目录

1、socket 判断 http 请求或 http 响应的传输结束？

【参考】：先把 header 直到\r\n\r\n 整个地址记录下来

1、如果是短连接，没有启用 keepalive，则可以通过是否关闭了连接来判断是否传输结束，即在读取时可判断 read() != -1。传输完毕就关闭 connection，即 recv 收到 0 个字节。

2、如果时长连接，那么一个 socket（tcp）可能发送和接收多次请求，那么如何判断每次的响应已经接收?

2-1、先读请求头，一直到\r\n\r\n 说明请求头结束，然后解析 http 头，如果 Content-Length=x 存在，则知道 http 响应的长度为 x。从头的末尾直接读取 x 字节就是响应内容。

2-2、如果 Content-Length=x 不存在，那么头类型为 Transfer-Encoding: chunked 说明响应的长度不固定，则在响应头结束后标记第一段流的长度，即直到流里有\r\n0\r\n\r\n 结束

2-3.、如果 recv 返回 SOCKET_ERROR 时，说明对方已经断开连接，但是可能是非正常断开(断网或者客户端进程结束).

2、Glide 源码分析

【参考】https://blog.csdn.net/guolin_blog/article/details/54895665

3、Android UI 线程和非 UI 线程的交互方式有哪些？

【参考】https://www.cnblogs.com/carlo/articles/4986734.html

4、Java ArrayList、LinkedList、 Vector 和数组的区别

【参考】
Vector 是多线程安全的，而 ArrayList 不是，这个可以从源码中看出，Vector 类中的方法很

多有 synchronized 进行修饰，这样就导致了 Vector 在效率上无法与 ArrayList 相比；

两个都是采用的线性连续空间存储元素，但是当空间不足的时候，两个类的增加方式是不同的，很多网友说 Vector 增加原来空间的一倍，ArrayList 增加原来空间的 50%，其实也差不多是这个意思，不过还有一点点问题可以从源码中看出，Vector 可以设置增长因子，而 ArrayList 不可以

ArrayList 是实现了基于动态数组的数据结构，LinkedList 基于链表的数据结构。
对于随机访问 get 和 set，ArrayList 觉得优于 LinkedList，因为 LinkedList 要移动指针
对于新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动数据。
所以： 有一个列表，要对其进行大量的插入和删除操作，在这种情况下 LinkedList 就是一个较好的选择； 需要随机地访问其中的元素时，使用 ArrayList 会提供比较好的性能

ArrayList 是一个可改变大小的数组.当更多的元素加入到 ArrayList 中时,其大小将会动态地增长.内部的元素可以直接通过 get 与 set 方法进行访问,因为 ArrayList 本质上就是一个数组.

LinkedList 是一个双链表,在添加和删除元素时具有比 ArrayList 更好的性能.但在 get 与 set 方面弱于 ArrayList.当然,这些对比都是指数据量很大或者操作很频繁的情况下的对比,如果数据和运算量很小,那么对比将失去意义.

Vector 和 ArrayList 类似,但属于强同步类。如果你的程序本身是线程安全的(thread-safe,没有在多个线程之间共享同一个集合/对象),那么使用 ArrayList 是更好的选择

5、Java 线程的生命周期

【参考】

一个线程的产生是从我们调用了 start 方法开始进入 Runnable 状态，即可以被调度运行状态，并没有真正开始运行，调度器可以将 CPU 分配给它，使线程进入 Running 状态，真正运行其中的程序代码。线程在运行过程中，有以下几个可能的去向：

（1）调度器在某个线程的执行过程中将 CPU 分配给了其它线程，则这个线程又变为 Runnable 状态，等待被调度。

（2）调度器将 CPU 分配给了该线程，执行过程中没有遇到任何阻隔，运行完成直接结束，也就是 run()方法执行完毕。

（3）线程在执行过程中请求锁，却得不到，这时候它要进入 lock pool 中等待对象的锁，等到锁后又会进入 Runnable 状态，可以被调度运行。

（4）线程在执行过程中遇到 wait()方法，它会被放入 wait pool 中等待，直到有 notify()或 interrupt()方法执行，它才会被唤醒或打断进入 lock pool 开始等待对象锁，等到锁后进入 Runnable 状态。

推荐在 run 方法中使用控制循环条件的方式来结束一个线程。

wait: 告诉当前线程放弃对象锁并进入等待状态, 直到其他线程进入同一对象锁并调用 notify 为止。

notify：唤醒同一对象锁中调用 wait 的第一个线程。

notifyAll：唤醒同一对象锁中调用 wait 的所有线程，具有最高优先级的线程首先被唤醒并执行。

yield：如果知道已经完成了在 run()方法的循环的一次迭代过程中所需要的工作，就可以给线程调度一个机制暗示：我的工作已经做的差不多了，可以让给别的线程使用 CPU 了。通过调用 yield()来实现。
当调用 yield 时，你也是在建议具有相同优先级的其他线程可以运行。
对于任何重要的控制或在调整应用时，都不恩那个依赖于 yield。实际上，yield 经常被误用。（yield 并不意味着退出和暂停，只是，告诉线程调度如果有人需要，可以先拿去，我过会再执行，没人需要，我继续执行）
调用 yield 的时候锁并没有被释放。

interrupt 简述

interrupt() 方法只是改变中断状态而已，它不会中断一个正在运行的线程。这一方法实际完成的是，给受阻塞的线程发出一个中断信号，这样受阻线程就得以退出阻塞的状态。 更确切的说，如果线程被 Object.wait, Thread.join 和 Thread.sleep 三种方法之一阻塞，此时调用该线程的 interrupt()方法，那么该线程将抛出一个 InterruptedException 中断异常（该线程必须事先预备好处理此异常），从而提早地终结被阻塞状态。如果线程没有被阻塞，这时调用 interrupt() 将不起作用，直到执行到 wait(),sleep(),join() 时，才马上会抛出 InterruptedException。

线 程 A 在执行 sleep,wait,join 时,线程 B 调用线程 A 的 interrupt 方法,的确这一个时候 A 会有 InterruptedException 异常抛出来。但这其实是在 sleep,wait,join 这些方法内部会不断检查中断状态的值,而自己抛出的 InterruptedException。 如果线程 A 正在执行一些指定的操作时如值,for,while,if, 调用方法等, 不会去检查中断状态, 则线程 A 不会抛出 InterruptedException,而会一直执行着自己的操作。

注意:

当线程 A 执行到 wait(),sleep(),join()时,抛出 InterruptedException 后，中断状态已经被系统复位了，线程 A 调用 Thread.interrupted()返回的是 false。

如果线程被调用了 interrupt()，此时该线程并不在 wait(),sleep(),join()时，下次执行 wait(),sleep(),join()时，一样会抛出 InterruptedException，当然抛出后该线程的中断状态也会被系统复位。

1. sleep() &interrupt()

线程 A 正在使用 sleep()暂停着: Thread.sleep(100000)，如果要取消它的等待状态,可以在正在执行的线程里(比如这里是 B)调用 a.interrupt()［a 是线程 A 对应到的 Thread 实例］，令线程 A 放弃睡眠操作。即，在线程 B 中执行 a.interrupt()，处于阻塞中的线程 a 将放弃睡眠操作。

当在 sleep 中时线程被调用 interrupt()时，就马上会放弃暂停的状态并抛出 InterruptedException。抛出异常的,是 A 线程。

2. wait() &interrupt()

线程 A 调用了 wait()进入了等待状态，也可以用 interrupt()取消。不过这时候要注意锁定的问题。线程在进入等待区,会把锁定解除,当对等待中的线程调用 interrupt()时，会先重新获取锁定，再抛出异常。在获取锁定之前，是无法抛出异常的。

3. join() &interrupt()

当线程以 join()等待其他线程结束时，当它被调用 interrupt()，它与 sleep()时一样，会马上跳到 catch 块里.。

注意，调用的 interrupt()方法，一定是调用被阻塞线程的 interrupt 方法。如在线程 a 中调用线程 t.interrupt()。

6、EventBus 源码分析

【参考】

https://www.jianshu.com/p/bda4ed3017ba

https://www.jianshu.com/p/f057c460c77e

7、BlockCanary 核心原理分析

【参考】

```
public static void loop() {
        final Looper me = myLooper();
        if (me == null) {
            throw new RuntimeEx ception("No Looper; Looper.prepare() wasn't called on this thread.");
```

```
}
final MessageQueue queue = me.mQueue;

// Make sure the identity of this thread is that of the local process,
// and keep track of what that identity token actually is.
Binder.clearCallingIdentity();
final long ident = Binder.clearCallingIdentity();

for (;;) {
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is quitting.
        return;
    }

    // This must be in a local variable, in case a UI event sets the logger
    final Printer logging = me.mLogging;
    if (logging != null) {
        logging.println(">>>>> Dispatching to " + msg.target + " " +
                msg.callback + ": " + msg.what);
    }

    final long traceTag = me.mTraceTag;
    if (traceTag != 0 && Trace.isTagEnabled(traceTag)) {
        Trace.traceBegin(traceTag, msg.target.getTraceName(msg));
    }
    try {
        msg.target.dispatchMessage(msg);
    } finally {
        if (traceTag != 0) {
            Trace.traceEnd(traceTag);
        }
    }

    if (logging != null) {
        logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
    }

    // Make sure that during the course of dispatching the
    // identity of the thread wasn't corrupted.
    final long newIdent = Binder.clearCallingIdentity();
    if (ident != newIdent) {
        Log.wtf(TAG, "Thread identity changed from 0x"
                + Long.toHexString(ident) + " to 0x"
```

```
                        + Long.toHexString(newIdent) + " while dispatching to "
                        + msg.target.getClass().getName() + " "
                        + msg.callback + " what=" + msg.what);
            }

            msg.recycleUnchecked();
        }
    }
```

原理：我们只需要计算打印这两天 log 的时间差，就能得到 dispatchMessage 的耗时，android 提供了 Looper.getMainLooper().setMessageLogging(Printer printer)来设置这个 logging 对象，所以只要自定义一个 Printer，然后重写 println(String x)方法即可实现耗时统计了。

通过在 DispatchMessage 的方法的执行时间，来判断卡顿，管是哪种回调方式，回调一定发生在 UI 线程。因此如果应用发生卡顿，一定是在 dispatchMessage 中执行了耗时操作。我们通过给主线程的 Looper 设置一个 Printer，打点统计 dispatchMessage 方法执行的时间，如果超出阀值，表示发生卡顿，则 dump 出各种信息，提供开发者分析性能瓶颈。
mBlockListener.onBlockEvent

https://blog.csdn.net/lhd201006/article/details/79044497

https://www.jianshu.com/p/e58992439793

8、一般我们说的 Bitmap 究竟占多大内存?

【参考】
options.inJustDecodeBounds = true;//只读取图片，不加载到内存中
options.inJustDecodeBounds = false;//加载到内存中

https://blog.csdn.net/lsyz0021/article/details/51356670

9、Android 内存优化总结&实践

【参考】https://mp.weixin.qq.com/s/2MsEAR9pQfMr1Sfs7cPdWQ

10、Android 中内存泄漏如何排查?

【参考】https://www.jianshu.com/p/ab4a7e353076

11、Java 中 sleep,wait,yield,join 的区别?

【参考】

1、sleep()方法
在指定时间内让当前正在执行的线程暂停执行，但不会释放“锁标志”。不推荐使用。
sleep()使当前线程进入阻塞状态，在指定时间内不会执行。

2、wait()方法
在其他线程调用对象的 notify 或 notifyAll 方法前，导致当前线程等待。线程会释放掉它所占有的“锁标志”，从而使别的线程有机会抢占该锁。当前线程必须拥有当前对象锁。如果当前线程不是此锁的拥有者，会抛出 IllegalMonitorStateException 异常。唤醒当前对象锁的等待线程使用 notify 或 notifyAll 方法，也必须拥有相同的对象锁，否则也会抛出 IllegalMonitorStateException 异常。

wait()和 notify()必须在 synchronized 函数或 synchronized block 中进行调用。如果在 non-synchronized 函数或 non-synchronized block 中进行调用，虽然能编译通过，但在运行时会发生 IllegalMonitorStateException 的异常。

3、yield 方法

暂停当前正在执行的线程对象。
yield()只是使当前线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状态后马上又被执行。
yield()只能使同优先级或更高优先级的线程有执行的机会。?
调用 yield 方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取 CPU 执行时间，这一点是和 sleep 方法不一样的。

4.join 方法

等待该线程终止。
等待调用 join 方法的线程结束，再继续执行。如：t.join();//主要用于等待 t 线程运行结束，若无此句，main 则会执行完毕，导致结果不可预测。
在很多情况下，主线程创建并启动了线程，如果子线程中药进行大量耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到 join()方法了。方法 join()的作用是等待线程对象销毁。

12、view 绘制过程，介绍 draw/onDraw 和 drawChild

【参考】https://blog.csdn.net/qq_27073205/article/details/46240293

13、垃圾回收 什么时候触发 GC?

【参考】https://blog.csdn.net/sunny243788557/article/details/52797088

14、OnLowMemory 和 OnTrimMemory 的比较

【参考】OnLowMemory 被回调时，已经没有后台进程；而 onTrimMemory 被回调时，还有后台进程。

OnLowMemory 是在最后一个后台进程被杀时调用，一般情况是 low memory killer 杀进程后触发；而 OnTrimMemory 的触发更频繁，每次计算进程优先级时，只要满足条件，都会触发。

通过一键清理后，OnLowMemory 不会被触发，而 OnTrimMemory 会被触发一次。

15、onRestart 什么时候调用?

【参考】

1、按下 home 键执行，再次打开这个 demo 执行；onRestart()--->onStart()--->onResume()三个方法。

2、点击界面的 btn，跳转到另一个 Activity1，从 Activity1 返回，会执行如下图 2；onRestart()--->onStart()--->onResume()三个方法

3、切换到其他的应用，从其他应用切换回来。onRestart()--->onStart()--->onResume()三个方法

16、Handler 如何防止内存泄漏?

【参考】

除了写弱引用这个方法后，还有一个就是 handler.removeCallbacksAndMessages(null);，就是移除所有的消息和回调，简单一句话就是清空了消息队列。注意，不要以为你 post 的是个 Runnable 或者只是 sendEmptyMessage。你可以看一下源码，在 handler 里面都是会把这些转成正统的 Message，放入消息队列里面，所以清空队列就意味着这个 Handler 直接被打成原型了，当然也就可以回收了。

17、JNI 和 Java 怎么互相传递数据?

【参考】 https://blog.csdn.net/qq_26222859/article/details/80900125

18、TCP 可靠性的保证机制总结

【参考】

1、检验和
2、序列号
3、确认应答机制（ACK）
4、超时重传机制
5、连接管理机制
6、流量控制
7、拥塞控制

https://blog.csdn.net/xuzhangze/article/details/80490362

19、synchronized 和 ReentrantLock 的区别和使用选择

【参考】

1、使用 synchronized 获得的锁存在一定缺陷：

>不能中断一个正在试图获得锁的线程

>试图获得锁时不能像 ReentrantLock 中的 trylock 那样设定超时时间 ，当一个线程获得了对象锁后，其他线程访问这个同步方法时，必须等待或阻塞，如果那个线程发生了死循环，对象锁就永远不会释放；

>每个锁只有单一的条件，不像 condition 那样可以设置多个

2、尽管 synchronized 存在上述的一些缺陷，在选择上还是以 synchronized 优先：

>如果 synchronized 关键字适合程序，尽量使用它，可以减少代码出错的几率和代码数量；（减少出错几率是因为在执行完 synchronized 包含完的最后一句语句后，锁会自动释放，不需要像 ReentrantLock 一样手动写 unlock 方法；）

>如果特别需要 Lock/Condition 结构提供的独有特性时，才使用他们；（比如设定一个线程长时间不能获取锁时设定超时时间或自我中断等功能。）

>许多情况下可以使用 java.util.concurrent 包中的一种机制，它会为你处理所有的加锁情况；（比如当我们在多线程环境下使用 HashMap 时，可以使用 ConcurrentHashMap 来处理多线程并发）

20、synchronized，volitale 的异同

【参考】
1、volatile 本质是在告诉 jvm 当前变量在寄存器中的值是不确定的,需要从主存中读取,synchronized 则是锁定当前变量,只有当前线程可以访问该变量,其他线程被阻塞住.
2、volatile 仅能使用在变量级别,synchronized 则可以使用在变量,方法.
3、volatile 仅能实现变量的修改可见性,但不具备原子特性,而 synchronized 则可以保证变量的修改可见性和原子性.
4、volatile 不会造成线程的阻塞,而 synchronized 可能会造成线程的阻塞.
5、volatile 标记的变量不会被编译器优化,而 synchronized 标记的变量可以被编译器优化.

21、okhttp 里的设计模式

【参考】
构建对象时，使用 Builder 设计模式
Interceptor，使用责任链模式

https://www.jianshu.com/p/5cd6775cbb51

22、SparseArray 和 ArrayMap 使用场景?

【参考】假设数据量都在千级以内的情况下:

1、假设 key 的类型已经确定为 int 类型。那么使用 SparseArray，由于它避免了自己主动装箱的过程，假设 key 为 long 类型，它还提供了一个 LongSparseArray 来确保 key 为 long 类型时的使用

2、假设 key 类型为其他的类型，则使用 ArrayMap

SparseArray 比 HashMap 更省内存，在某些条件下性能更好，主要是由于它避免了对 key 的自己主动装箱（int 转为 Integer 类型），它内部则是通过两个数组来进行数据存储的。一个存储 key，另外一个存储 value，为了优化性能，它内部对数据还採取了压缩的方式来表示稀疏数组的数据，从而节约内存空间。我们从源代码中能够看到 key 和 value 各自是用数组表示:

https://www.cnblogs.com/yjbjingcha/p/7074266.html

23、Android 中 getRawX()和 getX()区别

【参考】

getRawX( )即表示的是点击的位置距离屏幕的坐标
getX( )即表示的点击的位置相对于本身的坐标

24、Java GC 如何判断对象是否为垃圾?

【参考】查找内存中不再使用的对象

1、引用计数法

引用计数法就是如果一个对象没有被任何引用指向，则可视之为垃圾。这种方法的缺点就是不能检测到环的存在。

2、根搜索算法

根搜索算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。

https://www.cnblogs.com/hzzjj/p/6268432.html

25、Java 中存储机制-堆栈

【参考】https://www.cnblogs.com/zyj-bozhou/p/6723863.html

26、怎么自定义 RecyclerView.LayoutManager

【参考】https://www.jianshu.com/p/1837a801e599

27、Android 在 MotionEvent 手势事件

【参考】其中包括:

MotionEvent.ACTION_DOWN：当屏幕检测到第一个触点按下之后就会触发到这个事件。

MotionEvent.ACTION_MOVE：当触点在屏幕上移动时触发，触点在屏幕上停留也是会触发的，主要是由于它的灵敏度很高，而我们的手指又不可能完全静止（即使我们感觉不到移动，但其实我们的手指也在不停地抖动）。

MotionEvent.ACTION_POINTER_DOWN：当屏幕上已经有触点处于按下的状态的时候，再有新的触点被按下时触发。

MotionEvent.ACTION_POINTER_UP：当屏幕上有多个点被按住，松开其中一个点时触发（即非最后一个点被放开时）触发。

MotionEvent.ACTION_UP：当最后一个触点松开时被触发。

MotionEvent.ACTION_SCROLL：非触摸滚动，主要是由鼠标、滚轮、轨迹球触发。

MotionEvent.ACTION_CANCEL：不是由用户直接触发，有系统再需要的时候触发，例如当父 view 通过使函数 onInterceptTouchEvent()返回 true,从子 view 拿回处理事件的控制权是，就会给子 view 发一个 ACTION_CANCEL 事件，这里了 view 就再也不会收到事件了。可以将其视为 ACTION_UP 事件对待。

onInterceptTouchEvent()函数与 onTouchEvent()的区别:

1、onInterceptTouchEvent()是用于处理事件（类似于预处理，当然也可以不处理）并改变事件的传递方向，也就是决定是否允许 Touch 事件继续向下（子 view）传递，一但返回 True（代表事件在当前的 viewGroup 中会被处理），则向下传递之路被截断（所有子 view 将没有机会参与 Touch 事件），同时把事件传递给当前的 view 的 onTouchEvent()处理；返回 false，则把事件交给子 view 的 onInterceptTouchEvent()

2、onTouchEvent()用于处理事件，返回值决定当前 view 是否消费（consume）了这个事件，也就是说在当前 view 在处理完 Touch 事件后，是否还允许 Touch 事件继续向上（父 view）传递，一但返回 True，则父 view 不用操心自己来处理 Touch 事件。返回 true，则向上传递给父 view（注：可能你会觉得是否消费了有关系吗，反正我已经针对事件编写了处理代码？答案是有区别！比如 ACTION_MOVE 或者 ACTION_UP 发生的前提是一定曾经发生了 ACTION_DOWN，如果你没有消费 ACTION_DOWN，那么系统会认为 ACTION_DOWN 没有发生过，所以 ACTION_MOVE 或者 ACTION_UP 就不能被捕获。）

28、Http 1.0 和 Http 2.0 区别？

【参考】HTTP1.0

无状态、无连接

HTTP1.1

持久连接
请求管道化
增加缓存处理（新的字段如 cache-control）
增加 Host 字段、支持断点传输等（把文件分成几部分）

HTTP2.0
二进制分帧
多路复用（或连接共享）
头部压缩

29、View.getLocationInWindow 和 View.getLocationOnScreen 区别？

【参考】一个控件在其父窗口中的坐标位置

View.getLocationInWindow(int[] location)

一个控件在其整个屏幕上的坐标位置

View.getLocationOnScreen(int[] location)

30、http 和 https 的区别?

【参考】https 通信过程
1、在使用 HTTPS 是需要保证服务端配置正确了对应的安全证书
2、客户端发送请求到服务端
3、服务端返回公钥和证书到客户端
4、客户端接收后会验证证书的安全性,如果通过则会随机生成一个随机数,用公钥对其加密,发送到服务端
5、服务端接受到这个加密后的随机数后会用私钥对其解密得到真正的随机数,随后用这个随机数当做私钥对需要发送的数据进行对称加密
6、客户端在接收到加密后的数据使用私钥(即生成的随机值)对数据进行解密并且解析数据呈现结果给客户
7、SSL 加密建立

HTTPS 和 HTTP 的区别主要如下:
1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。
2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。
3、http 和 https 使用的是完全不同的连接方式, 用的端口也不一样, 前者是 80, 后者是 443。
4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

https://www.cnblogs.com/liyuhui-Z/p/7844880.html

31、为什么要有内部类？静态内部类和普通内部类区别是什么?

【参考】

1、内部类一般只为其外部类使用;
2、内部类提供了某种进入外部类的窗户;
3、也是最吸引人的原因，每个内部类都能独立地继承一个接口，而无论外部类是否已经继承了某个接口。因此，内部类使多重继承的解决方案变得更加完整。

内部类和外部类区别?

1、内部类是一个编译时概念，编译后外部类及其内部类会生成两个独立的 class 文件：OuterClass.class 和 OuterClass$InnerClass.class
2、内部类可以直接访问外部类的元素，但是外部类不可以直接访问内部类的元素

3、外部类可以通过内部类引用间接访问内部类元素

4、对于静态内部类来说，静态内部类是不依赖于外部类的，也就说可以在不创建外部类对象的情况下创建内部类的对象。另外，静态内部类是不持有指向外部类对象的引用的

静态内部类和普通内部类

静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内。
但是静态内部类却没有。没有这个引用就意味着：

静态内部类的创建是不需要依赖于外围类，可以直接创建
静态内部类不可以使用任何外围类的非 static 成员变量和方法，而内部类则都可以

注意：
1.非静态内部类中不允许定义静态成员
2.外部类的静态成员不可以直接使用非静态内部类
3.静态内部类，不能访问外部类的实例成员，只能访问外部类的类成

32、HashMap 在多线程时，有什么问题？以及 ConcurrentHashMap 的使用

【参考】一句话总结就是，并发环境下的 rehash 过程可能会带来循环链表，导致死循环致使线程挂掉。

因此并发环境下，建议使用 Java.util.concurrent 包中的 ConcurrentHashMap 以保证线程安全。

至于 HashTable，它并未使用分段锁，而是锁住整个数组，高并发环境下效率非常的低，会导致大量线程等待。同样的，Synchronized 关键字、Lock 性能都不如分段锁实现的 ConcurrentHashMap。

33、onStartCommand 的几种模式

【参考】START_NOT_STICKY
如果返回 START_NOT_STICKY，表示当 Service 运行的进程被 Android 系统强制杀掉之后，不会重新创建该 Service。当然如果在其被杀掉之后一段时间又调用了 startService，那么该 Service 又将被实例化。那什么情境下返回该值比较恰当呢?
如果我们某个 Service 执行的工作被中断几次无关紧要或者对 Android 内存紧张的情况下需要被杀掉且不会立即重新创建这种行为也可接受，那么我们便可将 onStartCommand 的返回值设置为 START_NOT_STICKY。

举个例子，某个 Service 需要定时从服务器获取最新数据：通过一个定时器每隔指定的 N 分钟让定时器启动 Service 去获取服务端的最新数据。当执行到 Service 的 onStartCommand 时，在该方法内再规划一个 N 分钟后的定时器用于再次启动该 Service 并开辟一个新的线程去执行网络操作。假设 Service 在从服务器获取最新数据的过程中被 Android 系统强制杀掉，Service 不会再重新创建，这也没关系，因为再过 N 分钟定时器就会再次启动该 Service 并重新获取数据。

START_STICKY
如果返回 START_STICKY，表示 Service 运行的进程被 Android 系统强制杀掉之后，Android 系统会将该 Service 依然设置为 started 状态（即运行状态），但是不再保存 onStartCommand 方法传入的 intent 对象，然后 Android 系统会尝试再次重新创建该 Service，并执行 onStartCommand 回调方法，但是 onStartCommand 回调方法的 Intent 参数为 null，也就是 onStartCommand 方法虽然会执行但是获取不到 intent 信息。如果你的 Service 可以在任意时刻运行或结束都没什么问题，而且不需要 intent 信息，那么就可以在 onStartCommand 方法中返回 START_STICKY，比如一个用来播放背景音乐功能的 Service 就适合返回该值。

START_REDELIVER_INTENT
如果返回 START_REDELIVER_INTENT，表示 Service 运行的进程被 Android 系统强制杀掉之后，与返回 START_STICKY 的情况类似，Android 系统会将再次重新创建该 Service，并执行 onStartCommand 回调方法，但是不同的是，Android 系统会再次将 Service 在被杀掉之前最后一次传入 onStartCommand 方法中的 Intent 再次保留下来并再次传入到重新创建后的 Service 的 onStartCommand 方法中，这样我们就能读取到 intent 参数。只要返回 START_REDELIVER_INTENT，那么 onStartCommand 重的 intent 一定不是 null。如果我们的 Service 需要依赖具体的 Intent 才能运行（需要从 Intent 中读取相关数据信息等），并且在强制销毁后有必要重新创建运行，那么这样的 Service 就适合返回 START_REDELIVER_INTENT

34、Intent 传递数据的限制大小

【参考】https://www.jianshu.com/p/f3dbcc37ce1a

35、RelativeLayout 的 onMeasure 方法，怎么 Measure 的?

【参考】发现 RelativeLayout 会根据 2 次排列的结果对子 View 各做一次 measure。
而在做横向的测量时，纵向的测量结果尚未完成，只好暂时使用 myHeight 传入子 View 系统。
这样必然会导致子 View 的高度和 RelativeLayout 的高度不同时，第 3 点中所说的优化会失效，在 View 系统足够复杂时，效率问题就会出现。

36、RemoteView 内部机制

【参考】:

1、通过 View 的 id 值获取对应的 view(target)。
2、SetOnClickPendingIntent 类中的成员变量 pendingIntent 生成相应的 OnClickListener。
3、为 target 设置监听。

https://blog.csdn.net/a15129095654/article/details/72814528

37、主线程的死循环是否一致耗费 CPU 资源?

【参考】

在主线程使用 Looper.loop 可以保证主线程一直在运行，事实上，在 Looper.loop 死循环之前，已经创建了一个 Binder 线程:

thread.attach 会建立 Binder 通道，创建新线程。attach(false)会创建一个 ApplicaitonThread 的 Binder 线程，用于接受 AMS 发来的消息，该 Binder 线程通过 ActivityThread 的 H 类型 Handler 将消息发送给主线程。ActivityThread 并不是线程类，只是它运行在主线程。

Handler 底层采用 Linux 的 pipe/epoll 机制，MessageQueue 没有消息的时候，便阻塞在 Looper.mQueue.next 方法中，此时主线程会释放 CPU 资源进入休眠，直到下个事件到达，当有新消息的时候，通过往 pipe 管道写数据来唤醒主线程工作。所以主线程大多数时候处于休眠状态，不会阻塞。

38、LruCache 和 DiskLruCache

【参考】https://blog.csdn.net/guolin_blog/article/details/28863651

39、OnLowMemory 和 OnTrimMemory 的比较

【参考】
1、OnLowMemory 被回调时，已经没有后台进程；而 onTrimMemory 被回调时，还有后台进程。

2、OnLowMemory 是在最后一个后台进程被杀时调用，一般情况是 low memory killer 杀进程后触发；而 OnTrimMemory 的触发更频繁，每次计算进程优先级时，只要满足条件，都会触发。

3、通过一键清理后，OnLowMemory 不会被触发，而 OnTrimMemory 会被触发一次。

OnTrimMemory 的参数是一个 int 数值，代表不同的内存状态：
TRIM_MEMORY_COMPLETE：内存不足，并且该进程在后台进程列表最后一个，马上就要被清理
TRIM_MEMORY_MODERATE：内存不足，并且该进程在后台进程列表的中部。
TRIM_MEMORY_BACKGROUND：内存不足，并且该进程是后台进程。
TRIM_MEMORY_UI_HIDDEN：内存不足，并且该进程的 UI 已经不可见了。

以上 4 个是 4.0 增加

TRIM_MEMORY_RUNNING_CRITICAL：内存不足(后台进程不足 3 个)，并且该进程优先级比较高，需要清理内存
TRIM_MEMORY_RUNNING_LOW：内存不足(后台进程不足 5 个)，并且该进程优先级比较高，需要清理内存
TRIM_MEMORY_RUNNING_MODERATE：内存不足(后台进程超过 5 个)，并且该进程优先级比较高，需要清理内存
以上 3 个是 4.1 增加

40、Android UI 卡顿监测框架 BlockCanary 原理分析

【参考】：https://www.jianshu.com/p/e58992439793

41、Java 中四种线程池的总结

【参考】：具体的 4 种常用的线程池实现如下：（返回值都是 ExecutorService）

1、Executors.newCacheThreadPool()：可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务。

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

2、Executors.newFixedThreadPool(int n)：创建一个定长线程池，以共享的无界队列方式来运行这些线程。

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

3、Executors.newScheduledThreadPool(int n)：创建一个定长线程池，支持定时及周期性任务执行

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

4、 Executors.newSingleThreadExecutor()：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO，优先级)执行。

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }
```

corePoolSize:核心池大小，意思是当超过这个范围的时候，就需要将新的线程放到等待队列中了即 workQueue；

maximumPoolSize:线程池最大线程数量，表明线程池能创建的最大线程数

keepAlivertime:当活跃线程数大于核心线程数，空闲的多余线程最大存活时间。

unit：存活时间的单位

workQueue:存放任务的队列---阻塞队列

handler:超出线程范围（maximumPoolSize）和队列容量的任务的处理程序

我们执行线程时都会调用到 ThreadPoolExecutor 的 execute()方法，现在我们来看看这个方法的源码（就是下面这段代码了，这里面有一些注释解析），我直接来解释一下吧：在这段代码中我们至少要看懂一个逻辑：当当前线程数小于核心池线程数时，只需要添加一个线程并且启动它，如果线程数数目大于核心线程池数目，我们将任务放到 workQueue 中，如果连 WorkQueue 满了，那么就要拒绝任务了

42、Service 和 IntentService 的区别?

【参考】：1、IntentService 是继承并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作，启动 IntentService 的方式和启动传统的 Service 一样，同时，当任务执行完后，IntentService 会自动停止，而不需要我们手动去控制或 stopSelf()。另外，可以启动 IntentService 多次，而每一个耗时操作会以工作队列的方式在 IntentService 的 onHandleIntent 回调方法中执行，并且，每次只会执行一个工作线程，执行完第一个再执行第二个，以此类推。

2、子类需继承 IntentService 并且实现里面的 onHandlerIntent 抽象方法来处理 intent 类型的任务请求。

3、子类需要重写默认的构造方法，且在构造方法中调用父类带参数的构造方法。

4、IntentService 类内部利用 HandlerThread+Handler 构建了一个带有消息循环处理机制的后台工作线程，客户端只需调用 Content#startService(Intent)将 Intent 任务请求放入后台工作队列中，且客户端无需关注服务是否结束，非常适合一次性的后台任务。比如浏览器下载文件，退出当前浏览器之后，下载任务依然存在后台，直到下载文件结束，服务自动销毁。只要当前 IntentService 服务没有被销毁，客户端就可以同时投放多个 Intent 异步任务请求，IntentService 服务端这边是顺序执行当前后台工作队列中的 Intent 请求的，也就是每一时刻只能执行一个 Intent 请求，直到该 Intent 处理结束才处理下一个 Intent。因为 IntentService 类内部利用 HandlerThread+Handler 构建的是一个单线程来处理异步任务。

43、onSaveInstanceState 和 onRestoreInstanceState 调用时机

【参考】03-09 12:14:32.529 2298-2298/com.example.myapplication I/MY_TEST: onPause
03-09 12:14:32.556 2298-2298/com.example.myapplication I/MY_TEST: onCreate2
03-09 12:14:32.557 2298-2298/com.example.myapplication I/MY_TEST: onStart2
03-09 12:14:32.557 2298-2298/com.example.myapplication I/MY_TEST: onResume2
03-09 12:14:32.981 2298-2298/com.example.myapplication I/MY_TEST: onSaveInstanceState 一个参数
03-09 12:14:32.981 2298-2298/com.example.myapplication I/MY_TEST: onStop

分割线----------

03-09 12:15:28.715 2298-2298/com.example.myapplication I/MY_TEST: onPause2
03-09 12:15:28.763 2298-2298/com.example.myapplication I/MY_TEST: onCreate
03-09 12:15:28.764 2298-2298/com.example.myapplication I/MY_TEST: onStart
03-09 12:15:28.764 2298-2298/com.example.myapplication I/MY_TEST: onRestoreInstanceState
03-09 12:15:28.767 2298-2298/com.example.myapplication I/MY_TEST: onActivityResult
03-09 12:15:28.767 2298-2298/com.example.myapplication I/MY_TEST: onResume
03-09 12:15:29.141 2298-2298/com.example.myapplication I/MY_TEST: onStop2
03-09 12:15:29.141 2298-2298/com.example.myapplication I/MY_TEST: onDestroy2

注意点：

1.跳转过程中，先执行 1 的 onpause，等等 onstop，等待 2 的 onresume 执行完之后，再执行 1 的 onstop、ondestory

2.onSaveInstanceState 每次隐藏 activity 都会在 onpause 之后执行（即使 activity 没有销毁也会执行）

3.onRestoreInstance 在 onstart 之后，onActivityResult 之前执行

44、LeakCanary 原理

【参考】 弱引用 WeakReference

被强引用的对象就算发生 OOM 也永远不会被垃圾回收机回收；被弱引用的对象，只要被

垃圾回收器发现就会立即被回收；被软引用的对象，具备内存敏感性，只有内存不足时才会被回收，常用来做内存敏感缓存器；虚引用则任意时刻都可能被回收，使用较少。

引用队列 ReferenceQueue

我们常用一个 WeakReference reference = new WeakReference(activity);，这里我们创建了一个 reference 来弱引用到某个 activity，当这个 activity 被垃圾回收器回收后，这个 reference 会被放入内部的 ReferenceQueue 中。也就是说，从队列 ReferenceQueue 取出来的所有 reference，它们指向的真实对象都已经成功被回收了。

1、利用 application.registerActivityLifecycleCallbacks(lifecycleCallbacks)
来监听整个生命周期内的 Activity onDestoryed 事件;

2、当某个 Activity 被 destory 后，将它传给 RefWatcher 去做观测，确保其后续会被正常回收;

3、RefWatcher 首先把 Activity 使用 KeyedWeakReference 引用起来，并使用一个 ReferenceQueue 来记录该 KeyedWeakReference 指向的对象是否已被回收;

4、AndroidWatchExecutor 会延迟 5 秒后，再开始检查这个弱引用内的 Activity 是否被正常回收。判断条件是：若 Activity 被正常回收，那么引用它的 KeyedWeakReference 会被自动放入 ReferenceQueue 中。

5、判断方式是：先看 Activity 对应的 KeyedWeakReference 是否已经放入 ReferenceQueue 中；如果没有，则手动 GC：gcTrigger.runGc();；然后再一次判断 ReferenceQueue 是否已经含有对应的 KeyedWeakReference。若还未被回收，则认为可能发生内存泄漏。

6、利用 HeapAnalyzer 对 dump 的内存情况进行分析并进一步确认，若确定发生泄漏，则利用 HeapAnalyzerService 发送通知。

7、弱引用与 ReferenceQueue 联合使用，如果弱引用关联的对象被回收，则会把这个弱引用加入到 ReferenceQueue 中；通过这个原理，可以看出 removeWeaklyReachableReferences() 执行后，会对应删除 KeyedWeakReference 的数据。如果这个引用继续存在，那么就说明没有被回收。

8、为了确保最大保险的判定是否被回收，一共执行了两次回收判定，包括一次手动 GC 后的回收判定。两次都没有被回收，很大程度上说明了这个对象的内存被泄漏了，但并不能 100% 保证；因此 LeakCanary 是存在极小程度的误差的。

https://blog.csdn.net/dawn4get/article/details/76473524

45、onWindowFocusChanged 执行时机

【参考】真正的 visible 时间点是 onWindowFocusChanged()函数被执行时，当前窗体得到或失去焦点的时候的时候调用。这是这个活动是否是用户可见的最好的指标。

46、HashSet 类是如何实现添加元素保证不重复的

【参考】

for 循环中，遍历 table 中的元素，
1，如果 hash 码值不相同，说明是一个新元素，存；

如果没有元素和传入对象（也就是 add 的元素）的 hash 值相等，那么就认为这个元素在 table 中不存在，将其添加进 table；

2（1），如果 hash 码值相同，且 equles 判断相等，说明元素已经存在，不存；

2（2），如果 hash 码值相同，且 equles 判断不相等，说明元素不存在，存；

如果有元素和传入对象的 hash 值相等，那么，继续进行 equles()判断，如果仍然相等，那么就认为传入元素已经存在，不再添加，结束，否则仍然添加；

可见 hashcode()和 equles()在此显得很关键了，下面就来解读一下 hashcode 和 equles：

首先要明确：只通过 hash 码值来判断两个对象时否相同合适吗？答案是不合适的，因为有可能两个不同的对象的 hash 码值相同；

什么是 hash 码值？

在 java 中存在一种 hash 表结构，它通过一个算法，计算出的结果就是 hash 码值；这个算法叫 hash 算法；

47、Java 并发容器

【参考】

ConcurrentHashMap

HashMap 不是线程安全的。
HashTable 容器使用 synchronized 来保证线程安全，在线程竞争激烈的情况下 HashTable 的效率非常低下。
ConcurrentHashMap 采用了 Segment 分段技术，容器里有多把锁，每把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率。
ConcurrentHashMap 结构：

CopyOnWriteArrayList
CopyOnWrite 容器即写时复制的容器。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。类似的有 CopyOnWriteArraySet。

48、Vector 与 ArrayList 与 CopyOnWriteArrayList 区别

【参考】1. Vector & ArrayList
1）Vector 的方法都是同步的(Synchronized),是线程安全的(thread-safe)，而 ArrayList 的方法不是，由于线程的同步必然要影响性能，因此,ArrayList 的性能比 Vector 好。

2)当 Vector 或 ArrayList 中的元素超过它的初始大小时,Vector 会将它的容量翻倍,而 ArrayList 只增加 50%的大小，这样,ArrayList 就有利于节约内存空间。

（1）同步性：
Vector 是线程安全的，也就是说是它的方法之间是线程同步的，而 ArrayList 是线程序不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用 ArrayList，因为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用 Vector，因为不需要我们自己再去考虑和编写线程安全的代码。

备注：对于 Vector&ArrayList、Hashtable&HashMap，要记住线程安全的问题，记住 Vector 与 Hashtable 是旧的，是 java 一诞生就提供了的，它们是线程安全的，ArrayList 与 HashMap 是 java2 时才提供的，它们是线程不安全的。所以，我们讲课时先讲老的。（2）数据增长：ArrayList 与 Vector 都有一个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加 ArrayList 与 Vector 的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要取得一定的平衡。Vector 默认增长为原来两倍，而 ArrayList 的增长策略在文档中没

有明确规定（从源代码看到的是增长为原来的 1.5 倍）。ArrayList 与 Vector 都可以设置初始的空间大小，Vector 还可以设置增长的空间大小，而 ArrayList 没有提供设置增长空间的方法。

2. Hashtable & HashMap
Hashtable 和 HashMap 它们的性能方面的比较类似 Vector 和 ArrayList，比如 Hashtable 的方法是同步的,而 HashMap 的不是。

3. ArrayList & LinkedList

ArrayList 的内部实现是基于内部数组 Object[],所以从概念上讲,它更象数组，但 LinkedList 的内部实现是基于一组连接的记录，所以，它更象一个链表结构，所以，它们在性能上有很大的差别:

从上面的分析可知,在 ArrayList 的前面或中间插入数据时,你必须将其后的所有数据相应的后移,这样必然要花费较多时间，所以,当你的操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用 ArrayList 会提供比较好的性能； 而访问链表中的某个元素时,就必须从链表的一端开始沿着连接方向一个一个元素地去查找,直到找到所需的元素为止，所以,当你的操作是在一列数据的前面或中间添加或删除数据，并且按照顺序访问其中的元素时，就应该使用 LinkedList 了。

4. Vector & ArrayList & CopyOnWriteArrayList

这三个集合类都继承 List 接口

1、ArrayList 是线程不安全的；
2、Vector 是比较古老的线程安全的，但性能不行；
3、CopyOnWriteArrayList 在兼顾了线程安全的同时，又提高了并发性，性能比 Vector 有不少提高

CopyOnWriteArrayList 几个要点
实现了 List 接口
内部持有一个 ReentrantLock lock = new ReentrantLock();
底层是用 volatile transient 声明的数组 array
读写分离，写时复制出一个新的数组，完成插入、修改或者移除操作后将新数组赋值给 array

ArrayList 几个重点
底层是数组，初始大小为 10
插入时会判断数组容量是否足够，不够的话会进行扩容
所谓扩容就是新建一个新的数组，然后将老的数据里面的元素复制到新的数组里面
移除元素的时候也涉及到数组中元素的移动，删除指定 index 位置的元素，然后将 index+1 至数组最后一个元素往前移动一个格

https://blog.csdn.net/linsongbin1/article/details/54581787

49、Android 系统为什么会设计 ContentProvider?

【参考】
在开发中，假如，A、B 进程有部分信息需要同步，这个时候怎么处理呢？设想这么一个场景，有个业务复杂的 Activity 非常占用内存，并引发 OOM，所以，想要把这个 Activity 放到单独进程，以保证 OOM 时主进程不崩溃。但是，两个整个 APP 有些信息需要保持同步，比如登陆信息等，无论哪个进程登陆或者修改了相应信息，都要同步到另一个进程中去，这个时候怎么做呢？

第一种：一个进程里面的时候，经常采用 SharePreference 来做，但是 SharePreference 不支持多进程，它基于单个文件的，默认是没有考虑同步互斥，而且，APP 对 SP 对象做了缓存，不好互斥同步，虽然可以通过 FileLock 来实现互斥，但同步仍然是一个问题。
第二种：基于 Binder 通信实现 Service 完成跨进程数据的共享，能够保证单进程访问数据，不会有互斥问题，可是同步的事情仍然需要开发者手动处理。
第三种：基于 Android 提供的 ContentProvider 来实现，ContentProvider 同样基于 Binder，不存在进程间互斥问题，对于同步，也做了很好的封装，不需要开发者额外实现。

ContentProvider 只是 Android 为了跨进程共享数据提供的一种机制，
本身基于 Binder 实现，
在操作数据上只是一种抽象，具体要自己实现
ContentProvider 只能保证进程间的互斥，无法保证进程内，需要自己实现

ContentResolver 接口的 notifyChange 函数来通知那些注册了监控特定 URI 的 ContentObserver 对象，使得它们可以相应地执行一些处理。ContentObserver 可以通过 registerContentObserver 进行注册。

既然是对外提供数据共享，那么如何限制对方的使用呢？

android:exported 属性非常重要。这个属性用于指示该服务是否能够被其他应用程序组件调用或跟它交互。如果设置为 true，则能够被调用或交互，否则不能。设置为 false 时，只有同一个应用程序的组件或带有相同用户 ID 的应用程序才能启动或绑定该服务

https://blog.csdn.net/happylishang/article/details/78550156

50、Service 和 Activity 通信

【参考】

Activity 调用 bindService (Intent service, ServiceConnection conn, int flags)方法，得到 Service 对象的一个引用，这样 Activity 可以直接调用到 Service 中的方法，如果要主动通知 Activity，我们可以利用回调方法

Service 向 Activity 发送消息，可以使用广播，当然 Activity 要注册相应的接收器。比如 Service 要向多个 Activity 发送同样的消息的话，用这种方法就更好

51、AsyncTask 内部维护了一个线程池，是串行还是并行，怎么维护的?

【参考】：串行 https://www.cnblogs.com/absfree/p/5357678.htm

52、死锁的产生条件？如何避免死锁

【参考】

死锁的产生是必须要满足一些特定条件的：
1.互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放
2.请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。
3.不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用
4.循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

基本思想：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，否则分配。这是一种动态策略。典型的避免死锁的算法试银行家算法。

https://blog.csdn.net/ZWE7616175/article/details/79881236

53、Android 消息机制实现

【参考】https://www.jianshu.com/p/f10cff5b4c25

54、Android 崩溃捕获机制

【参考】https://www.jianshu.com/p/fb28a5322d8a

55、Android 在程序崩溃或者捕获异常之后重新启动 app

【参考】

在 Android 应用开发中，偶尔会因为测试的不充分导致一些异常没有被捕获，这时应用会出现异常并强制关闭，这样会导致很不好的用户体验，为了解决这个问题，我们需要捕获相关的异常并做处理。

首先捕获程序崩溃的异常就必须了解一下 Java 中 UncaughtExceptionHandler 这个接口，这个接口在 Android 开发上面也是可以使用的，在 API 文档中，我们可以了解到：通过实现此接口，能够处理线程被一个无法捕获的异常所终止的情况。如上所述的情况，handler 将会报告线程终止和不明原因异常这个情况，如果没有自定义 handler， 线程管理组就被默认为报告异常的 handler。 ThreadGroup 这个类就是实现了 UncaughtExceptionHandler 这个接口， 如果想捕获异常我们可以实现这个接口或者继承 ThreadGroup， 并重载 uncaughtException 方法。

56、RecyclerView 体验优化及入坑总结

【参考】https://www.jianshu.com/p/90c31e97cc55

recyclerView 列表类控件卡顿优化

https://www.cnblogs.com/ldq2016/p/9039979.html

57、Activity 与 Service 通信的四种方式

【参考】

1、Binder
2、Intent

3、接口 Interface
4、Broadcast 广播接收

http://itindex.net/detail/45126-android-service-activity

58、Activity 之间的几种通信方式

【参考】

1、Intent
2、借助类的静态变量
3、借助全局变量/Application
4、借助外部工具
– 借助 SharedPreference
– 使用 Android 数据库 SQLite
– 赤裸裸的使用 File
– Android 剪切板
5、借助 Service

https://blog.csdn.net/cyanchen666/article/details/81982562

59、LRUCache 使用及原理

【参考】https://www.cnblogs.com/huhx/p/useLruCache.html

60、OOM 是否可以 try catch ？

【参考】https://blog.csdn.net/sinat_29912455/article/details/51125748

一般不适合这么处理

Java 中管理内存除了显式地 catch OOM 之外还有更多有效的方法：比如 SoftReference，WeakReference，硬盘缓存等。

在 JVM 用光内存之前，会多次触发 GC，这些 GC 会降低程序运行的效率。
如果 OOM 的原因不是 try 语句中的对象（比如内存泄漏），那么在 catch 语句中会继续抛出 OOM

https://www.jianshu.com/p/f62ab73a7de2

61、Activity 启动模式有几种?

【参考】https://www.cnblogs.com/lwbqqyumidi/p/3771542.html

62、ListView 与 RecyclerView 简单对比

【参考】https://blog.csdn.net/shu_lance/article/details/79566189

63、类加载机制

【参考】https://blog.csdn.net/noaman_wgs/article/details/74489549
https://www.cnblogs.com/aspirant/p/7200523.html

64、抽象类和接口区别?

【参考】http://www.importnew.com/12399.html

65、Https 通信过程

【参考】https://blog.csdn.net/mzh1992/article/details/53885098

66、AsyncTask 的优缺点

【参考】
缺点:
1、串行执行
2、内存泄露
3、结果丢失
4、必须要执行在主线程中

https://www.cnblogs.com/yanyojun/archive/2017/02/20/6414919.html
https://blog.csdn.net/boyupeng/article/details/49001215

67、如何计算一个 View 的层级

【参考】比如计算一个 LinearLayout 的嵌套层级：

```
int i = 0;
private void getParents(ViewParent view){

    if (view.getParent() == null) {
        Log.v("tag", "最终==="+i);
        return;
    }


    i++;
    ViewParent parent = view.getParent();
    Log.v("tag", "i===="+i);
    Log.v("tag", "parent===="+parent.toString());

    getParents(parent);
}
```

调用：

```
LinearLayout llVip;
getParents(llVip);
```

因为

public abstract class ViewGroup extends View implements ViewParent

ViewGroup 是 ViewParent 的实现类，所以可以直接转，

LinearLayout 是 ViewGroup 的子类

68、断点下载原理

【参考】https://blog.csdn.net/seu_calvin/article/details/53749776

69、Handler 主线程向子线程发消息

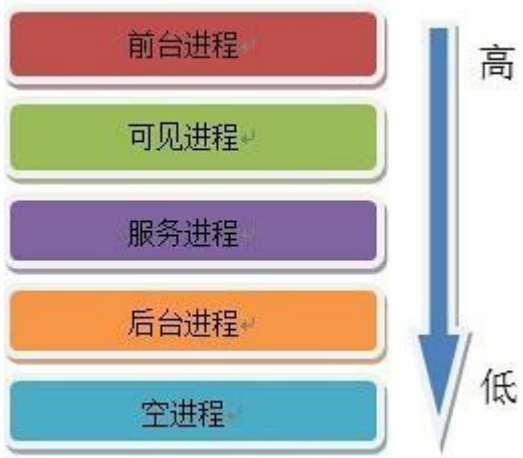
【参考】https://www.jianshu.com/p/34fb9f9815f4

70、Andorid 进程分类

【参考】

Android 中，同一个应用的所有组件在默认情况下都运行在同一个进程中，但也可以通过修改 manifest 文件中的 android：process 属性来指定该组件要运行中那个进程，也可以让不同应用的组件们运行在同一个进程中，当然这些应用要共享一个用户 ID 并并且有相同的数字证书。

Android 可能在某个时刻决定关闭一个进程，当决定要关闭那些进程的时候，系统会衡量每个进程与用户的紧密程度，这时候就跟 Android 中进程的级别有关了。像一个具有可见的 activity 的进程要比那些 activity 都是不可见的进程拥有更高的等级，更不容易被系统杀死。那么 Android 的进程等级有那些呢？首先，我们可以看一张图：



上面的图片就是 Android 系统中不同类型的进程和他们的优先级了。下面是每个进程的介绍。

1、前台进程
用户当前正在做的事情需要这个进程。如果满足下面的条件之一，一个进程就被认为是前台进程：
这个进程拥有一个正在与用户交互的 Activity(这个 Activity 的 onResume()方法被调用)。
这个进程拥有一个绑定到正在与用户交互的 activity 上的 Service。
这个进程拥有一个前台运行的 Service（service 调用了方法 startForeground()）.
这个进程拥有一个正在执行其任何一个生命周期回调方法（onCreate(),onStart(),或 onDestroy()）的 Service。
这个进程拥有正在执行其 onReceive()方法的 BroadcastReceiver。

通常，在任何时间点，只有很少的前台进程存在。它们只有在达到无法调合的矛盾时才会被杀－－如内存太小而不能继续运行时。通常，到了这时，设备就达到了一个内存分页调度状态，所以需要杀一些前台进程来保证用户界面的反应

2、可见进程
一个进程不拥有运行于前台的组件，但是依然能影响用户所见。满足下列条件时，进程即为可见：
这个进程拥有一个不在前台但仍可见的 Activity(它的 onPause()方法被调用)。当一个前台 activity 启动一个对话框时，就出了这种情况。

3、服务进程
一个可见进程被认为是极其重要的。并且，除非只有杀掉它才可以保证所有前台进程的运行，否则是不能动它的。
这个进程拥有一个绑定到可见 activity 的 Service。
一个进程不在上述两种之内，但它运行着一个被 startService()所启动的 service。
尽管一个服务进程不直接影响用户所见，但是它们通常做一些用户关心的事情（比如播放音乐或下载数据），所以系统不到前台进程和可见进程活不下去时不会杀它。

4、后台进程
一个进程拥有一个当前不可见的 activity(activity 的 onStop()方法被调用)。
这样的进程们不会直接影响到用户体验，所以系统可以在任意时刻杀了它们从而为前台、可见、以及服务进程们提供存储空间。通常有很多后台进程在运行。它们被保存在一个 LRU(最近最少使用)列表中来确保拥有最近刚被看到的 activity 的进程最后被杀。如果一个 activity 正确的实现了它的生命周期方法，并保存了它的当前状态，那么杀死它的进程将不会对用户的可视化体验造成影响。因为当用户返回到这个 activity 时，这个 activity 会恢复它所有的可见状态。

5、空进程
一个进程不拥有入何 active 组件。
保留这类进程的唯一理由是高速缓存，这样可以提高下一次一个组件要运行它时的启动速度。系统经常为了平衡在进程高速缓存和底层的内核高速缓存之间的整体系统资源而杀死它们。

https://www.cnblogs.com/jiuguimianju/p/4162102.html

71、AlertDialog，Toast 对 Activity 生命周期的影响

【参考】：无论 Dialog 弹出覆盖页面，对 Activity 生命周期没有影响，只有再启动另外一个 Activity 的时候才会进入 onPause 状态，而不是想象中的被覆盖或者不可见，同时通过 AlertDialog 源码或者 Toast 源码我们都可以发现它们实现的原理都是 windowmanager.addView();来添加的， 它们都是一个个 view ,因此不会对 activity 的生命周期有任何影响。

https://blog.csdn.net/cloud_castle/article/details/56011562

72、JVM 内存模型，内存区域

【参考】Java 内存划分：https://www.cnblogs.com/dolphin0520/p/3613043.html

73、Java 观察者模式

【参考】https://www.cnblogs.com/luohanguo/p/7825656.html

74、ContentProvider 与 DB 关系，provider 可以调用 db 么?

【参考】
1、ContentProvider 提供了对底层数据存储方式的抽象。比如上图中，底层使用了 SQLite 数据库，在用了 ContentProvider 封装后，即使你把数据库换成 MongoDB，也不会对上层数据使用层代码产生影响。

2、Android 框架中的一些类需要 ContentProvider 类型数据。如果你想让你的数据可以使用在如 SyncAdapter，Loader，CursorAdapter 等类上，那么你就需要为你的数据做一层 ContentProvider 封装。

3、第三个原因也是最主要的原因，是 ContentProvider 为应用间的数据交互提供了一个安全的环境。它准许你把自己的应用数据根据需求开放给其他应用进行增、删、改、查，而不用担心直接开放数据库权限而带来的安全问题。

https://www.jianshu.com/p/f5ec75a9cfea

75、onCreate 中的 Bundle 有什么用?

【参考】 Activity 中有一个名称叫 onCreate 的方法。该方法是在 Activity 创建时被系统调用，是一个 Activity 生命周期的开始。可是有一点容易被忽视，就是 onCreate 方法的参数 saveInsanceState。一般的程序开发中，很少用到这个参数。

onCreate 方法的完整定义如下:
public void onCreate(Bundle saveInsanceState){

```
    super.onCreate(saveInsanceState);
}
```

Bundle 类型的数据与 Map 类型的数据相似，都是以 key-value 的形式存储数据的。

从字面上看 saveInsanceState，是保存实例状态的。实际上，saveInsanceState 也就是保存 Activity 的状态的。那么，saveInsanceState 中的状态数据是从何处而来的呢？下面我们介绍 Activity 的另一个方法 saveInsanceState。

onsaveInsanceState 方法是用来保存 Activity 的状态的。当一个 Activity 在生命周期结束前，会调用该方法保存状态。

如下所示：

```
public void onSaveInsanceState(Bundle saveInsanceState){
    super.onSaveInsanceState(saveInsanceState);
}
```

在实际应用中，当一个 Activity 结束前，如果需要保存状态，就在 onsaveInsanceState 中，将状态数据以 key-value 的形式放入到 saveInsanceState 中。这样，当一个 Activity 被创建时，就能从 onCreate 的参数 saveInsanceState 中获得状态数据。

状态这个参数在实现应用中有很大的用途，比如：一个游戏在退出前，保存一下当前游戏运行的状态，当下次开启时能接着上次的继续玩下去。再比如：电子书程序，当一本小说被阅读到第 199 页后退出了（不管是内存不足还是用户自动关闭程序），当下次打开时，读者可能已忘记了上次已阅读到第几页了，但是，读者想接着上次的读下去。如果采用 saveInstallState 参数，就很容易解决上述问题。

https://blog.csdn.net/liubin8095/article/details/9328563

76、Parcel 类详解，Parcel 和 Parcelable 的区别?

【参考】：Parcel 在英文中有两个意思，其一是名词，为包裹，小包的意思； 其二为动词，意为打包，扎包。邮寄快递中的包裹也用的是这个词。Android 采用这个词来表示封装消息数据。这个是通过 IBinder 通信的消息的载 体。需要明确的是 Parcel 用来存放数据的是内存（RAM），而不是永久性介质（Nand 等）。

Parcelable,定义了将数据写入 Parcel，和从 Parcel 中读出的接口。一个实体（用类来表示），如果需要封装到消息中去，就必须实现这一接口，实现了这一接口，该实体就成为“可打包的”了。

Parcel 翻译过来就是打包的意思，其实就是包装了我们需要传输的数据，然后在 Binder 中传输，用于跨进程传输数据。
Parcel 提供了一套机制，可以将序列化之后的数据写入一个共享内存中，其他进程通过 Parcel 可以从这块共享内存读出字节流，并反序列化成对象
Parcel 是一个存放读取数据的容器，系统中的 binder 进程间通信(IPC)就使用了 Parcel 类来进行客户端与服务端数据交互，而且 AIDL 的数据也是通过 Parcel 来交互的。在 Java 层和 C++层都实现了 Parcel，由于它在 C/C++中，直接使用了内存来读取数据，因此，它更有效率。
https://www.jianshu.com/p/fc7f7d1b0551

77、什么要用多进程？有哪些方式？怎么使用多进程？

【参考】那么多进程应该能为我们带来什么呢？
我们都知道，android 平台对应用都有内存限制，其实这个理解有点问题，应该是说 android 平台对每个进程有内存限制，比如某机型对对进程限制是 24m，如果应用有两个进程，则该应该的总内存限制是 2*24m。使用多进程就可以使得我们一个 apk 所使用的内存限制加大几倍。
所以可以借此图片平台对应用的内存限制，比如一些要对图片、视频、大文件进程处理的好内存的应用可以考虑用多进程来解决应用操作不流畅问题。

开启多进程模式:
在 Android 中使用多进程只有一种方法,那就是在 AndroidManifest 中给四大组件(Activity,Service,Receiver,ContentProvider)指定 android:process 属性.除此之外没有其他的办法,也就是说我们无法给一个线程活一个实体类指定其运行时所在的进程.其实还有另一种非常规的多进程方法,那就是通过 JNI 在 native 层去 fork 一个新的进程,但这种方法属于特殊情况,并不是常用的创建多进程的方式,所以我们也暂不考虑这种情况

进程名以":"开头的进程属于当前应用的私有进程,其他应用的组件不可以和它跑在同一个进程中,而进程名不以":"开头的进程属于全局进程,其他应用通过 ShareUID 方式可以和它跑在同一个进程中.

用多进程的好处与坏处

好处:

1）分担主进程的内存压力。
当应用越做越大，内存越来越多，将一些独立的组件放到不同的进程，它就不占用主进程的内存空间了。当然还有其他好处，有心人会发现
2）使应用常驻后台，防止主进程被杀守护进程，守护进程和主进程之间相互监视，有一方被杀就重新启动它。Android 后台进程里有很多应用是多个进程的，因为它们要常驻后台，

特别是即时通讯或者社交应用，不过现在多进程已经被用烂了。

典型用法是在启动一个不可见的轻量级私有进程，在后台收发消息，或者做一些耗时的事情，或者开机启动这个进程，然后做监听等。

坏处：消耗用户的电量。

多占用了系统的空间，若所有应用都这样占用，系统内存很容易占满而导致卡顿。
应用程序架构会变得复杂，因为要处理多进程之间的通信。这里又是另外一个问题了。

多进程的缺陷

进程间的内存空间是不可见的。开启多进程后，会引发以下问题：

1）Application 的多次重建。
2）静态成员的失效。
3）文件共享问题。
4）断点调试问题。

https://blog.csdn.net/spencer_hale/article/details/54968092
https://blog.csdn.net/u010019468/article/details/72782098

78、如何获取一个图片的宽高？

【参考】 android 在不加载图片的前提下获得图片的宽高

```
public static int[] getImageWidthHeight(String path){
    BitmapFactory.Options options = new BitmapFactory.Options();

    /**
     * 最关键在此，把 options.inJustDecodeBounds = true;
     * 这里再 decodeFile()，返回的 bitmap 为空，但此时调用 options.outHeight 时，已经包含了图片的高了
     */
    options.inJustDecodeBounds = true;
    Bitmap bitmap = BitmapFactory.decodeFile(path, options); // 此时返回的 bitmap 为 null
    /**
     *options.outHeight 为原始图片的高
     */
    return new int[]{options.outWidth,options.outHeight};
}
```

通过 BitmapFactory 从不同位置获取 Bitmap

1.资源文件(drawable/mipmap/raw)

```
BitmapFactory.decodeResource(getResources(),
R.mipmap.slim_lose_weight_plan_copenhagen,options);
```

2.资源文件(assets)

```
InputStream is = getActivity().getAssets().open("bitmap.png");
BitmapFactory.decodeStream(is);
```

3.内存卡文件

```
bitmap = BitmapFactory.decodeFile("/sdcard/bitmap.png");
```

4.网络文件

```
bitmap = BitmapFactory.decodeStream(is);
```

可根据 BitmapFactory 获取图片时传入 option，通过上述方法获取图片的宽高

79、viewstub 可以多次 inflate 么？多次 inflate 会怎样?

【参考】在使用 viewstub 的时候要注意一点，viewstub 只能 inflate 一次，而且 setVisibility 也会间接的调用到 inflate，重复 inflate 会抛出异常：
java.lang.IllegalStateException:ViewStub must have a non-null ViewGroup viewParent

解决方法为设置一个 Boolean 类型的变量，标记 viewstub 是否已经 inflate，如果 viewstub 还未 inflate 则执行初始化操作，反之则不进行操作。其中要使用 ViewStub 中的 OnInflateListener()监听事件来判断是否已经填充,从而保证 viewstub 不重复的 inflate。
解决方法：
1.定义 boolean 变量和 ViewStub

```
boolean isInflate = false;
ViewStub mViewStub;
```

2.初始化 ViewStub，并为 ViewStub 添加 OnInflateListener()监听事件

```
mViewStub = (ViewStub)findViewById(R.id.viewstub_match_single);

    mViewStub.setOnInflateListener(new OnInflateListener() {

        @Override
```

```
            public void onInflate(ViewStub stub, View inflated) {

                isInflate = true;
            }
        });
```

3.填充 ViewStub

```
private void initViewStub(){//填充 ViewStub 的方法
            if(!isInflate){//如果没有填充则执行 inflate 操作
                View view = stubMatchSingle.inflate();
                //初始化 ViewStub 的 layout 里面的控件
                TextView mTv = (TextView) view.findViewById(R.id.txt_url);
                mTv.setOnClickListener(this);
            }
        }
```

80、VSync 机制介绍下?

【参考】https://blog.csdn.net/wangxueming/article/details/64457436

81、Android UI 线程和非 UI 线程的交互方式

【参考】：用于实现后台线程与 UI 线程的交互。

1、Handler
2、Activity.runOnUIThread(Runnable)
3、View.Post(Runnable)
4、View.PostDelayed(Runnabe,long)
5、AsyncTask

1. Handler

handler 是 android 中专门用来在线程之间传递信息类的工具。

2. Activity.runOnUIThread(Runnable)

这个方法相当简单，我们要做的只是以下几步

<1> 编写后台线程，这回你可以直接调用 UI 控件

<2> 创建后台线程的实例

<3> 调用 UI 线程对应的 Activity 的 runOnUIThread 方法，将后台线程实例作为参数传入其中。

注意：无需调用后台线程的 start 方法

3. View.Post(Runnable)
该方法和方法二基本相同，只是在后台线程中能操控的 UI 控件被限制了，只能是指定的 UI 控件 View。方法如下
<1> 编写后台线程，这回你可以直接调用 UI 控件，但是该 UI 控件只能是 View
<2> 创建后台线程的实例
<3> 调用 UI 控件 View 的 post 方法，将后台线程实例作为参数传入其中

4. View.PostDelayed(Runnabe, long)
该方法是方法三的补充，long 参数用于制定多少时间后运行后台进程

5. AsyncTask
AsyncTask 是一个专门用来处理后台进程与 UI 线程的工具。通过 AsyncTask，我们可以非常方便的进行后台线程和 UI 线程之间的交流。
AsyncTask 拥有 3 个重要参数
<1> Params，Params 是后台线程所需的参数。在后台线程进行作业的时候，他需要外界为其提供必要的参数，就好像是一个用于下载图片的后台进程，他需要的参数就是图片的下载地址
<2> Progress，Progress 是后台线程处理作业的进度。依旧上面的例子说，就是下载图片这个任务完成了多少，是 20%还是 60%。这个数字是由 Progress 提供
<3> Result，Result 是后台线程运行的结果，也就是需要提交给 UI 线程的信息。按照上面的例子来说，就是下载完成的图片
AsyncTask 还拥有 4 个重要的回调方法
<1> onPreExecute，onPreExecute 运行在 UI 线程，主要目的是为后台线程的运行做准备。当他运行完成后，他会调用 doInBackground 方法
<2> doInBackground，doInBackground 运行在后台线程，他用来负责运行任务。他拥有参数 Params，并且返回 Result。在后台线程的运行当中，为了能够更新作业完成的进度，需要在 doInbackground 方法中调用 PublishProgress 方法。该方法拥有参数 Progress。通过该方法可以更新 Progress 的数据。然后当调用完 PublishProgress 方法，他会调用 onProgressUpdate 方法用于更新进度
<3> onProgressUpdate，onProgressUpdate 运行在 UI 线程，主要目的是用来更新 UI 线程中显示进度的 UI 控件。他拥有 Progress 参数。在 doInBackground 中调用 PublishProgress 之后，就会自动调 onProgressUpdate 方法
<4> onPostExecute，onPostExecute 运行在 UI 线程，当 doInBackground 方法运行完后，他会调用 onPostExecute 方法，并传入 Result。在 onPostExecute 方法中，就可以将 Result 更新到 UI 控件上
注意：AsyncTask 实例只能执行一次，否则就出错哦

82、parcel 如何读写对象，细节？

【参考】

@Override

```
public void writeToParcel(Parcel dest, int flags) {
        // 序列化过程：必须按成员变量声明的顺序进行封装
        dest.writeInt(id);
        dest.writeString(name);
}

// 反序列过程：必须实现 Parcelable.Creator 接口，并且对象名必须为 CREATOR
// 读取 Parcel 里面数据时必须按照成员变量声明的顺序，Parcel 数据来源上面 writeToParcel 方法，读出来的数据供逻辑层使用
public static final Parcelable.Creator<Student> CREATOR = new Creator<Student>() {

        @Override
        public Student createFromParcel(Parcel source) {
                return new Student(source.readInt(), source.readString());
        }

        @Override
        public Student[] newArray(int size) {
                return new Student[size];
        }
};
```

83、dex 拆分

【参考】https://blog.csdn.net/jiangwei0910410003/article/details/50799573

84、Android classLoader 和 Java ClassLoader

【参考】https://blog.csdn.net/itachi85/article/details/78088701

85、HTTP 与 TCP 的区别和联系

【参考】
1、TCP 连接

手机能够使用联网功能是因为手机底层实现了 TCP/IP 协议，可以使手机终端通过无线网络建立 TCP 连接。TCP 协议可以对上层网络提供接口，使上层网络数据的传输建立在"无差别"

的网络之上。

建立起一个 TCP 连接需要经过"三次握手":
第一次握手：客户端发送 syn 包(syn=j)到服务器，并进入 SYN_SEND 状态，等待服务器确认；

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

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

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP 连接一旦建立，在通信双方中的任何一方主动关闭连 接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开 TCP 连接的请求，断开过程需要经过"四次握手"（过程就不细写 了，就是服务器和客户端交互，最终确定断开）

2、HTTP 连接

HTTP 协议即超文本传送协议(Hypertext Transfer Protocol)，是 Web 联网的基础，也是手机联网常用的协议之一，HTTP 协议是建立在 TCP 协议之上的一种应用。

HTTP 连接最显著的特点是客户端发送的每次请求都需要服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为"一次连接"。

1）在 HTTP 1.0 中，客户端的每次请求都要求建立一次单独的连接，在处理完本次请求后，就自动释放连接。

2）在 HTTP 1.1 中则可以在一次连接中处理多个请求，并且多个请求可以重叠进行，不需要等待一个请求结束后再发送下一个请求。

由于 HTTP 在每次请求结束后都会主动释放连接，因此 HTTP 连接是一种"短连接"，要保持客户端程序的在线状态，需要不断地向服务器发起连接请求。通常的 做法是即时不需要获得任何数据，客户端也保持每隔一段固定的时间向服务器发送一次"保持连接"的请求，服务器在收到该请求后对客户端进行回复，表明知道客 户端"在线"。若服务器长时间无法收到客户端的请求，则认为客户端"下线"，若客户端长时间无法收到服务器的回复，则认为网络已经断开。

TPC/IP 协议是传输层协议，主要解决数据如何在网络中传输，而 HTTP 是应用层协议，主要解决如何包装数据。TCP 协议对应于传输层，而 HTTP 协议对应于应用层，从本质上来说，二者没有可比性。Http 协议是建立在 TCP 协议基础之上的，当浏览器需要从服务器获取网页数据的时候，会发出一次 Http 请求。Http 会通过 TCP 建立起一个到服务器的连接通道。简单地说，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接

不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接 Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如 Apache）中设定这个时间。Http 是无状态的短连接，而 TCP 是有状态的长连接
https://www.cnblogs.com/baizhanshi/p/8482612.html

86、Java 四种引用

【参考】

强引用 StrongReference
如果一个对象具有强引用，那么垃圾回收器绝对不会回收它，当内存不足时宁愿抛出 OOM 错误，使得程序异常停止。
Object object = new Object(); 即是一个强引用。

软引用 SoftReference
如果一个对象只具有软引用，那么垃圾回收器在内存充足的时候不会回收它，而在内存不足时会回收这些对象。软引用对象被回收后，Java 虚拟机会把这个软引用加入到与之关联的引用队列中。

弱引用 WeakReference
如果一个对象只具有弱引用，那么垃圾回收器在扫描到该对象时，无论内存充足与否，都会回收该对象的内存。与软引用相同，弱引用对象被回收后，Java 虚拟机会把这个弱引用加入到与之关联的引用队列中。

虚引用 PhantomReference
虚引用并不决定对象生命周期，如果一个对象只具有虚引用，那么它和没有任何引用一样，任何时候都可能被回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。与软引用和弱引用不同的是，虚引用必须关联一个引用队列。
当垃圾回收器准备回收一个对象之前，如果发现它还具有虚引用，就会在对象回收前把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否加入了虚引用，来了解被引用的对象是否将要被回收，那么就可以在其被回收之前采取必要的行动。

弱引用解决内存泄露问题
虽然 Java 有垃圾回收机制，但是只要是自己管理的内存，就应该警惕内存泄露的问题，例如对象池、缓存中的过期对象都有可能引发内存泄露的问题。用 WeakHashMap 来作为缓存的容器可以有效解决这一问题。WeakHashMap 和 HashMap 几乎一样，唯一的区别就是它的键使用弱引用。当 WeakHashMap 的键标记为过期时，这个键对应的条目就会自动被移除。这就避免了内存泄漏问题。
另外在 Android 中常常用到 Handler 消息处理机制。当使用内部类（包括匿名内部类）来创建 Handler 时，Handler 对象会隐式的持有一个其外部类对象（通常是一个 Activity）的引用。而 Handler 通常会开启一个耗时的工作线程（例如下载获取数据），工作线程完成任务后，通过消息机制通知 Handler 在主线程做相应的处理。如果在工作线程执行的过程中关闭了 Activity，Activity 应该在 GC 检查时被回收掉。但因为 Activity 被 Handler 持有引

用，Handler 又被 Message 持有引用，Message 被 MessageQueue 持有引用，MessageQueue 被 Looper 持有引用，而 Looper 是线程本地变量，线程不销毁，它就不会被销毁。这样一个引用链，导致 Activity 无法被回收，造成内存泄漏。

1. 通过程序逻辑控制防止内存泄漏
比如在 Activity 关闭时停掉工作线程，移除消息队列中的消息对象，这样切断了 Activity 的引用，就可以正常的被回收了。
2. 将 Handler 声明成静态内部类
静态类不持有外部类的对象，所以 Activity 可以被正常的回收。但这个时候 Handler 无法操作 Activity 中的对象了，所以这个时候需要增加一个对 Activity 弱引用。

https://www.cnblogs.com/alias-blog/p/5793108.html
https://www.cnblogs.com/fengbs/p/7019687.html

87、Java 中可重入锁 ReentrantLock

【参考】https://www.cnblogs.com/-new/p/7256297.html

88、多线程打印 ABCABCABC 顺序输出

【参考】https://blog.csdn.net/whu_zcj/article/details/51518051

89、进程保活

【参考】https://www.cnblogs.com/dongweiq/p/5404331.html

90、EventBus 原理

【参考】https://www.jianshu.com/p/d9516884dbd4

91、OKhttp 缓存机制

【参考】https://blog.csdn.net/weixin_40255793/article/details/81018358

92、ConcurrentHashMap 原理分析

【参考】https://www.cnblogs.com/ITtangtang/p/3948786.html

93、Android 高效加载大图、多图解决方案，有效避免程序 OOM

【参考】 https://blog.csdn.net/guolin_blog/article/details/9316683

94、如何在多线程中使用 JNI?

【参考】https://blog.csdn.net/booirror/article/details/37778283

95、Android 性能优化之内存检测、卡顿优化、耗电优化、APK 瘦身工具

【参考】https://blog.csdn.net/csdn_aiyang/article/details/74989318

96、冷启动的一些优化方案

【参考】: https://www.jianshu.com/p/1c9a18e49482

97、实现一个 CAS(乐观锁)的方法

【参考】https://www.cnblogs.com/qjjazry/p/6581568.html

98、Android APP 性能优化的一些思考

【参考】 https://www.cnblogs.com/cr330326/p/8011523.html

99、jni 开发需要注意的问题

【参考】https://blog.csdn.net/tianxuhong/article/details/53338044

100、onLowMemory 执行流程

【参考】 https://blog.csdn.net/qq_23547831/article/details/51465071

101、垃圾回收 什么时候触发 GC?

【参考】
https://blog.csdn.net/wangming520liwei/article/details/79750924

102、Android 内存优化——常见内存泄露及优化方案

【参考】
https://www.jianshu.com/p/ab4a7e353076
https://blog.csdn.net/qq_23191031/article/details/63685756

103、Android 内存优化总结&实践

【参考】 https://mp.weixin.qq.com/s/2MsEAR9pQfMr1Sfs7cPdWQ

104、Glide 源码学习,了解 Glide 图片加载原理

【参考】https://www.jianshu.com/p/9d8aeaa5a329

105、布局篇之减少你的界面层级

【参考】 https://blog.csdn.net/u010015108/article/details/52459726

106、过度绘制分析及解决方案

【参考】 https://www.jianshu.com/p/95566b2711b6

107、一张图片加载到手机内存中真正的大小是怎么计算的

参考】 https://www.cnblogs.com/dasusu/p/9789389.html

108、算法与数据结构专项

1、百度，腾讯，阿里对数据结构和算法要求掌握适中就行

2、快手、头条、拼多多刷题就一般能过。

一般都是大厂需要手写代码，比如头条，快手，腾讯，阿里。
1、比如常见的单链表，反转，插入，删除，双链表插入，删除
2、常见的排序，堆排序，归并排序、快排等。
3、二叉树的前序遍历，中序遍历，后序遍历等
4、最大 K 问题。经典的广度、深度优先搜索算法
5、单链表是否有环等经典问题。 【备注：以上内容必掌握~】

leetcode 刷题技巧：
可以先刷面试高频类：https://leetcode.com/problemset/top-interview-questions/

常考题：https://leetcode-cn.com/explore/interview/card/bytedance/
腾讯常考题：https://leetcode-cn.com/problemset/50/
github：https://github.com/zhedahht/ChineseCodingInterviewAppendix

也可以针对哪个公司常考题刷，也可以分类别刷，比如动态规划不好就多刷几道。 或者暴力刷（推荐有很长刷题时间）。