以下面试题来自于百度、小米、乐视、美团、58、猎豹、360、新浪、搜狐内部题库

熟悉本文中列出的知识点会大大增加通过前两轮技术面试的几率。

**一、java基础**

**1.接口的意义**（百度）

规范、扩展、回调

**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

**3.内部类的作用**(百度，乐视)

1. 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。
2. 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。
3. 创建内部类对象的时刻并不依赖于外围类对象的创建。
4. 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。
5. 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

**4.父类的静态方法能否被子类重写，为什么？**（猎豹）

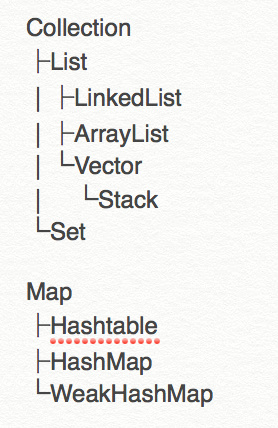
不能

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

**5.举1-2个排序算法，并使用java代码实现**（美团）

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

**6.列举java的集合和继承关系**（百度、美团）



**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。

**8.哪些情况下的对象会被垃圾回收机制处理掉**（乐视、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。

**9.进程和线程的区别**（猎豹）

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。

**10.Java中==和equals的区别，equals和hashCode的区别**（乐视）

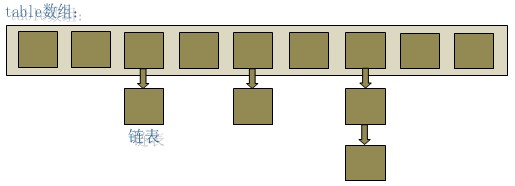
http://blog.csdn.net/tiantiandjava/article/details/46988461

**11.常见的排序算法时间复杂度**（小米）



**12.HashMap的实现原理**（美团）

1. 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。
2. 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。

**13.java 状态机**

http://www.jdon.com/designpatterns/designpattern\_State.htm

**14.java中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

**15.java int与integer的区别**

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

**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）

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 StringBuilder(“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

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

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

**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。

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。

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。

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。

**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。

1. 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。
2. 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。
3. 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.
4. 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。

**19.抽象类接口区别**（360）

1. 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
2. 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
3. 构造器  
   抽象类可以有构造器  
   接口不能有构造器
4. 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
5. 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。
6. 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。
7. 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。
8. 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。
9. 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。

**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

**21.java 内部类**（小米）

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

**22.Java中hashmap和hashtable的区别**（乐视、小米）

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

**23.ArrayMap VS HashMap**

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

**二、android基础**

**1.数据库的操作类型有哪些，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.

**2.是否使用过本地广播，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。

**3.是否使用过intentServer，作用是什么，AIDL解决了什么问题？**(小米)

<code class="language-none">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。</code>

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。

**4.Activity、Window、View三者的差别，fragment的特点？**（360）

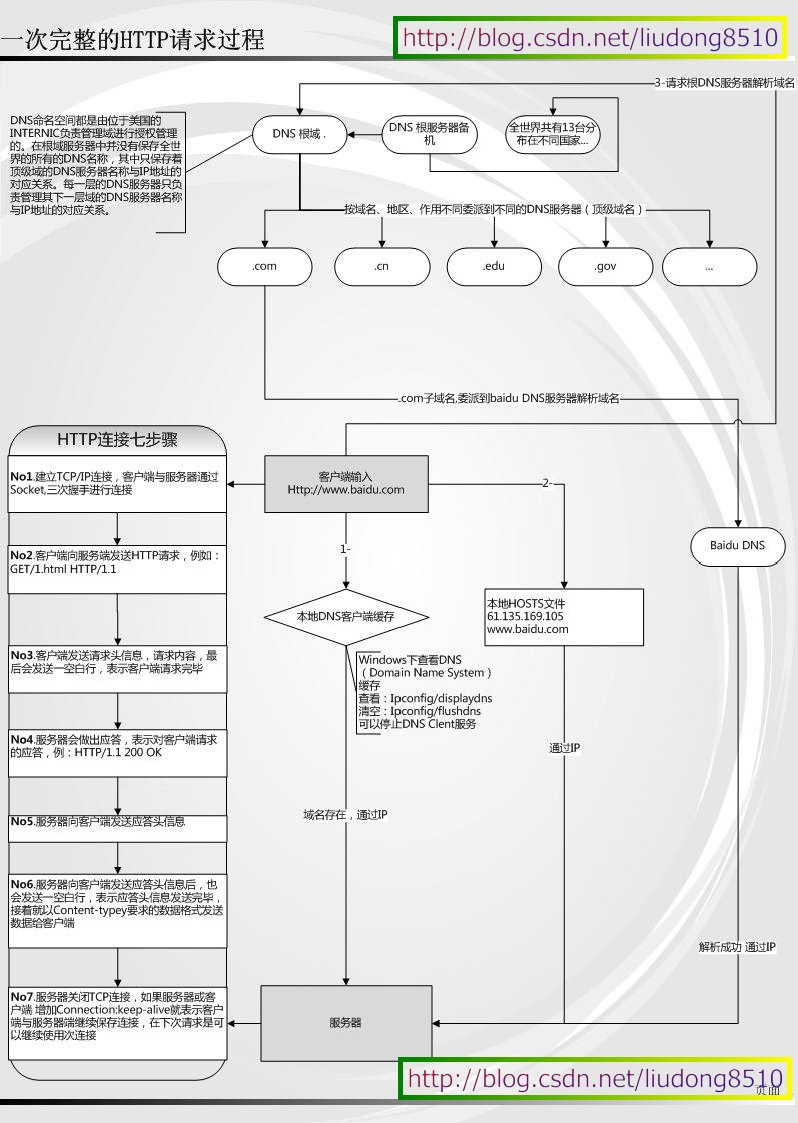
Activity像一个工匠（控制单元），Window像窗户（承载模型），View像窗花（显示视图） LayoutInflater像剪刀，Xml配置像窗花图纸。

1. 在Activity中调用attach，创建了一个Window
2. 创建的window是其子类PhoneWindow，在attach中创建PhoneWindow
3. 在Activity中调用setContentView(R.layout.xxx)
4. 其中实际上是调用的getWindow().setContentView()
5. 调用PhoneWindow中的setContentView方法
6. 创建ParentView： 作为ViewGroup的子类，实际是创建的DecorView(作为FramLayout的子类）
7. 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】
8. 调用到ViewGroup
9. 调用ViewGroup的removeAllView()，先将所有的view移除掉
10. 添加新的view：addView()

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。

**5.描述一次网络请求的流程**（新浪）

￼

**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。

**7.低版本SDK实现高版本api**（小米）

自己实现或@TargetApi annotation

**8.编译安卓系统**（百度）

1.创建一个大小写敏感的磁盘镜像文件 2.首先你要安装的就是Xcode 3.从macports.org安装MacPorts工具 4.从MacPorts安装make,git, and GPG packages 5.修改文件标识符限制 6.创建~/bin/目录 7.安装repo 8.设置repo可执行权限 9.修改~/bin/repo文件将第五行 10.新建目录gingerbread，并进入该目录 11.执行repoinit 12. reposync 13.设置必要的编译环境-编译 14.利用Eclipse调试代码

**9.launch mode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: 1.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。 2.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。 3.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。

**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/

**11.view绘制流程**（百度）

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

**12.多线程**（360）

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

**13.线程同步**（百度）

http://www.itzhai.com/java-based-notebook-thread-synchronization-problem-solving-synchronization-problems-synchronized-block-synchronized-methods.html#read-more

http://www.juwends.com/tech/android/android-inter-thread-comm.html

单例

<code class="language-none">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;

}

}</code>

**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：

<code class="language-none">public View getView(int position, ViewconvertView, ViewGroup parent) {

View view = new Xxx(...);

... ...

return view;

} </code>

修正示例代码：

<code class="language-none">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;

} </code>

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。

**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*BACKGROUND)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。

**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

**17.Android 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

**18.Android各个版本API的区别**

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

**19. Android代码中实现WAP方式联网**（360）

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

**20.如何保证service在后台不被kill**

一、onStartCommand方法，返回START\_STICKY

1、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。

2、START*NOT*STICKY 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。

3、START*REDELIVER*INTENT 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: 1.前台进程( FOREGROUND*APP)  2.可视进程(VISIBLE*APP )  3. 次要服务进程(SECONDARY*SERVER )  4.后台进程 (HIDDEN*APP)  5.内容供应节点(CONTENT*PROVIDER)  6.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。

**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 。 说明：只是对View树重新布局layout过程包括measure()和layout()过程，不会调用draw()过程，但不会重新绘制 任何视图包括该调用者本身。

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

**22.invalidate()和postInvalidate() 的区别及使用**（百度）

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

**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。

**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

**25.Android 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：

1. 绘制该View的背景
2. 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)
3. 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)
4. 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。

**26.LinearLayout对比RelativeLayout**（百度）

1.RelativeLayout会让子View调用2次onMeasure，LinearLayout 在有weight时，也会调用子View2次onMeasure 2.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。 3.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。

**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。

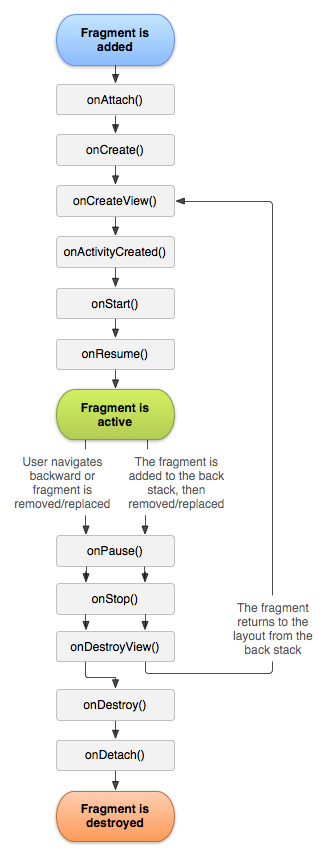
另外一个非常耗时的操作是请求layout。任何时候执行requestLayout()，会使得Android UI系统去遍历整个View的层级来计算出每一个view的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持View的层级是扁平化的，这样对提高效率很有帮助。

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。

**28.ContentProvider**（乐视）

http://blog.csdn.net/coder\_pig/article/details/47858489

**29.fragment生命周期**



**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

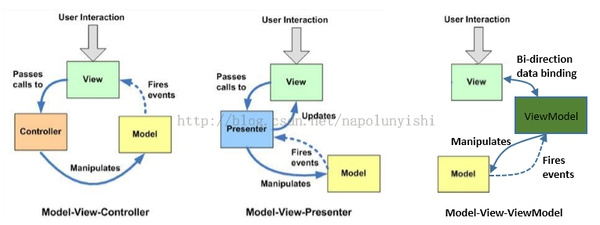
**31.Android Glide源码解析**

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

**32.Android 设计模式**

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

**33.架构设计**（搜狐）



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

**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。

注册掘金，阅读更多优质Android技术文章。