Android 性能优化 - 详解内存优化的来龙去脉

前言

APP内存的使用，是评价一款应用性能高低的一个重要指标。虽然现在智能手机的内存越来越大，但是一个好的应用应该将效率发挥到极致，精益求精。

什么是内存

通常情况下我们说的内存是指手机的RAM，它主要包括一下几个部分：
- 寄存器（Registers读音：['rɛdʒɪstɚ]）
速度最快的存储场所，因为寄存器位于处理器内部，所以在程序中我们无法控制。
- 栈（Stack）
存放**基本类型的对象和引用**，但是对象本身不存放在栈中，而是存放在堆中。

变量其实是分为两部分的：一部分叫变量名，另外一部分叫变量值，对于**局部变量**（基本类型的变量和对象的引用变量）而言，统一都存放在栈中，但是变量值中存储的内容就有在一定差异了：Java中存在8大基本类型，他们的变量值中存放的就是具体的数值，而其他的类型都叫做引用类型（对象也是引用类型，你只要记住**除了基本类型，都是引用类型**）他们的变量值中存放的是他们在堆中的引用（内存地址）。

在函数执行的时候，函数内部的局部变量就会在栈上创建，函数执行结束的时候这些存储单元会被自动释放。**栈内存分配运算内置于处理器的指令集中是一块连续的内存区域，效率很高，速度快，但是大小是操作系统预定好的所以分配的内存容量有限。**

- 堆（Heap）
  在堆上分配内存的过程称作 内存动态分配过程。在java中堆用于存放由new创建的对象和数组。堆中分配的内存，由java虚拟机自动垃圾回收器（GC）来管理(可见我们要进行的内存优化主要就是对堆内存进行优化)。堆是不连续的内存区域（因为系统是用链表来存储空闲内存地址，自然不是连续的），堆大小受限于计算机系统中有效的虚拟内存（32bit系统理论上是4G）

- 静态存储区/方法区（Static Field）
  是指在固定的位置上存放应用程序运行时一直存在的数据，java在内存中专门划分了一个静态存储区域来管理一些特殊的数据变量如静态的数据变量。

- 常量池（Constant Pool）
  顾名思义专门存放常量的。注意 String s = "java"中的“java”也是常量。JVM虚拟机为每个已经被转载的类型维护一个常量池。常量池就是该类型所有用到地常量的一个**有序集合**包括直接常量（基本类型，String）和对其他类型、字段和方法的符号引用。

总结：

1. 定义一个局部变量的时候，java虚拟机就会在栈中为其分配内存空间，局部变量的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。因为它们属于方法中的变量，生命周期随方法而结束。

2. 成员变量全部存储与堆中（包括基本数据类型，引用和引用的对象实体），因为它们属于类，类对象终究是要被new出来使用的。当堆中对象的作用域结束的时候，这部分内存也不会立刻被回收，而是等待系统GC进行回收。

3. 所谓的内存分析，就是分析Heap中的内存状态。

Android中的沙盒机制

大家可能都听说过iOS中有沙盒机制（sandbox），但是我们的Android系统中也存在沙盒机制，只不过没有IOS中的严格，所以常常被人忽略。

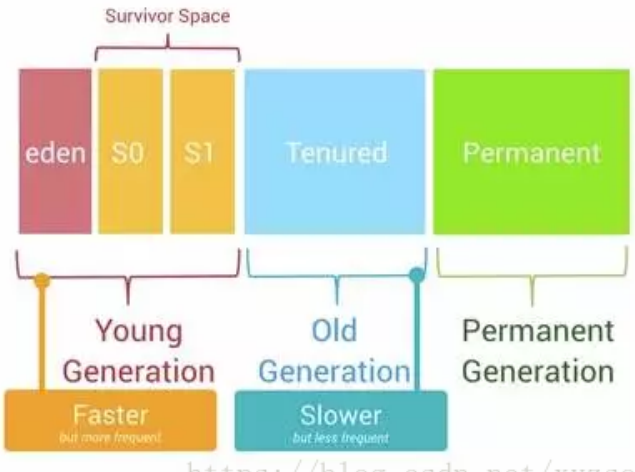
由于Android是建立在Linux系统之上的，所以Android系统继承了Linux的 *类Unix继承进程隔离机制与最小权限原则*，并且在原有Linux的进程管理基础上对UID的使用做了改进，形成了Android应用的”沙箱“机制。

> 普通的Linux中启动的应用通常和登陆用户相关联，同一用户的UID相同。但是Android中给不同的应用都赋予了不同的UID，这样不同的应用将不能相互访问资源。对应用而言，这样会更加封闭，安全。
> 引文来自Android的SandBox（沙箱）
>
> 在Android系统中，应用（通常）都在一个独立的沙箱中运行，即每一个Android应用程序都在它自己的进程中运行，都拥有一个独立的Dalvik虚拟机实例。Dalvik经过优化，允许在有限的内存中同时高效地运行多个虚拟机的实例，并且每一个Dalvik应用作为一个独立的Linux进程执行。Android这种基于Linux的进程“沙箱”机制，是整个安全设计的基础之一。
> 引文来自浅析Android沙箱模型

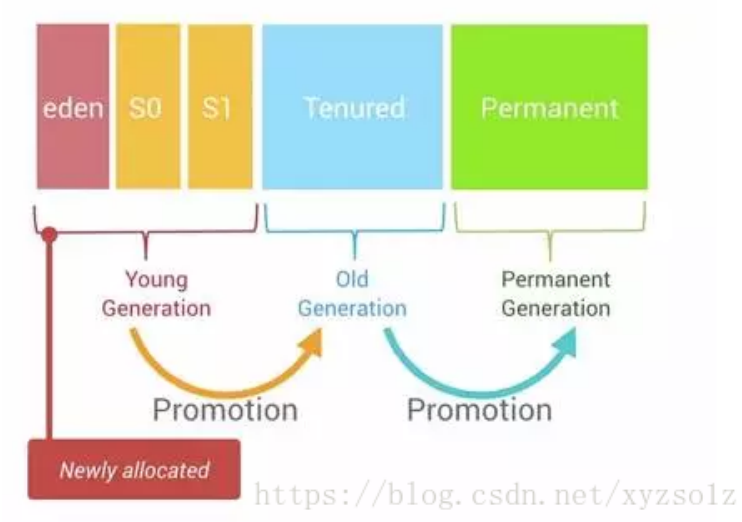
简单点说就是在Android的世界中每一个应用相当与一个Linux中的用户，他们相互独立，不能相互共享与访问，（这也就解释了Android系统中为什么需要进程间通信），**正是由于沙盒机制的存在最大程度的保护了应用之间的安全，但是也带来了每一个应用所分配的内存大小是有限制的问题。**

Generational Heap Memory内存模型的概述

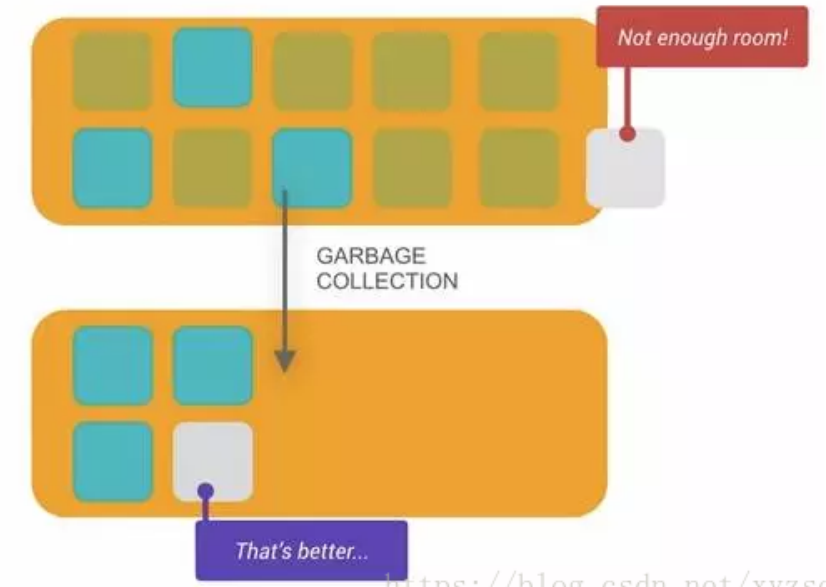
在Android和Java中都存在着一个Generational（读音：[ˌdʒenəˈreɪʃənl]） Heap Memory模型，**系统会根据内存中不同的内存数据类型分别执行不同的GC操作**。Generational Heap Memory模型主要由：Young Generation（新生代）、Old Generation（旧生代）、Permanent(读音：[ˈpɜ:rmənənt]) Generation三个区域组成，而且这三个区域存在明显的层级关系。所以此模型也可以成为**三级Generation的内存模型**。



其中Young Generation区域存放的是最近被创建对象，此区域最大的特点就是创建的快，被销毁的也很快。当对象在Young Generation区域停留的时间到达一定程度的时候，它就会被移动到Old Generation区域中，同理，最后他将会被移动到Permanent Generation区域中。



在三级Generation内存模型中，每一个区域的大小都是有固定值的，当进入的对象总大小到达某一级内存区域阀值的时候就会触发GC机制，进行垃圾回收，腾出空间以便其他对象进入。



不仅如此，不同级别的Generation区域GC是需要的时间也是不同的。同等对象数目下，Young Generation GC所需时间最短，Old Generation次之，Permanent Generation 需要的时间最长。当然GC执行的长短也和当前Generation区域中的对象数目有关。遍历查找20000个对象比起遍历50个对象自然是要慢很多的。

GC机制概述

与C++不用，在Java中，内存的分配是由程序完成的，而内存的释放是由垃圾收集器(Garbage Collection，GC)完成的，程序员不需要通过调用函数来释放内存，但也随之带来了内存泄漏的可能。简单点说：**对于 C++ 来说，内存泄漏就是new出来的对象没有 delete，俗称野指针；而对于 java 来说，就是 new 出来的 Object 放在 Heap 上无法被 GC回收。**

Android使用的主要开发语言是Java所以二者的GC机制原理也大同小异，所以我们只对于常见的JVM GC机制的分析，就能达到我们的目的。我还是先看看那二者的不同之处吧。

- **Dalvik 和标准Java虚拟机的主要区别**

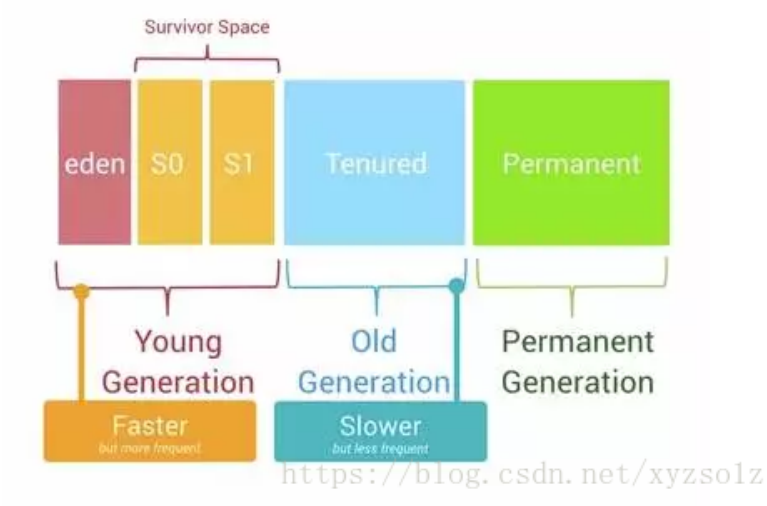
Dalvik虚拟机（DVM）是Android系统在java虚拟机（JVM）基础上优化得到的，DVM是基于寄存器的，而JVM是基于栈的，由于寄存器高效快速的特性，DVM的性能相比JVM更好。

Dalvik 和 java 字节码的区别

Dalvik执行.dex格式的字节码文件，JVM执行的是.class格式的字节码文件，Android程序在编译之后产生的.class 文件会被aapt工具处理生成R.class等文件，然后dx工具会把.class文件处理成.dex文件，最终资源文件和.dex文件等打包成.apk文件。

对于Young Generation（新生代）的GC

由于Young Generation通常存活的时间比较短，所以Young Generation采用了Copying算法进行回收，Copying算法就是扫描出存活的对象，并复制到一块新的空间中，这个过程就是下图Eden与Survivor Space之间的复制过程。Young Generation采用空闲指针的方式来控制GC触发，指针保存最后一个分配在Young Generation中分配空间地对象的位置。当有新的对象要分配内存空间的时候，就会主动检测空间是否足够，不够的情况下就出触发GC，当连续分配对象时，对象会逐渐从Eden移动到Survivor，最后移动到Old Generation。



对于Old Generation（旧生代）的GC

Old Generation与Young Generation不同，对象存活的时间比较长，比较稳固，因此采用标记（Mark）算法来进行回收。所谓标记就是扫描出存活的对象，然后在回收未必标记的对象。回收后的剩余空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。

如何判断对象是否可以被回收

从上面的一小节中我们知道了不同的区域GC机制是有所不同的，那么这些垃圾是如何被发现的呢？下面我们就看一下两种常见的判断方法：引用计数、对象引用遍历。

引用计数器

引用计数器是垃圾收集器中的早起策略。这种方法中，每个对象实体（不是它的引用）都有一个引用计数器。当一个对象创建的时候，且将该对象分配给一个每分配给一个变量，计数器就+1，当一个对象的某个引用超过了生命周期或者被设置一个新值时，对象计数器就-1，任何引用计数器为 0 的对象可以被当作垃圾收集。当一个对象被垃圾收集时，引用的任何对象技术 - 1。

*优点：*执行快，交织在程序运行中，对程序不被长时间打断的实时环境比较有利。
*缺点：*无法检测出循环引用。比如：对象A中有对象B的引用，而B中同时也有A的引用。

- **跟踪收集器**

现在的垃圾回收机制已经不太使用引用计数器的方法判断是否可回收，而是使用跟踪收集器方法。

现在大多数JVM采用对象引用遍历机制从程序的主要运行对象(如静态对象/寄存器/栈上指向的堆内存对象等)开始检查引用链，去递归判断对象收否可达，如果不可达，则作为垃圾回收，当然在便利阶段，GC必须记住那些对象是可达的，以便删除不可到达的对象，这称为标记（marking）对象。

下一步，GC就要删除这些不可达的对象，在删除时未必标记的对象，释放它们的内存的过程叫做清除（sweeping），而这样会造成内存碎片化，布局已分配给新的对象，但是他们集合起来还很大。所以很多GC机制还要重新组织内存中的对象，并进行压缩，形成大块、可利用的空间。

为了达到这个目的，GC需要停止程序的其他活动，阻塞进程。这里我们要注意的是：**不要频繁的引发GC，执行GC操作的时候，任何线程的任何操作都会需要暂停，等待GC操作完成之后，其他操作才能够继续运行, 故而如果程序频繁GC, 自然会导致界面卡顿.** 通常来说，单个的GC并不会占用太多时间，但是大量不停的GC操作则会显著占用帧间隔时间(16ms)。如果在帧间隔时间里面做了过多的GC操作，那么自然其他类似计算，渲染等操作的可用时间就变得少了。

Android内存泄露分析

对于 C++ 来说，内存泄漏就是new出来的对象没有 delete，俗称野指针；而对于 java 来说，就是 new 出来的 Object 放在 Heap 上无法被GC回收

GC过程与对象的引用类型是严重相关的，下面我们就看看Java中（Android中存在差异）对于引用的四种分类：

- 强引用（Strong Reference）:JVM宁愿抛出OOM，也不会让GC回收的对象
- 软引用（Soft Reference）：只有内存不足时，才会被GC回收。
- 弱引用（weak Reference）：在GC时，一旦发现弱引用，立即回收
- 虚引用（Phantom Reference）：任何时候都可以被GC回收，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否存在该对象的虚引用，来了解这个对象是否将要被回收。可以用来作为GC回收Object的标志。

级别	回收时机	用途	生存时间
强	从来不会	对象的一般状态	JVM停止运行时终止
软	在内存不足时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的二级高速缓冲器（内存不足才清空）	内存不足时终止
弱	在垃圾回收时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的一级高速缓冲器（系统发生gc则清空）	gc运行后终止
虚	在垃圾回收时	联合ReferenceQueue来跟踪对象被垃圾回收器回收的活动	gc运行后终止

注意Android中存在的差异
但是在2.3以后版本中，系统会优先将SoftReference的对象提前回收掉，即使内存够用，其他和Java中是一样的。所以谷歌官方建议用LruCache(least recentlly use 最少最近使用算法)。会将内存控制在一定的大小内，超出最大值时会自动回收，这个最大值开发者自己定。其实LruCache就是用了很多的HashMap，三百多行的代码

在开发过程中，保存对象，这时我很可以直接使用LruCache来代替，Bitmap对象：

在Android开发过程中，我们常常使用HasMap保存对象，但是为了防止内存泄漏，在保存内存占用较大、生命周期较长的对象的时候，尽量使用LruCache代替HasMap用于保存对象。

```
//指定最大缓存空间
private static final int MAX_SIZE = (int) (Runtime.getRuntime().maxMemory() / 8);
LruCache<String,Bitmap> mBitmapLruCache = new LruCache<>(MAX_SIZE);
```

而造成不能回收的根本原因就是：**堆内存中长生命周期的对象持有短生命周期对象的强/软引用，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收。**

如何监听系统发送GC

那么怎样才能去监听系统的GC过程呢？其实非常简单，系统每进行一次GC操作时，都会在LogCat中打印一条日志，我们只要去分析这条日志就可以了，日志的基本格式如下所示：
DVM中

D/dalvikvm(30615): GC FOR ALLOC freed 4442K, 25% free 20183K/26856K, paused 24ms , total 24ms

ART中

I/art(198): Explicit concurrent mark sweep GC freed 700(30KB) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 186us total 12.763ms

D/dalvikvm: <GC_Reason> <Amount_freed>, <Heap_stats>, <Pause_time>

原因，一般情况下一共有以下几种触发GC操作的原因：

- GC_CONCURRENT: 当我们应用程序的堆内存快要满的时候，系统会自动触发GC操作来释放内存。
- GC_FOR_MALLOC: 当我们的应用程序需要分配更多内存，可是现有内存已经不足的时候，系统会进行GC操作来释放内存。
- GC_HPROF_DUMP_HEAP: 当生成HPROF文件的时候，系统会进行GC操作，关于HPROF文件我们下面会讲到。
- GC_EXPLICIT: 这种情况就是我们刚才提到过的，主动通知系统去进行GC操作，比如调用System.gc()方法来通知系统。或者在DDMS中，通过工具按钮也是可以显式地告诉系统进行GC操作的。

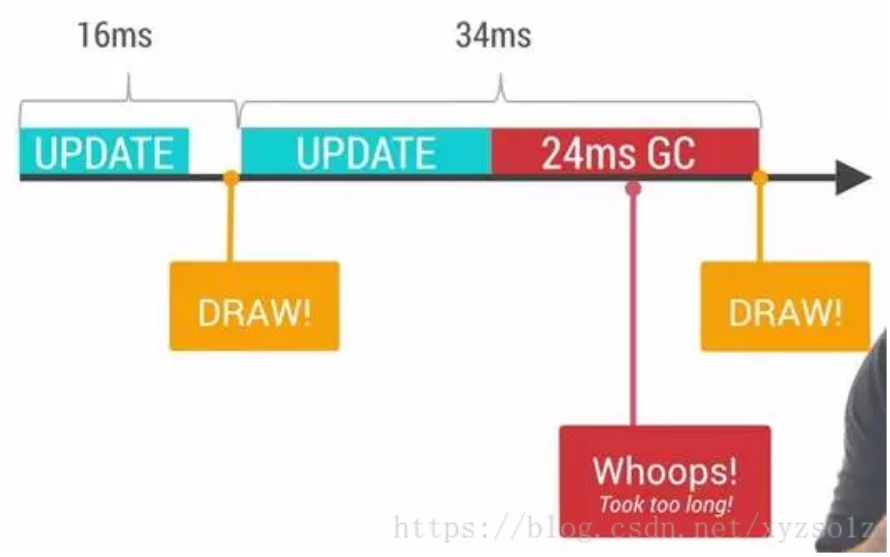
接下来第二部分Amount_freed，表示系统通过这次GC操作释放了多少内存。
然后Heap_stats中会显示当前内存的空闲比例以及使用情况（活动对象所占内存 / 当前程序总内存）。

最后Pause_time表示这次GC操作导致应用程序暂停的时间。

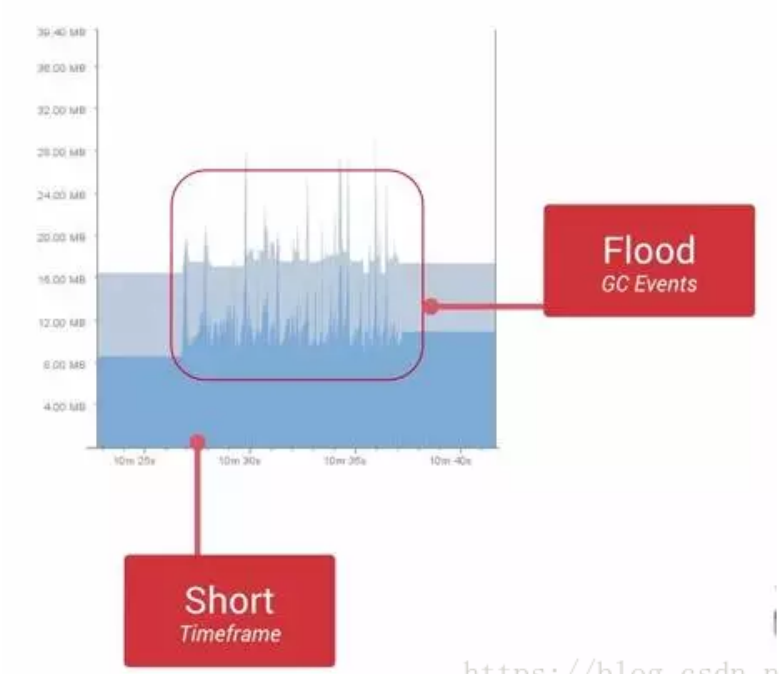
关于这个暂停的时间，Android在2.3的版本当中进行过一次优化，在2.3之前GC操作是不能并发进行的，也就是系统正在进行GC，那么应用程序就只能阻塞住等待GC结束。虽说这个阻塞的过程并不会很长，也就是几百毫秒，但是用户在使用我们的程序时还是有可能会感觉到略微的卡顿。
而自2.3之后，GC操作改成了并发的方式进行，就是说GC的过程中不会影响到应用程序的正常运行，但是在GC操作的开始和结束的时候会短暂阻塞一段时间，不过优化到这种程度，用户已经是完全无法察觉到了。

导致GC频繁执行有两个原因

由于GC会阻塞进程，所以我们不避免频繁的GC。
1. **Memory Churn（内存抖动）**，内存抖动是因为大量的对象被创建又在短时间内马上被释放。
2. 瞬间产生大量的对象会严重占用Young Generation的内存区域，当达到阀值，剩余空间不够的时候，也会触发GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加 Heap的压力，从而触发更多其他类型的GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。



解决上面的问题有简洁直观方法，如果你在**Memory Monitor**里面查看到短时间发生了多次内存的涨跌，这意味着很有可能发生了内存抖动。



常见内存泄露分析

1. 永远的单例（Singleton）

为了完美解决我们在程序中反复创建同一对象的问题，我们选用了单例模式，单例在我们的程序中随处可见，**但是由于单例模式的静态特性，使得它的生命周期和我们的应用一样长，一不小心让单例无限制的持有Activity的强引用就会导致内存泄漏。**例如：

```
public class SingleTon{

    private Context context;
    private static SingleTon singleTon;


    public static final SingleTon getInstance(Context context){
```

```
        this.context = context;
        return SingleHolder.INSTANCE;

    }
    private static class SingleHolder{

        private static final SingleTon INSTANCE = new SingleTon();

    }

}
```

解决办法：

这个错误很普遍，这个是一个很正常的单利模式，但是由于传入了一个Context，而这个Context的生命周期就的长短就尤为重要了。如果我们传入的是某个Activity的Context，而当这个Activity推出的时候，由于该Context的强引用被单例持有，那么这个Activity就等同于拥有了整个程序的生命周期。这种情况下，当Activity退出的时候内存并没有被回收，这就造成了内存泄漏。

正确的做法就是应该把传入的Context改为同应用生命周期一样长的Application中的Context。

```
public class BaseApplication extends Application{
    private static BaseApplication baseApplication;

    @Override
    public void onCreate(){
        super.onCreate();
        baseApplication = this;

    }
    public static Context getContext{
        baseApplication.getApplicationContext();
    }
}
```

当然我们可以直接重写Application，提供getContext方法,不必在依靠传入的参数：

```
 public static final SingleTon getInstance(Context context) {
        this.context = context.getApplicationContext;
        return SingleHolder.INSTANCE;
  }
```

2 Handler引起的内存泄漏

Handler引起的内存泄漏在我们开发中最为常见的。我们知道Handler、Message、MessageQueue都是相互关联在一起的，万一Handler发送的Message尚未被处理，那么该Message以及发送它的Handler对象都会被线程MessageQueue一直持有。

由于Handler属于TLS（Thread Local Storage）变量，生命周期和Activity是不一致的，因此这种实现方式很难保证跟Activity的生命周期一直，所以很容易无法释放内存。比如：

```
public class HandlerBadActivity extends AppCompatActivity {

    private final Handler handler = new Handler(){
       @Override
      public void handleMessage(Message msg) {
          super.handleMessage(msg);
     }
     };

     @Override
     protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_handler_bad);
         // 延迟5min发送一个消息
         handler.postDelayed(new Runnable() {
             @Override
             public void run() {
                 // write something
             }
         },1000*60*5);
         this.finish();
     }
}
```

我们在例子中生命了一个延时5分钟执行的Message，当该Activity退出的时候，延时任务（Message）还在主线成的MessageQueue中等待，此时的Message持有Handler的强引用，并且由于Handler是HandlerBadActivity的**非静态内部类，所以Handler会持有HandlerBadActivity的强引用**，此时HandlerBadActivity退出时无法进行内存回收，造成内存泄漏。

解决办法：

将Handler生命为静态内部类，这样它就不会持有外部来的引用了。这样以来Handler的的生命周期就与Activity无关了。不过倘若用到Context等外部类的非static对象，还是应该通过使用Application中与应用同生命周期的Context比较合适。比如：

```
public class HandlerGoodActivity extends AppCompatActivity {
    private static final class MyHandler extends Handler {
        private Context mActivity;
        public MyHandler(HandlerGoodActivity activity) {
            //使用生命周期与应用同长的getApplicationContext
            this.mActivity = activity.getApplicationContext();
        }

        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            if (mActivity != null) {
                // write something
            }
        }
    }

    private final MyHandler myHandler = new MyHandler(this);
```

```
    // 匿名内部类在static的时候绝对不会持有外部类的引用
    private static final Runnable RUNNABLE = new Runnable() {
        @Override
        public void run() {

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler_good);
        myHandler.postDelayed(RUNNABLE, 1000 * 60 * 5);

    }
```

虽然我们结局了Activity的内存泄漏问题，但是经过Handler发送的延时消息还在MessageQueue中，Looper也在等待处理消息，所以我们要在Activity销毁的时候处理掉队列中的消息。

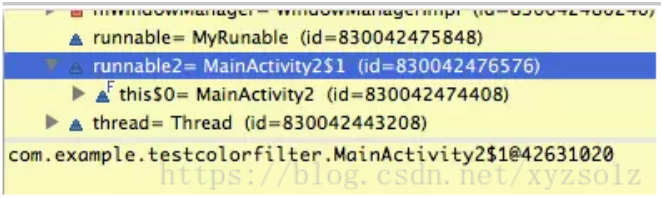
```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        //传入null，就表示移除所有Message和Runnable
        myHandler.removeCallbacksAndMessages(null);
    }
```

3 匿名内部类在异步线程中的使用

它们方便却暗藏杀机。Android开发经常会继承实现 Activity 或者 Fragment 或者 View。如果你使用了匿名类，而又被异步线程所引用，那得小心，如果没有任何措施同样会导致内存泄漏的：

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_inner_bad);
        Runnable runnable1 = new MyRunnable();
        Runnable runnable2 = new Runnable() {
            @Override
            public void run() {

            }
        };
    }

    private static class MyRunnable implements Runnable{
        @Override
        public void run() {

        }
    }
}
```

runnable1 和 runnable2的区别就是，runnable2使用了匿名内部类，我们看看引用时的引用内存



可以看到，runnable1是没有什么特别的。但runnable2多出了一个MainActivity的引用，若是这个引用再传入到一个异步线程，此线程在和Activity生命周期不一致的时候，也就造成了Activity的泄露。

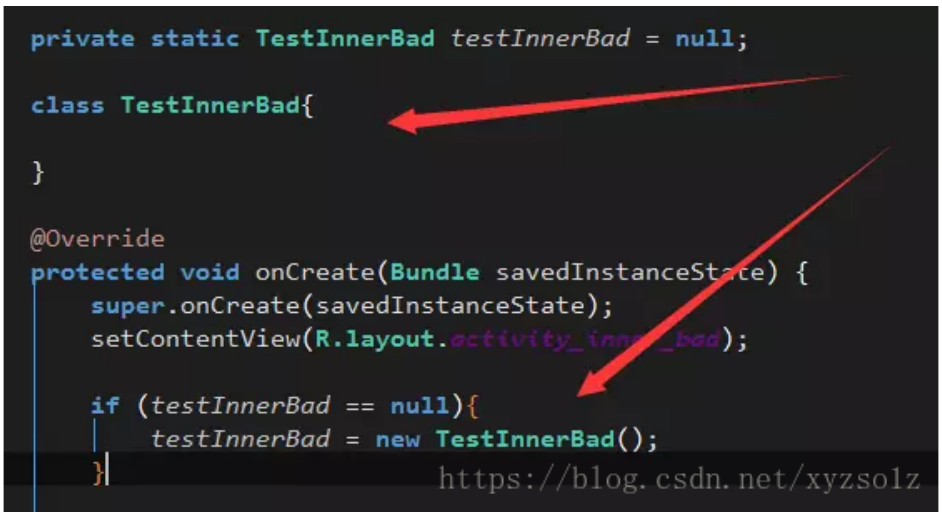
4 善用static成员变量

从前面的介绍我们知道，static修饰的变量位于内存的静态存储区，此变量与App的生命周期一致
这必然会导致一系列问题，如果你的app进程设计上是长驻内存的，那即使app切到后台，这部分内存也不会被释放。按照现在手机app内存管理机制，占内存较大的后台进程将优先回收，因为如果此app做过进程互保保活，那会造成app在后台频繁重启。当手机安装了你参与开发的app以后一夜时间手机被消耗空了电量、流量，你的app不得不被用户卸载或者静默。

这里修复的方法是：
不要在类初始时初始化静态成员。可以考虑lazy初始化（延迟加载）。架构设计上要思考是否真的有必要这样做，尽量避免。如果架构需要这么设计，那么此对象的生命周期你有责任管理起来。

5 避免使用

在我们的日常代码中，这样的情况似乎很常见，及直接写一个class就这么光秃秃的情况



这样就在Activity内部创建了一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据，这样虽然避免了资源的重复创建，不过这种写法却会造成内存泄漏，因为非静态内部类默认会持有外部类的引用，而该非静态内部类又创建了一个静态的实例，该实例的生命周期和应用的一样长，这就导致了该静态实例一直会持有该Activity的引用，导致Activity的内存资源不能正常回收。正确的做法为：

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，请按照上面推荐的使用 Application 的 Context。当然，Application 的 context 不是万能的，所以也不能随便乱用，对于有些地方则必须使用 Activity 的 Context，对于Application，Service，Activity三者的Context的应用场景如下：

功能	Application	Service	Activity
Start an Activity	NO1	NO1	YES
Show a Dialog	NO	NO	YES
Layout Inflation	YES	YES	YES
Start a Service	YES	YES	YES
Bind to a Service	YES	YES	YES
Send a Broadcast	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES
Load Resource Values	YES	YES	YES

其中： NO1表示 Application 和 Service 可以启动一个 Activity，不过需要创建一个新的 task 任务队列。而对于 Dialog 而言，只有在 Activity 中才能创建。

6 集合引发的内存泄漏

我们通常会把一些对象的引用加入到集合容器（比如ArrayList）中，当我们不再需要该对象时，并没有把它的引用从集合中清理掉，当集合中的内容过于大的时候，并且是static的时候就造成了内存泄漏，所有我们最好在onDestory情况并让其不可达

```
private List<String> nameList;
    private List<Fragment> list;
    @Override
    public void onDestroy() {
        super.onDestroy();
        if (nameList != null){
            nameList.clear();
            nameList = null;
        }
        if (list != null){
            list.clear();
            list = null;
        }
    }
```

7 webView引发的内存泄漏

WebView解析网页时会申请Native堆内存用于保存页面元素，当页面较复杂时会有很大的内存占用。如果页面包含图片，内存占用会更严重。并且打开新页面时，为了能快速回退，之前页面占用的内存也不会释放。有时浏览十几个网页，都会占用几百兆的内存。这样加载网页较多时，会导致系统不堪重负，最终强制关闭应用，也就是出现应用闪退或重启。

由于占用的都是Native堆内存，所以实际占用的内存大小不会显示在常用的DDMS Heap工具中（这里看到的只是Java虚拟机分配的内存，一般即使Native堆内存已经占用了几百兆，这里显示的还只是几兆或十几兆）。只有使用adb shell中的一些命令比如dumpsys meminfo 包名，或者在程序中使用Debug.getNativeHeapSize()才能看到。

据说由于WebView的一个BUG，即使它所在的Activity(或者Service)结束也就是onDestroy()之后，或者直接调用WebView.destroy()之后，它所占用这些内存也不会被释放。

解决这个问题最直接的方法是：把使用了WebView的Activity(或者Service)放在单独的进程里。然后在检测到应用占用内存过大有可能被系统干掉或者它所在的Activity(或者Service)结束后，调用System.exit(0)，主动Kill掉进程。由于

系统的内存分配是以进程为准的，进程关闭后，系统会自动回收所有内存。

8其他常见的引起内存泄漏原因

- 构造Adapter时，没有使用缓存的 convertView
- Bitmap在不使用的时候没有使用recycle()释放内存
- 非静态内部类的静态实例容易造成内存泄漏：即一个类中如果你不能够控制它其中内部类的生命周期（譬如Activity中的一些特殊Handler等），则尽量使用静态类和弱引用来处理（譬如ViewRoot的实现）。
- 警惕线程未终止造成的内存泄露；譬如在Activity中关联了一个生命周期超过Activity的Thread，在退出Activity时切记结束线程。一个典型的例子就是HandlerThread的run方法是一个死循环，它不会自己结束，线程的生命周期超过了Activity生命周期，我们必须手动在Activity的销毁方法中中调运thread.getLooper().quit();才不会泄露。
- 对象的注册与反注册没有成对出现造成的内存泄露；譬如注册广播接收器、注册观察者（典型的譬如数据库的监听）等。
- 创建与关闭没有成对出现造成的泄露；譬如Cursor资源必须手动关闭，WebView必须手动销毁，流等对象必须手动关闭等。
- 不要在执行频率很高的方法或者循环中创建对象（比如onMeasure），可以使用HashTable等创建一组对象容器从容器中取那些对象，而不用每次new与释放。
- 避免代码设计模式的错误造成内存泄露；譬如循环引用，A持有B，B持有C，C持有A，这样的设计谁都得不到释放。

总结

- Android内存优化主要是针对堆（Heap）而言的，当堆中对象的作用域结束的时候，这部分内存也不会立刻被回收，而是等待系统GC进行回收。
- Java中造成内存泄漏的根本原因是：堆内存中长生命周期的对象持有短生命周期对象的强/软引用，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收。

原文：Android 性能优化 - 详解内存优化的来龙去脉
相关文章：https://www.cnblogs.com/lianghe01/p/6617275.html