**Android学习笔记**

# UI相关

## 四种Activity启动模式

* standard 默认
* singleTop 可以多实例、栈顶复用
* singleTask 单例
* singleInstance 单例、独立的Task

## 常用Drawable

ColorDrawable、GradientDrawable(渐变)、NinePatchDrawable(.9)、

ClipDrawable(剪切)、ShapeDrawable、LayerDrawable(一组图片)

## 事件分发机制

1. dispatchTouchEvent事件分发：Activity、ViewGroup、View都有，以隧道方式从根节点依次往下传递直到最内层子节点。
2. onInterceptTouchEvent事件拦截：只有ViewGroup有，复写这个方法可以拦截子View的事件，把事件交给自己处理，执行自已的onTouchEvent方法。
3. onTouchEvent事件响应：Activity、ViewGroup、View都有，ACTION\_DOWN、ACTION\_MOVE(0次或多次)、ACTION\_UP三种类型返回true，表明消费了该事件，事件到此结束。返回false，没有消费该事件，会向上层传递。

## 自定义控件

1. 组合控件：将若干个系统控件组合成一个新的控件，便于复用。
2. 自绘控件：复写onDraw方法，通过画布Canvas和画笔Paint绘制需要的样式。
3. 继承控件：继承已有的控件，创建新控件，保留父控件的特性，并且还可以引入新特性。

三个核心方法

* onMeasure()测量，决定View的大小。
* onLayout()布局，控件子控件的位置
* onDraw()如果绘制这个View

调用invalidate()会执行onDraw方法，调用requestLayout()会执行onMeasure和onLayout，不一定会调用onDraw方法。

## 多分辨率适配

1. 尽量使用相对布局和线性布局权值。
2. 间距和间隔使用dp为单位
3. 针对不同的dpi切多套图片资源。
4. 使用.9图替代普通的png图片。
5. 必要时，可在代码中动态计算和设置控件大小。

## 动画

### View Animation（视图动画）

* 补间动画（Tween Animation）

Translate、scale、rotate、alpha

* 帧动画（Frame Animation）

视图动画只能作用于View，而且视图动画改变的只是View的绘制效果，View真正的属性并没有改变。比如，一个按钮做平移的动画，虽然按钮的确做了平移，但按钮可点击的区域并没随着平移而改变，还是在原来的位置。

### Property Animation（属性动画）

动画的对象除了传统的View对象，还可以是Object对象，动画之后，Object对象的属性值被实实在在的改变了。Property animation能够通过改变View对象的实际属性来实现View动画。

**转场动画：**

* overridePendingTransition
* 5.0新特性ActivityOptionsCompat

**分解（explode）**：从场景中心移入或移出视图。

**滑动（slide）**：从场景边缘移入或移出视图。

**淡入淡出（fade）**：通过调整透明度在场景中增添或移除视图。

## Fragment

### java.lang.IllegalStateException: Fragment already active

在 Fragment 没有被添加到 FragmentManager 之前，我们可以通过 Fragment.setArguments() 来设置参数，并在 Fragment 中，使用 getArguments() 来取得参数。在 Fragment 被添加到 FragmentManager 后，一旦被使用，我们再次调用 setArguments() 将会导致 java.lang.IllegalStateException: Fragment already active 异常。

# 消息机制

## Message消息机制

Message消息、MessageQueue消息队列、Looper消息循环，创建消息队列，并不断从队列中读取消息、Handler消息处理。

## 广播机制

* 普通广播：两种注册形式
* 系统广播：由系统自动发出
* 有序广播：优先级高的先收到，并可以对广播进行修改，还可以传参给后面的广播

## 跨进程通讯

由于android系统中应用程序之间不能共享内存。因此，在不同应用程序之间交互数据（跨进程通讯）就稍微麻烦一些。在android SDK中提供了4种用于跨进程通讯的方式。这4种方式正好对应于android系统中4种应用程序组件：Activity、Content Provider、Broadcast和Service。其中Activity可以跨进程调用其他应用程序的Activity；Content Provider可以跨进程访问其他应用程序中的数据（以Cursor对象形式返回），当然，也可以对其他应用程序的数据进行增、删、改操 作；Broadcast可以向android系统中所有应用程序发送广播，而需要跨进程通讯的应用程序可以监听这些广播；Service和Content Provider类似，也可以访问其他应用程序中的数据，但不同的是，Content Provider返回的是Cursor对象，而Service返回的是Java对象，这种可以跨进程通讯的服务叫AIDL服务。

### 使用Intent方式进行数据传递

通过Intent来启动访问其他应用程序的Activity，Activity的跨进程访问与进程内访问略有不同。虽然它们都需要Intent对象，但跨进程访问并不需要指定Context对象和Activity的Class对象，而需要指定的是要访问的Activity所对应的Action（一个字符串）。有些Activity还需要指定一个Uri（通过Intent构造方法的第2个参数指定）。

### ContentProvider

ContentProvider数据共享机制：为数据读写提供统一的接口，可以在不同的应用程序之间共享数据。

### 广播（Broadcast）

广播是一种被动跨进程通讯的方式。当某个程序向系统发送广播时，其他的应用程序只能被动地接收广播数据。这就象电台进行广播一样，听众只能被动地收听，而不能主动与电台进行沟通。

### AIDL服务

## Binder

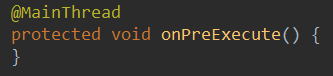
1. Binder架构：Binder是采用Client/Server架构，上层应用作为Client端向Android系统(系统作为Server端)请求服务，这个请求过程通过Binder IPC来完成的；
2. Binder选择理由：Binder是从性能、稳定性、安全性、语言层面4个角度综合考虑的结果；
3. Service与Binder的联系：Android OS中的Service几乎都是基于Binder通信的。例如：ActivityManagerService(用于控制Activity、Service、进程等) 这个服务作为Server端，间接继承Binder类，而相应的ActivityManager作为Client端，间接继承于BinderProxy类。

# 多线程

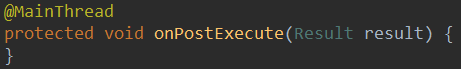
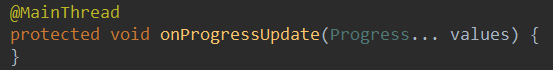
## AsyncTask

对Thread和Handler进行了封装，有四个重要的回调方法，分别是：onPreExecute、doInBackground, onProgressUpdate 和 onPostExecute。这四个方法会在AsyncTask的不同时期进行自动调用，我们只需要实现这几个方法的内部逻辑即可。这四个方法的一些参数和返回值都是基于泛型的，而且泛型的类型还不一样，所以在AsyncTask的使用中会遇到三种泛型参数：Params, Progress 和 Result。

这里写图片描述



这里写图片描述



* AsyncTask的实例必须在主线程中创建。
* AsyncTask的execute方法必须在主线程中调用。
* onPreExecute()、onPostExecute(Result),、doInBackground(Params…) 和 onProgressUpdate(Progress…)这四个方法都是回调方法，Android会自动调用，我们不应自己调用。
* 对于一个AsyncTack的实例，只能执行一次execute方法，在该实例上第二次执行execute方法时就会抛出异常。

# 三方服务

# 设计模式

## 单例模式

## 建造者模式

建造模式是将复杂的内部创建封装在内部，对于外部调用的人来说，只需要传入建造者和建造工具，对于内部是如何建造成成品的，调用者无需关心。

示例：AlertDialog.Builder

## 观察者模式

OnClickListener、广播机制等

## 工厂模式

生产固定的一些东西，如BitmapFactory。

## 适配器模式

Adapter

## 命令模式

Handler.post

## 责任链模式

使多个对象都有机会处理请求，从而避免请求的发送者和接受者直接的耦合关系，将这些对象连成一条链，并沿这条链传递该请求，直到有对象处理它为止。如事件分发机制。

## 装饰模式

包装模式（Wrapper），ContextWrapper

# 其他

## 内存泄漏

### 单例造成的内存泄漏

这是一个普通的单例模式，当创建这个单例的时候，需要传入一个Context，如果传入的是Activity的Context，就会引发内存泄漏。

### Handler造成的内存泄漏

由于mHandler是Handler的非静态匿名内部类的实例，所以它持有外部类Activity的引用，我们知道消息队列是在一个Looper线程中不断轮询处理消息，那么当这个Activity退出时消息队列中还有未处理的消息或者正在处理消息，而消息队列中的Message持有mHandler实例的引用，mHandler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏。

### 资源未关闭造成的内存泄漏

对于使用了BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

## Java堆&栈

* **栈：**

函数中定义的基本类型变量，对象的引用变量都在函数的栈内存中分配。  
栈内存特点，数数据一执行完毕，变量会立即释放，节约内存空间。  
栈内存中的数据，没有默认初始化值，需要手动设置。

* **堆：**

堆内存用来存放new创建的对象和数组。  
堆内存中所有的实体都有内存地址值。  
堆内存中的实体是用来封装数据的，这些数据都有默认初始化值。  
堆内存中的实体不再被指向时，JVM启动垃圾回收机制，自动清除，这也是JAVA优于C++的表现之一（C++中需要程序员手动清除）。

## Thread Runnable

* Thread是类，而Runnable是接口。Thread内部也是对Runnable进行封装。
* Runnable可以避免由于java的单继承机制带来的局限。可以再继承其他类的同时，还能实现多线程的功能。
* Runnable能增加程序的健壮性。代码能够被多个线程共享。
* Runnable适合于多个相同程序代码线程去处理统一资源的情况，把虚拟的cpu（线程）同程序的代码，数据有效分离，较好体现面向对象的编程的思想。

## 面向过程&面向对象

* 面向过程”是一种是事件为中心的编程思想。就是分析出解决问题所需的步骤，然后用函数把这写步骤实现，并按顺序调用。
* 面向对象是把一组数据结构和处理它们的方法组成对象（object），把相同行为的对象归纳为类（class），通过类的封装（encapsulation）隐藏内部细节，通过继承（inheritance）实现类的特化（specialization）／泛化（generalization），通过多态（polymorphism）实现基于对象类型的动态分派（dynamic dispatch）。