JAVA笔记

1. **JAVA虚拟机**
   1. **Java内存区域及对象**

**1.1.1 几个计算机的概念**

1、计算机存储单位

从小到大依次为位Bit、字节Byte、千字节KB、兆M、千兆GB、TB，相邻单位之间都是1024倍，1024为2的10次方，即

1Byte = 8bit，1K = 1024Byte，1M = 1024K，1G = 1024M，1T = 1024G

2、计算机存储元件

寄存器：中央处理器CPU的一部分，是计算机中读写速度最快的存储元件，但是容量很少

内存：属于独立的一个部件，是和CPU沟通的桥梁，用于存放CPU中的运算数据以及与外部存储器交换的数据。尽管在今天，对内存的读写速度已经很快了，但是由于寄存器是在CPU上的，所以对于内存的读写速度和对于寄存器的读写速度上还是有几个数量级的差距。但是没办法，对于内存的读写I/O操作是很难消除的，寄存器数量有限，不可能通过寄存器来完成所有的运算任务

3、内核空间和用户空间

连接内存和寄存器的是地址总线，地址总线的宽度影响了物理地址的索引范围，因为总线宽度决定了处理器一次可以从寄存器或内存中获取多少个Bit，同时也决定了处理器最大可以寻址的地址空间。比如32位CPU的系统，可寻址范围为0x00000000~0xFFFFFFFF，即232=4294967296个内存位置，每个内存位置1个字节，即32位CPU系统可以有4GB的内存空间。不过应用程序是不可以完全使用这些地址空间的，因为这些地址空间被划分为了内核空间和用户空间，程序只能使用用户空间的内存。内核空间主要是指操作系统运行时所使用的用于程序调度、虚拟内存的使用或者链接硬件资源的程序逻辑。区分内核空间和用户空间的目的主要是从系统的稳定性的角度考虑的。Windows 32操作系统默认内核空间和用户空间的比例是1:1，即2G内核空间、2G内存空间，32位Linux系统中默认比例则是1:3，即1G内核空间，3G内存空间。

4、字长

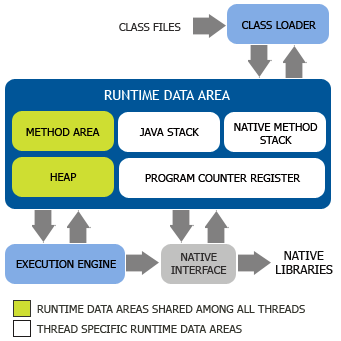
CPU的主要技术指标之一，指的是CPU一次能并行处理二进制的位数（Bit）。通常称处理字长为8位数据的CPU为8位CPU，32位CPU就是在同一时间内处理字长为32位的二进制数据。不过目前虽然CPU大多是64位的，但还是以32位字长运行

**1.1.2 前言**

说到Java内存区域，可能很多人第一反应是“堆栈”。首先堆栈不是一个概念，而是两个概念，堆和栈是两块不同的内存区域，简单理解的话，堆是用来存放对象而栈是用来执行程序的。其次，堆内存和栈内存的这种划分方式比较粗糙，这种划分方式只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块，Java内存区域的划分实际上远比这复杂。对于Java程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去配对delete/free代码，不容易出现内存泄露和内存溢出问题。但是，也正是因为Java把内存控制权交给了虚拟机，一旦出现内存泄露和内存溢出的问题，就难以排查，因此一个好的Java程序员应该去了解虚拟机的内存区域以及会引起内存泄露和内存溢出的场景。

运行时数据区域

Java虚拟机（JVM）内部定义了程序在运行时需要使用到的内存区域，从http://images.blogjava.net/blogjava\_net/nkjava/jvmstructure.png拷贝一张图下来



之所以要划分这么多区域出来是因为这些区域都有自己的用途，以及创建和销毁的时间。有些区域随着虚拟机进程的启动而存在，有的区域则依赖用户线程的启动和结束而销毁和建立。图中绿色部分就是所有线程之间共享的内存区域，而白色部分则是线程运行时独有的数据区域，从这个分类角度来看一下这几个数据区。

1、线程独有的内存区域

（1）PROGRAM COUNTER REGISTER，程序计数器

这块内存区域很小，它是当前线程所执行的字节码的行号指示器，字节码解释器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。Java方法这个计数器才有值，如果执行的是一个Native方法，那这个计数器是空的。

（2）JAVA STACK，虚拟机栈

生命周期和线程相同。每个方法执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，每一个方法从调用直至执行完毕的过程，就对应着一个栈帧在虚拟机中入栈到出栈的过程。栈的大小和具体JVM的实现有关，通常在256K~756K之间。

（3）NATIVE METHOD STACK，方法栈

和虚拟机栈起的作用一样，只不过方法栈为虚拟机使用到的Native方法服务。虚拟机规范并没有对这个区域有什么强制规定，因此我们使用的HotSpot虚拟机，就干脆没有这块区域了，它和虚拟机栈是一起的。

2、线程间共享的内存区域

（1）HEAP，堆

大多数应用，堆都是Java虚拟机所管理的内存中最大的一块，它在虚拟机启动时创建，此内存唯一的目的就是存放对象实例。由于现在垃圾收集器采用的基本都是分代收集算法，所以堆还可以细分为新生代和老年代，再细致一点还有Eden区、From Survivior区、To Survivor区，这个后面都会讲到的。

（2）METHOD AREA，方法区

这块区域用于存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据，虚拟机规范是把这块区域描述为堆的一个逻辑部分的，但实际它应该是要和堆区分开的。从上面提到的分代收集算法的角度看，HotSpot中，方法区≈永久代。不过JDK 7之后，我们使用的HotSpot应该就没有永久代这个概念了，会采用Native Memory来实现方法区的规划了。

（3）RUNTIME CONSTANT POOL，运行时常量池

上面的图中没有画出来，因为它是方法区的一部分。Class文件中除了有类的版本信息、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期间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，另外翻译出来的直接引用也会存储在这个区域中。这个区域另外一个特点就是动态性，Java并不要求常量就一定要在编译期间才能产生，运行期间也可以在这个区域放入新的内容，String.intern()方法就是这个特性的应用。

3、直接内存

想想还是把这块加上。直接内存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，而且也可能导致内存溢出问题。JDK1.4中新增加了NIO，引入了一种基于通道与缓冲区的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（包括RAM、SWAP区）大小以及处理器寻址空间的限制。

对象创建

Java是一门面向对象的语言，Java程序运行过程中无时无刻都有对象被创建出来。在语言层面上，创建对象（克隆、反序列化）就是一个new关键字而已，但是虚拟机层面上却不是如此。看一下在虚拟机层面上创建对象的步骤：

1、虚拟机遇到一条new指令，首先去检查这个指令的参数能否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经被加载、解析和初始化。如果没有，那么必须先执行类的初始化过程。

2、类加载检查通过后，虚拟机为新生对象分配内存。对象所需内存大小在类加载完成后便可以完全确定，为对象分配空间无非就是从Java堆中划分出一块确定大小的内存而已。这个地方会有两个问题：

（1）如果内存是规整的，那么虚拟机将采用的是指针碰撞法来为对象分配内存。意思是所有用过的内存在一边，空闲的内存在另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是Serial、ParNew这种基于压缩算法的，虚拟机采用这种分配方式。

（2）如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表法来为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的，再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，并更新列表上的内容。如果垃圾收集器选择的是CMS这种基于标记-清除算法的，虚拟机采用这种分配方式。

另外一个问题及时保证new对象时候的线程安全性。因为可能出现虚拟机正在给对象A分配内存，指针还没有来得及修改，对象B又同时使用了原来的指针来分配内存的情况。虚拟机采用了CAS配上失败重试的方式保证更新更新操作的原子性和TLAB两种方式来解决这个问题。

3、内存分配结束，虚拟机将分配到的内存空间都初始化为零值（不包括对象头）。这一步保证了对象的实例字段在Java代码中可以不用赋初始值就可以直接使用，程序能访问到这些字段的数据类型所对应的零值。

4、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中。

5、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。

对象定位方式

建立对象是为了使用对象，Java程序需要通过栈上的reference（引用）数据来操作堆上的具体对象。比如我们写了一句

Object obj = new Object()

而new Object()之后其实有两部分内容，一部分是类数据（比如代表类的Class对象）、一部分是实例数据

由于reference在Java虚拟机规范中只是一个指向对象new Object()的引用obj，并没有规定obj应该通过何种方式去定位、访问堆中对象的具体位置，所以对象访问方式也是取决于虚拟机而定的。主流方式有两种：

1、句柄访问。Java堆中划分出一块句柄池，obj指向的是对象的句柄地址，句柄中则包含了类数据的地址和实例数据的地址

2、指针访问。obj指向一个对象(该对象包含所有的实例数据和类数据的地址)

HotSpot虚拟机采用的是后者，不过前者的对象访问方式也是十分常见的。

* 1. **常用JVM命令参数**

（1）-Xms20M

表示设置堆容量的最小值为20M，必须以M为单位

（2）-Xmx20M

表示设置堆容量的最大值为20M，必须以M为单位。将-Xmx和-Xms设置为一样可以避免堆自动扩展。大的项目-Xmx和-Xms一般都要设置到10G、20G甚至还要高

（3）-verbose:gc

表示输出虚拟机中GC的详细情况

（4）-Xss128k

表示可以设置虚拟机栈的大小为128k

（5）-Xoss128k

表示设置本地方法栈的大小为128k。不过HotSpot并不区分虚拟机栈和本地方法栈，因此对于HotSpot来说这个参数是无效的

（6）-XX:PermSize=10M

表示JVM初始分配的永久代的容量，必须以M为单位

（7）-XX:MaxPermSize=10M

表示JVM允许分配的永久代的最大容量，必须以M为单位，大部分情况下这个参数默认为64M

（8）-Xnoclassgc

表示关闭JVM对类的垃圾回收

（9）-XX:+TraceClassLoading

表示查看类的加载信息

（10）-XX:+TraceClassUnLoading

表示查看类的卸载信息

（11）-XX:NewRatio=4

表示设置年轻代：老年代的大小比值为1：4，这意味着年轻代占整个堆的1/5

（12）-XX:SurvivorRatio=8

表示设置2个Survivor区：1个Eden区的大小比值为2:8，这意味着Survivor区占整个年轻代的1/5，这个参数默认为8

（13）-Xmn20M

表示设置年轻代的大小为20M

（14）-XX:+HeapDumpOnOutOfMemoryError

表示可以让虚拟机在出现内存溢出异常时Dump出当前的堆内存转储快照

（15）-XX:+UseG1GC

表示让JVM使用G1垃圾收集器

（16）-XX:+PrintGCDetails

表示在控制台上打印出GC具体细节

（17）-XX:+PrintGC

表示在控制台上打印出GC信息

（18）-XX:PretenureSizeThreshold=3145728

表示对象大于3145728（3M）时直接进入老年代分配，这里只能以字节作为单位

（19）-XX:MaxTenuringThreshold=1

表示对象年龄大于1，自动进入老年代

（20）-XX:CompileThreshold=1000

表示一个方法被调用1000次之后，会被认为是热点代码，并触发即时编译

（21）-XX:+PrintHeapAtGC

表示可以看到每次GC前后堆内存布局

（22）-XX:+PrintTLAB

表示可以看到TLAB的使用情况

（23）-XX:+UseSpining

开启自旋锁

（24）-XX:PreBlockSpin

更改自旋锁的自旋次数，使用这个参数必须先开启自旋锁

* 1. **内存溢出**

**1.3.1堆溢出**

Java堆唯一的作用就是存储对象实例，只要保证不断创建对象并且对象不被回收，那么对象数量达到最大堆容量限制后就会产生内存溢出异常了。所以测试的时候把堆的大小固定住并且让堆不可扩展即可。测试代码如下:

package com.xrq.test;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 测试内容：堆溢出

\*

\* 虚拟机参数：-Xms20M -Xmx20M -XX:+HeapDumpOnOutOfMemoryError

\*/

public class HeapOverflowTest

{

public static void main(String[] args)

{

List<HeapOverflowTest> list = new ArrayList<HeapOverflowTest>();

while (true)

{

list.add(new HeapOverflowTest());

}

}

}

这种异常很常见，也很好发现，因为都提示了“Java heap space”了，定位问题的话，根据异常堆栈分析就好了，行号都有指示。解决方案的话，可以调大堆的大小或者从代码上检视是否存在某些对象生命周期过长、持有状态时间过长的情况，长时间少程序运行期间的内存消耗。

**1.3.2栈溢出**

Java虚拟机规范中描述了如果线程请求的栈深度太深（换句话说方法调用的深度太深），就会产生栈溢出了。那么，我们只要写一个无限调用自己的方法，自然就会出现方法调用的深度太深的场景了。测试代码如下:

package com.xrq.test;

/\*\*

\* 测试内容：栈溢出测试（递归调用导致栈深度不断增加）

\*

\* 虚拟机参数：-Xss128k

\*/

public class StackOverflowTest

{

private int stackLength = 1;

public void stackLeak()

{

stackLength++;

stackLeak();

}

public static void main(String[] args) throws Throwable

{

StackOverflowTest stackOverflow = new StackOverflowTest();

try

{

stackOverflow.stackLeak();

}

catch (Throwable e)

{

System.out.println("stack length:" + stackOverflow.stackLength);

throw e;

}

}

}

后面都是一样的，忽略。通过不断创建线程的方式可以产生OutOfMemoryError，因为每个线程都有自己的栈空间。不过这个操作有危险就不做了，原因是Windows平台下，Java的线程是直接映射到操作系统的内核线程上的，如果写个死循环无限产生线程，那么可能会造成操作系统的假死。

上面无限产生线程的场景，从另外一个角度说，就是为每个线程的栈分配的内存空间越大，反而越容易产生内存溢出。其实这也很好理解，操作系统分配给进程的内存是有限制的，比如32位的Windows限制为2GB。虚拟机提供了了参数来控制Java堆和方法区这两部分内存的最大值，剩余内存为2GB-最大堆容量-最大方法区容量，程序计数器很小就忽略了，虚拟机进程本身的耗费也不算，剩下的内存就是栈的了。每个线程分配到的栈容量越大，可建立的线程数自然就越少，建立线程时就越容易把剩下的内存耗尽。

StackOverFlowError这个异常，有错误堆栈可以阅读，比较好定位。而且如果使用虚拟机默认参数，栈深度在大多数情况下，达到1000~2000完全没有问题，正常方法的调用这个深度应该是完全够了。但是如果建立过多线程导致的OutOfMemoryError，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减小最大堆容量和减小栈容量来换取更多的线程了。

**1.3.3方法区和运行时常量池溢出**

运行时常量池也是方法区的一部分，所以这两个区域一起看就可以了。这个区域的OutOfMemoryError可以利用String.intern()方法来产生。这是一个Native方法，意思是如果常量池中有一个String对象的字符串就返回池中的这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中去，并且返回此String对象的引用。测试代码如下:

package com.xrq.test;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 测试内容：常量池溢出（这个例子也可以说明运行时常量池为方法区的一部分）

\*

\* 虚拟机参数-XX:PermSize=10M -XX:MaxPermSize=10M

\*/

public class ConstantPoolOverflowTest

{

public static void main(String[] args)

{

List<String> list = new ArrayList<String>();

int i = 0;

while (true)

{

list.add(String.valueOf(i++).intern());

}

}

}

之前有讲过，对于HotSpot而言，方法区=永久代，这里看到OutOfMemoryError的区域是“PermGen space”，即永久代，那其实也就是方法区溢出了。注意一下JDK1.7下是不会有这个异常的，while循环将一直下去，因为JDK1.7之后溢出了永久代并采用Native Memory来实现方法区的规划了。

* 1. **Java垃圾回收（GC）机制详解**

哪些内存需要回收？

哪些内存需要回收是垃圾回收机制第一个要考虑的问题，所谓“要回收的垃圾”无非就是那些不可能再被任何途径使用的对象。那么如何找到这些对象？

1、引用计数法

这个算法的实现是，给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数值为0的对象就是不可能再被使用的。这种算法使用场景很多，但是，Java中却没有使用这种算法，因为这种算法很难解决对象之间相互引用的情况。

2、可达性分析法

这个算法的基本思想是通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链（即GC Roots到对象不可达）时，则证明此对象是不可用的。在Java语言中可以作为GC Roots的对象包括：

· 虚拟机栈中引用的对象

· 方法区中静态属性引用的对象

· 方法区中常量引用的对象

· 本地方法栈中JNI（即Native方法）引用的对象

**4种引用状态**

在JDK1.2之前，Java中引用的定义很传统：如果引用类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过于狭隘，一个对象只有被引用或者没被引用两种状态。我们希望描述这样一类对象：当内存空间还足够时，则能保留在内存中；如果内存空间在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用4种，这4种引用强度依次减弱。

1、强引用

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被引用的对象。

2、软引用

描述有些还有用但并非必需的对象。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进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用

3、弱引用

描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾收集器工作之后，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。Java中的类WeakReference表示弱引用

4、虚引用

这个引用存在的唯一目的就是在这个对象被收集器回收时收到一个系统通知，被虚引用关联的对象，和其生存时间完全没关系。Java中的类PhantomReference表示虚引用

**方法区回收**

虚拟机规范中不要求方法区一定要实现垃圾回收，而且方法区中进行垃圾回收的效率也确实比较低，但是HotSpot对方法区也是进行回收的，主要回收的是废弃常量和无用的类两部分。判断一个常量是否“废弃常量”比较简单，只要当前系统中没有任何一处引用该常量就好了，但是要判定一个类是否“无用的类”条件就要苛刻很多，类需要同时满足以下三个条件：

1、该类所有实例都已经被回收，也就是说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

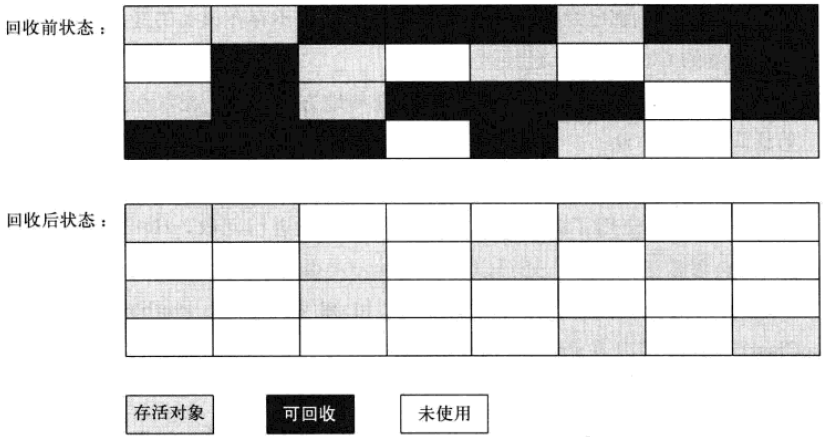
在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载功能，以保证方法区不会溢出。

**垃圾回收算法**

第一步考量了哪些对象进行回收后，第二步自然是如何对对象进行回收了。这里主要写几种垃圾回收算法的思想。

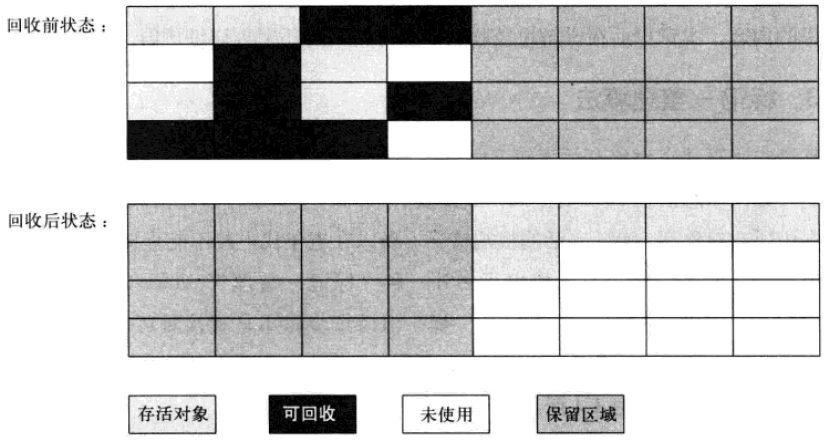
1、标记-清除（Mark-Sweep）算法

这是最基础的算法，标记-清除算法就如同它的名字样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。这种算法的不足主要体现在效率和空间，从效率的角度讲，标记和清除两个过程的效率都不高；从空间的角度讲，标记清除后会产生大量不连续的内存碎片， 内存碎片太多可能会导致以后程序运行过程中在需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。



2、复制（Copying）算法

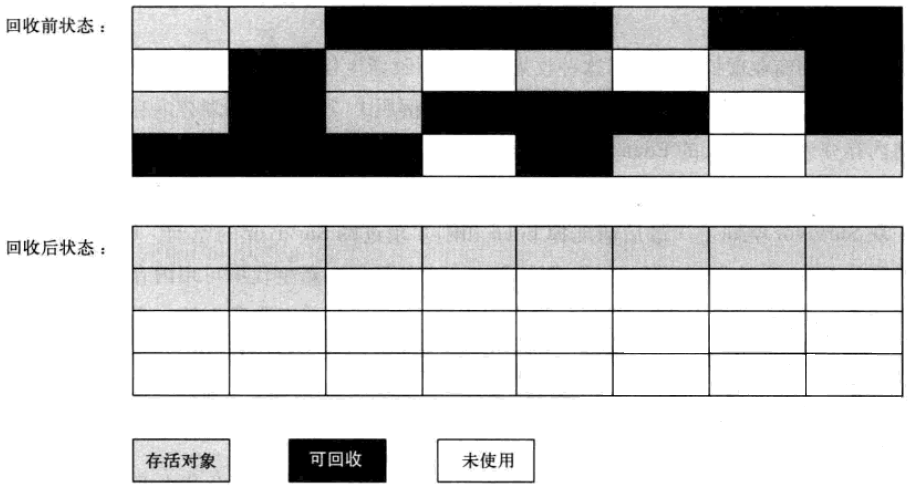
复制算法是为了解决效率问题而出现的，它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。复制算法的执行过程如图：



不过这种算法有个缺点，内存缩小为了原来的一半，这样代价太高了。现在的商用虚拟机都采用这种算法来回收新生代，不过研究表明1:1的比例非常不科学，因此新生代的内存被划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。每次回收时，将Eden和Survivor中还存活着的对象一次性复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden区和Survivor区的比例为8:1，意思是每次新生代中可用内存空间为整个新生代容量的90%。当然，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖老年代进行分配担保（Handle Promotion）。

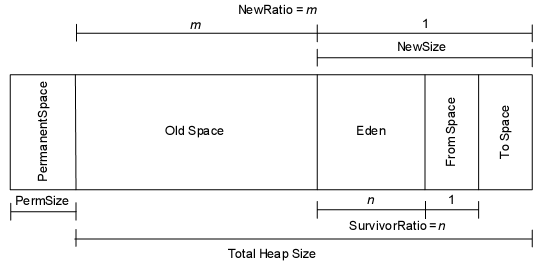
3、标记-整理（Mark-Compact）算法

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%存活，那么需要有额外的空间进行分配担保。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，有人提出了另外一种标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。标记-整理算法的工作过程如图：



**分代收集**

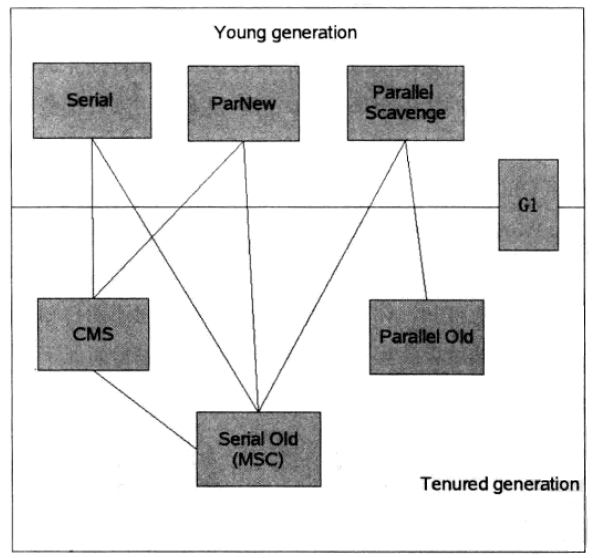
根据上面的内容，用一张图概括一下堆内存的布局



现代商用虚拟机基本都采用分代收集算法来进行垃圾回收。这种算法没什么特别的，无非是上面内容的结合罢了，根据对象的生命周期的不同将内存划分为几块，然后根据各块的特点采用最适当的收集算法。大批对象死去、少量对象存活的，使用复制算法，复制成本低；对象存活率高、没有额外空间进行分配担保的，采用标记-清理算法或者标记-整理算法。

**垃圾收集器**

垃圾收集器就是上面讲的理论知识的具体实现了。不同虚拟机所提供的垃圾收集器可能会有很大差别，我们使用的是HotSpot，HotSpot这个虚拟机所包含的所有收集器如图：



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，那说明它们可以搭配使用。虚拟机所处的区域说明它是属于新生代收集器还是老年代收集器。多说一句，我们必须姚明带一个道理：没有最好的垃圾收集器，更加没有万能的收集器，只能选择对具体应用最合适的收集器。这也是HotSpot为什么要实现这么多收集器的原因。OK，下面一个一个看一下收集器：

1、Serial收集器

最基本、发展历史最久的收集器，这个收集器是一个采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止。后者意味着，在用户不可见的情况下要把用户正常工作的线程全部停掉，这对很多应用是难以接受的。不过实际上到目前为止，Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器，因为它简单而高效。用户桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代停顿时间在几十毫秒最多一百毫秒，只要不是频繁发生，这点停顿是完全可以接受的。

2、ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集外，其余行为和Serial收集器完全一样，包括使用的也是复制算法。ParNew收集器除了多线程以外和Serial收集器并没有太多创新的地方，但是它却是Server模式下的虚拟机首选的新生代收集器，其中有一个很重要的和性能无关的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作（看图）。CMS收集器是一款几乎可以认为有划时代意义的垃圾收集器，因为它第一次实现了让垃圾收集线程与用户线程基本上同时工作。ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于线程交互的开销，该收集器在两个CPU的环境中都不能百分之百保证可以超越Serial收集器。当然，随着可用CPU数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU数量相同，在CPU数量非常多的情况下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

3、Parallel收集器

Parallel收集器也是一个新生代收集器，也是用复制算法的收集器，也是并行的多线程收集器，但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍吞吐量的概念。CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel收集器的目标则是达到一个可控制的吞吐量。所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总运行100分钟，垃圾收集1分钟，那吞吐量就是99%。另外，Parallel收集器是虚拟机运行在Server模式下的默认垃圾收集器。

停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；高吞吐量则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。

虚拟机提供了-XX:MaxGCPauseMillis和-XX:GCTimeRatio两个参数来精确控制最大垃圾收集停顿时间和吞吐量大小。不过不要以为前者越小越好，GC停顿时间的缩短是以牺牲吞吐量和新生代空间换取的。由于与吞吐量关系密切，Parallel收集器也被称为“吞吐量优先收集器”。Parallel收集器有一个-XX:+UseAdaptiveSizePolicy参数，这是一个开关参数，这个参数打开之后，就不需要手动指定新生代大小、Eden区和Survivor参数等细节参数了，虚拟机会根据当前系统的运行情况手机性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。如果对于垃圾收集器运作原理不太了解，以至于在优化比较困难的时候，使用Parallel收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。

4、 Serial Old收集器

Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理算法”，这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

5、 Parallel Old收集器

Parallel收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器在JDK 1.6之后的出现，“吞吐量优先收集器”终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel收集器+Parallel Old收集器的组合。

6、CMS收集器

CMS收集器是一种以获取最短回收停顿时间为目标的老年代收集器。目前很大一部分Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其注重服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验，CMS收集器就非常符合这类应用的需求。CMS收集器从名字就能看出是基于“标记-清除”算法实现的。

7、G1收集器

G1（Garbage-First）收集器是当今收集器技术发展的最前沿成果之一，JDK 7 Update 4后开始进入商用。在G1收集器之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1收集器不再是这样，使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region的集合。G1收集器跟踪各个Region里面的垃圾堆积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也是Garbage-First名称的由来）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

**垃圾收集器总结**

来看一下对垃圾收集器的总结，列了一张表：



**GC日志**

每种收集器的日志形式都是由它们自身的实现所决定的，换言之，每种收集器的日志格式都可以不一样。不过虚拟机为了方便用户阅读，将各个收集器的日志都维持了一定的共性，就以最前面的对象间相互引用的那个类ReferenceCountingGC的代码为例：

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseSerialGC”，使用Serial+Serial Old组合进行垃圾回收的日志。**

[GC [DefNew: 310K->194K(2368K), 0.0269163 secs] 310K->194K(7680K), 0.0269513 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]

[GC [DefNew: 2242K->0K(2368K), 0.0018814 secs] 2242K->2241K(7680K), 0.0019172 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System) [Tenured: 2241K->193K(5312K), 0.0056517 secs] 4289K->193K(7680K), [Perm : 2950K->2950K(21248K)], 0.0057094 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

Heap

def new generation total 2432K, used 43K [0x00000000052a0000, 0x0000000005540000, 0x0000000006ea0000)

eden space 2176K, 2% used [0x00000000052a0000, 0x00000000052aaeb8, 0x00000000054c0000)

from space 256K, 0% used [0x00000000054c0000, 0x00000000054c0000, 0x0000000005500000)

to space 256K, 0% used [0x0000000005500000, 0x0000000005500000, 0x0000000005540000)

tenured generation total 5312K, used 193K [0x0000000006ea0000, 0x00000000073d0000, 0x000000000a6a0000)

the space 5312K, 3% used [0x0000000006ea0000, 0x0000000006ed0730, 0x0000000006ed0800, 0x00000000073d0000)

compacting perm gen total 21248K, used 2982K [0x000000000a6a0000, 0x000000000bb60000, 0x000000000faa0000)

the space 21248K, 14% used [0x000000000a6a0000, 0x000000000a989980, 0x000000000a989a00, 0x000000000bb60000)

No shared spaces configured.

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseParNewGC”，使用ParNew+Serial Old组合进行垃圾回收的日志**

[GC [ParNew: 310K->205K(2368K), 0.0006664 secs] 310K->205K(7680K), 0.0007043 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [ParNew: 2253K->31K(2368K), 0.0032525 secs] 2253K->2295K(7680K), 0.0032911 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System) [Tenured: 2264K->194K(5312K), 0.0054415 secs] 4343K->194K(7680K), [Perm : 2950K->2950K(21248K)], 0.0055105 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

Heap

par new generation total 2432K, used 43K [0x0000000005550000, 0x00000000057f0000, 0x0000000007150000)

eden space 2176K, 2% used [0x0000000005550000, 0x000000000555aeb8, 0x0000000005770000)

from space 256K, 0% used [0x0000000005770000, 0x0000000005770000, 0x00000000057b0000)

to space 256K, 0% used [0x00000000057b0000, 0x00000000057b0000, 0x00000000057f0000)

tenured generation total 5312K, used 194K [0x0000000007150000, 0x0000000007680000, 0x000000000a950000)

the space 5312K, 3% used [0x0000000007150000, 0x0000000007180940, 0x0000000007180a00, 0x0000000007680000)

compacting perm gen total 21248K, used 2982K [0x000000000a950000, 0x000000000be10000, 0x000000000fd50000)

the space 21248K, 14% used [0x000000000a950000, 0x000000000ac39980, 0x000000000ac39a00, 0x000000000be10000)

No shared spaces configured.

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseParallelGC”，使用Parallel+Serial Old组合进行垃圾回收的日志**

**[GC [PSYoungGen: 4417K->288K(18688K)] 4417K->288K(61440K), 0.0007910 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]**

**[Full GC (System) [PSYoungGen: 288K->0K(18688K)] [PSOldGen: 0K->194K(42752K)] 288K->194K(61440K) [PSPermGen: 2941K->2941K(21248K)], 0.0032663 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]**

**Heap**

**PSYoungGen total 18688K, used 321K [0x0000000034190000, 0x0000000035660000, 0x0000000048f90000)**

**eden space 16064K, 2% used [0x0000000034190000,0x00000000341e05c0,0x0000000035140000)**

**from space 2624K, 0% used [0x0000000035140000,0x0000000035140000,0x00000000353d0000)**

**to space 2624K, 0% used [0x00000000353d0000,0x00000000353d0000,0x0000000035660000)**

**PSOldGen total 42752K, used 194K [0x000000000a590000, 0x000000000cf50000, 0x0000000034190000)**

**object space 42752K, 0% used [0x000000000a590000,0x000000000a5c0810,0x000000000cf50000)**

**PSPermGen total 21248K, used 2982K [0x0000000005190000, 0x0000000006650000, 0x000000000a590000)**

**object space 21248K, 14% used [0x0000000005190000,0x0000000005479980,0x0000000006650000)**

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC”，使用ParNew+CMS+Serial Old组合进行垃圾回收的日志**

[Full GC (System) [CMS: 0K->194K(62656K), 0.0080796 secs] 4436K->194K(81792K), [CMS Perm : 2941K->2940K(21248K)], 0.0081589 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

Heap

par new generation total 19136K, used 340K [0x0000000005540000, 0x0000000006a00000, 0x0000000006a00000)

eden space 17024K, 2% used [0x0000000005540000, 0x0000000005595290, 0x00000000065e0000)

from space 2112K, 0% used [0x00000000065e0000, 0x00000000065e0000, 0x00000000067f0000)

to space 2112K, 0% used [0x00000000067f0000, 0x00000000067f0000, 0x0000000006a00000)

concurrent mark-sweep generation total 62656K, used 194K [0x0000000006a00000, 0x000000000a730000, 0x000000000a940000)

concurrent-mark-sweep perm gen total 21248K, used 2981K [0x000000000a940000, 0x000000000be00000, 0x000000000fd40000)

这四段GC日志中提炼出一些共性：

1、日志的开头“GC”、“Full GC”表示这次垃圾收集的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有Full，则说明本次GC停止了其他所有工作线程。看到Full GC的写法是“Full GC(System)”，这说明是调用System.gc()方法所触发的GC。

2、“GC”中接下来的“DefNew”、“ParNew”、“PSYoungGen”、“CMS”表示的是新生代垃圾收集器的名称，“PSYoungGen”中的“PS”指的是“Parallel Scavenge”，它是Parallel收集器的全称。

3、以第一个为例，方括号内部的“320K->194K(2368K)”、“2242K->0K(2368K)”，指的是该区域已使用的容量->GC后该内存区域已使用的容量(该内存区总容量)。方括号外面的“310K->194K(7680K)”、“2242K->2241K(7680K)”则指的是GC前Java堆已使用的容量->GC后Java堆已使用的容量(Java堆总容量)。

4、还以第一个为例，再往后“0.0269163 secs”表示该内存区域GC所占用的时间，单位是秒。最后的“[Times: user=0.00 sys=0.00 real=0.03 secs]”则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的钟墙时间。后面两个的区别是，钟墙时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间所以如果user或sys超过real是完全正常的。

5、“Heap”后面就列举出堆内存目前各个年代的区域的内存情况

**触发GC的时机**

最后总结一下什么时候会触发一次GC，个人经验看，有三种场景会触发GC：

1、第一种场景应该很明显，当年轻代或者老年代满了，Java虚拟机无法再为新的对象分配内存空间了，那么Java虚拟机就会触发一次GC去回收掉那些已经不会再被使用到的对象

2、手动调用System.gc()方法，通常这样会触发一次的Full GC以及至少一次的Minor GC

3、程序运行的时候有一条低优先级的GC线程，它是一条守护线程，当这条线程处于运行状态的时候，自然就触发了一次GC了。这点也很好证明，不过要用到WeakReference的知识，后面写WeakReference的时候会专门讲到这个。

**内存溢出和内存泄露的区别**

1、内存溢出

内存溢出指的是程序在申请内存的时候，没有足够大的空间可以分配了。

2、内存泄露

内存泄露指的是程序在申请内存之后，没有办法释放掉已经申请到内存，它始终占用着内存，即被分配的对象可达但无用。内存泄露一般都是因为内存中有一块很大的对象，但是无法释放。

从定义上可以看出，内存泄露终将导致内存溢出。

注意，定位虚拟机问题内存问题的时候第一步就是要判断到底是内存溢出还是内存泄露，前者好判断，跟踪堆栈信息就可以了；后者比较复杂一点，一般都是老年代中的大对象没释放掉，要通过各种办法找出老年代中的大对象没有被释放的原因。

**并行和并发的区别**

这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，可以这么理解这两个名词：

1、并行Parallel

多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态

2、并发Concurrent

指用户线程与垃圾收集线程同时执行（但并不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个CPU上

**Minor GC和Full GC的区别**

1、新生代GC（Minor GC）

指发生在新生代的垃圾收集动作，因为大多数Java对象存活率都不高，所以Minor GC非常频繁，一般回收速度也比较快

2、老年代GC（Major GC/Full GC）

指发生在老年代的垃圾收集动作，出现了Major GC，经常会伴随至少一次的Minor GC（但并不是绝对的）。Major GC的速度一般要比Minor GC慢上10倍以上

**Client模式和Server模式的区别**

部分商用虚拟机中，Java程序最初是通过解释器对.class文件进行解释执行的，当虚拟机发现某个方法或代码块运行地特别频繁的时候，就会把这些代码认定为热点代码Hot Spot Code（这也是我们使用的虚拟机HotSpot名称的由来）。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器叫做即时编译器（Just In Time Compiler，即JIT编译器）。JIT编译器并不是虚拟机必需的部分，Java虚拟机规范并没有要求要有JIT编译器的存在，更没有限定或指导JIT编译器应该如何去实现。但是，JIT编译器性能的好坏、代码优化程度的高低却是衡量一款商用虚拟机优秀与否的最关键指标之一。

解释器和编译器其实和编译器各有优势：

1、当程序需要迅速启动和执行的时候，解释器可以先发挥作用，省去编译的时间，立即执行

2、在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率，我们使用的HotSpot中内置了两个JIT编译器，即C1编译器和C2编译器，默认采用的是解释器和一个编辑器配合的方式进行工作。HotSpot在启动的时候会根据自身版本以及宿主机器的硬件性能自动选择运行模式，比如会检测宿主机器是否为服务器、比如J2SE会检测主机是否有至少2个CPU和至少2GB的内存。

1、如果是，则虚拟机会以Server模式运行，该模式与C2编译器共同运行，更注重编译的质量，启动速度慢，但是运行效率高，适合用在服务器环境下，针对生产环境进行了优化

2、如果不是，则虚拟机会以Client模式运行，该模式与C1编译器共同运行，更注重编译的速度，启动速度快，更适合用在客户端的版本下，针对GUI进行了优化

**有两种方法查看虚拟机是运行在Client模式下还是Server模式下：**

1、在程序命令行运行“java -version”命令，查看的是你本地安装的虚拟机是信息

http://images2015.cnblogs.com/blog/801753/201509/801753-20150926103903178-913980972.png

2、比如我们用Eclipse或者MyEclipse运行程序，一般使用的都是工具自带的JRE，虚拟机并不是本地安装的虚拟机。这时候怎么办呢，可以通过在程序中运行下面的语句来查看虚拟机信息

System.out.println(System.getProperty("java.vm.name"));

我这里的运行结果是

Java HotSpot(TM) 64-Bit Server VM

当然要改变虚拟机运行的模式也可以，只需要改jvm.cfg就可以了。我们可以从以下几个地方找到jvm.cfg：

1、32位的JDK的文件路径是  JAVA\_HOME/jre/lib/i386/jvm.cfg

2、64位的JDK的文件路径是  JAVA\_HOME/jre/lib/amd64/jvm.cfg

3、MyEclipse在 .../Common/binary/com.sun.java.jdk.win32.x86\_64\_1.6.0.013/jre/lib/amd64/jvm.cfg

目前64位只支持Server模式，文件内容都是一样的，上面的注释不去管它，剩下的就是这些：

-server KNOWN

-client IGNORE

-hotspot ALIASED\_TO -server

-classic WARN

-native ERROR

-green ERROR

由于我的电脑装的是64位JDK，所以是“-client INGORE”。同时支持Server模式和Client模式的，应该是“-server KNOWN”和“-client KNOWN”，一般只需要变更这两个配置的先后顺序即可，但是前提是JAVA\_HOME/jre/bin目录下同时存在server和client两个文件夹，分别对应着各自的虚拟机，缺少一个，切换后就会报错。

* 1. **内存分配原则**

**1.5.1前言**

对象的内存分配，往大的方向上讲，就是在堆上分配，少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节决定于当前使用的是哪种垃圾收集器组合，当然还有虚拟机中与内存相关的参数。垃圾收集器组合一般就是Serial+Serial Old和Parallel+Serial Old，前者是Client模式下的默认垃圾收集器组合，后者是Server模式下的默认垃圾收集器组合，文章使用对比学习法对比Client模式下和Server模式下同一条对象分配原则有什么区别。

**1.5.2 TLAB**

首先讲讲什么是TLAB。内存分配的动作，可以按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。哪个线程需要分配内存，就在哪个线程的TLAB上分配。虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。这么做的目的之一，也是为了并发创建一个对象时，保证创建对象的线程安全性。TLAB比较小，直接在TLAB上分配内存的方式称为快速分配方式，而TLAB大小不够，导致内存被分配在Eden区的内存分配方式称为慢速分配方式。

**对象优先分配在Eden区上**

上面讲了不同的垃圾收集器组合对于内存分配规则是有影响的，看下影响在什么地方并解释一下原因，虚拟机参数为“-verbose:gc -XX:+PrintGCDetails -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8”，即10M新生代，10M老年代，10M新生代中8M的Eden区，两个Survivor区各1M。代码都是同一段：

public class EdenAllocationTest

{

private static final int \_1MB = 1024 \* 1024;

public static void main(String[] args)

{

byte[] allocation1 = new byte[2 \* \_1MB];

byte[] allocation2 = new byte[2 \* \_1MB];

byte[] allocation3 = new byte[2 \* \_1MB];

byte[] allocation4 = new byte[4 \* \_1MB];

}

}

Client模式下

[复制代码](javascript:void(0);)

[GC [DefNew: 6487K->194K(9216K), 0.0042856 secs] 6487K->6338K(19456K), 0.0043281 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

Heap

def new generation total 9216K, used 4454K [0x0000000005180000, 0x0000000005b80000, 0x0000000005b80000)

eden space 8192K, 52% used [0x0000000005180000, 0x00000000055a9018, 0x0000000005980000)

from space 1024K, 18% used [0x0000000005a80000, 0x0000000005ab0810, 0x0000000005b80000)

to space 1024K, 0% used [0x0000000005980000, 0x0000000005980000, 0x0000000005a80000)

tenured generation total 10240K, used 6144K [0x0000000005b80000, 0x0000000006580000, 0x0000000006580000)

the space 10240K, 60% used [0x0000000005b80000, 0x0000000006180048, 0x0000000006180200, 0x0000000006580000)

compacting perm gen total 21248K, used 2982K [0x0000000006580000, 0x0000000007a40000, 0x000000000b980000)

the space 21248K, 14% used [0x0000000006580000, 0x0000000006869890, 0x0000000006869a00, 0x0000000007a40000)

No shared spaces configured.

[复制代码](javascript:void(0);)

Server模式下

[复制代码](javascript:void(0);)

Heap

PSYoungGen total 9216K, used 6651K [0x000000000af20000, 0x000000000b920000, 0x000000000b920000)

eden space 8192K, 81% used [0x000000000af20000,0x000000000b59ef70,0x000000000b720000)

from space 1024K, 0% used [0x000000000b820000,0x000000000b820000,0x000000000b920000)

to space 1024K, 0% used [0x000000000b720000,0x000000000b720000,0x000000000b820000)

PSOldGen total 10240K, used 4096K [0x000000000a520000, 0x000000000af20000, 0x000000000af20000)

object space 10240K, 40% used [0x000000000a520000,0x000000000a920018,0x000000000af20000)

PSPermGen total 21248K, used 2972K [0x0000000005120000, 0x00000000065e0000, 0x000000000a520000)

object space 21248K, 13% used [0x0000000005120000,0x0000000005407388,0x00000000065e0000)

[复制代码](javascript:void(0);)

看到在Client模式下，最后分配的4M在新生代中，先分配的6M在老年代中；在Server模式下，最后分配的4M在老年代中，先分配的6M在新生代中。说明不同的垃圾收集器组合对于对象的分配是有影响的。讲下两者差别的原因：

1、Client模式下，新生代分配了6M，虚拟机在GC前有6487K，比6M也就是6144K多，多主要是因为TLAB和EdenAllocationTest这个对象占的空间，TLAB可以通过“-XX:+PrintTLAB”这个虚拟机参数来查看大小。OK，6M多了，然后来了一个4M的，Eden+一个Survivor总共就9M不够分配了，这时候就会触发一次Minor GC。但是触发Minor GC也没用，因为allocation1、allocation2、allocation3三个引用还存在，另一块1M的Survivor也不够放下这6M，那么这次Minor GC的效果其实是通过分配担保机制将这6M的内容转入老年代中。然后再来一个4M的，由于此时Minor GC之后新生代只剩下了194K了，够分配了，所以4M顺利进入新生代。

2、Server模式下，前面都一样，但是在GC的时候有一点区别。在GC前还会进行一次判断，如果要分配的内存>=Eden区大小的一半，那么会直接把要分配的内存放入老年代中。要分配4M，Eden区8M，刚好一半，而且老年代10M，够分配，所以4M就直接进入老年代去了。为了验证一下结论，我们把3个2M之后分配的4M改为3M看一下

[复制代码](javascript:void(0);)

public class EdenAllocationTest

{

private static final int \_1MB = 1024 \* 1024;

public static void main(String[] args)

{

byte[] allocation1 = new byte[2 \* \_1MB];

byte[] allocation2 = new byte[2 \* \_1MB];

byte[] allocation3 = new byte[2 \* \_1MB];

byte[] allocation4 = new byte[3 \* \_1MB];

}

}

[复制代码](javascript:void(0);)

运行结果为

[复制代码](javascript:void(0);)

[GC [PSYoungGen: 6487K->352K(9216K)] 6487K->6496K(19456K), 0.0035661 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC [PSYoungGen: 352K->0K(9216K)] [PSOldGen: 6144K->6338K(10240K)] 6496K->6338K(19456K) [PSPermGen: 2941K->2941K(21248K)], 0.0035258 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]

Heap

PSYoungGen total 9216K, used 3236K [0x000000000af40000, 0x000000000b940000, 0x000000000b940000)

eden space 8192K, 39% used [0x000000000af40000,0x000000000b269018,0x000000000b740000)

from space 1024K, 0% used [0x000000000b740000,0x000000000b740000,0x000000000b840000)

to space 1024K, 0% used [0x000000000b840000,0x000000000b840000,0x000000000b940000)

PSOldGen total 10240K, used 6338K [0x000000000a540000, 0x000000000af40000, 0x000000000af40000)

object space 10240K, 61% used [0x000000000a540000,0x000000000ab70858,0x000000000af40000)

PSPermGen total 21248K, used 2982K [0x0000000005140000, 0x0000000006600000, 0x000000000a540000)

object space 21248K, 14% used [0x0000000005140000,0x0000000005429890,0x0000000006600000)

[复制代码](javascript:void(0);)

看到3M在新生代中，6M通过分配担保机制进入老年代了。

**大对象直接进入老年代**

虚拟机参数为“-XX:+PrintGCDetails -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:PretenureSizeThreshold=3145728”，最后那个参数表示大于这个设置值的对象直接在老年代中分配，这样做的目的是为了避免在Eden区和两个Survivor区之间发生大量的内存复制。测试代码为：

[复制代码](javascript:void(0);)

public class OldTest

{

private static final int \_1MB = 1024 \* 1024;

public static void main(String[] args)

{

byte[] allocation = new byte[4 \* \_1MB];

}

}

[复制代码](javascript:void(0);)

**Client模式下**

[复制代码](javascript:void(0);)

Heap

def new generation total 9216K, used 507K [0x0000000005140000, 0x0000000005b40000, 0x0000000005b40000)

eden space 8192K, 6% used [0x0000000005140000, 0x00000000051bef28, 0x0000000005940000)

from space 1024K, 0% used [0x0000000005940000, 0x0000000005940000, 0x0000000005a40000)

to space 1024K, 0% used [0x0000000005a40000, 0x0000000005a40000, 0x0000000005b40000)

tenured generation total 10240K, used 4096K [0x0000000005b40000, 0x0000000006540000, 0x0000000006540000)

the space 10240K, 40% used [0x0000000005b40000, 0x0000000005f40018, 0x0000000005f40200, 0x0000000006540000)

compacting perm gen total 21248K, used 2972K [0x0000000006540000, 0x0000000007a00000, 0x000000000b940000)

the space 21248K, 13% used [0x0000000006540000, 0x00000000068272a0, 0x0000000006827400, 0x0000000007a00000)

No shared spaces configured.

[复制代码](javascript:void(0);)

**Server模式下**

[复制代码](javascript:void(0);)

Heap

PSYoungGen total 9216K, used 4603K [0x000000000afc0000, 0x000000000b9c0000, 0x000000000b9c0000)

eden space 8192K, 56% used [0x000000000afc0000,0x000000000b43ef40,0x000000000b7c0000)

from space 1024K, 0% used [0x000000000b8c0000,0x000000000b8c0000,0x000000000b9c0000)

to space 1024K, 0% used [0x000000000b7c0000,0x000000000b7c0000,0x000000000b8c0000)

PSOldGen total 10240K, used 0K [0x000000000a5c0000, 0x000000000afc0000, 0x000000000afc0000)

object space 10240K, 0% used [0x000000000a5c0000,0x000000000a5c0000,0x000000000afc0000)

PSPermGen total 21248K, used 2972K [0x00000000051c0000, 0x0000000006680000, 0x000000000a5c0000)

object space 21248K, 13% used [0x00000000051c0000,0x00000000054a72a0,0x0000000006680000)

[复制代码](javascript:void(0);)

看到**Client模式下4M直接进入了老年代，Server模式下4M还在新生代中**。产生这个差别的原因是**“-XX:PretenureSizeThreshold”这个参数对Serial+Serial Old垃圾收集器组合有效而对Parallel+Serial Old垃圾收集器组合无效**。

**其他几条原则**

上面列举的原则其实不重要，只是演示罢了，也不需要记住，因为实际过程中我们可能使用的并不是上面的垃圾收集器的组合，可能使用ParNew垃圾收集器，可能使用G1垃圾收集器。场景很多，重要的是要在实际使用的时候有办法知道使用的垃圾收集器对于对象分配有哪些原则，因为理解这些原则才是调优的第一步。下面列举一下对象分配的另外两条原则：

1、长期存活的对象将进入老年代。Eden区中的对象在一次Minor GC后没有被回收，则对象年龄+1，当对象年龄达到“-XX:MaxTenuringThreshold”设置的值的时候，对象就会被晋升到老年代中

2、Survivor空间中相同年龄的所有对象大小总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到“-XX:MaxTenuringThreshold”设置要求的年龄

* 1. **虚拟机性能监控与故障处理工具**

**1.6.1前言**

定位系统问题的时候，知识、经验是基础，数据是依据，工具是运用知识处理数据的手段。这里说的数据包括：运行日志、异常堆栈、GC日志、线程快照、堆转储快照等。经常使用适当的虚拟机监控和分析的工具可以加快分析数据、定位解决问题的速度。

**jps：虚拟机进程状况工具**

首先约定一下运行的代码都是以下这段：

[复制代码](javascript:void(0);)

1 public class TestMain

2 {

3 public static void main(String[] args)

4 {

5 while (true)

6 {

7

8 }

9 }

10 }

[复制代码](javascript:void(0);)

JDK的很多小工具的名字都参考了UNIX命令的命名方式，jps（JVM Process Status）是其中的典型。除了名字像UNIX的ps命令外，它的功能也和ps命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID（Local Virtual Machine Identifier,LVMID）。虽然功能比较单一，但它是使用最高的JDK命令行工具，因为其他的JDK工具大多需要输入它查询到的LVMID来确定要监控的是哪一个虚拟机进程。

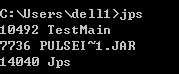
**jps命令格式：**

jps [ options ] [ hostid ]

**jps工具主要选项**

|  |  |
| --- | --- |
| **选    项** | **作            用** |
| -q | 只输出LVMID，省略主类的名称 |
| -m | 输出虚拟机进程启动时传递给主类main()函数的参数 |
| -l | 输出主类的全名，如果进程执行的是jar包，输出jar包路径 |
| -v | 输出虚拟机进程启动时的JVM参数 |

**jps执行样例**



某个虚拟机进程执行TestMain这个类的main方法，看到10492就是该虚拟机进程的ID

**jstat：虚拟机统计信息监控工具**

jstat（JVM Statistics Monitoring Tool）使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程（需要远程主机提供RMI支持）虚拟机进程中的类信息、内存、垃圾收集、JIT编译等运行数据，在没有GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。

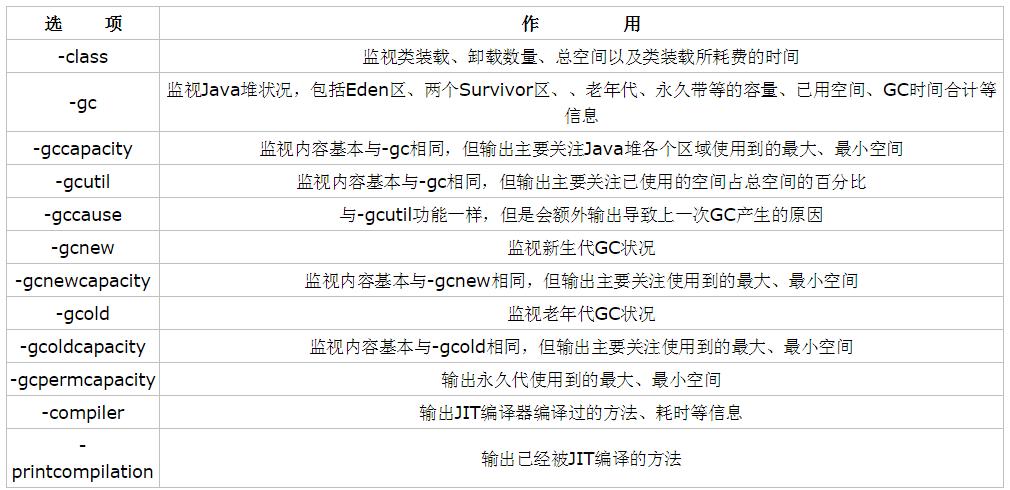
**jstat命令格式**

jstat [ option vmid [ interval [ s | ms ] [ count ] ] ]

这个VMID，对于本地虚拟机进程而言，VMID和LVMID是一致的。参数interval和count分别表示查询间隔和次数，如果省略这两个参数，说明只查询一次，假设需要每250毫秒查询一次进程2764的垃圾收集情况，一共查询20次，那命令应当是：

**jstat -gc 2764 250 20**

**jstat主要工具选项**



**jstat执行样例**

jstat监视选项众多，举一个例子来查看一下该命令如何查看监视结果

http://images2015.cnblogs.com/blog/801753/201509/801753-20150927175643069-377631724.png

查询结果表明，新生代Eden区（E，表示Eden）使用了2%的空间，两个Survivor区（S0、S1，表示Survivor0、Survivor1）都是空的，老年代（O，表示Old）和永久带（P。表示Permanent）则分别使用了0%和13.84%的空间。程序运行以来共发生Minor GC（YGC，表示Young GC）0次，总共耗时0秒；发生Full GC（FGC，表示Full GC）3次，Full GC共耗时（FGCT，Full GC Time）为0秒，所有GC总耗时（GCT，表示GC Time）0秒。

**jinfo：Java配置信息工具**

jinfo（Configuration Info for Java）的作用是实时地查看和调整虚拟机各项参数。使用jps命令的-v可以查看虚拟机启动时显式指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，可以使用jinfo的-flag选项进行查询，jinfo还可以使用-sysprops选项把虚拟机进程的System.getProperties()的内容打印出来。

用法是jinfo -opt pid 如：查看2788的MaxPerm大小可以用 jinfo -flag MaxPermSize 2788。

**jmap：Java内存映像工具**

jmap（Memory Map for Java）命令用于生成堆转储快照。如果不使用jmap命令，要想获取Java堆转储，可以使用“-XX:+HeapDumpOnOutOfMemoryError”参数，可以让虚拟机在OOM异常出现之后自动生成dump文件，Linux命令下可以通过kill -3发送进程退出信号也能拿到dump文件。

jmap的作用并不仅仅是为了获取dump文件，它还可以查询finalize执行队列、Java堆和永久代的详细信息，如空间使用率、当前使用的是哪种收集器等。和jinfo一样，jmap有不少功能在Windows平台下也是受限制的，除了生成dump文件的-dump选项和用于查看每个类的实例、空间占用统计的-histo选项在所有操作系统都提供之外，其余选项都只能在Linux和Solaris系统下使用。

**jmap命令格式**

jmap [ option ] vmid

**jmap工具主要选项**



**jstack：Java堆栈跟踪工具**

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的主要是定位线程长时间出现停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

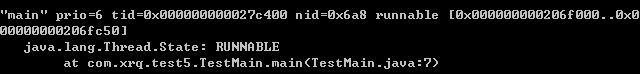
**jstack命令格式**

jstack [ option ] vmid

**jstack主要工具选项**

|  |  |
| --- | --- |
| **选    项** | **作             用** |
| -F | 当正常输出的请求不被响应时，强制输出线程堆栈 |
| -l | 除堆栈外，显示关于锁的附加信息 |
| -m | 如果调用到本地方法的时候，可以显示C/C++的堆栈 |

**jstack执行样例**

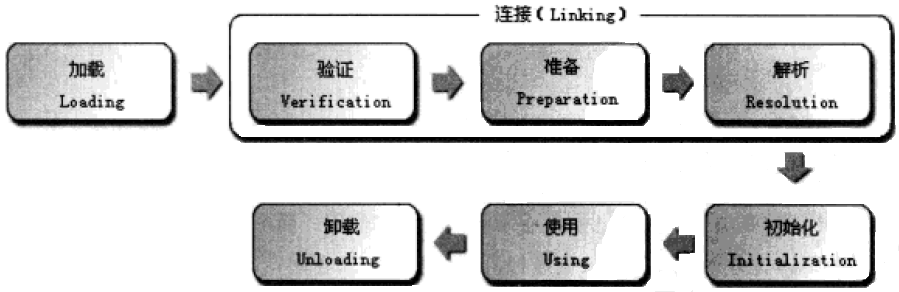


只截取一部分，看到这就是10492这个虚拟机进程ID当前时刻的线程快照。线程处于RUNNABLE状态，执行到了TestMain函数的第7行，并且一直停留在第7行。

* 1. **Java类加载机制**

**类使用的7个阶段**

类从被加载到虚拟机内存中开始，到卸载出内存，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initiallization）、使用（Using）和卸载（Unloading）这7个阶段。其中验证、准备、解析3个部分统称为连接（Linking），这七个阶段的发生顺序如下图：



图中，加载、验证、准备、初始化、卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段不一定：它在某些情况下可以初始化阶段之后在开始，这是为了支持Java语言的运行时绑定（也称为动态绑定）。接下来讲解加载、验证、准备、解析、初始化五个步骤，这五个步骤组成了一个完整的类加载过程。使用没什么好说的，卸载属于GC的工作，在之前GC的文章中已经有所提及了。

**加载Loading**

加载是类加载的第一个阶段。有两种时机会触发类加载：

1、预加载。虚拟机启动时加载，加载的是JAVA\_HOME/lib/下的rt.jar下的.class文件，这个jar包里面的内容是程序运行时非常常常用到的，像**java.lang.\*、java.util.\*、java.io.\***等等，因此随着虚拟机一起加载。要证明这一点很简单，写一个空的main函数，设置虚拟机参数为"-XX:+TraceClassLoading"来获取类加载信息，运行一下：

[复制代码](javascript:void(0);)

1 [Opened E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

2 [Loaded java.lang.Object from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

3 [Loaded java.io.Serializable from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

4 [Loaded java.lang.Comparable from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

5 [Loaded java.lang.CharSequence from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

6 [Loaded java.lang.String from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

7 [Loaded java.lang.reflect.GenericDeclaration from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

8 [Loaded java.lang.reflect.Type from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

9 [Loaded java.lang.reflect.AnnotatedElement from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

10 [Loaded java.lang.Class from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

11 [Loaded java.lang.Cloneable from E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\rt.jar]

12 ...

[复制代码](javascript:void(0);)

2、运行时加载。虚拟机在用到一个.class文件的时候，会先去内存中查看一下这个.class文件有没有被加载，如果没有就会按照类的全限定名来加载这个类。

那么，加载阶段做了什么，其实加载阶段做了有三件事情：

1、**获取.class文件的二进制流**

2、**将类信息、静态变量、字节码、常量这些.class文件中的内容放入方法区中**

3、**在内存中生成一个代表这个.class文件的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。一般这个Class是在堆里的，不过HotSpot虚拟机比较特殊，这个Class对象是放在方法区中的**

虚拟机规范对这三点的要求并不具体，因此虚拟机实现与具体应用的灵活度都是相当大的。例如第一条，根本没有指明二进制字节流要从哪里来、怎么来，因此单单就这一条，就能变出许多花样来：

· 从zip包中获取，这就是以后jar、ear、war格式的基础

· 从网络中获取，典型应用就是Applet

· 运行时计算生成，典型应用就是动态代理技术

· 由其他文件生成，典型应用就是JSP，即由JSP生成对应的.class文件

· 从数据库中读取，这种场景比较少见

总而言之，在类加载整个过程中，这部分是对于开发者来说可控性最强的一个阶段。

**验证：**

连接阶段的第一步，**这一阶段的目的是为了确保.class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全**。

Java语言本身是相对安全的语言（相对C/C++来说），但是前面说过，.class文件未必要从Java源码编译而来，可以使用任何途径产生，甚至包括用十六进制编辑器直接编写来产生.class文件。在字节码语言层面上，Java代码至少从语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。

验证阶段将做一下几个工作，具体就不细讲了，这是虚拟机实现层面的问题：

1、文件格式验证

这个地方要说一点和开发者相关的。.class文件的第5~第8个字节表示的是该.class文件的主次版本号，验证的时候会对这4个字节做一个验证，**高版本的JDK能向下兼容以前版本的.class文件，但不能运行以后的class文件**，即使文件格式未发生任何变化，虚拟机也必须拒绝执行超过其版本号的.class文件。举个具体的例子，如果一段.java代码是在JDK1.6下编译的，那么JDK1.6、JDK1.7的环境能运行这个.java代码生成的.class文件，但是JDK1.5、JDK1.4乃更低的JDK版本是无法运行这个.java代码生成的.class文件的。如果运行，会抛出java.lang.UnsupportedClassVersionError，这个小细节，务必注意。

2、元数据验证

3、字节码验证

4、符号引用验证

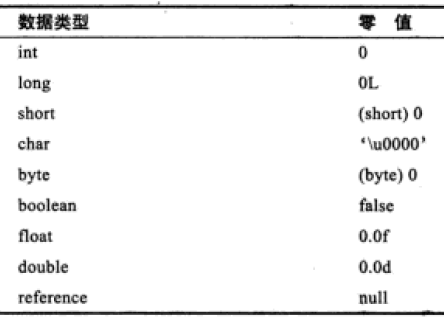
**准备**

**准备阶段是正式为类变量分配内存并设置其初始值的阶段，这些变量所使用的内存都将在方法区中分配**。关于这点，有两个地方注意一下：

1、这时候进行内存分配的仅仅是类变量（被static修饰的变量），而不是实例变量，实例变量将会在对象实例化的时候随着对象一起分配在Java堆中。

2、这个阶段赋初始值的变量指的是那些被final修饰的static变量，比如"public static int value = 123;"，value在准备阶段过后是0而不是123，给value赋值为123的动作将在初始化阶段才进行；比如"public static final int value = 123;"就不一样了，在准备阶段，虚拟机就会给value赋值为123。

各个数据类型的零值如下图：



**解析**

**解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程**。来了解一下符号引用和直接引用有什么区别：

1、符号引用。

这个其实是属于编译原理方面的概念，符号引用包括了下面三类常量：

· 类和接口的全限定名

· 字段的名称和描述符

· 方法的名称和描述符

这么说可能不太好理解，结合实际看一下，写一段很简单的代码：

[复制代码](javascript:void(0);)

1 package com.xrq.test6;

2

3 public class TestMain

4 {

5 private static int i;

6 private double d;

7

8 public static void print()

9 {

10

11 }

12

13 private boolean trueOrFalse()

14 {

15 return false;

16 }

17 }

[复制代码](javascript:void(0);)

用javap把这段代码的.class反编译一下：

[复制代码](javascript:void(0);)

Constant pool:

#1 = Class #2 // com/xrq/test6/TestMain

#2 = Utf8 com/xrq/test6/TestMain

#3 = Class #4 // java/lang/Object

#4 = Utf8 java/lang/Object

#5 = Utf8 i

#6 = Utf8 I

#7 = Utf8 d

#8 = Utf8 D

#9 = Utf8 <init>

#10 = Utf8 ()V

#11 = Utf8 Code

#12 = Methodref #3.#13 // java/lang/Object."<init>":()V

#13 = NameAndType #9:#10 // "<init>":()V

#14 = Utf8 LineNumberTable

#15 = Utf8 LocalVariableTable

#16 = Utf8 this

#17 = Utf8 Lcom/xrq/test6/TestMain;

#18 = Utf8 print

#19 = Utf8 trueOrFalse

#20 = Utf8 ()Z

#21 = Utf8 SourceFile

#22 = Utf8 TestMain.java

[复制代码](javascript:void(0);)

看到Constant Pool也就是常量池中有22项内容，其中带"Utf8"的就是符号引用。比如#2，它的值是"com/xrq/test6/TestMain"，表示的是这个类的全限定名；又比如#5为i，#6为I，它们是一对的，表示变量时Integer（int）类型的，名字叫做i；#6为D、#7为d也是一样，表示一个Double（double）类型的变量，名字为d；#18、#19表示的都是方法的名字。

那其实总而言之，符号引用和我们上面讲的是一样的，是对于类、变量、方法的描述。符号引用和虚拟机的内存布局是没有关系的，引用的目标未必已经加载到内存中了。

2、直接引用

直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同的虚拟机示例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经存在在内存中了。

**初始化**

初始化阶段是类加载过程的最后一步，初始化阶段是真正执行类中定义的Java程序代码（或者说是字节码）的过程。初始化过程是一个执行类构造器<clinit>()方法的过程，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。把这句话说白一点，其实**初始化阶段做的事就是给static变量赋予用户指定的值以及执行静态代码块**。

注意一下，**虚拟机会保证类的初始化在多线程环境中被正确地加锁、同步**，即如果多个线程同时去初始化一个类，那么只会有一个类去执行这个类的<clinit>()方法，其他线程都要阻塞等待，直至活动线程执行<clinit>()方法完毕。因此如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个进程阻塞。不过其他线程虽然会阻塞，但是执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程不会再次进入<clinit>()方法了，因为同一个类加载器下，一个类只会初始化一次。实际应用中这种阻塞往往是比较隐蔽的，要小心。

Java虚拟机规范严格规定了有且只有5种场景必须立即对类进行初始化，这4种场景也称为对一个类进行**主动引用**（其实还有一种场景，不过暂时我还没弄明白这种场景的意思，就先不写了）：

1、使用new关键字实例化对象、读取或者设置一个类的静态字段（被final修饰的静态字段除外）、调用一个类的静态方法的时候

2、使用java.lang.reflect包中的方法对类进行反射调用的时候

3、初始化一个类，发现其父类还没有初始化过的时候

4、虚拟机启动的时候，虚拟机会先初始化用户指定的包含main()方法的那个类

除了上面4种场景外，所有引用类的方式都不会触发类的初始化，称为被动引用，接下来看下被动引用的几个例子：

1、子类引用父类静态字段，不会导致子类初始化。至于子类是否被加载、验证了，前者可以通过"-XX:+TraceClassLoading"来查看

[复制代码](javascript:void(0);)

public class SuperClass

{

public static int value = 123;

static

{

System.out.println("SuperClass init");

}

}

public class SubClass extends SuperClass

{

static

{

System.out.println("SubClass init");

}

}

public class TestMain

{

public static void main(String[] args)

{

System.out.println(SubClass.value);

}

}

[复制代码](javascript:void(0);)

运行结果为

SuperClass init

123

2、通过数组定义引用类，不会触发此类的初始化

[复制代码](javascript:void(0);)

public class SuperClass

{

public static int value = 123;

static

{

System.out.println("SuperClass init");

}

}

public class TestMain

{

public static void main(String[] args)

{

SuperClass[] scs = new SuperClass[10];

}

}

[复制代码](javascript:void(0);)

运行结果为

3、引用静态常量时，常量在编译阶段会存入类的常量池中，本质上并没有直接引用到定义常量的类

[复制代码](javascript:void(0);)

public class ConstClass

{

public static final String HELLOWORLD = "Hello World";

static

{

System.out.println("ConstCLass init");

}

}

public class TestMain

{

public static void main(String[] args)

{

System.out.println(ConstClass.HELLOWORLD);

}

}

[复制代码](javascript:void(0);)

运行结果为

Hello World

在编译阶段通过常量传播优化，常量HELLOWORLD的值"Hello World"实际上已经存储到了NotInitialization类的常量池中，以后NotInitialization对常量ConstClass.HELLOWORLD的引用实际上都被转化为NotInitialization类对自身常量池的引用了。也就是说，实际上的NotInitialization的Class文件中并没有ConstClass类的符号引用入口，这两个类在编译成Class之后就不存在任何联系了。

* 1. **类加载器**

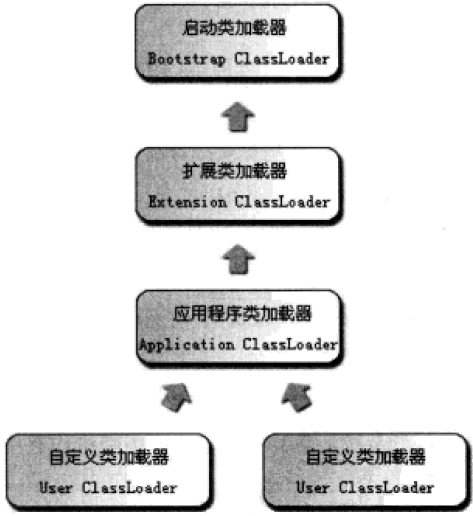
**类与类加载器**

虚拟机设计团队把类加载阶段张的"通过一个类的全限定名来获取此类的二进制字节流"这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为"类加载器"。类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限定于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话表达地再简单一点就是：比较两个类是否"相等"，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则即使这两个类来源于同一个.class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，这两个类必定不相等。

上面说的"相等"，包括代表类的.class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括使用instanceof关键字做对象所属关系判定等情况。

**类加载器模型**

从Java虚拟机的角度讲，只有两种不同的类加载器：启动类加载器Bootstrap ClassLoader，这个类加载器是由C++语言实现的，是虚拟机自身的一部分；其他类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全部继承自java.lang.ClassLoader。从开发人员的角度讲，类加载器还可以划分地更加细致一些，一张图就能说明：



这张图首先说两点：

1、这三个层次的类加载器并不是继承关系，而只是层次上的定义

2、它并不是一个强制性的约束模型，而是Java设计者推荐给开发者的一种类加载器实现方式

OK，然后一个一个类加载器来看：

1、启动类加载器Bootstrap ClassLoader

之前说过了这是一个嵌在JVM内核中的加载器。它负责加载的是JAVA\_HOME/lib下的类库，系统类加载器无法被Java程序直接应用

2、扩展类加载器Extension ClassLoader

这个类加载器由sun.misc.Launcher$ExtClassLoader实现，它负责用于加载JAVA\_HOME/lib/ext目录中的，或者被java.ext.dirs系统变量指定所指定的路径中所有类库，开发者可以直接使用扩展类加载器。java.ext.dirs系统变量所指定的路径的可以通过程序来查看

[复制代码](javascript:void(0);)

public class TestMain

{

public static void main(String[] args)

{

System.out.println(System.getProperty("java.ext.dirs"));

}

}

[复制代码](javascript:void(0);)

运行结果

E:\MyEclipse10\Common\binary\com.sun.java.jdk.win32.x86\_64\_1.6.0.013\jre\lib\ext;C:\Windows\Sun\Java\lib\ext

3、应用程序类加载器Application ClassLoader

这个类加载器由sun.misc.Launcher$AppClassLoader实现。这个类也一般被称为系统类加载器，写个小程序看下：

[复制代码](javascript:void(0);)

public class TestMain

{

public static void main(String[] args)

{

System.out.println(ClassLoader.getSystemClassLoader());

}

}

[复制代码](javascript:void(0);)

运行结果为：

sun.misc.Launcher$AppClassLoader@546b97fd

看到通过"ClassLoader.getSystemClassLoader"，得到的是sun.misc.Launcher$AppClassLoader，这也证明了JDK认为Application ClassLoader是系统类加载器。顺便根据类加载器模型，打印一下这个类的父加载器：

[复制代码](javascript:void(0);)

public class TestMain

{

public static void main(String[] args)

{

System.out.println(ClassLoader.getSystemClassLoader().getParent());

}

}

[复制代码](javascript:void(0);)

运行结果为：

sun.misc.Launcher$ExtClassLoader@535ff48b

看出Application ClassLoader的父加载器确实是Extension ClassLoader，符合图中的模型。那么再打印父加载器呢？按照我们的想法应该是Bootstrap ClassLoader了，看下是不是：

[复制代码](javascript:void(0);)

public class TestMain

{

public static void main(String[] args)

{

System.out.println(ClassLoader.getSystemClassLoader().getParent().getParent());

}

}

[复制代码](javascript:void(0);)

运行结果为：

null

这会打印出来的是null了。其实也很好理解，Bootstrap ClassLoader以外的ClassLoader都是Java实现的，因此这些ClassLoader势必在Java堆中有一份实例在，所以Extension ClassLoader和Application ClassLoader都能打印出内容来。但是Bootstrap ClassLoader是JVM的一部分，是用C/C++写的，不属于Java，自然在Java堆中也没有自己的空间，所以就返回null了。所以，**如果ClassLoader得到的是null，那么表示的ClassLoader就是Bootstrap ClassLoader**。

另外要说很重要的一点，反编译一下rt.jar，找到sun.misc.Launcher看一下Application ClassLoader的实现：

[复制代码](javascript:void(0);)

1 static class AppClassLoader extends URLClassLoader

2 {

3 public static ClassLoader getAppClassLoader(final ClassLoader paramClassLoader)

4 throws IOException

5 {

6 String str = System.getProperty("java.class.path");

7 final File[] arrayOfFile = str == null ? new File[0] : Launcher.getClassPath(str);

8 return (ClassLoader)AccessController.doPrivileged(new PrivilegedAction()

9 {

10 public Launcher.AppClassLoader run()

11 {

12 URL[] arrayOfURL = this.val$s == null ? new URL[0] : Launcher.pathToURLs(arrayOfFile);

13 return new Launcher.AppClassLoader(arrayOfURL, paramClassLoader);

14 }

15 });

16 }

[复制代码](javascript:void(0);)

重点就在第6行，Application ClassLoader只会加载java.class.path下的.class文件，java.class.path代表的是什么路径？打印一下：

[复制代码](javascript:void(0);)

public class TestMain

{

public static void main(String[] args)

{

System.out.println(System.getProperty("java.class.path"));

}

}

[复制代码](javascript:void(0);)

运行结果为：

F:\代码\MyEclipse\TestArticle\bin;F:\学习\第三方jar包\XStream\xstream-1.4.jar;F:\学习\第三方jar包\XStream\kxml2.jar

我这里有添加两个.jar到CLASSPATH下。那也可以下一个重要的结论了：

**Application ClassLoader只能加载项目bin目录下的.class文件**。

**双亲委派模型**

最后讲一下双亲委派模型，其实上面的类加载器模型图就是一个双亲委派模式的图，这里把它再讲清楚一点。

双亲委派模型是在JDK1.2期间被引入的，其工作过程可以分为两步：

1、**如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此。**

2、**只有当父加载器反馈自己无法完成这这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载**

所以，其实所有的加载请求最终都应该传送到顶层的启动类加载器中。双亲委派模型对于Java程序的稳定运作很重要，因为Java类随着它的加载器一起具备了一种带有优先级的层次关系。例如java.lang.Object，存放于rt.jar中，无论哪一个类加载器要去加载这个类，最终都是由Bootstrap ClassLoader去加载，因此Object类在程序的各种类加载器环境中都是一个类。相反，如果没有双亲委派模型，由各个类自己去加载的话，如果用户自己编写了一个java.lang.Object，并放在CLASSPATH下，那系统中将会出现多个不同的Object类，Java体系中最基础的行为也将无法保证，应用程序也将会变得一片混乱。

* 1. **Java内存模型**

**1.10.1 什么是Java内存模型**

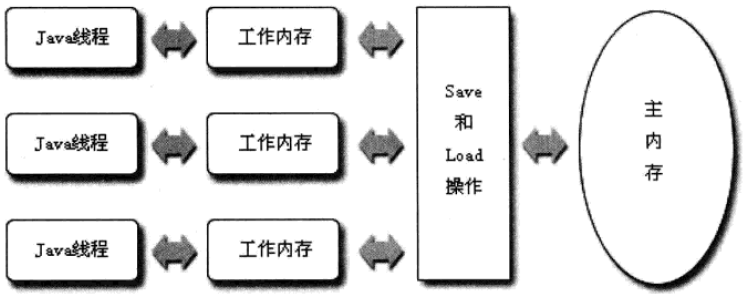
Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽掉各种硬件和操作系统的访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。在此之前，主流程序语言（如C/C++等）直接使用物理硬件和操作系统的内存模型，因此，会由于不同平台上内存模型的差异，有可能导致程序在一套平台上并发完全正常，而在另外一套平台上并发访问却经常出错，因此在某些场景下就不许针对不同的平台来编写程序。

Java内存模型即要定义得足够严谨，才能让Java的并发内存访问操作不会产生歧义；Java内存模型也必须定义地足够宽松，才能使得虚拟机的实现有足够的自由空间去利用硬件的各种特性来获取更好的执行速度。经过长时间的验证和修补，JDK1.5（实现了JSR-133）发布之后，Java内存模型已经成熟和完善起来了，一起来看一下。

**主内存和工作内存**

Java内存模型的主要目的是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。注意一下，此处的变量并不包括局部变量与方法参数，因为它们是线程私有的，不会被共享，自然也不会存在竞争，此处的变量应该是实例字段、静态字段和构成数组对象的元素。

Java内存模型规定了所有的变量都存储在主内存（Main Memory）中，每条线程还有自己的工作内存（Working Memory），线程的工作内存中保存了被该线程使用到的变量和主内存副本拷贝（注意**这里绝不会是整个对象的拷贝，试想一个10M的对象，在每个用到这个对象的工作内存中有一个10M的拷贝，内存还受得了？也就是一些在线程中用到的对象中的字段罢了**），**线程对变量所有的操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成**，线程、主内存、工作内存三者的交互关系如图：



**内存间相互交互**

关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java内存模型中定义了以下8种操作来完成，虚拟机实现时必须保证下面的每一种操作都是原子的、不可再分的：

1、lock（锁定）：作用于主内存中的变量，它把一个变量标识为一条线程独占的状态

2、unlock（解锁）：作用于主内存中的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定

3、read（读取）：作用于主内存中的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用

4、load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中

5、use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作

6、assign（赋值）：作用于工作内存中的变量，它把一个从执行引擎接收到的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作

7、store（存储）：作用于工作内存中的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用

8、write（写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量值放入主内存的变量中

Java内存模型还规定了在执行上述8种基本操作时必须满足以下规则：

1、不允许read和load、store和write操作之一单独出现

2、不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了滞后必须把该变化同步回主内存

3、不允许一个线程无原因地把数据从线程的工作内存同步回主内存中

4、一个新的变量只能从主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量

5、**一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁**

6、**如果对同一个变量执行lock操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值**

7、如果一个变量事先没有被lock操作锁定，那就不允许对它进行unlock操作，也不允许去unlock一个被其他线程锁定的变量

8、**对一个变量执行unlock操作之前，必须先把此变量同步回主内存中**

**volatile型变量的特殊规则**

关键字volatile可以说是Java虚拟机提供的最轻量级的同步机制。

一个变量被定义为volatile后，它将具备两种特性：

1、**保证此变量对所有线程的"可见性"**，所谓"可见性"是指当一条线程修改了这个变量的值，新值对于其它线程来说都是可以立即得知的，而普通变量不能做到这一点，普通变量的值在在线程间传递均需要通过主内存来完成，关于volatile关键字的操作请参见[volatile关键字使用举例](http://www.cnblogs.com/xrq730/p/4853578.html)，再强调一遍，**volatile只保证了可见性，并不保证基于volatile变量的运算在并罚下是安全的**

2、**使用volatile变量的第二个语义是禁止指令重排序优化**，普通变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。

总结一下Java内存模型对volatile变量定义的特殊规则：

1、在工作内存中，每次使用某个变量的时候都必须线从主内存刷新最新的值，用于保证能看见其他线程对该变量所做的修改之后的值

2、在工作内存中，每次修改完某个变量后都必须立刻同步回主内存中，用于保证其他线程能够看见自己对该变量所做的修改

3、volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序顺序相同

**原子性、可见性、有序性**

Java内存模型围绕着并发过程中如何处理原子性、可见性和有序性这三个特征来建立的，来逐个看一下：

1、原子性（Atomicity）

由Java内存模型来直接保证原子性变量操作包括read、load、assign、use、store、write，大致可以认为基本数据类型的访问读写是具备原子性的。如果应用场景需要一个更大的原子性保证，Java内存模型还提供了lock和unlock，尽管虚拟机没有把lock和unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐式地使用这两个操作，这两个字节码指令反映到Java代码中就是同步块----synchronized关键字

2、可见性（Visibility）

可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。volatile其实已经详细写了这一点，其实synchronized关键字也是可以实现可见性的，synchronized的可见性是由"对一个变量执行unlock操作之前，必须先把此变量同步回主内存中"这条规则获得的。另外，final关键字也可以实现可见性，因为被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把this传递出去，那在其他线程中就能看见final字段的值。·

3、有序性（Ordering）

Java程序中天然的有序性可以总结为一句话：**如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另外一个线程，所有的操作都是无须的**。前半句是指"线程内表现为穿行的语义"，后半句是指"指令重排序"和"工作内存与主内存同步延迟"现象。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由"一个变量在同一时刻只允许一条线程对其进行lock操作"这条规则获得的，这条规则规定了持有同一个锁的两个同步块只能串行地进入

**先行发生happens-before原则**

如果Java内存模型中所有的有序性都仅仅靠volatile和synchronized来完成，那么有一些操作将变得很繁琐，但是我们在编写Java代码时并未感觉到这一点，这是因为Java语言中有一个"先行发生（happens-before）"原则。这个原则非常重要，它是判断数据是否存在竞争、线程是否安全的主要依据，依靠这个原则，我们可以通过几条规则就判断出并发环境下两个操作之间是否可能存在冲突的问题。

所谓先行发生原则是指Java内存模型中定义的两项操作之间的偏序关系，如果说操作A先行发生于操作B，那么操作A产生的影响能够被操作b观察到，"影响"包括修改了内存中共享变量的值、发送了消息、调用了方法等。Java内存模型下有一些天然的，不需要任何同步协助器就已经存在的先行发生关系：

1、程序次序规则：在一个线程内，按照控制流顺序，控制流前面的操作先行发生于控制流后面的操作，说"控制流"是因为还要考虑到分支、循环结构

2、管程锁定规则：一个unlock操作先行发生于后面对**同一个锁**的lock操作

3、volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作

4、线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

5、线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测

6、线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

7、对象终结规则：一个对象的初始化完成先行发生于它的finalize()方法的开始

8、传递新：如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A必然先行发生于操作C

Java语言无须任何同步手段保障就能成立的先行发生规则就只有上面这些额，如果两个操之间的关系不在此列，并且无法通过下面规则推导出来的话，它们就没有顺序性保障。举一个例子来看一下：

[复制代码](javascript:void(0);)

private int i = 0;

public void setI(int i)

{

this.i = i;

}

public int getI()

{

return i;

}

[复制代码](javascript:void(0);)

很普通的一组getter/setter，假设A线程先调用了setI(1)，B线程再调用了同一个对象的getI()，那么B线程的返回值是什么？

依次分析一下先行发生原则中的各项规则。由于两个方法分别由两个线程分别调用，因此程序次序规则这里不适用；由于没有同步块，所以也就没有unlock和lock，因此管程锁定规则这里不适用；i没有被关键字volatile修饰，因此volatile变量规则这里不适用；后面的启动、终止、中断、对象终结也和这里完全没有关系，因此也都不适用。因为没有一个实用的先行发生规则，所以最后一条传递性也无从谈起，因此传递性也不适用。由于所有先行发生原则都不适用，因此尽管线程A的setI(1)操作在时间上先发生，但无法确定线程B的getI()的返回结果，换句话说，这里面的操作不是线程安全的。

那如何修复这个问题？至少有两种比较简单的办法：

1、setter/getter都定义成synchronized的，这样可以套用管程锁定规则

2、i定义为volatile变量，由于setter方法对i的修改不依赖于i的原值，满足volatile关键字的使用场景，这样可以套用volatile变量规则

* 1. **互斥同步、锁优化及synchronized和volatile**

**互斥同步**

互斥同步（Mutual Exclusion & Synchronization）是常见的一种并发正确性保证手段。同步是指子啊多个线程并发访问共享数据时，保证共享数据在同一时刻只能被一个（或者是一些，使用信号量的时候）线程使用。而互斥是实现同步的一种手段，临界区（Critial Section）、互斥量（Mutex）和信号量（Semaphore）都是主要的互斥实现方式。因此，在这四个字里面，互斥是因，同步是果；互斥是方法，同步是目的。

**synchronized的实现**

在Java中，大家都知道，synchronized关键字是最基本的互斥同步手段。看一段简单的代码：

[复制代码](javascript:void(0);)

public static void main(String[] args)

{

synchronized (TestMain.class)

{

}

}

[复制代码](javascript:void(0);)

这段代码被编译之后是这样的：

[复制代码](javascript:void(0);)

1 public static void main(java.lang.String[]);

2 flags: ACC\_PUBLIC, ACC\_STATIC

3 Code:

4 stack=2, locals=1, args\_size=1

5 0: ldc #1 // class com/xrq/test53/TestMain

6 2: dup

7 3: monitorenter

8 4: monitorexit

9 5: return

10 LineNumberTable:

11 line 7: 0

12 line 11: 5

13 LocalVariableTable:

14 Start Length Slot Name Signature

15 0 6 0 args [Ljava/lang/String;

[复制代码](javascript:void(0);)

关键就在第7行和第8行，在源代码被编译之后，Java虚拟机会利用monitorenter和monitorexit条字节码指令来处理synchronized这个关键字。

根据虚拟机规范的要求，在执行monitorenter指令时，首先要尝试获取对象的锁，如果这个对象没有被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加1，相应地，在执行monitorexit指令时会将锁计数器减1，当计数器为0时，锁就会被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

关于monitorenter和monitorexit，有两点是要特别注意的：

1、synchronized同步块对同一条线程来说是可重入的，不会出现把自己锁死的问题

2、同步块在已进入的线程执行完之前，会阻塞后面其它线程的进入

因为Java的线程是映射到操作系统的原生线程之上的，如果要阻塞或者唤醒一个线程，都需要操作系统来帮忙完成，这就需要**从用户态转换到核心态中**，因此状态转换需要耗费很多的处理器时间，对于代码简单的同步块，状态转换消耗的时间有可能比用户代码执行的时间还长，所以synchronized是Java语言中一个重量级（Heavyweight）锁，有经验的程序员都会在确实必要的情况下才使用这种操作。

顺便看一下HotSpot虚拟机对象头Mark Word：

|  |  |  |
| --- | --- | --- |
| **存 储 内 存** | **标 识 位** | **状    态** |
| 对象哈希吗、对象分代年龄 | 01 | 未锁定 |
| 指向锁记录的指针 | 00 | 轻量级锁定 |
| 指向重量级锁的指针 | 10 | 膨胀（重量级锁定） |
| 空，不需要记录信息 | 11 | GC标记 |
| 偏向线程ID、偏向时间戳、对象分代年龄 | 01 | 可偏向 |

看到有一个重量级锁定，指的就是重量级锁。

**volatile的实现**

对于volatile关键字，一个被volatile关键字修饰的变量，在生成汇编语言之后，大致会多出这么一条指令：

0x01a3de24:lock addl $0x0,(%esp) ;...f0830424 00

这个操作相当于是一个内存屏障，只有一个CPU访问内存时，并不需要内存屏障；但如果有两个或者更多CPU访问同一块内存时，且其中一个在观测另外一个，就需要内存屏障来保证一致性了。这句指令中的"addl $0x0,(%esp)"（把esp寄存器的值加0）显然是一个空操作（采用这个空操作而不是空指令nop是因为IA32手册规定lock前缀不允许配合nop指令使用），关键在于lock前缀，查询IA32手册，它的作用是使得本CPU的Cache写入了内存，该写入动作也会引起别的CPU或者别的内核无效化其Cache，这种操作相当于对Cache中的变量做了一次"store和write"操作，所以通过这样一个空操作，可让前面volatile变量的修改对其他CPU立即可见。

**自旋锁与自适应自旋**

互斥同步，对性能影响最大的是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核状态完成，这些操作给系统的并发性能带来了很大的压力。同时，虚拟机开发团队也注意到很多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如果物理机上有一个以上的处理器，能让两个或两个以上的线程同时并行执行，我们就可以让后面请求锁的那个线程"稍等一下"，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需要**让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁**。

在JDK1.4.2就已经引入了自旋锁，只不过默认是关闭的。**自旋不能代替阻塞**，且先不说处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但是它是要占据处理器时间的，因此如果锁被占用的时间很短，自旋等待的效果就非常好；反之，如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作，反而会带来性能上的浪费。因此自选等待必须有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了，自旋次数的默认值是10。

在JDK1.6之后引入了自适应的自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上自旋的时间以及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。另外如果对于某一个锁，自旋很少成功获得过，那么在以后要获得这个锁时将可能忽略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测就会越来越准确。

**锁消除**

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于[逃逸分析](http://www.cnblogs.com/xrq730/p/4857820.html)的支持，如果判断在一段代码中，堆上所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然无需进行。

**锁粗化**

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小----只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数尽可能变小，如果存在锁竞争，那等待锁的线程也能尽快拿到锁。

大部分情况下，上面的原则都是正确的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。

如果这么说不够直观，那么想想某段代码反复使用StringBuffer的append方法拼接字符串的例子吧。

**JAVA反射**

1. 反射的应用场合

在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息

2. 反射的作用

通过反射可以使程序代码访问装载到JVM 中的类的内部信息

获取已装载类的属性信息

获取已装载类的方法

获取已装载类的构造方法信息

反射主要解决动态编程,即使用反射时,所有的对象生成是动态的,因此调用的方法也是动态的.反射可以简化开发,但是代码的可读性很低.

Java中所有类型(包括基本类型)都对应一个Class对象,这个Class就是java.lang.Class。即每一个类型,在Class中都有一个Class对象跟它对应.Class 没有公共构造方法。注意不是没有,是没有公共的.

**如何获得Class对象**

1. 1.针对每一个对象.getCalss(),可以得到对应的Class.
2. 2.Class.forName(String),String的写法:包名.类名.就会创建包名.类名对应的那个对象
3. 注：1.2只适用于引用类型
4. 3.对于基本类型：封装类.TYPE代表了对应的基本类型的Class对象.Integer.TYPE对应的是**int**的Class对象
5. 注：3只适用于基本类型
6. 4.类型.Class。<第4种是通用的.>
7. 上面的4种方法,只有方法2是动态的,只要换一个包就可以了.它具有动态潜质.所以真正意义的想体现动态编程只能使用方法2.

每种类型的Class对象只有一个,即他们的地址只有一个,但是不同类型是不同的.

所以下面的打印结果都为true.

1. //对与引用类型
2. Class c1 = "".getClass();
3. Class c2 =     Class.forName("java.lang.String");
4. Class c3 = String.**class**;
5. System.out.println(c1 ==c2);//true
6. //对于基本类型
7. Class num1 = Integer.TYPE;
8. Class num2 = **int**.**class**;
9. System.out.println(num1 == num2);//true

**反射获取类中的成员的相关方法**

[获取构造<根据参数类型>](使用时一般用不带declared的)

1. Constructor<T> getConstructor(Class<?>... parameterTypes)
2. 返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共构造方法。
3. Constructor<?>[] getConstructors()
4. 返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法。
5. Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
6. 返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法。
7. Constructor<?>[] getDeclaredConstructors()
8. 返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。

[获取属性<根据属性名>](使用时一般用是带declared的,因为属性一般都是私有的)

1. Field getField(String name)
2. 返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段。
3. Field[] getFields()
4. 返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段。
5. Field getDeclaredField(String name)
6. 返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段。
7. Field[] getDeclaredFields()
8. 返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段。

[获取方法<方法名加上参数类型>](使用时一般用不带declared的)

1. Method getMethod(String name, Class<?>... parameterTypes)
2. 返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。
3. Method[] getMethods()
4. 返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。
5. Method getDeclaredMethod(String name, Class<?>... parameterTypes)
6. 返回一个 Method 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。
7. Method[] getDeclaredMethods()
8. 返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。
9. T newInstance()
10. 创建此 Class 对象所表示的类的一个新实例。 <**new** Instance()可以动态的创建对象>
11. String toString()
12. 将对象转换为字符串。

注意：

new Instance()调用的是无参构造,如果该类没有无参构造方法,则newInstance()会产生异常.

有declared的方法是支持私有,但是不支持继承,无declared的方法支持继承,不支持私有,且只能取出public的东西.

因此取属性的时候一般来说是带declared的,因为属性一般都是私有的,取方法时一般是不带declared的,取构造时一般也是不带declared的.

**利用反射对属性赋值**

Field中的方法

 Object get(Object obj)

  返回指定对象上此 Field 表示的字段的值。

      Field f = c.getXXField(属性名);

      值 = f.get(对象);

 void set(Object obj, Object value)

  将指定对象变量上此 Field 对象表示的字段设置为指定的新值。

  f.set(对象,值);

 Class<?> getType()

  返回一个 Class 对象，它标识了此 Field 对象所表示字段的声明类型。

      用于获取属性的类型(返回Class对象).

1. Class c = Student.**class**;
2. Object obj  = c.newInstance();            //创建Student类的对象
3. Field f = c.getDeclaredField("name");        //获取name属性
4. f.setAccessible(**true**);                    //设置私有可以访问.
5. f.set(obj, "zhangsan");
6. System.out.println(f.get(obj));             //获取obj的name属性的值.

**利用反射调用构造**

对于构造真正调用是在调用newInstance()方法时.

1. Class c = Class.forName("com.clazz.reflect.Student");
2. Constructor con = c.getConstructor();         //没有执行构造,
3. Object cObj = c.getConstructor().newInstance();//调用无参的构造方法
4. Constructor conAll = c.getConstructor(**int**.**class**,String.**class**,**int**.**class**);
5. Object caobj = conAll.newInstance(1001,"zjamgs",234235);//调用含参的构造方法.
6. System.out.println(caobj);                  //打印输出

**利用反射调用方法**

对象.方法名(值1,2,3);

Method m = c.getMethoed(方法名，参数类型...);

m.invoke(对象,方法调用的参数 )如果底层方法所需的形参数为 0，则所提供的 args 数组长度可以为 0 或 null。

1. Class c = Class.forName("com.clazz.reflect.Student");
2. Object obj = c.newInstance();    //创建Sutdent对象.
3. Method msetName = c.getMethod("setName", String.**class**);//obj无须转换类型
4. msetName.invoke(obj, "zhangsan");//调用方法setName, 并传参.
5. Method msetId = c.getMethod("setId", **int**.**class**);
6. msetId.invoke(obj, 409090202);
7. System.out.println(obj);

本篇文章为在工作中使用JAVA反射的经验总结，也可以说是一些小技巧，以后学会新的小技巧，会不断更新。本文不准备讨论JAVA反射的机制，网上有很多，大家随便google一下就可以了。

        在开始之前，我先定义一个测试类Student，代码如下：

**[Java]** [view plain](http://blog.csdn.net/hbcui1984/article/details/2719089) [copy](http://blog.csdn.net/hbcui1984/article/details/2719089)

1. **package** chb.test.reflect;
3. **public** **class** Student {
4. **private** **int** age;
5. **private** String name;
6. **public** **int** getAge() {
7. **return** age;
8. }
9. **public** **void** setAge(**int** age) {
10. **this**.age = age;
11. }
12. **public** String getName() {
13. **return** name;
14. }
15. **public** **void** setName(String name) {
16. **this**.name = name;
17. }
19. **public** **static** **void** hi(**int** age,String name){
20. System.out.println("大家好，我叫"+name+"，今年"+age+"岁");
21. }
22. }<pre></pre>

**一、JAVA反射的常规使用步骤**

    反射调用一般分为3个步骤：

* 得到要调用类的class
* 得到要调用的类中的方法(Method)
* 方法调用(invoke)

     代码示例：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Method m = cls.getDeclaredMethod("hi",**new** Class[]{**int**.**class**,String.**class**});
3. m.invoke(cls.newInstance(),20,"chb");

**二、方法调用中的参数类型**

        在方法调用中，参数类型必须正确，这里需要注意的是不能使用包装类替换基本类型，比如不能使用Integer.class代替int.class。

       如我要调用Student的setAge方法，下面的调用是正确的：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Method setMethod = cls.getDeclaredMethod("setAge",**int**.**class**);
3. setMethod.invoke(cls.newInstance(), 15);<pre></pre>

       而如果我们用Integer.class替代int.class就会出错，如：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Method setMethod = cls.getDeclaredMethod("setAge",Integer.**class**);
3. setMethod.invoke(cls.newInstance(), 15);<pre></pre>

       jvm会报出如下异常：

1. java.lang.NoSuchMethodException: chb.test.reflect.Student.setAge(java.lang.Integer)
2. at java.lang.Class.getDeclaredMethod(Unknown Source)
3. at chb.test.reflect.TestClass.testReflect(TestClass.java:23)**<pre></pre>**

**三、static方法的反射调用**

       static方法调用时，不必得到对象示例，如下：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Method staticMethod = cls.getDeclaredMethod("hi",**int**.**class**,String.**class**);
3. staticMethod.invoke(cls,20,"chb");//这里不需要newInstance
4. //staticMethod.invoke(cls.newInstance(),20,"chb");<pre></pre>

**四、private的成员变量赋值**

    如果直接通过反射给类的private成员变量赋值，是不允许的，这时我们可以通过setAccessible方法解决。代码示例：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Object student = cls.newInstance();//得到一个实例
3. Field field = cls.getDeclaredField("age");
4. field.set(student, 10);
5. System.out.println(field.get(student));<pre></pre>

     运行如上代码，系统会报出如下异常：

1. java.lang.IllegalAccessException: Class chb.test.reflect.TestClass can not access a member of class chb.test.reflect.Student with modifiers "private"
2. at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
3. at java.lang.reflect.Field.doSecurityCheck(Unknown Source)
4. at java.lang.reflect.Field.getFieldAccessor(Unknown Source)
5. at java.lang.reflect.Field.set(Unknown Source)
6. at chb.test.reflect.TestClass.testReflect(TestClass.java:20)**<pre></pre>**

    解决方法：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Object student = cls.newInstance();
3. Field field = cls.getDeclaredField("age");
4. field.setAccessible(**true**);//设置允许访问
5. field.set(student, 10);
6. System.out.println(field.get(student));<pre></pre>

其实，在某些场合下(类中有get,set方法)，可以先反射调用set方法，再反射调用get方法达到如上效果，代码示例：

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Object student = cls.newInstance();
4. Method setMethod = cls.getDeclaredMethod("setAge",Integer.**class**);
5. setMethod.invoke(student, 15);//调用set方法
7. Method getMethod = cls.getDeclaredMethod("getAge");
8. System.out.println(getMethod.invoke(student));//再调用get方法<pre></pre>

**反射技术优缺点**

反射提高了Java程序的灵活性和扩展性，降低耦合性，提高自适应能力。它允许程序创建和控制任何类的对象，无需提前硬编码目标类

**反射的缺点**

性能问题,使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此Java反射机制主要应用在对灵活性和扩展性要求很高的系统框架上,普通程序不建议使用。

使用反射会模糊程序内部逻辑 ,程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂。

**JAVA注解**

注解目前非常的流行，很多主流框架都支持注解，而且自己编写代码的时候也会尽量的去用注解，一时方便，而是代码更加简洁。

     注解的语法比较简单，除了@符号的使用之外，它基本与Java固有语法一致。Java SE5内置了三种标准注解：

     @Override，表示当前的方法定义将覆盖超类中的方法。

     @Deprecated，使用了注解为它的元素编译器将发出警告，因为注解@Deprecated是不赞成使用的代码，被弃用的代码。

     @SuppressWarnings，关闭不当编译器警告信息。

     上面这三个注解多少我们都会在写代码的时候遇到。Java还提供了4中注解，专门负责新注解的创建。

|  |  |
| --- | --- |
| @Target | 表示该注解可以用于什么地方，可能的ElementType参数有：  CONSTRUCTOR：构造器的声明  FIELD：域声明（包括enum实例）  LOCAL\_VARIABLE：局部变量声明  METHOD：方法声明  PACKAGE：包声明  PARAMETER：参数声明  TYPE：类、接口（包括注解类型）或enum声明 |
| @Retention | 表示需要在什么级别保存该注解信息。可选的RetentionPolicy参数包括：  SOURCE：注解将被编译器丢弃  CLASS：注解在class文件中可用，但会被VM丢弃  RUNTIME：VM将在运行期间保留注解，因此可以通过反射机制读取注解的信息。 |
| @Document | 将注解包含在Javadoc中 |
| @Inherited | 允许子类继承父类中的注解 |

　　定义一个注解的方式：

1 @Target(ElementType.METHOD)

2 @Retention(RetentionPolicy.RUNTIME)

3 public @interface Test {

4

5 }

除了@符号，注解很像是一个接口。定义注解的时候需要用到元注解，上面用到了@Target和@RetentionPolicy，它们的含义在上面的表格中已近给出。

     在注解中一般会有一些元素以表示某些值。注解的元素看起来就像接口的方法，唯一的区别在于可以为其制定默认值。没有元素的注解称为标记注解，上面的@Test就是一个标记注解。

     注解的可用的类型包括以下几种：所有基本类型、String、Class、enum、Annotation、以上类型的数组形式。元素不能有不确定的值，即要么有默认值，要么在使用注解的时候提供元素的值。而且元素不能使用null作为默认值。注解在只有一个元素且该元素的名称是value的情况下，在使用注解的时候可以省略“value=”，直接写需要的值即可。

     下面看一个定义了元素的注解。

1 @Target(ElementType.METHOD)

2 @Retention(RetentionPolicy.RUNTIME)

3 public @interface UseCase {

4 public String id();

5 public String description() default "no description";

6 }

  定义了注解，必然要去使用注解。

[复制代码](javascript:void(0);)

1 public class PasswordUtils {

2 @UseCase(id = 47, description = "Passwords must contain at least one numeric")

3 public boolean validatePassword(String password) {

4 return (password.matches("\\w\*\\d\\w\*"));

5 }

6

7 @UseCase(id = 48)

8 public String encryptPassword(String password) {

9 return new StringBuilder(password).reverse().toString();

10 }

11 }

[复制代码](javascript:void(0);)

使用注解最主要的部分在于对注解的处理，那么就会涉及到注解处理器。

     从原理上讲，注解处理器就是通过反射机制获取被检查方法上的注解信息，然后根据注解元素的值进行特定的处理。

[复制代码](javascript:void(0);)

public static void main(String[] args) {

List<Integer> useCases = new ArrayList<Integer>();

Collections.addAll(useCases, 47, 48, 49, 50);

trackUseCases(useCases, PasswordUtils.class);

}

public static void trackUseCases(List<Integer> useCases, Class<?> cl) {

for (Method m : cl.getDeclaredMethods()) {

UseCase uc = m.getAnnotation(UseCase.class);

if (uc != null) {

System.out.println("Found Use Case:" + uc.id() + " "

+ uc.description());

useCases.remove(new Integer(uc.id()));

}

}

for (int i : useCases) {

System.out.println("Warning: Missing use case-" + i);

}

}

**Annotation 的优点**

　　１、保存在 class 文件中，降低维护成本。

２、无需工具支持，无需解析。

３、编译期即可验证正确性，查错变得容易。

４、节省配置，减少配置文件大小

5、提升开发效率。

**Annotation 缺点**：

１、若要对配置项进行修改，不得不修改 Java 文件，重新编译打包应用。

２、配置项编码在 Java 文件中，可扩展性差。