我们来看下OOM发生的情景:

要创建一个对象，内存够吗？够！ok创建去；不够！进行垃圾回收，回收完毕了，现在够吗？够！ok创建去，不够！抛出OOM异常。

所以，OOM发生的条件是：在垃圾回收后内存还不够！那么怎么避免呢？ 第一就是不创建那么大的对象了，也就是**避免创建大对象**；第二就是让垃圾回收后内存够用，也就是**避免内存泄漏**。

**大对象**

这里的大对象是个泛指，泛指单个的大对象，或者一群小对象的集合体。我们知道，一个对象在内存里的大小，大部分是由它的成员变量的大小决定的，所以，如果我们仅仅需要一个对象内部的一些部分数据，就尽量不要创建或持有这个对象，而是只持有我们需要的部分，比如:

public class Music {

String songName;

String singer;

String lyric;

Date date;

String icon;

//...

}

// 只需要展示歌词

public class LyricActivity {

// 但是却持有了整个Music

public class Music;

}

比如上例，我们只需要Music的lyric，但是却让Activity持有了整个Music，我们又知道，成员变量的生命周期和持有它的对象一样，所以，在Activity被回收前，Music一直存活，垃圾回收都回收不掉，这就变相耗费了内存，这是不建议的。所以，我们应该**只取需要的**，对不需要的，不要持有， 这既可以避免OOM，也符合**最少知识原则**。

#### 零碎的小对象

我们还要避免创建大量的小对象，举个例子：我们都写布局文件，在一个xml中，要展示一个控件，我们可以使用组合的方式去实现，比如上面图片，下面按钮，大部分人都会这么写:

<LinearLayout oritation="vertival">

<ImageView/>

<Button/>

</LinearLayout>

这样当然可以，但是，我们需要创建3个View，我们完全可以自己定义一个View，比如继承自系统Button，在上面绘制一个图片，这样只需要一个View就可以，直接少用2/3内存。还有对象复用，比如RecyclerView，就有自己的cache，保证创建的item对象最多只够屏幕显示即可，这也是为了避免大量的持久的小对象， 小对象虽小，但是积少成多也会造成OOM。

#### Bitmap

最重要的是Bitmap，大多数人都知道将UI给的图片进行压缩，但是压缩的是自身所占的存储空间，充其量也就是优化了apk的大小， 而没节省运行时内存，我们知道Bitmap在内存中占用的大小是:

width \* height \* 4byte

因为Bitmap默认都是ARGB\_8888加载，也就是32bit，也就是4byte，那么一个1000\*1000的图片，在内存中占用的大小就是: 1000 \* 1000 \* 4 = 4\_000\_000byte，也就是4M，而一个int才4byte，这等价于100万个int，50万个long，等价于一个长度为200万的String，所以，Bitmap就是OOM的最大祸首， 我们在压缩图片的时候，一定要记得裁剪；能用drawable就不要用Bitmap，能不设置背景就不设置背景，比如一个View，它的90%都被子View盖住了，那么背景就没必要用Bitmap了，再比如RecyclerView，内部有ImageView，那么这些ImageView里面的图片可以适当裁剪，不影响视觉的情况下越小越好，对图片质量 要求不高的场景，就不再用ARGB\_8888加载了。

#### 局部变量与代码块

我们还要对局部变量进行优化，我们知道，局部变量在方法未调用完成，是不会被回收的，因为它是GCROOT，比如:

public void test(){

//1 创建一个大对象

BigObject big = getBitObject();

String name = big.xxx;

//big = null; //2

User user = new User();

user.name = name;

//...

//3 创建对象，此时内存不够了，需要GC，但是big不会被回收，因为当前方法还在运行。

TestObj obj = new TestObj();

}

我们知道，正在运行的方法的局部变量是GCRoot，JVM进行垃圾回收是不会回收的，除非它们被置为null，那么上面的big就不会被回收，就有可能抛出OOM异常，怎么办呢，我们可以直接将2处的注释打开，使得big被用完就置为null，等价于提前结束它的生命周期，最好的方法是代码块，比如:

public void test(){

User user = new User();

{

//1 创建一个大对象

BigObject big = getBitObject();

String name = big.xxx;

} //代码块结束，内部局部变量生命周期结束，可被回收。

user.name = name;

//...

//3 创建对象，此时内存不够了，需要GC，但是big不会被回收，因为当前方法还在运行。

TestObj obj = new TestObj();

}

加了代码块后，代码块内的变量的生命周期 在代码块结束后就自动结束，垃圾回收会直接回收。这等价于限制了对象的生命周期，我们应该合理使用代码块。

避免内存泄漏 我们先来看四种引用:

* 强引用: Object obj = new Object(); obj就是强引用。永远不会回收，即使OOM也不会回收
* 软引用: SoftReference，内存不够的时候会进行二次回收。
* 弱引用: WeakReference，碰到就回收。
* 虚引用: 只是用来通知一个对象是否被回收，和ReferenceQueue配合使用，如果在Queue里能找到，说明回收了。

我们来模拟一个流程:

* 1 User user = new User();创建一个对象，发现内存不够了
* 2 进行GC，碰到了obj1，发现它是强引用，跳过。
* 3 碰到了obj2，它是软引用，标记一下，继续向下。
* 4 碰到了obj3，直接回收。
* 5 对象扫描完了，内存够了，ok，创建user。
* 6 对象扫描完了，内存不够，回收刚刚标记的软引用，把软引用回收了。
* 7 回收后发现够了，ok，创建user。
* 8 不够，抛出OOM，此时obj1是强引用，没有回收，宁愿抛出OOM，也不回收强引用。

通过上面我们知道，强引用是绝不回收的；软引用是内存不够才二次回收，也就是说，内存不够的时候，先回收一次，回收后还不够，才回收，所以叫做二次回收；而弱引用，碰到就回收(即使内存足够)。

我们又知道，软引用和弱引用，在内存不足会回收(不管是一次回收还是二次回收，反正都会回收)，那么就不会造成内存泄漏，所以，只有强引用才会造成内存泄漏，而内存泄漏就是: 想回收却回收不了，也就是被别人占着，别人指的就是GCRoot，我们来看哪些可以作为GCRoot:

* 1 方法区的静态变量和常量
* 2 虚拟机栈正在引用的局部变量
* 3 活跃线程的成员变量

换言之，只要一个对象不被上述的变量引用，那么就不会内存泄漏。说白了，**不要让长生命周期的对象持有短生命周期的对象**(比如static的持有非static的，app一级的持有Activity一级的)，就能避免内存泄漏，如果真需要持有了，那么在短生命周期对象结束时，断开引用即可。

内存抖动

JVM垃圾回收的流程: 初始标记 -> 并发标记 -> 最终标记 -> 筛选回收。

这个过程需要停顿所有线程，当然很短的一瞬间，但是如果频繁触发垃圾回收，那么积少成多，集腋成裘，就会造成明显卡顿，这就叫内存抖动，那么什么情况下会造成内存抖动呢？答案就是: 短时间内创建大量的对象。比如:

for (int i = 0;i < 1000; i++) {

User user = new User();

//...

}

我们创建了1000次对象，在第100次时，内存不够，发生GC，在第200次时，内存不够发生GC...以此类推，发生10次GC，内存图看起来就是一上一下的山尖，叫做抖动。这样导致频繁回收，而回收又会停止所有线程，所以造成明显卡顿，这个要尽量避免。