

Android Code Style

Version 0.9.3

当前版本	Version 0.9.3
制定日期	2012/10/29
文档编号	Android Code Style
文档作者	李洪

修订历史记录

[illegible]

前言

Android经过几年的发展，现在已经成为最流行的移动平台之一，但Android毕竟还是一个较新的系统，而且它是移动平台，与传统的PC平台有本质的差别，我们在开发Android应用程序时，要从传统的思维方式转变过来，希望本文档能对大家有所帮助。

为什么写本文档

从第一次接触Android到现在，我们团队积累了很多关于Android开发的技术，解决方案等，这些东西都分散地存储在我们团队的每一个人的大脑中，这些技术只有他们自己知道，每当我们的团队的新成员加入时，总是要一遍一遍地给他们讲这些技术点，对于一个团队来说，这样的代价太大。为什么我们不能把这些有价值的技术点写到文档中呢？正是基于此考虑，在今年4月份的时候，我下定决心把我所知道的技术点，解决方案分类列出来，这就形成了本文档。

对于一个团队来说，知识总结是最重要的，我们必须把这些总结出来，形成文档，从而达到共享知识的目的，但是光总结知识是远远不够的，我们必须让这样的文档是专业的，准确的，只有这样，它才能更好的指导我们的工作，在一个专业和规范的文档的基础之上，我们做事才能得以规范，这就是通常所说的规范化，只有规范的团队，他们开发出来的产品质量才能更好。

在写本文档的过程中，我努力地做到让文档准确，有些内容来自Google的Android开发者官方文档，有的则是来自于网络，更多的是在项目过程中所积累下来的通用的解决方案。在本文档中，你能学习到如何正确处理位图的操作，Android应用程序的基本设计原则，同时也能知道Android应用程序的性能改善方案。这些技术都是从项目中总结出来的，它是经过验证与测试，因此，这些技术都应该是准确的。

本书读者

在本文档中，我尝试去解释为什么要这样做，而不直接告诉你怎么做，像如何利用XML来布局这样的问题，你在本文档中是找不到答案的，因为这些都是最基础的东西，我更多地阐述原理和通用的解决方案，可能不会涉及到具体的细节技术点，因此，本文档的读者应该是具有一定的Android开发基础，如果对Android是一无所知，那么你可能会对本文档很失望，不过没有关系，相信你在有一定的基础之后，再来阅读本文档，会收益颇多。

致谢

在写本文档的过程中，总是离不开同事与朋友的支持，感谢他们给我们提了很多宝贵的意见与建议。十分感谢郭凯在缓存设计与异步链式调用上给我的指导与帮助，有了他的思路我才得以顺利完成设计。

本文档所有大部分的知识都是在应用程序开发这一层，没有过多的涉及更加底层的技术，同时，还有些章节和内容需要完善与充实，这也是以后我的工作的一部分。一个人的能力，思维总是有限的，尽管我努力地做到准确，但由于时间仓促，错误总是无法避免，请读者见谅，同时也希望大家能给我提出宝贵的意见并参与进来，大家可以给我发邮件：

工作邮箱：lihong2@lzt.com.cn

私人邮箱：leehong2005@163.com

李洪

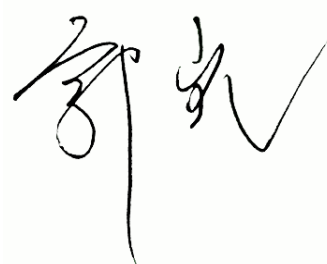
2012. 11. 09

序 言

产业的发展大体都有相似的通性：随着需求的扩大，对产品的要求越来越高，越来越大。产业的本身不再仅仅满足于生产，从而提出了一系列改变自身的方案，进行分工细化、任务明确、流程改进、整体创新等，最终促进生产力的提高乃至培育成新的产业。软件业的发展就是这样一个典型的例证。Android平台的开发也是一样的。

Android Coding Style正是在这样的通性下作成的。它总结了过去容易犯的错误，对一些容易混淆和误解的概念加以明确，对一些处理流程给出建议和策略。它是根据大量宝贵经验加以提炼的作品。

Android Coding Style的核心原则是改善开发效率，既然是改善就没有尽头，因为时间和技术的问题，Android Coding Style本身有许多未完善和不合理需要修正的地方请读者见谅。希望大家在查阅本书的时候，对于内容仔细研读思索，提出新的建议，不断完善它。成为我们的重要财富，帮助大家一起进步。



2012.11.5

目 录

1	Java Code Style	6
2	Android 命名规则	12
3	Android 设计	15
4	Android 工程结构	26
5	Android 界面	27
6	Android 程序性能与安全	42
7	Android 程序性能改善意见	59
8	Android 数据存储	67
9	Android Bitmap操作	80
10	Android 网络操作	96
11	Android 进程与线程	109
12	Android 四大组件	128
13	Android 多分辨率与多DPI	129
14	Android 程序发布	130
15	设计模式	131

1. Java Code Style

1.1 SVN代码管理

- 1, 尽量不要使用各种插件提交代码。
 - 容易漏掉新追加的文件。
 - 本地文件提交不完全。
- 2, 保持本地文件与SVN服务器的可控同步。
- 3, 上传到SVN的代码必须编译通过, 运行通过。

1.2 警告

- 1, Java代码中不允许出现在警告。
- 2, 无法消除的警告要用@SuppressWarnings。
- 3, 去掉无用的包名。
- 4, 确认一些没有用到的私有方法, 是否真的无用。

1.3 命名规则

• 基本原则

- 1, 变量, 方法, 类命名要表义, 严格禁止使用name1, name2等命名。
- 2, 命名不能太长, 适当使用简写或缩写。(最好不要超过25个字母)
- 3, 方法名以小写字母开始, 以后每个单词首字母大写。
- 4, 避免使用相似或者仅在大小写上有区别的名字。
- 5, 避免使用数字, 但可用2代替to, 用4代替for等, 如go2Jsp。

• 类、接口

- 1, 所有单词首字母都大写。使用能确切反应该、接口含义、功能等的词。一般采用名词。
- 2, 接口可带 I 前缀, 或able, ible, er等后缀。如ISerialable。
- 3, 不得将类直接定义在基本包下面。

• 字段

- 1, 常量必须程序完整的英文大写单词, 在词与词之前用下划线连接, 如public static final int DEFAULT_VALUE = 0;
- 2, 命名时应使用复数来表示它们代表多值, 如orderItems。
- 3, 代码中禁止使用硬编码, 把一些数字或字符串定义成常用量。
- 4, 静态字段以 s_ 作为前缀, 如s_instance。

• 缩写

Good	Bad
XmlHttpRequest	XMLHttpRequest
getCustomerId	getCustomerID
class Html	class HTML
String url	String URL
long id	long ID

1.4 注释

- 1, 版权声明, 从类文件第一行开始。

```
/*
 * System: Pascal
 * @version 1.00
 *
 * Copyright (C) 2012, TOSHIBA Corporation.
 *
 */
```

```
package com.toshiba.pascal.activity;    （与前面的注释之间空一行）
```

- 2, 类注释

```
/**
 * This class is used to create the alarm activity to show the alarms.
 *
 * @author XXX
 *
 * @date 2012/03/23
 */
```

- 3, 方法与成员变量注释

```
/**
 * The alarm information list.
 */
```

- 4, 使用TODO来标记临时的代码。

```
// TODO: Remove this code after the UriTable2 has been checked in.
```

示例代码如下:

```
/*
 * System: XXX
 * @version    1.00
 *
 * Copyright (C) 2011, TOSHIBA Corporation.
 *
 */

package com.toshiba.ebook.bookplace.util;

import android.util.Log;

/**
 * The Log Utility, we can control the output of log.
 *
 * @author XXX
 *
 * @date 2011/12/13
 */
public class LogUtil
{
    /**
     * This flag to indicate the log is enabled or disabled.
     */
    private static boolean s_isLogEnable = true;

    /**
     * Enable the log output.
     */
    public static void enableLog()
    {
        s_isLogEnable = true;
    }
}
```


1.5 表达式与语句

1, 判断中如果有常量, 则应将该常量前置。如 `if (null == user) ...`

2, 所有 `if` 语句必须用 `{}` 包括起来, 即便只有一句

```
if (true)                if 与 (之间有一个空格)
{
    // Do something
}
```

```
if (true)                禁止使用这种格式
    // Do something
```

3, 循环中必须有终止循环的条件或语句, 避免死循环。

4, 循环条件在每次的循环中会执行一行, 所以尽量避免在其中调用耗时或费资源的操作。

5, 在含有多种运算符的表达式中, 使用圆括号来避免运算符优先级问题。

```
if (a == b && c == d) // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

6, `switch`语句中, `break`与下一条`case`之间, 空一行。

1.6 错误与异常

1, 捕捉异常是为了处理它, 通常在异常`catch`块中输出异常信息。

```
try
{
}
catch (Exception e)
{
    e.printStackTrace()
}
finally
{
}
```

2, 资源释放的工作, 可以放到 `finally` 块部分去做。比如关闭数据库, 关闭`Cursor`, 断开`Http`连接等。

1.7 Class内部顺序

1, 每个`class`都应该按照一定的逻辑结构来排列基成员变量、方法、内部类等, 从而达到良好的可读性。

2, 总体上来说, 要按照先`public`, 后`protected`, 最后`private`, 函数的排布也应该有一个逻辑的先后顺序, 由重到轻。

3, 以下顺序可供参考:

- 定义TAG, 一般为`private` (可选)
- 定义`public`常量

- 定义private常量、内部类
- 定义private变量
- 定义public方法
- 定义protected方法
- 定义private方法

1.8 循环

循环的选择和使用规范：

- 1, 循环语句的选择: for/foreach/while/do...while 是否正确?
一般情况：计数循环用for 或者 foreach
条件循环或死循环或者复杂循环用：while/do...while
- 2, 循环边界检查是否正确？
- 3, 循环要尽可能的短，把长循环的内容抽取到方法中去.
- 4, 嵌套层数不应超过3层，要让循环清晰可读.
- 5, 循环条件是否清晰易懂？要尽量保持循环条件的简单性。

• 循环在编码时的注意点：

- 1, 在循环一个list或数组时，是否在循环中频繁使用一个其大小.

Don't

```
for (int ix = 0; ix < list.size(); ++ix)
```

Do

```
int size = list.size();  
for (int ix = 0; ix < size; ++ix) { }
```

- 2, 在不修改list中数据时，尽可能的使用 foreach 来代替普通的for循环.

```
int size = list.size();  
for (int ix = 0; ix < size; ++ix)  
{  
    // Do something  
}
```

这种方式可修改为（JAVA为for，C++/C#为 for each）

```
for (A a : lists)  
{  
    // Do something  
}
```

- 3, 在删除List元素时，应该注意要从后往前删除，而不是从前往后删除，后者容易出现Out of Index的错误.

比如要删除某个list中，符合某个condition的所有元素，

Don't

```
for (int ix = 0; ix < list.size(); )
{
    if (list.get(ix) == condition)
    {
        list.remove(ix);
    }
    else
    {
        ix++;
    }
}
```

Do

```
for (int ix = totalSize - 1; ix >=0; --ix)
{
    if (list.get(ix) == condition)
    {
        list.remove(ix);
    }
}
```

Event better

(先找到要删除的元素并添加到一个集合中，最后再把这个集合删除)

```
List<E> delList = new List<E>();
for (int ix = 0; i < size; ++ix)
{
    if (list.get(ix) == condition)
    {
        delList.add(list.get(ix);
    }
}

if (delList.size > 0)
{
    list.removeAll(delList);
}
```

2. Android 命名规则

我们在开发Android时，各种命名尽量符合Android的规则，在定义类，接口，资源名称时，多参考系统是如何定义的。这样当我们向第三方提供应用时，如果我们定义的Intent, Extra等不规范，容易造成误解或误用。关于规约部分，请参考“Java Code Style”章节。

其实核心就是要规范，怎么做才算是规范？

- 保持与Android的统一性与完整性
- 遵守Android应用程序的开发规则

条款1：资源命名

资源命名时，全部小写，单词之间用下划线隔开，根据模块 + 功能来命名。

下表列出了各种Icon的通用命名方式：

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

注意：

对于drawable，如果使用Resource#getDrawable(id)方法来加载Drawable，当这个资源id是R.drawable的第一个成员，在Android 2.3.x上，在某种情况下，可能会得到一个ColorDrawable，而不是BitmapDrawable，这确定是Android的一个Bug，Android 4.0.x已经修正这个问题。

条款2：字符串定义

Java代码中或布局XML中的android:text属性不允许直接定义字符串，显示在UI上的所有字符串必须定义到资源中(string.xml)中，这样做的好处有：

- 未定义在string.xml中的字符在Linux环境下会被编译器视为错误
- 可以轻松实现多语言
- 可以定义字符串数组或格式化的字符串

条款3: 自定义的异常类, 都以Exception结尾

条款4: Android对象命名规则

自定义的Intent Action, Extra等根据Android的规则定义。

Action:

```
static final String ACTION_XXX = "com.toshiba.intent.action.XXX";
```

Extra:

```
static final String EXTRA_XXX_X = "com.toshiba.intent.extra.XXX_X";
```

注意:

多参考Android源码, 多了解Android平台的命名规则。

条款5: 常量定义

1) 时间定义

```
public static final int TIME_MINUTE = 1 * 60; // 1 minute  
public static final int TIME_MINUTE = 1 * 60 * 1000;
```

2) 定义一组常量要区分两种模式:

- 组合常量或状态, 通常常量值为 0x00, 0x01, 0x02, 0x04... 的这种形式。
- 互斥常量或状态, 通常常量值为 0, 1, 2, 3... 的这种形式。

条款6: UI大小定义

对于UI中的View的大小定义, 需要注意以下几点:

- 把定义在Java代码中的View大小的值, 提取成一组常量, 统一定义在一个类里
- 把定义在XML布局中的View大小的值, 定义到dimension.xml中

这样做的好处有:

- 统一管理, 便于维护
- 避免Java或XML代码中出现硬编码
- UI大小的定义不会散落在代码各个地方
- 为以后对应多DPI, 多分辨提供方便, 在不同的DPI下面, 可能同一个布局的大小是不一样的, 如果我们将View的大小统一在一个地方, 那么只需要改动这一个地方就行了。

示例代码如下:

```
public static final int BOOK_VIEW_WIDTH = 200;  
public static final int BOOK_VIEW_HEIGHT = 300;
```

如果需要对应多分辨率或多DPI，只需要作如下修改：

```
public static final int BOOK_VIEW_WIDTH =  
    getValueById(R.dimen.book_width);  
public static final int BOOK_VIEW_HEIGHT =  
    getValueById(R.dimen.book_height);  
  
public int getValueById(int id) { ... }
```

通过调用getValueById()方法来从资源(dimension.xml)中取值。

使用这种方式就不需要修改Java代码，只需要修改定义的常量值就可以达到统一管理UI的目的。也可以很容易对应多分辨率或多DPI。

注意：

对于以上的建议，可能你会觉得开发会更加麻烦，因为要把很多常量提取到统一的文件中，但是，这样做会使项目的管理，维护变得更加容易，因此，我们建议采取这种方式来管理UI布局中View大小。

3. Android 设计

这一章节主要列出一些在开发Android应用程序中应当注意的事项，它更多是建议，按照这些事项来做，可能会对你有所帮助。

条款1: Activity中处理事件

通常情况下，把事件处理封装成一个方法，方法命名跟Android一致。如onXXXButtonClick(View v)，这样做的好处是，我们只需要关心事件处理，而不需要关心listener，而且从onXXX的方法命名，可读性也强，一看就知道是什么的事件处理。View相对应的listener里面，只需要调用这个onXXX方法就可以了。

多个View都需要注册View#OnClickListener时，所有的View都设置同一个listener，在#onClick()方法里面，根据Id来区分是哪个View响应了点击事件，这样做的好处有：

- 节省内存，不用创建多个listener实例
- 统一管理，方便维护

在Activity中处理事件的通常做法如下：

```
private View.OnClickListener m_listener = new View.OnClickListener()
{
    @Override                                     // View的listener相当于是事件分发
    public void onClick(View v)
    {
        switch (v.getId())
        {
            case R.id.main_albums_btn:
                onAlbumButtonClick(v);
                break;
        }
    }
};

private void onAlbumButton(View v)                // 把事件封装到方法中
{
    // Do something.
}
```

条款2：设计BaseActivity类

Android应用程序中，所有的Activity都应该有一个共同的基类BaseActivity，它继承于Activity，这样做的好处有：

- 定义共通的属性

可以把每个activity的共通属性定义在BaseActivity中。

比如可以把Activity切换动画效果定义在finish()，startActivity()方法中。

- Top activity

通过继承共同的基类，我们得到显示在最前面的界面，即Top Activity，这对于显示Dialog很有用，有时候我们在后台处理业务时，需要在任何界面都显示对话框，此时的对话框的Context就应该是Top Activity的。

BaseActivity的核心代码如下：

```
public class BaseActivity extends Activity
{
    protected void onResume()
    {
        Application app = this.getApplication();
        if (app instanceof PascalApplication) {
            ((PascalApplication)app).setTopActivity(this);
        }

        super.onResume();
    }

    @Override
    protected void onPause() {
        Application app = this.getApplication();

        if (app instanceof CustomApplication) {
            Activity topActivity =
                ((PascalApplication)app).getTopActivity();

            if (topActivity == this) {
                ((PascalApplication)app).setTopActivity(null);
            }
        }
        super.onPause();
    }

    public boolean isTopActivity() {
        return (this == ((PascalApplication)
            PascalApplication.getAppContext()).getTopActivity());
    }
}
```


条款3：设计LogUtil类

在Android应用程序开发过程中，有些地方只能通过打Log的方式来调试，如Touch事件等，使用Android提供的Log类，它无法打印出调用方法，类，所在行数等信息。我们可以设计一个LogUtil类，它可以输出调用方法名称，类名，行数等信息。

注意：

所有打Log的地方应当使用LogUtil类来代替Log类。

LogUtil类的核心实现如下：

```
package com.toshiba.ebook.bookplace.util;

import android.util.Log;

public class LogUtil
{
    private static boolean s_isLogEnable = true;

    public static void d(String tag, String msg)
    {
        if (s_isLogEnable)
        {
            // Call the method of Log class directory.
            StackTraceElement stackTrace =
                java.lang.Thread.currentThread().getStackTrace()[3];
            String fileInfo = stackTrace.getFileName() + "(" +
                stackTrace.getLineNumber() + ") " +
                stackTrace.getMethodName();

            Log.d(tag, fileInfo + ": " + msg);
        }
    }
}
```

条款4：设计Application类

Android应用程序一般需要提供提供一个Application类，这样做的好处有：

- 存储全局简单数据，如Top Activity，App Content等
- 初始化全局数据，运行时等

AndroidManifest.xml中这样使用：

```
<application android:icon="@drawable/melonpanicon"
    android:label="@string/app_name"
    android:name=".app.CustomeApplication" >
    ...
</application>
```

Application类的核心实现如下：

```
public class CustomApplication extends Application {

    private static Context m_appContext = null;           // App context
    private Activity m_topActivity = null;               // top activity

    public static Context getAppContext() {

        return m_appContext;
    }

    public void setTopActivity(Activity topActivity) {

        m_topActivity = topActivity;
    }

    public Activity getTopActivity() {

        return m_topActivity;
    }

    public void onCreate() {

        super.onCreate();
    }

}
```

条款5：异常的捕捉处理

异常是Java语言不要分割的一部分，它可以使你在某处集中精力处理你要解决的问题，而在另一处处理代码中产生的错误。其中“报告”功能是异常的精髓所在。

在使用异常时，应当注意以下几点：

- 在catch块中处理异常，打印异常信息或重新抛出
- 在finally块中执行清理操作，如网络连接，文件，数据库等
- 不要使用try-catch来处理业务逻辑，因为异常可能会中断业务的执行
- 不要在finally块中使用return，这会导致异常丢失
- 不要通过异常来判断业务执行成功与否，异常会消耗很多系统资源
- 考虑设计自定义的异常

下面代码示例了try-catch-finally的使用方法

```
public void Fun() throws Exception {  
  
    try {  
  
        // Do something  
    }  
    catch (Exception e) {  
  
        e.printStackTrace();    // Print the stack info  
        throw e;                // Throw the exception  
    }  
    finally {  
  
        // Release something  
    }  
}
```

条款6: @Override标注

子类重写基类的方法时，必须加上@Override标注，这样做的好处有：

- 使程序更健壮，通过编译器使程序更加安全，使得方法签名必须与基类一致
- 在子类中能清晰知道该方法是属于基类还是子类，对于代码理解有帮助

条款7: API的兼容性

在一个API不可避免要消亡或者改变的时候，我们应该接受并且面对这个事实，下面列举了几种保证兼容性的前提下，对API进行调整的办法：

- 将API标记为弃用(使用@Deprecated标注)，重新添加一个新的API
- 为API添加额外的参数或者参数选项来实现功能添加
- 将现有API拆成两部分，提供一个精简的核心API，过去的API通过封装核心API实现
- 在现有的API基础上进行封装，提供一个功能更丰富的包或者类

在设计的过程中，如果能按照下面的方式进行设计，会让这个API生命更长久：

- 面向用例的设计，收集用户建议，把自己模拟成用户，保证API设计易用和合理
- 保证后续的需求可以通过扩展的形式完成
- 第一版做尽量少的内容，尽量少做事性是抑止API设计出错的一个有效方案
- 对外提供清晰的API和文档规范，避免用户错误地使用API

除此之外，下面还列出了一些具体的设计方法：

- 方法优于属性
- 工厂方法优于构造
- 避免过多的继承
- 避免由于优化或者复用代码影响API
- 面向接口编程
- 对组件进行合理定位，确定暴露多少接口
- API尽可能少，如果一个API可以提供也可以不提供，那就暂时不要提供
- 一致性，是否与系统中其他模块的接口风格一致
- 简单明了，API应该容易理解，容易学习和使用的API才不容易被误用

条款8：注册与反注册

我们在设计一个模块的过程中，如果这个模块需要提供注册功能，通常需要注意下面几点：

- 提供Register功能，它使得可以与其他模块进行通信
- 提供Unregister功能，可以断开与其他模块的通信

注意：

通过Register/Unregister功能是成对出现(例如Broadcast Receiver)，如果只有Register，没有Unregister的话，之前注册的对象有可能无法从当前模块中清除。

条款9：一些编写代码的建议

下面列出了一些编写代码的建议，按照这些建议来做，可能会使你的代码更加清晰合理。

- 方法实现尽量控制在50行内，并且做到职责单一
- 共通的方法，尽可能提到Util类中，不要把相同的实现散落在各个地方
- 避免一些方法做相同的事情，但却有不同的方法名，应当保持统一性
- 如果很多数据要作为一个类的成员变量，那么，这些数据其实应该独立成一个类
- 定义方法，成员时，一定要多考虑其访问权限

条款10：关于Adapter

Adapter是数据(Data)与界面(UI)之间的桥梁，通过它能把数据显示在界面上(ListView)。关于Adapter的使用，需要注意以下几点：

- getView()

在Adapter#getView()方法中不要每次调用View#findViewById()方法，该方法会有效率问题，特别是当View层次复杂时，对于简单的View，实现一个ViewHolder类。

```
private class SongItemViewHolder
{
    TextView m_songNumber    = null;
    TextView m_songName      = null;
    TextView m_songDuration  = null;
    ImageView m_playingIcon  = null;
}
```

- 不要在Adapter#getView()方法中，每次返回一个新创建的View对象

- 不要在后台更改数据

在后台线程更改了数据，如果这些数据是绑定到Adapter上面的话，此时，如果UI发生Layout，那么它会调用Adapter#getView()方法，此时就可能会导致异常，错误是在后台线程更改了数据，但没有通知界面。

解决方法：

- 让Adapter依赖的数据是全局数据的一个子集
- 请参考系统Gallery3d, Music等程序，它们的设计都是如此

更多细节，请参考“Android 程序性能与安全”章节。

条款11：注意对象生命周期

在Android程序中如果用到的单实例的类，一定要注意：这个类的生命周期与application一样。一定不要让这个类引用比自己生命周期还短的类，比如Activity的Context, Listener等。这样可能会导致这些Context, Listener不能被GC回收，从而导致内存泄漏。

条款12：反射

在使用反射技术调用方法时，会传递参数，如果反射的方法没有参数，可以传null，但是在Linux环境下编译，会出现编译错误，错误是无法从null转换成Object[]，所以我们需要手动强转。如下面代码红色部分所示：

```
Class<Environment> c = (Class<Environment>)
                        Class.forName("android.os.Environment");
if (null != c)
{
    Method[] methods = c.getDeclaredMethods();
    Method method = c.getDeclaredMethod(
        "getExternalStorageDirectories",
        (Class<?>[])null);

    if (null != method)
    {
        File[] args = (File[]) method.invoke(
            null,
            (Object[])null);
    }
}
```

条款13：关于Broadcast Receiver

广播是Android的组件之一，它分为全局广播和局部广播。

- **全局广播：**

定义在Manifest.xml文件中，当设备重启或开机时，系统就会初始化这个广播实例。也就是说，当设备一开机，你的这个App进程就已经被运行起来了，只是没有启动任何一个Activity（UI）而已。

这种广播通常是应用于整个程序，不管Activity是否显示，都需要接收的广播消息，如亮度变化等。

- **内部广播：**

注册与反注册在Java代码中实现。

通常情况下，只是一个Activity需要关注这个广播，当Activity启动时，需要接收广播，当Activity退出后，不需要监听广播，如显示电池电量的UI，当这个UI显示时，需要接收电池电量变化的广播，当UI关闭时，不需要接收。

条款14: 单实例 (Singleton)

我们在开发Android程序过程中，通常会设计一个BL层，用来处理业务逻辑，这一层里面，通常会存数据在内存中，而通常情况下，这些BL类可能是单实例类，这样就可以在不同的模块里面共享数据。

```
public class MyBL {  
  
    protected static MyBL s_myBL = null;  
  
    public static synchronized MyBL getInstance() {  
  
        if (null == s_myBL) {  
            s_myBL = new MyBL(context);  
        }  
  
        return s_myBL;  
    }  
  
    public static synchronized void releaseInstance() {  
        s_myBL = null;  
    }  
  
    private MyBL() {  
  
    }  
  
}
```

上面代码示例了一个单项实例的BL类，它的构造方法是称有的，通过MyBL#getInstance()方法来取得MyBL类的实例，通过MyBL#releaseInstance()方法来把该实例设置为null。

注意:

由于是单项实例，MyBL的实例是一个静态(static)的对象，它的生命周期与Android Application的周期一样长，也就是说，即使你的Main activity都退出了，这些静态对象仍然存在，所以这里一定要注意，如果你的Main activity退出后，需要调用MyBL#releaseInstance()方法来把这些静态对象设置为null，这样GC是可能回收这些对象的。为什么要这样做呢？是因为Main activity已经退出，对于用户来说，他们认为程序已经关闭，那么内存中的数据其实是不应该有缓存的。

条款15: 封装Bitmap操作

在开发过程中，有可能我们会用到很多Bitmap，可能会涉及了从Bitmap的加载，保存等，我们应该把这些操作都封装起来，因为Bitmap的加载会涉及decode，它会分配大量内存，而这些操作完全有可能导致OOM异常，所以，我们应该把这些操作封装到一个或几个类中，从而能对其进行更好的控制。

通常我们这样设计：

- BitmapUtil
 - save
 - load
 - compute sample size
 - byte to bitmap
 - bitmap to byte
 - decode bitmap from network
 - ...

4. Android 工程结构

条款1: 包名分类

关于包名的分类，我们的建议有以下几点：

- 包名结构通常是：com.company.project.xxx
- 按照类型来分类，而不是功能，因为功能可能会随着需要变化而变化
- 可以把独立的功能，分成独立的包来管理，如云服务，可以专门建一个cloud的包

注意：

这里的建议不是绝对，需要根据具体情况而定，但我们建议尽量遵守这样的基本原则。

条款2: Android工程的包名分类

下面列出了Android工程的包名分类，它是按照类型来类：

com.toshiba.xxx	主包名，下面通常不放任何类。
com.toshiba.xxx.activity	这里面放工程所需要的activity类。
com.toshiba.xxx.app	这里面放实现的Application类。
com.toshiba.xxx.constant	这里面放定义的常量的类。
com.toshiba.xxx.data	这里面放数据存储，操作的类，通常是放数据库操作类。
com.toshiba.xxx.receiver	这里面放实现的各种Broadcast Receiver。（可选）
com.toshiba.xxx.provider	这里面放实现的各种Content Provider。（可选）
com.toshiba.xxx.service	这里面放实现的Service类。（可选）
com.toshiba.xxx.utils	这里面放各种Utility类。
com.toshiba.xxx.widget	这里面放自定义的widget类，如TosImageButton等。
com.toshiba.xxx.interfaces	这里面放定义的接口，如各种Listener等。
com.toshiba.xxx.entity	这里面放定义的实体类，就是各种数据结构类。
com.toshiba.xxx.bl	这里面放定义的业务逻辑类，如BookBL。
...	

条款3: 引用库

引用的第三方的库，都放到libs文件夹下面，在这里面，可以再新建不同的目录来管理不同的库。

条款4: 本地库

需要的so库，需要放到libs\armeabi\文件夹下面。

注意：

本地库so存放的目录必须是libs\armeabi\，否则运行时会找不到库而发生异常。

5. Android 界面

用户界面是你的应用程序最核心一部分，它直接与用户打交道，界面设计的好坏直接影响到你的应用程序的质量，因此，Android应用界面的设计是最重要的一部份之一。

在设计或实现界面时，我们一般应该考虑以下几点：

- 实现更加有效的导航
- 对应多分辨率与多屏幕
- 提供布局的性能
- 向“菜单”按钮说再见

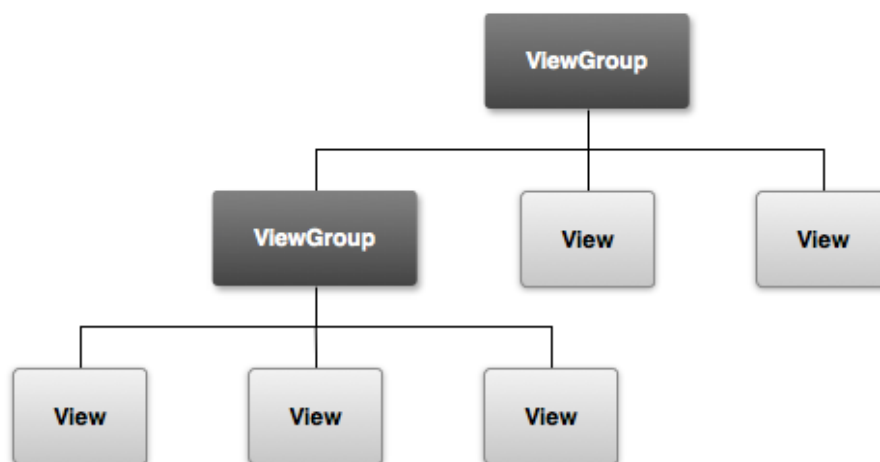
关于UI设计，可以参考Android的官方网站：

<http://developer.android.com/design/index.html>

5.1 UI概述

所有的用户界面上的控件都是从View或ViewGroup继承而来，View是所有控件的基类，ViewGroup是所有容器控件的基类，它能放置View或ViewGroup类型的控件。Android Framework提供了很多控件(Widget)，比如TextView, Button, LinearLayout, GridView等。

下图说明了UI的层次结构，所有的View都是放在一个ViewGroup中的，也就是说，在这个层次结构中，最顶层的是ViewGroup。



从上图中我们可以看到，最顶层的控件是一个ViewGroup，通常都是ViewGroup的派生类。

注意：

这里是View的层次结构图，并不是继承关系图，所有的控件的基类都是View。

我们可以在Java代码中或通过XML来实现一个界面，通常复杂的界面我们用XML来布局，因为XML更加简单明了，更加高效。比如，一个TextView和一个Button上下排列的布局的XML代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a Button" />
</LinearLayout>
```

当你加载这个布局资源后，系统会解析这个XML布局，并且实例化这些View，然后你可以通过 `findViewById(int resId)` 方法来得到各种View的实例。比如要得到Button的实例，示例代码如下：

```
setContentView(R.layout.main);
Button btn = (Button) findViewById(R.id.button);
btn.setText("This is a button");
```

5.2 布局

一个布局定义了用户界面的基础结构，我们有两种方式来实现布局：

- 在XML中定义UI元素
- 在运行时实例化UI元素

尽管Android framework提供了两种方式布局，但是通常首选XML布局，有如下优点：

- 界面与后台的逻辑处理分离，你可以修改界面，而不用修改源代码。
- 创建XML布局，适配多分辨率与多屏幕。
- XML布局可以做到可视化，布局更加简单、容易。

5.2.1 创建XML布局文件

每一个布局文件必须包含一个根元素(与XML结构一致)，它们必须是View或ViewGroup对象，如果根元素是ViewGroup，你可以定义其子元素。

布局文件的扩展名是.xml，它们放在Android工程的/res/layout/文件夹下面。

5.2.2 加载XML布局资源

我们一般是在Activity.onCreate()方法中加载XML布局资源，调用setContentView()方法，参数是资源文件的id，其格式是R.layout.layout_file_name，比如，如果XML布局文件名是main_layout.xml的话，那么加载资源的代码示例如下：

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main_layout);  
}
```

5.2.3 属性 (Attributes)

每一个View和ViewGroup对象都支持它们自己的XML属性，每一个View都会有自己独有的属性，有一些是公用的，如layout_width, layout_height等是公用的属性，而textSize属性是TextView的属性。

我们可以看源码，在View的构造方法里会去解析各种属性，从而设置到View对象的实例中。我们都可以到Google开发文档上面查阅View的属性，android.R.attr包下面就列出所有的属性。

这里，我列出一些常用到的布局属性

```
android:layout_width  
android:layout_height  
android:layout_gravity  
android:gravity  
android:layout_alignParentBottom  
android:layout_alignParentLeft  
android:layout_alignParentRight  
android:layout_alignParentTop  
android:layout_centerHorizontal  
android:layout_centerInParent  
android:layout_centerVertical  
android:layout_margin  
android:layout_toLeftOf  
android:layout_toRightOf  
android:layout_weight
```

5.2.4 ID

在View的层次树状结构中，每一个View对象都可以有一个唯一的id，用来标识View。当资源被编译后，它们的id是一个整型类型的数。定义View id的语法如下：

```
android:id="@+id/my_button"
```

注意：

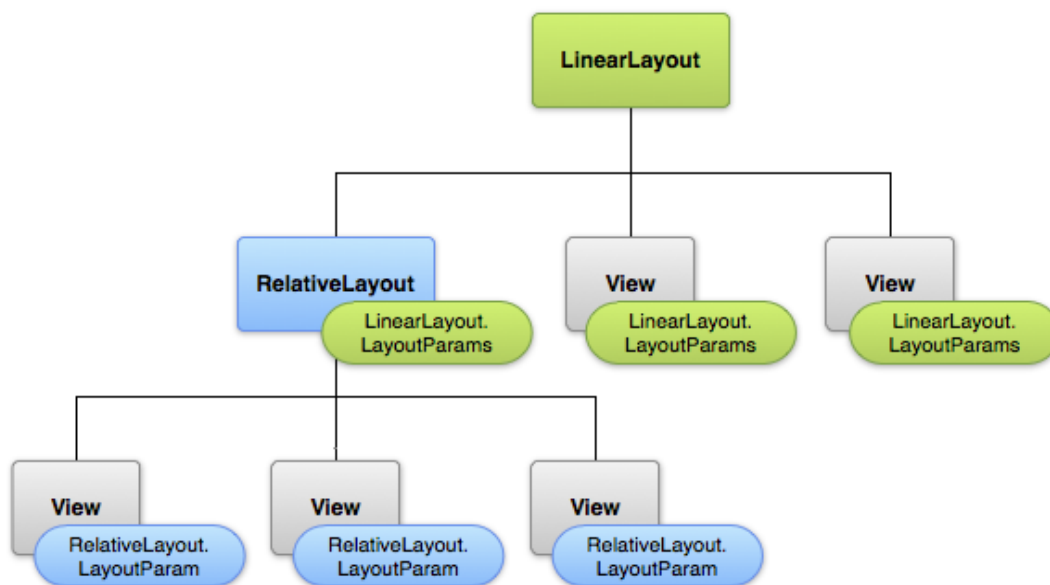
这里的加号(+)表示该id是一个新资源名称，它必须创建并添加到R.java文件中。如果你需要引用一个已经存在的id，@与id之间不需要用加号(+)连接。

5.2.5 Layout Parameters

在Android中，所有的View都需要用LayoutParameter来指定其布局的参数，不同的容器控件都有自己的布局参数，比如RelativeLayout，它的LayoutParams的类型是RelativeLayout.LayoutParams，它定义了View的宽，高，边距等参数，LinearLayout的布局参数的类型是LinearLayout.LayoutParams。不同的容器控件，定义了不同的布局参数。

所有的布局参数的共同基类是ViewGroup.LayoutParams。

下图示例了布局层次结构中的布局参数



注意：

从上图可以看出，每一个View的布局参数的类型必须是父容器的布局参数类型，也就是说，如果你把一个View放在RelativeLayout中，那么这个View的布局参数类型就是RelativeLayout.LayoutParams。如果不一致，就会抛出异常。

5.2.6 LinearLayout

线性布局是比较常用的一种布局，它使其child可以水平或垂直排列。

其布局形式如下图所示：



图3，LinearLayout布局形式(Horizontal)

LinearLayout有一些比较重要的属性：

- **android:orientation**

vertical 垂直布局
horizontal 水平布局

- **android:layout_weight**

该属性默认值为0，它可以指定每一个控件在LinearLayout中所占在比例大小。如果想让LinearLayout中的View平均分配LinearLayout的空间，我们可以把每一个View的高度(vertical layout)或宽度(horizontal layout)设置为"0dp"，然后将每一个View的layout_weight设置为"1"。

示例

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:hint="To" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="Subject" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="Send" />
</LinearLayout>
```

上述XML布局运行的UI如右图所示，图中的第三个EditText占满了LinearLayout剩余空间。

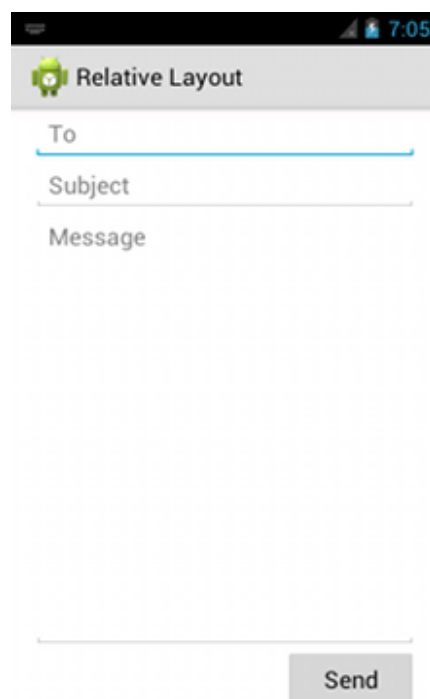


图4，layout_weight属性示例

5.2.7 自定义控件

我们可以通过扩展View及其派生类来自定义控件。

- 定义类

```
public static class MyEditText extends EditText { ... }
```

- 初始化

先调用super的构造方法。

- 重写方法

```
onDraw(), onSizeChanged(), onTouchEvent()...
```

- 使用

```
<com.android.notepad.MyEditText  
    id="@+id/note"  
    ... />
```

下表列出了Android framework调用的一些标准方法

Category	Methods	Description
Creation	Constructors	There is a form of the constructor that are called when the view is created from code and a form that is called when the view is inflated from a layout file. The second form should parse and apply any attributes defined in the layout file.
	onFinishInflate()	Called after a view and all of its children has been inflated from XML.
Layout	onMeasure(int, int)	Called to determine the size requirements for this view and all of its children.
	onLayout(boolean, int, int, int, int)	Called when this view should assign a size and position to all of its children.
	onSizeChanged(int, int, int, int)	Called when the size of this view has changed.
Drawing	onDraw(Canvas)	Called when the view should render its content.

Event processing	onKeyDown(int, KeyEvent)	Called when a new key event occurs.
	onKeyUp(int, KeyEvent)	Called when a key up event occurs.
	onTrackballEvent(MotionEvent)	Called when a trackball motion event occurs.
	onTouchEvent(MotionEvent)	Called when a touch screen motion event occurs.
Focus	onFocusChanged(boolean, int, Rect)	Called when the view gains or loses focus.
	onWindowFocusChanged(boolean)	Called when the window containing the view gains or loses focus.
Attaching	onAttachedToWindow()	Called when the view is attached to a window.
	onDetachedFromWindow()	Called when the view is detached from its window.
	onWindowVisibilityChanged(int)	Called when the visibility of the window containing the view has changed.

5.3 输入事件 (Input Events)

5.3.1 Event Listeners

我们可以注册各种listener来监听用户输入事件，这些回调方法有：

- **onClick()**
来自View.OnClickListener，当用户点击一个View的时候调用。通过调用View.setOnClickListener()方法来设置该接口对象。
- **onLongClick()**
来自View.OnLongClickListener，当用户长按一个View的时候调用。通过调用View.setOnLongClickListener()方法来设置该接口对象。
- **onFocusChange()**
来自View.OnFocusChangeListener，当View获得或失去焦点时被调用。
- **onKey()**
来自View.OnKeyListener，当用户按下或释放硬件按钮时，获得焦点的View会被调用该回调方法。
- **onTouch()**
来自View.OnTouchListener，当用户执行触摸时被调用。
- **onCreateContextMenu()**
来自View.OnCreateContextMenuListener，当创建菜单时被调用。

这些方法都是通过调用View.setXXXListener()方法来调用接口对象。

5.3.2 Event Handlers

我们也可以扩展View，重写onTouchEvent()方法，处理ACTION_DOWN，ACTION_MOVE，ACTION_UP，ACTION_CANCEL等事件消息。

5.4 事件路由机制

这一节大概讲述一下Android GUI的事件路由机制，我们在开发过程中，有可能要处理一些特殊的事件，如果不清楚其事件的分发机制，可能会带来很多的迷惑性，因此我们很有必要搞明白Android上面事件分发的原理。

先来看一下下面几个问题：

- 为什么点击一个button，事件不会分发到这个button下面的view上面？
- 为什么LinearLayout不会响应click事件，而button会响应？
- 在一个View上面按下后，然后移出该View，Move事件会分发给哪个View？

先看一看下图所示的布局：

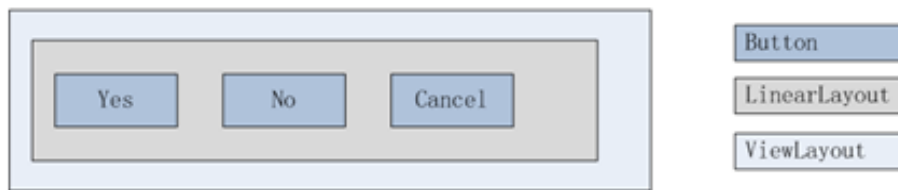


图5，UI布局示意图

上图中，三个Button放在一个LinearLayout之中，这个LinearLayout又放在一个ViewGroup中，相当于有三层。当用户点击“Yes”按钮时，ACTION_DOWN是如何传递的呢？首先，这个被点击的Button是最先有机会处理它，如果它没有处理，就交给它的直接父视图去处理，如果还没有处理，事件再次传递给上层父视图，直到达到最顶层的根视图。

路由示例说明

1、查找目标(target view)

- 判断当前view是否截断了ACTION_DOWN事件。
- 将事件再发分到该child，直到找到一个可以处理ACTION_DOWN事件的view。
- 如果view group没有child，它就会调用super.dispatchTouchEvent()方法，这样就有机会让view group来处理事件了。

这次是从上到下一层一层的递归调用，直接最终发现一个处理了ACTION_DOWN的view。也就是在ACTION_DOWN的时候返回true。在ViewGroup中有一个mMotionTarget的变量，用来保存响应事件的View。

下图示例了查找target view的流程图：

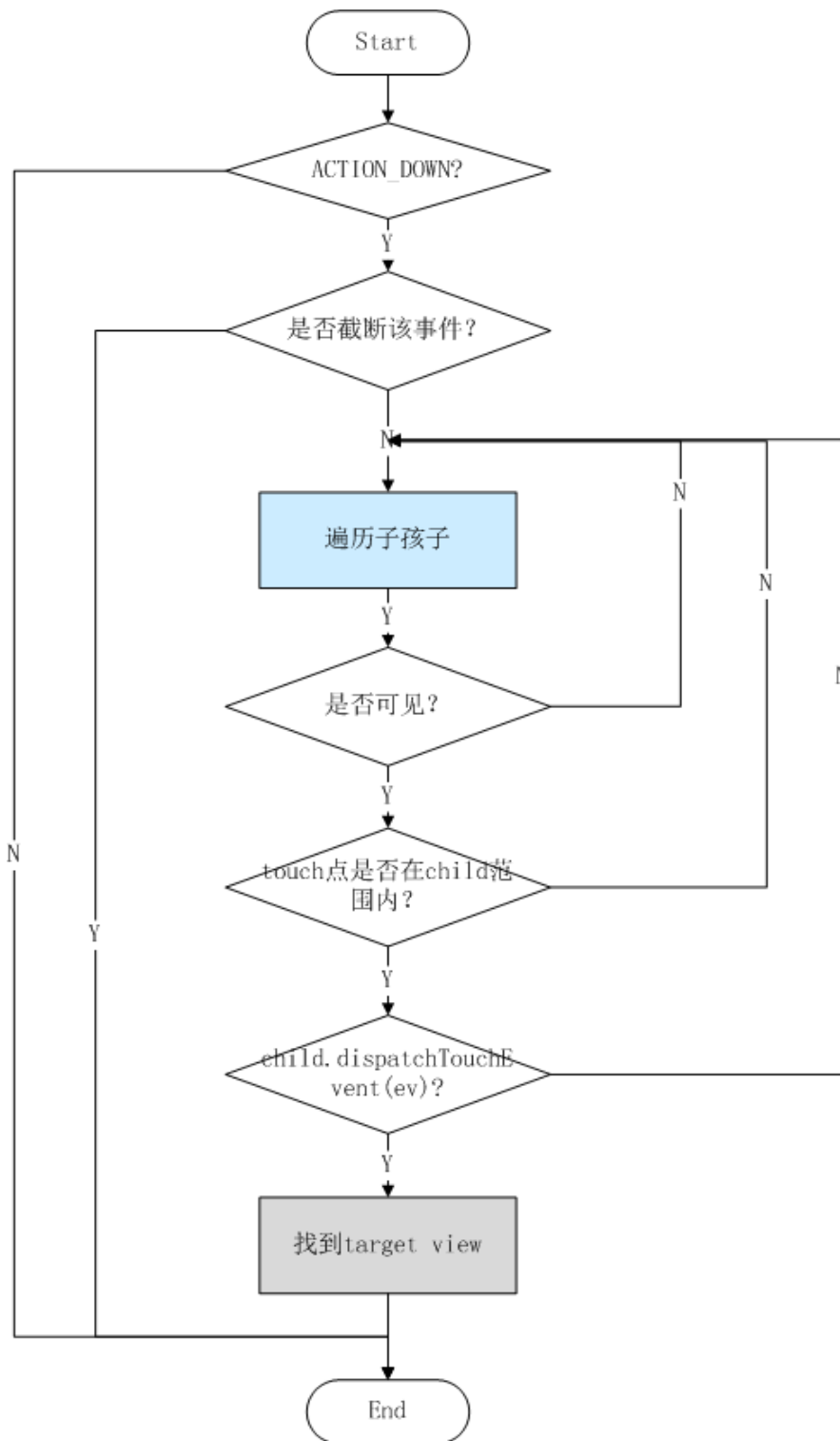


图6，事件分发流程图

2、向target view分发事件

当找到target view后，后续的事件(move, up)就会直接将事件分发到target view上面，注意，在每一层的view group里面都维护了一个target view，依次从上到下将事件分发，直到target view为空。

ViewGroup.dispatchTouchEvent() 方法实现代码如下：

```
/**
 * {@inheritDoc}
 */
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (!onFilterTouchEventForSecurity(ev)) {
        return false;
    }
    final int action = ev.getAction();
    final float xf = ev.getX();
    final float yf = ev.getY();
    final float scrolledXFloat = xf + mScrollX;
    final float scrolledYFloat = yf + mScrollY;
    final Rect frame = mTempRect;
    boolean disallowIntercept =
        (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
    if (action == MotionEvent.ACTION_DOWN) {
        if (mMotionTarget != null) {
            // this is weird, we got a pen down, but we thought it was
            // already down!
            // XXX: We should probably send an ACTION_UP to the current
            // target.
            mMotionTarget = null;
        }
        // If we're disallowing intercept or if we're allowing and we didn't
        // intercept
        if (disallowIntercept || !onInterceptTouchEvent(ev)) {
            // reset this event's action (just to protect ourselves)
            ev.setAction(MotionEvent.ACTION_DOWN);
            // We know we want to dispatch the event down, find a child
            // who can handle it, start with the front-most child.
            final int scrolledXInt = (int) scrolledXFloat;
            final int scrolledYInt = (int) scrolledYFloat;
```

```

final View[] children = mChildren;
final int count = mChildrenCount;
for (int i = count - 1; i >= 0; i--) {    // 从后向前遍历
    final View child = children[i];
    if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE
        || child.getAnimation() != null) {
        child.getHitRect(frame);
        if (frame.contains(scrolledXInt, scrolledYInt)) {
            // offset the event to the view's coordinate system
            final float xc = scrolledXFloat - child.mLeft;
            final float yc = scrolledYFloat - child.mTop;
            ev.setLocation(xc, yc);
            child.mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
            if (child.dispatchTouchEvent(ev)) {
                // Event handled, we have a target now.
                mMotionTarget = child;
                return true;
            }
            // The event didn't get handled, try the next view.
            // Don't reset the event's location, it's not
            // necessary here.
        }
    }
}

boolean isUpOrCancel = (action == MotionEvent.ACTION_UP) ||
    (action == MotionEvent.ACTION_CANCEL);
if (isUpOrCancel) {
    // Note, we've already copied the previous state to our local
    // variable, so this takes effect on the next event
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}
// The event wasn't an ACTION_DOWN, dispatch it to our target if
// we have one.
final View target = mMotionTarget;
if (target == null) {
    // We don't have a target, this means we're handling the
    // event as a regular view.
    ev.setLocation(xf, yf);
    if ((mPrivateFlags & CANCEL_NEXT_UP_EVENT) != 0) {
        ev.setAction(MotionEvent.ACTION_CANCEL);
    }
}

```

```
        mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
    }
    return super.dispatchTouchEvent(ev);
}

// if have a target, see if we're allowed to and want to intercept its
// events
if (!DisallowIntercept && onInterceptTouchEvent(ev)) {
    final float xc = scrolledXFloat - (float) target.mLeft;
    final float yc = scrolledYFloat - (float) target.mTop;
    mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
    ev.setAction(MotionEvent.ACTION_CANCEL);
    ev.setLocation(xc, yc);
    if (!target.dispatchTouchEvent(ev)) {
        // target didn't handle ACTION_CANCEL. not much we can do
        // but they should have.
    }
    // clear the target
    mMotionTarget = null;
    // Don't dispatch this event to our own view, because we already
    // saw it when intercepting; we just want to give the following
    // event to the normal onTouchEvent().
    return true;
}
if (isUpOrCancel) {
    mMotionTarget = null;
}
// finally offset the event to the target's coordinate system and
// dispatch the event.
final float xc = scrolledXFloat - (float) target.mLeft;
final float yc = scrolledYFloat - (float) target.mTop;
ev.setLocation(xc, yc);
if ((target.mPrivateFlags & CANCEL_NEXT_UP_EVENT) != 0) {
    ev.setAction(MotionEvent.ACTION_CANCEL);
    target.mPrivateFlags &= ~CANCEL_NEXT_UP_EVENT;
    mMotionTarget = null;
}
return target.dispatchTouchEvent(ev);
}
```


`View.dispatchTouchEvent()` 方法实现代码如下:

```
/**
 * Pass the touch screen motion event down to the target view, or this
 * view if it is the target.
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
public boolean dispatchTouchEvent(MotionEvent event) {
    if (!onFilterTouchEventForSecurity(event)) {
        return false;
    }
    if (mOnTouchListener != null &&
        (mViewFlags & ENABLED_MASK) == ENABLED &&
        mOnTouchListener.onTouch(this, event)) {
        return true;
    }
    return onTouchEvent(event);
}
```

6. Android 性能与安全

这一章部分请参考“Android程序性能改善意见”章节。

6.1 提高Android程序性能

6.1.1 Adapters

Adapter是Android程序中一个重要的部分，它是数据(Data)和界面(UI)之间的纽带，Adapter, Data, UI之间的关系如下图所示：

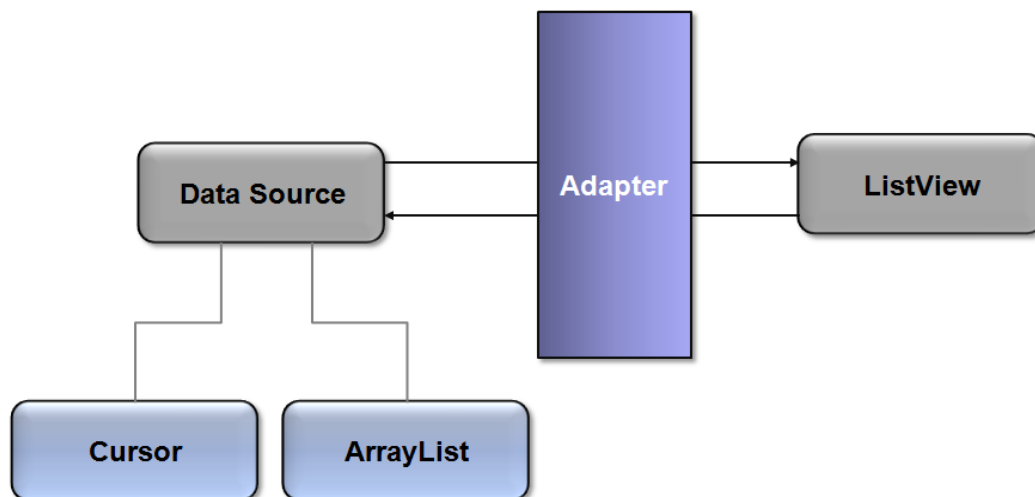
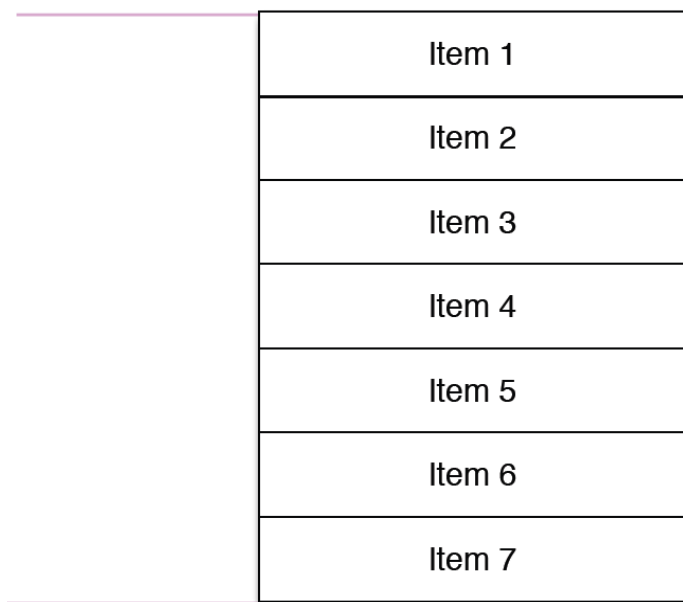


图1, Adapter, Data, UI之间的关系 (Man in the middle)

从上图中可以看出，Adapter是数据与UI之间的联系。

- a) 对于ListView的每一个子View，都是调用Adapter.getView()。
- b) 创建一个新的View，很费时间，性能代价很大。
- c) 如果ListView有100,000项，怎么办？



Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7

图2, ListView界面的数据

假如一个ListView的数据如上图所示。我们可以有以下三种方式来实现这个ListView。

Don't

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View item = mInflater.inflate(R.layout.list_item_icon_text, null);
    ((TextView) item.findViewById(R.id.text)).setText(DATA[position]);
    ((ImageView) item.findViewById(R.id.icon)).setImageBitmap(
        (position & 1) == 1 ? mIcon1 : mIcon2);

    return item;
}
```

上面的代码，在getView()方法的实现中，每次都是从一个XML布局中去加载一个View，这样做性能比较差，因为从一个XML中去加载一个View，性能开销比较大，而且如果这个XML布局特别复杂的话，那么性能更差。

Do

```
public View getView(int position, View convertView, ViewGroup parent)
{
    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.item, null);
    }

    ((TextView) convertView.findViewById(R.id.text)).
        setText(DATA[position]);
    ((ImageView) convertView.findViewById(R.id.icon)).
        setBitmap((position & 1) == 1 ? mIcon1 : mIcon2);

    return convertView;
}
```

上面的代码，复用了参数convertView，当convertView为null时，才从XML中加载布局。

ListView加载View时，会把这些View保存起来，当用户上下滑动时，它会把缓存的View通过参数(convertView)传给Adapter。但是这种方法还不是最好的，虽然利用了convertView，但是仍然会每次调用findViewById()方法从View中查找某一个View，这里也是有性能开销的。

Even better

```
static class ViewHolder
{
    TextView text;
    ImageView icon;
}
```

提供一个ViewHolder，它用来缓存加载出来的视图。

```
public View getView(int position, View convertView, ViewGroup parent)
{
    ViewHolder holder;

    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.list_item_icon_text,
                                         null);

        holder = new ViewHolder();
        holder.text = (TextView) convertView.findViewById(R.id.text);
        holder.icon = (ImageView) convertView.findViewById(R.id.icon);

        convertView.setTag(holder);
    }
    else {
        holder = (ViewHolder) convertView.getTag();
    }

    holder.text.setText(DATA[position]);
    holder.icon.setImageBitmap((position & 1) == 1 ? mIcon1 : mIcon2);

    return convertView;
}
```

这里提供的ViewHolder作为convertView的TAG，里面缓存界面的对象（ImageView，TextView）。当convertView不为空时，直接从ViewHolder中去取对象，而不是再去从convertView调用findViewById去查找。

下图显示了以上三种方式的效率对比。

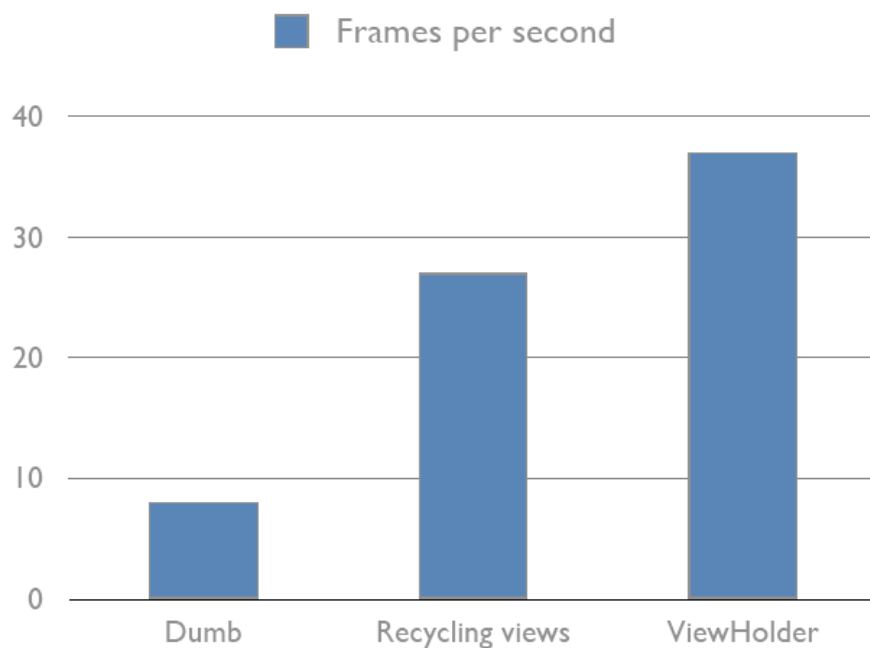


图3, 不同方式的效率对比

6.1.2 背景和图片

- a) 背景图片的drawables总是适应Views, 有可能会发生拉伸 (Stretching)。
- b) 运行时的缩放(Scaling)性能开销很大。
- c) 怎么办?

对于背景图片, 我们可以预先进行缩放。

```
// Rescales originalImage to the size of view using
// bitmap filtering for better results
Bitmap originalImage = Bitmap.createScaledBitmap(
    originalImage, // bitmap to resize
    view.getWidth(), // new width
    view.getHeight(), // new height
    true); // bilinear filtering
```

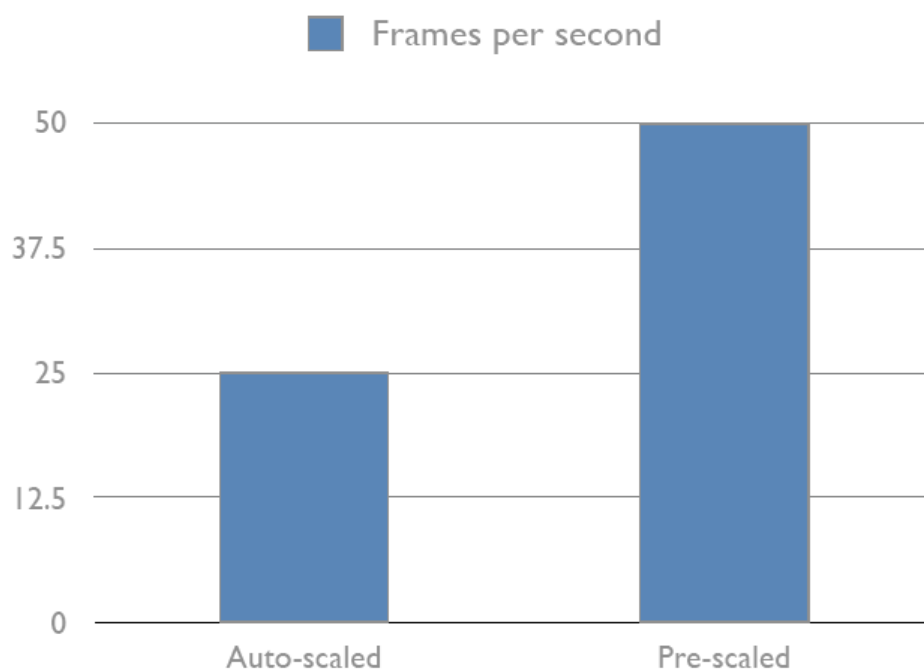


图4，自动缩放与预先缩放的性能对比

去掉Activity的背景

```
<!-- res/values/styles.xml -->
<resources>
    <style name="Theme.NoBackground" parent="android:Theme">
        <item name="android:windowBackground">@null</item>
    </style>
</resources>

<activity
    android:name="MyApplication"
    android:theme="@style/NoBackgroundTheme">
    <!-- intent filters and stuff -->
</activity>
```

一个Activity有背景与没有背景，性能差别小，没有背景比有背景性能稍好一些。

6.1.3 绘制和更新

- a) `invalidate()`
 - 使用简单
 - 性能开销大
- b) 无效区域
 - `invalidate(Rect)`
 - `invalidate(left, top, right, bottom)`

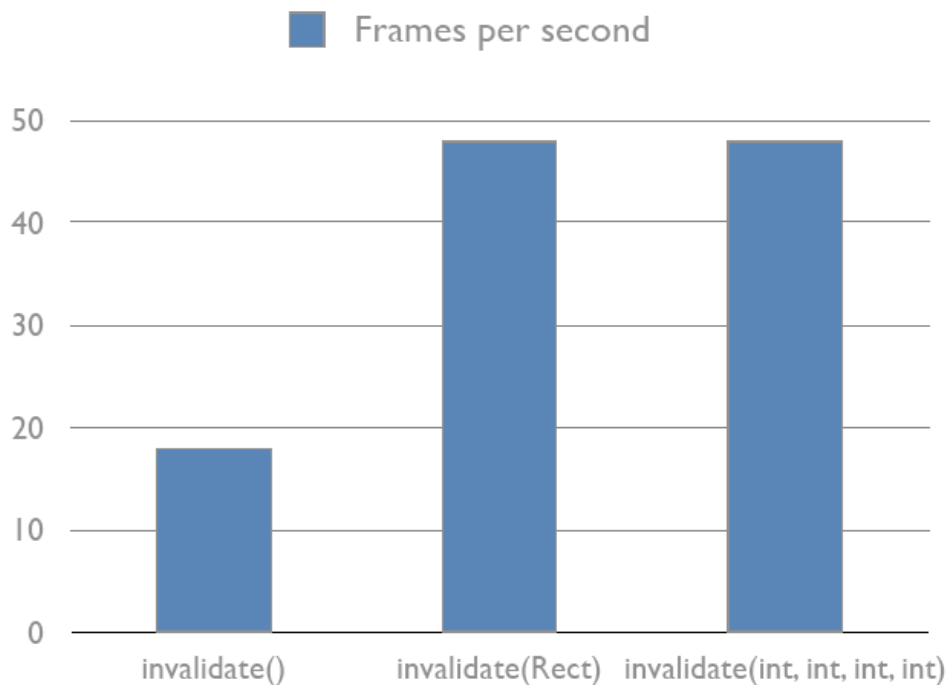


图5, `invalidate()` 方法的性能对比

6.1.4 视图和布局

- a) 如果很多Views
 - 启动时间更长
 - 测量大小更慢
 - 布局更慢
 - 绘制更慢
- b) 层次太深
 - 有可能出现`StackOverflowException`
 - 性能很差, 表现在`inflate`, `findViewById`, `draw`, `layout`, `measure`等方面
- c) 怎么办?

6.1.4.1 优化布局层次

首先，使用Android framework提供的布局元素，也不一定能使布局的性能最好，添加到布局里面的View需要初始化，布局，测量，绘制等。嵌套层次越深，性能越差，特别是使用了layout_weight参数的布局，性能更差，因为，它的每一个子孩子都需要测量两次。对于一个布局要经常加载，这一点非常重要，比如说一个布局应用于ListView或GridView。

假设一个Layout如下：



图6，应用于ListView的布局

第一种XML布局如下：

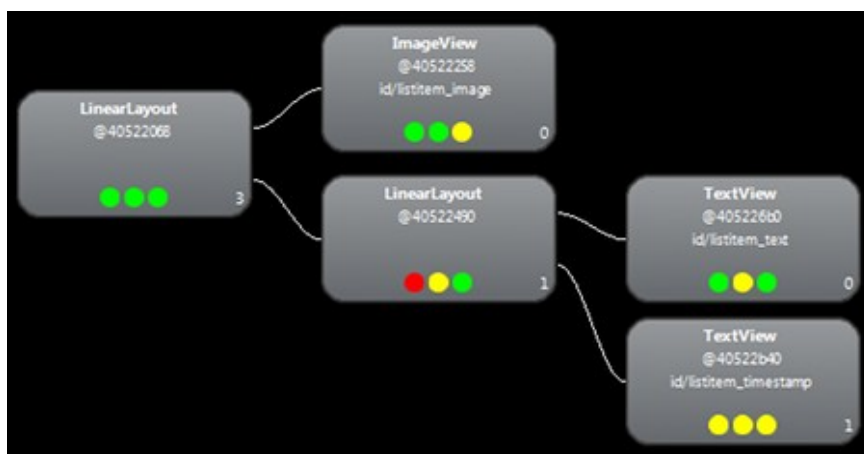


图7，XML布局结构

从上图可以看出，View层次是3层，最顶层是LinearLayout，它有两个child，最左边的是ImageView，右边是一个LinearLayout，布局方向垂直，它有两个child，都是TextView。

使用这种布局的性能分析如下：

- Measure: 0.977ms
- Layout: 0.167ms
- Draw: 2.717ms

第二种XML布局如下：

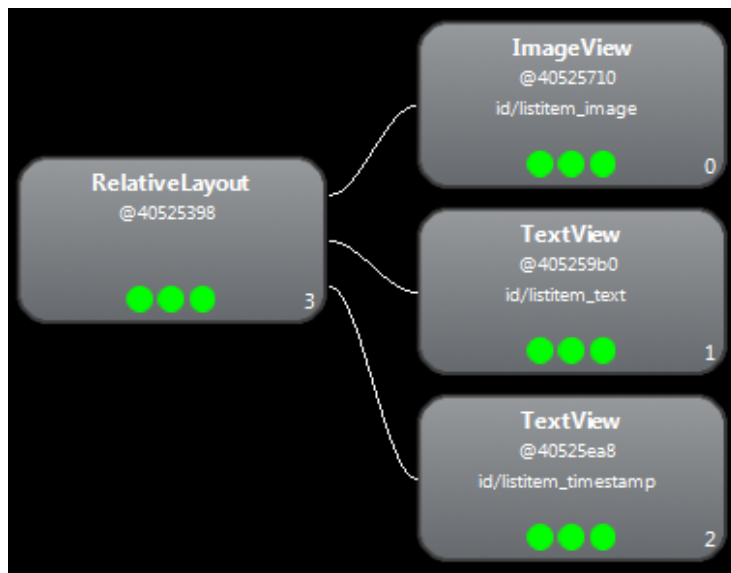


图8，XML布局结构2，使用RelativeLayout

从上图可以看出，这里用到RelativeLayout，View层次只有2层。

现在的性能分析结果如下：

- Measure: 0.598ms
- Layout: 0.110ms
- Draw: 2.146ms

这两种布局，改动很小，但性能差别很大，因为该布局是应用于ListView，它可能会频繁使用，如果在LinearLayout中使用了layout_weight（权重），它也可能导致性能变差。这里只是列出了一个很简单的例子，使用的时候，我们需要关心布局的层次对于性能的影响。

6.1.4.2 布局复用

在Android程序中，有可能会用到很多复杂的组合的组件(Component)，这些组件有可能被复用到其他的界面。我们可以写自定义的View，但用XML更简单，主要是使用<include/>和<merge/>。

<include/>

这个标签用来包括其他的XML布局。请看下面Android Home应用程序的布局。

```
<com.android.launcher.Workspace
    android:id="@+id/workspace"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"

    launcher:defaultScreen="1">

    <include android:id="@+id/cell1" layout="@layout/workspace_screen" />
    <include android:id="@+id/cell2" layout="@layout/workspace_screen" />
    <include android:id="@+id/cell3" layout="@layout/workspace_screen" />

</com.android.launcher.Workspace>
```

上面的XML代码中，引用了workspace_screen三次。在<include/>标签里面，我们可以使用layout_*等属性，也可以指定id。

```
<!-- override the layout height and width -->
<include layout="@layout/image_holder"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent" />

<!-- do not override layout dimensions; inherit them from image_holder -->
<include layout="@layout/image_holder" />
```

当你需要针对设置配置自定义某个UI的一部分时，<include/>标签很有用。例如，你的主布局可以放到/layout文件夹下面，里面的一部分布局（如title bar）可能横竖屏有不同的布局，此时就可以用<include/>标签，所引用的title bar布局分别放到/layout-land和/layout-port文件夹中。

<merge/>

<merge/>标签是为了性能而设计的，主要目的是减少View的层次，从而优化Android布局。它是对<include/>的一个补充，<include/>所引用的layout，里面必须要有一个顶层View，这样就有可能增加在整个View层次结构中的层数。看下面的示例：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:scaleType="center"
        android:src="@drawable/golden_gate" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="20dip"
        android:layout_gravity="center_horizontal|bottom"
        android:padding="12dip"
        android:background="#AA000000"
        android:textColor="#ffffff"
        android:text="Golden Gate" />
</FrameLayout>
```

上面的XML代码中，在一个FrameLayout中添加一个ImageView和TextView。在TextView显示在ImageView的上面。效果图如下图所示：



图9，XML布局的效果图

这个布局在整个UI中的层次结构如下图所示，图中蓝色高亮就是XML布局中的FrameLayout。

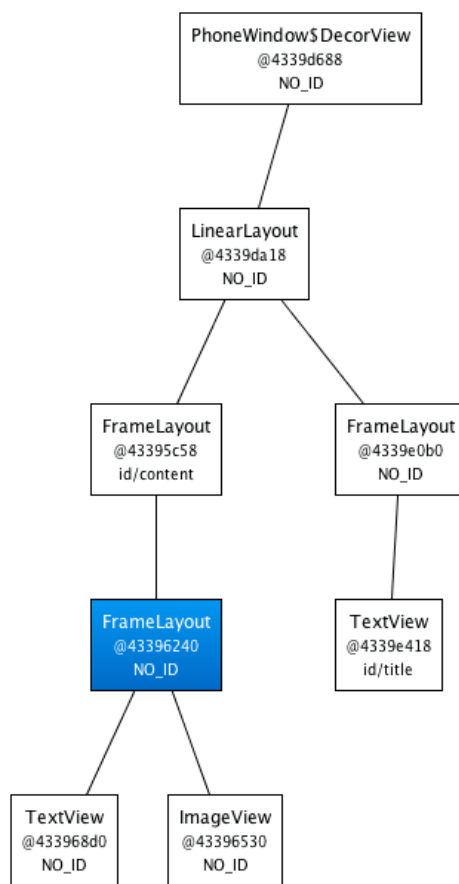


图10, View层次结构

从上图中, 我们得知, 我们定义的FrameLayout (蓝色高亮) 它的大小, 与其Parent一样, 我们也没有定义背景, 所在它其实没有任何用, 它的子View (TextView和ImageView) 其实可以直接放到它的Parent中。那怎么去掉无用的FrameLayout?

我们可以用<merge/>标签了。

```

<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:scaleType="center"
        android:src="@drawable/golden_gate" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="20dip"
        android:layout_gravity="center_horizontal|bottom"
        android:padding="12dip" android:background="#AA000000"
        android:textColor="#ffffff" android:text="Golden Gate" />
</merge>

```

使用<merge/>标签后，得到的XML层次结构如下图所示：

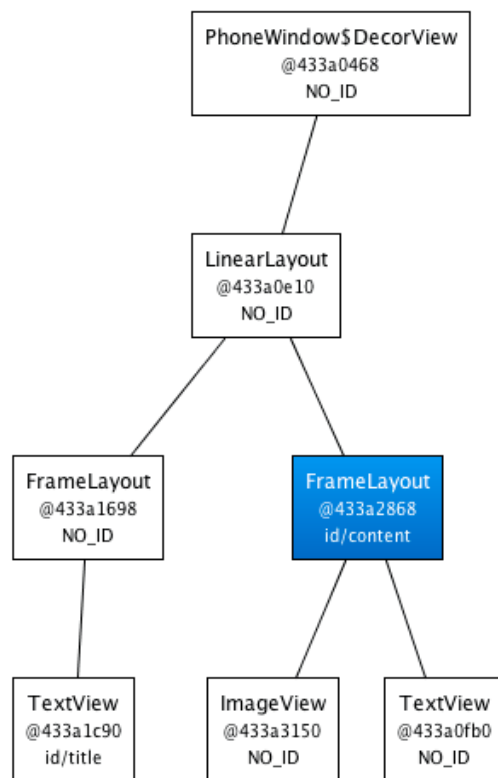


图11，使用<merge/>标签后的层次结构

很明显，View的层次结构减少了一层。

<merge/>限制

- <merge/>标签只能作为XML布局的根结点。
- 当加载一个以<merge/>开头的布局时，你必须指定父ViewGroup，并且设置attachToRoot为true。（请参考inflate(int, android.view.ViewGroup, boolean) 方法）

6.1.4.3 使用ViewStub

关于ViewStub，请参考

<http://developer.android.com/resources/articles/layout-tricks-stubs.html>

ViewStub可以使用我们延迟加载，比如，一个界面里面可能有一个隐藏的进度条，这个进度条在执行某种操作时才会出来，此时我们可以把这个进度条的布局放到ViewStub中，当要使用时才加载。使用ViewStub可以提高启动性能。

用法如下：

XML

```
<ViewStub
    android:id="@+id/stub_import"
    android:inflatedId="@+id/panel_import"
    android:layout="@layout/progress_overlay"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom" />
```

Java

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

6.1.4.4 调用频繁的方法

Masurement	onMeasure()
Layout	onLayout()
Drawing	draw() dispatchDraw() onDraw()
Event handling	dispatchTouchEvent() onTouchEvent()
Adapters	getView() BindView()

6.2, Android程序安全性

6.2.1 Android安全性概念

Android 包括一个应用程序框架、几个应用程序库和一个基于Dalvik虚拟机的运行时，所有这些都运行在Linux内核之上。通过利用Linux内核的优势，Android得到了大量操作系统服务，包括进程和内存管理、网络堆栈、驱动程序、硬件抽象层以及安全性。

在Linux上，一个用户ID识别一个用户，在Android上，一个用户ID，识别一个应用程序。应用程序在安装时被分配用户ID，应用程序在设备上的存续期间内，用户ID保持不变。权限是关于允许或限制应用程序（而不是用户）访问设备资源。

Android使用沙箱的概念来实现应用程序之间的分离和权限，以允许或拒绝一个应用程序访问设备的资源，比如说文件和目录、网络、传感器和API。为此，Android使用一些Linux实用工具（比如说进程级别的安全性、与应用程序相关的用户和组ID，以及权限），来实现应用程序被允许执行的操作。

沙箱的概念，可以表示为下图所示：

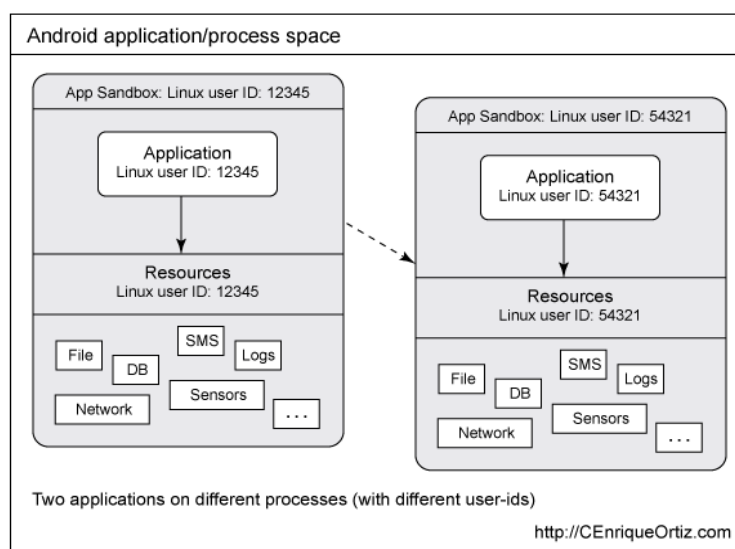


图12，两个Android应用程序，各自在其自己的基本沙箱或进程上

Android应用程序运行在它们自己的Linux进程上，并被分配一个惟一的用户ID。默认情况下，运行在基本沙箱进程中的应用程序没有被分配权限，因而防止了此类应用程序访问系统或资源。但是Android应用程序可以通过应用程序的manifest文件请求权限。

通过做到以下两点，Android 应用程序可以允许其他应用程序访问它们的资源：

- 声明适当的manifest权限。
- 与其他受信任的应用程序运行在同一进程中，从而共享对其数据和代码的访问。

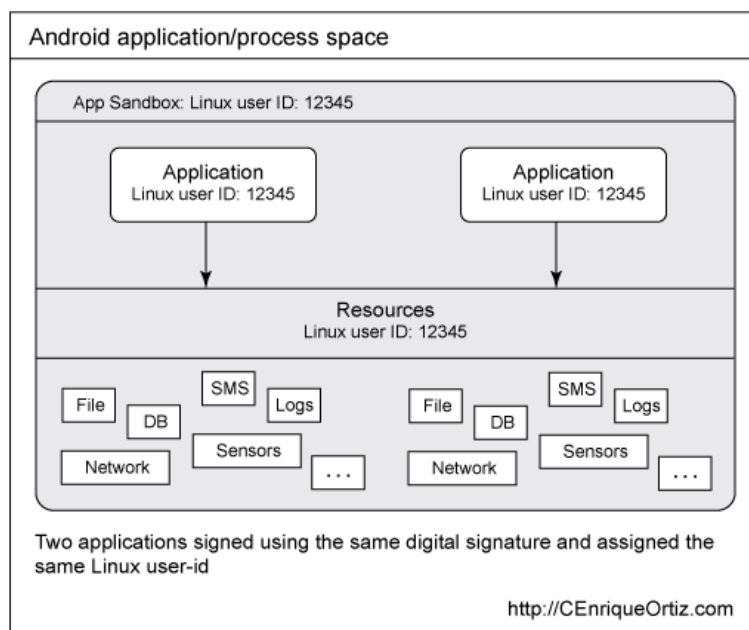


图13，两个Android应用程序，运行在同一进程上

6.2.2 应用程序签名

所有Android应用程序都必须被签名。应用程序或代码签名是一个这样的过程，即使用私有密钥数字地签署一个给定的应用程序，以便：

- 识别代码的作者
- 检测应用程序是否发生了改变
- 在应用程序之间建立信任

构建Android应用程序时可以采用**调试模式(Debug)**和**发布模式(Release)**：

- 使用Android构建工具（命令行和Eclipse ADT）构建的应用程序是用一个调试私有密钥自动签名的；这些应用程序被称为调试模式应用程序。调试模式应用程序用于测试，不能够发布。注意，未签名的或者使用调试私有密钥签名的应用程序不能够通过Android Market发布。

- 您准备发布自己的应用程序时，必须构建一个发布模式的版本，这意味着用私有密钥签署应用程序。

6.2.3 使用权限

权限是一种Android平台安全机制，旨在允许或限制应用程序访问受限的API和资源。默认情况下，Android应用程序没有被授予权限，这通过不允许它们访问设备上的受保护 API 或资源，确保了它们的安全。权限在安装期间通过manifest文件由应用程序请求，由用户授予或不授予。

Android定义长长的一系列 manifest 权限，以保护系统或其他应用程序的各个方面。要请求权限，可以在manifest文件中声明一个<user-permission>属性：

```
<uses-permission android:name="string" />
```

在manifest中声明权限

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0"
    package="com.cenriqueortiz.tutorials.datastore"
    android:installLocation="auto">

    <application

    </application>

    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
</manifest>
```

声明自定义权限

```
<permission
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.cenriqueortiz.android.ACCESS_FRIENDS_LIST"
    android:description="@string/permission_description"
    android:label="@string/permission_label"
    android:protectionLevel="normal"
>
</permission>
```

6.2.3 数据存储

6.2.3.1 使用内部存储

6.2.3.2 使用外部存储

6.2.3.3 使用Content Provider

7. Android 程序性能改善意见

在开发Android应用程序时，要多考虑程序的性能问题，因为在移动平台，性能是很重要的因素。关于Android程序性能改善，请参考

<http://developer.android.com/guide/practices/design/performance.html>

7.1 一般原则

- XML布局层次不要太多，层次越大，性能越差，特别是作为ListView, GridView的孩子。
- 在不影响接口设计的情况下，尽量使用实类型，而不是抽象类型，因为实类型到抽象类型之间有一个转换。比如用ArrayList<T>，而不是List<T>。
- 如果你所设计的类方法，不需要访问成员变量，应该把该方法设计成静态方法，比非静态效率提高15%-20%。
- 程序内部访问，尽量避免使用getter/setter，而是直接访问成员变量。
- 常量类型用static final声明。
- 注意循环的用法，这很可能影响的效率。请看下面的示例代码：

```
static class Foo {
    int mSplat;
}
Foo[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one() {
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;

    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two() {
    int sum = 0;
    for (Foo a : mArray) {
        sum += a.mSplat;
    }
}
```

zero() 方法最慢，因为JIT不会对取得数据长度进行优化。

one() 方法稍好一些。它把成员变量，数据长度放到局部变量里面，避免去查找。

two() 方法效率最快。

- 避免在循环中频繁构建和释放对象。
- 不再使用的对象应及时销毁。
- 避免使用太多的synchronized关键字来同步方法。
- 少用static字段，static对象生命周期与Application一样。
- 不要在View#onDraw()方法里面创建对象。
- 尽量减少Android程序布局中View的层次，View越多，效率就越低。

7.2 为反应灵敏而设计

在Android程序中，如果程序在一段时间内没有响应用户，它会显示一个提示对话框，叫做ANR对话框（Application Not Responding），如图1所示。

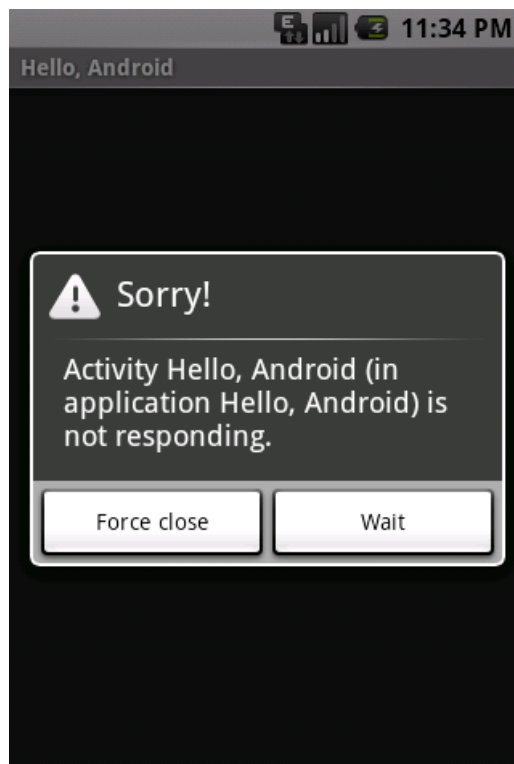


图1，ANR对话框

7.2.1 什么时候触发ANR？

在Android程序中，程序的响应性是由Activity Manager和Window Manager来监听的，它会在以下两种情况下显示ANR对话框：

- 在5秒内没有响应用户的输入（屏幕触摸，按键事件等）。
- 在10秒内没有执行完BroadcastReceiver。

7.2.2 如何避免ANR

- 耗时的工作，放到后台线程里去做，显示对话框提示用户，如果网络下载，文件操作。

- 对于游戏程序，计算的工作放到子线程中。

7.2.4 查看ANR的Log记录

当程序出现ANR后，系统会把出现ANR的原因记录下来，我们可以查看该文件并分析，从而可以帮助我们找到出现ANR的原因，这对我们提高程序性能很有帮助。

Log文件请参考下图所示

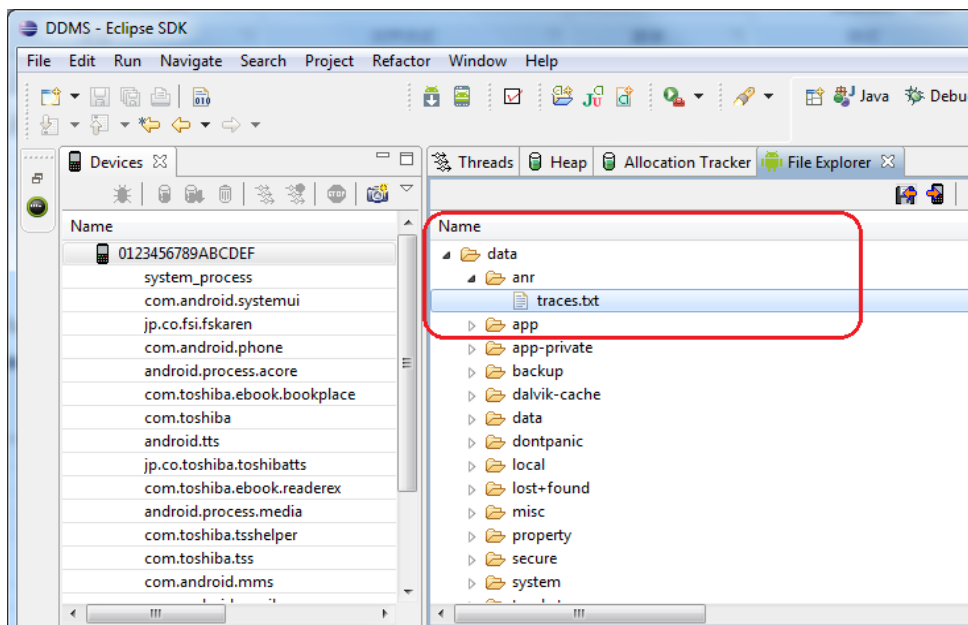


图2，ANR记录的文件

ANR的Log文件记录在 `data/anr/traces.txt` 文件中，我们可以导出来，它里面的格式一般是

```
DALVIK THREADS:
(mutexes: tll=0 tsl=0 tscl=0 ghl=0 hwl=1 hwll=0)
main prio=5 tid=1 NATIVE
  | group="main" sCount=1 dsCount=0 obj=0x2aaca178 self=0xce38
  | sysTid=7096 nice=0 sched=0/0 cgrp=[fopen-error:2] handle=1876218944
  at com.android.server.SystemServer.init1(Native Method)
  at com.android.server.SystemServer.main(SystemServer.java:637)
  at java.lang.reflect.Method.invokeNative(Native Method)
  at java.lang.reflect.Method.invoke(Method.java:507)
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:839)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597)
  at dalvik.system.NativeStart.main(Native Method)
```

7.2.3 滑动时停止加载数据

当ListView或GridView显示一些图片时，我们正确的做法是：

- 当开始滑动时，停止滑动时后台线程加载数据。
- 当停止滑动时，开始加载当前一屏的数据，不要全部加载。
- 不显示的数据，及时回收内存，防止OOM异常。

7.3 为准确而设计

7.3.1 不要丢掉数据

始终要记住，Android是一个移动平台的操作系统，别的Activity很有可能弹出来（如电话打入时），显示在最上面，此时，应用程序的onSaveInstanceState()和onPause()方法会被调用，应用程序有可能被应用程序杀死。在这种情况下，数据的存储很重要。

Do:

- 在Activity#onPause()方法里存储永久的数据。
- 不要阻塞Activity#onPause()方法。

7.3.2 不要暴露原始数据

不要把应用程序的私有数据暴露给别的程序，因为这样会导致安全问题。

Do:

- Android通常是通过提供ContentProvider来访问程序的数据。如果Contact程序等。
- 通过shared uid与其他APK共享数据。

7.3.3 不要打断用户

当用户正在运行一个程序时（如打电话），我们后台的程序是不能打断用户的操作。如果我们在后台运行了BroadcastReceiver或服务的话，在这里面不要调用startActivity来显示一个界面。如果这样做的话，都会打断正在运行的程序。

Do:

- 通过NotificationManager显示通知。

7.3.4 不要在一个Activity里面做过多的事情

Android程序应该是一个activity的集合，不要把所有的事情都放到的一个activity中去做，这样对于以后的维护相当困难。

Do:

- 当设计Android程序时，程序应该是activity的集合。

7.3.5 耗时工作放到线程

对于像网络，读写文件等操作很可能是很费时的，所有这些操作都应当放到后台线程，如果放在UI线程中，则有可能出现ANR错误。

Do:

- 用Thread或AsyncTask来启动后台任务。
- 提示用户，后台线程存在进行的任务。

7.3.6 扩展系统主题

Do:

- 注意版本的兼容性。

7.3.7 对应多屏幕分辨率与多DPI

要考虑到你的程序运行在不同的设备上的这种情况，需要注意以下几点：

- 在XML布局中，尽量灵活
- 不要直接在XML或Java写字体大小，View大小等常量
- 设计不同DPIs的资源图片
- 不同大小的屏幕，设计不同的布局

Do:

- View大小等常量提取到dimension.xml中，不同布局可以具体不同的dimension。

7.3.8 假设网络很慢

网络访问可能会受到设备环境，远程服务器等因素，会导致访问超时，所以网络访问需要特别设计。

Do:

- 设计超时机制
- 网络访问在后台线程完成
- 用户可以取消

7.3.9 节约设备电池

减少CPU运行时间，大量的CPU使用会很快把电池电量消耗殆尽，对于移动设备而言，电池续航时间尤其重要。

Do:

- 从设计架构上做到最优化
- 不要在后台线程中不断请求远程服务器
- 减少不必要的绘制

7.3.10 不要假设有触摸屏和键盘

对于有些设备，不具有触摸屏与键盘。

Do:

- 分析新产品面对的用户及设备

7.4 内存泄漏

这里所说的内存泄漏是指那些无法被GC回收的对象，这些对象通常是被其他对象强引用。

一般的Android程序是8MB的堆内存，有些是16MB，所以如果对象不能被回收，最后就会达到内存上限，最后就有可能出现OOM异常，或者是GC强制回收内存，导致程序运行很慢。GC回收内存很占时间。

Android中context可以作很多操作，但是最主要的功能是加载和访问资源。在android中有两种context，一种是 application context，一种是activity context，通常我们在各种类和方法间传递的是activity context。

比如一个activity的onCreate：

```
protected void onCreate(Bundle state)
{
    super.onCreate(state);
    TextView label = new TextView(this); //传递context给view control
    label.setText("Leaks are bad");
    setContentView(label);
}
```

把activity context传递给view，意味着view拥有一个指向activity的引用，进而引用activity占有的资源：view hierachy, resource等。

这样如果context发生内存泄露的话，就会泄露很多内存。

Leaking an entire activity是很容易的一件事。

当屏幕旋转的时候，系统会销毁当前的activity，保存状态信息，再创建一个新的。

比如我们写了一个应用程序，它需要加载一个很大的图片，我们不希望每次旋转屏幕的时候都销毁这个图片，重新加载。实现这个要求的简单想法就是定义一个静态的Drawable，这样Activity 类创建销毁它始终保存在内存中。

下面代码示例了内存泄漏：

```
public class myactivity extends Activity {
    private static Drawable sBackground;
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        TextView label = new TextView(this);
        label.setText("Leaks are bad");

        if (sBackground == null) {
            sBackground = getDrawable(R.drawable.large_bitmap);
        }
    }
}
```


这段程序看起来很简单，但是却问题很大。当屏幕旋转的时候会有泄漏，特别是频繁的旋转屏幕（即GC没法销毁activity）。

我们刚才说过，屏幕旋转的时候系统会销毁当前的activity。但是当drawable和view关联后，drawable保存了view的 reference，即setBackground保存了label的引用，而label保存了activity的引用。既然drawable不能销毁，它所引用和间接引用的都不能销毁，这样系统就没有办法销毁当前的activity，于是造成了内存泄露。GC对这种类型的内存泄露是无能为力的。

避免这种内存泄露的方法是：

避免activity中的任何对象的生命周期长过activity，避免由于对象对 activity的引用导致activity不能正常被销毁。我们可以使用application context。application context伴随application的一生，与activity的生命周期无关。application context可以通过Context#getApplicationContext() 或者Activity#getApplication() 方法获取。

7.5 避免内存泄漏，请记住以下几点：

- 1) 不要让生命周期长的对象引用activity context。即保证引用activity的对象的生命周期与activity一样。长生命周期不要引用比自己生命周期短的对象。
- 2) 对于生命长的对象，请有application context。
- 3) 避免非静态的内部类，尽量使用静态类，避免生命周期问题，注意内部类对外部对象引用导致的生命周期变化。
- 4) 资源没有显示释放
 - 没有反注册对象
注册了Broadcast Receiver，但是没有反注册。
注册了listener，但是没有反注册，导致被注册对象被listener引用，不能被回收。
 - 集合中的对象，没有清理造成内存泄漏。
通常我们把一些有用的对象添加到集合中，但是不用时，没有从集合中删除，导致这些对象被集合对象引用，无法回收。如果这个集合是static的话，情况更加严重。
- 5) 资源对象没有关闭
 - 使用的数据库cursor没有调用close()方法来关闭。
 - 操作文件的流，没有关闭。尽管GC可能回收这些未关闭的对象，但效率低下，所有我们在不使用后，主动关闭。
- 6) 一些不良代码造成的内存压力
 - Bitmap没有调用recycle()方法。我们可以主动调用recycle()方法来内存。
 - 构造Adapter时，没有使用convertView。

不良代码：

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View view = new Xxx(...);
    return view;
}
```

良好代码:

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View view = null;
    if (convertView != null)
    {
        view = convertView;
        populate(view, getItem(position));
    }
    else
    {
        view = new Xxx(...);
    }
    return view;
}
```

8. Android 数据存储

在Android系统中，一般有四种数据存储方式，不同的存储方式有着不同的应用。

- **Shared Preferences**

以键值对的形式存储数据，快速，轻量级存储。

- **File**

移动设备，可移动存储媒介，APK文件系统。

- **SQLite Databases**

存储结构化的数据。

- **Network**

存储数据到远程服务器。

8.1 Shared Preferences

Android Framework提供了SharedPreferences类来存储原始数据，可以是booleans, float, int, string等。

注意：这里只能是存储原始数据，不能存储自定义的类对象。

8.1.1 如何得到SharedPreferences对象？

- Context#getSharedPreferences() - 可以指定文件名称
- Context#getPreferences() - 不需要指定文件名称，只有你的Activity可以使用。

8.1.2 如何写数据？

- 调用edit()方法得到SharedPreferences.Editor的实例。
- 调用putXXX()方法来设置数据。
- 调用commit()方法，提交数据，最终会写入文件。

```
// Restore preferences
SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
boolean silent = settings.getBoolean("silentMode", false);
setSilent(silent);

// Wirte preferences
SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
SharedPreferences.Editor editor = settings.edit();
editor.putBoolean("silentMode", mSilentMode);
// Commit the edits!
editor.commit();
```

在项目中，我们通常会把Shared Preference操作封装，一般是如下形式：

```
public class SettingPreferenceUtil {

    private static final String PREF_NAME = "settings_prefs";
    private Editor mEditor = null;

    public beginSet()
    {
        mEditor = context.getSharedPreferences(PREF_NAME,
                                                0).edit();
    }

    public boolean endSet()
    {
        return mEditor.commit();
    }

    public void putString(String key, String val) {

    }

    public String getString(String key, String defVal) {

        return null;
    }
}
```

上面代码中的SettingPreferenceUtil类，提供了beginSet()和endSet()方法，所有的写操作都应当放到beginSet()和endSet()之间。

用法如下：

```
SettingPreferenceUtil spu = new SettingPreferenceUtil();
spu.beginSet();

spu.putString("key", "test string");
// ...

spu.endSet();
```

8.2 Files

8.2.1 使用内部存储

默认情况下，这些数据是存储在应用程序的内部存储，它对于外部应用程序是不可见的，当应用程序删除时，这些数据也会删除，路径是：`/data/data/<package_name>/files/`。

- `Context#openFileOutput()`，得到一个`FileOutputStream`对象。
- 调用`wirte()`方法。
- 调用`close()`方法关闭流。

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos =
    this.openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

8.2.2 使用外部存储

这里所说的外部存储，并不是指SD Card(SD Card其实也算是一种可移除的外部存储)，而是指Android系统的存储空间，它跟内部存储相对应。内部存储是指应用程序是存储空间，而外部存储与内部存储相对应。

注意：外部存储的文件很有可能被删除，所有的应用程序都可以读写这些文件。

- 检查媒体是否可用

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many
    // other states, but all we need
    // to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

- 访问外部存储的文件

Environment类提供了一系列方法来取得外部存储的路径，如Environment.getExternalStorageDirectory()方法返回一个File对象。

- 保存可共享的文件

通常情况下，使用Environment.getExternalStoragePublicDirectory()方法，你可以指定相应的参数来标识取什么类型的文件夹路径，如DIRECTORY_MUSIC，DIRECTORY_PICTURES等。一般公有的文件夹有：

```
Music/  
Podcasts/  
Ringtones/  
Alarms/  
Notifications/  
Pictures/  
Movies/  
Download/
```

8.2.3 读写文件

通常会用到的类：

- | | |
|--------------------|-----------|
| • File | 一个文件的抽象表示 |
| • FileInputStream | 表示输入流 |
| • FileOutputStream | 表示输出流 |

下面代码示例了从输入流中读数据，并写到输出流里。

```
public boolean copyStream(InputStream inputStream,
                          OutputStream outputStream) {
    try
    {
        byte[] buf = new byte[1024];
        int len = 0;
        while ((len = inputStream.read(buf)) > 0)
        {
            outputStream.write(buf, 0, len);
        }
        outputStream.flush();
        inputStream.close();
        inputStream = null;
        outputStream.close();
        outputStream = null;
        // Notify GC to do the clear job.
        System.gc();
        return true;
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    return false;
}
```

用法如下:

```
// Do flush bit job for destination file.
String src = "/mnt/sdcard/a.dat";
String dest = "/mnt/sdcard/b.dat";

InputStream inStream = new FileInputStream(src);
OutputStream outStream = new FileOutputStream(dest);

// Read and write the stream
copyStream(inStream, outStream);
```

8.3 使用数据库

Android中使用了开源的SQLite3作为其数据库，底层是用C/C++实现，上层用Java封装了，在framework层，我们通常用到的类有：

- SQLiteDatabase
- SQLiteOpenHelper
- Cursor
- ...

条款1：创建数据库表

我们会继承Android提供的SQLiteOpenHelper类，如MySQLiteOpenHelper，在#onCreate()方法里面，我们一般都用来创建数据表。示例代码如下：

```
@Override
public void onCreate(SQLiteDatabase db)
{
    try
    {
        PhotoDB.createTable(db);
        AudioDB.createTable(db);
        //...
    }
    catch (SQLException e)
    {
    }
}
```

注意：

上面代码中的PhotoDB，AudioDB提供了createTable()静态方法来创建表，这里也可以不用静态方法。

条款2：设计基类AbstractDB

设计一个基类AbstractDB，每个数据库表所对应的类都应该继承它。

- 提供一些共通方法：
 - getWritableDatabase
 - getReadableDatabase
 - closeDB
 - closeCursor
 - **abstract createTable**
 - **abstract upgradeTable**
 - ...
- 使用一个SQLiteDatabase实例，得到数据库实例都是调用基类的方法

下面代码示例了如何使整个应用程序中使用同一个SQLiteDatabase实例。这种情况下，应该在程序的主Activity退出时，关闭数据库。

```
@Override
public synchronized SQLiteDatabase getWritableDatabase()
{
    if (null == m_databaseInstance)
    {
        m_databaseInstance = super.getWritableDatabase();
    }

    return m_databaseInstance;
}
```

注意：所有的创建数据库表的类，都应该继承这个类，重写onCreateTable()方法。

条款3：实现数据库表类

每一张数据库表，应当对应一个类，这个类的命名格式：**表名+DB**，如PhotoDB。这个类的设计，应当注意以下几点：

- 这个类继承于AbstractDB
- 这个类应当提供一个方法用于创建表，createTable(SQLiteDatabase db)
- 必须把表名，定义成常量字段，不能直接写在SQL语句中
- 操作表时，尽量不要去拼SQL语句，而应该用SQLiteDatabase提供的方法
- 将各个表分别封装到一个类中，从而可以达到模块化，而且方便管理与维护

说明：

数据库表类的命名格式不是规定，只是一个建议，也可以是其他，如**表名+Table**。

- 操作数据时，一般按如下格式来写：

```
SQLiteDatabase db = getWritableDatabase();
Cursor cursor = null;
try
{
    // Do something.
}
catch (SQLException e)
{
    e.printStackTrace();
}
finally
{
    closeCursor(cursor);
    closeDB(db);
}
```

注意：所有的释放资源的操作，一定要放在finally里面，这样总是能保证资源会被释放。

- 关于事务处理，请参考下面的代码：

```
SQLiteDatabase db = getWritableDatabase();
try
{
    db.beginTransaction();

    // Do something.

    db.setTransactionSuccessful();
}
catch (SQLException e)
{
    e.printStackTrace();
}
finally
{
    db.endTransaction();
    closeDB(db);
}
```

注意：结束事务一定要放到finally里面，这样才能保证begin/end配对。

条款4：访问SQLiteDatabase的方法，加上try-catch-finally语句块

- 通过try-catch-finally总是能确保数据库，cursor能正常关闭
- 捕捉数据库异常

条款5：使用自增主键

根据Google官方文档，他们强烈推荐使用自增的字段来作为表的主键，这样始终能保证主键的唯一性。

条款6：重写onCreate、onUpgrade方法

在SQLiteOpenHelper的派生类，我们必须重写onCreate()，onUpgrade()方法。

- onCreate()

在这个方法里面，我们通常就是创建数据库表。

- onUpgrade()

这个方法在当数据库需要更新的时候，会被调用。我们在这个方法里面，主要是更新数据库，如删除表，更改表，删除数据等。如果执行了删除表操作，通常会手动调用onCreate()来创建表。

这个方法是否调用，主要是根据数据库的版本号，它是一个整数，在创建SQLiteOpenHelper派生类的实例时，需要传入数据库文件路径，版本号，如果这个版本号发生改变，那么onUpgrade()就会被调用。

上面代码显示了创建一个MyDBHelper类的实例。

```
MyDBHelper mDBHelper = new MyDBHelper(  
    context,  
    DATABASE_NAME,  
    null,  
    DATABASE_VERSION);
```

下面代码说明了在onUpgrade()方法里的实现代码，在删除表之后，再次调用onCreate()方法来创建表。

```

if (upgradeVersion != currentVersion) {

    // Delete the Tables.
    db.execSQL("DROP TABLE IF EXISTS system");
    db.execSQL("DROP INDEX IF EXISTS systemIndex1");
    db.execSQL("DROP TABLE IF EXISTS secure");
    db.execSQL("DROP TABLE IF EXISTS gservices");
    db.execSQL("DROP INDEX IF EXISTS gservicesIndex1");
    db.execSQL("DROP TABLE IF EXISTS bluetooth_devices");
    db.execSQL("DROP TABLE IF EXISTS bookmarks");

    onCreate(db);
}

```

条款7：数据库升级策略

在应用程序开发的过程中，数据库的升级是一个很重要的组成部分（如果用到了数据库），因为程序可能会有V1.0，V2.0，当用户安装新版本的程序后，必须要保证用户数据不能丢失，对于数据库设计，如果发生变更（如多添加一张表，表的字段增加或减少等），那么我们必须想好数据库的更新策略。

- 定义数据库版本

数据库的版本是一个整型值，在创建SQLiteOpenHelper时，会传入该数据库的版本，如果传入的数据库版本号比数据库文件中存储的版本号大的话，那么SQLiteOpenHelper#onUpgrade()方法就会被调用，我们的升级应该在该方法中完成。

- 如何写升级逻辑

假如我们开发的程序已经发布了两个版本：V1.0，V1.2，我们正在开发V1.3。每一版的数据库版本号分别是18，19，20。

对于这种情况，我们应该如何实现升级？

用户的选择有：

- 1) V1.0 → V1.3 DB 18 → 20
- 2) V1.1 → V1.3 DB 19 → 20

记住：数据库的每一个版本所代表的数据库必须是定义好的，比如说V18的数据库，它可能只有两张表TableA和TableB，如果V19要添加一张表TableC，如果V20要修改TableC，那么每一个版本所对应的数据库结构如下：

```

V18 ---> TableA, TableB
V19 ---> TableA, TableB, TableC
V20 ---> TableA, TableB, TableC （变更）

```

那我们在onUpgrade()方法里面怎么实现？

```

// Pattern for upgrade blocks:
//
//   if (upgradeVersion == [the DATABASE_VERSION you set] - 1) {
//       .. your upgrade logic..
//       upgradeVersion = [the DATABASE_VERSION you set]
//   }

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
    int upgradeVersion = oldVersion;

    if (18 == upgradeVersion) {
        // Create table C
        String sql = "CREATE TABLE ...";
        db.execSQL(sql);
        upgradeVersion = 19;
    }

    if (20 == upgradeVersion) {
        // Modify table C
        upgradeVersion = 20;
    }

    if (upgradeVersion != newVersion) {
        // Drop tables
        db.execSQL("DROP TABLE IF EXISTS " + tableName);
        // Create tables
        onCreate(db);
    }
}

```

从上面的代码可以看到，我们在onUpgrade()方法中，处理了数据库版本从18 -> 20的升级过程，这样做的话，不论用户从18 -> 20，还是从19 -> 20，最终程序的数据库都能升级到V20所对应的数据库结构。

- 如何保证数据不丢失

这是很重要的一部分，假设要更新TableC表，我们建议的做法是：

- 1) 将TableC重命名为TableC_temp

SQL语句可以这样写：

```
ALTER TABLE TableC RENAME TO TableC_temp;
```

- 2) 创建新的TableC表
- 3) 将数据从TableC_temp中插入到TableC表中

SQL语句可以这样写:

```
INSERT INTO TableC (Col1, Col2, Col3) SELECT (Col1, Col2, Col3) FROM
TableC_temp;
```

经过这三步，TableC就完成了更新，同时，也保留了原来表中的数据。

注意:

在onUpgrade()方法中，删除表时，注意使用事务处理，使得修改能立即反应到数据库文件中。

条款8: SQL语句

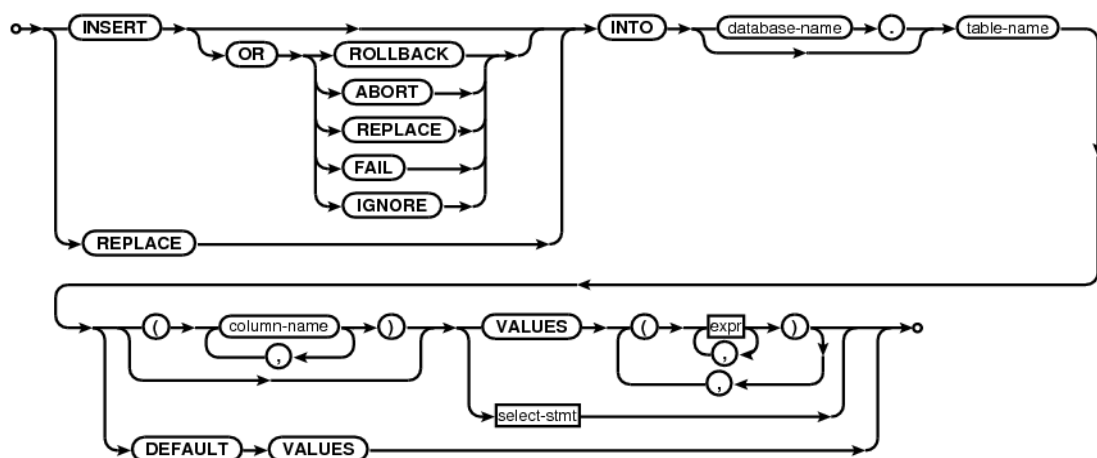
由于Android是使用开源的SQLite3作为其数据库，所以，我们在开发数据库模块时，一定要注意SQLite3支持哪些关键字，函数等，不是所有的关键字，SQLite都是支持的。

下面列出了一些参考链接:

- SQLite3官方文档

<http://sqlite.org/>

该网站提供了SQLite3的官方说明，很详细，比如说关于INSERT语句的用法如下:



- W3CSchool网站

<http://www.w3school.com.cn/sql/index.asp/>

注意:

SQL语句写得好坏能直接影响到数据库的操作。我曾经就遇到过SQL语句影响查询性能，更新3000条记录，用时30秒左右，但在对WHERE条件的字段加上索引后，性能提升到3~4秒。

8.4 Network

关于网络，请参考：

<http://developer.android.com/training/basics/network-ops/index.html>

使用网络，使用以下的包：

- java.net.*
- android.net.*

使用网络，需要添加以下的权限：

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="
    "android.permission.ACCESS_NETWORK_STATE" />
```

9. Android Bitmap操作

Bitmap操作在Android应用程序中应用得非常广泛，在操作Bitmap时，一定要注意防止其超过内存限制，如果超出的话，Android系统就会抛出Out Of Memory (OOM) 异常。有如下原因：

1，移动设备非资源要求很严格，Android设备通常仅为一个应用程序分配16MB的内存，一个应用程序应该优化内存，从而达到更好的性能。虚拟机内存定义请参见下表。（该表内容可参考Android Compatibility Definition Document (CDD)）

Screen Size	Screen Density	Application Memory
small/normal/large	ldpi/mdpi	16MB
small/normal/large	tvdpi/hdpi	32MB
small/normal/large	xhdpi	64MB
xlarge	mdpi	32MB
xlarge	tvdpi/hdpi	64MB
xlarge	xdpi	128MB

2，Bitmap会占很大内存，比如说2592x1936像素的图片，如果使用ARGB_8888格式的话，其内存大小可能是19MB（2592*1936*4字节），这么大的内存，对于一个应用来说，立即就达到其内存上限。

3，Android应用程序的UI可能一次加载很多Bitmap，比如说像ListView，GridView，它们有可能包含很多Bitmap，特别是当数据很多的情况下。

9.1 加载大Bitmap

9.1.1 读取Bitmap的大小及类型

BitmapFactory类提供了一系列方法来创建Bitmap（`decodeByteArray()`，`decodeFile()`，`decodeResource()`等），这些方法都有可能導致OOM异常，我们可以在创建Bitmap之前，先确定其大小。

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true; // Set true to get bitmap bounds
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

这里，只是得到了Bitmap的大小和类型，并不能减少内存的使用。

9.1.2 创建缩放的Bitmap

我们可以通过指定BitmapFactory.Options#inSampleSize来设定Bitmap的大小，如果sample size越大，那么加载出来的Bitmap大小就会越小。

这个Sample size通常是2的次方，即2，4，8，16等。

```
public static int calculateInSampleSize(
    BitmapFactory.Options options, int reqWidth, int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {
        if (width > height) {
            inSampleSize =
                Math.round((float)height / (float)reqHeight);
        } else {
            inSampleSize =
                Math.round((float)width / (float)reqWidth);
        }
    }
    return inSampleSize;
}
```

这个方法根据指定的大小，计算出sample size的值。

```
public static Bitmap decodeSampledBitmapFromResource(Resources res,
    int resId, int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options =
        new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);
    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options,
        reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

这个方法，创建了一个指定sample size的Bitmap，通过这种方式创建出来的Bitmap是经过缩放的。

上面计算sample size的方法，不是最好的实现，下面的方法更好一些，**getAppropriateSampleSize**方法，返回了动态计算出的确sample size值。

```
public static int getAppropriateSampleSize(
    InputStream is,
    int minSideLength,
    int maxNumOfPixels)
{
    int sampleSize = 1;

    try
    {
        BitmapFactory.Options opts = new BitmapFactory.Options();
        opts.inJustDecodeBounds = true;
        BitmapFactory.decodeStream(is, null, opts);

        if (opts.outHeight > 0 && opts.outWidth > 0)
        {
            sampleSize =
                computeSampleSize(opts,
                    minSideLength,
                    maxNumOfPixels);
        }
    }
    catch (OutOfMemoryError e)
    {
        e.printStackTrace();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

```
public static int computeSampleSize(BitmapFactory.Options options,
    int minSideLength,
    int maxNumOfPixels)
{
    int initialSize = computeInitialSampleSize(options,
        minSideLength, maxNumOfPixels);
    int roundedSize;
    if (initialSize <= 8)
    {
        roundedSize = 1;
        while (roundedSize < initialSize)
        {
            roundedSize <<= 1;
        }
    }
    else
    {
        roundedSize = (initialSize + 7) / 8 * 8;
    }

    return roundedSize;
}
```

```
private static int computeInitialSampleSize(
    BitmapFactory.Options options,
    int minSideLength,
    int maxNumOfPixels)
{
    double w = options.outWidth;
    double h = options.outHeight;

    final int UNCONSTRAINED = -1;
    int lowerBound = (maxNumOfPixels == UNCONSTRAINED) ?
        1 :
        (int) Math.ceil(Math
            .sqrt(w * h / maxNumOfPixels));
    int upperBound = (minSideLength == UNCONSTRAINED) ?
        128 :
        (int) Math.min(
            Math.floor(w / minSideLength),
            Math.floor(h / minSideLength));

    if (upperBound < lowerBound)
    {
        // return the larger one when there is no overlapping zone.
        return lowerBound;
    }
    if ((maxNumOfPixels == UNCONSTRAINED) &&
        (minSideLength == UNCONSTRAINED))
    {
        return 1;
    }
    else if (minSideLength == UNCONSTRAINED)
    {
        return lowerBound;
    }
    else
    {
        return upperBound;
    }
}
```

9.2 缓存Bitmap

9.2.1 使用内存缓存

对于移动应用程序，不应该把所有数据都保存在内存中，例如，当GridView或ListView显示很多图片时，我们应当怎么办？

- 动态加载，需要显示时才加载。
- 释放不需要显示的数据(Bitmap)。

下面的代码显示了ImageCache类。它是一个先进先出的队列（FIFO），这个队列存储一组数据对象，当达到最大容量时，删除第一个，删除的这个元素就是可以被释放的。

```
public static class ImageCache {
    private static final int MAX_CAPACITY = 50;
    private LinkedList<ImageInfo> m_linkedList =
        new LinkedList<ImageInfo>();

    public ImageInfo add(ImageInfo info) {
        if (null != info) {
            m_linkedList.add(info);
        }

        ImageInfo firstInfo = null;
        if (m_linkedList.size() > MAX_CAPACITY) {
            firstInfo = m_linkedList.removeFirst();
        }
        return firstInfo;
    }

    public void clear() {
        for (ImageInfo info : m_linkedList) {
            info.recycle();
        }
        m_linkedList.clear();
    }
}
```

注意：

当释放一个Bitmap时，一定要注意，这个Bitmap有可能仍然被一个View所引用，如果直接调用Bitmap#recycle()方法的话，那么View就会抛出异常，表明View正在使用一个已经被回收的Bitmap。因此，在释放Bitmap之前，先判断该Bitmap对象是否有View引用，如果没有，就可以安全释放。

比如说在GridView中显示很多图片，不可见的应该释放，但这些需要释放的Bitmap很有可能仍然被View所引用。

9.2.2 使用文件缓存

使用内存缓存，它能加速访问最近使用的Bitmap，但是，我们不能依赖这种缓存，假如GridView有很多数据，我们的App很有可能被别的程序（如电话打入）中断，这样，我们在内存中的缓存就很有可能被系统释放掉，这样，当我们的程序Resume时，仍然必须去处理每一个Bitmap。

在这种情况下，我们可以选择把Bitmap存到文件中，每次去加载这些文件，这种情况通常适用于Bitmap很小的情况下，比如MediaStore缓存的图片的缩略图，它把这些缩略图都是存在磁盘文件上，上层通过API去加载这些文件并解析成Bitmap。

注意：一般情况下，文件缓存的操作通常是在后台线程中去做。

9.3 在GridView显示Bitmap

如果在GridView上面显示很多的Bitmap，良好的做法通常是：
(这里GridView与ListView相似)

9.3.1 向GridView注册OnScrollListener

- 通过方法AbsListView#setOnScrollListener()注册。
- 当状态为Fling和Scroll时，设置标志量isBusy = true，从而停止后台线程工作。
- 当状态为Idle时，设置标志量isBusy = false，并唤醒后台线程开始工作。
- 在onScroll()方法中，记录显示的View的索引和总数。

下面代码显示了如何处理OnScrollListener

```
int m_visibleItemCount = 0;
int m_firstVisibleItem = 0;
boolean m_isBusy = false;

private OnScrollListener m_listener = new OnScrollListener() {

    @Override
    public void onScrollStateChanged(AbsListView view,
        int scrollState) {
        switch (scrollState)
        {
            case OnScrollListener.SCROLL_STATE_IDLE:
                m_isBusy = false;
                // Restart the thread to get thumb.
                m_updateThumbThread.restart();
                break;

            case OnScrollListener.SCROLL_STATE_FLING:
            case OnScrollListener.SCROLL_STATE_TOUCH_SCROLL:
                m_isBusy = true;
                break;
        }
    }

    @Override
    public void onScroll(AbsListView view,
        int firstVisibleItem, int visibleItemCount,
        int totalItemCount) {
        m_visibleItemCount = visibleItemCount;
        m_firstVisibleItem = firstVisibleItem;
    }
};
```

9.3.2 后台线程处理Bitmap

后台线程的run()方法处理代码如下

```
public void run() {
    do
    {
        final int nSize = (null != m_imageInfolist) ?
            m_imageInfolist.size() : 0;
        final int pos = m_firstVisibleItem;
        final int count = m_visibleItemCount;
        boolean shouldPause = false;
        ImageInfo outImageInfo = null;

        for (int i = pos; i < nSize && i < (count + pos); ++i)
        {
            outImageInfo = m_imageInfolist.get(i);
            // Only get thumb when the image has not thumb.
            if (null == outImageInfo.getImageThumb()) {
                // Get the image thumb.
                boolean succeed =
                    m_imageSearcher.getImageThumbnail(outImageInfo);
                if (succeed) {
                    // Recycle the image info.
                    ImageInfo removedInfo =
                        m_imageCache.add(outImageInfo);
                    if (null != removedInfo) {
                        // If not hold by any view.
                        if (!isHoldByView(removedInfo)) {
                            removedInfo.recycle();
                        }
                    }
                }
            }

            boolean sendMsg = (i < m_visibleItemCount + pos);
            if (sendMsg && succeed) {
                // Send the message to notify UI to update.
                Message msg = m_imageHandler.obtainMessage(
                    COMPLETE_GET_THUMBNAIL);
                m_imageHandler.sendMessage(msg);
            }
        }

        // If busy or thread stop, break the loop.
        if (m_isBusy || isStop()) {
            break;
        }

        // Set the flag to true, if the for loop ends normally,
        // we should pause the thread.
        shouldPause = true;
    }
}
```

```
    }

    // If the thread is stop,
    // we break the do while loop to exit the thread.
    if (isStop()) {
        break;
    }
    // If the grid view is busy.
    if (m_isBusy || shouldPause) {
        pause();
    }
}
while (true);
}
```

9.4 Image cache设计

在上一小节讲了如何在GridView显示很多图片，为了防止OOM，我们要做很多操作（主动释放bitmap内存），这样做虽然也能达到效果，但是比较复杂，它需要处理AbsListView的一些事件，这样做的话，无法做到共通化，如果在Gallery上面也要显示很多图片，我们仍然需要类似的处理，因此，这种方法没有从根本上解决内存溢出。

我们这里还是以在GridView中显示图片为例来说明cache的设计。

关于cache设计，请思考以下问题：

- 什么数据需要放到cache中？

我们需要把占内存大的对象放到cache之中，这里就是要把bitmap放到cache中。我们把bitmap单独存放在一个地方来管理，这个地方就叫做cache，它的容量是有限的，我们可能会不断地向这个cache中添加bitmap，也可能不断的删除那些没有被引用的bitmap。

Cache的基本设计思路：

- 将bitmap以key - value的形式存在cache中。
- 当Adapter中的view需要bitmap时，始终从cache中去取。
- 如果cache中存在的话，就直接返回。
- 如果cache中不存在的话，就启动thread去加载（从文件、网络等）。
- 用AsyncTask类去加载bitmap。

9.4.1 LruCache

• 这个Cache里面其实就是一个LinkedHashMap，任意时刻，当一个值被访问时，它就会被移动到队列的开始位置，所以这也是为什么要用LinkedHashMap的原因，因为频繁的做移动操作，为了提高性能，所以要用LinkedHashMap。当cache达到容量时，此时再向cache里面添加一个值，那么在队列最后的值就会从队列里面移除，这个值就可能被GC回收掉。

- 如果我们想主动释放内存，也是可以的，我们可以重写entryRemoved(Boolean, K, V, V)方法。
- 这个类是线程安全的，在多线程下面使用这个类，不会存在问题。

```
synchronized (cache) {
    if (cache.get(key) == null) {
        cache.put(key, value);
    }
}
```

- LruCache的APILevel是12，也就是说，我们在SDK 2.3.x以下是无法使用的，不过没关系，LruCache的源码不算复杂，我们可以直接把它拷贝到自己的工程目录就可以了。

9.4.2 AsyncTask<>

这个类是一个很重要的类，它封装了thread和handler，我们使用更加方便，不用关注handler。我们知道，在后台线程中是不能更新UI，而在很多情况下，我们在后台线程做完一件事情后，一般都会更新UI，通常的做法是向关联到UI线程Looper的handler发送message，在handler里面去处理这个message，从而更新UI。

用了AsyncTask<>之后，我们就不用关注handler了，只需要重写几个方法就行了。这个类有以下几个重要的方法：

- onPreExecute()

在UI线程里面调用，它在这个task执行后会立即调用，我们在这个方法里面通常是用于建立一个任务，比如显示一个等待对话框来通知用户。

- doInBackground(Params...)

这个方法从名字就可以看出，它是运行在后台线程的，在这个方法里面，我们做耗时的任务，比如访问网络、操作文件等。在这个方法里面，我们可以调用publishProgress(Progress...)来通知当前任务的进度，调用这个方法之后，onProgressUpdate(Progress...)方法就会调用，它运行在UI线程。

- onProgressUpdate(Progress...)

运行在UI线程，在调用publishProgress()方法之后。这个方法用来在UI上显示任何形式的进度，比如你可以显示一个等待对话框，也可以显示一个文本形式的log，还可以显示toast对话框。

- onPostExecute(Result)

当task结束后调用，它运行在UI线程。

- 取消一个task，我们可以在任何时候调用cancel(Boolean)来取消一个任务，当调用了cancel()方法后，onCancelled(Object)方法就会被调用，onPostExecute(Object)方法不会被调用，在doInBackground(Object[])方法中，我们可以用isCancelled()方法来检查任务是否取消。

- 几点原则

- a) AsyncTask实例必须在UI线程中创建
- b) execute(Params...)方法必须在UI线程中调用。

- c) 不用手动调用onPreExecute(), onPostExecute(), doInBackground(), onProgressUpdate() 方法。
- d) 一个任务只能被执行一次。

9.4.3 IAsyncView

```
public interface IAsyncView
{
    public void setImageBitmap(Bitmap bitmap);
    public void setImageDrawable(Drawable drawable);
    public void setAsyncDrawable(Drawable drawable);
    public Drawable getAsyncDrawable();
}
```

该接口定义了cache模块如何传递bitmap，该类通常由显示在UI上面的View来实现。

9.4.4 ImageWorker

该类封装了cache类，它最主要提供了loadImage()方法来根据数据加载一个bitmap，如果cache中存在，就直接返回，否则启动task来加载bitmap。

```
public void loadImage(Object data, IAsyncView imageView) {
    if (data == null) {
        return;
    }

    Bitmap bitmap = null;

    if (mImageCache != null) {
        bitmap = mImageCache.getBitmapFromMemCache(String.valueOf(data));
    }

    if (bitmap != null) {
        // Bitmap found in memory cache
        imageView.setImageBitmap(bitmap);
        // Here set the drawable to null.
        imageView.setAsyncDrawable(null);
    } else if (cancelPotentialWork(data, imageView)) {
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
```

```

        new AsyncDrawable(mResources, mLoadingBitmap, task);
imageView.setImageBitmap(mLoadingBitmap);
imageView.setAsyncDrawable(asyncDrawable);

// NOTE: This uses a custom version of AsyncTask that has been pulled from the
// framework and slightly modified. Refer to the docs at the top of the class
// for more info on what was changed.
task.executeOnExecutor(AsyncTask.DUAL_THREAD_EXECUTOR, data);
    }
}

```

BitmapWorkerTask内部类的实现如下

```

private class BitmapWorkerTask extends AsyncTask<Object, Void, Bitmap> {
    private Object data;
    private final WeakReference<IAsyncView> imageViewReference;

    public BitmapWorkerTask(IAsyncView imageView) {
        imageViewReference = new WeakReference<IAsyncView>(imageView);
    }

    @Override
    protected Bitmap doInBackground(Object... params) {
        data = params[0];
        final String dataString = String.valueOf(data);
        Bitmap bitmap = null;

        // Wait here if work is paused and the task is not cancelled
        synchronized (mPauseWorkLock) {
            while (mPauseWork && !isCancelled()) {
                try {
                    mPauseWorkLock.wait();
                } catch (InterruptedException e) {}
            }
        }

        // If the bitmap was not found in the cache and this task has not been cancelled by
        // another thread and the ImageView that was originally bound to this task is still
        // bound back to this task and our "exit early" flag is not set, then call the main
        // process method (as implemented by a subclass)
        if (bitmap == null && !isCancelled() && getAttachedImageView() != null

```

```

        && !mExitTasksEarly) {
            bitmap = processBitmap(params[0]);
        }

        // If the bitmap was processed and the image cache is available,
        // then add the processed image cache is available, then add the processed
        // bitmap to the cache for future use. Note we don't
        // check if the task was cancelled
        // here, if it was, and the thread is still running, we may as well add the
        // processed bitmap to our cache as it might be used again in the future
        if (bitmap != null && mImageCache != null) {
            mImageCache.addBitmapToCache(dataString, bitmap);
        }

        return bitmap;
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        // if cancel was called on this task or the "exit early" flag is set then we're done
        if (isCancelled() || mExitTasksEarly) {
            bitmap = null;
        }

        final IAsyncView imageView = getAttachedImageView();
        if (bitmap != null && imageView != null) {
            setImageBitmap(imageView, bitmap);
        }
    }
}

```

9.4.5 如何使用

1) 创建ImageWorker

```

ImageSearchUtil mSearchUtil = null;

String name = "cache_name";
// Create a image cache and set to ImageWorker class.
ImageCache imageCache = new ImageCache(this, name);

```

```

// ImageResizer class extends from ImageWorker.
mImageWorker = new ImageResizer(this);
// Disable the image fade in effect.
mImageWorker.setImageFadeIn(false);
// Set the cache to ImageWorker
mImageWorker.setImageCache(imageCache);
// Set the interface to process bitmap according to the data.
mImageWorker.setProcessBitmapListener(new IProcessBitmapListener()
{
    @Override
    public Bitmap onProcessBitmap(Object arg0)
    {
        if (arg0 instanceof MediaInfo)
        {
            // Load bitmap.
            return mSearchUtil.getImageThumbnail2((MediaInfo)arg0);
        }

        return null;
    }
});

```

上面的代码中，ImageWorker类提供了一个接口IProcessBitmapListener，它的声明如下：

```

public interface IProcessBitmapListener {
    // Process the passed data and return the bitmap.
    public Bitmap onProcessBitmap(Object data);
}

```

这个接口由调用者来实现，因为ImageWorker是一个共通的类，它内部并不知道如何加载一个bitmap，加载bitmap的方式有多种，从网络，从文件等，所以，我们提供一个接口，由调用者来做加载bitmap的事情。

2) 在Adapter中使用ImageWorker

```

@Override
public View getView(int position, View convertView, ViewGroup parent)
{
    GridViewImageView imageView = null;
    if (null == convertView)

```

```
{
    imageView = new GridViewImageView(CacheTestActivity.this);
    imageView.setBackgroundResource(android.R.drawable.picture_frame);
    imageView.setLayoutParams(new AbsListView.LayoutParams(200, 200));
    imageView.setScaleType(ScaleType.FIT_XY);
    convertView = imageView;
}

imageView = (GridViewImageView)convertView;
// Load the bitmap from cache always.
mImageWorker.loadImage(mDatas.getItemAt(position), imageView);

return convertView;
}
```

在getView()方法中，bitmap的始终来自于cache，别的地方不会对这些bitmap有任务引用。

GridViewImageView它实现了IAsyncView接口，代码如下：

```
class GridViewImageView extends ImageView implements IAsyncView
{
    public GridViewImageView(Context context)
    {
        super(context);
    }

    public GridViewImageView(Context context, AttributeSet attrs)
    {
        super(context, attrs);
    }

    public GridViewImageView(Context context, AttributeSet attrs, int defStyle)
    {
        super(context, attrs, defStyle);
    }

    Drawable mDrawable = null;

    @Override
    public Drawable getAsyncDrawable()
    {

```

```
        return mDrawable;
    }

    @Override
    public void setAsyncDrawable(Drawable drawable)
    {
        mDrawable = drawable;
    }
}
```

关于image cache的全部代码，下载地址：

- <http://download.csdn.net/detail/leehong2005/4690627>
- http://192.168.5.243/svn/NJ1S_Works/Demos/Android/19_ImageCache

10. Android 网络操作

10.1 Network

关于网络，请参考：

<http://developer.android.com/training/basics/network-ops/index.html>

使用网络，使用以下的包：

- java.net.*
- android.net.*

使用网络，需要添加以下的权限：

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="
    "android.permission.ACCESS_NETWORK_STATE" />
```

10.1.1 连接到网络

• 选择HTTP Client

大多数Android App都使用HTTP协议来发送并接收数据。Android包含了两个：URLConnection和HttpClient，它们都支持HTTPS，流上传，下载，可配置的超时，IPv6等。对于Gingerbread或更高的系统，推荐使用**URLConnection**。关于URLConnection和HttpClient会在后面详细说明。

• 检查网络连接

在连接到网络之前，我们应当去检查当前网络是否可用，调用ConnectivityManager#getActiveNetworkInfo()和isConnected()方法。注意：设备可能不在网络范围内，用户也有可能禁用了Wi-Fi。

```
public void myClickHandler(View view) {
    ...
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    } else {
        // display error
    }
}
```


- 连接并下载数据

通过URLConnection类，执行GET并下载数据，调用connect()方法后，可以通过调用getInputStream()方法来得到一个InputStream流，通过这个流，我们可以读取网络数据，此时就可以去解析这段流，转换成字符串，或者解码成Bitmap。

```
// Given a URL, establishes an HttpURLConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection)
            url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

        // Makes sure that the InputStream is closed after the app is
        // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

- 处理InputStream

a) 将流转换成Bitmap:

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

b) 将流转换成字符串:

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len)
    throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

c) 解析输入流 (SAX)

```
UpdateParserHandle handler = new UpdateParserHandle();

InputSource is = new InputSource(in);

// Use Simple API for XML to parse the XML file.
SAXParserFactory saxParserFac = SAXParserFactory.newInstance();
XMLReader reader = saxParserFac.newSAXParser().getXMLReader();
reader.setContentHandler(handler);
reader.parse(is);
```

d) 解析输入流 (DOM)

```
DocumentBuilderFactory factory
    = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();
Document documnet = builder.parse(in);
Element root = documnet.getDocumentElement();

parseElement(root, plist, null);
```

e) 解析输入流 (XML Pull)

```
XmlPullParser parser = Xml.newPullParser();
parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
parser.setInput(in, null);
parser.nextTag();

readUpdate(parser);
```

10.1.2 管理网络使用

• 检查设备网络连接

通常情况下，Wi-Fi网络是最快的，通常的策略是当Wi-Fi可用时，才去下载或上传大的数据，所以，我们的应用程序里在使用网络之前，应当先检查设置的网络是否可用。

主要使用两个类：

- `ConnectivityManager`
- `NetworkInfo`

下面代码示例了如何判断Wi-Fi和移动信号是否可用：

```
private static final String DEBUG_TAG = "NetworkStatusExample";

ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getNetworkInfo(
    ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo = connMgr.getNetworkInfo(
    ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

- 监听网络变化

通常是注册Broadcast Receiver, Action是CONNECTIVITY_ACTION。

注意:

程序中实现的NetworkReceiver, 通常不要注册在manifest中, 如果这样的话, 那么程序的APP会在任何时候会系统运行起来, 尽管用户可能根本就没有进行到你的程序中。所以, 通常情况下只需要在你的Main activity中的#onCreate中注册, 在#onDestory中反注册。

```
public class NetworkReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        ConnectivityManager conn = (ConnectivityManager)
            context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = conn.getActiveNetworkInfo();

        // Checks the user prefs and the network connection.
        // Based on the result, decides whether
        // to refresh the display or keep the current display.
        // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi connection.
        if (WIFI.equals(sPref) && networkInfo != null
            && networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
            // If device has its Wi-Fi connection, sets refreshDisplay
            // to true. This causes the display to be refreshed when the user
            // returns to the app.
            refreshDisplay = true;
            Toast.makeText(context, R.string.wifi_connected,
                Toast.LENGTH_SHORT).show();

            // If the setting is ANY network and there is a network connection
            // (which by process of elimination would be mobile), sets refreshDisplay to true.
        } else if (ANY.equals(sPref) && networkInfo != null) {
            refreshDisplay = true;

            // Otherwise, the app can't download content--either because there is no network
            // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and there
            // is no Wi-Fi connection.
            // Sets refreshDisplay to false.
        } else {
            refreshDisplay = false;
            Toast.makeText(context, R.string.lost_connection,
                Toast.LENGTH_SHORT).show();
        }
    }
}
```

```
}  
}
```

10.1.3 解析XML数据

从网络返回的数据，通常都是XML格式组织的，所以我们通过解析XML，从而得到数据。

对于Android，推荐使用XmlPullParser，因为它效率高并且可维护。得到其实例的方法：

- KXmlParser via XmlPullParserFactory.newPullParser().
- ExpatPullParser, via Xml.newPullParser().

具体细节请参考：

<http://developer.android.com/training/basics/network-ops/xml.html>

10.1.4 HttpURLConnection

- 使用这个类遵循以下模式：

- 1，调用URL#openConnection()并强制转换来得到HttpURLConnection对象。
- 2，准备请求，主要属性就是URL，在请求的Header里面可以包含一些元数据，比如安全证书，数据类型和Session cookies。
- 3，如果想要执行POST操作，必须调用**setDoOutput(true)**，然后调用getOutputStream()方法得到一个OutputStream，然后就可以向这个流里面写数据了。
- 4，处理Response，通常情况下，Response Headers里在包含了元数据，比如说数据类型，长度，修改日期，和Session cookies。通过getInputStream()方法得到一个InputStream，然后就可以从这个Stream里面读取数据。如果响应里没有数据体的话，那么这个方法就会返回一个空的Stream。
- 5，断开连接。一旦读取完数据，应当调用**disconnect()**方法来关闭与服务器的连接。释放连接后，被这个连接所引用的资源就可以被关闭或复用。

下面代码示例了得到<http://www.android.com/>的数据

```
URL url = new URL("http://www.android.com/");  
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();  
try {  
    InputStream in = new BufferedInputStream(urlConnection.getInputStream());  
    readStream(in);  
    finally {  
        urlConnection.disconnect();  
    }  
}
```

- HTTPS连接安全性

如果URL的scheme是“https”的话，那么URL.openConnection()返回的是HttpsURLConnection对象，这允许重写HostnameVerifier和SSLSocketFactory。

- 提交内容

为了能提交数据到服务器端，需要调用setDoOutput(true)。

为了更好的性能，

如果提交内容的长度是可知的，调用setFixedLengthStreamingMode(int)，

如果内容的长度不可知，调用setChunkedStreamingMode(int)。

否则，URLConnection就会在发送请求之前强制将提交的数据填充到请求数据体（Request body）的缓存区，这样就可能浪费很多内存并且增加了响应时间。

下面代码示例了如何上传数据

```
URLConnection urlConnection = (URLConnection) url.openConnection();
try {
    urlConnection.setDoOutput(true);
    urlConnection.setChunkedStreamingMode(0);

    OutputStream out = new BufferedOutputStream(
        urlConnection.getOutputStream());

    writeStream(out);

    InputStream in = new BufferedInputStream(urlConnection.getInputStream());
    readStream(in);
    finally {
        urlConnection.disconnect();
    }
}
```

- 性能

URLConnection返回的输入流和输出流是没有填充的。多数情况下都是用BufferedInputStream或BufferedOutputStream来对输入流或输出流包一层。如果你可以一次性操作完输入流或输出流，就可以不用该方式。

当与服务器端交换大量数据时，一般要限定一次读入内存的数据的字节数，除非你需要把所有数据一次性读入到内存中。

默认情况下，URLConnection使用gzip来压缩数据。由于getContentLength()方法返回的是传输的字节数，不能用它来指示已经从输入流中读出了多少字节。相反，当read()方法返回-1时，表示已经读取到输入流到最后了。

同时，可以设置属性来禁用Gzip压缩。

```
urlConnection.setRequestProperty("Accept-Encoding", "identity");
```

- HTTP认证

```
Authenticator.setDefault(new Authenticator() {  
    protected PasswordAuthentication getPasswordAuthentication() {  
        return new PasswordAuthentication(username, password.toCharArray());  
    };  
});
```

注意：

这不是一种安全的用户认证机制，特别的，用户名，密码在请求与响应过程中传输都是没有加密的。

10.1.5 HttpClient

HttpClient是一个抽象的接口，HTTP client封装了一系列对象来执行HTTP请求，比如说处理Cookies，认证，连接管理和其他机能。HTTP client是否线程安全，依赖于特定的client的具体实现与配置。

它的直接子类有：AndroidHttpClient，DefaultHttpClient。

下面代码示例了用一个默认的HttpClient来执行POST请求。

```
try {  
    // 创建一个默认的HttpClient  
    HttpClient httpclient = new DefaultHttpClient();  
    // 创建一个GET请求  
    HttpGet request = new HttpGet("www.google.com");  
    // 发送GET请求，并将响应内容转换成字符串  
    String response = httpclient.execute(request, new BasicResponseHandler());  
    Log.v("response text", response);  
  
    } catch (ClientProtocolException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

注意：

通常情况下，我们不需要每次请求都创建一个HttpClient，我们可以复用HttpClient。如果请求应用在整个程序中的话，我们可以使用单实例类来保证全局一个HttpClient。

在使用HttpClient之前，一般还会对其作一些设置：

- 这里可以用BasicHttpParams类来设置Http协议的参数。

```
HttpParams params = new BasicHttpParams();
// 设置一些基本参数
HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
HttpProtocolParams.setContentCharset(params,CHARSET);
HttpProtocolParams.setUseExpectContinue(params, true);
// 设置User Agent
HttpProtocolParams.setUserAgent(params,
    "Mozilla/5.0(Linux;U;Android 2.2.1;en-us;Nexus One Build.FRG83) "
    + "AppleWebKit/553.1(KHTML,like Gecko) Version/4.0 Mobile
    Safari/533.1");
```

- 设置连接参数

```
// 超时设置
/* 从连接池中取连接的超时时间 */
ConnManagerParams.setTimeout(params, 1000);
/* 连接超时 */
HttpConnectionParams.setConnectionTimeout(params, 2000);
/* 请求超时 */
HttpConnectionParams.setSoTimeout(params, 4000);
```

- 设置支持的协议

```
SchemeRegistry schReg = new SchemeRegistry();
schReg.register(new Scheme("http", PlainSocketFactory
    .getSocketFactory(), 80));
schReg.register(new Scheme("https", SSLSocketFactory
    .getSocketFactory(), 443));

// 使用线程安全的连接管理来创建HttpClient
ClientConnectionManager conMgr = new ThreadSafeClientConnManager(
    params, schReg);

// 创建HttpClient实例
customerHttpClient = new DefaultHttpClient(conMgr, params);
```


- 访问Server (GET/POST)

使用HttpGet或HttpPost来请求与提交数据。

```
String url = "http://wap.kaixin001.com/home/";
List<NameValuePair> params = new ArrayList<NameValuePair>();
params.add(new BasicNameValuePair("email", "firewings.r@gmail.com"));
params.add(new BasicNameValuePair("password", "954619"));

/* 建立HTTPPost对象 */
HttpPost httpRequest = new HttpPost(url);

/* 添加请求参数到请求对象 */
httpRequest.setEntity(new UrlEncodedFormEntity(params, HTTP.UTF_8));

/* 发送请求并等待响应 */
HttpResponse httpResponse = httpClient.execute(httpRequest);

/* 若状态码为200 ok */
if (httpResponse.getStatusLine().getStatusCode() == 200)
{
    /* 读返回数据 */
    strResult = EntityUtils.toString(httpResponse.getEntity());
}
```

- 关闭HttpClient

```
httpClient.getConnectionManager().shutdown();
```

- 中断请求

调用HttpPost或HttpGet类的`abort()`方法来中断请求。该方法声明在其基类HttpRequestBase里面。当请求被中断后，会抛出InterruptedException异常。

10.2 网络安全性

在访问一些服务器时，需要用户凭证，此时我们需要做一些验证的工作。

- HttpClient情况

Step 1: 自定义用于创建安全套接字的对象工厂，这个工厂利用一个X509TrustManager对象信任所有的访问。下面代码示例了如何创建安全证书（信任所有访问）。

```
public class CustomeSSLSocketFactory extends SSLSocketFactory
{
    private SSLContext sslContext = SSLContext.getInstance("TLS");

    public CustomeSSLSocketFactory (KeyStore truststore) throws
        NoSuchAlgorithmException, KeyManagementException,
        KeyStoreException, UnrecoverableKeyException
    {
        super(truststore);

        // Create a truster manager.
        TrustManager tm = new X509TrustManager()
        {
            public void checkClientTrusted(
                X509Certificate[] chain, String authType) throws
                CertificateException { }

            public void checkServerTrusted(
                X509Certificate[] chain, String authType) throws CertificateException { }

            public X509Certificate[] getAcceptedIssuers()
            {
                return null;
            }
        };

        sslContext.init(null, new TrustManager[] { tm }, null);
    }
}
```

```
@Override
public Socket createSocket(Socket socket, String host, int port,
    boolean autoClose) throws IOException, UnknownHostException
{
    return sslContext.getSocketFactory().createSocket(
        socket, host, port, autoClose);
}

@Override
public Socket createSocket() throws IOException
{
    return sslContext.getSocketFactory().createSocket();
}
}
```

Step 2: 应用授权

```
KeyStore trustStore =KeyStore.getInstance(KeyStore.getDefaultType());
trustStore.load(null,null);

SSLSocketFactory sslSocketFactory =
    new CustomeSSLSocketFactory (trustStore);
sslSocketFactory.setHostnameVerifier(
    SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);

SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
    new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));
schemeRegistry.register(
    new Scheme("https", sslSocketFactory, 443));

HttpParams params = new BasicHttpParams();
HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
HttpProtocolParams.setContentCharset(params, HTTP.UTF_8);

DefaultHttpClient client = new DefaultHttpClient(
    new ThreadSafeClientConnManager(params, schemeRegistry), params);
```

- HttpURLConnection情况

Step 1: 定义X509TrustManager对象信任所有的访问

```
private static X509TrustManager xtm = new X509TrustManager()
{
    public void checkClientTrusted(X509Certificate[] chain, String authType)
    { }

    public void checkServerTrusted(X509Certificate[] chain, String authType)
    { }

    public X509Certificate[] getAcceptedIssuers()
    {
        return null;
    }
};
```

Step 2: 应用授权

```
HttpsURLConnection conn = (HttpsURLConnection)url.openConnection();
AllowAllHostnameVerifier HOSTNAME_VERIFIER =
    new AllowAllHostnameVerifier();
X509TrustManager[] xtmArray = new X509TrustManager[] { xtm };

// Trust all certificates
SSLContext context = SSLContext.getInstance("TLS");
context.init(new KeyManager[0], xtmArray, new SecureRandom());
javax.net.ssl.SSLSocketFactory socketFactory =
    context.getSocketFactory();

conn.setSSLSocketFactory(socketFactory);
conn.setHostnameVerifier(HOSTNAME_VERIFIER);
```

11. Android 进程与线程

11.1 进程

当应用程序启动时，Android系统就会为这个应用程序创建一个新的Linux进程，在这个进程里面，有一个主线程。

进程的类型有以下几种，它们的重要性由高到低，重要性越高，越不容易被系统强制杀掉。

• 前台进程

一个进程在满足以下几种情况下会被视为前台进程：

- 它拥有一个正在与用户交互的Activity(Activity#onResume()方法已经被调用)。
- 它拥有一个Service，这个Service绑定到一个正在与用户交互的Activity。
- 它拥有一个运行在前台的Service，也就是说，调用了startForeground()方法。
- 它拥有一个Service，并且正在执行Service的生命周期回调方法，如onCreate()。
- 它拥有一个正在执行onReceive()方法的BroadcastReceiver。

一般情况下，在特定时间下，很少存在有前台进程。如果程序内存低的话，它是最后被杀死的。

• 可视进程

可视进程是指不存在前台的组件，但是它对于用户仍然可见。满足以下情况会被视为可视进程。

- 它拥有一个非前台，但仍然可见的Activity。比如说，如果一个前台进程弹出一个对话框，这种情况下，之前的activity在这个对话框后面，但仍然可见。
- 它拥有一个Service，它绑定到了一个前台或可见的activity上。

• 服务进程

通过调用startService()方法启动Service的进程，就是一个服务进程。虽然服务进程不会直接与用户交互，但是它所做的事情一般情况下都是用户所关心的，所以，系统会尽可能地保持该进程的优先级，除非没有足够的内存来运行这个服务进程了。

• 后台进程

当进程的Activity已经不可见了(onStop()方法已经被调用)，这种进程不会直接影响到用户交互，系统可能在任务时刻杀死后台进程，为前台，可视，服务进程回收内存。因此，我们应当正确实现Activity的生命周期方法，以确认Activity的当前状态，数据能正确存储，这样当activity再次显示时，能还原原来原来的状态。

• 空进程

这种进程不会有任何激活的组件。这种进程存在的唯一原因就是作为缓存，为了改善下次启动要使用的组件的时间。系统经常会杀死空进程。

11.2 使用进程间通信 (IPC)

- **Intent**

Intent是Android里面最基本的概念，它是模块/程序之间通信的基础。

- **Binder和AIDL接口**

通过AIDL接口，也可以与第三方程序通信，常用于Service与其他App通信。
通过Messenger (Ibinder)，也可以向第三方程序发送消息。

- **Broadcast Receiver**

广播是很常见的进程间通信的一种方式，调用sendBroadcast(Intent)方法。

- **Service**

我们可以使用第三个的Service来执行某项任务，同样，我们写的Service也可以让第三方程序调用起来，通过Context.startService或Context.bindService方法。

- **Activity**

我们可以启动第三方程序的Activity，比如可以启动Browser来打开一个URL。

11.3 Android应用程序的入口在哪？

在Android系统框架中，`android.app.Application`类并不是代表应用程序进程，它实际上代表了应用程序的上下文环境，而`ActivityThread`类才是真正的代表应用程序进程。

```
NaiveStart.main()
  ZygoteInit.main
    ZygoteInit$MethodAndArgsCall.run
      Method.Invoke
        method.invokeNative
          ActivityThread.main()
            Looper.loop()
              ....
```

从上面调用栈可以看出，每个应用程序都以`ActivityThread.main()`方法为入口进入到消息循环处理。对于一个进程来讲，我们需要这个闭合的处理框架。

下面是`ActivityThread.main()`方法的实现代码：

```
public static final void main(String[] args) {
    SamplingProfilerIntegration.start();

    Process.setArgV0("<pre-initialized>");

    Looper.prepareMainLooper();
    if (sMainThreadHandler == null) {
        sMainThreadHandler = new Handler();
    }

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

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

    Looper.loop();

    if (Process.supportsProcesses()) {
        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
}
```

```
}  
  
thread.detach();  
String name = (thread.mInitialApplication != null)  
    ? thread.mInitialApplication.getPackageName()  
    : "<unknown>";  
Slog.i(TAG, "Main thread of " + name + " is now exiting");  
}
```

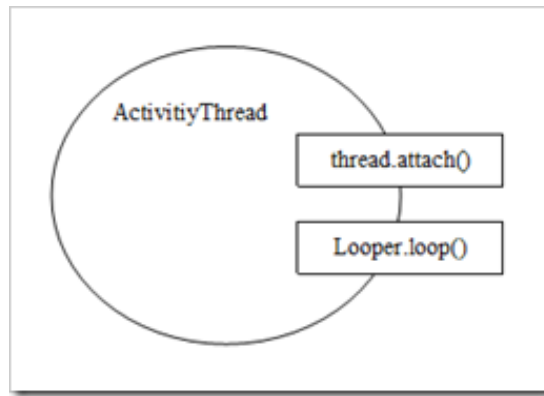


图1, ActivityThread创建消息循环

ActivityThread是应用程序进程空间的重要概念，它建立了应用进程运行的框架，并提供了一个IActivityThread接口与Activity Manager Service（简称为ASM）通信。通过IActivityThread接口Activity Manager Service可以将Activity的状态变化传递到客户端的Activity对象。

Activity Manager Service与应用进程的绑定分为两步：

- AMS建立应用进程，包含ID，名字，线程等。
- 应用进程绑定到AMS并与AMS建立通讯通道（通过IActivityThread）。

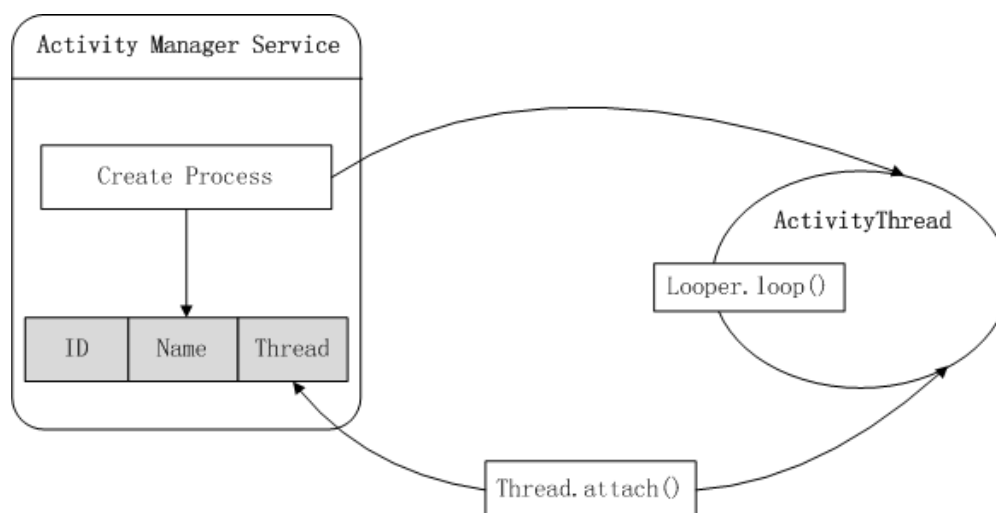


图2, ActivityThread运行框架

从ActivityThread源码中可以看出，在应用程序进程中的Activity都被放置在mActivities 数组中。这些ActivityRecord记录了应用程序进程中所有Activity子类的实例。每个Activity子类实例代表应用程序端与用户交互 的Activity。为了方便管理这些Activity，Activity Manager Service内核中也保留了一些对应的Activity，被称为HistoryRecord，它是activity栈的核心组成部分。

```
final HashMap<IBinder, ActivityClientRecord> mActivities
    = new HashMap<IBinder, ActivityClientRecord>();
```

在Android系统框架中Activity实际上有两个实体。一个在应用进程中与负责与用户交互的Activity，另外一个是在Activity Manager Service中具有管理功能的History Record。应用进程中的Activity 都记录在ActivityThread实例中的mActivity 数组中，而Activity Manager Service中的HistoryRecord实例放置在mHistory 栈中。mHistory 栈是Android 管理Activity的场所，处于栈顶的Activity就是用户看到的当前处于活动状态的Activity。

Activity的内核实体是依附在ProcessRecord的成员变量中，通过ProcessRecord可以访问到所有属于该Process 的Activity。通过IActivityThread接口，可以访问到它对应的Activity的方法。在Launch Activity时，Activity Manager Service将对应的HistoryRecord作为token传递到客服端与客服端的Activity建立联系。当Activity Manager Service中的Activity状态变化时，Activity Manager Service找到客服端的Activity，从而将消息或者动作传递应用程序中对应的Activity。

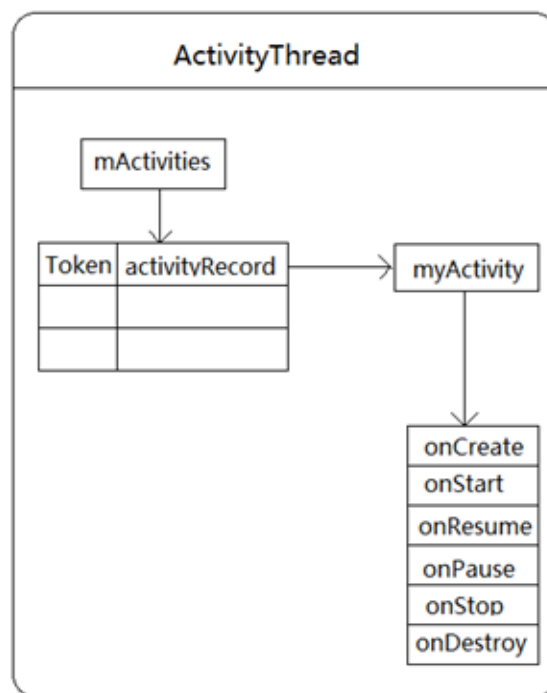


图3, ActivityThread管理activity的内部结构图

总结:

- 在客户端和服务端分别有一个管理activity的地方，服务端是在mHistory中，处于mHistory栈顶的就是当前处于running状态的activity，客户端是在mActivities中。
- 在startActivity时，首先会在ActivityManagerService中建立HistoryRecord，并加入到mHistory中，然后通过scheduleLaunchActivity在客户端创建ActivityRecord记录并加入到mActivities中。最终在ActivityThread发起请求，进入消息循环，完成activity的启动和窗口的管理等。

11.4 线程

当应用程序启动时，系统就会创建一个主线程，也叫UI线程，这个线程非常重要，因为它负责分发事件到恰当的UI，包括绘制，Touch事件等。用户与UI的交互，也是在这个线程中完成。

Android系统不会为每个组件实例分配一个单独的线程，所有的组件都是运行在主线程中，系统对于每个组件的调用，都是在这个线程中完成，因此，响应系统回调的方法(如onKeyDown()方法)总是运行在UI线程中。

注意：

- 不要阻塞UI线程（可能会出现ANR）。
- 不要在工作线程中操作UI的元素（如调用View#invalidate()方法）。

11.4.1 工作线程

当我们要做一些比较耗时的操作时（如读取文件，网络下载等），通常的做法是在后台线程中去做，这样UI线程不会被阻塞。

• 创建工作线程

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            mImageView.setImageBitmap(b);    // Is it valid?
        }
    }).start();
}
```

上面代码示例了创建一个工作线程，在线程执行体（Runnable）中，从网络上下载一张图片，然后将这个Bitmap设置到ImageView中。

上面的代码中，mImageView.setImageBitmap(b); 它违反了[不要在工作线程中更新UI的View]这一原则——它在工作线程中去修改ImageView，这可能出现未知情况。

它可能会抛出异常：

CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.

正确的做法如下：

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap =
                loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

这里调用了View.post(Runnable)方法修正上述问题。当然还可以用Handler的方式。

- 与UI线程交互

- View.post(Runnable)

将Runnable添加了消息队列中，操作具体的View时可以用这种方式。

- Handler

发送消息到消息队列，这种方式很常用，它不依赖于某一个View。这个Handler必须是关联到UI线程的消息循环。

调用Handler.post(Runnable)方法，该方法不用定义消息，这个Runnable运行在与Handler相关联的线程上，也就是说，如果Handler关联到主线程，那么Runnable运行在主线程，如果Handler关联到工作线程，那么Runnable运行在工作线程。

11.4.3 为什么在工作线程中不能更新UI?

如果在工作线程中去更新UI，会抛出CalledFromWrongThreadException这个异常，它是由下面代码抛出的。

```
void checkThread() {
    if (mThread != Thread.currentThread()) {
        throw new CalledFromWrongThreadException(
            "Only the original thread that created a view hierarchy can touch its views.");
    }
}
```

checkThread()是ViewRoot类的方法，ViewRoot.java的路径是

framework/base/core/java/android/view/ViewRoot.java

ViewRoot它实现了ViewParent，是View层次结构的根，请看ViewRoot的构造方法的实现：

```
public ViewRoot(Context context) {
    super();

    if (MEASURE_LATENCY && lt == null) {
        lt = new LatencyTimer(100, 1000);
    }

    // For debug only
    //++sInstanceCount;

    // Initialize the statics when this class is first instantiated. This is
    // done here instead of in the static block because Zygote does not
    // allow the spawning of threads.
    getWindowSession(context.getMainLooper());

    mThread = Thread.currentThread();
    mLocation = new WindowLeaked(null);
    mLocation.fillInStackTrace();
    mWidth = -1;
    mHeight = -1;
    mDirty = new Rect();
    mTempRect = new Rect();
    mVisRect = new Rect();
    mWinFrame = new Rect();
}
```

```

mWindow = new W(this, context);
mInputMethodCallback = new InputMethodCallback(this);
mViewVisibility = View.GONE;
mTransparentRegion = new Region();
mPreviousTransparentRegion = new Region();
mFirst = true; // true for the first time the view is added
mAdded = false;
mAttachInfo = new View.AttachInfo(sWindowSession, mWindow, this, this);
mViewConfiguration = ViewConfiguration.get(context);
mDensity = context.getResources().getDisplayMetrics().densityDpi;
}

```

上面最关键的是 `mThread = Thread.currentThread()` ;这行代码，它取得当前线程。

再看一看 `checkThread()` 方法的实现，它判断 `mThread` 是否等于 `Thread.currentThread()`，试想，`View` 都是在 `UI` 线程里面创建的，也就是说 `mThread = UI Thread`，如果你在工作线程里面去更新 `UI`，那么 `checkThread()` 方法就是运行在工作线程中，`Thread.currentThread()` 返回的是工作线程，因此，`checkThread()` 中的判断条件不满足，就抛出异常了。

最后，我们看看 `ViewRoot.checkThread()` 的调用顺序：

```

com.david.test.helloworld.MainActivity$TestThread2.run
-> android.widget.TextView.setText
  -> android.widget.TextView.checkForRelayout
    -> android.view.View.invalidate
      -> android.view.ViewGroup.invalidateChild
        -> android.view.ViewRoot.invalidateChildInParent
          -> android.view.ViewRoot.invalidateChild
            -> android.view.ViewRoot.checkThread

```

总结：

- `checkThread()` 方法在更新 `UI` 之前会被调用。
- 实例化 `ViewRoot` 的线程与调用 `checkThread()` 方法的线程必须是相同，否则抛出异常。

那么，工作线程能不能更新 `UI` 呢？

答案是能，只要我们保证实例化 `ViewRoot()` 的线程与更新 `View` 的线程是同一个线程就可以了。

```
class TestThread1 extends Thread {  
    @Override  
    public void run() {  
        Looper.prepare();  
  
        TextView tx = new TextView(MainActivity.this);  
        tx.setText("test111111111111111111");  
  
        WindowManager wm = MainActivity.this.getWindowManager();  
        WindowManager.LayoutParams params =  
            new WindowManager.LayoutParams(  
                250, 250, 200, 200,  
                WindowManager.LayoutParams.FIRST_SUB_WINDOW,  
                WindowManager.LayoutParams.TYPE_TOAST, PixelFormat.OPAQUE);  
  
        wm.addView(tx, params);  
  
        Looper.loop();  
    }  
}
```

在Activity.onCreate()方法中启动线程，在线程中创建的TextView最终会添加到屏幕上。

WindowManager.addView()方法中，会创建一个ViewRoot，因此，在ViewRoot里面保存的mThread其实就是工作线程的实例了。

总结：

- 子线程是能够更新UI的，这里只是研究线程相关的东西。
- 我们应该遵守google的文档描述，更新UI始终都在UI线程中完成。

11.4.4 消息循环

这里有三个很重要的概念：

- **Looper**

默认的工作线程是没有与之关联的消息循环，可以通过Looper类来运行消息循环。进入消息循环后，该消息循环会一直处理消息，直到退出消息循环。

调用Looper.prepare()方法来创建消息循环，调用Looper.loop()方法来开始消息循环。

下面代码示例了如何实现一个带有消息循环的线程类，这个线程类关联了一个Handler。

```
class LooperThread extends Thread {
    public Handler mHandler;

    public void run() {
        Looper.prepare();

        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // process incoming messages here
            }
        };

        Looper.loop();
    }
}
```

- **Message**

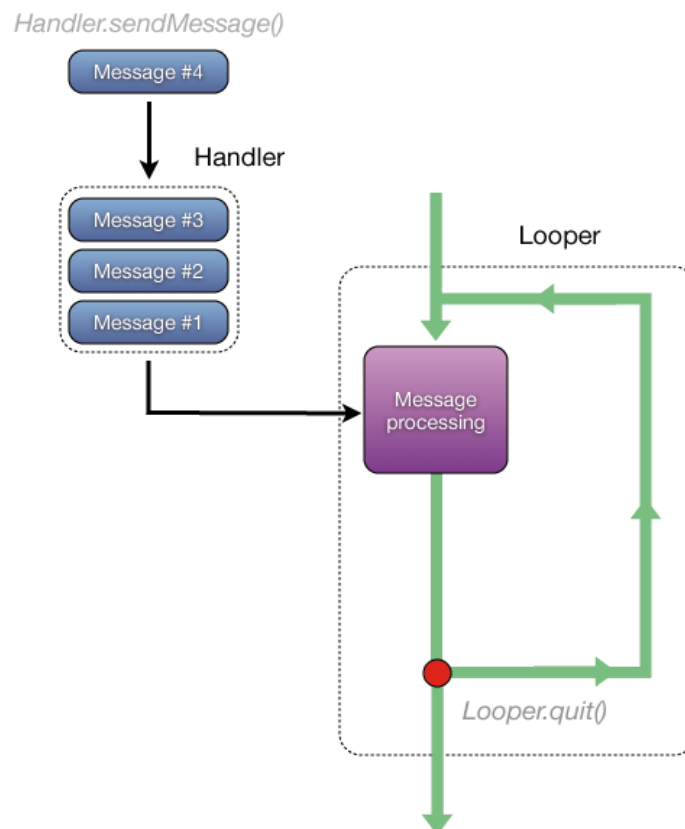
发送到消息队列的消息类，它封装了消息值，参数，数据对象等。得到Message实例的最好方式是调用**Message.obtain()**方法或**Handler.obtainMessage()**方法，这些方法会把消息放到一个可循环利用的对象池中。

• Handler

一个Handler类允许你发送并处理消息和Runnable对象，这些消息和Runnable对象都是关联到线程的消息队列。

每一个Handler的实例会关联一个线程和这个线程的消息队列。也就是说，如果这个Handler里面关联的是主线程的消息队列，那么对这个Handler来发送消息，那么其实就是发送到主线程的消息队列中。

下面图示了Looper，Message和Handler之间的关系



图二，Looper，Handler和Message的关系

- 通过Handler.sendMessage()发送一条消息到消息队列中。
- 通过Handler.removeMessage()从消息队列中删除一条消息。
- 线程通过Looper接收消息，然后调用Handler.handleMessage()来处理消息。
- 调用Looper.quit()方法退出消息循环，从而结束线程。

注意：

- 发送消息：handler.obtainMessage(what).sendToTarget();
- 给Handler指定消息循环：Handler(Looper looper)

11.4.5 线程的暂停与唤醒

假如我们设计一个MyThread类，它继承了Thread。

它提供的核心方法有：

- pause()
- restart()

这两个方法的实现如下：

```
public void pause() {  
  
    try {  
        synchronized (this) {  
            wait();  
        }  
    }  
    catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
  
public synchronized void restart()  
{  
    if (Thread.State.WAITING == getState())  
    {  
        // notify anybody waiting on "this"  
        notify();  
    }  
}
```

11.4.6 工作线程设计示例

这里示例了如何创建一个工作线程，这个线程里面创建了一个消息循环。

ThreadWorker类，它内部封装了线程，该线程拥有消息循环，实现了Runnable接口，它提供的方法有：

- ThreadWorker(String name)
- Runnable.run()
- getLooper()
- quit()

ThreadWorker(String name)

在构造方法里面启动了一个线程，它会等待直到消息循环创建成功。

```
public ThreadWorker(String name) {  
  
    // Start a thread.  
    Thread t = new Thread(null, this, name);  
    t.setPriority(Thread.MIN_PRIORITY);  
    t.start();  
  
    synchronized (m_lockObj) {  
  
        // Wait for until the loop succeeds to prepare.  
        while (null == m_looper) {  
  
            try {  
                m_lockObj.wait();  
            }  
            catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

在构造方法中，开始一个线程，并且一直等待，直到成功创建消息循环后才返回，也就是说，当构造方法返回时，其内部线程的消息循环已经创建好了。

Runnable.run()

在这个方法中，它创建消息循环，并开始消息循环。

```
@Override
public void run() {

    synchronized (m_lockObj) {

        // Initialize the looper of the current thread.
        Looper.prepare();

        // Get the looper.
        m_looper = Looper.myLooper();
        // Notification all waiting object.
        m_lockObj.notifyAll();
    }

    // Begin the loop cycle.
    Looper.loop();
}
```

getLooper()

它返回与这个线程关联的消息循环。

```
public Looper getLooper() {
    return m_looper;
}
```

quit()

调用这个方法退出消息循环。

```
public void quit() {
    m_looper.quit();
}
```

- 如何使用ThreadWorker类

```
// Create a thread, after the ThreadWorker() method finish,
// its loop has been created.
ThreadWorker albumWorker = new ThreadWorker("Album_thread_worker");

// The handler is associated to the loop of ThreadWorker thread.
Handler albumHandler = new AlbumArtHandler(m_albumWorker.getLooper());
```

当要通过工作线程做一些事情时，只需要向AlbumArtHandler发送消息报。

```
// First remove the message from loop
albumHandler.removeMessages(GET_ALBUM_ART_MESSAGE);

// Send the message to loop.
albumHandler.obtainMessage(GET_ALBUM_ART_MESSAGE,
    (int)albumId, id).sendToTarget();
```

注意：

这个AlbumArtHandler它是关联到工作线程，不要直接在Handler.handleMessage()方法里面访问UI元素。

- 什么时候使用ThreadWorker？

假如对于一个音乐播放器，它界面上会显示歌曲的专辑插图，取插图的工作可能很耗时，我们可能会怎么做？

- 直接在UI线程中完成（可能会阻塞UI）
- 启动一个线程去取插图，完成后更新UI

但是，如果用户快速地切换歌曲，那么程序可能会创建很多个线程，这可能会性能会变差，很多行为不可控制，此时应当怎么办？

合理的做法是：

只创建一个线程，当取完插图后，让这个线程挂起（由于Thread拥有looper，消息队列里面没有消息，该线程会自动挂起），需要再取插图时，再唤醒这个线程。这里就可以用到这个ThreadWorker类，需要取插图时，通过Handler发送一条消息，然后在这个Handler里面处理这个消息，**注意这里的Handler是关联到后台线程的，不能直接这它里面更新UI的View。**

11.4.7 异步链式调用的设计

考虑如下情况：

情况1：

访问网络（或其他耗时的事情）。通常的做法是：

- 显示一个ProgressDialog对话框，提示用户。
- 启动工作线程来执行耗时操作。
- 发送消息到关联到主线程的Handler里面，关闭对话框。

情况2：

从网络下载一个zip文件，下载完成之后，询问用户是否执行解压操作。通常的合理做法：

- 显示一个ProgressDialog对话框，提示用户。
- 启动线程执行下载操作。
- 发送消息到关联到主线程的Handler里面，关闭对话框，然后启动一个询问对话框。
- 用户如果点击[YES]，显示一个ProgressDialog对话框。
- 启动用线程执行解压操作。
- 发送消息到关联到主线程的Handler里面，关闭对话框。

实现这两种情况缺点：

- 定义Handler，发送消息，使得代码变得复杂，不易理解。
- 发送消息是异步处理，在某些情况下可能需要做等待操作。
- 流程执行混乱，不是流水作业。

基于以上情况，我们能不能也像流水线的操作那么调用我们的回调(Callback)，使用者只关心第一步干什么，第二步干什么，如果能这样的话，那么在哪步做什么都能明确定义出来，这就是链式调用。

请看下面的链式调用的写法(JavaScript)：

```
Async.go(initialArgument)
  .next(firstAsyncOperation)
  .next(secondAsyncOperation)
  .next(thirdAsyncOperation)
  .next(function(finalResult) { alert(finalResult); })
```

用户只需要添加每一步的task到一个队列里面，然后执行，这些task就会按添加的顺序执行，从而实现链式调用。

我们能不能设计出一个Android版本的异步链式调用的模块呢，请看下面。

- **Task**

我们抽象出每一步要做的事情，定义一个Task类，它是一个抽象类，有如下核心属性和方法：

- `mRunInBackground`

用来指示这个Task是运行在后台线程还是运行在主线程。

- `onExecutor(TaskOperation)`

我们需要实现该方法，在这里面执行我们想要做的事情。

- `onProgressUpdate(Object)`

我们可以重写该方法，来更新我们所做事情进度，这个方法运行在主线程。

- **TaskOperation**

- 1) 这个类里面包含了task的运行参数，上一个task的输出将会作为下一个task的输入。
- 2) 它可以指示继续或暂停执行下一个task。

- **TaskManager**

- 1) 管理task队列，始终从队列第一个开始执行，执行一个task后，这个task将从队列出移除。
- 2) 内部创建了一个带有消息循环的线程。
- 3) 执行task时，判断其运行的线程环境，如果运行在UI线程，发送消息到UI的Handler来执行。
- 4) 内部封装了Handler，用户不用关心是否发送消息。
- 5) 核心方法有：

- `next(Task)`
 - `execute()`
 - `execute(TaskOperation)`
 - `cancelCurrentTask()`
 - `removeTasks()`
 - `publishProgress(Object)`

这里只是给了一个最基本的设计思路，现在该设计还有完善的地方，具体的实现请参考相关的代码和测试工程。

源码下载：

http://192.168.5.243/svn/NJ1S_Works/Demos/Android/23_CoreLib

12. Android 四大组件

TBD

13. Android 多分辨率与多DPI

TBD

14. Android 程序发布

TBD

15. 设计模式

TBD

16. Chapter

TBD