Java知识点复习

Java基础篇

第一章 Java语言特点

继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。

多态：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。

封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对 象有哪些属性和行为，并不关注这些行为的细节是什么。

第二章 Java基本结构

基本数据类型

Java中的基本数据类型有8种基本数据结构：

主要分为8种基本类型：4种整型、2种浮点类型、1种字符类型和1种boolean类型。

数据类型	存储需求	取值范围	
int	4字节	超过20亿	
short	2字节	-2^16—2^16-1	
long	8字节		
byte	1字节	-128——127	
float	4字节	6.7位有效	
double	8字节	15位有效	
char	2字节	能存储中文	
boolean	1字节	true和false	

注意：Java没有无符号形式类型

```
        long a=1;
         //这里需要f后缀
        float f=10.0f;
        Long t=10L;
        char c='中';
```

常量

Java使用final修饰常量，const是Java的一个保留字段

字符串

是一个不可改变的类，String类内部是一个char数组和hashcode值。

在修改字符串时，将会new出一个新的字符串，然后将变量指向这块地址。

不可变字符串有一个优点：编译器可以让字符串共享，如果变量复制，原始字符串与复制字符串共享相同的字符。

检测字符串相等，使用equals()方法，==表示字符串是否在同一个位置上。

substring和+操作产生的结果不是共享的。

String .join()可以使用界定符，拼接多个字符串，这是一个静态方法。

字符串有length() 方法，数组有length属性

构建字符串

StringBuilder类是一个final修饰的类是线程不安全的，主要用于字符串拼接，在单线程下使用，高效。

StringBuffer类是一个final修饰的类是线程安全的，也是用于字符串拼接，加入synchronized关键字，效率低。

StringBuffer源码

```
public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    //内部是一个char数组
    private transient char[] toStringCache;
    static final long serialVersionUID = 3388685877147921107L;
     //初始值是16
     public StringBuffer() {
        super(16);
    }
    public StringBuffer(int capacity) {
        super(capacity);
    }
```

```
    // 在原来的字符串基础上加16
     public StringBuffer(String str) {
        super(str.length() + 16);
        append(str);
    }
 }

    //反转方法，这里是在原来的基础上进行反转
     @Override
    public synchronized StringBuffer reverse() {
        toStringCache = null;
        super.reverse();
        return this;
    }

//这个是new出一个新的字符串
   @Override
    public synchronized String toString() {
        if (toStringCache == null) {
            toStringCache = Arrays.copyOfRange(value, 0, count);
        }
        return new String(toStringCache, true);
    }

//substring方法也是new出一个新的方法
    @Override
    public synchronized String substring(int start, int end) {
        return super.substring(start, end);
    }
```

控制流程

switch语句，用于条件判断，case的类型有

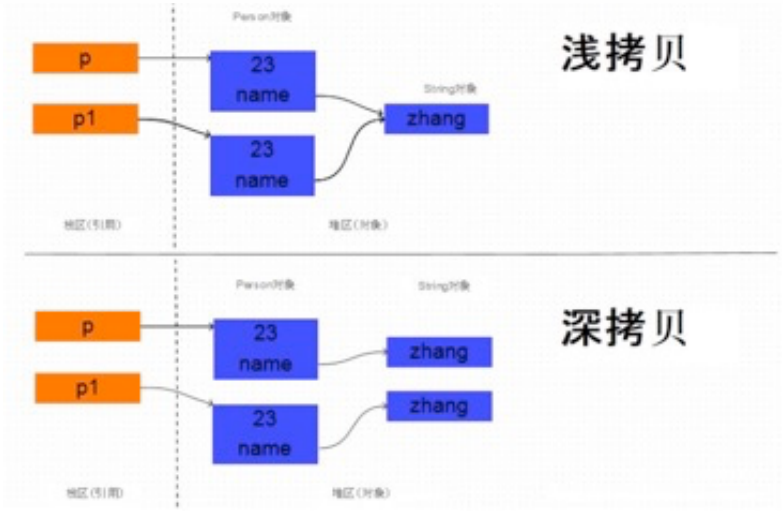
1.char、byte、short或者int常量的表达式

2.枚举常量

3.字符串字面量

对象拷贝

主要是用对象.clone()方法，这里面涉及深拷贝和浅拷贝



上图是浅拷贝和深拷贝。

对于基本类型，进行拷贝不会产生问题，对于复杂类型，例如String，需要自己实现：

1.实现Cloneable接口在Clone()方法中手动的new出相应的复杂类型。

数组拷贝

使用Arrays.copyOf()方法，拷贝的数组

下面两个数组会引用同一个引用。

```
        int[] arr1={1,2,3};
        int[] b= Arrays.copyOf(arr1,arr1.length);
```

数组排序

使用Arrays.sort(int a[],Comparator) 来进行排序，自己实现比较器，可以实现比较次序

第三章对象和类

重载

通过方法签名来进行重载，方法签名主要包含参数个数和参数类型，重载是编译时多态。

构造器

构造器，不可继承，this表示本类，super表示父类，使用这两个关键字需要放在方法开始行。

Java方法都是值传递。

初始化块

调用构造器的步骤：

静态代码块————————> 构造块代码————————>构造器

finalize方法

finalize方法将会在垃圾回收器清除对象之前调用，在实际应用中，不要依赖使用finalize方法回收任何资源，这是因为很难知道这个方法什么时候调用。

包

修饰符	当前类	同包	子类	其他包
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

第四章 继承

重写

运行时多态，在运行时，采用动态绑定调用方法，根据引用的实际类型调用重写的方法，要实现运行时多态需要有3个条件

1.继承

2.父类引用指向子类对象

3.调用覆盖的方法

需要注意的是，如果是private方法、static方法、final方法，采用的是静态绑定，根据隐式类型调用。

final

可以修饰类、方法和变量。不允许继承、修改

instanceOf

A instanceOf B用于检测A是否是B类型，如果A是null，不会抛出异常，只会返回false

抽象类

使用abstract修饰，抽象类不能被实例化，可以没有抽象方法，有构造器，有实例方法。

Object类

所有类的父类，类加载器第一个加载的类，其中主要的方法：

equals ():判断两个对象是否具有相同的引用，String重写了方法，主要是内容

hashCode():哈希值，默认是对象存储地址，String类是内容导出

clone():克隆对象

wait(long timeout):经典的等待／通知方式

notify():释放锁

finalize():析构函数

对象包装器与自动装箱

int是基本类型，Integer类是包装器类，有如下的包装器类：

Integer、Long、Float、Double、Byte、Character、Void和Boolean，前六个的父类是Number类。这些类是final修饰，不能继承。

将一个基本类型赋值给包装类，叫做装箱，反过来叫做拆箱。

Integer包装类

valueOf()方法，最后调用静态方法valueOf()方法

```
public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
```

可以看到，这里执行该方法的时候，使用了一个静态内部类Interger缓存，如果在缓存范围内使用的是这个类，否则使用的是构造器，下面是代码；

```
 private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =

 sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null) {
                try {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignore it.
                }
            }
            high = h;

            cache = new Integer[(high - low) + 1];
```

```
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }
```

这里的最大值和最小值是可以配置的，在调用valueOf()方法的时候，将新建的值存储在数组内，如果多个装箱操作在范围内，那么将会是同一个值。

对于构造方法：

```
 public Integer(int value) {
        this.value = value;
    }
public Integer(String s) throws NumberFormatException {
        this.value = parseInt(s, 10);
    }
```

值得注意的是可以将value是null，只是在使用的时候可能抛出空指针异常。

Long包装类

Long也有相关的缓存机制，代码如下：

```
public static Long valueOf(long l) {
        final int offset = 128;
        if (l >= -128 && l <= 127) { // will cache
            return LongCache.cache[(int)l + offset];
        }
        return new Long(l);
    }
public static Long valueOf(long l) {
        final int offset = 128;
        if (l >= -128 && l <= 127) { // will cache
            return LongCache.cache[(int)l + offset];
        }
        return new Long(l);
    }
```

这是与Integer机制相同的。注意构造器依旧没有使用缓存。

枚举类

通过关键字enum声明枚举类型，下面是Enum部分源码

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType,
                                            String name) {
        T result = enumType.enumConstantDirectory().get(name);
        if (result != null)
            return result;
        if (name == null)
            throw new NullPointerException("Name is null");
        throw new IllegalArgumentException(
            "No enum constant " + enumType.getCanonicalName() + "." + name);
    }
```

valueOf方法，将字符串转化为指定枚举类型。

```
  public final int ordinal() {
        return ordinal;
    }
```

ordinal方法，返回enum声明的位置，从位置0开始计数。

```
   public final int compareTo(E o) {
        Enum<?> other = (Enum<?>)o;
        Enum<E> self = this;
        if (self.getClass() != other.getClass() && // optimization
            self.getDeclaringClass() != other.getDeclaringClass())
            throw new ClassCastException();
        return self.ordinal - other.ordinal;
    }
```

比较的是两个枚举常量的位置。

下面是枚举测试

```
enum Size
{
    SMALL,MEDUIM,LARGE
}
Size s=Enum.valueOf(Size.class,"SMALL");
System.out.println(s.toString());
System.out.println(Size.SMALL.ordinal());
System.out.println(Size.LARGE.ordinal());
```

反射

指在运行状态中,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,都能调用它的任意一个方法.这种动态获取信息,以及动态调用对象方法的功能叫java语言的反射机制。

反射的三大基石：Class、Field和Method类

反射的实现原理是类加载机制，在高级篇将会讲到。

第五章 接口以及内部类

接口

属性是public final修饰

方法是抽象方法

接口可以多继承

```
interface Animal
{
    public final int count=10;
     //下面相当于 public void run();
    void run();

}
```

接口回调

先定义接口，然后写使用者和实现者，再利用参数把视线中传递给使用者，使用者利用接口调用实现者相应的功能。

Lambda表达式

内部类

内部类是定义在另一个类中的类，为什么要使用内部类呢？有三点原因：

1.内部类方法可以访问该类定义所在的作用域中的数据，包括私有的数据。

2.内部类可以对同一个包中的其他类隐藏起来。

3.想要定义一个回掉函数不想写大量的代码，使用匿名内部类比较便捷。

成员内部函数 ： 当作实例方法或变量一样理解

1定义位置：类以内，方法之外，没有static修饰

2.本身定义属性和方法：只能定义非静态的属性和方法。

3.能访问外部类的所有静态和非静态的属性或者方法。

实现代码如下：

```
package cn.edu.hust;
```

```
public class Chapter02
{
    public static int count=10;
    public static void test()
    {
        System.out.println("test....");
    }
    public void test2()
    {
        System.out.println("test2....");
    }
    class Animal
    {
        private int age=10;
        public  void cry()
        {
            test();
            test2();
            System.out.println();
        }
         //内部不能自定义静态方法
    }
    public static void main(String[] args)
    {
        Chapter02 t=new Chapter02();
        Chapter02.Animal anmal=t.new Animal();
        anmal.cry();
    }
}
```

Static inner class 静态内部类

位置定义，类以内，方法以外，static修饰

本身定义：可以定义静态和非静态的属性和方法。

能访问什么：外部类的静态属性和方法

Outer.Inner inner=new Outer.inner();

局部内部类

定义位置：方法里面的类，不能用public或者static修饰

本身定义：访问方法内final修饰的局部变量

只能在方法内创建

实例代码：

```
package cn.edu.hust.proxy.jdk;
```

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class GameProxy {
    public static  GameTask  invoke(ClassLoader classLoader, Class<?>[]
interfaces, final Object object)
    {
        return (GameTask) Proxy.newProxyInstance(classLoader, interfaces, new
InvocationHandler() {
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

                System.out.println("使用动态代理");
                return method.invoke(object,args);
            }
        });
    }
}
```

匿名内部类

没有名字，是一个局部内部类

作用：

不破坏访问权限的情况下，可以使用外部内的私有成员变量和方法

接口公开

java通过接口和内部类两种机制实现多继承。

Exception

异常，运行时的概念。

Throwable：运行时可能碰到的任何问题的总称。

(1)Error:非常严重的错误，系统不能处理

(2)Exception:从代码角度看时程序员可以处理的问题。

Exception下面有两个分支：

RuntimeExceptin：非受查异常，主要包含以下情况

1.错误的类型转换

2.数组的越界访问

3.访问null指针

这类错误主要是程序猿的问题

IOException：受查异常

课外考题：12、final，finally，finalize 的区别。
final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。
finally 是异常处理语句结构的一部分，表示总是执行。
finalize 是Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。

JDK1.8新特性

接口中允许添加静态方法和默认方法

例如下面的代码

```
interface Animal
{
    public static void cry()
    {
        System.out.println("嗷嗷嗷....");
    }
    default void eat()
    {
        System.out.println("eat something");
    }
```

默认方法可以不被重写，因为已经有默认实现。

- **Lambda 表达式** – Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中。
- **方法引用** – 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
- **默认方法** – 默认方法就是一个在接口里面有了一个实现的方法。
- **新工具** – 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。
- **Stream API** –新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。
- **Date Time API** – 加强对日期与时间的处理。
- **Optional 类** – Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
- **Nashorn, JavaScript 引擎** – Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

提高篇

ThreadLocal

ThreadLocal用于线程间的数据隔离。

ThreadLocal的应用场合，最适合的是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。

ThreadLocal是什么

当使用ThreadLocal维护变量时**ThreadLocal的确是数据的隔离，但是并非数据的复制，而是在每一个线程中创建一个新的数据对象，然后每一个线程使用的是不一样的。**

原理

ThreadLocal，连接ThreadLocalMap和Thread。来处理Thread的TheadLocalMap属性，包括init初始化属性赋值、get对应的变量，set设置变量等。通过当前线程，获取线程上的ThreadLocalMap属性，对数据进行get、set等操作。

ThreadLocalMap，用来存储数据，采用类似hashmap机制，存储了以threadLocal为key，需要隔离的数据为value的Entry键值对数组结构。

ThreadLocal，有个ThreadLocalMap类型的属性，存储的数据就放在这儿。

ThreadLocal、ThreadLocalMap、Thread之间的关系

ThreadLocalMap是ThreadLocal内部类，由ThreadLocal创建，Thread有ThreadLocal.ThreadLocalMap类型的属性。

Thread源码如下：

```
public
class Thread implements Runnable {
    /*...其他属性...*/

    /* ThreadLocal values pertaining to this thread. This map is maintained
     * by the ThreadLocal class. */
    ThreadLocal.ThreadLocalMap threadLocals = null;

    /*
     * InheritableThreadLocal values pertaining to this thread. This map is
     * maintained by the InheritableThreadLocal class.
     */
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

ThreadLocal代码如下：

```
public class ThreadLocal<T> {
    /**..其他属性和方法稍后介绍...*/
    /**
     * ThreadLocalMap is a customized hash map suitable only for
     * maintaining thread local values. No operations are exported
     * outside of the ThreadLocal class. The class is package private to
     * allow declaration of fields in class Thread.  To help deal with
     * very large and long-lived usages, the hash table entries use
     * WeakReferences for keys. However, since reference queues are not
     * used, stale entries are guaranteed to be removed only when
     * the table starts running out of space.
     */
    static class ThreadLocalMap {
```

由ThreadLocal对Thread的ThreadLocalMap进行赋值

```
/**
 * Create the map associated with a ThreadLocal. Overridden in
 * InheritableThreadLocal.
 *
 * @param t the current thread
 * @param firstValue value for the initial entry of the map
 */
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

ThreadLocalMap简介

看名字就知道是个map，没错，这就是个hashMap机制实现的map，用Entry数组来存储键值对，key是ThreadLocal对象，value则是具体的值。值得一提的是，为了方便GC，Entry继承了WeakReference，也就是弱引用。里面有一些具体关于如何清理过期的数据、扩容等机制，思路基本和hashmap差不多，有兴趣的可以自行阅读了解，这边只需知道大概的数据存储结构即可。

```
/**
 * ThreadLocalMap is a customized hash map suitable only for
 * maintaining thread local values. No operations are exported
 * outside of the ThreadLocal class. The class is package private to
 * allow declaration of fields in class Thread.  To help deal with
 * very large and long-lived usages, the hash table entries use
 * WeakReferences for keys. However, since reference queues are not
 * used, stale entries are guaranteed to be removed only when
 * the table starts running out of space.
 */
static class ThreadLocalMap {

    /**
```

```
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
     * ThreadLocal object).  Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table.  Such entries are referred to
     * as "stale entries" in the code that follows.
     */
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
```

Thread同步机制的比较

ThreadLocal和线程同步机制相比有什么优势呢?

Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

总结

ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

BIO、NIO和AIO

BIO

同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。

BIO模型程序开发起来较为简单，易于把握。

NIO

同步非阻塞，同时支持阻塞与非阻塞模式，主要使用的是Channel和Buffer以及Selector。

NIO本身是基于事件驱动的思想来实现的，其目的就是解决BIO的大并发问题，在BIO模型中，如果需要并发处理多个I/O请求，那就需要多线程来支持，NIO使用了多路复用器机制，以socket使用来说，多路复用器通过不断轮询各个连接的状态，只有在socket有流可读或者可写时，应用程序才需要去处理它，在线程的使用上，就不需要一个连接就必须使用一个处理线程了，而是只是有效请求时（确实需要进行I/O处理时），才会使用一个线程去处理，这样就避免了BIO模型下大量线程处于阻塞等待状态的情景。

NIO采用的是Reactor模式，他要求主线程只负责监听文件描述上是否有事件发生，有的话，就立即将该事件通知工作线程，除此之外，主线程不做任何其他实质性工作，读写数据，接受新的连接，以及处理客户请求均在工作线程中完成。

AIO

异步非阻塞I/O模型。

当然AIO的异步特性并不是Java实现的伪异步，而是使用了系统底层API的支持，在Unix系统下，采用了epoll IO模型，而windows便是使用了IOCP模型。

AIO采用的是Proactor模式，Proactor模式将所有I/O操作都交给主线程和内核来处理，工作线程仅仅负责业务逻辑。

同步、异步、组塞和非组塞

同步I/O　每个请求必须逐个地被处理，一个请求的处理会导致整个流程的暂时等待，这些事件无法并发地执行。用户线程发起I/O请求后需要等待或者轮询内核I/O操作完成后才能继续执行。

异步I/O　多个请求可以并发地执行，一个请求或者任务的执行不会导致整个流程的暂时等待。用户线程发起I/O请求后仍然继续执行，当内核I/O操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

阻塞　某个请求发出后，由于该请求操作需要的条件不满足，请求操作一直阻塞，不会返回，直到条件满足。

非阻塞　请求发出后，若该请求需要的条件不满足，则立即返回一个标志信息告知条件不满足，而不会一直等待。一般需要通过循环判断请求条件是否满足来获取请求结果。

需要注意的是，阻塞并不等价于同步，而非阻塞并非等价于异步。事实上这两组概念描述的是I/O模型中的两个不同维度。

同步和异步着重点在于多个任务执行过程中，后发起的任务是否必须等先发起的任务完成之后再进行。而不管先发起的任务请求是阻塞等待完成，还是立即返回通过循环等待请求成功。

而阻塞和非阻塞重点在于请求的方法是否立即返回（或者说是否在条件不满足时被阻塞）。

I/O多路复用

I/O多路复用会用到select或者poll函数，这两个函数也会使线程阻塞，但是和阻塞I/O所不同的是，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。

从流程上来看，使用select函数进行I/O请求和同步阻塞模型没有太大的区别，甚至还多了添加监视Channel，以及调用select函数的额外操作，增加了额外工作。但是，使用 select以后最大的优势是用户可以在一个线程内同时处理多个Channel的I/O请求。用户可以注册多个Channel，然后不断地调用select读取被激活的Channel，即可达到在同一个线程内同时处理多个I/O请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

调用select/poll该方法由一个用户态线程负责轮询多个Channel，直到某个阶段1的数据就绪，再通知实际的用户线程执行阶段2的拷贝。 通过一个专职的用户态线程执行非阻塞I/O轮询，模拟实现了阶段一的异步化。

序列与反序列化

序列化：将**Java**对象转换为字节序列的过程。

反序列化：把字节序列恢复为**Java**对象的过程。

如何实现Java序列化与反序列化

1）JDK类库中**序列化**API

java.io.ObjectOutputStream：表示对象输出流

它的writeObject(Object obj)方法可以对参数指定的obj对象进行**序列化**，把得到的字节序列写到一个目标输出流中。

java.io.ObjectInputStream：表示对象输入流

它的readObject()方法源输入流中读取字节序列，再把它们反**序列化**成为一个对象，并将其返回。

2）实现**序列化**的要求

只有实现了Serializable或Externalizable接口的类的对象才能被**序列化**，否则抛出异常。

3）实现**Java**对象**序列化**与反**序列化**的方法

假定一个Student类，它的对象需要**序列化**，可以有如下三种方法：

方法一：若Student类仅仅实现了**Serializable接口**，则可以按照以下方式进行**序列化**和反**序列化**

ObjectOutputStream采用默认的**序列化**方式，对Student对象的非transient的实例变量进行**序列化**。

ObjcetInputStream采用默认的反**序列化**方式，对对Student对象的非transient的实例变量进行反**序列化**。

方法二：若Student类仅仅实现了Serializable接口，并且还定义了readObject(ObjectInputStream in)和writeObject(ObjectOutputSteam out)，则采用以下方式进行**序列化**与反**序列化**。

ObjectOutputStream调用Student对象的writeObject(ObjectOutputStream out)的方法进行**序列化**。

ObjectInputStream会调用Student对象的readObject(ObjectInputStream in)的方法进行反**序列化**。

方法三：若Student类实现了**ExternalInalizable接口**，且Student类必须实现readExternal(ObjectInput in)和writeExternal(ObjectOutput out)方法，则按照以下方式进行**序列化**与反**序列化**。

ObjectOutputStream调用Student对象的writeExternal(ObjectOutput out))的方法进行**序列化**。

ObjectInputStream会调用Student对象的readExternal(ObjectInput in)的方法进行反**序列化**。

反序列化漏洞

如果Java应用对用户输入，即不可信数据做了反序列化处理，那么攻击者可以通过构造恶意输入，让反序列化产生非预期的对象，非预期的对象在产生过程中就有可能带来任意代码执行。

这个问题的根源在于类ObjectInputStream在反序列化时，没有对生成的对象的类型做限制；假若反序列化可以设置Java类型的白名单，那么问题的影响就小了很多。

总结：

1）**Java序列化**就是把对象转换成字节序列，而**Java**反**序列化**就是把字节序列还原成**Java**对象。

2）采用**Java序列化**与反**序列化**技术，一是可以实现数据的持久化，在MVC模式中很是有用；二是可以对象数据的远程通信。

设计模式

单例模式

java中单例模式是一种常见的设计模式，单例模式的写法有好几种，这里主要介绍三种：懒汉式单例、饿汉式单例、登记式单例。

单例模式有以下特点：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

懒汉模式

```
//懒汉式单例类.在第一次调用的时候实例化自己
public class Singleton {
    private Singleton() {}
    private static Singleton single=null;
    //静态工厂方法
    public static Singleton getInstance() {
         if (single == null) {
             single = new Singleton();
         }
        return single;
    }
}
```

线程安全的懒汉模式

```
public class Singleton {
    private static class LazyHolder {
       private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
       return LazyHolder.INSTANCE;
    }
}
```

饿汉模式

```
//饿汉式单例类.在类初始化时，已经自行实例化
public class Singleton1 {
    private Singleton1() {}
    private static final Singleton1 single = new Singleton1();
    //静态工厂方法
    public static Singleton1 getInstance() {
        return single;
    }
}
```

饿汉式在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

饿汉式和懒汉式区别：

从名字上来说，饿汉和懒汉，

饿汉就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了，

而懒汉比较懒，只有当调用getInstance的时候，才回去初始化这个单例。

代理模式

一：代理模式（静态代理）

代理模式是常用设计模式的一种，我们在软件设计时常用的代理一般是指静态代理，也就是在代码中显式指定的代理。

静态代理由 业务实现类、业务代理类 两部分组成。业务实现类 负责实现主要的业务方法，业务代理类负责对调用的业务方法作拦截、过滤、预处理，主要是在方法中首先进行预处理动作，然后调用业务实现类的方法，还可以规定调用后的操作。我们在需要调

二：动态代理的第一种实现——JDK动态代理

JDK动态代理所用到的代理类在程序调用到代理类对象时才由JVM真正创建，JVM根据传进来的 业务实现类对象 以及 方法名 ，动态地创建了一个代理类的class文件并被字节码引擎执行，然后通过该代理类对象进行方法调用。我们需要做的，只需指定代理类的预处理、调用后操作即可。

三：动态代理的第二种实现——CGlib

cglib是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对final修饰的类进行代理。

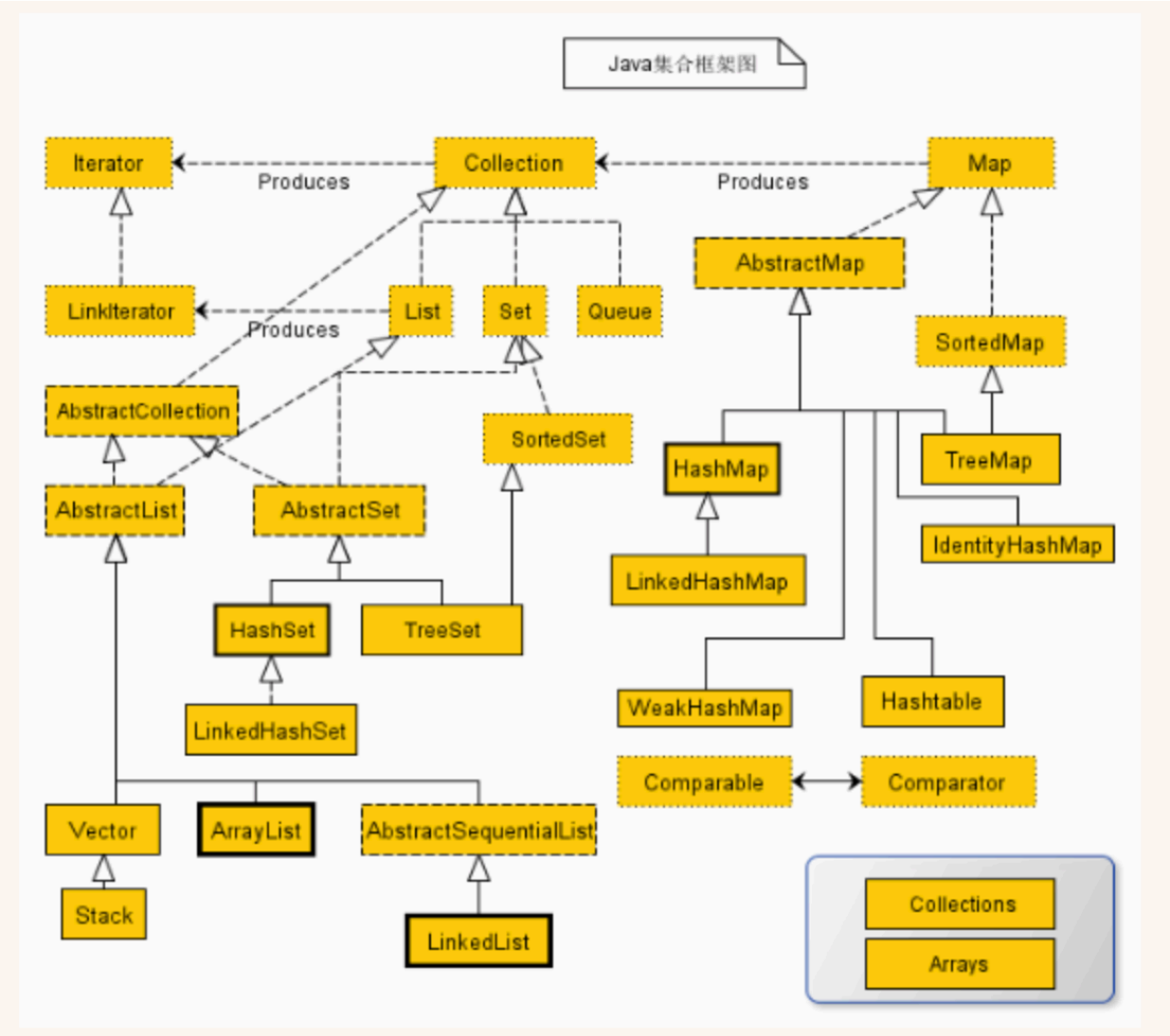
四：比较

静态代理是通过在代码中显式定义一个业务实现类一个代理，在代理类中对同名的业务方法进行包装，用户通过代理类调用被包装过的业务方法；

JDK动态代理是通过接口中的方法名，在动态生成的代理类中调用业务实现类的同名方法；

CGlib动态代理是通过继承业务类，生成的动态代理类是业务类的子类，通过重写业务方法进行代理；

Java集合



Iterator

所有集合实现的接口，有如下的方法：

boolean hasNext();是否有下一个元素

default void remove()：JDK1.8新增加的默认实现方法

E next();集合中的元素

常见集合考题

一：快速失败（fail—fast）

在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception。

原理：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。

注意：这里异常的抛出条件是检测到 modCount！=expectedmodCount 这个条件。如果集合发生变化时修改modCount值刚好又设置为了expectedmodCount值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的bug。

场景：java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

二：安全失败（fail—safe）

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。

缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

Collection接口

java中所有的集合类必须实现的接口，有以下的方法：

size():集合元素的多少。

Iteraor()：返回一个迭代器

boolean add(E e):添加元素

boolean remove(Object o);移除元素

boolean addAll(Collection<? extends E> c);添加参数集合中的元素都集合中

boolean removeAll(Collection<?> c);移除参数集合中的所有元素

List接口

List里存放的对象是有序的，同时也是可以重复的，List关注的是索引，拥有一系列和索引相关的方法，查询速度快。因为往list集合里插入或删除数据时，会伴随着后面数据的移动，所有插入删除数据速度慢。

Set接口

Set里存放的对象是无序，不能重复的，集合中的对象不按特定的方式排序(TreeSet除外)，只是简单地把对象加入集合中。

Map接口

Map集合中存储的是键值对，键不能重复，值可以重复。根据键得到值，对map集合遍历时先得到键的set集合，对set集合进行遍历，得到相应的值。

对比如下：

		是否有序	是否允许元素重复
Collection			
List		是	是
Set	AbstractSet	否	否
	HashSet		
	TreeSet	是（用二叉排序树）	
Map	AbstractMap	否	使用key-value来映射和存储数据，key必须唯一，value可以重复
	HashMap		
	TreeMap	是（用二叉排序树）	

ArrayList

ArrayList是基于数组实现的，是一个动态数组，其容量能自动增长，类似于C语言中的动态申请内存，动态增长内存

ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能被克隆。

ArrayList的实现：

对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析ArrayList的源代码：

1) 私有属性：

ArrayList定义只定义类两个私有属性：

```
/**
      * The array buffer into which the elements of the ArrayList are stored.
      * The capacity of the ArrayList is the length of this array buffer.
      */
     private transient Object[] elementData;

     /**
      * The size of the ArrayList (the number of elements it contains).
      *
      * @serial
      */
     private int size;
```

很容易理解，elementData存储ArrayList内的元素，size表示它包含的元素的数量。

有个关键字需要解释：transient。

Java的serialization提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特定对象的一个域上关闭serialization，可以在这个域前加上关键字transient。

```
public class UserInfo implements Serializable {
     private static final long serialVersionUID = 996890129747019948L;
     private String name;
     private transient String psw;

     public UserInfo(String name, String psw) {
         this.name = name;
         this.psw = psw;
     }

     public String toString() {
         return "name=" + name + ", psw=" + psw;
     }
 }

 public class TestTransient {
     public static void main(String[] args) {
         UserInfo userInfo = new UserInfo("张三", "123456");
         System.out.println(userInfo);
         try {
             // 序列化，被设置为transient的属性没有被序列化
             ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream(
                     "UserInfo.out"));
             o.writeObject(userInfo);
             o.close();
         } catch (Exception e) {
             // TODO: handle exception
```

```
            e.printStackTrace();
        }
        try {
            // 重新读取内容
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(
                    "UserInfo.out"));
            UserInfo readUserInfo = (UserInfo) in.readObject();
            //读取后psw的内容为null
            System.out.println(readUserInfo.toString());
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

被标记为transient的属性在对象被序列化的时候不会被保存。

接着回到ArrayList的分析中.....

2) 构造方法： ArrayList提供了三种方式的构造器，可以构造一个默认初始容量为10的空列表、构造一个指定初始容量的空列表以及构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

```
// ArrayList带容量大小的构造函数。
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        // 新建一个数组
        this.elementData = new Object[initialCapacity];
    }

    // ArrayList无参构造函数。默认容量是10。
    public ArrayList() {
        this(10);
    }

    // 创建一个包含collection的ArrayList
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
```

3) 元素存储：

ArrayList提供了set(int index, E element)、add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)这些添加元素的方法。下面我们一一讲解：

```
// 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。
public E set(int index, E element) {
   RangeCheck(index);

   E oldValue = (E) elementData[index];
   elementData[index] = element;
   return oldValue;
}
// 将指定的元素添加到此列表的尾部。
public boolean add(E e) {
   ensureCapacity(size + 1);
   elementData[size++] = e;
   return true;
}
// 将指定的元素插入此列表中的指定位置。
// 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。
public void add(int index, E element) {
   if (index > size || index < 0)
       throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
   // 如果数组长度不足，将进行扩容。
   ensureCapacity(size+1);  // Increments modCount!!
   // 将 elementData中从Index位置开始、长度为size-index的元素，
   // 拷贝到从下标为index+1位置开始的新的elementData数组中。
   // 即将当前位于该位置的元素以及所有后续元素右移一个位置。
   System.arraycopy(elementData, index, elementData, index + 1, size - index);
   elementData[index] = element;
   size++;
}
// 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。
public boolean addAll(Collection<? extends E> c) {
   Object[] a = c.toArray();
   int numNew = a.length;
   ensureCapacity(size + numNew);  // Increments modCount
   System.arraycopy(a, 0, elementData, size, numNew);
   size += numNew;
   return numNew != 0;
}
// 从指定的位置开始，将指定collection中的所有元素插入到此列表中。
public boolean addAll(int index, Collection<? extends E> c) {
   if (index > size || index < 0)
       throw new IndexOutOfBoundsException(
           "Index: " + index + ", Size: " + size);

   Object[] a = c.toArray();
```

```
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount

    int numMoved = size - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index, elementData, index + numNew,
numMoved);

    System.arraycopy(a, 0, elementData, index, numNew);
    size += numNew;
    return numNew != 0;
    }
```

书上都说ArrayList是基于数组实现的，属性中也看到了数组，具体是怎么实现的呢？比如就这个添加元素的方法，如果数组大，则在将某个位置的值设置为指定元素即可，如果数组容量不够了呢？

看到add(E e)中先调用了ensureCapacity(size+1)方法，之后将元素的索引赋给elementData[size]，而后size自增。例如初次添加时，size为0，add将elementData[0]赋值为e，然后size设置为1（类似执行以下两条语句elementData[0]=e;size=1）。将元素的索引赋给elementData[size]不是会出现数组越界的情况吗？这里关键就在ensureCapacity(size+1)中了。

4) 元素读取：

```
// 返回此列表中指定位置上的元素。
 public E get(int index) {
    RangeCheck(index);

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

5) 元素删除：

ArrayList提供了根据下标或者指定对象两种方式的删除功能。如下：

remove(int index):

```
// 移除此列表中指定位置上的元素。
 public E remove(int index) {
    RangeCheck(index);

    modCount++;
    E oldValue = (E) elementData[index];

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index, numMoved);
    elementData[--size] = null; // Let gc do its work

    return oldValue;
```

```
    }
```

首先是检查范围，修改modCount，保留将要被移除的元素，将移除位置之后的元素向前挪动一个位置，将list末尾元素置空（null），返回被移除的元素。

remove(Object o)

```
// 移除此列表中首次出现的指定元素（如果存在）。这是应为ArrayList中允许存放重复的元素。
 public boolean remove(Object o) {
    // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                // 类似remove(int index)，移除列表中指定位置上的元素。
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
        }
        return false;
    }
}
```

6) 调整数组容量ensureCapacity：

从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

```
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;  //增加50%+1
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
      // minCapacity is usually close to size, so this is a win:
      elementData = Arrays.copyOf(elementData, newCapacity);
    }
 }
```

从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是**其原容量的1.5倍(JDK1.8是1.5倍)**这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

Object oldData[] = elementData;//为什么要用到oldData[] 乍一看来后面并没有用到关于oldData，这句话显得多此一举！但是这是一个牵涉到内存管理的类， 所以要了解内部的问题。 而且为什么这一句还在if的内部，这跟elementData = Arrays.copyOf(elementData, newCapacity); 这句是有关系的，下面这句Arrays.copyOf的实现时新创建了newCapacity大小的内存，然后把老的elementData放入。好像也没有用到oldData，有什么问题呢。问题就在于旧的内存的引用是elementData，elementData指向了新的内存块，如果有一个局部变量oldData变量引用旧的内存块的话，在copy的过程中就会比较安全，因为这样证明这块老的内存依然有引用，分配内存的时候就不会被侵占掉，然后copy完成后这个局部变量的生命期也过去了，然后释放才是安全的。不然在copy的的时候万一新的内存或其他线程的分配内存侵占了这块老的内存，而copy还没有结束，这将是个严重的事情。

关于ArrayList和Vector区别如下：

ArrayList在内存不够时默认是扩展50% + 1个(JDK1.8是1.5倍)，Vector是默认扩展1倍。

Vector提供indexOf(obj, start)接口，ArrayList没有。

Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为线程安全需要更大的系统开销。

总结:

关于ArrayList的源码，给出几点比较重要的总结：

1、注意其三个不同的构造方法。无参构造方法构造的ArrayList的容量默认为10，带有Collection参数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData。

2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用Arrays.copyof()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList。

3、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。我们有必要对这两个方法的实现做下深入的了解。

首先来看Arrays.copyof()方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型版本的源码：

```
public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
}
```

很明显调用了另一个copyof方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源码如下：

```
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]>
newType) {
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为newlength的数组，调用System.arraycopy()方法，将原来数组中的元素复制到了新的数组中。

下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK中是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

4、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

5、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。

Vector

遗留集合，线程安全，数组实现，效率低下，底层是数组实现，实现的结构如下：

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

1）构造器

```
//这个构造器有初始化容器，扩容大小
public Vector(int initialCapacity, int capacityIncrement) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        this.elementData = new Object[initialCapacity];
        this.capacityIncrement = capacityIncrement;
    }
//初始化大小，自增为0
 public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    }
//初始化容量是10，自增为0
```

```
public Vector() {
        this(10);
    }
//使用集合初始化vector
 public Vector(Collection<? extends E> c) {
        elementData = c.toArray();
        elementCount = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, elementCount,
Object[].class);
    }
```

可以看出，在初始化Vector容器时，有四种构造器，默认容量为10，自定义初始化开始大小和自增大小。

2）添加方法

```
public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = obj;
    }

 public synchronized void insertElementAt(E obj, int index) {
        modCount++;
        if (index > elementCount) {
            throw new ArrayIndexOutOfBoundsException(index
                                                     + " > " + elementCount);
        }
        ensureCapacityHelper(elementCount + 1);
        System.arraycopy(elementData, index, elementData, index + 1, elementCount
- index);
        elementData[index] = obj;
        elementCount++;
    }

private void ensureCapacityHelper(int minCapacity) {
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

 private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacity);
        if (newCapacity - minCapacity < 0)
```

```
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
```

可以看出可以在特定位置插入元素，也可添加元素，通过源码追踪，可以看出如果没有设置自增的容量，那么将会扩容一倍，而且这些都是线程安全的。

3）删除元素

```
public synchronized boolean removeElement(Object obj) {
        modCount++;
        int i = indexOf(obj);
        if (i >= 0) {
            removeElementAt(i);
            return true;
        }
        return false;
    }

public synchronized void removeAllElements() {
        modCount++;
        // Let gc do its work
        for (int i = 0; i < elementCount; i++)
            elementData[i] = null;

        elementCount = 0;
    }

public boolean remove(Object o) {
        return removeElement(o);
    }

 public synchronized void removeElementAt(int index) {
        modCount++;
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +
                                                     elementCount);
        }
        else if (index < 0) {
```

```
            throw new ArrayIndexOutOfBoundsException(index);
        }
        int j = elementCount - index - 1;
        if (j > 0) {
            System.arraycopy(elementData, index + 1, elementData, index, j);
        }
        elementCount--;
        elementData[elementCount] = null; /* to let gc do its work */
    }
```

四种移除方法，可以根据元素、元素的位置移除元素，这是线程安全的。

LinkedList

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

主要特点是：实现底层是链表

1）构造器

```
 public LinkedList() {
    }
public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }
```

可以更具集合构建

2）添加方法

```
public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
    }
public boolean addAll(int index, Collection<? extends E> c) {
        checkPositionIndex(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        if (numNew == 0)
            return false;

        Node<E> pred, succ;
        if (index == size) {
            succ = null;
```

```
            pred = last;
        } else {
            succ = node(index);
            pred = succ.prev;
        }

        for (Object o : a) {
            @SuppressWarnings("unchecked") E e = (E) o;
            Node<E> newNode = new Node<>(pred, e, null);
            if (pred == null)
                first = newNode;
            else
                pred.next = newNode;
            pred = newNode;
        }

        if (succ == null) {
            last = pred;
        } else {
            pred.next = succ;
            succ.prev = pred;
        }

        size += numNew;
        modCount++;
        return true;
    }
   public void addFirst(E e) {
        linkFirst(e);
    }
public void addLast(E e) {
        linkLast(e);
    }
```

可以添加在尾部、首部，以及添加集合和添加元素

3）移除方法

移除元素、头节点和尾节点

适合元素增加和删除的场景。

HashSet

HashSet底层基于HashMap实现，只是实现了Set接口，非线程安全，保证元素顺序，允许null元素。

HashSet是**非同步的**。如果多个线程同时访问一个哈希 set，而其中至少一个线程修改了该 set，那么它必须 保持外部同步。这通常是通过对自然封装该 set 的对象执行同步操作来完成的。如果不存在这样的对象，则应该使用 **Collections.synchronizedSet 方法来“包装” set。最好在创建时完成这一操作，以防止对该 set 进行意外的不同步访问**：

1）构造方法

```
// 默认构造函数
public HashSet()

// 带集合的构造函数
public HashSet(Collection<? extends E> c)

// 指定HashSet初始容量和加载因子的构造函数
public HashSet(int initialCapacity, float loadFactor)

// 指定HashSet初始容量的构造函数
public HashSet(int initialCapacity)

// 指定HashSet初始容量和加载因子的构造函数,dummy没有任何作用
HashSet(int initialCapacity, float loadFactor, boolean dummy)
```

2）HashSet的主要方法有

```
 public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
 public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
```

HashSet只是简单的实现了Set接口，如果要获得某个元素只能通过迭代器访问。

需要注意的是

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

通过一个静态对象来存储value

TreeSet

TreeSet 是一个有序的集合，它的作用是提供有序的Set集合，底层是红黑树实现。它继承于AbstractSet抽象类，实现了 NavigableSet, Cloneable, java.io.Serializable接口。 TreeSet 继承于AbstractSet，所以它是一个Set集合，具有Set的属性和方法。 TreeSet 实现了NavigableSet接口，意味着它支持一系列的导航方法。比如查找与指定目标最匹配项。 TreeSet 实现了Cloneable接口，意味着它能被克隆。 TreeSet 实现了java.io.Serializable接口，意味着它支持序列化。

TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法。 TreeSet为基本操作（add、remove 和 contains）提供受保证的 log(n) 时间开销。 另外，TreeSet是非同步的。 它的iterator 方法返回的迭代器是fail-fast的。

1）TreeSet的构造器

```
// 默认构造函数。使用该构造函数，TreeSet中的元素按照自然排序进行排列。
TreeSet()

// 创建的TreeSet包含collection
TreeSet(Collection<? extends E> collection)

// 指定TreeSet的比较器
TreeSet(Comparator<? super E> comparator)

// 创建的TreeSet包含set
TreeSet(SortedSet<E> set)
```

2）API实现

```
boolean                  add(E object)
boolean                  addAll(Collection<? extends E> collection)
void                     clear()
Object                   clone()
boolean                  contains(Object object)
E                        first()
boolean                  isEmpty()
E                        last()
E                        pollFirst()
E                        pollLast()
E                        lower(E e)
E                        floor(E e)
E                        ceiling(E e)
E                        higher(E e)
boolean                  remove(Object object)
int                      size()
Comparator<? super E>    comparator()
Iterator<E>              iterator()
Iterator<E>              descendingIterator()
SortedSet<E>             headSet(E end)
NavigableSet<E>          descendingSet()
NavigableSet<E>          headSet(E end, boolean endInclusive)
SortedSet<E>             subSet(E start, E end)
NavigableSet<E>          subSet(E start, boolean startInclusive, E end, boolean
endInclusive)
NavigableSet<E>          tailSet(E start, boolean startInclusive)
SortedSet<E>             tailSet(E start)
```

总结:

(01) TreeSet实际上是TreeMap实现的。当我们构造TreeSet时；若使用不带参数的构造函数，则TreeSet的使用自然比较器；若用户需要使用自定义的比较器，则需要使用带比较器的参数。 (02) TreeSet是非线程安全的。 (03) TreeSet实现java.io.Serializable的方式。当写入到输出流时，依次写入“比较器、容量、全部元素”；当读出输入流时，再依次读取。

(04)要想取出Set中某个元素，只能遍历Set

LinkedHashSet

LinkedHashSet集合也是根据元素hashCode值来决定元素存储位置，但它同时使用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。也就是说，当遍历LinkedHashSet集合里元素时，HashSet将会按元素的添加顺序来访问集合里的元素。LinkedHashSet需要维护元素的插入顺序，因此性能略低于HashSet的性能，但在迭代访问Set里的全部元素时将有很好的性能，因为它以链表来维护内部顺序。继承自HashSet，内部也是实现了自己的构造器方法。

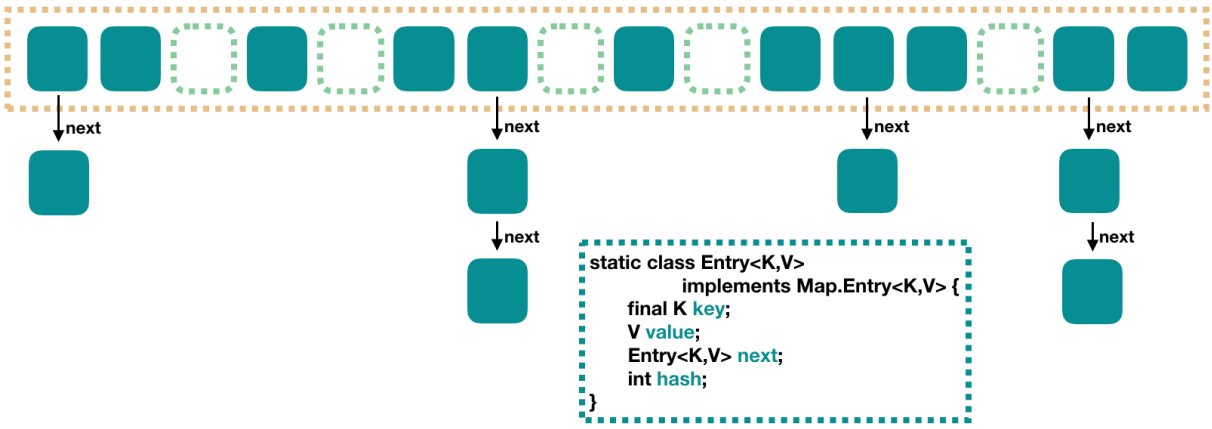
HashMap

Java7 HashMap

HashMap 是最简单的，一来我们非常熟悉，二来就是它不支持并发操作，所以源码也非常简单。

首先，我们用下面这张图来介绍 HashMap 的结构。

Java7 HashMap 结构



大方向上，HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。

上图中，每个绿色的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。

loadFactor：负载因子，默认为 0.75。

threshold：扩容的阈值，等于 capacity * loadFactor

put 过程分析

还是比较简单的，跟着代码走一遍吧。

数组初始化

在第一个元素插入 HashMap 的时候做一次数组的初始化，就是先确定初始的数组大小，并计算数组扩容的阈值。

这里有一个将数组大小保持为 2 的 n 次方的做法，Java7 和 Java8 的 HashMap 和 ConcurrentHashMap 都有相应的要求，只不过实现的代码稍微有些不同，后面再看到的时候就知道了。

计算具体数组位置

这个简单，我们自己也能 YY 一个：使用 key 的 hash 值对数组长度进行取模就可以了。

这个方法很简单，简单说就是取 hash 值的低 n 位。如在数组长度为 32 的时候，其实取的就是 key 的 hash 值的低 5 位，作为它在数组中的下标位置。

添加节点到链表中

找到数组下标后，会先进行 key 判重，如果没有重复，就准备将新值放入到链表的表头。

这个方法的主要逻辑就是先判断是否需要扩容，需要的话先扩容，然后再将这个新的数据插入到扩容后的数组的相应位置处的链表的表头。

数组扩容

前面我们看到，在插入新值的时候，如果当前的 size 已经达到了阈值，并且要插入的数组位置上已经有元素，那么就会触发扩容，扩容后，数组大小为原来的 2 倍。

扩容就是用一个新的大数组替换原来的小数组，并将原来数组中的值迁移到新的数组中。

由于是双倍扩容，迁移过程中，会将原来 table[i] 中的链表的所有节点，分拆到新的数组的 newTable[i] 和 newTable[i + oldLength] 位置上。如原来数组长度是 16，那么扩容后，原来 table[0] 处的链表中的所有元素会被分配到新数组中 newTable[0] 和 newTable[16] 这两个位置。代码比较简单，这里就不展开了。

get 过程分析

相对于 put 过程，get 过程是非常简单的。

1. 根据 key 计算 hash 值。
2. 找到相应的数组下标：hash & (length – 1)。
3. 遍历该数组位置处的链表，直到找到相等(==或equals)的 key。

getEntry(key):

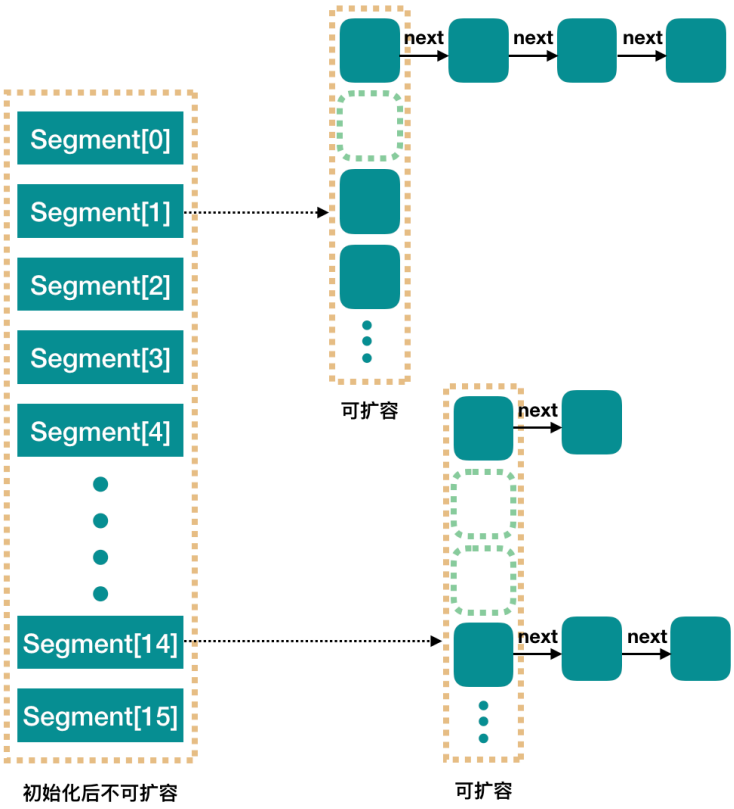
Java7 ConcurrentHashMap

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

Java7 ConcurrentHashMap 结构



concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

初始化

initialCapacity：初始容量，这个值指的是整个 ConcurrentHashMap 的初始容量，实际操作的时候需要平均分给每个 Segment。

loadFactor：负载因子，之前我们说了，Segment 数组不可以扩容，所以这个负载因子是给每个 Segment 内部使用的。

```
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // Find power-of-two sizes best matching arguments
    int sshift = 0;
    int ssize = 1;
    // 计算并行级别 ssize，因为要保持并行级别是 2 的 n 次方
    while (ssize < concurrencyLevel) {
```

```
            ++sshift;
            ssize <<= 1;
        }
        // 我们这里先不要那么烧脑，用默认值，concurrencyLevel 为 16，sshift 为 4
        // 那么计算出 segmentShift 为 28，segmentMask 为 15，后面会用到这两个值
        this.segmentShift = 32 - sshift;
        this.segmentMask = ssize - 1;

        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;

        // initialCapacity 是设置整个 map 初始的大小，
        // 这里根据 initialCapacity 计算 Segment 数组中每个位置可以分到的大小
        // 如 initialCapacity 为 64，那么每个 Segment 或称之为"槽"可以分到 4 个
        int c = initialCapacity / ssize;
        if (c * ssize < initialCapacity)
            ++c;
        // 默认 MIN_SEGMENT_TABLE_CAPACITY 是 2，这个值也是有讲究的，因为这样的话，对于具体
的槽上，
        // 插入一个元素不至于扩容，插入第二个的时候才会扩容
        int cap = MIN_SEGMENT_TABLE_CAPACITY;
        while (cap < c)
            cap <<= 1;

        // 创建 Segment 数组，
        // 并创建数组的第一个元素 segment[0]
        Segment<K,V> s0 =
            new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                             (HashEntry<K,V>[])new HashEntry[cap]);
        Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
        // 往数组写入 segment[0]
        UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
        this.segments = ss;
    }
```

初始化完成，我们得到了一个 Segment 数组。

我们就当是用 new ConcurrentHashMap() 无参构造函数进行初始化的，那么初始化完成后：

- Segment 数组长度为 16，不可以扩容
- Segment[i] 的默认大小为 2，负载因子是 0.75，得出初始阈值为 1.5，也就是以后插入第一个元素不会触发扩容，插入第二个会进行第一次扩容
- 这里初始化了 segment[0]，其他位置还是 null，至于为什么要初始化 segment[0]，后面的代码会介绍
- 当前 segmentShift 的值为 32 – 4 = 28，segmentMask 为 16 – 1 = 15，姑且把它们简单翻译为移位数和掩码，这两个值马上就会用到

put 过程分析

我们先看 put 的主流程，对于其中的一些关键细节操作，后面会进行详细介绍。

```
public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    // 1. 计算 key 的 hash 值
    int hash = hash(key);
    // 2. 根据 hash 值找到 Segment 数组中的位置 j
    //    hash 是 32 位，无符号右移 segmentShift(28) 位，剩下低 4 位，
    //    然后和 segmentMask(15) 做一次与操作，也就是说 j 是 hash 值的最后 4 位，也就
是槽的数组下标
    int j = (hash >>> segmentShift) & segmentMask;
    // 刚刚说了，初始化的时候初始化了 segment[0]，但是其他位置还是 null，
    // ensureSegment(j) 对 segment[j] 进行初始化
    if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
         (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
        s = ensureSegment(j);
    // 3. 插入新值到 槽 s 中
    return s.put(key, hash, value, false);
}
```

第一层皮很简单，根据 hash 值很快就能找到相应的 Segment，之后就是 Segment 内部的 put 操作了。

Segment 内部是由 数组+链表 组成的。

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 在往该 segment 写入前，需要先获取该 segment 的独占锁
    //    先看主流程，后面还会具体介绍这部分内容
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        // 这个是 segment 内部的数组
        HashEntry<K,V>[] tab = table;
        // 再利用 hash 值，求应该放置的数组下标
        int index = (tab.length - 1) & hash;
        // first 是数组该位置处的链表的表头
        HashEntry<K,V> first = entryAt(tab, index);

        // 下面这串 for 循环虽然很长，不过也很好理解，想想该位置没有任何元素和已经存在一
个链表这两种情况
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
```

```
                        // 覆盖旧值
                        e.value = value;
                        ++modCount;
                    }
                    break;
                }
                // 继续顺着链表走
                e = e.next;
            }
            else {
                // node 到底是不是 null，这个要看获取锁的过程，不过和这里都没有关系。
                // 如果不为 null，那就直接将它设置为链表表头；如果是null，初始化并设置为
链表表头。
                if (node != null)
                    node.setNext(first);
                else
                    node = new HashEntry<K,V>(hash, key, value, first);

                int c = count + 1;
                // 如果超过了该 segment 的阈值，这个 segment 需要扩容
                if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                    rehash(node); // 扩容后面也会具体分析
                else
                    // 没有达到阈值，将 node 放到数组 tab 的 index 位置，
                    // 其实就是将新的节点设置成原链表的表头
                    setEntryAt(tab, index, node);
                ++modCount;
                count = c;
                oldValue = null;
                break;
            }
        }
    } finally {
        // 解锁
        unlock();
    }
    return oldValue;
}
```

整体流程还是比较简单的，由于有独占锁的保护，所以 segment 内部的操作并不复杂。至于这里面的并发问题，我们稍后再进行介绍。

到这里 put 操作就结束了，接下来，我们说一说其中几步关键的操作。

初始化槽: ensureSegment

ConcurrentHashMap 初始化的时候会初始化第一个槽 segment[0]，对于其他槽来说，在插入第一个值的时候进行初始化。

这里需要考虑并发，因为很可能会有多个线程同时进来初始化同一个槽 segment[k]，不过只要有一个成功了就可以。

```
private Segment<K,V> ensureSegment(int k) {
    final Segment<K,V>[] ss = this.segments;
    long u = (k << SSHIFT) + SBASE; // raw offset
    Segment<K,V> seg;
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {
        // 这里看到为什么之前要初始化 segment[0] 了，
        // 使用当前 segment[0] 处的数组长度和负载因子来初始化 segment[k]
        // 为什么要用“当前”，因为 segment[0] 可能早就扩容过了
        Segment<K,V> proto = ss[0];
        int cap = proto.table.length;
        float lf = proto.loadFactor;
        int threshold = (int)(cap * lf);

        // 初始化 segment[k] 内部的数组
        HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];
        if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
            == null) { // 再次检查一遍该槽是否被其他线程初始化了。

            Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);
            // 使用 while 循环，内部用 CAS，当前线程成功设值或其他线程成功设值后，退出
            while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
                   == null) {
                if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))
                    break;
            }
        }
    }
    return seg;
}
```

总的来说，ensureSegment(int k) 比较简单，对于并发操作使用 CAS 进行控制。

> 我没搞懂这里为什么要搞一个 while 循环，CAS 失败不就代表有其他线程成功了吗，为什么要再进行判断?

获取写入锁: scanAndLockForPut

前面我们看到，在往某个 segment 中 put 的时候，首先会调用 node = tryLock() ? null : scanAndLockForPut(key, hash, value)，也就是说先进行一次 tryLock() 快速获取该 segment 的独占锁，如果失败，那么进入到 scanAndLockForPut 这个方法来获取锁。

```
private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
    HashEntry<K,V> first = entryForHash(this, hash);
    HashEntry<K,V> e = first;
    HashEntry<K,V> node = null;
    int retries = -1; // negative while locating node
```

```
    // 循环获取锁
    while (!tryLock()) {
        HashEntry<K,V> f; // to recheck first below
        if (retries < 0) {
            if (e == null) {
                if (node == null) // speculatively create node
                    // 进到这里说明数组该位置的链表是空的，没有任何元素
                    // 当然，进到这里的另一个原因是 tryLock() 失败，所以该槽存在并发，
不一定是该位置
                    node = new HashEntry<K,V>(hash, key, value, null);
                retries = 0;
            }
            else if (key.equals(e.key))
                retries = 0;
            else
                // 顺着链表往下走
                e = e.next;
        }
        // 重试次数如果超过 MAX_SCAN_RETRIES（单核1多核64），那么不抢了，进入到阻塞队列
等待锁
        //    lock() 是阻塞方法，直到获取锁后返回
        else if (++retries > MAX_SCAN_RETRIES) {
            lock();
            break;
        }
        else if ((retries & 1) == 0 &&
                 // 这个时候是有大问题了，那就是有新的元素进到了链表，成为了新的表头
                 //     所以这边的策略是，相当于重新走一遍这个 scanAndLockForPut 方法
                 (f = entryForHash(this, hash)) != first) {
            e = first = f; // re-traverse if entry changed
            retries = -1;
        }
    }
    return node;
}
```

这个方法有两个出口，一个是 tryLock() 成功了，循环终止，另一个就是重试次数超过了 MAX_SCAN_RETRIES，进到 lock() 方法，此方法会阻塞等待，直到成功拿到独占锁。

这个方法就是看似复杂，但是其实就是做了一件事，那就是获取该 segment 的独占锁，如果需要的话顺便实例化了一下 node。

扩容: rehash

重复一下，segment 数组不能扩容，扩容是 segment 数组某个位置内部的数组 HashEntry[] 进行扩容，扩容后，容量为原来的 2 倍。

首先，我们要回顾一下触发扩容的地方，put 的时候，如果判断该值的插入会导致该 segment 的元素个数超过阈值，那么先进行扩容，再插值，读者这个时候可以回去 put 方法看一眼。

该方法不需要考虑并发，因为到这里的时候，是持有该 segment 的独占锁的。

```
// 方法参数上的 node 是这次扩容后，需要添加到新的数组中的数据。
private void rehash(HashEntry<K,V> node) {
    HashEntry<K,V>[] oldTable = table;
    int oldCapacity = oldTable.length;
    // 2 倍
    int newCapacity = oldCapacity << 1;
    threshold = (int)(newCapacity * loadFactor);
    // 创建新数组
    HashEntry<K,V>[] newTable =
        (HashEntry<K,V>[]) new HashEntry[newCapacity];
    // 新的掩码，如从 16 扩容到 32，那么 sizeMask 为 31，对应二进制 '000...00011111'
    int sizeMask = newCapacity - 1;

    // 遍历原数组，老套路，将原数组位置 i 处的链表拆分到 新数组位置 i 和 i+oldCap 两个位
置
    for (int i = 0; i < oldCapacity ; i++) {
        // e 是链表的第一个元素
        HashEntry<K,V> e = oldTable[i];
        if (e != null) {
            HashEntry<K,V> next = e.next;
            // 计算应该放置在新数组中的位置，
            // 假设原数组长度为 16，e 在 oldTable[3] 处，那么 idx 只可能是 3 或者是 3
+ 16 = 19
            int idx = e.hash & sizeMask;
            if (next == null)   // 该位置处只有一个元素，那比较好办
                newTable[idx] = e;
            else { // Reuse consecutive sequence at same slot
                // e 是链表表头
                HashEntry<K,V> lastRun = e;
                // idx 是当前链表的头结点 e 的新位置
                int lastIdx = idx;

                // 下面这个 for 循环会找到一个 lastRun 节点，这个节点之后的所有元素是将
要放到一起的
                for (HashEntry<K,V> last = next;
                     last != null;
                     last = last.next) {
                    int k = last.hash & sizeMask;
                    if (k != lastIdx) {
                        lastIdx = k;
                        lastRun = last;
                    }
                }
                // 将 lastRun 及其之后的所有节点组成的这个链表放到 lastIdx 这个位置
```

```
                newTable[lastIdx] = lastRun;
                // 下面的操作是处理 lastRun 之前的节点，
                //    这些节点可能分配在另一个链表中，也可能分配到上面的那个链表中
                for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
                    V v = p.value;
                    int h = p.hash;
                    int k = h & sizeMask;
                    HashEntry<K,V> n = newTable[k];
                    newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
                }
            }
        }
    }
    // 将新来的 node 放到新数组中刚刚的 两个链表之一 的 头部
    int nodeIndex = node.hash & sizeMask; // add the new node
    node.setNext(newTable[nodeIndex]);
    newTable[nodeIndex] = node;
    table = newTable;
}
```

这里的扩容比之前的 HashMap 要复杂一些，代码难懂一点。上面有两个挨着的 for 循环，第一个 for 有什么用呢?

仔细一看发现，如果没有第一个 for 循环，也是可以工作的，但是，这个 for 循环下来，如果 lastRun 的后面还有比较多的节点，那么这次就是值得的。因为我们只需要克隆 lastRun 前面的节点，后面的一串节点跟着 lastRun 走就是了，不需要做任何操作。

我觉得 Doug Lea 的这个想法也是挺有意思的，不过比较坏的情况就是每次 lastRun 都是链表的最后一个元素或者很靠后的元素，那么这次遍历就有点浪费了。不过 Doug Lea 也说了，根据统计，如果使用默认的阈值，大约只有 1/6 的节点需要克隆。

get 过程分析

相对于 put 来说，get 真的不要太简单。

计算 hash 值，找到 segment 数组中的具体位置，或我们前面用的“槽”

槽中也是一个数组，根据 hash 找到数组中具体的位置

到这里是链表了，顺着链表进行查找即可

```
public V get(Object key) {
    Segment<K,V> s; // manually integrate access methods to reduce overhead
    HashEntry<K,V>[] tab;
    // 1. hash 值
    int h = hash(key);
    long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    // 2. 根据 hash 找到对应的 segment
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        // 3. 找到segment 内部数组相应位置的链表，遍历
```

```
        for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                 (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
             e != null; e = e.next) {
            K k;
            if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                return e.value;
        }
    }
    return null;
}
```

现在我们已经说完了 put 过程和 get 过程，我们可以看到 get 过程中是没有加锁的，那自然我们就需要去考虑并发问题。

添加节点的操作 put 和删除节点的操作 remove 都是要加 segment 上的独占锁的，所以它们之间自然不会有问题，我们需要考虑的问题就是 get 的时候在同一个 segment 中发生了 put 或 remove 操作。

remove 操作我们没有分析源码，所以这里说的读者感兴趣的话还是需要到源码中去求实一下的。

get 操作需要遍历链表，但是 remove 操作会"破坏"链表。

如果 remove 破坏的节点 get 操作已经过去了，那么这里不存在任何问题。

如果 remove 先破坏了一个节点，分两种情况考虑。 1、如果此节点是头结点，那么需要将头结点的 next 设置为数组该位置的元素，table 虽然使用了 volatile 修饰，但是 volatile 并不能提供数组内部操作的可见性保证，所以源码中使用了 UNSAFE 来操作数组，请看方法 setEntryAt。2、如果要删除的节点不是头结点，它会将要删除节点的后继节点接到前驱节点中，这里的并发保证就是 next 属性是 volatile 的。

Java8 HashMap

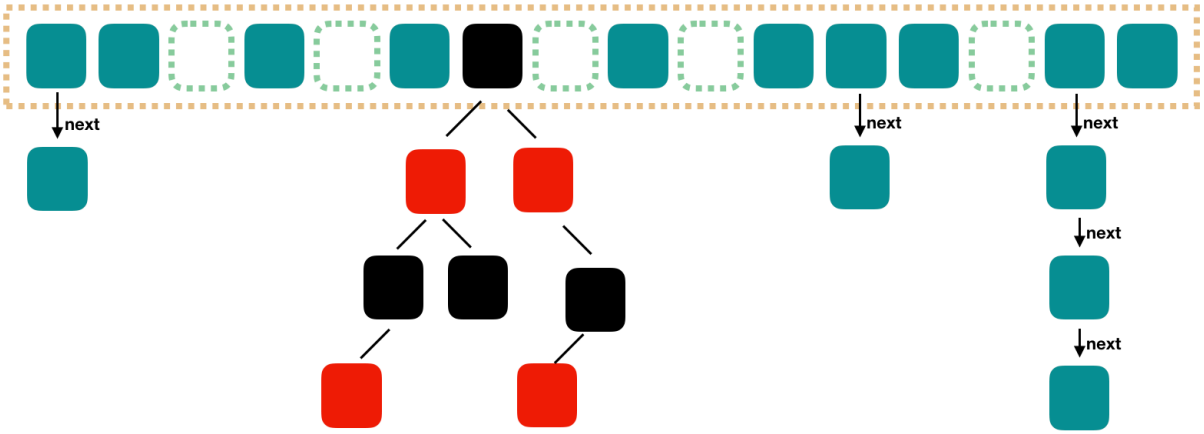
Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 O(n)。

为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

来一张图简单示意一下吧：

Java8 HashMap 结构



Java7 中使用 Entry 来代表每个 HashMap 中的数据节点，Java8 中使用 Node，基本没有区别，都是 key，value，hash 和 next 这四个属性，不过，Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode。

我们根据数组元素中，第一个节点数据类型是 Node 还是 TreeNode 来判断该位置下是链表还是红黑树的。

put 过程分析

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

// 第三个参数 onlyIfAbsent 如果是 true，那么只有在不存在该 key 时才会进行 put 操作
// 第四个参数 evict 我们这里不关心
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 第一次 put 值的时候，会触发下面的 resize()，类似 java7 的第一次 put 也要初始化数组长度
    // 第一次 resize 和后续的扩容有些不一样，因为这次是数组从 null 初始化到默认的 16 或自定义的初始容量
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 找到具体的数组下标，如果此位置没有值，那么直接初始化一下 Node 并放置在这个位置就可以了
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);

    else {// 数组该位置有数据
        Node<K,V> e; K k;
        // 首先，判断该位置的第一个数据和我们要插入的数据，key 是不是"相等"，如果是，取出这个节点
        if (p.hash == hash &&
```

```
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 如果该节点是代表红黑树的节点，调用红黑树的插值方法，本文不展开说红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            // 到这里，说明数组该位置上是一个链表
            for (int binCount = 0; ; ++binCount) {
                // 插入到链表的最后面(Java7 是插入到链表的最前面)
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // TREEIFY_THRESHOLD 为 8，所以，如果新插入的值是链表中的第 9 个
                    // 会触发下面的 treeifyBin，也就是将链表转换为红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                // 如果在该链表中找到了"相等"的 key(== 或 equals)
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 此时 break，那么 e 为链表中[与要插入的新值的 key "相等"]的 node
                    break;
                p = e;
            }
        }
        // e!=null 说明存在旧值的key与要插入的key"相等"
        // 对于我们分析的put操作，下面这个 if 其实就是进行 "值覆盖"，然后返回旧值
        if (e != null) {
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    // 如果 HashMap 由于新插入这个值导致 size 已经超过了阈值，需要进行扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

和 Java7 稍微有点不一样的地方就是，Java7 是先扩容后插入新值的，Java8 先插值再扩容，不过这个不重要。

数组扩容

resize() 方法用于初始化数组或数组扩容，每次扩容后，容量为原来的 2 倍，并进行数据迁移。

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) { // 对应数组扩容
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 将数组大小扩大一倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            // 将阈值扩大一倍
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // 对应使用 new HashMap(int initialCapacity) 初始化后，第一
次 put 的时候
        newCap = oldThr;
    else {// 对应使用 new HashMap() 初始化后，第一次 put 的时候
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }

    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;

    // 用新的数组大小初始化新的数组
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab; // 如果是初始化数组，到这里就结束了，返回 newTab 即可

    if (oldTab != null) {
        // 开始遍历原数组，进行数据迁移。
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                // 如果该数组位置上只有单个元素，那就简单了，简单迁移这个元素就可以了
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                // 如果是红黑树，具体我们就不展开了
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else {
                    // 这块是处理链表的情况，
```

```
                    // 需要将此链表拆成两个链表，放到新的数组中，并且保留原来的先后顺序
                    // loHead、loTail 对应一条链表，hiHead、hiTail 对应另一条链表，代码还是比较简单的
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        // 第一条链表
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        // 第二条链表的新的位置是 j + oldCap，这个很好理解
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}
```

get 过程分析

相对于 put 来说，get 真的太简单了。

计算 key 的 hash 值，根据 hash 值找到对应数组下标: hash & (length-1)

判断数组该位置处的元素是否刚好就是我们要找的，如果不是，走第三步

判断该元素类型是否是 TreeNode，如果是，用红黑树的方法取数据，如果不是，走第四步

遍历链表，直到找到相等(==或equals)的 key

```
ublic V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 判断第一个节点是不是就是需要的
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            // 判断是否是红黑树
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);

            // 链表遍历
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
```

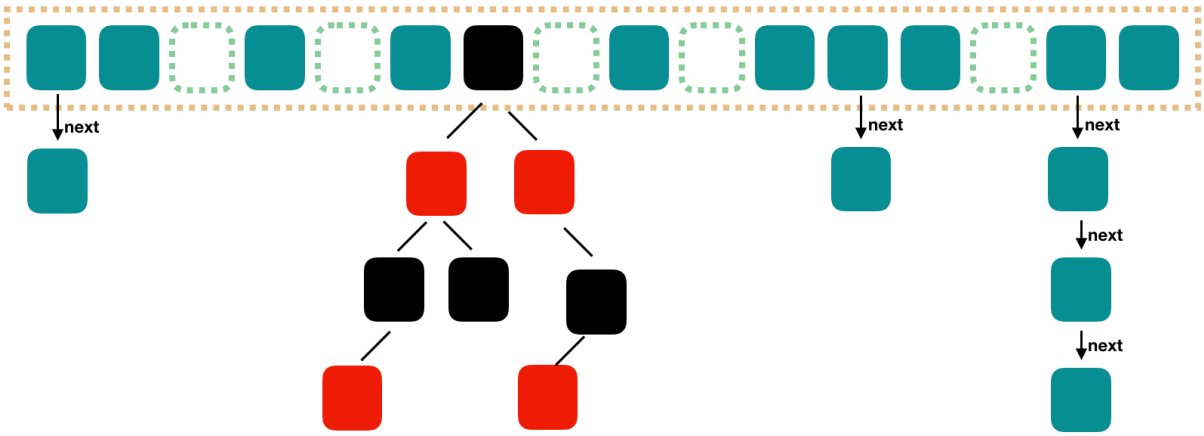
Java8 ConcurrentHashMap

Java7 中实现的 ConcurrentHashMap 说实话还是比较复杂的，Java8 对 ConcurrentHashMap 进行了比较大的改动。建议读者可以参考 Java8 中 HashMap 相对于 Java7 HashMap 的改动，对于 ConcurrentHashMap，Java8 也引入了红黑树。

说实话，Java8 ConcurrentHashMap 源码真心不简单，最难的在于扩容，数据迁移操作不容易看懂。

我们先用一个示意图来描述下其结构：

Java8 ConcurrentHashMap 结构



结构上和 Java8 的 HashMap 基本上一样，不过它要保证线程安全性，所以在源码上确实要复杂一些。

```
public ConcurrentHashMap() {
}
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

这个初始化方法有点意思，通过提供初始容量，计算了 sizeCtl，sizeCtl = 【(1.5 * initialCapacity + 1)，然后向上取最近的 2 的 n 次方】。如 initialCapacity 为 10，那么得到 sizeCtl 为 16，如果 initialCapacity 为 11，得到 sizeCtl 为 32。

sizeCtl 这个属性使用的场景很多，不过只要跟着文章的思路来，就不会被它搞晕了。

如果你爱折腾，也可以看下另一个有三个参数的构造方法，这里我就不说了，大部分时候，我们会使用无参构造函数进行实例化，我们也按照这个思路来进行源码分析吧。

put 过程分析

仔细地一行一行代码看下去：

```
public V put(K key, V value) {
    return putVal(key, value, false);
}
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 得到 hash 值
    int hash = spread(key.hashCode());
    // 用于记录相应链表的长度
```

```
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果数组"空"，进行数组初始化
        if (tab == null || (n = tab.length) == 0)
            // 初始化数组，后面会详细介绍
            tab = initTable();

        // 找该 hash 值对应的数组下标，得到第一个节点 f
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果数组该位置为空，
            //    用一次 CAS 操作将这个新值放入其中即可，这个 put 操作差不多就结束了，可以拉到最后面了
            //          如果 CAS 失败，那就是有并发操作，进到下一个循环就好了
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // hash 居然可以等于 MOVED，这个需要到后面才能看明白，不过从名字上也能猜到，肯定是因为在扩容
        else if ((fh = f.hash) == MOVED)
            // 帮助数据迁移，这个等到看完数据迁移部分的介绍后，再理解这个就很简单了
            tab = helpTransfer(tab, f);

        else { // 到这里就是说，f 是该位置的头结点，而且不为空

            V oldVal = null;
            // 获取数组该位置的头结点的监视器锁
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) { // 头结点的 hash 值大于 0，说明是链表
                        // 用于累加，记录链表的长度
                        binCount = 1;
                        // 遍历链表
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 如果发现了"相等"的 key，判断是否要进行值覆盖，然后也就可以 break 了
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            // 到了链表的最末端，将这个新值放到链表的最后面
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
```

```
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    else if (f instanceof TreeBin) { // 红黑树
                        Node<K,V> p;
                        binCount = 2;
                        // 调用红黑树的插值方法插入新节点
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            // binCount != 0 说明上面在做链表操作
            if (binCount != 0) {
                // 判断是否要将链表转换为红黑树，临界值和 HashMap 一样，也是 8
                if (binCount >= TREEIFY_THRESHOLD)
                    // 这个方法和 HashMap 中稍微有一点点不同，那就是它不是一定会进行红
黑树转换，
                    // 如果当前数组的长度小于 64，那么会选择进行数组扩容，而不是转换为红
黑树
                    //    具体源码我们就不看了，扩容部分后面说
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    //
    addCount(1L, binCount);
    return null;
}
```

put 的主流程看完了，但是至少留下了几个问题，第一个是初始化，第二个是扩容，第三个是帮助数据迁移，这些我们都会在后面进行一一介绍。

初始化数组：initTable

这个比较简单，主要就是初始化一个合适大小的数组，然后会设置 sizeCtl。

初始化方法中的并发问题是通过对 sizeCtl 进行一个 CAS 操作来控制的。

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        // 初始化的"功劳"被其他线程"抢去"了
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        // CAS 一下，将 sizeCtl 设置为 -1，代表抢到了锁
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    // DEFAULT_CAPACITY 默认初始容量是 16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    // 初始化数组，长度为 16 或初始化时提供的长度
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // 将这个数组赋值给 table，table 是 volatile 的
                    table = tab = nt;
                    // 如果 n 为 16 的话，那么这里 sc = 12
                    // 其实就是 0.75 * n
                    sc = n - (n >>> 2);
                }
            } finally {
                // 设置 sizeCtl 为 sc，我们就当是 12 吧
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

链表转红黑树: treeifyBin

前面我们在 put 源码分析也说过，treeifyBin 不一定就会进行红黑树转换，也可能是仅仅做数组扩容。我们还是进行源码分析吧。

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        // MIN_TREEIFY_CAPACITY 为 64
        // 所以，如果数组长度小于 64 的时候，其实也就是 32 或者 16 或者更小的时候，会进行数组扩容
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            // 后面我们再详细分析这个方法
            tryPresize(n << 1);
        // b 是头结点
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            // 加锁
            synchronized (b) {
```

```
                if (tabAt(tab, index) == b) {
                    // 下面就是遍历链表，建立一颗红黑树
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    // 将红黑树设置到数组相应位置中
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

扩容：tryPresize

如果说 Java8 ConcurrentHashMap 的源码不简单，那么说的就是扩容操作和迁移操作。

这个方法要完完全全看懂还需要看之后的 transfer 方法，读者应该提前知道这点。

这里的扩容也是做翻倍扩容的，扩容后数组容量为原来的 2 倍。

```
// 首先要说明的是，方法参数 size 传进来的时候就已经翻了倍了
private final void tryPresize(int size) {
    // c：size 的 1.5 倍，再加 1，再往上取最近的 2 的 n 次方。
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;

        // 这个 if 分支和之前说的初始化数组的代码基本上是一样的，在这里，我们可以不用管这
块代码
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2); // 0.75 * n
```

```
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        else if (tab == table) {
            // 我没看懂 rs 的真正含义是什么，不过也关系不大
            int rs = resizeStamp(n);

            if (sc < 0) {
                Node<K,V>[] nt;
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 2. 用 CAS 将 sizeCtl 加 1，然后执行 transfer 方法
                //    此时 nextTab 不为 null
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 1. 将 sizeCtl 设置为 (rs << RESIZE_STAMP_SHIFT) + 2)
            //     我是没看懂这个值真正的意义是什么？不过可以计算出来的是，结果是一个比
较大的负数
            //  调用 transfer 方法，此时 nextTab 参数为 null
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
        }
    }
}
```

这个方法的核心在于 sizeCtl 值的操作，首先将其设置为一个负数，然后执行 transfer(tab, null)，再下一个循环将 sizeCtl 加 1，并执行 transfer(tab, nt)，之后可能是继续 sizeCtl 加 1，并执行 transfer(tab, nt)。

所以，可能的操作就是执行 1 次 transfer(tab, null) + 多次 transfer(tab, nt)，这里怎么结束循环的需要看完 transfer 源码才清楚。

数据迁移：transfer

下面这个方法很点长，将原来的 tab 数组的元素迁移到新的 nextTab 数组中。

虽然我们之前说的 tryPresize 方法中多次调用 transfer 不涉及多线程，但是这个 transfer 方法可以在其他地方被调用，典型地，我们之前在说 put 方法的时候就说过了，请往上看 put 方法，是不是有个地方调用了 helpTransfer 方法，helpTransfer 方法会调用 transfer 方法的。

此方法支持多线程执行，外围调用此方法的时候，会保证第一个发起数据迁移的线程，nextTab 参数为 null，之后再调用此方法的时候，nextTab 不会为 null。

阅读源码之前，先要理解并发操作的机制。原数组长度为 n，所以我们有 n 个迁移任务，让每个线程每次负责一个小任务是最简单的，每做完一个任务再检测是否有其他没做完的任务，帮助迁移就可以了，而 Doug Lea 使用了一个 stride，简单理解就是步长，每个线程每次负责迁移其中的一部分，如每次迁移 16 个小任务。所以，我们就需要一个全局的调度者来安排哪个线程执行哪几个任务，这个就是属性 transferIndex 的作用。

第一个发起数据迁移的线程会将 transferIndex 指向原数组最后的位置，然后从后往前的 stride 个任务属于第一个线程，然后将 transferIndex 指向新的位置，再往前的 stride 个任务属于第二个线程，依此类推。当然，这里说的第二个线程不是真的一定指代了第二个线程，也可以是同一个线程，这个读者应该能理解吧。其实就是将一个大的迁移任务分为了一个个任务包。

```
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;

    // stride 在单核下直接等于 n，多核模式下为 (n>>>3)/NCPU，最小值是 16
    // stride 可以理解为”步长“，有 n 个位置是需要进行迁移的，
    //   将这 n 个任务分为多个任务包，每个任务包有 stride 个任务
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range

    // 如果 nextTab 为 null，先进行一次初始化
    //    前面我们说了，外围会保证第一个发起迁移的线程调用此方法时，参数 nextTab 为 null
    //       之后参与迁移的线程调用此方法时，nextTab 不会为 null
    if (nextTab == null) {
        try {
            // 容量翻倍
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        // nextTable 是 ConcurrentHashMap 中的属性
        nextTable = nextTab;
        // transferIndex 也是 ConcurrentHashMap 的属性，用于控制迁移的位置
        transferIndex = n;
    }

    int nextn = nextTab.length;

    // ForwardingNode 翻译过来就是正在被迁移的 Node
    // 这个构造方法会生成一个Node，key、value 和 next 都为 null，关键是 hash 为 MOVED
    // 后面我们会看到，原数组中位置 i 处的节点完成迁移工作后，
    //    就会将位置 i 处设置为这个 ForwardingNode，用来告诉其他线程该位置已经处理过了
    //    所以它其实相当于是一个标志。
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);


    // advance 指的是做完了一个位置的迁移工作，可以准备做下一个位置的了
```

```
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab

    /*
     * 下面这个 for 循环，最难理解的在前面，而要看懂它们，应该先看懂后面的，然后再倒回来
看
     *
     */

    // i 是位置索引，bound 是边界，注意是从后往前
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;

        // 下面这个 while 真的是不好理解
        // advance 为 true 表示可以进行下一个位置的迁移了
        //   简单理解结局：i 指向了 transferIndex，bound 指向了 transferIndex-stride
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;

            // 将 transferIndex 值赋给 nextIndex
            // 这里 transferIndex 一旦小于等于 0，说明原数组的所有位置都有相应的线程去
处理了
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ?
                                   nextIndex - stride : 0))) {
                // 看括号中的代码，nextBound 是这次迁移任务的边界，注意，是从后往前
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            if (finishing) {
                // 所有的迁移操作已经完成
                nextTable = null;
                // 将新的 nextTab 赋值给 table 属性，完成迁移
                table = nextTab;
                // 重新计算 sizeCtl：n 是原数组长度，所以 sizeCtl 得出的值将是新数组长
度的 0.75 倍
                sizeCtl = (n << 1) - (n >>> 1);
                return;
```

```
            }

            // 之前我们说过，sizeCtl 在迁移前会设置为 (rs << RESIZE_STAMP_SHIFT) + 2
            // 然后，每有一个线程参与迁移就会将 sizeCtl 加 1，
            // 这里使用 CAS 操作对 sizeCtl 进行减 1，代表做完了属于自己的任务
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                // 任务结束，方法退出
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;

                // 到这里，说明 (sc - 2) == resizeStamp(n) << RESIZE_STAMP_SHIFT，
                // 也就是说，所有的迁移任务都做完了，也就会进入到上面的 if(finishing){}
分支了
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
        // 如果位置 i 处是空的，没有任何节点，那么放入刚刚初始化的 ForwardingNode ”空节
点“
        else if ((f = tabAt(tab, i)) == null)
            advance = casTabAt(tab, i, null, fwd);
        // 该位置处是一个 ForwardingNode，代表该位置已经迁移过了
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            // 对数组该位置处的结点加锁，开始处理数组该位置处的迁移工作
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    // 头结点的 hash 大于 0，说明是链表的 Node 节点
                    if (fh >= 0) {
                        // 下面这一块和 Java7 中的 ConcurrentHashMap 迁移是差不多的，
                        // 需要将链表一分为二，
                        //   找到原链表中的 lastRun，然后 lastRun 及其之后的节点是一起
进行迁移的
                        //   lastRun 之前的节点需要进行克隆，然后分到两个链表中
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
```

```
                    else {
                        hn = lastRun;
                        ln = null;
                    }
                    for (Node<K,V> p = f; p != lastRun; p = p.next) {
                        int ph = p.hash; K pk = p.key; V pv = p.val;
                        if ((ph & n) == 0)
                            ln = new Node<K,V>(ph, pk, pv, ln);
                        else
                            hn = new Node<K,V>(ph, pk, pv, hn);
                    }
                    // 其中的一个链表放在新数组的位置 i
                    setTabAt(nextTab, i, ln);
                    // 另一个链表放在新数组的位置 i+n
                    setTabAt(nextTab, i + n, hn);
                    // 将原数组该位置处设置为 fwd，代表该位置已经处理完毕，
                    //    其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移
了
                    setTabAt(tab, i, fwd);
                    // advance 设置为 true，代表该位置已经迁移完毕
                    advance = true;
                }
                else if (f instanceof TreeBin) {
                    // 红黑树的迁移
                    TreeBin<K,V> t = (TreeBin<K,V>)f;
                    TreeNode<K,V> lo = null, loTail = null;
                    TreeNode<K,V> hi = null, hiTail = null;
                    int lc = 0, hc = 0;
                    for (Node<K,V> e = t.first; e != null; e = e.next) {
                        int h = e.hash;
                        TreeNode<K,V> p = new TreeNode<K,V>
                            (h, e.key, e.val, null, null);
                        if ((h & n) == 0) {
                            if ((p.prev = loTail) == null)
                                lo = p;
                            else
                                loTail.next = p;
                            loTail = p;
                            ++lc;
                        }
                        else {
                            if ((p.prev = hiTail) == null)
                                hi = p;
                            else
                                hiTail.next = p;
                            hiTail = p;
                            ++hc;
                        }
                    }
```

```
                        // 如果一分为二后，节点数少于 8，那么将红黑树转换回链表
                        ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                            (hc != 0) ? new TreeBin<K,V>(lo) : t;
                        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                            (lc != 0) ? new TreeBin<K,V>(hi) : t;

                        // 将 ln 放置在新数组的位置 i
                        setTabAt(nextTab, i, ln);
                        // 将 hn 放置在新数组的位置 i+n
                        setTabAt(nextTab, i + n, hn);
                        // 将原数组该位置处设置为 fwd，代表该位置已经处理完毕，
                        //    其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移
了
                        setTabAt(tab, i, fwd);
                        // advance 设置为 true，代表该位置已经迁移完毕
                        advance = true;
                    }
                }
            }
        }
    }
}
```

说到底，transfer 这个方法并没有实现所有的迁移任务，每次调用这个方法只实现了 transferIndex 往前 stride 个位置的迁移工作，其他的需要由外围来控制。

这个时候，再回去仔细看 tryPresize 方法可能就会更加清晰一些了。

get 过程分析

get 方法从来都是最简单的，这里也不例外：

1. 计算 hash 值
2. 根据 hash 值找到数组对应位置: (n – 1) & h
3. 根据该位置处结点性质进行相应查找
1. 如果该位置为 null，那么直接返回 null 就可以了
2. 如果该位置处的节点刚好就是我们需要的，返回该节点的值即可
3. 如果该位置节点的 hash 值小于 0，说明正在扩容，或者是红黑树，后面我们再介绍 find 方法
4. 如果以上 3 条都不满足，那就是链表，进行遍历比对即可

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 判断头结点是否就是我们需要的节点
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
```

```
                return e.val;
        }
        // 如果头结点的 hash 小于 0，说明 正在扩容，或者该位置是红黑树
        else if (eh < 0)
            // 参考 ForwardingNode.find(int h, Object k) 和 TreeBin.find(int h,
Object k)
            return (p = e.find(h, key)) != null ? p.val : null;

        // 遍历链表
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

总的来说，HashMap 的底层数组长度总是2的n次方的原因有两个，即当 length=2^n 时：

不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，空间利用率较高，查询速度也较快；

h&(length - 1) 就相当于对length取模，而且在速度、效率上比直接取模要快得多，即二者是等价不等效的，这是HashMap在速度和效率上的一个优化。

LinkedHashMap

HashMap 的直接子类LinkedHashMap继承了HashMap的所用特性，并且还通过额外维护一个双向链表保持了有序性, 通过对比LinkedHashMap和HashMap的实现有助于更好的理解HashMap。

HashTable

Hashtable同样是基于哈希表实现的，同样每个元素是一个key-value对，其内部也是通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

Hashtable也是JDK1.0引入的类，是线程安全的，能用于多线程环境中。

Hashtable同样实现了Serializable接口，它支持序列化，实现了Cloneable接口，能被克隆

Hashtable和HashMap到底有哪些不同呢

（1）基类不同：HashTable基于Dictionary类，而HashMap是基于AbstractMap。Dictionary是什么？它是任何可将键映射到相应值的类的抽象父类，而AbstractMap是基于Map接口的骨干实现，它以最大限度地减少实现此接口所需的工作。

（2）null不同：HashMap可以允许存在一个为null的key和任意个为null的value，但是HashTable中的key和value都不允许为null。

（3）线程安全：HashMap时单线程安全的，Hashtable是多线程安全的。

（4）遍历不同：HashMap仅支持Iterator的遍历方式，Hashtable支持Iterator和Enumeration两种遍历方式。

JDK源码剖析之Copy-On-Write容器

简介

Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

什么是CopyOnWrite容器

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

CopyOnWriteArrayList的实现原理

首先来看CopyOnWriteList添加元素的方法

```
public boolean add(E e) {
        //可重入锁
        final ReentrantLock lock = this.lock;
        //在对容器写之前进行加锁
        lock.lock();
        try {
            //写之前拿到原来容器的所有元素
            Object[] elements = getArray();
            //写之前容器的长度
            int len = elements.length;
            //将原来的容器元素拷贝到新的数组中，并且长度加1
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            //将需要添加到元素的元素放在尾部
            newElements[len] = e;
            //将CopyOnWriteArrayList的内部引用指向新的数组
            setArray(newElements);
            return true;
        } finally {
            //释放锁
            lock.unlock();
        }
    }

//setArray()方法内部源码
    final void setArray(Object[] a) {
        //CopyOnWriteArrayList的内部引用指向新的数组
        array = a;
```

```
    }
```

这里可以看出在进行添加元素之前，也就是在写容器之前需要加锁，这样避免多线程时复制多个数组；写容器的过程是将原数组的元素拷贝一份到新的数组，同时新的数组是原来的数组的长度+1，然后在将需要添加的元素添加到新数组中，最后将CopyOnWriteArrayList的内部引用指向新的数组。

CopyOnWriteArrayList写方法

```
    public E get(int index) {
         return get(getArray(), index);
     }
```

从上面的代码看出，获取元素的时候，没有加锁，可见这个操作是非常高效的；同时这个读取操作是在原数组的基础上得到的元素，这样会读到的旧的数据。

CopyOnWrite的应用场景

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。

CopyOnWrite的缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

内存占用问题。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

JVM深入理解篇

Javac原理剖析

Javac是什么?

Javac是一种编译器，能将一种语言规范转化为另一种语言规范。Javac的任务就是将Java源代码语言转化成JVM能够识别的一种语言，然后由JVM将JVM语言转化成当前这个机器能够识别的机器语言。

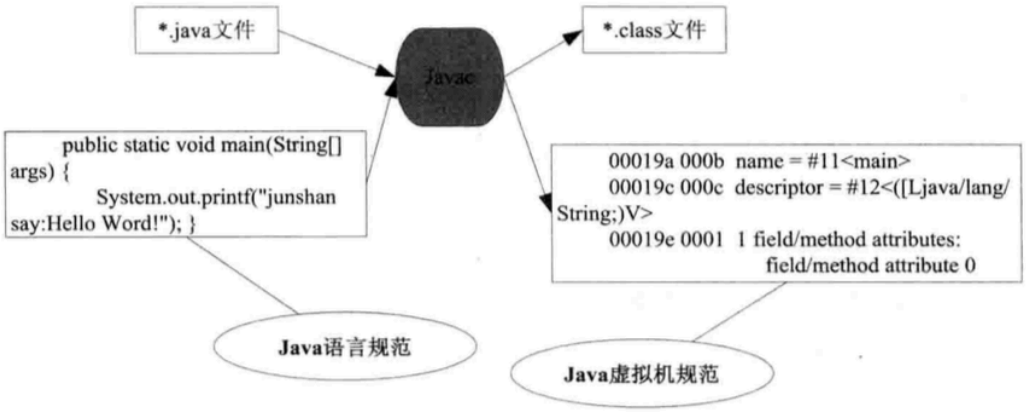


图 4-1 Javac 功能

Javac的任务就是将Java源码成Java字节码，也就是JVM能够识别二进制码。从表面上看就是上面的部分将.java文件转成.class文件，而实际上是将Java的源代码转化成一连串二进制数字，这些二进制数字是有格式的，只有JVM能够正确识别他们到底表达了什么意思。

Javac编译器的基本结构

Javac的作用是将符合规范源代码转化成为字节码，需要哪些过程呢？我们可以复习一下大学时候的编译原理知识。

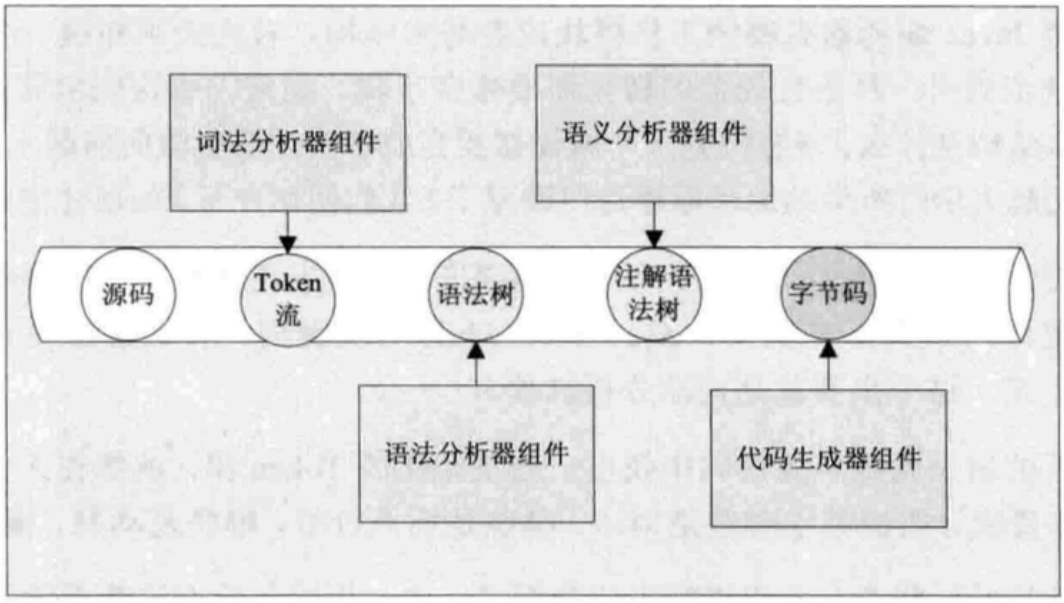


图 4-2 Javac 组件

如何编译程序呢?

1.词法分析:主要是读取源代码，一个字节为一节地读取进来，找些那些是语言的关键字例如，if、else等。词法分析就是从源代码中找出一些规范化的Token流，一个形象的比如就是人类能分清楚一个句子的单词和标点符号。

2.语法分析:检查这些关键词组合在一起是不是符合Java规范，如在if后面是不是跟着布尔表达式，就像人类能分清楚一个句子的主语谓语宾语，检查是否符合语法。语法分析的结果就是形成一个Java语言规范的抽象语法树。

3.语义分析:将一些复杂难懂的语法转化为简单易懂的语法。这个过程对应于将复杂难懂的文言文转化为白话文。语义分析的结果是将复杂难懂的语法转化为最简单的语法，对应于Java，可以将注解等转化为抽象语法树。

最后是代码生成将抽象语法树生成字节码，可以对应于人类的将中文转化为英文组合成新的句子。

Javac原理分析

词法分析器

我们可以通过一个简单的Java类来进行词法分析。

```
package compile;

public class Cifa {
    int a;
    int b=a;
    int c=a+1;

}
```

我们可以调用com.sun.tools.javac.main.Main类来实现手动编译指定的类。

```
public static int compile(String[] args) {
        com.sun.tools.javac.main.Main compiler =
            new com.sun.tools.javac.main.Main("javac");
        return compiler.compile(args);
    }
```

Javac的主要词法分析器的结构类是com.sun.tools.javac.parser.Lexer,这个默认实现类是com.sun.tools.javac.parser.Scanner，这个类会逐步读取Java源程序的单个字符，然后解析出符合Java语言规范的Token序列。所设计的类如下图

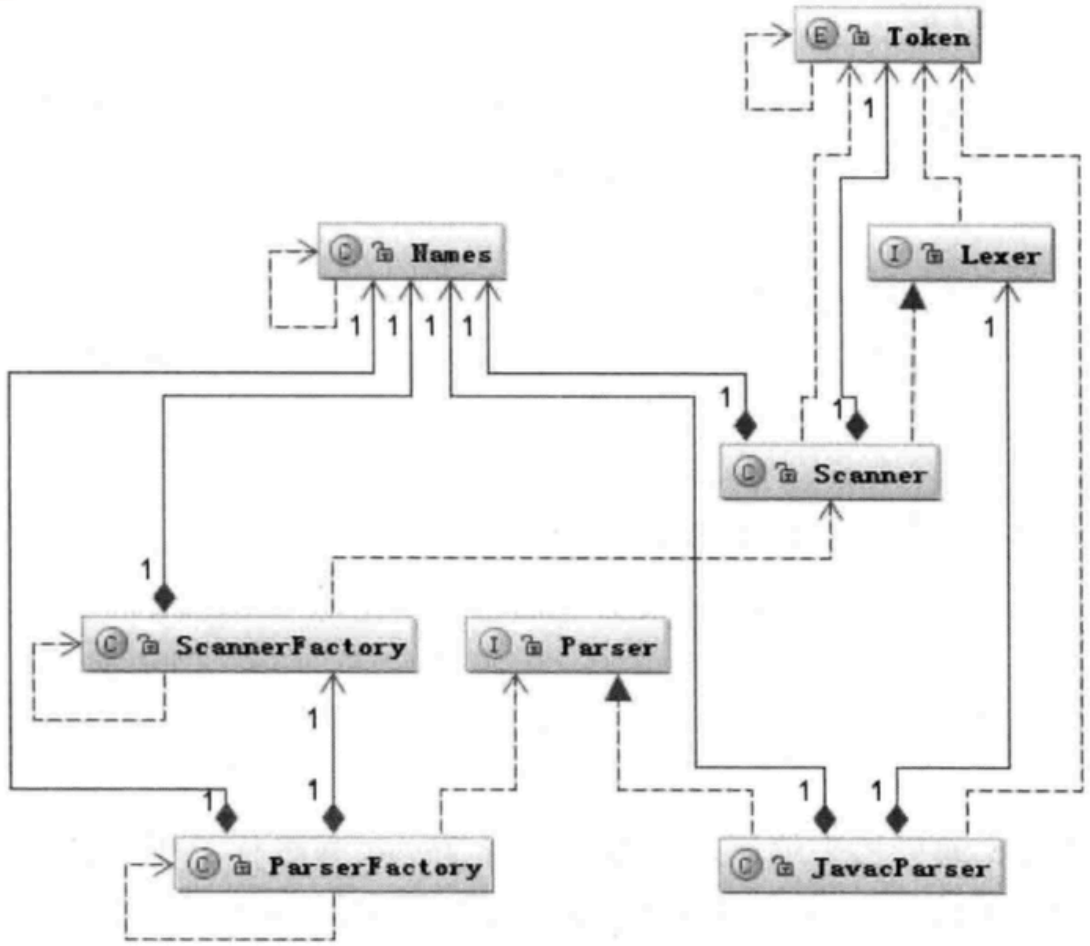


图 4-3　词法分析器设计的类图

这两个Factory生成两个接口类的Scanner和JavacParser，JavacParser规定哪些词是符合Java语言规范规定，而具体读取和归类不同的词法操作由Scanner完成，Token规定了完成Java语言的合法关健词，Names用来存储和表示解析后的词法。

词法分析额过程是在JavacParser的parseCompilationUnit()方法中完成，主要代码如下:

```
public JCTree.JCCompilationUnit parseCompilationUnit() {
        int pos = S.pos();
        JCExpression pid = null;
        String dc = S.docComment();
        JCModifiers mods = null;
        List<JCAnnotation> packageAnnotations = List.nil();
        if (S.token() == MONKEYS_AT)
            mods = modifiersOpt();//解析修饰符
        if (S.token() == PACKAGE) {//解析package声明
            if (mods != null) {
                checkNoMods(mods.flags);
                packageAnnotations = mods.annotations;
                mods = null;
```

```
                }
                S.nextToken();
                pid = qualident();
                accept(SEMI);
            }
            ListBuffer<JCTree> defs = new ListBuffer<JCTree>();
            boolean checkForImports = true;
            while (S.token() != EOF) {
                if (S.pos() <= errorEndPos) {
                    //跳过错误字符
                    skip(checkForImports, false, false, false);
                    if (S.token() == EOF)
                        break;
                }
                if (checkForImports && mods == null && S.token() == IMPORT) {
                    defs.append(importDeclaration());//解析 import 声明
                } else {//解析 class 类主体
                    JCTree def = typeDeclaration(mods);
                    if (keepDocComments && dc != null && docComments.get(def)
== dc) {
                        //如果在前面的类型声明中已经解析过了，那么在 top level 中将不
                        //再重复解析
                        dc = null;
                    }
                    if (def instanceof JCExpressionStatement)
                        def = ((JCExpressionStatement)def).expr;
                    defs.append(def);
                    if (def instanceof JCClassDecl)
                        checkForImports = false;
                    mods = null;
                }
            }
            JCTree.JCCompilationUnit toplevel = F.at(pos).TopLevel
(packageAnnotations, pid, defs.toList());
            attach(toplevel, dc);
            if (defs.elems.isEmpty())
                storeEnd(toplevel, S.prevEndPos());
            if (keepDocComments)
                toplevel.docComments = docComments;
            if (keepLineMap)
                toplevel.lineMap = S.getLineMap();
            return toplevel;
        }
```

这段代码表示从源文件的一个字符开始，按照Java语法规范依次出现packege、import、类定义，以及属性和方法定义等，最后构建一个抽象的语法树。

词法分析器的分析结构就是将这个列中的所有关键词匹配到Token类中的所有项中的任何一项。上述代码分析的Token流是:

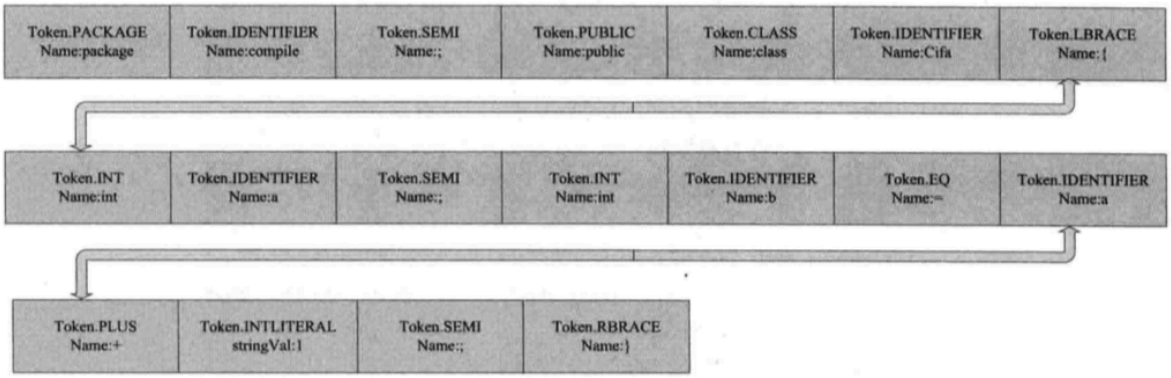


图 4-4　Token 流

这里有两个关键点是:Javac是如何分辨这一个个Token的呢？还有就是Javac如何知道啊哪些字符组合在一起就是一个Token的呢?

第一个问题的答案是这样的:Javac在进行词法分析时会由javacParser根据Java语言规范来控制什么顺序、什么地方应该出现什么Token。

判断当前的Token是不是Token.PACKAGE，使用qualident方法。这个方法的源代码是:

```
public JCExpression qualident() {
        JCExpression t = toP(F.at(S.pos()).Ident(ident()));
        while (S.token() == DOT) {
            int pos = S.pos();
            S.nextToken();
            t = toP(F.at(pos).Select(t, ident()));
        }
        return t;
    }
```

上述代码的执行流程如下:

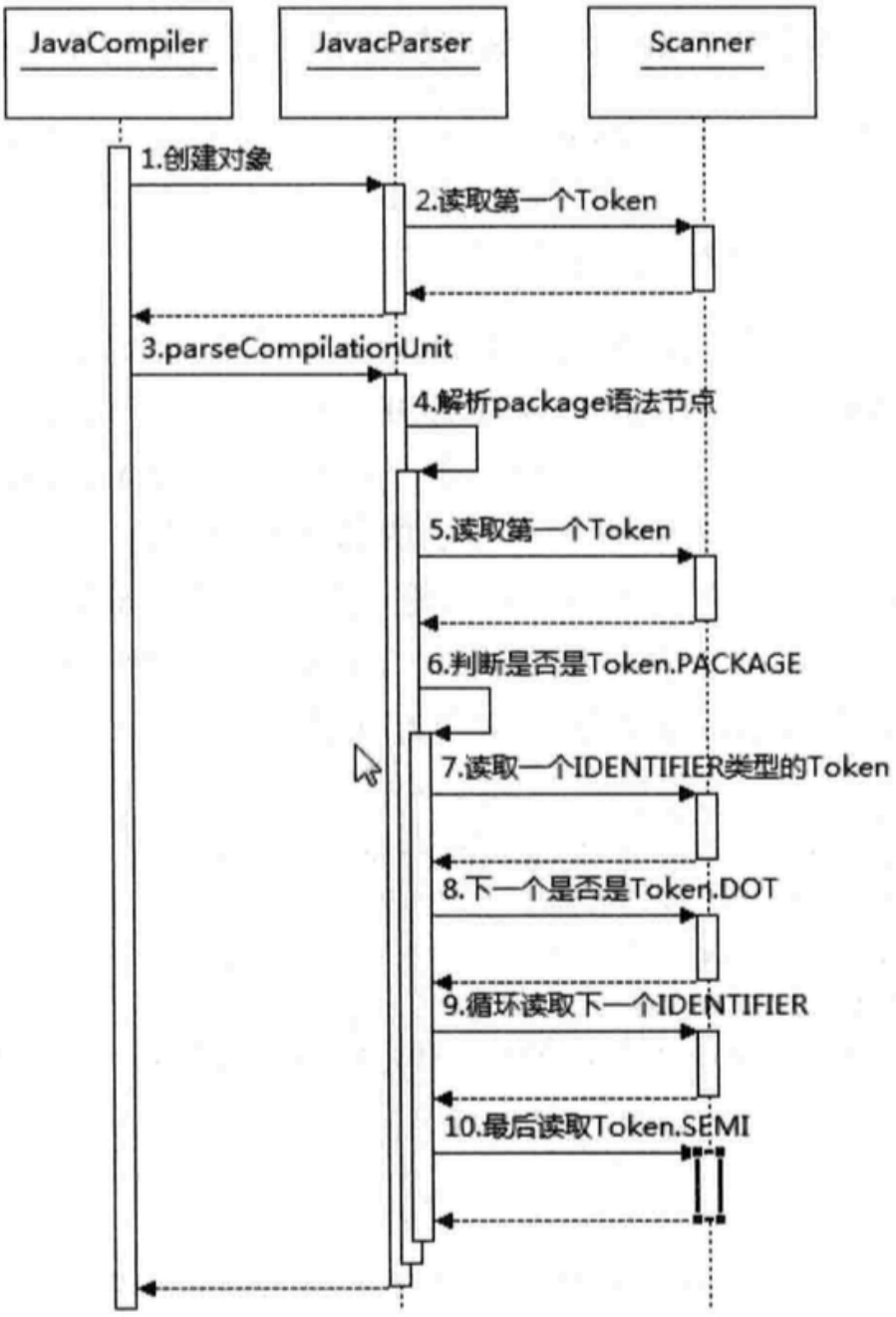


图 4-5　Package 的读取规则

通过上图，我们明白了Token的顺序规则，之后我们来解决下一个问题，如何判断哪些字符组合是一个Token 的规则是在Scanner的nextToken方法中定义，没调用一次这个方法会构成一个Token，而这个Token必须是com.sun.tools.javac.parser.Token中的任何元素之一。

那么如何将读取每个token转化呢？这个任务是在com.sun.tools.javac.parser.Keywords类中完成，Keywords负责将所有字符集合对应到Token集合中。

字符集合到Token转换相关的类关系如下图

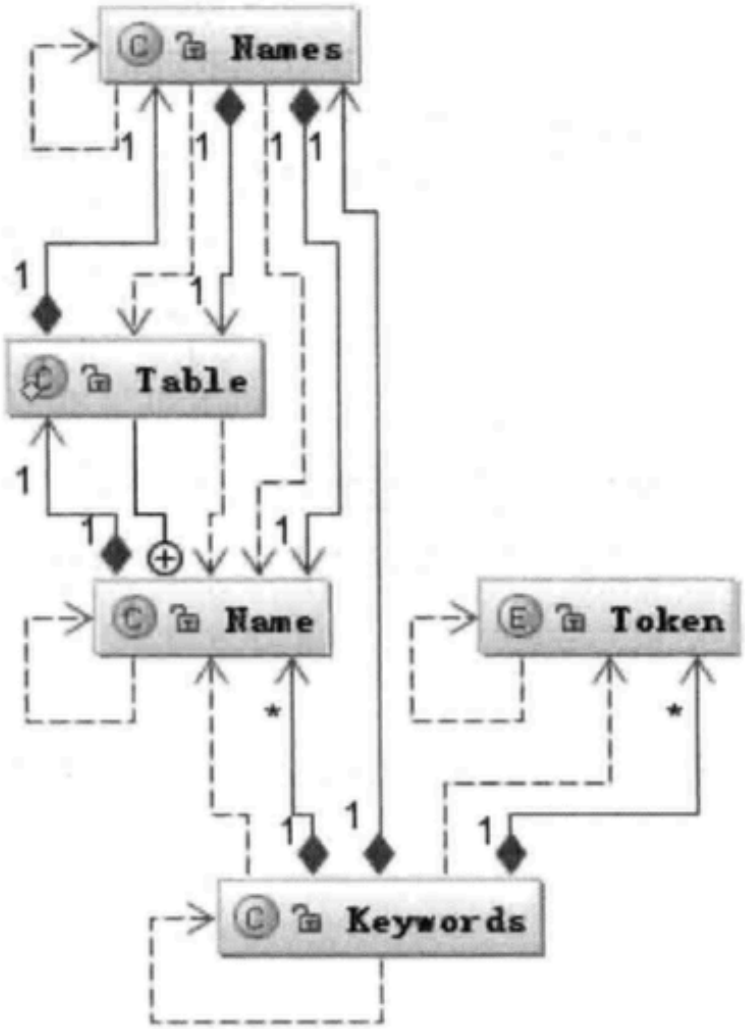


图 4-6　相关的类关系图

每个字符集合都会是一个Name对象，所有的Name对象都存储在Name.Table这个内部类中，这个类也就是对应的这类的符号表。而Keywords会将Token的对应关系，这个关系保存在Keywords类的key数组中，这个key数组只保存了在com.sun.tools.javac.parser.Token类中定义的所有Token到Name对象的关系，而其他的所有字符集合Keywords都会将它对应到Token.IDENTIFIER类型。

字符集合转化成Name对象，Name对象对应啊到Token的转换关系如下图;

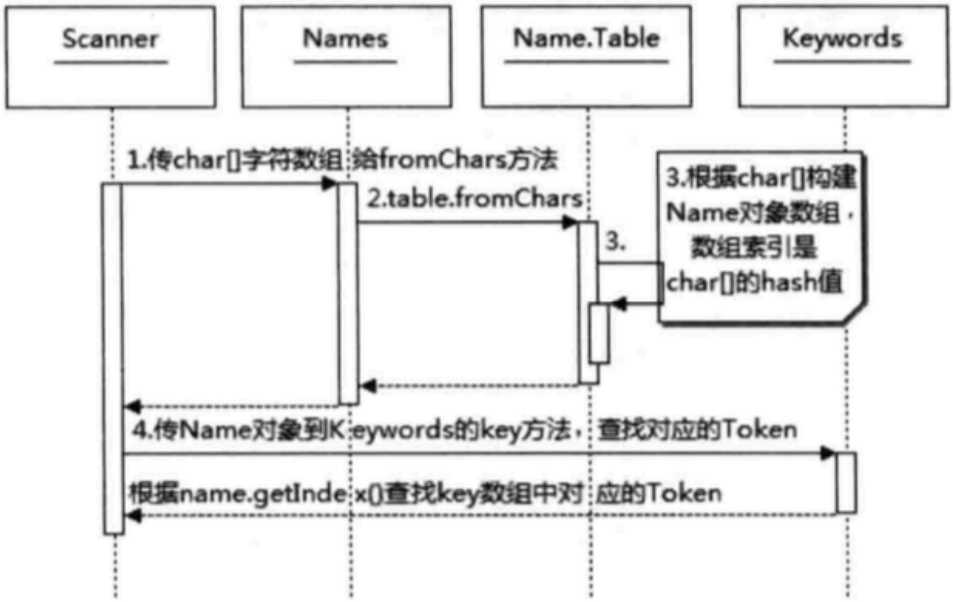


图 4-7　Name 与 Token 的对应关系

语法分析器

语法分析器是将词法分析器分析的Token流建成更加结构化的语法树，也就是将一个个单词组装成一句话，一个完整的语句。

Javac的语法树使得Java源码更加结构化，这种结构化可以为后面进一步处理提供方便。每个语法书上的节点都是com.sun.tools.javac.tree.JCTree的一个实例。

关于语法规则是:

1.每一个语法节点都会实现一个接口xxxTree,这个接口继承自com.sun.source.tree.Tree接口。

2.每个语法节点都是com.sun.tools.javac.tree.JCTree的子类，并且会实现第一节点中的xxxTree接口类，这个类的名称类似于JCxxx。

3.所有的JCxxx类都作为一个静态内部类定义在JCTree类中。

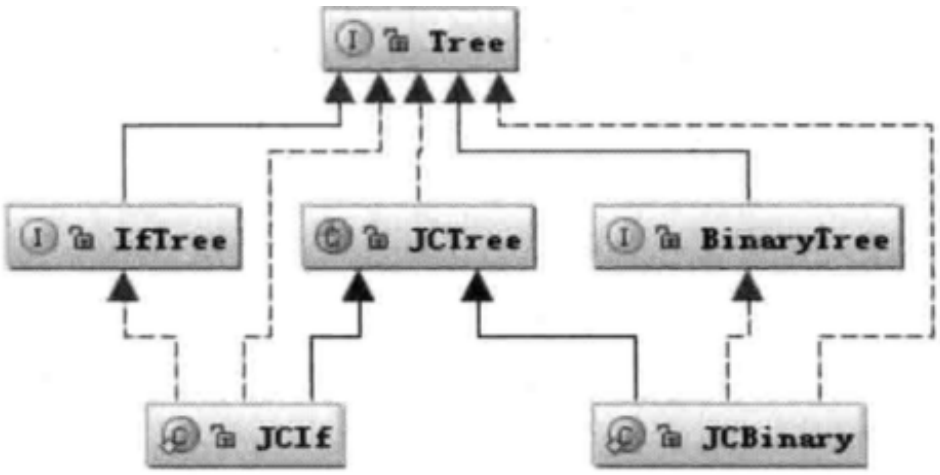


图 4-8　JCxxx 与 xxxTree 的类关系图

JCTree类中有如下三个重要的属性项:

1.Tree tag:每个语法节点都会用一个整形常数表示，并且每个节点类型的数值是在其哪一个的基础上加1，顶层节点TOPLEVEL是1，而IMPORT节点等于TOPLEVEL加1，等于2.

2.pos:也是一个整数,它存储的是这个语法节点在源代码中的起始位置，一个文件的位置是0，而-1表示不存在。

3.type:它表示的是这个节点是什么Java类型，如是int、float还是String。

在package的词法分析的过程是

```
JCExpression t = toP(F.at(S.pos()).Ident(ident()));
```

这行代码会调用TreeMaker类，根据Name对象构建成一个JCIdent语法节点，如果多几目录，将构建成JCFieldAccess语法节点，JCFieldAccess节点可以是嵌套关系。

下面是import语法树的构造代码:

```
JCTree importDeclaration() {
        int pos = S.pos();
        S.nextToken();
        boolean importStatic = false;
        if (S.token() == STATIC) {
            checkStaticImports();
            importStatic = true;
            S.nextToken();
        }
        JCExpression pid = toP(F.at(S.pos()).Ident(ident()));
        do {
            int pos1 = S.pos();
            accept(DOT);
            if (S.token() == STAR) {
                pid = to(F.at(pos1).Select(pid, names.asterisk));
                S.nextToken();
                break;
            } else {
                pid = toP(F.at(pos1).Select(pid, ident()));
            }
        } while (S.token() == DOT);
        accept(SEMI);
        return toP(F.at(pos).Import(pid, importStatic));
    }
```

整个JCImport节点的语法树如下:

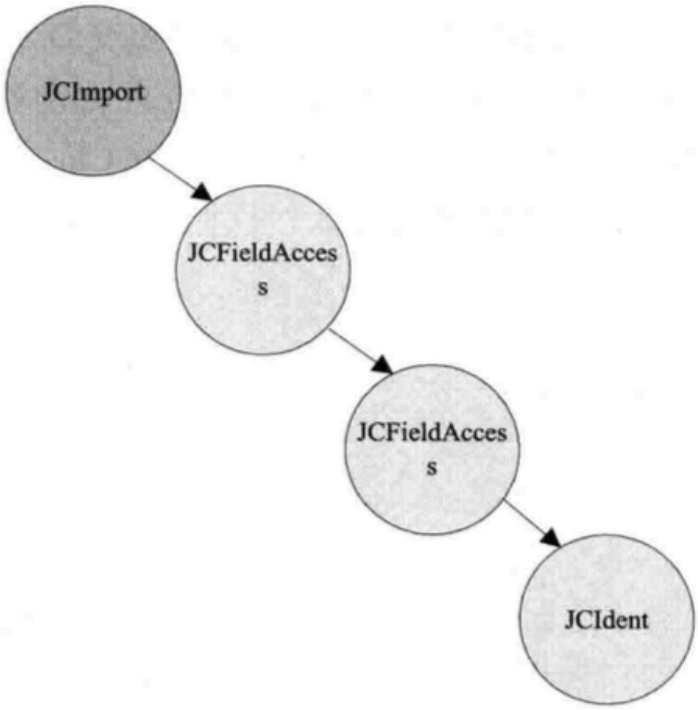


图 4-9　JCImport 语法树

Import节点解析完成之后就是类的鸡西，类包含interface、class、enum，下面是以calss为例来介绍class是如何解析成一颗语法树的。

```
JCClassDecl classDeclaration(JCModifiers mods, String dc) {
        int pos = S.pos();
        accept(CLASS);
        Name name = ident();
```

```
        List<JCTypeParameter> typarams = typeParametersOpt();

        JCTree extending = null;
        if (S.token() == EXTENDS) {
            S.nextToken();
            extending = parseType();
        }
        List<JCExpression> implementing = List.nil();
        if (S.token() == IMPLEMENTS) {
            S.nextToken();
            implementing = typeList();
        }
        List<JCTree> defs = classOrInterfaceBody(name, false);
        JCClassDecl result = toP(F.at(pos).ClassDef(
            mods, name, typarams, extending, implementing, defs));
        attach(result, dc);
        return result;
    }
```

第一个Token是Token.CLASS这个累的关键词，接下来是一个用户自定义的Token.IDENTIFIER，这个Token是类名。

最后解析整个classBody解析的结果保存在list集合中，最后将会把这些子节点添加到JCClassDeca这颗class树中。

下面的代码为例;

```
public class Yufa {
    int a;
    private int c = a + 1;

    public int getC() {
        return c;
    }

    public void setC(int c) {
        this.c = c;
    }
}
```

这段代码对应的语法树如下:

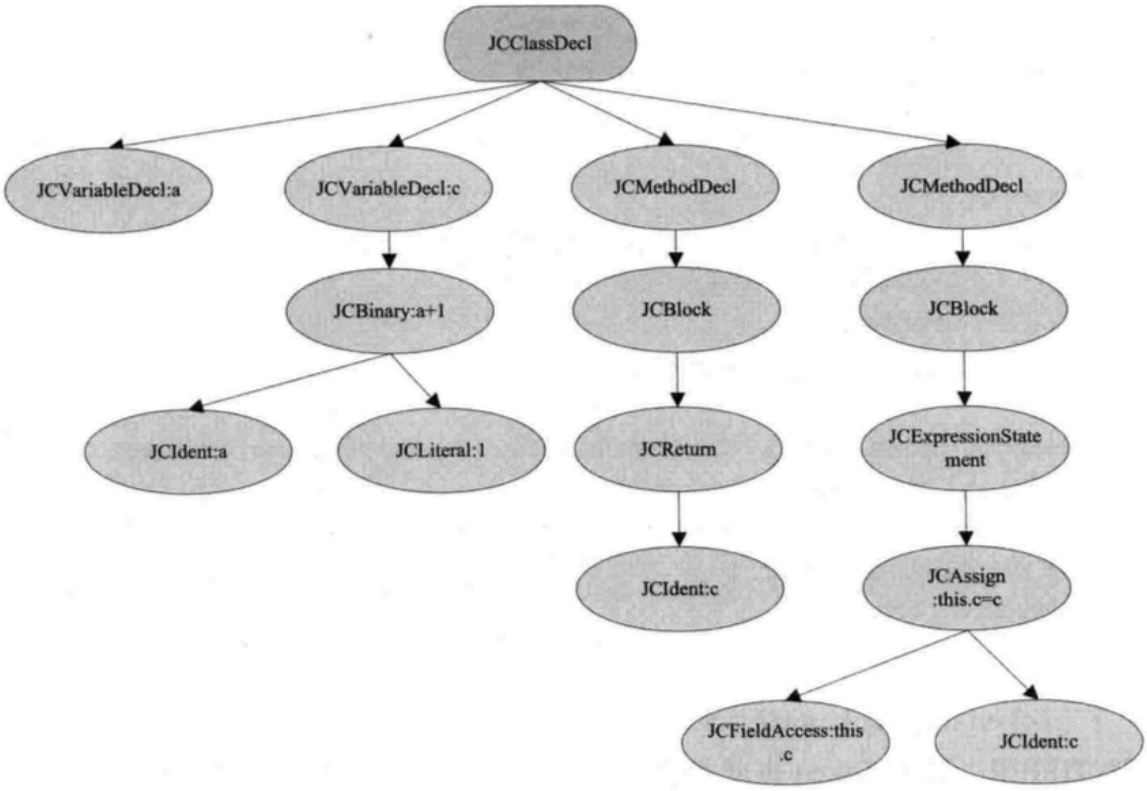


图 4-10　Yufa 类对应的语法树

上面的语法树去掉了一些节点类型。在将这个类解析完成之后，会将这个类节点添加到这个类对应的包路径的顶层节点中，这个顶层节点是JCCompilationUnit.JCCompilationUnit持有以package作为pid和JCClassDec1的集合，这个整个java文件被解析完成，这颗完整的语法树如下:

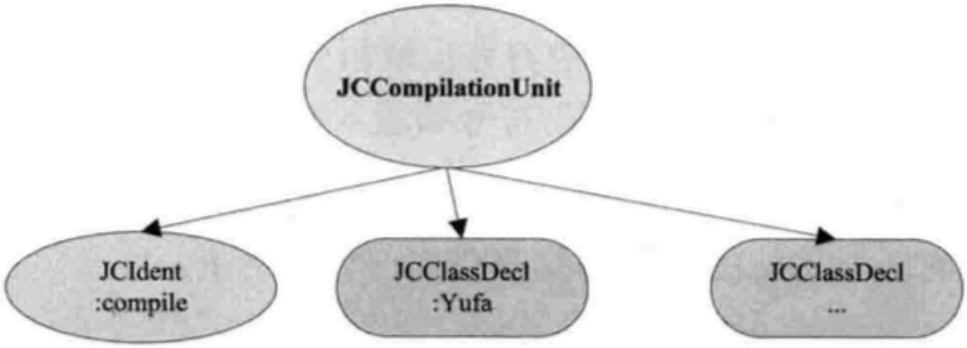


图 4-11　Yufa.java 对应的语法树

关于语法分析的一点需要说明的是，所有语法节点的生成树都是在TreeMaker类中完成的，TreeMaker实现了在JCTree.Factory接口中定义的所有节点的构成方法。

语义分析器

在语法书的基础上进一步处理，例如给类添加默认的构造器函数，检查变量在使用前是否已经初始化，将一些常量进行合并处理，检查操作变量类型是否匹配，检查所有的操作数据是否可达，检查checked exception异常是否已经捕获或者破除，解除Java语法糖等。

将在Java类中的符号输入到符号表中主要由com.sun.tools.javac.comp.Enter类来完成，这个类主要完成以下两个步骤:

1.将所有类中出现的符号输入到类自身的符号表中，所有类符号、类的参数类型符号、超类符号和继承的接口类型符号等都存储到一个未处理的列表中。

2.将这个未处理列表中所有的类都解析到各自的类符号列表中，这个操作是在MemberEnter.complete()中完成。

在上面的Yufa类中，会添加一个无参构造函数。

接下来使用com.sun.tools.javac.comp.Attr检查语法的合法性进行逻辑判断:

1.变量的类型是否匹配

2.变量在使用前是狗已经初始化

3.能够推导出泛型方法的参数类型

4.字符串常量的合并

例如:

```
public class Yufa {
    int a=0;
    private int c = a + 1;
    private int d = 1 + 1;
    private String s = "hello" + "word";
}
```

经过解析后，源代码变成:

```
public class Yufa {

    public Yufa() {
        super();
    }
    int a = 0;
    private int c = a + 1;
    private int d = 1 + 1;
    private String s = "helloword";
}
```

这里将两个字符串合并成一个字符串。

在标注完成以后,由com.sun.tools.javac.comp.Flow完成数据流分析，主要完成的工作如下:

1.检查变量的使用前时候已经被正确赋值，除了Java的原始类型，其他像String类型和对象的引用都必须在使用前先赋值

2.保证final修饰的变量不会被重复复制。

3.要确定方法的返回值类型

4.所有的CheckedException都要捕获或者向上抛出。

5.所有的语句都要被执行到。

语义分析的足后一个步骤是执行com.sun.tools.javac.comp.Flow,这就是在进一步对语法树进行语义分析，如消除一些无用的代码；去除永不真的条件判断；解除一些语法糖，类型的自动转化操作。

一些例子如下:

类型转化:

```
public class Yuyi {
    public static void main(String[] args) {
        Integer i = 1;
        Long l = i + 2L;
        System.out.println(l);
    }
}
```

经过自动转化后的代码如下所示：

```
public class Yuyi {

    public Yuyi() {
        super();
    }

    public static void main(String[] args) {
        Integer i = Integer.valueOf(1);
        Long l = Long.valueOf(i.intValue() + 2L);
        System.out.println(l);
    }
}
```

在转化后将 int 的“1”转化成 Integer.valueOf 的形式，而 Long 的格式也类似，除自动将 int 和 Integer 进行转化外，还通过 intValue 方法将 Integer 转化为 int 的格式。

for循环解析:

```
public class Yuyi {
    public static void main(String[] args) {
        List<Integer> array = Arrays.asList(1, 2, 3);
        for (Integer i : array) {
            System.out.println(i);
        }
    }
}
```

这时转化成的 for 循环会被解析成如下格式：

```
public class Yuyi {

    public Yuyi() {
        super();
    }

    public static void main(String[] args) {
        List array = Arrays.asList(new Integer[]{Integer.valueOf(1), Integer
valueOf(2), Integer.valueOf(3)});
        for (.java.util.Iterator i$ = array.iterator(); i$.hasNext(); ) {
            Integer i = (Integer)i$.next();
            {
```

内部类:

```
public class Yuyi {
    public void main(String[] args) {
        Inner inner = new Inner();
        inner.print();
    }
   class Inner{

        public void print(){
            System.out.println("Yuyi$inner.print");
        }
    }
}
```

在这段代码中定义了一个内部类 Inner，在一个 main 方法里创建这个类的对象，并调用它的一个方法。转化后的代码如下所示：

```
public class Yuyi {

    public Yuyi() {
        super();
    }

    public void main(String[] args) {
        Yuyi$Inner inner = new Yuyi$Inner(this);
        inner.print();
    }
    {
    }
}
```

Inner 类名称被转化成 Yuyi$Inner，并将 this 对象传给了 Yuyi$Inner 类，而 Inner 类被一个空的代码块代替了，那么这个 Inner 类被转化成了什么样子呢？代码如下：

```
class Yuyi$Inner {
    /*synthetic*/ final Yuyi this$0;

    Yuyi$Inner(/*synthetic*/ final Yuyi this$0) {
        this.this$0 = this$0;
        super();
    }

    public void print() {
        System.out.println("Yuyi$inner.print");
    }
}
```

Inner 类被重命名为 Yuyi$inner 类，创建了一个以 Yuyi 类为参数的构造函数，并且会持有 Yuyi 类的一个对象的引用。

Java虚拟机运行时数据区域

JVM的内存划分

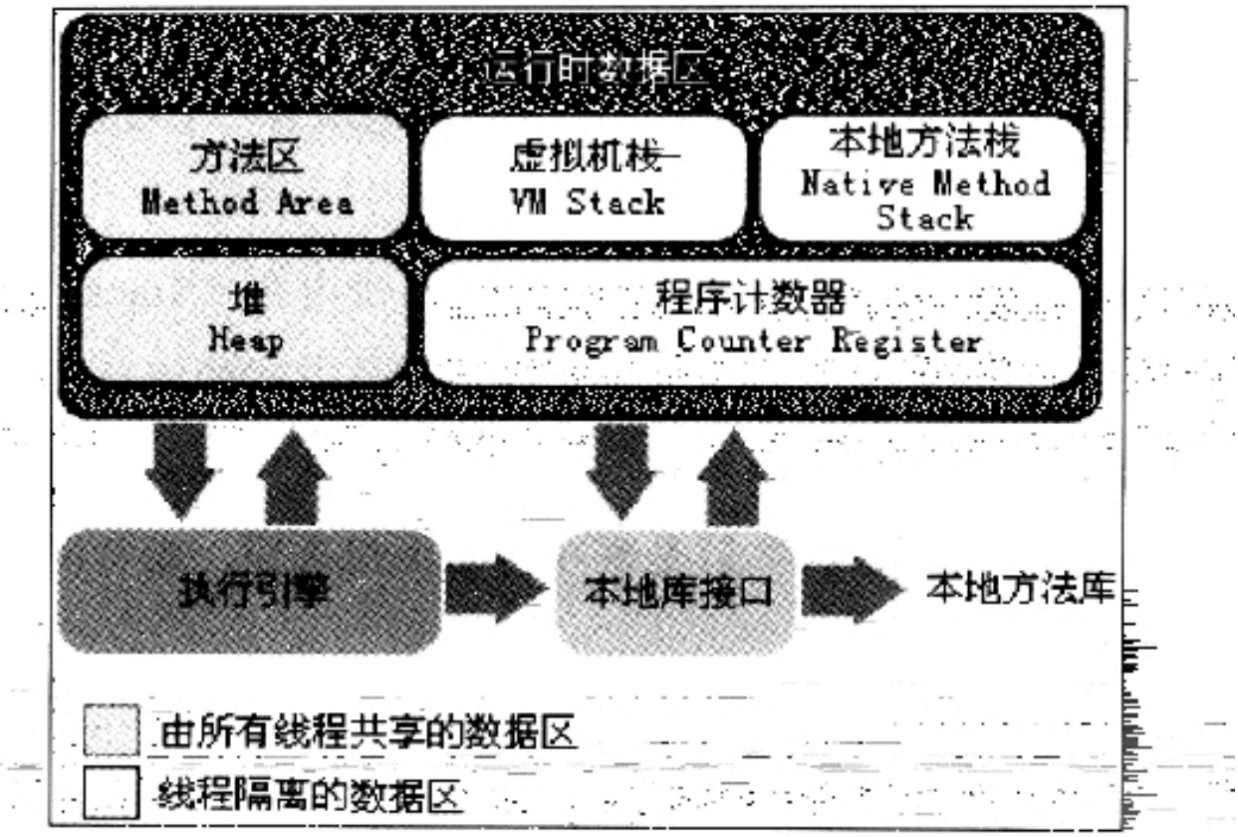


图 2-1　Java 虚拟机运行时数据区

程序计数器

当前线程所执行的字节码指示器，线程私有。

Java虚拟机栈

线程私有，存储局部变量，栈帧，操作数栈，动态链接，方法出口等信息。

本地方法栈

调用操作系统的类库

Java堆

线程共享，基本上所有的对象均在Java堆中，垃圾回收的主要区域

方法区

线程共享，存储类信息、常量、静态变量、及时编译后的代码数据

运行时常量池

方法区一部分，存储各种字面值和符号引用。

回收方法区

回收的内容：废弃常量和无用的类

判断一个常量废弃：没有引用指向这个常量

判断一个类无用：

1.实例对象全部回收

2.该类的加载器被回收

3.Class对象没有被任何地方引用

虚拟机对象探秘

对象的创建

创建过程

0.虚拟机遇到一条new指令时，首先将会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

1.在堆中分配一块内存区域，用于存放对象，

2.初始化对象中的各个数据变量和设置对象头，各个类型默认值

3.执行init方法，对于各个变量赋予初值

Java堆分配内存空间的方法：

1.指针碰撞：使用的内存和空闲内存用指针隔离开来

2.空闲列表：记录那些位置空闲

为了对象创建的线程，每个线程具有自己的本地线程缓冲区TLAB，当TLAB不够用时，需要再次请求分配TLAB需要同步。

对象的布局

对象的3个区域：

对象头、实例数据和对齐填充。

表 2-1　HotSpot 虚拟机对象头 Mark Word

存储内容	标 志 位	状　　态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

对象头是8字节的整数倍

实例数据：对象真正存储的有效数据

对齐填充：对象需要是8字节的整数倍

对象的访问定位

栈中含有对象引用

两种方式

1.句柄访问：堆中划分一部分空间作为句柄池，存储的句柄信息，大多数虚拟机采用

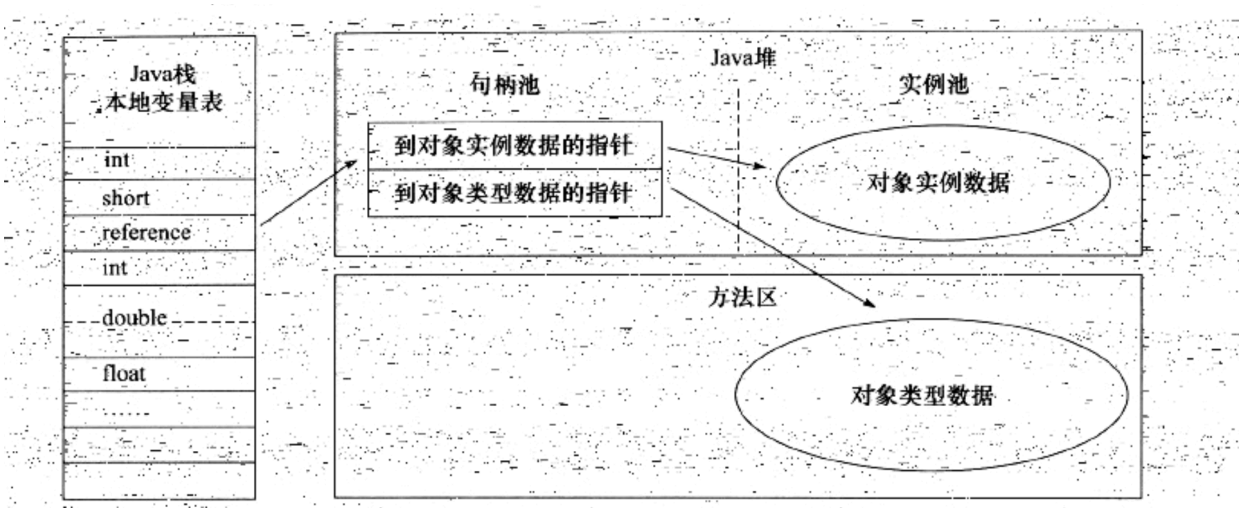


图 2-2　通过句柄访问对象

2.直接指针：直接对象的地址

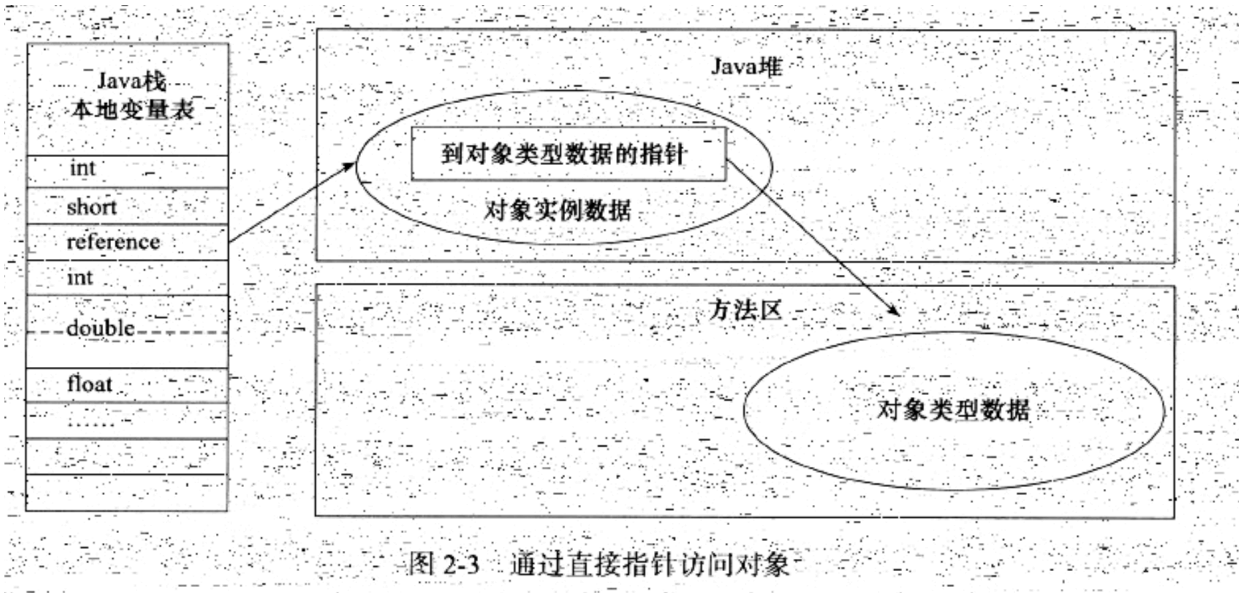


图 2-3　通过直接指针访问对象

句柄访问：不用改变句柄地址，引用不需要改变

内存溢出和内存泄漏

内存溢出：由于内存空间不足，导致的无法分配足够的内存

内存泄漏：分配过的内存无法回收，也无法访问

对象存活判断方法

引用指针算法

思想：又一个地方引用该对象时，那么计数器加1，引用失效时，计数器减1，计数器为0，对象可以被回收

缺点：两个对象内部变量互相引用时，可能永远不被回收

可达性分析算法

思想：通过一个GC Roots，从上向下搜索，走过的路径称为引用链，不在引用链中的对象可被回收。

可以作为GC Roots的对象：

虚拟机栈中的引用对象

类静态属性引用的对象

常量引用的对象

本地方法栈中的对象

引用类型

强引用(Strong Reference)：随处可见，比如使用new出来的对象

软引用(Soft Reference)：有用非必需，在发生内存溢出前，进行回收、

弱引用(Weak Reference):存活到下一次GC前

虚引用(Phantom Reference)：GC时收到系统通知

finlize()

对象被回收的时候，调用的方法，可以重写这个方法用于资源回收(不建议这样做，因为不知道何时能够调用),

垃圾回收算法

标记-清除算法

标记需要清除的区域，然后回收

优点：高效，实现简单

缺点：产生很多零碎空间

复制算法

主要用于年轻代回收

Eden和Survivor分为8:1，10%浪费掉

优点：实现简单，运行高效

缺点:需要移动对象，代价较高

标记整理

用于老年代

清除需要回收的对象，将存活对象整理在一起

优点:保留较大的空间

缺点：移动对象，代价高

分代收集

分代收集

新生代：复制算法

老年代：标记整理和标记清除

HotSpot的算法实现

枚举根节点

GC Roots需要上下文或者全局引用

枚举这个Root需要停顿Java执行，使用特定的数据结构，可以快速找到根节点

安全点

产生安全点的标准是是否具有让程序长时间执行的特征

线程安全停顿的方案：

1.抢占式中断：不现实

2.主动式中断：设置一个标志，线程主动去响应

安全区域

解决处于组塞或者睡眠的线程无法响应标志的问题

垃圾收集器

Serial收集器

单线程，新生代收集器

ParNew收集器

多线程版本的Serial收集器

Parallel Scavenge收集器

多线程收集器，缩短停顿时间

Serial Old收集器

单线程，老年代收集器

主要应用于客户端

Parallel Old收集器

老年代多线程收集器，标记-整理算法

CMS收集器

多线程老年代收集器

使用标记-清除算法

并发收集、低停顿

主要应用在B/S系统的服务端

CMS 收集器是一种以获取最短回收停顿时间为目标的收集器。收集的步骤如下:

1.初始标记:需要Stop The World,标记GC Roots能直接关联到的对象，数独较快

2.并发标记:进行GC Roots Tracing的过程

3.重新标记:需要Stop The World，修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，停顿时间一般比初始标记阶段长，比并发标记短

4.并发清除

缺点:

1.CMS收集器对CPU资源非常敏感

2.CMS收集器无法处理浮动垃圾，可能出现"Concurrent Mode Failure"失败而导致另一次Full GC的产生。

浮动垃圾:并发清除阶段，用户线程还在运行，伴随程序自然无法清除，之后等待下次，这就表示"浮动垃圾"。

3.CMS是一款基于"标记-清除"算法实现的收集器，这会产生大量的空间碎片。

G1收集器（特别重要）

多线程，分代收集器

主要应用于服务器

G1收集器的特点 :

1.并行与并发:缩短停顿时间，GC动作的时候，可以通过并发的方式让Java程序继续执行。

2.分代收集

3.空间停顿

4.可预测的停顿

G1收集器的运作大致可划分几步:

1.初始标记

2.并发标记

3.最终标记

4.筛选回收

JVM部分参数

参数名称	含义	默认值	

-Xms	初始堆大小	物理内存的1/64(<1GB)	默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制.
-Xmx	最大堆大小	物理内存的1/4(<1GB)	默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制
-Xmn	年轻代大小(1.4or lator)		**注意**：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小. 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8
-XX:NewSize	设置年轻代大小(for 1.3/1.4)		
-XX:MaxNewSize	年轻代最大值(for 1.3/1.4)		
-XX:PermSize	设置持久代(perm gen)初始值	物理内存的1/64	
-XX:MaxPermSize	设置持久代最大值	物理内存的1/4	
-Xss	每个线程的堆栈大小		JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。（校长） 和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"" -Xss is translated in a VM flag named ThreadStackSize" 一般设置这个值就可以了。
-XX:ThreadStackSize	Thread Stack Size		(0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.]
-XX:NewRatio	年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)		-XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5 Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。
-XX:SurvivorRatio	Eden区与Survivor区的大小比值		设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10
-XX:LargePageSizeInBytes	内存页的大小不可设置过大， 会影响Perm的大小		=128m
-XX:+UseFastAccessorMethods	原始类型的快速优化		
-XX:+DisableExplicitGC	关闭System.gc()		这个参数需要严格的测试
-XX:MaxTenuringThreshold	垃圾最大年龄		如果设置为0的话,则年轻代对象不经过Survivor区,直接进入年老代.对于年老代比较多的应用,可以提高效率.如果将此值设置为一个较大值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象再年轻代的存活 时间,增加在年轻代即被回收的概率 该参数只有在串行GC时才有效.
-XX:+AggressiveOpts	加快编译		
-XX:+UseBiasedLocking	锁机制的性能改善		
-Xnoclassgc	禁用垃圾回收		

-XX:SoftRefLRUPolicyMSPerMB	每兆堆空闲空间中SoftReference的存活时间	1s	softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap
-XX:PretenureSizeThreshold	对象超过多大是直接在旧生代分配	0	单位字节 新生代采用Parallel Scavenge GC时无效 另一种直接在旧生代分配的情况是大的数组对象,且数组中无外部引用对象.
-XX:TLABWasteTargetPercent	TLAB占eden区的百分比	1%	
-XX:+*CollectGen0First*	FullGC时是否先YGC	false	

内存分配与回收策略

对象优先分配在Eden

新生代大多分配在Eden，如果内存不足，将会发生一次Minor GC

Minor GC：新生代垃圾回收，非常频繁，数据较快

Full GC：老年代回收，一般伴有Minor GC，速度较慢

大对象直接进入老年代

很长的字符串和数组将会在老年代分配

长期存活的对象进入老年代

使用年龄计数器，Minor GC一次，存活的将会放入Suvivor区，年龄加1，到达15，放入老年区

Full GC 的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 区空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件:

1.调用 System.gc() 此方法的调用是建议 JVM 进行 Full GC，虽然只是建议而非一定，但很多情况下它会触发 Full GC，从而增加 Full GC 的频率，也即增加了间歇性停顿的次 数。因此强烈建议能不使用此方法就不要使用，让虚拟机自己去管理它的内存，可通过 -XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc()。

2 老年代空间不足 老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等，当执行 Full GC 后空间仍然不足，则抛出如下错误:

Java.lang.OutOfMemoryError: Java heap space 为避免以上两种状况引起的 Full GC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代 多存活一段时间及不要创建过大的对象及数组。

3 空间分配担保失败 使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果出现了 HandlePromotionFailure 担保失败，则会触发 Full GC。

4.永生代不足

JVM规范中运行时数据区域中的方法区，在HotSpot虚拟机中又被习惯称为永生代或者永生区，Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下也会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：java.lang.OutOfMemoryError: PermGen space 为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

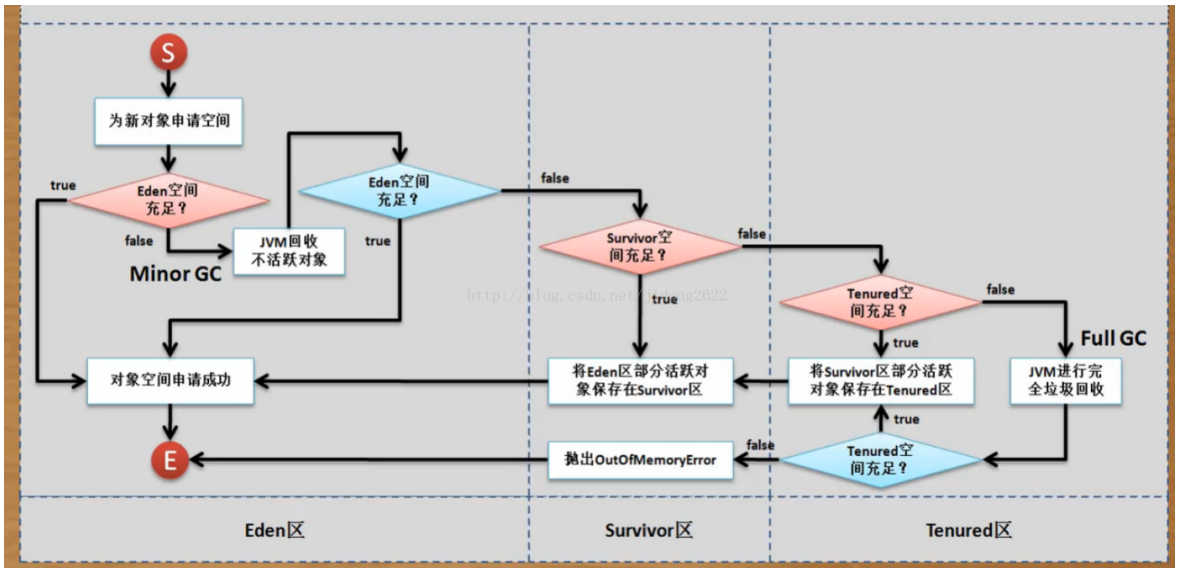
动态对象年龄判定

如果在survivor空间相同年龄所有对象大小的综合大于survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

垃圾回收执行流程

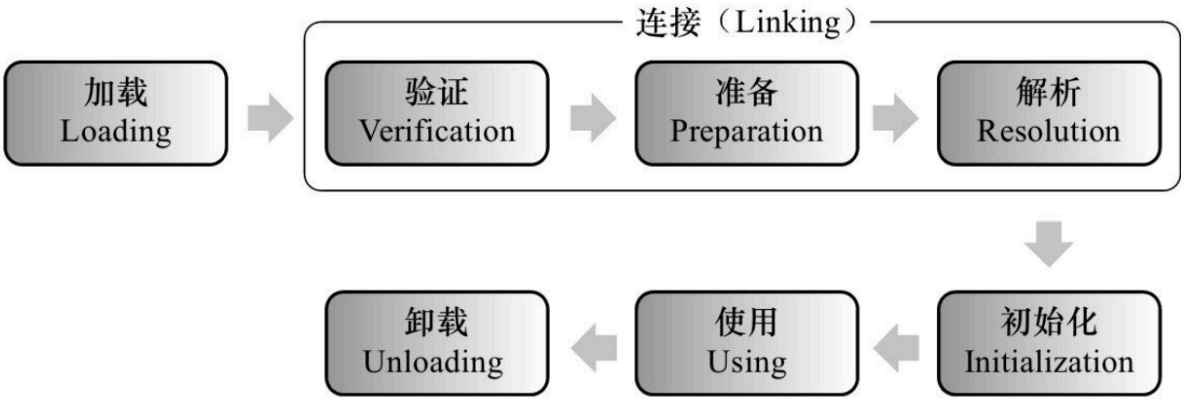
1.一个对象实例化时,先去看伊甸园有没有足够的空间 如果有,不进行垃圾回收 ,对象直接在伊甸园存储.

2. 如果伊甸园内存已满,会进行一次minor gc 然后再进行判断伊甸园中的内存是否足够 如果不足 则去看存活区的内存是否足够. 如果内存足够,把伊甸园部分活跃对象保存在存活区,然后把对象保存在伊甸园.
3. 如果内存不足,向老年代发送请求,查询老年代的内存是否足够 如果老年代内存足够,将部分存活区的活跃对象存入老年代.然后把伊甸园的活跃对象放入存活区,对象依旧保存在伊甸园.
4. 如果老年代内存不足,会进行一次full gc,之后老年代会再进行判断 内存是否足够,如果足够 同上. 如果不足，会抛出OutOfMemoryError.
5. 执行流程图：



类加载机制

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（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示。



加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，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再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都已HotSpot为基准。

加载

在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下3件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；
2. 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；
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口；

加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。 验证阶段大致会完成4个阶段的检验动作：

1. 文件格式验证：验证字节流是否符合Class文件格式的规范；例如：是否以魔术0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
2. 元数据验证：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。
3. 字节码验证：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。
4. 符号引用验证：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

那变量value在准备阶段过后的初始值为0而不是123.因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器()方法之中，所以把value赋值为123的动作将在初始化阶段才会执行。 至于“特殊情况”是指：public static final int value=123，即当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以标注为final之后，value的值在准备阶段初始化为123而非0.

解析

解析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。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。

符号引用:符号引用以一组符号来描述所引用的目标，符号可以是任何形字面量，只要使用时能无歧义的定位到目标即可。

直接引用:直接引用可以 是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。

初始化

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备极端，变量已经付过一次系统要求的初始值，而在初始化阶段，则根据程序猿通过程序制定的主管计划去初始化类变量和其他资源，或者说：初始化阶段是执行类构造器()方法的过程. ()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。如下：

```
public class Test
{
    static
    {
        i=0;
        System.out.println(i);//这句编译器会报错：Cannot reference a field before it is defined（非法向前应用）
    }
    static int i=1;
}
```

()方法与实例构造器()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类()方法执行之前，父类的()方法方法已经执行完毕，回到本文开篇的举例代码中，结果会打印输出：SSClass就是这个道理。 由于父类的()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。 ()方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生产()方法。 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成()方法。但接口与类不同的是，执行接口的()方法不需要先执行父接口的()方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的()方法。 虚拟机会保证一个类的()方法在多线程环境中被正确的加锁、同

步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的()方法，其他线程都需要阻塞等待，直到活动线程执行()方法完毕。如果在一个类的()方法中有好事很长的操作，就可能造成多个线程阻塞，在实际应用中这种阻塞往往是隐藏的。

```
package jvm.classload;

public class DealLoopTest
{
    static class DeadLoopClass
    {
        static
        {
            if(true)
            {
                System.out.println(Thread.currentThread()+"init DeadLoopClass");
                while(true)
                {
                }
            }
        }
    }

    public static void main(String[] args)
    {
        Runnable script = new Runnable(){
            public void run()
            {
                System.out.println(Thread.currentThread()+" start");
                DeadLoopClass dlc = new DeadLoopClass();
                System.out.println(Thread.currentThread()+" run over");
            }
        };

        Thread thread1 = new Thread(script);
        Thread thread2 = new Thread(script);
        thread1.start();
        thread2.start();
    }
}
```

运行结果：（即一条线程在死循环以模拟长时间操作，另一条线程在阻塞等待）

```
Thread[Thread-0,5,main] start
Thread[Thread-1,5,main] start
Thread[Thread-0,5,main]init DeadLoopClass
```

需要注意的是，其他线程虽然会被阻塞，但如果执行()方法的那条线程退出()方法后，其他线程唤醒之后不会再次进入()方法。同一个类加载器下，一个类型只会初始化一次。 将上面代码中的静态块替换如下：

```
static
        {
            System.out.println(Thread.currentThread() + "init DeadLoopClass");
            try
            {
                TimeUnit.SECONDS.sleep(10);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
```

运行结果：

```
Thread[Thread-0,5,main] start
Thread[Thread-1,5,main] start
Thread[Thread-1,5,main]init DeadLoopClass (之后sleep 10s)
Thread[Thread-1,5,main] run over
Thread[Thread-0,5,main] run over
```

虚拟机规范严格规定了有且只有5中情况（jdk1.7）必须对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1. 遇到new,getstatic,putstatic,invokestatic这失调字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
5. 当使用jdk1.7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初始化，则需要先出触发其初始化。

开篇已经举了一个范例：通过子类引用付了的静态字段，不会导致子类初始化。 这里再举两个例子。

1. 通过数组定义来引用类，不会触发此类的初始化：（SuperClass类已在本文开篇定义）

```
public class NotInitialization
{
    public static void main(String[] args)
    {
        SuperClass[] sca = new SuperClass[10];
    }
}
```

运行结果：（无）

2. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化：

```
{
    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(ConstClass.HELLOWORLD);
    }
}
```

运行结果：hello world

类加载器以及双亲委派

类加载器

实现根据一个类的全限定名来获取类的二进制字节流这个动作的代码块称为类加载器。

比较一个类是否相等，必须先比较两个类的类加载器是否一样，然后再比较是否来源于同一个Class文件。

类加载器的类别：

启动类加载器

扩展类加载

应用程序类加载器

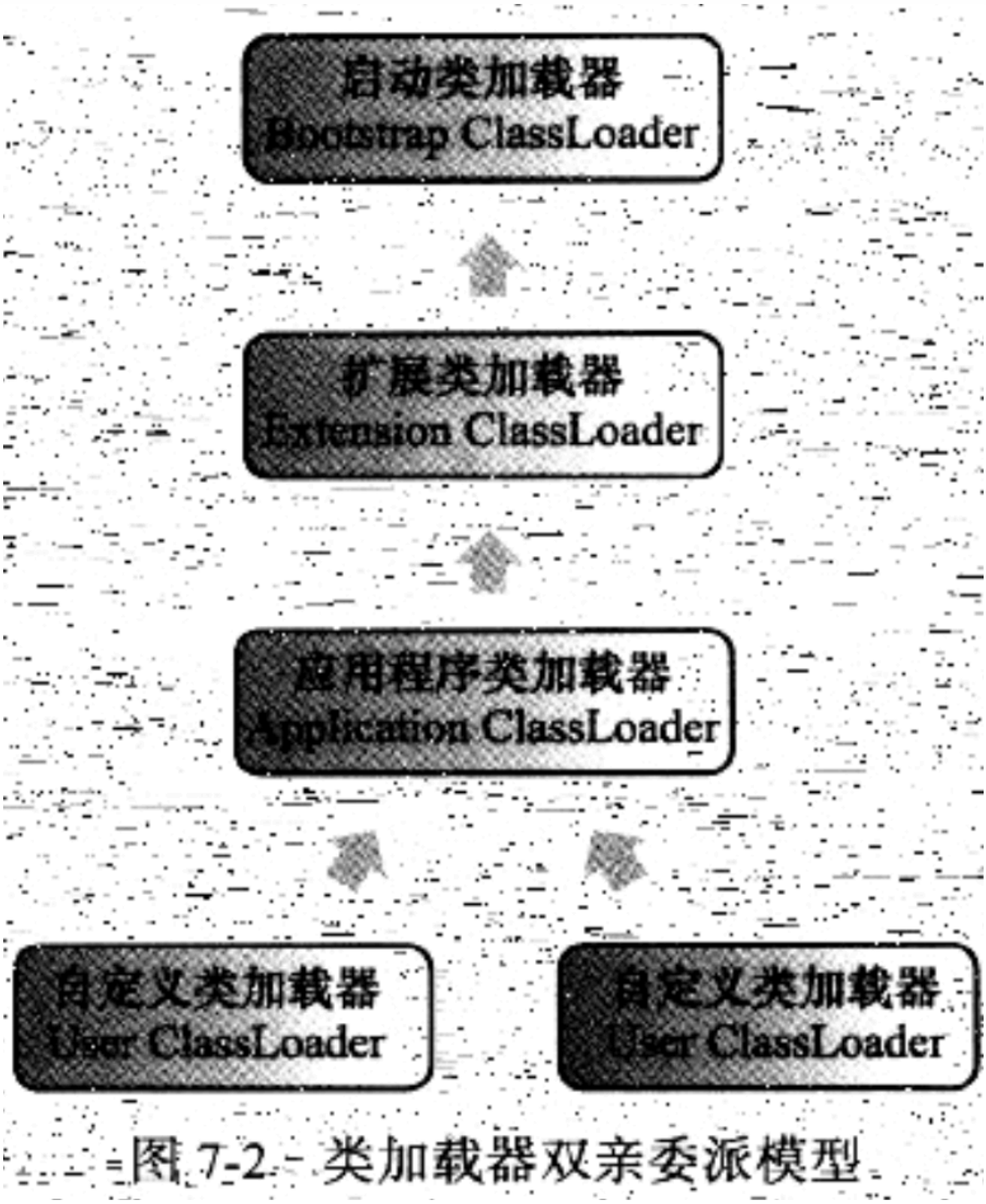


图 7-2 类加载器双亲委派模型

双亲委派

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是将这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，请求最后将会总会传递到顶层的启动类加载，只有当父加载器反馈加载不了这个请求时，自加载器才会尝试自己去加载。

好处

使用双亲委派模型来组织类加载器之间的关系，使得 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object，它存放在 rt.jar 中，无论哪个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同 一个类。相反，如果没有双亲委派模型，由各个类加载器自行加载的话，如果用户编写了

一个称为java.lang.Object 的类，并放在程序的 ClassPath 中，那 系统中将会出现多个不同的 Object 类，程序将变得一片混乱。如果开发者尝试编写一个与 rt.jar 类库中已有类重名的 Java 类，将会发现可以正常编译，但 是永远无法被加载运行。

破坏双亲委派模型

重写loadClass()方法，让自己实现类加载器直接去加载类。

Java内存模型

定义

Java内存模型主要用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。Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中变量存储到内存和从内存中取出变量这样的底层细节。

主内存和工作内存

Java内存模型规所有的变量都存储在主内存中，每个线程还有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，不能直接读写驻内存中的变量。

这里的主内存可以认为对应Java堆中的对象实例数据部分，而工作内存则对应于虚拟机栈中的部分区域。

什么是线程安全?

共享变量的操作要具体原子性、同时也要关注可见性、顺序性

CAS底层原理剖析

在Java并发中，我们最初接触的应该就是`synchronized`关键字了，但是`synchronized`属于重量级锁，很多时候会引起性能问题，`volatile`也是个不错的选择，但是`volatile`不能保证原子性，只能在某些场合下使用。

像`synchronized`这种独占锁属于**悲观锁**，它是在假设一定会发生冲突的，那么加锁恰好有用，除此之外，还有**乐观锁**，乐观锁的含义就是假设没有发生冲突，那么我正好可以进行某项操作，如果要是发生冲突呢，那我就重试直到成功，乐观锁最常见的就是`CAS`。

我们在读Concurrent包下的类的源码时，发现无论是**ReenterLock内部的AQS，还是各种Atomic开头的原子类**，内部都应用到了`CAS`，最常见的就是我们在并发编程时遇到的`i++`这种情况。传统的方法肯定是在方法上加上`synchronized`关键字:

```
public class Test {

    public volatile int i;

    public synchronized void add() {
        i++;
    }
}
```

但是这种方法在性能上可能会差一点，我们还可以使用`AtomicInteger`，就可以保证`i`原子的`++`了。

```
public class Test {

    public AtomicInteger i;

    public void add() {
        i.getAndIncrement();
    }
}
```

我们来看`getAndIncrement`的内部：

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

再深入到`getAndAddInt`():

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

这里我们见到`compareAndSwapInt`这个函数，它也是`CAS`缩写的由来。那么仔细分析下这个函数做了什么呢?

首先我们发现`compareAndSwapInt`前面的`this`，那么它属于哪个类呢，我们看上一步`getAndAddInt`，前面是`unsafe`。这里我们进入的`Unsafe`类。这里要对`Unsafe`类做个说明。结合`AtomicInteger`的定义来说：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
    ...
```

在`AtomicInteger`数据定义的部分，我们可以看到，其实实际存储的值是放在`value`中的，除此之外我们还获取了`unsafe`实例，并且定义了`valueOffset`。再看到`static`块，懂类加载过程的都知道，`static`块的加载发生于类加载的时候，是最先初始化的，这时候我们调用`unsafe`的`objectFieldOffset`从`Atomic`类文件中获取`value`的偏移量，那么`valueOffset`其实就是记录`value`的偏移量的。

再回到上面一个函数`getAndAddInt`，我们看`var5`获取的是什么，通过调用`unsafe`的`getIntVolatile(var1, var2)`，这是个native方法，具体实现到JDK源码里去看了，其实就是获取`var1`中，`var2`偏移量处的值。`var1`就是`AtomicInteger`，`var2`就是我们前面提到的`valueOffset`,这样我们就从内存里获取到现在`valueOffset`处的值了。

现在重点来了，`compareAndSwapInt (var1, var2, var5, var5 + var4)`其实换成`compareAndSwapInt (obj, offset, expect, update)`比较清楚，意思就是如果`obj`内的`value`和`expect`相等，就证明没有其他线程改变过这个变量，那么就更新它为`update`，如果这一步的`CAS`没有成功，那就采用自旋的方式继续进行`CAS`操作，取出乍一看这也是两个步骤了啊，其实在`JNI`里是借助于一个`CPU`指令完成的。所以还是原子操作。

CAS底层原理

CAS底层使用`JNI`调用C代码实现的，如果你有`Hotspot`源码，那么在`Unsafe.cpp`里可以找到它的实现：

```
static JNINativeMethod methods_15[] = {
    //省略一堆代码...
    {CC"compareAndSwapInt",  CC"("OBJ"J""I""I"")Z",
 FN_PTR(Unsafe_CompareAndSwapInt)},
    {CC"compareAndSwapLong", CC"("OBJ"J""J""J"")Z",
 FN_PTR(Unsafe_CompareAndSwapLong)},
    //省略一堆代码...
};
```

我们可以看到compareAndSwapInt实现是在`Unsafe_CompareAndSwapInt`里面，再深入到`Unsafe_CompareAndSwapInt`：

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe,
jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

p是取出的对象，addr是p中offset处的地址，最后调用了`Atomic::cmpxchg(x, addr, e)`, 其中参数x是即将更新的值，参数e是原内存的值。代码中能看到cmpxchg有基于各个平台的实现，这里我选择Linux X86平台下的源码分析：

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*
dest, jint     compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r"
(mp)
                    : "cc", "memory");
  return exchange_value;
}
```

这是一段小汇编，`__asm__`说明是ASM汇编，`__volatile__`禁止编译器优化

```
// Adding a lock prefix to an instruction on MP machine
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

`os::is_MP`判断当前系统是否为多核系统，如果是就给总线加锁，所以同一芯片上的其他处理器就暂时不能通过总线访问内存，保证了该指令在多处理器环境下的原子性。

在正式解读这段汇编前，我们来了解下嵌入汇编的基本格式：

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

- **template**就是`cmpxchgl %1,(%3)`表示汇编模板
- **output operands**表示输出操作数,`=a`对应eax寄存器
- **input operand** 表示输入参数，`%1`就是`exchange_value`,`%3`是`dest`,`%4`就是`mp`，`r`表示任意寄存器，`a`还是`eax`寄存器
- **list of clobbered registers**就是些额外参数，`cc`表示编译器`cmpxchgl`的执行将影响到标志寄存器,`memory`告诉编译器要重新从内存中读取变量的最新值，这点实现了`volatile`的感觉。

那么表达式其实就是`cmpxchgl exchange_value ,dest`，我们会发现`%2`也就是`compare_value`没有用上，这里就要分析`cmpxchgl`的语义了。`cmpxchgl`末尾`l`表示操作数长度为`4`，上面已经知道了。`cmpxchgl`会默认比较`eax`寄存器的值即`compare_value`和`exchange_value`的值，**如果相等，就把dest的值赋值给exchange_value,否则，将exchange_value赋值给eax**。具体汇编指令可以查看Intel手册CMPXCHG

最终，JDK通过CPU的`cmpxchgl`指令的支持，实现`AtomicInteger`的`CAS`操作的原子性。

Java性能调优篇

随着应用的数据量不断的增加，系统的反应一般会越来越慢，这个时候我们就需要性能调优。性能调优的步骤如下：

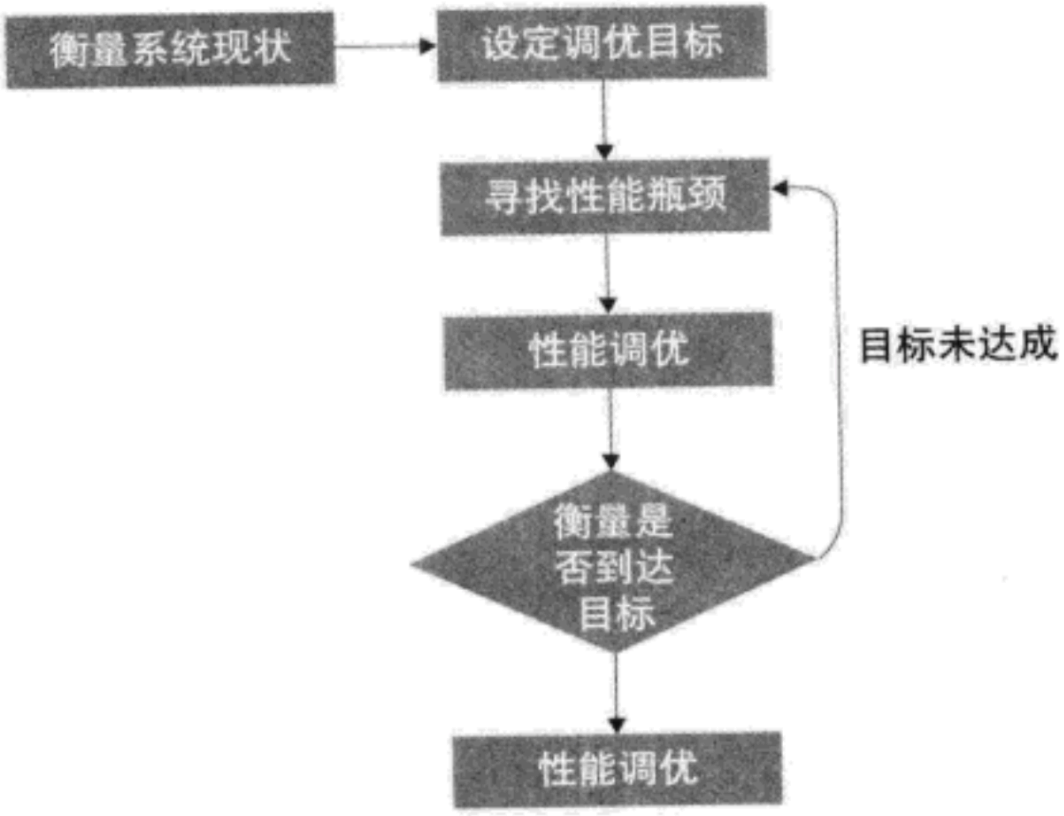


图 5.1　调优步骤

寻找性能瓶颈

通常性能瓶颈的表象是资源消耗过多、外部处理系统的不足，或者资源消耗不多，但是程序的响应速度却达不到要求。下面的分析针对于Linux。

CPU消耗分析

CPU主要用于中断、内核以及用户进程的处理；优先级为中断、内核和用户进程。我们首先有了解3个概念。

1.上下文切换

线程从CPU换出到下一次执行，称为一次上下文切换。上下文切换过多带来的影响：内核占用较多的CPU使用时间，响应速度下降。

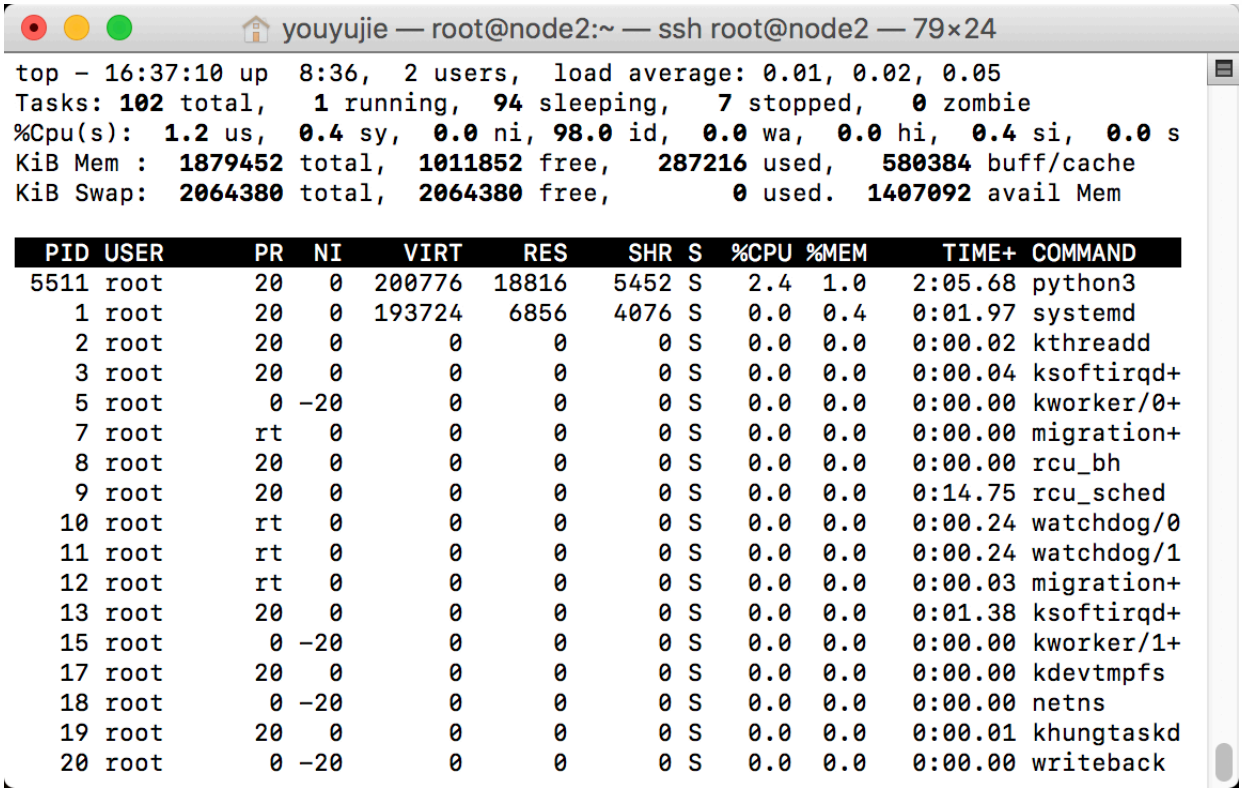
2.运行队列

每一个CPU核都维护一个可运行的线程队列，一般建议每个CPU的运行队列是1-3个。

利用率

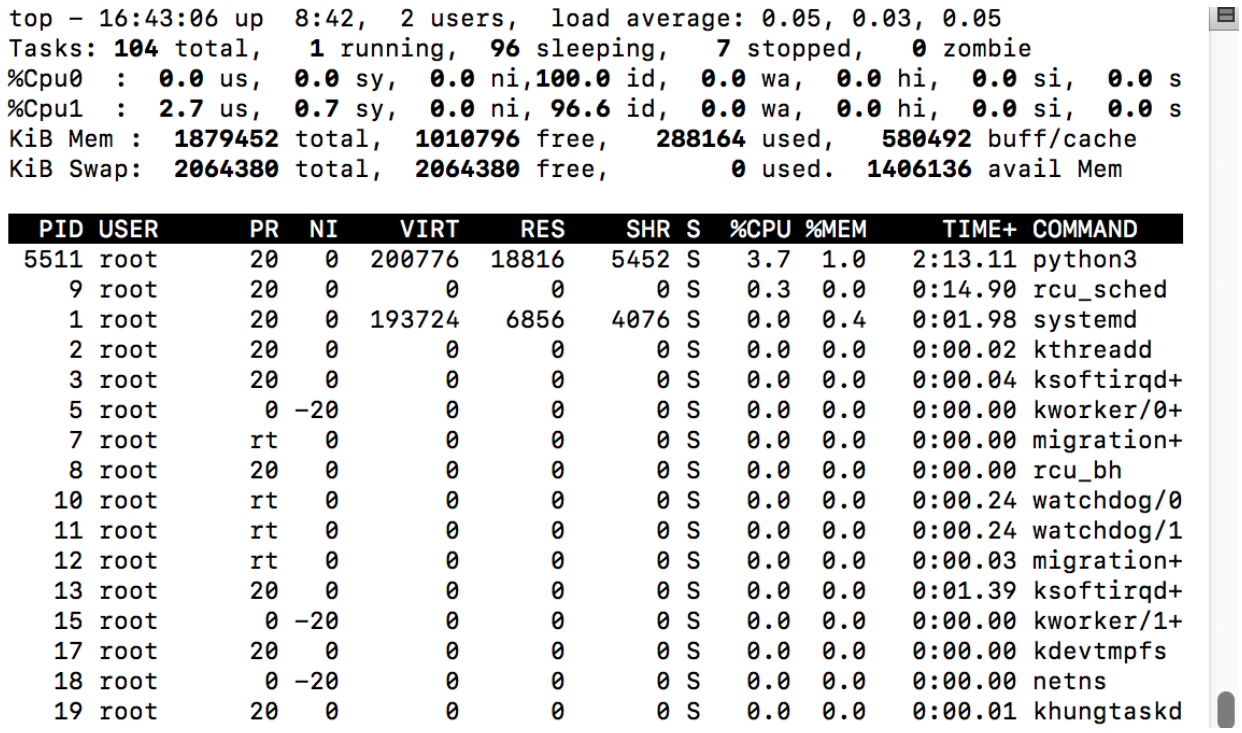
CPU利用率为CPU在用户进程、内核、中断处理、IO等待以及空闲5个部分使用百分比。可以使用top或者pidstat命令查看。

top



1.2%us,表示用户进程处理所占的百分比，0.4%sy表示系统内核线程处理所占用的百分比，0.0%表示改变优先级任务所占的百分比，98%id表示CPU空闲时间所占的百分比，0.0%wa表示等待IO所占用百分比，0.0%hi表示由于硬件中断所占用的百分比；0.4%si表示软件中断所占用的百分比；

可以在上面试图的基础上按1，就可查看每个CPU的运行情况。注意两图的变化：



可以查看每个进程所占用的CPU情况。

pidstat

需要安装sysstat，然后在使用pidstat 1 2，表示每隔1s输出，一共输出两次，如下

```
[root@node2 ~]# pidstat 1 2
Linux 3.10.0-693.el7.x86_64 (node2)     06/11/2018      _x86_64_        (2 CPU)

04:48:02 PM   UID       PID    %usr %system  %guest    %CPU   CPU  Command
04:48:03 PM     0      5511    2.91    0.97    0.00    3.88     1  python3
04:48:03 PM     0      5660    0.00    0.97    0.00    0.97     0  pidstat

04:48:03 PM   UID       PID    %usr %system  %guest    %CPU   CPU  Command
04:48:04 PM    27      1640    0.00    1.00    0.00    1.00     1  mysqld
04:48:04 PM     0      5511    2.00    0.00    0.00    2.00     1  python3

Average:      UID       PID    %usr %system  %guest    %CPU   CPU  Command
Average:       27      1640    0.00    0.49    0.00    0.49     -  mysqld
Average:        0      5511    2.46    0.49    0.00    2.96     -  python3
Average:        0      5660    0.00    0.49    0.00    0.49     -  pidstat
```

cpu表示当前使用CPU的个数，如果需要详细查看某一个进程，可以使用如下命令

```
[root@node2 ~]# pidstat -p 5511 -t 1 5
Linux 3.10.0-693.el7.x86_64 (node2)     06/11/2018      _x86_64_        (2 CPU)

04:51:23 PM   UID      TGID       TID    %usr %system  %guest    %CPU   CPU  Co
mmand
04:51:24 PM     0      5511         -    3.00    1.00    0.00    4.00     1  py
thon3
04:51:24 PM     0         -      5511    3.00    1.00    0.00    4.00     1  |_
_python3

04:51:24 PM   UID      TGID       TID    %usr %system  %guest    %CPU   CPU  Co
mmand
04:51:25 PM     0      5511         -    3.00    0.00    0.00    3.00     1  py
thon3
04:51:25 PM     0         -      5511    3.00    0.00    0.00    3.00     1  |_
_python3

04:51:25 PM   UID      TGID       TID    %usr %system  %guest    %CPU   CPU  Co
mmand
04:51:26 PM     0      5511         -    2.97    0.00    0.00    2.97     1  py
```

TID表示线程id，这个命令的好处是可以查看每一个线程的具体CPU和线程使用率。

需要知道的是CPU消耗主要集中在us，sy两个值中。

1.us

java造成高us的主要原因是：线程一直处于可运行状态、通常是循环、正则和无阻塞运行造成的；还有一个原因是频繁的GC。解决问题的思路是

先通过Linux命令定位出那一个线程的us较高，然后将线程的id转化为16进制，然后通过 kill -3 [javapid]或者jstack的方式dump出应用的java线程信息，通过之前转化出来的16进制值找到nid值的线程。该线程也就是CPU消耗型线程，这里需要多采样几次，以确保准确。

2.sy

造成这个的主要原因是：启动了很多的线程，且这些线程主要是阻塞和可执行之间切换，然后造成了大量的上下文切换。

解决思路同上。

文件IO消耗分析

Linux在操作文件的时候，先将文件放入文件缓存，直到内存不足或者进程使用。这是一种提升IO速度的方式。

1.pidstat

```
[root@node2 ~]# pidstat -d -t -p 5793
Linux 3.10.0-693.el7.x86_64 (node2)     06/11/2018      _x86_64_        (2 CPU)

05:33:12 PM   UID      TGID       TID   kB_rd/s   kB_wr/s kB_ccwr/s  Command
05:33:12 PM     0      5793         -      0.00      0.00      0.00  python3
05:33:12 PM     0         -      5793      0.00      0.00      0.00  |__python3
```

KB_rd/s表示每秒读，KB_wr/s每秒写

2.iostat

```
Linux 3.10.0-693.el7.x86_64 (node2)     06/11/2018      _x86_64_        (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.98    0.00    0.29    0.01    0.00   98.72

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
scd0              0.00         0.03         0.00       1028          0
sda               0.59         9.43         9.84     328009     342256
dm-0              0.69         8.37         5.41     291001     188036
dm-1              0.00         0.06         0.00       2228          0
dm-2              0.03         0.13         4.44       4617     154200
```

查看历史IO

网络IO分析

使用命令 sar -n FULL 1 2

内存消耗分析

JVM内存消耗过多会导致GC喜欢 ii 次难过频繁，CPU消耗增加，应用线程的执行速度严重下降，甚至造成OutOfMemoryError，最终导致Java进程退出。

vmstat

可使用这个命令查看信息和内存相关的主要是memory下的swpd、free、buff、cache以及swap下的si和so。

swpd：虚拟内存已使用的部分

free：空闲的物理内存

buff：用于缓冲的内存

cache：缓存的内存

swap的si：每秒从disk读至内存的数据量

swap的so：每秒从内存中xierudisk的数据量。

swap过高通常是由于物理内存不够用，swap消耗情况主要关注的是swap IO的状况。

Java应用是单进程应用，因此只要JVM的内存设置不是过大，是不会操作到swap区域的。

sar

```
[root@node2 ~]# sar -r 2 5
Linux 3.10.0-693.el7.x86_64 (node2)     06/12/2018      _x86_64_        (2 CPU)

11:07:57 AM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commi
t  kbactive   kbinact   kbdirty
11:07:59 AM   1442092    437360     23.27     20008    191764    766292     19.4
3    219020    142600       712
11:08:01 AM   1442092    437360     23.27     20008    191764    766292     19.4
3    219020    142600       712
11:08:03 AM   1442092    437360     23.27     20008    191764    766292     19.4
3    219020    142600       712
11:07:49 AM   1442060    437392     23.27     20008    191764    766292     19.4
3    219032    142600       712
11:07:51 AM   1442100    437352     23.27     20008    191764    766292     19.4
3    219032    142600       712
Average:      1442087    437365     23.27     20008    191764    766292     19.4
3    219025    142600       712
```

物理内存相关的信息主要是kbmemfree、kbmemused、%memused、kbbuffers、kbcached。当物理内存有空心啊时，linux会使用一些物理内存用于buffer 以及cache。一提升系统的运行效率；物理内存=kbmemfree+kbbuffers+kbcahched。

vmstat和sar的共同弱点是不能分析进程所占用的内存量。

top

可查看进程所消耗的内存量，看到的是JVM已分配的内存已加上Java应用所消耗的JM以外的物理内存，这会导致top中看到Java所消耗的内存带下可能超过-Xmx机上-XX：MaxPerSize设置内存大小，并且Java程序启动后知识占据-Xms的地址空间，但并没有占据实际的内存，只有在相应的地址空间被使用过后被计入消耗的内存中。很难根据top判断Java进程消耗的内存中有多少是属于JVM，一个小技巧是，对犹豫内存满而发生过Full GC的应用而言，多数情况先，可以认为java进程中显示出来的内存消耗值即为JVM -Xmx的值加上消耗的JVM外的内存值。

pidstat

使用命令 pidstat -r -p [pid][interval] [interval][times], [times]. 执行命令可查看该进程所占用的物理内存和虚拟内存大小。

```
12:02:48          PID  minflt/s  majflt/s      VSZ     RSS   %MEM  Command
12:02:49         2013      1.00      0.00 1822224 1615900  38.96  java
12:02:50         2013      0.00      0.00 1822224 1615900  38.96  java
12:02:51         2013      0.00      0.00 1822224 1615900  38.96  java
Average:         2013      0.33      0.00 1822224 1615900  38.96  java
```

图 5.13　pidstat 查看进程内存消耗

对物理内存的消耗

基于Direct BytBuffer可以很容易地实现对物理内存的直接操作，无须耗费JVM Heap区。

```
package cn.edu.hust;

import java.nio.ByteBuffer;

public class JVM1 {
    public static void main(String[] args) throws InterruptedException {
```

```
        Thread.sleep(20000);
        System.out.println("read to create bytes, so JVM heap will be used");
        byte[] bytes=new byte[128*1000*1000];
        bytes[0]=1;
        bytes[1]=2;
        Thread.sleep(10000);
        System.out.println("read to allocate & put direct bytebuffers,no JVM heap
should be used");

        ByteBuffer buffer=ByteBuffer.allocate(128*1024*1024);
        buffer.put(bytes);
        buffer.flip();
        Thread.sleep(10000);

        System.out.println("ready to gc,JVM heap will be freed");
        bytes=null;
        System.gc();
        Thread.sleep(10000);
        System.out.println("ready to get bytes,then JVM heap will be used");

        byte[] resultBytes=new byte[128*1000*1000];
        buffer.get(resultBytes);
        System.out.println("resultBytes[1] is:"+resultBytes[1]);
        Thread.sleep(10000);
        System.out.println("ready to gc all");
        buffer=null;
        resultBytes=null;
        System.gc();
        Thread.sleep(10000);
    }
}
```

在IDEA中配置 -Xms140M -Xmx140M参数执行上面的代码可以根据命令查看

表 5.1

	Java 进程内存	JVM 堆旧生代内存
ready to create bytes	上涨	98%左右
ready to allocate & put direct bytebuffer	翻了一倍	无变化
ready to gc	无变化	0.1%左右
ready to get bytes	无变化	98%左右
ready to gc all	回到 ready to create bytes 时的大小	0.1%左右

基于direct vytebuffer消耗的JVM heap外的物理内存，同样是给予GC方式释放。

对JVM内存的消耗

Java程序出现内存消耗过多、GC频繁或者OutOfmemeroyError情况，要首先分析其耗费的JVM外的物理内存还是JVM heap区。

程序执行慢情况分析

有些情况是资源消耗不多，但程序执行仍然慢，这种现象多出现访问量不是非常大的情况下，造成这种原因有三种：

锁竞争激烈

锁竞争激烈直接会造成程序执行慢。例如一个典型的例子是数据库连接池，通常数据库连接池提供的连接数是有限的。

未充分使用硬件资源

可以优化程序，充分发挥硬件资源的座椅哦难过，此时可进行一定的优化充分使用硬件资源，提升程序的执行速度。

数据量的增长

数据量增长通常也是造成程序执行慢的典型原因。

调优

JVM调优

常用的内存管理调优的方法，这些方法都是为了尽量降低GC所导致应用暂停时间。

代大小的调优

minot GC会远快于Full GC，各个代的大小设置直接决定了minor GC 和Full GC 触发的时机，在代大小的调优上，最关键的参数是-Xms -Xmx -Xmn -XX:SurivivorRatio -XX:MaxTenuringThreshold

-Xmn决定了新生代的空间大小，新生代Eden、S0和S1三个区域的比率可以通过-XX： SurvivorRatio来控制。

-XX:MaxTenuringThreshold控制对象经历多少次Minor GC后转入旧生代，通常又将此致值称为新生代存活周期，此参数只有在串行GC时有效，其他GC方式则有SunJDK自行决定。

1.避免新生代的大小设置过小

新生代设置过小，会产生两种比较明显的现象，一是minor GC的次数更加频繁；二是有可能导致minor GC对象直接进入旧生代，此时如进入旧生代的对象占据旧生代剩余空间，则出发FUll GC。

如果调大新生代大小外，如果能够调大JVM Heap的大小，那就更好了，但JVM Heap调大通常意味着单次GC时间的增加。

调整时的原则是在不能调大JVm Heap的情况下，尽可能方法新生代年代空间，尽量让对象在minor GC阶段被回收，但新生代空间也不可过大；在能够调大JVM Heap的情况下，可以按照增加的新生代空间大小增加JVM Heap大小，以保证旧生代空间够用。

2.避免新生代设置过大

新生代设置过大会带来两个典型想象，一是旧生代变小了，有可能导致Full GC频繁执行；二是minor GC的耗时大幅增加。

新生代通常不能设置过大，大多数场景都应设置得比旧生代小，通常推荐的比例是新生代占JVM Heap区大小的33%左右。

3.避免Survior区过小或过大

在无法调整JVM Heap以及新生代的大小时，合理调整Survivor区的带下也能带来一些效果。调大SurvivorRatio值意味着Eden区域变大，minor GC的触发次数会降低，但此时Surivor区域空间变小，如有超过Survivor空间大小的对象在minor GC欧仍没有被回收，则会直接进入旧生代；调小SurvivorRatio则意味着Eden区域变小，minorGC的触发次数增加，Survivor区域变大，就意味着可以存储更多在minor GC后存活的对象，避免起进入旧生代。

4.合理设置新生代存活周期

-Xms、-Xmx是用户调整整个JVM Heap区大小，在内存不够用的情况下可适当加大此值，这个值能调整到多大取决于操作系统位数以及CPU能力。

-Xmn适用于调整新生代的大小，新生代的大小决定了多少比例的对象有机会在minor GC阶段被回收，此值对应的也决定旧生代的带下。新生代越大，通常意味着多数对象能够在minor GC阶段被回收掉，但同时意味着旧生代的空间变小，可能造成更频繁的Full GC甚至时OutOfMemoryError。

在清楚掌握minorGC、FullGC的触发时机以及代大小的调整后，结合应用的状况通常就可较好设置代的大小，减少GC所占用的时间。

GC策略的优化

选择不同的GC收集器，会有不同的效果，CMS GC多数动作是和应用并发进行的，确实可以减少GC动作给应用造成的暂停。

对于web应用而言，在G1还不够成熟的情况下，CMS GC是不错的选择。

程序调优

CPU消耗严重的解决方法

1.CPU us高的解决方法

CPU us高的原因主要是执行程序无任何挂起动作，且一直执行，导致CPU没有机会去调度执行其他的线程，导致线程饿死的现象。

```
for (int k = 0; k < 10000; k++) {
        list.add(str+String.valueOf(k));
  if(k%50==0){
        try{
            Thread.sleep(1);
            }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
  }
}
```

以上图片增加sleep，这种修改方式是以损失单次执行性能为代价，但由于降低CPU的消耗，对多线程的应用而言，反而提高总体的平均性能。

对GC频繁造成的CPU us高的现象，则要通过JVM调优或程序调优，降低GC的执行次数。

2.CPU sy高的解决方法

CPU sy高的原因主要是线程的运行状态经常切换，这种情况，最简单的优化方法是减小线程数。

造成CPU sy高的原因除了启动的线程过多以外，还有一个重要的原因是线程之间锁竞争激烈，造成线程状态经常要切换，因此尽可能降低线程间的锁竞争也是常见的优化方法。

文件IO消耗严重的解决方法

造成文件IO消耗严重的原因主要是多线程在写大量的数据到同一个文件，导致文件很快变得很大，从而写入速度越来越慢，并造成各线程激烈争抢文件锁。通常采用几种方式调优：

1.异步写文件

将写文件的同步动作改成异步动作。

2.批量读写

频繁的读写操作对IO消耗很严重，批量操作将大幅度提升IO操作的性能。

3.限流

将文件IO消耗控制到一个能接受的范围。

4.限制文件大小

对于每个输出文件，都应做大小的限制。

网络IO消耗严重的解决方法

常用的调优方法为进行限流，限流通常是闲置发送packet的频率，从而在网络IO消耗可接受的情况下来发送packet。

对内存消耗的严重情况

1.释放不必要的引用，例如使用ThreadLocal

2.使用对象缓存池

3.采用合理的缓存失效算法，例如FIFO和LRU策略的CachePool

4.合理使用softReference和WeakReference

对象资源消耗不多，但程序执行慢的情况

锁竞争激烈

锁竞争的状态会比较明显，这时线程很容易处于等待锁的状况，从而导致性能小将以及CPU sy上升。

1.使用并发包中的类

给予CAS来做无需lock就可以实现资源一致性保证，主要的实现nonblocking的算法

2.使用Treiber算法

Treiber算法主要用于实现stack，基于Treiber算法实现无阻塞Stack。

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,newHead));
        return oldHead.item;
    }

    static class Node<E> {
        final E item;
        Node<E> next;

        public Node(E item) { this.item = item; }
    }
}
```

3.使用Michael-Scott非阻塞队列算法

和Treiber算法类似,也是基于CAS以及AtomicReference来实现队列的非阻塞操作,ConcurrentLinkedQueue就是典型的基于Michael-Scott实现的非阻塞队列。

从上面两种算法来看,基于CAS和AtomicReference来实现无阻塞算法是不错的选择。但值得注意的是,由于CAS是基于不断的循环比较来保证资源一致性的,对于冲突较多的应用场景而言,CAS会带来更高的CPU消耗,因此不一定采用CAS实现无阻塞的就一定比采用Lock方式的性能好。业界还有一些无阻塞算法的改进,如MCAS、WSTM20等

4.尽可能少用锁

尽可能让锁仅在需要的地方出现,通常没必要对整个方法加锁,而只对需要控制的资源做加锁操作。尽可能让锁最小化,例如一个操作中需要保护的资源只有HashMap,那么在加锁时则可只synchronized(map),而没必要synchronized(this).

5.拆分锁

把独占锁拆分为多把锁,常见的有读写锁拆分及类似ConcurrentHashMap中默认拆分为16把锁的方法。需要注意的是采用拆分锁后,全局性质的操作会变得比较复杂,例如ConcurrentHashMap的size操作。

6.去除读写操作的互斥锁

在修改时加锁,并复制对象进行修改,修改完后切换对象的引用,而读取操作时则不加锁,这种方式称为CopyOnWrite。CopyOnWriteArrayList就是其中的典型实现。这种做法好处是可以明显提升读的性能,适用读多写少的场合,坏处是造成更多的内存消耗。

未充分使用硬件资源

1.未充分使用CPU

对于JAVA应用而言,通常原因就是在能并行处理的场景中未使用足够的线程。

另外,单线程的计算,也可以拆分为多线程来分别计算,最后合并结果,JDK7中的fork-join框架可以给以上场景提供一个好的支撑方法

2.未充分使用内存

如数据的缓存、耗时资源的缓存(如数据库连接,网络连接)、页面片段的缓存

面试题

1.Java中的原子类？有哪些问题？JDK源码如何解决?

CAS底层实现，ABA问题，AtomicStampedReference:原子更新带有版本的引用类型，该类将整型值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用CAS进行原子更新时可能出现的ABA问题。

2.final变量重排序?

final重排序规则：

1.在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2.初次读一个包含final域的对象引用，与随后初次读这个final域，这两个操作之间不能重排序。

写final域的重排序规则：

写final域的重排序规则禁止把final域的写重排序到构造函数之外。这个规则的包含下面两个方面

1.JMM禁止编译器把final域的写重排序到构造函数之外。

2.编译器会在final域的写之后，构造函数return 之前，插入一个StoreStore屏障。这个屏障禁止把final域的写重排序到构造函数之外。

读final域的重排序规则:

读final域的重排序规则是:在一个线程中，初次读对象引用与初次读该对象包含的final域，JMM禁止处理器重排序这两个操作(注意，这个规则只是针对于这个处理器)。编译器会在读final域操作的前面插入一个LoadLoad屏障。

在读一个对象的final域之前，一定会读包含这个final域对象的引用。

final域为引用类型

在前面的基础上加入了如下约束:

在构造函数内对一个final引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作是不能重排序的。

3.构造代码块和构造函数的区别?

关于构造函数，以下几点要注意：

1.**对象一建立，就会调用与之相应的构造函数**，也就是说，不建立对象，构造函数时不会运行的。

2.构造函数的作用是用于给对象进行初始化。

3.一个对象建立，构造函数只运行一次，而一般方法可以被该对象调用多次。

关于构造代码块，以下几点要注意：

1. 构造代码块的作用是给对象进行初始化。
2. **对象一建立就运行构造代码块了，而且优先于构造函数执行**。这里要强调一下，有对象建立，才会运行构造代码块，类不能调用构造代码块的，而且**构造代码块与构造函数的执行顺序是前者先于后者执行**。
3. 构造代码块与构造函数的区别是：**构造代码块是给所有对象进行统一初始化，而构造函数是给对应的对象初始化**，因为构造函数是可以多个的，运行哪个构造函数就会建立什么样的对象，但无论建立哪个对象，都会先执行相同的构造代码块。也就是说，构造代码块中定义的是不同对象共性的初始化内容。