Android 开发工程师面试指南

deepwaterooo

July 11, 2017

Contents

Chapter 1

国内一线互联网公司内部面试题库

1.1 java

1.1.1 接口的意义-百度

- 规范、扩展、回调

1.1.2 抽象类的意义-乐视

- 为其子类提供一个公共的类型封装子类中得重复内容定义抽象方法，子类虽然有不同的实现但是定义是一致的

1.1.3 内部类的作用-乐视

- 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。
- 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。
- 创建内部类对象的时刻并不依赖于外围类对象的创建。
- 内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。
- 内部类提供了更好的封装，除了该外围类，其他类都不能访问

1.1.4 父类的静态方法能否被子类重写-猎豹

- 不能
- 子类继承父类后，用相同的静态方法和非静态方法，这时非静态方法覆盖父类中的方法（即方法重写），父类的该静态方法被隐藏（如果对象是父类则调用该隐藏的方法），另外子类可继承父类的静态与非静态方法，至于方法重载我觉得它其中一要素就是在同一类中，不能说父类中的什么方法与子类里的什么方法是方法重载的体现

1.1.5 java 排序算法-美团

- http://blog.csdn.net/qy1387/article/details/7752973

1.1.6 列举 java 的集合和继承关系-百度-美团

1.1.7 java 虚拟机的特性-百度-乐视

- Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用模式 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

1.1.8 哪些情况下的对象会被垃圾回收机制处理掉-美团-小米

- Java 垃圾回收机制最基本的做法是分代回收。内存中的区域被划分成不同的世代，对象根据其存活的时间被保存在对应世代的区域中。一般的实现是划分成 3 个世代：年轻、年老和永久。内存的分配是发生在年轻世代中的。当一个对象存活时间足够长的时候，它就会被复制到年老世代中。对于不同的世代可以使用不同的垃圾回收算法。进行世代划分的出发点是对应用中对象存活时间进行研究之后得出的统计规律。一般来说，一个应用中的大部分对象的存活时间都很短。比如局部变量的存活时间就只在方法的执行过程中。基于这一点，对于年轻世代的垃圾回收算法就可以很有针对性。

1.1.9 进程和线程的区别-猎豹-美团

- 简而言之, 一个程序至少有一个进程, 一个进程至少有一个线程。
- 线程的划分尺度小于进程，使得多线程程序的并发性高。
- 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
- 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，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。
- 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。
- 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动, 进程是系统进行资源分配和调度的一个独立单位.
- 线程是进程的一个实体, 是 CPU 调度和分派的基本单位, 它是比进程更小的能独立运行的基本单位. 线程自己基本上不拥有系统资源, 只拥有一点在运行中必不可少的资源 (如程序计数器, 一组寄存器和栈), 但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.
- 一个线程可以创建和撤销另一个线程; 同一个进程中的多个线程之间可以并发执行.
- 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些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。如果有兴趣深入的话，我建议你们看看《现代操作系统》或者《操作系统的设计与实现》。对就个问题说得比较清楚。

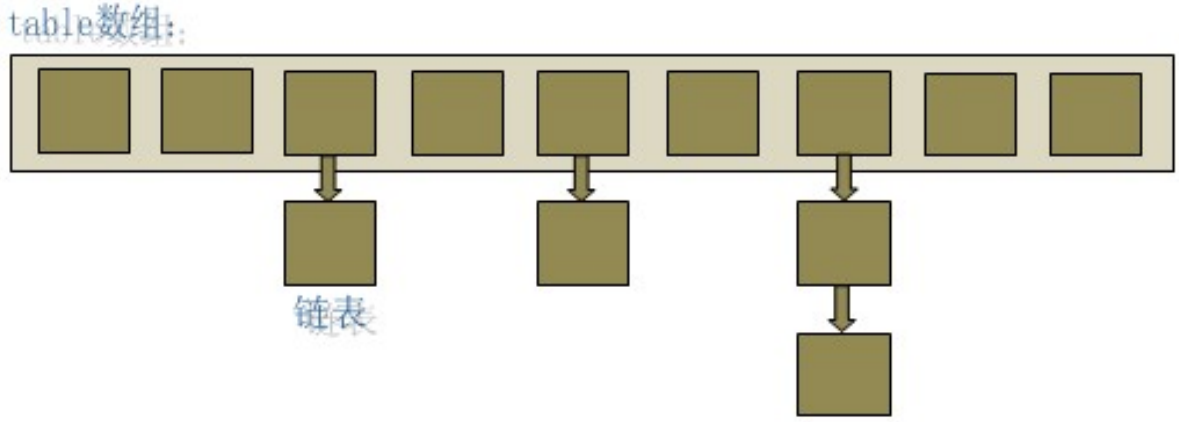
1.1.10 java 中 == 和 equals 和 hashCode 的区别-乐视

- http://blog.csdn.net/tiantiandjava/article/details/46988461

1.1.11 常见的排序算法时间复杂度-小米

1.1.12 HashMap 的实现原理-美团

- HashMap 概述：HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
- HashMap 的数据结构：在 java 编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap 也不例外。HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。



- 从上图中可以看出，HashMap 底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个 HashMap 的时候，就会初始化一个数组。

1.1.13 状态机

- http://www.jdon.com/designpatterns/designpattern_State.htm

1.1.14 int-char-long 各占多少字节数

- byte 位数 8 字节数 1
- short 16 2
- int 32 4
- long 64 8
- float 32 4
- double 64 8
- char 16 2

1.1.15 int 与 integer 的区别

- http://www.cnblogs.com/shenliang123/archive/2011/10/27/2226903.html

1.1.16 string-stringbuffer-stringbuilder 区别-小米-乐视-百度

- String 字符串常量
- StringBuffer 字符串变量（线程安全）
- StringBuilder 字符串变量（非线程安全）

1. String

   - 简要的说，String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后,JVM 的 GC 就会开始工作，那速度是一定会相当慢的。
   - 而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下，String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中，String 效率是远要比 StringBuffer 快的：

```
String S1 = "This is only a" + "simple" + " test";
StringBuffer Sb = new StringBuffer("This is only a").append("simple").append("test");
```

   - 你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个 String S1 = “This is only a”+ “simple”+ “test”; 其实就是：String S1 = “This is only a simple test”; 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：String S2 = “This is only a”; String S3 = “simple”; String S4 = “test”; String S1 = S2 +S3 + S4; 这时候 JVM 会规规矩矩的按照原来的方式去做
   - 在大部分情况下 StringBuffer > String

2. StringBuffer

   - Java.lang.StringBuffer 线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。
   - 可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。
   - StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。

- 例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append(”le”) 会使字符串缓冲区包含“startle”，而 z.insert(4, ”le”) 将更改字符串缓冲区，使之包含“starlet”。
- 在大部分情况下 StringBuilder > StringBuffer

3. java.lang.StringBuilder

   - java.lang.StringBuilder 一个可变的字符序列是 5.0 新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同

1.1.17 java 多态-乐视

- Java 多态性理解
- Java 中多态性的实现
- 什么是多态
- 面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。
- 多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）
- 实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

1. 多态的作用：消除类型之间的耦合关系

   - 现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。下面是多态存在的三个必要条件，要求大家做梦时都能背出来！
   - 多态存在的三个必要条件一、要有继承；二、要有重写；三、父类引用指向子类对象。

2. 多态的好处：

   - 1. 可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。
   - 2. 可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。
   - 3. 接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图 8.3 所示。图中超类 Shape 规定了两个实现多态的接口方法，computeArea() 以及 computeVolume()。子类，如 Circle 和 Sphere 为了实现多态，完善或者覆盖这两个接口方法。
   - 4. 灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。
   - 5. 简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

3. Java 中多态的实现方式：

   - 接口实现，继承父类进行方法重写，同一个类中进行方法重载。

1.1.18 什么导致线程阻塞-58-美团

- 线程的阻塞
- 为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持.
- 阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。
  - sleep() 方法：sleep() 允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。
  - suspend() 和 resume() 方法：两个方法配套使用，suspend() 使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。
  - yield() 方法：yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程.
  - wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.
- 初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。
- 上述的核心区别导致了一系列的细节上的区别。
  - 首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的 notify() 方法则导致因调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。
  - 其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 IllegalMonitorStateException 异常。

- wait() 和 notify() 方法的上述特性决定了它们经常和 synchronized 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized 方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和 wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

- 关于 wait() 和 notify() 方法最后再说明两点：

  - 第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。
  - 第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

- 谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

- 以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

1.1.19 抽象类接口区别-360

- 默认的方法实现抽象类可以有默认的方法实现完全是抽象的。接口根本不存在方法的实现

- 实现子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。子类使用关键字 implements 来实现接口。它需要提供接口中所有声明的方法的实现

- 构造器抽象类可以有构造器接口不能有构造器

- 与正常 Java 类的区别除了你不能实例化抽象类之外，它和普通 Java 类没有任何区接口是完全不同的类型

- 访问修饰符抽象方法可以有 public、protected 和 default 这些修饰符接口方法默认修饰符是 public。你不可以使用其它修饰符。

- main 方法抽象方法可以有 main 方法并且我们可以运行它接口没有 main 方法，因此我们不能运行它。

- 多继承抽象类在 java 语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

- 速度它比接口速度要快接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。

- 添加新方法如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。如果你往接口中添加方法，那么你必须改变实现该接口的类。

1.1.20 容器类之间的区别-乐视-美团

- http://www.cnblogs.com/yuanermen/archive/2009/08/05/1539917.html http://alexyyek.github.io/2015/04/06/Collection/ http://tianmaying.com/tutorial/java_collection

1.1.21 内部类

- http://www.cnblogs.com/chenssy/p/3388487.html

1.1.22 hashmap 和 hashtable 的区别-乐视-小米

- http://www.233.com/ncre2/JAVA/jichu/20100717/084230917.html

1.1.23 ArrayMap 对比 HashMap

- http://lvable.com/?p=217

1.2 Android

1.2.1 如何导入外部数据库

- 把原数据库包括在项目源码的 res/raw
- android 系统下数据库应该存放在 *data/data/com..*（*package name*） 目录下，所以我们需要做的是把已有的数据库传入那个目录下. 操作方法是用 FileInputStream 读取原数据库，再用 FileOutputStream 把读取到的东西写入到那个目录.

1.2.2 本地广播和全局广播有什么差别

- 因广播数据在本应用范围内传播，不用担心隐私数据泄露的问题。不用担心别的应用伪造广播，造成安全隐患。相比在系统内发送全局广播，它更高效。

1.2.3 intentService 作用是什么,AIDL 解决了什么问题-小米

- 生成一个默认的且与主线程互相独立的工作者线程来执行所有传送至 onStartCommand() 方法的 Intetnt。
- 生成一个工作队列来传送 Intent 对象给你的 onHandleIntent() 方法，同一时刻只传送一个 Intent 对象，这样一来，你就不必担心多线程的问题。在所有的请求 (Intent) 都被执行完以后会自动停止服务，所以，你不需要自己去调用 stopSelf() 方法来停止。
- 该服务提供了一个 onBind() 方法的默认实现，它返回 null
- 提供了一个 onStartCommand() 方法的默认实现，它将 Intent 先传送至工作队列，然后从工作队列中每次取出一个传送至 onHandleIntent() 方法，在该方法中对 Intent 对相应的处理。
- AIDL (Android Interface Definition Language) 是一种 IDL 语言，用于生成可以在 Android 设备上两个进程之间进行进程间通信 (interprocess communication, IPC) 的代码。如果在一个进程中（例如 Activity）要调用另一个进程中（例如 Service）对象的操作，就可以使用 AIDL 生成可序列化的参数。AIDL IPC 机制是面向接口的，像 COM 或 Corba 一样，但是更加轻量级。它是使用代理类在客户端和实现端传递数据。

1.2.4 Activity/Window/View 三者的差别,fragment 的特点-360

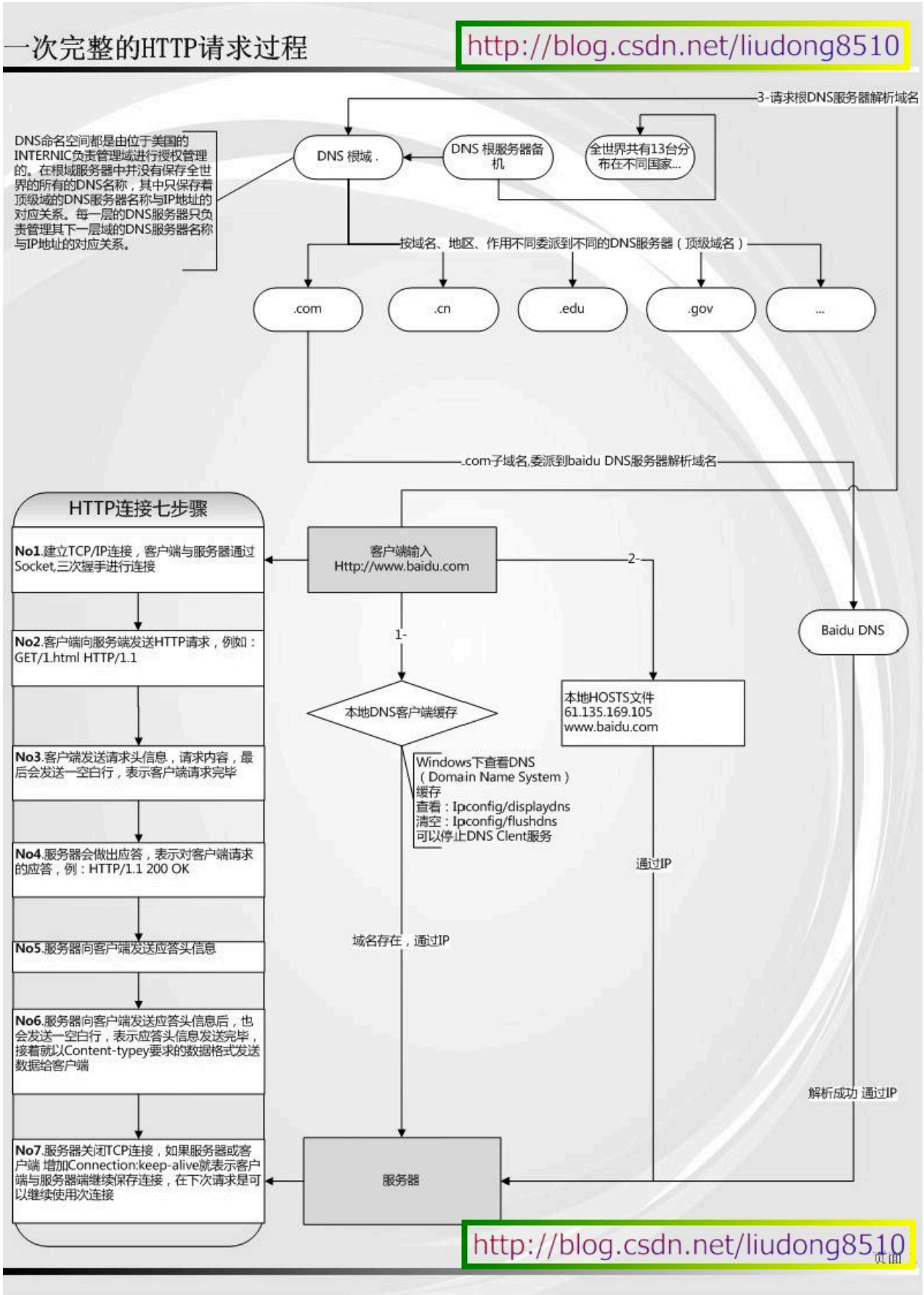
- Activity 像一个工匠（控制单元），Window 像窗户（承载模型），View 像窗花（显示视图）LayoutInflater 像剪刀，Xml 配置像窗花图纸。

  - 在 Activity 中调用 attach，创建了一个 Window
  - 创建的 window 是其子类 PhoneWindow，在 attach 中创建 PhoneWindow
  - 在 Activity 中调用 setContentView(R.layout.xxx)
  - 其中实际上是调用的 getWindow().setContentView()
  - 调用 PhoneWindow 中的 setContentView 方法
  - 创建 ParentView： 作为 ViewGroup 的子类，实际是创建的 DecorView(作为 FramLayout 的子类）
  - 将指定的 R.layout.xxx 进行填充 通过布局填充器进行填充【其中的 parent 指的就是 DecorView】
  - 调用到 ViewGroup
  - 调用 ViewGroup 的 removeAllView()，先将所有的 view 移除掉
  - 添加新的 view：addView()

1. fragment 特点

   - Fragment 可以作为 Activity 界面的一部分组成出现；
   - 可以在一个 Activity 中同时出现多个 Fragment，并且一个 Fragment 也可以在多个 Activity 中使用；
   - 在 Activity 运行过程中，可以添加、移除或者替换 Fragment；
   - Fragment 可以响应自己的输入事件，并且有自己的生命周期，它们的生命周期会受宿主 Activity 的生命周期影响。

1.2.5 描述一次网络请求的流程-新浪



1.2.6 Handler,Thread 和 HandlerThread 的差别-小米

- http://blog.csdn.net/guolin_blog/article/details/9991569
- http://droidyue.com/blog/2015/11/08/make-use-of-handlerthread/
- 从 Android 中 Thread（java.lang.Thread -> java.lang.Object）描述可以看出，Android 的 Thread 没有对 Java 的 Thread 做任何封装，但是 Android 提供了一个继承自 Thread 的类 HandlerThread（android.os.HandlerThread -> java.lang.Thread），这个类对 Java 的 Thread 做了很多便利 Android 系统的封装。
- android.os.Handler 可以通过 Looper 对象实例化，并运行于另外的线程中，Android 提供了让 Handler 运行于其它线程的线程实现，也是就 HandlerThread。HandlerThread 对象 start 后可以获得其 Looper 对象，并且使用这个 Looper 对象实例 Handler。

1.2.7 低版本 SDK 实现高版本 api-小米

- 自己实现或 @TargetApi annotation

1.2.8 Ubuntu 编译安卓系统-百度

- 进入源码根目录
- . build/envsetup.sh
- lunch
- full(编译全部)
- userdebug(选择编译版本)
- make -j8(开启 8 个线程编译)

1.2.9 LaunchMode 应用场景-百度-小米-乐视

- standard，创建一个新的 Activity。
- singleTop，栈顶不是该类型的 Activity，创建一个新的 Activity。否则，onNewIntent。
- singleTask，回退栈中没有该类型的 Activity，创建 Activity，否则，onNewIntent+ClearTop。
- 注意:
  - 设置了”singleTask” 启动模式的 Activity，它在启动的时候，会先在系统中查找属性值 affinity 等于它的属性值 taskAffinity 的 Task 存在；如果存在这样的 Task，它就会在这个 Task 中启动，否则就会在新的任务栈中启动。因此，如果我们想要设置了”singleTask” 启动模式的 Activity 在新的任务中启动，就要为它设置一个独立的 taskAffinity 属性值。
  - 如果设置了”singleTask” 启动模式的 Activity 不是在新的任务中启动时，它会在已有的任务中查看是否已经存在相应的 Activity 实例，如果存在，就会把位于这个 Activity 实例上面的 Activity 全部结束掉，即最终这个 Activity 实例会位于任务的 Stack 顶端中。
  - 在一个任务栈中只有一个”singleTask”启动模式的 Activity 存在。他的上面可以有其他的 Activity。这点与 singleInstance 是有区别的。

- singleInstance，回退栈中，只有这一个 Activity，没有其他 Activity。
- singleTop 适合接收通知启动的内容显示页面。
- 例如，某个新闻客户端的新闻内容页面，如果收到 10 个新闻推送，每次都打开一个新闻内容页面是很烦人的。
- singleTask 适合作为程序入口点。
- 例如浏览器的主界面。不管从多少个应用启动浏览器，只会启动主界面一次，其余情况都会走 onNewIntent，并且会清空主界面上面的其他页面。
- singleInstance 应用场景：
  - 闹铃的响铃界面。你以前设置了一个闹铃：上午 6 点。在上午 5 点 58 分，你启动了闹铃设置界面，并按 Home 键回桌面；在上午 5 点 59 分时，你在微信和朋友聊天；在 6 点时，闹铃响了，并且弹出了一个对话框形式的 Activity(名为 AlarmAlertActivity) 提示你到 6 点了 (这个 Activity 就是以 SingleInstance 加载模式打开的)，你按返回键，回到的是微信的聊天界面，这是因为 AlarmAlertActivity 所在的 Task 的栈只有他一个元素，因此退出之后这个 Task 的栈空了。如果是以 SingleTask 打开 AlarmAlertActivity，那么当闹铃响了的时候，按返回键应该进入闹铃设置界面。

1.2.10 Touch 事件传递流程-小米

http://hanhailong.com/2015/09/24/Android-%E4%B8%89%E5%BC%A0%E5%9B%BE%E6%90%9E%E5%AE%9ATouch%E4%BA%8B%E4%BB%B6%E4%BC%A0%E9%80%92%E6%9C%BA%E5%88%B6/

1.2.11 View 绘制流程-百度

http://www.codekk.com/blogs/detail/54cfab086c4761e5001b253f

1.2.12 多线程-360

- Activity.runOnUiThread(Runnable)
- View.post(Runnable),View.postDelay(Runnable,long)
- Handler
- AsyncTask

1.2.13 线程同步-百度

- http://www.itzhai.com/java-based-notebook-thread-synchronization-problem-solving-synchrhtml#read-more
- http://www.juwends.com/tech/android/android-inter-thread-comm.html
- 单例

```
public class Singleton {
    private volatile static Singleton mSingleton;

    private Singleton(){
    }

    public static Singleton getInstance(){
        if (mSingleton == null){ // A
            synchronized(Singleton.class){ // C
                if(mSingleton == null)
                    mSingleton = new Singleton(); // B
            }
        }
        return mSingleton;
    }
}
```

1.2.14 什么情况导致内存泄漏-美团

- 1. 资源对象没关闭造成的内存泄漏
  - 描述：资源性对象比如 (Cursor，File 文件等) 往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 java 虚拟机内，还存在于 java 虚拟机外。如果我们仅仅是把它的引用设置为 null, 而不关闭它们，往往会造成内存泄漏。因为有些资源性对象，比如 SQLiteCursor(在析构函数 finalize(), 如果我们没有关闭它，它自己会调 close() 关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的 close() 函数，将其关闭掉，然后才置为 null. 在我们的程序退出时一定要确保我们的资源性对象已经关闭。程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。
- 2. 构造 Adapter 时，没有使用缓存的 convertView
  - 描述：以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提供了方法：public View getView(int position, ViewconvertView, ViewGroup parent) 来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView() 方法完成的，getView() 的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象 (初始化时缓存中没有 view 对象则 convertView 是 null)。由此可以看出，如果我们不去使用 convertView，而是每次都在 getView() 中重新实例化一个 View 对象的话，即浪费资源也浪费时间，也会使得内存占用越来越大。ListView 回收 list item 的 view 对象的过程可以查看: android.widget.AbsListView.java –> voidaddScrapView(View scrap) 方法。示例代码:

```
public View getView(int position, ViewconvertView, ViewGroup parent) {
    View view = new Xxx(...);
    ....
```

```
    return view;
}
```

  - 修正示例代码:

```
public View getView(int position, ViewconvertView, ViewGroup parent) {
    View view = null;
    if (convertView != null) {
        view = convertView;
        populate(view, getItem(position));
    } else {
        view = new Xxx(...);
    }
    return view;
}
```

- 3.Bitmap 对象不在使用时调用 recycle() 释放内存
  - 描述：有时我们会手工的操作 Bitmap 对象，如果一个 Bitmap 对象比较占内存，当它不在被使用的时候，可以调用 Bitmap.recycle() 方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。可以看一下代码中的注释:
  - *** • Free up the memory associated with thisbitmap's pixels, and mark the • bitmap as "dead", meaning itwill throw an exception if getPixels() or • setPixels() is called, and will drawnothing. This operation cannot be • reversed, so it should only be called ifyou are sure there are no • further uses for the bitmap. This is anadvanced call, and normally need • not be called, since the normal GCprocess will free up this memory when • there are no more references to thisbitmap. *

- 4. 试着使用关于 application 的 context 来替代和 activity 相关的 context
  - 这是一个很隐晦的内存泄漏的情况。有一种简单的方法来避免 context 相关的内存泄漏。最显著地一个是避免 context 逃出他自己的范围之外。使用 Application context。这个 context 的生存周期和你的应用的生存周期一样长，而不是取决于 activity 的生存周期。如果你想保持一个长期生存的对象，并且这个对象需要一个 context, 记得使用 application 对象。你可以通过调用 Context.getApplicationContext() or Activity.getApplication() 来获得。更多的请看这篇文章如何避免 Android 内存泄漏。

- 5. 注册没取消造成的内存泄漏
  - 一些 Android 程序可能引用我们的 Anroid 程序的对象 (比如注册机制)。即使我们的 Android 程序已经结束了, 但是别的引用程序仍然还有对我们的 Android 程序的某个对象的引用, 泄漏的内存依然不能被垃圾回收。调用 registerReceiver 后未调用 unregisterReceiver。比如: 假设我们希望在锁屏界面 (LockScreen) 中，监听系统中的电话服务以获取一些信息 (如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory, 使得 $\text{system}_{process}$ 进程挂掉。虽然有些系统程序，它本身好像是可以自动取消注册的 (当然不及时)，但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

- 6. 集合中对象没清理造成的内存泄漏
  - 我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

1.2.15 ANR 定位和修正

如果开发机器上出现问题，我们可以通过查看/data/anr/traces.txt 即可，最新的 ANR 信息在最开始部分。

- 主线程被 IO 操作（从 4.0 之后网络 IO 不允许在主线程中）阻塞。
- 主线程中存在耗时的计算
- 主线程中错误的操作，比如 Thread.wait 或者 Thread.sleep 等 Android 系统会监控程序的响应状况，一旦出现下面两种情况，则弹出 ANR 对话框
- 应用在 5 秒内未响应用户的输入事件（如按键或者触摸）
- BroadcastReceiver 未在 10 秒内完成相关的处理
- Service 在特定的时间内无法处理完成 20 秒
- 使用 AsyncTask 处理耗时 IO 操作。
- 使用 Thread 或者 HandlerThread 时，调用 Process.setThreadPriority(Process.THREAD$_{PRIORITY\,BACKGRO}$ 设置优先级，否则仍然会降低程序响应，因为默认 Thread 的优先级和主线程相同。
- 使用 Handler 处理工作线程结果，而不是使用 Thread.wait() 或者 Thread.sleep() 来阻塞主线程。
- Activity 的 onCreate 和 onResume 回调中尽量避免耗时的代码
- BroadcastReceiver 中 onReceive 代码也要尽量减少耗时，建议使用 IntentService 处理。

1.2.16 什么情况导致 oom-乐视-美团

http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2015/0920/3478.html

- 1）使用更加轻量的数据结构
- 2）Android 里面使用 Enum
- 3）Bitmap 对象的内存占用
- 4）更大的图片
- 5）onDraw 方法里面执行对象的创建
- 6）StringBuilder

1.2.17 Service 与 Activity 之间通信的几种方式

- 通过 Binder 对象
- 通过 broadcast(广播) 的形式

1.2.18 Android 各个版本 API 的区别

- http://blog.csdn.net/lijun952048910/article/details/7980562

1.2.19 Android 代码中实现 WAP 方式联网-360

- http://blog.csdn.net/asce1885/article/details/7844159

1.2.20 如何保证 service 在后台不被 Kill

- 一、onStartCommand 方法，返回 START$_{STICKY}$
  - START$_{STICKY}$ 在运行 onStartCommand 后 service 进程被 kill 后，那将保留在开始状态，但是不保留那些传入的 intent。不久后 service 就会再次尝试重新创建，因为保留在开始状态，在创建 service 后将保证调用 onstartCommand。如果没有传递任何开始命令给 service，那将获取到 null 的 intent。
  - START$_{NOTSTICKY}$ 在运行 onStartCommand 后 service 进程被 kill 后，并且没有新的 intent 传递给它。Service 将移出开始状态，并且直到新的明显的方法（startService）调用才重新创建。因为如果没有传递任何未决定的 intent 那么 service 是不会启动，也就是期间 onstartCommand 不会接收到任何 null 的 intent。
  - START$_{REDELIVERINTENT}$ 在运行 onStartCommand 后 service 进程被 kill 后，系统将会再次启动 service，并传入最后一个 intent 给 onstartCommand。直到调用 stopSelf(int) 才停止传递 intent。如果在被 kill 后还有未处理好的 intent，那被 kill 后服务还是会自动启动。因此 onstartCommand 不会接收到任何 null 的 intent。
- 二、提升 service 优先级
  - 在 AndroidManifest.xml 文件中对于 intent-filter 可以通过 android:priority = "1000" 这个属性设置最高优先级，1000 是最高值，如果数字越小则优先级越低，同时适用于广播。
- 三、提升 service 进程优先级

  Android 中的进程是托管的，当系统进程空间紧张的时候，会依照优先级自动进行进程的回收。Android 将进程分为 6 个等级, 它们按优先级顺序由高到低依次是:
  - 前台进程 (FOREGROUND$_{APP}$)
  - 可视进程 (VISIBLE$_{APP}$)
  - 次要服务进程 (SECONDARY$_{SERVER}$)
  - 后台进程 (HIDDEN$_{APP}$)
  - 内容供应节点 (CONTENT$_{PROVIDER}$)
  - 空进程 (EMPTY$_{APP}$)

  当 service 运行在低内存的环境时，将会 kill 掉一些存在的进程。因此进程的优先级将会很重要，可以使用 startForeground 将 service 放到前台状态。这样在低内存时被 kill 的几率会低一些。
- 四、onDestroy 方法里重启 service
  - service +broadcast 方式，就是当 service 走 ondestory 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 service;

- 五、Application 加上 Persistent 属性
- 六、监听系统广播判断 Service 状态
  - 通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的 Service 是否还存活，别忘记加权限啊。

1.2.21 Requestlayout,onlayout,onDraw,DrawChild 区别与联系-猎豹

- requestLayout() 方法：会导致调用 measure() 过程和 layout() 过程。将会根据标志位判断是否需要 ondraw
- onLayout() 方法 (如果该 View 是 ViewGroup 对象，需要实现该方法，对每个子视图进行布局)
- 调用 onDraw() 方法绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)
- drawChild() 去重新回调每个子视图的 draw() 方法

1.2.22 invalidate() 和 postInvalidate() 的区别及使用-百度

http://blog.csdn.net/mars2639/article/details/6650876

1.2.23 Android 动画框架实现原理

- Animation 框架定义了透明度，旋转，缩放和位移几种常见的动画，而且控制的是整个 View，实现原理是每次绘制视图时 View 所在的 ViewGroup 中的 drawChild 函数获取该 View 的 Animation 的 Transformation 值，然后调用 canvas.concat(transformToApply.getMatrix())，通过矩阵运算完成动画帧，如果动画没有完成，继续调用 invalidate() 函数，启动下次绘制来驱动动画，动画过程中的帧之间间隙时间是绘制函数所消耗的时间，可能会导致动画消耗比较多的 CPU 资源，最重要的是，动画改变的只是显示，并不能相应事件。

1.2.24 Android 为每个应用程序分配的内存大小是多少-美团

- android 程序内存一般限制在 16M，也有的是 24M

1.2.25 View 刷新机制-百度-美团

- 由 ViewRoot 对象的 performTraversals() 方法调用 draw() 方法发起绘制该 View 树，值得注意的是每次发起绘图时，并不会重新绘制每个 View 树的视图，而只会重新绘制那些“需要重绘”的视图，View 类内部变量包含了一个标志位 DRAWN，当该视图需要重绘时，就会为该 View 添加该标志位。
- 调用流程：
- mView.draw() 开始绘制，draw() 方法实现的功能如下：
  - 绘制该 View 的背景
  - 为显示渐变框做一些准备操作 (见 5，大多数情况下，不需要改渐变框)
  - 调用 onDraw() 方法绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)

  - 调用 dispatchDraw () 方法绘制子视图 (如果该 View 类型不为 ViewGroup，即不包含子视图，不需要重载该方法) 值得说明的是，ViewGroup 类已经为我们重写了 dispatchDraw () 的功能实现，应用程序一般不需要重写该方法，但可以重载父类函数实现具体的功能。

1.2.26 LinearLayout 和 RelativeLayout 性能对比-百度

- RelativeLayout 会让子 View 调用 2 次 onMeasure，LinearLayout 在有 weight 时，也会调用子 View2 次 onMeasure

- RelativeLayout 的子 View 如果高度和 RelativeLayout 不同，则会引发效率问题，当子 View 很复杂时，这个问题会更加严重。如果可以，尽量使用 padding 代替 margin。

- 在不影响层级深度的情况下, 使用 LinearLayout 和 FrameLayout 而不是 RelativeLayout。

- 最后再思考一下文章开头那个矛盾的问题，为什么 Google 给开发者默认新建了个 RelativeLayout，而自己却在 DecorView 中用了个 LinearLayout。因为 DecorView 的层级深度是已知而且固定的，上面一个标题栏，下面一个内容栏。采用 RelativeLayout 并不会降低层级深度，所以此时在根节点上用 LinearLayout 是效率最高的。而之所以给开发者默认新建了个 RelativeLayout 是希望开发者能采用尽量少的 View 层级来表达布局以实现性能最优，因为复杂的 View 嵌套对性能的影响会更大一些。

1.2.27 优化自定义 view 百度-乐视-小米

- 为了加速你的 view，对于频繁调用的方法，需要尽量减少不必要的代码。先从 onDraw 开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致 GC，从而导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。

- 你还需要尽可能的减少 onDraw 被调用的次数，大多数时候导致 onDraw 都是因为调用了 invalidate(). 因此请尽量减少调用 invaildate() 的次数。如果可能的话，尽量调用含有 4 个参数的 invalidate() 方法而不是没有参数的 invalidate()。没有参数的 invalidate 会强制重绘整个 view。

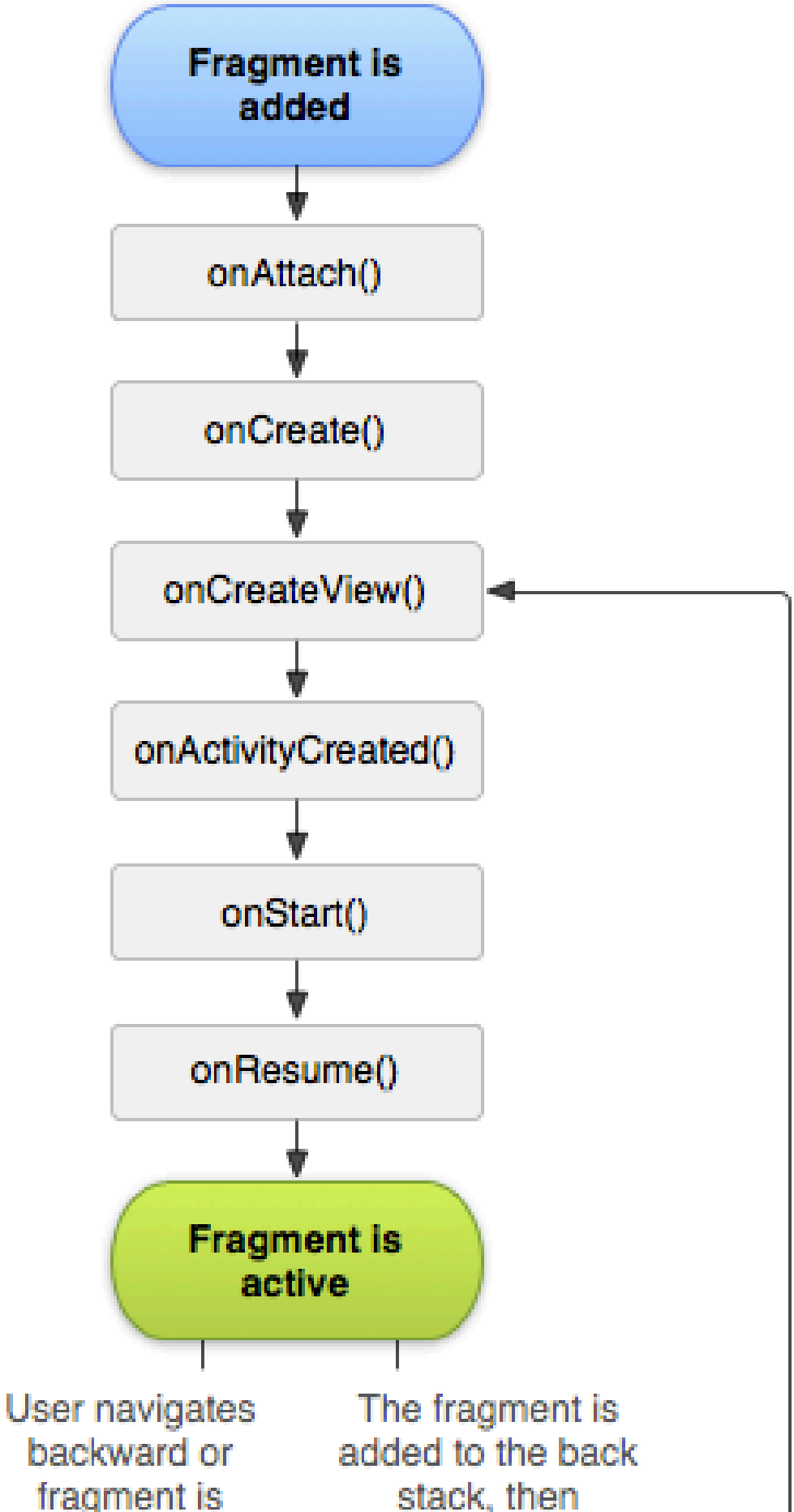
- 另外一个非常耗时的操作是请求 layout。任何时候执行 requestLayout()，会使得 Android UI 系统去遍历整个 View 的层级来计算出每一个 view 的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持 View 的层级是扁平化的，这样对提高效率很有帮助。

- 如果你有一个复杂的 UI，你应该考虑写一个自定义的 ViewGroup 来执行他的 layout 操作。与内置的 view 不同，自定义的 view 可以使得程序仅仅测量这一部分，这避免了遍历整个 view 的层级结构来计算大小。这个 PieChart 例子展示了如何继承 ViewGroup 作为自定义 view 的一部分。PieChart 有子 views，但是它从来不测量它们。而是根据他自身的 layout 法则，直接设置它们的大小。

1.2.28 ContentProvider-乐视

http://blog.csdn.net/coder_pig/article/details/47858489

1.2.29 Fragment 生命周期



1.2.30 volley 解析-美团-乐视

http://a.codekk.com/detail/Android/grumoon/Volley%20%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90

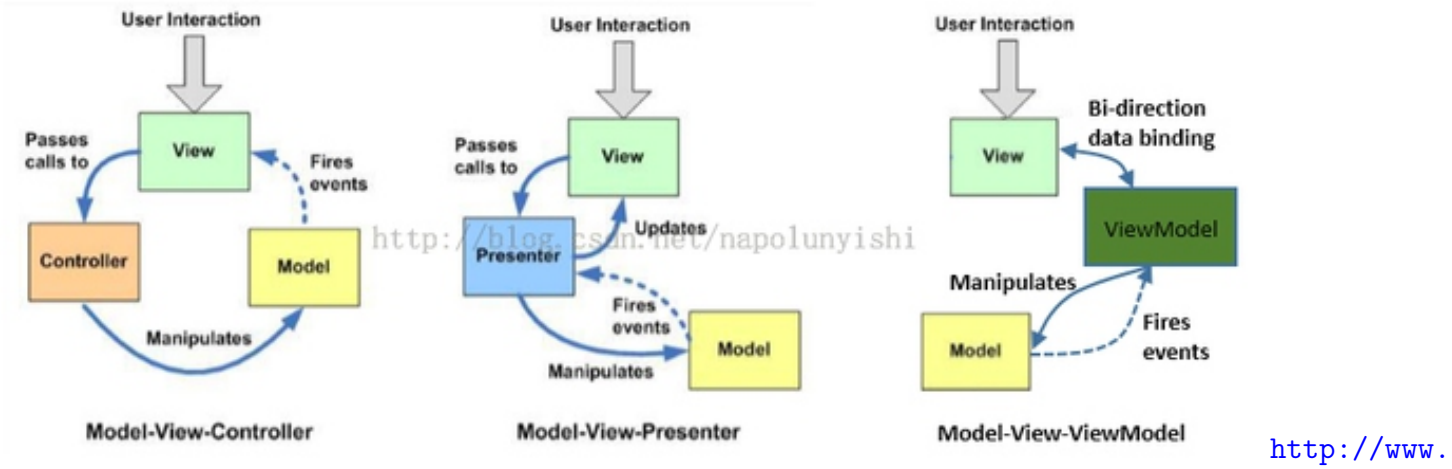
1.2.31 Glide 源码解析

http://www.lightskystreet.com/2015/10/12/glide_source_analysis/ http://frodoking.github.io/2015/10/10/android-glide/

1.2.32 Android 设计模式

http://blog.csdn.net/bboyfeiyu/article/details/44563871

1.2.33 架构设计-搜狐



http://www.tianmaying.com/tutorial/AndroidMVC

1.2.34 Android 属性动画特性-乐视-小米

- 如果你的需求中只需要对 View 进行移动、缩放、旋转和淡入淡出操作，那么补间动画确实已经足够健全了。但是很显然，这些功能是不足以覆盖所有的场景的，一旦我们的需求超出了移动、缩放、旋转和淡入淡出这四种对 View 的操作，那么补间动画就不能再帮我们忙了，也就是说它在功能和可扩展方面都有相当大的局限性，那么下面我们就来看看补间动画所不能胜任的场景。

- 注意上面我在介绍补间动画的时候都有使用“对 View 进行操作”这样的描述，没错，补间动画是只能够作用在 View 上的。也就是说，我们可以对一个 Button、TextView、甚至是 LinearLayout、或者其它任何继承自 View 的组件进行动画操作，但是如果我们想要对一个非 View 的对象进行动画操作，抱歉，补间动画就帮不上忙了。可能有的朋友会感到不能理解，我怎么会需要对一个非 View 的对象进行动画操作呢？这里我举一个简单的例子，比如说我们有一个自定义的 View，在这个 View 当中有一个 Point 对象用于管理坐标，然后在 onDraw() 方法当中就是根据这个 Point 对象的坐标值来进行绘制的。也就是说，如果我们可以对 Point 对象进行动画操作，那么整个自定义 View 的动画效果就有了。显然，补间动画是不具备这个功能的，这是它的第一个缺陷。

- 然后补间动画还有一个缺陷，就是它只能够实现移动、缩放、旋转和淡入淡出这四种动画操作，那如果我们希望可以对 View 的背景色进行动态地改变呢？很遗憾，我们只能靠自己去实现了。

说白了，之前的补间动画机制就是使用硬编码的方式来完成的，功能限定死就是这些，基本上没有任何扩展性可言。

- 最后，补间动画还有一个致命的缺陷，就是它只是改变了 View 的显示效果而已，而不会真正去改变 View 的属性。什么意思呢？比如说，现在屏幕的左上角有一个按钮，然后我们通过补间动画将它移动到了屏幕的右下角，现在你可以去尝试点击一下这个按钮，点击事件是绝对不会触发的，因为实际上这个按钮还是停留在屏幕的左上角，只不过补间动画将这个按钮绘制到了屏幕的右下角而已。

1.3 专题

1.3.1 性能优化

1. Android 性能优化典范 - 第 1 季

    - Render Performance Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的 60fps，为了能够实现 60fps，这意味着程序的大多数操作都必须在 16ms 内完成。我们可以通过一些工具来定位问题，比如可以使用 HierarchyViewer 来查找 Activity 中的布局是否过于复杂，也可以使用手机设置里面的开发者选项，打开 Show GPU Overdraw 等选项进行观察。你还可以使用 TraceView 来观察 CPU 的执行情况，更加快捷的找到性能瓶颈。
    - Understanding Overdraw Overdraw(过度绘制) 描述的是屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次的 UI 结构里面，如果不可见的 UI 也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的 CPU 以及 GPU 资源。Overdraw 有时候是因为你的 UI 布局存在大量重叠的部分，还有的时候是因为非必须的重叠背景。例如某个 Activity 有一个背景，然后里面的 Layout 又有自己的背景，同时子 View 又分别有自己的背景。仅仅是通过移除非必须的背景图片，这就能够减少大量的红色 Overdraw 区域，增加蓝色区域的占比。这一措施能够显著提升程序性能。
    - Understanding VSYNC Refresh Rate：代表了屏幕在一秒内刷新屏幕的次数，这取决于硬件的固定参数，例如 60Hz。Frame Rate：代表了 GPU 在一秒内绘制操作的帧数，例如 30fps，60fps。通常来说，帧率超过刷新频率只是一种理想的状况，在超过 60fps 的情况下，GPU 所产生的帧数据会因为等待 VSYNC 的刷新信息而被 Hold 住，这样能够保持每次刷新都有实际的新的数据可以显示。但是我们遇到更多的情况是帧率小于刷新频率。
    - Tool:Profile GPU Rendering 性能问题如此的麻烦，幸好我们可以有工具来进行调试。打开手机里面的开发者选项，选择 Profile GPU Rendering，选中 On screen as bars 的选项。
    - Why 60fps? 我们通常都会提到 60fps 与 16ms，可是知道为何会是以程序是否达到 60fps 来作为 App 性能的衡量标准吗？这是因为人眼与大脑之间的协作无法感知超过 60fps 的画面更新。开发 app 的性能目标就是保持 60fps，这意味着每一帧你只有 16ms=1000/60 的时间来处理所有的任务。
    - Android, UI and the GPU 在 Android 里面那些由主题所提供的资源，例如 Bitmaps，Drawables 都是一起打包到统一的 Texture 纹理当中，然后再传递到 GPU 里面，这意味着每次你需要使用这些资源的时候，都是直接从纹理里面进行获取渲染的。当然随着 UI 组件的越来越丰富，有了更多演变的形态。例如显示图片的时候，需要先经过 CPU 的计算加载到内存中，然后传递给 GPU 进行渲染。文字的显示更加复杂，需要先经过 CPU 换算成纹理，然后再交给 GPU 进行渲染，回到 CPU 绘制单个字符的时候，再重新引用经过 GPU 渲染的内容。动画则是一个更加复杂的操作流程。为了能够使得 App 流畅，我们需要在每一帧 16ms 以内处理完所有的 CPU 与 GPU 计算，绘制，渲染等等操作。

- Invalidations, Layouts, and Performance 任何时候 View 中的绘制内容发生变化时，都会重新执行创建 DisplayList，渲染 DisplayList，更新到屏幕上等一系列操作。这个流程的表现性能取决于你的 View 的复杂程度，View 的状态变化以及渲染管道的执行性能。举个例子，假设某个 Button 的大小需要增大到目前的两倍，在增大 Button 大小之前，需要通过父 View 重新计算并摆放其他子 View 的位置。修改 View 的大小会触发整个 HierarcyView 的重新计算大小的操作。如果是修改 View 的位置则会触发 HierarchView 重新计算其他 View 的位置。如果布局很复杂，这就会很容易导致严重的性能问题。我们需要尽量减少 Overdraw。
- Overdraw, Cliprect, QuickReject 我们可以通过 canvas.clipRect() 来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制，其他的区域会被忽视。这个 API 可以很好的帮助那些有多组重叠组件的自定义 View 来控制显示的区域。同时 clipRect 方法还可以帮助节约 CPU 与 GPU 资源，在 clipRect 区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制。
- Memory Churn and performance 执行 GC 操作的时候，所有线程的任何操作都会需要暂停，等待 GC 操作完成之后，其他操作才能够继续运行。Memory Churn 内存抖动，内存抖动是因为大量的对象被创建又在短时间内马上被释放。瞬间产生大量的对象会严重占用 Young Generation 的内存区域，当达到阀值，剩余空间不够的时候，也会触发 GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加 Heap 的压力，从而触发更多其他类型的 GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。
- Garbage Collection in Android 原始 JVM 中的 GC 机制在 Android 中得到了很大程度上的优化。Android 里面是一个三级 Generation 的内存模型，最近分配的对象会存放在 Young Generation 区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到 Old Generation，最后到 Permanent Generation 区域。如果不小心在最小的 for 循环单元里面执行了创建对象的操作，这将很容易引起 GC 并导致性能问题。通过 Memory Monitor 我们可以查看到内存的占用情况，每一次瞬间的内存降低都是因为此时发生了 GC 操作，如果在短时间内发生大量的内存上涨与降低的事件，这说明很有可能这里有性能问题。我们还可以通过 Heap and Allocation Tracker 工具来查看此时内存中分配的到底有哪些对象。
- Performance Cost of Memory Leaks 内存泄漏指的是那些程序不再使用的对象无法被 GC 识别，这样就导致这个对象一直留在内存当中，占用了宝贵的内存空间。显然，这还使得每级 Generation 的内存区域可用空间变小，GC 就会更容易被触发，从而引起性能问题。
- Memory Performance 通常来说，Android 对 GC 做了大量的优化操作，虽然执行 GC 操作的时候会暂停其他任务，可是大多数情况下，GC 操作还是相对很安静并且高效的。但是如果我们对内存的使用不恰当，导致 GC 频繁执行，这样就会引起不小的性能问题。
- Tool - Memory Monitor Android Studio 中的 Memory Monitor 可以很好的帮助我们查看程序的内存使用情况。
- Battery Performance 我们应该尽量减少唤醒屏幕的次数与持续的时间，使用 WakeLock 来处理唤醒的问题，能够正确执行唤醒操作并根据设定及时关闭操作进入睡眠状态。某些非必须马上执行的操作，例如上传歌曲，图片处理等，可以等到设备处于充电状态或者电量充足的时候才进行。触发网络请求的操作，每次都会保持无线信号持续一段时间，我们可以把零散的网络请求打包进行一次操作，避免过多的无线信号引起的电量消耗。关于网络请求引起无线信号的电量消耗
- Understanding Battery Drain on Android 使用 WakeLock 或者 JobScheduler 唤醒设备处理定时的任务之后，一定要及时让设备回到初始状态。每次唤醒无线信号进行数据传递，都会消耗很多电量，它比 WiFi 等操作更加的耗电
- Battery Drain and WakeLocks 这正是 JobScheduler API 所做的事情。它会根据当前的情况

与任务，组合出理想的唤醒时间，例如等到正在充电或者连接到 WiFi 的时候，或者集中任务一起执行。我们可以通过这个 API 实现很多免费的调度算法。

2. Android 性能优化典范 - 第 2 季

- Battery Drain and Networking 我们可以有针对性的把请求行为捆绑起来，延迟到某个时刻统一发起请求。这部分主要会涉及到 Prefetch(预取) 与 Compressed(压缩) 这两个技术。对于 Prefetch 的使用，我们需要预先判断用户在此次操作之后，后续零散的请求是否很有可能会马上被触发，可以把后面 5 分钟有可能会使用到的零散请求都一次集中执行完毕。对于 Compressed 的使用，在上传与下载数据之前，使用 CPU 对数据进行压缩与解压，可以很大程度上减少网络传输的时间。
- Wear & Sensors 首先我们需要尽量使用 Android 平台提供的既有运动数据，而不是自己去实现监听采集数据，因为大多数 Android Watch 自身记录 Sensor 数据的行为是有经过做电量优化的。其次在 Activity 不需要监听某些 Sensor 数据的时候需要尽快释放监听注册。还有我们需要尽量控制更新的频率，仅仅在需要刷新显示数据的时候才触发获取最新数据的操作。另外我们可以针对 Sensor 的数据做批量处理，待数据累积一定次数或者某个程度的时候才更新到 UI 上。最后当 Watch 与 Phone 连接起来的时候，可以把某些复杂操作的事情交给 Phone 来执行，Watch 只需要等待返回的结果。
- Smooth Android Wear Animation 在 Android 里面一个相对操作比较繁重的事情是对 Bitmap 进行旋转，缩放，裁剪等等。例如在一个圆形的钟表图上，我们把时钟的指针抠出来当做单独的图片进行旋转会比旋转一张完整的圆形图的所形成的帧率要高 56%。
- Android Wear Data Batching 仅仅在真正需要刷新界面的时候才发出请求，尽量把计算复杂操作的任务交给 Phone 来处理，Phone 仅仅在数据发生变化的时候才通知到 Wear，把零碎的数据请求捆绑一起再进行操作。
- Object Pools 使用对象池技术有很多好处，它可以避免内存抖动，提升性能，但是在使用的时候有一些内容是需要特别注意的。通常情况下，初始化的对象池里面都是空白的，当使用某个对象的时候先去对象池查询是否存在，如果不存在则创建这个对象然后加入对象池，但是我们也可以在程序刚启动的时候就事先为对象池填充一些即将要使用到的数据，这样可以在需要使用到这些对象的时候提供更快的首次加载速度，这种行为就叫做预分配。使用对象池也有不好的一面，程序员需要手动管理这些对象的分配与释放，所以我们需要慎重地使用这项技术，避免发生对象的内存泄漏。为了确保所有的对象能够正确被释放，我们需要保证加入对象池的对象和其他外部对象没有互相引用的关系。
- To Index or Iterate? for index 的方式有更好的效率，但是因为不同平台编译器优化各有差异，我们最好还是针对实际的方法做一下简单的测量比较好，拿到数据之后，再选择效率最高的那个方式。
- The Magic of LRU Cache 使用 LRU Cache 能够显著提升应用的性能，可是也需要注意 LRU Cache 中被淘汰对象的回收，否者会引起严重的内存泄露。
- Using LINT for Performance Tips Lint 已经集成到 Android Studio 中了，我们可以手动去触发这个工具，点击工具栏的 Analysis -> Inspect Code，触发之后，Lint 会开始工作，并把结果输出到底部的工具栏，我们可以逐个查看原因并根据指示做相应的优化修改。
- Hidden Cost of Transparency 通常来说，对于不透明的 View，显示它只需要渲染一次即可，可是如果这个 View 设置了 alpha 值，会至少需要渲染两次。
- Avoiding Allocations in onDraw() 首先 onDraw() 方法是执行在 UI 线程的，在 UI 线程尽量避免做任何可能影响到性能的操作。虽然分配内存的操作并不需要花费太多系统资源，但是这并不意味着是免费无代价的。设备有一定的刷新频率，导致 View 的 onDraw 方法会被频繁的调用，如果 onDraw 方法效率低下，在频繁刷新累积的效应下，效率低的问题会被扩大，然后会对性能有严重的影响。

- Tool: Strict Mode Android 提供了一个叫做 Strict Mode 的工具，我们可以通过手机设置里面的开发者选项，打开 Strict Mode 选项，如果程序存在潜在的隐患，屏幕就会闪现红色。我们也可以通过 StrictMode API 在代码层面做细化的跟踪，可以设置 StrictMode 监听那些潜在问题，出现问题时如何提醒开发者，可以对屏幕闪红色，也可以输出错误日志。
- Custom Views and Performance Useless calls to onDraw()：我们知道调用 View.invalidate() 会触发 View 的重绘，有两个原则需要遵守，第 1 个是仅仅在 View 的内容发生改变的时候才去触发 invalidate 方法，第 2 个是尽量使用 ClipRect 等方法来提高绘制的性能。Useless pixels：减少绘制时不必要的绘制元素，对于那些不可见的元素，我们需要尽量避免重绘。Wasted CPU cycles：对于不在屏幕上的元素，可以使用 Canvas.quickReject 把他们给剔除，避免浪费 CPU 资源。另外尽量使用 GPU 来进行 UI 的渲染，这样能够极大的提高程序的整体表现性能。
- Batching Background Work Until Later 1.AlarmManager 使用 AlarmManager 设置定时任务，可以选择精确的间隔时间，也可以选择非精确时间作为参数。除非程序有很强烈的需要使用精确的定时唤醒，否者一定要避免使用他，我们应该尽量使用非精确的方式。2.SyncAdapter 我们可以使用 SyncAdapter 为应用添加设置账户，这样在手机设置的账户列表里面可以找到我们的应用。这种方式功能更多，但是实现起来比较复杂。我们可以从这里看到官方的培训课程:http://developer.android.com/training/sync-adapters/index.html 3.JobSchedulor 这是最简单高效的方法，我们可以设置任务延迟的间隔，执行条件，还可以增加重试机制。
- Smaller Pixel Formats Android 的 Heap 空间是不会自动做兼容压缩的，意思就是如果 Heap 空间中的图片被收回之后，这块区域并不会和其他已经回收过的区域做重新排序合并处理，那么当一个更大的图片需要放到 heap 之前，很可能找不到那么大的连续空闲区域，那么就会触发 GC，使得 heap 腾出一块足以放下这张图片的空闲区域，如果无法腾出，就会发生 OOM。
- Smaller PNG Files 尽量减少 PNG 图片的大小是 Android 里面很重要的一条规范。相比起 JPEG，PNG 能够提供更加清晰无损的图片，但是 PNG 格式的图片会更大，占用更多的磁盘空间。到底是使用 PNG 还是 JPEG，需要设计师仔细衡量，对于那些使用 JPEG 就可以达到视觉效果的，可以考虑采用 JPEG 即可。
- Pre-scaling Bitmaps 对 bitmap 做缩放，这也是 Android 里面最遇到的问题。对 bitmap 做缩放的意义很明显，提示显示性能，避免分配不必要的内存。Android 提供了现成的 bitmap 缩放的 API，叫做 createScaledBitmap()
- Re-using Bitmaps 使用 inBitmap 属性可以告知 Bitmap 解码器去尝试使用已经存在的内存区域，新解码的 bitmap 会尝试去使用之前那张 bitmap 在 heap 中所占据的 pixel data 内存区域，而不是去问内存重新申请一块区域来存放 bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小。
- The Performance Lifecycle Gather：收集数据，Insight：分析数据，Action：解决问题

3. Android 性能优化典范 - 第 3 季

- Fun with ArrayMaps 为了解决 HashMap 更占内存的弊端，Android 提供了内存效率更高的 ArrayMap。它内部使用两个数组进行工作，其中一个数组记录 key hash 过后的顺序列表，另外一个数组按 key 的顺序记录 Key-Value 值
- Beware Autoboxing 有时候性能问题也可能是因为那些不起眼的小细节引起的，例如在代码中不经意的“自动装箱”。我们知道基础数据类型的大小：boolean(8 bits), int(32 bits), float(32 bits)，long(64 bits)，为了能够让这些基础数据类型在大多数 Java 容器中运作，会需要做一个 autoboxing 的操作，转换成 Boolean，Integer，Float 等对象

- SparseArray Family Ties 为了避免 HashMap 的 autoboxing 行为，Android 系统提供了 SparseBoolMap，SparseIntMap，SparseLongMap，LongSparseMap 等容器。
- The price of ENUMs Android 官方强烈建议不要在 Android 程序里面使用到 enum。
- Trimming and Sharing Memory Android 系统提供了一些回调来通知应用的内存使用情况，通常来说，当所有的 background 应用都被 kill 掉的时候，forground 应用会收到 onLowMemory() 的回调。在这种情况下，需要尽快释放当前应用的非必须内存资源，从而确保系统能够稳定继续运行。Android 系统还提供了 onTrimMemory() 的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调
- DO NOT LEAK VIEWS 避免使用异步回调, 避免使用 Static 对象, 避免把 View 添加到没有清除机制的容器里面
- Location & Battery Drain 其中存在的一个优化点是，我们可以通过判断返回的位置信息是否相同，从而决定设置下次的更新间隔是否增加一倍，通过这种方式可以减少电量的消耗
- Double Layout Taxation 布局中的任何一个 View 一旦发生一些属性变化，都可能引起很大的连锁反应。例如某个 button 的大小突然增加一倍，有可能会导致兄弟视图的位置变化，也有可能导致父视图的大小发生改变。当大量的 layout() 操作被频繁调用执行的时候，就很可能引起丢帧的现象。
- Network Performance 101 减少移动网络被激活的时间与次数, 压缩传输数据
- Effective Network Batching 发起网络请求与接收返回数据都是比较耗电的，在网络硬件模块被激活之后，会继续保持几十秒的电量消耗，直到没有新的网络操作行为之后，才会进入休眠状态。前面一个段落介绍了使用 Batching 的技术来捆绑网络请求，从而达到减少网络请求的频率。那么如何实现 Batching 技术呢？通常来说，我们可以会把那些发出的网络请求，先暂存到一个 PendingQueue 里面，等到条件合适的时候再触发 Queue 里面的网络请求。
- Optimizing Network Request Frequencies 前面的段落已经提到了应该减少网络请求的频率，这是为了减少电量的消耗。我们可以使用 Batching，Prefetching 的技术来避免频繁的网络请求。Google 提供了 GCMNetworkManager 来帮助开发者实现那些功能，通过提供的 API，我们可以选择在接入 WiFi，开始充电，等待移动网络被激活等条件下再次激活网络请求。
- Effective Prefetching 类似上面的情况会频繁触发网络请求，但是如果我们能够预先请求后续可能会使用到网络资源，避免频繁的触发网络请求，这样就能够显著的减少电量的消耗。可是预先获取多少数据量是很值得考量的，因为如果预取数据量偏少，就起不到减少频繁请求的作用，可是如果预取数据过多，就会造成资源的浪费。

4. Android 性能优化典范 - 第 4 季

- Cachematters for networking 想要使得 Android 系统上的网络访问操作更加的高效就必须做好网络数据的缓存。这是提高网络访问性能最基础的步骤之一。从手机的缓存中直接读取数据肯定比从网络上获取数据要更加的便捷高效，特别是对于那些会被频繁访问到的数据，需要把这些数据缓存到设备上，以便更加快速的进行访问。
- Optimizing Network Request Frequencies 首先我们要对网络行为进行分类，区分需要立即更新数据的行为和其他可以进行延迟的更新行为，为不同的场景进行差异化处理。其次要避免客户端对服务器的轮询操作，这样会浪费很多的电量与带宽流量。解决这个问题，我们可以使用 Google Cloud Message 来对更新的数据进行推送。然后在某些必须做同步的场景下，需要避免使用固定的间隔频率来进行更新操作，我们应该在返回的数据无更新的时候，使用双倍的间隔时间来进行下一次同步。最后更进一步，我们还可以通过判断当前设备的状态来决定同步的频率，例如判断设备处于休眠，运动等不同的状态设计各自不同时间间隔的同步频率。

- Effective Prefetching 到底预取多少才比较合适呢？一个比较普适的规则是，在 3G 网络下可以预取 1-5Mb 的数据量，或者是按照提前预期后续 1-2 分钟的数据作为基线标准。在实际的操作当中，我们还需要考虑当前的网络速度来决定预取的数据量，例如在同样的时间下，4G 网络可以获取到 12 张图片的数据，而 2G 网络则只能拿到 3 张图片的数据。所以，我们还需要把当前的网络环境情况添加到设计预取数据量的策略当中去。判断当前设备的状态与网络情况，可以使用前面提到过的 GCMNetworkManager。
- Adapting to Latency 一个典型的网络操作行为，通常包含以下几个步骤：首先手机端发起网络请求，到达网络服务运营商的基站，再转移到服务提供者的服务器上，经过解码之后，接着访问本地的存储数据库，获取到数据之后，进行编码，最后按照原来传递的路径逐层返回。常来说，我们可以把网络请求延迟划分为三档：例如把网络延迟小于 60ms 的划分为 GOOD，大于 220ms 的划分为 BAD，介于两者之间的划分为 OK（这里的 60ms，220ms 会需要根据不同的场景提前进行预算推测）。
- Minimizing Asset Payload 为了能够减小网络传输的数据量，我们需要对传输的数据做压缩的处理，这样能够提高网络操作的性能。首先需要做的是减少图片的大小，其次需要做的是减少序列化数据的大小。
- Service Performance Patterns Service 是 Android 程序里面最常用的基础组件之一，但是使用 Service 很容易引起电量的过度消耗以及系统资源的未及时释放。避免错误的使用 Service，例如我们不应该使用 Service 来监听某些事件的变化，不应该搞一个 Service 在后台对服务器不断的进行轮询 (应该使用 Google Cloud Messaging)。如果已经事先知道 Service 里面的任务应该执行在后台线程 (非默认的主线程) 的时候，我们应该使用 IntentService 或者结合 HanderThread，AsycnTask Loader 实现的 Service。
- Removing unused code Android 为我们提供了 Proguard 的工具来帮助应用程序对代码进行瘦身，优化，混淆的处理。它会帮助移除那些没有使用到的代码，还可以对类名，方法名进行混淆处理以避免程序被反编译。
- Removing unused resources 所幸的是，我们可以使用 Gradle 来帮助我们分析代码，分析引用的资源，对于那些没有被引用到的资源，会在编译阶段被排除在 APK 安装包之外，要实现这个功能，对我们来说仅仅只需要在 build.gradle 文件中配置 shrinkResource 为 true 就好了
- Perf Theory: Caching 当我们讨论性能优化的时候，缓存是最常见最有效的策略之一。无论是为了提高 CPU 的计算速度还是提高数据的访问速度，在绝大多数的场景下，我们都会使用到缓存。
- Perf Theory: Approximation(近似法) 例如使用一张比较接近实际大小的图片来替代原图，换取更快的加载速度。所以对于那些对计算结果要求不需要十分精确的场景，我们可以使用近似法则来提高程序的性能。
- Perf Theory: Culling(遴选，挑选) 一个提高性能的方法是逐步对数据进行过滤筛选，减小搜索的数据集，以此提高程序的执行性能。例如我们需要搜索到居住在某个地方，年龄是多少，符合某些特定条件的候选人，就可以通过逐层过滤筛选的方式来提高后续搜索的执行效率。
- Perf Theory: Threading 使用多线程并发处理任务，从某种程度上可以快速提高程序的执行性能。对于 Android 程序来说，主线程通常也成为 UI 线程，需要处理 UI 的渲染，响应用户的操作等等。
- Perf Theory: Batching 网络请求的批量执行是另外一个比较适合说明 batching 使用场景的例子，因为每次发起网络请求都相对来说比较耗时耗电，如果能够做到批量一起执行，可以大大的减少电量的消耗。

- Serialization performance 数据序列化的行为可能发生在数据传递过程中的任何阶段，例如网络传输，不同进程间数据传递，不同类之间的参数传递，把数据存储到磁盘上等等。通常情况下，我们会把那些需要序列化的类实现 Serializable 接口 (如下图所示)，但是这种传统的做法效率不高，实施的过程会消耗更多的内存。但是我们如果使用 GSON 库来处理这个序列化的问题，不仅仅执行速度更快，内存的使用效率也更高。Android 的 XML 布局文件会在编译的阶段被转换成更加复杂的格式，具备更加高效的执行性能与更高的内存使用效率。
- Smaller Serialized Data 数据呈现的顺序以及结构会对序列化之后的空间产生不小的影响。
- Caching UI data 缓存 UI 界面上的数据，可以采用方案有存储到文件系统，Preference，SQLite 等等，做了缓存之后，这样就可以在请求数据返回结果之前，呈现给用户旧的数据，而不是使用正在加载的方式让用户什么数据都看不到，当然在请求网络最新数据的过程中，需要有正在刷新的提示。至于到底选择哪个方案来对数据进行缓存，就需要根据具体情况来做选择了。
- CPU Frequency Scaling 调节 CPU 的频率会执行的性能产生较大的影响，为了最大化的延长设备的续航时间，系统会动态调整 CPU 的频率，频率越高执行代码的速度自然就越快。我们可以使用 Systrace 工具来导出 CPU 的执行情况，以便帮助定位性能问题。

5. Android 性能优化典范 - 第 5 季

- Threading Performance AsyncTask: 为 UI 线程与工作线程之间进行快速的切换提供一种简单便捷的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的使用场景。HandlerThread: 为某些回调方法或者等待某些任务的执行设置一个专属的线程，并提供线程任务的调度机制。ThreadPool: 把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。IntentService: 适合于执行由 UI 触发的后台 Service 任务，并可以把后台任务执行的情况通过一定的机制反馈给 UI。
- Understanding Android Threading 通常来说，一个线程需要经历三个生命阶段：开始，执行，结束。线程会在任务执行完毕之后结束，那么为了确保线程的存活，我们会在执行阶段给线程赋予不同的任务，然后在里面添加退出的条件从而确保任务能够执行完毕后退出。
- Memory & Threading 不要在任何非 UI 线程里面去持有 UI 对象的引用。系统为了确保所有的 UI 对象都只会被 UI 线程所进行创建，更新，销毁的操作，特地设计了对应的工作机制 (当 Activity 被销毁的时候，由该 Activity 所触发的非 UI 线程都将无法对 UI 对象进行操作，否者就会抛出程序执行异常的错误) 来防止 UI 对象被错误的使用。
- Good AsyncTask Hunting AsyncTask 虽然提供了一种简单便捷的异步机制，但是我们还是很有必要特别关注到他的缺点，避免出现因为使用错误而导致的严重系统性能问题。
- Getting a HandlerThread HandlerThread 比较合适处理那些在工作线程执行，需要花费时间偏长的任务。我们只需要把任务发送给 HandlerThread，然后就只需要等待任务执行结束的时候通知返回到主线程就好了。另外很重要的一点是，一旦我们使用了 HandlerThread，需要特别注意给 HandlerThread 设置不同的线程优先级，CPU 会根据设置的不同线程优先级对所有的线程进行调度优化。
- Swimming in Threadpools 线程池适合用在把任务进行分解，并发进行执行的场景。通常来说，系统里面会针对不同的任务设置一个单独的守护线程用来专门处理这项任务。
- The Zen of IntentService 默认的 Service 是执行在主线程的，可是通常情况下，这很容易影响到程序的绘制性能 (抢占了主线程的资源)。除了前面介绍过的 AsyncTask 与 HandlerThread，我们还可以选择使用 IntentService 来实现异步操作。IntentService 继承自普通 Service 同时又在内部创建了一个 HandlerThread，在 onHandlerIntent() 的回调里面处理扔到 IntentService 的任务。所以 IntentService 就不仅仅具备了异步线程的特性，还同时保留了 Service 不受主页面生命周期影响的特点。

- Threading and Loaders 当启动工作线程的 Activity 被销毁的时候，我们应该做点什么呢? 为了方便的控制工作线程的启动与结束，Android 为我们引入了 Loader 来解决这个问题。我们知道 Activity 有可能因为用户的主动切换而频繁的被创建与销毁，也有可能是因为类似屏幕发生旋转等被动原因而销毁再重建。在 Activity 不停的创建与销毁的过程当中，很有可能因为工作线程持有 Activity 的 View 而导致内存泄漏 (因为工作线程很可能持有 View 的强引用，另外工作线程的生命周期还无法保证和 Activity 的生命周期一致，这样就容易发生内存泄漏了)。除了可能引起内存泄漏之外，在 Activity 被销毁之后，工作线程还继续更新视图是没有意义的，因为此时视图已经不在界面上显示了。
- The Importance of Thread Priority 在 Android 系统里面,我们可以通过 android.os.Process.setThread 设置线程的优先级，参数范围从-20 到 24，数值越小优先级越高。Android 系统还为我们提供了以下的一些预设值，我们可以通过给不同的工作线程设置不同数值的优先级来达到更细粒度的控制。
- Profile GPU Rendering : M Update 从 Android M 系统开始，系统更新了 GPU Profiling 的工具来帮助我们定位 UI 的渲染性能问题。早期的 CPU Profiling 工具只能粗略的显示出 Process，Execute，Update 三大步骤的时间耗费情况。
- 官方性能优化系列教程 https://www.youtube.com/playlist?list=PLWz5rJ2EKKc9CBxr3BVjPTPoD

1.3.2 架构分析

- MVVM https://tech.meituan.com/android_mvvm.html
- MVP https://code.tutsplus.com/series/how-to-adopt-model-view-presenter-on-android--cms

1.3.3 阿里面试题

- 进程间通信方式
  - 通过 Intent 在 Activity、Service 或 BroadcastReceiver 间进行进程间通信，可通过 Intent 传递数据
  - AIDL 方式
  - Messenger 方式
  - 利用 ContentProvider
  - Socket 方式
  - 基于文件共享的方式
- 什么是协程
  - 我们知道多个线程相对独立，有自己的上下文，切换受系统控制；而协程也相对独立，有自己的上下文，但是其切换由自己控制，由当前协程切换到其他协程由当前协程来控制。
- 内存泄露是怎么回事
  - 由忘记释放分配的内存导致的
- 程序计数器，引到了逻辑地址（虚地址）和物理地址及其映射关系

  - 虚拟机中的程序计数器是 Java 运行时数据区中的一小块内存区域，但是它的功能和通常的程序计数器是类似的，它指向虚拟机正在执行字节码指令的地址。具体点儿说，当虚拟机执行的方法不是 native 的时，程序计数器指向虚拟机正在执行字节码指令的地址；当虚拟机执行的方法是 native 的时，程序计数器中的值是未定义的。另外，程序计数器是线程私有的，也就是说，每一个线程都拥有仅属于自己的程序计数器。

- 数组和链表的区别

  - 数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。
  - 链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。

- 二叉树的深度优先遍历和广度优先遍历的具体实现

http://www.i3geek.com/archives/794

- 堆的结构

  - 年轻代（Young Generation）、年老代（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

- bitmap 对象的理解

  - http://blog.csdn.net/angel1hao/article/details/51890938

- 什么是深拷贝和浅拷

  - 浅拷贝：使用一个已知实例对新创建实例的成员变量逐个赋值，这个方式被称为浅拷贝。
  - 深拷贝：当一个类的拷贝构造方法，不仅要复制对象的所有非引用成员变量值，还要为引用类型的成员变量创建新的实例，并且初始化为形式参数实例值。这个方式称为深拷贝

- 对象锁和类锁是否会互相影响

  - 对象锁：Java 的所有对象都含有 1 个互斥锁，这个锁由 JVM 自动获取和释放。线程进入 synchronized 方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待；synchronized 方法正常返回或者抛异常而终止，JVM 会自动释放对象锁。这里也体现了用 synchronized 来加锁的 1 个好处，方法抛异常的时候，锁仍然可以由 JVM 来自动释放。
  - 类锁：对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步。其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的。我们都知道，java 类可能会有很多个对象，但是只有 1 个 Class 对象，也就是说类的不同实例之间共享该类的 Class 对象。Class 对象其实也仅仅是 1 个 java 对象，只不过有点特殊而已。由于每个 java 对象都有 1 个互斥锁，

而类的静态方法是需要 Class 对象。所以所谓的类锁，不过是 Class 对象的锁而已。获取类的 Class 对象有好几种，最简单的就是 MyClass.class 的方式。类锁和对象锁不是同 1 个东西，一个是类的 Class 对象的锁，一个是类的实例的锁。也就是说：1 个线程访问静态 synchronized 的时候，允许另一个线程访问对象的实例 synchronized 方法。反过来也是成立的，因为他们需要的锁是不同的。

- looper 架构
  - http://wangkuiwu.github.io/2014/08/26/MessageQueue/
- 自定义控件原理
  - http://www.jianshu.com/p/988326f9c8a3
- binder 工作原理
  - Binder 是客户端和服务端进行通讯的媒介
- ActivityThread，Ams，Wms 的工作原理
  - ActivityThread: 运行在应用进程的主线程上，响应 ActivityManangerService 启动、暂停 Activity，广播接收等消息。ams: 统一调度各应用程序的 Activity、内存管理、进程管理
- Java 中 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. JVM 不保证此方法总被调用.
- 一个文件中有 100 万个整数，由空格分开，在程序中判断用户输入的整数是否在此文件中。说出最优的方法
- 两个进程同时要求写或者读，能不能实现？如何防止进程的同步？
- volatile 的意义？
- 防止 CPU 指令重排序
- 单例

```
1  public class Singleton {
2      private volatile static Singleton mSingleton;
3
4      private Singleton(){
5      }
6
7      public static Singleton getInstance() {
8          if (mSingleton == null) {//A
9              synchronized(Singleton.class) {//C
10                 if(mSingleton == null)
11                     mSingleton = new Singleton();//B
```

```
            }
        }
        return mSingleton;
    }
}
```

- Given a string, determine if it is a palindrome(回文,如果不清楚,按字面意思脑补下), considering only alphanumeric characters and ignoring cases.
  - For example, "A man, a plan, a canal: Panama" is a palindrome. "race a car" is not a palindrome.
  - Note: Have you consider that the string might be empty? This is a good question to ask during an interview. For the purpose of this problem, we define empty string as valid palindrome.

```
public boolean isPalindrome(String palindrome) {
    char[] palindromes = palidrome.toCharArray();
    if (palindromes.lengh == 0) {
        return true
    }
    Arraylist<Char> temp = new Arraylist();
    for (int i=0;i<palindromes.length;i++) {
 ^^I^^Iif ((palindromes[i]>'a' && palindromes[i]<'z')||palindromes[i]>'A' && palindr
            temp.add(palindromes[i].toLowerCase());
    }

    for (int i=0;i<temp.size()/2;i++) {
        if (temp.get(i) != temp.get(temp.size()-i)) {
            //
            return false;
        }
    }
    return true;
}
```

- 烧一根不均匀的绳，从头烧到尾总共需要 1 个小时。现在有若干条材质相同的绳子，问如何用烧绳的方法来计时一个小时十五分钟呢
  - 用两根绳子，一个绳子两头烧，一个一头烧。

1.3.4 腾讯

- 2000 万个整数，找出第五十大的数字?
  - 冒泡、选择、建堆
- 从网络加载一个 10M 的图片，说下注意事项
  - 图片缓存、异常恢复、质量压缩
- 自定义 View 注意事项

  - 渲染帧率、内存

- 项目中常用的设计模式
  - 单例、观察者、适配器、建造者。。
- JVM 的理解 http://www.infoq.com/cn/articles/java-memory-model-1