目录

[第一财经APP项目重构建议 3](#_Toc24900)

[现在第一财经APP存在的问题（从技术的角度看） 3](#_Toc27133)

[怎样的APP才是一款好的APP 3](#_Toc6327)

[从不同的角度判断一款好的APP 3](#_Toc6156)

[项目架构的根本原则 4](#_Toc1263)

[项目架构层级 4](#_Toc5456)

[接口层 5](#_Toc22529)

[核心层 7](#_Toc2468)

[界面层 8](#_Toc18443)

[模型层 9](#_Toc4501)

[界面篇 9](#_Toc27302)

[规范性 9](#_Toc12950)

[单一性（低耦合） 10](#_Toc17616)

[简洁性 11](#_Toc26036)

[如何优化内存总结 13](#_Toc31552)

[减小对象的内存占用 13](#_Toc5415)

[1）使用更加轻量的数据结构 13](#_Toc11842)

[2）避免在Android里面使用Enum 14](#_Toc2866)

[3）减小Bitmap对象的内存占用 14](#_Toc9613)

[4）使用更小的图片 14](#_Toc1323)

[内存对象的重复利用 14](#_Toc21427)

[1）复用系统自带的资源 15](#_Toc29607)

[2）注意在ListView/GridView的优化 16](#_Toc13845)

[3）Bitmap对象的复用 16](#_Toc31003)

[避免对象的内存泄露 18](#_Toc32137)

[1）注意Activity的泄漏 19](#_Toc626)

[2）考虑使用Application Context而不是Activity Context 20](#_Toc3341)

[3）注意临时Bitmap对象的及时回收 20](#_Toc29032)

[4）注意监听器的注销 20](#_Toc1857)

[5）注意缓存容器中的对象泄漏 21](#_Toc20139)

[6）注意WebView的泄漏 21](#_Toc10364)

[7）注意Cursor对象是否及时关闭 21](#_Toc27151)

[内存使用策略优化 21](#_Toc23220)

[1）谨慎使用large heap 21](#_Toc16729)

[2）综合考虑设备内存阈值与其他因素设计合适的缓存大小 22](#_Toc6556)

[3）onLowMemory()与onTrimMemory() 22](#_Toc2670)

[4）资源文件需要选择合适的文件夹进行存放 24](#_Toc12441)

[5）Try catch某些大内存分配的操作 24](#_Toc25573)

[6）谨慎使用static对象 24](#_Toc27011)

[7）特别留意单例对象中不合理的持有 25](#_Toc1877)

[8）珍惜Services资源 25](#_Toc14924)

[9）优化布局层次，减少内存消耗 25](#_Toc29391)

[10）谨慎使用“抽象”编程 25](#_Toc12546)

[11）使用nano protobufs序列化数据 25](#_Toc27489)

[12）谨慎使用依赖注入框架 26](#_Toc6350)

[13）谨慎使用多进程 26](#_Toc2208)

[14）使用ProGuard来剔除不需要的代码 27](#_Toc26574)

[15）谨慎使用第三方libraries 27](#_Toc14725)

[16）考虑不同的实现方式来优化内存占用 27](#_Toc17676)

# 第一财经APP项目重构建议

## 现在第一财经APP存在的问题（从技术的角度看）

1. 整个项目结构不明朗；
2. 整个项目没有按照各个模块独立出来；
3. 整个项目对于常量的管理很混乱；
4. **整个APP运行起来的时候内存没有优化好有卡顿的现象（偶现闪退）；**
5. 整个项目命名不规范；
6. 在布局文件中直接写数字、字符串、color没有写到values中去；
7. **动画卡顿，跳转衔接不流畅；**
8. 对于style的运用不到位；
9. **APP中有些图标在高分辨率的手机上运行起来很模糊；**
10. **有些界面在低分辨率手机上显示不全；**
11. APP布局重用不够，重复布局代码太多；
12. 布局代码冗长过度重绘严重。

## 怎样的APP才是一款好的APP

### 从不同的角度判断一款好的APP

#### 从用户的角度来说

* 能够解决用户实际问题；
* 产品结构简单，信息架构清晰，便捷容易上手；
* 交互优雅，有整体的风格；
* 核心功能突出，次要功能含蓄 ；
* **运行足够流畅，不会经常崩溃和访问速度慢；**
* **有合理的提示以及过度动画；**
* **产品有趣味性，互动性。**

#### 开发的角度来说

* 整个项目具有可扩展性的明朗的项目架构（项目架构的搭建）；
* 运行足够流畅，不会经常崩溃和访问速度慢（内存优化）；
* 适配不同的手机设备包括不同分辨率和不同内存大小的手机设备（APP的兼容性）；
* 有合理的提示以及过度动画（优秀的衔接动画）；
* 减少用户没必要的操作，替用户做某些事儿（保存用户的痕迹）。

## 项目架构的根本原则

项目架构的根本原则:**高内聚低耦合**

内聚：一个模块内各个元素彼此结合的紧密程度，高内聚就是一个模块内各个元素彼此结合的紧密程度高。

所谓高内聚是指一个软件模块是由相关性很强的代码组成，只负责一项任务，也就是常说的**单一责任原则**。

耦合：一个软件结构内不同模块之间互连程度的度量(耦合性也叫块间联系。指软件系统结构中各模块间相互联系紧密程度的一种度量。模块之间联系越紧密，其耦合性就越强，模块的独立性则越差，模块间耦合的高低取决于模块间接口的复杂性，调用的方式以及传递的信息。)

## 项目架构层级

项目分为了四个层级：

1. 模型层、

模型层定义了所有的模型；

1. 接口层、

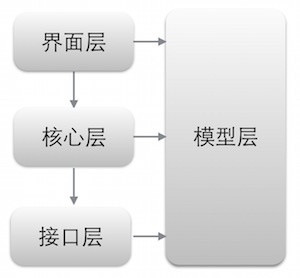
接口层封装了服务器提供的API；

1. 核心层、

核心层处理所有业务逻辑；

1. 界面层。

界面层就处理界面的展示。

几个层级之间的关系如下图所示：  


### 接口层

接口层封装了网络底层的API，并提供给核心层调用。比如：

PostEngine，请求引擎类，对请求的发送和响应结果进行处理；

Response，响应类，封装了Http请求返回的数据结构；

Api，接口类，定义了所有接口方法；

ApiImpl，接口实现类，实现所有接口方法。

PostEngine将请求封装好发送到服务器，并对响应结果的json数据转化为Response对象返回。Response其实就是响应结果的json数据实体类，json数据是有固定结构的，分为三类，如下：

{**"event"**:"0",**"msg"**:"success"}

{**"event"**:"0",**"msg"**:"success",**"obj"**:{...}}

{**"event"**:"0",**"msg"**:"success",**"objList"**:[{...},{...}],**"currentPage"**:1,**"pageSize"**:20,**"maxCount"**:2,**"maxPage"**:1}

event为返回码，0表示成功，msg则是返回的信息，obj是返回的单个数据对象，objList是返回的数据对象数组，currentPage表示当前页，pageSize则表示当前页最多对象数量，maxCount表示对象数据总量，maxPage表示总共有多少页。根据此结构，Response基本的定义如下：

**publicclassResponse**<T>{

**private**Stringevent;

**private**Stringmsg;

**private**Tobj;

**private**TobjList;

**private**intcurrentPage;

**private**intpageSize;

**private**intmaxCount;

**private**intmaxPage;

}

每个属性名称都要与json数据对应的名称相一致，否则无法转化。obj和objList用泛型则可以转化为相应的具体对象了。

Api接口类定义了所有的接口方法，方法定义类似如下：

**public**Response<Void>login(StringloginName,Stringpassword);

**public**Response<VersionInfo>getLastVersion();

**public**Response<List<Coupon>>listNewCoupon(intcurrentPage,intpageSize);

ApiImpl则实现所有Api接口了，实现代码类似如下：

**@Override**

**public**Response<Void>login(StringloginName,Stringpassword){

**try**{

Stringmethod=Api.LOGIN;

List<NameValuePair>params=**new**ArrayList<NameValuePair>();

params.add(**new**BasicNameValuePair("loginName",loginName));

params.add(**new**BasicNameValuePair("password",EncryptUtil.makeMD5(password)));

TypeToken<Response<Void>>typeToken=**new**TypeToken<Response<Void>>(){};

**return**postEngine.specialHandle(method,params,typeToken);

}**catch**(Exceptione){

*//异常处理*

}

}

实现中将请求参数和返回的类型定义好，调用PostEngine对象进行处理。

### 核心层

核心层介于接口层和界面层之间，主要处理业务逻辑，集中做数据处理。向上，给界面层提供数据处理的接口，称为Action；向下，调用接口层向服务器请求数据。向上的Action中定义的方法类似如下：

**public**voidgetCustomer(StringloginName,CallbackListener<Customer>callbackListener);

这是一个获取用户信息的方法，因为需要向接口层请求服务器Api数据，所以添加了callback监听器，在callback里对返回的数据结果进行操作。CallbackListener就定义了一个成功和一个失败的方法，代码如下：

**publicinterfaceCallbackListener**<T>{

*/\*\**

*\* 请求的响应结果为成功时调用*

*\* @param data 返回的数据*

*\*/*

**public**voidonSuccess(Tdata);

*/\*\**

*\* 请求的响应结果为失败时调用*

*\* @param errorEvent 错误码*

*\* @param message 错误信息*

*\*/*

**public**voidonFailure(StringerrorEvent,Stringmessage);

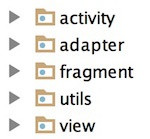
}

接口的实现基本分为两步：

1. 参数检查，检查参数的合法性，包括非空检查、边界检查、有效性检查等；
2. 使用异步任务调用接口层的Api，返回响应结果。

需要注意的是，Action是面向界面的，界面上的数据可能需要根据不同情况调用不同的Api。  
后续扩展可以在这里添加缓存，但也要视不同情况而定。

### 界面层

界面层处于最上层，其核心就是负责界面的展示。  
因为产品需求的变化，可能这里就需要建立多个app的界面，这是一个很繁琐的事情。  
界面层package的定义可以根据不同类型划分，主要分为以下几个包：  
  
其中，activity、adapter、fragment各自都有一个基类，做统一的处理，比如定义了一些共用的常量、对象和方法等。  
界面层是最复杂，最容易变得混乱不堪，最容易出问题的层级。

### 模型层

模型层横跨所有层级，封装了所有数据实体类，基本上也是跟json的obj数据一致的，在接口层会将obj转化为相应的实体类，再通过Action传到界面层。另外，模型层还定义了一些常量，比如用户状态、阅读状态等，对于一些常量的定义时一定要写好注释同时管理好常量的分布。

## 界面篇

将项目分为了四个层级：模型层、接口层、核心层、界面层。其中，最上层的界面，是变化最频繁的一个层面，也是最复杂最容易出问题的一个层面，如果规划不好，很容易做着做着，又乱成一团了，下面就只从界面层级介绍构建界面层级时遵循的原则。  
要规划好界面层，至少应该遵循几条基本的原则：

1. 保持规范性：定义好开发规范，包括书写规范、命名规范、注释规范等，并按照规范严格执行；
2. 保持单一性：布局就只做布局，内容就只做内容，各自分离好；每个方法、每个类，也只做一件事情；
3. 保持简洁性：保持代码和结构的简洁，每个方法，每个类，每个包，每个文件，都不要塞太多代码或资源，感觉多了就应该拆分（每个方法完成独立的任务；每个类都只代表独立的对象抽象；每个文件负责独立的模块）。

### 规范性

每个人的编码习惯和风格都不同，不说那些缺乏良好编码习惯的开发人员，就连那些已经养成良好编码习惯的人员，很多方面都会不同。比如缩进，有的喜欢4个空格，有的喜欢两个空格；比如变量名，有的喜欢m开头，例如mValue，有的喜欢直接就命名为value。如果不设定好规范，让每个人都按照自己的习惯和风格去编码，久了肯定乱，尤其当团队中存在还没养成良好编码习惯的人员时，更容易乱。所谓无规矩不成方圆，若无规范，久必乱。定义好规范，才能统一风格，才可提高代码可读性，同时也提高了维护性，还减低了引入bug的机会。

#### 缩进

很多人都习惯用Tab缩进，不管是规范4个空格还是2个空格，统一设置好Tab缩进的size就好了，这样就不用让每个人都去敲空格。

#### 命名

一个好的命名，一眼就可以从名字中看到它是干嘛的，做什么用，什么类型等等。举个id命名的例子，看到有些团队喜欢将一些控件缩写，比如TextView缩写为tv，ListView缩写为lv，这种缩写倒是挺简洁的，但是并不能一眼就能看出它是什么，对于不熟悉的人来说，谁知道tv和lv是什么啊，还不如用text和list更明确些。较为规范的id命名结构为：控件\_范围\_功能，例如：edit\_login\_password，这是一个登录页的密码输入框。

#### 单位

文字大小的单位应该统一用sp，其他元素用dp。因为这两个单位是与设备分辨率无关的，能够解决在不同分辨率设备上显示效果不同的问题。另外，SDK里面，对文字大小系统默认是用sp单位的，但其他元素单位默认却不是dp，而是px的，同时也没有提供dp的设置接口，所以，自己写两个dp和px转换的方法是很有必要的。

最重要的并不在于规范怎么定义，而是在于规范的严格执行。如果规范定义好了，但却不遵守，那规范就等于形同虚设，因此，规范一旦设定，就要严格执行。

### 单一性（低耦合）

我们都知道，面向对象设计中，有一个基本原则就是单一职责原则，它规定一个类应该只有一个发生变化的原因。而这里说的单一性，不只是规定类，也规定了方法、包，甚至到最大层面的分层架构。**保持单一性是降低耦合度的关键标准，其目的就是各方面的解耦。架构上的分层就是最大层面的解耦，而方法上的单一就是最小层面的解耦了**。

#### 界面的单一

界面上的单一，**首先是界面的布局和界面的数据应该分离。**这一点，Android已经用layout和Activity做好解耦了，我们只要确保用layout文件排好布局，在Activity展示数据就好了。另外，**界面数据的获取和展示也应该分离。**很多开发团队习惯将数据的获取和展示都放在Activity或Fragment里完成的，请求接口、获取数据、检查数据、显示数据更新UI，全都在界面上完成的。这样子的话，当数据的获取发生改变时，比如要添加缓存，这时候界面就需要改动了，当数据的展示也需要修改时，比如某个控件要展示其他数据，界面也一样需要改动，也就是说，界面上已经有两个发生变化的原因，这就违反了单一职责原则。  
界面上的单一，**就是要保持界面上每个维度都做好分离，从界面的布局，到数据的获取，数据的检查，数据的展示。**

#### 包和类的单一

定义包之前，**需要先想好它的职责是什么，明确定义并确保它只有一个职责**。

com.keegan.activity，就是activity类的包，不会有其他组件；

com.keegan.adapter，就是存放各种适配器的包；

com.keegan.util就是工具包了。

同样，类的定义，也是需要明确它的单一职责。有些人习惯将adapter写在Activity里，因为觉得这个adapter只在这个Activity里用到，没必要再把它独立出来。这么做了一段时间之后，觉得实在糟糕透了，重复的代码无法复用，界面上的一点小需求调整时，很多代码需要跟着调整。后来，进行了一番重构，将所有adapter独立了出来，并抽象出了一个adapter的基类，自此，当需要再添加adapter时，编写的代码量大大减少了，当界面需求调整时，修改的地方也大大减少了。所以，不要让一个类做太多事情，要分离好各种元素，每个元素只做一件简单的事。

#### 方法的单一

方法的单一，**表现为一个方法是对一个行为的封装**。然而，一个行为又可以拆分为多个步骤，每个步骤其实也是一个更细的行为，又可以封装成一个新的方法。因此，方法嵌套方法是一种常态。那么，保持方法的单一性，关键并不在于怎么定义这个方法的行为，而在于这个行为要怎么拆分成更细的行为。举个例子，通常在Activity的onCreate方法，做数据的初始化，细分出来就分为了：**控件的初始化、逻辑变量的初始化、数据的加载和展示。数据的加载和展示可以再细分：从缓存加载数据、从网络加载数据、展示数据。每个细化的行为都应该封装为一个独立的方法，这样，才真正符合方法的单一性**。

#### 资源文件的单一

Android提供了各种资源文件，strings.xml用来存储字符串，arrays.xml用来存储字符串数组，colors.xml用来存储颜色值，dimens.xml用来存储尺寸值，等等。资源文件的单一，是说所有相关的资源信息要在资源文件里定义并引用到代码或布局文件里，而不是在代码或布局文件里直接定义。很多开发人员，为了图方便，应用界面中出现的字符串经常在代码或布局文件里直接定义的，尺寸值也是，这样造成的结果就是，当某些字符串需要修改时，比如要支持国际化，或一些尺寸值需要修改时，通常是很多地方都要修改。因此，就必须规范好，应用界面中的字符串统一在strings.xml中定义，颜色值统一在colors.xml中定义，尺寸值统一在dimens.xml中定义，代码或布局里需要用到的都去引用资源文件相应的字段。

要保持单一性，必定伴随着重构。需求总会变动，代码总会扩展，扩展了慢慢就会破坏原有的单一性，因此就需要重构，再次保持单一性。不断扩展，不断重构，这样才能不断保持良好的单一性。

### 简洁性

代码最怕的就是臃肿，臃肿的代码可读性差，维护麻烦，扩展更不用说了。没有人会喜欢看臃肿的代码，去维护更痛苦。让代码保持简洁，会让人看得舒服，一目了然，维护和扩展起来也都非常方便。简洁的代码，甚至不需要写注释，只从代码就能让人一眼看懂其做了什么。简洁也并不只表现在代码上，类、包、资源文件等的命名和组织结构等也同样需要保持简洁。  
如何保持简洁？这个问题并没有一个标准的答案，但有一个判断是否简洁的简单标准，那就是：直接阅读代码就能够理解代码的意图，如果意图不够明显，那就说明这段代码还不够简洁。类、包、资源文件等等，也是同样的评判标准。下面是保持简洁有一定作用的一些操作方法。

#### 包的组织

按照组件类型来分包，而不是按业务模块来分包。业务有可能会变，但组件类型是基本不变的。另外，新加入的开发人员，对业务不熟悉，但对组件是很清楚的，理解快，入手也快。

#### 类和接口的命名

组件类的命名添加该组件的后缀，例如：Activity类命名添加Activity后缀，Fragment类命令添加Fragment后缀，适配器添加Adapter后缀，等等。实体类则可添加BO的后缀名称，工具类添加util后缀，接口的实现类添加Impl的后缀。接口的命名也一样，比如，接口层的接口后缀都带上了Api，核心层的接口后缀都带Action。

#### 资源文件的分类

strings.xml文件用来存储应用中的所有字符串，包括页面标题，按钮文字，标签文字，提示文字等等，应该做好分类并统一存放。下面是一些规范的分类方法，如果某个分类的字符串数量太多了，还可以拆分出来放到一个独立的文件，比如页面标题，可以拆分到strings\_title.xml文件里，其他资源文件也可以用类似的方式进行处理：

页面标题，命名格式为：title\_{页面}

按钮文字，命名格式为：btn\_{按钮事件}

标签文字，命名格式为：label\_{标签文字}

选项卡文字，命名格式为：tab\_{选项卡文字}

消息框文字，命名格式为：toast\_{消息}

编辑框的提示文字，命名格式为：hint\_{提示信息}

图片的描述文字，命名格式为：desc\_{图片文字}

对话框的文字，命名格式为：dialog\_{文字}

## 如何优化内存总结

可以从四个方面着手

* 首先是减小对象的内存占用，
* 其次是内存对象的重复利用，
* 然后是避免对象的内存泄露，
* 最后是内存使用策略优化。

## 减小对象的内存占用

避免OOM的第一步就是要尽量减少新分配出来的对象占用内存的大小，尽量使用更加轻量的对象。

### 1）使用更加轻量的数据结构

例如，我们可以考虑使用ArrayMap/SparseArray而不是HashMap等传统数据结构。图8演示了HashMap的简要工作原理，相比起Android专门为移动操作系统编写的ArrayMap容器，在大多数情况下，都显示效率低下，更占内存。通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效，在于他们避免了对key与value的自动装箱（autoboxing），并且避免了装箱后的解箱。

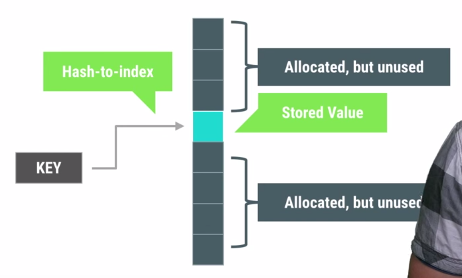


图8  HashMap简要工作原理

关于更多ArrayMap/SparseArray的讨论，请参考《 Android性能优化典范（三）》的前三个段落。

### 2）避免在Android里面使用Enum

Android官方培训课程提到过“Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.”，具体原理请参考《Android性能优化典范（三）》，所以请避免在Android里面使用到枚举。

### 3）减小Bitmap对象的内存占用

Bitmap是一个极容易消耗内存的大胖子，减小创建出来的Bitmap的内存占用可谓是重中之重，通常来说有以下2个措施：

* inSampleSize：缩放比例，在把图片载入内存之前，我们需要先计算出一个合适的缩放比例，避免不必要的大图载入。
* decode format：解码格式，选择ARGB\_8888/RBG\_565/ARGB\_4444/ALPHA\_8，存在很大差异。

### 4）使用更小的图片

在涉及给到资源图片时，我们需要特别留意这张图片是否存在可以压缩的空间，是否可以使用更小的图片。尽量使用更小的图片不仅可以减少内存的使用，还能避免出现大量的InflationException。假设有一张很大的图片被XML文件直接引用，很有可能在初始化视图时会因为内存不足而发生InflationException，这个问题的根本原因其实是发生了OOM。

## 内存对象的重复利用

大多数对象的复用，最终实施的方案都是利用对象池技术，要么是在编写代码时显式地在程序里创建对象池，然后处理好复用的实现逻辑。要么就是利用系统框架既有的某些复用特性，减少对象的重复创建，从而降低内存的分配与回收（如图9所示）。

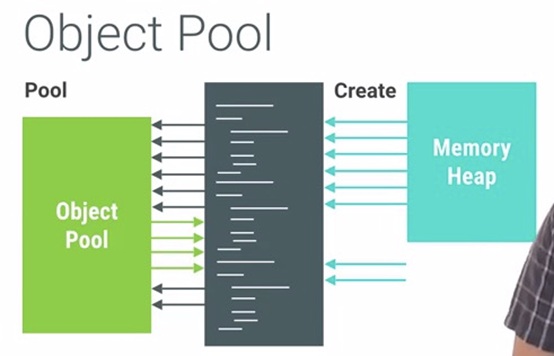


图9  对象池技术

在Android上面最常用的一个缓存算法是LRU(Least Recently Use)，简要操作原理如图10所示。

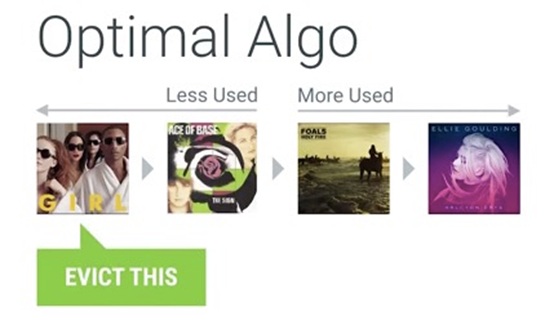


图10  LRU简要操作原理

### 1）复用系统自带的资源

Android系统本身内置了很多的资源，比如字符串、颜色、图片、动画、样式以及简单布局等，这些资源都可以在应用程序中直接引用。这样做不仅能减少应用程序的自身负重，减小APK的大小，还可以在一定程度上减少内存的开销，复用性更好。但是也有必要留意Android系统的版本差异性，对那些不同系统版本上表现存在很大差异、不符合需求的情况，还是需要应用程序自身内置进去。

### 2）注意在ListView/GridView的优化

等出现大量重复子组件的视图里对ConvertView的复用，

如图11所示。

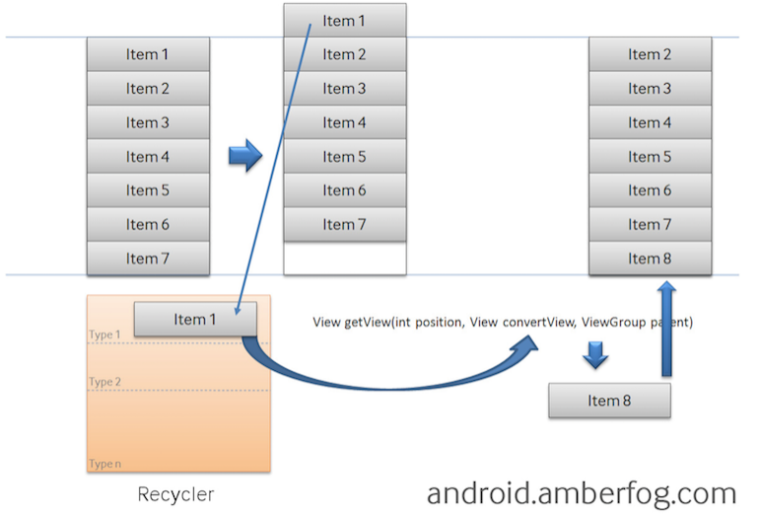


图11

### 3）Bitmap对象的复用

在ListView与GridView等显示大量图片的控件里，需要使用LRU的机制来缓存处理好的Bitmap，如图12所示。

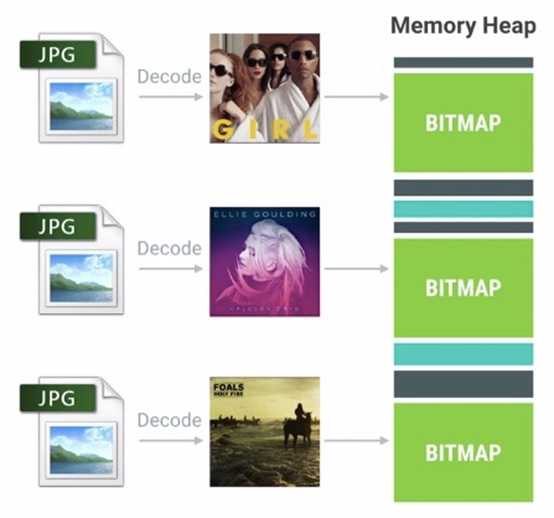


图12

* 利用inBitmap的高级特性提高Android系统在Bitmap分配与释放执行效率(注：3.0以及4.4以后存在一些使用限制上的差异)。使用inBitmap属性可以告知Bitmap解码器去尝试使用已经存在的内存区域，新解码的Bitmap会尝试去使用之前那张Bitmap在Heap中所占据的pixel data内存区域，而不是去问内存重新申请一块区域来存放Bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小，如图13所示。

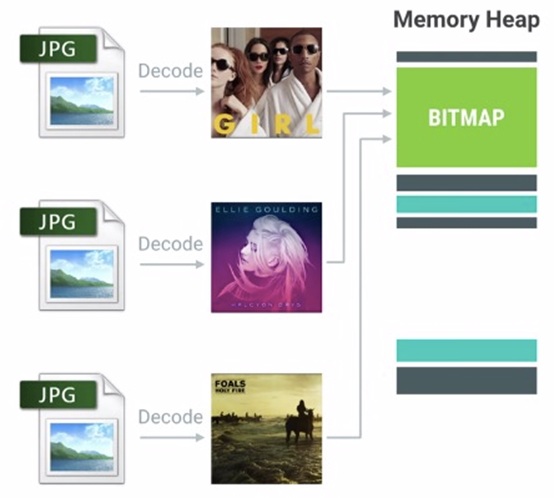


图13  利用inBitmap的高级特性提高Android在Bitmap分配与释放执行效率

使用inBitmap需要注意几个限制条件：

* 在SDK 11 -> 18之间，重用的Bitmap大小必须是一致的。例如给inBitmap赋值的图片大小为100-100，那么新申请的Bitmap必须也为100-100才能够被重用。从SDK 19开始，新申请的Bitmap大小必须小于或者等于已经赋值过的Bitmap大小。
* 新申请的Bitmap与旧的Bitmap必须有相同的解码格式。例如大家都是8888的，如果前面的Bitmap是8888，那么就不能支持4444与565格式的Bitmap了。我们可以创建一个包含多种典型可重用Bitmap的对象池，这样后续的Bitmap创建都能够找到合适的“模板”去进行重用，如图14所示。



图14

另外，在2.x的系统上，尽管Bitmap是分配在Native层，但还是无法避免被计算到OOM的引用计数器里。这里提示一下，不少应用会通过反射vBitmapFactory.Options里面的inNativeAlloc来达到扩大使用内存的目的，但是如果大家都这么做，对系统整体会造成一定的负面影响，建议谨慎采纳。

4）避免在onDraw方法里面执行对象的创建

类似onDraw等频繁调用的方法，一定需要注意避免在这里做创建对象的操作，因为他会迅速增加内存的使用，而且很容易引起频繁的gc，甚至是内存抖动。

5）StringBuilder

在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”。

## 避免对象的内存泄露

内存对象的泄漏，会导致一些不再使用的对象无法及时释放，这样一方面占用了宝贵的内存空间，很容易导致后续需要分配内存的时候，空闲空间不足而出现OOM。显然，这还使得每级Generation的内存区域可用空间变小，GC就会更容易被触发，容易出现内存抖动，从而引起性能问题（如图15所示）。

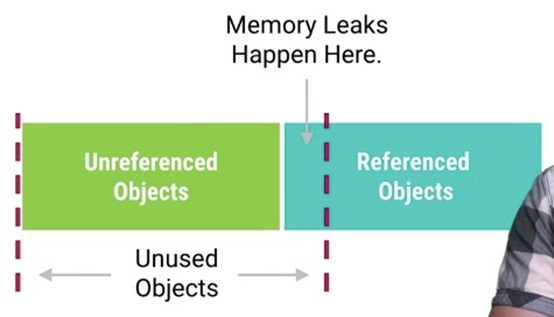


图15

最新的LeakCanary开源控件，可以很好的帮助我们发现内存泄露的情况。另外也可以使用传统的MAT工具查找内存泄露。

### 1）注意Activity的泄漏

通常来说，Activity的泄漏是内存泄漏里面最严重的问题，它占用的内存多，影响面广，我们需要特别注意以下两种情况导致的Activity泄漏：

* 内部类引用导致Activity的泄漏

最典型的场景是Handler导致的Activity泄漏，如果Handler中有延迟的任务或者是等待执行的任务队列过长，都有可能因为Handler继续执行而导致Activity发生泄漏。此时的引用关系链是Looper -> MessageQueue -> Message -> Handler -> Activity。为了解决这个问题，可以在UI退出之前，执行remove Handler消息队列中的消息与runnable对象。或者是使用Static + WeakReference的方式来达到断开Handler与Activity之间存在引用关系的目的。

* Activity Context被传递到其他实例中，这可能导致自身被引用而发生泄漏。

内部类引起的泄漏不仅仅会发生在Activity上，其他任何内部类出现的地方，都需要特别留意！我们可以考虑尽量使用static类型的内部类，同时使用WeakReference的机制来避免因为互相引用而出现的泄露。

### 2）考虑使用Application Context而不是Activity Context

对于大部分非必须使用Activity Context的情况（Dialog的Context就必须是Activity Context），我们都可以考虑使用Application Context而不是Activity的Context，这样可以避免不经意的Activity泄露。

### 3）注意临时Bitmap对象的及时回收

虽然在大多数情况下，我们会对Bitmap增加缓存机制，但是在某些时候，部分Bitmap是需要及时回收的。例如临时创建的某个相对比较大的bitmap对象，在经过变换得到新的bitmap对象之后，应该尽快回收原始的bitmap，这样能够更快释放原始bitmap所占用的空间。

需要特别留意的是Bitmap类里面提供的createBitmap()方法，如图16所示：

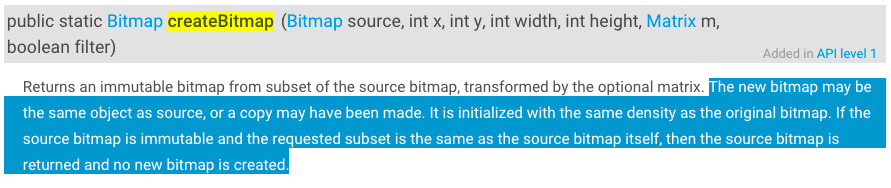


图16  createBitmap()方法

这个函数返回的bitmap有可能和source bitmap是同一个，在回收的时候，需要特别检查source bitmap与return bitmap的引用是否相同，只有在不等的情况下，才能够执行source bitmap的recycle方法。

### 4）注意监听器的注销

在Android程序里面存在很多需要register与unregister的监听器，我们需要确保在合适的时候及时unregister那些监听器。自己手动add的listener，需要记得及时remove这个listener。

### 5）注意缓存容器中的对象泄漏

有时候，我们为了提高对象的复用性把某些对象放到缓存容器中，可是如果这些对象没有及时从容器中清除，也是有可能导致内存泄漏的。例如，针对2.3的系统，如果把drawable添加到缓存容器，因为drawable与View的强应用，很容易导致activity发生泄漏。而从4.0开始，就不存在这个问题。解决这个问题，需要对2.3系统上的缓存drawable做特殊封装，处理引用解绑的问题，避免泄漏的情况。

### 6）注意WebView的泄漏

Android中的WebView存在很大的兼容性问题，不仅仅是Android系统版本的不同对WebView产生很大的差异，另外不同的厂商出货的ROM里面WebView也存在着很大的差异。更严重的是标准的WebView存在内存泄露的问题。所以通常根治这个问题的办法是为WebView开启另外一个进程，通过AIDL与主进程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，从而达到内存的完整释放。

### 7）注意Cursor对象是否及时关闭

在程序中我们经常会进行查询数据库的操作，但时常会存在不小心使用Cursor之后没有及时关闭的情况。这些Cursor的泄露，反复多次出现的话会对内存管理产生很大的负面影响，我们需要谨记对Cursor对象的及时关闭。

## 内存使用策略优化

### 1）谨慎使用large heap

正如前面提到的，Android设备根据硬件与软件的设置差异而存在不同大小的内存空间，他们为应用程序设置了不同大小的Heap限制阈值。你可以通过调用getMemoryClass()来获取应用的可用Heap大小。在一些特殊的情景下，你可以通过在manifest的application标签下添加largeHeap=true的属性来为应用声明一个更大的heap空间。然后，你可以通过getLargeMemoryClass()来获取到这个更大的heap size阈值。然而，声明得到更大Heap阈值的本意是为了一小部分会消耗大量RAM的应用(例如一个大图片的编辑应用)。不要轻易的因为你需要使用更多的内存而去请求一个大的Heap Size。只有当你清楚的知道哪里会使用大量的内存并且知道为什么这些内存必须被保留时才去使用large heap。因此请谨慎使用large heap属性。使用额外的内存空间会影响系统整体的用户体验，并且会使得每次gc的运行时间更长。在任务切换时，系统的性能会大打折扣。另外, large heap并不一定能够获取到更大的heap。在某些有严格限制的机器上，large heap的大小和通常的heap size是一样的。因此即使你申请了large heap，你还是应该通过执行getMemoryClass()来检查实际获取到的heap大小。

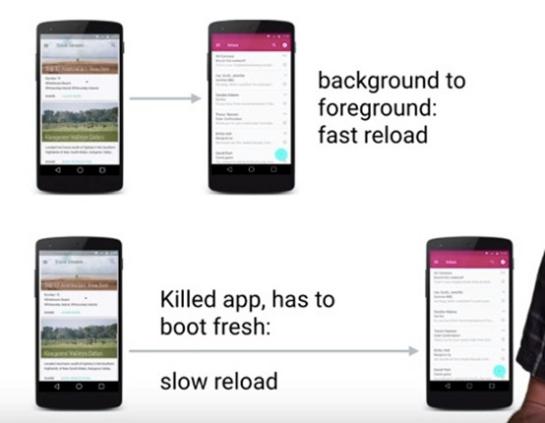
### 2）综合考虑设备内存阈值与其他因素设计合适的缓存大小

例如，在设计ListView或者GridView的Bitmap LRU缓存的时候，需要考虑的点有：

* 应用程序剩下了多少可用的内存空间?
* 有多少图片会被一次呈现到屏幕上？有多少图片需要事先缓存好以便快速滑动时能够立即显示到屏幕？
* 设备的屏幕大小与密度是多少? 一个xhdpi的设备会比hdpi需要一个更大的Cache来hold住同样数量的图片。
* 不同的页面针对Bitmap的设计的尺寸与配置是什么，大概会花费多少内存？
* 页面图片被访问的频率？是否存在其中的一部分比其他的图片具有更高的访问频繁？如果是，也许你想要保存那些最常访问的到内存中，或者为不同组别的位图(按访问频率分组)设置多个LruCache容器。

### 3）onLowMemory()与onTrimMemory()

Android用户可以随意在不同的应用之间进行快速切换。为了让background的应用能够迅速的切换到forground，每一个background的应用都会占用一定的内存。Android系统会根据当前的系统的内存使用情况，决定回收部分background的应用内存。如果background的应用从暂停状态直接被恢复到forground，能够获得较快的恢复体验，如果background应用是从Kill的状态进行恢复，相比之下就显得稍微有点慢，如图17所示。

 图17  从Kill状态进行恢复体验更慢

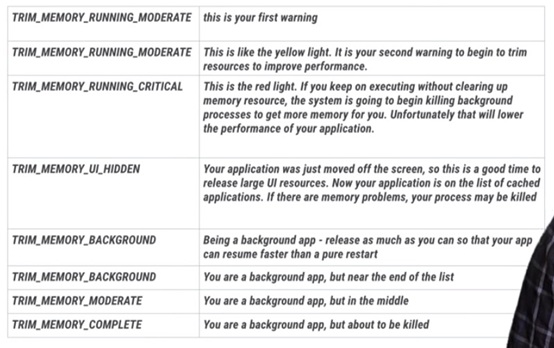
* onLowMemory()：Android系统提供了一些回调来通知当前应用的内存使用情况，通常来说，当所有的background应用都被kill掉的时候，forground应用会收到onLowMemory()的回调。在这种情况下，需要尽快释放当前应用的非必须的内存资源，从而确保系统能够继续稳定运行。
* onTrimMemory(int)：Android系统从4.0开始还提供了onTrimMemory()的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调，同时在这个回调里面会传递以下的参数，代表不同的内存使用情况，收到onTrimMemory()回调的时候，需要根据传递的参数类型进行判断，合理的选择释放自身的一些内存占用，一方面可以提高系统的整体运行流畅度，另外也可以避免自己被系统判断为优先需要杀掉的应用。
* TRIM\_MEMORY\_UI\_HIDDEN：你的应用程序的所有UI界面被隐藏了，即用户点击了Home键或者Back键退出应用，导致应用的UI界面完全不可见。这个时候应该释放一些不可见的时候非必须的资源

当程序正在前台运行的时候，可能会接收到从onTrimMemory()中返回的下面的值之一：

* TRIM\_MEMORY\_RUNNING\_MODERATE：你的应用正在运行并且不会被列为可杀死的。但是设备此时正运行于低内存状态下，系统开始触发杀死LRU Cache中的Process的机制。
* TRIM\_MEMORY\_RUNNING\_LOW：你的应用正在运行且没有被列为可杀死的。但是设备正运行于更低内存的状态下，你应该释放不用的资源用来提升系统性能。
* TRIM\_MEMORY\_RUNNING\_CRITICAL：你的应用仍在运行，但是系统已经把LRU Cache中的大多数进程都已经杀死，因此你应该立即释放所有非必须的资源。如果系统不能回收到足够的RAM数量，系统将会清除所有的LRU缓存中的进程，并且开始杀死那些之前被认为不应该杀死的进程，例如那个包含了一个运行态Service的进程。

当应用进程退到后台正在被Cached的时候，可能会接收到从onTrimMemory()中返回的下面的值之一：

* TRIM\_MEMORY\_BACKGROUND: 系统正运行于低内存状态并且你的进程正处于LRU缓存名单中最不容易杀掉的位置。尽管你的应用进程并不是处于被杀掉的高危险状态，系统可能已经开始杀掉LRU缓存中的其他进程了。你应该释放那些容易恢复的资源，以便于你的进程可以保留下来，这样当用户回退到你的应用的时候才能够迅速恢复。
* TRIM\_MEMORY\_MODERATE: 系统正运行于低内存状态并且你的进程已经已经接近LRU名单的中部位置。如果系统开始变得更加内存紧张，你的进程是有可能被杀死的。
* TRIM\_MEMORY\_COMPLETE: 系统正运行于低内存的状态并且你的进程正处于LRU名单中最容易被杀掉的位置。你应该释放任何不影响你的应用恢复状态的资源。



因为onTrimMemory()的回调是在API 14才被加进来的，对于老的版本，你可以使用onLowMemory)回调来进行兼容。onLowMemory相当与TRIM\_MEMORY\_COMPLETE。

请注意：当系统开始清除LRU缓存中的进程时，虽然它首先按照LRU的顺序来执行操作，但是它同样会考虑进程的内存使用量以及其他因素。占用越少的进程越容易被留下来。

### 4）资源文件需要选择合适的文件夹进行存放

我们知道hdpi/xhdpi/xxhdpi等等不同dpi的文件夹下的图片在不同的设备上会经过scale的处理。例如我们只在hdpi的目录下放置了一张100100的图片，那么根据换算关系，xxhdpi的手机去引用那张图片就会被拉伸到200200。需要注意到在这种情况下，内存占用是会显著提高的。对于不希望被拉伸的图片，需要放到assets或者nodpi的目录下。

### 5）Try catch某些大内存分配的操作

在某些情况下，我们需要事先评估那些可能发生OOM的代码，对于这些可能发生OOM的代码，加入catch机制，可以考虑在catch里面尝试一次降级的内存分配操作。例如decode bitmap的时候，catch到OOM，可以尝试把采样比例再增加一倍之后，再次尝试decode。

### 6）谨慎使用static对象

因为static的生命周期过长，和应用的进程保持一致，使用不当很可能导致对象泄漏，在Android中应该谨慎使用static对象（如图19所示）。



图19

### 7）特别留意单例对象中不合理的持有

虽然单例模式简单实用，提供了很多便利性，但是因为单例的生命周期和应用保持一致，使用不合理很容易出现持有对象的泄漏。

### 8）珍惜Services资源

如果你的应用需要在后台使用service，除非它被触发并执行一个任务，否则其他时候Service都应该是停止状态。另外需要注意当这个service完成任务之后因为停止service失败而引起的内存泄漏。 当你启动一个Service，系统会倾向为了保留这个Service而一直保留Service所在的进程。这使得进程的运行代价很高，因为系统没有办法把Service所占用的RAM空间腾出来让给其他组件，另外Service还不能被Paged out。这减少了系统能够存放到LRU缓存当中的进程数量，它会影响应用之间的切换效率，甚至会导致系统内存使用不稳定，从而无法继续保持住所有目前正在运行的service。 建议使用IntentService，它会在处理完交代给它的任务之后尽快结束自己。更多信息，请阅读 Running in a Background Service。

### 9）优化布局层次，减少内存消耗

越扁平化的视图布局，占用的内存就越少，效率越高。我们需要尽量保证布局足够扁平化，当使用系统提供的View无法实现足够扁平的时候考虑使用自定义View来达到目的。

### 10）谨慎使用“抽象”编程

很多时候，开发者会使用抽象类作为”好的编程实践”，因为抽象能够提升代码的灵活性与可维护性。然而，抽象会导致一个显著的额外内存开销：他们需要同等量的代码用于可执行，那些代码会被mapping到内存中，因此如果你的抽象没有显著的提升效率，应该尽量避免他们。

### 11）使用nano protobufs序列化数据

Protocol buffers是由Google为序列化结构数据而设计的，一种语言无关，平台无关，具有良好的扩展性。类似XML，却比XML更加轻量，快速，简单。如果你需要为你的数据实现序列化与协议化，建议使用nano protobufs。关于更多细节，请参考 protobuf readme的”Nano version”章节。

### 12）谨慎使用依赖注入框架

使用类似Guice或者RoboGuice等框架注入代码，在某种程度上可以简化你的代码。图20是使用RoboGuice前后的对比图：



图20  使用RoboGuice前后对比图

使用RoboGuice之后，代码是简化了不少。然而，那些注入框架会通过扫描你的代码执行许多初始化的操作，这会导致你的代码需要大量的内存空间来mapping代码，而且mapped pages会长时间的被保留在内存中。除非真的很有必要，建议谨慎使用这种技术。

### 13）谨慎使用多进程

使用多进程可以把应用中的部分组件运行在单独的进程当中，这样可以扩大应用的内存占用范围，但是这个技术必须谨慎使用，绝大多数应用都不应该贸然使用多进程，一方面是因为使用多进程会使得代码逻辑更加复杂，另外如果使用不当，它可能反而会导致显著增加内存。当你的应用需要运行一个常驻后台的任务，而且这个任务并不轻量，可以考虑使用这个技术。

一个典型的例子是创建一个可以长时间后台播放的Music Player。如果整个应用都运行在一个进程中，当后台播放的时候，前台的那些UI资源也没有办法得到释放。类似这样的应用可以切分成2个进程：一个用来操作UI，另外一个给后台的Service。

### 14）使用ProGuard来剔除不需要的代码

ProGuard能够通过移除不需要的代码，重命名类，域与方法等等对代码进行压缩，优化与混淆。使用ProGuard可以使得你的代码更加紧凑，这样能够减少mapping代码所需要的内存空间。

### 15）谨慎使用第三方libraries

很多开源的library代码都不是为移动网络环境而编写的，如果运用在移动设备上，并不一定适合。即使是针对Android而设计的library，也需要特别谨慎，特别是在你不知道引入的library具体做了什么事情的时候。例如，其中一个library使用的是nano protobufs, 而另外一个使用的是micro protobufs。这样一来，在你的应用里面就有2种protobuf的实现方式。这样类似的冲突还可能发生在输出日志，加载图片，缓存等等模块里面。另外不要为了1个或者2个功能而导入整个library，如果没有一个合适的库与你的需求相吻合，你应该考虑自己去实现，而不是导入一个大而全的解决方案。

### 16）考虑不同的实现方式来优化内存占用

在某些情况下，设计的某个方案能够快速实现需求，但是这个方案却可能在内存占用上表现的效率不够好。例如：

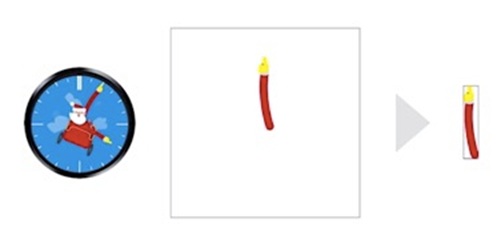


图21

对于上面这样一个时钟表盘的实现，最简单的就是使用很多张包含指针的表盘图片，使用帧动画实现指针的旋转。但是如果把指针扣出来，单独进行旋转绘制，显然比载入N多张图片占用的内存要少很多。当然这样做，代码复杂度上会有所增加，这里就需要在优化内存占用与实现简易度之间进行权衡了。