果 即时编译器（和处理器）需要保证程序能够遵守 as-if-serial 属性。通俗地说，就是在单线程情况下，要给程序一个顺序执行的假象。即经过重排序的执行结果要与顺序执行的结果保持一致。
- 2 存在数据依赖关系的不允许重排序

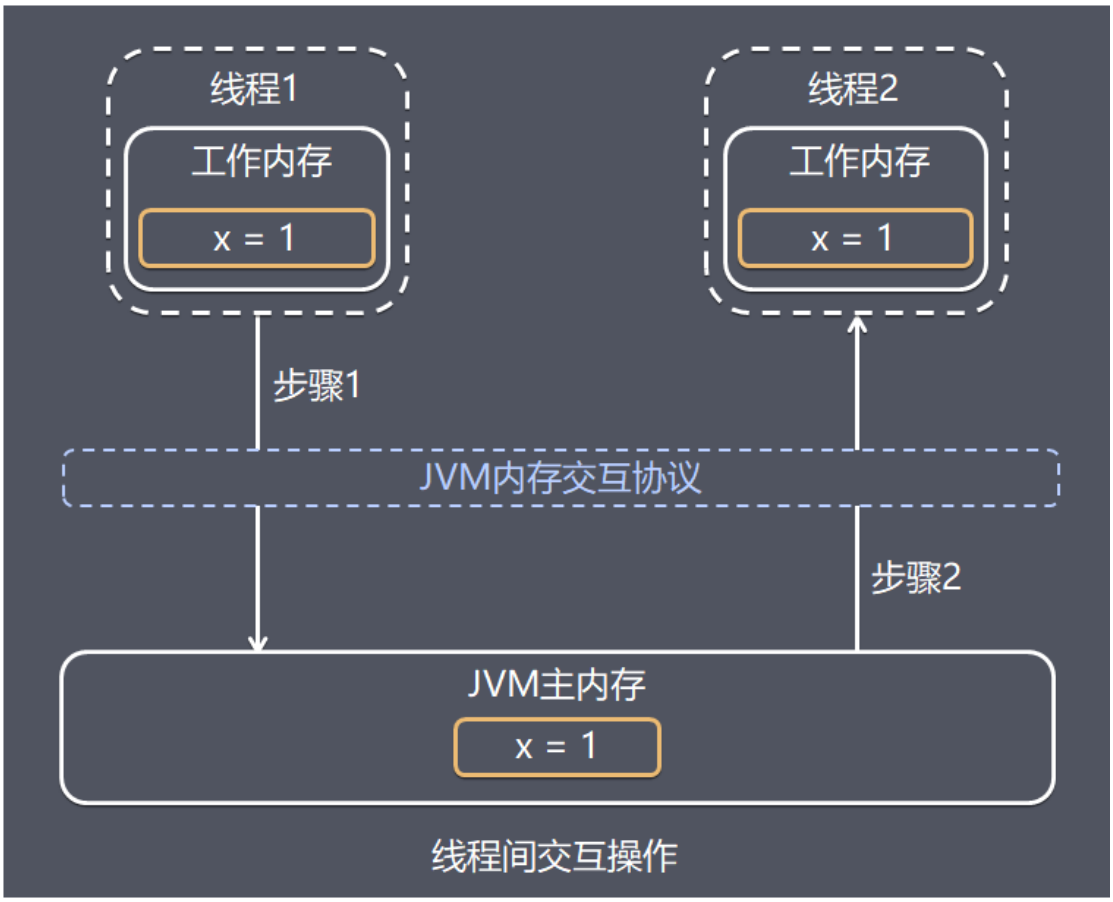
多线程环境下，如果线程处理逻辑之间存在依赖关系，有可能因为指令重排序导致运行结果与预期不同，后面再展开 Java 内存模型如何解决这种情况。

3 Java 内存间的交互操作

在理解 Java 内存模型的系列协议、特殊规则之前，我们先理解 Java 中内存间的交互操作。

交互操作流程

为了更好理解内存的交互操作，以线程通信为例，我们看看具体如何进行线程间值的同步：



线程 1 和线程 2 都有主内存中共享变量 x 的副本，初始时，这 3 个内存中 x 的值都为 0。线程 1 中更新 x 的值为 1 之后同步到线程 2 主要涉及 2 个步骤：

- 1 线程 1 把线程工作内存中更新过的 x 的值刷新到主内存中
- 2 线程 2 到主内存中读取线程 1 之前已更新过的 x 变量

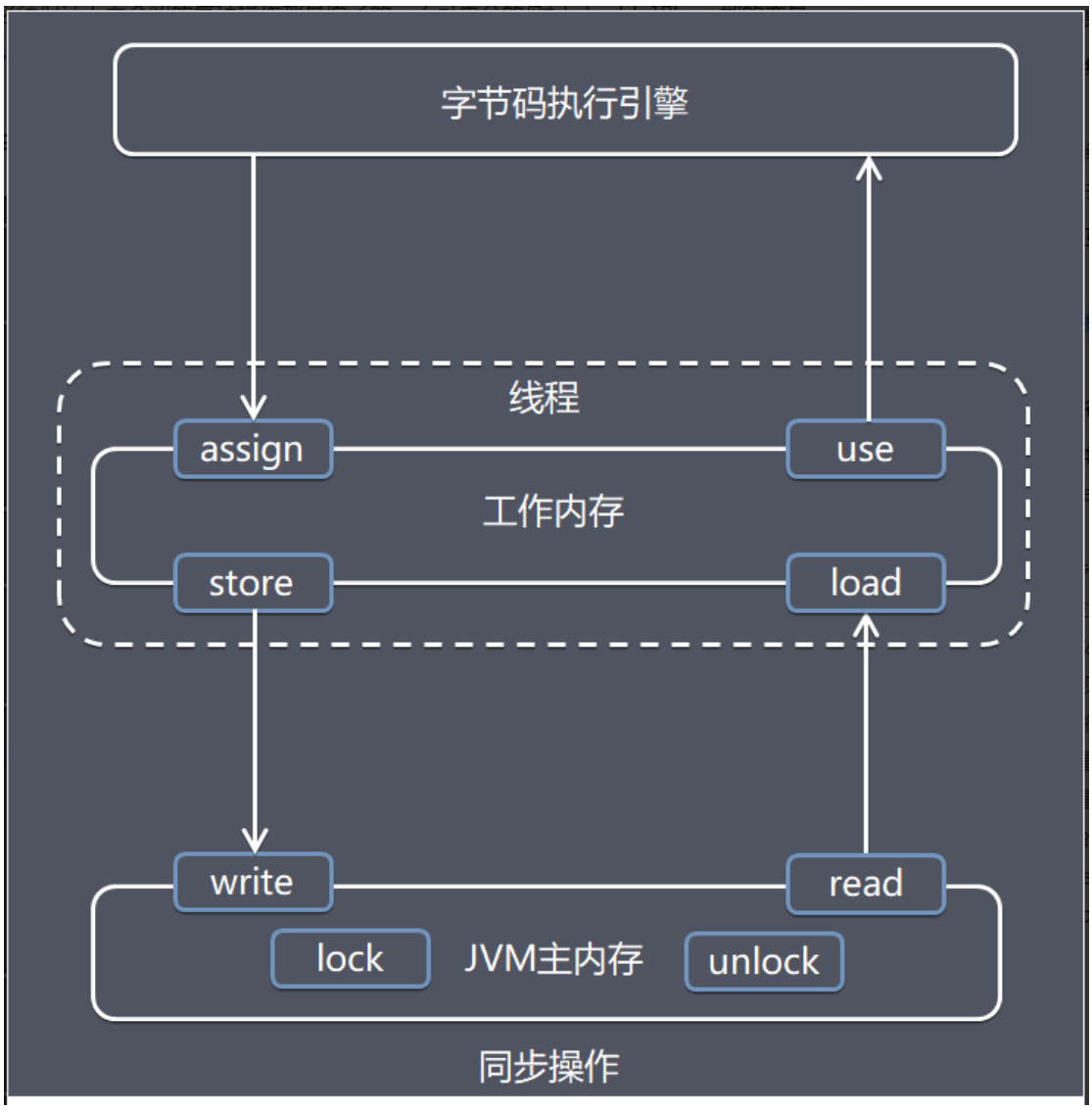
从整体上看，这 2 个步骤是线程 1 在向线程 2 发消息，这个通信过程必须经过主内存。线程对变量的所有操作（读取，赋值）都必须在**工作内存中**进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，实现各个线程提供共享变量的可见性。

内存交互的基本操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了下面介绍 8 种操作来完成。

虚拟机实现时必须保证下面介绍的每种操作都是原子的，不可再分的(对于 double 和 long 型的变量来说，load、store、read、和 write 操作在某些平台上允许有例外，后面会介绍）。

8 种基本操作



- lock (锁定) 作用于**主内存**的变量，它把一个变量标识为一条线程独占的状态。
- unlock (解锁) 作用于**主内存**的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read (读取) 作用于**主内存**的变量，它把一个变量的值从主内存**传输**到线程的工作内存中，以便随后的 load 动作使用。
- load (载入) 作用于**工作内存**的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。
- use (使用) 作用于**工作内存**的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值得字节码指令时就会执行这个操作。
- assign (赋值) 作用于**工作内存**的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store (存储) 作用于**工作内存**的变量，它把工作内存中一个变量的值传送到主内存中，以便随后 write 操作使用。

- write (写入) 作用于**主内存**的变量，它把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

4 Java 内存模型运行规则

4.1 内存交互基本操作的 3 个特性

在介绍内存的交互的具体的 8 种基本操作之前，有必要先介绍一下操作的 3 个特性，Java 内存模型是围绕着在并发过程中如何处理这 3 个特性来建立的，这里先给出定义和基本实现的简单介绍，后面会逐步展开分析。

原子性(Atomicity) **即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行**。即使在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程所干扰。

可见性(Visibility) **是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值**。 正如上面“交互操作流程”中所说明的一样，JMM 是通过在线程 1 变量工作内存修改后将新值同步回主内存，线程 2 在变量读取前从主内存刷新变量值，这种**依赖主内存作为传递媒介**的方式来实现可见性。

有序性(Ordering) 有序性规则表现在以下两种场景: 线程内和线程间

- 线程内 从某个线程的角度看方法的执行，指令会按照一种叫“串行”（as-if-serial）的方式执行，此种方式已经应用于顺序编程语言。
- 线程间 这个线程“观察”到其他线程并发地执行非同步的代码时，由于指令重排序优化，任何代码都有可能交叉执行。唯一起作用的约束是：对于同步方法，同步块(synchronized 关键字修饰)以及 volatile 字段的操作仍维持相对有序。

Java 内存模型的一系列运行规则看起来有点繁琐，但总结起来，是**围绕原子性、可见性、有序性特征建立**。归根究底，是为实现共享变量的在多个线程的工作内存的**数据一致性**，多线程并发，指令重排序优化的环境中程序能如预期运行。

4.2 happens-before 关系

介绍系列规则之前，首先了解一下 happens-before 关系：用于描述下 2 个操作的内存可见性：**如果操作 A happens-before 操作 B，那么 A 的结果对 B 可见**。happens-before 关系的分析需要分为**单线程和多线程**的情况：

单线程下的 happens-before 字节码的先后顺序天然包含 happens-before 关系：因为单线程内共享一份工作内存，不存在数据一致性的问题。 在

程序控制流路径中靠前的字节码 happens-before 靠后的字节码，即靠前的字节码执行完之后操作结果对靠后的字节码可见。然而，这并不意味着前者一定在后者之前执行。实际上，如果后者不依赖前者的运行结果，那么它们可能会被重排序。

多线程下的 happens-before 多线程由于每个线程有共享变量的副本，如果没有对共享变量做同步处理，线程 1 更新执行操作 A 共享变量的值之后，线程 2 开始执行操作 B，此时操作 A 产生的结果对操作 B 不一定可见。

为了方便程序开发，Java 内存模型实现了下述支持 happens-before 关系的操作：

- 程序次序规则 一个线程内，按照代码顺序，书写在前面的操作 happens-before 书写在后面的操作。
- 锁定规则 一个 unLock 操作 happens-before 后面对同一个锁的 lock 操作。
- volatile 变量规则 对一个变量的写操作 happens-before 后面对这个变量的读操作。
- 传递规则 如果操作 A happens-before 操作 B，而操作 B 又 happens-before 操作 C，则可以得出操作 A happens-before 操作 C。
- 线程启动规则 Thread 对象的 start()方法 happens-before 此线程的每个一个动作。
- 线程中断规则 对线程 interrupt()方法的调用 happens-before 被中断线程的代码检测到中断事件的发生。
- 线程终结规则 线程中所有的操作都 happens-before 线程的终止检测，我们可以通过 Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行。
- 对象终结规则 一个对象的初始化完成 happens-before 他的 finalize()方法的开始

4.3 内存屏障

Java 中如何保证底层操作的有序性和可见性？可以通过内存屏障。

内存屏障是被插入两个 CPU 指令之间的一种指令，用来禁止处理器指令发生重排序（像屏障一样），从而保障**有序性**的。另外，为了达到屏障的效果，它也会使处理器写入、读取值之前，将主内存的值写入高速缓存，清空无效队列，从而保障**可见性**。

举个例子：

```
Store1;
Store2;
Load1;
StoreLoad;  //内存屏障
Store3;
Load2;
Load3;
```

对于上面的一组 CPU 指令（Store 表示写入指令，Load 表示读取指令），StoreLoad 屏障之前的 Store 指令无法与 StoreLoad 屏障之后的 Load 指令进行交换位置，即**重排序**。但是 StoreLoad 屏障之前和之后的指令是可以互换位置的，即 Store1 可以和 Store2 互换，Load2 可以和 Load3 互换。

常见有 4 种屏障

- LoadLoad 屏障： 对于这样的语句 Load1; LoadLoad; Load2，在 Load2 及后续读取操作要读取的数据被访问前，保证 Load1 要读取的数据被读取完毕。
- StoreStore 屏障： 对于这样的语句 Store1; StoreStore; Store2，在 Store2 及后续写入操作执行前，保证 Store1 的写入操作对其它处理器可见。
- LoadStore 屏障： 对于这样的语句 Load1; LoadStore; Store2，在 Store2 及后续写入操作被执行前，保证 Load1 要读取的数据被读取完毕。
- StoreLoad 屏障： 对于这样的语句 Store1; StoreLoad; Load2，在 Load2 及后续所有读取操作执行前，保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的（冲刷写缓冲器，清空无效化队列）。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

Java 中对内存屏障的使用在一般的代码中不太容易见到，常见的有 volatile 和 synchronized 关键字修饰的代码块(后面再展开介绍)，还可以通过 Unsafe 这个类来使用内存屏障。

4.4 8 种操作同步的规则

JMM 在执行前面介绍 8 种基本操作时，为了保证内存间数据一致性，JMM 中规定需要满足以下规则：

- 规则 1：如果要把一个变量从主内存中复制到工作内存，就需要按顺序的执行 read 和 load 操作，如果把变量从工作内存中同步回主内存中，就要按顺序的执行 store 和 write 操作。但 Java 内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。
- 规则 2：不允许 read 和 load、store 和 write 操作之一单独出现。
- 规则 3：不允许一个线程丢弃它的最近 assign 的操作，即变量在工作内存中改变了之后必须同步到主内存中。
- 规则 4：不允许一个线程无原因的（没有发生过任何 assign 操作）把数据从工作内存同步回主内存中。
- 规则 5：一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load 或 assign ）的变量。即就是对一个变量实施 use 和 store 操作之前，必须先执行过了 load 或 assign 操作。
- 规则 6：一个变量在同一个时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。所以 lock 和 unlock 必须成对出现。
- 规则 7：如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值。

- 规则 8：如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作；也不允许去 unlock 一个被其他线程锁定的变量。
- 规则 9：对一个变量执行 unlock 操作之前，必须先把此变量同步到主内存中（执行 store 和 write 操作）

看起来这些规则有些繁琐，其实也不难理解：

- 规则 1、规则 2 工作内存中的共享变量作为主内存的副本，主内存变量的值同步到工作内存需要 read 和 load 一起使用，工作内存中的变量的值同步回主内存需要 store 和 write 一起使用，这 2 组操作各自都是是一个固定的有序搭配，不允许单独出现。
- 规则 3、规则 4 由于工作内存中的共享变量是主内存的副本，为保证数据一致性，当工作内存中的变量被字节码引擎重新赋值，必须同步回主内存。如果工作内存的变量没有被更新，不允许无原因同步回主内存。
- 规则 5 由于工作内存中的共享变量是主内存的副本，必须从主内存诞生。
- 规则 6、7、8、9 为了并发情况下安全使用变量，线程可以基于 lock 操作独占主内存中的变量，其他线程不允许使用或 unlock 该变量，直到变量被线程 unlock。

4.5 volatile 型变量的特殊规则

volatile 的中文意思是不稳定的，易变的，用 volatile 修饰变量是为了保证变量的可见性。

volatile 的语义

volatile 主要有下面 2 种语义

语义 1 保证可见性

保证了不同线程对该变量操作的内存可见性。

这里保证可见性是不等同于 volatile 变量并发操作的安全性，保证可见性具体一点解释：

线程写 volatile 变量的过程：

- 1 改变线程工作内存中 volatile 变量副本的值
- 2 将改变后的副本的值从工作内存刷新到主内存

线程读 volatile 变量的过程：

- 1 从主内存中读取 volatile 变量的最新值到线程的工作内存中
- 2 从工作内存中读取 volatile 变量的副本

但是如果多个线程同时把更新后的变量值同时刷新回主内存，可能导致得到的值不是预期结果：

举个例子： 定义 volatile int count = 0，2 个线程同时执行 count++操作，每个线程都执行 500 次，最终结果小于 1000，原因是每个线程执行 count++需要以下 3 个步骤：

- 步骤 1 线程从主内存读取最新的 count 的值
- 步骤 2 执行引擎把 count 值加 1，并赋值给线程工作内存
- 步骤 3 线程工作内存把 count 值保存到主内存 有可能某一时刻 2 个线程在步骤 1 读取到的值都是 100，执行完步骤 2 得到的值都是 101，最后刷新了 2 次 101 保存到主内存。

语义 2 禁止进行指令重排序

具体一点解释，禁止重排序的规则如下：

- 当程序执行到 volatile 变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
- 在进行指令优化时，不能将在对 volatile 变量访问的语句放在其后面执行，也不能把 volatile 变量后面的语句放到其前面执行。

普通的变量仅仅会保证该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证赋值操作的顺序与程序代码中的执行顺序一致。

举个例子：

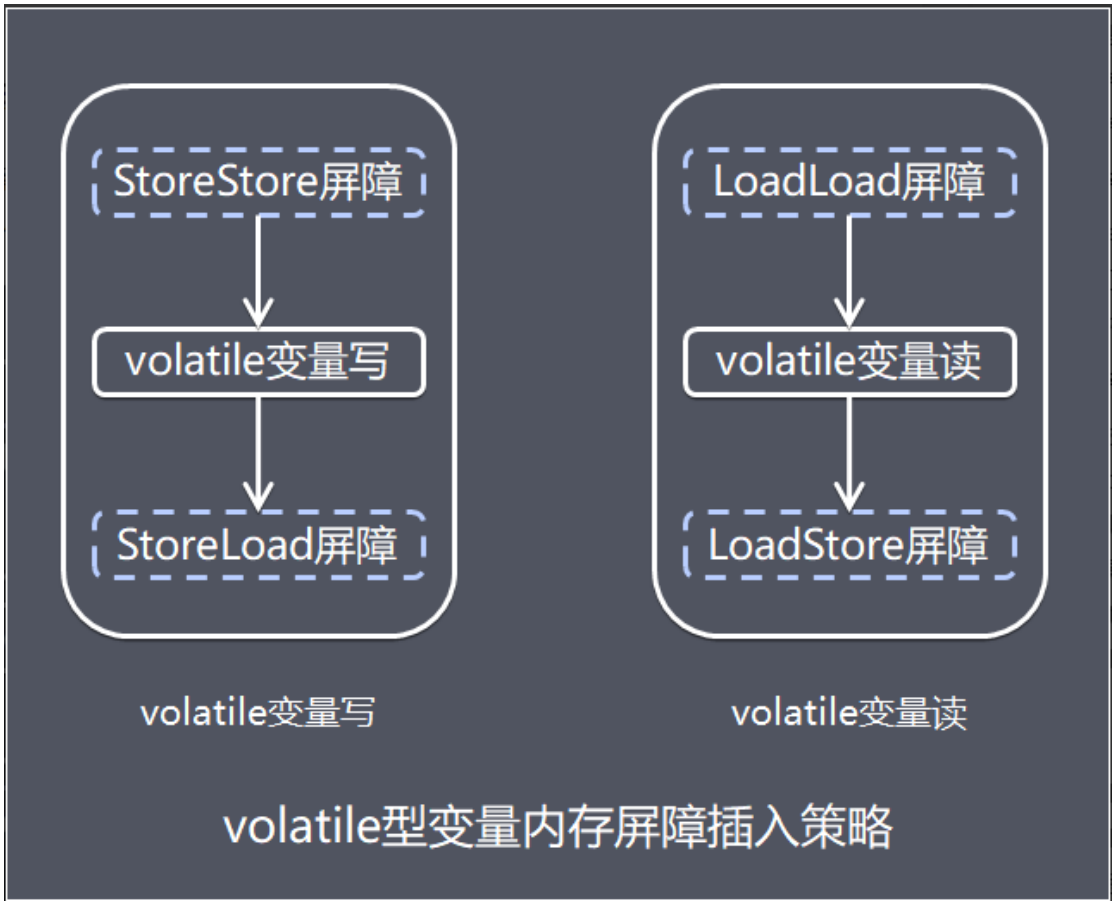
```
volatile boolean initialized = false;

// 下面代码线程 A 中执行
// 读取配置信息，当读取完成后将 initialized 设置为 true 以通知其他线程配置可用
doSomethingReadConfg();
initialized = true;

// 下面代码线程 B 中执行
// 等待 initialized 为 true，代表线程 A 已经把配置信息初始化完成 while
(!initialized) {
    sleep();
}
// 使用线程 A 初始化好的配置信息 doSomethingWithConfig();复制代码
```

上面代码中如果定义 initialized 变量时没有使用 volatile 修饰，就有可能会由于指令重排序的优化，导致线程 A 中最后一句代码 "initialized = true" 在“doSomethingReadConfg()”之前被执行，这样会导致线程 B 中使用配置信息的代

码就可能出现错误，而 volatile 关键字就禁止重排序的语义可以避免此类情况发生。

volatile 型变量实现原理

具体实现方式是在编译期生成字节码时，会在指令序列中增加内存屏障来保证，下面是基于保守策略的 JMM 内存屏障插入策略：



在每个 volatile 写操作的前面插入一个 StoreStore 屏障。 该屏障除了保证了屏障之前的写操作和该屏障之后的写操作不能重排序，还会保证了 volatile 写操作之前，任何的读写操作都会先于 volatile 被提交。

在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。 该屏障除了使 volatile 写操作不会与之后的读操作重排序外，还会刷新处理器缓存，使 volatile 变量的写更新对其他线程可见。

在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。 该屏障除了使 volatile 读操作不会与之前的写操作发生重排序外，还会刷新处理器缓存，使 volatile 变量读取的为最新值。

在每个 volatile 读操作的后面插入一个 LoadStore 屏障。 该屏障除了禁止了 volatile 读操作与其之后的任何写操作进行重排序，还会刷新处理器缓存，使其他线程 volatile 变量的写更新对 volatile 读操作的线程可见。

volatile 型变量使用场景

总结起来，就是“一次写入，到处读取”，某一线程负责更新变量，其他线程只读取变量(不更新变量)，并根据变量的新值执行相应逻辑。例如状态标志位更新，观察者模型变量值发布。

4.6 final 型变量的特殊规则

我们知道，final 成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误。 final 关键字的可见性是指：被 final 修饰的字段在声明时或者构造器中，一旦初始化完成，那么在其他线程无须同步就能正确看见 final 字段的值。这是因为一旦初始化完成，final 变量的值立刻回写到主内存。

4.7 synchronized 的特殊规则

通过 synchronized 关键字包住的代码区域，对数据的读写进行控制：

- 读数据 当线程进入到该区域读取变量信息时，对数据的读取也不能从工作内存读取，只能从内存中读取，保证读到的是最新的值。
- 写数据 在同步区内对变量的写入操作，在离开同步区时就将当前线程内的数据刷新到内存中，保证更新的数据对其他线程的可见性。

4.8 long 和 double 型变量的特殊规则

Java 内存模型要求 lock、unlock、read、load、assign、use、store、write 这 8 种操作都具有原子性，但是对于 64 位的数据类型(long 和 double)，在模型中特别定义相对宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作分为 2 次 32 位的操作来进行。也就是说虚拟机可选择不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性。由于这种非原子性，有可能导致其他线程读到同步未完成的“32 位的半个变量”的值。

不过实际开发中，Java 内存模型强烈建议虚拟机把 64 位数据的读写实现为具有原子性，目前各种平台下的商用虚拟机都选择把 64 位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要把用到的 long 和 double 变量专门声明为 volatile。

5 总结

由于 Java 内存模型涉及系列规则，网上的文章大部分就是对这些规则进行解析，但是很多没有解释为什么需要这些规则，这些规则的作用，其实这是不利于初学者学习的，容易绕进去这些繁琐规则不知所以然，下面谈谈我的一点学习知识的个人体会：

学习知识的过程不是等同于只是理解知识和记忆知识，而是要对知识解决的问题的输入和输出建立连接，知识的本质是解决问题，所以在学习之前要理解问题，理解这个问题要的输出和输出，而知识就是输入到输出的一个关系映射。知识的学习要结合大量的例子来**理解这个映射关系，然后压缩知识**，华罗庚说过：“把一本书读厚，然后再读薄”，解释的就是这个道理，先结合大量的例子理解知识，然后再压缩知识。

以学习 Java 内存模型为例：

- 理解问题，明确输入输出 首先理解 Java 内存模型是什么，有什么用，解决什么问题
- 理解内存模型系列协议 结合大量例子理解这些协议规则
- 压缩知识 大量规则其实就是通过数据同步协议，保证内存副本之间的数据一致性，同时防止重排序对程序的影响。

6. 垃圾回收算法（JVM）

JVM 类加载机制、垃圾回收算法对比、Java 虚拟机结构

当你讲到分代回收算法的时候，不免会被追问到新生对象是怎么从年轻代到老年代的，以及可以作为 root 结点的对象有哪些两个问题。

1、谈谈对 JVM 的理解?
2、JVM 内存区域，开线程影响哪块区域内存？
3、对 Dalvik、ART 虚拟机有什么了解？对比？

ART 的机制与 Dalvik 不同。

在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器（just in time ，JIT）转换为机器码，这会拖慢应用的运行效率。

而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，极大的提高了程序的运行效率，同时减少了手机的耗电量，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动(首次)和执行都会变得更加快速。

优点：
系统性能的显著提升。
应用启动更快、运行更快、体验更流畅、触感反馈更及时。

更长的电池续航能力。

支持更低的硬件。

缺点：

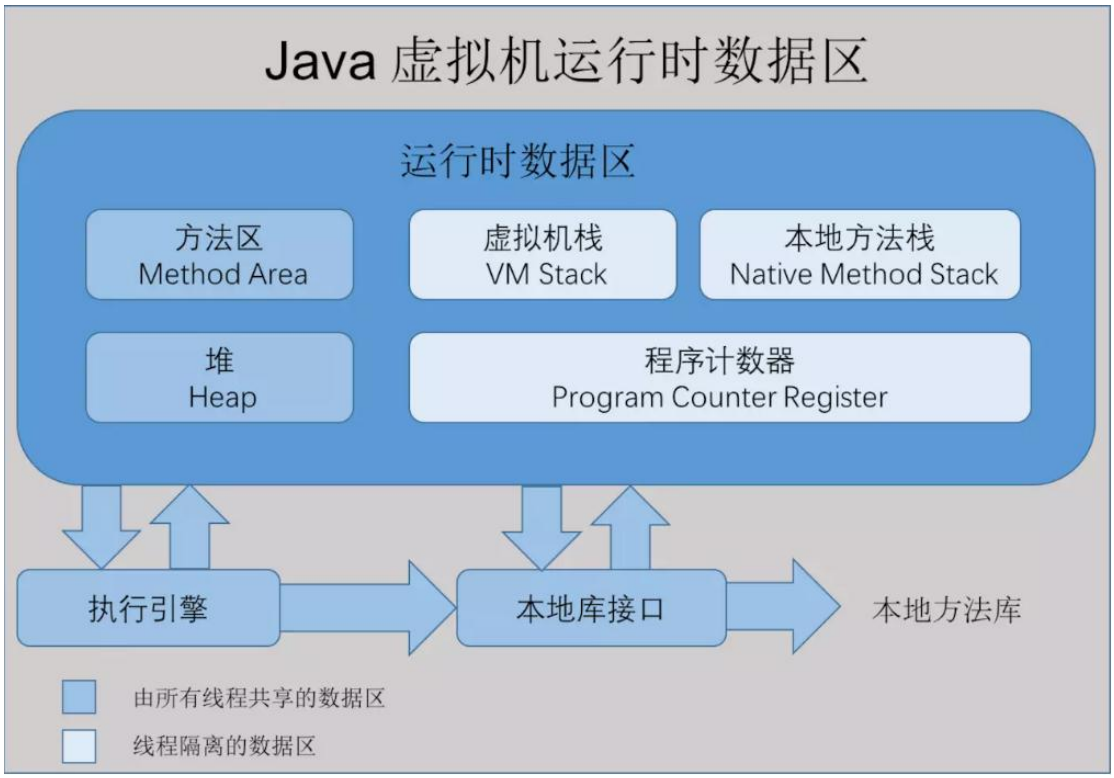
机器码占用的存储空间更大，字节码变为机器码之后，可能会增加 10%-20%（不过在应用包中，可执行的代码常常只是一部分。比如最新的 Google+ APK 是 28.3 MB，但是代码只有 6.9 MB。）

应用的安装时间会变长。

7、垃圾回收机制和调用 System.gc()的区别？

1. Java 内存区域与内存溢出异常

1.1 运行时数据区域



1.1.1 程序计数器

内存空间小，线程私有。字节码解释器工作是就是通过改变这个计数器的值来选取下一条需

要执行指令的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖计数器完成

如果线程正在执行一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器的值则为 (Undefined)。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

1.1.2 Java 虚拟机栈

线程私有，生命周期和线程一致。描述的是 Java 方法执行的内存模型：每个方法在执行时都会床创建一个栈帧(Stack Frame)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行结束，就对应着一个栈帧从虚拟机栈中入栈到出栈的过程。

局部变量表：存放了编译期可知的各种基本类型(boolean、byte、char、short、int、float、long、double)、对象引用(reference 类型)和 returnAddress 类型(指向了一条字节码指令的地址)

StackOverflowError：线程请求的栈深度大于虚拟机所允许的深度。
OutOfMemoryError：如果虚拟机栈可以动态扩展，而扩展时无法申请到足够的内存。

1.1.3 本地方法栈

区别于 Java 虚拟机栈的是，Java 虚拟机栈为虚拟机执行 Java 方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。也会有 StackOverflowError 和 OutOfMemoryError 异常。

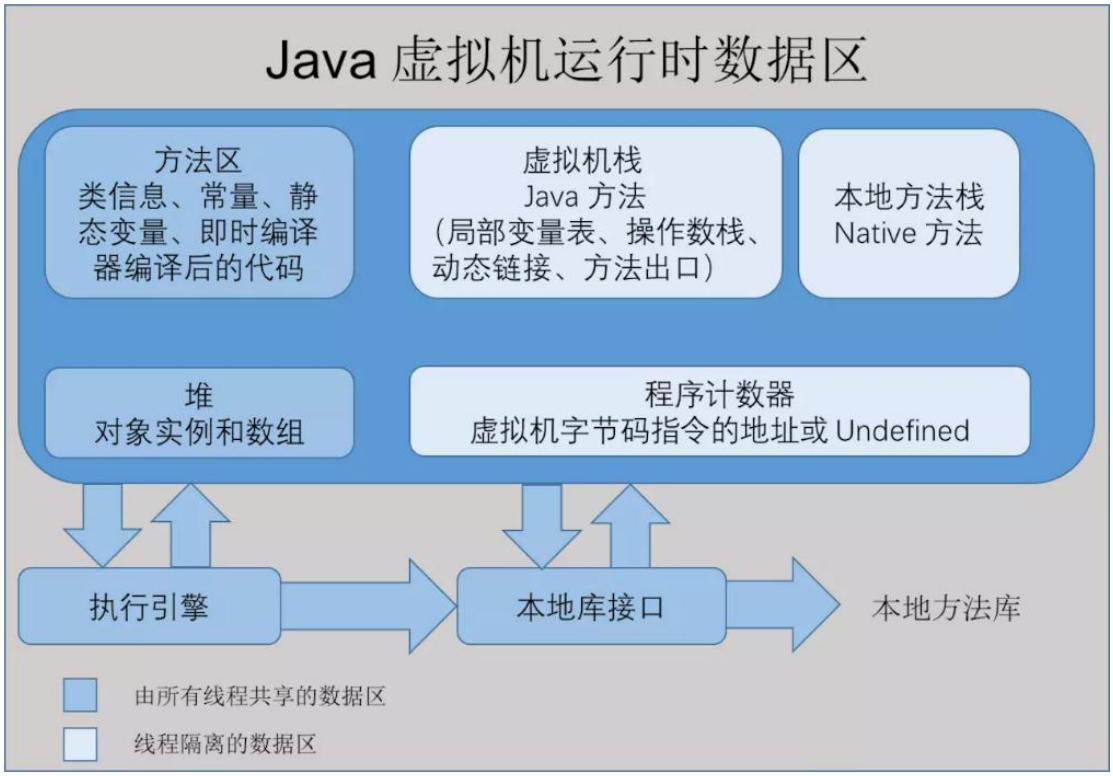
1.1.4 Java 堆

对于绝大多数应用来说，这块区域是 JVM 所管理的内存中最大的一块。线程共享，主要是存放对象实例和数组。内部会划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer, TLAB)。可以位于物理上不连续的空间，但是逻辑上要连续。

OutOfMemoryError：如果堆中没有内存完成实例分配，并且堆也无法再扩展时，抛出该异常。

1.1.5 方法区

属于共享内存区域，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。



1.1.6 运行时常量池

属于方法区一部分，用于存放编译期生成的各种字面量和符号引用。编译器和运行期(String 的 intern())都可以将常量放入池中。内存有限，无法申请时抛出 OutOfMemoryError。

1.1.7 直接内存

非虚拟机运行时数据区的部分

在 JDK 1.4 中新加入 NIO (New Input/Output) 类，引入了一种基于通道(Channel)和缓存(Buffer)的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。可以避免在 Java 堆和 Native 堆中来回的数据耗时操作。
OutOfMemoryError：会受到本机内存限制，如果内存区域总和大于物理内存限制从而导致动态扩展时出现该异常。

1.2 HotSpot 虚拟机对象探秘
主要介绍数据是如何创建、如何布局以及如何访问的。

1.2.1 对象的创建

创建过程比较复杂，建议看书了解，这里提供个人的总结。

遇到 new 指令时，首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，执行相应的

类加载。

类加载检查通过之后，为新对象分配内存(内存大小在类加载完成后便可确认)。在堆的空闲内存中划分一块区域(‘指针碰撞-内存规整’或‘空闲列表-内存交错’的分配方式)。

前面讲的每个线程在堆中都会有私有的分配缓冲区(TLAB)，这样可以很大程度避免在并发情况下频繁创建对象造成的线程不安全。

内存空间分配完成后会初始化为 0(不包括对象头)，接下来就是填充对象头，把对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息存入对象头。

执行 new 指令后执行 init 方法后才算一份真正可用的对象创建完成。

1.2.2 对象的内存布局

在 HotSpot 虚拟机中，分为 3 块区域：对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)

对象头(Header)：包含两部分，第一部分用于存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等，32 位虚拟机占 32 bit，64 位虚拟机占 64 bit。官方称为 ‘Mark Word’。第二部分是类型指针，即对象指向它的类的元数据指针，虚拟机通过这个指针确定这个对象是哪个类的实例。另外，如果是 Java 数组，对象头中还必须有一块用于记录数组长度的数据，因为普通对象可以通过 Java 对象元数据确定大小，而数组对象不可以。

实例数据(Instance Data)：程序代码中所定义的各种类型的字段内容(包含父类继承下来的和子类中定义的)。

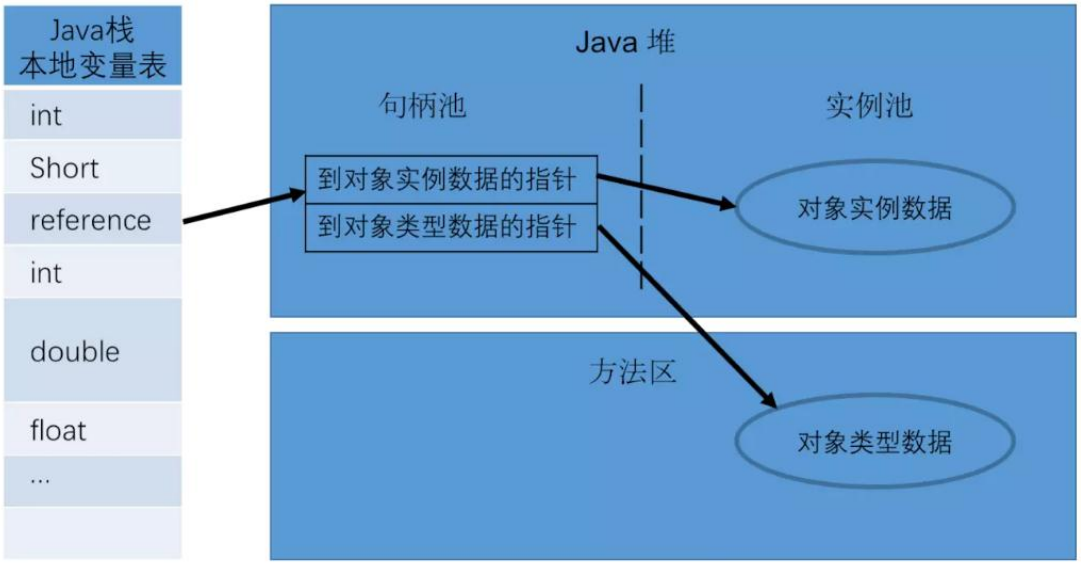
对齐填充(Padding)：不是必然需要，主要是占位，保证对象大小是某个字节的整数倍。

1.2.3 对象的访问定位

使用对象时，通过栈上的 reference 数据来操作堆上的具体对象。

通过句柄访问

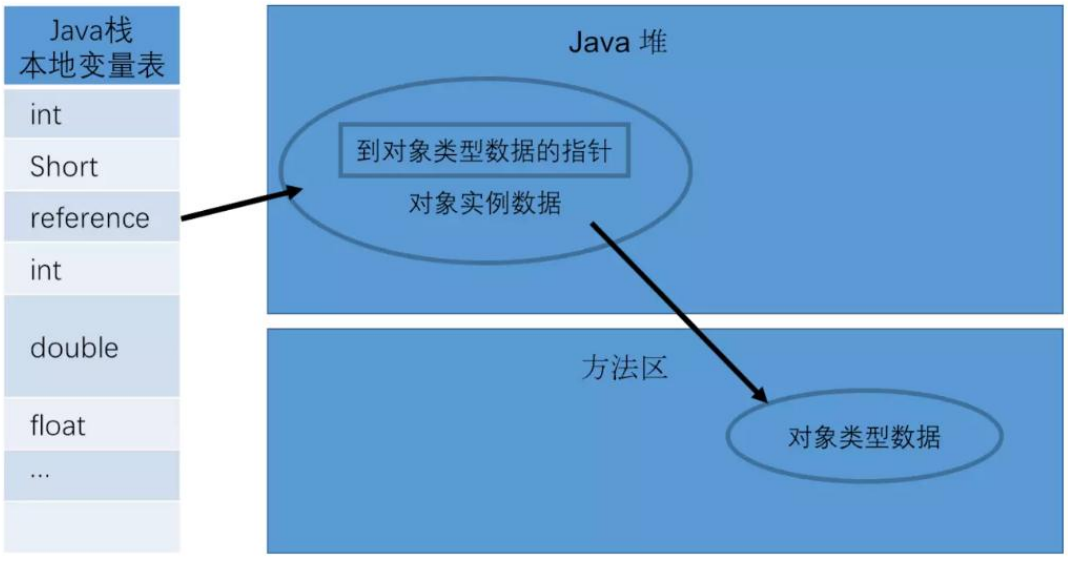
Java 堆中会分配一块内存作为句柄池。reference 存储的是句柄地址。详情见图。



通过句柄访问对象

使用直接指针访问

reference 中直接存储对象地址



通过直接指针访问对象

比较：使用句柄的最大好处是 reference 中存储的是稳定的句柄地址，在对象移动(GC)是只改变实例数据指针地址，reference 自身不需要修改。直接指针访问的最大好处是速度快，节省了一次指针定位的时间开销。如果是对象频繁 GC 那么句柄方法好，如果是对象频繁访问则直接指针访问好。

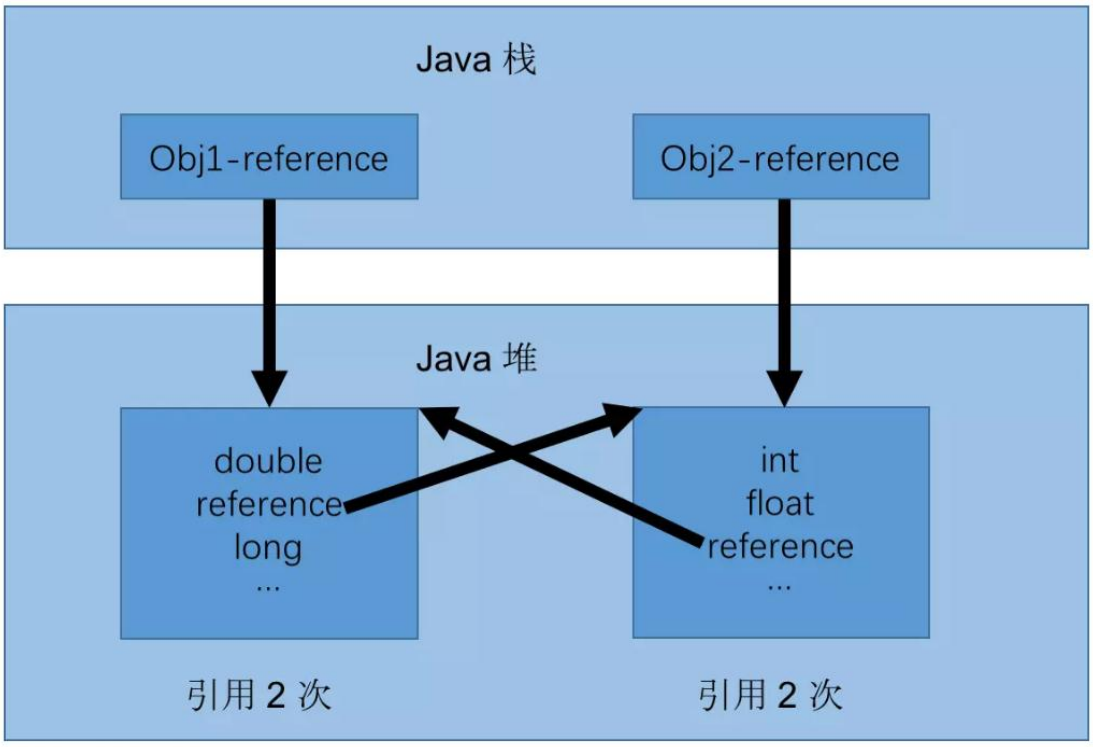
1.3 实战
// 待填

2. 垃圾回收器与内存分配策略
2.1 概述
程序计数器、虚拟机栈、本地方法栈 3 个区域随线程生灭(因为是线程私有)，栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期才知道那些对象会创建，这部分内存的分配和回收都是动态的，垃圾回收期所关注的就是这部分内存。

2.2 对象已死吗？
在进行内存回收之前要做的事情就是判断那些对象是‘死’的，哪些是‘活’的。

2.2.1 引用计数法

给对象添加一个引用计数器。但是难以解决循环引用问题。

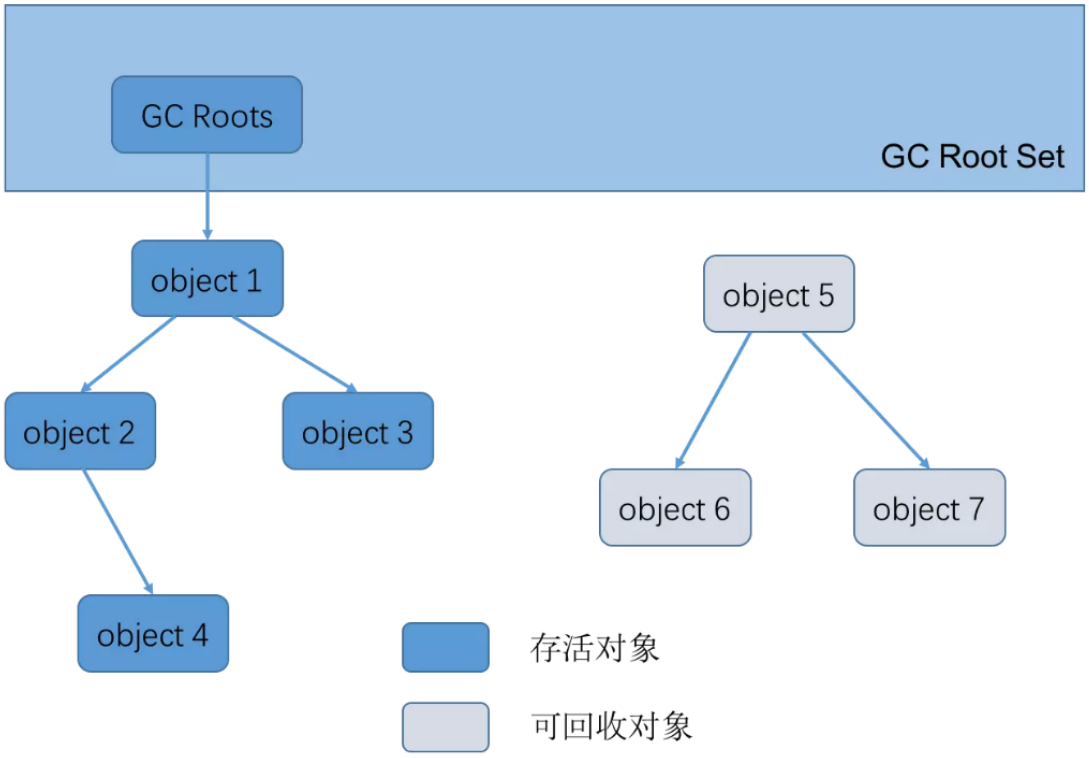


从图中可以看出，如果不下小心直接把 Obj1-reference 和 Obj2-reference 置 null。则在 Java 堆当中的两块内存依然保持着互相引用无法回收。

2.2.2 可达性分析法

通过一系列的‘GC Roots’的对象作为起始点，从这些节点出发所走过的路径称为引用链。

当一个对象到 GC Roots 没有任何引用链相连的时候说明对象不可用。



可作为 GC Roots 的对象：

虚拟机栈(栈帧中的本地变量表)中引用的对象
方法区中类静态属性引用的对象
方法区中常量引用的对象
本地方法栈中 JNI(即一般说的 Native 方法) 引用的对象

2.2.3 再谈引用

前面的两种方式判断存活时都与‘引用’有关。但是 JDK 1.2 之后，引用概念进行了扩充，下面具体介绍。

下面四种引用强度一次逐渐减弱

强引用

类似于 Object obj = new Object(); 创建的，只要强引用在就不回收。

软引用

SoftReference 类实现软引用。在系统要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。

弱引用

WeakReference 类实现弱引用。对象只能生存到下一次垃圾收集之前。在垃圾收集器工作时，无论内存是否足够都会回收掉只被弱引用关联的对象。

虚引用

PhantomReference 类实现虚引用。无法通过虚引用获取一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

2.2.4 生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是“facebook”的，这时候它们暂时出于“缓刑”阶段，一个对象的真正死亡至少要经历两次标记过程：如果对象在进行中可达性分析后发现没有与 GC Roots 相连接的引用链，那他将会被第一次标记并且进行一次筛选，筛选条件是此对象是否有必要执行 finalize() 方法。当对象没有覆盖 finalize() 方法，或者 finalize() 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象竟会放置在一个叫做 F-Queue 的队列中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的“执行”是指虚拟机会出发这个方法，并不承诺或等待他运行结束。finalize() 方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize() 中成功拯救自己 —— 只要重新与引用链上的任何一个对象简历关联即可。

finalize() 方法只会被系统自动调用一次。

2.2.5 回收方法区

在堆中，尤其是在新生代中，一次垃圾回收一般可以回收 70% ~ 95% 的空间，而永久代的垃圾收集效率远低于此。

永久代垃圾回收主要两部分内容：废弃的常量和无用的类。

判断废弃常量：一般是判断没有该常量的引用。

判断无用的类：要以下三个条件都满足

该类所有的实例都已经回收，也就是 Java 堆中不存在该类的任何实例
加载该类的 ClassLoader 已经被回收
该类对应的 java.lang.Class 对象没有任何地方呗引用，无法在任何地方通过反射访问该类的方法
2.3 垃圾回收算法
仅提供思路

2.3.1 标记 —— 清除算法

直接标记清除就可。

两个不足：

效率不高
空间会产生大量碎片

2.3.2 复制算法

把空间分成两块，每次只对其中一块进行 GC。当这块内存使用完时，就将还存活的对象复制到另一块上面。

解决前一种方法的不足，但是会造成空间利用率低下。因为大多数新生代对象都不会熬过第一次 GC。所以没必要 1 : 1 划分空间。可以分一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 空间和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 上，最后清理 Eden 和 Survivor 空间。大小比例一般是 8 : 1 : 1，每次浪费 10% 的 Survivor 空间。但是这里有一个问题就是如果存活的大于 10% 怎么办？这里采用一种分配担保策略：多出来的对象直接进入老年代。

2.3.3 标记-整理算法

不同于针对新生代的复制算法，针对老年代的特点，创建该算法。主要是把存活对象移到内存的一端。

2.3.4 分代回收

根据存活对象划分几块内存区，一般是分为新生代和老年代。然后根据各个年代的特点制定相应的回收算法。

新生代

每次垃圾回收都有大量对象死去，只有少量存活，选用复制算法比较合理。

老年代

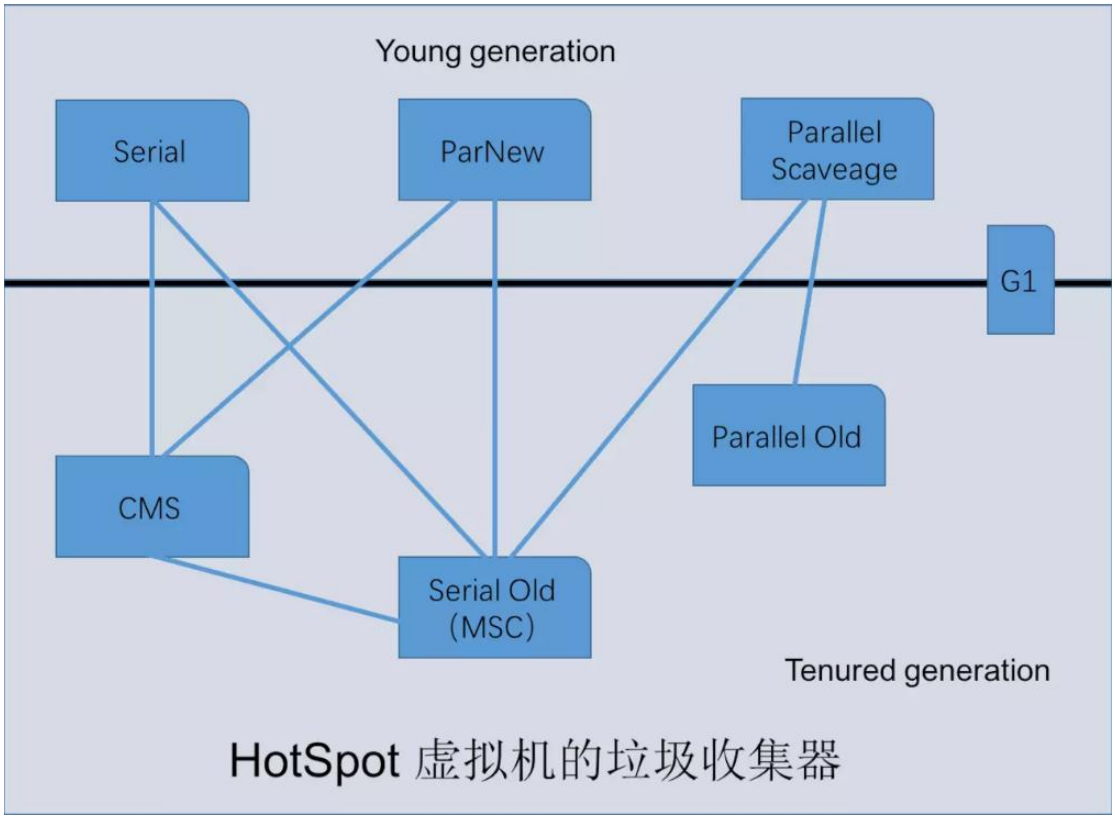
老年代中对象存活率较高、没有额外的空间分配对它进行担保。所以必须使用 标记 —— 清除 或者 标记 —— 整理 算法回收。

2.4 HotSpot 的算法实现

// 待填

2.5 垃圾回收器

收集算法是内存回收的理论，而垃圾回收器是内存回收的实践。

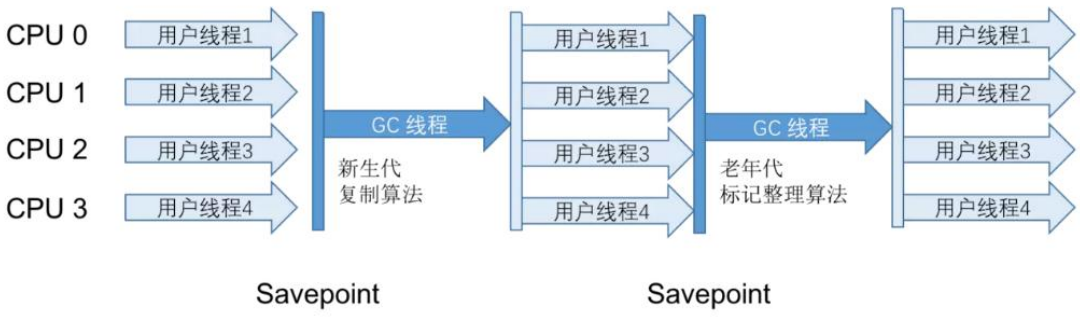


HotSpot 虚拟机的垃圾收集器

说明：如果两个收集器之间存在连线说明他们之间可以搭配使用。

2.5.1 Serial 收集器

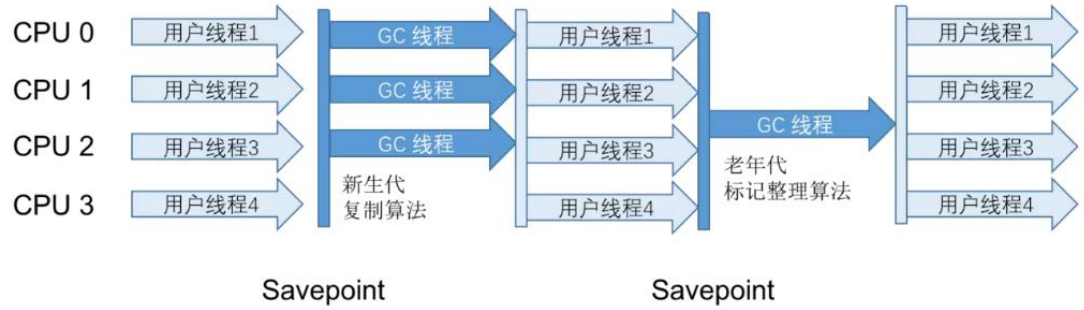
这是一个单线程收集器。意味着它只会使用一个 CPU 或一条收集线程去完成收集工作 ,并且在进行垃圾回收时必须暂停其它所有的工作线程直到收集结束。



Serial / Serial Old 收集器运行示意图

2.5.2 ParNew 收集器

可以认为是 Serial 收集器的多线程版本。



ParNew / Serial Old 收集器运行示意图

并行：Parallel

指多条垃圾收集线程并行工作，此时用户线程处于等待状态

并发：Concurrent

指用户线程和垃圾回收线程同时执行(不一定是并行，有可能是交叉执行)，用户进程在运行，而垃圾回收线程在另一个 CPU 上运行。

2.5.3 Parallel Scavenge 收集器

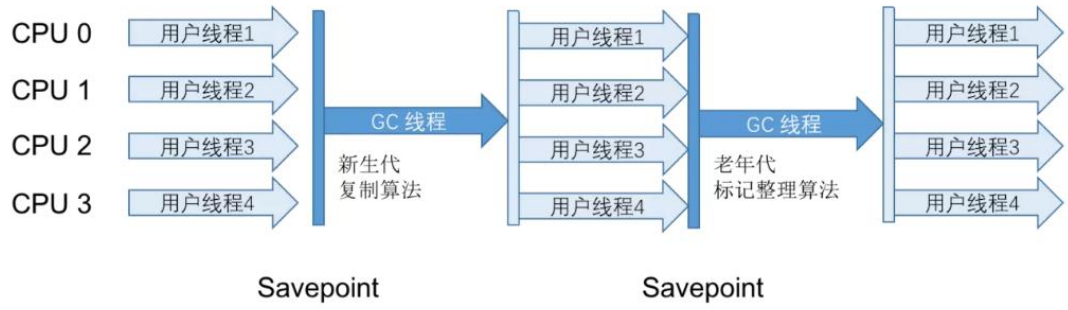
这是一个新生代收集器，也是使用复制算法实现，同时也是并行的多线程收集器。

CMS 等收集器的关注点是尽可能地缩短垃圾收集时用户线程所停顿的时间，而 Parallel Scavenge 收集器的目的是达到一个可控制的吞吐量(Throughput = 运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间))。

作为一个吞吐量优先的收集器，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整停顿时间。这就是 GC 的自适应调整策略(GC Ergonomics)。

2.5.4 Serial Old 收集器

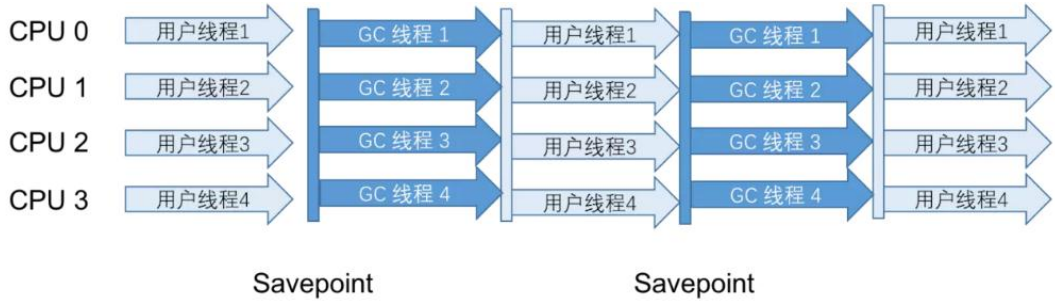
收集器的老年代版本，单线程，使用 标记 —— 整理。



Serial / Serial Old 收集器运行示意图

2.5.5 Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本。多线程，使用 标记 —— 整理



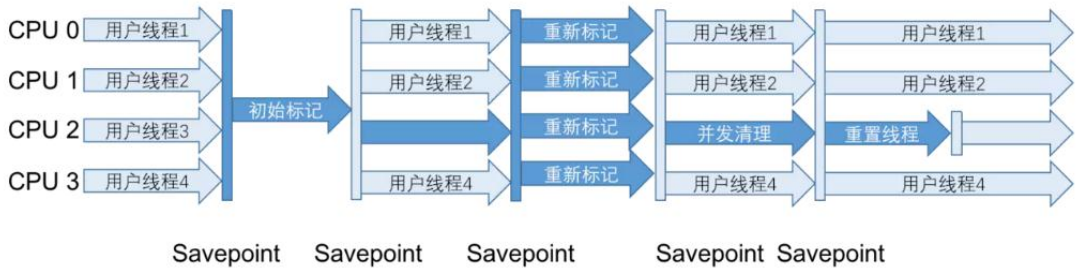
Parallel Scavenge / Parallel Old 收集器运行示意图

2.5.6 CMS 收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为目标的收集器。基于标记 —— 清除 算法实现。

运作步骤:

初始标记(CMS initial mark)：标记 GC Roots 能直接关联到的对象
并发标记(CMS concurrent mark)：进行 GC Roots Tracing
重新标记(CMS remark)：修正并发标记期间的变动部分
并发清除(CMS concurrent sweep)



Concurrent Mark Sweep 收集器运行示意图

缺点：对 CPU 资源敏感、无法收集浮动垃圾、标记 —— 清除 算法带来的空间碎片

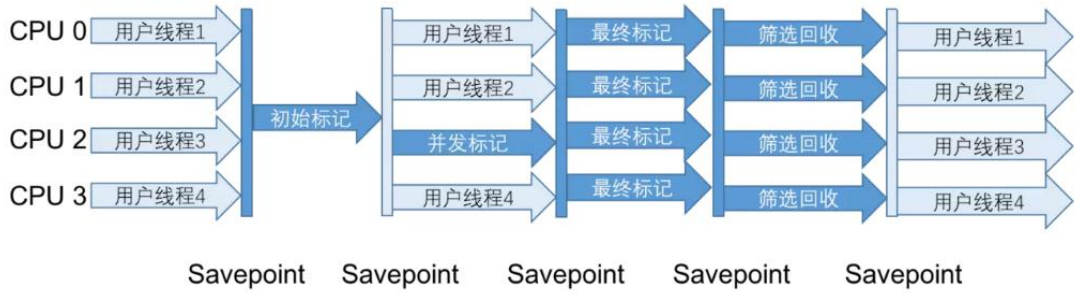
2.5.7 G1 收集器

面向服务端的垃圾回收器。

优点：并行与并发、分代收集、空间整合、可预测停顿。

运作步骤:

初始标记(Initial Marking)
并发标记(Concurrent Marking)
最终标记(Final Marking)
筛选回收(Live Data Counting and Evacuation)



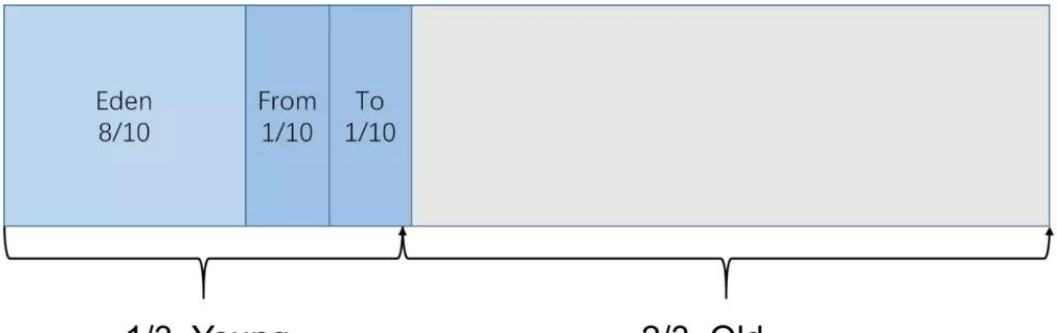
G1 收集器运行示意图

2.6 内存分配与回收策略

2.6.1 对象优先在 Eden 分配

对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲区，将线程优先在（TLAB）上分配。少数情况会直接分配在老年代中。

一般来说 Java 堆的内存模型如下图所示：



Java 堆的内存模型

新生代 GC (Minor GC)

发生在新生代的垃圾回收动作，频繁，速度快。

老年代 GC (Major GC / Full GC)

发生在老年代的垃圾回收动作，出现了 Major GC 经常会伴随至少一次 Minor GC(非绝对)。Major GC 的速度一般会比 Minor GC 慢十倍以上。

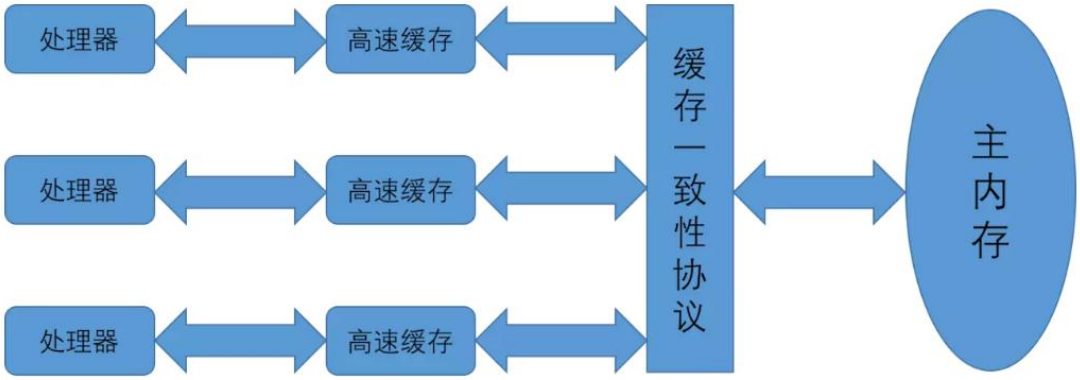
2.6.2 大对象直接进入老年代

2.6.3 长期存活的对象将进入老年代

2.6.4 动态对象年龄判定

2.6.5 空间分配担保

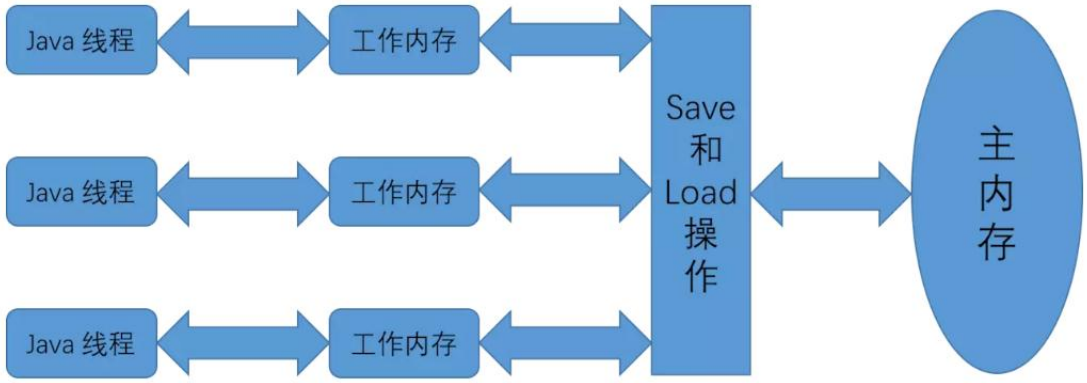
3. Java 内存模型与线程



处理器、高速缓存、主内存间的交互关系

3.1 Java 内存模型

屏蔽掉各种硬件和操作系统的内存访问差异。



线程、工作内存、主内存三者的交互关系

3.1.1 主内存和工作内存之间的交互

操作	作用对象	解释
lock	主内存	把一个变量标识为一条线程独占的状态
unlock	主内存	把一个处于锁定状态的变量释放出来，释放后才可被其他线程锁定
read	主内存	把一个变量的值从主内存传输到线程工作内存中，以便 load 操作使用
load	工作内存	把 read 操作从主内存中得到的变量值放入工作内存中
use	工作内存	把工作内存中一个变量的值传递给执行引擎， 每当虚拟机遇到一个需要使用到变量值的字节码指令时将会执行这个操作
assign	工作内存	把一个从执行引擎接收到的值赋接收到的值赋给工作内存的变量， 每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作
store	工作内存	把工作内存中的一个变量的值传送到主内存中，以便 write 操作
write	工作内存	把 store 操作从工作内存中得到的变量的值放入主内存的变量中

3.1.2 对于 volatile 型变量的特殊规则

关键字 volatile 是 Java 虚拟机提供的最轻量级的同步机制。

一个变量被定义为 volatile 的特性：

保证此变量对所有线程的可见性。但是操作并非原子操作，并发情况下不安全。
如果不符合 运算结果并不依赖变量当前值，或者能够确保只有单一的线程修改变量的值 和变量不需要与其他的状态变量共同参与不变约束 就要通过加锁(使用 synchronize 或 java.util.concurrent 中的原子类)来保证原子性。

禁止指令重排序优化。
通过插入内存屏障保证一致性。

3.1.3 对于 long 和 double 型变量的特殊规则

Java 要求对于主内存和工作内存之间的八个操作都是原子性的，但是对于 64 位的数据类

型，有一条宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性。这就是 long 和 double 的非原子性协定。

3.1.4 原子性、可见性与有序性

回顾下并发下应该注意操作的那些特性是什么，同时加深理解。

原子性(Atomicity)
由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store 和 write。大致可以认为基本数据类型的操作是原子性的。同时 lock 和 unlock 可以保证更大范围操作的原子性。而 synchronize 同步块操作的原子性是用更高层次的字节码指令 monitorenter 和 monitorexit 来隐式操作的。

可见性(Visibility)
是指当一个线程修改了共享变量的值，其他线程也能够立即得知这个通知。主要操作细节就是修改值后将值同步至主内存(volatile 值使用前都会从主内存刷新)，除了 volatile 还有 synchronize 和 final 可以保证可见性。同步块的可见性是由“对一个变量执行 unlock 操作之前，必须先把此变量同步会主内存中(store、write 操作)”这条规则获得。而 final 可见性是指：被 final 修饰的字段在构造器中一旦完成，并且构造器没有把 “this” 的引用传递出去(this 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象)，那在其他线程中就能看见 final 字段的值。

有序性(Ordering)
如果在被线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句指“线程内表现为串行的语义”，后半句是指“指令重排”现象和“工作内存与主内存同步延迟”现象。Java 语言通过 volatile 和 synchronize 两个关键字来保证线程之间操作的有序性。volatile 自身就禁止指令重排，而 synchronize 则是由“一个变量在同一时刻指允许一条线程对其进行 lock 操作”这条规则获得，这条规则决定了持有同一个锁的两个同步块只能串行的进入。

3.1.5 先行发生原则

也就是 happens-before 原则。这个原则是判断数据是否存在竞争、线程是否安全的主要依据。先行发生是 Java 内存模型中定义的两项操作之间的偏序关系。

天然的先行发生关系

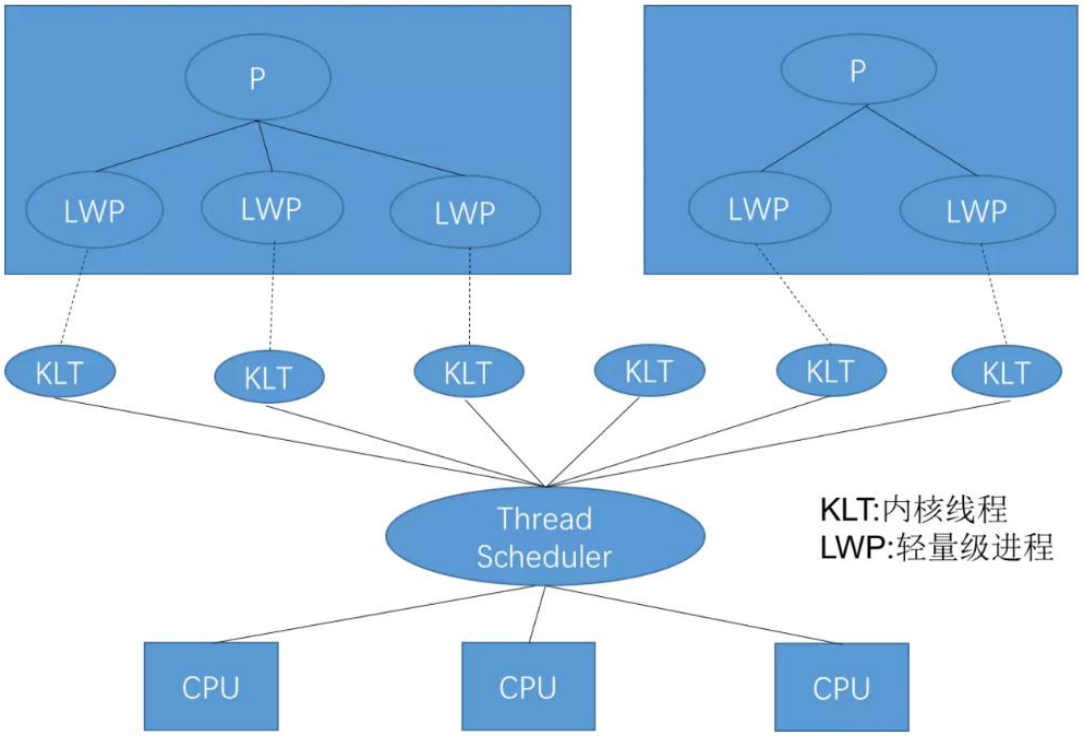
规则	解释
程序次序规则	在一个线程内，代码按照书写的控制流顺序执行
管程锁定规则	一个 unlock 操作先行发生于后面对同一个锁的 lock 操作
volatile 变量规则	volatile 变量的写操作先行发生于后面对这个变量的读操作
线程启动规则	Thread 对象的 start() 方法先行发生于此线程的每一个动作
线程终止规则	线程中所有的操作都先行发生于对此线程的终止检测 (通过 Thread.join() 方法结束、 Thread.isAlive() 的返回值检测)
线程中断规则	对线程 interrupt() 方法调用优先发生于被中断线程的代码检测到中断事件的发生 (通过 Thread.interrupted() 方法检测)
对象终结规则	一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize() 方法的开始
传递性	如果操作 A 先于 操作 B 发生，操作 B 先于 操作 C 发生，那么操作 A 先于 操作 C

3.2 Java 与线程

3.2.1 线程的实现

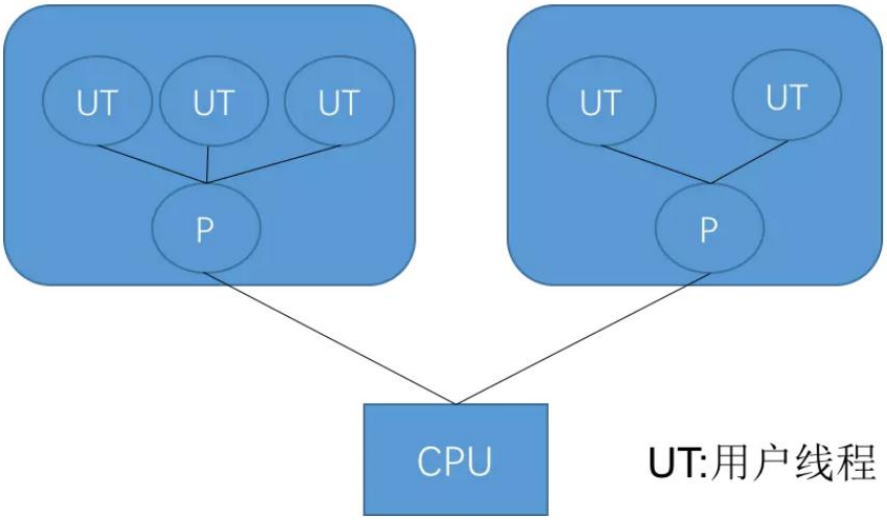
使用内核线程实现

直接由操作系统内核支持的线程，这种线程由内核完成切换。程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口 —— 轻量级进程(LWP)，轻量级进程就是我们通常意义上所讲的线程，每个轻量级进程都有一个内核级线程支持。



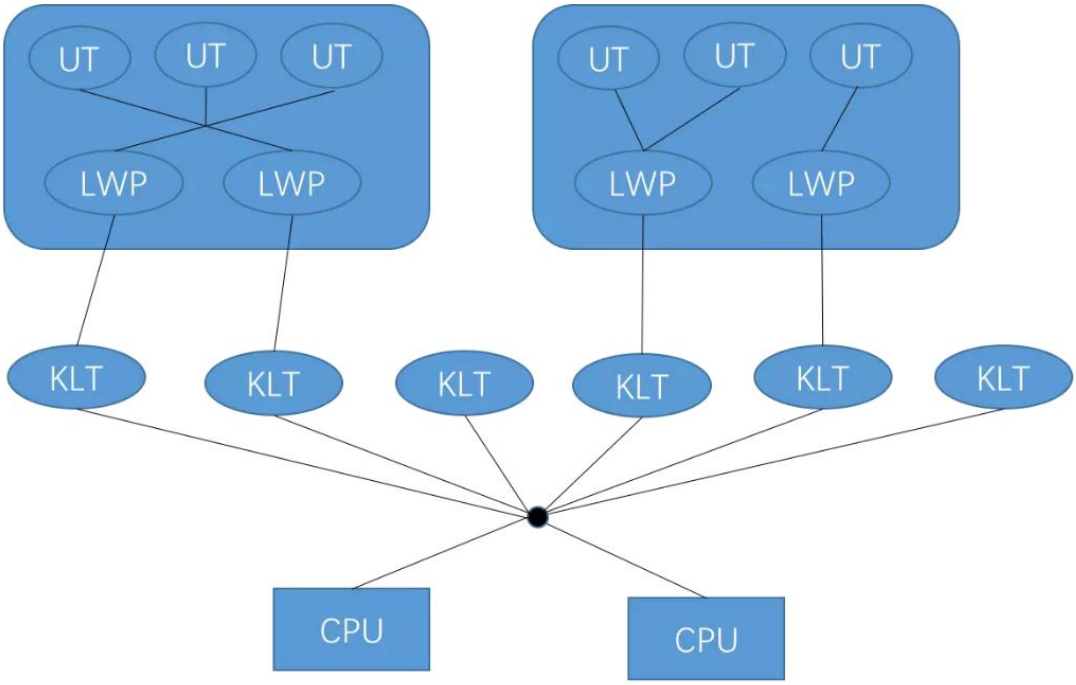
使用用户线程实现

广义上来说，只要不是内核线程就可以认为是用户线程，因此可以认为轻量级进程也属于用户线程。狭义上说是完全建立在用户空间的线程库上的并且内核系统不可感知的。



使用用户线程夹加轻量级进程混合实现

直接看图



Java 线程实现

平台不同实现方式不同，可以认为是一条 Java 线程映射到一条轻量级进程。

3.2.2 Java 线程调度

协同式线程调度

线程执行时间由线程自身控制，实现简单，切换线程自己可知，所以基本没有线程同步问题。坏处是执行时间不可控，容易阻塞。

抢占式线程调度

每个线程由系统来分配执行时间。

3.2.3 状态转换

五种状态：

新建(new)
创建后尚未启动的线程。

运行(Runable)
Runable 包括了操作系统线程状态中的 Running 和 Ready，也就是出于此状态的线程有可能正在执行，也有可能正在等待 CPU 为他分配时间。

无限期等待(Waiting)
出于这种状态的线程不会被 CPU 分配时间，它们要等其他线程显示的唤醒。

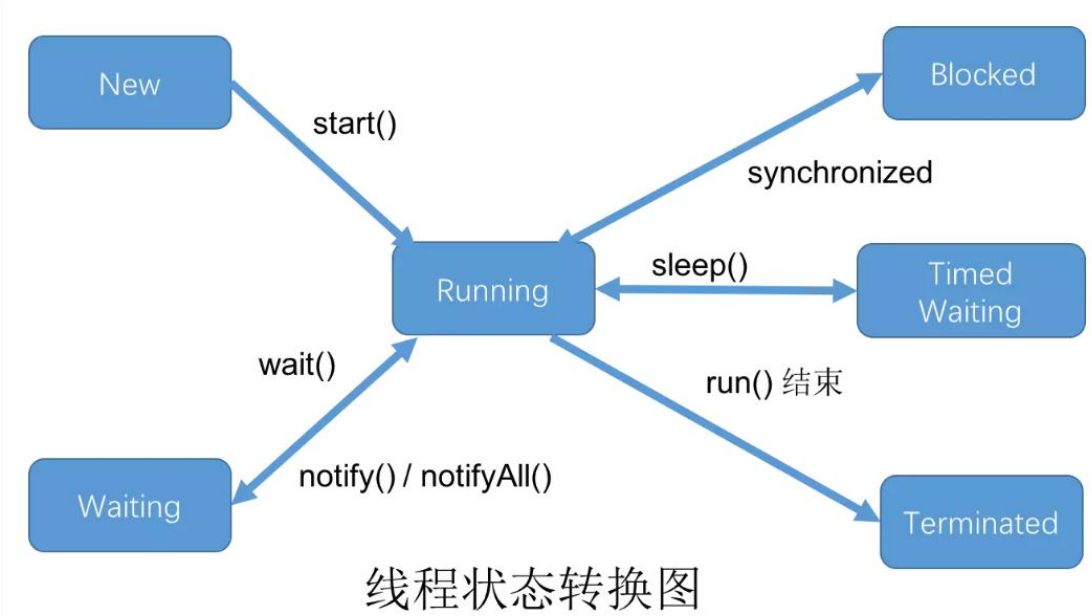
以下方法会然线程进入无限期等待状态：
1.没有设置 Timeout 参数的 Object.wait() 方法。
2.没有设置 Timeout 参数的 Thread.join() 方法。
3.LookSupport.park() 方法。

限期等待(Timed Waiting)
处于这种状态的线程也不会分配时间，不过无需等待配其他线程显示地唤醒，在一定时间后他们会由系统自动唤醒。

以下方法会让线程进入限期等待状态：
1.Thread.sleep() 方法。
2.设置了 Timeout 参数的 Object.wait() 方法。
3.设置了 Timeout 参数的 Thread.join() 方法。
4.LockSupport.parkNanos() 方法。
5.LockSupport.parkUntil() 方法。

阻塞(Blocked)
线程被阻塞了，“阻塞状态”和“等待状态”的区别是：“阻塞状态”在等待着获取一个排他锁，这个时间将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

结束(Terminated)
已终止线程的线程状态。



线程状态转换图

4. 线程安全与锁优化

// 待填

5. 类文件结构
// 待填

有点懒了。。。先贴几个网址吧。

1. Official：The class File Format
2.亦山：《Java 虚拟机原理图解》 1.1、class 文件基本组织结构

6. 虚拟机类加载机制
虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、装换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型。

在 Java 语言中，类型的加载、连接和初始化过程都是在程序运行期间完成的。

6.1 类加载时机
类的生命周期(7 个阶段)

其中加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的。解析阶段可以在初始化之后再开始(运行时绑定或动态绑定或晚期绑定)。

以下五种情况必须对类进行初始化(而加载、验证、准备自然需要在此之前完成)：

遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时没初始化触发初始化。使用场景：使用 new 关键字实例化对象、读取一个类的静态字段(被 final 修饰、已在编译期把结果放入常量池的静态字段除外)、调用一个类的静态方法。
使用 java.lang.reflect 包的方法对类进行反射调用的时候。
当初始化一个类的时候，如果发现其父类还没有进行初始化，则需先触发其父类的初始化。

当虚拟机启动时，用户需指定一个要加载的主类(包含 main() 方法的那个类)，虚拟机会先初始化这个主类。

当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需先触发其初始化。

前面的五种方式是对一个类的主动引用，除此之外，所有引用类的方法都不会触发初始化，佳作被动引用。举几个例子~

```
public class SuperClass {
    static {
        System.out.println("SuperClass init!");
    }
    public static int value = 1127;
}

public class SubClass extends SuperClass {
    static {
        System.out.println("SubClass init!");
    }
}

public class ConstClass {
    static {
        System.out.println("ConstClass init!");
    }
    public static final String HELLOWORLD = "hello world!"
}

public class NotInitialization {
    public static void main(String[] args) {
        System.out.println(SubClass.value);
        /**
         *    output : SuperClass init!
         *
         * 通过子类引用父类的静态对象不会导致子类的初始化
         * 只有直接定义这个字段的类才会被初始化
         */

        SuperClass[] sca = new SuperClass[10];
        /**
         *    output :
         *
         * 通过数组定义来引用类不会触发此类的初始化
         * 虚拟机在运行时动态创建了一个数组类
```

```
         */

        System.out.println(ConstClass.HELLOWORLD);
        /**
         *   output :
         *
         * 常量在编译阶段会存入调用类的常量池当中，本质上并没有直接引用到定义类常量的类，
         * 因此不会触发定义常量的类的初始化。
         * “hello world”在编译期常量传播优化时已经存储到 NotInitialization 常量池中了。
         */
    }
}
```

6.2 类的加载过程

6.2.1 加载

通过一个类的全限定名来获取定义次类的二进制流(ZIP 包、网络、运算生成、JSP 生成、数据库读取)。
将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法去这个类的各种数据的访问入口。
数组类的特殊性：数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型最终是要靠类加载器去创建的，数组创建过程如下：

如果数组的组件类型是引用类型，那就递归采用类加载加载。
如果数组的组件类型不是引用类型，Java 虚拟机会把数组标记为引导类加载器关联。
数组类的可见性与他的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为 public。
内存中实例的 java.lang.Class 对象存在方法区中。作为程序访问方法区中这些类型数据的外部接口。
加载阶段与连接阶段的部分内容是交叉进行的，但是开始时间保持先后顺序。

6.2.2 验证

是连接的第一步，确保 Class 文件的字节流中包含的信息符合当前虚拟机要求。

文件格式验证

是否以魔数 0xCAFEBABE 开头
主、次版本号是否在当前虚拟机处理范围之内
常量池的常量是否有不被支持常量的类型（检查常量 tag 标志）

指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量
CONSTANT_Utf8_info 型的常量中是否有不符合 UTF8 编码的数据
Class 文件中各个部分集文件本身是否有被删除的附加的其他信息
……
只有通过这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面 3 个验证阶段全部是基于方法区的存储结构进行的，不再直接操作字节流。

元数据验证

这个类是否有父类（除 java.lang.Object 之外）
这个类的父类是否继承了不允许被继承的类（final 修饰的类）
如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法
类中的字段、方法是否与父类产生矛盾（覆盖父类 final 字段、出现不符合规范的重载）
这一阶段主要是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

字节码验证

保证任意时刻操作数栈的数据类型与指令代码序列都鞯配合工作（不会出现按照 long 类型读一个 int 型数据）
保证跳转指令不会跳转到方法体以外的字节码指令上
保证方法体中的类型转换是有效的（子类对象赋值给父类数据类型是安全的，反过来不合法的）
……
这是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段对类的方法体进行校验分析，保证校验类的方法在运行时不会做出危害虚拟机安全的事件。

符号引用验证

符号引用中通过字符创描述的全限定名是否能找到对应的类
在指定类中是否存在符方法的字段描述符以及简单名称所描述的方法和字段
符号引用中的类、字段、方法的访问性（private、protected、public、default）是否可被当前类访问
……
最后一个阶段的校验发生在迅疾将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验，还有以上提及的内容。
符号引用的目的是确保解析动作能正常执行，如果无法通过符号引用验证将抛出一个 java.lang.IncompatibleClass.ChangeError 异常的子类。如 java.lang.IllegalAccessError、java.lang.NoSuchFieldError、java.lang.NoSuchMethodError 等。

6.2.3 准备

这个阶段正式为类分配内存并设置类变量初始值，内存在方法去中分配(含 static 修饰的变量不含实例变量)。

public static int value = 1127;
这句代码在初始值设置之后为 0，因为这时候尚未开始执行任何 Java 方法。而把 value 赋值为 1127 的 putstatic 指令是程序被编译后，存放于 clinit() 方法中，所以初始化阶段才会对 value 进行赋值。

基本数据类型的零值

数据类型	零值	数据类型	零值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null

特殊情况：如果类字段的字段属性表中存在 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 1127。

6.2.4 解析

这个阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用
符号引用以一组符号来描述所引用的目标，符号可以使任何形式的字面量。
直接引用
直接引用可以使直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用和迅疾的内存布局实现有关
解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行，分别对应于常量池的 7 中常量类型。

6.2.5 初始化

前面过程都是以虚拟机主导，而初始化阶段开始执行类中的 Java 代码。

6.3 类加载器

通过一个类的全限定名来获取描述此类的二进制字节流。

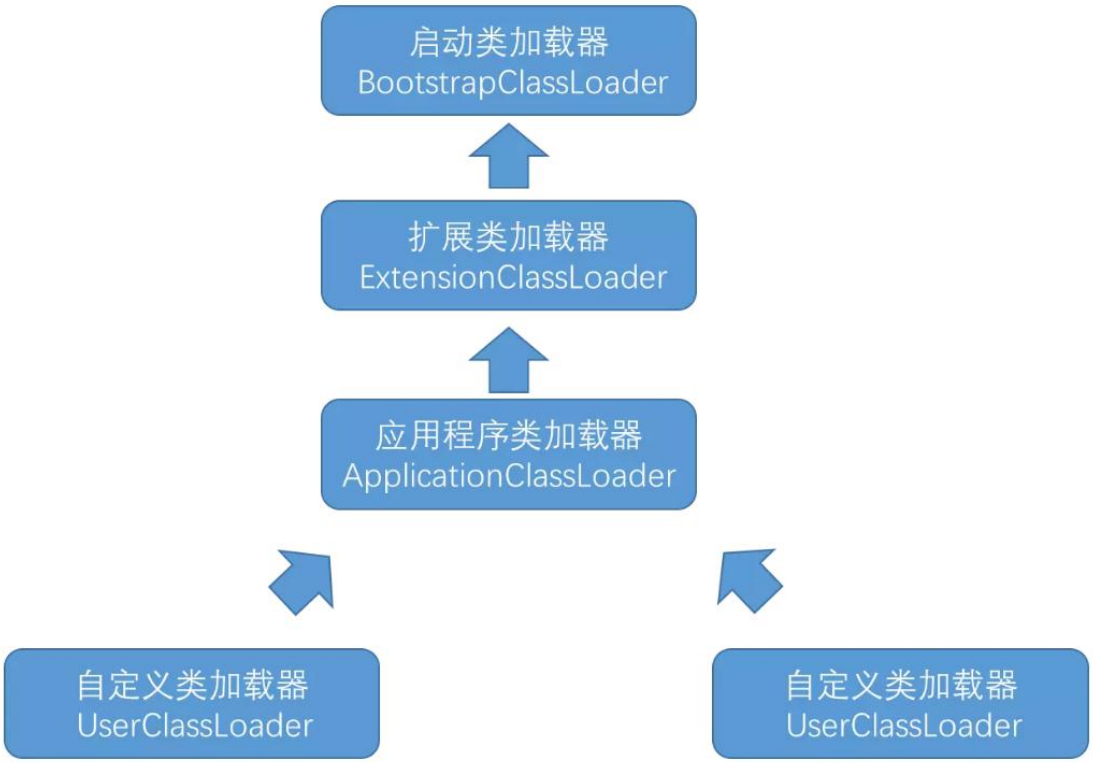
6.3.1 双亲委派模型

从 Java 虚拟机角度讲，只存在两种类加载器：一种是启动类加载器（C++ 实现，是虚拟机的一部分）；另一种是其他所有类的加载器（Java 实现，独立于虚拟机外部且全继承自 java.lang.ClassLoader）

启动类加载器
加载 lib 下或被 -Xbootclasspath 路径下的类

扩展类加载器
加载 lib/ext 或者被 java.ext.dirs 系统变量所指定的路径下的类

引用程序类加载器
ClassLoader 负责，加载用户路径上所指定的类库。



除顶层启动类加载器之外，其他都有自己的父类加载器。
工作过程：如果一个类加载器收到一个类加载的请求，它首先不会自己加载，而是把这个请求委派给父类加载器。只有父类无法完成时子类才会尝试加载。

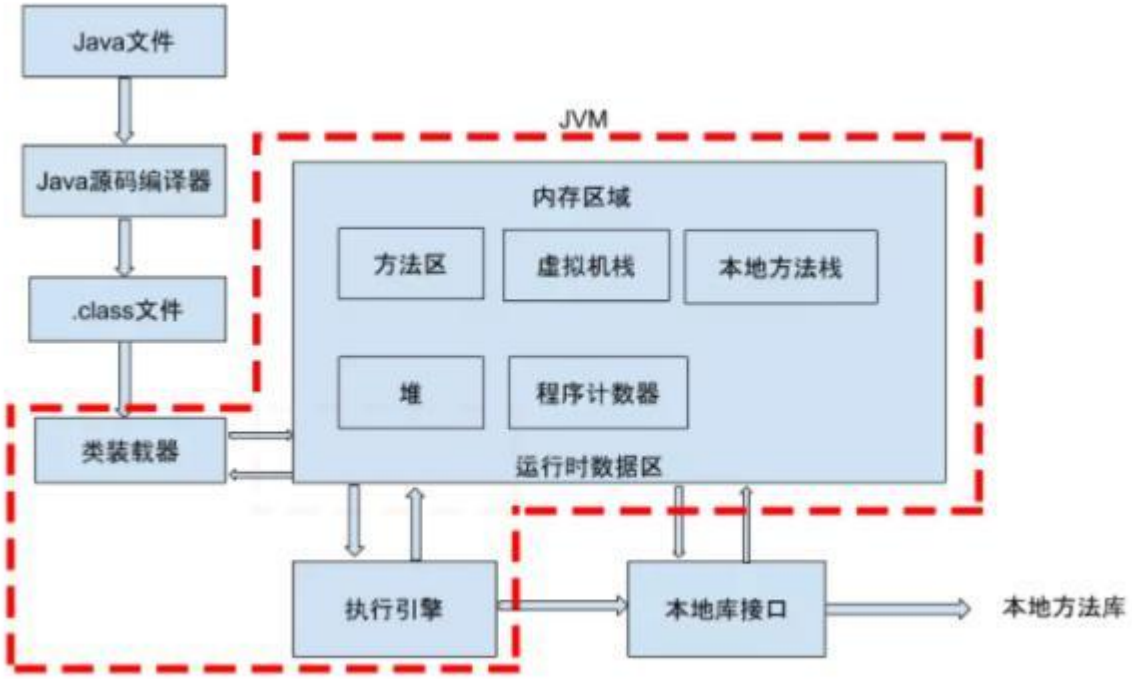
6.3.2 破坏双亲委派模型

keyword：线程上下文加载器(Thread Context ClassLoader)

8. 类加载过程

一、什么是类的加载

在介绍类的加载机制之前，先来看看，类的加载机制在整个 java 程序运行期间处于一个什么环节，下面使用一张图来表示：



从上图可以看，java 文件通过编译器变成了.class 文件，接下来类加载器又将这些.class 文件加载到 JVM 中。其中**类装载器**的**作用**其实就是类的加载。今天我们要讨论的就是这个环节。有了这个印象之后我们再来看**类的加载**的概念：

其实可以一句话来解释：类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 java.lang.Class 对象，用来封装类在方法区内的数据结构。

到现在为止，我们基本上对类加载机制处于整个程序运行的环节位置，还有类加载机制的概念有了基本的印象。在类加载.class 文件之前，还有两个问题需要我们去弄清楚：

1、在什么时候才会启动类加载器？

其实，类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM 规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class 文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才

报告错误（LinkageError 错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

2、从哪个地方去加载.class 文件

在这里进行一个简单的分类。例举了 5 个来源

（1）本地磁盘

（2）网上加载.class 文件（Applet）

（3）从数据库中

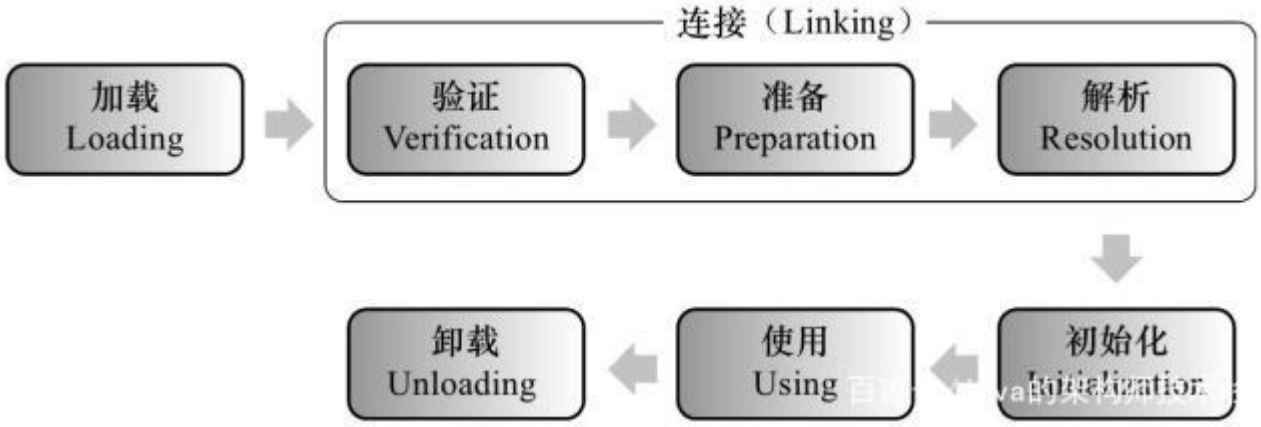
（4）压缩文件中（ZAR，jar 等）

（5）从其他文件生成的（JSP 应用）

有了这个认识之后，下面就开始讲讲，类加载机制了。首先看的就是类加载机制的过程。

二、类加载的过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。它们的顺序如下图所示：



其中**类加载的过程包括了加载、验证、准备、解析、初始化五个阶段**。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始。另外注意这里的几个阶段是按顺序**开始**，而不是按顺序**进行**或**完成**，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

下面就一个一个去分析一下这几个过程。

1、加载

”加载“是”类加机制”的第一个过程，在加载阶段，虚拟机主要完成三件事：

（1）通过一个类的全限定名来获取其定义的二进制字节流

（2）将这个字节流所代表的的静态存储结构转化为方法区的运行时数据结构

（3）在堆中生成一个代表这个类的 Class 对象，作为方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段是可控性最强的阶段，因为程序员可以使用系统的类加载器加载，还可以使用自己的类加载器加载。我们在最后一部分会详细介绍这个类加载器。在这里我们只需要知道类加载器的作用就是上面虚拟机需要完成的三件事，仅此而已就好了。

2、验证

验证的主要作用就是确保被加载的类的正确性。也是连接阶段的第一步。说白了也就是我们加载好的.class 文件不能对我们的虚拟机有危害，所以先检测验证一下。他主要是完成四个阶段的验证：

（1）文件格式的验证：验证.class 文件字节流是否符合 class 文件的格式的规范，并且能够被当前版本的虚拟机处理。这里面主要对魔数、主版本号、常量池等等的校验（魔数、主版本号都是.class 文件里面包含的数据信息、在这里可以不用理解）。

（2）元数据验证：主要是对字节码描述的信息进行语义分析，以保证其描述的信息符合 java 语言规范的要求，比如说验证这个类是不是有父类，类中的字段方法是不是和父类冲突等等。

（3）字节码验证：这是整个验证过程最复杂的阶段，主要是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在元数据验证阶段对数据类型做出验证后，这个阶段主要对类的方法做出分析，保证类的方法在运行时不会做出威海虚拟机安全的事。

（4）符号引用验证：它是验证的最后一个阶段，发生在虚拟机将符号引用转化为直接引用的时候。主要是对类自身以外的信息进行校验。目的是确保解析动作能够完成。

对整个类加载机制而言，验证阶段是一个很重要但是非必需的阶段，如果我们的代码能够确保没有问题，那么我们就没有必要去验证，毕竟验证需要花费一定的的时间。当然我们可以使用-Xverfity:none 来关闭大部分的验证。

3、准备

准备阶段主要为类变量分配内存并设置初始值。这些内存都在方法区分配。

在这个阶段我们只需要注意两点就好了，也就是类变量和初始值两个关键词：

（1）类变量（static）会分配内存，但是实例变量不会，实例变量主要随着对象的实例化一块分配到 java 堆中，

（2）这里的初始值指的是数据类型默认值，而不是代码中被显示赋予的值。比如

public static int value = 1; //在这里准备阶段过后的 value 值为 0，而不是 1。赋值为 1 的动作在初始化阶段。

当然还有其他的默认值。

数据类型	默认值	数据类型	默认值
int	0	boolean	false
long	0L	float	0.0f
short	0(short)	double	0.0d

注意，在上面 value 是被 static 所修饰的准备阶段之后是 0，但是如果同时被 final 和 static 修饰准备阶段之后就是 1 了。我们可以理解为 static final 在编译器就将结果放入调用它的类的常量池中了。

4、解析

解析阶段主要是虚拟机将常量池中的符号引用转化为直接引用的过程。什么是符号应用和直接引用呢？

符号引用：以一组符号来描述所引用的目标，可以是任何形式的字面量，只要是能无歧义的定位到目标就好，就好比在班级中，老师可以用张三来代表你，也可以用你的学号来代表你，但无论任何方式这些都只是一个代号（符号），这个代号指向你（符号引用）直接引用：直接引用是可以指向目标的指针、相对偏移量或者是一个能直接或间接定位到目标的句柄。和虚拟机实现的内存有关，不同的虚拟机直接引用一般不同。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行。

5、初始化

这是类加载机制的最后一步，在这个阶段，java 程序代码才开始真正执行。我们知道，在准备阶段已经为类变量赋过一次值。在初始化阶端，程序员可以根据自己的需求来赋值了。一句话描述这个阶段就是执行类构造器< clinit >()方法的过程。

在初始化阶段，主要为类的静态变量赋予正确的初始值，JVM 负责对类进行初始化，主要对类变量进行初始化。在 Java 中对类变量进行初始值设定有两种方式：

①声明类变量是指定初始值

②使用静态代码块为类变量指定初始值

JVM 初始化步骤

1、假如这个类还没有被加载和连接，则程序先加载并连接该类

2、假如该类的直接父类还没有被初始化，则先初始化其直接父类

3、假如类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

创建类的实例，也就是 new 的方式访问某个类或接口的静态变量，或者对该静态变量赋值调用类的静态方法反射（如 Class.forName(“com.shengsiyuan.Test”)）初始化某个类的子类，则其父类也会被初始化 Java 虚拟机启动时被标明为启动类的类（JavaTest），直接使用 java.exe 命令来运行某个主类好了，到目前为止就是类加载机制的整个过程，但是还有一个重要的概念，那就是类加载器。在加载阶段其实我们提到过类加载器，说是在后面详细说，在这就好好地介绍一下类加载器。

三、类加载器

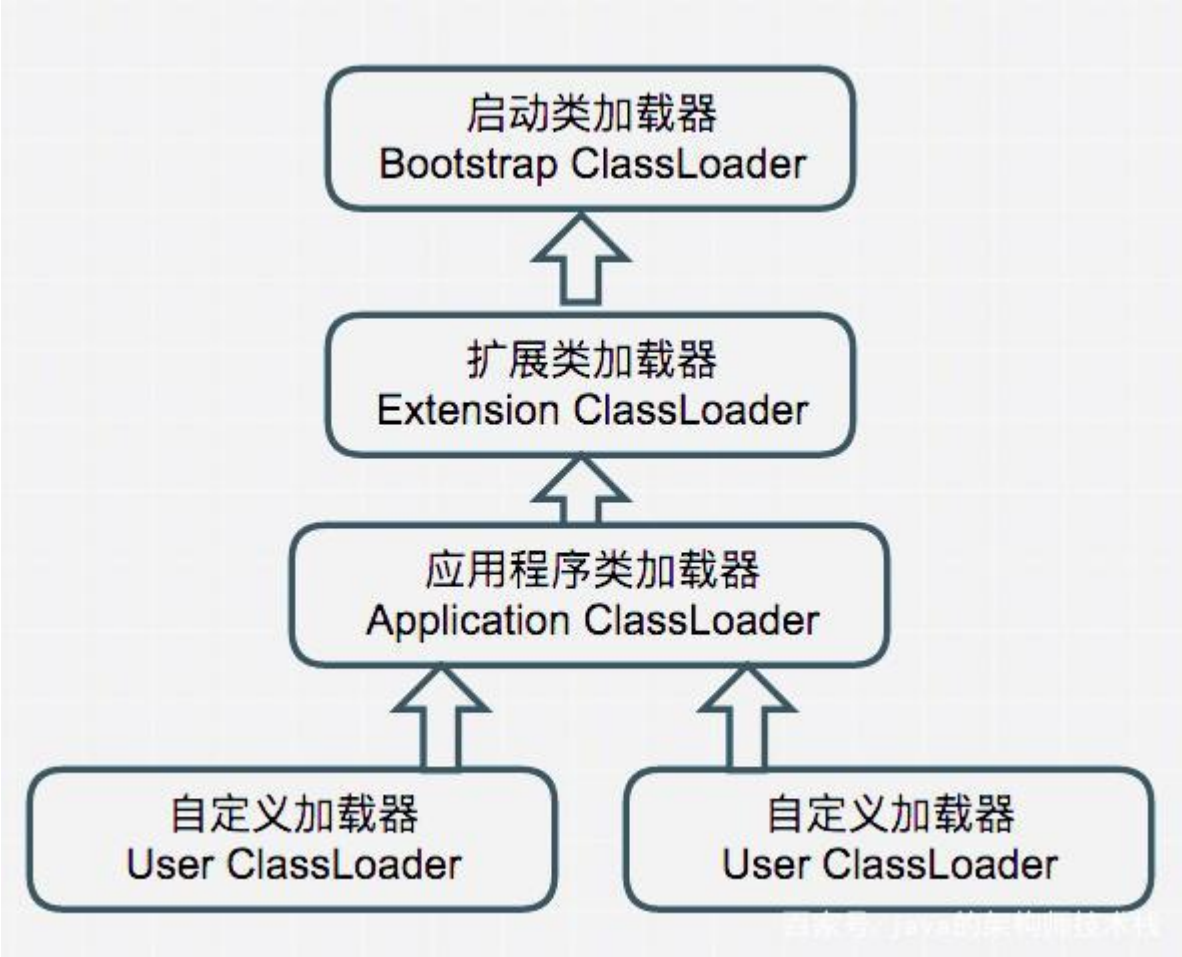
虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类。

1、Java 语言系统自带有三个类加载器:

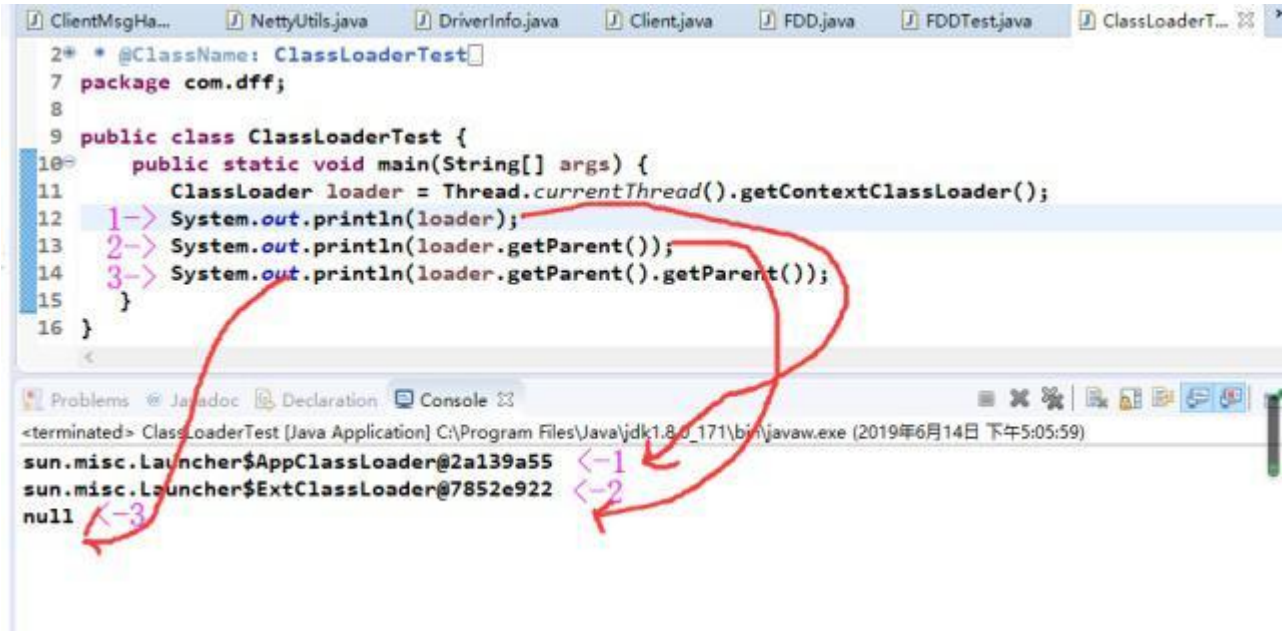
Bootstrap ClassLoader ：最顶层的加载类，主要加载核心类库，也就是我们环境变量下面%JRE_HOME%\lib 下的 rt.jar、resources.jar、charsets.jar 和 class 等。另外需要注意的是可以通过启动 jvm 时指定-Xbootclasspath 和路径来改变 Bootstrap ClassLoader 的加载目录。比如 java -Xbootclasspath/a:path 被指定的文件追加到默认的 bootstrap 路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些 jar 包是不是存在于这个目录。Extention ClassLoader ：扩展的类加载器，加载目录%JRE_HOME%\lib\ext 目录下的 jar 包和 class 文件。还可以加载-D java.ext.dirs 选项指定的目录。Appclass Loader：也称为 SystemAppClass。 加载当前应用的 classpath 的所有类。我们看到 java 为我们提供了三个类加载器，应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。这三种类加载器的加载顺序是什么呢？

Bootstrap ClassLoader > Extention ClassLoader > Appclass Loader

一张图来看一下他们的层次关系


启动类加载器
Bootstrap ClassLoader
扩展类加载器
Extension ClassLoader
应用程序类加载器
Application ClassLoader
自定义加载器
User ClassLoader
自定义加载器
User ClassLoader

代码验证一下：


ClientMsgHa...
NettyUtils.java
DriverInfo.java
Client.java
FDD.java
FDDTest.java
ClassLoaderT...
* @ClassName: ClassLoaderTest
package com.dff;
public class ClassLoaderTest {
public static void main(String[] args) {
ClassLoader loader = Thread.currentThread().getContextClassLoader();
1-> System.out.println(loader);
2-> System.out.println(loader.getParent());
3-> System.out.println(loader.getParent().getParent());
}
}
Problems
Javadoc
Declaration
Console
<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jdk1.8.0_171\bin\javaw.exe (2019年6月14日 下午5:05:59)
sun.misc.Launcher$AppClassLoader@2a139a55
<-1
sun.misc.Launcher$ExtClassLoader@7852e922
<-2
null
<-3

从上面的结果可以看出，并没有获取到 ExtClassLoader 的父 Loader，原因是 Bootstrap Loader（引导类加载器）是用 C 语言实现的，找不到一个确定的返回父 Loader 的方式，于是就返回 null。

2、类加载的三种方式

认识了这三种类加载器，接下来我们看看类加载的三种方式。

（1）通过命令行启动应用时由 JVM 初始化加载含有 main()方法的主类。

（2）通过 Class.forName()方法动态加载，会默认执行初始化块（static{}），但是 Class.forName(name,initialize,loader)中的 initialze 可指定是否要执行初始化块。

（3）通过 ClassLoader.loadClass()方法动态加载，不会执行初始化块。

下面代码来演示一下

首先我们定义一个 FDD 类

```
public class FDD {static { System.out.println("我是静态代码块。。。。"); }}
```

然后我们看一下如何去加载

```
package com.fdd.test;public class FDDloaderTest { public static void main(String[] args) throws ClassNotFoundException { ClassLoader loader = HelloWorld.class.getClassLoader(); System.out.println(loader);// 一、使用 ClassLoader.loadClass()来加载类，不会执行初始化块 loader.loadClass("Fdd"); //二、 使用 Class.forName()来加载类，默认会执行初始化块 Class.forName("Fdd"); //三、使用 Class.forName()来加载类，指定 ClassLoader，初始化时不执行静态块 Class.forName("Fdd", false, loader); } }
```

上面是同不同的方式去加载类，结果是不一样的。

3、双亲委派原则

他的工作流程是： 当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。这个理解起来就简单了，比如说，另外一个人给小费，自己不会先去直接拿来塞自己钱包，我们先把钱给领导，领导再给领导，一直到公司老板，老板不想要了，再一级一级往下分。老板要是要这个钱，下面的领导和自己就一分钱没有了。（例子不好，理解就好）

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个 Object 对象。双亲委派原则归纳一下就是：

可以避免重复加载，父类已经加载了，子类就不需要再次加载更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心 api，会带来相关隐患。**4、自定义类加载器**

在这一部分第一小节中，我们提到了 java 系统为我们提供的三种类加载器，还给出了他们的层次关系图，最下面就是自定义类加载器，那么我们如何自己定义类加载器呢？这主要有两种方式

（1）遵守双亲委派模型：继承 ClassLoader，重写 findClass()方法。

（2）破坏双亲委派模型：继承 ClassLoader,重写 loadClass()方法。 通常我们推荐采用第一种方法自定义类加载器，最大程度上的遵守双亲委派模型。

我们看一下实现步骤

（1）创建一个类继承 ClassLoader 抽象类

（2）重写 findClass()方法

（3）在 findClass()方法中调用 defineClass()

代码实现一下：

```
public class MyClassLoader extends ClassLoader {private String libPath; public DiskClassLoader(String path) { libPath = path; }@Override protected Class<?> findClass(String name) throws ClassNotFoundException { String fileName = getFileName(name); File file = new File(libPath,fileName); try { FileInputStream is = new FileInputStream(file); ByteArrayOutputStream bos = new ByteArrayOutputStream(); int len = 0; try { while ((len = is.read()) != -1) { bos.write(len); } } catch (IOException e) { e.printStackTrace(); } byte[] data = bos.toByteArray(); is.close(); bos.close(); return defineClass(name,data,0,data.length); } catch (IOException e) {e.printStackTrace(); } return super.findClass(name); } //获取要加载的 class 文件名 private String getFileName(String name) { int index = name.lastIndexOf('.'); if(index == -1){ return name+".class"; }else{return name.substring(index+1)+".class"; } }}
```

接下来我们就可以自己去加载类了，使用方法大体就是两行

```
MyClassLoader diskLoader = new MyClassLoader("D:\\lib");//加载 class 文件，注意是 com.fdd.TestClass c = diskLoader.loadClass("com.fdd.Test");
```

9. 反射

定义

JAVA 反射机制**是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 java 语言的反射机制。**

用途

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放，这时候就可以利用 Java 的反射机制通过反射来获取所需的私有成员或是方法。当然，也不是所有的都适合反射，之前就遇到一个案例，通过反射得到的结果与预期不符。阅读源码发现，经过层层调用后在最终返回结果的地方对应用的权限进行了校验，对于没有权限的应用返回值是没有意义的缺省值，否则返回实际值起到保护用户的隐私目的。

反射机制的相关类

与 Java 反射相关的类如下：

类名	用途
Class类	代表类的实体，在运行的Java应用程序中表示类和接口
Field类	代表类的成员变量（成员变量也称为类的属性）
Method类	代表类的方法
Constructor类	代表类的构造方法

Class 类

Class 代表类的实体，在运行的 Java 应用程序中表示类和接口。在这个类中提供了很多有用的方法，这里对他们简单的分类介绍。

- **获得类相关的方法**

方法	用途
asSubclass(Class<U> clazz)	把传递的类的对象转换成代表其子类的对象
Cast	把对象转换成代表类或是接口的对象
getClassLoader()	获得类的加载器
getClasses()	返回一个数组，数组中包含该类中所有公共类和接口类的对象
getDeclaredClasses()	返回一个数组，数组中包含该类中所有类和接口类的对象
forName(String className)	根据类名返回类的对象
getName()	获得类的完整路径名字
newInstance()	创建类的实例
getPackage()	获得类的包
getSimpleName()	获得类的名字
getSuperclass()	获得当前类继承的父类的名字
getInterfaces()	获得当前类实现的类或是接口

- 获得类中属性相关的方法

方法	用途
getField(String name)	获得某个公有的属性对象
getFields()	获得所有公有的属性对象
getDeclaredField(String name)	获得某个属性对象
getDeclaredFields()	获得所有属性对象

- 获得类中注解相关的方法

方法	用途
getAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的公有注解对象
getAnnotations()	返回该类所有的公有注解对象
getDeclaredAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的所有注解对象
getDeclaredAnnotations()	返回该类所有的注解对象

- 获得类中构造器相关的方法

方法	用途
getConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的公有构造方法
getConstructors()	获得该类的所有公有构造方法
getDeclaredConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的构造方法
getDeclaredConstructors()	获得该类所有构造方法

- 获得类中方法相关的方法

方法	用途
getMethod(String name, Class...<?> parameterTypes)	获得该类某个公有的方法
getMethods()	获得该类所有公有的方法
getDeclaredMethod(String name, Class...<?> parameterTypes)	获得该类某个方法
getDeclaredMethods()	获得该类所有方法

- 类中其他重要的方法

方法	用途
isAnnotation()	如果是注解类型则返回true
isAnnotationPresent(Class<? extends Annotation> annotationClass)	如果是指定类型注解类型则返回true
isAnonymousClass()	如果是匿名类则返回true
isArray()	如果是一个数组类则返回true
isEnum()	如果是枚举类则返回true
isInstance(Object obj)	如果obj是该类的实例则返回true
isInterface()	如果是接口类则返回true
isLocalClass()	如果是局部类则返回true
isMemberClass()	如果是内部类则返回true

Field 类

Field 代表类的成员变量（成员变量也称为类的属性）。

方法	用途
equals(Object obj)	属性与obj相等则返回true
get(Object obj)	获得obj中对应的属性值
set(Object obj, Object value)	设置obj中对应属性值

Method 类

Method 代表类的方法。

方法	用途
invoke(Object obj, Object... args)	传递object对象及参数调用该对象对应的方法

Constructor 类

Constructor 代表类的构造方法。

方法	用途
newInstance(Object... initargs)	根据传递的参数创建类的对象

示例

为了演示反射的使用，首先构造一个与书籍相关的 model——Book.java，然后通过反射方法示例创建对象、反射私有构造方法、反射私有属性、反射私有方法，最后给出两个比较复杂的反射示例——获得当前 ZenMode 和关机 Shutdown。

- **被反射类 Book.java**

```
public class Book{
    private final static String TAG = "BookTag";

    private String name;
```

```
    private String author;

    @Override
    public String toString() {
        return "Book{" +
                "name='" + name + '\'' +
                ", author='" + author + '\'' +
                '}';
    }

    public Book() {
    }

    private Book(String name, String author) {
        this.name = name;
        this.author = author;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    private String declaredMethod(int index) {
        String string = null;
        switch (index) {
            case 0:
                string = "I am declaredMethod 1 !";
                break;
            case 1:
                string = "I am declaredMethod 2 !";
                break;
            default:
```

```
            string = "I am declaredMethod 1 !";
        }

        return string;
    }}
```

- **反射逻辑封装在 ReflectClass.java**

```
public class ReflectClass {
    private final static String TAG = "peter.log.ReflectClass";

    // 创建对象
    public static void reflectNewInstance() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Object objectBook = classBook.newInstance();
            Book book = (Book) objectBook;
            book.setName("Android 进阶之光");
            book.setAuthor("刘望舒");
            Log.d(TAG,"reflectNewInstance book = " + book.toString());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 反射私有的构造方法
    public static void reflectPrivateConstructor() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Constructor<?> declaredConstructorBook =
classBook.getDeclaredConstructor(String.class,String.class);
            declaredConstructorBook.setAccessible(true);
            Object objectBook =
declaredConstructorBook.newInstance("Android 开发艺术探索","任玉刚");
            Book book = (Book) objectBook;
            Log.d(TAG,"reflectPrivateConstructor book = " +
book.toString());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

```
    // 反射私有属性
    public static void reflectPrivateField() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Object objectBook = classBook.newInstance();
            Field fieldTag = classBook.getDeclaredField("TAG");
            fieldTag.setAccessible(true);
            String tag = (String) fieldTag.get(objectBook);
            Log.d(TAG,"reflectPrivateField tag = " + tag);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 反射私有方法
    public static void reflectPrivateMethod() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Method methodBook =
classBook.getDeclaredMethod("declaredMethod",int.class);
            methodBook.setAccessible(true);
            Object objectBook = classBook.newInstance();
            String string = (String) methodBook.invoke(objectBook,0);

            Log.d(TAG,"reflectPrivateMethod string = " + string);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 获得系统 Zenmode 值
    public static int getZenMode() {
        int zenMode = -1;
        try {
            Class<?> cServiceManager =
Class.forName("android.os.ServiceManager");
            Method mGetService = cServiceManager.getMethod("getService",
String.class);
            Object oNotificationManagerService = mGetService.invoke(null,
Context.NOTIFICATION_SERVICE);
            Class<?> cINotificationManagerStub =
Class.forName("android.app.INotificationManager$Stub");
```

```
            Method mAsInterface =
cINotificationManagerStub.getMethod("asInterface",IBinder.class);
            Object oINotificationManager =
mAsInterface.invoke(null,oNotificationManagerService);
            Method mGetZenMode =
cINotificationManagerStub.getMethod("getZenMode");
            zenMode = (int) mGetZenMode.invoke(oINotificationManager);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        return zenMode;
    }

    // 关闭手机
    public static void shutDown() {
        try {
            Class<?> cServiceManager =
Class.forName("android.os.ServiceManager");
            Method mGetService =
cServiceManager.getMethod("getService",String.class);
            Object oPowerManagerService =
mGetService.invoke(null,Context.POWER_SERVICE);
            Class<?> cIPowerManagerStub =
Class.forName("android.os.IPowerManager$Stub");
            Method mShutdown =
cIPowerManagerStub.getMethod("shutdown",boolean.class,String.class,bo
olean.class);
            Method mAsInterface =
cIPowerManagerStub.getMethod("asInterface",IBinder.class);
            Object oIPowerManager =
mAsInterface.invoke(null,oPowerManagerService);
            mShutdown.invoke(oIPowerManager,true,null,true);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void shutdownOrReboot(final boolean shutdown, final
boolean confirm) {
        try {
            Class<?> ServiceManager =
Class.forName("android.os.ServiceManager");
```

```
            // 获得 ServiceManager 的 getService 方法
            Method getService = ServiceManager.getMethod("getService",
java.lang.String.class);
            // 调用 getService 获取 RemoteService
            Object oRemoteService = getService.invoke(null,
Context.POWER_SERVICE);
            // 获得 IPowerManager.Stub 类
            Class<?> cStub =
Class.forName("android.os.IPowerManager$Stub");
            // 获得 asInterface 方法
            Method asInterface = cStub.getMethod("asInterface",
android.os.IBinder.class);
            // 调用 asInterface 方法获取 IPowerManager 对象
            Object oIPowerManager = asInterface.invoke(null,
oRemoteService);
            if (shutdown) {
                // 获得 shutdown()方法
                Method shutdownMethod =
oIPowerManager.getClass().getMethod(
                        "shutdown", boolean.class, String.class,
boolean.class);
                // 调用 shutdown()方法
                shutdownMethod.invoke(oIPowerManager, confirm, null,
false);
            } else {
                // 获得 reboot()方法
                Method rebootMethod =
oIPowerManager.getClass().getMethod("reboot",
                        boolean.class, String.class, boolean.class);
                // 调用 reboot()方法
                rebootMethod.invoke(oIPowerManager, confirm, null,
false);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }}
```

- **调用相应反射逻辑方法**

```
        try {
            // 创建对象
            ReflectClass.reflectNewInstance();
```

```
        // 反射私有的构造方法
        ReflectClass.reflectPrivateConstructor();

        // 反射私有属性
        ReflectClass.reflectPrivateField();

        // 反射私有方法
        ReflectClass.reflectPrivateMethod();
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    Log.d(TAG," zenmode = " + ReflectClass.getZenMode());
```

Log 输出结果如下：

```
08-27 15:11:37.999 11987-11987/com.android.peter.reflectdemo
D/peter.log.ReflectClass: reflectNewInstance book = Book{name='Android
进阶之光', author='刘望舒'}08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateConstructor book = Book{name='Android 开发艺术探索',
author='任玉刚'}08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateField tag = BookTag08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateMethod string = I am declaredMethod 1 !08-27 15:11:38.004
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectDemo:
zenmode = 0
```

总结

本文列举了反射机制使用过程中常用的、重要的一些类及其方法，更多信息和用法需要近一步的阅读 Google 提供的相关文档和示例。

在阅读 Class 类文档时发现一个特点，以通过反射获得 Method 对象为例，一般会提供四种方法，getMethod(parameterTypes)、getMethods()、getDeclaredMethod(parameterTypes)和 getDeclaredMethods()。getMethod(parameterTypes)用来获取某个公有的方法的对象，getMethods()获得该类所有公有的方法，getDeclaredMethod(parameterTypes)获得该类某个方法，getDeclaredMethods()获得该类所有方法。**带有 Declared 修饰的方法可以反射到私有的方法，没有 Declared 修饰的只能用来反射公有的方法。**其他的 Annotation、Field、Constructor 也是如此。

在 ReflectClass 类中还提供了两种反射 PowerManager.shutdown()的方法，在调用的时候会输出如下 log，提示没有相关权限。之前在项目中尝试反射其他方法的时候还遇到过有权限和没权限返回的值不一样的情况。**如果源码中明确进行了权限验证，而你的应用又无法获得这个权限的话，建议就不要浪费时间反射了。**

```
W/System.err: java.lang.reflect.InvocationTargetException
W/System.err:     at java.lang.reflect.Method.invoke(Native Method)
W/System.err:     at .ReflectClass.shutDown(ReflectClass.java:104)
W/System.err:     at .MainActivity$1.onClick(MainActivity.java:25)
W/System.err:     at android.view.View.performClick(View.java:6259)
W/System.err:     at
android.view.View$PerformClick.run(View.java:24732)
W/System.err:     at
android.os.Handler.handleCallback(Handler.java:789)
W/System.err:     at
android.os.Handler.dispatchMessage(Handler.java:98)
W/System.err:     at android.os.Looper.loop(Looper.java:164)
W/System.err:     at
android.app.ActivityThread.main(ActivityThread.java:6592)
W/System.err:     at java.lang.reflect.Method.invoke(Native Method)
W/System.err:     at
com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:24
0)
W/System.err:     at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:769)
W/System.err: Caused by: java.lang.SecurityException: Neither user
10224 nor current process has android.permission.REBOOT.
W/System.err:     at
android.os.Parcel.readException(Parcel.java:1942)
W/System.err:     at
android.os.Parcel.readException(Parcel.java:1888)
W/System.err:     at
android.os.IPowerManager$Stub$Proxy.shutdown(IPowerManager.java:787)
W/System.err:  ... 12 more
```

10. 多线程和线程池

（1）无论 synchronized 关键字加在方法上还是对象上，他取得的锁都是对象，而不是把一段代码或函数当作锁——而且同步方法很可能还会被其他线程的

对象访问。

（2）每个对象只有一个锁（lock）和之相关联。

（3）实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

说下 java 中的线程创建方式，线程池的工作原理。

java 中有三种创建线程的方式，或者说四种

1.继承 Thread 类实现多线程

2.实现 Runnable 接口

3.实现 Callable 接口

4.通过线程池

线程池的工作原理：线程池可以减少创建和销毁线程的次数，从而减少系统资源的消耗，当一个任务提交到线程池时

a. 首先判断核心线程池中的线程是否已经满了，如果没满，则创建一个核心线程执行任务，否则进入下一步

b. 判断工作队列是否已满，没有满则加入工作队列，否则执行下一步

c. 判断线程数是否达到了最大值，如果不是，则创建非核心线程执行任务，否则执行饱和策略，默认抛出异常

11.HTTP、HTTPS、TCP/IP、Socket 通信、三次握手四次挥手过程

- 1.ARP 协议:在 IP 以太网中，当一个上层协议要发包时，有了该节点的 IP 地址，ARP 就能提供该节点的 MAC 地址。
- 2.HTTP HTTPS 的区别:
    - 1.HTTPS 使用 TLS(SSL)进行加密
    - 2.HTTPS 缺省工作在 TCP 协议 443 端口
    - 3.它的工作流程一般如以下方式:

- 1.完成 TCP 三次同步握手
- 2.客户端验证服务器数字证书，通过，进入步骤 3
- 3.DH 算法协商对称加密算法的密钥、hash 算法的密钥
- 4.SSL 安全加密隧道协商完成
- 5.网页以加密的方式传输，用协商的对称加密算法和密钥加密，保证数据机密性；用协商的 hash 算法进行数据完整性保护，保证数据不被篡改

- 3.http 请求包结构，http 返回码的分类，400 和 500 的区别

  - 1.包结构：

    - 1.请求：请求行、头部、数据
    - 2.返回：状态行、头部、数据
  - 2.http 返回码分类：1 到 5 分别是，消息、成功、重定向、客户端错误、服务端错误

- 4.Tcp

  - 1.可靠连接，三次握手，四次挥手

    - 1.三次握手：防止了服务器端的一直等待而浪费资源，例如只是两次握手，如果 s 确认之后 c 就掉线了，那么 s 就会浪费资源
      - 1.syn-c = x，表示这消息是 x 序号
      - 2.ack-s = x + 1，表示 syn-c 这个消息接收成功。syn-s = y，表示这消息是 y 序号。
      - 3.ack-c = y + 1，表示 syn-s 这条消息接收成功
  - 2.四次挥手：TCP 是全双工模式

    - 1.fin-c = x，表示现在需要关闭 c 到 s 了。ack-c = y,表示上一条 s 的消息已经接收完毕
    - 2.ack-s = x + 1，表示需要关闭的 fin-c 消息已经接收到了，同意关闭
    - 3.fin-s = y + 1，表示 s 已经准备好关闭了，就等 c 的最后一条命令
    - 4.ack-c = y + 1，表示 c 已经关闭，让 s 也关闭
  - 3.滑动窗口，停止等待、后退 N、选择重传
  - 4.拥塞控制，慢启动、拥塞避免、加速递减、快重传快恢复

TCP 协议与 UDP 协议的区别

TCP（Transmission Control Protocol，传输控制协议）是面向连接的协议，也就是说，在收发数据前，必须和对方建立可靠的连接。一个

TCP 连接必须要经过三次“对话”才能建立起来 ,其中的过程非常复杂，只简单的描述下这三次对话的简单过程：主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，可以吗？”，这是第一次对话；主机 B 向主机 A 发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你什么时候发？”，这是第二次对话；主机 A 再发出一个数据包确认主机 B 的要求同步：“我现在就发，你接着吧！”，这是第三次对话。三次“对话”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机 A 才向主机 B 正式发送数据。详细点说就是：

TCP 三次握手过程

1 主机 A 通过向主机 B 发送一个含有同步序列号的标志位的数据段给主机 B ,向主机 B 请求建立连接,通过这个数据段,

主机 A 告诉主机 B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我.

2 主机 B 收到主机 A 的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机 A,也告诉主机 A 两件事:

我已经收到你的请求了,你可以传输数据了;你要用哪伟序列号作为起始数据段来回应我

3 主机 A 收到这个数据段后,再发送一个确认应答,确认已收到主机 B 的数据段:"我已收到回复,我现在要开始传输实际数据了

这样 3 次握手就完成了,主机 A 和主机 B 就可以传输数据了.

3 次握手的特点

没有应用层的数据

SYN 这个标志位只有在 TCP 建产连接时才会被置 1

握手完成后 SYN 标志位被置 0

TCP 建立连接要进行 3 次握手,而断开连接要进行 4 次

1 当主机 A 完成数据传输后,将控制位 FIN 置 1,提出停止 TCP 连接的请求

2 主机 B 收到 FIN 后对其作出响应,确认这一方向上的 TCP 连接将关闭,将 ACK 置 1

3 由 B 端再提出反方向的关闭请求,将 FIN 置 1

4 主机 A 对主机 B 的请求进行确认,将 ACK 置 1,双方向的关闭结束.

由 TCP 的三次握手和四次断开可以看出,TCP 使用面向连接的通信方式,大大提高了数据通信的可靠性,使发送数据端

和接收端在数据正式传输前就有了交互,为数据正式传输打下了可靠的基础

名词解释

ACK TCP 报头的控制位之一,对数据进行确认.确认由目的端发出,用它

来告诉发送端这个序列号之前的数据段

都收到了.比如,确认号为 X,则表示前 X-1 个数据段都收到了,只有当 ACK=1 时,确认号才有效,当 ACK=0 时,确认号无效,这时会要求重传数据,保证数据的完整性.

SYN 同步序列号,TCP 建立连接时将这个位置 1

FIN 发送端完成发送任务位,当 TCP 完成数据传输需要断开时,提出断开连接的一方将这位置 1

UDP（User Data Protocol，用户数据报协议）

（1） UDP 是一个非连接的协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP 传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；在接收端，UDP 把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

（2） 由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息。

（3） UDP 信息包的标题很短，只有 8 个字节，相对于 TCP 的 20 个字节信息包的额外开销很小。

（4）吞吐量不受拥挤控制算法的调节 ,只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。

（5）UDP 使用**尽最大努力交付，**即不保证可靠交付，因此主机不需要维持复杂的链接状态表（这里面有许多参数）。

（6）UDP 是**面向报文**的。发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付给 IP 层。既不拆分，也不合并，而是保留这些报文的边界，因此，应用程序需要选择合适的报文大小。

小结 TCP 与 UDP 的区别：

1.基于连接与无连接；

2.对系统资源的要求（TCP 较多，UDP 少）；

3.UDP 程序结构较简单；

4.流模式与数据报模式 ；

5.TCP 保证数据正确性，UDP 可能丢包，TCP 保证数据顺序，UDP 不保证。

以下内容来自百度百科：

第一次握手：建立连接时，客户端发送 syn 包（syn=j）到服务器，并进入 SYN_SENT 状态，等待服务器确认；SYN：同步序列编号（***Synchronize Sequence Numbers***）。

第二次握手 :服务器收到 syn 包 ,必须确认客户的 SYN(ack=j+1)，同时自己也发送一个 SYN 包（ syn=k ），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK(ack=k+1），此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。

完成三次握手，客户端与服务器开始传送数据，在上述过程中，还有一些重要的概念：

未连接队列

在三次握手协议中，服务器维护一个未连接队列，该队列为每个客户端的 SYN 包（syn=j）开设一个条目，该条目表明服务器已收到 SYN 包，并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于 SYN_RECV 状态，当服务器收到客户的确认包时，删除该条目，服务器进入 ESTABLISHED 状态。

关闭 TCP 连接：改进的三次握手

对于一个已经建立的连接，TCP 使用改进的三次握手来释放连接（使用一个带有 FIN 附加标记的报文段）。TCP 关闭连接的步骤如下：

第一步，当主机 A 的应用程序通知 TCP 数据已经发送完毕时，TCP 向主机 B 发送一个带有 FIN 附加标记的报文段（FIN 表示英文 finish）。

第二步，主机 B 收到这个 FIN 报文段之后，并不立即用 FIN 报文段回复主机 A，而是先向主机 A 发送一个确认序号 ACK，同时通知自己相应的应用程序：对方要求关闭连接（先发送 ACK 的目的是为了防止在这段时间内，对方重传 FIN 报文段）。

第三步，主机 B 的应用程序告诉 TCP：我要彻底的关闭连接，TCP 向主机 A 送一个 FIN 报文段。

第四步，主机 A 收到这个 FIN 报文段后，向主机 B 发送一个 ACK 表示连接彻底释放。

12. 设计模式（六大基本原则、项目中常用的设计模式、手写单例等）

1.单一职责原则:不要存在多于一个导致类变更的原因。

通俗的说:即一个类只负责一项职责。

2.里氏替换原则:所有引用基类的地方必须能透明地使用其子类

的对象。

通俗的说:当使用继承时。类 B 继承类 A 时，除添加新的方法完成新增功能 P2 外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。子类可以扩展父类的功能，但不能改变父类原有的功能。

3.依赖倒置原则:高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

通俗的说:在 java 中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的 ,是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。依赖倒置原则的核心思想是**面向接口编程.**

4.接口隔离原则:客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

通俗的说:建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

5.迪米特法则:一个对象应该对其他对象保持最少的了解

通俗的说:尽量降低类与类之间的耦合。

6.开闭原则:一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

通俗的说:**用抽象构建框架，用实现扩展细节。**因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。

1、你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。
总体来说设计模式分为三大类：
创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2、单例设计模式

最好理解的一种设计模式，分为懒汉式和饿汉式。

- 饿汉式：

```
public class Singleton {
    // 直接创建对象
    public static Singleton instance = new Singleton();

    // 私有化构造函数
    private Singleton() {
    }

    // 返回对象实例
    public static Singleton getInstance() {
        return instance;
    }
}
```

- 懒汉式：

```
public class Singleton {
    // 声明变量
    private static volatile Singleton singleton2 = null;

    // 私有构造函数
    private Singleton2() {
    }

    // 提供对外方法
    public static Singleton2 getInstance() {
        if (singleton2 == null) {
            synchronized (Singleton2.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
```

3、工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。

- 工厂方法模式

工厂方法模式分为三种：普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。
多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。
静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

- 普通工厂模式

```
public interface Sender {
    public void Send();
}
public class MailSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is mail sender!");
    }
}
public class SmsSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
public class SendFactory {
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}
```

多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
public class SendFactory {
    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}

public class FactoryTest {
    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.send();
    }
}
```

- 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {
    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}


public class FactoryTest {
    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.send();
    }
}
```

- 抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要

拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

```
public interface Provider {
    public Sender produce();
}
---------------------------------------------------------------------------
public interface Sender {
    public void send();
}
---------------------------------------------------------------------------
public class MailSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is mail sender!");
    }
}
---------------------------------------------------------------------------
public class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is sms sender!");
    }
}
---------------------------------------------------------------------------
public class SendSmsFactory implements Provider {

    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
```

```
public class SendMailFactory implements Provider {

    @Override
    public Sender produce() {
        return new MailSender();
    }
}
------------------------------------------------------------------------------
public class Test {
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.send();
    }
}
```

4、建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 Test 结合起来得到的。

```
public class Builder {
    private List<Sender> list = new ArrayList<Sender>();

    public void produceMailSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new MailSender());
        }
    }

    public void produceSmsSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new SmsSender());
        }
    }
}
```

```
public class TestBuilder {
    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.produceMailSender(10);
    }
}
```

5、适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

- 类的适配器模式

```
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}
-----------------------------------------------------------------
public interface Targetable {
    /* 与原类中的方法相同 */
    public void method1();
    /* 新类的方法 */
    public void method2();
}
public class Adapter extends Source implements Targetable {
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}
public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```

对象的适配器模式

基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题。

```
public class Wrapper implements Targetable {
    private Source source;

    public Wrapper(Source source) {
        super();
        this.source = source;
    }

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}
----------------------------------------------------------------
public class AdapterTest {

        public static void main(String[] args) {
            Source source = new Source();
            Targetable target = new Wrapper(source);
            target.method1();
            target.method2();
        }
    }
```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

6、装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

```
public interface Sourceable {
    public void method();
}
-----------------------------------------------------
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
-----------------------------------------------------
public class Decorator implements Sourceable {
    private Sourceable source;
    public Decorator(Sourceable source) {
        super();
        this.source = source;
    }

    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}
-----------------------------------------------------
public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

7、策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，

且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

```
public interface ICalculator {
    public int calculate(String exp);
}
-------------------------------------------------------------
public class Minus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "-");
        return arrayInt[0] - arrayInt[1];
    }
}
-------------------------------------------------------------
public class Plus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "\\+");
        return arrayInt[0] + arrayInt[1];
    }
}
-------------------------------------------------------------
public class AbstractCalculator {
    public int[] split(String exp, String opt) {
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}
```

```
public class StrategyTest {
    public static void main(String[] args) {
        String exp = "2+8";
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}
```

8、观察者模式（Observer）

观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

```
public interface Observer {
    public void update();
}

public class Observer1 implements Observer {
    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}

public class Observer2 implements Observer {
    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }
}

public interface Subject {
     /*增加观察者*/
    public void add(Observer observer);

    /*删除观察者*/
    public void del(Observer observer);
```

```
    /*通知所有的观察者*/
    public void notifyObservers();

    /*自身的操作*/
    public void operation();
}

public abstract class AbstractSubject implements Subject {

    private Vector<Observer> vector = new Vector<Observer>();

    @Override
    public void add(Observer observer) {
        vector.add(observer);
    }

    @Override
    public void del(Observer observer) {
        vector.remove(observer);
    }

    @Override
    public void notifyObservers() {
        Enumeration<Observer> enumo = vector.elements();
        while (enumo.hasMoreElements()) {
            enumo.nextElement().update();
        }
    }
}

public class MySubject extends AbstractSubject {

    @Override
    public void operation() {
        System.out.println("update self!");
         notifyObservers();
    }
}

public class ObserverTest {
    public static void main(String[] args) {
        Subject sub = new MySubject();
        sub.add(new Observer1());
```

```
        sub.add(new Observer2());
        sub.operation();
    }
}
```

13. 断点续传

原理解析
在开发当中，“断点续传”这种功能很实用和常见，听上去也是比较有“逼格”的感觉。所以通常我们都有兴趣去研究研究这种功能是如何实现的？
以 Java 来说，网络上也能找到不少关于实现类似功能的资料。但是呢，大多数都是举个 Demo 然后贴出源码，真正对其实现原理有详细的说明很少。
于是我们在最初接触的时候，很可能就是直接 Crtl + C/V 代码，然后捣鼓捣鼓，然而最终也能把效果弄出来。但初学时这样做其实很显然是有好有坏的。
好处在于，源码很多，解释很少；如果我们肯下功夫，针对于别人贴出的代码里那些自己不明白的东西去查资料，去钻研。最终多半会收获颇丰。
坏处也很明显：作为初学者，面对一大堆的源码，感觉好多东西都很陌生，就很容易望而生畏。即使最终大致了解了用法，但也不一定明白实现原理。

我们今天就一起从最基本的角度切入，来看看所谓的“断点续传”这个东西是不是真的如此“高逼格”。
其实在接触一件新的“事物”的时候，将它拟化成一些我们本身比较熟悉的事物，来参照和对比着学习。通常会事半功倍。
如果我们刚接触“断点续传”这个概念，肯定很难说清楚个一二三。那么，“玩游戏”我们肯定不会陌生。

OK，那就假设我们现在有一款“通关制的 RPG 游戏”。想想我们在玩这类游戏时通常会怎么做？
很明显，第一天我们浴血奋战，大杀四方，假设终于来到了第四关。虽然激战正酣，但一看墙上的时钟，已经凌晨 12 点，该睡觉了。
这个时候就很尴尬了，为了能够在下一次玩的时候，顺利接轨上我们本次游戏的进度，我们应该怎么办呢？
很简单，我们不关掉游戏，直接去睡觉，第二天再接着玩呗。这样是可以，但似乎总觉着有哪里让人不爽。
那么，这个时候，如果这个游戏有一个功能叫做“存档”，就很关键了。我们直接选择存档，输入存档名“第四关”，然后就可以关闭游戏了。

等到下次进行游戏时，我们直接找到“第四关”这个存档，然后进行读档，就可以接着进行游戏了。

这个时候，所谓的“断点续传”就很好理解了。我们顺着我们之前“玩游戏”的思路来理一下：
假设，现在有一个文件需要我们进行下载，当我们下载了一部分的时候，出现情况了，比如：电脑死机、没电、网络中断等等。
其实这就好比我们之前玩游戏玩着玩着，突然 12 点需要去睡觉休息了是一个道理。OK，那么这个时候的情况是：

如果游戏不能存档，那么则意味着我们下次游戏的时候，这次已经通过的 4 关的进度将会丢失，无法接档。
对应的，如果“下载”的行为无法记录本次下载的一个进度。那么，当我们再次下载这个文件也就只能从头来过。
话到这里，其实我们已经发现了，对于我们以上所说的行为，关键就在于一个字“续”!
而我们要实现让一种断开的行为“续”起来的目的，关键就在于要有“介质”能够记录和读取行为出现”中断”的这个节点的信息。

转化到编程世界
实际上这就是“断点续传”最最基础的原理，用大白话说就是：我们要在下载行为出现中断的时候，记录下中断的位置信息，然后在下次行为中读取。
有了这个位置信息之后，想想我们该怎么做。是的，很简单，在新的下载行为开始的时候，直接从记录的这个位置开始下载内容，而不再从头开始。
好吧，我们用大白话掰扯了这么久的原理，开始觉得无聊了。那么我们现在最后总结一下，然后就来看看我们应该怎么把原理转换到编程世界中去。

当“上传(下载)的行为”出现中断，我们需要记录本次上传(下载)的位置(position)。
当“续”这一行为开始，我们直接跳转到 postion 处继续上传(下载)的行为。
显然问题的关键就在于所谓的“position”，以我们举的“通关游戏来说”，可以用“第几关”来作为这个 position 的单位。
那么转换到所谓的“断点续传”，我们该使用什么来衡量“position”呢？很显然，回归二进制，因为这里的本质无非就是文件的读写。

那么剩下的工作就很简单了，先是记录 position，这似乎都没什么值得说的，因为只是数据的持久化而已(内存，文件，数据库)，我们有很多方式。

另一个关键在于当“续传”的行为开始，我们需要需要从上次记录的 position 位置开始读写操作，所以我们需要一个类似于“指针”功能的东西。
我们当然也可以自己想办法去实现这样一个“指针”，但高兴地是，Java 已经为我们提供了这样的一个类，那就是 RandomAccessFile。
这个类的功能从名字就很直观的体现了，能够随机的去访问文件。我们看一下 API 文档中对该类的说明：

此类的实例支持对随机访问文件的读取和写入。随机访问文件的行为类似存储在文件系统中

的一个大型 byte 数组。

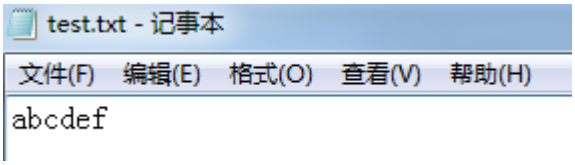
如果随机访问文件以读取/写入模式创建，则输出操作也可用；输出操作从文件指针开始写入字节，并随着对字节的写入而前移此文件指针。

写入隐含数组的当前末尾之后的输出操作导致该数组扩展。该文件指针可以通过 getFilePointer 方法读取，并通过 seek 方法设置。

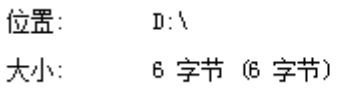
看完 API 说明，我们笑了，是的，这不正是我们要的吗？那好吧，我们磨刀磨了这么久了，还不去砍砍柴吗？

实例演示
既然是针对于文件的“断点续传”，那么很明显，我们先搞一个文件出来。也许音频文件，图像文件什么的看上去会更上档次一点。
但我们已经说了，在计算机大兄弟眼中，它们最终都将回归“二进制”。所以我们这里就创建一个简单的”txt”文件，因为 txt 更利于理解。

我们在 D 盘的根目录下创建一个名为”test.txt”的文件，文件内容很简单，如图所示：



没错，我们输入的内容就是简单的 6 个英语字母。然后我们右键→属性：



我们看到，文件现在的大小是 6 个字节。这也就是为什么我们说，所有的东西到最后还是离不开“二进制”。
是的，我们都明白，因为我们输入了 6 个英文字母，而 1 个英文字母将占据的存储空间是 1 个字节(即 8 个比特位)。
目前为止，我们看到的都很无聊，因为这基本等于废话，稍微有计算机常识的人都知道这些知识。别着急，我们继续。

在 Java 中对一个文件进行读写操作很简单。假设现在的需求如果是“把 D 盘的这个文件写入到 E 盘”，那么我们会提起键盘，啪啪啪啪，搞定！
但其实所谓的文件的“上传(下载)”不是也没什么不同吗？区别就仅仅在于行为由“仅仅在本机之间”转变成了”本机与服务器之间”的文件读写。
这时我们会说，“别逼逼了，这些谁都知道，‘断点续传’呢？“，其实到了这里也已经很简单了，我们再次明确，断点续传要做的无非就是：
前一次读写行为如果出现中断，请记录下此次读写完成的文件内容的位置信息；当“续传开始”则直接将指针移到此处，开始继续读写操作。

反复的强调原理，实际上是因为只要弄明白了原理，剩下的就只是招式而已了。这就就像武侠小说里的“九九归一”大法一样，最高境界就是回归本源。
任何复杂的事物，只要明白其原理，我们就能将其剥离，还原为一个个简单的事物。同理，一系列简单的事物，经过逻辑组合，就形成了复杂的事物。

下面，我们马上就将回归混沌，以最基本的形式模拟一次“断点续传”。在这里我们连服务器的代码都不去写了，直接通过一个本地测试类搞定。
我们要实现的效果很简单：将在 D 盘的”test.txt”文件写入到 E 盘当中，但中途我们会模拟一次”中断”行为，然后在重新继续上传，最终完成整个过程。
也就是说，我们这里将会把“D 盘”视作一台电脑，并且直接将”E 盘”视作一台服务器。那么这样我们甚至都不再与 http 协议扯上半毛钱关系了，（当然实际开发我们肯定是还是得与它扯上关系的 ^<^)，从而只关心最基本的文件读写的”断”和”续”的原理是怎么样的。

为了通过对比加深理解，我们先来写一段正常的代码，即正常读写，不发生中断：

```
public class Test {

    public static void main(String[] args) {
        // 源文件与目标文件
        File sourceFile = new File("D:/", "test.txt");
        File targetFile = new File("E:/", "test.txt");
        // 输入输出流
        FileInputStream fis = null;
        FileOutputStream fos = null;
        // 数据缓冲区
        byte[] buf = new byte[1];

        try {
            fis = new FileInputStream(sourceFile);
            fos = new FileOutputStream(targetFile);
            // 数据读写
            while (fis.read(buf) != -1) {
                System.out.println("write data...");
                fos.write(buf);
            }
        } catch (FileNotFoundException e) {
            System.out.println("指定文件不存在");
        } catch (IOException e) {
            // TODO: handle exception
        } finally {
            try {
                // 关闭输入输出流
                if (fis != null)
                    fis.close();
```

```
                if (fos != null)
                    fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
    }
}
```

该段代码运行，我们就会发现在 E 盘中已经成功拷贝了一份“test.txt”。这段代码很简单，唯一稍微说一下就是:
我们看到我们将 buf，即缓冲区 设置的大小是 1，这其实就代表我们每次 read，是读取一个字节的数据(即 1 个英文字母)。

现在，我们就来模拟这个读写中断的行为，我们将之前的代码完善如下：

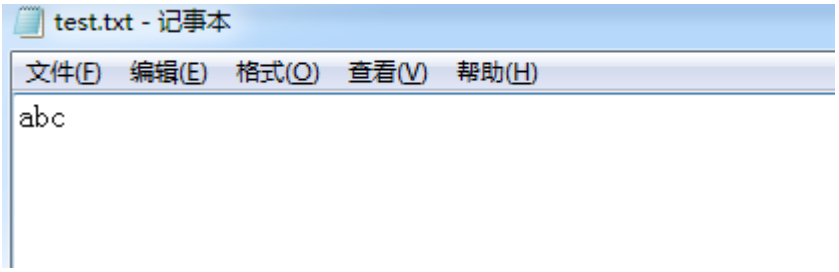
```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.RandomAccessFile;

public class Test {

    private static int position = -1;

    public static void main(String[] args) {
        // 源文件与目标文件
        File sourceFile = new File("D:/", "test.txt");
        File targetFile = new File("E:/", "test.txt");
        // 输入输出流
        FileInputStream fis = null;
        FileOutputStream fos = null;
        // 数据缓冲区
        byte[] buf = new byte[1];

        try {
            fis = new FileInputStream(sourceFile);
            fos = new FileOutputStream(targetFile);
            // 数据读写
            while (fis.read(buf) != -1) {
                fos.write(buf);
```

```
                    // 当已经上传了 3 字节的文件内容时，网络中断了，抛出异常
                    if (targetFile.length() == 3) {
                        position = 3;
                        throw new FileAccessException();
                    }
                }
            } catch (FileAccessException e) {
                keepGoing(sourceFile,targetFile, position);
            } catch (FileNotFoundException e) {
                System.out.println("指定文件不存在");
            } catch (IOException e) {
                // TODO: handle exception
            } finally {
                try {
                    // 关闭输入输出流
                    if (fis != null)
                        fis.close();

                    if (fos != null)
                        fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
        }

        private static void keepGoing(File source,File target, int position) {
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            try {
                RandomAccessFile readFile = new RandomAccessFile(source, "rw");
                RandomAccessFile writeFile = new RandomAccessFile(target, "rw");
                readFile.seek(position);
                writeFile.seek(position);

                // 数据缓冲区
                byte[] buf = new byte[1];
                // 数据读写
```

```
                while (readFile.read(buf) != -1) {
                    writeFile.write(buf);
                }
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

}

class FileAccessException extends Exception {

}
```

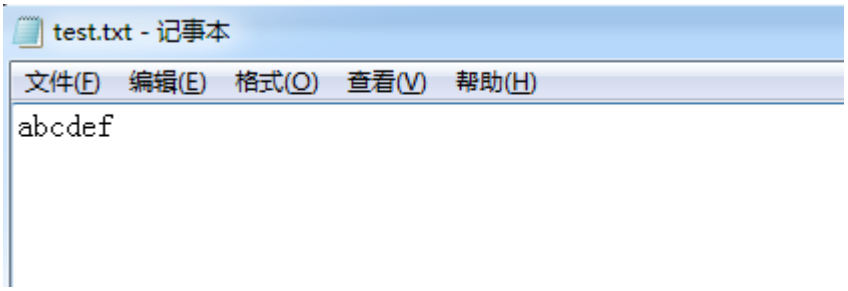
总结一下，我们在这次改动当中都做了什么工作：

首先，我们定义了一个变量 position，记录在发生中断的时候，已完成读写的位置。（这是为了方便，实际来说肯定应该讲这个值存到文件或者数据库等进行持久化）
然后在文件读写的 while 循环中，我们去模拟一个中断行为的发生。这里是当 targetFile 的文件长度为 3 个字节则模拟抛出一个我们自定义的异常。(我们可以想象为实际下载中，已经上传(下载)了”x”个字节的内容，这个时候网络中断了，那么我们就在网络中断抛出的异常中将”x”记录下来)。
剩下的就如果我们之前说的一样，在“续传”行为开始后，通过 RandomAccessFile 类来包装我们的文件，然后通过 seek 将指针指定到之前发生中断的位置进行读写就搞定了。
（实际的文件下载上传，我们当然需要将保存的中断值上传给服务器，这个方式通常为 httpConnection.setRequestProperty(“RANGE”,”bytes=x”);）
在我们这段代码，开启”续传“行为，即 keepGoing 方法中：我们起头让线程休眠 10 秒钟，这正是为了让我们运行程序看到效果。
现在我们运行程序，那么文件就会开启“由 D 盘上传到 E 盘的过程”，我们首先点开 E 盘，会发现的确多了一个 test.txt 文件，打开它发现内容如下：



没错，这个时候我们发现内容只有“abc”。这是在我们预料以内的，因为我们的程序模拟在文件上传了 3 个字节的时候发生了中断。

Ok，我们静静的等待 10 秒钟过去，然后再点开该文件，看看是否能够成功：



通过截图我们发现内容的确已经变成了“abc”，由此也就完成了续传。

14. Java 四大引用

从 JDK1.2 版本开始，把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期。这四种级别由

高到低依次为：强引用、软引用、弱引用和虚引用。

强引用(StrongReference)

我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可

少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序

异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

软引用(SoftReference)

如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器

没有回收它，该对象就可以被程序使用。

弱引用（WeakReference）

在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够

与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱

引用的对象。 弱引用可以和一个引用队列(ReferenceQueue)联合使用，如果弱引用所引用的对象

被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

虚引用(PhantomReference)
　如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。虚 引用主要用来跟

踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列(ReferenceQueue)联合

使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加

入到与之关联的引用队列中。程序可以通过判断引用队列中是 否已经加入了虚引用，来了解被引用的对象是否将要

被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必

要的行动。

下面看两个 Demo

```
public class Demo1 {

    public static void main(String[] args) {

        //这就是一个强引用
        String str="hello";
        //现在我们由上面的强引用创建一个软引用,所以现在 str 有两个引用指向它
        SoftReference<String>  soft=new SoftReference<String>(str);
        str=null;
        //可以使用 get()得到引用指向的对象
        System.out.println(soft.get());//输出 hello

    }
}
```

```
public class Demo2 {
```

```
    public static void main(String[] args) {

        //这就是一个强引用
        String str="hello";
        ReferenceQueue<? super String> q=new ReferenceQueue<String>();
        //现在我们由上面的强引用创建一个虚引用,所以现在 str 有两个引用指向它
        PhantomReference<String>    p=new PhantomReference<String>(str, q);
        //可以使用 get()得到引用指向的对象
        System.out.println(q.poll());//输出 null

    }
}
```

下面再看一个，首先创建一个 Store 类，内部定义一个很大的数组，目的是创建对象时，会得到更多的内存，以提高回收的可能性！

```
public class Store {

    public static final int SIZE = 10000;
    private double[] arr = new double[SIZE];
    private String id;

    public Store() {

    }

    public Store(String id) {
        super();
        this.id = id;
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println(id + "被回收了");
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

```
    @Override
    public String toString() {
        return id;
    }

}
```

依次创建软引用，弱引用，虚引用个 10 个！

```
public class Demo3 {

    public static ReferenceQueue<Store> queue = new ReferenceQueue<Store>();

    public static void checkQueue()
    {
        if(queue!=null)
        {
            @SuppressWarnings("unchecked")
            Reference<Store>    ref =(Reference<Store>)queue.poll();
            if(ref!=null)
                System.out.println(ref+"......"+ref.get());
        }
    }

    public static void main(String[] args) {

        HashSet<SoftReference<Store>> hs1 = new HashSet<SoftReference<Store>>();
        HashSet<WeakReference<Store>> hs2 = new HashSet<WeakReference<Store>>();

        //创建 10 个软引用
        for(int i=1;i<=10;i++)
        {
            SoftReference<Store>        soft        =        new        SoftReference<Store>(new
Store("soft"+i),queue);
            System.out.println("create soft"+soft.get());
            hs1.add(soft);
        }
        System.gc();
        checkQueue();

        //创建 10 个弱引用
        for(int i=1;i<=10;i++)
        {
```

```
                WeakReference<Store> weak = new WeakReference<Store>(new
Store("weak"+i),queue);
                System.out.println("create weak"+weak.get());
                hs2.add(weak);
            }

            System.gc();
            checkQueue();
            //创建 10 个虚引用
            HashSet<PhantomReference<Store>> hs3 = new
HashSet<PhantomReference<Store>>();
            for(int i=1;i<=10;i++)
            {
                PhantomReference<Store> phantom = new PhantomReference<Store>(new
Store("phantom"+i),queue);
                System.out.println("create phantom  "+phantom.get());
                hs3.add(phantom);
            }
            System.gc();
            checkQueue();
        }
}
```

程序执行结果：

```
create softsoft1
create softsoft2
create softsoft3
create softsoft4
create softsoft5
create softsoft6
create softsoft7
create softsoft8
create softsoft9
create softsoft10
create weakweak1
create weakweak2
create weakweak3
create weakweak4
create weakweak5
create weakweak6
create weakweak7
create weakweak8
create weakweak9
create weakweak10
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
create phantom  null
java.lang.ref.WeakReference@96cf11......null
weak10被回收了
phantom10被回收了
phantom9被回收了
phantom8被回收了
phantom7被回收了
```


可以看到虚引用和弱引用被回收掉。。。

15. Java 的泛型

一. 泛型概念的提出（为什么需要泛型）？

首先，我们看下下面这段简短的代码：

```
public class GenericTest {


    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("qqyumidi");
        list.add("corn");
        list.add(100);


        for (int i = 0; i < list.size(); i++) {
            String name = (String) list.get(i); // 1
            System.out.println("name:" + name);
        }
    }
}
```

定义了一个 List 类型的集合，先向其中加入了两个字符串类型的值，随后加入一个 Integer 类型的值。这是完全允许的，因为此时 list 默认的类型为 Object 类型。在之后的循环中，由于忘记了之前在 list 中也加入了 Integer 类型的值或其他编码原因，很容易出现类似于 //1 中的错误。因为编译阶段正常，而运行时会出现“java.lang.ClassCastException”异常。因此，导致此类错误编码过程中不易发现。

在如上的编码过程中，我们发现主要存在两个问题：

1.当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了 Object 类型，但其运行时类型任然为其本身类型。

2.因此，//1 处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“java.lang.ClassCastException”异常。

那么有没有什么办法可以使集合能够记住集合内元素各类型，且能够达到只要编译时不出现问题，运行时就不会出现“java.lang.ClassCastException”异常呢？答案就是使用泛型。

二.什么是泛型？

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

看着好像有点复杂，首先我们看下上面那个例子采用泛型的写法。

```
public class GenericTest {


    public static void main(String[] args) {

        /*

        List list = new ArrayList();

        list.add("qqyumidi");

        list.add("corn");

        list.add(100);

        */


        List<String> list = new ArrayList<String>();

        list.add("qqyumidi");

        list.add("corn");

        //list.add(100);   // 1  提示编译错误


        for (int i = 0; i < list.size(); i++) {

            String name = list.get(i); // 2

            System.out.println("name:" + name);

        }

    }

}
```

采用泛型写法后，在//1 处想加入一个 Integer 类型的对象时会出现编译错误，通过 List<String>，直接限定了 list 集合中只能含有 String 类型的元素，从而在//2 处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是 String 类型了。

结合上面的泛型定义，我们知道在 List<String>中，String 是类型实参，也就是说，相应的 List 接口中肯定含有类型形参。且 get()方法的返回结果也直接是此形参类型（也就是对应的传入的类型实参）。下面就来看看 List 接口的的具体定义：

```
public interface List<E> extends Collection<E> {


    int size();
```

```
    boolean isEmpty();

    boolean contains(Object o);

    Iterator<E> iterator();

    Object[] toArray();

    <T> T[] toArray(T[] a);

    boolean add(E e);

    boolean remove(Object o);

    boolean containsAll(Collection<?> c);

    boolean addAll(Collection<? extends E> c);

    boolean addAll(int index, Collection<? extends E> c);

    boolean removeAll(Collection<?> c);

    boolean retainAll(Collection<?> c);

    void clear();

    boolean equals(Object o);

    int hashCode();
```

```
    E get(int index);

    E set(int index, E element);

    void add(int index, E element);

    E remove(int index);

    int indexOf(Object o);

    int lastIndexOf(Object o);

    ListIterator<E> listIterator();

    ListIterator<E> listIterator(int index);

    List<E> subList(int fromIndex, int toIndex);
}
```

我们可以看到，在 List 接口中采用泛型化定义之后，<E>中的 E 表示类型形参，可以接收具体的类型实参，并且此接口定义中，凡是出现 E 的地方均表示相同的接受自外部的类型实参。

自然的，ArrayList 作为 List 接口的实现类，其定义形式是：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

```

```
    public E get(int index) {
        rangeCheck(index);
        checkForComodification();
        return ArrayList.this.elementData(offset + index);
    }

    //...省略掉其他具体的定义过程

}
```

由此，我们从源代码角度明白了为什么//1 处加入 Integer 类型对象编译错误，且//2 处 get()到的类型直接就是 String 类型了。

三.自定义泛型接口、泛型类和泛型方法

从上面的内容中，大家已经明白了泛型的具体运作过程。也知道了接口、类和方法也都可以使用泛型去定义，以及相应的使用。是的，在具体使用时，可以分为泛型接口、泛型类和泛型方法。

自定义泛型接口、泛型类和泛型方法与上述 Java 源码中的 List、ArrayList 类似。如下，我们看一个最简单的泛型类和方法定义：

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        System.out.println("name:" + name.getData());
    }

}

class Box<T> {

    private T data;
```

```
    public Box() {

    }

    public Box(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }

}
```

在泛型接口、泛型类和泛型方法的定义过程中，我们常见的如 T、E、K、V 等形式的参数常用于表示泛型形参，由于接收来自外部使用时候传入的类型实参。**那么对于不同传入的类型实参，生成的相应对象实例的类型是不是一样的呢？**

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);

        System.out.println("name class:" + name.getClass());      // com.qqyumidi.Box
        System.out.println("age class:" + age.getClass());        // com.qqyumidi.Box
        System.out.println(name.getClass() == age.getClass());    // true

    }
```

```
}
```

由此，我们发现，在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为 Box），当然，在逻辑上我们可以理解成多个不同的泛型类型。

究其原因，在于 Java 中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，也就是说，成功编译过后的 class 文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

四.类型通配符

接着上面的结论，我们知道，Box<Number>和 Box<Integer>实际上都是 Box 类型，现在需要继续探讨一个问题，那么在逻辑上，类似于 Box<Number>和 Box<Integer>是否可以看成具有父子关系的泛型类型呢？

为了弄清这个问题，我们继续看下下面这个例子：

```
public class GenericTest {


    public static void main(String[] args) {


        Box<Number> name = new Box<Number>(99);

        Box<Integer> age = new Box<Integer>(712);


        getData(name);

        //The method getData(Box<Number>) in the type GenericTest is

        //not applicable for the arguments (Box<Integer>)

        getData(age);   // 1


    }

    public static void getData(Box<Number> data){
```

```
        System.out.println("data :" + data.getData());

    }


}
```

我们发现，在代码//1 处出现了错误提示信息：The method getData(Box<Number>) in the t ype GenericTest is not applicable for the arguments (Box<Integer>)。显然，通过提示信息，我们知道 Box<Number>在逻辑上不能视为 Box<Integer>的父类。那么，原因何在呢？

```
public class GenericTest {


    public static void main(String[] args) {


        Box<Integer> a = new Box<Integer>(712);

        Box<Number> b = a;  // 1

        Box<Float> f = new Box<Float>(3.14f);

        b.setData(f);        // 2


    }


    public static void getData(Box<Number> data) {

        System.out.println("data :" + data.getData());

    }


}


class Box<T> {


    private T data;


    public Box() {
```

```
    }

    public Box(T data) {
        setData(data);
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

}
```

这个例子中，显然//1 和//2 处肯定会出现错误提示的。在此我们可以使用反证法来进行说明。

假设 Box<Number>在逻辑上可以视为 Box<Integer>的父类，那么//1 和//2 处将不会有错误提示了，那么问题就出来了，通过 getData()方法取出数据时到底是什么类型呢？Integer? Float? 还是 Number？且由于在编程过程中的顺序不可控性，导致在必要的时候必须要进行类型判断，且进行强制类型转换。显然，这与泛型的理念矛盾，因此，**在逻辑上 Box<Number>不能视为 Box<Integer>的父类。**

好，那我们回过头来继续看"类型通配符"中的第一个例子，我们知道其具体的错误提示的深层次原因了。那么如何解决呢？总部能再定义一个新的函数吧。这和 Java 中的多态理念显然是违背的，因此，我们需要一个在逻辑上可以用来表示同时是 Box<Integer>和 Box<Number>的父类的一个引用类型，由此，类型通配符应运而生。

类型通配符一般是使用 ？代替具体的类型实参。注意了，此处是类型实参，而不是类型形参！且 Box<?>在逻辑上是 Box<Integer>、Box<Number>...等所有 Box<具体类型实参>的父类。由此，我们依然可以定义泛型方法，来完成此类需求。

```
public class GenericTest {

    public static void main(String[] args) {
```

```
        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);
        Box<Number> number = new Box<Number>(314);

        getData(name);
        getData(age);
        getData(number);
    }

    public static void getData(Box<?> data) {
        System.out.println("data :" + data.getData());
    }

}
```

有时候，我们还可能听到**类型通配符上限和类型通配符下限**。具体有是怎么样的呢？

在上面的例子中，如果需要定义一个功能类似于 getData()的方法，但对类型实参又有进一步的限制：只能是 Number 类及其子类。此时，需要用到类型通配符上限。

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);
        Box<Number> number = new Box<Number>(314);

        getData(name);
        getData(age);
        getData(number);

        //getUpperNumberData(name); // 1
```

```
        getUpperNumberData(age);    // 2

        getUpperNumberData(number); // 3

    }


    public static void getData(Box<?> data) {

        System.out.println("data :" + data.getData());

    }


    public static void getUpperNumberData(Box<? extends Number> data){

        System.out.println("data :" + data.getData());

    }


}
```

此时，显然，在代码//1 处调用将出现错误提示，而//2 //3 处调用正常。

16. final、finally、finalize 的区别

1. final

在 java 中，final 可以用来修饰类，方法和变量（成员变量或局部变量）。下面将对其详细介绍。

1.1 修饰类

当用 final 修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用 final 修饰，但要注意：

final 类中所有的成员方法都会隐式的定义为 final 方法。

1.2 修饰方法

使用 final 方法的原因主要有两个：

(1) 把方法锁定，以防止继承类对其进行更改。

(2) 效率，在早期的 java 版本中，会将 final 方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有多大提升。因此在最近版本中，不需要 final 方法进行这些优化了。

final 方法意味着"最后的、最终的"含义，即此方法不能被重写。

注意：若父类中 final 方法的访问权限为 private，将导致子类中不能直接继承该方法，因此，此时可以在子类中定义相同方法名的函数，此时不会与重写 final 的矛盾，而是在子类中重新地定义了新方法。

```
class A{
    private final void getName(){

    }
}
public class B extends A{
    public void getName(){

    }

    public static void main(String[]args){
        System.out.println("OK");
    }
}
```

1.3 修饰变量

final 成员变量表示常量，只能被赋值一次，赋值后其值不再改变。类似于 C++中的 const。

当 final 修饰一个基本数据类型时，表示该基本数据类型的值一旦在初始化后便不能发生变化；如果 final 修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的。本质上是一回事，因为引用的值是一个地址，final 要求值，即地址的值不发生变化。

final 修饰一个成员变量（属性），必须要显示初始化。这里有两种初始化方式，一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

当函数的参数类型声明为 final 时，说明该参数是只读型的。即你可以读取使用该参数，但是无法改变该参数的值。

```
public class Main{
    public static void main(String[] args) {
        //String str=new String("ABC");
        final String str1="Hello KT";
        //final String str2=str;
        System.out.println(str1);
        //System.out.println(str2);
        str1="KT";
        //str="CBA";
        System.out.println(str1);
        //System.out.println(str2);
    }
}
```

```
C++Test
ktao@ktao-PC:~/Desktop/C++Test$ javac test1.java
Picked up _JAVA_OPTIONS:    -Dawt.useSystemAAFontSettings=gasp
test1.java:1: 错误: 类Main是公共的，应在名为 Main.java 的文件中声明
public class Main{
       ^
test1.java:8: 错误: 无法为最终变量str1分配值
                str1="KT";
                ^
2 个错误
```

在 java 中，String 被设计成 final 类，那为什么平时使用时，String 的值可以被改变呢？

字符串常量池是 java 堆内存中一个特殊的存储区域，当我们建立一个 String 对象时，假设常量池不存在该字符串，则创建一个，若存在则直接引用已经存在的字符串。当我们对 String 对象值改变的时候，例如 String a="A"; a="B" 。a 是 String 对象的一个引用（我们这里所说的 String 对象其实是指字符串常量），当 a="B"执行时，并不是原本 String 对象("A")发生改变，而是创建一个新的对象("B")，令 a 引用它。

2. finally

finally 作为异常处理的一部分，它只能用在 try/catch 语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。（×）（这句话其实存在一定的问题）

很多人都认为 finally 语句块一定会执行，但真的是这样么？答案是否定的，例如下面这个例子：

```java
public class Test{
    public static int test(){
        int i=1;
        //if(i==1){
        //  return 0;
        //}
        System.out.println("The previous statement of try block");
        i=i/0;
        try{
            System.out.println("try block");
            return i;
        }
        finally{
            System.out.println("finally block");
        }
    }
    public static void main(String[] args) {
        System.out.println("return value of test(): "+test());
    }
}
```

```
ktao
ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS:   -Dawt.useSystemAAFontSettings=gasp
The previous statement of try block
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.test(Test.java:8)
        at Test.main(Test.java:18)
ktao@ktao-PC:~$
```

当我们去掉注释的三行语句，执行结果为：

```
^[[Aktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS:   -Dawt.useSystemAAFontSettings=gasp
return value of test(): 0
```

为什么在以上两种情况下都没有执行 finally 语句呢，说明什么问题？

只有与 finally 对应的 try 语句块得到执行的情况下，finally 语句块才会执行。以上两种情况在执行 try 语句块之前已经返回或抛出异常，所以 try 对应的 finally 语句并没有执行。

但是，在某些情况下，即使 try 语句执行了，finally 语句也不一定执行。例如以下情况：

```java
public class Test{
    public static int test(){
        int i=1;
        //if(i==1){
        //  return 0;
        //}
        try{
            System.out.println("try block");
            System.exit(0);
            return i;
        }
        finally{
            System.out.println("finally block");
        }
    }
    public static void main(String[] args) {
        System.out.println("return value of test(): "+test());
    }
}
```

```
ktao
ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS:    -Dawt.useSystemAAFontSettings=gasp
try block
ktao@ktao-PC:~$
```

finally 语句块还是没有执行，为什么呢？因为我们在 try 语句块中执行了 System.exit (0) 语句，终止了 Java 虚拟机的运行。那有人说了，在一般的 Java 应用中基本上是不会调用这个 System.exit(0) 方法的。OK ！没有问题，我们不调用 System.exit(0) 这个方法，那么 finally 语句块就一定会执行吗？

再一次让大家失望了，答案还是否定的。当一个线程在执行 try 语句块或者 catch 语句块时被打断（interrupted）或者被终止（killed），与其相对应的 finally 语句块可能不会执行。还有更极端的情况，就是在线程运行 try 语句块或者 catch 语句块时，突然死机或者断电，finally 语句块肯定不会执行了。可能有人认为死机、断电这些理由有些强词夺理，没有关系，我们只是为了说明这个问题。

易错点

在 try-catch-finally 语句中执行 return 语句。我们看如下代码：

```java
public class Test {
    public static void main(String[] args) {
        System.out.println(test(null) + "," + test("0") + "," + test("
    }

    public static int test(String str) {
        try {
            return str.charAt(0) - '0';
        } catch (NullPointerException e1) {

            return 1;
        } catch (StringIndexOutOfBoundsException e2) {

            return 2;
        } catch (Exception e3) {

            return 3;
        } finally {
            return 4;
        }

    }
}
```

```
ktao
ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS:    -Dawt.useSystemAAFontSettings=gasp
4,4,4
ktao@ktao-PC:~$
```

答案：4,4,4 。 为什么呢？

首先 finally 语句在改代码中一定会执行，从运行结果来看，每次 return 的结果都是 4（即 finally 语句），仿佛其他 return 语句被屏蔽掉了。

事实也确实如此，因为 finally 用法特殊，所以会撤销之前的 return 语句，继续执行最后的 finally 块中的代码。

3. finalize

finalize()是在 java.lang.Object 里定义的，也就是说每一个对象都有这么个方法。这个方法在 gc 启动，该对象被回收的时候被调用。其实 gc 可以回收大部分的对象（凡是 new 出来的对象，gc 都能搞定，一般情况下我们又不会用 new 以外的方式去创建对象），所以一般是不需要程序员去实现 finalize 的。特殊情况下，需要程序员实现 finalize，当对象被回收的时候释放一些资源，比如：一个 socket 链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现 finalize，关闭这个链接。

使用 finalize 还需要注意一个事，调用 super.finalize();

一个对象的 finalize()方法只会被调用一次，而且 finalize()被调用不意味着 gc 会立即回收该对象，所以有可能调用 finalize()后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会调用 finalize()，产生问题。 所以，推荐不要使用 finalize()方法，它跟析构函数不一样。

17. 接口、抽象类的区别

抽象类

抽象类是用来捕捉子类的通用特性的 。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。以 JDK 中的 GenericServlet 为例：

```
1
2  public abstract class GenericServlet implements Servlet, ServletConfig, Serializable {
3      // abstract method
4      abstract void service(ServletRequest req, ServletResponse res);
5
6      void init() {
7          // Its implementation
8      }
       // other method related to Servlet
   }
9
```

当 HttpServlet 类继承 GenericServlet 时，它提供了 service 方法的实现：

```
1  public class HttpServlet extends GenericServlet {
2      void service(ServletRequest req, ServletResponse res) {
```

```
3         // implementation
4       }
5
6       protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
7         // Implementation
8       }
9
10      protected void doPost(HttpServletRequest req, HttpServletResponse resp) {
11        // Implementation
12      }
13
14      // some other methods related to HttpServlet
15    }
```

接口

接口是抽象方法的集合。如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。这就像契约模式，如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。以 Externalizable 接口为例：

```
1   public interface Externalizable extends Serializable {
2
```

```
3     void writeExternal(ObjectOutput out) throws IOException;
4
5     void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
6  }
```

当你实现这个接口时，你就需要实现上面的两个方法：

```
1
2  public class Employee implements Externalizable {
3
4     int employeeId;
5     String employeeName;
6
7     @Override
8     public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
9        employeeId = in.readInt();
10       employeeName = (String) in.readObject();
11
12    }
13
14    @Override
15    public void writeExternal(ObjectOutput out) throws IOException {
16
17       out.writeInt(employeeId);
18       out.writeObject(employeeName);
19    }
20 }
```

17

18

19

抽象类和接口的对比

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用**extends**关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字**implements**来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常Java类的区别	除了你不能实例化抽象类之外，它和普通Java类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有**public**、**protected**和**default**这些修饰符	接口方法默认修饰符是**public**。你不可以使用其它修饰符。
main方法	抽象方法可以有main方法并且我们可以运行它	接口没有main方法，因此我们不能运行它。（java8以后接口可以有default和static方法，所以可以运行main方法）
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

什么时候使用抽象类和接口

- 如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。
- 如果你想实现多重继承，那么你必须使用接口。由于 **Java 不支持多继承**，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。
- 如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

什么时候使用抽象类？

抽象类让你可以定义一些默认行为并促使子类提供任意特殊化行为。

例如 :Spring 的依赖注入就使得代码实现了集合框架中的接口原则和抽象实现。

Java8 中的默认方法和静态方法

Oracle 已经开始尝试向接口中引入默认方法和静态方法，以此来减少抽象类和接口之间的差异。现在，我们可以为接口提供默认实现的方法了并且不用强制子类来实现它。这类内容我将在下篇博客进行阐述。

从 java 容器类的设计讨论抽象类和接口的应用

除了前面提到的一个问题 :多态到底是用抽象类还是接口实现？我还看到有人评论说：现在都是提倡面向接口编程，使用抽象类的都被称为上世纪的老码农了。哈哈。看到这个说法我也是苦笑不得。不过，面向接口编程的确是一个趋势，java 8 已经支持接口实现默认方法和静态方法了，抽象类和接口之间的差异越来越小。闲话少说，我们开始讨论抽象类和接口的应用。

full_container_taxonomy

上图是 java 容器类的类继承关系。我们以容器类中 ArrayList 为例子来讨论抽象类和接口的应用。

ArrayList 类继承关系

ArrayList

上图是 ArrayList 的类继承关系。可以看到，ArrayList 的继承关系中既使用了抽象类，也使用了接口。

- 最顶层的接口是 Iterable，表示这是可迭代的类型。所有容器类都是可迭代的，这是一个极高的抽象。
- 第二层的接口是 Collection，这是单一元素容器的接口。集合，列表都属于此类。
- 第三层的接口是 List，这是所有列表的接口。

通过三个接口，我们可以找到容器类的三个抽象特性，实现这些接口就意味着拥有这些接口的特性。

- AbstractCollection 实现了 Collection 中的部分方法。
- AbstractList 实现了 AbstractCollection 和 List 中的部分方法。

上面的抽象类提供了一些方法的默认实现，给具体类提供了复用代码。

纯抽象类实现

如果我们像一个老码农一样，用抽象类来实现上面的接口会有怎样的效果？那么，类图可能变成这样。

AbtractList

抽象类在这里存在着一个很大的问题，它不能多继承，在抽象的层次上没有接口高，也没有接口灵活。例如说：

List + AbstractCollection -> AbstractList

Set + AbstractCollection -> AbstractSet

单纯用抽象类无法实现像接口一样灵活的扩展。

纯接口实现

如果我们像一个新码农一样，用纯接口来实现呢？

InterfaceList

这样写理论上没有问题，实际写代码的时候问题就来了。所有的接口都要提供实现，于是你不得不在各个实现类中重复代码。

总结

经过上面的讨论，我们得出两个结论：

- 抽象类和接口并不能互相替代。
- 抽象类和接口各有不可替代的作用。

从容器类的类关系图中可以看到，接口主要是用来抽象类型的共性，例如说，容器的可迭代特性。抽象类主要是给具体实现类提供重用的代码，例如说，List 的一些默认方法。

抽象类和接口的使用时机

那么，什么时候该用抽象类，什么时候该用接口呢？

要解决上面的问题，我们先从弄清楚抽象类和接口之间的关系。首先，我们都知道类对事物的抽象，定义了事物的属性和行为。而抽象类是不完全的类，具有抽象方法。接口则比类的抽象层次更高。所以，我们可以这样理解它们之间的关系：**类是对事物的抽象，抽象类是对类的抽象，接口是对抽象类的抽象。**

从这个角度来看 java 容器类，你会发现，它的设计正体现了这种关系。不是吗？从 Iterable 接口，到 AbstractList 抽象类，再到 ArrayList 类。

现在回答前面的问题：在设计类的时候，首先考虑用接口抽象出类的特性，当你发现某些方法可以复用的时候，可以使用抽象类来复用代码。简单说，**接口用于抽象事物的特性，抽象类用于代码复用。**

当然，不是所有类的设计都要从接口到抽象类，再到类。程序设计本就没有绝对的范式可以遵循。上面的说法只是提供一个角度来理解抽象类和接口的关系，每个人都会有自己的理解，有人认为两者一点关系都没有，这也有道理。总之，模式和语法是死的，人是活的。

第二章 Android d 面试题解析大全

1.自定义 View

1 良好的自定义 View

易用，标准，开放。
一个设计良好的自定义 view 和其他设计良好的类很像。封装了某个具有易用性接口的功能组合，这些功能能够有效地使用 CPU 和内存，并且十分开放的。但是，除了开始一个设计良好的类之外，一个自定义 view 应该：
l 符合安卓标准
l 提供能够在 Android XML 布局中工作的自定义样式属性
l 发送可访问的事件
l 与多个 Android 平台兼容。
Android 框架提供了一套基本的类和 XML 标签来帮您创建一个新的，满足这些要求的 view。忘记提供属性和事件是很容易的，尤其是当您是这个自定义 view 的唯一用户时。请花一些时间来仔细的定义您 view 的接口以减少未来维护时所耗费的时间。一个应该遵从的准则是：暴露您 view 中所有影响可见外观的属性或者行为。

2 创建自定义 View（步骤）

2.1 继承 View 完全自定义或继承 View 的派生子类

必须提供一个能够获取 Context 和作为属性的 AttributeSet 对象的构造函数，获取属性，当 view 从 XML 布局中创建了之后，XML 标签中所有的属性都从资源包中读取出来并作为一个 AttributeSet 传递给 view 的构造函数。
View 派生出来的直接或间接子类：ImageView, Button, CheckBox, SurfaceView, TextView, ViewGroup, AbsListView
ViewGourp 派生出来的直接或间接子类：AbsoluteLayout, FrameLayout, RelativeLayout, LinearLayout
所有基类、派生类都是 Android framework 层集成的标准系统类，可直接引用 SDK 中这些系统类及其 API

2.2 定义自定义属性

l 在资源元素<declare-styleable>中为您的 view 定义自定义属性。
在项目组添加<declare-styleable>资源。这些资源通常是放在 res/values/attrs.xm 文件里。如下是 attrs.xml 文件的一个例子：

```
1
<resources>;
<declare-styleable name="PieChart">
<attr name="showText" format="boolean" />
<attr name="labelPosition" format="enum">
<enum name="left" value="0"/>
<enum name="right" value="1"/>
</attr>
</declare-styleable>
```

```
</resources>
```

l 在您的 XML 布局中使用指定属性的值。

布局 XML 文件中可以像内建属性一样使用它们。唯一不同是自定义属性属于不同的命名空间。

http://schemas.android.com/apk/res/[你的自定义 View 所在的包路径]

```
1
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
<com.example.customviews.charting.PieChart
custom:showText="true"
custom:labelPosition="left" />
</LinearLayout>
```

l 在运行时取得属性值。

l 将取回的属性值应用到您的 view 中。

2.3 获取自定义属性

当 view 从 XML 布局中创建了之后，XML 标签中所有的属性都从资源包中读取出来并作为一个 AttributeSet 传递给 view 的构造函数。尽管从 AttributeSet 中直接读取值是可以的，但是这样做有一些缺点：

l 带有值的资源引用没有进行处理

l 样式并没有得到允许

取而代之的是，将 AttributeSet 传递给 obtainStyledAttributes()方法。这个方法传回了一个 TypedArray 数组，包含了已经解除引用和样式化的值。

为了时能能够更容易的调用 obtainStyledAttributes()方法，Android 资源编译器做了大量的工作。res 文件夹 中的每个<declare-styleable>资源，生成的 R.java 都定义了一个属性 ID 的数组以及一套定义了指向数组中的每一个属性 的常量。您可以使用预定义的常量从 TypedArry 中读取属性。下例是 PieChart 类是如何读取这些属性的：

```
1
public PieChart(Context ctx, AttributeSet attrs) {
super(ctx, attrs);
TypedArray a = context.getTheme().obtainStyledAttributes(
attrs,
R.styleable.PieChart,
0, 0);
try {
mShowText = a.getBoolean(R.styleable.PieChart_showText, false);
mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);
} finally {
a.recycle();
}
}
```

注意,TypedArry 对象是一个共享的资源，使用完毕必须回收它。

2.4 添加属性和事件

属性是控制 view 的行为和外观的强有力的方式，但是只有 view 在初始化后这些属性才可读。为了提供动态的行为，需要暴露每个自定义属性的一对 getter 和 setter。下面的代码片段显示 PieChart 是如何提供 showText 属性的。

```
1
public boolean isShowText() {
return mShowText;
}
public void setShowText(boolean showText) {
mShowText = showText;
invalidate();
requestLayout();
}
```

注意，setShowText 调用了 invalidate()和 requestLayout()。这些调用关键是为了保证 view 行为是可靠的。你必须在改变这个可能改变外观的属性后废除这个 view，这样系统才知道需要重绘。同样，如果属性的变化可能影响尺寸或者 view 的形状，您需要请求一个新的布局。忘记调用这些方法可能导致难以寻找的 bug。

自定义 view 同样需要支持和重要事件交流的事件监听器。

2.5 自定义绘制（实施）

绘制自定义视图里最重要的一步是重写 onDraw()方法. onDraw()的参数是视图可以用来绘制自己的 Canvas 对象. Canvas 定义用来绘制文本、线条、位图和其他图像单元. 你可以在 onDraw()里使用这些方法创建你的自定义用户界面(UI).

android.graphics 框架把绘图分成了两部分:

l 画什么，由 Canvas 处理

l 怎么画，由 Paint 处理

例如, Canvas 提供画线条的方法，而 Paint 提供定义线条颜色的方法. Canvas 提供画矩形的方法，而 Paint 定义是否用颜色填充矩形或让它为空. 简而言之, Canvas 定义你可以在屏幕上画的形状，而 Paint 为你画的每个形状定义颜色、样式、字体等等.

onDraw()不提供 3d 图形 api 的支持。如果你需要 3d 图形支持，必须继承 SurfaceView 而不是 View，并且通过单独的线程画图。

3 优化

3.1 降低刷新频率

为了提高 view 的运行速度，减少来自于频繁调用的程序的不必要的代码。从 onDraw()

方法开始调用，这会给你带来最好的回报。特别地，在 onDraw()方法中你应该减少冗余代码，冗余代码会带来使你 view 不连贯的垃圾回收。初始化的冗余对象，或者动画之间的，在动画运行时，永远都不会有所贡献。
加之为了使 onDraw()方法更有依赖性，你应该尽可能的不要频繁的调用它。大部分时候调用 onDraw()方法就是调用 invalidate()的结果，所以减少不必要的调用 invalidate()方法。有可能的，调用四种参数不同类型的 invalidate()，而不是调用无参的版本。无参变量需要刷新整个 view，而四种参数类型的变量只需刷新指定部分的 view. 这种高效的调用更加接近需求，也能减少落在矩形屏幕外的不必 要刷新的页面。

3.2 使用硬件加速

作为 Android3.0，Android2D 图表系统可以通过大部分新的 Android 装置自带 GPU（图表处理单元）来增加，对于许多应用程序 来说，GPU 硬件加速度能带来巨大的性能增加，但是对于每一个应用来讲，并不都是正确的选择。Android 框架层更好地为你提供了控制应用程序部分硬件 是否增加的能力。
怎样在你的应用，活动，或者窗体级别中使用加速度类，请查阅 Android 开发者指南中的 Hardware Acceleration 类。注意到在开发者指南中的附加说明，你必须在你的 AndroidManifest.xml 文件中的<uses-sdk android:targetSdkVersion="11"/>中将应用目标 API 设置到 11 或者更高的级别。
一旦你使用硬件加速度类，你可能没有看到性能的增长，手机 GPUs 非常擅长某些任务，例如测量，翻转，和平移位图类的图片。特别地，他们不擅长其他的任务，例如画直线和曲线。为了利用 GPU 加速度类，你应该增加 GPU 擅长的操作数量，和减少 GPU 不擅长的操作数量。

. 1. 事件拦截分发

. 2. 解决过的一些性能问题，在项目中的实际运用

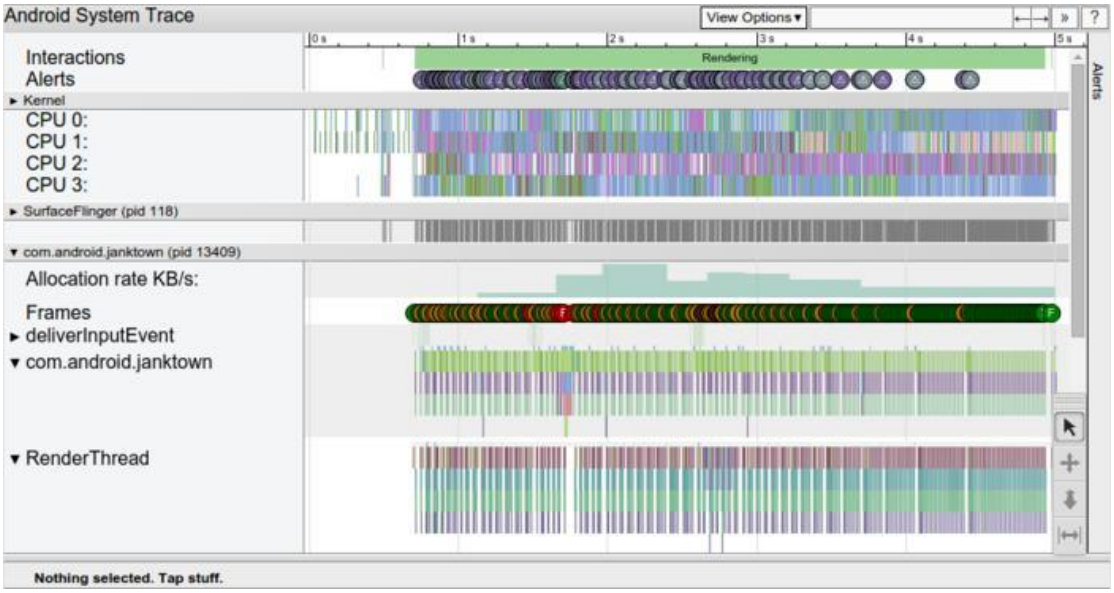
. 3. 性能优化工具

1. 性能优化分析工具学习

在开始代码优化之前，先得学会使用性能分析工具。以下三个工具都是谷歌官方推出的，可以帮助我们定位分析问题，从而优化我们的 APP。

System Trace

Systrace 是一个收集和检测时间信息的工具，它能显示 CPU 和时间被消耗在哪儿了,每个进程和线程都在其 CPU 时间片内做了
什么事儿. 而且会指示哪个地方出了问题,以及给出 Fix 建议。给出的结果 trace 文件是以 html 形式打开的，直接用浏览器打开
查看十分方便。打开方法：打开 DDMS 后，连接手机，点击手机上方一排按钮中的 SysTrace 按钮。
打开的效果如下图：



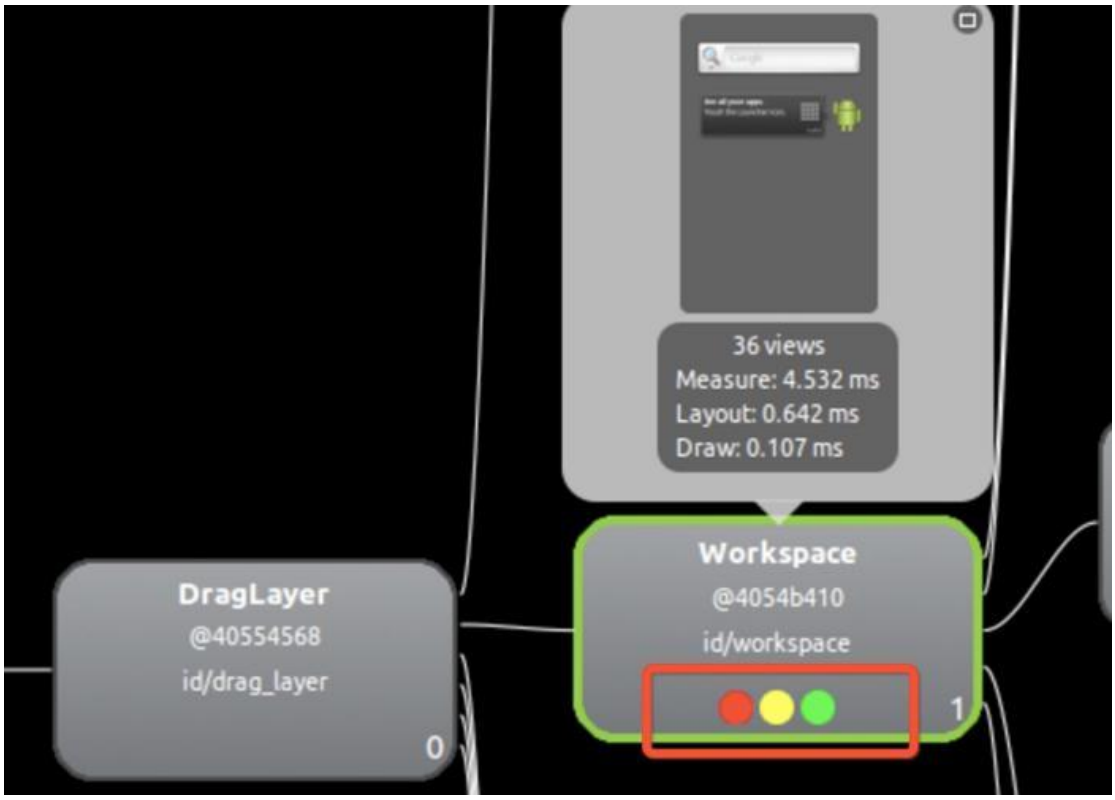
在代码中打点方式如下

```
Trace.beginSection("name");
//要检测运行时间的代码
Trace.endSection();
```

Hierarchy Viewer

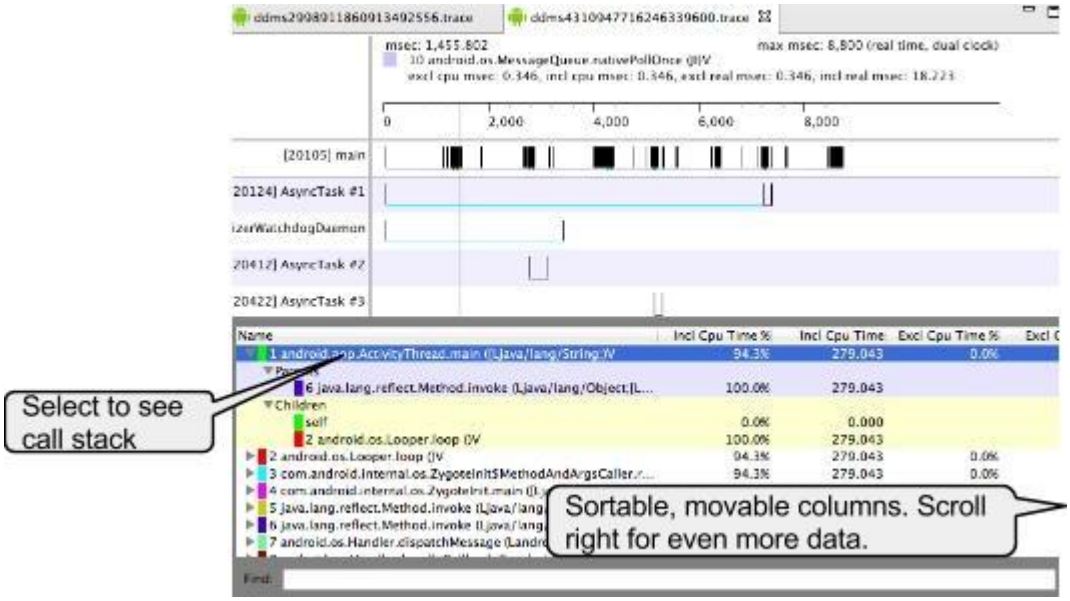
Hierarchy Viewer 提供了一个可视化的界面来观测布局的层级，让我们可以优化布局层级,删除多余的不必要的 View
层级，提升布局速度。另外，开发者模式中调试 GPU 过度绘制选项也可以进行视图层级调试。在 SDK-> tools 目录下
打开 hierarchyviewer.bat 即可。

效果如下图：



TraceView

一个图形化的工具，用来展示和分析方法的执行时间。也是一款性能优化的神器。可以通过像打 log 一样的方式去定位代码的执行时
间，从而可以准确定位是哪一段代码的执行消耗了太多时间。相比 SysTrace，功能更强大，使用起来也更复杂。



2. 布局优化

布局优化相对比较容易，优化可以先从布局来展开。使用 Hierarchy Viewer 和开发者模式中关于布局绘制的选项，可以查到一些问题然后进行修改。

布局嵌套过深： 有的时候为了赶进度，布局设计的不是很好。层级嵌套过深的话，深度遍历各个节点会非常消耗时间，这也是布局优化余地最大的一个点了。很多过深的层级是不必要的。如果布局真的很复杂，不深度嵌套没法实现想要的效果。试试最新的约束布局 Constraintlayout 吧。没有使用过的话，下面这篇官方文档可以帮助你：Constraintlayout 官方介绍文档

使用合适的布局： 三种常见的 ViewGroup 的绘制速度：FrameLayout > LinerLayout > RelativeLayout。当然，如果用 RelativeLayout 可以避免布局嵌套的话是值得的。可以根据这些去决定选用什么样的布局。
化 列表控件优化 不论是 ListView 还是 RecycleView 都有优化点，一个是 convertView 的复用，一个是 ViewHolder 的使用避免重复遍历节点。当然这些都是基础中的基础了。如果发现项目中的代码 ListView 或者 RecycleView 的使用不规范的话，赶紧进行修改吧。

使用 include 标签： 在布局文件中，<include>标签可以指定插入一段布局文件到当前布局。这样的话既提高了布局复用，也减少了我们的代码书写。另外，<merge>标签可以和<include>的标签一起使用从而减少布局层级。

ViewStub 延时加载： 有些布局，比如网络出错的布局，没必要在所有时候都加载出来。使用 ViewStub 可以实现按需加载。ViewStub 本身没有宽高，加载起来几乎不消耗什么资源。当对他 setVisibility(View.VISIBLE)的时候会调用它引用的真实布局填充到当前位置，从而实现了延时加载，节省了正常加载的时间。

移除 Activity 默认背景 只要我们不需要 Activity 的默认背景，就可以移除掉，以减少 Activity 启动时的渲染时间，提升启动效率。移动方法见下：

```
<style name="MyStyle" parent="AppTheme">
        <item name="android:windowNoTitle">true</item>
        <item name="android:windowBackground">@null</item>
</style>
```

3. 线程优化

线程的创建和销毁会带来比较大的性能开销。因此线程优化也很有必要。查看项目中是否存在随意 new thread，线程缺乏管理的情况。使用 AsyncTask 或者线程池对线程进行管理，可以提升 APP 的性能。另外，我比较推荐使用 Rxjava 来实现异步操作，既方便又优雅。推荐一篇 Rxjava 的入门文章

4. 内存泄露

内存泄露会导致 APP 占用内存过高，影响效率，严重的话会导致 OOM。因此如果项目存在内存泄露的话要优先解决。查找内存泄露可以用 LeakCanary 等工具，具体怎么解决，有哪些泄露点，以后有时间也写篇总结。

5. 算法优化

毋庸置疑，使用合适的算法处理事务可以大幅提升 APP 的性能。当然算法不是我的强项，也只能给出一些大致的点：查询考虑二分查找节省时间，尽量不要使用耗时的递归算法。必要的时候可以空间换时间来提高 APP 运行效率。

6. 其他优化点

异步处理耗时任务 在 Activity、Fragemnt 的 onCreate 等初始化方法中，如果执行了太耗时的操作（例如读取各种数据），会影响页面的加载速度，让用户觉得 APP 太慢。这时候可以异步处理这些耗时任务，减小应用启动的时候的负担。

替换矢量图 尽管矢量图有诸多优点，但矢量图的绘制是消耗性能的。在应用初始化加载等比较影响用户体验的地方，还是建议使用 Bitmap 来代替矢量图，提高 APP 开启效率。

正则表达式 经小伙伴用 TraceView 不断的打点发现，正则表达式非常消耗时间。因此尽管正则表达式非常优雅，涉及到性能问题的时候，可以改为其他判断方式来提高 APP 性能。

浮点类型 在 Java 中浮点类型的运算大概比整型数据慢两倍，因此整型数据能解决的问题尽量用整型。

减少冗余 log 开发的时候用于调试的 log，在项目上线的时候没用的要及时删除。当然有用的 log 还是要留下，以便以后分析问题。

删除无用资源 没用用的资源会增大 APK 大小，既然没有用了，上线的时候当然要及时删除。

Lint 代码检查 使用 Lint 等静态代码检查工具可以帮助我们发现很多隐藏的问题。Lint 检查出来的问题越少，说明代码越规范，越不容易出现各种问题，APP 性能自

然也会提升。

滥用全局广播 全局广播也是十分消耗性能的一个点。对于应用内的通讯，使用接口回调，EventBus 等手段比起广播是更好地选择。动态注册广播的时候，也不要忘了广播的注销。

. 4. 性能优化 （讲讲你自己项目中做过的性能优化）

. 6. Http[s] 请求慢的解决办法（ DNS 、携问 带数据、直接访问 IP ）

. 7. 缓存自己如何实现 （e LRUCache 原理 ）

LruCache 中 Lru 算法的实现就是通过 LinkedHashMap 来实现的。LinkedHashMap 继承于 HashMap，它使用了一个双向链表来存储 Map 中的 Entry 顺序关系，对于 get、put、remove 等操作，LinkedHashMap 除了要做 HashMap 做的事情，还做些调整 Entry 顺序链表的工作。
LruCache 中将 LinkedHashMap 的顺序设置为 LRU 顺序来实现 LRU 缓存，每次调用 get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。
调用 put 插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

. 8. 图形图像相关：L OpenGL S ES 管线流程、L EGL 的认识、r Shader 相关

. 9. SurfaceView 、TextureView 、GLSurfaceView 区别及使用场景

. 10. 动画、差值器、估值器(Android d 中的 w View 画 动画和属性动画 - - 简书、 Android 画 动画介绍与使用）

. 11. MVC 、 MVP 、 MVVM

1.mvc:数据、View、Activity，View 将操作反馈给 Activity，Activitiy 去获取数据，数据通过观察者模式刷新给 View。循环依赖

o 1.Activity 重，很难单元测试

o 2.View 和 Model 耦合严重

2.mvp:数据、View、Presenter，View 将操作给 Presenter，Presenter 去获取数据，数据获取好了返回给 Presenter，Presenter 去刷新 View。PV，PM 双向依赖

o 1.接口爆炸

o 2.Presenter 很重

3.mvvm:数据、View、ViewModel，View 将操作给 ViewModel，ViewModel 去获取数据，数据和界面绑定了，数据更新界面更新。

o 1.viewModel 的业务逻辑可以单独拿来测试

o 2.一个 view 对应一个 viewModel 业务逻辑可以分离，不会出现全能类

o 3.数据和界面绑定了，不用写垃圾代码，但是复用起来不舒服

. 12. Handler、、ThreadLocal、、AsyncTask 、IntentService 原理及应用

1.MessageQueue：读取会自动删除消息，单链表维护，在插入和删除上有优势。在其 next()中会无限循环，不断判断是否有消息，有就返回这条消息并移除。

2.Looper：Looper 创建的时候会创建一个 MessageQueue，调用 loop()方法的时候消息循环开始，loop()也是一个死循环，会不断调用 messageQueue 的 next()，当有消息就处理，否则阻塞在 messageQueue 的 next()中。当 Looper 的 quit()被调用的时候会调用 messageQueue 的 quit(),此时 next()会返回 null，然后 loop()方法也跟着退出。

3.Handler：在主线程构造一个 Handler，然后在其他线程调用 sendMessage(),此时主线程的 MessageQueue 中会插入一条 message，然后被 Looper 使用。

4.系统的主线程在 ActivityThread 的 main()为入口开启主线程，其中定义了内部类 Activity.H 定义了一系列消息类型，包含四大组件的启动停止。

5.MessageQueue 和 Looper 是一对一关系，Handler 和 Looper 是多对一

. 13. Gradle （y Groovy 语法、e Gradle 插件开发基础）

为何要学 Groovy

Gradle 是目前 Android 主流的构建工具，不管你是通过命令行还是通过 AndroidStudio 来 build，最终都是通过 Gradle 来实现的。所以学习 Gradle 非常重要。
目前国内对 Android 领域的探索已经越来越深，不少技术领域如插件化、热修复、构建系统等都对 Gradle 有迫切的需求，不懂 Gradle 将无法完成上述事情。所以 Gradle 必须要学习。
Gradle 不单单是一个配置脚本，它的背后是几门语言：

Groovy Language
Gradle DSL
Android DSL

DSL 的全称是 Domain Specific Language，即领域特定语言，或者直接翻译成“特定领域的语言”，再直接点，其实就是这个语言不通用，只能用于特定的某个领域，俗称“小语言”。因此 DSL 也是语言。
实际上，Gradle 脚本大多都是使用 groovy 语言编写的。
Groovy 是一门 jvm 语言，功能比较强大，细节也很多，全部学习的话比较耗时，对我们来说收益较小，并且玩转 Gradle 并不需要学习 Groovy 的全部细节，所以其实我们只需要学一些 Groovy 基础语法与 API 即可。
为何要使用 Groovy
Groovy 是一种基于 JVM 的敏捷开发语言，它结合了众多脚本语言的强大的特性，由于同时又能与 Java 代码很好的结合。一句话：既有面向对象的特性又有纯粹的脚本语言的特性。
由于 Groovy 运行在 JVM 上，因此也可以使用 Java 语言编写的组建。
简单来说，Groovy 提供了更加灵活简单的语法，大量的语法糖以及闭包特性可以让你用更少的代码来实现和 Java 同样的功能。
行 如何编译运行 Groovy
Groovy 是一门 jvm 语言，它最终是要编译成 class 文件然后在 jvm 上执行，所以 Java 语言的特性 Groovy 都支持，Groovy 支持 99%的 java 语法，我们完全可以在 Groovy 代码中直接粘贴 java 代码。
可以安装 Groovy sdk 来编译和运行。但是我并不想搞那么麻烦，毕竟我们的最终目的只是学习 Gradle。
推荐大家通过这种方式来编译和运行 Groovy。
在当面目录下创建 build.gradle 文件，在里面创建一个 task，然后在 task 中编写 Groovy 代码即可，如下所示：

```
task(testGroovy).doLast{
println "开始运行自定义 task"
test()
}
def test() {
println "执行 Groovy 语法的代码"
System.out. println ( "执行 Java 语法的代码!" );
}
```

然后在命令行终端中执行如下命令即可：

gradle testGroovy
> Configure project : app
> Task :app: testGroovy
开始运行自定义 task
执行 Groovy 语法的代码
执行 Java 语法的代码!
BUILDSUCCESSFUL in 3 s 1 actionable task: 1 executed
我们知道，在 Android 项目中，我们只要更改 build.gradle 文件一点内容，AS 就会提示我们同步：

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

但是在我们测试 Groovy 时中，我们更改 build.gradle 文件后可以不必去同步，执行命令时会自动执行你修改后的最新逻辑。

最基本的语法

Groovy 中的类和方法默认都是 public 权限的，所以我们可以省略 public 关键字，除非我们想使用 private。
Groovy 中的类型是弱化的，所有的类型都可以动态推断，但是 Groovy 仍然是强类型的语言，类型不匹配仍然会报错。
Groovy 中通过 def 关键字来声明变量和方法。
Groovy 中很多东西都是可以省略的，比如
语句后面的分号是可以省略的
def 和 变量的类型 中的其中之一是可以省略的(不能全部省略)
def 和 方法的返回值类型 中的其中之一是可以省略的(不能全部省略)
方法调用时的圆括号是可以省略的
方法声明中的参数类型是可以省略的
方法的最后一句表达式可作为返回值返回，而不需要 return 关键字
省略 return 关键字并不是一个好的习惯，就如同 if else
while 后面只有一行语句时可以省略大括号一样，以后如果添加了其他语句，很有可能会导致逻辑错误

```
def int a = 1 ; //如果 def 和 类型同时存在，IDE 会提示你"def 是不需要的(is unnecessary)"def String b = "hello world" //省略分号，存在分号时也会提示你 unnecessarydefc = 1 //省略类型
def hello () { //省略方法声明中的返回值类型
println ( "hello world" );
println "hello groovy" //省略方法调用时的圆括号
return1 ;
}
def hello ( String msg) {
println "hello" + msg //省略方法调用时的圆括号
1 ; //省略 return
}
```

```
int hello(msg) { //省略方法声明中的参数类型
println msg
return1 // 这个 return 不能省略
println "done" //这一行代码是执行不到的，IDE 会提示你 Unreachable
statement，但语法没错
}
```

支持的数据类型

在 Groovy 中，数据类型有：

- Java 中的基本数据类型
- Java 中的对象
- Closure（闭包）
- 加强的 List、Map 等集合类型
- 加强的 File、Stream 等 IO 类型

类型可以显示声明，也可以用 def 来声明，用 def 声明的类型 Groovy 将会进行类型推断。

基本数据类型和对象和 Java 中的一致，只不过在 Gradle 中，对象默认的修饰符为 public。

String

String 的特色在于字符串的拼接，比如

```
def a = 1def b = "hello"def c = "a=${a}, b=${b}"
println c //a=1, b=hello
```

闭包

Groovy 中有一种特殊的类型，叫做 Closure，翻译过来就是闭包，这是一种类似于 C 语言中函数指针的东西。

闭包用起来非常方便，在 Groovy 中，闭包作为一种特殊的数据类型而存在，闭包可以作为方法的参数和返回值，也可以作为一个变量而存在。

闭包可以有返回值和参数，当然也可以没有。下面是几个具体的例子：

```
def test () {
def closure = { String parameters -> //闭包的基本格式
printlnparameters
}
def closure2 = { a, b -> // 省略了闭包的参数类型
println "a=${a}, b=${b}"
}
def closure3 = { a ->
a+ 1 //省略了 return
}
def closure4 = { //省略了闭包的参数声明
println "参数为 ${it}" //如果闭包不指定参数，那么它会有一个隐含
的参数 it
}
closure( "包青天" ) //包青天
closure2 10086 , "包青天" //a=10086, b=包青天
```

```
println closure3( 1 ) //2
//println closure3 2 //不允许省略圆括号，会提示：Cannot get
property '1' on null object
closure4() //参数为 null
closure4 //不允许省略圆括号，但是并不会报错
closure4 10086//参数为 10086
}
```

闭包的一个难题是如何确定闭包的参数(包括参数的个数、参数的类型、参数的意义)，尤其当我们调用 Groovy 的 API 时，这个时候没有其他办法，只有查询 Groovy 的文档才能知道。

List 和 和 Map

Groovy 加强了 Java 中的集合类，比如 List、Map、Set 等。

基本使用如下：

```
def emptyList = []
def list = [ 10086 , "hello" , true ] list [ 1 ] = "world"
assert list [ 1 ] == "world"
println list [ 0 ] //10086list << 5 //相当于 add()
assert 5 in list// 调用包含方法
println list //[10086, world, true, 5]
def range = 1..5
assert 2 in rangeprintln range //1..5println range .size() //5
def emptyMap = [:]
def map = [ "id" : 1 , "name" : "包青天" ] map << [age: 29 ] //添加元素 map [ "id" ]
= 10086 //访问元素方式一 map .name = "哈哈" //访问元素方式二，这种方式
最简单 println map //{id=10086, name=哈哈, age=29}
```

可以看到，通过 Groovy 来操作 List 和 Map 显然比 Java 简单的多。

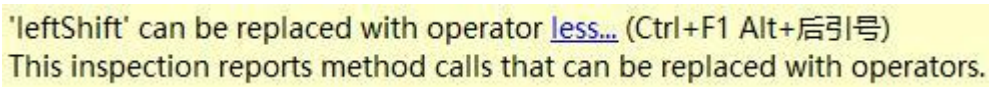
上面有一个看起来很奇怪的操作符<<，其实这并没有什么大不了，<<表示向 List 中添加新元素的意思，这一点从 List 文档 当也能查到。

```
public List leftShift( Object value)
    Overloads the left shift operatorto provide an easy way to append
```

objects to a List. 重载左移位运算符，以提供将对象 append 到 List 的简单方法。

```
    Parameters: value - an Object to be added to the List.
    Returns: sameList, after the value was added to it.
```

实际上，这个运算符是大量使用的，并且当你用 leftShift 方法时 IDE 也会提示你让你使用左移位运算符<<替换：




```
def list = [ 1 , 2 ] list .leftShift 3
assert list == [ 1 , 2 , 3 ] list << 4
println list //[1, 2, 3, 4]
```

闭包的参数

这里借助 Map 再讲述下如何确定闭包的参数。比如我们想遍历一个 Map，我们想采用 Groovy 的方式，通过查看文档，发现它有如下两个方法，看起来和遍历有关：

Map each(Closure closure)：Allows a Map to be iterated through using a closure.

Map eachWithIndex(Closure closure)：Allows a Map to be iterated through using a closure.

可以发现，这两个 each 方法的参数都是一个闭包，那么我们如何知道闭包的参数呢？当然不能靠猜，还是要查文档。

public Map each (Closure closure)

Allows a Map to be iterated through using a closure.If the closure takes one parameter then it will be passed the Map.Entry otherwise if the closure takes two parameters then it will be passed the key and the value.

In general, the order in which the map contents are processed cannot be guaranteed(通常无法保证处理元素的顺序). In practise(在实践中), specialized forms of Map(特 殊 形 式 的 Map), e.g. a TreeMap will have its contents processed according to the natural ordering(自然顺序) of the map.

```
def result = ""
[ a:1 , b:3 ].each {key, value -> result += "$key$value" } //两个参数 assert result == "a1b3"
def result = ""
[ a:1 , b:3 ].each { entry -> result +=entry} //一个参数 assert result == "a=1b=3"
[ a:1 , b: 3 ].each { println "[${it.key} :${it.value}]" } //一个隐含的参数 it，key 和 value 是属性名
```

试想一下，如果你不知道查文档，你又怎么知道 each 方法如何使用呢？光靠从网上搜，API 文档中那么多接口，搜的过来吗？记得住吗？

加强的 IO

在 Groovy 中，文件访问要比 Java 简单的多，不管是普通文件还是 xml 文件。怎么使用呢？查来 File 文档。

publicObject eachLine(Closure closure)

Iterates through this file line by line. Each line is passed tothe given 1 or 2 arg closure. The file is read using areader which is closed before this method returns.

Parameters: closure- a closure (arg 1 is line, optional arg 2 is line number starting at line 1)

Returns: the last value returned by the closure

可以看到，eachLine 方法也是支持 1 个或 2 个参数的，这两个参数分别是什么意思，就需

要我们学会读文档了，一味地从网上搜例子，多累啊，而且很难彻底掌握：

```
def file = new File( "a.txt" )
file.eachLine { line, lineNo ->
println "${lineNo} ${line}" //行号，内容
}
```

file.eachLine { line ->
println "${line}" //内容
}
除了 eachLine，File 还提供了很多 Java 所没有的方法，大家需要浏览下大概有哪些方法，然后需要用的时候再去查就行了，这就是学习 Groovy 的正道。
问 访问 xml 文件
Groovy 访问 xml 有两个类：XmlParser 和 XmlSlurper，二者几乎一样，在性能上有细微的差别，不过这对于本文不重要。
groovy.util.XmlParser 的 API 文档
文档中的案例：
def xml = '<root><one a1="uno!"/><two>Sometext!</two></root>'//
或者 def xml = new XmlParser().parse(new File("filePath.xml"))def
rootNode = new XmlParser().parseText(xml) //根节点 assert rootNode.name() == 'root'//根节点的名称 assert rootNode.one[0]. @a 1 == 'uno!' //根节点中的子节点 one 的 a1 属性的值 assert rootNode.two.text() == 'Some text!'
//根节点中的子节点 two 的内容
rootNode.children().each { assert it.name() in ['one' , 'two'] }
更多的细节查文档即可。

其他的一些语法特性：

Getter 和 和 Setter
当你在 Groovy 中创建一个 beans 的时候，通常我们称为 POGOS(Plain Old Groovy Objects)，Groovy 会自动帮我们创建 getter/setter 方法。
当你对 getter/setter 方法有特殊要求，你尽可提供自己的方法，Groovy 默认的 getter/setter 方法会被替换。
构造器
有一个 bean
class Server {
String name
Cluster cluster
}
初始化一个实例的时候你可能会这样写：
def server = new Server () server .name = "Obelix"server .cluster = aCluster
其实你可以用带命名的参数的默认构造器，会大大减少代码量：
def server = new Server(name:"Obelix" , cluster: aCluster)
Class 类型
在 Groovy 中 Class 类型的.class 后缀不是必须的，比如：
def func(Class clazz) {
println clazz

```
} func(File.class) //class java.io.Filefunc(File) //class
java.io.File
```

用 使用 with() 操作符

当更新一个实例的时候，你可以使用 with()来省略相同的前缀，比如：

```
Book book = new Book()
book. with {
id = 1 //等价于 book.id = 1
name = "包青天"
start( 10086 )
stop( "包青天" )
}
```

判断是否为真

所有类型都能转成布尔值，比如 null 和 void 相当于 0 或者相当于 false，其他则相当于 true，所以：

```
if ( name ) {}//等价于 if ( name != null && name .length > 0 ) {}
```

在 Groovy 中可以在类中添加 asBoolean()方法来自定义是否为真。

简洁的三元表达式

在 Groovy 中，三元表达式可以更加简洁，比如：

```
def result = name ?: ""//等价于 def result = name != null ? name : ""
```

捕获任何异常

如果你实在不想关心 try 块里抛出何种异常，你可以简单的捕获所有异常，并且可以省略异常类型：

```
try {
// ...
} catch (any) { //可以省略异常类型
// something bad happens
}
```

这里的 any 并不包括 Throwable，如果你真想捕获 everything，你必须明确的标明你想捕获 Throwable

简洁的非空判断

在 java 中，你要获取某个对象的值必须要检查是否为 null，这就造成了大量的 if 语句；

在 Groovy 中，非空判断可以用?.表达式，比如：

```
println order?.customer?.address //等价于 if (order != null ) {
if (order.getCustomer() != null ) {
if (order.getCustomer().getAddress() != null ) {
System. out .println(order.getCustomer().getAddress());
}
}
}
```

使用断言

在 Groovy 中，可以使用 assert 来设置断言，当断言的条件为 false 时，程序将会抛出异常：

```
def check ( String name ) {
assert name // 检查方法传入的参数是否为空， name non- null and non- empty
according to Groovy Truth
assert name ?.size() > 3
}
```

==和 和 equals

Groovy 里的 is()方法等同于 Java 里的==。

Groovy 中的==是更智能的 equals()，比较两个类的时候，你应该使用 a.is(b)而不是==。

Groovy 中的==可以自动避免 NullPointerException 异常

```
status == "包青天"
//等价于 Java 中的 status != null && status .equals( "包青天" )
```

switch 方法

在 Groovy 中，switch 方法变得更加灵活，可以同时支持更多的参数类型：

```
def x = nulldef result = ""switch (x) {
case "foo" : result = "found foo" //没有 break 时会继续向下判断
case "bar" : result += "bar"
break
case [ 4 , 5 , 6 ]: result = "list" //匹配集合中的元素
break
case 12..30 : result= "range" //匹配某个范围内的元素
break
case Integer: result = "integer" //匹配 Integer 类型
break
case { it > 3 }: result = "number > 3" //匹配表达式
break
case Number: result = "number" //匹配 Number 类型
break default: result = "default"
}
println result
```

字符串分行

Java 中，字符串过长需要换行时我们一般会这样写：

```
throw new PluginException("Failed to execute command list -applications: " +
" The group with name " +
parameterMap.groupname[0] +
"is not compatible group of type " +
SERVER_TYPE_NAME)
```

Groovy 中你可以用 \ 字符，而不需要添加一堆的双引号：

```
thrownew PluginException("Failed to execute command list -applications:\
The group with name ${parameterMap.groupname[ 0 ]} \ is not compatible group
of type ${SERVER_TYPE_NAME} ")
```

或者使用多行字符串""":

```
thrownew PluginException("""Failed to execute command list -applications:
```

The group with name ${parameterMap.groupname[0]}
is not compatible group of type ${SERVER_TYPE_NAME)} """)
Groovy 中，单引号引起来的字符串是 java 字符串，不能使用占位符来替换变量，双引号引起的字符串则是 java 字符串或者 Groovy 字符串。
Import 别名
在 java 中使用两个类名相同但包名不同的两个类，像 java.util.List 和 java.wt.List，你必须使用完整的包名才能区分。Groovy 中则可以使用 import 别名：
import java.util. Listas jurist //使用别名 import java.awt. List as
aList import java.awt.WindowConstants as WC import static pkg.SomeClass.foo
//静态引入方法

. 14. 热修复、插件化

插件化框架描述：k dynamicLoadApk 为例子

1.可以通过 DexClassLoader 来对 apk 中的 dex 包进行加载访问

2.如何加载资源是个很大的问题，因为宿主程序中并没有 apk 中的资源，所以调用 R 资源会报错，所以这里使用了 Activity 中的实现 ContextImpl 的 getAssets()和 getResources()再加上反射来实现。

3.由于系统启动 Activity 有很多初始化动作要做，而我们手动反射很难完成，所以可以采用接口机制，将 Activity 的大部分生命周期提取成接口，然后通过代理 Activity 去调用插件 Activity 的生命周期。同时如果像增加一个新生命周期方法的时候，只需要在接口中和代理中声明一下就行。

4.缺点：

o 1.慎用 this，因为在 apk 中使用 this 并不代表宿主中的 activity，当然如果 this 只是表示自己的接口还是可以的。除此之外可以使用 that 代替 this。
o 2.不支持 Service 和静态注册的 Broadcast
o 3.不支持 LaunchMode 和 Apk 中 Activity 的隐式调用。

热修复：x Andfix 为例子

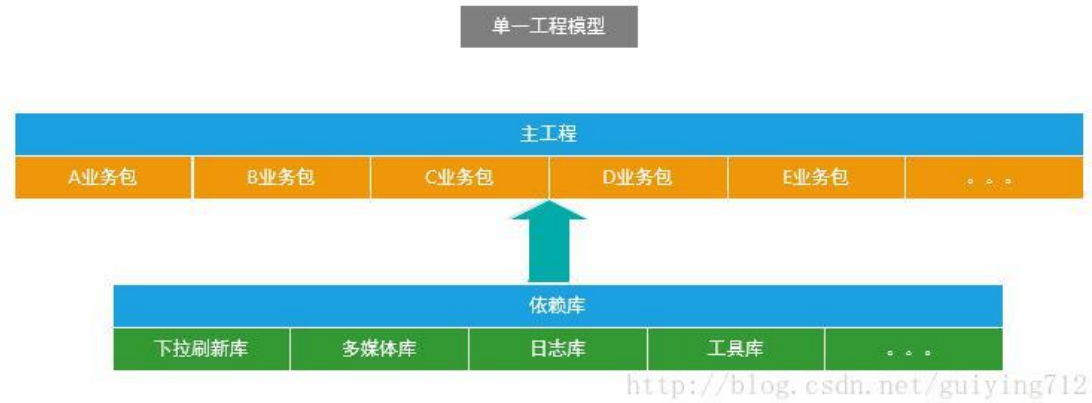
1.大致原理：apkpatch 将两个 apk 做一次对比，然后找出不同的部分。可以看到生成的 apatch 了文件，后缀改成 zip 再解压开，里面有一个 dex 文件。通过 jadx 查看一下源码，里面就是被修复的代码所在的类文件,这些更改过的类都加上了一个_CF 的后缀，并且变动的方法都被加上了一个叫@MethodReplace 的 annotation，通过 clazz 和 method 指定了需要替换的方法。然后客户端 sdk 得到补丁文件后就会根据 annotation 来寻找需要替换的方法。最后由 JNI 层完成方法的替换。

2.无法添加新类和新的字段、补丁文件很容易被反编译、加固平台可能会使热补丁功能失效

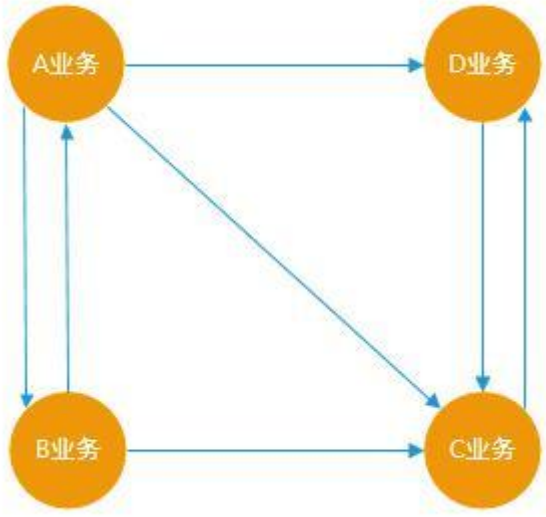
. 15. 组件化架构思路

1 、为什么要项目组件化

随着 APP 版本不断的迭代，新功能的不断增加，业务也会变的越来越复杂，APP 业务模块的数量有可能还会继续增加，而且每个模块的代码也变的越来越多，这样发展下去单一工程下的 APP 架构势必会影响开发效率，增加项目的维护成本，每个工程师都要熟悉如此之多的代码，将很难进行多人协作开发，而且 Android 项目在编译代码的时候电脑会非常卡，又因为单一工程下代码耦合严重，每修改一处代码后都要重新编译打包测试，导致非常耗时，最重要的是这样的代码想要做单元测试根本无从下手，所以必须要有更灵活的架构代替过去单一的工程架构。



上图是目前比较普遍使用的 Android APP 技术架构，往往是在一个界面中存在大量的业务逻辑，而业务逻辑中充斥着各种网络请求、数据操作等行为，整个项目中也没有模块的概念，只有简单的以业务逻辑划分的文件夹，并且业务之间也是直接相互调用、高度耦合在一起的；



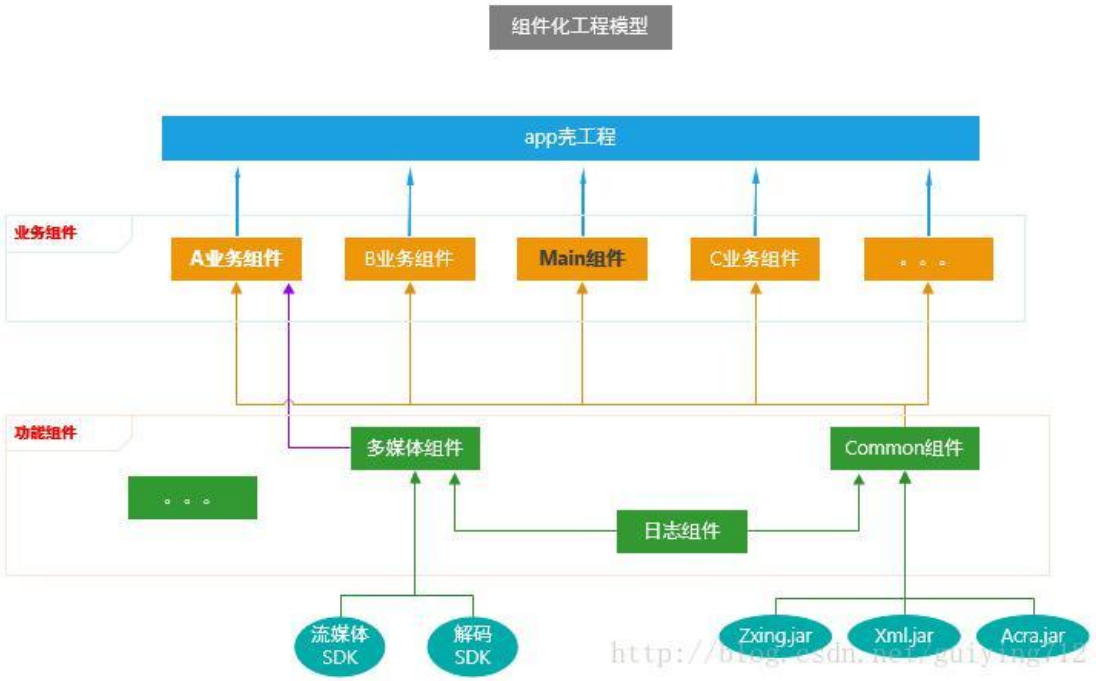
上图单一工程模型下的业务关系，总的来说就是：你中有我，我中有你，相互依赖，无法分离。

然而随着产品的迭代，业务越来越复杂，随之带来的是项目结构复杂度的极度增加，此时我们会面临如下几个问题：

1、实际业务变化非常快，但是单一工程的业务模块耦合度太高，牵一发而动全身；

2、对工程所做的任何修改都必须要编译整个工程；
3、功能测试和系统测试每次都要进行；
4、团队协同开发存在较多的冲突.不得不花费更多的时间去沟通和协调，并且在开发过程中，任何一位成员没办法专注于自己的功能点，影响开发效率；
5、不能灵活的对业务模块进行配置和组装；
为了满足各个业务模块的迭代而彼此不受影响，更好的解决上面这种让人头疼的依赖关系，就需要整改 App 的架构。

2 、如何组件化



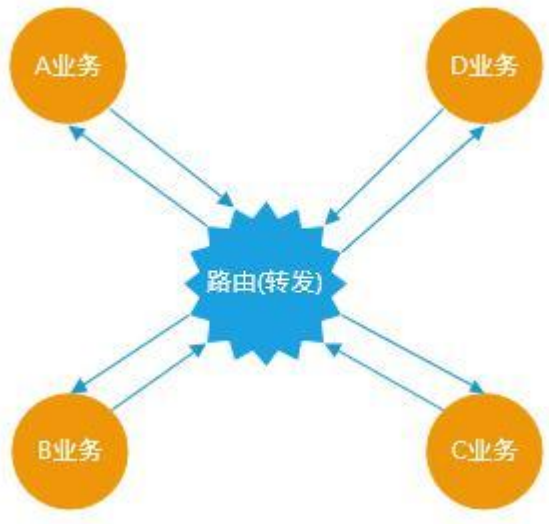
上图是组件化工程模型，为了方便理解这张架构图，下面会列举一些组件化工程中用到的名词的含义：

名词	含义
集成模式	所有的业务组件被“app 壳工程”依赖，组成一个完整的 APP；
组件模式	可以独立开发业务组件，每一个业务组件就是一个 APP；
app 壳工程	负责管理各个业务组件，和打包 apk，没有具体的业务功能；
业务组件	根据公司具体业务而独立形成一个的工程；
功能组件	提供开发 APP 的某些基础功能，例如打印日志、树状图等；
Main 组件	属于业务组件，指定 APP 启动页面、主界面；
Common 组件	属于功能组件，支撑业务组件的基础，提供多数业务组件需要的功能，例如提供网络请求功能；

**

Android APP 组件化架构的目标是告别结构臃肿，让各个业务变得相对独立，业务组件在组件模式下可以独立开发，而在集成模式下又可以变为 arr 包集成到“app 壳工程”中，组成一个完整功能的 APP；

从组件化工程模型中可以看到， 业务组件之间是独立的，没有关联的，这些业务组件在集成模式下是一个个 library，被 app 壳工程所依赖，组成一个具有完整业务功能的 APP 应用，但是在组件开发模式下，业务组件又变成了一个个 application，它们可以独立开发和调试，由于在组件开发模式下，业务组件们的代码量相比于完整的项目差了很远，因此在运行时可以显著减少编译时间。



这是组件化工程模型下的业务关系，业务之间将不再直接引用和依赖，而是通过“路由”这样

一个中转站间接产生联系，而 Android 中的路由实际就是对 URL Scheme 的封装；

如此规模大的架构整改需要付出更高的成本，还会涉及一些潜在的风险，但是整改后的架构

能够带来很多好处：
1、加快业务迭代速度，各个业务模块组件更加独立，不再出现业务耦合情况；
2、稳定的公共模块采用依赖库方式，提供给各个业务线使用，减少重复开发和维护工作量；
3、迭代频繁的业务模块采用组件方式，各业务研发可以互不干扰、提升协作效率，并控制产品质量；
4、为新业务随时集成提供了基础，所有业务可上可下，灵活多变；
5、降低团队成员熟悉项目的成本，降低项目的维护难度；
6、加快编译速度，提高开发效率；
7、控制代码权限，将代码的权限细分到更小的粒度；

3 、组件化实施流程

1 ）组件模式和集成模式的转换
Android Studio 中的 Module 主要有两种属性，分别为：
1、application 属性，可以独立运行的 Android 程序，也就是我们的 APP；
apply plugin: ‘com.android.application’

2、library 属性，不可以独立运行，一般是 Android 程序依赖的库文件；
apply plugin: ‘com.android.library’

Module 的属性是在每个组件的 build.gradle 文件中配置的，当我们在组件模式开发时，业务组件应处于 application 属性，这时的业务组件就是一个 Android App，可以独立开发和调试；而当我们转换到集成模式开发时，业务组件应该处于 library 属性，这样才能被我们的“app 壳工程”所依赖，组成一个具有完整功能的 APP；
但是我们如何让组件在这两种模式之间自动转换呢？总不能每次需要转换模式的时候去每个业务组件的 Gralde 文件中去手动把 Application 改成 library 吧？如果我们的项目只有两三个组件那么这个办法肯定是可行的，手动去改一遍也用不了多久，但是在大型项目中我们可能会有十几个业务组件，再去手动改一遍必定费时费力，这时候就需要程序员发挥下懒的本质了。
试想，我们经常在写代码的时候定义静态常量，那么定义静态常量的目的什么呢？当一个常量需要被好几处代码引用的时候，把这个常量定义为静态常量的好处是当这个常量的值需要改变时我们只需要改变静态常量的值，其他引用了这个静态常量的地方都会被改变， 做到了一次改变 ， 到处生效；根据这个思想，那么我们就可以在我们的代码中的某处定义一个决定
业务组件属性的常量，然后让所有业务组件的 build.gradle 都引用这个常量，这样当我们改变了常量值的时候，所有引用了这个常量值的业务组件就会根据值的变化改变自己的属性；可是问题来了？静态常量是用 Java 代码定义的，而改变组件属性是需要在 Gradle 中定义的，Gradle 能做到吗？
Gradle 自动构建工具有一个重要属性，可以帮助我们完成这个事情。每当我们用 AndroidStudio 创建一个 Android 项目后，就会在项目的根目录中生成一个文件 gradle.properties，我们将使用这个文件的一个重要属性：在 在 Android 项目中的任何一个 build.gradle 文件中都可以把 gradle.properties 中的常量读取出来；那么我们在上面提
到解决办法就有了实际行动的方法，首先我们在 gradle.properties 中定义一个常量

值 isModule （是否是组件开发模式，true 为是，false 为否）：

```
# 每次更改“isModule”的值后，需要点击 "Sync Project"按钮
isModule=false
```

然后我们在业务组件的 build.gradle 中读取 isModule，但是 gradle.properties 还有一个重要属性： gradle.properties 中的数据类型都是 String 类型，使用其他数据类型需要自行转换；也就是说我们读到 isModule 是个 String 类型的值，而我们需要的是 Boolean 值，代码如下：

```
if (isModule.toBoolean()) {
apply plugin: 'com.android.application'
} else {
apply plugin: 'com.android.library'
}
```

这样我们第一个问题就解决了，当然了 每次改变 isModule 的值后，都要同步项目才能生效；

2 ）组件之间 t AndroidManifest 合并问题

在 AndroidStudio 中每一个组件都会有对应的 AndroidManifest.xml，用于声明需要的权限、Application、Activity、Service、Broadcast 等，当项目处于组件模式时，业务组件的 AndroidManifest.xml 应该具有一个 Android APP 所具有的的所有属性，尤其是声明 Application 和要 launch 的 Activity，但是当项目处于集成模式的时候，每一个业务组件的 AndroidManifest.xml 都要合并到“app 壳工程”中，要是每一个业务组件都有自己的 Application 和 launch 的 Activity，那么合并的时候肯定会冲突，试想一个 APP 怎么可能会有多个 Application 和 launch 的 Activity 呢？

但是大家应该注意到这个问题是在组件开发模式和集成开发模式之间转换引起的问题，而在上一节中我们已经解决了组件模式和集成模式转换的问题，另外大家应该都经历过将 Android 项目从 Eclipse 切换到 AndroidStudio 的过程，由于 Android 项目在 Eclipse 和 AndroidStudio 开发时 AndroidManifest.xml 文件的位置是不一样的，我们需要在 build.gradle 下 中指定下 AndroidManifest.xml 的位置，AndroidStudio 才能读取到 AndroidManifest.xml，这样解决办法也就有了，我们可以 为组件开发模式下的业务组件再个 创建一个 AndroidManifest.xml ， 然后根据 isModule 指定 AndroidManifest.xml 的文件

路径，让业务组件在集成模式和组件模式下使用不同的 AndroidManifest.xml，这样表单冲突的问题就可以规避了。

上图是组件化项目中一个标准的业务组件目录结构，首先我们在 main 文件夹下创建一个 module 文件夹用于存放组件开发模式下业务组件的 AndroidManifest.xml，而 AndroidStudio 生成的 AndroidManifest.xml 则依然保留，并用于集成开发模式下业务组件的表单；然后我们需要在业务组件的 build.gradle 中指定表单的路径，代码如下：

```
sourceSets {
main {
if (isModule.toBoolean()) {
manifest.srcFile 'src/main/module/AndroidManifest.xml'
} else {
manifest.srcFile 'src/main/AndroidManifest.xml'
}
```

```
}
}
```

这样在不同的开发模式下就会读取到不同的 AndroidManifest.xml ，然后我们需要修改这两个表单的内容以为我们不同的开发模式服务。

首先是集成开发模式下的 AndroidManifest.xml，前面我们说过集成模式下，业务组件的表单是绝对不能拥有自己的 Application 和 launch 的 Activity 的，也不能声明 APP 名称、图标等属性，总之 app 壳工程有的属性，业务组件都不能有，下面是一份 标准的集成开发模式下业务组件的 AndroidManifest.xml：

```
<manifestxmlns:android="http://schemas.android.com/apk/res/android"
package="com.guiying.girls">
<application android:theme="@style/AppTheme">
<activity
android:name=".main.GirlsActivity"
android:screenOrientation="portrait" />
<activity
android:name=".girl.GirlActivity"
android:screenOrientation="portrait"
android:theme="@style/AppTheme.NoActionBar" />
</application>
</manifest>
```

我在这个表单中只声明了应用的主题，而且这个主题还是跟 app 壳工程中的主题是一致的，都引用了 common 组件中的资源文件，在这里声明主题是为了方便这个业务组件中有使用默认主题的 Activity 时就不用再给 Activity 单独声明 theme 了。

然后是组件开发模式下的表单文件：

```
<manifestxmlns:android="http://schemas.android.com/apk/res/android"
package="com.guiying.girls">
<application
android:name="debug.GirlsApplication"
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/girls_name"
android:supportsRtl="true"
android:theme="@style/AppTheme">
<activity
android:name=".main.GirlsActivity"
android:screenOrientation="portrait">
<intent-filter>
<action android:name="android.intent.action.MAIN"/>
<categoryandroid:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<activity
android:name=".girl.GirlActivity"
android:screenOrientation="portrait"
```

```
android:theme="@style/AppTheme.NoActionBar" />
</application>
</manifest>
```

组件模式下的业务组件表单就是一个 Android 项目普通的 AndroidManifest.xml，这里就不在过多介绍了。

3 ）全局 t Context 的获取及组件数据初始化

当 Android 程序启动时，Android 系统会为每个程序创建一个 Application 类的对象，并且只创建一个，application 对象的生命周期是整个程序中最长的，它的生命周期就等于这个程序的生命周期。在默认情况下应用系统会自动生成 Application 对象，但是如果我们自定义了 Application，那就需要在 AndroidManifest.xml 中声明告知系统，实例化的时候，是实例化我们自定义的，而非默认的。

但是我们在组件化开发的时候，可能为了数据的问题每一个组件都会自定义一个 Application 类，如果我们在自己的组件中开发时需要获取 全局的 Context，一般都会直接获取 application 对象，但是当所有组件要打包合并在一起的时候就会出现问题，因为最后程序只有一个 Application，我们组件中自己定义的 Application 肯定是没法使用的，因此我们需要想办法再任何一个业务组件中都能获取到全局的 Context，而且这个 Context 不管是在组件开发模式还是在集成开发模式都是生效的。

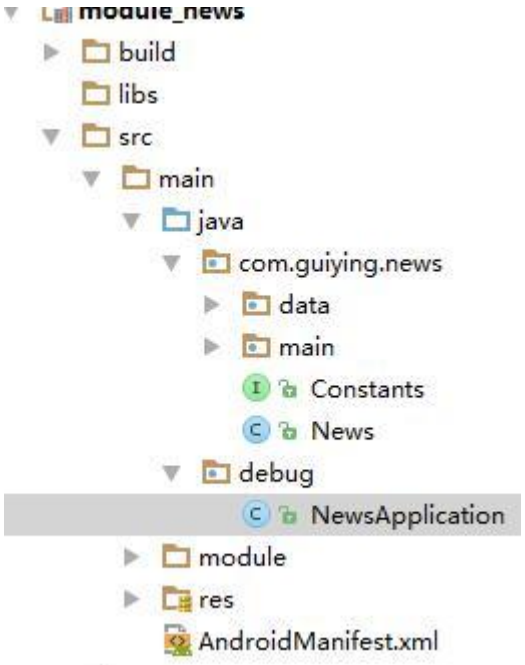
在 组件化工程模型图中，功能组件集合中有一个 Common 组件， Common 有公共、公用、共同的意思，所以这个组件中主要封装了项目中需要的基础功能，并且每一个业务组件都要依赖 Common 组件，Common 组件就像是万丈高楼的地基，而业务组件就是在 Common 组件这个地基上搭建起来我们的 APP 的，Common 组件会专门在一个章节中讲解，这里只讲 Common 组件中的一个功能，在 Common 组件中我们封装了项目中用到的各种 Base 类，这些基类中就有 BaseApplication 类。

BaseApplication 主要用于各个业务组件和 app 壳工程中声明的 Application 类继承用的，只要各个业务组件和 app 壳工程中声明的 Application 类继承了 BaseApplication，当应用启动时 BaseApplication 就会被动实例化，这样从 BaseApplication 获取的 Context 就会生效，也就从根本上解决了我们不能直接从各个组件获取全局 Context 的问题；

这时候大家肯定都会有个疑问？不是说了业务组件不能有自己的 Application 吗，怎么还让他们继承 BaseApplication 呢？其实我前面说的是业务组件不能在集成模式下拥有自己的 Application，但是这不代表业务组件也不能在组件开发模式下拥有自己的 Application，其实业务组件在组件开发模式下必须要有自己的 Application 类，一方面是为了让 BaseApplication 被实例化从而获取 Context，还有一个作用是， 业务组件自己的 Application 可以在组件开发模式下初始化一些数据，例如在组件开发模式下，A 组件没有登录页面也没法登录，因此就无法获取到 Token，这样请求网络就无法成功，因此我们需要在 A 组件这个 APP 启动后就应该已经登录了，这时候组件自己的 Application 类就有了用武之地，我们在组件的 Application 的 onCreate 方法中模拟一个登陆接口，在登陆成功后将数据保存到本地，这样就可以处理 A 组件中的数据业务了；另外我们也 可以在组件
Application 中初始化一些第三方库。

但是，实际上业务组件中的 Application 在最终的集成项目中是没有什么实际作用的，组件自己的 Application 仅限于在组件模式下发挥功能，因此我们需要在将项目从组件模式转换到集成模式后将组件自己的 Application 剔除出我们的项目；在 AndroidManifest 合并问题小节中介绍了如何在不同开发模式下让 Gradle 识别组件表单的路径，这个方法也同样适用

于 Java 代码：



我们在 Java 文件夹下创建一个 debug 文件夹，用于存放不会在业务组件中引用的类，例如上图中的 NewsApplication ，你甚至在 可以在 debug 文件夹中创建一个 Activity ，然后
组件表单中声明启动这个 Activity， ， 在这个 Activity 中不用 setContentView， ， 只需要在启
动你的目标 Activity 的时候传递参数就行，这样就就可以解决组件模式下某些 Activity 需
要 要 getIntent 数据而没有办法拿到的情况，代码如下；

```
public class LauncherActivityextends AppCompatActivity {
@Override
protectedvoidonCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
request();
Intent intent = newIntent(this, TargetActivity.class);
intent.putExtra("name", "avcd");
intent.putExtra("syscode", "023e2e12ed");
startActivity(intent);
finish();
}
//申请读写权限
private void request() {
AndPermission.with(this)
.requestCode(110)
.permission(Manifest.permission.WRITE_EXTERNAL_STORAGE,
Manifest.permission.CAMERA,
Manifest.permission.READ_PHONE_STATE)
.callback(this)
```

```
.start();
}
}
```

接下来在业务组件的 build.gradle 中，根据 isModule 是否是集成模式将 debug 这个 Java 代码文件夹排除：

```
sourceSets {
main {
if (isModule.toBoolean()) {
manifest.srcFile 'src/main/module/AndroidManifest.xml'
} else {
manifest.srcFile 'src/main/AndroidManifest.xml'
//集成开发模式下排除 debug 文件夹中的所有 Java 文件
java{
exclude 'debug/**'
}
}
}
}
```

4 library 依赖问题

在介绍这一节的时候，先说一个问题，在 组件化工程模型图中，多媒体组件和 Common 组件都依赖了日志组件，而 A 业务组件有同时依赖了多媒体组件和 Common 组件，这时候就

会有人问，你这样搞岂不是日志组件要被重复依赖了，而且 Common 组件也被每一个业务组件依赖了，这样不出问题吗？

其实大家完全没有必要担心这个问题，如果真有重复依赖的问题，在你编译打包的时候就会报错，如果你还是不相信的话可以反编译下最后打包出来的 APP，看看里面的代码你就知道了。组件只是我们在代码开发阶段中为了方便叫的一个术语，在组件被打包进 APP 的时候是没有这个概念的，这些组件最后都会被打包成 arr 包，然后被 app 壳工程所依赖，在构

建 APP 的过程中 Gradle 会自动将重复的 arr 包排除，APP 中也就不会存在相同的代码了；但是虽然组件是不会重复了，但是我们还是要考虑另一个情况，我们在 build.gradle 中 compile 的第三方库，例如 AndroidSupport 库经常会被一些开源的控件所依赖，而我们自己一定也会 compile AndroidSupport 库 ，这就会造成第三方包和我们自己的包存在重复加载，解决办法就是找出那个多出来的库，并将多出来的库给排除掉，而且 Gradle 也是支持这样做的，分别有两种方式： 根据组件名排除或者根据包名排除，下面以排除 support-v4 库为例：

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
compile("com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"){
exclude module: 'support-v4'//根据组件名排除
exclude group:'android.support.v4'//根据包名排除
}
```

```
}
```

library 重复依赖的问题算是都解决了，但是我们在开发项目的时候会依赖很多开源库，而这
些库每个组件都需要用到，要是每个组件都去依赖一遍也是很麻烦的，尤其是给这些库升级的时候，为了方便我们统一管理第三方库，我们将给给整个工程提供统一的依赖第三方库的入口，前面介绍的 Common 库的作用之一就是统一依赖开源库，因为其他业务组件都依赖了 Common 库，所以这些业务组件也就间接依赖了 Common 所依赖的开源库。

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
//AndroidSupport
compile
"com.android.support:appcompat-v7:$rootProject.supportLibraryVersion"
compile "com.android.support:design:$rootProject.supportLibraryVersion"
compile "com.android.support:percent:$rootProject.supportLibraryVersion"
//网络请求相关
compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
compile
"com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
//稳定的
compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
compile "com.orhanobut:logger:$rootProject.loggerVersion"
compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
compile "com.google.code.gson:gson:$rootProject.gsonVersion"
compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"
compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"
//router
compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

5 ）组件之间调用和通信

在组件化开发的时候，组件之间是没有依赖关系，我们不能在使用显示调用来跳转页面了，因为我们组件化的目的之一就是解决模块间的强依赖问题，假如现在要从 A 业务组件跳转到业务 B 组件，并且要携带参数跳转，这时候怎么办呢？而且组件这么多怎么管理也是个问题，这时候就需要引入“路由”的概念了，由本文开始的组件化模型下的业务关系图可知路
由就是起到一个转发的作用。

这里我将介绍开源库的“ActivityRouter”，有兴趣的同学情直接去 ActivityRouter 的 Github 主页学习：ActivityRouter，ActivityRouter 支持给 Activity 定义 URL，这样就可以通过 URL 跳转到 Activity，并且支持从浏览器以及 APP 中跳入我们的 Activity，而且还支持通过 url

调用方法。下面将介绍如何将 ActivityRouter 集成到组件化项目中以实现组件之间的调用；

1、首先我们需要在 Common 组件中的 build.gradle 将 ActivityRouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
//router
compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

2、这一步我们需要先了解 APT 这个概念，APT(Annotation Processing Tool) 是一种处理注解的工具，它对源代码文件进行检测找出其中的 Annotation ，使用 Annotation 进行。额外的处理。 Annotation 处理器在处理 Annotation 时可以根据源文件中的 Annotation
生成额外的源文件和其它的文件(文件具体内容由 Annotation 处理器的编写者决定) ，APT
还会编译生成的源文件和原来的源文件，将它们一起生成 class 文件。在这里我们将在每一个 业务组件的 build.gradle 都引入 ActivityRouter 的 Annotation 处理器，我们将会在声明组件和 Url 的时候使用，annotationProcessor 是 Android 官方提供的 Annotation 处理器插件，代码如下：

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
}
```

3、接下来需要在 app 壳工程的 AndroidManifest.xml 配置，到这里 ActivityRouter 配置就算完成了：

```
<!--声明整个应用程序的路由协议-->
<activity
android:name="com.github.mzule.activityrouter.router.RouterActivity"
android:theme="@android:style/Theme.NoDisplay">
<intent-filter>
<action android:name="android.intent.action.VIEW"/>
<categoryandroid:name="android.intent.category.DEFAULT" />
<categoryandroid:name="android.intent.category.BROWSABLE" />
<data android:scheme="@string/global_scheme"/> <!-- 改成自己的
scheme -->
</intent-filter>
</activity>
<!--发送崩溃日志界面-->
```

4、接下来我们将声明项目中的业务组件，声明方法如下：

```
@Module("girls")public class Girls{
}
```

在每一个业务组件的 java 文件的根目录下创建一个类，用 注解@Module 声明这个业务组

件；
然后在“app 壳工程”的 应用 用 Application 中使用 注解@Modules 管理我们声明的所有业务
组件，方法如下：
@Modules({"main", "girls", "news"})public class MyApplication extends
BaseApplication {
}
到这里组件化项目中的所有业务组件就被声明和管理起来了，组件之间的也就可以互相调用了，当然前提是要给业务组件中的 Activity 定义 URL。
5、例如我们给 Girls 组件中的 GirlsActivity 使用 注解@Router 定义一个 URL：“news”，方法如下：
@Router("girls")public class GirlsActivity extends BaseActionBarActivity {
private GirlsView mView;
private GirlsContract.Presenter mPresenter;
@Override
protectedint setTitleId() {
return R.string.girls_activity_title;
}
@Override
protectedvoidonCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
mView = new GirlsView(this);
setContentView(mView);
mPresenter = new GirlsPresenter(mView);
mPresenter.start();
}
}
然后我们就可以过 在项目中的任何一个地方通过 URL 址 地址 ： ： module://girls, 调用
GirlsActivity，方法如下：
Routers.open(MainActivity.this, "module://girls");
1
组件之间的调用解决后，另外需要解决的就是组件之间的通信，例如 A 业务组件中有消息
列表 ， 而用户在 B 组件中操作某个事件后会产生一条新消息 ， 需要通知 A 组件刷新消息列
表，这样业务场景需求可以使用 Android 广播来解决，也可以使用第三方的事件总线来实现，比如 EventBus。
6 ）组件之间资源名冲突
因为我们拆分出了很多业务组件和功能组件，在把这些组件合并到“app 壳工程”时候就有可
能会出现资源名冲突问题，例如 A 组件和 B 组件都有一张叫做“ic_back” 的图标，这时候
在集成模式下打包 APP 就会编译出错，解决这个问题最简单的办法就是 在项目中约定资

源
文件命名规约，比如强制使每个资源文件的名称以组件名开始，这个可以根据实际情况和开发人员制定规则。当然了万能的 Gradle 构建工具也提供了解决方法，通过在在组件的 build.gradle 中添加如下的代码：

```
//设置了 resourcePrefix 值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
//但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源，所有图片资源仍
然需要手动去修改资源名。
resourcePrefix"girls_"
```

但是设置了这个属性后有个问题 ， 所有的资源名必须以指定的字符串做前缀 ， 否则会报错 ，
而且 resourcePrefix 这个值只能限定 xml 里面的资源 ， 并不能限定图片资源 ， 所有图片资
源仍然需要手动去修改资源名；所以我并不推荐使用这种方法来解决资源名冲突。

4 、组件化项目的工程类型

在组件化工程模型中主要有：app 壳工程、业务组件和功能组件 3 种类型，而业务组件中
的 的 Main 组件和功能组件中的 Common 组件比较特殊，下面将分别介绍。
1 ）p app 壳工程
app 壳工程是从名称来解释就是一个空壳工程，没有任何的业务代码，也不能有 Activity，但它又必须被单独划分成一个组件，而不能融合到其他组件中，是因为它有如下几点重要功能：
1、app 壳工程中声明了我们 Android 的 应用的 Application，这个 Application 必须继承自
Common 组件中的 BaseApplication（如果你无需实现自己的 Application 可以直接在表单声明 BaseApplication），因为只有这样，在打包应用后才能让 BaseApplication 中的 Context 生效，当然你还可以在这个 Application 中初始化我们工程中使用到的库文件，还可以在这里解决 Android 引用方法数不能超过 65535 的限制，对崩溃事件的捕获和发送也可以在这里声明。
2、app 的 壳工程的 AndroidManifest.xml 是我 Android 应用的根表单，应用的名称、图标
以及是否支持备份等等属性都是在这份表单中配置的，其他组件中的表单最终在集成开发模式下都被合并到这份 AndroidManifest.xml 中。
3、app 的 壳工程的 build.gradle 是比较特殊的，app 壳不管是在集成开发模式还是组件开
发模式，它的属性始终都是：com.android.application，因为最终其他的组件都要被 app 壳工程所依赖，被打包进 app 壳工程中，这一点从组件化工程模型图中就能体现出来，所以 app 壳工程是不需要单独调试单独开发的。另外 Android 应用的打包签名，以及 buildTypes 和 defaultConfig 都需要在这里配置，而它的 dependencies 则需要根据 isModule 的值分别依赖不同的组件，在组件开发模式下 app 壳工程只需要依赖 Common 组件，或者为了防止
报错也可以根据实际情况依赖其他功能组件，而在集成模式下 app 壳工程必须依赖所有在

应用 Application 中声明的业务组件，并且不需要再依赖任何功能组件。

份 下面是一份 app 程 壳工程 的 的 build.gradle 文件：

```
apply plugin: 'com.android.application'
static def buildTime() {
return new Date().format("yyyyMMdd");
}
android {
signingConfigs{
release {
keyAlias 'guiying712'
keyPassword 'guiying712'
storeFilefile('/mykey.jks')
storePassword 'guiying712'
}
}
compileSdkVersion rootProject.ext.compileSdkVersion
buildToolsVersion rootProject.ext.buildToolsVersion
defaultConfig {
applicationId "com.guiying.androidmodulepattern"
minSdkVersion rootProject.ext.minSdkVersion
targetSdkVersion rootProject.ext.targetSdkVersion
versionCode rootProject.ext.versionCode
versionName rootProject.ext.versionName
multiDexEnabled false
//打包时间
resValue "string", "build_time", buildTime()
}
buildTypes {
release {
//更改 AndroidManifest.xml 中预先定义好占位符信息
//manifestPlaceholders =[app_icon: "@drawable/icon"]
// 不显示 Log
buildConfigField "boolean", "LEO_DEBUG", "false"
//是否 zip 对齐
zipAlignEnabled true
// 缩减 resource 文件
shrinkResources true
//Proguard
minifyEnabled true
proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
//签名
signingConfig signingConfigs.release
}
```

```
debug {
//给 applicationId 添加后缀“.debug”
applicationIdSuffix".debug"
//manifestPlaceholders =[app_icon: "@drawable/launch_beta"]
buildConfigField "boolean", "LOG_DEBUG", "true"
zipAlignEnabled false
shrinkResources false
minifyEnabled false
debuggable true
}
}
}
dependencies {
compile fileTree(dir: 'libs',include: ['*.jar'])
annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
if (isModule.toBoolean()) {
compile project(':lib_common')
} else {
compile project(':module_main')
compile project(':module_girls')
compile project(':module_news')
}
}
```

2 ）功能组件和 n Common 组件

功能组件是为了支撑业务组件的某些功能而独立划分出来的组件，功能实质上跟项目中引入的第三方库是一样的，功能组件的特征如下：

1、功能组件的 AndroidManifest.xml 是一张空表，这张表中只有功能组件的包名；

2、功能组件不管是在集成开发模式下还是组件开发模式下属性始终是：com.android.library，所以功能组件是不需要读取 gradle.properties 中的 isModule 值的；另外功能组件的 build.gradle 也无需设置 buildTypes，只需要 dependencies 这个功能组件需要的 jar 包和开源库。

份 下面是一份 通 普通 的 的功能组件的 build.gradle 文件：

```
apply plugin: 'com.android.library'
android {
compileSdkVersion rootProject.ext.compileSdkVersion
buildToolsVersion rootProject.ext.buildToolsVersion
defaultConfig {
minSdkVersion rootProject.ext.minSdkVersion
targetSdkVersion rootProject.ext.targetSdkVersion
versionCode rootProject.ext.versionCode
versionName rootProject.ext.versionName
}
}
```

```
dependencies {
compile fileTree(dir: 'libs',include: ['*.jar'])
}
```

Common 组件除了有功能组件的普遍属性外，还具有其他功能：

1、Common 组件的 AndroidManifest.xml 不是一张空表，这张表中声明了我们 Android 应用用到的所有使用权限 uses-permission 和 uses-feature，放到这里是因为在组件开发模式下，所有业务组件就无需在自己的 AndroidManifest.xm 声明自己要用到的权限了。

2、Common 组件的 build.gradle 需要统一依赖业务组件中用到的 第三方依赖库和 jar 包，例如我们用到的 ActivityRouter、Okhttp 等等。

3、Common 组件中封装了 Android 应用的 Base 类和网络请求工具、图片加载工具等等，公用的 widget 控件也应该放在 Common 组件中；业务组件中都用到的数据也应放于 Common 组件中，例如保存到 SharedPreferences 和 DataBase 中的登陆数据；

4、Common 组件的资源文件中需要放置项目公用的 Drawable、layout、sting、dimen、color 和 style 等等，另外项目中的 Activity 主题必须定义在 Common 中，方便和 BaseActivity 配合保持整个 Android 应用的界面风格统一。

份 下面是一份 Common 的 功能组件的 build.gradle 文件：

```
apply plugin: 'com.android.library'
android {
compileSdkVersion rootProject.ext.compileSdkVersion
buildToolsVersion rootProject.ext.buildToolsVersion
defaultConfig {
minSdkVersion rootProject.ext.minSdkVersion
targetSdkVersion rootProject.ext.targetSdkVersion
versionCode rootProject.ext.versionCode
versionName rootProject.ext.versionName
}
}
dependencies {
compile fileTree(dir: 'libs',include: ['*.jar'])
//AndroidSupport
compile
"com.android.support:appcompat-v7:$rootProject.supportLibraryVersion"
compile "com.android.support:design:$rootProject.supportLibraryVersion"
compile "com.android.support:percent:$rootProject.supportLibraryVersion"
//网络请求相关
compile "com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
compile
"com.squareup.retrofit2:retrofit-mock:$rootProject.retrofitVersion"
compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
//稳定的
compile "com.github.bumptech.glide:glide:$rootProject.glideVersion"
compile "com.orhanobut:logger:$rootProject.loggerVersion"
compile "org.greenrobot:eventbus:$rootProject.eventbusVersion"
```

```
compile "com.google.code.gson:gson:$rootProject.gsonVersion"
compile "com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"
compile "com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion"
compile "com.github.GrenderG:Toasty:$rootProject.toastyVersion"
//router
compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

2 ）业务组件和 n Main 组件

业务组件就是根据业务逻辑的不同拆分出来的组件，业务组件的特征如下：

1、业务组件中要有两张 AndroidManifest.xml，分别对应组件开发模式和集成开发模式，这两张表的区别请查看 组件之间 AndroidManifest 合并问题 小节。

2、业务组件在集成模式下是不能有自己的 Application 的，但在组件开发模式下又必须实现

自己的 Application 并且要继承自 Common 组件的 BaseApplication，并且这个 Application 不能被业务组件中的代码引用，因为它的功能就是为了使业务组件从 BaseApplication 中获取的全局 Context 生效，还有初始化数据之用。

3、业务组件有 debug 文件夹，这个文件夹在集成模式下会从业务组件的代码中排除掉，所以 debug 文件夹中的类不能被业务组件强引用，例如组件模式下的 Application 就是置于这个文件夹中，还有组件模式下开发给目标 Activity 传递参数的用的 launch Activity 也应该置于 debug 文件夹中；

4、业务组件必须在自己的 Java 文件夹中创建业务组件声明类，以使 app 程 壳工程 的 中的 应

用 用 Application 能够引用，实现组件跳转，具体请查看 组件之间调用和通信 小节；

5、的 业务组件必须在自己的 build.gradle 据 中根据 isModule 值的不同改变自己的属性，在

组件模式下是：com.android.application，而在集成模式下 com.android.library；同时还需要在 build.gradle 配置资源文件，如 指定不同开发模式下的 AndroidManifest.xml 文件路径，排除 debug 文件夹等；业务组件还必须在 dependencies 中依赖 Common 组件，并且引入 ActivityRouter 的注解处理器 annotationProcessor，以及依赖其他用到的功能组件。

的 下面是一份普通业务组件的 build.gradle 文件：

```
if (isModule.toBoolean()) {
apply plugin: 'com.android.application'
} else {
apply plugin: 'com.android.library'
}
android {
compileSdkVersion rootProject.ext.compileSdkVersion
buildToolsVersion rootProject.ext.buildToolsVersion
defaultConfig {
minSdkVersion rootProject.ext.minSdkVersion
targetSdkVersion rootProject.ext.targetSdkVersion
versionCode rootProject.ext.versionCode
versionName rootProject.ext.versionName
```

```
}
sourceSets {
main{
if (isModule.toBoolean()) {
manifest.srcFile 'src/main/module/AndroidManifest.xml'
} else {
manifest.srcFile 'src/main/AndroidManifest.xml'
//集成开发模式下排除 debug 文件夹中的所有 Java 文件
java{
exclude 'debug/**'
}
}
}
}
//设置了 resourcePrefix 值后，所有的资源名必须以指定的字符串做前缀，否则会报错。
//但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源，所有图片资源仍
然需要手动去修改资源名。
//resourcePrefix "girls_"
}
dependencies {
compile fileTree(dir: 'libs',include: ['*.jar'])
annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
compile project(':lib_common')
}
```

Main 组件除了有业务组件的普遍属性外，还有一项重要功能：

1、Main 组件集成模式下的 AndroidManifest.xml 是跟其他业务组件不一样的，Main 组件的

表单中声明了我们整个 Android 应用的 launch Activity，这就是 Main 组件的独特之处；所以我建议 SplashActivity、登陆 Activity 以及主界面都应属于 Main 组件，也就是说 Android 应用启动后要调用的页面应置于 Main 组件。

```
<activity
android:name=".splash.SplashActivity"
android:launchMode="singleTop"
android:screenOrientation="portrait"
android:theme="@style/SplashTheme">
<intent-filter>
<action android:name="android.intent.action.MAIN"/>
<categoryandroid:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

5 、组件化项目的混淆方案

组件化项目的 Java 代码混淆方案采用在集成模式下集中在 app 壳工程中混淆，各个业务
组件不配置混淆文件。集成开发模式下在 app 壳工程中 build.gradle 文件的 release 构建类
型中开启混淆属性，其他 buildTypes 配置方案跟普通项目保持一致，Java 混淆配置文件也放置在 app 壳工程中，各个业务组件的混淆配置规则都应该在 app 壳工程中的混淆配置文
件中添加和修改。
之所以不采用在每个业务组件中开启混淆的方案，是因为 被 组件在集成模式下都被 Gradle
了 构建成了 release 类型的 arr 包，一旦业务组件的代码被混淆，而这时候代码中又出现了
bug，将很难根据日志找出导致 bug 的原因；另外每个业务组件中都保留一份混淆配置文件非常不便于修改和管理，这也是不推荐在业务组件的 build.gradle 文件中配置 buildTypes（构建类型）的原因。

6 、工程的 e build.gradle 和 和 s gradle.properties 文件

1 ）组件化工程的 e build.gradle 文件
在组件化项目中因为每个组件的 build.gradle 都需要配置 compileSdkVersion、buildToolsVersion 和 defaultConfig 等的版本号，而且每个组件都需要用到 annotationProcessor，的 为了能够使组件化项目中的所有组件的 build.gradle 中的这些配置
都能保持统一，并且也是为了方便修改版本号，我们统一在 Android 工程根目录下的 build.gradle 中定义这些版本号，当然为了方便管理 Common 组件中的第三方开源库的版本号，最好也在这里定义这些开源库的版本号，然后在各个组件的 build.gradle 中引用 Android 工程根目录下的 build.gradle 定义的版本号，组件化工程的 build.gradle 文件代码如下：

```
buildscript {
repositories {
jcenter()
mavenCentral()
}
dependencies {
//classpath"com.android.tools.build:gradle:$localGradlePluginVersion"
//$localGradlePluginVersion 是 gradle.properties 中的数据
classpath"com.android.tools.build:gradle:$localGradlePluginVersion"
}
}
allprojects {
repositories {
jcenter()
mavenCentral()
//Add theJitPack repository
```

```
maven { url "https://jitpack.io" }
//支持 arr 包
flatDir {
dirs'libs'
}
}
}
taskclean(type: Delete){
delete rootProject.buildDir
}
// Define versions in a single place//时间：2017.2.13；每次修改版本号都要添加修改时间
ext {
// Sdk and tools
//localBuildToolsVersion 是 gradle.properties 中的数据
buildToolsVersion =localBuildToolsVersion
compileSdkVersion =25
minSdkVersion = 16
targetSdkVersion = 25
versionCode = 1
versionName = "1.0"
javaVersion = JavaVersion.VERSION_1_8
// App dependenciesversion
supportLibraryVersion = "25.3.1"
retrofitVersion = "2.1.0"
glideVersion ="3.7.0"
loggerVersion = "1.15"
eventbusVersion = "3.0.0"
gsonVersion = "2.8.0"
photoViewVersion = "2.0.0"
//需检查升级版本
annotationProcessor= "1.1.7"
routerVersion = "1.2.2"
easyRecyclerVersion= "4.4.0"
cookieVersion = "v1.0.1"
toastyVersion = "1.1.3"
}
```

2 ）组件化工程的 s gradle.properties 文件

在组件化实施流程中我们了解到 gradle.properties 有两个属性对我们非常有用：

1、在 Android 项目中的任何一个 build.gradle 文件中都可以把 gradle.properties 中的常量读取出来，不管这个 build.gradle 是组件的还是整个项目工程的 build.gradle；

2、gradle.properties 中的数据类型都是 String 类型，使用其他数据类型需要自行转换；

利用 gradle.properties 的属性不仅可以解决集成开发模式和组件开发模式的转换，而且还可以解决在多人协同开发 Android 项目的时候，因为开发团队成员的 Android 开发环境（开

发
环境指 Android SDK 和 AndroidStudio）不一致而导致频繁改变线上项目的 build.gradle 配置。
在每个 Android 组件的 build.gradle 中有一个属性：buildToolsVersion，表示构建工具的版本号，这个属性值对应 AndroidSDK 中的 Android SDK Build-tools，正常情况下 build.gradle 中的 buildToolsVersion 跟你电脑中 Android SDK Build-tools 的最新版本是一致的，比如现在 Android SDK Build-tools 的最新的版本是：25.0.3，那么我的 Android 项目中 build.gradle 中的 buildToolsVersion 版本号也是 25.0.3，但是一旦一个 Android 项目是由好几个人同时开发，总会出现每个人的开发环境 Android SDK Build-tools 是都是不一样的，并不是所有人都会经常升级更新 Android SDK，而且代码是保存到线上环境的（例如使用 SVN/Git 等工具），某个开发人员提交代码后线上 Android 项目中 build.gradle 中的 buildToolsVersion 也会被不断地改变。
另外一个原因是因为 Android 工程的根目录下的 build.gradle 声明了 Android Gradle 构建工具，而这个工具也是有版本号的，而且 Gradle Build Tools 的版本号跟 AndroidStudio 版本号一致的，但是有些开发人员基本很久都不会升级自己的 AndroidStudio 版本，导致团队中每个开发人员的 Gradle Build Tools 的版本号也不一致。
如果每次同步代码后这两个工具的版本号被改变了，开发人员可以自己手动改回来，并且不要把改动工具版本号的代码提交到线上环境，这样还可以勉强继续开发；但是很多公司都会使用持续集成工具（例如 Jenkins）用于持续的软件版本发布，而 Android 出包是需要 Android SDK Build-tools 和 GradleBuild Tools 配合的，一旦提交到线上的版本跟持续集成工具所依赖的 Android 环境构建工具版本号不一致就会导致 Android 打包失败。
为了解决上面问题就必须将 Android 项目中 build.gradle 中的 buildToolsVersion 和 GradleBuildTools 版本号从线上代码隔离出来，保证线上代码的 buildToolsVersion 和 Gradle Build Tools 版本号不会被人为改变。
具体的实施流程大家可以查看我的这篇博文：AndroidStudio 本地化配置 gradle 的 buildToolsVersion 和 gradleBuildTools

7 、组件化项目 r Router 的其他方案 -ARouter

在组件化项目中使用到的跨组件跳转库 ActivityRouter 可以使用阿里巴巴的开源路由项目：阿里巴巴 ARouter；
ActivityRouter 和 ARouter 的接入组件化项目的方式是一样的，ActivityRouter 提供的功能目前 ARouter 也全部支持，但是 ARouter 还支持依赖注入解耦，页面、拦截器、服务等组件均会自动注册到框架。对于大家来说，没有最好的只有最适合的，大家可以根据自己的项目选择合适的 Router。
下面将介绍 ARouter 的基础使用方法，更多功能还需大家去 Github 自己学习；
1、首先 ARouter 这个框架是需要初始化 SDK 的，所以你需要在“app 壳工程”中的应用 Application 中加入下面的代码，在 注意：在 debug 要 模式下一定要 openDebug：

```
if (BuildConfig.DEBUG) {
//一定要在 ARouter.init 之前调用 openDebug
ARouter.openDebug();
ARouter.openLog();
}
ARouter.init(this);
```

2、首先我们依然需要在 Common 组件中的 build.gradle 将 ARouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
//router
compile 'com.alibaba:arouter-api:1.2.1.1'
}
```

3、然后在每一个 业务组件的 build.gradle 都引入 ARouter 的 Annotation 处理器，代码如下：

```
android {
defaultConfig {
...
javaCompileOptions {
annotationProcessorOptions {
arguments = [ moduleName : project.getName() ]
}
}
}
}
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
annotationProcessor'com.alibaba:arouter-compiler:1.0.3'
}
```

4、 由于 ARouter 支持自动注册到框架，所以我们不用像 ActivityRouter 那样在各个组件
中声明组件，当然更不需要在 Application 中管理组件了。 我们给 Girls 组件 中的 GirlsActivity 添加注解：@Route(path = “/girls/list”)，需要注意的是 这里的路径至少需要有两级 ，/xx/xx，之所以这样是因为 ARouter 使用了路径中第一段字符串(/*/) 作为分组，比
如像上面的”girls”,而分组这个概念就有点类似于 ActivityRouter 中的组件声明 @Module ，代码如下：

```
@Route(path = "/girls/list")publicclass GirlsActivityextends
BaseActionBarActivity {
private GirlsView mView;
private GirlsContract.Presenter mPresenter;
@Override
protectedint setTitleId() {
return R.string.girls_activity_title;
}
@Override
protectedvoidonCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
mView = new GirlsView(this);
setContentView(mView);
```

```
mPresenter = new GirlsPresenter(mView);
mPresenter.start();
}
}
```

然后我们就可以过 在项目中的任何一个地方通过 URL 址 地址 ： ： /girls/list, 调用 GirlsActivity，方法如下：

```
ARouter.getInstance().build("/girls/list").navigation();
```

. 16. 系统打包流程

一、添加 android 平台

添加之后，在项目目录的 platforms 下会生成一个 android 文件夹。ionic cordova platform add android

二、cordova 编译应用

使用 build 命令编译应用的发布版本，这个过程需要你的 android sdk 和环境变量、java jdk 和环境变量、android 的 gradle 配置没有错误。ionic cordova build android --prod --release
编译成功之后，在项目路径 platforms/android/build/outputs/apk/android-release-unsigned.apk 未签名文件，这个时候的 apk 还不能被安装到手机上。

三、生成签名文件

```
keytool -genkey -v -keystore jhy-release-key.jks -keyalg RSA
-keysize 2048 -validity 10000 -alias alias_jhy
```

输入的密码要记住，其他姓名地区等信息随便填吧，最好还是记住，成功之后在主目录下就生成了 jhy-release-key.keystore 文件，命令中 jhy-release-key.keystore 是生成文件的名字，alias_jhy 是别名，随便起但是要记住，一会签名要用到，其他信息如加密、有效日期等就不说了，无需改动。
生成后会提示:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore jhy-release-key.jks -destkeystore jhy-release-key.jks -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。执行命令:keytool -importkeystore -srckeystore jhy-release-key.jks -destkeystore jhy-release-key.jks -deststoretype pkcs12
执行结果:Warning: 已将 "jhy-release-key.jks" 迁移到 Non JKS/JCEKS。将 JKS 密钥库作为 "jhy-release-key.jks.old" 进行了备份。

四、签名应用文件

把在第二步生成的 android-release-unsigned.apk 拷贝到与生成的
jhy-release-key.jks 同一目录下，也就是项目的主目录下，执行命令:
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
jhy-release-key.jks android-release-unsigned.apk alias_jhy
输入签名文件的密码，成功之后主目录下的
android-release-unsigned.apk 就被签名成功了，会比原来未被签名的
apk 文件大一点，能够安装到手机或 android 虚拟机上了。
签名完成后会提示没有时间戳，忽略即可
检测是否签名成功:
apksigner verify android-release-unsigned.apk
也可用以下命令签名并生成新 apk 文件
jarsigner -verbose -keystore jhy-release-key.jks -signedjar
notepad.apk android-release-unsigned.apk alias_jhy

17. Android 有哪些存储数据的方式。

实现数据存储的 5 种方式。

使用 SharedPreferences 存储数据

SharedPreferences 是 Android 平台上一个轻量级的存储类，主要是保存一些常用的配置比如窗口状态，一般在
onSaveInstance State 保存一般使用 SharedPreferences 完成，它提供了 Android 平台常规的 Long 长 整形、
的保存。
它是什么样的处理方式呢？SharedPreferences 类似过去 Windows 系统上的 ini 配置文件，但是它分为多种权限
android123 提示最终是以 xml 方式来保存，整体效率来看不是特别的高，对于常规的轻量级而言比 SQLite 要好
可以考虑自己定义文件格式。xml 处理时 Dalvik 会通过自带底层的本地 XML Parser 解析，比如 XMLpull 方式，
较好。
它的本质是基于 XML 文件存储 key-value 键值对数据，通常用来存储一些简单的配置信息。
其存储位置在/data/data/<包名>/shared_prefs 目录下。
SharedPreferences 对象本身只能获取数据而不支持存储和修改，存储修改是通过 Editor 对象实现。
实现 SharedPreferences 存储的步骤如下：
1.根据 Context 获取 SharedPreferences 对象
2.利用 edit()方法获取 Editor 对象。

3.通过 Editor 对象存储 key-value 键值对数据。
4.通过 commit()方法提交数据。
下面是示例代码：

```
1
10
11
12
13
public class MainActivity extends Activity {
@Override
publicvoid onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
//获取 SharedPreferences 对象
Context ctx = MainActivity.this;
SharedPreferences sp = ctx.getSharedPreferences("SP", MODE_PRIVATE);
//存入数据
Editor editor =sp.edit();
editor.putString("STRING_KEY", "string");
editor.putInt("INT_KEY", 0);
14
15
16
17
18
19
20
21
22
editor.putBoolean("BOOLEAN_KEY", true);
editor.commit();
//返回 STRING_KEY 的值
Log.d("SP", sp.getString("STRING_KEY", "none"));
//如果 NOT_EXIST 不存在，则返回值为"none"
Log.d("SP", sp.getString("NOT_EXIST", "none"));
}
}
```

这段代码执行过后，即在/data/data/com.test/shared_prefs 目录下生成了一个 SP.xml 文件，一个应用可以创建
SharedPreferences 对象与 SQLite 数据库相比，免去了创建数据库，创建表，写 SQL 语句等诸多操作，相对而言
SharedPreferences 也有其自身缺陷，比如其职能存储 boolean，int，float，long 和 String 五种简单的数据类
询等。所以不论 SharedPreferences 的数据存储操作是如何简单，它也只能是存储方式的一

种补充，而无法完全
的其他数据存储方式。

文件存储数据

关于文件存储，Activity 提供了 openFileOutput()方法可以用于把数据输出到文件中，具体的实现过程与在 J2SE
一样的。
文件可用来存放大量数据，如文本、图片、音频等。
默认位置：/data/data/<包>/files/***.***。
代码示例：

```
1
2
12
13
14
public void save()
{
try{
FileOutputStream outStream=this.openFileOutput("a.txt",Context.MODE_WORLD_READABLE);
outStream.write(text.getText().toString().getBytes());
outStream.close();
Toast.makeText(MyActivity.this,"Saved",Toast.LENGTH_LONG).show();
} catch (FileNotFoundException e) {
return;
}
catch(IOException e){
return ;
}
}
```

openFileOutput()方法的第一参数用于指定文件名称，不能包含路径分隔符“/” ，如果文件不存在，Android
创建的文件保存在/data/data//files 目录，如： /data/data/cn.itcast.action/files/itcast.txt ，通过点击 Eclipse
View”-“Other”，在对话窗口中展开 android 文件夹，选择下面的 File Explorer 视图，然后在 File Explorer 视
目录就可以看到该文件。
openFileOutput()方法的第二参数用于指定操作模式，有四种模式，分别为：
Context.MODE_PRIVATE = 0
Context.MODE_APPEND= 32768
Context.MODE_WORLD_READABLE = 1
Context.MODE_WORLD_WRITEABLE = 2
Context.MODE_PRIVATE：为默认操作模式，代表该文件是私有数据，只能被应用本身访问，在该模式下，写入的

如果想把新写入的内容追加到原文件中。可以使用 Context.MODE_APPEND
Context.MODE_APPEND：模式会检查文件是否存在，存在就往文件追加内容，否则就创建新文件。
Context.MODE_WORLD_READABLE 和 Context.MODE_WORLD_WRITEABLE 用来控制其他应用是否有权限读
MODE_WORLD_READABLE：表示当前文件可以被其他应用读取;
MODE_WORLD_WRITEABLE：表示当前文件可以被其他应用写入。
如果希望文件被其他应用读和写，可以传入： openFileOutput("itcast.txt", Context.MODE_WORLD_READAB
Context.MODE_WORLD_WRITEABLE); android 有一套自己的安全模型，当应用程序(.apk)在安装时系统就会分
应用要去访问其他资源比如文件的时候，就需要 userid 匹配。默认情况下，任何应用创建的文件，sharedprefer
有的(位于/data/data//files)，其他程序无法访问。
除非在创建时指定了 Context.MODE_WORLD_READABLE 或者 Context.MODE_WORLD_WRITEABLE ，只有这
读取文件示例：

```
1
2
3
4
5
6
7
8
9
public void load()
{
try {
FileInputStream inStream=this.openFileInput("a.txt");
ByteArrayOutputStream stream=new ByteArrayOutputStream();
byte[] buffer=new byte[1024];
int length=-1;
while((length=inStream.read(buffer))!=-1) {
stream.write(buffer,0,length);
}
stream.close();
inStream.close();
text.setText(stream.toString());
Toast.makeText(MyActivity.this,"Loaded",Toast.LENGTH_LONG).show();
} catch (FileNotFoundException e) {
e.printStackTrace();
}
catch (IOException e){
```

return ;
}
}
对于私有文件只能被创建该文件的应用访问，如果希望文件能被其他应用读和写，可以在创建文件时，指定
Context.MODE_WORLD_READABLE 和 Context.MODE_WORLD_WRITEABLE 权限。
Activity 还提供了 getCacheDir()和 getFilesDir()方法： getCacheDir()方法用于获取 /data/data//cache 目录 g
/data/data//files 目录。
把文件存入 SDCard：
使用 Activity 的 openFileOutput()方法保存文件，文件是存放在手机空间上，一般手机的存储空间不是很大，存
放像视频这样的大文件，是不可行的。对于像视频这样的大文件，我们可以把它存放在 SDCard。
SDCard 是干什么的?你可以把它看作是移动硬盘或 U 盘。 在模拟器中使用 SDCard，你需要先创建一张 SDCard
只是镜像文件)。
创建 SDCard 可以在 Eclipse 创建模拟器时随同创建，也可以使用 DOS 命令进行创建，如下： 在 Dos 窗口中进入
tools 目录，输入以下命令创建一张容量为 2G 的 SDCard，文件后缀可以随便取，建议使用.img： mksdcard 2
D:\AndroidTool\sdcard.img 在程序中访问 SDCard，你需要申请访问 SDCard 的权限。
在 AndroidManifest.xml 中加入访问 SDCard 的权限如下:
1
2
3
4
5
<!-- 在 SDCard 中创建与删除文件权限 -->
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<!-- 往 SDCard 写入数据权限 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
要往 SDCard 存放文件，程序必须先判断手机是否装有 SDCard，并且可以进行读写。
注意：访问 SDCard 必须在 AndroidManifest.xml 中加入访问 SDCard 的权限。
1
2
3
4
5
6
7
if(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)){
File sdCardDir = Environment.getExternalStorageDirectory();//获取 SDCard 目录
File saveFile = new File(sdCardDir, “a.txt”);

```
FileOutputStream outStream = new FileOutputStream(saveFile);
outStream.write("test".getBytes());
outStream.close();
}
```

Environment.getExternalStorageState()方法用于获取 SDCard 的状态，如果手机装有 SDCard，并且可以进行
等于 Environment.MEDIA_MOUNTED。
Environment.getExternalStorageDirectory()方法用于获取 SDCard 的目录，当然要获取 SDCard 的目录，你也

```
File sdCardDir = new File("/sdcard"); //获取 SDCard 目录
File saveFile = new File(sdCardDir, "itcast.txt");
//上面两句代码可以合成一句：
File saveFile = new File("/sdcard/a.txt");
FileOutputStream outStream = new FileOutputStream(saveFile);
outStream.write("test".getBytes());
outStream.close();
```

SQLite 数据库存储数据

SQLite 是轻量级嵌入式数据库引擎，它支持 SQL 语言，并且只利用很少的内存就有很好的性能。此外它还是开源
许多开源项目((Mozilla, PHP, Python)都使用了 SQLite.SQLite 由以下几个组件组成：SQL 编译器、内核、后端
用虚拟机和虚拟数据库引擎(VDBE)，使调试、修改和扩展 SQLite 的内核变得更加方便。
特点：
面向资源有限的设备，
没有服务器进程,
所有数据存放在同一文件中跨平台，
可自由复制。
SQLite 内部结构：
SQLite 基本上符合 SQL-92 标准，和其他的主要 SQL 数据库没什么区别。它的优点就是高效，Android 运行时环
SQLite 和其他数据库最大的不同就是对数据类型的支持，创建一个表时，可以在 CREATE TABLE 语句中指定某
把任何数据类型放入任何列中。当某个值插入数据库时，SQLite 将检查它的类型。如果该类型与关联的列不匹配
转换成该列的类型。如果不能转换，则该值将作为其本身具有的类型存储。比如可以把一个字符串(String)放入 IN
为“弱类型”(manifest typing.)。 此外，SQLite 不支持一些标准的 SQL 功能，特别是外键约束(FOREIGN KE
transcaction 和 RIGHT OUTER JOIN 和 FULL OUTER JOIN, 还有一些 ALTER TABLE 功能。 除了上述功能外
SQL 系统，拥有完整的触发器，交易等等。
Android 集成了 SQLite 数据库 Android 在运行时(run-time)集成了 SQLite，所以每个 Android

应用程序都

对于熟悉 SQL 的开发人员来时，在 Android 开发中使用 SQLite 相当简单。但是，由于 JDBC 会消耗太多的系

手机这种内存受限设备来说并不合适。因此，Android 提供了一些新的 API 来使用 SQLite 数据库，Android

这些 API。

数据库存储在 data/< 项目文件夹 >/databases/ 下。Android 开发中使用 SQLite 数据库 Activites 可以通

Service 访问一个数据库。

下面会详细讲解如果创建数据库，添加数据和查询数据库。创建数据库 Android 不自动提供数据库。在 Android

必须自己创建数据库，然后创建表、索引，填充数据。

Android 提供了 SQLiteOpenHelper 帮助你创建一个数据库，你只要继承 SQLiteOpenHelper 类，就可以轻松

SQLiteOpenHelper 类根据开发应用程序的需要，封装了创建和更新数据库使用的逻辑。

SQLiteOpenHelper 的子类，至少需要实现三个方法：

1 构造函数，调用父类 SQLiteOpenHelper 的构造函数。这个方法需要四个参数：上下文环境(例如，一个 Acti

选的游标工厂(通常是 Null)，一个代表你正在使用的数据库模型版本的整数。

2onCreate()方法，它需要一个 SQLiteDatabase 对象作为参数，根据需要对这个对象填充表和初始化数据。

3onUpgrage() 方法，它需要三个参数，一个 SQLiteDatabase 对象，一个旧的版本号和一个新的版本号，这样

数据库从旧的模型转变到新的模型。

下面示例代码展示了如何继承 SQLiteOpenHelper 创建数据库：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
public class DatabaseHelper extends SQLiteOpenHelper {
```

```
DatabaseHelper(Context context, String name, CursorFactory cursorFactory, int version)
{
super(context, name, cursorFactory, version);
}
@Override
publicvoid onCreate(SQLiteDatabase db) {
// TODO 创建数据库后，对数据库的操作
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
// TODO 更改数据库版本的操作
}
@Override
public void onOpen(SQLiteDatabase db) {
19
20
21
22
super.onOpen(db);
// TODO 每次成功打开数据库后首先被执行
}
}
```

接下来讨论具体如何创建表、插入数据、删除表等等。调用 getReadableDatabase() 或 getWriteableDatabas

SQLiteDatabase 实例，具体调用那个方法，取决于你是否需要改变数据库的内容：

```
1
2
db=(new DatabaseHelper(getContext())).getWritableDatabase();
return (db == null) ? false : true;
```

上面这段代码会返回一个 SQLiteDatabase 类的实例，使用这个对象，你就可以查询或者修改数据库。 当你完成

的 Activity 已经关闭)，需要调用 SQLiteDatabase 的 Close() 方法来释放掉数据库连接。 创建表和索引 为了

SQLiteDatabase 的 execSQL() 方法来执行 DDL 语句。如果没有异常，这个方法没有返回值。例如，你可以执行如下代码：

```
1db.execSQL("CREATE TABLE mytable (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, value R
```

这条语句会创建一个名为 mytable 的表，表有一个列名为 _id，并且是主键，这列的值是会自动增长的整数(例如

会给这列自动赋值)，另外还有两列：title(字符) 和 value(浮点数)。SQLite 会自动为主键列创建索引。 通

库时创建了表和索引。

如果你不需要改变表的 schema，不需要删除表和索引 . 删除表和索引，需要使用 execSQL() 方法调用 DROP IN

句。 给表添加数据 上面的代码，已经创建了数据库和表，现在需要给表添加数据。有两种方法可以给表添加数据
像上面创建表一样，你可以使用 execSQL() 方法执行 INSERT, UPDATE, DELETE 等语句来更新表的数据。exec
返回结果的 SQL 语句。
例如： db.execSQL("INSERT INTO widgets (name, inventory)"+ "VALUES ('Sprocket',5)");
另一种方法是使用 SQLiteDatabase 对象的 insert(), update(), delete() 方法。这些方法把 SQL 语句的一部分
示例如下：

```
ContentValues cv=new ContentValues();
cv.put(Constants.TITLE, "example title");
cv.put(Constants.VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
db.insert("mytable", getNullColumnHack(), cv);
```

update()方法有四个参数，分别是表名，表示列名和值的 ContentValues 对象，可选的 WHERE 条件和可选的
串，这些字符串会替换 WHERE 条件中的“?”标记。
update() 根据条件，更新指定列的值，所以用 execSQL() 方法可以达到同样的目的。 WHERE 条件和其参数和
似。
例如：

```
String[] parms=new String[] {"this is a string"};
db.update("widgets", replacements, "name=?", parms);
```

delete() 方法的使用和 update() 类似，使用表名，可选的 WHERE 条件和相应的填充 WHERE 条件的字符串。
UPDATE, DELETE，有两种方法使用 SELECT 从 SQLite 数据库检索数据。
1.使用 rawQuery() 直接调用 SELECT 语句; 使用 query() 方法构建一个查询。
Raw Queries 正如 API 名字，rawQuery()是最简单的解决方法。通过这个方法你就可以调用 SQL SELECT 语
例如： Cursor c=db.rawQuery("SELECTname FROMsqlite_master WHERE type='table' AND name='my
在上面例子中，我们查询 SQLite 系统表(sqlite_master)检查 table 表是否存在。返回值是一个 cursor 对象，
询结果。 如果查询是动态的，使用这个方法就会非常复杂。
例如，当你需要查询的列在程序编译的时候不能确定，这时候使用 query() 方法会方便很多。
Regular Queries query() 方法用 SELECT 语句段构建查询。SELECT 语句内容作为 query() 方法的参数，比如
字段名，WHERE 条件，包含可选的位置参数，去替代 WHERE 条件中位置参数的值，GROUP BY 条件，HAVI
他参数可以是 null。所以，以前的代码段可以可写成：

```
1
2
3
String[] columns={"ID", "inventory"};
String[] parms={"snicklefritz"};
```

Cursor result=db.query("widgets", columns, "name=?",parms, null, null, null);

使用游标

不管你如何执行查询，都会返回一个 Cursor，这是 Android 的 SQLite 数据库游标，

使用游标，你可以：

通过使用 getCount() 方法得到结果集中有多少记录;

通过 moveToFirst(), moveToNext(), 和 isAfterLast() 方法遍历所有记录;

通过 getColumnNames() 得到字段名;

通过 getColumnIndex() 转换成字段号;

通过 getString()，getInt()等方法得到给定字段当前记录的值;

通过 requery() 方法重新执行查询得到游标;

通过 close() 方法释放游标资源;

例如，下面代码遍历 mytable 表：

1

2

3

4

5

6

7

8

9

10

```
Cursor result=db.rawQuery("SELECT ID, name, inventory FROM mytable");
result.moveToFirst();
while (!result.isAfterLast()) {
int id=result.getInt(0);
String name=result.getString(1);
int inventory=result.getInt(2);
// do something useful with these
result.moveToNext();
}
result.close();
```

在 Android 中使用 SQLite 数据库管理工具 在其他数据库上作开发，一般都使用工具来检查和处理数据库的内的 API。

使用 Android 模拟器，有两种可供选择的方法来管理数据库。

首先，模拟器绑定了 sqlite3 控制台程序，可以使用 adb shell 命令来调用他。只要你进入了模拟器的 shell，在

命令就可以了。

数据库文件一般存放在：/data/data/your.app.package/databases/your-db-name 如果你喜欢使用更友好的

到你的开发机上，使用 SQLite-aware 客户端来操作它。这样的话，你在一个数据库的拷贝上操作，如果你想要

你需要把数据库备份回去。

把数据库从设备上考出来，你可以使用 adbpull 命令(或者在 IDE 上做相应操作)。

存储一个修改过的数据库到设备上，使用 adb push 命令。 一个最方便的 SQLite 客户端是 FireFox SQLite M
有平台使用。
下图是 SQLite Manager 工具：
如果你想要开发 Android 应用程序，一定需要在 Android 上存储数据，使用 SQLite 数据库是一种非常好的选

使用 ContentProvider 存储数据

Android 这个系统和其他的操作系统还不太一样，我们需要记住的是，数据在 Android 当中是私有的，当然这些
数据以及一些其他类型的数据。那这个时候有读者就会提出问题，难道两个程序之间就没有办法对于数据进行交换
不会让这种情况发生的。解决这个问题主要靠 ContentProvider。一个 Content Provider 类实现了一组标准的方
应用保存或读取此 Content Provider 的各种数据类型。也就是说，一个程序可以通过实现一个 Content Provide
暴露出去。外界根本看不到，也不用看到这个应用暴露的数据在应用当中是如何存储的，或者是用数据库存储还是
上获得，这些一切都不重要，重要的是外界可以通过这一套标准及统一的接口和程序里的数据打交道，可以读取程
序的数据，当然，中间也会涉及一些权限的问题。
一个程序可以通过实现一个 ContentProvider 的抽象接口将自己的数据完全暴露出去，而且 ContentProviders 是
将数据暴露，也就是说 ContentProvider 就像一个“数据库”。那么外界获取其提供的数据，也就应该与从数据
样，只不过是采用 URI 来表示外界需要访问的“数据库”。
Content Provider 提供了一种多应用间数据共享的方式，比如：联系人信息可以被多个应用程序访问。
Content Provider 是个实现了一组用于提供其他应用程序存取数据的标准方法的类。应用程序可以在 Content Pr
询数据 修改数据 添加数据 删除数据
标准的 Content Provider: Android 提供了一些已经在系统中实现的标准 Content Provider，比如联系人信息，
Content Provider 来访问设备上存储的联系人信息，图片等等。
查询记录:
在 Content Provider 中使用的查询字符串有别于标准的 SQL 查询。很多诸如 select, add, delete, modify 等操作
来进行，这种 URI 由 3 个部分组成， “content://”，代表数据的路径，和一个可选的标识数据的 ID。
以下是一些示例 URI:
content://media/internal/images 这个 URI 将返回设备上存储的所有图片
content://contacts/people/ 这个 URI 将返回设备上的所有联系人信息
content://contacts/people/45 这个 URI 返回单个结果(联系人信息中 ID 为 45 的联系人记

录)
尽管这种查询字符串格式很常见，但是它看起来还是有点令人迷惑。为此，Android 提供一系列的帮助类(在 and
包含了很多以类变量形式给出的查询字符串，这种方式更容易让我们理解一点，参见下例:
MediaStore.Images.Media.INTERNAL_CONTENT_URI Contacts.People.CONTENT_URI
因此，如上面 content://contacts/people/45 这个 URI 就可以写成如下形式：
Uri person = ContentUris.withAppendedId(People.CONTENT_URI, 45);
然后执行数据查询: Cursor cur = managedQuery(person, null, null, null);
这个查询返回一个包含所有数据字段的游标，我们可以通过迭代这个游标来获取所有的数据：
1
2
3
4
5
6
7
8
9
```
package com.wissen.testApp;
public class ContentProviderDemo extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
displayRecords();
}
private void displayRecords() {
```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

```
27
28
29
30
31
//该数组中包含了所有要返回的字段
String columns[] = new String[] { People.NAME, People.NUMBER };
Uri mContacts = People.CONTENT_URI;
Cursor cur = managedQuery(
mContacts,
columns, // 要返回的数据字段
null, // WHERE 子句
null, // WHERE 子句的参数
null // Order-by 子句
);
if (cur.moveToFirst()) {
String name = null;
String phoneNo = null;
do {
// 获取字段的值
name = cur.getString(cur.getColumnIndex(People.NAME));
phoneNo =cur.getString(cur.getColumnIndex(People.NUMBER));
Toast.makeText(this, name + "" + phoneNo, Toast.LENGTH_LONG).show();
} while (cur.moveToNext());
}
}
}
```

上例示范了一个如何依次读取联系人信息表中的指定数据列 name 和 number。

修改记录:

我们可以使用 ContentResolver.update()方法来修改数据，我们来写一个修改数据的方法:

```
1
2
3
4
5
6
private void updateRecord(int recNo, String name) {
Uri uri = ContentUris.withAppendedId(People.CONTENT_URI, recNo);
ContentValues values = newContentValues();
values.put(People.NAME, name);
getContentResolver().update(uri, values, null, null);
}
```

现在你可以调用上面的方法来更新指定记录： updateRecord(10, ”XYZ”); //更改第 10 条记录的 name 字段值

添加记录:

要增加记录，我们可以调用 ContentResolver.insert()方法，该方法接受一个要增加的记录的目标 URI，以及一个

象，调用后的返回值是新记录的 URI，包含记录号。

上面的例子中我们都是基于联系人信息簿这个标准的 Content Provider，现在我们继续来创建一个 insertRecord(

进行数据的添加：

```
1
2
3
private void insertRecords(String name, String phoneNo) {
ContentValues values = new ContentValues();
values.put(People.NAME, name);
4
5
6
7
8
9
10
11
Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
Log.d(”ANDROID”, uri.toString());
Uri numberUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);
values.clear();
values.put(Contacts.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.NUMBER, phoneNo);
getContentResolver().insert(numberUri, values);
}
```

这样我们就可以调用 insertRecords(name, phoneNo)的方式来向联系人信息簿中添加联系人姓名和电话号码。

删除记录:

Content Provider 中的 getContextResolver.delete()方法可以用来删除记录。

下面的记录用来删除设备上所有的联系人信息：

```
1
2
3
4
private void deleteRecords() {
Uri uri =People.CONTENT_URI;
getContentResolver().delete(uri, null, null);
}
```

你也可以指定 WHERE 条件语句来删除特定的记录：

```
getContentResolver().delete(uri, “NAME=” + “‘XYZ XYZ’”, null);
```

这将会删除 name 为‘XYZ XYZ’的记录。
创建 Content Provider:
至此我们已经知道如何使用 Content Provider 了，现在让我们来看下如何自己创建一个 Content Provider。
要创建我们自己的 Content Provider 的话，我们需要遵循以下几步：
1. 创建一个继承了 ContentProvider 父类的类
2. 定义一个名为 CONTENT_URI，并且是 public static final 的 Uri 类型的类变量，你必须为其指定一个唯一的字
类的全名称，
如：public static final Uri CONTENT_URI = Uri.parse(“content://com.google.android.MyContentProvide
3. 创建你的数据存储系统。大多数 Content Provider 使用 Android 文件系统或 SQLite 数据库来保持数据，但是
方式来存储。
4. 定义你要返回给客户端的数据列名。如果你正在使用 Android 数据库，则数据列的使用方式就和你以往所熟悉
你必须为其定义一个叫_id 的列，它用来表示每条记录的唯一性。
5. 如果你要存储字节型数据，比如位图文件等，那保存该数据的数据列其实是一个表示实际保存文件的 URI 字符
应的文件数据，处理这种数据类型的 Content Provider 需要实现一个名为_data 的字段，_data 字段列出了该文件
精确路径。这个字段不仅是供客户端使用，而且也可以供 ContentResolver 使用。客户端可以调用 ContentResol
方法来处理该 URI 指向的文件资源，如果是 ContentResolver 本身的话，由于其持有的权限比客户端要高，所以
6. 声明 public staticString 型的变量，用于指定要从游标处返回的数据列。
7. 查询返回一个 Cursor 类型的对象。所有执行写操作的方法如 insert(), update() 以及 delete()都将被监听。我
ContentResover().notifyChange()方法来通知监听器关于数据更新的信息。
8. 在 AndroidMenifest.xml 中使用标签来设置 Content Provider。
9. 如果你要处理的数据类型是一种比较新的类型，你就必须先定义一个新的 MIME 类型，以供 ContentProvider
MIME 类型有两种形式:
一种是为指定的单个记录的，还有一种是为多条记录的。这里给出一种常用的格式：
vnd.android.cursor.item/vnd.yourcompanyname.contenttype(单个记录的 MIME 类型) 比如，一个请求列车
content://com.example.transportationprovider/trains/122 可能就会返回 typevnd.android.cursor.item/vnd
MIME 类型。
vnd.android.cursor.dir/vnd.yourcompanyname.contenttype (多个记录的 MIME 类型) 比如，一个请求所有列
content://com.example.transportationprovider/trains 可能就会返回 vnd.android.cursor.dir/vnd.example.r
下列代码将创建一个 Content Provider，它仅仅是存储用户名称并显示所有的用户名称(使用

SQLLite 数据库存储

```
1
2
3
4
5
package com.wissen.testApp;
public class MyUsers {
public static final String AUTHORITY = “com.wissen.MyContentProvider”;
// BaseColumn 类中已经包含了 _id 字段
public static final class User implementsBaseColumns {
6
7
8
9
10
public static final UriCONTENT_URI = Uri.parse(”content://com.wissen.MyContentProvider”);
// 表数据列
publicstatic final String USER_NAME = “USER_NAME”;
}
}
```

上面的类中定义了 Content Provider 的 CONTENT_URI，以及数据列。下面我们将定义基于上面的类来定义实际

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
package com.wissen.testApp.android;
public class MyContentProvider extendsContentProvider {
private SQLiteDatabase sqlDB;
private DatabaseHelper dbHelper;
private static finalString DATABASE_NAME = “Users.db”;
private static finalint DATABASE_VERSION = 1;
```

```
private static finalString TABLE_NAME = “User”;
private static finalString TAG =“MyContentProvider”;
private static class DatabaseHelper extends SQLiteOpenHelper {
DatabaseHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
@Override
public voidonCreate(SQLiteDatabase db) {
//创建用于存储数据的表
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
db.execSQL(”Create table ” +TABLE_NAME+ “( _id INTEGER PRIMARY KEY AUTOINCREMEN
}
@Override
public voidonUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
db.execSQL(”DROP TABLE IF EXISTS ” + TABLE_NAME);
onCreate(db);
}
}
@Override
public intdelete(Uri uri, String s, String[] as) {
return 0;
}
@Override
```

```
public String getType(Uri uri) {
return null;
}
@Override
public Uri insert(Uri uri, ContentValues contentvalues) {
sqlDB = dbHelper.getWritableDatabase();
long rowId = sqlDB.insert(TABLE_NAME, “”, contentvalues);
if (rowId > 0) {
Uri rowUri = ContentUris.appendId(MyUsers.User.CONTENT_URI.buildUpon(), rowId).build();
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
getContext().getContentResolver().notifyChange(rowUri, null);
return rowUri;
}
throw new SQLException(”Failed to insert row into ” + uri);
}
@Override
public boolean onCreate() {
dbHelper = new DatabaseHelper(getContext());
return (dbHelper== null) ? false : true;
}
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String
sortOr
```

```
SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
SQLiteDatabase db = dbHelper.getReadableDatabase();
qb.setTables(TABLE_NAME);
Cursor c = qb.query(db, projection, selection, null, null, null, sortOrder);
c.setNotificationUri(getContext().getContentResolver(), uri);
return c;
}
@Override
public intupdate(Uri uri, ContentValuescontentvalues, String s, String[] as) {
return 0;
60
61
}
}
```

一个名为 MyContentProvider 的 Content Provider 创建完成了，它用于从 Sqlite 数据库中添加和读取记录。

Content Provider 的入口需要在 AndroidManifest.xml 中配置:

之后，让我们来使用这个定义好的 Content Provider:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
package com.wissen.testApp;
public class MyContentDemo extends Activity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
insertRecord(”MyUser”);
displayRecords();
}
private void insertRecord(String userName){
ContentValues values = new ContentValues();
values.put(MyUsers.User.USER_NAME, userName);
```

```
getContentResolver().insert(MyUsers.User.CONTENT_URI, values);
}
private void displayRecords() {
16
17
18
19
20
21
22
23
24
25
26
27
28
29
String columns[] = new String[] { MyUsers.User._ID, MyUsers.User.USER_NAME };
Uri myUri = MyUsers.User.CONTENT_URI;
Cursor cur = managedQuery(myUri, columns,null, null, null );
if (cur.moveToFirst()) {
Stringid = null;
StringuserName = null;
do {
id = cur.getString(cur.getColumnIndex(MyUsers.User._ID));
userName = cur.getString(cur.getColumnIndex(MyUsers.User.USER_NAME));
Toast.makeText(this, id + ” ” + userName, Toast.LENGTH_LONG).show();
} while (cur.moveToNext());
}
}
}
```

上面的类将先向数据库中添加一条用户数据，然后显示数据库中所有的用户数据。

网络存储数据

前面介绍的几种存储都是将数据存储在本地设备上，除此之外，还有一种存储(获取)数据的方式，通过网络来实现

我们可以调用 WebService 返回的数据或是解析 HTTP 协议实现网络数据交互。

具体需要熟悉 java.net.*，Android.net.*这两个包的内容，在这就不赘述了，请大家参阅相关文档。

下面是一个通过地区名称查询该地区的天气预报，以 POST 发送的方式发送请求到 webservicex.net 站点，访问 We

站点上提供查询天气预报的服务。

代码如下：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
package com.android.weather;
import java.util.ArrayList;
import java.util.List;
import org.apache.http.HttpResponse;
import org.apache.http.NameValuePair;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.protocol.HTTP;
import org.apache.http.util.EntityUtils;
import android.app.Activity;
import android.os.Bundle;
public class MyAndroidWeatherActivity extends Activity {
//定义需要获取的内容来源地址
private static finalString SERVER_URL =
"http://www.webservicex.net/WeatherForecast.asmx/GetWeatherByPlaceName";
19
20
21
22
23
24
25
26
27
```

```
28
29
30
31
32
33
34
35
36
37
38
39
40
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
HttpPost request = new HttpPost(SERVER_URL); //根据内容来源地址创建一个 Http 请求
// 添加一个变量
List<NameValuePair> params = new ArrayList<NameValuePair>();
// 设置一个地区名称
params.add(new BasicNameValuePair("PlaceName","NewYork")); //添加必须的参数
try{
//设置参数的编码
request.setEntity(new UrlEncodedFormEntity(params, HTTP.UTF_8));
//发送请求并获取反馈
HttpResponse httpResponse = new DefaultHttpClient().execute(request);
//解析返回的内容
if(httpResponse.getStatusLine().getStatusCode() != 404){
```

. 18. SharedPrefrence 源码和问题点；

1.储存于硬盘上的 xml 键值对，数据多了会有性能问题

2.ContextImpl 记录着 SharedPreferences 的重要数据，文件路径和实例的键值对

3.在 xml 文件全部内加载到内存中之前，读取操作是阻塞的，在 xml 文件全部内加载到内存中之后，是直接读取内存中的数据

4.apply 因为是异步的没有返回值, commit 是同步的有返回值能知道修改是否提交成功

5.多并发的提交 commit 时，需等待正在处理的 commit 数据更新到磁盘文件后才会继续往下执行，从而降低效率; 而 apply 只是原子更新到内存，后调用 apply 函数会直接覆盖前面内存数据，从一定程度上提高很多效率。 3.edit()每次都是创建新的 EditorImpl 对象.

6.博客推荐：全面剖析 SharedPreferences

```
41
42
43
44
45
46
47
48
String result = EntityUtils.toString(httpResponse.getEntity());
System.out.println(result);
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

别忘记了在配置文件中设置访问网络权限：

```
<uses-permission android:name="android.permission.INTERNET" />
```

. 19. sqlite 相关

一.SQLite 的介绍

1.SQLite 简介

SQLite 是一款轻型的数据库，是遵守 ACID 的关联式数据库管理系统，它的设计目标是嵌入 式的，而且目前已经在很多嵌入式产品中使用了它，它占用资源非常的低，在嵌入式设备中，可能只需要几百 K 的内存就够了。它能够支持 Windows/Linux/Unix 等等主流的操作系统，同时能够跟很多程序语言相结合，比如 Tcl、PHP、Java、C++、.Net 等，还有 ODBC 接口，同样比起 Mysql、PostgreSQL 这两款开源世界著名的数据库管理系统来讲，它的处理速度比他们都快。

2.SQLite 的特点：

轻量级

SQLite 和 C/S 模式的数据库软件不同，它是进程内的数据库引擎，因此不存在数据库的客户端和服务器。使用 SQLite 一般只需要带上它的一个动态 库，就可以享受它的全部功能。而且那个动态库的尺寸也挺小，以版本 3.6.11 为例，Windows 下 487KB、Linux 下 347KB。

不需要"安装"

SQLite 的核心引擎本身不依赖第三方的软件，使用它也不需要"安装"。有点类似那种绿色软件。

单一文件

数据库中所有的信息（比如表、视图等）都包含在一个文件内。这个文件可以自由复制到其它目录或其它机器上。

跨平台/可移植性

除了主流操作系统 windows，linux 之后，SQLite 还支持其它一些不常用的操作系统。

弱类型的字段

同一列中的数据可以是不同类型

开源

这个相信大家都懂的!!!!!!!!!!!!

3.SQLite 数据类型

一般数据采用的固定的静态数据类型，而 SQLite 采用的是动态数据类型，会根据存入值自动判断。SQLite 具有以下五种常用的数据类型：

NULL: 这个值为空值

VARCHAR(n)：长度不固定且其最大长度为 n 的字串，n 不能超过 4000。

CHAR(n)：长度固定为 n 的字串，n 不能超过 254。

INTEGER: 值被标识为整数,依据值的大小可以依次被存储为 1,2,3,4,5,6,7,8.

REAL: 所有值都是浮动的数值,被存储为 8 字节的 IEEE 浮动标记序号.

TEXT: 值为文本字符串,使用数据库编码存储(TUTF-8, UTF-16BE or UTF-16-LE).

BLOB: 值是 BLOB 数据块，以输入的数据格式进行存储。如何输入就如何存储,不改 变格式。

DATA ：包含了 年份、月份、日期。

TIME： 包含了 小时、分钟、秒。

相信学过数据库的童鞋对这些数据类型都不陌生的!!!!!!!!!!

二.SQLiteDatabase 的介绍

Android 提供了创建和是用 SQLite 数据库的 API。SQLiteDatabase 代表一个数据库对象，提供了操作数据库的一些方法。在 Android 的 SDK 目录下有 sqlite3 工具，我们可以利用它创建数据库、创建表和执行一些 SQL 语句。下面是 SQLiteDatabase 的常用方法。

SQLiteDatabase 的常用方法

方法名称 方法表示含义

openOrCreateDatabase(String
path,SQLiteDatabase.CursorFactory factory)
打开或创建数据库

insert(String table,String
nullColumnHack,ContentValues values)
插入一条记录

delete(String table,String
whereClause,String[] whereArgs)
删除一条记录

query(String table,String[] columns,String
selection,String[]selectionArgs,String groupBy,String
having,String orderBy)
查询一条记录

update(String table,ContentValues values,String
whereClause,String[] whereArgs)
修改记录

execSQL(String sql) 执行一条 SQL 语句

close() 关闭数据库

Google 公司命名这些方法的名称都是非常形象的。例如 openOrCreateDatabase,我们从字面英文含义就能看出这是个打开或创建数据库的方法。

1、打开或者创建数据库

在 Android 中使用 SQLiteDatabase 的静态方法 openOrCreateDatabase(String path,SQLiteDatabae.CursorFactory factory)打开或者创建一个数据库。它会自动去检测是否存在这个数据库，如果存在则打开，不存在则创建一个数据库；创建成功则返回一个 SQLiteDatabase 对象，否则抛出异常 FileNotFoundException。

下面是创建名为“stu.db”数据库的代码：

openOrCreateDatabase(String path,SQLiteDatabae.CursorFactory factory)

参数 1 数据库创建的路径

参数 2 一般设置为 null 就可以了

```
1 db=SQLiteDatabase.openOrCreateDatabase("/data/data/com.lingdududu.db/databases
/stu.db",null);
```

2、创建表

创建一张表的步骤很简单：

编写创建表的 SQL 语句

调用 SQLiteDatabase 的 execSQL()方法来执行 SQL 语句

下面的代码创建了一张用户表，属性列为：id（主键并且自动增加）、sname（学生姓名）、snumber（学号）

```
1
2
3
4
5
6
private void createTable(SQLiteDatabase
db){
//创建表 SQL 语句
String stu_table="createtable
usertable(_id integer primary key
autoincrement,sname text,snumber text)";
//执行 SQL 语句
db.execSQL(stu_table);
}
```

3、插入数据

插入数据有两种方法：

①SQLiteDatabase 的 insert(String table,String nullColumnHack,ContentValues values)方法，

参数 1 表名称，

参数 2 空列的默认值

参数 3ContentValues 类型的一个封装了列名称和列值的 Map；

②编写插入数据的 SQL 语句，直接调用 SQLiteDatabase 的 execSQL()方法来执行

第一种方法的代码：

```
1
2
3
```

```
4
5
6
7
private void insert(SQLiteDatabase
db){
//实例化常量值
ContentValues cValue
= new ContentValues();
//添加用户名
cValue.put("sname","xiaoming");
8
9
10
//添加密码
cValue.put("snumber","01005");
//调用 insert()方法插入数据
db.insert("stu_table",null,cValue);
}
```

第二种方法的代码：

```
1
2
3
4
5
6
private void insert(SQLiteDatabase
db){
//插入数据 SQL 语句
String stu_sql="insert into
stu_table(sname,snumber)
values('xiaoming','01005')";
//执行 SQL 语句
db.execSQL(sql);
}
```

4、删除数据

删除数据也有两种方法：

①调用 SQLiteDatabase 的 delete(String table,String whereClause,String[] whereArgs)方法

参数 1 表名称

参数 2 删除条件

参数 3 删除条件值数组

②编写删除 SQL 语句，调用 SQLiteDatabase 的 execSQL()方法来执行删除。

第一种方法的代码：

1
2
3
4
5
6
7

```
private void delete(SQLiteDatabase db) {
//删除条件
String whereClause = "id=?";
//删除条件参数
String[] whereArgs = {String.valueOf(2)};
//执行删除
db.delete("stu_table",whereClause,whereArgs);
8 }
```

第二种方法的代码：

1
2
3
4
5
6

```
private void delete(SQLiteDatabase
db) {
//删除 SQL 语句
String sql = "delete fromstu_table
where _id = 6";
//执行 SQL 语句
db.execSQL(sql);
}
```

5、修改数据

修改数据有两种方法：

①调用 SQLiteDatabase 的 update(String table,ContentValues values,String whereClause, String[] whereArgs)方法

参数 1 表名称

参数 2 跟行列 ContentValues 类型的键值对 Key-Value

参数 3 更新条件（where 字句）

参数 4 更新条件数组

②编写更新的 SQL 语句，调用 SQLiteDatabase 的 execSQL 执行更新。

第一种方法的代码：

1
2
3
4

5
6
7
8
9
10
11

```
private void update(SQLiteDatabasedb) {
//实例化内容值 ContentValues values = new
ContentValues();
//在 values 中添加内容
values.put("snumber","101003");
//修改条件
String whereClause = "id=?";
//修改添加参数
String[] whereArgs={String.valuesOf(1)};
//修改
db.update("usertable",values,whereClause,whereArgs);
}
```

第二种方法的代码：

1
2
3
4
5
6

```
private void update(SQLiteDatabase
db){
//修改 SQL 语句
String sql = "update stu_table set
snumber = 654321 where id = 1";
//执行 SQL
db.execSQL(sql);
}
```

6、查询数据

在 Android 中查询数据是通过 Cursor 类来实现的，当我们使用 SQLiteDatabase.query() 方法时，会得到一个 Cursor 对象，Cursor 指向的就是每一条数据。它提供了很多有关查询的方法，具体方法如下：

```
public Cursor query(String table,String[] columns,String
selection,String[]selectionArgs,String groupBy,String having,String
orderBy,String limit);
```

各个参数的意义说明：

参数 table:表名称

参数 columns:列名称数组

参数 selection:条件字句，相当于 where
参数 selectionArgs:条件字句，参数数组
参数 groupBy:分组列
参数 having:分组条件
参数 orderBy:排序列
参数 limit:分页查询限制
参数 Cursor:返回值，相当于结果集 ResultSet
Cursor 是一个游标接口，提供了遍历查询结果的方法，如移动指针方法 move()，获得列值方法 getString()等.
Cursor 游标常用方法
方法名称 方法描述
getCount() 获得总的数据项数
isFirst() 判断是否第一条记录
isLast() 判断是否最后一条记录
moveToFirst() 移动到第一条记录
moveToLast() 移动到最后一条记录
move(int offset) 移动到指定记录
moveToNext() 移动到下一条记录
moveToPrevious() 移动到上一条记录
getColumnIndexOrThrow(String columnName) 根据列名称获得列索引
getInt(int columnIndex) 获得指定列索引的 int 类型值
getString(int columnIndex)
获得指定列缩影的 String 类型
值
下面就是用 Cursor 来查询数据库中的数据，具体代码如下：

```
1
2
3
4
5
6
7
8
9
10
private void query(SQLiteDatabase db) {
//查询获得游标
Cursor cursor = db.query
("usertable",null,null,null,null,null,null);
//判断游标是否为空
if(cursor.moveToFirst() {
//遍历游标
for(int i=0;i<cursor.getCount();i++){
cursor.move(i);
```

```
11
12
13
14
15
16
17
18
19
//获得 ID
int id = cursor.getInt(0);
//获得用户名
String username=cursor.getString(1);
//获得密码
String password=cursor.getString(2);
//输出用户信息
System.out.println(id+":"+sname+":"+snumber);
}
}
}
```

7、删除指定表

编写插入数据的 SQL 语句，直接调用 SQLiteDatabase 的 execSQL()方法来执行

```
1
2
3
4
5
6
private void drop(SQLiteDatabase
db){
//删除表的 SQL 语句
String sql ="DROP TABLE
stu_table";
//执行 SQL
db.execSQL(sql);
}
```

三.SQLiteOpenHelper

该类是 SQLiteDatabase 一个辅助类。这个类主要生成一 个数据库，并对数据库的版本进行管理。当在程序当中调用这个类的方法 getWritableDatabase()或者 getReadableDatabase()方法的时候，如果当时没有数据，那么 Android 系统就会自动生成一个数据库。 SQLiteOpenHelper 是一个抽象类，我们通常需要继承它，并且实现里面的 3 个函数：

1.onCreate（SQLiteDatabase）

在数据库第一次生成的时候会调用这个方法，也就是说，只有在创建数据库的时候才会调用，

当然也有一些其它的情况，一般我们在这个方法里边生成数据库表。

2. onUpgrade（SQLiteDatabase，int，int）

当数据库需要升级的时候，Android 系统会主动的调用这个方法。一般我们在这个方法里边删除数据表，并建立新的数据表，当然是否还需要做其他的操作，完全取决于应用的需求。

3. onOpen（SQLiteDatabase）：

这是当打开数据库时的回调函数，一般在程序中不是很常使用。

写了这么多，改用用实际例子来说明上面的内容了。下面这个操作数据库的实例实现了创建数据库，创建表以及数据库的增删改查的操作。

该实例有两个类：

com.lingdududu.testSQLite 调试类

com.lingdududu.testSQLiteDb 数据库辅助类

SQLiteActivity.java

```
1
2
3
4
5
6
7
8
```

```
package com.lingdududu.testSQLite;
import com.lingdududu.testSQLiteDb.StuDBHelper;
import android.app.Activity;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
/*
* @author lingdududu
*/
public class SQLiteActivity extends Activity {
/** Called when the activity is first created. */
//声明各个按钮
private Button createBtn;
private Button insertBtn;
private Button updateBtn;
private Button queryBtn;
private Button deleteBtn;
private Button ModifyBtn;
@Override
public void onCreate(Bundle savedInstanceState) {
```

```
27
28
29
30
31
32
33
34
35
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
//调用 creatView 方法
creatView();
//setListener 方法
setListener();
}
//通过 findViewById 获得 Button 对象的方法
private void creatView(){
createBtn = (Button)findViewById(R.id.createDatabase);
updateBtn = (Button)findViewById(R.id.updateDatabase);
insertBtn = (Button)findViewById(R.id.insert);
ModifyBtn = (Button)findViewById(R.id.update);
queryBtn = (Button)findViewById(R.id.query);
deleteBtn = (Button)findViewById(R.id.delete);
}
//为按钮注册监听的方法
private void setListener(){
createBtn.setOnClickListener(new CreateListener());
updateBtn.setOnClickListener(new UpdateListener());
insertBtn.setOnClickListener(new InsertListener());
ModifyBtn.setOnClickListener(new ModifyListener());
queryBtn.setOnClickListener(new QueryListener());
deleteBtn.setOnClickListener(new DeleteListener());
}
//创建数据库的方法
class CreateListener implements OnClickListener{
@Override
public void onClick(View v) {
//创建 StuDBHelper 对象
StuDBHelper dbHelper
63
64
65
66
```

```
67
95
96
97
98
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可读的 SQLiteDatabase 对象
SQLiteDatabase db =dbHelper.getReadableDatabase();
}
}
//更新数据库的方法
class UpdateListener implements OnClickListener{
@Override
public void onClick(View v) {
// 数据库版本的更新,由原来的 1 变为 2
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,2);
SQLiteDatabase db =dbHelper.getReadableDatabase();
}
}
//插入数据的方法
class InsertListener implements OnClickListener{
@Override
public void onClick(View v) {
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getWritableDatabase();
//生成 ContentValues 对象 //key:列名，value:想插入的值
ContentValues cv= new ContentValues();
//往 ContentValues 对象存放数据，键-值对模式
cv.put("id", 1);
cv.put("sname", "xiaoming");
cv.put("sage", 21);
cv.put("ssex", "male");
99
100
101
102
103
104
105
106
107
```

```
108
109
110
//调用 insert 方法，将数据插入数据库
db.insert("stu_table", null, cv);
//关闭数据库
db.close();
}
}
//查询数据的方法
class QueryListener implements OnClickListener{
@Override
public void onClick(View v) {
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getReadableDatabase();
//参数 1：表名
//参数 2：要想显示的列
//参数 3：where 子句
//参数 4：where 子句对应的条件值
//参数 5：分组方式
//参数 6：having 条件
//参数 7：排序方式
Cursor cursor =
db.query("stu_table", new String[]{"id","sname","sage","ssex"}, "id=?
", new String[]{"1"}, null, null,null);
while(cursor.moveToNext()){
String name = cursor.getString(cursor.getColumnIndex("sname"));
String age = cursor.getString(cursor.getColumnIndex("sage"));
String sex = cursor.getString(cursor.getColumnIndex("ssex"));
System.out.println("query------->" + "姓名："+name+" "+"年龄："+age+" "+"
性别："+sex);
}
//关闭数据库
db.close();
135
136
137
138
139
140
141
142
```

```
143
144
145
146
147
167
}
}
//修改数据的方法
class ModifyListener implements OnClickListener{
@Override
public void onClick(View v) {
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getWritableDatabase();
ContentValues cv= new ContentValues();
cv.put("sage", "23");
//where 子句 "?"是占位符号，对应后面的"1",
String whereClause="id=?";
String [] whereArgs = {String.valueOf(1)};
//参数 1 是要更新的表名
//参数 2 是一个 ContentValeus 对象
//参数 3 是 where 子句
db.update("stu_table", cv, whereClause, whereArgs);
}
}
//删除数据的方法
class DeleteListener implements OnClickListener{
@Override
public void onClick(View v) {
StuDBHelper dbHelper
= new StuDBHelper(SQLiteActivity.this,"stu_db",null,1);
//得到一个可写的数据库
SQLiteDatabase db =dbHelper.getReadableDatabase();
String whereClauses = "id=?";
String [] whereArgs = {String.valueOf(2)};
//调用 delete 方法，删除数据
db.delete("stu_table", whereClauses, whereArgs);
}
}
}
StuDBHelper.java
1
```

```
2
3
4
5
6
7
8
9
10
11
package com.lingdududu.testSQLiteDb;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;
public class StuDBHelper extends SQLiteOpenHelper {
private static final String TAG = "TestSQLite";
public static final int VERSION = 1;
//必须要有构造函数
public StuDBHelper(Context context, Stringname, CursorFactory
factory,
int version) {
super(context, name, factory, version);
}
// 当第一次创建数据库的时候，调用该方法
public void onCreate(SQLiteDatabase db) {
String sql = "create table stu_table(id int,sname
varchar(20),sage int,ssex varchar(10))";
//输出创建数据库的日志信息
Log.i(TAG, "create Database------------>");
//execSQL 函数用于执行 SQL 语句
28
29
30
31
32
33
34
db.execSQL(sql);
}
//当更新数据库的时候执行该方法
public void onUpgrade(SQLiteDatabase
db, int oldVersion, int newVersion) {
```

```
//输出更新数据库的日志信息
Log.i(TAG, "update Database---------->");
}
}
```

main.xml

```
1
2
3
4
5
6
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello"
/>
<Button
android:id="@+id/createDatabase"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="创建数据库"
/>
<Button
android:id="@+id/updateDatabase"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="更新数据库"
24
25
26
27
33
38
48
/>
<Button
android:id="@+id/insert"
```

```
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="插入数据"
/>
<Button
android:id="@+id/update"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="更新数据"
/>
<Button
android:id="@+id/query"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="查询数据"
/>
<Button
android:id="@+id/delete"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="删除数据"
/>
</LinearLayout>
```

程序运行的效果图：

使用 adb 命令查看数据库：

1.在命令行窗口输入 adb shell 回车，就进入了 Linux 命令行，现在就可以使用 Linux 的命令了。

2.ls 回车，显示所有的东西，其中有个 data。
3.cd data 回车，再 ls 回车，cddata 回车，ls 回车后就会看到很多的 com.....，那就是系统上的应用程序包名，找到你数据库程序的包名，然后进入。
4.进去后在查看所有，会看到有 databases,进入 databases，显示所有就会发现你的数据库名字，这里使用的是"stu_db"。
5.sqlite3 stu_db 回车就进入了你的数据库了，然后“.schema”就会看到该应用程序的所有表及建表语句。
6.之后就可以使用标准的 SQL 语句查看刚才生成的数据库及对数据执行增删改查了。
注：ls,cd 等命令都是 linux 的基本命令，不了解的同学可以看看有关这方面的资料。
下面介绍几个在 SQLite 中常用到的 adb 命令：
查看
.database 显示数据库信息；
.tables 显示表名称；
.schema 命令可以查看创建数据表时的 SQL 命令；
.schema table_name 查看创建表 table_name 时的 SQL 的命令；
插入记录
insert into table_name values (field1, field2, field3...);
查询
select * from table_name;查看 table_name 表中所有记录；
select * from table_name where field1='xxxxx'; 查询符合指定条件的记录；
删除
drop table_name; 删除表；
drop index_name; 删除索引；
----------查询，插入，删除等操作数据库的语句记得不要漏了;----------
sqlite3 stu_db
sqlite3 stu_db
SQLite version 3.6.22
Enter ".help"for instructions
Enter SQL statements terminated witha ";"
sqlite> .schema
.schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE stu_table(id int,sname varchar(20),sage int,ssex varchar(10)); --->创建的表
sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|21|male
sqlite>
插入数据
sqlite> insert into stu_table values(2,'xiaohong',20,'female');
插入的数据记得要和表中的属性一一对应
insert into stu_table values(2,'xiaohong',20,'female');
sqlite> select * from stu_table;

select * from stu_table;
1|xiaoming|21|male
2|xiaohong|20|female ----------> 插入的数据
sqlite>
当点击修改数据的按钮时候
sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|23|male ---------->年龄被修改为 23
2|xiaohong|20|female
sqlite>
当点击删除数据的按钮
sqlite> select * from stu_table;
select * from stu_table;
1|xiaoming|23|male id=2 的数据已经被删除

. 20. 如何判断一个 P APP 在前台还是后台？

六种方法的区别

方法	判断原理	需要权限	可以判断其他应用位于前台	特点
方法一	RunningTask	否	Android4.0 系列可以,5.0 以上机器不行	5.0 此方法被废弃
方法二	RunningProcess	否	当 App 存在后台常驻的 Service 时失效	无
方法三	ActivityLifecycleCallbacks	否	否	简单有效,代码最少
方法四	UsageStatsManager	是	是	需要用户手动授权
方法五	通过 Android 无障碍功能实现	否	是	需要用户手动授权
方法六	读取/proc 目录下的信息	否	是	当 proc 目录下文件夹过多时,过多的 IO 操作会引起耗时

方法一：通过 RunningTask

原理

当一个 App 处于前台的时候，会处于 RunningTask 的这个栈的栈顶，所以我们可以取出 RunningTask 的栈顶的任务进程，看他与我们的想要判断的 App 的包名是否相同，来达到效果

缺点

getRunningTask 方法在 Android5.0 以上已经被废弃，只会返回自己和系统的一些不敏感的 task，不再返回其他应用的 task，用此方法来判断自身 App 是否处于后台，仍然是有效的，但是无法判断其他应用是否位于前台，因为不再能获取信息

方法二：通过 RunningProcess

原理

通过 runningProcess 获取到一个当前正在运行的进程的 List，我们遍历这个 List 中的每一个进程，判断这个进程的一个 importance 属性是否是前台进程，并且包名是否与我们判断的 APP 的包名一样，如果这两个条件都符合，那么这个 App 就处于前台

缺点：

在聊天类型的 App 中，常常需要常驻后台来不间断的获取服务器的消息，这就需要我们把 Service 设置成 START_STICKY，kill 后会被重启（等待 5 秒左右）来保证 Service 常驻后台。如果 Service 设置了这个属性，这个 App 的进程就会被判断是前台，代码上的表现就是 appProcess.importance 的值永远是 ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND，这样就永远无法判断出到底哪个是前台了。

方法三：通过 ActivityLifecycleCallbacks

原理

AndroidSDK14 在 Application 类里增加了 ActivityLifecycleCallbacks，我们可以通过这个 Callback 拿到 App 所有 Activity 的生命周期回调。

```
public interface ActivityLifecycleCallbacks {
void onActivityCreated(Activity activity , Bundle
savedInstanceState );
void onActivityStarted(Activity activity );
void onActivityResumed(Activity activity );
void onActivityPaused(Activity activity );
void onActivityStopped(Activity activity );
void onActivitySaveInstanceState(Activity activity , Bundle
outState );
void onActivityDestroyed(Activity activity );
}
```

知道这些信息，我们就可以用更官方的办法来解决问题，当然还是利用方案二里

的 Activity 生命周期的特性，我们只需要在 Application 的 onCreate()里去注册上述接口，然后由 Activity 回调回来运行状态即可。

可能还有人在纠结，我用 back 键切到后台和用 Home 键切到后台，一样吗？以上方法适用吗？在 Android 应用开发中一般认为 back 键是可以捕获的，而 Home 键是不能捕获的（除非修改 framework）,但是上述方法从 Activity 生命周期着手解决问题，虽然这两种方式的 Activity 生命周期并不相同，但是二者都会执行 onStop（）；所以并不关心到底是触发了哪个键切入后台的。另外,Application 是否被销毁,都不会影响判断的正确性

方法四：：通过使用 r UsageStatsManager 获取

原理

通过使用 UsageStatsManager 获取，此方法是 Android5.0 之后提供的新 API，可以获取一个时间段内的应用统计信息，但是必须满足一下要求

使用前提

1.此方法只在 android5.0 以上有效

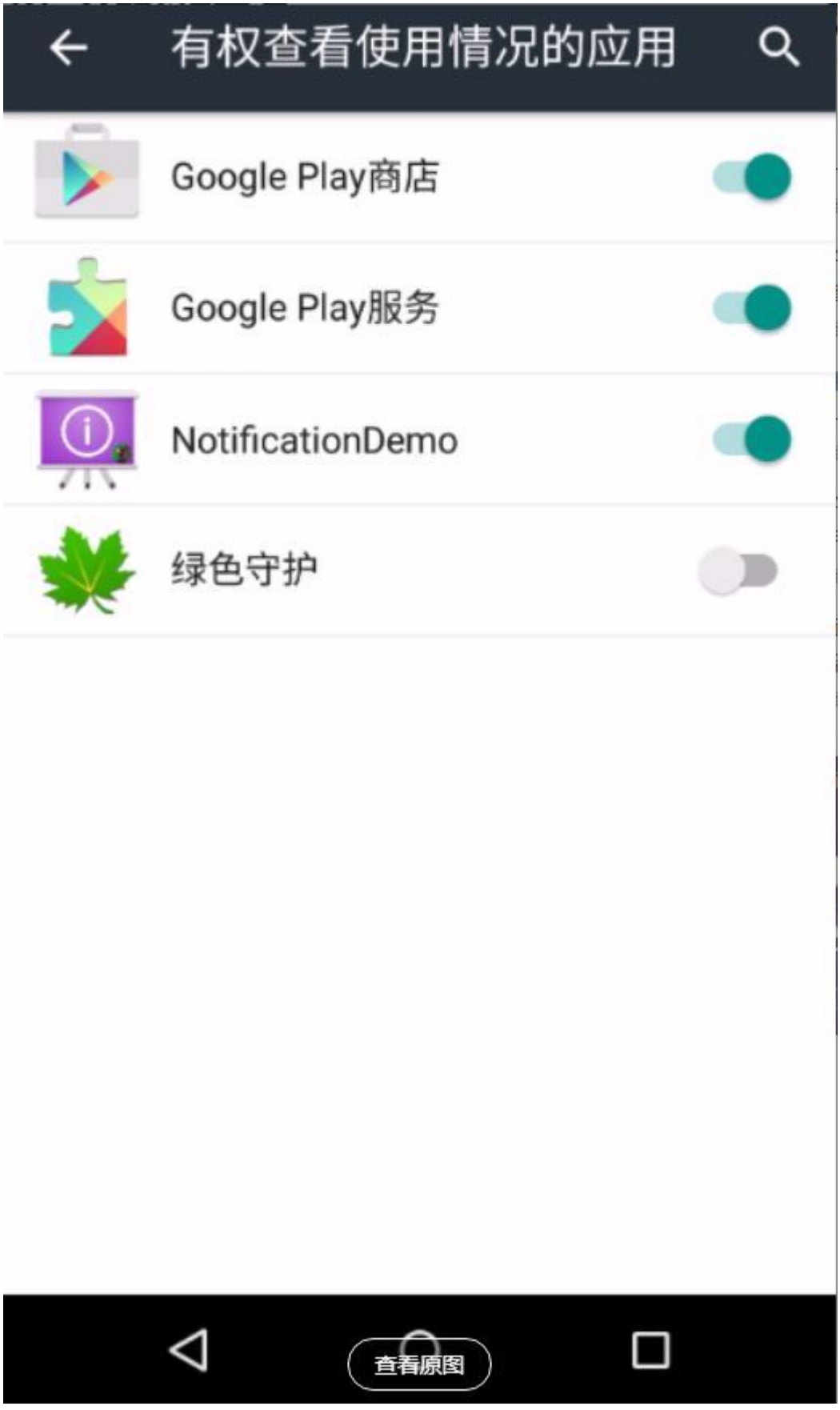
2.AndroidManifest 中加入此权限

<uses- permissionxmlns : tools ="http://schemas.android.com/tools"

android : name ="android.permission.PACKAGE_USAGE_STATS"

tools : ignore ="ProtectedPermissions" />

3.打开手机设置，点击安全-高级，在有权查看使用情况的应用中，为这个 App 打上勾



enterimagedescriptionhere

方法五：通过 d Android 自带的无障碍功能

非常感谢@EffectiveMatrix 大神带来的新的判断前后台的方法
此方法属于他原创，具体的博文参照这里
http://effmx.com/articles/tong-guo-android-fu-zhu-gong-neng-accessibility-service-jian-ce-ren-yi-qian-tai-jie-mian/
此方法无法直观的通过下拉通知视图来进行前后台的观察，请到 LogCat 中进行观察即可，以下是 LogCat 中打印的信息

enterimagedescriptionhere
原理
Android 辅助功能(AccessibilityService)为我们提供了一系列的事件回调，帮助我们指示一些用户界面的状态变化。我们可以派生辅助功能类，进而对不同的 AccessibilityEvent 进行处理。
同样的，这个服务就可以用来判断当前的前台应用
优势
AccessibilityService 有非常广泛的 ROM 覆盖，特别是非国产手机，从 Android API Level 8(Android 2.2) 到 Android Api Level 23(Android 6.0)
AccessibilityService 不再需要轮询的判断当前的应用是不是在前台，系统会在窗口状态发生变化的时候主动回调，耗时和资源消耗都极小
1. 不需要权限请求
2. 它是一个稳定的方法，与 “方法 6”读取 /proc 目录不同，它并非利用 Android 一些设计上的漏洞，可以长期使用的可能很大
3. 可以用来判断任意应用甚至 Activity, PopupWindow, Dialog 对象是否处于前台
劣势
需要要用户开启辅助功能
辅助功能会伴随应用被“强行停止”而剥夺

方法六 ： 读取 x Linux 系统内核保存在 c /proc 目录下的

process 进程信息

此方法并非我原创，原作者是国外的大神，GitHub 项目在这里，也一并加入到工程中，供大家做全面的参考选择
原理
无意中看到乌云上有人提的一个漏洞，Linux 系统内核会把 process 进程信息保存在/proc 目录下，Shell 命令去获取的他，再根据进程的属性判断是否为前台
优点
不需要任何权限
可以判断任意一个应用是否在前台，而不局限在自身应用
缺点
当/proc 下文件夹过多时,此方法是耗时操作
用法

```
List<AndroidAppProcess> processes =
ProcessManager.getRunningAppProcesses();
```

获取任一正在运行的 App 进程的详细信息

```
AndroidAppProcess process = processes .get( location );String processName
= process . name ;
Stat stat = process .stat();int pid = stat .getPid();int parentProcessId
= stat .ppid();long startTime = stat .stime();int policy =
stat .policy();char state = stat .state();
Statm statm = process .statm();long totalSizeOfProcess =
statm .getSize();long residentSetSize = statm .getResidentSetSize();
PackageInfo packageInfo = process .getPackageInfo( context ,0);String
appName = packageInfo . applicationInfo .loadLabel( pm ).toString();
```

判断是否在前台

```
if(ProcessManager.isMyProcessInTheForeground()){
// do stuff}
```

获取一系列正在运行的 App 进程的详细信息

```
List<ActivityManager.RunningAppProcessInfo> processes =
ProcessManager.getRunningAppProcessInfo( c
```

. 21. 混合开发

r Flutter 与 如何与 d Android S iOS 通信？
Flutter 通过 PlatformChannel 与原生进行交互，其中 PlatformChannel 分为三种：
1. BasicMessageChannel：用于传递字符串和半结构化的信息。
2. MethodChannel：用于传递方法调用。Flutter 主动调用 Native 的方法，并获取相应的返回值。
3. EventChannel：用于数据流（event streams）的通信。

Android Framework 高频面试题总结

AMS 、PMS

1.AMS 概述

AMS 是系统的引导服务，应用进程的启动、切换和调度、四大组件的启动和管理都需要 AMS 的支持。从这里可以看出 AMS 的功能会十分的繁多，当然它并不是一个类承担这个重责，它有一些关联类，这在文章后面会讲到。AMS 的涉及的知识点非常多，这篇文章主要会讲解 AMS 的以下几个知识点：

- AMS 的启动流程。
- AMS 与进程启动。
- AMS 家族。

2.AMS 的启动流程

AMS 的启动是在 SyetemServer 进程中启动的，在 Android 系统启动流程（三）解析 SyetemServer 进程启动过程这篇文章中提及过，这里从 SyetemServer 的 main 方法开始讲起：

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
      new SystemServer().run();
  }
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {
      ...
         System.loadLibrary("android_servers");//1
      ...
         mSystemServiceManager = new
SystemServiceManager(mSystemContext);//2
         LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
      ...
       try {
         Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER,
"StartServices");
```

```
            startBootstrapServices();//3
            startCoreServices();//4
            startOtherServices();//5
        } catch (Throwable ex) {
            Slog.e("System",
"************************************************");
            Slog.e("System", "************ Failure starting system
services", ex);
            throw ex;
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
        }
        ...
    }
```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 中的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 BatteryService、UsageStatsService 和 WebViewUpdateService。注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService、VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、4、5 的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和一些不需要立即启动的服务。系统服务总共大约有 80 多个，我们主要来查看引导服务 AMS 是如何启动的，注释 3 处的 startBootstrapServices 方法如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
    private void startBootstrapServices() {
        Installer installer =
mSystemServiceManager.startService(Installer.class);
        // Activity manager runs the show.
        mActivityManagerService = mSystemServiceManager.startService(

ActivityManagerService.Lifecycle.class).getService();//1

mActivityManagerService.setSystemServiceManager(mSystemServiceManager
);
        mActivityManagerService.setInstaller(installer);
      ...
    }
```

在注释 1 处调用了 SystemServiceManager 的 startService 方法，方法的参数是 ActivityManagerService.Lifecycle.class：

frameworks/base/services/core/java/com/android/server/SystemServiceManager.java

```
  @SuppressWarnings("unchecked")
    public <T extends SystemService> T startService(Class<T> serviceClass)
{
        try {
           ...
            final T service;
            try {
                Constructor<T> constructor =
serviceClass.getConstructor(Context.class);//1
                service = constructor.newInstance(mContext);//2
            } catch (InstantiationException ex) {
              ...
            }
            // Register it.
            mServices.add(service);//3
            // Start it.
            try {
                service.onStart();//4
            } catch (RuntimeException ex) {
                throw new RuntimeException("Failed to start service " + name
                        + ": onStart threw an exception", ex);
            }
            return service;
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
        }
    }
```

startService 方法传入的参数是 Lifecycle.class，Lifecycle 继承自 SystemService。首先，通过反射来创建 Lifecycle 实例，注释 1 处得到传进来的 Lifecycle 的构造器 constructor，在注释 2 处调用 constructor 的 newInstance 方法来创建 Lifecycle 类型的 service 对象。接着在注释 3 处将刚创建的 service 添加到 ArrayList 类型的 mServices 对象中来完成注册。最后在注释 4 处调用 service 的 onStart 方法来启动 service，并返回该 service。Lifecycle 是 AMS 的内部类，代码如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
    public static final class Lifecycle extends SystemService {
        private final ActivityManagerService mService;
        public Lifecycle(Context context) {
            super(context);
            mService = new ActivityManagerService(context);//1
```

```
        }
        @Override
        public void onStart() {
            mService.start();//2
        }
        public ActivityManagerService getService() {
            return mService;//3
        }
    }
```

上面的代码结合 SystemServiceManager 的 startService 方法来分析，当通过反射来创建 Lifecycle 实例时，会调用注释 1 处的方法创建 AMS 实例，当调用 Lifecycle 类型的 service 的 onStart 方法时，实际上是调用了注释 2 处 AMS 的 start 方法。在 SystemServer 的 startBootstrapServices 方法的注释 1 处，调用了如下代码：

```
mActivityManagerService = mSystemServiceManager.startService(
            ActivityManagerService.Lifecycle.class).getService();
```

我们知道 SystemServiceManager 的 startService 方法最终会返回 Lifecycle 类型的对象，紧接着又调用了 Lifecycle 的 getService 方法，这个方法会返回 AMS 类型的 mService 对象，见注释 3 处，这样 AMS 实例就会被创建并且返回。

3.AMS 与进程启动

在 Android 系统启动流程（二）解析 Zygote 进程启动过程这篇文章中，我提到了 Zygote 的 Java 框架层中，会创建一个 Server 端的 Socket，这个 Socket 用来等待 AMS 来请求 Zygote 来创建新的应用程序进程。要启动一个应用程序，首先要保证这个应用程序所需要的应用程序进程已经被启动。AMS 在启动应用程序时会检查这个应用程序需要的应用程序进程是否存在，不存在就会请求 Zygote 进程将需要的应用程序进程启动。Service 的启动过程中会调用 ActiveServices 的 bringUpServiceLocked 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```
  private String bringUpServiceLocked(ServiceRecord r, int intentFlags,
boolean execInFg,
            boolean whileRestarting, boolean permissionsReviewRequired)
            throws TransactionTooLargeException {
  ...
  final String procName = r.processName;//1
  ProcessRecord app;
  if (!isolated) {
            app = mAm.getProcessRecordLocked(procName, r.appInfo.uid,
false);//2
```

```
            if (DEBUG_MU) Slog.v(TAG_MU, "bringUpServiceLocked:
appInfo.uid=" + r.appInfo.uid
                    + " app=" + app);
            if (app != null && app.thread != null) {//3
                try {
                    app.addPackage(r.appInfo.packageName,
r.appInfo.versionCode,
                    mAm.mProcessStats);
                    realStartServiceLocked(r, app, execInFg);//4
                    return null;
                } catch (TransactionTooLargeException e) {
              ...
            }
        } else {
            app = r.isolatedProc;
        }
 if (app == null && !permissionsReviewRequired) {//5
            if ((app=mAm.startProcessLocked(procName, r.appInfo, true,
intentFlags,
                    "service", r.name, false, isolated, false)) == null)
{//6
              ...
            }
            if (isolated) {
                r.isolatedProc = app;
            }
        }
 ...     }
```

在注释 1 处得到 ServiceRecord 的 processName 的值赋值给 procName ，其中 ServiceRecord 用来描述 Service 的 android:process 属性。注释 2 处将 procName 和 Service 的 uid 传入到 AMS 的 getProcessRecordLocked 方法中，来查询是否存在一个与 Service 对应的 ProcessRecord 类型的对象 app，ProcessRecord 主要用来记录运行的应用程序进程的信息。注释 5 处判断 Service 对应的 app 为 null 则说明用来运行 Service 的应用程序进程不存在，则调用注释 6 处的 AMS 的 startProcessLocked 方法来创建对应的应用程序进程，
具体的过程请查看 Android 应用程序进程启动过程（前篇）。

4. AMS 家族

ActivityManager 是一个和 AMS 相关联的类，它主要对运行中的 Activity 进行管理，这些管理工作并不是由 ActivityManager 来处理的，而是交由 AMS 来处理，ActivityManager 中的方法会通过 ActivityManagerNative（以后简称 AMN）的 getDefault 方法来得到 ActivityManagerProxy(以后简称 AMP)，通过 AMP 就可以和

AMN 进行通信，而 AMN 是一个抽象类，它会将功能交由它的子类 AMS 来处理，因此，AMP 就是 AMS 的代理类。AMS 作为系统核心服务，很多 API 是不会暴露给 ActivityManager 的，因此 ActivityManager 并不算是 AMS 家族一份子。

为了讲解 AMS 家族，这里拿 Activity 的启动过程举例，Activity 的启动过程中会调用 Instrumentation 的 execStartActivity 方法，如下所示。

frameworks/base/core/java/android/app/Instrumentation.java

```
public ActivityResult execStartActivity(
          Context who, IBinder contextThread, IBinder token, Activity
target,
          Intent intent, int requestCode, Bundle options) {
     ...
       try {
          intent.migrateExtraStreamToClipData();
          intent.prepareToLeaveProcess(who);
          int result = ActivityManagerNative.getDefault()
             .startActivity(whoThread, who.getBasePackageName(),
intent,

intent.resolveTypeIfNeeded(who.getContentResolver()),
                    token, target != null ? target.mEmbeddedID : null,
                    requestCode, 0, null, options);
          checkStartActivityResult(result, intent);
       } catch (RemoteException e) {
          throw new RuntimeException("Failure from system", e);
       }
       return null;
   }
```

execStartActivity 方法中会调用 AMN 的 getDefault 来获取 AMS 的代理类 AMP。接着调用了 AMP 的 startActivity 方法，先来查看 AMN 的 getDefault 方法做了什么，如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
 static public IActivityManager getDefault() {
       return gDefault.get();
   }
   private static final Singleton<IActivityManager> gDefault = new
Singleton<IActivityManager>() {
       protected IActivityManager create() {
          IBinder b = ServiceManager.getService("activity");//1
          if (false) {
             Log.v("ActivityManager", "default service binder = " + b);
          }
          IActivityManager am = asInterface(b);//2
```

```
            if (false) {
                Log.v("ActivityManager", "default service = " + am);
            }
            return am;
        }+
    };}
```

getDefault 方法调用了 gDefault 的 get 方法，我们接着往下看，gDefault 是一个 Singleton 类。注释 1 处得到名为"activity"的 Service 引用，也就是 IBinder 类型的 AMS 的引用。接着在注释 2 处将它封装成 AMP 类型对象，并将它保存到 gDefault 中，此后调用 AMN 的 getDefault 方法就会直接获得 AMS 的代理对象 AMP。注释 2 处的 asInterface 方法如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
static public IActivityManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }
    IActivityManager in =
        (IActivityManager)obj.queryLocalInterface(descriptor);
    if (in != null) {
        return in;
    }
    return new ActivityManagerProxy(obj);}
```

asInterface 方法的主要作用就是将 IBinder 类型的 AMS 引用封装成 AMP，AMP 的构造方法如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
class ActivityManagerProxy implements IActivityManager{
    public ActivityManagerProxy(IBinder remote)
    {
        mRemote = remote;
    }...
 }
```

AMP 的构造方法中将 AMS 的引用赋值给变量 mRemote ，这样在 AMP 中就可以使用 AMS 了。

其中 IActivityManager 是一个接口，AMN 和 AMP 都实现了这个接口，用于实现代理模式和 Binder 通信。

再回到 Instrumentation 的 execStartActivity 方法，来查看 AMP 的 startActivity 方法，AMP 是 AMN 的内部类，代码如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
public int startActivity(IApplicationThread caller, String
callingPackage, Intent intent,
        String resolvedType, IBinder resultTo, String resultWho, int
requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle options)
throws RemoteException {
    ...
      data.writeInt(requestCode);
      data.writeInt(startFlags);
    ...
      mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);//1
      reply.readException();+
      int result = reply.readInt();
      reply.recycle();
      data.recycle();
      return result;
  }
```

首先会将传入的参数写入到 Parcel 类型的 data 中。在注释 1 处，通过 IBinder 类型对象 mRemote（AMS 的引用）向服务端的 AMS 发送一个 START_ACTIVITY_TRANSACTION 类型的进程间通信请求。那么服务端 AMS 就会从 Binder 线程池中读取我们客户端发来的数据，最终会调用 AMN 的 onTransact 方法，如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
  @Override
  public boolean onTransact(int code, Parcel data, Parcel reply, int
flags)
        throws RemoteException {
      switch (code) {
      case START_ACTIVITY_TRANSACTION:
      {
      ...
        int result = startActivity(app, callingPackage, intent,
resolvedType,
                resultTo, resultWho, requestCode, startFlags,
profilerInfo, options);
        reply.writeNoException();
        reply.writeInt(result);
        return true;
      }
  }
```

onTransact 中会调用 AMS 的 startActivity 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
@Override
public final int startActivity(IApplicationThread caller, String
callingPackage,
        Intent intent, String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions) {
    return startActivityAsUser(caller, callingPackage, intent,
resolvedType, resultTo,
            resultWho, requestCode, startFlags, profilerInfo, bOptions,
            UserHandle.getCallingUserId());
}
```

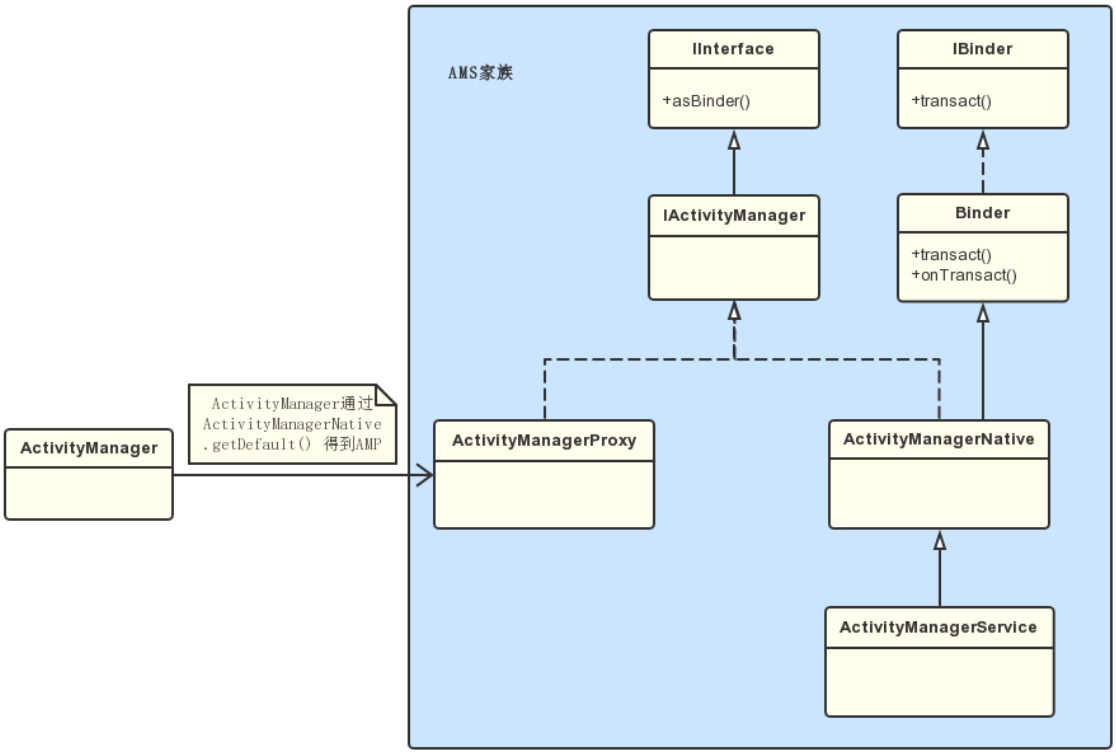
startActivity 方法会最后 return startActivityAsUser 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
@Override
public final int startActivityAsUser(IApplicationThread caller, String
callingPackage,
        Intent intent, String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int
userId) {
    enforceNotIsolatedCaller("startActivity");
    userId = mUserController.handleIncomingUser(Binder.getCallingPid(),
Binder.getCallingUid(),
            userId, false, ALLOW_FULL_ONLY, "startActivity", null);
    return mActivityStarter.startActivityMayWait(caller, -1,
callingPackage, intent,
            resolvedType, null, null, resultTo, resultWho, requestCode,
startFlags,
            profilerInfo, null, null, bOptions, false, userId, null,
null);
  }
```

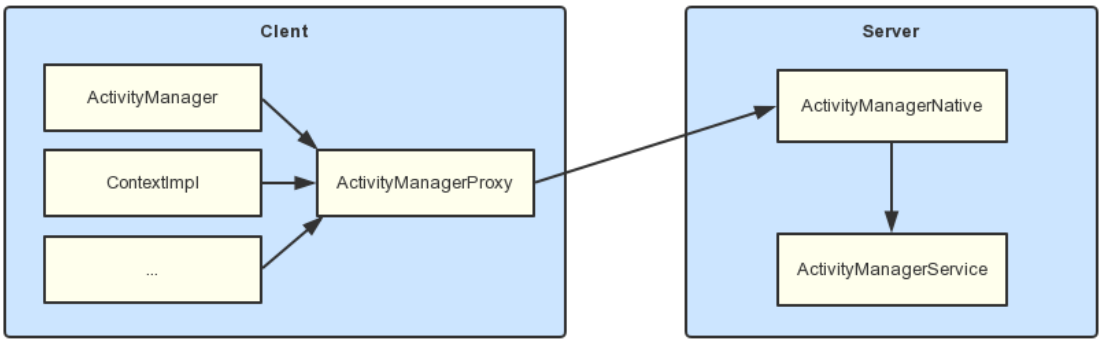
startActivityAsUser 方法最后会 return ActivityStarter 的 startActivityMayWait 方法，这一调用过程已经脱离了本节要讲的 AMS 家族，因此这里不做介绍了，具体的调用过程可以查看 Android 深入四大组件（一）应用程序启动过程（后篇）这篇文章。

在 Activity 的启动过程中提到了 AMP、AMN 和 AMS，它们共同组成了 AMS 家族的主要部分，如下图所示。



AMP 是 AMN 的内部类，它们都实现了 IActivityManager 接口，这样它们就可以实现代理模式，具体来讲是远程代理：AMP 和 AMN 是运行在两个进程的，AMP 是 Client 端，AMN 则是 Server 端，而 Server 端中具体的功能都是由 AMN 的子类 AMS 来实现的，因此，AMP 就是 AMS 在 Client 端的代理类。AMN 又实现了 Binder 类，这样 AMP 可以和 AMS 就可以通过 Binder 来进行进程间通信。

ActivityManager 通过 AMN 的 getDefault 方法得到 AMP，通过 AMP 就可以和 AMN 进行通信，也就是间接的与 AMS 进行通信。除了 ActivityManager，其他想要与 AMS 进行通信的类都需要通过 AMP，如下图所示。



PMS 之前言

PMS 的创建过程分为两个部分进行讲解，分别是 SyetemServer 处理部分和 PMS 构造方法。其中 SyetemServer 处理部分和 AMS 和 WMS 的创建过程是类似的，可以将它们进行对比，这样可以更好的理解和记忆这一知识点。

1. PMS 之 SyetemServer 处理部分

PMS 是在 SyetemServer 进程中被创建的，SyetemServer 进程用来创建系统服务，不了解它的可以查看 Android 系统启动流程（三）解析 SyetemServer 进程启动过程这篇文章。
从 SyetemServer 的入口方法 main 方法开始讲起，如下所示。
frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
      new SystemServer().run();
   }
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。
frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {
    try {
        ...
        //创建消息 Looper
         Looper.prepareMainLooper();
        //加载了动态库 libandroid_servers.so
        System.loadLibrary("android_servers");//1
        performPendingShutdown();
        // 创建系统的 Context
        createSystemContext();
```

```
        // 创建 SystemServiceManager
        mSystemServiceManager = new
SystemServiceManager(mSystemContext);//2
        mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
        LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
        SystemServerInitThreadPool.get();
    } finally {
        traceEnd();
    }
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices();//3
        //启动核心服务
        startCoreServices();//4
        //启动其他服务
        startOtherServices();//5
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "************************************************");
        Slog.e("System", "************ Failure starting system services",
ex);
        throw ex;
    } finally {
        traceEnd();
    }
    ...}
```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 中的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService。注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService、VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、4、5 的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和一些不需要立即启动的服务。这些系统服务总共有 100 多个，我们熟知的 AMS 属于引导服务，WMS 属于其他服务，本文要讲的 PMS 属于引导服务，因此这里列出引导服务以及它们的作用，见下表。

引导服务	作用

引导服务	作用
Installer	系统安装 apk 时的一个服务类，启动完成 Installer 服务之后才能启动其他的系统服务
ActivityManagerService	负责四大组件的启动、切换、调度。
PowerManagerService	计算系统中和 Power 相关的计算，然后决策系统应该如何反应
LightsService	管理和显示背光 LED
DisplayManagerService	用来管理所有显示设备
UserManagerService	多用户模式管理
SensorService	为系统提供各种感应器服务
PackageManagerService	用来对 apk 进行安装、解析、删除、卸载等等操作

查看启动引导服务的注释 3 处的 startBootstrapServices 方法。
frameworks/base/services/java/com/android/server/SystemServer.java

```
private void startBootstrapServices() {
    ...
    traceBeginAndSlog("StartPackageManagerService");
    mPackageManagerService = PackageManagerService.main(mSystemContext,
installer,
            mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
mOnlyCore);//1
    mFirstBoot = mPackageManagerService.isFirstBoot();//2
    mPackageManager = mSystemContext.getPackageManager();
    traceEnd();
    ...}
```

注释 1 处的 PMS 的 main 方法主要用来创建 PMS，其中最后一个参数 mOnlyCore 代表是否只扫描系统的目录，它在本篇文章中会出现多次，一般情况下它的值为 false。注释 2 处获取 boolean 类型的变量 mFirstBoot，它用于表示 PMS 是否首次被启动。mFirstBoot 是后续 WMS 创建时所需要的参数，从这里就可以看出系统服务之间是有依赖关系的，它们的启动顺序不能随意被更改。

2. PMS 构造方法

PMS 的 main 方法如下所示。
frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
  public static PackageManagerService main(Context context, Installer
installer,
          boolean factoryTest, boolean onlyCore) {
```

```
        PackageManagerServiceCompilerMapping.checkProperties();
        PackageManagerService m = new PackageManagerService(context,
installer,
                factoryTest, onlyCore);
        m.enableSystemUserPackages();
        ServiceManager.addService("package", m);
        return m;
    }
```

main 方法主要做了两件事，一个是创建 PMS 对象，另一个是将 PMS 注册到 ServiceManager 中。
PMS 的构造方法大概有 600 多行，分为 5 个阶段，每个阶段会打印出相应的 EventLog，EventLog 用于打印 Android 系统的事件日志。

1. BOOT_PROGRESS_PMS_START（开始阶段）
2. BOOT_PROGRESS_PMS_SYSTEM_SCAN_START（扫描系统阶段）
3. BOOT_PROGRESS_PMS_DATA_SCAN_START（扫描 Data 分区阶段）
4. BOOT_PROGRESS_PMS_SCAN_END（扫描结束阶段）
5. BOOT_PROGRESS_PMS_READY（准备阶段）

2.1 开始阶段

PMS 的构造方法中会获取一些包管理需要属性，如下所示。
frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
public PackageManagerService(Context context, Installer installer,
          boolean factoryTest, boolean onlyCore) {
        LockGuard.installLock(mPackages, LockGuard.INDEX_PACKAGES);
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "create package
manager");
        //打印开始阶段日志
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
                SystemClock.uptimeMillis())
        ...
        //用于存储屏幕的相关信息
        mMetrics = new DisplayMetrics();
        //Settings 用于保存所有包的动态设置
        mSettings = new Settings(mPackages);
        //在 Settings 中添加多个默认的 sharedUserId
        mSettings.addSharedUserLPw("android.uid.system",
Process.SYSTEM_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);//1
```

```
        mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        ...
        mInstaller = installer;
        //创建 Dex 优化工具类
        mPackageDexOptimizer = new PackageDexOptimizer(installer,
mInstallLock, context,
                "*dexopt*");
        mDexManager = new DexManager(this, mPackageDexOptimizer,
installer, mInstallLock);
        mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());
        mOnPermissionChangeListeners = new OnPermissionChangeListeners(
                FgThread.get().getLooper());
        getDefaultDisplayMetrics(context, mMetrics);
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "get system
config");
        //得到全局系统配置信息。
        SystemConfig systemConfig = SystemConfig.getInstance();
        //获取全局的 groupId
        mGlobalGids = systemConfig.getGlobalGids();
        //获取系统权限
        mSystemPermissions = systemConfig.getSystemPermissions();
        mAvailableFeatures = systemConfig.getAvailableFeatures();
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        mProtectedPackages = new ProtectedPackages(mContext);
        //安装 APK 时需要的锁，保护所有对 installd 的访问。
        synchronized (mInstallLock) {//1
        //更新 APK 时需要的锁，保护内存中已经解析的包信息等内容
        synchronized (mPackages) {//2
            //创建后台线程 ServiceThread
            mHandlerThread = new ServiceThread(TAG,
                    Process.THREAD_PRIORITY_BACKGROUND, true
/*allowIo*/);
            mHandlerThread.start();
            //创建 PackageHandler 绑定到 ServiceThread 的消息队列
            mHandler = new
PackageHandler(mHandlerThread.getLooper());//3
            mProcessLoggingHandler = new ProcessLoggingHandler();
            //将 PackageHandler 添加到 Watchdog 的检测集中
```

```
            Watchdog.getInstance().addThread(mHandler,
WATCHDOG_TIMEOUT);//4

            mDefaultPermissionPolicy = new
DefaultPermissionGrantPolicy(this);
            mInstantAppRegistry = new InstantAppRegistry(this);
            //在 Data 分区创建一些目录
            File dataDir = Environment.getDataDirectory();//5
            mAppInstallDir = new File(dataDir, "app");
            mAppLib32InstallDir = new File(dataDir, "app-lib");
            mAsecInternalPath = new File(dataDir, "app-asec").getPath();
            mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
            //创建多用户管理服务
            sUserManager = new UserManagerService(context, this,
                    new UserDataPreparer(mInstaller, mInstallLock,
mContext, mOnlyCore), mPackages);
             ...
               mFirstBoot
= !mSettings.readLPw(sUserManager.getUsers(false))//6
          ...     }
```

在开始阶段中创建了很多 PMS 中的关键对象并赋值给 PMS 中的成员变量，下面简单介绍这些成员变量。

- mSettings ：用于保存所有包的动态设置。注释 1 处将系统进程的 sharedUserId 添加到 Settings 中，sharedUserId 用于进程间共享数据，比如两个 App 的之间的数据是不共享的，如果它们有了共同的 sharedUserId，就可以运行在同一个进程中共享数据。
- mInstaller ：Installer 继承自 SystemService，和 PMS、AMS 一样是系统的服务（虽然名称不像是服务），PMS 很多的操作都是由 Installer 来完成的，比如 APK 的安装和卸载。在 Installer 内部，通过 IInstalld 和 installd 进行 Binder 通信，由位于 nativie 层的 installd 来完成具体的操作。
- systemConfig:用于得到全局系统配置信息。比如系统的权限就可以通过 SystemConfig 来获取。
- mPackageDexOptimizer ： Dex 优化的工具类。
- mHandler（PackageHandler 类型） ：PackageHandler 继承自 Handler，在注释 3 处它绑定了后台线程 ServiceThread 的消息队列。PMS 通过 PackageHandler 驱动 APK 的复制和安装工作，具体的请看在 Android 包管理机制（三）PMS 处理 APK 的安装这篇文章。

  PackageHandler 处理的消息队列如果过于繁忙，有可能导致系统卡住， 因此在注释 4 处将它添加到 Watchdog 的监测集中。

  Watchdog 主要有两个用途，一个是定时检测系统关键服务（AMS 和 WMS 等）是否可能发生死锁，还有一个是定时检测线程的消息队列是否长时间处于工作状态（可能阻塞等待了很长时间）。如果出现上述问题，Watchdog 会将日志保存起来，必要时还会杀掉自己所在的进程，也就是 SystemServer 进程。
- sUserManager（UserManagerService 类型） ：多用户管理服务。

除了创建这些关键对象，在开始阶段还有一些关键代码需要去讲解：

- 注释 1 处和注释 2 处加了两个锁，其中 mInstallLock 是安装 APK 时需要的锁，保护所有对 installd 的访问；mPackages 是更新 APK 时需要的锁，保护内存中已经解析的包信息等内容。
- 注释 5 处后的代码创建了一些 Data 分区中的子目录，比如/data/app。
- 注释 6 处会解析 packages.xml 等文件的信息，保存到 Settings 的对应字段中。packages.xml 中记录系统中所有安装的应用信息，包括基本信息、签名和权限。如果 packages.xml 有安装的应用信息，那么注释 6 处 Settings 的 readLPw 方法会返回 true，mFirstBoot 的值为 false，说明 PMS 不是首次被启动。

2.2 扫描系统阶段

```
...public PackageManagerService(Context context, Installer installer,
          boolean factoryTest, boolean onlyCore) {...
          //打印扫描系统阶段日志

EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
                startTime);
          ...
          //在/system 中创建 framework 目录
          File frameworkDir = new File(Environment.getRootDirectory(),
"framework");
          ...
          //扫描/vendor/overlay 目录下的文件
          scanDirTracedLI(new File(VENDOR_OVERLAY_DIR), mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR
                | PackageParser.PARSE_TRUSTED_OVERLAY, scanFlags |
SCAN_TRUSTED_OVERLAY, 0);
          mParallelPackageParserCallback.findStaticOverlayPackages();
          //扫描/system/framework 目录下的文件
          scanDirTracedLI(frameworkDir, mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR
                | PackageParser.PARSE_IS_PRIVILEGED,
                scanFlags | SCAN_NO_DEX, 0);
          final File privilegedAppDir = new
File(Environment.getRootDirectory(), "priv-app");
          //扫描 /system/priv-app 目录下的文件
          scanDirTracedLI(privilegedAppDir, mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR
                | PackageParser.PARSE_IS_PRIVILEGED, scanFlags, 0);
```

```
        final File systemAppDir = new
File(Environment.getRootDirectory(), "app");
        //扫描/system/app 目录下的文件
        scanDirTracedLI(systemAppDir, mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);
        File vendorAppDir = new File("/vendor/app");
        try {
            vendorAppDir = vendorAppDir.getCanonicalFile();
        } catch (IOException e) {
            // failed to look up canonical path, continue with original
one
        }
        //扫描 /vendor/app 目录下的文件
        scanDirTracedLI(vendorAppDir, mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

       //扫描/oem/app 目录下的文件
        final File oemAppDir = new File(Environment.getOemDirectory(),
"app");
        scanDirTracedLI(oemAppDir, mDefParseFlags
                | PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

        //这个列表代表有可能有升级包的系统 App
        final List<String> possiblyDeletedUpdatedSystemApps = new
ArrayList<String>();//1
        if (!mOnlyCore) {
            Iterator<PackageSetting> psit =
mSettings.mPackages.values().iterator();
            while (psit.hasNext()) {
                PackageSetting ps = psit.next();
                if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0)
{
                    continue;
                }
                //这里的 mPackages 的是 PMS 的成员变量，代表
scanDirTracedLI 方法扫描上面那些目录得到的
                final PackageParser.Package scannedPkg =
mPackages.get(ps.name);
                if (scannedPkg != null) {
                    if (mSettings.isDisabledSystemPackageLPr(ps.name))
{//2
```

```
                        ...
                            //将这个系统 App 的 PackageSetting 从 PMS 的
mPackages 中移除
                            removePackageLI(scannedPkg, true);
                            //将升级包的路径添加到 mExpectingBetter 列表中
                            mExpectingBetter.put(ps.name, ps.codePath);
                        }
                        continue;
                    }

                    if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
                       ...
                    } else {
                        final PackageSetting disabledPs =
mSettings.getDisabledSystemPkgLPr(ps.name);
                        //这个系统App升级包信息在mDisabledSysPackages中,
但是没有发现这个升级包存在
                        if (disabledPs.codePath == null
|| !disabledPs.codePath.exists()) {//5

possiblyDeletedUpdatedSystemApps.add(ps.name);//
                        }
                    }
                }
            }
            ...         }
```

/system 可以称作为 System 分区，里面主要存储谷歌和其他厂商提供的 Android 系统相关文件和框架。Android 系统架构分为应用层、应用框架层、系统运行库层（Native 层）、硬件抽象层（HAL 层）和 Linux 内核层，除了 Linux 内核层在 Boot 分区，其他层的代码都在 System 分区。下面列出 System 分区的部分子目录。

目录	含义
app	存放系统App，包括了谷歌内置的App也有厂商或者运营商提供的App
framework	存放应用框架层的jar包
priv-app	存放特权App
lib	存放so文件
fonts	存放系统字体文件
media	存放系统的各种声音，比如铃声、提示音，以及系统启动播放的动画

上面的代码还涉及到/vendor 目录，它用来存储厂商对 Android 系统的定制部分。

系统扫描阶段的主要工作有以下 3 点：

1. 创建/system 的子目录，比如/system/framework、/system/priv-app 和/system/app 等等
2. 扫描系统文件，比如/vendor/overlay、/system/framework、/system/app 等等目录下的文件。
3. 对扫描到的系统文件做后续处理。

主要来说第 3 点，一次 OTA 升级对于一个系统 App 会有三种情况：

- 这个系统 APP 无更新。
- 这个系统 APP 有更新。
- 新的 OTA 版本中，这个系统 APP 已经被删除。

当系统 App 升级，PMS 会将该系统 App 的升级包设置数据（PackageSetting）存储到 Settings 的 mDisabledSysPackages 列表中（具体见 PMS 的 replaceSystemPackageLIF 方法），mDisabledSysPackages 的类型为 `ArrayMap<String, PackageSetting>`。mDisabledSysPackages 中的信息会被 PMS 保存到 packages.xml 中的`<updated-package>`标签下（具体见 Settings 的 writeDisabledSysPackageLPr 方法）。
注释 2 处说明这个系统 App 有升级包，那么就将该系统 App 的 PackageSetting 从 mDisabledSysPackages 列表中移除，并将系统 App 的升级包的路径添加到 mExpectingBetter 列表中，mExpectingBetter 的类型为 `ArrayMap<String, File>` 等待后续处理。
注释 5 处如果这个系统 App 的升级包信息存储在 mDisabledSysPackages 列表中，但是没有发现这个升级包存在，则将它加入到 possiblyDeletedUpdatedSystemApps 列表中，意为“系统 App 的升级包可能被删除”，之所以是“可能”，是因为系统还没有扫描 Data 分区，只能暂放到 possiblyDeletedUpdatedSystemApps 列表中，等到扫描完 Data 分区后再做处理。

2.3 扫描 Data 分区阶段

```
public PackageManagerService(Context context, Installer installer,
          boolean factoryTest, boolean onlyCore) {
    ...
    mSettings.pruneSharedUsersLPw();
    //如果不是只扫描系统的目录，那么就开始扫描 Data 分区。
    if (!mOnlyCore) {
        //打印扫描 Data 分区阶段日志

EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
              SystemClock.uptimeMillis());
        //扫描/data/app 目录下的文件
        scanDirTracedLI(mAppInstallDir, 0, scanFlags |
SCAN_REQUIRE_KNOWN, 0);
        //扫描/data/app-private 目录下的文件
        scanDirTracedLI(mDrmAppPrivateInstallDir, mDefParseFlags
              | PackageParser.PARSE_FORWARD_LOCK,
              scanFlags | SCAN_REQUIRE_KNOWN, 0);
        //扫描完 Data 分区后，处理 possiblyDeletedUpdatedSystemApps 列表
        for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
            PackageParser.Package deletedPkg =
mPackages.get(deletedAppName);
            // 从 mSettings.mDisabledSysPackages 变量中移除去此应用
            mSettings.removeDisabledSystemPackageLPw(deletedAppName);
            String msg;
          //1：如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，说明
是残留的 App 信息，后续会删除它的数据。
            if (deletedPkg == null) {
                msg = "Updated system package " + deletedAppName
                      + " no longer exists; it's data will be wiped";
                // Actual deletion of code and data will be handled by later
                // reconciliation step
            } else {
            //2：如果这个系统 App 在 mPackages 中，说明是存在于 Data 分区，
不属于系统 App，那么移除其系统权限。
                msg = "Updated system app + " + deletedAppName
                      +"no longer present; removing system privileges for
"
                      + deletedAppName;
                deletedPkg.applicationInfo.flags &=
~ApplicationInfo.FLAG_SYSTEM;
```

```
                PackageSetting deletedPs =
mSettings.mPackages.get(deletedAppName);
                deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;
            }
            logCriticalInfo(Log.WARN, msg);
        }
         //遍历 mExpectingBetter 列表
        for (int i = 0; i < mExpectingBetter.size(); i++) {
            final String packageName = mExpectingBetter.keyAt(i);
            if (!mPackages.containsKey(packageName)) {
                //得到系统 App 的升级包路径
                final File scanFile = mExpectingBetter.valueAt(i);
                logCriticalInfo(Log.WARN, "Expected better " + packageName
                        + " but never showed up; reverting to system");
                int reparseFlags = mDefParseFlags;
                //3：根据系统 App 所在的目录设置扫描的解析参数
                if (FileUtils.contains(privilegedAppDir, scanFile)) {
                    reparseFlags = PackageParser.PARSE_IS_SYSTEM
                            | PackageParser.PARSE_IS_SYSTEM_DIR
                            | PackageParser.PARSE_IS_PRIVILEGED;
                }
                ...
                //将 packageName 对应的包设置数据（PackageSetting）添加到
mSettings 的 mPackages 中
                mSettings.enableSystemPackageLPw(packageName);//4
                try {
                    //扫描系统 App 的升级包
                    scanPackageTracedLI(scanFile, reparseFlags, scanFlags,
0, null);//5
                } catch (PackageManagerException e) {
                    Slog.e(TAG, "Failed to parse original system package:
"
                            + e.getMessage());
                }
            }
        }
    }
   //清除 mExpectingBetter 列表
    mExpectingBetter.clear();...}
```

/data 可以称为 Data 分区，它用来存储所有用户的个人数据和配置文件。下面列出 Data 分区部分子目录：

目录	含义
app	存储用户自己安装的App
data	存储所有已安装的App数据的目录，每个App都有自己单独的子目录
app-private	App的私有存储空间
app-lib	存储所有App的Jni库
system	存放系统配置文件
anr	用于存储ANR发生时系统生成的traces.txt文件

扫描 Data 分区阶段主要做了以下几件事：

1. 扫描/data/app 和/data/app-private 目录下的文件。
2. 遍历 possiblyDeletedUpdatedSystemApps 列表，注释 1 处如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，说明是残留的 App 信息，后续会删除它的数据。注释 2 处如果这个系统 App 的包信息在 mPackages 中，说明是存在于 Data 分区，不属于系统 App，那么移除其系统权限。
3. 遍历 mExpectingBetter 列表，注释 3 处根据系统 App 所在的目录设置扫描的解析参数，注释 4 处的方法内部会将 packageName 对应的包设置数据（PackageSetting）添加到 mSettings 的 mPackages 中。注释 5 处扫描系统 App 的升级包，最后清除 mExpectingBetter 列表。

2.4 扫描结束阶段

```
//打印扫描结束阶段日志
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
        SystemClock.uptimeMillis());
Slog.i(TAG, "Time to scan packages: "
        + ((SystemClock.uptimeMillis()-startTime)/1000f)
        + " seconds");
int updateFlags = UPDATE_PERMISSIONS_ALL;
// 如果当前平台 SDK 版本和上次启动时的 SDK 版本不同，重新更新 APK 的授权
if (ver.sdkVersion != mSdkVersion) {
    Slog.i(TAG, "Platform changed from " + ver.sdkVersion + " to "
            + mSdkVersion + "; regranting permissions for internal storage");
    updateFlags |= UPDATE_PERMISSIONS_REPLACE_PKG | UPDATE_PERMISSIONS_REPLACE_ALL;
}
```

```
            updatePermissionsLPw(null, null,
StorageManager.UUID_PRIVATE_INTERNAL, updateFlags);
            ver.sdkVersion = mSdkVersion;
           //如果是第一次启动或者是 Android M 升级后的第一次启动，需要初
始化所有用户定义的默认首选 App
            if (!onlyCore && (mPromoteSystemApps || mFirstBoot)) {
                for (UserInfo user : sUserManager.getUsers(true)) {
                    mSettings.applyDefaultPreferredAppsLPw(this,
user.id);
                    applyFactoryDefaultBrowserLPw(user.id);
                    primeDomainVerificationsLPw(user.id);
                }
            }
           ...
            //OTA 后的第一次启动，会清除代码缓存目录。
            if (mIsUpgrade && !onlyCore) {
                Slog.i(TAG, "Build fingerprint changed; clearing code
caches");
                for (int i = 0; i < mSettings.mPackages.size(); i++) {
                    final PackageSetting ps =
mSettings.mPackages.valueAt(i);
                    if
(Objects.equals(StorageManager.UUID_PRIVATE_INTERNAL, ps.volumeUuid))
{
                        clearAppDataLIF(ps.pkg, UserHandle.USER_ALL,
                                StorageManager.FLAG_STORAGE_DE |
StorageManager.FLAG_STORAGE_CE
                                        |
Installer.FLAG_CLEAR_CODE_CACHE_ONLY);
                    }
                }
                ver.fingerprint = Build.FINGERPRINT;
            }
            ...
           // 把 Settings 的内容保存到 packages.xml 中
            mSettings.writeLPr();
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
```

扫描结束结束阶段主要做了以下几件事：

1. 如果当前平台 SDK 版本和上次启动时的 SDK 版本不同，重新更新 APK 的授权。
2. 如果是第一次启动或者是 Android M 升级后的第一次启动，需要初始化所有用户定义的默认首选 App。
3. OTA 升级后的第一次启动，会清除代码缓存目录。

4. 把 Settings 的内容保存到 packages.xml 中，这样此后 PMS 再次创建时会读到此前保存的 Settings 的内容。

2.5 准备阶段

```
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
            SystemClock.uptimeMillis());
    ...
    mInstallerService = new PackageInstallerService(context, this);//1
    ...
    Runtime.getRuntime().gc();//2
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "loadFallbacks");
    FallbackCategoryProvider.loadFallbacks();
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    mInstaller.setWarnIfHeld(mPackages);
    LocalServices.addService(PackageManagerInternal.class, new
PackageManagerInternalImpl());//3
    Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);}
```

注释 1 处创建 PackageInstallerService，PackageInstallerService 是用于管理安装会话的服务，它会为每次安装过程分配一个 SessionId，在 Android 包管理机制（二）PackageInstaller 安装 APK 这篇文章中提到过 PackageInstallerService。
注释 2 处进行一次垃圾收集。注释 3 处将 PackageManagerInternalImpl（PackageManager 的本地服务）添加到 LocalServices 中，LocalServices 用于存储运行在当前的进程中的本地服务。

Activity 启动流程，App 启动流程

Activity 的启动模式

- 1.standard:默认标准模式，每启动一个都会创建一个实例，
- 2.singleTop：栈顶复用，如果在栈顶就调用 onNewIntent 复用，从 onResume()开始
- 3.singleTask：栈内复用，本栈内只要用该类型 Activity 就会将其顶部的 activity 出栈
- 4.singleInstance：单例模式，除了 3 中特性，系统会单独给该 Activity 创建一个栈，

1.什么是 Zygote 进程

1.1 简单介绍

- Zygote 进程是所有的 android 进程的父进程，包括 SystemServer 和各种应用进程都是通过 Zygote 进程 fork 出来的。Zygote（孵化）进程相当于是 android 系统的根进程，后面所有的进程都是通过这个进程 fork 出来的
- 虽然 Zygote 进程相当于 Android 系统的根进程，但是事实上它也是由 Linux 系统的 init 进程启动的。

1.2 各个进程的先后顺序

- init 进程 --> Zygote 进程 --> SystemServer 进程 -->各种应用进程

1.3 进程作用说明

- init 进程：linux 的根进程，android 系统是基于 linux 系统的，因此可以算作是整个 android 操作系统的第一个进程；
- Zygote 进程：android 系统的根进程，主要作用：可以作用 Zygote 进程 fork 出 SystemServer 进程和各种应用进程；
- SystemService 进程：主要是在这个进程中启动系统的各项服务，比如 ActivityManagerService，PackageManagerService，WindowManagerService 服务等等；
- 各种应用进程：启动自己编写的客户端应用时，一般都是重新启动一个应用进程，有自己的虚拟机与运行环境；

2.Zygote 进程的启动流程

2.1 源码位置

- 位置：frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
- Zygote 进程 mian 方法主要执行逻辑：
  - 初始化 DDMS；
  - 注册 Zygote 进程的 socket 通讯；
  - 初始化 Zygote 中的各种类，资源文件，OpenGL，类库，Text 资源等等；
  - 初始化完成之后 fork 出 SystemServer 进程；
  - fork 出 SystemServer 进程之后，关闭 socket 连接；

2.2 ZygoteInit 类的 main 方法

- init 进程在启动 Zygote 进程时一般都会调用 ZygoteInit 类的 main 方法，因此这里看一下该方法的具体实现(基于 android23 源码)；
  - 调用 enableDdms()，设置 DDMS 可用，可以发现 DDMS 启动的时机还是比较早的，在整个 Zygote 进程刚刚开始要启动额时候就设置可用。
  - 之后初始化各种参数
  - 通过调用 registerZygoteSocket 方法，注册为 Zygote 进程注册 Socket
  - 然后调用 preload 方法实现预加载各种资源
  - 然后通过调用 startSystemServer 开启 SystemServer 服务，这个是重点

```
public static void main(String argv[]) {
```

```
    try {
        //设置 ddms 可以用
        RuntimeInit.enableDdms();
        SamplingProfilerIntegration.start();
        boolean startSystemServer = false;
        String socketName = "zygote";
        String abiList = null;
        for (int i = 1; i < argv.length; i++) {
            if ("start-system-server".equals(argv[i])) {
                startSystemServer = true;
            } else if (argv[i].startsWith(ABI_LIST_ARG)) {
                abiList =
argv[i].substring(ABI_LIST_ARG.length());
            } else if (argv[i].startsWith(SOCKET_NAME_ARG))
{
                socketName =
argv[i].substring(SOCKET_NAME_ARG.length());
            } else {
                throw new RuntimeException("Unknown
command line argument: " + argv[i]);
            }
        }

        if (abiList == null) {
            throw new RuntimeException("No ABI list
supplied.");
        }

        registerZygoteSocket(socketName);

EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
            SystemClock.uptimeMillis());
        preload();

EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
            SystemClock.uptimeMillis());

SamplingProfilerIntegration.writeZygoteSnapshot();

        gcAndFinalize();
        Trace.setTracingEnabled(false);

        if (startSystemServer) {
```

```
            startSystemServer(abiList, socketName);
        }

        Log.i(TAG, "Accepting command socket
connections");
        runSelectLoop(abiList);

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

2.3 registerZygoteSocket(socketName)分析

- 调用 registerZygoteSocket（String socketName）为 Zygote 进程注册 socket

```
private static void registerZygoteSocket(String
socketName) {
    if (sServerSocket == null) {
        int fileDesc;
        final String fullSocketName =
ANDROID_SOCKET_PREFIX + socketName;
        try {
            String env = System.getenv(fullSocketName);
            fileDesc = Integer.parseInt(env);
        } catch (RuntimeException ex) {
            throw new RuntimeException(fullSocketName + "
unset or invalid", ex);
        }

        try {
            FileDescriptor fd = new FileDescriptor();
            fd.setInt$(fileDesc);
            sServerSocket = new LocalServerSocket(fd);
        } catch (IOException ex) {
            throw new RuntimeException(
                    "Error binding to local socket '" +
fileDesc + "'", ex);
        }
    }
}
```

2.4 preLoad()方法分析

- 源码如下所示

```
static void preload() {
    Log.d(TAG, "begin preload");
    preloadClasses();
    preloadResources();
    preloadOpenGL();
    preloadSharedLibraries();
    preloadTextResources();
    // Ask the WebViewFactory to do any initialization that
must run in the zygote process,
    // for memory sharing purposes.
    WebViewFactory.prepareWebViewInZygote();
    Log.d(TAG, "end preload");
}
```

- 大概操作是这样的：
  - preloadClasses()用于初始化 Zygote 中需要的 class 类；
  - preloadResources()用于初始化系统资源；
  - preloadOpenGL()用于初始化 OpenGL；
  - preloadSharedLibraries()用于初始化系统 libraries；
  - preloadTextResources()用于初始化文字资源；
  - prepareWebViewInZygote()用于初始化 webview;

2.5 startSystemServer()启动进程

- 这段逻辑的执行逻辑就是通过 Zygote fork 出 SystemServer 进程

```
private static boolean startSystemServer(String abiList,
String socketName)
        throws MethodAndArgsCaller, RuntimeException {
    long capabilities = posixCapabilitiesAsBits(
        OsConstants.CAP_BLOCK_SUSPEND,
        OsConstants.CAP_KILL,
        OsConstants.CAP_NET_ADMIN,
        OsConstants.CAP_NET_BIND_SERVICE,
        OsConstants.CAP_NET_BROADCAST,
        OsConstants.CAP_NET_RAW,
        OsConstants.CAP_SYS_MODULE,
        OsConstants.CAP_SYS_NICE,
        OsConstants.CAP_SYS_RESOURCE,
        OsConstants.CAP_SYS_TIME,
        OsConstants.CAP_SYS_TTY_CONFIG
```

```
        );
        /* Hardcoded command line to start the system server
    */
        String args[] = {
            "--setuid=1000",
            "--setgid=1000",

    "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1
    009,1010,1018,1021,1032,3001,3002,3003,3006,3007",
            "--capabilities=" + capabilities + "," +
    capabilities,
            "--nice-name=system_server",
            "--runtime-args",
            "com.android.server.SystemServer",
        };
        ZygoteConnection.Arguments parsedArgs = null;

        int pid;

        try {
            parsedArgs = new
    ZygoteConnection.Arguments(args);

    ZygoteConnection.applyDebuggerSystemProperty(parsedArg
    s);

    ZygoteConnection.applyInvokeWithSystemProperty(parsedA
    rgs);

            /* Request to fork the system server process */
            pid = Zygote.forkSystemServer(
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids,
                    parsedArgs.debugFlags,
                    null,
                    parsedArgs.permittedCapabilities,
                    parsedArgs.effectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        /* For child process */
        if (pid == 0) {
            if (hasSecondZygote(abiList)) {
```

```
                waitForSecondaryZygote(socketName);
            }

            handleSystemServerProcess(parsedArgs);
        }

        return true;
    }
```

-

3.SystemServer 进程启动流程

3.1 SystemServer 进程简介

- SystemServer 进程主要的作用是在这个进程中启动各种系统服务，比如 ActivityManagerService，PackageManagerService，WindowManagerService 服务，以及各种系统性的服务其实都是在 SystemServer 进程中启动的，而当我们的应用需要使用各种系统服务的时候其实也是通过与 SystemServer 进程通讯获取各种服务对象的句柄的。

3.2 SystemServer 的 main 方法

- 如下所示，比较简单，只是 new 出一个 SystemServer 对象并执行其 run 方法，查看 SystemServer 类的定义我们知道其实 final 类型的，所以我们一般不能重写或者继承。

```
/**
 * The main entry point from zygote.
 */                                          main方法入口
public static void main(String[] args) {
    new SystemServer().run();
}

                                                        更多可以看:
public SystemServer() {
    // Check for factory test mode.
    mFactoryTestMode = FactoryTest.getMode();
    // Remember if it's runtime restart(when sys.bo
    mRuntimeRestart = "1".equals(SystemProperties.g
}
```

-

3.3 查看 run 方法

- 代码如下所示

- 首先判断系统当前时间，若当前时间小于 1970 年 1 月 1 日，则一些初始化操作可能会处所，所以当系统的当前时间小于 1970 年 1 月 1 日的时候，设置系统当前时间为该时间点。
- 然后是设置系统的语言环境等
- 接着设置虚拟机运行内存，加载运行库，设置 SystemServer 的异步消息

```
private void run() {
    if (System.currentTimeMillis() <
EARLIEST_SUPPORTED_TIME) {
        Slog.w(TAG, "System clock is before 1970; setting
to 1970.");

SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TI
ME);
    }

    if
(!SystemProperties.get("persist.sys.language").isEmpty
()) {
        final String languageTag =
Locale.getDefault().toLanguageTag();

        SystemProperties.set("persist.sys.locale",
languageTag);
        SystemProperties.set("persist.sys.language",
"");
        SystemProperties.set("persist.sys.country", "");
        SystemProperties.set("persist.sys.localevar",
"");
    }

    Slog.i(TAG, "Entered the Android system server!");

EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_SYSTEM_
RUN, SystemClock.uptimeMillis());

    SystemProperties.set("persist.sys.dalvik.vm.lib.2",
VMRuntime.getRuntime().vmLibrary());

    if (SamplingProfilerIntegration.isEnabled()) {
        SamplingProfilerIntegration.start();
        mProfilerSnapshotTimer = new Timer();
        mProfilerSnapshotTimer.schedule(new TimerTask() {
            @Override
            public void run() {
```

```
SamplingProfilerIntegration.writeSnapshot("system_server", null);
                }
            }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
        }

        // Mmmmmm... more memory!
        VMRuntime.getRuntime().clearGrowthLimit();

        // The system server has to run all of the time, so it needs to be
        // as efficient as possible with its memory usage.

VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);

        // Some devices rely on runtime fingerprint generation, so make sure
        // we've defined it before booting further.
        Build.ensureFingerprintProperty();

        // Within the system server, it is an error to access Environment paths without
        // explicitly specifying a user.
        Environment.setUserRequired(true);

        // Ensure binder calls into the system always run at foreground priority.
        BinderInternal.disableBackgroundScheduling(true);

        // Prepare the main looper thread (this thread).
        android.os.Process.setThreadPriority(

android.os.Process.THREAD_PRIORITY_FOREGROUND);
        android.os.Process.setCanSelfBackground(false);
        Looper.prepareMainLooper();

        // Initialize native services.
        System.loadLibrary("android_servers");

        // Check whether we failed to shut down last time we tried.
        // This call may not return.
        performPendingShutdown();
```

```
    // Initialize the system context.
    createSystemContext();

    // Create the system service manager.
    mSystemServiceManager = new
SystemServiceManager(mSystemContext);
    LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);

    // Start services.
    try {
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    } catch (Throwable ex) {
        Slog.e("System",
"******************************************");
        Slog.e("System", "************ Failure starting
system services", ex);
        throw ex;
    }

    // For debug builds, log event loop stalls to dropbox
for analysis.
    if (StrictMode.conditionallyEnableDebugLogging()) {
        Slog.i(TAG, "Enabled StrictMode for system server
main thread.");
    }

    // Loop forever.
    Looper.loop();
    throw new RuntimeException("Main thread loop
unexpectedly exited");
}
```

- 然后下面的代码是：

```
// Initialize the system context.
createSystemContext();
// Create the system service manager.
mSystemServiceManager = new
SystemServiceManager(mSystemContext);
LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
// Start services.try {
```

```
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    } catch (Throwable ex) {
        Slog.e("System",
"************************************************");
        Slog.e("System", "************ Failure starting
system services", ex);
        throw ex;
    }
```

-

3.4 run 方法中 createSystemContext()解析

- 调用 createSystemContext()方法：
  - 可以看到在 SystemServer 进程中也存在着 Context 对象，并且是通过 ActivityThread.systemMain 方法创建 context 的，这一部分的逻辑以后会通过介绍 Activity 的启动流程来介绍，这里就不在扩展，只知道在 SystemServer 进程中也需要创建 Context 对象。

```
private void createSystemContext() {
    ActivityThread activityThread = ActivityThread.systemMain();
    mSystemContext = activityThread.getSystemContext();

mSystemContext.setTheme(android.R.style.Theme_DeviceDe
fault_Light_DarkActionBar);
}
```

3.5 mSystemServiceManager 的创建

- 看 run 方法中，通过 SystemServiceManager 的构造方法创建了一个新的 SystemServiceManager 对象，我们知道 SystemServer 进程主要是用来构建系统各种 service 服务的，而 SystemServiceManager 就是这些服务的管理对象。
- 然后调用：
  - 将 SystemServiceManager 对象保存 SystemServer 进程中的一个数据结构中。

```
LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
```

- 最后开始执行：

```
// Start services.try {
    startBootstrapServices();
    startCoreServices();
    startOtherServices();
} catch (Throwable ex) {
    Slog.e("System",
"************************************************");
```

```
    Slog.e("System", "************ Failure starting
system services", ex);
    throw ex;
}
```

-
- 里面主要涉及了是三个方法：
  - startBootstrapServices() 主要用于启动系统 Boot 级服务
  - startCoreServices() 主要用于启动系统核心的服务
  - startOtherServices() 主要用于启动一些非紧要或者是非需要及时启动的服务

4.启动服务

4.1 启动哪些服务

- 在开始执行启动服务之前总是会先尝试通过 socket 方式连接 Zygote 进程，在成功连接之后才会开始启动其他服务。

```
    // Start services.
    try {
        traceBeginAndSlog( name: "StartServices");
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "**************************
        Slog.e("System", "************ Failure startin
        throw ex;
    } finally {
        traceEnd();
    }
```

这里面是启动服务，接下来

更多可以

-

4.2 启动服务流程源码分析

- 首先看一下 startBootstrapServices 方法：

```
private void startBootstrapServices() {
    Installer installer =
mSystemServiceManager.startService(Installer.class);

    mActivityManagerService =
mSystemServiceManager.startService(
```

```
ActivityManagerService.Lifecycle.class).getService();

mActivityManagerService.setSystemServiceManager(mSyste
mServiceManager);
    mActivityManagerService.setInstaller(installer);

    mPowerManagerService =
mSystemServiceManager.startService(PowerManagerService
.class);

    mActivityManagerService.initPowerManagement();

    // Manages LEDs and display backlight so we need it to
bring up the display.

mSystemServiceManager.startService(LightsService.class
);

    // Display manager is needed to provide display metrics
before package manager
    // starts up.
    mDisplayManagerService =
mSystemServiceManager.startService(DisplayManagerServi
ce.class);

    // We need the default display before we can initialize
the package manager.

mSystemServiceManager.startBootPhase(SystemService.PHA
SE_WAIT_FOR_DEFAULT_DISPLAY);

    // Only run "core" apps if we're encrypting the device.
    String cryptState =
SystemProperties.get("vold.decrypt");
    if (ENCRYPTING_STATE.equals(cryptState)) {
        Slog.w(TAG, "Detected encryption in progress - only
parsing core apps");
        mOnlyCore = true;
    } else if (ENCRYPTED_STATE.equals(cryptState)) {
        Slog.w(TAG, "Device encrypted - only parsing core
apps");
        mOnlyCore = true;
    }
```

```
    // Start the package manager.
    Slog.i(TAG, "Package Manager");
    mPackageManagerService =
PackageManagerService.main(mSystemContext, installer,
            mFactoryTestMode !=
FactoryTest.FACTORY_TEST_OFF, mOnlyCore);
    mFirstBoot = mPackageManagerService.isFirstBoot();
    mPackageManager =
mSystemContext.getPackageManager();

    Slog.i(TAG, "User Service");
    ServiceManager.addService(Context.USER_SERVICE,
UserManagerService.getInstance());

    // Initialize attribute cache used to cache resources
from packages.
    AttributeCache.init(mSystemContext);

    // Set up the Application instance for the system
process and get started.
    mActivityManagerService.setSystemProcess();

    // The sensor service needs access to package manager
service, app ops
    // service, and permissions service, therefore we start
it after them.
    startSensorService();
}
```

- 先执行：

```
Installer installer =
mSystemServiceManager.startService(Installer.class);
```

- mSystemServiceManager 是系统服务管理对象，在 main 方法中已经创建完成，这里我们看一下其 startService 方法的具体实现：
  - 可以看到通过反射器构造方法创建出服务类，然后添加到 SystemServiceManager 的服务列表数据中，最后调用了 service.onStart()方法，因为传递的是 Installer.class

```
public <T extends SystemService> T startService(Class<T>
serviceClass) {
    final String name = serviceClass.getName();
    Slog.i(TAG, "Starting " + name);
```

```
    // Create the service.
    if
(!SystemService.class.isAssignableFrom(serviceClass)) {
        throw new RuntimeException("Failed to create " +
name
                + ": service must extend " +
SystemService.class.getName());
    }
    final T service;
    try {
        Constructor<T> constructor =
serviceClass.getConstructor(Context.class);
        service = constructor.newInstance(mContext);
    } catch (InstantiationException ex) {
        throw new RuntimeException("Failed to create
service " + name
                + ": service could not be instantiated", ex);
    } catch (IllegalAccessException ex) {
        throw new RuntimeException("Failed to create
service " + name
                + ": service must have a public constructor
with a Context argument", ex);
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException("Failed to create
service " + name
                + ": service must have a public constructor
with a Context argument", ex);
    } catch (InvocationTargetException ex) {
        throw new RuntimeException("Failed to create
service " + name
                + ": service constructor threw an exception",
ex);
    }

    // Register it.
    mServices.add(service);

    // Start it.
    try {
        service.onStart();
    } catch (RuntimeException ex) {
        throw new RuntimeException("Failed to start
service " + name
                + ": onStart threw an exception", ex);
```

```
        }
        return service;
    }
```

- 看一下 Installer 的 onStart 方法：
  - 很简单就是执行了 mInstaller 的 waitForConnection 方法，这里简单介绍一下 Installer 类，该类是系统安装 apk 时的一个服务类，继承 SystemService（系统服务的一个抽象接口），需要在启动完成 Installer 服务之后才能启动其他的系统服务。

```
@Overridepublic void onStart() {
    Slog.i(TAG, "Waiting for installd to be ready.");
    mInstaller.waitForConnection();
}
```

- 然后查看 waitForConnection（）方法：
  - 通过追踪代码可以发现，其在不断的通过 ping 命令连接 Zygote 进程（SystemServer 和 Zygote 进程通过 socket 方式通讯，其他进程通过 Binder 方式通讯）

```
public void waitForConnection() {
    for (;;) {
        if (execute("ping") >= 0) {
            return;
        }
        Slog.w(TAG, "installd not ready");
        SystemClock.sleep(1000);
    }
}
```

- 继续看 startBootstrapServices 方法：
  - 这段代码主要是用于启动 ActivityManagerService 服务，并为其设置 SysServiceManager 和 Installer。ActivityManagerService 是系统中一个非常重要的服务，Activity，service，Broadcast，contentProvider 都需要通过其余系统交互。

```
// Activity manager runs the show.mActivityManagerService
= mSystemServiceManager.startService(

ActivityManagerService.Lifecycle.class).getService();
mActivityManagerService.setSystemServiceManager(mSyste
mServiceManager);
mActivityManagerService.setInstaller(installer);
```

- 首先看一下 Lifecycle 类的定义：
  - 可以看到其实 ActivityManagerService 的一个静态内部类，在其构造方法中会创建一个 ActivityManagerService，通过刚刚对 Installer 服务的分析我们知道，SystemServiceManager 的 startService 方法会调用服务的 onStart()方法，而在 Lifecycle 类的定义中我们看到其 onStart()方法直接调用了 mService.start()方法，mService 是 Lifecycle 类中对 ActivityManagerService 的引用

```
public static final class Lifecycle extends SystemService
{
    private final ActivityManagerService mService;
```

```
        public Lifecycle(Context context) {
            super(context);
            mService = new ActivityManagerService(context);
        }

        @Override
        public void onStart() {
            mService.start();
        }

        public ActivityManagerService getService() {
            return mService;
        }
    }
```

4.3 启动部分服务

- 启动 PowerManagerService 服务：
  - 启动方式跟上面的 ActivityManagerService 服务相似都会调用其构造方法和 onStart 方法，PowerManagerService 主要用于计算系统中和 Power 相关的计算，然后决策系统应该如何反应。同时协调 Power 如何与系统其它模块的交互，比如没有用户活动时，屏幕变暗等等。

```
mPowerManagerService =
mSystemServiceManager.startService(PowerManagerService
.class);
```

  - 然后是启动 LightsService 服务
    - 主要是手机中关于闪光灯，LED 等相关的服务；也是会调用 LightsService 的构造方法和 onStart 方法；

```
mSystemServiceManager.startService(LightsService.cl
ass);
```

  - 然后是启动 DisplayManagerService 服务
    - 主要是手机显示方面的服务

```
mDisplayManagerService =
mSystemServiceManager.startService(DisplayManagerSe
rvice.class);
```

- **然后是启动 PackageManagerService，该服务也是 android 系统中一个比较重要的服务**
  - 包括多 apk 文件的安装，解析，删除，卸载等等操作。
  - 可以看到 PackageManagerService 服务的启动方式与其他服务的启动方式有一些区别，直接调用了 PackageManagerService 的静态 main 方法

```
Slog.i(TAG, "Package Manager");
mPackageManagerService =
PackageManagerService.main(mSystemContext, installer,
```

```
        mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
mOnlyCore);
mFirstBoot = mPackageManagerService.isFirstBoot();
mPackageManager = mSystemContext.getPackageManager();
```

- 看一下其 main 方法的具体实现：
  - 可以看到也是直接使用 new 的方式创建了一个 PackageManagerService 对象，并在其构造方法中初始化相关变量，最后调用了 ServiceManager.addService 方法，主要是通过 Binder 机制与 JNI 层交互

```
public static PackageManagerService main(Context context,
Installer installer,
        boolean factoryTest, boolean onlyCore) {
    PackageManagerService m = new
PackageManagerService(context, installer,
            factoryTest, onlyCore);
    ServiceManager.addService("package", m);
    return m;
}
```

- 然后查看 startCoreServices 方法：
  - 可以看到这里启动了 BatteryService（电池相关服务），UsageStatsService，WebViewUpdateService 服务等。

```
private void startCoreServices() {
    // Tracks the battery level.  Requires LightService.

mSystemServiceManager.startService(BatteryService.clas
s);

    // Tracks application usage stats.

mSystemServiceManager.startService(UsageStatsService.c
lass);
    mActivityManagerService.setUsageStatsManager(

LocalServices.getService(UsageStatsManagerInternal.cla
ss));
    // Update after UsageStatsService is available, needed
before performBootDexOpt.

mPackageManagerService.getUsageStatsIfNoPackageUsageIn
fo();

    // Tracks whether the updatable WebView is in a ready
state and watches for update installs.
```

```
mSystemServiceManager.startService(WebViewUpdateServic
e.class);
}
```

总结：

- SystemServer 进程是 android 中一个很重要的进程由 Zygote 进程启动；
- SystemServer 进程主要用于启动系统中的服务；
- SystemServer 进程启动服务的启动函数为 main 函数；
- SystemServer 在执行过程中首先会初始化一些系统变量，加载类库，创建 Context 对象，创建 SystemServiceManager 对象等之后才开始启动系统服务；
- SystemServer 进程将系统服务分为三类：boot 服务，core 服务和 other 服务，并逐步启动
- SertemServer 进程在尝试启动服务之前会首先尝试与 Zygote 建立 socket 通讯，只有通讯成功之后才会开始尝试启动服务；
- 创建的系统服务过程中主要通过 SystemServiceManager 对象来管理，通过调用服务对象的构造方法和 onStart 方法初始化服务的相关变量；
- 服务对象都有自己的异步消息对象，并运行在单独的线程中；

3. Binder 机制（IPC、AIDL 的使用）

1、什么是 AIDL 以及如何使用（★★★★）

①aidl 是 Android interface definition Language 的英文缩写，意思 Android 接口定义语言。

②使用 aidl 可以帮助我们发布以及调用远程服务，实现跨进程通信。

③将服务的 aidl 放到对应的 src 目录，工程的 gen 目录会生成相应的接口类

我们通过 bindService（Intent，ServiceConnect，int）方法绑定远程服务，在 bindService 中有一个 ServiceConnec 接口，我们需要覆写该类的 onServiceConnected(ComponentName,IBinder)方法，这个方法的第二个参数 IBinder 对象其实就是已经在 aidl 中定义的接口，因此我们可以将 IBinder

对象强制转换为 aidl 中的接口类。

我们通过 IBinder 获取到的对象（也就是 aidl 文件生成的接口）其实是系统产生的代理对象，该代理对象既可以跟我们的进程通信，又可以跟远程进程通信，作为一个中间的角色实现了进程间通信。

2、AIDL 的全称是什么?如何工作?能处理哪些类型的数据？（★★★）

AIDL 全称 Android Interface Definition Language（AndRoid 接口描述语言） 是一种接口描述语言; 编译器可以通过 aidl 文件生成一段代码，通过预先定义的接口达到两个进程内部通信进程跨界对象访问的目的。需要完成 2 件事情: 1. 引入 AIDL 的相关类.; 2. 调用 aidl 产生的 class.理论上，参数可以传递基本数据类型和 String，还有就是 Bundle 的派生类，不过在 Eclipse 中,目前的 ADT 不支持 Bundle 做为参数。

3. android 的 IPC 通信方式，线程（进程间）通信机制有哪些

- 1.ipc 通信方式：binder、contentprovider、socket
- 2.操作系统进程通讯方式：共享内存、socket、管道

4. 为什么使用 Parcelable，好处是什么？

实现机制

```
* Interface for classes whose instances can be written to
* and restored from a {@link Parcel}.  Classes implementing the Parcelable
* interface must also have a static field called <code>CREATOR</code>, which
* is an object implementing the {@link Parcelable.Creator Parcelable.Creator}
```

```
 * interface.
```

如上，摘自 Parcelable 注释：如果想要写入 Parcel 或者从中恢复，则必须 implements Parcelable 并且必须有一个 static field 而且名字必须是 CREATOR....

好吧，感觉好复杂。有如下疑问：

1、Parcelable 是干啥的？为什么需要它？

2、Parcel 又是干啥的？

3、如果是写入 Parcel 中、从 Parcelable 中恢复，那要 Parcelable 岂不是"多此一举"？

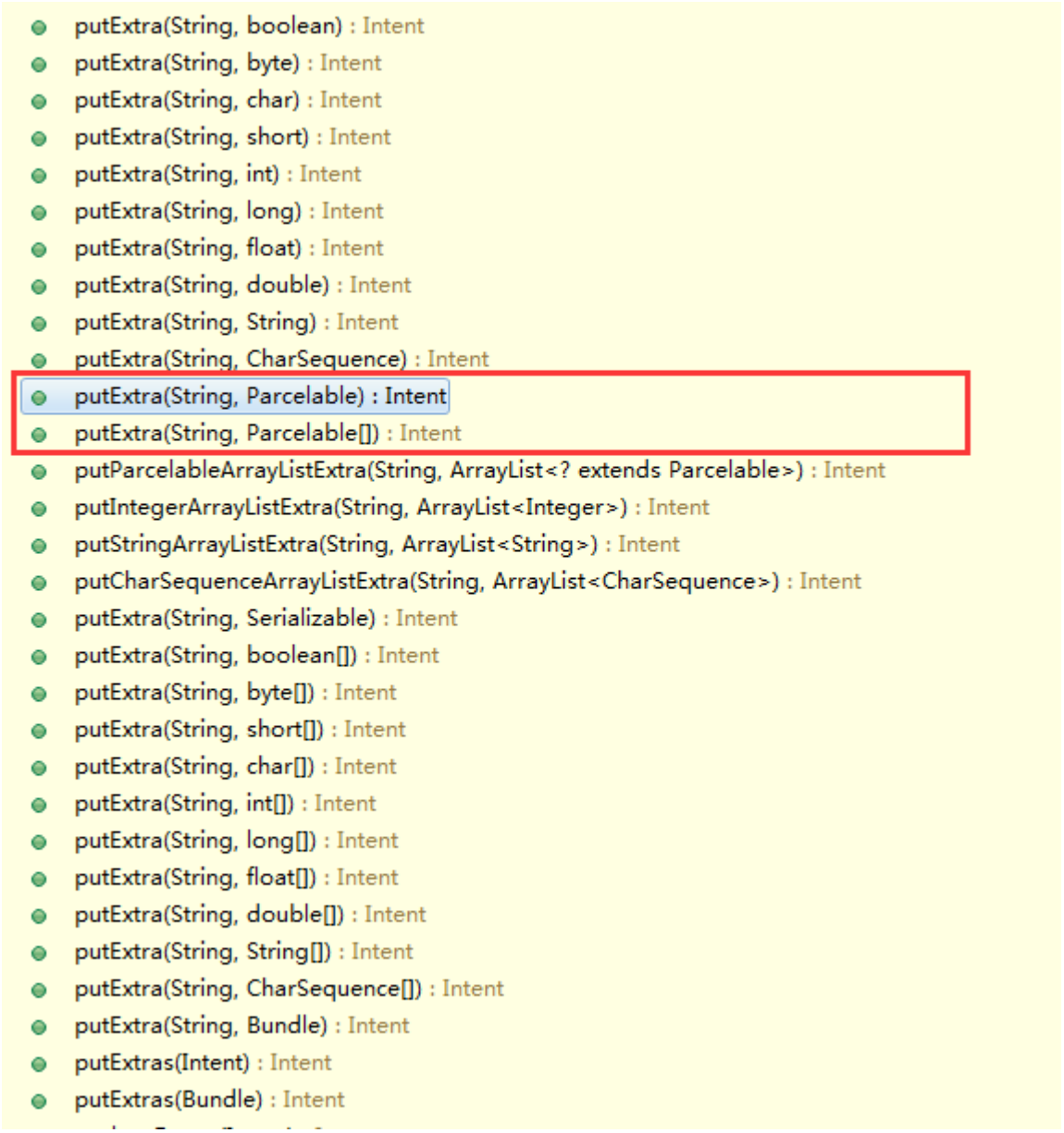
下面逐个回答上述问题：

1、Parcelable 是干啥的？从源码看：

```
public interface Parcelable {

    ...

    public void writeToParcel(Parcel dest, int flags);

    ...
```

简单来说，Parcelable 是一个 interface，有一个方法 writeToParcel(Parcel dest, int flags)，该方法接收两个参数，其中第一个参数类型是 Parcel。看起来 Parcelable 好像是对 Parcelable 的一种包装，从实际开发中，会在方法 writeToParcel 中调用 Parcel 的某些方法，完成将对象写入 Parcelable 的过程。

为什么往 Parcel 写入或恢复数据，需要继承 Parcelable 呢？我们看 Intent 的 putExtra 系列方法：



往 Intent 中添加数据，无法就是添加以上各种类型，简单的数据类型有对应的方法，如 putExtra(String, String)，复杂一点的有 putExtra(String, Bundle),putExtra(String, Serializable)、putExtra(String, Bundle)、putExtra(String, Parcelable)、putExtra(String, Parcelable[])。现在想想，如果往 Intent 里添加一个我们自定义的类型对象 person（Person 类的实例），咋整？总不能用 putExtra(String,person)吧？为啥，类型不符合啊！如果 Person 没有基础任何类，那它不可以用 putExtra 的任何一个方法，比较不存在 putExtra(String, Object）这样一个方法。

那为了可以用 putExtra 方法，Person 就需要继承一个可以用 putExtra 方法的类（接口），可以继承 Parcelable——继承其他类（接口）也没有问题。

现在捋一捋：为了使用 putExtra 方法，需要继承 Parcelable 类——事实上，还有更深的含义，且看后面。

2、Parcel 又是干啥的？前面说过，继承了 Parcelable 接口的类，如果不是抽象类，必须实现方法 writeToParcel，该方法有一个 Parcel 类型的参数，Parcel 源码：

```
public final class Parcel {
...
    public static Parcel obtain() {
        final Parcel[] pool = sOwnedPool;
        synchronized (pool) {
            Parcel p;
            for (int i=0; i<POOL_SIZE; i++) {
                p = pool[i];
                if (p != null) {
                    pool[i] = null;
                    if (DEBUG_RECYCLE) {
                        p.mStack = new RuntimeException();
                    }
                    return p;
                }
            }
        }
        return new Parcel(0);
    }
...public final native void writeInt(int val);
public final native void writeLong(long val);


...
```

Parcel 是一个 final 不可继承类，其代码很多，其中重要的一些部分是它有许多 native 的函数，在 writeToParcel 中调用的这些方法都直接或间接的调用 native 函数完成。

现在有一个问题，在 public void writeToParcel(Parcel dest, int flags)中调用 dest 的函数，这个 dest 是传入进来的，是形参，那实参在哪里？没有看到有什么地方生成了一个 Parcel 的实例，然后调用 writeToParcel 啊？？那它又不可能凭空出来。现在回到 Intent 这边来，看看它的内部做了什么事：

```
        Intent i = new Intent();
```

```
        Person person = new Person();
        i.putExtra("person", person);
        i.setClass(this, SecondeActivity.class);
        startActivity(i);
```

为了简单说明情况，我写了如上的代码，就不解释了。看看 putExtra 做了什么事情，看源码：

```
    public Intent putExtra(String name, Parcelable value) {
        if (mExtras == null) {
            mExtras = new Bundle();
        }
        mExtras.putParcelable(name, value);
        return this;
    }
```

这里调用的 putExtra 的第二个参数是 Parcelable 类型的，也印证了前面必须要类型符合（这里多说一句，面向对象的六大原则里有一个非常非常重要的"里氏替换"原则，子类出现的地方可以用父类代替，这样所有继承了 Parcelable 的类都可以传入这个 putExtra 中）。原来这里用到了 Bundle 类，看源码：

```
    public void putParcelable(String key, Parcelable value) {
        unparcel();
        mMap.put(key, value);
        mFdsKnown = false;
    }
```

mMap 是一个 Map。看到这里，原来我们传入的 person 被写入了 Map 里面了。这个 Bundle 也是继承自 Parcelable 的。其他 putExtra 系列的方法都是调用这个 mMap 的 put。为什么要用 Bundle 的类里的 Map？统一管理啊！所有传到 Intent 的 extra 我都不管，交给 Bundle 类来管理了，这样 Intent 类就不会太笨重（面向对象六大原则之迪米特原则——我不管你怎么整，整对了就行）。看 Bundle 源码：

```
public final class Bundle implements Parcelable, Cloneable {
    private static final String LOG_TAG = "Bundle";
    public static final Bundle EMPTY;
    //Bundle 类一加载就生成了一个 Bundle 实例
```

```
    static {
        EMPTY = new Bundle();
        EMPTY.mMap = Collections.unmodifiableMap(new HashMap<String, Object>());
    }    /* package */ Map<String, Object> mMap = null;
    /* package */ Parcel mParcelledData = null;
```

看到这里，还是没有发现 Parcel 实例在什么地方生成，继续往下看，看 startActivity 这个方法，找到最后会发现最终启动 Activity 的是一个 ActivityManagerNative 类，查看对应的方法：

```
    public int startActivity(IApplicationThread caller, Intent intent,
            String resolvedType, IBinder resultTo, String resultWho, int
requestCode,
            int startFlags, String profileFile,
            ParcelFileDescriptor profileFd, Bundle options) throws
RemoteException {
        Parcel data = Parcel.obtain(); //在这里生成了 Parcel 实例
        Parcel reply = Parcel.obtain(); //又生成了一个 Parcel 实例
        data.writeInterfaceToken(IActivityManager.descriptor);
        data.writeStrongBinder(caller != null ? caller.asBinder() : null);
        intent.writeToParcel(data, 0);
        data.writeString(resolvedType);
        data.writeStrongBinder(resultTo);
        data.writeString(resultWho);
        data.writeInt(requestCode);
        data.writeInt(startFlags);
        data.writeString(profileFile);
        if (profileFd != null) {
            data.writeInt(1);
            profileFd.writeToParcel(data,
Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            data.writeInt(0);
        }
        if (options != null) {
```

```
            data.writeInt(1);
            options.writeToParcel(data, 0);
        } else {
            data.writeInt(0);
        }
        mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);
        reply.readException();
        int result = reply.readInt();
        reply.recycle();
        data.recycle();
        return result;
    }
```

千呼万唤的 Parcel 对象终于出现了，这里生成了俩 Parcel 对象：data 和 reply，主要的是 data 这个实例。obtain 是一个 static 方法，用于从 Parcel 池（pool）中找出一个可用的 Parcel，如果都不可用，则生成一个新的。每一个 Parcel（Java）都与一个 C++的 Parcel 对应。

```
    public static Parcel obtain() {
        final Parcel[] pool = sOwnedPool;
        synchronized (pool) {
            Parcel p;
            for (int i=0; i<POOL_SIZE; i++) {
                p = pool[i];
                if (p != null) {
                    pool[i] = null;
                    if (DEBUG_RECYCLE) {
                        p.mStack = new RuntimeException();
                    }
                    return p;
                }
            }
        }
        return new Parcel(0);
```

```
    }
```

在 intent.writeToParcel(data, 0)里，查看源码：

```
    public void writeToParcel(Parcel out, int flags) {
        out.writeString(mAction);
        Uri.writeToParcel(out, mData);
        out.writeString(mType);
        out.writeInt(mFlags);
        out.writeString(mPackage);
        ComponentName.writeToParcel(mComponent, out);

        if (mSourceBounds != null) {
            out.writeInt(1);
            mSourceBounds.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }

        if (mCategories != null) {
            out.writeInt(mCategories.size());
            for (String category : mCategories) {
                out.writeString(category);
            }
        } else {
            out.writeInt(0);
        }

        if (mSelector != null) {
            out.writeInt(1);
            mSelector.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }
```

```
        if (mClipData != null) {
            out.writeInt(1);
            mClipData.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }

        out.writeBundle(mExtras); //终于把我们自定义的 person 实例送走了
    }
```

看到这里 Parcel 实例终于生成了，但是我们重写的从 Parcelable 接口而来的 writeToParcel 这个方法在什么地方被调用呢？从上面的 Intent 中的 out.writeBundle(mExtras)-->writeBundle(Bundle val)-->writeToParcel(Parcel parcel, int flags)-->writeMapInternal(Map<String,Object> val)-->writeValue(Object v)-->writeParcelable(Parcelable p, int parcelableFlags)（除了 out.writeBundle(mExtras)这个方法，其他的方法都是在 Bundle 和 Parcel 里面调来调去的，真心累！）：

```
    public final void writeParcelable(Parcelable p, int parcelableFlags) {
        if (p == null) {
            writeString(null);
            return;
        }
        String name = p.getClass().getName();
        writeString(name);
        p.writeToParcel(this, parcelableFlags);//调用自己实现的方法
    }
```

OK，终于出来了。。。到这里，写入的过程已经出来了。

那如何恢复呢？这里用到的是我们自己写的 createFromParcel 这个方法，从一个 Intent 中恢复 person：

```
        Intent i= getIntent();
        Person p = i.getParcelableExtra("person");
```

调啊调，调到这个：

```
    public final <T extends Parcelable> T readParcelable(ClassLoader loader) {
        String name = readString();
        if (name == null) {
            return null;
        }
        Parcelable.Creator<T> creator;
        synchronized (mCreators) {
            HashMap<String,Parcelable.Creator> map = mCreators.get(loader);
            if (map == null) {
                map = new HashMap<String,Parcelable.Creator>();
                mCreators.put(loader, map);
            }
            creator = map.get(name);
            if (creator == null) {
                try {
                    Class c = loader == null ?
                        Class.forName(name) : Class.forName(name, true, loader);
                    Field f = c.getField("CREATOR");
                    creator = (Parcelable.Creator)f.get(null);
                }
                catch (IllegalAccessException e) {
                    Log.e(TAG, "Class not found when unmarshalling: "
                                    + name + ", e: " + e);
                    throw new BadParcelableException(
                            "IllegalAccessException when unmarshalling: " + name);
                }
                catch (ClassNotFoundException e) {
                    Log.e(TAG, "Class not found when unmarshalling: "
                                    + name + ", e: " + e);
                    throw new BadParcelableException(
                            "ClassNotFoundException when unmarshalling: " + name);
                }
                catch (ClassCastException e) {
```

```
                    throw new BadParcelableException("Parcelable protocol requires
a "
                                        + "Parcelable.Creator object called "
                                        + " CREATOR on class " + name);
                }
                catch (NoSuchFieldException e) {
                    throw new BadParcelableException("Parcelable protocol requires
a "
                                        + "Parcelable.Creator object called "
                                        + " CREATOR on class " + name);
                }
                if (creator == null) {
                    throw new BadParcelableException("Parcelable protocol requires
a "
                                        + "Parcelable.Creator object called "
                                        + " CREATOR on class " + name);
                }

                map.put(name, creator);
            }
        }

        if (creator instanceof Parcelable.ClassLoaderCreator<?>) {
            return
((Parcelable.ClassLoaderCreator<T>)creator).createFromParcel(this, loader);
        }
        return creator.createFromParcel(this); //调用我们自定义的那个方法
    }
```

最后终于从我们自定义的方法中恢复那个保存的实例。

Android 图像显示相关流程，Vsync 信号等

Android Vsync 原理浅析

Preface

Android 中，Client 测量和计算布局，SurfaceFlienger（server）用来渲染绘制界面，client 和 server 的是通过匿名共享内存（SharedClient）通信。

每个应用和 SurfaceFlienger 之间都会创建一个 SharedClient，一个 SharedClient 最多可以创建 31 个 SharedBufferStack，每个 surface 对应一个 SharedBufferStack，也就是一个 Window。也就意味着，每个应用最多可以创建 31 个窗口。

Android 4.1 之后，AndroidOS 团队对 Android Display 进行了不断地进化和改变。引入了三个核心元素：Vsync，Triple Butter，Choreographer。

首先来理解一下，图形界面的绘制，大概是有 CPU 准备数据，然后通过驱动层把数据交给 GPU 来进行绘制。图形 API 不允许 CPU 和 GPU 直接通信，所以就有了图形驱动（Graphics Driver）来进行联系。Graphics Driver 维护了一个序列（Display List），CPU 不断把需要显示的数据放进去，GPU 不断取出来进行显示。

其中 Choreographer 起调度的作用。统一绘制图像到 Vsync 的某个时间点。

Choreographer 在收到 Vsync 信号时，调用用户设置的回调函数。函数的先后顺序如下：

CALLBACK_INPUT：与输入事件有关
CALLBACK_ANIMATION：与动画有关
CALLBACK_TRAVERSAL：与 UI 绘制有关

Vsync 是什么呢？首先来说一下什么是 FPS，FPS 就是 Frame Per Second（每秒的帧数）的缩写，我们知道，FPS>=60 时，我们就不会觉得动画卡顿。当 FPS=60 时是个什么概念呢？1000/60≈16.6，也就是说在大概 16ms 中，我们要进行一次屏幕的刷新绘制。Vsync 是垂直同步的缩写。这里我们可以简单的理解成，这就是一个时间中断。例如，每 16ms 会有一个 Vsync 信号，那么系统在每次拿到 Vsync 信号时刷新屏幕，我们就不会觉得卡顿了。

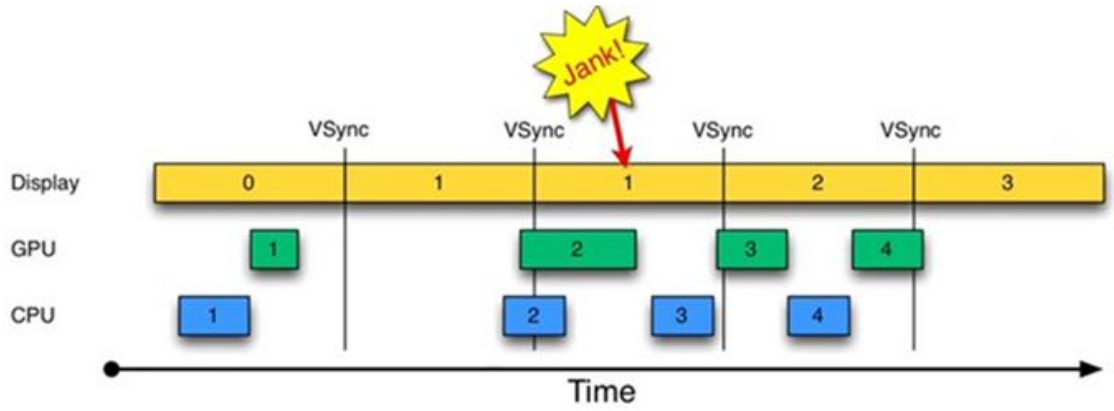
但实现起来还是有点困难的。

多重缓冲是什么技术呢？我们先来说双重缓冲。在 Linux 上，通常使用 FrameBuffer 来做显示输出。双重缓冲会创建一个 FrontBuffer 和一个 BackBuffer，顾名思义，FrontBuffer 是当前显示的页面，BackBuffer 是下一个要显示的画面。然后滚动电梯式显示数据。为什么呢？这样好在哪里呢？首先**他并不是不卡了，他还是会卡。**但是如果是单重缓冲，页面可能会有这种情况：A 面数据需要显示，

然后是 B 面数据显示，B 面数据显示需要耗费一定时间，但是这个时间里，C 面数据也请求了展示，我们可能会看到，在展示 C 面数据的时候，还有 B 面数据的残影...

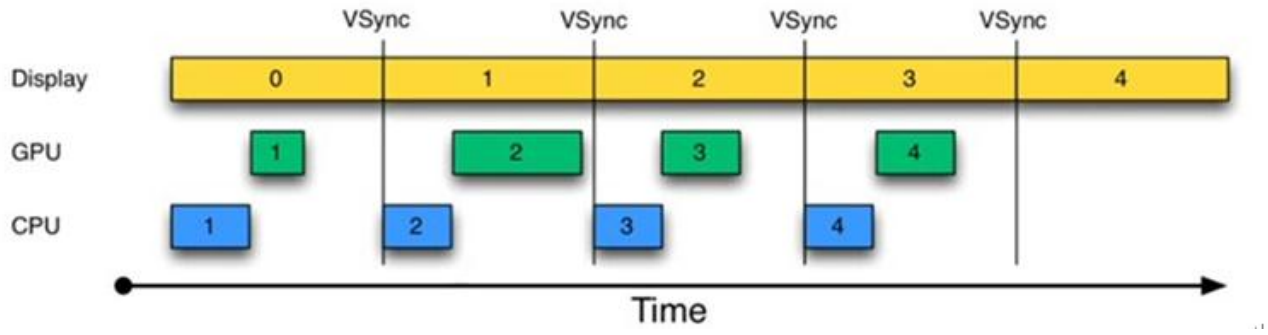
下面分情况来具体说明一下（Vsync 每 16 秒一次）。

1.没有使用 Vsync 的情况



可以看出，在第一个 16ms 之内，一切正常。然而在第二个 16ms 之内，几乎是在时间段的最后 CPU 才计算出了数据，交给了 Graphics Driver,导致 GPU 也是在第二段的末尾时间才进行了绘制，整个动作延后到了第三段内。从而影响了下一个画面的绘制。这时会出现 Jank（闪烁，可以理解为卡顿或者停顿）。那么在第二个 16ms 前半段的时间 CPU 和 GPU 干什么了？哦，他们可能忙别的事情了。这就是卡顿出现的原因和情况。CPU 和 GPU 很随意，爱什么时候刷新什么时候刷新，很随意。

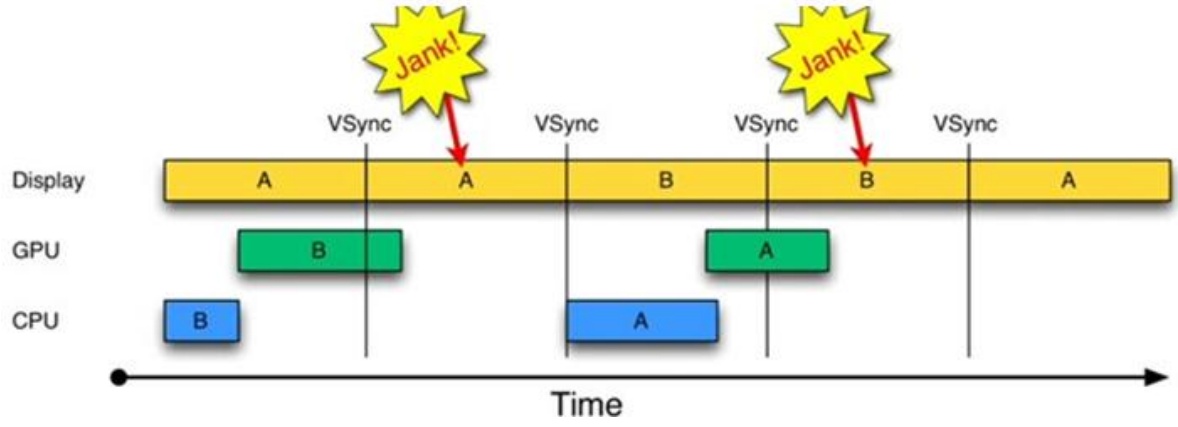
2.有 Vsync 的情况



如果，按照之前的前提来说，Vsync 每 16ms 一次，那么在每次发出 Vsync 命令时，CPU 都会进行刷新的操作。也就是在每个 16ms 的第一时间，CPU 就会想赢 Vsync 的命令，来进行数据刷新的动作。CPU 和 GPU 的刷新时间，和 Display 的 FPS 是一致的。因为只有到发出 Vsync 命令的时候，CPU 和 GPU 才会进行刷新或显示的动作。图中是正常情况。那么不正常情况是怎么个情况？我们先来说一下双重缓冲，然后再说。

3.双重缓冲

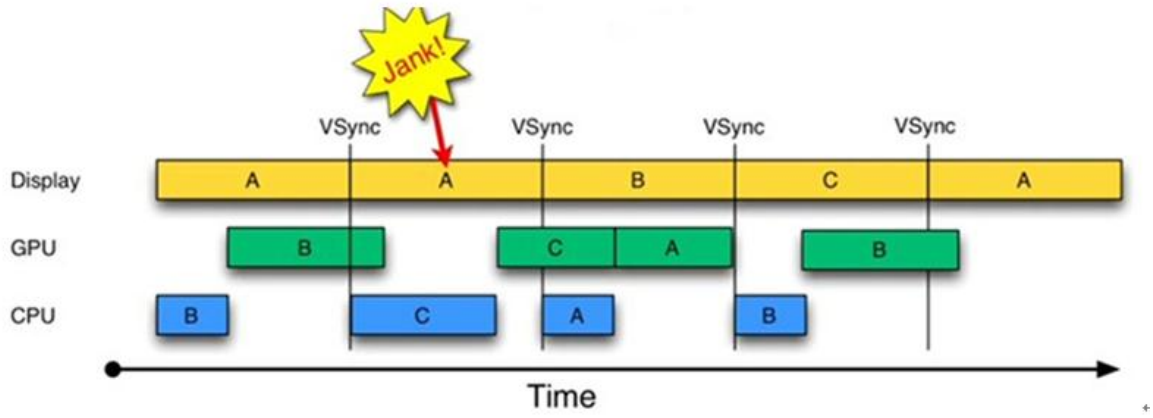
逻辑就是和之前一样。多重缓冲页面在 Back Buffer，然后根据需求来显示不同数据。但是会有什么问题呢（这就是 2 中提到的问题）？



首先我们看 Display 行，A 页面需要了两个时间单位，为什么？因为 B Buffer 在处理的时候太耗时了。然后导致了，在第一个 Vsync 发出的时候，还在 GPU 还在绘制 B Buffer。那么，刚好，第一个 Vsync 发出之后很短的时间，A 页面展示完了，B Buffer 的也在一开始的时候就不进行计算了。那么接下来的时间呢？屏幕还是展示着 B Buffer，这时候就会造成 Jank 现象。他不会动，因为他在等下一个 Vsync 过来的时候，才会显示下一个数据。

那么，如图，在 A Buffer 过来的时候，展示 B 页面的数据。这个时候！重复了上一个情况，也是太耗时了，然后又覆盖了下一个 Vysnc 发
出的时间，再次造成卡顿！依次类推，会造成多次卡顿。这个时候就有了三重缓冲的概念。

4.三重缓冲



首先看图。我们看到，B Buffer 依旧很耗时，同样覆盖了第一个 Vsync 发出的时间点。但是，在第一个 Vsync 发出的时候，C Buffer 站了出来，说，我来展示这个页面，你去缓冲 A 后面需要缓冲的页面吧！然后会发生什么？然后就是出现了一个 Jank…，但是这个 Jank 只在这一个时间单位出现，是可以忽略不计的。因为之后的逻辑都是顺畅的了。依次类推，除了 A 和 B 两个图层在交替显示，还有个"第三者"在不断帮他们两个可能需要展示的数据进行缓冲。但是注意了：

只有在需要时，才会进行三重缓冲。正常情况下，只使用二级缓冲！

另外，缓冲区不是越多越好。上图，C 页面在第四个时间段才展示出来，就是因为中间多了一个 Buffer（C Buffer）来进行缓冲。

但是，虽然谷歌给了你这么牛逼的前提逻辑，实际开发中你写的 APP 还是会卡，为什么呢？原因大概有两点：

1.界面太复杂。

2.主线程(UI 线程)太忙。他可能还在处理用户交互或者其他事情。

第三章：网络相关面试题

网络面试题

一、HTTP/HTTPS

1 、HTTP 与 与 HTTPS 有什么区别？

HTTPS 是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利
用 SSL/TLS 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份 认证，保护交换数据的隐私与完整性。
HTPPS 和 HTTP 的概念：
HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标
的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，

因此加密的详细内容就需要 SSL。它是一个 URI scheme（抽象标识符体系），句法类同 http: 体系。用于安全的 HTTP 数据传输。https:URL 表明它使用了 HTTP，但 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP 与 TCP 之间）。这个系统的最初研发由
网景公司进行，提供了身份验证与加密通讯方法，现在它被广泛用于万维网上安全敏感的通讯，例如交易支付方面。
超文本传输协议 (HTTP-Hypertext transfer protocol) 是一种详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。
https 协议需要到 ca 申请证书，一般免费证书很少，需要交费。http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议 http 和 https 使用的是完全不
同的连接方式用的端口也不一样,前者是 80,后者是 443。http 的连接很简单,是无状态的 HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议 要比 http 协
议安全 HTTPS 解决的问题：1．信任主机的问题. 采用 https 的 server 必须从 CA 申请一个
用于证明服务器用途类型的证书. 改证书只有用于对应的 server 的时候,客户度才信任次主机 2．防止通讯过程中的数据的泄密和被窜改
如下图所示，可以很明显的看出两个的区别：

HTTP　　HTTPS

注：TLS 是 SSL 的升级替代版，具体发展历史可以参考传输层安全性协议。
HTTP 与 HTTPS 在写法上的区别也是前缀的不同，客户端处理的方式也不同，具体说来：如果 URL 的协议是 HTTP，则客户端会打开一条到服务端端口 80（默认）的连接，并向其发送老的 HTTP 请求。如果 URL 的协议是 HTTPS，则客户端会打开一条到服务端端口 443（默认）的连接，然后与服务器握手，以二进制格式与服务器交换一些 SSL 的安全参数，附
上加密的 HTTP 请求。 所以你可以看到，HTTPS 比 HTTP 多了一层与 SSL 的连接，这也就
是客户端与服务端 SSL 握手的过程，整个过程主要完成以下工作：
交换协议版本号 选择一个两端都了解的密码 对两端的身份进行认证 生成临时的会话密钥，以便加密信道。 SSL 握手是一个相对比较复杂的过程，更多关于 SSL 握手的过程细节

可以参考 TLS/SSL 握手过程

SSL/TSL 的常见开源实现是 OpenSSL，OpenSSL 是一个开放源代码的软件库包，应用程序可以使用这个包来进行安全通信，避免窃听，同时确认另一端连接者的身份。这个包广泛被应用在互联网的网页服务器上。 更多源于 OpenSSL 的技术细节可以参考 OpenSSL。

2 、 Http1.1 和 Http1.0 及 2.0 的区别？

HTTP1.0 和 和 HTTP1.1 的一些区别

HTTP1.0 最早在网页中使用是在 1996 年，那个时候只是使用一些较为简单的网页上和网络请求上，而 HTTP1.1 则在 1999 年才开始广泛应用于现在的各大浏览器网络请求中，同时 HTTP1.1 也是当前使用最为广泛的 HTTP 协议。 主要区别主要体现在：

1、缓存处理，在 HTTP1.0 中主要使用 header 里的 If-Modified-Since,Expires 来做为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等更多可供选择的缓存头来控制缓存策略。

2、带宽优化及网络连接的使用，HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3、错误通知的管理，在 HTTP1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

4、Host 头处理，在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）。

5、长连接，HTTP 1.1 支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟，在 HTTP1.1 中默认开启 Connection： keep-alive，一定程度上弥补了 HTTP1.0 每次请求都要创建连接的缺点。

SPDY

在讲 Http1.1 和 Http2.0 的区别之前，还需要说下 SPDY，它是 HTTP1.x 的优化方案：2012 年 google 如一声惊雷提出了 SPDY 的方案，优化了 HTTP1.X 的请求延迟，解决了 HTTP1.X 的安全性，具体如下：

1、降低延迟，针对 HTTP 高延迟的问题，SPDY 优雅的采取了多路复用（multiplexing）。多路复用通过多个请求 stream 共享一个 tcp 连接的方式，解决了 HOL blocking 的问题，降低了延迟同时提高了带宽的利用率。

2、请求优先级（request prioritization）。多路复用带来一个新的问题是，在连接共享的基础之上有可能会导致关键请求被阻塞。SPDY 允许给每个 request 设置优先

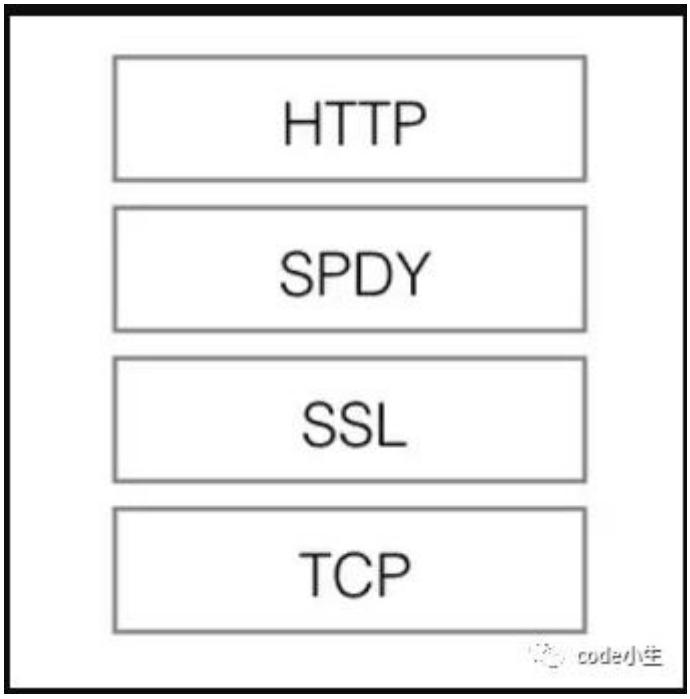
级，这样重要的请求就会优先得到响应。比如浏览器加载首页，首页的 html 内容应该优先展示，之后才是各种静态资源文件，脚本文件等加载，这样可以保证用户能第一时间看到网页内容。

3、header 压缩。前面提到 HTTP1.x 的 header 很多时候都是重复多余的。选择合适的压缩算法可以减小包的大小和数量。

4、基于 HTTPS 的加密协议传输，大大提高了传输数据的可靠性。

5、服务端推送（server push），采用了 SPDY 的网页，例如我的网页有一个 sytle.css 的请求，在客户端收到 sytle.css 数据的同时，服务端会将 sytle.js 的文件推送给客户端，当客户端再次尝试获取 sytle.js 时就可以直接从缓存中获取到，不用再发请求了。

SPDY 构成图：



SPDY 位于 HTTP 之下，TCP 和 SSL 之上，这样可以轻松兼容老版本的 HTTP 协议(将 HTTP1.x

的内容封装成一种新的 frame 格式)，同时可以使用已有的 SSL 功能。

HTTP2.0 和 和 HTTP1.X 相比的新特性

新的二进制格式（Binary Format），HTTP1.x 的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。

多路复用（MultiPlexing），即连接共享，即每一个 request 都是是用作连接共享机制的。一个 request 对应一个 id，这样一个连接上可以有多个 request，每个连接的 request 可以随机的混杂在一起，接收方可以根据 request 的 id 将 request 再归属到各自不同的服务端请求里面。

header 压缩，如上文中所言，对前面提到过 HTTP1.x 的 header 带有大量信息，而且每次都要重复发送，HTTP2.0 使用 encoder 来减少需要传输的 header 大小，通讯双方各自 cache 一份 header fields 表，既避免了重复 header 的传输，又减小了需要传输的大小。

服务端推送（server push），同 SPDY 一样，HTTP2.0 也具有 server push 功能。

需要更深的理解请点击这里

3 、s Https 请求慢的解决办法

1 、不通过 S DNS 解析，直接访问 IP

2 、解决连接无法复用

http/1.0 协议头里可以设置 Connection:Keep-Alive 或者 Connection:Close，选择是否允许在一定时间内复用连接（时间可由服务器控制）。但是这对 App 端的请求成效不大，因为 App 端的请求比较分散且时间跨度相对较大。

方案 1.基于 tcp 的长连接 (主要） 移动端建立一条自己的长链接通道，通道的实现是基于 tcp 协议。基于 tcp 的 socket 编程技术难度相对复杂很多，而且需要自己定制协议。但信息

的上报和推送变得更及时，请求量爆发的时间点还能减轻服务器压力（避免频繁创建和销毁连接）

方案 2.http long-polling 客户端在初始状态发送一个 polling 请求到服务器，服务器并不会马上返回业务数据，而是等待有新的业务数据产生的时候再返回，所以链接会一直被保持。一但结束当前连接，马上又会发送一个新的 polling 请求，如此反复，保证一个连接被保持。

存在问题： 1）增加了服务器的压力 2）网络环境复杂场景下，需要考虑怎么重建健康的连接通道 3)polling 的方式稳定性不好 4)polling 的 response 可能被中间代理 cache 住 ……

方案 3.http streaming 和 long-polling 不同的是，streaming 方式通过再 server response 的头部增加“Transfer Encoding:chuncked”来告诉客户端后续还有新的数据到来 存在问题：1）有些代理服务器会等待服务器的 response 结束之后才将结果推送给请求客户端。streaming 不会结束 response 2）业务数据无法按照请求分割 ……

方案 4.web socket 和传统的 tcp socket 相似，基于 tcp 协议，提供双向的数据通道。它的优势是提供了 message 的概念，比基于字节流的 tcp socket 使用更简单。技术较新，不是所有浏览器都提供了支持。

3 、解决 d head f of e line blocking

它的原因是队列的第一个数据包（队头）受阻而导致整列数据包受阻

使用 http pipelining，确保几乎在同一时间把 request 发向了服务器

4 、p Http 的 的 t request 和 和 e response 的协议组成

1 、Request 组成

客户端发送一个 HTTP 请求到服务器的请求消息包括以下格式：

请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。

请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本。

Get 请求例子

GET /562f25980001b1b106000338.jpg HTTP/1.1
Host img.mukewang.com
User-Agent Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/51.0.2704.106 Safari/537.36
Accept image/webp,image/*,*/*;q=0.8
Referer http://www.imooc.com/
Accept-Encoding gzip, deflate, sdch
Accept-Language zh-CN,zh;q=0.8

第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的 HTTP 版本. GET 说明请求类型为 GET,[/562f25980001b1b106000338.jpg]为要访问的资源，该行的最后一部分说明使用的是 HTTP1.1 版本。 第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息 从第二行起为请求头部，HOST 将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等 第三部分：空行，请求头部后面的空行是必须的 即使第四部分的请求数据为空，也必须有空行。 第四部分：请求数据也叫主体，可以添加任意的其他数据。 这个例子的请求数据为空。

POST 请求例子

POST / HTTP1.1
Host:www.wrox.com
User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
Content-Type:application/x-www-form-urlencoded
Content-Length:40
Connection: Keep-Alive
name=Professional%20Ajax&publisher=Wiley

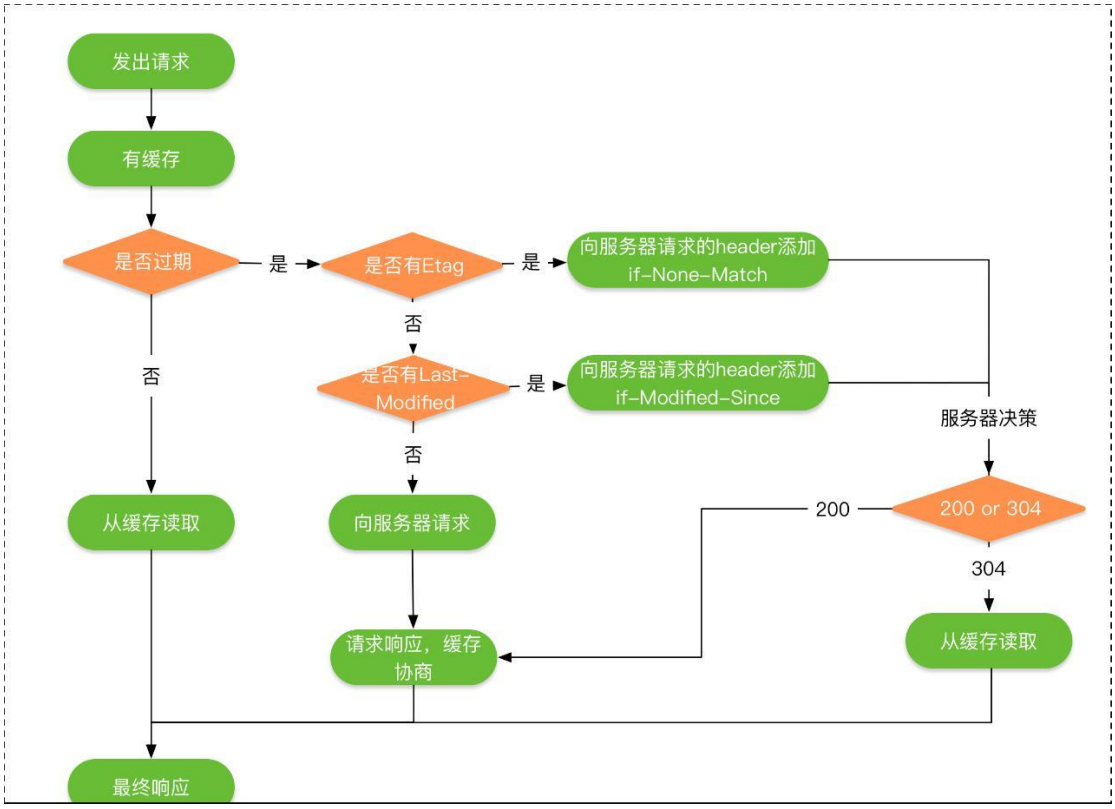
第一部分：请求行，第一行明了是 post 请求，以及 http1.1 版本。
第二部分：请求头部，第二行至第六行。
第三部分：空行，第七行的空行。
第四部分：请求数据，第八行。

2 、Response 组成

一般情况下，服务器接收并处理客户端发过来的请求后会返回一个 HTTP 的响应消息。
HTTP 响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。
第一部分：状态行，由 HTTP 协议版本号， 状态码， 状态消息 三部分组成。
第一行为状态行,（HTTP/1.1）表明 HTTP 版本为 1.1 版本，状态码为 200，状态消息为（ok）
第二部分：消息报头，用来说明客户端要使用的一些附加信息
第二行和第三行为消息报头， Date:生成响应的日期和时间；Content-Type:指定了 MIME 类型的 HTML(text/html),编码类型是 UTF-8
第三部分：空行，消息报头后面的空行是必须的
第四部分：响应正文，服务器返回给客户端的文本信息。
空行后面的 html 部分为响应正文。

5 、谈谈对 http 缓存的了解。

HTTP 的缓存机制也是依赖于请求和响应 header 里的参数类实现的，最终响应式从缓存中去，还是从服务端重新拉取，HTTP 的缓存机制的流程如下所示：



HTTP 的缓存可以分为两种：
强制缓存：需要服务端参与判断是否继续使用缓存，当客户端第一次请求数据是，服务端返回了缓存的过期时间（Expires 与 Cache-Control），没有过期就可以继续使用缓存，否则则不适用，无需再向服务端询问。 对比缓存：需要服务端参与判断是否继续使用缓存，当客户端第一次请求数据时，服务端会将缓存标识（Last-Modified/If-Modified-Since 与 Etag/If-None-Match）与数据一起返回给客户端，客户端将两者都备份到缓存中 ，再次请求数据时，客户端将上次备份的缓存 标识发送给服务端，服务端根据缓存标识进行判断，如果返回 304，则表示通知客户端可以继续使用缓存。 强制缓存优先于对比缓存。
上面提到强制缓存使用的的两个标识：

Expires：Expires 的值为服务端返回的到期时间，即下一次请求时，请求时间小于服务端返回的到期时间，直接使用缓存数据。到期时间是服务端生成的，客户端和服务端的时间可能有误差。 Cache-Control：Expires 有个时间校验的问题，所有 HTTP1.1 采用 Cache-Control 替代 Expires。 Cache-Control 的取值有以下几种：

private: 客户端可以缓存。 public: 客户端和代理服务器都可缓存。 max-age=xxx: 缓存的内容将在 xxx 秒后失效 no-cache: 需要使用对比缓存来验证缓存数据。 no-store: 所有内容都不会缓存，强制缓存，对比缓存都不会触发。 我们再来看看对比缓存的两个标识：
Last-Modified/If-Modified-Since
Last-Modified 表示资源上次修改的时间。
当客户端发送第一次请求时，服务端返回资源上次修改的时间：
Last-Modified: Tue, 12 Jan 2016 09:31:27 GMT
客户端再次发送，会在 header 里携带 If-Modified-Since。将上次服务端返回的资源时间上传给服务端。

If-Modified-Since: Tue, 12 Jan 2016 09:31:27 GMT

服务端接收到客户端发来的资源修改时间，与自己当前的资源修改时间进行对比，如果自己的资源修改时间大于客户端发来的资源修改时间，则说明资源做过修改， 则返回 200 表示需要重新请求资源，否则返回 304 表示资源没有被修改，可以继续使用缓存。

上面是一种时间戳标记资源是否修改的方法，还有一种资源标识码 ETag 的方式来标记是否修改，如果标识码发生改变，则说明资源已经被修改，ETag 优先级高于 Last-Modified。

Etag/If-None-Match

ETag 是资源文件的一种标识码，当客户端发送第一次请求时，服务端会返回当前资源的标识码：

ETag: "5694c7ef-24dc"

客户端再次发送，会在 header 里携带上次服务端返回的资源标识码：

If-None-Match:"5694c7ef-24dc" 服务端接收到客户端发来的资源标识码，则会与自己当前的资源吗进行比较，如果不同，则说明资源已经被修改，则返回 200，如果相同则说明资源没有被修改，返回 304，客户端可以继续使用缓存。

6 、 Http 长连接。

Http1.0 是短连接，HTTP1.1 默认是长连接，也就是默认 Connection 的值就是 keep-alive。但是长连接实质是指的 TCP 连接，而不是 HTTP 连接。TCP 连接是一个双向的通道，它是可

以保持一段时间不关闭的，因此 TCP 连接才有真正的长连接和短连接这一说。

Http1.1 为什么要用使用 tcp 长连接？

长连接是指的 TCP 连接，也就是说复用的是 TCP 连接。即长连接情况下，多个 HTTP 请求

可以复用同一个 TCP 连接，这就节省了很多 TCP 连接建立和断开的消耗。

此外，长连接并不是永久连接的。如果一段时间内（具体的时间长短，是可以在 header 当中进行设置的，也就是所谓的超时时间），这个连接没有 HTTP 请求发出的话，那么这个长连接就会被断掉。

需要更深的理解请点击这里

7 、 Https 加密原理。

加密算法的类型基本上分为了两种：

对称加密，加密用的密钥和解密用的密钥是同一个，比较有代表性的就是 AES 加密算法；

非对称加密，加密用的密钥称为公钥，解密用的密钥称为私钥，经常使用到的 RSA 加密算法就是非对称加密的；

此外，还有 Hash 加密算法

HASH 算法：MD5, SHA1, SHA256

相比较对称加密而言，非对称加密安全性更高，但是加解密耗费的时间更长，速度慢。

想了解更多加密算法请点击这里

HTTPS = HTTP + SSL，HTTPS 的加密就是在 SSL 中完成的。

这就要从 CA 证书讲起了。CA 证书其实就是数字证书，是由 CA 机构颁发的。至于 CA 机构的权威性，那么是毋庸置疑的，所有人都是信任它的。CA 证书内一般会包含以下内容：

证书的颁发机构、版本

证书的使用者

证书的公钥
证书的有效时间
证书的数字签名 Hash 值和签名 Hash 算法
...

验 客户端如何校验 CA 证书？

CA 证书中的 Hash 值，其实是用证书的私钥进行加密后的值（证书的私钥不在 CA 证书中）。然后客户端得到证书后，利用证书中的公钥去解密该 Hash 值，得到 Hash-a ；然后再利用证书内的签名 Hash 算法去生成一个 Hash-b 。最后比较 Hash-a 和 Hash-b 这两个的值。如果相等，那么证明了该证书是对的，服务端是可以被信任的；如果不相等，那么就说明该证书是错误的，可能被篡改了，浏览器会给出相关提示，无法建立起 HTTPS 连接。除此之外，还会校验 CA 证书的有效时间和域名匹配等。

HTTPS 的 中的 SSL 握手建立过程

假设现在有客户端 A 和服务器 B ：

1 、 首 先 ， 客 户 端 A 访 问 服 务 器 B ， 比 如 我 们 用 浏 览 器 打 开 一 个 网

页 www.baidu.com ，这时，浏览器就是客户端 A ，百度的服务器就是服务器 B 了。这时候客户端 A 会生成一个随机数 1，把随机数 1 、自己支持的 SSL 版本号以及加密算法等这些信息告诉服务器 B 。

2、服务器 B 知道这些信息后，然后确认一下双方的加密算法，然后服务端也生成一个随机数 B ，并将随机数 B 和 CA 颁发给自己的证书一同返回给客户端 A 。

3、客户端 A 得到 CA 证书后，会去校验该 CA 证书的有效性，校验方法在上面已经说过了。校验通过后，客户端生成一个随机数 3 ，然后用证书中的公钥加密随机数 3 并传输给服务端 B 。

4、服务端 B 得到加密后的随机数 3，然后利用私钥进行解密，得到真正的随机数 3。

5、最后，客户端 A 和服务端 B 都有随机数 1、随机数 2、随机数 3，然后双方利用这三个随机数生成一个对话密钥。之后传输内容就是利用对话密钥来进行加解密了。这时就是利用了对称加密，一般用的都是 AES 算法。

6、客户端 A 通知服务端 B ，指明后面的通讯用对话密钥来完成，同时通知服务器 B 客户端 A 的握手过程结束。

7、服务端 B 通知客户端 A，指明后面的通讯用对话密钥来完成，同时通知客户端 A 服务器 B 的握手过程结束。

8、SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户端 A 和服务器 B 开始使用相同的对话密钥进行数据通讯。

简化如下：

1、客户端和服务端建立 SSL 握手，客户端通过 CA 证书来确认服务端的身份；

2、互相传递三个随机数，之后通过这随机数来生成一个密钥；

3、互相确认密钥，然后握手结束；

4、数据通讯开始，都使用同一个对话密钥来加解密；

可以发现，在 HTTPS 加密原理的过程中把对称加密和非对称加密都利用了起来。即利用了非对称加密安全性高的特点，又利用了对称加密速度快，效率高的好处。

需要更深的理解请点击这里

8 、S HTTPS 如何防范中间人攻击？

什么是中间人攻击？

当数据传输发生在一个设备（PC/手机）和网络服务器之间时，攻击者使用其技能和工具将自己置于两个端点之间并截获数据；尽管交谈的两方认为他们是在与对方交谈，但是实际上他们是在与干坏事的人交流，这便是中间人攻击。

有几种攻击方式？

1、嗅探：嗅探或数据包嗅探是一种用于捕获流进和流出系统/网络的数据包的技术。网络中的数据包嗅探就好像电话中的监听。

2、数据包注入： 在这种技术中，攻击者会将恶意数据包注入常规数据中。这样用户便不会注意到文件/恶意软件，因为它们是合法通讯流的一部分。

3、会话劫持： 在你登录进你的银行账户和退出登录这一段期间便称为一个会话。这些会话通常都是黑客的攻击目标，因为它们包含潜在的重要信息。在大多数案例中，黑客会潜伏在会话中，并最终控制它。

4、SSL 剥离： 在 SSL 剥离攻击中，攻击者使 SSL/TLS 连接剥落，随之协议便从安全的 HTTPS 变成了不安全的 HTTP。

S HTTPS 如何防范中间人攻击：

请见 https 加密原理。

9 、有哪些响应码，分别都代表什么意思？

1** 信息，服务器收到请求，需要请求者继续执行操作

2** 成功，操作被成功接收并处理

3** 重定向，需要进一步的操作以完成请求

4** 客户端错误，请求包含语法错误或无法完成请求

5** 服务器错误，服务器在处理请求的过程中发生了错误

二、 TCP/UDP

1 、为什么 p tcp 要经过三次握手，四次挥手？

重要标志位

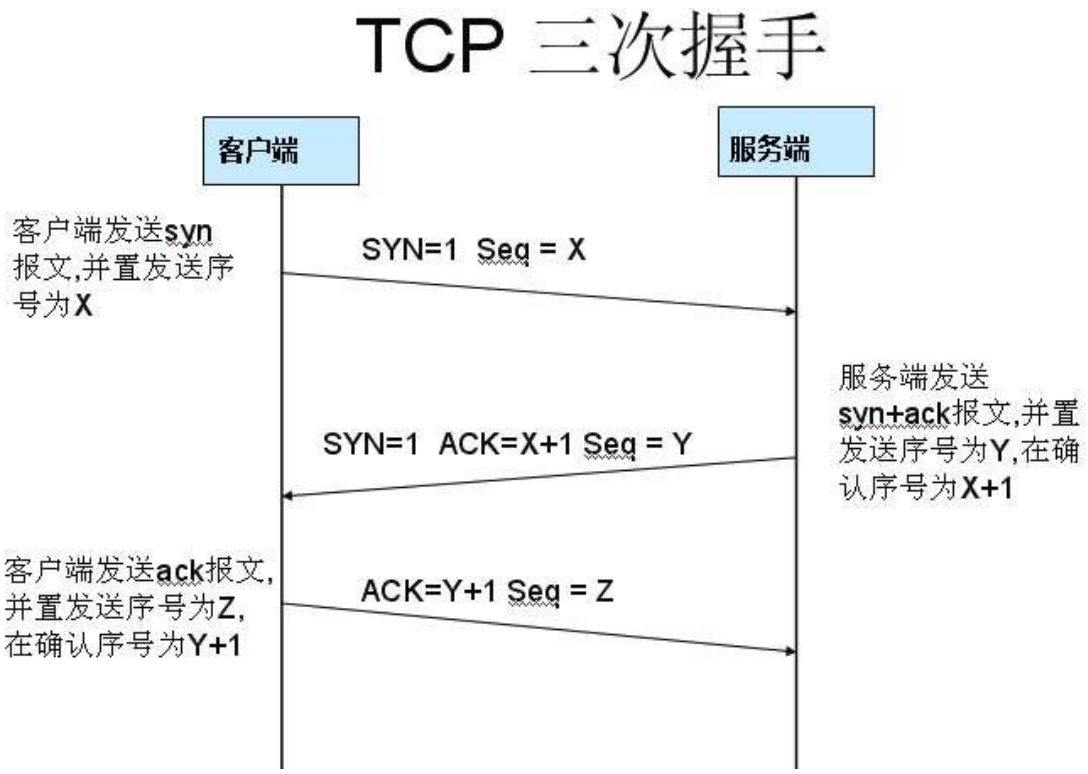
ACK ： TCP 协议规定，只有 ACK=1 时有效，也规定连接建立后所有发送的报文的 ACK 必须为 1

SYN(SYNchronization) ： 在连接建立时用来同步序号。当 SYN=1 而 ACK=0 时，表明这是一个连接请求报文。对方若同意建立连接，则应在响应报文中使 SYN=1 和 ACK=1. 因此, SYN 置 1 就表示这是一个连接请求或连接接受报文。

FIN （finis）即完，终结的意思， 用来释放一个连接。当 FIN = 1 时，表明此报文段的发送方的数据已经发送完毕，并要求释放连接。

三次握手、四次挥手过程

三次握手：



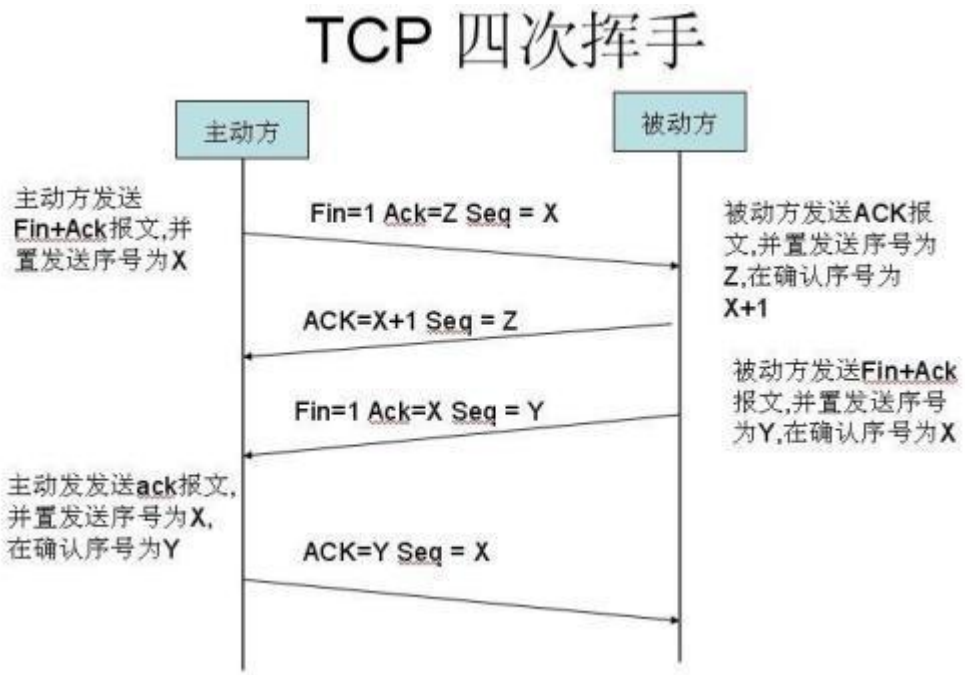
第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 位置为 1，Sequence Number 为 x；然后，客户端进入 SYN_SEND 状态，等待服务器的确认；

第二次握手：服务器收到 SYN 报文段。服务器收到客户端的 SYN 报文段，需要对这个 SYN 报文段进行确认，设置 Acknowledgment Number 为 x+1(Sequence Number+1)；同时，自己自己还要发送 SYN 请求信息，将 SYN 位置为 1，Sequence Number 为 y；服务器端将

上述所有信息放到一个报文段（即 SYN+ACK 报文段）中，一并发送给客户端，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 报文段。然后将 Acknowledgment Number 设置为 y+1，向服务器发送 ACK 报文段，这个报文段发送完毕以后，客户端和服务器端都进入 ESTABLISHED 状态，完成 TCP 三次握手。

四次挥手：



第一次分手：主机 1（可以使客户端，也可以是服务器端），设置 Sequence Number 和 Acknowledgment Number，向主机 2 发送一个 FIN 报文段；此时，主机 1 进入 FIN_WAIT_1 状态；这表示主机 1 没有数据要发送给主机 2 了；

第二次分手：主机 2 收到了主机 1 发送的 FIN 报文段，向主机 1 回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加 1；主机 1 进入 FIN_WAIT_2 状态；主机 2 告诉主机 1，我“同意”你的关闭请求；

第三次分手：主机 2 向主机 1 发送 FIN 报文段，请求关闭连接，同时主机 2 进入 LAST_ACK

状态；

第四次分手：主机 1 收到主机 2 发送的 FIN 报文段，向主机 2 发送 ACK 报文段，然后主机

1 进入 TIME_WAIT 状态；主机 2 收到主机 1 的 ACK 报文段以后，就关闭连接；此时，主机

1 等待 2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，主机 1 也可以关闭

连接了。

“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生

错误”。主要目的防止 server 端一直等待，浪费资源。换句话说，即是为了保证服务端能收接受到客户端的信息并能做出正确的应答而进行前两次(第一次和第二次)握手，为了保证客户端能够接收到服务端的信息并能做出正确的应答而进行后两次(第二次和第三次)握手。

“四次挥手”原因是因为 tcp 是全双工模式，接收到 FIN 时意味将没有数据再发来，但是还是

可以继续发送数据。

2 2 、P TCP 可靠传输原理实现（滑动窗口）。

确认和重传：接收方收到报文后就会进行确认，发送方一段时间没有收到确认就会重传。

数据校验。

数据合理分片与排序，TCP 会对数据进行分片，接收方会缓存为按序到达的数据，重新排序后再提交给应用层。

流程控制：当接收方来不及接收发送的数据时，则会提示发送方降低发送的速度，防止包丢失。

拥塞控制：当网络发生拥塞时，减少数据的发送。

关于滑动窗口、流量控制、拥塞控制实现原理请点击这里

3 、p Tcp 和 和 p Udp 的区别？

1、基于连接与无连接；

2、对系统资源的要求（TCP 较多，UDP 少）；

3、UDP 程序结构较简单；

4、流模式与数据报模式 ；

5、TCP 保证数据正确性，UDP 可能丢包；

6、TCP 保证数据顺序，UDP 不保证。

4 、如何设计在 P UDP 证 上层保证 P UDP 的可靠性传输？

传输层无法保证数据的可靠传输，只能通过应用层来实现了。实现的方式可以参照 tcp 可靠

性传输的方式。如不考虑拥塞处理，可靠 UDP 的简单设计如下：

1、添加 seq/ack 机制，确保数据发送到对端

2、添加发送和接收缓冲区，主要是用户超时重传。

3、添加超时重传机制。

具体过程即是：送端发送数据时，生成一个随机 seq=x，然后每一片按照数据大小分配 seq。数据到达接收端后接收端放入缓存，并发送一个 ack=x 的包，表示对方已经收到了数据。发送端收到了 ack 包后，删除缓冲区对应的数据。时间到后，定时任务检查是否需要重传数

据。

目前有如下开源程序利用 udp 实现了可靠的数据传输。分别为 RUDP、RTP、UDT:

1、RUDP（Reliable User Datagram Protocol）

RUDP 提供一组数据服务质量增强机制，如拥塞控制的改进、重发机制及淡化服务器算法等。

2、RTP（Real Time Protocol）

RTP 为数据提供了具有实时特征的端对端传送服务，如在组播或单播网络服务下的交互式视频音频或模拟数据。

3、UDT（UDP-based Data Transfer Protocol）

UDT 的主要目的是支持高速广域网上的海量数据传输。

关于 RUDP、RTP、UDT 的更多介绍请查看此处

三、其它重要网络概念

1 、t socket 断线重连怎么实现，心跳机制又是怎样实现？

socket 概念

套接字（socket）是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。

为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP / IP 协议交

互

提供了套接字(Socket)接口。应 用层可以和传输层通过 Socket 接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

建立 t socket 连接

建立 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ，另一个运行于服务器端，称为 ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端--套接字的地址和端口号，然后就向服务器端套接字提出连接请求。 连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发 给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

SOCKET 连接与 TCP 连接

创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（TCP 或 UDP），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。

Socket 连接与 HTTP 连接

由于通常情况下 Socket 连接就是 TCP 连接，因此 Socket 连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网 络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。

而 HTTP 连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户

端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是 Socket 连接，服务器就可以直接将数 据传送给客户端；若双方建立的

是 HTTP 连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端，因此，客户端定时向服务器端发送连接请求， 不仅可以保持在线，同时也是在“询问”服务器是否有新的数据，如果有就将数据传给客户端。TCP(Transmission Control Protocol) 传输控制协议

socket 断线重连实现

正常连接断开客户端会给服务端发送一个 fin 包，服务端收到 fin 包后才会知道连接断开。而

断网断电时客户端无法发送 fin 包给服务端，所以服务端没办法检测到客户端已经短线。为

了缓解这个问题，服务端需要有个心跳逻辑，就是服务端检测到某个客户端多久没发送任何数据过来就认为客户端已经断开， 这需要客户端定时向服务端发送心跳数据维持连接。

心跳机制实现

长连接的实现：心跳机制，应用层协议大多都有 HeartBeat 机制，通常是客户端每隔一小

段
时间向服务器发送一个数据包，通知服务器自己仍然在线。并传输一些可能必要的数据。使用心跳包的典型协议是 IM，比如 QQ/MSN/飞信等协议
1、在 TCP 的机制里面，本身是存在有心跳包的机制的，也就是 TCP 的选项：SO_KEEPALIVE。系统默认是设置的 2 小时的心跳频率。但是它检查不到机器断电、网线拔出、防火墙这些断
线。 而且逻辑层处理断线可能也不是那么好处理。一般，如果只是用于保活还是可以的。通过使用 TCP 的 KeepAlive 机制（修改那个 time 参数），可以让连接每隔一小段时间就产生一些 ack 包，以降低被踢掉的风险，当然，这样的代价是额外的网络和 CPU 负担。
2、应用层心跳机制实现。
2 2 、e Cookie 与 与 n Session 的作用和原理。
Session 是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中。
Cookie 是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现 Session 的一种方式。
Session ：
由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来
识具体的用户，这个机制就是 Session.典型的场景比如购物车，当你点击下单按钮时，由于 HTTP 协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的 Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个 Session 是保存在服务端的，有一个唯一标识。在服务端保存 Session 的方法很多，内存、数据库、文件都有。集群的时候也要考虑 Session 的转移，在大型的网站，一般会有专门的 Session 服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的。
具体到 Web 中的 Session 指的就是用户在浏览某个网站时，从进入网站到浏览器关闭所经过的这段时间，也就是用户浏览这个网站所花费的时间。因此从上述的定义中我们可以看到，Session 实际上是一个特定的时间概念。
当客户端访问服务器时，服务器根据需求设置 Session，将会话信息保存在服务器上，同时将标示 Session 的 SessionId 传递给客户端浏览器，
浏览器将这个 SessionId 保存在内存中，我们称之为无过期时间的 Cookie。浏览器关闭后，这个 Cookie 就会被清掉，它不会存在于用户的 Cookie 临时文件。
以后浏览器每次请求都会额外加上这个参数值，服务器会根据这个 SessionId，就能取得客户端的数据信息。
如果客户端浏览器意外关闭，服务器保存的 Session 数据不是立即释放，此时数据还会存在，只要我们知道那个 SessionId,就可以继续通过请求获得此 Session 的信息，因为此时后台的 Session 还存在，当然我们可以设置一个 Session 超时时间，一旦超过规定时间没有客户端请求时，服务器就会清除对应 SessionId 的 Session 信息。
Cookie
Cookie 是由服务器端生成，发送给 User-Agent（一般是 web 浏览器），浏览器会将 Cookie 的 key/value 保存到某个目录下的文本文件内，下次请求同一网站时就发送该 Cookie 给服务器（前提是浏览器设置为启用 Cookie）。Cookie 名称和值可以由服务器端开发自己定义，对于 JSP 而言也可以直接写入 Sessionid，这样服务器可以知道该用户是否合法用户以及是否需要重新登录等。
3 、 IP 报文中的内容。

版本：IP 协议的版本，目前的 IP 协议版本号为 4，下一代 IP 协议版本号为 6。
首部长度：IP 报头的长度。固定部分的长度（20 字节）和可变部分的长度之和。共占 4 位。最大为 1111，即 10 进制的 15，代表 IP 报头的最大长度可以为 15 个 32bits（4 字节），也
就是最长可为 15*4=60 字节，除去固定部分的长度 20 字节，可变部分的长度最大为 40 字节。
服务类型：Type Of Service。
总长度：IP 报文的总长度。报头的长度和数据部分的长度之和。
标识：唯一的标识主机发送的每一分数据报。通常每发送一个报文，它的值加一。当 IP 报文长度超过传输网络的 MTU（最大传输单元）时必须分片，这个标识字段的值被复制到所有数据分片的标识字段中，使得这些分片在达到最终目的地时可以依照标识字段的内容重新组成原先的数据。
标志：共 3 位。R、DF、MF 三位。目前只有后两位有效，DF 位：为 1 表示不分片，为 0 表示分片。MF：为 1 表示“更多的片”，为 0 表示这是最后一片。
片位移：本分片在原先数据报文中相对首位的偏移位。（需要再乘以 8）
生存时间：IP 报文所允许通过的路由器的最大数量。每经过一个路由器，TTL 减 1，当为 0 时，路由器将该数据报丢弃。TTL 字段是由发送端初始设置一个 8 bit 字段.推荐的初始值由分配数字 RFC 指定，当前值为 64。发送 ICMP 回显应答时经常把 TTL 设为最大值 255。
协议：指出 IP 报文携带的数据使用的是那种协议，以便目的主机的 IP 层能知道要将数据报
上交到哪个进程（不同的协议有专门不同的进程处理）。和端口号类似，此处采用协议号，TCP 的协议号为 6，UDP 的协议号为 17。ICMP 的协议号为 1，IGMP 的协议号为 2.
首部校验和：计算 IP 头部的校验和，检查 IP 报头的完整性。
源 IP 地址：标识 IP 数据报的源端设备。
目的 IP 地址：标识 IP 数据报的目的地址。
最后就是可变部分和数据部分。

四、常见网络流程机制

1 、 浏览器输入地址到返回结果发生了什么？
总体来说分为以下几个过程:
1、DNS 解析，此外还有 DNSy 优化（DNS 缓存、DNS 负载均衡）
2、TCP 连接
3、发送 HTTP 请求
4、服务器处理请求并返回 HTTP 报文
5、浏览器解析渲染页面
6、连接结束
Web 前端的本质
将信息快速并友好的展示给用户并能够与用户进行交互。
如何尽快的加载资源（网络优化）？
答案就是能不从网络中加载的资源就不从网络中加载，当我们合理使用缓存，将资源放在浏览器端，这是最快的方式。如果资源必须从网络中加载，则要考虑缩短连接时间，即 DNS 优化部分;减少响应内容大小，即对内容进行压缩。另一方面，如果加载的资源数比较少的话，也可以快速的响应用户。

第四章：三方源码高频面试总结

1.Glide ：加载、缓存、LRU 算法（如何自己设计一个大图加载框架）（LRUCache 原理）

如何阅读源码
在开始解析 Glide 源码之前，我想先和大家谈一下该如何阅读源码，这个问题也是我平时被问得比较多的，因为很多人都觉得阅读源码是一件比较困难的事情。

那么阅读源码到底困难吗？这个当然主要还是要视具体的源码而定。比如同样是图片加载框架，我读 Volley 的源码时就感觉酣畅淋漓，并且对 Volley 的架构设计和代码质量深感佩服。读 Glide 的源码时却让我相当痛苦，代码极其难懂。当然这里我并不是说 Glide 的代码写得不好，只是因为 Glide 和复杂程度和 Volley 完全不是在一个量级上的。

那么，虽然源码的复杂程度是外在的不可变条件，但我们却可以通过一些技巧来提升自己阅读源码的能力。这里我和大家分享一下我平时阅读源码时所使用的技巧，简单概括就是八个字：抽丝剥茧、点到即止。应该认准一个功能点，然后去分析这个功能点是如何实现的。但只要去追寻主体的实现逻辑即可，千万不要试图去搞懂每一行代码都是什么意思，那样很容

易会陷入到思维黑洞当中，而且越陷越深。因为这些庞大的系统都不是由一个人写出来的，每一行代码都想搞明白，就会感觉自己是在盲人摸象，永远也研究不透。如果只是去分析主体的实现逻辑，那么就有比较明确的目的性，这样阅读源码会更加轻松，也更加有成效。

而今天带大家阅读的 Glide 源码就非常适合使用这个技巧，因为 Glide 的源码太复杂了，千万不要试图去搞明白它每行代码的作用，而是应该只分析它的主体实现逻辑。那么我们本篇文章就先确立好一个目标，就是要通过阅读源码搞明白下面这行代码：

```
Glide.with(this).load(url).into(imageView);
```

到底是如何实现将一张网络图片展示到 ImageView 上面的。先将 Glide 的一整套图片加载机制的基本流程梳理清楚，然后我们再通过后面的几篇文章具体去了解 Glide 源码方方面面的细节。

准备好了吗？那么我们现在开始。

源码下载

既然是要阅读 Glide 的源码，那么我们自然需要先将 Glide 的源码下载下来。其实如果你是使用在 build.gradle 中添加依赖的方式将 Glide 引入到项目中的，那么源码自动就已经下载下来了，在 Android Studio 中就可以直接进行查看。

不过，使用添加依赖的方式引入的 Glide，我们只能看到它的源码，但不能做任何的修改，如果你还需要修改它的源码的话，可以到 GitHub 上面将它的完整源码下载下来。

Glide 的 GitHub 主页的地址是：https://github.com/bumptech/glide

不过在这个地址下载到的永远都是最新的源码，有可能还正在处于开发当中。而我们整个系列都是使用 Glide 3.7.0 这个版本来进行讲解的，因此如果你需要专门去下载 3.7.0 版本的源码，可以到这个地址进行下载：https://github.com/bumptech/glide/tree/v3.7.0

开始阅读

我们在上一篇文章中已经学习过了，Glide 最基本的用法就是三步走：先 with()，再 load()，最后 into()。那么我们开始一步步阅读这三步走的源码，先从 with()看起。

1. with()

with()方法是 Glide 类中的一组静态方法，它有好几个方法重载，我们来看一下 Glide 类中所有 with()方法的方法重载：

```
public class Glide {

    ...

    public static RequestManager with(Context context) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
```

```
        return retriever.get(context);
    }

    public static RequestManager with(Activity activity) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(activity);
    }

    public static RequestManager with(FragmentActivity activity) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(activity);
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    public static RequestManager with(android.app.Fragment fragment) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(fragment);
    }

    public static RequestManager with(Fragment fragment) {
        RequestManagerRetriever retriever = RequestManagerRetriever.get();
        return retriever.get(fragment);
    }
}
```

可以看到，with()方法的重载种类非常多，既可以传入 Activity，也可以传入 Fragment 或者是 Context。每一个 with()方法重载的代码都非常简单，都是先调用 RequestManagerRetriever 的静态 get()方法得到一个 RequestManagerRetriever 对象，这个静态 get()方法就是一个单例实现，没什么需要解释的。然后再调用 RequestManagerRetriever 的实例 get()方法，去获取 RequestManager 对象。

而 RequestManagerRetriever 的实例 get()方法中的逻辑是什么样的呢？我们一起来看一看：

```
public class RequestManagerRetriever implements Handler.Callback {

    private static final RequestManagerRetriever INSTANCE = new RequestManagerRetriever();

    private volatile RequestManager applicationManager;

    ...

    /**
     * Retrieves and returns the RequestManagerRetriever singleton.
     */
    public static RequestManagerRetriever get() {
```

```
        return INSTANCE;
    }

    private RequestManager getApplicationManager(Context context) {
        // Either an application context or we're on a background thread.
        if (applicationManager == null) {
            synchronized (this) {
                if (applicationManager == null) {
                    // Normally pause/resume is taken care of by the fragment we add to the fragment or activity.
                    // However, in this case since the manager attached to the application will not receive lifecycle
                    // events, we must force the manager to start resumed using ApplicationLifecycle.
                    applicationManager = new RequestManager(context.getApplicationContext(),
                            new ApplicationLifecycle(), new EmptyRequestManagerTreeNode());
                }
            }
        }
        return applicationManager;
    }

    public RequestManager get(Context context) {
        if (context == null) {
            throw new IllegalArgumentException("You cannot start a load on a null Context");
        } else if (Util.isOnMainThread() && !(context instanceof Application)) {
            if (context instanceof FragmentActivity) {
                return get((FragmentActivity) context);
            } else if (context instanceof Activity) {
                return get((Activity) context);
            } else if (context instanceof ContextWrapper) {
                return get(((ContextWrapper) context).getBaseContext());
            }
        }
        return getApplicationManager(context);
    }

    public RequestManager get(FragmentActivity activity) {
        if (Util.isOnBackgroundThread()) {
            return get(activity.getApplicationContext());
        } else {
            assertNotDestroyed(activity);
```

```
                FragmentManager fm = activity.getSupportFragmentManager();
                return supportFragmentGet(activity, fm);
            }
        }

        public RequestManager get(Fragment fragment) {
            if (fragment.getActivity() == null) {
                throw new IllegalArgumentException("You cannot start a load on a fragment before it is attached");
            }
            if (Util.isOnBackgroundThread()) {
                return get(fragment.getActivity().getApplicationContext());
            } else {
                FragmentManager fm = fragment.getChildFragmentManager();
                return supportFragmentGet(fragment.getActivity(), fm);
            }
        }

        @TargetApi(Build.VERSION_CODES.HONEYCOMB)
        public RequestManager get(Activity activity) {
            if (Util.isOnBackgroundThread() || Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
                return get(activity.getApplicationContext());
            } else {
                assertNotDestroyed(activity);
                android.app.FragmentManager fm = activity.getFragmentManager();
                return fragmentGet(activity, fm);
            }
        }

        @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
        private static void assertNotDestroyed(Activity activity) {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1 && activity.isDestroyed()) {
                throw new IllegalArgumentException("You cannot start a load for a destroyed activity");
            }
        }

        @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
        public RequestManager get(android.app.Fragment fragment) {
            if (fragment.getActivity() == null) {
                throw new IllegalArgumentException("You cannot start a load on a fragment before it is attached");
```

```
        }
        if (Util.isOnBackgroundThread() || Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR1) {
            return get(fragment.getActivity().getApplicationContext());
        } else {
            android.app.FragmentManager fm = fragment.getChildFragmentManager();
            return fragmentGet(fragment.getActivity(), fm);
        }
    }

    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
    RequestManagerFragment getRequestManagerFragment(final android.app.FragmentManager fm) {
        RequestManagerFragment current = (RequestManagerFragment) fm.findFragmentByTag(FRAGMENT_TAG);
        if (current == null) {
            current = pendingRequestManagerFragments.get(fm);
            if (current == null) {
                current = new RequestManagerFragment();
                pendingRequestManagerFragments.put(fm, current);
                fm.beginTransaction().add(current, FRAGMENT_TAG).commitAllowingStateLoss();
                handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER, fm).sendToTarget();
            }
        }
        return current;
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    RequestManager fragmentGet(Context context, android.app.FragmentManager fm) {
        RequestManagerFragment current = getRequestManagerFragment(fm);
        RequestManager requestManager = current.getRequestManager();
        if (requestManager == null) {
            requestManager = new RequestManager(context, current.getLifecycle(), current.getRequestManagerTreeNode());
            current.setRequestManager(requestManager);
        }
        return requestManager;
    }

    SupportRequestManagerFragment getSupportRequestManagerFragment(final FragmentManager fm) {
        SupportRequestManagerFragment current = (SupportRequestManagerFragment)
```

```
fm.findFragmentByTag(FRAGMENT_TAG);
            if (current == null) {
                current = pendingSupportRequestManagerFragments.get(fm);
                if (current == null) {
                    current = new SupportRequestManagerFragment();
                    pendingSupportRequestManagerFragments.put(fm, current);
                    fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
                    handler.obtainMessage(ID_REMOVE_SUPPORT_FRAGMENT_MANAGER,
fm).sendToTarget();
                }
            }
            return current;
        }

        RequestManager supportFragmentGet(Context context, FragmentManager fm) {
            SupportRequestManagerFragment current =
getSupportRequestManagerFragment(fm);
            RequestManager requestManager = current.getRequestManager();
            if (requestManager == null) {
                requestManager = new RequestManager(context, current.getLifecycle(),
current.getRequestManagerTreeNode());
                current.setRequestManager(requestManager);
            }
            return requestManager;
        }

        ...
}
```

上述代码虽然看上去逻辑有点复杂，但是将它们梳理清楚后还是很简单的。RequestManagerRetriever 类中看似有很多个 get()方法的重载，什么 Context 参数，Activity 参数，Fragment 参数等等，实际上只有两种情况而已，即传入 Application 类型的参数，和传入非 Application 类型的参数。

我们先来看传入 Application 参数的情况。如果在 Glide.with()方法中传入的是一个 Application 对象，那么这里就会调用带有 Context 参数的 get()方法重载，然后会在第 44 行调用 getApplicationManager()方法来获取一个 RequestManager 对象。其实这是最简单的一种情况，因为 Application 对象的生命周期即应用程序的生命周期，因此 Glide 并不需要做什么特殊的处理，它自动就是和应用程序的生命周期是同步的，如果应用程序关闭的话，Glide 的加载也会同时终止。

接下来我们看传入非 Application 参数的情况。不管你在 Glide.with()方法中传入的是 Activity、FragmentActivity、v4 包下的 Fragment、还是 app 包下的 Fragment，最终的流程都是一样的，那就是会向当前的 Activity 当中添加一个隐藏的 Fragment。具体添加的逻辑是在上述代码的

第 117 行和第 141 行，分别对应的 app 包和 v4 包下的两种 Fragment 的情况。那么这里为什么要添加一个隐藏的 Fragment 呢？因为 Glide 需要知道加载的生命周期。很简单的一个道理，如果你在某个 Activity 上正在加载着一张图片，结果图片还没加载出来，Activity 就被用户关掉了，那么图片还应该继续加载吗？当然不应该。可是 Glide 并没有办法知道 Activity 的生命周期，于是 Glide 就使用了添加隐藏 Fragment 的这种小技巧，因为 Fragment 的生命周期和 Activity 是同步的，如果 Activity 被销毁了，Fragment 是可以监听到的，这样 Glide 就可以捕获这个事件并停止图片加载了。

这里额外再提一句，从第 48 行代码可以看出，如果我们是在非主线程当中使用的 Glide，那么不管你是传入的 Activity 还是 Fragment，都会被强制当成 Application 来处理。不过其实这就属于是在分析代码的细节了，本篇文章我们将会把目光主要放在 Glide 的主线工作流程上面，后面不会过多去分析这些细节方面的内容。

总体来说，第一个 with()方法的源码还是比较好理解的。其实就是为了得到一个 RequestManager 对象而已，然后 Glide 会根据我们传入 with()方法的参数来确定图片加载的生命周期，并没有什么特别复杂的逻辑。不过复杂的逻辑还在后面等着我们呢，接下来我们开始分析第二步，load()方法。

2. load()

由于 with()方法返回的是一个 RequestManager 对象，那么很容易就能想到，load()方法是在 RequestManager 类当中的，所以说我们首先要看的就是 RequestManager 这个类。不过在上一篇文章中我们学过，Glide 是支持图片 URL 字符串、图片本地路径等等加载形式的，因此 RequestManager 中也有很多个 load()方法的重载。但是这里我们不可能把每个 load()方法的重载都看一遍，因此我们就只选其中一个加载图片 URL 字符串的 load()方法来进行研究吧。

RequestManager 类的简化代码如下所示：

```
public class RequestManager implements LifecycleListener {

    ...

    /**
     * Returns a request builder to load the given {@link String}.
     * signature.
     *
     * @see #fromString()
     * @see #load(Object)
     *
     * @param string A file path, or a uri or url handled by {@link com.bumptech.glide.load.model.UriLoader}.
     */
    public DrawableTypeRequest<String> load(String string) {
        return (DrawableTypeRequest<String>) fromString().load(string);
    }
```

```
    /**
     * Returns a request builder that loads data from {@link String}s using an empty signature.
     *
     * <p>
     *      Note - this method caches data using only the given String as the cache key. If the
data is a Uri outside of
     *      your control, or you otherwise expect the data represented by the given String to
change without the String
     *      identifier changing, Consider using
     *      {@link GenericRequestBuilder#signature(Key)} to mixin a signature
     *      you create that identifies the data currently at the given String that will invalidate
the cache if that data
     *      changes. Alternatively, using {@link DiskCacheStrategy#NONE} and/or
     *      {@link DrawableRequestBuilder#skipMemoryCache(boolean)} may be appropriate.
     * </p>
     *
     * @see #from(Class)
     * @see #load(String)
     */
    public DrawableTypeRequest<String> fromString() {
        return loadGeneric(String.class);
    }

    private <T> DrawableTypeRequest<T> loadGeneric(Class<T> modelClass) {
        ModelLoader<T, InputStream> streamModelLoader =
Glide.buildStreamModelLoader(modelClass, context);
        ModelLoader<T, ParcelFileDescriptor> fileDescriptorModelLoader =
                Glide.buildFileDescriptorModelLoader(modelClass, context);
        if (modelClass != null && streamModelLoader == null && fileDescriptorModelLoader ==
null) {
            throw new IllegalArgumentException("Unknown type " + modelClass + ". You must
provide a Model of a type for"
                    + " which there is a registered ModelLoader, if you are using a custom
model, you must first call"
                    + " Glide#register with a ModelLoaderFactory for your custom model
class");
        }
        return optionsApplier.apply(
                new DrawableTypeRequest<T>(modelClass, streamModelLoader,
fileDescriptorModelLoader, context,
                        glide, requestTracker, lifecycle, optionsApplier));
    }
```

...

}

RequestManager 类的代码是非常多的，但是经过我这样简化之后，看上去就比较清爽了。在我们只探究加载图片 URL 字符串这一个 load()方法的情况下，那么比较重要的方法就只剩下上述代码中的这三个方法。

那么我们先来看 load()方法，这个方法中的逻辑是非常简单的，只有一行代码，就是先调用了 fromString()方法，再调用 load()方法，然后把传入的图片 URL 地址传进去。而 fromString()方法也极为简单，就是调用了 loadGeneric()方法，并且指定参数为 String.class，因为 load()方法传入的是一个字符串参数。那么看上去，好像主要的工作都是在 loadGeneric()方法中进行的了。

其实 loadGeneric()方法也没几行代码，这里分别调用了 Glide.buildStreamModelLoader()方法和 Glide.buildFileDescriptorModelLoader()方法来获得 ModelLoader 对象。ModelLoader 对象是用于加载图片的，而我们给 load()方法传入不同类型的参数，这里也会得到不同的 ModelLoader 对象。不过 buildStreamModelLoader()方法内部的逻辑还是蛮复杂的，这里就不展开介绍了，要不然篇幅实在收不住，感兴趣的话你可以自己研究。由于我们刚才传入的参数是 String.class，因此最终得到的是 StreamStringLoader 对象，它是实现了 ModelLoader 接口的。

最后我们可以看到，loadGeneric()方法是要返回一个 DrawableTypeRequest 对象的，因此在 loadGeneric()方法的最后又去 new 了一个 DrawableTypeRequest 对象，然后把刚才获得的 ModelLoader 对象，还有一大堆杂七杂八的东西都传了进去。具体每个参数的含义和作用就不解释了，我们只看主线流程。

那么这个 DrawableTypeRequest 的作用是什么呢？我们来看下它的源码，如下所示：

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuilder<ModelType>
implements DownloadOptions {
    private final ModelLoader<ModelType, InputStream> streamModelLoader;
    private final ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A, ImageVideoWrapper, Z, R> buildProvider(Glide
glide,
            ModelLoader<A, InputStream> streamModelLoader,
            ModelLoader<A, ParcelFileDescriptor> fileDescriptorModelLoader, Class<Z>
resourceClass,
            Class<R> transcodedClass,
            ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null && fileDescriptorModelLoader == null) {
            return null;
        }
```

```
        if (transcoder == null) {
            transcoder = glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider = glide.buildDataProvider(ImageVideoWrapper.class,
                resourceClass);
        ImageVideoModelLoader<A> modelLoader = new ImageVideoModelLoader<A>(streamModelLoader,
                fileDescriptorModelLoader);
        return new FixedLoadProvider<A, ImageVideoWrapper, Z, R>(modelLoader, transcoder, dataLoadProvider);
    }

    DrawableTypeRequest(Class<ModelType> modelClass, ModelLoader<ModelType, InputStream> streamModelLoader,
            ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader, Context context, Glide glide,
            RequestTracker requestTracker, Lifecycle lifecycle, RequestManager.OptionsApplier optionsApplier) {
        super(context, modelClass,
                buildProvider(glide, streamModelLoader, fileDescriptorModelLoader, GifBitmapWrapper.class,
                        GlideDrawable.class, null),
                glide, requestTracker, lifecycle);
        this.streamModelLoader = streamModelLoader;
        this.fileDescriptorModelLoader = fileDescriptorModelLoader;
        this.optionsApplier = optionsApplier;
    }

    /**
     * Attempts to always load the resource as a {@link android.graphics.Bitmap}, even if it could actually be animated.
     *
     * @return A new request builder for loading a {@link android.graphics.Bitmap}
     */
    public BitmapTypeRequest<ModelType> asBitmap() {
        return optionsApplier.apply(new BitmapTypeRequest<ModelType>(this, streamModelLoader,
                fileDescriptorModelLoader, optionsApplier));
    }

    /**
     * Attempts to always load the resource as a {@link
```

```
com.bumptech.glide.load.resource.gif.GifDrawable}.
        * <p>
        *      If the underlying data is not a GIF, this will fail. As a result, this should only be used
if the model
        *      represents an animated GIF and the caller wants to interact with the GIfDrawable
directly. Normally using
        *       just an {@link DrawableTypeRequest} is sufficient because it will determine
whether or
        *      not the given data represents an animated GIF and return the appropriate animated
or not animated
        *      {@link android.graphics.drawable.Drawable} automatically.
        * </p>
        *
        *      @return A new request builder for loading a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
        */
       public GifTypeRequest<ModelType> asGif() {
           return optionsApplier.apply(new GifTypeRequest<ModelType>(this,
streamModelLoader, optionsApplier));
       }

       ...
}
```

这个类中的代码本身就不多，我只是稍微做了一点简化。可以看到，最主要的就是它提供了 asBitmap()和 asGif()这两个方法。这两个方法我们在上一篇文章当中都是学过的，分别是用于强制指定加载静态图片和动态图片。而从源码中可以看出，它们分别又创建了一个 BitmapTypeRequest 和 GifTypeRequest，如果没有进行强制指定的话，那默认就是使用 DrawableTypeRequest。

好的，那么我们再回到 RequestManager 的 load()方法中。刚才已经分析过了，fromString()方法会返回一个 DrawableTypeRequest 对象，接下来会调用这个对象的 load()方法，把图片的 URL 地址传进去。但是我们刚才看到了，DrawableTypeRequest 中并没有 load()方法，那么很容易就能猜想到，load()方法是在父类当中的。

DrawableTypeRequest 的父类是 DrawableRequestBuilder，我们来看下这个类的源码：

```
public class DrawableRequestBuilder<ModelType>
        extends GenericRequestBuilder<ModelType, ImageVideoWrapper, GifBitmapWrapper,
GlideDrawable>
        implements BitmapOptions, DrawableOptions {

    DrawableRequestBuilder(Context context, Class<ModelType> modelClass,
            LoadProvider<ModelType, ImageVideoWrapper, GifBitmapWrapper,
GlideDrawable> loadProvider, Glide glide,
```

```
                RequestTracker requestTracker, Lifecycle lifecycle) {
        super(context, modelClass, loadProvider, GlideDrawable.class, glide, requestTracker,
lifecycle);
        // Default to animating.
        crossFade();
    }

    public DrawableRequestBuilder<ModelType> thumbnail(
            DrawableRequestBuilder<?> thumbnailRequest) {
        super.thumbnail(thumbnailRequest);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> thumbnail(
            GenericRequestBuilder<?, ?, ?, GlideDrawable> thumbnailRequest) {
        super.thumbnail(thumbnailRequest);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> thumbnail(float sizeMultiplier) {
        super.thumbnail(sizeMultiplier);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> sizeMultiplier(float sizeMultiplier) {
        super.sizeMultiplier(sizeMultiplier);
        return this;
    }

    @Override
    public                                          DrawableRequestBuilder<ModelType>
decoder(ResourceDecoder<ImageVideoWrapper, GifBitmapWrapper> decoder) {
        super.decoder(decoder);
        return this;
    }

    @Override
    public      DrawableRequestBuilder<ModelType>      cacheDecoder(ResourceDecoder<File,
GifBitmapWrapper> cacheDecoder) {
        super.cacheDecoder(cacheDecoder);
        return this;
```

```
    }

    @Override
    public DrawableRequestBuilder<ModelType> encoder(ResourceEncoder<GifBitmapWrapper>
encoder) {
        super.encoder(encoder);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> priority(Priority priority) {
        super.priority(priority);
        return this;
    }

    public DrawableRequestBuilder<ModelType> transform(BitmapTransformation...
transformations) {
        return bitmapTransform(transformations);
    }

    public DrawableRequestBuilder<ModelType> centerCrop() {
        return transform(glide.getDrawableCenterCrop());
    }

    public DrawableRequestBuilder<ModelType> fitCenter() {
        return transform(glide.getDrawableFitCenter());
    }

    public DrawableRequestBuilder<ModelType> bitmapTransform(Transformation<Bitmap>...
bitmapTransformations) {
        GifBitmapWrapperTransformation[] transformations =
                new GifBitmapWrapperTransformation[bitmapTransformations.length];
        for (int i = 0; i < bitmapTransformations.length; i++) {
            transformations[i] = new GifBitmapWrapperTransformation(glide.getBitmapPool(),
bitmapTransformations[i]);
        }
        return transform(transformations);
    }

    @Override
    public DrawableRequestBuilder<ModelType>
transform(Transformation<GifBitmapWrapper>... transformation) {
        super.transform(transformation);
        return this;
```

```
    }

    @Override
    public DrawableRequestBuilder<ModelType> transcoder(
            ResourceTranscoder<GifBitmapWrapper, GlideDrawable> transcoder) {
        super.transcoder(transcoder);
        return this;
    }

    public final DrawableRequestBuilder<ModelType> crossFade() {
        super.animate(new DrawableCrossFadeFactory<GlideDrawable>());
        return this;
    }

    public DrawableRequestBuilder<ModelType> crossFade(int duration) {
        super.animate(new DrawableCrossFadeFactory<GlideDrawable>(duration));
        return this;
    }

    public DrawableRequestBuilder<ModelType> crossFade(int animationId, int duration) {
        super.animate(new DrawableCrossFadeFactory<GlideDrawable>(context, animationId,
                duration));
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> dontAnimate() {
        super.dontAnimate();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> animate(ViewPropertyAnimation.Animator
animator) {
        super.animate(animator);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> animate(int animationId) {
        super.animate(animationId);
        return this;
    }
```

```
@Override
public DrawableRequestBuilder<ModelType> placeholder(int resourceId) {
        super.placeholder(resourceId);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> placeholder(Drawable drawable) {
        super.placeholder(drawable);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> fallback(Drawable drawable) {
        super.fallback(drawable);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> fallback(int resourceId) {
        super.fallback(resourceId);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> error(int resourceId) {
        super.error(resourceId);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> error(Drawable drawable) {
        super.error(drawable);
        return this;
}

@Override
public DrawableRequestBuilder<ModelType> listener(
                RequestListener<? super ModelType, GlideDrawable> requestListener) {
        super.listener(requestListener);
        return this;
}
@Override
public DrawableRequestBuilder<ModelType> diskCacheStrategy(DiskCacheStrategy strategy)
```

```
{
        super.diskCacheStrategy(strategy);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> skipMemoryCache(boolean skip) {
        super.skipMemoryCache(skip);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> override(int width, int height) {
        super.override(width, height);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> sourceEncoder(Encoder<ImageVideoWrapper>
sourceEncoder) {
        super.sourceEncoder(sourceEncoder);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> dontTransform() {
        super.dontTransform();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> signature(Key signature) {
        super.signature(signature);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> load(ModelType model) {
        super.load(model);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> clone() {
```

```
        return (DrawableRequestBuilder<ModelType>) super.clone();
    }

    @Override
    public Target<GlideDrawable> into(ImageView view) {
        return super.into(view);
    }

    @Override
    void applyFitCenter() {
        fitCenter();
    }

    @Override
    void applyCenterCrop() {
        centerCrop();
    }
}
```

DrawableRequestBuilder 中有很多个方法，这些方法其实就是 Glide 绝大多数的 API 了。里面有不少我们在上篇文章中已经用过了，比如说 placeholder()方法、error()方法、diskCacheStrategy()方法、override()方法等。当然还有很多暂时还没用到的 API，我们会在后面的文章当中学习。

到这里，第二步 load()方法也就分析结束了。为什么呢？因为你会发现 DrawableRequestBuilder 类中有一个 into()方法（上述代码第 220 行），也就是说，最终 load()方法返回的其实就是一个 DrawableTypeRequest 对象。那么接下来我们就要进行第三步了，分析 into()方法中的逻辑。

3. into()

如果说前面两步都是在准备开胃小菜的话，那么现在终于要进入主菜了，因为 into()方法也是整个 Glide 图片加载流程中逻辑最复杂的地方。

不过从刚才的代码来看，into()方法中并没有任何逻辑，只有一句 super.into(view)。那么很显然，into()方法的具体逻辑都是在 DrawableRequestBuilder 的父类当中了。

DrawableRequestBuilder 的父类是 GenericRequestBuilder，我们来看一下 GenericRequestBuilder 类中的 into()方法，如下所示：

```
public Target<TranscodeType> into(ImageView view) {
    Util.assertMainThread();
    if (view == null) {
        throw new IllegalArgumentException("You must pass in a non null View");
    }
    if (!isTransformationSet && view.getScaleType() != null) {
```

```
            switch (view.getScaleType()) {
                case CENTER_CROP:
                    applyCenterCrop();
                    break;
                case FIT_CENTER:
                case FIT_START:
                case FIT_END:
                    applyFitCenter();
                    break;
                //$CASES-OMITTED$
                default:
                    // Do nothing.
            }
        }
        return into(glide.buildImageViewTarget(view, transcodeClass));
    }
```

这里前面一大堆的判断逻辑我们都可以先不用管，等到后面文章讲 transform 的时候会再进行解释，现在我们只需要关注最后一行代码。最后一行代码先是调用了 glide.buildImageViewTarget()方法，这个方法会构建出一个 Target 对象，Target 对象则是用来最终展示图片用的，如果我们跟进去的话会看到如下代码：

```
<R> Target<R> buildImageViewTarget(ImageView imageView, Class<R> transcodedClass) {
        return imageViewTargetFactory.buildTarget(imageView, transcodedClass);
    }
```

这里其实又是调用了 ImageViewTargetFactory 的 buildTarget()方法，我们继续跟进去，代码如下所示：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view, Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new BitmapImageViewTarget(view);
        } else if (Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new DrawableImageViewTarget(view);
        } else {
            throw new IllegalArgumentException("Unhandled class: " + clazz
                    + ", try .as*(Class).transcode(ResourceTranscoder)");
        }
    }
}
```

可以看到，在 buildTarget()方法中会根据传入的 class 参数来构建不同的 Target 对象。那如果

你要分析这个 class 参数是从哪儿传过来的，这可有得你分析了，简单起见我直接帮大家梳理清楚。这个 class 参数其实基本上只有两种情况，如果你在使用 Glide 加载图片的时候调用了 asBitmap()方法，那么这里就会构建出 BitmapImageViewTarget 对象，否则的话构建的都是 GlideDrawableImageViewTarget 对象。至于上述代码中的 DrawableImageViewTarget 对象，这个通常都是用不到的，我们可以暂时不用管它。

也就是说，通过 glide.buildImageViewTarget()方法，我们构建出了一个 GlideDrawableImageViewTarget 对象。那现在回到刚才 into()方法的最后一行，可以看到，这里又将这个参数传入到了 GenericRequestBuilder 另一个接收 Target 对象的 into()方法当中了。我们来看一下这个 into()方法的源码：

```
public <Y extends Target<TranscodeType>> Y into(Y target) {
    Util.assertMainThread();
    if (target == null) {
        throw new IllegalArgumentException("You must pass in a non null Target");
    }
    if (!isModelSet) {
        throw new IllegalArgumentException("You must first set a model (try #load())");
    }
    Request previous = target.getRequest();
    if (previous != null) {
        previous.clear();
        requestTracker.removeRequest(previous);
        previous.recycle();
    }
    Request request = buildRequest(target);
    target.setRequest(request);
    lifecycle.addListener(target);
    requestTracker.runRequest(request);
    return target;
}
```

这里我们还是只抓核心代码，其实只有两行是最关键的，第 15 行调用 buildRequest()方法构建出了一个 Request 对象，还有第 18 行来执行这个 Request。

Request 是用来发出加载图片请求的，它是 Glide 中非常关键的一个组件。我们先来看 buildRequest()方法是如何构建 Request 对象的：

```
private Request buildRequest(Target<TranscodeType> target) {
    if (priority == null) {
        priority = Priority.NORMAL;
    }
    return buildRequestRecursive(target, null);
}
```

```
private Request buildRequestRecursive(Target<TranscodeType> target, ThumbnailRequestCoordinator parentCoordinator) {
    if (thumbnailRequestBuilder != null) {
        if (isThumbnailBuilt) {
            throw new IllegalStateException("You cannot use a request as both the main request and a thumbnail, "
                    + "consider using clone() on the request(s) passed to thumbnail()");
        }
        // Recursive case: contains a potentially recursive thumbnail request builder.
        if (thumbnailRequestBuilder.animationFactory.equals(NoAnimation.getFactory())) {
            thumbnailRequestBuilder.animationFactory = animationFactory;
        }

        if (thumbnailRequestBuilder.priority == null) {
            thumbnailRequestBuilder.priority = getThumbnailPriority();
        }

        if (Util.isValidDimensions(overrideWidth, overrideHeight)
                && !Util.isValidDimensions(thumbnailRequestBuilder.overrideWidth,
                        thumbnailRequestBuilder.overrideHeight)) {
           thumbnailRequestBuilder.override(overrideWidth, overrideHeight);
        }

        ThumbnailRequestCoordinator coordinator = new ThumbnailRequestCoordinator(parentCoordinator);
        Request fullRequest = obtainRequest(target, sizeMultiplier, priority, coordinator);
        // Guard against infinite recursion.
        isThumbnailBuilt = true;
        // Recursively generate thumbnail requests.
        Request thumbRequest = thumbnailRequestBuilder.buildRequestRecursive(target, coordinator);
        isThumbnailBuilt = false;
        coordinator.setRequests(fullRequest, thumbRequest);
        return coordinator;
    } else if (thumbSizeMultiplier != null) {
        // Base case: thumbnail multiplier generates a thumbnail request, but cannot recurse.
        ThumbnailRequestCoordinator coordinator = new ThumbnailRequestCoordinator(parentCoordinator);
        Request fullRequest = obtainRequest(target, sizeMultiplier, priority, coordinator);
        Request thumbnailRequest = obtainRequest(target, thumbSizeMultiplier, getThumbnailPriority(), coordinator);
        coordinator.setRequests(fullRequest, thumbnailRequest);
        return coordinator;
    } else {
```

```
            // Base case: no thumbnail.
            return obtainRequest(target, sizeMultiplier, priority, parentCoordinator);
        }
    }

    private Request obtainRequest(Target<TranscodeType> target, float sizeMultiplier, Priority priority,
            RequestCoordinator requestCoordinator) {
        return GenericRequest.obtain(
                loadProvider,
                model,
                signature,
                context,
                priority,
                target,
                sizeMultiplier,
                placeholderDrawable,
                placeholderId,
                errorPlaceholder,
                errorId,
                fallbackDrawable,
                fallbackResource,
                requestListener,
                requestCoordinator,
                glide.getEngine(),
                transformation,
                transcodeClass,
                isCacheable,
                animationFactory,
                overrideWidth,
                overrideHeight,
                diskCacheStrategy);
    }
```

可以看到，buildRequest()方法的内部其实又调用了 buildRequestRecursive()方法，而 buildRequestRecursive()方法中的代码虽然有点长，但是其中 90%的代码都是在处理缩略图的。如果我们只追主线流程的话，那么只需要看第 47 行代码就可以了。这里调用了 obtainRequest()方法来获取一个 Request 对象，而 obtainRequest()方法中又去调用了 GenericRequest 的 obtain()方法。注意这个 obtain()方法需要传入非常多的参数，而其中很多的参数我们都是比较熟悉的，像什么 placeholderId、errorPlaceholder、diskCacheStrategy 等等。因此，我们就有理由猜测，刚才在 load()方法中调用的所有 API，其实都是在这里组装到 Request 对象当中的。那么我们进入到这个 GenericRequest 的 obtain()方法瞧一瞧：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback,
        ResourceCallback {
```

```
    ...

    public static <A, T, Z, R> GenericRequest<A, T, Z, R> obtain(
            LoadProvider<A, T, Z, R> loadProvider,
            A model,
            Key signature,
            Context context,
            Priority priority,
            Target<R> target,
            float sizeMultiplier,
            Drawable placeholderDrawable,
            int placeholderResourceId,
            Drawable errorDrawable,
            int errorResourceId,
            Drawable fallbackDrawable,
            int fallbackResourceId,
            RequestListener<? super A, R> requestListener,
            RequestCoordinator requestCoordinator,
            Engine engine,
            Transformation<Z> transformation,
            Class<R> transcodeClass,
            boolean isMemoryCacheable,
            GlideAnimationFactory<R> animationFactory,
            int overrideWidth,
            int overrideHeight,
            DiskCacheStrategy diskCacheStrategy) {
        @SuppressWarnings("unchecked")
        GenericRequest<A, T, Z, R> request = (GenericRequest<A, T, Z, R>)
REQUEST_POOL.poll();
        if (request == null) {
            request = new GenericRequest<A, T, Z, R>();
        }
        request.init(loadProvider,
                model,
                signature,
                context,
                priority,
                target,
                sizeMultiplier,
                placeholderDrawable,
                placeholderResourceId,
                errorDrawable,
                errorResourceId,
```

```
                fallbackDrawable,
                fallbackResourceId,
                requestListener,
                requestCoordinator,
                engine,
                transformation,
                transcodeClass,
                isMemoryCacheable,
                animationFactory,
                overrideWidth,
                overrideHeight,
                diskCacheStrategy);
        return request;
    }

    ...
}
```

可以看到，这里在第 33 行去 new 了一个 GenericRequest 对象，并在最后一行返回，也就是说，obtain()方法实际上获得的就是一个 GenericRequest 对象。另外这里又在第 35 行调用了 GenericRequest 的 init()，里面主要就是一些赋值的代码，将传入的这些参数赋值到 GenericRequest 的成员变量当中，我们就不再跟进去看了。

好，那现在解决了构建 Request 对象的问题，接下来我们看一下这个 Request 对象又是怎么执行的。回到刚才的 into()方法，你会发现在第 18 行调用了 requestTracker.runRequest()方法来去执行这个 Request，那么我们跟进去瞧一瞧，如下所示：

```
/**
 * Starts tracking the given request.
 */
public void runRequest(Request request) {
    requests.add(request);
    if (!isPaused) {
        request.begin();
    } else {
        pendingRequests.add(request);
    }
}
```

这里有一个简单的逻辑判断，就是先判断 Glide 当前是不是处理暂停状态，如果不是暂停状态就调用 Request 的 begin()方法来执行 Request，否则的话就先将 Request 添加到待执行队列里面，等暂停状态解除了之后再执行。

暂停请求的功能仍然不是这篇文章所关心的，这里就直接忽略了，我们重点来看这个 begin()方法。由于当前的 Request 对象是一个 GenericRequest，因此这里就需要看 GenericRequest 中的 begin()方法了，如下所示：

```
@Override
public void begin() {
    startTime = LogTime.getLogTime();
    if (model == null) {
        onException(null);
        return;
    }
    status = Status.WAITING_FOR_SIZE;
    if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
        onSizeReady(overrideWidth, overrideHeight);
    } else {
        target.getSize(this);
    }
    if (!isComplete() && !isFailed() && canNotifyStatusChanged()) {
        target.onLoadStarted(getPlaceholderDrawable());
    }
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished run method in " + LogTime.getElapsedMillis(startTime));
    }
}
```

这里我们来注意几个细节，首先如果 model 等于 null，model 也就是我们在第二步 load()方法中传入的图片 URL 地址，这个时候会调用 onException()方法。如果你跟到 onException()方法里面去看看，你会发现它最终会调用到一个 setErrorPlaceholder()当中，如下所示：

```
private void setErrorPlaceholder(Exception e) {
    if (!canNotifyStatusChanged()) {
        return;
    }
    Drawable error = model == null ? getFallbackDrawable() : null;
    if (error == null) {
       error = getErrorDrawable();
    }
    if (error == null) {
        error = getPlaceholderDrawable();
    }
    target.onLoadFailed(e, error);
}
```

这个方法中会先去获取一个 error 的占位图，如果获取不到的话会再去获取一个 loading 占位图，然后调用 target.onLoadFailed()方法并将占位图传入。那么 onLoadFailed()方法中做了什么呢？我们看一下：

```
public abstract class ImageViewTarget<Z> extends ViewTarget<ImageView, Z> implements GlideAnimation.ViewAdapter {
```

```
    ...

    @Override
    public void onLoadStarted(Drawable placeholder) {
        view.setImageDrawable(placeholder);
    }

    @Override
    public void onLoadFailed(Exception e, Drawable errorDrawable) {
        view.setImageDrawable(errorDrawable);
    }

    ...
}
```

很简单，其实就是将这张 error 占位图显示到 ImageView 上而已，因为现在出现了异常，没办法展示正常的图片了。而如果你仔细看下刚才 begin()方法的第 15 行，你会发现它又调用了一个 target.onLoadStarted()方法，并传入了一个 loading 占位图，在也就说，在图片请求开始之前，会先使用这张占位图代替最终的图片显示。这也是我们在上一篇文章中学过的 placeholder()和 error()这两个占位图 API 底层的实现原理。

好，那么我们继续回到 begin()方法。刚才讲了占位图的实现，那么具体的图片加载又是从哪里开始的呢？是在 begin()方法的第 10 行和第 12 行。这里要分两种情况，一种是你使用了 override() API 为图片指定了一个固定的宽高，一种是没有指定。如果指定了的话，就会执行第 10 行代码，调用 onSizeReady()方法。如果没指定的话，就会执行第 12 行代码，调用 target.getSize()方法。这个 target.getSize()方法的内部会根据 ImageView 的 layout_width 和 layout_height 值做一系列的计算，来算出图片应该的宽高。具体的计算细节我就不带着大家分析了，总之在计算完之后，它也会调用 onSizeReady()方法。也就是说，不管是哪种情况，最终都会调用到 onSizeReady()方法，在这里进行下一步操作。那么我们跟到这个方法里面来：

```
@Override
public void onSizeReady(int width, int height) {
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));
    }
    if (status != Status.WAITING_FOR_SIZE) {
        return;
    }
    status = Status.RUNNING;
    width = Math.round(sizeMultiplier * width);
    height = Math.round(sizeMultiplier * height);
    ModelLoader<A, T> modelLoader = loadProvider.getModelLoader();
    final DataFetcher<T> dataFetcher = modelLoader.getResourceFetcher(model, width, height);
    if (dataFetcher == null) {
```

```
            onException(new Exception("Failed to load model: \'" + model + "\'"));
            return;
        }
        ResourceTranscoder<Z, R> transcoder = loadProvider.getTranscoder();
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTime));
        }
        loadedFromMemoryCache = true;
        loadStatus = engine.load(signature, width, height, dataFetcher, loadProvider, transformation,
transcoder,
                priority, isMemoryCacheable, diskCacheStrategy, this);
        loadedFromMemoryCache = resource != null;
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));
        }
}
```

从这里开始，真正复杂的地方来了，我们需要慢慢进行分析。先来看一下，在第 12 行调用了 loadProvider.getModelLoader()方法，那么我们第一个要搞清楚的就是，这个 loadProvider 是什么？要搞清楚这点，需要先回到第二步的 load()方法当中。还记得 load()方法是返回一个 DrawableTypeRequest 对象吗？刚才我们只是分析了 DrawableTypeRequest 当中的 asBitmap()和 asGif()方法，并没有仔细看它的构造函数，现在我们重新来看一下 DrawableTypeRequest 类的构造函数：

```
public class DrawableTypeRequest<ModelType> extends DrawableRequestBuilder<ModelType>
implements DownloadOptions {

    private final ModelLoader<ModelType, InputStream> streamModelLoader;
    private final ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A, ImageVideoWrapper, Z, R> buildProvider(Glide
glide,
            ModelLoader<A, InputStream> streamModelLoader,
            ModelLoader<A, ParcelFileDescriptor> fileDescriptorModelLoader, Class<Z>
resourceClass,
            Class<R> transcodedClass,
            ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null && fileDescriptorModelLoader == null) {
            return null;
        }
        if (transcoder == null) {
            transcoder = glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z> dataLoadProvider =
```

```
glide.buildDataProvider(ImageVideoWrapper.class,
                resourceClass);
        ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
                fileDescriptorModelLoader);
        return new FixedLoadProvider<A, ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
    }

    DrawableTypeRequest(Class<ModelType> modelClass, ModelLoader<ModelType,
InputStream> streamModelLoader,
            ModelLoader<ModelType, ParcelFileDescriptor> fileDescriptorModelLoader,
Context context, Glide glide,
            RequestTracker requestTracker, Lifecycle lifecycle, RequestManager.OptionsApplier
optionsApplier) {
        super(context, modelClass,
                buildProvider(glide, streamModelLoader, fileDescriptorModelLoader,
GifBitmapWrapper.class,
                        GlideDrawable.class, null),
                glide, requestTracker, lifecycle);
        this.streamModelLoader = streamModelLoader;
        this.fileDescriptorModelLoader = fileDescriptorModelLoader;
        this.optionsApplier = optionsApplier;
    }

    ...
}
```

可以看到，这里在第 29 行，也就是构造函数中，调用了一个 buildProvider()方法，并把 streamModelLoader 和 fileDescriptorModelLoader 等参数传入到这个方法中，这两个 ModelLoader 就是之前在 loadGeneric()方法中构建出来的。

那么我们再来看一下 buildProvider()方法里面做了什么，在第 16 行调用了 glide.buildTranscoder()方法来构建一个 ResourceTranscoder，它是用于对图片进行转码的，由于 ResourceTranscoder 是一个接口，这里实际会构建出一个 GifBitmapWrapperDrawableTranscoder 对象。

接下来在第 18 行调用了 glide.buildDataProvider()方法来构建一个 DataLoadProvider，它是用于对图片进行编解码的，由于 DataLoadProvider 是一个接口，这里实际会构建出一个 ImageVideoGifDrawableLoadProvider 对象。

然后在第 20 行，new 了一个 ImageVideoModelLoader 的实例，并把之前 loadGeneric()方法中构建的两个 ModelLoader 封装到了 ImageVideoModelLoader 当中。

最后，在第 22 行，new 出一个 FixedLoadProvider，并把刚才构建的出来的

GifBitmapWrapperDrawableTranscoder、ImageVideoModelLoader、ImageVideoGifDrawableLoadProvider 都封装进去，这个也就是 onSizeReady()方法中的 loadProvider 了。

好的，那么我们回到 onSizeReady()方法中，在 onSizeReady()方法的第 12 行和第 18 行，分别调用了 loadProvider 的 getModelLoader()方法和 getTranscoder()方法，那么得到的对象也就是刚才我们分析的 ImageVideoModelLoader 和 GifBitmapWrapperDrawableTranscoder 了。而在第 13 行，又调用了 ImageVideoModelLoader 的 getResourceFetcher()方法，这里我们又需要跟进去瞧一瞧了，代码如下所示：

```
public class ImageVideoModelLoader<A> implements ModelLoader<A, ImageVideoWrapper> {
    private static final String TAG = "IVML";

    private final ModelLoader<A, InputStream> streamLoader;
    private final ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader;

    public ImageVideoModelLoader(ModelLoader<A, InputStream> streamLoader,
            ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader) {
        if (streamLoader == null && fileDescriptorLoader == null) {
            throw new NullPointerException("At least one of streamLoader and
fileDescriptorLoader must be non null");
        }
        this.streamLoader = streamLoader;
        this.fileDescriptorLoader = fileDescriptorLoader;
    }

    @Override
    public DataFetcher<ImageVideoWrapper> getResourceFetcher(A model, int width, int height)
{
        DataFetcher<InputStream> streamFetcher = null;
        if (streamLoader != null) {
            streamFetcher = streamLoader.getResourceFetcher(model, width, height);
        }
        DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher = null;
        if (fileDescriptorLoader != null) {
            fileDescriptorFetcher = fileDescriptorLoader.getResourceFetcher(model, width,
height);
        }

        if (streamFetcher != null || fileDescriptorFetcher != null) {
            return new ImageVideoFetcher(streamFetcher, fileDescriptorFetcher);
        } else {
            return null;
        }
```

```
    }

    static class ImageVideoFetcher implements DataFetcher<ImageVideoWrapper> {
        private final DataFetcher<InputStream> streamFetcher;
        private final DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher;

        public ImageVideoFetcher(DataFetcher<InputStream> streamFetcher,
                DataFetcher<ParcelFileDescriptor> fileDescriptorFetcher) {
            this.streamFetcher = streamFetcher;
            this.fileDescriptorFetcher = fileDescriptorFetcher;
        }

        ...
    }
}
```

可以看到，在第 20 行会先调用 streamLoader.getResourceFetcher()方法获取一个 DataFetcher，而这个 streamLoader 其实就是我们在 loadGeneric()方法中构建出的 StreamStringLoader，调用它的 getResourceFetcher()方法会得到一个 HttpUrlFetcher 对象。然后在第 28 行 new 出了一个 ImageVideoFetcher 对象，并把获得的 HttpUrlFetcher 对象传进去。也就是说，ImageVideoModelLoader 的 getResourceFetcher()方法得到的是一个 ImageVideoFetcher。

那么我们再次回到 onSizeReady()方法，在 onSizeReady()方法的第 23 行，这里将刚才获得的 ImageVideoFetcher、GifBitmapWrapperDrawableTranscoder 等等一系列的值一起传入到了 Engine 的 load()方法当中。接下来我们就要看一看，这个 Engine 的 load()方法当中，到底做了什么？代码如下所示：

```
public class Engine implements EngineJobListener,
        MemoryCache.ResourceRemovedListener,
        EngineResource.ResourceListener {

    ...

    public <T, Z, R> LoadStatus load(Key signature, int width, int height, DataFetcher<T> fetcher,
            DataLoadProvider<T, Z> loadProvider, Transformation<Z> transformation,
ResourceTranscoder<Z, R> transcoder,
            Priority priority, boolean isMemoryCacheable, DiskCacheStrategy
diskCacheStrategy, ResourceCallback cb) {
        Util.assertMainThread();
        long startTime = LogTime.getLogTime();

        final String id = fetcher.getId();
        EngineKey key = keyFactory.buildKey(id, signature, width, height,
loadProvider.getCacheDecoder(),
                loadProvider.getSourceDecoder(), transformation, loadProvider.getEncoder(),
```

```
                transcoder, loadProvider.getSourceEncoder());

        EngineResource<?> cached = loadFromCache(key, isMemoryCacheable);
        if (cached != null) {
            cb.onResourceReady(cached);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from cache", startTime, key);
            }
            return null;
        }

        EngineResource<?> active = loadFromActiveResources(key, isMemoryCacheable);
        if (active != null) {
            cb.onResourceReady(active);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from active resources", startTime,
key);
            }
            return null;
        }

        EngineJob current = jobs.get(key);
        if (current != null) {
            current.addCallback(cb);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Added to existing load", startTime, key);
            }
            return new LoadStatus(cb, current);
        }

        EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheable);
        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width, height, fetcher,
loadProvider, transformation,
                transcoder, diskCacheProvider, diskCacheStrategy, priority);
        EngineRunnable runnable = new EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load", startTime, key);
        }
        return new LoadStatus(cb, engineJob);
    }
```

```
    ...
}
```

load()方法中的代码虽然有点长，但大多数的代码都是在处理缓存的。关于缓存的内容我们会在下一篇文章当中学习，现在只需要从第 45 行看起就行。这里构建了一个 EngineJob，它的主要作用就是用来开启线程的，为后面的异步加载图片做准备。接下来第 46 行创建了一个 DecodeJob 对象，从名字上来看，它好像是用来对图片进行解码的，但实际上它的任务十分繁重，待会我们就知道了。继续往下看，第 48 行创建了一个 EngineRunnable 对象，并且在 51 行调用了 EngineJob 的 start()方法来运行 EngineRunnable 对象，这实际上就是让 EngineRunnable 的 run()方法在子线程当中执行了。那么我们现在就可以去看看 EngineRunnable 的 run()方法里做了些什么，如下所示：

```
@Override
public void run() {
    if (isCancelled) {
        return;
    }
    Exception exception = null;
    Resource<?> resource = null;
    try {
        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        exception = e;
    }
    if (isCancelled) {
        if (resource != null) {
            resource.recycle();
        }
        return;
    }
    if (resource == null) {
        onLoadFailed(exception);
    } else {
        onLoadComplete(resource);
    }
}
```

这个方法中的代码并不多，但我们仍然还是要抓重点。在第 9 行，这里调用了一个 decode()方法，并且这个方法返回了一个 Resource 对象。看上去所有的逻辑应该都在这个 decode()方法执行的了，那我们跟进去瞧一瞧：

```
private Resource<?> decode() throws Exception {
```

```
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}
```

decode()方法中又分了两种情况，从缓存当中去 decode 图片的话就会执行 decodeFromCache()，否则的话就执行 decodeFromSource()。本篇文章中我们不讨论缓存的情况，那么就直接来看 decodeFromSource()方法的代码吧，如下所示：

```
private Resource<?> decodeFromSource() throws Exception {
    return decodeJob.decodeFromSource();
}
```

这里又调用了 DecodeJob 的 decodeFromSource()方法。刚才已经说了，DecodeJob 的任务十分繁重，我们继续跟进看一看吧：

```
class DecodeJob<A, T, Z> {

    ...

    public Resource<Z> decodeFromSource() throws Exception {
        Resource<T> decoded = decodeSource();
        return transformEncodeAndTranscode(decoded);
    }

    private Resource<T> decodeSource() throws Exception {
        Resource<T> decoded = null;
        try {
            long startTime = LogTime.getLogTime();
            final A data = fetcher.loadData(priority);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Fetched data", startTime);
            }
            if (isCancelled) {
                return null;
            }
            decoded = decodeFromSourceData(data);
        } finally {
            fetcher.cleanup();
        }
        return decoded;
    }

    ...
```

}

主要的方法就这些，我都帮大家提取出来了。那么我们先来看一下 decodeFromSource()方法，其实它的工作分为两部，第一步是调用 decodeSource()方法来获得一个 Resource 对象，第二步是调用 transformEncodeAndTranscode()方法来处理这个 Resource 对象。

那么我们先来看第一步，decodeSource()方法中的逻辑也并不复杂，首先在第 14 行调用了 fetcher.loadData()方法。那么这个 fetcher 是什么呢？其实就是刚才在 onSizeReady()方法中得到的 ImageVideoFetcher 对象，这里调用它的 loadData()方法，代码如下所示：

```
@Override
public ImageVideoWrapper loadData(Priority priority) throws Exception {
    InputStream is = null;
    if (streamFetcher != null) {
        try {
            is = streamFetcher.loadData(priority);
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching input stream, trying ParcelFileDescriptor", e);
            }
            if (fileDescriptorFetcher == null) {
                throw e;
            }
        }
    }
    ParcelFileDescriptor fileDescriptor = null;
    if (fileDescriptorFetcher != null) {
        try {
            fileDescriptor = fileDescriptorFetcher.loadData(priority);
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching ParcelFileDescriptor", e);
            }
            if (is == null) {
                throw e;
            }
        }
    }
    return new ImageVideoWrapper(is, fileDescriptor);
}
```

可以看到，在 ImageVideoFetcher 的 loadData()方法的第 6 行，这里又去调用了 streamFetcher.loadData()方法，那么这个 streamFetcher 是什么呢？自然就是刚才在组装 ImageVideoFetcher 对象时传进来的 HttpUrlFetcher 了。因此这里又会去调用 HttpUrlFetcher 的 loadData()方法，那么我们继续跟进去瞧一瞧：

```
public class HttpUrlFetcher implements DataFetcher<InputStream> {

    ...

    @Override
    public InputStream loadData(Priority priority) throws Exception {
        return loadDataWithRedirects(glideUrl.toURL(), 0 /*redirects*/, null /*lastUrl*/, glideUrl.getHeaders());
    }

    private InputStream loadDataWithRedirects(URL url, int redirects, URL lastUrl, Map<String, String> headers)
            throws IOException {
        if (redirects >= MAXIMUM_REDIRECTS) {
            throw new IOException("Too many (> " + MAXIMUM_REDIRECTS + ") redirects!");
        } else {
            // Comparing the URLs using .equals performs additional network I/O and is generally broken.
            //                                                                                    See http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html.
            try {
                if (lastUrl != null && url.toURI().equals(lastUrl.toURI())) {
                    throw new IOException("In re-direct loop");
                }
            } catch (URISyntaxException e) {
                // Do nothing, this is best effort.
            }
        }
        urlConnection = connectionFactory.build(url);
        for (Map.Entry<String, String> headerEntry : headers.entrySet()) {
           urlConnection.addRequestProperty(headerEntry.getKey(), headerEntry.getValue());
        }
        urlConnection.setConnectTimeout(2500);
        urlConnection.setReadTimeout(2500);
        urlConnection.setUseCaches(false);
        urlConnection.setDoInput(true);

        // Connect explicitly to avoid errors in decoders if connection fails.
        urlConnection.connect();
        if (isCancelled) {
            return null;
        }
        final int statusCode = urlConnection.getResponseCode();
        if (statusCode / 100 == 2) {
```

```
                return getStreamForSuccessfulRequest(urlConnection);
            } else if (statusCode / 100 == 3) {
                String redirectUrlString = urlConnection.getHeaderField("Location");
                if (TextUtils.isEmpty(redirectUrlString)) {
                    throw new IOException("Received empty or null redirect url");
                }
                URL redirectUrl = new URL(url, redirectUrlString);
                return loadDataWithRedirects(redirectUrl, redirects + 1, url, headers);
            } else {
                if (statusCode == -1) {
                    throw new IOException("Unable to retrieve response code from HttpUrlConnection.");
                }
                throw new IOException("Request failed " + statusCode + ": " + urlConnection.getResponseMessage());
            }
        }

        private InputStream getStreamForSuccessfulRequest(HttpURLConnection urlConnection)
                throws IOException {
            if (TextUtils.isEmpty(urlConnection.getContentEncoding())) {
                int contentLength = urlConnection.getContentLength();
                stream = ContentLengthInputStream.obtain(urlConnection.getInputStream(), contentLength);
            } else {
                if (Log.isLoggable(TAG, Log.DEBUG)) {
                    Log.d(TAG, "Got non empty content encoding: " + urlConnection.getContentEncoding());
                }
                stream = urlConnection.getInputStream();
            }
            return stream;
        }

        ...
}
```

经过一层一层地跋山涉水，我们终于在这里找到网络通讯的代码了！之前有朋友跟我讲过，说 Glide 的源码实在是太复杂了，甚至连网络请求是在哪里发出去的都找不到。我们也是经过一段一段又一段的代码跟踪，终于把网络请求的代码给找出来了，实在是太不容易了。

不过也别高兴得太早，现在离最终分析完还早着呢。可以看到，loadData()方法只是返回了一个 InputStream，服务器返回的数据连读都还没开始读呢。所以我们还是要静下心来继续分析，回到刚才 ImageVideoFetcher 的 loadData()方法中，在这个方法的最后一行，创建了一个 ImageVideoWrapper 对象，并把刚才得到的 InputStream 作为参数传了进去。

然后我们回到再上一层，也就是 DecodeJob 的 decodeSource()方法当中，在得到了这个 ImageVideoWrapper 对象之后，紧接着又将这个对象传入到了 decodeFromSourceData()当中，来去解码这个对象。decodeFromSourceData()方法的代码如下所示：

```
private Resource<T> decodeFromSourceData(A data) throws IOException {
    final Resource<T> decoded;
    if (diskCacheStrategy.cacheSource()) {
        decoded = cacheAndDecodeSourceData(data);
    } else {
        long startTime = LogTime.getLogTime();
        decoded = loadProvider.getSourceDecoder().decode(data, width, height);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Decoded from source", startTime);
        }
    }
    return decoded;
}
```

可以看到，这里在第 7 行调用了 loadProvider.getSourceDecoder().decode()方法来进行解码。loadProvider 就是刚才在 onSizeReady()方法中得到的 FixedLoadProvider，而 getSourceDecoder()得到的则是一个 GifBitmapWrapperResourceDecoder 对象，也就是要调用这个对象的 decode()方法来对图片进行解码。那么我们来看下 GifBitmapWrapperResourceDecoder 的代码：

```
public class GifBitmapWrapperResourceDecoder implements ResourceDecoder<ImageVideoWrapper, GifBitmapWrapper> {

    ...

    @SuppressWarnings("resource")
    // @see ResourceDecoder.decode
    @Override
    public Resource<GifBitmapWrapper> decode(ImageVideoWrapper source, int width, int height) throws IOException {
        ByteArrayPool pool = ByteArrayPool.get();
        byte[] tempBytes = pool.getBytes();
        GifBitmapWrapper wrapper = null;
        try {
            wrapper = decode(source, width, height, tempBytes);
        } finally {
            pool.releaseBytes(tempBytes);
        }
        return wrapper != null ? new GifBitmapWrapperResource(wrapper) : null;
    }
```

```
    private GifBitmapWrapper decode(ImageVideoWrapper source, int width, int height, byte[] bytes) throws IOException {
        final GifBitmapWrapper result;
        if (source.getStream() != null) {
            result = decodeStream(source, width, height, bytes);
        } else {
            result = decodeBitmapWrapper(source, width, height);
        }
        return result;
    }

    private GifBitmapWrapper decodeStream(ImageVideoWrapper source, int width, int height, byte[] bytes)
            throws IOException {
        InputStream bis = streamFactory.build(source.getStream(), bytes);
        bis.mark(MARK_LIMIT_BYTES);
        ImageHeaderParser.ImageType type = parser.parse(bis);
        bis.reset();
        GifBitmapWrapper result = null;
        if (type == ImageHeaderParser.ImageType.GIF) {
            result = decodeGifWrapper(bis, width, height);
        }
        // Decoding the gif may fail even if the type matches.
        if (result == null) {
            // We can only reset the buffered InputStream, so to start from the beginning of the stream, we need to
            // pass in a new source containing the buffered stream rather than the original stream.
            ImageVideoWrapper forBitmapDecoder = new ImageVideoWrapper(bis, source.getFileDescriptor());
            result = decodeBitmapWrapper(forBitmapDecoder, width, height);
        }
        return result;
    }

    private GifBitmapWrapper decodeBitmapWrapper(ImageVideoWrapper toDecode, int width, int height) throws IOException {
        GifBitmapWrapper result = null;
        Resource<Bitmap> bitmapResource = bitmapDecoder.decode(toDecode, width, height);
        if (bitmapResource != null) {
            result = new GifBitmapWrapper(bitmapResource, null);
        }
        return result;
    }
```

```
    ...
}
```

首先，在 decode()方法中，又去调用了另外一个 decode()方法的重载。然后在第 23 行调用了 decodeStream()方法，准备从服务器返回的流当中读取数据。decodeStream()方法中会先从流中读取 2 个字节的数据，来判断这张图是 GIF 图还是普通的静图，如果是 GIF 图就调用 decodeGifWrapper()方法来进行解码，如果是普通的静图就用调用 decodeBitmapWrapper()方法来进行解码。这里我们只分析普通静图的实现流程，GIF 图的实现有点过于复杂了，无法在本篇文章当中分析。

然后我们来看一下 decodeBitmapWrapper()方法，这里在第 52 行调用了 bitmapDecoder.decode()方法。这个 bitmapDecoder 是一个 ImageVideoBitmapDecoder 对象，那么我们来看一下它的代码，如下所示：

```
public class ImageVideoBitmapDecoder implements ResourceDecoder<ImageVideoWrapper, Bitmap> {
    private final ResourceDecoder<InputStream, Bitmap> streamDecoder;
    private final ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDescriptorDecoder;

    public ImageVideoBitmapDecoder(ResourceDecoder<InputStream, Bitmap> streamDecoder,
            ResourceDecoder<ParcelFileDescriptor, Bitmap> fileDescriptorDecoder) {
        this.streamDecoder = streamDecoder;
        this.fileDescriptorDecoder = fileDescriptorDecoder;
    }

    @Override
    public Resource<Bitmap> decode(ImageVideoWrapper source, int width, int height) throws IOException {
        Resource<Bitmap> result = null;
        InputStream is = source.getStream();
        if (is != null) {
            try {
                result = streamDecoder.decode(is, width, height);
            } catch (IOException e) {
                if (Log.isLoggable(TAG, Log.VERBOSE)) {
                    Log.v(TAG, "Failed to load image from stream, trying FileDescriptor", e);
                }
            }
        }
        if (result == null) {
            ParcelFileDescriptor fileDescriptor = source.getFileDescriptor();
            if (fileDescriptor != null) {
                result = fileDescriptorDecoder.decode(fileDescriptor, width, height);
            }
```

```
            }
            return result;
        }

        ...
}
```

代码并不复杂，在第 14 行先调用了 source.getStream()来获取到服务器返回的 InputStream，然后在第 17 行调用 streamDecoder.decode()方法进行解码。streamDecode 是一个 StreamBitmapDecoder 对象，那么我们再来看这个类的源码，如下所示：

```
public class StreamBitmapDecoder implements ResourceDecoder<InputStream, Bitmap> {

    ...

    private final Downsampler downsampler;
    private BitmapPool bitmapPool;
    private DecodeFormat decodeFormat;

    public StreamBitmapDecoder(Downsampler downsampler, BitmapPool bitmapPool,
DecodeFormat decodeFormat) {
        this.downsampler = downsampler;
        this.bitmapPool = bitmapPool;
        this.decodeFormat = decodeFormat;
    }

    @Override
    public Resource<Bitmap> decode(InputStream source, int width, int height) {
        Bitmap bitmap = downsampler.decode(source, bitmapPool, width, height,
decodeFormat);
        return BitmapResource.obtain(bitmap, bitmapPool);
    }

    ...
}
```

可以看到，它的 decode()方法又去调用了 Downsampler 的 decode()方法。接下来又到了激动人心的时刻了，Downsampler 的代码如下所示：

```
public abstract class Downsampler implements BitmapDecoder<InputStream> {

    ...

    @Override
    public Bitmap decode(InputStream is, BitmapPool pool, int outWidth, int outHeight,
DecodeFormat decodeFormat) {
```

```
final ByteArrayPool byteArrayPool = ByteArrayPool.get();
final byte[] bytesForOptions = byteArrayPool.getBytes();
final byte[] bytesForStream = byteArrayPool.getBytes();
final BitmapFactory.Options options = getDefaultOptions();
// Use to fix the mark limit to avoid allocating buffers that fit entire images.
RecyclableBufferedInputStream bufferedStream = new RecyclableBufferedInputStream(
        is, bytesForStream);
// Use to retrieve exceptions thrown while reading.
// TODO(#126): when the framework no longer returns partially decoded Bitmaps or provides a way to determine
// if a Bitmap is partially decoded, consider removing.
ExceptionCatchingInputStream exceptionStream =
        ExceptionCatchingInputStream.obtain(bufferedStream);
// Use to read data.
// Ensures that we can always reset after reading an image header so that we can still attempt to decode the
// full image even when the header decode fails and/or overflows our read buffer. See #283.
MarkEnforcingInputStream invalidatingStream = new MarkEnforcingInputStream(exceptionStream);
try {
    exceptionStream.mark(MARK_POSITION);
    int orientation = 0;
    try {
        orientation = new ImageHeaderParser(exceptionStream).getOrientation();
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.WARN)) {
            Log.w(TAG, "Cannot determine the image orientation from header", e);
        }
    } finally {
        try {
            exceptionStream.reset();
        } catch (IOException e) {
            if (Log.isLoggable(TAG, Log.WARN)) {
                Log.w(TAG, "Cannot reset the input stream", e);
            }
        }
    }
    options.inTempStorage = bytesForOptions;
    final int[] inDimens = getDimensions(invalidatingStream, bufferedStream, options);
    final int inWidth = inDimens[0];
    final int inHeight = inDimens[1];
    final int degreesToRotate = TransformationUtils.getExifOrientationDegrees(orientation);
```

```
                final int sampleSize = getRoundedSampleSize(degreesToRotate, inWidth, inHeight,
outWidth, outHeight);
                final Bitmap downsampled =
                        downsampleWithSize(invalidatingStream, bufferedStream, options, pool,
inWidth, inHeight, sampleSize,
                                decodeFormat);
                // BitmapFactory swallows exceptions during decodes and in some cases when
inBitmap is non null, may catch
                // and log a stack trace but still return a non null bitmap. To avoid displaying
partially decoded bitmaps,
                // we catch exceptions reading from the stream in our
ExceptionCatchingInputStream and throw them here.
                final Exception streamException = exceptionStream.getException();
                if (streamException != null) {
                    throw new RuntimeException(streamException);
                }
                Bitmap rotated = null;
                if (downsampled != null) {
                    rotated = TransformationUtils.rotateImageExif(downsampled, pool,
orientation);
                    if (!downsampled.equals(rotated) && !pool.put(downsampled)) {
                        downsampled.recycle();
                    }
                }
                return rotated;
            } finally {
                byteArrayPool.releaseBytes(bytesForOptions);
                byteArrayPool.releaseBytes(bytesForStream);
                exceptionStream.release();
                releaseOptions(options);
            }
        }

        private Bitmap downsampleWithSize(MarkEnforcingInputStream is,
RecyclableBufferedInputStream bufferedStream,
                BitmapFactory.Options options, BitmapPool pool, int inWidth, int inHeight, int
sampleSize,
                DecodeFormat decodeFormat) {
            // Prior to KitKat, the inBitmap size must exactly match the size of the bitmap we're
decoding.
            Bitmap.Config config = getConfig(is, decodeFormat);
            options.inSampleSize = sampleSize;
            options.inPreferredConfig = config;
            if ((options.inSampleSize == 1 || Build.VERSION_CODES.KITKAT <=
```

```
Build.VERSION.SDK_INT) && shouldUsePool(is)) {
                int targetWidth = (int) Math.ceil(inWidth / (double) sampleSize);
                int targetHeight = (int) Math.ceil(inHeight / (double) sampleSize);
                // BitmapFactory will clear out the Bitmap before writing to it, so getDirty is safe.
                setInBitmap(options, pool.getDirty(targetWidth, targetHeight, config));
            }
            return decodeStream(is, bufferedStream, options);
        }

        /**
         * A method for getting the dimensions of an image from the given InputStream.
         *
         * @param is The InputStream representing the image.
         * @param options The options to pass to
         *                {@link BitmapFactory#decodeStream(InputStream, android.graphics.Rect,
         *                    BitmapFactory.Options)}.
         * @return an array containing the dimensions of the image in the form {width, height}.
         */
        public int[] getDimensions(MarkEnforcingInputStream is, RecyclableBufferedInputStream
bufferedStream,
                BitmapFactory.Options options) {
            options.inJustDecodeBounds = true;
            decodeStream(is, bufferedStream, options);
            options.inJustDecodeBounds = false;
            return new int[] { options.outWidth, options.outHeight };
        }

        private static Bitmap decodeStream(MarkEnforcingInputStream is,
RecyclableBufferedInputStream bufferedStream,
                BitmapFactory.Options options) {
            if (options.inJustDecodeBounds) {
                // This is large, but jpeg headers are not size bounded so we need something
large enough to minimize
                // the possibility of not being able to fit enough of the header in the buffer to get
the image size so
                // that we don't fail to load images. The BufferedInputStream will create a new
buffer of 2x the
                // original size each time we use up the buffer space without passing the mark so
this is a maximum
                // bound on the buffer size, not a default. Most of the time we won't go past our
pre-allocated 16kb.
                is.mark(MARK_POSITION);
            } else {
                // Once we've read the image header, we no longer need to allow the buffer to
```

```
expand in size. To avoid
                // unnecessary allocations reading image data, we fix the mark limit so that it is
no larger than our
                // current buffer size here. See issue #225.
                bufferedStream.fixMarkLimit();
            }
            final Bitmap result = BitmapFactory.decodeStream(is, null, options);
            try {
                if (options.inJustDecodeBounds) {
                    is.reset();
                }
            } catch (IOException e) {
                if (Log.isLoggable(TAG, Log.ERROR)) {
                    Log.e(TAG, "Exception loading inDecodeBounds=" +
options.inJustDecodeBounds
                            + " sample=" + options.inSampleSize, e);
                }
            }

            return result;
        }

        ...
}
```

可以看到，对服务器返回的 InputStream 的读取，以及对图片的加载全都在这里了。当然这里其实处理了很多的逻辑，包括对图片的压缩，甚至还有旋转、圆角等逻辑处理，但是我们目前只需要关注主线逻辑就行了。decode()方法执行之后，会返回一个 Bitmap 对象，那么图片在这里其实也就已经被加载出来了，剩下的工作就是如果让这个 Bitmap 显示到界面上，我们继续往下分析。

回到刚才的 StreamBitmapDecoder 当中，你会发现，它的 decode()方法返回的是一个 Resource<Bitmap>对象。而我们从 Downsampler 中得到的是一个 Bitmap 对象，因此这里在第 18 行又调用了 BitmapResource.obtain()方法，将 Bitmap 对象包装成了 Resource<Bitmap>对象。代码如下所示：

```
public class BitmapResource implements Resource<Bitmap> {
    private final Bitmap bitmap;
    private final BitmapPool bitmapPool;

    /**
     * Returns a new {@link BitmapResource} wrapping the given {@link Bitmap} if the Bitmap
is non-null or null if the
     * given Bitmap is null.
     *
```

```
     * @param bitmap A Bitmap.
     * @param bitmapPool A non-null {@link BitmapPool}.
     */
    public static BitmapResource obtain(Bitmap bitmap, BitmapPool bitmapPool) {
        if (bitmap == null) {
            return null;
        } else {
            return new BitmapResource(bitmap, bitmapPool);
        }
    }

    public BitmapResource(Bitmap bitmap, BitmapPool bitmapPool) {
        if (bitmap == null) {
            throw new NullPointerException("Bitmap must not be null");
        }
        if (bitmapPool == null) {
            throw new NullPointerException("BitmapPool must not be null");
        }
        this.bitmap = bitmap;
        this.bitmapPool = bitmapPool;
    }

    @Override
    public Bitmap get() {
        return bitmap;
    }

    @Override
    public int getSize() {
        return Util.getBitmapByteSize(bitmap);
    }

    @Override
    public void recycle() {
        if (!bitmapPool.put(bitmap)) {
            bitmap.recycle();
        }
    }
}
```

BitmapResource 的源码也非常简单，经过这样一层包装之后，如果我还需要获取 Bitmap，只需要调用 Resource<Bitmap>的 get()方法就可以了。

然后我们需要一层层继续向上返回，StreamBitmapDecoder 会将值返回到 ImageVideoBitmapDecoder 当中，而 ImageVideoBitmapDecoder 又会将值返回到

GifBitmapWrapperResourceDecoder 的 decodeBitmapWrapper()方法当中。由于代码隔得有点太远了，我重新把 decodeBitmapWrapper()方法的代码贴一下：

```
private GifBitmapWrapper decodeBitmapWrapper(ImageVideoWrapper toDecode, int width, int height) throws IOException {
    GifBitmapWrapper result = null;
    Resource<Bitmap> bitmapResource = bitmapDecoder.decode(toDecode, width, height);
    if (bitmapResource != null) {
        result = new GifBitmapWrapper(bitmapResource, null);
    }
    return result;
}
```

可以看到，decodeBitmapWrapper()方法返回的是一个 GifBitmapWrapper 对象。因此，这里在第 5 行，又将 Resource<Bitmap>封装到了一个 GifBitmapWrapper 对象当中。这个 GifBitmapWrapper 顾名思义，就是既能封装 GIF，又能封装 Bitmap，从而保证了不管是什么类型的图片 Glide 都能从容应对。我们顺便来看下 GifBitmapWrapper 的源码吧，如下所示：

```
public class GifBitmapWrapper {
    private final Resource<GifDrawable> gifResource;
    private final Resource<Bitmap> bitmapResource;

    public GifBitmapWrapper(Resource<Bitmap> bitmapResource, Resource<GifDrawable> gifResource) {
        if (bitmapResource != null && gifResource != null) {
            throw new IllegalArgumentException("Can only contain either a bitmap resource or a gif resource, not both");
        }
        if (bitmapResource == null && gifResource == null) {
            throw new IllegalArgumentException("Must contain either a bitmap resource or a gif resource");
        }
        this.bitmapResource = bitmapResource;
        this.gifResource = gifResource;
    }

    /**
     * Returns the size of the wrapped resource.
     */
    public int getSize() {
        if (bitmapResource != null) {
            return bitmapResource.getSize();
        } else {
            return gifResource.getSize();
        }
```

```
    }

    /**
     * Returns the wrapped {@link Bitmap} resource if it exists, or null.
     */
    public Resource<Bitmap> getBitmapResource() {
        return bitmapResource;
    }

    /**
     * Returns the wrapped {@link GifDrawable} resource if it exists, or null.
     */
    public Resource<GifDrawable> getGifResource() {
        return gifResource;
    }
}
```

还是比较简单的，就是分别对 gifResource 和 bitmapResource 做了一层封装而已，相信没有什么解释的必要。

然后这个 GifBitmapWrapper 对象会一直向上返回，返回到 GifBitmapWrapperResourceDecoder 最外层的 decode()方法的时候，会对它再做一次封装，如下所示：

```
@Override
public Resource<GifBitmapWrapper> decode(ImageVideoWrapper source, int width, int height)
throws IOException {
    ByteArrayPool pool = ByteArrayPool.get();
    byte[] tempBytes = pool.getBytes();
    GifBitmapWrapper wrapper = null;
    try {
        wrapper = decode(source, width, height, tempBytes);
    } finally {
        pool.releaseBytes(tempBytes);
    }
    return wrapper != null ? new GifBitmapWrapperResource(wrapper) : null;
}
```

可以看到，这里在第 11 行，又将 GifBitmapWrapper 封装到了一个 GifBitmapWrapperResource 对象当中，最终返回的是一个 Resource<GifBitmapWrapper> 对象。这个 GifBitmapWrapperResource 和刚才的 BitmapResource 是相似的，它们都实现的 Resource 接口，都可以通过 get()方法来获取封装起来的具体内容。GifBitmapWrapperResource 的源码如下所示：

```
public class GifBitmapWrapperResource implements Resource<GifBitmapWrapper> {
    private final GifBitmapWrapper data;
```

```
    public GifBitmapWrapperResource(GifBitmapWrapper data) {
        if (data == null) {
            throw new NullPointerException("Data must not be null");
        }
        this.data = data;
    }

    @Override
    public GifBitmapWrapper get() {
        return data;
    }

    @Override
    public int getSize() {
        return data.getSize();
    }

    @Override
    public void recycle() {
        Resource<Bitmap> bitmapResource = data.getBitmapResource();
        if (bitmapResource != null) {
            bitmapResource.recycle();
        }
        Resource<GifDrawable> gifDataResource = data.getGifResource();
        if (gifDataResource != null) {
            gifDataResource.recycle();
        }
    }
}
```

经过这一层的封装之后，我们从网络上得到的图片就能够以 Resource 接口的形式返回，并且还能同时处理 Bitmap 图片和 GIF 图片这两种情况。

那么现在我们可以回到 DecodeJob 当中了，它的 decodeFromSourceData()方法返回的是一个 Resource<T>对象，其实也就是 Resource<GifBitmapWrapper>对象了。然后继续向上返回，最终返回到 decodeFromSource()方法当中，如下所示：

```
public Resource<Z> decodeFromSource() throws Exception {
    Resource<T> decoded = decodeSource();
    return transformEncodeAndTranscode(decoded);
}
```

刚才我们就是从这里跟进到 decodeSource()方法当中，然后执行了一大堆一大堆的逻辑，最终得到了这个 Resource<T>对象。然而你会发现，decodeFromSource()方法最终返回的却是一个 Resource<Z>对象，那么这到底是怎么回事呢？我们就需要跟进到 transformEncodeAndTranscode()方法来瞧一瞧了，代码如下所示：

```
private Resource<Z> transformEncodeAndTranscode(Resource<T> decoded) {
    long startTime = LogTime.getLogTime();
    Resource<T> transformed = transform(decoded);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Transformed resource from source", startTime);
    }
    writeTransformedToCache(transformed);
    startTime = LogTime.getLogTime();
    Resource<Z> result = transcode(transformed);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Transcoded transformed from source", startTime);
    }
    return result;
}

private Resource<Z> transcode(Resource<T> transformed) {
    if (transformed == null) {
        return null;
    }
    return transcoder.transcode(transformed);
}
```

首先，这个方法开头的几行 transform 还有 cache，这都是我们后面才会学习的东西，现在不用管它们就可以了。需要注意的是第 9 行，这里调用了一个 transcode()方法，就把 Resource<T>对象转换成 Resource<Z>对象了。

而 transcode()方法中又是调用了 transcoder 的 transcode()方法，那么这个 transcoder 是什么呢？其实这也是 Glide 源码特别难懂的原因之一，就是它用到的很多对象都是很早很早之前就初始化的，在初始化的时候你可能完全就没有留意过它，因为一时半会根本就用不着，但是真正需要用到的时候你却早就记不起来这个对象是从哪儿来的了。

那么这里我来提醒一下大家吧，在第二步 load()方法返回的那个 DrawableTypeRequest 对象，它的构建函数中去构建了一个 FixedLoadProvider 对象，然后我们将三个参数传入到了 FixedLoadProvider 当中，其中就有一个 GifBitmapWrapperDrawableTranscoder 对象。后来在 onSizeReady()方法中获取到了这个参数，并传递到了 Engine 当中，然后又由 Engine 传递到了 DecodeJob 当中。因此，这里的 transcoder 其实就是这个 GifBitmapWrapperDrawableTranscoder 对象。那么我们来看一下它的源码：

```
public class GifBitmapWrapperDrawableTranscoder implements ResourceTranscoder<GifBitmapWrapper, GlideDrawable> {
    private final ResourceTranscoder<Bitmap, GlideBitmapDrawable> bitmapDrawableResourceTranscoder;

    public GifBitmapWrapperDrawableTranscoder(
```

```
            ResourceTranscoder<Bitmap, GlideBitmapDrawable>
bitmapDrawableResourceTranscoder) {
        this.bitmapDrawableResourceTranscoder = bitmapDrawableResourceTranscoder;
    }

    @Override
    public Resource<GlideDrawable> transcode(Resource<GifBitmapWrapper> toTranscode) {
        GifBitmapWrapper gifBitmap = toTranscode.get();
        Resource<Bitmap> bitmapResource = gifBitmap.getBitmapResource();
        final Resource<? extends GlideDrawable> result;
        if (bitmapResource != null) {
            result = bitmapDrawableResourceTranscoder.transcode(bitmapResource);
        } else {
            result = gifBitmap.getGifResource();
        }
        return (Resource<GlideDrawable>) result;
    }

    ...
}
```

这里我来简单解释一下，GifBitmapWrapperDrawableTranscoder 的核心作用就是用来转码的。因为 GifBitmapWrapper 是无法直接显示到 ImageView 上面的，只有 Bitmap 或者 Drawable 才能显示到 ImageView 上。因此，这里的 transcode()方法先从 Resource<GifBitmapWrapper>中取出 GifBitmapWrapper 对象，然后再从 GifBitmapWrapper 中取出 Resource<Bitmap>对象。

接下来做了一个判断，如果 Resource<Bitmap>为空，那么说明此时加载的是 GIF 图，直接调用 getGifResource()方法将图片取出即可，因为 Glide 用于加载 GIF 图片是使用的 GifDrawable 这个类，它本身就是一个 Drawable 对象了。而如果 Resource<Bitmap>不为空，那么就需要再做一次转码，将 Bitmap 转换成 Drawable 对象才行，因为要保证静图和动图的类型一致性，不然逻辑上是不好处理的。

这里在第 15 行又进行了一次转码，是调用的 GlideBitmapDrawableTranscoder 对象的 transcode()方法，代码如下所示：

```
public class GlideBitmapDrawableTranscoder implements ResourceTranscoder<Bitmap,
GlideBitmapDrawable> {
    private final Resources resources;
    private final BitmapPool bitmapPool;

    public GlideBitmapDrawableTranscoder(Context context) {
        this(context.getResources(), Glide.get(context).getBitmapPool());
    }

    public GlideBitmapDrawableTranscoder(Resources resources, BitmapPool bitmapPool) {
```

```
            this.resources = resources;
            this.bitmapPool = bitmapPool;
        }

        @Override
        public Resource<GlideBitmapDrawable> transcode(Resource<Bitmap> toTranscode) {
            GlideBitmapDrawable drawable = new GlideBitmapDrawable(resources,
toTranscode.get());
            return new GlideBitmapDrawableResource(drawable, bitmapPool);
        }

        ...
}
```

可以看到，这里 new 出了一个 GlideBitmapDrawable 对象，并把 Bitmap 封装到里面。然后对 GlideBitmapDrawable 再进行一次封装，返回一个 Resource<GlideBitmapDrawable>对象。

现在再返回到 GifBitmapWrapperDrawableTranscoder 的 transcode()方法中，你会发现它们的类型就一致了。因为不管是静图的 Resource<GlideBitmapDrawable>对象，还是动图的 Resource<GifDrawable>对象，它们都是属于父类 Resource<GlideDrawable>对象的。因此 transcode()方法也是直接返回了 Resource<GlideDrawable>，而这个 Resource<GlideDrawable> 其实也就是转换过后的 Resource<Z>了。

那么我们继续回到 DecodeJob 当中，它的 decodeFromSource()方法得到了 Resource<Z>对象，当然也就是 Resource<GlideDrawable>对象。然后继续向上返回会回到 EngineRunnable 的 decodeFromSource()方法，再回到 decode()方法，再回到 run()方法当中。那么我们重新再贴一下 EngineRunnable run()方法的源码：

```
@Override
public void run() {
    if (isCancelled) {
        return;
    }
    Exception exception = null;
    Resource<?> resource = null;
    try {
        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        exception = e;
    }
    if (isCancelled) {
        if (resource != null) {
```

```
                resource.recycle();
            }
            return;
        }
        if (resource == null) {
            onLoadFailed(exception);
        } else {
            onLoadComplete(resource);
        }
}
```

也就是说，经过第 9 行 decode()方法的执行，我们最终得到了这个 Resource<GlideDrawable>对象，那么接下来就是如何将它显示出来了。可以看到，这里在第 25 行调用了 onLoadComplete()方法，表示图片加载已经完成了，代码如下所示：

```
private void onLoadComplete(Resource resource) {
    manager.onResourceReady(resource);
}
```

这个 manager 就是 EngineJob 对象，因此这里实际上调用的是 EngineJob 的 onResourceReady()方法，代码如下所示：

```
class EngineJob implements EngineRunnable.EngineRunnableManager {

    private static final Handler MAIN_THREAD_HANDLER = new
Handler(Looper.getMainLooper(), new MainThreadCallback());

    private final List<ResourceCallback> cbs = new ArrayList<ResourceCallback>();

    ...

    public void addCallback(ResourceCallback cb) {
        Util.assertMainThread();
        if (hasResource) {
            cb.onResourceReady(engineResource);
        } else if (hasException) {
            cb.onException(exception);
        } else {
            cbs.add(cb);
        }
    }

    @Override
    public void onResourceReady(final Resource<?> resource) {
        this.resource = resource;
        MAIN_THREAD_HANDLER.obtainMessage(MSG_COMPLETE, this).sendToTarget();
```

```
    }

    private void handleResultOnMainThread() {
        if (isCancelled) {
            resource.recycle();
            return;
        } else if (cbs.isEmpty()) {
            throw new IllegalStateException("Received a resource without any callbacks to notify");
        }
        engineResource = engineResourceFactory.build(resource, isCacheable);
        hasResource = true;
        engineResource.acquire();
        listener.onEngineJobComplete(key, engineResource);
        for (ResourceCallback cb : cbs) {
            if (!isInIgnoredCallbacks(cb)) {
                engineResource.acquire();
                cb.onResourceReady(engineResource);
            }
        }
        engineResource.release();
    }

    @Override
    public void onException(final Exception e) {
        this.exception = e;
        MAIN_THREAD_HANDLER.obtainMessage(MSG_EXCEPTION, this).sendToTarget();
    }

    private void handleExceptionOnMainThread() {
        if (isCancelled) {
            return;
        } else if (cbs.isEmpty()) {
            throw new IllegalStateException("Received an exception without any callbacks to notify");
        }
        hasException = true;
        listener.onEngineJobComplete(key, null);
        for (ResourceCallback cb : cbs) {
            if (!isInIgnoredCallbacks(cb)) {
                cb.onException(exception);
            }
        }
    }
```

```
    private static class MainThreadCallback implements Handler.Callback {

        @Override
        public boolean handleMessage(Message message) {
            if (MSG_COMPLETE == message.what || MSG_EXCEPTION == message.what) {
                EngineJob job = (EngineJob) message.obj;
                if (MSG_COMPLETE == message.what) {
                    job.handleResultOnMainThread();
                } else {
                    job.handleExceptionOnMainThread();
                }
                return true;
            }
            return false;
        }
    }

    ...
}
```

可以看到，这里在 onResourceReady()方法使用 Handler 发出了一条 MSG_COMPLETE 消息，那么在 MainThreadCallback 的 handleMessage()方法中就会收到这条消息。从这里开始，所有的逻辑又回到主线程当中进行了，因为很快就需要更新 UI 了。

然后在第 72 行调用了 handleResultOnMainThread()方法，这个方法中又通过一个循环，调用了所有 ResourceCallback 的 onResourceReady()方法。那么这个 ResourceCallback 是什么呢？答案在 addCallback()方法当中，它会向 cbs 集合中去添加 ResourceCallback。那么这个 addCallback()方法又是哪里调用的呢？其实调用的地方我们早就已经看过了，只不过之前没有注意，现在重新来看一下 Engine 的 load()方法，如下所示：

```
public class Engine implements EngineJobListener,
        MemoryCache.ResourceRemovedListener,
        EngineResource.ResourceListener {

    ...

    public <T, Z, R> LoadStatus load(Key signature, int width, int height, DataFetcher<T> fetcher,
            DataLoadProvider<T, Z> loadProvider, Transformation<Z> transformation,
ResourceTranscoder<Z, R> transcoder, Priority priority,
            boolean isMemoryCacheable, DiskCacheStrategy diskCacheStrategy,
ResourceCallback cb) {

        ...
```

```
            EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheable);
            DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>(key, width, height, fetcher,
loadProvider, transformation,
                    transcoder, diskCacheProvider, diskCacheStrategy, priority);
            EngineRunnable runnable = new EngineRunnable(engineJob, decodeJob, priority);
            jobs.put(key, engineJob);
            engineJob.addCallback(cb);
            engineJob.start(runnable);

            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Started new load", startTime, key);
            }
            return new LoadStatus(cb, engineJob);
        }

        ...
}
```

这次把目光放在第 18 行上面，看到了吗？就是在这里调用的 EngineJob 的 addCallback()方法来注册的一个 ResourceCallback。那么接下来的问题就是，Engine.load()方法的 ResourceCallback 参数又是谁传过来的呢？这就需要回到 GenericRequest 的 onSizeReady()方法当中了，我们看到 ResourceCallback 是 load()方法的最后一个参数，那么在 onSizeReady()方法中调用 load()方法时传入的最后一个参数是什么？代码如下所示：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback,
            ResourceCallback {

        ...

        @Override
        public void onSizeReady(int width, int height) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));
            }
            if (status != Status.WAITING_FOR_SIZE) {
                return;
            }
            status = Status.RUNNING;
            width = Math.round(sizeMultiplier * width);
            height = Math.round(sizeMultiplier * height);
            ModelLoader<A, T> modelLoader = loadProvider.getModelLoader();
            final DataFetcher<T> dataFetcher = modelLoader.getResourceFetcher(model, width,
height);
            if (dataFetcher == null) {
                onException(new Exception("Failed to load model: \'" + model + "\'"));
```

```
                return;
            }
            ResourceTranscoder<Z, R> transcoder = loadProvider.getTranscoder();
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTime));
            }
            loadedFromMemoryCache = true;
            loadStatus = engine.load(signature, width, height, dataFetcher, loadProvider,
transformation,
                    transcoder, priority, isMemoryCacheable, diskCacheStrategy, this);
            loadedFromMemoryCache = resource != null;
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));
            }
        }

        ...
}
```

请将目光锁定在第 29 行的最后一个参数，this。没错，就是 this。GenericRequest 本身就实现了 ResourceCallback 的接口，因此 EngineJob 的回调最终其实就是回调到了 GenericRequest 的 onResourceReady()方法当中了，代码如下所示：

```
public void onResourceReady(Resource<?> resource) {
    if (resource == null) {
        onException(new Exception("Expected to receive a Resource<R> with an object of " +
transcodeClass
                + " inside, but instead got null."));
        return;
    }
    Object received = resource.get();
    if (received == null || !transcodeClass.isAssignableFrom(received.getClass())) {
        releaseResource(resource);
        onException(new Exception("Expected to receive an object of " + transcodeClass
                + " but instead got " + (received != null ? received.getClass() : "") + "{" +
received + "}"
                + " inside Resource{" + resource + "}."
                + (received != null ? "" : " "
                    + "To indicate failure return a null Resource object, "
                    + "rather than a Resource object containing null data.")
        ));
        return;
    }
    if (!canSetResource()) {
        releaseResource(resource);
```

```
            // We can't set the status to complete before asking canSetResource().
            status = Status.COMPLETE;
            return;
        }
        onResourceReady(resource, (R) received);
    }

    private void onResourceReady(Resource<?> resource, R result) {
        // We must call isFirstReadyResource before setting status.
        boolean isFirstResource = isFirstReadyResource();
        status = Status.COMPLETE;
        this.resource = resource;
        if (requestListener == null || !requestListener.onResourceReady(result, model, target,
loadedFromMemoryCache,
                isFirstResource)) {
            GlideAnimation<R> animation = animationFactory.build(loadedFromMemoryCache,
isFirstResource);
            target.onResourceReady(result, animation);
        }
        notifyLoadSuccess();
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logV("Resource ready in " + LogTime.getElapsedMillis(startTime) + " size: "
                    + (resource.getSize() * TO_MEGABYTE) + " fromCache: " +
loadedFromMemoryCache);
        }
    }
```

这里有两个 onResourceReady()方法，首先在第一个 onResourceReady()方法当中，调用 resource.get()方法获取到了封装的图片对象，也就是 GlideBitmapDrawable 对象，或者是 GifDrawable 对象。然后将这个值传入到了第二个 onResourceReady()方法当中，并在第 36 行调用了 target.onResourceReady()方法。

那么这个 target 又是什么呢？这个又需要向上翻很久了，在第三步 into()方法的一开始，我们就分析了在 into()方法的最后一行，调用了 glide.buildImageViewTarget()方法来构建出一个 Target，而这个 Target 就是一个 GlideDrawableImageViewTarget 对象。

那么我们去看 GlideDrawableImageViewTarget 的源码就可以了，如下所示：

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideDrawable> {
    private static final float SQUARE_RATIO_MARGIN = 0.05f;
    private int maxLoopCount;
    private GlideDrawable resource;

    public GlideDrawableImageViewTarget(ImageView view) {
        this(view, GlideDrawable.LOOP_FOREVER);
```

```
    }

    public GlideDrawableImageViewTarget(ImageView view, int maxLoopCount) {
        super(view);
        this.maxLoopCount = maxLoopCount;
    }

    @Override
    public void onResourceReady(GlideDrawable resource, GlideAnimation<? super
GlideDrawable> animation) {
        if (!resource.isAnimated()) {
            float viewRatio = view.getWidth() / (float) view.getHeight();
            float drawableRatio = resource.getIntrinsicWidth() / (float)
resource.getIntrinsicHeight();
            if (Math.abs(viewRatio - 1f) <= SQUARE_RATIO_MARGIN
                    && Math.abs(drawableRatio - 1f) <= SQUARE_RATIO_MARGIN) {
                resource = new SquaringDrawable(resource, view.getWidth());
            }
        }
        super.onResourceReady(resource, animation);
        this.resource = resource;
        resource.setLoopCount(maxLoopCount);
        resource.start();
    }

    @Override
    protected void setResource(GlideDrawable resource) {
        view.setImageDrawable(resource);
    }

    @Override
    public void onStart() {
        if (resource != null) {
            resource.start();
        }
    }

    @Override
    public void onStop() {
        if (resource != null) {
            resource.stop();
        }
    }
}
```

在 GlideDrawableImageViewTarget 的 onResourceReady()方法中做了一些逻辑处理，包括如果是 GIF 图片的话，就调用 resource.start()方法开始播放图片，但是好像并没有看到哪里有将 GlideDrawable 显示到 ImageView 上的逻辑。

确实没有，不过父类里面有，这里在第 25 行调用了 super.onResourceReady()方法，GlideDrawableImageViewTarget 的父类是 ImageViewTarget，我们来看下它的代码吧：

```
public abstract class ImageViewTarget<Z> extends ViewTarget<ImageView, Z> implements GlideAnimation.ViewAdapter {

    ...

    @Override
    public void onResourceReady(Z resource, GlideAnimation<? super Z> glideAnimation) {
        if (glideAnimation == null || !glideAnimation.animate(resource, this)) {
            setResource(resource);
        }
    }

    protected abstract void setResource(Z resource);

}
```

可以看到，在 ImageViewTarget 的 onResourceReady()方法当中调用了 setResource()方法，而 ImageViewTarget 的 setResource()方法是一个抽象方法，具体的实现还是在子类那边实现的。

那子类的 setResource()方法是怎么实现的呢？回头再来看一下 GlideDrawableImageViewTarget 的 setResource()方法，没错，调用的 view.setImageDrawable()方法，而这个 view 就是 ImageView。代码执行到这里，图片终于也就显示出来了。

那么，我们对 Glide 执行流程的源码分析，到这里也终于结束了。

一、EventBus

1.1、EventBus

EventBus 是一个 Android 事件发布/订阅框架，通过解耦发布者和订阅者简化 Android 事件传递，这里的事件可以理解为消息，本文中统一称为事件。事件传递既可用于 Android 四大组件间通讯，也可以用户异步线程和主线程间通讯等等。

传统的事件传递方式包括：Intent、Handler、BroadCastReceiver、Interface 回调，相比之下 EventBus 的优点是代码简洁，使用简单，并将事件发布和订阅充分解耦。可简化 Activities, Fragments, Threads, Services 等组件间的消息传递，可替代 Intent、

Handler、BroadCast、接口等传统方案，更快，代码更小，50K 左右的 jar 包，代码更优雅，彻底解耦。

1.2、概念

事件(Event)：又可称为消息，本文中统一用事件表示。其实就是一个对象，可以是网络请求返回的字符串，也可以是某个开关状态等等。事件类型(EventType)指事件所属的 Class。

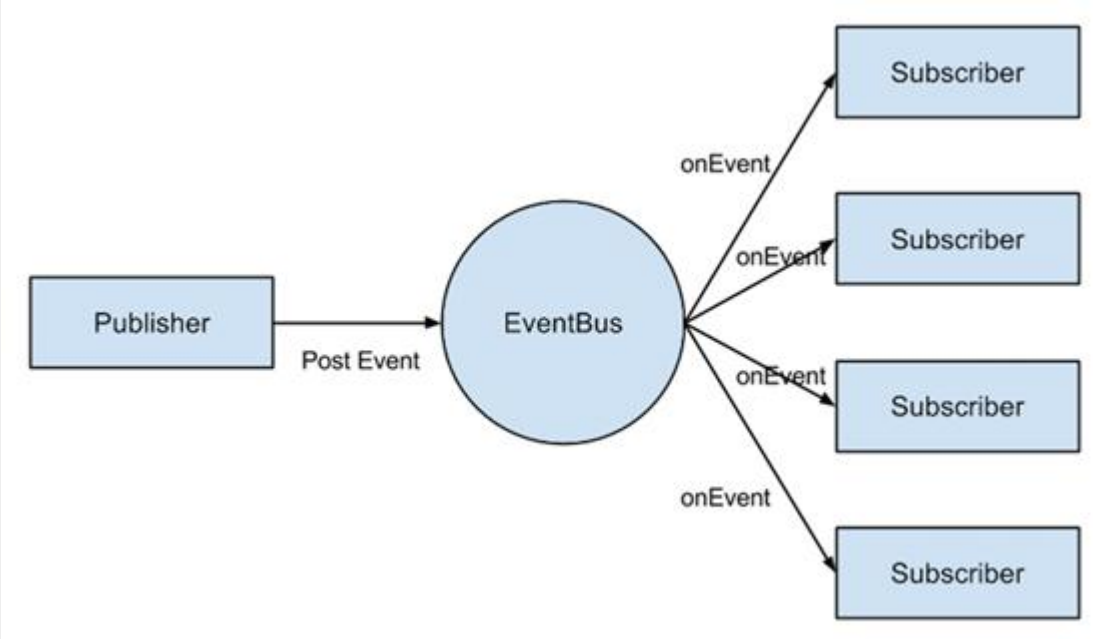
事件分为一般事件和 Sticky 事件，相对于一般事件，Sticky 事件不同之处在于，当事件发布后，再有订阅者开始订阅该类型事件，依然能收到该类型事件最近一个 Sticky 事件。

订阅者(Subscriber)：订阅某种事件类型的对象。当有发布者发布这类事件后，EventBus 会执行订阅者的 onEvent 函数，这个函数叫事件响应函数。订阅者通过 register 接口订阅某个事件类型，unregister 接口退订。订阅者存在优先级，优先级高的订阅者可以取消事件继续向优先级低的订阅者分发，默认所有订阅者优先级都为 0。

发布者(Publisher)：发布某事件的对象，通过 post 接口发布事件。

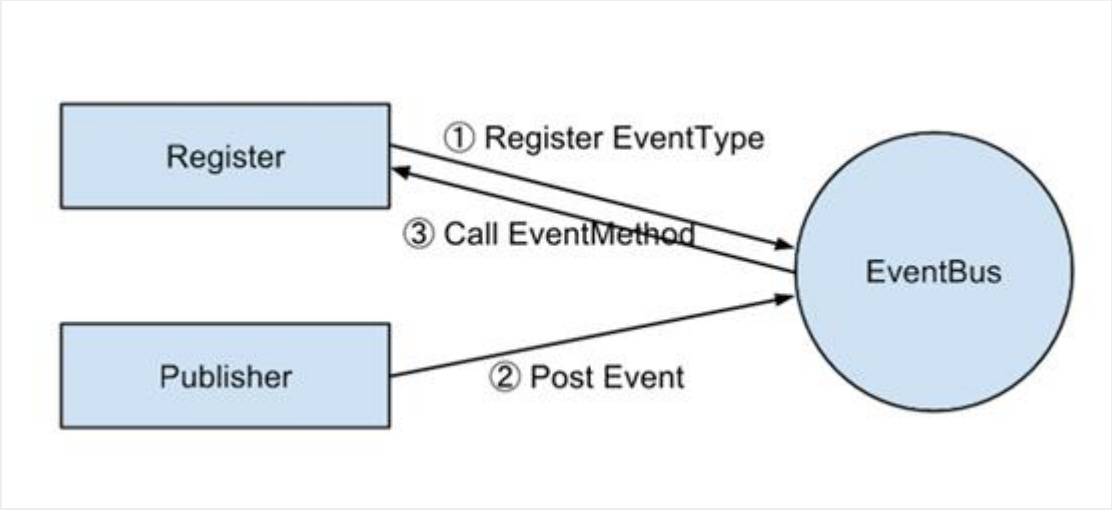
1.3、订阅者、发布者、EventBus 关系图

EventBus 负责存储订阅者、事件相关信息，订阅者和发布者都只和 EventBus 关联。



事件响应流程

订阅者首先调用 EventBus 的 register 接口订阅某种类型的事件，当发布者通过 post 接口发布该类型的事件时，EventBus 执行调用者的事件响应函数。



二、EventBus 优势

2.1、对比 Java 监听器接口（Listener Interfaces）

在 Java 中，特别是 Android，一个常用的模式就是使用"监听器（Listeners）"接口。在此模式中，一个实现了监听器接口的类必须将自身注册到它想要监听的类中去。这就意味着监听与被监听之间属于强关联关系。这种关系就使得单元测试很难进行开展。

2.2、对比本地广播管理器（LocalBroadcastManager）

另一项技术就是在组件间通过本地广播管理器（LocalBroadcastManager）进行消息的发送与监听。虽然这对于解耦有很好的帮助，但它的 API 不如 EventBus 那样简洁。此外，如果你不注意保持 Intent extras 类型的一致，它还可能引发潜在的运行时/类型检测错误。

使用 EventBus 不仅使代码变得清晰，而且增强了类型安全（type-safe）。当用 Intent 传递数据时，在编译时并不能检查出所设的 extra 类型与收到时的类型一致。所以一个很常见的错误便是你或者你团队中的其他人改变了 Intent 所传递的数据，但忘记了对全部的接收器（receiver）进行更新。这种错误在编译时是无法被发现的，只有在运行时才会发现问题。

而使用 EventBus 所传递的消息则是通过你所定义的 Event 类。由于接收者方法是直接与这些类实例打交道，所以所有的数据均可以进行类型检查，这样任何由于类型不一致所导致的错误都可以在编译时刻被发现。

另外就是你的 Event 类可以定义成任何类型。通常会为了表示事件而显式地创建明确命名的类，你也通过 EventBus 发送/接收任何类。通过这种方法，你就不必受限于那些只能添加到 Intent extras 中的简单数据类型了。例如，你可以发送一个和 ORM 模型类实例，并且在接收端直接处理与 ORM 操作相关的类实例。

三、EventBus 使用

3.1 基本使用

（1）自定义一个类，可以是空类，比如：

```java
public class AnyEventType {
    public AnyEventType(){}
 }
```

（2）在要接收消息的页面注册：

```java
eventBus.register(this);
```

（3）发送消息

```java
eventBus.post(new AnyEventType event);
```

（4）接受消息的页面实现(共有四个函数，各功能不同，这是其中之一，可以选择性的实现，这里先实现一个)：

```java
public void onEvent(AnyEventType event) {}
```

（5）解除注册

```java
eventBus.unregister(this);
```

3.2 具体案例

```java
package com.example.tryeventbus_simple;

import com.harvic.other.FirstEvent;

import de.greenrobot.event.EventBus;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class SecondActivity extends Activity {
    private Button btn_FirstEvent;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        btn_FirstEvent = (Button) findViewById(R.id.btn_first_event);

        btn_FirstEvent.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                EventBus.getDefault().post(
                        new FirstEvent("FirstEvent btn clicked"));
            }
        });
    }
}
```

```
15. public class MainActivity extends Activity {
16. 
17.     Button btn;
18.     TextView tv;
19. 
20.     @Override
21.     protected void onCreate(Bundle savedInstanceState) {
22.         super.onCreate(savedInstanceState);
23.         setContentView(R.layout.activity_main);
24. 
25.         EventBus.getDefault().register(this);
26. 
27.         btn = (Button) findViewById(R.id.btn_try);
28.         tv = (TextView)findViewById(R.id.tv);
29. 
30.         btn.setOnClickListener(new View.OnClickListener() {
31. 
32.             @Override
33.             public void onClick(View v) {
34.                 // TODO Auto-generated method stub
35.                 Intent intent = new Intent(getApplicationContext(),
36.                         SecondActivity.class);
37.                 startActivity(intent);
38.             }
39.         });
40.     }
41. 
42.     public void onEventMainThread(FirstEvent event) {
43. 
44.         String msg = "onEventMainThread收到了消息: " + event.getMsg();
45.         Log.d("harvic", msg);
46.         tv.setText(msg);
47.         Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
48.     }
49. 
50.     @Override
51.     protected void onDestroy(){
52.         super.onDestroy();
53.         EventBus.getDefault().unregister(this);
```

```
01. package com.harvic.other;
02. 
03. public class FirstEvent {
04. 
05.     private String mMsg;
06.     public FirstEvent(String msg) {
07.         // TODO Auto-generated constructor stub
08.         mMsg = msg;
09.     }
10.     public String getMsg(){
11.         return mMsg;
12.     }
13. }
```

3.3 onEvent 函数使用解析

前一篇给大家装简单演示了 EventBus 的 onEventMainThread（）函数的接收，其实 EventBus 还有另外有个不同的函数，他们分别是：

1、onEvent
2、onEventMainThread
3、onEventBackgroundThread
4、onEventAsync

这四种订阅函数都是使用 onEvent 开头的，它们的功能稍有不同,在介绍不同之前先介绍两个概念：告知观察者事件发生时通过 EventBus.post 函数实现，这个过程叫做事件的发布，观察者被告知事件发生叫做事件的接收，是通过下面的订阅函数实现的。
onEvent:如果使用 onEvent 作为订阅函数，那么该事件在哪个线程发布出来的，onEvent 就会在这个线程中运行，也就是说发布事件和接收事件线程在同一个线程。使用这个方法时，在 onEvent 方法中不能执行耗时操作，如果执行耗时操作容易导致事件分发延迟。
onEventMainThread:如果使用 onEventMainThread 作为订阅函数，那么不论事件是在哪个线程中发布出来的，onEventMainThread 都会在 UI 线程中执行，接收事件就会在 UI 线程中运行，这个在 Android 中是非常有用的，因为在 Android 中只能在 UI 线程中更新 UI，所以在 onEvnetMainThread 方法中是不能执行耗时操作的。
onEventBackground:如果使用 onEventBackgrond 作为订阅函数，那么如果事件是在 UI 线程中发布出来的，那么 onEventBackground 就会在子线程中运行，如果事件本来就是子线程中发布出来的，那么 onEventBackground 函数直接在该子线程中执行。
onEventAsync：使用这个函数作为订阅函数，那么无论事件在哪个线程发布，都会创建新的子线程在执行 onEventAsync.

3.4 EvetntBus3.0 使用解析

注册一般是在 onCreate 和 onStart 里注册，尽量不要在 onResume，可能出现多次注册的情况，比如下面这个异常：

可以先判断下：

```
1    if (!EventBus.getDefault().isRegistered(this)) {
2        EventBus.getDefault().register(this);
3    }
```

取消注册 要写到 onDestroy 方法里，不要写到 onStop 里，有时会出现异常的哦

EventBus 3 和之前版本的 EventBus 不兼容，这里采用注解的方法来接收事件，四种注解 @Subscrible、@Subscrible(threadMode = ThreadMode.ASYNC)、@Subscribe(threadMode = ThreadMode.BACKGROUND)、

@Subscribe(threadMode = ThreadMode.MAIN)分别对应之前的 onEvent()、onEventAsync()、onEventBackground()、onEventMainThread()。

EventBus 3 采用注解后，方法名没有限制了，参数只有一个，和发送者 post 的参数对应配对，在未声明 threadMode 时，默认的线程模式为 ThreadMode.POSTING，只有在该模式下才可以取消线程。

由于可在任何地方都可以 post 一个事件，那么在不同线程之间传递事件，比如在工作线程传递一个事件更新 UI 线程中的一个控件，则需要注意 threadMode 的切换。

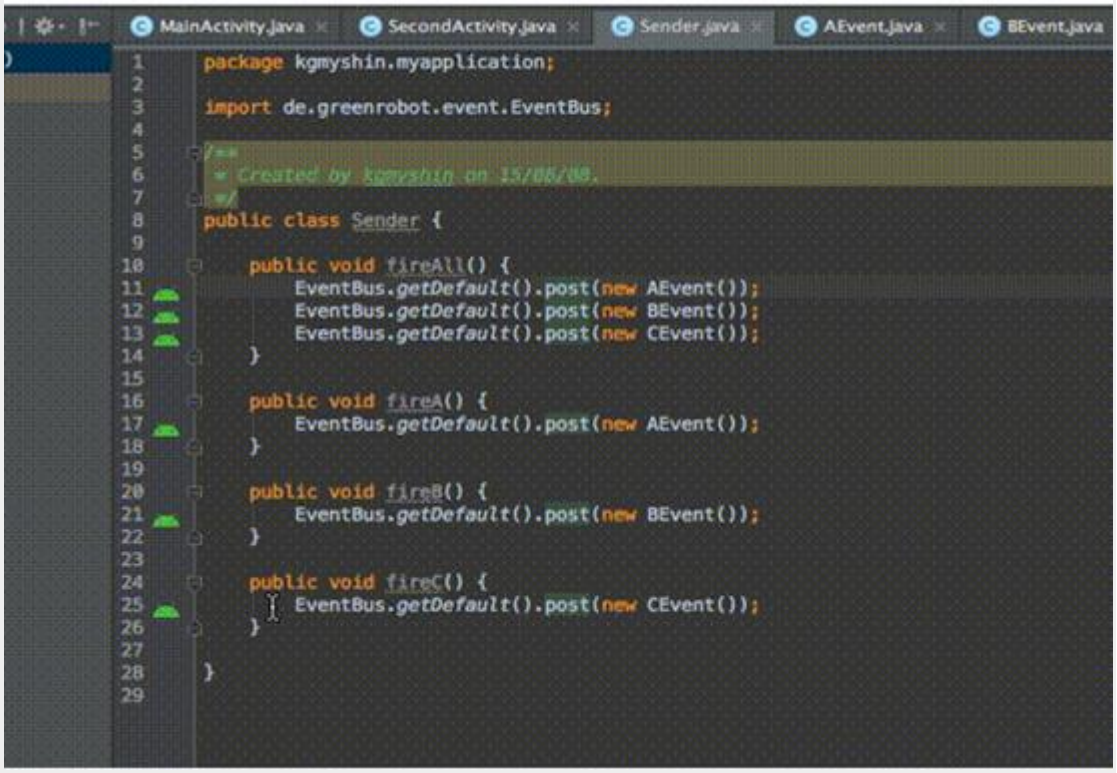
如果遇到订阅事件无法执行的情况，分析后发现是订阅事件的 Activity 还未执行的原因。找到原因就好办了，这时候就需要用到 postSticky。

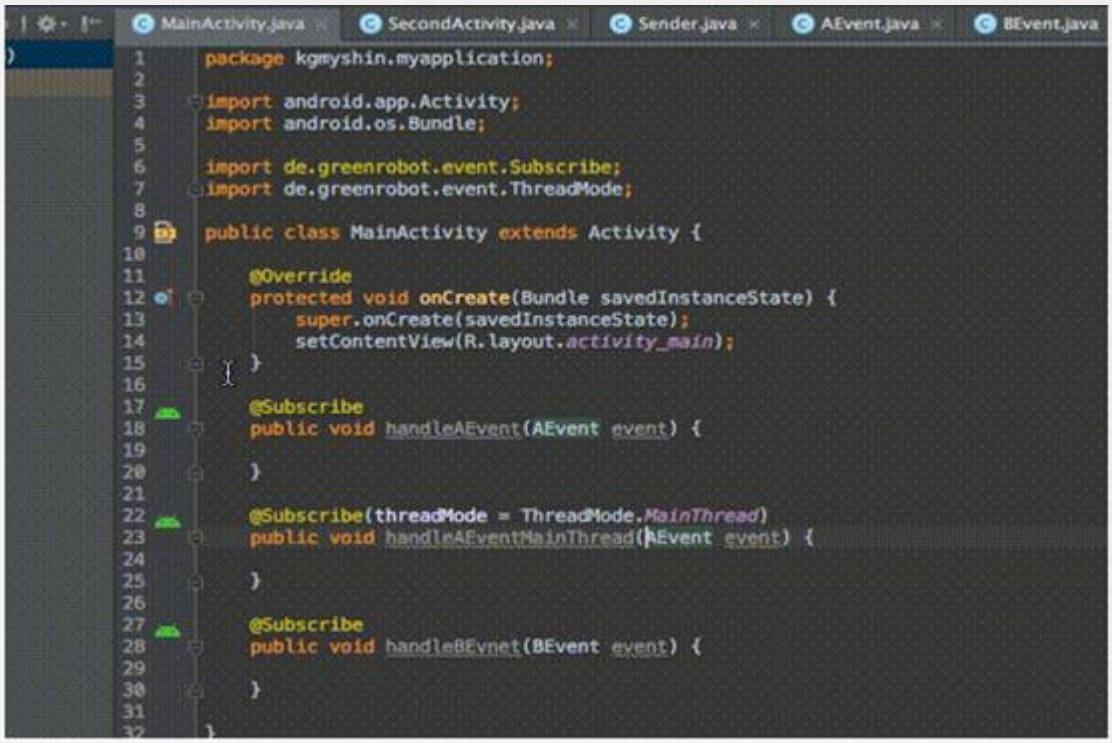
发布事件时用 postSticky 操作：

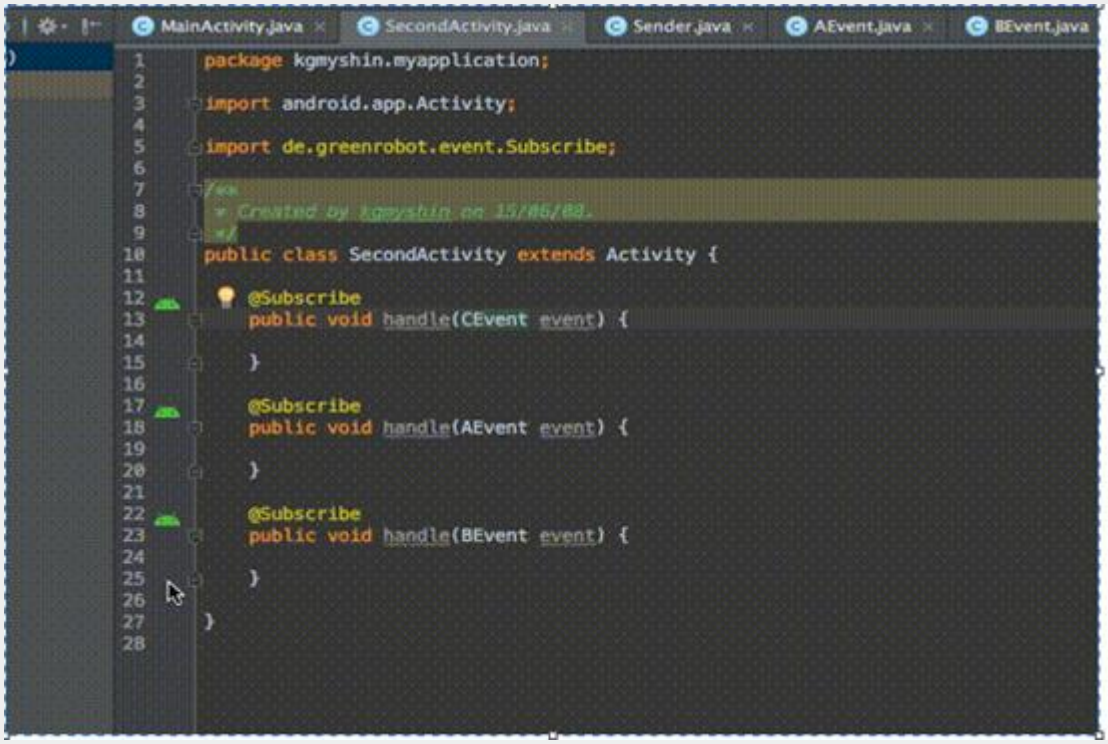
```
EventBus.getDefault().postSticky(event);
```

订阅时，添加 sticky = true

```
@Subscribe(sticky = true)     //看下 `@Subscribe` 源码知道 `sticky` 默认是 `false`
public void onEvent(Event e) {
    ---
}
```







四、EventBus 源码解析

4.1 register

EventBus.getDefault().register(this); EventBus.getDefault()其实就是个单例，和我们传统的 getInstance 一个意思：

```
/** Convenience singleton for apps using a process-wide EventBus instance. */
    public static EventBus getDefault() {
        if (defaultInstance == null) {
            synchronized (EventBus.class) {
                if (defaultInstance == null) {
                    defaultInstance = new EventBus();
                }
            }
        }
        return defaultInstance;
    }
```

使用了双重判断的方式，防止并发的问题，还能极大的提高效率。

register 公布给我们使用的有 4 个：

```
  public void register(Object subscriber) {
          register(subscriber, DEFAULT_METHOD_NAME, false, 0);
      }
   public void register(Object subscriber, int priority) {
          register(subscriber, DEFAULT_METHOD_NAME, false, priority);
      }
  public void registerSticky(Object subscriber) {
          register(subscriber, DEFAULT_METHOD_NAME, true, 0);
      }
  public void registerSticky(Object subscriber, int priority) {
          register(subscriber, DEFAULT_METHOD_NAME, true, priority);
      }
```

```
private synchronized void register(Object subscriber, String methodName, boolean sticky, int priority) {
        List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriber.getClass(),
                methodName);
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            subscribe(subscriber, subscriberMethod, sticky, priority);
        }
    }
```

调用内部类 SubscriberMethodFinder 的 findSubscriberMethods 方法，传入了 subscriber 的 class，以及 methodName，返回一个 List<SubscriberMethod>。

那么不用说，肯定是去遍历该类内部所有方法，然后根据 methodName 去匹配，匹配成功的封装成 SubscriberMethod，最后返回一个 List。

```
01. List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass, String eventMethodName) {
02.         String key = subscriberClass.getName() + '.' + eventMethodName;
03.         List<SubscriberMethod> subscriberMethods;
04.         synchronized (methodCache) {
05.             subscriberMethods = methodCache.get(key);
06.         }
07.         if (subscriberMethods != null) {
08.             return subscriberMethods;
09.         }
10.         subscriberMethods = new ArrayList<SubscriberMethod>();
11.         Class<?> clazz = subscriberClass;
12.         HashSet<String> eventTypesFound = new HashSet<String>();
13.         StringBuilder methodKeyBuilder = new StringBuilder();
14.         while (clazz != null) {
15.             String name = clazz.getName();
16.             if (name.startsWith("java.") || name.startsWith("javax.") || name.startsWith("android.")) {
17.                 // Skip system classes, this just degrades performance
18.                 break;
19.             }
20.
21.             // Starting with EventBus 2.2 we enforc
22.             Method[] methods = clazz.getMethods();
23.             for (Method method : methods) {
24.                 String methodName = method.getName();
25.                 if (methodName.startsWith(eventMethodName)) {
26.                     int modifiers = method.getModifiers();
27.                     if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_IGNORE) == 0) {
28.                         Class<?>[] parameterTypes = method.getParameterTypes();
29.                         if (parameterTypes.length == 1) {
30.                             String modifierString = methodName.substring(eventMethodName.length());
31.                             ThreadMode threadMode;
32.                             if (modifierString.length() == 0) {
33.                                 threadMode = ThreadMode.PostThread;
34.                             } else if (modifierString.equals("MainThread")) {
35.                                 threadMode = ThreadMode.MainThread;
36.                             } else if (modifierString.equals("BackgroundThread")) {
37.                                 threadMode = ThreadMode.BackgroundThread;
38.                             } else if (modifierString.equals("Async")) {
```

22行：看到没，clazz.getMethods();去得到所有的方法

25-29行：分别判断了是否以onEvent开头，是否是public且非static和abstract方法，是否是一个参数。如果都复合，才进入封装的部分

32-45行：也比较简单，根据方法的后缀，来确定threadMode，threadMode是个枚举类型：就四种情况

```
33.                             threadMode = ThreadMode.PostThread;
34.                         } else if (modifierString.equals("MainThread")) {
35.                             threadMode = ThreadMode.MainThread;
36.                         } else if (modifierString.equals("BackgroundThread")) {
37.                             threadMode = ThreadMode.BackgroundThread;
38.                         } else if (modifierString.equals("Async")) {
39.                             threadMode = ThreadMode.Async;
40.                         } else {
41.                             if (skipMethodVerificationForClasses.containsKey(clazz)) {
42.                                 continue;
43.                             } else {
44.                                 throw new EventBusException("Illegal onEvent method, check for typos: " + method);
45.                             }
46.                         }
47.                         Class<?> eventType = parameterTypes[0];
48.                         methodKeyBuilder.setLength(0);
49.                         methodKeyBuilder.append(methodName);
50.                         methodKeyBuilder.append('>').append(eventType.getName());
51.                         String methodKey = methodKeyBuilder.toString();
52.                         if (eventTypesFound.add(methodKey)) {
53.                             // Only add if not already found in a sub class
54.                             subscriberMethods.add(new SubscriberMethod(method, threadMode, eventType));
55.                         }
56.                     }
57.                 } else if (!skipMethodVerificationForClasses.containsKey(clazz)) {
58.                     Log.d(EventBus.TAG, "Skipping method (not public, static or abstract): " + clazz + "."
59.                             + methodName);
60.                 }
61.             }
62.         }
63.         clazz = clazz.getSuperclass();
64.     }
65.     if (subscriberMethods.isEmpty()) {
66.         throw new EventBusException("Subscriber " + subscriberClass + " has no public methods called "
67.                 + eventMethodName);
68.     } else {
69.         synchronized (methodCache) {
70.             methodCache.put(key, subscriberMethods);
71.         }
72.         return subscriberMethods;
```

54行 将method, threadMode, eventType传入构造了：new SubscriberMethod(method, threadMode, eventType)，添加到List，最终放回

63行：clazz = clazz.getSuperclass();
可以看到，会扫描所有的父类，不仅仅是当前类

继续回到 register:

```
01. for (SubscriberMethod subscriberMethod : subscriberMethods) {
02.             subscribe(subscriber, subscriberMethod, sticky, priority);
03.         }
```

```
01.  // Must be called in synchronized block
02.      private void subscribe(Object subscriber, SubscriberMethod subscriberMethod, boolean sticky, int priority) {
03.          subscribed = true;
04.          Class<?> eventType = subscriberMethod.eventType;
05.          CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
06.          Subscription newSubscription = new Subscription(subscriber, subscriberMethod, priority);
07.          if (subscriptions == null) {
08.              subscriptions = new CopyOnWriteArrayList<Subscription>();
09.              subscriptionsByEventType.put(eventType, subscriptions);
10.          } else {
11.              for (Subscription subscription : subscriptions) {
12.                  if (subscription.equals(newSubscription)) {
13.                      throw new EventBusException("Subscriber " + subscriber.getClass() + " already registered to event "
14.                              + eventType);
15.                  }
16.              }
17.          }
18.
19.          // Starting with EventBus 2.2 we enforced methods to be public (might change with annotations again)
20.          // subscriberMethod.method.setAccessible(true);
21.
22.          int size = subscriptions.size();
23.          for (int i = 0; i <= size; i++) {
24.              if (i == size || newSubscription.priority > subscriptions.get(i).priority) {
25.                  subscriptions.add(i, newSubscription);
26.                  break;
27.              }
28.          }
29.
30.          List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
31.          if (subscribedEvents == null) {
32.              subscribedEvents = new ArrayList<Class<?>>();
33.              typesBySubscriber.put(subscriber, subscribedEvents);
34.          }
35.          subscribedEvents.add(eventType);
36.
```

4-17. 这里的subscriptionsByEventType是个Map，key：eventType ； value：CopyOnWriteArrayList<Subscription> ；这个Map其实就是EventBus存储方法的地方，一定要记住

22-28行：实际上，就是添加newSubscription；并且是按照优先级添加的。可以看到，优先级越高，会插到在当前List的前面

30-35. 根据subscriber存储它所有的eventType ； 依然是map；key：subscriber，value：List<eventType> ;知道就行，非核心代码，主要用于isRegister的判断

```
37.          if (sticky) {
38.              Object stickyEvent;
39.              synchronized (stickyEvents) {
40.                  stickyEvent = stickyEvents.get(eventType);
41.              }
42.              if (stickyEvent != null) {
43.                  // If the subscriber is trying to abort the event, it will fail (event is not tracked in posting state)
44.                  // --> Strange corner case, which we don't take care of here.
45.                  postToSubscription(newSubscription, stickyEvent, Looper.getMainLooper() == Looper.myLooper());
46.              }
47.          }
48.      }
```

37-47 判断sticky；如果为true，从stickyEvents中根据eventType去查找有没有stickyEvent，如果有则立即发布去执行。stickyEvent其实就是我们post时的参数

到此，我们 register 就介绍完了。

你只要记得一件事：扫描了所有的方法，把匹配的方法最终保存在 subscriptionsByEventType（Map，key：eventType ； value：CopyOnWriteArrayList<Subscription> ）中；

eventType 是我们方法参数的 Class，Subscription 中则保存着 subscriber, subscriberMethod（method, threadMode, eventType）, priority；包含了执行改方法所需的一切。

4.2 post

```
01. /** Posts the given event to the event bus. */
02.    public void post(Object event) {
03.        PostingThreadState postingState = currentPostingThreadState.get();
04.        List<Object> eventQueue = postingState.eventQueue;
05.        eventQueue.add(event);
06.
07.        if (postingState.isPosting) {
08.            return;
09.        } else {
10.            postingState.isMainThread = Looper.getMainLooper() == Looper.myLooper();
11.            postingState.isPosting = true;
12.            if (postingState.canceled) {
13.                throw new EventBusException("Internal error. Abort state was not reset");
14.            }
15.            try {
16.                while (!eventQueue.isEmpty()) {
17.                    postSingleEvent(eventQueue.remove(0), postingState);
18.                }
19.            } finally {
20.                postingState.isPosting = false;
21.                postingState.isMainThread = false;
22.            }
23.        }
24.    }
```

把我们传入的event，保存到了当前线程中的一个变量PostingThreadState的eventQueue中。

10行：判断当前是否是UI线程

16-18行：遍历队列中的所有的event，调用postSingleEvent(eventQueue.remove(0), postingState)方法。

这里大家会不会有疑问，每次post都会去调用整个队列么，那么不会造成方法多次调用么？

可以看到第7-8行，有个判断，就是防止该问题的，isPosting=true了，就不会往下走了

```
01. private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {
02.         Class<? extends Object> eventClass = event.getClass();
03.         List<Class<?>> eventTypes = findEventTypes(eventClass);
04.         boolean subscriptionFound = false;
05.         int countTypes = eventTypes.size();
06.         for (int h = 0; h < countTypes; h++) {
07.             Class<?> clazz = eventTypes.get(h);
08.             CopyOnWriteArrayList<Subscription> subscriptions;
09.             synchronized (this) {
10.                 subscriptions = subscriptionsByEventType.get(clazz);
11.             }
12.             if (subscriptions != null && !subscriptions.isEmpty()) {
13.                 for (Subscription subscription : subscriptions) {
14.                     postingState.event = event;
15.                     postingState.subscription = subscription;
16.                     boolean aborted = false;
17.                     try {
18.                         postToSubscription(subscription, event, postingState.isMainThread);
19.                         aborted = postingState.canceled;
20.                     } finally {
21.                         postingState.event = null;
22.                         postingState.subscription = null;
23.                         postingState.canceled = false;
24.                     }
25.                     if (aborted) {
26.                         break;
27.                     }
28.                 }
29.                 subscriptionFound = true;
30.             }
31.         }
32.         if (!subscriptionFound) {
33.             Log.d(TAG, "No subscribers registered for event " + eventClass);
34.             if (eventClass != NoSubscriberEvent.class && eventClass != SubscriberExceptionEvent.class) {
35.                 post(new NoSubscriberEvent(this, event));
```

2-3行：根据event的Class，去得到一个List<Class<?>>；其实就是得到event当前对象的Class，以及父类和接口的Class类型；主要用于匹配，比如你传入Dog extends Dog，他会把Animal也装到该List中

6-31行：遍历所有的Class，到subscriptionsByEventType去查找subscriptions

```java
private void postToSubscription(Subscription subscription, Object event, boolean isMainThread) {
        switch (subscription.subscriberMethod.threadMode) {
        case PostThread:
            invokeSubscriber(subscription, event);
            break;
        case MainThread:
            if (isMainThread) {
                invokeSubscriber(subscription, event);
            } else {
                mainThreadPoster.enqueue(subscription, event);
            }
            break;
        case BackgroundThread:
            if (isMainThread) {
                backgroundPoster.enqueue(subscription, event);
            } else {
                invokeSubscriber(subscription, event);
            }
            break;
        case Async:
            asyncPoster.enqueue(subscription, event);
            break;
        default:
            throw new IllegalStateException("Unknown thread mode: " + subscription.subscriberMethod.threadMode);
        }
    }
```

```java
void invokeSubscriber(Subscription subscription, Object event) throws Error {
          subscription.subscriberMethod.method.invoke(subscription.subscriber, event);
}
```

到此，我们完整的源码分析就结束了，总结一下：**register** 会把当前类中匹配的方法，存入一个 **map**，而 **post** 会根据实参去 **map** 查找进行反射调用。分析这么久，一句话就说完了**~~**

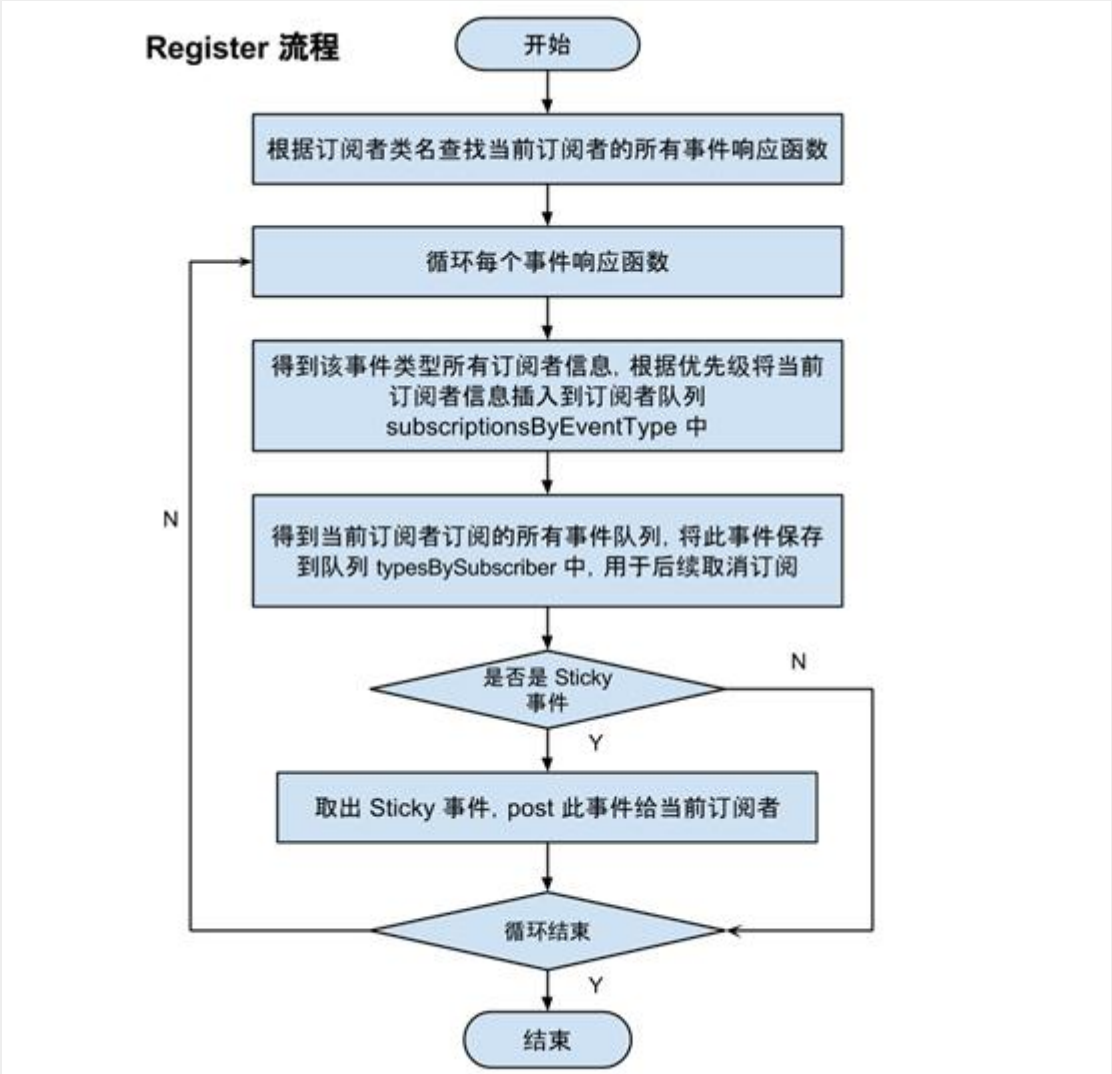
其实不用发布者，订阅者，事件，总线这几个词或许更好理解，以后大家问了 **EventBus**，可以说，就是在一个单例内部维持着一个 **map** 对象存储了一堆的方法；**post** 无非就是根据参数去查找方法，进行反射调用。

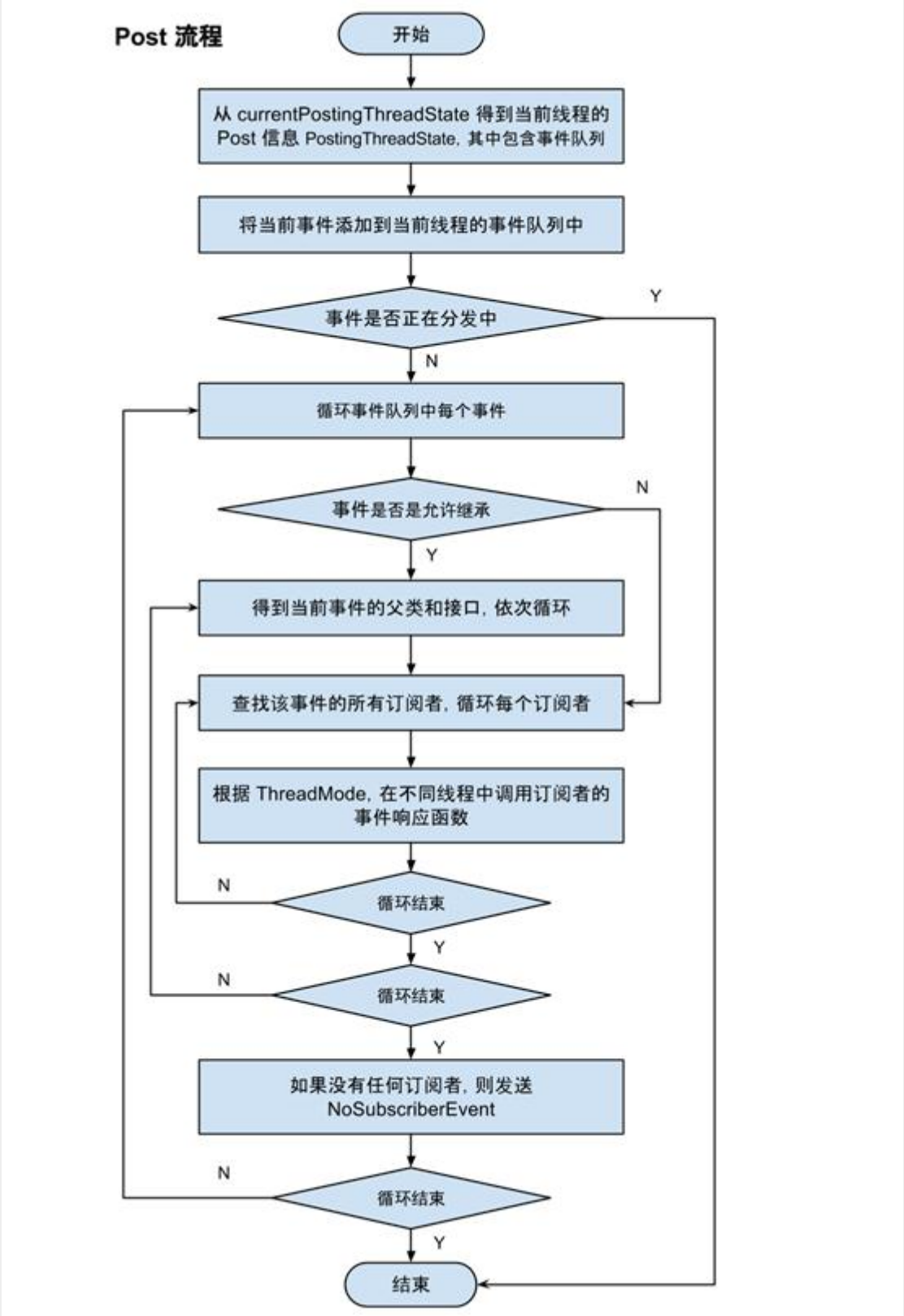
介绍了register和post；大家获取还能想到一个词sticky，在register中，如何sticky为true，会去stickyEvents去查找事件，然后立即去post；

那么这个stickyEvents何时进行保存事件呢？

其实evevntbus中，除了post发布事件，还有一个方法也可以：

```java
public void postSticky(Object event) {
        synchronized (stickyEvents) {
            stickyEvents.put(event.getClass(), event);
        }
        // Should be posted after it is putted, in case the subscriber wants to remove immediately
        post(event);
    }
```


Register 流程
开始
根据订阅者类名查找当前订阅者的所有事件响应函数
循环每个事件响应函数
得到该事件类型所有订阅者信息，根据优先级将当前订阅者信息插入到订阅者队列 subscriptionsByEventType 中
得到当前订阅者订阅的所有事件队列，将此事件保存到队列 typesBySubscriber 中，用于后续取消订阅
是否是 Sticky 事件
N
Y
取出 Sticky 事件，post 此事件给当前订阅者
循环结束
N
Y
结束


Post 流程
开始
从 currentPostingThreadState 得到当前线程的 Post 信息 PostingThreadState，其中包含事件队列
将当前事件添加到当前线程的事件队列中
事件是否正在分发中
Y
N
循环事件队列中每个事件
事件是否是允许继承
N
Y
得到当前事件的父类和接口，依次循环
查找该事件的所有订阅者，循环每个订阅者
根据 ThreadMode，在不同线程中调用订阅者的事件响应函数
N
循环结束
Y
N
循环结束
Y
如果没有任何订阅者，则发送 NoSubscriberEvent
N
循环结束
Y
结束

2. LeakCanary

「Leakcanary 」是我们经常用于检测内存泄漏的工具，简单的使用方式，内存泄漏的可视化，是我们开发中必备的工具之一。

一、使用

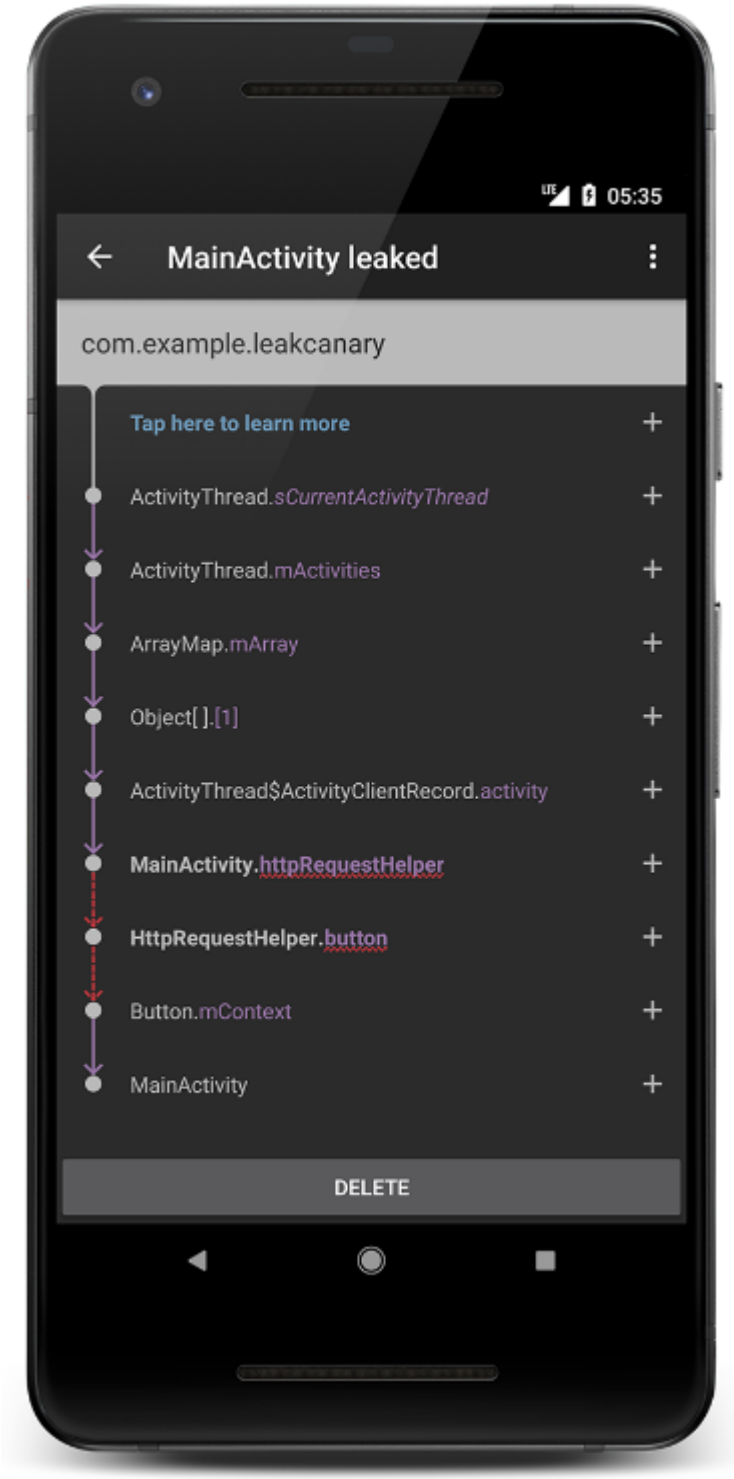
1、配置

```
dependencies {
  debugImplementation
'com.squareup.leakcanary:leakcanary-android:1.6.3'
  releaseImplementation
'com.squareup.leakcanary:leakcanary-android-no-op:1.6.3'
  // Optional, if you use support library fragments:
  debugImplementation
'com.squareup.leakcanary:leakcanary-support-fragment:1.6.3'}
```

2、简单使用

```
public class ExampleApplication extends Application {

  @Override public void onCreate() {
    super.onCreate();
    if (LeakCanary.isInAnalyzerProcess(this)) {
      // This process is dedicated to LeakCanary for heap analysis.
      // You should not init your app in this process.
      return;
    }
    LeakCanary.install(this);
    // Normal app init code...
  }}
```

超级简单的配置和使用方式。最后就会得出以下的事例说明。



二、准备工作

1、Reference

Reference 把内存分为 4 种状态，Active 、 Pending 、 Enqueued 、 Inactive。

- Active 一般说来内存一开始被分配的状态都是 Active
- Pending 快要放入队列（ReferenceQueue）的对象，也就是马上要回收的对象
- Enqueued 对象已经进入队列，已经被回收的对象。方便我们查询某个对象是否被回收
- Inactive 最终的状态，无法变成其他的状态。

2、ReferenceQueue

引用队列，在 Reference 被回收的时候，Reference 会被添加到 ReferenceQueue 中

3、如果检测一个对象是否被回收

需要采用 Reference + ReferenceQueue

- 创建一个引用队列 queue
- 创建 Reference 对象（通常用弱引用）并关联引用队列
- 在 Reference 被回收的时候，Reference 会被添加到 queue 中

```
//创建一个引用队列  ReferenceQueue queue = new ReferenceQueue();
// 创建弱引用，此时状态为 Active，并且 Reference.pending 为空，// 当前 Reference.queue = 上面创建的 queue，并且 next=null  // reference 创建并关联 queueWeakReference reference = new WeakReference(new Object(), queue);
// 当 GC 执行后，由于是弱引用，所以回收该 object 对象，并且置于 pending 上，此时 reference 的状态为 PENDING  System.gc();
 // ReferenceHandler 从 pending 中取下该元素，并且将该元素放入到 queue 中，//此时 Reference 状态为 ENQUEUED，Reference.queue = ReferenceENQUEUED
 // 当从 queue 里面取出该元素，则变为 INACTIVE，Reference.queue = Reference.NULL  Reference reference1 = queue.remove();
```

在 Reference 类加载的时候，Java 虚拟机会会创建一个最大优先级的后台线程，这个线程的工作就是不断检测 pending 是否为 null，如果不为 null，那么就将它放到 ReferenceQueue。因为 pending 不为 null，就说明引用所指向的对象已经被 GC，变成了不也达。

4、ActivityLifecycleCallbacks

用于监听所有 Activity 生命周期的回调方法。

```
 private final Application.ActivityLifecycleCallbacks lifecycleCallbacks =
```

```
    new Application.ActivityLifecycleCallbacks() {
      @Override public void onActivityCreated(Activity activity, Bundle
savedInstanceState) {
      }

      @Override public void onActivityStarted(Activity activity) {
      }

      @Override public void onActivityResumed(Activity activity) {
      }

      @Override public void onActivityPaused(Activity activity) {
      }

      @Override public void onActivityStopped(Activity activity) {
      }

      @Override public void onActivitySaveInstanceState(Activity
activity, Bundle outState) {
      }

      @Override public void onActivityDestroyed(Activity activity) {
        ActivityRefWatcher.this.onActivityDestroyed(activity);
      }
    };
```

5、Heap Dump

Heap Dump 也叫堆转储文件，是一个 Java 进程在某个时间点上的内存快照。

三、原理说明

1、监听 Activity 的生命周期。
2、在 onDestory 的时候，创建对应的 Actitity 的 Refrence 和 相应的 RefrenceQueue，启动后台进程去检测。
3、一段时间后，从 RefrenceQueue 中读取，如果有这个 Actitity 的 Refrence，那么说明这个 Activity 的 Refrence 已经被回收，但是如果 RefrenceQueue 没有这个 Actitity 的 Refrence 那就说明出现了内存泄漏。
4、dump 出 hprof 文件，找到泄漏路径。

分析源码

程序的唯一入口 LeakCanary.*install*(this);

1、install

DisplayLeakService 这个类负责发起 Notification 以及将结果记录下来写在文件里面。以后每次启动 LeakAnalyzerActivity 就从这个文件里读取历史结果，并展示给我们。

```
public static RefWatcher install(Application application) {
  return install(application, DisplayLeakService.class);
}
public static RefWatcher install(Application application,
    Class<? extends AbstractAnalysisResultService>
listenerServiceClass) {
  //如果在主线程 那么返回一个无用的 RefWatcher  详解 1.1
  if (isInAnalyzerProcess(application)) {
    return RefWatcher.DISABLED;
  }
  //把 DisplayLeakActivity 设置为可用  用于显示 DisplayLeakActivity
就是我们看到的那个分析界面
  enableDisplayLeakActivity(application);
  // 详解 1.2
  HeapDump.Listener heapDumpListener =
      new ServiceHeapDumpListener(application, listenerServiceClass);
  //详解 2
  RefWatcher refWatcher = androidWatcher(application,
heapDumpListener);
  详解 3
  ActivityRefWatcher.installOnIcsPlus(application, refWatcher);
  return refWatcher;
}
```

1.1 isInAnalyzerProcess

因为 分析的进程是硬外一个独立进程 所以要判断是否是主进程，这个工作需要在 AnalyzerProcess 中进行。

```
public static boolean isInAnalyzerProcess(Context context) {
  return isInServiceProcess(context, HeapAnalyzerService.class);
}
```

把 App 的进程 和 这个 Service 进程进行对比 。

```
private static boolean isInServiceProcess(Context context,
    Class<? extends Service> serviceClass) {
```

```
    PackageManager packageManager = context.getPackageManager();
    PackageInfo packageInfo;
    try {
      packageInfo =
packageManager.getPackageInfo(context.getPackageName(),
GET_SERVICES);
    } catch (Exception e) {
      Log.e("AndroidUtils", "Could not get package info for " +
context.getPackageName(), e);
      return false;
    }
    String mainProcess = packageInfo.applicationInfo.processName;

    ComponentName component = new ComponentName(context, serviceClass);
    ServiceInfo serviceInfo;
    try {
      serviceInfo = packageManager.getServiceInfo(component, 0);
    } catch (PackageManager.NameNotFoundException ignored) {
      // Service is disabled.
      return false;
    }

    if (serviceInfo.processName.equals(mainProcess)) {
      Log.e("AndroidUtils",
          "Did not expect service " + serviceClass + " to run in main
process " + mainProcess);
      // Technically we are in the service process, but we're not in the
service dedicated process.
      return false;
    }

    int myPid = android.os.Process.myPid();
    ActivityManager activityManager =
        (ActivityManager)
context.getSystemService(Context.ACTIVITY_SERVICE);
    ActivityManager.RunningAppProcessInfo myProcess = null;

    for (ActivityManager.RunningAppProcessInfo process :
activityManager.getRunningAppProcesses()) {
      if (process.pid == myPid) {
        myProcess = process;
        break;
      }
    }
```

```
    if (myProcess == null) {
      Log.e("AndroidUtils", "Could not find running process for " +
myPid);
      return false;
    }
 //把 App 的进程 和 这个 Service 进程进行对比
    return myProcess.processName.equals(serviceInfo.processName);
  }
```

1.2 ServiceHeapDumpListener

设置 DisplayLeakService 和 HeapAnalyzerService 的可用。
analyze 方法，开始分析 HeapDump。

```
public final class ServiceHeapDumpListener implements HeapDump.Listener
{

  private final Context context;
  private final Class<? extends AbstractAnalysisResultService>
listenerServiceClass;

  public ServiceHeapDumpListener(Context context,
     Class<? extends AbstractAnalysisResultService>
listenerServiceClass) {
    LeakCanary.setEnabled(context, listenerServiceClass, true);
    LeakCanary.setEnabled(context, HeapAnalyzerService.class, true);
    this.listenerServiceClass = checkNotNull(listenerServiceClass,
"listenerServiceClass");
    this.context = checkNotNull(context,
"context").getApplicationContext();
  }

  @Override public void analyze(HeapDump heapDump) {
    checkNotNull(heapDump, "heapDump");
    HeapAnalyzerService.runAnalysis(context, heapDump,
listenerServiceClass);
  }}
```

2、RefWatcher

```
  private final Executor watchExecutor;
  private final DebuggerControl debuggerControl;
```

```
private final GcTrigger gcTrigger;
private final HeapDumper heapDumper;
private final Set<String> retainedKeys;
private final ReferenceQueue<Object> queue;
private final HeapDump.Listener heapdumpListener;
```

- watchExecutor：执行内存泄漏检测的 Executor。
- debuggerControl：用于查询是否在 debug 调试模式下，调试中不会执行内存泄漏检测。
- gcTrigger：GC 开关，调用系统 GC。
- heapDumper：用于产生内存泄漏分析用的 dump 文件。即 dump 内存 head。
- retainedKeys：保存待检测和产生内存泄漏的引用的 key。
- queue：用于判断弱引用持有的对象是否被 GC。
- heapdumpListener：用于分析 dump 文件，生成内存泄漏分析报告。

这里创建我们所需要的 RefWatcher。

```
public static RefWatcher androidWatcher(Application app,
HeapDump.Listener heapDumpListener) {
    DebuggerControl debuggerControl = new AndroidDebuggerControl();
    AndroidHeapDumper heapDumper = new AndroidHeapDumper(app);
    heapDumper.cleanup();
    return new RefWatcher(new AndroidWatchExecutor(), debuggerControl,
GcTrigger.DEFAULT,
        heapDumper, heapDumpListener);
}
```

3、ActivityRefWatcher

```
public static void installOnIcsPlus(Application application,
RefWatcher refWatcher) {
    if (SDK_INT < ICE_CREAM_SANDWICH) {
      // If you need to support Android < ICS, override onDestroy() in your
base activity.
      return;
    }
    ActivityRefWatcher activityRefWatcher = new
ActivityRefWatcher(application, refWatcher);
    activityRefWatcher.watchActivities();
}
//注册 lifecycleCallbacks
public void watchActivities() {
    // Make sure you don't get installed twice.
    stopWatchingActivities();
```

```
application.registerActivityLifecycleCallbacks(lifecycleCallbacks);
  }
  //ArrayList<ActivityLifecycleCallbacks> mActivityLifecycleCallbacks
  //就是 mActivityLifecycleCallbacks 的添加
  public void
registerActivityLifecycleCallbacks(ActivityLifecycleCallbacks
callback) {
       synchronized (mActivityLifecycleCallbacks) {
          mActivityLifecycleCallbacks.add(callback);
       }
    }
  // 注销 lifecycleCallbacks
   public void stopWatchingActivities() {

application.unregisterActivityLifecycleCallbacks(lifecycleCallbacks);
  }
  //  //就是 mActivityLifecycleCallbacks 的 移除
  public void
unregisterActivityLifecycleCallbacks(ActivityLifecycleCallbacks
callback) {
       synchronized (mActivityLifecycleCallbacks) {
          mActivityLifecycleCallbacks.remove(callback);
       }
    }
```

本质就是在 Activity 的 onActivityDestroyed 方法里 执行 refWatcher.watch(activity);

```
private final Application.ActivityLifecycleCallbacks
lifecycleCallbacks =
     new Application.ActivityLifecycleCallbacks() {
       @Override public void onActivityCreated(Activity activity, Bundle
savedInstanceState) {
       }

       @Override public void onActivityStarted(Activity activity) {
       }

       @Override public void onActivityResumed(Activity activity) {
       }

       @Override public void onActivityPaused(Activity activity) {
```

```
        }

        @Override public void onActivityStopped(Activity activity) {
        }

        @Override public void onActivitySaveInstanceState(Activity
activity, Bundle outState) {
        }

        @Override public void onActivityDestroyed(Activity activity) {
          ActivityRefWatcher.this.onActivityDestroyed(activity);
        }
      };

  void onActivityDestroyed(Activity activity) {
    refWatcher.watch(activity);
  }
```

4、watch

```
  public void watch(Object watchedReference, String referenceName) {
    checkNotNull(watchedReference, "watchedReference");
    checkNotNull(referenceName, "referenceName");
    if (debuggerControl.isDebuggerAttached()) {
      return;
    }
    //随机生成 watchedReference 的 key 保证其唯一性
    final long watchStartNanoTime = System.nanoTime();
    String key = UUID.randomUUID().toString();
    retainedKeys.add(key);
    //这个一个弱引用的子类拓展类 用于 我们之前所说的 watchedReference
和  queue 的联合使用
    final KeyedWeakReference reference =
        new KeyedWeakReference(watchedReference, key, referenceName,
queue);

    watchExecutor.execute(new Runnable() {
      @Override public void run() {
      //重要方法，确然是否 内存泄漏
        ensureGone(reference, watchStartNanoTime);
      }
    });
  }
```

```
final class KeyedWeakReference extends WeakReference<Object> {
  public final String key;
  public final String name;

  KeyedWeakReference(Object referent, String key, String name,
      ReferenceQueue<Object> referenceQueue) {
    super(checkNotNull(referent, "referent"),
checkNotNull(referenceQueue, "referenceQueue"));
    this.key = checkNotNull(key, "key");
    this.name = checkNotNull(name, "name");
  }}
```

5、ensureGone

```
  void ensureGone(KeyedWeakReference reference, long watchStartNanoTime)
{
    long gcStartNanoTime = System.nanoTime();

    long watchDurationMs = NANOSECONDS.toMillis(gcStartNanoTime -
watchStartNanoTime);
  //把 queue 的引用 根据 key 从 retainedKeys 中引出 。
  //retainedKeys 中剩下的就是没有分析和内存泄漏的引用的 key
  removeWeaklyReachableReferences();
  //如果内存没有泄漏 或者处于 debug 模式那么就直接返回
    if (gone(reference) || debuggerControl.isDebuggerAttached()) {
      return;
    }
    //如果内存依旧没有被释放 那么在 GC 一次
    gcTrigger.runGc();
    //再次 清理下 retainedKeys
    removeWeaklyReachableReferences();
    //最后还有 就是说明内存泄漏了
    if (!gone(reference)) {
      long startDumpHeap = System.nanoTime();
      long gcDurationMs = NANOSECONDS.toMillis(startDumpHeap -
gcStartNanoTime);//dump 出 Head 报告
      File heapDumpFile = heapDumper.dumpHeap();

      if (heapDumpFile == null) {
        // Could not dump the heap, abort.
        return;
      }
```

```
    long heapDumpDurationMs = NANOSECONDS.toMillis(System.nanoTime() -
startDumpHeap);
    //最后进行分析 这份 HeapDump
    //LeakCanary 分析内存泄露用的是一个第三方工具 HAHA 别笑 真的是这
个名字
    heapdumpListener.analyze(
        new HeapDump(heapDumpFile, reference.key, reference.name,
watchDurationMs, gcDurationMs,
            heapDumpDurationMs));
    }
  }
  private void removeWeaklyReachableReferences() {
    // WeakReferences are enqueued as soon as the object to which they
point to becomes weakly
    // reachable. This is before finalization or garbage collection has
actually happened.
    KeyedWeakReference ref;
    while ((ref = (KeyedWeakReference) queue.poll()) != null) {
      retainedKeys.remove(ref.key);
    }
  }

  private boolean gone(KeyedWeakReference reference) {
    return !retainedKeys.contains(reference.key);
  }
```

6、haha

大家有兴趣可以分析下这个分析库的原理。在这里就不深入研究了。

最后把分析的引用链 写入文件中，发通知。

```
  @TargetApi(HONEYCOMB) @Override
  protected final void onHeapAnalyzed(HeapDump heapDump, AnalysisResult
result) {
    String leakInfo = leakInfo(this, heapDump, result);
    Log.d("LeakCanary", leakInfo);

    if (!result.leakFound || result.excludedLeak) {
      afterDefaultHandling(heapDump, result, leakInfo);
      return;
    }

    File leakDirectory = DisplayLeakActivity.leakDirectory(this);
```

```
    int maxStoredLeaks =
getResources().getInteger(R.integer.__leak_canary_max_stored_leaks);
    File renamedFile = findNextAvailableHprofFile(leakDirectory,
maxStoredLeaks);

    if (renamedFile == null) {
      // No file available.
      Log.e("LeakCanary",
          "Leak result dropped because we already store " + maxStoredLeaks
+ " leak traces.");
      afterDefaultHandling(heapDump, result, leakInfo);
      return;
    }

    heapDump = heapDump.renameFile(renamedFile);

    File resultFile = DisplayLeakActivity.leakResultFile(renamedFile);
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream(resultFile);
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(heapDump);
      oos.writeObject(result);
    } catch (IOException e) {
      Log.e("LeakCanary", "Could not save leak analysis result to disk",
e);
      afterDefaultHandling(heapDump, result, leakInfo);
      return;
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException ignored) {
        }
      }
    }

    PendingIntent pendingIntent =
        DisplayLeakActivity.createPendingIntent(this,
heapDump.referenceKey);

    String contentTitle =
        getString(R.string.__leak_canary_class_has_leaked,
classSimpleName(result.className));
```

```
    String contentText =
getString(R.string.__leak_canary_notification_message);

    NotificationManager notificationManager =
        (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

    Notification notification;
    if (SDK_INT < HONEYCOMB) {
      notification = new Notification();
      notification.icon = R.drawable.__leak_canary_notification;
      notification.when = System.currentTimeMillis();
      notification.flags |= Notification.FLAG_AUTO_CANCEL;
      notification.setLatestEventInfo(this, contentTitle, contentText,
pendingIntent);
    } else {
      Notification.Builder builder = new Notification.Builder(this) //
          .setSmallIcon(R.drawable.__leak_canary_notification)
          .setWhen(System.currentTimeMillis())
          .setContentTitle(contentTitle)
          .setContentText(contentText)
          .setAutoCancel(true)
          .setContentIntent(pendingIntent);
      if (SDK_INT < JELLY_BEAN) {
        notification = builder.getNotification();
      } else {
        notification = builder.build();
      }
    }
    notificationManager.notify(0xDEAFBEEF, notification);
    afterDefaultHandling(heapDump, result, leakInfo);
  }
```

总结

其实沿着源码分析很容易让人无法自拔，所以我们更要跳出来看到本质。

1、监听 Activity 的生命周期，在 onDestory 方法里调用 RefWatcher 的 watch 方法。
watch 方法监控的是 Activity 对象
2、给 Activyty 的 Reference 生成唯一性的 key 添加到 retainedKeys 。生成 KeyedWeakReference 对象 ，Activity 的弱引用和 ReferenceQueue 关联。执行 ensureGone 方法。
3、如果 retainedKeys 中没有 该 Reference 的 key 那么就说明没有内存泄漏。
4、如果有，那么 analyze 分析我们 HeadDump 文件。建立导致泄漏的引用链。

5、引用链传递给 APP 进程的 DisplayLeakService，以通知的形式展示出来。

3. ARouter

前言

阅读本文之前，建议读者：

- 对 Arouter 的使用有一定的了解。
- 对 Apt 技术有所了解。

Arouter 是一款 Alibaba 出品的优秀的路由框架，本文不对其进行全面的分析，只对其最重要的功能进行源码以及思路分析，至于其拦截器，降级，ioc 等功能感兴趣的同学请自行阅读源码，强烈推荐阅读云栖社区的官方介绍。

对于一个框架的学习和讲解，我个人喜欢先将其最核心的思路用简单一两句话总结出来：ARouter 通过 Apt 技术，生成保存**路径(路由 path)**和**被注解(@Router)的组件类**的映射关系的类，利用这些保存了映射关系的类，Arouter 根据用户的请求 postcard（明信片）寻找到要跳转的目标地址(class),使用 Intent 跳转。

原理很简单，可以看出来，该框架的核心是利用 apt 生成的映射关系，这里要用到 Apt 技术，读者可以自行搜索了解一下。

分析

我们先看最简单的代码的使用：

首先需要在需要跳转的组件添加注解

```
@Route(path = "/main/homepage")public class HomeActivity extends BaseActivity
{

        onCreate()

        ....

}
```

然后在需要跳转的时候调用

```
Arouter.getInstance().build("main/hello").navigation;
```

这里的路径"main/hello"是用户唯一配置的东西，我们需要通过这个 path 找到对应的 Activity。最简单的思路就是通过 APT 技术，寻找到所有带有注解@Router 的组件，将其注解值 path 和对应的 Activity 保存到一个 map 里，比如像下面这样：

```
class RouterMap {

    public Map getAllRoutes {
```

```
            Map map = new HashMap<String,Class<?>>;

            map.put("/main/homepage",HomeActivity.class);

            map.put("/main/setting",SettingActivity.class);

            map.put("/login/register",LoginRegisterActivity.class);

            ....


        return map;

    }

}
```

然后在工程代码中将这个 map 加载到内存中，需要的时候直接 get(path)就可以了，这种方案似乎能解决我们的问题。

发现问题

上面的思路确实能够实现路由功能，但是这么做会存在一个较大的问题：对于一个大型项目，组件数量会很多，可能会有一两百或者更多，把这么多组件都放到这个 Map 里，显然会对内存造成很大的压力，因此，Arouter 作为一款阿里出品的优秀框架，显然是要解决这个问题的。

这里建议读者自行思考一下，如何解决一次性加载所有映射关系带来的内存损耗问题，我在思考这个问题的时候首先想到的是“懒加载”，但是仅仅懒加载是不够的，因为懒加载后如果还是一次性把所有映射关系加载进来，内存损耗还是一样大的。因此，再深入思考一下，可能还能想出解决一个思路：分段懒加载，思路有了，如何实现呢？这里还是建议大家在阅读下面的内容之前思考一下，或许你能想到一套不同于 Arouter 的方案来哦。

Arouter 采用的方法就是“分组+按需加载”，分组还带来的另一个好处是便于管理，下面我们来看一下实现原理。

解决步骤一：分组

首先看如何分组的，Arouter 在一层 map 之外，增加了一层 map，我们看 WareHouse 这个类，里面有两个静态 Map：

```
    static Map<String, Class<? extends IRouteGroup>> groupsIndex = new
HashMap<>();

    static Map<String, RouteMeta> routes = new HashMap<>();
```

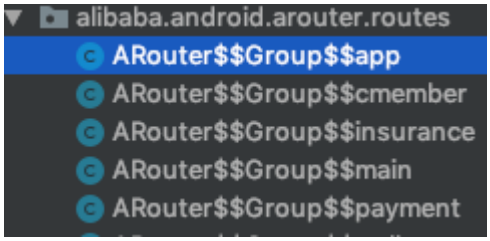
- groupsIndex 保存了 group 名字到 IRouteGroup 类的映射，这一层映射就是 Arouter 新增的一层映射关系。
-
-

routes 保存了路径 path 到 RouteMeta 的映射，其中，RouteMeta 是目标地址的包装。这一层映射关系跟我门自己方案里的 map 是一致的，我们路径跳转也是要用这一层来映射。

这里出现了两个我们不认识的类，IRouteGroup 和 RouteMeta，后者很简单，就是对跳转目标的封装，我们后续称其为"目标"，其内包含了目标组件的信息，比如 Activity 的 Class。那 IRouteGroup 是个什么东西？

```
public interface IRouteGroup {
    /**
     * Fill the atlas with routes in group.
     */
    void loadInto(Map<String, RouteMeta> atlas);
}
```

一个接口，只有一个方法 loadInto，都有谁实现了这个接口呢？我拿我手上的一个项目为例，Arouter 通过 apt 生成了下面几个类：



这几个类都以 Arouter$$Group 开头，我们随便拿一个看看：

```
public class ARouter$$Group$$main implements IRouteGroup {
    @Override
    public void loadInto(Map<String, RouteMeta> atlas) {
        atlas.put("/main/fa/leakscan", RouteMeta.build(RouteType.ACTIVITY,
MainFaLeakScanActivity.class, "/main/fa/leakscan", "main", }}, -1, 1));
        atlas.put("/main/login", RouteMeta.build(RouteType.ACTIVITY,
LoginActivity.class, "/main/login", "main", null, -1, -2147483648));
        atlas.put("/main/register", RouteMeta.build(RouteType.ACTIVITY,
RegPhoneActivity.class, "/main/register", "main", null, -1, -2147483648));
    }
}
```

我们看到，他实现了 loadInto 方法，在这个方法中，它往这个 HashMap 中填充了好多数据，填充的是什么呢？填充的是路径 path 和它对应的目标 RouteMeta，也就是我们最终

需要的那层映射关系。而且，我们还能观察到：这个类下面所有的路由 path 都有一个共同点，即全是"/main"开头的，也就是说，这个类加载的映射关系，都是在一个组内的。因此我们总结出：

Arouter 通过 apt 技术，为每个组生成了一个以 Arouter$$Group 开头的类，这个类负责向 atlas 这个 Hashmap 中填充组内路由数据。

IRouteGroup 正如其名字，它就是一个能装载该组路由映射数据的类，其实有点像个工具类，为了方便后续讲解，我们姑且称上面这样一个实现了 IRouteGroup 的类叫做"组加载器"，本质是一个类。上图中的类是一个组加载器，其他所有以 Arouter$$Group 开头的类都是一个"组加载器"。回到之前的主线，Warehoust 中的两个 Hashmap，其中 groupsIndex 这个 map 中保存的是什么呢？我们通过它的调用找到这一行代码(已简化)：

```
for (String className : routerMap) {
    if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEPARATOR +SUFFIX_ROOT)) {
        ((IRouteRoot) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.groupsIndex);
    }
}
```

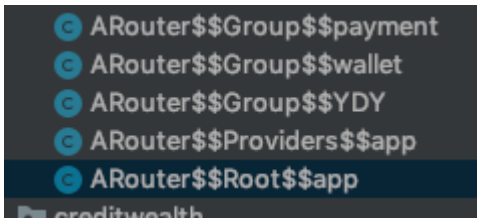
其中 `ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME + SEPARATOR +SUFFIX_ROOT` 这行代码是几个静态字符串拼起来的，它等于 `com.alibaba.android.arouter.routes.Arouter$$Root` 。另外 routerMap 是什么呢？它是一个 HashSet<String>：

```
routerMap = ClassUtils.getFileNameByPackageName(mContext, ROUTE_ROOT_PAKCAGE);
```

这一行代码对它进行了初始化，目的是找到 `com.alibaba.android.arouter.routes` 这个包名下所有的类，将其类名保存到 routerMap 中。因此，上面的代码意思就是将 `com.alibaba.android.arouter.routes` 包下所有名字以 `com.alibaba.android.arouter.routes.Arouter$$Root` 开头的类找出来，通过反射实例化并强转成 IRouteRoot，然后调用 loadInto 方法。这里又出来一个新的接口：IRouteRoot，我们看代码：

```
public interface IRouteRoot {

    /**
     * Load routes to input
     * @param routes input
     */
    void loadInto(Map<String, Class<? extends IRouteGroup>> routes);
}
```

跟 IRouteGroup 长得还挺像，也是 loadInto，我们看它的实现。还是以我的项目为例，在 apt 生成的文件夹下查找：



最底下一行，有个 Arouter$$Root$$app，它符合前面名字规则，我们进去看看：

```
public class ARouter$$Root$$app implements IRouteRoot {

@Override

public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {

    routes.put("YDY", ARouter$$Group$$YDY.class);

    routes.put("app", ARouter$$Group$$app.class);

    routes.put("main", ARouter$$Group$$main.class);

    routes.put("payment", ARouter$$Group$$payment.class);

    routes.put("wallet", ARouter$$Group$$wallet.class);

  }

}
```

这个类实现了 IRouteRoot，在 loadInto 方法中，他将组名和组对应的"组加载器"保存到了 routes 这个 map 中。也就是说，这个类将所有的"组加载器"给索引了下来，通过任意一个组名，可以找到对应的"组加载器"，我们再回到前面讲的初始化 Arouter 时候的方法中：

```
((IRouteRoot)
(Class.forName(className).getConstructor().newInstance())).loadInto(Warehou
se.groupsIndex);
```

理解了吧，这个方法的意义就在于将所有的组路由加载类索引到了 groupsIndex 这个 map 中。因此我们就明白了：

WareHouse 中的 groupsIndex 保存的是组名到"组加载器"的映射关系

说句题外话：回过头想想前面用到的两个接口：IRouteGroup 和 IRouteRoot，它们其实是 apt 生成的类和我们项目中代码之间沟通的桥梁，熟悉 AIDL 的同学可能会觉得很熟悉，二者其实是异曲同工的，两个系统进行交互的时候都是通过接口来沟通的。当然，在使用 apt 生成的类时，我们需要用到反射技术。

总结一下 Arouter 的分组设计：Arouter 在原来 path 到目标的 map 外，加了一个新的 map，该 map 保存了组名到"组加载器"的映射关系。其中"组加载器"是一个类，可以加载其组内的 path 到目标的映射关系。

到此为止，Arouter 只是完成了分组工作，但这么做的目的是什么呢？别着急，前面的都只是铺垫，接下来才是这个分组设计发挥作用的地方，我们进入"按需加载"的代码分析：

解决步骤二：按需加载

之前说过，Arouter 使用的是分组按需加载，分组是为了按需做准备的。我们看 Arouter 是怎么按需加载的，我们还是从代码的使用入手：

```
Arouter.getInstance().build("main/hello").navigation;
```

在 navigation 这个方法中，最终会跳转到这里：

```
protected Object navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallback callback) {
        try {
            //请关注这一行
            LogisticsCenter.completion(postcard);
        } catch (NoRouteFoundException ex) {
            logger.warning(Consts.TAG, ex.getMessage());
            ....//简化代码
        }
        //调用 Intent 跳转
        return _navigation(context, postcard, requestCode, callback)
```

最后一行的 return 语句很简单，就是去调用 Intent 唤起组件了，我们看前面 try 中的第一行 `LogisticsCenter.completion(postcard)`,我们进到这个函数里：

```
//从缓存里取路由信息
RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath());//如果为空，需要加载该组的路由 if (null == routeMeta) {
  Class<? extends IRouteGroup> groupMeta =
Warehouse.groupsIndex.get(postcard.getGroup());
  IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstance();
  iGroupInstance.loadInto(Warehouse.routes);
  Warehouse.groupsIndex.remove(postcard.getGroup());
} //如果不为空，走后续流程 else {
    postcard.setDestination(routeMeta.getDestination());
    ...
}
```

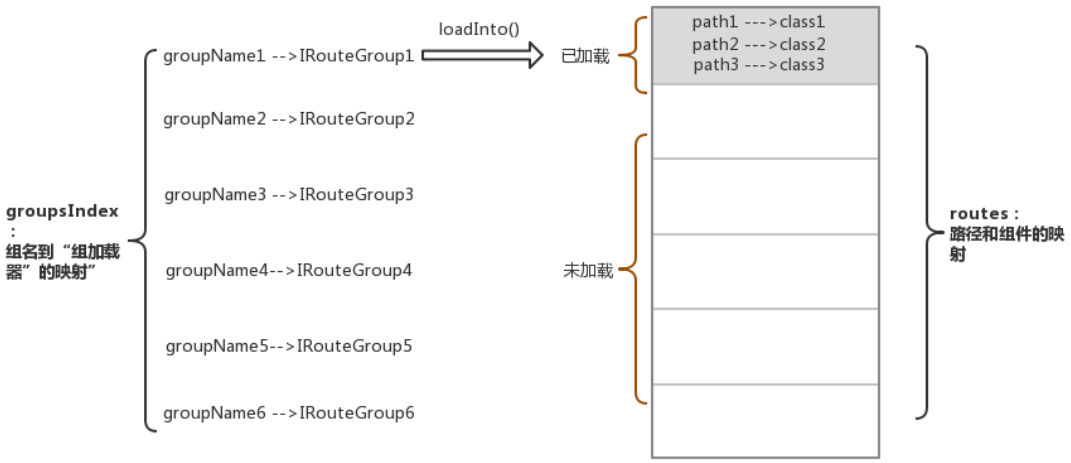
这段代码就是“按需加载”的核心逻辑所在了，我对其进行了简化，分析其逻辑是这样的：

- 首先从 Warehouse.routes(前面说了，这里存放的是 path 到目标的映射)里拿到目标信息，如果找不到，说明这个信息还没加载，需要加载，实际上，刚开始这个 routes 里面什么都没有。
- 加载流程：首先从 Warehouse.groupsIndex 里获取“组加载器”，组加载器是一个类，需要通过反射将其实例化，实例化为 iGroupInstance，接着调用组加载器的加载方法 loadInto，将该组的路由映射关系加载到 Warehouse.routes 中，加载完成后，routes 中就缓存下来当前组的所有路由映射了，因此这个组加载器其实就没用了，为了节省内存，将其从 Warehouse.groupsIndex 移除。
- 如果之前加载过，则在 Warehouse.routes 里面是可以找到路有映射关系的，因此直接将目标信息 routeMeta 传递给 postcard，保存在 postcard 中，这样 postcard 就知道了最终要去哪个组件了。

到此为止分组按需加载的逻辑就都分析完了，通过这两个步骤，解决了路由映射一次性加载到内存占用内存过大的缺点，这是 Arouter 这个框架优秀的重要原因之一。当然 Arouter 还有一些优秀的功能，比如拦截器，依赖注入等，总之，功能全，性能好，使用方便，这些都是 Arouter 受欢迎的原因，这点值得我们所有开发者去学习。

总结

最后结合一张图总结一下 Arouter 的分组按需加载的逻辑：



图中左侧 groupsIndex 是“组映射”，右侧 routes 是“路由映射”。Arouter 在初始化的时候，通过反射技术，将所有的“组加载器”索引到 groupsIndex 这个 map 中，而此时，右侧的 routes 还是空的。在用户调用 navigation()进行跳转的时候，会根据路径提取组名，由组名根据 groupsIndex 获取到相应组的“组加载器”，由组加载器加载对应组内的路由信息，此时保存全局“路由目标映射的”routes 这个 map 中就保存了刚才组的所有路由映射关系了。同样，当其他组请求时，其他组也会加载组对应的路由映射，这样就实现了整个 App 运行时，只有用到的组才会加到内存中，没有去过的组就不会加载到内存中，达到了节省内存的目的。

插件化（不同插件化机制原理与流派，优缺点。局限性）

1.动态加载 so 库（其实可以放在外部存储，我们常用的是放在内部存储）

2.classloader 动态加载外部可执行文件，如 dex,jar,apk

关于动态加载，细分为三种：
第一，简单的动态加载；这种不适用插件的 activity。使用插件的 fragment，或者只是用 dex 中的逻辑，或者用代码编写布局的方式替换布局。实现起来比较简单，比较稳定，但是有限制，适用于界面改动小，或者不改动，逻辑 sdk，登录注册界面等等。

第二，静态代理 activity；这种是使用插件的 activity，但是实际上使用的是 host 的 activity。使用插件的 activity，需要解决两个问题，

一是生命周期，

关于生命周期，通过在 host 提前注册好需要的 activity，然后拿到插件 activity 引用（两种方式，通过反射拿到，再则通过把插件 activity 接口化，通过接口调用的方式，更推荐这种，方便 host 控制插件），在各个生命周期方法中同步调用。

二是 R 资源。

关于 R 资源，需要通过反射拿到 AssertManager,调用其 addAssertPath，得到 Resource 资源对象。相当于多维护了一套 R 资源。实际上，我们平日用的就有两套 R 资源，应用层的一套，然后系统层的一套。

第三，动态注册 activity

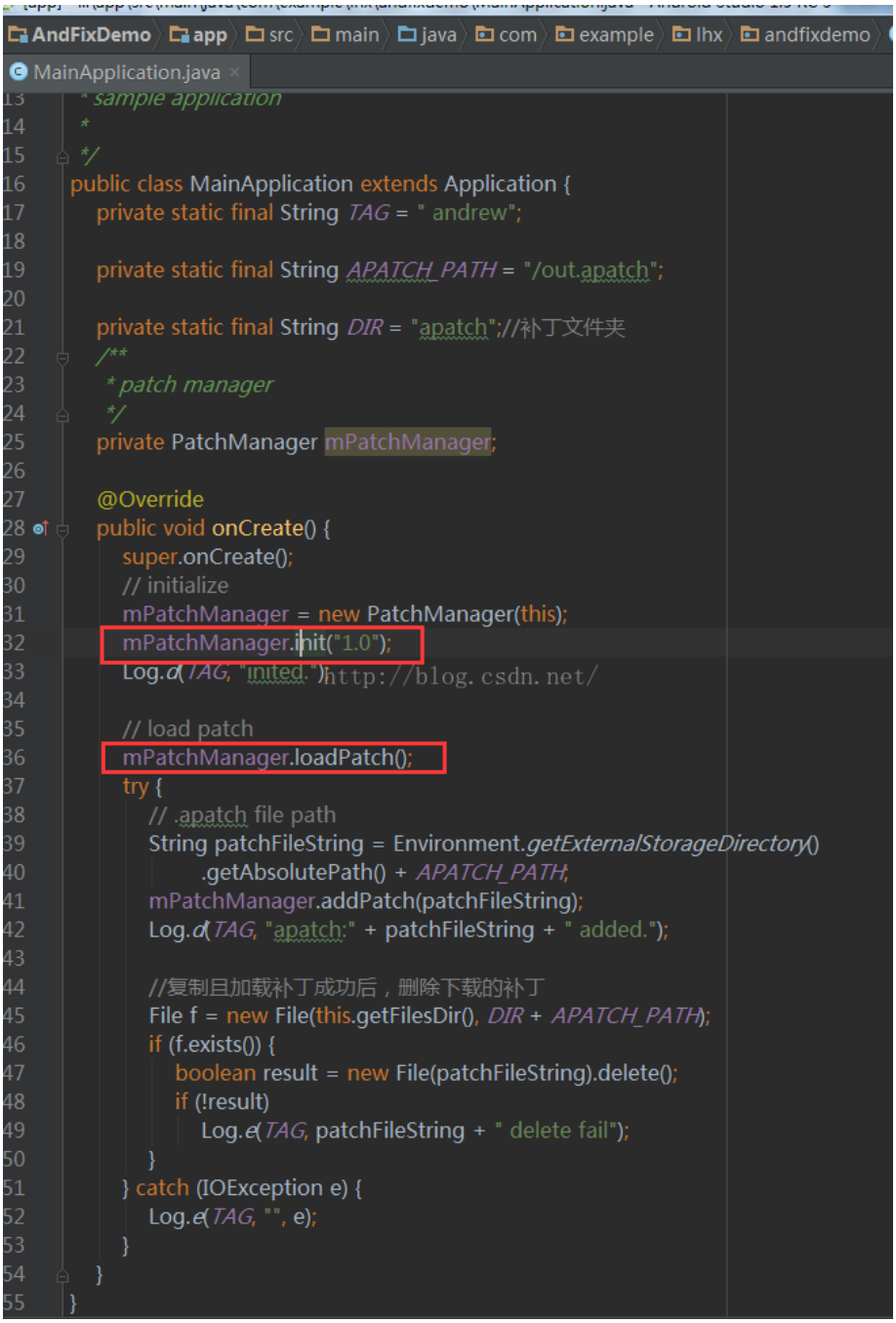
关于动态注册 activity，他解决生命周期和 R 资源的方式就更简单了，相当于是通过标准模式启动了一个 activity，自然而然该 activity 就具备了生命周期和 R 资源。那么这种该如何做呢？

首先，host 中，要有一个通用的 PluginActivity，当要启动 PluginActivity 的时候，我们复写 ClassLoader 的 loadClass 方法，当要找的事 PluginActivity,我们通过 DexMaker 生成一个 TargetActivity extends PluginActivity. 然后 return TargetActivity 实例。这样就完美解决了生命周期和 R 资源的问题。

如果插件里面需要有多个 activity，如何从 TargetActivity 中跳转到其他 Activity 呢，实际上我们知道所有的跳转都会走到 startActivityForResult，我们只需要在用 DexMaker 生成 TargetActivity 的时候，复写对应的 startActivityForResult 方法就行了。

热修复

。首先看下 Demo 里面 Application 的代码。



2.一開始实例化 PatchManager。然后调用 init()这种方法，我们跟进去看看。

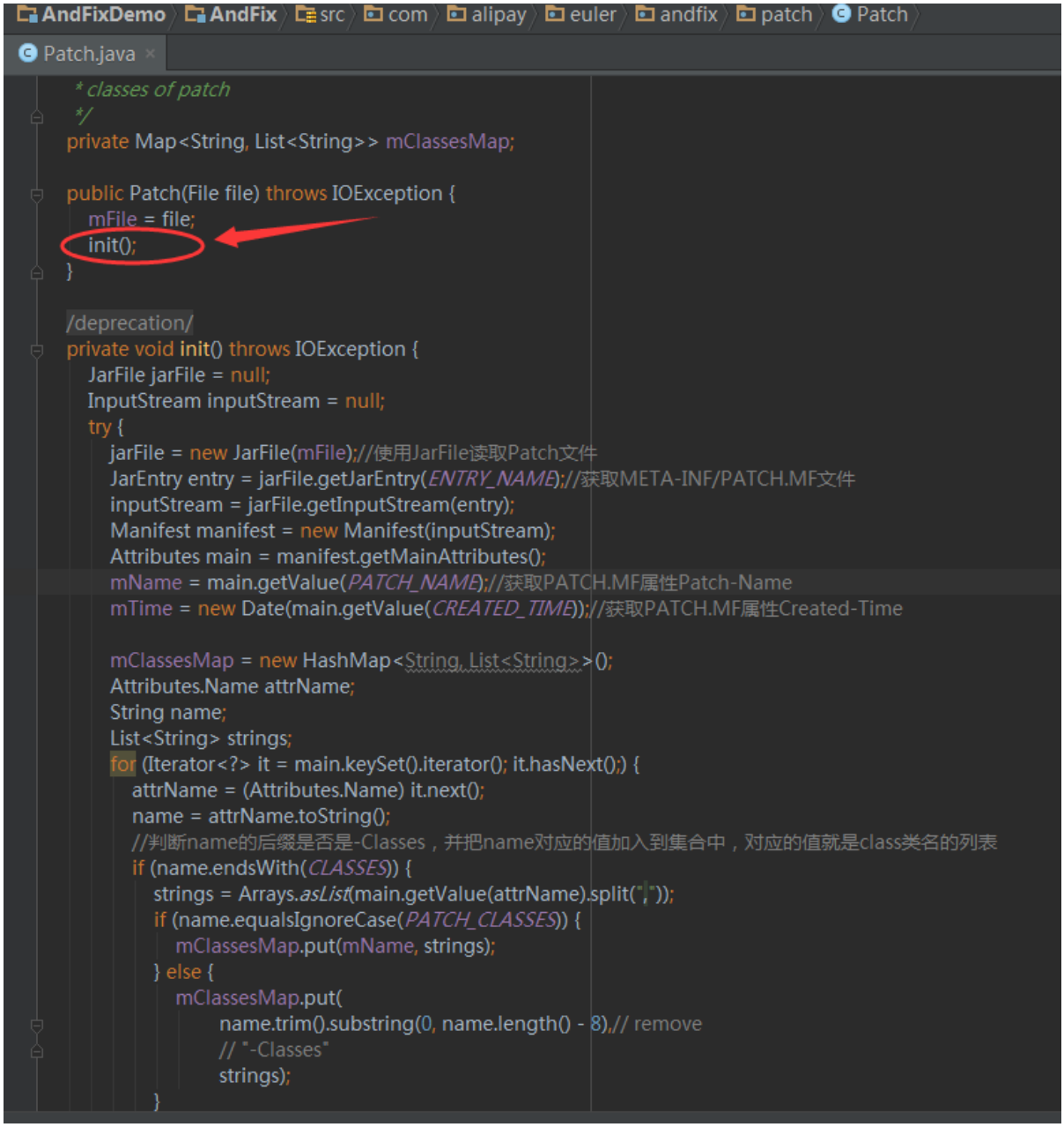
我凝视的非常具体，大致就是从 SharedPreferences 读取曾经存的版本号和你传过来的版本号进行比对，假设两者版本号不一致就删除本地 patch。否则调用 initPatchs()这种方法。

```
/**
 * initialize
 *
 * @param appVersion App version
 */
public void init(String appVersion) {
    if (!mPatchDir.exists() && !mPatchDir.mkdirs()) {// make directory fail
        Log.e(TAG, "patch dir create error.");
        return;
    } else if (!mPatchDir.isDirectory()) {//如果遇到同名的文件，则将该同名文件删除
        mPatchDir.delete();
        return;
    }
    //在该文件下放入一个名为_andfix_的SharedPreferences文件
    SharedPreferences sp = mContext.getSharedPreferences(SP_NAME,
            Context.MODE_PRIVATE);//存储关于patch文件的信息
    //根据你传入的版本号和之前的对比，做不同的处理
    String ver = sp.getString(SP_VERSION, null);
    //如果从_andfix_这个文件获取的ver不是null，而且这个ver和外部初始化时传进来的版本号一致
    if (ver == null || !ver.equalsIgnoreCase(appVersion)) {
        cleanPatch();//删除本地patch文件
        sp.edit().putString(SP_VERSION, appVersion).commit();//并把传入的版本号保存
    } else {
        initPatchs();//mPatchDir文件夹下的文件作为参数传给了addPatch(File)方法
    }
}
```

3.分析下 initPatchs()它做了什么，事实上代码非常 easy，就是把 mPatchDir 目录下的文件作为参数传给了 addPatch(File)方法，然后调用 addPatch()方法。addPatch 方法的作用看以下的凝视，写的非常清楚。

```
private void initPatchs() {
    File[] files = mPatchDir.listFiles();
    for (File file : files) {
        addPatch(file);
    }
}

/**
 * add patch file
 *
 * @param file
 * @return patch
 */
//把扩展名为.apatch的文件传给Patch做参数，初始化对应的Patch，
//并把刚初始化的Patch加入到我们之前看到的Patch集合mPatchs中
private Patch addPatch(File file) {
    Patch patch = null;
    if (file.getName().endsWith(SUFFIX)) {
        try {
            patch = new Patch(file);//实例化Patch对象
            mPatchs.add(patch);//把patch实例存储到内存的集合中,在PatchManager实例化集合
        } catch (IOException e) {
            Log.e(TAG, "addPatch", e);
        }
    }
    return patch;
}
```


4.我们能够看到 addPatch()方法里面会实例化 Patch，我们跟进去看看实例化过程中，它又干了什么事。



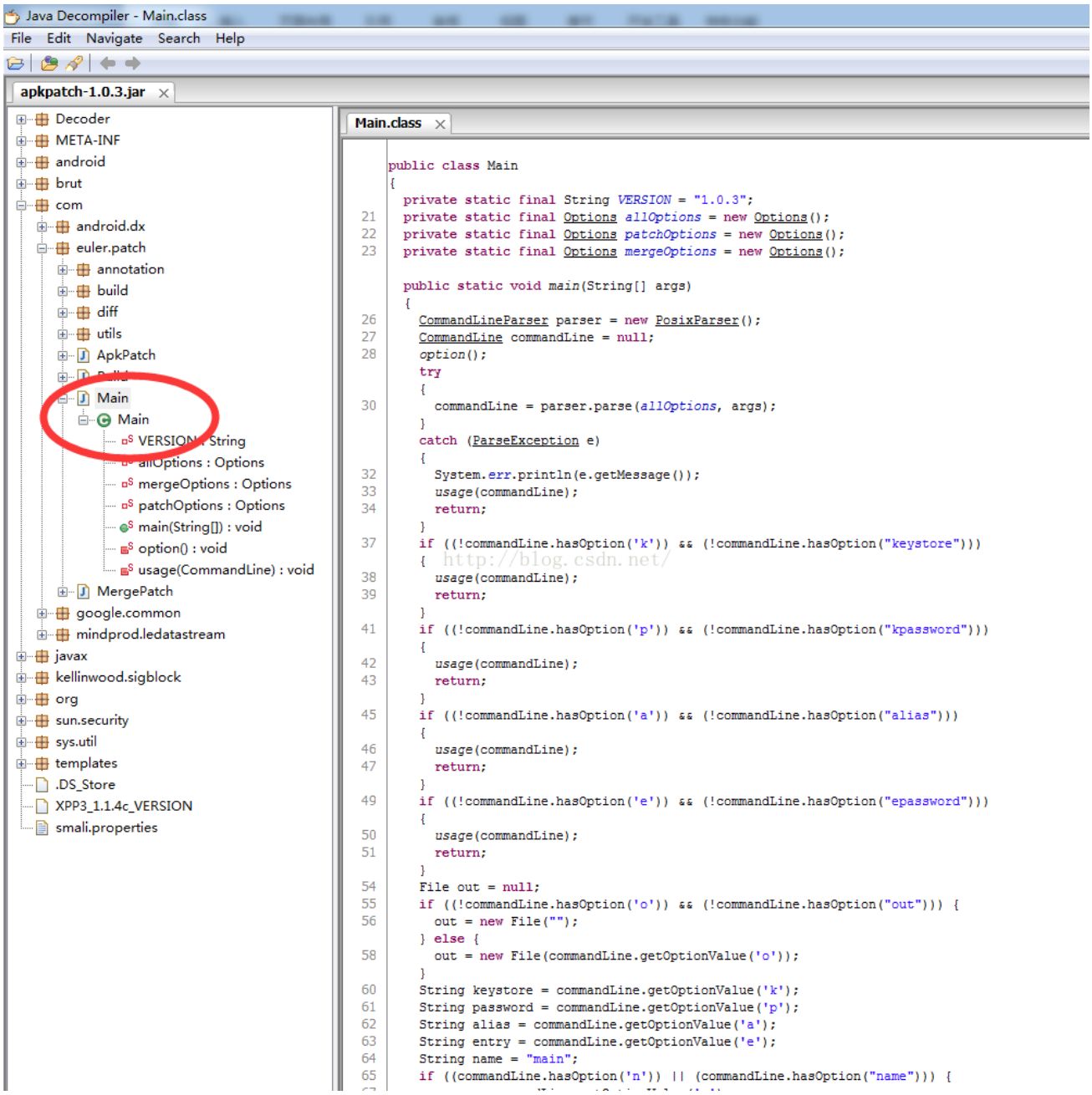
它里面调用了 init()方法，能够看到里面有

JarFile, JarEntry, Manifest, Attributes,通过它们一层层的从 Jar 文件里获取对应的值，提到这里大家可能会奇怪,明明是.patch 文件，怎么又变成 Jar 文件了？事实上是通过阿里打补丁包工具生成补丁的时候写入对应的值。补丁文件事实上就相到于 jar 包。仅仅只是它们的扩展名

不同而已。提到这里我们就来单独的探索下，补丁文件是怎么一步步生成的。由于阿里没有对打补丁工具进行加密和混淆。我们能够使用 jdgui 打开查看。

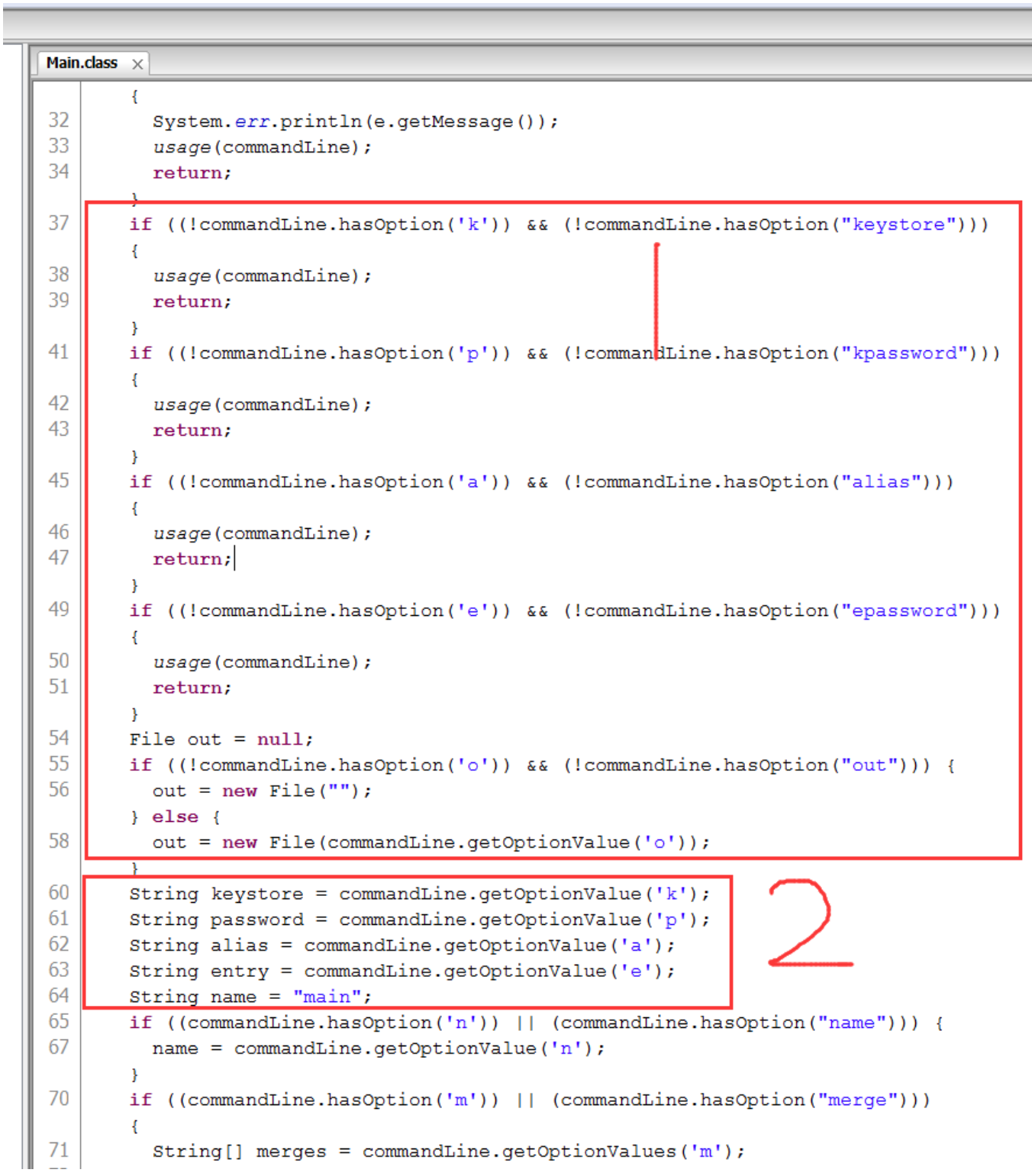
所需对应的工具代码 demo 等我都统一放在以下的下载链接里面。有须要的自行取下。

5.好了，我们如今来分析下补丁文件怎样生成的，用 jdgui 打开 apkpatch-1.0.3。先从 main 方法開始。

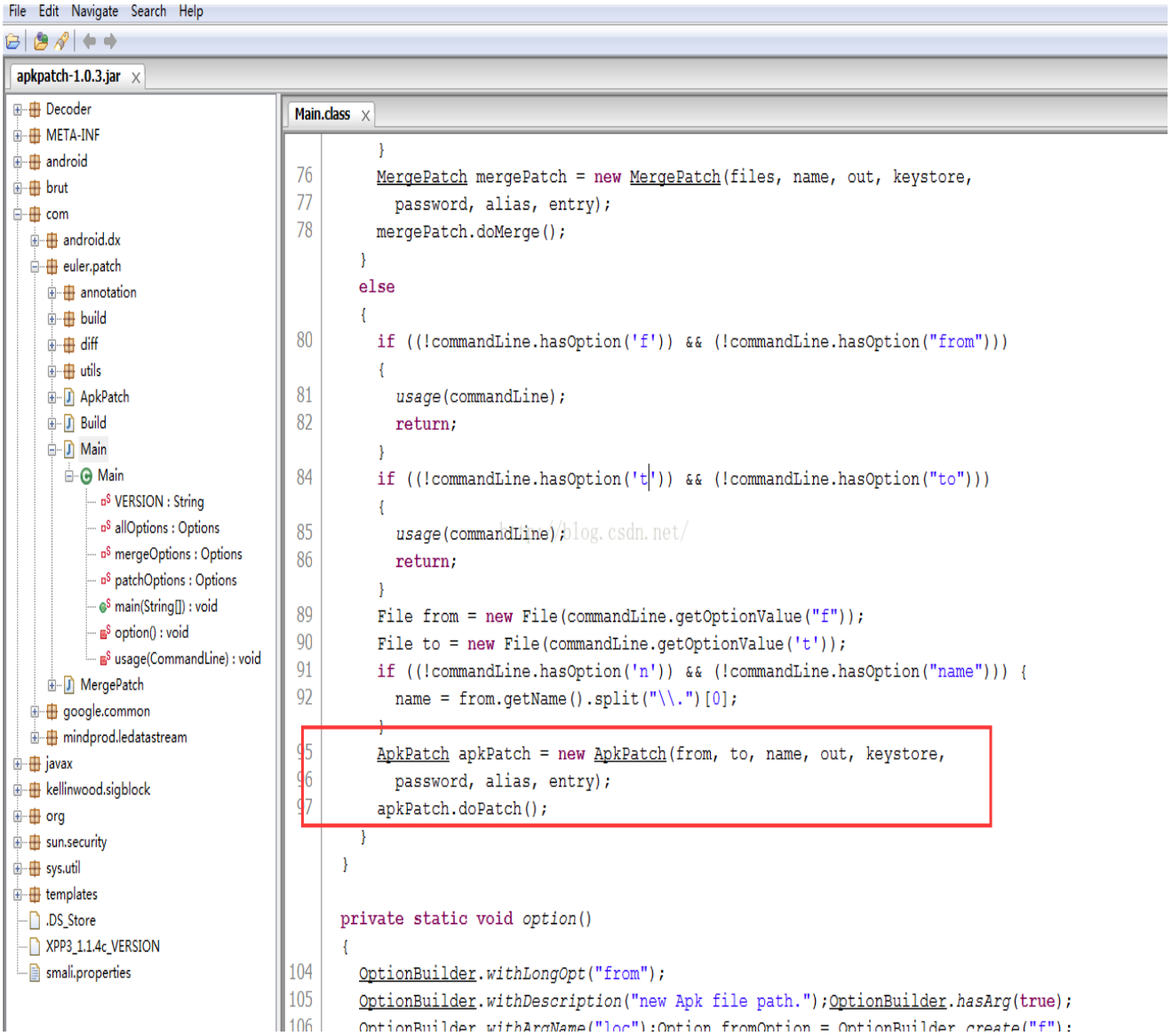


能够看到：下图 1 部分就是我们前面介绍怎样使用命令行打补丁包的命令，检查命令行是否有那些参数。假设没有要求的参数，就给用户对应的提示。

第二部分。我们在打正式包的时候，会指定 keystore，password，alias，entry 相关参数。另外 name 就是最后生成的文件，能够忽略。



Main 函数最后一个方法是我们的大头戏。上面的参数传给 ApkPatch 进行初始化。然后调用 doPatch()方法。



我们再跟进去。看看 ApkPatch 初始化的过程中，做了什么。

Main.class ApkPatch.class

```
package com.euler.patch;

import brut.androlib.mod.SmaliMod;

public class ApkPatch
  extends Build
{
  private File from;
  private File to;
  private Set<String> classes;

  public ApkPatch(File from, File to, String name, File out, String keystore, String password, String alias, String entry)
  {
41  super(name, out, keystore, password, alias, entry);
42    this.from = from;
43    this.to = to;
  }

  public void doPatch()
  {
    try
    {
48    File smaliDir = new File(this.out, "smali");
```


调用了父类的方法，我们再看看父类 Build.

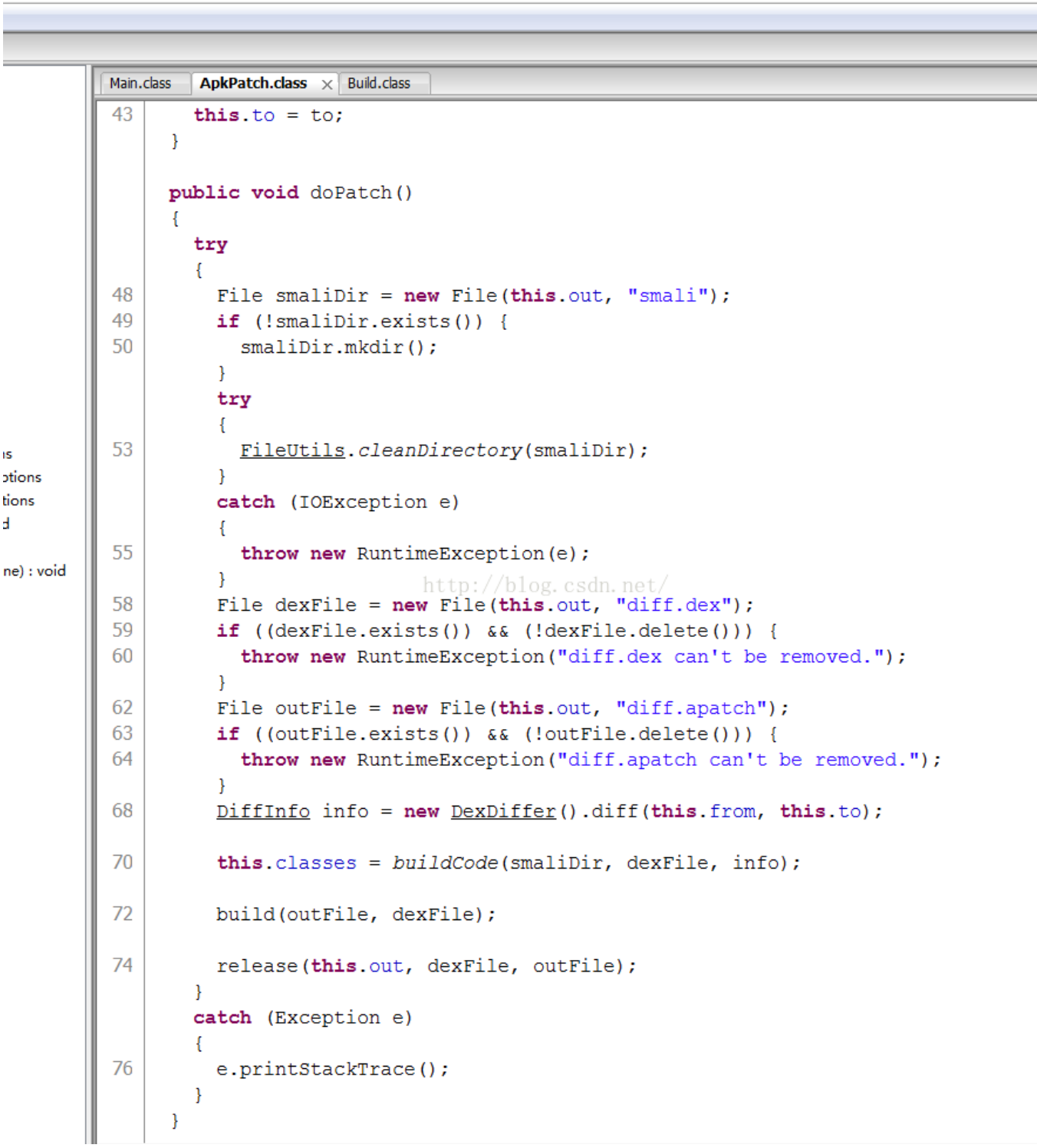
Main.class | ApkPatch.class | Build.class

```
package com.euler.patch;

import com.euler.patch.build.PatchBuilder;

public abstract class Build
{
  protected static final String SUFFIX = ".apatch";
  protected String name;
  private String keystore;
  private String password;
  private String alias;
  private String entry;
  protected File out;

  public Build(String name, File out, String keystore, String password, String alias, String entry)
  {
36  this.name = name;
37  this.out = out;
38  this.keystore = keystore;
39  this.password = password;
40  this.alias = alias;
41  this.entry = entry;
42  if (!out.exists()) {
43    out.mkdirs();
44  } else if (!out.isDirectory()) {
45    throw new RuntimeException("output path must be directory.");
    }
  }
```


干的事情事实上比较简单。就是给变量进行赋值。能够看到 out,我们的输出文件就是这么来的，没有的话，它会自己创建一个。

然后我们再回到 apkPatch.doPatch()这种方法。

看看这种方法里面是什么。



这种方法主要做的就是在我们的 out 输出文件里生成一个 smali 目录，还有 diff.dex, diff.apatch 文件。

能够找到你的输出文件确认下。

```
        }
        DiffInfo info = new DexDiffer().diff(this.from, this.to);

        this.classes = buildCode(smaliDir, dexFile, info);
                                http://blog.csdn.net/
        build(outFile, dexFile);

        release(this.out, dexFile, outFile);
```

DiffInfo 相当于一个存储新包和旧包差异信息的容器来，通过 diff 方法将二者的差异信息给 info.然后就是三个最重要的方法，buildCode(), build(),release()。我们接下来一个个的看下，他们到底为何这么重要。

Main.class | ApkPatch.class | Build.class

```
76        e.printStackTrace();
        }
      }

      private static Set<String> buildCode(File smaliDir, File dexFile, DiffInfo info)
        throws IOException, RecognitionException, FileNotFoundException
      {
83      Set<String> classes = new HashSet();
84      Set<DexBackedClassDef> list = new HashSet();
85      list.addAll(info.getAddedClasses());
86      list.addAll(info.getModifiedClasses());

88      baksmaliOptions options = new baksmaliOptions();

90      options.deodex = false;
91      options.noParameterRegisters = false;
92      options.useLocalsDirective = true;
93      options.useSequentialLabels = true;
94      options.outputDebugInfo = true;
95      options.addCodeOffsets = false;
96      options.jobs = -1;
97      options.noAccessorComments = false;
98      options.registerInfo = 0;
99      options.ignoreErrors = false;
100     options.inlineResolver = null;
101     options.checkPackagePrivateAccess = false;
102     if (!options.noAccessorComments) {
103       options.syntheticAccessorResolver = new SyntheticAccessorResolver(
104         list);
        }
106     ClassFileNameHandler outFileNameHandler = new ClassFileNameHandler(
107       smaliDir, ".smali");
108     ClassFileNameHandler inFileNameHandler = new ClassFileNameHandler(
109       smaliDir, ".smali");
110     DexBuilder dexBuilder = DexBuilder.makeDexBuilder();
113     for (DexBackedClassDef classDef : list)
        {
114       String className = classDef.getType();
115       baksmali.disassembleClass(classDef, outFileNameHandler, options);
116       File smaliFile = inFileNameHandler.getUniqueFilenameForClass(
117         TypeGenUtil.newType(className));
118       classes.add(TypeGenUtil.newType(className)
119         .substring(1, TypeGenUtil.newType(className).length() - 1)
120         .replace('/', '.'));
121       SmaliMod.assembleSmaliFile(smaliFile, dexBuilder, true, true);
        }
124     dexBuilder.writeTo(new FileDataStore(dexFile));

126     return classes;
      }
```


看到 baksmali 和 smali，反编译过 apk 的同学一定不陌生，这就是 dex 的打包工具和解包工具。关于这个详细就不深入了，有兴趣的同学能够深入了解下。这种方法的返回值将 DiffInfo 中新加入的 classes

和改动过的 classes 做了一个重命名。然后保存了起来，同一时候，将相关内容写入 smali 文件里。

为什么要进行重命名，事实上是为了防止和之前安装的 Dex 文件名称字冲突。

Main.class | ApkPatch.class | Build.class

```
  protected void release(File outDir, File dexFile, File outFile)
    throws NoSuchAlgorithmException, FileNotFoundException, IOException
  {
51  MessageDigest messageDigest = MessageDigest.getInstance("md5");
52  FileInputStream fileInputStream = new FileInputStream(dexFile);
53  byte[] buffer = new byte[8192];
54  int len = 0;
55  while ((len = fileInputStream.read(buffer)) > 0) {
56    messageDigest.update(buffer, 0, len);
    }
59  String md5 = HexUtil.hex(messageDigest.digest());
60  fileInputStream.close();
61  outFile.renameTo(new File(outDir, this.name + "-" + md5 + ".apatch"));
  }

  protected void build(File outFile, File dexFile)
    throws KeyStoreException, FileNotFoundException, IOException, NoSuchAlgorithmException, CertificateException, Unrecove
  {
67  KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
68  KeyStore.PrivateKeyEntry privateKeyEntry = null;
69  InputStream is = new FileInputStream(this.keystore);
70  keyStore.load(is, this.password.toCharArray());
71  privateKeyEntry = (KeyStore.PrivateKeyEntry)keyStore.getEntry(this.alias,
72    new KeyStore.PasswordProtection(this.entry.toCharArray()));

74  PatchBuilder builder = new PatchBuilder(outFile, dexFile,
75    privateKeyEntry, System.out);
76  builder.writeMeta(getMeta());
77  builder.sealPatch();
  }

  protected abstract Manifest getMeta();
}
```

接下来看看 build(outFile, dexFile)，首先从 keystone 里面获取应用相关签名，将 getMeta()中获取的 Manifest 内容写入"META-INF/PATCH.MF"文件里。getMeta()方法上面，实例化 PatchBuilder,然后调用 writeMeta(getMeta())。我们走进去先看看。

Main.class | ApkPatch.class | Build.class | PatchBuilder.class

```
package com.euler.patch.build;

import java.io.File;

public class PatchBuilder
{
  private SignedJarBuilder mBuilder;

  public PatchBuilder(File outFile, File dexFile, KeyStore.PrivateKeyEntry key, PrintStream verboseStrea
  {
    try
    {
22    this.mBuilder = new SignedJarBuilder(new FileOutputStream(outFile, false), key.getPrivateKey(),

24      (X509Certificate)key.getCertificate());
25    this.mBuilder.writeFile(dexFile, "classes.dex");
    }
    catch (Exception e)
    {
27    e.printStackTrace();
    }
  }
```

这个就是将 dexFile 和签名相关信息写入 classes.dex 文件里。能够有点蒙。

我们就看看 writeFile()方法。

```
public SignedJarBuilder(OutputStream out, PrivateKey key, X509Certificate certi
  throws IOException, NoSuchAlgorithmException
{
  this.mOutputJar = new JarOutputStream(new BufferedOutputStream(out));
  this.mOutputJar.setLevel(9);
  this.mKey = key;
  this.mCertificate = certificate;
  if ((this.mKey != null) && (this.mCertificate != null))
  {
    this.mManifest = new Manifest();
    Attributes main = this.mManifest.getMainAttributes();
    main.putValue("Manifest-Version", "1.0");
    main.putValue("Created-By", "1.0 (ApkPatch)");
    this.mBase64Encoder = new BASE64Encoder();
    this.mMessageDigest = MessageDigest.getInstance("SHA1");
  }
}
                                        http://blog.csdn.net/

public void writeFile(File inputFile, String jarPath)
  throws IOException
{
  FileInputStream fis = new FileInputStream(inputFile);
  try
  {
    JarEntry entry = new JarEntry(jarPath);
    entry.setTime(inputFile.lastModified());
    writeEntry(fis, entry);
  }
  finally
  {
    fis.close();
  }
}
```

SignedJarBuilder 的构造方法做了一些初始化和赋值操作。提到这个是方便可以理解 writeFile()这种方法。

writeFile 里面调用了 writeEntry(),我们看看它。

Main.class | ApkPatch.class | Build.class | PatchBuilder.class | SignedJarBuilder.class

```
238         this.mOutputJar.close();
          }
          catch (IOException localIOException) {}
        }
      }

      private void writeEntry(InputStream input, JarEntry entry)
        throws IOException
      {
252     this.mOutputJar.putNextEntry(entry);
        int count;
255     while ((count = input.read(this.mBuffer)) != -1)
        {
          int count;
256       this.mOutputJar.write(this.mBuffer, 0, count);
258       if (this.mMessageDigest != null) {
259         this.mMessageDigest.update(this.mBuffer, 0, count);
          }
        }
263     this.mOutputJar.closeEntry();
264     if (this.mManifest != null)
        {
266       Attributes attr = this.mManifest.getAttributes(entry.getName());
267       if (attr == null)
          {
268         attr = new Attributes();
269         this.mManifest.getEntries().put(entry.getName(), attr);
          }
271       attr.putValue("SHA1-Digest", this.mBase64Encoder.encode(this.mMessageDigest.digest()));
        }
      }
```

这种方法就是从 input 输入流中读取 buffer 数据然后写入到 entry。然后联系到我上面提到的将 dexfile 和签名相关信息写入到 classes.dex 里面。应该能好理解点。

上面提了一大堆，我们的东西准备的差点儿相同了，如今就看看最后一个方法 ApkPatch release(this.out, dexFile, outFile)

Main.class | ApkPatch.class | Build.class | PatchBuilder.class | SignedJarBuilder.class

```
5       throw new RuntimeException("output path must be directory.");
      }
    }

    protected void release(File outDir, File dexFile, File outFile)
      throws NoSuchAlgorithmException, FileNotFoundException, IOException
    {
1     MessageDigest messageDigest = MessageDigest.getInstance("md5");
2     FileInputStream fileInputStream = new FileInputStream(dexFile);
3     byte[] buffer = new byte[8192];
4     int len = 0;
5     while ((len = fileInputStream.read(buffer)) > 0) {
6       messageDigest.update(buffer, 0, len);
      }
9     String md5 = HexUtil.hex(messageDigest.digest());
0     fileInputStream.close();
1     outFile.renameTo(new File(outDir, this.name + "-" + md5 + ".apatch"));
    }
```

这种方法就是将 dexFile 进行 md5 加密，把 build(outFile, dexFile);

函数中生成的 outFile 重命名。哈哈。看到”.patch”有没有非常激动！

！

我们的补丁包一開始的命名就是一长串。好了，到这里，补丁文件就生成了。接下来我们看看，怎么来使用它。

坚持就是胜利，立即你就要熬过头了...没办法。别人团队花了这么长时间做的，想分析就得花时间。

相关资料工具及 demo 下载地址：

http://pan.baidu.com/s/1hsdcs7a

4. RXJava （RxJava 的线程切换原理）

前言

很多项目使用流行的 Rxjava2 + Retrofit 搭建网络框架，Rxjava 现在已经发展到 Rxjava2，之前一直都只是再用 Rxjava，但从来没有了解下 Rxjava 的内部实现，接下来一步步来分析 Rxjava2 的源码，Rxjava2 分 Observable 和 Flowable 两种（无被压和有被压），我们今天先从简单的无背压的 observable 来分析。源码基于 rxjava:2.1.1。

一、Rxjava 如何创建事件源、发射事件、何时发射事件、如何将观察者和被观察者关联起来

简单的例子

先来段最简单的代码，直观的了解下整个 Rxjava 运行的完整流程。

```
1 private void doSomeWork() { 2 Observable<String> observable = Observable.create(new ObservableOnSubscribe<String>() { 3 @Override 4 public void subscribe(ObservableEmitter<String> e) throws Exception { 5 e.onNext("a"); 6 e.onComplete(); 7 } 8 }); 9 Observer observer = new Observer<String>() {10 11 @Override12 public void onSubscribe(Disposable d) {13 Log.i("lx", " onSubscribe : " + d.isDisposed());14 }15 16 @Override17 public void onNext(String str) {18 Log.i("lx", " onNext : " + str);19 }20 21 @Override22 public void onError(Throwable e) {23 Log.i("lx", " onError : " + e.getMessage());24 }25 26 @Override27 public void onComplete() {28 Log.i("lx", " onComplete");29 }30 };31 observable.subscribe(observer);32 }
```

上面代码之所以将 observable 和 observer 单独声明，最后再调用 observable.subscribe(observer);
是为了分步来分析：

1. 被观察者 Observable 如何生产事件的
2. 被观察者 Observable 何时生产事件的
3. 观察者 Observer 是何时接收到上游事件的
4. Observable 与 Observer 是如何关联在一起的

Observable

Observable 是数据的上游，即事件生产者

首先来分析事件是如何生成的，直接看代码 Observable.create()方法。

```
1 @SchedulerSupport(SchedulerSupport.NONE)2     public static <T> Observable<T>
create(ObservableOnSubscribe<T> source) {    // ObservableOnSubscribe 是个接口，
只包含 subscribe 方法，是事件生产的源头。3         ObjectHelper.requireNonNull(source,
"source is null"); // 判空 4         return RxJavaPlugins.onAssembly(new
ObservableCreate<T>(source));5     }
```

最重要的是 RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));这句代码。
继续跟踪进去

```
 1 /** 2      * Calls the associated hook function. 3      * @param <T> the value
type 4      * @param source the hook's input value 5      * @return the value
returned by the hook 6      */ 7     @SuppressWarnings({ "rawtypes", "unchecked" })
8     @NonNull 9     public static <T> Observable<T> onAssembly(@NonNull
Observable<T> source) {10         Function<? super Observable, ? extends
Observable> f = onObservableAssembly;11         if (f != null) {12
return apply(f, source);13         }14         return source;15     }
```

看注释，原来这个方法是个 hook function。通过调试得知静态对象 onObservableAssembly 默认为 null， 所以此方法直接返回传入的参数 source。
onObservableAssembly 可以通过静态方法 RxJavaPlugins. setOnObservableAssembly () 设置全局的 Hook 函数， 有兴趣的同学可以自己去试试。 这里暂且不谈，我们继续返回代码。
现在我们明白了:

```
1  Observable<String> observable=Observable.create(new
ObservableOnSubscribe<String>() {2     ...3     ...4 })
```

相当于：

```
1 Observable<String> observable=new ObservableCreate(new
ObservableOnSubscribe<String>() {2     ...3     ...4 }))
```

好了，至此我们明白了，事件的源就是 new ObservableCreate()对象，将 ObservableOnSubscribe 作为参数传递给 ObservableCreate 的构造函数。
事件是由接口 ObservableOnSubscribe 的 subscribe 方法上产的，至于何时生产事件，稍后再分析。

Observer

Observer 是数据的下游，即事件消费者
Observer 是个 interface，包含 ：

```
1    void onSubscribe(@NonNull Disposable d);2    void onNext(@NonNull T t);3
void onError(@NonNull Throwable e);4    void onComplete();
```

上游发送的事件就是再这几个方法中被消费的。上游何时发送事件、如何发送，稍后再表。

subscribe

重点来了，接下来最重要的方法来了：observable.subscribe(observer);
从这个方法的名字就知道，subscribe 是订阅，是将观察者(observer)与被观察者(observable)连接起来的方法。只有 subscribe 方法执行后，上游产生的事件才能被下游接收并处理。其实自然的方式应该是 observer 订阅(subscribe) observable，但这样会打断 rxjava 的链式结构。所以采用相反的方式。
接下来看源码，只列出关键代码

```
1 public final void subscribe(Observer<? super T> observer) { 2
ObjectHelper.requireNonNull(observer, "observer is null"); 3        ..... 4
observer = RxJavaPlugins.onSubscribe(this, observer); // hook ，默认直接返回
observer 5        ..... 6        subscribeActual(observer);  // 这个才是真正实现订
阅的方法。 7        ..... 8    } 9 10 // subscribeActual 是抽象方法，所以需要到实现
类中去看具体实现，也就是说实现是在上文中提到的 ObservableCreate 中 11 protected abstract
void subscribeActual(Observer<? super T> observer);
```

接下来我们来看 ObservableCreate.java：

```
1 public ObservableCreate(ObservableOnSubscribe<T> source) { 2
this.source = source;  // 事件源，生产事件的接口，由我们自己实现 3    } 4  5
@Override 6    protected void subscribeActual(Observer<? super T> observer) { 7
```

```
CreateEmitter<T> parent = new CreateEmitter<T>(observer); // 发射器 8
observer.onSubscribe(parent);  //直接回调了观察者的 onSubscribe 9 10        try {11
// 调用了事件源 subscribe 方法生产事件，同时将发射器传给事件源。 12            // 现在我们
明白了，数据源生产事件的 subscribe 方法只有在 observable.subscribe(observer)被执行 13
后才执行的。 换言之，事件流是在订阅后才产生的。14            //而 observable 被创建出来时
并不生产事件，同时也不发射事件。15          source.subscribe(parent);  16        }
catch (Throwable ex) {17            Exceptions.throwIfFatal(ex);18
parent.onError(ex);19        }20    }
```

现在我们明白了，数据源生产事件的 subscribe 方法只有在 observable.subscribe(observer)被执行后才执行的。 换言之，事件流是在订阅后才产生的。而 observable 被创建出来时并不生产事件，同时也不发射事件。

接下来我们再来看看事件是如何被发射出去，同时 observer 是如何接收到发射的事件的

CreateEmitter<T> parent = new CreateEmitter<T>(observer);

CreateEmitter 实现了 ObservableEmitter 接口，同时 ObservableEmitter 接口又继承了 Emitter 接口。

CreateEmitter 还实现了 Disposable 接口，这个 disposable 接口是用来判断是否中断事件发射的。

从名称上就能看出，这个是发射器，故名思议是用来发射事件的，正是它将上游产生的事件发射到下游的。

Emitter 是事件源与下游的桥梁。

CreateEmitter 主要包括方法：

```
1 void onNext(@NonNull T value);2    void onError(@NonNull Throwable error);3
void onComplete();4    public void dispose() ;5    public boolean isDisposed();
```

是不是跟 observer 的方法很像？

我们来看看 CreateEmitter 中这几个方法的具体实现：

只列出关键代码

```
 1 public void onNext(T t) { 2        if (!isDisposed()) { // 判断事件是否需要被
丢弃 3            observer.onNext(t); // 调用 Emitter 的 onNext,它会直接调用 observer
的 onNext 4        } 5      } 6      public void onError(Throwable t) { 7
if (!isDisposed()) { 8              try { 9                  observer.onError(t);
// 调用 Emitter 的 onError，它会直接调用 observer 的 onError10              } finally
{11                  dispose();  // 当 onError 被触发时,执行 dispose(), 后续 onNext,
onError,  onComplete 就不会继 12                                  续发射事件了
13              }14          }15      }16 17      @Override18
public void onComplete() {19          if (!isDisposed()) {20              try
```

```
{21                    observer.onComplete(); // 调用 Emitter 的 onComplete，它会直
接调用 observer 的 onComplete22                } finally {23
dispose();  // 当 onComplete 被触发时，也会执行 dispose()，后续 onNext，onError，
onComplete24                                    同样不会继续发射事件了
25                }26            }27        }
```

CreateEmitter 的 onError 和 onComplete 方法任何一个执行完都会执行 dispose()中断事件发射，所以 observer 中的 onError 和 onComplete 也只能有一个被执行。
现在终于明白了，事件是如何被发射给下游的。当订阅成功后，数据源 ObservableOnSubscribe 开始生产事件，调用 Emitter 的 onNext，onComplete 向下游发射事件，

Emitter 包含了 observer 的引用，又调用了 observer onNext，onComplete，这样下游 observer 就接收到了上游发射的数据。

总结

Rxjava 的流程大概是：

1. **Observable.create 创建事件源，但并不生产也不发射事件。**
2. **实现 observer 接口，但此时没有也无法接受到任何发射来的事件。**
3. **订阅 observable.subscribe(observer)， 此时会调用具体 Observable 的实现类中的 subscribeActual 方法，**
   此时会才会真正触发事件源生产事件，事件源生产出来的事件通过 Emitter 的 onNext，onError，onComplete 发射给 observer 对应的方法由下游 observer 消费掉。从而完成整个事件流的处理。

observer 中的 onSubscribe 在订阅时即被调用，并传回了 Disposable， observer 中可以利用 Disposable 来随时中断事件流的发射。

今天所列举的例子是最简单的一个事件处理流程，没有使用线程调度，Rxjava 最强大的就是异步时对线程的调度和随时切换观察者线程，未完待续。

上面分析了 Rxjava 是如何创建事件源，如何发射事件，何时发射事件，也清楚了上游和下游是如何关联起来的。
下面着重来分析下 Rxjava 强大的线程调度是如何实现的。

二、RxJava 的线程调度机制

简单的例子

```
1 private void doSomeWork() { 2 Observable.create(new
ObservableOnSubscribe<String>() { 3 @Override 4 public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
Thread.sleep(2000); 7 e.onNext("a"); 8
e.onComplete(); 9 }10 }).subscribe(new Observer<String>() {11
@Override12 public void onSubscribe(Disposable d) {13
Log.i("lx", " onSubscribe: " +
Thread.currentThread().getName());14 }15 @Override16
public void onNext(String str) {17 Log.i("lx", " onNext: " +
Thread.currentThread().getName());18 }19 @Override20
public void onError(Throwable e) {21 Log.i("lx", " onError: " +
Thread.currentThread().getName());22 }23 @Override24
public void onComplete() {25 Log.i("lx", " onComplete: " +
Thread.currentThread().getName());26 }27 });28 }
```

运行结果：

```
1 com.rxjava2.android.samples I/lx: onSubscribe: main2
com.rxjava2.android.samples I/lx: subscribe: main3
com.rxjava2.android.samples I/lx: onNext: main4 com.rxjava2.android.samples
I/lx: onComplete: main
```

因为此方法笔者是在 main 线程中调用的，所以没有进行线程调度的情况下，所有方法都运行在 main 线程中。但我们知道 Android 的 UI 线程是不能做网络操作，也不能做耗时操作，所以一般我们把网络或耗时操作都放在非 UI 线程中执行。接下来我们就来感受下 Rxjava 强大的线程调度能力。

```
1 private void doSomeWork() { 2 Observable.create(new
ObservableOnSubscribe<String>() { 3 @Override 4 public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
Thread.sleep(2000); 7 e.onNext("a"); 8
e.onComplete(); 9 }10 }).subscribeOn(Schedulers.io()) //增加了
这一句 11 .subscribe(new Observer<String>() {12 @Override13
```

```
public void onSubscribe(Disposable d) {14                Log.i("lx", " onSubscribe:
" + Thread.currentThread().getName());15            }16            @Override17
public void onNext(String str) {18                Log.i("lx", " onNext: " +
Thread.currentThread().getName());19            }20            @Override21
public void onError(Throwable e) {22                Log.i("lx", " onError: " +
Thread.currentThread().getName());23            }24            @Override25
public void onComplete() {26                Log.i("lx", " onComplete: " +
Thread.currentThread().getName());27            }28        });29    }
```

运行结果：

```
1 com.rxjava2.android.samples I/lx:  onSubscribe: main2
com.rxjava2.android.samples I/lx:  subscribe: RxCachedThreadScheduler-13
com.rxjava2.android.samples I/lx:  onNext: RxCachedThreadScheduler-14
com.rxjava2.android.samples I/lx:  onComplete: RxCachedThreadScheduler-1
```

只增加了 subscribeOn 这一句代码， 就发生如此神奇的现象，除了 onSubscribe 方法还运行在 main 线程（订阅发生的线程）其它方法全部都运行在一个名为 RxCachedThreadScheduler-1 的线程中。我们来看看 rxjava 是怎么完成这个线程调度的。

线程调度 subscribeOn

首先我们先分析下 Schedulers.io()这个东东。

```
1  @NonNull2    public static Scheduler io() {3        return
RxJavaPlugins.onIoScheduler(IO); // hook function4        // 等价于 5
return IO;6    }
```

再看看 IO 是什么， IO 是个 static 变量，初始化的地方是

```
1 IO = RxJavaPlugins.initIoScheduler(new IOTask()); // 又是 hook function2 // 等
价于 3 IO = callRequireNonNull(new IOTask());4 // 等价于 5 IO = new IOTask().call();
```

继续看看 IOTask

```
1 static final class IOTask implements Callable<Scheduler> {2        @Override3
public Scheduler call() throws Exception {4            return IoHolder.DEFAULT;5
// 等价于 6            return new IoScheduler();7        }8    }
```

代码层次很深，为了便于记忆，我们再回顾一下：

```
1 Schedulers.io()等价于 new IoScheduler()2 3     // Schedulers.io()等价于 4
@NonNull5     public static Scheduler io() {6         return new
IoScheduler();7     }
```

好了，排除了其他干扰代码，接下来看看 IoScheduler()是什么东东了

IoScheduler 看名称就知道是个 IO 线程调度器，根据代码注释得知，它就是一个用来创建和缓存线程的线程池。看到这个豁然开朗了，原来 Rxjava 就是通过这个调度器来调度线程的，至于具体怎么实现我们接着往下看

```
 1 public IoScheduler() { 2         this(WORKER_THREAD_FACTORY); 3     } 4      5
public IoScheduler(ThreadFactory threadFactory) { 6         this.threadFactory =
threadFactory; 7         this.pool = new AtomicReference<CachedWorkerPool>(NONE);
8         start(); 9     }10 11     @Override12     public void start() {13
CachedWorkerPool update = new CachedWorkerPool(KEEP_ALIVE_TIME, KEEP_ALIVE_UNIT,
threadFactory);14         if (!pool.compareAndSet(NONE, update)) {15
update.shutdown();16         }17     }18     19     CachedWorkerPool(long
keepAliveTime, TimeUnit unit, ThreadFactory threadFactory) {20
this.keepAliveTime = unit != null ? unit.toNanos(keepAliveTime) : 0L;21
this.expiringWorkerQueue = new ConcurrentLinkedQueue<ThreadWorker>();22
this.allWorkers = new CompositeDisposable();23             this.threadFactory =
threadFactory;24 25             ScheduledExecutorService evictor = null;26
Future<?> task = null;27             if (unit != null) {28                 evictor
= Executors.newScheduledThreadPool(1, EVICTOR_THREAD_FACTORY);29
task = evictor.scheduleWithFixedDelay(this, this.keepAliveTime,
this.keepAliveTime, TimeUnit.NANOSECONDS);30             }31
evictorService = evictor;32             evictorTask = task;33         }
```

从上面的代码可以看出，new IoScheduler()后 Rxjava 会创建 CachedWorkerPool 的线程池，同时也创建并运行了一个名为 RxCachedWorkerPoolEvictor 的清除线程，主要作用是清除不再使用的一些线程。

但目前只创建了线程池并没有实际的 thread，所以 Schedulers.io()相当于只做了线程调度的前期准备。

OK，终于可以开始分析 Rxjava 是如何实现线程调度的。回到 Demo 来看 subscribeOn()方法的内部实现：

```
1 public final Observable<T> subscribeOn(Scheduler scheduler) {2
ObjectHelper.requireNonNull(scheduler, "scheduler is null");3        return
RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this,
scheduler));4     }
```

很熟悉的代码 RxJavaPlugins.onAssembly,上一篇已经分析过这个方法，就是个 hook function， 等价于直接 return new ObservableSubscribeOn<T>(this, scheduler);， 现在知道了这里的 scheduler 其实就是 IoScheduler。

跟踪代码进入 ObservableSubscribeOn，
可以看到这个 ObservableSubscribeOn 继承自 Observable，并且扩展了一些属性，增加了 scheduler。 各位看官，这不就是典型的装饰模式嘛，Rxjava 中大量用到了装饰模式，后面还会经常看到这种 wrap 类。

上篇文章我们已经知道了 Observable.subscribe()方法最终都是调用了对应的实现类的 subscribeActual 方法。我们重点分析下 subscribeActual:

```
 1 @Override 2     public void subscribeActual(final Observer<? super T> s) { 3
final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s); 4  5
// 没有任何线程调度，直接调用的，所以下游的 onSubscribe 方法没有切换线程，  6         //本
文 demo 中下游就是观察者，所以我们明白了为什么只有 onSubscribe 还运行在 main 线程 7
s.onSubscribe(parent); 8  9
parent.setDisposable(scheduler.scheduleDirect(new
SubscribeTask(parent)));10     }
```

SubscribeOnObserver 也是装饰模式的体现， 是对下游 observer 的一个 wrap，只是添加了 Disposable 的管理。

接下来分析最重要的 scheduler.scheduleDirect(new SubscribeTask(parent))

```
 1 // 这个类很简单，就是一个 Runnable，最终运行上游的 subscribe 方法 2     final class
SubscribeTask implements Runnable { 3        private final SubscribeOnObserver<T>
parent; 4  5        SubscribeTask(SubscribeOnObserver<T> parent) { 6
this.parent = parent; 7        } 8  9        @Override10        public void run()
{11            source.subscribe(parent);12        }13    }14    @NonNull15
public Disposable scheduleDirect(@NonNull Runnable run, long delay, @NonNull
TimeUnit unit) {16        // IoSchedular 中的 createWorker()17        final
Worker w = createWorker();18        // hook decoratedRun=run;19        final
Runnable decoratedRun = RxJavaPlugins.onSchedule(run);20        // decoratedRun
的 wrap，增加了 Dispose 的管理 21        DisposeTask task = new
```

```
DisposeTask(decoratedRun, w);22         // 线程调度 23         w.schedule(task,
delay, unit);24 25         return task;26     }
```

回到 IoSchedular

```
 1 public Worker createWorker() { 2         // 工作线程是在此时创建的 3         return
new EventLoopWorker(pool.get()); 4     } 5      6     public Disposable
schedule(@NonNull Runnable action, long delayTime, @NonNull TimeUnit unit) { 7
if (tasks.isDisposed()) { 8                 // don't schedule, we are unsubscribed
9                 return EmptyDisposable.INSTANCE;10             }11             //
action 中就包含上游 subscribe 的 runnable12             return
threadWorker.scheduleActual(action, delayTime, unit, tasks);13         }
```

最终线程是在这个方法内调度并执行的。

```
 1 public ScheduledRunnable scheduleActual(final Runnable run, long delayTime,
@NonNull TimeUnit unit, @Nullable DisposableContainer parent) { 2         //
decoratedRun = run, 包含上游 subscribe 方法的 runnable 3         Runnable
decoratedRun = RxJavaPlugins.onSchedule(run); 4  5         // decoratedRun 的 wrap,
增加了 dispose 的管理 6         ScheduledRunnable sr = new
ScheduledRunnable(decoratedRun, parent); 7  8         if (parent != null) { 9
if (!parent.add(sr)) {10                 return sr;11             }12         }13 14
// 最终 decoratedRun 被调度到之前创建或从线程池中取出的线程，15         // 也就是说在
RxCachedThreadScheduler-x 运行 16         Future<?> f;17         try {18
if (delayTime <= 0) {19                 f =
executor.submit((Callable<Object>)sr);20             } else {21                 f =
executor.schedule((Callable<Object>)sr, delayTime, unit);22             }23
sr.setFuture(f);24         } catch (RejectedExecutionException ex) {25
if (parent != null) {26                 parent.remove(sr);27             }28
RxJavaPlugins.onError(ex);29         }30 31         return sr;32     }
```

至此我们终于明白了 Rxjava 是如何调度线程并执行的，通过 subscribeOn 方法将上游生产事件的方法运行在指定的调度线程中。

```
1 com.rxjava2.android.samples I/lx:  onSubscribe: main2
com.rxjava2.android.samples I/lx:  subscribe: RxCachedThreadScheduler-13
```

```
com.rxjava2.android.samples I/lx:  onNext: RxCachedThreadScheduler-14
com.rxjava2.android.samples I/lx:  onComplete: RxCachedThreadScheduler-1
```

从上面的运行结果来看，因为上游生产者已被调度到 RxCachedThreadScheduler-1 线程中，同时发射事件并没有切换线程，所以发射后消费事件的 onNext onErro onComplete 也在 RxCachedThreadScheduler-1 线程中。

总结

1. Schedulers.io()等价于 new IoScheduler()。
2. new IoScheduler() Rxjava 创建了线程池，为后续创建线程做准备，同时创建并运行了一个清理线程 RxCachedWorkerPoolEvictor，定期执行清理任务。
3. subscribeOn()返回一个 ObservableSubscribeOn 对象，它是 Observable 的一个装饰类，增加了 scheduler。
4. 调用 subscribe()方法，在这个方法调用后，subscribeActual()被调用，才真正执行了 IoSchduler 中的 createWorker()创建线程并运行，最终将上游 Observable 的 subscribe()方法调度到新创建的线程中运行。

现在了解了被观察者执行线程是如何被调度到指定线程中执行的，但很多情况下，我们希望观察者（事件下游）处理事件最好在 UI 线程执行，比如更新 UI 操作等。下面分析下游何时调度，如何调度由于篇幅问题。

三、Rxjava 如何对观察者线程进行调度

简单的例子

```
1 private void doSomeWork() { 2        Observable.create(new
ObservableOnSubscribe<String>() { 3           @Override 4           public void
subscribe(ObservableEmitter<String> e) throws Exception { 5
Log.i("lx", " subscribe: " + Thread.currentThread().getName()); 6
e.onNext("a"); 7              e.onComplete(); 8           }
9       }).subscribeOn(Schedulers.io())10          .observeOn(AndroidSchedule
rs.mainThread())11          .subscribe(new Observer<String>() {12
@Override13           public void onSubscribe(Disposable d) {14
Log.i("lx", " onSubscribe: " +
Thread.currentThread().getName());15           }16           @Override17
public void onNext(String str) {18              Log.i("lx", " onNext: " +
Thread.currentThread().getName());19           }20           @Override21
public void onError(Throwable e) {22              Log.i("lx", " onError: " +
```

```
Thread.currentThread().getName());23            }24            @Override25
public void onComplete() {26                Log.i("lx", " onComplete: " +
Thread.currentThread().getName());27            }28        });29    }
```

看看运行结果：

```
1 com.rxjava2.android.samples I/lx:  onSubscribe: main2
com.rxjava2.android.samples I/lx:  subscribe: RxCachedThreadScheduler-13
com.rxjava2.android.samples I/lx:  onNext: main4 com.rxjava2.android.samples
I/lx:  onComplete: main
```

从结果可以看出，事件的生产线程运行在 RxCachedThreadScheduler-1 中，而事件的消费线程则被调度到了 main 线程中。关键代码是因为这句.observeOn(AndroidSchedulers.mainThread())。下面我们着重分析下这句代码都做了哪些事情。

AndroidSchedulers.mainThread()

先来看看 AndroidSchedulers.mainThread()是什么？贴代码

```
1  /** A {@link Scheduler} which executes actions on the Android main thread. */2
public static Scheduler mainThread() {3        return
RxAndroidPlugins.onMainThreadScheduler(MAIN_THREAD);4    }
```

注释已经说的很明白了，是一个在主线程执行任务的 scheduler，接着看

```
 1 private static final Scheduler MAIN_THREAD =
RxAndroidPlugins.initMainThreadScheduler( 2           new Callable<Scheduler>()
{ 3              @Override public Scheduler call() throws Exception { 4
return MainHolder.DEFAULT; 5               } 6           }); 7            8
public static Scheduler initMainThreadScheduler(Callable<Scheduler> scheduler)
{ 9    if (scheduler == null) {10        throw new
NullPointerException("scheduler == null");11    }12
Function<Callable<Scheduler>, Scheduler> f = onInitMainThreadHandler;13    if
(f == null) {14        return callRequireNonNull(scheduler);15    }16    return
applyRequireNonNull(f, scheduler);17    }
```

代码很简单，这个 AndroidSchedulers.mainThread()想当于 new HandlerScheduler(new Handler(Looper.getMainLooper())),原来是利用 Android 的 Handler 来调度到 main 线程的。

我们再看看 HandlerScheduler，它与我们上节分析的 IOScheduler 类似，都是继承自 Scheduler,所以 AndroidSchedulers.mainThread()其实就是是创建了一个运行在 main thread 上的 scheduler。
好了，我们再回过头来看 observeOn 方法。

observeOn

```
1 public final Observable<T> observeOn(Scheduler scheduler) { 2        return observeOn(scheduler, false, bufferSize()); 3    } 4     5    public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bufferSize) { 6        ObjectHelper.requireNonNull(scheduler, "scheduler is null"); 7 ObjectHelper.verifyPositive(bufferSize, "bufferSize"); 8        return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this, scheduler, delayError, bufferSize)); 9    }10
```

重点是这个 new ObservableObserveOn,看名字是不是有种似成相识的感觉，还记得上篇的 ObservableSubscribeOn 吗？ 它俩就是亲兄弟，是继承自同一个父类。

重点还是这个方法，我们前文已经提到了，Observable 的 subscribe 方法最终都是调用 subscribeActual 方法。下面看看这个方法的实现：

```
1    @Override 2    protected void subscribeActual(Observer<? super T> observer) { 3        // scheduler 就是前面提到的 HandlerScheduler，所以进入 else 分支 4 if (scheduler instanceof TrampolineScheduler) { 5 source.subscribe(observer); 6        } else { 7            // 创建 HandlerWorker 8            Scheduler.Worker w = scheduler.createWorker(); 9            // 调用上游 Observable 的 subscribe，将订阅向上传递 10            source.subscribe(new ObserveOnObserver<T>(observer, w, delayError, bufferSize));11        }12    }
```

从上面代码可以看到使用了 ObserveOnObserver 类对 observer 进行装饰，好了，我们再来看看 ObserveOnObserver。

我们已经知道了，事件源发射的事件，是通过 observer 的 onNext,onError,onComplete 发射到下游的。所以看看 ObserveOnObserver 的这三个方法是如何实现的。
由于篇幅问题，我们只分析 onNext 方法，onError 和 onComplete 方法有兴趣的同学可以自己分析下。

```
1 @Override 2     public void onNext(T t) { 3         if (done) { 4             return;
5         } 6         7         // 如果是非异步方式，将上游发射的时间加入到队列 8
if (sourceMode != QueueDisposable.ASYNC) { 9
queue.offer(t);10         }11         schedule();12     }13     14     void
schedule() {15         // 保证只有唯一任务在运行 16         if (getAndIncrement() == 0)
{17             // 调用的就是 HandlerWorker 的 schedule 方法 18
worker.schedule(this);19         }20     }21     22         @Override23
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {24
if (run == null) throw new NullPointerException("run == null");25             if
(unit == null) throw new NullPointerException("unit == null");26 27             if
(disposed) {28                 return Disposables.disposed();29             }30 31
run = RxJavaPlugins.onSchedule(run);32 33             ScheduledRunnable scheduled
= new ScheduledRunnable(handler, run);34 35             Message message =
Message.obtain(handler, scheduled);36             message.obj = this; // Used as
token for batch disposal of this worker's runnables.37 38
handler.sendMessageDelayed(message, Math.max(0L, unit.toMillis(delay)));39 40
// Re-check disposed state for removing in case we were racing a call to dispose().41
if (disposed) {42                 handler.removeCallbacks(scheduled);43
return Disposables.disposed();44             }45 46             return
scheduled;47         }
```

schedule 方法将传入的 run 调度到对应的 handle 所在的线程来执行，这个例子里就是有 main 线程来完成。 再回去看看前面传入的 run 吧。

回到 ObserveOnObserver 中的 run 方法：

```
1 @Override 2     public void run() { 3         // 此例子中代码不会进入这个分支，至于
这个 drainFused 是什么，后面章节再讨论。 4         if (outputFused) { 5
drainFused(); 6         } else { 7             drainNormal(); 8         } 9     }10
11     void drainNormal() {12         int missed = 1;13 14         final
SimpleQueue<T> q = queue;15         final Observer<? super T> a = actual;16 17
for (;;) {18             if (checkTerminated(done, q.isEmpty(), a)) {19
return;20             }21 22             for (;;) {23                 boolean d =
done;24                 T v;25 26                 try {27                     // 从队列
中 queue 中取出事件 28                     v = q.poll();29                 } catch
(Throwable ex) {30                     Exceptions.throwIfFatal(ex);31
s.dispose();32                     q.clear();33
a.onError(ex);34                     worker.dispose();35
return;36                 }37                 boolean empty = v == null;38 39
```

```
if (checkTerminated(d, empty, a)) {40
return;41                }42 43                if (empty) {44
break;45                }46                //调用下游 observer 的 onNext 将事件 v 发射出
去 47                a.onNext(v);48                }49 50                missed =
addAndGet(-missed);51                if (missed == 0) {52
break;53                }54        }55    }
```

至此我们明白了 RXjava 是如何调度消费者线程了。

消费者线程调度流程概括

Rxjava 调度消费者现在的流程，以 observeOn(AndroidSchedulers.mainThread())为例。

1. AndroidSchedulers.mainThread()先创建一个包含 handler 的 Scheduler, 这个 handler 是主线程的 handler。
2. observeOn 方法创建 ObservableObserveOn,它是上游 Observable 的一个装饰类，其中包含前面创建的 Scheduler 和 bufferSize 等.
3. 当订阅方法 subscribe 被调用后，ObservableObserveOn 的 subscribeActual 方法创建 Scheduler.Worker 并调用上游的 subscribe 方法，同时将自身接收的参数'observer'用装饰类 ObserveOnObserver 装饰后传递给上游。
4. 当上游调用被 ObserveOnObserver 的 onNext、onError 和 onComplete 方法时，ObserveOnObserver 将上游发送的事件通通加入到队列 queue 中，然后再调用 scheduler 将处理事件的方法调度到对应的线程中（本例会调度到 main thread）。 处理事件的方法将 queue 中保存的事件取出来，调用下游原始的 observer 再发射出去。
5. 经过以上流程，下游处理事件的消费者线程就运行在了 observeOn 调度后的 thread 中。

总结

经过前面两节的分析，我们已经明白了 Rxjava 是如何对线程进行调度的。

- Rxjava 的 subscribe 方法是由下游一步步向上游进行传递的。会调用上游的 subscribe，直到调用到事件源。
  如： `source.subscribe(xxx);`

而上游的 source 往往是经过装饰后的 Observable, Rxjava 就是利用 ObservableSubscribeOn 将 subscribe 方法调度到了指定线程运行，生产者线程最终会运行在被调度后的线程中。但多次调用 subscribeOn 方法会怎么样呢？ 我们知道因为 subscribe 方法是由下而上传递的，所以事件源的生产者线程最终都只会运行在第一次执行 subscribeOn 所调度的线程中，换句话就是多次调用 subscribeOn 方法，只有第一次有效。

- Rxjava 发射事件是由上而下发射的，上游的 onNext、onError、onComplete 方法会调用下游传入的 observer 的对应方法。往往下游传递的 observer 对象也是经过装饰后的 observer 对象。Rxjava 就是利用 ObserveOnObserver 将执行线程调度后，再调用下游对应的 onNext、onError、onComplete 方法，这样下游消费者就运行再了指定的线程内。 那么多次调用 observeOn 调度不同的线程会怎么样呢？ 因为事件是由上而下发射的，所以每次用 observeOn 切换完线程后，对下游的事件消费都有效，比如下游的 map 操作符。最终的事件消费线程运行在最后一个 observeOn 切换后线程中。
- 另外通过源码可以看到 onSubscribe 运行在 subscribe 的调用线程中,这个就不具体分析了。

5.Retrofit （Retrofit 在 OkHttp 上做了哪些封装？动态代理和静态代理的区别，是怎么实现的）

源码解析

从 Builder 模式创建实例开始看起

首先我们先从上面的第 4 步开始解析源码，有下面这段代码：

```
retrofit = new Retrofit.Builder()

        .baseUrl(GankConfig.HOST)

        .addConverterFactory(GsonConverterFactory.create(date_gson))//添加一个转换器，将 gson 数据转换为 bean 类

        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())//添加一个适配器，与 RxJava 配合使用

        .build();
```

很明显这个是使用了 Builder 模式，接下来我们一步一步来看里面做了什么？首先是 Builder()。

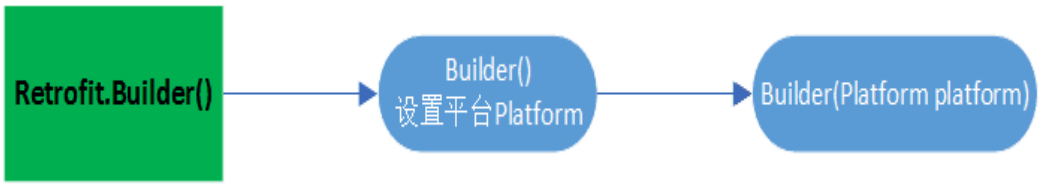
```
public Builder() {

    this(Platform.get());

}


Builder(Platform platform) {

    this.platform = platform;
```

```
    //添加转换器，请见下面关于 addConverterFactory()的讲解
    converterFactories.add(new BuiltInConverters());
  }
```

构造方法中的参数是 Platform 的静态方法 get()，接下来就看看 get()。

```
private static final Platform PLATFORM = findPlatform();

static Platform get() {
  return PLATFORM;
}

private static Platform findPlatform() {
  try {
    Class.forName("android.os.Build");
    if (Build.VERSION.SDK_INT != 0) {
      return new Android();
    }
  } catch (ClassNotFoundException ignored) {
  }
  try {
    Class.forName("java.util.Optional");
    return new Java8();
  } catch (ClassNotFoundException ignored) {
  }
  return new Platform();
}
}
```

可以看到，Retrofit 支持多平台，包括 Android 与 JAVA8，它会根据不同的平台设置不同的线程池。先来看看到目前为止我们分析到哪里了



接下来看一下 baseUrl()方法。

```
public Builder baseUrl(String baseUrl) {
    checkNotNull(baseUrl, "baseUrl == null");
    HttpUrl httpUrl = HttpUrl.parse(baseUrl);
    if (httpUrl == null) {
      throw new IllegalArgumentException("Illegal URL: " + baseUrl);
    }
    return baseUrl(httpUrl);
  }
```

很容易理解，baseUrl()是配置服务器的地址的，如果为空，那么就会抛出异常。

接着是 addConverterFactory()

```
private final List<Converter.Factory> converterFactories = new ArrayList<>();
public Builder addConverterFactory(Converter.Factory factory) {
    converterFactories.add(checkNotNull(factory, "factory == null"));
    return this;
}
```

大家是不是还记得刚才在 Builder()方法初始化的时候，有这样一行代码：

```
converterFactories.add(new BuiltInConverters());
```

可以看到，converterFactories 在初始化的时候就已经添加了一个默认的 Converter，那我们手动添加的这个 GsonConverter 是干什么用的呢？

```
public final class GsonConverterFactory extends Converter.Factory {


  public static GsonConverterFactory create() {
    return create(new Gson());
  }
```

```
public static GsonConverterFactory create(Gson gson) {
return new GsonConverterFactory(gson);
}

private final Gson gson;

private GsonConverterFactory(Gson gson) {
if (gson == null) throw new NullPointerException("gson == null");
this.gson = gson;
}

@Override
public Converter<ResponseBody, ?> responseBodyConverter(Type type,
Annotation[] annotations,
    Retrofit retrofit) {
  TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
return new GsonResponseBodyConverter<>(gson, adapter);
}

@Override
public Converter<?, RequestBody> requestBodyConverter(Type type,
    Annotation[] parameterAnnotations, Annotation[] methodAnnotations,
Retrofit retrofit) {
  TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
return new GsonRequestBodyConverter<>(gson, adapter);
}
}
```

其实这个 Converter 主要的作用就是将 HTTP 返回的数据解析成 Java 对象，我们常见的网络传输数据有 Xml、Gson、protobuf 等等，而 GsonConverter 就是将 Gson 数据转换为我们的 Java 对象，而不用我们重新去解析这些 Gson 数据。

接着看 addCallAdapterFactory()

```
private final List<CallAdapter.Factory> adapterFactories = new ArrayList<>();

public Builder addCallAdapterFactory(CallAdapter.Factory factory) {
    adapterFactories.add(checkNotNull(factory, "factory == null"));
    return this;
}
```

可以看到，CallAdapter 同样也被一个 List 维护，也就是说用户可以添加多个 CallAdapter，那 Retrofit 总得有一个默认的吧，默认的是什么呢？请看接下来的 build()。

最后看一下 build()

```
public Retrofit build() {
  //检验 baseUrl
  if (baseUrl == null) {
    throw new IllegalStateException("Base URL required.");
  }

  //创建一个 call，默认情况下使用 okhttp 作为网络请求器
  okhttp3.Call.Factory callFactory = this.callFactory;
  if (callFactory == null) {
    callFactory = new OkHttpClient();
  }

  Executor callbackExecutor = this.callbackExecutor;
  if (callbackExecutor == null) {
    callbackExecutor = platform.defaultCallbackExecutor();
  }

  List<CallAdapter.Factory> adapterFactories = new
ArrayList<>(this.adapterFactories);
  //添加一个默认的 callAdapter

adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
```

```
  List<Converter.Factory> converterFactories = new
ArrayList<>(this.converterFactories);


  return new Retrofit(callFactory, baseUrl, converterFactories,
adapterFactories,
      callbackExecutor, validateEagerly);
  }
```

首先 Retrofit 会新建一个 call，其实质就是 OKHttp，作用就是网络请求器；接着在上一点中我们困惑的 callAdapter 也已经能够得到解决了，首先 Retrofit 有一个默认的 callAdapter，请看下面这段代码：

```
adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));


CallAdapter.Factory defaultCallAdapterFactory(Executor callbackExecutor) {
  if (callbackExecutor != null) {
    return new ExecutorCallAdapterFactory(callbackExecutor);
  }
  return DefaultCallAdapterFactory.INSTANCE;
}
final class ExecutorCallAdapterFactory extends CallAdapter.Factory {
  final Executor callbackExecutor;

  ExecutorCallAdapterFactory(Executor callbackExecutor) {
    this.callbackExecutor = callbackExecutor;
  }

  @Override
  public CallAdapter<?, ?> get(Type returnType, Annotation[] annotations,
Retrofit retrofit) {
    if (getRawType(returnType) != Call.class) {
      return null;
    }
    final Type responseType = Utils.getCallResponseType(returnType);
```

```
    return new CallAdapter<Object, Call<?>>() {
      @Override public Type responseType() {
        return responseType;
      }

      @Override public Call<Object> adapt(Call<Object> call) {
        return new ExecutorCallbackCall<>(callbackExecutor, call);
      }
    };
  }

  static final class ExecutorCallbackCall<T> implements Call<T> {
    final Executor callbackExecutor;
    final Call<T> delegate;

    ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
      this.callbackExecutor = callbackExecutor;
      this.delegate = delegate;
    }

    @Override public void enqueue(final Callback<T> callback) {
      if (callback == null) throw new NullPointerException("callback == null");

      delegate.enqueue(new Callback<T>() {
        @Override public void onResponse(Call<T> call, final Response<T> response)
{
          callbackExecutor.execute(new Runnable() {
            @Override public void run() {
              if (delegate.isCanceled()) {
                // Emulate OkHttp's behavior of throwing/delivering an IOException
on cancellation.
```

```
                callback.onFailure(ExecutorCallbackCall.this, new
IOException("Canceled"));
              } else {
                callback.onResponse(ExecutorCallbackCall.this, response);
              }
            }
          });
        }

        @Override public void onFailure(Call<T> call, final Throwable t) {
          callbackExecutor.execute(new Runnable() {
            @Override public void run() {
              callback.onFailure(ExecutorCallbackCall.this, t);
            }
          });
        }
      });
    }

    @Override public boolean isExecuted() {
      return delegate.isExecuted();
    }

    @Override public Response<T> execute() throws IOException {
      return delegate.execute();
    }

    @Override public void cancel() {
      delegate.cancel();
    }
```

```
    @Override public boolean isCanceled() {
      return delegate.isCanceled();
    }

    @SuppressWarnings("CloneDoesntCallSuperClone") // Performing deep clone.
    @Override public Call<T> clone() {
      return new ExecutorCallbackCall<>(callbackExecutor, delegate.clone());
    }

    @Override public Request request() {
      return delegate.request();
    }
  }
}
```

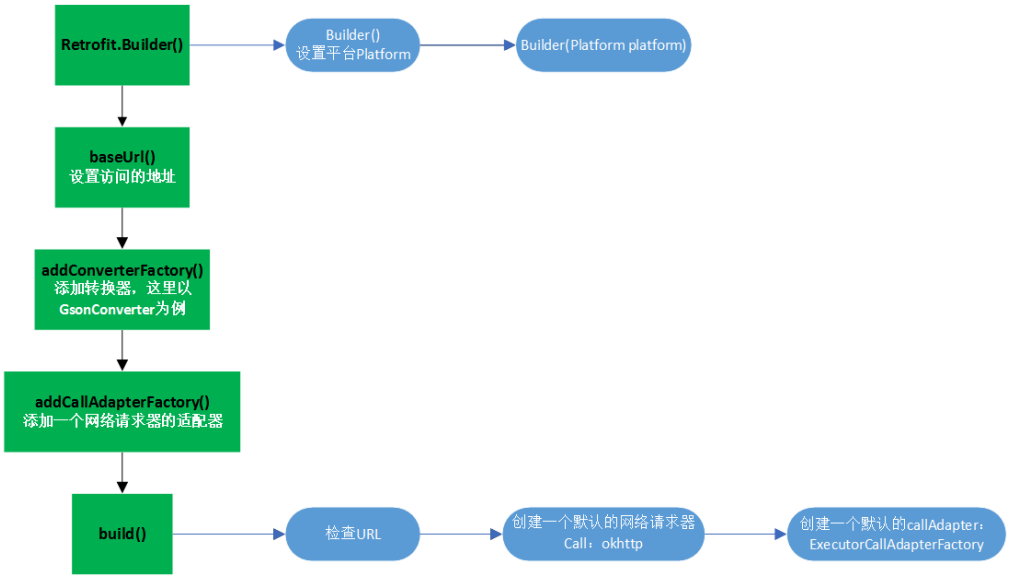
可以看到默认的 callAdapter 是 ExecutorCallAdapterFactory。callAdapter 其实也是运用了适配器模式，其实质就是网络请求器 Call 的适配器，而在 Retrofit 中 Call 就是指 OKHttp，那么 CallAdapter 就是用来将 OKHttp 适配给不同的平台的，在 Retrofit 中提供了四种 CallAdapter，分别如下：

- ExecutorCallAdapterFactory（默认使用）
- GuavaCallAdapterFactory
- Java8CallAdapterFactory
- RxJavaCallAdapterFactory

为什么要提供如此多的适配器呢？首先是易于扩展，例如用户习惯使用什么适配器，只需要添加即可使用；再者 RxJava 如此火热，因为其切换线程十分的方便，不需要手动使用 handler 切换线程，而 Retrofit 使用了支持 RxJava 的适配器之后，功能也会更加强大。

综上我们已经将使用 Builder 模式创建出来的 Retrofit 实例分析完毕了，我们只需要对相关的功能进行配置即可，Retrofit 负责接收我们配置的功能然后进行对象的初始化，这个也就是 Builder 模式

屏蔽掉创建对象的复杂过程的好处。现在我们再次用流程图来梳理一下刚才的思路。



网络请求接口的创建

我最初使用 Retrofit 的时候觉得有一个地方十分神奇，如下：

```
GankRetrofit gankRetrofit=retrofit.create(GankRetrofit.class);
```

```
GankData data= gankRetrofit.getDailyData(2017, 9, 1);
```

要想解惑，首先得对动态代理有所了解，如果你对动态代理还不是很清楚，请点击这里了解动态代理的原理，之后再接着往下看。

我们就以这里为切入点开始分析吧！首先是 create()

```
public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
      eagerlyValidateMethods(service);
    }
    //重点看这里
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[]
{ service },
        new InvocationHandler() {
          private final Platform platform = Platform.get();

          @Override public Object invoke(Object proxy, Method method, Object[]
args)
              throws Throwable {
            // If the method is a method from Object then defer to normal invocation.
            if (method.getDeclaringClass() == Object.class) {
              return method.invoke(this, args);
            }
            if (platform.isDefaultMethod(method)) {
              return platform.invokeDefaultMethod(method, service, proxy, args);
            }
            //下面就会讲到哦
            ServiceMethod<Object, Object> serviceMethod =
                (ServiceMethod<Object, Object>) loadServiceMethod(method);

            //下一小节讲到哦
            OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod,
args);

            //下两个小节讲哦
            return serviceMethod.callAdapter.adapt(okHttpCall);
```

```
        }

    });

}
```

我们主要看 Proxy.newProxyInstance 方法，它接收三个参数，第一个是一个类加载器，其实哪个类的加载器都无所谓，这里为了方便就选择了我们所定义的借口的类加载器;第二个参数是我们定义的接口的 class 对象，第三个则是一个 InvocationHandler 匿名内部类。
那大家应该会有疑问了，这个 newProxyInstance 到底有什么用呢？其实他就是通过动态代理生成了网络请求接口的代理类，代理类生成之后，接下来我们就可以使用 `ankRetrofit.getDailyData(2017, 9, 1);` 这样的语句去调用 getDailyData 方法，当我们调用这个方法的时候就会被动态代理拦截，直接进入 InvocationHandler 的 invoke 方法。下面就来讲讲它。

invoke 方法

它接收三个参数，第一个是动态代理，第二个是我们要调用的方法，这里就是指 `getDailyData`，第三个是一个参数数组，同样的这里就是指 `2017, 9, 1`，收到方法名和参数之后，紧接着会调用 loadServiceMethod 方法来生产过一个 ServiceMethod 对象，这里的一个 ServiceMethod 对象就对应我们在网络接口里定义的一个方法，相当于做了一层封装。接下来重点来看 loadServiceMethod 方法。

loadServiceMethod 方法

```
  ServiceMethod<?, ?> loadServiceMethod(Method method) {

    ServiceMethod<?, ?> result = serviceMethodCache.get(method);

    if (result != null) return result;


    synchronized (serviceMethodCache) {

      result = serviceMethodCache.get(method);

      if (result == null) {

        result = new ServiceMethod.Builder<>(this, method).build();

        serviceMethodCache.put(method, result);

      }

    }

    return result;

  }
```

它调用了 ServiceMethod 类，而 ServiceMethod 也使用了 Builder 模式，直接先看 Builder 方法。

```
    Builder(Retrofit retrofit, Method method) {

      this.retrofit = retrofit;
```

```
        //获取接口中的方法名
        this.method = method;

        //获取方法里的注解
        this.methodAnnotations = method.getAnnotations();

        //获取方法里的参数类型
        this.parameterTypes = method.getGenericParameterTypes();

        //获取接口方法里的注解内容
        this.parameterAnnotationsArray = method.getParameterAnnotations();
    }
```

再来看 build 方法

```
public ServiceMethod build() {
      callAdapter = createCallAdapter();
      responseType = callAdapter.responseType();
      if (responseType == Response.class || responseType ==
okhttp3.Response.class) {
        throw methodError("'"
            + Utils.getRawType(responseType).getName()
            + "' is not a valid response body type. Did you mean ResponseBody?");
      }
      responseConverter = createResponseConverter();

      for (Annotation annotation : methodAnnotations) {
        parseMethodAnnotation(annotation);
      }

      if (httpMethod == null) {
        throw methodError("HTTP method annotation is required (e.g., @GET, @POST,
etc.).");
```

```
    }


    if (!hasBody) {
      if (isMultipart) {
        throw methodError(
            "Multipart can only be specified on HTTP methods with request body (e.g., @POST).");
      }
      if (isFormEncoded) {
        throw methodError("FormUrlEncoded can only be specified on HTTP methods with "
            + "request body (e.g., @POST).");
      }
    }


    int parameterCount = parameterAnnotationsArray.length;
    parameterHandlers = new ParameterHandler<?>[parameterCount];
    for (int p = 0; p < parameterCount; p++) {
      Type parameterType = parameterTypes[p];
      if (Utils.hasUnresolvableType(parameterType)) {
        throw parameterError(p, "Parameter type must not include a type variable or wildcard: %s",
            parameterType);
      }


      Annotation[] parameterAnnotations = parameterAnnotationsArray[p];
      if (parameterAnnotations == null) {
        throw parameterError(p, "No Retrofit annotation found.");
      }
```

```
        parameterHandlers[p] = parseParameter(p, parameterType,
parameterAnnotations);

      }


      if (relativeUrl == null && !gotUrl) {
        throw methodError("Missing either @%s URL or @Url parameter.",
httpMethod);
      }
      if (!isFormEncoded && !isMultipart && !hasBody && gotBody) {
        throw methodError("Non-body HTTP method cannot contain @Body.");
      }
      if (isFormEncoded && !gotField) {
        throw methodError("Form-encoded method must contain at least one
@Field.");
      }
      if (isMultipart && !gotPart) {
        throw methodError("Multipart method must contain at least one @Part.");
      }


      return new ServiceMethod<>(this);
    }
```

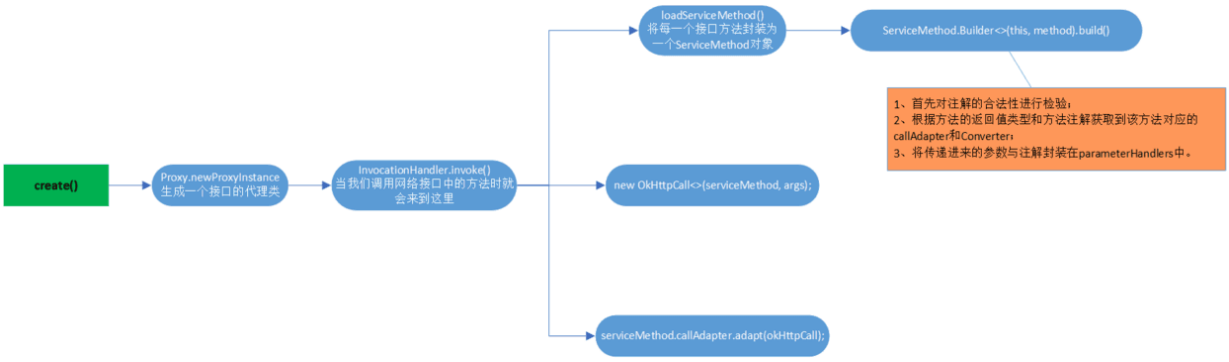
代码稍微有点长，但是思路很清晰，主要的工作有

1、首先对注解的合法性进行检验，例如，HTTP 的请求方法是 GET 还是 POST，如果不是就会抛出异常；

2、根据方法的返回值类型和方法注解从 Retrofit 对象的的 callAdapter 列表和 Converter 列表中分别获取到该方法对应的 callAdapter 和 Converter；

3、将传递进来的参数与注解封装在 parameterHandlers 中，为后面的网络请求做准备。

先用流程图梳理一下刚才的思路：



分析到这里，我们总算是明白了最初的两行代码原来干了这么多事情，J 神真的是流弊啊！接下来我们就来看一下网络请求部分。

使用 OkHttpCall 进行网络请求

回头看一下上一小节讲解 create 方法时我们有这一行代码：

```
OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args);
```

他将我们刚才得到的 serviceMethod 与我们实际传入的参数传递给了 OkHttpCall，接下来就来瞧瞧这个类做了些什么？

```
final class OkHttpCall<T> implements Call<T> {

  private final ServiceMethod<T, ?> serviceMethod;

  private final Object[] args;


  private volatile boolean canceled;


  // All guarded by this.

  private okhttp3.Call rawCall;

  private Throwable creationFailure; // Either a RuntimeException or IOException.

  private boolean executed;


  OkHttpCall(ServiceMethod<T, ?> serviceMethod, Object[] args) {

    this.serviceMethod = serviceMethod;

    this.args = args;

  }
```

```
}
```

很可惜，我们好像没有看到比较有用的东西，只是将传进来的参数进行了赋值，那我们就接着看 create 方法中的最后一行吧！

callAdapter 的使用

create 方法的最后一行是这样的：

```
return serviceMethod.callAdapter.adapt(okHttpCall);
```

最后是调用了 callAdapter 的 adapt 方法，上面我们讲到 Retrofit 在决定使用什么 callAdapter 的时候是看我们在接口中定义的方法的返回值的，而在我们的例子中使用的是 `RxJava2CallAdapter`，因此我们就直接看该类中的 adapt 方法吧！

```
@Override public Object adapt(Call<R> call) {
    Observable<Response<R>> responseObservable = isAsync
        ? new CallEnqueueObservable<>(call)
        : new CallExecuteObservable<>(call);

    Observable<?> observable;
    if (isResult) {
      observable = new ResultObservable<>(responseObservable);
    } else if (isBody) {
      observable = new BodyObservable<>(responseObservable);
    } else {
      observable = responseObservable;
    }

    if (scheduler != null) {
      observable = observable.subscribeOn(scheduler);
    }

    if (isFlowable) {
      return observable.toFlowable(BackpressureStrategy.LATEST);
    }
```

```
    if (isSingle) {
      return observable.singleOrError();
    }
    if (isMaybe) {
      return observable.singleElement();
    }
    if (isCompletable) {
      return observable.ignoreElements();
    }
    return observable;
  }
```

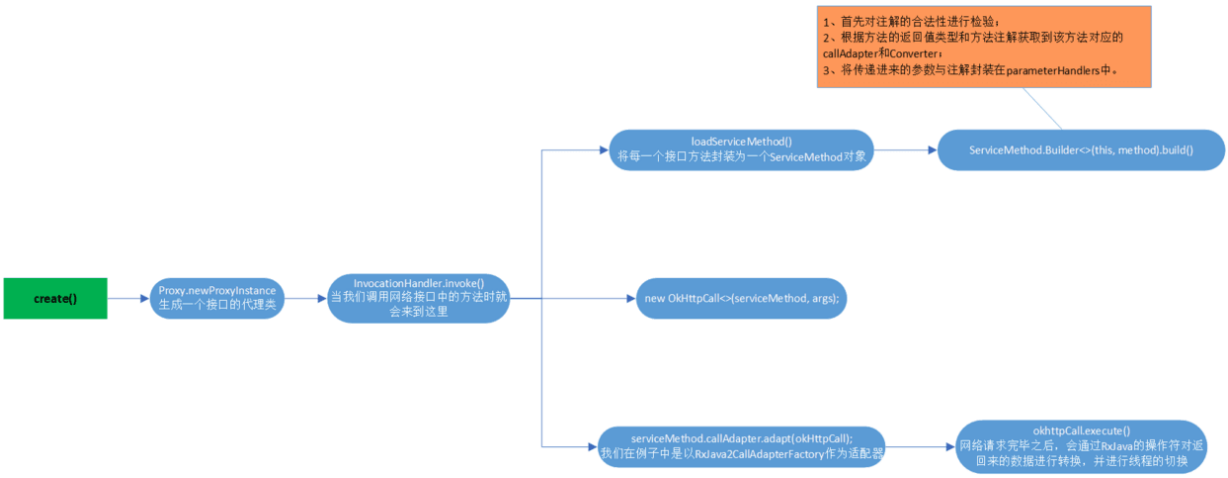
首先在 adapt 方法中会先判断是同步请求还是异步请求，这里我们以同步请求为例，直接看 CallExecuteObservable。

```
final class CallExecuteObservable<T> extends Observable<Response<T>> {
  private final Call<T> originalCall;

  CallExecuteObservable(Call<T> originalCall) {
    this.originalCall = originalCall;
  }

  @Override protected void subscribeActual(Observer<? super Response<T>>
observer) {
    // Since Call is a one-shot type, clone it for each new observer.
    Call<T> call = originalCall.clone();
    observer.onSubscribe(new CallDisposable(call));

    boolean terminated = false;
    try {
      //重点看这里
      Response<T> response = call.execute();
      if (!call.isCanceled()) {
```

```
        observer.onNext(response);
      }
      if (!call.isCanceled()) {
        terminated = true;
        observer.onComplete();
      }
    } catch (Throwable t) {
      Exceptions.throwIfFatal(t);
      if (terminated) {
        RxJavaPlugins.onError(t);
      } else if (!call.isCanceled()) {
        try {
          observer.onError(t);
        } catch (Throwable inner) {
          Exceptions.throwIfFatal(inner);
          RxJavaPlugins.onError(new CompositeException(t, inner));
        }
      }
    }
  }

  private static final class CallDisposable implements Disposable {
    private final Call<?> call;

    CallDisposable(Call<?> call) {
      this.call = call;
    }

    @Override public void dispose() {
      call.cancel();
```

```
    }

    @Override public boolean isDisposed() {
      return call.isCanceled();
    }
  }
}
```

在 subscribeActual 方法中去调用了 OKHttpCall 的 execute 方法开始进行网络请求，网络请求完毕之后，会通过 RxJava 的操作符对返回来的数据进行转换，并进行线程的切换，至此，Retrofit 的一次使用也就结束了。最后我们再用一张完整的流程图总结上述的几个过程。

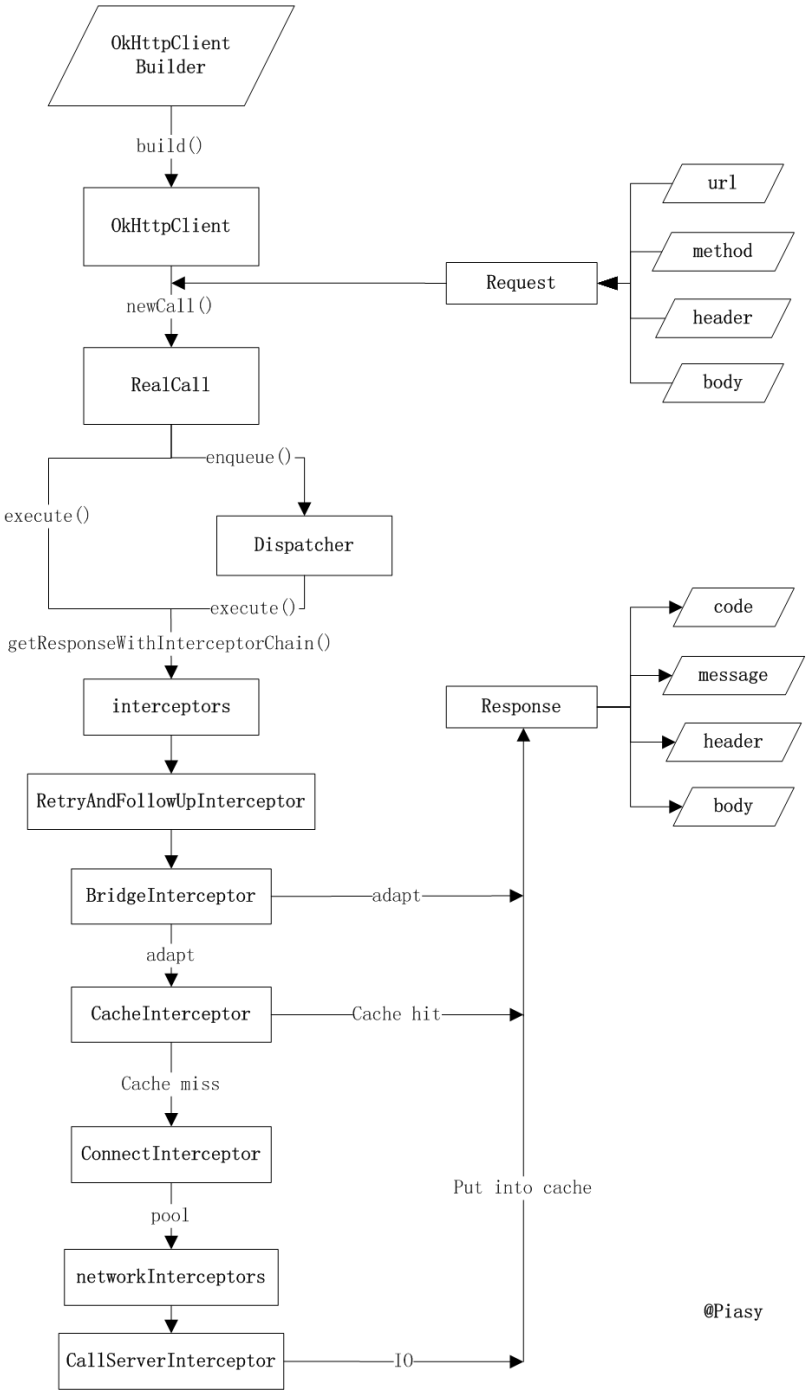


6. OkHttp

1，整体思路

从使用方法出发，首先是怎么使用，其次是我们使用的功能在内部是如何实现的，实现方案上有什么技巧，有什么范式。全文基本上是对 OkHttp 源码的一个分析与导读，非常建议大家下载 OkHttp 源码之后，跟着本文，过一遍源码。对于技巧和范式，由于目前我的功力还不到位，分析内容没多少，欢迎大家和我一起讨论。

首先放一张完整流程图（看不懂没关系，慢慢往后看）：



2，基本用例

来自 OkHttp 官方网站。

2.1，创建 OkHttpClient 对象

```
OkHttpClient client = new OkHttpClient();
```

咦，怎么不见 builder？莫急，且看其构造函数：

```
public OkHttpClient() {
  this(new Builder());
}
```

原来是方便我们使用，提供了一个“快捷操作”，全部使用了默认的配置。

OkHttpClient.Builder 类成员很多，后面我们再慢慢分析，这里先暂时略过：

```
public Builder() {
  dispatcher = new Dispatcher();
  protocols = DEFAULT_PROTOCOLS;
  connectionSpecs = DEFAULT_CONNECTION_SPECS;
  proxySelector = ProxySelector.getDefault();
  cookieJar = CookieJar.NO_COOKIES;
  socketFactory = SocketFactory.getDefault();
  hostnameVerifier = OkHostnameVerifier.INSTANCE;
  certificatePinner = CertificatePinner.DEFAULT;
  proxyAuthenticator = Authenticator.NONE;
  authenticator = Authenticator.NONE;
  connectionPool = new ConnectionPool();
  dns = Dns.SYSTEM;
  followSslRedirects = true;
  followRedirects = true;
  retryOnConnectionFailure = true;
  connectTimeout = 10_000;
  readTimeout = 10_000;
  writeTimeout = 10_000;
}
```

2.2，发起 HTTP 请求

```
String run(String url) throws IOException {
  Request request = new Request.Builder()
      .url(url)
      .build();

  Response response = client.newCall(request).execute();
  return response.body().string();
}
```

OkHttpClient 实现了 Call.Factory，负责根据请求创建新的 Call，在 拆轮子系列：拆 Retrofit 中我们曾和它发生过一次短暂的遭遇：

> callFactory 负责创建 HTTP 请求，HTTP 请求被抽象为了 okhttp3.Call 类，它表示一个已经准备好，可以随时执行的 HTTP 请求

那我们现在就来看看它是如何创建 Call 的：

```
/**
  * Prepares the {@code request} to be executed at some point in the future.
  */
@Override public Call newCall(Request request) {
  return new RealCall(this, request);
```

```
}
```

如此看来功劳全在 RealCall 类了，下面我们一边分析同步网络请求的过程，一边了解 RealCall 的具体内容。

2.2.1，同步网络请求

我们首先看 RealCall#execute：

```
@Override public Response execute() throws IOException {
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Executed");  // (1)
    executed = true;
  }
  try {
    client.dispatcher().executed(this);                                 // (2)
    Response result = getResponseWithInterceptorChain();                // (3)
    if (result == null) throw new IOException("Canceled");
    return result;
  } finally {
    client.dispatcher().finished(this);                                 // (4)  }
}
```

这里我们做了 4 件事：

1. 检查这个 call 是否已经被执行了，每个 call 只能被执行一次，如果想要一个完全一样的 call，可以利用 call#clone 方法进行克隆。

2. 利用 client.dispatcher().executed(this) 来进行实际执行，dispatcher 是刚才看到的 OkHttpClient.Builder 的成员之一，它的文档说自己是异步 HTTP 请求的执行策略，现在看来，同步请求它也有掺和。

3. 调用 getResponseWithInterceptorChain() 函数获取 HTTP 返回结果，从函数名可以看出，这一步还会进行一系列"拦截"操作。

4. 最后还要通知 dispatcher 自己已经执行完毕。

dispatcher 这里我们不过度关注，在同步执行的流程中，涉及到 dispatcher 的内容只不过是告知它我们的执行状态，比如开始执行了（调用 executed），比如执行完毕了（调用 finished），在异步执行流程中它会有更多的参与。

真正发出网络请求，解析返回结果的，还是 getResponseWithInterceptorChain：

```
private Response getResponseWithInterceptorChain() throws IOException {
  // Build a full stack of interceptors.
  List<Interceptor> interceptors = new ArrayList<>();
  interceptors.addAll(client.interceptors());
  interceptors.add(retryAndFollowUpInterceptor);
  interceptors.add(new BridgeInterceptor(client.cookieJar()));
  interceptors.add(new CacheInterceptor(client.internalCache()));
  interceptors.add(new ConnectInterceptor(client));
  if (!retryAndFollowUpInterceptor.isForWebSocket()) {
    interceptors.addAll(client.networkInterceptors());
  }
  interceptors.add(new CallServerInterceptor(
      retryAndFollowUpInterceptor.isForWebSocket()));

  Interceptor.Chain chain = new RealInterceptorChain(
```

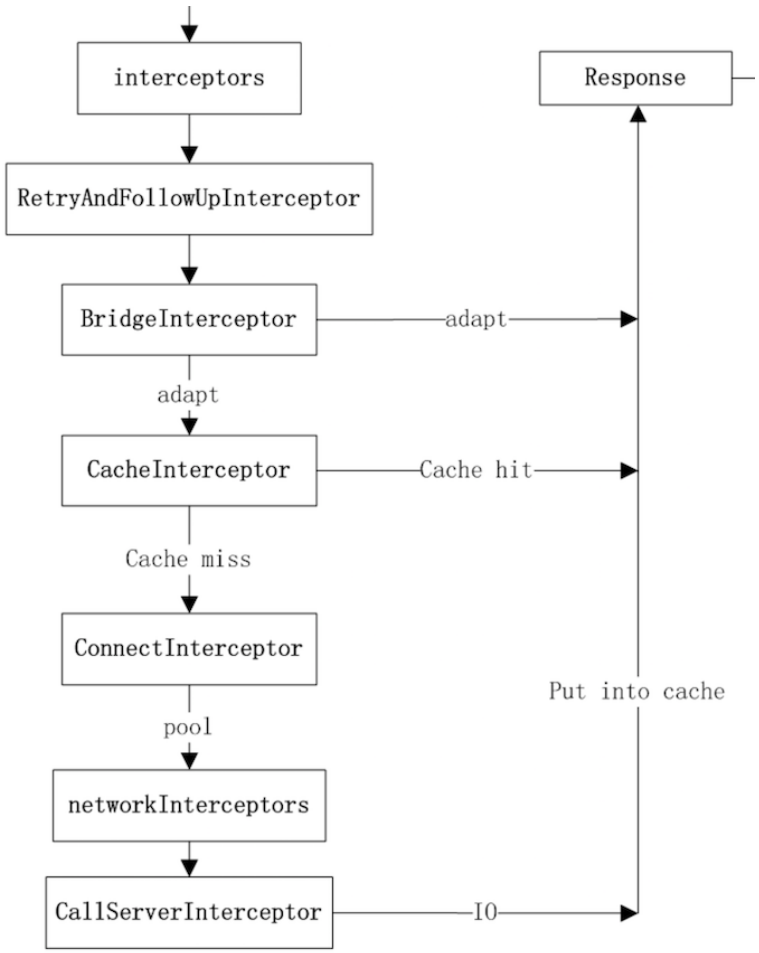
```
    interceptors, null, null, null, 0, originalRequest);
  return chain.proceed(originalRequest);
}
```

在 OkHttp 开发者之一介绍 OkHttp 的文章里面，作者讲到：

> the whole thing is just a stack of built-in interceptors.

可见 Interceptor 是 OkHttp 最核心的一个东西，不要误以为它只负责拦截请求进行一些额外的处理（例如 cookie），**实际上它把实际的网络请求、缓存、透明压缩等功能都统一了起来**，每一个功能都只是一个 Interceptor，它们再连接成一个 Interceptor.Chain，环环相扣，最终圆满完成一次网络请求。

从 getResponseWithInterceptorChain 函数我们可以看到，Interceptor.Chain 的分布依次是：



1. 在配置 OkHttpClient 时设置的 interceptors；

2. 负责失败重试以及重定向的 RetryAndFollowUpInterceptor；

3. 负责把用户构造的请求转换为发送到服务器的请求、把服务器返回的响应转换为用户友好的响应的 BridgeInterceptor；

4. 负责读取缓存直接返回、更新缓存的 CacheInterceptor；

5. 负责和服务器建立连接的 ConnectInterceptor；

6. 配置 OkHttpClient 时设置的 networkInterceptors；

7. 负责向服务器发送请求数据、从服务器读取响应数据的 CallServerInterceptor。

在这里，位置决定了功能，最后一个 Interceptor 一定是负责和服务器实际通讯的，重定向、缓存等一定是在实际通讯之前的。

责任链模式在这个 Interceptor 链条中得到了很好的实践（感谢 Stay 一语道破，自愧弗如）。

> 它包含了一些命令对象和一系列的处理对象，每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象。该模式还描述了往该处理链的末尾添加新的处理对象的方法。

对于把 Request 变成 Response 这件事来说，每个 Interceptor 都可能完成这件事，所以我们循着链条让每个 Interceptor 自行决定能否完成任务以及怎么完成任务（自力更生或者交给下一个 Interceptor）。这样一来，完成网络请求这件事就彻底从 RealCall 类中剥离了出来，简化了各自的责任和逻辑。两个字：优雅！

责任链模式在安卓系统中也有比较典型的实践，例如 view 系统对点击事件（TouchEvent）的处理，具体可以参考 Android 设计模式源码解析之责任链模式中相关的分析。

回到 OkHttp，在这里我们先简单分析一下 ConnectInterceptor 和 CallServerInterceptor，看看 OkHttp 是怎么进行和服务器的实际通信的。

2.2.1.1，建立连接：ConnectInterceptor

```
@Override public Response intercept(Chain chain) throws IOException {
  RealInterceptorChain realChain = (RealInterceptorChain) chain;
  Request request = realChain.request();
  StreamAllocation streamAllocation = realChain.streamAllocation();
```

```
  // We need the network to satisfy this request. Possibly for validating a
conditional GET.
  boolean doExtensiveHealthChecks = !request.method().equals("GET");
  HttpCodec httpCodec = streamAllocation.newStream(client,
doExtensiveHealthChecks);
  RealConnection connection = streamAllocation.connection();

  return realChain.proceed(request, streamAllocation, httpCodec, connection);
}
```

实际上建立连接就是创建了一个 HttpCodec 对象，它将在后面的步骤中被使用，

那它又是何方神圣呢？它是对 HTTP 协议操作的抽象，有两个实现：

Http1Codec 和 Http2Codec，顾名思义，它们分别对应 HTTP/1.1 和 HTTP/2 版本的实现。

在 Http1Codec 中，它利用 Okio 对 Socket 的读写操作进行封装，Okio 以后有机会再进行分析，现在让我们对它们保持一个简单地认识：它对 java.io 和 java.nio 进行了封装，让我们更便捷高效的进行 IO 操作。

而创建 HttpCodec 对象的过程涉及到 StreamAllocation、RealConnection，代码较长，这里就不展开，这个过程概括来说，就是找到一个可用的 RealConnection，再利用 RealConnection 的输入输出（BufferedSource 和 BufferedSink）创建 HttpCodec 对象，供后续步骤使用。

2.2.1.2，发送和接收数据：CallServerInterceptor

```
@Override public Response intercept(Chain chain) throws IOException {
  HttpCodec httpCodec = ((RealInterceptorChain) chain).httpStream();
```

```
  StreamAllocation streamAllocation = ((RealInterceptorChain)
chain).streamAllocation();
  Request request = chain.request();

  long sentRequestMillis = System.currentTimeMillis();
  httpCodec.writeRequestHeaders(request);

  if (HttpMethod.permitsRequestBody(request.method()) && request.body() != null)
{
    Sink requestBodyOut = httpCodec.createRequestBody(request,
request.body().contentLength());
    BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut);
    request.body().writeTo(bufferedRequestBody);
    bufferedRequestBody.close();
  }

  httpCodec.finishRequest();

  Response response = httpCodec.readResponseHeaders()
      .request(request)
      .handshake(streamAllocation.connection().handshake())
      .sentRequestAtMillis(sentRequestMillis)
      .receivedResponseAtMillis(System.currentTimeMillis())
      .build();

  if (!forWebSocket || response.code() != 101) {
    response = response.newBuilder()
        .body(httpCodec.openResponseBody(response))
        .build();
  }

  if ("close".equalsIgnoreCase(response.request().header("Connection"))
      || "close".equalsIgnoreCase(response.header("Connection"))) {
    streamAllocation.noNewStreams();
  }
```

```
  // 省略部分检查代码

  return response;
}
```

我们抓住主干部分：

1. 向服务器发送 request header；
2. 如果有 request body，就向服务器发送；
3. 读取 response header，先构造一个 Response 对象；
4. 如果有 response body，就在 3 的基础上加上 body 构造一个新的 Response 对象；

这里我们可以看到，核心工作都由 HttpCodec 对象完成，而 HttpCodec 实际上利用的是 Okio，而 Okio 实际上还是用的 Socket，所以没什么神秘的，只不过一层套一层，层数有点多。

其实 Interceptor 的设计也是一种分层的思想，每个 Interceptor 就是一层。

为什么要套这么多层呢？分层的思想在 TCP/IP 协议中就体现得淋漓尽致，分层简化了每一层的逻辑，每层只需要关注自己的责任（单一原则思想也在此体现），而各层之间通过约定的接口/协议进行合作（面向接口编程思想），共同完成复杂的任务。

简单应该是我们的终极追求之一，尽管有时为了达成目标不得不复杂，但如果有另一种更简单的方式，我想应该没有人不愿意替换。

2.2.2，发起异步网络请求

```
client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        System.out.println(response.body().string());
    }
});
// RealCall#enqueue
@Override public void enqueue(Callback responseCallback) {
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Executed");
    executed = true;
  }
  client.dispatcher().enqueue(new AsyncCall(responseCallback));
}
// Dispatcher#enqueuesynchronized void enqueue(AsyncCall call) {
  if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) <
maxRequestsPerHost) {
    runningAsyncCalls.add(call);
    executorService().execute(call);
  } else {
    readyAsyncCalls.add(call);
  }
}
```

这里我们就能看到 dispatcher 在异步执行时发挥的作用了，如果当前还能执行一个并发请求，那就立即执行，否则加入 readyAsyncCalls 队列，而正在执行的请求执行完毕之后，会调用 promoteCalls() 函数，来把 readyAsyncCalls 队列中的 AsyncCall “提升”为 runningAsyncCalls，并开始执行。

这里的 AsyncCall 是 RealCall 的一个内部类，它实现了 Runnable，所以可以被提交到 ExecutorService 上执行，而它在执行时会调
用 getResponseWithInterceptorChain() 函数，并把结果通
过 responseCallback 传递给上层使用者。

这样看来，同步请求和异步请求的原理是一样的，都是
在 getResponseWithInterceptorChain() 函数中通过 Interceptor 链条来实现的网络请求逻辑，而异步则是通过 ExecutorService 实现。

2.3，返回数据的获取

在上述同步（Call#execute() 执行之后）或者异步（Callback#onResponse() 回调中）请求完成之后，我们就可以从 Response 对象中获取到响应数据了，包括 HTTP status code，status message，response header，response body 等。

这里 body 部分最为特殊，因为服务器返回的数据可能非常大，所以必须通过数据流的方式来进行访问（当然也提供了诸如 string() 和 bytes() 这样的方法将流内的数据一次性读取完毕），而响应中其他部分则可以随意获取。

响应 body 被封装到 ResponseBody 类中，该类主要有两点需要注意：

1. 每个 body 只能被消费一次，多次消费会抛出异常；
2. body 必须被关闭，否则会发生资源泄漏；

在 2.2.1.2，发送和接收数据：CallServerInterceptor 小节中，我们就看过了 body 相关的代码：

```
if (!forWebSocket || response.code() != 101) {
  response = response.newBuilder()
      .body(httpCodec.openResponseBody(response))
      .build();
}
```

由 HttpCodec#openResponseBody 提供具体 HTTP 协议版本的响应 body，而 HttpCodec 则是利用 Okio 实现具体的数据 IO 操作。

这里有一点值得一提，OkHttp 对响应的校验非常严格，HTTP status line 不能有任何杂乱的数据，否则就会抛出异常，在我们公司项目的实践中，由于服务器的问题，偶尔 status line 会有额外数据，而服务端的问题也毫无头绪，导致我们不得不忍痛继续使用 HttpUrlConnection，而后者在一些系统上又存在各种其他的问题，例如魅族系统发送 multi-part form 的时候就会出现没有响应的问题。

2.4，HTTP 缓存

在 2.2.1，同步网络请求 小节中，我们已经看到了 Interceptor 的布局，在建立连接、和服务器通讯之前，就是 CacheInterceptor，在建立连接之前，我们检查响应是否已经被缓存、缓存是否可用，如果是则直接返回缓存的数据，否则就进行后面的流程，并在返回之前，把网络的数据写入缓存。

这块代码比较多，但也很直观，主要涉及 HTTP 协议缓存细节的实现，而具体的缓存逻辑 OkHttp 内置封装了一个 Cache 类，它利用 DiskLruCache，用磁盘上的有限大小空间进行缓存，按照 LRU 算法进行缓存淘汰，这里也不再展开。

我们可以在构造 OkHttpClient 时设置 Cache 对象，在其构造函数中我们可以指定目录和缓存大小：

```
public Cache(File directory, long maxSize);
```

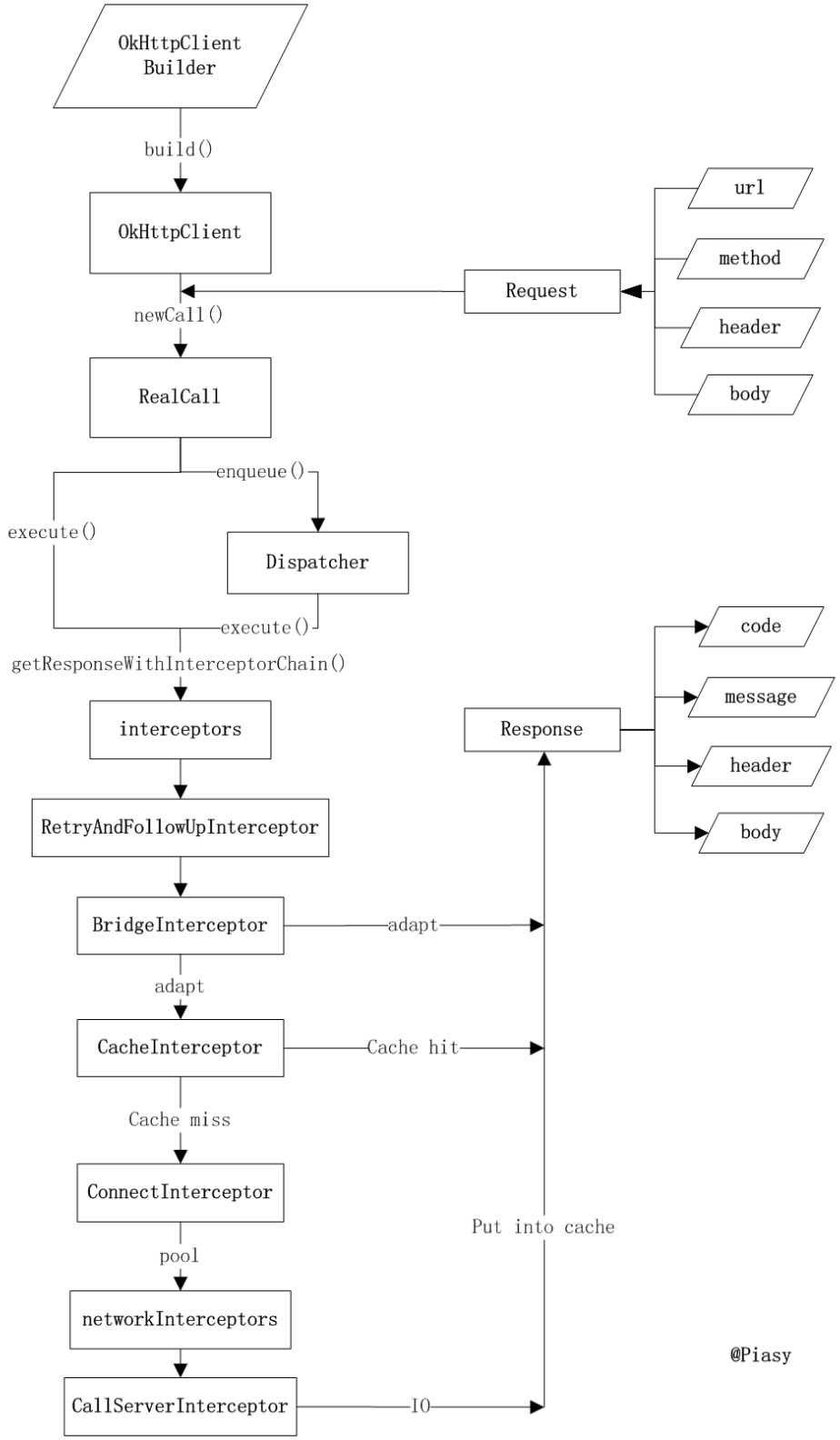
而如果我们对 OkHttp 内置的 Cache 类不满意，我们可以自行实

现 InternalCache 接口，在构造 OkHttpClient 时进行设置，这样就可以使用我们自定义的缓存策略了。

3，总结

OkHttp 还有很多细节部分没有在本文展开，例如 HTTP2/HTTPS 的支持等，但建立一个清晰的概览非常重要。对整体有了清晰认识之后，细节部分如有需要，再单独深入将更加容易。

在文章最后我们再来回顾一下完整的流程图：



- OkHttpClient 实现 Call.Factory，负责为 Request 创建 Call；

- RealCall 为具体的 Call 实现，其 enqueue() 异步接口通过 Dispatcher 利用 ExecutorService 实现，而最终进行网络请求时和同步 execute() 接口一致，都是通过 getResponseWithInterceptorChain() 函数实现；

- getResponseWithInterceptorChain() 中利用 Interceptor 链条，分层实现缓存、透明压缩、网络 IO 等功能；

第五章 Kotlin 相关

1.从原理分析 Kotlin 的延迟初始化：lateinit var 和 by lazy

Koltin 中属性在声明的同时也要求要被初始化，否则会报错。 例如以下代码:

```
private var name0: String //报错 private var name1: String = ″xiaoming″ //不报错 private var name2: String? = null //不报错复制代码
```

可是有的时候，我并不想声明一个类型可空的对象，而且我也没办法在对象一声明的时候就为它初始化，那么这时就需要用到 Kotlin 提供的**延迟初始化**。

Kotlin 中有两种延迟初始化的方式。一种是 **lateinit var**，一种是 **by lazy**。

lateinit var

```
private lateinit var name: String 复制代码
```

lateinit var 只能用来修饰类属性，不能用来修饰局部变量，并且只能用来修饰对象，不能用来修饰基本类型(因为基本类型的属性在类加载后的准备阶段都会被初始化为默认值)。

lateinit var 的作用也比较简单，就是让编译期在检查时不要因为属性变量未被初始化而报错。

Kotlin 相信当开发者显式使用 lateinit var 关键字的时候，他一定也会在后面某个合理的时机将该属性对象初始化的(然而，谁知道呢，也许他用完才想起还没初始化)。

by lazy

by lazy 本身是一种属性委托。属性委托的关键字是 by。by lazy 的写法如下：

```
//用于属性延迟初始化 val name: Int by lazy { 1 }
//用于局部变量延迟初始化 public fun foo() {
    val bar by lazy { "hello" }
    println(bar)
}复制代码
```

以下以 name 属性为代表来讲解 by kazy 的原理，局部变量的初始化也是一样的原理。

by lazy 要求属性声明为 val，即不可变变量，在 java 中相当于被 final 修饰。

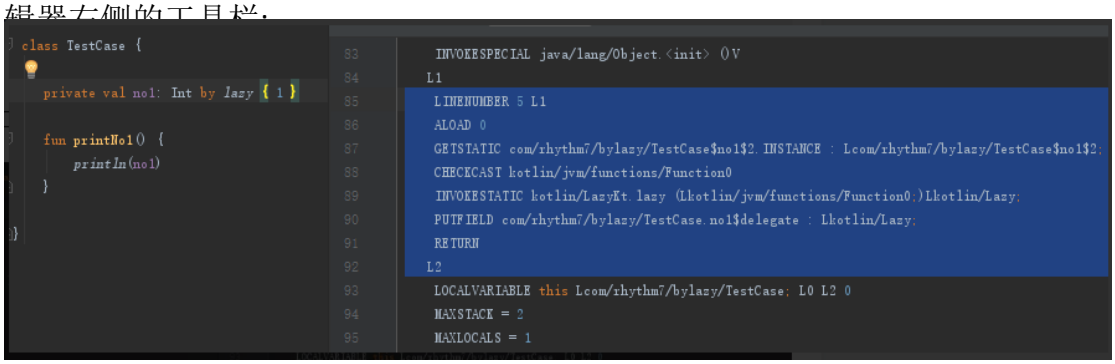
这意味着该变量一旦初始化后就不允许再被修改值了(基本类型是值不能被修改，对象类型是引用不能被修改)。{}内的操作就是返回唯一一次初始化的结果。

by lazy 可以使用于类属性或者局部变量。

写一段最简单的代码分析 by lazy 的实现:

```
class TestCase {

  private val name: Int by lazy { 1 }

  fun printname() {
     println(name)
  }

}复制代码
```

在 IDEA 中点击 toolbar 中的 **Tools -> Kotlin -> Show Kotlin ByteCode**，查看编辑器右侧的工具栏：



 不想看字节码分析的可以直接跳过，每段字节码后面都有 java/kotlin 版本的解释

更完整的字节码片段如下:

```
public <init>()V
 L0
  LINENUMBER 3 L0
  ALOAD 0
  INVOKESPECIAL java/lang/Object.<init> ()V
 L1
  LINENUMBER 5 L1
  ALOAD 0
  GETSTATIC com/rhythm7/bylazy/TestCase$name$2.INSTANCE :
Lcom/rhythm7/bylazy/TestCase$name$2;
  CHECKCAST kotlin/jvm/functions/Function0
  INVOKESTATIC kotlin/LazyKt.lazy
(Lkotlin/jvm/functions/Function0;)Lkotlin/Lazy;
  PUTname com/rhythm7/bylazy/TestCase.name$delegate : Lkotlin/Lazy;
  RETURN
 L2
  LOCALVARIABLE this Lcom/rhythm7/bylazy/TestCase; L0 L2 0
  MAXSTACK = 2
  MAXLOCALS = 1 复制代码
```

该段代码是在字节码生成的 `public <clinit>()V` 方法内的。之所以是在该方法内，是因为非单例 object 的 Kotlin 类的属性初始化代码语句经过编译器处理后都会被收集到该方法内，如果是 object 对象，对应的属性初始化代码语句则会被收集到 `static <clinit>()V` 方法中。另外，在字节码中，这两个方法是拥有不同方法签名的，这与语言级别上判断两个方法是否相同的方式有所不同。前者是实例构造方法，后者是类构造方法。

L0 与 L1 之间的字节码代表调用了 Object()的构造方法，这是默认的父类构造方法。L2 之后的是本地变量表说明。L1 与 L2 之间的字节码对应如下 kotlin 代码：

```
private val name: Int by lazy { 1 }
```

L1 与 L2 之间这段字节码的意思是:
源代码行号 5 对应字节码方法体内的行号 1； 将 this(非静态方法默认的第一个本地变量)推送至栈顶;
获取静态变量 com.rhythm7.bylazy.TestCase$name$2.INSTANCE;
检验 INSTANCE 能否转换为 kotlin.jvm.functions.Function0 类;
调用静态方法 kotlin.LazyKt.lazy(kotlin.jvm.functions.Function0)，将 INSTANCE 作为参数传入，并获得一个 kotlin.Lazy 类型的返回值;
将以上返回值赋值给 com.rhythm7.bylazy.TestCase.name$delegate;
最后结束方法。

相当于 java 代码:

```
TestCase() {
    name$delegate = LazyKt.lazy((Function0)name$2.INSTANCE)
}
```

其中 name$delegate 是编译后生成的属性，对象类型为 Lazy。

```
private final Lkotlin/Lazy; name$delegate  复制代码
```

name$2 都是编译后生成的内部类。

```
final class com/rhythm7/bylazy/TestCase$name$2 extends
kotlin/jvm/internal/Lambda  implements kotlin/jvm/functions/Function0
```

name$2 继承了 kotlin.jvm.internal.Lambda 类并实现了 kotlin.jvm.functions.Function0 接口， 可以看出 name$2 其实就是 kotlin 函数参数类型 ()->T 的具体实现，通过字节码分析不难知道 name$2.INSTANCE 则是该实现类的一个静态对象实例。
所以以上字节码又相当于 Koltin 中的:

```
init {
    name$delegate = lazy(()->{})
}
```

然而，这些代码的作用仅仅是给一个编译期生成的属性变量赋值而已，并没有其他的操作。

真正实现属性变量延迟初始化的地方其实是在属性 name 的 getter 方法里。

如果在 java 代码中调用过 kotlin 代码，会发现 java 代码中只能通过 setter 或 getter 的方式访问 koltin 编写的对象属性，这是因为 kotlin 中默认会对属性添加 private 修饰符,并根据该属性变量是 val 还是 var 生成 getter 或 getter 和 setter 一起生成。然后又根据对该属性的访问权限给 getter 和 setter 添加对应的访问权限修饰符(默认是 public)。

查看 getName()的具体实现:

```
private final getName()I
 L0
  ALOAD 0
  GETFIELD com/rhythm7/bylazy/TestCase.name$delegate : Lkotlin/Lazy;
  ASTORE 1
  ALOAD 0
  ASTORE 2
  GETSTATIC com/rhythm7/bylazy/TestCase.$$delegatedProperties :
[Lkotlin/reflect/KProperty;
  ICONST_0
  AALOAD
  ASTORE 3
 L1
  ALOAD 1
  INVOKEINTERFACE kotlin/Lazy.getValue ()Ljava/lang/Object;
 L2
  CHECKCAST java/lang/Number
  INVOKEVIRTUAL java/lang/Number.intValue ()I
  IRETURN
 L3
  LOCALVARIABLE this Lcom/rhythm7/bylazy/TestCase; L0 L3 0
  MAXSTACK = 2
  MAXLOCALS = 4
```

相当于 java 代码:

```
private final int getName(){
    Lazy var1 = this.name$delegate;
    KProperty var2 = this.$$delegatedProperties[0]
    return ((Number)var1.getValue()).intValue()
}复制代码
```

可以看到name的getter方法其实是返回了 `name$delegate.getValue()`方法。`$$delegatedProperties` 是编译后自动生成的属性，但在此处并没有用到，所以不用关心。

那么现在我们要关心的就只有 `name$delegate.getValue()`，也就是 Lazy 类 `getValue()`方法的具体实现了。

先看 LazyKt.lazy(()->T)的实现:

```
public fun <T> lazy(initializer: () -> T): Lazy<T> =
SynchronizedLazyImpl(initializer)复制代码
```

再看 SynchronizedLazyImpl 类的具体实现:

```
private object UNINITIALIZED_VALUE
private class SynchronizedLazyImpl<out T>(initializer: () -> T, lock: Any?
= null) : Lazy<T>, Serializable {
    private var initializer: (() -> T)? = initializer
    @Volatile private var _value: Any? = UNINITIALIZED_VALUE
    // final field is required to enable safe publication of constructed
instance
    private val lock = lock ?: this

    override val value: T
        get() {
            val _v1 = _value
            if (_v1 !== UNINITIALIZED_VALUE) {
                @Suppress("UNCHECKED_CAST")
                return _v1 as T
            }

            return synchronized(lock) {
                val _v2 = _value
                if (_v2 !== UNINITIALIZED_VALUE) {
                    @Suppress("UNCHECKED_CAST") (_v2 as T)
                }
                else {
                    val typedValue = initializer!!()
                    _value = typedValue
                    initializer = null
                    typedValue
                }
            }
        }

......
}
```

以上代码的阅读难度就非常低了。

SynchronizedLazyImpl 继承了 Lazy 类，并指定了泛型类型，然后重写了 Lazy 父类的 getValue()方法。 getValue()方法中会对_value 是否已初始化做判断，并返回_value，从而实现 value 的延迟初始化的作用。

注意，对 value 的初始化行为本身是线程安全的。

总结

总结一下，当一个属性 name 需要 by lazy 时，具体是怎么实现的:

1. 生成一个该属性的附加属性:name$$delegate；
2. 在构造器中，将使用 lazy(()->T)创建的 Lazy 实例对象赋值给 name$$delegate；
3. 当该属性被调用，即其 getter 方法被调用时返回 name$$delegate.getVaule()，而 name$$delegate.getVaule()方法的返回结果是对象 name$$delegate 内部的_value 属性值，在 getVaule()第一次被调用时会将_value 进行初始化，往后都是直接将_value 的值返回，从而实现属性值的唯一一次初始化。

那么，再总结一下，lateinit var 和 by lazy 哪个更好用?

首先两者的应用场景是略有不同的。

然后，虽然两者都可以推迟属性初始化的时间，但是 lateinit var 只是让编译期忽略对属性未初始化的检查，后续在哪里以及何时初始化还需要开发者自己决定。

而 by lazy 真正做到了声明的同时也指定了延迟初始化时的行为，在属性被第一次被使用的时候能自动初始化。但这些功能是要为此付出一丢丢代价的。

作者：咸鱼不思议
链接：https://juejin.im/post/5affc369f265da0b9b079629

2.From Java To Kotlin

打印日志

- Java

```
System.out.print("Amit Shekhar");System.out.println("Amit Shekhar");
```

- Kotlin

```
print("Amit Shekhar")println("Amit Shekhar")
```

常量与变量

- Java

```
String name = "Amit Shekhar";final String name = "Amit Shekhar";
```

- Kotlin

```
var name = "Amit Shekhar"val name = "Amit Shekhar"
```

---

null 声明

- Java

```
String otherName;
otherName = null;
```

- Kotlin

```
var otherName : String?
otherName = null
```

空判断

- Java

```
if (text != null) {
    int length = text.length();
}
```

- Kotlin

```
text?.let {
    val length = text.length
}// or simplyval length = text?.length
```

字符串拼接

- Java

```
String firstName = "Amit";String lastName = "Shekhar";String message = "My name is: "
+ firstName + " " + lastName;
```

- Kotlin

```
val firstName = "Amit"val lastName = "Shekhar"val message = "My name is: $firstName $lastName"
```

换行

- Java

```
String text = "First Line\n" +
              "Second Line\n" +
              "Third Line";
```

- Kotlin

```
val text = """        |First Line        |Second Line        |Third Line        """.trimMargin()
```

三元表达式

- Java

```
String text = x > 5 ? "x > 5" : "x <= 5";
```

- Kotlin

```
val text = if (x > 5)
              "x > 5"
           else "x <= 5"
```

操作符

- java

```
final int andResult  = a & b;final int orResult   = a | b;final int xorResult  = a ^ b;final int rightShift = a >> 2;final int leftShift  = a << 2;final int unsignedRightShift = a >>> 2;
```

- Kotlin

```
val andResult  = a and bval orResult   = a or bval xorResult  = a xor bval rightShift
= a shr 2val leftShift  = a shl 2val unsignedRightShift = a ushr 2
```

类型判断和转换（声明式)

- Java

```
if (object instanceof Car) {
}Car car = (Car) object;
```

- Kotlin

```
if (object is Car) {
}var car = object as Car
```

类型判断和转换（隐式)

- Java

```
if (object instanceof Car) {
   Car car = (Car) object;
}
```

- Kotlin

```
if (object is Car) {
   var car = object // 聪明的转换
}
```

多重条件

- Java

```
if (score >= 0 && score <= 300) { }
```

- Kotlin

```
if (score in 0..300) { }
```

更灵活的 case 语句

- Java

```
int score = // some score;String grade;switch (score) {
    case 10:
    case 9:
        grade = "Excellent";
        break;
    case 8:
    case 7:
    case 6:
        grade = "Good";
        break;
    case 5:
    case 4:
        grade = "OK";
        break;
    case 3:
    case 2:
    case 1:
        grade = "Fail";
        break;
    default:
        grade = "Fail";
}
```

- Kotlin

```
var score = // some scorevar grade = when (score) {
    9, 10 -> "Excellent"
    in 6..8 -> "Good"
    4, 5 -> "OK"
    in 1..3 -> "Fail"
    else -> "Fail"
}
```

for 循环

- Java

```
for (int i = 1; i <= 10 ; i++) { }
for (int i = 1; i < 10 ; i++) { }
for (int i = 10; i >= 0 ; i--) { }
for (int i = 1; i <= 10 ; i+=2) { }
for (int i = 10; i >= 0 ; i-=2) { }
for (String item : collection) { }
for (Map.Entry<String, String> entry: map.entrySet()) { }
```

- Kotlin

```
for (i in 1..10) { }
for (i in 1 until 10) { }
for (i in 10 downTo 0) { }
for (i in 1..10 step 2) { }
for (i in 10 downTo 0 step 2) { }
for (item in collection) { }
for ((key, value) in map) { }
```

更方便的集合操作

- Java

```
final List<Integer> listOfNumber = Arrays.asList(1, 2, 3, 4);
final Map<Integer, String> keyValue = new HashMap<Integer, String>();
map.put(1, "Amit");
map.put(2, "Ali");
map.put(3, "Mindorks");
// Java 9final List<Integer> listOfNumber = List.of(1, 2, 3, 4);
final Map<Integer, String> keyValue = Map.of(1, "Amit",
                                             2, "Ali",
                                             3, "Mindorks");
```

- Kotlin

```
val listOfNumber = listOf(1, 2, 3, 4)val keyValue = mapOf(1 to "Amit",
                    2 to "Ali",
                    3 to "Mindorks")
```

遍历

- Java

```
// Java 7 and belowfor (Car car : cars) {
  System.out.println(car.speed);
}
// Java 8+
cars.forEach(car -> System.out.println(car.speed));
// Java 7 and belowfor (Car car : cars) {
  if (car.speed > 100) {
    System.out.println(car.speed);
  }
}
// Java 8+
cars.stream().filter(car -> car.speed > 100).forEach(car ->
System.out.println(car.speed));
```

- Kotlin

```
cars.forEach {
    println(it.speed)
}

cars.filter { it.speed > 100 }
      .forEach { println(it.speed)}
```

方法定义

- Java

```
void doSomething() {
   // logic here
}
void doSomething(int... numbers) {
   // logic here
}
```

- Kotlin

```
fun doSomething() {
   // logic here
```

```
}
fun doSomething(vararg numbers: Int) {
    // logic here
}
```

带返回值的方法

- Java

```
int getScore() {
    // logic here
    return score;
}
```

- Kotlin

```
fun getScore(): Int {
    // logic here
    return score
}
// as a single-expression function
fun getScore(): Int = score
```

无结束符号

- Java

```
int getScore(int value) {
    // logic here
    return 2 * value;
}
```

- Kotlin

```
fun getScore(value: Int): Int {
    // logic here
    return 2 * value
}
// as a single-expression function
fun getScore(value: Int): Int = 2 * value
```

constructor 构造器

- Java

```
public class Utils {

    private Utils() {
      // This utility class is not publicly instantiable
    }

    public static int getScore(int value) {
        return 2 * value;
    }

}
```

- Kotlin

```
class Utils private constructor() {

    companion object {

        fun getScore(value: Int): Int {
            return 2 * value
        }

    }
}
// another way
object Utils {

    fun getScore(value: Int): Int {
        return 2 * value
    }

}
```

Get Set 构造器

- Java

```java
public class Developer {

    private String name;
    private int age;

    public Developer(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Developer developer = (Developer) o;

        if (age != developer.age) return false;
        return name != null ? name.equals(developer.name) : developer.name == null;

    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
```

```
    }

    @Override
    public String toString() {
        return "Developer{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- Kotlin

```
data class Developer(val name: String, val age: Int)
```

原型扩展

- Java

```
public class Utils {

    private Utils() {
      // This utility class is not publicly instantiable
    }

    public static int triple(int value) {
        return 3 * value;
    }

}
int result = Utils.triple(3);
```

- Kotlin

```
fun Int.triple(): Int {
  return this * 3
}
var result = 3.triple()
```

- Java

```
public enum Direction {
        NORTH(1),
```

```
        SOUTH(2),
        WEST(3),
        EAST(4);

        int direction;

        Direction(int direction) {
            this.direction = direction;
        }

        public int getDirection() {
            return direction;
        }
    }
```

- Kotlin

```
enum class Direction constructor(direction: Int) {
    NORTH(1),
    SOUTH(2),
    WEST(3),
    EAST(4);

    var direction: Int = 0
        private set

    init {
        this.direction = direction
    }
}
```

3. 怎么用 Kotlin 去提高生产力：Kotlin Tips

汇总 Kotlin 相对于 Java 的优势，以及怎么用 Kotlin 去简洁、务实、高效、安全的开发，每个 tip 都有详细的说明和案例代码，争取把每个 tip 分析得清楚易懂，会不断的更新维护 tips，欢迎 fork 进来加入我们一起来维护，有问题的话欢迎提 Issues。

- 推荐：Android 模块化通信项目 module-service-manager，支持模块间功能服务/View/Fragment 的通信调用等，通过注解标示模块内需要暴露出来的服务和 View，然后 gradle 插件会通过 transform 来 hook 编译过程，扫描出注解信息后再利用 asm 生成代码来向服务管理中心注册对应的服务和 View，之后模块间就可以利用框架这个桥梁来调用和通信了
- 推荐：Kotlin 的实践项目 debug_view_kotlin，用 Kotlin 实现的 Android 浮层调试控制台，实时的显示内存、FPS、App 启动时间、Activity 启动时间、文字 Log

- 推荐：数据预加载项目 and-load-aot，通过提前加载数据来提高页面启动速度，利用编译时注解生成加载方法的路由，在 Activity 启动前就去加载数据

目录

- Tip1-更简洁的字符串
  - 1、三个引号 2、字符串模版
- Tip2-Kotlin 中大多数控制结构都是表达式
  - 1、语句和表达式 2、if 3、when
- Tip3-更好调用的函数：显式参数名及默认参数值
  - 1、显式参数名 2、默认参数值 3、@JvmOverloads
- Tip4-扩展函数和属性
  - 1、扩展函数 2、扩展属性
- Tip5-懒初始化 bylazy 和延迟初始化 lateinit
  - 1、by lazy 2、lateinit
- Tip6-不用再手写 findViewById
  - 1、Activity 2、子 View 或者 include 标签 3、Fragment
- Tip7-利用局部函数抽取重复代码
  - 1、局部函数 2、扩展函数
- Tip8-使用数据类来快速实现 model 类
- Tip9-用类委托来快速实现装饰器模式
- Tip10-Lambda 表达式简化 OnClickListener
- Tip11-with 函数来简化代码
- Tip12-apply 函数来简化代码
- Tip13-在编译阶段避免掉 NullPointerException
  - 1、可空和不可空类型 2、let 3、Elvis 操作符
- Tip14-运算符重载
- Tip15-高阶函数简化代码
- Tip16-用 Lambda 来简化策略模式

Tip1-更简洁的字符串

回到目录

三个引号

详见案例代码 KotlinTip1

Kotlin 中的字符串基本 Java 中的类似，有一点区别是加入了三个引号"""来方便长篇字符的编写。 而在 Java 中，这些都需要转义，先看看 java 中的式例

```
    public void testString1() {
        String str1 = "abc";
        String str2 = "line1\n" +
                "line2\n" +
                "line3";
        String js = "function myFunction()\n" +
                "{\n" +
                "    document.getElementById(\"demo\").innerHTML=\"My First
JavaScript Function\";\n" +
                "}";
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(js);
    }
```

kotlin 除了有单个双引号的字符串，还对字符串的加强，引入了**三个引号**，"""中可以包含换行、反斜杠等等特殊字符：

```
/** kotlin 对字符串的加强，三个引号"""中可以包含换行、反斜杠等等特殊字符* */fun testString()
{
    val str1 = "abc"
    val str2 = """line1\n        line2        line3        """
    val js = """        function myFunction()
{            document.getElementById("demo").innerHTML="My First JavaScript
Function";        }        """.trimIndent()
    println(str1)
    println(str2)
    println(js)
}
```

字符串模版

同时，Kotlin 中引入了**字符串模版**，方便字符串的拼接，可以用$符号拼接变量和表达式

```
/** kotlin 字符串模版，可以用$符号拼接变量和表达式* */fun testString2() {
    val strings = arrayListOf("abc", "efd", "gfg")
    println("First content is $strings")
    println("First content is ${strings[0]}")
    println("First content is ${if (strings.size > 0) strings[0] else "null"}")
```

```
}
```

值得注意的是，在 Kotlin 中，美元符号$是特殊字符，在字符串中不能直接显示，必须经过转义，方法 1 是用反斜杠，方法二是${'$'}

```
/** Kotlin 中，美元符号$是特殊字符，在字符串中不能直接显示，必须经过转义，方法 1 是用反斜杠，方法二是${'$'}* */fun testString3() {
    println("First content is \$strings")
    println("First content is ${'$'}strings")
}
```

Tip2-Kotlin 中大多数控制结构都是表达式

回到目录

首先，需要弄清楚一个概念**语句和表达式**，然后会介绍控制结构表达式的优点：**简洁**

语句和表达式是什么？

- 表达式有值，并且能作为另一个表达式的一部分使用
- 语句总是包围着它的代码块中的顶层元素，并且没有自己的值

Kotlin 与 Java 的区别

- Java 中，所有的控制结构都是语句，也就是控制结构都没有值
- Kotlin 中，除了循环（for、do 和 do/while）以外，大多数控制结构都是表达式(if/when 等)

详见案例代码 tip2

Example1：if 语句

java 中，if 是语句，没有值，必须显式的 return

```
/** java 中的 if 语句* */public int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

kotlin 中，if 是表达式，不是语句，因为表达式有值，可以作为值 return 出去

```
/** kotlin 中，if 是表达式，不是语句，类似于 java 中的三目运算符 a > b ? a : b* */fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}
```

上面的 if 中的分支最后一行语句就是该分支的值，会作为函数的返回值。这其实跟 java 中的三元运算符类似，

```
/** java 的三元运算符* */public int max2(int a, int b) {
    return a > b ? a : b;
}
```

上面是 java 中的三元运算符，kotlin 中 if 是表达式有值，完全可以替代，**故 kotlin 中已没有三元运算符了**，用 if 来替代。 上面的 max 函数还可以简化成下面的形式

```
/** kotlin 简化版本* */fun max2(a: Int, b: Int) = if (a > b) a else b
```

Example2：when 语句

Kotlin 中的 when 非常强大，完全可以取代 Java 中的 switch 和 if/else，同时，**when 也是表达式**，when 的每个分支的最后一行为当前分支的值 先看一下 java 中的 switch

```
    /*    * java 中的 switch    * */
    public String getPoint(char grade) {
        switch (grade) {
            case 'A':
                return "GOOD";
            case 'B':
            case 'C':
                return "OK";
            case 'D':
                return "BAD";
            default:
                return "UN_KNOW";
        }
    }
```

java 中的 switch 有太多限制，我们再看看 Kotlin 怎样去简化的

```
/** kotlin 中，when 是表达式，可以取代 Java 中的 switch，when 的每个分支的最后一行为当前分支的值* */fun getPoint(grade: Char) = when (grade) {
    'A' -> "GOOD"
    'B', 'C' -> {
        println("test when")
        "OK"
    }
    'D' -> "BAD"
    else -> "UN_KNOW"
```

```
}
```

同样的，when 语句还可以取代 java 中的 if/else if，其是表达式有值，并且更佳简洁

```
    /*    * java 中的 if else    * */
    public String getPoint2(Integer point) {
        if (point > 100) {
            return "GOOD";
        } else if (point > 60) {
            return "OK";
        } else if (point.hashCode() == 0x100) {
            //...
            return "STH";
        } else {
            return "UN_KNOW";
        }
    }
```

再看看 kotlin 的版本，使用**不带参数的 when**，只需要 6 行代码

```
/** kotlin 中，when 是表达式，可以取代 java 的 if/else，when 的每个分支的最后一行为当前分支的值* */fun getPoint2(grade: Int) = when {
    grade > 90 -> "GOOD"
    grade > 60 -> "OK"
    grade.hashCode() == 0x100 -> "STH"
    else -> "UN_KNOW"
}
```

Tip3-更好调用的函数-显式参数名及默认参数值

回到目录

显式参数名

Kotlin 的函数更加好调用，主要是表现在两个方面：1，显式的**标示参数名**，可以方便代码阅读；2，函数可以有**默认参数值**，可以大大**减少 Java 中的函数重载**。 例如现在需要实现一个工具函数，打印列表的内容： 详见案例代码 KotlinTip3

```
/** 打印列表的内容* */fun <T> joinToString(collection: Collection<T>,
                      separator: String,
                      prefix: String,
                      postfix: String): String {
    val result = StringBuilder(prefix)
    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
```

```
    }
    result.append(postfix)
    return result.toString()
}/** 测试* */fun printList() {
    val list = listOf(2, 4, 0)
    // 不标明参数名
    println(joinToString(list, " - ", "[", "]"))
    // 显式的标明参数名称
    println(joinToString(list, separator = " - ", prefix = "[", postfix = "]"))
}
```

如上面的代码所示，函数 joinToString 想要打印列表的内容，需要传入四个参数：列表、分隔符、前缀和后缀。 由于参数很多，在后续使用该函数的时候不是很直观的知道每个参数是干什么用的，这时候可以显式的标明参数名称，增加代码可读性。

默认参数值

同时，定义函数的时候还可以给函数默认的参数，如下所示：

```
/** 打印列表的内容，带有默认的参数，可以避免 java 的函数重载* */fun <T>
joinToString2(collection: Collection<T>,
                      separator: String = ", ",
                      prefix: String = "",
                      postfix: String = ""): String {
    val result = StringBuilder(prefix)
    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}/** 测试* */fun printList3() {
    val list = listOf(2, 4, 0)
    println(joinToString2(list, " - "))
    println(joinToString2(list, " , ", "["))
}
```

这样有了默认参数后，在使用函数时，如果不传入该参数，默认会使用默认的值，这样可以避免 Java 中大量的函数重载。

@JvmOverloads

在 java 与 kotlin 的混合项目中，会发现用 kotlin 实现的带默认参数的函数，在 java 中去调用的化就不能利用这个特性了，还是需要给所有参数赋值，像下面 java 这样：

```
List<Integer> arr = new ArrayList<Integer>() {{add(2);add(4);add(0);}};String res = joinToString2(arr, "-", "", "");System.out.println(res);
```

这时候可以在 kotlin 的函数前添加注解@JvmOverloads，添加注解后翻译为 class 的时候 kotlin 会帮你去生成多个函数实现函数重载，kotlin 代码如下：

```
/** 通过注解@JvmOverloads 解决 java 调用 kotlin 时不支持默认参数的问题* */@JvmOverloadsfun
<T> joinToString2New(collection: Collection<T>,
                        separator: String = ", ",
                        prefix: String = "",
                        postfix: String = ""): String {
    val result = StringBuilder(prefix)
    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
```

这样以后，java 调用 kotlin 的带默认参数的函数就跟 kotlin 一样方便了：

```
List<Integer> arr = new ArrayList<Integer>() {{add(2);add(4);add(0);}};String res = joinToString2New(arr, "-");System.out.println(res);String res2 = joinToString2New(arr, "-", "=>");System.out.println(res2);
```

Tip4-扩展函数和属性

回到目录

扩展函数和扩展属性是 Kotlin 非常方便实用的一个功能，它可以让我们随意的扩展第三方的库，你如果觉得别人给的 SDK 的 Api 不好用，或者不能满足你的需求，这时候你可以用扩展函数完全去自定义。

扩展函数

例如 String 类中，我们想获取最后一个字符，String 中没有这样的直接函数，你可以用.后声明这样一个扩展函数： 详见案例代码 KotlinTip4

```
/** 扩展函数* */fun String.lastChar(): Char = this.get(this.length - 1)/** 测试* */fun testFunExtension() {
    val str = "test extension fun";
    println(str.lastChar())
}
```

这样定义好 lastChar()函数后，之后只需要 import 进来后，就可以用 String 类直接调用该函数了，跟调用它自己的方法没有区别。这样可以避免重复代码和一些静态工具类，而且代码更加简洁明了。 例如我们可以改造上面 tip3 中的打印列表内容的函数：

```
/** 用扩展函数改造 Tip3 中的列表打印内容函数* */fun <T>
Collection<T>.joinToString3(separator: String = ", ",
                                    prefix: String = "",
                                    postfix: String = ""): String {
    val result = StringBuilder(prefix)
    for ((index, element) in withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
fun printList4() {
    val list = listOf(2, 4, 0)
    println(list.joinToString3("/"))
}
```

扩展属性

除了扩展函数，还可以扩展属性，例如我想实现 String 和 StringBuilder 通过属性去直接获得最后字符：

```
/** 扩展属性 lastChar 获取 String 的最后一个字符* */val String.lastChar: Char
    get() = get(length - 1)/** 扩展属性 lastChar 获取 StringBuilder 的最后一个字符* */var
StringBuilder.lastChar: Char
    get() = get(length - 1)
    set(value: Char) {
        setCharAt(length - 1, value)
    }/** 测试* */fun testExtension() {
    val s = "abc"
    println(s.lastChar)
    val sb = StringBuilder("abc")
    println(sb.lastChar)
}
```

定义好扩展属性后，之后只需 import 完了就跟使用自己的属性一样方便了。

Why？Kotlin 为什么能实现扩展函数和属性这样的特性？

在 Kotlin 中要理解一些语法，只要认识到 **Kotlin 语言最后需要编译为 class 字节码，Java 也是编译为 class 执行，也就是可以大致理解为 Kotlin 需要转成 Java 一样的语法结构**，Kotlin 就是一种**强大的语法糖**而已，Java 不具备的功能 Kotlin 也不能越界的。

- 那 Kotlin 的扩展函数怎么实现的呢？介绍一种万能的办法去理解 Kotlin 的语法：**将 Kotlin 代码转化成 Java 语言**去理解，步骤如下：
    - 在 Android Studio 中选择 Tools ---> Kotlin ---> Show Kotlin Bytecode 这样就把 Kotlin 转化为 class 字节码了
    - class 码阅读不太友好，点击左上角的 Decompile 就转化为 Java
- 再介绍一个小窍门，在前期对 Kotlin 语法不熟悉的时候，可以先用 Java 写好代码，再利用 AndroidStudio 工具**将 Java 代码转化为 Kotlin 代码**，步骤如下：
    - 在 Android Studio 中选中要转换的 Java 代码 ---> 选择 Code ---> Convert Java File to Kotlin File

我们看看将上面的扩展函数转成 Java 后的代码

```
/** 扩展函数会转化为一个静态的函数，同时这个静态函数的第一个参数就是该类的实例对象* */public static final char lastChar(@NotNull String $receiver) {
    Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
    return $receiver.charAt($receiver.length() - 1);
}/** 获取的扩展属性会转化为一个静态的 get 函数，同时这个静态函数的第一个参数就是该类的实例对象* */public static final char getLastChar(@NotNull StringBuilder $receiver) {
    Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
    return $receiver.charAt($receiver.length() - 1);
}/** 设置的扩展属性会转化为一个静态的 set 函数，同时这个静态函数的第一个参数就是该类的实例对象* */public static final void setLastChar(@NotNull StringBuilder $receiver, char value)
{
    Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
    $receiver.setCharAt($receiver.length() - 1, value);
}
```

查看上面的代码可知：对于扩展函数，转化为 Java 的时候其实就是一个静态的函数，同时这个静态函数的第一个参数就是该类的实例对象，这样把类的实例传入函数以后，函数内部就可以访问到类的公有方法。 对于扩展属性也类似，获取的扩展属性会转化为一个静态的 get 函数，同时这个静态函数的第一个参数就是该类的实例对象，设置的扩展属性会转化为一个静态的 set 函数，同时这个静态函数的第一个参数就是该类的实例对象。 函数内部可以访问公有的方法和属性。顶层的扩展函数是 static 的，**不能被 override**

-

从上面转换的源码其实可以看到**扩展函数和扩展属性适用的地方和缺陷**：

-
  - 扩展函数和扩展属性内**只能访问到类的公有方法和属性**，私有的和 protected 是访问不了的
    - 扩展函数**不是真的修改了原来的类**，定义一个扩展函数不是将新成员函数插入到类中，扩展函数的类型是"静态的"，不是在运行时决定类型，案例代码 StaticllyExtension.kt

```
open class C
class D : C()
fun C.foo() = "c"
fun D.foo() = "d"
/** https://kotlinlang.org/docs/reference/extensions.html* Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class,* but merely make new functions callable with the dot-notation on variables of this type. Extension functions are* dispatched statically.* */fun printFoo(c: C) {
  println(c.foo())
}
fun testStatically() {
  printFoo(C()) // print c
  printFoo(D()) // also print c
}
```

    - 上面的案例中即使调用 printFoo(D())还是打印出 c，而不是 d。转成 java 中会看到下面的代码，D 类型在调用的时候会强制转换为 C 类型：

```
public static final String foo(@NotNull C $receiver) {
  Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
  return "c";
}
public static final String foo(@NotNull D $receiver) {
  Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
  return "d";
}
public static final void printFoo(@NotNull C c) {
  Intrinsics.checkParameterIsNotNull(c, "c");
  String var1 = foo(c);
  System.out.println(var1);
}public static final void testStatically() {
  printFoo(new C());
  printFoo((C)(new D()));
```

```
}
```

-

声明扩展函数作为类的成员变量

-
  - 上面的例子扩展函数是作为顶层函数，如果把扩展函数申明为类的成员变量，即扩展函数的作用域就在类的内部有效，案例代码 ExtensionsAsMembers.kt

```
open class D {
}
class D1 : D() {
}
open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()   // call the extension function
    }
}
class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}
fun testAsMembers() {
    C().caller(D())   // prints "D.foo in C"
    C1().caller(D())  // prints "D.foo in C1" - dispatch receiver is resolved virtually
    C().caller(D1())  // prints "D.foo in C" - extension receiver is resolved statically
    C1().caller(D1()) // prints "D.foo in C1"
}
```

函数 caller 的类型是 D，即使调用 C().caller(D1())，打印的结果还是 D.foo in C，而不是 D1.foo in C，不是运行时来动态决定类型，成员扩展函数申明为 open， 一旦在子类中被 override，就调用不到在父类中的扩展函数，在子类中的作用域内的只能访问到 override 后的函数，不能像普通函数 override 那样通过 super 关键字访问了。

- 下面再举几个扩展函数的例子，让大家感受一下扩展函数的方便：

```
/** show toast in activity* */fun Activity.toast(msg: String) {
    Toast.makeText(this, msg, Toast.LENGTH_SHORT).show()
}
val Context.inputMethodManager: InputMethodManager?
    get() = getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
/** hide soft input* */fun Context.hideSoftInput(view: View) {
    inputMethodManager?.hideSoftInputFromWindow(view.windowToken, 0)
}

/** * screen width in pixels */val Context.screenWidth
    get() = resources.displayMetrics.widthPixels
/** * screen height in pixels */val Context.screenHeight
    get() = resources.displayMetrics.heightPixels
/** * returns dip(dp) dimension value in pixels * @param value dp */fun
Context.dip2px(value: Int): Int = (value * resources.displayMetrics.density).toInt()
```

Tip5-懒初始化 bylazy 和延迟初始化 lateinit

回到目录

懒初始化 by lazy

懒初始化是指推迟一个变量的初始化时机，变量在使用的时候才去实例化，这样会更加的高效。因为我们通常会遇到这样的情况，一个变量直到使用时才需要被初始化，或者仅仅是它的初始化依赖于某些无法立即获得的上下文。 详见案例代码 KotlinTip5

```
/** 懒初始化 api 实例* */val purchasingApi: PurchasingApi by lazy {
    val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl(API_URL)
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
    retrofit.create(PurchasingApi::class.java)
}
```

像上面的代码，retrofit 生成的 api 实例会在首次使用到的时候才去实例化。需要注意的是 by lazy 一般只能修饰 val 不变的对象，不能修饰 var 可变对象。

```
class User(var name: String, var age: Int)
```

```
/** 懒初始化 by lazy* */val user1: User by lazy {
    User("jack", 15)
}
```

延迟初始化 lateinit

另外，对于 var 的变量，如果类型是非空的，是必须初始化的，不然编译不通过，这时候需要用到 lateinit 延迟初始化，使用的时候再去实例化。

```
/** 延迟初始化 lateinit* */lateinit var user2: User
fun testLateInit() {
    user2 = User("Lily", 14)
}
```

by lazy 和 lateinit 的区别

- by lazy 修饰 val 的变量
- lateinit 修饰 var 的变量，且变量是非空的类型

Tip6-不用再手写 findViewById

回到目录

在 Activity 中使用

在 Android 的 View 中，会有很多代码是在声明一个 View，然后通过 findViewById 后从 xml 中实例化赋值给对应的 View。在 kotlin 中可以完全解放出来了，利用 kotlin-android-extensions 插件，不用再手写 findViewById。步骤如下： 详见案例代码 KotlinTip6

- 步骤 1，在项目的 gradle 中 apply plugin: 'kotlin-android-extensions'
- 步骤 2，按照原来的习惯书写布局 xml 文件

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/tip6Tv"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content" />

    <ImageView
        android:id="@+id/tip6Img"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/tip6Btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

- 步骤 3，在 java 代码中 import 对应的布局就可以开始使用了，View 不用提前声明，插件会自动根据布局的 id 生成对应的 View 成员（其实没有生成属性，原理见下面）

```
import com.sw.kotlin.tips.R/** 导入插件生成的 View* */import
kotlinx.android.synthetic.main.activity_tip6.*

class KotlinTip6 : Activity() {

    /*    * 自动根据 layout 的 id 生成对应的 view    * */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_tip6)
        tip6Tv.text = "Auto find view for TextView"
        tip6Img.setImageBitmap(null)
        tip6Btn.setOnClickListener {
            test()
        }
    }

    private fun test(){
        tip6Tv.text = "update"
    }

}
```

像上面代码这样，Activity 里的三个 View 自动生成了，不用再去声明，然后 findViewById，然后转型赋值，是不是减少了很多没必要的代码，让代码非常的干净。

Why？原理是什么？插件帮我们做了什么？

要看原理还是将上面的代码转为 java 语言来理解，参照 tips4 提供的方式转换为如下的 java 代码：

```
public final class KotlinTip6 extends Activity {
   private HashMap _$_findViewCache;

   protected void onCreate(@Nullable Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      this.setContentView(2131296284);
      TextView var10000 = (TextView)this._$_findCachedViewById(id.tip6Tv);
      Intrinsics.checkExpressionValueIsNotNull(var10000, "tip6Tv");
      var10000.setText((CharSequence)"Auto find view for TextView");

((ImageView)this._$_findCachedViewById(id.tip6Img)).setImageBitmap((Bitmap)null);

((Button)this._$_findCachedViewById(id.tip6Btn)).setOnClickListener((OnClickListene
r)(new OnClickListener() {
         public final void onClick(View it) {
            KotlinTip6.this.test();
         }
      }));
   }

   private final void test() {
      TextView var10000 = (TextView)this._$_findCachedViewById(id.tip6Tv);
      Intrinsics.checkExpressionValueIsNotNull(var10000, "tip6Tv");
      var10000.setText((CharSequence) "update");
   }

   public View _$_findCachedViewById(int var1) {
      if (this._$_findViewCache == null) {
         this._$_findViewCache = new HashMap();
      }

      View var2 = (View)this._$_findViewCache.get(Integer.valueOf(var1));
      if (var2 == null) {
         var2 = this.findViewById(var1);
         this._$_findViewCache.put(Integer.valueOf(var1), var2);
      }

      return var2;
   }
```

```
    public void _$_clearFindViewByIdCache() {
        if (this._$_findViewCache != null) {
            this._$_findViewCache.clear();
        }
    }
}
```

如上面的代码所示，在编译阶段，插件会帮我们生成视图缓存，视图由一个 Hashmap 结构的_$_findViewCache 变量缓存， 会根据对应的 id 先从缓存里查找，缓存没命中再去真正调用 findViewById 查找出来，再存在 HashMap 中。

子 View 或者 include 标签中 findViewById

子子 View 或者 include 标签中，同样可以省略 findViewById，但需要主要默认的 activity 的布局 import 是不会将这个 include 的 View 引入进来

```
<include layout="@layout/layout_tip6"/>

//include layout
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/test_inside_id"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/app_name"/>

</FrameLayout>
```

需要我们引入对应的 View 的 id，像这样 import kotlinx.android.synthetic.main.layout_tip6.*

```
//导入插件生成的 Viewimport kotlinx.android.synthetic.main.activity_tip6.*//include
layout 的 Viewimport kotlinx.android.synthetic.main.layout_tip6.*

test_inside_id.text = "Test include"
```

在 Fragment 中 findViewById

在 Fragment 中也类似，但有一点需要注意但地方，例子如下：

```
class Tip6Fragment : Fragment() {
```

```
    override fun onCreateView(inflater: LayoutInflater?, container: ViewGroup?,
savedInstanceState: Bundle?): View? {
        val view = inflater?.inflate(R.layout.fragment_tip6, container, false)
        /*        * 这时候不能在 onCreateView 方法里用 view，需要在 onViewCreate 里，原理
是插件用了 getView 来 findViewById        * */
        // tip6Tv.text = "test2"
        return view
    }

    /*    * 需要在 onViewCreate 里，原理是插件用了 getView 来 findViewById    * */
    override fun onViewCreated(view: View?, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        tip6Tv.text = "test"
    }
}
```

如上所示，Fragment 需要注意，不能在 onCreateView 方法里用 view，不然会出现空指针异常，需要在 onViewCreate 里，原理是插件用了 getView 来 findViewById， 我们看看将上面的代码转成 java 后的代码：

```
public final class Tip6Fragment extends Fragment {
   private HashMap _$_findViewCache;

   @Nullable
   public View onCreateView(@Nullable LayoutInflater inflater, @Nullable ViewGroup
container, @Nullable Bundle savedInstanceState) {
      View view = inflater != null?inflater.inflate(2131296286, container, false):null;
      return view;
   }

   public void onViewCreated(@Nullable View view, @Nullable Bundle savedInstanceState)
{
      super.onViewCreated(view, savedInstanceState);
      TextView var10000 = (TextView)this._$_findCachedViewById(id.tip6Tv);
      Intrinsics.checkExpressionValueIsNotNull(var10000, "tip6Tv");
      var10000.setText((CharSequence)"test");
   }

   public View _$_findCachedViewById(int var1) {
      if (this._$_findViewCache == null) {
         this._$_findViewCache = new HashMap();
      }

      View var2 = (View)this._$_findViewCache.get(Integer.valueOf(var1));
      if (var2 == null) {
         View var10000 = this.getView();
```

```
            if (var10000 == null) {
               return null;
            }

            var2 = var10000.findViewById(var1);
            this._$_findViewCache.put(Integer.valueOf(var1), var2);
         }

         return var2;
      }

      public void _$_clearFindViewByIdCache() {
         if (this._$_findViewCache != null) {
            this._$_findViewCache.clear();
         }

      }

      // $FF: synthetic method
      public void onDestroyView() {
         super.onDestroyView();
         this._$_clearFindViewByIdCache();
      }
}
```

跟 Activity 中类似，会有一个 View 的 HashMap，关键不同的地方在_$*findCachedViewById 里面，需要 getView 获得当前 Fragment 的 View， 故在 onViewCreated 中 getView 还是空的，原理就好理解了。另外在 onDestroyView 会调用*$_clearFindViewByIdCache 方法清掉缓存。

Tip7-利用局部函数抽取重复代码

回到目录

局部函数抽取代码

Kotlin 中提供了函数的嵌套，在函数内部还可以定义新的函数。这样我们可以在函数中嵌套这些提前的函数，来抽取重复代码。如下面的案例所示: 详见案例代码 KotlinTip7

```
class User(val id: Int, val name: String, val address: String, val email: String)
fun saveUser(user: User) {
    if (user.name.isEmpty()) {
        throw IllegalArgumentException("Can't save user ${user.id}: empty Name")
    }
```

```
    if (user.address.isEmpty()) {
        throw IllegalArgumentException("Can't save user ${user.id}: empty Address")
    }
    if (user.email.isEmpty()) {
        throw IllegalArgumentException("Can't save user ${user.id}: empty Email")
    }
    // save to db ...
}
```

上面的代码在判断 name、address 等是否为空的处理其实很类似。这时候，我们可以利用在函数内部嵌套的声明一个通用的判空函数将相同的代码抽取到一起：

```
/** 利用局部函数抽取相同的逻辑，去除重复的代码* */fun saveUser2(user: User) {
    fun validate(value: String, fildName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException("Can't save user ${user.id}: empty
$fildName")
        }
    }

    validate(user.name, "Name")
    validate(user.address, "Address")
    validate(user.email, "Email")
    // save to db ...
}
```

扩展函数抽取代码

除了利用嵌套函数去抽取，此时，其实也可以用扩展函数来抽取，如下所示：

```
/** 利用扩展函数抽取逻辑* */fun User.validateAll() {
    fun validate(value: String, fildName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException("Can't save user $id: empty $fildName")
        }
    }

    validate(name, "Name")
    validate(address, "Address")
    validate(email, "Email")
}
fun saveUser3(user: User) {
    user.validateAll()
    // save to db ...
}
```

Tip8-使用数据类来快速实现 model 类

回到目录

在 java 中要声明一个 model 类需要实现很多的代码，首先需要将变量声明为 private，然后需要实现 get 和 set 方法，还要实现对应的 hashcode equals toString 方法等，如下所示：详见案例代码 Tip8

```
public static class User {

    private String name;
    private int age;
    private int gender;
    private String address;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getGender() {
        return gender;
    }

    public void setGender(int gender) {
        this.gender = gender;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
```

```
        }

        @Override
        public String toString() {
            return "User{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    ", gender=" + gender +
                    ", address='" + address + '\'' +
                    '}';
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            User user = (User) o;

            if (age != user.age) return false;
            if (gender != user.gender) return false;
            if (name != null ? !name.equals(user.name) : user.name != null) return false;
            return address != null ? address.equals(user.address) : user.address ==
null;
        }

        @Override
        public int hashCode() {
            int result = name != null ? name.hashCode() : 0;
            result = 31 * result + age;
            result = 31 * result + gender;
            result = 31 * result + (address != null ? address.hashCode() : 0);
            return result;
        }
    }
```

这段代码 Java 需要 70 行左右，而如果用 kotlin，只需要一行代码就可以做到。

```
/** Kotlin 会为类的参数自动实现 get set 方法* */class User(val name: String, val age: Int,
val gender: Int, var address: String)
/** 用 data 关键词来声明一个数据类，除了会自动实现 get set，还会自动生成 equals hashcode
toString* */data class User2(val name: String, val age: Int, val gender: Int, var address:
String)
```

对于 Kotlin 中的类，会为它的参数自动实现 get set 方法。而如果加上 data 关键字，还会自动生成 equals hashcode toString。原理其实数据类中的大部分代码都是模版代码，Kotlin 聪明的将这个模版代码的实现放在了编译器处理的阶段。

Tip9-用类委托来快速实现装饰器模式

回到目录

通过继承的实现容易导致脆弱性，例如如果需要修改其他类的一些行为，这时候 Java 中的一种策略是采用**装饰器模式**：创建一个新类，实现与原始类一样的接口并将原来的类的实例作为一个成员变量。 与原始类拥有相同行为的方法不用修改，只需要直接转发给原始类的实例。如下所示： 详见案例代码 KotlinTip9

```
/** 常见的装饰器模式，为了修改部分的函数，却需要实现所有的接口函数* */class
CountingSet<T>(val innerSet: MutableCollection<T> = HashSet<T>()) :
MutableCollection<T> {

    var objectAdded = 0

    override val size: Int
        get() = innerSet.size

    /*    * 需要修改的方法    * */
    override fun add(element: T): Boolean {
        objectAdded++
        return innerSet.add(element)
    }

    /*    * 需要修改的方法    * */
    override fun addAll(elements: Collection<T>): Boolean {
        objectAdded += elements.size
        return innerSet.addAll(elements)
    }

    override fun contains(element: T): Boolean {
        return innerSet.contains(element)
    }

    override fun containsAll(elements: Collection<T>): Boolean {
        return innerSet.containsAll(elements)
    }

    override fun isEmpty(): Boolean {
        return innerSet.isEmpty()
    }
```

```
    override fun clear() {
        innerSet.clear()
    }

    override fun iterator(): MutableIterator<T> {
        return innerSet.iterator()
    }

    override fun remove(element: T): Boolean {
        return innerSet.remove(element)
    }

    override fun removeAll(elements: Collection<T>): Boolean {
        return innerSet.removeAll(elements)
    }

    override fun retainAll(elements: Collection<T>): Boolean {
        return innerSet.retainAll(elements)
    }

}
```

如上所示，想要修改 HashSet 的某些行为函数 add 和 addAll，需要实现 MutableCollection 接口的所有方法，将这些方法转发给 innerSet 去具体的实现。虽然只需要修改其中的两个方法，其他代码都是模版代码。 **只要是重复的模版代码，Kotlin 这种全新的语法糖就会想办法将它放在编译阶段再去生成。** 这时候可以用到**类委托 by 关键字**，如下所示：

```
/** 通过 by 关键字将接口的实现委托给 innerSet 成员变量，需要修改的函数再去 override 就可以了*
*/class CountingSet2<T>(val innerSet: MutableCollection<T> = HashSet<T>()) :
MutableCollection<T> by innerSet {

    var objectAdded = 0

    override fun add(element: T): Boolean {
        objectAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        objectAdded += elements.size
        return innerSet.addAll(elements)
    }
}
```

通过 by 关键字将接口的实现委托给 innerSet 成员变量，需要修改的函数再去 override 就可以了，通过类委托将 10 行代码就可以实现上面接近 100 行的功能，简洁明了，去掉了模版代码。

Tip10-Lambda 表达式简化 OnClickListener

回到目录

详见案例代码 KotlinTip10 lambda 表达式可以简化我们的代码。以 Android 中常见的 OnClickListener 来说明，在 Java 中我们一般这样设置：

```
TextView textView = new TextView(context);
textView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // handle click
    }
});
```

Java 中需要声明一个匿名内部类去处理，这种情况可以用 lambda 表达式来简化。

- lambda 表达式一般长这样
    - { x:Int, y:Int -> x+y }
    - 参数 -> 表达式 并且始终在大括号中
    - it 作为默认参数名
    - lambda 捕捉，当捕捉 final 变量时，它的值和 lambda 代码一起存储
    - 非 final 变量，它的值被封装在一个特殊的包装器中，这个包装器的引用会和 lambda 代码一起存储

我们来看看 Kotlin 中的例子：

```
val textView = TextView(context)

/*    * 传统方式    * */
textView.setOnClickListener(object : android.view.View.OnClickListener {
    override fun onClick(v: android.view.View?) {
        // handle click
    }
})

/*    * lambda 的方式    * */
textView.setOnClickListener({ v ->
    {
        // handle click
    }
```

```
    })
```

当 lambda 的参数没有使用时可以省略，省略的时候用 it 来替代

```
    /*    * lambda 的参数如果没有使用可以省略，省略的时候用 it 来替代    * */
    textView.setOnClickListener({
        // handle click
    })
```

lambda 在参数的最后一个的情况可以将之提出去

```
    /*    * lambda 在参数的最后一个的情况可以将之提出去    * */
    textView.setOnClickListener() {
        // handle click
    }
```

lambda 提出去之后，函数如果没有其他参数括号可以省略

```
    /*    * lambda 提出去之后，函数如果没有其他参数括号可以省略    * */
    textView.setOnClickListener {
        // handle click
    }
```

我们再看看如果自己去实现一个带 lambda 参数的函数应该怎么去定义：

```
interface OnClickListener {
    fun onClick()
}
class View {
    var listener: OnClickListener? = null;

    /*    * 传统方式    * */
    fun setOnClickListener(listener: OnClickListener) {
        this.listener = listener
    }

    fun doSth() {
        // some case:
        listener?.onClick()
    }

    /*    * 声明 lambda 方式，listener: () -> Unit    * */
    fun setOnClickListener(listener: () -> Unit) {

    }
}
```

在函数参数中需要声明 lambda 的类型后，再调用该函数的时候就可以传入一个 lambda 表达式了。

Tip11-with 函数来简化代码

回到目录

- with 函数原型：

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()
```

- with 函数并不是扩展函数，返回值是最后一行，可以直接调用对象的方法

Kotlin 中可以用 with 语句来省略同一个变量的多次声明，例如下面的函数 详见案例代码 KotlinTip11

```
/**打印字母表函数，在函数内 result 变量在好几处有使用到* */fun alphabet(): String {
    val result = StringBuilder()
    result.append("START\n")
    for (letter in 'A'..'Z') {
        result.append(letter)
    }
    result.append("\nEND")
    return result.toString()
}
```

在上面的函数中，result 变量出现了 5 次，如果用 with 语句，可以将这 5 次都不用再出现了，我们来一步一步地看是怎么实现的：

```
/** 通过 with 语句，将 result 作为参数传入，在内部就可以通过 this 来表示 result 变量了* */fun
alphabet2(): String {
    val result = StringBuilder()
    return with(result) {
        this.append("START\n")
        for (letter in 'A'..'Z') {
            this.append(letter)
        }
        this.append("\nEND")
        this.toString()
    }
}
```

通过 with 语句，将 result 作为参数传入，在内部就可以通过 this 来表示 result 变量了，而且这个 this 是可以省略的

```
/** 通过 with 语句,将 result 参数作为参数,在内部 this 也可以省略掉* */fun alphabet3(): String
{
    val result = StringBuilder()
    return with(result) {
        append("START\n")
```

```
        for (letter in 'A'..'Z') {
            append(letter)
        }
        append("\nEND")
        toString()
    }
}
```

在内部 this 省略掉后，现在只有一个 result 了，这个其实也是没必要的，于是出现了下面的最终版本：

```
/** 通过 with 语句，可以直接将对象传入，省掉对象的声明* */fun alphabet4(): String {
    return with(StringBuilder()) {
        append("START\n")
        for (letter in 'A'..'Z') {
            append(letter)
        }
        append("\nEND")
        toString()
    }
}
```

像上面这样，我们可以把同一个变量的显式调用从 5 次变为 0 次，发现 Kotlin 的魅力了吧。

Tip12-apply 函数来简化代码

回到目录

- apply 函数原型：

```
inline fun <T> T.apply(block: T.() -> Unit): T { block(); return this }
```

- apply 函数，在函数范围内，可以任意调用该对象的任意方法，并返回该对象

除了用上面的 with 可以简化同一个变量的多次声明，还可以用 apply 关键字，我们来改造一下 tip11 中的函数： 详见案例代码 KotlinTip12

```
/** 用 apply 语句简化代码，在 apply 的大括号里可以访问类的公有属性和方法* */fun alphabet5()
= StringBuilder().apply {
    append("START\n")
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nEND")
}.toString()
```

像上面这样的，通过 apply 后，在 apply 的大括号里可以访问类的公有属性和方法。这在对应类的初始化是非常方便的，例如下面的例子

```
/** 用 apply 语句简化类的初始化，在类实例化的时候，就可以通过 apply 把需要初始化的步骤全部实现，非常的简洁* */fun testApply(context: Context) {
    var imgView = ImageView(context).apply {
        setBackgroundColor(0)
        setImageBitmap(null)
    }

    var textView = TextView(context).apply {
        text = "content"
        textSize = 20.0f
        setPadding(10, 0, 0, 0)
    }

    var user = User().apply {
        age = 15
        name = "Jack"
        val a = address
        address = "bbb"
    }
}
```

在类实例化的时候，就可以通过 apply 把需要初始化的步骤全部实现，非常的简洁

Tip13-在编译阶段避免掉 NullPointerException

回到目录

可空类型和不可空类型

NullPointerException 是 Java 程序员非常头痛的一个问题，我们知道 Java 中分受检异常和非受检异常，NullPointerException 是非受检异常，也就是说 NullPointerException 不需要显示的去 catch 住，往往在运行期间，程序就可能报出一个 NullPointerException 然后 crash 掉，Kotlin 作为一门高效安全的语言，它尝试在编译阶段就把空指针问题显式的检测出来，把问题留在了编译阶段，让程序更加健壮。 详见案例代码 KotlinTip13

- Kotlin 中类型分为可空类型和不可空类型，通过？代表可空，不带？代表不可为空

```
fun testNullType() {
    val a: String = "aa"
    /*    * a 是非空类型，下面的给 a 赋值为 null 将会编译不通过     * */
    // a = null
    a.length
```

```
    /*    * ？声明是可空类型，可以赋值为 null    * */
    var b: String? = "bb"
    b = null

    /*    * b 是可空类型，直接访问可空类型将编译不通过，需要通过?.或者!!.来访问    * */
    // b.length
    b?.length
    b!!.length
}
```

- 对于一个不可为空类型：如果直接给不可为空类型赋值一个可能为空的对象就在编译阶段就不能通过
- 对于一个可空类型：通过？声明，在访问该类型的时候直接访问不能编译通过，需要通过?.或者!!.
    - ?. 代表着如果该类型为空的话就返回 null 不做后续的操作，如果不为空的话才会去访问对应的方法或者属性
    - !!. 代表着如果该类型为空的话就抛出 NullPointerException，如果不为空就去访问对应的方法或者属性， 所以只有在很少的特定场景才用这种符号，代表着程序不处理这种异常的 case 了，会像 java 代码一样抛出 NullPointerException。 而且代码中一定不用出现下面这种代码，会让代码可读性很差而且如果有空指针异常，我们也不能马上发现是哪空了：

```
    /*    * 不推荐这样的写法：链式的连续用!!.    * */
    val user = User()
    user!!.name!!.subSequence(0,5)!!.length
```

对应一个可空类型，每次对它的访问都需要带上?.判断

```
val user: User? = User()

    /*    * 每次访问都用用?.判断    * */
    user?.name
    user?.age
    user?.toString()
```

但这样多了很多代码，kotlin 做了一些优化，

```
    /*    * 或者提前判断是否为空，如果不为空在这个分支里会自动转化为非空类型就可以直接访问了
* */
    if (user != null) {
        user.name
        user.age
        user.toString()
    }
```

通过 if 提前判断类型是否为空，如果不为空在这个分支里会**自动转化为非空类型**就可以直接访问了。

let 语句简化对可空对象对访问

- let 函数原型：

```
inline fun <T, R> T.let(block: (T) -> R): R = block(this)
```

- let 函数默认当前这个对象作为闭包的 it 参数，返回值是函数里面最后一行，或者指定 return。

上面的代码还可以用?.let 语句进行，如下所示：

```
    /*     * 通过 let 语句，在?.let 之后，如果为空不会有任何操作，只有在非空的时候才会执行 let 之后的操作     * */
    user?.let {
        it.name
        it.age
        it.toString()
    }
```

通过 let 语句，在?.let 之后，如果为空不会有任何操作，只有在非空的时候才会执行 let 之后的操作

Elvis 操作符 ?: 简化对空值的处理

如果值可能为空，对空值的处理可能会比较麻烦，像下面这样：

```
/** 对空值的处理* */fun testElvis(input: String?, user: User?) {
    val a: Int?
    if (input == null) {
        a = -1
    } else {
        a = input?.length
    }

    if (user == null) {
        var newOne = User()
        newOne.save()
    } else {
        user.save()
    }
}
```

Elvis 操作符?:能够简化上面的操作，?:符号会在符号左边为空的情况才会进行下面的处理，不为空则不会有任何操作。**跟?.let 正好相反**，例如我们可以用两行代码来简化上面从操作：

```
/** * Elvis 操作符 ?: 简化对空值的处理 */fun testElvis2(input: String?, user: User?) {
    val b = input?.length ?: -1;
    user?.save() ?: User().save()
}
```

Tip14-运算符重载

回到目录

Kotlin 支持对运算符的重载，这对于对一些对象的操作更加灵活直观。

- 使用 operator 来修饰 plus\minus 函数
- 可重载的二元算术符
    - A * B times
    - A / B div
    - A % B mod
    - A + B plus
    - A - B minus

以下面对坐标点 Point 的案例说明怎么去重载运算符： 详见案例代码 KotlinTip14

```
class Point(val x: Int, val y: Int) {

    /*    * plus 函数重载对 Point 对象的加法运算符    * */
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }

    /*    * minus 函数重载对 Point 对象的减法运算符    * */
    operator fun minus(other: Point): Point {
        return Point(x - other.x, y - other.y)
    }

    override fun toString(): String {
        return "[x:$x, y:$y]"
    }

}
```

如上所示，通过 plus 函数重载对 Point 对象的加法运算符，通过 minus 函数重载对 Point 对象的减法运算符，然后就可以用+、-号对两个对象进行操作了：

```
fun testOperator() {
```

```
    val point1 = Point(10, 10)
    val point2 = Point(4, 4)
    val point3 = point1 + point2
    println(point3)
    println(point1 - point2)
}
```

Tip15-高阶函数简化代码

回到目录

- 高阶函数：以另一个函数作为参数或者返回值的函数
- 函数类型
    - (Int, String) -> Unit
    - 参数类型->返回类型 Unit 不能省略

```
    val list = listOf(2, 5, 10)
    /*    * 传入函数来过滤    * */
    println(list.filter { it > 4 })

    /*    * 定义函数类型    * */
    val sum = { x: Int, y: Int -> x + y }
    val action = { println(42) }

    val sum2: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
    val action2: () -> Unit = { println(42) }
```

函数作为参数

函数作为参数，即高阶函数中，函数的参数可以是一个函数类型，例如要定义一个函数，该函数根据传入的操作函数来对 2 和 3 做相应的处理。 详见案例代码 KotlinTip15

```
/** 定义对 2 和 3 的操作函数* */fun twoAndThree(operator: (Int, Int) -> Int) {
    val result = operator(2, 3)
    println("Result:$result")
}
fun test03() {
    twoAndThree { a, b -> a + b }
    twoAndThree { a, b -> a * b }
}
```

operator 是函数类型，函数的具体类型为(Int, Int) -> Int，即输入两个 Int 返回一个 Int 值。定义完了后就可以像上面这样使用了。 再举一个例子，实现 String 类的字符过滤：

```
/** 函数作为参数，实现 String 类的字符过滤* */fun String.filter(predicate: (Char) ->
Boolean): String {
    val sb = StringBuilder()
    for (index in 0 until length) {
        val element = get(index)
        if (predicate(element)) sb.append(element)
    }
    return sb.toString()
}
fun test04() {
    println("12eafsfsfdbzzsa".filter { it in 'a'..'f' })
}
```

像上面这样 predicate 是函数类型，它会根据传入的 char 来判断得到一个 Boolean 值。

函数作为返回值

函数作为返回值也非常实用，例如我们的需求是根据不同的快递类型返回不同计价公式，普通快递和高级快递的计价规则不一样，这时候我们可以将计价规则函数作为返回值：

```
enum class Delivery {
    STANDARD, EXPEDITED
}
/** 根据不同的运输类型返回不同的快递方式* */fun getShippingCostCalculator(delivery:
Delivery): (Int) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { 6 + 2.1 * it }
    }
    return { 1.3 * it }
}
fun test05() {
    val calculator1 = getShippingCostCalculator(Delivery.EXPEDITED)
    val calculator2 = getShippingCostCalculator(Delivery.STANDARD)
    println("Ex costs ${calculator1(5)}")
    println("St costs ${calculator2(5)}")
}
```

如果是普通快递，采用 1.3 * it 的规则计算价格，如果是高级快递按照 6 + 2.1 * it 计算价格，根据不同的类型返回不同的计价函数。

Tip16-用 Lambda 来简化策略模式

回到目录

策略模式是常见的模式之一，java 的例子如下。 详见案例代码 Tip16

```
/**     * 定义策略接口      */
    public interface Strategy {
        void doSth();
    }

    /**     * A 策略      */
    public static class AStrategy implements Strategy {
        @Override
        public void doSth() {
            System.out.println("Do A Strategy");
        }
    }

    /**     * B 策略      */
    public static class BStrategy implements Strategy {
        @Override
        public void doSth() {
            System.out.println("Do B Strategy");
        }
    }

    /**     * 策略实施者      */
    public static class Worker {

        private Strategy strategy;

        public Worker(Strategy strategy) {
            this.strategy = strategy;
        }

        public void work() {
            System.out.println("START");
            if (strategy != null) {
                strategy.doSth();
            }
            System.out.println("END");
        }
    }
```

如上面的例子所示，有 A、B 两种策略，Worker 根据不同的策略做不同的工作，使用策略时：

```
    Worker worker1 = new Worker(new AStrategy());
    Worker worker2 = new Worker(new BStrategy());
    worker1.work();
    worker2.work();
```

在java中实现这种策略模式难免需要先定义好策略的接口，然后根据接口实现不同的策略，在 Kotlin 中完全可以用用 Lambda 来简化策略模式，上面的例子用 Kotlin 实现：

```
/** * 策略实施者 * @param strategy lambda 类型的策略 */class Worker(private val strategy:
() -> Unit) {
    fun work() {
        println("START")
        strategy.invoke()
        println("END")
    }
}
/** 测试* */fun testStrategy() {
    val worker1 = Worker({
        println("Do A Strategy")
    })
    val bStrategy = {
        println("Do B Strategy")
    }
    val worker2 = Worker(bStrategy)
    worker1.work()
    worker2.work()
}
```

不需要先定义策略的接口，直接把策略以 lambda 表达式的形式传进来就行了。

4. 使用 Kotlin Reified 让泛型更简单安全

我们在编程中，出于复用和高效的目的，我们使用到了泛型。但是泛型在 JVM 底层采取了类型擦除的实现机制，Kotlin 也是这样。然后这也带来了一些问题和对应的解决方案。这里我们介绍一个 reified 用法，来实现更好的处理泛型。

类型擦除

如下面的代码，在编译成 class 文件后，就采用了类型擦除

```java
public class TestTypeErasure {
    public List<String> list = new ArrayList<>();

    public void test() {
        list.add("123");
        System.out.println(list.get(0));
    }
}
```

- list 实例真实的保存是多个 `Object`
- `list.add("123")`实际上是`"123"`作为 `Object` 存入集合中
- `System.out.println(list.get(0));`是从 `list` 实例中读取出来 `Object` 然后转换成 `String`才能使用的

辅助证明的字节码内容

```
Compiled from "TestTypeErasure.java"
public class TestTypeErasure {
  //省略部分代码


  public void test();
    Code:
       0: aload_0
       1: getfield      #4                  // Field list:Ljava/util/List;
       4: ldc           #5                  // String 123
       6: invokeinterface #6,  2            // InterfaceMethod java/util/List.add:(Ljava/lang/Objec
      11: pop
      12: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      15: aload_0
      16: getfield      #4                  // Field list:Ljava/util/List;
      19: iconst_0
      20: invokeinterface #8,  2            // InterfaceMethod java/util/List.get:(I)Ljava/lang/Obj
      25: checkcast     #9                  // class java/lang/String
      28: invokevirtual #10                 // Method java/io/PrintStream.println:(Ljava/lang/Strin
      31: return
}
```

其中

- 第 6 行对应的 `6: invokeinterface #6, 2 // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z` 对应添加元素参数的类型为 `Object`
- 第 20 行对应的 `20: invokeinterface #8, 2 // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;` 对应的获取元素的返回类型为 `Object`
- 第 25 行为进行类型转换操作

类型擦除带来的问题

安全问题:未检查的异常

```
//unchecked cast
fun <T> Int.toType(): T? {
    return (this as? T)
}
```

- 上面的代码会导致编译器警告 `unchecked cast`
- 上面的代码由于在转换类型时，没有进行检查，所以有可能会导致运行时崩溃

当我们执行这样的代码时

```
fun testCast() {
    println(1.toType<String>()?.substring(0))
}
```

会得到 `java.lang.Integer cannot be cast to java.lang.String` 的类型错误。

显式传递 Class

针对前面的问题，我们最常用的办法就是显式传递 class 信息

```
//need pass class explicitly
fun <T> Any.toType(clazz: Class<T>): T? {
    return if (clazz.isInstance(this)) {
        this as? T
    } else {
        null
    }
}
```

但是显式传递 Class 信息也会感觉有一些问题，尤其是下面这段代码

```
fun <T> Bundle.plus(key: String, value: T, clazz: Class<T>) {
    when(clazz) {
        Long::class.java -> putLong(key, value as Long)
        String::class.java -> putString(key, value as String)
        Char::class.java -> putChar(key, value as Char)
        Int::class.java -> putInt(key, value as Int)
    }
}
```

- 上面的代码（传 value 值和 clazz）我们会感觉到明显的有一些笨拙，不够智能。
- 但是这也是基于 Java 的类型擦除机制导致无法再运行时得到 `T` 的类型信息，无法改进（至少在 Java 中）

可能导致更多方法的产生

同时，由于上面的显式传递 Class 信息比较麻烦和崩溃，我们有时候会增加更多的方法，比如下面的这样。

```
class Bundle {
    fun putInt(key: String, value: Int) {
        println("Bundle.putInt key=$key;value=$value")
    }

    fun putLong(key: String, value: Long) {

    }

    fun putString(key: String, value: String) {

    }

    fun putChar(key: String, value: Char) {

    }
}
```

- 上面的 `putInt`,`putLong`,`putString` 和 `putChar` 没有泛型引入
- 我们没有排除显式传递 Class 参数之外的优雅实现，比如我们只提供一个叫做 `put(key: String,value: T)`

reified 方式

不过，好在 Kotlin 有一个对应的解决方案，这就是我们今天文章标题提到的 reified（中文意思：具体化）

使用 reified 很简单，主要分为两步

- 在泛型类型前面增加 `reified`
- 在方法前面增加 `inline`（必需的）

接下来我们使用 reified 改进之前的方法

类型转换改进后的代码

```
//much better way using reified
inline fun <reified T> Any.asType(): T? {
    return if (this is T) {
        this
    } else {
        null
    }
}
```

方法传参不需要多余传递参数类型信息

```
inline fun <reified T> Bundle.plus(key: String, value: T) {
    when(value) {
        is Long -> putLong(key, value)
        is String -> putString(key, value)
        is Char -> putChar(key, value)
        is Int-> putInt(key, value)
    }
}
```

reified 实现原理

不是说，泛型是使用了类型擦除么，为什么运行时能得到 `T` 的类型信息呢？

是的，采用类型擦除没有错，至于能在运行时得到 `T` 的类型信息是如何做到的，就需要了解 `reified` 的内部机制了。

其原理为

- Kotlin 编译器会将 reified 方法 `asType` 内联(inline)到调用的地方(call-site)
- 方法被内联到调用的地方后，泛型 T 会被替换成具体的类型

所以 **reified 使得泛型的方法假装在运行时能够获取泛型的类信息**

为了便于理解，我们举个例子，如下是我们的代码

```
fun testCast2() {
    println(1.asType<String>()?.substring(0))
}
```

对应的反编译后的 java 代码

```
public static final void testCast2() {
      Object $this$asType$iv = 1;
      int $i$f$asType = false;
      String var10000 = (String)($this$asType$iv instanceof String ? $this$asType$iv : null);
      String var3;
      /**
   * 后续的代码对应的Kotlin代码(也包含了部分call-site的逻辑，比如substring)
   return if (this is T) {
        this
      } else {
        null
      }
      */

      //inline和reified替换开始
      if ((String)($this$asType$iv instanceof String ? $this$asType$iv : null) != null) {
         var3 = var10000;
         byte var4 = 0;
         boolean var2 = false;
         if (var3 == null) {
            throw new TypeCastException("null cannot be cast to non-null type java.lang.String");
         }

         var10000 = var3.substring(var4);
         Intrinsics.checkExpressionValueIsNotNull(var10000, "(this as java.lang.String).substring(s
      } else {
         var10000 = null;
      }
      //inline和reified替换结束
      var3 = var10000;
      $i$f$asType = false;
      System.out.println(var3);
   }
```

all in(lined)?

既然是 inline，应该是把被 inline 的方法全部提取到调用处(call-site)吧？

- 是的，通常是这样，不过 reified 可能有一些差异
- reified 方法并不会完全 inline 所有的方法实现，而是更加智能一些的类型匹配中断提取。

```
fun testBundlePlusLong() {
    Bundle().plus("hello", 1000L)
}

fun testBundlePlusString() {
    Bundle().plus("hello", "World")
}

fun testBundlePlusChar() {
    Bundle().plus("hello", 'h')
}

fun testBundlePlusInt() {
    Bundle().plus("hello", 1)
}
```

再次贴一些 Bundle.plus 实现

```
inline fun <reified T> Bundle.plus(key: String, value: T) {
    when(value) {
        is Long -> putLong(key, value)
        is String -> putString(key, value)
        is Char -> putChar(key, value)
        is Int-> putInt(key, value)
    }
}
```

上面的 when 表达式的类型检查次序依次为

- Long
- String
- Char
- Int

反编译后的方法如下(类型不同，提取的方法体也不同)

```
public static final void testBundlePlusLong() {
      Bundle $this$plus$iv = new Bundle();
      String key$iv = "hello";
      long value$iv = 1000L;
      int $i$f$plus = false;
      //第一个就是Long类型，无需包含后面的检查代码
      $this$plus$iv.putLong(key$iv, value$iv);
   }

   public static final void testBundlePlusString() {
      Bundle $this$plus$iv = new Bundle();
      String key$iv = "hello";
      Object value$iv = "World";
      int $i$f$plus = false;
      //不是Long类型，需要继续匹配，找到String类型，终止inline后续代码
      if (value$iv instanceof Long) {
         $this$plus$iv.putLong(key$iv, ((Number)value$iv).longValue());
      } else {
         $this$plus$iv.putString(key$iv, value$iv);
      }

   }

   public static final void testBundlePlusChar() {
      Bundle $this$plus$iv = new Bundle();
      String key$iv = "hello";
      Object value$iv = 'h';
      int $i$f$plus = false;
      //不是Long类型，需要继续匹配，
      if (value$iv instanceof Long) {
         $this$plus$iv.putLong(key$iv, ((Number)value$iv).longValue());
          //不是String类型，需要继续匹配，
      } else if (value$iv instanceof String) {
         $this$plus$iv.putString(key$iv, (String)value$iv);
      } else {
         //找到String类型，终止inline后续代码
         $this$plus$iv.putChar(key$iv, value$iv);
      }

   }

   public static final void testBundlePlusInt() {
      Bundle $this$plus$iv = new Bundle();
```

```
41
42      public static final void testBundlePlusInt() {
43         Bundle $this$plus$iv = new Bundle();
44         String key$iv = "hello";
45         Object value$iv = 1;
46         int $i$f$plus = false;
47         //最差的一种情况，inline全部的方法体实现
48         if (value$iv instanceof Long) {
49            $this$plus$iv.putLong(key$iv, ((Number)value$iv).longValue());
50         } else if (value$iv instanceof String) {
51            $this$plus$iv.putString(key$iv, (String)value$iv);
52         } else if (value$iv instanceof Character) {
53            $this$plus$iv.putChar(key$iv, (Character)value$iv);
54         } else {
55            $this$plus$iv.putInt(key$iv, ((Number)value$iv).intValue());
56         }
57
58      }
```

以上就是关于 reified 的内容，其实在 Kotlin 中有很多的特性是依赖于编译器的工作来实现的。

5. Kotlin 里的 Extension Functions 实现原理分析

Kotlin 里的 Extension Functions

Kotlin 里有所谓的扩展函数(Extension Functions)，支持给现有的 java 类增加函数。

- https://kotlinlang.org/docs/reference/extensions.html

比如给 `String` 增加一个 `hello` 函数，可以这样子写：

```
1fun String.hello(world : String) : String {
2    return "hello " + world + this.length;
3}
4fun main(args: Array<String>) {
5    System.out.println("abc".hello("world"));
6}
```

可以看到在 main 函数里，直接可以在 `String` 上调用 `hello` 函数。

执行后，输出结果是：

```
hello world3
```

可以看到在 `hello` 函数里的 `this` 引用的是`"abc"`。

刚开始看到这个语法还是比较新奇的 ,那么怎样实现的呢？**如果不同的库都增加了同样的函数会不会冲突？**

反编绎生成的字节码，结果是：

```
@NotNull
public static final String hello(@NotNull String $receiver, @NotNull String world) {
   return "hello " + world + $receiver.length();
}
public static final void main(@NotNull String[] args) {
   System.out.println(hello("abc", "world"));
}
```

可以看到，实际上是增加了一个 static public final 函数。

并且新增加的函数是在自己的类里的，并不是在 String 类里。即不同的库新增加的扩展函数都是自己类里的，不会冲突。

lombok 里的 @ExtensionMethod 实现

lombok 里也提供了类似的`@ExtensionMethod` 支持。

- https://projectlombok.org/features/experimental/ExtensionMethod

和上面的例子一样，给 String 类增加一个 `hello` 函数：

- 需要定义一个 `class Extensions`

- 再用`@ExtensionMethod`声明

```
class Extensions {
    public static String hello(String receiver, String world) {
        return "hello " + world + receiver.length();
    }
}
@ExtensionMethod({
        Extensions.class
})

public class Test {
    public static void main(String[] args) {
        System.out.println("abc".hello("world"));
    }
}
```

执行后，输出结果是：

```
hello world3
```

可以看到在`hello`函数里，第一个参数`String receiver`就是`"abc"`本身。

和上面 kotlin 的例子不一样的是，kotlin 里直接可以用`this`。

生成的字节码反编绎结果是：

```
class Extensions {
  public static String hello(String receiver, String world) {
    return "hello " + world + receiver.length();
  }
}
public class Test {
  public static void main(String[] args) {
    System.out.println(Extensions.hello("abc", "world"));
  }
}
```

可以看到所谓的`@ExtensionMethod`实际上也是一个语法糖。

设计动机

- https://kotlinlang.org/docs/reference/extensions.html#motivation

据 kotlin 的文档：各种 `FileUtils`，`StringUtils` 类很麻烦。

比如像下面处理 `List`，在 java 里可以用 `java.util.Collections`

```
// Java
Collections.swap(list, Collections.binarySearch(list,
    Collections.max(otherList)),
    Collections.max(list));
```

简化下 import，可以变为

```
// Java
swap(list, binarySearch(list, max(otherList)), max(list));
```

但是还不够清晰，各种 `import *`也是比较烦的。利用扩展函数，可以直接这样子写：

```
// Java
list.swap(list.binarySearch(otherList.max()), list.max());
```

总结

- kotlin 的 Extension Functions 和 lombok 的`@ExtensionMethod`实际上都是增加 public static final 函数
- 不同的库增加的同样的 Extension Functions 不会冲突
- 设计的动机是减少各种 utils 类。

6. Kotlin 系列之顶层函数和属性

遇到的问题

我们都知道，Java 中，所有的代码都是依托于类而存在，我们所谓的函数作为类的方法，我们所谓的属性作为类的属性。但是在有些情况下，我们发现有些方法可能不是属于某一个特定类，有些属性也不是属于某一个特定的类。所以我们就创建了很多的 Java 工具类和属性的常量类，就像下面这样。

Java 代码

```
public class Constant {

    public static final String BASE_URL = "http://www.xxx.top:8080";

}

public class StrUtil {

    public static String joinToStr(Collection<String> collection){

        //...

        return "";

    }

}
```

其实上面的类只是为了承载我们的静态属性和方法，作为了静态方法和属性的容器，这就是我们目前遇到的问题，一大堆无用的容器类，那让我们看看 Kotlin 中是如何处理这个问题的。

顶层函数

见名知意，原来在 Java 中，类处于顶层，类包含属性和方法，在 Kotlin 中，函数站在了类的位置，我们可以直接把函数放在代码文件的顶层，让它不从属于任何类。就像下面这样，我们在一个 Str.kt 文件中写入如下的 Kotlin 代码。

Kotlin 代码

```
package util


fun joinToStr(collection: Collection<String>): String{
    //....
    return "";
}
```

请注意，我们把它放在了 util 包中，这在我们厚密啊调用这个类时非常重要。

让我们现来看看在另一个 Kotlin 类中怎么调用。

Kotlin 代码

```
import util.joinToStr


fun main(args: Array<String>){
    joinToStr(collection = listOf("123", "456"))
}
```

看到了吗？我们可以通过 import 包名.函数名来导入我们将要使用的函数，然后就可以直接使用了，是不是超级方便。那我们再来看看在 Java 中如何调用上面的方法。

Java 代码

```
import java.util.ArrayList;
import util.StrKt;
```

```
public class Main {

    public static void main(String[] args) {

        StrKt.joinToStr(new ArrayList<>());

    }

}
```

因为在 Java 中，类还是必须要存在的，所以编译器将 Str.kt 文件里的代码放在了一个 StrKt 的类中，然后把我们定义的 Kotlin 的函数作为静态方法放在其中，所以在 Java 中是先通过 mport 导入这个类，然后通过类名.方法名来调用。

可能有时候你觉得 Kotlin 为你自动生成的这个类名不好，那你可以通过@file:JvmName 注解来自定义类名，就像下面这样。

```
@file:JvmName("StrUtil")

package util


fun joinToStr(collection: Collection<String>): String{

    //....

    return "";

}
```

而且要注意，这个注解必须放在文件的开头，包名的前面。

在 Java 中导入类和调用的时候就要使用我们自定义的类名来进行操作了，就像这样。

```
import java.util.ArrayList;

import util.StrUtil;
```

```
public class Main {

    public static void main(String[] args) {

        StrUtil.joinToStr(new ArrayList<>());

    }

}
```

顶层属性

了解了顶层函数，下面再看看顶层属性。顶层属性也就是把属性直接放在文件顶层，不依附于类。我们可以在顶层定义的属性包括 var 变量和 val 常量，就像下面这样。

Kotlin 代码

```
package config


var count = 0


val REQUEST_URL = "http://localhost:8080/"


const val BASE_URL = "http://www.xxx.top/"
```

这里我定义了三个顶层属性，可能有些地方你还不太能看得懂，不急，我们先看看在 Kotlin 和 Java 中怎么用，然后我们再理解。

在 Kotlin 中使用

```
import config.count
```

```
fun main(args: Array<String>){

    //使用 var 变量
    count++

    //使用 val 常量
    config.REQUEST_URL

    //使用 const val 常量
    config.BASE_URL
}
```

你会发现在 Kotlin 中只要导入就可以直接使用了，与顶层属性的使用是一样的。

在 Java 中使用

```
import config.ApiConfigKt;

public class Main {
    public static void main(String[] args) {

        //使用 var 变量
        ApiConfigKt.setCount(12);
        System.out.println(ApiConfigKt.getCount());
```

```
        //使用 val 常量
        System.out.println(ApiConfigKt.getREQUEST_URL());

        //使用 const val 常量
        System.out.println(ApiConfigKt.BASE_URL);

    }
}
```

首先导入是肯定的，通过类名类调用这个也和顶层函数里面的规则一致。然后你会发现 Kotlin 为 var 变量生成了 get 和 set 方法，然后为 val 常量生成了 get 方法，最后我们的 const val 常量则等同于 Java 的 public static final 常量，可以直接通过类名.常量名来调用。

写在最后

Kotlin 中通过使用顶层函数和顶层属性帮助我们消除了 Java 中常见的静态工具类，使我们的代码更加整洁，值得一试。

————————————————————

原文链接：https://blog.csdn.net/bingjianIT/article/details/79134670

7.Kotlin 兼容 Java 遇到的最大的 “坑”

一个 Realm 的小例子

Realm 在 2016 年与 RxJava、Retrofit 这样的框架一起，在 Android 开发领域内着实小小的火了一把，如果大家对它不了解，没关系，传送门 biu ~ Realm

我们先按照官网的说明配置好 gradle 依赖，话说呀，这互联网发展这么快，新时代的框架一出来，逼格果断就体现在完善的构建和开发生态，你发布的东西还只是一个 jar 包，人家呢，早上了 maven 不说，还要搞几个 gradle 任务来方便你开发：

```
buildscript {
    ext.kotlin_version = '1.1.1'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.0'
        //这里将 realm 的 gradle 插件加入 gradle 构建的运行时
        classpath "io.realm:realm-gradle-plugin:3.0.0"
    }
}
...
apply plugin: 'com.android.application'
apply plugin: 'realm-android' //应用插件，Realm 会在这里添加自己的一些构建任务
...
```

其实 gradle 插件开发也是一个很有意思的话题，如果大家有需要，我后面也可以写几篇文章介绍下（悄悄告诉你们，其实我早就想写了，这不是 kotlin 版的 gradle 还没有正式发布么！）。

有了这个我们就可以开始写个 Realm 的 demo 了。

小明：等等！我还有一事不明，你怎么不添加 realm 的依赖就要开始写 demo 了啊！

艾玛，要么说小明人家就是明白人呢，我前面写了一大堆，只不过是添加了 gradle 构建的依赖而已，而我们的程序想要使用 realm，必须依赖 realm 的运行时库才行。那

这么说我是不是漏掉了什么？当然没有，怎么会呢，我这么聪（dou）明（bi）的人，我可是一步一步照着官网的步骤抄的！

其实呀，realm 的运行时依赖早在我们 apply plugin 的时候就已经添加进来的，realm-android 这个插件除了添加了一些它需要的 gradle 任务之外，也顺手帮我们把依赖添加了。嗯，就是酱紫，如果有那个同学学(xian)有(de)余(dan)力(teng)，可以翻一翻 realm 插件的源码。

来来来，赶紧看 demo，不然有些人该内急了~

首先在 Application 当中初始化它：

```
class App : Application() {
    override fun onCreate() {
        super.onCreate()
        Realm.init(this)
        Realm.setDefaultConfiguration(
                RealmConfiguration.Builder()
                .deleteRealmIfMigrationNeeded()
                .schemaVersion(1)
                .build())
    }
}
```

定义一个 User 类：

```
data class User(@PrimaryKey var id: Int, val name: String) :
RealmObject()
```

接着我们开始存数据和查数据啦：

```
add.setOnClickListener {
    Realm.getDefaultInstance().use {
        it.beginTransaction()
```

```
        val d = it.createObject(User::class.java,
it.where(User::class.java).count())
        d.name = "User ${d.id}"
        it.commitTransaction()
    }

}

query.setOnClickListener {
    Realm.getDefaultInstance().use {
        it.where(User::class.java).findAll().map {
            Log.d(TAG, it.toString())
        }
    }
}
```

想得挺美，结果呢？编译不通过。

```
Error:A default public constructor with no argument must be declared
in User if a custom constructor is declared.
```

无参构造方法

这就让我想到上周的文章，那篇文章里面我们其实就发现症结根本不是什么 Int 和 Integer，而是无参构造方法。JavaBean 是 Java 的一个概念，我其实甚至有些觉得 Java 的设计者们通过 JavaBean 这样的概念来弥补语言本身的缺陷——不管怎样，JavaBean 是不能没有无参构造的，。

Kotlin 呢，语言层面就有类似于 JavaBean 的东西，那就是 data class，这俩孩子实在太像了，以至于大家经常把 data class 当做 JavaBean 来使。嗯，你信不信 Kotlin 的设计者也是这么想的呢？当然，用 data class 这样一个名正言顺的“亲儿子”数据类来替代 JavaBean 这么个语言层面没有任何支持和认可的“野孩子”，应该算是 JavaBean 莫大的荣幸了，可问题又出在 Java 语言本身构造方法滥用的潜在问题上了。

在 Java 中，构造方法真心是一个很没有存在感的东西，大家总是根据自己的喜好来随意的定义很多个构造方法的版本，而最终忽视掉它们的内在联系，导致没有正常走完初始化逻辑的实例满天飞，这家伙如果是导弹，我估计也不需要解放军就可以直接把台湾给统一了。

说了这么多，我主要是想吐槽两个点：第一个就是 Java 本身语言设计层面几乎没有任何照顾到数据类的体现（可千万别说 clone 和 Serialize），第二个就是 Java 对其对象的实例化过程的把控太过于儿戏。

这两点呢，Kotlin 都做的很好，我现在写 Kotlin 经常被迫认真思考一个类该如何正确初始化，这显然对于我们的程序结构和逻辑梳理有莫大的好处。可是结果呢？Java 时代的那些框架们受不了了。Kotlin 背靠着 Java 这座大山，Java 就像它的父母一样，父母的观念再老再陈旧，Kotlin 也得做好自己该做的，一方面是向现在看来陈旧但在过去已经非常革命的观念致敬，另一方面嘛，如果 Java 不支持个几十万首付，Kotlin 能买得起房吗？

哇塞，我好能扯啊。

其实想要解决 default public constructor 这样的问题，Kotlin 官方已经想到了，那就是 noarg。嗯，我原以为我提一句 noarg 大家就会知道是什么了，看来是我想的简单了，毕竟这个东西在 1.0.6 才出来，当时我还在介绍这个版本的时候提到了它的使用方法，朋友们可能还没有接触过，没关系，下面我再贴一些写法，大家一看就明白：

首先你要做的就是定义一个注解：

```
annotation class PoKo
```

接着 gradle 配置一下脚本的依赖：

```
buildscript {
...
    dependencies {
        ...
         classpath
"org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
        ...
    }
}
```

加了运行时环境，那么我们就可以使用 noarg 插件了：

```
apply plugin: "kotlin-noarg"

noArg {
    annotation("net.println.kotlin.realm.PoKo")
}
```

配置完之后，PoKo 这个注解就有了超能力，所有被它标注的类在编译时都会生成一个无参的构造方法，于是我们给 User 加一个 PoKo 的注解：

```
@PoKo data class User(...) : RealmObject()
```

搞定，果断去编译一下！！

final 还是不 final，这是个问题

本来兴高采烈的以为不就是个无参构造的问题嘛，结果编译的时候又爆出了新的问题：

```
Error:(31, 61) error: cannot inherit from final User
```

好家伙，这究竟发生了什么。。原来 Realm 在编译的时候生成了一个类：

```
public class UserRealmProxy extends net.println.kotlin.realm.User
    implements RealmObjectProxy, UserRealmProxyInterface
```

这个类要继承我这个 User 类，结果就报错了。

下面是理（che）论（dan）时间。我们说在 C++ 当中给合适的变量、函数参数、函数返回值甚至函数加上 const 是个好习惯，大家没有意见吧？同样的，Java 当中给那些不变的量、不能被继承的类、不能被覆写的方法加上 final 也是个好习惯，大家也没有意见吧？那么问题来了，大家有几个人这么干了？是不是不到万不得已，才懒得写那个 final 呢，五个字母呢，你是想累死宝宝啊？我就知道 Effective Java 这本书看了也白看，因为大家经常明知道什么是好习惯却还是要对着干，这个不是因为大家不喜欢好习惯，而是因为坚持好习惯需要成本！不瞒各位说，我中午为了坚持午休的好习惯，牺牲了跟组里面的小伙伴一起开黑上分的机会，还得装着拥护“人民 ri 报”关于“小学生打排位太坑”的评论，我容易么我。。

嗯，扯远了。还是说 final 的事儿，Kotlin 就做的很好，它默认所有的类、变量、方法都是 final 的，想要继承？来，过来申请我给你审批。。。你看，这样从根儿解决问题，我们再也不用为了坚持好习惯而发愁了，因为我们根本不需要坚持，难道你想要坚持坏习惯嘛？

可是 Java 及其框架们呢？原来到北京买房有钱就行，现在呢，商住都不让买了啊（什么？你说广州都不让卖了？）。那叫一个不适应，这可不是得闹事儿么。

Kotlin 官方考虑到 Java 帮它出首付买房的事儿，想了想算了，还是出个什么插件，解决下这个问题吧，于是 allopen 闪亮登场！allopen 的原理跟 noarg 极其类似，

它是在编译器对指定的类进行去“final”化，你别看你写代码的时候 User 还是个 final 的类，不过编译成字节码之后这天呀可就变了。

关于 allopen 的使用，跟 noarg 简直不要太像，先定义一个注解：

```
annotation class PoKo // How old r U!
```

可以跟 noarg 公用同一个注解，也可以自己另外单独定义一个，这个不要紧。

接着 gradle 配置搞起：

```
buildscript {
    ...
    dependencies {
        ...
        classpath
"org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
        ...
    }
}
```

接着就是应用插件，配置注解一气呵成：

```
apply plugin: "kotlin-allopen"

allOpen {
    annotation("net.println.kotlin.realm.PoKo")
}
```

编译运行~

ps：如果加了 allopen 和 noarg 之后编译仍然提示原来的错误，记得狠狠地 clean 一下才行哈。

认真脸：真的 “坑” 吗

前面说了 Kotlin 的两个 “坑”，都是关于 data class 的。有人认为这么说 Kotlin 不公平，毕竟人家 Kotlin 也是可以写出下面的代码的：

```
class User{
    var id: String? = null
    var name: String? = null
}
```

尽管你在为 Kotlin 打抱不平，不过如果你真要写这样的代码，我建议你还是用 Java 吧。你不属于 Kotlin。。。

Kotlin 这么美的语言，怎么能写这么丑陋的东西呢？这就好比有人说为什么空类型强转为非空类型一定要两个感叹号呢，用一个不就够了么，两个看起来好丑呀！

```
var user: User? = getUser()
user!!.name = "小明" //小明，他们说你丑！
```

有人回答说：明明这就是丑陋的东西，为什么要美化？掩盖事物的本质只能让事情变得更糟糕！

我们用 Kotlin 企图兼容 Java 的做法，本来就是权宜之计，兼容必然带来新旧两种观念的冲突以及丑陋的发生，这么说来，我倒是更愿意期待 Kotlin Native 的出现了。

8.Kotlin 的协程

协程是什么

协程并不是 Kotlin 提出来的新概念，其他的一些编程语言，例如：Go、Python 等都可以在语言层面上实现协程，甚至是 Java，也可以通过使用扩展库来间接地支持协程。

当在网上搜索协程时，我们会看到：

- Kotlin 官方文档说「本质上，协程是轻量级的线程」。
- 很多博客提到「不需要从用户态切换到内核态」、「是协作式的」等等。

作为 Kotlin 协程的初学者，这些概念并不是那么容易让人理解。这些往往是作者根据自己的经验总结出来的，只看结果，而**不管过程就不容易理解协程**。

「协程 Coroutines」源自 Simula 和 Modula-2 语言，这个术语早在 1958 年就被 Melvin Edward Conway 发明并用于构建汇编程序，说明**协程是一种编程思想**，并不局限于特定的语言。

Go 语言也有协程，叫 Goroutines，从英文拼写就知道它和 Coroutines 还是有些差别的（设计思想上是有关系的），否则 Kotlin 的协程完全可以叫 Koroutines 了。

因此，对一个新术语，我们需要知道什么是「标准」术语，什么是变种。

当我们讨论协程和线程的关系时，很容易**陷入中文的误区**，两者都有一个「程」字，就觉得有关系，其实就英文而言，Coroutines 和 Threads 就是两个概念。

从 Android 开发者的角度去理解它们的关系：

- 我们所有的代码都是跑在线程中的，而线程是跑在进程中的。
- 协程没有直接和操作系统关联，但它不是空中楼阁，它也是跑在线程中的，可以是单线程，也可以是多线程。
- 单线程中的协程总的执行时间并不会比不用协程少。
- Android 系统上，如果在主线程进行网络请求，会抛出 `NetworkOnMainThreadException`，对于在主线程上的协程也不例外，这种场景使用协程还是要切线程的。

协程设计的初衷是为了解决并发问题，让「协作式多任务」实现起来更加方便。这里就先不展开「协作式多任务」的概念，等我们学会了怎么用再讲。

视频里讲到，协程就是 Kotlin 提供的一套线程封装的 API，但并不是说协程就是为线程而生的。

不过，我们学习 Kotlin 中的协程，一开始确实可以从线程控制的角度来切入。因为在 Kotlin 中，协程的一个典型的使用场景就是线程控制。就像 Java 中的 `Executor` 和 Android 中的 `AsyncTask`，Kotlin 中的协程也有对 Thread API 的封装，让我们可以在写代码时，不用关注多线程就能够很方便地写出并发操作。

在 Java 中要实现并发操作通常需要开启一个 `Thread`：

```
new Thread(new Runnable() {
      @Override
      public void run() {
            ...
      }
}).start();
```

这里仅仅只是开启了一个新线程，至于它何时结束、执行结果怎么样，我们在主线程中是无法直接知道的。

Kotlin 中同样可以通过线程的方式去写：

```
Thread({
      ...
}).start()
```

可以看到，和 Java 一样也摆脱不了直接使用 `Thead` 的那些困难和不方便：

- 线程什么时候执行结束
- 线程间的相互通信
- 多个线程的管理

我们可以用 Java 的 `Executor` 线程池来进行线程管理：

```
val executor = Executors.newCachedThreadPool()
executor.execute({
      ...
})
```

用 Android 的 `AsyncTask` 来解决线程间通信：

```
object : AsyncTask<T0, T1, T2> {
      override fun doInBackground(vararg args: T0): String { ... }
      override fun onProgressUpdate(vararg args: T1) { ... }
      override fun onPostExecute(t3: T3) { ... }
}
```

`AsyncTask` 是 Android 对线程池 `Executor` 的封装，但它的缺点也很明显：

- 需要处理很多回调，如果业务多则容易陷入「回调地狱」。
- 硬是把业务拆分成了前台、中间更新、后台三个函数。

看到这里你很自然想到使用 RxJava 解决回调地狱，它确实可以很方便地解决上面的问题。

RxJava，准确来讲是 ReactiveX 在 Java 上的实现，是一种响应式程序框架，我们通过它提供的「Observable」的编程范式进行链式调用，可以很好地消除回调。

使用协程，同样可以像 Rx 那样有效地消除回调地狱，不过无论是设计理念，还是代码风格，两者是有很大区别的，协程在写法上和普通的顺序代码类似。

> 这里并不会比较 RxJava 和协程哪个好，或者讨论谁取代谁的问题，我这里只给出一个建议，你最好都去了解下，因为协程和 Rx 的设计思想本来就不同。

下面的例子是使用协程进行网络请求获取用户信息并显示到 UI 控件上：

```
launch({
      val user = api.getUser() //
      nameTv.text = user.name   //
})
```

这里只是展示了一个代码片段，`launch` 并不是一个顶层函数，它必须在一个对象中使用，我们之后再讲，这里只关心它内部业务逻辑的写法。

`launch` 函数加上实现在 `{}` 中具体的逻辑，就构成了一个协程。

通常我们做网络请求，要不就传一个 callback，要不就是在 IO 线程里进行阻塞式的同步调用，而在这段代码中，上下两个语句分别工作在两个线程里，但写法上看起来和普通的单线程代码一样。

这里的 `api.getUser` 是一个**挂起函数**，所以能够保证 `nameTv.text` 的正确赋值，这就涉及到了协程中最著名的「非阻塞式挂起」。这个名词看起来不是那么容易理解，我们后续的文章会专门对这个概念进行讲解。现在先把这个概念放下，只需要记住协程就是这样写的就行了。

这种「用同步的方式写异步的代码」看起来很方便吧，那么我们来看看协程具体好在哪。

协程好在哪

开始之前

在讲之前，我们需要先了解一下「闭包」这个概念，调用 Kotlin 协程中的 API，经常会用到闭包写法。

其实闭包并不是 Kotlin 中的新概念，在 Java 8 中就已经支持。

我们先以 `Thread` 为例，来看看什么是闭包：

```
// 创建一个 Thread 的完整写法
Thread(object : Runnable {
      override fun run() {
            ...
     }
})

// 满足 SAM，先简化为
Thread({
     ...
})

// 使用闭包，再简化为
Thread {
     ...
}
```

形如 `Thread {...}` 这样的结构中 `{}` 就是一个闭包。

在 Kotlin 中有这样一个语法糖：当函数的最后一个参数是 lambda 表达式时，可以将 lambda 写在括号外。这就是它的闭包原则。

在这里需要一个类型为 `Runnable` 的参数，而 `Runnable` 是一个接口，且只定义了一个函数 `run`，这种情况满足了 Kotlin 的 SAM，可以转换成传递一个 lambda 表达式（第二段），因为是最后一个参数，根据闭包原则我们就可以直接写成 `Thread {...}`（第三段） 的形式。

对于上文所使用的 `launch` 函数，可以通过闭包来进行简化 ：

```
launch {
```

```
    ...
}
```

基本使用

前面提到，`launch` 函数不是顶层函数，是不能直接用的，可以使用下面三种方法来创建协程：

```
// 方法一，使用 runBlocking 顶层函数
runBlocking {
    getImage(imageId)
}

// 方法二，使用 GlobalScope 单例对象
//
GlobalScope.launch {
    getImage(imageId)
}

// 方法三，自行通过 CoroutineContext 创建一个 CoroutineScope 对象
//
val coroutineScope = CoroutineScope(context)
coroutineScope.launch {
    getImage(imageId)
}
```

- 方法一通常适用于单元测试的场景，而业务开发中不会用到这种方法，因为它是线程阻塞的。
- 方法二和使用 `runBlocking` 的区别在于不会阻塞线程。但在 Android 开发中同样不推荐这种用法，因为它的生命周期会和 app 一致，且不能取消（什么是协程的取消后面的文章会讲）。
- 方法三是比较推荐的使用方法，我们可以通过 `context` 参数去管理和控制协程的生命周期（这里的 `context` 和 Android 里的不是一个东西，是一个更通用的概念，会有一个 Android 平台的封装来配合使用）。

  关于 `CoroutineScope` 和 `CoroutineContext` 的更多内容后面的文章再讲。

  协程最常用的功能是并发，而并发的典型场景就是多线程。可以使用 `Dispatchers.IO` 参数把任务切到 IO 线程执行：

```
coroutineScope.launch(Dispatchers.IO) {
    ...
}
```

也可以使用 `Dispatchers.Main` 参数切换到主线程：

```
coroutineScope.launch(Dispatchers.Main) {
    ...
}
```

所以在「协程是什么」一节中讲到的异步请求的例子完整写出来是这样的：

```
coroutineScope.launch(Dispatchers.Main) {    // 在主线程开启协程
    val user = api.getUser() // IO 线程执行网络请求
    nameTv.text = user.name    // 主线程更新 UI
}
```

而通过 Java 实现以上逻辑，我们通常需要这样写：

```
api.getUser(new Callback<User>() {
    @Override
    public void success(User user) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                nameTv.setText(user.name);
            }
        })
    }

    @Override
    public void failure(Exception e) {
        ...
    }
});
```

这种回调式的写法，打破了代码的顺序结构和完整性，读起来相当难受。

协程的「1 到 0」

对于回调式的写法，如果并发场景再复杂一些，代码的嵌套可能会更多，这样的话维护起来就非常麻烦。但如果你使用了 Kotlin 协程，多层网络请求只需要这么写：

```
coroutineScope.launch(Dispatchers.Main) {           // 开始协程：主线程
      val token = api.getToken()                          // 网络请求：
IO 线程
      val user = api.getUser(token)                      // 网络请求：
IO 线程
      nameTv.text = user.name                             // 更新 UI：
主线程
}
```

如果遇到的场景是多个网络请求需要等待所有请求结束之后再对 UI 进行更新。比如以下两个请求：

```
api.getAvatar(user, callback)
api.getCompanyLogo(user, callback)
```

如果使用回调式的写法，那么代码可能写起来既困难又别扭。于是我们可能会选择妥协，通过先后请求代替同时请求：

```
api.getAvatar(user) { avatar ->
      api.getCompanyLogo(user) { logo ->
            show(merge(avatar, logo))
      }
}
```

在实际开发中如果这样写，本来能够并行处理的请求被强制通过串行的方式去实现，可能会导致等待时间长了一倍，也就是性能差了一倍。

而如果使用协程，可以直接把两个并行请求写成上下两行，最后再把结果进行合并即可：

```
coroutineScope.launch(Dispatchers.Main) {
      //
      val avatar = async { api.getAvatar(user) }      // 获取用户头像
      val logo = async { api.getCompanyLogo(user) } // 获取用户所在公
司的 logo
      val merged = suspendingMerge(avatar, logo)      // 合并结果
      //
```

```
        show(merged) // 更新 UI
}
```

可以看到，即便是比较复杂的并行网络请求，也能够通过协程写出结构清晰的代码。需要注意的是 `suspendingMerge` 并不是协程 API 中提供的方法，而是我们自定义的一个可「挂起」的结果合并方法。至于挂起具体是什么，可以看下一篇文章。

让复杂的并发代码，写起来变得简单且清晰，是协程的优势。

这里，两个没有相关性的后台任务，因为用了协程，被安排得明明白白，互相之间配合得很好，也就是我们之前说的「协作式任务」。

本来需要回调，现在直接没有回调了，这种从 1 到 0 的设计思想真的妙哉。

在了解了协程的作用和优势之后，我们再来看看协程是怎么使用的。

协程怎么用

在项目中配置对 Kotlin 协程的支持

在使用协程之前，我们需要在 `build.gradle` 文件中增加对 Kotlin 协程的依赖：

- 项目根目录下的 `build.gradle`：

```
buildscript {
        ...
        //
        ext.kotlin_coroutines = '1.3.1'
        ...
}
```

- Module 下的 `build.gradle`：

```
dependencies {
        ...
        //
        implementation
"org.jetbrains.kotlinx:kotlinx-coroutines-core:$kotlin_coroutines"
        //
        implementation
"org.jetbrains.kotlinx:kotlinx-coroutines-android:$kotlin_coroutines"
```

```
    ...
}
```

Kotlin 协程是以官方扩展库的形式进行支持的。而且，我们所使用的「核心库」和 「平台库」的版本应该保持一致。

- 核心库中包含的代码主要是协程的公共 API 部分。有了这一层公共代码，才使得协程在各个平台上的接口得到统一。
- 平台库中包含的代码主要是协程框架在具体平台的具体实现方式。因为多线程在各个平台的实现方式是有所差异的。

完成了以上的准备工作就可以开始使用协程了。

开始使用协程

协程最简单的使用方法，其实在前面章节就已经看到了。我们可以通过一个 `launch` 函数实现线程切换的功能：

```
//
coroutineScope.launch(Dispatchers.IO) {
    ...
}
```

这个 `launch` 函数，它具体的含义是：我要创建一个新的协程，并在指定的线程上运行它。这个被创建、被运行的所谓「协程」是谁？就是你传给 `launch` 的那些代码，这一段连续代码叫做一个「协程」。

所以，什么时候用协程？当你需要切线程或者指定线程的时候。你要在后台执行任务？切！

```
launch(Dispatchers.IO) {
    val image = getImage(imageId)
}
```

然后需要在前台更新界面？再切！

```
coroutineScope.launch(Dispatchers.IO) {
    val image = getImage(imageId)
    launch(Dispatch.Main) {
        avatarIv.setImageBitmap(image)
    }
}
```

好像有点不对劲？这不还是有嵌套嘛。

如果只是使用 `launch` 函数，协程并不能比线程做更多的事。不过协程中却有一个很实用的函数：`withContext` 。这个函数可以切换到指定的线程，并在闭包内的逻辑执行结束之后，自动把线程切回去继续执行。那么可以将上面的代码写成这样：

```
coroutineScope.launch(Dispatchers.Main) {          //
      val image = withContext(Dispatchers.IO) {    //
            getImage(imageId)                                  //
      }
      avatarIv.setImageBitmap(image)                    //
}
```

这种写法看上去好像和刚才那种区别不大，但如果你需要频繁地进行线程切换，这种写法的优势就会体现出来。可以参考下面的对比：

```
// 第一种写法
coroutineScope.launch(Dispachers.IO) {
      ...
      launch(Dispachers.Main){
            ...
            launch(Dispachers.IO) {
                  ...
                  launch(Dispacher.Main) {
                        ...
                  }
            }
      }
}

// 通过第二种写法来实现相同的逻辑
coroutineScope.launch(Dispachers.Main) {
      ...
      withContext(Dispachers.IO) {
            ...
      }
      ...
      withContext(Dispachers.IO) {
            ...
      }
      ...
```

```
}
```

由于可以"自动切回来"，消除了并发代码在协作时的嵌套。由于消除了嵌套关系，我们甚至可以把 `withContext` 放进一个单独的函数里面：

```
launch(Dispachers.Main) {                    //
      val image = getImage(imageId)
      avatarIv.setImageBitmap(image)        //
}
//
fun getImage(imageId: Int) = withContext(Dispatchers.IO) {
     ...
}
```

这就是之前说的「用同步的方式写异步的代码」了。

不过如果只是这样写，编译器是会报错的：

```
fun getImage(imageId: Int) = withContext(Dispatchers.IO) {
      // IDE 报错 Suspend function'withContext' should be called only
from a coroutine or another suspend funcion
}
```

意思是说，`withContext` 是一个 `suspend` 函数，它需要在协程或者是另一个 `suspend` 函数中调用。

suspend

`suspend` 是 Kotlin 协程最核心的关键字，几乎所有介绍 Kotlin 协程的文章和演讲都会提到它。它的中文意思是「暂停」或者「可挂起」。如果你去看一些技术博客或官方文档的时候，大概可以了解到：「代码执行到 `suspend` 函数的时候会『挂起』，并且这个『挂起』是非阻塞式的，它不会阻塞你当前的线程。」

上面报错的代码，其实只需要在前面加一个 `suspend` 就能够编译通过：

```
//
suspend fun getImage(imageId: Int) = withContext(Dispatchers.IO) {
     ...
```

```
}
```

本篇文章到此结束，而 `suspend` 具体是什么，「非阻塞式」又是怎么回事，函数怎么被挂起，这些疑问的答案，将在下一篇文章全部揭晓。

9.Kotlin 协程「挂起」的本质

上期回顾

在协程上一期中我们知道了下面知识点：

- 协程究竟是什么
- 协程到底好在哪里
- 协程具体怎么用

大部分情况下，我们都是用 `launch` 函数来创建协程，其实还有其他两个函数也可以用来创建协程：

- `runBlocking`
- `async`

`runBlocking` 通常适用于单元测试的场景，而业务开发中不会用到这个函数，因为它是线程阻塞的。

接下来我们主要来对比 `launch` 与 `async` 这两个函数。

- 相同点：它们都可以用来启动一个协程，返回的都是 `Coroutine`，我们这里不需要纠结具体是返回哪个类。
- 不同点：`async` 返回的 `Coroutine` 多实现了 `Deferred` 接口。

关于 `Deferred` 更深入的知识就不在这里过多阐述，它的意思就是延迟，也就是结果稍后才能拿到。

我们调用 `Deferred.await()` 就可以得到结果了。

接下来我们继续看看 `async` 是如何使用的，先回忆一下上期中获取头像的场景：

```
2
coroutineScope.launch(Dispatchers.Main) {
3
    //
4
    val avatar: Deferred = async { api.getAvatar(user) }    // 获取用户头像
5
    val logo: Deferred = async { api.getCompanyLogo(user) } // 获取用户所在公司的 logo
6
    //
7
    show(avatar.await(), logo.await())                      // 更新 UI
8
}
```

可以看到 avatar 和 logo 的类型可以声明为 `Deferred`，通过 `await` 获取结果并且更新到 UI 上显示。

`await` 函数签名如下：

```
1
2
public suspend fun await(): T
```

前面有个关键字是之前没有见过的 —— `suspend`，这个关键字就对应了上期最后我们留下的一个问号：协程最核心的那个「非阻塞式」的「挂起」到底是怎么回事？

所以接下来，我们的核心内容就是来好好说一说这个「挂起」。

「挂起」的本质

协程中「挂起」的对象到底是什么？挂起线程，还是挂起函数？都不对，**我们挂起的对象是协程。**

还记得协程是什么吗？启动一个协程可以使用 `launch` 或者 `async` 函数，协程其实就是这两个函数中闭包的代码块。

`launch`，`async` 或者其他函数创建的协程，在执行到某一个 `suspend` 函数的时候，这个协程会被「suspend」，也就是被挂起。

那此时又是从哪里挂起？**从当前线程挂起。换句话说，就是这个协程从正在执行它的线程上脱离。**

注意，不是这个协程停下来了！是脱离，当前线程不再管这个协程要去做什么了。

suspend 是有暂停的意思，但我们在协程中应该理解为：当线程执行到协程的 suspend 函数的时候，暂时不继续执行协程代码了。

我们先让时间静止，然后兵分两路，分别看看这两个互相脱离的线程和协程接下来将会发生什么事情：

线程：

前面我们提到，挂起会让协程从正在执行它的线程上脱离，具体到代码其实是：

协程的代码块中，线程执行到了 suspend 函数这里的时候，就暂时不再执行剩余的协程代码，跳出协程的代码块。

那线程接下来会做什么呢？

如果它是一个后台线程：

- 要么无事可做，被系统回收
- 要么继续执行别的后台任务

跟 Java 线程池里的线程在工作结束之后是完全一样的：回收或者再利用。

如果这个线程它是 Android 的主线程，那它接下来就会继续回去工作：也就是一秒钟 60 次的界面刷新任务。

一个常见的场景是，获取一个图片，然后显示出来：

```
// 主线程中
GlobalScope.launch(Dispatchers.Main) {
    val image = suspendingGetImage(imageId)   // 获取图片
    avatarIv.setImageBitmap(image)                // 显示出来
}
```

```
suspend fun suspendingGetImage(id: String) =
withContext(Dispatchers.IO) {
                                                                    9
  ...
                                                                   10
}
```

这段执行在主线程的协程，它实质上会往你的主线程 `post` 一个 `Runnable`，这个 `Runnable` 就是你的协程代码：

```
                                                                    1
                                                                    2
handler.post {
                                                                    3
  val image = suspendingGetImage(imageId)
                                                                    4
  avatarIv.setImageBitmap(image)
                                                                    5
}
```

当这个协程被挂起的时候，就是主线程 `post` 的这个 `Runnable` 提前结束，然后继续执行它界面刷新的任务。

关于线程，我们就看完了。 这个时候你可能会有一个疑问，那 `launch` 包裹的剩下代码怎么办？
所以接下来，我们来看看协程这一边。

协程：

线程的代码在到达 `suspend` 函数的时候被掐断，接下来协程会从这个 `suspend` 函数开始继续往下执行，不过是在**指定的线程**。

谁指定的？是 `suspend` 函数指定的，比如我们这个例子中，函数内部的 `withContext` 传入的 `Dispatchers.IO` 所指定的 IO 线程。

`Dispatchers` 调度器，它可以将协程限制在一个特定的线程执行，或者将它分派到一个线程池，或者让它不受限制地运行，关于 `Dispatchers` 这里先不展开了。

那我们平日里常用到的调度器有哪些？

常用的 `Dispatchers`，有以下三种：

- `Dispatchers.Main`：Android 中的主线程
- `Dispatchers.IO`：针对磁盘和网络 IO 进行了优化，适合 IO 密集型的任务，比如：读写文件，操作数据库以及网络请求
- `Dispatchers.Default`：适合 CPU 密集型的任务，比如计算

回到我们的协程，它从 `suspend` 函数开始脱离启动它的线程，继续执行在 `Dispatchers` 所指定的 IO 线程。

紧接着在 `suspend` 函数执行完成之后，协程为我们做的最爽的事就来了：会**自动帮我们把线程再切回来**。

这个「切回来」是什么意思？

我们的协程原本是运行在**主线程**的，当代码遇到 suspend 函数的时候，发生线程切换，根据 `Dispatchers` 切换到了 IO 线程；

当这个函数执行完毕后，线程又切了回来，「切回来」也就是协程会帮我再 `post` 一个 `Runnable`，让我剩下的代码继续回到主线程去执行。

我们从线程和协程的两个角度都分析完成后，终于可以对协程的「挂起」suspend 做一个解释：

协程在执行到有 suspend 标记的函数的时候，会被 suspend 也就是被挂起，而所谓的被挂起，就是切个线程；

不过区别在于，**挂起函数在执行完成之后，协程会重新切回它原先的线程**。

再简单来讲，在 Kotlin 中所谓的挂起，就是**一个稍后会被自动切回来的线程调度操作**。

这个「切回来」的动作，在 Kotlin 里叫做 resume，恢复。

通过刚才的分析我们知道：挂起之后是需要恢复。

而恢复这个功能是协程的，如果你不在协程里面调用，恢复这个功能没法实现，所以也就回答了这个问题：为什么挂起函数必须在协程或者另一个挂起函数里被调用。

再细想下这个逻辑：一个挂起函数要么在协程里被调用，要么在另一个挂起函数里被调用，那么它其实直接或者间接地，总是会在一个协程里被调用的。

所以，要求 `suspend` 函数只能在协程里或者另一个 suspend 函数里被调用，

还是为了要让协程能够在 `suspend` 函数切换线程之后再切回来。

怎么就「挂起」了？

我们了解到了什么是「挂起」后，再接着看看这个「挂起」是怎么做到的。

先随便写一个自定义的 `suspend` 函数：

```
1
2
suspend fun suspendingPrint() {
3
    println("Thread: ${Thread.currentThread().name}")
4
}
5

6
I/System.out: Thread: main
```

输出的结果还是在主线程。

为什么没切换线程？因为它不知道往哪切，需要我们告诉它。

对比之前例子中 `suspendingGetImage` 函数代码：

```
1
2
```

```
//
3
suspend fun suspendingGetImage(id: String) =
withContext(Dispatchers.IO) {
4
  ...
5
}
```

我们可以发现不同之处其实在于 `withContext` 函数。

其实通过 `withContext` 源码可以知道，它本身就是一个挂起函数，它接收一个 `Dispatcher` 参数，依赖这个 `Dispatcher` 参数的指示，你的协程被挂起，然后切到别的线程。

所以这个 `suspend`，其实并不是起到把任何把协程挂起，或者说切换线程的作用。

真正挂起协程这件事，是 Kotlin 的协程框架帮我们做的。

所以我们想要自己写一个挂起函数，仅仅只加上 `suspend` 关键字是不行的，还需要函数内部直接或间接地调用到 Kotlin 协程框架自带的 `suspend` 函数才行。

suspend 的意义？

这个 `suspend` 关键字，既然它并不是真正实现挂起，那它的作用是什么？

它其实是一个提醒。

函数的创建者对函数的使用者的提醒：我是一个耗时函数，我被我的创建者用挂起的方式放在后台运行，所以请在协程里调用我。

为什么 `suspend` 关键字并没有实际去操作挂起，但 Kotlin 却把它提供出来？

因为它本来就不是用来操作挂起的。

挂起的操作 —— 也就是切线程，依赖的是挂起函数里面的实际代码，而不是这个关键字。

所以这个关键字，**只是一个提醒**。

还记得刚才我们尝试自定义挂起函数的方法吗？

```
//
suspend fun suspendingPrint() {
    println("Thread: ${Thread.currentThread().name}")
}
```

如果你创建一个 `suspend` 函数但它内部不包含真正的挂起逻辑，编译器会给你一个提醒：`redundant suspend modifier`，告诉你这个 `suspend` 是多余的。

因为你这个函数实质上并没有发生挂起，那你这个 `suspend` 关键字只有一个效果：就是限制这个函数只能在协程里被调用，如果在非协程的代码中调用，就会编译不通过。

所以，创建一个 `suspend` 函数，为了让它包含真正挂起的逻辑，要在它内部直接或间接调用 Kotlin 自带的 `suspend` 函数，你的这个 `suspend` 才是有意义的。

怎么自定义 suspend 函数？

在了解了 `suspend` 关键字的来龙去脉之后，我们就可以进入下一个话题了：怎么自定义 `suspend` 函数。

这个「怎么自定义」其实分为两个问题：

- 什么时候需要自定义 `suspend` 函数？
- 具体该怎么写呢？

什么时候需要自定义 suspend 函数

如果你的某个函数比较耗时，也就是要等的操作，那就把它写成 `suspend` 函数。这就是原则。

耗时操作一般分为两类：I/O 操作和 CPU 计算工作。比如文件的读写、网络交互、图片的模糊处理，都是耗时的，通通可以把它们写进 `suspend` 函数里。

另外这个「耗时」还有一种特殊情况，就是这件事本身做起来并不慢，但它需要等待，比如 5 秒钟之后再做这个操作。这种也是 `suspend` 函数的应用场景。

具体该怎么写

给函数加上 `suspend` 关键字，然后在 `withContext` 把函数的内容包住就可以了。

提到用 `withContext` 是因为它在挂起函数里功能最简单直接：把线程自动切走和切回。

当然并不是只有 `withContext` 这一个函数来辅助我们实现自定义

的 `suspend` 函数，比如还有一个挂起函数叫 `delay`，它的作用是等待一段时间后再继续往下执行代码。

使用它就可以实现刚才提到的等待类型的耗时操作：

```
suspend fun suspendUntilDone() {
    while (!done) {
        delay(5)
    }
}
```

这些东西，在我们初步使用协程的时候不用立马接触，可以先把协程最基本的方法和概念理清楚。

总结

我们今天整个文章其实就在理清一个概念：什么是挂起？**挂起，就是一个稍后会被自动切回来的线程调度操作。**

好，关于协程中的「挂起」我们就解释到这里。

可能你心中还会存在一些疑惑：

- 协程中挂起的「非阻塞式」到底是怎么回事？
- 协程和 RxJava 在切换线程方面功能是一样的，都能让你写出避免嵌套回调的复杂并发代码，那协程还有哪些优势，或者让开发者使用协程的理由？

这些疑惑的答案，我们都会在下一篇中全部揭晓。

在协程系列的前两篇文章中，我们介绍了：

- 协程就是个线程框架
- 协程的挂起本质就是线程切出去再切回来

因为时间比较久远了，忘了也很正常，不过今天我们要说的这个「非阻塞式」的概念，即使你没有协程的基础，也可以看懂。

10. 到底什么是「非阻塞式」挂起？协程真的更轻量级吗？

什么是「非阻塞式挂起」

非阻塞式是相对阻塞式而言的。

编程语言中的很多概念其实都来源于生活，就像脱口秀的段子一样。

线程阻塞很好理解，现实中的例子就是交通堵塞，它的核心有 3 点：

- 前面有障碍物，你过不去（线程卡了）
- 需要等障碍物清除后才能过去（耗时任务结束）
- 除非你绕道而行（切到别的线程）

从语义上理解「非阻塞式挂起」，讲的是「非阻塞式」这个是挂起的一个特点，也就是说，协程的挂起，就是非阻塞式的，协程是不讲「阻塞式的挂起」的概念的。

我们讲「非阻塞式挂起」，其实它有几个前提：并没有限定在一个线程里说这件事，因为挂起这件事，本来就是涉及到多线程。

就像视频里讲的，阻塞不阻塞，都是针对单线程讲的，一旦切了线程，肯定是非阻塞的，你都跑到别的线程了，之前的线程就自由了，可以继续做别的事情了。

所以「非阻塞式挂起」，其实就是在讲协程在挂起的同时切线程这件事情。

为什么要讲非阻塞式挂起

既然第三篇说的「非阻塞式挂起」和第二篇的「挂起要切线程」是同一件事情，那还有讲的必要吗？

是有的。因为它在写法上和单线程的阻塞式是一样的。

协程只是在写法上「看起来阻塞」，其实是「非阻塞」的，因为在协程里面它做了很多工作，其中有一个就是帮我们切线程。

第二篇讲挂起，重点是说切线程先切过去，然后再切回来。

第三篇讲非阻塞式，重点是说线程虽然会切，但写法上和普通的单线程差不多。

让我们来看看下面的例子：

```
main {
      GlobalScope.launch(Dispatchers.Main) {
            //
            val user = suspendingRequestUser()
            updateView(user)
      }

      private suspend fun suspendingRequestUser() : User =
withContext(Dispatchers.IO) {
            api.requestUser()
      }
}
```

从上面的例子可以看到，耗时操作和更新 UI 的逻辑像写单线程一样放在了一起，只是在外面包了一层协程。

而正是这个协程解决了原来我们单线程写法会卡线程这件事。

阻塞的本质

首先，所有的代码本质上都是阻塞式的，而只有比较耗时的代码才会导致人类可感知的等待，比如在主线程上做一个耗时 50 ms 的操作会导致界面卡掉几帧，这种是我们人眼能观察出来的，而这就是我们通常意义所说的「阻塞」。

举个例子，当你开发的 app 在性能好的手机上很流畅，在性能差的老手机上会卡顿，就是在说同一行代码执行的时间不一样。

视频中讲了一个网络 IO 的例子，IO 阻塞更多是反映在「等」这件事情上，它的性能瓶颈是和网络的数据交换，你切多少个线程都没用，该花的时间一点都少不了。

而这跟协程半毛钱关系没有，切线程解决不了的事情，协程也解决不了。

协程与线程

协程我们讲了 3 期，Kotlin 协程和线程是无法脱离开讲的。

别的语言我不说，在 Kotlin 里，协程就是基于线程来实现的一种更上层的工具 API，类似于 Java 自带的 Executor 系列 API 或者 Android 的 Handler 系列 API。

只不过呢，协程它不仅提供了方便的 API，在设计思想上是一个**基于线程的上层框架**，你可以理解为新造了一些概念用来帮助你更好地使用这些 API，仅此而已。

就像 ReactiveX 一样，为了让你更好地使用各种操作符 API，新造了 Observable 等概念。

说到这里，Kotlin 协程的三大疑问：协程是什么、挂起是什么、挂起的非阻塞式是怎么回事，就已经全部讲完了。非常简单：

- 协程就是切线程；
- 挂起就是可以自动切回来的切线程；
- 挂起的非阻塞式指的是它能用看起来阻塞的代码写出非阻塞的操作，就这么简单。

当然了，这几句是总结，它们背后的原理你是一定要掌握住的。如果忘了，再去把之前的视频和文章看一遍就好。

视频中还纠正了官方文档里面的一个错误，这里就不再重复了，最后想表达一点：

Kotlin 协程并没有脱离 Kotlin 或者 JVM 创造新的东西，它只是将多线程的开发变得更简单了，可以说是因为 Kotlin 的诞生而顺其自然出现的东西，从语法上看它很神奇，但从原理上讲，它并不是魔术。

希望通过协程系列的讲解能帮助读者上手 Kotlin 协程，不再觉得害怕不敢上手，欢迎继续关注「码上开学」的后续文章，期待和你共同进步。

练习题

使用协程实现一个网络请求：

- 等待时显示 Loading；
- 请求成功或者出错让 Loading 消失；
- 请求失败需要提示用户请求失败了;
- 让你的协程写法上看上去像单线程。

11. 资源混淆是如何影响到 Kotlin 协程的

导言

随着 kotlin 的使用，协程也慢慢在我们工程中被开始被使用起来，但在我们工程中却遇到了一个问题，经过资源混淆处理之后的 apk 包，协程却不如期工作。那么两者到底有什么关联呢，资源混淆又是如何影响到协程的使用的，通过阅读本篇你会马上知晓。

本篇会从如下几个方面讲述这个问题

问题定义->问题分析->问题解决

问题定义

看下面这段 demo 代码：

```
package com.example.coroutinenotworkdemo

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.provider.Settings
import android.util.Log
import android.widget.Toast
import android.widget.Toast.LENGTH_SHORT
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.*
import kotlin.coroutines.CoroutineContext
import kotlin.coroutines.resume
import kotlin.coroutines.suspendCoroutine
import kotlin.system.measureTimeMillis

class MainActivity : AppCompatActivity(), CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = Job()
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        clickid.setOnClickListener {
            GlobalScope.async {
                Log.i("pisa","start call async")
                val cost=measureTimeMillis {
                    val result=demoSupendFun()
                    Log.i("pisa","get result=$result")
                    //下面经过资源混淆之后，withContext 里面的块没得到执行。。
                    withContext(Dispatchers.Main){
                        textview.text=result
                    }
                }
                Log.i("pisa","cost=$cost")
                0
            }
            Toast.makeText(this,"click result",LENGTH_SHORT)
        }

    }

    suspend fun demoSupendFun(): String {
        return suspendCoroutine {
            //模拟一个异步请求，然后回调，得到结果
            async {
                delay(1000)
                it.resume("get result")
            }
        }
    }
}
```

复制代码

我们发现经过资源混淆之后，下面这段代码中，`textview.text=result` 始终没有得到执行。

```
withContext(Dispatchers.Main){
    textview.text=result
}
```

复制代码

那么这是为什么呢？

问题分析

既然跟资源混淆有关，那么我们看看经过资源混淆之后的 apk 和之前的 apk 到底又哪些改变。 **资源混淆**用的是之前微信开源的的[andResguard][1],简单来说，资源混淆包括如下几个步骤：

1. 解压缩 apk
2. 混淆算法开始混淆 res 文件，并改下 resources.arsc 文件
3. 用 7zip 重压缩 apk，重签名

看起来，1 和 2 对于影响到协程使用可能性很低，那么 3 呢，在对比前后 apk 过程中我们马上发现混淆前后的 apk 的 METF-INF 文件相差比较大，混淆后只保留了 SF，MF，RSA 文件，而混淆前的 apk 的 METF-INF 文件中包含了一些 kotlin_module 信息以及 services 文件夹，那么会不会和这些文件的丢失有关呢。

怎么验证呢。很简单，gradle 里面配置 packageOptions 主动移除 META-INF 文件夹下的 kotlin_module 文件和 services 文件夹，然后 debug 调试一下发现问题复现。那么肯定和这里有关啦。

现在先不急着马上解决它，让我们看看为啥这几个文件的丢失就会导致上面那段协程代码工作不正常呢。既然有 demo，那我们单步调试进去看看吧。

上面例子中调用了 `async` 函数，通过源码可以知道，如果 start 参数是用的默认的情况下，那么最后都会走到 `startCoroutineCancellable` 函数，而这个函数内部会调用 runSafely，**内部所有的异常都会被这个函数 catch 住**，所以业务层没抛 crash，直接把这个问题隐藏了，也给快速定位问题加大了难度。

 既然用 demo 复现了这个问题，那么单步调试一下，看看 `withContext` 里面到底挂在了哪里?最终调试发现，果然这里 `runSafely` 里面 `catch` 住了一个 `exception`，异常信息如下： `Module with the Main dispatcher is missing.Add dependency providing the Main dispatcher, e.g. 'kotlinx-coroutines-android` 所以上面 `withContext` 里面的代码就没有执行到了。

那么这里的 MainDispatcher 是什么呢？原来是在调用 withContext 来切换线程的时候，会用到类 `MainCoroutineDispatcher`。这个类是个抽象类，会经过 `MainDispatcherFactory` 工厂来创建具体的 dispatcher，在 Android 上是

AndroidDispatcherFactory 来负责创建，MainDispatcherFactory 这个类是通过自定义的 ServiceLoader 加载进来的,在 kotlin 中定义了一个 FastServiceLoader，这个类与 java 的 ServiceLoader 最大的区别是跳过了 jar 的校验，可以直接从 jar 包中加载某一个类的信息，如果用常规的 ServiceLoader 是需要读取整个 jar 包之后，在定位到对应的 class 文件信息，加载进来，这整个过程是一个非常耗时的操作，可能导致 android 设备发生 ANR 的现象。

看看 FastServiceLoader 是如何加载 AndroidDispatcherFactory 的，如下图所示：

 看到这个类瞬间明白了，kotlin 在编译的时候，会在 META-INF 文件夹下生成一个 services 的文件夹信息，该文件夹下面放一些支持类的信息，那么具体在放了哪些类呢，在源码当中有一个 pro 文件可以说明一切。  这样在调用相关类的时候会优先先用 FastServiceLoader 加载该类。一旦加载不到，就会构造一个 MissingMainCoroutineDispatcher，并调用 missing 方法抛出异常。 

问题解决

经过上述问题分析之后，其实解决方案就非常简单了。**修改资源混淆重打包的流程，在重签名的时候保留 META-INF 的 servcies 文件夹信息即可**

回顾总结

再来回顾一下问题的解决过程，虽然最终解决的方案比较简单，但有两个点需要我们特别关注一下

1. 协程当中 async 内部有 try catch 机制，所以任何异常都会被内部 catch 住，而这个在我们开发当中很容易导致一些问题没有及时发现
2. 在遇到一些奇怪的问题的时候，小而简单的 demo 外加源码阅读是必要的，这样方便我们快速能够追查到问题原因所在。

作者：腾讯音乐技术团队

链接：https://juejin.im/post/5cf0dc58f265da1b9612ea89

12.破解 Kotlin 协程

假定你对协程（Coroutine）一点儿都不了解，通过阅读本文看看是否能让你明白协程是怎么一回事。

1. 引子

我之前写过一些协程的文章，很久以前了。那会儿还是很痛苦的，毕竟 kotlinx.coroutines 这样强大的框架还在襁褓当中，于是乎我写的几篇协程的文章几乎就是在告诉大家如何写这样一个框架——那种感觉简直糟糕透了，因为没有几个人会有这样的需求。

这次准备从协程用户（也就是程序员你我他啦）的角度来写一下，希望对大家能有帮助。

2. 需求确认

在开始讲解协程之前，我们需要先确认几件事儿：

1. 你用过线程对吧？
2. 你写过回调对吧？
3. 你用过 RxJava 类似的框架吗？

看下你的答案：

- 如果上面的问题的回答都是 “Yes”，那么太好了，这篇文章非常适合你，因为你已经意识到回调有多么可怕，并且找到了解决方案；

- 如果前两个是 “Yes”，没问题，至少你已经开始用回调了，你是协程潜在的用户；
- 如果只有第一个是 “Yes”，那么，可能你刚刚开始学习线程，那你还是先打好基础再来吧~

3. 一个常规例子

我们通过 Retrofit 发送一个网络请求，其中接口如下：

```
interface GitHubServiceApi {
    @GET("users/{login}")
    fun getUser(@Path("login") login: String): Call<User>
}

data class User(val id: String, val name: String, val url: String)
```

Retrofit 初始化如下：

```
val gitHubServiceApi by lazy {
    val retrofit = retrofit2.Retrofit.Builder()
            .baseUrl("https://api.github.com")
            .addConverterFactory(GsonConverterFactory.create())
            .build()

    retrofit.create(GitHubServiceApi::class.java)
}
```

那么我们请求网络时：

```
gitHubServiceApi.getUser("bennyhuo").enqueue(object : Callback<User> {
    override fun onFailure(call: Call<User>, t: Throwable) {
        handler.post { showError(t) }
    }

    override fun onResponse(call: Call<User>, response: Response<User>) {
        handler.post { response.body()?.let(::showUser) ?: showError(NullPointerException()) }
```

```
    }
})
```

请求结果回来之后，我们切换线程到 UI 线程来展示结果。这类代码大量存在于我们的逻辑当中，它有什么问题呢？

- 通过 Lambda 表达式，我们让线程切换变得不是那么明显，但它仍然存在，一旦开发者出现遗漏，这里就会出现问题
- 回调嵌套了两层，看上去倒也没什么，但真实的开发环境中逻辑一定比这个复杂的多，例如登录失败的重试
- 重复或者分散的异常处理逻辑，在请求失败时我们调用了一次 `showError`，在数据读取失败时我们又调用了一次，真实的开发环境中可能会有更多的重复

Kotlin 本身的语法已经让这段代码看上去好很多了，如果用 Java 写的话，你的直觉都会告诉你：你在写 Bug。

> 如果你不是 Android 开发者，那么你可能不知道 handler 是什么东西，没关系，你可以替换为 `SwingUtilities.invokeLater{ ... }`(Java Swing)，或者 `setTimeout({ ... }, 0)`(Js) 等等。

4. 改造成协程

你当然可以改造成 RxJava 的风格，但 RxJava 比协程抽象多了，因为除非你熟练使用那些 operator，不然你根本不知道它在干嘛(试想一下 `retryWhen`)。

协程就不一样了，毕竟编译器加持，它可以很简洁的表达出代码的逻辑，不要想它背后的实现逻辑，它的运行结果就是你直觉告诉你的那样。

对于 Retrofit，改造成协程的写法，有两种，分别是通过 CallAdapter 和 suspend 函数。

4.1 CallAdapter 的方式

我们先来看看 CallAdapter 的方式，这个方式的本质是让接口的方法返回一个协程的 Job：

```
interface GitHubServiceApi {
    @GET("users/{login}")
    fun getUser(@Path("login") login: String): Deferred<User>
}
```

注意 Deferred 是 Job 的子接口。

那么我们需要为 Retrofit 添加对 `Deferred` 的支持，这需要用到开源库：

```
implementation 'com.jakewharton.retrofit:retrofit2-kotlin-coroutines-adapter:0.9.2'
```

构造 Retrofit 实例时添加：

```
val gitHubServiceApi by lazy {
    val retrofit = retrofit2.Retrofit.Builder()
            .baseUrl("https://api.github.com")
            .addConverterFactory(GsonConverterFactory.create())
            //添加对 Deferred 的支持
            .addCallAdapterFactory(CoroutineCallAdapterFactory())
            .build()

    retrofit.create(GitHubServiceApi::class.java)
}
```

那么这时候我们发起请求就可以这么写了：

```
GlobalScope.launch(Dispatchers.Main) {
    try {
        showUser(gitHubServiceApi.getUser("bennyhuo").await())
    } catch (e: Exception) {
        showError(e)
    }
}
```

> 说明：`Dispatchers.Main` 在不同的平台上的实现不同，如果在 Android 上为 `HandlerDispatcher`，在 Java Swing 上为 `SwingDispatcher` 等等。

首先我们通过 `launch` 启动了一个协程，这类似于我们启动一个线程，`launch` 的参数有三个，依次为协程上下文、协程启动模式、协程体：

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext, // 上下文
    start: CoroutineStart = CoroutineStart.DEFAULT,  // 启动模式
    block: suspend CoroutineScope.() -> Unit // 协程体
): Job
```

启动模式不是一个很复杂的概念，不过我们暂且不管，默认直接允许调度执行。

上下文可以有很多作用，包括携带参数，拦截协程执行等等，多数情况下我们不需要自己去实现上下文，只需要使用现成的就好。上下文有一个重要的作用就是线程切换，`Dispatchers.Main` 就是一个官方提供的上下文，它可以确保 `launch` 启动的协程体运行在 UI 线程当中（除非你自己在 `launch` 的协程体内部进行线程切换、或者启动运行在其他有线程切换能力的上下文的协程）。

换句话说，在例子当中整个 `launch` 内部**你看到的代码**都是运行在 UI 线程的，尽管 `getUser` 在执行的时候确实切换了线程，但返回结果的时候会再次切回来。这看上去有些费解，因为直觉告诉我们，`getUser` 返回了一个 `Deferred` 类型，它的 `await` 方法会返回一个 `User` 对象，意味着 `await` 需要等待请求结果返回才可以继续执行，那么 `await` 不会阻塞 UI 线程吗？

答案是：不会。当然不会，不然那 `Deferred` 与 `Future` 又有什么区别呢？这里 `await` 就很可疑了，因为它实际上是一个 suspend 函数，这个函数只能在协程体或者其他 suspend 函数内部被调用，它就像是回调的语法糖一样，它通过一个叫 `Continuation` 的接口的实例来返回结果：

```
1@SinceKotlin("1.3")
2public interface Continuation<in T> {
3    public val context: CoroutineContext
4    public fun resumeWith(result: Result<T>)
5}
```

1.3 的源码其实并不是很直接，尽管我们可以再看下 `Result` 的源码，但我不想这么做。更容易理解的是之前版本的源码：

```
1@SinceKotlin("1.1")
2public interface Continuation<in T> {
3    public val context: CoroutineContext
4    public fun resume(value: T)
5    public fun resumeWithException(exception: Throwable)
6}
```

相信大家一下就能明白，这其实就是个回调嘛。如果还不明白，那就对比下 Retrofit 的 `Callback`：

```
1public interface Callback<T> {
2  void onResponse(Call<T> call, Response<T> response);
3  void onFailure(Call<T> call, Throwable t);
```

```
4}
```

有结果正常返回的时候，`Continuation` 调用 `resume` 返回结果，否则调用 `resumeWithException` 来抛出异常，简直与 `Callback` 一模一样。

所以这时候你应该明白，这段代码的执行流程本质上是一个异步回调：

```
1GlobalScope.launch(Dispatchers.Main) {
2    try {
3        //showUser 在 await 的 Continuation 的回调函数调用后执行
4        showUser(gitHubServiceApi.getUser("bennyhuo").await())
5    } catch (e: Exception) {
6        showError(e)
7    }
8}
```

而代码之所以可以看起来是同步的，那就是编译器的黑魔法了，你当然也可以叫它“语法糖”。

这时候也许大家还是有问题：我并没有看到 `Continuation` 啊，没错，这正是我们前面说的编译器黑魔法了，在 Java 虚拟机上，`await` 这个方法的签名其实并不像我们看到的那样：

```
1public suspend fun await(): T
```

它真实的签名其实是：

```
1kotlinx/coroutines/Deferred.await (Lkotlin/coroutines/Continuation;)Lj
ava/lang/Object;
```

即接收一个 `Continuation` 实例，返回 `Object` 的这么个函数，所以前面的代码我们可以大致理解为：

```
1//注意以下不是正确的代码，仅供大家理解协程使用
2GlobalScope.launch(Dispatchers.Main) {
3    gitHubServiceApi.getUser("bennyhuo").await(object: Continuation<U
```

```
4ser>{
5            override fun resume(value: User) {
6                showUser(value)
7            }
8            override fun resumeWithException(exception: Throwable){
9                showError(exception)
1            }
0    })
1}
1
```

而在 `await` 当中，大致就是：

```
1//注意以下并不是真实的实现，仅供大家理解协程使用
2fun await(continuation: Continuation<User>): Any {
3    ... // 切到非 UI 线程中执行，等待结果返回
4    try {
5        val user = ...
6        handler.post{ continuation.resume(user) }
7    } catch(e: Exception) {
8        handler.post{ continuation.resumeWithException(e) }
9    }
10}
```

这样的回调大家一看就能明白。讲了这么多，请大家记住一点：从执行机制上来讲，协程跟回调没有什么本质的区别。

4.2 suspend 函数的方式

suspend 函数是 Kotlin 编译器对协程支持的唯一的黑魔法（表面上的，还有其他的我们后面讲原理的时候再说）了，我们前面已经通过 `Deferred` 的 `await` 方法对它有了个大概的了解，我们再来看看 Retrofit 当中它还可以怎么用。

> Retrofit 当前的 release 版本是 2.5.0，还不支持 suspend 函数。因此想要尝试下面的代码，需要最新的 Retrofit 源码的支持；

当然，也许你看到这篇文章的时候，Retrofit 的新版本已经支持这一项特性了呢。

首先我们修改接口方法：

```
@GET("users/{login}")
suspend fun getUser(@Path("login") login: String): User
```

这种情况 Retrofit 会根据接口方法的声明来构造 `Continuation`，并且在内部封装了 `Call` 的异步请求（使用 enqueue），进而得到 `User` 实例，具体原理后面我们有机会再介绍。使用方法如下：

```
GlobalScope.launch {
    try {
        showUser(gitHubServiceApi.getUser("bennyhuo"))
    } catch (e: Exception) {
        showError(e)
    }
}
```

它的执行流程与 `Deferred.await` 类似，我们就不再详细分析了。

5. 协程到底是什么

好，坚持读到这里的朋友们，你们一定是异步代码的“受害者”，你们肯定遇到过“回调地狱”，它让你的代码可读性急剧降低；也写过大量复杂的异步逻辑处理、异常处理，这让你的代码重复逻辑增加；因为回调的存在，还得经常处理线程切换，这似乎并不是一件难事，但随着代码体量的增加，它会让你抓狂，线上上报的异常因线程使用不当导致的可不在少数。

而**协程**可以帮你优雅的处理掉这些。

协程本身是一个脱离语言实现的概念，我们“很严谨”（哈哈）的给出维基百科的定义：

> Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

简单来说就是，协程是一种非抢占式或者说协作式的计算机程序并发调度的实现，程序可以主动挂起或者恢复执行。这里还是需要有点儿操作系统的知识的，我们在 Java 虚拟机上所认识到的线程大多数的实现是映射到内核的线程的，也就是说线程当中的代码逻辑在线程抢到 CPU 的时间片的时候才可以执行，否则就得歇着，当然这对于我们开发者来说是透明的；而经常听到所谓的协程更轻量的意思是，协程并不会映射成内核线程或者其他这么重的资源，它的调度在用户态就可以搞定，任务之间的调度并非抢占式，而是协作式的。

> 关于并发和并行：正因为 CPU 时间片足够小，因此即便一个单核的 CPU，也可以给我们营造多任务同时运行的假象，这就是所谓的“并发”。并行才是真正的同时运行。并发的话，更像是 Magic。

如果大家熟悉 Java 虚拟机的话，就想象一下 Thread 这个类到底是什么吧，为什么它的 run 方法会运行在另一个线程当中呢？谁负责执行这段代码的呢？

显然，咋一看，Thread 其实是一个对象而已，run 方法里面包含了要执行的代码——仅此而已。协程也是如此，如果你只是看标准库的 API，那么就太抽象了，但我们开篇交代了，学习协程不要上来去接触标准库，kotlinx.coroutines 框架才是我们用户应该关心的，而这个框架里面对应于 Thread 的概念就是 Job 了，大家可以看下它的定义：

```
public interface Job : CoroutineContext.Element {
    ...
    public val isActive: Boolean
    public val isCompleted: Boolean
    public val isCancelled: Boolean

    public fun start(): Boolean
    public fun cancel(cause: CancellationException? = null)
    public suspend fun join()
    ...
}
```

我们再来看看 Thread 的定义：

```
public class Thread implements Runnable {
    ...
    public final native boolean isAlive();
    public synchronized void start() { ... }
    @Deprecated
    public final void stop() { ... }
    public final void join() throws InterruptedException  { ... }
    ...
}
```

这里我们非常贴心的省略了一些注释和不太相关的接口。我们发现，Thread 与 Job 基本上功能一致，它们都承载了一段代码逻辑（前者通过 run 方法，后者通过构造协程用到的 Lambda 或者函数），也都包含了这段代码的运行状态。

而真正调度时二者才有了本质的差异，具体怎么调度，我们只需要知道调度结果就能很好的使用它们了。

6. 小结

我们先通过例子来引入，从大家最熟悉的代码到协程的例子开始，演化到协程的写法，让大家首先能从感性上对协程有个认识，最后我们给出了协程的定义，也告诉大家协程究竟能做什么。

这篇文章没有追求什么内部原理，只是企图让大家对协程怎么用有个第一印象。如果大家仍然感觉到迷惑，不怕，后面我将再用几篇文章从例子入手来带着大家分析协程的运行，而原理的分析，会放到大家能够熟练掌握协程之后再来探讨。

第六章 Flutter 相关

1. Dart 当中的 「..」表示什么意思？

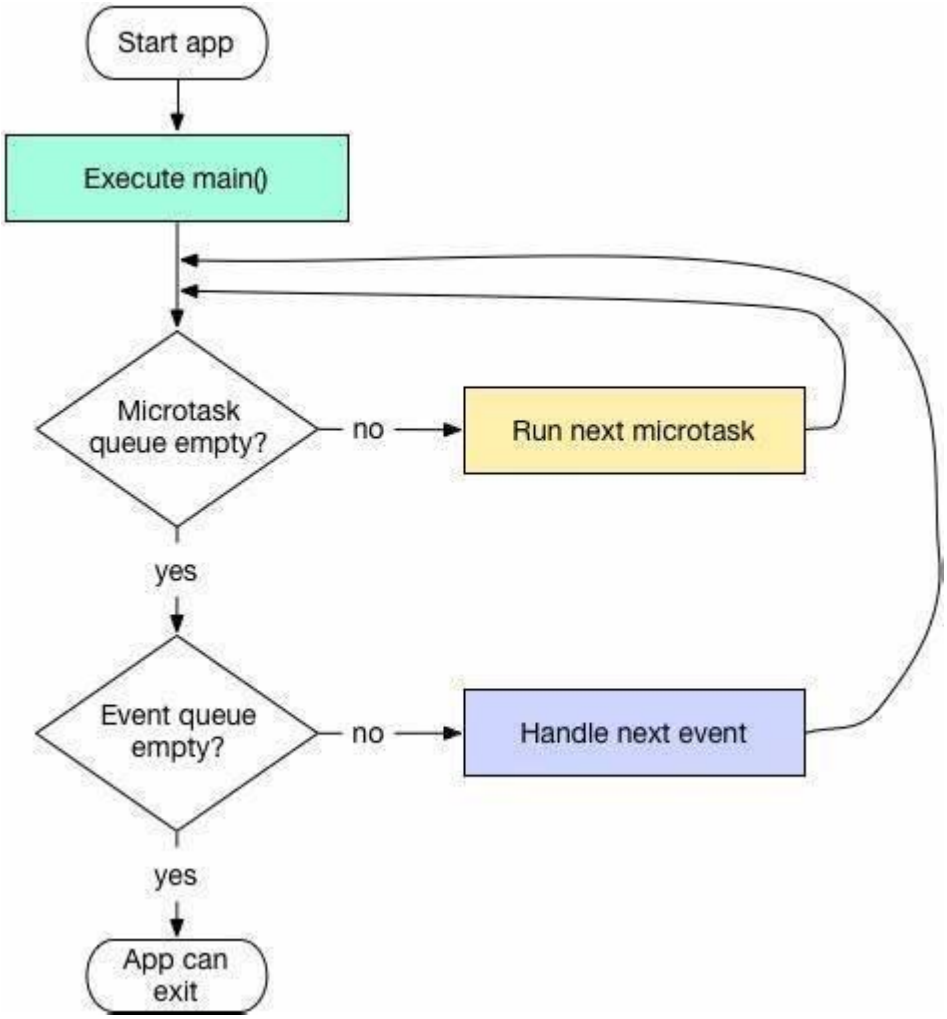
Dart 当中的 「..」意思是 「级联操作符」，为了方便配置而使用。「..」和「.」不同的是 调用「..」后返回的相当于是 this，而「.」返回的则是该方法返回的值 。

2. Dart 的作用域

Dart 没有 「public」「private」等关键字，默认就是公开的，私有变量使用 下划线 _开头。

3. Dart 是不是单线程模型？是如何运行的？

Dart 是单线程模型，运行的的流程如下图。



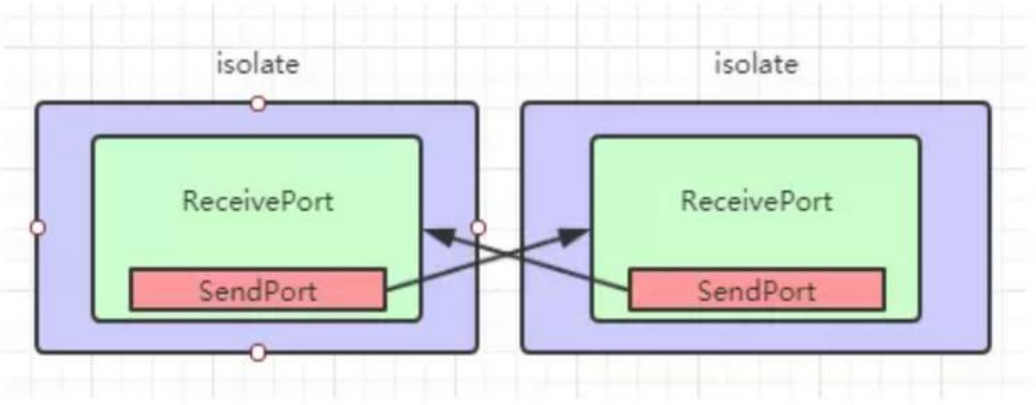
简单来说，Dart 在单线程中是以小心循环机制来运行的，包含两个任务队列，，

一个是微任务队列　microtask queue，另一个叫做事件队列　event queue。

当 Flutter 应用启动后，消息循环机制便启动了。首先会按照先进先出的顺序逐个执行微任务队列中的任务，当所有微任务队列执行完后便开始执行事件队列中的任务，事件任务执行完毕后再去执行微任务，如此循环往复，生生不息。

4. Dart 是如何实现多任务并行的？

前面说过， Dart 是单线程的，不存在多线程，那如何进行多任务并行的呢？其实，Dart 的多线程和前端的多线程有很多的相似之处。Flutter 的多线程主要依赖 Dart 的并发编程、异步和事件驱动机制。



isolate交互.png

简单的说 ,在 Dart 中 ,一个 Isolate 对象其实就是一个 isolate 执行环境的引用，一般来说我们都是通过当前的 isolate 去控制其他的 isolate 完成彼此之间的交互，而当我们想要创建一个新的 Isolate 可以使用 Isolate.spawn 方法获取返回的一个新的 isolate 对象，两个 isolate 之间使用 SendPort 相互发送消息，而 isolate 中也存在了一个与之对应的 ReceivePort 接受消息用来处理，但是我们需要注意的是，ReceivePort 和 SendPort 在每个 isolate 都有一对，只有同一个 isolate 中的 ReceivePort 才能接受到当前类的 SendPort 发送的消息并且处理。

5. 说一下 Dart 异步编程中的 Future 关键字？

前面说过，Dart 在单线程中是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。

在 Java 并发编程开发中，经常会使用 Future 来处理异步或者延迟处理任务等操作。而在 Dart 中，执行一个异步任务同样也可以使用 Future 来处理。在 Dart 的每一个 Isolate 当中，执行的优先级为 ： Main > MicroTask > EventQueue。

6. 说一下 Dart 异步编程中的 Stream 数据流？

在 Dart 中，Stream 和 Future 一样，都是用来处理异步编程的工具。它们的区别在于，Stream 可以接收多个异步结果，而 Future 只有一个。

Stream 的创建可以使用 Stream.fromFuture，也可以使用 StreamController 来创建和控制。还有一个注意点是：普通的 Stream 只可以有一个订阅者，如果想要多订阅的话，要使用 asBroadcastStream()。

7. Stream 有哪两种订阅模式？分别是怎么调用的？

Stream 有两种订阅模式：单订阅(single) 和 多订阅（broadcast）。单订阅就是只能有一个订阅者，而广播是可以有多个订阅者。这就有点类似于消息服务（Message Service）的处理模式。单订阅类似于点对点，在订阅者出现之前会持有数据，在订阅者出现之后就才转交给它。而广播类似于发布订阅模式，可以

同时有多个订阅者，当有数据时就会传递给所有的订阅者，而不管当前是否已有订阅者存在。

Stream 默认处于单订阅模式，所以同一个 stream 上的 listen 和其它大多数方法只能调用一次，调用第二次就会报错。但 Stream 可以通过 transform() 方法（返回另一个 Stream）进行连续调用。通过 Stream.asBroadcastStream() 可以将一个单订阅模式的 Stream 转换成一个多订阅模式的 Stream，isBroadcast 属性可以判断当前 Stream 所处的模式。

8. await for 如何使用？

await for 是不断获取 stream 流中的数据，然后执行循环体中的操作。它一般用在直到 stream 什么时候完成，并且必须等待传递完成之后才能使用，不然就会一直阻塞。

```
Stream<String> stream = new Stream<String>.fromIterable(['不开心', '面试', '没', '过']);
main() async{
    await for(String s in stream){
    print(s);
  }
}
```

9. 说一下 mixin 机制？

mixin 是 Dart 2.1 加入的特性，以前版本通常使用 abstract class 代替。简单来说，mixin 是为了解决继承方面的问题而引入的机制，Dart 为了支持多重继承，引入了 mixin 关键字，它最大的特殊处在于： mixin 定义的类不能有构造

方法，这样可以避免继承多个类而产生的父类构造方法冲突。

mixins 的对象是类，mixins 绝不是继承，也不是接口，而是一种全新的特性，可以 mixins 多个类，mixins 的使用需要满足一定条件。

Flutter

1. 请简单介绍下 Flutter 框架，以及它的优缺点？

Flutter 是 Google 推出的一套开源跨平台 UI 框架，可以快速地在 Android、iOS 和 Web 平台上构建高质量的原生用户界面。同时，Flutter 还是 Google 新研发的 Fuchsia 操作系统的默认开发套件。在全世界，Flutter 正在被越来越多的开发者和组织使用，并且 Flutter 是完全免费、开源的。Flutter 采用现代响应式框架构建，其中心思想是使用组件来构建应用的 UI。当组件的状态发生改变时，组件会重构它的描述，Flutter 会对比之前的描述，以确定底层渲染树从当前状态转换到下一个状态所需要的最小更改。

优点

- 热重载（Hot Reload），利用 Android Studio 直接一个 ctrl+s 就可以保存并重载，模拟器立马就可以看见效果，相比原生冗长的编译过程强很多；
- 一切皆为 Widget 的理念，对于 Flutter 来说，手机应用里的所有东西都是 Widget，通过可组合的空间集合、丰富的动画库以及分层课扩展的架构实现了富有感染力的灵活界面设计；
- 借助可移植的 GPU 加速的渲染引擎以及高性能本地代码运行时以达到跨平

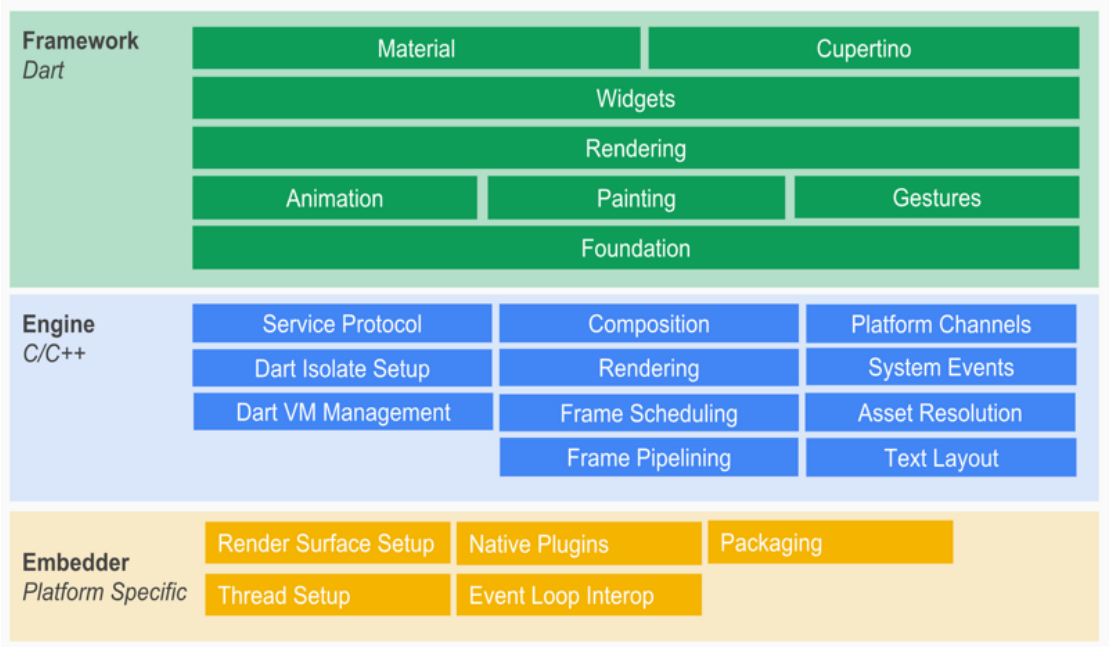
台设备的高质量用户体验。 简单来说就是：最终结果就是利用 Flutter 构建的应用在运行效率上会和原生应用差不多。

缺点

- 不支持热更新；
- 三方库有限，需要自己造轮子；
- Dart 语言编写，增加了学习难度，并且学习了 Dart 之后无其他用处，相比 JS 和 Java 来说。

2. 介绍下 Flutter 的理念架构

其实也就是下面这张图。



由上图可知，Flutter 框架自下而上分为 Embedder、Engine 和 Framework 三层。其中，Embedder 是操作系统适配层，实现了渲染 Surface 设置，线程设置，以及平台插件等平台相关特性的适配；Engine 层负责图形绘制、文字排版和提供 Dart 运行时，Engine 层具有独立虚拟机，正是由于它的存在，Flutter 程序才能运行在不同的平台上，实现跨平台运行；Framework 层则是使用 Dart 编写的一套基础视图库，包含了动画、图形绘制和手势识别等功能，是使用频率最高的一层。

3. 介绍下 FFlutter 的 FrameWork 层和 Engine 层，以及它们的作用

Flutter 的 FrameWork 层是用 Drat 编写的框架（SDK），它实现了一套基础库，包含 Material（Android 风格 UI）和 Cupertino（iOS 风格）的 UI 界面，下面是通用的 Widgets（组件），之后是一些动画、绘制、渲染、手势库等。这个纯 Dart 实现的 SDK 被封装为了一个叫作 dart:ui 的 Dart 库。我们在使用 Flutter 写 App 的时候，直接导入这个库即可使用组件等功能。

Flutter 的 Engine 层是 Skia 2D 的绘图引擎库，其前身是一个向量绘图软件，Chrome 和 Android 均采用 Skia 作为绘图引擎。Skia 提供了非常友好的 API，并且在图形转换、文字渲染、位图渲染方面都提供了友好、高效的表现。Skia 是跨平台的，所以可以被嵌入到 Flutter 的 iOS SDK 中，而不用去研究 iOS 闭源的 Core Graphics / Core Animation。Android 自带了 Skia，所以 Flutter Android SDK 要比 iOS SDK 小很多。

4. 介绍下 Widget、State、Context 概念

- **Widget**：在 Flutter 中，几乎所有东西都是 Widget。将一个 Widget 想象为一个可视化的组件（或与应用可视化方面交互的组件），当你需要构建与布局直接或间接相关的任何内容时，你正在使用 Widget。
- **Widget 树**：Widget 以树结构进行组织。包含其他 Widget 的 widget 被称为父 Widget(或 widget 容器)。包含在父 widget 中的 widget 被称为子 Widget。

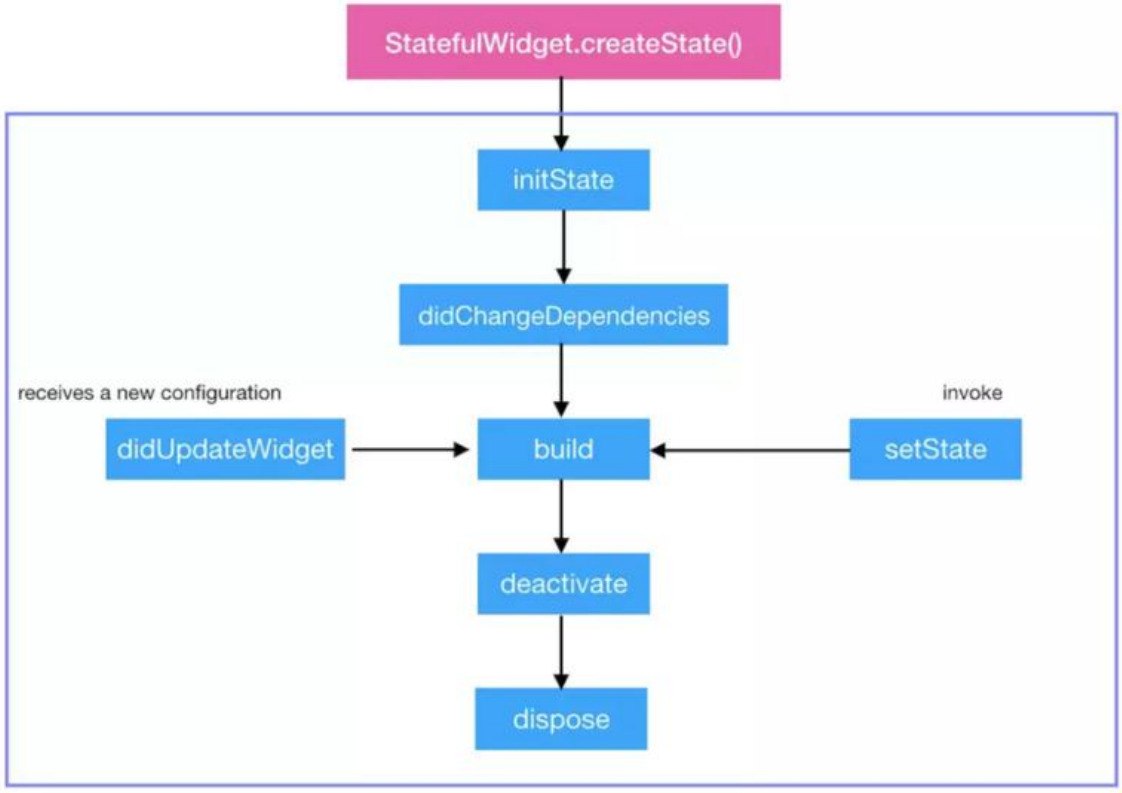
- **Context**：仅仅是已创建的所有 Widget 树结构中的某个 Widget 的位置引用。简而言之，将 context 作为 widget 树的一部分，其中 context 所对应的 widget 被添加到此树中。一个 context 只从属于一个 widget，它和 widget 一样是链接在一起的，并且会形成一个 context 树。
- **State**：定义了 StatefulWidget 实例的行为，它包含了用于”交互/干预“Widget 信息的行为和布局。应用于 State 的任何更改都会强制重建 Widget。

5. 简述 Widget 的 StatelessWidget 和 StatefulWidget 两种状态组件类

- **StatelessWidget**: 一旦创建就不关心任何变化，在下次构建之前都不会改变。它们除了依赖于自身的配置信息（在父节点构建时提供）外不再依赖于任何其他信息。比如典型的 Text、Row、Column、Container 等，都是 StatelessWidget。它的生命周期相当简单：初始化、通过 build()渲染。
- **StatefulWidget**: 在生命周期内，该类 Widget 所持有的数据可能会发生变化，这样的数据被称为 State，这些拥有动态内部数据的 Widget 被称为 StatefulWidget。比如复选框、Button 等。State 会与 Context 相关联，并且此关联是永久性的，State 对象将永远不会改变其 Context，即使可以在树结构周围移动，也仍将与该 context 相关联。当 state 与 context 关联时，state 被视为已挂载。StatefulWidget 由两部分组成，在初始化时必须要在 createState()时初始化一个与之相关的 State 对象。

6. StatefulWidget 的生命周期

Flutter 的 Widget 分为 StatelessWidget 和 StatefulWidget 两种。其中，StatelessWidget 是无状态的，StatefulWidget 是有状态的，因此实际使用时，更多的是 StatefulWidget。StatefulWidget 的生命周期如下图。



- **initState()**：Widget 初始化当前 State，在当前方法中是不能获取到 Context 的，如想获取，可以试试 Future.delayed()
- **didChangeDependencies()**：在 initState() 后调用，State 对象依赖关系发生变化的时候也会调用。

- **deactivate()**：当 State 被暂时从视图树中移除时会调用这个方法，页面切换时也会调用该方法，和 Android 里的 onPause 差不多。
- **dispose()**：Widget 销毁时调用。
- **didUpdateWidget**：Widget 状态发生变化的时候调用。

7. 简述 Widgets、RenderObjects 和 Elements 的关系

首先看一下这几个对象的含义及作用。

- **Widget**：仅用于存储渲染所需要的信息。
- **RenderObject**：负责管理布局、绘制等操作。
- **Element**：才是这颗巨大的控件树上的实体。

Widget 会被 inflate（填充）到 Element，并由 Element 管理底层渲染树。Widget 并不会直接管理状态及渲染,而是通过 State 这个对象来管理状态。Flutter 创建 Element 的可见树，相对于 Widget 来说，是可变的，通常界面开发中，我们不用直接操作 Element,而是由框架层实现内部逻辑。就如一个 UI 视图树中，可能包含有多个 TextWidget(Widget 被使用多次)，但是放在内部视图树的视角，这些 TextWidget 都是填充到一个个独立的 Element 中。Element 会持有 renderObject 和 widget 的实例。记住，Widget 只是一个配置，RenderObject 负责管理布局、绘制等操作。

在第一次创建 Widget 的时候，会对应创建一个 Element， 然后将该元素插入树中。如果之后 Widget 发生了变化，则将其与旧的 Widget 进行比较，并且相应地更新 Element。重要的是，Element 不会被重建，只是更新而已。

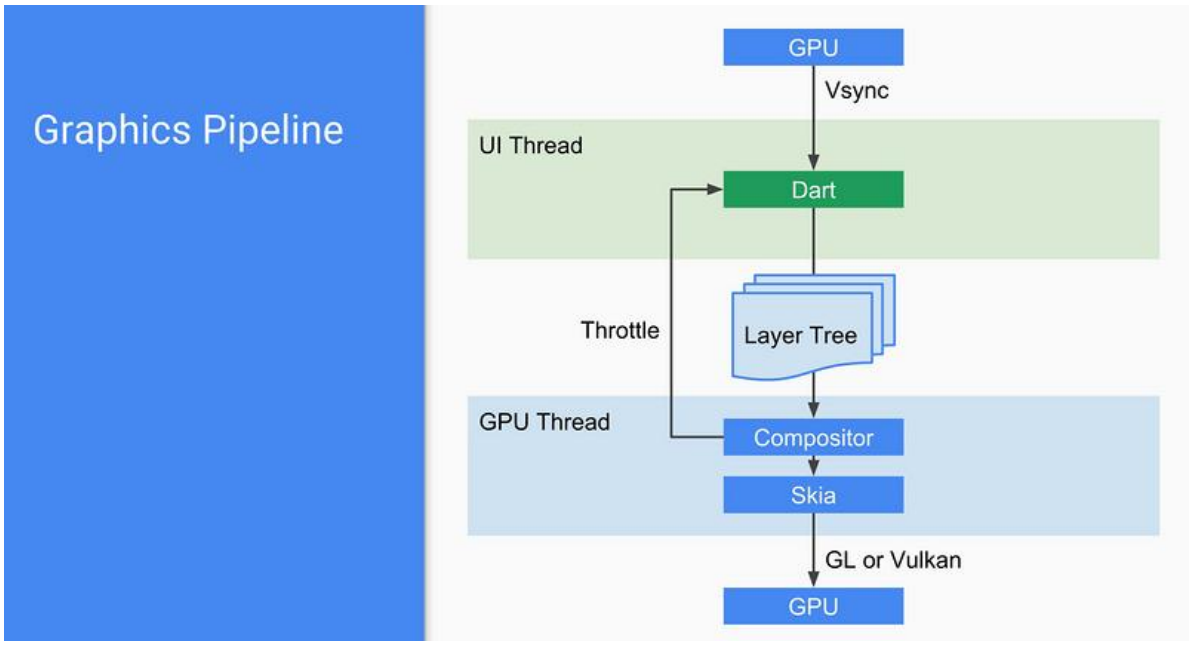
8. 什么是状态管理，你了解哪些状态管理框架？

Flutter 中的状态和前端 React 中的状态概念是一致的。React 框架的核心思想是组件化，应用由组件搭建而成，组件最重要的概念就是状态，状态是一个组件的 UI 数据模型，是组件渲染时的数据依据。

Flutter 的状态可以分为全局状态和局部状态两种。常用的状态管理有 ScopedModel、BLoC、Redux / FishRedux 和 Provider。详细使用情况和差异可以自行了解。

9. 简述 Flutter 的绘制流程

Flutter 的绘制流程如下图所示。



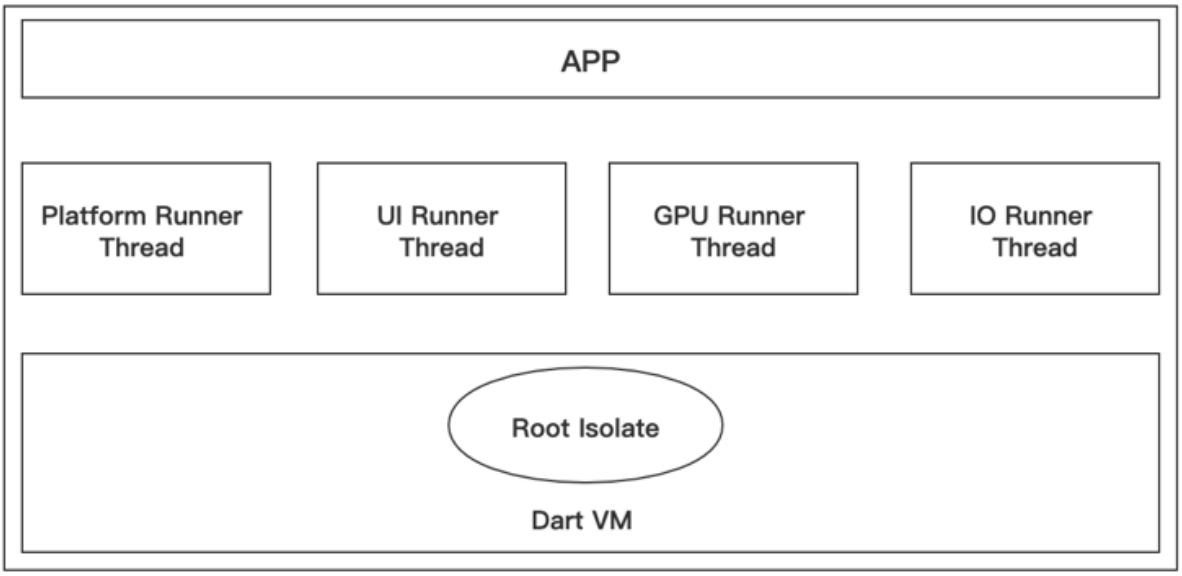
Flutter 只关心向 GPU 提供视图数据，GPU 的 VSync 信号同步到 UI 线程，UI 线程使用 Dart 来构建抽象的视图结构，这份数据结构在 GPU 线程进行图层合成，视图数据提供给 Skia 引擎渲染为 GPU 数据，这些数据通过 OpenGL 或者 Vulkan 提供给 GPU。

10. 简述 Flutter 的线程管理模型

默认情况下，Flutter Engine 层会创建一个 Isolate，并且 Dart 代码默认就运行在这个主 Isolate 上。必要时可以使用 spawnUri 和 spawn 两种方式来创建新的 Isolate，在 Flutter 中，新创建的 Isolate 由 Flutter 进行统一的管理。

事实上，Flutter Engine 自己不创建和管理线程，Flutter Engine 线程的创建和管

理是 Embeder 负责的 ,Embeder 指的是将引擎移植到平台的中间层代码 ,Flutter Engine 层的架构示意图如下图所示。



在 Flutter 的架构中，Embeder 提供四个 Task Runner，分别是 Platform Task Runner、UI Task Runner Thread、GPU Task Runner 和 IO Task Runner，每个 Task Runner 负责不同的任务，Flutter Engine 不在乎 Task Runner 运行在哪个线程，但是它需要线程在整个生命周期里面保持稳定。

11. Flutter 是如何与原生 Android、iOS 进行通信的？

Flutter 通过 PlatformChannel 与原生进行交互，其中 PlatformChannel 分为三种：

BasicMessageChannel ：用于传递字符串和半结构化的信息。

MethodChannel ：用于传递方法调用（ method invocation ）。

EventChannel：用于数据流（ event streams ）的通信。

同时 Platform Channel 并非是线程安全的 ，更多详细可查阅闲鱼技术的《深入理解 Flutter Platform Channel》

12. 简述 Flutter 的热重载

Flutter 的热重载是基于 JIT 编译模式的代码增量同步。由于 JIT 属于动态编译，能够将 Dart 代码编译成生成中间代码，让 Dart VM 在运行时解释执行，因此可以通过动态更新中间代码实现增量同步。

热重载的流程可以分为 5 步，包括：扫描工程改动、增量编译、推送更新、代码合并、Widget 重建。Flutter 在接收到代码变更后，并不会让 App 重新启动执行，而只会触发 Widget 树的重新绘制，因此可以保持改动前的状态，大大缩短了从代码修改到看到修改产生的变化之间所需要的时间。

另一方面，由于涉及到状态的保存与恢复，涉及状态兼容与状态初始化的场景，热重载是无法支持的，如改动前后 Widget 状态无法兼容、全局变量与静态属性的更改、main 方法里的更改、initState 方法里的更改、枚举和泛型的更改等。

可以发现，热重载提高了调试 UI 的效率，非常适合写界面样式这样需要反复查看修改效果的场景。但由于其状态保存的机制所限，热重载本身也有一些无法支持的边界。