**JVM性能调优**

# 自动内存管理工具

## 概述：

运行时数据区包括方法区，虚拟机栈，本地方法栈，堆，程序计数器

程序计数器：对应于每个线程，用于确定线程执行到了哪一行。

虚拟机栈：对应于每个线程。生命周期和线程相同。描述的是方法执行的内存模型。每个方法在执行的时候创建一个栈帧。栈帧用于存储局部变量表、操作数栈、动态连接、方法出口等信息。

本地方法栈：同虚拟机栈，但是针对本地调用。在hospot中的将虚拟机栈和本地方法栈合二为一。

堆：被所有线程共享的一块内存。存放所有对象实例。是垃圾收集器管理的主要区域。也叫GC堆。

方法区：属于堆的一个逻辑部分。各个线程共享的内存区域。用于存储被加载的类信息，常量，静态变量，即时编译后的代码等数据。

## 程序计数器

对应于每个线程，用于确定线程执行到了哪一行。

各条线程间程序计数器互不影响。程序计数器的存储内存被称为线程私有内存

如果线程正在执行一个方法，线程内的计数器标识正在执行的虚拟机字节码指令的地址

如果线程正在执行的是一个本地方法，这个计数器为空

## 虚拟机栈

对应于每个线程。生命周期和线程相同。描述的是方法执行的内存模型。每个方法在执行的时候创建一个栈帧。栈帧用于存储局部变量表、操作数栈、动态连接、方法出口等信息。

### 栈帧

每个方法在执行的时候创建一个栈帧。栈帧用于存储局部变量表、操作数栈、动态连接、方法返回地址等信息。

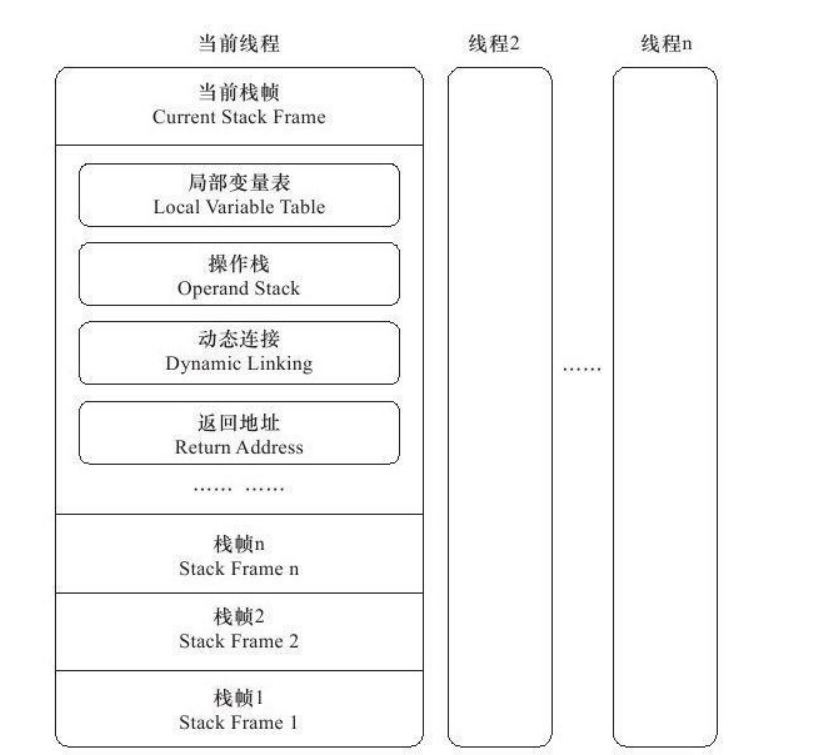
在程序编译的时候即已确定每个方法需要多大的局部变量表，多深的操作数栈，并且写入到方法表的Code属性中。因此一个栈帧需要分配多少内存，不会受到程序运行期间的影响，而仅仅取决于具体的虚拟机实现。

每一个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机中入栈到出栈的过程。

是方法运行时的基础数据结构。

一个线程中的方法调用链很长，比如方法A调用了方法B，方法B调用了方法C，这样当方法A被调用时，方法A，B，C都会同时处于执行状态。即有三个栈帧同时处于虚拟机栈中。其中A处于顶部，B，C依次，A叫当前栈帧。对于执行引擎来说只有当前栈帧是有效的，与这个栈帧关联的A方法叫当前方法。

概念模型如图：



#### 局部变量表

表示栈帧中的局部变量的存储，里面存放了方法参数和方法内部定义的局部变量。在java程序编译为class文件时，就在方法的code属性的max\_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

存放了编译器可知的各种基本数据类型、对象引用和return Address(执行了一条字节码指令的地址)。

比如一个方法 void test(){ int a =0; ...} 那么在执行这个方法的时候会生成一个栈帧，栈帧中的局部变量表中会存储a=0的信息。

局部变量表的容量以变量槽(slot)为单位。比如一个slot可以存放一个32以内的数据类型。对于64位的数据类型，以高位对齐的方式为其分配两个连续的slot空间比如long和double。

虚拟机通过索引定位的方式使用局部变量表。范围从0到局部变量表的最大slot数量。

如果是实例方法，局部变量表中的第0位索引的slot默认是用于传递方法所属对象实例的引用，即this。

为了节省栈帧空间，slot会复用。比如方法体中的变量，当作用域没有覆盖整个方法体且当前字节码PC计数器的值已经超出了变量的作用域，那这个变量对应的slot就可以交给其他变量来使用。

局部变量表不像类变量存在”准备阶段”，类变量有两次赋初始值的过程。一次在准备阶段，赋予系统初始值，一次在初始化阶段，赋予程序员定义的初始值。因此，即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然有一个明确的初始值。但局部变量如果没有赋初始值是不能使用的，编译即会报错。

#### 操作数栈

也成为操作栈，是一个后入先出栈。操作数栈的最大深度也在编译的时候写入方法的Code属性的max\_stacks数据项中。操作数栈的每一个元素可以是任意的JAVA数据类型，包括long和double。32位数据类型栈容量为1，64位的栈容量为2。在方法执行的时候，操作树栈的深度都不会超过在max\_stacks数据项中设定的最大值。

当一个方法刚刚开始执行的时候，操作数栈是空的，执行过程中就会从操作数栈中写入和提取内容。比如算术运算的时候或者调用其它方法的时候都是通过操作数栈来进行参数传递的。

例如：整数加法的字节码指令iadd在运行的时候操作数栈中最接近栈顶的两个元素

已经存入了两个int型的数值，当执行这个指令时，会将这两个int值出栈并相加，然后将相加

的结果入栈。

#### 动态连接

每个栈帧都包含一个指向运行时常量池 [1] 中该栈帧所属方法的引用，持有这个引用是为

了支持方法调用过程中的动态连接（Dynamic Linking）

#### 方法返回地址

方法正常退出时，调用者的PC计数器的值可以作为返回地址，栈帧中很可

能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈

帧中一般不会保存这部分信息。

方法退出的过程实际上就等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复

上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈

中，调整PC计数器的值以指向方法调用指令后面的一条指令等。

#### 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧之中，例如与

调试相关的信息，这部分信息完全取决于具体的虚拟机实现，这里不再详述。在实际开发

中，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，称为栈帧信息。

## 本地方法栈

与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行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（譬如Sun HotSpot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。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”

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；再细致一点的有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。

## 方法区

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、静态变量、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。

在hotspot中也被称为永久代。

### 运行时常量池

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）

运行时常量池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。

比如 private static final String aaa= “ddddddd”;

## 直接内存

直接内存并不是虚拟机运行时数据的一部分，也不是Java虚拟机规范中定义的内存区域。

NIO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。

## Hotspot对象探秘

### 对象的创建

1. 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
2. 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。本地线程分配缓冲解决并发分配空间的线程安全问题。例如：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指针来分 配内存的情况。

1. 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（不包括对象头），

如果使用TLAB，这一工作过程也可以提前至TLAB分配时进行。这一步操作保证了对象 的实例字段在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。

1. 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（Object Header）之中

1. 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，所有的字段都还为零。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

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（HashCode）、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（未开启压缩指针）中分别为32bit和64bit，官方称它为“Mark Word”。

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。

#### 实例数据

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

型的字段内容。无论是从父类继承下来的，还是在子类中定义的，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（FieldsAllocationStyle）和字段在Java源码中定义顺

序的影响。

#### 对齐填充

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（1倍或者2倍），

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

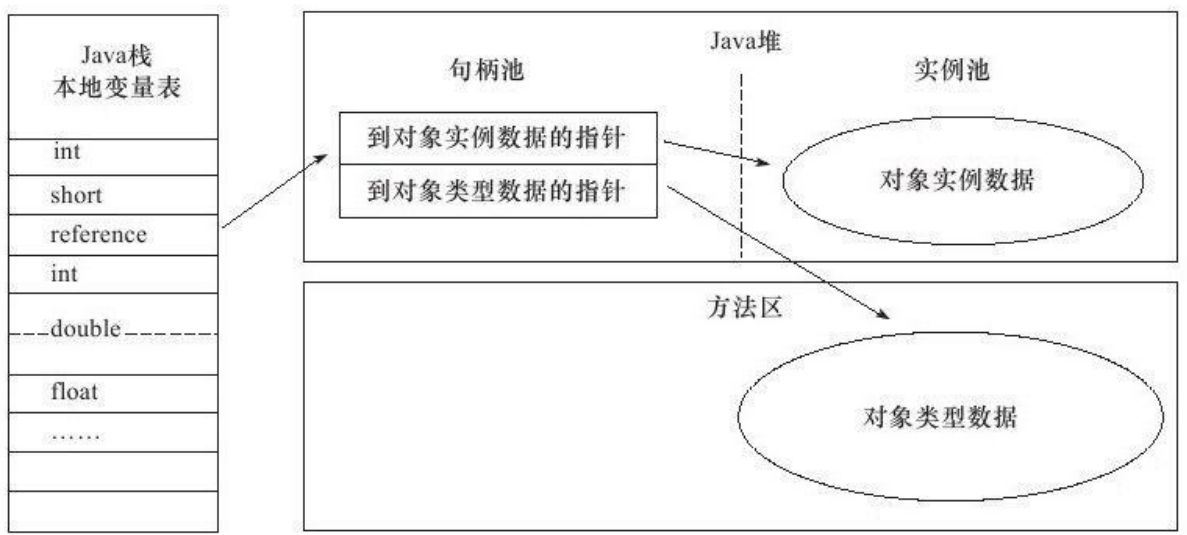
通过栈上的reference数据来操作堆上的

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。

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

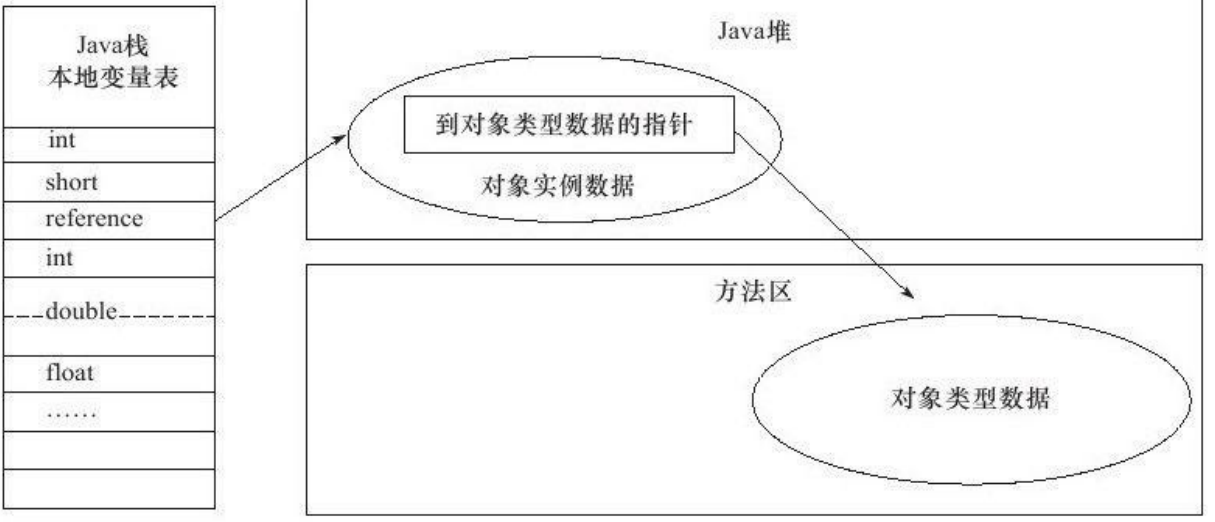
存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信

息。如下所示：



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放置访问类型数据的

相关信息，而reference中存储的直接就是对象地址，hotspot的使用方式。如图所示



## OutofMemoryError异常

目的：

1 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

2 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，知道什么样的代码可能会导致这些区域内存溢出，以及出现这些异常后该如何处理。

### 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。

限制Java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：+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后进行分析。

### 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

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。

定义了大量的本地变量，增大此方法栈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缩小。

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。

### 方法区和运行时常量池溢出

由于运行时常量池是方法区的一部分，因此这两个区域的溢出测试就放在一起进行。前

面提到JDK 1.7开始逐步“去永久代”的事情，在此就以测试代码观察一下这件事对程序的实际影响。

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。

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。对于这些区域的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。虽

然直接使用Java SE API也可以动态产生类（如反射时的GeneratedConstructorAccessor和动态代理等），但在本次实验中操作起来比较麻烦。在代码清单2-8中，笔者借助CGLib [1] 直接操作字节码运行时生成了大量的动态类。

值得特别注意的是，我们在这个例子中模拟的场景并非纯粹是一个实验，这样的应用经

常会出现在实际应用中：当前的很多主流框架，如Spring、Hibernate，在对类进行增强时，

都会使用到CGLib这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的

Class可以加载入内存。另外，JVM上的动态语言（例如Groovy等）通常都会持续创建类来实现语言的动态性，随着这类语言的流行，也越来越容易遇到与代码清单2-8相似的溢出场

景。

方法区溢出也是一种常见的内存溢出异常，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状况。这类场景除了上面提到的程序使用了CGLib字节码增强和动态语言之外，常见的还有：大量JSP或动态产生JSP文件的应用（JSP第一次运行时需要编译为Java类）、基于OSGi的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等。

### 本机直接内存溢出

由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了NIO，那就可以考虑检查一下是不是这方面的原因

## 垃圾回收器

Java内存运行时区域的各个部分，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的（尽管在运行期会由JIT编译器进行一些优化，但在本章基于概念模型的讨论中，大体上可以认为是编译期可知的），因此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，内存自然就跟随着回收了。而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法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

### 引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

但是它很难解决对象之间相互循环引用的问题。

举个简单的例子，请看代码清单3-1中的testGC（）方法：对象objA和objB都有字段

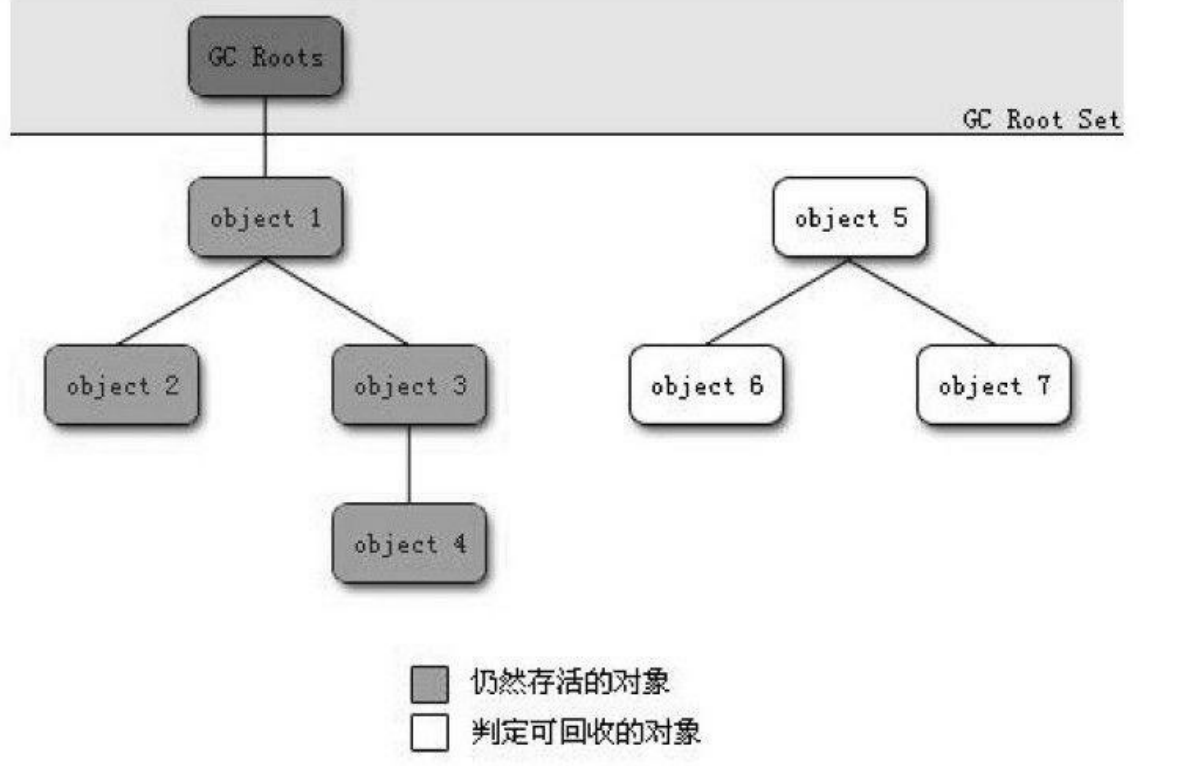
instance，赋值令objA.instance=objB及objB.instance=objA，除此之外，这两个对象再无任何引用，实际上这两个对象已经不可能再被访问，但是它们因为互相引用着对方，导致它们的引用计数都不为0，于是引用计数算法无法通知GC收集器回收它们。

### 可达性分析算法

在主流的商用程序语言（Java、C#，甚至包括前面提到的古老的Lisp）的主流实现中，

都是称通过可达性分析（Reachability Analysis）来判定对象是否存活的。

这个算法的基本思路就是通过一系列的称为“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（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。



在Java语言中，可作为GC Roots的对象包括下面几种：

虚拟机栈（栈帧中的本地变量表）中引用的对象。

方法区中类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（即一般说的Native方法）引用的对象。

### 引用

在JDK 1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（StrongReference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（PhantomReference）4种，这4种引用强度依次逐渐减弱。

1. 强引用就是指在程序代码之中普遍存在的，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。

1. 软引用是用来描述一些还有用但并非必需的对象。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JDK 1.2之后，提供了SoftReference类来实现软引用。

1. 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JDK 1.2之后，提供了WeakReference类来实现弱引用。

1. 虚引用也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

JDK 1.2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“缓刑”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相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛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的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，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（更极端的情况），将很可能会导致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（this关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了。

上面关于对象死亡时finalize（）方法的描述可能带有悲情的艺术色彩，笔者并不鼓励大家使用这种方法来拯救对象。相反，笔者建议大家尽量避免使用它，因为它不是C/C++中的析构函数，而是Java刚诞生时为了使C/C++程序员更容易接受它所做出的一个妥协。它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。有些教材中描述它适合做“关闭外部资源”之类的工作，这完全是对这个方法用途的一种自我安慰。finalize（）能做的所有工作，使用try-finally或者其他方式都可以做得更好、更及时，所以笔者建议大家完全可以忘掉Java语言中有这个方法的存在。

### 回收方法区

很多人认为方法区（或者HotSpot虚拟机中的永久代）是没有垃圾收集的，Java虚拟机规

范中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，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个String对象是叫做“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（接口）、方法、字段的符号引用也与此类似。

判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”：

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。

虚拟机可以对满足上述3个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是

和对象一样，不使用了就必然会回收。是否对类进行回收，HotSpot虚拟机提供了-Xnoclassgc

参数进行控制，还可以使用-verbose：class以及-XX：+TraceClassLoading、-XX：

+TraceClassUnLoading查看类加载和卸载信息，其中-verbose：class和-XX：

+TraceClassLoading可以在Product版的虚拟机中使用，-XX：+TraceClassUnLoading参数需要

FastDebug版的虚拟机支持。

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的功能，以保证永久代不会溢出。

### 垃圾收集算法

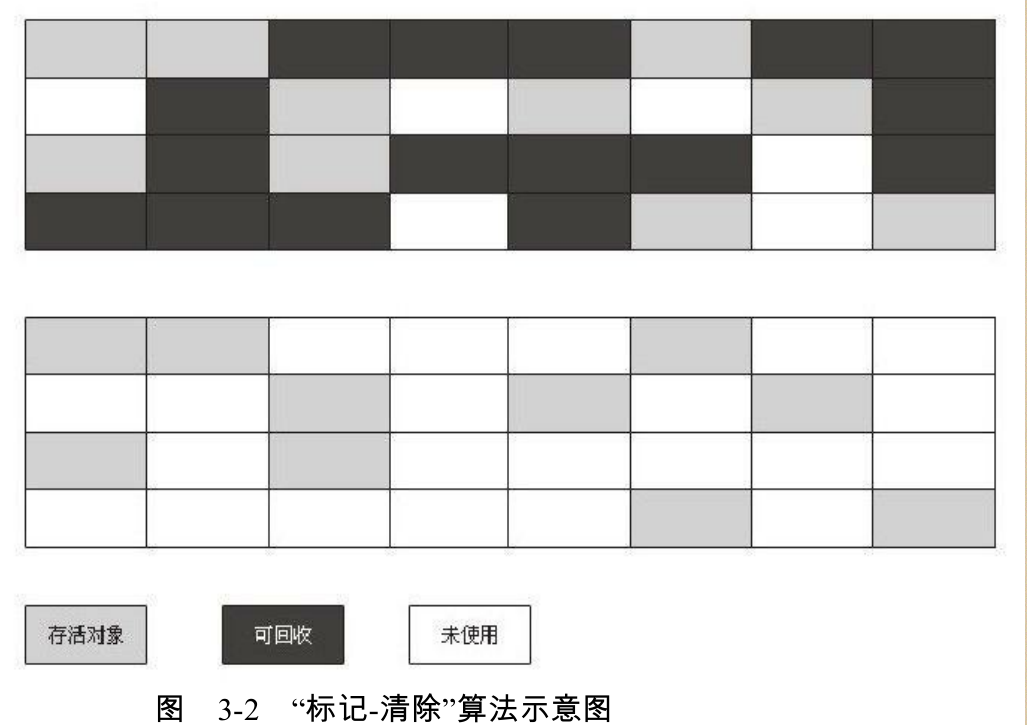
#### 标记-清除算法

算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经介绍过了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。

缺点：

1. 效率问题，标记和清除两个过程的效率都不高；
2. 空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中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另一次垃圾收集动作。

示意图如下：

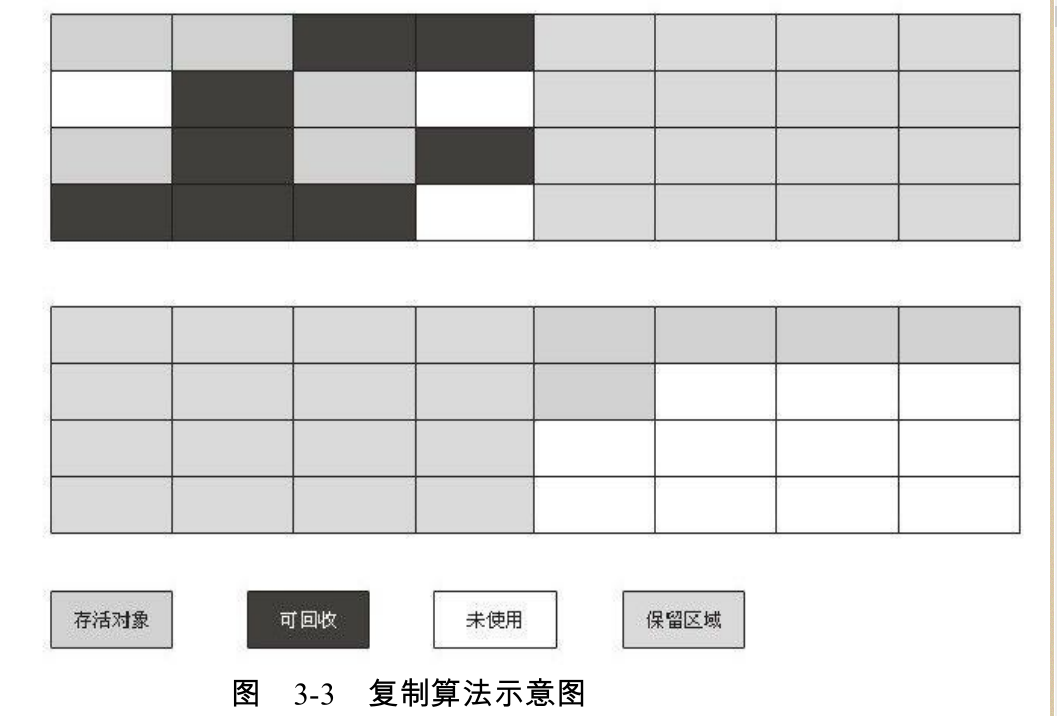


#### 复制算法

解决了效率问题，空间换性能。

将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

示意图如下：



现在的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生

代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor [1] 。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块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和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，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，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

如果另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。

#### 标记-整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的

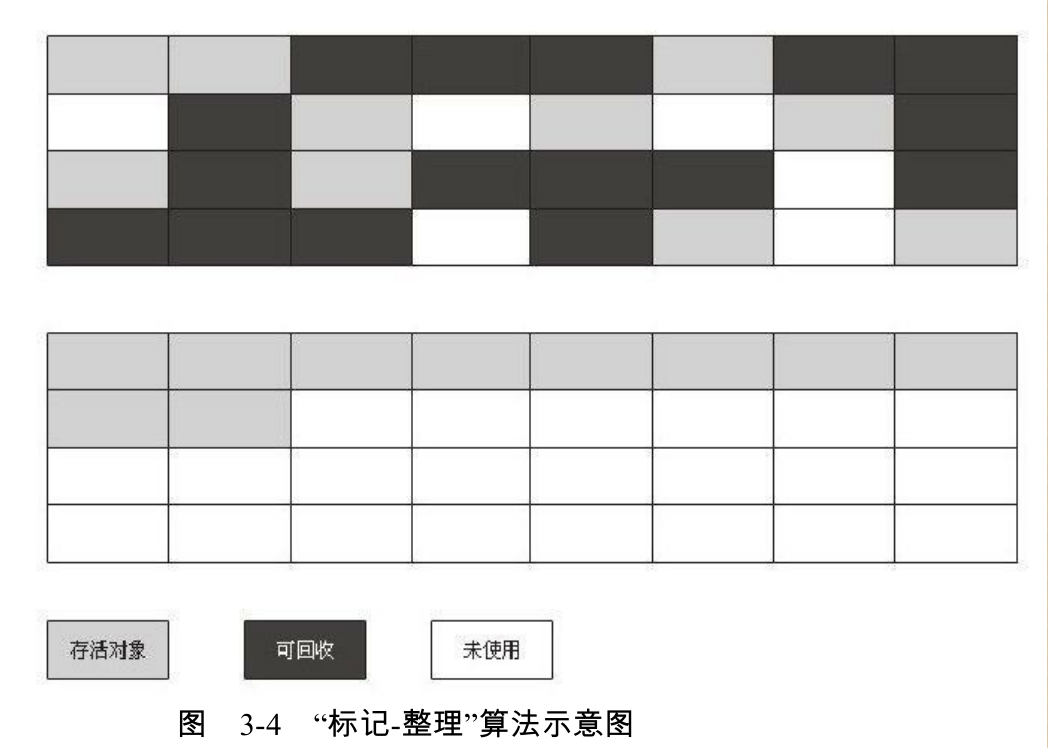
是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中

所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程

仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

示意图如下：



#### 分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算

法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代

中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付

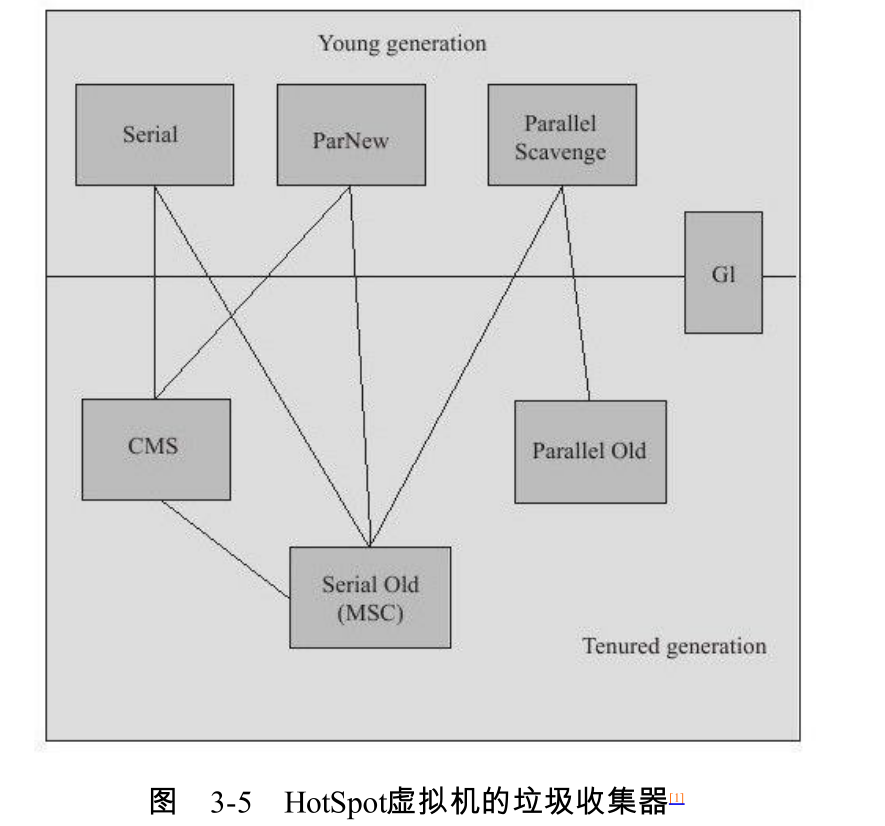
出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间

对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

### 垃圾收集器

收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

hotspot中的垃圾收集器示意图如下：



如上图所示，包括7种垃圾收集器，其中新生代中的收集器包括Serial,ParNew,Parallel Scavenge,GI,老年代中的垃圾收集器包括CMS,Serial Old(MSC)，Parallel Old和GI。

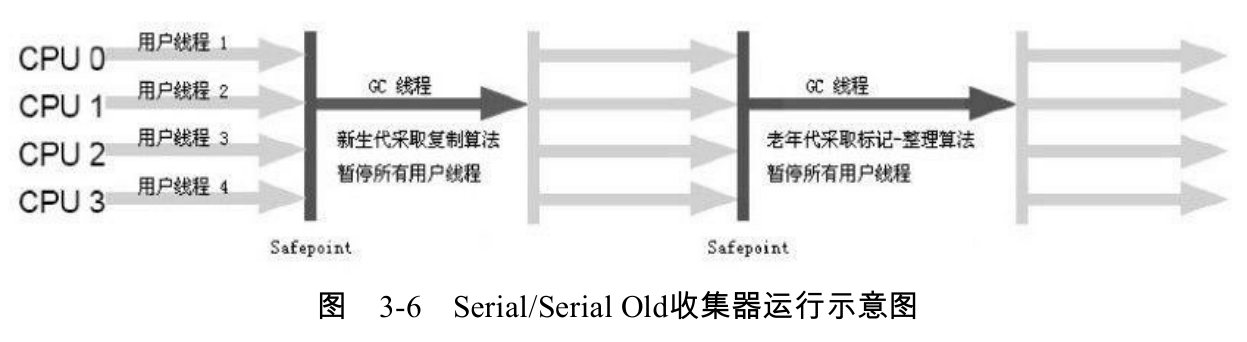
#### Serial收集器

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机

新生代收集的唯一选择。

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

示意图如下：



从JDK 1.3开始，一直到现在最新的JDK 1.7，HotSpot虚拟机开发团队为消除或者减少工

作线程因内存回收而导致停顿的努力一直在进行着，从Serial收集器到Parallel收集器，再到Concurrent Mark Sweep（CMS）乃至GC收集器的最前沿成果Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除（这里暂不包括RTSJ中的收集器）。寻找更优秀的垃圾收集器的工作仍在继续！

实际上到现在为止，它依然是虚拟机运行在Client模式下的默认新生代收集器。

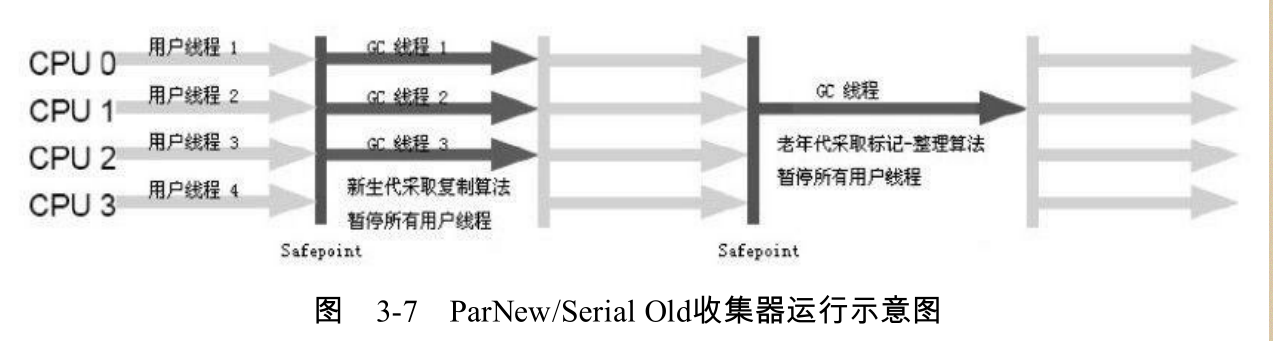
优点：简单而高效（与其他收集器的单线程比）。

对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户的桌面应用场景中，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（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，Serial收集器对于运行在Client模式下的虚拟机来说是一个很好的选择。

#### 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之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX：SurvivorRatio、-XX：PretenureSizeThreshold、-XX：HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew收集器的工作过程如图3-7所示。



ParNew收集器除了多线程收集之外，其他与Serial收集器相比并没有太多创新之处，但

它却是许多运行在Server模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。

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，甚至由于存在

线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。当然，随着可以使用的CPU的数量的增加，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的数量相同，在CPU非常多（譬如32个，现在CPU动辄就4核加超线程，服务器超过32个逻辑CPU的情况越来越多了）的环境下，可以使用-XX：ParallelGCThreads参数来限制垃圾收集的线程数。

注意,从ParNew收集器开始，后面还会接触到几款并发和并行的收集器。在大家可能

产生疑惑之前，有必要先解释两个名词：并发和并行。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，它们可以解释如下。

●并行（Parallel）：指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。

●并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，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。

#### Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行

的多线程收集器。“吞吐量优先”收集器。

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点

是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是CPU用于运行用户代码的时间与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

拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

停顿时间越短就越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，而高

吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集

停顿时间的-XX：MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX：GCTimeRatio参

数。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回

收花费的时间不超过设定值。不过大家不要认为如果把这个参数的值设置得稍小一点就能使

得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取

的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾

收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每

次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时

间的比率，相当于是吞吐量的倒数。如果把此参数设置为19，那允许的最大GC时间就占总

时间的5%（即1/（1+19）），默认值为99，就是允许最大1%（即1/（1+99））的垃圾收集

时间。

由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。除上

述两个参数之外，Parallel Scavenge收集器还有一个参数-XX：+UseAdaptiveSizePolicy值得关

注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、

Eden与Survivor区的比例（-XX：SurvivorRatio）、晋升老年代对象年龄（-XX：PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信

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，这种调节方式称为GC

自适应的调节策略（GC Ergonomics） [1] 。如果读者对于收集器运作原来不太了解，手工优化存在困难的时候，使用Parallel Scavenge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。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

#### Serial Old收集器

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整

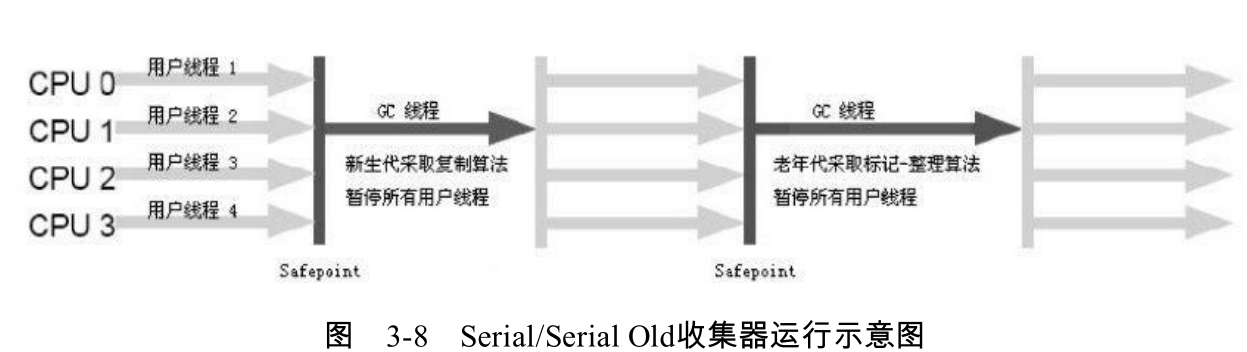
理”算法。

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。如果在Server模式

下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge

收集器搭配使用 [1] ，另一种用途就是作为CMS收集器的后备预案，在并发收集发生ConcurrentMode Failure时使用。

Serial Old收集器的工作过程如图所示：



需要说明一下，Parallel Scavenge收集器架构中本身有PS MarkSweep收集器来进行老年代

收集，并非直接使用了Serial Old收集器，但是这个PS MarkSweep收集器与Serial Old的实现非常接近，所以在官方的许多资料中都是直接以Serial Old代替PS MarkSweep进行讲解，这里笔者也采用这种方式。

#### Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

这个收集器是在JDK 1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直

处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了

Serial Old（PS MarkSweep）收集器外别无选择（还记得上面说过Parallel Scavenge收集器无

法与CMS收集器配合工作吗？）。由于老年代Serial Old收集器在服务端应用性能上的“拖

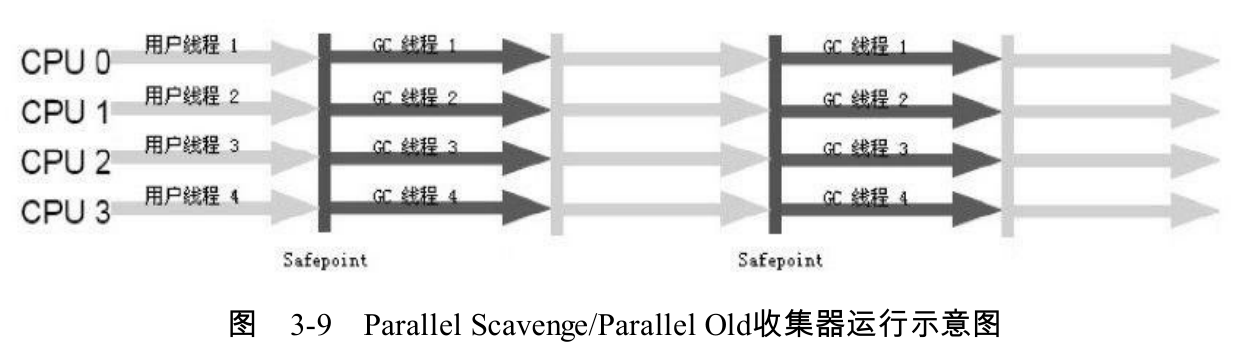
累”，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于

单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合“给力”。

直到Parallel Old收集器出现后，“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 Scavenge加Parallel Old

收集器。Parallel Old收集器的工作过程如图3-9所示。



#### CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集

器。

目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重

视服务的响应速度，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。

从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实现

的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是

标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC RootsTracing

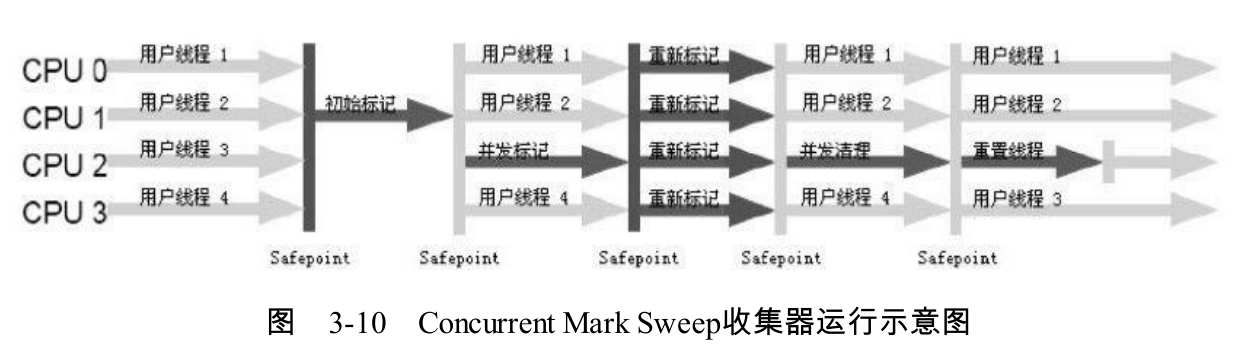
的过程，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变

动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远

比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起

工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。通过图3-10可以比较清楚地看到CMS收集器的运作步骤中并发和需要停顿的时间。



CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停

顿，Sun公司的一些官方文档中也称之为并发低停顿收集器（Concurrent Low Pause

Collector）。但是CMS还远达不到完美的程度，它有以下3个明显的缺点：

1. CMS收集器对CPU资源非常敏感。

其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用 户线程停顿，但是会因为占用了一部分线程（或者说 CPU资源）而导致应用程序变慢， 总吞吐量会降低。CMS默认启动的回收线程数是（CPU数量+3）/4，也就是当CPU在4 个以上时，并发回收时垃圾收集线程不少于25%的CPU资 源，并且随着CPU数量的 增加而下降。但是当CPU不足4个（譬如2个）时，CMS对 用户程序的影响就可 能变得很大，如果本来CPU负载就比较大，还分出一半的运算能 力去执行收集器 线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无 法接受。为了 应付这种情况，虚拟机提供了一种称为“增量式并发收集器”（Incremental Concurrent Mark Sweep/i-CMS）的CMS收集器变种，所做的事情和单CPU年代PC机操 作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记、清理的时候让GC 线程、用户线程交替运行，尽量减少GC线程的独占资源的时间，这样整个垃圾收集的 过程会更长，但对用户程序的影响就会显得少一些，也就是速度下降没有那么明显。实 践证明，增量时的CMS收集器效果很一般，在目前版本中，i-CMS已经被声明为 “deprecated”，即不再提倡用户使用。

1. CMS收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode

Failure”失败而导致另一次Full GC的产生。

由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾 不断产生，这一部分垃圾出现在标记过程之后， CMS无法在当次收集中处理掉它们， 只好留待下一次GC时再清理掉。这一部分垃圾就 称为“浮动垃圾”。

也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间

给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满 了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK 1.5的 默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守 的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX： CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获 取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运 行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败， 这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集， 这样停顿时间就很长了。所以说参数-XX：CMSInitiatingOccupancyFraction设置得太高很 容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

1. 会产生空间碎片

CMS是一款基于“标记—清除”算法实现的收集器，如果读者对前面这种算法介绍还 有印象的话，就可能想到这意味着收集结束时会有大量空间碎片产生。空间碎片过 多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无 法找到足够大的连续空间来分配当前对象，不得不提前触发一次FullGC。为了解决这个 问题，CMS收集器提供了一个-XX：+UseCMSCompactAtFullCollection开关参数（默认就 是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程， 内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。虚拟机 设计者还提供了另外一个参数-XX：CMSFullGCsBeforeCompaction，这个参数是用于设置 执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值为0，表示每次进入 FullGC时都进行碎片整理）。

#### 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。

G1是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是（在比较长

期的）未来可以替换掉JDK 1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点。

1. 并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。
2. 分代收集：与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。
3. 空间整合：与CMS的“标记—清理”算法不同，G1从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。
4. 可预测的停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

在G1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堆划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（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java

堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），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（这也就是Garbage-First名称的来由）。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。

G1把内存“化整为零”的思路，理解起来似乎很容易，但其中的实现细节却远远没有想象

中那样简单，否则也不会从2004年Sun实验室发表第一篇G1的论文开始直到今天（将近10年时间）才开发出G1的商用版。笔者以一个细节为例：把Java堆分为多个Region后，垃圾收集是否就真的能以Region为单位进行了？听起来顺理成章，再仔细想想就很容易发现问题所在：Region不可能是孤立的。一个对象分配在某个Region中，它并非只能被本Region中的其他对象引用，而是可以与整个Java堆任意的对象发生引用关系。那在做可达性判定确定对象是否存活的时候，岂不是还得扫描整个Java堆才能保证准确性？这个问题其实并非在G1中才有，只是在G1中更加突出而已。在以前的分代收集中，新生代的规模一般都比老年代要小许多，新生代的收集也比老年代要频繁许多，那回收新生代中的对象时也面临相同的问题，如果回收新生代时也不得不同时扫描老年代的话，那么Minor GC的效率可能下降不少。在G1收集器中，Region之间的对象引用以及其他收集器中的新生代与老年代之间的对象引用，虚拟机都是使用Remembered Set来避免全堆扫描的。G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在对Reference类型的数据进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set之中。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏。

如果不计算维护Remembered Set的操作，G1收集器的运作大致可划分为以下几个步骤：

初始标记（Initial Marking）

并发标记（Concurrent Marking）

最终标记（Final Marking）

筛选回收（Live Data Counting and Evacuation）

1. 初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改

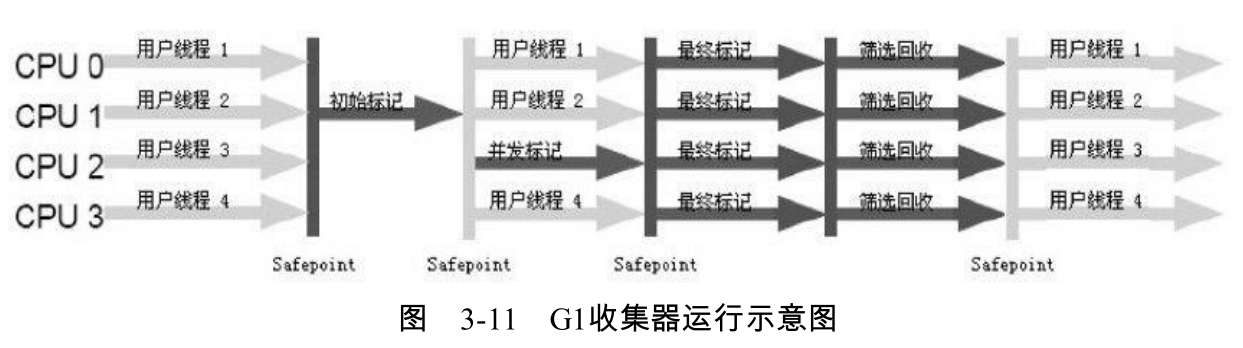
TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用 的Region中创建新对象，这阶段需要停顿线程，但耗时很短。

1. 并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。
2. 最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动

的那一部分标记记录。

1. 筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。

G1收集器运行示意图：



### 理解GC日志

每一种收集器的日志形式都是由它们自身的实现所决定的，换而言之，每个收集器的日

志格式都可以不一样。但虚拟机设计者为了方便用户阅读，将各个收集器的日志都维持一定

的共性，例如以下两段典型的GC日志：

33.125：[GC[DefNew：3324K-＞152K（3712K），0.0025925 secs]3324K-＞152K（11904K），0.0031680 secs]

100 .667：[F u l l G C[T e n u r e d：0 K-＞2 1 0 K（1 0 2 4 0 K），0.0 1 4 9 1 4 2 s e c s]4603K-＞210K（19456K），[Perm：2999K-＞2999K（21248K）]，0.0150007 secs][Times：user=0.01 sys=0.00，real=0.02 secs]

最前面的数字“33.125：”和“100.667：”代表了GC发生的时间，这个数字的含义是从Java

虚拟机启动以来经过的秒数。

GC日志开头的“[GC”和“[Full GC”说明了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”，说明这次GC是发生了Stop-The-World的，例如下面这段新生代收集器ParNew的日志也会出现“[Full GC”（这一般是因为出现了分配担保失败之类的问题，所以才导致STW）。如果是调用System.gc（）方法所触发的收集，那么在这里将显示“[Full GC（System）”。

[Full GC 283.736：[ParNew：261599K-＞261599K（261952K），0.0000288 secs]

接下来的“[DefNew”、“[Tenured”、“[Perm”表示GC发生的区域，这里显示的区域名称与

使用的GC收集器是密切相关的，例如上面样例所使用的Serial收集器中的新生代名为“DefaultNew Generation”，所以显示的是“[DefNew”。如果是ParNew收集器，新生代名称就会变为“[ParNew”，意为“Parallel New Generation”。如果采用Parallel Scavenge收集器，那它配套的新生代称为“PSYoungGen”，老年代和永久代同理，名称也是由收集器决定的。

后面方括号内部的“3324K-＞152K（3712K）”含义是“GC前该内存区域已使用容量-＞

GC后该内存区域已使用容量（该内存区域总容量）”。而在方括号之外的“3324K-＞

152K（11904K）”表示“GC前Java堆已使用容量-＞GC后Java堆已使用容量（Java堆总容

量）”。

再往后，“0.0025925 secs”表示该内存区域GC所占用的时间，单位是秒。有的收集器会

给出更具体的时间数据，如“[Times：user=0.01 sys=0.00，real=0.02 secs]”，这里面的user、

sys和real与Linux的time命令所输出的时间含义一致，分别代表用户态消耗的CPU时间、内核态消耗的CPU事件和操作从开始到结束所经过的墙钟时间（Wall Clock Time）。CPU时间与墙钟时间的区别是，墙钟时间包括各种非运算的等待耗时，例如等待磁盘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，所以读者看到user或sys时间超过real时间是完全正常的。

#### 垃圾收集器参数总结





### 内存分配与回收策略

对象的内存分配，就是在堆上分配（但也可能经过JIT编译后被拆散为标

量类型并间接地栈上分配 [1] ），对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB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，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

接下来的代码在测试时使用Client模式虚拟机运行，没有手工指定收集器组合，换句话说，验证的是在使用Serial/Serial Old收集器下（ParNew/Serial Old收集器组合的规则也基本一致）的内存分配和回收的策略。

#### 对象优先在新生代Eden分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟

机将发起一次Minor GC。虚拟机提供了-XX：+PrintGCDetails这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前的内存各区域分配情况。在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看得很清楚。

新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝

生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。

老年代GC（Major GC/Full GC）：指发生在老年代的GC，出现了Major GC，经常会伴

随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行

Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

新生代Minor GC示例：

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

/\*\*

\* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8

\*

\*/

public static void testAllocation() {

byte[] allocation1, allocation2, allocation3, allocation4;

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

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

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

allocation4 = new byte[4 \* \_1MB];// 出现一次Minor GC

｝

运行结果:

Heap

PSYoungGen total 9216K, used 7456K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)

eden space 8192K, 91% used [0x00000000ff600000,0x00000000ffd48328,0x00000000ffe00000)

from space 1024K, 0% used [0x00000000fff00000,0x00000000fff00000,0x0000000100000000)

to space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)

ParOldGen total 10240K, used 4096K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)

object space 10240K, 40% used [0x00000000fec00000,0x00000000ff000010,0x00000000ff600000)

Metaspace used 2696K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 288K, capacity 386K, committed 512K, reserved 1048576K

尝试分配3个2MB大小和1个4MB大小的对象，

在运行时通过-Xms20M、-Xmx20M、-Xmn10M这3个参数限制了Java堆大小为20MB，不可扩展，其中10MB分配给新生代，剩下的10MB分配给老年代。-XX：SurvivorRatio=8决定了新生代中Eden区与一个Survivor区的空间比例是8:1，从输出的结果也可以清晰地看到“edenspace 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为9216KB（Eden区+1个Survivor区的总容量）。

#### 大对象直接进入老年代

所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长

的字符串以及数组（笔者列出的例子中的byte[]数组就是典型的大对象）。大对象对虚拟机

的内存分配来说就是一个坏消息（替Java虚拟机抱怨一句，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个-XX：PretenureSizeThreshold参数，令大于这个设置值的对象直接在老

年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制（复习一下：新生代采用复制算法收集内存）。

#### 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象

应放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对

象年龄（Age）计数器。如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就将会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX：

MaxTenuringThreshold设置。

代码示例：

/\*\*

\* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio

\* =8 -XX:MaxTenuringThreshold=1 -XX:+PrintTenuringDistribution

\* 长期存活的对象会进入老年代

\*/

public static void testTenuringThreshold() {

byte[] allocation1, allocation2, allocation3;

allocation1 = new byte[\_1MB / 4];

// 什么时候进入老年代取决于XX：MaxTenuringThreshold设置

allocation2 = new byte[4 \* \_1MB];

allocation3 = new byte[4 \* \_1MB];

allocation3 = null;

allocation3 = new byte[4 \* \_1MB];

}

如下是运行结果：

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 1（max 1）

-age 1：414664 bytes，414664 total

：4859K-＞404K（9216K），0.0065012 secs]4859K-＞4500K（19456K），0.0065283 secs][Times：user=0.02 sys=0.00，real=0.02 secs]

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 1（max 1）

：4500K-＞0K（9216K），0.0009253 secs]8596K-＞4500K（19456K），0.0009458 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

Heap

def new generation total 9216K,used 4178K[0x029d0000，0x033d0000，0x033d0000）

eden space 8192K，51%used[0x029d0000，0x02de4828，0x031d0000）

from space 1024K，0%used[0x031d0000，0x031d0000，0x032d0000）

to space 1024K，0%used[0x032d0000，0x032d0000，0x033d0000）

tenured generation total 10240K,used 4500K[0x033d0000，0x03dd0000，0x03dd0000）

the space 10240K，43%used[0x033d0000，0x03835348，0x03835400，0x03dd0000）

compacting perm gen total 12288K,used 2114K[0x03dd0000，0x049d0000，0x07dd0000）

the space 12288K，17%used[0x03dd0000，0x03fe0998，0x03fe0a00，0x049d0000）

No shared spaces configured.

以MaxTenuringThreshold=15参数来运行的结果：

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 15（max 15）

-age 1：414664 bytes，414664 total

：4859K-＞404K（9216K），0.0049637 secs]4859K-＞4500K（19456K），0.0049932 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 15（max 15）

-age 2：414520 bytes，414520 total

：4500K-＞404K（9216K），0.0008091 secs]8596K-＞4500K（19456K），0.0008305 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

Heap

def new generation total 9216K,used 4582K[0x029d0000，0x033d0000，0x033d0000）

eden space 8192K，51%used[0x029d0000，0x02de4828，0x031d0000）

from space 1024K，39%used[0x031d0000，0x03235338，0x032d0000）

to space 1024K，0%used[0x032d0000，0x032d0000，0x033d0000）

tenured generation total 10240K,used 4096K[0x033d0000，0x03dd0000，0x03dd0000）

the space 10240K，40%used[0x033d0000，0x037d0010，0x037d0200，0x03dd0000）

compacting perm gen total 12288K,used 2114K[0x03dd0000，0x049d0000，0x07dd0000）

the space 12288K，17%used[0x03dd0000，0x03fe0998，0x03fe0a00，0x049d0000）

No shared spaces configured.

#### 动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到

了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于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MaxTenuringThreshold中要求的年龄。

执行代码清单3-8中的testTenuringThreshold2（）方法，并设置-XX：MaxTenuringThreshold=15，会发现运行结果中Survivor的空间占用仍然为0%，而老年代比预期增加了6%，也就是说，allocation1、allocation2对象都直接进入了老年代，而没有等到15岁的临界年龄。因为这两个对象加起来已经到达了512KB，并且它们是同年的，满足同年对象达到Survivor空间的一半规则。我们只要注释掉其中一个对象new操作，就会发现另外一个就不会晋升到老年代中去了。

运行结果如下：

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 1（max 15）

-age 1：676824 bytes，676824 total

：5115K-＞660K（9216K），0.0050136 secs]5115K-＞4756K（19456K），0.0050443 secs][Times：user=0.00 sys=0.01，real=0.01 secs]

[GC[DefNew

Desired Survivor size 524288 bytes,new threshold 15（max 15）

：4756K-＞0K（9216K），0.0010571 secs]8852K-＞4756K（19456K），0.0011009 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

Heap

def new generation total 9216K,used 4178K[0x029d0000，0x033d0000，0x033d0000）

eden space 8192K，51%used[0x029d0000，0x02de4828，0x031d0000）

from space 1024K，0%used[0x031d0000，0x031d0000，0x032d0000）

to space 1024K，0%used[0x032d0000，0x032d0000，0x033d0000）

tenured generation total 10240K,used 4756K[0x033d0000，0x03dd0000，0x03dd0000）

the space 10240K，46%used[0x033d0000，0x038753e8，0x03875400，0x03dd0000）

compacting perm gen total 12288K,used 2114K[0x03dd0000，0x049d0000，0x07dd0000）

the space 12288K，17%used[0x03dd0000，0x03fe09a0，0x03fe0a00，0x049d0000）

No shared spaces configured.

#### 空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有

对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

下面解释一下“冒险”是冒了什么风险，前面提到过，新生代使用复制收集算法，但为了

内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor

GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要

老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。与生活中的贷款担保类

似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多

少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋

升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进

行Full GC来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说，如果某次Minor GC存活

后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。

如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full GC过于频繁。

以HandlePromotionFailure=false参数来运行的结果：

[GC[DefNew：6651K-＞148K（9216K），0.0078936 secs]6651K-＞4244K（19456K），0.0079192 secs][Times：user=0.00 sys=0.02，real=0.02 secs]

[G C[D e f N e w：6 3 7 8 K-＞6 3 7 8 K（9 2 1 6 K），0.0 0 0 0 2 0 6 s e c s][T e n u r e d：4096K-＞4244K（10240K），0.0042901 secs]10474K-＞

4244K（19456K），[Perm：2104K-＞2104K（12288K）]，0.0043613 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

以HandlePromotionFailure=true参数来运行的结果：

[GC[DefNew：6651K-＞148K（9216K），0.0054913 secs]6651K-＞4244K（19456K），0.0055327 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

[GC[DefNew：6378K-＞148K（9216K），0.0006584 secs]10474K-＞4244K（19456K），0.0006857 secs][Times：user=0.00 sys=0.00，real=0.00 secs]

## 虚拟机性能监控与故障处理工具

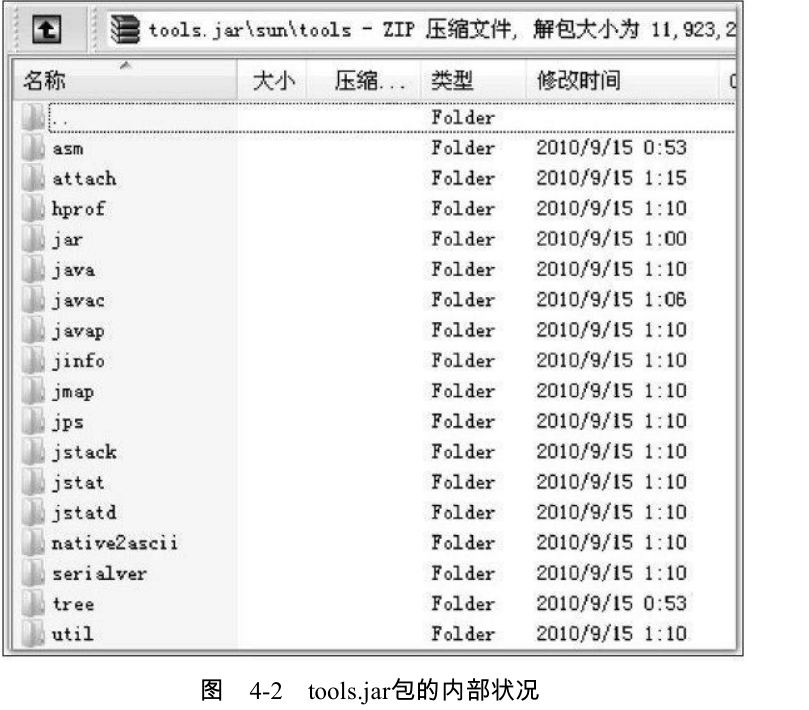
给一个系统定位问题的时候，知识、经验是关键基础，数据是依据，工具是运用知识处

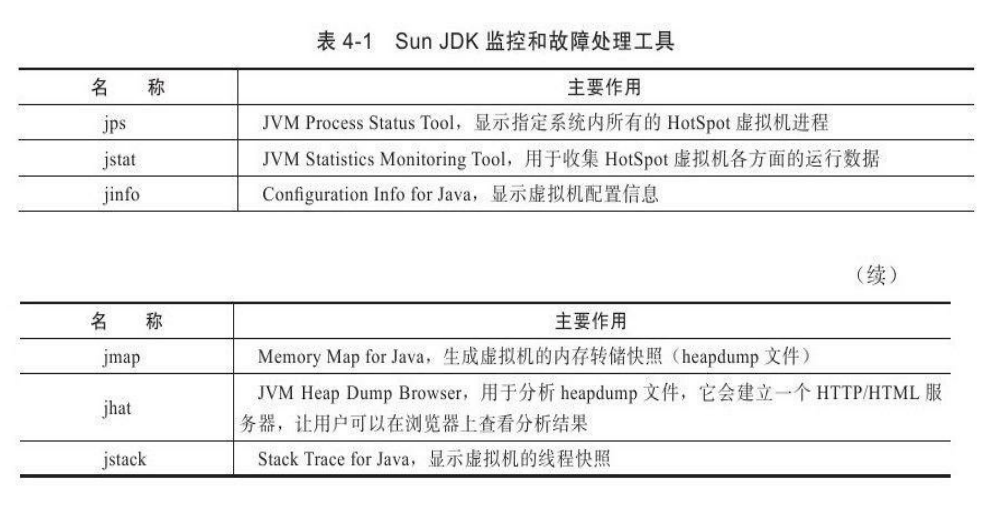
理数据的手段。这里说的数据包括：运行日志、异常堆栈、GC日志、线程快照

（threaddump/javacore文件）、堆转储快照（heapdump/hprof文件）等。经常使用适当的虚拟机监控和分析的工具可以加快我们分析数据、定位解决问题的速度。

### JDK的命令行工具







### jps：虚拟机进程状况工具

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

Tool）是其中的典型。除了名字像UNIX的ps命令之外，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（Main Class,main（）函数所在的类）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，但它是使用频率最高的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。对于本地虚拟机进程来说，LVMID与操作系统的进程ID（Process Identifier,PID）是一致的，使用Windows的任务管理器或者UNIX的ps命令也可以查询到虚拟机进程的LVMID，但如果同时启动了多个虚拟机进程，无法根据进程名称定位时，那就只能依赖jps命令显示主类的功能才能区分了。

jsp命令格式：jps[options][hostid]

示例：

D：\Develop\Java\jdk1.6.0\_21\bin＞jps -l

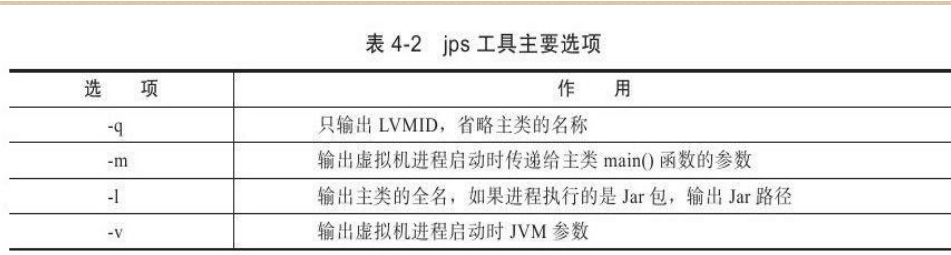
2388 D：\Develop\glassfish\bin\..\modules\admin-cli.jar

2764 com.sun.enterprise.glassfish.bootstrap.ASMain

3788 sun.tools.jps.Jps

jps可以通过RMI协议查询开启了RMI服务的远程虚拟机进程状态，hostid为RMI注册表中

注册的主机名。



### jstat：虚拟机统计信息监视工具。

jstat（JVM Statistics Monitoring Tool）是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 [1] 虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据，在没有GUI图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

jstat命令格式为：jstat[option vmid[interval[s|ms][count]]]

对于命令格式中的VMID与LVMID需要特别说明一下：如果是本地虚拟机进程，VMID与

LVMID是一致的，如果是远程虚拟机进程，那VMID的格式应当是：

[protocol：][//]lvmid[@hostname[：port]/servername]

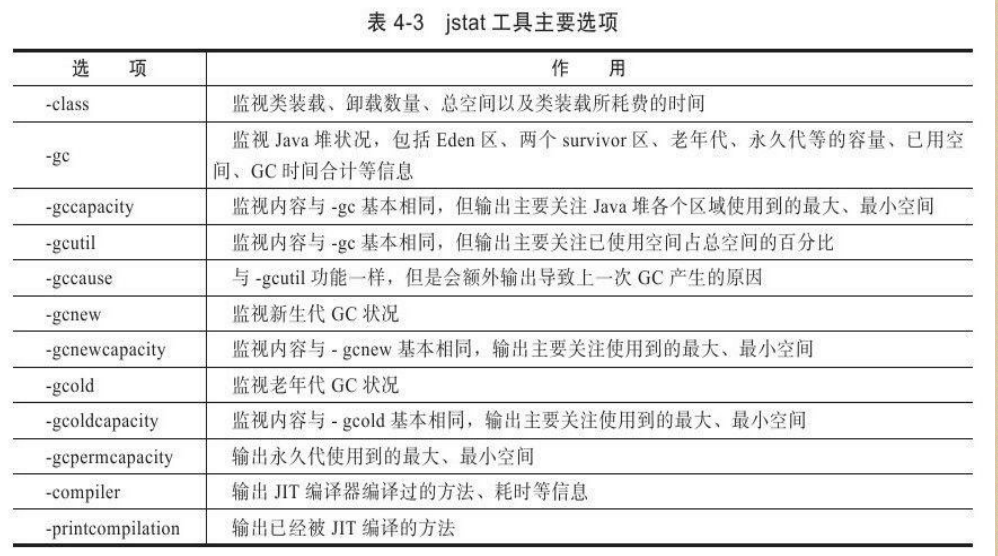
参数interval和count代表查询间隔和次数，如果省略这两个参数，说明只查询一次。假设

需要每250毫秒查询一次进程2764垃圾收集状况，一共查询20次，那命令应当是：

jstat -gc 2764 250 20

选项option代表着用户希望查询的虚拟机信息，主要分为3类：类装载、垃圾收集、运行

期编译状况，具体选项及作用请参考下表中的描述：



jstat监视选项众多，囿于版面原因无法逐一演示，这里仅举监视一台刚刚启动的

GlassFish v3服务器的内存状况的例子来演示如何查看监视结果。监视参数与输出结果如代码：

jstat 执行样例：

D：\Develop\Java\jdk1.6.0\_21\bin＞jstat -gcutil 2764

S0 S1 E O P YGC YGCT FGC FGCT GCT

0.00 0.00 6.20 41.42 47.20 16 0.105 3 0.472 0.577

查询结果表明：这台服务器的新生代Eden区（E，表示Eden）使用了6.2%的空间，两个

Survivor区（S0、S1，表示Survivor0、Survivor1）里面都是空的，老年代（O，表示Old）和

永久代（P，表示Permanent）则分别使用了41.42%和47.20%的空间。程序运行以来共发生

Minor GC（YGC，表示Young GC）16次，总耗时0.105秒，发生Full GC（FGC，表示Full

GC）3次，Full GC总耗时（FGCT，表示Full GC Time）为0.472秒，所有GC总耗时（GCT，

表示GC Time）为0.577秒。

使用jstat工具在纯文本状态下监视虚拟机状态的变化，确实不如后面将会提到的

VisualVM等可视化的监视工具直接以图表展现那样直观。但许多服务器管理员都习惯了在文本控制台中工作，直接在控制台中使用jstat命令依然是一种常用的监控方式。

### 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参数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，除了去找资料外，就只能使用jinfo的-flag选项进行查询了（如果只限于

JDK 1.6或以上版本的话，使用java-XX：+PrintFlagsFinal查看参数默认值也是一个很好的选

择），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。这个命令在JDK 1.5时期已经随着Linux版的JDK发布，当时只提供了信息查询的功

能，JDK 1.6之后，jinfo在Windows和Linux平台都有提供，并且加入了运行期修改参数的能

力，可以使用-flag[+|-]name或者-flag name=value修改一部分运行期可写的虚拟机参数值。

JDK 1.6中，jinfo对于Windows平台功能仍然有较大限制，只提供了最基本的-flag选项。

jinfo命令格式：jinfo[option]pid

执行样例：查询CMSInitiatingOccupancyFraction参数值。

C：\＞jinfo-flag CMSInitiatingOccupancyFraction 1444

-XX：CMSInitiatingOccupancyFraction=85

### jmap：Java内存映像工具

jmap（Memory Map for Java）命令用于生成堆转储快照（一般称为heapdump或dump文

件）。如果不使用jmap命令，要想获取Java堆转储快照，还有一些比较“暴力”的手段：譬如在第2章中用过的-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，通过-XX：+HeapDumpOnCtrlBreak参数则可以使用[Ctrl]+[Break]键让虚拟机生成dump文件，又或者在Linux系统下通过Kill-3命令发送进程退出信号“吓唬”一下虚拟机，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、当前用的是哪种收集器等。

和jinfo命令一样，jmap有不少功能在Windows平台下都是受限的，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/Solaris下使用。

jmap命令格式：jmap[option]vmid

option选项的合法值与具体含义见表。



使用jmap生成dump文件例子：

C：\Users\IcyFenix＞jmap-dump：format=b,file=eclipse.bin 3500

Dumping heap to C：\Users\IcyFenix\eclipse.bin……

Heap dump file created

### jhat：虚拟机堆转储快照分析工具

Sun JDK提供jhat（JVM Heap Analysis Tool）命令与jmap搭配使用，来分析jmap生成的堆

转储快照。jhat内置了一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以在浏览器中查看。不过实事求是地说，在实际工作中，除非笔者手上真的没有别的工具可用，

否则一般都不会去直接使用jhat命令来分析dump文件，主要原因有二：一是一般不会在部署应用程序的服务器上直接分析dump文件，即使可以这样做，也会尽量将dump文件复制到其他机器 [1] 上进行分析，因为分析工作是一个耗时而且消耗硬件资源的过程，既然都要在其他机器进行，就没有必要受到命令行工具的限制了；另一个原因是jhat的分析功能相对来说比较简陋，后文将会介绍到的VisualVM，以及专业用于分析dump文件的Eclipse MemoryAnalyzer、IBM HeapAnalyzer [2] 等工具，都能实现比jhat更强大更专业的分析功能。代码清单4-3演示了使用jhat分析4.2.4节中采用jmap生成的Eclipse IDE的内存快照文件。

使用jhat分析dump文件例子：

C：\Users\IcyFenix＞jhat eclipse.bin

Reading from eclipse.bin……

Dump file created Fri Nov 19 22：07：21 CST 2010

Snapshot read,resolving……

Resolving 1225951 objects……

Chasing references,expect 245 dots……

Eliminating duplicate references……

Snapshot resolved.

Started HTTP server on port 7000

Server is ready.

屏幕显示“Server is ready.”的提示后，用户在浏览器中键入http://localhost：7000/就可以

看到分析结果，如图所示：



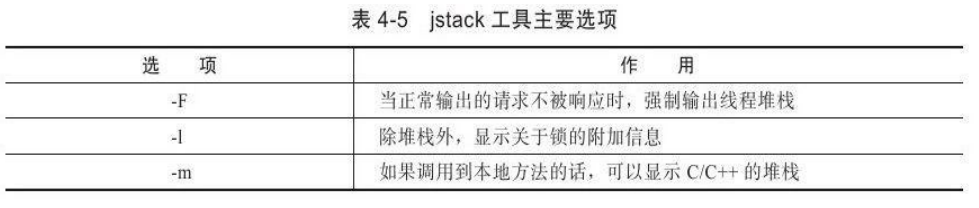
### jstack：Java堆栈跟踪工具

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照（一般称为

threaddump或者javacore文件）。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，生成线程快照的主要目的是定位线程出现长时间停顿的原因，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常见原因。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，或者等待着什么资源。

jstack命令格式：jstack[option]vmid

option选项的合法值与具体含义见表。



使用jstack查看Eclipse线程堆栈的例子，例子中的3500是通过jps命令查询到的LVMID。

C：\Users\IcyFenix＞jstack-l 3500

2010-11-19 23：11：26

Full thread dump Java HotSpot（TM）64-Bit Server VM（17.1-b03 mixed mode）：

"[ThreadPool Manager]-Idle Thread"daemon prio=6 tid=0x0000000039dd4000 nid=0xf50 in Object.wait（）[0x000000003c96f000]

java.lang.Thread.State：WAITING（on object monitor）

at java.lang.Object.wait（Native Method）

-waiting on＜0x0000000016bdcc60＞（a org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor）

at java.lang.Object.wait（Object.java：485）

at org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor.run（Executor.java：106）

-locked＜0x0000000016bdcc60＞（a org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor）

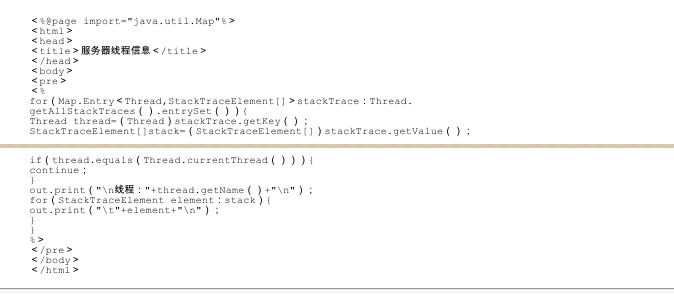
Locked ownable synchronizers：

-None

在JDK 1.5中，java.lang.Thread类新增了一个getAllStackTraces（）方法用于获取虚拟机

中所有线程的StackTraceElement对象。使用这个方法可以通过简单的几行代码就完成jstack的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈，如代码清单4-5所示，这是笔者的一个小经验。

查看线程状况的JSP页面：



### HSDIS：JIT生成代码反汇编

在Java虚拟机规范中，详细描述了虚拟机指令集中每条指令的执行过程、执行前后对操

作数栈、局部变量表的影响等细节。这些细节描述与Sun的早期虚拟机（Sun Classic VM）高度吻合，但随着技术的发展，高性能虚拟机真正的细节实现方式已经渐渐与虚拟机规范所描述的内容产生了越来越大的差距，虚拟机规范中的描述逐渐成了虚拟机实现的“概念模

型”——即实现只能保证规范描述等效。基于这个原因，我们分析程序的执行语义问题（虚

拟机做了什么）时，在字节码层面上分析完全可行，但分析程序的执行行为问题（虚拟机是

怎样做的、性能如何）时，在字节码层面上分析就没有什么意义了，需要通过其他方式解

决。分析程序如何执行，通过软件调试工具（GDB、Windbg等）来断点调试是最常见的手

段，但是这样的调试方式在Java虚拟机中会遇到很大困难，因为大量执行代码是通过JIT编译器动态生成到CodeBuffer中的，没有很简单的手段来处理这种混合模式的调试（不过相信虚拟机开发团队内部肯定是有内部工具的）。因此，不得不通过一些特别的手段来解决题，

基于这种背景，本节的主角——HSDIS插件就正式登场了

HSDIS是一个Sun官方推荐的HotSpot虚拟机JIT编译代码的反汇编插件，它包含在HotSpot

虚拟机的源码之中，但没有提供编译后的程序。在Project Kenai的网站 [1] 也可以下载到单独的源码。它的作用是让HotSpot的-XX：+PrintAssembly指令调用它来把动态生成的本地代码还原为汇编代码输出，同时还生成了大量非常有价值的注释，这样我们就可以通过输出的代码来分析问题。读者可以根据自己的操作系统和CPU类型从Project Kenai的网站上下载编译好的插件，直接放到JDK\_HOME/jre/bin/client和JDK\_HOME/jre/bin/server目录中即可。如果没有找到所需操作系统（譬如Windows的就没有）的成品，那就得自己使用源码编译一下 [2] 。还需要注意的是，如果读者使用的是Debug或者FastDebug版的HotSpot，那可以直接通过-XX：+PrintAssembly指令使用插件；如果使用的是Product版的HotSpot，那还要额外加入一个-XX：+UnlockDiagnosticVMOptions参数。笔者以代码清单4-6中的简单测试代码为例演示一下这个插件的使用。

测试代码：

public class Bar{

int a=1；

static int b=2；

public int sum（int c）{

return a+b+c；

}

public static void main（String[]args）{

new Bar（）.sum（3）；

}

}

编译这段代码，并使用以下命令执行。

java-XX：+PrintAssembly-Xcomp-XX：CompileCommand=dontinline，\*Bar.sum-XX：Compi leCommand=compileonly，\*Bar.sum test.Bar

其中，参数-Xcomp是让虚拟机以编译模式执行代码，这样代码可以“偷懒”，不需要执行

足够次数来预热就能触发JIT编译 [3] 。两个-XX：CompileCommand意思是让编译器不要内联sum（）并且只编译sum（），-XX：+PrintAssembly就是输出反汇编内容。如果一切顺利的话，那么屏幕上会出现类似下面所示的内容。

[Disassembling for mach='i386']

[Entry Point]

[Constants]

#{method}'sum''（I）I'in'test/Bar'

#this：ecx='test/Bar'

#parm0：edx=int

#[sp+0x20]（sp of caller）

……

0x01cac407：cmp 0x4（%ecx），%eax

0x01cac40a：jne 0x01c6b050；{runtime\_call}

[Verified Entry Point]

0x01cac410：mov%eax，-0x8000（%esp）

0x01cac417：push%ebp

0x01cac418：sub$0x18，%esp；\*aload\_0

；-test.Bar：sum@0（line 8）

；block B0[0，10]

0x01cac41b：mov 0x8（%ecx），%eax；\*getfield a

；-test.Bar：sum@1（line 8）

0x01cac41e：mov$0x3d2fad8，%esi；{oop（a

'java/lang/Class'='test/Bar'）}

0x01cac423：mov 0x68（%esi），%esi；\*getstatic b

；-test.Bar：sum@4（line 8）

0x01cac426：add%esi，%eax

0x01cac428：add%edx，%eax

0x01cac42a：add$0x18，%esp

0x01cac42d：pop%ebp

0x01cac42e：test%eax，0x2b0100；{poll\_return}

0x01cac434：ret

上段代码并不多，下面一句句进行说明。

1）mov%eax，-0x8000（%esp）：检查栈溢。

2）push%ebp：保存上一栈帧基址。

3）sub$0x18，%esp：给新帧分配空间。

4）mov 0x8（%ecx），%eax：取实例变量a，这里0x8（%ecx）就是ecx+0x8的意思，前

面“[Constants]”节中提示了“this：ecx='test/Bar'”，即ecx寄存器中放的就是this对象的地址。偏

移0x8是越过this对象的对象头，之后就是实例变量a的内存位置。这次是访问“Java堆”中的数

据。

5）mov$0x3d2fad8，%esi：取test.Bar在方法区的指针。

6）mov 0x68（%esi），%esi：取类变量b，这次是访问“方法区”中的数据。

7）add%esi，%eax和add%edx，%eax：做两次加法，求a+b+c的值，前面的代码把a放在

eax中，把b放在esi中，而c在[Constants]中提示了，“parm0：edx=int”，说明c在edx中。

8）add$0x18，%esp：撤销栈帧。

9）pop%ebp：恢复上一栈帧。

10）test%eax，0x2b0100：轮询方法返回处的SafePoint。

11）ret：方法返回。

[1]Project Kenai：http://kenai.com/projects/base-hsdis。

[2]HLLVM圈子中有已编译好的：http://hllvm.group.iteye.com/。

[3]-Xcomp在较新的HotSpot中被移除了，如果读者的虚拟机无法使用这个参数，请加个循环

预热代码，触发JIT编译。

## JDK可视化工具

JDK中除了提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole和VisualVM，这两个工具是JDK的正式成员，没有被贴上“unsupported and experimental”的标签。

### JConsole：Java监视与管理控制台

JConsole（Java Monitoring and Management Console）是一种基于JMX的可视化监视、管

理工具。它管理部分的功能是针对JMX MBean进行管理，由于MBean可以使用代码、中间件服务器的管理控制台或者所有符合JMX规范的软件进行访问，所以本节将会着重介绍

JConsole监视部分的功能.

详细使用见书中示例，重要！

### VisualVM：多合一故障处理工具

VisualVM（All-in-One Java Troubleshooting Tool）是到目前为止随JDK发布的功能最强大

的运行监视和故障处理程序，并且可以预见在未来一段时间内都是官方主力发展的虚拟机故

障处理工具。官方在VisualVM的软件说明中写上了“All-in-One”的描述字样，预示着它除了运行监视、故障处理外，还提供了很多其他方面的功能。如性能分析

（Profiling），VisualVM的性能分析功能甚至比起JProfiler、YourKit等专业且收费的Profiling

工具都不会逊色多少，而且VisualVM的还有一个很大的优点：不需要被监视的程序基于特殊Agent运行，因此它对应用程序的实际性能的影响很小，使得它可以直接应用在生产环境

中。这个优点是JProfiler、YourKit等工具无法与之媲美的。

详细使用见书中示例，重要！

### 其它

除了JDK自带的工具之外，常用的故障处理工具还有很多，如果读者使用的是非Sun系

列的JDK、非HotSpot的虚拟机，就需要使用对应的工具进行分析，如：

IBM的Support Assistant [1] 、Heap Analyzer [2] 、Javacore Analyzer [3] 、Garbage Collector

Analyzer [4] 适用于IBM J9 VM。

HP的HPjmeter [5] 、HPjtune适用于HP-UX、SAP、HotSpot VM。

Eclipse的Memory Analyzer Tool

[6] （MAT）适用于HP-UX、SAP、HotSpot VM，安装IBM

DTFJ插件后可支持IBM J9 VM。

BEA的JRockit Mission Control [7] 适用于JRockit VM。

## 调优案例分析与实战

见书中第5章分析，重要！

# 思考

线程局部变量是存放在线程的虚拟机栈中吗？

局部变量表是对应线程的，那么它的生命周期是怎样的？

垃圾收集机制中引用计数器的缺点是什么？