2019Android 多线程总结

1. 什么是线程

线程就是进程中运行的多个子任务，是操作系统调用的最小单元

2. 线程的状态

New:新建状态，new 出来，还没有调用 start

Runnable:可运行状态，调用 start 进入可运行状态，可能运行也可能没有运行，取决于操作系统的调度

Blocked:阻塞状态，被锁阻塞，暂时不活动，阻塞状态是线程阻塞在进入 synchronized 关键字修饰的方法或代码块(获取锁)时的状态。

Waiting：等待状态，不活动，不运行任何代码，等待线程调度器调度，wait sleep

Timed Waiting:超时等待，在指定时间自行返回

Terminated:终止状态，包括正常终止和异常终止

3. 线程的创建

a.继承 Thread 重写 run 方法

b.实现 Runnable 重写 run 方法

c.实现 Callable 重写 call 方法

实现 Callable 和实现 Runnable 类似，但是功能更强大，具体表现在

a.可以在任务结束后提供一个返回值，Runnable 不行

b.call 方法可以抛出异常，Runnable 的 run 方法不行

c.可以通过运行 Callable 得到的 Fulture 对象监听目标线程调用 call 方法的结果，得到返回值，（fulture.get(),调用后会阻塞，直到获取到返回值）

4. 线程中断

一般情况下，线程不执行完任务不会退出，但是在有些场景下，我们需要手动控制线程中断结束任务，Java 中有提供线程中断机制相关的 Api,每个线程都一个状态位用于标识当前线程对象是否是中断状态

public boolean isInterrupted() //判断中断标识位是否是 true，不会改变标识位 public void interrupt() //将中断标识位设置为 truepublic static boolean interrupted() //判断当前线程是否被中断，并且该方法调用结束的时候会清空中断标识位

需要注意的是 interrupt（）方法并不会真的中断线程，它只是将中断标识位设置为 true,具体是否要中断由程序来判断，如下，只要线程中断标识位为 false,也就是没有中断就一直执行线程方法

```
new Thread(new Runnable(){
    while(!Thread.currentThread().isInterrupted()){
        //执行线程方法
    }
}).start();
```

前边我们提到了线程的六种状态，New Runnable Blocked Waiting Timed Waiting Terminated，那么在这六种状态下调用线程中断的代码会怎样呢，New 和 Terminated 状态下，线程不会理会线程中断的请求，既不会设置标记位，在 Runnable 和 Blocked 状态下调用 interrupt 会将标志位设置位 true,在 Waiting 和 Timed Waiting 状态下会发生 InterruptedException 异常，针对这个异常我们如何处理？

1.在 catch 语句中通过 interrupt 设置中断状态，因为发生中断异常时，中断标志位会被复位，我们需要重新将中断标志位设置为 true，这样外界可以通过这个状态判断是否需要中断线程

```
try{
    ....
}catch(InterruptedException e){
```

```
    Thread.currentThread().interrupt();
}
```

2.更好的做法是，不捕获异常，直接抛出给调用者处理，这样更灵活

5. Thread 为什么不能用 stop 方法停止线程

从 SUN 的官方文档可以得知，调用 Thread.stop()方法是不安全的，这是因为当调用 Thread.stop()方法时，会发生下面两件事：

1. 即刻抛出 ThreadDeath 异常，在线程的 run()方法内，任何一点都有可能抛出 ThreadDeath Error，包括在 catch 或 finally 语句中。
2. 释放该线程所持有的所有的锁。调用 thread.stop()后导致了该线程所持有的所有锁的突然释放，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。

6. 重入锁与条件对象，同步方法和同步代码块

。。。

7. volatile 关键字

volatile 为实例域的同步访问提供了免锁机制，如果声明一个域为 volatile，那么编译器和虚拟机就直到该域可能被另一个线程并发更新

8. java 内存模型

堆内存是被所有线程共享的运行时内存区域，存在可见性的问题。线程之间共享变量存储在主存中，每个线程都有一个私有的本地内存，本地内存存储了该线程共享变量的副本（本地内存是一个抽象概念，并不真实存在），两个线程要通信的话，首先 A 线程把本地内存更新过的共享变量更新到主存中，然后 B 线程去主存中读取 A 线程更新过的共享变量，也就是说假设线程 A 执行了 i = 1

这行代码更新主线程变量 i 的值，会首先在自己的工作线程中堆变量 i 进行赋值，然后再写入主存当中，而不是直接写入主存

9. 原子性 可见性 有序性

原子性：对基本数据类型的读取和赋值操作是原子性操作，这些操作不可被中断，是一步到位的，例如 x=3 是原子性操作，而 y = x 就不是，它包含两步：第一读取 x，第二将 x 写入工作内存；x++也不是原子性操作，它包含三部，第一，读取 x，第二，对 x 加 1，第三，写入内存。原子性操作的类如：AtomicInteger AtomicBoolean AtomicLong AtomicReference

可见性：指线程之间的可见性，既一个线程修改的状态对另一个线程是可见的。volatile 修饰可以保证可见性，它会保证修改的值会立即被更新到主存，所以对其他线程是可见的，普通的共享变量不能保证可见性，因为被修改后不会立即写入主存，何时被写入主存是不确定的，所以其他线程去读取的时候可能读到的还是旧值

有序性：Java 中的指令重排序（包括编译器重排序和运行期重排序）可以起到优化代码的作用，但是在多线程中会影响到并发执行的正确性，使用 volatile 可以保证有序性，禁止指令重排

volatile 可以保证可见性 有序性，但是无法保证原子性，在某些情况下可以提供优于锁的性能和伸缩性，替代 sychronized 关键字简化代码，但是要严格遵循使用条件。

10. 线程池 ThreadPoolExecutor

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

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

b. 判断工作队列是否已满，没有满则加入工作队列，否则执行下一步

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

11. 线程池的种类

1.FixedThreadPool:可重用固定线程数的线程池，只有核心线程，没有非核心线程，核心线程不会被回收，有任务时，有空闲的核心线程就用核心线程执行，没有则加入队列排队

2.SingleThreadExecutor:单线程线程池，只有一个核心线程，没有非核心线程，当任务到达时，如果没有运行线程，则创建一个线程执行，如果正在运行则加入队列等待，可以保证所有任务在一个线程中按照顺序执行，和 FixedThreadPool 的区别只有数量

3.CachedThreadPool:按需创建的线程池，没有核心线程，非核心线程有 Integer.MAX_VALUE 个，每次提交

任务如果有空闲线程则由空闲线程执行，没有空闲线程则创建新的线程执行，适用于大量的需要立即处理的并且耗时较短的任务

4.ScheduledThreadPoolExecutor:继承自 ThreadPoolExecutor,用于延时执行任务或定期执行任务，核心线程数固定，线程总数为 Integer.MAX_VALUE

12. 线程同步机制与原理，举例说明

为什么需要线程同步？当多个线程操作同一个变量的时候，存在这个变量何时对另一个线程可见的问题，也就是可见性。每一个线程都持有主存中变量的一个副本，当他更新这个变量时，首先更新的是自己线程中副本的变量值，然后会将这个值更新到主存中，但是是否立即更新以及更新到主存的时机是不确定的，这就导致当另一个线程操作这个变量的时候，他从主存中读取的这个变量还是旧的值，导致两个线程不同步的问题。线程同步就是为了保证多线程操作的可见性和原子性，比如我们用 synchronized 关键字包裹一端代码，我们希望这段代码执行完成后，对另一个线程立即可见，另一个线程再次操作的时候得

到的是上一个线程更新之后的内容，还有就是保证这段代码的原子性，这段代码可能涉及到了好几部操作，我们希望这好几步的操作一次完成不会被中间打断，锁的同步机制就可以实现这一点。一般说的 synchronized 用来做多线程同步功能，其实 synchronized 只是提供多线程互斥，而对象的 wait()和 notify()方法才提供线程的同步功能。JVM 通过 Monitor 对象实现线程同步，当多个线程同时请求 synchronized 方法或块时，monitor 会设置几个虚拟逻辑数据结构来管理这些多线程。新请求的线程会首先被加入到线程排队队列中，线程阻塞，当某个拥有锁的线程 unlock 之后，则排队队列里的线程竞争上岗（synchronized 是不公平竞争锁，下面还会讲到）。如果运行的线程调用对象的 wait()后就释放锁并进入 wait 线程集合那边，当调用对象的 notify()或 notifyall()后，wait 线程就到排队那边。这是大致的逻辑。

13. arrayList 与 linkedList 的读写时间复杂度

（1）ArrayList：ArrayList 是一个泛型类，底层采用数组结构保存对象。数组结构的优点是便于对集合进行快速的随机访问，即如果需要经常根据索引位置访问集合中的对象，使用由 ArrayList 类实现的 List 集合的效率较好。数组结构的缺点是向指定索引位置插入对象和删除指定索引位置对象的速度较慢，并且插入或删除对象的索引位置越小效率越低，原因是当向指定的索引位置插入对象时，会同时将指定索引位置及之后的所有对象相应的向后移动一位。

（2）LinkedList：LinkedList 是一个泛型类，底层是一个双向链表，所以它在执行插入和删除操作时比 ArrayList 更加的高效，但也因为链表的数据结构，所以在随机访问方面要比 ArrayList 差。

ArrayList 是线性表（数组）
get() 直接读取第几个下标，复杂度 O(1)
add(E) 添加元素，直接在后面添加，复杂度 O（1）
add(index, E) 添加元素，在第几个元素后面插入，后面的元素需要向后移动，复杂度 O（n）
remove（）删除元素，后面的元素需要逐个移动，复杂度 O（n）

LinkedList 是链表的操作

get() 获取第几个元素，依次遍历，复杂度 O(n)

add(E) 添加到末尾，复杂度 O(1)

add(index, E) 添加第几个元素后，需要先查找到第几个元素，直接指针指向操作，复杂度 O(n)

remove（）删除元素，直接指针指向操作，复杂度 O(1)

14. 为什么 HashMap 线程不安全（hash 碰撞与扩容导致）

HashMap 的底层存储结构是一个 Entry 数组，每个 Entry 又是一个单链表，一旦发生 Hash 冲突的的时候，HashMap 采用拉链法解决碰撞冲突，因为 hashMap 的 put 方法不是同步的，所以他的扩容方法也不是同步的，在扩容过程中，会新生成一个新的容量的数组，然后对原数组的所有键值对重新进行计算和写入新的数组，之后指向新生成的数组。当多个线程同时检测到 hashmap 需要扩容的时候就会同时调用 resize 操作，各自生成新的数组并 rehash 后赋给该 map 底层的数组 table，结果最终只有最后一个线程生成的新数组被赋给 table 变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的 table 作为原始数组，这样也会有问题。扩容的时候可能会引发链表形成环状结构

15. 进程线程的区别

1.地址空间：同一进程的线程共享本进程的地址空间，而进程之间则是独立的地址空间。

2.资源拥有：同一进程内的线程共享本进程的资源如内存、I/O、cpu 等，但是进程之间的资源是独立的。

3.一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

4.进程切换时，消耗的资源大，效率不高。所以涉及到频繁的切换时，使用线程要好于进程。同样如果要求同时进行并且又要共享某些变量的并发操作，只

能用线程不能用进程

5.执行过程：每个独立的进程程有一个程序运行的入口、顺序执行序列和程序入口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

6.线程是处理器调度的基本单位，但是进程不是。

7.两者均可并发执行。

16. Binder 的内存拷贝过程

相比其他的 IPC 通信，比如消息机制、共享内存、管道、信号量等，Binder 仅需一次内存拷贝，即可让目标进程读取到更新数据，同共享内存一样相当高效，其他的 IPC 通信机制大多需要 2 次内存拷贝。Binder 内存拷贝的原理为：进程 A 为 Binder 客户端，在 IPC 调用前，需将其用户空间的数据拷贝到 Binder 驱动的内核空间，由于进程 B 在打开 Binder 设备(/dev/binder)时，已将 Binder 驱动的内核空间映射(mmap)到自己的进程空间，所以进程 B 可以直接看到 Binder 驱动内核空间的内容改动

17. 传统 IPC 机制的通信原理（2 次内存拷贝）

1.发送方进程通过系统调用（copy_from_user）将要发送的数据存拷贝到内核缓存区中。

2.接收方开辟一段内存空间，内核通过系统调用（copy_to_user）将内核缓存区中的数据拷贝到接收方的内存缓存区。

种传统 IPC 机制存在 2 个问题：

1.需要进行 2 次数据拷贝，第 1 次是从发送方用户空间拷贝到内核缓存区，第 2 次是从内核缓存区拷贝到接收方用户空间。

2.接收方进程不知道事先要分配多大的空间来接收数据，可能存在空间上的浪费。

18. Java 内存模型（记住堆栈是内存分区，不是模型）

Java 内存模型(即 Java Memory Model，简称 JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。由于 JVM 运行程序的实体是线程，而每个线程创建时 JVM 都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而 Java 内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝，前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成

19. 类的加载过程

类加载过程主要包含加载、验证、准备、解析、初始化、使用、卸载七个方面，下面一一阐述。

1.加载：获取定义此类的二进制字节流，生成这个类的 java.lang.Class 对象

2.验证：保证 Class 文件的字节流包含的信息符合 JVM 规范，不会给 JVM 造成危害

3.准备：准备阶段为变量分配内存并设置类变量的初始化

4.解析：解析过程是将常量池内的符号引用替换成直接引用

5.初始化：不同于准备阶段，本次初始化，是根据程序员通过程序制定的计划去初始化类的变量和其他资源。这些资源有 static{}块，构造函数，父类的初始化等

6.使用：使用过程就是根据程序定义的行为执行

7.卸载：卸载由 GC 完成。

20. 什么情况下会触发类的初始化

1、 遇到 new，getstatic，putstatic，invokestatic 这 4 条指令；

2、 使用 java.lang.reflect 包的方法对类进行反射调用；

3、 初始化一个类的时候，如果发现其父类没有进行过初始化，则先初始化其父类（注意！如果其父类是接口的话，则不要求初始化父类）；

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 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getstatic，REF_putstatic,REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则先触发其类初始化；

21. 双亲委托模式

类加载器查找 class 所采用的是双亲委托模式，所谓双亲委托模式就是判断该类是否已经加载，如果没有则不是自身去查找而是委托给父加载器进行查找，这样依次进行递归，直到委托到最顶层的 Bootstrap ClassLoader,如果 Bootstrap ClassLoader 找到了该 Class,就会直接返回，如果没找到，则继续依次向下查找，如果还没找到则最后交给自身去查找

22. 双亲委托模式的好处

1.避免重复加载，如果已经加载过一次 Class，则不需要再次加载，而是直接读取已经加载的 Class

2.更加安全，确保，java 核心 api 中定义类型不会被随意替换，比如，采用双亲委托模式可以使得系统在 Java 虚拟机启动时旧加载了 String 类，也就无法用自定义的 String 类来替换系统的 String 类，这样便可以防止核心 API 库被随意篡改。

23. 死锁的产生条件，如何避免死锁

死锁的四个必要条件

1.互斥条件：一个资源每次只能被一个进程使用

2.请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
3.不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。
4.循环等待条件: 若干进程间形成首尾相接循环等待资源的关系
这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

避免死锁的方法：系统对进程发出每一个系统能够满足的资源申请进行动态检查,并根据检查结果决定是否分配资源,如果分配后系统可能发生死锁,则不予分配,否则予以分配，这是一种保证系统不进入死锁状态的动态策略。

在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免发生死锁。 一般来说互斥条件是无法破坏的，所以在预防死锁时主要从其他三个方面入手 ：
(1)破坏请求和保持条件：在系统中不允许进程在已获得某种资源的情况下，申请其他资源，即要想出一个办法，阻止进程在持有资源的同时申请其它资源。
方法一：在所有进程开始运行之前，必须一次性的申请其在整个运行过程中所需的全部资源，
方法二：要求每个进程提出新的资源申请前，释放它所占有的资源

(2)破坏不可抢占条件：允许对资源实行抢夺。
方式一：如果占有某些资源的一个进程进行进一步资源请求被拒绝，则该进程必须释放它最初占有的资源，如果有必要，可再次请求这些资源和另外的资源。
方式二：如果一个进程请求当前被另一个进程占有的资源，则操作系统可以抢占另一个进程，要求它释放资源，只有在任意两个进程的优先级都不相同的条件下，该方法才能预防死锁。

(3)破坏循环等待条件

对系统所有资源进行线性排序并赋予不同的序号，这样我们便可以规定进程在申请资源时必须按照序号递增的顺序进行资源的申请，当以后要申请时需检查要申请的资源的编号大于当前编号时，才能进行申请。

利用银行家算法避免死锁：

所谓银行家算法，是指在分配资源之前先看清楚，资源分配后是否会导致系统死锁。如果会死锁，则不分配，否则就分配。

按照银行家算法的思想，当进程请求资源时，系统将按如下原则分配系统资源：

24. App 启动流程

App 启动时，AMS 会检查这个应用程序所需要的进程是否存在，不存在就会请求 Zygote 进程启动需要的应用程序进程，Zygote 进程接收到 AMS 请求并通过 fock 自身创建应用程序进程，这样应用程序进程就会获取虚拟机的实例，还会创建 Binder 线程池（ProcessState.startThreadPool()）和消息循环(ActivityThread looper.loop),然后 App 进程，通过 Binder IPC 向 sytem_server 进程发起 attachApplication 请求；system_server 进程在收到请求后，进行一系列准备工作后，再通过 Binder IPC 向 App 进程发送 scheduleLaunchActivity 请求；App 进程的 binder 线程（ApplicationThread）在收到请求后，通过 handler 向主线程发送 LAUNCH_ACTIVITY 消息；主线程在收到 Message 后，通过反射机制创建目标 Activity，并回调 Activity.onCreate()等方法。到此，App 便正式启动，开始进入 Activity 生命周期，执行完 onCreate/onStart/onResume 方法，UI 渲染结束后便可以看到 App 的主界面。

25. Android 单线程模型

Android 单线程模型的核心原则就是：只能在 UI 线程(Main Thread)中对 UI 进行处理。当一个程序第一次启动时，Android 会同时启动一个对应的 主线程（Main Thread），主线程主要负责处理与 UI 相关的事件，如：用户的按键事

件，用户接触屏幕的事件以及屏幕绘图事 件，并把相关的事件分发到对应的组件进行处理。所以主线程通常又被叫做 UI 线 程。在开发 Android 应用时必须遵守单线程模型的原则： Android UI 操作并不是线程安全的并且这些操作必须在 UI 线程中执行。

Android 的单线程模型有两条原则：

1.不要阻塞 UI 线程。

2.不要在 UI 线程之外访问 Android UI toolkit（主要是这两个包中的组件：android.widget and android.view

26. RecyclerView 在很多方面能取代 ListView，Google 为什么没把 ListView 划上一条过时的横线？

ListView 采用的是 RecyclerBin 的回收机制在一些轻量级的 List 显示时效率更高。

27. HashMap 如何保证元素均匀分布

hash & (length-1)

通过 Key 值的 hashCode 值和 hashMap 长度-1 做与运算

hashmap 中的元素，默认情况下，数组大小为 16，也就是 2 的 4 次方，如果要自定义 HashMap 初始化数组长度，也要设置为 2 的 n 次方大小，因为这样效率最高。因为当数组长度为 2 的 n 次幂的时候，不同的 key 算出的 index 相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了