线程隔离的数据区

由所有线程共享的数据区

本地方法库

本地库接口

执行引擎

本地方法栈 Native Method Stack

虚拟机栈 VM Stack

方法区 Method Area

**运行时数据区**

程序计数器 Program Counter Register

堆 Heap

程序计数器(Program Counter Register):是一块较小的内存区域，它可以看作时当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器(对于多核处理器来说一个内核)都只会执行一条线程中的指令，因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要一个独立的程序计数器，各条线程之间互不影响，独立存储，我们称这类内存区域为”线程私有”的内存。

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器值则为空。此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

Java虚拟机栈(Java Virtual Machine Stacks):也是线程私有的,它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧(Stack Frame)用于存储局部变量表，操作数栈，动态连接，方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型(boolean, byte, char, short, int, float, long double),对象引用(reference类型，它不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置)和returnAddress类型(指向了一条字节码指令的地址)

其中64位长度的long和double类型的数据会占有2个局部变量空间(Slot),其余的数据类型只占用1个，局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对这个区域规定了两种异常情况，如果线程请求的栈深度大于虚拟机的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展(当前大部分的Java虚拟机都可以动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈)，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

本地方法栈(Native Method Stack)与虚拟机栈非常相似，它们之间的区别不过是虚拟机栈执行Java方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言，使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

Java堆(Java Heap)是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范描述是:所有的对象实例以及数组都在堆上分配，但是随着JIT编译器(把Java字节码转换成可以直接发送给处理器的指令的程序)的发展与逃逸分析技术逐渐成熟，栈上分配，标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么”绝对”了。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称作“GC堆”(Garbage Collected Heap)，从内存回收的角度来看，由于现在收集器基本都采用分代收集算法。所以Java堆中还可以细分为：新生代和老年代：在细致一点的有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer, TLAB).不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，即可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的(通过-Xmx和-Xms控制)。如果在堆中没有内存完成实例分配，并且堆也无法在扩展时，将会抛出OutOfMemoryError异常。

方法区(Method Area)与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息，常量，静态常量，即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap(非堆)，目的应该是与Java堆区分开来。

对于习惯在HotSpot虚拟机上开发，部署程序的开发者来说，很多人都更愿意把方法去称为”永久代”(Permanent Generation),本质上两者并不等价，仅仅是因为HotSpot虚拟机的设计团队选择把GC分代收集扩展至方法区，或者说使用永久代来实现方法区而已，原则上，如何实现方法区属于虚拟机实现细节，不受虚拟机规范约束，但使用永久代来实现方法区，现在看来并不是一个好主意，因为这样更容易遇到内存溢出问题(永久代有-XX:MaxPermSize的上限)，在目前已经发布的JDK1.7的HotSpot中，已经把原来放在永久代的字符串常量池移出。

Java虚拟机规范对方法区限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说，这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是必要的。

当方法区无法满足内存非配需求时，将抛出OutOfMemoryError异常。

运行时常量池(Runtime Constant Pool)是方法区的一部分，Class文件中除了有类的版本，字段，方法，接口等描述信息外，还有一项信息是常量池(Constant Pool Table),用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

Java虚拟机对Class文件每一部分(自然*也包括常量池)的*格式都有严格规定，每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可，装载和执行，但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行区间也可能将新的常量池放入池中。这种特性被开发人员利用得比较多的便是String的intern()方法

在JDK6中，常量池位于PermGen(永久代)中，PermGen是一块主要用于存放已加载的类信息和字符串池的大小固定的区域。执行intern()方法时，若常量池中不存在等值的字符串，JVM就会在常量池中创建一个等值的字符串，然后返回该字符串的引用。

在JDK7中，常量池从PpermGen区域移到Java堆区，执行intern操作，如果常量池已经存在该字符串，则直接返回字符串的引用，否则复制该字符串对象的引用到常量池中并返回。

intern(）方法优点：执行速度非常快，直接使用==进行比较要比使用equals(）方法快很多；内存占用少。

当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

直接内存(Direct Memory)并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现。

在JDK 1.4中新加入了NIO(New Input/Output)类，引入了一种基于通道(Channel)与缓冲区(Buffer)的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存大小以及处理器寻址空间的限制。服务器管理员在配置虚拟机参数的时，会根据实际内存设置-Xmx等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现OutOfMemoryError异常。

对象的创建

虚拟机遇到一条new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载，解析和初始化过。如果没有，那必须先执行相应的类加载过程。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来，假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为指针碰撞(Bump the Ppointer)。如果Java堆中的内存并不是归整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞，

虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”。选择哪种分配方式由Java堆是否归整决定的，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的阵阵来分配内存的清空。解决方案又2种：一种是堆分配内存空间的动作进行同步处理——实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一下块内存，称为本地线程分配缓冲区(TLAB)。

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值，如果使用TLAB，这一工作过程也可以提前至TLAB分配进行。这一步操作保证了对象实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例，如何才能找到类的元数据信息，对象的哈希码，对象的GC分代年龄等信息。这些信息存放在对象的对象头之中。

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从Java程序的视角来看，对象创建才刚刚开始——<init>方法还没有执行，所有字段都还是零。所以，一般来说(由字节码中是否跟随invokespecial指令所决定)，执行new指令之后会接着执行<init>方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

对象的内存布局：在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头(Header)，实例数据(Instance Data)和对齐填充(Padding).

对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码，GC分代年龄，锁状态标志，线程持有的锁，偏向线程ID，偏向时间戳等，这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，官方称它为“Mark Word”。对象需要存储的运行时数据很多，其实已经超出了32位，64位Bitmap结构所能记录的限度，但是对象头信息是与对象自身定义的数据无关的额外存储成本。考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态服用自己的存储空间。如果对象处于未被锁定的状态下，那么Mark Word的32bit空间中的25bit用于存储对象的哈希码，4bit用于存储对象分代年龄，2bit用于存储锁标志位，1bit固定位0.



对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中却无法确定数组的大小。

接下来的实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段空间，无论是从父类继承下来的，还是子类中定义的，都需要记录起来。这部分的存储顺序会受到虚拟机分配策略参数和字段在Java源码中定义顺序的影响。HotSpot虚拟机默认的分配策略位longs/doubles, ints, shorts/chars, bytes/booleans, 从分配策略中可以看出，相同宽度的字段总是被分配到一起。在满足这个前提条件的情况下，在父类中定义的变量会出现在子类之前。如果CompactFields参数值为true，那么子类之中较窄的变量也可能会被插入到父类变量的空隙之中。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说，就是对象的大小必须是8字节的整数倍，而对象头部分正好是8字节的倍数，因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

对象的访问定位

建立对象是为了使用对象，我们的Java程序需要通过栈上的reference数据来操作堆上的具体对象。由于reference类型在Java虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用应该通过何种方式去定位，访问堆中的对象的具体位置，所以对象访问方式也是取决于虚拟机实现而定的。目前主流的访问方式有使用句柄和直接指针两种。

1）如果使用句柄访问的话，那么Java堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

|  |  |  |  |
| --- | --- | --- | --- |
| Java栈  实例池  Java堆   |  | | --- | | 句柄池 | | 到对象实例数据的指针 | | 到对象类型数据的指针 |   本地变量表 |
| int  对象实例数据 |
| short |
| reference |
| int  方法区 |
| double |
| float  对象类型数据 |

通过句柄访问对象

如果使用直接指针访问，那么Java堆对象的布局中就必须考虑如何防止访问类型数据的相关信息，而reference中存储的直接就是对象地址。

Java堆

|  |
| --- |
| Java栈  本地变量表  到对象类型数据的指针  对象实例数据 |
| int |
| short |
| reference |
| int  方法区 |
| double |
| float  对象类型数据 |

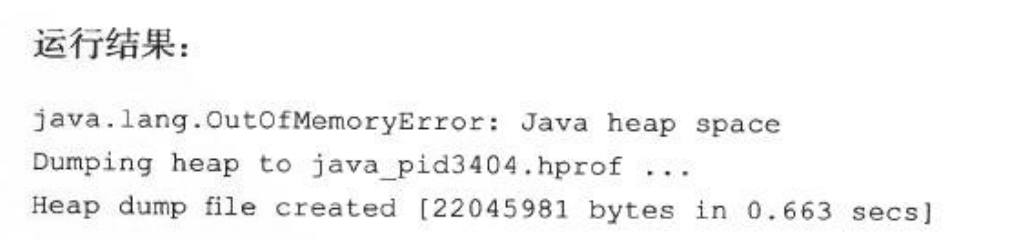
这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动(垃圾收集时移动对象是非常普遍的行为)时只会改变句柄中的数据指针，而reference本身不需要修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

实战：OutOfMemoryError异常

第一:通过代码验证Java虚拟机规范中描述的各个运行时区域存储的内容。

第二:希望读者在工作中遇到实际的内存溢出异常时，能根据异常的信息快速判断是哪个区域的内存溢出。



Java堆内存的OOM异常是实际应用中常见的内存溢出异常情况。当出现Java堆内存溢出时。异常堆栈信息”java.lang.OutOfMemoryError”会跟着进一步提示”Java heap space”。

Java堆溢出：Java堆用于存储对象实例，只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆的容量限制后就会产生内存溢出异常。

代码限制堆的大小为20MB，不可扩展(将堆的最小值-Xms参数与最大值-Xmx参数设置为一样即可避免堆自动扩展)，通过参数-XX:HeapDumpOnOutOfMemoeyError可以让虚拟机在出现内存溢出异常时Dump出当前内存堆转储快照以便事后进行分析。

要解决这个区域的异常，一般的手段是先通过内存映像分析工具(如Eclipse Memory Analyzer)对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏(Memory Leak)还是内存溢出(Memory Overflow).

内存泄漏:是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行数据减慢甚至系统崩溃等严重后果。

内存溢出:是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于能提供的最大内存。

虚拟机栈和本地方法栈溢出

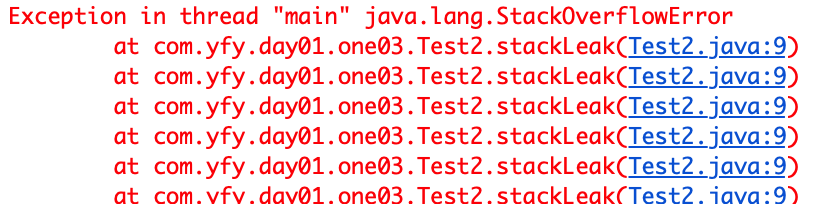
由于在HotSpot虚拟机中并不区分虚拟机栈和本地方法栈，因此，对于HotSpot来说，虽然-Xoss参数(设置本地方法栈大小)存在，但实际上是无效的，栈容量只由-Xss参数设定，关于虚拟机栈和本地方法栈，在Java虚拟机规范在描述了两种异常：

1）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。

2）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。

这里把异常分成两种情况，看似更加严谨，但却存在着一些互相重叠的地方；当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。

1）使用-Xss参数减少栈内存容量。结果:抛出StackOverflowError异常，异常出现时输出的堆栈深度相应缩小。

2）定义了大量的本地变量，增大此方法帧中本地变量表的长度。结果:抛出StackOverflowError异常时输出的堆栈深度相应的缩小。 

在单个线程下，无论是由栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverflowError异常。

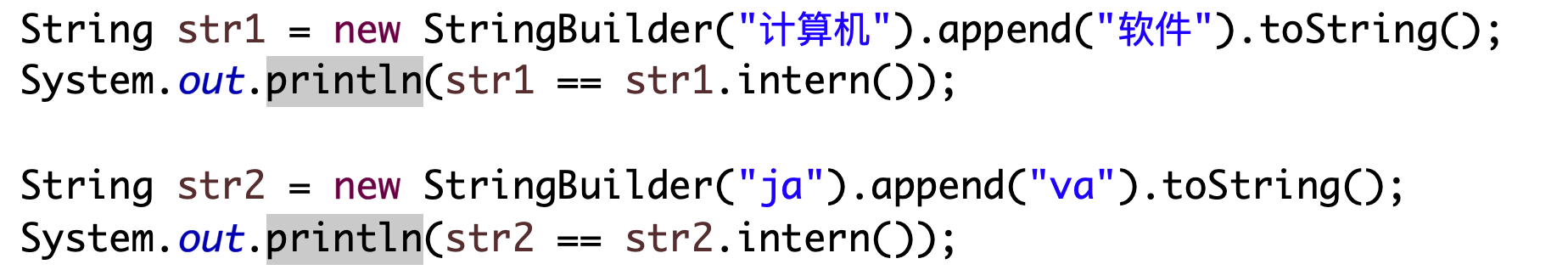
如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常，但是这样产生的内存溢出异常与栈空间是否足够大并不存在任何联系，或者准确地说，在这种情况下，为每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

其实原因不难理解，操作系统分配给每个进程的内存是有限制的，譬如32位的Window限制位2GB。虚拟机提供了参数来控制Java堆和方法区的这两部分内存的最大值。剩余的内存位2GB(操作系统限制)减去Xmx(最大堆容量)，在减去MaxPermSize(最大方法区容量)，程序计数器消耗内存很小，可以忽略掉。如果虚拟机进程本身耗费的内存不计算在内，剩下的内存就由虚拟机栈和本地方法栈瓜分了。每个线程分配到的栈容量越大，可以建立的线程数量就越少，建立线程时越容易把剩下的内存消耗尽。

但是，如果是建立过多线程导致的内存溢出，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。如果没有这方法的处理经验，这种通过“减少内存”的手段来解决内存溢出的方式会比较难以想到。

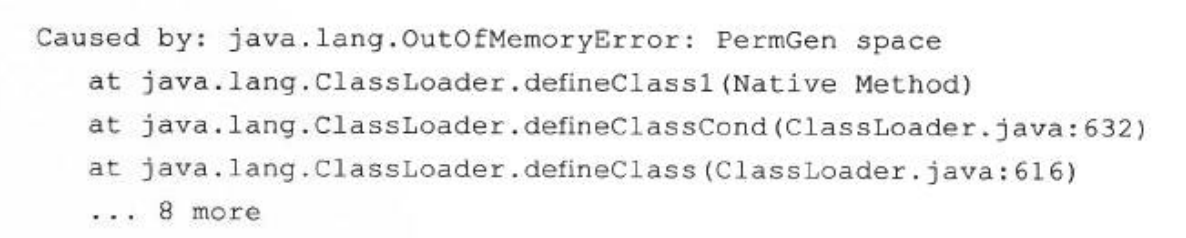
方法区和运行时常量池溢出

String.intern()是一个Native方法，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。在JDK 1.6及之前的版本中，由于常量池分配在永久代内，我们可以通过-XX:PermSize和-XX:MaxPermSize限制方法区大小，从而间接限制其中常量池的容量。



在JDK1.6中运行，会得到两个false，而在JDK1.7中运行会得到一个true和一个false。差生差异的原因是：在JDK1.6中，intern()方法会把首次遇到的字符串实例复制到永久代中，返回的也是永久代中这个字符串实例的引用，而由StringBuilder创建的字符串实例在java堆上，所以必然不是同一个引用。将返回false。而JDK 1.7的intern()方法不会在复制实现，只是在常量池中记录首次出现的实例引用，因此intern()返回的引用和由StringBuilder创建的那个字符串实例是同一个。对str2比较返回false是因为“Java”这个字符串在执行StringBuild.toString()之前已经出现过，字符串常量池中已经有它的引用了，不符合“首次出现”的原则，而“计算机软件”这个字符串则是首次出现的，因此返回true。

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。



方法区溢出也是一种常见的内存异常，一个类要被垃圾收集器回收掉，判定条件是比较苛刻的。在经常动态生成大量Class的应用中，需要特别注意类的回收情况。

本机直接内存溢出

DirectMemory(直接内存)容量可通过-XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆最大值(-Xmx指定)一样。由于DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果读者发现OOM之后Dump文件很小，而程序中又直接或间接使用NIO，那就可以考虑检查一下是不是这方法的原因。

第三章 垃圾收集器与内存分配策略

思考GC需要完成的3件事情：

1）哪些内存需要回收？

2）什么时候回收？

3）如何回收？

当需要排查各种内存溢出，内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶劲时，我们就需要对这些“自动化”的技术实施必要的监控和调节。其中程序计数器，虚拟机栈，本地方法栈3个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知。因为这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟着回收了。而Java堆和方法区则不一样，一个接口中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存。

判断对象是否存活：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器的值就加1；当引用失效时，计数器的值就减1；任何时刻计数器为0的对象就是不可能再被使用的。但是，至少主流的Java虚拟机里面没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题。

可达性分析算法：来判断对象是否存活的。这个算法的基本思路就是通过一些列的称为“GC Roots”的对象作为起始点。从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象时不可用的。

object 5

GC Roots

object 1

object 7

object 6

ogj3d5

object 3

object 2

判定可回收对象

object 4

仍然存活的对象

在Java语言中，可作为GC Roots的对象包括下面几种：

1）虚拟机栈(栈帧中的本地变量表)中引用的对象。

2）方法区中类静态属性引用的对象。

3）方法区中常量引用的对象。

4）本地方法栈中JNI引用的对象。

在谈引用:在JDK1.2之后，Java堆引用的概念进行了扩充，将引用分为强引用(Strong Reference),软引用(Soft Reference),弱引用(Weak Reference),虚引用(Phantom Reference)4种，这4种引用强度一次逐渐减弱。

1）强引用就是指在程序代码中普遍存在的，类似”Object obj = new Object()”,这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉引用的对象。

1. 软引用时用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK1.2之后，提供了SoftReference类来实现软引用。
2. 弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK1.2之后，提供了WeakReference类来实现弱引用。
3. 虚引用也被称为幽灵引用或者幻影引用。它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚拟引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。

生存还是死亡：即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓行”，阶段，要真正宣告一个对象死亡，至少要经历两次标记过程；如果对象在进行可达性分析后发现没有与GC Roots相链接的引用链，那它将会被第一次标记并且进行一次筛选，刷选的条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法。或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会放置在一个叫做F-Queue的队列中，并在稍后由一个由虚拟机自动建立的，低优先级的Finalizer线程区执行它。这里所谓的执行是指虚拟机会触发这个方法，但并不承若会等待它运行结束，这样做的原因是，如果一个对象在finalize()方法中执行缓慢，或者发生了死循环，将很会导致F-Queue队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己赋值给某个类变量或者对象成员变量，那在第二次标记时它将被移出“即将回收”的集合；如果对象这时候没有逃脱，那基本上它就真的被回收了。

这是因为任何一个对象finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的finalize()方法不会被再次执行，因此第二段代码的自救行动失败了。

回收方法区：Java虚拟机规范确实说过可以不要求虚拟机在方法区实现垃圾收集，而且在方法区中进行垃圾收集的“性价比”一般比较低：在堆中，尤其是在新生代中，常规应用进行一次垃圾收集一般可以回收70%-95%的空间，而永久代的垃圾收集效率远低于此。

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收Java堆中的对象非常类似，以常量池中字面量的回收为例，加入一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个String对象叫做“abc”的，换句话说，就是没有任何String对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果这时发生内存回收，而且必要的话，这个“abc”常量就会被系统清理出常量池。常量池中的其他类，方法，字段的符号引用也与此类似。

什么是无用类

1）该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。

2）加载该类的ClassLoader已经被回收。

3）该类对象应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。



