andorid开发也做了3年有余了，也面试很多加企业，借此机会分享一下，我们中遇到过的问题以及解决方案吧，希望能够对正在找工作的andoird程序员有一定的帮助。特别献上整理过的50道面试题目

## 1.listView的优化方式

* 重用convertView
* viewHolder
* static class viewHolder
* 在列表里面有图片的情况下,监听滑动不加载图片
* 多个不同布局，可以创建不同的viewHolder和convertView进行重用

## 2.listView展示数据几种形式

* 从sqlite拉取数据源显示
* 从xml使用pull解析拉取数据源显示
* 从网络上拉取数据源显示

## 3.ipc

进程间通信主要包括管道, 系统IPC(Inter-Process Communication，进程间通信)(包括消息队列,信号,共享存储), 套接字(SOCKET). 目的:

* 数据传输：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
* 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
* 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
* 资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
* 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

进程通过与内核及其它进程之间的互相通信来协调它们的行为。Linux支持多种进程间通信（IPC）机制，信号和管道是其中的两种。除此之外，Linux还支持System V 的IPC机制（用首次出现的Unix版本命名）。

## 4.Parcel的机制

Android中的Parcel机制

实现了Bundle传递对象

使用Bundle传递对象，首先要将其序列化，但是，在Android中要使用这种传递对象的方式需要用到Android Parcel机制，即，Android实现的轻量级的高效的对象序列化和反序列化机制。

JAVA中的Serialize机制，译成串行化、序列化……，其作用是能将数据对象存入字节流当中，在需要时重新生成对象。主要应用是利用外部存储设备保存对象状态，以及通过网络传输对象等。

Android中的新的序列化机制

在Android系统中，定位为针对内存受限的设备，因此对性能要求更高，另外系统中采用了新的IPC（进程间通信）机制，必然要求使用性能更出色的对象传输方式。在这样的环境下，Parcel被设计出来，其定位就是轻量级的高效的对象序列化和反序列化机制。

Android中序列化有以下几个特征：

1. 整个读写全是在内存中进行，所以效率比JAVA序列化中使用外部存储器会高很多；
2. 读写时是4字节对齐的
3. 如果预分配的空间不够时，会一次多分配50%；
4. 对于普通数据，使用的是mData内存地址，对于IBinder类型的数据以及FileDescriptor使用的是mObjects内存地址。后者是通过flatten\_binder()和unflatten\_binder()实现的，目的是反序列化时读出的对象就是原对象而不用重新new一个新对象。

代码： activity代码：

 Intent mIntent =newIntent(this,ParcelableDemo.class);
 Bundle mBundle =newBundle();
 mBundle.putParcelable(PAR\_KEY, mPolice);
 mIntent.putExtras(mBundle);

实体类：

public class Police implements Parcelable {

 private String name;
 private int workTime;

 public String getName() {
 returnname;
 }

 public void setName(String name) {
 this.name = name;
 }

 public int getWorkTime() {
 returnworkTime;
 }

 public void setWorkTime(int workTime) {
 this.workTime = workTime;
 }

 public static final Parcelable.Creator<Police> CREATOR =newCreator<Police>() {

 @Override
 public Police createFromParcel(Parcel source) {
 Police police =newPolice();
 police.name = source.readString();
 police.workTime = source.readInt();
 returnpolice;
 }

 @Override
 public Police[] newArray(int size) {
 returnnewPolice[size];
 }
 };

 @Override
 public int describeContents() {
 return0;
 }

 @Override
 public void writeToParcel(Parcel parcel, int flags) {
 parcel.writeString(name);
 parcel.writeInt(workTime);
 }
}

## 5.JNI调用

1. Eclipse中新建android工程 工程名 JNItest Package名com.ura.test Activity名 JNItest 应用程序名 JNItest
2. 编辑main.xml
* <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout\_width="fill\_parent"
android:layout\_height="fill\_parent"
>
<TextView
android:id="@+id/JNITest"
android:layout\_width="fill\_parent"
android:layout\_height="wrap\_content"
android:text="@string/JNITest"
/>
</LinearLayout>
* (3)编辑java文件

package com.ura.test;

 import android.app.Activity;
 import android.os.Bundle;
 import android.widget.TextView;
 public class JNITest extends Activity {
 /\*\* Called when the activity is first created. \*/
 static {
 System.loadLibrary("JNITest");
 }
 public native String GetTest();
 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 String str =GetTest();
 TextView JNITest = (TextView)findViewById(R.id.JNITest);
 JNITest.setText(str);
 }
 }

(4)生成head文件 编译上面工程声称class文件，然后用javah工具生成c/c++ 头文件 javah -classpath bin -d jni com.ura.test.JNItest 生成的头文件如下

/\* DO NOT EDIT THIS FILE - it is machine generated \*/
 #include <jni.h>
 /\* Header for class com\_ura\_test\_JNITest \*/

 #ifndef \_Included\_com\_ura\_test\_JNITest
 #define \_Included\_com\_ura\_test\_JNITest
 #ifdef \_\_cplusplus
 extern "C" {
 #endif
 /\*
 \* Class: com\_ura\_test\_JNITest
 \* Method: GetTest
 \* Signature: ()Ljava/lang/String;
 \*/
 JNIEXPORT jstring JNICALL Java\_com\_ura\_test\_JNITest\_GetTest
 (JNIEnv \*, jobject);

 #ifdef \_\_cplusplus
 }
 #endif
 #endif

(5)编写c/c++文件如下

include "com\_ura\_test\_JNITest.h"

#define LOG\_TAG "JNITest"
#undef LOG
#include <utils/Log.h>

 JNIEXPORT jstring JNICALL Java\_com\_ura\_test\_JNITest\_GetTest
 (JNIEnv \* env, jobject obj)
 {
 return (\*env)->NewStringUTF(env, (char \*)"JNITest Native String");
 LOGD("Hello LIB!\n");

 }

(6)编写android.mk文件

LOCAL\_PATH:= $(call my-dir)
include $(CLEAR\_VARS)
LOCAL\_SRC\_FILES:= \
 com\_ura\_test\_JNITest.c
LOCAL\_C\_INCLUDES := \
 $(JNI\_H\_INCLUDE)
LOCAL\_SHARED\_LIBRARIES := libutils
LOCAL\_PRELINK\_MODULE := false
LOCAL\_MODULE := libJNITest
include $(BUILD\_SHARED\_LIBRARY)

(7)编译生成动态库 新建文件夹 ~/mydroid/external/libJNITest 把上面编写好的头文件，c/c++源文件，make文件拷贝进上面目录中 \* 需要注意的是把PRELINK\_MOUDULE设置成false 否则需要重新做成img文件再烧入。 在 ubuntu中执行

cd
cd mydroid/build/
envsetup.sh
cd ~/mydroid
cd external/libJNITest/
mm

编译成功的后会在下面目录中生成libJNITest.so文件 ~mydroid/out/target/product/generic/system/lib/ (8)在模拟器中执行程序 首先要把动态库拷进/system/lib中。 启动模拟器

adb shell
adb remount
adb push libJNITest.so /system/lib

确认拷贝成功

 cd /system/lib
 ls

然后不要关闭模拟器（关掉再开动态库就没了，因为模拟器rom是只读） 执行java程序JNITest 会看到屏幕上打印出

JNITest Native String 6.细谈四大组件

## activity

### 1.什么是activity？

四大组件之一,一般的,一个用户交互界面对应一个activity setContentView() ,// 要显示的布局 button.setOnclickLinstener{ } , activity 是Context的子类,同时实现了window.callback和keyevent.callback, 可以处理与窗体用户交互的事件.

里面不能进行耗时操作

我开发常用的的有ListActivity , PreferenceActivity ,TabAcitivty等…

如果界面有共同的特点或者功能的时候,还会自己定义一个BaseActivity.

### 2.activity生命周期？

Activity生命周期 1 完整生命周期 onCreate()
--> onStart() --> onResume() 可以在手机上看见activity ---> onPause() --> onStop() 看不见了 ---> onDestory() 销毁了

2 前台生命周期 onstart() ---> onStop()之间进行切换 onCreate() --> onStart() --> onResume() 现在有一个activity完全覆盖 onPause() ----> onStop() 如果上面的activity关闭 onRestart() ---> onStart() --> onResume()

3 可视生命周期 onResume() ---> onPause()之间进行切换 onCreate() --> onStart() --> onResume() 现在有一个activity没有完全覆盖 onPause() 如果上面的activity关闭 onResume()

### 3.横竖屏切换时候activity的生命周期？

这个生命周期跟清单文件里的配置有关系 1、不设置Activity的android:configChanges时，切屏会重新调用各个生命周期 默认首先销毁当前activity,然后重新加载

2、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

游戏开发中, 屏幕的朝向都是写死的.

### 4.如何将一个activity设置成窗口的样式？

可以自定义一个activity的样式,详细见手机卫士的程序详细信息 android:theme="@style/FloatActivity" E:9

### 5.activity启动模式？

Activity启动模式任务堆栈 Activity中的任务是与用户交互的一组Activity的集合，Activity会被按打开顺序安排在一个堆栈里。 任务栈：并不是Activity是Activity的引用（内存地址） standard 标准模式 每次激活Activity时都会创建Activity，并放入任务栈中 默认模式 singleTop 独享堆栈顶端 如果在任务的栈顶正好存在该Activity的实例，就重用该实例，否者就会创建新的实例并放入栈顶(即使栈中已经存在该Activity实例，只要不在栈顶，都会创建实例) 浏览器的书签 singleTask 独享任务堆栈 如果在栈中已经有该Activity的实例，就重用该实例(会调用实例的onNewIntent())。重用时，会让该实例回到栈顶，因此在它上面的实例将会被移除栈。如果栈中不存在该实例，将会创建新的实例放入栈中 浏览器的主页 singleInstance单例 在一个新栈中创建该Activity实例，并让多个应用共享该栈中的该Activity实例。一旦该模式的Activity的实例存在于某个栈中，任何应用再激活该Activity时都会重用该栈中的实例，其效果相当于多个应用程序共享一个应用，不管谁激活该Activity都会进入同一个应用中 通话界面 Singletop：如果重复使用上一次的Activity，就重用。 singleTask：如果使用已经实例化Activity，就重用，并且删除重用Activity前面的Activity，重用的Activity置顶任务栈。 singleInstance：在一个新栈中创建该Activity实例，并让多个应用共享该栈中的该Activity实例。（调用Activity和重用Activity不在一个栈中） singleTop 、singleTask 、singleInstance 优化性能、重用Activity。 6.后台activity被系统回收了怎么办？如果后台activity由于某种原因被系统回收了，如何保存之前状态？ 除了在栈顶的activity,其他的activity都有可能在内存不足的时候被系统回收,一个activity越处于栈底,被回收的可能性越大.

protected void onSaveInstanceState(Bundle outState) {
 super.onSaveInstanceState(outState);
 outState.putLong("id", 1234567890);
}
public void onCreate(Bundle savedInstanceState) {
//判断savedInstanceState是不是空.
//如果不为空就取出来
 super.onCreate(savedInstanceState);
}

### 7.如何退出activity,如何安全退出已调用多个activity的application？

退出activity 直接调用 finish () 方法 . //用户点击back键 就是退出一个activity 退出activity 会执行 onDestroy()方法 . 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。

 //安全结束进程 android.os.Process.killProcess(android.os.Process.myPid());

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。

 List<Activity> lists ; 在application 全集的环境里面
 lists = new ArrayList<Activity>();

lists.add(activity);

for(Activity activity: lists) { activity.finish(); }

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。 //给某个activity 注册接受接受广播的意图
registerReceiver(receiver, filter)

//如果过接受到的是 关闭activity的广播 就调用finish()方法 把当前的activity finish()掉

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。

上面是网上的一些做法.

其实 可以通过 intent的flag 来实现.. intent.setFlag(FLAG\_ACTIVITY\_CLEAR\_TOP)激活一个新的activity,然后在新的activity的oncreate方法里面 finish掉. 8.两activity之间怎么传递数据？ 基本数据类型可以通过. Intent 传递数据 在A activity中

Intent intent = new Intent();
intent.putExtra(name, value)
 Bundle bundle = new Bundle();
 bundle.putBoolean(key,value);
 intent.putExtras(bundle);
extras.putDouble(key, value)
// 通过intent putExtra 方法 基本数据类型 都传递

 Intent i = getIntent();
 i.getExtras();

intent.getStringExtra("key","value");
intent.getBooleanExtra("key","value")
 Bundle bundle = new Bundle();
 bumdle.putShort(key, value);
 intent.putExtras(bumdle);
intent.putExtras(bundle)

|  |
| --- |
| Application 全局里面存放 对象 ,自己去实现自己的application的这个类, |
| 基础系统的application , 每个activity都可以取到 |

让对象实现 implements Serializable 接口把对象存放到文件上. 让类实现Serializable 接口,然后可以通过ObjectOutputStream //对象输出流

 File file = new File("c:\1.obj");
 FileOutputStream fos = new FileOutputStream(file);
 ObjectOutputStream oos = new ObjectOutputStream(fos);

 Student stu = new Student();
 oos.writeObject(stu);

 //从文件中把对象读出来
 ObjectInputStream ois = new ObjectInputStream(arg0);
 Student stu1 = (Student) ois.readObject();

文件/网络

intent.setData(Uri) Uri.fromFile(); //大图片的传递

### 9.讲一讲对activity的理解？

把上面的几点用自己的心得写出来

## service

### 1.什么是Service以及描述下它的生命周期。Service有哪些启动方法，有什么区别，怎样停用Service？

在Service的生命周期中，被回调的方法比Activity少一些，只有onCreate, onStart, onDestroy, onBind和onUnbind。 通常有两种方式启动一个Service,他们对Service生命周期的影响是不一样的。 1 通过startService Service会经历 onCreate 到onStart，然后处于运行状态，stopService的时候调用onDestroy方法。 如果是调用者自己直接退出而没有调用stopService的话，Service会一直在后台运行。 2 通过bindService
Service会运行onCreate，然后是调用onBind， 这个时候调用者和Service绑定在一起。调用者退出了，Srevice就会调用onUnbind->onDestroyed方法。 所谓绑定在一起就共存亡了。调用者也可以通过调用unbindService方法来停止服务，这时候Srevice就会调用onUnbind->onDestroyed方法。 需要注意的是如果这几个方法交织在一起的话，会出现什么情况呢？ 一个原则是Service的onCreate的方法只会被调用一次，就是你无论多少次的startService又bindService，Service只被创建一次。 如果先是bind了，那么start的时候就直接运行Service的onStart方法，

如果先是start，那么bind的时候就直接运行onBind方法。

如果service运行期间调用了bindService，这时候再调用stopService的话，service是不会调用onDestroy方法的，service就stop不掉了，只能调用UnbindService, service就会被销毁

如果一个service通过startService 被start之后，多次调用startService 的话，service会多次调用onStart方法。多次调用stopService的话，service只会调用一次onDestroyed方法。

如果一个service通过bindService被start之后，多次调用bindService的话，service只会调用一次onBind方法。

多次调用unbindService的话会抛出异常。 2.service是否在main thread中执行, service里面是否能执行耗时的操作? 默认情况,如果没有显示的指定service所运行的进程, Service和activity是运行在当前app所在进程的main thread(UI主线程)里面 service里面不能执行耗时的操作(网络请求,拷贝数据库,大文件 ) 在子线程中执行 new Thread(){}.start();

特殊情况 ,可以在清单文件配置 service 执行所在的进程 ,让service在另外的进程中执行 3.怎么让在启动一个Activity是就启动一个service？ 在activity的onCreate()方法里面 startService(); 4.Activity怎么和service绑定，怎么在activity中启动自己对应的service？ startService() 一旦被创建 调用着无关 没法使用service里面的方法 bindService () 把service 与调用者绑定 ,如果调用者被销毁, service会销毁 bindService() 我们可以使用service 里面的方法 bindService(). 让activity能够访问到 service里面的方法 构建一个intent对象, Intent service = new Intent(this,MyService.class); 通过bindService的方法去启动一个服务, bindService(intent, new MyConn(), BIND\_AUTO\_CREATE); ServiceConnection 对象(重写onServiceConnected和OnServiceDisconnected方法) 和BIND\_AUTO\_CREATE. private class myconn implements ServiceConnection

 {

 public void onServiceConnected(ComponentName name, IBinder service) {
 // TODO Auto-generated method stub
 //可以通过IBinder的对象 去使用service里面的方法
 }

 public void onServiceDisconnected(ComponentName name) {
 // TODO Auto-generated method stub

 }

 }

5.不用service，B页面为音乐播放，从A跳转到B，再返回，如何使音乐继续播放？ 这个问题问的很山寨.默认不做任何处理,B里面的音乐都能播放. 遇到问题, 可以随机应变,灵活发挥,多考虑些细节,比如说这个题就可以这样说,说说你对startActivityForResult的理解() B的结束的时候 setResult()

A会调用到onActivityResult() 就会获取到resultCode A开启B的时候,用startActivityForResult()方法, B返回的时候把播放的状态信息返回给A ,A继续播放音乐. seekTo(resultCode) 6.什么是IntentService？有何优点？ 普通的service ,默认运行在ui main 主线程 Sdk给我们提供的方便的,带有异步处理的service类, 可以在OnHandleIntent() 处理耗时的操作 7.什么时候使用Service? 后台操作，耗时操作的时候

 拥有service的进程具有较高的优先级
官方文档告诉我们，Android系统会尽量保持拥有service的进程运行，只要在该service已经被启动(start)或者客户端连接(bindService)到它。当内存不足时，需要保持，拥有service的进程具有较高的优先级。

1． 如果service正在调用onCreate, onStartCommand或者onDestory方法，那么用于当前service的进程相当于前台进程以避免被killed。 2． 如果当前service已经被启动(start)，拥有它的进程则比那些用户可见的进程优先级低一些，但是比那些不可见的进程更重要，这就意味着service一般不会被killed. 3． 如果客户端已经连接到service (bindService),那么拥有Service的进程则拥有最高的优先级，可以认为service是可见的。 4． 如果service可以使用startForeg round(int, Notification)方法来将service设置为前台状态，那么系统就认为是对用户可见的，并不会在内存不足时killed。 如果有其他的应用组件作为Service,Activity等运行在相同的进程中，那么将会增加该进程的重要性。 1.Service的特点可以让他在后台一直运行,可以在service里面创建线程去完成耗时的操作. new Thread(){ TimerTask // 循环的执行一个定时的任务

}.start();
2.Broadcast receiver捕获到一个事件之后,可以起一个service来完成一个耗时的操作. ANR new Service()

 3.远程的service如果被启动起来,可以被多次bind, 但不会重新create. 索爱手机X10i的人脸识别的service可以被图库使用,可以被摄像机,照相机等程序使用.

画廊 摄像机 照相机 bindService() Ibinder的对象, 访问service 8.如何在让Service杀不死？ Android开发的过程中，每次调用startService(Intent)的时候，都会调用该Service对象的onStartCommand(Intent,int,int)方法，然后在onStartCommand方法中做一些处理。 从Android官方文档中，我们知道onStartCommand有4种int返回值，首先简单地讲讲int返回值的作用。 一、onStartCommand有4种返回值： START\_STICKY：如果service进程被kill掉，保留service的状态为开始状态，但不保留递送的intent对象。随后系统会尝试重新创建service，由于服务状态为开始状态，所以创建服务后一定会调用onStartCommand(Intent,int,int)方法。如果在此期间没有任何启动命令被传递到service，那么参数Intent将为null。 START\_NOT\_STICKY：“非粘性的”。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统不会自动重启该服务。 START\_REDELIVER\_INTENT：重传Intent。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统会自动重启该服务，并将Intent的值传入。 START\_STICKY\_COMPATIBILITY：START\_STICKY的兼容版本，但不保证服务被kill后一定能重启。

二、创建不被杀死的service 1.在service中重写下面的方法，这个方法有三个返回值， START\_STICKY（或START\_STICKY\_COMPATIBILITY）是service被kill掉后自动重写创建 @Override public int onStartCommand(Intent intent, int flags, int startId) { return START\_STICKY\_COMPATIBILITY; //return super.onStartCommand(intent, flags, startId); } 或 @Override public int onStartCommand(Intent intent, int flags, int startId) { flags = START\_STICKY; return super.onStartCommand(intent, flags, startId); // return START\_REDELIVER\_INTENT; } @Override public void onStart(Intent intent, int startId) { // 再次动态注册广播 IntentFilter localIntentFilter = new IntentFilter("android.intent.action.USER\_PRESENT"); localIntentFilter.setPriority(Integer.MAX\_VALUE);// 整形最大值 myReceiver searchReceiver = new myReceiver(); registerReceiver(searchReceiver, localIntentFilter); super.onStart(intent, startId); } 2.在Service的onDestroy()中重启Service. public void onDestroy() { Intent localIntent = new Intent(); localIntent.setClass(this, MyService.class); // 销毁时重新启动Service this.startService(localIntent); } 3.创建一个广播 public class myReceiver extends BroadcastReceiver { @Override public void onReceive(Context context, Intent intent) { context.startService(new Intent(context, Google.class)); } } 4.AndroidManifest.xml中注册广播myReceiver及MyService服务 注：解锁，启动，切换场景激活广播需加权限，如启动完成，及手机机状态等。 亲测ZTE U795手机Android 4.0.4版本adb push到system:persistent="true" 变成核心程序，在360杀掉进程的时候，myReceiver照样有效，保证service重生。呃 KILL问题： 1. settings 中stop service onDestroy方法中，调用startService进行Service的重启。 2.settings中force stop 应用 捕捉系统进行广播（action为android.intent.action.PACKAGE\_RESTARTED） 3. 借助第三方应用kill掉running task 提升service的优先级，程序签名，或adb push到system相较于/data/app下的应用，放在/system/app下的应用享受更多的特权，比如若在其Manifest.xml文件中设置persistent属性为true，则可使其免受out-of-memory killer的影响。如应用程序'Phone'的AndroidManifest.xml文件： ... 设置后app提升为系统核心级别

## Broadcast Receiver

### 1.什么是Broadcast Receiver

下面是Android Doc中关于BroadcastReceiver的概述：①广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的──比如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播──比如说，通知其它应用程序一些数据下载完成并处于可用状态。 ②应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的通知信息予以响应。所有的接收器均继承自BroadcastReceiver基类。 ③广播接收器没有用户界面。然而，它们可以启动一个activity来响应它们收到的信息，或者用NotificationManager来通知用户。通知可以用很多种方式来吸引用户的注意力──闪动背灯、震动、播放声音等等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。

有很多广播接收者 ,系统已经实现了. 广播分两种 有序广播 无序广播 指定接收者的有序广播 . sendOrderedBroadcast(intent,receiverPermission,resultReceiver,scheduler,initialCode,initialData,initialExtras) 接受者一定会获取到 广播的事件

sendStickyBroadcast(intent) //阴魂不散 广播接受者在onReceive 方法获取到广播的事件

Wifi设置 等待wifi状态更新完毕

是不可以被拦截掉的 -1000 - 1000 abortBroadcast();

代码配置优先级比xml配置优先级的级别高，因为代码运行在内存中，而清单在系统中

手机卫士中自定义一个broadcast receiver sms\_received

来获取短信到来的广播, 根据黑名单来判断是否拦截该短信. 画画板生成图片后,发送一个sd挂载的通知,通知系统的gallery去获取到新的图片. Intent intent = newIntent(Intent.ACTION\_MEDIA\_MOUNTED,Uri.parse("file://"+Environment.getExternalStorageDirectory())); sendBroadcast(intent); 2.什么时候使用Broadcast Receiver 用于接收系统的广播通知, 系统会有很多sd卡挂载,手机重启,广播通知,低电量,来电,来短信等…. 3.如何使用Broadcast Receiver 设置广播接收者的优先级,设置广播接受者的action名字 等… 详细见工程代码.

 <intent-filter android:priority="1000">
 <action android:name="android.intent.action.NEW\_OUTGOING\_CALL"/>
 </intent-filter>
 </receiver>
 <receiver android:name=".SmsReceiver">
 <intent-filter android:priority="1000">
 <action android:name="android.provider.Telephony.SMS\_RECEIVED"/>
 </intent-filter>
 </receiver>
 <receiver android:name=".BootCompleteReceiver">
 <intent-filter >
 <action android:name="android.intent.action.BOOT\_COMPLETED" />
 </intent-filter>
 </receiver>

可以通过代码 registerReceiver(receiver,filter)

## ContentProvider

### 1.什么是ContentProvider

ContentProvider内容提供者 ContentProvider 进程间通讯，进程间数据的访问/对外共享数据用 优点：提供了统一的访问方式 原理分析图

### 2.什么时候使用ContentProvider

需要访问别人的数据的时候 ### 3.如何使用ContentProvider 1.先是提供的数据类型等数据的类。package org.juetion.cp;

import android.net.Uri;
import android.provider.BaseColumns;

/\*\*
 \* 提供的数据类型等数据。
 \* Created by juetionke on 13-12-21.
 \*/
public class MyProviderMetaData {

 public static final String AUTHORIY = "org.juetion.cp.MyContentProvider";
 /\*\*
 \* 数据库名称
 \*/
 public static final String DATABASE\_NAME = "MyProvider.db";
 /\*\*
 \* 数据库版本
 \*/
 public static final int DATABASE\_VERSION = 1;
 /\*\*
 \* 表名
 \*/
 public static final String USERS\_TABLE\_NAME = "users";

 /\*\*
 \* 继承了BaseColumns，所以已经有了\_ID
 \*/
 public static final class UserTableMetaData implements BaseColumns {
 /\*\*
 \* 表名
 \*/
 public static final String TABLE\_NAME = "users";
 /\*\*
 \* 访问该ContentProvider的URI
 \*/
 public static final Uri CONTENT\_URI = Uri.parse("content://" + AUTHORIY + "/users");
 /\*\*
 \* 该ContentProvider所返回的数据类型定义
 \*/
 public static final String CONTENT\_TYPE = "vnd.android.cursor.dir/org.juetion.user";
 public static final String CONTENT\_TYPE\_ITEM = "vnd.android.cursor.item/org.juetion.user";
 /\*\*
 \* 列名
 \*/
 public static final String USER\_NAME = "name";
 public static final String USER\_AGE = "age";
 /\*\*
 \* 默认的排序方法
 \*/
 public static final String DEFAULT\_SORT\_ORDER = "\_id desc";
 }
}

2，继承ContentProvider的类：

package org.juetion.cp;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;

import org.juetion.sqlite3.DatabaseHelper;

import java.util.HashMap;

/\*\*
 \* Created by juetionke on 13-12-21.
 \*/
public class MyContentProvider extends ContentProvider {

 /\*\*
 \* 定义规则
 \*/
 public static final UriMatcher uriMatcher;
 public static final int USERS\_COLLECTION = 1;//用于标记
 public static final int USERS\_SINGLE = 2;//用于标记
 private DatabaseHelper databaseHelper;//这里的数据共享是共享Sqlite里的数据，当然，可以试用其他，如文本数据共享。
 static {
 uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);//试用一个没有规则的Uri。然后下面自己匹配。
 uriMatcher.addURI(MyProviderMetaData.AUTHORIY,"/users",USERS\_COLLECTION);//自己定义的规则，有点像路由器，是uri匹配的方案。
 uriMatcher.addURI(MyProviderMetaData.AUTHORIY,"/users/#",USERS\_SINGLE);//同上。
 }

 /\*\*
 \* 为列定义别名
 \*/
 public static HashMap<String,String> usersMap;
 static {
 usersMap = new HashMap<String, String>();
 usersMap.put(MyProviderMetaData.UserTableMetaData.\_ID, MyProviderMetaData.UserTableMetaData.\_ID);
 usersMap.put(MyProviderMetaData.UserTableMetaData.USER\_NAME, MyProviderMetaData.UserTableMetaData.USER\_NAME);
 usersMap.put(MyProviderMetaData.UserTableMetaData.USER\_AGE, MyProviderMetaData.UserTableMetaData.USER\_AGE);
 }

 @Override
 public boolean onCreate() {
 Log.i("juetion","onCreate");
 databaseHelper = new DatabaseHelper(getContext(), MyProviderMetaData.DATABASE\_NAME);//这里的实现，常见前篇关于Sqlite的文章。
 return true;
 }

 @Override
 public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
 Log.i("juetion","query");
 SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();//写入查询条件，有点像Hibernate。
 switch (uriMatcher.match(uri)) {//判断查询的是单个数据还是多个数据。
 case USERS\_SINGLE:
 sqLiteQueryBuilder.setTables(MyProviderMetaData.UserTableMetaData.TABLE\_NAME);//需要查询的表
 sqLiteQueryBuilder.setProjectionMap(usersMap);//列的别名定义
 sqLiteQueryBuilder.appendWhere(MyProviderMetaData.UserTableMetaData.\_ID + "=" + uri.getPathSegments().get(1));
 //查询条件，uri.getPathSegments().get(1)，getPathSegments是将内容根据／划分成list。
 break;
 case USERS\_COLLECTION:
 sqLiteQueryBuilder.setTables(MyProviderMetaData.UserTableMetaData.TABLE\_NAME);
 sqLiteQueryBuilder.setProjectionMap(usersMap);
 break;
 }
 String orderBy;//判断sortOrder是否为空，加入默认。
 if (TextUtils.isEmpty(sortOrder)) {
 orderBy = MyProviderMetaData.UserTableMetaData.DEFAULT\_SORT\_ORDER;
 } else {
 orderBy = sortOrder;
 }
 SQLiteDatabase sqLiteDatabase = databaseHelper.getWritableDatabase();
 Cursor cursor = sqLiteQueryBuilder.query(sqLiteDatabase, projection, selection, selectionArgs, null, null, sortOrder);//可以使用下面的方法，不过此时sqLiteDatabase将会没有用。
 //Cursor cursor = sqLiteDatabase.query(MyProviderMetaData.UserTableMetaData.TABLE\_NAME, projection, selection, selectionArgs, null, null, orderBy);
 cursor.setNotificationUri(getContext().getContentResolver(),uri);
 return cursor;
 }

 /\*\*
 \* 根据传入的URI，返回URI说表示的数据类型
 \* @param uri
 \* @return
 \*/
 @Override
 public String getType(Uri uri) {
 Log.i("juetion","getType");
 switch (uriMatcher.match(uri)) {//匹配uri的规则
 case USERS\_COLLECTION:
 return MyProviderMetaData.UserTableMetaData.CONTENT\_TYPE;
 case USERS\_SINGLE:
 return MyProviderMetaData.UserTableMetaData.CONTENT\_TYPE\_ITEM;
 default:
 throw new IllegalArgumentException("Unknown URI" + uri);
 }
 }

 @Override
 public Uri insert(Uri uri, ContentValues values) {
 Log.i("juetion","insert");
 SQLiteDatabase sqLiteDatabase = databaseHelper.getWritableDatabase();
 long rowId = sqLiteDatabase.insert(MyProviderMetaData.UserTableMetaData.TABLE\_NAME, null, values);
 if (rowId > 0) {
 Uri insertUserUri = ContentUris.withAppendedId(MyProviderMetaData.UserTableMetaData.CONTENT\_URI, rowId);//简单来说就是字符串拼凑一下。只不过是uri专用的。
 //通知监听器
 getContext().getContentResolver().notifyChange(insertUserUri,null);
 return insertUserUri;
 }else
 throw new IllegalArgumentException("Failed to insert row into" + uri);
 }

 @Override
 public int delete(Uri uri, String selection, String[] selectionArgs) {
 Log.i("juetion","delete");
 return 0;
 }

 @Override
 public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
 Log.i("juetion","update");
 return 0;
 }
}

还有重要的一点，再第二个APP的AndroidManifest.xml里面需要添加

<!-- provider name填写ContentProvider那个类的全称，authorities填写MyProviderMetaData里的AUTHORIY -->
 <provider
 android:authorities="org.juetion.cp.MyContentProvider"
 android:name="org.juetion.cp.MyContentProvider"/>

以下代码是在第一个APP里面的。 关于使用。只需要将uri的string提供给第一个APP。 例如在第一个APP的Activity调用数据插入：

ContentValues contentValues = new ContentValues();
 contentValues.put("name","zhangsan");
 contentValues.put("age",19);
 Uri uri = getContentResolver().insert(Uri.parse("content://org.juetion.cp.MyContentProvider/users"),contentValues);
 Log.i("juetion", "insert uri-->" + uri.toString());

例如在第一个APP的Activity调用数据的查询：

Cursor cursor = getContentResolver().query(Uri.parse("content://org.juetion.cp.MyContentProvider/users"),
 new String[]{"name", "age"},
 null, null, null);
 while (cursor.moveToNext()) {
 Log.i("juetion", cursor.getString(cursor.getColumnIndex("name")));
 }

### 4.请介绍下ContentProvider是如何实现数据共享的。

ContentProvider 可以屏蔽数据操作的细节 文件 xml MyContentProvider 可以在不同应用程序之间共享数据 sharedpreference db 把自己的数据通过uri的形式共享出去 android 系统下 不同程序 数据默认是不能共享访问

 需要去实现一个类去继承ContentProvider

 public class PersonContentProvider extends ContentProvider{
 public boolean onCreate(){
 //..
 }
query(Uri, String[], String, String[], String)
insert(Uri, ContentValues)
update(Uri, ContentValues, String, String[])
delete(Uri, String, String[])

联系人的信息 sms的内容content://sms/ } 安全共同点 android安全学习 签名作用 1.sharedUserId 一样并且签名一次 可以实现数据共享 2.升级应用 权限：细粒度的特权管理 权限与操作关联 应用需要显式申请权限 用户对权限可知（不可控） 对特权权限单独控制 四大组件 exported = true 等于public exported = false 等于private 默认组件private 如果该组件设置了intent-filter默认是public 如果同时希望是private，就需要主动设置expoted=false Securing Activities 可知指定权限才能启动activity service同上 BoradcastReceiver可以设置接发的权限 ContentPrivider 可设置读写Permission

## 7.多线程管理

本篇随笔将讲解一下Android的多线程的知识，以及如何通过AsyncTask机制来实现线程之间的通信。 一、Android当中的多线程 在Android当中，当一个应用程序的组件启动的时候，并且没有其他的应用程序组件在运行时，Android系统就会为该应用程序组件开辟一个新的线程来执行。默认的情况下，在一个相同Android应用程序当中，其里面的组件都是运行在同一个线程里面的，这个线程我们称之为Main线程。当我们通过某个组件来启动另一个组件的时候，这个时候默认都是在同一个线程当中完成的。当然，我们可以自己来管理我们的Android应用的线程，我们可以根据我们自己的需要来给应用程序创建额外的线程。 二、Main Thread 和 Worker Thread 在Android当中，通常将线程分为两种，一种叫做Main Thread，除了Main Thread之外的线程都可称为Worker Thread。 当一个应用程序运行的时候，Android操作系统就会给该应用程序启动一个线程，这个线程就是我们的Main Thread，这个线程非常的重要，它主要用来加载我们的UI界面，完成系统和我们用户之间的交互，并将交互后的结果又展示给我们用户，所以Main Thread又被称为UI Thread。 Android系统默认不会给我们的应用程序组件创建一个额外的线程，所有的这些组件默认都是在同一个线程中运行。然而，某些时候当我们的应用程序需要完成一个耗时的操作的时候，例如访问网络或者是对数据库进行查询时，此时我们的UI Thread就会被阻塞。例如，当我们点击一个Button，然后希望其从网络中获取一些数据，如果此操作在UI Thread当中完成的话，当我们点击Button的时候，UI线程就会处于阻塞的状态，此时，我们的系统不会调度任何其它的事件，更糟糕的是，当我们的整个现场如果阻塞时间超过5秒钟(官方是这样说的)，这个时候就会出现 ANR (Application Not Responding)的现象，此时，应用程序会弹出一个框，让用户选择是否退出该程序。对于Android开发来说，出现ANR的现象是绝对不能被允许的。 另外，由于我们的Android UI控件是线程不安全的，所以我们不能在UI Thread之外的线程当中对我们的UI控件进行操作。因此在Android的多线程编程当中，我们有两条非常重要的原则必须要遵守： 绝对不能在UI Thread当中进行耗时的操作，不能阻塞我们的UI Thread 不能在UI Thread之外的线程当中操纵我们的UI元素 三、如何处理UI Thread 和 Worker Thread之间的通信 既然在Android当中有两条重要的原则要遵守，那么我们可能就有疑问了？我们既不能在主线程当中处理耗时的操作，又不能在工作线程中来访问我们的UI控件，那么我们比如从网络中要下载一张图片，又怎么能将其更新到UI控件上呢？这就关系到了我们的主线程和工作线程之间的通信问题了。在Android当中，提供了两种方式来解决线程直接的通信问题，一种是通过Handler的机制(这种方式在后面的随笔中将详细介绍),还有一种就是今天要详细讲解的 AsyncTask 机制。 四、AsyncTask AsyncTask：异步任务，从字面上来说，就是在我们的UI主线程运行的时候，异步的完成一些操作。AsyncTask允许我们的执行一个异步的任务在后台。我们可以将耗时的操作放在异步任务当中来执行，并随时将任务执行的结果返回给我们的UI线程来更新我们的UI控件。通过AsyncTask我们可以轻松的解决多线程之间的通信问题。 怎么来理解AsyncTask呢？通俗一点来说，AsyncTask就相当于Android给我们提供了一个多线程编程的一个框架，其介于Thread和Handler之间，我们如果要定义一个AsyncTask，就需要定义一个类来继承AsyncTask这个抽象类，并实现其唯一的一个 doInBackgroud 抽象方法。要掌握AsyncTask，我们就必须要一个概念，总结起来就是: 3个泛型，4个步骤。 3个泛型指的是什么呢？我们来看看AsyncTask这个抽象类的定义，当我们定义一个类来继承AsyncTask这个类的时候，我们需要为其指定3个泛型参数： AsyncTask　 Params: 这个泛型指定的是我们传递给异步任务执行时的参数的类型 Progress: 这个泛型指定的是我们的异步任务在执行的时候将执行的进度返回给UI线程的参数的类型 Result: 这个泛型指定的异步任务执行完后返回给UI线程的结果的类型 我们在定义一个类继承AsyncTask类的时候，必须要指定好这三个泛型的类型，如果都不指定的话，则都将其写成Void，例如： AsyncTask 4个步骤：当我们执行一个异步任务的时候，其需要按照下面的4个步骤分别执行 onPreExecute(): 这个方法是在执行异步任务之前的时候执行，并且是在UI Thread当中执行的，通常我们在这个方法里做一些UI控件的初始化的操作，例如弹出要给ProgressDialog doInBackground(Params... params): 在onPreExecute()方法执行完之后，会马上执行这个方法，这个方法就是来处理异步任务的方法，Android操作系统会在后台的线程池当中开启一个worker thread来执行我们的这个方法，所以这个方法是在worker thread当中执行的，这个方法执行完之后就可以将我们的执行结果发送给我们的最后一个 onPostExecute 方法，在这个方法里，我们可以从网络当中获取数据等一些耗时的操作 onProgressUpdate(Progess... values): 这个方法也是在UI Thread当中执行的，我们在异步任务执行的时候，有时候需要将执行的进度返回给我们的UI界面，例如下载一张网络图片，我们需要时刻显示其下载的进度，就可以使用这个方法来更新我们的进度。这个方法在调用之前，我们需要在 doInBackground 方法中调用一个 publishProgress(Progress) 的方法来将我们的进度时时刻刻传递给 onProgressUpdate 方法来更新 onPostExecute(Result... result): 当我们的异步任务执行完之后，就会将结果返回给这个方法，这个方法也是在UI Thread当中调用的，我们可以将返回的结果显示在UI控件上 为什么我们的AsyncTask抽象类只有一个 doInBackground 的抽象方法呢？？原因是，我们如果要做一个异步任务，我们必须要为其开辟一个新的Thread，让其完成一些操作，而在完成这个异步任务时，我可能并不需要弹出要给ProgressDialog，我并不需要随时更新我的ProgressDialog的进度条，我也并不需要将结果更新给我们的UI界面，所以除了 doInBackground 方法之外的三个方法，都不是必须有的，因此我们必须要实现的方法是 doInBackground 方法。 五、通过AsyncTask来从网络上下载一张图片 下面我们就通过两个代码示例，来看看如何通过AsyncTask来从网络上下载一张图片，并更新到我们的ImageView控件上。 ①下载图片时，弹出一个ProgressDialog，但是不显示实时进度 我们来看看布局文件：

<RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
>

 <ImageView
 android:id="@+id/imageView"
 android:layout\_width="wrap\_content"
 android:layout\_height="200dp"
 android:layout\_alignParentRight="true"
 android:layout\_alignParentTop="true"
 android:scaleType="fitCenter"/>

 <Button
 android:id="@+id/button"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content"
 android:layout\_below="@+id/imageView"
 android:layout\_centerHorizontal="true"
 android:layout\_marginTop="41dp"
 android:text="从网络上下载一张图片"/>

</RelativeLayout>

就是很简单的一个ImageView控件和一个Button控件，当点击Button控件时，弹出一个ProgressDialog，然后开启一个异步任务，从网络中下载一张图片，并更新到我们的ImageView上。这里还要注意一点，如果我们要使用手机访问网络，必须还要给其授权才行，在后续的学习当中，将会详细讲解Android当中的授权的知识。我们来看看 AndroidManifest.xml文件：

<?xml version="1.0" encoding="utf-8"?><manifestxmlns:android="http://schemas.android.com/apk/res/android"
 package="com.xiaoluo.android\_asynctast"
 android:versionCode="1"
 android:versionName="1.0"
>

 <uses-sdk
 android:minSdkVersion="8"
 android:targetSdkVersion="18"/>

 <!-- 授权手机能够访问网络 -->
 <uses-permissionandroid:name="android.permission.INTERNET"/>

 <application
 android:allowBackup="true"
 android:icon="@drawable/ic\_launcher"
 android:label="@string/app\_name"
 android:theme="@style/AppTheme"
>

 <activity
 android:name="com.xiaoluo.android\_asynctast.MainActivity"
 android:label="@string/app\_name"
>

 <intent-filter>

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

 <categoryandroid:name="android.intent.category.LAUNCHER"/>
 </intent-filter>

 </activity>

 </application>

</manifest>

接下来我们来看看我们的Activity代码：

publicclass MainActivityextends Activity { private Button button; private ImageView imageView; private ProgressDialog progressDialog; privatefinal String IMAGE\_PATH = "http://developer.android.com/images/home/kk-hero.jpg"; // private final String IMAGE\_PATH2 = "http://ww2.sinaimg.cn/mw690/69c7e018jw1e6hd0vm3pej20fa0a674c.jpg"; @Override protectedvoid onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity\_main);

 button = (Button)findViewById(R.id.button);
 imageView = (ImageView)findViewById(R.id.imageView);
 // 弹出要给ProgressDialog
 progressDialog =new ProgressDialog(MainActivity.this);
 progressDialog.setTitle("提示信息");
 progressDialog.setMessage("正在下载中，请稍后......");
 // 设置setCancelable(false); 表示我们不能取消这个弹出框，等下载完成之后再让弹出框消失
 progressDialog.setCancelable(false);
 // 设置ProgressDialog样式为圆圈的形式 progressDialog.setProgressStyle(ProgressDialog.STYLE\_SPINNER);

 button.setOnClickListener(new View.OnClickListener()
 {
 @Override
 publicvoid onClick(View v)
 {　　　　　　　　　// 在UI Thread当中实例化AsyncTask对象，并调用execute方法
 new MyAsyncTask().execute(IMAGE\_PATH);
 }
 });
}

/\*\*
 \* 定义一个类，让其继承AsyncTask这个类
 \* Params: String类型，表示传递给异步任务的参数类型是String，通常指定的是URL路径
 \* Progress: Integer类型，进度条的单位通常都是Integer类型
 \* Result：byte[]类型，表示我们下载好的图片以字节数组返回
 \*@author xiaoluo
 \*
 \*/
publicclass MyAsyncTaskextends AsyncTask<String, Integer,byte[]>
{
 @Override
 protectedvoid onPreExecute()
 {
 super.onPreExecute();
 // 在onPreExecute()中我们让ProgressDialog显示出来 progressDialog.show();
 }
 @Override
 protectedbyte[] doInBackground(String... params)
 {
 // 通过Apache的HttpClient来访问请求网络中的一张图片
 HttpClient httpClient =new DefaultHttpClient();
 HttpGet httpGet =new HttpGet(params[0]);
 byte[] image =newbyte[]{};
 try
 {
 HttpResponse httpResponse = httpClient.execute(httpGet);
 HttpEntity httpEntity = httpResponse.getEntity();
 if(httpEntity !=null && httpResponse.getStatusLine().getStatusCode() == HttpStatus.SC\_OK)
 {
 image = EntityUtils.toByteArray(httpEntity);
 }
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 finally
 {
 httpClient.getConnectionManager().shutdown();
 }
 return image;
 }
 @Override
 protectedvoid onProgressUpdate(Integer... values)
 {
 super.onProgressUpdate(values);
 }
 @Override
 protectedvoid onPostExecute(byte[] result)
 {
 super.onPostExecute(result);
 // 将doInBackground方法返回的byte[]解码成要给Bitmap
 Bitmap bitmap = BitmapFactory.decodeByteArray(result, 0, result.length);
 // 更新我们的ImageView控件 imageView.setImageBitmap(bitmap);
 // 使ProgressDialog框消失 progressDialog.dismiss();
 }
}

@Override
publicboolean onCreateOptionsMenu(Menu menu)
{
 getMenuInflater().inflate(R.menu.main, menu);
 returntrue;
}

} ``` 我们来看看效果图：

②带有进度条更新的下载一张网络图片 下面这个代码示例，将会在下载图片的时候，显示进度条的更新，配置文件都不变，我们来看看Activity代码：

publicclass MainActivityextends Activity
{
 private Button button;
 private ImageView imageView;
 private ProgressDialog progressDialog;
 privatefinal String IMAGE\_PATH = "http://developer.android.com/images/home/kk-hero.jpg";
// private final String IMAGE\_PATH2 = "http://ww2.sinaimg.cn/mw690/69c7e018jw1e6hd0vm3pej20fa0a674c.jpg"; @Override
 protectedvoid onCreate(Bundle savedInstanceState)
 {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity\_main);

 button = (Button)findViewById(R.id.button);
 imageView = (ImageView)findViewById(R.id.imageView);
 // 弹出要给ProgressDialog
 progressDialog =new ProgressDialog(MainActivity.this);
 progressDialog.setTitle("提示信息");
 progressDialog.setMessage("正在下载中，请稍后......");
 // 设置setCancelable(false); 表示我们不能取消这个弹出框，等下载完成之后再让弹出框消失
 progressDialog.setCancelable(false);
 // 设置ProgressDialog样式为水平的样式 progressDialog.setProgressStyle(ProgressDialog.STYLE\_HORIZONTAL);

 button.setOnClickListener(new View.OnClickListener()
 {
 @Override
 publicvoid onClick(View v)
 {
 new MyAsyncTask().execute(IMAGE\_PATH);
 }
 });
 }

 /\*\*
 \* 定义一个类，让其继承AsyncTask这个类
 \* Params: String类型，表示传递给异步任务的参数类型是String，通常指定的是URL路径
 \* Progress: Integer类型，进度条的单位通常都是Integer类型
 \* Result：byte[]类型，表示我们下载好的图片以字节数组返回
 \*@author xiaoluo
 \*
 \*/
 publicclass MyAsyncTaskextends AsyncTask<String, Integer,byte[]>
 {
 @Override
 protectedvoid onPreExecute()
 {
 super.onPreExecute();
 // 在onPreExecute()中我们让ProgressDialog显示出来 progressDialog.show();
 }
 @Override
 protectedbyte[] doInBackground(String... params)
 {
 // 通过Apache的HttpClient来访问请求网络中的一张图片
 HttpClient httpClient =new DefaultHttpClient();
 HttpGet httpGet =new HttpGet(params[0]);
 byte[] image =newbyte[]{};
 try
 {
 HttpResponse httpResponse = httpClient.execute(httpGet);
 HttpEntity httpEntity = httpResponse.getEntity();
 InputStream inputStream =null;
 ByteArrayOutputStream byteArrayOutputStream =newByteArrayOutputStream();
 if(httpEntity !=null && httpResponse.getStatusLine().getStatusCode() == HttpStatus.SC\_OK)
 {
 // 得到文件的总长度
 long file\_length = httpEntity.getContentLength();
 // 每次读取后累加的长度
 long total\_length = 0;
 int length = 0;
 // 每次读取1024个字节
 byte[] data =newbyte[1024];
 inputStream = httpEntity.getContent();
 while(-1 != (length = inputStream.read(data)))
 {
 // 每读一次，就将total\_length累加起来
 total\_length += length;
 // 边读边写到ByteArrayOutputStream当中
 byteArrayOutputStream.write(data, 0, length);
 // 得到当前图片下载的进度
 int progress = ((int)(total\_length/(float)file\_length) \* 100);
 // 时刻将当前进度更新给onProgressUpdate方法 publishProgress(progress);
 }
 }
 image = byteArrayOutputStream.toByteArray();
 inputStream.close();
 byteArrayOutputStream.close();
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 finally
 {
 httpClient.getConnectionManager().shutdown();
 }
 return image;
 }
 @Override
 protectedvoid onProgressUpdate(Integer... values)
 {
 super.onProgressUpdate(values);
 // 更新ProgressDialog的进度条
 progressDialog.setProgress(values[0]);
 }
 @Override
 protectedvoid onPostExecute(byte[] result)
 {
 super.onPostExecute(result);
 // 将doInBackground方法返回的byte[]解码成要给Bitmap
 Bitmap bitmap = BitmapFactory.decodeByteArray(result, 0, result.length);
 // 更新我们的ImageView控件 imageView.setImageBitmap(bitmap);
 // 使ProgressDialog框消失 progressDialog.dismiss();
 }
 }

 @Override
 publicboolean onCreateOptionsMenu(Menu menu)
 {
 getMenuInflater().inflate(R.menu.main, menu);
 returntrue;
 }

}

我们来看看效果图：

这样我们就能够通过AsyncTask来实现从网络中下载一张图片，然后将其更新到UI控件中，并时时刻刻的更新当前的进度这个功能了。 六、AsyncTask的重要知识点 在上面两节已经详细讲解了AsyncTask的工作原理了，这里我们还要补充一下AsyncTask的一些其他知识点： 1.Cancelling a Task 我们可以在任何时刻来取消我们的异步任务的执行，通过调用 cancel(boolean)方法，调用完这个方法后系统会随后调用 isCancelled() 方法并且返回true。如果调用了这个方法，那么在 doInBackgroud() 方法执行完之后，就不会调用 onPostExecute() 方法了，取而代之的是调用 onCancelled() 方法。为了确保Task已经被取消了，我们需要经常调用 isCancelled() 方法来判断，如果有必要的话。 2.在使用AsyncTask做异步任务的时候必须要遵循的原则： AsyncTask类必须在UI Thread当中加载，在Android Jelly\_Bean版本后这些都是自动完成的 AsyncTask的对象必须在UI Thread当中实例化 execute方法必须在UI Thread当中调用 不要手动的去调用AsyncTask的onPreExecute, doInBackground, publishProgress, onProgressUpdate, onPostExecute方法，这些都是由Android系统自动调用的 AsyncTask任务只能被执行一次

到此，有关AsyncTask的总结就到此为止了，本篇随笔主要讲解了Android中的多线程知识，并且详细地讲解了 AsyncTask 异步任务的概念和实现机制，并通过实例来了解 AsyncTask 的执行过程，最后还补充了 AsyncTask 的一些重要知识点，包括如何取消一个 AsyncTask 以及，我们在使用 AsyncTask 时所必须遵循的规则。 8.android内存优化及管理

1、使用优化过的数据容器。 在Android framework下，建议使用优化过的数据容器比如：SparseArray,SparseBooleanArray,LongSparseArray。通用的HashMap实现的内存使用率非常的低，因为他需要为每一个mapping创建一个分离的entry object。另外，SparseArray类避免了系统对有些key的自动装箱，因而带来了更高的效率。 2、注意内存的开销。[size=12.800000190734863px] 注意你使用的语言和第三方库的成本和开销，要自始至终的将这些因素考虑在你的程序设计中。通常，有些事情表面上看着没什么问题但实际上的开销让人惊叹。比如： ·枚举相对于静态常量来说，需要两倍甚至更多的内存。你应该完全避免在Android中使用枚举。 [size=12.666666984558105px]·每一个在java中的类（包括匿名内部类）使用大约500 bytes的代码量。 ·每一个类的实例拥有12-16 bytes的RAM消耗。 ·放置一个单独的实体到HashMap中，一个额外加的实体对象分配需要花费32 bytes。 [size=12.666666984558105px]3、关于代码的抽象 抽象是一个好的编程的基础，因为抽象可以提高代码的灵活性和可维护性。然而抽象也带来了一定的花销，一般情况下，他们有更多的代码需要执行，需要更多的时间和更多RAM来将这些代码映射到内存中。因此，如果你的抽象不能带来巨大的好处，你就应该割掉你的赘肉。 4、避免依赖注入框架 虽然注入框架给我们带来了很多方便，但是在使用这些框架的时候，框架需要花费很多时间来扫描我们自己写的代码，甚至会将哪些你根本不需要的东西也加载到内存中。 5、小心的使用扩展库 很多扩展库的代码不是针对手机环境开发的，可能在用到移动客户端的时候会导致很低的效率。因此在使用之前，需要评估一下其占用内存的大小。 即使库针对手机开发，也会有潜在的危险，因为每一个Library做的事情不尽相同。比如，一个Library使用nano protobufs而另一个使用micro protobufs。现在，在你的app中就有两个protobuf。类似情况经常发生。 6、使用混淆器移除不必要的代码 ProGuard工具通过移除无用代码，使用语意模糊来保留类，字段和方法来压缩，优化和混淆代码。可以使你的代码更加完整，更少的RAM 映射页。 7、使用多个进程（注意是process 不是 thread ok？） 如果这适合你的app，可能帮助你管理你的app的内存就是将你的app多个部分分配到多个进程中。该技术必须小心使用并且大多数应用不应该运行在多个进程下。这个技术的主要应用是后台工作跟天台工作一样重要的情况下。典型应用就是：当音乐播放器从服务器下载并长时间播放音乐，一般将其分为两个进程：一个是UI，另一个位置后台服务的运行。 like this:

1 <serviceandroid:name=".PlaybackService" android:process=":background"/>

[size=12.666666984558105px]process后面需要记住要有个":",这表示该进程属于你的app。 一般情况下，一块基本的空进程需要的内存大小在1.4m左右。

1 adb shell dumpsys meminfo com.example.android.apis:empty
8、基本性能优化方法的基本原则：

1）不要做你不必要的工作；

2）不要申请不必要的内存； 例如，你明明知道一个方法返回一个String之后，你需要对这个String重新进行修改，那么就不要返回一个String，返回一个StringBuffer会是你更好的选择。 再比如，使用int比使用Integer占用更少的空间。这个大家肯定都是晓得的。 数组比一个Map拥有更好的性能。 如果你的方法不需要访问类字段，那么让你的方法是static的吧，这将会带来15%-20%速度的提升。 对于常量，请尽量使用static and final定义。如果使用final定义常量之后，会减少编译器在类生成时初始化方法调用时对常量的存储，对于int型常量，将会直接使用其数值来进行替换，而对于String对象将会使用相对廉价的“string constant”指令来替换字段查找表。虽然这个方法并不完全对所有类型都有效，但是，将常量声明为static final绝对是一个好的做法。 避免Getters/Setters。虽然在一般的面向对象的设计模式中使用Getter和Setter是稀松平常的事情，但是在Android中使用getters/Setters是一个非常糟糕的主意，方法的调用相对于直接查找字段来说十分的昂贵。在没有JIT的情况下，直接对字段进行访问要比通过Getter访问快了近3倍。在有JIT的情况下，前者比后者快近7倍。 使用最新的循环方式。比如增强for。 避免使用浮点类型。在某些可以的情况下，将浮点替换成整型数据，然后进行计算会得到更精确的结果和更快的速度。 小心使用Native Methods。这里需要纠正的是，Native 方法并不一定能提高你应用的速度，有些甚至会拖后腿，因为，首先来说就需要一部分开销在Java-native transition上，而且JIT并不能对其进行优化。另外你需要为每个你想要在其上运行的系统结构上进行编译；即便是同一个处理器上，你也可能需要多个版本，比如为G1上的ARM处理器编译的就不能很好的在Nexus One的ARM上运行。Native代码最主要的用途是，你已经有了很多native 代码，并且你迫切希望接入Android中。而不是使用Native Method来提高你应用中某部分代码的运行速度。 [size=12.666666984558105px] 对于效率的提高除了使用遵守上面两条外基本准则外，选择合适的算法和数据结构也是非常关键的。

9、关于UI上的一些问题[size=12.800000190734863px] Hierarchy Viewer[size=12.666666984558105px] [size=12.666666984558105px]通过他，可以看到你自己的Layout文件存在的问题。你可以看到你的Layout每一部分计算，布局，渲染所需要的时间。尽量的使你的Layout扁平话，深度最好保持 在三层之内[size=12.666666984558105px] 。RelativeLayout[size=12.666666984558105px] 是解决使用LinearLayout堆叠多层问题的利剑。那些为了方便 使[size=12.666666984558105px] 用LinearLayout的layout\_weight属性[size=12.666666984558105px] 的哥们，需要重点注意，这个属性真的可以减慢measure速度。所以在使用之前，一定要再三考虑，是否真的不能通过其他方法来完成你要的效果？ 官方文档上 推荐使用RelativeLayout和GridLayout来避免Layout深度过深的问题[size=12.666666984558105px] 。 之前看文档，Google提供一个叫 ViewSub[size=12.666666984558105px] 的控件来优化那些不是必须要立即在UI上显示的控件，感兴趣的同学可以去看看。在API Level 1中就提供了这个东西，但是在实际开发中很少见到有人用或者提及（可能是我孤陋寡闻，公司就两个Android开发，另一个还要转IOS，我们俩的Android技术就代表了我们公司的Android技术能力，想想真悲哀！），其实蛮好用的。 重用Layout。可以使用 将其他布局嵌入到当前布局中。 ListView的优化：ViewHolder的使用；AsyncTask的使用；针对ListView当前滑动状态，对图片数据的加载进行控制；（ListView在配以AsyncTask加载图片时需要注[size=12.666666984558105px] 意图片的加载完显示的位置以及图片的缓存问题，具体可以参考Google的Demo[size=12.666666984558105px] ）[size=12.666666984558105px] [size=12.800000190734863px]10、将大消耗操作交给多个线程。 11、如果你的应用需要发送Broadcast但是又不希望别的应用获取到，或者你不希望处理别的应用发送的同样的action，那么请使用LocalBroadcastManager。 [size=12.666666984558105px] 该类是在Android Support v4中提供的，用来在同一个应用内的不同组件之间发送Broadcast。好处上面说了，可以保证应用的私密性。会比全局广播有更高的效率，但是官方文档没有说明具体数值。具体使用方法：

1 LocalBroadcastManager.getInstance(this).registerReceiver( mStateReceiver, mStatusIntentFilter);LocalBroadcastManager.getInstance(this).sendBroadcast(localBroadcastIntent);

使用oc计数器的方式可以更好的控制内存

Android OnLowMemory和OnTrimMemory

OnLowMemory OnLowMemory是Android提供的API，在系统内存不足，所有后台程序（优先级为background的进程，不是指后台运行的进程）都被杀死时，系统会调用OnLowMemory。系统提供的回调有： Application.onLowMemory() Activity.OnLowMemory() Fragement.OnLowMemory() Service.OnLowMemory() ContentProvider.OnLowMemory() 除了上述系统提供的API，还可以自己实现ComponentCallbacks，通过API注册，这样也能得到OnLowMemory回调。例如： public static class MyCallback implements ComponentCallbacks {

 @Override
 public void onConfigurationChanged(Configuration arg) {

 }

 @Override
 public void onLowMemory() {
 //do release operation
 }
}

然后，通过Context.registerComponentCallbacks ()在合适的时候注册回调就可以了。通过这种自定义的方法，可以在很多地方注册回调，而不需要局限于系统提供的组件。 OnTrimMemory OnTrimMemory是Android 4.0之后提供的API，系统会根据不同的内存状态来回调。系统提供的回调有： Application.onTrimMemory() Activity.onTrimMemory() Fragement.OnTrimMemory() Service.onTrimMemory() ContentProvider.OnTrimMemory() OnTrimMemory的参数是一个int数值，代表不同的内存状态： TRIM\_MEMORY\_COMPLETE：内存不足，并且该进程在后台进程列表最后一个，马上就要被清理 TRIM\_MEMORY\_MODERATE：内存不足，并且该进程在后台进程列表的中部。 TRIM\_MEMORY\_BACKGROUND：内存不足，并且该进程是后台进程。 TRIM\_MEMORY\_UI\_HIDDEN：内存不足，并且该进程的UI已经不可见了。
以上4个是4.0增加 TRIM\_MEMORY\_RUNNING\_CRITICAL：内存不足(后台进程不足3个)，并且该进程优先级比较高，需要清理内存 TRIM\_MEMORY\_RUNNING\_LOW：内存不足(后台进程不足5个)，并且该进程优先级比较高，需要清理内存 TRIM\_MEMORY\_RUNNING\_MODERATE：内存不足(后台进程超过5个)，并且该进程优先级比较高，需要清理内存

 以上3个是4.1增加

系统也提供了一个ComponentCallbacks2，通过Context.registerComponentCallbacks()注册后，就会被系统回调到。 OnLowMemory和OnTrimMemory的比较 1，OnLowMemory被回调时，已经没有后台进程；而onTrimMemory被回调时，还有后台进程。 2，OnLowMemory是在最后一个后台进程被杀时调用，一般情况是low memory killer 杀进程后触发；而OnTrimMemory的触发更频繁，每次计算进程优先级时，只要满足条件，都会触发。 3，通过一键清理后，OnLowMemory不会被触发，而OnTrimMemory会被触发一次。 9.图片缓存机制

目前很多商业应用都会涉及到从网络上读取图片数据的问题，为了节约用户流量，应用一般会将图片缓存起来。图片缓存一般分为内存缓存和外存缓存。内存缓存运用java的缓存机制，在程序完全退出后，缓存所在的内存空间可能被其它应用程序占用从而丢失。外存缓存一般放在程序特有的访问空间或者sd卡中，在sd卡中存放的资源为公有资源，其它程序也可以访问，且对用户来讲没有一个强制清除缓存的规范机制。综合以上，本文采用将缓存图片放置在程序的特有空间中， 其它应用程序无法访问，且用户可以在应用程序管理中的"清除数据"选项中清除缓存。 本文提供三种缓存策略：（1）LRU算法，固定缓存图片数量(max\_num)，当图片数量超出max\_num时，将缓存中最近用的最少的图片删除。（2）FTU算法，固定每张图片的缓存时限，以最后一次使用算起，超过时限后删除。（3）FMU算法，在存储器中固定一定大小的存储空间，超过固定空间后将缓存中占用最大尺寸的图片删除。使用时只需要向方法体中传递图片的URL即可。

代码片段

使用方法： 1.导入jar; 2. 获取服务； 3.提交url,交给程序去判断是否下载。

public class ImagecachetacticsdemoActivity extends Activity { public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.item); /*FMU*/ imageCacheManager = ImageCacheManager.getImageCacheService(this, ImageCacheManager.MODE\_FIXED\_MEMORY\_USED, "memory"); imageCacheManager.setMax\_Memory(1024 \* 1024); /*FTU*/ // imageCacheManager = ImageCacheManager.getImageCacheService(this, // ImageCacheManager.MODE\_FIXED\_TIMED\_USED, "time"); // imageCacheManager.setDelay\_millisecond(3 \* 60 \* 1000); /*LRU*/ // imageCacheManager = ImageCacheManager.getImageCacheService(this, // ImageCacheManager.MODE\_LEAST\_RECENTLY\_USED, "num"); // imageCacheManager.setMax\_num(5); // imageCacheManager = ImageCacheManager.getImageCacheService(this, // ImageCacheManager.MODE\_NO\_CACHE\_USED, "nocache"); mImageView = (ImageView) findViewById(R.id.imageView); new DownloadTask() .execute("http://www.touxiang99.com/uploads/allimg/110417/1\_110417112640\_2.jpg"); } private class DownloadTask extends AsyncTask { @Override protected Bitmap doInBackground(String... params) { try { return imageCacheManager.downlaodImage(new URL(params[0])); } catch (IOException e) { e.printStackTrace(); } return null; } @Override protected void onPostExecute(Bitmap result) { mImageView.setImageBitmap(result); super.onPostExecute(result); } @Override protected void onPreExecute() { mImageView.setImageResource(R.drawable.ic\_launcher); super.onPreExecute(); } } private ImageView mImageView; private ImageCacheManager imageCacheManager; }

10.Handler机制

11.什么是ANR 如何避免它？

ANR：Application NotResponding，五秒 　　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。当出现下列情况时，Android就会显示ANR对话框了： 　　对输入事件(如按键、触摸屏事件)的响应超过5秒 　　意向接受器(intentReceiver)超过10秒钟仍未执行完毕 　　Android应用程序完全运行在一个独立的线程中(例如main)。这就意味着，任何在主线程中运行的，需要消耗大量时间的操作都会引发ANR。因为此时，你的应用程序已经没有机会去响应输入事件和意向广播(Intentbroadcast)。 　　因此，任何运行在主线程中的方法，都要尽可能的只做少量的工作。特别是活动生命周期中的重要方法如onCreate()和 onResume()等更应如此。潜在的比较耗时的操作，如访问网络和数据库;或者是开销很大的计算，比如改变位图的大小，需要在一个单独的子线程中完成(或者是使用异步请求，如数据库操作)。但这并不意味着你的主线程需要进入阻塞状态已等待子线程结束 -- 也不需要调用Therad.wait()或者Thread.sleep()方法。取而代之的是，主线程为子线程提供一个句柄(Handler)，让子线程在即将结束的时候调用它(xing:可以参看Snake的例子，这种方法与以前我们所接触的有所不同)。使用这种方法涉及你的应用程序，能够保证你的程序对输入保持良好的响应，从而避免因为输入事件超过5秒钟不被处理而产生的ANR。这种实践需要应用到所有显示用户界面的线程，因为他们都面临着同样的超时问题。 12.sqlite操作使用

本帖最后由 y\_m 于 2013-6-2 00:27 编辑SQLite数据库 数据库:它就是一个软件，需要安装，安装完后就有自己的目录结构。 都有客户端和服务端，所有的数据库都实现了SQL标准 SQLite数据库:它是一个轻量级数据库，设计目的是嵌入式的，而且它占用的资源非常少 注意:除了主键不能存储任意的类型之外，其他的字段可以存放任意的数据类型 SqliteDatabase它要求:表的主键的字段名最好是\_id，一定要\_id 否则报错;

Cmd操作指令： sqlite3 qjq.db 进入数据库 .tables 查看数据库里面的表 创建数据库文件： 三种方式： 第一种通过上下文创建数据库： public class DBsqlite { private Context context; public DBsqlite(Context context) { super(); this.context = context; } public void createDB() { //通过上下文创建数据库 context.openOrCreateDatabase("persons.db", Context.MODE\_PRIVATE, null); } ｝ 第二种SQLiteDatabase创建数据库 public void createDB(){ String dir="/data/data/"+context.getPackageName(); File file=new File(dir,"persons.db"); SQLiteDatabase.openOrCreateDatabase(file, null); } 第三种创建一个help类继承SQLiteOpenHelper 实现DBhelp构造onCreate方法onUpgrade方法 public class DBhelp extends SQLiteOpenHelper { public DBhelp(Context context) { // 上下文 ，数据库名，游标工厂 ，数据版本 super(context, "persons.db", null, 2); // TODO Auto-generated constructor stub } //数据库第一次创建之后调用该方法。创建表、视图。。。 或者初始化表信息 public void onCreate(SQLiteDatabase db) { // 创建数据库 db.execSQL("create table fish(\_id integer primary key autoincrement,name text)"); } @Override //当数据版本被改变则会执行该方法super(context, "persons.db", null, 3)； public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) { // 版本修改表添加一列 db.execSQL("ALTER TABLE fish ADD amount integer"); } Crud 以下分别用两种方式crud了，一种是面向SQL ，一种是面向对象，但是面向对象的源码里面其实也是在帮你拼接sql。 public class OtherFishService { private SQLiteOpenHelper mOpenHelper; public OtherFishService(Context context) { // TODO Auto-generated constructor stub mOpenHelper = new DBHelper(context); } /\*\* \* 插入数据 \* @param name \*/ public void insert(String name){ SQLiteDatabase db = mOpenHelper.getWritableDatabase(); // String sql = "insert into fish(name) values(?)"; if(db.isOpen()){ // db.execSQL(sql, new Object[]{name}); //insert into fish //ContentValues里面就是要插入的值 ContentValues values = new ContentValues(); values.put("name", name); db.insert("fish", "\_id", values); db.close(); } } public List query(){ List fishs = new ArrayList(); SQLiteDatabase db = mOpenHelper.getReadableDatabase(); // String sql ="select \* from fish"; if(db.isOpen()){ //cursor 就是resultset // Cursor cursor = db.rawQuery(sql, null); Cursor cursor = db.query("fish",//表名 new String[]{"\*"},//要查询的列名 null,//查询条件 null,//条件参数 null,//分组 null,//条件 null);//排序 while(cursor.moveToNext()){ //得到\_id的下标 int column\_index = cursor.getColumnIndex("\_id"); //得到\_id的值 int \_id = cursor.getInt(column\_index); String name = cursor.getString(cursor.getColumnIndex("name")); Fish fish = new Fish(\_id, name); fishs.add(fish); } //cursor使用完成之后一定要关闭 cursor.close(); db.close(); } return fishs; } public void update(Fish fish){ SQLiteDatabase db = mOpenHelper.getWritableDatabase(); if(db.isOpen()){ // String sql = "update Fish set name = ? where \_id = ?"; // db.execSQL(sql,new Object[]{fish.name,fish.\_id}); ContentValues values = new ContentValues(); values.put("name",fish.name); String whereClause = " \_id = ?"; String[] whereArgs = new String[]{fish.\_id+""}; db.update("fish", values, whereClause, whereArgs); } } public void delete(int \_id){ SQLiteDatabase db = mOpenHelper.getWritableDatabase(); if(db.isOpen()){ // String sql = "delete from fish where \_id = ?"; // db.execSQL(sql,new Object[]{\_id}); String whereClause = " \_id = ?"; String[] whereArgs = new String[]{\_id+""}; db.delete("fish", whereClause, whereArgs); } } }

SqliteDatabase事务列表显示多条数据事务，以及事务的完整性SQLiteDatabase控制事务方法： beginTransaction(); 开启事务 setTransactionSuccessful(); 设置事务成功 endTransaction();结束事务 db.beginTransaction(); //开始事务 try { db.execSQL("update student set amount=amount-300 where name=?", new Object[]{"张三"}); db.execSQL("update student set amount=amount+300 where name=?", new Object[]{"李四"}); db.setTransactionSuccessful(); //设置事务成功

 } catch (Exception e) {
 System.out.println("事务执行出现异常，不能设置事务成功！");
 }finally{
 //无论是否出现异常，都要结束事务。
 db.endTransaction();
 }

cmd常见指令： Sqlite3 xx.db 查看xx数据库 .tables 查看所有表 .exit 退出 .databases 查看所有数据库 标签： include标签实现组件复用 将另一个布局文件加载到当前布局文件，实现组件复用 属性 android:visibility=有三个选项 列表显示多条数据 simpleAdapter 代码： OtherFishService ofs = new OtherFishService(this); List fishs = ofs.query(); List>data=new ArrayList>(); for(Fish f:fishs){ Map map = new HashMap(); map.put("\_id", f.\_id); map.put("name",f.name); map.put("age", f.age); data.add(map); } String[] from = new String[]{"\_id","name","age"}; int[] to = new int[]{R.id.tv\_id,R.id.tv\_name,R.id.tv\_age}; SimpleAdapter adapter = new SimpleAdapter(this, data, R.layout.item, from, to); lv.setAdapter(adapter); 以上可以实现但是过于麻烦！ 我们还可以使用专门为数据库提供的 Cursoradapter OtherFishService ofs = new OtherFishService(this); Cursor c = ofs.getAllCursor(); String[] from = new String[]{"\_id","name","age"}; int[] to = new int[]{R.id.tv\_id,R.id.tv\_name,R.id.tv\_age}; SimpleCursorAdapter adapter = new SimpleCursorAdapter( this, R.layout.item, c, from, to); lv.setAdapter(adapter); 我们不单单只用android给我们提过的CursorAdapter我们还可以自定义一个CursorAdapter 自定义CursorAdapter public class MyCursorAdapter extends CursorAdapter { //布局加载器（是一个服务，用来加载xml布局文件到java代码中） private LayoutInflater mInflater;

 public MyCursorAdapter(Context context, Cursor c) {
 super(context, c);
 mInflater=(LayoutInflater)context.getSystemService(Context.LAYOUT\_INFLATER\_SERVICE);

 mInflater = LayoutInflater.from(context);

 }
 //创建一个item的布局
 @Override
 public View newView(Context context, Cursor cursor, ViewGroup parent) {
 // TODO Auto-generated method stub
 //通过布局加载器加载布局
 View view = mInflater.inflate(R.layout.item, null);
 return view;
 }
 //把数据绑定给item里面的布局
 @Override
 public void bindView(View view, Context context, Cursor cursor) {
 // TODO Auto-generated method stub
 //1 先得到控件
 //2 得到数据
 //3 绑定数据给控件
 TextView tv\_id = (TextView) view.findViewById(R.id.tv\_id);
 TextView tv\_name = (TextView) view.findViewById(R.id.tv\_name);
 TextView tv\_age = (TextView) view.findViewById(R.id.tv\_age);

 String \_id = cursor.getString(0);
 String name = cursor.getString(1);
 String age = cursor.getString(2);

 tv\_id.setText(\_id);
 tv\_name.setText(name);
 tv\_age.setText(age);

 int position = cursor.getPosition();
 if(position%2 ==0){
 view.setBackgroundColor(Color.RED);
 }else{
 view.setBackgroundColor(Color.GREEN);
 }

 }

} 为了实现 我们想要的并且系统没提供的效果所以我们要自定义Adapter 不仅可以继承CursorAdapter 实现自定义CursorAdapter ，我们还可以通 过继承BaseAdapter 。

private class MyCursorAdapter extends BaseAdapter {
 private Context context;
 private Cursor c;
 private LayoutInflater mInflater;

 public MycursorAdapter1(Context context, Cursor c) {
 super();
 this.context = context;
 this.c = c;
 mInflater = (LayoutInflater) getSystemService(Context.LAYOUT\_INFLATER\_SERVICE);
 }
 @Override
 public int getCount() {
 // TODO Auto-generated method stub
 return c.getCount();
 }
 @Override
 public Object getItem(int position) {
 Person p = null;
 if (c.moveToPosition(position)) {
 String id = c.getString(0);
 String name = c.getString(1);
 String age = c.getString(2);
 p = new Person(id, name, age);
 }
 return p;
 }
 @Override
 public long getItemId(int position) {
 // TODO Auto-generated method stub
 return position;
 }
 @Override
 public View getView(int position, View convertView, ViewGroup parent) {
 //1得到视图
//2从视图中拿到要显示对象
//3从cursor里面拿值
//4绑定
//5返回

注意：getView方法实际会执行原本次数2倍（例：7条数据与，getView执行7\*2次），第一遍执行是为了测试有多少条数据，第2遍执行才是真正的现实操作

 View view = mInflater.inflate(R.layout.item, null);
 TextView tv\_id = (TextView) view.findViewById(R.id.bt\_id);
 TextView tv\_name = (TextView) view.findViewById(R.id.bt\_name);
 TextView tv\_age = (TextView) view.findViewById(R.id.bt\_age);
 if (c.moveToPosition(position)) {
 String id = c.getString(0);
 String name = c.getString(1);
 String age = c.getString(2);
 tv\_id.setText(id);
 tv\_name.setText(name);
 tv\_age.setText(age);

 }
 if ( position% 2 == 0) {
 view.setBackgroundColor(Color.RED);
 }
 return view;
 }
 }

当我们用完cursor的时候，怎么关闭他？ //应用回收资源 退出程序时调用该方法 @Override protected void onDestroy() { // TODO Auto-generated method stub super.onDestroy();r Cursor c = adapter.getCursor(); if(c != null && !c.isClosed()){ c.close(); } } 13请描述一下Intent 和 Intent Filter。

Android 中通过 Intent 对象来表示一条消息，一个 Intent 对象不仅包含有这个消息的目的地，还可以包含消息的内容，这好比一封Email，其中不仅应该包含收件地址，还可以包含具体的内容。对于一个 Intent 对象，消息“目的地”是必须的，而内容则是可选项。 通过Intent 可以实现各种系统组件的调用与激活.

Intent filter: 可以理解为邮局或者是一个信笺的分拣系统… 这个分拣系统通过3个参数来识别 Action: 动作 view Data: 数据uri uri Category : 而外的附加信息 Action 匹配 Action 是一个用户定义的字符串，用于描述一个 Android 应用程序组件，一个 Intent Filter 可以包含多个 Action。在 AndroidManifest.xml 的 Activity 定义时可以在其 节点指定一个 Action 列表用于标示 Activity 所能接受的“动作”，例如： ……

如果我们在启动一个 Activity 时使用这样的 Intent 对象： Intent intent =new Intent(); intent.setAction("cn.itcast.action"); startActivity(intent);

那么所有的 Action 列表中包含了“cn.itcast”的 Activity 都将会匹配成功。 Android 预定义了一系列的 Action 分别表示特定的系统动作。这些 Action 通过常量的方式定义在 android.content. Intent中，以“ACTION\_”开头。我们可以在 Android 提供的文档中找到它们的详细说明。 URI 数据匹配 一个 Intent 可以通过 URI 携带外部数据给目标组件。在 节点中，通过 节点匹配外部数据。 mimeType 属性指定携带外部数据的数据类型，scheme 指定协议，host、port、path 指定数据的位置、端口、和路径。如下： Intent intent = new Intent(); intent.setAction(Intent.ACTION\_CALL); insent.setData( Uri.parse(tel:12345)); startAcitivty(); 电话的uri tel: 12345 http://www.baidu.com 自己定义的uri itcast://cn.itcast/person/10

如果在 Intent Filter 中指定了这些属性，那么只有所有的属性都匹配成功时 URI 数据匹配才会成功。 Category 类别匹配 节点中可以为组件定义一个 Category 类别列表，当 Intent 中包含这个列表的所有项目时 Category 类别匹配才会成功。 默认是DEFAULT14.Intent传递数据时，可以传递哪些类型数据？

1.一般的基本数据类型 Intent .putextra() intent.getextra(); Parselable Serializable

 2.数据的uri, intent.setData() intent.getData();

15.Android事件传递机制

Android中dispatchTouchEvent,onInterceptTouchEvent, onTouchEvent的理解ec android中的事件类型分为按键事件和屏幕触摸事件，Touch事件是屏幕触摸事件的基础事件，有必要对它进行深入的了解。 一个最简单的屏幕触摸动作触发了一系列Touch事件：ACTION\_DOWN->ACTION\_MOVE->ACTION\_MOVE->ACTION\_MOVE...->ACTION\_MOVE->ACTION\_UP android的事件处理分为3步。 1）public booleandispatchTouchEvent(MotionEvent ev) 这个方法用来分发TouchEvent 2）public boolean onInterceptTouchEvent(MotionEvent ev) 这个方法用来拦截TouchEvent 3）public boolean onTouchEvent(MotionEvent ev) 这个方法用来处理TouchEvent 假设当前Activity 布局如下：

dispatchTouchEvent事件分发 当TouchEvent发生时，首先Activity将TouchEvent传递给最顶层的View， TouchEvent最先到达最顶层 view 的 dispatchTouchEvent 。然后由 dispatchTouchEvent 方法进行分发，如果dispatchTouchEvent返回true ，则交给这个view的onTouchEvent处理，如果dispatchTouchEvent返回 false ，则交给这个 view 的 onInterceptTouchEvent方法来决定是否要拦截这个事件， 如果onInterceptTouchEvent返回 true ，也就是拦截掉了，则交给它的 onTouchEvent 来处理，如果onInterceptTouchEvent返回 false ，那么就传递给子 view，由子 view 的 dispatchTouchEvent 再来开始这个事件的分发。 如图：

事件拦截：onInterceptTouchEvent onInterceptTouchEvent用于改变事件的传递方向。决定传递方向的是返回值，返回为false时事件会传递给子控件，返回值为true时事件会传递给当前控件的onTouchEvent()，这就是所谓的Intercept(拦截)。 [tisa ps:正确的使用方法是，在此方法内仅判断事件是否需要拦截，然后返回。即便需要拦截也应该直接返回true，然后由onTouchEvent方法进行处理。] onTouchEvent用于处理事件，返回值决定当前控件是否消费（consume）了这个事件。尤其对于ACTION\_DOWN事件，返回true，表示我想要处理后续事件（ACTION\_MOVE或者ACTION\_UP）；返回false，表示不关心此事件，并返回由父类进行处理。 在没有重写onInterceptTouchEvent()和onTouchEvent()的情况下(他们的返回值都是false), 对上面这个布局，MotionEvent事件的传递顺序如下：

当某个控件的onInterceptTouchEvent()返回值为true时，就会发生截断，事件被传到当前控件的onTouchEvent()。如我们将LayoutView2的onInterceptTouchEvent()返回值为true,则传递流程变成：

如果我们同时将LayoutView2的onInterceptTouchEvent()和onTouchEvent()设置成true，那么LayoutView2将消费被传递的事件，同时后续事件（如跟着ACTION\_DOWN的ACTION\_MOVE或者ACTION\_UP）会直接传给LayoutView2的onTouchEvent(),不传给其他任何控件的任何函数。同时传递给子空间一个ACTION\_CANCEL事件。传递流程变成（图中没有画出ACTION\_CANCEL事件）：

小总结：onInterceptTouchEvent是自rootiew向下传递, onTouchEvent正好相反。 16.请介绍下Android中常用的五种布局。

FrameLayout（帧布局），LinearLayout （线性布局），AbsoluteLayout（绝对布局），RelativeLayout（相对布局），TableLayout（表格布局） FrameLayout 从屏幕的左上角开始布局,叠加显示, 实际应用 播放器的暂停按钮.
LinearLayout 线性布局，这个东西，从外框上可以理解为一个div，他首先是一个一个从上往下罗列在屏幕上。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。 AbsoluteLayout 绝对布局犹如div指定了absolute属性，用X,Y坐标来指定元素的位置android:layout\_x="20px" view android:layout\_y="12px" fwvga 854*480apk qq斗地主 qq游戏大厅800*480 800*480.apk fwvga 854*480 指定平板机型的游戏开发中经常用到绝对布局 widget 绝对布局 指定机型的平板游戏开发. 2.3 3.0 1. 界面布局 任务管理器 gridview 2. 手机 任务管理 listview

lephone lepad 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="@id/aaa" 该元素在 id为aaa的下面 android:layout\_toLeftOf="@id/bbb" 改元素的左边是bbb 相对于父元素的地方 android:layout\_alignParentLeft="true" 在父元素左对齐 android:layout\_alignParentRight="true" 在父元素右对齐

Android oa客户端. TableLayout

表格布局类似Html里面的Table。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，设定他的对齐方式android:gravity="" 。 每一个布局都有自己适合的方式，另外，这五个布局元素可以相互嵌套应用，做出美观的界面。 oa 自动化 生成报表 ,图标 表示 webview
css div webview 17.谈谈UI中，Padding和Margin有什么区别？ Padding 文字对边框, margin是控件对父窗体. Padding 盒子里面的内容距离盒子的距离 , margin 盒子与盒子之间的距离 18.请解释下在单线程模型中Message、Handler、Message Queue、Looper之间的关系。

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。 19.AIDL的全称是什么？如何工作？

AIDL的英文全称是Android Interface Define Language当A进程要去调用B进程中的service时，并实现通信，我们通常都是通过AIDL来操作的A工程：首先我们在net.blogjava.mobile.aidlservice包中创建一个RemoteService.aidl文件，在里面我们自定义一个接口，含有方法get。ADT插件会在gen目录下自动生成一个RemoteService.java文件，该类中含有一个名为RemoteService.stub的内部类，该内部类中含有aidl文件接口的get方法。说明一：aidl文件的位置不固定，可以任意然后定义自己的MyService类，在MyService类中自定义一个内部类去继承RemoteService.stub这个内部类，实现get方法。在onBind方法中返回这个内部类的对象，系统会自动将这个对象封装成IBinder对象，传递给他的调用者。其次需要在AndroidManifest.xml文件中配置MyService类，代码如下： 为什么要指定调用AIDL服务的ID,就是要告诉外界MyService这个类能够被别的进程访问，只要别的进程知道这个ID，正是有了这个ID,B工程才能找到A工程实现通信。说明：AIDL并不需要权限B工程： 首先我们要将A工程中生成的RemoteService.java文件拷贝到B工程中，在bindService方法中绑定aidl服务 绑定AIDL服务就是将RemoteService的ID作为intent的action参数。 说明：如果我们单独将RemoteService.aidl文件放在一个包里，那个在我们将gen目录下的该包拷贝到B工程中。如果我们将RemoteService.aidl文件和我们的其他类存放在一起，那么我们在B工程中就要建立相应的包，以保证RmoteService.java文件的报名正确，我们不能修改RemoteService.java文件 bindService(new Inten("net.blogjava.mobile.aidlservice.RemoteService"), serviceConnection, Context.BIND\_AUTO\_CREATE); ServiceConnection的onServiceConnected(ComponentName name, IBinder service)方法中的service参数就是A工程中MyService类中继承了RemoteService.stub类的内部类的对象。

20.请解释下Android程序运行时权限与文件系统权限的区别。

Android程序执行需要读取到安全敏感项必需在androidmanifest.xml中声明相关权限请求, 打电话,访问网络,获取坐标,读写sd卡,读写联系人等..安装的时候会提示用户…

文件系统的权限是linux权限. 比如说sharedpreference里面的Context.Mode.private Context.Mode.world\_read\_able Context.Mode\_world\_writeable 777自己 同组 其他21.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

Intent intent =newIntent();
intent.setAction("android.intent.action.VIEW");
Uri content\_url =Uri.parse("http://www.163.com");
intent.setData(content\_url);
intent.setClassName("com.android.browser","com.android.browser.BrowserActivity");
startActivity(intent); 只要修改以intent.setClassName("com.android.browser","com.android.browser.BrowserActivity"); 中相应的应用程序packagename 和要启动的activity即可启动其他浏览器来 uc浏览器"："com.uc.browser", "com.uc.browser.ActivityUpdate“ opera浏览器："com.opera.mini.android", "com.opera.mini.android.Browser" qq浏览器："com.tencent.mtt", "com.tencent.mtt.MainActivity"22.对android主线程的运用和理解。

主ui线程不能执行耗时的操作23.对android虚拟机的理解，包括内存管理机制垃圾回收机制。dalvik和art区别

虚拟机很小,空间很小,谈谈移动设备的虚拟机的大小限制 16M , 谈谈加载图片的时候怎么处理大图片的, outmemoryException BitmapFactory.option 垃圾回收,没有引用的对象,在某个时刻会被系统gc掉 .

Dalvik和标准Java虚拟机(JVM)首要差别 Dalvik 基于寄存器，而 JVM 基于栈。 基于寄存器的虚拟机对于编译后变大的程序来说，在它们执行的时候，花费的时间更短。 Dalvik和Java运行环境的区别 1：Dalvik主要是完成对象生命周期管理，堆栈管理，线程管理，安全和异常管理，以及垃圾回收等等重要功能。 2：Dalvik负责进程隔离和线程管理，每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行。 3：不同于Java虚拟机运行java字节码，Dalvik虚拟机运行的是其专有的文件格式Dex 4:dex文件格式可以减少整体文件尺寸，提高I/o操作的类查找速度。 5:odex是为了在运行过程中进一步提高性能，对dex文件的进一步优化。 6：所有的Android应用的线程都对应一个Linux线程，虚拟机因而可以更多的依赖操作系统的线程调度和管理机制 7：有一个特殊的虚拟机进程Zygote，他是虚拟机实例的孵化器。它在系统启动的时候就会产生，它会完成虚拟机的初始化，库的加载，预制类库和初始化的操作。如果系统需要一个新的虚拟机实例，它会迅速复制自身，以最快的数据提供给系统。对于一些只读的系统库，所有虚拟机实例都和Zygote共享一块内存区域。 8：Dalvik是由Dan Bornstein编写的，名字来源于他的祖先曾经居住过名叫Dalvík的小渔村，村子位于冰岛Eyjafjörður

许多GC实现都是在对象开头的地方留一小块空间给GC标记用。Dalvik VM则不同，在进行GC的时候会单独申请一块空间，以位图的形式来保存整个堆上的对象的标记，在GC结束后就释放该空间。 dalvik是执行的时候编译+运行，安装比较快，开启应用比较慢，应用占用空间小ART是安装的时候就编译好了，执行的时候直接就可以运行的，安装慢，开启应用快，占用空间大用个比喻来说就是，骑自行车dalvik 是已经折叠起来的自行车，每次骑都要先组装自行车才能骑ART 是已经组装好的自行车，每次骑直接上车就能走人效率高在开启的时候，运行中的速度是差不多的24.Framework工作方式及原理，Activity是如何生成一个view的，机制是什么。

主要用到反射和代理的知识。主要用pull解析xml，然后通过反射得到属性，在onDrew画出来

反射 , 配置文件 可以讲下activity的源码,比如说 每个activity里面都有window.callback和keyevent.callback,一些回调的接口或者函数吧. 框架把activity创建出来就会调用里面的这些回调方法,会调用activity生命周期相关的方法.

Activity创建一个view是通过ondraw 画出来的, 画这个view之前呢,还会调用onmeasure方法来计算显示的大小.25.android本身的一些限制，比如apk包大小限制，读取大文件时的时间限

这个问题问的有问题, apk包大小限制不好说, 极品飞车有100M 还是能装到手机上, 世面google market 上大程序 主程序 很小 5~10M 下载sdcard 200~300M 15分钟之内 申请退款 apk包,精简包, 素材存放在服务器. 游戏程序.

读大文件的时间限制应该是main线程里面的时间限制吧.5秒.26.Android程序与Java程序的区别？

Android程序用android sdk开发,java程序用javasdk开发. Android SDK引用了大部分的Java SDK，少数部分被Android SDK抛弃，比如说界面部分，java.awt swing package除了java.awt.font被引用外，其他都被抛弃，在Android平台开发中不能使用。 android sdk 添加工具jar httpclient , pull opengl

将Java 游戏或者j2me程序移植到Android平台的过程中， Android SDK 与Java SDK的区别是很需要注意的地方。 sampleDataAdpter("YY-MM-DD")27.Android中Task任务栈的分配

首先我们来看下Task的定义，Google是这样定义Task的：a task is what the user experiences as an "application." It's a group of related activities, arranged in a stack. A task is a stack of activities, not a class or an element in the manifest file. 这意思就是说Task实际上是一个Activity栈，通常用户感受的一个Application就是一个Task。从这个定义来看，Task跟Service或者其他Components是没有任何联系的，它只是针对Activity而言的。 Activity有不同的启动模式, 可以影响到task的分配 Task，简单的说，就是一组以栈的模式聚集在一起的Activity组件集合。它们有潜在的前后驱关联，新加入的Activity组件，位于栈顶，并仅有在栈顶的Activity，才会有机会与用户进行交互。而当栈顶的Activity完成使命退出的时候，Task会将其退栈，并让下一个将跑到栈顶的Activity来于用户面对面，直至栈中再无更多Activity，Task结束。 事件 Task栈（粗体为栈顶组件） 点开Email应用，进入收件箱（Activity A） A 选中一封邮件，点击查看详情（Activity B） AB 点击回复，开始写新邮件（Activity C） ABC 写了几行字，点击选择联系人，进入选择联系人界面（Activity D） ABCD 选择好了联系人，继续写邮件 ABC 写好邮件，发送完成，回到原始邮件 AB 点击返回，回到收件箱 A 退出Email程序 null 如上表所示，是一个实例。从用户从进入邮箱开始，到回复完成，退出应用整个过程的Task栈变化。这是一个标准的栈模式，对于大部分的状况，这样的Task模型，足以应付，但是，涉及到实际的性能、开销等问题，就会变得残酷许多。

比如，启动一个浏览器，在Android中是一个比较沉重的过程，它需要做很多初始化的工作，并且会有不小的内存开销。但与此同时，用浏览器打开一些内容，又是一般应用都会有的一个需求。设想一下，如果同时有十个运行着的应用（就会对应着是多个Task），都需要启动浏览器，这将是一个多么残酷的场面，十个Task栈都堆积着很雷同的浏览器Activity， 是多么华丽的一种浪费啊。 于是你会有这样一种设想，浏览器Activity，可不可以作为一个单独的Task而存在，不管是来自那个Task的请求，浏览器的Task，都不会归并过去。这样，虽然浏览器Activity本身需要维系的状态更多了，但整体的开销将大大的减少，这种舍小家为大家的行为，还是很值得歌颂的 standard", "singleTop", "singleTask", "singleInstance"。

standard模式， 是默认的也是标准的Task模式，在没有其他因素的影响下，使用此模式的Activity，会构造一个Activity的实例，加入到调用者的Task栈中去，对于使用频度一般开销一般什么都一般的Activity而言，standard模式无疑是最合适的，因为它逻辑简单条理清晰，所以是默认的选择。

而singleTop模式，基本上于standard一致，仅在请求的Activity正好位于栈顶时，有所区别。此时，配置成singleTop的Activity，不再会构造新的实例加入到Task栈中，而是将新来的Intent发送到栈顶Activity中，栈顶的Activity可以通过重载onNewIntent来处理新的Intent（当然，也可以无视...）。这个模式，降低了位于栈顶时的一些重复开销，更避免了一些奇异的行为（想象一下，如果在栈顶连续几个都是同样的Activity，再一级级退出的时候，这是怎么样的用户体验...），很适合一些会有更新的列表Activity展示。一个活生生的实例是，在Android默认提供的应用中，浏览器（Browser）的书签Activity（BrowserBookmarkPage），就用的是singleTop。

singleTask，和singleInstance，则都采取的另辟Task的蹊径。 标志为singleTask的Activity，最多仅有一个实例存在，并且，位于以它为根的Task中。所有对该Activity的请求，都会跳到该Activity的Task中展开进行。singleTask，很象概念中的单件模式，所有的修改都是基于一个实例，这通常用在构造成本很大，但切换成本较小的Activity中。最典型的例子，还是浏览器应用的主Activity（名为Browser...），它是展示当前tab，当前页面内容的窗口。它的构造成本大，但页面的切换还是较快的，于singleTask相配，还是挺天作之合的。

singleInstance显得更为极端一些。在大部分时候singleInstance与singleTask完全一致，唯一的不同在于，singleInstance的Activity，是它所在栈中仅有的一个Activity，如果涉及到的其他Activity，都移交到其他Task中进行。这使得singleInstance的Activity，像一座孤岛，彻底的黑盒，它不关注请求来自何方，也不计较后续由谁执行。在Android默认的各个应用中，很少有这样的Activity，在我个人的工程实践中，曾尝试在有道词典的快速取词Activity中采用过， 是因为我觉得快速取词入口足够方便（从notification中点选进入），并且会在各个场合使用，应该做得完全独立。

大的apk 拆成 很多小的apk ●Activity的android:affinity属性 1.配置后 当启动这个activity时就先去找有没有activity的亲和力属性相同 有就加入这个 activity所在的任务中没有就新开任务 2.affinity起作用需要的条件而者具备一个: 1.intent包含FLAG\_ACTIVITY\_NEW\_TASK标记 2.activity元素启用了allowTaskReparenting属性.28.在Android中，怎么节省内存的使用，怎么主动回收内存？

回收已经使用的资源, 合理的使用缓存 合理设置变量的作用范围… application 对象 //未来的某一段时间执行 System.gc();29.不同工程中的方法是否可以相互调用？

可以,列举aidl访问远程服务的例子.30.dvm的进程和Linux的进程, 应用程序的进程是否为同一个概念 是同一概念

Dvm的进程是dalivk虚拟机进程,每个android程序都运行在自己的进程里面, 每个android程序系统都会给他分配一个单独的liunx uid(user id), 每个dvm都是linux里面的一个进程.所以说这两个进程是一个进程.31.在Android中是如何实现判断区分电话的状态，去电，来电、未接来电？

Day8 showAddressService.java TelephoneyManger.listen();32.谈谈Android的优点和不足之处。

1、开放性,开源ophone 阿里云( 完全兼容android) 2、挣脱运营商束缚 3、丰富的硬件选择 mtk android 4、不受任何限制的开发商 5、无缝结合的Google应用

缺点也有5处： 1、安全问题、隐私问题 2、卖手机的不是最大运营商 3、运营商对Android手机仍然有影响 4、山寨化严重 5、过分依赖开发商，缺乏标准配置33.Android系统中GC什么情况下会出现内存泄露呢?

视频编解码/内存泄露 检测内存泄露 工具

导致内存泄漏主要的原因是，先前申请了内存空间而忘记了释放。如果程序中存在对无用对象的引用，那么这些对象就会驻留内存，消耗内存，因为无法让垃圾回收器GC验证这些对象是否不再需要。如果存在对象的引用，这个对象就被定义为"有效的活动"，同时不会被释放。要确定对象所占内存将被回收，我们就要务必确认该对象不再会被使用。典型的做法就是把对象数据成员设为null或者从集合中移除该对象。但当局部变量不需要时，不需明显的设为null，因为一个方法执行完毕时，这些引用会自动被清理。 Java带垃圾回收的机制,为什么还会内存泄露呢?

Vector v = new Vector(10);
for (int i = 1; i < 100; i++) {
　Object o = new Object(); 　 v.add(o); 　 o = null;
}//此时，所有的Object对象都没有被释放，因为变量v引用这些对象。

Java 内存泄露的根本原因就是 保存了不可能再被访问的变量类型的引用34.Android UI中的View如何刷新。

在主线程中 拿到view调用Invalide()方法,查看画画板里面更新imageview的方法

在子线程里面可以通过postInvalide()方法;35. 简单描述下Android 数字签名。

Android 数字签名 在Android系统中，所有安装到系统的应用程序都必有一个数字证书，此数字证书用于标识应用程序的作者和在应用程序之间建立信任关系 Android系统要求每一个安装进系统的应用程序都是经过数字证书签名的，数字证书的私钥则保存在程序开发者的手中。Android将数字证书用来标识应用程序的作者和在应用程序之间建立信任关系，不是用来决定最终用户可以安装哪些应用程序。 这个数字证书并不需要权威的数字证书签名机构认证(CA)，它只是用来让应用程序包自我认证的。 同一个开发者的多个程序尽可能使用同一个数字证书，这可以带来以下好处。 (1)有利于程序升级，当新版程序和旧版程序的数字证书相同时，Android系统才会认为这两个程序是同一个程序的不同版本。如果新版程序和旧版程序的数字证书不相同，则Android系统认为他们是不同的程序，并产生冲突，会要求新程序更改包名。

(2)有利于程序的模块化设计和开发。Android系统允许拥有同一个数字签名的程序运行在一个进程中，Android程序会将他们视为同一个程序。所以开发者可以将自己的程序分模块开发，而用户只需要在需要的时候下载适当的模块。 在签名时，需要考虑数字证书的有效期： (1)数字证书的有效期要包含程序的预计生命周期，一旦数字证书失效，持有改数字证书的程序将不能正常升级。 (2)如果多个程序使用同一个数字证书，则该数字证书的有效期要包含所有程序的预计生命周期。 (3)Android Market强制要求所有应用程序数字证书的有效期要持续到2033年10月22日以后。 Android数字证书包含以下几个要点： (1)所有的应用程序都必须有数字证书，Android系统不会安装一个没有数字证书的应用程序 (2)Android程序包使用的数字证书可以是自签名的，不需要一个权威的数字证书机构签名认证 (3)如果要正式发布一个Android ，必须使用一个合适的私钥生成的数字证书来给程序签名，而不能使用adt插件或者ant工具生成的调试证书来发布。 (4)数字证书都是有有效期的，Android只是在应用程序安装的时候才会检查证书的有效期。如果程序已经安装在系统中，即使证书过期也不会影响程序的正常功能。36.android中的动画有哪几类，它们的特点和区别是什么？

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； 可以通过布局文件,可以通过代码 1、 控制View的动画 a) alpha(AlphaAnimation) 渐变透明
b) scale(ScaleAnimation) 渐变尺寸伸缩 c) translate(TranslateAnimation) 画面转换、位置移动
d) rotate(RotateAnimation) 画面转移，旋转动画

2、 控制一个Layout里面子View的动画效果 a) layoutAnimation(LayoutAnimationController) b) gridAnimation(GridLayoutAnimationController) 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。37.说说mvc模式的原理，它在android中的运用

Android中没有mvc，只有在j2ee中有，是用于逻辑也界面分离的 MVC英文即Model-View-Controller，即把一个应用的输入、处理、输出流程按照Model、View、Controller的方式进行分离，这样一个应用被分成三个层——模型层、视图层、控制层。

Android中界面部分也采用了当前比较流行的MVC框架，在Android中M就是应用程序中二进制的数据，V就是用户的界面。Android的界面直接采用XML文件保存的，界面开发变的很方便。在Android中C也是很简单的，一个Activity可以有多个界面，只需要将视图的ID传递到setContentView()，就指定了以哪个视图模型显示数据。

在Android SDK中的数据绑定，也都是采用了与MVC框架类似的方法来显示数据。在控制层上将数据按照视图模型的要求（也就是Android SDK中的Adapter）封装就可以直接在视图模型上显示了，从而实现了数据绑定。比如显示Cursor中所有数据的ListActivity，其视图层就是一个ListView，将数据封装为ListAdapter，并传递给ListView，数据就在ListView中显示。38.通过点击一个网页上的url 就可以完成程序的自动安装,描述下原理

Day11 AddJavascriptInterface new Object{ callphone(); installapk(); }

将js与java相互调用再来一个例子，解决相互调用之间的关系。

首先说明一重要代码的情况：

android中的关键代码： webview.getSettings().setJavaScriptEnabled(true); webview.addJavascriptInterface(object,"name"); //把Name="name"的对象添加到object中。object如果是this，就是window.name webview.loadUrl("file:///android\_asset/index.html"); //注意这个资源的位置放在assets文件夹下。

js或html中调用android中方法代码： js中使用 window.name.java中的方法()； android中调用js的function方法： Callfunction(){ webview.loadUrl("javascript: function ()"); 需要注意的地方，很多数据类型js中不认识，最好是在android那边封装好，提供必要的方法接口。比如传到js中的list，在js中是没办法去得到里面的元素的。 addJavascriptInterface：addJavascriptInterface方法中要绑定的Java对象及方法要运行在另外的线程中，不能运行在构造他的线程中，也就是说不能运行在当前的activity线程中，就是把这个方法绑定到页面中，js也可以调用。 文档中的解释： Use this function to bind an object to Javascript so that the methods can be accessed from Javascript. The Java object that is bound runs in another thread and not in the thread that it was constructed in.

下面给出具体的测试代码： 1、Activity 代码

public class TestWebView extends Activity {
 private WebView mWebView;
 private List<String> list;
 private int mkeyCode;

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 mWebView = (WebView) findViewById(R.id.htmlview);

 initData();

 WebSettings webSettings = mWebView.getSettings();

 // 是否允许在webview中执行javascript
 webSettings.setJavaScriptEnabled(true);

 mWebView.addJavascriptInterface(this, "javatojs");

 //加载网页
 mWebView.loadUrl("file:///android\_asset/index.html");
 }

 @Override
 public boolean onKeyUp(int keyCode, KeyEvent event) {
 mkeyCode = keyCode;
 Log.i("AA","keyCode="+keyCode);
 mWebView.loadUrl("javascript: OnKeyUp()");
 return super.onKeyUp(keyCode, event);
 }

 public int getKeyCode(){
 return mkeyCode;
 }

 void initData() {
 list = new ArrayList<String>();
 for (int i = 0; i < 5; i++) {
 list.add("我是List中的第" + (i + 1) + "行");
 }
 }

 /\*\*
 \* 该方法将在js脚本中，通过window.javatojs.....()进行调用
 \*
 \* @return
 \*/
 public Object getObject(int index) {
 Log.i("A","getObject");
 return list.get(index);
 }

 public int getSize() {
 Log.i("A","getSize");
 return list.size();
 }

 public void Callfunction() {
 Log.i("A","Callfunction");
 mWebView.loadUrl("javascript: GetList()");
 }

 public void printStr(String str){
 Log.i("A","GetList:" + str);
 }
}

2、js 代码 index.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>demotitle></title>

<script language="javascript">

function GetList(){
 var i=window.javatojs.getSize();

 for(var n=0;n<i;n++){
 var jsdata= window.javatojs.getObject(n);//拿到activity里面的属性javadata
 window.javatojs.printStr("test");
 }
}

function OnKeyUp() {
 var keycode = window.javatojs.getKeyCode();
 window.javatojs.printStr(keycode);
}

</script>
</head>

 <body style="background-color:#000;">
 <table width="400" align="center"><tr><td>
 <p> </p>
 <p>
<table width="400" align="center">
<div id="output" >test</div>
 <input type="submit" value="buttons"
 onclick="document.getElementById('output').innerHTML=window.javatojs.Callfunction()"/>
 <br>

 </p>
 </td></tr></table>
</body>

</html>

以上代码主要测试js与java相互调用，而由于按键这种系统事件被webview截获掉，有如下两种方式进行处理

1、把方向键的流程改成：先传给webcore，假如没处理，再在webview里面处理，这个需要修改webview.java代码 2、直接应用搞定，java捕获按键，然后调js函数，上面代码就是使用这种方法。

测试结果如下： 点击buttons按钮： I/A ( 4990): Callfunction I/A ( 4990): getSize I/A ( 4990): getObject I/A ( 4990): GetList:test I/A ( 4990): getObject I/A ( 4990): GetList:test I/A ( 4990): getObject I/A ( 4990): GetList:test I/A ( 4990): getObject I/A ( 4990): GetList:test I/A ( 4990): getObject I/A ( 4990): GetList:test

you press KEY\_RIGHT I/AA ( 4990): keyCode=22 I/A ( 4990): GetList:22

you press KEY\_UP I/AA ( 4990): keyCode=19 I/A ( 4990): GetList:19

you press KEY\_DOWN I/AA ( 4990): keyCode=20 I/A ( 4990): GetList:20

you press KEY\_LEFT I/AA ( 4990): keyCode=21 I/A ( 4990): GetList:21

这里为何使用这种方式，是因为对于上下左右及确定这种功能键被webview截取掉了，无法传递到webcore中，而只能重载OnKeyDown/OnKeyUp方法，再由js调用java方法来获取得。 对于数字键的处理可以直接在js中进行处理： logcat中会有明显的打印，对于这些键没有截掉，所以可以直接获取得到： D/webcore ( 4990): proc key: code=12 D/webcore ( 4990): proc key: nativeKey return false D/webcore ( 4990): proc key: nativeKey return true js代码可以如此编写：

 <script language="JavaScript">

document.onkeypress = grab\_keypress\_event;
document.onirkeypress = grab\_irkeypress\_event;
document.onsystemevent = grab\_system\_event;
document.onkeydown = grab\_keydown\_event;
document.onkeyup = grab\_keyup\_event;

function init()
{
 document.getElementById("txt\_keypress").innerHTML = "";
 document.getElementById("txt\_irkey").innerHTML= "";
 document.getElementById("txt\_systemevent").innerHTML="";
 document.getElementById("txt\_keydown").innerHTML="";
 document.getElementById("txt\_keyup").innerHTML="";
}

function grab\_keypress\_event(event)
{
 var keycode = event.keyCode;
 document.getElementById("txt\_keypress").innerHTML=keycode;
}

function grab\_irkeypress\_event(event)
{
 var keycode = event.keyCode;
 document.getElementById("txt\_irkey").innerHTML=keycode;
}

function grab\_system\_event(event)
{
 var keycode = event.which;
 document.getElementById("txt\_systemevent").innerHTML=keycode;
}

function grab\_keydown\_event(event)
{
 var keycode = event.keyCode;
 var type = event.type;
 var mod = event.modifiers;
 document.getElementById("txt\_keydown").innerHTML=keycode;
}

function grab\_keyup\_event(event)
{
 var keycode = event.keyCode;
 document.getElementById("txt\_keyup").innerHTML=keycode;
}
 </script>

39.Service和Activity在同一个线程吗

默认情况同一线程 main主线程 ui线程40.ViewStub的应用

在开发应用程序的时候，经常会遇到这样的情况，会在运行时动态根据条件来决定显示哪个View或某个布局。那么最通常的想法就是把可能用到的View都写在上面，先把它们的可见性都设为View.GONE，然后在代码中动态的更改它的可见性。这样的做法的优点是逻辑简单而且控制起来比较灵活。但是它的缺点就是，耗费资源。虽然把View的初始可见View.GONE但是在Inflate布局的时候View仍然会被Inflate，也就是说仍然会创建对象，会被实例化，会被设置属性。也就是说，会耗费内存等资源。 推荐的做法是使用android.view.ViewStub，ViewStub是一个轻量级的View，它一个看不见的，不占布局位置，占用资源非常小的控件。可以为ViewStub指定一个布局，在Inflate布局的时候，只有ViewStub会被初始化，然后当ViewStub被设置为可见的时候，或是调用了ViewStub.inflate()的时候，ViewStub所向的布局就会被Inflate和实例化，然后ViewStub的布局属性都会传给它所指向的布局。这样，就可以使用ViewStub来方便的在运行时，要还是不要显示某个布局。 但ViewStub也不是万能的，下面总结下ViewStub能做的事儿和什么时候该用ViewStub，什么时候该用可见性的控制。 首先来说说ViewStub的一些特点： 1. ViewStub只能Inflate一次，之后ViewStub对象会被置为空。按句话说，某个被ViewStub指定的布局被Inflate后，就不会够再通过ViewStub来控制它了。 2. ViewStub只能用来Inflate一个布局文件，而不是某个具体的View，当然也可以把View写在某个布局文件中。 基于以上的特点，那么可以考虑使用ViewStub的情况有： 1. 在程序的运行期间，某个布局在Inflate后，就不会有变化，除非重新启动。 因为ViewStub只能Inflate一次，之后会被置空，所以无法指望后面接着使用ViewStub来控制布局。所以当需要在运行时不止一次的显示和隐藏某个布局，那么ViewStub是做不到的。这时就只能使用View的可见性来控制了。 2. 想要控制显示与隐藏的是一个布局文件，而非某个View。 因为设置给ViewStub的只能是某个布局文件的Id，所以无法让它来控制某个View。 所以，如果想要控制某个View(如Button或TextView)的显示与隐藏，或者想要在运行时不断的显示与隐藏某个布局或View，只能使用View的可见性来控制。 下面来看一个实例 在这个例子中，要显示二种不同的布局，一个是用TextView显示一段文字，另一个则是用ImageView显示一个图片。这二个是在onCreate()时决定是显示哪一个，这里就是应用ViewStub的最佳地点。 先来看看布局，一个是主布局，里面只定义二个ViewStub，一个用来控制TextView一个用来控制ImageView，另外就是一个是为显示文字的做的TextView布局，一个是为ImageView而做的布局：

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout\_width="fill\_parent"
 android:layout\_height="fill\_parent"
 android:gravity="center\_horizontal" >
 <ViewStub
 android:id="@+id/viewstub\_demo\_text"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content"
 android:layout\_marginLeft="5dip"
 android:layout\_marginRight="5dip"
 android:layout\_marginTop="10dip"
 android:layout="@layout/viewstub\_demo\_text\_layout"/>
 <ViewStub
 android:id="@+id/viewstub\_demo\_image"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content"
 android:layout\_marginLeft="5dip"
 android:layout\_marginRight="5dip"
 android:layout="@layout/viewstub\_demo\_image\_layout"/>
</LinearLayout >

为TextView的布局：

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content" >
 <TextView
 android:id="@+id/viewstub\_demo\_textview"
 android:layout\_width="fill\_parent"
 android:layout\_height="wrap\_content"
 android:background="#aa664411"
 android:textSize="16sp"/>
</LinearLayout >

为ImageView的布局：

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content" >
 <ImageView
 android:id="@+id/viewstub\_demo\_imageview"
 android:layout\_width="wrap\_content"
 android:layout\_height="wrap\_content"/>
</LinearLayout >

下面来看代码，决定来显示哪一个，只需要找到相应的ViewStub然后调用其infalte()就可以获得相应想要的布局：

package com.effective;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewStub;
import android.widget.ImageView;
import android.widget.TextView;

public class ViewStubDemoActivity extends Activity {
 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.viewstub\_demo\_activity);
 if ((((int) (Math.random() \* 100)) & 0x01) == 0) {
 // to show text
 // all you have to do is inflate the ViewStub for textview
 ViewStub stub = (ViewStub) findViewById(R.id.viewstub\_demo\_text);
 stub.inflate();
 TextView text = (TextView) findViewById(R.id.viewstub\_demo\_textview);
 text.setText("The tree of liberty must be refreshed from time to time" +
 " with the blood of patroits and tyrants! Freedom is nothing but " +
 "a chance to be better!");
 } else {
 // to show image
 // all you have to do is inflate the ViewStub for imageview
 ViewStub stub = (ViewStub) findViewById(R.id.viewstub\_demo\_image);
 stub.inflate();
 ImageView image = (ImageView) findViewById(R.id.viewstub\_demo\_imageview);
 image.setImageResource(R.drawable.happy\_running\_dog);
 }
 }
}

运行结果：

使用的时候的注意事项： 1. 某些布局属性要加在ViewStub而不是实际的布局上面，才会起作用，比如上面用的android:layout\_margin\*系列属性，如果加在TextView上面，则不会起作用，需要放在它的ViewStub上面才会起作用。而ViewStub的属性在inflate()后会都传给相应的布局。 41.android开发中怎么去调试bug

逻辑错误 1.断点 debug 2. logcat , 界面布局,显示 hierarchyviewer.bat42.书写出android工程的目录结构以及相关作用

下面是HelloAndroid项目在eclipse中的目录层次结构：

由上图可以看出项目的根目录下共有九个文件（夹），下面就这九个文件（夹）进行详解： 1.1src文件夹和assets文件夹： 每个Android程序都包含资源目录（src）和资产目录（assets），资源和资产听起来感觉没有多大差别，但在存储外部内容时用资源（src）比较多，其中它们的区别在于存放在资源（src）下的内容可以通过应用程序的R类进行访问，而存放在资产（assets）下的内容会保持原始文件的格式，如果需要访问，则必须使用AssetManager以字节流的方式来读取，用起来非常的不方便。为了方便使用，通常文件和数据都会保存在资源（src）目录下 1.2res(Resource)目录：资源目录 可以存放一些图标，界面文件和应用中用到的文字信息，下图为res目录截图: 1.2.1 drawable-\*dpi文件夹:将图标按分辨率的高低放入不同的目录，其中draeable-hdpi用来存放高分辨率的图标，drawable-mdpi用来存放中等分辨率的图标，drawable-ldpi用来存放低分辨率的图标 1.2.2 values文件夹：用来存放文字的信息 （1）strings.xml:用来定义字符串和数值

<string name="hello">Hello World, Hello 3G</string>
<string name="app\_name">Android1.1</string>
 <string name="test">哥想你了</string>
 <string name="startButton">按钮1</string>
 <string name="start">按钮1</string>

 每个string标签生命了一个字符串，name属性指定它的引用值 （2）为什么要把这些出现的文字单独放在strings.xml文件中？ 答案：一是为了国际化，如果需要将文件中的文字换成别的国家的语言，就可以只需要替换掉一个strings.xml文件就可以了 二是为了减少应用的体积，例如，我们要在应用中使用“哥想你了”这句话1000次，如果我们没有将“哥想你了”定义在strings.xml文件中，而是直接在应用中使用时写上这几个字，那么我们就会在应用中写4000个字。4000个字和4个字占用的内存可是有很大差距的啊，况且手机的内存本来就小，所以应该是能省就省 （3）另外还有arrays.xml,color.xml等定义数组，颜色的，都最好用单独的一个xml文档 1.2.3 layout文件：用来存放界面信息 本例中的布局文件是自动生成的“main.xml”

<TextView
 android:layout\_width="fill\_parent"
 android:layout\_height="wrap\_content"
 android:text="@string/test"/>

 元素：线性布局的意思，在该元素下的所有子元素都会根据他的”orientation”属性来决定是按行还是按列或者按逐个显示进行布局的 元素：是一种显示控件，他的”text”属性指定了在这个元素上显示的内容 1.3 gen目录：gen目录下只有一个自动生成的“R.java”文件

/\*AUTO-GENERATED FILE. DO NOT MODIFY.
 \*
 \* This class was automatically generated bythe
 \* aapt tool from the resource data itfound. It
 \* should not be modified by hand.
 \*/

packagecn.csdn.android.demo;

public final class R {
 public static final class attr {
 }
 public static final class drawable {
 public static final int ic\_launcher=0x7f020000;
 }
 public static final class id {
 public static final int button1=0x7f050000;
 public static final int radioButton1=0x7f050001;
 public static final int toggleButton1=0x7f050002;
 }
 public static final class layout {
 public static final int main=0x7f030000;
 }
 public static final class string {
 public static final int app\_name=0x7f040001;
 public static final int hello=0x7f040000;
 public static final int start=0x7f040004;
 public static final int startButton=0x7f040003;
 public static final int test=0x7f040002;
 }
}

R.java文件：默认有attr,drawable,layout,string这四个静态内部类，每个静态内部类对应一中资源，如layout静态内部类对应layout中的界面文件，string静态内部类对应string内部的string标签。如果在layout中在增加一个界面文件或者在string内增加一个string标签，R.java会自动在其对应的内部类增加所增加的内容。 R.java除了自动标识资源的索引功能外，还有另一个功能，就是当res文件中的某个资源在应用中没有被用到，在这个应用被编译时，系统不会把对应的资源编译到应用中的APR包中。 1.4 AndroidManifest.xml 功能清单文件 每个应用程序都会有一个AndroidManifest在它的根目录里面。这个清单为Android系统提供了这个应用的基本信息，系统在运行之前必须知道这些信息，另外，如果我们使用系统自带的服务，如拨号服务，应用安装服务等，都必须在AndroidManifest.xml文件中声明权限 AndroidManifest.xml的功能： 命名应用程序的Java应用包，这个包名用来唯一标识应用程序； 描述应用程序的组件，对实现每个组件和公布其功能的类进行命名，这些声明使得Android系统了解这些组件以及它们在什么条件下可以被启动 决定哪个组件运行在哪个进程里面 声明应用程序必须具备的权限，用以访问受保护的API，以及和其他进程的交互 声明应用程序其他的必备权限，用以组件之间的交互 列举application所需要链接的库 以HelloAndroid项目的功能清单为例子进行讲解：

<?xml version="1.0"encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="cn.csdn.android.demo"
 android:versionCode="1"
 android:versionName="1.0">

 <uses-sdk android:minSdkVersion="8"/>

 <application
 android:icon="@drawable/ic\_launcher"
 android:label="@string/app\_name">
 <activity
 android:label="@string/app\_name"
 android:name=".HelloActivity">
 <intent-filter >
 <action android:name="android.intent.action.MAIN" />

 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>
 </application>
</manifest>

1.4.1 元素

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="cn.csdn.android.demo"
 android:versionCode="1"
 android:versionName="1.0">

元素是AndroidManifest.xml的根元素,”xmlns:android”是指该文件的命名空间，“package”属性是Android应用所在的包，“android:versionCode”指定应用的版本号，如果应用不断升级，则需要修改这个值，”android:versionName”是版本的名称，这个可以根据自己的喜爱改变 1.4.2 元素

<application
 android:icon="@drawable/ic\_launcher"
 android:label="@string/app\_name">
 <activity
 android:label="@string/app\_name"
 android:name=".HelloActivity">
 <intent-filter >
 <action android:name="android.intent.action.MAIN"/>

 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>
 </application>

元素是一个很重要的元素，开发组件都会在此下定义 元素的”icon”属性是用来设定应用的图标，其中“@drawable/ic\_launcher”的意思是:在R.java文件中的drawable静态内部类下的icon，如下图所示 元素的“label”属性用来设定应用的名称，其中“@string/app\_name”和上述的一样，也是R.java文件中的string静态内部类下的app\_name 1.4.3 元素

<activity
 android:label="@string/app\_name"
 android:name=".HelloActivity" >
 <intent-filter >
 <action android:name="android.intent.action.MAIN" />

 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>

元素的作用是注册一个activity信息，当我们在创建“HelloAndroid”这个项目时，指定了“Created Activity”属性为“HelloActivity”，然后ADT在生成项目时帮我们自动创建了一个Activity，就是“HelloActivity.java”； 元素的“name“属性指定的是Activity的类名，其中“.HelloActivity”中的“.”指的是元素中的“package”属性中指定的当前包，所以“.HelloActivity”就相当于“cn.csdn.android.demo.HelloActivity.java”，如果Activity在应用的包中可以不写“.”，但是为了避免出错，还是写上这个点把 1.4.4元素

 <category android:name="android.intent.category.LAUNCHER" />

 如果直接翻译的话是“意图过滤器”，组件通过告诉它们所具备的功能，就是能响应意图类型，在intent中设置action, data, categroy之后在对应的intentfilter中设置相同的属性即可通过过滤被activity调用 1.5<project.properties>应用要求运行的最低Android版本 1.6 存放Android自身的jar包 43.ddms 和traceview的区别.

daivilk debug manager system 1.在应用的主activity的onCreate方法中加入Debug.startMethodTracing("要生成的traceview文件的名字"); 2.同样在主activity的onStop方法中加入Debug.stopMethodTracing(); 3.同时要在AndroidManifest.xml文件中配置权限 3.重新编译，安装，启动服务，测试完成取对应的traceview文件(adb pull /sdcard/xxxx.trace)。 4.直接在命令行输入traceview xxxxtrace，弹出traceview窗口，分析对应的应用即可。

traceview 分析程序执行时间和效率

KPI : key performance information : 关键性能指标: splash界面不能超过5秒 从splash 界面加载mainactivity 不能超过0.7秒 44.谈谈NDK NDK全称：Native Development Kit。 1、NDK是一系列工具的集合。 \* NDK提供了一系列的工具，帮助开发者快速开发C（或C++）的动态库，并能自动将so和java应用一起打包成apk。这些工具对开发者的帮助是巨大的。[1] \* NDK集成了交叉编译器，并提供了相应的mk文件隔离平台、CPU、API等差异，开发人员只需要简单修改mk文件（指出“哪些文件需要编译”、“编译特性要求”等），就可以创建出so。 \* NDK可以自动地将so和Java应用一起打包，极大地减轻了开发人员的打包工作。 2、NDK提供了一份稳定、功能有限的API头文件声明。 Google明确声明该API是稳定的，在后续所有版本中都稳定支持当前发布的API。从该版本的NDK中看出，这些API支持的功能非常有限，包含有：C标准库（libc）、标准数学库（libm）、压缩库（libz）、Log库（liblog）。 45.请介绍下Android的数据存储方式。

一.SharedPreferences方式 二sdcard 三内部存储 四SqliteDatabase 　　五. 网络存储方式46.谈谈推送,优缺点以及实现原理

本文主旨在于，对目前Android平台上最主流的几种消息推送方案进行分析和对比，比较客观地反映出这些推送方案的优缺点，帮助大家选择最合适的实施方案。

方案1、使用GCM服务（Google Cloud Messaging） 简介：Google推出的云消息服务，即第二代的C2DM。 优点：Google提供的服务、原生、简单，无需实现和部署服务端。 缺点：Android版本限制（必须大于2.2版本），该服务在国内不够稳定、需要用户绑定Google帐号，受限于Google。

方案2、使用XMPP协议（Openfire + Spark + Smack） 简介：基于XML协议的通讯协议，前身是Jabber，目前已由IETF国际标准化组织完成了标准化工作。 优点：协议成熟、强大、可扩展性强、目前主要应用于许多聊天系统中，且已有开源的Java版的开发实例androidpn。 缺点：协议较复杂、冗余（基于XML）、费流量、费电，部署硬件成本高。

方案3、使用MQTT协议（更多信息见：http://mqtt.org/） 简介：轻量级的、基于代理的“发布/订阅”模式的消息传输协议。 优点：协议简洁、小巧、可扩展性强、省流量、省电，目前已经应用到企业领域（参考：http://mqtt.org/software），且已有C++版的服务端组件rsmb。 缺点：不够成熟、实现较复杂、服务端组件rsmb不开源，部署硬件成本较高。

方案4、使用HTTP轮循方式 简介：定时向HTTP服务端接口（Web Service API）获取最新消息。 优点：实现简单、可控性强，部署硬件成本低。 缺点：实时性差。

对各个方案的优缺点的研究和对比，推荐使用MQTT协议的方案进行实现，主要原因是：MQTT最快速，也最省流量（固定头长度仅为2字节），且极易扩展，适合二次开发。接下来，我们就来分析使用MQTT方案进行Android消息的原理和方法，并架设自己的推送服务。 1、推送原理分析

实际上，其他推送系统（包括GCM、XMPP方案）的原理都与此类似。

2、推送服务端准备

a> 下载&解压rsmb安装包（下载地址：http://www.alphaworks.ibm.com/tech/rsmb） b> 进入对应的目录，比如32位的Linux系统则应该进入linux\_ia32目录。 c> 编辑配置文件broker\_1883.cfg，配置如下：

[html] view plaincopy port 1883
max\_inflight\_messages 10
max\_queued\_messages 1000
d> 运行./broker broker\_1883.cfg，显示如下： 20120823 110454.039 CWNAN9999I Really Small Message Broker 20120823 110454.039 CWNAN9997I Licensed Materials - Property of IBM 20120823 110454.039 CWNAN9996I Copyright IBM Corp. 2007, 2010 All Rights Reserved 20120823 110454.039 CWNAN9995I US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp. 20120823 110454.039 CWNAN0049I Configuration file name is broker\_1883.cfg 20120823 110454.040 CWNAN0053I Version 1.2.0, Aug 18 2010 17:03:35 20120823 110454.040 CWNAN0054I Features included: bridge 20120823 110454.040 CWNAN9993I Author: Ian Craggs (icraggs@uk.ibm.com) 20120823 110454.040 CWNAN0014I MQTT protocol starting, listening on port 1883 ... ... 这样，推送服务的服务端就已经准备好了，监听1883端口。

3、推送客户端准备

a> 下载&解压AndroidPushNotificationsDemo项目（下载地址：https://github.com/tokudu/AndroidPushNotificationsDemo） b> 将该项目导入Eclipse中（File -> Export -> Existing Projects into Workspace） c> 修改PushService.java中的MQTT\_HOST常量为推送服务端的IP地址。 d> 启动Android模拟器，并安装该项目。

注意：在新版本的Android SDK中可能会遇到以下错误。 ... ... 08-23 02:28:44.184: W/dalvikvm(282): VFY: unable to find class referenced in signature (Lcom/ibm/mqtt/MqttPersistence;) 08-23 02:28:44.194: I/dalvikvm(282): Failed resolving Lcom/tokudu/demo/PushService$MQTTConnection;interface35′Lcom/ibm/mqtt/MqttSimpleCallback;′08−2302:28:44.194:W/dalvikvm(282):Linkofclass′Lcom/tokudu/demo/PushService$MQTTConnection;' failed 08-23 02:28:44.194: E/dalvikvm(282): Could not find class 'com.tokudu.demo.PushService$MQTTConnection′,referencedfrommethodcom.tokudu.demo.PushService.connect08−2302:28:44.194:W/dalvikvm(282):VFY:unabletoresolvenew−instance42(Lcom/tokudu/demo/PushService$MQTTConnection;) in Lcom/tokudu/demo/PushService; ... ... 08-23 02:28:44.404: E/AndroidRuntime(282): java.lang.VerifyError: com.tokudu.demo.PushService 08-23 02:28:44.404: E/AndroidRuntime(282): at com.tokudu.demo.PushActivity$1.onClick(PushActivity.java:32)08−2302:28:44.404:E/AndroidRuntime(282):atandroid.view.View.performClick(View.java:2408)08−2302:28:44.404:E/AndroidRuntime(282):atandroid.view.View$PerformClick.run(View.java:8816) 08-23 02:28:44.404: E/AndroidRuntime(282): at android.os.Handler.handleCallback(Handler.java:587) 08-23 02:28:44.404: E/AndroidRuntime(282): at android.os.Handler.dispatchMessage(Handler.java:92) 08-23 02:28:44.404: E/AndroidRuntime(282): at android.os.Looper.loop(Looper.java:123) 08-23 02:28:44.404: E/AndroidRuntime(282): at android.app.ActivityThread.main(ActivityThread.java:4627) 08-23 02:28:44.404: E/AndroidRuntime(282): at java.lang.reflect.Method.invokeNative(Native Method) 08-23 02:28:44.404: E/AndroidRuntime(282): at java.lang.reflect.Method.invoke(Method.java:521) 08-23 02:28:44.404: E/AndroidRuntime(282): at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:868) 08-23 02:28:44.404: E/AndroidRuntime(282): at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:626) 08-23 02:28:44.404: E/AndroidRuntime(282): at dalvik.system.NativeStart.main(Native Method) ... ... 原因是发布的时候没有加入wmqtt.jar包，解决办法如下： 1> 在项目根目录下创建libs目录，并把wmqtt.jar包移入该目录。 2> 重新配置项目的Java Build Path（右键菜单中的Properties选项中）。 3> 重新打包发布即可。

运行效果如下：

点击“Start Push Service”按钮即可开启推送服务。这时我们可以看到rsmb的服务日志中打出以下提示： 20120823 113742.297 CWNAN0033I Connection attempt to listener 1883 received from client tokudu/9774d56d682e549c on address 192.168.28.39:3345 其中的“9774d56d682e549c”就是对应的客户端ID号。

4、发送服务准备

a> 下载&解压PHP版的发送服务端代码send\_mqtt.zip（下载地址：http://download.csdn.net/detail/shagoo/4520102） b> 修改etc/config.php中推送服务端的IP地址和端口号，即MQTT\_SERVER\_HOST和MQTT\_SERVER\_POST常量。 c> 打开对应的URL地址，就可以看到发送服务的界面，实际上就是向对应的推送客户端推送消息。47.谈谈数据加密

数据加密又称密码学，它是一门历史悠久的技术，指通过加密算法和加密密钥将明文转变为密文，而解密则是通过解密算法和解密密钥将密文恢复为明文。数据加密目前仍是计算机系统对信息进行保护的一种最可靠的办法。它利用密码技术对信息进行加密，实现信息隐蔽，从而起到保护信息的安全的作用。用自己的话来说就是，只有双方才知道的协议。 数据加密 - 密码算法分类 1按发展进程分密码的发展:古典密码,对称密钥 密码公开密钥密码. 2按加密模式分对称算法:序列密码和分组密码. 经典密码 代替密码: 简单代替多名或同音代替多表代替多字母或多码代替换位密码: •对称加密算法 DES AES •非对称公钥算法 RSA 背包密码McEliece密码Rabin 椭圆曲线EIGamal D\_H 48.解决问题和思考问题的方式

首先查看官方提供的API，通过自己对功能的理解，然后通过网络的途径，下载demo可以选择 github ，查找问题可以选择 stackover flow，然后可以问身边的朋友。49.列举7到12个设计模式 以及它们的应用场景

设计模式，提供了很多软件工程问题所需处理的解决方案。 根据模式的目的可分为3类： 1.创建型模式：与对象的创建有关。 2.结构性模式：处理类与对象的组合。 3.行为性模式：对类或对象怎样交互和怎样 分配职责进行描述。 面向对象设计的2个基本原则: 1.针对接口编程，而不是针对实现编程。 2.优先使用对象组合，而不是类继承。 面向对象设计的5个设计原则： 1.单一职责原则(SRP) 2.开放封闭原则(OCP) 3.Liskov替换原则(LSP) 4.依赖倒置原则(DIP) 5.接口隔离原则(ISP) 23中设计模式： 1.创建型模式： (1).工厂方法模式 (2).抽象工厂模式 (3).创建者模式 (4).原型模式 (5).单例模式 2.结构型模式： (6).适配器模式 (7).桥模式 (8).组合模式 (9).装饰模式 (10).外观模式 (11).享元模式 (12).代理模式 3.行为型模式 (13).解释器模式 (14).模板方法模式 (15).职责链模式 (16).命令模式 (17).迭代器模式 (18).中介者模式 (19).备忘录模式 (20).观察者模式 (21).状态模式 (22).策略模式 (23).访问者模式 除此之外，后来人发现很多新的模式，如空模式等。 下面列举几个常见的问题导致重新设计，可能需要设计模式来分析解决： 1.通过显示的指定一个类来创建对象 2.对特殊操作的依赖 3.对硬件和软件平台的依赖 4.对对象表示或实现的依赖 5.算法依赖 6.紧耦合 7.通过生产子类来扩展功能 8.不能方便的对类进行修改 软件的设计臭味： 1.僵化性 2.脆弱性 3.顽固性 4.粘滞性 5.不必要的复杂性 6.不必要的重复 7.晦涩性 ... ... 总而言之，一句话，面向对象特性+原则+模式，折腾来折腾去就是这么个回事。

设计模式（Design Patterns） ——可复用面向对象软件的基础 设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。本章系Java之美[从菜鸟到高手演变]系列之设计模式，我们会以理论与实践相结合的方式来进行本章的学习，希望广大程序爱好者，学好设计模式，做一个优秀的软件工程师！ 在阅读过程中有任何问题，请及时联系：egg。 邮箱：xtfggef@gmail.com 微博：http://weibo.com/xtfggef 如有转载，请说明出处：http://blog.csdn.net/zhangerqing 企业级项目实战(带源码)地址：http://zz563143188.iteye.com/blog/1825168 23种模式java实现源码及收集五年的开发资料下载地址： http://pan.baidu.com/share/home?uk=4076915866&view=share 一、设计模式的分类 总体来说设计模式分为三大类： 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。 其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：

二、设计模式的六大原则 1、开闭原则（Open Close Principle） 开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。 2、里氏代换原则（Liskov Substitution Principle） 里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科 3、依赖倒转原则（Dependence Inversion Principle） 这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。 4、接口隔离原则（Interface Segregation Principle） 这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。 5、迪米特法则（最少知道原则）（Demeter Principle） 为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。 6、合成复用原则（Composite Reuse Principle） 原则是尽量使用合成/聚合的方式，而不是使用继承。 三、Java的23中设计模式 从这一块开始，我们详细介绍Java中23种设计模式的概念，应用场景等情况，并结合他们的特点及设计模式的原则进行分析。 1、工厂方法模式（Factory Method） 工厂方法模式分为三种： 11、普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。首先看下关系图：

举例如下：（我们举一个发送邮件和短信的例子） 首先，创建二者的共同接口： java public interface Sender { public void Send(); } 其次，创建实现类：java public class MailSender implements Sender {
@Override
public void Send() {
System.out.println("this is mailsender!");
}
} java public class SmsSender implements Sender {

@Override
public void Send() {
 System.out.println("this is sms sender!");
}

}
最后，建工厂类：java public class SendFactory {

public Sender produce(String type) {
 if ("mail".equals(type)) {
 return new MailSender();
 } else if ("sms".equals(type)) {
 return new SmsSender();
 } else {
 System.out.println("请输入正确的类型!");
 return null;
 }
}

}

我们来测试下：
public class FactoryTest {

 public static void main(String[] args) {
 SendFactory factory = new SendFactory();
 Sender sender = factory.produce("sms");
 sender.Send();
 }
}

输出：this is sms sender! 22、多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。关系图：

将上面的代码做下修改，改动下SendFactory类就行，如下：

public class SendFactory {
 public Sender produceMail(){
 return new MailSender();
 }

 public Sender produceSms(){
 return new SmsSender();
 }
}

测试类如下：

public class FactoryTest {

 public static void main(String[] args) {
 SendFactory factory = new SendFactory();
 Sender sender = factory.produceMail();
 sender.Send();
 }
}

输出：this is mailsender! 33、静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

public class SendFactory {

 public static Sender produceMail(){
 return new MailSender();
 }

 public static Sender produceSms(){
 return new SmsSender();
 }
}

public class FactoryTest {

 public static void main(String[] args) {
 Sender sender = SendFactory.produceMail();
 sender.Send();
 }
}

输出：this is mailsender! 总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。 2、抽象工厂模式（Abstract Factory） 工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。

请看例子： java public interface Sender { public void Send(); } 两个实现类：java public class MailSender implements Sender {
@Override
public void Send() {
System.out.println("this is mailsender!");
}
}
java public class SmsSender implements Sender {

@Override
public void Send() {
 System.out.println("this is sms sender!");
}

} 两个工厂类：java public class SendMailFactory implements Provider {

@Override
public Sender produce(){
 return new MailSender();
}

}
java public class SendSmsFactory implements Provider{

@Override
public Sender produce() {
 return new SmsSender();
}

}
在提供一个接口：java public interface Provider {
public Sender produce();
}
测试类：java public class Test {

public static void main(String[] args) {
 Provider provider = new SendMailFactory();
 Sender sender = provider.produce();
 sender.Send();
}

}
其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！ 3、单例模式（Singleton） 单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处： 1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。 2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。 3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。 首先我们写一个简单的单例类：java public class Singleton {

/\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/
private static Singleton instance = null;

/\* 私有构造方法，防止被实例化 \*/
private Singleton() {
}

/\* 静态工程方法，创建实例 \*/
public static Singleton getInstance() {
 if (instance == null) {
 instance = new Singleton();
 }
 return instance;
}

/\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
public Object readResolve() {
 return instance;
}

} 这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：java public static synchronized Singleton getInstance() {
if (instance == null) {
instance = new Singleton();
}
return instance;
}
```

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

public static Singleton getInstance() {
 if (instance == null) {
 synchronized (instance) {
 if (instance == null) {
 instance = new Singleton();
 }
 }
 }
 return instance;
 }

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间，然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例： a>A、B线程同时进入了第一个if判断 b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton(); c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。 d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。 e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。 所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

private static class SingletonFactory{
 private static Singleton instance = new Singleton();
 }
 public static Singleton getInstance(){
 return SingletonFactory.instance;
 }

实际情况是，单例模式使用内部类来维护单例的实现，JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

public class Singleton {

 /\* 私有构造方法，防止被实例化 \*/
 private Singleton() {
 }

 /\* 此处使用一个内部类来维护单例 \*/
 private static class SingletonFactory {
 private static Singleton instance = new Singleton();
 }

 /\* 获取实例 \*/
 public static Singleton getInstance() {
 return SingletonFactory.instance;
 }

 /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
 public Object readResolve() {
 return getInstance();
 }
}

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字，也是可以的：

public class SingletonTest {

 private static SingletonTest instance = null;

 private SingletonTest() {
 }

 private static synchronized void syncInit() {
 if (instance == null) {
 instance = new SingletonTest();
 }
 }

 public static SingletonTest getInstance() {
 if (instance == null) {
 syncInit();
 }
 return instance;
 }
}

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。 补充：采用"影子实例"的办法为单例对象的属性同步更新

public class SingletonTest {

 private static SingletonTest instance = null;
 private Vector properties = null;

 public Vector getProperties() {
 return properties;
 }

 private SingletonTest() {
 }

 private static synchronized void syncInit() {
 if (instance == null) {
 instance = new SingletonTest();
 }
 }

 public static SingletonTest getInstance() {
 if (instance == null) {
 syncInit();
 }
 return instance;
 }

 public void updateProperties() {
 SingletonTest shadow = new SingletonTest();
 properties = shadow.getProperties();
 }
}

通过单例模式的学习告诉我们： 1、单例模式理解起来简单，但是具体实现起来还是有一定的难度。 2、synchronized关键字锁定的是对象，在用的时候，一定要在恰当的地方使用（注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁）。 到这儿，单例模式基本已经讲完了，结尾处，笔者突然想到另一个问题，就是采用类的静态方法，实现单例模式的效果，也是可行的，此处二者有什么不同？ 首先，静态类不能实现接口。（从类的角度说是可以的，但是那样就破坏了静态了。因为接口中不允许有static修饰的方法，所以即使实现了也是非静态的） 其次，单例可以被延迟初始化，静态类一般在第一次加载是初始化。之所以延迟加载，是因为有些类比较庞大，所以延迟加载有助于提升性能。 再次，单例类可以被继承，他的方法可以被覆写。但是静态类内部方法都是static，无法被覆写。 最后一点，单例类比较灵活，毕竟从实现上只是一个普通的Java类，只要满足单例的基本需求，你可以在里面随心所欲的实现一些其它功能，但是静态类不行。从上面这些概括中，基本可以看出二者的区别，但是，从另一方面讲，我们上面最后实现的那个单例模式，内部就是用一个静态类来实现的，所以，二者有很大的关联，只是我们考虑问题的层面不同罢了。两种思想的结合，才能造就出完美的解决方案，就像HashMap采用数组+链表来实现一样，其实生活中很多事情都是这样，单用不同的方法来处理问题，总是有优点也有缺点，最完美的方法是，结合各个方法的优点，才能最好的解决问题！ 4、建造者模式（Builder） 工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。我们看一下代码： 还和前面一样，一个Sender接口，两个实现类MailSender和SmsSender。最后，建造者类如下：

public class Builder {

 private List<Sender> list = new ArrayList<Sender>();

 public void produceMailSender(int count){
 for(int i=0; i<count; i++){
 list.add(new MailSender());
 }
 }

 public void produceSmsSender(int count){
 for(int i=0; i<count; i++){
 list.add(new SmsSender());
 }
 }
}

测试类：

public class Test {

 public static void main(String[] args) {
 Builder builder = new Builder();
 builder.produceMailSender(10);
 }
}

从这点看出，建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定。 5、原型模式（Prototype） 原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。本小结会通过对象的复制，进行讲解。在Java中，复制对象是通过clone()实现的，先创建一个原型类：

public class Prototype implements Cloneable {

 public Object clone() throws CloneNotSupportedException {
 Prototype proto = (Prototype) super.clone();
 return proto;
 }
}

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，我会在另一篇文章中，关于解读Java中本地方法的调用，此处不再深究。在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念： 浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。 深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。 此处，写一个深浅复制的例子：

public class Prototype implements Cloneable, Serializable {

 private static final long serialVersionUID = 1L;
 private String string;

 private SerializableObject obj;

 /\* 浅复制 \*/
 public Object clone() throws CloneNotSupportedException {
 Prototype proto = (Prototype) super.clone();
 return proto;
 }

 /\* 深复制 \*/
 public Object deepClone() throws IOException, ClassNotFoundException {

 /\* 写入当前对象的二进制流 \*/
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 ObjectOutputStream oos = new ObjectOutputStream(bos);
 oos.writeObject(this);

 /\* 读出二进制流产生的新对象 \*/
 ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
 ObjectInputStream ois = new ObjectInputStream(bis);
 return ois.readObject();
 }

 public String getString() {
 return string;
 }

 public void setString(String string) {
 this.string = string;
 }

 public SerializableObject getObj() {
 return obj;
 }

 public void setObj(SerializableObject obj) {
 this.obj = obj;
 }

}

class SerializableObject implements Serializable {
 private static final long serialVersionUID = 1L;
}

要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。 我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。首先，我们来看看类的适配器模式，先看类图：

核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

public class Source {

 public void method1() {
 System.out.println("this is original method!");
 }
}

public interface Targetable {

 /\* 与原类中的方法相同 \*/
 public void method1();

 /\* 新类的方法 \*/
 public void method2();
}

public class Adapter extends Source implements Targetable {

 @Override
 public void method2() {
 System.out.println("this is the targetable method!");
 }
}

Adapter类继承Source类，实现Targetable接口，下面是测试类：

public class AdapterTest {

 public static void main(String[] args) {
 Targetable target = new Adapter(); 接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户
 target.method1();
 target.method2();
 }
}

输出： this is original method! this is the targetable method! 这样Targetable接口的实现类就具有了Source类的功能。 对象的适配器模式 基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。看图：

只需要修改Adapter类的源码即可：

public class Wrapper implements Targetable {

 private Source source;

 public Wrapper(Source source){
 super();
 this.source = source;
 }
 @Override
 public void method2() {
 System.out.println("this is the targetable method!");
 }

 @Override
 public void method1() {
 source.method1();
 }
}

测试类：

public class AdapterTest {

 public static void main(String[] args) {
 Source source = new Source();
 Targetable target = new Wrapper(source);
 target.method1();
 target.method2();
 }
}

输出与第一种一样，只是适配的方法不同而已。 第三种适配器模式是接口的适配器模式，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：

这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

public interface Sourceable {

 public void method1();
 public void method2();
}

抽象类Wrapper2：

public abstract class Wrapper2 implements Sourceable{

 public void method1(){}
 public void method2(){}
}

public class SourceSub1 extends Wrapper2 {
 public void method1(){
 System.out.println("the sourceable interface's first Sub1!");
 }
}

public class SourceSub2 extends Wrapper2 {
 public void method2(){
 System.out.println("the sourceable interface's second Sub2!");
 }
}

public class WrapperTest {

 public static void main(String[] args) {
 Sourceable source1 = new SourceSub1();
 Sourceable source2 = new SourceSub2();

 source1.method1();
 source1.method2();
 source2.method1();
 source2.method2();
 }
}

测试输出： the sourceable interface's first Sub1! the sourceable interface's second Sub2! 达到了我们的效果！ 讲了这么多，总结一下三种适配器模式的应用场景： 类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。 对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。 接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。 7、装饰模式（Decorator） 顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：

Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

public interface Sourceable {
 public void method();
}

public class Source implements Sourceable {

 @Override
 public void method() {
 System.out.println("the original method!");
 }
}

public class Decorator implements Sourceable {

 private Sourceable source;

 public Decorator(Sourceable source){
 super();
 this.source = source;
 }
 @Override
 public void method() {
 System.out.println("before decorator!");
 source.method();
 System.out.println("after decorator!");
 }
}

测试类：

public class DecoratorTest {

 public static void main(String[] args) {
 Sourceable source = new Source();
 Sourceable obj = new Decorator(source);
 obj.method();
 }
}

输出： before decorator! the original method! after decorator! 装饰器模式的应用场景： 1、需要扩展一个类的功能。 2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。） 缺点：产生过多相似的对象，不易排错！ 8、代理模式（Proxy） 其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：

根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

public interface Sourceable {
 public void method();
}

public class Source implements Sourceable {

 @Override
 public void method() {
 System.out.println("the original method!");
 }
}

public class Proxy implements Sourceable {

 private Source source;
 public Proxy(){
 super();
 this.source = new Source();
 }
 @Override
 public void method() {
 before();
 source.method();
 atfer();
 }
 private void atfer() {
 System.out.println("after proxy!");
 }
 private void before() {
 System.out.println("before proxy!");
 }
}

测试类：

public class ProxyTest {

 public static void main(String[] args) {
 Sourceable source = new Proxy();
 source.method();
 }

}

输出： before proxy! the original method! after proxy! 代理模式的应用场景： 如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法： 1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。 2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。 使用代理模式，可以将功能划分的更加清晰，有助于后期维护！ 9、外观模式（Facade） 外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）

我们先看下实现类：

public class CPU {

 public void startup(){
 System.out.println("cpu startup!");
 }

 public void shutdown(){
 System.out.println("cpu shutdown!");
 }
}

public class Memory {

 public void startup(){
 System.out.println("memory startup!");
 }

 public void shutdown(){
 System.out.println("memory shutdown!");
 }
}

public class Disk {

 public void startup(){
 System.out.println("disk startup!");
 }

 public void shutdown(){
 System.out.println("disk shutdown!");
 }
}

public class Computer {
 private CPU cpu;
 private Memory memory;
 private Disk disk;

 public Computer(){
 cpu = new CPU();
 memory = new Memory();
 disk = new Disk();
 }

 public void startup(){
 System.out.println("start the computer!");
 cpu.startup();
 memory.startup();
 disk.startup();
 System.out.println("start computer finished!");
 }

 public void shutdown(){
 System.out.println("begin to close the computer!");
 cpu.shutdown();
 memory.shutdown();
 disk.shutdown();
 System.out.println("computer closed!");
 }
}

User类如下：

public class User {

 public static void main(String[] args) {
 Computer computer = new Computer();
 computer.startup();
 computer.shutdown();
 }
}

输出： start the computer! cpu startup! memory startup! disk startup! start computer finished! begin to close the computer! cpu shutdown! memory shutdown! disk shutdown! computer closed! 如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观模式！ 10、桥接模式（Bridge） 桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：

实现代码： 先定义接口：

public interface Sourceable {
 public void method();
}

分别定义两个实现类：

public class SourceSub1 implements Sourceable {

 @Override
 public void method() {
 System.out.println("this is the first sub!");
 }
}

public class SourceSub2 implements Sourceable {

 @Override
 public void method() {
 System.out.println("this is the second sub!");
 }
}

定义一个桥，持有Sourceable的一个实例：

public abstract class Bridge {
 private Sourceable source;

 public void method(){
 source.method();
 }

 public Sourceable getSource() {
 return source;
 }

 public void setSource(Sourceable source) {
 this.source = source;
 }
}

public class MyBridge extends Bridge {
 public void method(){
 getSource().method();
 }
}

测试类：

public class BridgeTest {

 public static void main(String[] args) {

 Bridge bridge = new MyBridge();

 /\*调用第一个对象\*/
 Sourceable source1 = new SourceSub1();
 bridge.setSource(source1);
 bridge.method();

 /\*调用第二个对象\*/
 Sourceable source2 = new SourceSub2();
 bridge.setSource(source2);
 bridge.method();
 }
}

output： this is the first sub! this is the second sub! 这样，就通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们JDBC连接的原理，有数据库学习基础的，一结合就都懂了。

11、组合模式（Composite） 组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便，看看关系图：

直接来看代码：

public class TreeNode {

 private String name;
 private TreeNode parent;
 private Vector<TreeNode> children = new Vector<TreeNode>();

 public TreeNode(String name){
 this.name = name;
 }

 public String getName() {
 return name;
 }

 public void setName(String name) {
 this.name = name;
 }

 public TreeNode getParent() {
 return parent;
 }

 public void setParent(TreeNode parent) {
 this.parent = parent;
 }

 //添加孩子节点
 public void add(TreeNode node){
 children.add(node);
 }

 //删除孩子节点
 public void remove(TreeNode node){
 children.remove(node);
 }

 //取得孩子节点
 public Enumeration<TreeNode> getChildren(){
 return children.elements();
 }
}

public class Tree {

 TreeNode root = null;

 public Tree(String name) {
 root = new TreeNode(name);
 }

 public static void main(String[] args) {
 Tree tree = new Tree("A");
 TreeNode nodeB = new TreeNode("B");
 TreeNode nodeC = new TreeNode("C");

 nodeB.add(nodeC);
 tree.root.add(nodeB);
 System.out.println("build the tree finished!");
 }
}

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。 12、享元模式（Flyweight） 享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。 看个例子：

看下数据库连接池的代码：

public class ConnectionPool {

 private Vector<Connection> pool;

 /\*公有属性\*/
 private String url = "jdbc:mysql://localhost:3306/test";
 private String username = "root";
 private String password = "root";
 private String driverClassName = "com.mysql.jdbc.Driver";

 private int poolSize = 100;
 private static ConnectionPool instance = null;
 Connection conn = null;

 /\*构造方法，做一些初始化工作\*/
 private ConnectionPool() {
 pool = new Vector<Connection>(poolSize);

 for (int i = 0; i < poolSize; i++) {
 try {
 Class.forName(driverClassName);
 conn = DriverManager.getConnection(url, username, password);
 pool.add(conn);
 } catch (ClassNotFoundException e) {
 e.printStackTrace();
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
 }

 /\* 返回连接到连接池 \*/
 public synchronized void release() {
 pool.add(conn);
 }

 /\* 返回连接池中的一个数据库连接 \*/
 public synchronized Connection getConnection() {
 if (pool.size() > 0) {
 Connection conn = pool.get(0);
 pool.remove(conn);
 return conn;
 } else {
 return null;
 }
 }
}

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！本章讲解了7种结构型模式，因为篇幅的问题，剩下的11种行为型模式， 本章是关于设计模式的最后一讲，会讲到第三种设计模式——行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。这段时间一直在写关于设计模式的东西，终于写到一半了，写博文是个很费时间的东西，因为我得为读者负责，不论是图还是代码还是表述，都希望能尽量写清楚，以便读者理解，我想不论是我还是读者，都希望看到高质量的博文出来，从我本人出发，我会一直坚持下去，不断更新，源源动力来自于读者朋友们的不断支持，我会尽自己的努力，写好每一篇文章！希望大家能不断给出意见和建议，共同打造完美的博文！

先来张图，看看这11中模式的关系： 第一类：通过父类与子类的关系进行实现。第二类：两个类之间。第三类：类的状态。第四类：通过中间类

13、策略模式（strategy） 策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数，关系图如下：

图中ICalculator提供同意的方法， AbstractCalculator是辅助类，提供辅助方法，接下来，依次实现下每个类： 首先统一接口：

public interface ICalculator {
 public int calculate(String exp);
}

辅助类：

public abstract class AbstractCalculator {

 public int[] split(String exp,String opt){
 String array[] = exp.split(opt);
 int arrayInt[] = new int[2];
 arrayInt[0] = Integer.parseInt(array[0]);
 arrayInt[1] = Integer.parseInt(array[1]);
 return arrayInt;
 }
}

三个实现类： ```java public class Plus extends AbstractCalculator implements ICalculator {

@Override
public int calculate(String exp) {
 int arrayInt[] = split(exp,"\\+");
 return arrayInt[0]+arrayInt[1];
}

}

public class Minus extends AbstractCalculator implements ICalculator {

 @Override
 public int calculate(String exp) {
 int arrayInt[] = split(exp,"-");
 return arrayInt[0]-arrayInt[1];
 }

}

public class Multiply extends AbstractCalculator implements ICalculator {

 @Override
 public int calculate(String exp) {
 int arrayInt[] = split(exp,"\\\*");
 return arrayInt[0]\*arrayInt[1];
 }
}

简单的测试类：

public class StrategyTest {

 public static void main(String[] args) {
 String exp = "2+8";
 ICalculator cal = new Plus();
 int result = cal.calculate(exp);
 System.out.println(result);
 }
}

输出：10 策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。 14、模板方法模式（Template Method） 解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用，先看个关系图：

就是在AbstractCalculator类中定义一个主方法calculate，calculate()调用spilt()等，Plus和Minus分别继承AbstractCalculator类，通过对AbstractCalculator的调用实现对子类的调用，看下面的例子：

public abstract class AbstractCalculator {

 /\*主方法，实现对本类其它方法的调用\*/
 public final int calculate(String exp,String opt){
 int array[] = split(exp,opt);
 return calculate(array[0],array[1]);
 }

 /\*被子类重写的方法\*/
 abstract public int calculate(int num1,int num2);

 public int[] split(String exp,String opt){
 String array[] = exp.split(opt);
 int arrayInt[] = new int[2];
 arrayInt[0] = Integer.parseInt(array[0]);
 arrayInt[1] = Integer.parseInt(array[1]);
 return arrayInt;
 }
}

public class Plus extends AbstractCalculator {

 @Override
 public int calculate(int num1,int num2) {
 return num1 + num2;
 }
}

测试类：

public class StrategyTest {

 public static void main(String[] args) {
 String exp = "8+8";
 AbstractCalculator cal = new Plus();
 int result = cal.calculate(exp, "\\+");
 System.out.println(result);
 }
}

我跟踪下这个小程序的执行过程：首先将exp和"\+"做参数，调用AbstractCalculator类里的calculate(String,String)方法，在calculate(String,String)里调用同类的split()，之后再调用calculate(int ,int)方法，从这个方法进入到子类中，执行完return num1 + num2后，将值返回到AbstractCalculator类，赋给result，打印出来。正好验证了我们开头的思路。 15、观察者模式（Observer） 包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该 记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。先来看看关系图：

我解释下这些类的作用：MySubject类就是我们的主对象，Observer1和Observer2是依赖于MySubject的对象，当MySubject变化时，Observer1和Observer2必然变化。AbstractSubject类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当MySubject变化时，负责通知在列表内存在的对象。我们看实现代码： 一个Observer接口：

public interface Observer {
 public void update();
}

两个实现类：

public class Observer1 implements Observer {

 @Override
 public void update() {
 System.out.println("observer1 has received!");
 }
}

public class Observer2 implements Observer {

 @Override
 public void update() {
 System.out.println("observer2 has received!");
 }

}

Subject接口及实现类：

public interface Subject {

 /\*增加观察者\*/
 public void add(Observer observer);

 /\*删除观察者\*/
 public void del(Observer observer);

 /\*通知所有的观察者\*/
 public void notifyObservers();

 /\*自身的操作\*/
 public void operation();
}

public abstract class AbstractSubject implements Subject {

 private Vector<Observer> vector = new Vector<Observer>();
 @Override
 public void add(Observer observer) {
 vector.add(observer);
 }

 @Override
 public void del(Observer observer) {
 vector.remove(observer);
 }

 @Override
 public void notifyObservers() {
 Enumeration<Observer> enumo = vector.elements();
 while(enumo.hasMoreElements()){
 enumo.nextElement().update();
 }
 }
}

public class MySubject extends AbstractSubject {

 @Override
 public void operation() {
 System.out.println("update self!");
 notifyObservers();
 }

}

测试类：

public class ObserverTest {

 public static void main(String[] args) {
 Subject sub = new MySubject();
 sub.add(new Observer1());
 sub.add(new Observer2());

 sub.operation();
 }

}

输出： update self! observer1 has received! observer2 has received! 这些东西，其实不难，只是有些抽象，不太容易整体理解，建议读者：根据关系图，新建项目，自己写代码（或者参考我的代码）,按照总体思路走一遍，这样才能体会它的思想，理解起来容易！ 16、迭代子模式（Iterator） 顾名思义，迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见，如果对集合类比较熟悉的话，理解本模式会十分轻松。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。我们看下关系图：

这个思路和我们常用的一模一样，MyCollection中定义了集合的一些操作，MyIterator中定义了一系列迭代操作，且持有Collection实例，我们来看看实现代码： 两个接口：

public interface Collection {

 public Iterator iterator();

 /\*取得集合元素\*/
 public Object get(int i);

 /\*取得集合大小\*/
 public int size();
}

public interface Iterator {
 //前移
 public Object previous();

 //后移
 public Object next();
 public boolean hasNext();

 //取得第一个元素
 public Object first();
}

两个实现：

public class MyCollection implements Collection {

 public String string[] = {"A","B","C","D","E"};
 @Override
 public Iterator iterator() {
 return new MyIterator(this);
 }

 @Override
 public Object get(int i) {
 return string[i];
 }

 @Override
 public int size() {
 return string.length;
 }
}

public class MyIterator implements Iterator {

 private Collection collection;
 private int pos = -1;

 public MyIterator(Collection collection){
 this.collection = collection;
 }

 @Override
 public Object previous() {
 if(pos > 0){
 pos--;
 }
 return collection.get(pos);
 }

 @Override
 public Object next() {
 if(pos<collection.size()-1){
 pos++;
 }
 return collection.get(pos);
 }

 @Override
 public boolean hasNext() {
 if(pos<collection.size()-1){
 return true;
 }else{
 return false;
 }
 }

 @Override
 public Object first() {
 pos = 0;
 return collection.get(pos);
 }

}

测试类：

public class Test {

 public static void main(String[] args) {
 Collection collection = new MyCollection();
 Iterator it = collection.iterator();

 while(it.hasNext()){
 System.out.println(it.next());
 }
 }
}

输出：A B C D E 此处我们貌似模拟了一个集合类的过程，感觉是不是很爽？其实JDK中各个类也都是这些基本的东西，加一些设计模式，再加一些优化放到一起的，只要我们把这些东西学会了，掌握好了，我们也可以写出自己的集合类，甚至框架！ 17、责任链模式（Chain of Responsibility）接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。先看看关系图：

Abstracthandler类提供了get和set方法，方便MyHandle类设置和修改引用对象，MyHandle类是核心，实例化后生成一系列相互持有的对象，构成一条链。

public interface Handler {
 public void operator();
}

public abstract class AbstractHandler {

 private Handler handler;

 public Handler getHandler() {
 return handler;
 }

 public void setHandler(Handler handler) {
 this.handler = handler;
 }

}

public class MyHandler extends AbstractHandler implements Handler {

 private String name;

 public MyHandler(String name) {
 this.name = name;
 }

 @Override
 public void operator() {
 System.out.println(name+"deal!");
 if(getHandler()!=null){
 getHandler().operator();
 }
 }
}

public class Test {

 public static void main(String[] args) {
 MyHandler h1 = new MyHandler("h1");
 MyHandler h2 = new MyHandler("h2");
 MyHandler h3 = new MyHandler("h3");

 h1.setHandler(h2);
 h2.setHandler(h3);

 h1.operator();
 }
}

输出： h1deal! h2deal! h3deal! 此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。 18、命令模式（Command） 命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看看关系图：

Invoker是调用者（司令员），Receiver是被调用者（士兵），MyCommand是命令，实现了Command接口，持有接收对象，看实现代码：

public interface Command {
 public void exe();
}

public class MyCommand implements Command {

 private Receiver receiver;

 public MyCommand(Receiver receiver) {
 this.receiver = receiver;
 }

 @Override
 public void exe() {
 receiver.action();
 }
}

public class Receiver {
 public void action(){
 System.out.println("command received!");
 }
}

public class Invoker {

 private Command command;

 public Invoker(Command command) {
 this.command = command;
 }

 public void action(){
 command.exe();
 }
}

public class Test {

 public static void main(String[] args) {
 Receiver receiver = new Receiver();
 Command cmd = new MyCommand(receiver);
 Invoker invoker = new Invoker(cmd);
 invoker.action();
 }
}

输出：command received! 这个很哈理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！ 其实每个设计模式都是很重要的一种思想，看上去很熟，其实是因为我们在学到的东西中都有涉及，尽管有时我们并不知道，其实在Java本身的设计之中处处都有体现，像AWT、JDBC、集合类、IO管道或者是Web框架，里面设计模式无处不在。因为我们篇幅有限，很难讲每一个设计模式都讲的很详细，不过我会尽我所能，尽量在有限的空间和篇幅内，把意思写清楚了，更好让大家明白。本章不出意外的话，应该是设计模式最后一讲了，首先还是上一下上篇开头的那个图：

本章讲讲第三类和第四类。 19、备忘录模式（Memento） 主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下：

Original类是原始类，里面有需要保存的属性value及创建一个备忘录类，用来保存value值。Memento类是备忘录类，Storage类是存储备忘录的类，持有Memento类的实例，该模式很好理解。直接看源码：

public class Original {

 private String value;

 public String getValue() {
 return value;
 }

 public void setValue(String value) {
 this.value = value;
 }

 public Original(String value) {
 this.value = value;
 }

 public Memento createMemento(){
 return new Memento(value);
 }

 public void restoreMemento(Memento memento){
 this.value = memento.getValue();
 }
}

public class Memento {

 private String value;

 public Memento(String value) {
 this.value = value;
 }

 public String getValue() {
 return value;
 }

 public void setValue(String value) {
 this.value = value;
 }
}

public class Storage {

 private Memento memento;

 public Storage(Memento memento) {
 this.memento = memento;
 }

 public Memento getMemento() {
 return memento;
 }

 public void setMemento(Memento memento) {
 this.memento = memento;
 }
}

测试类：

public class Test {

 public static void main(String[] args) {

 // 创建原始类
 Original origi = new Original("egg");

 // 创建备忘录
 Storage storage = new Storage(origi.createMemento());

 // 修改原始类的状态
 System.out.println("初始化状态为：" + origi.getValue());
 origi.setValue("niu");
 System.out.println("修改后的状态为：" + origi.getValue());

 // 回复原始类的状态
 origi.restoreMemento(storage.getMemento());
 System.out.println("恢复后的状态为：" + origi.getValue());
 }
}

输出： 初始化状态为：egg 修改后的状态为：niu 恢复后的状态为：egg 简单描述下：新建原始类时，value被初始化为egg，后经过修改，将value的值置为niu，最后倒数第二行进行恢复状态，结果成功恢复了。其实我觉得这个模式叫“备份-恢复”模式最形象。 20、状态模式（State） 核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿QQ来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图：

State类是个状态类，Context类可以实现切换，我们来看看代码：

package com.xtfggef.dp.state;

/\*\*
 \* 状态类的核心类
 \* 2012-12-1
 \* @author erqing
 \*
 \*/
public class State {

 private String value;

 public String getValue() {
 return value;
 }

 public void setValue(String value) {
 this.value = value;
 }

 public void method1(){
 System.out.println("execute the first opt!");
 }

 public void method2(){
 System.out.println("execute the second opt!");
 }
}

```java package com.xtfggef.dp.state;

/\*\* \* 状态模式的切换类 2012-12-1 \* @author erqing \*
\*/
public class Context {

private State state;

public Context(State state) {
 this.state = state;
}

public State getState() {
 return state;
}

public void setState(State state) {
 this.state = state;
}

public void method() {
 if (state.getValue().equals("state1")) {
 state.method1();
 } else if (state.getValue().equals("state2")) {
 state.method2();
 }
}

}
测试类：

public class Test {

 public static void main(String[] args) {

 State state = new State();
 Context context = new Context(state);

 //设置第一种状态
 state.setValue("state1");
 context.method();

 //设置第二种状态
 state.setValue("state2");
 context.method();
 }
}

输出：

execute the first opt! execute the second opt! 根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。21、访问者模式（Visitor） 访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。—— From 百科 简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。简单关系图：

来看看原码：一个Visitor类，存放要访问的对象，

public interface Visitor {
 public void visit(Subject sub);
}

public class MyVisitor implements Visitor {

 @Override
 public void visit(Subject sub) {
 System.out.println("visit the subject："+sub.getSubject());
 }
}

Subject类，accept方法，接受将要访问它的对象，getSubject()获取将要被访问的属性，

public interface Subject {
 public void accept(Visitor visitor);
 public String getSubject();
}

public class MySubject implements Subject {

 @Override
 public void accept(Visitor visitor) {
 visitor.visit(this);
 }

 @Override
 public String getSubject() {
 return "love";
 }
}

测试：

```java public class Test {

public static void main(String[] args) {

 Visitor visitor = new MyVisitor();
 Subject sub = new MySubject();
 sub.accept(visitor);
}

}
输出：visit the subject：love

该模式适用场景：如果我们想为一个现有的类增加新功能，不得不考虑几个事情：1、新功能会不会与现有功能出现兼容性问题？2、以后会不会再需要添加？3、如果类不允许修改代码怎么办？面对这些问题，最好的解决方法就是使用访问者模式，访问者模式适用于数据结构相对稳定的系统，把数据结构和算法解耦，22、中介者模式（Mediator） 中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。先看看图：

User类统一接口，User1和User2分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高，为了解耦，引入了Mediator类，提供统一接口，MyMediator为其实现类，里面持有User1和User2的实例，用来实现对User1和User2的控制。这样User1和User2两个对象相互独立，他们只需要保持好和Mediator之间的关系就行，剩下的全由MyMediator类来维护！基本实现：

java public interface Mediator { public void createMediator(); public void workAll(); }java public class MyMediator implements Mediator {

private User user1;
private User user2;

public User getUser1() {
 return user1;
}

public User getUser2() {
 return user2;
}

@Override
public void createMediator() {
 user1 = new User1(this);
 user2 = new User2(this);
}

@Override
public void workAll() {
 user1.work();
 user2.work();
}

}
```java public abstract class User {

private Mediator mediator;

public Mediator getMediator(){
 return mediator;
}

public User(Mediator mediator) {
 this.mediator = mediator;
}

public abstract void work();

}
```java public class User1 extends User {

public User1(Mediator mediator){
 super(mediator);
}

@Override
public void work() {
 System.out.println("user1 exe!");
}

}
```java public class User2 extends User {

public User2(Mediator mediator){
 super(mediator);
}

@Override
public void work() {
 System.out.println("user2 exe!");
}

}
测试类：

```java public class Test {

public static void main(String[] args) {
 Mediator mediator = new MyMediator();
 mediator.createMediator();
 mediator.workAll();
}

}
输出：

user1 exe! user2 exe! 23、解释器模式（Interpreter） 解释器模式是我们暂时的最后一讲，一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。

Context类是一个上下文环境类，Plus和Minus分别是用来计算的实现，代码如下：

java public interface Expression { public int interpret(Context context); }java public class Plus implements Expression {

@Override
public int interpret(Context context) {
 return context.getNum1()+context.getNum2();
}

}
```java public class Minus implements Expression {

@Override
public int interpret(Context context) {
 return context.getNum1()-context.getNum2();
}

}
```java public class Context {

private int num1;
private int num2;

public Context(int num1, int num2) {
 this.num1 = num1;
 this.num2 = num2;
}

public int getNum1() {
 return num1;
}
public void setNum1(int num1) {
 this.num1 = num1;
}
public int getNum2() {
 return num2;
}
public void setNum2(int num2) {
 this.num2 = num2;
}

}
```java public class Test {

public static void main(String[] args) {

 // 计算9+2-8的值
 int result = new Minus().interpret((new Context(new Plus()
 .interpret(new Context(9, 2)), 8)));
 System.out.println(result);
}

}
最后输出正确的结果：3。

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！ 设计模式基本就这么大概讲完了，总体感觉有点简略，的确，这么点儿篇幅，不足以对整个23种设计模式做全面的阐述，此处读者可将它作为一个理论基础去学习，通过这四篇博文，先基本有个概念，虽然我讲的有些简单，但基本都能说明问题及他们的特点，如果对哪一个感兴趣，可以继续深入研究！同时我也会不断更新，尽量补全遗漏、修正不足，欢迎广大读者及时提出好的建议，我们一起学习！项目中涉及到的代码，已经放到了我的资源里：http://download.csdn.net/detail/zhangerqing/4835830（因为我不喜欢不劳而获，所以没有免积分，只设置了5个，如果有人实在没积分又急要，那么联系我吧，我给你发过去）。50.谈谈你对框架的理解，设计框架的时候你是怎么考虑的，重构项目的时候你都遵循什么原则。

1.框架不要为应用做过多的假设 关于框架为应用做过多的假设， 一个非常具体的现象就是， 框架越俎代庖， 把本来是应 用要做的事情揽过来自己做。 这是一种典型的吃力不讨好的做法。 框架越俎代庖， 也许会使 得某一个具体应用的开发变得简单， 却会给其它更多想使用该框架的应用增加了本没有必要 的束缚和负担。 2.使用接口，保证框架提供的所有重要实现都是可以被替换的 框架终究不是应用， 所以框架无法考虑所有应用的具体情况， 保证所有重要的组件的实 现都是可以被替换的， 这一点非常重要， 它使得应用可以根据当前的实际情况来替换掉框架 提供的部分组件的默认实现。 使用接口来定义框架中各个组件及组件间的联系， 将提高框架 的可复用性。 3.框架应当简洁、一致、且目标集中 框架应当简洁， 不要包含那些对框架目标来说无关紧要的东西， 保证框架中的每个组件 的存在都是为了支持框架目标的实现。包含过多无谓的元素（类、接口、枚举等） ，会使框 架变得难以理解， 尝试将这些对于框架核心目标不太重要的元素转移到类库中， 可以使得框 架更清晰、目标更集中。 4.提供一个常用的骨架，但是不要固定骨架的结构，使骨架也是可以组装的 比如说， 如果是针对某种业务处理的框架， 那么框架不应该只提供一套不可变更的业务 处理流程，而是应该将处理流程 “单步” 化，使得各个步骤是可以重新组装的，如此一来，应 用便可以根据实际情况来改变框架默认的处理流程。 这种框架的可定制化能力可以极大地提 高框架的可复用性。 5.不断地重构框架 如果说设计和实现一个高质量的框架有什么秘诀？答案只有一个，重构、不断地重构。 重构框架的实现代码、 甚至重构框架的设计。 重构的驱动力源于几个方面， 比如对要解决的 本质问题有了更清晰准备的认识，在使用框架的时候发现某些组件职责不明确、难以使用， 框架的层次结构不够清晰等。51.LRU算法

假设 序列为 4 3 4 2 3 1 4 2 物理块有3个 则 首轮 4调入内存 4 次轮 3调入内存 3 4 之后 4调入内存 4 3 之后 2调入内存 2 4 3 之后 3调入内存 3 2 4 之后 1调入内存 1 3 2（因为最少使用的是4，所以丢弃4） 之后 4调入内存 4 1 3（原理同上） 最后 2调入内存 2 4 1 在指定内存中如果超过内存剔除最近最少用的。

## 51.自定义控件的生命周期

|  |  |
| --- | --- |
| 方法声明 | 功能描述 |
| onFinishInflate() | 当View中所有的子控件均被映射成xml后触发 |
| onMeasure( int , int ) | 确定所有子元素的大小 |
| onLayout( boolean , int , int , int , int ) | 当View分配所有的子元素的大小和位置时触发 |
| onSizeChanged( int , int , int , int ) | 当view的大小发生变化时触发 |
| onDraw(Canvas) | view渲染内容的细节 |
| onKeyDown( int , KeyEvent) | 有按键按下后触发 |
| onKeyUp( int , KeyEvent) | 有按键按下后弹起时触发 |
| onTrackballEvent(MotionEvent) | 轨迹球事件 |
| onTouchEvent(MotionEvent) | 触屏事件 |
| onFocusChanged( boolean , int , Rect) | 当View获取或失去焦点时触发 |
| onWindowFocusChanged( boolean ) | 当窗口包含的view获取或失去焦点时触发 |
| onAttachedToWindow() | 当view被附着到一个窗口时触发 |
| onDetachedFromWindow() | 当view离开附着的窗口时触发，Android123提示该方法和 |
| onWindowVisibilityChanged( int ) | 当窗口中包含的可见的view发生变化时触发 |