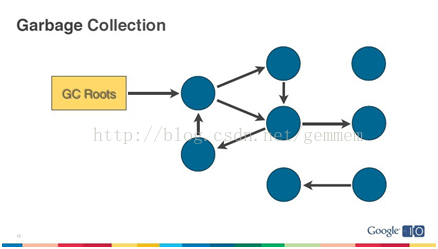
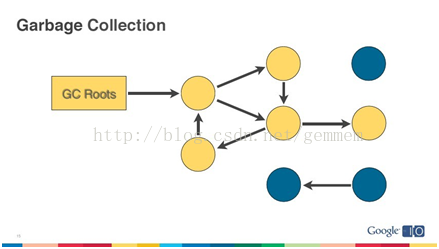
**首先了解一下dalvik的Garbage Collection：**

****

GC会选择一些它了解还存活的对象作为内存遍历的根节点（GC Roots），比方说thread stack中的变量，JNI中的全局变量，zygote中的对象（class loader加载）等，然后开始对heap进行遍历。到最后，部分没有直接或者间接引用到GC Roots的就是需要回收的垃圾，会被GC回收掉。如下图蓝色部分。



Java内存泄漏指的是进程中某些对象（垃圾对象）已经没有使用价值了，但是它们却可以直接或间接地引用到gc roots导致无法被GC回收。无用的对象占据着内存空间，使得实际可使用内存变小，形象地说法就是内存泄漏了。下面分析一些可能导致内存泄漏的情景。

**常见的内存泄漏**

1. **非静态内部类的静态实例容易造成内存泄漏**

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **public** **class** MainActivityextends Activity
2. {
3. **static** Demo sInstance = **null**;
5. @Override
6. **public** **void** onCreate(BundlesavedInstanceState)
7. {
8. **super**.onCreate(savedInstanceState);
9. setContentView(R.layout.activity\_main);
10. **if** (sInstance == **null**)
11. {
12. sInstance= **new** Demo();
13. }
14. }
15. **class** Demo
16. {
17. voiddoSomething()
18. {
19. System.out.print("dosth.");
20. }
21. }
22. }

上面的代码中的sInstance实例类型为静态实例，在第一个MainActivity act1实例创建时，sInstance会获得并一直持有act1的引用。当MainAcitivity销毁后重建，因为sInstance持有act1的引用，所以act1是无法被GC回收的，进程中会存在2个MainActivity实例（act1和重建后的MainActivity实例），这个act1对象就是一个无用的但一直占用内存的对象，即无法回收的垃圾对象。所以，对于lauchMode不是singleInstance的Activity， 应该避免在activity里面实例化其非静态内部类的静态实例。

1. **activity使用静态成员**

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **private** **static** Drawable sBackground;
2. @Override
3. **protected** **void** onCreate(Bundle state) {
4. **super**.onCreate(state);
6. TextView label = **new** TextView(**this**);
7. label.setText("Leaks are bad");
9. **if** (sBackground == **null**) {
10. sBackground = getDrawable(R.drawable.large\_bitmap);
11. }
12. label.setBackgroundDrawable(sBackground);
14. setContentView(label);
15. }

由于用静态成员sBackground 缓存了drawable对象，所以activity加载速度会加快，但是这样做是错误的。因为在android 2.3系统上，它会导致activity销毁后无法被系统回收。label .setBackgroundDrawable函数调用会将label赋值给sBackground的成员变量mCallback。上面代码意味着：sBackground（GC Root）会持有TextView对象，而TextView持有Activity对象。所以导致Activity对象无法被系统回收。

下面看看android4.0为了避免上述问题所做的改进。

先看看android 2.3的Drawable.Java对setCallback的实现：

    public final void setCallback(Callback cb){

        mCallback = cb;

}

再看看android 4.0的Drawable.Java对setCallback的实现：

    public final void setCallback(Callback cb){

        mCallback = newWeakReference<Callback> (cb);

}

在android 2.3中要避免内存泄漏也是可以做到的, 在activity的onDestroy时调用

sBackgroundDrawable.setCallback(null)。

以上2个例子的内存泄漏都是因为Activity的引用的生命周期超越了activity对象的生命周期。也就是常说的Context泄漏，因为activity就是context。

想要避免context相关的内存泄漏，需要注意以下几点：

·不要对activity的context长期引用(一个activity的引用的生存周期应该和activity的生命周期相同)

·如果可以的话，尽量使用关于application的context来替代和activity相关的context

·如果一个acitivity的非静态内部类的生命周期不受控制，那么避免使用它；正确的方法是使用一个静态的内部类，并且对它的外部类有一WeakReference，就像在ViewRootImpl中内部类W所做的那样。

1. **使用handler时的内存问题**

我们知道，Handler通过发送Message与主线程交互，Message发出之后是存储在MessageQueue中的，有些Message也不是马上就被处理的。在Message中存在一个 target，是Handler的一个引用，如果Message在Queue中存在的时间越长，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。 所以正确处理Handler等之类的内部类，应该将自己的Handler定义为静态内部类。

HandlerThread的使用也需要注意：

  当我们在activity里面创建了一个HandlerThread，代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **public** classMainActivity **extends** Activity
2. {
3. @Override
4. **public** **void** onCreate(BundlesavedInstanceState)
5. {
6. **super**.onCreate(savedInstanceState);
7. setContentView(R.layout.activity\_main);
8. Thread mThread = newHandlerThread("demo", Process.THREAD\_PRIORITY\_BACKGROUND);
9. mThread.start();
10. MyHandler mHandler = **new** MyHandler( mThread.getLooper( ) );
11. …….
12. …….
13. …….
14. }
15. @Override
16. **public** **void** onDestroy()
17. {
18. **super**.onDestroy();
19. }
20. }

这个代码存在泄漏问题，因为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，当横竖屏切换，HandlerThread线程的数量会随着activity重建次数的增加而增加。应该在onDestroy时将线程停止掉：**mThread.getLooper().quit();另外，对于不是HandlerThread的线程，也应该确保activity消耗后，线程已经终止，可以这样做：**在onDestroy时调用**mThread.join();**

**4、注册某个对象后未反注册**

注册广播接收器、注册观察者等等，比如：

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在LockScreen中定义一个PhoneStateListener的对象，同时将它注册到TelephonyManager服务中。对于LockScreen对象，当需要显示锁屏界面的时候就会创建一个LockScreen对象，而当锁屏界面消失的时候LockScreen对象就会被释放掉。

　　但是如果在释放LockScreen对象的时候忘记取消我们之前注册的PhoneStateListener对象，则会导致LockScreen无法被GC回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的LockScreen对象没有办法被回收而引起OutOfMemory,使得system\_process进程挂掉。

虽然有些系统程序，它本身好像是可以自动取消注册的(当然不及时)，但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

**5、集合中对象没清理造成的内存泄露**

　　我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，如果没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是static的话，那情况就更严重了。

比如某公司的ROM的锁屏曾经就存在内存泄漏问题：

这个泄漏是因为LockScreen每次显示时会注册几个callback，它们保存在KeyguardUpdateMonitor的ArrayList<InfoCallback>、ArrayList<SimStateCallback>等ArrayList实例中。但是在LockScreen解锁后，这些callback没有被remove掉，导致ArrayList不断增大， callback对象不断增多。这些callback对象的size并不大，heap增长比较缓慢，需要长时间地使用手机才能出现OOM，由于锁屏是驻留在system\_server进程里，所以导致结果是手机重启。

**6、资源对象没关闭造成的内存泄露**

　　资源性对象比如(Cursor，File文件等)往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于Java虚拟机内，还存在于Java虚拟机外。如果我们仅仅是把它的引用设置为null,而不关闭它们，往往会造成内存泄露。因为有些资源性对象，比如SQLiteCursor(在析构函数finalize(),如果我们没有关闭它，它自己会调close()关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该立即调用它的close()函数，将其关闭掉，然后再置为null.在我们的程序退出时一定要确保我们的资源性对象已经关闭。

　　程序中经常会进行查询数据库的操作，但是经常会有使用完毕Cursor后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在长时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

**7、一些不良代码成内存压力**

有些代码并不造成内存泄露，但是它们或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存，对内存的回收和分配造成很大影响的，容易迫使虚拟机不得不给该应用进程分配更多的内存，增加vm的负担，造成不必要的内存开支。

7.1，Bitmap使用不当

    第一、及时的销毁。

    虽然，系统能够确认Bitmap分配的内存最终会被销毁，但是由于它占用的内存过多，所以很可能会超过Java堆的限制。因此，在用完Bitmap时，要及时的recycle掉。recycle并不能确定立即就会将Bitmap释放掉，但是会给虚拟机一个暗示：“该图片可以释放了”。

  第二、设置一定的采样率。

    有时候，我们要显示的区域很小，没有必要将整个图片都加载出来，而只需要记载一个缩小过的图片，这时候可以设置一定的采样率，那么就可以大大减小占用的内存。如下面的代码：

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **private** ImageView preview;
2. BitmapFactory.Options options = newBitmapFactory.Options();
3. options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一
4. Bitmap bitmap =BitmapFactory.decodeStream(cr.openInputStream(uri), **null**, options); preview.setImage

第三、巧妙的运用软引用（SoftRefrence）

有些时候，我们使用Bitmap后没有保留对它的引用，因此就无法调用Recycle函数。这时候巧妙的运用软引用，可以使Bitmap在内存快不足时得到有效的释放。如下：

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. SoftReference<Bitmap>  bitmap\_ref  = **new** SoftReference<Bitmap>(BitmapFactory.decodeStream(inputstream));
2. ……
3. ……
4. **if** (bitmap\_ref .get() != **null**)
5. bitmap\_ref.get().recycle();
6. 7.2，构造Adapter时，没有使用缓存的 convertView
7. 以构造ListView的BaseAdapter为例，在BaseAdapter中提共了方法：
8. public View getView(intposition, View convertView, ViewGroup parent)
9. 来向ListView提供每一个item所需要的view对象。初始时ListView会从BaseAdapter中根据当前的屏幕布局实例化一定数量的view对象，同时ListView会将这些view对象缓存起来。当向上滚动ListView时，原先位于最上面的list item的view对象会被回收，然后被用来构造新出现的最下面的list item。这个构造过程就是由getView()方法完成的，getView()的第二个形参 View convertView就是被缓存起来的list item的view对象(初始化时缓存中没有view对象则convertView是null)。
10. 由此可以看出，如果我们不去使用convertView，而是每次都在getView()中重新实例化一个View对象的话，即浪费时间，也造成内存垃圾，给垃圾回收增加压力，如果垃圾回收来不及的话，虚拟机将不得不给该应用进程分配更多的内存，造成不必要的内存开支。ListView回收list item的view对象的过程可以查看:
11. android.widget.AbsListView.Java--> void addScrapView(View scrap) 方法。
12. Java代码：

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **public** View getView(**int** position, View convertView, ViewGroupparent) {
2. View view = newXxx(...);
3. **return** view;
4. }

　修正示例代码：

　　Java代码：

**[java]** [view plaincopy](http://blog.csdn.net/gemmem/article/details/13017999)

1. **public** View getView(intposition, View convertView, ViewGroup parent) {
2. View view = **null**;
3. **if** (convertView != **null**){
4. view = convertView;
5. populate(view, getItem(position));
6. } **else** {
7. view = **new** Xxx(...);
8. }
9. **return** view;
10. }

7.3、不要在经常调用的方法中创建对象，尤其是忌讳在循环中创建对象。可以适当的使用 hashtable ， vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次 new 之后又丢弃。