目錄

Android开发艺术探索——读书笔记

本书是一本Android进阶类书籍，采用理论、源码和实践相结合的方式来阐述高水准的Android应用开发要点。本书从三个方面来组织内容。

1. 介绍Android开发者不容易掌握的一些知识点
2. 结合Android源代码和应用层开发过程，融会贯通，介绍一些比较深入的知识点
3. 介绍一些核心技术和Android的性能优化思想

Activity的生命周期和启动模式

1.1 Activity的生命周期全面分析

用户正常使用情况下的生命周期 & 由于Activity被系统回收或者设备配置改变导致Activity被销毁重建情况下的生命周期。

1.1.1 典型情况下的生命周期分析

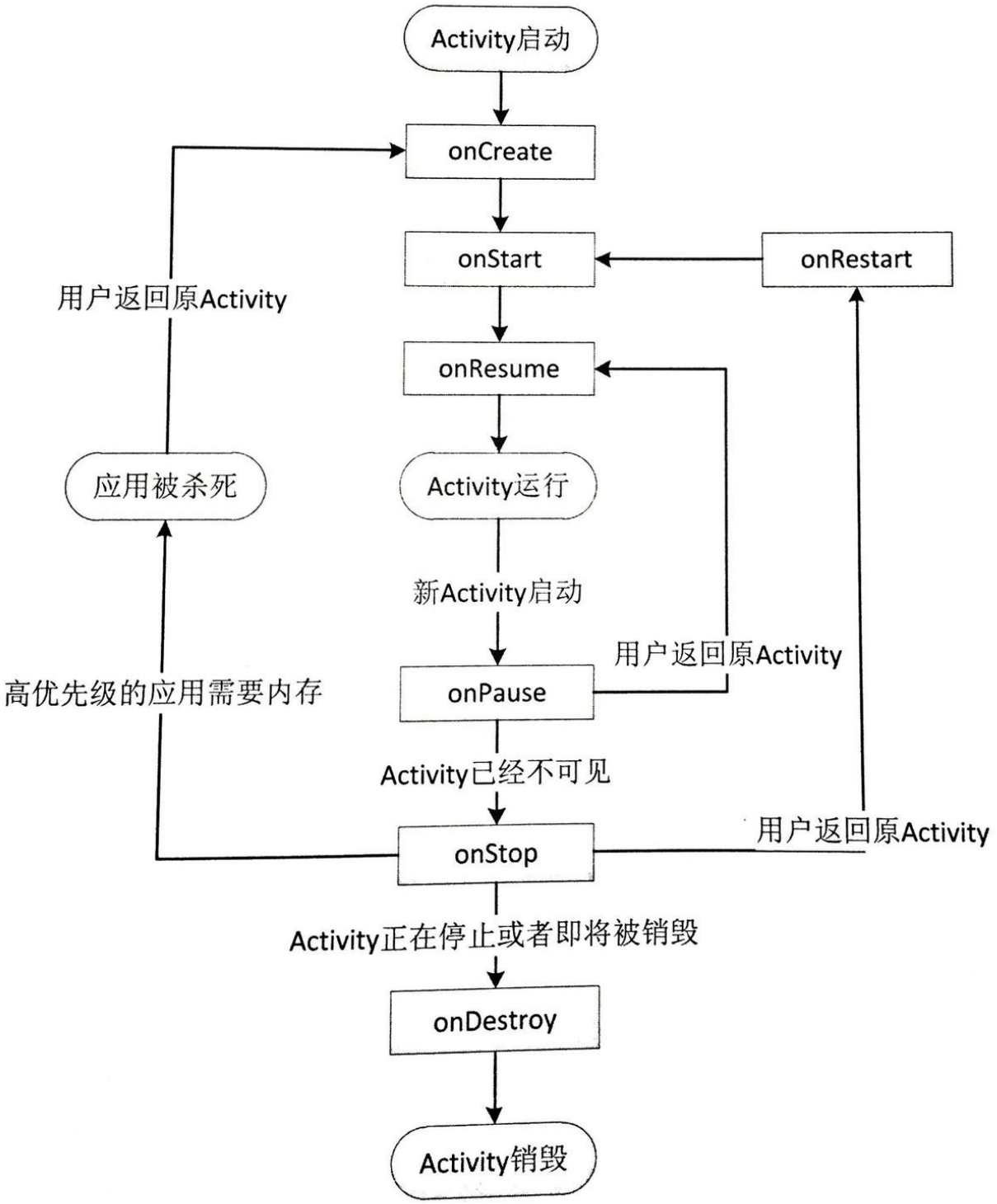


图 1-1　Activity 生命周期的切换过程

1. Activity第一次启动：**onCreate->onStart->onResume**。
2. Activity切换到后台（用户打开新的Activity或者切换到桌面），**onPause->onStop**。
3. Activity从后台到前台，重新可见，**onRestart->onStart->onResume**。
4. 用户退出Activity，**onPause->onStop->onDestroy**。
5. onStart开始到onStop之前，Activity可见。onResume到onPause之前，Activity可以接受用户交互。

6. 在新Activity启动之前，栈顶的Activity需要先onPause后，新Activity才能启动。所以不能在**onPause**执行耗时操作。

1.1.2 异常情况下的生命周期分析

1 系统配置变化导致Activity销毁重建

例如Activity处于竖屏状态，如果突然旋转屏幕，由于系统配置发生了改变，Activity就会被销毁并重新创建。

- 在异常情况下系统会在**onStop**之前调用**onSaveInstanceState**来保存状态。Activity重新创建后，会在**onStart**之后调用**onRestoreInstanceState**来恢复之前保存的数据。
- 保存数据的流程： Activity被意外终止，调用onSaveIntanceState保存数据-> Activity委托Window，Window委托它上面的顶级容器一个ViewGroup（书上说很可能就是DecorView）。然后顶层容器在通知所有子元素来保存数据。 每个View都有 `onSaveInstanceState` 和 `onRestoreInstanceState` 方法。查看 `TextView` 源码可以发现保存了文本选中状态和文本内容。
- 系统只在**Activity**异常终止的时候才会调用 **`onSaveInstanceState`** 和 **`onRestoreInstanceState`** 方法。其他情况不会触发。

2 资源内存不足导致低优先级的Activity被回收

1. 前台- 可见非前台（被对话框遮挡的Activity）-后台，这三种Activity优先级从高到低。
2. `android:configChanges="orientation"` 在manifest中指定 `configChanges` 在系统配置变化后不重新创建Activity，也不会执行 **`onSaveInstanceState`** 和 **`onRestoreInstanceState`** 方法，而是调用 **`onConfigurationChnaged`** 方法。
3. `configChanges` 一般常用三个选项：
   - locale 系统语言变化
   - keyborardHidden 键盘的可访问性发生了变化，比如用户调出了键盘
   - orientation 屏幕方向变化

1.2 Activity的启动模式

1.2.1 Activity的LaunchMode

Android使用栈来管理Activity。

1 standard

- 每次启动都会重新创建一个实例，不管这个Activity在栈中是否已经存在。

- 谁启动了这个Activity，那么Activity就运行在启动它的那个Activity所在的栈中。
- 用Application去启动Activity时会报错，提示非Activity的Context没有所谓的任务栈。解决办法是为待启动Activity制定**FLAG_ACTIVITY_NEW_TASH**标志位，这样就会为它创建一个新的任务栈。

2 singleTop

- 如果新Activity位于任务栈的栈顶，那么此Activity不会被重新创建，同时回调 `onNewIntent` 方法。
- 如果新Activity已经存在但不是位于栈顶，那么新Activity仍然会被创建。

3 singleTask

- 这是一种单实例模式
- 只要Activity在栈中存在，那么多次启动这个Activity都不会重新创建实例，同时也会回调 `onNewIntent` 方法。
- 同时会导致在Activity之上的栈内Activity出栈。

4 singleIntance

- 具有singleTask模式的所有特性，同时具有此模式的Activity只能单独的位于一个任务栈中

TaskAffinity属性

TaskAffinity参数标识了一个Activity所需要的任务栈的名字。为字符串，且中间必须包含包名分隔符“.”。默认情况下，所有**Activity**所需的任务栈名字为应用包名。TashAffinity属性主要和singleTask启动模式或者 `allowTaskReparenting` 属性配对使用，其他情况下没有意义。 应用A启动了应用B的某个Activity后，如果Activity的allowTaskReparenting属性为true的话，那么当应用B被启动后，此Activity会直接从应用A的任务栈转移到应用B的任务栈中。 打个比方就是，应用A启动了应用B的ActivityX，然后按Home回到桌面，单击应用B的图标，这时并不会启动B的主Activity，而是重新显示已经被应用A启动的ActivityX。这是因为ActivityX的TaskAffinity值肯定不和应用A的任务栈相同（因为包名不同）。所以当应用B被启动以后，发现ActivityX原本所需的任务栈已经被创建了，所以把ActivityX从A的任务栈中转移过来了。

设置启动模式

1. manifest中 设置下的 `android:launchMode` 属性。
2. 启动Activity的 `intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);` 。
3. 两种同时存在时，以第二种为准。第一种方式无法直接为Activity添加FLAG_ACTIVITY_CLEAR_TOP标识，第二种方式无法指定singleInstance模式。
4. 可以通过命令行 `adb shell dumpsys activity` 命令查看栈中的Activity信息。

1.2.2 Activity的Flags

这些FLAG可以设定启动模式、可以影响Activity的运行状态。

- FLAG_ACTIVITY_CLEAR_TOP 具有此标记位的Activity启动时，同一个任务栈中位于它上面的Activity都要出栈，一般和FLAG_ACTIVITY_NEW_TASK配合使用。效果和singleTask一样。
- FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS 如果设置，新的Activity不会在最近启动的Activity的列表(就是安卓手机里显示最近打开的Activity那个系统级的UI)中保存。

1.3 IntentFilter的匹配规则

启动Activity分为两种：

1. 显示调用 明确指定被启动对象的组件信息，包括包名和类名
2. 隐式调用 不需要明确指定组件信息，需要Intent能够匹配目标组件中的IntentFilter中所设置的过滤信息。
3. IntentFilter中的过滤信息有action、category、data。
4. 只有一个Intent同时匹配action类别、category类别、data类别才能成功启动目标Activity。
5. 一个Activity可以有多个intent-filter，一个Intent只要能匹配任何一组intent-filter即可成功启动对应的Activity。

1 action

1. action是一个字符串。
2. 一个intent-filter可以有多个aciton，只要Intent中的action能够和任何一个action相同即可成功匹配。匹配是指与action的字符串完全一样。
3. Intent中如果没有指定action，那么匹配失败。

2 category

1. category是一个字符串。
2. Intent可以没有category，但是如果你一旦有category，不管有几个，每个都能够与intent-filter中的其中一个category相同。
3. 系统在 `startActivity` 和 `startActivityForResult` 的时候，会默认为Intent加上 `android.intent.category.DEFAULT` 这个category，所以为了我们的activity能够接收隐式调用，就必须在intent-filter中加上 `android.intent.category.DEFAULT` 这个category。

3 data

1. data的匹配规则与action一样，如果intent-filter中定义了data，那么Intent中必须要定义可匹配的data。
2. intent-filter中data的语法：

```
<data android:scheme="string"
        android:host="string"
        android:port="string"
        android:path="string"
        android:pathPattern="string"
        android:pathPrefix="string"
        android:mimeType="string"/>
```

3. Intent中的data有两部分组成：**mimeType**和**URI**。mimeType是指媒体类型，比如image/jpeg、audio/mpeg4-generic和video/等，可以表示图片、文本、视频等不同的媒体格式。
   - URI的结构：`<scheme>://<host>:<port>/[<path>|<pathPrefix>|<pathPattern>]`

     ```
     //实际例子
     content://com.example.project:200/folder/subfolder/etc
     http://www.baidu.com:80/search/info
     ```

     - scheme：URI的模式，比如http、file、content等，默认值是 **`file`** 。
     - host：URI的主机名
     - port：URI的端口号
     - path、pathPattern和pathPrefix：这三个参数描述路径信息。
       - path、pathPattern可以表示完整的路径信息，其中pathPattern可以包含通配符 `*` ，表示0个或者多个任意字符。
       - pathPrefix只表示路径的前缀信息。
   - Intent指定data时，必须调用 `setDataAndType` 方法， `setData` 和 `setType` 会清除另一方的值。

隐式调用需注意

1. 当通过隐式调用启动Activity时，没找到对应的Activity系统就会抛出 `android.content.ActivityNotFoundException` 异常，所以需要判断是否有Activity能够匹配我们的隐式Intent。
   i. 采用 `PackageManager` 的 `resloveActivity` 方法

```
 public abstract List<ResolveInfo> queryIntentActivityies(Intent intent,int f
lags);
 public abstract ResolveInfo resloveActivity(Intent intent,int flags);
```

以上的第二个参数使用 `MATCH_DEFAULT_ONLY` ，这个标志位的含义是仅仅匹配那些在intent-filter中声明了 `android.intent.category.DEFAULT` 这个category的Activity。因为如果把不含这个category的Activity匹配出来了，由于不含DEFAULT这个category的Activity是无法接受隐式Intent的从而导致startActivity失败。

ii. 采用 `Intent` 的 `resloveActivity` 方法

2. 下面的action和category用来表明这是一个入口Activity并且会出现在系统的应用列表中，二者缺一不可。

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

2 IPC机制

2.1 Android IPC 简介

1. IPC即Inter-Process Communication，含义为进程间通信或者跨进程通信，是指两个进程之间进行数据交换的过程。
2. 线程是CPU调度的最小单元，是一种有限的系统资源。进程一般指一个执行单元，在PC和移动设备上是指一个程序或者应用。进程与线程是包含与被包含的关系。一个进程可以包含多个线程。最简单的情况下一个进程只有一个线程，即主线程（例如Android的UI线程）。
3. 任何操作系统都需要有相应的IPC机制。
4. 在Android中，IPC的使用场景大概有以下：
    i. 有些模块由于特殊原因需要运行在单独的进程中。
    ii. 通过多进程来获取多份内存空间。
    iii. 当前应用需要向其他应用获取数据。

2.2 Android中的多进程模式

2.2.1 开启多进程模式

给四大组件在**Manifest**中指定 `android:process` 属性。这个属性的值就是进程名。

> tips：使用 `adb shell ps` 或 `adb shell ps|grep 包名` 查看当前所存在的进程信息。

2.2.2 多线程模式的运行机制

Android为每个进程都分配了一个独立的虚拟机，不同虚拟机在内存分配上有不同的地址空间，导致不同的虚拟机访问同一个类的对象会产生多份副本。例如不同进程的Activity对静态变量的修改，对其他进程不会造成任何影响。
所有运行在不同进程的四大组件，只要它们之间需要通过内存在共享数据，都会共享失败。
四大组件之间不可能不通过中间层来共享数据。

多进程会带来以下问题：

1. 静态成员和单例模式完全失效。
2. 线程同步锁机制完全失效。
   这两点都是因为不同进程不在同一个内存空间下，锁的对象也不是同一个对象。
3. SharedPreferences的可靠性下降。

SharedPreferences底层是 通过读/写XML文件实现的，并发读/写会导致一定几率的数据丢失。

4. Application会多次创建。
   由于系统创建新的进程的同时分配独立虚拟机，其实这就是启动一个应用的过程。

在多进程模式中，不同进程的组件拥有独立的虚拟机、**Application**以及内存空间。
实现跨进程的方式有很多：

1. Intent传递数据。
2. 共享文件和SharedPreferences。
3. 基于Binder的Messenger和AIDL。
4. Socket。

2.3 IPC基础概念介绍

主要介绍 `Serializable` 、 `Parcelable` 、 `Binder` 。

2.3.1 Serializable接口

1. `Serializable` 是Java提供的一个序列化接口（空接口），为对象提供标准的序列化和反序列化操作。
2. 只需要一个类去实现 `Serializable` 接口并声明一个 `serialVersionUID` 即可实现序列化。
3. 如果不手动指定 `serialVersionUID` 的值，反序列化时当前类有所改变（比如增删了某些成员变量），那么系统就会重新计算当前类的hash值并赋值给 `serialVersionUID` 。这个时候当前类的 `serialVersionUID` 就和序列化数据中的 `serialVersionUID` 不一致，导致反序列化失败，程序就出现crash。
4. 静态成员变量属于类不属于对象，不参与序列化过程，其次 `transient` 关键字标记的成员变量不参与序列化过程。

2.3.2 Parcelable接口

1. `Parcelable` 内部包装了可序列化的数据。
2. 序列化功能由 `writeToParcel` 方法完成,最终是通过 `Parcel` 的一系列writer方法来完成。

```
@Override
public void writeToParcel(Parcel out, int flags) {
    out.writeInt(code);
    out.writeString(name);
}
```

3. 反序列化功能由 `CREATOR` 来完成，其内部表明了如何创建序列化对象和数组，通过 `Parcel` 的一系列read方法来完成。

```
public static final Creator<Book> CREATOR = new Creator<Book>() {
    @Override
    public Book createFromParcel(Parcel in) {
        return new Book(in);
    }
    @Override
    public Book[] newArray(int size) {
        return new Book[size];
    }
};

 protected Book(Parcel in) {
    code = in.readInt();
  name = in.readString();
}
```

4. 内容描述功能由 `describeContents` 方法完成，几乎所有情况下都应该返回0，仅当当前对象中存在文件描述符时返回1。

```
public int describeContents() {
        return 0;
}
```

5. `Serializable` 是Java的序列化接口，使用简单但开销大，序列化和反序列化过程需要大量I/O操作。而 `Parcelable` 是Android中的序列化方式，适合在Android平台使用，效率高但是使用麻烦。 **`Parcelable`** 主要在内存序列化上， **`Parcelable`** 也可以将对象序列化到存储设备中或者将对象序列化后通过网络传输，但是稍显复杂，推荐使用 **`Serializable`** 。

2.3.3 Binder

1. Binder是Android中的一个类，实现了 `IBinder` 接口。从IPC角度说，Binder是Andoird的一种跨进程通讯方式。从**Android Framework**角度来说，Binder是 `ServiceManager` 连接各种Manager(`ActivityManager`、 `WindowManager`)和相应 `ManagerService` 的桥梁。从Android应用层来说，Binder是客户端和服务端进行通信的媒介，当bindService时，服务端返回一个包含服务端业务调用的Binder对象，通过这个Binder对象，客户端就可以获取服务器端提供的服务或者数据（包括普通服务和基于AIDL的服务）。
2. Android中Binder主要用于 `Service` ，包括AIDL和Messenger。普通Service的Binder不涉及进程间通信，Messenger的底层其实是AIDL，所以下面通过AIDL分析Binder的工作机制。

2.3.3.1 由系统根据AIDL文件自动生成.java文件

1. Book.java
   表示图书信息的实体类，实现了Parcelable接口。
2. Book.aidl
   Book类在AIDL中的声明。

```
package com.ryg.chapter_2.aidl;
parcelable Book;
```

3. IBookManager.aidl
   定义的管理Book实体的一个接口，包含 `getBookList` 和 `addBook` 两个方法。

系统为IBookManager.aidl生产的Binder类，在 `gen` 目录下的**IBookManager.java**类。

IBookManager继承了 `IInterface` 接口，所有在**Binder**中传输的接口都需要继承**IInterface**接口。结构如下：

1. 声明了 `getBookList` 和 `addBook` 方法，还声明了两个整型id分别标识这两个方法，用于标识在 `transact` 过程中客户端请求的到底是哪个方法。
2. 声明了一个内部类 `Stub` ，这个 `Stub` 就是一个Binder类，当客户端和服务端位于同一进程时，方法调用不会走跨进程的 `transact` 。当二者位于不同进程时，方法调用需要走 `transact` 过程，这个逻辑有 `Stub` 的内部代理类 `Proxy` 来完成。
3. 这个接口的核心实现就是它的内部类 `Stub` 和 `Stub` 的内部代理类 `Proxy` 。

2.3.3.2 Stub和Proxy类的内部方法和定义

1. **DESCRIPTOR**
   Binder的唯一标识，一般用Binder的类名表示。
2. **asInterface(android.os.IBinder obj)**
   将服务端的Binder对象转换为客户端所需的AIDL接口类型的对象，如果C/S位于同一进程，此方法返回就是服务端的Stub对象本身，否则返回的就是系统封装后的Stub.proxy对象。
3. **asBinder**
   返回当前Binder对象。
4. **onTransact**
   这个方法运行在服务端的Binder线程池中，由客户端发起跨进程请求时，远程请求会通过系统底层封装后交由此方法来处理。该方法的原型是
   `java public Boolean onTransact(int code,Parcelable data,Parcelable reply,int flags)`
   i. 服务端通过code确定客户端请求的目标方法是什么，
   ii. 接着从data取出目标方法所需的参数，然后执行目标方法。
   iii. 执行完毕后向reply写入返回值（如果有返回值）。
   iv. 如果这个方法返回值为false，那么服务端的请求会失败，利用这个特性我们可以来做权限验证。

5. **Proxy#getBookList 和Proxy#addBook**
   i. 这个方法运行在客户端，首先该方法所需要的输入型对象Parcel对象_data，输出型Parcel对象_reply和返回值对象List。
   ii. 然后把该方法的参数信息写入_data（如果有参数），3
   iii. 接着调用transact方法发起**RPC**（远程过程调用），同时当前线程挂起，
   iv. 然后服务端的onTransact方法会被调用知道RPC过程返回后，当前线程继续执行，并从_reply中取出RPC过程的返回结果，最后返回_reply中的数据。

之所以提供AIDL文件，是为了方便系统为我们生成代码，我们完全可以自己实现Binder。

2.3.3.3 可以给Binder设置一个死亡代理，当Binder死亡时，我们就会收到通知。

1. 声明一个 `DeathRecipient` 对象。 `DeathRecipient` 只有一个方法 `binderDied` ，当Binder死亡的时候，系统就会回调 `DeathRecipient` 方法。

```
private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient(){
    @Override
    public void binderDied(){
        if(mBookManager == null){
            return;
        }
        mBookManager.asBinder().unlinkToDeath(mDeathRecipient,0);
        mBookManager = null;
        // TODO：接下来重新绑定远程Service
    }
}
```

2. Binder有两个很重要的方法 `linkToDeath` 和 `unlinkToDeath` 。通过 `linkToDeath` 为Binder设置一个死亡代理。

```
mService = IBookManager.Stub.asInterface(binder);
binder.linkToDeath(mDeathRecipient,0);
```

3. 另外，可以通过Binder的 `isBinderAlive` 判断Binder是否死亡。

2.4 Android中的IPC方式

主要有以下方式：

1. Intent中附加extras来传递消息
2. 共享文件
3. Binder方式

4. 四大组件之一的ContentProvider
5. Socket

2.4.1 使用Bundle

四大组件中的三大组件（Activity、Service、Receiver）都支持在Intent中传递 `Bundle` 数据。Bundle实现了Parcelable接口，**当我们在一个进程中启动了另一个进程的Activity、Service、Receiver，可以再Bundle中附加我们需要传输给远程进程的消息并通过Intent发送出去。被传输的数据必须能够被序列化。

2.4.2 使用文件共享

一些概念：

1. 两个进程通过读写同一个文件来交换数据。还可以通过 `ObjectOutputStream` / `ObjectInputStream` 序列化一个对象到文件中，或者在另一个进程从文件中反序列这个对象。注意：反序列化得到的对象只是内容上和序列化之前的对象一样，本质是两个对象。
2. 文件并发读写会导致读出的对象可能不是最新的，并发写的话那就更严重了（书本原文，个人理解这里的严重应该是指数据丢失。）。所以文件共享方式适合对数据同步要求不高的进程之间进行通信，并且要妥善处理并发读写问题。
3. `SharedPreferences` 底层实现采用XML文件来存储键值对。系统对它的读/写有一定的缓存策略，即在内存中会有一份 `SharedPreferences` 文件的缓存，因此在多进程模式下，系统对它的读/写变得不可靠，面对高并发读/写时 `SharedPreferences` 有很大几率丢失数据，因此不建议在**IPC**中使用 **`SharedPreferences`** 。

2.4.3 使用Messenger

Messenger可以在不同进程间传递Message对象。是一种轻量级的IPC方案，底层实现是AIDL。
具体使用时，分为服务端和客户端：

1. 服务端：创建一个Service来处理客户端请求，同时创建一个Handler并通过它来创建一个Messenger，然后再Service的onBind中返回Messenger对象底层的Binder即可。

```
private final Messenger mMessenger = new Messenger (new xxxHandler());
```

2. 客户端：绑定服务端的Sevice，利用服务端返回的IBinder对象来创建一个Messenger，通过这个Messenger就可以向服务端发送消息了，消息类型是 `Message` 。如果需要服务端响应，则需要创建一个Handler并通过它来创建一个Messenger（和服务端一样），并通过 `Message` 的 `replyTo` 参数传递给服务端。服务端通过**Message**的 **`replyTo`** 参数就可

以回应客户端了。

3. 总而言之，就是客户端和服务端 拿到对方的**Messenger**来发送 `Message` 。只不过客户端通过 `bindService` 而服务端通过 `message.replyTo` 来获得对方的Messenger。
4. Messenger中有一个 `Hanlder` 以串行的方式处理队列中的消息。不存在并发执行，因此我们不用考虑线程同步的问题。

2.4.4 使用AIDL

如果有大量的并发请求，使用Messenger就不太适合，同时如果需要跨进程调用服务端的方法，Messenger就无法做到了。这时我们可以使用AIDL。
流程如下：

1. 服务端需要创建Service来监听客户端请求，然后创建一个AIDL文件，将暴露给客户端的接口在AIDL文件中声明，最后在Service中实现这个AIDL接口即可。
2. 客户端首先绑定服务端的Service，绑定成功后，将服务端返回的Binder对象转成AIDL接口所属的类型，接着就可以调用AIDL中的方法了。

注意事项：

1. AIDL支持的数据类型：
   i. 基本数据类型、String、CharSequence
   ii. List：只支持ArrayList，里面的每个元素必须被AIDL支持
   iii. Map：只支持HashMap，里面的每个元素必须被AIDL支持
   iv. Parcelable
   v. 所有的AIDL接口本身也可以在AIDL文件中使用
2. 自定义的Parcelable对象和AIDL对象，不管它们与当前的AIDL文件是否位于同一个包，都必须显式import进来。
3. 如果AIDL文件中使用了自定义的Parcelable对象，就必须新建一个和它同名的AIDL文件，并在其中声明它为Parcelable类型。

```
package com.ryg.chapter_2.aidl;

parcelable Book;
```

4. AIDL接口中的参数除了基本类型以外都必须表明方向in/out。AIDL接口文件中只支持方法，不支持声明静态常量。建议把所有和AIDL相关的类和文件放在同一个包中，方便管理。

```
void addBook(in Book book);
```

5. AIDL方法是在服务端的**Binder**线程池中执行的，因此当多个客户端同时连接时，管理数据的集合直接采用 `CopyOnWriteArrayList` 来进行自动线程同步。类似的还

有 `ConcurrentHashMap` 。

6. 因为客户端的listener和服务端的listener不是同一个对象，所以 `RecmoteCallbackList` 是系统专门提供用于删除跨进程listener的接口，支持管理任意的AIDL接口，因为所有AIDL接口都继承自 `IInterface` 接口。

   ```
   public class RemoteCallbackList<E extends IInterface>
   ```

   它内部通过一个Map接口来保存所有的AIDL回调，这个Map的key是 `IBinder` 类型，value是 `Callback` 类型。当客户端解除注册时，遍历服务端所有listener，找到和客户端listener具有相同Binder对象的服务端listenr并把它删掉。

7. 客户端RPC的时候线程会被挂起，由于被调用的方法运行在服务端的Binder线程池中，可能很耗时，不能在主线程中去调用服务端的方法。

2.4.5 使用ContentProvider

1. ContentProvider是四大组件之一，其底层实现和Messenger一样是Binder。ContentProvider天生就是用来进程间通信，只需要实现一个自定义或者系统预设置的ContentProvider，通过ContentResolver的query、update、insert和delete方法即可。
2. 创建ContentProvider，只需继承ContentProvider实现 `onCreate` 、 `query` 、 `update` 、 `insert` 、 `getType` 六个抽象方法即可。除了 `onCreate` 由系统回调并运行在主线程，其他五个方法都由外界调用并运行在Binder线程池中。

2.4.6 使用Socket

Socket可以实现计算机网络中的两个进程间的通信，当然也可以在本地实现进程间的通信。服务端Service监听本地端口，客户端连接指定的端口，建立连接成功后，拿到 `Socket` 对象就可以向服务端发送消息或者接受服务端发送的消息。

以上集中**IPC**方式都是感性的总结，具体代码请参考这里。

2.5 Binder连接池

AIDL是一种最常用的**IPC**方式，是日常开发中涉及**IPC**时的首选。前面提到AIDL的流程是 客户端在Service的onBind方法中拿到继承AIDL的Stub对象，然后客户端就可以通过这个Stub对象进行RPC。
那么如果项目庞大，有多个业务模块都需要使用AIDL进行IPC，随着AIDL数量的增加，我们不能无限制地增加Service，我们需要把所有AIDL放在同一个Service中去管理。
流程是：

1. 服务端只有一个Service，我们应该把所有AIDL放在一个Service中去管理，不同业务模块之间是不能有耦合的
2. 服务端提供一个 `queryBinder` 接口，这个借口能够根据业务模块的特征来返回响应的Binder对象给客户端
3. 不同的业务模块拿到所需的Binder对象就可以进行RPC了

详细源码参考这里。

2.6 选用合适的IPC方式

表 2-2 IPC 方式的优缺点和适用场景

名 称	优 点	缺 点	适 用 场 景
Bundle	简单易用	只能传输 Bundle 支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发场景，并且无法做到进程间的即时通信	无并发访问情形，交换简单的数据实时性不高的场景
AIDL	功能强大，支持一对多并发通信，支持实时通信	使用稍复杂，需要处理好线程同步	一对多通信且有 RPC 需求
Messenger	功能一般，支持一对多串行通信，支持实时通信	不能很好处理高并发情形，不支持 RPC，数据通过 Message 进行传输，因此只能传输 Bundle 支持的数据类型	低并发的一对多即时通信，无 RPC 需求，或者无须要返回结果的 RPC 需求
ContentProvider	在数据源访问方面功能强大，支持一对多并发数据共享，可通过 Call 方法扩展其他操作	可以理解为受约束的 AIDL，主要提供数据源的 CRUD 操作	一对多的进程间的数据共享
Socket	功能强大，可以通过网络传输字节流，支持一对多并发实时通信	实现细节稍微有点烦琐，不支持直接的 RPC	网络数据交换

3 View的事件体系

3.1 view的基础知识

3.1.1什么是view

View是Android中所有控件的基类，View的本身可以是单个空间，也可以是多个控件组成的一组控件，即ViewGroup，ViewGroup继承自View，其内部可以有子View，这样就形成了View树的结构。

3.1.2 View的位置参数

View的位置主要由它的四个顶点来决定，即它的四个属性：top、left、right、bottom，分别表示View左上角的坐标点（top，left）以及右下角的坐标点（right，bottom）。同时，我们可以得到View的大小：

```
width = right - left
height = bottom - top
```

而这四个参数可以由以下方式获取：

- Left = getLeft();
- Right = getRight();
- Top = getTop();
- Bottom = getBottom();
  Android3.0后，View增加了x、y、translationX和translationY这几个参数。其中x和y是View左上角的坐标，而translationX和translationY是View左上角相对于容器的偏移量。他们之间的换算关系如下：
  x = left + translationX;
  y = top + translationY;
  注意：View在平移的过程中，top和left不会改变，改变的是x、y、translationX和translaY。

3.1.3 MotionEvent和TouchSlop

1 MotionEvent

在手指接触到屏幕后会产生乙烯类的点击事件，如

- 点击屏幕后离开松开，事件序列为DOWN->UP
- 点击屏幕滑动一会再松开，事件序列为DOWN->MOVE->...->MOVE->UP 通过MotionEven对象我们可以得到事件发生的x和y坐标，我们可以通过getX/getY和getRawX/getRawY得到，它们的区别是：getX/getY返回的是相对于当前**View**左上角的**x**和**y**坐标，getRawX/getRawY返回的是相对于手机屏幕左上角的**x**和**y**坐标。

2 TouchSloup

TouchSloup是系统所能识别出的被认为是滑动的最小距离，这是一个常量，与设备有关，可通过以下方法获得：

```
ViewConfiguration.get(getContext()).getScaledTouchSloup().
```

3.1.4 VelocityTracker、GestureDetector和Scroller

1 VelocityTracker

速度追踪，用于追踪手指在滑动过程中的速度，包括水平放向速度和竖直方向速度。 使用方法：

1. 在View的onTouchEvent方法中追踪当前单击事件的速度

   ```
   VelocityRracker velocityTracker = VelocityTracker.obtain();
   velocityTracker.addMovement(event);
   ```

2. 计算速度，获得水平速度和竖直速度

   ```
   velocityTracker.computeCurrentVelocity(1000);
   int xVelocity = (int)velocityTracker.getXVelocity();
   int yVelocity = (int)velocityTracker.getYVelocity();
   ```

   注意，获取速度之前必须先计算速度，即调用computeCurrentVelocity方法，这里指的速度是指一段时间内手指滑过的像素数，1000指的是1000毫秒，得到的是1000毫秒内滑过的像素数。速度可正可负：

   ```
   速度 = （终点位置 - 起点位置） / 时间段
   ```

1. 最后，当不需要使用的时候，需要调用clear()方法重置并回收内存：

```
velocityTracker.clear();
velocityTracker.recycle();
```

2 GestureDetector

手势检测，用于辅助检测用户的单击、滑动、长按、双击等行为。 使用方法：

2. 创建一个GestureDetector对象并实现OnGestureListener接口，根据需要，也可实现OnDoubleTapListener接口从而监听双击行为：

```
GestureDetector mGestureDetector = new GestureDetector(this);
//解决长按屏幕后无法拖动的现象
mGestureDetector.setIsLongpressEnabled(false);
```

3. 在目标View的OnTouchEvent方法中添加以下实现：

```
boolean consume = mGestureDetector.onTouchEvent(event);
return consume;
```

4. 实现OnGestureListener和OnDoubleTapListener接口中的方法，其中常用的方法有：onSingleTapUp(单击)、onFling(快速滑动)、onScroll(拖动)、onLongPress(长按)和onDoubleTap（双击）。 建议：如果只是监听滑动相关的，可以自己在onTouchEvent中实现，如果要监听双击这种行为，那么就使用GestureDetector。

3 Scroller

弹性滑动对象，用于实现View的弹性滑动。其本身无法让View他行滑动，需要和View的computeScroll方法配合使用才能完成这个功能。 使用方法：

```
Scroller scroller = new Scroller(mContext);
//缓慢移动到指定位置
private void smoothScrollTo(int destX,int destY){
 int scrollX = getScrollX();
 int delta = destX - scrollX;
 //1000ms内滑向destX,效果就是慢慢滑动
 mScroller.startScroll(scrollX,0,delta,0,1000);
 invalidata();
}
@Override
public void computeScroll(){
 if(mScroller.computeScrollOffset()){
     scrollTo(mScroller.getCurrX,mScroller.getCurrY());
     postInvalidate();
 }
}
```

原理下节讲。

3.2 View的滑动

3.2.1 使用scrollTo/scrollBy

5. scrollBy实际调用了scrollTo，它实现了基于当前位置的相对滑动，而scrollTo则实现了绝对滑动。
6. scrollTo和scrollBy只能改变View的内容位置而不能改变View在布局中的位置。
7. 滑动偏移量mScrollX和mScrollY的正负与实际滑动方向相反，即从左向右滑动，mScrollX为负值，从上往下滑动mScrollY为负值。

3.2.2 使用动画

使用动画移动View，主要是操作View的translationX和translationY属性，既可以采用传统的View动画，也可以采用属性动画，如果使用属性动画，为了能够兼容3.0以下的版本，需要采用开源动画库nineolddandroids。 如使用属性动画：(View在100ms内向右移动100像素)

```
ObjectAnimator.ofFloat(targetView,"translationX",0,100).setDuration(100).start();
```

3.2.3 改变布局属性

通过改变布局属性来移动View，即改变LayoutParams。

3.2.4 各种滑动方式的对比

8. scrollTo/scrollBy：操作简单，适合对View内容的滑动；
9. 动画：操作简单，主要适用于没有交互的View和实现复杂的动画效果；
10. 改变布局参数：操作稍微复杂，适用于有交互的View。

3.3 弹性滑动

3.3.1 使用Scroller

使用Scroller实现弹性滑动的典型使用方法如下：

```
Scroller scroller = new Scroller(mContext);
//缓慢移动到指定位置
private void smoothScrollTo(int destX,int dextY){
    int scrollX = getScrollX();
    int deltaX = destX - scrollX;
    //1000ms内滑向destX，效果就是缓慢滑动
    mScroller.startSscroll(scrollX,0,deltaX,0,1000);
    invalidate();
}

@override
public void computeScroll(){
    if(mScroller.computeScrollOffset()){
        scrollTo(mScroller.getCurrX(),mScroller.getCurrY());
        postInvalidate();
    }
}
```

从上面代码可以知道，我们首先会构造一个Scroller对象，并调用他的startScroll方法，该方法并没有让view实现滑动，只是把参数保存下来，我们来看看startScroll方法的实现就知道了：

```
public void startScroll(int startX,int startY,int dx,int dy,int duration){
    mMode = SCROLL_MODE;
    mFinished = false;
    mDuration = duration;
    mStartTime = AnimationUtils.currentAminationTimeMills();
    mStartX = startX;
    mStartY = startY;
    mFinalX = startX + dx;
    mFinalY = startY + dy;
    mDeltaX = dx;
    mDeltaY = dy;
    mDurationReciprocal = 1.0f / (float)mDuration;
}
```

可以知道，startScroll方法的几个参数的含义，startX和startY表示滑动的起点，dx和dy表示的是滑动的距离，而duration表示的是滑动时间，注意，这里的滑动指的是***View***内容的滑动，在startScroll方法被调用后，马上调用invalidate方法，这是滑动的开始，***invalidate***方法会导致***View***的重绘，在***View***的***draw***方法中调用***computeScroll***方法，***computeScroll***又会去向***Scroller***获取当前的***scrollX***和***scrollY***；然后通过***scrollTo***方法实现滑动，接着又调用***postInvalidate***方法进行第二次重绘，一直循环，知道***computeScrollOffset()***方法返回值为***false***才结束整个滑动过程。 我们可以看看computeScrollOffset方法是如何获得当前的scrollX和scrollY的：

```
  public boolean computeScrollOffset(){
      ...
      int timePassed = (int)(AnimationUtils.currentAnimationTimeMills() - mStartTi
me);

      if(timePassed < mDuration){
          switch(mMode){
          case SCROLL_MODE:
              final float x = mInterpolator.getInterpolation(timePassed * mDuratio
nReciprocal);
              mCurrX = mStartX + Math.round(x * mDeltaX);
              mCurrY = mStartY + Math.round(y * mDeltaY);
              break;
          ...
          }
      }
      return true;
  }
```

到这里我们就基本明白了，computeScroll向Scroller获取当前的scrollX和scrollY其实是通过计算时间流逝的百分比来获得的，每一次重绘距滑动起始时间会有一个时间间距，通过这个时间间距Scroller就可以得到View当前的滑动位置，然后就可以通过scrollTo方法来完成View的滑动了。

3.3.2 通过动画

动画本身就是一种渐近的过程，因此通过动画来实现的滑动本身就具有弹性。实现也很简单：

```
  ObjectAnimator.ofFloat(targetView,"translationX",0,100).setDuration(100).start()
;
```

当然，我们也可以利用动画来模仿Scroller实现View弹性滑动的过程：

```
  final int startX = 0;
  final int deltaX = 100;
  ValueAnimator animator = ValueAnimator.ofInt(0,1).setDuration(1000);
  animator.addUpdateListener(new AnimatorUpdateListener(){
      @override
      public void onAnimationUpdate(ValueAnimator animator){
          float fraction = animator.getAnimatedFraction();
          mButton1.scrollTo(startX + (int) (deltaX * fraction) , 0);
      }
  });
  animator.start();
```

上面的动画本质上是没有作用于任何对象上的，他只是在1000ms内完成了整个动画过程，利用这个特性，我们就可以在动画的每一帧到来时获取动画完成的比例，根据比例计算出View所滑动的距离。

3.3.3 使用延时策略

延时策略的核心思想是通过发送一系列延时信息从而达到一种渐近式的效果，具体可以通过Hander和View的postDelayed方法，也可以使用线程的sleep方法。下面以Handler为例：

```
  private static final int MESSAGE_SCROLL_TO = 1;
  private static final int FRAME_COUNT = 30;
  private static final int DELATED_TIME = 33;

  private int mCount = 0;

  @suppressLint("HandlerLeak")
  private Handler handler = new handler(){
      public void handleMessage(Message msg){
          switch(msg.what){
          case MESSAGE_SCROLL_TO:
                  mCount ++ ;
                  if (mCount <= FRAME_COUNT){
                      float fraction = mCount / (float) FRAME_COUNT;
                      int scrollX = (int) (fraction * 100);
                      mButton1.scrollTo(scrollX,0);
                      mHandelr.sendEmptyMessageDelayed(MESSAGE_SCROLL_TO , DELAYED
_TIME);
                  }
                  break;
          default : break;
          }
      }
  }
```

3.4 View的事件分发机制

3.4.1 点击事件的传递规则

首先我们先看看下面一段伪代码，通过它我们可以理解到点击事件的传递规则：

```
public boolean dispatchTouchEvent (MotionEvent ev){
    boolean consume = false;
    if (onInterceptTouchEvnet(ev){
        consume = onTouchEvent(ev);
    } else {
        consume = child.dispatchTouchEnvet(ev);
    }

    return consume;
}
```

上面代码主要涉及到以下三个方法：

public boolean dispatchTouchEvent(MotionEvent ev); 这个方法用来进行事件的分发 ***public boolean onInterceptTouchEvent(MotionEvent ev);*** 这个方法用来判断是否拦截事件 ***public boolean onTouchEvent(MotionEvent ev);*** 这个方法用来处理点击事件。

下面理一理点击事件的传递规则：对于一个根***ViewGroup***，点击事件产生后，首先会传递给他，这时候就会调用他的***onDispatchTouchEvent***方法，如果***Viewgroup***的***onInterceptTouchEvent***方法返回***true***表示他要拦截事件，接下来事件就会交给***ViewGroup***处理，调用***ViewGroup***的***onTouchEvent***方法；如果***ViewGroup***的***onInteceptTouchEvent***方法返回值为***false***，表示***ViewGroup***不拦截该事件，这时事件就传递给他的子***View***，接下来子***View***的***dispatchTouchEvent***方法，如此反复直到事件被最终处理。

当一个View需要处理事件时，如果它设置了OnTouchListener，那么onTouch方法会被调用，如果onTouch返回false，则当前View的onTouchEvent方法会被调用，返回true则不会被调用，同时，在onTouchEvent方法中如果设置了OnClickListener，那么他的onClick方法会被调用。由此可见处理事件时的优先级关系：***onTouchListener > onTouchEvent > onClickListener***

关于事件传递的机制，这里给出一些结论：

1. 一个事件系列以down事件开始，中间包含数量不定的move事件，最终以up事件结束。
2. 正常情况下，一个事件序列只能由一个View拦截并消耗。
3. 某个View拦截了事件后，该事件序列只能由它去处理，并且它的onInterceptTouchEvent不会再被调用。
4. 某个View一旦开始处理事件，如果它不消耗ACTION_DOWN事件（onTouchEvnet返回false），那么同一事件序列中的其他事件都不会交给他处理，并且事件将重新交由他的父元素去处理，即父元素的onTouchEvent被调用。
5. 如果View不消耗ACTION_DOWN以外的其他事件，那么这个事件将会消失，此时父元素的onTouchEvent并不会被调用，并且当前View可以持续收到后续的事件，最终消失的点击事件会传递给Activity去处理。
6. ViewGroup默认不拦截任何事件。
7. View没有onInterceptTouchEvent方法，一旦事件传递给它，它的onTouchEvent方法会被

调用。

8. View的onTouchEvent默认消耗事件，除非他是不可点击的（clickable和longClickable同时为false）。
9. View的enable不影响onTouchEvent的默认返回值。
10. onClick会发生的前提是当前View是可点击的，并且收到了down和up事件。
11. 事件传递过程总是由外向内的，即事件总是先传递给父元素，然后由父元素分发给子View，通过requestDisallowInterceptTouchEvent方法可以在子元素中干预父元素的分发过程，但是ACTION_DOWN事件除外。

3.5 滑动冲突

在界面中，只要内外两层同时可以滑动，这个时候就会产生滑动冲突。

3.5.1 常见的滑动冲突场景

12. 外部滑动和内部滑动方向不一致；
13. 外部滑动方向和内部滑动方向一致；
14. 上面两种情况的嵌套。

3.5.2 滑动冲突的处理规则

15. 对于场景一，处理的规则是：当用户左右（上下）滑动时，需要让外部的View拦截点击事件，当用户上下（左右）滑动的时候，需要让内部的View拦截点击事件。根据滑动的方向判断谁来拦截事件。
16. 对于场景二，由于滑动方向一致，这时候只能在业务上找到突破点，根据业务需求，规定什么时候让外部View拦截事件，什么时候由内部View拦截事件。
17. 场景三的情况相对比较复杂，同样根据需求在业务上找到突破点。

3.5.3 滑动冲突的解决方式

18. 外部拦截法：所谓的外部拦截法是指点击事件都先经过父容器的拦截处理，如果父容器需要此事件就拦截，否则就不拦截。下面是伪代码：

```
public boolean onInterceptTouchEvent (MotionEvent event){
    boolean intercepted = false;
    int x = (int) event.getX();
    int y = (int) event.getY();
    switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN:
            intercepted = false;
            break;
    case MotionEvent.ACTION_MOVE:
            if (父容器需要当前事件) {
                intercepted = true;
            } else {
                intercepted = flase;
            }
            break;
    }
    case MotionEvent.ACTION_UP:
            intercepted = false;
            break;
    default : break;
}
mLastXIntercept = x;
mLastYIntercept = y;
return intercepted;
```

19. 内部拦截法：内部拦截法是指父容器不拦截任何事件，所有的事件都传递给子元素，如果子元素需要此事件就直接消耗，否则就交由父容器进行处理。这种方法与Android事件分发机制不一致，需要配合requestDisallowInterceptTouchEvent方法才能正常工作。下面是伪代码：

```
public boolean dispatchTouchEvent ( MotionEvent event ) {
    int x = (int) event.getX();
    int y = (int) event.getY();

    switch (event.getAction) {
    case MotionEvent.ACTION_DOWN:
            parent.requestDisallowInterceptTouchEvent(true);
            break;
    case MotionEvent.ACTION_MOVE:
            int deltaX = x - mLastX;
            int deltaY = y - mLastY;
            if (父容器需要此类点击事件) {
                parent.requestDisallowInterceptTouchEvent(false);
            }
            break;
    case MotionEvent.ACTION_UP:
            break;
    default : break;
    }

    mLastX = x;
    mLastY = y;
    return super.dispatchTouchEvent(event);
}
```

除了子元素需要做处理外，父元素也要默认拦截除了ACTION_DOWN以外的其他事件，这样当子元素调用parent.requestDisallowInterceptTouchEvent(false)方法时，父元素才能继续拦截所需的事件。因此，父元素要做以下修改：

```
public boolean onInterceptTouchEvent (MotionEvent event) {
    int action = event.getAction();
    if(action == MotionEvent.ACTION_DOWN) {
        return false;
    } else {
        return true;
    }
}
```

至于具体的实现可以根据实际需要去修改拦截成立的条件，开发艺术艺术中也给出了实例，具体可参考书中P161-P173。

4 View的工作原理

主要内容

自定义View、View的底层工作原理，比如View的测量流程、布局流程、绘制流程。

4.1 初识ViewRoot和DecorView

1. ViewRoot的实现是 `ViewRootImpl` 类，是连接WindowManager和DecorView的纽带，View的三大流程（**mearsure**、**layout**、**draw**）均是通过ViewRoot来完成。当Activity对象被创建完毕后，会将DecorView添加到Window中，同时创建 `ViewRootImpl` 对象，并将 `ViewRootImpl` 对象和DecorView建立连接。

```
root = new ViewRootImpl(view.getContext(),display);
root.setView(view,wparams, panelParentView);
```

2. View的三大流程:
   i. measure用来测量View的宽高
   ii. layout来确定View在父容器中的位置
   iii. draw负责将View绘制在屏幕上
3. View的绘制流程从ViewRoot的 `performTraversals` 方法开始：
   i. performTraversals会依次调用 `performMeasure` 、 `performLayout` 和 `performDraw` 三个方法，这三个方法分别完成顶级View的measure、layout和draw这三大流程。
   ii. 其中 `performMeasure` 中会调用 `measure` 方法，在 `measure` 方法中又会调用 `onMeasure` 方法，在 `onMeasure` 方法中则会对所有子元素进行measure过程，这样就完成了一次measure过程；子元素会重复父容器的measure过程，如此反复完成了整个View数的遍历。另外两个过程同理。

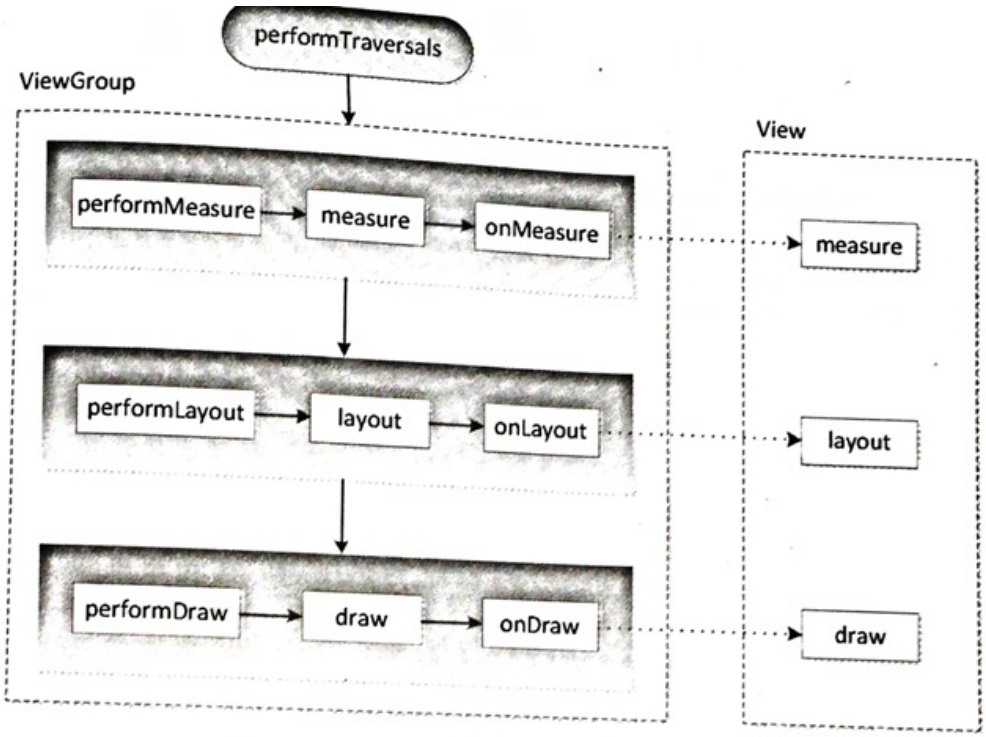


图 4-1　performTraversals 的工作流程图

4. Measure完成后，`getMeasuredWidth` / `getMeasureHeight` 方法来获取View测量后的宽/高。
5. Layout过程决定了View的四个顶点的坐标和实际View的宽高，完成后可通过 `getTop` 、 `getBotton` 、 `getLeft` 和 `getRight` 拿到View的四个定点坐标。
6. DecorView作为顶级View，其实是一个 `FrameLayout` ，它包含一个竖直方向的 `LinearLayout` ，这个 `LinearLayout` 分为标题栏和内容栏两个部分。在**Activity**通过**setContextView**所设置的布局文件其实就是被加载到内容栏之中的。这个内容栏的id是 `R.android.id.content` ，通过 `ViewGroup content = findViewById(R.android.id.content);` 可以得到这个contentView。**View**层的事件都是先经过**DecorView**，然后才传递到子**View**。

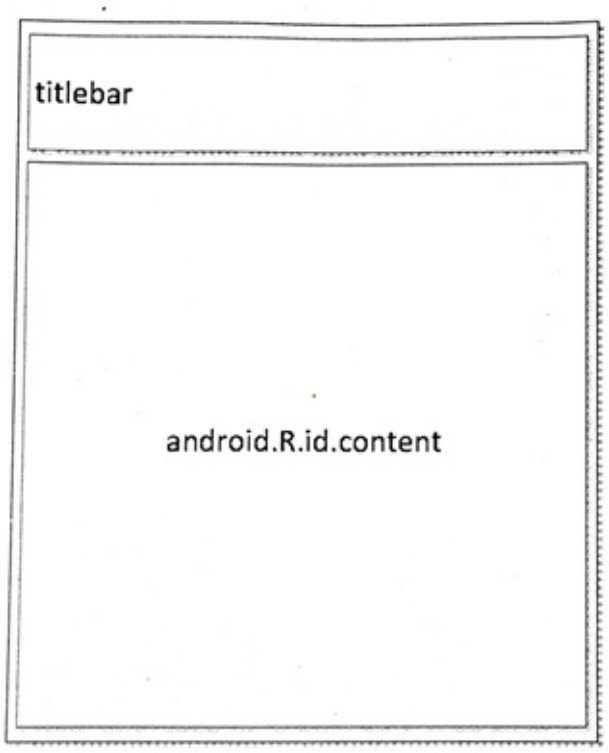


图 4-2　顶级 View：DecorView 的结构

4. 理解MeasureSpec

测量过程，系统将View的 `LayoutParams` 根据父容器所施加的规则转换成对应的**MeasureSpec**，然后根据这个**MeasureSpec**来测量出View的宽高。

4.2.1 MeasureSpec

1. MeasureSpec代表一个32位int值，高2位代表SpecMode（测量模式），低30位代表SpecSize（在某个测量模式下的规格大小）。
2. SpecMode有三种：
   i. **UNSPECIFIED** ：父容器不对View进行任何限制，要多大给多大，一般用于系统内部
   ii. **EXACTLY**：父容器检测到View所需要的精确大小，这时候View的最终大小就是SpecSize所指定的值，对应LayoutParams中的 `match_parent` 和具体数值这两种模式
   iii. **AT_MOST** ：对应View的默认大小，不同View实现不同，View的大小不能大于父容器的SpecSize，对应 `LayoutParams` 中的 `wrap_content`

4.2.2 MeasureSpec和LayoutParams的对应关系

View的**MeasureSpec**由父容器的**MeasureSpec**和自身的**LayoutParams**共同决定。View的measure过程由ViewGroup传递而来，参考ViewGroup的 `measureChildWithMargins` 方法，通过调用子元素的 `getChildMeasureSpec` 方法来得到子元素的**MeasureSpec**，再调用子元素的 `measure` 方法。书中根据源码总结出以下View的MeasureSpec创建规则：

表 4-1 普通 View 的 MeasureSpec 的创建规则

parentSpecMode / chlidLayoutParams	EXACTLY	AT_MOST	UNSPECIFIED
dp/px	EXACTLY childSize	EXACTLY childSize	EXACTLY childSize
match_parent	EXACTLY parentSize	AT_MOST parentSize	UNSPECIFIED 0
warap_content	AT_MOST parentSize	AT_MOST parentSize	UNSPECIFIED 0

稍有Android开发经验的人应该都能感受到这种规则：

1. 当View采用固定宽/高时（即设置固定的dp/px）,不管父容器的MeasureSpec是什么，View的MeasureSpec都是EXACTLY模式，并且大小遵循我们设置的值。
2. 当View的宽/高是 `match_parent` 时，View的MeasureSpec都是EXACTLY模式并且其大小等于父容器的剩余空间。
3. 当View的宽/高是 `wrap_content` 时，View的MeasureSpec都是AT_MOST模式并且其大小不能超过父容器的剩余空间。
4. 父容器的UNSPECIFIED模式，一般用于系统内部多次Measure时，表示一种测量的状态，一般来说我们不需要关注此模式。

> questions:
> UNSPECIFIED模式在系统内部多次Measure情况下被使用，具体是什么时候？表现又是什么？是在View的测量过程中发生的吗？书中没有说清楚，但是作为开发者，的确是没有接触到UNSPECIFED的情况。

4.3 View的工作流程

4.3.1 measure过程

分两种情况：

1. View通过 `measure` 方法就完成了测量过程
2. ViewGroup除了完成自己的测量过程还会便利调用所有子View的 `measure` 方法，而且各个子View还会递归执行这个过程。

1 View的measure过程

直接继承View的自定义控件需要重写 `onMeasure` 方法并设置 `wrap_content` （即specMode是 `AT_MOST` 模式）时的自身大小，否则在布局中使用 `wrap_content` 相当于使用 `match_parent` 。对于非 `wrap_content` 的情形，我们沿用系统的测量值即可。

2 ViewGroup的measure过程

ViewGroup是一个抽象类，没有重写View的 `onMeasure` 方法，但是它提供了一个 `measureChildren` 方法。这是因为不同的**ViewGroup**子类有不同的布局特性，导致他们的测量细节各不相同，比如 **`LinearLayout`** 和 **`RelativeLayout`** ,因此**ViewGroup**没办法同一实现 **`onMeasure`** 方法。 measureChildren方法的流程：

1. 取出子View的 `LayoutParams`
2. 通过 `getChildMeasureSpec` 方法来创建子元素的 `MeasureSpec`
3. 将 `MeasureSpec` 直接传递给View的measure方法来进行测量

> 文章通过LinearLayout的onMeasure方法里来分析ViewGroup的measure过程。
>
> 1. LinearLayout在布局中如果使用match_parent或者具体数值，测量过程就和View一直，即高度为specSize
> 2. LinearLayout在布局中如果使用wrap_content，那么它的高度就是所有子元素所占用的高度总和，并且不超过它的父容器的剩余空间。
> 3. 当然LinearLayout的的最终高度同时也把竖直方向的padding考虑在内

View的measure过程是三大流程中最复杂的一个，measure完成以后，通过 `getMeasuredWidth/Height` 方法就可以正确获取到View的测量后宽/高。在某些情况下，系统可能需要多次measure才能确定最终的测量宽/高，所以在onMeasure中拿到的宽/高很可能不是准确的。同时View的measure过程和Activity的生命周期并不是同步执行，因此无法保证在Activity的 `onCreate``onStart``onResume` 时某个View就已经测量完毕。所以有以下四种方式来获取View的宽高：

1. Activity/View#onWindowFocusChanged。 onWindowFocusChanged这个方法的含义是：VieW已经初始化完毕了，宽高已经准备好了，需要注意：它会被调用多次，当Activity的窗口得到焦点和失去焦点均会被调用。
2. view.post(runnable)。 通过post将一个runnable投递到消息队列的尾部，当Looper调用此runnable的时候，View也初始化好了。
3. ViewTreeObserver。 使用 `ViewTreeObserver` 的众多回调可以完成这个功能，比如 `OnGlobalLayoutListener` 这个接口，当View树的状态发送改变或View树内部的View的可见性发生改变时， `onGlobalLayout` 方法会被回调。需要注意的是，伴随着View树状态的改变， `onGlobalLayout` 会被回调多次。
4. view.measure(int widthMeasureSpec,int heightMeasureSpec)。
    i. match_parent： 无法measure出具体的宽高，因为不知道父容器的剩余空间，无法测量出View的大小

ii. 具体的数值（dp/px）：

```
 int widthMeasureSpec = MeasureSpec.makeMeasureSpec(100,MeasureSpec.EXACTLY);
 int heightMeasureSpec = MeasureSpec.makeMeasureSpec(100,MeasureSpec.EXACTLY)
;
 view.measure(widthMeasureSpec,heightMeasureSpec);
```

iii. wrap_content：

```
 int widthMeasureSpec = MeasureSpec.makeMeasureSpec((1<<30)-1,MeasureSpec.AT_
MOST);
 //View的尺寸使用30位二进制表示，最大值30个1，在AT_MOST模式下，我们用View理论上能支持的
最大值去构造MeasureSpec是合理的
 int heightMeasureSpec = MeasureSpec.makeMeasureSpec((1<<30)-1,MeasureSpec.AT
_MOST);
 view.measure(widthMeasureSpec,heightMeasureSpec);
```

4.3.2 layout过程

layout的作用是ViewGroup用来确定子View的位置，当ViewGroup的位置被确定后，它会在onLayout中遍历所有的子View并调用其layout方法，在 `layout` 方法中， `onLayout` 方法又会被调用。

`layout` 方法确定View本身的位置，源码流程如下： n

1. `setFrame` 确定View的四个顶点位置，即确定了View在父容器中的位置
2. 调用 `onLayout` 方法，确定所有子View的位置，View和ViewGroup均没有真正实现 `onLayout` 方法。

> 文章通过LinearLayout的 `onLayout` 方法里来分析ViewGroup的 `onLayout` 过程。
>
> 1. 遍历所有子View并调用 `setChildFrame` 方法来为子元素指定对应的位置
> 2. `setChildFrame` 方法实际上调用了子View的 `layout` 方法，形成了递归

4.3.3 draw过程

View的绘制过程遵循如下几步：

1. 绘制背景 `drawBackground(canvas)`
2. 绘制自己 `onDraw`
3. 绘制children `dispatchDraw` 遍历所有子View的 `draw` 方法
4. 绘制装饰 `onDrawScrollBars`

> tips:： ViewGroup会默认启用`setWillNotDraw`为ture，导致系统不会去执行`onDraw`，所以自定义ViewGroup需要通过onDraw来绘制内容时，必须显式的关闭`WILL_NOT_DRAW`这个标记位，即调用`setWillNotDraw(false);`

4.4 自定义View

4.4.1 自定义View的分类

1. 继承View 通过`onDraw`方法来实现一些效果，需要自己支持`wrap_content`，并且**padding**也要去进行处理。
2. 继承ViewGroup 实现自定义的布局方式，需要合适地处理**ViewGroup**的测量、布局这两个过程，并同时处理子**View**的测量和布局过程。
3. 继承特定的View子类（如TextView、Button） 扩展某种已有的控件的功能，且不需要自己去管理`wrap_content`和**padding**。
4. 继承特定的ViewGroup子类（如LinearLayout）

4.4.2 自定义View须知

1. 直接继承View或ViewGroup的控件， 需要在onmeasure中对wrap_content做特殊处理。指定**wrap_content**模式下的默认宽/高。
2. 直接继承View的控件，如果不在draw方法中处理padding，那么padding属性就无法起作用。直接继承ViewGroup的控件也需要在onMeasure和onLayout中考虑padding和子元素margin的影响，不然padding和子元素的margin无效。
3. 尽量不啊哟在View中使用Handler，因为没必要。View内部提供了post系列的方法，完全可以替代Handler的作用。
4. View中有线程和动画，需要在View的onDetachedFromWindow中停止。
5. View带有滑动嵌套情形时，需要处理好滑动冲突

4.4.3 自定义View的思想

1. 掌握基本功，比如View的弹性滑动、滑动冲突、绘制原理等
2. 面对新的自定义View时，对其分类并选择合适的实现思路。

7 Android动画深入分析

Android动画分为三种：

1. View动画
2. 帧动画
3. 属性动画

本章学习内容：

1. 介绍View动画和自定义View动画
2. View动画一些特殊的使用场景
3. 对属性动画全面性的介绍
4. 使用动画的一些注意事项

7.1 View动画

View动画的作用对象是View，支持四种动画效果：

1. 平移
2. 缩放
3. 旋转
4. 透明

7.1.1 View动画的种类

上述四种变换效果对应着Animation四个子类：`TranslateAnimation`、`ScaleAnimation`、`RotateAnimation`和`AlphaAnimation`。这四种动画皆可以通过XML定义，也可以通过代码来动态创建。

xml定义动画

1. `<set>`标签表示动画集合，对应AnimationSet类，可以包含一个或若干个动画，内部还可以嵌套其他动画集合。两个属性：
   i. `android:interpolator` 表示动画集合所采用的插值器，插值器影响动画速度，比如非匀速动画就需要通过插值器来控制动画的播放过程。
   ii. `android:shareInterpolator` 表示集合中的动画是否和集合共享同一个插值器，如果集合不指定插值器，那么子动画就需要单独指定所需的插值器或默认值。
2. `<translate>`、`<scale>`、`<rotate>`、`<alpha>`这几个子标签分别代表四种变换效果。

3. 定义完View动画的xml后，通过以下代码应用动画：

```
Aniamation anim = AnimationUtils.loadAnimation(context,R.anim.animation_test);
view.startAnimation(anim);
```

代码动态创建动画

```
AlphaAnimation alphaAnimation = new AlphaAnimation(0,1);
alphaAnimation.setDuration(1500);
view.startAnimation(alphaAnimation);
```

7.1.2 自定义View动画

需要继承 `Animation` 这个抽象类，重写它的 `initialize` 和 `applyTransformation` 方法。在 `initialize` 方法中做一些初始化工作，在 `applyTransformation` 中进行相应的矩阵变换即可，很多时候需要采用 `Camera` 来简化矩阵变换的过程。自定义View动画的过程主要是矩阵变换的过程。

7.1.3 帧动画

帧动画是顺序播放一组预先定义好的图片，使用简单，但容易引起OOM，所以在使用帧动画时应尽量避免使用过多尺寸较大的图片。

7.2 View动画的特殊使用场景

7.2.1 LayoutAnimation

作用于ViewGroup，为ViewGroup指定一个动画，当它的子元素出场时都会具有这种动画效果，一般用在ListView上。

7.2.2 Activity的切换效果

我们可以自定义Activity的切换效果，主要通过在 `startActivity` 或者 `finish` 的后面增加 `overridePendingTransition(int enterAnim , int exitAnim)` 方法

7.3 属性动画

API 11后加入，可以在一个时间间隔内完成对象从一个属性值到另一个属性值的改变。因此与View动画相比，属性动画几乎无所不能，只要对象有这个属性，它都能实现动画效果。API11以下可以通过 `nineoldandroids` 库来兼容以前版本。
属性动画有以下三种使用方法：

1. ObjectAnimator

```
ObjectAnimator.ofFloat(view,"translationY",values).start();
```

2. ValueAnimator

```
ValueAnimator colorAnim = ObjectAnimator.ofInt(view,"backgroundColor",/*red*/0xff
ff8080,/*blue*/0xff8080ff);
colorAnim.setDuration(2000);
colorAnim.setEvaluator(new ArgbEvaluator());
colorAnim.setRepeatCount(ValueAnimator.INFINITE);
colorAnim.setRepeatMode(ValueAnimator.REVERSE);
colorAnim.start();
```

3. AnimatorSet

```
AnimatorSet set = new AnimatorSet();
set.playTogether(animator1,animator2,animator3);
set.setDuration(3*1000).start();
```

也可以通过在xml中定义在 `res/animator/` 目录下。具体如下：

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
 <objectAnimator
 ...../>
 <animator
 ...../>
</set>
```

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(context , R.animator.ani
m);
set.setTarget(view);
set.start();
```

`<set>` 标签对应 `AnimatorSet` ，`<animator>` 对应 `ValueAnimator` ，而 `<objectAnimator>` 则对应 `ObjectAnimator` 。

7.3.2 理解差值器和估值器

1. 时间插值器（TimeInterpolator）的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比，系统预置的有LinearInterpolator（线性插值器：匀速动画），AccelerateDecelerateInterpolator（加速减速插值器：动画两头慢中间快），DecelerateInterpolator(减速插值器：动画越来越慢）。
2. 估值器（TypeEvaluator）的作用是根据当前属性改变的百分比来计算改变后的属性值。系统预置有IntEvaluator 、FloatEvaluator 、ArgbEvaluator。
3. 具体来说 对于一个作用在view上改变其宽度属性、持续40ms的属性动画来说，就是 当时间t=20ms时，时间流逝了50%，那么view的宽度属性应该改变了多少呢？这个就由**Interpolator**和**Evaluator**的算法来决定。

7.3.3 属性动画的监听器

```
public static interface AnimatorListener {
    void onAnimationStart(Animator animation); //动画开始
    void onAnimationEnd(Animator animation); //动画结束
    void onAnimationCancel(Animator animation); //动画取消
    void onAnimationRepeat(Animator animation); //动画重复播放
}
```

为了方便开发，系统提供了AnimatorListenerAdapter类，它是AnimatorListener的适配器类，可以有选择的实现以上4个方法。

```
 /**
     * Implementors of this interface can add themselves as update listeners
     * to an <code>ValueAnimator</code> instance to receive callbacks on every animati
on
     * frame, after the current frame's values have been calculated for that
     * <code>ValueAnimator</code>.
     */
    public static interface AnimatorUpdateListener {
        /**
         * <p>Notifies the occurrence of another frame of the animation.</p>
         *
         * @param animation The animation which was repeated.
         */
        void onAnimationUpdate(ValueAnimator animation);

    }
```

AnimatorUpdateListener会监听整个动画的过程，动画由许多帧组成的，每播放一帧，onAnimationUpdate就会调用一次。

7.3.4 对任意属性做动画

1. 属性动画要求作用的对象提供该属性的get和set方法，属性动画根据外界传递的该属性的初始值和最终值，通过多次调用set方法来实现动画效果。
2. 如果被作用的对象没有set/get方法，可以：

    i. 请给你的对象加上get和set方法，如果你有权限的话（对于SDK或者其他第三方类库的类无法加上的）

    ii. 用一个类来包装原始对象，间接为其提供get和set方法

```
//包装View类 用于给属性动画调用 从而包装了set get
public class ViewWrapper {
 private View target;
 public ViewWrapper(View target) {
     this.target = target;
 }
 public int getWidth() {
     return target.getLayoutParams().width;
 }
 public void setWidth(int width) {
     target.getLayoutParams().width = width;
     target.requestLayout();
 }
}

//使用：
ViewWrapper wrapper = new ViewWrapper(mButton);
ObjectAnimator.ofInt(mButton,"width",500).setDuration(3000).start();
```

    iii. 采用ValueAnimator，监听动画过程，自己实现属性的改变；

```
 private void performAnimate(final View target, final int start, final int en
d) {
        ValueAnimator valueAnimator = ValueAnimator.ofInt(1, 100);
        valueAnimator.addUpdateListener(new AnimatorUpdateListener() {

            // 持有一个IntEvaluator对象，方便下面估值的时候使用
            private IntEvaluator mEvaluator = new IntEvaluator();

            @Override
            public void onAnimationUpdate(ValueAnimator animator) {
                // 获得当前动画的进度值，整型，1-100之间
                int currentValue = (Integer) animator.getAnimatedValue();
                Log.d(TAG, "current value: " + currentValue);

                // 获得当前进度占整个动画过程的比例，浮点型，0-1之间
                float fraction = animator.getAnimatedFraction();
                // 直接调用整型估值器通过比例计算出宽度，然后再设给Button
                target.getLayoutParams().width = mEvaluator.evaluate(fractio
n, start, end);
                target.requestLayout();
            }
        });

        valueAnimator.setDuration(5000).start();
    }
```

7.3.5 属性动画的工作原理

属性动画需要运行在有Looper的线程中，系统通过反射调用被作用对象**get/set**方法。

7.4 使用动画的注意事项

1. 使用帧动画时，当图片数量较多且图片分辨率较大的时候容易出现**OOM**，需注意，尽量避免使用帧动画。
2. 使用无限循环的属性动画时，在Activity退出时即使停止，否则将导致Activity无法释放从而造成内存泄露。
3. 动画在3.0以下的系统存在兼容性问题，特殊场景可能无法正常工作，需做好适配工作。
4. View动画是对View的影像做动画，并不是真正的改变了View的状态，因此有时候会出现动画完成后View无法隐藏（setVisibility(View.GONE）失效），这时候调用`view.clearAnimation()`清理View动画即可解决。
5. 不要使用px，使用px会导致不同设备上有不同的效果。
6. View动画是对View的影像做动画，View的真实位置没有变动，动画完成后的新位置是无法触发点击事件的。属性动画是真实改变了View的属性，所以动画完成后的位置可以接受触摸事件。

7. 使用动画的过程中，使用硬件加速可以提高动画的流畅度。

8 理解Window和WindowMananger

Window是一个抽象类，具体实现是`PhoneWindow`。不管是`Activity`、`Dialog`、`Toast`它们的视图都是附加在Window上的，因此**Window**实际上是**View**的直接管理者。`WindowManager`是外界访问Window的入口，通过**WindowManager**可以创建**Window**，而Window的具体实现位于`WindowManagerService`中，WindowManager和WindowManagerService的交互是一个IPC过程。

8.1 Window和WindowManager

下面代码演示了通过WindowManager添加Window的过程：

```
mWindowManager = (WindowManager) getSystemService(Context.WINDOW_SERVICE);
mFloatingButton = new Button(this);
            mFloatingButton.setText("click me");
            mLayoutParams = new WindowManager.LayoutParams(
                    LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT, 0, 0,
                    PixelFormat.TRANSPARENT);
            mLayoutParams.flags = LayoutParams.FLAG_NOT_TOUCH_MODAL
                    | LayoutParams.FLAG_NOT_FOCUSABLE
                    | LayoutParams.FLAG_SHOW_WHEN_LOCKED;
            mLayoutParams.type = LayoutParams.TYPE_SYSTEM_ERROR;
            mLayoutParams.gravity = Gravity.LEFT | Gravity.TOP;
            mLayoutParams.x = 100;
            mLayoutParams.y = 300;
            mFloatingButton.setOnTouchListener(this);
            mWindowManager.addView(mFloatingButton, mLayoutParams);
```

WindowManager的flags和type这两个属性比较重要：
Flags代表Window的属性，控制Window的显示特性

1. FLAG_NOT_FOCUSABLE 在此模式下，Window不需要获取焦点，也不需要接收各种输入事件，这个标记同时会启用FLAG_NOT_TOUCH_MODAL，最终事件会直接传递给下层具有焦点的Window。
2. FLAG_NOT_TOUCH_MODAL 在此模式下，系统将当前Window区域以外的点击事件传递给底层的Window，当前Window区域内的单击事件则自己处理。一般需要开启此标记。
3. FLAG_SHOW_WHEN_LOCKED 开启此模式Window将显示在锁屏界面上。

type参数表示Window的类型。

1. 应用Window

2. 子Window 如Dialog
3. 系统Window 如Toast和系统状态栏

Window是分层的，每个Window对应一个z-ordered，层级大的会覆盖在层级小的上面，和HTM的z-index概念一样。在三类Window中，应用Window的层级范围是1~99，子Window的层级范围是1000~1999，系统Window的层级范围是2000~2999，这些值对应**WindowManager.LayoutParams**的**type**参数。一般系统Window选用 `TYPE_SYSTEM_OVERLAY` 或者 `TYPE_SYSTEM_ERROR` （同时需要权限 `<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />` ）。

WindowManager提供的功能很简单，常用的只有三个方法：

1. 添加View
2. 更新View
3. 删除View

这个三个方法定义在 `ViewManager` 中，而WindowManager继承了ViewManager。

```
public interface ViewManager
{
    /**
     * Assign the passed LayoutParams to the passed View and add the view to the windo
w.
     * <p>Throws {@link android.view.WindowManager.BadTokenException} for certain prog
ramming
     * errors, such as adding a second view to a window without removing the first vie
w.
     * <p>Throws {@link android.view.WindowManager.InvalidDisplayException} if the win
dow is on a
     * secondary {@link Display} and the specified display can't be found
     * (see {@link android.app.Presentation}).
     * @param view The view to be added to this window.
     * @param params The LayoutParams to assign to view.
     */
    public void addView(View view, ViewGroup.LayoutParams params);
    public void updateViewLayout(View view, ViewGroup.LayoutParams params);
    public void removeView(View view);
}
```

8.2 Window的内部机制

Window是一个抽象的概念，每一个**Window**都对应着一个**View**和一个**ViewRootImpl**，**Window**和**View**通过**ViewRootImpl**来建立联系。因此Window并不是实际存在的，它是以View的形式存在的。所以WindowManager的三个方法都是针对View的，说明**View**才是**Window**存在的实体。在实际使用中无法直接访问Window，必须通过WindowManager来访问Window。

8.2.1 Window的添加过程

8.2.2 Window的删除过程

8.2.3 Window的更新过程

以上三个小节都是通过WindowManager的真正实现类——`WindowManagerImpl`类的三大方法的源码来对Window的内部机制进行分析。
具体点说就是，WindowManagerImpl并没有直接实现Window的三大操作，而是全部交给`WindowManagerGlobal`处理。然后在`WindowManagerGlobal`内部都是通过`ViewRootImpl`里的一个Binder对象`mWindowSession`（`IWindowSession`类型）进行IPC调用`WindowManagerService`进行Window的三大操作。

8.3 Window的创建过程

由之前的分析可以知道，View是Android中视图的呈现方式，但是View不能单独存在，必须附着在Window这个抽象的概念上面，因此有视图的地方就有Window。这些视图包括：Activity、Dialog、Toast、PopUpWindow等等。

8.3.1 Activity的Window创建过程

8.3.2 Dialog的Window创建过程

8.3.3 Toast的Window创建过程

由于以上三个小节有大量源码的分析，脱离了具体的代码片段不方便做笔记。只做简单概括：

1. 在创建视图并显示出来时，首先是通过创建一个Window对象，然后通过WindowManager对象的`addView(View view, ViewGroup.LayoutParams params);`方法将`contentView`添加到Window中，完成添加和显示视图这两个过程。
2. 在关闭视图时，通过WindowManager来移除DecorView，`mWindowManager.removeViewImmediate( view);`。
3. Toast比较特殊，具有定时取消功能，所以系统采用了Handler，内部有两类IPC过程：
    i. Toast访问`NotificationManagerService`
    ii. `NotificationManagerService`回调Toast里的`TN`接口

显示和隐藏Toast都通过NotificationManagerService（NMS）来实现，而NMS运行在系统进程中，所以只能通过IPC来进行显示/隐藏Toast。而TN是一个Binder类，在Toast和NMS进行IPC的过程中，当NMS处理Toast的显示/隐藏请求时会跨进程回调TN中的方法，这时由于TN

运行再Binder线程池中，所以需要通过Handler将其切换到当前线程（即发起Toast请求所在的线程），然后通过WindowManager的 `addView/removewView` 方法真正完成显示和隐藏Toast。

本章节对源码的分析侧重的是整体流程，避免深入代码逻辑无法自拔的情形。

9 四大组件的工作过程

本章的意义在于加深对四大组件工作方式的认识，有助于加深对Android整体的体系结构的认识。很多情况下，只有对Android的体系结构有一定认识，在实际开发中才能写出优秀的代码。 读者对四大组件的工作过程有一个感性的认识并且能够给予上层开发一些指导意义。

9.1 四大组件的运行状态

1. Activity是一种展示型组件，用于向用户直接地展示一个界面，并且可以接收用户的输入信息从而进行交互，扮演的是一个前台界面的角色。
2. Service是一种计算型组件，在后台执行一系列计算任务。它本身还是运行在主线程中的，所以耗时的逻辑仍需要单独的线程去完成。
3. BroadcastReceiver是一种消息型组件，用于在不同的组件乃至不同的应用之间传递消息。
4. ContentProvider是一种数据共享型组件，用于向其他组件乃至其他应用共享数据。其内部的增删改查四大方法是在Binder线程池中被调用的。

9.2 Activity的工作过程

主要分析Activity启动流程。

1. Activity的所有 `startActivity` 重载方法最终都会调用 `startActivityForResult` 。
2. 调用 `mInstrumentation.execStartActivity.execStartActivity()` 方法。
3. 在execStartActivity方法内，真正启动Activity
   由 `ActivityManagerNative.getDefault()` 的 `startActivity` 方法实现。
4. ActivityManagerNative继承 `Binder` 并实现 `IActivityManager` 这个Binder接口。具体实现是 `ActivityManagerService` 。
5. 所以Activity的启动过程实际上在AMS中：
   i. `checkStartActivityResult ()` 方法检查启动Activity的结果（包括检查有无在manifest注册）
   ii. `startActivity()` 方法将Activity的启动过程交给一个 `ActivityStackSupervisor` 对象mStackSupervisor的 `startActivityMayWait` 方法。
   iii. 最终Activity的启动过程是在 `ActivityStackSupervisor` 和 `ActivityStack` 之间传递，如下图。

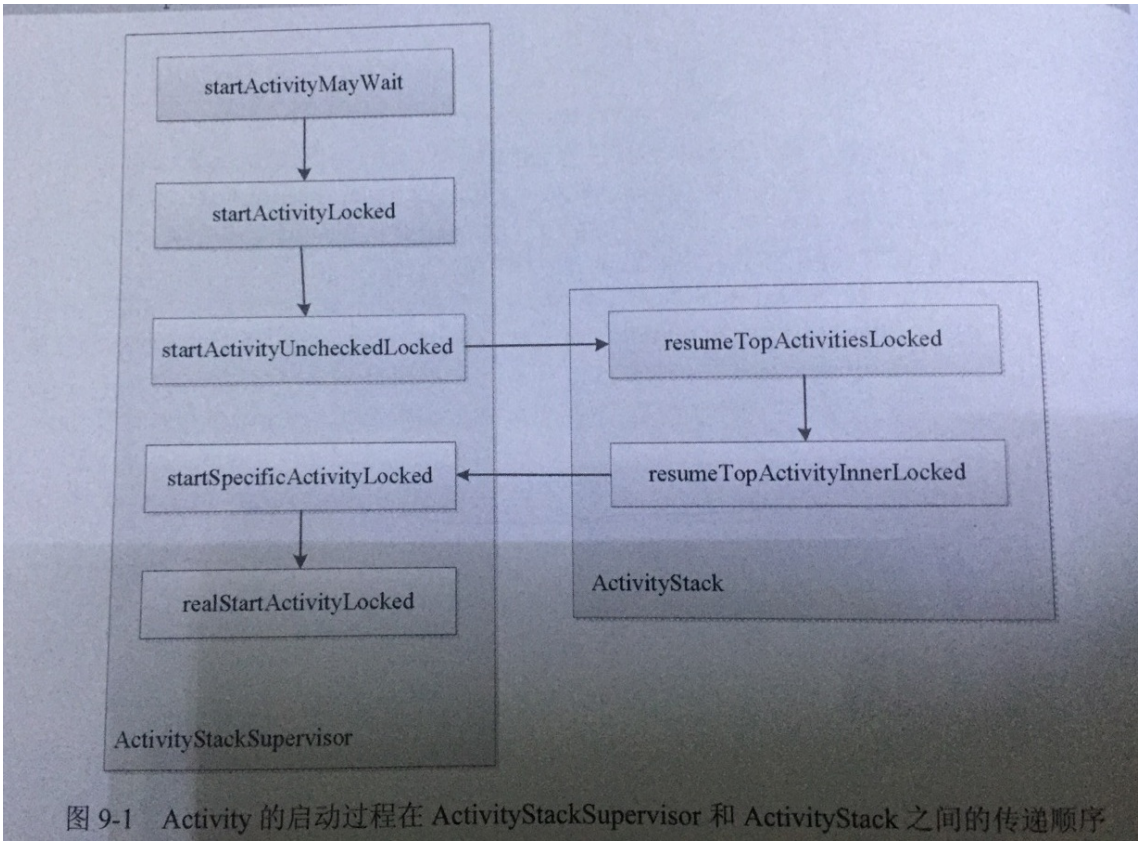


图 9-1 Activity 的启动过程在 ActivityStackSupervisor 和 ActivityStack 之间的传递顺序

6. 在最后的 `ActivityStackSupervisor. realStartActivityLocked()` 中，调用了 `app.thread.scheduleLaunchActivity()` 方法。 这个app.thread是 `ApplicationThread` 类型，继承于 `IApplicationThread` 是一个Binder类，内部是各种启动/停止 Service/Activity 的接口。
7. 在ApplicationThread中， `scheduleLaunchActivity()` 用来启动Activity，里面的实现就是发送一个Activity的消息（封装成 `从ActivityClientRecord` 对象）交给Handler处理。这个Handler有一个简洁的名字 `H` 。
8. 在H的 `handleMessage()` 方法里，通过 `handleLaunchActivity()` 方法完成Activity对象的创建和启动。主要完成了如下几件事：
    i. 从ActivityClientRecord对象中获取待启动的Activity组件信息
    ii. 通过 `Instrumentation` 的 `newActivity` 方法使用类加载器创建Activity对象
    iii. 通过 `LoadedApk` 的makeApplication方法尝试创建Application对象，通过类加载器实现（如果Application已经创建过了就不会再创建）
    iv. 创建 `ContextImpl` 对象并通过Activity的 `attach` 方法完成一些重要数据的初始化
    v. 通过 `mInstrumentation.callActivityOnCreate(activity, r.state)` 方法调用Activity的 `onCreate` 方法

9.3 Service的工作过程

Service有两种工作状态：

1. 启动状态：执行后台计算

2. 绑定状态：用于其他组件与Service交互

9.3.1 Service的启动过程

1. Service的启动从 `ContextWrapper` 的 `startService` 开始
2. 在ContextWrapper中，大部分操作通过一个 `ContextImpl` 对象mBase实现
3. 在ContextImpl中， `mBase.startService()` 会调用 `startServiceCommon` 方法，而startServiceCommon方法又会通过 `ActivityManagerNative.getDefault()` （实际上就是**AMS**）这个对象来启动一个服务。
4. AMS会通过一个 `ActiveService` 对象（辅助AMS进行Service管理的类）mServices来完成启动Service： `mServices.startServiceLocked()` 。
5. 在mServices.startServiceLocked()最后会调用 `startServiceInnerLocked()` 方法：将Service的信息包装成一个 `ServiceRecord` 对象，通过 `bringUpServiceLocked()` 方法来处理，bringUpServiceLocked()又调用了 `realStartServiceLocked()` 方法，这才真正地去启动一个Service了。
6. realStartServiceLocked()方法的工作如下：
   i. `app.thread.scheduleCreateService()` 来创建Service并调用其onCreate()生命周期方法
   ii. `sendServiceArgsLocked()` 调用其他生命周期方法，如onStartCommand()
   iii. app.thread对象是 `IApplicationThread` 类型，实际上就是一个Binder，具体实现是ApplicationThread继承ApplictionThreadNative
7. 具体看app.thread.scheduleCreateService()：通过 `sendMessage(H.CREATE_SERVICE , s)` ，这个过程和Activity启动过程类似，同时通过发送消息给Handler H来完成的。
8. H会接受这个CREATE_SERVICE消息并通过ActivityThread的 `handleCreateService()` 来完成Service的最终启动。
9. handleCreateService()完成了以下工作：
   i. 通过ClassLoader创建Service对象
   ii. 创建Service内部的Context对象
   iii. 创建Application，并调用其onCreate()（只会有一次）
   iv. 通过 `service.attach()` 方法建立Service与context的联系（与Activity类似）
   v. 调用service的 `onCreate()` 生命周期方法，至此，**Service**已经启动了
   vi. 将Service对象存储到ActivityThread的一个ArrayMap中

9.3.2 Service的绑定过程

和service的启动过程类似的：

1. Service的绑定是从 `ContextWrapper` 的 `bindService` 开始
2. 在ContextWrapper中，交给 `ContextImpl` 对象 `mBase.bindService()`
3. 最终会调用ContextImpl的 `bindServiceCommon` 方法，这个方法完成两件事：
   i. 将客户端的ServiceConnection转化成 `ServiceDispatcher.InnerConnection` 对象。

ServiceDispatcher连接**ServiceConnection**和**InnerConnection**。

i. 这个过程通过 `LoadedApk` 的 `getServiceDispatcher` 方法来实现，将客户端的ServiceConnection和ServiceDispatcher的映射关系存在一个ArrayMap中。

ii. 通过AMS来完成Service的具体绑定过程 `ActivityManagerNative.getDefault().bindService()`

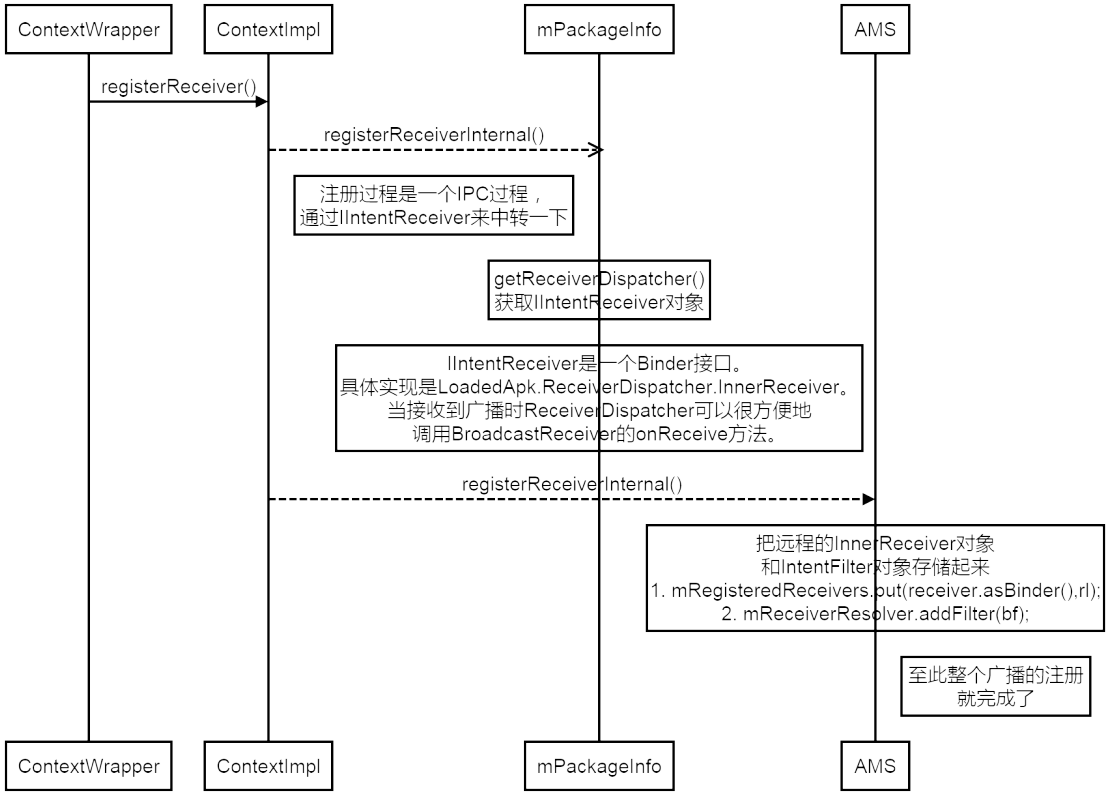
4. AMS中，bindService()方法再调用 `bindServiceLocked()` ，bindServiceLocked()再调用 `bringUpServiceLocked()` ，bringUpServiceLocked()又会调用 `realStartServiceLocked()` 。
5. AMS的realStartServiceLocked()会调用 `ActiveServices` 的 `requrestServiceBindingLocked()` 方法，最终是调用了ServiceRecord对象r的 `app.thread.scheduleBindService()` 方法。
6. ApplicationThread的一系列以***schedule***开头的方法，内部都通过Handler H来中转：scheduleBindService()内部也是通过 `sendMessage(H.BIND_SERVICE , s)`
7. 在H内部接收到BIND_SERVICE这类消息时就交给 `ActivityThread` 的 `handleBindService()` 方法处理：
   i. 根据Servcie的token取出Service对象
   ii. 调用Service的 `onBind()` 方法，至此，**Service**就处于绑定状态了。
   iii. 这时客户端还不知道已经成功连接Service，需要调用客户端的binder对象来调用客户端的ServiceConnection中的 `onServiceConnected()` 方法，这个通过 `ActivityManagerNative.getDefault().publishService()` 进行。
   ActivityManagerNative.getDefault()就是**AMS***。
8. AMS的publishService()交给ActivityService对象 `mServices` 的 `publishServiceLocked()` 来处理，核心代码就一句话 `c.conn.connected(r.name,service)` 。对象c的类型是 `ConnectionRecord` ，c.conn就是ServiceDispatcher.InnerConnection对象，service就是Service的onBind方法返回的Binder对象。
9. c.conn.connected(r.name,service)内部实现是交给了 `mActivityThread.post(new RunnConnection(name ,service,0));` 实现。ServiceDispatcher的mActivityThread是一个Handler，其实就是ActivityThread中的H。这样一来RunConnection就经由H的post方法从而运行在主线程中，因此客户端**ServiceConnection**中的方法是在主线程中被回调的。
10. RunConnection的定义如下：
    i. 继承Runnable接口， `run()` 方法的实现也是简单调用了ServiceDispatcher的 `doConnected` 方法。
    ii. 由于ServiceDispatcher内部保存了客户端的ServiceConntion对象，可以很方便地调用ServiceConntion对象的 `onServiceConnected` 方法。
    iii. 客户端的onServiceConnected方法执行后，Service的绑定过程也就完成了。
    iv. 根据步骤8、9、10service绑定后通过ServiceDispatcher通知客户端的过程可以说明**ServiceDispatcher**起着连接**ServiceConnection**和**InnerConnection**的作用。 至于Service的停止和解除绑定的过程，系统流程都是类似的。

9.4 BoradcastReceiver的工作过程

9.4.1 广播的注册过程

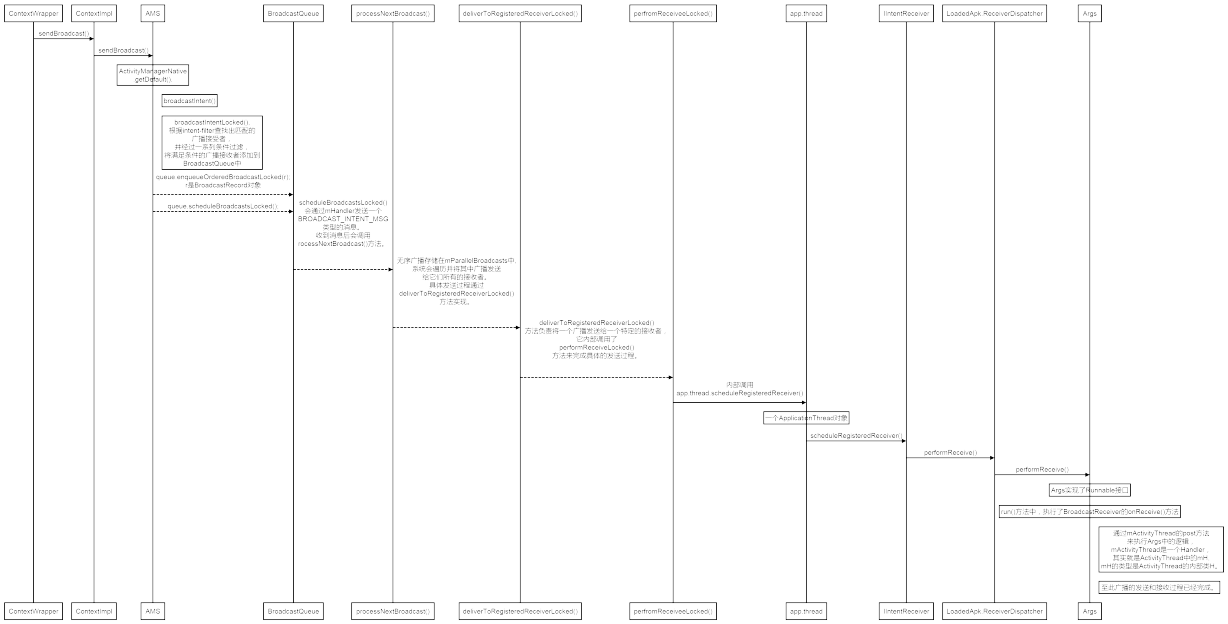
广播的注册分为：

1. 静态注册，由PMS(PackageManagerService）完成整个注册过程
2. 动态注册，如下图



9.4.2 广播的发送和接收过程

广播的发送有几种：普通广播、有序广播和粘性广播，他们的发送/接收流程是类似的，因此只分析普通广播的实现。



10 Android的消息机制

从开发的角度来说，Handler是Android消息机制的上层接口。Handler的运行需要底层的`MessageQueue`和`Looper`的支撑。

1. MessageQueue是一个消息队列，内部存储了一组消息，以队列的形式对外提供插入和删除的工作，内部采用单链表的数据结构来存储消息列表。
2. Lopper会以无限循环的形式去查找是否有新消息，如果有就处理消息，否则就一直等待着。

线程是默认没有Looperd的，使用Handler就必须为线程创建Looper。

我们经常提到的主线程，也叫UI线程，它就是ActivityThread

10.1 Android的消息机制概述

1. Handler的主要作用是将某个任务切换到Handler所在的线程中去执行。为什么Android要提供这个功能呢？这是因为Android规定访问UI只能通过主线程，如果子线程访问UI,程序会抛出异常；ViewRootImpl在checkThread方法中做了判断。
2. 由于Android不建议在主线程进行耗时操作，否则可能会导致ANR。那我们耗时操作在子线程执行完毕后，我们需要将一些更新UI的操作切换到主线程当中去。所以系统就提供了Handler。
3. 系统为什么不允许在子线程中去访问UI呢？ 因为Android的UI控件不是线程安全的，多线程并发访问可能会导致UI控件处于不可预期的状态，为什么不加锁？因为加锁机制会让UI访问逻辑变得复杂；其次锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。所以**Android**采用了高效的单线程模型来处理**UI**操作。
4. Handler创建时会采用当前线程的Looper来构建内部的消息循环系统，如果当前线程没有Looper就会报错。Handler可以通过post方法发送一个Runnable到消息队列中，也可以通过send方法发送一个消息到消息队列中，其实post方法最终也是通过send方法来完成。
5. MessageQueue的enqueueMessage方法最终将这个消息放到消息队列中，当Looper发现有新消息到来时，处理这个消息，最终消息中的Runnable或者Handler的handleMessage方法就会被调用，注意Looper是运行Handler所在的线程中的，这样一来业务逻辑就切换到了Handler所在的线程中去执行了。

10.2 Android的消息机制分析

10.2.1 ThreadLocal的工作原理

ThreadLocal是一个线程内部的数据存储类，通过它可以在指定线程中存储数据，数据存储后，只有在指定线程中可以获取到存储的数据，对于其他线程来说无法获得数据。对于**Handler**来说，它需要获取当前线程的**Looper**,而**Looper**的作用于就是线程并且不同的线程具有不同的**Looper**，通过**ThreadLocal**可以轻松实现线程中的存取。 通过ThreadLocal可以让监听器作为线程内的全局对象而存在，在线程内部只要通过get方法就可以获取到监听器。ThreadLocal原理：不同线程访问同一个ThreadLoacl的get方法，ThreadLocal的get方法会从各自的线程中取出一个数组，然后再从数组中根据当前**ThreadLocal**的索引去查找对应的**Value**值。

ThreadLocal的set方法：

```
public void set(T value) {
    Thread currentThread = Thread.currentThread();
    //通过values方法获取当前线程中的ThreadLoacl数据——localValues
    Values values = values(currentThread);
    if (values == null) {
        values = initializeValues(currentThread);
    }
    values.put(this, value);
}
```

1. 在 `localValues` 内部有一个数组： `private Object[] table` ，**ThreadLocal**的值就存在这个数组中。
2. ThreadLocal的值在table数组中的存储位置总是ThreadLocal的reference字段所标识的对象的下一个位置。

ThreadLocal的get方法：

```
public T get() {
    // Optimized for the fast path.
    Thread currentThread = Thread.currentThread();
    Values values = values(currentThread);//找到localValues对象
    if (values != null) {
        Object[] table = values.table;
        int index = hash & values.mask;
        if (this.reference == table[index]) {//找到ThreadLocal的reference对象在table数组
中的位置
            return (T) table[index + 1];//reference字段所标识的对象的下一个位置就是ThreadL
的值
        }
    } else {
        values = initializeValues(currentThread);
    }
    return (T) values.getAfterMiss(this);
}
```

从ThreadLocal的set/get方法可以看出，它们所操作的对象都是当前线程的**localValues**对象的**table**数组，因此在不同线程中访问同一个ThreadLocal的set/get方法，它们对ThreadLocal的读/写操作仅限于各自线程的内部。

10.2.2 消息队列的工作原理

1. 消息队列指的是MessageQueue，主要包含两个操作：插入和读取。读取操作本身会伴随着删除操作。
2. MessageQueue内部通过一个单链表的数据结构来维护消息列表，这种数据结构在插入和删除上的性能比较有优势。
3. 插入和读取对应的方法分别是：`enqueueMessage`和`next`方法。
   i. enqueueMessage()的源码实现主要操作就是单链表的插入操作
   ii. next()的源码实现也是从单链表中取出一个元素的操作，next()方法是一个无线循环的方法，如果消息队列中没有消息，那么next方法会一直阻塞在这里。当有新消息到来时，next()方法会返回这条消息并将其从单链表中移除。

10.2.3 Looper的工作原理

1. Looper在Android的消息机制中扮演着消息循环的角色，具体来说就是它会不停地从MessageQueue中查看是否有新消息，如果有新消息就会立即处理，否则就一直阻塞在那里。
2. 通过`Looper.prepare()`方法即可为当前线程创建一个Looper，再通过`Looper.loop()`开启消息循环。`prepareMainLooper()`方法主要给主线程也就是ActivityThread创建Looper使用的，本质也是通过prepare方法实现的。
3. Looper提供quit和quitSafely来退出一个Looper，区别在于quit会直接退出Looper，而quitSafely会把消息队列中已有的消息处理完毕后才安全地退出。 Looper退出后，这时候通过Handler发送的消息会失败，Handler的send方法会返回false。
4. 在子线程中，如果手动为其创建了Looper，在所有事情做完后，应该调用Looper的quit方法来终止消息循环，否则这个子线程就会一直处于等待状态；而如果退出了Looper以后，这个线程就会立刻终止，因此建议不需要的时候终止Looper。
5. loop()方法会调用MessageQueue的`next()`方法来获取新消息，而next是是一个阻塞操作，但没有信息时，next方法会一直阻塞在那里，这也导致loop方法一直阻塞在那里。如果MessageQueue的next方法返回了新消息，Looper就会处理这条消息：`msg.target.dispatchMessage(msg)`,这里的msg.target是发送这条消息的Handler对象，这样Handler发送的消息最终又交给Handler来处理了。

10.2.4 Handler的工作原理

Handler的工作主要包含消息的发送和接收过程。

通过post的一系列方法和send的一系列方法来实现。**Handler**发送过程仅仅是向消息队列中插入了一条消息。MessageQueue的next方法就会返回这条消息给Looper，Looper拿到这条消息就开始处理，最终消息会交给**Handler**的**dispatchMessage()**来处理，这时Handler就进入了处理消息的阶段。dispatchMessage()方法运行在Looper所在的线程上。

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        //Message的callback是一个Runnable,
       //也就是Handler的 post方法所传递的Runnable参数
        handleCallback(msg);
    } else {
        //如果给Handler设置了Callback的实现，
        //则调用Callback的handleMessage(msg)
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        //调用Handler的handleMessage方法来处理消息，
        //该Handler子类需重写handlerMessage(msg)方法
        handleMessage(msg);
    }
}
private static void handleCallback(Message message) {
//直接执行Runnable中的run()方法
    message.callback.run();
}
public interface Callback {
//不想继承Handler子类，可以通过Callback来实现handleMessage
    public boolean handleMessage(Message msg);
}
//默认空实现
public void handleMessage(Message msg) {
}
```

10.3 主线程的消息循环

1. Android的主线程就是ActivityThread，主线程的入口方法为 `main(String[] args)` ，在main方法中系统会通过Looper.prepareMainLooper()来创建主线程的Looper以及MessageQueue，并通过Looper.loop()来开启主线程的消息循环。

```
public static void main(String[] args) {
  ...
  Process.setArgV0("<pre-initialized>");

  Looper.prepareMainLooper();//创建主线程的Looper

  ActivityThread thread = new ActivityThread();
  thread.attach(false);

  if (sMainThreadHandler == null) {
      sMainThreadHandler = thread.getHandler();
  }

  AsyncTask.init();

  if (false) {
      Looper.myLooper().setMessageLogging(new
              LogPrinter(Log.DEBUG, "ActivityThread"));
  }

  Looper.loop();//开启looper

  throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

主线程的消息循环开始后，ActivityThread还需要一个Handler来和消息队列进行交互，这个Handler就是ActivityTread.H，它内部定义了一组消息类型，主要包含四大组件的启动和停止过程。

Android主线程的消息循环模型

ActivityThread通过ApplicationThread和AMS进行进程间通信，AMS以进程间通信的方式完成ActivityThread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H发送消息，H收到消息后会将ApplicationThread中的逻辑切换到ActivityTread中去执行，即切换到主线程中去执行。四大组件的启动过程基本上都是这个流程。

> 在简书上看到的一些思考： Looper.loop()，这里是一个死循环，如果主线程的Looper终止，则应用程序会抛出异常。那么问题来了，既然主线程卡在这里了，（1）那Activity为什么还能启动；（2）点击一个按钮仍然可以响应？ 问题1：startActivity的时候，会向AMS（ActivityManagerService）发一个跨进程请求（AMS运行在系统进程中），之后AMS启动对应的Activity；AMS也需要调用App中Activity的生命周期方法（不同进程不可直接调用），AMS会发送跨进程请求，然后由App的ActivityThread中的ApplicationThread会来处理，ApplicationThread会通过主线程线程的Handler将执行逻辑切换到主线程。重点来了，主线程的Handler把消息添加到了MessageQueue，Looper.loop会拿到该消息，并在主线程中执行。这就解释了为什么主线程的Looper是个死循环，而Activity还能启动，因为四大组件的生命周期都是以消息的形式通过**UI**线程的**Handler**发送，由**UI**线程的**Looper**执行的。 问题2：和问题1原理一样，点击一个按钮最终都是由系统发消息来进行的，都经过了Looper.loop()处理。 问题2详细分析请看原书作者的Android中MotionEvent的来源和ViewRootImpl。

11 Android的线程和线程池

1. 在Android系统，线程主要分为主线程和子线程，主线程处理和界面相关的事情，而子线程一般用于执行耗时操作。
2. 在Android中，线程的形态有很多种：
   i. AsyncTask封装了线程池和Handler。
   ii. HandlerThread是具有消息循环的线程，内部可以使用handler
   iii. IntentService是一种Service，内部采用HandlerThread来执行任务，当任务执行完毕后IntentService会自动退出。由于它是一种Service，所以不容易被系统杀死
3. 操作系统中，线程是操作系统调度的最小单元，同时线程又是一种受限的系统资源，其创建和销毁都会有相应的开销。同时当系统存在大量线程时，系统会通过时间片轮转的方式调度每个线程，因此线程不可能做到绝对的并发，除非线程数量小于等于**CPU**的核心数。
4. 频繁创建销毁线程不明智，使用线程池是正确的做法。线程池会缓存一定数量的线程，通过线程池就可以避免因为频繁创建和销毁线程所带来的系统开销。

11.1 主线程和子线程

1. 主线程主要处理界面交互逻辑，由于用户随时会和界面交互，所以主线程在任何时候都需要有较高响应速度，则不能执行耗时的任务；
2. android3.0开始，网络访问将会失败并抛出NetworkOnMainThreadException这个异常，这样做是为了避免主线程由于被耗时操作所阻塞从而现ANR现象

11.2 Android中的线程形态

11.2.1 AsyncTask

1. 三个参数（都可为Void)：
   i. Params：参数
   ii. Progress：执行进度
   iii. Result：返回值
2. 四个方法 ：
   i. `onPreExecute()` 主线程执行，异步方法执行前调用。
   ii. `doInBackground(Params...params)` 线程池中执行，用于执行异步任务；在方法内部用publishProgress 来更新任务进度。
   iii. `onProgressUpdate(Progress...value)` 主线程执行，后台任务进度状态改变时被调用。

iv. `onPostExecute(Result result)` 主线程执行，异步任务执行之后被调用

执行顺序： onPreExecute->doInBackground->onPostExecute 如果取消了异步任务，会回调onCancelled()，onPostExecute则不会被调用

AsyncTask的类必须在主线程加载，Android4.1及以上已经被系统自动完成了；**AsyncTask**对象必须在主线程创建；execute方法需要在UI线程调用；一个AsyncTask对象只能调用一次；Android1.6之前串行执行，Android1.6采用线程池并行处理任务，Android3.0开始，又采用一个线程来串行执行任务，但也可以通过 `executeOnExecutor()` 方法来并行执行任务。

11.2.2 AsyncTask的工作原理

1. AsyncTask中有两个线程池（ `SerialExecutor` 和 `THREAD_POOL_EXECUTOR` ）和一个 `InternalHandler` ，其中线程池SerialExecutor用于任务排队，THREAD_POOL_EXECUTOR用于真正执行任务，InternalHandler用于将执行环境切换到主线程。
2. AsyncTask的排队过程：系统首先会把AsyncTask的Params参数封装成FutureTask对象，它充当Runnable的作用，接下来这个FutureTask会交给SerialExecutor的 `execute()` 方法处理，execute()方法首先会把FutereTask对象插入到任务队列 `mTasks` 中去；如果没有正在活动的AsyncTask任务，就会执行下一个AsyncTask任务；同时当一个AsyncTask任务执行完成后，AsyncTask会继续执行其他任务直到所有任务都执行为止，可以看出默认情况，AsyncTask是串行执行的（Android3.0后）。

   AsyncTask工作的时序图后续补上。

11.2.3 HandlerThread

1. HandlerThread继承了Thread,是一种可以使用Handler的Thread
2. 在run方法中通过 `looper.prepare()` 来开启消息循环，这样就可以在HandlerThread中创建Handler了
3. 外界可以通过一个Handler的消息方式来通知HandlerThread来执行具体任务；确定不使用之后，可以通过 `quit` 或 `quitSafely` 方法来终止线程执行
4. 具体使用场景是IntentService

11.2.4 IntentService

IntentSercie是一种特殊的Service，继承了**Service**并且是抽象类，任务执行完成后会自动停止，优先级远高于普通线程，适合执行一些高优先级的后台任务； IntentService封装了 `HandlerThread` 和 `Handler`

1. `onCreate` 方法自动创建一个HandlerThread
2. 然后用它的Looper构造了一个Handler对象 `mServiceHandler` ，这样通过mServiceHandler

发送的消息都会在HandlerThread执行；

3. IntentServiced的 `onHandlerIntent` 方法是一个抽象方法，需要在子类实现，onHandlerIntent方法执行后，stopSelt(int startId)就会停止服务，如果存在多个后台任务，执行完最后一个**stopSelf(int startId)**才会停止服务。

11.3 Android线程池

优点：

1. 重用线程池的线程，减少线程创建和销毁带来的性能开销
2. 控制线程池的最大并发数，避免大量线程互相抢系统资源导致阻塞
3. 提供定时执行和间隔循环执行功能

11.3.1 ThreadPoolExecutor（熟悉后可自定义线程池）

Executor是一个接口，线程池的具体实现在ThreadPoolExecutor；它提供了一系列的参数来配置线程池；Android的线程池 大部分都是通 过Executor提供的工厂方法创建的

1. ThreadPoolExecutor常见构造参数

1. corePoolSize： 线程池的核心线程数，默认情况下，核心线程会一直存活(设置了超时机制除外， allowCoreThreadTimeOut属性为true时开启）
2. maxinmumPoolSize： 线程池能容纳的最大线程数，当活动的线程达到这个数值之后，后续新任务会被阻塞
3. keepAliveTime： 非核心线程闲置的超时时长，超过这个时长，非核心线程就会被回收,当allowCoreThreadTimeOut为true时，keepAliveTime同样作用于核心线程。
4. unit：keepAliveTime的时间单位，这是一个枚举，常用有TimeUnit.MILLISECONDS(毫秒)、TimeUnit.SECONDS（秒）、TimeUnit.MINUTES(分钟)
5. workQueue： 线程池中的任务队列，通过execute方法提交的Runnable对象会存储在这个参数中
6. threadFactory： 线程工厂，为线程池提供创建线程的功能，是个接口，提供Thread newThread(Runnable r)方法
7. RejectedExecutionHandle：当线程池无法执行新任务时，可能由于线程队列已满或无法成功执行任务，这时候 ThreadPoolExecutor会调用handler的 rejectedExecution的方法，默认会抛出RejectedExecutionException

2. ThreadPoolExecutor执行任务大致遵循如下规则:

1. 如果线程池中的线程数量未达到核心线程的数量，那么会直接启动一个核心线程来执行任务

2. 如果线程池中的线程数量已经达到或超过核心线程数量，那么任务会被插入到任务队列中排队等待执行
3. 如果步骤2中无法将任务插入到任务队列中，往往是因为任务队列已满，这个时候如果线程数量未达到线程池规定的最大值，那么会立刻启动一个非核心线程来执行任务
4. 如果步骤3中线程数量达到线程池规定的最大值，线程池会拒绝执行任务，并会调用RejectedExecutionHandler的rejectedExecution方法来通知调用者

3. AsyncTask的THREAD_POOL_EXECUTOR线程池配置:

1. 核心线程数等于CPU核心数+1
2. 线程池最大线程数为CPU核心数的2倍+1
3. 核心线程无超时机制，非核心线程的闲置超时时间为1秒
4. 任务队列容量是128

11.3.2 常见的4个线程池

1. `FixedThreadPool`：线程数量固定的线程池，当所有线程都处于活动状态时，新任务会处于等待状态，只有核心线程并且不会回收（无超时机制），能快速的响应外界请求。
2. `CachedThreadPool`：线程数量不定的线程池，最大线程数为Integer.MAX_VALUE(相当于任意大),当所有线程都处于活动状态时，会创建新线程来处理任务；线程池的空闲进程超时时长为60秒，超过就会被回收；任何任务都会被立即执行，适合执行大量的耗时较少的任务。
3. `ScheduledThreadPool`：核心线程数量固定，非核心线程数量无限制，非核心线程闲置时会被立刻回收，用于执行定时任务和具有固定周期的重复任务。
4. `SingleThreadExecutor`：只有一个核心线程，所有任务都在这个线程中串行执行，不需要处理线程同步问题

12 Bitmap的加载和Cache

主要介绍：

1. 如何高效地加载一个Bitmap
2. Android中常用的缓存策略
    i. LruCache——内存缓存
    ii. DiskLruCache——磁盘缓存
3. 如何优化列表的卡顿

12.1 Bitmap的高效加载

1. BitmapFactory类提供四种方法：`decodeFile`、`decodeResource`、`decodeStream`和`decodeByteArray`；其中decodeFile和decodeResource间接的调用了decodeStream方法；这四个方法最终在Android底层实现，对应着BitmapFactory类的几个native方法。
2. 如何高效的加载Bitmap？核心思想：按需加载；很多时候ImageView并没有原始图片那么大，所以没必要加载原始大小的图片。采用`BitmapFactory.Options`来加载所需尺寸的图片。 通过BitmapFactory.Options来缩放图片，主要是用到了它的`inSampleSize`参数，即采样率。 inSampleSize应该为2的倍数，如果不是系统会向下取整并选择一个最接近2的指数来代替；缩放比例为1/（inSampleSize的二次方）。
3. Bitmap内存占用：拿一张10241024像素的图片来说，假定采用ARGB8888格式存储，那么它占用的内存为10241024*4，即4MB。
4. 通过采样率高效地加载图片，代码示例：

```
public static Bitmap decodeBitmapFromResource(Resources res, int resId, int reqWid
th, int reqHeight) {
  BitmapFactory.Options options = new BitmapFactory.Options();
  //1. 将BitmapFactory.Options的inJustDecodeBounds参数设置为true并加载图片。
  options.inJustDecodeBounds = true;
  BitmapFactory.decodeResource(res, resId, options);

  //2. 根据采样率的规则并结合目标View的所需大小计算出采样率inSampleSize。
  options.inSampleSize = calcuateInSampleSize(options, reqWidth, reqHeight);

  //3. 将BitmapFactory.Options的inJustDecodeBounds参数设置为false，然后重新加载图片。
  options.inJustDecodeBounds = false;
  return BitmapFactory.decodeResource(res, resId, options);
}
//获取采样率
private static int calcuateInSampleSize(BitmapFactory.Options options, int reqWidt
h, int reqHeight) {
  int width = options.outWidth;
  int height = options.outHeight;
  int inSampleSize = 1;

  if (height > reqHeight || width > reqWidth) {
      int halfHeight = height / 2;
      int halfWidth = width / 2;
      while ((halfHeight / inSampleSize) >= reqHeight && (halfWidth / inSampleSize
) >= reqWidth) {
      //inSampleSize应该为2的倍数
          inSampleSize *= 2;
      }
  }
  return inSampleSize;
}
// 显示图片
Bitmap bitmap = DecodeBitmap.decodeBitmapFromResource(getResources(), R.mipmap.hai
mei2, 400, 400);
imageView.setImageBitmap(bitmap);
```

当inJustDecodeBounds参数为true时，BitmapFactory只会解析图片的原始宽/高信息，并不会真正的加载图片，所以这个操作是轻量级的。需要注意这时候BitmapFactory获取的图片宽/高信息和图片的位置与程序运行的设备有关。

12.2 Android中的缓存策略

1. 如何减少流量消耗？缓存。当程序第一次从网络上加载图片后，将其缓存在存储设备中，下次使用这张图片的时候就不用再从网络从获取了。一般情况会把图片存一份到内存中，一份到存储设备中，如果内存中没找到就去存储设备中找，还没有找到就从网络上下载。

2. 目前常用的缓存算法是LRU，是近期最少使用算法，当缓存满时，优先淘汰那些近期最少使用的缓存对象。采用LRU算法的缓存有两种：`LruCache`（内存缓存）和`DiskLruCache`（存储缓存）。

12.2.1 LruCache

1. LruCache是Android3.1所提供的一个缓存类，通过support-v4兼容包可以兼容到早期的Android版本。
2. LruCache是一个泛型类，是线程安全的，内部采用LinkedHashMap以强引用的方式存储外界缓存对象
3. 提供`get(String key)`和`put(String key , V value)`方法来完成缓存的获取和添加操作，当缓存满时，LruCache会移除较早的使用的缓存对象
4. LruCache初始化时需重写`sizeOf(String key ,V value)`方法，用于计算缓存对象的大小。

> 强/软/弱引用的概念：
>
> - 强引用：直接的对象引用
> - 软引用：当一个对象只有软引用存在的时候，系统内存不足的时此对象会被GC回收。
> - 弱引用：当一个对象只有弱引用存在的时候，此对象随时会被GC回收。

12.2.2 DiskLruCache

DiskLruCache用于实现磁盘缓存，DiskLruCache得到了Android官方文档推荐，但它不属于Android SDK的一部分，源码在这里。

1. DiskLruCache不能通过构造方法来创建，提供了`open()`方法来创建自身。
2. 通过`edit(String key)`方法来获取Editor对象
3. 通过Editor的`newOutputStream(int cacheIndex)`方法打开一个文件输出流，然后通过这个流将下载的图片写入到文件系统中。
4. 调用Editor的`commit()`来提交写入操作。如果图片下载过程中发生异常，通过Editor的`abort()`来回退整个操作。

DiskLruCache的缓存查找：

1. 将图片url转换为key，通过`get(String key)`方法得到一个Snapshot对象。
2. 通过Snapshort对象即可得到缓存的文件输入流
3. 从输入流中得到Bitmap对象
4. 通过`BitmapFactory.Options`对象来加载一张缩放后的图片，对FileInputStream的缩放存在问题，因为FileInputStream是一种有序的文件流，而两次`decodeStream`调用影响了文件流的位置属相，导致第二次`decodeStream`时得到的是null。所以一般通过文件流来得到对应的文件描述符，通过`BitmapFactory.decodeFileDescriptor()`来加载一张缩放后的图

片。

12.2.3 ImageLoader的实现

1. 图片压缩功能
2. 内存缓存和磁盘缓存
3. 同步加载和异步加载的接口设计 详细请看随书源码

12.3 ImageLoader的使用

12.3.1 照片墙效果

实现照片墙效果，如果图片都需要是正方形；这样做很快，自定义一个ImageView，重写onMeasure方法。

```
    @Override
    protected void onMeasure(int widthMeasureSpec,int heightMeasureSpec){
       super.onMeasure(widthMeasureSpec,widthMeasureSpec);//将原来的参数heightMeasureSpe
c换成widthMeasureSpec
    }
```

12.3.2 优化列表的卡顿现象

1. 不要在getView中执行耗时操作，不要在getView中直接加载图片。
2. 控制异步任务的执行频率：如果用户刻意频繁上下滑动，getView方法会不停调用，从而产生大量的异步任务。可以考虑在列表滑动停止加载图片；给ListView或者GridView设置 `setOnScrollListener` 并在 `OnScrollListener` 的 `onScrollStateChanged` 方法中判断列表是否处于滑动状态，如果是的话就停止加载图片。
3. 大部分情况下，可以使用硬件加速解决莫名卡顿问题，通过设置 `android:hardwareAccelerated="true"` 即可为Activity开启硬件加速。

13 综合技术

本章主要讲解:

1. CrashHandler来监视App的crash信息
2. 通过Google的multiDex方案解决Android方法数超过65536的问题
3. Android动态加载
4. 反编译

13.1 使用CrashHandler来获取应用的crash信息

如何检测崩溃并了解详细的crash信息？首先需实现一个uncaughtExceptionHandler对象，在它的uncaughtException方法中获取异常信息并将其存储到SD卡或者上传到服务器中，然后调用Thread的setDefaultUncaughtExceptionHandler为当前进程的所有线程设置异常处理器。

```
public class CrashHandler implements Thread.UncaughtExceptionHandler {
    private static final String TAG = "CrashHandler";
    private static final boolean DEBUG = true;

    private static final String PATH = Environment.getExternalStorageDirectory().getPa
th() + "/CrashTest/log/";
    private static final String FILE_NAME = "crash";
    private static final String FILE_NAME_SUFFIX = ".trace";

    private static CrashHandler sInstance = new CrashHandler();
    private Thread.UncaughtExceptionHandler mDefaultCrashHandler;
    private Context mContext;

    private CrashHandler() {
    }

    public static CrashHandler getInstance() {
        return sInstance;
    }

    public void init(Context context) {
        mDefaultCrashHandler = Thread.getDefaultUncaughtExceptionHandler();
        Thread.setDefaultUncaughtExceptionHandler(this);
        mContext = context.getApplicationContext();
    }

    /**
     * 这个是最关键的函数，当程序中有未被捕获的异常，系统将会自动调用#uncaughtException方法
     * thread为出现未捕获异常的线程，ex为未捕获的异常，有了这个ex，我们就可以得到异常信息。
     */
```

```
    @Override
    public void uncaughtException(Thread thread, Throwable ex) {
        try {
            //导出异常信息到SD卡中
            dumpExceptionToSDCard(ex);
            uploadExceptionToServer();
            //这里可以通过网络上传异常信息到服务器，便于开发人员分析日志从而解决bug
        } catch (IOException e) {
            e.printStackTrace();
        }

        ex.printStackTrace();

        //如果系统提供了默认的异常处理器，则交给系统去结束我们的程序，否则就由我们自己结束自己
        if (mDefaultCrashHandler != null) {
            mDefaultCrashHandler.uncaughtException(thread, ex);
        } else {
            Process.killProcess(Process.myPid());
        }
    }

    private void dumpExceptionToSDCard(Throwable ex) throws IOException {
        //伪代码 本方法用于实现将错误信息存储到SD卡中
    }



    private void uploadExceptionToServer() {
        //伪代码 本方法用于将错误信息上传至服务器
    }
}
```

然后在Application初始化的时候为线程设置CrashHandler，这样之后，Crash就会通过我们自己的异常处理器来处理异常了。

```
public class BaseApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        CrashHandler crashHandler = CrashHandler.getInstance();
        crashHandler.init(this);
    }
}
```

书中附带源码

13.2 使用multidex来解决方法数越界

1. 在Android中单个dex文件所能包含的最大方法数为65536，这个是包含Android FrameWork、依赖jar包以及应用本身代码中的所有方法。达到这个65536后，编译器编译时会抛出DexIndexOverflowException异常。
2. 如何解决？Google提供了multidex解决方案。在Android5.0之前需要引入Google提供的android-support-multidex.jar；从5.0开始系统默认支持了multidex，它可以从apk文件中加载多个dex文件。
3. 使用步骤：

    i. 修改对应工程目录下的build.gradle文件，在defaultConfig中添加multiDexEnabled true这个配置项。

    ii. 在build.gradle的dependencies中添加multidex的依赖：compile 'com.android.support:multidex:1.0.0'

```
android {
 compileSdkVersion 23
 buildToolsVersion "23.0.3"
 defaultConfig {
   applicationId "cn.hudp.androiddevartnote"
   minSdkVersion 14
   targetSdkVersion 23
   versionCode 1
   versionName "1.0"
   multiDexEnabled true  //关键部分
 }
 buildTypes {
   release {
       minifyEnabled false
       proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguar
d-rules.pro'
   }
 }
 }
 dependencies {
 compile fileTree(dir: 'libs', include: ['*.jar'])
 compile 'com.android.support:appcompat-v7:23.4.0'
 compile 'com.android.support:multidex:1.0.0' //关键部分
}
```

    iii. 代码中加入支持multidex功能。

        i. 第一种方案，在manifest文件中指定Application为MultiDexApplication。

        ii. 第二种方案，让应用的Application继承MultiDexApplication。

        iii. 第三种方案，重写 `attachBaseContext` 方法，这个方法比onCreate还要先执行。

```
public class BaseApplication extends Application {
 @Override
 protected void attachBaseContext(Context base) {
     super.attachBaseContext(base);
     MultiDex.install(this);
 }
}
```

采用上面的配置项后，如果这个应用方法数没有越界，那么**Gradle**是不会生成多个**dex**文件的，当方法数越界后，Gradle就会在apk中打包2个或多个dex文件。当需要指定主**dex**文件中所包含的类，这时候就需要通过**--multi-dex-list**来选项来实现这个功能。

```
//在对应工程目录下的build.gradle文件，加入
afterEvaluate {
println "afterEvaluate"
tasks.matching {
it.name.startsWith('dex')
}.each { dx ->
def listFile = project.rootDir.absolutePath + '/app/maindexlist.txt'
println "root dir:" + project.rootDir.absolutePath
println "dex task found: " + dx.name
if (dx.additionalParameters == null) {
   dx.additionalParameters = []
}
dx.additionalParameters += '--multi-dex'
dx.additionalParameters += '--main-dex-list=' + listFile
dx.additionalParameters += '--minimal-main-dex'
}
}
//maindexlist.txt
com/ryg/multidextest/TestApplication.class
com/ryg/multidextest/MainActivity.class
// multidex 这9个类必须在主Dex中
android/support/multidex/MultiDex.class
android/support/multidex/MultiDexApplication.class
android/support/multidex/MultiDexExtractor.class
android/support/multidex/MultiDexExtractor$1.class
android/support/multidex/MultiDex$V4.class
android/support/multidex/MultiDex$V14.class
android/support/multidex/MultiDex$V19.class
android/support/multidex/ZipUtil.class
android/support/multidex/ZipUtil$CentralDirectory.class
```

需要注意**multidex**的**jar**中的**9**个类必须要打包到主**dex**中，因为**Application**的**attachBaseContext**方法中需要用到**MultiDex.install(this)**需要用到**MultiDex**。

Multidex的缺点：

1. 启动速度会降低，由于应用启动时会加载额外的dex文件，这将导致应用的启动速度降低，甚至产生ANR现象。
2. 因为Dalvik linearAlloc的bug，可以导致使用multidex的应用无法在Android4.0之前的手机上运行，需要做大量兼容性测试。

13.3 Android动态加载技术

各种插件化方案都需要解决3个基础性问题：

1. 资源访问，因为插件中凡是以R开头的资源文件都不能访问了。
2. Activity的生命周期管理，因为宿主动态将Activity.java加载到内存的时候，是不具备Activity的任何特性的，只是一个普通的java类。
3. ClassLoader的管理，为了避免多个ClassLoader加载了同一个类所引发的类型转换错误。

需要熟悉一下作者的插件化开源框架：dynamic-load-apk

13.4 反编译初步

1. 使用**dex2jar**和**jd-gui**反编译apk
2. 使用**apktool**对apk进行二次打包

以上网上资料特别多，不赘述。

14 JNI与NDK编程

Java的跨平台特性导致其本地交互的能力不够强大，一些和操作系统相关的特性Java无法完成，于是Java提供了JNI专门用于和本地代码交互。通过Java JNI，用户可以调用C、C++所编写的本地代码。

NDK是Android所提供的一个工具集合，通过NDK可以再Android中更加方便地通过JNI来访问本地代码（C/C++）。NDK还提供了交叉编译器，开发人员只需要简单修改**mk**文件就可以生成特定CPU平台的动态库。NDK的用途和优点：

1. 代码的保护。由于apk的java层代码很容易被反编译，而C/C++库反汇难度较大。
2. 可以方便地使用C/C++开源库。
3. 便于移植，用C/C++写的库可以方便在其他平台上再次使用
4. 提供程序在某些特定情形下的执行效率，但是并不能明显提升Android程序的性能。

14.1 JNI的开发流程

1. 在Java中声明natvie方法
2. 用C/C++实现natvie方法
3. 编译运行

14.2 NDK的开发流程

1 下载并配置NDK

下载好NDK开发包，并且配置好NDK的全局变量。

2 创建一个Android项目，并声明所需的native方法

```
public static native String getStringFromC();
```

3 实现Android项目中所声明的native方法

1. 生成C/C++的头文件

    i. 打开控制台，用cd命令切换到当前项目当前目录
    ii. 使用javah命令生成头文件

```
javah -classpath bin\classes;C:\MOX\AndroidSDK\platforms\android-23\android.j
ar -d jni cn.hudp.hellondk.MainActivity
```

> 说明：bin\classes 为项目的class文件的相对路径 ；C:\MOX\AndroidSDK\platforms\android-23\android.jar 为android.jar的全路径，因为我们的的Activity使用到了Android SDK，所以生成头文件时需要他; -d jni就是生成的头文件输出到项目的jni文件夹下； 最后跟的cn.hudp.hellondk.MainActivity是native方法所在的类的包名和类名。

2. 编写修改对应的android.mk文件（**mk**文件是**NDK**开发所用到的配置文件）

```
# Copyright (C) 2009 The Android Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
## **看这里看这里** 对应Java部分 System.loadLibrary(String libName) 的libname
LOCAL_MODULE    := hello
## **看这里看这里** 对应c/c++的实现文件名，请看图2
LOCAL_SRC_FILES := hello.c
include $(BUILD_SHARED_LIBRARY)
```

3. 编写Application.mk，来指定需生成的平台对应的动态库，这里是全平台支持，也可以特殊指定。 `APP_ABI := all`

4 切换到jni目录的父目录，然后通过ndk-build命令编译产生so库

1. `ndk-build` 命令会默认指定jni目录为本地源码的目录
2. 将编译好的so库放到Android项目中的 `app/src/main/jniLbis` 目录下，或者通过如下**app**的**gradle**设置新的存放**so**库的目录：

```
android{
    ……
    sourceSets.main{
        jniLibs.srcDir 'src/main/jni_libs'
    }
}
```

还可以通过 `defaultConfig` 区域添加NDK选项

```
android{
    ……
    defaultConfig{
        ……
        ndk{
            moduleName "jni-test"
        }
    }
}
```

还可以在 `productFlavors` 设置动态打包不同平台CPU对应的so库进apk（缩小**APK**体积）

```
```gradle
android{
    ……
    productFlavors{
        arm{
            ndk{
                adiFilter "armeabi"
            }
        }
        x86{
            ndk{
                adiFilter "x86"
            }
        }
    }
}
```
```

5 在Android中调用

```
public class MainActivity extends Activity {
    public static native String getStringFromC();
    static{//在静态代码块中调用所需要的so文件，参数对应.so文件所对应的LOCAL_MODULE；
        System.loadLibrary("hello");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //在需要的地方调用native方法
        Toast.makeText(getApplicationContext(), get(), Toast.LENGTH_LONG).show();
    }
}
```

15 Android性能优化

Android设备作为一种移动设备，不管是内存还是CPU的性能都受到了一定的限制，也意味着Android程序不可能无限制的使用内存和CPU资源，过多的使用内存容易导致OOM，过多的使用CPU资源容易导致手机变得卡顿甚至无响应（ANR）。这也对开发人员提出了更高的要求。 本章主要介绍一些有效的性能优化方法。主要包括布局优化、绘制优化、内存泄漏优化、响应速度优化、ListView优化、Bitmap优化、线程优化等；同时还介绍了ANR日志的分析方法。

Google官方的Android性能优化典范专题短视频课程是学习Android性能优化极佳的课程，目前已更新到第五季； youku地址

1. 布局优化的思想就是尽量减少布局文件的层级，这样绘制界面时工作量就少了，那么程序的性能自然就高了。
   i. 删除无用的控件和层级
   ii. 其次就是有选择的使用性能较低的ViewGroup，如果布局中既可以使用**Linearlayout**也可以使用**RelativeLayout**，那就是用**LinearLayout**，因为RelativeLayout功能比较复杂，它的布局过程需要花费更多的CPU时间。
   iii. 有时候通过LinearLayou无法实现产品效果，需要通过嵌套来完成，这种情况还是推荐使用**RelativeLayout**，因为ViewGroup的嵌套相当于增加了布局的层级，同样降低程序性能。
2. 另一种手段是采用标签、标签和ViewStub。

   i. **include**标签

   `<include>` 标签用于布局重用，可以将一个指定的布局文件加载到当前布局文件中。 `<include>` 只支持android:layout开头的属性，当然*android:id*这个属性是个特例；如果指定了*android:layout*这种属性，那么要求android:layout*width*和*android:layout_height*必须存在，否则*android:layout*属性无法生效。如果 `<include>` 指定了id属性，同时被包含的布局文件的根元素也指定了id属性，会以 `<include>` 指定的这个id属性为准。

   ii. **merge**标签

   `<merge>` 标签一般和 `<include>` 标签一起使用从而减少布局的层级。如果当前布局是一个竖直方向的LinearLayout，这个时候被包含的布局文件也采用竖直的LinearLayout，那么显然被包含的布局文件中的这个LinearLayout是多余的，通过 `<merge>` 标签就可以去掉多余的那一层LinearLayout。

   iii. **ViewStub**

ViewStub意义在于按需加载所需的布局文件，因为实际开发中，有很多布局文件在正常情况下是不会现实的，比如网络异常的界面，这个时候就没必要在整个界面初始化的时候将其加载进来，在需要使用的时候再加载会更好。在需要加载ViewStub布局时：

```
((ViewStub)findViewById(R.id.stub_import)).setVisibility(View.VISIBLE);
//或者
View importPanel = ((ViewStub)findViewById(R.id.stub_import)).inflate();
```

当ViewStub通过setVisibility或者inflate方法加载后，ViewStub就会被它内部的布局替换掉，ViewStub也就不再是整个布局结构的一部分了。

15.1.2 绘制优化

View的onDraw方法要避免执行大量的操作；

1. onDraw中不要创建大量的局部对象，因为onDraw方法会被频繁调用，这样就会在一瞬间产生大量的临时对象，不仅会占用过多内存还会导致系统频繁GC，降低程序执行效率。
2. onDraw也不要做耗时的任务，也不能执行成千上万的循环操作，尽管每次循环都很轻量级，但大量循环依然十分抢占CPU的时间片，这会造成View的绘制过程不流畅。根据Google官方给出的标准，View绘制保持在60fps是最佳的，这也就要求每帧的绘制时间不超过**16ms(1000/60)**；所以要尽量降低onDraw方法的复杂度。

15.1.3 内存泄露优化

内存泄露是最容易犯的错误之一，内存泄露优化主要分两个方面；一方面是开发过程中避免写出有内存泄露的代码，另一方面是通过一些分析工具如**LeakCanary**或**MAT**来找出潜在的内存泄露继而解决。

1. 静态变量导致的内存泄露 比如Activity内，一静态Conext引用了当前Activity，所以当前Activity无法释放。或者一静态变量，内部持有了当前Activity，Activity在需要释放的时候依然无法释放。
2. 单例模式导致的内存泄露 比如单例模式持有了Activity，而且也没用解注册的操作。因为单例模式的生命周期和Application保存一致，生命周期比Activity要长，这样一来就导致Activity对象无法及时被释放。
3. 属性动画导致的内存泄露 属性动画中有一类无限循环的动画，如果在Activity播放了此类动画并且没有在onDestroy中去停止动画，那么动画会一直播放下去，并且这个时候Activity的View会被动画持有，而View又持有了Activity，最终导致Activity无法释放。解决办法是在Activity的onDrstroy中调用animator.cancel()来停止动画。

15.1.4 响应速度优化和ANR日志分析

响应速度优化的核心思想就是避免在主线程中去做耗时操作，将耗时操作放在其他线程当中去执行。Activity如果**5**秒无法响应屏幕触摸事件或者键盘输入事件就会触发ANR，而B**roadcastReceiver**如果**10**秒还未执行完操作也会出现**ANR**。

当一个进程发生ANR以后系统会在**/data/anr**的目录下创建一个文件**traces.txt**，通过分析该文件就能定位出ANR的原因。

15.1.5 ListView优化和Bitmap优化

ListView/GridView优化：

1. 采用ViewHolder避免在getView中执行耗时操作
2. 其次通过列表的滑动状态来控制任务的执行频率，比如快速滑动时不是和开启大量异步任务
3. 最后可以尝试开启硬件加速使得ListView的滑动更加流畅。

Bitmap优化：主要是想是根据需要对图片进行采样显示，详细请参考12章。

15.1.6 线程优化

线程优化的思想是采用线程池，避免程序存在大量的Thread。详细参考第11章的内容。

15.1.7 一些性能优化的小建议

1. 避免创建过多的对象，尤其在循环、onDraw这类方法中，谨慎创建对象；
2. 不要过多的使用枚举，枚举占用的内存空间比整形大。
3. 常量使用static final来修饰；
4. 使用一些Android特有的数据结构，比如 `SparseArray` 和 `Pair` 等，他们都具有更好的性能；
5. 适当的使用软引用和弱引用；
6. 采用内存缓存和磁盘缓存；
7. 尽量采用静态内部类，这样可以避免非静态内部类隐式持有外部类所导致的内存泄露问题。

15.3 提高程序的可维护性

提高可读性：

1. 命名规范
2. 代码之间排版需留出合理的空白来区分不同的代码块
3. 针对非常关键的代码添加注释。

代码的层级性

1. 不要把一段业务逻辑放在一个方法或者一个类中全部实现，要把它分成几个子逻辑，然后每个子逻辑做自己的事情，这样即显得代码层级分明，这样利于提高程序的可扩展性。
2. 恰当的使用设计模式可以提高代码的可维护性和可扩展性，Android程序容易遇到性能瓶颈，要控制设计的度，不能太牵强，避免过度设计。作者推荐查看《大话设计模式》和《**Android**源码设计模式解析和实战》这两本书来学习设计模式。