# android

### 设计模式与使用场景

a.建造者模式：  
将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。  
使用场景比如最常见的AlertDialog,拿我们开发过程中举例，比如Camera开发过程中，可能需要设置一个初始化的相机配置，设置摄像头方向，闪光灯开闭，成像质量等等，这种场景下就可以使用建造者模式

装饰者模式：动态的给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。装饰者模式可以在不改变原有类结构的情况下曾强类的功能，比如Java中的BufferedInputStream 包装FileInputStream，举个开发中的例子，比如在我们现有网络框架上需要增加新的功能，那么再包装一层即可，装饰者模式解决了继承存在的一些问题，比如多层继承代码的臃肿，使代码逻辑更清晰  
观察者模式：  
代理模式：  
门面模式：  
单例模式：  
生产者消费者模式：

### java语言的特点与OOP思想

这个通过对比来描述，比如面向对象和面向过程的对比，针对这两种思想的对比，还可以举个开发中的例子，比如播放器的实现，面向过程的实现方式就是将播放视频的这个功能分解成多个过程，比如，加载视频地址，获取视频信息，初始化解码器，选择合适的解码器进行解码，读取解码后的帧进行视频格式转换和音频重采样，然后读取帧进行播放，这是一个完整的过程，这个过程中不涉及类的概念，而面向对象最大的特点就是类，封装继承和多态是核心，同样的以播放器为例，一面向对象的方式来实现，将会针对每一个功能封装出一个对象，吧如说Muxer，获取视频信息，Decoder,解码，格式转换器，视频播放器，音频播放器等，每一个功能对应一个对象，由这个对象来完成对应的功能，并且遵循单一职责原则，一个对象只做它相关的事情

说下java中的线程创建方式，线程池的工作原理。  
java中有三种创建线程的方式，或者说四种  
1.继承Thread类实现多线程  
2.实现Runnable接口  
3.实现Callable接口  
4.通过线程池

线程池的工作原理：线程池可以减少创建和销毁线程的次数，从而减少系统资源的消耗，当一个任务提交到线程池时  
a. 首先判断核心线程池中的线程是否已经满了，如果没满，则创建一个核心线程执行任务，否则进入下一步  
b. 判断工作队列是否已满，没有满则加入工作队列，否则执行下一步  
c. 判断线程数是否达到了最大值，如果不是，则创建非核心线程执行任务，否则执行饱和策略，默认抛出异常

### 说下 handler 原理

Handler，Message，looper 和 MessageQueue 构成了安卓的消息机制，handler创建后可以通过 sendMessage 将消息加入消息队列，然后 looper不断的将消息从 MessageQueue 中取出来，回调到 Hander 的 handleMessage方法，从而实现线程的通信。

从两种情况来说，第一在UI线程创建Handler,此时我们不需要手动开启looper，因为在应用启动时，在ActivityThread的main方法中就创建了一个当前主线程的looper，并开启了消息队列，消息队列是一个无限循环，为什么无限循环不会ANR?因为可以说，应用的整个生命周期就是运行在这个消息循环中的，安卓是由事件驱动的，Looper.loop不断的接收处理事件，每一个点击触摸或者Activity每一个生命周期都是在Looper.loop的控制之下的，looper.loop一旦结束，应用程序的生命周期也就结束了。我们可以想想什么情况下会发生ANR，第一，事件没有得到处理，第二，事件正在处理，但是没有及时完成，而对事件进行处理的就是looper，所以只能说事件的处理如果阻塞会导致ANR，而不能说looper的无限循环会ANR

另一种情况就是在子线程创建Handler,此时由于这个线程中没有默认开启的消息队列，所以我们需要手动调用looper.prepare(),并通过looper.loop开启消息

主线程Looper从消息队列读取消息，当读完所有消息时，主线程阻塞。子线程往消息队列发送消息，并且往管道文件写数据，主线程即被唤醒，从管道文件读取数据，主线程被唤醒只是为了读取消息，当消息读取完毕，再次睡眠。因此loop的循环并不会对CPU性能有过多的消耗。

### 内存泄漏的场景和解决办法

1.非静态内部类的静态实例  
非静态内部类会持有外部类的引用，如果非静态内部类的实例是静态的，就会长期的维持着外部类的引用，组织被系统回收，解决办法是使用静态内部类

2.多线程相关的匿名内部类和非静态内部类  
匿名内部类同样会持有外部类的引用，如果在线程中执行耗时操作就有可能发生内存泄漏，导致外部类无法被回收，直到耗时任务结束，解决办法是在页面退出时结束线程中的任务

3.Handler内存泄漏  
Handler导致的内存泄漏也可以被归纳为非静态内部类导致的，Handler内部message是被存储在MessageQueue中的，有些message不能马上被处理，存在的时间会很长，导致handler无法被回收，如果handler是非静态的，就会导致它的外部类无法被回收，解决办法是1.使用静态handler，外部类引用使用弱引用处理2.在退出页面时移除消息队列中的消息

4.Context导致内存泄漏  
根据场景确定使用Activity的Context还是Application的Context,因为二者生命周期不同，对于不必须使用Activity的Context的场景（Dialog）,一律采用Application的Context,单例模式是最常见的发生此泄漏的场景，比如传入一个Activity的Context被静态类引用，导致无法回收

5.静态View导致泄漏  
使用静态View可以避免每次启动Activity都去读取并渲染View，但是静态View会持有Activity的引用，导致无法回收，解决办法是在Activity销毁的时候将静态View设置为null（View一旦被加载到界面中将会持有一个Context对象的引用，在这个例子中，这个context对象是我们的Activity，声明一个静态变量引用这个View，也就引用了activity）

6.WebView导致的内存泄漏  
WebView只要使用一次，内存就不会被释放，所以WebView都存在内存泄漏的问题，通常的解决办法是为WebView单开一个进程，使用AIDL进行通信，根据业务需求在合适的时机释放掉

7.资源对象未关闭导致  
如Cursor，File等，内部往往都使用了缓冲，会造成内存泄漏，一定要确保关闭它并将引用置为null

8.集合中的对象未清理  
集合用于保存对象，如果集合越来越大，不进行合理的清理，尤其是入股集合是静态的

9.Bitmap导致内存泄漏  
bitmap是比较占内存的，所以一定要在不使用的时候及时进行清理，避免静态变量持有大的bitmap对象

10.监听器未关闭  
很多需要register和unregister的系统服务要在合适的时候进行unregister,手动添加的listener也需要及时移除

### 如何避免OOM?

1.使用更加轻量的数据结构：如使用ArrayMap/SparseArray替代HashMap,HashMap更耗内存，因为它需要额外的实例对象来记录Mapping操作，SparseArray更加高效，因为它避免了Key Value的自动装箱，和装箱后的解箱操作

2.便面枚举的使用，可以用静态常量或者注解@IntDef替代

3.Bitmap优化:  
a.尺寸压缩：通过InSampleSize设置合适的缩放  
b.颜色质量：设置合适的format，ARGB\_6666/RBG\_545/ARGB\_4444/ALPHA\_6，存在很大差异  
c.inBitmap: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，但复用存在一些限制，具体体现在：在Android 4.4之前只能重用相同大小的Bitmap的内存，而Android 4.4及以后版本则只要后来的Bitmap比之前的小即可。使用inBitmap参数前，每创建一个Bitmap对象都会分配一块内存供其使用，而使用了inBitmap参数后，多个Bitmap可以复用一块内存，这样可以提高性能

4.StringBuilder替代String: 在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”

5.避免在类似onDraw这样的方法中创建对象，因为它会迅速占用大量内存，引起频繁的GC甚至内存抖动

6.减少内存泄漏也是一种避免OOM的方法

说下 Activity 的启动模式，生命周期，两个 Activity 跳转的生命周期，如果一个 Activity 跳转另一个 Activity 再按下 Home 键在回到 Activity 的生命周期是什么样的

### 启动模式

Standard 模式:Activity 可以有多个实例，每次启动 Activity，无论任务栈中是否已经有这个Activity的实例，系统都会创建一个新的Activity实例

SingleTop模式:当一个singleTop模式的Activity已经位于任务栈的栈顶，再去启动它时，不会再创建新的实例,如果不位于栈顶，就会创建新的实例

SingleTask模式:如果Activity已经位于栈顶，系统不会创建新的Activity实例，和singleTop模式一样。但Activity已经存在但不位于栈顶时，系统就会把该Activity移到栈顶，并把它上面的activity出栈

SingleInstance模式:singleInstance 模式也是单例的，但和singleTask不同，singleTask 只是任务栈内单例，系统里是可以有多个singleTask Activity实例的，而 singleInstance Activity 在整个系统里只有一个实例，启动一singleInstanceActivity 时，系统会创建一个新的任务栈，并且这个任务栈只有他一个Activity

生命周期  
onCreate onStart onResume onPause onStop onDestroy

两个 Activity 跳转的生命周期  
1.启动A  
onCreate - onStart - onResume

2.在A中启动B  
ActivityA onPause  
ActivityB onCreate  
ActivityB onStart  
ActivityB onResume  
ActivityA onStop

3.从B中返回A（按物理硬件返回键）  
ActivityB onPause  
ActivityA onRestart  
ActivityA onStart  
ActivityA onResume  
ActivityB onStop  
ActivityB onDestroy

4.继续返回  
ActivityA onPause  
ActivityA onStop  
ActivityA onDestroy

### onRestart 的调用场景

（1）按下home键之后，然后切换回来，会调用onRestart()。  
（2）从本Activity跳转到另一个Activity之后，按back键返回原来Activity，会调用onRestart()；  
（3）从本Activity切换到其他的应用，然后再从其他应用切换回来，会调用onRestart()；

说下 Activity 的横竖屏的切换的生命周期，用那个方法来保存数据，两者的区别。触发在什么时候在那个方法里可以获取数据等。

是否了 SurfaceView，它是什么？他的继承方式是什么？他与View的区别(从源码角度，如加载，绘制等)。

SurfaceView中采用了双缓冲机制，保证了UI界面的流畅性，同时 SurfaceView 不在主线程中绘制，而是另开辟一个线程去绘制，所以它不妨碍UI线程；

SurfaceView 继承于View，他和View主要有以下三点区别：  
（1）View底层没有双缓冲机制，SurfaceView有；  
（2）view主要适用于主动更新，而SurfaceView适用与被动的更新，如频繁的刷新  
（3）view会在主线程中去更新UI，而SurfaceView则在子线程中刷新；  
SurfaceView的内容不在应用窗口上，所以不能使用变换（平移、缩放、旋转等）。也难以放在ListView或者ScrollView中，不能使用UI控件的一些特性比如View.setAlpha()

View：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在UI主线程内更新画面，速度较慢。  
SurfaceView：基于view视图进行拓展的视图类，更适合2D游戏的开发；是view的子类，类似使用双缓机制，在新的线程中更新画面所以刷新界面速度比view快，Camera预览界面使用SurfaceView。  
GLSurfaceView：基于SurfaceView视图再次进行拓展的视图类，专用于3D游戏开发的视图；是SurfaceView的子类，openGL专用。

### 如何实现进程保活

a: Service 设置成 START\_STICKY kill 后会被重启(等待5秒左右)，重传Intent，保持与重启前一样  
b: 通过 startForeground将进程设置为前台进程， 做前台服务，优先级和前台应用一个级别，除非在系统内存非常缺，否则此进程不会被 kill  
c: 双进程Service： 让2个进程互相保护对方，其中一个Service被清理后，另外没被清理的进程可以立即重启进程  
d: 用C编写守护进程(即子进程) : Android系统中当前进程(Process)fork出来的子进程，被系统认为是两个不同的进程。当父进程被杀死的时候，子进程仍然可以存活，并不受影响(Android5.0以上的版本不可行）联系厂商，加入白名单  
e.锁屏状态下，开启一个一像素Activity

### 说下冷启动与热启动是什么，区别，如何优化，使用场景等。

app冷启动： 当应用启动时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用， 这个启动方式就叫做冷启动（后台不存在该应用进程）。冷启动因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化Application类，再创建和初始化MainActivity类（包括一系列的测量、布局、绘制），最后显示在界面上。

app热启动： 当应用已经被打开， 但是被按下返回键、Home键等按键时回到桌面或者是其他程序的时候，再重新打开该app时， 这个方式叫做热启动（后台已经存在该应用进程）。热启动因为会从已有的进程中来启动，所以热启动就不会走Application这步了，而是直接走MainActivity（包括一系列的测量、布局、绘制），所以热启动的过程只需要创建和初始化一个MainActivity就行了，而不必创建和初始化Application

冷启动的流程  
当点击app的启动图标时，安卓系统会从Zygote进程中fork创建出一个新的进程分配给该应用，之后会依次创建和初始化Application类、创建MainActivity类、加载主题样式Theme中的windowBackground等属性设置给MainActivity以及配置Activity层级上的一些属性、再inflate布局、当onCreate/onStart/onResume方法都走完了后最后才进行contentView的measure/layout/draw显示在界面上

冷启动的生命周期简要流程：：  
Application构造方法 –> attachBaseContext()–>onCreate –>Activity构造方法 –> onCreate() –> 配置主体中的背景等操作 –>onStart() –> onResume() –> 测量、布局、绘制显示

冷启动的优化主要是视觉上的优化，解决白屏问题，提高用户体验，所以通过上面冷启动的过程。能做的优化如下：

减少 onCreate()方法的工作量

不要让 Application 参与业务的操作

不要在 Application 进行耗时操作

不要以静态变量的方式在 Application 保存数据

减少布局的复杂度和层级

减少主线程耗时

为什么冷启动会有白屏黑屏问题？原因在于加载主题样式Theme中的windowBackground等属性设置给MainActivity发生在inflate布局当onCreate/onStart/onResume方法之前，而windowBackground背景被设置成了白色或者黑色，所以我们进入app的第一个界面的时候会造成先白屏或黑屏一下再进入界面。解决思路如下

1.给他设置 windowBackground 背景跟启动页的背景相同，如果你的启动页是张图片那么可以直接给 windowBackground 这个属性设置该图片那么就不会有一闪的效果了

<style name=``"Splash\_Theme"` `parent=``"@android:style/Theme.NoTitleBar"``>`

<item name=``"android:windowBackground"``>@drawable/splash\_bg</item>`

<item name=``"android:windowNoTitle"``>``true``</item>`

</style>`

2.采用世面的处理方法，设置背景是透明的，给人一种延迟启动的感觉。,将背景颜色设置为透明色,这样当用户点击桌面APP图片的时候，并不会"立即"进入APP，而且在桌面上停留一会，其实这时候APP已经是启动的了，只是我们心机的把Theme里的windowBackground 的颜色设置成透明的，强行把锅甩给了手机应用厂商（手机反应太慢了啦）

<style name=``"Splash\_Theme"` `parent=``"@android:style/Theme.NoTitleBar"``>`

<item name=``"android:windowIsTranslucent"``>``true``</item>`

<item name=``"android:windowNoTitle"``>``true``</item>`

</style>`

3.以上两种方法是在视觉上显得更快，但其实只是一种表象，让应用启动的更快，有一种思路，将 Application 中的不必要的初始化动作实现懒加载，比如，在SpashActivity 显示后再发送消息到 Application，去初始化，这样可以将初始化的动作放在后边，缩短应用启动到用户看到界面的时间

### Android 中的线程有那些,原理与各自特点

AsyncTask,HandlerThread,IntentService

AsyncTask原理：内部是Handler和两个线程池实现的，Handler用于将线程切换到主线程，两个线程池一个用于任务的排队，一个用于执行任务，当AsyncTask执行execute方法时会封装出一个FutureTask对象，将这个对象加入队列中，如果此时没有正在执行的任务，就执行它，执行完成之后继续执行队列中下一个任务，执行完成通过Handler将事件发送到主线程。AsyncTask必须在主线程初始化，因为内部的Handler是一个静态对象，在AsyncTask类加载的时候他就已经被初始化了。在Android3.0开始，execute方法串行执行任务的，一个一个来，3.0之前是并行执行的。如果要在3.0上执行并行任务，可以调用executeOnExecutor方法

HandlerThread原理：继承自 Thread，start开启线程后，会在其run方法中会通过Looper 创建消息队列并开启消息循环，这个消息队列运行在子线程中，所以可以将HandlerThread 中的 Looper 实例传递给一个 Handler，从而保证这个 Handler 的 handleMessage 方法运行在子线程中，Android 中使用 HandlerThread的一个场景就是 IntentService

IntentService原理：继承自Service，它的内部封装了 HandlerThread 和Handler，可以执行耗时任务，同时因为它是一个服务，优先级比普通线程高很多，所以更适合执行一些高优先级的后台任务，HandlerThread底层通过Looper消息队列实现的，所以它是顺序的执行每一个任务。可以通过Intent的方式开启IntentService，IntentService通过handler将每一个intent加入HandlerThread子线程中的消息队列，通过looper按顺序一个个的取出并执行，执行完成后自动结束自己，不需要开发者手动关闭

### ANR的原因

1.耗时的网络访问  
2.大量的数据读写  
3.数据库操作  
4.硬件操作（比如camera)  
5.调用thread的join()方法、sleep()方法、wait()方法或者等待线程锁的时候  
6.service binder的数量达到上限  
7.system server中发生WatchDog ANR  
8.service忙导致超时无响应  
9.其他线程持有锁，导致主线程等待超时  
10.其它线程终止或崩溃导致主线程一直等待

### 三级缓存原理

当 Android 端需要获得数据时比如获取网络中的图片，首先从内存中查找（按键查找），内存中没有的再从磁盘文件或sqlite中去查找，若磁盘中也没有才通过网络获取

### LruCache 底层实现原理：

LruCache 中 Lru 算法的实现就是通过 LinkedHashMap 来实现的。LinkedHashMap 继承于 HashMap，它使用了一个双向链表来存储 Map中的Entry顺序关系，  
对于get、put、remove等操作，LinkedHashMap除了要做HashMap做的事情，还做些调整Entry顺序链表的工作。  
LruCache中将LinkedHashMap的顺序设置为LRU顺序来实现LRU缓存，每次调用get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。  
调用put插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

### 说下你对 Collection 这个类的理解。

Collection是集合框架的顶层接口，是存储对象的容器,Colloction定义了接口的公用方法如add remove clear等等，它的子接口有两个，List和Set,List的特点有元素有序，元素可以重复，元素都有索引（角标），典型的有  
Vector:内部是数组数据结构，是同步的（线程安全的）。增删查询都很慢。  
ArrayList:内部是数组数据结构，是不同步的（线程不安全的）。替代了Vector。查询速度快，增删比较慢。  
LinkedList:内部是链表数据结构，是不同步的（线程不安全的）。增删元素速度快。

而Set的是特点元素无序，元素不可以重复  
HashSet：内部数据结构是哈希表，是不同步的。  
Set集合中元素都必须是唯一的，HashSet作为其子类也需保证元素的唯一性。  
判断元素唯一性的方式：  
通过存储对象（元素）的hashCode和equals方法来完成对象唯一性的。  
如果对象的hashCode值不同，那么不用调用equals方法就会将对象直接存储到集合中；  
如果对象的hashCode值相同，那么需调用equals方法判断返回值是否为true，  
若为false, 则视为不同元素，就会直接存储；  
若为true， 则视为相同元素，不会存储。  
如果要使用HashSet集合存储元素，该元素的类必须覆盖hashCode方法和equals方法。一般情况下，如果定义的类会产生很多对象，通常都需要覆盖equals，hashCode方法。建立对象判断是否相同的依据。

TreeSet：保证元素唯一性的同时可以对内部元素进行排序，是不同步的。  
判断元素唯一性的方式：  
根据比较方法的返回结果是否为0，如果为0视为相同元素，不存；如果非0视为不同元素，则存。  
TreeSet对元素的排序有两种方式：  
方式一：使元素（对象）对应的类实现Comparable接口，覆盖compareTo方法。这样元素自身具有比较功能。  
方式二：使TreeSet集合自身具有比较功能，定义一个比较器Comparator，将该类对象作为参数传递给TreeSet集合的构造函数

### 说下AIDL的使用与原理

aidl是安卓中的一种进程间通信方式

说下你对广播的理解  
说下你对服务的理解，如何杀死一个服务。服务的生命周期(start与bind)。  
是否接触过蓝牙等开发  
设计一个ListView左右分页排版的功能自定义View，说出主要的方法。  
-说下binder序列化与反序列化的过程，与使用过程  
是否接触过JNI/NDK，java如何调用C语言的方法  
-如何查看模拟器中的SP与SQList文件。如何可视化查看布局嵌套层数与加载时间。  
你说用的代码管理工具什么，为什么会产生代码冲突，该如何解决  
说下你对后台的编程有那些认识，聊些前端那些方面的知识。  
说下你对线程池的理解，如何创建一个线程池与使用。  
说下你用过那些注解框架，他们的原理是什么。自己实现过，或是理解他的工作过程吗？  
说下java虚拟机的理解，回收机制，JVM是如何回收对象的，有哪些方法等  
一些java与Android源码相关知识等

### 1、 Android的四大组件是哪些，它们的作用？

答：Activity：Activity是Android程序与用户交互的窗口，是Android构造块中最基本的一种，它需要为保持各界面的状态，做很多持久化的事情，妥善管理生命周期以及一些跳转逻辑

service：后台服务于Activity，封装有一个完整的功能逻辑实现，接受上层指令，完成相关的事物，定义好需要接受的Intent提供同步和异步的接口

Content Provider：是Android提供的第三方应用数据的访问方案，可以派生Content Provider类，对外提供数据，可以像数据库一样进行选择排序，屏蔽内部数据的存储细节，向外提供统一的借口模型，大大简化上层应用，对数据的整合提供了更方便的途径

BroadCast Receiver：接受一种或者多种Intent作触发事件，接受相关消息，做一些简单处理，转换成一条Notification，统一了Android的事件广播模型

### 2、 请介绍下Android中常用的五种布局。

常用五种布局方式，分别是：FrameLayout（框架布局），LinearLayout （线性布局），AbsoluteLayout（绝对布局），RelativeLayout（相对布局），TableLayout（表格布局）。

一、FrameLayout：所有东西依次都放在左上角，会重叠，这个布局比较简单，也只能放一点比较简单的东西。二、LinearLayout：线性布局，每一个LinearLayout里面又可分为垂直布局（android:orientation="vertical"）和水平布局（android:orientation="horizontal" ）。当垂直布局时，每一行就只有一个元素，多个元素依次垂直往下；水平布局时，只有一行，每一个元素依次向右排列。三、AbsoluteLayout：绝对布局用X,Y坐标来指定元素的位置，这种布局方式也比较简单，但是在屏幕旋转时，往往会出问题，而且多个元素的时候，计算比较麻烦。四、RelativeLayout：相对布局可以理解为某一个元素为参照物，来定位的布局方式。主要属性有：相对于某一个元素android:layout\_below、      android:layout\_toLeftOf相对于父元素的地方android:layout\_alignParentLeft、android:layout\_alignParentRigh；五、TableLayout：表格布局，每一个TableLayout里面有表格行TableRow，TableRow里面可以具体定义每一个元素。每一个布局都有自己适合的方式，这五个布局元素可以相互嵌套应用，做出美观的界面。

### 3、 android中的动画有哪几类，它们的特点和区别是什么

答：两种，一种是Tween动画、还有一种是Frame动画。Tween动画，这种实现方式可以使视图组件移动、放大、缩小以及产生透明度的变化;另一种Frame动画，传统的动画方法，通过顺序的播放排列好的图片来实现，类似电影。

### 4、 android 中有哪几种解析xml的类？官方推荐哪种？以及它们的原理和区别。

答：XML解析主要有三种方式，SAX、DOM、PULL。常规在PC上开发我们使用Dom相对轻松些，但一些性能敏感的数据库或手机上还是主要采用SAX方式，SAX读取是单向的，优点:不占内存空间、解析属性方便，但缺点就是对于套嵌多个分支来说处理不是很方便。而DOM方式会把整个XML文件加载到内存中去，这里Android开发网提醒大家该方法在查找方面可以和XPath很好的结合如果数据量不是很大推荐使用，而PULL常常用在J2ME对于节点处理比较好，类似SAX方式，同样很节省内存，在J2ME中我们经常使用的KXML库来解析。

### 5、 ListView的优化方案

答：1、如果自定义适配器，那么在getView方法中要考虑方法传进来的参数contentView是否为null，如果为null就创建contentView并返回，如果不为null则直接使用。在这个方法中尽可能少创建view。

  2、给contentView设置tag（setTag（）），传入一个viewHolder对象，用于缓存要显示的数据，可以达到图像数据异步加载的效果。

  3、如果listview需要显示的item很多，就要考虑分页加载。比如一共要显示100条或者更多的时候，我们可以考虑先加载20条，等用户拉到列表底部的时候再去加载接下来的20条。

### 6、 请介绍下Android的数据存储方式。

答：使用SharedPreferences存储数据；文件存储数据；SQLite数据库存储数据；使用ContentProvider存储数据；网络存储数据；

Preference，File， DataBase这三种方式分别对应的目录是/data/data/Package Name/Shared\_Pref, /data/data/Package Name/files, /data/data/Package Name/database 。

一：使用SharedPreferences存储数据

首先说明SharedPreferences存储方式，它是 Android提供的用来存储一些简单配置信息的一种机制，例如：登录用户的用户名与密码。其采用了Map数据结构来存储数据，以键值的方式存储，可以简单的读取与写入，具体实例如下：

void ReadSharedPreferences(){

String strName,strPassword;

SharedPreferences   user = getSharedPreferences(“user\_info”,0);

strName = user.getString(“NAME”,””);

strPassword = user getString(“PASSWORD”,””);

}

void WriteSharedPreferences(String strName,String strPassword){

SharedPreferences   user = getSharedPreferences(“user\_info”,0);

uer.edit();

user.putString(“NAME”, strName);

user.putString(“PASSWORD” ,strPassword);

user.commit();

}

数据读取与写入的方法都非常简单，只是在写入的时候有些区别：先调用edit()使其处于编辑状态，然后才能修改数据，最后使用commit()提交修改的数据。实际上SharedPreferences是采用了XML格式将数据存储到设备中，在DDMS中的File Explorer中的/data/data/<package name>/shares\_prefs下。使用SharedPreferences是有些限制的：只能在同一个包内使用，不能在不同的包之间使用。

二：文件存储数据

文件存储方式是一种较常用的方法，在Android中读取/写入文件的方法，与 Java中实现I/O的程序是完全一样的，提供了openFileInput()和openFileOutput()方法来读取设备上的文件。具体实例如下:

String fn = “moandroid.log”;

FileInputStream fis = openFileInput(fn);

FileOutputStream fos = openFileOutput(fn,Context.MODE\_PRIVATE);

三：网络存储数据

网络存储方式，需要与Android 网络数据包打交道，关于Android 网络数据包的详细说明，请阅读Android SDK引用了Java SDK的哪些package？。

四：ContentProvider

1、ContentProvider简介

当应用继承ContentProvider类，并重写该类用于提供数据和存储数据的方法，就可以向其他应用共享其数据。虽然使用其他方法也可以对外共享数据，但数据访问方式会因数据存储的方式而不同，如：采用文件方式对外共享数据，需要进行文件操作读写数据；采用sharedpreferences共享数据，需要使用sharedpreferences API读写数据。而使用ContentProvider共享数据的好处是统一了数据访问方式。

2、Uri类简介

Uri代表了要操作的数据，Uri主要包含了两部分信息：1.需要操作的ContentProvider ，2.对ContentProvider中的什么数据进行操作，一个Uri由以下几部分组成：

1.scheme：ContentProvider（内容提供者）的scheme已经由Android所规定为：content://…

2.主机名（或Authority）：用于唯一标识这个ContentProvider，外部调用者可以根据这个标识来找到它。

3.路径（path）：可以用来表示我们要操作的数据，路径的构建应根据业务而定，如下：

要操作contact表中id为10的记录，可以构建这样的路径:/contact/10

要操作contact表中id为10的记录的name字段， contact/10/name

要操作contact表中的所有记录，可以构建这样的路径:/contact?

要操作的数据不一定来自数据库，也可以是文件等他存储方式，如下:

要操作xml文件中contact节点下的name节点，可以构建这样的路径：/contact/name

如果要把一个字符串转换成Uri，可以使用Uri类中的parse()方法，如下：

Uri uri = Uri.parse("content://com.changcheng.provider.contactprovider/contact")

3、UriMatcher、ContentUrist和ContentResolver简介

因为Uri代表了要操作的数据，所以我们很经常需要解析Uri，并从 Uri中获取数据。Android系统提供了两个用于操作Uri的工具类，分别为UriMatcher 和ContentUris 。掌握它们的使用，会便于我们的开发工作。

UriMatcher：用于匹配Uri，它的用法如下：

1.首先把你需要匹配Uri路径全部给注册上，如下：

//常量UriMatcher.NO\_MATCH表示不匹配任何路径的返回码(-1)。

UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

//如果match()方法匹配[content://com.changcheng.sqlite.provider.contactprovider /contact](/home/thtfit/文档\\x/content://com.changcheng.sqlite.provider.contactprovider%C2%A0/contact)路径，返回匹配码为1

uriMatcher.addURI(“com.changcheng.sqlite.provider.contactprovider”, “contact”, 1);//添加需要匹配uri，如果匹配就会返回匹配码

//如果match()方法匹配 <content://com.changcheng.sqlite.provider.contactprovider/contact/230>路径，返回匹配码为2

uriMatcher.addURI(“com.changcheng.sqlite.provider.contactprovider”, “contact/#”, 2);//#号为通配符

2.注册完需要匹配的Uri后，就可以使用uriMatcher.match(uri)方法对输入的Uri进行匹配，如果匹配就返回匹配码，匹配码是调用 addURI()方法传入的第三个参数，假设匹配 content://com.changcheng.sqlite.provider.contactprovider/contact路径，返回的匹配码为1。

ContentUris：用于获取Uri路径后面的ID部分，它有两个比较实用的方法：

withAppendedId(uri, id)用于为路径加上ID部分

parseId(uri)方法用于从路径中获取ID部分

ContentResolver：当外部应用需要对ContentProvider中的数据进行添加、删除、修改和查询操作时，可以使用 ContentResolver 类来完成，要获取ContentResolver 对象，可以使用Activity提供的getContentResolver()方法。 ContentResolver使用insert、delete、update、query方法，来操作数据。

### 7、activity的启动模式有哪些？是什么含义？

答：在android里，有4种activity的启动模式，分别为：

“standard” (默认)

“singleTop”

“singleTask”

“singleInstance”

它们主要有如下不同：

1. 如何决定所属task

“standard”和”singleTop”的activity的目标task，和收到的Intent的发送者在同一个task内，除非intent包括参数FLAG\_ACTIVITY\_NEW\_TASK。

如果提供了FLAG\_ACTIVITY\_NEW\_TASK参数，会启动到别的task里。

“singleTask”和”singleInstance”总是把activity作为一个task的根元素，他们不会被启动到一个其他task里。

2. 是否允许多个实例

“standard”和”singleTop”可以被实例化多次，并且存在于不同的task中，且一个task可以包括一个activity的多个实例；

“singleTask”和”singleInstance”则限制只生成一个实例，并且是task的根元素。 singleTop要求如果创建intent的时候栈顶已经有要创建的Activity的实例，则将intent发送给该实例，而不发送给新的实例。

3. 是否允许其它activity存在于本task内

“singleInstance”独占一个task，其它activity不能存在那个task里；如果它启动了一个新的activity，不管新的activity的launch mode 如何，新的activity都将会到别的task里运行（如同加了FLAG\_ACTIVITY\_NEW\_TASK参数）。

而另外三种模式，则可以和其它activity共存。

4. 是否每次都生成新实例

“standard”对于没一个启动Intent都会生成一个activity的新实例；

“singleTop”的activity如果在task的栈顶的话，则不生成新的该activity的实例，直接使用栈顶的实例，否则，生成该activity的实例。

比如现在task栈元素为A-B-C-D（D在栈顶），这时候给D发一个启动intent，如果D是 “standard”的，则生成D的一个新实例，栈变为A－B－C－D－D。

如果D是singleTop的话，则不会生产D的新实例，栈状态仍为A-B-C-D

如果这时候给B发Intent的话，不管B的launchmode是”standard” 还是 “singleTop” ，都会生成B的新实例，栈状态变为A-B-C-D-B。

“singleInstance”是其所在栈的唯一activity，它会每次都被重用。

“singleTask”如果在栈顶，则接受intent，否则，该intent会被丢弃，但是该task仍会回到前台。

当已经存在的activity实例处理新的intent时候，会调用onNewIntent()方法 如果收到intent生成一个activity实例，那么用户可以通过back键回到上一个状态；如果是已经存在的一个activity来处理这个intent的话，用户不能通过按back键返回到这之前的状态。

### 8、跟activity和Task 有关的 Intent启动方式有哪些？其含义？

核心的Intent Flag有：

FLAG\_ACTIVITY\_NEW\_TASK

FLAG\_ACTIVITY\_CLEAR\_TOP

FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED

FLAG\_ACTIVITY\_SINGLE\_TOP

FLAG\_ACTIVITY\_NEW\_TASK

  如果设置，这个Activity会成为历史stack中一个新Task的开始。一个Task（从启动它的Activity到下一个Task中的 Activity）定义了用户可以迁移的Activity原子组。Task可以移动到前台和后台；在某个特定Task中的所有Activity总是保持相同的次序。

  这个标志一般用于呈现“启动”类型的行为：它们提供用户一系列可以单独完成的事情，与启动它们的Activity完全无关。

使用这个标志，如果正在启动的Activity的Task已经在运行的话，那么，新的Activity将不会启动；代替的，当前Task会简单的移入前台。参考FLAG\_ACTIVITY\_MULTIPLE\_TASK标志，可以禁用这一行为。

  这个标志不能用于调用方对已经启动的Activity请求结果。

FLAG\_ACTIVITY\_CLEAR\_TOP

  如果设置，并且这个Activity已经在当前的Task中运行，因此，不再是重新启动一个这个Activity的实例，而是在这个Activity上方的所有Activity都将关闭，然后这个Intent会作为一个新的Intent投递到老的Activity（现在位于顶端）中。

  例如，假设一个Task中包含这些Activity：A，B，C，D。如果D调用了startActivity()，并且包含一个指向Activity B的Intent，那么，C和D都将结束，然后B接收到这个Intent，因此，目前stack的状况是：A，B。

  上例中正在运行的Activity B既可以在onNewIntent()中接收到这个新的Intent，也可以把自己关闭然后重新启动来接收这个Intent。如果它的启动模式声明为 “multiple”(默认值)，并且你没有在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，那么它将关闭然后重新创建；对于其它的启动模式，或者在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，都将把这个Intent投递到当前这个实例的onNewIntent()中。

  这个启动模式还可以与FLAG\_ACTIVITY\_NEW\_TASK结合起来使用：用于启动一个Task中的根Activity，它会把那个Task中任何运行的实例带入前台，然后清除它直到根Activity。这非常有用，例如，当从Notification Manager处启动一个Activity。

FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED

    如果设置这个标志，这个activity不管是从一个新的栈启动还是从已有栈推到栈顶，它都将以the front door of the task的方式启动。这就讲导致任何与应用相关的栈都讲重置到正常状态（不管是正在讲activity移入还是移除），如果需要，或者直接重置该栈为初始状态。

FLAG\_ACTIVITY\_SINGLE\_TOP

  如果设置，当这个Activity位于历史stack的顶端运行时，不再启动一个新的

FLAG\_ACTIVITY\_BROUGHT\_TO\_FRONT

  这个标志一般不是由程序代码设置的，如在launchMode中设置singleTask模式时系统帮你设定。

FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET

  如果设置，这将在Task的Activity stack中设置一个还原点，当Task恢复时，需要清理Activity。也就是说，下一次Task带着 FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED标记进入前台时（典型的操作是用户在主画面重启它），这个Activity和它之上的都将关闭，以至于用户不能再返回到它们，但是可以回到之前的Activity。

  这在你的程序有分割点的时候很有用。例如，一个e-mail应用程序可能有一个操作是查看一个附件，需要启动图片浏览Activity来显示。这个 Activity应该作为e-mail应用程序Task的一部分，因为这是用户在这个Task中触发的操作。然而，当用户离开这个Task，然后从主画面选择e-mail app，我们可能希望回到查看的会话中，但不是查看图片附件，因为这让人困惑。通过在启动图片浏览时设定这个标志，浏览及其它启动的Activity在下次用户返回到mail程序时都将全部清除。

FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS

  如果设置，新的Activity不会在最近启动的Activity的列表中保存。

FLAG\_ACTIVITY\_FORWARD\_RESULT

  如果设置，并且这个Intent用于从一个存在的Activity启动一个新的Activity，那么，这个作为答复目标的Activity将会传到这个新的Activity中。这种方式下，新的Activity可以调用setResult(int)，并且这个结果值将发送给那个作为答复目标的 Activity。

FLAG\_ACTIVITY\_LAUNCHED\_FROM\_HISTORY

  这个标志一般不由应用程序代码设置，如果这个Activity是从历史记录里启动的（常按HOME键），那么，系统会帮你设定。

FLAG\_ACTIVITY\_MULTIPLE\_TASK

  不要使用这个标志，除非你自己实现了应用程序启动器。与FLAG\_ACTIVITY\_NEW\_TASK结合起来使用，可以禁用把已存的Task送入前台的行为。当设置时，新的Task总是会启动来处理Intent，而不管这是是否已经有一个Task可以处理相同的事情。

  由于默认的系统不包含图形Task管理功能，因此，你不应该使用这个标志，除非你提供给用户一种方式可以返回到已经启动的Task。

  如果FLAG\_ACTIVITY\_NEW\_TASK标志没有设置，这个标志被忽略。

FLAG\_ACTIVITY\_NO\_ANIMATION

  如果在Intent中设置，并传递给Context.startActivity()的话，这个标志将阻止系统进入下一个Activity时应用 Acitivity迁移动画。这并不意味着动画将永不运行——如果另一个Activity在启动显示之前，没有指定这个标志，那么，动画将被应用。这个标志可以很好的用于执行一连串的操作，而动画被看作是更高一级的事件的驱动。

FLAG\_ACTIVITY\_NO\_HISTORY

  如果设置，新的Activity将不再历史stack中保留。用户一离开它，这个Activity就关闭了。这也可以通过设置noHistory特性。

FLAG\_ACTIVITY\_NO\_USER\_ACTION

  如果设置，作为新启动的Activity进入前台时，这个标志将在Activity暂停之前阻止从最前方的Activity回调的onUserLeaveHint()。

  典型的，一个Activity可以依赖这个回调指明显式的用户动作引起的Activity移出后台。这个回调在Activity的生命周期中标记一个合适的点，并关闭一些Notification。

  如果一个Activity通过非用户驱动的事件，如来电或闹钟，启动的，这个标志也应该传递给Context.startActivity，保证暂停的Activity不认为用户已经知晓其Notification。

FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP

  If set and this intent is being used to launch a new activity from an existing one, the current activity will not be counted as the top activity for deciding whether the new intent should be delivered to the top instead of starting a new one. The previous activity will be used as the top, with the assumption being that the current activity will finish itself immediately.

FLAG\_ACTIVITY\_REORDER\_TO\_FRONT

  如果在Intent中设置，并传递给Context.startActivity()，这个标志将引发已经运行的Activity移动到历史stack的顶端。

  例如，假设一个Task由四个Activity组成：A,B,C,D。如果D调用startActivity()来启动Activity B，那么，B会移动到历史stack的顶端，现在的次序变成A,C,D,B。如果FLAG\_ACTIVITY\_CLEAR\_TOP标志也设置的话，那么这个标志将被忽略。

### 9、 请描述下Activity的生命周期。

答：activity的生命周期方法有：onCreate()、onStart()、onReStart()、onResume()、onPause()、onStop()、onDestory()；

可见生命周期：从onStart()直到系统调用onStop()

前台生命周期：从onResume()直到系统调用onPause()

### 10、 activity在屏幕旋转时的生命周期

答：不设置Activity的android:configChanges时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次；设置Activity的android:configChanges="orientation"时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次；设置Activity的android:configChanges="orientation|keyboardHidden"时，切屏不会重新调用各个生命周期，只会执行onConfigurationChanged方法

### 11、 如何启用Service，如何停用Service。

服务的开发比较简单，如下：

第一步：继承Service类

1

public class SMSService extends Service {}

第二步：在AndroidManifest.xml文件中的<application>节点里对服务进行配置:<service android:name=".SMSService" />

服务不能自己运行，需要通过调用Context.startService()或Context.bindService()方法启动服务。这两个方法都可以启动Service，但是它们的使用场合有所不同。使用startService()方法启用服务，调用者与服务之间没有关连，即使调用者退出了，服务仍然运行。使用bindService()方法启用服务，调用者与服务绑定在了一起，调用者一旦退出，服务也就终止，大有“不求同时生，必须同时死”的特点。

如果打算采用Context.startService()方法启动服务，在服务未被创建时，系统会先调用服务的onCreate()方法，接着调用onStart()方法。如果调用startService()方法前服务已经被创建，多次调用startService()方法并不会导致多次创建服务，但会导致多次调用onStart()方法。采用startService()方法启动的服务，只能调用Context.stopService()方法结束服务，服务结束时会调用onDestroy()方法。

如果打算采用Context.bindService()方法启动服务，在服务未被创建时，系统会先调用服务的onCreate()方法，接着调用onBind()方法。这个时候调用者和服务绑定在一起，调用者退出了，系统就会先调用服务的onUnbind()方法，接着调用onDestroy()方法。如果调用bindService()方法前服务已经被绑定，多次调用bindService()方法并不会导致多次创建服务及绑定(也就是说onCreate()和onBind()方法并不会被多次调用)。如果调用者希望与正在绑定的服务解除绑定，可以调用unbindService()方法，调用该方法也会导致系统调用服务的onUnbind()-->onDestroy()方法。

服务常用生命周期回调方法如下：

onCreate() 该方法在服务被创建时调用，该方法只会被调用一次，无论调用多少次startService()或bindService()方法，服务也只被创建一次。

onDestroy()该方法在服务被终止时调用。

与采用Context.startService()方法启动服务有关的生命周期方法

onStart() 只有采用Context.startService()方法启动服务时才会回调该方法。该方法在服务开始运行时被调用。多次调用startService()方法尽管不会多次创建服务，但onStart() 方法会被多次调用。

与采用Context.bindService()方法启动服务有关的生命周期方法

onBind()只有采用Context.bindService()方法启动服务时才会回调该方法。该方法在调用者与服务绑定时被调用，当调用者与服务已经绑定，多次调用Context.bindService()方法并不会导致该方法被多次调用。

onUnbind()只有采用Context.bindService()方法启动服务时才会回调该方法。该方法在调用者与服务解除绑定时被调用

### 12、 注册广播有几种方式，这些方式有何优缺点？请谈谈Android引入广播机制的用意。

答：首先写一个类要继承BroadcastReceiver

第一种:在清单文件中声明,添加

<receive android:name=".IncomingSMSReceiver " >

<intent-filter>

<action android:name="android.provider.Telephony.SMS\_RECEIVED")

<intent-filter>

<receiver>

第二种使用代码进行注册如:

IntentFilter filter =  new IntentFilter("android.provider.Telephony.SMS\_RECEIVED");

IncomingSMSReceiver receiver = new IncomgSMSReceiver();

registerReceiver(receiver.filter);

两种注册类型的区别是：

1)第一种不是常驻型广播，也就是说广播跟随程序的生命周期。

2)第二种是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

### 13、 请解释下在单线程模型中Message、Handler、Message Queue、Looper之间的关系。

答：简单的说，Handler获取当前线程中的looper对象，looper用来从存放Message的MessageQueue中取出Message，再有Handler进行Message的分发和处理.

Message Queue(消息队列)：用来存放通过Handler发布的消息，通常附属于某一个创建它的线程，可以通过Looper.myQueue()得到当前线程的消息队列

Handler：可以发布或者处理一个消息或者操作一个Runnable，通过Handler发布消息，消息将只会发送到与它关联的消息队列，然也只能处理该消息队列中的消息

Looper：是Handler和消息队列之间通讯桥梁，程序组件首先通过Handler把消息传递给Looper，Looper把消息放入队列。Looper也把消息队列里的消息广播给所有的

Handler：Handler接受到消息后调用handleMessage进行处理

Message：消息的类型，在Handler类中的handleMessage方法中得到单个的消息进行处理

在单线程模型下，为了线程通信问题，Android设计了一个Message Queue(消息队列)， 线程间可以通过该Message Queue并结合Handler和Looper组件进行信息交换。下面将对它们进行分别介绍：

1. Message

    Message消息，理解为线程间交流的信息，处理数据后台线程需要更新UI，则发送Message内含一些数据给UI线程。

2. Handler

    Handler处理者，是Message的主要处理者，负责Message的发送，Message内容的执行处理。后台线程就是通过传进来的 Handler对象引用来sendMessage(Message)。而使用Handler，需要implement 该类的 handleMessage(Message)方法，它是处理这些Message的操作内容，例如Update UI。通常需要子类化Handler来实现handleMessage方法。

3. Message Queue

    Message Queue消息队列，用来存放通过Handler发布的消息，按照先进先出执行。

    每个message queue都会有一个对应的Handler。Handler会向message queue通过两种方法发送消息：sendMessage或post。这两种消息都会插在message queue队尾并按先进先出执行。但通过这两种方法发送的消息执行的方式略有不同：通过sendMessage发送的是一个message对象,会被 Handler的handleMessage()函数处理；而通过post方法发送的是一个runnable对象，则会自己执行。

4. Looper

    Looper是每条线程里的Message Queue的管家。Android没有Global的Message Queue，而Android会自动替主线程(UI线程)建立Message Queue，但在子线程里并没有建立Message Queue。所以调用Looper.getMainLooper()得到的主线程的Looper不为NULL，但调用Looper.myLooper() 得到当前线程的Looper就有可能为NULL。对于子线程使用Looper，API Doc提供了正确的使用方法：这个Message机制的大概流程：

    1. 在Looper.loop()方法运行开始后，循环地按照接收顺序取出Message Queue里面的非NULL的Message。

    2. 一开始Message Queue里面的Message都是NULL的。当Handler.sendMessage(Message)到Message Queue，该函数里面设置了那个Message对象的target属性是当前的Handler对象。随后Looper取出了那个Message，则调用 该Message的target指向的Hander的dispatchMessage函数对Message进行处理。在dispatchMessage方法里，如何处理Message则由用户指定，三个判断，优先级从高到低：

    1) Message里面的Callback，一个实现了Runnable接口的对象，其中run函数做处理工作；

    2) Handler里面的mCallback指向的一个实现了Callback接口的对象，由其handleMessage进行处理；

    3) 处理消息Handler对象对应的类继承并实现了其中handleMessage函数，通过这个实现的handleMessage函数处理消息。

    由此可见，我们实现的handleMessage方法是优先级最低的！

    3. Handler处理完该Message (update UI) 后，Looper则设置该Message为NULL，以便回收！

    在网上有很多文章讲述主线程和其他子线程如何交互，传送信息，最终谁来执行处理信息之类的，个人理解是最简单的方法——判断Handler对象里面的Looper对象是属于哪条线程的，则由该线程来执行！

    1. 当Handler对象的构造函数的参数为空，则为当前所在线程的Looper；

2. Looper.getMainLooper()得到的是主线程的Looper对象，Looper.myLooper()得到的是当前线程的Looper对象。

### 14、 简要解释一下activity、 intent 、intent filter、service、Broadcase、BroadcaseReceiver

答：一个activity呈现了一个用户可以操作的可视化用户界面；一个service不包含可见的用户界面，而是在后台运行，可以与一个activity绑定，通过绑定暴露出来接口并与其进行通信；一个broadcast receiver是一个接收广播消息并做出回应的component，broadcast receiver没有界面；一个intent是一个Intent对象，它保存了消息的内容。对于activity和service来说，它指定了请求的操作名称和待操作数据的URI，Intent对象可以显式的指定一个目标component。如果这样的话，android会找到这个component(基于manifest文件中的声明)并激活它。但如果一个目标不是显式指定的，android必须找到响应intent的最佳component。它是通过将Intent对象和目标的intent filter相比较来完成这一工作的；一个component的intent filter告诉android该component能处理的intent。intent filter也是在manifest文件中声明的。

### 15、 说说mvc模式的原理，它在android中的运用,android的官方建议应用程序的开发采用mvc模式。何谓mvc？

mvc是model,view,controller的缩写，mvc包含三个部分：

模型（model）对象：是应用程序的主体部分，所有的业务逻辑都应该写在该层。

视图（view）对象：是应用程序中负责生成用户界面的部分。也是在整个mvc架构中用户唯一可以看到的一层，接收用户的输入，显示处理结果。

控制器（control）对象：是根据用户的输入，控制用户界面数据显示及更新model对象状态的部分，控制器更重要的一种导航功能，响应用户出发的相关事件，交给m层处理。

android鼓励弱耦合和组件的重用，在android中mvc的具体体现如下：

1)视图层（view）：一般采用xml文件进行界面的描述，使用的时候可以非常方便的引入，当然，如果你对android了解的比较的多了话，就一定可以想到在android中也可以使用JavaScript+html等的方式作为view层，当然这里需要进行java和javascript之间的通信，幸运的是，android提供了它们之间非常方便的通信实现。

2)控制层（controller）：android的控制层的重任通常落在了众多的acitvity的肩上，这句话也就暗含了不要在acitivity中写代码，要通过activity交割model业务逻辑层处理，这样做的另外一个原因是android中的acitivity的响应时间是5s，如果耗时的操作放在这里，程序就很容易被回收掉。

3)模型层（model）：对数据库的操作、对网络等的操作都应该在model里面处理，当然对业务计算等操作也是必须放在的该层的。

### 16、 什么是ANR 如何避免它？

答：ANR：Application Not Responding。在Android中，活动管理器和窗口管理器这两个系统服务负责监视应用程序的响应，当用户操作的在5s内应用程序没能做出反应，BroadcastReceiver在10秒内没有执行完毕，就会出现应用程序无响应对话框，这既是ANR。

避免方法：Activity应该在它的关键生命周期方法（如onCreate()和onResume()）里尽可能少的去做创建操作。潜在的耗时操作，例如网络或数据库操作，或者高耗时的计算如改变位图尺寸，应该在子线程里（或者异步方式）来完成。主线程应该为子线程提供一个Handler，以便完成时能够提交给主线程。

### 17、 什么情况会导致Force Close ？如何避免？能否捕获导致其的异常？

答：程序出现异常，比如nullpointer。

避免：编写程序时逻辑连贯，思维缜密。能捕获异常，在logcat中能看到异常信息

### 18、 描述一下android的系统架构

android系统架构分从下往上为linux 内核层、运行库、应用程序框架层、和应用程序层。

linuxkernel：负责硬件的驱动程序、网络、电源、系统安全以及内存管理等功能。

libraries和 android runtime：libraries：即c/c++函数库部分，大多数都是开放源代码的函数库，例如webkit（引擎），该函数库负责 android网页浏览器的运行，例如标准的c函数库libc、openssl、sqlite等，当然也包括支持游戏开发2dsgl和 3dopengles，在多媒体方面有mediaframework框架来支持各种影音和图形文件的播放与显示，例如mpeg4、h.264、mp3、 aac、amr、jpg和png等众多的多媒体文件格式。android的runtime负责解释和执行生成的dalvik格式的字节码。

applicationframework（应用软件架构），java应用程序开发人员主要是使用该层封装好的api进行快速开发。

applications:该层是java的应用程序层，android内置的googlemaps、e-mail、即时通信工具、浏览器、mp3播放器等处于该层，java开发人员开发的程序也处于该层，而且和内置的应用程序具有平等的位置，可以调用内置的应用程序，也可以替换内置的应用程序。

上面的四个层次，下层为上层服务，上层需要下层的支持，调用下层的服务，这种严格分层的方式带来的极大的稳定性、灵活性和可扩展性，使得不同层的开发人员可以按照规范专心特定层的开发。

android应用程序使用框架的api并在框架下运行，这就带来了程序开发的高度一致性，另一方面也告诉我们，要想写出优质高效的程序就必须对整个 applicationframework进行非常深入的理解。精通applicationframework，你就可以真正的理解android的设计和运行机制，也就更能够驾驭整个应用层的开发。

### 19、 请介绍下ContentProvider是如何实现数据共享的。

一个程序可以通过实现一个Content provider的抽象接口将自己的数据完全暴露出去，而且Content providers是以类似数据库中表的方式将数据暴露。Content providers存储和检索数据，通过它可以让所有的应用程序访问到，这也是应用程序之间唯一共享数据的方法。

要想使应用程序的数据公开化，可通过2种方法：创建一个属于你自己的Content provider或者将你的数据添加到一个已经存在的Content provider中，前提是有相同数据类型并且有写入Content provider的权限。

如何通过一套标准及统一的接口获取其他应用程序暴露的数据？

Android提供了ContentResolver，外界的程序可以通过ContentResolver接口访问ContentProvider提供的数据。

### 20、 Service和Thread的区别？

答：servie是系统的组件，它由系统进程托管（servicemanager）；它们之间的通信类似于client和server，是一种轻量级的ipc通信，这种通信的载体是binder，它是在linux层交换信息的一种ipc。而thread是由本应用程序托管。1). Thread：Thread 是程序执行的最小单元，它是分配CPU的基本单位。可以用 Thread 来执行一些异步的操作。

2). Service：Service 是android的一种机制，当它运行的时候如果是Local Service，那么对应的 Service 是运行在主进程的 main 线程上的。如：onCreate，onStart 这些函数在被系统调用的时候都是在主进程的 main 线程上运行的。如果是Remote Service，那么对应的 Service 则是运行在独立进程的 main 线程上。

既然这样，那么我们为什么要用 Service 呢？其实这跟 android 的系统机制有关，我们先拿 Thread 来说。Thread 的运行是独立于 Activity 的，也就是说当一个 Activity 被 finish 之后，如果你没有主动停止 Thread 或者 Thread 里的 run 方法没有执行完毕的话，Thread 也会一直执行。因此这里会出现一个问题：当 Activity 被 finish 之后，你不再持有该 Thread 的引用。另一方面，你没有办法在不同的 Activity 中对同一 Thread 进行控制。

举个例子：如果你的 Thread 需要不停地隔一段时间就要连接服务器做某种同步的话，该 Thread 需要在 Activity 没有start的时候也在运行。这个时候当你 start 一个 Activity 就没有办法在该 Activity 里面控制之前创建的 Thread。因此你便需要创建并启动一个 Service ，在 Service 里面创建、运行并控制该 Thread，这样便解决了该问题（因为任何 Activity 都可以控制同一 Service，而系统也只会创建一个对应 Service 的实例）。

因此你可以把 Service 想象成一种消息服务，而你可以在任何有 Context 的地方调用 Context.startService、Context.stopService、Context.bindService，Context.unbindService，来控制它，你也可以在 Service 里注册 BroadcastReceiver，在其他地方通过发送 broadcast 来控制它，当然这些都是 Thread 做不到的。

### 21、 Android本身的api并未声明会抛出异常，则其在运行时有无可能抛出runtime异常，你遇到过吗？诺有的话会导致什么问题？如何解决？

答：会，比如nullpointerException。我遇到过，比如textview.setText()时，textview没有初始化。会导致程序无法正常运行出现forceclose。打开控制台查看logcat信息找出异常信息并修改程序。

### 22、 IntentService有何优点?

答：Acitivity的进程，当处理Intent的时候，会产生一个对应的Service； Android的进程处理器现在会尽可能的不kill掉你；非常容易使用

### 23、 如果后台的Activity由于某原因被系统回收了，如何在被系统回收之前保存当前状态？

答：重写onSaveInstanceState()方法，在此方法中保存需要保存的数据，该方法将会在activity被回收之前调用。通过重写onRestoreInstanceState()方法可以从中提取保存好的数据

### 24、 如何将一个Activity设置成窗口的样式。

答：<activity>中配置：android :theme="@android:style/Theme.Dialog"

另外android:theme="@android:style/Theme.Translucent" 是设置透明

### 25、 如何退出Activity？如何安全退出已调用多个Activity的Application？

答：对于单一Activity的应用来说，退出很简单，直接finish()即可。当然，也可以用killProcess()和System.exit()这样的方法。

对于多个activity，1、记录打开的Activity：每打开一个Activity，就记录下来。在需要退出时，关闭每一个Activity即可。2、发送特定广播：在需要结束应用时，发送一个特定的广播，每个Activity收到广播后，关闭即可。3、递归退出：在打开新的Activity时使用startActivityForResult，然后自己加标志，在onActivityResult中处理，递归关闭。为了编程方便，最好定义一个Activity基类，处理这些共通问题。

在2.1之前，可以使用ActivityManager的restartPackage方法。

它可以直接结束整个应用。在使用时需要权限android.permission.RESTART\_PACKAGES。

注意不要被它的名字迷惑。

可是，在2.2，这个方法失效了。在2.2添加了一个新的方法，killBackground Processes()，需要权限 android.permission.KILL\_BACKGROUND\_PROCESSES。可惜的是，它和2.2的restartPackage一样，根本起不到应有的效果。

另外还有一个方法，就是系统自带的应用程序管理里，强制结束程序的方法，forceStopPackage()。它需要权限android.permission.FORCE\_STOP\_PACKAGES。并且需要添加android:sharedUserId="android.uid.system"属性。同样可惜的是，该方法是非公开的，他只能运行在系统进程，第三方程序无法调用。

因为需要在Android.mk中添加LOCAL\_CERTIFICATE := platform。

而Android.mk是用于在Android源码下编译程序用的。

从以上可以看出，在2.2，没有办法直接结束一个应用，而只能用自己的办法间接办到。

现提供几个方法，供参考：

1、抛异常强制退出：

该方法通过抛异常，使程序Force Close。

验证可以，但是，需要解决的问题是，如何使程序结束掉，而不弹出Force Close的窗口。

2、记录打开的Activity：

每打开一个Activity，就记录下来。在需要退出时，关闭每一个Activity即可。

3、发送特定广播：

在需要结束应用时，发送一个特定的广播，每个Activity收到广播后，关闭即可。

4、递归退出

在打开新的Activity时使用startActivityForResult，然后自己加标志，在onActivityResult中处理，递归关闭。

除了第一个，都是想办法把每一个Activity都结束掉，间接达到目的。但是这样做同样不完美。你会发现，如果自己的应用程序对每一个Activity都设置了nosensor，在两个Activity结束的间隙，sensor可能有效了。但至少，我们的目的达到了，而且没有影响用户使用。为了编程方便，最好定义一个Activity基类，处理这些共通问题。

### 26、 AIDL的全称是什么？如何工作？能处理哪些类型的数据？

答：全称是：Android Interface Define Language

在Android中, 每个应用程序都可以有自己的进程. 在写UI应用的时候, 经常要用到Service. 在不同的进程中, 怎样传递对象呢?显然, Java中不允许跨进程内存共享. 因此传递对象, 只能把对象拆分成操作系统能理解的简单形式, 以达到跨界对象访问的目的. 在J2EE中,采用RMI的方式, 可以通过序列化传递对象. 在Android中, 则采用AIDL的方式. 理论上AIDL可以传递Bundle,实际上做起来却比较麻烦。

AIDL(AndRoid接口描述语言)是一种借口描述语言; 编译器可以通过aidl文件生成一段代码，通过预先定义的接口达到两个进程内部通信进程的目的. 如果需要在一个Activity中, 访问另一个Service中的某个对象, 需要先将对象转化成AIDL可识别的参数(可能是多个参数), 然后使用AIDL来传递这些参数, 在消息的接收端, 使用这些参数组装成自己需要的对象.

AIDL的IPC的机制和COM或CORBA类似, 是基于接口的，但它是轻量级的。它使用代理类在客户端和实现层间传递值. 如果要使用AIDL, 需要完成2件事情: 1. 引入AIDL的相关类.; 2. 调用aidl产生的class.

AIDL的创建方法:

AIDL语法很简单,可以用来声明一个带一个或多个方法的接口，也可以传递参数和返回值。 由于远程调用的需要, 这些参数和返回值并不是任何类型.下面是些AIDL支持的数据类型:

1. 不需要import声明的简单Java编程语言类型(int,boolean等)

2. String, CharSequence不需要特殊声明

3. List, Map和Parcelables类型, 这些类型内所包含的数据成员也只能是简单数据类型, String等其他比支持的类型.

(另外: 我没尝试Parcelables, 在Eclipse+ADT下编译不过, 或许以后会有所支持)

### 27、 请解释下Android程序运行时权限与文件系统权限的区别。

答：运行时权限Dalvik( android授权)

文件系统 linux 内核授权

### 28、 系统上安装了多种浏览器，能否指定某浏览器访问指定页面？请说明原由。

通过直接发送Uri把参数带过去，或者通过manifest里的intentfilter里的data属性

### 29、 android系统的优势和不足

答：Android平台手机 5大优势：

一、开放性

在优势方面，Android平台首先就是其开发性，开发的平台允许任何移动终端厂商加入到Android联盟中来。显著的开放性可以使其拥有更多的开发者，随着用户和应用的日益丰富，一个崭新的平台也将很快走向成熟。开放性对于Android的发展而言，有利于积累人气，这里的人气包括消费者和厂商，而对于消费者来讲，随大的受益正是丰富的软件资源。开放的平台也会带来更大竞争，如此一来，消费者将可以用更低的价位购得心仪的手机。

二、挣脱运营商的束缚

在过去很长的一段时间，特别是在欧美地区，手机应用往往受到运营商制约，使用什么功能接入什么网络，几乎都受到运营商的控制。从去年iPhone 上市 ，用户可以更加方便地连接网络，运营商的制约减少。随着EDGE、HSDPA这些2G至3G移动网络的逐步过渡和提升，手机随意接入网络已不是运营商口中的笑谈，当你可以通过手机IM软件方便地进行即时聊天时，再回想不久前天价的彩信和图铃下载业务，是不是像噩梦一样？互联网巨头Google推动的Android终端天生就有网络特色，将让用户离互联网更近。

三、丰富的硬件选择

这一点还是与Android平台的开放性相关，由于Android的开放性，众多的厂商会推出千奇百怪，功能特色各具的多种产品。功能上的差异和特色，却不会影响到数据同步、甚至软件的兼容，好比你从诺基亚 Symbian风格手机 一下改用苹果 iPhone ，同时还可将Symbian中优秀的软件带到iPhone上使用、联系人等资料更是可以方便地转移，是不是非常方便呢？

四、不受任何限制的开发商

Android平台提供给第三方开发商一个十分宽泛、自由的环境，不会受到各种条条框框的阻扰，可想而知，会有多少新颖别致的软件会诞生。但也有其两面性，血腥、暴力、情色方面的程序和游戏如可控制正是留给Android难题之一。

五、无缝结合的Google应用

如今叱诧互联网的Google已经走过10年度历史，从搜索巨人到全面的互联网渗透，Google服务如地图、邮件、搜索等已经成为连接用户和互联网的重要纽带，而Android平台手机将无缝结合这些优秀的Google服务。

再说Android的5大不足：

一、安全和隐私

由于手机 与互联网的紧密联系，个人隐私很难得到保守。除了上网过程中经意或不经意留下的个人足迹，Google这个巨人也时时站在你的身后，洞穿一切，因此，互联网的深入将会带来新一轮的隐私危机。

二、首先开卖Android手机的不是最大运营商

众所周知，T-Mobile在23日，于美国纽约发布 了Android首款手机G1。但是在北美市场，最大的两家运营商乃AT&T和Verizon，而目前所知取得Android手机销售权的仅有 T-Mobile和Sprint，其中T-Mobile的3G网络相对于其他三家也要逊色不少，因此，用户可以买账购买G1，能否体验到最佳的3G网络服务则要另当别论了！

三、运营商仍然能够影响到Android手机

在国内市场，不少用户对购得移动定制机不满，感觉所购的手机被人涂画了广告一般。这样的情况在国外市场同样出现。Android手机的另一发售运营商Sprint就将在其机型中内置其手机商店程序。

四、同类机型用户减少

在不少手机论坛都会有针对某一型号的子论坛，对一款手机的使用心得交流，并分享软件资源。而对于Android平台手机，由于厂商丰富，产品类型多样，这样使用同一款机型的用户越来越少，缺少统一机型的程序强化。举个稍显不当的例子，现在山寨机泛滥，品种各异，就很少有专门针对某个型号山寨机的讨论和群组，除了哪些功能异常抢眼、颇受追捧的机型以外。

五、过分依赖开发商缺少标准配置

在使用PC端的Windows Xp系统的时候，都会内置微软Windows Media Player这样一个浏览器程序，用户可以选择更多样的播放器，如Realplay或暴风影音等。但入手开始使用默认的程序同样可以应付多样的需要。在 Android平台中，由于其开放性，软件更多依赖第三方厂商，比如Android系统的SDK中就没有内置音乐 播放器，全部依赖第三方开发，缺少了产品的统一性。

### 30、 Android dvm的进程和Linux的进程, 应用程序的进程是否为同一个概念

答：DVM指dalivk的虚拟机。每一个Android应用程序都在它自己的进程中运行，都拥有一个独立的Dalvik虚拟机实例。而每一个DVM都是在Linux 中的一个进程，所以说可以认为是同一个概念。

### 31、 sim卡的EF文件是什么？有何作用

答：sim卡的文件系统有自己规范，主要是为了和手机通讯，sim本 身可以有自己的操作系统，EF就是作存储并和手机通讯用的

### 32、 嵌入式操作系统内存管理有哪几种， 各有何特性

页式，段式，段页，用到了MMU,虚拟空间等技术

### 33、 什么是嵌入式实时操作系统, Android 操作系统属于实时操作系统吗?

嵌入式实时操作系统是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统作出快速响应，并控制所有实时任务协调一致运行的嵌入式操作系统。主要用于工业控制、 军事设备、 航空航天等领域对系统的响应时间有苛刻的要求，这就需要使用实时系统。又可分为软实时和硬实时两种，而android是基于linux内核的，因此属于软实时。

### 34、 一条最长的短信息约占多少byte?

中文70(包括标点)，英文160，160个字节。

### 35、 如何将SQLite数据库(dictionary.db文件)与apk文件一起发布

解答：可以将dictionary.db文件复制到Eclipse Android工程中的res aw目录中。所有在res aw目录中的文件不会被压缩，这样可以直接提取该目录中的文件。可以将dictionary.db文件复制到res aw目录中

### 36、 如何将打开res aw目录中的数据库文件?

解答：在Android中不能直接打开res aw目录中的数据库文件，而需要在程序第一次启动时将该文件复制到手机内存或SD卡的某个目录中，然后再打开该数据库文件。

复制的基本方法是使用getResources().openRawResource方法获得res aw目录中资源的 InputStream对象，然后将该InputStream对象中的数据写入其他的目录中相应文件中。在Android SDK中可以使用SQLiteDatabase.openOrCreateDatabase方法来打开任意目录中的SQLite数据库文件。

### 37、 DDMS和TraceView的区别?

DDMS是一个程序执行查看器，在里面可以看见线程和堆栈等信息，TraceView是程序性能分析器 。

### 38、 java中如何引用本地语言

可以用JNI（java native interface  java 本地接口）接口 。

### 39、 谈谈Android的IPC（进程间通信）机制

IPC是内部进程通信的简称， 是共享"命名管道"的资源。Android中的IPC机制是为了让Activity和Service之间可以随时的进行交互，故在Android中该机制，只适用于Activity和Service之间的通信，类似于远程方法调用，类似于C/S模式的访问。通过定义AIDL接口文件来定义IPC接口。Servier端实现IPC接口，Client端调用IPC接口本地代理。

### 40、 NDK是什么

NDK是一些列工具的集合，NDK提供了一系列的工具，帮助开发者迅速的开发C/C++的动态库，并能自动将so和java 应用打成apk包。

NDK集成了交叉编译器，并提供了相应的mk文件和隔离cpu、平台等的差异，开发人员只需简单的修改mk文件就可以创建出so

有良好的JAVA基础，熟练掌握面向对象思想：

### 理解面向对象：

面向对象是一种思想，是基于面向过程而言的，就是说面向对象是将功能等通过对象来实现，将功能封装进对象之中，让对象去实现具体的细节；这种思想是将数据作为第一位，而方法或者说是算法作为其次，这是对数据一种优化，操作起来更加的方便，简化了过程。面向对象有三大特征：封装性、继承性、多态性，其中封装性指的是隐藏了对象的属性和实现细节，仅对外提供公共的访问方式，这样就隔离了具体的变化，便于使用，提高了复用性和安全性。对于继承性，就是两种事物间存在着一定的所属关系，那么继承的类就可以从被继承的类中获得一些属性和方法；这就 提高了代码的复用性。继承是作为多态的前提的。多态是说父类或接口的引用指向了子类对A象，这就提高了程序的扩展性，也就是说只要实现或继承了同一个接口或类，那么就可以使用父类中相应的方法，提高程序扩展性，但是多态有一点不好之处在于：父类引用不能访问子类中的成员。

举例来说：就是：比如说你要去饭店吃饭，你只需要饭店，找到饭店的服务员，跟她说你要吃什么，然后叫会给你做出来让你吃，你并不需要知道这个饭是怎么错的，你只需要面向这个服务员，告诉他你要吃什么，然后他也只需要面向你吃完收到钱就好，不需要知道你怎么对这个饭进行吃。

1、特点：

1：将复杂的事情简单化。

       2：面向对象将以前的过程中的执行者，变成了指挥者。

       3：面向对象这种思想是符合现在人们思考习惯的一种思想。

2、面向对象的三大特征：封装，继承、多态

1.封装：只隐藏对象的属性和实现细节，仅对外提供公共访问方式

好处：将变化隔离、便于使用、提高复用性、提高安全性

原则：将不需要对外提供的内容隐藏起来；把属性隐藏，提供公共方法对其访问

2.继承：提高代码复用性；继承是多态的前提

注：

①子类中所有的构造函数都会默认访问父类中的空参数的构造函数，默认第一行有super()；若无空参数构造函数，子类中需指定；另外，子类构造函数中可自己用this指定自身的其他构造函数。

3.多态

是父类或接口定义的引用变量可以指向子类或具体实现类的实例对象

好处：提高了程序的扩展性

弊端：当父类引用指向子类对象时，虽提高了扩展性，但只能访问父类中具备的方法，不可访问子类中的方法；即访问的局限性。

前提：实现或继承关系；覆写父类方法。

### 熟练使用集合、IO流及多线程

##### 一、集合：

1、特点：存储对象；长度可变；存储对象的类型可不同；

2、集合框架：

2）Collection

###### （1）List：有序的；元素可重复，有索引

（add(index, element)、add(index, Collection)、remove(index)、set(index,element)、get(index)、subList(from, to)、listIterator()）

①ArrayList：底层是数组结构，查询快，增删慢，不同步。

②LinkedList：底层是链表结构，增删快，查询慢，不同步

addFist();addLast()  getFirst();getLast()

removeFirst();removeLast() 获取并删除元素，无元素将抛异常：NoSuchElementException

替代的方法(JDK1.6)：

offerFirst();offerLast();

peekFirst();peekLast();无元素返回null

pollFirst();pollLast();删除并返回此元素，无元素返回null

③Vector：底层是数组结构，线程同步，被ArrayList取代了

注：了对于判断是否存在，以及删除等操作，以依赖的方法是元素的hashCode和equals方法

ArrayList判断是否存在和删除操作依赖的是equals方法

###### （2）Set：无序的，无索引，元素不可重复

①HashSet：底层是哈希表，线程不同步，无序、高效

保证元素唯一性：通过元素的hashCode和equals方法。若hashCode值相同，则会判断equals的结果是否为true；hashCode不同，不会调用equals方法

LinkedHashSet：有序，是HashSet的子类

②TreeSet：底层是二叉树，可对元素进行排序，默认是自然顺序

       保证唯一性：Comparable接口的compareTo方法的返回值

===》TreeSet两种排序方式：两种方式都存在时，以比较器为主

第一种：自然排序（默认排序）：

       添加的对象需要实现Comparable接口，覆盖compareTo方法

第二种：比较器

       添加的元素自身不具备比较性或不是想要的比较方式。将比较器作为参数传递进去。

       定义一个类，实现Comparator接口，覆盖compare方法。当主要条件相同时，比较次要条件。

###### 3）Map集合：

（1）HashTable：底层数据结构是哈希表，不可存入null键和null值。同步的

       Properties继承自HashTable，可保存在流中或从流中加载，是集合和IO流的结合产物

（2）HashMap：底层数据结构是哈希表；允许使用null键和null值，不同步，效率高

       TreeMap：

              底层数据结构时二叉树，不同步，可排序

              与Set很像，Set底层就是使用了Map集合

方法：

V put(K key, V value) ;  void putAll(Map m)

void clear();  V remove(Object key)

boolean containsKey(Object key);  containsValue(Object key);  isEmpty()

V get(Object key); int size(); Collection<V> values()

Set<K> keySet();  Set<Map.Entry<K,V>> entrySet()

###### 2.3、Map集合两种取出方式：

第一种：Set<K> keySet()

       取出Map集合中的所有键放于Set集合中，然后再通过键取出对应的值

Set<String> keySet = map.keySet();

Iterator<String> it = keySet.iterator();

while(it.hasNext()){

       String key = it.next();

       String value = map.get(key);

//…..

}

第二种：Set<Map.Entry<K,V>> entrySet()

       取出Map集合中键值对的映射放于Set集合中，然后通过Map集合中的内部接口，然后通过其中的方法取出

Set<Map.Entry<String,String>> entrySet = map.entrySet();

Iterator<Map.Entry<String,String>> it = entrySet.iterator();

While(it.hasNext()){

       Map.Entry<String,String> entry = it.next();

       String key = entry.getKey();

       String value = entry.getValue();

       //……

}

###### 2.4、Collection和Map的区别：

Collection：单列集合，一次存一个元素

Map：双列集合，一次存一对集合，两个元素（对象）存在着映射关系

###### 2.5、集合工具类：

Collections：操作集合（一般是list集合）的工具类。方法全为静态的

sort(List list);对list集合进行排序; sort(List list, Comparator c) 按指定比较器排序

fill(List list, T obj);将集合元素替换为指定对象；

swap(List list, int I, int j)交换集合指定位置的元素

shuffle(List list); 随机对集合元素排序

reverseOrder() ：返回比较器，强行逆转实现Comparable接口的对象自然顺序

reverseOrder(Comparator c)：返回比较器，强行逆转指定比较器的顺序

###### 2.6、Collection和Collections的区别：

Collections：java.util下的工具类，实现对集合的查找、排序、替换、线程安全化等操作。

Collection：是java.util下的接口，是各种单列集合的父接口，实现此接口的有List和Set集合，存储对象并对其进行操作。

3、Arrays：

       用于操作数组对象的工具类，全为静态方法

asList()：将数组转为list集合

       好处：可通过list集合的方法操作数组中的元素：

isEmpty()、contains()、indexOf()、set()

       弊端：数组长度固定，不可使用集合的增删操作。

如果数组中存储的是基本数据类型，asList会将数组整体作为一个元素存入集合

集合转为数组：Collection.toArray()；

       好处：限定了对集合中的元素进行增删操作，只需获取元素

##### 二、IO流

###### 1、结构：

字节流：InputStream，OutputStream

字符流：Reader，Writer

Reader：读取字符流的抽象类

       BufferedReader：将字符存入缓冲区，再读取

              LineNumberReader：带行号的字符缓冲输入流

       InputStreamReader：转换流，字节流和字符流的桥梁，多在编码的地方使用

              FileReader：读取字符文件的便捷类。

Writer：写入字符流的抽象类

       BufferedWriter：将字符存入缓冲区，再写入

       OutputStreamWriter：转换流，字节流和字符流的桥梁，多在编码的地方使用

              FileWriter：写入字符文件的便捷类。

InputStream：字节输入流的所有类的超类

       ByteArrayInputStream：含缓冲数组，读取内存中字节数组的数据，未涉及流

       FileInputStream：从文件中获取输入字节。媒体文件

              BufferedInputStream：带有缓冲区的字节输入流

              DataInputStream：数据输入流，读取基本数据类型的数据

       ObjectInputStream：用于读取对象的输入流

       PipedInputStream：管道流，线程间通信，与PipedOutputStream配合使用

       SequenceInputStream：合并流，将多个输入流逻辑串联。

OutputStream：此抽象类是表示输出字节流的所有类的超类

       ByteArrayOutputStream：含缓冲数组，将数据写入内存中的字节数组，未涉及流

       FileOutStream：文件输出流，将数据写入文件

              BufferedOutputStream：带有缓冲区的字节输出流

              PrintStream：打印流，作为输出打印

              DataOutputStream：数据输出流，写入基本数据类型的数据

       ObjectOutputStream：用于写入对象的输出流

       PipedOutputStream：管道流，线程间通信，与PipedInputStream配合使用

###### 2、流操作规律：

       明确源和目的：

              数据源：读取，InputStream和Reader

              目的：写入：OutStream和Writer

       数据是否是纯文本：

              是：字符流，Reader，Writer

              否：字节流，InputStream，OutStream

       明确数据设备：

              源设备：内存、硬盘、键盘

              目的设备：内存、硬盘、控制台

       是否提高效率：用BufferedXXX

3、转换流：将字节转换为字符，可通过相应的编码表获得

       转换流都涉及到字节流和编码表

##### 三、多线程

--à进程和线程：

1）进程是静态的，其实就是指开启的一个程序；而线程是动态的，是真正执行的单元，执行的过程。其实我们平时看到的进程，是线程在执行着，因为线程是作为进程的一个单元存在的。

2）同样作为基本的执行单元，线程是划分得比进程更小的执行单位。

3）每个进程都有一段专用的内存区域。与此相反，线程却共享内存单元（包括代码和数据），通过共享的内存单元来实现数据交换、实时通信与必要的同步操作。

###### 1、创建线程的方式：

创建方式一：继承Thread

    1：定义一个类继承Thread

    2：覆盖Thread中的run方法（将线程运行的代码放入run方法中）。

    3：直接创建Thread的子类对象

    4：调用start方法（内部调用了线程的任务（run方法））；作用：启动线程，调用run方法

方式二：实现Runnable

    1：定义类实现Runnable接口

    2：覆盖Runnable接口中的run方法，将线程的任务代码封装到run中

    3：通过Thread类创建线程对象

4、并将Runnable接口的子类对象作为Thread类的构造函数参数进行传递

作为参数传递的原因是让线程对象明确要运行的run方法所属的对象。

区别：

       继承方式：线程代码放在Thread子类的run方法中

       实现方式：线程存放在接口的子类run方法中；避免了单继承的局限性，建议使用。

###### 2、线程状态：

新建：start()

临时状态：具备cpu的执行资格，但是无执行权

运行状态：具备CPU的执行权，可执行

冻结状态：通过sleep或者wait使线程不具备执行资格，需要notify唤醒，并处于临时状态。

消亡状态：run方法结束或者中断了线程，使得线程死亡。

###### 3、多线程安全问题：

多个线程共享同一数据，当某一线程执行多条语句时，其他线程也执行进来，导致数据在某一语句上被多次修改，执行到下一语句时，导致错误数据的产生。

因素：多个线程操作共享数据；多条语句操作同一数据

解决：

       原理：某一时间只让某一线程执行完操作共享数据的所有语句。

       办法：使用锁机制：synchronized或lock对象

###### 4、线程的同步：

当两个或两个以上的线程需要共享资源，他们需要某种方法来确定资源在某一刻仅被一个线程占用，达到此目的的过程叫做同步（synchronization）。

同步代码块：synchronized(对象){}，将需要同步的代码放在大括号中，括号中的对象即为锁。

同步函数：放于函数上，修饰符之后，返回类型之前。

###### 5、wait和sleep的区别：（执行权和锁区分）

wait：可指定等待的时间，不指定须由notify或notifyAll唤醒。

       线程会释放执行权，且释放锁。

sleep：必须制定睡眠的时间，时间到了自动处于临时（阻塞）状态。

       即使睡眠了，仍持有锁，不会释放执行权。

##### Android下 的进程与线程：

1、进程的生命周期：

1）、进程的创建及回收：

       进程是被系统创建的，当内存不足的时候，又会被系统回收

2）、进程的级别：

Foreground Process       前台进程

Visible Process              可视进程

Service Process             服务进程：可以提高级别的

Background Process       后台进程

Empty Process              空进程（无组件启动，做进程缓存使用，恢复速度快）

☆  Android技能

### 熟练掌握Android四大组件，常用的布局文件，自定义控件等

Android中4大组件是：ContentProvider、Activity、BroadcastReceiver和Service

清单文件：

1、所有的应用程序必须要有清单文件

在manifest节点下需要声明当前应用程序的包名

2、包名：声明包的名字，必须唯一

       如果两个应用程序的包名和签名都相同，后安装的会覆盖先安装的

3、声明的程序的组件（4大组件）

       其中比较特殊的是广播接收者，可以不在清单文件中配置，可以通过代码进行注册

4、声明程序需要的权限：保护用户的隐私

5、可以控制服务在单独的进程中的，四大组件都可以配置这个属性process

在组件节点配置process：

       如：android:process="xxx.ooo.xxx"

比如说：处理图片的时候，会很耗内存，就需要在单独的新的进程中，可以减少内存溢出的几率

##### 一、ContentProvider 内容提供者

###### 1、特点

①、可以将应用中的数据对外进行共享；

②、数据访问方式统一，不必针对不同数据类型采取不同的访问策略；

③、内容提供者将数据封装，只暴露出我们希望提供给其他程序的数据（这点有点类似Javabeans）；

④、内容提供者中数据更改可被监听；

###### 2、创建内容提供者

定义类继承ContentProvider，根据需要重写其内容方法(6个方法)：

onCreate()                                    创建内容提供者时，会调用这个方法，完成一些初始化操作；

crud相应的4个方法              用于对外提供CRUD操作；

getType()                                      返回当前Url所代表数据的MIME类型：

返回的是单条记录：以vnd.android.cursor.item/ 开头，如：vnd.android.cursor.item/person

返回的是多条记录：以vnd.android.cursor.dir/ 开头，如：vnd.android.cursor.dir/person

在清单文件的<application>节点下进行配置，<provider>标签中需要指定name、authorities、exported属性

name：             为全类名；

authorities：   是访问Provider时的路径，要唯一；

exported：      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

URI代表要操作的数据，由scheme、authorites、path三部分组成：

content://com.itheima.sqlite.provider/person

scheme：         固定为content，代表访问内容提供者；

authorites：    <provider>节点中的authorites属性；

path：               程序定义的路径，可根据业务逻辑定义；

操作 URI的UriMather与ContentUris工具类：

       当程序调用CRUD方法时会传入Uri

UriMatcher：表示URI匹配器，可用于添加Uri匹配模式，与匹配Uri（见下代码）；

ContentUris：用于操作Uri路径后面的ID部分,2个重要的方法：

withAppendedId(uri, id)  为路径加上ID部分；

parseId(uri) 用于从路径中获取ID部分；

示例代码（内容提供者类）：

public class HeimaProvider extends ContentProvider {

       private static final int PERSON = 1;                   // 匹配码

       private static final int STUDENT = 2;                // 匹配码

       private static final int PERSON\_ID = 3;             // 匹配码

       private MyHelper helper;

       /\*\* Uri匹配器 \*/

       private UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

       @Override

       public boolean onCreate() {

              System.out.println("onCreate...");

              helper = new MyHelper(getContext());

              // == 添加 uri 匹配模式, 设置匹配码（参数3） Uri如果匹配就会返回相应的匹配码 ==

              uriMatcher.addURI("com.itheima.sqlite.provider", "person", PERSON);

              uriMatcher.addURI("com.itheima.sqlite.provider", "#", PERSON\_ID);                          // #表示匹配数字，\*表示匹配文本

              uriMatcher.addURI("com.itheima.sqlite.provider", "student", STUDENT);

              return true;

       }

       @Override

       public Uri insert(Uri uri, ContentValues values) {

              SQLiteDatabase db = helper.getWritableDatabase();

              switch (uriMatcher.match(uri)) {                                      // 匹配uri

              case PERSON:

                     long id = db.insert("person", "id", values);

                     db.close();

                     return ContentUris.withAppendedId(uri, id);                     // 在原uri上拼上id，生成新的uri并返回；

              case STUDENT:

                     long insert = db.insert("student", "id", values);

                     System.out.println("数据文件中，没有student表，也不会报错");

                     db.close();

                     return ContentUris.withAppendedId(uri, insert);         // 为路径上，加上ID

              default:

                     throw new IllegalArgumentException(String.format("Uri：%s 不是合法的uri地址", uri));

              }

       }

       @Override

       public int delete(Uri uri, String selection, String[] selectionArgs) {

              SQLiteDatabase db = helper.getWritableDatabase();

              switch (uriMatcher.match(uri)) {                                             // 匹配uri

              case PERSON\_ID:

                     long parseId = ContentUris.parseId(uri);                           // 获取传过来的ID值

                     selection = "id=?";                                                          // 设置查询条件

                     selectionArgs = new String[] { parseId + "" };                  // 查询条件值

              case PERSON:

                     int delete = db.delete("person", selection, selectionArgs);

                     db.close();

                     return delete;

              default:

                     throw new IllegalArgumentException(String.format("Uri：%s 不是合法的uri地址", uri));

              }

       }

       @Override

       public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {

              SQLiteDatabase db = helper.getWritableDatabase();

              switch (uriMatcher.match(uri)) {

              case PERSON\_ID:

                     long parseId = ContentUris.parseId(uri);                           // 获取传过来的ID值

                     selection = "id=?";                                                          // 设置查询条件

                     selectionArgs = new String[] { parseId + "" };                  // 查询条件值

              case PERSON:

                     int update = db.update("person", values, selection, selectionArgs);

                     db.close();

                     return update;

              default:

                     throw new IllegalArgumentException(String.format("Uri：%s 不是合法的uri地址", uri));

              }

       }

       @Override

       public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {

              SQLiteDatabase db = helper.getWritableDatabase();

              switch (uriMatcher.match(uri)) {

              case PERSON\_ID:

                     // == 根据ID查询 ==

                     long parseId = ContentUris.parseId(uri);                           // 获取传过来的ID值

                     selection = "id=?";                                                          // 设置查询条件

                     selectionArgs = new String[] { parseId + "" };                  // 查询条件值

              case PERSON:

                     Cursor cursor = db.query("person", projection, selection, selectionArgs, null, null, sortOrder);

                     // == 注意：此处的 db与cursor不能关闭 ==

                     return cursor;

              default:

                     throw new IllegalArgumentException(String.format("Uri：%s 不是合法的uri地址", uri));

              }

       }

       // 返回传入URI的类型，可用于测试URI是否正确

       @Override

       public String getType(Uri uri) {

              switch (uriMatcher.match(uri)) {

              case PERSON\_ID:

                     return "vnd.android.cursor.item/person";             // 表示单条person记录

              case PERSON:

                     return "vnd.android.cursor.dir/person";        // 表单多个person记录

              default:

                     return null;

              }

       }

}

清单中的配置：

 <provider

     android:exported="true"

     android:name="com.itheima.sqlite.provider.HeimaProvider"

     android:authorities="com.itheima.sqlite.provider" />

authorities 可以配置成如下形式（系统联系人的）：

       android:authorities="contacts;com.android.contacts"

“;” 表示的是可使用 contacts, 与 com.android.contacts

###### 3、内容解析者ContentResolver

通过Context获得ContentResolver内容访问者对象（内容提供者的解析器对象）；

调用ContentResolver对象的方法即可访问内容提供者

测试类代码：

public class HeimaProviderTest extends AndroidTestCase {

       /\*\* 测试添加数据 \*/

       public void testInsert() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/person");

              ContentValues values = new ContentValues();

              values.put("name", "小翼");

              values.put("balance", 13000);

              Uri insert = resolver.insert(uri, values);               // 获取返回的uri，如：content://com.itheima.sqlite.provider/7

              System.out.println(insert);

       }

       /\*\* 测试删除 \*/

       public void testRemove() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/person");

              int count = resolver.delete(uri, "id=?", new String[] { 3 + "" });

              System.out.println("删除了" + count + "行");

       }

       /\*\* 测试更新 \*/

       public void testUpdate() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/person");

              ContentValues values = new ContentValues();

              values.put("name", "小赵 update");

              values.put("balance", 56789);

              int update = resolver.update(uri, values, "id=?", new String[] { 6 + "" });

              System.out.println("更新了" + update + "行");

       }

       /\*\* 测试查询 \*/

       public void testQueryOne() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/person");

              Cursor c = resolver.query(uri, new String[] { "name", "balance" }, "id=?", new String[] { 101 + "" }, null);

              if (c.moveToNext()) {

                     System.out.print(c.getString(0));

                     System.out.println(" " + c.getInt(1));

              }

              c.close();

       }

       /\*\*测试查询全部 \*/

       public void testQueryAll() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/person");

              Cursor c = resolver.query(uri, new String[] { "id", "name", "balance" }, null, null, "name desc");

              while (c.moveToNext()) {

                     System.out.println(c.getInt(0) + ", " + c.getString(1) + ", " + c.getInt(2));

              }

              c.close();

       }

       /\*\* 测试查询一条 \*/

       public void testQueryOneWithUriId() {

              ContentResolver resolver = this.getContext().getContentResolver();

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider/3");          // 查询ID为3的记录

              Cursor c = resolver.query(uri, new String[] { "id", "name", "balance" }, null, null, null);

              if (c.moveToNext()) {

                     System.out.println(c.getInt(0) + ", " + c.getString(1) + ", " + c.getInt(2));

              }

              c.close();

       }

       /\*\* 测试获取内容提供者的返回类型 \*/

       public void testGetType() {

              ContentResolver resolver = this.getContext().getContentResolver();

              System.out.println(resolver.getType(Uri.parse("content://com.itheima.sqlite.provider/2")));

              System.out.println(resolver.getType(Uri.parse("content://com.itheima.sqlite.provider/person")));

       }

}

###### 4、监听内容提供者的数据变化

在内容提供者中可以通知其他程序数据发生变化

通过Context的getContentResolver()方法获取ContentResolver

调用其notifyChange()方法发送数据修改通知，发送到系统的公共内存（消息信箱中）

在其他程序中可以通过ContentObserver监听数据变化

通过Context的getContentResolver()方法获取ContentResolver

调用其registerContentObserver()方法指定对某个Uri注册ContentObserver

自定义ContentObserver，重写onChange()方法获取数据

示例代码（发通知部分）：

       public int delete(Uri uri, String selection, String[] selectionArgs) {

              SQLiteDatabase db = helper.getWritableDatabase();

                     int delete = db.delete("person", selection, selectionArgs);

                     // == 通过内容访问者对象ContentResolve 发通知给所有的Observer ==

                     getContext().getContentResolver().notifyChange(uri, null);

                     db.close();

                     return delete;

              }

       }

监听部分：

       // 注册内容观察者事件

       private void initRegisterContentObserver() {

              Uri uri = Uri.parse("content://com.itheima.sqlite.provider");      // 监听的URI

              // == 第2个参数：true表示监听的uri的后代都可以监听到 ==

              getContentResolver().registerContentObserver(uri, true, new ContentObserver(new Handler()) {

                     public void onChange(boolean selfChange) {                    // 接到通知就执行

                            personList = personDao.queryAll();

                            ((BaseAdapter) personListView.getAdapter()).notifyDataSetChanged();

                     }

              });

       }

###### 5、区别Provider/Resolver/Observer

1）ContentProvider：内容提供者

       把一个应用程序的私有数据（如数据库）信息暴露给别的应用程序，让别的应用程序可以访问；

       在数据库中有对应的增删改查的方法，如果要让别的应用程序访问，需要有一个路径uri：

通过content:// 路径对外暴露，uri写法：content://主机名/表名

2）ContentResolver：内容解析者

       根据内容提供者的路径，对数据进行操作（crud）；

3）ContentObserver：内容观察者

       可以理解成android系统包装好的回调，数据发送变化时，会执行回调中的方法；

       ContentResolver发送通知，ContentObserver监听通知；

当A的数据发生变化的时候，A就会显示的通知一个内容观察者，不指定观察者，就会发消息给一个路径

##### 二、Activity活动

描述：

1）表示用户交互的一个界面（活动），每一个activity对应一个界面

2）是所有View的容器：button,textview,imageview；我们在界面上看到的都是一个个的view

3）有个ActivityManager的管理服务类，用于维护与管理Activity的启动与销毁；

Activity启动时，会把Activity的引用放入任务栈中

4）一个应用程序可以被别的应用程序的activity开启

       此时，是将此应用程序的引用加入到了开启的那个activity的任务栈中了

5） activity是运行在自己的程序进程里面的

       在一个应用程序中，可以申请单独的进程，然此应用程序中的一个组件在新的进程中运行

6）可以在activity里面添加permission标签,调用者必须加入这个权限

       与钱打交道的界面，都不允许被其他应用程序随意打开

如果觉得那个activity比较重要，可以在清单文件中配置，防止别人随意打开，需要配置一个权限

自定义权限：

在清单文件中配置permission，创建一个新的权限

       创建后，就会在清单文件中生成这个权限了

此时，需要开启这个界面，就需要使用这个权限

Tips：

       \*不可使用中文文本，需要使用字符串，抽取出来

\*声明之后，会在gen的目录下，多出来一个文件：Manifest的文件，系统也存在一个这样的文件

###### 1、创建Activity

1）定义类继承自Activity类；

2）在清单文件中Application节点中声明<activity>节点；

       <activity

            android:name="com.itheima.activity.MainActivity"

            android:label="@string/app\_name" >

            <!-- 程序的入口，LAUNCHER表示桌面快捷方式，进入的是此Activity -->

            <intent-filter>

                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />  <!—启动时，默认匹配 --

            </intent-filter>

        </activity>

###### 2、启动Activity

通过意图（Intent）来启动一个Activity；

显示启动：

显示启动一般用于自己调用自己的情况（在当前应用找），这样的启动方式比较快速，创建Intent后指定包名和类名；

       Intent intent = new Intent(this, OtherActivity.class);

       startActivity(intent);             // 启动新的Activity

       或者：

       Intent intent = new Intent();

       intent.setClassName("com.itheima.activity", "com.itheima.activity.OtherActivity"); // 包名、全类名

       startActivity(intent);             // 启动新的Activity

2）隐式启动：

一般用于调用别人的Activity，创建Intent后指定动作和数据以及类型；

       // 电话

       Intent intent = new Intent();

       intent.setAction(Intent.ACTION\_CALL);                                        // 设置动作

       intent.setData(Uri.parse("tel://123456"));                                 // 设置数据

       // 网页

       intent.setAction(Intent.ACTION\_VIEW);

       intent.setData(Uri.parse("http://192.168.1.45:8080/androidWeb"));

       // 音频/视频，设置type

       intent.setAction(Intent.ACTION\_VIEW);

       intent.setDataAndType(Uri.parse("file:///mnt/sdcard/daqin.mp3"), "audio/\*");  // 设置数据和数据类型，将启动音频播放器（vedio）

3）为隐式启动配置意图过滤器：

显式意图是指在创建意图时指定了组件，而隐式意图则不指定组件，通过动作、类型、数据匹配对应的组件；

在清单文件中定义<activity>时需要定义<intent-filter>才能被隐式意图启动；

<intent-filter>中至少配置一个<action>和一个<category>，否则无法被启动；

Intent对象中设置的action、category、data在<intent-filter>必须全部包含Activity才能启动；

<intent-filter>中的<action>、<category>、<data>都可以配置多个，Intent对象中不用全部匹配，每样匹配一个即可启动；

如果一个意图可以匹配多个Activity，Android系统会提示选择；

         <!-- 注册 Activity， lable 表示Activity的标题 -->

        <activity

            android:name="com.itheima.activity.OtherActivity"

            android:label="OtherActivity" >

            <!-- 配置隐式意图，匹配http -->

            <intent-filter>

                <action android:name="android.intent.action.VIEW" />           <!—必须，表示动作为View -->

                <data android:scheme="http" />                                             <!—http开头-->

                <category android:name="android.intent.category.DEFAULT" /> <!-- 必须，表示启动时，默认匹配 -->

            </intent-filter>

            <!-- 匹配tel -->

             <intent-filter>

                <action android:name="android.intent.action.CALL" />

                <data android:scheme="tel" />

                <category android:name="android.intent.category.DEFAULT" /> <!-- 必须，表示启动 -->

            </intent-filter>

            <!-- 匹配 音频、视频 -->

            <intent-filter>

                <action android:name="android.intent.action.VIEW" />

                <data android:scheme="file" android:mimeType="audio/\*" />   <!—文件协议l类型 -->

                <data android:scheme="file" android:mimeType="video/\*" />

                <category android:name="android.intent.category.DEFAULT" /> <!-- 必须，表示启动 -->

            </intent-filter>

     </activity>

###### 3、启动时传递数据

可通过意图Intent对象实现Activity之间的数据传递；

使用Intent.putExtra()方法装入一些数据, 被启动的Activity可在 onCreate方法中getIntent()获取；

可传输的数据类型: a.基本数据类型(数组),  b. String(数组),  c. Bundle(Map),  d. Serializable(Bean), e.Parcelable（放在内存一个共享空间里）;

基本类型：

       Intent intent = new Intent(this, OtherActivity.class);

       intent.putExtra("name", "张飞");         // 携带数据

       intent.putExtra("age", 12);

       startActivity(intent);

一捆数据：

       Intent intent = new Intent(this, OtherActivity.class);

       Bundle b1 = new Bundle();

       b1.putString("name", "赵云");

       b1.putInt("age", 25);

       Bundle b2 = new Bundle();

       b2.putString("name", "关羽");

       b2.putInt("age", 44);

       intent.putExtra("b1", b1);

       intent.putExtra("b2", b2);

序列化对象(须实现序列化接口)：

       Intent intent = new Intent(this, OtherActivity.class);

       Person p = new Person("张辽", 44);

       intent.putExtra("p", p);

接收数据：

       在OtherActivity 的onCreate()方法，通过 getIntent().get 相关的数据的方法来获取数据；

###### 4、关闭时返回数据

基本流程：

使用startActivityForResult(Intent intent, int requestCode) 方法打开Activity；

重写onActivityResult(int requestCode, int resultCode, Intent data) 方法；

新Activity中调用setResult(int resultCode, Intent data) 设置返回数据之后，关闭Activity就会调用上面的onActivityResult方法；

注意：新的Activity的启动模式不能设置成 singleTask（如果已创建，会使用以前创建的）与singleInstance（单例，单独的任务栈），

         不能被摧毁（执行不到finish方法），父Activity中的 onActivityResult方法将不会执行；

finish()：表示关闭当前Activity，会调用onDestroy方法；

Activity\_A:

       public void openActivityB(View v) {

              Intent intent = new Intent(this, Activity\_B.class);

              Person p = new Person("张辽", 44);

              intent.putExtra("p", p);

              startActivityForResult(intent, 100);                                         // 此方法，启动新的Activity，等待返回结果, 结果一旦返回，自动执行onActivityResult()方法

       }

       protected void onActivityResult(int requestCode, int resultCode, Intent data) {

              if(data == null) {                                                           // 没有数据，不执行

                     return;

              }

              System.out.println(requestCode + ", " + resultCode);         // code 可用来区分，哪里返回的数据

              String name = data.getStringExtra("name");

              int age = data.getIntExtra("age", -1);

       }

Activity\_B:

       public void close(View v) {

              // == 关闭当前Activity时，设置返回的数据 ==

              Intent intent = new Intent();

              intent.putExtra("name", "典韦");

              intent.putExtra("age", 55);

              setResult(200, intent);

              finish();                        // 关闭，类似于点击了后退

       }

###### 5、生命周期

1）Acitivity三种状态

运行：activity在最前端运行;

停止：activity不可见，完全被覆盖;

暂停：activity可见，但前端还有其他activity<>，注意：在当前Activitiiy弹出的对话框是Activity的一部分，弹出时，不会执行onPause方法；

2）生命周期相关的方法（都是系统自动调用，都以 on 开头）：

onCreate：      创建时调用，或者程序在暂停、停止状态下被杀死之后重新打开时也会调用；

onStart：                   onCreate之后或者从停止状态恢复时调用；

onResume：   onStart之后或者从暂停状态恢复时调用，从停止状态恢复时由于调用onStart，也会调用onResume（界面获得焦点）；

onPause：       进入暂停、停止状态，或者销毁时会调用（界面失去焦点）；

onStop：          进入停止状态，或者销毁时会调用；

onDestroy：    销毁时调用；

onRestart：    从停止状态恢复时调用；

3）生命周期图解：

       应用启动时,执行onCreate onStart onResume，退出时执行：onPause onStop onDestroy;

###### 6、横竖屏切换与信息的保存恢复

切换横竖屏时，会自动查找layout-port 、layout-land中的布局文件，默认情况下，

切换时，将执行摧毁onPause onStop onDestroy，再重置加载新的布局onCreate onStart onResume；

切换时如果要保存数据, 可以重写: onSaveInstanceState();

恢复数据时, 重写: onRestoreInstanceState();

è固定横屏或竖屏:                                 android:screenOrientation="landscape"

è横竖屏切换, 不摧毁界面(程序继续执行) android:configChanges="orientation|keyboardHidden|screenSize"

保存信息状态的相关方法:

onSaveInstanceState：

在Activity被动的摧毁或停止的时候调用（如横竖屏切换,来电），用于保存运行数据，可以将数据存在在Bundle中；

onRestoreInstanceState：

该方法在Activity被重新绘制的时候调用，例如改变屏幕方向，onSavedInstanceState可为onSaveInstanceState保存的数据

###### 7、启动模式

1）任务栈的概念

问：一个手机里面有多少个任务栈？

答：一般情况下，有多少个应用正在运行，就对应开启多少个任务栈；

       一般情况下，每开启一个应用程序就会创建一个与之对应的任务栈；

       二般情况下，如launchMode为 singleInstance，就创建自己单独的任务栈；

2）任务栈的作用：

它是存放Activity的引用的，Activity不同的启动模式，对应不同的任务栈的存放；

可通过getTaskId()来获取任务栈的ID，如果前面的任务栈已经清空，新开的任务栈ID+1，是自动增长的；

3）启动模式：

在AndroidManifest.xml中的<activity>标签中可以配置android:launchMode属性，用来控制Actvity的启动模式；

在Android系统中我们创建的Acitivity是以栈的形式呈现的：

①、standard：默认的，每次调用startActivity()启动时都会创建一个新的Activity放在栈顶；

②、singleTop：启动Activity时，指定Activity不在任务栈栈顶就创建，如在栈顶，则不会创建，会调用onNewInstance()，复用已经存在的实例

③、singleTask：在任务栈里面只允许一个实例，如果启动的Activity不存在就创建，如果存在直接跳转到指定的Activity所在位置，

                     如：栈内有ABCD，D想创建A, 即A上的BCD相应的Activity将移除；

④、singleInstance：（单例）开启一个新的任务栈来存放这个Activity的实例，在整个手机操作系统里面只有一个该任务栈的实例存在，此模式开启的Activity是运行在自己单独的任务栈中的；

4）应用程序、进程、任务栈的区别

①、应用程序：

四大组件的集合

在清单文件中都放在application节点下

对于终端用户而言，会将其理解为activity

②、进程：

操作系统分配的独立的内存空间，一般情况下，一个应用程序会对应一个进程，特殊情况下，会有多个进程

一个应用程序会对应一个或多个进程

③、任务栈：task stack（back stack）后退栈

       记录用户的操作步骤，维护用户的操作体验，

       专门针对于activity而言的，只用于activity

       一般使用standard，其他情况用别的

5）启动模式的演示

1、创建两个activity，布局中设置两个按钮，分别开启两个activity

第一、standard启动模式的：开启几个就会在任务栈中存在几个任务

01和02都是存在于一个任务栈中的

第二、在清单文件中将02的启动模式改为singletop，

此时02处于栈顶，就只会创建一个02的任务，再开启02，也不会创建新的

第三、将02的启动模式改为singletask

       如果02上面有其他任务栈，就会将其他的清除掉，利用这个已经创建的02

       当开启02的时候，即先将01清除，然后利用下面的02

第四、将02的启动模式改为singleinstance

       可以通过打印任务栈的id（调用getTaskId()方法）得知，两个activity不在同一个任务栈中

若先开启三个01，在开启02，任务栈如图：

再开启01，任务栈的示意图如下：

此时按返回键，会先一层一层清空01，最后再清空02

空进程：任务栈清空，意味着程序退出了，但进程留着，这个就是空进程，容易被系统回收；

###### 8、内存管理

       Android系统在运行多个进程时，如果系统资源不足，会强制结束一些进程,优先选择哪个进程来结束是有优先级的。

会按照以下顺序杀死：

①、空：  进程中没有任何组件；

②、后台：进程中只有停止状态的Activity；

③、服务：进程中有正在运行的服务；

④、可见：进程中有一个暂停状态的Activity；

⑤、前台：进程中正在运行一个Activity；

Activity在退出的时候进程不会销毁, 会保留一个空进程方便以后启动. 但在内存不足时进程会被销毁；

Activity中不要在Activity做耗时的操作, 因为Activity切换到后台之后（Activity停止了）, 内存不足时, 也容易被销毁；

##### 三、BroadcastReceiver 广播接收者

系统的一些事件，比如来电，来短信，等等，会发广播；可监听这些广播，并进行一些处理；

Android3.2以后，为了安全起见，对于刚安装的应用，需要通过点击进入应用（界面，用户确认之后），接收者才能起作用；

以后即使没有启动其界面，也能接收到广播；

###### 1、定义广播接收者

1）定义类继承BroadcastReceiver，重写onReceive方法

2）清单文件中声明<receiver>，需要在其中配置<intent-filter>指定接收广播的类型；

3）当接收到匹配广播之后就会执行onReceive方法；

4）有序广播中，如果要控制多个接收者之间的顺序，可在<intent-filter>配置priority属性，系统默认为0，值越大，优先级越高；

5）BroadcastReceiver除了在清单文件中声明，也可以在代码中声明，使用registerReceiver方法注册Receiver；

 <!-- 配置广播接收者，监听播出电话 -->

     <receiver android:name="com.itheima.ipdialer.CallReceiver" >

          <intent-filter>

               <action android:name="android.intent.action.NEW\_OUTGOING\_CALL" />

           </intent-filter>

 </receiver>

###### 2、广播的分类

1）普通广播：

普通广播不可中断，不能互相传递数据；

2）有序广播：

广播可中断，通过调用abortBroadcast()方法；

接收者之间可以传递数据；

###### 3、广播接收者的注册方式

4大组件中，只有广播接收者是一个非常特殊的组件，其他3大组件都需要在清单文件中注册；

广播接收者，有2中注册方式：清单文件与代码方式，区别：

1）清单文件注册广播接收者，只要应用程序被部署到手机上，就立刻生效，不管进程是否处于运行状态；

2）代码方式，如果代码运行了，广播接收者才生效，如果代码运行结束，广播接收者，就失效；

这属于动态注册广播，临时用一下，用的时候，register，不用时unregister；

代码方式示例：

       // 广播接收者

       private class InnerReceiver extends BroadcastReceiver {

              @Override

              public void onReceive(Context context, Intent intent) {

                     String phone = getResultData();

                     String address = AddressDao.queryAddress(getApplicationContext(), phone);

                     showAddress(address);

              }

       }

       // 注册示例代码

       public void onCreate() {

              // == 服务启动时，注册广播接收者 ==

              innerReceiver = new InnerReceiver();

              // 指定意图过滤器

              IntentFilter filter = new IntentFilter(Intent.ACTION\_NEW\_OUTGOING\_CALL);

              this.registerReceiver(innerReceiver, filter);

       }

       // 销毁

       public void onDestroy() {

              // == 服务停止时，移除广播接收者 ==

              this.unregisterReceiver(innerReceiver);

              innerReceiver = null;

              super.onDestroy();

       }

###### 4、发送广播

1）发送普通广播

①、使用sendBroadcast()方法可发送普通广播；

②、通过Intent确定广播类型，可携带数据，所有接收者都可以接收到数据，数据不能被修改，不会中断；

接收者无序（试验测试，是按照安装顺序来接收的）;

③、广播时，可设置接收者权限，仅当接收者含有权限才能接收；

④、接收者的<receiver>也可设置发送方权限，只接受含有相应权限应用的广播；

发送者：

       Intent intent = new Intent("com.itheima.broadcast.TEST");       // 指定动作；接收者，需要配置 intent filter才能接受到此广播

       intent.setFlags(Intent.FLAG\_INCLUDE\_STOPPED\_PACKAGES);  // 包含未启动的过的应用（也可以收到广播），默认为不包含

       intent.putExtra("data", "这是来着广播发送者发来的贺电");           // 广播发送者intent中的数据，接收者，修改不了

       sendBroadcast(intent, null);                                                    // 发送无序广播，异步获取数据，不可中断，接收者之间不可传数据

接收者：

       public class AReceiver extends BroadcastReceiver {

              public void onReceive(Context context, Intent intent) {

                     System.out.println("AReceiver： " + intent.getStringExtra("data"));

              }

       }

      <receiver android:name="com.itheima.a.AReceiver">

            <intent-filter android:priority="2" >

                <action android:name="com.itheima.broadcast.TEST" />   <!—接收指定动作的广播 -->

            </intent-filter>

      </receiver>

注意：

       如果要在广播接收者中打开Activity，需要设置一下Intent.FLAG\_ACTIVITY\_NEW\_TASK，因为广播接收者是没有Activity任务栈的

所以需要加上这个标记，方能在广播接收者中打开Activity，如：

       public void onReceive(Context context, Intent intent) {

              Log.i(TAG, "打电话了。。。");

              String phone = this.getResultData();

              if ("2008".equals(phone)) {

                     // == 打开手机防盗功能界面 ==

                     Intent safeIntent = new Intent(context, LostFindActivity.class);

                     safeIntent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);                   // 使Activity也能在Receiver中启动

                     context.startActivity(safeIntent);

                     abortBroadcast();                                // 中断广播

                     setResultData(null);                            // 把电话号码设为null，就没有了通话记录

              }

       }

2）发送有序广播

sendOrderedBroadcast() 发送有序广播；

通过Intent确定广播类型, 携带数据，Intent的数据同样修改无效；

跟普通广播一样，也可以设置相应的权限；

接收者可在<intent-filter>定义android:priority定义优先级，数字越大，优先级越高；

有序广播会被接收者逐个接收，中途可以中断，或添加、修改数据；

可以指定一个自己的广播接收者, 这个接收者将最后一个收到广播、不会被中断、不需要任何权限、不需要配置；

可以指定一个Handler用来在自己的接收者中进行线程通信；

发送者：

       Intent intent = new Intent("com.itheima.broadcast.TEST");       // 指定动作；接收者，需要配置 intent filter才能接受到此广播

       intent.setFlags(Intent.FLAG\_INCLUDE\_STOPPED\_PACKAGES);  // 包含未启动的过的应用（也可以收到广播），默认为不包含

       intent.putExtra("data", "这是来着广播发送者发来的贺电");           // 广播发送者的intent中的数据，接收者，修改不了

       // == 有序广播时，传递的数据可修改 ==

       Bundle bundle = new Bundle();

       bundle.putString("name", "关羽");

       bundle.putInt("age", 22);

       /\* 定义权限，要求接收者，要有 com.itheima.permission.broadcast.RECEIVE 才能接收；

        \* 配置了最后接收者，Creceiver,无论你们怎么弄，我都可以收到广播，而且我不要配置，不要权限

       \*  handle为null,表示使用系统默认的

       \*  传递了数据 1， “MainActivity”, bundle 这些都是可以在接收者修改的

       \*/

       this.sendOrderedBroadcast(intent, "com.itheima.permission.broadcast.RECEIVE", new CReceiver(), null, 1, "MainActivity", bundle);

    <!-- 定义一个权限 -->

    <permission android:name="com.itheima.permission.broadcast.RECEIVE" >

</permission>

<!- 使用该权限-->

<uses-permission android:name="com.itheima.permission.broadcast.RECEIVE" />

###### 接收者AReceive：

       public void onReceive(Context context, Intent intent) {

              System.out.println("AReceiver： " + intent.getStringExtra("data"));

              Bundle bundle = this.getResultExtras(true);         // 设置为true，表示即使没有传递Bundle数据，不会出现空指针

              String message = String.format("%s : %s : %s, %s", getResultCode(), getResultData(), bundle.getString("name"), bundle.getInt("age"));

              System.out.println(message);                             // 如果优先级高于其他接收者，将打印发送者的数据

              // == 修改有序发送者，发来的数据 ==

              bundle.putString("name", "赵子龙");

              bundle.putInt("age", 222);

              this.setResult(2, "AReceiver", bundle);

              // == 修改Intent中的数据，无效 ==

              intent.putExtra("data", "AReceiver 修改了数据");

              this.setResultData("这是来自AReceiver的信息");

              // this.abortBroadcast();                                           // 中断，比它优先级低的接收者，将不能接收到广播了

       }

       <!-- 要求广播发送者必须有对应的权限，我才收 -->

      <receiver  android:name="com.itheima.a.AReceiver"

            android:permission="com.itheima.permission.broadcast.RECEIVE" >

            <intent-filter android:priority="2" >

                <action android:name="com.itheima.broadcast.TEST" />

            </intent-filter>

        </receiver>

接收者BReceive:代码及配置与上类似，只是优先级比A的低

###### 5、广播的生命周期

广播接收者的生命周期非常短暂的，在接收到广播的时候创建，onReceive()方法结束之后销毁；

广播接收者中不要做一些耗时的工作，否则会弹出Application No Response错误对话框；

最好也不要在广播接收者中创建子线程做耗时的工作，因为广播接收者被销毁后进程就成为了空进程，很容易被系统杀掉；

耗时的较长的工作最好放在服务中完成；

##### 四、Service服务

Service是一种在后台长期运行的，没有界面的组件，由其他组件调用开始运行；

服务不太会被kill，即使在内存不足时被kill，当内存恢复时，服务会自动复活，例如下载就可以放入服务中来做，下载时，启动服务，完成时，关闭服务；

###### 1、创建与使用Service

1）、定义类继承Service, 清单文件中声明<service>，同样也可以配置意图过滤；

2）、使用Intent来开启Service，在其他组件中调用startService方法;

3）、停止Service，调用stopService方法;

###### 2、生命周期

Service中的生命周期方法(Context调用执行)：

1）startService()             如果没创建就先onCreate()再startCommand(), 如果已创建就只执行startCommand();

2）stopService()              执行onDestroy()

3）bindService()              如果没有创建就先onCreate()再onBind()

4）unbindService()          如果服务是在绑定时启动的, 先执行onUnbind()再执行onDestroy(). 如果服务在绑定前已启动, 那么只执行onUnbind()；

###### 3、开启服务的2种方式

2种不同开启方式的区别：

1）startService：

       开启服务，服务一旦开启，就长期就后台运行，即使调用者退出来，服务还会长期运行；

       资源不足时，被杀死，资源足够时，又会复活；

2）bindService：

       绑定服务，绑定服务的生命周期会跟调用者关联起来，调用者退出，服务也会跟着销毁；

       通过绑定服务，可以间接的调用服务里面的方法（onBind返回IBinder）；

###### 4、服务混合调用生命周期

一般的调用顺序：

①、start  -> stop                    开启 –> 结束

②、bind  -> unbind                 绑定（服务开启） -> 解绑（服务结束）

混合调用：

①、start –> bind -> stop->unbind->ondestroy                通常不会使用这种模式

       开启 –> 绑定 –> 结束（服务停不了）->解除绑定（服务才可停掉）

②、start –> bind -> unbind -> stop                          经常使用

       开启 –> 绑定 –> 解绑（服务继续运行）-> stop（不用时，再停止服务）

         这样保证了服务长期后台运行，又可以调用服务中的方法

##### 五、Android四大组件

1.    ContentProvider

              共享应用程序内的数据, 在数据修改时可以监听

Activity

              供用户操作的界面

BroadcastReceiver

              用来接收广播, 可以根据系统发生的一些时间做出一些处理

Service

              长期在后台运行的, 没有界面的组件, 用来在后台执行一些耗时的操作

### 熟悉掌握ListView的优化及异步任务加载网络数据

##### 一、异步任务加载网络数据：

在Android中提供了一个异步任务的类AsyncTask，简单来说，这个类中的任务是运行在后台线程中的，并可以将结果放到UI线程中进行处理，它定义了三种泛型，分别是Params、Progress和Result，分别表示请求的参数、任务的进度和获得的结果数据。

1、使用原因：

1）是其中使用了线程池技术，而且其中的方法很容易实现调用

2）可以调用相关的方法，在开启子线程前和后，进行界面的更新

3）一旦任务多了，不用每次都new新的线程，可以直接使用

2、执行的顺序：

onPreExecute()【执行前开启】--- > doInBackground() --- > onProgressUpdate() --- > onPostExecute()

3、执行过程：

当一个异步任务开启后，执行过程如下：

1）、onPreExecute()：

这个方法是执行在主线程中的。这步操作是用于准备好任务的，作为任务加载的准备工作。建议在这个方法中弹出一个提示框。

2）、doInBackground()：

这个方法是执行在子线程中的。在onPreExecute()执行完后，会立即开启这个方法，在方法中可以执行耗时的操作。需要将请求的参数传递进来，发送给服务器，并将获取到的数据返回，数据会传给最后一步中；这些值都将被放到主线程中，也可以不断的传递给下一步的onProgressUpdate()中进行更新。可以通过不断调用publishProgress()，将数据（或进度）不断传递给onProgressUpdate()方法，进行不断更新界面。

3）、onProgressUpdate()：

这个方法是执行在主线程中的。publishProgress()在doInBackground()中被调用后，才开启的这个方法，它在何时被开启是不确定的，执行这个方法的过程中，doInBackground()是仍在执行的，即子线程还在运行着。

4）、onPostExecute()：

这个方法是执行在主线程中的。当后台的子线程执行完毕后才调用此方法。doInBackground()返回的结果会作为参数被传递过来。可以在这个方法中进行更新界面的操作。

5）、execute()：

       最后创建AsyncTask自定义的类，开启异步任务。

3、实现原理：

1）、线程池的创建：

在创建了AsyncTask的时候，会默认创建一个线程池ThreadPoolExecutor，并默认创建出5个线程放入到线程池中，最多可防128个线程；且这个线程池是公共的唯一一份。

、任务的执行：

在execute中，会执行run方法，当执行完run方法后，会调用scheduleNext()不断的从双端队列中轮询，获取下一个任务并继续放到一个子线程中执行，直到异步任务执行完毕。

3）、消息的处理：

在执行完onPreExecute()方法之后，执行了doInBackground()方法，然后就不断的发送请求获取数据；在这个AsyncTask中维护了一个InternalHandler的类，这个类是继承Handler的，获取的数据是通过handler进行处理和发送的。在其handleMessage方法中，将消息传递给onProgressUpdate()进行进度的更新，也就可以将结果发送到主线程中，进行界面的更新了。

4、需要注意的是：

①、这个AsyncTask类必须由子类调用

②、虽然是放在子线程中执行的操作，但是不建议做特别耗时的操作，如果操作过于耗时，建议使用线程池ThreadPoolExecutor和FutureTask

示例代码：

private class DownloadFilesTask extends AsyncTask&lt;URL, Integer, Long&gt; {

    protected Long doInBackground(URL... urls) {

        int count = urls.length;

        long totalSize = 0;

        for (int i = 0; i < count; i++) {

            totalSize += Downloader.downloadFile(urls[i]);

            publishProgress((int) ((i / (float) count) \* 100));

            // Escape early if cancel() is called

            if (isCancelled()) break;

        }

        return totalSize;

    }

    protected void onProgressUpdate(Integer... progress) {

        setProgressPercent(progress[0]);

    }

    protected void onPostExecute(Long result) {

        showDialog("Downloaded " + result + " bytes");

    }

}

new DownloadFilesTask().execute(url1, url2, url3);

##### 二、ListView优化：

ListView的工作原理

首先来了解一下ListView的工作原理（可参见http://mobile.51cto.com/abased-410889.htm），如图：

1、如果你有几千几万甚至更多的选项(item)时，其中只有可见的项目存在内存（内存内存哦，说的优化就是说在内存中的优化！！！）中，其他的在Recycler中

2、ListView先请求一个type1视图(getView)然后请求其他可见的项目。convertView在getView中是空(null)的

3、当item1滚出屏幕，并且一个新的项目从屏幕低端上来时，ListView再请求一个type1视图。convertView此时不是空值了，它的值是item1。你只需设定新的数据然后返回convertView，不必重新创建一个视图

一、复用convertView，减少findViewById的次数

1、优化一：复用convertView

Android系统本身为我们考虑了ListView的优化问题，在复写的Adapter的类中，比较重要的两个方法是getCount()和getView()。界面上有多少个条显示，就会调用多少次的getView()方法；因此如果在每次调用的时候，如果不进行优化，每次都会使用View.inflate(….)的方法，都要将xml文件解析，并显示到界面上，这是非常消耗资源的：因为有新的内容产生就会有旧的内容销毁，所以，可以复用旧的内容。

优化：

在getView()方法中，系统就为我们提供了一个复用view的历史缓存对象convertView，当显示第一屏的时候，每一个item都会新创建一个view对象，这些view都是可以被复用的；如果每次显示一个view都要创建一个，是非常耗费内存的；所以为了节约内存，可以在convertView不为null的时候，对其进行复用

2、优化二：缓存item条目的引用——ViewHolder

   findViewById()这个方法是比较耗性能的操作，因为这个方法要找到指定的布局文件，进行不断地解析每个节点：从最顶端的节点进行一层一层的解析查询，找到后在一层一层的返回，如果在左边没找到，就会接着解析右边，并进行相应的查询，直到找到位置（如图）。因此可以对findViewById进行优化处理，需要注意的是：

》》》》特点：xml文件被解析的时候，只要被创建出来了，其孩子的id就不会改变了。根据这个特点，可以将孩子id存入到指定的集合中，每次就可以直接取出集合中对应的元素就可以了。

优化：

在创建view对象的时候，减少布局文件转化成view对象的次数；即在创建view对象的时候，把所有孩子全部找到，并把孩子的引用给存起来

①定义存储控件引用的类ViewHolder

这里的ViewHolder类需要不需要定义成static，根据实际情况而定，如果item不是很多的话，可以使用，这样在初始化的时候，只加载一次，可以稍微得到一些优化

不过，如果item过多的话，建议不要使用。因为static是Java中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（比如Context的情况最多），这时就要尽量避免使用了。

   class ViewHolder{

                 //定义item中相应的控件

          }

②创建自定义的类：ViewHolder holder = null;

③将子view添加到holder中：

在创建新的listView的时候，创建新的ViewHolder，把所有孩子全部找到，并把孩子的引用给存起来

通过view.setTag(holder)将引用设置到view中

通过holder，将孩子view设置到此holder中，从而减少以后查询的次数

④在复用listView中的条目的时候，通过view.getTag()，将view对象转化为holder，即转化成相应的引用，方便在下次使用的时候存入集合。

  通过view.getTag(holder)获取引用（需要强转）

示例代码：

public class ActivityDemo extends Activity {

    private ListView listview1;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

           super.onCreate(savedInstanceState);

           listview1 = (ListView) findViewById(R.id.listview1);

           MyAdapter adapter = new MyAdapter();

           listview1.setAdapter(adapter);

    }

    private class MyAdapter extends BaseAdapter{

           @Override

           public int getCount() {

                  return 40;

           }

           @Override

           public Object getItem(int position) {

                  return position;

           }

           @Override

           public long getItemId(int position) {

                  return position;

           }

           @Override

           public View getView(int position, View convertView, ViewGroup parent) {

                  ViewHolder holder = null;

                  if(convertView!=null && convertView instanceof RelativeLayout){    //注意：这里不一定用RelativeLayout，根据XML文件中的根节点来确定

                         holder = (ViewHolder) convertView.getTag();

                  }else{

                         //1、复用历史缓存view对象，检索布局问转化成view对象的次数

                         convertView = View.inflate(ActivityDemo.this, R.layout.item, null);

                         //2、在创建view对象的时候，把所有的子view找到，把子view的引用存起来

                         holder = new ViewHolder();

                         holder.ivIcon = (ImageView) convertView.findViewById(R.id.iv\_icon);

                         holder.tvContent = (TextView) convertView.findViewById(R.id.tv\_content);

                         convertView.setTag(holder);

                         /\*    实现存储子view引用的另一种方式：

                                convertView.setTag(holder.ivIcon);

                                convertView.setTag(holder.tvContent);  \*/

                  }

                  //直接复用系统提供的历史缓存对象convertView

                  return convertView;

           }

    }

class ViewHolder{

           public ImageView ivIcon;

           public TextView tvContent;

    }

}

二、ListView中数据的分批及分页加载：

需求：

ListView有一万条数据，如何显示；如果将十万条数据加载到内存，很消耗内存

解决办法：

优化查询的数据：先获取几条数据显示到界面上

进行分批处理---à优化了用户体验

进行分页处理---à优化了内存空间

说明：

一般数据都是从数据库中获取的，实现分批（分页）加载数据，就需要在对应的DAO中有相应的分批（分页）获取数据的方法，如findPartDatas ()

1、准备数据：

   在dao中添加分批加载数据的方法：findPartDatas ()

   在适配数据的时候，先加载第一批的数据，需要加载第二批的时候，设置监听检测何时加载第二批

2、设置ListView的滚动监听器：setOnScrollListener(new OnScrollListener{….})

①、在监听器中有两个方法：滚动状态发生变化的方法(onScrollStateChanged)和listView被滚动时调用的方法(onScroll)

②、在滚动状态发生改变的方法中，有三种状态：

手指按下移动的状态：                 SCROLL\_STATE\_TOUCH\_SCROLL: // 触摸滑动

惯性滚动（滑翔（flgin）状态）：  SCROLL\_STATE\_FLING: // 滑翔

静止状态：                                   SCROLL\_STATE\_IDLE: // 静止

3、对不同的状态进行处理：

分批加载数据，只关心静止状态：关心最后一个可见的条目，如果最后一个可见条目就是数据适配器（集合）里的最后一个，此时可加载更多的数据。在每次加载的时候，计算出滚动的数量，当滚动的数量大于等于总数量的时候，可以提示用户无更多数据了。

示例代码:（详细代码请参见【6.3-ListView数据的分批加载.doc】）

// 给listview注册一个滚动的监听器.

lv\_call\_sms\_safe.setOnScrollListener(new OnScrollListener() {

   // 当滚动状体发生变化的时候调用的方法

   @Override

   public void onScrollStateChanged(AbsListView view, int scrollState) {

          switch (scrollState) {

          case SCROLL\_STATE\_FLING: // 滑翔

                 break;

          case SCROLL\_STATE\_IDLE: // 静止

                 // 在静止状态下 关心最后一个可见的条目 如果最后一个可见条目就是 数据适配器里面的最后一个 , 加载更多数据.

                 int position = lv\_call\_sms\_safe.getLastVisiblePosition(); // 位置从0开始

                 int size = blackNumbers.size();// 从1开始的.

                 if (position == (size - 1)) {

                        Log.i(TAG, "拖动到了最后一个条目,加载更多数据");

                        startIndex += maxNumber;

                        if(startIndex>=totalCount){

                               Toast.makeText(getApplicationContext(), "没有更多数据了..", 0).show();

                               return;

                        }

                        fillData();

                        break;

                 }

                 break;

          case SCROLL\_STATE\_TOUCH\_SCROLL: // 触摸滑动

                 break;

          }

   }

   // 当listview被滚动的时候 调用的方法

   @Override

   public void onScroll(AbsListView view, int firstVisibleItem, int visibleItemCount, int totalItemCount) {

   }

});

/\*\*

\* 填充数据

\*/

private void fillData() {

   // 通知用户一下正在获取数据

   ll\_loading.setVisibility(View.VISIBLE);

   new Thread() {

          public void run() {

                 // 获取全部的黑名单号码

                 if (blackNumbers != null) {

                        blackNumbers.addAll(dao.findPartBlackNumbers(startIndex,maxNumber));

                 } else {

                        blackNumbers = dao.findPartBlackNumbers(startIndex,maxNumber);

                 }

                 handler.sendEmptyMessage(0);

                 // lv\_call\_sms\_safe.setAdapter(new CallSmsSafeAdapter());

          };

   }.start();

}

##### 三、复杂ListView的处理：（待进一步总结）

说明：

   listView的界面显示是通过getCount和getView这两个方法来控制的

   getCount：返回有多少个条目

   getView：返回每个位置条目显示的内容

提供思路：

   对于含有多个类型的item的优化处理：由于ListView只有一个Adapter的入口，可以定义一个总的Adapter入口，存放各种类型的Adapter

以安全卫士中的进程管理的功能为例。效果如图：

1、定义两个（或多个）集合

   每个集合中存入的是对应不同类型的内容（这里为：用户程序（userAppinfos）和系统程序的集合（systemAppinfos））

2、在初始化数据（填充数据）中初始化两个集合

   如，此处是在fillData方法中初始化

3、在数据适配器中，复写对应的方法

   getCount()：计算所有需要显示的条目个数，这里包括listView和textView

   getView()：对显示在不同位置的条目进行if处理

4、数据类型的判断

   需要注意的是，在复用view的时候，需要对convertView进行类型判断，是因为这里含有各种不同类型的view，在view滚动显示的时候，对于不同类型的view不能复用，所有需要判断

示例代码：

获取条目个数

public int getCount() {

   // 用户程序个数 + 系统程序个数

   return userAppinfos.size() + 1 + systemAppinfos.size() + 1;

}

类型判断：

if (convertView != null && convertView instanceof RelativeLayout) {

   view = convertView;

   holder = (ViewHolder) view.getTag();

} else {

   //……..

}

getView中条目位置的选择：

if (position == 0) {// 显示一个textview的标签 , 告诉用户用户程序有多少个

          TextView tv = new TextView(getApplicationContext());

          tv.setBackgroundColor(Color.GRAY);

          tv.setTextColor(Color.WHITE);

          tv.setText("用户程序:" + userAppinfos.size() + "个");

          return tv;

   } else if (position == (userAppinfos.size() + 1)) {

          TextView tv = new TextView(getApplicationContext());

          tv.setBackgroundColor(Color.GRAY);

          tv.setTextColor(Color.WHITE);

          tv.setText("系统程序:" + systemAppinfos.size() + "个");

          return tv;

   } else if (position <= userAppinfos.size()) {// 用户程序

          appInfo = userAppinfos.get(position - 1);

   } else {// 系统程序

          appInfo = systemAppinfos.get(position - 1 - userAppinfos.size() - 1);

   }

##### 四、ListView中图片的优化：

1、处理图片的方式：

如果自定义Item中有涉及到图片等等的，一定要狠狠的处理图片，图片占的内存是ListView项中最恶心的，处理图片的方法大致有以下几种：

①、不要直接拿路径就去循环decodeFile();使用Option保存图片大小、不要加载图片到内存去

②、拿到的图片一定要经过边界压缩

③、在ListView中取图片时也不要直接拿个路径去取图片，而是以WeakReference（使用WeakReference代替强引用。

比如可以使用WeakReference mContextRef）、SoftReference、WeakHashMap等的来存储图片信息，是图片信息不是图片哦！

④、在getView中做图片转换时，产生的中间变量一定及时释放

2、异步加载图片基本思想：

（待进一步总结，详见曹睿新闻案例【E:\JAVA\SOURCE\TSource\lessons\Android\PROJECT\PhoneLottory\day06\Optimization】）

1）、 先从内存缓存中获取图片显示（内存缓冲）

2）、获取不到的话从SD卡里获取（SD卡缓冲）

3）、都获取不到的话从网络下载图片并保存到SD卡同时加入内存并显示（视情况看是否要显示）

原理：

优化一：先从内存中加载，没有则开启线程从SD卡或网络中获取，这里注意从SD卡获取图片是放在子线程里执行的，否则快速滑屏的话会不够流畅。

优化二：于此同时，在adapter里有个busy变量，表示listview是否处于滑动状态，如果是滑动状态则仅从内存中获取图片，没有的话无需再开启线程去外存或网络获取图片。

优化三：ImageLoader里的线程使用了线程池，从而避免了过多线程频繁创建和销毁，有的童鞋每次总是new一个线程去执行这是非常不可取的，好一点的用的AsyncTask类，其实内部也是用到了线程池。在从网络获取图片时，先是将其保存到sd卡，然后再加载到内存，这么做的好处是在加载到内存时可以做个压缩处理，以减少图片所占内存。

Tips：这里可能出现图片乱跳（错位）的问题：

图片错位问题的本质源于我们的listview使用了缓存convertView，假设一种场景，一个listview一屏显示九个item，那么在拉出第十个item的时候，事实上该item是重复使用了第一个item，也就是说在第一个item从网络中下载图片并最终要显示的时候，其实该item已经不在当前显示区域内了，此时显示的后果将可能在第十个item上输出图像，这就导致了图片错位的问题。所以解决之道在于可见则显示，不可见则不显示。在ImageLoader里有个imageViews的map对象，就是用于保存当前显示区域图像对应的url集，在显示前判断处理一下即可。

Adapter示例代码：

public class LoaderAdapter extends BaseAdapter{

        private static final String TAG = "LoaderAdapter";

        private boolean mBusy = false;             //是否处于滑动中

        public void setFlagBusy(boolean busy) {

                this.mBusy = busy;

        }

        private ImageLoader mImageLoader;

        private int mCount;

        private Context mContext;

        private String[] urlArrays;

        public LoaderAdapter(int count, Context context, String []url) {

                this.mCount = count;

                this.mContext = context;

                urlArrays = url;

                mImageLoader = new ImageLoader(context);

        }

        public ImageLoader getImageLoader(){

                return mImageLoader;

        }

        @Override

        public int getCount() {

                return mCount;

        }

         @Override

        public Object getItem(int position) {

                return position;

        }

         @Override

        public long getItemId(int position) {

                return position;

        }

        @Override

        public View getView(int position, View convertView, ViewGroup parent) {

                 ViewHolder viewHolder = null;

                if (convertView == null) {  //加载新创建的view

                        convertView = LayoutInflater.from(mContext).inflate(R.layout.list\_item, null);

                        viewHolder = new ViewHolder();

                        viewHolder.mTextView = (TextView) convertView.findViewById(R.id.tv\_tips);

                        viewHolder.mImageView = (ImageView) convertView.findViewById(R.id.iv\_image);

                        convertView.setTag(viewHolder);

                } else {

                        viewHolder = (ViewHolder) convertView.getTag();

                }

                String url = "";

                url = urlArrays[position % urlArrays.length];

viewHolder.mImageView.setImageResource(R.drawable.ic\_launcher);

                if (!mBusy) {

                        mImageLoader.DisplayImage(url, viewHolder.mImageView, false);

                        viewHolder.mTextView.setText("--" + position + "--IDLE ||TOUCH\_SCROLL");

                } else {

                        mImageLoader.DisplayImage(url, viewHolder.mImageView, true);

                        viewHolder.mTextView.setText("--" + position + "--FLING");

                }

//复用历史缓存view

                return convertView;

        }

        static class ViewHolder {

                TextView mTextView;

                ImageView mImageView;

        }

}

3、内存缓冲机制：

首先限制内存图片缓冲的堆内存大小，每次有图片往缓存里加时判断是否超过限制大小，超过的话就从中取出最少使用的图片并将其移除。

当然这里如果不采用这种方式，换做软引用也是可行的，二者目的皆是最大程度的利用已存在于内存中的图片缓存，避免重复制造垃圾增加GC负担；OOM溢出往往皆因内存瞬时大量增加而垃圾回收不及时造成的。只不过二者区别在于LinkedHashMap里的图片缓存在没有移除出去之前是不会被GC回收的，而SoftReference里的图片缓存在没有其他引用保存时随时都会被GC回收。所以在使用LinkedHashMap这种LRU算法缓存更有利于图片的有效命中，当然二者配合使用的话效果更佳，即从LinkedHashMap里移除出的缓存放到SoftReference里，这就是内存的二级缓存。

本例采用的是LRU算法，先看看MemoryCache的实现

public class MemoryCache {

        private static final String TAG = "MemoryCache";

        // 放入缓存时是个同步操作

        // LinkedHashMap构造方法的最后一个参数true代表这个map里的元素将按照最近使用次数由少到多排列，即LRU

        // 这样的好处是如果要将缓存中的元素替换，则先遍历出最近最少使用的元素来替换以提高效率

        private Map<String, Bitmap> cache = Collections

                        .synchronizedMap(new LinkedHashMap<String, Bitmap>(10, 1.5f, true));

        // 缓存中图片所占用的字节，初始0，将通过此变量严格控制缓存所占用的堆内存

        private long size = 0;// current allocated size

        // 缓存只能占用的最大堆内存

        private long limit = 1000000;// max memory in bytes

         public MemoryCache() {

                // use 25% of available heap size

                setLimit(Runtime.getRuntime().maxMemory() / 10);

        }

        public void setLimit(long new\_limit) {

                limit = new\_limit;

                Log.i(TAG, "MemoryCache will use up to " + limit / 1024. / 1024. + "MB");

        }

        public Bitmap get(String id) {

                try {

                        if (!cache.containsKey(id))

                                return null;

                        return cache.get(id);

                } catch (NullPointerException ex) {

                        return null;

                }

        }

        public void put(String id, Bitmap bitmap) {

                try {

                        if (cache.containsKey(id))

                                size -= getSizeInBytes(cache.get(id));

                        cache.put(id, bitmap);

                        size += getSizeInBytes(bitmap);

                        checkSize();

                } catch (Throwable th) {

                        th.printStackTrace();

                }

        }

         /\*\*

         \* 严格控制堆内存，如果超过将首先替换最近最少使用的那个图片缓存

         \*

         \*/

        private void checkSize() {

                Log.i(TAG, "cache size=" + size + " length=" + cache.size());

                if (size > limit) {

                        // 先遍历最近最少使用的元素

                        Iterator<Entry<String, Bitmap>> iter = cache.entrySet().iterator();

                        while (iter.hasNext()) {

                                Entry<String, Bitmap> entry = iter.next();

                                size -= getSizeInBytes(entry.getValue());

                                iter.remove();

                                if (size <= limit)

                                        break;

                        }

                        Log.i(TAG, "Clean cache. New size " + cache.size());

                }

        }

        public void clear() {

          cache.clear();

        }

        /\*\*

         \* 图片占用的内存

             \* <a href="\"http://www.eoeandroid.com/home.php?mod=space&uid=2768922\"" target="\"\_blank\"">@Param</a> bitmap

            \* @return

         \*/

        long getSizeInBytes(Bitmap bitmap) {

                if (bitmap == null)

                        return 0;

                return bitmap.getRowBytes() \* bitmap.getHeight();

        }

}

##### 五、ListView的其他优化：

1、尽量避免在BaseAdapter中使用static 来定义全局静态变量：

static是Java中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（比如Context的情况最多），这时就要尽量避免使用了。

2、尽量使用getApplicationContext：

如果为了满足需求下必须使用Context的话：Context尽量使用Application Context，因为Application的Context的生命周期比较长，引用它不会出现内存泄露的问题

3、尽量避免在ListView适配器中使用线程：

因为线程产生内存泄露的主要原因在于线程生命周期的不可控制。之前使用的自定义ListView中适配数据时使用AsyncTask自行开启线程的，这个比用Thread更危险，因为Thread只有在run函数不 结束时才出现这种内存泄露问题，然而AsyncTask内部的实现机制是运用了线程执行池（ThreadPoolExcutor）,这个类产生的Thread对象的生命周期是不确定的，是应用程序无法控制的，因此如果AsyncTask作为Activity的内部类，就更容易出现内存泄露的问题。解决办法如下：

①、将线程的内部类，改为静态内部类。

②、在线程内部采用弱引用保存Context引用

示例代码：

public abstract class WeakAsyncTask extends  AsyncTask {

protected WeakReference mTarget;

public WeakAsyncTask(WeakTarget target) {

mTarget = new WeakReference(target);

}

@Override

protected final void onPreExecute() {

final WeakTarget target = mTarget.get();

if (target != null) {

this.onPreExecute(target);

}

}

   @Override

protected final Result doInBackground(Params... params) {

final WeakTarget target = mTarget.get();

if (target != null) {

return this.doInBackground(target, params);

} else {

return null;

}

}

@Override

protected final void onPostExecute(Result result) {

 final WeakTarget target = mTarget.get();

if (target != null) {

this.onPostExecute(target, result);

}

}

 protected void onPreExecute(WeakTarget target) {

// No default action

 }

   protected abstract Result doInBackground(WeakTarget target, Params... params);

protected void onPostExecute(WeakTarget target, Result result) {

 // No default action

}

 }

##### 六、ScrollView和ListView的冲突问题：【摘自网络】

解决方法之一：

在ScrollView添加一个ListView会导致listview控件显示不全，这是因为两个控件的滚动事件冲突导致。所以需要通过listview中的item数量去计算listview的显示高度，从而使其完整展示，如下提供一个方法供大家参考。

示例代码：

public void setListViewHeightBasedOnChildren(ListView listView) {

ListAdapter listAdapter = listView.getAdapter();

if (listAdapter == null) {

return;

}

int totalHeight = 0;

for (int i = 0; i < listAdapter.getCount(); i++) {

View listItem = listAdapter.getView(i, null, listView);

listItem.measure(0, 0);

totalHeight += listItem.getMeasuredHeight();

}

ViewGroup.LayoutParams params = listView.getLayoutParams();

params.height = totalHeight + (listView.getDividerHeight() \* (listAdapter.getCount() - 1));

params.height += 5;//if without this statement,the listview will be a little short

listView.setLayoutParams(params);

}

### 熟悉XML/JSON解析数据，以及数据存储方式

数据的存储方式包括：File、SheredPreferences、XML/JSON、数据库、网络

XML/JSON解析数据：

##### 一、XML解析

1.解析 \*\*\*\*\*

              获取解析器: Xml.newPullParser()

              设置输入流: setInput()

              获取当前事件类型: getEventType()

              解析下一个事件, 获取类型: next()

              获取标签名: getName()

              获取属性值: getAttributeValue()

              获取下一个文本: nextText()

              获取当前文本: getText()

              5种事件类型: START\_DOCUMENT, END\_DOCUMENT, START\_TAG, END\_TAG, TEXT

示例代码：

public List<Person> getPersons(InuptStream in){

       XmlPullParser parser=Xml.newPullParser();//获取解析器

       parser.setInput(in,"utf-8");

       for(int type=){   //循环解析

       }

}

2.生成 \*

              获取生成工具: Xml.newSerializer()

              设置输出流: setOutput()

              开始文档: startDocument()

              结束文档: endDocument()

              开始标签: startTag()

              结束标签: endTag()

              属性: attribute()

              文本: text()

示例代码：

XmlSerializer serial=Xml.newSerializer();//获取xml序列化工具

serial.setOuput(put,"utf-8");

serial.startDocument("utf-8",true);

serial.startTag(null,"persons");

for(Person p:persons){

       serial.startTag(null,"persons");

       serial.attribute(null,"id",p.getId().toString());

       serial.startTag(null,"name");

       serial.attribute(null,"name",p.getName().toString());

       serial.endTag(null,"name");

       serial.startTag(null,"age");

       serial.attribute(null,"age",p.getAge().toString());

       serial.endTag(null,"age");

       serial.endTag(null,"persons");

}

##### （二）JSON解析

1、JSON书写格式：

1）JSON的规则很简单：对象是一个无序的“‘名称/值’对”集合。

一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’对”之间使用“,”（逗号）分隔。

2）规则如下:

①映射用冒号（“：”）表示。名称:值

②并列的数据之间用逗号（“，”）分隔。名称1:值1,名称2:值2

③映射的集合（对象）用大括号（“{}”）表示。{名称1:值,名称2:值2}

④并列数据的集合（数组）用方括号(“[]”)表示。

  [

    {名称1:值,名称2:值2},

    {名称1:值,名称2:值2}

  ]

⑤元素值可具有的类型：string, number, object, array, true, false, null

2、举例：

1）JSON对象（键值对或键值对的集合）

例1、{ "name": "Obama"}

例2、{ "name": "Romney","age": 56}

例3、{ "city":{"name": "bj"},"weatherinfo":{"weather": "sunny"}}

例4、{

         "city":{"name": "北京",“city\_id”:"101010100"},

         "weatherinfo":{"weather": "sunny","temp":"29度"}

      }

2）JSON数组

例1、

[

   { "name": "张三", "age":22, "email": "zhangsan@qq.com" },

   { "name": "李四", "age":23, "email": "lisi@qq.com"},

   { "name": "王五", "age":24, "email": "wangwu@qq.com" }

]

例2、

{ "student":

   [

      { "name": "张三", "age":22, "email": "zhangsan@qq.com" },

      { "name": "李四", "age":23, "email": "lisi@qq.com"},

      { "name": "王五", "age":24, "email": "wangwu@qq.com" }

    ]

}

[ {

     title : "国家发改委：台湾降油价和大陆没可比性",

     description : "国家发改委副主任朱之鑫",

     image : "http://192.168.1.101/Web/img/a.jpg",

     comment : 163

},{

     title : "国家发改委：台湾降油价和大陆没可比性",

     description : "国家发改委副主任朱之鑫",

     image : "http://192.168.1.101/Web/img/b.jpg",

     comment : 0

}, {

     title : "国家发改委：台湾降油价和大陆没可比性",

     description : "国家发改委副主任朱之鑫",

     image : "http://192.168.1.101/Web/img/c.jpg",

     comment : 0

} ];

3、在Android中使用json

在Android中内置了JSON的解析API，在org.json包中包含了如下几个类：

       JSONArray、JSONObject、JSONStringer、JSONTokener和一个异常类JSONException

4、JSON解析：

解析步骤

1）、读取html文件源代码，获取一个json字符串

       InputStream in = conn.getInputStream();

       String jsonStr = DataUtil.Stream2String(in);//将流转换成字符串的工具类

2）、将字符串传入响应的JSON构造函数中

①、通过构造函数将json字符串转换成json对象

       JSONObject  jsonObject = new JSONObject(jsonStr)；

②、通过构造函数将json字符串转换成json数组：

JSONArray array = new JSONArray(jsonStr);

3）、解析出JSON中的数据信息：

①、从json对象中获取你所需要的键所对应的值

       JSONObject  json=jsonObject.getJSONObject("weatherinfo");

       String city = json.getString("city");

       String temp = json.getString("temp")

②、遍历JSON数组，获取数组中每一个json对象，同时可以获取json对象中键对应的值

       for (int i = 0; i < array.length(); i++) {

              JSONObject obj = array.getJSONObject(i);

              String title=obj.getString("title");

              String description=obj.getString("description");

       }

注意：

①json数组并非全是由json对象组成的数组

②json数组中的每一个元素数据类型可以不相同

如：[94043,90210]或者["zhangsan",24]类似于javascript中的数组

5、生成JSON对象和数组：

1）生成JSON：

方法1、创建一个map，通过构造方法将map转换成json对象

       Map<String, Object> map = new HashMap<String, Object>();

       map.put("name", "zhangsan");

       map.put("age", 24);

       JSONObject json = new JSONObject(map);

方法2、创建一个json对象，通过put方法添加数据

       JSONObject json=new JSONObject();

       json.put("name", "zhangsan");

       json.put("age", 24);

2）生成JSON数组：

方法1、创建一个list，通过构造方法将list转换成json对象

       Map<String, Object> map1 = new HashMap<String, Object>();

       map1.put("name", "zhangsan");

       map1.put("age", 24);

       Map<String, Object> map2 = new HashMap<String, Object>();

       map2.put("name", "lisi");

       map2.put("age", 25);

       List<Map<String, Object>> list=new ArrayList<Map<String,Object>>();

       list.add(map1);

       list.add(map2);

       JSONArray array=new JSONArray(list);

       System.out.println(array.toString());

### 精通Android下的Hendler机制，并能熟练使用

Message：消息；其中包含了消息ID，消息对象以及处理的数据等，由MessageQueue统一列队，终由Handler处理

Handler：处理者；负责Message发送消息及处理。Handler通过与Looper进行沟通，从而使用Handler时，需要实现handlerMessage(Message msg)方法来对特定的Message进行处理，例如更新UI等（主线程中才行）

MessageQueue：消息队列；用来存放Handler发送过来的消息，并按照FIFO（先入先出队列）规则执行。当然，存放Message并非实际意义的保存，而是将Message以链表的方式串联起来的，等Looper的抽取。

Looper：消息泵，不断从MessageQueue中抽取Message执行。因此，一个线程中的MessageQueue需要一个Looper进行管理。Looper是当前线程创建的时候产生的（UI Thread即主线程是系统帮忙创建的Looper，而如果在子线程中，需要手动在创建线程后立即创建Looper[调用Looper.prepare()方法]）。也就是说，会在当前线程上绑定一个Looper对象。

Thread：线程；负责调度消息循环，即消息循环的执行场所。

知识要点

一、说明

1、handler应该由处理消息的线程创建。

2、handler与创建它的线程相关联，而且也只与创建它的线程相关联。handler运行在创建它的线程中，所以，如果在handler中进行耗时的操作，会阻塞创建它的线程。

二、一些知识点

1、Android的线程分为有消息循环的线程和没有消息循环的线程，有消息循环的线程一般都会有一个Looper。主线程（UI线程）就是一个消息循环的线程。

2、获取looper：

Looper.myLooper();      //获得当前的Looper

Looper.getMainLooper () //获得UI线程的Lopper

3、Handle的初始化函数（构造函数），如果没有参数，那么他就默认使用的是当前的Looper，如果有Looper参数，就是用对应的线程的Looper。

4、如果一个线程中调用Looper.prepare()，那么系统就会自动的为该线程建立一个消息队列，然后调用 Looper.loop();之后就进入了消息循环，这个之后就可以发消息、取消息、和处理消息。

### 消息处理机制原理：

一、大致流程：

在创建Activity之前，当系统启动的时候，先加载ActivityThread这个类，在这个类中的main函数，调用了Looper.prepareMainLooper();方法进行初始化Looper对象；然后创建了主线程的handler对象（Tips：加载ActivityThread的时候，其内部的Handler对象[静态的]还未创建）；随后才创建了ActivityThread对象；最后调用了Looper.loop();方法，不断的进行轮询消息队列的消息。也就是说，在ActivityThread和Activity创建之前（同样也是Handler创建之前，当然handler由于这两者初始化），就已经开启了Looper的loop()方法，不断的进行轮询消息。需要注意的是，这个轮询的方法是阻塞式的，没有消息就一直等待（实际是等着MessageQueue的next()方法返回消息）。在应用一执行的时候，就已经开启了Looper，并初始化了Handler对象。此时，系统的某些组件或者其他的一些活动等发送了系统级别的消息，这个时候主线程中的Looper就可以进行轮询消息，并调用msg.target.dispatchMessage(msg)（msg.target即为handler）进行分发消息，并通过handler的handleMessage方法进行处理；所以会优于我们自己创建的handler中的消息而处理系统消息。

0、准备数据和对象：

①、如果在主线程中处理message（即创建handler对象），那么如上所述，系统的Looper已经准备好了（当然，MessageQueue也初始化了），且其轮询方法loop已经开启。【系统的Handler准备好了，是用于处理系统的消息】。【Tips：如果是子线程中创建handler，就需要显式的调用Looper的方法prepare()和loop()，初始化Looper和开启轮询器】

②、通过Message.obtain()准备消息数据（实际是从消息池中取出的消息）

③、创建Handler对象，在其构造函数中，获取到Looper对象、MessageQueue对象（从Looper中获取的），并将handler作为message的标签设置到msg.target上

1、发送消息：sendMessage()：通过Handler将消息发送给消息队列

2、给Message贴上handler的标签：在发送消息的时候，为handler发送的message贴上当前handler的标签

3、开启HandlerThread线程，执行run方法。

4、在HandlerThread类的run方法中开启轮询器进行轮询：调用Looper.loop()方法进行轮询消息队列的消息

【Tips：这两步需要再斟酌，个人认为这个类是自己手动创建的一个线程类，Looper的开启在上面已经详细说明了，这里是说自己手动创建线程（HandlerThread）的时候，才会在这个线程中进行Looper的轮询的】

5、在消息队列MessageQueue中enqueueMessage(Message msg, long when)方法里，对消息进行入列，即依据传入的时间进行消息入列（排队）

6、轮询消息：与此同时，Looper在不断的轮询消息队列

7、在Looper.loop()方法中，获取到MessageQueue对象后，从中取出消息（Message msg = queue.next()）

8、分发消息：从消息队列中取出消息后，调用msg.target.dispatchMessage(msg);进行分发消息

9、将处理好的消息分发给指定的handler处理，即调用了handler的dispatchMessage(msg)方法进行分发消息。

10、在创建handler时，复写的handleMessage方法中进行消息的处理

11、回收消息：在消息使用完毕后，在Looper.loop()方法中调用msg.recycle()，将消息进行回收，即将消息的所有字段恢复为初始状态

测试代码：

/\*\*

 \* Handler 构造函数测试

 \* @author zhaoyu 2013-10-5 上午9:56:38

 \*/

public class HandlerConstructorTest extends Activity {

       private Handler handler1 = new Handler(new Callback() {

              @Override

              public boolean handleMessage(Message msg) {

                     System.out.println("使用了Handler1中的接口Callback");

                     return false;           // 此处，如果返回 false，下面的 handlerMessage方法会执行，true ，下面的不执行

              }

       });

       private Handler handler2 = new Handler() {

              public void handleMessage(Message msg) {

                     System.out.println("Handler2");

              }

       };

       protected void onCreate(Bundle savedInstanceState) {

              super.onCreate(savedInstanceState);

              //消息1

Message obtain1 = Message.obtain();

              obtain1.obj = "sendMessage";

              obtain1.what = 1;

              handler1.sendMessage(obtain1);

              //消息2

              Message obtain2 = handler2.obtainMessage();

              handler2.sendMessage(obtain2);    //①

//            handler2.dispatchMessage(obtain2);      //②

       }

}

二、详细解释：

1、准备Looper对象

两种情况初始化Looper对象：

1）在主线程中不需要显式的创建Looper对象，直接创建Handler对象即可；因为在主线程ActivityThread的main函数中已经自动调用了创建Looper的方法：Looper.prepareMainLooper();，并在最后调用了Looper.loop()方法进行轮询。

2）如果在子线程中创建Handler对象，需要创建Looper对象，即调用显式的调用Looper.prepare()

初始化Looper的工作：

1）初始化Looper对象：通过调用Looper.prepare()初始化Looper对象，在这个方法中，新创建了Looper对象

2）将Looper绑定到当前线程：在初始化中，调用sThreadLocal.set(new Looper(quitAllowed))方法，将其和ThreadLocal进行绑定

在ThreadLocal对象中的set方法，是将当前线程和Looper绑定到一起：首先获取到当前的线程，并获取线程内部类Values，通过Thread.Values的put方法，将当前线程和Looper对象进行绑定到一起。即将传入的Looper对象挂载到当前线程上。

Tips：在Looper对象中，可以通过getThread()方法，获取到当前线程，即此Looper绑定的线程对象。

源代码：

Looper中：

public static void prepare() {

        prepare(true);

    }

    private static void prepare(boolean quitAllowed) {

        if (sThreadLocal.get() != null) {

            throw new RuntimeException("Only one Looper may be created per thread");

        }

        sThreadLocal.set(new Looper(quitAllowed));

    }

ThreadLocal中：

public void set(T value) {

        Thread currentThread = Thread.currentThread();

        Values values = values(currentThread);

        if (values == null) {

            values = initializeValues(currentThread);

        }

        values.put(this, value);

    }

2、创建消息Message：

消息的创建可以通过两种方式：

1）new Message()

2）Message.obtain()：【当存在多个handler的时候，可以通过Message.obtain(Handler handler)创建消息，指定处理的handler对象】

Tips：建议使用第二种方式更好一些。原因：

       因为通过第一种方式，每有一个新消息，都要进行new一个Message对象，这会创建出多个Message，很占内存。

       而如果通过obtain的方法，是从消息池sPool中取出消息。每次调用obtain()方法的时候，先判断消息池是否有消息（if (sPool != null)），没有则创建新消息对象，有则从消息池中取出消息，并将取出的消息从池中移除【具体看obtain()方法】

public static Message obtain() {

        synchronized (sPoolSync) {

            if (sPool != null) {

                Message m = sPool;

                sPool = m.next;

                m.next = null;

                sPoolSize--;

                return m;

            }

        }

        return new Message();

    }

public Message() {

    }

3、创建Handler对象

两种形式创建Handler对象：

1）创建无参构造函数的Handler对象：

2）创建指定Looper对象的Handler对象

最终都会调用相应的含有Callback和boolean类型的参数的构造函数

【这里的Callback是控制是否分发消息的，其中含有一个返回值为boolean的handleMessage(Message msg)方法进行判断的；

  boolean类型的是参数是判断是否进行异步处理，这个参数默认是系统处理的，我们无需关心】

在这个构造函数中，进行了一系列的初始化工作：

①、获取到当前线程中的Looper对象

②、通过Looper对象，获取到消息队列MessageQueue对象

③、获取Callback回调对象

④、获取异步处理的标记

源代码：

①、创建无参构造函数的Handler对象：

public Handler() {

        this(null, false);

    }

public Handler(Callback callback, boolean async) {

        if (FIND\_POTENTIAL\_LEAKS) {

            final Class<? extends Handler> klass = getClass();

            if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) && (klass.getModifiers() & Modifier.STATIC) == 0) {

                Log.w(TAG, "The following Handler class should be static or leaks might occur: " +klass.getCanonicalName());

            }

        }

        mLooper = Looper.myLooper();

        if (mLooper == null) {

            throw new RuntimeException("Can't create handler inside thread that has not called Looper.prepare()");

        }

        mQueue = mLooper.mQueue;

        mCallback = callback;

        mAsynchronous = async;

    }

②、创建指定Looper对象的Handler对象

public Handler(Looper looper) {

        this(looper, null, false);

    }

public Handler(Looper looper, Callback callback, boolean async) {

        mLooper = looper;

        mQueue = looper.mQueue;

        mCallback = callback;

        mAsynchronous = async;

    }

4、Handler对象发送消息：

1）Handler发送消息给消息队列：

Handler对象通过调用sendMessage(Message msg)方法，最终将消息发送给消息队列进行处理

这个方法（所有重载的sendMessage）最终调用的是enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis)

（1）先拿到消息队列：在调用到sendMessageAtTime(Message msg, long uptimeMillis)方法的时候，获取到消息队列（在创建Handler对象时获取到的）

（2）当消息队列不为null的时候（为空直接返回false，告知调用者处理消息失败），再调用处理消息入列的方法：

enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis)

这个方法，做了三件事：

①、为消息打上标签：msg.target = this;：将当前的handler对象这个标签贴到传入的message对象上，为Message指定处理者

②、异步处理消息：msg.setAsynchronous(true);，在asyn为true的时候设置

③、将消息传递给消息队列MessageQueue进行处理：queue.enqueueMessage(msg, uptimeMillis);

public final boolean sendMessage(Message msg){

        return sendMessageDelayed(msg, 0);

    }

public final boolean sendMessageDelayed(Message msg, long delayMillis){

        if (delayMillis < 0) {

            delayMillis = 0;

        }

        return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

    }

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

        MessageQueue queue = mQueue;

        if (queue == null) {

            RuntimeException e = new RuntimeException(

                    this + " sendMessageAtTime() called with no mQueue");

            Log.w("Looper", e.getMessage(), e);

            return false;

        }

        return enqueueMessage(queue, msg, uptimeMillis);

    }

private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {

        msg.target = this;

        if (mAsynchronous) {

            msg.setAsynchronous(true);

        }

        return queue.enqueueMessage(msg, uptimeMillis);

    }

2）MessageQueue消息队列处理消息：

在其中的enqueueMessage(Message msg, long when)方法中，工作如下：

在消息未被处理且handler对象不为null的时候，进行如下操作（同步代码块中执行）

①、将传入的处理消息的时间when（即为上面的uptimeMillis）赋值为当前消息的when属性。

②、将next()方法中处理好的消息赋值给新的消息引用：Message p = mMessages;

       在next()方法中：不断的从消息池中取出消息，赋值给mMessage，当没有消息发来的时候，Looper的loop()方法由于是阻塞式的，就一直等消息传进来

③、当传入的时间为0，且next()方法中取出的消息为null的时候，将传入的消息msg入列，排列在消息队列上，此时为消息是先进先出的

       否则，进入到死循环中，不断的将消息入列，根据消息的时刻（when）来排列发送过来的消息，此时消息是按时间的先后进行排列在消息队列上的

final boolean enqueueMessage(Message msg, long when) {

        if (msg.isInUse()) {

            throw new AndroidRuntimeException(msg + " This message is already in use.");

        }

        if (msg.target == null) {

            throw new AndroidRuntimeException("Message must have a target.");

        }

        boolean needWake;

        synchronized (this) {

            if (mQuiting) {

                RuntimeException e = new RuntimeException(msg.target + " sending message to a Handler on a dead thread");

                Log.w("MessageQueue", e.getMessage(), e);

                return false;

            }

            msg.when = when;

            Message p = mMessages;

            if (p == null || when == 0 || when < p.when) {

                // New head, wake up the event queue if blocked.

                msg.next = p;

                mMessages = msg;

                needWake = mBlocked;

            } else {

                needWake = mBlocked && p.target == null && msg.isAsynchronous();

                Message prev;

                for (;;) {

                    prev = p;

                    p = p.next;

                    if (p == null || when < p.when) {

                        break;

                    }

                    if (needWake && p.isAsynchronous()) {

                        needWake = false;

                    }

                }

                msg.next = p; // invariant: p == prev.next

                prev.next = msg;

            }

        }

        if (needWake) {

            nativeWake(mPtr);

        }

        return true;

    }

5、轮询Message

1）开启loop轮询消息

当开启线程的时候，执行run方法，在HandlerThread类中，调用的run方法中将开启loop进行轮询消息队列：

在loop方法中，先拿到MessageQueue对象，然后死循环不断从队列中取出消息，当消息不为null的时候，通过handler分发消息：msg.target.dispatchMessage(msg)。消息分发完之后，调用msg.recycle()回收消息，

2）回收消息：

在Message的回收消息recycle()这个方法中：首先调用clearForRecycle()方法，将消息的所有字段都恢复到原始状态【如flags=0，what=0，obj=null，when=0等等】

然后在同步代码块中将消息放回到消息池sPool中，重新利用Message对象

源代码：

Looper.loop()

public static void loop() {

        final Looper me = myLooper();

        if (me == null) {

            throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");

        }

        final MessageQueue queue = me.mQueue;

        Binder.clearCallingIdentity();

        final long ident = Binder.clearCallingIdentity();

        for (;;) {

            Message msg = queue.next(); // might block

            if (msg == null) {

               return;

            }

            // This must be in a local variable, in case a UI event sets the logger

            Printer logging = me.mLogging;

            if (logging != null) {

                logging.println(">>>>> Dispatching to " + msg.target + " " +

                        msg.callback + ": " + msg.what);

            }

            msg.target.dispatchMessage(msg);

            if (logging != null) {

                logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);

            }

            final long newIdent = Binder.clearCallingIdentity();

            if (ident != newIdent) {

                Log.wtf(TAG, “……”);

            }

            msg.recycle();

        }

    }

msg.recycle();：

public void recycle() {

        clearForRecycle();

        synchronized (sPoolSync) {

            if (sPoolSize < MAX\_POOL\_SIZE) {

                next = sPool;

                sPool = this;

                sPoolSize++;

            }

        }

    }

/\*package\*/ void clearForRecycle() {

        flags = 0;

        what = 0;

        arg1 = 0;

        arg2 = 0;

        obj = null;

        replyTo = null;

        when = 0;

        target = null;

        callback = null;

        data = null;

    }

6、处理Message

在Looper.loop()方法中调用了msg.target.dispatchMessage(msg);的方法，就是调用了Handler中的dispatchMessage(Message msg)方法：

1）依据Callback中的handleMessage(msg)的真假判断是否要处理消息，如果是真则不进行消息分发，则不处理消息，否则进行处理消息

2）当Callback为null或其handleMessage(msg)的返回值为false的时候，进行分发消息，即调用handleMessage(msg)处理消息【这个方法需要自己复写】

/\*\*

     \* Subclasses must implement this to receive messages.

     \*/

    public void handleMessage(Message msg) {

    }

    /\*\*

     \* Handle system messages here.

     \*/

    public void dispatchMessage(Message msg) {

        if (msg.callback != null) {

            handleCallback(msg);

        } else {

            if (mCallback != null) {

                if (mCallback.handleMessage(msg)) {

                    return;

                }

            }

            handleMessage(msg);

        }

    }

==========

场景一：

在主线程中创建Handler，其中复写了handlerMessager方法（处理message，更新界面）

然后创建子线程，其中创建Message对象，并设置消息，通过handler发送消息

示例代码：

public class MainActivity2 extends Activity implements OnClickListener{

    private Button bt\_send;

       private TextView tv\_recieve;

       private Handler handler = new Handler(){

              @Override

              public void handleMessage(Message msg) {

                     super.handleMessage(msg);

                     tv\_recieve.setText((String) msg.obj);

              }

       };

       @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_main);

        bt\_send = (Button) findViewById(R.id.bt\_send);

        tv\_recieve = (TextView) findViewById(R.id.tv\_recieve);

        bt\_send.setOnClickListener(this);

        tv\_recieve.setOnClickListener(this);

       }

       @Override

       public void onClick(View v) {

              switch (v.getId()) {

              case R.id.bt\_send:

                     new Thread(){

                            public void run() {

                                   Message msg = new Message();

                                   msg.obj = "消息来了"+ System.currentTimeMillis();

                                   handler.sendMessage(msg);

                            }

                     }.start();

                     break;

              }

       }

}

执行过程：

1、Looper.prepare()

在当前线程(主线程)中准备一个Looper对象，即轮询消息队列MessageQueue的对象；此方法会创建一个Looper，在Looper的构造函数中，初始化的创建了一个MessageQueue对象(用于存放消息)，并准备好了一个线程供调用

2、new Hnader():

在当前线程中创建出Handler，需要复写其中的handleMessage(Message msg)，对消息进行处理（更新UI）。在创建Handler中，会将Looper设置给handler，并随带着MessageQueue对象；其中Looper是通过调用其静态方法myLooper()，返回的是ThreadLocal中的currentThread，并准备好了MessageQueue【mQueue】

3、Looper.loop():

无限循环，对消息队列进行不断的轮询，如果没有获取到消息，就会结束循环；如果有消息，直接从消息队列中取出消息，并通过调用msg.target.dispatchMessage(msg)进行分发消息给各个控件进行处理。

[其中的msg.target实际就是handler]。

4、创建子线程，handler.sendMessage(msg)

在handler.sendMessage(msg)方法中，实际上最终调用sendMessageAtTime(Message msg，long uptimeMillis)方法[sendMessageXXX方法都是最终调用的sendMessageAtTime方法]；此方法返回的enqueueMessage(queue，msg，uptimeMillis)，实际上返回的是MessageQueue中的enqueueMessage(msg，uptimeMillis)，其中进行的操作时对存入的消息进行列队，即根据接收到的消息的时间先后进行排列[使用的单链形式]；然后将消息就都存入到了消息队列中，等待着handler进行处理。

场景二：

创建两个子线程，一个线程中创建Handler并进行处理消息，另一个线程使用handler发送消息。

示例代码：

public class MainActivity extends Activity implements OnClickListener{

   private Button bt\_send;

       private TextView tv\_recieve;

       private Handler handler;

       @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_main);

        bt\_send = (Button) findViewById(R.id.bt\_send);

        tv\_recieve = (TextView) findViewById(R.id.tv\_recieve);

        bt\_send.setOnClickListener(this);

        tv\_recieve.setOnClickListener(this);

        new Thread(){

               public void run() {

                      //Looper.prepare();

                      handler = new Handler(Looper.getMainLooper()){

                             @Override

                             public void handleMessage(Message msg) {

                                    super.handleMessage(msg);

                                    tv\_recieve.setText((String) msg.obj);

                             }

                      };

                      //Looper.loop();

               }

        }.start();

    }

       @Override

       public void onClick(View v) {

              switch (v.getId()) {

              case R.id.bt\_send:

                     new Thread(){

                            public void run() {

                                   Message msg = new Message();

                                   msg.obj = "消息来了"+ System.currentTimeMillis();

                                   handler.sendMessage(msg);

                            }

                     }.start();

                     break;

              }

       }

}

简单说明执行过程:

说明：在子线程中是不能更新界面的操作的，只能放在主线程中进行更新。所以必须将处理的消息放到主线程中，才能进行更新界面，否则会报错

1、子线程中创建Handler，并处理消息

1）创建Handler：

源码如下：

public Handler(Looper looper, Callback callback, boolean async) {

  mLooper = looper;

  mQueue = looper.mQueue;

  mCallback = callback;

  mAsynchronous = async;

}

这个构造函数做了一下几步工作：

①、创建轮询器：

由于新创建的子线程中没有轮询器，就需要创建一个轮询器，才能进行消息的轮询处理。传入的是主线程的轮询器，就已经将这个looper绑定到主线程上了【传入哪个线程的Looper，就绑定在哪个线程上】

②、将消息队列加入到轮询器上。

消息队列MessageQueue是存放handler发来的消息的，等着Looper进行轮询获取；在一个线程中的MessageQueue需要一个Looper进行管理，所以两者需要同在一个线程中。

③、回调和异步加载。（此处不做分析[其实我还没分析好]）

需要注意的是界面更新：

上面说到了，在子线程中是不可以进行更新界面的操作的，这就需要使用带有轮询器参数的handler构造函数进行创建，传入主线程的轮询器：Looper.getMainLooper()，从而将消息加入到主线程的消息队列之中。因此就可进行在handleMessage方法中进行处理消息更新界面了。

2）、消息处理：

复写其中的handleMessage(Message msg)，对消息进行处理（更新UI）。在创建Handler中，会将Looper设置给handler，并随带着MessageQueue对象；其中Looper是通过调用其静态方法myLooper()，返回的是ThreadLocal中的currentThread，并准备好了MessageQueue【mQueue】

虽然是在子线程中编写的代码，但是由于传入的是主线程的looper，所以，Looper从MessageQueue队列中轮询获取消息、再进行更新界面的操作都是在主线程中执行的。

3）、Looper.loop():

说明：由于传入的是主线程的Looper，而在主线程中已经有这一步操作了，所以这里就不需要进行显示的调用了。但是主线程在这个时候是做了这个轮询的操作的。

无限循环，对消息队列进行不断的轮询，如果没有获取到消息，就会结束循环；如果有消息，直接从消息队列中取出消息，并通过调用msg.target.dispatchMessage(msg)进行分发消息给各个控件进行处理。

[其中的msg.target实际就是handler]。

2、创建子线程，发送消息handler.sendMessage(msg)

新开一个子线程，发送消息给另一个子线程

在handler.sendMessage(msg)方法中，实际上最终调用sendMessageAtTime(Message msg，long uptimeMillis)方法[sendMessageXXX方法都是最终调用的sendMessageAtTime方法]

此方法返回的enqueueMessage(queue，msg，uptimeMillis)，实际上返回的是MessageQueue中的enqueueMessage(msg，uptimeMillis)，其中进行的操作时对存入的消息进行列队，即根据接收到的消息的时间先后进行排列[使用的单链形式]；然后将消息就都存入到了消息队列中，等待着handler进行处理。

### 对各种引用的简单了解

##### 1.1 临界状态的处理

临界状态：

       当缓存内容过多，同时系统，内存又相对较低时的状态；

临界状态处理：

低内存预警：

       每当进行数据缓存时需要判断当前系统的内存值是否低于应用预设的最低内存；

       如果是，提示用户应用将在低内存环境下运行；

       Tips：

                     Intent.ACTION\_DEVICE\_STORAGE\_LOW;

                     设备内存不足时发出的广播,此广播只能由系统使用，其它APP不可用；

                     Intent.ACTION\_DEVICE\_STORAGE\_OK;

                     设备内存从不足到充足时发出的广播,此广播只能由系统使用，其它APP不可用；

构建高速缓存（扩展）

##### 1.2对象的引用的级别

       在JDK 1.2以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。即只有对象处于可触及（reachable）状态，程序才能使用它。

       从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。

这4种级别由高到低依次为：强引用、软引用、弱引用和虚引用；

###### 1.2.1强引用（StrongReference）

如：Object object = new Object();

特点：

强引用是使用最普遍的引用；

如果一个对象具有强引用，那垃圾回收器绝不会回收它，内存不足时，宁抛异常OOM，程序终止也不回收；

###### 1.2.2软引用（SoftReference）

JDK提供创建软引用的类SoftReference：

       通过“袋子”(sr) 来拿“内容”(object)；

       系统发现不足时，会将“袋子”中的“内容”回收，这时，将拿到null了，此时，这个“壳”也没有用了，需要干掉；

       Object object = new Object();                                 // 占用系统内容较多的对象            （内容）

       SoftReference sr = new SoftReference(object);         // 将object对象的引用级别降低      （袋子）

SoftReference的特点是它的实例保存对一个Java对象的软引用，该软引用的存在不妨碍垃圾收集线程对该Java对象的回收。

一旦SoftReference保存了对一个Java对象的软引用后，在垃圾线程对这个Java对象回收前，SoftReference类所提供的get()方法返回Java对象的强引用。

另外，一旦垃圾线程回收该Java对象之后，get()方法将返回null；

特点：

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；

如果内存空间不足了，就会回收这些对象的内存，会在抛出OOM之前回收掉；

       c.  只要垃圾回收器没有回收它，该对象就可以被程序使用，软引用可用来实现内存敏感的高速缓存；

       d.    软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，

              Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

说明一下软引用：

       Object object = new Object();                                 // 占用系统内容较多的对象            （内容）

       SoftReference sr = new SoftReference(object);         // 将object对象的引用级别降低      （袋子）

此时，对于这个Object对象，有两个引用路径：

一个是来自SoftReference对象的软引用；

一个来自变量object的强引用，所以这个Object对象是强可及对象；

随即，我们可以结束object对这个Object实例的强引用:

       object = null;

此后，这个Object对象成为了软可及对象；

如果垃圾收集线程进行内存垃圾收集，并不会因为有一个SoftReference对该对象的引用而始终保留该对象；

Java虚拟机的垃圾收集线程对软可及对象和其他一般Java对象进行了区别对待:

       软可及对象的清理是由垃圾收集线程根据其特定算法按照内存需求决定的。

       也就是说，垃圾收集线程会在虚拟机抛出OutOfMemoryError之前回收软可及对象，而且虚拟机会尽可能优先回收长时间闲置不用的软可及对象，

       对那些刚刚构建的或刚刚使用过的“新”软可反对象会被虚拟机尽可能保留。

       在回收这些对象之前，我们可以通过，Object anotherRef=(Object)aSoftRef.get()，重新获得对该实例的强引用。

       回收之后，调用get()方法就只能得到null了。

###### 1.2.3弱引用（WeakReference）

       弱引用与软引用的区别：只具有弱引用的对象拥有更短暂的生命周期。

特点：

生命周期比软引用更短；

在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存；

不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象；

类似于软引用，弱引用也可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，

Java虚拟机就会把这个弱引用加入到与之关联的引用队列中

###### 1.2.4虚引用（PhantomReference）

       “虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

特点：

形同虚设；

可用来跟踪对象被垃圾回收器回收的活动；

虚引用与软引用和弱引用的一个区别在于：

虚引用必须和引用队列 （ReferenceQueue）联合使用，

当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中；

1.3 ReferenceQueue与软引用结合使用

ReferenceQueue的作用：

引用队列，在检测到适当的可到达性更改后，垃圾回收器将已注册的引用对象添加到该队列中；

       利用ReferenceQueue的特性，即用来清除失去了软引用对象的SoftReference；

为什么需要ReferenceQueue：

       作为一个Java对象，SoftReference对象除了具有保存软引用的特殊性之外，也具有Java对象的一般性。

       所以，当软可及对象（SoftReference袋中对象）被回收之后，虽然这个SoftReference对象的get()方法返回null；

       但这个SoftReference对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量SoftReference对象带来的内存泄漏。

       这时候需要用到ReferenceQueue类；

       如果在创建SoftReference对象的时候，使用了一个ReferenceQueue对象作为参数提供给SoftReference的构造方法，如:

              Object object = new Object();                                                      // 占用系统内容较多的对象            （内容）

              ReferenceQueue queue = new ReferenceQueue();                                 // 装SoftReference的队列

              SoftReference sr=new SoftReference(object, queue);

那么当这个SoftReference所软引用的object被垃圾收集器回收的同时，sr所强引用的SoftReference对象被列入ReferenceQueue。

也就是说，ReferenceQueue中保存的对象是Reference对象，而且是已经失去了它所软引用的对象的Reference对象。

另外从ReferenceQueue这个名字也可以看出，它是一个队列；

当我们调用它的poll()方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个Reference对象。

       在任何时候，我们都可以调用ReferenceQueue的poll()方法来检查是否有它所关心的非强可及对象被回收。

       如果队列为空，将返回一个null,否则该方法返回队列中前面的一个Reference对象。

       利用这个方法，我们可以检查哪个SoftReference所软引用的对象已经被回收。

       于是我们可以把这些失去所软引用的对象的SoftReference对象清除掉。

示例：

一、当界面显示较多的时候，内存就会占用很多，会导致手机内容不足

在UIManager中：

1、判断手机当前的可用内存（如10M——看成应用需要的最大内存（峰值内存））

       在创建的BASEVIEWS的map时处理：

       ①、BASEVIEWS显示大小：要动态的配置，依据内存大小变动，不好

       ②、降低对象的引用级别  <== 目的

              关于Java对象引用级别：强引用、软引用、弱引用、虚引用【1.2以后出现】

                     关于强引用：当new出来一个，为强引用；GC在内存不足时，宁可抛出OOM（内存溢出）的异常也不回收强引用的对象

                     使用软引用：在出现异常之前，让GC在OOM之前就把引用回收掉。软引用：SoftReference

       采用②方案：关于返回键的处理——直接返回到首页，同时提示用户应用在低内存下运行

手机彩票代码处理：

1、在一加载的时候，进行判断，当内存够用的时候，创建出强引用集合，不足时，创建软引用的map

private static Map<String, BaseView> BASEVIEWS;// key:子类的简单名称

static {

              if (MemoryManager.hasAcailMemory()) {

                     BASEVIEWS = new HashMap<String, BaseView>();// key:子类的简单名称

              } else {

                     BASEVIEWS = new SoftHashMap<String, BaseView>();// 软引用的map

              }

       }

2、处理返回键：

由于占有内存空间最大的就是BASEVIEWS这个存放界面的集合，当内存不够的时候，回收掉这个集合，即将显示过的界面都清除掉

但是新显示的界面不会有问题，因为是新创建的，不会被干掉。但是当点击返回键的时候，是找不到集合中的内容的，就会出现空指针异常

所以，在处理返回键的时候，就需要特别注意：就直接创建出首页，并返回，提示用户。【主要针对测试用的】

è 当目标view为空的时候，即不存在历史view 的时候，返回主页

①、提示用户：在低内存下运行：

       PromptManager.showToast(getContext(), "应用在低内存下运行");

②、清空返回键

       clear();

③、显示首页

       changeView(Hall.class, null);

二、方案二的具体实现：创建软引用的集合：

一）分析：

1、创建出软引用的集合SoftHashMap：

public class SoftHashMap<K, V> extends HashMap<K, V>

2、目的：降低对象的引用级别

①、将V的引用级别降低

②、回收“空袋子”：即存储Object的软引用SoftReference

3、SoftReference(T referent, ReferenceQueue<? super T> q)：

       参数1：强引用，相当于手机

       参数2：队列，存“袋子”的队列

①、指定好ReferenceQueue，是存“袋子”的队列，会依据V（强引用（手机））存对应的软引用（袋子）；

②、当GC回收的时候，ReferenceQueue会进行查询，如果不为空，会将“空袋子”（没有强引用的软引用）存入到这个队列中

如果队列中有值，说明有“空袋子”。这样，只需要循环这个队列，就可以将“空袋子”进行回收掉了

二）具体实现：

1、创建集合：

①、临时的HashMap:

private HashMap<K, SoftValue<K, V>> temp;

将HashMap中的V包装了一层，就类似于给V加了个软引用【类似给手机加了个袋子】，让系统可以把V回收掉

②、创建存软引用的队列（里面是装V的袋子）：

       private ReferenceQueue<V> queue;

2、在构造函数中初始化集合：

Tips：

       new一个对象，是作为强引用存在的；

将这个强引用对象（类比为手机）放到SoftReference（类比于袋子）中，就相当于将手机放入袋子中

这样就实现了降低对象的引用级别

当内存不足的时候，GC会将占用空间较多的Object回收，而不会将sr回收掉

如：

//Object object = new Object();// 占有系统内存较多的对象

       // SoftReference sr = new SoftReference(object);// 将object的对象引用级别降低了

①、初始化两个集合：

@temp = new HashMap<K, SoftValue<K, V>>();

//在操作的时候，是操作的temp这个Map，即存入到这个集合中的对象，而不是系统中的东西，

       @queue = new ReferenceQueue<V>();

       // ReferenceQueue<V>是一个队列，里面放的是软引用（“空袋子”），依据V存的

需要重写用到的方法：

       但凡涉及到了super（HashMap中的数据，都不能使用，因为没有软引用的功能）

3、重写put方法：

①、创建出V的对象：SoftReference<V> sr = new SoftReference<V>(value);

②、创建软引用，将强引用对象封装到软引用中（类似于将手机装入袋子）： temp.put(key, sr);

不能调用super.put(key，value)，这样就调用了HashMap这个集合了；我们需要调用的是temp这个集合（含有软引用的集合）

直接put到temp这个集合中，才能操作到这个集合中的对象

③、返回的为null，或者返回put方法的返回值也可以

4、重写get方法：

Tips：也不能用super.get(key)，因为没存到父类集合中

①、通过temp这个集合获取到，获取到的是装强引用的软引用对象（即装手机的袋子）：

SoftReference<V> sr = temp.get(key);

②、返回软引用的对象：sr.get();

5、重写containsKey方法：

①、如果V（手机）被GC回收了，此方法就没有意义了，就无法调用临时map（temp）的containsKey

       所以，只需要判断V是否为空，就能得到是否包含了对应的key的值。

因此，判断获得的强引用的值是否为空，不为空，才调用containsKey

V v = get(key);

              boolean isContain = false;

              if (v != null) {

                     isContain = true;

              }

              return isContain;

6、回收软引用：

Tips：

       虽然软引用（袋子）占用内存不多，但是在低内存状态下运行的

如果软引用中都没有引用的“强引用的对象”了，就无需这个软引用的集合了（即“空袋子”）

即：强引用（手机）都没了，要这个软引用（空袋子）也没什么用处了

①、回收“空袋子”：

@方案1：循环temp中的所有内存，若发现V==null，再temp中删除对应的空袋子

              没必要循环temp集合，循环清空每个“空袋子”：

\*因为当还没有到OOM（内存溢出）的时候，这个循环没有意义，因为没有强引用被回收掉，所以不会回收掉“袋子”的

\*当内存充足的时候，是不会执行这个清空方法的，也没必要清空

@方案2：让GC记录一下回收的内容（集合中:存储空袋子的引用），如果GC回收内容了，集合的size>0，再循环回收

\*进行轮询，获取“空袋子”，

                     poll()：会轮询此队列，查看是否存在可用的引用对象；如果有的话，进行移除并返回；没有返回null

                            SoftValue<K, V> poll = (SoftValue<K, V>) queue.poll();//获取到的是值

              \*循环中再次获取poll，直到集合中没有元素了，就不在循环了

                     集合中的remove方法：remove(key)是没有依据值（value）进行删除的【因为poll返回的是value】

                     这时，就需要改造一下这个remove，创建加强版的“袋子”（见下）：

                     当poll（poll()方法返回的值）不为空，再次循环，直到为空，说明“空袋子”都清空了：

                     while (poll != null) {

                            temp.remove(poll.key);

                            poll = (SoftValue<K, V>) queue.poll();

                     }

创建加强版的“袋子”：存储一下key：

       因为系统中的集合中没有直接依据值删除指定的元素

       只能remove(Object key)，现在是通过加强功能，直接通过“袋子”的key（类似标签）进行删除；

       因此要加强存“袋子”的队列ReferenceQueue<V>

①、创建自定义的类SoftValue<K, V>，继承ReferenceQueue<V>

②、创建构造函数

       通过构造传递key，从而可以获取对应的value

/\*\*

        \* 加强版的袋子：存储一下key

        \*/

       private class SoftValue<K, V> extends SoftReference<V> {

              private Object key;

              public SoftValue(K key, V r, ReferenceQueue<? super V> q) {

                     super(r, q);

                     this.key = key;

              }

       }

示例代码：

import java.lang.ref.ReferenceQueue;

import java.lang.ref.SoftReference;

import java.util.HashMap;

/\*\*

 \* 软引用的map

 \*

 \* @author Administrator

 \*

 \* @param <K>

 \* @param <V>

 \*/

public class SoftHashMap<K, V> extends HashMap<K, V> {

       // 降低对象的引用级别

       // ①将V的应用级别降低

       // ②回收"空袋子"

       private HashMap<K, SoftValue<K, V>> temp;

       private ReferenceQueue<V> queue;// 装V的袋子

       public SoftHashMap() {

              // Object object = new Object();// 占有系统内存较多的对象

              // SoftReference sr = new SoftReference(object);// 将object的对象引用级别降低了

              temp = new HashMap<K, SoftValue<K, V>>();

              queue = new ReferenceQueue<V>();

       }

       @Override

       public V put(K key, V value) {

              SoftValue<K, V> sr = new SoftValue<K, V>(key, value, queue);

              temp.put(key, sr);

              return null;

       }

       @Override

       public V get(Object key) {

              clearNullSR();// 清理空袋子

              SoftValue<K, V> sr = temp.get(key);// 如果是空袋子——已经被回收了，获取到的对象为null

              if (sr != null) {

                     return sr.get();

              } else {

                     return null;

              }

       }

       @Override

       public boolean containsKey(Object key) {

              // temp.containsKey(key);//如果V（手机）被GC回收了

              clearNullSR();

              return temp.containsKey(key);

       }

       /\*\*

        \* 回收"空袋子"

        \*/

       private void clearNullSR() {

              // 方案一：循环temp中所有的内容，如果发现V=null，在temp中删除对应空袋子

              // 方案二：让GC，记录一下回收的内容（集合中:存储空袋子的引用），如果GC回收内容了，集合的size>0

              SoftValue<K, V> poll = (SoftValue<K, V>) queue.poll();

              while (poll != null) {

                     temp.remove(poll.key);

                     poll = (SoftValue<K, V>) queue.poll();

              }

       }

       /\*\*

        \* 加强版的袋子：存储一下key

        \*/

       private class SoftValue<K, V> extends SoftReference<V> {

              private Object key;

              public SoftValue(K key, V r, ReferenceQueue<? super V> q) {

                     super(r, q);

                     this.key = key;

              }

       }

}

### 图片的缓存

##### 高效加载大图片

我们在编写Android程序的时候经常要用到许多图片，不同图片总是会有不同的形状、不同的大小，但在大多数情况下，这些图片都会大于我们程序所需要的大小。比如说系统图片库里展示的图片大都是用手机摄像头拍出来的，这些图片的分辨率会比我们手机屏幕的分辨率高得多。大家应该知道，我们编写的应用程序都是有一定内存限制的，程序占用了过高的内存就容易出现OOM(OutOfMemory)异常。我们可以通过下面的代码看出每个应用程序最高可用内存是多少。

int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);

Log.d("TAG", "Max memory is " + maxMemory + "KB");

因此在展示高分辨率图片的时候，最好先将图片进行压缩。压缩后的图片大小应该和用来展示它的控件大小相近，在一个很小的ImageView上显示一张超大的图片不会带来任何视觉上的好处，但却会占用我们相当多宝贵的内存，而且在性能上还可能会带来负面影响。下面我们就来看一看，如何对一张大图片进行适当的压缩，让它能够以最佳大小显示的同时，还能防止OOM的出现。

BitmapFactory这个类提供了多个解析方法(decodeByteArray, decodeFile, decodeResource等)用于创建Bitmap对象，我们应该根据图片的来源选择合适的方法。比如SD卡中的图片可以使用decodeFile方法，网络上的图片可以使用decodeStream方法，资源文件中的图片可以使用decodeResource方法。这些方法会尝试为已经构建的bitmap分配内存，这时就会很容易导致OOM出现。为此每一种解析方法都提供了一个可选的BitmapFactory.Options参数，将这个参数的inJustDecodeBounds属性设置为true就可以让解析方法禁止为bitmap分配内存，返回值也不再是一个Bitmap对象，而是null。虽然Bitmap是null了，但是BitmapFactory.Options的outWidth、outHeight和outMimeType属性都会被赋值。这个技巧让我们可以在加载图片之前就获取到图片的长宽值和MIME类型，从而根据情况对图片进行压缩。如下代码所示：

[java] view plaincopy

BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true;

BitmapFactory.decodeResource(getResources(), R.id.myimage, options);

int imageHeight = options.outHeight;

int imageWidth = options.outWidth;

String imageType = options.outMimeType;

为了避免OOM异常，最好在解析每张图片的时候都先检查一下图片的大小，除非你非常信任图片的来源，保证这些图片都不会超出你程序的可用内存。

现在图片的大小已经知道了，我们就可以决定是把整张图片加载到内存中还是加载一个压缩版的图片到内存中。以下几个因素是我们需要考虑的：

预估一下加载整张图片所需占用的内存。

为了加载这一张图片你所愿意提供多少内存。

用于展示这张图片的控件的实际大小。

当前设备的屏幕尺寸和分辨率。

比如，你的ImageView只有128\*96像素的大小，只是为了显示一张缩略图，这时候把一张1024\*768像素的图片完全加载到内存中显然是不值得的。

那我们怎样才能对图片进行压缩呢？通过设置BitmapFactory.Options中inSampleSize的值就可以实现。比如我们有一张2048\*1536像素的图片，将inSampleSize的值设置为4，就可以把这张图片压缩成512\*384像素。原本加载这张图片需要占用13M的内存，压缩后就只需要占用0.75M了(假设图片是ARGB\_8888类型，即每个像素点占用4个字节)。下面的方法可以根据传入的宽和高，计算出合适的inSampleSize值：

[java] view plaincopy

public static int calculateInSampleSize(BitmapFactory.Options options,

        int reqWidth, int reqHeight) {

    // 源图片的高度和宽度

    final int height = options.outHeight;

    final int width = options.outWidth;

    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {

        // 计算出实际宽高和目标宽高的比率

        final int heightRatio = Math.round((float) height / (float) reqHeight);

        final int widthRatio = Math.round((float) width / (float) reqWidth);

        // 选择宽和高中最小的比率作为inSampleSize的值，这样可以保证最终图片的宽和高

        // 一定都会大于等于目标的宽和高。

        inSampleSize = heightRatio < widthRatio ? heightRatio : widthRatio;

    }

    return inSampleSize;

}

使用这个方法，首先你要将BitmapFactory.Options的inJustDecodeBounds属性设置为true，解析一次图片。然后将BitmapFactory.Options连同期望的宽度和高度一起传递到到calculateInSampleSize方法中，就可以得到合适的inSampleSize值了。之后再解析一次图片，使用新获取到的inSampleSize值，并把inJustDecodeBounds设置为false，就可以得到压缩后的图片了。

[java] view plaincopy

public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,

        int reqWidth, int reqHeight) {

    // 第一次解析将inJustDecodeBounds设置为true，来获取图片大小

    final BitmapFactory.Options options = new BitmapFactory.Options();

    options.inJustDecodeBounds = true;

    BitmapFactory.decodeResource(res, resId, options);

    // 调用上面定义的方法计算inSampleSize值

    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);

    // 使用获取到的inSampleSize值再次解析图片

    options.inJustDecodeBounds = false;

    return BitmapFactory.decodeResource(res, resId, options);

}

下面的代码非常简单地将任意一张图片压缩成100\*100的缩略图，并在ImageView上展示。

[java] view plaincopy

mImageView.setImageBitmap(

    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));

##### 使用图片缓存技术

在你应用程序的UI界面加载一张图片是一件很简单的事情，但是当你需要在界面上加载一大堆图片的时候，情况就变得复杂起来。在很多情况下，（比如使用ListView, GridView 或者 ViewPager 这样的组件），屏幕上显示的图片可以通过滑动屏幕等事件不断地增加，最终导致OOM。

为了保证内存的使用始终维持在一个合理的范围，通常会把被移除屏幕的图片进行回收处理。此时垃圾回收器也会认为你不再持有这些图片的引用，从而对这些图片进行GC操作。用这种思路来解决问题是非常好的，可是为了能让程序快速运行，在界面上迅速地加载图片，你又必须要考虑到某些图片被回收之后，用户又将它重新滑入屏幕这种情况。这时重新去加载一遍刚刚加载过的图片无疑是性能的瓶颈，你需要想办法去避免这个情况的发生。

这个时候，使用内存缓存技术可以很好的解决这个问题，它可以让组件快速地重新加载和处理图片。下面我们就来看一看如何使用内存缓存技术来对图片进行缓存，从而让你的应用程序在加载很多图片的时候可以提高响应速度和流畅性。

内存缓存技术对那些大量占用应用程序宝贵内存的图片提供了快速访问的方法。其中最核心的类是LruCache (此类在android-support-v4的包中提供) 。这个类非常适合用来缓存图片，它的主要算法原理是把最近使用的对象用强引用存储在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。

在过去，我们经常会使用一种非常流行的内存缓存技术的实现，即软引用或弱引用 (SoftReference or WeakReference)。但是现在已经不再推荐使用这种方式了，因为从 Android 2.3 (API Level 9)开始，垃圾回收器会更倾向于回收持有软引用或弱引用的对象，这让软引用和弱引用变得不再可靠。另外，Android 3.0 (API Level 11)中，图片的数据会存储在本地的内存当中，因而无法用一种可预见的方式将其释放，这就有潜在的风险造成应用程序的内存溢出并崩溃。

为了能够选择一个合适的缓存大小给LruCache, 有以下多个因素应该放入考虑范围内，例如：

你的设备可以为每个应用程序分配多大的内存？

设备屏幕上一次最多能显示多少张图片？有多少图片需要进行预加载，因为有可能很快也会显示在屏幕上？

你的设备的屏幕大小和分辨率分别是多少？一个超高分辨率的设备（例如 Galaxy Nexus) 比起一个较低分辨率的设备（例如 Nexus S），在持有相同数量图片的时候，需要更大的缓存空间。

图片的尺寸和大小，还有每张图片会占据多少内存空间。

图片被访问的频率有多高？会不会有一些图片的访问频率比其它图片要高？如果有的话，你也许应该让一些图片常驻在内存当中，或者使用多个LruCache 对象来区分不同组的图片。

你能维持好数量和质量之间的平衡吗？有些时候，存储多个低像素的图片，而在后台去开线程加载高像素的图片会更加的有效。

并没有一个指定的缓存大小可以满足所有的应用程序，这是由你决定的。你应该去分析程序内存的使用情况，然后制定出一个合适的解决方案。一个太小的缓存空间，有可能造成图片频繁地被释放和重新加载，这并没有好处。而一个太大的缓存空间，则有可能还是会引起 java.lang.OutOfMemory 的异常。

下面是一个使用 LruCache 来缓存图片的例子：

[java] view plaincopy

private LruCache<String, Bitmap> mMemoryCache;

@Override

protected void onCreate(Bundle savedInstanceState) {

    // 获取到可用内存的最大值，使用内存超出这个值会引起OutOfMemory异常。

    // LruCache通过构造函数传入缓存值，以KB为单位。

    int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);

    // 使用最大可用内存值的1/8作为缓存的大小。

    int cacheSize = maxMemory / 8;

    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {

        @Override

        protected int sizeOf(String key, Bitmap bitmap) {

            // 重写此方法来衡量每张图片的大小，默认返回图片数量。

            return bitmap.getByteCount() / 1024;

        }

    };

}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {

    if (getBitmapFromMemCache(key) == null) {

        mMemoryCache.put(key, bitmap);

    }

}

public Bitmap getBitmapFromMemCache(String key) {

    return mMemoryCache.get(key);

}

在这个例子当中，使用了系统分配给应用程序的八分之一内存来作为缓存大小。在中高配置的手机当中，这大概会有4兆(32/8)的缓存空间。一个全屏幕的 GridView 使用4张 800x480分辨率的图片来填充，则大概会占用1.5兆的空间(800\*480\*4)。因此，这个缓存大小可以存储2.5页的图片。

当向 ImageView 中加载一张图片时,首先会在 LruCache 的缓存中进行检查。如果找到了相应的键值，则会立刻更新ImageView ，否则开启一个后台线程来加载这张图片。

[java] view plaincopy

public void loadBitmap(int resId, ImageView imageView) {

    final String imageKey = String.valueOf(resId);

    final Bitmap bitmap = getBitmapFromMemCache(imageKey);

    if (bitmap != null) {

        imageView.setImageBitmap(bitmap);

    } else {

        imageView.setImageResource(R.drawable.image\_placeholder);

        BitmapWorkerTask task = new BitmapWorkerTask(imageView);

        task.execute(resId);

    }

}

BitmapWorkerTask 还要把新加载的图片的键值对放到缓存中。

[java] view plaincopy

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {

    // 在后台加载图片。

    @Override

    protected Bitmap doInBackground(Integer... params) {

        final Bitmap bitmap = decodeSampledBitmapFromResource(

                getResources(), params[0], 100, 100);

        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);

        return bitmap;

    }

}

掌握了以上两种方法，不管是要在程序中加载超大图片，还是要加载大量图片，都不用担心OOM的问题了!不过仅仅是理论地介绍不知道大家能不能完全理解，在后面的文章中我会演示如何在实际程序中灵活运用上述技巧来避免程序OOM

###### Android照片墙的实现：

照片墙这种功能现在应该算是挺常见了，在很多应用中你都可以经常看到照片墙的身影。它的设计思路其实也非常简单，用一个GridView控件当作“墙”，然后随着GridView的滚动将一张张照片贴在“墙”上，这些照片可以是手机本地中存储的，也可以是从网上下载的。制作类似于这种的功能的应用，有一个非常重要的问题需要考虑，就是图片资源何时应该释放。因为随着GridView的滚动，加载的图片可能会越来越多，如果没有一种合理的机制对图片进行释放，那么当图片达到一定上限时，程序就必然会崩溃。

今天我们照片墙应用的实现，重点也是放在了如何防止由于图片过多导致程序崩溃上面。主要的核心算法使用了Android中提供的LruCache类，这个类是3.1版本中提供的，如果你是在更早的Android版本中开发，则需要导入android-support-v4的jar包。

第一个要考虑的问题就是，我们从哪儿去收集这么多的图片呢？这里我从谷歌官方提供的Demo里将图片源取了出来，我们就从这些网址中下载图片，代码如下所示：

新建或打开activity\_main.xml作为程序的主布局，加入如下代码：

[html] view plaincopy

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    android:layout\_width="wrap\_content"

    android:layout\_height="wrap\_content" >

    <GridView

        android:id="@+id/photo\_wall"

        android:layout\_width="match\_parent"

        android:layout\_height="wrap\_content"

        android:columnWidth="90dip"

        android:stretchMode="columnWidth"

        android:numColumns="auto\_fit"

        android:verticalSpacing="10dip"

        android:gravity="center"

        ></GridView>

</LinearLayout>

可以看到，我们在这个布局文件中仅加入了一个GridView，这也就是我们程序中的“墙”，所有的图片都将贴在这个“墙”上。

接着我们定义GridView中每一个子View的布局，新建一个photo\_layout.xml布局，加入如下代码：

[html] view plaincopy

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    android:layout\_width="wrap\_content"

    android:layout\_height="wrap\_content" >

    <ImageView

        android:id="@+id/photo"

        android:layout\_width="90dip"

        android:layout\_height="90dip"

        android:src="@drawable/empty\_photo"

        android:layout\_centerInParent="true"

        />

</RelativeLayout>

在每一个子View中我们就简单使用了一个ImageView来显示一张图片。这样所有的布局就已经定义好了。

接下来新建PhotoWallAdapter做为GridView的适配器，代码如下所示：

[java] view plaincopy

public class PhotoWallAdapter extends ArrayAdapter<String> implements OnScrollListener {

    /\*\*

     \* 记录所有正在下载或等待下载的任务。

     \*/

    private Set<BitmapWorkerTask> taskCollection;

    /\*\*

     \* 图片缓存技术的核心类，用于缓存所有下载好的图片，在程序内存达到设定值时会将最少最近使用的图片移除掉。

     \*/

    private LruCache<String, Bitmap> mMemoryCache;

    /\*\*

     \* GridView的实例

     \*/

    private GridView mPhotoWall;

    /\*\*

     \* 第一张可见图片的下标

     \*/

    private int mFirstVisibleItem;

    /\*\*

     \* 一屏有多少张图片可见

     \*/

    private int mVisibleItemCount;

    /\*\*

     \* 记录是否刚打开程序，用于解决进入程序不滚动屏幕，不会下载图片的问题。

     \*/

    private boolean isFirstEnter = true;

    public PhotoWallAdapter(Context context, int textViewResourceId, String[] objects,

            GridView photoWall) {

        super(context, textViewResourceId, objects);

        mPhotoWall = photoWall;

        taskCollection = new HashSet<BitmapWorkerTask>();

        // 获取应用程序最大可用内存

        int maxMemory = (int) Runtime.getRuntime().maxMemory();

        int cacheSize = maxMemory / 8;

        // 设置图片缓存大小为程序最大可用内存的1/8

        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {

            @Override

            protected int sizeOf(String key, Bitmap bitmap) {

                return bitmap.getByteCount();

            }

        };

        mPhotoWall.setOnScrollListener(this);

    }

    @Override

    public View getView(int position, View convertView, ViewGroup parent) {

        final String url = getItem(position);

        View view;

        if (convertView == null) {

            view = LayoutInflater.from(getContext()).inflate(R.layout.photo\_layout, null);

        } else {

            view = convertView;

        }

        final ImageView photo = (ImageView) view.findViewById(R.id.photo);

        // 给ImageView设置一个Tag，保证异步加载图片时不会乱序

        photo.setTag(url);

        setImageView(url, photo);

        return view;

    }

    /\*\*

     \* 给ImageView设置图片。首先从LruCache中取出图片的缓存，设置到ImageView上。如果LruCache中没有该图片的缓存，

     \* 就给ImageView设置一张默认图片。

     \*

     \* @param imageUrl

     \*            图片的URL地址，用于作为LruCache的键。

     \* @param imageView

     \*            用于显示图片的控件。

     \*/

    private void setImageView(String imageUrl, ImageView imageView) {

        Bitmap bitmap = getBitmapFromMemoryCache(imageUrl);

        if (bitmap != null) {

            imageView.setImageBitmap(bitmap);

        } else {

            imageView.setImageResource(R.drawable.empty\_photo);

        }

    }

    /\*\*

     \* 将一张图片存储到LruCache中。

     \*

     \* @param key

     \*            LruCache的键，这里传入图片的URL地址。

     \* @param bitmap

     \*            LruCache的键，这里传入从网络上下载的Bitmap对象。

     \*/

    public void addBitmapToMemoryCache(String key, Bitmap bitmap) {

        if (getBitmapFromMemoryCache(key) == null) {

            mMemoryCache.put(key, bitmap);

        }

    }

    /\*\*

     \* 从LruCache中获取一张图片，如果不存在就返回null。

     \*

     \* @param key

     \*            LruCache的键，这里传入图片的URL地址。

     \* @return 对应传入键的Bitmap对象，或者null。

     \*/

    public Bitmap getBitmapFromMemoryCache(String key) {

        return mMemoryCache.get(key);

    }

    @Override

    public void onScrollStateChanged(AbsListView view, int scrollState) {

        // 仅当GridView静止时才去下载图片，GridView滑动时取消所有正在下载的任务

        if (scrollState == SCROLL\_STATE\_IDLE) {

            loadBitmaps(mFirstVisibleItem, mVisibleItemCount);

        } else {

            cancelAllTasks();

        }

    }

    @Override

    public void onScroll(AbsListView view, int firstVisibleItem, int visibleItemCount,

            int totalItemCount) {

        mFirstVisibleItem = firstVisibleItem;

        mVisibleItemCount = visibleItemCount;

        // 下载的任务应该由onScrollStateChanged里调用，但首次进入程序时onScrollStateChanged并不会调用，

        // 因此在这里为首次进入程序开启下载任务。

        if (isFirstEnter && visibleItemCount > 0) {

            loadBitmaps(firstVisibleItem, visibleItemCount);

            isFirstEnter = false;

        }

    }

    /\*\*

     \* 加载Bitmap对象。此方法会在LruCache中检查所有屏幕中可见的ImageView的Bitmap对象，

     \* 如果发现任何一个ImageView的Bitmap对象不在缓存中，就会开启异步线程去下载图片。

     \*

     \* @param firstVisibleItem

     \*            第一个可见的ImageView的下标

     \* @param visibleItemCount

     \*            屏幕中总共可见的元素数

     \*/

    private void loadBitmaps(int firstVisibleItem, int visibleItemCount) {

        try {

            for (int i = firstVisibleItem; i < firstVisibleItem + visibleItemCount; i++) {

                String imageUrl = Images.imageThumbUrls[i];

                Bitmap bitmap = getBitmapFromMemoryCache(imageUrl);

                if (bitmap == null) {

                    BitmapWorkerTask task = new BitmapWorkerTask();

                    taskCollection.add(task);

                    task.execute(imageUrl);

                } else {

                    ImageView imageView = (ImageView) mPhotoWall.findViewWithTag(imageUrl);

                    if (imageView != null && bitmap != null) {

                        imageView.setImageBitmap(bitmap);

                    }

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

    /\*\*

     \* 取消所有正在下载或等待下载的任务。

     \*/

    public void cancelAllTasks() {

        if (taskCollection != null) {

            for (BitmapWorkerTask task : taskCollection) {

                task.cancel(false);

            }

        }

    }

    /\*\*

     \* 异步下载图片的任务。

     \*

     \* @author guolin

     \*/

    class BitmapWorkerTask extends AsyncTask<String, Void, Bitmap> {

        /\*\*

         \* 图片的URL地址

         \*/

        private String imageUrl;

        @Override

        protected Bitmap doInBackground(String... params) {

            imageUrl = params[0];

            // 在后台开始下载图片

            Bitmap bitmap = downloadBitmap(params[0]);

            if (bitmap != null) {

                // 图片下载完成后缓存到LrcCache中

                addBitmapToMemoryCache(params[0], bitmap);

            }

            return bitmap;

        }

        @Override

        protected void onPostExecute(Bitmap bitmap) {

            super.onPostExecute(bitmap);

            // 根据Tag找到相应的ImageView控件，将下载好的图片显示出来。

            ImageView imageView = (ImageView) mPhotoWall.findViewWithTag(imageUrl);

            if (imageView != null && bitmap != null) {

                imageView.setImageBitmap(bitmap);

            }

            taskCollection.remove(this);

        }

        /\*\*

         \* 建立HTTP请求，并获取Bitmap对象。

         \*

         \* @param imageUrl

         \*            图片的URL地址

         \* @return 解析后的Bitmap对象

         \*/

        private Bitmap downloadBitmap(String imageUrl) {

            Bitmap bitmap = null;

            HttpURLConnection con = null;

            try {

                URL url = new URL(imageUrl);

                con = (HttpURLConnection) url.openConnection();

                con.setConnectTimeout(5 \* 1000);

                con.setReadTimeout(10 \* 1000);

                con.setDoInput(true);

                con.setDoOutput(true);

                bitmap = BitmapFactory.decodeStream(con.getInputStream());

            } catch (Exception e) {

                e.printStackTrace();

            } finally {

                if (con != null) {

                    con.disconnect();

                }

            }

            return bitmap;

        }

    }

}

PhotoWallAdapter是整个照片墙程序中最关键的一个类了，这里我来重点给大家讲解一下。首先在PhotoWallAdapter的构造函数中，我们初始化了LruCache类，并设置了最大缓存容量为程序最大可用内存的1/8，接下来又为GridView注册了一个滚动监听器。然后在getView()方法中，我们为每个ImageView设置了一个唯一的Tag，这个Tag的作用是为了后面能够准确地找回这个ImageView，不然异步加载图片会出现乱序的情况。之后调用了setImageView()方法为ImageView设置一张图片，这个方法首先会从LruCache缓存中查找是否已经缓存了这张图片，如果成功找到则将缓存中的图片显示在ImageView上，否则就显示一张默认的空图片。

看了半天，那到底是在哪里下载图片的呢？这是在GridView的滚动监听器中进行的，在onScrollStateChanged()方法中，我们对GridView的滚动状态进行了判断，如果当前GridView是静止的，则调用loadBitmaps()方法去下载图片，如果GridView正在滚动，则取消掉所有下载任务，这样可以保证GridView滚动的流畅性。在loadBitmaps()方法中，我们为屏幕上所有可见的GridView子元素开启了一个线程去执行下载任务，下载成功后将图片存储到LruCache当中，然后通过Tag找到相应的ImageView控件，把下载好的图片显示出来。

由于我们使用了LruCache来缓存图片，所以不需要担心内存溢出的情况，当LruCache中存储图片的总大小达到容量上限的时候，会自动把最近最少使用的图片从缓存中移除。

最后新建或打开MainActivity作为程序的主Activity，代码如下所示：

[java] view plaincopy

public class MainActivity extends Activity {

    /\*\*

     \* 用于展示照片墙的GridView

     \*/

    private GridView mPhotoWall;

    /\*\*

     \* GridView的适配器

     \*/

    private PhotoWallAdapter adapter;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_main);

        mPhotoWall = (GridView) findViewById(R.id.photo\_wall);

        adapter = new PhotoWallAdapter(this, 0, Images.imageThumbUrls, mPhotoWall);

        mPhotoWall.setAdapter(adapter);

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        // 退出程序时结束所有的下载任务

        adapter.cancelAllTasks();

    }

}

MainActivity中的代码非常简单，没什么需要说明的了，在Activity被销毁时取消掉了所有的下载任务，避免程序在后台耗费流量。另外由于我们使用了网络功能，别忘了在AndroidManifest.xml中加入网络权限的声明。

### 能够对图片的优化进行相应的处理

1、利用“三级缓存”实现图片的优化

第一次是要从服务器获取的，以后每次显示图片，首先判断内存中获取，如果没有，再从本地缓存目录中获取，如果还没有，最后就需要发送请求从服务器获取。服务器端下载的图片是使用Http的缓存机制，每次执行将本地图片的时间发送给服务器，如果返回码是304，说明服务端的图片和本地的图片是相同的，直接使用本地保存的图片，如果返回码是200，则开始下载新的图片并实现缓存。在从服务器获取到图片后，需要再在本地和内存中分别存一份，这样下次直接就可以从内存中直接获取了，这样就加快了显示的速度，提高了用户的体验。

2、图片过大导致内存溢出：

模拟器的RAM比较小，由于每张图片先前是压缩的情况，放入到Bitmap的时候，大小会变大，导致超出RAM内存

★android 中用bitmap 时很容易内存溢出，报如下错误：Java.lang.OutOfMemoryError : bitmap size exceeds VM budget

解决：

方法1： 主要是加上这段：等比例缩小图片

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;

1）通过getResource()方法获取资源：

       //解决加载图片 内存溢出的问题

        //Options 只保存图片尺寸大小，不保存图片到内存

       BitmapFactory.Options opts = new BitmapFactory.Options();

       //缩放的比例，缩放是很难按准备的比例进行缩放的，其值表明缩放的倍数，SDK中建议其值是2的指数值,值越大会导致图片不清晰

       opts.inSampleSize = 2;

       Bitmap bmp = null;

       bmp = BitmapFactory.decodeResource(getResources(), mImageIds[position],opts);

       ...

       //回收

       bmp.recycle();

2）通过Uri取图片资源

private ImageView preview;

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一

Bitmap bitmap = BitmapFactory.decodeStream(cr.openInputStream(uri), null, options);

preview.setImageBitmap(bitmap);

以上代码可以优化内存溢出，但它只是改变图片大小，并不能彻底解决内存溢出。

3）通过路径获取图片资源

private ImageView preview;

private String fileName= "/sdcard/DCIM/Camera/2010-05-14 16.01.44.jpg";

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一

Bitmap b = BitmapFactory.decodeFile(fileName, options);

preview.setImageBitmap(b);

filePath.setText(fileName);

方法2：对图片采用软引用，及时地进行recyle()操作

SoftReference<Bitmap> bitmap;

bitmap = new SoftReference<Bitmap>(pBitmap);

if(bitmap != null){

if(bitmap.get() != null && !bitmap.get().isRecycled()){

bitmap.get().recycle();

bitmap = null;

}

}

### 1、为何使用软引用：

       由于创建的集合中存的都是强引用的对象，对于强引用的对象，垃圾回收器是绝对不会回收这个强引用的对象的，除非手动将对象置为null，垃圾回收器才会在适当的时候将其回收掉。当内存不足的时候，即使抛出了OOM异常，程序终止了也不会回收这个强引用对象。所以为了避免在低内存下缓存图片而导致OOM异常的出现，需要降低对象的引用级别，这就涉及到了软引用。

简单来说，软引用就相当于一个“袋子”，而强引用就相当于袋子中的内容，可以比喻为将手机(内容)存入到袋子中，即用软引用包裹强引用。软引用SoftReference的特点就在于它的实例保存了对一个java对象的软引用，该软引用的存在并不妨碍垃圾回收线程对该java对象的回收。SoftReference保存了对一个java对象的软引用后，在垃圾回收此java对象之前，SoftReference所提供的的get()方法返回的是java对象的强引用；一旦垃圾线程回收该java对象之后，get()返回的是null。

2、软引用的特点：

1）、在内存空间充足时，即使一个对象具有软引用，垃圾回收器也不会回收它。

2）、当内存不足时，会在出现OOM异常之前回收掉这些对象的内存空间。

3）、只要垃圾回收器没回收这个对象，则程序就可以继续使用这个对象，因此软引用可用来实现内存敏感的高速缓存。

4）、软引用和引用队列ReferenceQueue联合使用时，当软引用所引用的对象被垃圾回收器回收后，java虚拟机就会将这个软引用加入到与之关联的引用队列中。

需要说明的是，对于强引用对象（如new一个对象），如果被软引用所引用后，不为null时是作为强可及对象存在的，如果为null后，是作为软可及对象存在的；当垃圾回收的时候，不会因为这个对象被软引用所引用而保留该对象的。而是会在OOM异常之前优先回收掉长时间闲置的软可及对象，尽可能保留“新”的软可及对象。如果想重新获得对该实例的强引用，可以通过调用软引用的get()方法获得对象后继续使用。

3、引用队列ReferenceQueue：

虽然SoftReference对象具有保存软引用的特性，但是也还是具有java对象的一般性的。也就是说当软引用所引用的对象被回收之后，这个软引用的对象实际上并没有什么存在的价值了，这就需要一个适当的清除机制，避免由于大量的SoftReference对象的存在而带来新的内存泄露的问题。这就需要将这些“空袋子”回收掉，这就需要使用引用队列ReferenceQueue这个类。

使用方法：就是将创建的强引用和引用队列作为参数传递到软引用的构造方法中，从而实现在SoftReference所引用的object在被垃圾回收器回收的同时，这个软引用对象会被加入到引用队列ReferenceQueue之中。我们可以通过ReferenceQueue的poll()方法监控到是否有非强可及对象被回收了。当队列为空时，说明没有软引用加入到队列中，即没有非强可及引用被回收，否则poll()方法会返回队列中前面一个Reference对象。通过这个方法，我们就可以将无引用对象的软引用SoftReference回收掉，这就避免了大量SoftReference未被回收导致的内存泄露。

4、构建高级缓存：

在应用之中，当我们将图片加载到内存中，需要考虑内存的优化问题，同样会涉及到OOM的问题。如果图片过多的话，内存就吃不消了，这就需要考虑应用如何在低内存的情况下运行了，需要为应用构建高级缓存，来保证在低内存的情况下也能正常运行。首先，在缓存图片的时候，需要判断手机的当前可用内存是否充足，如果内存不足，就需要使用缓存的Map来存储，保证在高速缓存下运行。但是在java中并没有提供软引用的Map集合，只提供了一个WeakHashMap这个针对弱引用提供的实现类。这时候就需要我们手动创建一个具体的实现类，作为软引用的集合来使用。

1、自定义软引用集合，继承HashMap<K, V>，这就相当于一个“袋子”

2、在构造函数中初始化：

       临时集合：创建一个临时的HashMap，其中的V应当作为软引用存在，即使用自定义的软引用的类[此类是加入队列的软引用]。可以理解为将手机（强引用）放入袋子（软引用）中，在将袋子贴上一个标签，放入队列中存储。

       引用队列：创建存放软引用的队列ReferenceQueue。

3、重写用到集合的方法：put、get、以及containsKey等用到的方法（即在实际使用中用到了哪些方法），在put方法中，需要先创建一个软引用对象，接收传入的value，即包裹强引用对象；将这个软引用加入到临时的集合中，这样就可以操作软引用的集合了。同样的get方法也是从这个临时存储软引用的集合中取值。

4、回收“空袋子”。当强引用对象被回收后，软引用也需要被回收掉：通过调用引用队列中的poll方法，不断的循环，检测队列是否为空，即检测队列中是否有“空袋子”软引用，如果有则从临时的集合中移除掉。在get和containKey方法中调用此回收方法。】

### 掌握OOM异常的处理，并可以对应用进行相应的优化

##### 一、内存溢出如何产生的

Android的虚拟机是基于寄存器的Dalvik，它的最大堆大小一般是16M，有的机器为24M。因此我们所能利用的内存空间是有限的。如果我们的内存占用超过了一定的水平就会出现OutOfMemory的错误。

内存溢出的几点原因总结：

1、资源释放问题：

程序代码的问题，长期保持某些资源（如Context）的引用，造成内存泄露，资源得不到释放

2、对象内存过大问题：

保存了多个耗用内存过大的对象（如Bitmap），造成内存超出限制

3、static：

static是Java中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（Context的情况最多），这时就要谨慎对待了。

public class ClassName {

     private static Context mContext;

     //省略

}

以上的代码是很危险的，如果将Activity赋值到mContext的话。那么即使该Activity已经onDestroy，但是由于仍有对象保存它的引用，因此该Activity依然不会被释放。

我们举Android文档中的一个例子。

private static Drawable sBackground;

@Override

      protected void onCreate(Bundle state) {

super.onCreate(state);

TextView label = new TextView(this);

label.setText("Leaks are bad");

if (sBackground == null) {

sBackground = getDrawable(R.drawable.large\_bitmap);

}

label.setBackgroundDrawable(sBackground);

setContentView(label);

}

    sBackground, 是 一个静态的变量，但是我们发现，我们并没有显式的保存Contex的引用，但是，当Drawable与View连接之后，Drawable就将View设 置为一个回调，由于View中是包含Context的引用的，所以，实际上我们依然保存了Context的引用。这个引用链如下：

    Drawable->TextView->Context

    所以，最终该Context也没有得到释放，发生了内存泄露。

针对static的解决方案：

第一、应该尽量避免static成员变量引用资源耗费过多的实例，比如Context。

    第二、Context尽量使用Application Context，因为Application的Context的生命周期比较长，引用它不会出现内存泄露的问题。

    第三、使用WeakReference代替强引用。比如可以使用WeakReference<Context> mContextRef;

    该部分的详细内容也可以参考Android文档中Article部分。

4、线程导致内存溢出：

线程产生内存泄露的主要原因在于线程生命周期的不可控。我们来考虑下面一段代码。

public class MyActivity extends Activity {

    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        new MyThread().start();

    }

    private class MyThread extends Thread{

@Override

        public void run() {

            super.run();

            //do somthing

        }

    }

}

这段代码很平常也很简单，是我们经常使用的形式。我们思考一个问题：假设MyThread的run函数是一个很费时的操作，当我们开启该线程后，将设备的横屏变为了竖屏，一 般情况下当屏幕转换时会重新创建Activity，按照我们的想法，老的Activity应该会被销毁才对，然而事实上并非如此。

    由于我们的线程是Activity的内部类，所以MyThread中保存了Activity的一个引用，当MyThread的run函数没有结束时，MyThread是不会被销毁的，因此它所引用的老的Activity也不会被销毁，因此就出现了内存泄露的问题。

有些人喜欢用Android提供的AsyncTask，但事实上AsyncTask的问题更加严重，Thread只有在run函数不结束时才出现这种内存泄露问题，然而AsyncTask内部的实现机制是运用了ThreadPoolExcutor,该类产生的Thread对象的生命周期是不确定的，是应用程序无法控制的，因此如果AsyncTask作为Activity的内部类，就更容易出现内存泄露的问题。

针对这种线程导致的内存泄露问题的解决方案：

    第一、将线程的内部类，改为静态内部类。

    第二、在线程内部采用弱引用保存Context引用。

##### 二、避免内存溢出的方案：

###### 1、图片过大导致内存溢出：

模拟器的RAM比较小，由于每张图片先前是压缩的情况，放入到Bitmap的时候，大小会变大，导致超出RAM内存

★android 中用bitmap 时很容易内存溢出，报如下错误：Java.lang.OutOfMemoryError : bitmap size exceeds VM budget

解决：

方法1： 主要是加上这段：等比例缩小图片

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;

1）通过getResource()方法获取资源：

       //解决加载图片 内存溢出的问题

        //Options 只保存图片尺寸大小，不保存图片到内存

       BitmapFactory.Options opts = new BitmapFactory.Options();

       //缩放的比例，缩放是很难按准备的比例进行缩放的，其值表明缩放的倍数，SDK中建议其值是2的指数值,值越大会导致图片不清晰

       opts.inSampleSize = 2;

       Bitmap bmp = null;

       bmp = BitmapFactory.decodeResource(getResources(), mImageIds[position],opts);

       ...

       //回收

       bmp.recycle();

2）通过Uri取图片资源

private ImageView preview;

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一

Bitmap bitmap = BitmapFactory.decodeStream(cr.openInputStream(uri), null, options);

preview.setImageBitmap(bitmap);

以上代码可以优化内存溢出，但它只是改变图片大小，并不能彻底解决内存溢出。

3）通过路径获取图片资源

private ImageView preview;

private String fileName= "/sdcard/DCIM/Camera/2010-05-14 16.01.44.jpg";

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一

Bitmap b = BitmapFactory.decodeFile(fileName, options);

preview.setImageBitmap(b);

filePath.setText(fileName);

方法2：对图片采用软引用，及时地进行recyle()操作

SoftReference<Bitmap> bitmap;

bitmap = new SoftReference<Bitmap>(pBitmap);

if(bitmap != null){

if(bitmap.get() != null && !bitmap.get().isRecycled()){

bitmap.get().recycle();

bitmap = null;

}

}

具体见“各种引用的简单了解”中的示例

###### 2、复用listView：

方法：对复杂的listview进行合理设计与编码：

Adapter中：

@Override

public View getView(int position, View convertView, ViewGroup parent) {

    ViewHolder holder;

    if(convertView!=null && convertView instanceof LinearLayout){

           holder = (ViewHolder) convertView.getTag();

    }else{

           convertView = View.inflate(MainActivity.this, R.layout.item, null);

           holder = new ViewHolder();

           holder.tv = (TextView) convertView.findViewById(R.id.tv);

           convertView.setTag(holder);

    }

    holder.tv.setText("XXXX");

    holder.tv.setTextColor(Color.argb(180, position\*4, position\*5, 255-position\*2));

    return convertView;

}

class ViewHolder{

       private TextView tv;

}

###### 3、界面切换

方法1：单个页面，横竖屏切换N次后 OOM

1、看看页面布局当中有没有大的图片，比如背景图之类的。

去除xml中相关设置，改在程序中设置背景图（放在onCreate()方法中）：

          Drawable bg = getResources().getDrawable(R.drawable.bg);

          XXX.setBackgroundDrawable(rlAdDetailone\_bg);

          在Activity destory时注意，bg.setCallback(null); 防止Activity得不到及时的释放

 2. 跟上面方法相似，直接把xml配置文件加载成view 再放到一个容器里

然后直接调用 this.setContentView(View view);方法，避免xml的重复加载

方法2： 在页面切换时尽可能少地重复使用一些代码

比如：重复调用数据库，反复使用某些对象等等......

###### 4、内存分配：

方法1：Android堆内存也可自己定义大小和优化Dalvik虚拟机的堆内存分配

    注意若使用这种方法：project build target 只能选择 <= 2.2 版本，否则编译将通不过。 所以不建议用这种方式

    private final static int CWJ\_HEAP\_SIZE= 6\*1024\*1024;

    private final static float TARGET\_HEAP\_UTILIZATION = 0.75f;

    VMRuntime.getRuntime().setMinimumHeapSize(CWJ\_HEAP\_SIZE);

    VMRuntime.getRuntime().setTargetHeapUtilization(TARGET\_HEAP\_UTILIZATION);

##### 常见的内存使用不当的情况

###### 1、查询数据库没有关闭游标

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。

Cursor cursor = null;

try {

  cursor = getContentResolver().query(uri ...);

  if (cursor != null && cursor.moveToNext()) {

  ... ...

  }

} finally {

  if (cursor != null) {

      try {

             cursor.close();

      } catch (Exception e) {

             //ignore this

      }

  }

}

###### 2、构造Adapter时，没有使用缓存的 convertView

以构造ListView的BaseAdapter为例，在BaseAdapter中提供了方法：

public View getView(int position, View convertView, 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：

public View getView(int position, View convertView, 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

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：

###### 4、释放对象的引用

当一个生命周期较短的对象A，被一个生命周期较长的对象B保有其引用的情况下，在A的生命周期结束时，要在B中清除掉对A的引用。

示例A：

public class DemoActivity extends Activity {

  ... ...

  private Handler mHandler = ...

  private Object obj;

  public void operation() {

  obj = initObj();

  ...

  [Mark]

  mHandler.post(new Runnable() {

       public void run() {

       useObj(obj);

       }

  });

  }

}

  我们有一个成员变量 obj，在operation()中我们希望能够将处理obj实例的操作post到某个线程的MessageQueue中。在以上的代码中，即便是 mHandler所在的线程使用完了obj所引用的对象，但这个对象仍然不会被垃圾回收掉，因为DemoActivity.obj还保有这个对象的引用。 所以如果在DemoActivity中不再使用这个对象了，可以在[Mark]的位置释放对象的引用，而代码可以修改为：

... ...

public void operation() {

  obj = initObj();

  ...

  final Object o = obj;

  obj = null;

  mHandler.post(new Runnable() {

  public void run() {

  useObj(o);

  }

  }

}

... ...

 示例B:

  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。

###### 5、其他

  Android应用程序中最典型的需要注意释放资源的情况是在Activity的生命周期中，在onPause()、onStop()、 onDestroy()方法中需要适当的释放资源的情况。由于此情况很基础，在此不详细说明，具体可以查看官方文档对Activity生命周期的介绍，以 明确何时应该释放哪些资源。

##### 三、Android性能优化的一些方案

###### 1、优化Dalvik虚拟机的堆内存分配

1）首先内存方面，可以参考 Android堆内存也可自己定义大小和优化Dalvik虚拟机的堆内存分配

对于Android平台来说，其托管层使用的Dalvik JavaVM从目前的表现来看还有很多地方可以优化处理，比如我们在开发一些大型游戏或耗资源的应用中可能考虑手动干涉GC处理，使用 dalvik.system.VMRuntime类提供的setTargetHeapUtilization方法可以增强程序堆内存的处理效率。当然具体原理我们可以参考开源工程，这里我们仅说下使用方法：

private final static floatTARGET\_HEAP\_UTILIZATION = 0.75f;

在程序onCreate时就可以调用：

VMRuntime.getRuntime().setTargetHeapUtilization(TARGET\_HEAP\_UTILIZATION);

2）Android堆内存也可自己定义大小

对于一些大型Android项目或游戏来说在算法处理上没有问题外，影响性能瓶颈的主要是Android自己内存管理机制问题，目前手机厂商对RAM都比较吝啬，对于软件的流畅性来说RAM对性能的影响十分敏感。

除了上次Android开发网提到的优化Dalvik虚拟机的堆内存分配外，我们还可以强制定义自己软件的对内存大小，我们使用Dalvik提供的 dalvik.system.VMRuntime类来设置最小堆内存为例:

private final static int CWJ\_HEAP\_SIZE = 6\* 1024\* 1024 ;

VMRuntime.getRuntime().setMinimumHeapSize(CWJ\_HEAP\_SIZE); //设置最小heap内存为6MB大小

当然对于内存吃紧来说还可以通过手动干涉GC去处理，我们将在下次提到具体应用。

###### 2、基础类型上，因为Java没有实际的指针，在敏感运算方面还是要借助NDK来完成。

Android123提示游戏开发者，这点比较有意思的是Google 推出NDK可能是帮助游戏开发人员，比如OpenGL ES的支持有明显的改观，本地代码操作图形界面是很必要的。

###### 3、图形对象优化：

这里要说的是Android上的Bitmap对象销毁，可以借助recycle()方法显示让GC回收一个Bitmap对象，通常对一个不用的Bitmap可以使用下面的方式，如

if(bitmapObject.isRecycled()==false) //如果没有回收

              bitmapObject.recycle();

###### 4、处理GIF动画：

目前系统对动画支持比较弱智对于常规应用的补间过渡效果可以，但是对于游戏而言一般的美工可能习惯了GIF方式的统一处理

目前Android系统仅能预览GIF的第一帧，可以借助J2ME中通过线程和自己写解析器的方式来读取GIF89格式的资源。

5、对于大多数Android手机没有过多的物理按键可能我们需要想象下了做好手势识别 GestureDetector 和重力感应来实现操控。通常我们还要考虑误操作问题的降噪处理。

##### 四、图片占用进程的内存算法简介

android中处理图片的基础类是Bitmap，顾名思义，就是位图。占用内存的算法如下：

图片的width\*height\*Config。

如果Config设置为ARGB\_8888，那么上面的Config就是4。一张480\*320的图片占用的内存就是480\*320\*4 byte。

在默认情况下android进程的内存占用量为16M，因为Bitmap除了java中持有数据外，底层C++的 skia图形库还会持有一个SKBitmap对象，因此一般图片占用内存推荐大小应该不超过8M。这个可以调整，编译源代码时可以设置参数。

##### 五、内存监测工具 DDMS --> Heap

  无论怎么小心，想完全避免bad code是不可能的，此时就需要一些工具来帮助我们检查代码中是否存在会造成内存泄漏的地方。Android tools中的DDMS就带有一个很不错的内存监测工具Heap(这里我使用eclipse的ADT插件，并以真机为例，在模拟器中的情况类似)。

用 Heap监测应用进程使用内存情况的步骤如下：

1. 启动eclipse后，切换到DDMS透视图，并确认Devices视图、Heap视图都是打开的；

2. 将手机通过USB链接至电脑，链接时需要确认手机是处于“USB调试”模式，而不是作为“Mass Storage”；

3. 链接成功后，在DDMS的Devices视图中将会显示手机设备的序列号，以及设备中正在运行的部分进程信息；

4. 点击选中想要监测的进程，比如system\_process进程；

5. 点击选中Devices视图界面中最上方一排图标中的“Update Heap”图标；

6. 点击Heap视图中的“Cause GC”按钮；

7. 此时在Heap视图中就会看到当前选中的进程的内存使用量的详细情况。

 说明：

a) 点击“Cause GC”按钮相当于向虚拟机请求了一次gc操作；

b) 当内存使用信息第一次显示以后，无须再不断的点击“Cause GC”，Heap视图界面会定时刷新，在对应用的不断的操作过程中就可以看到内存使用的变化；

c) 内存使用信息的各项参数根据名称即可知道其意思，在此不再赘述。

  如何才能知道我们的程序是否有内存泄漏的可能性呢。这里需要注意一个值：Heap视图中部有一个Type叫做data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在data object一行中有一列是“Total Size”，其值就是当前进程中所有Java数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：

a) 不断的操作当前应用，同时注意观察data object的Total Size值；

b) 正常情况下Total Size值都会稳定在一个有限的范围内，也就是说由于程序中的的代码良好，没有造成对象不被垃圾回收的情况，所以说虽然我们不断的操作会不断的生成很多对 象，而在虚拟机不断的进行GC的过程中，这些对象都被回收了，内存占用量会会落到一个稳定的水平；

c) 反之如果代码中存在没有释放对象引用的情况，则data object的Total Size值在每次GC后不会有明显的回落，随着操作次数的增多Total Size的值会越来越大，

  直到到达一个上限后导致进程被kill掉。

d) 此处已system\_process进程为例，在我的测试环境中system\_process进程所占用的内存的data object的Total Size正常情况下会稳定在2.2~2.8之间，而当其值超过3.55后进程就会被kill。

  总之，使用DDMS的Heap视图工具可以很方便的确认我们的程序是否存在内存泄漏的可能性。

### 熟悉Android中的动画，选择器，样式和主题的使用

##### 一、动画：

1、动画的分类：

1）、Tween动画：这种实现方式可以使视图组件移动、放大、缩小以及产生透明度的变化；

2）、Frame动画：传统的动画方法，通过顺序的播放排列好的图片来实现，类似电影。

###### 1）Frame 帧动画 AnimationDrawable

【参考api文档实现示例：/sdk/docs/guide/topics/resources/animation-resource.html#Frame】

1、使用AnimationDrawable来操作：

在res目录下，新建drawable与anim目录：

drawable放入帧动画图片

anim目录下新建帧动画xml文件来表示帧动画；

布局文件：

     <ImageView

        android:id="@+id/iv"

        android:onClick="start"

        android:layout\_width="wrap\_content"

        android:layout\_height="wrap\_content" />

帧动画文件rocket.xml：

      <?xml version="1.0" encoding="utf-8"?>

       <!-- oneshot 是否只播放一次 -->

       <animation-list xmlns:android="http://schemas.android.com/apk/res/android"

              android:oneshot="false" >

      <item  android:drawable="@drawable/girl\_1" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_2" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_3" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_4" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_5" android:duration="200"/>

     <item  android:drawable="@drawable/girl\_6" android:duration="200"/>

     <item  android:drawable="@drawable/girl\_7" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_8" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_9" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_10" android:duration="200"/>

      <item  android:drawable="@drawable/girl\_11" android:duration="200"/>

       </animation-list>

代码：

 public class MainActivity extends Activity {

       private ImageView iv;

       private AnimationDrawable anim;

       @Override

       protected void onCreate(Bundle savedInstanceState) {

              super.onCreate(savedInstanceState);

              setContentView(R.layout.activity\_main);

              iv = (ImageView) findViewById(R.id.iv);

              iv.setBackgroundResource(R.anim.rocket);                 // 把动画设置为背景

              anim = (AnimationDrawable) iv.getBackground();      // 获取背景

       }

       public void start(View v) {

              if(anim.isRunning()) {

                     anim.stop();

              }

              anim.start();

       }

}

###### 2）Tween动画：

①、有点类似以前弄的图片，处理，如旋转，缩放等，但Tween动画，注重的是动画过程，而不是结果；

②、创建方法：

使用xml文件来定义动画，然后通过AnimationUtils来加载，获取动画对象

使用代码方法，如：

       // 旋转动画(这里设置：围绕自己的中心点旋转)

       RotateAnimation ra = new RotateAnimation(0, 360, Animation.RELATIVE\_TO\_SELF, 0.5f, Animation.RELATIVE\_TO\_SELF, 0.5f);

       ra.setDuration(1500);                                        // 旋转一次时间

       ra.setRepeatCount(Animation.INFINITE);          // 重复次数无限

       iv\_scan.startAnimation(ra);                               // 开启动画

分类：

1、透明动画(alpha.xml)

<set xmlns:android="http://schemas.android.com/apk/res/android"

    android:shareInterpolator="false" >

    <!-- 透明动画 -->

    <alpha

        android:repeatMode="reverse"             // 反转,播放动画

          android:repeatCount="infinite"              // 重复播放

        android:duration="1000"

        android:fromAlpha="1"

        android:toAlpha="0.2" />

</set>

2、缩放动画(scale.xml)

    <!-- 缩放动画 -->

 <set>

      <scale

        android:duration="1000"

        android:fromXScale="1.0"                  // 起始x缩放级别,

        android:fromYScale="1.0"                  // 起始y缩放级别

          android:toXScale="2"                           // 目标x缩放级别, 这里设置为放大一倍

        android:toYScale="2"

        android:pivotX="0"                                   // 动画中心点设置；0 基于左上角；50%基于自身中央，50%p基于父容器中央, 大于0基于此像素

        android:pivotY="0"

        android:repeatCount="infinite"

        android:repeatMode="reverse"/>

</set>

3、位移动画(translate.xml)

    <!-- 位移动画 -->

    <translate

        android:duration="1000"

        android:fromXDelta="0"                            // 起始位移位置

        android:fromYDelta="0"

        android:repeatCount="infinite"

        android:repeatMode="reverse"

        android:toXDelta="100%"               // 移动到哪里，这里设置为，移动自身的右下角位置 100%

        android:toYDelta="100%" />

4、旋转动画(rotate.xml)

     <!-- 旋转动画 -->

    <rotate

        android:duration="1000"

        android:fromDegrees="0"            // 旋转角度范围设置

          android:toDegrees="360"

        android:pivotX="50%"                // 动画中心点设置

        android:pivotY="50%"

        android:repeatCount="infinite"

        android:repeatMode="restart"

       />

5、组合动画(all.xml)

      <!-- 组合动画：旋转 + 缩放 + 透明 -->

    <rotate

        android:duration="1000"

        android:fromDegrees="0"

        android:interpolator="@android:anim/linear\_interpolator"         // 动画篡改器，设置匀速转动，不出现完成后，停顿

        android:pivotX="50%"

        android:pivotY="50%"

        android:repeatCount="infinite"

        android:repeatMode="restart"

        android:toDegrees="360" />

    <scale

        android:duration="1000"

        android:fromXScale="1.0"

        android:fromYScale="1.0"

        android:pivotX="50%"

        android:pivotY="50%"

        android:repeatCount="infinite"

        android:repeatMode="reverse"

        android:toXScale="2"

        android:toYScale="2" />

    <alpha

        android:duration="1000"

        android:fromAlpha="1"

        android:repeatCount="infinite"

        android:repeatMode="reverse"

        android:toAlpha="0.2" />

代码：

 public class MainActivity extends Activity {

       private ImageView imageView;

       @Override

       protected void onCreate(Bundle savedInstanceState) {

              super.onCreate(savedInstanceState);

              setContentView(R.layout.activity\_main);

              imageView = (ImageView) findViewById(R.id.imageView);

       }

       public void onClick(View v) {

              Animation anim = null;         // 动画对象

              switch (v.getId()) {

                     case R.id.alphaBT:                              // 透明动画

                            anim = AnimationUtils.loadAnimation(this, R.anim.alpha);              // 根据xml获取动画对象

                            break;

                     case R.id.rorateBT:                             // 旋转动画

                            anim = AnimationUtils.loadAnimation(this, R.anim.rotate);

                            break;

                     case R.id.scaleBT:                               // 缩放动画

                            anim = AnimationUtils.loadAnimation(this, R.anim.scale);

                            break;

                     case R.id.transalteBT:                          // 位移动画

                            anim = AnimationUtils.loadAnimation(this, R.anim.translate);

                            break;

                     case R.id.all:

                            anim = AnimationUtils.loadAnimation(this, R.anim.all);

                            break;

              }

              if (anim != null) {

                     imageView.startAnimation(anim);        // 启动动画

              }

       }

}

###### 动画篡改器interpolator

       Interpolator 定义了动画的变化速度，可以实现匀速、正加速、负加速、无规则变加速等；有以下几类（更多参考API）：

AccelerateDecelerateInterpolator，延迟减速，在动作执行到中间的时候才执行该特效。

AccelerateInterpolator, 会使慢慢以(float)的参数降低速度。

LinearInterpolator，平稳不变的，上面旋转动画中使用到了；

DecelerateInterpolator，在中间加速,两头慢

CycleInterpolator，曲线运动特效，要传递float型的参数。

API Demo View 中有对应的动画插入器示例，可供参考；

###### xml实现动画插入器：

1、动画定义文件 /res/anim/目录下shake.xml ：

       <translate xmlns:android="http://schemas.android.com/apk/res/android"

                  android:duration="500"

                  android:fromXDelta="0"

                  android:interpolator="@anim/cycle\_3"

                  android:toXDelta="10" />

2、interpolator 指定动画按照哪一种方式进行变化, cycle\_3文件如下：

       <cycleInterpolator xmlns:android="http://schemas.android.com/apk/res/android" android:cycles="3" />

       表示循环播放动画3次；

3、使用动画的，程序代码：

      Animation shake = AnimationUtils.loadAnimation(this, R.anim.shake);

       et\_phone.startAnimation(shake);

##### 二、样式与主题

1、样式

1）、定义样式

设置样式，在values文件夹下的任意文件中的<resources>中配置<style>标签

<style name="itheima1">

    <item name="android:layout\_width">match\_parent</item>

    <item name="android:layout\_height">wrap\_content</item>

    <item name="android:textColor">#ff0000</item>

    <item name="android:textSize">30sp</item>

</style>

2）、继承样式，在<style>标签中配置属性parent

<style name="itheima2" parent="itheima1">

    <item name="android:gravity">center</item>

    <item name="android:textColor">#00ff00</item>

</style>

<style name="itheima3" parent="itheima2">

    <item name="android:gravity">right</item>

    <item name="android:textColor">#0000ff</item>

</style>

3）、使用样式

在layout文件的标签中配置style属性

<TextView

    style="@style/itheima1"

    android:text="一段文本" />

2、主题

styles.xml中也可以为Activity定义属性

<style name="AppTheme" parent="AppBaseTheme">

    <item name="android:windowNoTitle">true</item>

    <item name="android:windowFullscreen">true</item>

</style>

在AndroidManifest.xml文件中<activity>或者<application>节点上可以使用theme属性引用

<activity

    android:name="com.itheima.style.MainActivity"

    android:theme="@style/AppTheme" />

##### 三、选择器：

一）、创建xml文件:

在drawable/xxx.xml下常见xml文件，在同目录下记得要放相关图片

<?xml version="1.0" encoding="utf-8" ?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">

<!-- 默认时的背景图片-->

  <item android:drawable="@drawable/pic1" />

<!-- 没有焦点时的背景图片 -->

  <item android:state\_window\_focused="false"

        android:drawable="@drawable/pic1" />

<!-- 非触摸模式下获得焦点并单击时的背景图片 -->

  <item android:state\_focused="true" android:state\_pressed="true"   android:drawable= "@drawable/pic2" />

<!-- 触摸模式下单击时的背景图片-->

<item android:state\_focused="false" android:state\_pressed="true"   android:drawable="@drawable/pic3" />

<!--选中时的图片背景-->

  <item android:state\_selected="true"   android:drawable="@drawable/pic4" />

<!--获得焦点时的图片背景-->

  <item android:state\_focused="true"   android:drawable="@drawable/pic5" />

</selector>

二）使用xml文件：

1、使用方法：

1）、方法一：

（1）在listview中配置android:listSelector="@drawable/xxx

（2）在listview的item中添加属性android：background="@drawable/xxx"

2）、方法二：

Drawable drawable = getResources().getDrawable(R.drawable.xxx);

       ListView.setSelector(drawable);

但是这样会出现列表有时候为黑的情况，需要加上：

android:cacheColorHint="@android:color/transparent"使其透明。

2、相关属性：

android:state\_selected ：是选中

android:state\_focused ：是获得焦点

android:state\_pressed ：是点击

android:state\_enabled ：是设置是否响应事件,指所有事件

根据这些状态同样可以设置button的selector效果。也可以设置selector改变button中的文字状态。

3、Button文字效果

1）以下是配置button中的文字效果：

drawable/button\_font.xml

<?xml version="1.0" encoding="utf-8"?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:state\_selected="true" android:color="#FFF" />

    <item android:state\_focused="true" android:color="#FFF" />

    <item android:state\_pressed="true" android:color="#FFF" />

    <item android:color="#000" />

</selector>

2）Button还可以实现更复杂的效果，例如渐变

drawable/button\_color.xml

<?xml version="1.0" encoding="utf-8"?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">        /

<item android:state\_pressed="true">//定义当button 处于pressed 状态时的形态。

<shape>

<gradient  android:startColor="#8600ff" />

<stroke   android:width="2dp" android:color="#000000" />

<corners android:radius="5dp" />

<padding android:left="10dp" android:top="10dp"

android:bottom="10dp" android:right="10dp"/>

</shape>

</item>

<item android:state\_focused="true">//定义当button获得 focus时的形态

<shape>

<gradient android:startColor="#eac100"/>

<stroke android:width="2dp" android:color="#333333"  color="#ffffff"/>

<corners android:radius="8dp" />

<padding android:left="10dp" android:top="10dp"

android:bottom="10dp" android:right="10dp"/>

</shape>

</item>

</selector>

3）最后，需要在包含 button的xml文件里添加两项。

例如main.xml 文件，需要在<Button />里加两项

android：focusable="true"

android:background="@drawable/button\_color"

三）语法示例：

1、文件位置：

res/color/filename.xml，文件名被做资源的ID

2、语法示例

<?xml version="1.0" encoding="utf-8"?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:state\_selected="true" android:color="@color/white" />

    <item android:state\_focused="true" android:color="@color/white" />

    <item android:state\_pressed="true" android:color="@color/white" />

    <item android:state\_enabled="true" android:color="@color/black"/>

    <item android:state\_enabled="false" android:color="@color/white"/>

    <item android:state\_window\_focused="false" android:color="@color/black"/>

    <item android:color="@color/black" />

</selector>

3、属性

android:color：十六进制颜色，必须的。颜色是用RGB值来指定的，并且可选择alpha通道。

    这个值始终是用#字符开头，后面跟的是Appha-Red-Green-Blue信息，格式如下：

        #RGB

        #ARGB

        #RRGGBB

        #AARRGGBB

android:state\_pressed：一个布尔值

如果这个项目是在对象被按下时使用，那么就要设置为true。（如，按钮被触摸或点击时。）false应该用于默认的非按下状态。

android:state\_focused：一个布尔值

如果这个项目是在对象获取焦点时使用，那么就要设置为true。如，一个选项标签被打开时。

如果这个项目要用于对象没有被被选择的时候，那么就要设置为false。

android:state\_checkable：一个布尔值

如果这个项目要用于对象的可选择状态，那么就要设置为true。

如果这个项目要用于不可选状态，那么就要设置为false。（它只用于一个对象在可选和不可选之间的转换）。

android:state\_checked：一个布尔值

如果这个项目要用于对象被勾选的时候，那么就要设置为true。否者设为false。

android:state\_enabled：一个布尔值

如果这个项目要用于对象可用状态（接受触摸或点击事件的能力），那么就要设置为true，否者设置为false。

android:state\_window\_focused：一个布尔值

如果这个项目要用于应用程序窗口的有焦点状态（应用程序是在前台），那么就要设置为true，否者设置false。

4、注意

A：要记住，状态列表中一个与对象当前状态匹配的项目会被使用。因此，如果列表中的第一项没有包含以上任何一种状态属性，那么每次都会使用这个项目，因此默认设置应该始终被放到最后。

B：如果出现失去焦点，背景色延迟的情况，不要使用magin。

C：drawable下的selector可是设置状态背景列表(可以让view的背景在不同状态时变化)说明:也可以定义状态背景列表,但是是定义在drawable文件夹下,用的不是color属性,而是drawable属性。

##### 四）、自定义选择器

（shape和选择器如何同时使用。例如：如何让一个按钮即是圆角的，又能在点击的时候出现颜色变化。）

1、定义xml文件，Root Element选择shape

①创建view被按下的布局文件：

进行相应的属性配置，如：

<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"

    android:shape="rectangle" >

 <!-- 此处表示是一个矩形 -->

    <corners android:radius="3dp" />

 <!-- 此处表示是一个圆角 -->

    <solid android:color="#33000000" />

</shape>

②创建view正常显示的布局（新建一个xml同），配置如下：

<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"

    android:shape="rectangle" >

 <!-- 此处表示是一个矩形 -->

    <corners android:radius="3dp" />

 <!-- 此处表示是一个圆角 -->

    <solid android:color="#00000000" />

</shape>

2、创建背景选择器：（Root Element为selector）

<?xml version="1.0" encoding="utf-8"?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:drawable="@drawable/bg\_pressed" android:state\_pressed="true"/>

    <!-- pressed -->

    <item android:drawable="@drawable/bg\_pressed" android:state\_focused="true"/>

    <!-- focused -->

    <item android:drawable="@drawable/bg\_normal"/>

    <!-- 默认 -->

</selector>

3、将上面定义好的布局文件设定到选择器中（红字）

在需要使用背景资源的布局文件中选择上面创建的背景选择器（selector）

设置布局的clickable为true，并设置点击事件

此时在界面中点击控件时就会变化颜色

### 熟悉android系统下消息推送机制

##### 一、推送方式简介：

当前随着移动互联网的不断加速，消息推送的功能越来越普遍，不仅仅是应用在邮件推送上了，更多的体现在手机的APP上。当我们开发需要和服务器交互的应用程序时，基本上都需要获取服务器端的数据，比如《地震应急通》就需要及时获取服务器上最新的地震信息。

1、概念：

所谓的消息推送就是从服务器端向移动终端发送连接，传输一定的信息。比如一些新闻客户端，每隔一段时间收到一条或者多条通知，这就是从服务器端传来的推送消息；还比如常用的一些IM软件如微信、GTalk等，都具有服务器推送功能。

推送技术通过自动传送信息给用户，来减少用于网络上搜索的时间。它根据用户的兴趣来搜索、过滤信息，并将其定期推给用户，帮助用户高效率地发掘有价值的信息。

2、要获取服务器上不定时更新的信息，一般来说有两种方法：

第一种是客户端使用Pull（拉）的方式，就是隔一段时间就去服务器上获取一下信息，看是否有更新的信息出现。

第二种就是 服务器使用Push（推送）的方式，当服务器端有新信息了，则把最新的信息Push到客户端上。这样，客户端就能自动的接收到消息。

虽然Pull和Push两种方式都能实现获取服务器端更新信息的功能，但是明显来说Push方式比Pull方式更优越。因为Pull方式更费客户端的网络流量，更主要的是费电量，还需要我们的程序不停地去监测服务端的变化。

##### 二、常见消息推送方案的原理：

1、轮询(Pull)方式：

客户端定时向服务器发送询问消息，一旦服务器有变化则立即同步消息。应用程序应当阶段性的与服务器进行连接并查询是否有新的消息到达，你必须自己实现与服务器之间的通信，例如消息排队等。而且你还要考虑轮询的频率，如果太慢可能导致某些消息的延迟，如果太快，则会大量消耗网络带宽和电池。

2、SMS(Push)方式：

通过拦截SMS消息并且解析消息内容来了解服务器的命令。这个方案的好处是，可以实现完全的实时操作；但是问题是这个方案的成本相对比较高，且依赖于运营商。

3、持久连接(Push)方式：

客户端和服务器之间建立长久连接，这样就可以实现消息的及时行和实时性。

这个方案可以解决由轮询带来的性能问题，但是还是会消耗手机的电池。我们需要开一个服务来保持和服务器端的持久连接（苹果就和谷歌的C2DM是这种机制）。但是对于Android系统，当系统可用资源较低，系统会强制关闭我们的服务或者是应用，这种情况下连接会强制中断。（Apple的推送服务之所以工作的很好，是因为每一台手机仅仅保持一个与服务器之间的连接，事实上C2DM也是这么工作的。即所有的推送服务都是经由一个代理服务器完成的，这种情况下只需要和一台服务器保持持久连接即可。C2DM=Cloud to Device Messaging）。

　　相比之下第三种还是最可行的。为软件编写系统服务或开机启动功能；或者如果系统资源较低，服务被关闭后可以在onDestroy ()方法里面再重启该服务，进而实现持久连接的方式。

##### 三、消息推送解决方案概述

###### 1、C2DM云端推送方案

在Android手机平台上，Google提供了C2DM（Cloudto Device Messaging）服务。Android Cloud to Device Messaging (C2DM)是一个用来帮助开发者从服务器向Android应用程序发送数据的服务。该服务提供了一个简单的、轻量级的机制，允许服务器可以通知移动应用程序直接与服务器进行通信，以便于从服务器获取应用程序更新和用户数据。C2DM服务负责处理诸如消息排队等事务并向运行于目标设备上的应用程序分发这些消息。

C2DM操作过程示例图：

这个服务存在很大的问题：

1）C2DM内置于Android的2.2系统上，无法兼容老的1.6到2.1系统。

2）C2DM需要依赖于Google官方提供的C2DM服务器，由于国内的网络环境，这个服务经常不可用，如果想要很好的使用，我们的App Server必须也在国外，这个恐怕不是每个开发者都能够实现的。

3) 不像在iPhone中，他们把硬件系统集成在一块了。所以对于我们开发者来说，如果要在我们的应用程序中使用C2DM的推送功能，因为对于不同的这种硬件厂商平台，比如摩托罗拉、华为、中兴做一个手机，他们可能会把Google的这种服务去掉，尤其像在国内就很多这种，把Google这种原生的服务去掉。买了一些像什么山寨机或者是华为这种国产机，可能Google的服务就没有了。而像在国外出的那些可能会内置。

即然C2DM无法满足我们的要求，那么我们就需要自己来实现Android手机客户端与App Server之间的通信协议，保证在App Server想向指定的Android设备发送消息时，Android设备能够及时的收到。

###### 2、MQTT协议实现Android推送

　　采用MQTT协议实现Android推送功能也是一种解决方案。MQTT是一个轻量级的消息发布/订阅协议，它是实现基于手机客户端的消息推送服务器的理想解决方案。

　　wmqtt.jar 是IBM提供的MQTT协议的实现。我们可以从如下站点下载（http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24006006）它。可以将该jar包加入自己的Android应用程序中。可以从这里（https://github.com/tokudu/AndroidPushNotificationsDemo）下载该项目的实例代码，并且可以找到一个采用PHP书写的服务器端实现（https://github.com/tokudu/PhpMQTTClient）。

架构如下图所示：

###### 3、RSMB实现推送功能

　　Really Small Message Broker (RSMB) ，是一个简单的MQTT代理，同样由IBM提供，其查看地址是：http://www.alphaworks.ibm.com/tech/rsmb。缺省打开1883端口，应用程序当中，它负责接收来自服务器的消息并将其转发给指定的移动设备。SAM是一个针对MQTT写的PHP库。我们可以从这个http://pecl.php.net/package/sam/download/0.2.0地址下载它。send\_mqtt.php是一个通过POST接收消息并且通过SAM将消息发送给RSMB的PHP脚本。

###### 4、XMPP协议实现Android推送

1）XMPP：

XMPP全称Extensible Messaging and Presence Protocol，前身是Jabber项目，是一种以XML为基础的开放式即时通讯协议。XMPP因为被Google Talk和网易泡泡应用而被广大网民所接触。XMPP的关键特色是，分散式的即时通讯系统，以及使用XML串流。XMPP目前被IETF国际标准组织完成了标准化工作。

2）Android push notification(androidpn)

AndroidPN 是一个基于XMPP协议的java开源实现，它包含了完整的客户端和服务器端。该服务器端基本是在另外一个开源工程openfire基础上修改实现的。

androidpn实现意图如下图所示：

androidpn客户端需要用到一个基于java的开源XMPP协议包asmack，这个包同样也是基于openfire下的另外一个开源项目smack，不过我们不需要自己编译，可以直接把androidpn客户端里面的asmack.jar拿来使用。客户端利用asmack中提供的XMPPConnection类与服务器建立持久连接，并通过该连接进行用户注册和登录认证，同样也是通过这条连接，接收服务器发送的通知。

3）androidpn服务器端：

androidpn服务器端也是java语言实现的，基于openfire开源工程，不过它的Web部分采用的是spring框架，这一点与openfire是不同的。Androidpn服务器包含两个部分，一个是侦听在5222端口上的XMPP服务，负责与客户端的XMPPConnection类进行通信，作用是用户注册和身份认证，并发送推送通知消息。另外一部分是Web服务器，采用一个轻量级的HTTP服务器，负责接收用户的Web请求。服务器的这两方式，意义非凡：当相应的TCP端口被防火墙封闭，可以使用轮询的方式进行访问，因此又有助于通过防火墙。

服务器架构如下：

最上层包含四个组成部分，分别是SessionManager，Auth Manager，PresenceManager以及Notification Manager。SessionManager负责管理客户端与服务器之间的会话，Auth Manager负责客户端用户认证管理，Presence Manager负责管理客户端用户的登录状态，NotificationManager负责实现服务器向客户端推送消息功能。

这个解决方案的最大优势就是简单，我们不需要象C2DM那样依赖操作系统版本，也不会担心某一天Google服务器不可用。利用XMPP协议我们还可以进一步的对协议进行扩展，实现更为完善的功能。采用这个方案，我们目前只能发送文字消息，不过对于推送来说一般足够了，因为我们不能指望通过推送得到所有的数据，一般情况下，利用推送只是告诉手机端服务器发生了某些改变，当客户端收到通知以后，应该主动到服务器获取最新的数据，这样才是推送服务的完整实现。 XMPP协议书相对来说还是比较简单的，值得我们进一步研究。

4）androidpn不足：

androidpn是一个基于XMPP协议的java开源Android push notification实现。它包含了完整的客户端和服务器端。但也存在一些不足之处：

①、比如时间过长时，就再也收不到推送的信息了。

②、性能上也不够稳定。

③、如果将消息从服务器上推送出去，就不再管理了，不管消息是否成功到达客户端手机上。

如果我们要使用androidpn，则还需要做大量的工作，需要理解XMPP协议、理解Androidpn的实现机制，需要调试内部存在的BUG。

###### 5、使用第三方平台

　　目前国内、国外有一些推送平台可供使用，但是涉及到收费问题、保密问题、服务质量问题、扩展问题等等，又不得不是我们望而却步。

###### 6、自己搭建一个推送平台。

　　这不是一件轻松的工作，当然可以根据各自的需要采取合适的方案。

========================================================================================================================

##### 四、Android Push Notification实现信息推送使用

　AndroidPn项目就是使用XMPP协议实现信息推送的一个开源项目。在这里介绍其使用过程。

 1、Apndroid Push Notification的特点：

1）快速集成:提供一种比C2DM更加快捷的使用方式，避免各种限制.

2）无需架设服务器:通过使用"云服务"，减少额外服务器负担.

3）可以同时推送消息到网站页面，android 手机

4）耗电少，占用流量少.

2、具体配置过程：

1）下载并解压androidpn的压缩包

①、首先， 我们需要下载androidpn-client-0.5.0.zip和androidpn-server-0.5.0-bin.zip。

　　下载地址：http://sourceforge.net/projects/androidpn/

 ②、解压两个包，Eclipse导入client，配置好目标平台，打开raw/androidpn.properties文件，配置客户端程序。

2）配置：

A、如果是模拟器来运行客户端程序，把xmppHost配置成10.0.2.2[模拟器把10.0.2.2认为是所在主机的地址，127.0.0.1是模拟器本身的回环地址，10.0.2.1表示网关地址，10.0.2.3表示DNS地址，10.0.2.15表示目标设备的网络地址]，关于模拟器的详细信息，大家可参阅相关资料。

　　xmppPort=5222 是服务器的xmpp服务监听端口

 　　运行androidpn-server-0.5.0\bin\run.bat启动服务器，从浏览器访问http://127.0.0.1:7070/index.do (androidPN Server有个轻量级的web服务器，在7070端口监听请求，接受用户输入的文本消息)

　　运行客户端，客户端会向服务器发起连接请求，注册成功后，服务器能识别客户端，并维护和客户端的IP长连接。

 B、如果是在同一个局域网内的其他机器的模拟器测试(或者使用同一无线路由器wifi上网的真机) ，则需要把这个值设置为服务器机器的局域网ip

　　例如：你的电脑和android手机都通过同一个无线路由器wifi上网，电脑的ip地址为 192.168.1.2 而手机的ip地址为192.168.1.3，这个时候需要把这个值修改为xmppHost=192.168.1.1 或是电脑的IP地址，就可以在手机上使用了.

 C、如果是不在同一个局域网的真机测试，我们需要将这个值设置为服务器的IP地址。

具体配置如下图所示：

 3）示例：

　　我的电脑IP是：192.168.8.107

A、服务器运行主界面：

B、推送信息如下界面所示：

C、测试结果如下图所示：

熟悉掌握常见的设计模式：单例模式、工厂模式、策略模式、代理模式、装饰模式、模板模式

熟悉UML设计，可以设计程序的用例图、类图、活动图等

### 对Cocos2d游戏引擎有一定的了解和实践，并接触过处理3D图形和模型库的OpenGL

在进行游戏界面的绘制工作中，需要处理大量的工作，这些工作有很多共性的操作；并且对于游戏界面的切换，元素动作的处理，都已经有人做好了这些工作，并将其封装到框架中，其中Cocos2d-android就是这样一个框架。

Cocos2d实现游戏的绘制：

1、实现步骤：

首先来说，要想绘制出游戏界面，按照谷歌文档中的说明，需要实现两步操作：

①、所有的SurfaceView和SurfaceHolder.Callback，被UI Thread调用

也就是说需要接收用户的操作

②、确保所绘制的进程是有效的：

就要调用SurfaceHolder.Callback中的创建方法creat被调用和销毁方法destroy被调用

2、具体的实现：

1）、Cocos2d中有CCGLSurfaceView这个类，是继承于SurfaceView的，并实现了SurfaceHolder.Callback的接口。创建出这个对象，就有了绘制游戏界面的容器。

2）、绘制容器中的画面和元素，还要接受用户的操作；就需要将绘制的操作放在一个子线程中执行，UI Thread这个线程接收用户的操作；通过GLThread这个类实现不断的绘制界面的操作。

GLThread绘制线程的实现：

①、复写了run方法，在run方法中调用了GLThread自己的run方法：guardedRun

       此方法中，通过while(true)不停的绘制，其中有相应的标记进行控制

       绘制的方法：mRenderer.onDrawFrame(gl);【绘制一帧】

              【void org.cocos2d.opengl.GLSurfaceView.Renderer.onDrawFrame(GL10 gl)】

②、Canvas和GL10这个接口如何进行处理绘制的：

       在Canvas中，Bitmap和GL是互斥的，一个为null，另一个必须不为null

       Cocos2d底层用到的是OpenGL的信息，所以方法中传递的是gl的接口

③、GLThread的开启：

       @、在GLSurfaceView中的setRenderer方法中开启的：

mGLThread = new GLThread(renderer);

        mGLThread.start();

       @、在CCDirector（继承了GLSurfaceView.Renderer）的initOpenGLViewWithView方法中调用了setRenderer

       @、的调用是由attachInView(View view)方法返回的

       最终是由导演CCDirector进行调用，这是导演的第一个工作，

       attachInView(View view)的作用是将导演和SurfaceView进行绑定，绑定时，将绘制线程开启起来

（3）由此，大致过程如下：

①、创建出CCGLSurfaceView（即对应的SurfaceView），设置显示setContentView(surfaceView)

②、紧随其后，创建出导演CCDirector【通过单例获取：director=CCDirector.sharedDirector();】

③、通过调用导演中的attachInView(surfaceView)，传入surfaceView：

       这样就建立了CCDirector和SurfaceView之间的关系

       并且还开启了绘制线程，进行绘制：

              attachInView(View view)方法调用了initOpenGLViewWithView方法【都是导演中的方法】

              initOpenGLViewWithView方法调用了setRenderer【开启绘制线程用的】

              在setRenderer中创建了绘制线程，并开启起来

mGLThread = new GLThread(renderer);

               mGLThread.start();

3、界面元素的展示：

上面的操作只是创建出界面，可以不断绘制界面中的内容，要想丰富界面，就需要添加元素到界面中。

Cocos2的架构：

①、Cocos2D Graphic图形引擎②、CocosDenshion Audio声音引擎③、物理引擎④、Lua脚本库

其中对于图形引擎，在Cocos2d中，绘制游戏就相当于在拍电影

由导演类CCDirector控制这个游戏元素的展现和消失；其中还包括场景类CCScene和精灵类CCSprite

说明：

1）CCDirector（导演）：

引擎的控制者，控制场景的切换，游戏引擎属性的设置 【管理整棵大树】

2）CCScene （场景）：场景类

例如游戏的闪屏，主菜单，游戏主界面等。 【类似于树根，树干】

3）CCLayer（布景）：图层类

每个图层都有自己的触发事件，该事件只能对其拥有的元素有效，而图层之上的元素所包含的元素，是不受其事件管理的【类似于树枝】

4）CCSprite（人物）：精灵类，

界面上显示的最小单元【类似于树叶】

5）CCNode：

引擎中最重要的元素，所有可以被绘制的东西都是派生于此。它可以包含其它CCNode，可以执行定时器操作，可以执行CCAction。

                     CCScene，CCLayer，CCSprite的父类

6）CCAction（动作）：动作类

如平移、缩放、旋转等动作

示例代码：

public class MainActivity extends Activity {

       private CCDirector director;

       @Override

       protected void onCreate(Bundle savedInstanceState) {

              super.onCreate(savedInstanceState);

              //创建surfaceView

              CCGLSurfaceView surfaceView = new CCGLSurfaceView(this);

              setContentView(surfaceView);

              //创建导演

              director = CCDirector.sharedDirector();

              /\*    设置相关参数

               \*/

              //横屏显示

              director.setDeviceOrientation(CCDirector.kCCDeviceOrientationLandscapeLeft);

              //设置屏幕大小

              director.setScreenSize(480, 320);

              //显示帧率

              director.setDisplayFPS(true);

              //①建立CCDirector和SurfaceView之间的关系；开启绘制线程

              director.attachInView(surfaceView);

              /\*

               \* 管理显示内容

               \*/

              //创建场景

              CCScene scene = CCScene.node();

//            FirstLayer layer = new FirstLayer();

//            ActionLayer layer = new ActionLayer();

              DemoLayer layer = new DemoLayer();

              //添加场景中的layer

              scene.addChild(layer);

              director.runWithScene(scene);

       }

       @Override

       protected void onResume() {

              director.onResume();

              super.onResume();

       }

       @Override

       protected void onPause() {

              director.onPause();

              super.onPause();

       }

       @Override

       protected void onDestroy() {

              director.end();

              super.onDestroy();

       }

}

public class FirstLayer extends CCLayer {

       private static final String TAG = "FristLayer";

       private int count;

       public FirstLayer(){

              // 一个场景里面只能有一个layer可以处理用户的Touch

              this.setIsTouchEnabled(true);

              count = 0;

              init();

       }

       /\*

        \* 初始化

        \* 展示精灵并实现动画

        \*/

       private void init() {

              //创建精灵

              CCSprite sprite = CCSprite.sprite("z\_1\_01.png");

              this.addChild(sprite);

              sprite.setAnchorPoint(0, 0);

              CCSprite spritex = CCSprite.sprite("z\_1\_01.png");

              spritex.setFlipX(true);

              spritex.setAnchorPoint(0, 0);

              spritex.setPosition(100, 0);

              this.addChild(spritex, 0, 10);

              CCSprite spritey = CCSprite.sprite("z\_1\_01.png");

              spritey.setFlipY(true);

              spritey.setAnchorPoint(0, 0);

              spritey.setPosition(0, 100);

              this.addChild(spritey);

       }

       @Override

       public boolean ccTouchesBegan(MotionEvent event) {

              // 坐标转换：将MotionEvent封装的手机屏幕坐标系的坐标信息转换成Cocos2D的坐标系

              CGPoint touchPos= this.convertTouchToNodeSpace(event);

              CCSprite sprite = (CCSprite) this.getChildByTag(10);

              boolean containsPoint = CGRect.containsPoint(sprite.getBoundingBox(), touchPos);

              if(containsPoint){

//                   sprite.setOpacity(new Random().nextInt(255));

//                   count++;

//                   sprite.setVertexZ(1.0f+count);//最大132？

//                   Log.i(TAG, "count=="+count);

                     //移除精灵

//                   sprite.removeSelf();

                     //隐藏精灵

                     sprite.setVisible(false);

                     /\*

                      \* Tips:此处不能使用sprite.removeSelf();

                      \* 否则在第二次点击的时候，就会挂掉，因为再次点击的时候，精灵已经从layer中移除出去了

                      \*/

              }

              return super.ccTouchesBegan(event);

       }

}

### 有一定的屏幕适配经验

手机自适应主要分为两种情况：

横屏和竖屏的切换，以及分辨率大小不同。

##### 一、横竖屏切换：

1、Android应用程序支持横竖屏幕的切换，android中每次屏幕的切换动会重启Activity，所以应该在Activity销毁（执行onPause()方法和onDestroy()方法）前保存当前活动的状态；在Activity再次创建的时候载入配置，那样，进行中的游戏就不会自动重启了！有的程序适合从竖屏切换到横屏，或者反过来，这个时候怎么办呢？可以在配置Activity的地方进行如下的配置android:screenOrientation="portrait"（landscape是横向，portrait是纵向）。这样就可以保证是竖屏总是竖屏了。

2、而有的程序是适合横竖屏切换的。如何处理呢？首先要在配置Activity的时候进行如下的配置：

android:configChanges="keyboardHidden|orientation"，

另外需要重写Activity的onConfigurationChanged方法。

实现方式如下：

@Override

publicvoidonConfigurationChanged(ConfigurationnewConfig){

super.onConfigurationChanged(newConfig);

if(this.getResources().getConfiguration().orientation==Configuration.ORIENTATION\_LANDSCAPE){

//landdonothingisok

}elseif(this.getResources().getConfiguration().orientation==Configuration.ORIENTATION\_PORTRAIT){

//portdonothingisok

}

}

##### 二、分辨率问题：

对于分辨率问题，官方给的解决办法是创建不同的layout文件夹，这就需要对每种分辨率的手机都要写一个布局文件，虽然看似解决了分辨率的问题，但是如果其中一处或多处有修改了，就要每个布局文件都要做出修改，这样就造成很大的麻烦。那么可以通过以下几种方式解决：

一）使用layout\_weight

目前最为推荐的Android多屏幕自适应解决方案。

    该属性的作用是决定控件在其父布局中的显示权重，一般用于线性布局中。其值越小，则对应的layout\_width或layout\_height的优先级就越高（一般到100作用就不太明显了）；一般横向布局中，决定的是layout\_width的优先级；纵向布局中，决定的是layout\_height的优先级。

    传统的layout\_weight使用方法是将当前控件的layout\_width和layout\_height都设置成fill\_parent,这样就可以把控件的显示比例完全交给layout\_weight；这样使用的话，就出现了layout\_weight越小，显示比例越大的情况（即权重越大，显示所占的效果越小）。不过对于2个控件还好，如果控件过多，且显示比例也不相同的时候，控制起来就比较麻烦了，毕竟反比不是那么好确定的。于是就有了现在最为流行的0px设值法。看似让人难以理解的layout\_height=0px的写法，结合layout\_weight，却可以使控件成正比例显示，轻松解决了当前Android开发最为头疼的碎片化问题之一。

先看下面的styles（style\_layout.xml）

<?xml version="1.0" encoding="utf-8"?>

<resources>

<!-- 全屏幕拉伸-->

  <style name="layout\_full">

    <item name="android:layout\_width">fill\_parent</item>

    <item name="android:layout\_height">fill\_parent</item>

  </style>

<!-- 固定自身大小-->

  <style name="layout\_wrap">

    <item name="android:layout\_width">wrap\_content</item>

    <item name="android:layout\_height">wrap\_content</item>

  </style>

<!-- 横向分布-->

  <style name="layout\_horizontal" parent="layout\_full">

    <item name="android:layout\_width">0px</item>

  </style>

<!-- 纵向分布-->

  <style name="layout\_vertical" parent="layout\_full">

    <item name="android:layout\_height">0px</item>

  </style>

</resources>

可以看到，layout\_width和layout\_height两个属性被我封装成了4个style， 根据实际布局情况，选用当中的一种，不需要自己设置

二）清单文件配置：【不建议使用这种方式，需要对不同的界面写不同的布局】

需要在AndroidManifest.xml文件的<manifest>元素如下添加子元素

<supports-screensandroid:largeScreens="true"

android:normalScreens="true"

android:anyDensity="true"

android:smallScreens="true"

android:xlargeScreens="true">

</supports-screens>

以上是为我们的屏幕设置多分辨率支持（更准确的说是适配大、中、小三种密度）。

Android:anyDensity="true"，这一句对整个的屏幕都起着十分重要的作用，值为true，我们的应用程序当安装在不同密度的手机上时，程序会分别加载hdpi,mdpi,ldpi文件夹中的资源。相反，如果值设置为false，即使我们在hdpi,mdpi,ldpi，xdpi文件夹下拥有同一种资源，那么应用也不会自动地去相应文件夹下寻找资源。而是会在大密度和小密度手机上加载中密度mdpi文件中的资源。

有时候会根据需要在代码中动态地设置某个值，可以在代码中为这几种密度分别设置偏移量,但是这种方法最好不要使用，最好的方式是在xml文件中不同密度的手机进行分别设置。这里地图的偏移量可以在values-xpdi，values-hpdi,values-mdpi,values-ldpi四种文件夹中的dimens.xml文件进行设置。

三）、其他：

说明：

       在不同分辨率的手机模拟器下，控件显示的位置会稍有不同

       通过在layout中定义的布局设置的参数，使用dp（dip），会根据不同的屏幕分辨率进行适配

       但是在代码中的各个参数值，都是使用的像素（px）为单位的

技巧：

1、尽量使用线性布局，相对布局，如果屏幕放不下了，可以使用ScrollView（可以上下拖动）

ScrowView使用的注意：

在不同的屏幕上显示内容不同的情况，其实这个问题我们往往是用滚动视图来解决的，也就是ScrowView；需要注意的是ScrowView中使用layout\_weight是无效的，既然使用ScrowView了，就把它里面的控件的大小都设成固定的吧。

2、指定宽高的时候，采用dip的单位，dp单位动态匹配

3、由于android代码中写的单位都是像素，所有需要通过工具类进行转化

4、尽量使用9-patch图，可以自动的依据图片上面显示的内容被拉伸和收缩。其中在编辑的时候，灰色区域是被拉伸的，上下两个点控制水平方向的拉伸，左右两点控制垂直方向的拉伸

       工具在adt-bundle-windows-x86-20130522\sdk\tools目录下的draw9patch.bat

三、手机分辨率px和dp的关系：

dp：是dip的简写，指密度无关的像素。

       指一个抽象意义上的像素，程序用它来定义界面元素。一个与密度无关的，在逻辑尺寸上，与一个位于像素密度为160DPI的屏幕上的像素是一致的。要把密度无关像素转换为屏幕像素，可以用这样一个简单的公式：pixels=dips\*(density/160)。举个例子，在DPI为240的屏幕上，1个DIP等于1.5个物理像素。

强烈推荐你用DIP来定义你程序的界面布局，因为这样可以保证你的UI在各种分辨率的屏幕上都可以正常显示。

/\*\*

     \* 根据手机的分辨率从 px(像素) 的单位 转成为 dp

     \*/

    public static int px2dip(Context context, float pxValue) {

        final float scale = context.getResources().getDisplayMetrics().density;

        return (int) (pxValue / scale + 0.5f);

    }

/\*\*

     \* 根据手机的分辨率从 dip 的单位 转成为 px(像素)

     \*/

    public static int dip2px(Context context, float dpValue) {

        final float scale = context.getResources().getDisplayMetrics().density;

        return (int) (dpValue \* scale + 0.5f);

    }

对OAuth2认证有一定的了解

转到分享界面后，进行OAuth2认证：

以新浪为例：

第一步、WebView加载界面，传递参数

使用WebView加载登陆网页，通过Get方法传递三个参数：应用的appkey、回调地址和展示方式display(如手机设备为mobile)；

如：https://auth.sina.com.cn/oauth2/authorize?client\_id=1750636396&redirect\_uri=http://vdisk.weibo.com/&display=mobile

第二步、回调地址获取code

       当点击登陆（或授权）的时候，会将自定义的回调地址发送到相应的服务器端，这个回调地址只是为了从相应的服务器端（如新浪）获取到一个code；可以在WebViewClient的shouldOverrideUrlLoading方法中捕获到，然后获取到这个跳转的URL后，截取其中的code，如：http://vdisk.weibo.com/?code=3ea97ac6d5c1016a70d1c16e98b6f9ca

第三步、获取token

       通过这个code到相应的服务器获取到token【当然不仅仅是获取到token这个认证令牌，还有令牌有效期、uid，如果有权限的话，有的还会返回刷新令牌的token】，这些数据需要加密后保存在本地。

然后下次再登陆的时候，就可以直接登陆，然后通过发送给服务器端token等数据，获取到相应的数据

### 对Android底层有一定的认识，研究过相关的Android源码

我将从以下几方面简单说明：

##### 一、系统架构：

一）、系统分层：（由下向上）【如图】

1、安卓系统分为四层，分别是Linux内核层、Libraries层、FrameWork层，以及Applications层；

其中Linux内核层包含了Linux内核和各种驱动；

Libraries层包含各种类库（动态库（也叫共享库）、android运行时库、Dalvik虚拟机），编程语言主要为C或C++

FrameWork层大部分使用java语言编写，是android平台上Java世界的基石

Applications层是应用层，我们在这一层进行开发，使用java语音编写

2、Dalvik VM和传统JVM的区别：

传统的JVM：编写.java文件 à 编译为.class文件 à 打包成.jar文件

Dalvik VM：  编写.java文件 à 编译为.class文件 à 打包成.dex文件 à 打包成.apk文件(通过dx工具)

       将所有的类整合到一个文件中，提高了效率。更适合在手机上运行

1、Linux内核层[LINUX KERNEL]：

包含Linux内核和驱动模块（比如USB、Camera、蓝牙等）。

Android2.2（代号Froyo）基于Linux内核2.6版本。

2、Libraries层[LIBRARIES]：

这一层提供动态库（也叫共享库）、android运行时库、Dalvik虚拟机等。

编程语言主要为C或C++，所以可以简单的看成Native层。

3、FrameWork层[APPLICATION FRAMEWORK]：

这一层大部分用java语言编写，它是android平台上Java世界的基石。

4、Applications层[APPLICATION]：应用层

如图所示：

系统分层的图整体简化为下面的一张图，对应如下：

FrameWork层       --------à        Java世界

Libraries层           --------à        Native世界

Linux内核层         --------à        Linux OS

Java世界和Native世界间的通信是通过JNI层

JNI层和Native世界都可以直接调用系统底层

二）、系统编译：

1、主要步骤：系统环境的准备，下载源码、编译源码、输出结果：

目前系统的编译环境只支持Ubuntu 以及 Mac OS 两种操作系统，磁盘的控件要足够大

在下载源码的时候，由于Android源码使用Git进行管理，需要下载一些工具，如apt-get install git-core curl

源码下载好后，进行编译：首先搭建环境，部署JDK（不同的源码编译时需要的JDK版本不同，如2.2需要JDK5，2.3需要1.6），

                                          然后设置编译环境：使用. build/envsetup.sh脚本；选择编译目标（可以根据自己需要的版本进行不同的搭配）

                                          最后通过make –j4的命令进行编译。（make是编译的函数即命令，j4指的是cpu处理器的核数：单核的是j4 x i；双核的是j8）

最后将编译好的结果进行输出：所有的编译产物都位于 /out 目录下

2、编译流程图

##### 二、系统的启动：

通过Linux内核将Linux系统中用户空间的第一个进程init启动起来，这是安卓世界第一个被启动的进程；

然后在init中会加载init.rc的配置文件，并开启系统的守护进程（守护media（多媒体的装载）和孵化器zygote（Java世界的开启）），其实此时调试桥的守护进程也被开启起来了；

然后会处理一些动作执行，在app\_main.cpp中会将Zygote孵化器（Zygote是整个java世界的基础，整个安卓世界中（包括framework和app等apk）都是由孵化器启动的）启动起来：

在app\_main中，会调用AppRuntime的start方法开启AppRuntime，其实开启的是其父类AndroidRuntime的start方法被调用，zygote由此就被调用了，此时Native层的右上角有一块区域即ANDROID RUNTIME就启动起来了；

与此同时，AppRuntime会调用ZygoteInit的main方法启动ZygoteInit（整个的APPLICATION 和FRAMEWORK都会由ZygoteInit带起来的，JNI也被启动起来）：

在ZygoteInit中会调用SystemServer这个类，在SystemServer的main方法中启动init1()方法，将system\_init.cpp开启起来，在init1()方法中，将整个Native世界（即LIBRARIES层）开启起来了

然后在system\_init.cpp会调用SystemServer的init2()方法开启ServerThread，通过ServerThread将framework层开启起来（所有的就全部开启起来了），即java世界（APPLICATION FRAMEWORK）就被启动了；此时ActivityManager，WindowManager，PackageManager（最主要，所有的清单文件及apk都有它管理）等等framework层全部开启起来

一）安卓系统的总体启动顺序：

1、通过LINUX内核，将init进程启动起来（是Linux系统中用户空间的第一个进程）

2、将ANDROID RUNTIME这一块的内容启动完毕

3、分为两步分别启动LIBRARIES（即Native世界）和APPLICATION FRAMEWORK（即java世界）

1）先启动LIBRARIES（即Native世界）

2）后启动APPLICATION FRAMEWORK（即java世界）【ActivityManager，WindowManager，电源管理等等】

二）具体启动流程

一）、启动流程：

1、init进程：——安卓世界第一个被启动的进程

       加载一堆配置文件，核心加载的init.rc配置文件，其中包含了孵化器和守护进程都被开启了

1）、启动服务：开启ServerManager

守护进程启动（Daemon Process）：/system/bin/servicemanager

守护的是：

①、Java世界的开启：onrestart  restart  zygote

②、多媒体的装载：onrestart  restart  media

@、adbd的守护也被开启起来了，即调试桥的守护进程也被开启起来了

2）、启动孵化器Zygote

       在app\_main中启动孵化器Zygote，整个安卓世界中（包括framework和app等apk）都是由孵化器启动的

       【此时虚拟机还没开启起来，只是配置了一些vm的参数】

3、app\_main：——开启孵化器

       app\_main中，调用AppRuntime的start方法，将Native层的右上角有一块区域，即ANDROID RUNTIME启动起来

其中的start方法实际是其父类AndroidRuntime的start方法

【此时VM虚拟机被开启起来了，通过start方法开启，在AndroidRuntime中并设置了默认的内存大小16M】

【注册JNI，并启动孵化器Zygote】

4、ZygoteInit开启

       AppRuntime被启动后，会调用ZygoteInit的main方法，启动ZygoteInit；

       然后，整个的APPLICATION 和FRAMEWORK都会由ZygoteInit带起来的

5、SystemServer启动：

       ZygoteInit调用SystemServer这个类，在SystemServer的main方法中启动init1()方法，将system\_init.cpp开启起来

       在init1()方法中，将整个Native世界开启起来了

6、ServerThread启动（开启framework层）

调用SystemServer的init2()方法开启ServerThread，通过ServerThread将framework层开启起来（所有的就全部开启起来了）

       此时ActivityManager，WindowManager，PackageManager（最主要，所有的清单文件及apk都有它管理）等等framework层全部开启起来

二）、具体介绍：

1、启动入口：init进程

@、源码位置：/system/core/init/init.c

@、进程入口：main方法

1）创建文件夹，挂载设备【通过mkdir的命令创建，挂载一些系统设备后】

2）重定向输入输出，如错误信息输出【设置了一些输入输出的处理】

3）设置日志输出【一些系统的日志】

4）init.rc系统启动的配置文件【加载了相关的信息，不同版本的手机所特有的配置信息】

①、文件位置：/system/core/rootdir

②、守护进程启动（Daemon Process）：/system/bin/servicemanager

守护的是

Java世界的开启：onrestart  restart  zygote

       多媒体的装载：onrestart  restart  media

       adbd守护也被开启起来了，即调试桥（adb[Android Debug Bridge]）的守护进程（adbd[Android Debug Bridge Daemon]）也被开启起来了

③、启动Zygote——app\_main.cpp【Zygote是整个java世界的基础】

              当编译之后，在system/bin/app\_process下会有孵化器的启动Xzygote

              守护进程被开启之后，紧接着Zygote也被启动起来了

5）解析和当前设备相关的配置信息（/init.%s.rc）

Tips：

当解析完init.rc和设备配置信息后会获取到一系列Action

Init将动作的执行划分为四个阶段（优先级由大到小）:

early-init        ：初期

Init                ：初始化阶段

early-boot       ：系统启动的初期

boot               ：系统启动

6）处理动作执行：这个阶段Zygote将被启动

7）无限循环阶段，等待一些事情发生

2、Zygote简介：

@、Zygote启动：app\_main.cpp

1）Zygote简介：

①、本身为Native的应用程序

②、由init进程通过init.rc加载

2）功能分析：

①、Main方法中AppRuntime.start(),工作由父类AndroidRuntime来完成

②、在AndroidRuntime中开启了如下内容：

@startVM——开启虚拟机（查看堆内存设置）：默认16M【】

@注册JNI函数【此时还在Native层，需要将连接java和c的桥（即JNI）搭建好】

@启动“com.android.internal.os.ZygoteInit”的main方法

【系统级别的包（由runtime的start方法开启的这个包）】

start方法实际是其父类AndroidRuntime的

@进入java世界的入口

3、ServiceThread的简介：（java世界所做的事情）

1）preloadClasses();：预加载class

       读取一个preloaded-classes的配置文件

       此文件的内容非常多，这就是安卓系统启动慢的原因之一

       此时会有一个垃圾回收的操作gc()，将无用的回收掉

2）ZygoteInit在main方法中利用JNI开启com.android.server.SystemServer

3）启动system\_init.cpp处理Native层的服务

4）然后调用SystemServer的init2()

5）启动ServiceThread，启动android服务

6）Launcher启动

##### 三、开机时的时间消耗：

1、ZygoteInit.main()中会预加载类

目录：framework/base/preload-class

ZygoteInit.main()会加载很多的类，将近1800多个（安卓2.3的）

2、开机时会对系统所有的apk进行扫描

       需要将所有的应用展现给用户，就需要对apk进行扫描，扫描所有的包

       data目录下有个apk的包

       system目录下有个apk的包

       framework目录下也有相关的包

3、SystemServer创建的那些Service

##### 四、安卓工程的启动过程

1、Eclipse将.java源文件编译成.class

2、使用dx工具将所有.class文件转换为.dex文件

3、再将.dex文件和所有资源打包成.apk文件

4、将.apk文件安装到虚拟机完成程序安装

5、启动程序 – 开启进程 – 开启主线程

6、创建Activity对象 – 执行OnCreate()方法

7、按照main.xml文件初始化界面

=================================

### 应用程序启动：

###### 一、解析清单文件并加载

应用程序的启动需要从PackageManagerService说起，由于应用程序是有PackageManager管理的，可以简单认为PackageManagerService是为应用程序启动的做了一些准备工作，才能将应用程序开启起来。

1、PackageManagerService(资料)读取所有应用程序的Mainfest信息，并且建立信息库存储在系统级共享内存中

1）解析：

PackageManagerService在启动后，会进行解析的工作，它会重点监控一些文件：system/framework、system/app、data/app、data/app\_private；一旦将数据存入到这些文件中，就会去解析

2）权限分配：

PackageManagerService会建立底层userids和groupids同上层permissions之间的映射，就会给一些底层用户分配权限，

进行权限的映射，UID和GroupID，都会分配相应的权限

3）保存数据：

PackageManagerService还有重要的一个操作就是将解析的每个apk的信息保存到packages.xml和packages.list文件里，

在packages.list记录了如下数据:pkgName，userId，debugFlag，dataPath(包的数据路径)

【下次再开机的时候，不会再扫描每个apk了，只需要读取packages.xml和packages.list文件即可】

除了这两个主要的工作外，还会进行一些其他的操作，如检测文件等

2、Launcher就将PackageManagerService已经解析并处理好的数据都加载到内存中，从内存中就能获取到相应的数据，

并展示到手机上【之所以可以展示在手机桌面上，就是因为在清单文件中配置了如下的内容：】

              <action android:name="android.intent.action.MAIN" />：应用程序的入口

<category android:name="android.intent.category.LAUNCHER" />：配置了这个属性就可以显示在列表中

点击图标，应用就被开启起来了：

###### 二、Activity的启动与生命周期的监控

应用程序被开启后，是需要开启并创建Activity，加载相应的view，从而展示出应用程序

1、Activity是通过startActivity开启起来的，startActivity是由有Context调用的，其具体的实现类是ContextImpl

在ContextImpl中的startActivity方法中，会调用ActivityThread的相关方法【mMainThread.getInstrumentation().execStartActivity()】；可以追溯到Instrumentation这个类，其中的execStartActivity()的方法中实现了startActivity的调用：ActivityManagerNative.getDefault().startActivity，由此可以看出是底层进行处理。

2、ActivityMonitor监控Activity

当Activity实例创建的时候，就会给Activity配置一个监视器ActivityMonitor，监控Activity的声明周期：

在Instrumentation的execStartActivity()的方法中，上来先判断ActivityMonitor是否为null：在第一次开启Activity的时候，ActivityMonitor还是null的，就会调用ActivityManagerNative.getDefault().startActivity(…..)，是在操作native底层的信息，从而执行startActivity，再去开启一个Activity。

简单来说，就是通过调用JNI，调用startActivity方法，开启Activity；创建好了之后，随即也创建好了Activity的监视器ActivityMonitor

3、在ActivityMonitor中就有Activity各种生命周期的监控

①、在newActivity方法中：

可以通过拿到Activity的字节码，创建一个Activity，并将这个Activity返回

       还会调用attach方法，传入ActivityThread的线程

②、在各种生命周期的方法中，调用activity的各自的生命周期的方法

总结：

       1、通过PackageManagerService将所有用到的资源加载进内存中

       2、在Launcher中，将view等控件加载到ViewGroup中，点击每个item会有相应的操作

       3、在公开的文档中是找不到具体调用startActivity的类的，而是由系统完成调用的，实现了Activity的启动

实际就是通过Context的实现类ContextImpl进行调用的，一步步转到底层（ActivityManagerNative）实现调用

       4、另一个重要的类就是ActivityMonitor，监控Activity生命周期的；在其newActivity方法中创建了Activity，并调用了attach方法；

也就是说当一个Activity被创建的时候，就会绑定一个ActivityMonitor，用来监控Activity的生命周期

###### 三、应用程序启动的时序图：

对Activity、Window和View三者间的关系有一定的见解

一、简述如何将Activity展现在手机上

Tips：

Activity本身是没办法处理显示什么控件（view）的，是通过PhoneWindow进行显示的

换句话说：activity就是在造PhoneWindow，显示的那些view都交给了PhoneWindow处理显示

1、在Activity创建时调用attach方法：

2、attach方法中会调用PolicyManager.makeNewWindow()

实际工作的是IPolicy接口的makeNewWindow方法

①、其中创建了一个window（可以比喻为一个房子上造了一个窗户）：mWindow = PolicyManager.makeNewWindow(this);

②、在window这个类中，才调用了setContentView()，这是最终的调用

       在Activity的setContentView方法中，实际上是调用：getWindow().setContentView(view, params);

       这里的getWindow()就是获取到一个Window对象

Tips：

       为啥attch优先于onCreate调用，就是由于在attch方法中，会创建window，有了window才能调用setContentView

3、在IPolicy的实现类中创建了PhoneWindow：

①、由mWindow = PolicyManager.makeNewWindow(this);，

②、这里的makeNewWindow(this);方法中，返回的是：return sPolicy.makeNewWindow(context);

③、这个sPolicy实际是一个接口，其实现类是Policy，其中只是创建了一个PhoneWindow

4、在PhoneWindow的setContentView中向ViewGroup（root）中添加了需要显示的内容

①、PhoneWindow是继承Window的

②、setContentView这个方法中，需要先判断一个mContentParent是否为空，因为在默认进来的时候，什么都没创建呢

       此时需要创建：installDecor()，DecorView是最根上的显示的

       可以通过adt中的的tools中有个hierarchyviewer.bat的工具，可以查看手机的结构

③、DecorView：是继承与FrameLayout的，作为parent存在，最初显示的

④、下次再加载的时候，mContentParent就不为空了，会将其中的所有的view移除掉，然后在通过布局填充器加载布局

二、三者关系：

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()

# java

### 垃圾回收算法

1.标记-清除算法

标记-清除算法是最基本的算法，和他的名字一样，分为两个步骤，一个步骤是标记需要回收的对象。在标记完成后统一回收被标记的对象。这个算法两个问题。一个是效率问题，标记和清除的效率不高。第二个问题是标记-清除之后会有大量不连续的碎片空间，如果我们需要更大的连续内存就必须GC。

2.复制算法

复制算法，不同于标记-清除，复制算法大多数用于新生代，它需要大小相等的两块内存，每次只使用一块内存，当GC的时候会把这块内存存活的对象复制到另外一块内存上面，解决了时间效率和空间碎片问题。在新生代中会把他分为三个内存一个Eden 两个Survivor默认是8比1，开始会使用一个Eden和一个Surivivor装载内存，清除时会将这两个保留的对象都保存另外在Survivor中，并且年龄加1（以后提升为老年代），如果超出了Survivor中的限制会用老年代的内存担保。

3.标记-整理算法

主要特点是，解决了碎片问题。标记整理算法，标记过程和第一算法一样，但是他处理的时候会让存活的对象向一边移动解决了空间碎片问题，用于老年代的处理。

4.分代收集算法

分代收集算法不是新思想，只是把上面的算法结合起来了。

### AtomicInteger实现原理

AtomicInteger使用value来保存值,value是volatile的，保证了可见性。

对于get方法直接返回value，对于自增一或者添加值使用了CAS自旋锁，使用了一个死循环，如果cas返回为true就可以退出循环。对于CAS 全称是compare and swap比较和交换,CAS需要三个操作数，一个是变量内存地址，一个是excepted过期值，一个是现在要更新的值，我们操作的时候仅当V符合旧预期的值的时候才能更新我们新的。对于它的自增的操作，首先是卸载一个for循环里面然后获得当前的值，给这个值+1，然后进行cvs要是失败会再次Get()最新值再次写.

参考：http://ifeve.com/atomic-operation/

synchronized和lock的区别

主要有三个区别1、用法区别，性能区别，锁机制的区别。

（1）对于用法区别：synchronized可以在方法中上使用也可以在特定代码块中使用，括号中表示需要锁的对象，如果在方法上就是对该对象的锁，如果是在类的方法上就是类的锁，使用Lock必须自己用代码显示申明何时开启锁，何时关闭锁。synchronized是jvm的底层实现，而Lock是由代码执行。

（2）对于性能的区别：ReentrantLock 功能上要多于synchronized，多了锁投票，定时锁等。如果在小规模的竞争上synchronized效率比较高，如果在大规模的竞争上synchronize就比较低而Lock基本不变、

（3）锁的机制也不同：synchronized获得锁和释放锁都是在块中，都是自动释放，不会引起死锁，而Lock需要自己定位释放，不然会引起死锁。在Lock中也使用了tryLock方法用非阻塞的方式获取锁。

在lock中用一个锁变量和队列维护同步。

### gc停顿原因，如何降低GC停顿

原因：gc停顿的意思就像是在整个分析期间冻结在某个时间点上，具体的原因是防止在分析的时候，对象引用关系还在不断的变化，如果没有GC停顿很有可能分析不准确。

如何降低：在Serial的老年代垃圾收集器中，会把所有线程的暂停，停下来收集哪些是死亡对象。在CMS和G1中都采取了初始标记、并发标记、短暂GC停顿重新标记，初始标记会直接记录能GC ROOTS 关联的对象，在并发标记的时候有一个线程来标记，这个时候对象的发生的变化都会记录下来，在重新标记的时候会修正，这样就会降低GC停顿时间

### jvm如何调优，参数怎么调？如何利用工具分析jvm状态？

合理的分配内存，分配栈和堆的内存，在堆中我们还可以详细划分新生代和老年代的内存比例，在新生代中我们也可以划分Eden和Surivior的内存比例（调该比例大小），合理的划分内存区域大小，可以帮助我们jvm调优，我们采取合适的垃圾回收器，比如在新生代启用serial垃圾回收器，在老年代采用cms并发标记，可以降低GC停顿，当然也可以尝试去采用G1垃圾回收器

### jvm中类加载过程

类加载到类被卸载过程包括7个阶段

1.加载  通过类的全限定名把类文件的二进制流加入进来，通过这个字节流（这个二进制流也是我们代理类的方法），然后通过这个二进制流把静态存储结构转化为运行时方法区的结构（不包括类变量，类变量在准备阶段），在内 存中生成一个Class对象，作为方法区访问的入口。

2.验证  验证是验证Class文件的字节流包含的信息是否符合当前虚拟机的要求规范，防止恶意攻击、

3.准备  在方法区为类变量分配内存和设置初始值，这个时候的初始值是数据的0值，不是我们定义的值，如果是常量的话准备阶段就会设置为我们定义的值

4.解析  将符号引用(这里的符号引用指的是字面量的形式，只需要无歧义地定位到目标)替换为直接变量

5.初始化  类初始化 阶段是我们加载过程的最后一步，执行类构造器，合并static语句，有static的顺序决定。

6.使用

7.卸载

### Spring中bean的加载机制，bean生成具体步骤

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

spring中Bean的加载机制其实就是IOC容器的初始化，比如我们这里定义了一个IOC容器，BeanFactroy的子类ClassXMLPathApplicationContext,在他的构造函数中我们会把xml路径写进去这个步骤就是定位，接下来就是BeanDefiniton的载入，在构造函数当中有一个refresh()的函数，这个就是载入BeanDefinition的接口，这个方法进去之后是一个同步代码代码块，把之前的容器销毁和关闭创建了一个BeanFatroy，就像对我们的容器重新启动一样，然后我们对BeanDefiniton载入和解析解析完毕之后会把beanDefinition和beanName放入BeanFactory的HashMap中维护。在这里Bean已经被创建完成，然后我们就像IOC容器索要Bean，如果是第一次索要会触发依赖注入，会递归的调用gebBean实现依赖出入.

### 讲下java锁的原理

对于synchronized关键字，在jvm中在编译的时候在同步块的前后形成监视进入和监视退出两个字节码，这两个字节码都需要一个引用类型的参数来指定要锁定和解锁的对象，如果指定了的话就使用指定的对象，如果没有指定看是类方法还是对象方法来决定，在执行监视进入的指令的时候，会判断能否进入，进入成功之后会把锁计数器加1，如果不成功就会继续等待和其他的线程竞争，出锁的时候会把锁计数器减1变为0，也就是释放了锁。在这里要说明一点java的线程时映射到系统原生线程之上，如果要阻塞或者唤醒一个线程都需要操作系统帮忙，这就需要从用户态转换到核心态中，因此状态转换需要耗费很多的处理器时间。有可能比用户的代码执行时间还长。在jdk1.6之后对synchronized优化是非常的号的，比如锁粗化，锁自旋，锁消除。轻量级锁和偏向锁。

而对于ReentrantLock是代码上的实现

### 线程和进程的区别

进程是一段正在执行的程序，线程也叫作”轻量级进程“，他是程序执行的最小单元，一个进程可以有多个线程，各个线程之间共享程序的内存空间（比如说堆空间）及一些进程级的资源，进程和进程之间不能共享内存只能共享磁盘文件，线程也有4中状态：就绪，运行，挂起，死亡。

(新版本)进程是程序执行时的一个实例，从内核的观点看，进程的目的就是担当分配系统资源的基本单位。

线程是进程的一个执行流，是cpu调度和分配的基本单位，它是比进程更小的能独立运行的基本单位。一个进程由几个线程组成，线程和同属一个进程的其他的线程共享进程所拥有的全部资源。

进程-资源分配的最小单位。线程-程序执行的最小单位。

进程由独立的空间地址，线程没有单独的地址空间 同一进程内的共享进程的地址空间，只有自己独立的堆栈和局部变量。对于我们来说实现一个多线程的任务比实现一个多进程的任务好，

为什么分配线程比分配一般的对象更需要花费更大的代价？

首先他的资源非常“节俭”。我们知道，在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段，堆栈段和数据段，这是一种“昂贵”的多任务工作方式。而运行一个进程中的多个进程，他们彼此之间使用相同的地址空间，共享进程内的大部分数据，启动一个进程的时间远大于一个线程的时间。

(1)地址空间:进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;

(2)资源拥有:进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源

(3)线程是处理器调度的基本单位,但进程不是.

### Spring AOP是怎么实现

首先简单说一下Spring AOP几个比较重要的名词：

通知：定义在连接点做什么，为切面增强提供织入接口，意思就是说增强的内容以及增强的方式

切点：PointCut:决定通知应用于什么方法上或者类上。（点：方法或者类）

通知器：连接切点和通知结合起来，可以让我们知道那个通知应用于哪个结点之上，这个结合为应用使用Ioc容器配置AoP应用

开始我们使用AOP的时候是用得java编写的切面通知使用XML配置，后面我们摒弃了采用@AspectJ注解对其进行标注

然后AOP的实现原理有一个ProxyFactoryBean（代理工厂），这个ProxyFactoryBea是在Spring Ioc环境之中，创建AOP应用的最底层。在ProxyFactoryBean中我们会配置好通知器Advisor，在ProxyFactory需要为目标对象生成代理对象。ProxyFactory有一个getObject方法，在我们IOC容器中如果获取这个bean会自动调用这个方法，首先第一步初始化通知器链，通知器链只会初始化一次，使用标志位判断，遍历通知器，把所有通知器加入拦截器链，接下来就是代理对象的生成，利用目标对象以及拦截器我们可以正确的生成代理对象，这里生成代理对象有两种方法一种是jdk一种是cglib，在得到AopProxy代理对象之后，我们首先会根据配置来对拦截器是否与当前的调用方法想匹配，如果当前方法匹配就会发挥作用,他会遍历Proxy代理对象中设置拦截器链的所有拦截器，拦截器调用完成之后才是目标对象的调用，这个时候会有一个注册机制，在拦截器中运行的拦截器，会注册，我们就不需要再判断。Aop的源代码中也大量使用了IOC容器，比如从IOC中找到通知器。

### SpringMVC的主要流程

首先我们会在web.xml中配置DispathcerServlet，这个就是SpringMVC的入口,DispathcerServlet的父类FrameWorkServlet首先会初始化WebApplicationContext,DispacherServlet初始化了9个组件，初始完毕后我们开始进入，FramkeworkServlet中做了三件事一个是调用doService模板方法具体处理请求。将当前所有的请求都合并在一个方法里面和我们的httpServlet做法有点不同，在DispathcerServlet中有一个doService的方法，其中调用了doDispatch这也是最核心的首先根据request找到Handler,根据Handler找到了HderAdapter，用HandlerAdapter处理Handler其中包含一些参数的处理，处理完成后就行方法调用之后得到结果然后把View渲染给用户或者把数据发给用户。

详细版本：

1.输入一个网址，比如http请求，首先我们tomcat服务器，会对请求创建出我们request和response，然后就交给我们对应的servlet处理。

### 创建线程方式

实现runnable接口重写run方法，继承Thread,利用线程池来创建。

### 想让所有线程都等到一个时刻同时执行有哪些方法

CountDownLatch:CountDownLatch首先我们在构造函数当中传入了一个标志值，然后在需要阻塞的地方调用await()，直到其他线程把countDown减少完。这个是不可重用的。

CyclicBarrier：和他的名字一样栅栏，我们对他的构造函数传入一个栅栏值，在需要阻塞的地方调用await的时候我们就对其基础值加一，直到等于栅栏值。调用CylicBarrier的reset方法可以对他进行重置。

Semaphore信号量：Semaphore可以同时控制访问的线程个数，如果需要这个资源我们会acquire()以阻塞的方式去请求，如果没有可用的信号量，就等待,release释放信号量，这个机制有点类似于锁。

在jdk1.7中提供了一个同步器Phaser，作用和countdownLatch，CyclicBarrier类似，但PHaser的使用方式更为灵活。使用register注册方法递增计数器，使用arriveAndDeregister()方法来递减计数器，使用arriveAndAwaitAdvane()方法阻塞线程，当计数器归0唤醒。

### volatile的作用

参看博客：http://blog.csdn.net/libing13820393394/article/details/48582999

第一：volatile是Java虚拟机提供的最轻量级的同步机制，使变量对所有的线程可见，保证了可见性，但是并不能保证它的原子性。

第二个：禁止指令重排序优化。普通变量仅仅保证在该方法所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行一样。从硬件的方面来说，并不是指令任意重拍，他只是把多条指令不安程序规定顺序分发给电路处理单元，比如说2\*3+5 2\*3之间是有依赖，5就可以排到他们前面。volatile会帮助我们加入内存屏障防止重排序。volatile读操作性能消耗与普通变量几乎没区别，写操作会慢一些，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

注意：对于volatile修饰的变量，jvm只是保证从主内存加载到线程的工作的内存是最新的

### 谈一谈java内存模型

（1）java虚拟机规范试图定义一种JAVA内存模型来屏蔽掉各种硬件和操作系统的内存访问的差异。

（2）java内存模型的主要目标是定义程序中各个变量的访问规则，这里的变量不包含局部变量和方法参数，而是指的是实例字段、静态字段、和构成数组对象的元素。

（3）java内存模型规定了所有的变量都存储在主内存中，而线程内的局部变量在自己的工作内存中，并且还有被该线程使用到的变量的主内存 的副本拷贝，线程对变量的操作（读取、赋值）都在工作内存中进行，不能直接读写主内存的变量，不同的线程无法直接访问对方工作内存的变量，线程键的变量值的传递需要通过主内存来完成，在内存模型中比较重要的就是工作线程和主内存的交互。

### 内存之间的交互：

java内存模型定义的操作：

Lock (锁定)

Unlock（解锁）

Read（读取）

Load（载入）

Use（使用）

Assign（赋值）

Store（存储）

Write（写入）

变量从主内存到工作内存：按照顺序执行read load操作

变量从工作内存到主内存：按照顺序执行Store write操作

重排序：

包括：编译器优化重排序、指令级并行重排序、内存系统重排序

### 什么时候使用LinkedList?

首先分写LinkedList和ArrayList的不同，在经常插入和删除的时候，在实现栈和队列的时候，不适合随机查找元素。

Object有哪些方法(九大方法),clone是深复制还是浅复制，finalize一般在什么时候使用：

wait,notify,notifyall,clone,getclass,toString,equals,hashcode,finalize。

1、Clone()方法

private保护方法，实现对象的浅复制，只有类实现了Clonable接口才可以调用该方法，否则抛出CloneNotSupportException。clone是浅复制，复制完成后其中的变量引用还是和以前的一样，如果要实现深复制需要我们把所有的变量引用都递归复制一次，然后再赋值。（或者额使用序列化，也可以实现深拷贝）如果我们要自己实现clone()方法必须要实现克隆接口clonable。

2、Equals()方法

在object中与==是一样的，子类一般需要重写该方法

3、hashCode()方法

该方法用于哈希查找，重写了equals方法一般都要重写hashcode方法，这个方法在一些具有哈希功能的collection中使用

4、getClass()方法

final方法，获得运行时的类型

5、Wait()方法

使得当前的线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。Wait方法会一直等待，直到获得锁（到了睡眠的时间间隔也会唤醒自己）或者被中断掉。

调用该方法，当前的线程会进入到睡眠的状态，直到调用该对象的notify方法、notifyAll方法、调用interrupt中断该线程，时间间隔到了。

6、Notify()方法

唤醒在该对象上的等待的某个线程

7、notifyAll()方法

唤醒在该对象上的等待到所有的线程

8、toString()方法

把对象转换成string类型进行输出

9、finalize()方法

finalize在我们垃圾回收器回收这个对象的时候工作，可以做一些后续的工作，即进行一些必要的清理和清除的工作，比如说关闭流。当然我们也可以在这个里面对我们即将被回收的对象逃出回收。这里需要注意的是系统只会调用一次finalize()方法。但是一般我们不推荐使用这个方法，因为这个方法是为了对开始C和C++程序员的一种妥协，因为C中有析构函数，这个方法运行代价高，不确定大，我们还是会推荐使用try{}finally，他做的方法try{}finally都可以做。

### 如何管理线程（主要介绍各种线程池的实现）

使用线程池来管理线程

在Java中实现多种线程池

我们使用executors工厂产生我们的线程池，当线程池达到负载的时候会在我们线程池管理的Runnable阻塞队列中等待，不会像线程那样竞争CPU

第一种 newFixedThreadPool,和它的名字一样这是一个固定线程池，我们可以设置基本大小也就是我们没有任何任务执行的时候的大小，最大大小，只有在工作队列满了才能达到最大大小。

第二种 newCachedThreadPool这种线程池把大小设置为Integer.MAX\_VALUE,基本大小设置为0，空闲超时设置1分钟，这种线程池可以无限扩展，并且当需求降低时会自动收缩。

第三种 newSingleThreadPool这种线程池把基本大小设置为1，最大大小都设置为1，只允许同一时刻一个线程。

固定线程池和单线程池固定默认使用的是阻塞队列无界的LinkedBlockingQueue，在这个阻塞队列中可以无限增长。但是对于我们的newCachedThreadPool来说他的线程池是无限大的，不需要阻塞等待，我们这里使用的是SynchronousQueue来避免排队，其实这个东西不是一个队列，是直接在线程之间进行移交，当线程池的大小小于所需要的时候，要么创建一个要么拒绝一个。我们一般在使用的时候可以扩展，使用使用信号量来控制提交速率。

### 如何让线程A等待线程B结束

1.使用join方法可以等待A线程结束,或者单线程池中 阻塞队列的方式让A先获得单线程池的线程，然后B一直阻塞，知道A释放自己的线程。

### 如何优化jvm参数

，首先设置堆的大小，一般设置xmx和xms大小相同，如果老年代容易溢出可以扩充老年代，也要适当的调整永久代大小，选择自己合适的收集器，调整新生代对象年龄阀值等。

### 什么是守护线程

线程会分为两种：

普通线程和守护线程。在JVM启动时创建的所有线程中，除了主线程其他都是守护线程，比如说垃圾回收器就是守护线程，当普通线程全部退出的时候守护线程也会退出，我们自己也可以手动设置手动线程在线程启动之前，但是我们应该尽可能少使用守护线程，因为我们很少有操作可以在不进行清理就可以安全地抛弃，比如说I/O操作。

### TCP如何控制拥塞

 拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。

发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。

为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：

当cwnd<ssthresh时，使用慢开始算法。也就是乘法算法

当cwnd>ssthresh时，改用拥塞避免算法。也就是加法算法

当cwnd=ssthresh时，慢开始与拥塞避免算法任意。

当出现拥塞的时候就把心的门限值设为此时窗口大小的一般，窗口大小设置为1，再重新执行上面的步骤。

当收到连续三个重传的时候这就需要快重传和快恢复了，当收到连续三个重传 这个时候发送方就要重传自己的信息，然后门限减半但是这个时候并不是网络阻塞，窗口只会减半执行拥塞避免算法。

### ThreadLoacl？

我们使用ThreadLocal为每个使用该类型的变量提供了一个独立的副本，具体的实现是在每个线程中保存了一个ThreadLocalMap,这个ThreadLoaclMap会在我们第一次使用ThreadLoal中的set方法创建出来，set方法就是保存在ThreadLocalMap中，该变量为key，值为value,get方法也从这个HashMap中找。

### OSI网络模型？

网卡在哪一层（物理层）

交换机在哪一层（链路层）

路由器在哪一层（网络层）

传输TCP

会话 SQL

表示 IMG

html在应用层

### HTTP1.0和Http1.1区别？

1.0默认是多次tcp连接多次请求，然后增加了keep alive功能，但是必须在request Header手动增加Connection:keepalive

1.1是一次tcp连接多次请求，新的persistence功能

### POST和GET方法的区别？

长的说：

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

也就是说，GET只需要汽车跑一趟就把货送到了，而POST得跑两趟，第一趟，先去和服务器打个招呼“嗨，我等下要送一批货来，你们打开门迎接我”，然后再回头把货送过去。

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑！跳入需谨慎。为什么？

1. GET与POST都有自己的语义，不能随便混用。

2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

1.get是从服务器上获取数据，post是向服务器传送数据。

2.get是通过URL来传递数据，POST是通过表单传递，因此get数据限制在1024k,而POST没有限制

3.在java服务器端get是通过request.qureySting post通过request.getParameterNames和reque.getParameterValue

4.get是安全的，幂等的 POST即不安全又不幂等(多次操作和一次操作一样)

在rest中设计的话，一般get用来查询数据,POST用来添加数据,PUT用来更新数据,Delete用来删除数据

### filter执行顺序？

多个filter的执行顺序是web.xml中的配置顺序

影响SQL执行效率的因素？

1.is null和is not null

2.通配符的like

3.order by

4.not

5.in和exists

### GBK和UTF-8的区别

GBK包含全部中文字符； UTF-8则包含全世界所有国家需要用到的字符。

GBK的文字编码是双字节来表示的，即不论中、英文字符均使用双字节来表示，只不过为区分中文，将其最高位都定成1。

至于UTF－8编码则是用以解决国际上字符的一种多字节编码，它对英文使用8位（即一个字节），中文使用24位（三个字节）来编码。对于英文字符较多的论坛则用UTF－8节省空间。

UTF8是国际编码，它的通用性比较好，外国人也可以浏览论坛 GBK是国家编码，通用性比UTF8差，不过UTF8占用的数据库比GBK大~

### stringBuffer和StringBuilder组

1.三者在执行速度方面的比较：StringBuilder >  StringBuffer  >  String

看servlet和Filter的实现原理

StringBuffer是线程安全的，St不是线程安全的，内部的实现是使用char数

### 什么是rest

一次网站访问的全过程：

楼主提到TCP/IP分层的时候用的是[网络接口层](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E6%8E%A5%E5%8F%A3%E5%B1%82&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHbYryc1uHmLuhRLPhP-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDzP1mkn1nY)，那么楼主的TCP/IP分层概念应该是：应用层、传输层、网络层、[网络接口层](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E6%8E%A5%E5%8F%A3%E5%B1%82&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHbYryc1uHmLuhRLPhP-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDzP1mkn1nY)（包含了[七层模型](https://www.baidu.com/s?wd=%E4%B8%83%E5%B1%82%E6%A8%A1%E5%9E%8B&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHbYryc1uHmLuhRLPhP-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDzP1mkn1nY)中的数据链路层和物理层）。

我尝试回答一下楼主的问题，希望大家继续拍砖，如果访问[www.163.COM](http://www.163.com/)这个网站，那么他的主要过程应该是：

一、主机向[DNS服务器](https://www.baidu.com/s?wd=DNS%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHbYryc1uHmLuhRLPhP-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDzP1mkn1nY)发起域名解析请求，以得到相对应的IP地址

二、应用层应用HTTP协议发送数据

三、数据到达传输层封装成数据段，主机使用1024以后的随机源端口号，目标端口号为80

四、数据段到达网络层封装成数据包，加入主机源IP地址和目标IP地址

五、数据包到达[网络接口层](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E6%8E%A5%E5%8F%A3%E5%B1%82&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHbYryc1uHmLuhRLPhP-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDzP1mkn1nY)首先封装成数据帧，加入源MAC地址和目标MAC地址（注：此目标MAC地址为本地网关的MAC地址，源和目的MAC地址在数据转发的过程中，会由路由器不断的改变）。封装后将数据转换为物理层的数据流，通过互联网发送至目标服务器。

### 什么时候抛出InvalidMonitorStateException异常?为什么？

调用 wait ()/notify ()/notifyAll ()中的任何一个方法时，如果当前线程没有获得该对象的锁，

那么就会抛出 IllegalMonitorStateException 的异常

也就是说程序在没有执行对象的任何同步块或者同步方法时，

仍然尝试调用 wait ()/notify ()/notifyAll ()时。由于该异常是 RuntimeExcpetion 的子类，

所以该异常不一定要捕获(尽管你可以捕获只要你愿意

作为 RuntimeException，此类异常不会在 wait (),notify (),notifyAll ()的方法签名提及。

### Collections.synchronizedXX 方法的原理

返回了一个同步容器，在这个同步容器中的所有方法都有一个锁为当前对象或者指定锁的同步块，用这种阻塞同步的方法可以让我们容器同步

### 什么是Future

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

1.cancel方法用来取消任务

2.isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。

3.isDone()表示是否完成

4.get()获得执行结果，这个方法会一直阻塞

5.在时间范围内获取执行结果

FutureTask是Future的实现类

### [说出数据连接池的工作机制是什么?](http://blog.csdn.net/anglekill/article/details/6276257)

J2EE服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。

   调用：客户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其表记为 忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量有配置参数决定。

释放：当使用的池连接调用完成后，池驱动程序将此连接表记为空闲， 其他调用就可以使用这个连接。

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数来设定的。无论这些数据库连接是否被 使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超 过最大连接数量时，这些请求将被加入到等待队列中。

数据库连接池的最小连接数和最大连接数的设置要考虑到下列几个因素：

1) 最小连接数是连接池一直保持的数据库连接，所以如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费；

2) 最大连接数是连接池能申请的最大连接数，如果数据库连接请求超过此数，后面的数据库连接请求将被加入到等待队列中，这会影响之后的数据库操作。

3) 如果最小连接数与最大连接数相差太大，那么最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

### 存储过程和函数的区别

存储过程是用户定义的一系列sql语句的集合，涉及特定表或其它对象的任务，用户可以调用存储过程，而函数通常是数据库已定义的方法，它接收参数并返回某种类型的值并且不涉及特定用户表。

### 事务是什么？

事务是作为一个逻辑单元执行的一系列操作。

### 游标的作用？如何知道游标已经到了最后？

游标用于定位结果集的行，通过判断全局变量@@FETCH\_STATUS可以判断是否到了最后，通常此变量不等于0表示出错或到了最后。

### 系统进程间通信的方式

 管道( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

 命名管道 (named pipe) ： 命名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

 信号量( semophore ) ： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

 消息队列( message queue ) ： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

 信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

 共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

 套接字( socket ) ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

### jvm调优:内存溢出和内存泄露：

溢出解决：

1.在代码中减少不必要的实例构造

2.设置堆和永久代的大小 -xms堆最小 -xmx堆最大

内存泄露：

内存泄露不能通过配置解决代码的问题。比如资源在使用完毕后没有释放，一些对象存在无效引用我们不能回收。

### http和https的区别

http协议是无状态的明文传输，Https而SSL+HTTP协议构建的可进行加密传输。https的服务器必须向CA申请一个证明服务器用途的证书，而客户端通过该证书确认服务器，所以银行都是https，所有的通讯都是在密钥加密的情况下，而密钥则是通过证书交换，所以第三方拦截的数据没有密钥也没有用。

SSL用以保障在Internet上数据传输之安全，利用[数据加密](http://baike.baidu.com/view/696431.htm)(Encryption)技术，可确保数据在网络上之传输过程中不会被截取及窃听。

### 虚拟机性能监控状况

jps:显示系统内所有进程的信息。

jstat:收集虚拟机各种运行状态信息的命令行工具。 -gc,监视java堆 -class 就是监视类加载，还可以监视编译状况。

jinfo:java配置信息工具。

jmap:用于生成堆转储快照 有些选项只能在linux下才能看见。

jhat:配合jmap。

jstack:堆栈追踪。

### Servlet生命周期

1.加载：在Servlet容器启动的时候会通过类加载器加载我们的Servlet的Class文件。

2.创建：在创建过程的时候如果没有在web.xml文件中使用load-on-startup我们在第一次访问我们的Servlet的时候会初始化实例，如果配置了这个并且大于1会在容器启动的时候就创建。

3.初始化：init()初始化的方法只会被调用一次。在我们实例化被创建之后就会执行初始化。

4.处理客户请求：service()在Servlet的service方法中会根据不同的http方法来调用。

5.卸载：destroy()当我们Servlet需要被卸载的时候就会调用我们的destory()方法，随后垃圾回收器会回收。

### Minor GC和FULL GC

当我们需要向新生代中分配内存时出现不足的情况：会出现Minor GC,在新生代中 都是朝生夕灭的对象，频率比较高。Minor GC发生在新生代。

FULL GC指在老年代发生的GC，一般情况下出现了FULL GC都会伴随着一次Minor GC。

为什么MinorGC 和FULL GC速度有差距呢？

在Minor GC中使用的copy算法，在FullGC中使用的是标记清除 或者  标记整理算法。

copy算法是用空间换时间 mark（标记）和copy（复制）是一个动作。

但是mark-sweap或mark-compact都是分为两个阶段，先标记再清除是两步骤。

所以Minro GC速度会快于FullGC。

### JVM调优问题

对于JVM调优的重点是垃圾回收和内存管理。

垃圾回收我们可以使用我们的cms垃圾回收器。

对于内存管理有：

永久代溢出、栈溢出、堆溢出的情况

永久代溢出：

针对永久代溢出在JVM默认情况只有64M的永久代大小，很多东西都需要我们永久代区内存，尤其是使用Spring等框架的时候会有cglib的动态字节码生成class，都会存储在我们的永久代。所以我们需要扩充永久代防止内存溢出。

堆溢出：

对于堆溢出我们也是比较常见的比如说我们使用容器的时候没有释放内存很有可能就会导致堆溢出，需要动态扩展。

栈溢出：

对于栈我们也可以设置提高。

### 单例模式

##### （1）恶汉式的单例模式

利用静态static的方式进行实例化，在类被加载时就会创建实例。

/\*\*

 \* 饿汉式实现单例模式

 \*/

public class Singleton {

    private static Singleton instance = new Singleton();//在类加载时实例单例对象

    private Singleton() {//私有的构造器

    }

    public static Singleton getInstance() {//返回一个实例对象

        return instance;

    }

}

##### (2)懒汉式实现单例模式

在被第一次引用时才去创建对象。

/\*\*

 \* 懒汉式实现单例模式

 \*/

public class Singleton {

    private static Singleton instance;//创建私有的静态变量

    private Singleton() {//私有的构造函数

    }

        // synchronized方法,多线程情况下保证单例对象唯一

public static synchronized Singleton getInstance() {

//如果实例对象为空，就重新去实例化

        if (instance == null) {

            instance = new Singleton();

        }

        return instance;

    }

}

分析：这中方法的实现，效率不高，因为该方法定义为同步的方法。

##### (3)双重锁实现的单例模式double check

/\*\*

 \* DCL实现单例模式

 \*/

public class Singleton {

    private static valotile Singleton instance = null;//这里要加入valotile关键字，避免指令重排序，  可能先赋值但是没有分配内存

    private Singleton() {

    }

    public static Singleton getInstance() {

        // 两层判空，第一层是为了避免不必要的同步

        // 第二层是为了在null的情况下创建实例

        if (instance == null) {

            synchronized (Singleton.class) {

                if (instance == null) {

                    instance = new Singleton();

                }

            }

        }

        return instance;

    }

}

分析：资源的利用率较高，在需要的时候去初始化实例，而且可以保证线程的安全，该方法没有去进行同步锁，效率比较好。

##### (4)静态内部类实现单例模式

/\*\*

 \* 静态内部类实现单例模式

 \*/

public class Singleton {

    private Singleton() {

}

//返回实例的方法

    public static Singleton getInstance() {

        return SingletonHolder.instance;

    }

    /\*\*

     \* 静态内部类

     \*/

private static class SingletonHolder {

    //静态私有的实例对象

        private static Singleton instance = new Singleton();

    }

}

分析：第一次加载类时不会去初始化instance,只有第一次调用getInstance()方法时，虚拟机才会加载内部类，初始化instance

可以保证线程的安全，单例对象的唯一，延迟了单例的初始化。

##### (5)枚举单例

/\*\*

 \* 枚举实现单例模式

 \*/

public enum SingletonEnum {

    INSTANCE;//直接定义一个实例对象

    public void doSomething() {

        System.out.println("do something");

    }

}

分析：枚举实例的创建是线程安全的，即使反序列化也不会生成新的实例，在任何的情况下都是单例的。

### 设计模式6大原则

1.单一职责。2.里氏替换 3.依赖导致4接口隔离5.迪米特法则6.开闭原则。

### XML和JSON优缺点

XML的优点

　　A.格式统一，符合标准；

　　B.容易与其他系统进行远程交互，数据共享比较方便。

      C.可读性高

XML的缺点

　　 A.XML文件庞大，文件格式复杂，传输占带宽；

　　 B.服务器端和客户端都需要花费大量代码来解析XML，导致服务器端和客户端代码变得异常复杂且不易维护；

　　 C.客户端不同浏览器之间解析XML的方式不一致，需要重复编写很多代码；

　　 D.服务器端和客户端解析XML花费较多的资源和时间。

JSON的优缺点

JSON的优点：

　　A.数据格式比较简单，易于读写，格式都是压缩的，占用带宽小；

　　B.易于解析，客户端JavaScript可以简单的通过eval()进行JSON数据的读取；

      C.支持多种语言，包括ActionScript, C, C#, ColdFusion, Java, JavaScript, Perl, PHP, Python, Ruby等服务器端语言，便于服务器端的解析；

　　D.在PHP世界，已经有PHP-JSON和JSON-PHP出现了，偏于PHP序列化后的程序直接调用，PHP服务器端的对象、数组等能直接生成JSON格式，便于客户端的访问提取；

　　E.因为JSON格式能直接为服务器端代码使用，大大简化了服务器端和客户端的代码开发量，且完成任务不变，并且易于维护。

JSON的缺点

　　A.没有XML格式这么推广的深入人心和喜用广泛，没有XML那么通用性；

　　B.JSON格式目前在Web Service中推广还属于初级阶段。

       C：可读性低。

### 四种读取XML文件读取的办法

##### 1.DOM生成和解析XML文档

为XML文档的已解析版本定义了一组接口。解析器读入整个文档，然后构建一个驻留内存的树结构。

优点：整个文档树在内存中，便于操作；支持删除，修改，重新排列等。

缺点:  把整个文档调入内存，存在很多无用的节点，浪费了时间和空间。

##### 2.SAX为解决DOM

1、边读边解析，应用于大型XML文档

2、只支持读

3、访问效率低

4、顺序访问

##### 3.DOM4J生成和解析XML文档(解析工具) 性能最好 SUM的JAXM也大量采用DOM4J

HIBERNATE采用DOM4J

虽然DOM4J代表了完全独立的开发结果，但最初，它是JDOM的一种智能分支。它合并了许多超出基本XML文档表示的功能，包括集成的XPath支持、XML Schema支持以及用于大文档或流化文档的基于事件的处理。它还提供了构建文档表示的选项，它通过DOM4J API和标准DOM接口具有并行访问功能。从2000下半年开始，它就一直处于开发之中。

    为支持所有这些功能，DOM4J使用接口和抽象基本类方法。DOM4J大量使用了API中的Collections类，但是在许多情况下，它还提供一些替代方法以允许更好的性能或更直接的编码方法。直接好处是，虽然DOM4J付出了更复杂的API的代价，但是它提供了比JDOM大得多的灵活性。

    在添加灵活性、XPath集成和对大文档处理的目标时，DOM4J的目标与JDOM是一样的：针对Java开发者的易用性和直观操作。它还致力于成为比JDOM更完整的解决方案，实现在本质上处理所有Java/XML问题的目标。在完成该目标时，它比JDOM更少强调防止不正确的应用程序行为。

    DOM4J是一个非常非常优秀的Java XML API，具有性能优异、功能强大和极端易用使用的特点，同时它也是一个开放源代码的软件。如今你可以看到越来越多的Java软件都在使用DOM4J来读写XML，特别值得一提的是连Sun的JAXM也在用DOM4J.

##### 4.JDOM

JDOM

优点:①是基于树的处理XML的Java API，把树加载在内存中

        ②没有向下兼容的限制，因此比DOM简单

        ③速度快，缺陷少

        ④具有SAX的JAVA规则

缺点:

     ①不能处理大于内存的文档

     ②JDOM表示XML文档逻辑模型。不能保证每个字节真正变换。

     ③针对实例文档不提供DTD与模式的任何实际模型。

     ④不支持与DOM中相应遍历包。

最适合于:JDOM具有树的便利，也有SAX的JAVA规则。在需要平衡时使用

### 如何防止Sql注入

有两种办法

1.第一种消毒，通过正则匹配过滤请求数据中可能注入的SQL。

2.使用预编译手段preparedStatemengt。

### DB第一范式，第二范式，第三范式

第一范式：没一列属性不可再分,没有多值属性

第二范式：在符合第一范式的基础上，存在主键

第三范式：在符合第二范式的基础上，非关键字独立于其他的非关键字，并且依赖关键字。不能存在传递依赖。

### public、protected、private、默认权限

private:用于修饰类和方法，只允许该类访问。

默认：只允许在同一个类和同一个包中进行访问。

protected:用于修饰类和方法，允许该类和子类访问以及同一个包中访问。

public:用于修饰类和方法，允许该包下面和其他包的访问，即在全局范围都可以访问。

### 数据库事务

事务的特性：

原子性：事务是不可再分的；

一致性：事务的实行前后，数据库的状态保持一致；

隔离性：事务的并发访问，事务之间的执行互不干扰；

持久性：事务结束后数据永久保存在数据库中。

### 什么是脏读？

脏读就是一个事务读取了该数据并且对该数据做出了修改，另一个事务也读取了该修改后的数据但是前一个事务并没有提交，这是脏数据。

读取到保存在数据库内存中的数据。

### 什么是不可重复读？

一个事务：在同一个事务中读取同一数据，得到的内容不同。一个事务读取另外一个事务更新的数据，导致二次的查询的数据不一致。

### 什么是幻读？

幻读是当事务不独立发生的。比如一个事务删除了所有数据，另一个事务又插入了一条，那么第一个事务的用户会发现表中还没有修改的数据行。一个事务读取到另外一个事务提交的数据，导致查询的结果不一致的问题。

### 数据库的隔离级别：

Read uncommitted：未提交读：三中都有可能发生

Read committed :已提交读 避免脏读

Repeated read:重复读：避免脏读 不可重复读

Serializable:串行化读   都可以避免

### WebService到底是什么

         一言以蔽之：WebService是一种跨编程语言和跨操作系统平台的远程调用技术。

 所谓跨编程语言和跨操作平台，就是说服务端程序采用java编写，客户端程序则可以采用其他编程语言编写，反之亦然！跨操作系统平台则是指服务端程序和客户端程序可以在不同的操作系统上运行。

         所谓远程调用，就是一台计算机a上的一个程序可以调用到另外一台计算机b上的一个对象的方法，譬如，银联提供给商场的pos刷卡系统，商场的POS机转账调用的转账方法的代码其实是跑在银行服务器上。再比如，amazon，天气预报系统，淘宝网，校内网，百度等把自己的系统服务以webservice服务的形式暴露出来，让第三方网站和程序可以调用这些服务功能，这样扩展了自己系统的市场占有率，往大的概念上吹，就是所谓的SOA应用。

         其实可以从多个角度来理解WebService，从表面上看，WebService就是一个应用程序向外界暴露出一个能通过Web进行调用的API，也就是说能用编程的方法通过Web来调用这个应用程序。我们把调用这个WebService的应用程序叫做客户端，而把提供这个WebService的应用程序叫做服务端。从深层次看，WebService是建立可互操作的分布式应用程序的新平台，是一个平台，是一套标准。它定义了应用程序如何在Web上实现互操作性，你可以用任何你喜欢的语言，在任何你喜欢的平台上写Web service ，只要我们可以通过Web service标准对这些服务进行查询和访问。

   WebService平台需要一套协议来实现分布式应用程序的创建。任何平台都有它的数据表示方法和类型系统。要实现互操作性，WebService平台必须提供一套标准的类型系统，用于沟通不同平台、编程语言和组件模型中的不同类型系统。Web service平台必须提供一种标准来描述Web service，让客户可以得到足够的信息来调用这个Web service。最后，我们还必须有一种方法来对这个Web service进行远程调用,这种方法实际是一种远程过程调用协议(RPC)。为了达到互操作性，这种RPC协议还必须与平台和编程语言无关。

### java中锁的优化

1.减少锁持有的时间，可以减少其它线程的等待时间，不能让一个线程一直控制着某个锁不释放，导致竞争加剧。

2.减少锁的粒度，合适的锁的代码块，可以减少竞争，控制锁的范围。

3.锁分离，将锁安功能划分，比如读写锁，读读不互斥，读写互斥，写写互斥，保证了线程的安全，提高了性能。比如阻塞队列中的take和put

4.锁粗化，如果对同一个锁不停的进行请求，同步和释放，这个消耗是非常的大的，所以适当的时候可以粗化。

5.锁消除，编译器可以帮助我们优化比如一些代码根本不需要锁。

### 虚拟机内的锁优化

1.偏向锁:偏向当前已经占有锁的线程，在无竞争的时候，之前获得锁的线程再次获得锁时，会判断是否偏向锁指向我，那么该线程将不用再次获得锁，直接进入同步块。

2.轻量级锁:偏向锁失败后，利用cas补救补救失败就会升级为重量级锁。

3.自旋锁：会做空操作，并且不停地尝试拿到这个锁。

### java中一亿个数找前10000个最大的

先利用Hash法去重复，去除大量的之后 然后等量的分成100份 用小顶堆 来获得10000个，再把所有的1万个都合在一起就OK

### java中线程的状态

java中的线程的状态有5种(新建、就绪、运行、阻塞、结束)

1.新建:创建后尚未启动的线程处于这种状态，新建出一个线程对象。

2.就绪状态：当针对该对象掉用了start()方法，该线程等待获取CPU的使用权

3.运行状态:在就绪状态下，获得了CPU处于运行状态。

4.阻塞:

等待阻塞：运行的线程执行wait方法，JVM会把该线程放入等待池

同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被其他的线程锁占用，则jvm会把该线程放入锁池中。

其他阻塞：运行的线程在执行sleep()方法或者join()方法时，或者发出IO请求，JVM会把线程置为阻塞状态。

5.结束:

也就是我们的死亡，表明线程结束。

### Maven的生命周期

maven有三套相互独立的生命周期

1.clean生命周期

pre-clean,clean,post-clean

2.default生命周期 构建项目

1.validate:验证工程是否正确，所有需要的资源是否可用

2.compile:编译项目源代码

3.test:使用合适的单元框架来测试已编译的源代码。

4.Package:把已编译的代码打包成可发布的格式,jar。

4）Package：把已编译的代码打包成可发布的格式，比如jar。

5）integration-test：如有需要，将包处理和发布到一个能够进行集成测试的环境。

6）verify：运行所有检查，验证包是否有效且达到质量标准。

7）install：把包安装到maven本地仓库，可以被其他工程作为依赖来使用。

8）Deploy：在集成或者发布环境下执行，将最终版本的包拷贝到远程的repository，使得其他的开发者或者工程可以共享。

3.

site生命周期：建立和发布项目站点，phase如下

1）pre-site：生成项目站点之前需要完成的工作

2）site：生成项目站点文档

3）post-site：生成项目站点之后需要完成的工作

4）site-deploy：将项目站点发布到服务器

### 数据库索引

##### 什么是索引？

（1）索引是对记录集多个字段进行排序的方法。

（2）也是一个数据结构，在一张表中为一个字段创建索引，将创建另外一个数据结构，包含字段的数值以及指向相关记录的指针，就可以对该数据结构进行二分法排序，当需要查询时就可以降低时间复杂度。

优势：快速存取数据；保证数据记录的唯一性；实现表和表之间的参照完整性；在使用order by group by子句进行数据的检索时，利用索引可以减少排序和分组的时间。

弊端：建立索引表也是会需要额外的空间。

##### 索引的工作原理：

    在对表中记录进行搜索时并不是对表中的数据进行全部的扫描遍历，而是查看在索引中定义的有序列，一旦在索引中找到了要查询的记录，就会得到一个指针，它会指向相应的表中数据所保存的位置。

##### 索引的类型：

（1）聚集索引：数据页在物理上的有序的存储，数据页的物理顺序是按照聚集索引的顺序进行排列。在聚集索引中数据页聚集索引的叶子节点，数据页之间通过双向的链表形式相连接，实际的数据存储在叶节点中。

（2）非聚集索引：叶子节点不存放具体的数据页信息，只存放索引的键值。非聚集索引的叶子节点包含着指向具体数据的指针，数据页之间没有连接，是相对独立的。

（3）唯一索引：在整个表中仅仅会出现一次（主键约束/UNIQUE）

（4）非唯一索引：在提取数据时允许出现重复的值。

（5）单一索引和组合索引

##### [哪些情况下索引会失效？](http://www.cnblogs.com/hongfei/archive/2012/10/22/2732589.html)

1.条件中有or但是前后没有同时使用索引

2.多列索引，不是使用前面部分

3.like查询是以%开头

4.字符类型应该加单引号 防止转换为int类型

##### 数据库查询优化(Sql)

1、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

2、对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

3、应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
     select id from t where num is null  
     可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：  
     select id from t where num=0

4、尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：  
     select id from t where num=10 or num=20  
     可以这样查询：  
     select id from t where num=10  
     union all  
     select id from t where num=20

 5、下面的查询也将导致全表扫描：(不能前置百分号)  
     select id from t where name like ‘�c%’  
    若要提高效率，可以考虑全文检索。

 6、in 和 not in 也要慎用，否则会导致全表扫描，如：  
     select id from t where num in(1,2,3)  
     对于连续的数值，能用 between 就不要用 in 了：  
     select id from t where num between 1 and 3

 7、如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：  
     select id from t where num=@num  
     可以改为强制查询使用索引：  
     select id from t with(index(索引名)) where num=@num

 8、应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：  
     select id from t where num/2=100  
     应改为:  
     select id from t where num=100\*2

 9、应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：  
     select id from t where substring(name,1,3)=’abc’–name以abc开头的id  
     select id from t where datediff(day,createdate,’2005-11-30′)=0–’2005-11-30′生成的id  
     应改为:  
     select id from t where name like ‘abc%’  
     select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′

 10、不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

 11、在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。

 12、不要写一些没有意义的查询，如需要生成一个空表结构：  
     select col1,col2 into #t from t where 1=0  
     这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：  
     create table #t(…)

 13、很多时候用 exists 代替 in 是一个好的选择：  
     select num from a where num in(select num from b)  
     用下面的语句替换：  
     select num from a where exists(select 1 from b where num=a.num)

 14、并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

 15、索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

 16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

 17、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

 18、尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

 19、任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

 20、尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

 21、避免频繁创建和删除临时表，以减少系统表资源的消耗。

 22、临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使 用导出表。

 23、在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

 24、如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

 25、尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

 26、使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

 27、与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

 28、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

 29、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

 30、尽量避免大事务操作，提高系统并发能力。查询sql语句中哪些比较慢

1.慢查询日志，一般设置查询超过两秒就记录

2.processlist:显示哪些线程正在运行.

3.explain关键字可以让我们更好的优化

##### 数据库设计优化:

1.反范式设计，尽量是单表，可以高效利用索引

2.可以使用查询缓存，Mysql也会自带查询缓存

3.尽量查询通过搜索引擎查不通过我们

4.key,value数据库

##### 锁的优化策略

1.读写分离，锁分离

2.减少锁持有时间，可以减少其他的锁的持有时间

3.以正确的顺序获得和释放锁

4.适当的锁的范围扩大或者缩小，控制锁的粒度

Spring Bean中作用域

 singleton：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例。

 prototype：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例。

 request：对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的Bean实例。只有在Web应用中使用Spring时，该作用域才有效。

 session：对于每次HTTP Session，使用session定义的Bean都将产生一个新实例。同样只有在Web应用中使用Spring时，该作用域才有效。

 Global session：每个全局的HTTP Session，使用session定义的Bean都将产生一个新实例。典型情况下，仅在使用portlet context的时候有效。同样只有在Web应用中使用Spring时，该作用域才有效。

### java中启定时任务

1.利用sleep特性//休眠

2.time和timerTask//定时器

3.ScheduledExecutorService service.scheduleAtFixedRate(runnable, 10, 1, TimeUnit.SECONDS);

 //任务调度服务

### 操作系统如何进行分页调度

用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

### linux内核的三种主要调度策略：

1，SCHED\_OTHER 分时调度策略，

2，SCHED\_FIFO实时调度策略，先到先服务

3，SCHED\_RR实时调度策略，时间片轮转

### TCP和UDP相关

TCP通过什么方式提供可靠性:

1.超时重发，发出报文段要是没有收到及时的确认，会重发。

2.数据包的校验，也就是校验首部数据和。

3.对失序的数据重新排序

4.进行流量控制，防止缓冲区溢出

5.快重传和快恢复

6.TCP会将数据截断为合理的长度

TCP和UDP的区别:

1.UDP是无连接的，TCP必须三次握手建立连接

2.UDP是面向报文，没有拥塞控制，所以速度快，适合多媒体通信要求，比如及时聊天，支持一对一，一队多。多对一，多对多。

3.TCP只能是一对一的可靠性传输

TCP的RPC，在协议栈的下层，能够灵活的对字段进行定制，减少网络传输字节数，降低网络开销，提高性能，实现更大的吞吐量和并发数。但是实现代价高，底层复杂，难以得到开源社区的支持，难以实现跨平台

### 集群调优

1.load

load是被定义为特定时间间隔内运行队列中的平均线程数，uptime查看，一般load不大于3，我们认为负载是正常的，如果每个CPU的线程数大于5，表示负载就非常高了。

2.CPU利用率

查看cpu的消耗的情况命令：top | grep Cpu

查看磁盘的剩余空间命令：df -h

查看系统的内存的使用情况：free -m

心跳检测方法

1.使用ping命令

对于full gc导致不响应，网络攻击这种 ping展示不明确

2.使用curl

访问我们的自测地址

3.对不同的功能使用curl检测，在response中加入状态头，表示正常

可以计算qps通过28原则

innodb存储引擎通过预写事务日志的方式保障事务的原子性，也就是在写入数据之前，先将数据操作写入日志，这种成为预写日志

轻量级锁认为在程序运行过程中，绝大部分的锁，在整个同步周期内都是不存在竞争的，利用cas操作避免互斥开销。

偏向锁是jdk1.6中引入的一项优化，甚至可以省掉CAS操作，偏向锁偏向第一个获得他锁的线程，如果在接下来执行过程中，这个锁没有被其他线程获取，则持有偏向锁的线程永远不需要同步。

### GC调优

查看GC日志，根据GC日志来优化

我们可以通过jps找到我们虚拟机正在运行的进程。参数 通过jps -help了解。

Jstat -gc 或者 -gcutil 查看堆使用情况-class 配合Jps得到进程

BTrace原理利用hotspot虚拟中的热替换，把代码动态的替换到java程序内，可在不需要重启的时候就可以排除问题

JConsole

我们也可以使用JConsole来分析这是一个图形化的，比较容易让我们操作

使用VisualVM 进行远程连接 使用JMX方式，也有修改tomcat的catalina就行了

### 内部类去访问外部变量，为什么需要加final?

题目有点问题，并不是所有的外部变量才加final,我们的内部类访问我们的成员变量就不需要加final,但是访问局部变量就必须要加final，因为方法（main方法）结束我们栈帧也就销毁了，但是我们内部类在堆中并没有被销毁，如果引用了成员变量，这时候被销毁了肯定是不行的，所以我们就需要成员变量设置为final，让其在方法（main方法）结束时不会被销毁。

### 泛型

泛型的作用:在我们没有泛型的时候，我们通过对Object的引用来对参数的任意化，任意化有个缺点就是要做显示的强制类型转换，强制转换有一个不好的地方是运行的时候才会报错，泛型的好处实在编译的时候检查类型安全，所以泛型的特点就是简单安全，泛型的原理是类型擦除，java的泛型是伪泛型，在编译期间，所有的泛型信息都会被擦除掉。在生成的java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器编译的时候去掉。比如List<Object>信息在编译后都是List。

### nginx和apache的对比

##### 1.nginx相对于apache来说

（1）轻量级占用的内存少；

（2）抗并发,nginx是异步非阻塞，apache是阻塞的，在高并发的情况下，nginx的性能优；

（3）高度模块化的设计，编写模块相对简单；

（4）社区活跃，各种高性能模块有。

适合场景：apache适合于动态的请求，而负载均衡和静态请求适合nginx，nginx采用的是epoll，并且有自己的一套sendfile系统，减少在内存中的赋值次数。

##### 2.apache 相对于nginx 的优点：

（1）rewrite ，比nginx 的rewrite 强大 ；

（2）模块超多，基本想到的都可以找到 ；

（3）少bug ，nginx 的bug 相对较多 ；

（4）超稳定 。

### 线程、进程的共享和独立

共享的部分：

1.进程代码段

2.进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)

3.进程打开的文件描述符、

4.信号的处理器、

5.进程的当前目录

6.进程用户ID与进程组ID

### 线程独有的内容包括：

1.线程ID

2.寄存器组的值

3.线程的堆栈

4.错误返回码

5.线程的信号屏蔽码

最大的优势就是线程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，线程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在线程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

### 简单的说一下nginx的优点

1.作为高性能的web服务器:相比Apache,Nginx使用更少的资源，支持更多的并发连接，体现更高的效率，这点让Nginx受到虚拟主机提供商的欢迎。一个Nginx实例能够轻松支持高达5万并发。

2.作为负载均衡服务器:Nginx即可以在内部直接支持PHP，也可以支持作为HTTP代理服务器对外进行服务，独有的send files系统，减少文件复制次数。

3.作为邮件代理服务器:也比Apache好很多。

4.Nginx安装简单，配置文件非常简洁。启动容易，7\*24小时几乎不间断，可以进行热更新。

### BIO

在BIO中读和写都是同步阻塞的，阻塞的时间取决于对方I/O线程的处理速度和网络的传输速度。本质上来讲，我们是无法保证生产环境的网络状况和对端的应用程序可以足够快，应用程序是不应该依赖对方的处理速度，它的可靠性就非常差。BIO就算用线程池实现，要是所有可用线程都被阻塞到故障点中，后续的所有I/O消息都将在队列中排队。

### NIO

（1）提供了高速，面向块的I/O。

（2）在NIO中所有数据都是用缓冲区来处理的，也就是使用我们的jvm中的direct memory（直接内存）。缓冲区是一个数组，但是缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置等信息。

（3）在NIO中channel（通道）也是特别重要的他是我们数据读写的通道，一般来说流比如inputStream和outputStream都是单向的，而通道是双向的，是全双工的。

（4）多路复用器Selector也是比较重要的，掌握它对于我们的NIO编程来说是比较重要的。多路复用器提供选择已经就绪的任务的能力。Selector会不断轮训注册在其上的Channel,如果某个Channel上面发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，通过Selection Key可以获取就绪Channel的集合，进行后续的I/o操作。我们只需要一个线程就可以管理我们多个客户端。

### 垃圾收集器

##### 1.Serial收集器

    Serial收集器是JAVA虚拟机中最基本、历史最悠久的收集器，在JDK 1.3.1之前是JAVA虚拟机新生代收集的唯一选择。Serial收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个 CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

     Serial收集器到JDK1.7为止，它依然是JAVA虚拟机运行在Client模式下的默认新生代收集器。它也有着优于其他收集器的地方：简单而高 效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集 效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会 再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，Serial收集器对于运行在 Client模式下的虚拟机来说是一个很好的选择。

##### 2. Parallel（并行）收集器

    这是 JVM 的缺省收集器。就像它的名字，其最大的优点是使用多个线程来通过扫描并压缩堆。串行收集器在GC时会停止其他所有工作线程（stop-the- world），CPU利用率是最高的，所以适用于要求高吞吐量（throughput）的应用，但停顿时间（pause time）会比较长，所以对web应用来说就不适合，因为这意味着用户等待时间会加长。而并行收集器可以理解是多线程串行收集，在串行收集基础上采用多线 程方式进行GC，很好的弥补了串行收集的不足，可以大幅缩短停顿时间（如下图表示的停顿时长高度，并发比并行要短），因此对于空间不大的区域（如 young generation），采用并行收集器停顿时间很短，回收效率高，适合高频率执行。

##### 3.CMS收集器

    CMS（Concurrent Mark Sweep）收集器是基于“标记-清除”算法实现的，它使用多线程的算法去扫描堆（标记）并对发现的未使用的对象进行回收（清除）。整个过程分为6个步骤，包括：

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

并发预清理（CMS-concurrent-preclean）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

并发重置（CMS-concurrent-reset）

    其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一 般会比初始标记阶段稍长一些，但远比并发标记的时间短。其他动作都是并发的。

    需要注意的是，CMS收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序的运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次 收集中处理掉它们，只好留待下一次GC时再将其清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，即还需要预留足够的 内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使 用。在默认设置下，CMS收集器在老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数 -XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数以获取更好的性能。要是CMS运行期 间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置 得太高将会很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

try catch有return 在字节码中 可以看到会出现Goto跳转行数，跳转到finally中的return

### Tomcat总结构

最外层的Server提供接口访问内部的Service服务。

Service服务的作用就是把Connector 在Connetor中监听端口和Container连接起来，方便我们的操纵控制。

在tomcat中生命周期是统一的由Lifecycle来管理的

在lifecycle中有两个方法比较重要start和stop,调用server的start会去调用自己下面所有service的start方法

Connector最重要的功能就是接受连接请求然后分配线程处理

在Container中 有4个级别Engine,Host,Context，warpper，这四个组件不是平行的，而是父子关系，Engine 包含 Host,Host 包含 Context，Context 包含 Wrapper。通常一个 Servlet class 对应一个 Wrapper，如果有多个 Servlet 就可以定义多个 Wrapper，如果有多个 Wrapper 就要定义一个更高的 Container 了，如 Context。在Host中有个value比较重要，类似于一个管道，和拦截器链差不多，我们在中间可以进行一些处理。

在Engine中只能添加子容器Host，不能添加父容器.Engine下可以配置多个虚拟主机Virtual Host，每个虚拟主机都有一个域名

当Engine获得一个请求时，它把该请求匹配到某个Host上，然后把该请求交给该Host来处理Engine有一个默认虚拟主机，当请求无法匹配到任何一个Host上的时候，将交给该默认Host来处理

一个Host就类似于一个虚拟主机，用来管理应用。代表一个Virtual Host，虚拟主机，每个虚拟主机和某个网络域名Domain Name相匹配

每个虚拟主机下都可以部署(deploy)一个或者多个Web App，每个Web App对应于一个Context，有一个Context path

当Host获得一个请求时，将把该请求匹配到某个Context上，然后把该请求交给该Context来处理

匹配的方法是“最长匹配”，所以一个path==""的Context将成为该Host的默认Context

所有无法和其它Context的路径名匹配的请求都将最终和该默认Context匹配。

Context 代表 Servlet 的 Context，它具备了 Servlet 运行的基本环境，理论上只要有 Context 就能运行 Servlet 了。简单的 Tomcat 可以没有 Engine 和 Host。

Context 最重要的功能就是管理它里面的 Servlet 实例，Servlet 实例在 Context 中是以 Wrapper 出现的，还有一点就是 Context 如何才能找到正确的 Servlet 来执行它呢？

一个Context对应于一个Web Application，一个Web Application由一个或者多个Servlet组成

Context在创建的时候将根据配置文件CATALINAHOME/conf/web.xml和CATALINAHOME/conf/web.xml和WEBAPP\_HOME/WEB-INF/web.xml载入Servlet类

当Context获得请求时，将在自己的映射表(mapping table)中寻找相匹配的Servlet类

如果找到，则执行该类，获得请求的回应，并返回.

### 线程池参数

JDK1.5中引入了强大的concurrent包，其中最常用的莫过了线程池的实现。ThreadPoolExecutor（线程池执行器），它给我们带来了极大的方便，但同时，对于该线程池不恰当的设置也可能使其效率并不能达到预期的效果，甚至仅相当于或低于单线程的效率。

ThreadPoolExecutor类可设置的参数主要有：

         （1）corePoolSize  基本大小

核心线程数，核心线程会一直存活，即使没有任务需要处理。当线程数小于核心线程数时，即使现有的线程空闲，线程池也会优先创建新线程来处理任务，而不是直接交给现有的线程处理。

核心线程在allowCoreThreadTimeout被设置为true时会超时退出，默认情况下不会退出。

 （2）maxPoolSize  最大大小

当线程数大于或等于核心线程corePoolSize，且任务队列已满时，线程池会创建新的线程，直到线程数量达到maxPoolSize。如果线程数已等于maxPoolSize，且任务队列已满，则已超出线程池的处理能力，线程池会拒绝处理任务而抛出异常。

 keepAliveTime  大于coolPoolSize的 会退出

当线程空闲时间达到keepAliveTime，该线程会退出，直到线程数量等于corePoolSize。如果allowCoreThreadTimeout设置为true，则所有线程均会退出直到线程数量为0。

 allowCoreThreadTimeout  是否退出核心线程

是否允许核心线程空闲退出，默认值为false。

 queueCapacity

任务队列容量。从maxPoolSize的描述上可以看出，任务队列的容量会影响到线程的变化，因此任务队列的长度也需要恰当的设置。

### 线程池按以下行为执行任务

 当线程数小于核心线程数时，创建线程。

 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。

 当线程数大于等于核心线程数，且任务队列已满

 若线程数小于最大线程数，创建线程

 若线程数等于最大线程数，抛出异常，拒绝任务

### 系统负载

参数的设置跟系统的负载有直接的关系，下面为系统负载的相关参数：

 tasks，每秒需要处理的最大任务数量

 tasktime，处理每个任务所需要的时间

 responsetime，系统允许任务最大的响应时间，比如每个任务的响应时间不得超过2秒。

##### 参数设置

###### corePoolSize:

每个任务需要tasktime秒处理，则每个线程每钞可处理 1/tasktime个任务。系统每秒有tasks个任务需要处理，则需要的线程数为：tasks/(1/tasktime)，即 tasks\*tasktime个线程数。假设系统每秒任务数为100~1000，每个任务耗时0.1秒，则需要100\*0.1至1000\*0.1，即 10~100个线程。那么corePoolSize应该设置为大于10，具体数字最好根据8020原则，即80%情况下系统每秒任务数，若系统80%的情 况下第秒任务数小于200，最多时为1000，则corePoolSize可设置为20。

###### queueCapacity:

任务队列的长度要根据核心线程数，以及系统对任务响应时间的要求有关。队列长度可以设置为(corePoolSize/tasktime)\*responsetime： (20/0.1)\*2=400，即队列长度可设置为400。

队列长度设置过大，会导致任务响应时间过长，切忌以下写法：

LinkedBlockingQueue queue = new LinkedBlockingQueue();

这实际上是将队列长度设置为Integer.MAX\_VALUE，将会导致线程数量永远为corePoolSize，再也不会增加，当任务数量陡增时，任务响应时间也将随之陡增。

###### maxPoolSize:

当系统负载达到最大值时，核心线程数已无法按时处理完所有任务，这时 就需要增加线程。每秒200个任务需要20个线程，那么当每秒达到1000个任务时，则需要(1000-queueCapacity)\* (20/200)，即60个线程，可将maxPoolSize设置为60。

###### keepAliveTime:

线程数量只增加不减少也不行。当负载降低时，可减少线程数量，如果一个线程空闲时间达到keepAliveTiime，该线程就退出。默认情况下线程池最少会保持corePoolSize个线程。

###### allowCoreThreadTimeout:

默认情况下核心线程不会退出，可通过将该参数设置为true，让核心线程也退出。

以上关于线程数量的计算并没有考虑CPU的情况。若结合CPU的情况，比如，当线程数量达到50时，CPU达到100%，则将 maxPoolSize设置为60也不合适，此时若系统负载长时间维持在每秒1000个任务，则超出线程池处理能力，应设法降低每个任务的处理时间 (tasktime)。

使用ThreadLocal解决SimpleDateFormat，

privatestatic ThreadLocal<DateFormat> threadLocal = new ThreadLocal<DateFormat>() {        @Override        protected DateFormat initialValue() {            returnnew SimpleDateFormat("yyyy-MM-dd HH:mm:ss");        }    };

或者推荐Joda-Time处理时间比较推荐

在java线程中有6个状态也就是Thread中的枚举类

NEW,RUNNABLE，WAITING,TIME\_WAITING,BLOCKED,TERMINATED

应该在try{} catch{}中重新设置中断状态，因为发出中断异常被退出了。

### java内存模型

jvm规范定了jvm内存模型来屏蔽掉各种操作系统，虚拟机实现厂商和硬件的内存访问差异，确保java程序在所有操作系统和平台上能够实现一次编写，到处运行的效果。

### 为什么IP协议也能够进行数据的不可靠传输，还需要Udp

1.我们需要端口号来实现应用程序间的区分。

2.UDP校验和可以实现传输层的校验，虽然UDP协议不具备纠错能力，但是可以对出错的数据包进行丢弃，而IP的校验只是在校验IP报头，而不是整个数据包，整个数据包的校验是在传输层完成的，如果出错了，就会把出错的数据包丢弃。这也就是为什么需要有传输层

### 进程通信中管道和共享内存谁的速度快？

1.管道通信方式的中间介质是文件，通常称这种文件为管道文件。两个进程利用管道进行通信，一个进程为写进程，另一个进程为读进程。写进程通过往管道文件中写入信息，读进程通过读端从管道文件中读取信息，这样的方式进行通信。

2.共享内存是最快的可用IPC形式，他通过把共享的内存空间映射到进程的地址空间，进程间的数据传递不在通过执行任何进入内核的系统调用，节约了时间。java内存模型就采用的是共享内存的方式。各个进程通过公共的内存区域进行通信。

### java线程池shutdown和shutdownNow的区别？

##### Shutdown()方法

当线程池调用该方法时,线程池的状态则立刻变成SHUTDOWN状态。此时，则不能再往线程池中添加任何任务，否则将会抛出RejectedExecutionException（拒绝执行异常）异常。但是，此时线程池不会立刻退出，直到添加到线程池中的任务都已经处理完成，才会退出。

##### shutdownNow()方法

 根据JDK文档描述，大致意思是：执行该方法，线程池的状态立刻变成STOP状态，并试图停止所有正在执行的线程，不再处理还在池队列中等待的任务，当然，它会返回那些未执行的任务。

     它试图终止线程的方法是通过调用Thread.interrupt()方法来实现的，但是大家知道，这种方法的作用有限，如果线程中没有sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线程的。所以，ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。

### Integer的装箱和拆箱的基本原理

Integer包装类是Java最基本的语法糖优化，如果我们写一段程序，通过反编译就会看到，通过的是Integer.valueOf()或者Interger.intValue()来进行转换的。

Integer和int比较会自动拆箱，

当Integer和integer比较的时候是不会自动拆箱的，除非遇到了算术符，才会调用intValue()方法去拆箱。并且在Integer中的equals是不会处理数据类型转换关系的，使用时是需要慎用，在equals方法中判断为Integer类的才会执行真正的判断流程也就是拆箱去判断，所以不会处理数据类型转换比如Long。如果是基本类型比较，编译器会隐形的把int转换为long，就可以得出正确的结论。

为什么不推荐使用resume和suspend？？

因为在使用suspend()去挂起线程的时候，suspend在导致线程暂停的同时，不会去释放任何锁的资源。必须要等待resume()操作，被挂起的线程才能继续。如果我们resume()操作意外地在suspend()前就执行了，那么被挂起的线程可能很难有机会被继续执行。并且，更严重的是：所占用的锁不会被释放，因此可能会导致整个系统工作不正常

yield是谦让，调用后会使当前线程让出CPU，但是注意的地方让出CPU并不表示当前线程不执行了。当前线程在让出CPU后，还会进行CPU资源的争夺，有可能刚刚一让马上又进。

### 数据库事务的隔离级别

1.未提交读 都不能解决

2.已提交读 能解决脏读

3.可重复读 能解决脏读，不可重复读

4.序列化读 能解决脏读，不可重复读，幻读

### Docker和虚拟机的比较

1.传统的虚拟机在宿主机操作系统上面会利用虚拟机管理程序去模拟完整的一个虚拟机操作系统，docker只是在操作系统上的虚拟化，直接复用本地主机的操作系统，非常轻量级。

docker启动速度一般在秒级，虚拟机启动速度一般在分钟级。

2.对于资源的使用一般是mb,一台机器可以有上千个docker容器，但是虚拟机占用资源为GB，只能支持几个。

3.性能:接近原生，由于又虚拟化了一层所以效率低。

4.docker采用类似git的命令学习升本低，指令简单。

5.虚拟机隔离性是完全隔离，容器是安全隔离

### lucence组件

每一个词都会有一个倒排表，多个可以合并

为了标识webSocket:会在请求头中写一个upgrade:webSocket

与HTTP比较

同样作为应用层的协议，WebSocket在现代的软件开发中被越来越多的实践，和HTTP有很多相似的地方，这里将它们简单的做一个纯个人、非权威的比较：

相同点

 都是基于TCP的应用层协议。

 都使用Request/Response模型进行连接的建立。

 在连接的建立过程中对错误的处理方式相同，在这个阶段WS可能返回和HTTP相同的返回码。

 都可以在网络中传输数据。

不同点

 WS使用HTTP来建立连接，但是定义了一系列新的header域，这些域在HTTP中并不会使用。

 WS的连接不能通过中间人来转发，它必须是一个直接连接。

 WS连接建立之后，通信双方都可以在任何时刻向另一方发送数据。f

 WS连接建立之后，数据的传输使用帧来传递，不再需要Request消息。

 WS的数据帧有序。

### 微服务架构

 首先看一下[微服务架构](http://mp.weixin.qq.com/s?__biz=MzA5Nzc4OTA1Mw==&mid=408409940&idx=1&sn=031cf076887a6e995eda05cf25249f05&scene=21" \l "wechat_redirect)的定义：微服务（MSA）是一种架构风格，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。它有如下几个特征：

 小，且只干一件事情。

 独立部署和生命周期管理。

 异构性

 轻量级通信，RPC或者Restful。

### BIO通信模型图:

BIO以是一客户端一线程，一个Acceptor线程来接受请求，之后为每一个客户端都创建一个新的线程进行链路处理，最大的问题缺乏弹性伸缩能力，客户端并发访问量增加后，线程数急剧膨胀。可以用线程池缓解但是还是不行。

NIO非阻塞IO解决了这个问题，一个线程就可以管理多个Socket.

在NIO中有三个需要我们了解Buffer,Channel,Selector。

Buffer:NIO是面向缓冲区，IO是面向流的。缓冲区实质上一个数组

，缓冲区不仅仅是一个数组，缓冲区提供了对数据结构化访问以及维护读写位置等信息。

### 通道Channel:

Channel是一个通道，网络数据通过Channel读取和写入。通道与流的不同之处在于通道是双向的，流是一个方向上移动，通道可以用于读写操作，特别是在UNIX网络编程模型底层操作系统的通道都是全双工的，同时支持读写操作。

Selector:多路复用器NIo编程的基础，多路复用器提供选择就绪任务的 能力。简单来说Selector会不断注册在其上的Channel，如果某个Channel上面发生读或者写时间，这个Channel就处于就绪状态，会被Selector轮询出来，通过SelectionKey,一个多路复用器可以同时轮询多个Channel，JDK使用了epoll代理传统select实现，所以没有最大连接句柄fd的限制，意味着一个线程负责Selector的轮询，就可以接入成千上万的客户端

select/poll 和 epoll： select poll顺序扫描fd，就绪就返回fd。epoll则是 采用事件驱动，用回调的方式返回fd。

### [面向对象设计七大原则](http://www.cnblogs.com/sunflower627/p/4718702.html)

1. 单一职责原则（Single Responsibility Principle）

每一个类应该专注于做一件事情。

2. 里氏替换原则（Liskov Substitution Principle）

超类存在的地方，子类是可以替换的。

3. 依赖倒置原则（Dependence Inversion Principle）

实现尽量依赖抽象，不依赖具体实现。

4. 接口隔离原则（Interface Segregation Principle）

应当为客户端提供尽可能小的单独的接口，而不是提供大的总的接口。

5. 迪米特法则（Law Of Demeter）

又叫最少知识原则，一个软件实体应当尽可能少的与其他实体发生相互作用。

6. 开闭原则（Open Close Principle）

面向扩展开放，面向修改关闭。

7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）

尽量使用合成/聚合达到复用，尽量少用继承。原则： 一个类中有另一个类的对象。

### 拦截器和过滤器的区别

强类型:不允许隐形转换

弱类型:允许隐形转换

静态类型:编译的时候就知道每一个变量的类型，因为类型错误而不能做的事情是语法错误.

动态类型:编译得时候不知道每一个变量的类型，因为类型错误而不能做的事情是运行时错误

编译语言和解释语言：解释性编程语言，每个语句都是执行的时候才翻译而且是一句一句的翻译就很低。

编译的语言就只需要一次 就可以了

### 继承Thread 和 接口Runnabel的区别

主要是继承和实现接口的两个区别,如果只想重写run方法就可以使用，如果不重写其他方法 就使用runnable，如果使用实现接口的实现，让自己方便管理线程以及让线程复用，可以使用线程池去创建。

### hash算法（特点、哈希函数构造、解决冲突的策略）

##### 哈希表的概念：

哈希表就是一种以 键-值(key-indexed) 存储数据的结构，我们只要输入待查找的值即key，即可查找到其对应的值。

##### 哈希表的实现思路：

如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。对于冲突的情况，则需要处理地址的冲突问题。所以，一方面要构造出良好的哈希函数，对键值集合进行哈希，另外一方面需要设计出良好的解决冲突的算法，处理哈希碰撞的冲突。

##### 哈希表的查找步骤：

（1）使用哈希函数将被查找的键转换为数组的索引。在理想的情况下，不同的键会被转换为不同的索引值，但是在有些情况下我们需要处理多个键被哈希到同一个索引值的情况。所以哈希查找的第二个步骤就是处理冲突

（2）处理哈希碰撞冲突。有很多处理哈希碰撞冲突的方法，本文后面会介绍拉链法和线性探测法。

##### 哈希表的思想：

是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库)即可在时间和空间上做出取舍。

##### 哈希表的工作步骤：

 1)   哈希(Hash)函数是一个映象，即将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可；

 2)  由于哈希函数是一个压缩映象，因此，在一般情况下，很容易产生“冲突”现象，即： key1！=key2，而  f  (key1) = f(key2)。键不同，但是对应的取值相同。

3).  只能尽量减少冲突而不能完全避免冲突，这是因为通常关键字集合比较大，其元素包括所有可能的关键字，而地址集合的元素仅为哈希表中的地址值。

##### 哈希函数的构造方法：

1、直接地址法

以数据元素的关键字k本身或者他的线性的函数作为它 的哈希地址，也就是H(k)=k,或者H(k)=a\*k+b;

适用的场景：地址集合的大小==关键字的集合。

2、数字分析法

取数据元素关键字中某些取值较均匀的数字位作为哈希地址的方法

适用的场景：能预先估计出全体关键字的每一位上各种数字出现的频度。

3、折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位），这方法称为折叠法

适用场景：关键字的数字位数特别多。

4、平方取中法

先取关键字的平方，然后根据可使用空间的大小，选取平方数是中间几位为哈希地址。

适用场景：通过取平方扩大差别，平方值的中间几位和这个数的每一位都相关，则对不同的关键字得到的哈希函数值不易产生冲突，由此产生的哈希地址也较为均匀。

5、减去法

6、基数转换法

7、除留余数法

8、随机数法

9、随机乘数法

10、旋转法

##### 构造哈希哈希函数的原则：

1、计算哈希函数的时间

2、关键子的长度

3、哈希表的长度

4、关键字的分布的情况

5、记录查找频率

##### 哈希函数的冲突解决的方法：

1、开放地址法

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

          Hi=（H（key）+di）% m   i=1，2，…，n

    其中H（key）为哈希函数，m 为表长，di称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：

l         线性探测再散列

    dii=1，2，3，…，m-1

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

l         二次探测再散列

    di=12，-12，22，-22，…，k2，-k2    ( k<=m/2 )

    这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

l         伪随机探测再散列

    di=伪随机数序列。

2、再哈希法

这种方法是同时构造多个不同的哈希函数：

    Hi=RH1（key）  i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

3、链地址法

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

4、建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

### 类加载的方式

1.通过new

2.通过反射利用当前线程的classloader

3.自己继承实现一个classloader实现自己的类装载器

class.forName 和 Classloader.loadClass区别

Classloader

HashMap内部是怎么实现的？（拉链式结构）

核心：hashMap采用拉链法，构成“链表的数组”

存储示意图：

### 索引index建立规则:

（1）一般情况下通过hash(key)%length实现，元素存储在数组中的索引是由key的哈希值对数组的长度取模得到。

（2）hashmap也是一个线性的数组实现的，里面定义一个内部类Entry,属性包括key、value、next.Hashmap的基础就是一个线性数组，该数组为Entry[] ,map里面的内容都保存在entry[]数组中。

（3）确定数组的index:hashcode%table.length

     数组的下标index相同，但是不表示hashcode相同。

实现随机存储的方法：

// 存储时:  
int hash = key.hashCode(); // 这个hashCode方法这里不详述,只要理解每个key的hash是一个固定的int值  
int index = hash % Entry[].length;  
Entry[index] = value;  
  
// 取值时:  
int hash = key.hashCode();  
int index = hash % Entry[].length;  
return Entry[index];

put方法的实现：

如果两个key通过hash%Entry[].length得到的index相同，会不会有覆盖的危险？

这里HashMap里面用到链式数据结构的一个概念。上面我们提到过Entry类里面有一个next属性，作用是指向下一个Entry。打个比方， 第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起。所以疑问不用担心。也就是说数组中存储的是最后插入的元素。

get方法的实现：

先定位到数组元素，再遍历该元素处的链表

table的大小：

table的初始的大小并不是initialCapacity，是initialCapacity的2的n次幂

目的在于：当哈希表的容量超过默认的容量时，必须重新调整table的大小，当容量已经达到最大的可能的值时，这时需要创建一张新的表，将原来的表映射到该新表。

req.getSession().invalidate(); 销毁session

req.getSession().setMaxInactiveInterval(30); 设置默认session的过期时间,tomcat的默认过期时间是 30分钟

### 利用线程池的优势：

1、降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

2、提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

3、提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

boolean 单独使用的时候还会被变成int,4个字节

boolean[] 数组使用的时候变成byte,1个字节

### Cache和Buffer的区别

Cache:缓存区，位于cpu和主内存之间容量很小但速度很快的存储器，因为CPU的速度远远高于主内存的速度，CPU从内存中读取数据许要等待很长的时间，而Cache保存着CPU刚用过的数据或循环使用的部分数据，Cache读取数据更快，减少cpu等待时间。

Buffer:缓冲区，用于存储速度不同步的情况，处理系统两端速度平衡，为了减小短期内突发I/O的影响，起到流量整形的作用。速度慢的可以先把数据放到buffer，然后达到一定程度在读取数据

### 内连接：

必须两个表互相匹配才会出现

外链接

左外链接:左边不加限制

右外连接：右边不加限制

全连接：左右都不加限制

### 三、如何创建索引

全文检索的索引创建过程一般有以下几步：

##### 第一步：一些要索引的原文档(Document)。

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to go out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

##### 第二步：将原文档传给分词器(Tokenizer)。

分词器(Tokenizer)会做以下几件事情(此过程称为Tokenize)：

1. 将文档分成一个一个单独的单词。

2. 去除标点符号。

3. 去除停词(Stop word)。

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中停词(Stop word)如：“the”,“a”，“this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

经过分词(Tokenizer)后得到的结果称为词元(Token)。

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

##### 第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

对于英语，语言处理组件(Linguistic Processor)一般做以下几点：

1. 变为小写(Lowercase)。

2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。

3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引 中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询 语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。

ZK实现分布式锁- 是否存在，并且最小的

根据ZK中节点是否存在，可以作为分布式锁的锁状态，以此来实现一个分布式锁，下面是分布式锁的基本逻辑：

 客户端调用create()方法创建名为“/dlm-locks/lockname/lock-”的临时顺序节点。

 客户端调用getChildren(“lockname”)方法来获取所有已经创建的子节点。

 客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点是所有节点中序号最小的，那么就认为这个客户端获得了锁。

 如果创建的节点不是所有节点中需要最小的，那么则监视比自己创建节点的序列号小的最大的节点，进入等待。直到下次监视的子节点变更的时候，再进行子节点的获取，判断是否获取锁。

而且zk的临时节点可以直接避免网络断开或主机宕机，锁状态无法清除的问题，顺序节点可以避免惊群效应。这些特性都使得利用ZK实现分布式锁成为了最普遍的方案之一。

Redus实现分布式锁，使用setNX （set if not exists）

getset(先写新值返回旧值，用于分辨是不是首次操作) 防止网络断开后 会设置超时

<http://blog.csdn.net/ugg/article/details/41894947>

SETNX 可以直接加锁操作，比如说对某个关键词foo加锁，客户端可以尝试

SETNX foo.lock <current unix time>

如果返回1，表示客户端已经获取锁，可以往下操作，操作完成后，通过

DEL foo.lock

命令来释放锁。

### 处理死锁

在 上面的处理方式中，如果获取锁的客户端端执行时间过长，进程被kill掉，或者因为其他异常崩溃，导致无法释放锁，就会造成死锁。所以，需要对加锁要做时 效性检测。因此，我们在加锁时，把当前时间戳作为value存入此锁中，通过当前时间戳和Redis中的时间戳进行对比，如果超过一定差值，认为锁已经时 效，防止锁无限期的锁下去，但是，在大并发情况，如果同时检测锁失效，并简单粗暴的删除死锁，再通过SETNX上锁，可能会导致竞争条件的产生，即多个客 户端同时获取锁。

C1获取锁，并崩溃。C2和C3调用SETNX上锁返回0后，获得foo.lock的时间戳，通过比对时间戳，发现锁超时。

C2 向foo.lock发送DEL命令。

C2 向foo.lock发送SETNX获取锁。

C3 向foo.lock发送DEL命令，此时C3发送DEL时，其实DEL掉的是C2的锁。

C3 向foo.lock发送SETNX获取锁。

此时C2和C3都获取了锁，产生竞争条件，如果在更高并发的情况，可能会有更多客户端获取锁。所以，DEL锁的操作，不能直接使用在锁超时的情况下，幸好我们有GETSET方法，假设我们现在有另外一个客户端C4，看看如何使用GETSET方式，避免这种情况产生。

C1获取锁，并崩溃。C2和C3调用SETNX上锁返回0后，调用GET命令获得foo.lock的时间戳T1，通过比对时间戳，发现锁超时。

C4 向foo.lock发送GESET命令，

GETSET foo.lock <current unix time>

并得到foo.lock中老的时间戳T2

如果T1=T2，说明C4获得时间戳。

如果T1!=T2，说明C4之前有另外一个客户端C5通过调用GETSET方式获取了时间戳，C4未获得锁。只能sleep下，进入下次循环中。

现在唯一的问题是，C4设置foo.lock的新时间戳，是否会对锁产生影响。其实我们可以看到C4和C5执行的时间差值极小，并且写入foo.lock中的都是有效时间错，所以对锁并没有影响。

为 了让这个锁更加强壮，获取锁的客户端，应该在调用关键业务时，再次调用GET方法获取T1，和写入的T0时间戳进行对比，以免锁因其他情况被执行DEL意 外解开而不知。以上步骤和情况，很容易从其他参考资料中看到。客户端处理和失败的情况非常复杂，不仅仅是崩溃这么简单，还可能是客户端因为某些操作被阻塞 了相当长时间，紧接着 DEL 命令被尝试执行(但这时锁却在另外的客户端手上)。也可能因为处理不当，导致死锁。还有可能因为sleep设置不合理，导致Redis在大并发下被压垮。 最为常见的问题还有

 AOF重写带有子进程副本保证安全

### Java虚拟机

Java内存结构，分区，每个区放置什么

程序计数器：（线程私有）当前线程所执行的字节码的行号指示器，通过改变这个计数器的值来选取下一条需要执行的字节码的指令，以程序中分支、循环和跳转等流程的控制都离不开这个计数器的指示。

虚拟机栈：（线程私有），每个方法在执行时都会创建一个栈桢，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。一个方法从调用到执行完成的过程，对应的栈桢在虚拟机栈的进出过程。当线程结束时，虚拟机栈中的数据会被自动的释放。

局部变量表：基本数据类型、对象的引用、返回地址，局部变量表锁需要的内存空间是在程序编译时就已经会被确定好的。

本地方法栈：（线程私有）虚拟机栈是为执行java方法所服务的，而本地方法栈是为了虚拟机使用到的本地方法锁服务的。

堆区：（线程共享）java堆是被所有的线程所共享的一片区域，所有的对象的实例和数组都会在堆区尽心分配。java堆细分：新生代和老年代；也可能会划分出多个线程锁共享额分配缓冲区TLAB;

Java堆可以在物理上不连续的内存空间中，只要逻辑上连续就可以。

方法区：（线程共享）存储已经被虚拟机加载过的类的信息、常量、静态变量和及时编译器编译后的代码。在方法区中一个区域叫做：运行时常量池，用于存放编译后生成的字面量和符号的引用。

### 堆的分代

(1)年轻代：

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个Eden区，两个Survivor区(一般而言)。  
大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当另一个Survivor区也满了的时候，从前一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。

(2)年老代：

在年轻代中经历了N（可配置）次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

(3)持久代：

用于存放静态数据，如 Java Class, Method 等。持久代对垃圾回收没有显著影响。

### OOM异常的处理思路

对象的创建方法，对象的内存的分配，对象的访问定位

对象的创建：

(1)第一步，当遇到一个new的指令，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，需要先执行相应的类加载过程；

(2)第二步，根据类加载完成后确定的内存大小，为对象分配内存；

(3)第三步，需要对分配到的内存空间都初始化为零值；

(4)第四步，虚拟机要对对象设置一些基本信息，如对象是那个类的实例、对象的哈希码、对象的GC分代年龄信息、如何才能找到类的元数据信息等，到这里虚拟机创建对象的工作已经完成；

(5)第五步，从程序的角度，我们还需要对对象进行初始化操作。

### 对象的内存分配：

(1)对象头:

存储hashcode 、gc分代年龄以及一些必要的自身的运行时数据

(2)实例数据：

存储真实的数据信息

(3)对齐填充：

仅仅起到占位符的作用

对象的访问定位：

通过句柄池的访问，在句柄池中保存着到对象实例数据的指针以及到对象类型的数据的指针

通过直接的指针服访问，通过引用直接指向java堆的对象的实例数据

### GC的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？

标记清除法：

就是先标记哪些对象实例不用，然后直接清除。缺点就是产生大量的内存碎片，下次若要存储一个大的对象，无法找到连续内存而又必须提前GC

标记整理：

也就是先标记，然后对存活的对象进行移动，全部移动到一端，然后再对其它的内存进行清理。

复制算法：

把内存分成相等的AB两块，每次只使用其中的一块。比如当A内存使用完后，就把A中还存活着的对象复制到另外一块内存中去（B），然后再把已经使用过的内存清理掉。优点：这样就不用考虑内存碎片的问题了。缺点：内存减半，代价略高。

### GC收集器有哪些？CMS收集器与G1收集器的特点。

对于新生代的收集器：

Serial单线程收集器 parnew多线程收集器  parallelSccavenge收集器

对于老年代的收集器：

CMS并发收集低停顿收集器  serial Old单线程收集器  parallel Old多线程收集器

CMS收集器：

优点：并发收集、低停顿

缺点：

(1)对cpu资源非常的敏感，在并发的阶段虽然不会导致用户的线程停顿，但是会由于占用一部分的线程导致应用程序变慢，总的吞吐量会降低；

(2)无法去处理浮动垃圾；

(3)基于“标记-清除”算法的收集器，所以会出现碎片。

G1收集器：

优点：

(1)能充分利用cpu、多核的优势，使用多个cpu缩短停顿的时间；

(2)分代收集，不要其他收集器的配合便可以独立管理整个的GC堆；

(3)空间整合：整体基于“标记-清理”算法的实现，局部是基于“复制”算法的实现；

(4)可以预测的停顿

Minor GC、Full GC分别在什么时候发生？

Minor GC:新生代GC，当jvm无法为一个新的对象分配空间时会触发

Full GC:整个堆空间的GC

### 类加载的五个过程：加载、连接、初始化。

类的加载：将类的class文件读入内存，并创建一个叫做java.lang.Class对象，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

这些类的class文件的来源：

(1)从本地文件系统中加载class文件

(2)从jar包中加载class文件，比如jdbc编程时

(3)通过网络加载class文件

(4)把一个java源文件动态编译，并执行加载

##### 连接：

(1)验证：验证阶段用于检验被加载的类是否具有正确的内部结构，并和其他的类协调一致

(2)准备：为类的类变量分配内存，并去设置默认的值

(3)解析：将类的二进制数据中的符号引用替换成直接引用。

##### 初始化：

主要是对类变量进行初始化。

(1)如果该类还没有被加载和连接，则先进行加载连接

(2)如果该类的直接父类还没有被初始化，则先初始化其直接父类

(3)类中如果有初始化的语句则先去执行这些初始化语句。

### 反射

概念：在运行的状态中，对于任何一个类或者对象，可以知道其任意的方法和属性，这种动态地调用其属性和方法的手段叫做反射。利用的反编译的手段

##### 一、通过三种方式来获取Employee类型,获取类：

(1)Class c1 = Class.forName(“Employee”);

(2)Class c2 =Employee.class;

(3)Employee e = new Employee(); Class c3 = e.getClass();

##### 二、得到class的实例：

Object o = c1.newInstance();

##### 三、获取所有的属性

Field[] fs = c.getDeclaredFields();

##### 四、获取所有的方法

GetDeclareMethods();

### 多线程（线程锁）

##### 线程的状态：

新建状态、就绪状态、运行状态、阻塞状态、死亡状态（线程状态转换图）

##### 多线程的创建和启动：

(1)继承Thread类，重写类的run方法，调用对象的start方法启动

(2)实现Runnable接口，并重写该接口的run方法，该方法同样是线程的执行体，创建runnable实现类的实例，并以此实例作为Thread类的target来创建thread对象，该thread对象才是真的线程对象。

(3)使用Callable和Future接口创建线程。具体是创建Callable接口的实现类，并实现clall()方法。并使用FutureTask类来包装Callable实现类的对象，且以此FutureTask对象作为Thread对象的target来创建线程。

方法3：

// 创建MyCallable对象

Callable<Integer> myCallable = new MyCallable();

//使用FutureTask来包装MyCallable对象

FutureTask<Integer> ft = new FutureTask<Integer>(myCallable);

//FutureTask对象作为Thread对象的target

Thread thread = new Thread(ft);

//线程进入到就绪状态

thread.start();

##### 线程同步的方法：sychronized、lock、reentrantLock等

synchronized修饰同步监视器:修饰可能被并发访问的共享资源充当同步监视器；

synchronized修饰方法，同步方法的同步监视器是this,也就是调用该方法的对象；

synchronizedd可以用来修饰方法，可以修饰代码块，但是不能修饰构造器和成员变量；

使用lock锁对象，每次只能有一个线程对lock对象进行加锁和释放锁，线程开始访问该锁对象时必须先获得锁lock

基本用法：

Private final ReentrantLock lock = new ReentrantLock();

Lock.lock();

Try(){

}catch(Exception e){

}finally{}

Lock.unlock();

##### 锁的等级：内置锁、对象锁、类锁、方法锁。

内置锁：每一个java对象都可以用做一个实现同步的锁，这个锁成为内置锁。当一个线程进入同步代码块或者方法的时候会自动获得该锁，在退出同步代码块或者方法时会释放该锁。

获得内置锁的方法：进入这个锁的保护的同步代码块或者方法

注意：java内置锁是一个互斥锁，最多只有一个线程能够获得该锁。

对象锁：对象锁是用于对象实例方法，或者一个对象实例上的。

类锁：类锁用于类的静态方法或者一个类的class对象上，一个类的对象实例有多个，但是每个类只有一个class对象，即不同对象实例的对象锁是互不干扰的，每一个类都有一个类锁。类锁只是概念上的，并不是真实存在的。

方法锁：synchronized修饰方法，同步方法的同步监视器是this,也就是调用该方法的对象；

### ThreadLocal的设计理念与作用。

作用：

ThreadLocal类只能去创建一个被线程访问的变量，如果一段代码含有一个ThreadLocal变量的引用，即使两个线程同时执行这段代码，它们也无法访问到对方的ThreadLocal变量。

创建ThreadLocal的方式：

private ThreadLocal myThreadLocal = new ThreadLocal();

我们可以看到，通过这段代码实例化了一个ThreadLocal对象。

我们只需要实例化对象一次，并且也不需要知道它是被哪个线程实例化。

虽然所有的线程都能访问到这个ThreadLocal实例，但是每个线程却只能访问到自己通过调用ThreadLocal的set()方法设置的值。即使是两个不同的线程在同一个ThreadLocal对象上设置了不同的值，他们仍然无法访问到对方的值。

如何为ThreadLocal对象赋值和取值：

一旦创建了一个ThreadLocal变量，你可以通过如下代码设置某个需要保存的值：

myThreadLocal.set("A thread local value”);

可以通过下面方法读取保存在ThreadLocal变量中的值：

String threadLocalValue = (String) myThreadLocal.get();

get()方法返回一个Object对象，set()对象需要传入一个Object类型的参数。

初始化该ThreadLocal变量：

通过创建一个ThreadLocal的子类重写initialValue()方法，来为一个ThreadLocal对象指定一个初始值。

### ThreadPool用法与优势。

优势：

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

用法：

线程池的创建：

new  ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds,runnableTaskQueue, handler);

参数：corePoolSize:线程池的基本大小

maximumPoolSize:线程池的最大大小

runnableTaskQueue:任务队列

keepAliveTime:线程活动保持时间

执行方式：

threadsPool.execute(handler);

threadsPool.submit(handler);

线程池的关闭：

Shutdown和shutdownNow方法实现

线程池的工作流程分析：

先将任务提交的顺序为核心线程池、队列、线程池、当这三个关节都不能执行用户所提交的线程时，则抛出“无法执行的任务”。

### 字节流和字符流

(1)java中字节流处理的最基本的单位是单个字节。通常用来处理二进制数据，最基本的两个字节流类是InputStream和OutputStream,这两个类都为抽象类。

字节流在默认的情况下是不支持缓存的。每次调用一次read方法都会请求操作系统来读取一个字节，往往会伴随一次磁盘的IO,如果要使用内存提高读取的效率，应该使用BufferedInputStream。

(2)字符流处理的最基本的单元是unicode（码元），通常用来来处理文本数据。

输入字符流（文件到内存）：把要读取的字节序列按照指定的编码方式解码为相应的字符序列，从而可以存在内存中。

输出字符流（内存到文件）：把要写入文件的字符序列转为指定的编码方式下的字节序列，然后写入文件中。

区别如下：

1、字节流操作的基本单元为字节；字符流操作的基本单元为Unicode码元。

unicode的编码范围：0x0000~0XFFFF,在这个范围的每个数字都有一个字符与之对应

2、字节流默认不使用缓冲区；字符流使用缓冲区。

3、字节流通常用于处理二进制数据，实际上它可以处理任意类型的数据，但它不支持直接写入或读取Unicode码元；字符流通常处理文本数据，它支持写入及读取Unicode码元。

### 序列化（常见的序列化操作）

含义：

java序列化：将java对象转换为字节序列的过程；

java反序列化：将字节序列恢复为java对象的过程

序列化的目的：

实现数据的持久化，将数据永久地保存在磁盘上，通常放在文件中；

利用序列化实现远程的通讯，在网络上传送对象的字节序列.

##### 实现序列化的三种方法：

（1）某实体类仅仅实现了serializable接口（常用）

序列化步骤：

步骤一：创建一个对象输出流，它可以包装一个其它类型的目标输出流，如文件输出流：

ObjectOutputStream out = new ObjectOutputStream(new fileOutputStream(“D:\\objectfile.obj”));

步骤二：通过对象输出流的writeObject()方法写对象：

//Hello对象的字节流将输入到文件

out.writeObject(“Hello”);

反序列化步骤：

步骤一：创建一个对象输入流，它可以包装一个其它类型输入流，如文件输入流：

ObjectInputStream in = new ObjectInputStream(new fileInputStream(“D:\\objectfile.obj”));

步骤二：通过对象输出流的readObject()方法读取对象：

//将从文件中读取到字节序列转化为对象

String obj1 = (String)in.readObject();

（2）若实体类仅仅实现了Serializable接口，并且还定义了readObject(ObjectInputStream in)和writeObject(ObjectOutputSteam out)，则采用以下方式进行序列化与反序列化。

ObjectOutputStream调用该对象的writeObject(ObjectOutputStream out)的方法进行序列化。

ObjectInputStream会调用该对象的readObject(ObjectInputStream in)的方法进行反序列化。

（3）若Student类实现了Externalnalizable接口，且Student类必须实现readExternal(ObjectInput in)和writeExternal(ObjectOutput out)方法，则按照以下方式进行序列化与反序列化。

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

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

String,StringBuffer,StringBuilder的区别，应用场景

1)在执行的速度上：StringBuilder>StringBuffer>String

2)String是字符串常量 StringBuffer和StringBuilder是字符串变量

例子1：

String s = “abcd”;

s=s+1;

Syos(s);

底层执行：首先创建一个对象s,赋予abcd.然后又创建新的对象s,之前的对象并没有发生变化，利用string操作字符串时，是在不断创建新的对象，而原来的对象由于没有了引用，会被GC,这样执行的效率会很低。

例子2：

String str2 = “This is only a”;

String str3 = “ simple”;

String str4 = “ test”;

String str1 = str2 +str3 + str4;

同理：str2 str3 str3没有被引用，但是创建了新的对象str1,执行速度上会很慢。

StringBuilder:线程非安全的

StringBuffer:线程安全的

例子3：

StringBuffer builder = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

应用场景：

A:使用要操作少量的数据时，使用String

B:单线程操作字符串缓冲区下操作大量数据使用StringBulider

C:多线程操作字符串缓冲区下操作大量的数据使用StringBuffer

### HashMap和HashTable的区别

HashMap是线程不安全的；允许有null的键和值；执行的效率高一点；方法不是synchronize的要提供外同步；包含有containsvalue和containskey的方法

HashTable是线程安全的；不允许有null的键和值；效率稍微低些；方法是synchronize的；包含contains方法

### ＝＝ 与 equals区别

==：对于基本数据类型的变量，直接比较存储的值是否相等；作用于引用类型的变量，则比较的是该变量所指向的地址是否相同。

equals:不同作用于基本数据类型的变量，如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址（相当于直接使用父类的equals方法，而该方法则是用==进行的比较，所以结果和用==比较的效果是一样的）；但是比如String Date类对equals进行了重写，比较的是字面量。

### final关键字

对于基本的数据类型，使用final关键字将使得数值恒定不变；

对于对象引用，final则是引用恒定不变，一但被初始化指向一个对象，它就不会再指向另外一个对象，但是该对象本身是可以被修改的；

对于类，如果不想继承某个类，可以将该类设置为fianl形式，该类不会有子类；

对于方法，final修饰的方法不会被重写

对于空白的final,对于没有给定初始值的fianl,编译器会在使用前初始化该final修饰的变量

对于宏变量：被final修饰的变量为宏常量 在编译的阶段被其本身的值直接替换

### short s1=1；s1 ＝ s1+1；

表达式类型的自动提升，一个short类型的变量和一个int型的数在一起进行运算，会将short类型的数隐式转换为int参与运算，但是该运算的结果为int类型是不同直接赋值给一个short类型的，必须进行强制的类型转换，否则编译是通不过的。

八种基本数据类型的大小，以及他们的封装类。

类型转换：byte (1字节)--->short(1)/char(2)--->int(4)--->long(8)--->float(4)--->double(8)

分装类：Byte Short Character Integer Long Float Double

### Switch能否用string做参数？（分版本讨论）

(1)在jdk1.7版本前不支持string作为参数，仅仅支持byte、short、char，因为可以转换为int,但是long和string不能转换为int，所以不能使用。

(2)在jdk1.7之后，支持使用string作为case的参数，实际匹配的是该字符串的hash值，然后用equals进行安全性检查。Switch支持String其实是一个语法糖，在编译后的字节码文件中都会被还原成原生的类型，并在相应的位置插入强制转换的代码，底层的JVM在switch上并没有修改;当传入switch是null时，在运行时对一个null调用hashcode()方法，会抛出空指针异常。

### Object有哪些公用方法？

object是所有类的父类，任何类都默认继承Object类

9、Clone

private保护方法，实现对象的浅复制，只有类实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportException

10、Equals

在object中与==是一样的，子类一般需要重写该方法

11、hashCode

该方法用于哈希查找，重写了equals方法一般都要重写hashcode方法，这个方法在一些具有哈希功能的collection中使用

12、getClass

final方法，获得运行时的类型

13、wait方法

使得当前的线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。Wait方法会一直等待，直到获得锁（到了睡眠的时间间隔也会唤醒自己）或者被中断掉。

调用该方法，当前的线程会进入到睡眠的状态，直到调用该对象的notify方法、notifyAll方法、调用interrupt中断该线程，时间间隔到了。

14、Notify

唤醒在该对象上的等待的某个线程

15、notifyAll

唤醒在该对象上的等待到所有的线程

16、toString

把对象转换成string类型进行输出

### Java的四种引用，强弱软虚，用到的场景。

引用的级别：

强引用>软引用>弱引用>虚引用

强引用：如果一个对象具有强引用，垃圾回收器绝对不会回收它。当内存空间不足时，jvm宁愿抛出outofmemoryError,使得程序的异常终止。

软引用：如果一个对象具有软引用，则内存空间足够，垃圾回收机制就不会去回收它，当内存不足时，就会进行回收。如果软引用所引用的对象被垃圾回收器回收，java虚拟机就会把这个软引用加入到与之关联的引用队列。

应用场景：实现内存敏感的高速缓存

弱引用：在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

应用场景：gc运行后终止

虚引用：就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

### Hashcode的作用。

(1)Hashcode 的存在主要用于解决查找的快捷性，比如在hashmap、hashtable中，hashcode是用来在散列的存储结构中确定对象的存储的位置的。

(2)如果两个对象相同，就是通过equals方法比较返回true,两个对象装在一个桶里，也就是hashcode也要一定相同。

(3)两个对象的hashcode相同，并不一定代表两个对象就是相同的，只能说明他们存储在一个桶里。

(4)一般重写了equals方法，也尽量去重写hashcode方法，保证先找到该桶，再去找到对应的类，通过equals方法进行比较。

### ArrayList、LinkedList、Vector的区别。

1、ArrayList:是基于动态数组的数据结构，LinkedList的基于链表的数据结构

2、对于随机访问get和set,ArrayList性能较好，因为LinkedList会去移动指针

3、对于新增和删除的操作，linkedList只需要修改指针的指向，性能较好，但是arrayList会移动数据。

vector的特点：

1、vector的方法都是线程同步的，是线程安全的，但是arraylist和linkedlist不是，由于线程的同步必然会影响性能，所以vector的性能不太高。

2、当vector或者arraylist的元素超过它的初始的大小时，vector会将容量翻倍，但是arraylist只会增加50%，这样有利于节约内存的空间。

Map、Set、List、Queue、Stack的特点与用法。

### HashMap和ConcurrentHashMap的区别

hashMap不是线程安全的；

concurrentHashMap是线程安全的；在其中引入了“分段锁”，而不是将所有的方法加上synchronized，因为那样就变成了hashtable.

所谓“分段锁”，就是把一个大的Map拆分成N个小的hashtable,根据key.hashcode()决定把key放在哪一个hashtable中。

通过把整个Map分为N个Segment（类似HashTable），可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。

### TreeMap、HashMap、LindedHashMap的区别。

HashMap:根据键的hashcode值进行存储数据，根据键可以直接获取它的值，具有快速访问的特点，遍历时取得数据是随机的，hashmap最多只允许一条记录的键为null（set无序不重复），允许多条记录的值为Null;如果要保证线程的同步，应该使用Collections.synchronizedMap()方法进行包装，或者使用ConcurrentHashMap

LinkedHashMap：保存了记录的插入的顺序，在迭代遍历Linkedhashmap时，先得到的记录肯定是先插入的，它遍历的速度只和实际的数据有关和容量没关。

TreeMap:实现的是SortMap，能够把保存的记录按照键进行排序，默认会按照键值的升序进行排序，当遍历TreeMap时得到的记录是排序过后的。

### Collection包结构，与Collections的区别。

Collection是一个集合的接口，提供了对集合对象进行操作的通用的方法。

在它下面的子接口：set、list、map

java.util.Collections是一个包装的类，包含有各种的有关集合操作的静态方法，比如包含对集合的搜索、排序、线程安全化等一系列的操作，此类不能被实例化，相当于是操作集合的工具类，服务于java的collection的框架。

### 介绍下Concurrent包

concurrent包基本有3个package组成    
(1)java.util.concurrent：提供大部分关于并发的接口和类，如BlockingQueue,Callable,ConcurrentHashMap,ExecutorService, Semaphore等

(2)java.util.concurrent.atomic：提供所有原子操作的类， 如AtomicInteger, AtomicLong等；

(3)java.util.concurrent.locks:提供锁相关的类, 如Lock, ReentrantLock, ReadWriteLock, Condition等；

concurrent包的优点：    
1. 首先，功能非常丰富，诸如线程池(ThreadPoolExecutor)，CountDownLatch等并发编程中需要的类已经有现成的实现，不需要自己去实现一套； 毕竟jdk1.4对多线程编程的主要支持几乎就只有Thread, Runnable,synchronized等

2. concurrent包里面的一些操作是基于硬件级别的CAS(compare and swap),就是在cpu级别提供了原子操作，简单的说就可以提供无阻塞、无锁定的算法； 而现代cpu大部分都是支持这样的算法的；

### Try-catch -finally，try里有return，finally还执行么？

任然会执行。

1、不管有木有出现异常，finally块中代码都会执行；  
2、当try和catch中有return时，finally仍然会执行；  
3、finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；  
4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

Excption与Error包结构。OOM你遇到过哪些情况，SOF你遇到过哪些情况。

### Java面向对象的三个特征与含义。

###### 封装：

是指将某事物的属性和行为包装到对象中，这个对象只对外公布需要公开的属性和行为，而这个公布也是可以有选择性的公布给其它对象。在[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库)中能使用private、protected、public三种修饰符或不用（即默认defalut）对外部对象访问该对象的属性和行为进行限制。

###### 继承：

是子对象可以继承父对象的属性和行为，亦即父对象拥有的属性和行为，其子对象也就拥有了这些属性和行为。

###### 多态：

java的引用变量有两种类型，一个是编译时的类型，一个是运行时的类型，编译时类型由申明该变量时的类型决定，运行时的类型由实际赋值给该变量的对象所决定，如果编译时的类型和运行时的类型不一致就可能出现所谓的多态。

在java中把一个子类的对象直接赋值给一个父类的引用变量，当运行该引用变量的方法时，其方法行为总是表现出子类方法的行为特征，这就有可能出现，相同类型的变量，调用同一个方法时呈现多种不同的行为特征，出现了“多态”

Override和Overload的含义和区别。

Overload：方法重载，在同一个类中，方法名相同，参数列表不同，至于方法的修饰符，反回值的类型，与方法的重载没有任何的联系。

Override:方法重写，两同两小一大

两同：方法名称相同、参数列表相同

两小：返回值类型要小或者相等；抛出的异常要小或者相等

一大：子类方法的访问权限要相等或者更大

Interface与abstract类的区别。

实例化：

都不能被实例化

类：一个类只能继承一次abstract类；一个类可以实现多个interface

数据成员：可以有自己的；接口的数据成员必须定义成static final的

方法：可以有私有的，非abstract方法必须实现；接口中不可以有私有的方法，默认都是public abstract的

变量：可以有私有的，其值可以在子类中重新定义，也可以重新赋值；接口中不可以有私有的成员变量，默认是public static final 实现类中不能去重新定义和改变其值

### Java IO与NIO

IO是面向流的，NIO是面向缓冲区

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

### 阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

### 选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### wait()和sleep()的区别。

(1)sleep方法，该方法属于thread类；wait方法属于object类

(2)sleep方法导致程序会暂停执行指定的时间，让出cpu给其他的线程，但是他还是监控状态的保持者，当到达指定的时间又会自动恢复运行。也就是调用sleep方法线程不会释放对象锁；调用wait方法会释放对象锁，进入到等待此对象的等待锁定池，只有当针对此对象调用了notify()方法后，才会获取对象锁进入运行的状态。

foreach与正常for循环效率对比。

For循环可以从前向后遍历，也可以从后向前遍历，可以不逐个遍历，通常用于已知次数的循环。

foreach循环不能向迭代变量赋值，通常对集合对象从头到位进行读取，其有优化的存在。

### Java与C++对比。

1、指针

java语言不提供指针，增加了自动的内存管理，有效的防止c/c++中的指针操作失误。

2、多重继承

C++支持多重继承，java不支持多重继承，但是允许实现多个接口。

3、数据类型和类

java将数据和方法结合起来，分装到类中，每个对象都可以实现自己的特点和方法；而c++允许将函数和变量定义全局的。

4、内存管理

java可以对所有的对象进行内存管理，自动回收不再使用的对象的内存；c++必须由程序员显式分配内存释放内存。

5、操作符的重载

C++支持操作符的重载，java不允许进行操作符的重载。

6、预处理功能

java不支持预处理功能，c++有一个预编译的阶段，也就是预处理器。

7、字符串

C++不支持字符串，java中支持字符串，是java的类对象。

8、数组

java引入了真正的数组，不同于c++中利用指针实现的伪数组。

9、类型的转换

C++中有时会出现数据类型的隐含转换，设计到自动强制类型的转换问题，比如存在将浮点数直接转换为整数的情况，java不支持自动的强制类型转换，如果需要，必须显示进行强制的类型转换。

10、异常

java中使用try{}catch(){}finally{}进行异常的处理，c++没有。

### HTTP和HTTPS的区别

https：是http的安全版本，利用ssl可以对所传输的数据进行加密，默认端口是443

### cookie和session的区别

(1)cookie数据存放在客户的浏览器上，session数据放在服务器上。

(2)cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗,如果主要考虑到安全应当使用session。

(3)session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用COOKIE。

(4)单个cookie在客户端的限制是4K，就是说一个站点在客户端存放的COOKIE不能4K。  
(5)所以：将登陆信息等重要信息存放为SESSION;其他信息如果需要保留，可以放在COOKIE中

### 网路

##### TCP三次握手、四次挥手，各个状态的名称和含义timewait的作用？

ACK:tcp协议规定，只有ack=1时才有效，在连接建立后所有发送的豹纹的ack=1

Syn(SYNchronization):在连接建立时用来同步序号。

当SYN=1而ACK=0:这是一个连接请求报文；

当对方同意建立连接时，则应该使得SYN=1而且ACK=1;

当SYN=1:这是一个连接请求或者连接接受报文

FIN(finis):终结的意思，用来释放一个连接。当fin=1,表示次报文段的发送方的数据已经发送完毕并要求释放连接。

A的状态：关闭状态--->同步已发送--->已建立

B的状态：关闭状态--->监听状态--->同步收到--->已建立

A:建立状态--->终止等待1--->终止等待2--->等待2MSL

B:建立状态--->关闭等待--->最后确认

##### Timewait的作用？

(1)为了保证A发送最后一个ACK报文能到达B,因为这个ACK报文有可能会丢失，这样会使得处在最后确认阶段的B收不到已经发送的FIN+ACK的确认信息，B会超时重传该报文段，在2MSL的时间内，A会收到信息，重传依次确认，重启该计时器。

(2)保证在2MSL的时间内，所有在本网络上产生的报文段都消失，使得在新的连接中不会出现旧的连接请求的报文段。

（2）SYN攻击防范

TCP/IP层次架构，每层的作用和协议

OSI模型：应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

TCP/IP模型：应用层、传输层、网络互联层、主机到网络层

##### 协议：

(1)应用层：FTP、TELNET、HTTP | SNMP、TFTP、NTP

将OSI模型的会话层和表示层整合成应用层，应用层面向不同的网络应用引入了不同的应用层协议。

(2)传输层：TCP|UDP

    功能是使得源端主机和目标端主机上的对等实体可以进行会话，定义了两种服务质量不同的协议，分别是TCP和UDP协议。

TCP协议是一个面向连接的、可靠的协议。它将一台主机发出的字节流无差错地发往互联网上的其他主机。在发送端，它负责把上层传送下来的字节流分成报文段并传递给下层。在接收端，它负责把收到的报文进行重组后递交给上层。TCP协议还要处理端到端的流量控制，以避免缓慢接收的接收方没有足够的缓冲区接收发送方发送的大量数据。　　  
    UDP协议是一个不可靠的、无连接协议。主要适用于不需要对报文进行排序和流量控制的场合。

(3)网络互联层：IP

网络互联层是整个TCP/IP协议栈的核心。功能是把分组发往目标网络或者主机。为了尽快发送分组，可能会沿着不同的路径同时进行分组传递。因此，分组到达的顺序和发送的顺序可能会不一致，这就需要上层必须对分组进行排序。同时它可以将不同类型的网络进行互联，完成拥塞控制的功能。

(4)主机到网络层：以太网、令牌环网、PPP

该层未被定义，具体的实现方式随着网络类型的不同而不同。

##### TCP拥塞控制

拥塞：计算机网络中的带宽、交换节点中的缓存和处理机都是网络中的资源，当在某一个时间，对网络中的某一个资源的需求超出了该资源所能提供的部分，网络的性能会变坏，就出现了拥塞。

拥塞控制：防止过多的数据注入到网路，使得网络中的路由器和链路不至于过载。拥塞控制是一个全局的过程，和流量控制不同，流量控制是点对点的通信量的控制。

慢开始和拥塞避免：

发送方维持一个叫做拥塞窗口的状态变量，拥塞窗口取决于网络的拥塞程度，并且会动态的变化。发送方让自己的发送窗口等于拥塞窗口，考虑接受方的接受能力，发送窗口可能会小于拥塞窗口。

慢开始算法：不要一开始就发送大量的数据，先探测下网络的拥塞程度，由小到大逐渐增加拥塞窗口的数量。

拥塞避免算法：让拥塞窗口缓慢增长，每进过一个往返时间就把发送方的拥塞窗口cwnd+1,而不是加倍，此时拥塞窗口按照线性的规律缓慢增长。

结合使用：为了防止拥塞窗口增长过大引发网络的拥塞，设置一个慢开始门限ssthresh状态变量。其用法：

当cwnd<ssthresh,使用慢开始算法

当cwnd>ssthresh,使用拥塞避免算法

当cwnd=ssthresh,慢开始算法和拥塞避免算法随意。

当遇到网络拥塞时，就把慢开始门限设置为出现拥塞时发送窗口大小的一半，同时将拥塞的窗口设置为1，再重新开始执行慢开始算法。

###### 滑动窗口是什么设计的？

窗口：是一段可以被发送者发送的字节序列，其连续的范围称为“窗口”

滑动：这段“允许发送的范围”是随着发送的过程而不断变换的，表现的形式就是“按照顺序滑动”

流量控制：

(1)TCP利用滑动窗口实现流量的控制机制

(2)如何考虑流量控制中的传输效率

流量控制，接受方传递信息给发送方，使其发送数据不要太快，是一种端到端的控制，主要的方式是返回的ack中会包含自己的接受的窗口的大小，发送方收到该窗口的大小时会控制自己的数据发送。

传递效率：单个发送字节单个确认，和窗口有一个空余即通知发送方发送一个字节，会增加网络中许多不必要的报文，因为会为一个字节数据添加40个字节的头部。

###### TCP/UDP的区别

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接。

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付。  
3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的  
UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）  
4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信  
5、TCP首部开销20字节;UDP的首部开销小，只有8个字节

6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠

###### TCP报文结构

紧急比特URG:URG=1,注解该报文应该尽快送达，而不需要按照原来的的队列次序依次送达

确认比特ACK:只有当ACK=1,确认序号字段才有意义

急迫比特PSH:当PSH=1时，注解恳求远地TCP将本报文段立即传送给应用层

复位比特RST:当注解呈现严重错误时，必须开释连接，进行重新的传输连接

同步比特SYN:当SYN=1而ACK=0时，这是一个连接请求报文段，若对方赞成连接请求会将SYN=1而且ACK=1

终止比特FIN:当FIN=1,注解字符串已经发送完毕，并请求开释传输连接。

###### HTTP的报文结构（请求报文+响应报文）

HTTP请求报文：(1)请求行+(2)请求头部+(3)请求正文

(1)请求行：请求方法+URL+协议版本

请求方法：常用GET、POST

协议版本:HTTP/主版本号.次版本号  常用HTTP/1.0和HTTP/1.1

(2)为请求报文添加的一些附加的信息，“名/值”组成，并且是每行一对 用冒号进行分割

在请求头部存在空行，表示请求头部的结束，接下来是请求正文！

###### 区别get和post方式

对于get方式没有请求的正文，对于post方式有请求的正文。

###### HTTP响应的报文格式：

(1)状态行+(2)响应头部+(3)响应正文

(1)状态行：协议版本+状态码+状态码描述

(2)响应头部：也是由键值对所组成

(3)响应正文，由服务器端接受数据

##### http request的几种类型（8种）

(1)OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'\*'的请求来测试服务器的功能性。   
(2)HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。   
(3)GET：向特定的资源发出请求。   
(4)POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。   
(5)PUT：向指定资源位置上传其最新内容。   
(6)DELETE：请求服务器删除Request-URI所标识的资源。   
(7)TRACE：回显服务器收到的请求，主要用于测试或诊断。

(8)CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

##### GET方式和POST方式对比：

GET方式：请求数据放在HTTP包头；使用明文传送，不安全；长度较小，一般为1024B；应用的场景为查询数据；如果传送的是英文数字或者是数字，直接发送，如果传送的是中文字符或则是其他的字符，则会进行BASE64编码

POST方式：请求数据放在HTTP正文；可明文或者密文传送，较为安全；长度一般没有限制；应用在修改数据上。

##### http1.0 和 http1.1的区别

(1)HTTP1.0 规定浏览与服务器只是保持短暂的连接，浏览器每次请求都需要和服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不去跟踪每个客户也不去记录每个客户过去的请求。HTTP1.0 不支持HOST请求字段

(2)HTTP1.1支持久连接，在一个TCP上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟；

允许客户端不用等待上一次请求的返回结果，就可以 去发送下一个请求，但是服务器端必须按照接受的客户端请求的先后顺序依次会送响应的结果，这样客户端才能够区分出每次请求的响应的内容。

HTTP1.0 支持HOST请求字段，这样就可以使用一个IP地址和端口号，在此基础上使用不同的主机名创建多个虚拟的WEB站点；

HTTP1.1 提供了与身份认证、状态管理和cache缓存等机制

##### http怎么去处理长连接

http1.1 默认支持长连接，即一次tcp连接，允许发送多个http请求。

当web服务器看到keep-alive值时，会建立长连接。

##### 电脑上访问一个网页的整个过程是怎样的？DNS、HTTP、TCP、OSPF、IP、ARP

步骤1：当访问www.baidu.com时，会先从本地的host文件中获取该域名对应的IP地址，如果找不到就会用DNS协议来获取IP，在该DNS协议中，计算机会由本地的DNS服务器来解析该域名，最终找到对应的IP地址。

步骤2：接下来是使用TCP协议，建立TCP连接，在建立连接之前需要，为了将给服务器的消息带给服务器，则需要OSPF\IP\ARP协议的支持，IP告诉该消息从哪里出发，去向那里；消息的传送会经过一个个的路由器，OSPF会利用路由算法找出最佳的通往目的地址的路径；ARP负责找到下一个节点的地址，ARP协议使用的MAC地址，整个的发送的过程涉及到每一个节点的MAP地址。

步骤3：通过步骤2的解析IP,现在可以和服务器建立TCP连接了，这时客户端便可以将Http请求数据发送给服务器端。服务器端进行处理，然后以http response的形式发送给客户端。

##### IP地址的分类

A类地址：1个字节的网络号+3个字节的主机地址  0.0.0.0~126.255.255.255

B类地址：2个字节的网络号+2个字节的主机地址  128.0.0.0~191.255.255.255

C类地址：3个字节的网络号+1个字节的主机地址  192.0.0.0~223.255.255.255

D类地址：多播地址

E类地址：保留为今后使用

##### 路由器和交换机的区别

交换机：为数据桢从一个端口到另外一个端口的转发提供了低时延、低开销的通路，使得任意端口接受的数据帧都能够从其他的端口送出。

路由器：网络连接和路由选择，用于网络层的数据转发。

### 如何设计一个高并发的系统？

① 数据库的优化，包括合理的事务隔离级别、SQL语句优化、索引的优化

② 使用缓存，尽量减少数据库 IO

③ 分布式数据库、分布式缓存

④ 服务器的负载均衡

### 设计模式

##### 简单工厂模式

有一个抽象的产品父类将所有的具体的产品抽象出来，达到复用的目的。同时有一个简单工厂维护一个对抽象产品的依赖，在该简单工厂中去负责实例的创建，在该工厂中去实例不同的对象，往往需要利用case判断语句去动态实例化相关的类。

##### 工厂方法模式

创建对象的接口，让子类去决定具体实例化的对象，把简单的内部逻辑的判断，转移到了客户端，让客户端去动态地实例化相关的子类。工厂方法模式克服了简单工厂违背开放-封闭原则的特点。

##### 抽象工厂模式

提供创建一系列相关或者相互依赖对象的接口，而无需指定他们具体的类。

##### 职责链模式

使得多个对象都有机会去处理请求，从而避免请求的 发送者和接受者之间的耦合关系，将这些对象连成一条链，并沿着这条链去传递该请求，直到有一个对象处理它为之。

##### 单例模式

###### （2）恶汉式的单例模式

利用静态static的方式进行实例化，在类被加载时就会创建实例。

/\*\*

 \* 饿汉式实现单例模式

 \*/

public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() {

    }

    public static Singleton getInstance() {

        return instance;

    }

}

###### (6)懒汉式实现单例模式

在被第一次引用时才去创建对象。

/\*\*

 \* 懒汉式实现单例模式

 \*/

public class Singleton {

    private static Singleton instance;//创建私有的静态变量

    private Singleton() {//私有的构造函数

    }

    // synchronized方法,多线程情况下保证单例对象唯一

public static synchronized Singleton getInstance() {

//如果实例对象为空，就重新去实例化

        if (instance == null) {

            instance = new Singleton();

        }

        return instance;

    }

}

分析：这中方法的实现，效率不高，因为该方法定义为同步的方法。

###### (7)双重锁实现的单例模式

/\*\*

 \* DCL实现单例模式

 \*/

public class Singleton {

    private static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {

        // 两层判空，第一层是为了避免不必要的同步

        // 第二层是为了在null的情况下创建实例

        if (instance == null) {

            synchronized (Singleton.class) {

                if (instance == null) {

                    instance = new Singleton();

                }

            }

        }

        return instance;

    }

}

分析：资源的利用率较高，在需要的时候去初始化实例，而且可以保证线程的安全，该方法没有去进行同步锁，效率比较好。

###### (8)静态内部类实现单例模式

/\*\*

 \* 静态内部类实现单例模式

 \*/

public class Singleton {

    private Singleton() {

}

//返回实例的方法

    public static Singleton getInstance() {

        return SingletonHolder.instance;

    }

    /\*\*

     \* 静态内部类

     \*/

private static class SingletonHolder {

    //静态私有的实例对象

        private static Singleton instance = new Singleton();

    }

}

分析：第一次加载类时不会去初始化instance,只有第一次调用getInstance()方法时，虚拟机才会加载内部类，初始化instance

可以保证线程的安全，单例对象的唯一，延迟了单例的初始化。

###### (9)枚举单例

/\*\*

 \* 枚举实现单例模式

 \*/

public enum SingletonEnum {

    INSTANCE;

    public void doSomething() {

        System.out.println("do something");

    }

}

分析：枚举实例的创建是线程安全的，即使反序列化也不会生成新的实例，在任何的情况下都是单例的。

##### 适配器模式

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口步兼容而不能一起工作的类变得可以一起工作。

target是我们所期望的接口的类型，包含一个request方法，通过使用adapter去实现该接口，并实现其中的request方法，在adapter中建立一个私有的adaptee对象，在adapter重写的方法中去调用specificRequest方法，这样适配器adapter就构建好了。只需要在客户端，创建adapter实例，调用request方法就可以利用多态的方式，实现了specificRequest()方法。

##### 观察者模式

定义了一种一对多的依赖关系，让多个观察者可以同时去监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有的观察者对象，使得他们能够自动更新自己。

Subject:把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者，可以增加删除观察者对象。

Observer:抽象观察者，为所有的具体的观察者定义一个接口，在得到主题时更新自己。

concreteObserver:具体的观察者，实现更新的方法

concreteSubject:具体的主题