1.ListView与RecyclerView的区别
2.RecyclerView的拖拽怎么实现?
3.ListView回收机制
4.MVP和MVVM的区别
5.AsyncTask内部实现原理
    1) AsyncTask的使用简介
    2) AsyncTask的局限性
    3) AsyncTask的工作原理
    4) AsyncTask各个方法的作用
6.service两种启动方式有什么区别?
    1）start启动方式：
    2）bind启动方式：
7.说说图片三级缓存
    1）为什么要三级缓存?
    2）什么事三级缓存?
    3）图片异步加载缓存方案的工作流程
    5）设计三级缓存
    6）LruCache 源码解析
        1. 简介
        2. 使用方法
        3. 效果展示
        4. 源码分析
            4.1 LruCache 原理概要解析
            4.2 LruCache 的唯一构造方法
            4.3 LruCache.get(K key)
            4.4 LinkedHashMap.get(Object key)
            4.5 LinkedHashMap.makeTail(LinkedEntry<K, V> e)
            4.6 LruCache.put(K key, V value)
            4.7 LruCache.trimToSize(int maxSize)
            4.8 覆写 entryRemoved 的作用
            4.9 LruCache 局部同步锁
        5. 开源项目中的使用
        6. 总结
    7）DiskLruCache 源码解析
        （1）存储位置
        （4）打开缓存
        （3）写入缓存
        （4）读取缓存
        （5）移除缓存
        （6）其它API
        （7）解读journal
8.Handler机制
    （1）消息类：Message类
    （2）消息通道：Looper
    （3）消息操作类：Handler类
    （4）Android 运行的进程
    (5) Handlers

(6) Message
8.ListView图片加载错乱的原理和解决方案
9.数据库的操作类型有哪些，如何导入外部数据库?
10.是否使用过本地广播，和全局广播有什么差别?
11.是否使用过 IntentService，作用是什么， AIDL 解决了什么问题? (小米)
12.Binder机制
13.屏幕适配的处理技巧都有哪些?
(1) 相关重要概念
(2) 为什么要进行Android屏幕适配
(3) 屏幕适配问题的本质
(4) 解决方案
1) “布局”匹配
设想这么一个场景
2) “布局组件”匹配
3) “图片资源”匹配
4) ”用户界面流程“匹配

1.ListView与RecyclerView的区别

链接： https://www.jianshu.com/p/f592f3715ae2

从以下三点切入：

1）布局效果对比

2）常用功能与API对比

3）在`Android` L引入嵌套滚动机制 `（NestedScrolling）`

`ListView`与`RecyclerView`的简单使用：

ListView：

1. 继承重写`BaseAdapter`类；
2. 自定义`ViewHolder`与`convertView`的优化（判断是否为null）；

   RecyclerView：
3. 继承重写`RecyclerView.Adapter`与`RecyclerView.ViewHolder`
4. 设置`LayoutManager`，以及`layout`的布局效果

   区别：
5. `ViewHolder`的编写规范化，`ListView`是需要自己定义的，而`RecyclerView`是规范好的；
6. `RecyclerView`复用`item`全部搞定，不需要像`ListView`那样`setTag()`与`getTag()`；
7. `RecyclerView`多了一些`LayoutManager`工作，但实现了布局效果多样化；

   布局效果：

`ListView` 的布局比较单一，只有一个纵向效果；

`RecyclerView` 的布局效果丰富， 可以在`LayoutMananger`中设置：线性布局（纵向，横向），表格布局，瀑布流布局，在`RecyclerView` 中，如果存在的`LayoutManager`不能满足需求，可以在`LayoutManager`的API中自定义`Layout`：

例如：`scrollToPosition()，setOrientation()，getOrientation()，findViewByPosition()`等等；

空数据处理：

在`ListView`中有个`setEmptyView()` 用来处理`Adapter`中数据为空的情况；但是在`RecyclerView`中没有这个API，所以在`RecyclerView`中需要进行一些数据判断来实现数据为空的情况；

HeaderView 与 FooterView：

在`ListView`中可以通过`addHeaderView()` 与 `addFooterView()`来添加头部`item`与底部 `item`，当我们需要实现的下拉刷新或者上拉加载的情况；而且这两个API不会影响 `Adapter` 的编写；但是`RecyclerView`中并没有这两个API，所以当我们需要在`RecyclerView`添加头部`item`或者底部 `item` 的时候，我们可以在 `Adapter` 中自己编写，根据 `ViewHolder` 的 `Type` 与 `View` 来实现自己的 `Header，Footter` 与普通的 `item`，但是这样就会影响到 `Adapter` 的数据，比如 `position`，添加了 `Header` 与 `Footter` 后，实际的 `position` 将大于数据的`position`；

局部刷新：

在`ListView`中通常刷新数据是用 `notifyDataSetChanged()` ，但是这种刷新数据是全局刷新的（每个item的数据都会重新加载一遍），这样一来就会非常消耗资源；`RecyclerView` 中可以实现局部刷新，例如：`notifyItemChanged()`；

但是如果要在 `ListView` 实现局部刷新，依然是可以实现的，当一个 `item` 数据刷新时，我们可以在`Adapter`中，实现一个`onItemChanged()`方法，在方法里面获取到这个 `item` 的 `position`（可以通过`getFirstVisiblePosition()`），然后调用 `getView()` 方法来刷新这个 `item` 的数据；

动画效果：

在 `RecyclerView` 中，已经封装好API来实现自己的动画效果；有许多动画API，例如：`notifyItemChanged()，notifyDataInserted()，notifyItemMoved()`等等；如果我们需要实现自己的动画效果，我们可以通过相应的接口实现自定义的动画效果（`RecyclerView.ItemAnimator`类），然后调用`RecyclerView.setItemAnimator()`(默认的有`SimpleItemAnimator`与`DefaultItemAnimator`)；但是ListView并没有实现动画效果，但我们可以在`Adapter` 自己实现item的动画效果；

ItemTouchHelper：

创建`ItemTouchHelper`实例，然后在`ItemTouchHelper.SimpleCallback()`，然后在`Callback`中实现`getMovementFlags()，onMove()，onSwiped()`， 最后调用`RecyclerView`的`attachToRecyclerView`方法；提供的滑动和删除`Item`的工具类。

Item点击事件：

在ListView中有`onItemClickListener(), onItemLongClickListener(), onItemSelectedListener(),` 但是添加HeaderView与FooterView后就不一样了，因为HeaderView与FooterView都会算进position中，这时会发现position会出现变化，可能会抛出数组越界，为了解决这个问题，我们在getItemId()方法（在该方法中HeaderView与FooterView返回的值是-1）中通过返回id来标志对应的item，而不是通过position来标记；但是我们可以在Adapter中针对每个item写在getView()中会比较合适；而在RecyclerView中，提供了唯一一个API：addOnItemTouchListener()，监听item的触摸事件；我们可以通过RecyclerView的addOnItemTouchListener()加上系统提供的GestureDetector来实现像ListView那样监听某个item某个 操作方法；

嵌套滚动机制：

在事件分发机制中，Touch事件在进行分发的时候，由父View向子View传递，一旦子View消费这个事件的话，那么接下来的事件分发的时候，父View将不接受，由子View进行处理；但是与Android的事件分发机制不同，嵌套滚动机制（Nested Scrolling）可以弥补这个不足，能让子View与父View同时处理这个Touch事件，主要实现在于NestedScrollingChild与NestedScrollingParent这两个接口；而在RecyclerView中，实现的是NestedScrollingChild，所以能实现嵌套滚动机制；

ListView就没有实现嵌套滚动机制；

总结：

这里只是客观的分析ListView与RecyclerView的差异，而在实际场景中，我们应该根据自己的需求来选择使用RecyclerView还是ListView，毕竟，适合业务需求的才是最好的。

2.RecyclerView的拖拽怎么实现?

ItemTouchHelper是support v7包提供的处理关于在RecyclerView上添加拖动排序与滑动删除的非常强大的工具类。它是RecyclerView.ItemDecoration的子类，也就是说它可以轻易的添加到几乎所有的LayoutManager和Adapter中。见`DefaultItemTouchHelpCallback.java`文件

具体说一下`ItemTouchHelper.Callback`这个抽象类：

`getMovementFlags()`

用于设置是否处理拖拽事件和滑动事件，以及拖拽和滑动操作的方向，有以下两种情况：

如果是列表类型的，拖拽只有`ItemTouchHelper.UP、ItemTouchHelper.DOWN`两个方向

如果是网格类型的，拖拽则有`UP、DOWN、LEFT、RIGHT`四个方向

另外，滑动方向列表类型的，有START和END两个方法，如果是网格类型的一般不设置支持滑动操作可以将swipeFlags = 0置为0，表示不支持滑动操作！

最后，需要调用`return makeMovementFlags(dragFlags, swipeFlags)`;将设置的标志位return回去！

`onMove()`

如果我们设置了相关的dragFlags ，那么当我们长按item的时候就会进入拖拽并在拖拽过程中不断回调onMove()方法,我们就在这个方法里获取当前拖拽的item和已经被拖拽到所处位置的item的ViewHolder。

`onSwipe()`

如果我们设置了相关的swipeFlags，那么当我们滑动item的时候就会调用onSwipe()方法，一般的话在使用LinearLayoutManager的时候，在这个方法里可以删除item，来实现滑动删除！

就是说，如果我们不重写这两个方法，那么拖拽和滑动都是默认开启的，如果需要我们自定义拖拽和滑动，可以设置为false，然后调用startDrag()和startSwipe()方法来开启！

还有两个方法，可以使用户交互更加友好：

1) `public void onSelectedChanged(RecyclerView.ViewHolder viewHolder, int actionState)`

这个方法在选中Item的时候（拖拽或者滑动开始的时候）调用，通常这个方法里我们可以改变选中item的背景颜色等，高亮表示选中来提高用户体验。

需要注意的是，这里的第二个参数int actionState，它有以下3个值，分别表示3种状态：

ACTION_STATE_IDLE：闲置状态

ACTION_STATE_SWIPE：滑动状态

ACTION_STATE_DRAG：拖拽状态

我们可以根据这个状态值，作不同的逻辑处理！

2) `public void clearView(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder)`

这个方法在当手指松开的时候（拖拽或滑动完成的时候）调用，这时候我们可以将item恢复为原来的状态。

最后在代码中：

```
ItemTouchHelper itemTouchHelper = new ItemTouchHelper(new
SimpleItemTouchCallback(data, adapter));

itemTouchHelper.attachToRecyclerView(recyclerView);
```

3.ListView回收机制

链接：http://www.cnblogs.com/qiengo/p/3628235.html#_Toc383693206

4.MVP和MVVM的区别

链接：**https://www.cnblogs.com/dubo-/p/5619077.html**

https://blog.csdn.net/victoryzn/article/details/78392128

https://www.jianshu.com/p/ebd2c5914d20

MVC MVC模式最初生根于服务器端的Web开发，后来渐渐能够胜任客户端Web开发，能够满足其复杂性和丰富性。

MVC是Model-View-Controller的缩写，它将应用程序划分为三个部分：

Model: 模型（用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法）

View: 视图（渲染页面）

Controller: 控制器（M和V之间的连接器，用于控制应用程序的流程，及页面的业务逻辑）

MVC特点：

MVC模式的特点在于实现关注点分离，即应用程序中的数据模型与业务和展示逻辑解耦。在客户端web开发中，就是将模型(M-数据、操作数据)、视图(V-显示数据的HTML元素)之间实现代码分离，松散耦合，使之成为一个更容易开发、维护和测试的客户端应用程序。

View 传送指令到 Controller；

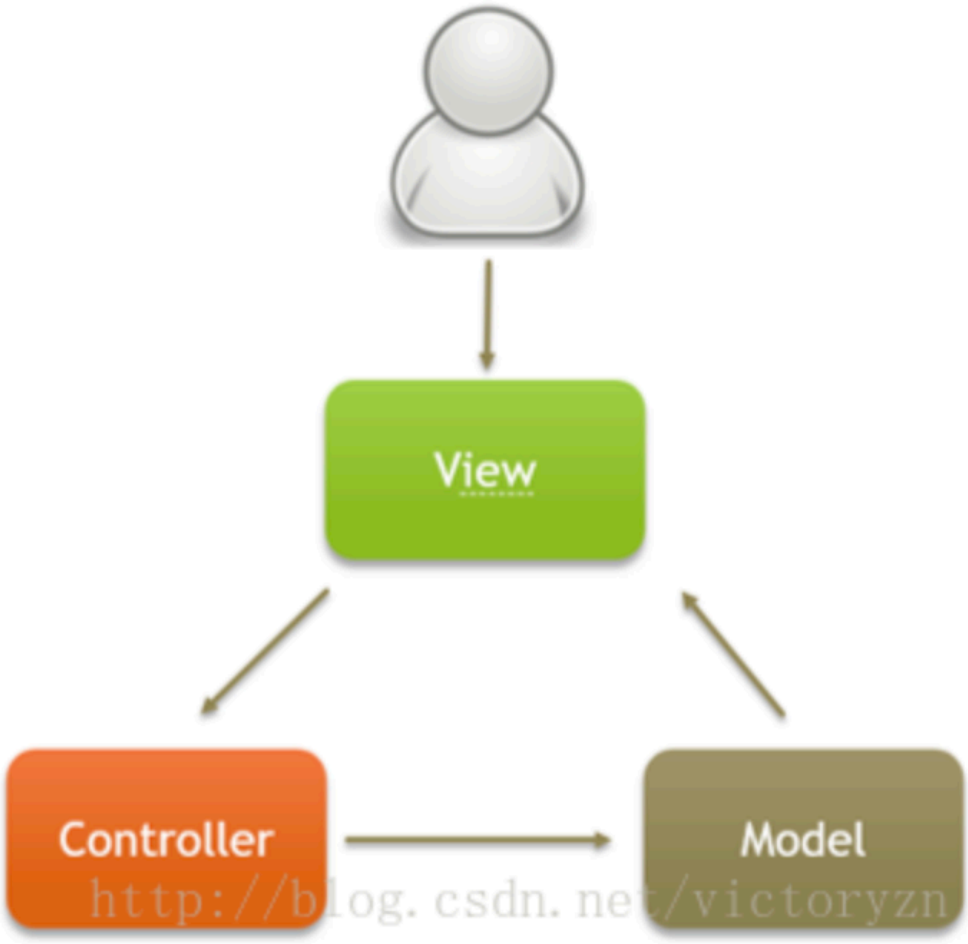
Controller 完成业务逻辑后，要求 Model 改变状态；

Model 将新的数据发送到 View，用户得到反馈。

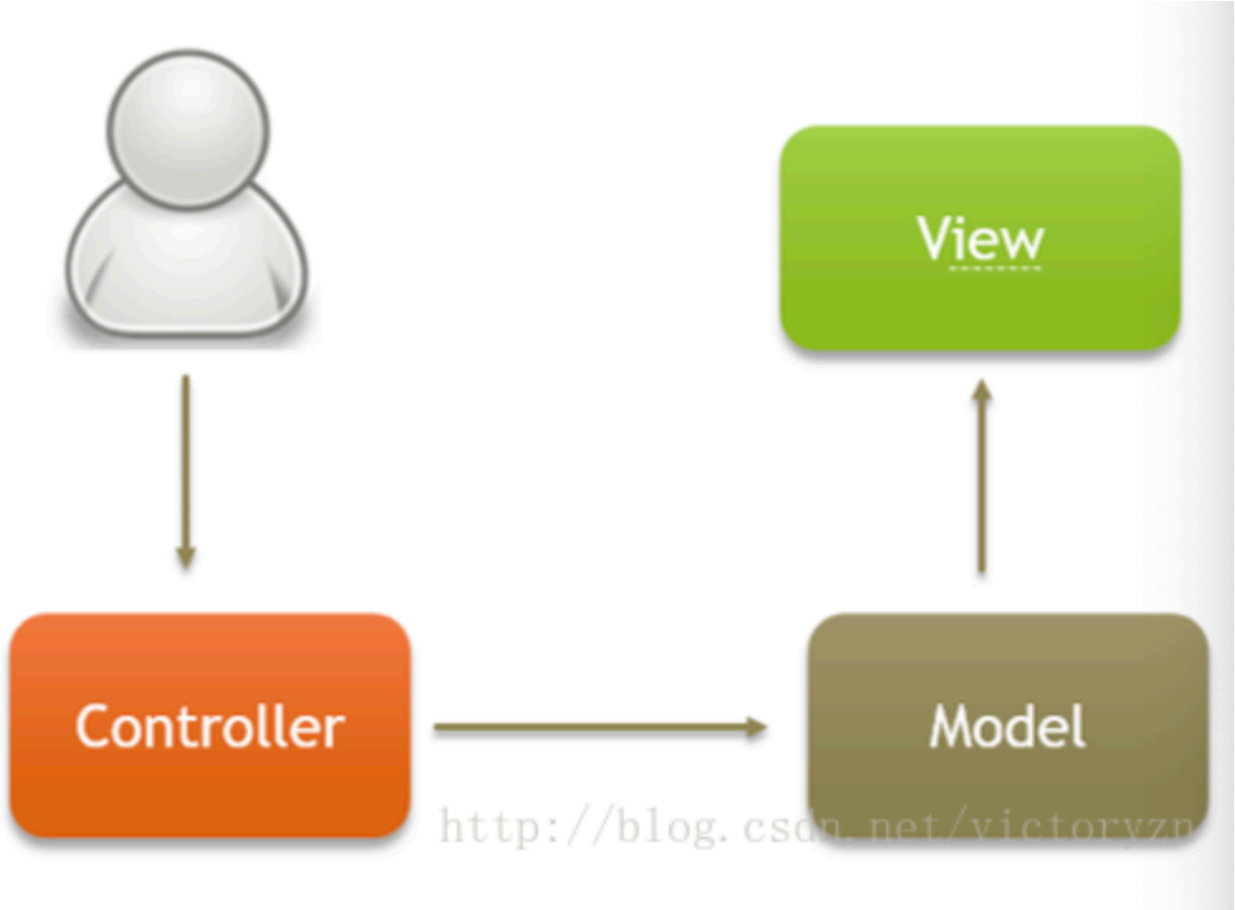
MVC流程：

MVC流程一共有两种，在日常开发中都会使用到。

一种是通过 View 接受指令，传递给 Controller，然后对模型进行修改或者查找底层数据，最后把改动渲染在视图上。



另一种是通过controller接受指令，传给Controller：



MVC优点：

耦合性低，视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码。 重用性高 生命周期成本低 MVC使开发和维护用户接口的技术含量降低 可维护性高，分离视图层和业务逻辑层也使得WEB应用更易于维护和修改 部署快 MVC缺点：

不适合小型，中等规模的应用程序，花费大量时间将MVC应用到规模并不是很大的应用程序通常会得不偿失。

视图与控制器间过于紧密连接，视图与控制器是相互分离，但却是联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

视图对模型数据的低效率访问，依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

MVC应用：

在web app 流行之初， MVC 就应用在了java（struts2）和C#（ASP.NET）服务端应用中，后来在客户端应用程序中，基于MVC模式，AngularJS应运而生。

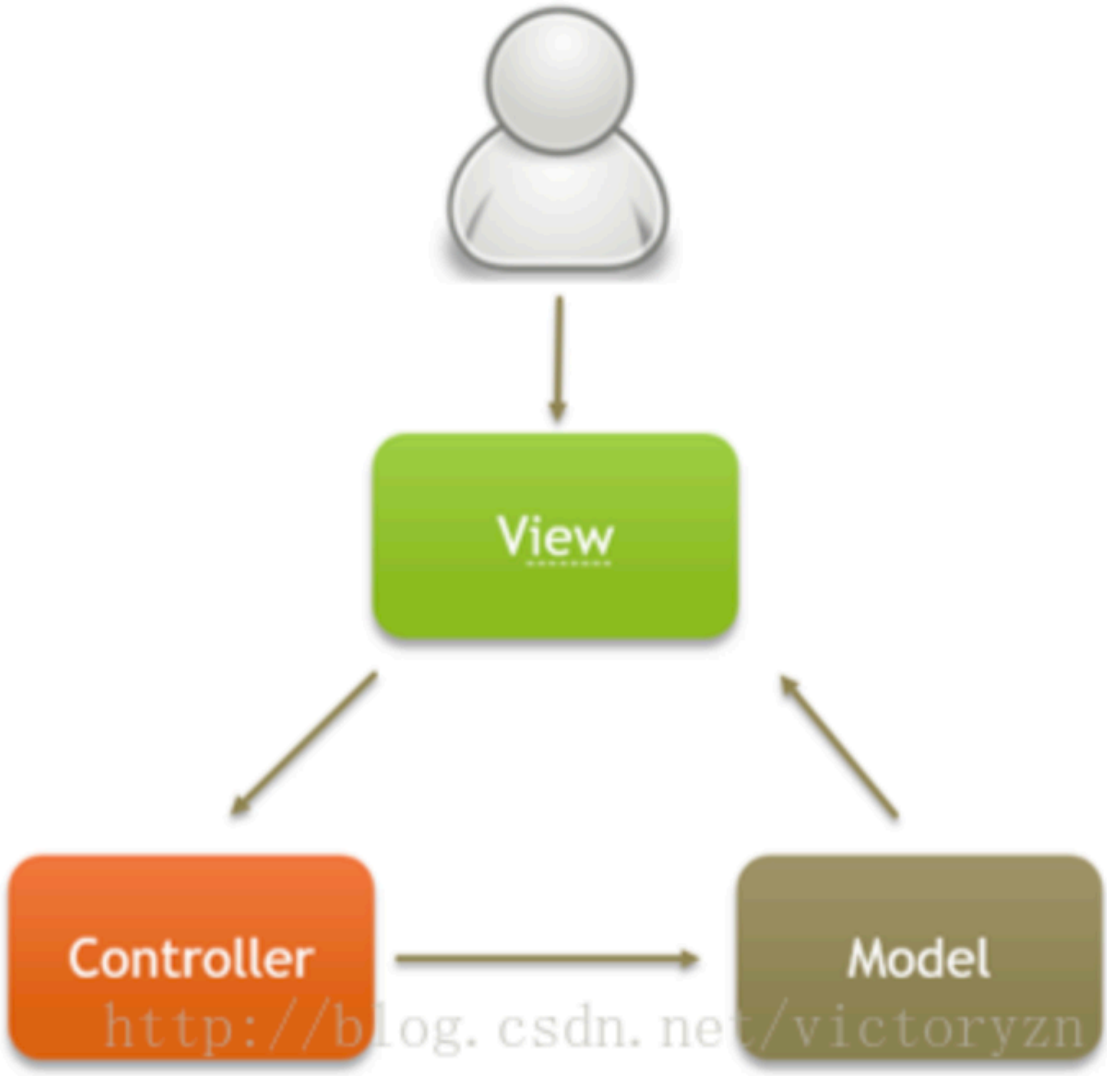
MVP MVP（Model-View-Presenter）是MVC的改良模式，由IBM的子公司Taligent提出。和MVC的相同之处在于：Controller/Presenter负责业务逻辑，Model管理数据，View负责显示只不过是将Controller 改名为 Presenter，同时改变了通信方向。

MVP特点：

M、V、P之间双向通信。 View 与 Model 不通信，都通过 Presenter 传递。Presenter完全把Model和View进行了分离，主要的程序逻辑在Presenter里实现。 View 非常薄，不部署任何业务逻辑，称为”被动视图”（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。
Presenter与具体的View是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更View时候可以保持Presenter的不变，这样就可以重用。不仅如此，还可以编写测试用的View，模拟用户的各种操作，从而实现对Presenter的测试–从而不需要使用自动化的测试工具。 与MVC区别：


View
Presenter
Model

在MVP中，View并不直接使用Model，它们之间的通信是通过Presenter (MVC中的Controller)来进行的，所有的交互都发生在Presenter内部。



在MVC中，View会直接从Model中读取数据而不是通过 Controller。 MVP优点：

模型与视图完全分离，我们可以修改视图而不影响模型； 可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter内部； 我们可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁； 如果我们把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）。 MVP缺点：

视图和Presenter的交互会过于频繁，使得他们的联系过于紧密。也就是说，一旦视图变更了，presenter也要变更。

MVP应用： 可应用与Android开发。

MVVM MVVM是Model-View-ViewModel的简写。微软的WPF(Windows Presentation Foundation–微软推出的基于Windows 的用户界面框架)带来了新的技术体验, 使得软件UI层更加细节化、可定制化。与此同时，在技术层面，WPF也带来了 诸如Binding（绑定）、Dependency Property（依赖属性）、Routed Events（路由事件）、Command（命令）、DataTemplate（数据模板）、ControlTemplate（控制模板）等新特性。MVVM模式其实是MV模式与WPF结合的应用方式时发展演变

过来的一种新型架构模式。它立足于原有MVP框架并且把WPF的新特性糅合进去，以应对客户日益复杂的需求变化。

MVVM优点：

MVVM模式和MVC模式类似，主要目的是分离视图（View）和模型（Model），有几大优点：

低耦合，视图（View）可以独立于Model变化和修改，一个ViewModel可以绑定到不同的"View"上，当View变化的时候Model可以不变，当Model变化的时候View也可以不变。

可重用性，可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑。

独立开发，开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面设计，使用Expression Blend可以很容易设计界面并生成xml代码。

可测试，界面向来是比较难于测试的，而现在测试可以针对ViewModel来写。

5.AsyncTask内部实现原理

链接：https://www.cnblogs.com/absfree/p/5357678.html

https://www.jianshu.com/p/ee1342fcf5e7

1) AsyncTask的使用简介

AsyncTask是对Handler与线程池的封装。使用它的方便之处在于能够更新用户界面，当然这里更新用户界面的操作还是在主线程中完成的，但是由于AsyncTask内部包含一个Handler，所以可以发送消息给主线程让它更新UI。另外，AsyncTask内还包含了一个线程池。使用线程池的主要原因是避免不必要的创建及销毁线程的开销。设想下面这样一个场景：有100个只需要0.001ms就能执行完毕的任务，如果创建100个线程来执行这些任务，执行完任务的线程就进行销毁。那么创建与销毁进程的开销就很可能成为了影响性能的瓶颈。通过使用线程池，我们可以实现维护固定数量的线程，不管有多少任务，我们都始终让线程池中的线程轮番上阵，这样就避免了不必要的开销。

在这里简单介绍下AsyncTask的使用方法，为后文对它的工作原理的研究做铺垫，关于AsyncTask的详细介绍大家可以参考官方文档或是相关博文。

AsyncTask是一个抽象类，我们在使用时需要定义一个它的派生类并重写相关方法。AsyncTask类的声明如下：

```
1 public abstract class AsyncTask<Params, Progress, Result>
```

我们可以看到，AsyncTask是一个泛型类，它的三个类型参数的含义如下：

- Params：doInBackground方法的参数类型；
- Progress：AsyncTask所执行的后台任务的进度类型；

- Result：后台任务的返回结果类型。

我们再来看一下AsyncTask类主要为我们提供了哪些方法：

```
onPreExecute() //此方法会在后台任务执行前被调用，用于进行一些准备工作
doInBackground(Params... params) //此方法中定义要执行的后台任务，在这个方法中可以调用publishProgress来更新任务进度（publishProgress内部会调用onProgressUpdate方法）
onProgressUpdate(Progress... values) //由publishProgress内部调用，表示任务进度更新
onPostExecute(Result result) //后台任务执行完毕后，此方法会被调用，参数即为后台任务的返回结果
onCancelled() //此方法会在后台任务被取消时被调用
```

以上方法中，除了doInBackground方法由AsyncTask内部线程池执行外，其余方法均在主线程中执行。

2) AsyncTask的局限性

AsyncTask的优点在于执行完后台任务后可以很方便的更新UI，然而使用它存在着诸多的限制。先抛开内存泄漏问题，使用AsyncTask主要存在以下局限性：

- 在Android 4.1版本之前，AsyncTask类必须在主线程中加载，这意味着对AsyncTask类的第一次访问必须发生在主线程中；在Android 4.1以及以上版本则不存在这一限制，因为ActivityThread（代表了主线程）的main方法中会自动加载AsyncTask
- AsyncTask对象必须在主线程中创建
- AsyncTask对象的execute方法必须在主线程中调用
- 一个AsyncTask对象只能调用一次execute方法

接下来，我们从源码的角度去探究一下AsyncTask的工作原理，并尝试着搞清楚为什么会存在以上局限性。

3) AsyncTask的工作原理

首先，让我们来看一下AsyncTask类的构造器都做了些什么：

```
 1 public AsyncTask() {
 2         mWorker = new WorkerRunnable<Params, Result>() {
 3             public Result call() throws Exception {
 4                 mTaskInvoked.set(true);
 5 
 6 Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
 7                 //noinspection unchecked
 8                 Result result = doInBackground(mParams);
 9                 Binder.flushPendingCommands();
10                 return postResult(result);
11             }
12         };
```

```
13 13
14 14         mFuture = new FutureTask<Result>(mWorker) {
15 15             @Override
16 16             protected void done() {
17 17                 try {
18 18                     postResultIfNotInvoked(get());
19 19                 } catch (InterruptedException e) {
20 20                     android.util.Log.w(LOG_TAG, e);
21 21                 } catch (ExecutionException e) {
22 22                     throw new RuntimeException("An error occurred while
   executing doInBackground()",
23 23                             e.getCause());
24 24                 } catch (CancellationException e) {
25 25                     postResultIfNotInvoked(null);
26 26                 }
27 27             }
28 28         };
29 29     }
```

在第2行到第12行，初始化了mWorker，它是一个派生自WorkRunnable类的对象。WorkRunnable是一个抽象类，它实现了Callable接口。我们再来看一下第4行开始的call方法的定义，首先将mTaskInvoked设为true表示当前任务已被调用过，然后在第6行设置线程的优先级。在第8行我们可以看到，调用了AsyncTask对象的doInBackground方法开始执行我们所定义的后台任务，并获取返回结果存入result中。最后将任务返回结果传递给postResult方法。关于postResult方法我们会在下文进行分析。由此我们可以知道，实际上AsyncTask的成员mWorker包含了AyncTask最终要执行的任务（即mWorker的call方法）。

接下来让我们看看对mFuture的初始化。我们可以看到mFuture是一个FutureTask的直接子类（匿名内部类）的对象，在FutureTask的构造方法中我们传入了mWorker作为参数。我们使用的是FutureTask的这个构造方法：

```
1     public FutureTask(Callable<V> callable) {
2         if (callable == null)
3             throw new NullPointerException();
4         this.callable = callable;
5         this.state = NEW;       // ensure visibility of callable
6     }
```

也就是说，mFuture是一个封装了我们的后台任务的FutureTask对象，FutureTask类实现了FutureRunnable接口，通过这个接口可以方便的取消后台任务以及获取后台任务的执行结果，具体介绍请看这里：Java并发编程：Callable、Future和FutureTask。

从上面的分析我们知道了，当mWorker中定义的call方法被执行时，doInBackground就会开始执行，我们定义的后台任务也就真正开始了。那么这个call方法什么时候会被调用呢？我们可以看到经过层层封装，实际上是mFuture对象封装了call方法，当mFuture对象被提交到AsyncTask包含的线程池执行时，call方法就会被调用，我们定义的后台任务也就开始执行了。下面我们来看一下mFuture是什么时候被提交到线程池执行的。

首先来看一下execute方法的源码：

```
1  public final AsyncTask<Params, Progress, Result> execute(Params... params)
   {
2          return executeOnExecutor(sDefaultExecutor, params);
3  }
```

我们可以看到它接收的参数是Params类型的参数，这个参数会一路传递到doInBackground方法中。execute方法仅仅是调用了executeOnExecutor方法，并将executeOnExecutor方法的返回值作为自己的返回值。我们注意到，传入了sDefaultExecutor作为executeOnExecutor方法的参数，那么sDefaultExecutor是什么呢？简单的说，它是AsyncTask的默认执行器（线程池）。AsyncTask可以以串行（一个接一个的执行）或并行（一并执行）两种方式来执行后台任务，在Android3.0及以后的版本中，默认的执行方式是串行。这个sDefaultExecutor就代表了默认的串行执行器（线程池）。也就是说我们平常在AsyncTask对象上调用execute方法，使用的是串行方式来执行后台任务。关于线程池更加详细的介绍与分析请见：深入理解Java之线程池

我们再来看一下executeOnExecutor方法都做了些什么：

```
1   1 public final AsyncTask<Params, Progress, Result>
    executeOnExecutor(Executor exec,
2   2             Params... params) {
3   3         if (mStatus != Status.PENDING) {
4   4             switch (mStatus) {
5   5                 case RUNNING:
6   6                     throw new IllegalStateException("Cannot execute
    task:"
7   7                             + " the task is already running.");
8   8                 case FINISHED:
9   9                     throw new IllegalStateException("Cannot execute
    task:"
10  10                            + " the task has already been executed "
11  11                            + "(a task can be executed only once)");
12  12            }
13  13        }
14  14
15  15        mStatus = Status.RUNNING;
16  16
17  17        onPreExecute();
18  18
19  19        mWorker.mParams = params;
20  20        exec.execute(mFuture);
21  21
22  22        return this;
23  23    }
```

从以上代码的第4行到第12行我们可以知道，当AsyncTask对象的当前状态为RUNNING或FINISHED时，调用execute方法会抛出异常，这意味着不能对正在执行任务的AsyncTask对象或是已经执行完任务的AsyncTask对象调用execute方法，这也就解释了我们上面提到的局限中的最后一条。

接着我们看到第17行存在一个对onPreExecute方法的调用，这表示了在执行后台任务前确实会调用onPreExecute方法。

在第19行，把我们传入的execute方法的params参数赋值给了mWorker的mParams成员变量；而后在第20行调用了exec的execute方法，并传入了mFuture作为参数。exec就是我们传进来的sDefaultExecutor。那么接下来我们看看sDefaultExecutor究竟是什么。在AsyncTask类的源码中，我们可以看到这句：

```
1  private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
```

sDefaultExecutor被赋值为SERIAL_EXECUTOR，那么我们来看一下SERIAL_EXECUTOR：

```
1  public static final Executor SERIAL_EXECUTOR = new SerialExecutor();
```

现在，我们知道了实际上sDefaultExecutor是一个SerialExecutor对象，我们来看一下SerialExecutor类的源码：

```
1  private static class SerialExecutor implements Executor {
2          final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
3          Runnable mActive;
4
5          public synchronized void execute(final Runnable r) {
6              mTasks.offer(new Runnable() {
7                  public void run() {
8                      try {
9                          r.run();
10                     } finally {
11                         scheduleNext();
12                     }
13                 }
14             });
15             if (mActive == null) {
16                 scheduleNext();
17             }
18         }
19
20         protected synchronized void scheduleNext() {
21             if ((mActive = mTasks.poll()) != null) {
22                 THREAD_POOL_EXECUTOR.execute(mActive);
23             }
24         }
25     }
```

我们来看一下execute方法的实现。mTasks代表了SerialExecutor这个串行线程池的任务缓存队列，在第6行，我们用offer方法向任务缓存队列中添加一个任务，任务的内容如第7行到第13行的run方法定义所示。我们可以看到，run方法中：第9行调用了mFuture（第5行的参数r就是我们传入的mFuture）的run方法，而mFuture的run方法内部会调用mWorker的call方法，然后就会调用doInBackground方法，我们的后台任务也就开始执行了。那么我们提交到任务缓存队列中的任务什么时候会被执行呢？我

们接着往下看。

首先我们看到第三行定义了一个Runnable变量mActive，它代表了当前正在执行的AsyncTask对象。第15行判断mActive是否为null，若为null，就调用scheduleNext方法。如第20行到24行所示，在scheduleNext方法中，若缓存队列非空，则调用THREAD_POOL_EXECUTOR.execute方法执行从缓存队列中取出的任务，这时我们的后台任务便开始你真正执行了。

通过以上的分析，我们可以知道SerialExecutor所完成的工作主要是把任务加到任务缓存队列中，而真正执行任务的是THREAD_POOL_EXECUTOR。我们来看下THREAD_POOL_EXECUTOR是什么：

```
1  public static final Executor THREAD_POOL_EXECUTOR
2            = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE,
KEEP_ALIVE,
3                    TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
```

从上面的代码我们可以知道，它是一个线程池对象。根据AsyncTask的源码，我们可以获取它的各项参数如下：

```
1  1 private static final int CPU_COUNT =
Runtime.getRuntime().availableProcessors();
2  2 private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
3  3 private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
4  4 private static final int KEEP_ALIVE = 1;
5  5
6  6 private static final ThreadFactory sThreadFactory = new ThreadFactory()
{
7      private final AtomicInteger mCount = new AtomicInteger(1);
8  8
9      public Thread newThread(Runnable r) {
10         return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
11     }
12 };
13 13
14 14 private static final BlockingQueue<Runnable> sPoolWorkQueue =
15 15             new LinkedBlockingQueue<Runnable>(128);
```

由以上代码我们可以知道：

- CORE_POOL_SIZE为CPU数加一；
- MAXIMUM_POOL_SIZE为CPU数的二倍加一；
- 存活时间为1秒；
- 任务缓存队列为LinkedBlockingQueue。

现在，我们已经了解到了从我们调用AsyncTask对象的execute方法开始知道后台任务执行完都发生了什么。现在让我们回过头来看一看之前提到的postResult方法的源码：

```
private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
            new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

在以上源码中，先调用了getHandler方法获取AsyncTask对象内部包含的sHandler，然后通过它发送了一个MESSAGE_POST_RESULT消息。我们来看看sHandler的相关代码：

```
1  private static final InternalHandler sHandler = new InternalHandler();
2
3  private static class InternalHandler extends Handler {
4          public InternalHandler() {
5              super(Looper.getMainLooper());
6          }
7
8          @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
9          @Override
10         public void handleMessage(Message msg) {
11             AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
12             switch (msg.what) {
13                 case MESSAGE_POST_RESULT:
14                     // There is only one result
15                     result.mTask.finish(result.mData[0]);
16                     break;
17                 case MESSAGE_POST_PROGRESS:
18                     result.mTask.onProgressUpdate(result.mData);
19                     break;
20             }
21         }
22 }
```

从以上代码中我们可以看到，sHandler是一个静态的Handler对象。我们知道创建Handler对象时需要当前线程的Looper，所以我们为了以后能够通过sHandler将执行环境从后台线程切换到主线程（即在主线程中执行handleMessage方法），我们必须使用主线程的Looper，因此必须在主线程中创建sHandler。这也就解释了为什么必须在主线程中加载AsyncTask类，是为了完成sHandler这个静态成员的初始化工作。

在以上代码第10行开始的handleMessage方法中，我们可以看到，当sHandler收到MESSAGE_POST_RESULT方法后，会调用finish方法，finish方法的源码如下：

```
1 private void finish(Result result) {
2         if (isCancelled()) {
3             onCancelled(result);
4         } else {
5             onPostExecute(result);
6         }
7         mStatus = Status.FINISHED;
8 }
```

在第2行，会通过调用isCancelled方法判断AsyncTask任务是否被取消，若取消了则调用onCancelled方法，否则调用onPostExecute方法；在第7行，把mStatus设为FINISHED，表示当前AsyncTask对象已经执行完毕。

经过了以上的分析，我们大概了解了AsyncTask的内部运行逻辑，知道了它默认使用串行方式执行任务。那么如何让它以并行的方式执行任务呢？ 阅读了以上的代码后，我们不难得到结论，只需调用executeOnExecutor方法，并传入THREAD_POOL_EXECUTOR作为其线程池即可。

优点：

1.方便异步通信，不需使用 “任务线程（如继承`Thread`类） + `Handler`”的复杂组合。

2.节省资源，采用线程池的缓存线程 + 复用线程，避免了频繁创建 & 销毁线程所带来的系统资源开销。

4) AsyncTask各个方法的作用

6.service两种启动方式有什么区别?

1）start启动方式：

步骤：

1.定义一个类继承`Service`。

2.在`Manifest.xml`文件中配置该`Service`。 3.使用Context的`startService(Intent)`方法启动该Service。 4.不再使用时，调用`stopService(Intent)`方法停止该服务。

生命周期：

```
onCreate()--->onStartCommand()（onStart()方法已过时） ---> onDestory()
```

说明：如果服务已经开启，不会重复的执行`onCreate()`， 而是会调用`onStart()`和`onStartCommand()`。 服务停止的时候调用 `onDestory()`。服务只会被停止一次。

特点：一旦服务开启跟调用者(开启者)就没有任何关系了。开启者退出了，开启者挂了，服务还在后台长期的运行。开启者不能调用服务里面的方法。

2）bind启动方式：

步骤：

1.定义一个类继承 `Service` 。 2.在 `Manifest.xml` 文件中配置该 `Service`。 3.使用Context的 `bindService(Intent, ServiceConnection, int)` 方法启动该Service。 4.不再使用时，调用 `unbindService(ServiceConnection)` 方法停止该服务。

生命周期：

```
onCreate()` --->`onBind()`--->`onunbind()`--->`onDestory()
```

注意：绑定服务不会调用 `onstart()` 或者 `onstartcommand()` 方法

特点：bind的方式开启服务，绑定服务，调用者挂了，服务也会跟着挂掉。 绑定者可以调用服务里面的方法。

绑定者如何调用服务里的方法呢?

1.先定义一个service子类，并在功能清单中注册。

```
class MyService : Service() {

    override fun onBind(intent: Intent): IBinder? {
        return MyBinder()
    }

    override fun onCreate() {
        super.onCreate()
        Log.e("TAG", "onCreate()")
    }

    override fun onStartCommand(intent: Intent?, flags: Int, startId:
Int): Int {
        Log.e("TAG", "onStartCommand()")
        return super.onStartCommand(intent, flags, startId)
    }

    override fun stopService(name: Intent?): Boolean {
        Log.e("TAG", "stopService()")
        return super.stopService(name)
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.e("TAG", "onDestroy()")
    }
}


```

```
    /**
     * 该类用于在onBind方法执行后返回的对象,
     * 该对象对外提供了该服务里的方法
     */
class MyBinder : Binder() {

    fun testBindService() {
        Log.e("TAG", "测试绑定服务")
    }
}
```

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initAction()
    }

    lateinit var mIntent: Intent
    private val serviceConnection: ServiceConnection = object :
ServiceConnection {
        override fun onServiceDisconnected(name: ComponentName?) {
            Log.e("TAG", "onServiceDisconnected()")
        }

        override fun onServiceConnected(name: ComponentName?, service:
IBinder?) {
            val myBinder = service as MyBinder
            myBinder.testBindService()
            Log.e("TAG", "onServiceConnected()")
        }

    }

    private fun initAction() {
        btn_start.setOnClickListener {
            mIntent = Intent()
            mIntent.setClass(this@MainActivity, MyService::class.java)
            startService(mIntent)
        }
        btn_stop.setOnClickListener {
            stopService(mIntent)
        }

        btn_bind.setOnClickListener {
            mIntent = Intent()
            mIntent.setClass(this@MainActivity, MyService::class.java)
```

```
            bindService(mIntent, serviceConnection,
Context.BIND_AUTO_CREATE)
        }

        btn_unbind.setOnClickListener {
           unbindService(serviceConnection)
        }

    }
}
```

绑定本地服务调用方法的步骤：

1. 在服务的内部创建一个内部类 提供一个方法，可以间接调用服务的方法
2. 实现服务的onbind方法，返回的就是这个内部类
3. 在activity 绑定服务。bindService();
4. 在服务成功绑定的回调方法onServiceConnected， 会传递过来一个 IBinder对象
5. 强制类型转化为自定义的接口类型，调用接口里面的方法。

区别:

startService启动Service ,Service有独立的生命周期，不依赖该组件； 多次调用startService方法，会重复调用onStartCommand方法； 必须通过stopService或者stopSelf来停止服务（IntentService会自动调用stopSelf方法）

bindService启动Service，多次调用此方法，只会调用一次onBind方法； bindService,Service 依赖于此组件，该组件销毁后，Service也会随之销毁。

扩展： 1、同一个Service，先启动startService，然后在bindService，如何把服务停掉?

无论被startService调用多少次，如需要stopService或者stopSelf方法 一次； 调用n次bindService，必须调用一次unBindService方法；

因此，需要调用一次stopService(或者stopSelf)方法，和一次unBindService方法，执行顺序没有要求，最后一个stopService或者unBindService方法会导致Service的 onDestory执行。

2、Service的生命方法是运行在那个线程中?

Service默认运行在主线程，所以其生命方法也是运行在主线程中，如果需要在Service中进行耗时操作，必须另起线程（或者使用IntentService）否则会引起ANR。

7.说说图片三级缓存

1）为什么要三级缓存?

- 为用户节省流量，对相同资源减少多次重复的网络请求；
- 部分业务需要。例如有些业务需要在用户断网时也可以进行一些浏览或操作；
- 各缓存读取速度不相同，结合使用提高效率；

2）什么事三级缓存?

所谓三级缓存，指的是：内存缓存，本地缓存（或者叫文件缓存），网络缓存（我个人认为把网络算在缓存里其实是不太合适的）。

1. 内存缓存：只有当APP运行时才会涉及到。内存虽然有容量限制，但是从内存读取信息是速度最快的。
2. 本地缓存：信息以文件的形式存储在本地。只要不清除这些文件，那么信息就一直持久化的保存着。需要时可以通过流的方式进行读取。本地容量大，速度次于内存。
3. 网络：信息存储在远端Server。通过网络获取信息。完全依赖网络情况，速度相对上面两者来说要慢。

3）图片异步加载缓存方案的工作流程

5）设计三级缓存

（1）定义接口

```
public interface ImageCache {
    Bitmap getBitmap(String url);

    void putBitmap(String url, Bitmap bitmap
}
```

（2）实现内存缓存

```
public class MemoryCache implements ImageCache {
    private LruCache<String, Bitmap> mLruCache;
    private static final int MAX_LRU_CACHE_SIZE = (int)
(Runtime.getRuntime().maxMemory() / 8);

    public MemoryCache() {
        //初始化LruCache
        initLruCache();
    }

    private void initLruCache() {
          mLruCache = new LruCache<String, Bitmap>(MAX_LRU_CACHE_SIZE) {
             @Override
             protected int sizeOf(String key, Bitmap bitmap) {
                  return bitmap.getRowBytes() * bitmap.getHeight();
             }
          };
   }

    @Override
    public Bitmap getBitmap(String url) {
         return mLruCache.get(url);
    }
```

```
23
24      @Override
25      public void putBitmap(String url, Bitmap bitmap) {
26          mLruCache.put(url, bitmap);
27      }
28  }
```

（3）实现本地缓存

DiskLruCache是Google自己写的一个类，用来做本地缓存方案十分方便。

```
1   public class DiskCache implements ImageCache {
2       private DiskLruCache mDiskLruCache;
3       private static final String DISK_LRU_CACHE_UNIQUE = "Image";
4       private static final int MAX_DISK_LRU_CACHE_SIZE = 10 * 1024 * 1024;
5
6       ExecutorService mExecutorsService =
    Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
7
8       public DiskCache(Context context) {
9           //初始化DiskLruCache
10          initDiskLruCache(context);
11      }
12
13      private void initDiskLruCache(Context context) {
14          try {
15              File cacheDir = getDiskCacheDir(context,
    DISK_LRU_CACHE_UNIQUE);
16              if ( ! cacheDir.exists() ) {
17                  cacheDir.mkdirs();
18              }
19              mDiskLruCache = DiskLruCache.open(cacheDir,
    getAppVersion(context), 1, MAX_DISK_LRU_CACHE_SIZE);
20          } catch ( IOException e ) {
21              e.printStackTrace();
22          }
23      }
24
25      private File getDiskCacheDir(Context context, String uniqueName) {
26          String cachePath;
27          if (
    Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
    || ! Environment.isExternalStorageRemovable() ) {
28              cachePath = context.getExternalCacheDir().getPath();
29          } else {
30              cachePath = context.getCacheDir().getPath();

31          }
32          return new File(cachePath + File.separator + uniqueName);
```

```
33      }
34
35      private int getAppVersion(Context context) {
36          try {
37              PackageInfo info = context.getPackageManager()
38
.getPackageInfo(context.getPackageName(), 0);
39              return info.versionCode;
40          } catch ( PackageManager.NameNotFoundException e ) {
41              e.printStackTrace();
42          }
43          return 1;
44      }
45
46      @Override
47      public Bitmap getBitmap(String url) {
48          String bitmapUrlMD5 = Md5Util.getMD5String(url);
49          Bitmap bitmap = null;
50          DiskLruCache.Snapshot snapshot = null;
51          try {
52              snapshot = mDiskLruCache.get(bitmapUrlMD5);
53          } catch ( IOException e ) {
54              e.printStackTrace();
55          }
56          if ( snapshot != null ) {
57              InputStream inputStream = snapshot.getInputStream(0);
58              bitmap = BitmapFactory.decodeStream(inputStream);
59          }
60          return bitmap;
61      }
62
63      @Override
64      public void putBitmap(String url, final Bitmap bitmap) {
65          final String bitmapUrlMD5 = Md5Util.getMD5String(url);
66          mExecutorsService.submit(new Runnable() {
67              @Override
68              public void run() {
69                  writeFileToDisk(mDiskLruCache, bitmap, bitmapUrlMD5);
70              }
71          });
72      }
73
74      private static void writeFileToDisk(DiskLruCache diskLruCache, Bitmap
bitmap, String bitmapUrlMD5) {
75          DiskLruCache.Editor editor = null;
76          OutputStream outputStream = null;
77          try {
78              editor = diskLruCache.edit(bitmapUrlMD5);
79              if ( editor != null ) {
```

```
                outputStream = editor.newOutputStream(0);
                if ( bitmap.compress(Bitmap.CompressFormat.PNG, 100,
outputStream) ) {
                    editor.commit();
                }
            }
        } catch ( Exception e ) {
            try {
                if ( editor != null ) {
                    editor.abort();
                }
            } catch ( Exception e1 ) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        } finally {
            try {
                diskLruCache.flush();
            } catch ( Exception e ) {

            }
        }
    }
```

可以看到本地缓存的时候对url做了一次MD5加密。这是为了从安全考虑。毕竟直接把url暴露在文件上实在不太雅观。

（4）完成内存缓存加本地缓存的双缓存逻辑实现

对于图片的获取：先从内存缓存获取图片。如果不为空直接返回。如果为空，再从本地缓存获取图片。

对于图片的保存：就是往内存缓存和本地缓存分别添加图片。

```
public class MemoryAndDiskCache implements ImageCache {
    private MemoryCache mMemoryCache;
    private DiskCache mDiskCache;

    public MemoryAndDiskCache(Context context) {
        mMemoryCache = new MemoryCache();
        mDiskCache = new DiskCache(context);
    }

    @Override
    public Bitmap getBitmap(String url) {
        Bitmap bitmap = mMemoryCache.getBitmap(url);
        if (bitmap != null) {
            return bitmap;
        } else {
            bitmap = mDiskCache.getBitmap(url);
            return bitmap;
```

```
        }
    }

    @Override
    public void putBitmap(String url, Bitmap bitmap) {
        mMemoryCache.putBitmap(url, bitmap);
        mDiskCache.putBitmap(url, bitmap);
    }
}
```

（5）实现ImageLoader类

这个类中我们会在构造函数中传入ImageCache的实例。那么在获取和保存图片时，只需要调用接口中定义的两个方法即可，无需关注细节。实现细节完全交由构造函数中传入的ImageCache实例。当要获取图片时，先调用ImageCache接口实例的getBitmap方法，如果为空。那么需要我们从网络下载图片。下载完成后我们只要调用ImageCache接口示例的putBitmap方法，即可完成整个图片缓存方案。

```
public class ImageLoader {
    private ImageCache mImageCache;

    public ImageLoader(ImageCache imageCache) {
        mImageCache = imageCache;
    }

    public void displayImage(String url, ImageView imageView, int
defaultImageRes) {
        imageView.setImageResource(defaultImageRes);
        imageView.setTag(url);

        Bitmap bitmap = mImageCache.getBitmap(url);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
        } else {
            downloadImage(imageView, url);
        }
    }

    private void downloadImage(final ImageView imageView, final String
url) {
        Call<ResponseBody> resultCall =
ServiceFactory.getServices().downloadImage(url);
        resultCall.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call,
Response<ResponseBody> response) {
                if (response != null && response.body() != null) {
                    InputStream inputStream =
response.body().byteStream();
```

```
                    Bitmap bitmap =
BitmapFactory.decodeStream(inputStream);
                    if ( TextUtils.equals((String) imageView.getTag(),
url)) {
                        imageView.setImageBitmap(bitmap);
                    }
                    mImageCache.putBitmap(url, bitmap);
                }
            }

            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
            }
        });
    }
}
```

6）LruCache 源码解析

1. 简介

> LRU 是 Least Recently Used 最近最少使用算法。

> 曾经，在各大缓存图片的框架没流行的时候。有一种很常用的内存缓存技术：SoftReference 和 WeakReference（软引用和弱引用）。但是走到了 Android 2.3（Level 9）时代，垃圾回收机制更倾向于回收 SoftReference 或 WeakReference 的对象。后来，又来到了 Android3.0，图片缓存在内容中，因为不知道要在是什么时候释放内存，没有策略，没用一种可以预见的场合去将其释放。这就造成了内存溢出。

2. 使用方法

当成一个 Map 用就可以了，只不过实现了 LRU 缓存策略。

使用的时候记住几点即可：

- **1.（必填）**你需要提供一个缓存容量作为构造参数。
- **2.（必填）** 覆写 `sizeOf` 方法，自定义设计一条数据放进来的容量计算，如果不覆写就无法预知数据的容量，不能保证缓存容量限定在最大容量以内。
- **3.（选填）** 覆写 `entryRemoved` 方法，你可以知道最少使用的缓存被清除时的数据（evicted, key, oldValue, newVaule）。
- **4.（记住）**LruCache是线程安全的，在内部的 get、put、remove 包括 trimToSize 都是安全的（因为都上锁了）。
- **5.（选填）** 还有就是覆写 `create` 方法。

一般做到 **1、2、3、4就足够了，5可以无视** 。

以下是 一个 **LruCache 实现 Bitmap 小缓存的案例**, `entryRemoved` 里的自定义逻辑可以无视，想看的可以去到我的我的展示 demo 里的看自定义 `entryRemoved` 逻辑。

```
private static final float ONE_MIB = 1024 * 1024;
```

```
// 7MB
private static final int CACHE_SIZE = (int) (7 * ONE_MIB);
private LruCache<String, Bitmap> bitmapCache;
this.bitmapCache = new LruCache<String, Bitmap>(CACHE_SIZE) {
    protected int sizeOf(String key, Bitmap value) {
        return value.getByteCount();
    }

    @Override
    protected void entryRemoved(boolean evicted, String key, Bitmap
oldValue, Bitmap newValue) {
        ...
    }
};
```

3. 效果展示

LruCache 效果展示

4. 源码分析

4.1 LruCache 原理概要解析

LruCache 就是 **利用 LinkedHashMap 的一个特性（ accessOrder＝true 基于访问顺序 ）再加上对 LinkedHashMap 的数据操作上锁实现的缓存策略。**

LruCache 的数据缓存是内存中的。

- 1.首先设置了内部 `LinkedHashMap` 构造参数 `accessOrder=true`， 实现了数据排序按照访问顺序。
- 2.然后在每次 `LruCache.get(K key)` 方法里都会调用 `LinkedHashMap.get(Object key)`。
- 3.如上述设置了 `accessOrder=true` 后，每次 `LinkedHashMap.get(Object key)` 都会进行 `LinkedHashMap.makeTail(LinkedEntry<K, V> e)`。
- 4.`LinkedHashMap` 是双向循环链表，然后每次 `LruCache.get` -> `LinkedHashMap.get` 的数据就被放到最末尾了。
- 5.在 `put` 和 `trimToSize` 的方法执行下，如果发生数据量移除，会优先移除掉最前面的数据（因为最新访问的数据在尾部）。

具体解析在： *4.2*、*4.3*、*4.4*、*4.5* 。

4.2 LruCache 的唯一构造方法

```
/**
 * LruCache的构造方法：需要传入最大缓存个数
 */
public LruCache(int maxSize) {

    ...

    this.maxSize = maxSize;
```

```
    /*
     * 初始化LinkedHashMap
     * 第一个参数：initialCapacity，初始大小
     * 第二个参数：loadFactor，负载因子=0.75f
     * 第三个参数：accessOrder=true，基于访问顺序；accessOrder=false，基于插入顺序
     */
    this.map = new LinkedHashMap<K, V>(0, 0.75f, true);
}
```

第一个参数 `initialCapacity` 用于初始化该 LinkedHashMap 的大小。

先简单介绍一下 第二个参数 `loadFactor`，这个其实的 HashMap 里的构造参数，涉及到**扩容问题**，比如 HashMap 的最大容量是100，那么这里设置0.75f的话，到75容量的时候就会扩容。

主要是第三个参数 `accessOrder=true`，**这样的话 LinkedHashMap 数据排序就会基于数据的访问顺序，从而实现了 LruCache 核心工作原理。**

4.3 LruCache.get(K key)

```
/**
 * 根据 key 查询缓存，如果存在于缓存或者被 create 方法创建了。
 * 如果值返回了，那么它将被移动到双向循环链表的的尾部。
 * 如果如果没有缓存的值，则返回 null。
 */
public final V get(K key) {

    ...

    V mapValue;
    synchronized (this) {
        // 关键点：LinkedHashMap每次get都会基于访问顺序来重整数据顺序
        mapValue = map.get(key);
        // 计算 命中次数
        if (mapValue != null) {
            hitCount++;
            return mapValue;
        }
        // 计算 丢失次数
        missCount++;
    }

    /*
     * 官方解释：
     * 尝试创建一个值，这可能需要很长时间，并且Map可能在create()返回的值时有所不同。如果在create()执行的时候，一个冲突的值被添加到Map，我们在Map中删除这个值，释放被创造的值。
     *
     */
    V createdValue = create(key);
```

```
29      if (createdValue == null) {
30          return null;
31      }
32
33      /**************************
34       * 不覆写create方法走不到下面 *
35       **************************/
36
37      /*
38       * 正常情况走不到这里
39       * 走到这里的话 说明 实现了自定义的 create(K key) 逻辑
40       * 因为默认的 create(K key) 逻辑为null
41       */
42      synchronized (this) {
43          // 记录 create 的次数
44          createCount++;
45          // 将自定义create创建的值，放入LinkedHashMap中，如果key已经存在，会返回 之
    前相同key 的值
46          mapValue = map.put(key, createdValue);
47
48          // 如果之前存在相同key的value，即有冲突。
49          if (mapValue != null) {
50              /*
51               * 有冲突
52               * 所以 撤销 刚才的 操作
53               * 将 之前相同key 的值 重新放回去
54               */
55              map.put(key, mapValue);
56          } else {
57              // 拿到键值对，计算出在容量中的相对长度，然后加上
58              size += safeSizeOf(key, createdValue);
59          }
60      }
61
62      // 如果上面 判断出了 将要放入的值发生冲突
63      if (mapValue != null) {
64          /*
65           * 刚才create的值被删除了，原来的之前相同key 的值被重新添加回去了
66           * 告诉自定义的entryRemoved 方法
67           */
68          entryRemoved(false, key, createdValue, mapValue);
69          return mapValue;
70      } else {
71          // 上面 进行了 size += 操作 所以这里要重整长度
72          trimToSize(maxSize);
73          return createdValue;
74      }
75  }
```

上述的 `get` 方法表面并没有看出哪里有实现了 LRU 的缓存策略。主要在 `mapValue = map.get(key)`;里，**调用了 LinkedHashMap 的 get 方法，再加上 LruCache 构造里默认设置 LinkedHashMap 的 accessOrder=true**。

4.4 LinkedHashMap.get(Object key)

```
/**
 * Returns the value of the mapping with the specified key.
 *
 * @param key
 *            the key.
 * @return the value of the mapping with the specified key, or {@code null}
 *         if no mapping for the specified key is found.
 */
@Override public V get(Object key) {
    /*
     * This method is overridden to eliminate the need for a polymorphic
     * invocation in superclass at the expense of code duplication.
     */
    //jdk1.7
    if (key == null) {
        HashMapEntry<K, V> e = entryForNullKey;
        if (e == null)
            return null;
        if (accessOrder)
            makeTail((LinkedEntry<K, V>) e);
        return e.value;
    }

    int hash = Collections.secondaryHash(key);
    HashMapEntry<K, V>[] tab = table;
    for (HashMapEntry<K, V> e = tab[hash & (tab.length - 1)];
         e != null; e = e.next) {
        K eKey = e.key;
        if (eKey == key || (e.hash == hash && key.equals(eKey))) {
            if (accessOrder)
                makeTail((LinkedEntry<K, V>) e);
            return e.value;
        }
    }
    return null;
}

//jdk1.8
 Node<K,V> e;
        if ((e = getNode(hash(key), key)) == null)
            return null;
        if (accessOrder)
```

```
            afterNodeAccess(e);
        return e.value;
```

其实仔细看 `if (accessOrder)` 的逻辑即可，如果 `accessOrder=true` 那么每次 `get` 都会执行 N 次 `makeTail(LinkedEntry<K, V> e)` 。

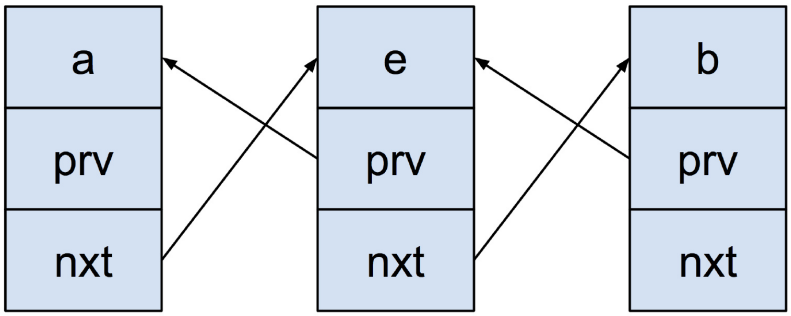
接下来看看：

4.5 LinkedHashMap.makeTail(LinkedEntry<K, V> e)

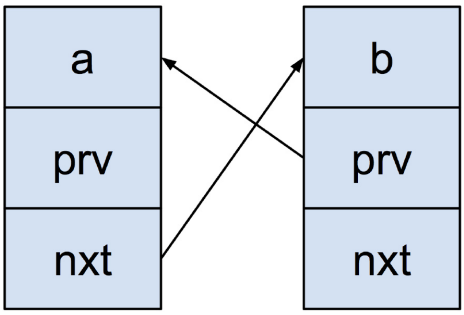
```
/**
 * Relinks the given entry to the tail of the list. Under access ordering,
 * this method is invoked whenever the value of a  pre-existing entry is
 * read by Map.get or modified by Map.put.
 */
//jdk1.7
private void makeTail(LinkedEntry<K, V> e) {
    // Unlink e
    e.prv.nxt = e.nxt;
    e.nxt.prv = e.prv;

    // Relink e as tail
    LinkedEntry<K, V> header = this.header;
    LinkedEntry<K, V> oldTail = header.prv;
    e.nxt = header;
    e.prv = oldTail;
    oldTail.nxt = header.prv = e;
    modCount++;
}
//jdk1.8
    void afterNodeAccess(Node<K,V> e) { // move node to last
        LinkedHashMapEntry<K,V> last;
        if (accessOrder && (last = tail) != e) {
            LinkedHashMapEntry<K,V> p =
                (LinkedHashMapEntry<K,V>)e, b = p.before, a = p.after;
            p.after = null;
            if (b == null)
                head = a;
            else
                b.after = a;
            if (a != null)
                a.before = b;
            else
                last = b;
            if (last == null)
                head = p;
            else {
                p.before = last;
                last.after = p;
```

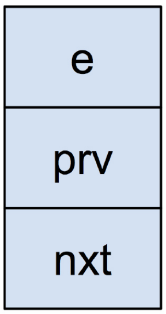
```
40                  }
41                  tail = p;
42                  ++modCount;
43              }
44      }
```

// Unlink e

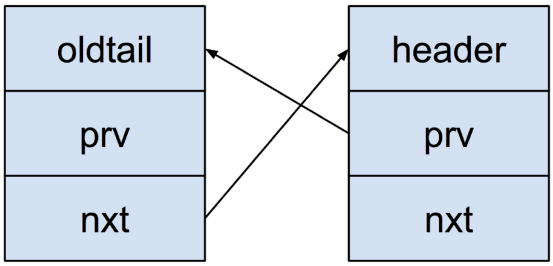


e.prv.nxt=e.nxt
e.nxt.prv=e.prv

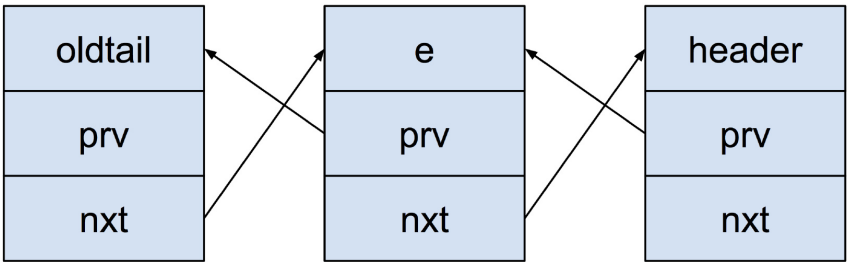




// Relink e as tail




```
LinkedEntry<K, V> header = this.header;
LinkedEntry<K, V> oldTail = header.prv;
e.nxt = header;
e.prv = oldTail;
oldTail.nxt = header.prv = e;
```



LinkedHashMap 是双向循环链表，然后此次 **LruCache.get -> LinkedHashMap.get** 的数据就被放到最末尾了。

以上就是 LruCache 核心工作原理。

接下来介绍 **LruCache 的容量溢出策略。**

4.6 LruCache.put(K key, V value)

```
public final V put(K key, V value) {
    ...
    synchronized (this) {
        ...
        // 拿到键值对，计算出在容量中的相对长度，然后加上
        size += safeSizeOf(key, value);
        ...
    }
    ...
    trimToSize(maxSize);
    return previous;
}
```

记住几点：

- **1.**put 开始的时候确实是把值放入 LinkedHashMap 了，**不管超不超过你设定的缓存容量。**
- **2.**然后根据 `safeSizeOf` 方法计算 此次添加数据的容量是多少，并且加到 `size` 里 。
- **3.**说到 `safeSizeOf` 就要讲到 `sizeOf(K key, V value)` 会计算出此次添加数据的大小 。
- **4.**直到 put 要结束时，进行了 `trimToSize` 才判断 `size` 是否 大于 `maxSize` 然后进行最近很少访问数据的移除。

4.7 LruCache.trimToSize(int maxSize)

```
public void trimToSize(int maxSize) {
    /*
     * 这是一个死循环,
     * 1.只有 扩容 的情况下能立即跳出
     * 2.非扩容的情况下，map的数据会一个一个删除，直到map里没有值了，就会跳出
     */
    while (true) {
        K key;
        V value;
        synchronized (this) {
            // 在重新调整容量大小前，本身容量就为空的话，会出异常的。
            if (size < 0 || (map.isEmpty() && size != 0)) {
                throw new IllegalStateException(
                        getClass().getName() + ".sizeOf() is reporting
inconsistent results!");
            }
            // 如果是 扩容 或者 map为空了，就会中断，因为扩容不会涉及到丢弃数据的情况
            if (size <= maxSize || map.isEmpty()) {
                break;
            }

            Map.Entry<K, V> toEvict = map.entrySet().iterator().next();
            key = toEvict.getKey();
            value = toEvict.getValue();
            map.remove(key);
            // 拿到键值对，计算出在容量中的相对长度，然后减去。
            size -= safeSizeOf(key, value);
            // 添加一次收回次数
            evictionCount++;
        }
        /*
         * 将最后一次删除的最少访问数据回调出去
         */
        entryRemoved(true, key, value, null);
    }
}
```

简单描述：会判断之前 `size` 是否大于 `maxSize` 。是的话，直接跳出后什么也不做。不是的话，证明已经溢出容量了。由 `makeTail` 图已知，最近经常访问的数据在最末尾。拿到一个存放 key 的 Set，然后一直一直从头开始删除，删一个判断是否溢出，直到没有溢出。

4.8 覆写 entryRemoved 的作用

entryRemoved被LruCache调用的场景：

- **1.（put）** put 发生 key 冲突时被调用，**evicted=false，key=此次 put 的 key，oldValue=被覆盖的冲突 value，newValue=此次 put 的 value**。
- **2.（trimToSize）** trimToSize 的时候，只会被调用一次，就是最后一次被删除的最少访问数据带回来。**evicted=true，key=最后一次被删除的 key，oldValue=最后一次被删除的 value，newValue=null（此次没有冲突，只是 remove）**。
- **3.（remove）** remove的时候，存在对应 key，并且被成功删除后被调用。**evicted=false，key=此次 put的 key，oldValue=此次删除的 value，newValue=null（此次没有冲突，只是 remove）**。
- **4.（get后半段，查询丢失后处理情景，不过建议忽略）** get 的时候，正常的话不实现自定义 `create` 的话，代码上看 get 方法只会走一半，如果你实现了自定义的 `create(K key)` 方法，并且在 你 create 后的值放入 LruCache 中发生 key 冲突时被调用，**evicted=false，key=此次 get 的 key，oldValue=被你自定义 create(key)后的 value，newValue=原本存在 map 里的 key-value**。

解释一下第四点吧：**<1>.**第四点是这样的，先 get(key)，然后没拿到，丢失。**<2>.**如果你提供了 自定义的 `create(key)` 方法，那么 LruCache 会根据你的逻辑自造一个 value，但是当放入的时候发现冲突了，但是已经放入了。**<3>.**此时，会将那个冲突的值再让回去覆盖，此时调用上述4.的 entryRemoved。

因为 HashMap 在数据量大情况下，拿数据可能造成丢失，导致前半段查不到，你自定义的 `create(key)` 放入的时候发现又查到了**（有冲突）**。然后又急忙把原来的值放回去，此时你就白白create一趟，无所作为，还要走一遍entryRemoved。

综上就如同注释写的一样：

```
/**
 * 1.当被回收或者删掉时调用。该方法当value被回收释放存储空间时被remove调用
 * 或者替换条目值时put调用，默认实现什么都没做。
 * 2.该方法没用同步调用，如果其他线程访问缓存时，该方法也会执行。
 * 3.evicted=true：如果该条目被删除空间 （表示 进行了trimToSize or remove）
evicted=false：put冲突后 或 get里成功create后导致
 * 4.newValue!=null，那么则被put()或get()调用。
 */
protected void entryRemoved(boolean evicted, K key, V oldValue, V newValue)
{
}
```

可以参考我的 demo 里的 `entryRemoved` 。

4.9 LruCache 局部同步锁

在 `get`，`put`，`trimToSize`，`remove` 四个方法里的 `entryRemoved` 方法都不在同步块里。因为 `entryRemoved` 回调的参数都属于方法域参数，不会线程不安全。

> 本地方法栈和程序计数器是线程隔离的数据区

5. 开源项目中的使用

square/picasso

6. 总结

LruCache重要的几点：

- **1.**LruCache 是通过 LinkedHashMap 构造方法的第三个参数的 `accessOrder=true` 实现了 `LinkedHashMap` 的数据排序**基于访问顺序** （最近访问的数据会在链表尾部），在容量溢出的时候，将链表头部的数据移除。从而，实现了 LRU 数据缓存机制。
- **2.**LruCache 在内部的get、put、remove包括 trimToSize 都是安全的（因为都上锁了）。
- **3.**LruCache 自身并没有释放内存，将 LinkedHashMap 的数据移除了，如果数据还在别的地方被引用了，还是有泄漏问题，还需要手动释放内存。
- **4.**覆写 `entryRemoved` 方法能知道 LruCache 数据移除是是否发生了冲突，也可以去手动释放资源。
- **5.** `maxSize` 和 `sizeOf(K key, V value)` 方法的覆写息息相关，必须相同单位。（比如 maxSize 是7MB，自定义的 sizeOf 计算每个数据大小的时候必须能算出与MB之间有联系的单位）

7）DiskLruCache 源码解析

（1）存储位置

`DiskLruCache` 并没有限制数据的缓存位置，可以自由地进行设定，但是通常情况下多数应用程序都会将缓存的位置选择为 `/sdcard/Android/data/<application package>/cache` 这个路径。选择在这个位置有两点好处：第一，这是存储在SD卡上的，因此即使缓存再多的数据也不会对手机的内置存储空间有任何影响，只要SD卡空间足够就行。第二，这个路径被Android系统认定为应用程序的缓存路径，当程序被卸载的时候，这里的数据也会一起被清除掉，这样就不会出现删除程序之后手机上还有很多残留数据的问题。

（4）打开缓存

DiskLruCache是不能new出实例的，如果我们要创建一个DiskLruCache的实例，则需要调用它的open()方法，接口如下所示：

`public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)open()` 方法接收四个参数，第一个参数指定的是数据的缓存地址，第二个参数指定当前应用程序的版本号，第三个参数指定同一个key可以对应多少个缓存文件，基本都是传1，第四个参数指定最多可以缓存多少字节的数据。

其中缓存地址前面已经说过了，通常都会存放在 `/sdcard/Android/data/<application package>/cache` 这个路径下面，但同时我们又需要考虑如果这个手机没有SD卡，或者SD正好被移除了的情况，因此比较优秀的程序都会专门写一个方法来获取缓存地址，如下所示：

```
public File getDiskCacheDir(Context context, String uniqueName) {
    String cachePath;
    if
(Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
            || !Environment.isExternalStorageRemovable()) {
        cachePath = context.getExternalCacheDir().getPath();
    } else {
        cachePath = context.getCacheDir().getPath();
    }
    return new File(cachePath + File.separator + uniqueName);
}
```

可以看到，当SD卡存在或者SD卡不可被移除的时候，就调用getExternalCacheDir()方法来获取缓存路径，否则就调用getCacheDir()方法来获取缓存路径。前者获取到的就是`/sdcard/Android/data/<application package>/cache`这个路径，而后者获取到的是`/data/data/<application package>/cache`这个路径。

接着又将获取到的路径和一个`uniqueName`进行拼接，作为最终的缓存路径返回。那么这个uniqueName又是什么呢？其实这就是为了对不同类型的数据进行区分而设定的一个唯一值，比如说在网易新闻缓存路径下看到的`bitmap、object`等文件夹。

接着是应用程序版本号，我们可以使用如下代码简单地获取到当前应用程序的版本号：

```
public int getAppVersion(Context context) {
    try {
        PackageInfo info =
context.getPackageManager().getPackageInfo(context.getPackageName(), 0);
        return info.versionCode;
    } catch (NameNotFoundException e) {
        e.printStackTrace();
    }
    return 1;
}
```

需要注意的是，每当版本号改变，缓存路径下存储的所有数据都会被清除掉，因为DiskLruCache认为当应用程序有版本更新的时候，所有的数据都应该从网上重新获取。

后面两个参数就没什么需要解释的了，第三个参数传1，第四个参数通常传入10M的大小就够了，这个可以根据自身的情况进行调节。

因此，一个非常标准的open()方法就可以这样写：

```
DiskLruCache mDiskLruCache = null;
try {
    File cacheDir = getDiskCacheDir(context, "bitmap");
    if (!cacheDir.exists()) {
        cacheDir.mkdirs();
    }
    mDiskLruCache = DiskLruCache.open(cacheDir, getAppVersion(context), 1,
10 * 1024 * 1024);
} catch (IOException e) {
    e.printStackTrace();
}
```

首先调用getDiskCacheDir()方法获取到缓存地址的路径，然后判断一下该路径是否存在，如果不存在就创建一下。接着调用DiskLruCache的open()方法来创建实例，并把四个参数传入即可。

有了DiskLruCache的实例之后，我们就可以对缓存的数据进行操作了，操作类型主要包括写入、访问、移除等，我们一个个进行学习。

（3）写入缓存

先来看写入，比如说现在有一张图片，地址是https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg，那么为了将这张图片下载下来，就可以这样写：

```
private boolean downloadUrlToStream(String urlString, OutputStream
outputStream) {
    HttpURLConnection urlConnection = null;
    BufferedOutputStream out = null;
    BufferedInputStream in = null;
    try {
        final URL url = new URL(urlString);
        urlConnection = (HttpURLConnection) url.openConnection();
        in = new BufferedInputStream(urlConnection.getInputStream(), 8 *
1024);
        out = new BufferedOutputStream(outputStream, 8 * 1024);
        int b;
        while ((b = in.read()) != -1) {
            out.write(b);
        }
        return true;
    } catch (final IOException e) {
        e.printStackTrace();
    } finally {
        if (urlConnection != null) {
            urlConnection.disconnect();
        }
        try {
            if (out != null) {
                out.close();
            }
```

```
            if (in != null) {
                in.close();
            }
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }
    return false;
}
```

这段代码相当基础，相信大家都看得懂，就是访问 `urlString` 中传入的网址，并通过 `outputStream` 写入到本地。有了这个方法之后，下面我们就可以使用 `DiskLruCache` 来进行写入了，写入的操作是借 `DiskLruCache.Editor` 这个类完成的。类似地，这个类也是不能new的，需要调用DiskLruCache的edit()方法来获取实例，接口如下所示：`public Editor edit(String key) throws IOException` 可以看到，edit()方法接收一个参数key，这个key将会成为缓存文件的文件名，并且必须要和图片的URL是一一对应的。那么怎样才能让key和图片的URL能够一一对应呢？直接使用URL来作为key？不太合适，因为图片URL中可能包含一些特殊字符，这些字符有可能在命名文件时是不合法的。其实最简单的做法就是将图片的URL进行MD5编码，编码后的字符串肯定是唯一的，并且只会包含0-F这样的字符，完全符合文件的命名规则。

那么我们就写一个方法用来将字符串进行MD5编码，代码如下所示：

```
public String hashKeyForDisk(String key) {
    String cacheKey;
    try {
        final MessageDigest mDigest = MessageDigest.getInstance("MD5");
        mDigest.update(key.getBytes());
        cacheKey = bytesToHexString(mDigest.digest());
    } catch (NoSuchAlgorithmException e) {
        cacheKey = String.valueOf(key.hashCode());
    }
    return cacheKey;
}

private String bytesToHexString(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < bytes.length; i++) {
        String hex = Integer.toHexString(0xFF & bytes[i]);
        if (hex.length() == 1) {
            sb.append('0');
        }
        sb.append(hex);
    }
    return sb.toString();
}

```

代码很简单，现在我们只需要调用一下hashKeyForDisk()方法，并把图片的URL传入到这个方法中，就可以得到对应的key了。

因此，现在就可以这样写来得到一个DiskLruCache.Editor的实例：

```
String imageUrl = "https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg";
String key = hashKeyForDisk(imageUrl);
DiskLruCache.Editor editor = mDiskLruCache.edit(key);
```

有了 `DiskLruCache.Editor` 的实例之后，我们可以调用它的 `newOutputStream()` 方法来创建一个输出流，然后把它传入到 `downloadUrlToStream()` 中就能实现下载并写入缓存的功能了。注意 `newOutputStream()` 方法接收一个 `index` 参数，由于前面在设置 `valueCount` 的时候指定的是1，所以这里 `index` 传0就可以了。在写入操作执行完之后，我们还需要调用一下 `commit()` 方法进行提交才能使写入生效，调用 `abort()` 方法的话则表示放弃此次写入。

因此，一次完整写入操作的代码如下所示：

```
new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            String imageUrl = "https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg";
            String key = hashKeyForDisk(imageUrl);
            DiskLruCache.Editor editor = mDiskLruCache.edit(key);
            if (editor != null) {
                OutputStream outputStream = editor.newOutputStream(0);
                if (downloadUrlToStream(imageUrl, outputStream)) {
                    editor.commit();
                } else {
                    editor.abort();
                }
            }
            mDiskLruCache.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}).start();
```

由于这里调用了 `downloadUrlToStream()` 方法来从网络上下载图片，所以一定要确保这段代码是在子线程当中执行的。注意在代码的最后我还调用了一下flush()方法，这个方法并不是每次写入都必须要调用的，但在这里却不可缺少，我会在后面说明它的作用。

现在的话缓存应该是已经成功写入了，我们进入到SD卡上的缓存目录里看一下，如下图所示：

可以看到，这里有一个文件名很长的文件，和一个 `journal` 文件，那个文件名很长的文件自然就是缓存的图片了，因为是使用了MD5编码来进行命名的。

（4）读取缓存

缓存已经写入成功之后，接下来我们就该学习一下如何读取了。读取的方法要比写入简单一些，主要是借助`DiskLruCache`的`get()`方法实现的，接口如下所示：

`public synchronized Snapshot get(String key) throws IOException`很明显，`get()`方法要求传入一个key来获取到相应的缓存数据，而这个key毫无疑问就是将图片URL进行MD5编码后的值了，因此读取缓存数据的代码就可以这样写：

```
String imageUrl = "https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg";
String key = hashKeyForDisk(imageUrl);
DiskLruCache.Snapshot snapShot = mDiskLruCache.get(key);
```

很奇怪的是，这里获取到的是一个`DiskLruCache.Snapshot`对象，这个对象我们该怎么利用呢？很简单，只需要调用它的`getInputStream()`方法就可以得到缓存文件的输入流了。同样地，`getInputStream()`方法也需要传一个index参数，这里传入0就好。有了文件的输入流之后，想要把缓存图片显示到界面上就轻而易举了。所以，一段完整的读取缓存，并将图片加载到界面上的代码如下所示：

```
try {
    String imageUrl = "https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg";
    String key = hashKeyForDisk(imageUrl);
    DiskLruCache.Snapshot snapShot = mDiskLruCache.get(key);
    if (snapShot != null) {
        InputStream is = snapShot.getInputStream(0);
        Bitmap bitmap = BitmapFactory.decodeStream(is);
        mImage.setImageBitmap(bitmap);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

我们使用了`BitmapFactory`的`decodeStream()`方法将文件流解析成`Bitmap`对象，然后把它设置到`ImageView`当中。如果运行一下程序，将会看到如下效果：

OK，图片已经成功显示出来了。注意这是我们从本地缓存中加载的，而不是从网络上加载的，因此即使在你手机没有联网的情况下，这张图片仍然可以显示出来。

（5）移除缓存

学习完了写入缓存和读取缓存的方法之后，最难的两个操作你就都已经掌握了，那么接下来要学习的移除缓存对你来说也一定非常轻松了。移除缓存主要是借助`DiskLruCache`的`remove()`方法实现的，接口如下所示：

`public synchronized boolean remove(String key) throws IOException`相信你已经相当熟悉了，`remove()`方法中要求传入一个key，然后会删除这个key对应的缓存图片，示例代码如下：

```
try {
    String imageUrl = "https://img-my.csdn.net/uploads/201309/01/1378037235_7476.jpg";
    String key = hashKeyForDisk(imageUrl);
    mDiskLruCache.remove(key);
} catch (IOException e) {
    e.printStackTrace();
}
```

用法虽然简单，但是你要知道，这个方法我们并不应该经常去调用它。因为你完全不需要担心缓存的数据过多从而占用SD卡太多空间的问题，DiskLruCache会根据我们在调用open()方法时设定的缓存最大值来自动删除多余的缓存。只有你确定某个key对应的缓存内容已经过期，需要从网络获取最新数据的时候才应该调用remove()方法来移除缓存。

（6）其它API

除了写入缓存、读取缓存、移除缓存之外，DiskLruCache还提供了另外一些比较常用的API，我们简单学习一下。

1.size()

这个方法会返回当前缓存路径下所有缓存数据的总字节数，以byte为单位，如果应用程序中需要在界面上显示当前缓存数据的总大小，就可以通过调用这个方法计算出来。

2.flush()

这个方法用于将内存中的操作记录同步到日志文件（也就是journal文件）当中。这个方法非常重要，因为`DiskLruCache`能够正常工作的前提就是要依赖于journal文件中的内容。前面在讲解写入缓存操作的时候我有调用过一次这个方法，但其实并不是每次写入缓存都要调用一次`flush()`方法的，频繁地调用并不会带来任何好处，只会额外增加同步journal文件的时间。比较标准的做法就是在`Activity`的`onPause()`方法中去调用一次`flush()`方法就可以了。

3.close()

这个方法用于将`DiskLruCache`关闭掉，是和`open()`方法对应的一个方法。关闭掉了之后就不能再调用`DiskLruCache`中任何操作缓存数据的方法，通常只应该在Activity的`onDestroy()`方法中去调用close()方法。

4.delete()

这个方法用于将所有的缓存数据全部删除，比如说网易新闻中的那个手动清理缓存功能，其实只需要调用一下`DiskLruCache`的`delete()`方法就可以实现了。

（7）解读journal

前面已经提到过，DiskLruCache能够正常工作的前提就是要依赖于journal文件中的内容，因此，能够读懂journal文件对于我们理解DiskLruCache的工作原理有着非常重要的作用。那么journal文件中的内容到底是什么样的呢？我们来打开瞧一瞧吧，如下图所示：

由于现在只缓存了一张图片，所以journal中并没有几行日志，我们一行行进行分析。第一行是个固定的字符串“libcore.io.DiskLruCache”，标志着我们使用的是DiskLruCache技术。第二行是DiskLruCache的版本号，这个值是恒为1的。第三行是应用程序的版本号，我们在open()方法里传入的版本号是什么这里就会显示什么。第四行是valueCount，这个值也是在open()方法中传入的，通常情况下都为1。第五行是一个空行。前五行也被称为journal文件的头，这部分内容还是比较好理解的，但是接下来的部分就要稍微动点脑筋了。

第六行是以一个DIRTY前缀开始的，后面紧跟着缓存图片的key。通常我们看到DIRTY这个字样都不代表着什么好事情，意味着这是一条脏数据。没错，每当我们调用一次DiskLruCache的edit()方法时，都会向journal文件中写入一条DIRTY记录，表示我们正准备写入一条缓存数据，但不知结果如何。然后调用commit()方法表示写入缓存成功，这时会向journal中写入一条CLEAN记录，意味着这条“脏”数据被“洗干净了”，调用abort()方法表示写入缓存失败，这时会向journal中写入一条REMOVE记录。也就是说，每一行DIRTY的key，后面都应该有一行对应的CLEAN或者REMOVE的记录，否则这条数据就是“脏”的，会被自动删除掉。

如果你足够细心的话应该还会注意到，第七行的那条记录，除了CLEAN前缀和key之外，后面还有一个152313，这是什么意思呢？其实，DiskLruCache会在每一行CLEAN记录的最后加上该条缓存数据的大小，以字节为单位。152313也就是我们缓存的那张图片的字节数了，换算出来大概是148.74K，和缓存图片刚刚好一样大，如下图所示：

前面我们所学的size()方法可以获取到当前缓存路径下所有缓存数据的总字节数，其实它的工作原理就是把journal文件中所有CLEAN记录的字节数相加，求出的总合再把它返回而已。

除了`DIRTY、CLEAN、REMOVE`之外，还有一种前缀是READ的记录，这个就非常简单了，每当我们调用get()方法去读取一条缓存数据时，就会向journal文件中写入一条READ记录。因此，像网易新闻这种图片和数据量都非常大的程序，journal文件中就可能会有大量的READ记录。

那么你可能会担心了，如果我不停频繁操作的话，就会不断地向journal文件中写入数据，那这样journal文件岂不是会越来越大？这倒不必担心，DiskLruCache中使用了一个redundantOpCount变量来记录用户操作的次数，每执行一次写入、读取或移除缓存的操作，这个变量值都会加1，当变量值达到2000的时候就会触发重构journal的事件，这时会自动把journal中一些多余的、不必要的记录全部清除掉，保证journal文件的大小始终保持在一个合理的范围内。

好了，这样的话我们就算是把DiskLruCache的用法以及简要的工作原理分析完了。至于DiskLruCache的源码还是比较简单的， 限于篇幅原因就不在这里展开了，感兴趣的朋友可以自己去摸索。

8.Handler机制

通信的同步（Synchronous）：指向客户端发送请求后，**必须要在服务端有回应后客户端才继续发送其它的请求**，所以这时所有请求将会在服务端得到同步，直到服务端返回请求。

通信的异步（Asynchronous）：指客户端在发送请求后，**不必等待服务端的回应就可以发送下一个请求**。

所谓**同步调用**，就是在一个函数或方法调用时，没有得到结果之前，该调用就不返回，直到返回结果。**异步调用**和同步是相对的，在一个异步调用发起后，**被调用者立即返回给调用者**，但是调用者不能立刻得到结果，被调用都在实际处理这个调用的请求完成后，通过状态、通知或回调等方式来通知调用者请求处理的结果。

android的消息处理有三个核心类：Looper,Handler和Message。其实还有一Message Queue（消息队列），但是MQ被封装到Looper里面了，我们不会直接与MQ打交道，所以它不算是个核心类。

（1）消息类：Message类

android.os.Message的主要功能是进行消息的封装，同时可以指定消息的操作形式，Message类定义的变量和常用方法如下：

（1）`public int what`：变量，用于定义此Message属于何种操作

（2）`public Object obj`：变量，用于定义此Message传递的信息数据，通过它传递信息

（3）`public int arg1`：变量，传递一些整型数据时使用

（4）`public int arg2`：变量，传递一些整型数据时使用

（5）`public Handler getTarget()`：普通方法，取得操作此消息的Handler对象。

在整个消息处理机制中，**message**又叫task，**封装了**任务携带的**信息**和处理该任务的**handler**。
message的用法比较简单，但是有这么几点需要注意：

（1）尽管Message有public的默认构造方法，但是你应该通过 `Message.obtain()` 来从消息池中获得空消息对象，以节省资源。

（2）如果你的message只需要携带简单的**int**信息，请优先使用 `Message.arg1` 和 `Message.arg2` 来传递信息，这比用Bundle更省内存

（3）擅用 `message.what` 来**标识信息**，以便用不同方式处理message。

（4）使用 `setData()` 存放 `Bundle` 对象。？？？

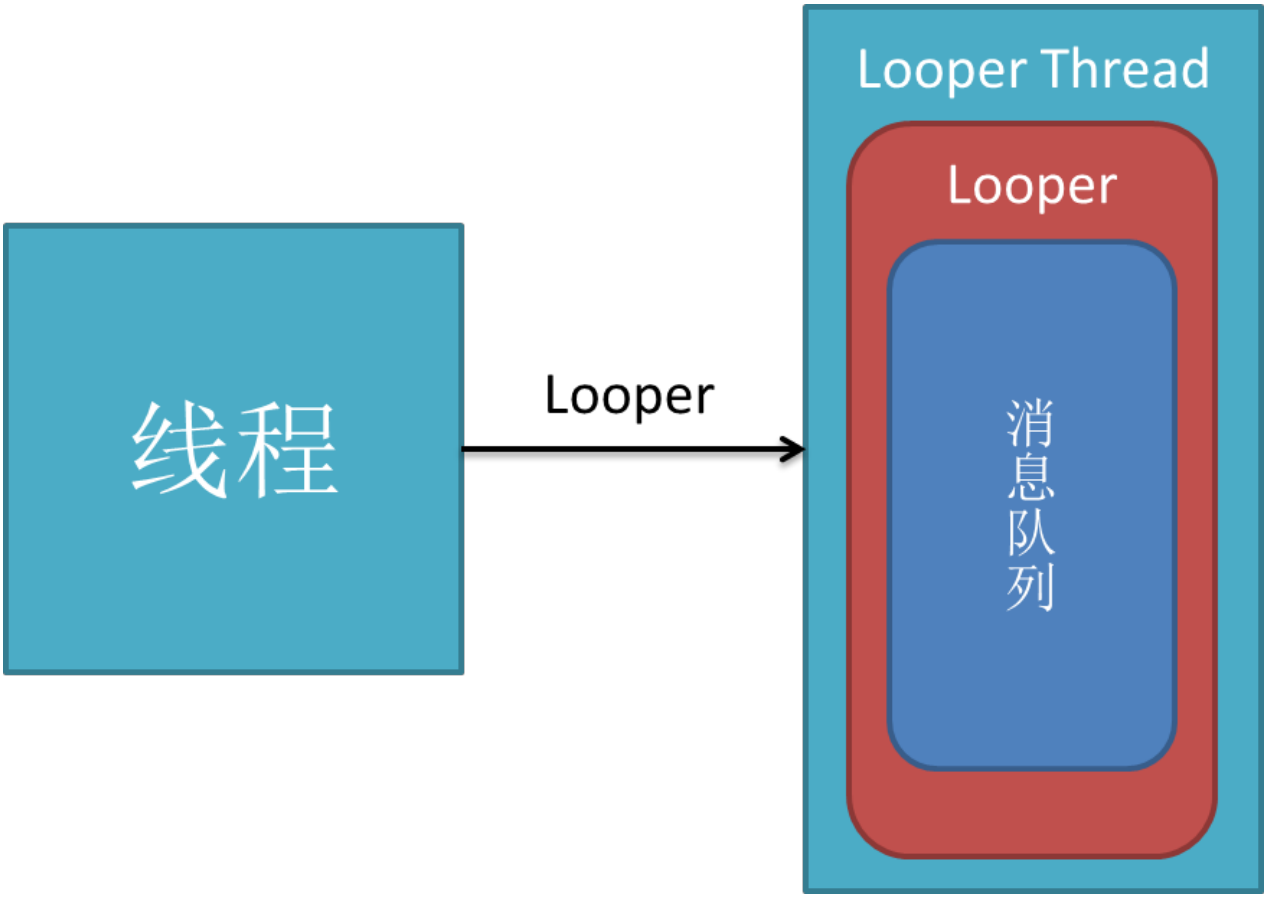
（2）消息通道：Looper

在使用Handler处理Message时，需要**Looper**（通道）来完成。在一个**Activity中，系统会自动帮用户启动Looper对象**，而在一个用户**自定义的类中，则需要用户手工调用Looper类中的方法，然后才可以正常启动Looper对象**。Looper的字面意思是“循环者”，它被设计**用来使一个普通线程变成Looper线程**。所谓Looper线程就是循环工作的线程。在程序开发中（尤其是GUI开发中），我们经常会需要一个线程不断循环，一旦有新任务则执行，执行完继续等待下一个任务，这就是Looper线程。使用Looper类创建Looper线程很简单：

```
public class LooperThread extends Thread {
    @Override
    public void run() {
        // 将当前线程初始化为Looper线程
        Looper.prepare();

        // ...其他处理，如实例化handler

        // 开始循环处理消息队列
        Looper.loop();
    }
}
```

通过上面两行核心代码，你的线程就升级为Looper线程了！那么这两行代码都做了些什么呢?

1)Looper.prepare()：创建Loop而对象。



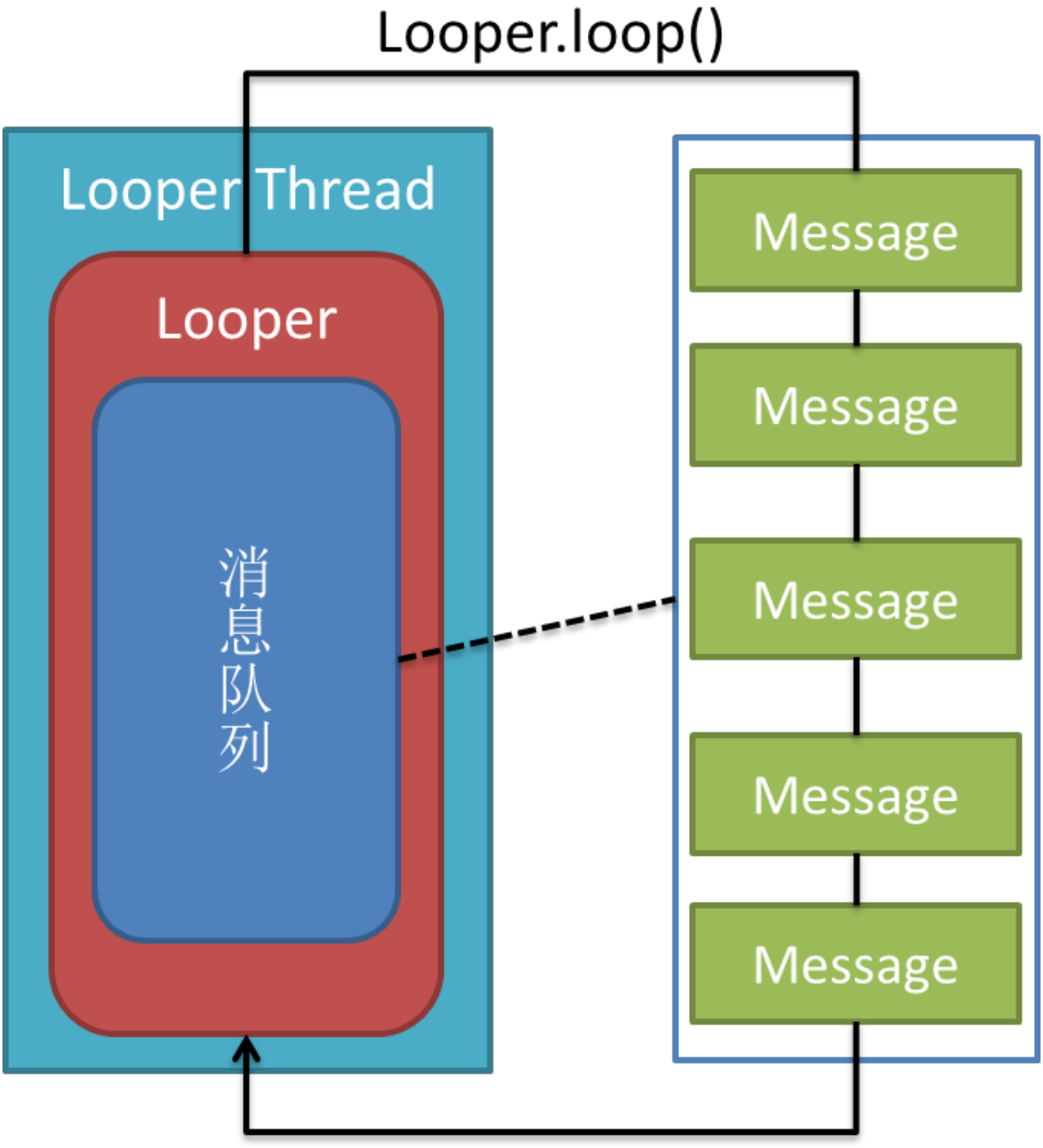
通过上图可以看到，现在你的线程中有一个Looper对象，它的内部维护了一个消息队列MQ。注意，**一个Thread只能有一个Looper对象**，为什么呢？来看一下源码

```
public class Looper {
    // 每个线程中的Looper对象其实是一个ThreadLocal，即线程本地存储(TLS)对象
    private static final ThreadLocal sThreadLocal = new ThreadLocal();
    // Looper内的消息队列
    final MessageQueue mQueue;
    // 当前线程
    Thread mThread;
    //其他属性
    // 每个Looper对象中有它的消息队列，和它所属的线程
    private Looper() {
        mQueue = new MessageQueue();
        mRun = true;
        mThread = Thread.currentThread();
    }
    // 我们调用该方法会在调用线程的TLS中创建Looper对象
    public static final void prepare() {
        if (sThreadLocal.get() != null) {
            // 试图在有Looper的线程中再次创建Looper将抛出异常
            throw new RuntimeException("Only one Looper may be created per
thread");
        }
        sThreadLocal.set(new Looper());
```

```
    }
    // 其他方法
}
```

prepare()背后的工作方式一目了然，其核心就是将looper对象定义为ThreadLocal。

2）Looper.loop()：循环获取MQ中的消息，并发送给相应Handler对象。



调用loop方法后，Looper线程就开始真正工作了，它不断从自己的MQ中取出队头的消息(也叫任务)执行。其源码分析如下：

```
public static final void loop() {
        Looper me = myLooper();  //得到当前线程Looper
        MessageQueue queue = me.mQueue;  //得到当前looper的MQ
```

```
4
5           Binder.clearCallingIdentity();
6           final long ident = Binder.clearCallingIdentity();
7           // 开始循环
8           while (true) {
9               Message msg = queue.next(); // 取出message
10              if (msg != null) {
11                  if (msg.target == null) {
12                      // message没有target为结束信号，退出循环
13                      return;
14                  }
15                  // 日志
16                  if (me.mLogging!= null) me.mLogging.println(
17                          ">>>>> Dispatching to " + msg.target + " "
18                          + msg.callback + ": " + msg.what
19                          );
20                  // 非常重要！将真正的处理工作交给message的target，即后面要讲的
    handler
21                  msg.target.dispatchMessage(msg);
22                  // 日志
23                  if (me.mLogging!= null) me.mLogging.println(
24                          "<<<<< Finished to    " + msg.target + " "
25                          + msg.callback);
26
27                  final long newIdent = Binder.clearCallingIdentity();
28                  if (ident != newIdent) {
29                      Log.wtf("Looper", "Thread identity changed from 0x"
30                              + Long.toHexString(ident) + " to 0x"
31                              + Long.toHexString(newIdent) + " while
    dispatching to "
32                              + msg.target.getClass().getName() + " "
33                              + msg.callback + " what=" + msg.what);
34                  }
35                  // 回收message资源
36                  msg.recycle();
37              }
38          }
39      }
```

除了prepare()和loop()方法，Looper类还提供了一些有用的方法，比如**Looper.myLooper()得到当前线程looper对象**：

```
1       public static final Looper myLooper() {
2           // 在任意线程调用Looper.myLooper()返回的都是那个线程的looper
3           return (Looper)sThreadLocal.get();
4       }
```

getThread()得到looper对象所属线程：

```
    public Thread getThread() {
        return mThread;
    }
```

quit()方法结束looper循环：

```
    public void quit() {
        // 创建一个空的message，它的target为NULL，表示结束循环消息
        Message msg = Message.obtain();
        // 发出消息
        mQueue.enqueueMessage(msg, 0);
    }
```

综上，Looper有以下几个要点：

1）**每个线程有且只能有一个Looper对象**，它是一个ThreadLocal

2）Looper内部**有一个消息队列**，loop()方法调用后线程开始不断从队列中取出消息执行

3）**Looper使一个线程变成Looper线程**。

（3）消息操作类：Handler类

Message对象封装了所有的消息，而这些消息的操作需要android.os.Handler类完成。什么是handler？**handler起到了处理MQ上的消息**的作用（只处理由自己发出的消息），即**通知MQ它要执行一个任务(sendMessage)，并在loop到自己的时候执行该任务(handleMessage)，整个过程是异步的。handler创建时会关联一个looper，默认的构造方法将关联当前线程的looper，不过这也是可以set的**。默认的构造方法：

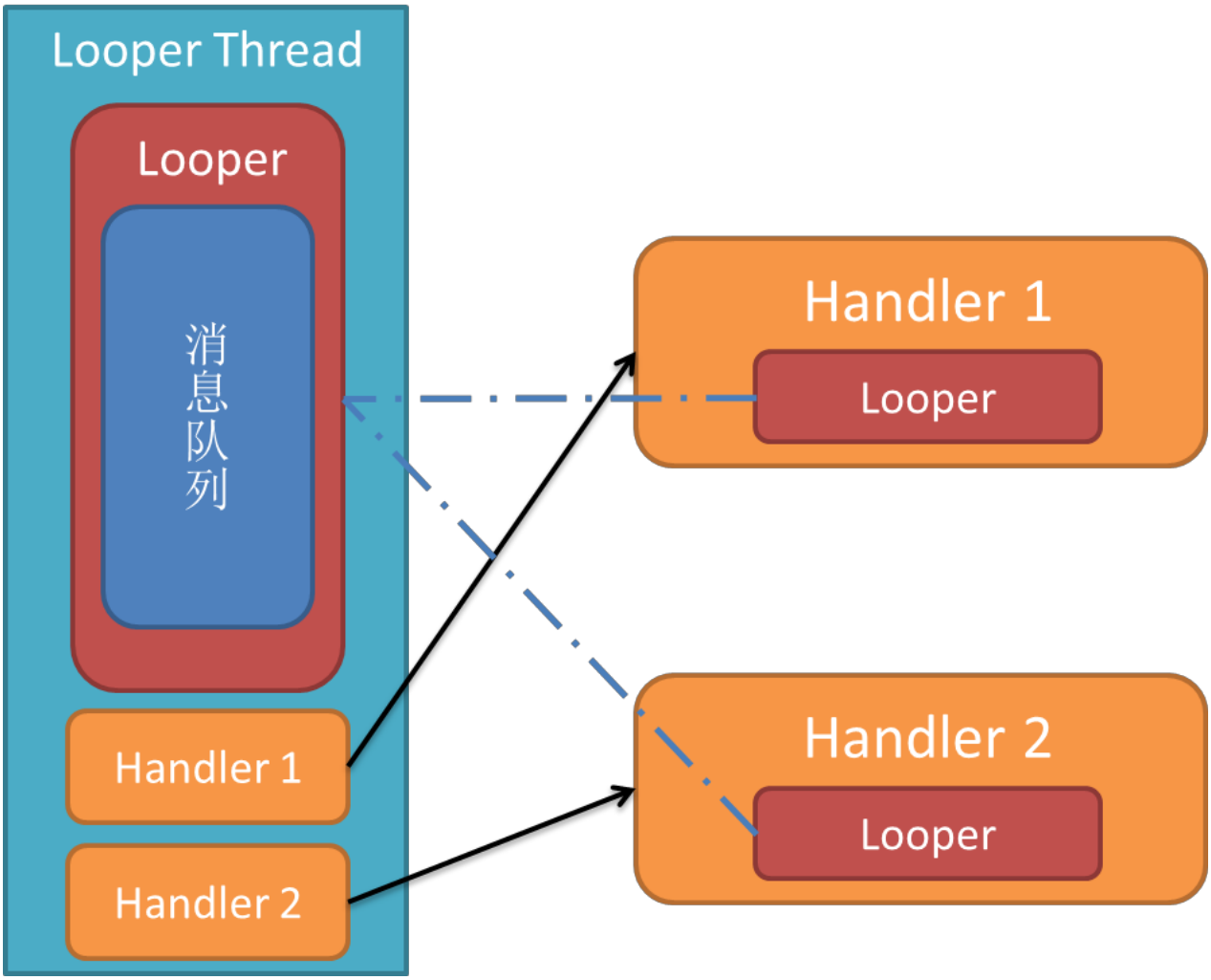
```
public class handler {
    final MessageQueue mQueue;  // 关联的MQ
    final Looper mLooper;  // 关联的looper
    final Callback mCallback;
    // 其他属性
    public Handler() {
        if (FIND_POTENTIAL_LEAKS) {
            final Class<? extends Handler> klass = getClass();
            if ((klass.isAnonymousClass() || klass.isMemberClass() ||
klass.isLocalClass()) &&
                    (klass.getModifiers() & Modifier.STATIC) == 0) {
                Log.w(TAG, "The following Handler class should be static
or leaks might occur: " + klass.getCanonicalName());
            }
        }
        // 默认将关联当前线程的looper
        mLooper = Looper.myLooper();
        // looper不能为空，即该默认的构造方法只能在looper线程中使用
        if (mLooper == null) {
            throw new RuntimeException(
```

```
                "Can't create handler inside thread that has not called Looper.prepare()");
        }
        // 重要！！！直接把关联looper的MQ作为自己的MQ，因此它的消息将发送到关联looper的MQ上
        mQueue = mLooper.mQueue;
        mCallback = null;
    }

    // 其他方法
}
```

下面我们就可以为之前的LooperThread类加入Handler：

```
public class LooperThread extends Thread {
    private Handler handler1;
    private Handler handler2;

    @Override
    public void run() {
        // 将当前线程初始化为Looper线程
        Looper.prepare();

        // 实例化两个handler
        handler1 = new Handler();
        handler2 = new Handler();

        // 开始循环处理消息队列
        Looper.loop();
    }
}
```

加入handler后的效果如下图：



可以看到，**一个线程可以有多个Handler，但是只能有一个Looper！**

Handler发送消息

有了handler之后，我们就可以使用

post(Runnable)

postAtTime(Runnable, long)

postDelayed(Runnable, long)

sendEmptyMessage(int)

sendMessage(Message)

sendMessageAtTime(Message, long)

sendMessageDelayed(Message, long)

这些方法向MQ上发送消息了。**光看这些API你可能会觉得handler能发两种消息，一种是Runnable对象，一种是message对象，这是直观的理解，但其实post发出的Runnable对象最后都被封装成message对象了**，见源码：

```
1  // 此方法用于向关联的MQ上发送Runnable对象，它的run方法将在handler关联的looper线程中执行
2      public final boolean post(Runnable r)
```

```
    {
       // 注意getPostMessage(r)将runnable封装成message
       return  sendMessageDelayed(getPostMessage(r), 0);
    }

    private final Message getPostMessage(Runnable r) {
        Message m = Message.obtain();  //得到空的message
        m.callback = r;  //将runnable设为message的callback,
        return m;
    }

    public boolean sendMessageAtTime(Message msg, long uptimeMillis)
    {
        boolean sent = false;
        MessageQueue queue = mQueue;
        if (queue != null) {
            msg.target = this;  // message的target必须设为该handler!
            sent = queue.enqueueMessage(msg, uptimeMillis);
        }
        else {
            RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with no mQueue");
            Log.w("Looper", e.getMessage(), e);
        }
        return sent;
    }
```

通过handler发出的message有如下特点：

1.**message.target为该handler对象**，这确保了looper执行到该message时能找到处理它的handler，即loop()方法中的关键代码

```
msg.target.dispatchMessage(msg);
```

2.post发出的message，**其callback为Runnable对象**

Handler处理消息

说完了消息的发送，再来看下handler如何处理消息。消息的处理是通过核心方法dispatchMessage(Message msg)与钩子方法handleMessage(Message msg)

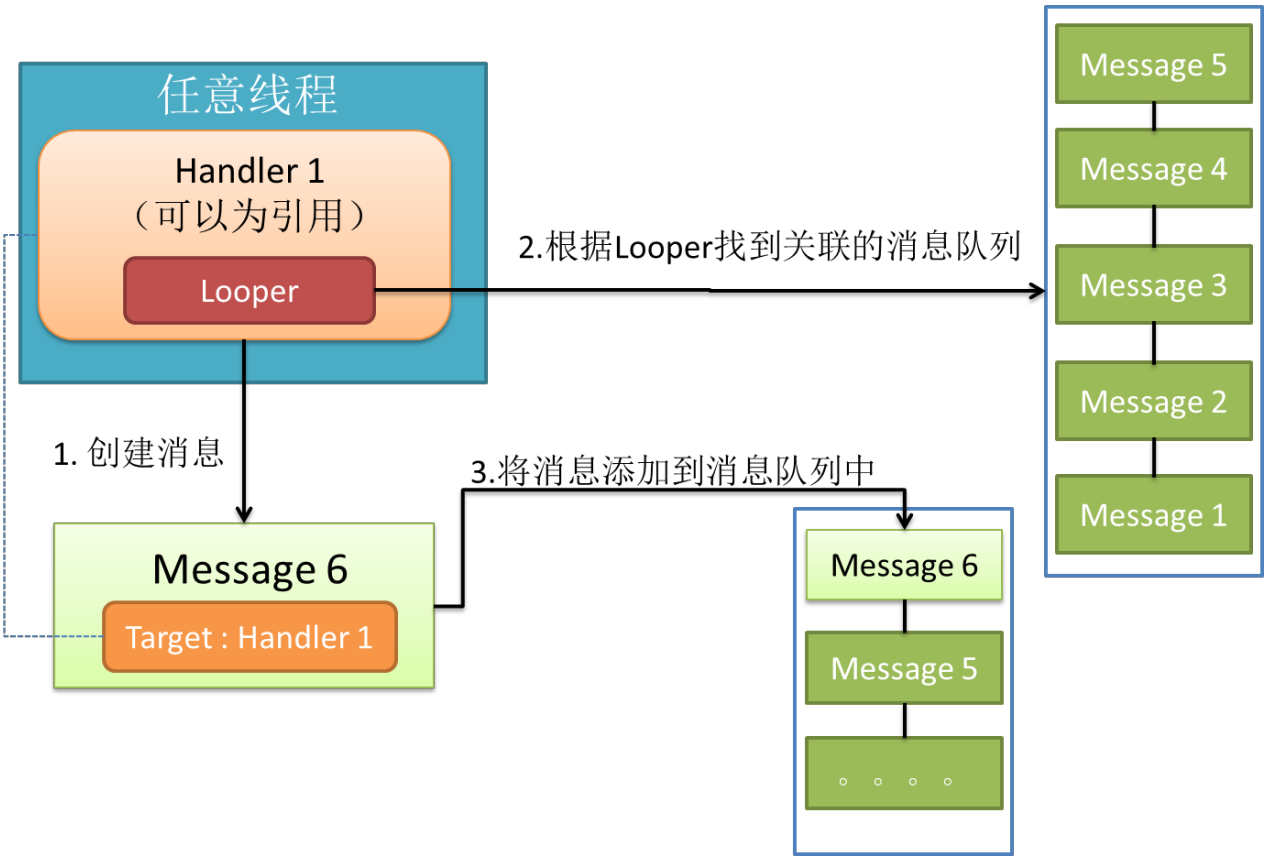
完成的，见源码

```
/**
 * Handle system messages here.
 */
    public void dispatchMessage(Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
```

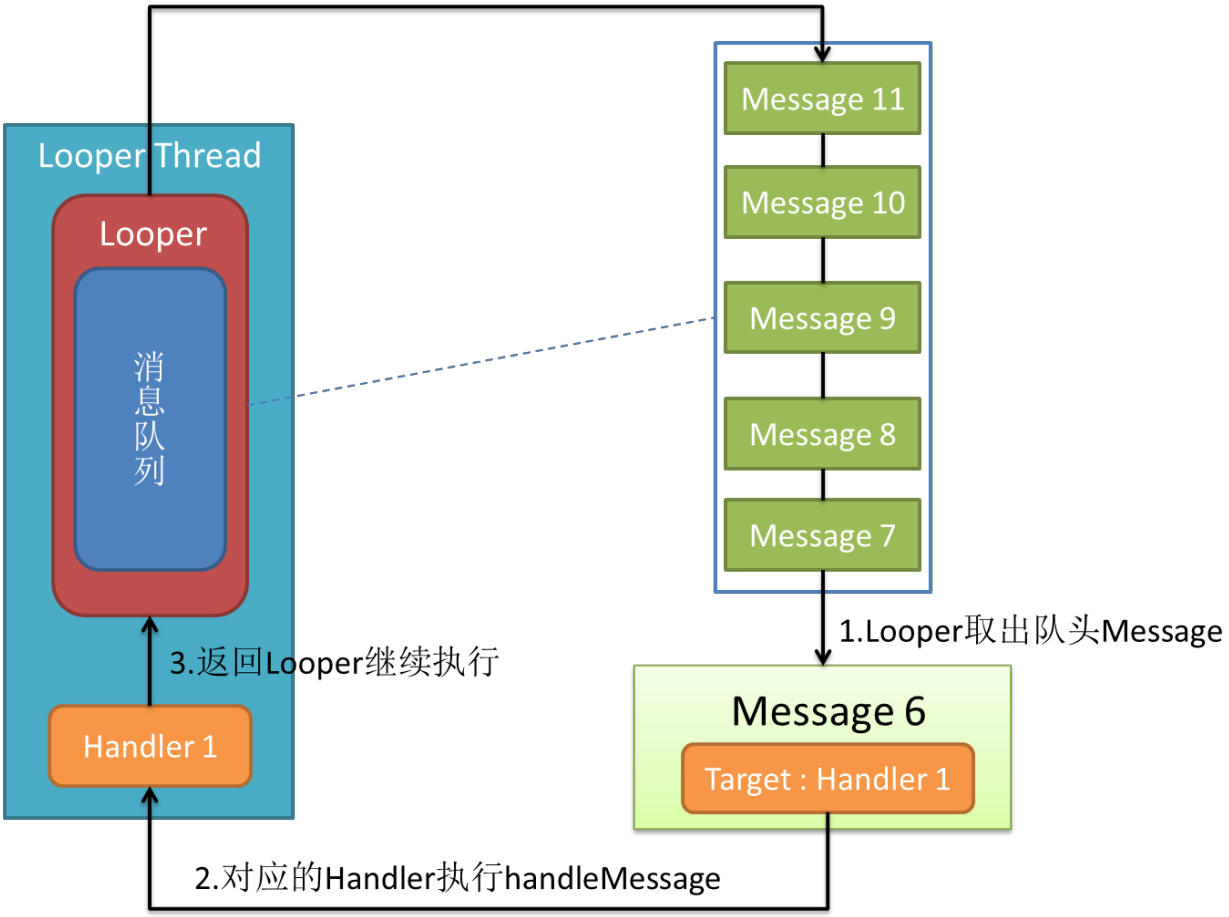
```
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            handleMessage(msg);
        }
    }
```

可以看到，除了handleMessage(Message msg)和Runnable对象的run方法由开发者实现外（实现具体逻辑），handler的内部工作机制对开发者是透明的。Handler拥有下面两个重要的特点：

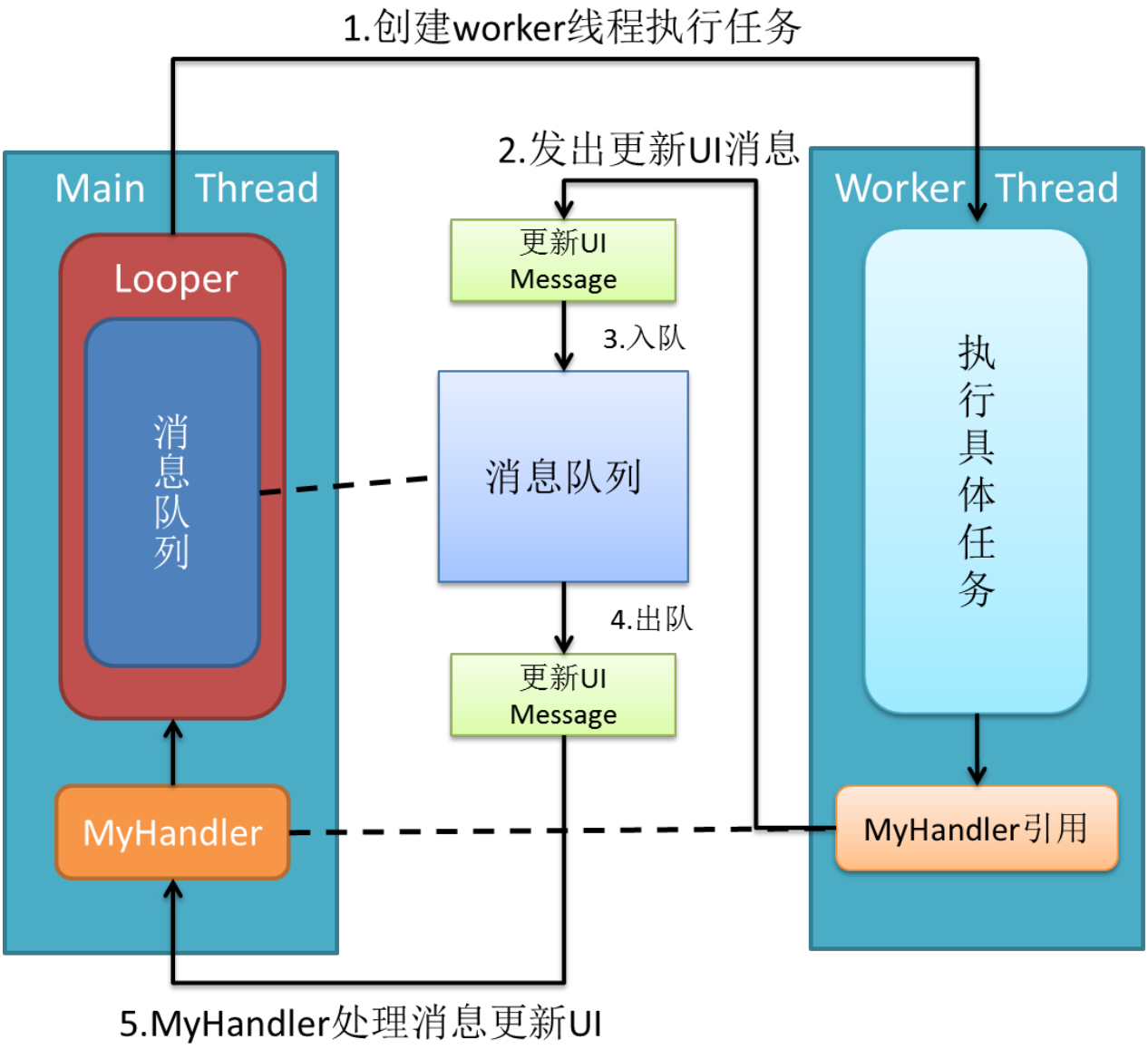
1)handler可以在**任意线程发送消息**，这些消息会被添加到关联的MQ上



2)**消息的处理是通过核心方法dispatchMessage(Message msg)与钩子方法handleMessage(Message msg)完成的**，handler是在它**关联的looper线程中处理消息**的。



这就解决了android最经典的不能在其他非主线程中更新UI的问题。**android的主线程也是一个looper线程**(looper在android中运用很广)，我们在其中创建的handler默认将关联主线程MQ。因此，利用**handler的一个solution就是在activity中创建handler并将其引用传递给worker thread，worker thread执行完任务后使用handler发送消息通知activity更新UI**。(过程如图)



（4）Android 运行的进程

为了更好的了解Handler的机制,我们应该首先,将Android系统整个运行进程都要烂熟于心,下面是**android 进程运行图**:

从图中我们可以看到,**当我们从外部调用组件的时候,Service 和 ContentProvider 是从线程池那里获取线程,而Activity 和BroadcastReceiver是直接在主线程运行**,为了,追踪线程,我们可以用 debug 方法,或者使用一个工具类,这里,我们创建一个用于监视线程的工具类

```
/**
 * @author Tom_achai
 * @date 2011-11-20
 *
*/
public class Utils {
   public static long getThreadId(){
        Thread t = Thread.currentThread();
```

```
        return t.getId();
    }

    /**
      * 获取单独线程信息
      * @return
     */
    public static String getThreadSignature(){
        Thread t = Thread.currentThread();
        long l = t.getId();
        String name = t.getName();
        long p = t.getPriority();
        String gname = t.getThreadGroup().getName();
        return ("(Thread):"+name+":(id)"+ l +"(:priority)" + p + ":
(group)" + gname );
    }



    /**
     *获取当前线程 信息
     */
    public static void logThreadSignature(){
        Log.d("ThreadUtils", getThreadSignature());
    }

    public static void logThreadSignature(String name ){
        Log.d("ThreadUtils", name + ":"+getThreadSignature());
    }

    public static void sleepForInSecs(int secs){
        try{
            Thread.sleep(secs * 1000);
        }catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }

    /**
      * 讲String放进Bundle 中
      * @param message
      * @return
     */
    public static Bundle getStringAsBundle(String message){
        Bundle b = new Bundle();
        b.putString("message", message);
        return b;
    }
```

```
57
58      /**
59        *
60        * 获取Bundle的String
61        * @param b
62        * @return
63       */
64      public static String getStringFromABundle(Bundle b){
65          return b.getString("message");
66      }
67  }
```

有了这样一个类就可以方便我们观察线程的运行

好了,现在准备好以后就进入正题Handler

(5) Handlers

为什么要使用Handlers?

因为,当我们的主线程队列,如果处理一个消息超过5秒,android 就会抛出一个 ANR(无响应)的消息,所以,我们需要把一些要处理比较长的消息,放在一个单独线程里面处理,把处理以后的结果,返回给主线程运行,就需要用的**Handler来进行线程建的通信,关系如下图**;

下面是**Handler,Message,Message Queue 之间的关系图**

这个图有4个地方关系到handlers

1, 主线程(Main thread)

2, 主线程队列(Main thread queue)

3,Hanlder

4,Message

上面的四个地方,主线程,和主线程的队列我们无需处理,所以,我们主要是处理Handler 和 Message 之间的关系.

我们每发出一个Message,Message就会落在主线程的队列当中,然后,Handler就可以调用Message绑定的数据,对主线程的组件进行操作.

（6）Message

作为handler接受的对象,我们有必要知道Message这个数据类型是个怎样的数据类型

从官方文档中我们可以知道message 关于数据的字段

public int what	
public int arg1	
public int arg2	
public Object obj	

从上面的表格可以看出,message 提供了一个对象来存储对象,而且,还提供了三个int字段用来存储少量int类型

当然,除了以上三个Message 自有的字段外,我们还可以通过**setData(Bundle b)**,来存储一个Bundle对象,来存储更丰富的数据类型,例如,图片等等.

在初始化我们的message的时候就可以为我们的Message默认字段赋值,注意赋值顺序!!!

```
Message msg = obtainMessage();
//设置我们what 字段的初值，注意顺序！！！
Message msg = mHandler.obtainMessage(int what);

//下面同理
Message msg = mHandler.obtainMessage(int what,Object object);
Message msg = mHandler.obtainMessage(int what,int arg1,int arg2);
Message msg = mHandler.obtainMessage(int what,int arg1,int arg2,
Object obj);
```

handler机制的原理 （原理图）

andriod提供了Handler 和 Looper 来满足线程间的通信。Handler先进先出原则。Looper类用来管理特定线程内对象之间的消息交换(MessageExchange)。

1)Looper: 一个线程可以产生一个Looper对象，由它来管理此线程里的MessageQueue(消息队列)。 2)Handler: 你可以构造Handler对象来与Looper沟通，以便push新消息到MessageQueue里;或者接收Looper从Message Queue取出)所送来的消息。

3) Message Queue(消息队列):用来存放线程放入的消息。

线程：**UIthread 通常就是main thread**，而Android启动程序时会替它建立一个MessageQueue。

1.Handler创建消息

每一个消息都需要被指定的Handler处理，通过Handler创建消息便可以完成此功能。Android消息机制中引入了消息池。Handler创建消息时首先查询消息池中是否有消息存在，如果有直接从消息池中取得，如果没有则重新初始化一个消息实例。使用消息池的好处是：消息不被使用时，并不作为垃圾回收，而是放入消息池，可供下次Handler创建消息时使用。消息池提高了消息对象的复用，减少系统垃圾回收的次数。

2.Handler发送消息

UI主线程初始化第一个Handler时会通过ThreadLocal创建一个Looper，该**Looper与UI主线程一一对应**。使用ThreadLocal的目的是保证每一个线程只创建唯一一个Looper。之后其他Handler初始化的时候直接获取第一个Handler创建的Looper。Looper初始化的时候会创建一个消息队列MessageQueue。至此，**主线程、消息循环、消息队列之间的关系是1:1:1**。

Hander持有对UI主线程消息队列MessageQueue和消息循环Looper的引用，子线程可以通过Handler将消息发送到UI线程的消息队列MessageQueue中。

3.Handler处理消息

UI主线程通过Looper循环查询消息队列UI_MQ，当发现有消息存在时会将消息从消息队列中取出。首先分析消息，通过消息的参数判断该消息对应的Handler，然后将消息分发到指定的Handler进行处理。

Android 异步消息处理机制 让你深入理解 Looper、Handler、Message三者关系

很多人面试肯定都被问到过，请问Android中的Looper , Handler , Message有什么关系？本篇博客目的首先为大家从源码角度介绍3者关系，然后给出一个容易记忆的结论。

1、 概述

Handler 、 Looper 、Message 这三者都与Android异步消息处理线程相关的概念。那么什么叫异步消息处理线程呢？异步消息处理线程启动后会进入一个无限的循环体之中，每循环一次，从其内部的消息队列中取出一个消息，然后回调相应的消息处理函数，执行完成一个消息后则继续循环。若消息队列为空，线程则会阻塞等待。

说了这一堆，那么和Handler 、 Looper 、Message有啥关系？其实Looper负责的就是创建一个MessageQueue，然后进入一个无限循环体不断从该MessageQueue中读取消息，而消息的创建者就是一个或多个Handler 。

2、源码解析

Looper

对于Looper主要是prepare()和loop()两个方法。 首先看prepare()方法

```
1  public static final void prepare() {
2       if (sThreadLocal.get() != null) {
3           throw new RuntimeException("Only one Looper may be created
   per thread");
4       }
5       sThreadLocal.set(new Looper(true));
6  }
```

sThreadLocal是一个ThreadLocal对象，可以在一个线程中存储变量。可以看到，**在第5行，将一个Looper的实例放入了ThreadLocal**，并且**2-4行判断了sThreadLocal是否为null，否则抛出异常**。这也就说明了Looper.prepare()方法不能被调用两次，同时也**保证了一个线程中只有一个Looper实例**~相信有些哥们一定遇到这个错误。 下面看Looper的构造方法：

```
private Looper(boolean quitAllowed) {
     mQueue = new MessageQueue(quitAllowed);
     mRun = true;
     mThread = Thread.currentThread();
}
```

在构造方法中，创建了一个MessageQueue（消息队列）。然后我们看loop()方法：

```
public static void loop() {
     final Looper me = myLooper();
     if (me == null) {
         throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
     }
     final MessageQueue queue = me.mQueue;

     // Make sure the identity of this thread is that of the local process,
     // and keep track of what that identity token actually is.
     Binder.clearCallingIdentity();
     final long ident = Binder.clearCallingIdentity();

     for (;;) {
         Message msg = queue.next(); // might block
         if (msg == null) {
             // No message indicates that the message queue is quitting.
             return;
         }

         // This must be in a local variable, in case a UI event sets the logger
         Printer logging = me.mLogging;
         if (logging != null) {
             logging.println(">>>>> Dispatching to " + msg.target + " " +
                     msg.callback + ": " + msg.what);
         }

         msg.target.dispatchMessage(msg);

         if (logging != null) {
             logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
         }

         // Make sure that during the course of dispatching the
         // identity of the thread wasn't corrupted.
```

```
35          final long newIdent = Binder.clearCallingIdentity();
36          if (ident != newIdent) {
37              Log.wtf(TAG, "Thread identity changed from 0x"
38                      + Long.toHexString(ident) + " to 0x"
39                      + Long.toHexString(newIdent) + " while
    dispatching to "
40                      + msg.target.getClass().getName() + " "
41                      + msg.callback + " what=" + msg.what);
42          }
43
44          msg.recycle();
45      }
46  }
```

第2行：

```
1  public static Looper myLooper() {
2  return sThreadLocal.get();
3  }
```

方法**直接返回了sThreadLocal存储的Looper实例**，如果me为null则抛出异常，也就是说looper方法必须在prepare方法之后运行。 第6行：拿到该looper实例中的mQueue（消息队列） 13到45行：就进入了我们所说的无限循环。 14行：取出一条消息，如果没有消息则阻塞。 27行：使用**调用 msg.target.dispatchMessage(msg);把消息交给msg的target的dispatchMessage方法去处理。Msg的target是什么呢？其实就是handler对象**，下面会进行分析。 44行：释放消息占据的资源。

Looper主要作用： 1、与当前线程绑定，保证一个线程只会有一个Looper实例，同时一个Looper实例也只有一个MessageQueue。 2、loop()方法，不断从MessageQueue中去取消息，交给消息的target属性的dispatchMessage去处理。 好了，我们的异步消息处理线程已经有了消息队列（MessageQueue），也有了在无限循环体中取出消息的哥们，现在缺的就是发送消息的对象了，于是乎：Handler登场了。

Handler

使用Handler之前，我们都是初始化一个实例，比如用于更新UI线程，我们会在声明的时候直接初始化，或者在onCreate中初始化Handler实例。所以我们首先看Handler的构造方法，看其如何与MessageQueue联系上的，它在子线程中发送的消息（一般发送消息都在非UI线程）怎么发送到MessageQueue中的。

```
1  public Handler() {
2       this(null, false);
3  }
4  public Handler(Callback callback, boolean async) {
5       if (FIND_POTENTIAL_LEAKS) {
6           final Class<? extends Handler> klass = getClass();
7           if ((klass.isAnonymousClass() || klass.isMemberClass() ||
    klass.isLocalClass()) &&
8                   (klass.getModifiers() & Modifier.STATIC) == 0) {
```

```
9               Log.w(TAG, "The following Handler class should be static
or leaks might occur: " +
10                  klass.getCanonicalName());
11          }
12      }
13
14      mLooper = Looper.myLooper();
15      if (mLooper == null) {
16          throw new RuntimeException(
17              "Can't create handler inside thread that has not called
Looper.prepare()");
18      }
19      mQueue = mLooper.mQueue;
20      mCallback = callback;
21      mAsynchronous = async;
22   }
```

14行：通过Looper.myLooper()获取了当前线程保存的Looper实例，然后在19行又获取了这个Looper实例中保存的MessageQueue（消息队列），这样就**保证了handler的实例与我们Looper实例中MessageQueue关联上了。**

然后看我们最常用的sendMessage方法

```
1  public final boolean sendMessage(Message msg){
2      return sendMessageDelayed(msg, 0);
3   }
4
```

```
1  public final boolean sendEmptyMessageDelayed(int what, long
delayMillis) {
2      Message msg = Message.obtain();
3      msg.what = what;
4      return sendMessageDelayed(msg, delayMillis);
5  }
```

```
1  public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
2      if (delayMillis < 0) {
3          delayMillis = 0;
4      }
5      return sendMessageAtTime(msg, SystemClock.uptimeMillis() +
delayMillis);
6  }
```

```
1  public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
2       MessageQueue queue = mQueue;
3       if (queue == null) {
4           RuntimeException e = new RuntimeException(
5                   this + " sendMessageAtTime() called with no
   mQueue");
6           Log.w("Looper", e.getMessage(), e);
7           return false;
8       }
9       return enqueueMessage(queue, msg, uptimeMillis);
10   }
```

辗转反则最后调用了sendMessageAtTime，在此方法内部有直接获取MessageQueue然后调用了enqueueMessage方法**，我们再来看看此方法：

```
1  private boolean enqueueMessage(MessageQueue queue, Message msg, long
   uptimeMillis) {
2       msg.target = this;
3       if (mAsynchronous) {
4           msg.setAsynchronous(true);
5       }
6       return queue.enqueueMessage(msg, uptimeMillis);
7   }
```

enqueueMessage中首先为meg.target赋值为this，如果大家还记得Looper的loop方法会取出每个msg然后交给msg,target.dispatchMessage(msg)去处理消息，也就是把当前的handler作为msg的target属性。最终会调用queue的enqueueMessage的方法，也就是说handler发出的消息，最终会保存到消息队列中去。

现在已经很清楚了Looper会调用prepare()和loop()方法，在当前执行的线程中保存一个Looper实例，这个实例会保存一个MessageQueue对象，然后当前线程进入一个无限循环中去，不断从MessageQueue中读取Handler发来的消息。然后再回调创建这个消息的handler中的dispathMessage方法，下面我们赶快去看一看这个方法：

```
1  public void dispatchMessage(Message msg) {
2       if (msg.callback != null) {
3           handleCallback(msg);
4       } else {
5           if (mCallback != null) {
6               if (mCallback.handleMessage(msg)) {
7                   return;
8               }
9           }
10          handleMessage(msg);
11      }
12   }
```

可以看到，第10行，调用了handleMessage方法，下面我们去看这个方法：

```
/**
     * Subclasses must implement this to receive messages.
 */
public void handleMessage(Message msg) {
}

```

可以看到这是一个空方法，为什么呢，因为消息的最终回调是由我们控制的，我们在创建handler的时候都是复写handleMessage方法，然后根据msg.what进行消息处理。

```
private Handler mHandler = new Handler()
    {
        public void handleMessage(android.os.Message msg)
        {
            switch (msg.what)
            {
            case value:

                break;

            default:
                break;
            }
        };
    };
```

到此，这个流程已经解释完毕，让我们首先总结一下

1、首先Looper.prepare()在本线程中保存一个Looper实例，然后该实例中保存一个MessageQueue对象；因为Looper.prepare()在一个线程中只能调用一次，所以MessageQueue在一个线程中只会存在一个。

2、Looper.loop()会让当前线程进入一个无限循环，不断从MessageQueue的实例中读取消息，然后回调msg.target.dispatchMessage(msg)方法。

3、Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue相关联。

4、Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。

5、在构造Handler实例时，我们会重写handleMessage方法，也就是msg.target.dispatchMessage(msg)最终调用的方法。

好了，总结完成，大家可能还会问，那么在Activity中，我们并没有显示的调用Looper.prepare()和Looper.loop()方法，为啥Handler可以成功创建呢，这是因为在Activity的启动代码中，已经在当前UI线程调用了Looper.prepare()和Looper.loop()方法。

Handler post

今天有人问我，你说Handler的post方法创建的线程和UI线程有什么关系？

其实这个问题也是出现这篇博客的原因之一；这里需要说明，有时候为了方便，我们会直接写如下代码：

```
mHandler.post(new Runnable()
        {
            @Override
            public void run()
            {
                Log.e("TAG", Thread.currentThread().getName());
                mTxt.setText("yoxi");
            }
        });
```

然后run方法中可以写更新UI的代码，其实这个Runnable并没有创建什么线程，而是发送了一条消息，下面看源码：

```
public final boolean post(Runnable r)
 {
    return  sendMessageDelayed(getPostMessage(r), 0);
 }

```

```
private static Message getPostMessage(Runnable r) {
     Message m = Message.obtain();
     m.callback = r;
     return m;
 }
```

可以看到，在getPostMessage中，得到了一个Message对象，然后将我们创建的Runable对象作为callback属性，赋值给了此message.

注：产生一个Message对象，可以new ，也可以使用Message.obtain()方法；两者都可以，但是更建议使用obtain方法，因为Message内部维护了一个Message池用于Message的复用，避免使用new 重新分配内存。

```
public final boolean sendMessageDelayed(Message msg, long
delayMillis)
 {
     if (delayMillis < 0) {
         delayMillis = 0;
     }
     return sendMessageAtTime(msg, SystemClock.uptimeMillis() +
delayMillis);
 }

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
     MessageQueue queue = mQueue;
```

```
11      if (queue == null) {
12          RuntimeException e = new RuntimeException(
13                  this + " sendMessageAtTime() called with no
mQueue");
14          Log.w("Looper", e.getMessage(), e);
15          return false;
16      }
17      return enqueueMessage(queue, msg, uptimeMillis);
18   }
```

最终和handler.sendMessage一样，调用了sendMessageAtTime，然后调用了enqueueMessage方法，给msg.target赋值为handler，最终加入MessagQueue.

可以看到，这里msg的callback和target都有值，那么会执行哪个呢?

其实上面已经贴过代码，就是dispatchMessage方法:

```
1  public void dispatchMessage(Message msg) {
2      if (msg.callback != null) {
3          handleCallback(msg);
4      } else {
5          if (mCallback != null) {
6              if (mCallback.handleMessage(msg)) {
7                  return;
8              }
9          }
10         handleMessage(msg);
11     }
12  }
```

好了，关于Looper , Handler , Message 这三者关系上面已经叙述的非常清楚了。

希望图片可以更好的帮助大家的记忆~~

其实Handler不仅可以更新UI，你完全可以在一个子线程中去创建一个Handler，然后使用这个handler实例在任何其他线程中发送消息，最终处理消息的代码都会在你创建Handler实例的线程中运行。

```
1  new Thread()
2          {
3              private Handler handler;
4              public void run()
5              {
6
7                  Looper.prepare();
8
9                  handler = new Handler()
```

```
                {
                    public void handleMessage(android.os.Message msg)
                    {

Log.e("TAG",Thread.currentThread().getName());
                    };
                };
                Looper.loop();

```

（7）Android之Handler用法总结 （Handler与Thread\TimerTask的结合使用思路，不用细看代码。）

方法一：(java习惯，在android平台开发时这样是不行的，因为它违背了单线程模型)

刚刚开始接触android线程编程的时候，习惯好像java一样，试图用下面的代码解决问题

```
new Thread( new Runnable() {
public void run() {
myView.invalidate();
}
}).start();
```

可以实现功能，刷新UI界面。但是这样是不行的，因为它违背了单线程模型：Android UI操作并不是线程安全的并且这些操作必须在UI线程中执行。

方法二： (Thread+Handler)

查阅了文档和apidemo后，发觉常用的方法是利用Handler来实现UI线程的更新的。

Handler来根据接收的消息，处理UI更新。Thread线程发出Handler消息，通知更新UI。

```
Handler myHandler = new Handler() {
 public void handleMessage(Message msg) {
      switch (msg.what) {
           case TestHandler.GUIUPDATEIDENTIFIER:
                myBounceView.invalidate();
                break;
      }
      super.handleMessage(msg);
 }
};
```

```
class myThread implements Runnable {
 public void run() {
      while (!Thread.currentThread().isInterrupted()) {

           Message message = new Message();
           message.what = TestHandler.GUIUPDATEIDENTIFIER;
```

```
                TestHandler.this.myHandler.sendMessage(message);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

方法三：（java习惯。Android平台中，这样做是不行的，这跟Android的线程安全有关）

在Android平台中需要反复按周期执行方法可以使用Java上自带的TimerTask类，TimerTask相对于Thread来说对于资源消耗的更低，除了使用Android自带的AlarmManager使用Timer定时器是一种更好的解决方法。 我们需要引入import java.util.Timer; 和 import java.util.TimerTask;

```
public class JavaTimer extends Activity {

Timer timer = new Timer();
TimerTask task = new TimerTask(){
public void run() {
   setTitle("hear me?");
}
};

public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);

timer.schedule(task, 10000);

}
}
```

方法四：(TimerTask + Handler)

通过配合Handler来实现timer功能的!

```
public class TestTimer extends Activity {

Timer timer = new Timer();
Handler handler = new Handler(){
public void handleMessage(Message msg) {
   switch (msg.what) {
   case 1:
       setTitle("hear me?");
       break;
   }
```

```
    super.handleMessage(msg);
}

};

TimerTask task = new TimerTask(){
public void run() {
   Message message = new Message();
   message.what = 1;
   handler.sendMessage(message);
}
};

public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);

timer.schedule(task, 10000);
}
}
```

方法五：(Runnable + Handler.postDelayed(runnable,time))

在Android里定时更新 UI，通常使用的是 *java.util.Timer*, *java.util.TimerTask*, android.*os*.Handler 组合。实际上Handler 自身已经提供了定时的功能。

```
private Handler handler = new Handler();

private Runnable myRunnable= new Runnable() {
public void run() {

   if (run) {
       handler.postDelayed(this, 1000);
       count++;
   }
   tvCounter.setText("Count: " + count);

}
};
```

知识点总结补充：

很多初入Android或Java开发的新手对Thread、Looper、Handler和Message仍然比较迷惑，衍生的有HandlerThread、java.util.concurrent、Task、AsyncTask由于目前市面上的书籍等资料都没有谈到这些问题，今天就这一问题做更系统性的总结。我们创建的Service、Activity以及Broadcast均是一个主线程处理，这里我们可以理解为**UI线程**。但是在操作一些耗时操作时，比如I/O读写的大文件读写，数据库操作以及网络下载需要很长时间，为了不阻塞用户界面，出现ANR的响应提示窗口，这个时候我们可以考虑使用Thread线程来解决。

对于从事过J2ME开发的程序员来说Thread比较简单，直接匿名创建重写run方法，调用start方法执行即可。或者从Runnable接口继承，但对于Android平台来说UI控件都没有设计成为**线程安全类型**，所以需要引入一些同步的机制来使其刷新，这点Google在设计Android时倒是参考了下Win32的消息处理机制。

1. 对于线程中的刷新一个View为基类的界面，可以**使用postInvalidate()方法**在线程中来处理，其中还提供了一些重写方法比如postInvalidate(int left,int top,int right,int bottom) 来刷新一个矩形区域，以及延时执行，比如postInvalidateDelayed(long delayMilliseconds)或postInvalidateDelayed(long delayMilliseconds,int left,int top,int right,int bottom) 方法，其中第一个参数为毫秒

2. 当然推荐的方法是通过一个**Handler来处理**这些，可以在一个线程的run方法中调用handler对象的 postMessage或sendMessage方法来实现，Android程序内部维护着一个消息队列，会轮训处理这些，如果你是Win32程序员可以很好理解这些消息处理，不过相对于Android来说没有提供PreTranslateMessage这些干涉内部的方法。

3. Looper又是什么呢? ，其实Android中每一个Thread都跟着一个Looper，Looper可以帮助Thread维护一个消息队列，但是Looper和Handler没有什么关系，我们从开源的代码可以看到Android还提供了一个Thread继承类HanderThread可以帮助我们处理，在HandlerThread对象中可以通过getLooper方法获取一个Looper对象控制句柄，我们可以将其这个Looper对象映射到一个Handler中去来实现一个线程同步机制，Looper对象的执行需要初始化Looper.prepare方法就是昨天我们看到的问题，同时推出时还要释放资源，使用Looper.release方法。

4.Message 在Android是什么呢? 对于Android中Handler可以传递一些内容，通过Bundle对象可以封装String、Integer以及Blob二进制对象，我们通过在线程中**使用Handler对象的sendEmptyMessage或sendMessage方法来传递一个Bundle对象到Handler处理器**。对于Handler类提供了重写方法**handleMessage(Message msg) 来判断，通过msg.what来区分每条信息**。将Bundle解包来实现Handler类更新UI线程中的内容实现控件的刷新操作。相关的Handler对象有关消息发送sendXXXX相关方法如下，同时还有postXXXX相关方法，这些和Win32中的道理基本一致，一个为发送后直接返回，一个为处理后才返回 .

5. java.util.concurrent对象分析，对于过去从事Java开发的程序员不会对Concurrent对象感到陌生吧，他是JDK 1.5以后新增的重要特性作为掌上设备，我们不提倡使用该类，考虑到Android为我们已经设计好的Task机制，这里不做过多的赘述，相关原因参考下面的介绍:

6. 在Android中还提供了一种有别于线程的处理方式，就是**Task以及AsyncTask**，从开源代码中可以看到是针对Concurrent的封装，开发人员可以方便的处理这些异步任务。

   Android](http://lib.csdn.net/base/15)应用程序是通过消息来驱动的，系统为每一个应用程序维护一个消息队例，应用程序的主线程不断地从这个消息队例中获取消息（Looper），然后对这些消息进行处理（Handler），这样就实现了通过消息来驱动应用程序的执行，本文将详细分析Android应用程序的消息处理机制。

前面我们学习Android应用程序中的Activity启动（Android应用程序启动过程源代码分析和Android应用程序内部启动Activity过程（startActivity）的源代码分析）、Service启动（Android系统在新进程中启动自定义服务过程（startService）的原理分析和Android应用程序绑定服务（bindService）的过程源代码分析）以及广播发送（Android应用程序发送广播（sendBroadcast）的过程分析）时，它们都有一个共同的特点，当ActivityManagerService需要与应用程序进行并互时，如加载Activity和Service、处理广播待，会通过Binder进程间通信机制来知会应用程序，应用程序接收到这个请求时，它不是马上就处理这个请求，而是将这个请求封装成一个消息，然后把这个消息放在应用程序的消息队列中去，然后再通过消息循环来处理这个消

息。这样做的好处就是消息的发送方只要把消息发送到应用程序的消息队列中去就行了，它可以马上返回去处理别的事情，而不需要等待消息的接收方去处理完这个消息才返回，这样就可以提高系统的并发性。实质上，这就是一种异步处理机制。

这样说可能还是比较笼统，我们以Android应用程序启动过程源代码分析一文中所介绍的应用程序启动过程的一个片断来具体看看是如何这种消息处理机制的。在这篇文章中，要启动的应用程序称为Activity，它的默认Activity是MainActivity，它是由Launcher来负责启动的，而Launcher又是通过ActivityManagerService来启动的，当ActivityManagerService为这个即将要启的应用程序准备好新的进程后，便通过一个Binder进程间通信过程来通知这个新的进程来加载MainActivity，如下图所示：

它对应Android应用程序启动过程中的Step 30到Step 35，有兴趣的读者可以回过头去参考Android应用程序启动过程源代码分析一文。这里的Step 30中的scheduleLaunchActivity是ActivityManagerService通过Binder进程间通信机制发送过来的请求，它请求应用程序中的ActivityThread执行Step 34中的performLaunchActivity操作，即启动MainActivity的操作。这里我们就可以看到，Step 30的这个请求并没有等待Step 34这个操作完成就返回了，它只是把这个请求封装成一个消息，然后通过Step 31中的queueOrSendMessage操作把这个消息放到应用程序的消息队列中，然后就返回了。应用程序发现消息队列中有消息时，就会通过Step 32中的handleMessage操作来处理这个消息，即调用Step 33中的handleLaunchActivity来执行实际的加载MainAcitivy类的操作。

了解Android应用程序的消息处理过程之后，我们就开始分样它的实现原理了。与Windows应用程序的消息处理过程一样，Android应用程序的消息处理机制也是由消息循环、消息发送和消息处理这三个部分组成的，接下来，我们就详细描述这三个过程。

1.消息循环

在消息处理机制中，消息都是存放在一个消息队列中去，而应用程序的主线程就是围绕这个消息队列进入一个无限循环的，直到应用程序退出。如果队列中有消息，应用程序的主线程就会把它取出来，并分发给相应的Handler进行处理；如果队列中没有消息，应用程序的主线程就会进入空闲等待状态，等待下一个消息的到来。在Android应用程序中，这个消息循环过程是由Looper类来实现的，它定义在frameworks/base/core/Java/android/os/Looper.java文件中，在分析这个类之前，我们先看一下Android应用程序主线程是如何进入到这个消息循环中去的。

在Android应用程序进程启动过程的源代码分析一文中，我们分析了Android应用程序进程的启动过程，Android应用程序进程在启动的时候，会在进程中加载ActivityThread类，并且执行这个类的main函数，应用程序的消息循环过程就是在这个main函数里面实现的，我们来看看这个函数的实现，它定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

```
1  public final class ActivityThread {
2      ......
3
4      public static final void main(String[] args) {
5          ......
6
7          Looper.prepareMainLooper();
8
9          ......
```

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

        ......

        Looper.loop();

        ......

        thread.detach();

        ......
    }
}
```

这个函数做了两件事情，一是在主线程中创建了一个ActivityThread实例，二是通过Looper类使主线程进入消息循环中，这里我们只关注后者。

首先看Looper.prepareMainLooper函数的实现，这是一个静态成员函数，定义在frameworks/base/core/java/android/os/Looper.java文件中：

```
public class Looper {
    ......

    private static final ThreadLocal sThreadLocal = new ThreadLocal();

    final MessageQueue mQueue;

    ......

    /** Initialize the current thread as a looper.
    * This gives you a chance to create handlers that then reference
    * this looper, before actually starting the loop. Be sure to call
    * {@link #loop()} after calling this method, and end it by calling
    * {@link #quit()}.
    */
    public static final void prepare() {
        if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be created per thread");
        }
        sThreadLocal.set(new Looper());
    }

    /** Initialize the current thread as a looper, marking it as an application's main
```

```
    *  looper. The main looper for your application is created by the Android environment,
    *  so you should never need to call this function yourself.
    * {@link #prepare()}
    */

    public static final void prepareMainLooper() {
        prepare();
        setMainLooper(myLooper());
        if (Process.supportsProcesses()) {
            myLooper().mQueue.mQuitAllowed = false;
        }
    }

    private synchronized static void setMainLooper(Looper looper) {
        mMainLooper = looper;
    }

    /**
    * Return the Looper object associated with the current thread. Returns
    * null if the calling thread is not associated with a Looper.
    */
    public static final Looper myLooper() {
        return (Looper)sThreadLocal.get();
    }

    private Looper() {
        mQueue = new MessageQueue();
        mRun = true;
        mThread = Thread.currentThread();
    }

    ......
}
```

函数prepareMainLooper做的事情其实就是在线程中创建一个Looper对象，这个Looper对象是存放在sThreadLocal成员变量里面的，成员变量sThreadLocal的类型为ThreadLocal，表示这是一个线程局部变量，即保证每一个调用了prepareMainLooper函数的线程里面都有一个独立的Looper对象。在线程是创建Looper对象的工作是由prepare函数来完成的，而在创建Looper对象的时候，会同时创建一个消息队列MessageQueue，保存在Looper的成员变量mQueue中，后续消息就是存放在这个队列中去。消息队列在Android应用程序消息处理机制中最重要的组件，因此，我们看看它的创建过程，即它的构造函数的实现，实现frameworks/base/core/java/android/os/MessageQueue.java文件中：

```
public class MessageQueue {
    ......

    private int mPtr; // used by native code

    private native void nativeInit();

    MessageQueue() {
        nativeInit();
    }

    ......
}
```

它的初始化工作都交给JNI方法nativeInit来实现了，这个JNI方法定义在frameworks/base/core/jni/android_os_MessageQueue.cpp文件中：

```
static void android_os_MessageQueue_nativeInit(JNIEnv* env, jobject obj) {
 NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
 if (! nativeMessageQueue) {
     jniThrowRuntimeException(env, "Unable to allocate native queue");
     return;
 }

 android_os_MessageQueue_setNativeMessageQueue(env, obj, nativeMessageQueue);
}
```

在JNI中，也相应地创建了一个消息队列NativeMessageQueue，NativeMessageQueue类也是定义在frameworks/base/core/jni/android_os_MessageQueue.cpp文件中，它的创建过程如下所示：

```
NativeMessageQueue::NativeMessageQueue() {
 mLooper = Looper::getForThread();
 if (mLooper == NULL) {
     mLooper = new Looper(false);
     Looper::setForThread(mLooper);
 }
}
```

它主要就是在内部创建了一个Looper对象，注意，这个Looper对象是实现在JNI层的，它与上面Java层中的Looper是不一样的，不过它们是对应的，下面我们进一步分析消息循环的过程的时候，读者就会清楚地了解到它们之间的关系。

这个Looper的创建过程也很重要，不过我们暂时放一放，先分析完android_os_MessageQueue_nativeInit函数的执行，它创建了本地消息队列NativeMessageQueue对象之后，接着调用android_os_MessageQueue_setNativeMessageQueue函数来把这个消息队列对象保存在前面我们在Java层中创建的MessageQueue对象的mPtr成员变量里面：

```
static void android_os_MessageQueue_setNativeMessageQueue(JNIEnv* env, jobject messageQueueObj,
        NativeMessageQueue* nativeMessageQueue) {
    env->SetIntField(messageQueueObj, gMessageQueueClassInfo.mPtr,
            reinterpret_cast<jint>(nativeMessageQueue));
}
```

这里传进来的参数messageQueueObj即为我们前面在Java层创建的消息队列对象，而gMessageQueueClassInfo.mPtr即表示在Java类MessageQueue中，其成员变量mPtr的偏移量，通过这个偏移量，就可以把这个本地消息队列对象natvieMessageQueue保存在Java层创建的消息队列对象的mPtr成员变量中，这是为了后续我们调用Java层的消息队列对象的其它成员函数进入到JNI层时，能够方便地找回它在JNI层所对应的消息队列对象。

我们再回到NativeMessageQueue的构造函数中，看看JNI层的Looper对象的创建过程，即看看它的构造函数是如何实现的，这个Looper类实现在frameworks/base/libs/utils/Looper.cpp文件中：

```
Looper::Looper(bool allowNonCallbacks) :
    mAllowNonCallbacks(allowNonCallbacks),
    mResponseIndex(0) {
    int wakeFds[2];
    int result = pipe(wakeFds);
    ......

    mWakeReadPipeFd = wakeFds[0];
    mWakeWritePipeFd = wakeFds[1];


    ......

#ifdef LOOPER_USES_EPOLL
    // Allocate the epoll instance and register the wake pipe.
    mEpollFd = epoll_create(EPOLL_SIZE_HINT);
    ......

    struct epoll_event eventItem;
    memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data field union
    eventItem.events = EPOLLIN;
    eventItem.data.fd = mWakeReadPipeFd;
    result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
    ......
```

```
#else
    ......
#endif

    ......
}
```

这个构造函数做的事情非常重要，它跟我们后面要介绍的应用程序主线程在消息队列中没有消息时要进入等待状态以及当消息队列有消息时要把应用程序主线程唤醒的这两个知识点息息相关。它主要就是通过pipe系统调用来创建了一个管道了：

```
int wakeFds[2];
int result = pipe(wakeFds);
......

mWakeReadPipeFd = wakeFds[0];
mWakeWritePipeFd = wakeFds[1];
```

管道是Linux系统中的一种进程间通信机制，具体可以参考前面一篇文章Android学习启动篇推荐的一本书《Linux内核源代码情景分析》中的第6章--传统的Uinx进程间通信。简单来说，管道就是一个文件，在管道的两端，分别是两个打开文件文件描述符，这两个打开文件描述符都是对应同一个文件，其中一个是用来读的，别一个是用来写的，一般的使用方式就是，一个线程通过读文件描述符中来读管道的内容，当管道没有内容时，这个线程就会进入等待状态，而另外一个线程通过写文件描述符来向管道中写入内容，写入内容的时候，如果另一端正有线程正在等待管道中的内容，那么这个线程就会被唤醒。这个等待和唤醒的操作是如何进行的呢，这就要借助Linux系统中的epoll机制了。 Linux系统中的epoll机制为处理大批量句柄而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著减少程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。但是这里我们其实只需要监控的IO接口只有mWakeReadPipeFd一个，即前面我们所创建的管道的读端，为什么还需要用到epoll呢？有点用牛刀来杀鸡的味道。其实不然，这个Looper类是非常强大的，它除了监控内部所创建的管道接口之外，还提供了addFd接口供外界面调用，外界可以通过这个接口把自己想要监控的IO事件一并加入到这个Looper对象中去，当所有这些被监控的IO接口上面有事件发生时，就会唤醒相应的线程来处理，不过这里我们只关心刚才所创建的管道的IO事件的发生。

要使用Linux系统的epoll机制，首先要通过epoll_create来创建一个epoll专用的文件描述符：

```
mEpollFd = epoll_create(EPOLL_SIZE_HINT);
```

1. mEpollFd = epoll_create(EPOLL_SIZE_HINT);

传入的参数EPOLL_SIZE_HINT是在这个mEpollFd上能监控的最大文件描述符数。

接着还要通过epoll_ctl函数来告诉epoll要监控相应的文件描述符的什么事件：

```
struct epoll_event eventItem;
memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data field union
eventItem.events = EPOLLIN;
eventItem.data.fd = mWakeReadPipeFd;
result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
```

这里就是告诉mEpollFd，它要监控mWakeReadPipeFd文件描述符的EPOLLIN事件，即当管道中有内容可读时，就唤醒当前正在等待管道中的内容的线程。 C++层的这个Looper对象创建好了之后，就返回到JNI层的NativeMessageQueue的构造函数，最后就返回到Java层的消息队列MessageQueue的创建过程，这样，Java层的Looper对象就准备好了。有点复杂，我们先小结一下这一步都做了些什么事情：

A. 在Java层，创建了一个Looper对象，这个Looper对象是用来进入消息循环的，它的内部有一个消息队列MessageQueue对象mQueue；

B. 在JNI层，创建了一个NativeMessageQueue对象，这个NativeMessageQueue对象保存在Java层的消息队列对象mQueue的成员变量mPtr中；

C. 在C++层，创建了一个Looper对象，保存在JNI层的NativeMessageQueue对象的成员变量mLooper中，这个对象的作用是，当Java层的消息队列中没有消息时，就使Android应用程序主线程进入等待状态，而当Java层的消息队列中来了新的消息后，就唤醒Android应用程序的主线程来处理这个消息。

回到ActivityThread类的main函数中，在上面这些工作都准备好之后，就调用Looper类的loop函数进入到消息循环中去了：

```
public class Looper {
    ......

    public static final void loop() {
        Looper me = myLooper();
        MessageQueue queue = me.mQueue;

        ......

        while (true) {
            Message msg = queue.next(); // might block
            ......

            if (msg != null) {
                if (msg.target == null) {
                    // No target is a magic identifier for the quit message.
                    return;
                }

                ......
```

```
                msg.target.dispatchMessage(msg);

                ......

                msg.recycle();
            }
        }
    }

    ......
}
```

这里就是进入到消息循环中去了，它不断地从消息队列mQueue中去获取下一个要处理的消息msg，如果消息的target成员变量为null，就表示要退出消息循环了，否则的话就要调用这个target对象的dispatchMessage成员函数来处理这个消息，这个target对象的类型为Handler，下面我们分析消息的发送时会看到这个消息对象msg是如设置的。

这个函数最关键的地方便是从消息队列中获取下一个要处理的消息了，即MessageQueue.next函数，它实现frameworks/base/core/java/android/os/MessageQueue.java文件中：

```
public class MessageQueue {
    ......

    final Message next() {
        int pendingIdleHandlerCount = -1; // -1 only during first iteration
        int nextPollTimeoutMillis = 0;

        for (;;) {
            if (nextPollTimeoutMillis != 0) {
                Binder.flushPendingCommands();
            }
            nativePollOnce(mPtr, nextPollTimeoutMillis);

            synchronized (this) {
                // Try to retrieve the next message.  Return if found.
                final long now = SystemClock.uptimeMillis();
                final Message msg = mMessages;
                if (msg != null) {
                    final long when = msg.when;
                    if (now >= when) {
                        mBlocked = false;
                        mMessages = msg.next;
                        msg.next = null;
                        if (Config.LOGV) Log.v("MessageQueue", "Returning message: " + msg);
                        return msg;
```

```
                    } else {
                        nextPollTimeoutMillis = (int) Math.min(when -
now, Integer.MAX_VALUE);
                    }
                } else {
                    nextPollTimeoutMillis = -1;
                }

                // If first time, then get the number of idlers to
run.
                if (pendingIdleHandlerCount < 0) {
                    pendingIdleHandlerCount = mIdleHandlers.size();
                }
                if (pendingIdleHandlerCount == 0) {
                    // No idle handlers to run.  Loop and wait some
more.
                    mBlocked = true;
                    continue;
                }

                if (mPendingIdleHandlers == null) {
                    mPendingIdleHandlers = new
IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
                }
                mPendingIdleHandlers =
mIdleHandlers.toArray(mPendingIdleHandlers);
            }

            // Run the idle handlers.
            // We only ever reach this code block during the first
iteration.
            for (int i = 0; i < pendingIdleHandlerCount; i++) {
                final IdleHandler idler = mPendingIdleHandlers[i];
                mPendingIdleHandlers[i] = null; // release the
reference to the handler

                boolean keep = false;
                try {
                    keep = idler.queueIdle();
                } catch (Throwable t) {
                    Log.wtf("MessageQueue", "IdleHandler threw
exception", t);
                }

                if (!keep) {
                    synchronized (this) {
                        mIdleHandlers.remove(idler);
                    }
                }
```

```
            }

            // Reset the idle handler count to 0 so we do not run
them again.
            pendingIdleHandlerCount = 0;

            // While calling an idle handler, a new message could
have been delivered
            // so go back and look again for a pending message
without waiting.
            nextPollTimeoutMillis = 0;
        }
    }

    ......
}
```

调用这个函数的时候，有可能会让线程进入等待状态。什么情况下，线程会进入等待状态呢？两种情况，一是当消息队列中没有消息时，它会使线程进入等待状态；二是消息队列中有消息，但是消息指定了执行的时间，而现在还没有到这个时间，线程也会进入等待状态。消息队列中的消息是按时间先后来排序的，后面我们在分析消息的发送时会看到。

执行下面语句是看看当前消息队列中有没有消息：

```
nativePollOnce(mPtr, nextPollTimeoutMillis);
```

1. nativePollOnce(mPtr, nextPollTimeoutMillis);

这是一个JNI方法，我们等一下再分析，这里传入的参数mPtr就是指向前面我们在JNI层创建的NativeMessageQueue对象了，而参数nextPollTimeoutMillis则表示如果当前消息队列中没有消息，它要等待的时候，for循环开始时，传入的值为0，表示不等待。

当前nativePollOnce返回后，就去看看消息队列中有没有消息：

```
final Message msg = mMessages;
if (msg != null) {
    final long when = msg.when;
    if (now >= when) {
        mBlocked = false;
        mMessages = msg.next;
        msg.next = null;
        if (Config.LOGV) Log.v("MessageQueue", "Returning message: "
+ msg);
        return msg;
    } else {
        nextPollTimeoutMillis = (int) Math.min(when - now,
Integer.MAX_VALUE);
    }
} else {
```

```
    nextPollTimeoutMillis = -1;
}
```

如果消息队列中有消息，并且当前时候大于等于消息中的执行时间，那么就直接返回这个消息给Looper.loop消息处理，否则的话就要等待到消息的执行时间：

```
nextPollTimeoutMillis = (int) Math.min(when - now, Integer.MAX_VALUE);
```

如果消息队列中没有消息，那就要进入无穷等待状态直到有新消息了：

```
nextPollTimeoutMillis = -1;
```

-1表示下次调用nativePollOnce时，如果消息中没有消息，就进入无限等待状态中去。

这里计算出来的等待时间都是在下次调用nativePollOnce时使用的。

这里说的等待，是空闲等待，而不是忙等待，因此，在进入空闲等待状态前，如果应用程序注册了IdleHandler接口来处理一些事情，那么就会先执行这里IdleHandler，然后再进入等待状态。IdlerHandler是定义在MessageQueue的一个内部类：

```
public class MessageQueue {
    ......

    /**
    * Callback interface for discovering when a thread is going to block
    * waiting for more messages.
    */
    public static interface IdleHandler {
        /**
        * Called when the message queue has run out of messages and will now
        * wait for more.  Return true to keep your idle handler active, false
        * to have it removed.  This may be called if there are still messages
        * pending in the queue, but they are all scheduled to be dispatched
        * after the current time.
        */
        boolean queueIdle();
    }

    ......
}
```

它只有一个成员函数queueIdle，执行这个函数时，如果返回值为false，那么就会从应用程序中移除这个IdleHandler，否则的话就会在应用程序中继续维护着这个IdleHandler，下次空闲时仍会再执会这个IdleHandler。MessageQueue提供了addIdleHandler和removeIdleHandler两注册和删除IdleHandler。

回到MessageQueue函数中，它接下来就是在进入等待状态前，看看有没有IdleHandler是需要执行的：

```
// If first time, then get the number of idlers to run.
if (pendingIdleHandlerCount < 0) {
    pendingIdleHandlerCount = mIdleHandlers.size();
}
if (pendingIdleHandlerCount == 0) {
    // No idle handlers to run.  Loop and wait some more.
    mBlocked = true;
    continue;
}

if (mPendingIdleHandlers == null) {
    mPendingIdleHandlers = new
IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
}
mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
```

如果没有，即pendingIdleHandlerCount等于0，那下面的逻辑就不执行了，通过continue语句直接进入下一次循环，否则就要把注册在mIdleHandlers中的IdleHandler取出来，放在mPendingIdleHandlers数组中去。

接下来就是执行这些注册了的IdleHanlder了：

```
// Run the idle handlers.
// We only ever reach this code block during the first iteration.
for (int i = 0; i < pendingIdleHandlerCount; i++) {
   final IdleHandler idler = mPendingIdleHandlers[i];
   mPendingIdleHandlers[i] = null; // release the reference to the
handler

   boolean keep = false;
   try {
         keep = idler.queueIdle();
   } catch (Throwable t) {
         Log.wtf("MessageQueue", "IdleHandler threw exception", t);
   }

   if (!keep) {
         synchronized (this) {
                mIdleHandlers.remove(idler);
         }
   }
```

```
19  }
```

执行完这些IdleHandler之后，线程下次调用nativePollOnce函数时，就不设置超时时间了，因为，很有可能在执行IdleHandler的时候，已经有新的消息加入到消息队列中去了，因此，要重nextPollTimeoutMillis的值：

```
1  // While calling an idle handler, a new message could have been
   delivered
2  // so go back and look again for a pending message without waiting.
3  nextPollTimeoutMillis = 0;
```

分析完MessageQueue的这个next函数之后，我们就要深入分析一下JNI方法nativePollOnce了，看看它是如何进入等待状态的，这个函数定义在frameworks/base/core/jni/android_os_MessageQueue.cpp文件中：

```
1  static void android_os_MessageQueue_nativePollOnce(JNIEnv* env,
   jobject obj,
2       jint ptr, jint timeoutMillis) {
3   NativeMessageQueue* nativeMessageQueue =
   reinterpret_cast<NativeMessageQueue*>(ptr);
4   nativeMessageQueue->pollOnce(timeoutMillis);
5  }
```

这个函数首先是通过传进入的参数ptr取回前面在Java层创建MessageQueue对象时在JNI层创建的NatvieMessageQueue对象，然后调用它的pollOnce函数：

```
1  void NativeMessageQueue::pollOnce(int timeoutMillis) {
2   mLooper->pollOnce(timeoutMillis);
3  }
```

这里将操作转发给mLooper对象的pollOnce函数处理，这里的mLooper对象是在C++层的对象，它也是在前面在JNI层创建的NatvieMessageQueue对象时创建的，它的pollOnce函数定义在frameworks/base/libs/utils/Looper.cpp文件中：

```
1  int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents,
   void** outData) {
2      int result = 0;
3      for (;;) {
4          ......
5
6          if (result != 0) {
7              ......
8
9              return result;
10         }
11
12         result = pollInner(timeoutMillis);
```

```
    }
}
```

为了方便讨论，我们把这个函数的无关部分都去掉，它主要就是调用pollInner函数来进一步操作，如果pollInner返回值不等于0，这个函数就可以返回了。

函数pollInner的定义如下：

```
int Looper::pollInner(int timeoutMillis) {
    ......

    int result = ALOOPER_POLL_WAKE;

    ......

#ifdef LOOPER_USES_EPOLL
    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);
    bool acquiredLock = false;
#else
    ......
#endif

    if (eventCount < 0) {
        if (errno == EINTR) {
            goto Done;
        }

        LOGW("Poll failed with an unexpected error, errno=%d", errno);
        result = ALOOPER_POLL_ERROR;
        goto Done;
    }

    if (eventCount == 0) {
        ......
        result = ALOOPER_POLL_TIMEOUT;
        goto Done;
    }

    ......

#ifdef LOOPER_USES_EPOLL
    for (int i = 0; i < eventCount; i++) {
        int fd = eventItems[i].data.fd;
        uint32_t epollEvents = eventItems[i].events;
        if (fd == mWakeReadPipeFd) {
            if (epollEvents & EPOLLIN) {
```

```
                awoken();
            } else {
                LOGW("Ignoring unexpected epoll events 0x%x on wake
read pipe.", epollEvents);
            }
        } else {
            ......
        }
    }
    if (acquiredLock) {
        mLock.unlock();
    }
Done: ;
#else
    ......
#endif

    ......

    return result;
}
```

这里，首先是调用epoll_wait函数来看看epoll专用文件描述符mEpollFd所监控的文件描述符是否有IO事件发生，它设置监控的超时时间为timeoutMillis：

```
int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS,
timeoutMillis);
```

回忆一下前面的Looper的构造函数，我们在里面设置了要监控mWakeReadPipeFd文件描述符的EPOLLIN事件。

当mEpollFd所监控的文件描述符发生了要监控的IO事件后或者监控时间超时后，线程就从epoll_wait返回了，否则线程就会在epoll_wait函数中进入睡眠状态了。返回后如果eventCount等于0，就说明是超时了：

```
if (eventCount == 0) {
 ......
 result = ALOOPER_POLL_TIMEOUT;
 goto Done;
}
```

如果eventCount不等于0，就说明发生要监控的事件：

```
for (int i = 0; i < eventCount; i++) {
    int fd = eventItems[i].data.fd;
    uint32_t epollEvents = eventItems[i].events;
    if (fd == mWakeReadPipeFd) {
        if (epollEvents & EPOLLIN) {
            awoken();
        } else {
            LOGW("Ignoring unexpected epoll events 0x%x on wake read pipe.", epollEvents);
        }
    } else {
            ......
    }
}
```

这里我们只关注mWakeReadPipeFd文件描述符上的事件，如果在mWakeReadPipeFd文件描述符上发生了EPOLLIN就说明应用程序中的消息队列里面有新的消息需要处理了，接下来它就会先调用awoken函数清空管道中的内容，以便下次再调用pollInner函数时，知道自从上次处理完消息队列中的消息后，有没有新的消息加进来。

函数awoken的实现很简单，它只是把管道中的内容都读取出来：

```
void Looper::awoken() {
    ......

    char buffer[16];
    ssize_t nRead;
    do {
        nRead = read(mWakeReadPipeFd, buffer, sizeof(buffer));
    } while ((nRead == -1 && errno == EINTR) || nRead == sizeof(buffer));
}
```

因为当其它的线程向应用程序的消息队列加入新的消息时，会向这个管道写入新的内容来通知应用程序主线程有新的消息需要处理了，下面我们分析消息的发送的时候将会看到。

这样，消息的循环过程就分析完了，这部分逻辑还是比较复杂的，它利用Linux系统中的管道（pipe）进程间通信机制来实现消息的等待和处理，不过，了解了这部分内容之后，下面我们分析消息的发送和处理就简单多了。

```
    2. 消息的发送
```

应用程序的主线程准备就好消息队列并且进入到消息循环后，其它地方就可以往这个消息队列中发送消息了。我们继续以文章开始介绍的Android应用程序启动过程源代码分析一文中的应用程序启动过为例，说明应用程序是如何把消息加入到应用程序的消息队列中去的。

在Android应用程序启动过程源代码分析这篇文章的Step 30中，ActivityManagerService通过调用ApplicationThread类的scheduleLaunchActivity函数通知应用程序，它可以加载应用程序的默认Activity了，这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

```
public final class ActivityThread {

 ......

 private final class ApplicationThread extends
ApplicationThreadNative {

     ......

     // we use token to identify this activity without having to send
the
     // activity itself back to the activity manager. (matters more
with ipc)
     public final void scheduleLaunchActivity(Intent intent, IBinder
token, int ident,
             ActivityInfo info, Bundle state, List<ResultInfo>
pendingResults,
             List<Intent> pendingNewIntents, boolean notResumed,
boolean isForward) {
         ActivityClientRecord r = new ActivityClientRecord();

         r.token = token;
         r.ident = ident;
         r.intent = intent;
         r.activityInfo = info;
         r.state = state;

         r.pendingResults = pendingResults;
         r.pendingIntents = pendingNewIntents;

         r.startsNotResumed = notResumed;
         r.isForward = isForward;

         queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
     }

     ......

 }

 ......
}
```

这里把相关的参数都封装成一个ActivityClientRecord对象r，然后调用queueOrSendMessage函数来往应用程序的消息队列中加入一个新的消息（H.LAUNCH_ACTIVITY），这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

```
public final class ActivityThread {

 ......

 private final class ApplicationThread extends
ApplicationThreadNative {

     ......

     // if the thread hasn't started yet, we don't have the handler,
so just
     // save the messages until we're ready.
     private final void queueOrSendMessage(int what, Object obj) {
         queueOrSendMessage(what, obj, 0, 0);
     }

     ......

     private final void queueOrSendMessage(int what, Object obj, int
arg1, int arg2) {
         synchronized (this) {
             ......
             Message msg = Message.obtain();
             msg.what = what;
             msg.obj = obj;
             msg.arg1 = arg1;
             msg.arg2 = arg2;
             mH.sendMessage(msg);
         }
     }

     ......

 }

 ......
}
```

在queueOrSendMessage函数中，又进一步把上面传进来的参数封装成一个Message对象msg，然后通过mH.sendMessage函数把这个消息对象msg加入到应用程序的消息队列中去。这里的mH是ActivityThread类的成员变量，它的类型为H，继承于Handler类，它定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

```
public final class ActivityThread {
```

```
    ......

    private final class H extends Handler {

        ......

        public void handleMessage(Message msg) {
            ......
            switch (msg.what) {
            ......
            }

        ......

    }

    ......
}
```

这个H类就是通过其成员函数handleMessage函数来处理消息的了，后面我们分析消息的处理过程时会看到。 ActivityThread类的这个mH成员变量是什么时候创建的呢？我们前面在分析应用程序的消息循环时，说到当应用程序进程启动之后，就会加载ActivityThread类的main函数里面，在这个main函数里面，在通过Looper类进入消息循环之前，会在当前进程中创建一个ActivityThread实例：

```
public final class ActivityThread {
    ......

    public static final void main(String[] args) {
        ......

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

        ......
    }
}
```

在创建这个实例的时候，就会同时创建其成员变量mH了：

```
public final class ActivityThread {
    ......

    final H mH = new H();

    ......
}
```

前面说过，H类继承于Handler类，因此，当创建这个H对象时，会调用Handler类的构造函数，这个函数定义在frameworks/base/core/java/android/os/Handler.java文件中：

```
public class Handler {
    ......

    public Handler() {
        ......

        mLooper = Looper.myLooper();
        ......

        mQueue = mLooper.mQueue;
        ......
    }


    final MessageQueue mQueue;
    final Looper mLooper;
    ......
}
```

在Hanlder类的构造函数中，主要就是初始成员变量mLooper和mQueue了。这里的myLooper是Looper类的静态成员函数，通过它来获得一个Looper对象，这个Looper对象就是前面我们在分析消息循环时，在ActivityThread类的main函数中通过Looper.prepareMainLooper函数创建的。Looper.myLooper函数实现在frameworks/base/core/java/android/os/Looper.java文件中：

```
public class Looper {
    ......

    public static final Looper myLooper() {
        return (Looper)sThreadLocal.get();
    }

    ......
}
```

有了这个Looper对象后，就可以通过Looper.mQueue来访问应用程序的消息队列了。

有了这个Handler对象mH后，就可以通过它来往应用程序的消息队列中加入新的消息了。回到前面的queueOrSendMessage函数中，当它准备好了一个Message对象msg后，就开始调用mH.sendMessage函数来发送消息了，这个函数定义在frameworks/base/core/java/android/os/Handler.java文件中：

```
public class Handler {
    ......

```

```
    public final boolean sendMessage(Message msg)
    {
        return sendMessageDelayed(msg, 0);
    }

    public final boolean sendMessageDelayed(Message msg, long
delayMillis)
    {
        if (delayMillis < 0) {
            delayMillis = 0;
        }
        return sendMessageAtTime(msg, SystemClock.uptimeMillis() +
delayMillis);
    }

    public boolean sendMessageAtTime(Message msg, long uptimeMillis)
    {
        boolean sent = false;
        MessageQueue queue = mQueue;
        if (queue != null) {
            msg.target = this;
            sent = queue.enqueueMessage(msg, uptimeMillis);
        }
        else {
            ......
        }
        return sent;
    }

    ......
}
```

在发送消息时，是可以指定消息的处理时间的，但是通过sendMessage函数发送的消息的处理时间默认就为当前时间，即表示要马上处理，因此，从sendMessage函数中调用sendMessageDelayed函数，传入的时间参数为0，表示这个消息不要延时处理，而在sendMessageDelayed函数中，则会先获得当前时间，然后加上消息要延时处理的时间，即得到这个处理这个消息的绝对时间，然后调用sendMessageAtTime函数来把消息加入到应用程序的消息队列中去。

在sendMessageAtTime函数，首先得到应用程序的消息队列mQueue，这是在Handler对象构造时初始化好的，前面已经分析过了，接着设置这个消息的目标对象target，即这个消息最终是由谁来处理的：

```
msg.target = this;
```

这里将它赋值为this，即表示这个消息最终由这个Handler对象来处理，即由ActivityThread对象的mH成员变量来处理。

函数最后调用queue.enqueueMessage来把这个消息加入到应用程序的消息队列中去，这个函数实现在frameworks/base/core/java/android/os/MessageQueue.java文件中：

```
public class MessageQueue {
    ......

    final boolean enqueueMessage(Message msg, long when) {
        ......

        final boolean needWake;
        synchronized (this) {
            ......

            msg.when = when;
            //Log.d("MessageQueue", "Enqueing: " + msg);
            Message p = mMessages;
            if (p == null || when == 0 || when < p.when) {
                msg.next = p;
                mMessages = msg;
                needWake = mBlocked; // new head, might need to wake up
            } else {
                Message prev = null;
                while (p != null && p.when <= when) {
                    prev = p;
                    p = p.next;
                }
                msg.next = prev.next;
                prev.next = msg;
                needWake = false; // still waiting on head, no need to wake up
            }

        }
        if (needWake) {
            nativeWake(mPtr);
        }
        return true;
    }

    ......
}
```

把消息加入到消息队列时，分两种情况，一种当前消息队列为空时，这时候应用程序的主线程一般就是处于空闲等待状态了，这时候就要唤醒它，另一种情况是应用程序的消息队列不为空，这时候就不需要唤醒应用程序的主线程了，因为这时候它一定是在忙着处于消息队列中的消息，因此不会处于空闲等待的状态。

第一种情况比较简单，只要把消息放在消息队列头就可以了：

```
msg.next = p;
mMessages = msg;
needWake = mBlocked; // new head, might need to wake up
```

第二种情况相对就比较复杂一些了，前面我们说过，当往消息队列中发送消息时，是可以指定消息的处理时间的，而消息队列中的消息，就是按照这个时间从小到大来排序的，因此，当把新的消息加入到消息队列时，就要根据它的处理时间来找到合适的位置，然后再放进消息队列中去：

```
Message prev = null;
while (p != null && p.when <= when) {
    prev = p;
    p = p.next;
}
msg.next = prev.next;
prev.next = msg;
needWake = false; // still waiting on head, no need to wake up
```

把消息加入到消息队列去后，如果应用程序的主线程正处于空闲等待状态，就需要调用natvieWake函数来唤醒它了，这是一个JNI方法，定义在frameworks/base/core/jni/android_os_MessageQueue.cpp文件中：

```
static void android_os_MessageQueue_nativeWake(JNIEnv* env, jobject obj, jint ptr) {
 NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
 return nativeMessageQueue->wake();
}
```

这个JNI层的NativeMessageQueue对象我们在前面分析消息循环的时候创建好的，保存在Java层的MessageQueue对象的mPtr成员变量中，这里把它取回来之后，就调用它的wake函数来唤醒应用程序的主线程，这个函数也是定义在frameworks/base/core/jni/android_os_MessageQueue.cpp文件中：

```
void NativeMessageQueue::wake() {
 mLooper->wake();
}
```

这里它又通过成员变量mLooper的wake函数来执行操作，这里的mLooper成员变量是一个C++层实现的Looper对象，它定义在frameworks/base/libs/utils/Looper.cpp文件中：

```
void Looper::wake() {
    ......

    ssize_t nWrite;
    do {
        nWrite = write(mWakeWritePipeFd, "W", 1);
    } while (nWrite == -1 && errno == EINTR);

    .......
}
```

这个wake函数很简单，只是通过打开文件描述符mWakeWritePipeFd往管道的写入一个"W"字符串。其实，往管道写入什么内容并不重要，往管道写入内容的目的是为了唤醒应用程序的主线程。前面我们在分析应用程序的消息循环时说到，当应用程序的消息队列中没有消息处理时，应用程序的主线程就会进入空闲等待状态，而这个空闲等待状态就是通过调用这个Looper类的pollInner函数来进入的，具体就是在pollInner函数中调用epoll_wait函数来等待管道中有内容可读的。

这时候既然管道中有内容可读了，应用程序的主线程就会从这里的Looper类的pollInner函数返回到JNI层的nativePollOnce函数，最后返回到Java层中的MessageQueue.next函数中去，这里它就会发现消息队列中有新的消息需要处理了，于就会处理这个消息。

3. 消息的处理

前面在分析消息循环时，说到应用程序的主线程是在Looper类的loop成员函数中进行消息循环过程的，这个函数定义在frameworks/base/core/java/android/os/Looper.java文件中：

```
public class Looper {
    ......

    public static final void loop() {
        Looper me = myLooper();
        MessageQueue queue = me.mQueue;

        ......

        while (true) {
            Message msg = queue.next(); // might block
            ......

            if (msg != null) {
                if (msg.target == null) {
                    // No target is a magic identifier for the quit
message.
                    return;
                }

                ......

```

```
                msg.target.dispatchMessage(msg);

                ......

                msg.recycle();
            }
        }
    }

    ......
}
```

它从消息队列中获得消息对象msg后，就会调用它的target成员变量的dispatchMessage函数来处理这个消息。在前面分析消息的发送时说过，这个消息对象msg的成员变量target是在发送消息的时候设置好的，一般就通过哪个Handler来发送消息，就通过哪个Handler来处理消息。

我们继续以前面分析消息的发送时所举的例子来分析消息的处理过程。前面说到，在Android应用程序启动过程源代码分析这篇文章的Step 30中，ActivityManagerService通过调用ApplicationThread类的scheduleLaunchActivity函数通知应用程序，它可以加载应用程序的默认Activity了，而ApplicationThread类的scheduleLaunchActivity函数最终把这个请求封装成一个消息，然后通过ActivityThread类的成员变量mH来把这个消息加入到应用程序的消息队列中去。现在要对这个消息进行处理了，于是就会调用H类的dispatchMessage函数进行处理。

H类没有实现自己的dispatchMessage函数，但是它继承了父类Handler的dispatchMessage函数，这个函数定义在frameworks/base/core/java/android/os/ Handler.java文件中：

```
public class Handler {
    ......

    public void dispatchMessage(Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            handleMessage(msg);
        }
    }

    ......
}
```

这里的消息对象msg的callback成员变量和Handler类的mCallBack成员变量一般都为null，于是，就会调用Handler类的handleMessage函数来处理这个消息，由于H类在继承Handler类时，重写了handleMessage函数，因此，这里调用的实际上是H类的handleMessage函数，这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

```
public final class ActivityThread {

 ......

 private final class H extends Handler {

     ......

     public void handleMessage(Message msg) {
         ......
         switch (msg.what) {
         case LAUNCH_ACTIVITY: {
             ActivityClientRecord r = (ActivityClientRecord)msg.obj;


             r.packageInfo = getPackageInfoNoCheck(
                 r.activityInfo.applicationInfo);
             handleLaunchActivity(r, null);
         } break;
         ......
         }

     ......

 }

 ......
}
```

因为前面在分析消息的发送时所举的例子中，发送的消息的类型为H.LAUNCH_ACTIVITY，因此，这里就会调用ActivityThread类的handleLaunchActivity函数来真正地处理这个消息了，后面的具体过程就可以参考Android应用程序启动过程源代码分析这篇文章了。

至此，我们就从消息循环、消息发送和消息处理三个部分分析完Android应用程序的消息处理机制了，为了更深理解，这里我们对其中的一些要点作一个总结：

A. Android应用程序的消息处理机制由消息循环、消息发送和消息处理三个部分组成的。

B. Android应用程序的主线程在进入消息循环过程前，会在内部创建一个Linux管道（Pipe），这个管道的作用是使得Android应用程序主线程在消息队列为空时可以进入空闲等待状态，并且使得当应用程序的消息队列有消息需要处理时唤醒应用程序的主线程。

C. Android应用程序的主线程进入空闲等待状态的方式实际上就是在管道的读端等待管道中有新的内容可读，具体来说就是是通过Linux系统的Epoll机制中的epoll_wait函数进行的。

D. 当往Android应用程序的消息队列中加入新的消息时，会同时往管道中的写端写入内容，通过这种方式就可以唤醒正在等待消息到来的应用程序主线程。

E. 当应用程序主线程在进入空闲等待前，会认为当前线程处理空闲状态，于是就会调用那些已经注册了的IdleHandler接口，使得应用程序有机会在空闲的时候处理一些事情。

8.ListView图片加载错乱的原理和解决方案

ListView item缓存机制：为了使得性能更优，ListView会缓存行item(某行对应的View)。ListView通过adapter的getView函数获得每行的item。

滑动过程中

1）如果某行item已经滑出屏幕，若该item不在缓存内，则put进缓存，否则更新缓存；

2）获取滑入屏幕的行item之前会先判断缓存中是否有可用的item，如果有，做为convertView参数传递给adapter的getView。

出现的问题：

1）行item图片显示重复，当前行item显示了之前某行item的图片。

比如ListView滑动到第2行会异步加载某个图片，但是加载很慢，加载过程中listView已经滑动到了第14行，且滑动过程中该图片加载结束，第2行已不在屏幕内，根据上面介绍的缓存原理，第2行的view可能被第14行复用，这样我们看到的就是第14行显示了本该属于第2行的图片，造成显示重复。

2）行item图片显示闪烁

如果第14行图片又很快加载结束，所以我们看到第14行先显示了第2行的图片，立马又显示了自己的图片进行覆盖造成闪烁错乱。

解决方法

通过上面的分析我们知道了出现错乱的原因是异步加载及对象被复用造成的，如果每次getView能给对象一个标识，在异步加载完成时比较标识与当前行item的标识是否一致，一致则显示，否则不做处理即可。

9.数据库的操作类型有哪些，如何导入外部数据库?

使用数据库的方式有哪些?

（1）`openOrCreateDatabase(String path);`

（2）继承 `SqliteOpenHelper` 类对数据库及其版本进行管理(`onCreate,onUpgrade`)

当在程序当中调用这个类的方法 `getWritableDatabase()` 或者 `getReadableDatabase()` 的时候才会打开数据库。如果当时没有数据库文件的时候，系统就会自动生成一个数据库。

1） 操作的类型：增删改查CRUD

直接操作SQL语句：`SQliteDatabase.execSQL(sql);`

面向对象的操作方式：`SQLiteDatabase.insert(table, nullColumnHack, ContentValues);`

如何导入外部数据库?

一般外部数据库文件可能放在SD卡或者res/raw或者assets目录下面。

写一个DBManager的类来管理，数据库文件搬家，先把数据库文件复制到”/data/data/包名/databases/”目录下面，然后通过db.openOrCreateDatabase(db文件),打开数据库使用(通过流的方式读取和写入)。

Android关于数据库并发操作的问题和解决办法

`SQLiteDatabaseLockedException: database is locked` 和 `java.lang.IllegalStateException: attempt to re-open an already-closed object.` 这两个是我们最常见的数据库并发异常，SQLite是文件级别的锁.SQLite3对于并发的处理机制是允许同一个进程的多个线程同时读取一个数据库，但是任何时刻只允许一个线程/进程写入数据库。在操行写操作时，数据库文件被琐定，此时任何其他读/写操作都被阻塞，如果阻塞超过5秒钟(默认是5秒，能过重新编译sqlite可以修改超时时间)，就报"database is locked"错误。我们在操作数据库是应该是一个数据库对应一个SQliteOpenHelper对象，因为每个数据库的操作方法不同。helper又会对应一个manager用于管理该数据库的增删改查。那么helper对象的获取应该是写到其对应的manager类中的。将helper的对象单例化后，确保使用不同线程使用的是同一个数据库连接。

场景为：线程A打开数据，正在使用数据库，这时cpu片段分到线程B，线程A挂起。线程B进入执行获取打开db时没有问题，线程B进行操作，在片段时间内数据操作完成，最后关闭数据库database.close()。线程B执行结束，线程A执行，插入数据或者其他操作，怎么数据库关闭了呢，然后抛出java.lang.IllegalStateException: attempt to re-open an already-closed object异常。

解决方法：

1.所有的数据库都加同步锁，影响性能；

2.引入一个用于计算的静态变量mCount，数据库打开 ++，数据库关闭 — ，知道mCount为0时，关闭数据库。

10.是否使用过本地广播，和全局广播有什么差别?

引入本地广播的机制是为了解决安全性的问题：

1）正在发送的广播不会脱离应用程序，比如担心app的数据泄露；

2）其他的程序无法发送到我的应用程序内部，不担心安全漏洞。（比如：如何做一个杀不死的服务---监听火爆的app 比如微信、友盟、极光的广播，来启动自己。）

3） 发送本地广播比发送全局的广播高效。（全局广播要维护的广播集合表效率更低。全局广播，意味着可以跨进程，就需要底层的支持。）

本地广播不能用静态注册。----静态注册：可以做到程序停止后还能监听。

使用：

（1）注册

```
LocalBroadcastManager.getInstance(this).registerReceiver(new XXXBroadCastReceiver(), new IntentFilter(action));
```

（2）取消注册：

```
LocalBroadcastManager.getInstance(this).unregisterReceiver(receiver)
```

1、本地广播：发送的广播事件不被其他应用程序获取，也不能响应其他应用程序发送的广播事件。本地广播只能被动态注册，不能静态注册。动态注册或方法时需要用到 LocalBroadcastManager。

2、全局广播：发送的广播事件可被其他应用程序获取，也能响应其他应用程序发送的广播事件（可以通过 exported–是否监听其他应用程序发送的广播 在清单文件中控制） 全局广播既可以动态注册，也可以静态注册。

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

如果有一个任务，可以分成很多个子任务，需要按照顺序来完成，如果需要放到一个服务中完成，那么使用IntentService是最好的选择。

一般我们所使用的Service是运行在主线程当中的，所以在service里面编写耗时的操作代码，则会阻塞主线程造成ANR。为了解决这样的问题，谷歌引入了IntentService.

IntentService的优点：

（1）它创建一个独立的工作线程来处理所有一个一个intent。

（2）创建了一个工作队列，来逐个发送intent给onHandleIntent()

（3）不需要主动调用stopSelf()来结束服务，因为源码里面自己实现了自动关闭。

（4）默认实现了onBind()返回的null。

（5）默认实现的onStartCommand()的目的是将intent插入到工作队列。

总结：使用IntentService的好处有哪些：

首先，省去了手动开线程的麻烦；

第二，不用手动停止service；

第三，由于设计了工作队列，可以启动多次---startService(),但是只有一个service实例和一个工作线程。一个一个顺序执行。

AIDL 解决了什么问题?

AIDL的全称：Android Interface Definition Language，安卓接口定义语言。

由于Android系统中的进程之间不能共享内存，所以需要提供一些机制在不同的进程之间进行数据通信。

远程过程调用：RPC—Remote Procedure Call。 安卓就是提供了一种IDL的解决方案来公开自己的服务接口。AIDL:可以理解为双方的一个协议合同。双方都要持有这份协议---文本协议 xxx.aidl文件（安卓内部编译的时候会将aidl协议翻译生成一个xxx.java文件---代理模式：Binder驱动有关的，Linux底层通讯有关的。）

在系统源码里面有大量用到aidl，比如系统服务。

电视机顶盒系统开发。你的服务要暴露给别的开发者来使用。

12.Binder机制

Binder是Android系统进程间通信（IPC）方式之一。Linux已经拥有的进程间通信IPC手段包括(Internet Process Connection)： 管道（Pipe）、信号（Signal）和跟踪（Trace）、插口（Socket）、报文队列（Message）、共享内存（Share Memory）和信号量（Semaphore）。本文详细介绍Binder作为Android主要IPC方式的优势。

一、引言

基于Client-Server的通信方式广泛应用于从互联网和数据库访问到嵌入式手持设备内部通信等各个领域。智能手机平台特别是Android系统中，为了向应用开发者提供丰富多样的功能，这种通信方式更是无处不在，诸如媒体播放，视音频频捕获，到各种让手机更智能的传感器（加速度，方位，温度，光亮度等）都由不同的Server负责管理，应用程序只需做为Client与这些Server建立连接便可以使用这些服务，花很少的时间和精力就能开发出令人眩目的功能。Client-Server方式的广泛采用对进程间通信（IPC）机制是一个挑战。目前linux支持的IPC包括传统的管道，System V IPC，即消息队列/共享内存/信号量，以及socket中只有socket支持Client-Server的通信方式。当然也可以在这些底层机制上架设一套协议来实现Client-Server通信，但这样增加了系统的复杂性，在手机这种条件复杂，资源稀缺的环境下可靠性也难以保证。

另一方面是传输性能。socket作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难以使用。

表 1 各种IPC方式数据拷贝次数

IPC	**数据拷贝次数**
共享内存	0
Binder	1
Socket/管道/消息队列	2

还有一点是出于安全性考虑。终端用户不希望从网上下载的程序在不知情的情况下偷窥隐私数据，连接无线网络，长期操作底层设备导致电池很快耗尽等等。传统IPC没有任何安全措施，完全依赖上层协议来确保。首先传统IPC的接收方无法获得对方进程可靠的UID和PID（用户ID进程ID），从而无法鉴别对方身份。Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志。使用传统IPC只能由用户在数据包里填入UID和PID，但这样不可靠，容易被恶意程序利用。可靠的身份标记只有由IPC机制本身在内核中添加。其次传统IPC访问接入点是开放的，无法建立私有通道。比如命名管道的名称，systemV的键值，socket的ip地址或文件名都是开放的，只要知道这些接入点的程序都可以和对端建立连接，不管怎样都无法阻止恶意程序通过猜测接收方地址获得连接。

基于以上原因，Android需要建立一套新的IPC机制来满足系统对通信方式，传输性能和安全性的要求，这就是Binder。**Binder基于Client-Server通信模式，传输过程只需一次拷贝，为发送发添加UID/PID身份，既支持实名Binder也支持匿名Binder，安全性高。**

二、面向对象的 Binder IPC

Binder使用Client-Server通信方式：一个进程作为Server提供诸如视频/音频解码，视频捕获，地址本查询，网络连接等服务；多个进程作为Client向Server发起服务请求，获得所需要的服务。要想实现Client-Server通信必须实现以下两点：一是server必须有确定的访问接入点或者说地址来接受Client的请求，并且Client可以通过某种途径获知Server的地址；二是制定Command-Reply协议来传输数据。

例如在网络通信中Server的访问接入点就是Server主机的IP地址+端口号，传输协议为TCP协议。**对Binder而言，Binder可以看成Server提供的实现某个特定服务的访问接入点， Client通过这个'地址'向Server发送请求来使用该服务；对Client而言，Binder可以看成是通向Server的管道入口，要想和某个Server通信首先必须建立这个管道并获得管道入口。**

与其它IPC不同，Binder使用了面向对象的思想来描述作为访问接入点的Binder及其在Client中的入口：Binder是一个实体位于Server中的对象，该对象提供了一套方法用以实现对服务的请求，就象类的成员函数。遍布于client中的入口可以看成指向这个binder对象的'指针'，一旦获得了这个'指针'就可以调用该对象的方法访问server。在Client看来，通过Binder'指针'调用其提供的方法和通过指针调用其它任何本地对象的方法并无区别，尽管前者的实体位于远端Server中，而后者实体位于本地内存中。'指针'是C++的术语，而更通常的说法是引用，即Client通过Binder的引用访问Server。而软件领域另一个术语'句柄'也可以用来表述Binder在Client中的存在方式。从通信的角度看，Client中的Binder也可以看作是Server Binder的'代理'，在本地代表远端Server为Client提供服务。本文中会使用'引用'或'句柄'这个两广泛使用的术语。

面向对象思想的引入将进程间通信转化为通过对某个Binder对象的引用调用该对象的方法，而其独特之处在于Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。最诱人的是，这个引用和java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其它进程，让大家都能访问同一Server，就象将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。形形色色的Binder对象以及星罗棋布的引用仿佛粘接各个应用程序的胶水，这也是Binder在英文里的原意。

当然面向对象只是针对应用程序而言，对于Binder驱动和内核其它模块一样使用C语言实现，没有类和对象的概念。Binder驱动为面向对象的进程间通信提供底层支持。

三、Binder 通信模型

Binder框架定义了四个角色：Server，Client，ServiceManager（以后简称SMgr）以及Binder驱动。其中Server，Client，SMgr运行于用户空间，驱动运行于内核空间。这四个角色的关系和互联网类似：Server是服务器，Client是客户终端，SMgr是域名服务器（DNS），驱动是路由器。

3.1 Binder 驱动

和路由器一样，Binder驱动虽然默默无闻，却是通信的核心。尽管名叫'驱动'，实际上和硬件设备没有任何关系，只是实现方式和设备驱动程序是一样的。它工作于内核态，驱动负责进程之间Binder通信的建立，Binder在进程之间的传递，Binder引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。

3.2 ServiceManager 与实名Binder

和DNS类似，SMgr的作用是将字符形式的Binder名字转化成Client中对该Binder的引用，使得Client能够通过Binder名字获得对Server中Binder实体的引用。注册了名字的Binder叫实名Binder，就象每个网站除了有IP地址外还有自己的网址。Server创建了Binder实体，为其取一个字符形式，可读易记的名字，将这个Binder连同名字以数据包的形式通过Binder驱动发送给SMgr，通知SMgr注册一个名叫张三的Binder，它位于某个Server中。驱动为这个穿过进程边界的Binder创建位于内核中的实体节点以及SMgr对实体的引用，将名字及新建的引用打包传递给SMgr。SMgr收数据包后，从中取出名字和引用填入一张查找表中。

细心的读者可能会发现其中的蹊跷：SMgr是一个进程，Server是另一个进程，Server向SMgr注册Binder必然会涉及进程间通信。当前实现的是进程间通信却又要用到进程间通信，这就好象蛋可以孵出鸡前提却是要找只鸡来孵蛋。Binder的实现比较巧妙：预先创造一只鸡来孵蛋：SMgr和其它进程同样采用Binder通信，SMgr是Server端，有自己的Binder对象（实体），其它进程都是Client，需要通过这个Binder的引用来实现Binder的注册，查询和获取。SMgr提供的Binder比较特殊，它没有名字也不需要注册，当一个进程使用BINDER_SET_CONTEXT_MGR命令将自己注册成SMgr时Binder驱动会自动为它创建Binder实体（这就是那只预先造好的鸡）。其次这个Binder的引用在所有Client中都固定为0而无须通过其它手段获得。也就是说，一个Server若要向SMgr注册自己Binder就必需通过0这个引用号和SMgr的Binder通信。类比网络通信，0号引用就好比域名服务器的地址，你必须预先手工或动态配置好。要注意这里说的Client是相对SMgr而言的，一个应用程序可能是个提供服务的Server，但对SMgr来说它仍然是个Client。

3.3 Client 获得实名Binder的引用

Server向SMgr注册了Binder实体及其名字后，Client就可以通过名字获得该Binder的引用了。Client也利用保留的0号引用向SMgr请求访问某个Binder：我申请获得名字叫张三的Binder的引用。SMgr收到这个连接请求，从请求数据包里获得Binder的名字，在查找表里找到该名字对应的条目，从条目中取出Binder的引用，将该引用作为回复发送给发起请求的Client。从面向对象的角度，这个Binder对象现在有了两个引用：一个位于SMgr中，一个位于发起请求的Client中。如果接下来有更多的Client请求该Binder，系统中就会有更多的引用指向该Binder，就象java里一个对象存在多个引用一样。而且类似的这些指向Binder的引用是强类型，从而确保只要有引用Binder实体就不会被释放掉。通过以上过程可以看出，SMgr象个火车票代售点，收集了所有火车的车票，可以通过它购买到乘坐各趟火车的票-得到某个Binder的引用。

3.4 匿名 Binder

并不是所有Binder都需要注册给SMgr广而告之的。Server端可以通过已经建立的Binder连接将创建的Binder实体传给Client，当然这条已经建立的Binder连接必须是通过实名Binder实现。由于这个Binder没有向SMgr注册名字，所以是个匿名Binder。Client将会收到这个匿名Binder的引用，通过这个引用向位于Server中的实体发送请求。匿名Binder为通信双方建立一条私密通道，只要Server没有把匿名Binder发给别的进程，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。

四、Binder 内存映射和接收缓存区管理

暂且撇开Binder，考虑一下传统的IPC方式中，数据是怎样从发送端到达接收端的呢？通常的做法是，发送方将准备好的数据存放在缓存区中，调用API通过系统调用进入内核中。内核服务程序在内核空间分配内存，将数据从发送方缓存区复制到内核缓存区中。接收方读数据时也要提供一块缓存区，内核将数据从内核缓存区拷贝到接收方提供的缓存区中并唤醒接收线程，完成一次数据发送。这种存储-转发机制有两个缺陷：首先是效率低下，需要做两次拷贝：用户空间->内核空间->用户空间。Linux使用copy_from_user()和copy_to_user()实现这两个跨空间拷贝，在此过程中如果使用了高端内存（high memory），这种拷贝需要临时建立/取消页面映射，造成性能损失。其次是接收数据的缓存要由接收方提供，可接收方不知道到底要多大的缓存才够用，只能开辟尽量大的空间或先调用API接收消息头获得消息体大小，再开辟适当的空间接收消息体。两种做法都有不足，不是浪费空间就是浪费时间。

Binder采用一种全新策略：由Binder驱动负责管理数据接收缓存。我们注意到Binder驱动实现了mmap()系统调用，这对字符设备是比较特殊的，因为mmap()通常用在有物理存储介质的文件系统上，而象Binder这样没有物理介质，纯粹用来通信的字符设备没必要支持mmap()。Binder驱动当然不是为了在物理介质和用户空间做映射，而是用来创建数据接收的缓存空间。先看mmap()是如何使用的：

fd = open("/dev/binder", O_RDWR);

mmap(NULL, MAP_SIZE, PROT_READ, MAP_PRIVATE, fd, 0);

这样Binder的接收方就有了一片大小为MAP_SIZE的接收缓存区。mmap()的返回值是内存映射在用户空间的地址，不过这段空间是由驱动管理，用户不必也不能直接访问（映射类型为PROT_READ，只读映射）。

接收缓存区映射好后就可以做为缓存池接收和存放数据了。前面说过，接收数据包的结构为binder_transaction_data，但这只是消息头，真正的有效负荷位于data.buffer所指向的内存中。这片内存不需要接收方提供，恰恰是来自mmap()映射的这片缓存池。在数据从发送方向接收方拷贝时，驱动会根据发送数据包的大小，使用最佳匹配算法从缓存池中找到一块大小合适的空间，将数据从发送缓存区复制过来。要注意的是，存放binder_transaction_data结构本身以及表4中所有消息的内存空间还是得由接收者提供，但这些数据大小固定，数量也不多，不会给接收方造成不便。映射的缓存池要足够大，因为接收方的线程池可能会同时处理多条并发的交互，每条交互都需要从缓存池中获取目的存储区，一旦缓存池耗竭将产生导致无法预期的后果。

有分配必然有释放。接收方在处理完数据包后，就要通知驱动释放data.buffer所指向的内存区。在介绍Binder协议时已经提到，这是由命令BC_FREE_BUFFER完成的。

通过上面介绍可以看到，驱动为接收方分担了最为繁琐的任务：分配/释放大小不等，难以预测的有效负荷缓存区，而接收方只需要提供缓存来存放大小固定，最大空间可以预测的消息头即可。在效率上，由于mmap()分配的内存是映射在接收方用户空间里的，所有总体效果就相当于对有效负荷数据做了一次从发送方用户空间到接收方用户空间的直接数据拷贝，省去了内核中暂存这个步骤，提升了一倍的性能。顺便再提一点，Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy_from_user()拷贝到内核空间，再用copy_to_user()拷贝到另一个用户空间。为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间。所以调用copy_from_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间，这就是Binder只需一次拷贝的‘秘密’。

五、Binder 接收线程管理

Binder通信实际上是位于不同进程中的线程之间的通信。假如进程S是Server端，提供Binder实体，线程T1从Client进程C1中通过Binder的引用向进程S发送请求。S为了处理这个请求需要启动线程T2，而此时线程T1处于接收返回数据的等待状态。T2处理完请求就会将处理结果返回给T1，T1被唤醒得到处理结果。在这过程中，T2仿佛T1在进程S中的代理，代表T1执行远程任务，而给T1的感觉就是象穿越到S中执行一段代码又回到了C1。为了使这种穿越更加真实，驱动会将T1的一些属性赋给T2，特别是T1的优先级nice，这样T2会使用和T1类似的时间完成任务。很多资料会用‘线程迁移’来形容这种现象，容易让人产生误解。一来线程根本不可能在进程之间跳来跳去，二来T2除了和T1优先级一样，其它没有相同之处，包括身份，打开文件，栈大小，信号处理，私有数据等。

对于Server进程S，可能会有许多Client同时发起请求，为了提高效率往往开辟线程池并发处理收到的请求。怎样使用线程池实现并发处理呢？这和具体的IPC机制有关。拿socket举例，Server端的socket设置为侦听模式，有一个专门的线程使用该socket侦听来自Client的连接请求，即阻塞在accept()上。这个socket就象一只会生蛋的鸡，一旦收到来自Client的请求就会生一个蛋 – 创建新socket并从accept()返回。侦听线程从线程池中启动一个工作线程并将刚下的蛋交给该线程。后续业务处理就由该线程完成并通过这个单与Client实现交互。

可是对于Binder来说，既没有侦听模式也不会下蛋，怎样管理线程池呢？一种简单的做法是，不管三七二十一，先创建一堆线程，每个线程都用BINDER_WRITE_READ命令读Binder。这些线程会阻塞在驱动为该Binder设置的等待队列上，一旦有来自Client的数据驱动会从队列中唤醒一个线程来处理。这样做简单直观，省去了线程池，但一开始就创建一堆线程有点浪费资源。于是Binder协议引入了专门命令或消息帮助用户管理线程池，包括：

· INDER_SET_MAX_THREADS

· BC_REGISTER_LOOP

· BC_ENTER_LOOP

· BC_EXIT_LOOP

· BR_SPAWN_LOOPER

首先要管理线程池就要知道池子有多大，应用程序通过INDER_SET_MAX_THREADS告诉驱动最多可以创建几个线程。以后每个线程在创建，进入主循环，退出主循环时都要分别使用BC_REGISTER_LOOP，BC_ENTER_LOOP，BC_EXIT_LOOP告知驱动，以便驱动收集和记录当前线程池的状态。每当驱动接收完数据包返回读Binder的线程时，都要检查一下是不是已经没有闲置线程了。如果是，而且线程总数不会超出线程池最大线程数，就会在当前读出的数据包后面再追加一条BR_SPAWN_LOOPER消息，告诉用户线程即将不够用了，请再启动一些，否则下一个请求可能不能及时响应。新线程一启动又会通过BC_xxx_LOOP告知驱动更新状态。这样只要线程没有耗尽，总是有空闲线程在等待队列中随时待命，及时处理请求。

关于工作线程的启动，Binder驱动还做了一点小小的优化。当进程P1的线程T1向进程P2发送请求时，驱动会先查看一下线程T1是否也正在处理来自P2某个线程请求但尚未完成（没有发送回复）。这种情况通常发生在两个进程都有Binder实体并互相对发时请求时。假如驱动在进程P2中发现了这样的线程，比如说T2，就会要求T2来处理T1的这次请求。因为T2既然向T1发送了请求尚未得到返回包，说明T2肯定（或将会）阻塞在读取返回包的状态。这时候可以让T2顺便做点事情，总比等在那里闲着好。而且如果T2不是线程池中的线程还可以为线程池分担部分工作，减少线程池使用率。

六、数据包接收队列与（线程）等待队列管理

通常数据传输的接收端有两个队列：数据包接收队列和（线程）等待队列，用以缓解供需矛盾。当超市里的进货（数据包）太多，货物会堆积在仓库里；购物的人（线程）太多，会排队等待在收银台，道理是一样的。在驱动中，每个进程有一个全局的接收队列，也叫to-do队列，存放不是发往特定线程的数据包；相应地有一个全局等待队列，所有等待从全局接收队列里收数据的线程在该队列里排队。每个线程有自己私有的to-do队列，存放发送给该线程的数据包；相应的每个线程都有各自私有等待队列，专门用于本线程等待接收自己to-do队列里的数据。虽然名叫队列，其实线程私有等待队列中最多只有一个线程，即它自己。

由于发送时没有特别标记，驱动怎么判断哪些数据包该送入全局to-do队列，哪些数据包该送入特定线程的to-do队列呢？这里有两条规则。规则1：Client发给Server的请求数据包都提交到Server进程的全局to-do队列。不过有个特例，就是上节谈到的Binder对工作线程启动的优化。经过优化，来自T1的请求不是提交给P2的全局to-do队列，而是送入了T2的私有to-do队列。规则2：对同步请求的返回数据包（由BC_REPLY发送的包）都发送到发起请求的线程的私有to-do队列中。如上面的例子，如果进程P1的线程T1发给进程P2的线程T2的是同步请求，那么T2返回的数据包将送进T1的私有to-do队列而不会提交到P1的全局to-do队列。

数据包进入接收队列的潜规则也就决定了线程进入等待队列的潜规则，即一个线程只要不接收返回数据包则应该在全局等待队列中等待新任务，否则就应该在其私有等待队列中等待Server的返回数据。还是上面的例子，T1在向T2发送同步请求后就必须等待在它私有等待队列中，而不是在P1的全局等待队列中排队，否则将得不到T2的返回的数据包。

这些潜规则是驱动对Binder通信双方施加的限制条件，体现在应用程序上就是同步请求交互过程中的线程一致性：1) Client端，等待返回包的线程必须是发送请求的线程，而不能由一个线程发送请求包，另一个线程等待接收包，否则将收不到返回包；2) Server端，发送对应返回数据包的线程必须是收到请求数据包的线程，否则返回的数据包将无法送交发送请求的线程。这是因为返回数据包的目的Binder不是用户指定的，而是驱动记录在收到请求数据包的线程里，如果发送返回包的线程不是收到请求包的线程驱动将无从知晓返回包将送往何处。

接下来探讨一下Binder驱动是如何递交同步交互和异步交互的。我们知道，同步交互和异步交互的区别是同步交互的请求端（client）在发出请求数据包后须要等待应答端（Server）的返回数据包，而异步交互的发送端发出请求数据包后交互即结束。对于这两种交互的请求数据包，驱动可以不管三七二十一，统统丢到接收端的to-do队列中一个个处理。但驱动并没有这样做，而是对异步交互做了限流，令其为同步交互让路，具体做法是：对于某个Binder实体，只要有一个异步交互没有处理完毕，例如正在被某个线程处理或还在任意一条to-do队列中排队，那么接下来发给该实体的异步交互包将不再投递到to-do队列中，而是阻塞在驱动为该实体开辟的异步交互接收队列（Binder节点的async_todo域）中，但这期间同步交互依旧不受限制直接进入to-do队列获得处理。一直到该异步交互处理完毕下一个异步交互方可以脱离异步交互队列进入to-do队列中。之所以要这么做是因为同步交互的请求端需要等待返回包，必须迅速处理完毕以免影响请求端的响应速度，而异步交互属于‘发射后不管’，稍微延时一点不会阻塞其它线程。所以用专门队列将过多的异步交互暂存起来，以免突发大量异步交互挤占Server端的处理能力或耗尽线程池里的线程，进而阻塞同步交互。

七、总结

Binder使用Client-Server通信方式，安全性好，简单高效，再加上其面向对象的设计思想，独特的接收缓存管理和线程池管理方式，成为Android进程间通信的中流砥柱。

Binder架构

Binder 通信采用 C/S 架构，从组件视角来说，包含 Client、 Server、 ServiceManager 以及 Binder 驱动，其中 ServiceManager 用于管理系统中的各种服务。 Binder 在 framework 层进行了封装，通过 JNI 技术调用 Native（C/C++）层的 Binder 架构。

Binder 在 Native 层以 ioctl 的方式与 Binder 驱动通讯。

Binder机制

首先需要注册服务端，只有注册了服务端，客户端才有通讯的目标，服务端通过 ServiceManager 注册服务，注册的过程就是向 Binder 驱动的全局链表 binder_procs 中插入服务端的信息（binder_proc 结构体，每个 binder_proc 结构体中都有 todo 任务队列），然后向 ServiceManager 的 svcinfo 列表中缓存一下注册的服务。

有了服务端，客户端就可以跟服务端通讯了，通讯之前需要先获取到服务，拿到服务的代理，也可以理解为引用。比如下面的代码：

```
//获取WindowManager服务引用
WindowManager wm =
(WindowManager)getSystemService(getApplication().WINDOW_SERVICE);
```

获取服务端的方式就是通过 ServiceManager 向 svcinfo 列表中查询一下返回服务端的代理，svcinfo 列表就是所有已注册服务的通讯录，保存了所有注册的服务信息。

有了服务端的引用我们就可以向服务端发送请求了，通过 BinderProxy 将我们的请求参数发送给 ServiceManager，通过共享内存的方式使用内核方法 copy_from_user() 将我们的参数先拷贝到内核空间，这时我们的客户端进入等待状态，然后 Binder 驱动向服务端的 todo 队列里面插入一条事务，执行完之后把执行结果通过 copy_to_user() 将内核的结果拷贝到用户空间（这里只是执行了拷贝命令，并没有拷贝数据，binder只进行一次拷贝），唤醒等待的客户端并把结果响应回来，这样就完成了一次通讯。

Binder驱动

用户空间/内核空间 详细解释可以参考 Kernel Space Definition； 简单理解如下：

Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。 为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

Kernel space 可以执行任意命令，调用系统的一切资源； User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。

系统调用/内核态/用户态 虽然从逻辑上抽离出用户空间和内核空间；但是不可避免的的是，总有那么一些用户空间需要访问内核的资源；比如应用程序访问文件，网络是很常见的事情，怎么办呢?

```
Kernel space can be accessed by user processes only through the use of system calls.
```

用户空间访问内核空间的唯一方式就是系统调用；通过这个统一入口接口，所有的资源访问都是在内核的控制下执行，以免导致对用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。用户软件良莠不齐，要是它们乱搞把系统玩坏了怎么办？因此对于某些特权操作必须交给安全可靠的内核来执行。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）此时处理器处于特权级最高的（0级）内核代码中执行。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。处理器在特权等级高的时候才能执行那些特权CPU指令。

内核模块/驱动 通过系统调用，用户空间可以访问内核空间，那么如果一个用户空间想与另外一个用户空间进行通信怎么办呢？很自然想到的是让操作系统内核添加支持；传统的 Linux 通信机制，比如 Socket，管道等都是内核支持的；但是 Binder 并不是 Linux 内核的一部分，它是怎么做到访问内核空间的呢？ Linux 的动态可加载内核模块（Loadable Kernel Module，LKM）机制解决了这个问题；模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行。这样，Android系统可以通过添加一个内核模块运行在内核空间，用户进程之间的通过这个模块作为桥梁，就可以完成通信了。

在 Android 系统中，这个运行在内核空间的，负责各个用户进程通过 Binder 通信的内核模块叫做 Binder 驱动;

驱动程序一般指的是设备驱动程序（Device Driver），是一种可以使计算机和设备通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作；

驱动就是操作硬件的接口，为了支持Binder通信过程，Binder 使用了一种“硬件”，因此这个模块被称之为驱动。

熟悉了上面这些概念，我们再来看下上面的图，用户空间中 binder_open(), binder_mmap(), binder_ioctl() 这些方法通过 system call 来调用内核空间 Binder 驱动中的方法。内核空间与用户空间共享内存通过 copy_from_user(), copy_to_user() 内核方法来完成用户空间与内核空间内存的数据传输。 Binder驱动中有一个全局的 binder_procs 链表保存了服务端的进程信息。

Binder 进程与线程

对于底层Binder驱动，通过 binder_procs 链表记录所有创建的 binder_proc 结构体，binder 驱动层的每一个 binder_proc 结构体都与用户空间的一个用于 binder 通信的进程一一对应，且每个进程有且只有一个 ProcessState 对象，这是通过单例模式来保证的。在每个进程中可以有很多个线程，每个线程对应一个 IPCThreadState 对象，IPCThreadState 对象也是单例模式，即一个线程对应一个 IPCThreadState 对象，在 Binder 驱动层也有与之相对应的结构，那就是 Binder_thread 结构体。在 binder_proc 结构体中通过成员变量 rb_root threads，来记录当前进程内所有的 binder_thread。

Binder 线程池：每个 Server 进程在启动时创建一个 binder 线程池，并向其中注册一个 Binder 线程；之后 Server 进程也可以向 binder 线程池注册新的线程，或者 Binder 驱动在探测到没有空闲 binder 线程时主动向 Server 进程注册新的的 binder 线程。对于一个 Server 进程有一个最大 Binder 线程数限制，默认为16个 binder 线程，例如 Android 的 system_server 进程就存在16个线程。对于所有 Client 端进程的 binder 请求都是交由 Server 端进程的 binder 线程来处理的。

ServiceManager 启动

了解了 Binder 驱动，怎么与 Binder 驱动进行通讯呢？那就是通过 ServiceManager，好多文章称 ServiceManager 是 Binder 驱动的守护进程，大管家，其实 ServiceManager 的作用很简单就是提供了查询服务和注册服务的功能。下面我们来看一下 ServiceManager 启动的过程。

ServiceManager 分为 framework 层和 native 层，framework 层只是对 native 层进行了封装方便调用，图上展示的是 native 层的 ServiceManager 启动过程。

ServiceManager 的启动是系统在开机时，init 进程解析 init.rc 文件调用 service_manager.c 中的 main() 方法入口启动的。 native 层有一个 binder.c 封装了一些与 Binder 驱动交互的方法。

ServiceManager 的启动分为三步，首先打开驱动创建全局链表 binder_procs，然后将自己当前进程信息保存到 binder_procs 链表，最后开启 loop 不断的处理共享内存中的数据，并处理 BR_xxx 命令（ioctl 的命令，BR 可以理解为 binder reply 驱动处理完的响应）。

ServiceManager 注册服务

注册 MediaPlayerService 服务端，我们通过 ServiceManager 的 addService() 方法来注册服务。

首先 ServiceManager 向 Binder 驱动发送 BC_TRANSACTION 命令（ioctl 的命令，BC 可以理解为 binder client 客户端发过来的请求命令）携带 ADD_SERVICE_TRANSACTION 命令，同时注册服务的线程进入等待状态 waitForResponse()。 Binder 驱动收到请求命令向 ServiceManager 的 todo 队列里面添加一条注册服务的事务。事务的任务就是创建服务端进程 binder_node 信息并插入到 binder_procs 链表中。

事务处理完之后发送 BR_TRANSACTION 命令，ServiceManager 收到命令后向 svcinfo 列表中添加已经注册的服务。最后发送 BR_REPLY 命令唤醒等待的线程，通知注册成功。

进行一次完整通讯

我们在使用 Binder 时基本都是调用 framework 层封装好的方法，AIDL 就是 framework 层提供的傻瓜式使用方式。假设服务已经注册完，我们来看看客户端怎么执行服务端的方法。

首先我们通过 ServiceManager 获取到服务端的 BinderProxy 代理对象，通过调用 BinderProxy 将参数，方法标识（例如：TRANSACTION_test，AIDL中自动生成）传给 ServiceManager，同时客户端线程进入等待状态。

ServiceManager 将用户空间的参数等请求数据复制到内核空间，并向服务端插入一条执行执行方法的事务。事务执行完通知 ServiceManager 将执行结果从内核空间复制到用户空间，并唤醒等待的线程，响应结果，通讯结束。

13.屏幕适配的处理技巧都有哪些?

（1）相关重要概念

屏幕尺寸

- 含义：手机对角线的物理尺寸
- 单位：英寸（inch），1英寸=2.54cm

> Android手机常见的尺寸有5寸、5.5寸、6寸等等

屏幕分辨率

- 含义：手机在横向、纵向上的像素点数总和

> 1. 一般描述成屏幕的"宽x高”=AxB
> 2. 含义：屏幕在横向方向（宽度）上有A个像素点，在纵向方向 （高）有B个像素点
> 3. 例子：1080x1920，即宽度方向上有1080个像素点，在高度方向上有1920个像素点

- 单位：px（pixel），1px=1像素点

> UI设计师的设计图会以px作为统一的计量单位

- Android手机常见的分辨率：320x480、480x800、720x1280、1080x1920

屏幕像素密度

- 含义：每英寸的像素点数
- 单位：dpi（dots per ich）

> 假设设备内每英寸有160个像素，那么该设备的屏幕像素密度=160dpi

- 安卓手机对于每类手机屏幕大小都有一个相应的屏幕像素密度：

密度类型	代表的分辨率（px）	屏幕像素密度（dpi）
低密度（ldpi）	240x320	120
中密度（mdpi）	320x480	160
高密度（hdpi）	480x800	240
超高密度（xhdpi）	720x1280	320
超超高密度（xxhdpi）	1080x1920	480

屏幕尺寸、分辨率、像素密度三者关系

一部手机的分辨率是宽x高，屏幕大小是以寸为单位，那么三者的关系是：

密度无关像素

- 含义：density-independent pixel，叫dp或dip，与终端上的实际物理像素点无关。
- 单位：dp，可以保证在不同屏幕像素密度的设备上显示相同的效果

> 1. Android开发时用dp而不是px单位设置图片大小，是Android特有的单位
> 2. 场景：假如同样都是画一条长度是屏幕一半的线，如果使用px作为计量单位，那么在480x800分辨率手机上设置应为240px；在320x480的手机上应设置为160px，二者设置就不同了；如果使用dp为单位，在这两种分辨率下，160dp都显示为屏幕一半的长度。

- dp与px的转换 因为ui设计师给你的设计图是以px为单位的，Android开发则是使用dp作为单位的，那么我们需要进行转换：

密度类型	代表的分辨率（px）	屏幕密度（dpi）	换算（px/dp）	比例
低密度（ldpi）	240x320	120	1dp=0.75px	3
中密度（mdpi）	320x480	160	1dp=1px	4
高密度（hdpi）	480x800	240	1dp=1.5px	6
超高密度（xhdpi）	720x1280	320	1dp=2px	8
超超高密度（xxhdpi）	1080x1920	480	1dp=3px	12

在Android中，规定以160dpi（即屏幕分辨率为320x480）为基准：1dp=1px

独立比例像素

- 含义：scale-independent pixel，叫sp或sip
- 单位：sp

> 1. Android开发时用此单位设置文字大小，可根据字体大小首选项进行缩放
> 2. 推荐使用12sp、14sp、18sp、22sp作为字体设置的大小，不推荐使用奇数和小数，容易造

> 成精度的丢失问题；小于12sp的字体会太小导致用户看不清

（2）为什么要进行Android屏幕适配

由于Android系统的开放性，任何用户、开发者、OEM厂商、运营商都可以对Android进行定制，于是导致：

- Android系统碎片化：小米定制的MIUI、魅族定制的flyme、华为定制的EMUI等等

> 当然都是基于Google原生系统定制的

- Android机型屏幕尺寸碎片化：5寸、5.5寸、6寸等等
- Android屏幕分辨率碎片化：320x480、480x800、720x1280、1080x1920

> 据友盟指数显示，统计至2015年12月，支持Android的设备共有27796种

当Android系统、屏幕尺寸、屏幕密度出现碎片化的时候，就很容易出现同一元素在不同手机上显示不同的问题。

> 试想一下这么一个场景： 为4.3寸屏幕准备的UI设计图，运行在5.0寸的屏幕上，很可能在右侧和下侧存在大量的空白；而5.0寸的UI设计图运行到4.3寸的设备上，很可能显示不下。

为了保证用户获得一致的用户体验效果：

> 使得某一元素在Android不同尺寸、不同分辨率的手机上具备相同的显示效果

于是，我们便需要对Android屏幕进行适配。

（3）屏幕适配问题的本质

- 使得“布局”、“布局组件”、“图片资源”、“用户界面流程”匹配**不同的屏幕尺寸**

> 使得布局、布局组件自适应屏幕尺寸； 根据屏幕的配置来加载相应的UI布局、用户界面流程

- 使得“图片资源”匹配**不同的屏幕密度**

（4）解决方案

1）“布局”匹配

本质1：使得布局元素自适应屏幕尺寸

- 做法 使用相对布局（RelativeLayout），禁用绝对布局（AbsoluteLayout）

开发中，我们使用的布局一般有：

- 线性布局（Linearlayout）
- 相对布局（RelativeLayout）
- 帧布局（FrameLayout）
- 绝对布局（AbsoluteLayout）

> 由于绝对布局（AbsoluteLayout）适配性极差，所以极少使用。

对于线性布局（Linearlayout）、相对布局（RelativeLayout）和帧布局（FrameLayout）需要根据需求进行选择，但要记住：

- RelativeLayout 布局的子控件之间使用**相对位置**的方式排列，因为RelativeLayout讲究的是**相对位置**，即使屏幕的大小改变，视图之前的相对位置都不会变化，与屏幕大小无关，灵活性很强
- LinearLayout 通过多层嵌套LinearLayout和组合使用"wrap_content"和"match_parent"已经可以构建出足够复杂的布局。但是LinearLayout**无法准确地控制子视图之间的位置关系**，只能简单的一个挨着一个地排列

所以，对于屏幕适配来说，使用相对布局（RelativeLayout）将会是更好的解决方案

本质2：根据屏幕的配置来加载相应的UI布局

> 应用场景：需要为不同屏幕尺寸的设备设计不同的布局

- 做法：**使用限定符**
- 作用：通过配置**限定符**使得程序在运行时根据当前设备的配置（屏幕尺寸）自动加载合适的布局资源
- 限定符类型：
- 尺寸（size）限定符
- 最小宽度（Smallest-width）限定符
- 布局别名
- 屏幕方向（Orientation）限定符

尺寸（size）限定符

- 使用场景：当一款应用显示的内容较多，希望进行以下设置：

> - 在平板电脑和电视的屏幕（>7英寸）上：实施**“双面板”**模式以同时显示更多内容
> - 在手机较小的屏幕上：使用**单面板**分别显示内容

因此，我们可以使用尺寸限定符（layout-large）通过创建一个文件

```
res/layout-large/main.xml
```

来完成上述设定：

> - 让系统在屏幕尺寸>7英寸时采用适配平板的双面板布局
> - 反之（默认情况下）采用适配手机的单面板布局

文件配置如下：

- 适配手机的单面板（默认）布局：res/layout/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

- 适配尺寸>7寸平板的双面板布局：： res/layout-large/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

> 请注意：

- 两个布局名称均为main.xml，只有布局的目录名不同：第一个布局的目录名为：layout，第二个布局的目录名为：layout-large，包含了尺寸限定符（large）
- 被定义为大屏的设备(7寸以上的平板)会自动加载包含了large限定符目录的布局，而小屏设备会加载另一个默认的布局

但要注意的是，这种方式只适合Android 3.2版本之前。

最小宽度（Smallest-width）限定符

> - 背景：上述提到的限定符“large”具体是指多大呢？似乎没有一个定量的指标，这便意味着可能没办法准确地根据当前设备的配置（屏幕尺寸）自动加载合适的布局资源
> - 例子：比如说large同时包含着5寸和7寸，这意味着使用“large”限定符的话我没办法实现为5寸和7寸的平板电脑分别加载不同的布局

于是，在Android 3.2及之后版本，引入了最小宽度（Smallest-width）限定符

> 定义：通过指定某个最小宽度（以 dp 为单位）来精确定位屏幕从而加载不同的UI资源

- 使用场景

> 你需要为标准 7 英寸平板电脑匹配双面板布局（其最小宽度为 600 dp），在手机（较小的屏幕上）匹配单面板布局

解决方案：您可以使用上文中所述的单面板和双面板这两种布局，但您应使用 **sw600dp** 指明双面板布局仅适用于最小宽度为 600 dp 的屏幕，而不是使用 large 尺寸限定符。

> - sw xxxdp，即small width的缩写，其不区分方向，即无论是宽度还是高度，只要大于xxxdp，就采用次此布局

- 例子：使用了layout-sw 600dp的最小宽度限定符，即无论是宽度还是高度，只要大于600dp，就采用layout-sw 600dp目录下的布局

代码展示：

- 适配手机的单面板（默认）布局：res/layout/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

- 适配尺寸>7寸平板的双面板布局：res/layout-sw600dp/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

> - 对于最小宽度≥ 600 dp 的设备 系统会自动加载 layout-sw600dp/main.xml（双面板）布

> 局，否则系统就会选择 layout/main.xml（单面板）布局 （这个选择过程是Android系统自动选择的）

使用布局别名

设想这么一个场景

> 当你需要同时为Android 3.2版本前和Android 3.2版本后的手机进行屏幕尺寸适配的时候，由于尺寸限定符仅用于Android 3.2版本前，最小宽度限定符仅用于Android 3.2版本后，所以这会带来一个问题，为了很好地进行屏幕尺寸的适配，你需要同时维护layout-sw600dp和layout-large的两套main.xml平板布局，如下：

- 适配手机的单面板（默认）布局：res/layout/main.xml
- 适配尺寸>7寸平板的双面板布局（Android 3.2前）：res/layout-large/main.xml
- 适配尺寸>7寸平板的双面板布局（Android 3.2后）res/layout-sw600dp/main.xml

最后的两个文件的xml内容是完全相同的，这会带来：**文件名的重复从而带来一些列后期维护的问题**

于是为了要解决这种重复问题，我们引入了“布局别名”

还是上面的例子，你可以定义以下布局：

- 适配手机的单面板（默认）布局：res/layout/main.xml
- 适配尺寸>7寸平板的双面板布局：res/layout/main_twopanes.xml

然后加入以下两个文件，以便进行Android 3.2前和Android 3.2后的版本双面板布局适配：

1. res/values-large/layout.xml（Android 3.2之前的双面板布局）

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

1. res/values-sw600dp/layout.xml（Android 3.2及之后的双面板布局）

```
<resources>
<item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

> 注：

- 最后两个文件有着相同的内容，但是它们并没有真正去定义布局，它们仅仅只是**将main设置成了@layout/main_twopanes的别名**
- 由于这些文件包含 large 和 sw600dp 选择器，因此，系统会将此文件匹配到不同版本的>7寸平板上： a. 版本低于 3.2 的平板会匹配 large的文件 b. 版本高于 3.2 的平板会匹配 sw600dp的文件

这样两个layout.xml都只是引用了@layout/main_twopanes，就避免了重复定义布局文件的情况

屏幕方向（Orientation）限定符

- 使用场景：根据屏幕方向进行布局的调整

> 取以下为例子：

> - 小屏幕, 竖屏: 单面板
> - 小屏幕, 横屏: 单面板
> - 7 英寸平板电脑，纵向：单面板，带操作栏
> - 7 英寸平板电脑，横向：双面板，宽，带操作栏
> - 10 英寸平板电脑，纵向：双面板，窄，带操作栏
> - 10 英寸平板电脑，横向：双面板，宽，带操作栏
> - 电视，横向：双面板，宽，带操作栏

方法是：

- 先定义类别：单/双面板、是否带操作栏、宽/窄

> 定义在 res/layout/ 目录下的某个 XML 文件中

- 再进行相应的匹配：屏幕尺寸（小屏、7寸、10寸）、方向（横、纵）

> 使用布局别名进行匹配

1. 在 res/layout/ 目录下的某个 XML 文件中定义所需要的布局类别 （单/双面板、是否带操作栏、宽/窄） res/layout/onepane.xml:(单面板)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/onepane_with_bar.xml:(单面板带操作栏)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
                  android:id="@+id/linearLayout1"
                  android:gravity="center"
                  android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
                   android:layout_height="wrap_content"
                   android:layout_width="wrap_content"
                   android:src="@drawable/logo"
                   android:paddingRight="30dp"
                   android:layout_gravity="left"
                   android:layout_weight="0" />
```

```
        <View android:layout_height="wrap_content"
              android:id="@+id/view1"
              android:layout_width="wrap_content"
              android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
                android:background="@drawable/button_bg"
                android:layout_height="match_parent"
                android:layout_weight="0"
                android:layout_width="120dp"
                style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/twopanes.xml:(双面板，宽布局)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

res/layout/twopanes_narrow.xml:(双面板，窄布局)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
```

```
                android:layout_width="200dp"
                android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
                android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
                android:layout_width="fill_parent" />
</LinearLayout>
```

2.使用布局别名进行相应的匹配 (屏幕尺寸（小屏、7寸、10寸）、方向（横、纵）)
res/values/layouts.xml:（默认布局）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>

    <bool name="has_two_panes">false</bool>
</resources>
```

> 可为resources设置bool，通过获取其值来动态判断目前已处在哪个适配布局

res/values-sw600dp-land/layouts.xml （大屏、横向、双面板、宽-Andorid 3.2版本后）

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-sw600dp-port/layouts.xml （大屏、纵向、单面板带操作栏-Andorid 3.2版本后）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-large-land/layouts.xml （大屏、横向、双面板、宽-Andorid 3.2版本前）

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-large-port/layouts.xml （大屏、纵向、单面板带操作栏-Andorid 3.2版本前）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

> 这里没有完全把全部尺寸匹配类型的代码贴出来，大家可以自己去尝试把其补充完整

2）“布局组件”匹配

本质：使得布局组件自适应屏幕尺寸

- 做法 使用"wrap_content"、"match_parent"和"weight“来控制视图组件的宽度和高度

> - "wrap_content" 相应视图的宽和高就会被设定成所需的最小尺寸以适应视图中的内容
> - "match_parent"(在Android API 8之前叫作"fill_parent") 视图的宽和高延伸至充满整个父布局
> - "weight" 1.定义：是线性布局（Linelayout）的一个独特比例分配属性 2.作用：使用此属性设置权重，然后按照比例对界面进行空间的分配，公式计算是：控件宽度=控件设置宽度+剩余空间所占百分比宽幅 具体可以参考这篇文章，讲解得非常详细

通过使用"wrap_content"、"match_parent"和"weight"来替代硬编码的方式定义视图大小&位置，你的视图要么仅仅使用了需要的那边一点空间，要么就会充满所有可用的空间，即按需占据空间大小，能让你的布局元素充分适应你的屏幕尺寸

3）“图片资源”匹配

本质：使得图片资源在不同屏幕密度上显示相同的像素效果

- 做法：**使用自动拉伸位图：Nine-Patch的图片类型** 假设需要匹配不同屏幕大小，你的图片资源也必须自动适应各种屏幕尺寸

> 使用场景：一个按钮的背景图片必须能够随着按钮大小的改变而改变。 使用普通的图片将无法实现上述功能,因为运行时会均匀地拉伸或压缩你的图片

- 解决方案：使用自动拉伸位图（nine-patch图片），后缀名是.9.png，它是一种被特殊处理过的PNG图片，设计时可以指定图片的拉伸区域和非拉伸区域；使用时，系统就会根据控件的大小自动地拉伸你想要拉伸的部分

> 1.必须要使用.9.png后缀名，因为系统就是根据这个来区别nine-patch图片和普通的PNG图片的；

2.当你需要在一个控件中使用nine-patch图片时,如

```
android:background="@drawable/button"
```

系统就会根据控件的大小自动地拉伸你想要拉伸的部分

4）”用户界面流程“匹配

> - 使用场景：我们会根据设备特点显示恰当的布局，但是这样做，会使得用户界面流程可能会有所不同。
> - 例如，如果应用处于双面板模式下，点击左侧面板上的项即可直接在右侧面板上显示相关内容；而如果该应用处于单面板模式下，点击相关的内容应该跳转到另外一个Activity进行后续的处理。

本质：根据屏幕的配置来加载相应的用户界面流程

- 做法 进行用户界面流程的自适应配置：

1. 确定当前布局
2. 根据当前布局做出响应
3. 重复使用其他活动中的片段
4. 处理屏幕配置变化

- 步骤1：确定当前布局 由于每种布局的实施都会稍有不同，因此我们需要先确定当前向用户显示的布局。例如，我们可以先了解用户所处的是“单面板”模式还是“双面板”模式。要做到这一点，可以通过查询指定视图是否存在以及是否已显示出来。

```
public class NewsReaderActivity extends FragmentActivity {
    boolean mIsDualPane;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);

        View articleView = findViewById(R.id.article);
        mIsDualPane = articleView != null &&
                        articleView.getVisibility() == View.VISIBLE;
    }
}
```

> 这段代码用于查询“报道”面板是否可用，与针对具体布局的硬编码查询相比，这段代码的灵活性要大得多。

- 步骤2：根据当前布局做出响应 有些操作可能会因当前的具体布局而产生不同的结果。

> 例如，在新闻阅读器示例中，如果用户界面处于双面板模式下，那么点击标题列表中的标题就会在右侧面板中打开相应报道；但如果用户界面处于单面板模式下，那么上述操作就会启动一个独立活动：

```
@Override
public void onHeadlineSelected(int index) {
    mArtIndex = index;
    if (mIsDualPane) {
        /* display article on the right pane */
        mArticleFragment.displayArticle(mCurrentCat.getArticle(index));
    } else {
        /* start a separate activity */
        Intent intent = new Intent(this, ArticleActivity.class);
        intent.putExtra("catIndex", mCatIndex);
        intent.putExtra("artIndex", index);
        startActivity(intent);
    }
}
```

- 步骤3：重复使用其他活动中的片段 多屏幕设计中的重复模式是指，对于某些屏幕配置，已实施界面的一部分会用作面板；但对于其他配置，这部分就会以独立活动的形式存在。

> 例如，在新闻阅读器示例中，对于较大的屏幕，新闻报道文本会显示在右侧面板中；但对于较小的屏幕，这些文本就会以独立活动的形式存在。

在类似情况下，通常可以在多个活动中重复使用相同的 Fragment 子类以避免代码重复。例如，在双面板布局中使用了 ArticleFragment：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

 android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

 android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

然后又在小屏幕的Activity布局中重复使用了它：

```
ArticleFragment frag = new ArticleFragment();
getSupportFragmentManager().beginTransaction().add(android.R.id.content,
frag).commit();
```

- 步骤3：处理屏幕配置变化 如果我们使用独立Activity实施界面的独立部分，那么请注意，我们可能需要对特定配置变化（例如屏幕方向的变化）做出响应，以便保持界面的一致性。

> 例如，在运行 Android 3.0 或更高版本的标准 7 英寸平板电脑上，如果新闻阅读器示例应用运行在纵向模式下，就会在使用独立活动显示新闻报道；但如果该应用运行在横向模式下，就会使用双面板布局。

也就是说，如果用户处于纵向模式下且屏幕上显示的是用于阅读报道的活动，那么就需要在检测到屏幕方向变化（变成横向模式）后执行相应操作，即停止上述活动并返回主活动，以便在双面板布局中显示相关内容：

```
public class ArticleActivity extends FragmentActivity {
    int mCatIndex, mArtIndex;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        mCatIndex = getIntent().getExtras().getInt("catIndex", 0);
        mArtIndex = getIntent().getExtras().getInt("artIndex", 0);

        // If should be in two-pane mode, finish to return to main
activity
        if (getResources().getBoolean(R.bool.has_two_panes)) {
            finish();
            return;
        }
        ...
}
```

https://www.jianshu.com/p/ec5a1a30694b

https://juejin.im/post/5b7671ede51d4566590c75aa