安卓内存模型、内存管理与优化总结

deepwaterooo

January 17, 2024

Contents

1 常见内存泄漏场景

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- 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）。

1.1 【资源性对象未】关闭释放

- 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，然后再置为 null。例如 Bitmap 等资源未关闭会造成内存泄漏，此时我们应该在 Activity 销毁时及时关闭。
- 这种，安卓中的使用情况狠多，可以列举如下：

```
1.BroadcastReceiver 没有反注册：安卓广播，四大组件之一，深入透彻理解与使用
2.Cursor 没有及时关闭：数据库操作相关，远程服务器中数据库的查询读写等
3. 各种流没有关闭：文件流？跨进程消息流？
4.Bitmap 没有调用 recycle 进行回收
5.Activity 退出时，没有取消 RxJava 和协程开启的异步任务：这个，改天还要再总结一下
6.Webview：项目实现一下
7.EventBus 未注销造成的内存泄漏，我们应该在 Activity 销毁时及时注销。【注册对象未注销】
```

1.2 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。我们可以为 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。
- 一般情况下，在应用中只要使用一次 Webview，它占用的内存就不会被释放，解决方案：我们可以为 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。

1.3 【静态变量】：存储、持有大数据对象

- 尽量避免使用静态变量存储数据，特别是大数据对象，建议使用数据库存储。
- 前面分析过静态变量是存储在方法区的，而方法区是一个生命周期比较长，不容易被回收的区域，如果静态变量存储的数据内存占用较大，就很容易出现内存泄露并发生 OOM。

1.4 单例

- 优先使用 Application 的 Context; 如需使用 Activity 的 Context，可以在传入 Context 时使用弱引用进行封装，然后，在使用到的地方从弱引用中获取 Context，如果获取不到，则直接 return 即可。
- 单例中如果使用了 Activity 的 context，则会造成内存泄露，解决方案：使用 Application 的 context。
- 或者使用弱引用去包裹 context，在使用的时候去获取，如果获取不到，说明被回收了，返回注入一个新的 Activity 的 context。

1.5 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。此时，我们可以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，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。
- 现在原理狠透彻了，但是还是，纪录一下别人的、更为透彻的讲解，方便自己将来查询。
- 首先来讲解下【静态内部类】和【非静态内部类】区别：
- 1. 【静态内部类】不持有外部类的引用
  - 而【非静态内部类】持有外部类的引用
  - 【非静态内部类】可以访问外部类的所有属性，方法，即使是 private，就是因为这个原因，
  - 而【静态内部类】只能访问【外部类的静态方法和静态属性】。
- 2. 【静态内部类】不依赖外部类
  - 非静态内部类和外部类是寄生关系的，同生死。而静态内部类不依赖外部类，外部类被回收了，他自己并不会被回收，你可以理解为是一个新的类：编译后格式：外部类 $ 内部类。
  - 这点从构造方法也可以看出：
    * 非静态内部类：Inner in = new Outer().new Inner();

* 静态内部类：Inner in = new Outer.Inner();

- 非静态内部类需要创建一个外部对象才能创建内部，所以是共生关系。这里共生是指外部类没了，内部类也就没了，而反过来如果内部类没了，外部类是可能还存在的。
- 而【静态内部类】并没有创建一个外部对象，所以是独立存在的一个对象，形式如内部类，其实是一个新的类。
- 通过上面的分析，可知，如果是非静态的内部类的静态实例会一直持有外部类的引用，如果外部类是一个 Activity 或者持有一个 Activity 的引用，则就可能导致内存泄露，

1.6 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。解决方案如下所示：
- 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。
- 需要注意的是，AsyncTask 内部也是 Handler 机制，同样存在内存泄漏风险，但其一般是临时性的。对于类似 AsyncTask 或是线程造成的内存泄漏，我们也可以将 AsyncTask 和 Runnable 类独立出来或者使用静态内部类。

1.7 容器中的对象没清理造成的内存泄漏

- 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

1.8 使用 ListView 时造成的内存泄漏

- 在构造 Adapter 时，使用缓存的 convertView。【这个的再底层的细节与原理，改天补一下】

2 内存泄漏监控

- 一般使用 LeakCanary 进行内存泄漏的监控即可，除了其基本的使用外，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，可以拿到完整的 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.calss, 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。

3 一、Android 内存管理机制

- reference: https://juejin.cn/post/6844904096541966350

- 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。

3.1 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

3.1.1 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。

3.1.2 2、InUse（应用）

此时对象至少被一个强引用持有。

3.1.3 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。

3.1.4 4、Unreachable（不可达）

该对象不再被任何强引用持有。

3.1.5 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。

3.1.6 6、Finalized（终结）

等待垃圾回收器回收该对象空间。

3.1.7 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。

3.2 2、Java 内存分配模型

- JVM 将整个内存划分为了几块，分别如下所示：
  - 1)、方法区：存储类信息、常量、静态变量等。=> 所有线程共享
  - 2)、虚拟机栈：存储局部变量表、操作数栈等。
  - 3)、本地方法栈：不同与虚拟机栈为 Java 方法服务、它是为 Native 方法服务的。
  - 4)、堆：内存最大的区域，每一个对象实际分配内存都是在堆上进行分配的，，而在虚拟机栈中分配的只是引用，这些引用会指向堆中真正存储的对象。此外，堆也是垃圾回收器（GC）所主要作用的区域，并且，内存泄漏也都是发生在这个区域。=> 所有线程共享
  - 5)、程序计数器：存储当前线程执行目标方法执行到了第几行。

3.3 3、Android 内存分配模型

- 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，下面我们就来简单了解下。

3.3.1 Dalvik

- Linear Alloc
- Zygote Space
- Alloc Space

3.3.2 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只读的属性，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。

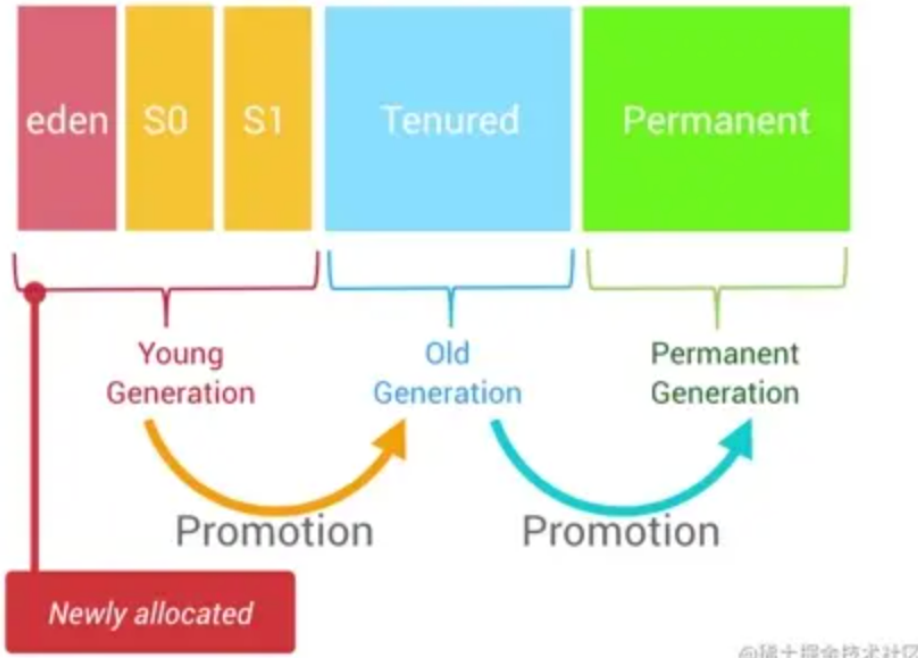
3.4 4、Java 内存回收算法

1. 1)、标记-清除算法
    - 实现原理
        - 标记出所有需要回收的对象。
        - 统一回收所有被标记的对象。
    - 特点
        - 标记和清除效率不高。
        - 产生大量不连续的内存碎片。
2. 2)、复制算法
    - 实现原理
        - 将内存划分为大小相等的两块。
        - 一块内存用完之后复制存活对象至另一块。
        - 清理另一块内存。
    - 特点
        - 实现简单，运行高效。
        - 浪费一半空间，代价大。
3. 3)、标记-整理算法
    - 实现原理
        - 标记过程与”标记-清除“算法一样。
        - 存活对象往一端进行移动。
        - 清理其余内存。
    - 特点
        - 避免”标记-清除”算法导致的内存碎片。
        - 避免复制算法的空间浪费。
4. 4)、分代收集算法（大多数虚拟机厂商所选用的算法）
    - 结合多种收集算法的优势。
    - 新生代对象存活率低 => “复制”算法（注意这里每一次的复制比例都是可以调整的，如一次仅复制 30% 的存活对象）。
    - 老年代对象存活率高 => “标记-整理”算法。

3.5 5、Android 内存回收机制

- 对于 Android 设备来说，我们每打开一个 APP，它的内存都是弹性分配的，并且其分配值与最大值是受具体设备而定的。
- 此外，我们需要注意区分如下两种 OOM 场景：
    - 1)、内存真正不足：例如 APP 当前进程最大内存上限为 512 MB，当超过这个值就表明内存真正不足了。
    - 2)、可用内存不足：手机系统内存极度紧张，就算 APP 当前进程最大内存上限为 512 MB，我们只分配了 200 MB，也会产生内存溢出，因为系统的可用内存不足了。

- 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，当此 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。(在 JDK 1.8 及之后的版本，在本地内存中实现的元空间（Meta-space）已经代替了永久代)

3.5.1 4、内存对象的处理过程小结

- 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。
- 此外，执行 GC 占用的时间与 Generation 和 Generation 中的对象数量有关，如下所示：
  - Young Generation < Old Generation < Permanent Generation
  - Generation 中的对象数量与执行时间成反比。

3.5.2 5、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。

3.5.3 6、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。

3.5.4 7、Dalvik 与 ART 区别

- 1）、Dalivk 仅固定一种回收算法。
- 2）、ART 回收算法可运行期选择。
- 3）、ART 具备内存整理能力，减少内存空洞。

3.6 6、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, external 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)
- 我们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/9351K：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）之类的数据。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。

3.7 7、Low Memory Killer 机制

- LMK 机制是针对于手机系统所有进程而制定的，当我们手机内存不足的情况下，LMK 机制就会针对我们所有进程进行回收，而其对于不同的进程，它的回收力度也是有不同的，目前系统的进程类型主要有如下几种:
  - 1)、前台进程
  - 2)、可见进程
  - 3)、服务进程
  - 4)、后台进程
  - 5)、空进程
- 从前台进程到空进程，进程优先级会越来越低，因此，它被 LMK 机制杀死的几率也会相应变大。此外，LMK 机制也会综合考虑回收收益，这样就能保证我们大多数进程不会出现内存不足的情况。

4 二、优化内存的意义

- 优化内存的意义不言而喻，总的来说可以归结为如下四点：
  - 1、减少 OOM，提高应用稳定性。
  - 2、减少卡顿，提高应用流畅度。
  - 3、减少内存占用，提高应用后台运行时的存活率。
  - 4、减少异常发生和代码逻辑隐患。
- 需要注意的是，出现 OOM 是因为内存溢出导致，这种情况不一定会发生在相对应的代码处，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。

5 三、避免内存泄漏

- 1、内存泄漏的定义
  - 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。
- 2、使用 MAT 来查找内存泄漏: **【这片，等改天真正安装了软件，自己试执行的时候再整理】**
- 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，以达到减少内存消耗的目的。
- 使用步骤
- 1、在 eclipse.org/mat/downloa…
- 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：
- java 复制代码./hprof-conv file.hprof converted.hprof
- 4、通过 MAT 打开转换后的 HPROF 文件。
- 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 的分析维度是不同的。下面分别介绍下它们，如下所示：
- 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 这两个视图，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。
- 查找内存泄漏具体位置
- 常规方式
- 1、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。
- 2、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)
- 3、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：
- 1、在两个 HPROF 文件中，把 Histogram 或者 Dominator Tree 增加到 Compare Basket。
- 2、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。
- 需要注意的是，如果目标不太明确，可以直接定位 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。
- 高效方式
- 在本人平时的项目开发中，一般会使用如下几种方式来快速对指定页面进行内存泄漏的检测（也称为运行时内存分析优化）：
- 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、从首页开始用依次 dump 出每个页面的内存快照文件，然后利用 MAT 的对比功能，找出每个页面相对于上个页面内存里主要增加了哪些东西，做针对性优化。
- 4、利用 Android Memory Profiler 实时观察进入每个页面后的内存变化情况，然后对产生的内存较大波峰做分析。
- 此外，除了运行时内存的分析优化，我们还可以对 App 的静态内存进行分析与优化。静态内存指的是在伴随着 App 的整个生命周期一直存在的那部分内存，那我们怎么获取这部分内存快照呢?
- 首先，确保打开每一个主要页面的主要功能，然后回到首页，进开发者选项去打开" 不保留后台活动"。然后，将我们的 app 退到后台，GC，dump 出内存快照。最后，我们就可以将对 dump 出的内存快照进行分析，看看有哪些地方是可以优化的，比如加载的图片、应用中全局的单例数据配置、静态内存与缓存、埋点数据、内存泄漏等等。

6 四、优化内存空间

6.1 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。下面，我们就分别来介绍下这几种引用。
- 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。
  - 这里要注意，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。

6.2 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
- 对于内存复用，有如下四种可行的方式：
  - 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。

6.3 3、使用最优的数据类型

6.3.1 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。
- 此外，Android 自身还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。

6.3.2 2、使用 IntDef 和 StringDef 替代枚举类型

- 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'
```

6.3.3 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：表明缓存对象是为了腾出空间而被清理。否则，表明缓存对象的 entry 是被 remove 移除或者被 put 覆盖。
- 注意
  - 分配 LruCache 大小时应考虑应用剩余内存有多大。

6.4 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。下面，我总结了减少内存开销的几种常用方式，如下所示：
- 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 就是宽和高都变为原来 1/4 大小的图片
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);
```

6.5 5、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，可以从这个软引用集合中找到最适合重用的 Bitmap，来重用它的内存区域。

- 需要注意，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。

6.6 6、图片放置优化

- 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。

6.7 7、在 App 可用内存过低时主动释放内存

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

6.8 8、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。

6.9 9、避免创作不必要的对象

- 例如，我们可以在字符串拼接的时候使用 StringBuffer，StringBuilder。

6.10 10、自定义 View 中的内存优化

- 例如，在 onDraw 方法里面不要执行对象的创建，一般来说，都应该在自定义 View 的构造器中创建对象。

6.11 11、其它的内存优化注意事项

- 除了上面的一些内存优化点之外，这里还有一些内存优化的点我们需要注意，如下所示：
- 尽使用 static final 优化成员变量。
- 使用增强型 for 循环语法。
- 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。
- 在合适的时候适当采用软引用和弱引用。
- 采用内存缓存和磁盘缓存。
- 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。

7 五、图片管理模块的设计与实现：

- 【其实，亲爱的表哥的活宝妹，今天不想弄图片相关】，可是顺手，还是 5 分钟，把这个文档整理完，用作亲爱的表哥的活宝妹将来的学习参考
- 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。
- 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) {
        imageView.setImageDrawable(bitmapDrawable);
    } else {
        final BitmapLoadTask task = new BitmapLoadTask(url, imageView);
        task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
    }
}

private class BitmapLoadTask extends AsyncTask<Void, Void, Bitmap> {

    private String mUrl;
    private final WeakReference<ImageView> imageViewWeakReference;
    public BitmapLoadTask(String url, ImageView imageView) {
        mUrl = url;
        imageViewWeakReference = new WeakReference<ImageView>(imageView);
    }

    @Override
    protected Bitmap doInBackground(Void... params) {
        Bitmap bitmap = null;
        BitmapDrawable drawable = null;
        synchronized (mPauseWorkLock) {
            while (mPauseWork && !isCancelled()) {
                try {
                    mPauseWorkLock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        if (bitmap == null
                && !isCancelled()
                && imageViewWeakReference.get() != null
                && !mExitTasksEarly) {
            bitmap = downLoadBitmap(mUrl);
        }
        return bitmap;
   }

   @Override
   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, BitmapFactory.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() {
^^I^^Ireturn 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```java = 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。

7.1 2、实现三级缓存

- 内存--本地--网络

7.1.1 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：

```java
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().maxMemory() / 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。

7.1.2 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：

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        mReusableBitmaps = Collections.synchronizedSet(newHashSet<SoftReference<Bitmap>>());
}
```

- 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：

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

- 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：

```
public static Bitmap decodeSampledBitmapFromStream(InputStream is, BitmapFactory.Options options, ImageCache cache) {
     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
         addInBitmapOptions(options, cache);
     }
     return BitmapFactory.decodeStream(is, null, options);
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
private static void addInBitmapOptions(BitmapFactory.Options options, ImageCache cache) {
     options.inMutable = true;
     if (cache != null) {
         Bitmap inBitmap = cache.getBitmapFromReusableSet(options);
         if (inBitmap != null) {
             options.inBitmap = inBitmap;
         }

}
```

- 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.inSampleSize;
    int byteCount = width * height * getBytesPerPixel(candidate.getConfig());
    return byteCount <= candidate.getAllocationByteCount();
}
```

7.1.3 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. 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 valueCou9nt, 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:" + cacheFile.getPath() + "\r\n" + "Disk Size:" + cacheSize)
                try {
                    mDiskLruCache = DiskLruCache.open(cacheFile, MiniImageLoaderConfig.VESION_IMAGELOADER, 1, cacheSi
                    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. 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("MD5");
        mDigest.update(key.getBytes());
        cacheKey = bytesToHexString(mDigest.digest());
    } catch (NoSuchAlgorithmException e) {
        cacheKey = String.valueOf(key.hashCode());
```

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

- 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 bitmapconfig) {
    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(key);
               if(null == snapShot){
                   return null;
               }
               InputStream is = snapShot.getInputStream(0);
               if(is != null){
                   final BitmapFactory.Options options = bitmapconfig.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, mBitmapConfig);
    }
    if (bitmap == null && !isCancelled()
            && imageViewReference.get() != null && !mExitTasksEarly) {
        bitmap = downLoadBitmap(mUrl, mBitmapConfig);
    }
    if (bitmap != null) {
        getmImageCache().addToCache(mUrl, bitmap);
    }

    return bitmap;
}
```

7.2 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，功能更丰富，内部实现更复杂，而 Fresco 最大的亮点在于它的内存管理，特别是在低端机和 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。这里总结下 Fresco 的优点，如下所示：
  - 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。

8 六、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，优化完后再进行检查，直到达到预定的性能指标。