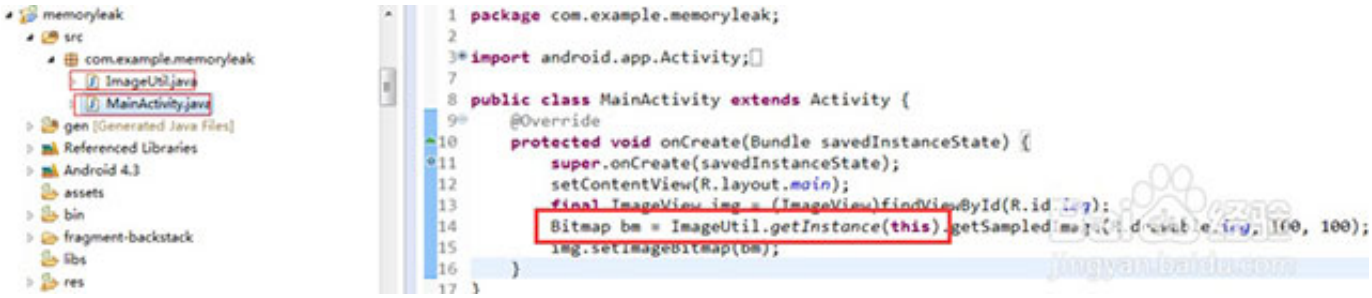
内存泄露从入门到精通三部曲之基础知识篇

2015.11.5 腾讯Bugly 微信分享

一、首先以一个内存泄露实例来开始本节基础概念的内容：

实例1：(单例导致内存对象无法释放而泄露)




```
public class ImageUtil {
    private static ImageUtil instance;
    private Context context;
    private ImageUtil(Context context){
        this.context = context;
    }
    public static ImageUtil getInstance(Context context){
        if(instance == null){
            instance = new ImageUtil(context);
        }
        return instance;
    }

    public Bitmap getSampledImage(int resId, int reqWidth, int reqHeight) {
        // First decode with inJustDecodeBounds=true to check dimensions
        final BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
```

可以看出ImageUtil这个工具类是一个单例，并引用了activity的context。

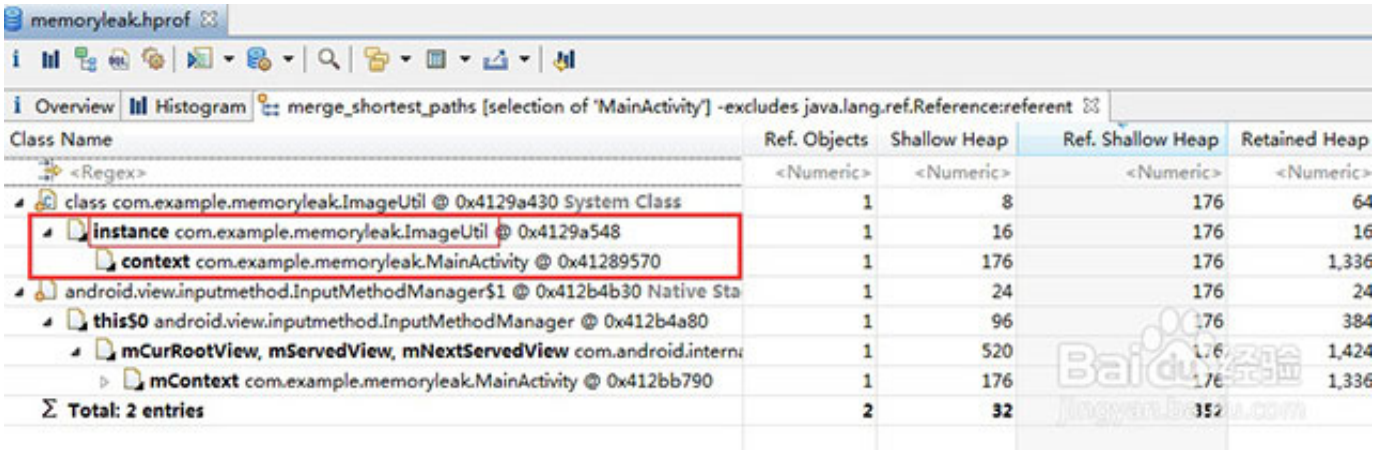
试想这个场景，应用起来以后，转屏。转屏以后，旧MainActivity会destroy，新MainActivity会重建，导致单例ImageUtil重新getInstance。很不幸的是，由于instance已经不是空的了，所以ImageUtil不会重建，还持有之前的Context，也就是之前的那个MainActivity实例的context，因此会造成两个问题：

功能问题：使用ImageUitl访问context相关内容时可能会发生异常（因为当前context并不是当前activity的context）；

内存泄露：旧context被生命周期更长的静态变量持有而导致activity无法释放造成泄漏！（因此静态变量

是很容易因此内存泄露的！）

使用工具可以看到ImageUtil引用了MainActivity导致MainActivity驻留内存发生泄漏



备注：本系列部分概念和例子引用来自网络。

二、内存泄露，我们要研究的泄露对象到底是什么?

首先我们来了解程序运行时，所需内存的分配策略：

按照编译原理的观点,程序运行时的内存分配有三种策略,分别是静态的,栈式的,和堆式的，对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、堆区和栈区。他们的功能不同，对他们使用方式也就不同。

静态存储区（方法区）：内存在程序编译的时候就已经分配好，这块内存在程序整个运行期间都存在。它主要存放静态数据、全局static数据和常量。

栈区：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆区：亦称动态内存分配。程序在运行的时候用malloc或new申请任意大小的内存，程序员自己负责在适当的时候用free或delete释放内存（Java则依赖垃圾回收器）。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。 但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉。

接下来我们集中说下堆和栈的区别：

在函数中（说明是局部变量）定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

堆内存用于存放所有由new创建的对象（内容包括该对象其中的所有成员变量）和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

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），所以堆的空间比较灵活，比较大。栈是一块连续的内存区域，大小是操作系统预定好的，windows下栈大小是2M（也有是1M，在编译时确定，VC中可设置）。

对于堆，频繁的new/delete会造成大量内存碎片，使程序效率降低。对于栈，它是先进后出的队列，进出一一对应，不产生碎片，运行效率稳定高。

举一个关于变量存储位置的实例2：

```
public class A{
    int a = 1;
    person p = new person();

    public void method(){
        int a2 = 1;
        person p2 = new person();
    }
}
A aObject = new A();
```

A类内的局部变量都存在于栈中，包括基本数据类型a2和对象引用p2。
而p2指向的person对象是存在于堆中。

aObject对象实体存放在堆中，包括这个对象的所有成员变量a和p；
而p这个引用指向的person类对象也是存在堆中。
而aObject这个引用存在与栈中。

结论：

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。

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。

回到我们的问题：内存泄露需要关注的是什么？

我们这里说的内存泄露，是针对，也只针对堆内存，他们存放的就是引用指向的对象实体。

三、那么第二个问题就是，内存为什么会泄露？

为了判断Java中是否有内存泄露，我们首先必须了解Java是如何管理（堆）内存的。Java的内存管理就是对象的分配和释放问题。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，但它只能回收无用并且不再被其它对象引用的那些对象所占用的空间。

Java的内存垃圾回收机制是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，当遍历一遍后得到上述这些无法回收的对象和他们所引用的对象链，组成无法回收的对象集合，而其他孤立对象（集）就作为垃圾回收。GC为了能够正确释放对象，必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC都需要进行监控。监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

在Java中，这些无用的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。虽然，我们有几个函数可以访问GC，例如运行GC的函数System.gc()，但是根据Java语言规范定义，该函数不保证JVM的垃圾收集器一定会执行。因为不同的JVM实现者可能使用不同的算法管理GC。通常GC的线程的优先级别较低。JVM调用GC的策略也有很多种，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，我们不需要关心这些。

至此，我们来看看Java中需要被回收的垃圾：

```
 {

Person p1 = new Person();
```

……

}

引用句柄p1的作用域是从定义到“}”处，执行完这对大括号中的所有代码后，产生的Person对象就会变成垃圾，因为引用这个对象的句柄p1已超过其作用域，p1失效，在栈中被销毁，因此堆上的Person对象不再被任何句柄引用了。 因此person变为垃圾，会被回收。

从上面的例子和解释，可以看到一个很关键的词：引用。

通俗的讲，通过A能调用并访问到B，那就说明A持有B的引用，或A就是B的引用，B的引用计数＋１.

（１）比如 Person p1 = new Person();通过Ｐ１能操作Person对象，因此Ｐ１是Person的引用；

（２）比如类O中有一个成员变量是I类对象，因此我们可以使用o.i的方式来访问I类对象的成员，因此o持有一个i对象的引用。

GC过程与对象的引用类型是严重相关的，我们来看看Java对引用的分类Strong reference, SoftReference, WeakReference, PhatomReference

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

讲多一步，这里的软引用/弱引用一般是做什么的呢?

在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

假设我们的应用会用到大量的默认图片，比如应用中有默认的头像，默认游戏图标等等，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生OutOfMemory异常。这时，我们可以考虑使用软/弱引用技术来避免这个问题发生。以下就是高速缓冲器的雏形:

首先定义一个HashMap，保存软引用对象。

1.private Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String, SoftReference<Bitmap>>();

再来定义一个方法，保存Bitmap的软引用到HashMap。

```
public class CacheBySoftRef {
    // 首先定义一个HashMap，保存软引用对象。
    private Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String,
SoftReference<Bitmap>>();
    // 再来定义一个方法，保存Bitmap的软引用到HashMap。
    public void addBitmapToCache(String path) {
        // 强引用的Bitmap对象
        Bitmap bitmap = BitmapFactory.decodeFile(path);
        // 软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = new SoftReference<Bitmap>(bitmap);
        // 添加该对象到Map中使其缓存
        imageCache.put(path, softBitmap);
    }
    // 获取的时候，可以通过SoftReference的get()方法得到Bitmap对象。
    public Bitmap getBitmapByPath(String path) {
        // 从缓存中取软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = imageCache.get(path);
        // 判断是否存在软引用
        if (softBitmap == null) {
            return null;
        }
        // 通过软引用取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空 ，如果未被回收，则可重复使
用，提高速度。
        Bitmap bitmap = softBitmap.get();
        return bitmap;
    }
}
```

使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从

而避免内存达到上限，避免Crash发生。

如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

回到我们的问题，为什么内存会泄露?

堆内存中的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，这就是Java中内存泄露的根本原因。