Android内存优化

转载自：**Android性能优化之内存优化**

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

内存优化可以说是性能优化中最重要的优化点之一，可以说，如果你没有掌握系统的内存优化方案，就不能说你对Android的性能优化有过多的研究与探索。本篇，笔者将带领大家一起来系统地学习Android中的内存优化。

可能有不少读者都知道，在内存管理上，JVM拥有垃圾内存回收的机制，自身会在虚拟机层面自动分配和释放内存，因此不需要像使用C/C++一样在代码中分配和释放某一块内存。Android系统的内存管理类似于JVM，通过new关键字来为对象分配内存，内存的释放由GC来回收。并且Android系统在内存管理上有一个Generational Heap Memory模型，当内存达到某一个阈值时，系统会根据不同的规则自动释放可以释放的内存。即便有了内存管理机制，但是，如果不合理地使用内存，也会造成一系列的性能问题，比如内存泄漏、内存抖动、短时间内分配大量的内存对象等等。下面，我就先谈谈Android的内存管理机制。

一、Android内存管理机制

我们都知道，应用程序的内存分配和垃圾回收都是由Android虚拟机完成的，在Android 5.0以下，使用的是Dalvik虚拟机，5.0及以上，则使用的是ART虚拟机。

1.1、Java对象生命周期

Java代码编译后生成的字节码.class文件从从文件系统中加载到虚拟机之后，便有了JVM上的Java对象，Java对象在JVM上运行有7个阶段，如下：

- Created
- InUse
- Invisible
- Unreachable
- Collected
- Finalized
- Deallocated

1、Created（创建）

Java对象的创建分为如下几步：

- 1、为对象分配存储空间。
- 2、构造对象。
- 3、从超类到子类对static成员进行初始化，类的static成员的初始化在ClassLoader加载该类时进行。
- 4、超类成员变量按顺序初始化，递归调用超类的构造方法。
- 5、子类成员变量按顺序初始化，一旦对象被创建，子类构造方法就调用该对象并为某些变量赋值。

2、InUse（应用）

此时对象至少被一个强引用持有。

3、Invisible（不可见）

当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该对象仍然是存在的。简单的例子就是程序的执行已经超出了该对象的作用域了。但是，该对象仍可能被虚拟机下的某些已装载的静态变量线程或JNI等强引用持有，这些特殊的强引用称为“GC Root”。被这些GC Root强引用的对象会导致该对象的内存泄漏，因而无法被GC回收。

4、Unreachable（不可达）

该对象不再被任何强引用持有。

5、Collected（收集）

当GC已经对该对象的内存空间重新分配做好准备时，对象进入收集阶段，如果该对象重写了finalize()方法，则执行它。

6、Finalized（终结）

等待垃圾回收器回收该对象空间。

7、Deallocated（对象空间重新分配）

GC对该对象所占用的内存空间进行回收或者再分配，则该对象彻底消失。

注意：

- 1、不需要使用该对象时，及时置空。
- 2、访问本地变量优于访问类中的变量。

1.2 内存分配

在Android系统中，堆实际上就是一块匿名共享内存。Android虚拟机仅仅只是把它封装成一个mSpace，由底层C库来管理，并且仍然使用libc提供的函数malloc和free来分配和释放内存。
大多数静态数据会被映射到一个共享的进程中。常见的静态数据包括Dalvik Code、app resources、so文件等等。
在大多数情况下，Android通过显示分配共享内存区域（如ashmem或者gralloc）来实现动态RAM区域能够在不同进程之间共享的机制。例如，Window Surface在App和Screen Compositor之间使用共享的内存，Cursor Buffers在Content Provider和Clients之间共享内存。
上面说过，对于Android Runtime有两种虚拟机，Dalvik和ART，它们分配的内存区域块是不同的：

Dalvik

- Linear Alloc
- Zygote Space
- Alloc Space

ART

- Non Moving Space
- Zygote Space
- Alloc Space
- Image Space
- Large Obj Space

不管是Dlavik还是ART，运行时堆都分为LinearAlloc（类似于ART的Non Moving Space）、Zygote Space和Alloc Space。Dalvik中的Linear Alloc是一个线性内存空间，是一个只读区域，主要用来存储虚拟机中的类，因为类加载后只需要读的属性，并且不会改变它。把这些只读属性以及在整个进程的生命周期都不能结束的永久数据放到线性分配器中管理，能很好地减少堆混乱和GC扫描，提升内存管理的性能。

Zygote Space在Zygote进程和应用程序进程之间共享，Allocation Space则是每个进程独占。Android系统的第一个虚拟机由Zygote进程创建并且只有一个Zygote Space。但是当Zygote进程在fork第一个应用程序进程之前，会将已经使用的那部分堆内存划分为一部分，还没有使用的堆内存划分为另一部分，也就是Allocation Space。但无论是应用程序进程，还是Zygote进程，当他们需要分配对象时，都是在各自的Allocation Space堆上进行。

当在ART运行时，还有另外两个区块，即ImageSpace和Large Object Space。

- Image Space：存放一些预加载类，类似于Dalvik中的Linear Alloc。与Zygote Space一样，在Zygote进程和应用程序进程之间共享。
- Large Object Space：离散地址的集合，分配一些大对象，用于提高GC的管理效率和整体性能。

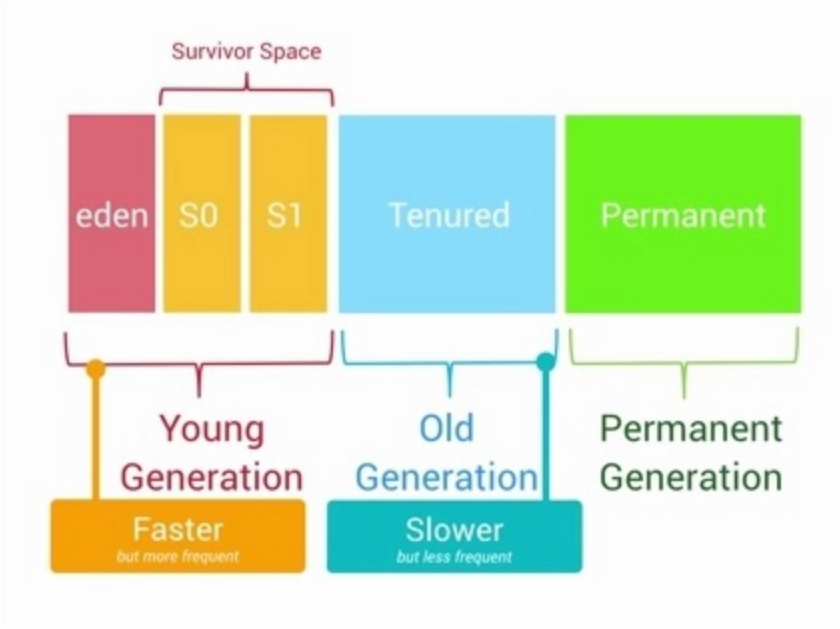
注意：Image Space的对象只创建一次，而Zygote Space的对象需要在系统每次启动时，根据运行情况都重新创建一遍。

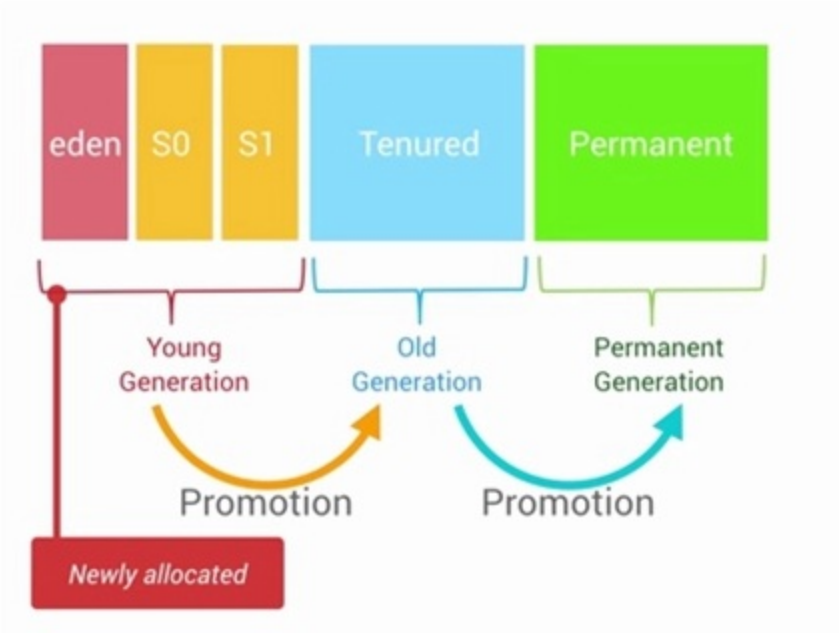
1.3、内存回收机制

在Android的高级系统版本中，针对Heap空间有一个Generational Heap Memory的模型，其中将整个内存分为三个区域：

- Young Generation（年轻代）
- Old Generation（年老代）
- Permanent Generation（持久代）

模型示意图如下所示：





1、Young Generation

由一个Eden区和两个Survivor区组成，程序中生成的大部分新的对象都在Eden区中，当Eden区满时，还存活的对象将被复制到其中一个Survivor区，当次Survivor区满时，此区存活的对象又被复制到另一个Survivor区，当这个Survivor区也满时，会将其中存活的对象复制到年老代。

2、Old Generation

一般情况下，年老代中的对象生命周期都比较长。

3、Permanent Generation

用于存放静态的类和方法，持久代对垃圾回收没有显著影响。

总结：内存对象的处理过程如下：

- 1、对象创建后在Eden区。
- 2、执行GC后，如果对象仍然存活，则复制到S0区。

- 3、当S0区满时，该区域存活对象将复制到S1区，然后S0清空，接下来S0和S1角色互换。
- 4、当第3步达到一定次数（系统版本不同会有差异）后，存活对象将被复制到Old Generation。
- 5、当这个对象在Old Generation区域停留的时间达到一定程度时，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。

系统在Young Generation、Old Generation上采用不同的回收机制。每一个Generation的内存区域都有固定的大小。随着新的对象陆续被分配到此区域，当对象总的大小临近这一级别内存区域的阈值时，会触发GC操作，以便腾出空间来存放其他新的对象。

执行GC占用的时间与Generation和Generation中的对象数量有关：

- Young Generation < Old Generation < Permanent Generation
- Gener中的对象数量与执行时间成正比。

4、Young Generation GC

由于其对象存活时间短，因此基于Copying算法（扫描出存活的对象，并复制到一块新的完全未使用的控件中）来回收。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在Young Generation区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。

5、Old Generation GC

由于其对象存活时间较长，比较稳定，因此采用Mark（标记）算法（扫描出存活的对象，然后再回收未被标记的对象，回收后对空出的空间要么合并，要么标记出来便于下次分配，以减少内存碎片带来的效率损耗）来回收。

1.4、GC类型

在Android系统中，GC有三种类型：

- kGcCauseForAlloc：分配内存不够引起的GC，会Stop World。由于是并发GC，其它线程都会停止，直到GC完成。
- kGcCauseBackground：内存达到一定阈值触发的GC，由于是一个后台GC，所以不会引起Stop World。
- kGcCauseExplicit：显示调用时进行的GC，当ART打开这个选项时，使用System.gc时会进行GC。

接下来，我们来学会如何分析Android虚拟机中的GC日志，日志如下：

```
D/dalvikvm(7030): GC_CONCURRENT freed 1049K, 60% free 2341K/9351K, ext
ernal 3502K/6261K, paused 3ms 3ms
```

GC_CONCURRENT是当前GC时的类型，GC日志中有以下几种类型：

- GC_CONCURRENT：当应用程序中的Heap内存占用上升时（分配对象大小超过384k），避免Heap内存满了而触发的GC。如果发现有大量的GC_CONCURRENT出现，说明应用中可能一直有大于384k的对象被分配，而这一般都是一些临时对象被反复创建，可能是**对象复用不够所导致的**。
- GC_FOR_MALLOC：这是由于Concurrent GC没有及时执行完，而应用又需要分配更多的内存，这时不得不停下来进行Malloc GC。
- GC_EXTERNAL_ALLOC：这是为external分配的内存执行的GC。
- GC_HPROF_DUMP_HEAP：创建一个HPROF profile的时候执行。

- GC_EXPLICIT：显示调用了System.GC()。（尽量避免）

再回到上面打印的日志:

- freed 1049k 表明在这次GC中回收了多少内存。
- 60% free 2341k/6261K 表明回收后60%的Heap可用，存活的对象大小为2341kb，heap大小是9351kb。
- external 3502/6261K 是Native Memory的数据。存放Bitmap Pixel Data（位图数据）或者堆以外内存（NIO Direct Buffer）之类的。第一个值说明在Native Memory中已分配3502kb内存，第二个值是一个浮动的GC阈值，当分配内存达到这个值时，会触发一次GC。
- paused 3ms 3ms 表明GC的暂停时间，如果是Concurrent GC，会看到两个时间，一个开始，一个结束，且时间很短，如如果是其他类型的GC，很可能只会看到一个时间，且这个时间是相对比较长的。并且，越大的Heap Size在GC时导致暂停的时间越长。

注意：在ART模式下，多了一个Large Object Space，这部分内存并不是分配在堆上，但还是属于应用程序的内存空间。

在Dalvik虚拟机下，GC的操作都是并发的，也就意味着每次触发GC都会导致其它线程暂停工作（包括UI线程）。而在ART模式下，GC时不像Dalvik仅有一种回收算法，ART在不同的情况下会选择不同的回收算法，比如Alloc内存不够时会采用非并发GC，但在Alloc后，发现内存达到一定阈值时又会触发并发GC。所以在ART模式下，并不是所有的GC都是非并发的。

总体来看，在GC方面，与Dalvik相比，ART更为高效，不仅仅是GC的效率，大大地缩短了Pause时间，而且在内存分配上对大内存分配单独的区域，还能有算法在后台做内存整理，减少内存碎片。因此，在ART虚拟机下，可以避免较多的类似GC导致的卡顿问题。

二、优化内存的意义

- 减少OOM，提高应用稳定性。
- 减少卡顿，提高应用流畅度。
- 减少内存占用，提高应用后台运行时的存活率。
- 减少异常发生和代码逻辑隐患。

注意：出现OOM是因为内存溢出导致，这种情况不一定会发生在相同的代码，也不一定是出现OOM的代码使用内存有问题，而是刚好执行到这段代码。

三、避免内存泄漏

3.1 内存泄漏定义

Android系统虚拟机的垃圾回收是通过虚拟机GC机制来实现的。GC会选择一些还存活的对象作为内存遍历的根节点GC Roots，通过对GC Roots的可达性来判断是否需要回收。内存泄漏就是在当前应用周期内不再使用的对象被GC Roots引用，导致不能回收，使实际可使用内存变小。

3.2 使用MAT来查找内存泄漏

MAT工具可以帮助开发者定位导致内存泄漏的对象，以及发现大的内存对象，然后解决内存泄漏并通过优化内存对象，达到减少内存消耗的目的。

3.2.1 使用步骤

1、在https://eclipse.org/mat/downloads.php下载MAT客户端。
2、从Android Studio进入Profile的Memory视图，选择需要分析的应用进程，对应用进行怀疑有内存问题的操作，结束操作后，主动GC几次，最后export dump文件。
3、因为Android Studio保存的是Android Dalvik/ART格式的.hprof文件，所以需要转换成J2SE HPROF格式才能被MAT识别和分析。Android SDK自带了一个转换工具在SDK的platform-tools下，其中转换语句为：

```
./hprof-conv file.hprof converted.hprof
```

4、通过MAT打开转换后的HPROF文件。

3.2.2 MAT视图

在MAT窗口上，OverView是一个总体概览，显示总体的内存消耗情况和疑似问题。MAT提供了多种分析维度，其中Histogram、Dominator Tree、Top Consumers和Leak Suspects的分析维度不同。下面分别介绍下：
1、Histogram
列出内存中的所有实例类型对象和其个数以及大小，并在顶部的regex区域支持正则表达式查找。
2、Dominator Tree
列出最大的对象及其依赖存活的Object。相比Histogram，能更方便地看出引用关系。
3、Top Consumers
通过图像列出最大的Object。
4、Leak Suspects
通过MAT自动分析内存泄漏的原因和泄漏的一份总体报告。
分析内存最常用的是Histogram和Dominator Tree两个视图，视图中一共有四列：

- Class Name：类名。
- Objects：对象实例个数。
- Shallow Heap：对象自身占用的内存大小，不包括它引用的对象。非数组的常规对象的Shallow Heap Size由其成员变量的数量和类型决定，数组的Shallow Heap Size由数组元素的类型（对象类型、基本类型）和数组长度决定。真正的内存都在堆上，看起来是一堆原生的byte[]、char[]、int[]，对象本身的内存都很小。因此Shallow Heap对分析内存泄漏意义不是很大。
- Retained Heap：是当前对象大小与当前对象可直接或间接引用到的对象的大小总和，包括被递归释放的。即：Retained Size就是当前对象被GC后，从Heap上总共能释放掉的内存大小。

3.2.3 查找内存泄漏具体位置

常规方式：

- 按照包名类型分类进行实例筛选或直接使用顶部Regex选取特定实例。
- 右击选中被怀疑的实例对象，选择Merge Shortest Paths to GC Root->exclude all phantom/weak/soft etc references。(显示GC Roots最短路径的强引用)
- 分析引用链或通过代码逻辑找出原因。

还有一种更快速的方法就是对比泄漏前后的HPROF数据：

- 在两个HPROF文件中，把Histogram或者Dominator Tree增加到Compare Basket。

- 在Compare Basket中单击！，生成对比结果视图。这样就可以对比相同的对象在不同阶段的对象实例个数和内存占用大小，如明显只需要一个实例的对象，或者不应该增加的对象实例个数却增加了，说明发生了内存泄漏，就需要去代码中定位具体的原因并解决。

注意：如果目标不太明确，可以直接定位当RetainedHeap最大的Object，通过Select incoming references查看引用链，定位到可疑的对象，然后通过Path to GC Roots分析引用链。

此外，我们知道，当Hash集合中过多的对象返回相同的Hash值时，会严重影响性能，这时可以用Map Collision Ratio查找导致Hash集合的碰撞率较高的罪魁祸首。

在本人平时的项目开发中，一般会使用如下两种方式来快速对指定页面进行内存泄漏的检测：

- 1、shell命令+LeakCanary+MAT：运行程序，所有功能跑一遍，确保没有改出问题，完全退出程序，手动触发GC，然后使用adb shell dumpsys meminfo packagename -d命令查看退出界面后Objects下的Views和Activities数目是否为0，如果不是则通过LeakCanary检查可能存在内存泄露的地方，最后通过MAT分析，如此反复，改善满意为止。
- 2、Profile MEMORY：运行程序，对每一个页面进行内存分析检查。首先，反复打开关闭页面5次，然后收到GC（点击Profile MEMORY左上角的垃圾桶图标），如果此时total内存还没有恢复到之前的数值，则可能发生了内存泄露。此时，再点击Profile MEMORY左上角的垃圾桶图标旁的heap dump按钮查看当前的内存堆栈情况，选择按包名查找，找到当前测试的Activity，如果引用了多个实例，则表明发生了内存泄露。

3.3 常见内存泄漏场景

1、资源性对象未关闭

对于资源性对象不再使用时，应该立即调用它的close()函数，将其关闭，然后在置为null。

2、注册对象未注销

3、类的静态变量持有大数据对象

4、非静态内部类的静态实例

该实例的生命周期和应用一样长，这就导致该静态实例一直持有该Activity的引用，Activity的内存资源不能正常回收。

解决方案：

将内部类设为静态内部类或将内部类抽取来作为一个单例，如果需要使用Context，尽量使用Application Context，如果需要使用Activity Context，就记得用完后置空让GC可以回收，否则还是会内存泄漏。

5、Handler临时性内存泄漏

Message发出之后存储在MessageQueue中，在Message中存在一个target，它是Handler的一个引用，Message在Queue中存在的时间过长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。并且消息队列是在一个Looper线程中不断地轮询处理消息，当这个Activity退出时，消息队列中还有未处理的消息或者正在处理的消息，并且消息队列中的Message持有Handler实例的引用，Handler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏。

解决方案：

- 1、使用一个静态Handler内部类，然后对Handler持有的对象（一般是Activity）使用弱引用，这样在回收时，也可以回收Handler持有的对象。
- 2、在Activity的Destroy或者Stop时，应该移除消息队列中的消息，避免Looper线程的消息队列中有待处理的消息需要处理。

注意：AsyncTask内部也是Handler机制，同样存在内存泄漏风险，当其一般是临时性的。

6、容器中的对象没清理造成的内存泄漏

7、WebView

WebView都存在内存泄漏的问题，在应用中只要使用一次WebView，内存就不会被释放掉。
解决方案：
为WebView开启一个独立的进程，使用AIDL与应用的主进程进行通信，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，达到正常释放内存的目的。

3.4 内存泄漏监控

一般使用LeakCanary进行内存泄漏的监控即可，具体使用和原理分析请参见我之前的文章Android主流三方库源码分析（六、深入理解Leakcanary源码）。
除了基本使用外，我们还可以自定义处理结果，首先，继承DisplayLeakService实现一个自定义的监控处理Service，代码如下：

```
public class LeakCnaryService extends DisplayLeakServcie {
    private final String TAG = “LeakCanaryService”;
    @Override
    protected void afterDefaultHandling(HeapDump heapDump,  AnalysisResult result,  String leakInfo) {
        ...
    }
}
```

重写afterDefaultHanding方法，在其中处理需要的数据，三个参数的定义如下：

- heapDump：堆内存文件，可以拿到完成的hprof文件，以使用MAT分析。
- result：监控到的内存状态，如是否泄漏等。
- leakInfo：leak trace详细信息，除了内存泄漏对象，还有设备信息。

然后在install时，使用自定义的LeakCanaryService即可，代码如下：

```
public class BaseApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        mRefWatcher = LeakCanary.install(this, LeakCanaryService.class, AndroidExcludedRefs.createAppDefaults().build());
    }
```

```
    ...
}
```

经过这样的处理，就可以在LeakCanaryService中实现自己的处理方式，如丰富的提示信息，把数据保存在本地、上传到服务器进行分析。
注意：LeakCanaryService需要在AndroidManifest中注册。

四、优化内存空间

4.1 对象引用

从Java 1.2版本开始引入了三种对象引用方式：SoftReference、WeakReference和PhantomReference三个引用类，引用类的主要功能就是能够引用但仍可以被垃圾回收器回收的对象。在引入引用类之前，只能使用Strong Reference，如果没有指定对象引用类型，默认是强引用。

1、强引用

如果一个对象具有强引用，GC就绝对不会回收它。当内存空间不足时，JVM会抛出OOM错误。

2、软引用

如果一个对象只具有软引用，则内存空间足够，GC时就不会回收它；如果内存不足，就会回收这些对象的内存。可用来实现内存敏感的高速缓存。
软引用可以和一个ReferenceQueue（引用队列）联合使用，如果软引用引用的对象被垃圾回收器回收，JVM会把这个软引用加入与之关联的引用队列中。

3、弱引用

在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间是否足够，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。
注意：可能需要运行多次GC，才能找到并释放弱引用对象。

4、虚引用

只能用于跟踪即将对被引用对象进行的收集。虚拟机必须与ReferenceQueue类联合使用。因为它能够充当通知机制。

4.2 减少不必要的内存开销

1、AutoBoxing

自动装箱的核心就是把基础数据类型转换成对应的复杂类型。在自动装箱转化时，都会产生一个新的对象，这样就会产生更多的内存和性能开销。如int只占4字节，而Integer对象有16字节，特别是HashMap这类容器，进行增、删、改、查操作时，都会产生大量的自动装箱操作。
检测方式：使用TraceView查看耗时，如果发现调用了大量的integer.value，就说明发生了AutoBoxing。

2、内存复用

- 资源复用：通用的字符串、颜色定义、简单页面布局的复用。
- 视图复用：可以使用ViewHolder实现ConvertView复用。
- 对象池：显示创建对象池，实现复用逻辑，对相同的类型数据使用同一块内存空间。
- Bitmap对象的复用：使用inBitmap属性可以告知Bitmap解码器尝试使用已经存在的内存区域，新解码的bitmap会尝试使用之前那张bitmap在heap中占据的pixel data内存区域。

4.3 使用最优的数据类型的

1、HashMap与ArrayMap

HashMap是一个散列链表，向HashMap中put元素时，先根据key的HashCode重新计算hash值，根据hash值得到这个元素在数组中的位置，如果数组该位置上已经存放有其它元素了，那么这个位置上的元素将以链表的形式存放，新加入的放在链头，最后加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。也就是说，向HashMap插入一个对象前，会给一个通向Hash阵列的索引，在索引的位置中，保存了这个Key对象的值。这意味着需要考虑的一个最大问题是冲突，当多个对象散列于阵列相同位置时，就会有散列冲突的问题。因此，HashMap会配置一个大的数组来减少潜在的冲突，并且会有其他逻辑防止链接算法和一些冲突的发生。
ArrayMap提供了和HashMap一样的功能，但避免了过多的内存开销，方法是使用两个小数组，而不是一个大数组。并且ArrayMap在内存上是连续不间断的。
总体来说，在ArrayMap中执行插入或者删除操作时，从性能角度上看，比HashMap还要更差一些，但如果只涉及很小的对象数，比如1000以下，就不需要担心这个问题了。因为此时ArrayMap不会分配过大的数组。

2、枚举类型

使用枚举类型的dex size是普通常量定义的dex size的13倍以上，同时，运行时的内存分配，一个enum值的声明会消耗至少20bytes。
枚举最大的优点是类型安全，但在Android平台上，枚举的内存开销是直接定义常量的三倍以上。所以Android提供了注解的方式检查类型安全。目前提供了int型和String型两种注解方式：IntDef和StringDef，用来提供编译期的类型检查。
注意：使用IntDef和StringDef需要在Gradle配置中引入相应的依赖包：

```
compile 'com.android.support:support-annotations:22.0.0'
```

3、LruCache

最近最少使用缓存，使用强引用保存需要缓存的对象，它内部维护了一个由LinkedHashMap组成的双向列表，不支持线程安全，LruCache对它进行了封装，添加了线程安全操作。当其中的一个值被访问时，它被放到队列的尾部，当缓存将满时，队列头部的值（最近最少被访问的）被丢弃，之后可以被GC回收。
除了普通的get/set方法之外，还有sizeOf方法，它用来返回每个缓存对象的大小。此外，还有entryRemoved方法，当一个缓存对象被丢弃时调用的方法，当第一个参数为true：表明环处对象是为了腾出空间而被清理时。否则，表明缓存对象的entry被remove移除或者被put覆盖时。
注意：分配LruCache大小时应考虑应用剩余内存有多大。

4、图片内存优化

在Android默认情况下，当图片文件解码成位图时，会被处理成32bit/像素。红色、绿色、蓝色和透明通道各8bit，即使是没有透明通道的图片，如JEPG隔世是没有透明通道的，但然后会处理成32bit位图，这样分配的32bit中的8bit透明通道数据是没有任何用处的，这完全没有必要，并且在这些图片被屏幕渲染之前，它们首先要被作为纹理传送到GPU，这意味着每一张图片会同时占用CPU内存和GPU内存。
减少内存开销常用方式如下：

- 1、设置位图的规格：当显示小图片或对图片质量要求不高时可以考虑使用RGB_565，用户头像或圆角图片一般可以尝试ARGB_4444。通过设置inPreferredConfig参数来实现不同的位图规格，代码如下所示：

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inPreferredConfig = Bitmap.Config.RGB_565;
BitmapFactory.decodeStream(is, null, options);
```

- 2、inSampleSize：位图功能对象中的inSampleSize属性实现了位图的缩放功能，代码如下所示：

```
BitampFactory.Options options = new BitmapFactory.Options();
// 设置为4就是1/4大小的图片。因此，图片大小总会比原始图片小一倍以上。
options.inSampleSize = 4;
BitmapFactory.decodeSream(is, null, options);
```

- 3、inScaled，inDensity和inTargetDensity实现更细的缩放图片：当inScaled设置为true时，系统会按照现有的密度来划分目标密度，代码如下所示：

```
BitampFactory.Options options = new BitampFactory.Options();
options.inScaled = true;
options.inDensity = srcWidth;
options.inTargetDensity = dstWidth;
BitmapFactory.decodeStream(is, null, options);
```

上述三种方案的缺点：使用了过多的算法，导致图片显示过程需要更多的时间开销，如果图片很多的话，就影响到图片的显示效果。最好的方案是结合这两个方法，达到最佳的性能结合，首先使用inSampleSize处理图片，转换为接近目标的2次幂，然后用inDensity和inTargetDensity生成最终想要的准确大小，因为inSampleSize会减少像素的数量，而基于输出密码的需要对像素重新过滤。但获取资源图片的大小，需要设置位图对象的inJustDecodeBounds值为true，然后继续解码图片文件，这样才能生产图片的宽高数据，并允许继续优化图片。总体的代码如下所示：

```
BitmapFactory.Options options = new BitampFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeStream(is, null, options);
options.inScaled = true;
```

```
options.inDensity = options.outWidth;
options.inSampleSize = 4;
Options.inTargetDensity = desWith * options.inSampleSize;
options.inJustDecodeBounds = false;
BitmapFactory.decodeStream(is, null, options);
```

4、inBitmap

可以结合LruCache来实现，在LruCache移除超出cache size的图片时，暂时缓存Bitamp到一个软引用集合，需要创建新的Bitamp时，可以从这个软用用集合中找到最适合重用的Bitmap，来重用它的内存区域。

注意：新申请的Bitmap与旧的Bitmap必须有相同的解码格式，并且在Android 4.4之前，只能重用相同大小的Bitamp的内存区域，而Android 4.4之后可以重用任何bitmap的内存区域。

五、图片管理模块设计与实现

在设计一个模块时，需要考虑以下几点：

- 1、单一职责
- 2、避免不同功能之间的耦合
- 3、接口隔离

在编写代码前先画好UML图，确定每一个对象、方法、接口的功能，首先尽量做到功能单一原则，在这个基础上，再明确模块与模块的直接关系，最后使用代码实现。

5.1 实现异步加载功能

1.实现网络图片显示

ImageLoader是实现图片加载的基类，其中ImageLoader有一个内部类BitmapLoadTask是继承AsyncTask的异步下载管理类，负责图片的下载和刷新，MiniImageLoader是ImageLoader的子类，维护类一个ImageLoader的单例，并且实现了基类的网络加载功能，因为具体的下载在应用中有不同的下载引擎，抽象成接口便于替换。代码如下所示：

```
public abstract class ImageLoader {
    private boolean mExitTasksEarly = false;   //是否提前结束
    protected boolean mPauseWork = false;
    private final Object mPauseWorkLock = new   Object();
    protected ImageLoader() {
    }
    public void loadImage(String url, ImageView imageView) {
        if (url == null) {
            return;
        }
        BitmapDrawable bitmapDrawable = null;
        if (bitmapDrawable != null) {
```

```
13              imageView.setImageDrawable(bitmapDrawable);
14          } else {
15              final BitmapLoadTask task = new BitmapLoadTask(url, imag
    eView);
16              task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
17          }
18      }
19      private class BitmapLoadTask extends AsyncTask<Void, Void, Bitma
    p> {
20          private String mUrl;
21          private final WeakReference<ImageView> imageViewWeakReferenc
    e;
22          public BitmapLoadTask(String url, ImageView imageView) {
23              mUrl = url;
24              imageViewWeakReference = new WeakReference<ImageView>(im
    ageView);
25          }
26          @Override
27          protected Bitmap doInBackground(Void... params) {
28              Bitmap bitmap = null;
29              BitmapDrawable drawable = null;
30              synchronized (mPauseWorkLock) {
31                  while (mPauseWork && !isCancelled()) {
32                      try {
33                          mPauseWorkLock.wait();
34                      } catch (InterruptedException e) {
35                          e.printStackTrace();
36                      }
37                  }
38              }
39              if (bitmap == null
40                      && !isCancelled()
41                      && imageViewWeakReference.get() != null
42                      && !mExitTasksEarly) {
43                  bitmap = downLoadBitmap(mUrl);
44              }
45              return bitmap;
46          }
47          @Override
48          protected void onPostExecute(Bitmap bitmap) {
```

```
            if (isCancelled() || mExitTasksEarly) {
                bitmap = null;
            }
            ImageView imageView = imageViewWeakReference.get();
            if (bitmap != null && imageView != null) {
                setImageBitmap(imageView, bitmap);
            }
        }
        @Override
        protected void onCancelled(Bitmap bitmap) {
            super.onCancelled(bitmap);
            synchronized (mPauseWorkLock) {
                mPauseWorkLock.notifyAll();
            }
        }
    }
    public void setPauseWork(boolean pauseWork) {
        synchronized (mPauseWorkLock) {
            mPauseWork = pauseWork;
            if (!mPauseWork) {
                mPauseWorkLock.notifyAll();
            }
        }
    }
    public void setExitTasksEarly(boolean exitTasksEarly) {
        mExitTasksEarly = exitTasksEarly;
        setPauseWork(false);
    }
    private void setImageBitmap(ImageView imageView, Bitmap bitmap)
{
        imageView.setImageBitmap(bitmap);
    }
    protected abstract Bitmap downLoadBitmap(String    mUrl);
}
```

setPauseWork方法是图片加载线程控制接口，pauseWork控制图片模块的暂停和继续工作，一般在listView等控件中，滑动时停止加载图片，保证滑动流畅。另外，具体的图片下载和解码是和业务强相关的，因此在ImageLoader中不做具体的实现，只是定义类一个抽象方法。

MiniImageLoader是一个单例，保证一个应用只维护一个ImageLoader，减少对象开销，并管理应用中所有的图片加载。MiniImageLoader代码如下所示：

```
public class MiniImageLoader extends ImageLoader {
    private volatile static MiniImageLoader sMiniImageLoader = null;
    private ImageCache mImageCache = null;
    public static MiniImageLoader getInstance() {
        if (null == sMiniImageLoader) {
            synchronized (MiniImageLoader.class) {
                MiniImageLoader tmp = sMiniImageLoader;
                if (tmp == null) {
                    tmp = new MiniImageLoader();
                }
                sMiniImageLoader = tmp;
            }
        }
        return sMiniImageLoader;
    }
    public MiniImageLoader() {
        mImageCache = new ImageCache();
    }
    @Override
    protected Bitmap downLoadBitmap(String mUrl) {
        HttpURLConnection urlConnection = null;
        InputStream in = null;
        try {
            final URL url = new URL(mUrl);
            urlConnection = (HttpURLConnection) url.openConnection
();
            in = urlConnection.getInputStream();
            Bitmap bitmap = decodeSampledBitmapFromStream(in, null);
            return bitmap;
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (urlConnection != null) {
                urlConnection.disconnect();
                urlConnection = null;
            }
            if (in != null) {
```

```
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return null;
    }
    public Bitmap decodeSampledBitmapFromStream(InputStream is, Bitm
apFactory.Options options) {
        return BitmapFactory.decodeStream(is, null, options);
    }
}
```

其中，volatile保证了对象从主内存加载。并且，上面的try ...cache层级太多，Java中有一个Closeable接口，该接口标识类一个可关闭的对象，因此可以写如下的工具类：

```
public class CloseUtils {
    public static void closeQuietly(Closeable closeable) {
        if (null != closeable) {
            try {
                closeable.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

改造后如下所示：

```
finally {
    if  (urlConnection != null) {
        urlConnection.disconnect();
    }
    CloseUtil.closeQuietly(in);
}
```

同时，为了使ListView在滑动过程中更流畅，在滑动时暂停图片加载，减少系统开销，代码如下所示：

```
listView.setOnScrollListener(new AbsListView.OnScrollListener() {
    @Override
    public void onScrollStateChanged(AbsListView absListView, int scrollState) {
        if (scorllState == AbsListView.OnScrollListener.SCROLL_STAE_FLING) {
            MiniImageLoader.getInstance().setPauseWork(true);
        } else {
            MiniImageLoader.getInstance().setPauseWork(false);
        }
    }
}
```

2 单个图片内存优化

这里使用一个BitmapConfig类来实现参数的配置，代码如下所示:

```
public class BitmapConfig {
    private int mWidth, mHeight;
    private Bitmap.Config mPreferred;
    public BitmapConfig(int width, int height) {
        this.mWidth = width;
        this.mHeight = height;
        this.mPreferred = Bitmap.Config.RGB_565;
    }
    public BitmapConfig(int width, int height, Bitmap.Config preferred) {
        this.mWidth = width;
        this.mHeight = height;
        this.mPreferred = preferred;
    }
    public BitmapFactory.Options getBitmapOptions() {
        return getBitmapOptions(null);
    }
    // 精确计算，需要图片is流现解码，再计算宽高比
    public BitmapFactory.Options getBitmapOptions(InputStream is) {
        final BitmapFactory.Options options = new BitmapFactory.Options();
        options.inPreferredConfig = Bitmap.Config.RGB_565;
        if (is != null) {
            options.inJustDecodeBounds = true;
```

```
            BitmapFactory.decodeStream(is, null, options);
            options.inSampleSize = calculateInSampleSize(options, mWidth, mHeight);
        }
        options.inJustDecodeBounds = false;
        return options;
    }
    private static int calculateInSampleSize(BitmapFactory.Options options, int mWidth, int mHeight) {
        final int height = options.outHeight;
        final int width = options.outWidth;
        int inSampleSize = 1;
        if (height > mHeight || width > mWidth) {
            final int halfHeight = height / 2;
            final int halfWidth = width / 2;
            while ((halfHeight / inSampleSize) > mHeight
                    && (halfWidth / inSampleSize) > mWidth) {
                inSampleSize *= 2;
            }
        }
        return inSampleSize;
    }
}
```

然后，调用MiniImageLoader的downLoadBitmap方法，增加获取BitmapFactory.Options的步骤：

```
final URL url = new URL(urlString);
urlConnection = (HttpURLConnection) url.openConnection();
in = urlConnection.getInputStream();
final BitmapFactory.Options options =    mConfig.getBitmapOptions(in);
in.close();
urlConnection.disconnect();
urlConnection = (HttpURLConnection)    url.openConnection();
in = urlConnection.getInputStream();
Bitmap bitmap = decodeSampledBitmapFromStream(in,    options);
```

优化后仍存在一些问题：

- 1.相同的图片，每次都要重新加载；

- 2.整体内存开销不可控，虽然减少了单个图片开销，但是在片非常多的情况下，没有合理管理机制仍然对性能有严重影的。

为了解决这两个问题，就需要有内存池的设计理念，通过内存池控制整体图片内存，不重新加载和解码已经显示过的图片。

5.2 实现三级缓存

内存一本地一网络

1、内存缓存

使用软引用和弱引用（SoftReference or WeakReference）来实现内存池是以前的常用做法，但是现在不建议。从API 9起（Android 2.3）开始，Android系统垃圾回收器更倾向于回收持有软引用和弱引用的对象，所以不是很靠谱，从Android 3.0开始（API 11）开始，图片的数据无法用一种可遇见的方式将其释放，这就存在潜在的内存溢出风险。

使用LruCache来实现内存管理是一种可靠的方式，它的主要算法原理是把最近使用的对象用强引用来存储在LinkedHashMap中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。使用LruCache实现一个图片的内存缓存的代码如下所示：

```
public class MemoryCache {
    private final int DEFAULT_MEM_CACHE_SIZE = 1024 * 12;
    private LruCache<String, Bitmap> mMemoryCache;
    private final String TAG = "MemoryCache";
    public MemoryCache(float sizePer) {
        init(sizePer);
    }
    private void init(float sizePer) {
        int cacheSize = DEFAULT_MEM_CACHE_SIZE;
        if (sizePer > 0) {
            cacheSize = Math.round(sizePer * Runtime.getRuntime().ma
xMemory() / 1024);
        }
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
            @Override
            protected int sizeOf(String key, Bitmap value) {
                final int bitmapSize = getBitmapSize(value) / 1024;
                return bitmapSize == 0 ? 1 : bitmapSize;
            }
            @Override
            protected void entryRemoved(boolean evicted, String key,
Bitmap oldValue, Bitmap newValue) {
               super.entryRemoved(evicted, key, oldValue, newValue);
            }
```

```
        };
    }
    @TargetApi(Build.VERSION_CODES.KITKAT)
    public int getBitmapSize(Bitmap bitmap) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
            return bitmap.getAllocationByteCount();
        }
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR1) {
            return bitmap.getByteCount();
        }
        return bitmap.getRowBytes() * bitmap.getHeight();
    }
    public Bitmap getBitmap(String url) {
        Bitmap bitmap = null;
        if (mMemoryCache != null) {
            bitmap = mMemoryCache.get(url);
        }
        if (bitmap != null) {
            Log.d(TAG, "Memory cache exiet");
        }
        return bitmap;
    }
    public void addBitmapToCache(String url, Bitmap bitmap) {
        if (url == null || bitmap == null) {
            return;
        }
        mMemoryCache.put(url, bitmap);
    }
    public void clearCache() {
        if (mMemoryCache != null) {
            mMemoryCache.evictAll();
        }
    }
}
```

上述代码中cacheSize百分比占比多少合适？可以基于以下几点来考虑：

- 1.应用中内存的占用情况，除了图片以外，是否还有大内存的数据需要缓存到内存。
- 2.在应用中大部分情况要同时显示多少张图片，优先保证最大图片的显示数量的缓存支持。
- 3.Bitmap的规格，计算出一张图片占用的内存大小。

- 4.图片访问的频率。

在应用中，如果有一些图片的访问频率要比其它的大一些，或者必须一直显示出来，就需要一直保持在内存中，这种情况可以使用多个LruCache对象来管理多组Bitmap，对Bitmap进行分级，不同级别的Bitmap放到不同的LruCache中。

2、bitmap内存复用

从Android3.0开始Bitmap支持内存复用，也就是BitmapFactoy.Options.inBitmap属性，如果这个属性被设置有效的目标用对象，decode方法就在加载内容时重用已经存在的bitmap，这意味着Bitmap的内存被重新利用，这可以减少内存的分配回收，提高图片的性能。代码如下所示：

```
1 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
2         mReusableBitmaps = Collections.synchronizedSet(newHashSet<Sof
  tReference<Bitmap>>());
3 }
```

因为inBitmap属性在Android3.0以后才支持，在entryRemoved方法中加入软引用集合，作为复用的源对象，之前是直接删除，代码如下所示：

```
1 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
2     mReusableBitmaps.add(new SoftReference<Bitmap>(oldValue));
3 }
```

同样在3.0以上判断，需要分配一个新的bitmap对象时，首先检查是否有可复用的bitmap对象：

```
 1 public static Bitmap decodeSampledBitmapFromStream(InputStream is, B
   itmapFactory.Options options, ImageCache cache) {
 2      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
 3          addInBitmapOptions(options, cache);
 4      }
 5      return BitmapFactory.decodeStream(is, null, options);
 6  }
 7 @TargetApi(Build.VERSION_CODES.HONEYCOMB)
 8 private static void addInBitmapOptions(BitmapFactory.Options option
   s, ImageCache cache) {
 9      options.inMutable = true;
10      if (cache != null) {
11          Bitmap inBitmap = cache.getBitmapFromReusableSet(options);
12          if (inBitmap != null) {
13              options.inBitmap = inBitmap;
14          }
15      }
```

```
 }
```

接着，我们使用cache.getBitmapForResubleSet方法查找一个合适的bitmap赋值给inBitmap。代码如下所示：

```
// 获取inBitmap,实现内存复用
public Bitmap getBitmapFromReusableSet(BitmapFactory.Options options) {
    Bitmap bitmap = null;
    if (mReusableBitmaps != null && !mReusableBitmaps.isEmpty()) {
        final Iterator<SoftReference<Bitmap>> iterator = mReusableBitmaps.iterator();
        Bitmap item;
        while (iterator.hasNext()) {
            item = iterator.next().get();
            if (null != item && item.isMutable()) {
                if (canUseForInBitmap(item, options)) {
                    Log.v("TEST", "canUseForInBitmap!!!!");
                    bitmap = item;
                    // Remove from reusable set so it can't be used again
                    iterator.remove();
                    break;
                }
            } else {
                // Remove from the set if the reference has been cleared.
                iterator.remove();
            }
        }
    }
    return bitmap;
}
```

上述方法从软引用集合中查找规格可利用的Bitamp作为内存复用对象，因为使用inBitmap有一些限制，在Android 4.4之前，只支持同等大小的位图。因此使用了canUseForInBitmap方法来判断该Bitmap是否可以复用，代码如下所示：

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private static boolean canUseForInBitmap(
        Bitmap candidate, BitmapFactory.Options targetOptions) {
```

```
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT) {
        return candidate.getWidth() == targetOptions.outWidth
                && candidate.getHeight() == targetOptions.outHeight
                && targetOptions.inSampleSize == 1;
    }
    int width = targetOptions.outWidth / targetOptions.inSampleSize;
    int height = targetOptions.outHeight / targetOptions.inSampleSiz
e;
    int byteCount = width * height * getBytesPerPixel(candidate.getC
onfig());
    return byteCount <= candidate.getAllocationByteCount();
}
```

3、磁盘缓存

由于磁盘读取时间是不可预知的，所以图片的解码和文件读取都应该在后台进程中完成。DisLruCache是Android提供的一个管理磁盘缓存的类。

- 1、首先调用DiskLruCache的open方法进行初始化，代码如下：

```
public static DiskLruCache open(File directory, int appVersion, int v
alueCou9nt, long maxSize)
```

directory一般建议缓存到SD卡上。appVersion发生变化时，会自动删除前一个版本的数据。valueCount是指Key与Value的对应关系，一般情况下是1对1的关系。maxSize是缓存图片的最大缓存数据大小。初始化DiskLruCache的代码如下所示：

```
private void init(final long cacheSize,final File cacheFile) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (mDiskCacheLock) {
                if(!cacheFile.exists()){
                    cacheFile.mkdir();
                }
                MLog.d(TAG,"Init DiskLruCache cache path:" + cacheFi
le.getPath() + "\r\n" + "Disk Size:" + cacheSize);
                try {
                    mDiskLruCache = DiskLruCache.open(cacheFile, Min
iImageLoaderConfig.VESION_IMAGELOADER, 1, cacheSize);
                    mDiskCacheStarting = false;
                    // Finished initialization
```

```
                    mDiskCacheLock.notifyAll();
                    // Wake any waiting threads
                }catch(IOException e){
                    MLog.e(TAG,"Init err:" + e.getMessage());
                }
            }
        }
    }).start();
}
```

如果在初始化前就要操作写或者读会导致失败，所以在整个DiskCache中使用的Object的wait/notifyAll机制来避免同步问题。

- 2、写入DiskLruCache

首先，获取Editor实例，它需要传入一个key来获取参数，Key必须与图片有唯一对应关系，但由于URL中的字符可能会带来文件名不支持的字符类型，所以取URL的MD4值作为文件名，实现Key与图片的对应关系，通过URL获取MD5值的代码如下所示:

```
private String hashKeyForDisk(String key) {
    String cacheKey;
    try {
        final MessageDigest mDigest = MessageDigest.getInstance("MD
5");
        mDigest.update(key.getBytes());
        cacheKey = bytesToHexString(mDigest.digest());
    } catch (NoSuchAlgorithmException e) {
        cacheKey = String.valueOf(key.hashCode());
    }
    return cacheKey;
}
private String bytesToHexString(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < bytes.length; i++) {
        String hex = Integer.toHexString(0xFF & bytes[i]);
        if (hex.length() == 1) {
            sb.append('0');
        }
        sb.append(hex);
    }
    return sb.toString();
}
```

然后，写入需要保存的图片数据，图片数据写入本地缓存的整体代码如下所示：

```
public void saveToDisk(String imageUrl, InputStream in) {
    // add to disk cache
    synchronized (mDiskCacheLock) {
        try {
            while (mDiskCacheStarting) {
                try {
                    mDiskCacheLock.wait();
                } catch (InterruptedException e) {}
            }
            String key = hashKeyForDisk(imageUrl);
            MLog.d(TAG,"saveToDisk get key:" + key);
            DiskLruCache.Editor editor = mDiskLruCache.edit(key);
            if (in != null && editor != null) {
                // 当 valueCount指定为1时，index传0即可
                OutputStream outputStream = editor.newOutputStream
(0);
                MLog.d(TAG, "saveToDisk");
                if (FileUtil.copyStream(in,outputStream)) {
                    MLog.d(TAG, "saveToDisk commit start");
                    editor.commit();
                    MLog.d(TAG, "saveToDisk commit over");
                } else {
                    editor.abort();
                    MLog.e(TAG, "saveToDisk commit abort");
                }
            }
            mDiskLruCache.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

接着，读取图片缓存，通过DiskLruCache的get方法实现，代码如下所示：

```
public Bitmap  getBitmapFromDiskCache(String imageUrl,BitmapConfig b
itmapconfig) {
    synchronized (mDiskCacheLock) {
```

```
        // Wait while disk cache is started from background thread
        while (mDiskCacheStarting) {
            try {
                mDiskCacheLock.wait();
            } catch (InterruptedException e) {}
        }
        if (mDiskLruCache != null) {
            try {
                String key = hashKeyForDisk(imageUrl);
                MLog.d(TAG,"getBitmapFromDiskCache get key:" + key);
                DiskLruCache.Snapshot snapShot = mDiskLruCache.get(k
ey);
                if(null == snapShot){
                    return null;
                }
                InputStream is = snapShot.getInputStream(0);
                if(is != null){
                    final BitmapFactory.Options options = bitmapconf
ig.getBitmapOptions();
                    return BitmapUtil.decodeSampledBitmapFromStream
(is, options);
                }else{
                    MLog.e(TAG,"is not exist");
                }
            }catch (IOException e){
                MLog.e(TAG,"getBitmapFromDiskCache ERROR");
            }
        }
    }
    return null;
}
```

最后，要注意读取并解码Bitmap数据和保存图片数据都是有一定耗时的IO操作。所以这些方法都是在ImageLoader中的doInBackground方法中调用，代码如下所示：

```
@Override
protected Bitmap doInBackground(Void... params) {
    Bitmap bitmap = null;
    synchronized (mPauseWorkLock) {
        while (mPauseWork && !isCancelled()) {
```

```
            try {
                mPauseWorkLock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    if (bitmap == null && !isCancelled()
            && imageViewReference.get() != null && !mExitTasksEarly)
{
        bitmap = getmImageCache().getBitmapFromDisk(mUrl, mBitmapCon
fig);
    }
    if (bitmap == null && !isCancelled()
            && imageViewReference.get() != null && !mExitTasksEarly)
{
        bitmap = downLoadBitmap(mUrl, mBitmapConfig);
    }
    if (bitmap != null) {
        getmImageCache().addToCache(mUrl, bitmap);
    }
    return bitmap;
}
```

5.3 图片加载三方库

目前使用最广泛的有Picasso、Glide和Fresco。Glide和Picasso比较相似，但是Glide相对于Picasso来说，功能更多，内部实现更复杂，对Glide有兴趣的同学可以阅读这篇文章Android主流三方库源码分析（三、深入理解Glide源码）。Fresco最大的亮点在与它的内存管理，特别实在低端机和Android 5.0以下的机器上的优势更加明显，而使用Fresco将很好地解决图片占用内存大的问题。因为，Fresco会将图片放到一个特别的内存区域，当图片不再显示时，占用的内存会自动释放。以下总结以下其优点：

- 1、内存管理。
- 2、渐进式呈现：先呈现大致的图片轮廓，然后随着图片下载的继续，呈现逐渐清晰的图片。
- 3、支持更多的图片格式:如Gif和Webp。
- 4、图像加载策略丰富：其中的Image Pipeline可以为同一个图片指定不同的远程路径，比如先显示已经存在本地缓存中的图片，等高清图下载完成之后在显示高清图集。

缺点：

- 安装包过大，所以对图片加载和显示要求不是比较高的情况下建议使用Glide。

六、内存优化

对于内存泄漏，其本质可理解为无法回收无用的对象。这里我总结了我在项目中遇到的一些常见的内存泄漏案例（包含解决方案）和常见的内存优化技术。

6.1、常见的内存泄漏案例(完善3.3小节）：

- 1、单例造成的内存泄漏（使用Application的Context）
- 2、非静态内部类创建静态实例造成的内存泄漏（将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，就使用Application的Context）
- 3、Handler造成的内存泄漏（将Handler类独立出来或者使用静态内部类）
- 4、线程造成的内存泄漏（将AsyncTask和Runnable类独立出来或者使用静态内部类）
- 5、BraodcastReceiver、Bitmap等资源未关闭造成的内存泄漏（应该在Activity销毁时及时关闭或者注销）
- 6、使用ListView时造成的内存泄漏（在构造Adapter时，使用缓存的convertView）
- 7、集合容器中的内存泄露（在退出程序之前，将集合里的东西clear，然后置为null，再退出程序）
- 8、WebView造成的泄露（为WebView另外开启一个进程，通过AIDL与主线程进行通信，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）

6.2、常见的内存优化点：

1、只需要UI提供一套高分辨率的图，图片建议放在drawable-xxhdpi文件夹下，这样在低分辨率设备中图片的大小只是压缩，不会存在内存增大的情况。如若遇到不需缩放的文件，放在drawable-nodpi文件夹下。

2、图片优化：

- 颜色模式：RGB_8888->RGB_565
- 降低图片大小
- 降低采样率

3、在App退到后台内存紧张即将被Kill掉时选择重写onTrimMemory()方法去释放掉图片缓存、静态缓存来自保。

4、item被回收不可见时释放掉对图片的引用：

- ListView：因此每次item被回收后再次利用都会重新绑定数据，只需在ImageView onDetachFromWindow的时候释放掉图片引用即可。
- RecyclerView：因为被回收不可见时第一选择是放进mCacheView中，这里item被复用并不会只需bindViewHolder来重新绑定数据，只有被回收进mRecyclePool中后拿出来复用才会重新绑定数据，因此重写Recycler.Adapter中的onViewRecycled()方法来使item被回收进RecyclePool的时候去释放图片引用。

5、集合优化：Android提供了一系列优化过后的数据集合工具类，如SparseArray、SparseBooleanArray、LongSparseArray，使用这些API可以让我们的程序更加高效。HashMap工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间。

6、避免创作不必要的对象：字符串拼接使用StringBuffer，StringBuilder。

7、onDraw方法里面不要执行对象的创建。

8、使用static final 优化成员变量。

9、使用增强型for循环语法。
10、在没有特殊原因的情况下，尽量使用基本数据类型来代替封装数据类型，int比Integer要更加有效，其它数据类型也是一样。
11、适当采用软引用和弱引用。
12、采用内存缓存和磁盘缓存。
13、尽量采用静态内部类，可避免潜在由于内部类导致的内存泄漏。

七、总结

对于内存优化，一般都是通过使用MAT等工具来进行检查和使用LeakCanary等内存泄漏监控工具来进行监控，以此来发现问题，再分析问题原因，解决发现的问题或者对当前的实现逻辑进行优化，优化完后在进行检查，直到达到预定的性能指标。下一篇，将会深入分析一下Android系统的存储优化相关技术，尽请期待~

参考链接：

1、Android应用性能优化最佳实践
2、必知必会 | Android 性能优化的方面方面都在这儿