**1、内存溢出**   
内存溢出：OOM（OutOfMemoryError）异常，即**程序需要内存超出了虚拟机可以分配内存的最大范围**。在Java 虚拟机规范的描述中，除了程序计数器外，虚拟机内存的其他区域都可能发生异常。

**2、内存溢出区域**   
常见的内存溢出分为以下几种：   
2.1 **Java堆溢出**  
Java 堆用于存储对象实例，只要不断地创建对象，并且**防止**垃圾回收机制清除这些对象，那么在**对象数量达到最大堆限制就会产生内存溢出异常。**

测试方案：无限循环new对象实例出来，在List中保存引用，防止GC回收，最终会产生OOM ,异常堆栈信息并提示**Java heap space**。

2.2 **虚拟机栈和本地方法栈溢出**关于**虚拟机栈和本地方法栈**，Java虚拟机规范中定义了两种异常：   
如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出**StackOverflowError** 异常（同一个栈堆满了内存）。   
如果虚拟机在**扩展栈时**无法申请到足够的内存空间，则抛出**OutOfMemoryError**异常。 （多个线程的栈不停的横向扩展空间）  
测试方案： 单线程条件下，通过不断递归调用方法，如不断累加的方法，如下所示：

public class JavaVMStackSOF{

private int stackLength=1;

public void stackLeak(){

stackLength++;//累加变量

stackLeak();//调用自身

}

}

最终会产生**StackOverflowError栈**溢出异常；

多线程条件下，无限循环地创建线程，并为每个线程无限循环的增加内存，最终会导致**OutOfMemoryError**异常。

这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制**Java堆和方法区这两部分内存的最大值**，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，**剩下的内存便给了虚拟机栈和本地方法栈**，栈中每个线程分配到的**栈容量越大**，可以建立的**线程数量自然就越少**。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过**减少最大堆和每个线程的栈容量来换取更多的线程**。

2.3 方法区和运行时常量池溢出   
运行时常量池是方法区的一部分。方法区用于存放Class的相关信息，如类名，访问修饰符，常量池，字段描述，方法描述等。测试方法：   
（1）对于非常量池部分，运行时**生成大量的动态类填满方法区**;   
（2）对于常量池部分，**无限循环调用String的intern()方法产生不同的String对象实例**，并在List中保存其引用，以防止被GC回收，最终会产生溢出。

2.4 本机直接内存溢出   
此类内存溢出一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果发现OOM之后Dump文件很小，而**程序中又直接或间接使用了NIO,**可以考虑一下是不是这方面原因。

**3、内存泄露**

内存泄漏是指**无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放**，从而造成内存空间的浪费称为内存泄漏。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你OOM。

Java内存泄漏的根本原因是**长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，**尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收。

3.1 内存泄露场景：具体主要有如下几大类：

3.1.1 静态集合类引起内存泄漏   
像**HashMap、Vector等的使用最容易出现内存泄露**，这些静态变量的生命周期和应用程序一致，他们所引用的**所有的对象Object也不能被释放**，因为他们也将一直被Vector等引用着。

Static Vector v = new Vector(10);

for (int i = 1; i<100; i++){

Object o = new Object();

v.add(o);

o = null;

}

在这个例子中，循环申请Object 对象，并将所申请的对象放入一个Vector 中，如果仅仅释放引用本身（o=null），**那么Vector 仍然引用该对象，所以这个对象对GC 来说是不可回收的**。因此，如果对象加入到Vector 后，还必须从Vector 中删除，最简单的方法就是将Vector对象设置为null。

3.1.3 监听器

在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

3.1.4 **各种连接**   
比如**数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接**，除非其显式的**调用了其close（）方法将其连接关闭**，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。

3.1.5 内部类和外部模块的引用   
内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如： public void registerMsg(Object b); 这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B是否提供相应的操作去除引用。

3.1.6 单例模式   
不正确使用**单例模式是引起内存泄漏的一个常见问题**，单例对象在初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被JVM正常回收，导致内存泄漏，考虑下面的例子：

class A{

public A(){

B.getInstance().setA(this);

}

....

}

//B类采用单例模式

class B{

private A a;

private static B instance=new B();

public B(){}

public static B getInstance(){

return instance;

}

public void setA(A a){

this.a=a;

}

//getter...

}

显然B采用singleton模式，它持有一个A对象的引用**，而这个A类的对象将不能被回收。**

Java内存泄漏就是没有及时清理内存垃圾，导致系统无法再给你提供内存资源（内存资源耗尽）；

而Java内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

[内存溢出](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E6%BA%A2%E5%87%BA&tn=67012150_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrjb0T1dBnH0znjcsnj-hPymzP1I-0ZwV5fKWUMw85HmLnjDznHRsgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT4PWD3rjnLnW6sPHf3Pj0Y)类似数组越届，超出你能存储的数据的上限

[内存泄漏](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F&tn=67012150_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrjb0T1dBnH0znjcsnj-hPymzP1I-0ZwV5fKWUMw85HmLnjDznHRsgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT4PWD3rjnLnW6sPHf3Pj0Y)，就是内存使用完毕后，不能释放回收重新使用

**Java内存泄露与溢出的区别**

**内存溢出就是你要求分配的内存超出了系统能给你的，**系统不能满足需求，于是产生溢出。

**Java内存泄漏就是没有及时清理内存垃圾**，导致系统无法再给你提供内存资源（内存资源耗尽）。

看到上面的解释，可能有些朋友还是不太理解吧。没问题，看以下例子

1.Java**内存泄露是说程序逻辑问题,**造成申请的内存无法释放.这样的话无论多少内存,早晚都会被占用光的.  
最简单的例子就是死循环了.由于程序判断错误导经常发生此事

2.Java内存泄漏是指在**堆上分配的内存没有被释放，从而失去对其控制**。这样会造成程序能使用的内存越来越少，导致系统运行速度减慢，严重情况会使程序当掉。

3.关于内存溢出有点出入。比如说你申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。  
举个现实中的例子：

比如有一个桶，装满了水.你丢个苹果进去。桶的水正常。如果你放个大石头。水就出溢出，内存溢出也就是这个原理。

**区别：内存溢出，提供的内存不够；Java内存泄漏，无法再提供内存资源**

可能大家会问内存泄露与溢出是考JAVA哪方面？考这个有什么用？

我个人觉的是考大家对JAVA是怎么管理内存这一块的知识?对下是对Java是如何管理内存的解释

**Java是如何管理内存**

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(GarbageCollection，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**对象开始检查引用链**，当遍历一遍后发现**没有被引用的孤立对象就作为垃圾回收**。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。

在JVM中如果98%的时间是用于GC且可用的Heap size 不足2%的时候将抛出此异常信息。

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；

JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4。

JVM内存的最大值跟操作系统有很大的关系。32位处理器虽然可控内存空间有4GB，但是具体的操作系统会给一个限制，这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，Linux系统下为2G-3G），而64bit以上的处理器就不会有限制了。

*注意：如果Xms超过了Xmx值，或者堆最大值和非堆最大值的总和超过了物理内存或者操作系统的最大限制都会引起服务器启动不起来。*

该错误常见场合：

a) Web上传文件时。

b) 开启大型文件或从数据库中一次取了太多的数据。

**相关问题**

**1. Q: Java中会存在内存泄漏吗？** A:  J**ava中也存在内存泄露。当被分配的对象可达但已无用**（未对作废数据内存单元的引用置null）即会引起。

         如：Java代码

1. Vector v=**new** Vector(10);
2. **for** (**int** i=1;i<100; i ) {
3. Object o=**new** Object();
4. v.add(o);
5. o=**null**;
6. }
7. **// 此时，所有的Object对象都没有被释放，因为变量v引用这些对象。**
8. **// 对象加入到Vector后，还必须从Vector中删除，最简单释放方法就是将Vector对象设置为null。**

**2. Q: 内存泄露、溢出的异同？**

     A: 同：**都会导致应用程序运行出现问题，性能下降或挂起。**

         异：

         1) 内存泄露是导致内存溢出的原因之一；**内存泄露积累起来将导致内存溢出**。

         2) **内存泄露可以通过完善代码来避免**；内存溢出可以通过**调整配置来减少发生频率，但无法彻底避免。**

**3. 如何检测内存泄露？**

 A: 可以通过一些性能监测分析工具，**如 JProfiler、Optimizeit Profiler。**

**4. Q: 如何避免内存泄露、溢出？** A: 1) 尽早释放无用对象的引用。

**好的办法是使用临时变量的时候，让引用变量在退出活动域后自动设置为null**，暗示垃圾收集器来收集该对象，防止发生内存泄露。

          2) **程序进行字符串处理时，尽量避免使用String，而应使用StringBuffer。**

          因为每一个String对象都会独立占用内存一块区域，如：

Java代码

String str = "aaa";String str2 = "bbb"; String str3 = str str2; // **假如执行此次之后str , str2再不被调用，那么它们就会在内存中等待GC回收； // 假如程序中存在过多的类似情况就会出现内存错误；**

         3) 尽量少用静态变量。

         因为静态变量是全局的，GC不会回收。

         4) 避免集中创建对象尤其是大对象，如果可以的话尽量使用流操作。

         JVM会突然需要大量内存，这时会触发GC优化系统内存环境；

5)**尽量运用对象池技术以提高系统性能。**

         生命周期长的对象拥有生命周期短的对象时容易引发内存泄漏，例如大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理，然后解决一块释放一块的策略。

         6) 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。

         7) 优化配置。

**5. Q: 内存溢出的解决方案？** A: 一是从代码层面进行优化完善，尽量避免该情况发生；

         二是调整优化服务器配置：

**1) 设置-Xms、-Xmx相等；**

**2) 设置NewSize、MaxNewSize相等；**

**3) 设置Heap size, PermGen space:**

            Tomcat 的配置示例：修改%TOMCAT\_HOME%/bin/catalina.bat or catalina.sh

            在“echo "Using CATALINA\_BASE: $CATALINA\_BASE"”上面加入以下行：

Cmd代码

1. set JAVA\_OPTS=-Xms800m -Xmx800m -XX:PermSize=128M -XX:MaxNewSize=256m -XX:MaxPermSize=256m

-----------------------------/////////////////////////////