BONC
东方国信

让数据改变工作与生活

Information Change Work And Life

www.bonc.com.cn

BONC

1 **JVM结构**

2 JVM类加载机制

3 JVM运行时数据区

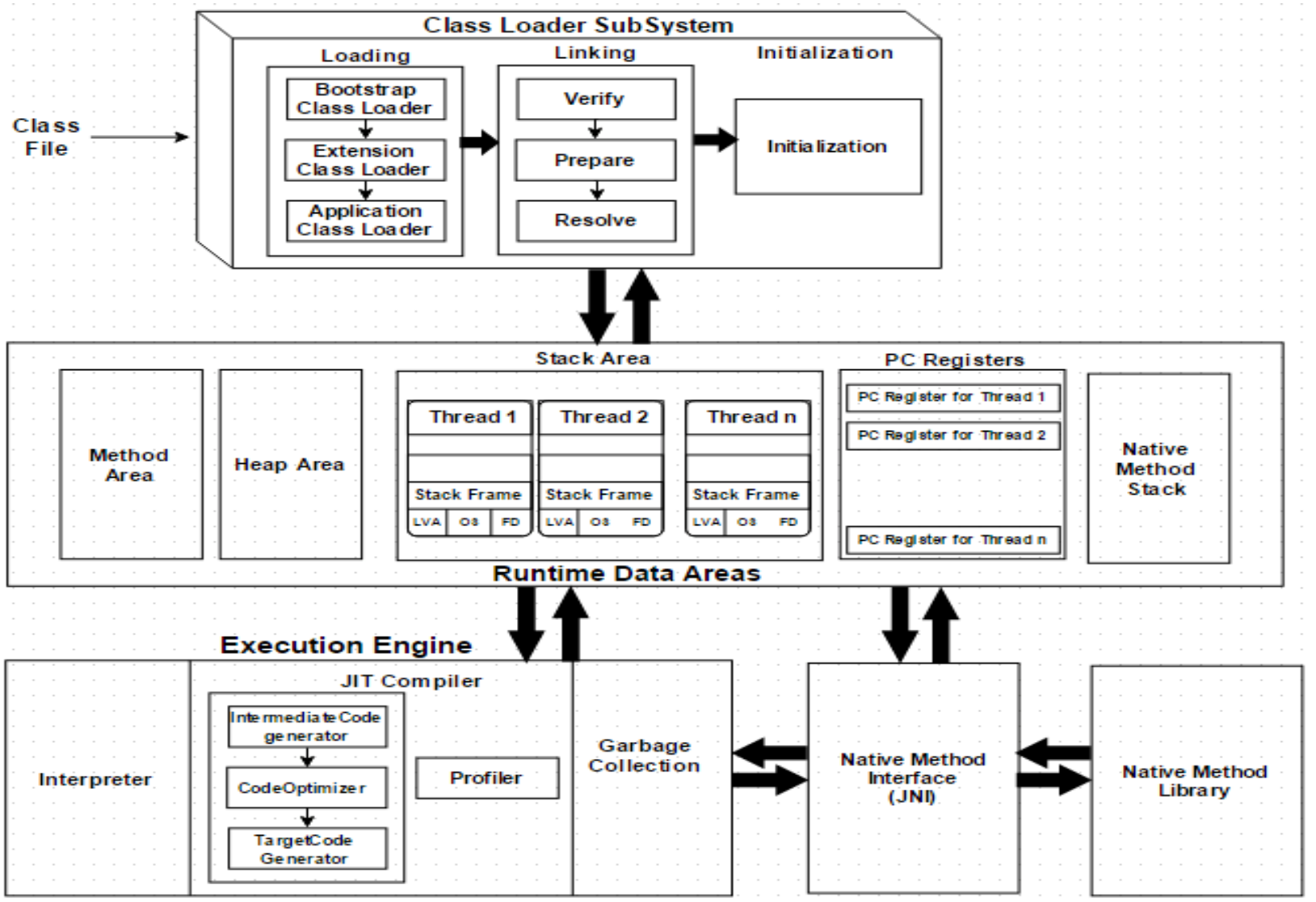
4 执行引擎

5 JVM GC

6 监控工具

- PC计数器
- 栈
- 堆
- 方法区
- 类加载器
- 执行引擎

JVM结构

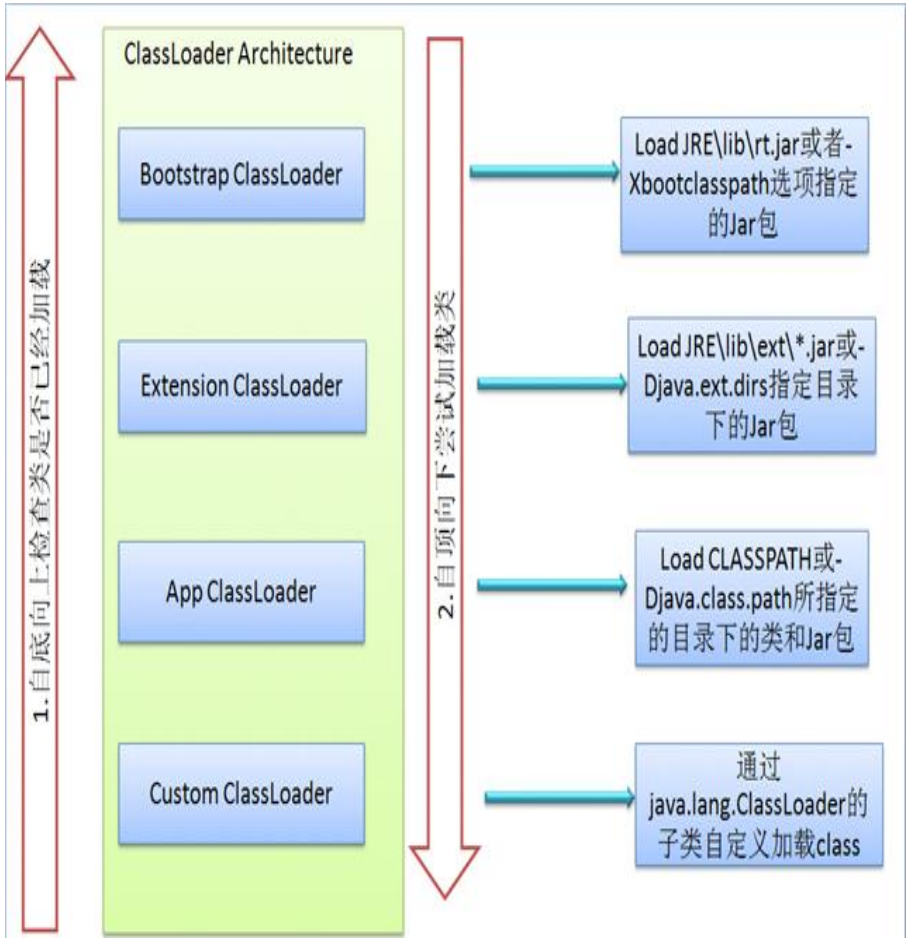

Class Loader SubSystem
Loading
Linking
Initialization
Class File
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Stack Area
PC Registers
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas
Execution Engine
JIT Compiler
Intermediate Code generator
CodeOptimizer
Target Code Generator
Profiler
Interpreter
Garbage Collection
Native Method Interface (JNI)
Native Method Library

1 JVM类加载器

2 JVM类加载机制

- Java提供了动态的装载特性；它会在运行时的第一次引用到一个class的时候对它进行装载和链接，而不是在编译期进行。JVM的类装载器负责动态装载。Java类装载器有如下几个特点：
  - 层级结构：Java里的类装载器被组织成了有父子关系的层级结构。Bootstrap类装载器是所有装载器的父亲。
  - 代理模式：基于层级结构，类的装载可以在装载器之间进行代理。当装载器装载一个类时，首先会检查它是否在父装载器中进行装载了。如果上层的装载器已经装载了这个类，这个类会被直接使用。反之，类装载器会请求装载这个类。
  - 可见性限制：一个子装载器可以查找父装载器中的类，但是一个父装载器不能查找子装载器里的类。
  - 不允许卸载：类装载器可以装载一个类但是不可以卸载它，不过可以删除当前的类装载器，然后创建一个新的类装载器。**Note:**Java虚拟机本身会始终引用这些类加载器（自带的类加载器包括根类加载器、扩展类加载器和系统类加载器），而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的.由用户自定义的类加载器加载的类是可以被卸载的

类加载器



每个类装载器都有一个自己的命名空间用来保存已装载的类。**当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name)进行搜索来检测这个类是否已经被加载了。**如果两个类的全局限定名是一样的，但是如果命名空间不一样的话，那么它们还是不同的类。不同的命名空间表示class被不同的类装载器装载。当一个类装载器（class loader）被请求装载类时，它首先按照顺序在上层装载器、父装载器以及自身的装载器的缓存里检查这个类是否已经存在。简单来说，就是在缓存里查看这个类是否已经被自己装载过了，如果没有的话，继续查找父类的缓存，直到在bootstrap类装载器里也没有找到的话，它就会自己在文件系统里去查找并且加载这个类。

启动类加载器（Bootstrap class loader）：这个类装载器是在JVM启动的时候创建的。它负责装载Java API，包含Object对象。和其他的类装载器不同的地方在于这个装载器是通过native code来实现的，而不是用Java代码。

扩展类加载器（Extension class loader）：它装载除了基本的Java API以外的扩展类。它也负责装载其他的安全扩展功能。

系统类加载器（System class loader）：如果说bootstrap class loader和extension class loader负责加载的是JVM的组件，那么system class loader负责加载的是应用程序类。它负责加载用户在$CLASSPATH里指定的类。

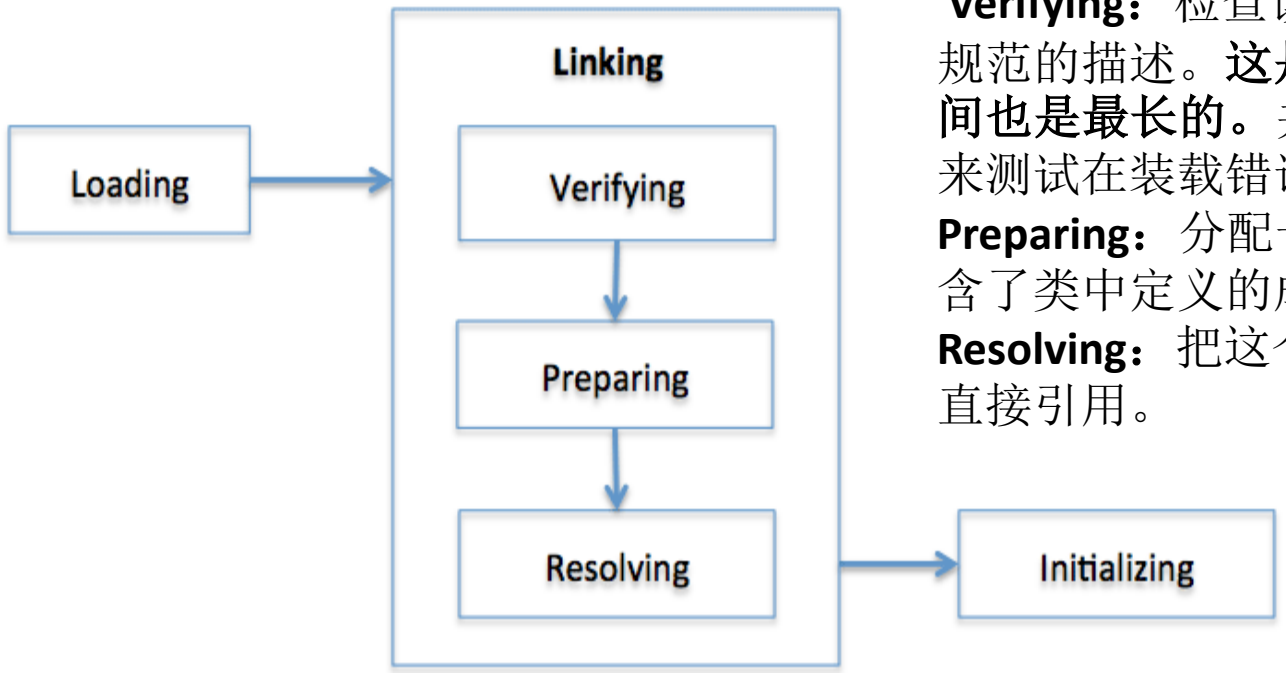
用户自定义类加载器（User-defined class loader）:这是应用程序开发者用直接用代码实现的类装载器。

1 JVM类加载器

2 **JVM类加载机制**

类加载器机制

如果类装载器查找到一个没有装载的类，它会按照下图的流程来装载和链接这个类：



Loading： 类的信息从文件中获取并且载入到JVM的内存里。
Verifying： 检查读入的结构是否符合Java语言规范以及JVM规范的描述。**这是类装载中最复杂的过程，并且花费的时间也是最长的。**并且JVM TCK工具的大部分场景的用例也用来测试在装载错误的类的时候是否会出现错误。
Preparing： 分配一个结构用来存储类信息，这个结构中包含了类中定义的成员变量，方法和接口的信息。
Resolving： 把这个类的常量池中的所有的符号引用改变成直接引用。

Initializing： 把类中的变量初始化成合适的值。执行静态初始化程序，把静态变量初始化成指定的值。
JVM规范定义了上面的几个任务，不过它允许具体执行的时候能够有些灵活的变动

主要完成3件事：

- 通过类全限定名获取(可以是class文件、zip、jar、运行时生成)定义此类的二进制字节流
- 通过此字节流所代表的静态存储结构转化为方法区的运行时数据结构
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据结构的访问入口

注意:

1、对于java.lang.Class对象，java规范并未明确规定是在java堆中，对于HotSpot而言，存放在方法区中

2、数组类有点不同，因为数组类不通过类加载器创建而是由VM直接创建。但是数组的ComponentType(去掉维度)还是由类加载器创建。创建规则遵循以下原则：

- 如果ComponentType是reference类型，那就递归用加载器加载此ComponentType，此数组将在加载该ComponentType的类加载器的namespace上被标识
- 如果不是reference类型(int)，则把数组标识为引导类加载器关联
- 如果ComponentType是reference类型,其可见性与ComponentType可见性一致，否则为public

Verifying

是linking的第一阶段，主要目的是确保class字节流中包含的信息符合虚拟机规范，并且不会危害虚拟机自身安全。

- 文件格式验证(字节流验证)
  - 是否以魔数0xCAFEBABE开头
  - 主次版本号在当前虚拟机处理范围之内
  - 常量池常量是否有不被支持的常量类型
  - 指向常量池各种索引值是否指向不存在的常量或不符合类型的常量

  ……
- 元数据验证(方法区存储结构验证)
  - 此类是否有父类(除java.lang.Object以外，所有类都有父类)
  - 是否继承了不允许继承的父类(final类)
  - 如果不是抽象类，是否实现其父类或接口中要求实现的所有方法
  - 字段、方法是否与父类产生矛盾(final方法)

  ……

- 字节码验证(方法区存储结构验证)

通过数据流和控制流分析，确定程序语义是合法、符合逻辑的，是对方法体进行校验

  - 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作
  - 保证跳转指令不会跳到方法体以外的字节码指令上
  - 保证方法体中类型转换是有效的

  ……

- 符号引用验证(方法区存储结构验证)

  校验符号引用转化为直接引用,以确保解析动作能正常执行

  - 符号引用中通过字符串描述的全限定名是否能找到对应的类
  - 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段
  - 符号引用中的类、字段、方法的访问性(private default protected public)是否能背当前类访问

  ……

■ 正式为类变量分配内存并设置类变量初始值阶段，所使用的内存都在方法区中分配.

类变量为static修饰的变量，而不是实例变量，实例变量将会在实例化时随着对象在堆中分配

public static int age=20

在准备阶段后，age值为0，而不是20，因为这时未执行任何java方法，而把age赋值为20的putstatic指令在程序被编译后，存放于类构造器<clinit>()方法中，所以把age赋值为20的动作是在**Initializing,其基本数据类型零值如下表所示**

数据类型	零值	数据类型	零值
int	0	boolean	false
long	0L	float	0.0f
short	(short)0	double	0.0d
char	‘\u0000’	reference	null
byte	(byte)0		

特殊情况： static final int value = 20则直接赋值为20

Resolving

BONC

- 将常量池中的符号引用替换为直接引用的过程
- 符号引用：常量池中的CONSTANT_[Class|Fieldref|Methodref]_info的常量，其引用的对象不一定在内存中
- 直接引用:类似指向目标对象的指针、相对偏移等，其引用的对象已经在内存中

Initializing

BONC

- 执行类构造器<clinit>()方法过程，按照程序员计划初始化类变量，即static变量(可能指令重排)和static{}代码块执行

类加载及其初始化总结

BONC

1.在栈内建立变量

2.类加载进内存

~~*3.执行静态代码块*~~

4.在堆内存中开辟空间，分配内存地址

5.在堆内存中建立对象的特有属性，并进行默认初始化

6.对属性进行显示初始化

7.对对象进行构造代码块初始化

8.对对象进行对应的构造函数初始化

9.将内存地址赋给栈内存中的变量

对象创建

虚拟机遇到new指令时，首先检查指令参数在常量池中是否定位到一个类的符号引用，并且检查此类是否被加载、解析初始化过，如果没有则先执行类的加载，类加载检查通过后，vm将为对象分配堆内存(当类加载完成后对象所需内存确定)

内存分配完成后，vm将分配到的内存空间初始化为零值(不包括对象头)，之后对对象的对象头进行设置，如：对象是哪个类的的实例、如何查询类的元数据信息、对象哈希码、GC分代年龄等

此时，从vm看，一个新的对象已经产生，但从java程序视角来看，对象创建才刚刚开始,<init>方法还未执行，所有字段还是零值，执行new指令过后会接着执行<init>方法，这样才会按照程序员的意愿初始化，这样一个可用对象才真正创建出来

对象的内存布局

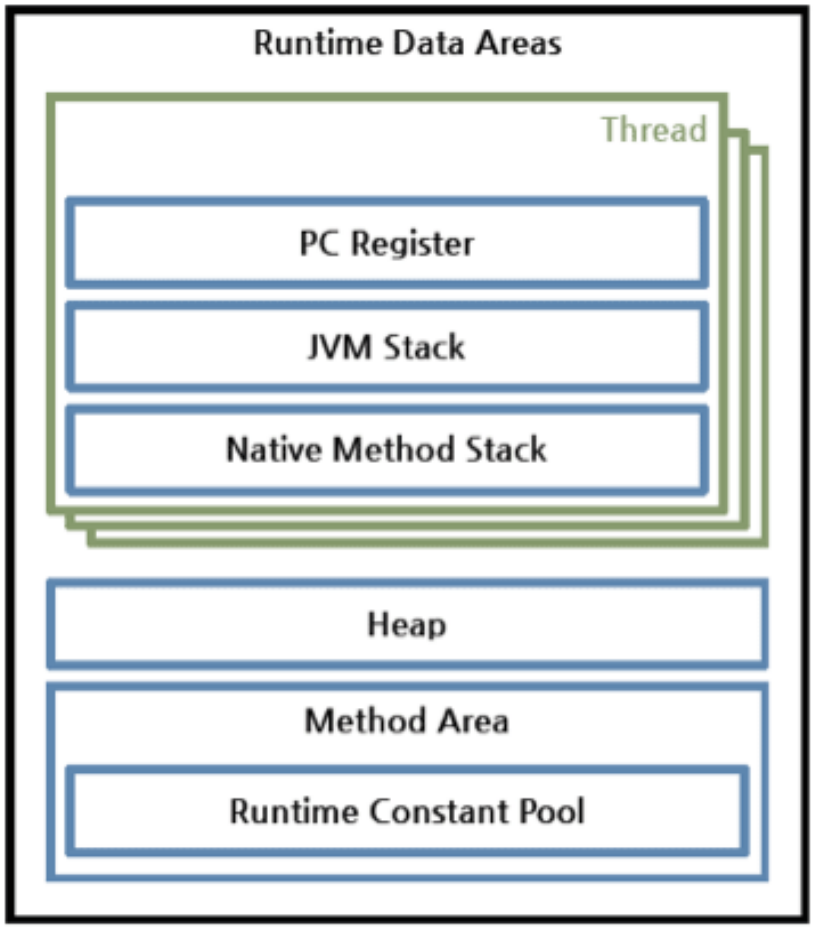
在hotspot vm中对象在内存中存储分为3部分：对象头(Header)、实例数据(Instance Data)和对齐填充(padding)

- 对象头包括两部分信息：
  1、对象自身运行时数据(MarkWord)，如hashCode、GC分代年龄、锁状态标识、线程持有的锁、偏向线程ID、偏向时间戳等
  2、类型指针，指向类元数据指针
- 实例数据：对象有效数据，程序定义的字段内容。
- 对其填充：hotspot vm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。

BONC

1. **方法区**
2. 栈
3. 堆
4. 非堆

Runtime Data Areas



- 运行时数据区是在JVM运行的时候操作系统所分配的内存区。运行时内存区可以划分为6个区域。在这6个区域中，一个PC Register,JVM stack 以及Native Method Statck都是按照线程创建的，Heap,Method Area以及Runtime Constant Pool都是被所有线程公用的。

PC 指当前指令（或操作码）的地址，本地指令除外。如果当前方法是 native 方法，那么PC 的值为 undefined。所有的 CPU 都有一个 PC，典型状态下，每执行一条指令 PC 都会自增，因此 PC 存储了指向下一条要被执行的指令地址。JVM 用 PC 来跟踪指令执行的位置，PC 将实际上是指向方法区（Method Area）的一个内存地址，是jvm中唯一没有OOM的区域

方法区

- 方法区是所有线程共享的，它是在JVM启动的时候创建的。它保存所有被JVM加载的类和接口的运行时常量池，成员变量以及方法的信息，静态变量以及方法的字节码。JVM的提供者可以通过不同的方式来实现方法区。在Oracle 的HotSpot JVM里，方法区被称为永久区或者永久代（PermGen）。是否对方法区进行垃圾回收对JVM的实现是可选的。Java8(meta区)之前，也叫permgen，所有线程共享，用于保存类信息，如类的字段、方法、常量池等信息，如果系统中使用大量反射，可能在运行时生成大量类，可能会发生OOM，故此需设置合理大小或合理利用反射，java8之前使用-XX:MaxPermSize设置最大值，java8已经不存在，java8叫metaspzce区，其为非堆内存(系统内存)，-XX:MaxMetaspaceSize
- Classloader 引用

所有的类加载之后都包含一个加载自身的加载器的引用，反过来每个类加载器都包含它们加载的所有类的引用。

- 运行时常量池：这个区域和class文件里constant_pool是相对应的。这个区域是包含在方法区里的，不过，对于JVM的操作而言，它是一个核心的角色。因此在JVM规范里特别提到了它的重要性。除了包含每个类和接口的常量，它也包含了所有方法和变量的引用。简而言之，**当一个方法或者变量被引用时，JVM通过运行时常量区来查找方法或者变量在内存里的实际地址**。常量池中可以存储多种类型的数据：
  - ✓数值型常量
  - ✓字段引用
  - ✓方法引用
  - ✓属性
- 字段数据

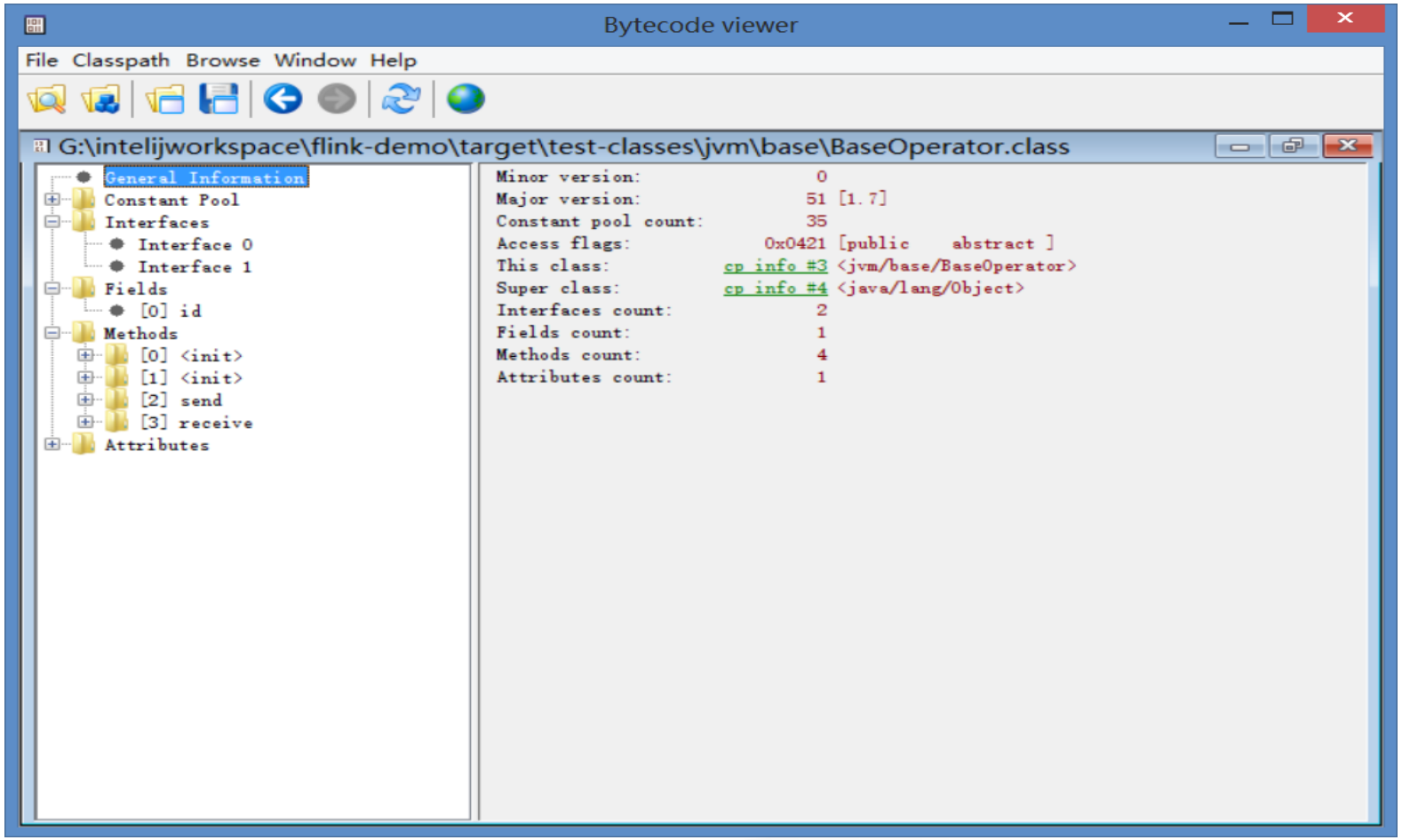
  针对每个字段的信息
  字段名
  类型
  修饰符
  属性（Attribute）

方法区

BONC

方法数据
- 每个方法
- 方法名
- 返回值类型
- 参数类型（按顺序）
- 修饰符
- 属性

•方法代码
- 每个方法
- 字节码
- 操作数栈大小
- 局部变量大小
- 局部变量表



方法区

异常表

异常表像这样存储每个异常处理信息

- 每个异常处理器
- 开始点
- 结束点
- 异常处理代码的程序计数器（PC）偏移量
- 被捕获的异常类对应的常量池下标

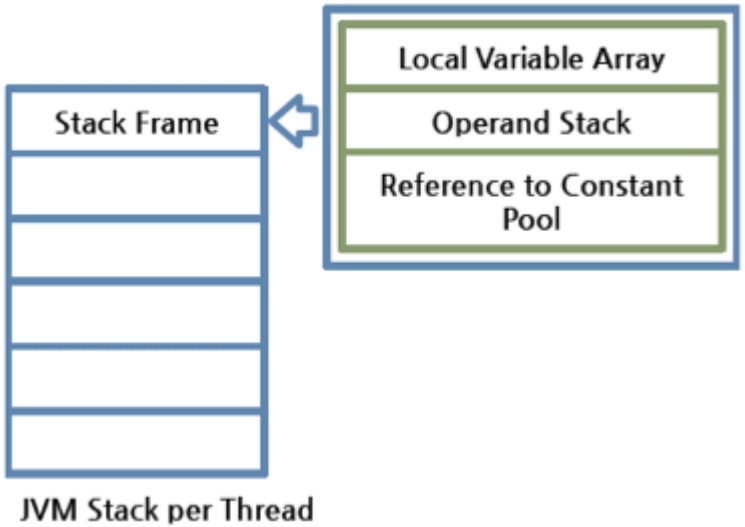
如果一个方法有定义 try-catch 或者 try-finally 异常处理器，那么就会创建一个异常表。它为每个异常处理器和 finally 代码块存储必要的信息，包括处理器覆盖的代码块区域和处理异常的类型。

当方法抛出异常时，JVM 会寻找匹配的异常处理器。如果没有找到，那么方法会立即结束并弹出当前栈帧，这个异常会被重新抛到调用这个方法的方法中（在新的栈帧中）。如果所有的栈帧都被弹出还没有找到匹配的异常处理器，那么这个线程就会终止。如果这个异常在最后一个非守护进程抛出（比如这个线程是主线程），那么也有会导致 JVM 进程终止。

Finally 异常处理器匹配所有的异常类型，且不管什么异常抛出 finally 代码块都会执行。在这种情况下，当没有异常抛出时，finally 代码块还是会在方法最后执行。这种靠在代码 return 之前跳转到 finally 代码块来实现

所有线程共享同一个方法区，因此访问方法区数据的和动态链接的进程必须线程安全。如果两个线程试图访问一个还未加载的类的字段或方法，必须只加载一次，而且两个线程必须等它加载完毕才能继续执行法区存储了每个类的信息

栈



每个线程启动的时候，都会创建一个JVM栈。它是用来保存栈帧的。JVM只会在JVM栈上对栈帧进行push和pop的操作。如果出现了异常，堆栈跟踪信息的每一行都代表一个栈帧立的信息，这些信息是通过类似于printStackTrace()这样的方法来展示的。每个线程拥有自己的栈，栈包含每个方法执行的栈帧。栈是一个后进先出（LIFO）的数据结构，因此当前执行的方法在栈的顶部。每次方法调用时，一个新的栈帧创建并压栈到栈顶。当方法正常返回或抛出未捕获的异常时，栈帧就会出栈。除了栈帧的压栈和出栈，栈不能被直接操作。

栈的限制：栈可以是动态分配也可以固定大小。如果线程请求一个超过允许范围的空间，就会抛出一个StackOverflowError。如果线程需要一个新的栈帧，但是没有足够的内存可以分配，就会抛出一个 OutOfMemoryError。

栈帧

每当一个方法在JVM上执行的时候，都会创建一个栈帧，并且会添加到当前线程的JVM堆栈上。当这个方法执行结束的时候，这个栈帧就会被移除。每个栈帧里都包含有当前正在执行的方法所属类的**本地变量数组，操作数栈，以及运行时常量池的引用。**本地变量数组的和操作数栈的大小都是在编译时确定的。因此，一个方法的栈帧的大小也是固定不变的。

每个栈帧包含：

- ✓局部变量数组
- ✓返回值
- ✓操作数栈
- ✓类当前方法的运行时常量池引用

局部变量数组

局部变量数组包含了方法执行过程中的所有变量，包括 this 引用、所有方法参数、其他局部变量。对于类方法（也就是静态方法），方法参数从下标 0 开始，对于对象方法，位置0保留为 this。

局部变量包括：boolean,、byte、char、long、short、int、float、double、reference、returnAddress，除了 long 和 double 类型以外，所有的变量类型都占用局部变量数组的一个位置。long 和 double 需要占用局部变量数组两个连续的位置，因为它们是 64 位双精度，其它类型都是 32 位单精度

方法实际运行的工作空间。每个方法都在操作数栈和局部变量数组之间交换数据，并且压入或者弹出其他方法返回的结果。操作数栈所需的最大空间是在编译期确定的。因此，操作数栈的大小也可以在编译期间确定。操作数栈在执行字节码指令过程中被用到，其主要用于保存计算过程的中间结果，同时作为计算过程中变量的临时存储空间。这种方式类似于原生 CPU 寄存器。大部分 JVM 字节码把时间花费在操作数栈的操作上：入栈、出栈、复制、交换、产生消费变量的操作。因此，局部变量数组和操作数栈之间的交换变量指令操作通过字节码频繁执行。
本地方法栈(Native method stack)：供用非Java语言实现的本地方法的堆栈。换句话说，它是用来调用通过JNI(Java Native Interface Java本地接口）调用的C/C++代码。根据具体的语言，一个C堆栈或者C++堆栈会被创建

动态链接

每个栈帧都有一个运行时常量池的引用。这个引用指向栈帧当前运行方法所在类的常量池。通过这个引用支持动态链接（dynamic linking）。

运行环境包括对指向当前类和当前方法的解释器符号表的指针,用于支持方法代码的动态链接。方法的class 文件代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用,装载必要的类以解释还没有定义的符号,并把变量访问翻译成与这些变量运行时的存储结构相应的偏移地址。动态链接方法和变量使得方法中使用的其它类的变化不会影响到本程序的代码.

返回值

正常的方法返回 :如果当前方法正常地结束了,在执行了一条具有正确类型的返回指令时,调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的寄存器,并把调用者的程序计数器增加一个恰当的数值,以跳过已执行过的方法调用指令,然后在调用者的执行环境中继续执行去。
异常捕捉 :异常情况在 Java 中被称作 Error(错误)或 Exception(异常),是 Throwable 类的子类,
在程序中的原因是:

①动态链接错,如无法找到所需的 class 。

②运行时错,如对一个空指针的引用。程序使用了 throw 语句。

当异常发生时,Java 虚拟机采取如下措施:

检查与当前方法相联系的 catch 子句表。每个 catch 子句包含其有效指令范围,能够处理的异常类型,以及处理异常的代码块地址。

与异常相匹配的 catch 子句应该符合下面的条件:造成异常的指令在其指令范围之内,发生的异常类型是其能处理的异常类型的子类型。如果找到了匹配的 catch 子句,那么系统转移到指定的异常处理块处执行;如果没有找到异常处理块,重复寻找匹配的 catch 子句的过程,直到当前方法的所有嵌套的 catch 子句都被检查过。
由于虚拟机从第一个匹配的 catch 子句处继续执行,所以 catch 子句表中的顺序是很重要的。因为 Java代码是结构化的,因此总可以把某个方法的所有的异常处理器都按序排列到一个表中,对任意可能的程序计数器的值,都可以用线性的顺序找到合适的异常处理块,以处理在该程序计数器值下发生的异常情况。
如果找不到匹配的catch 子句,那么当前方法得到一个"未截获异常"的结果并返回到当前方法的调用者,好像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块,那么这种错误将被传播下去。如果错误被传播到最顶层,那么系统将调用一个缺省的异常处理块。

堆被用来在运行时分配类实例、数组。不能在栈上存储数组和对象（逃逸分析和标量替换除外）。因为栈帧被设计为创建以后无法调整大小。栈帧只存储指向堆中对象或数组的引用。与局部变量数组（每个栈帧中的）中的原始类型和引用类型不同，对象总是存储在堆上以便在方法结束时不会被移除。对象只能由垃圾回收器移除。

为了支持垃圾回收机制，堆被分为了下面三个区域：

- 新生代
  - 经常被分为 Eden 和 两个Survivor
- 老年代
- 永久代

- 内存管理
  - 对象和数组永远不会显式回收，而是由垃圾回收器自动回收。通常，过程是这样的：

1. Minor垃圾回收将发生在新生代。依旧存活的对象将从 eden 区移到 survivor 区。
2. Major垃圾回收一般会导致应用进程暂停，它将在三个区内移动对象。仍然存活的对象将被从新生代移动到老年代。
3. 当老年代空间不足进行回收(FULL GC)

Tenured回收条件

BONC

1、System.gc()方法的调用

此方法的调用是建议JVM进行Full GC,虽然只是建议而非一定,但很多情况下它会触发 Full GC,从而增加Full GC的频率,也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存，可通过通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

2、老年代代空间不足

老年代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：Java.lang.OutOfMemoryError: Java heap space

为避免以上两种状况引起的Full GC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组

3、 PermGen空间不足

JVM规范中运行时数据区域中的方法区，在HotSpot虚拟机中又被习惯称为永生代或者永生区，Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下也会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：java.lang.OutOfMemoryError: PermGen space

为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC

4、CMS GC时出现promotion failed和concurrent mode failure

对于采用CMS进行老年代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。

promotion failed是在进行Minor GC时，survivor space放不下、对象只能放入老年代，而此时老年代也放不下造成的；concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老年代，而此时老年代空间不足造成的（有时候“空间不足”是CMS GC时当前的浮动垃圾过多导致暂时性的空间不足触发Full GC）。

应对对措施为：增大survivor space、老年代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX: CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。

5、堆中分配很大的对象

所谓大对象，是指需要大量连续内存空间的java对象，例如很长的数组，此种对象会直接进入老年代，而老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发JVM进行Full GC。

为了解决这个问题，CMS垃圾收集器提供了一个可配置的参数，即-XX:+UseCMSCompactAtFullCollection开关参数，用于在“享受”完Full GC服务之后额外免费赠送一个碎片整理的过程，内存整理的过程无法并发的，空间碎片问题没有了，但提顿时间不得不变长了，JVM设计者们还提供了另外一个参数 -XX:CMSFullGCsBeforeCompaction,这个参数用于设置在执行多少次不压缩的Full GC后,跟着来一次带压缩的

非堆内存

非堆内存指的是那些逻辑上属于 JVM 一部分对象，但实际上不在堆上创建。
非堆内存包括：

- 永久代(方法区、驻留字符串(interned strings))
- 代码缓存（Code Cache）：用于编译和存储那些被 JIT 编译器编译成原生代码的方法。

1 解释器

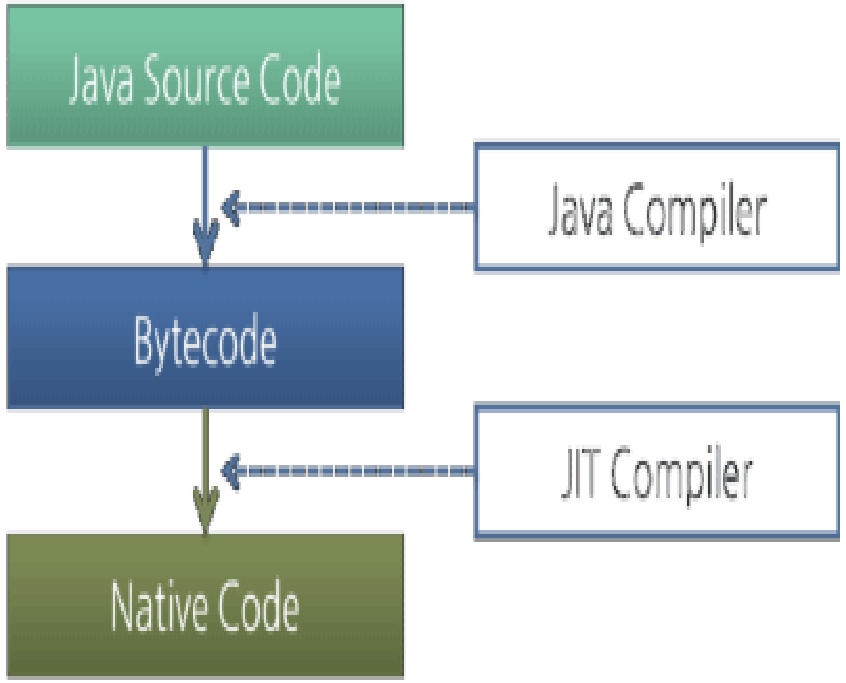
2 即时编译

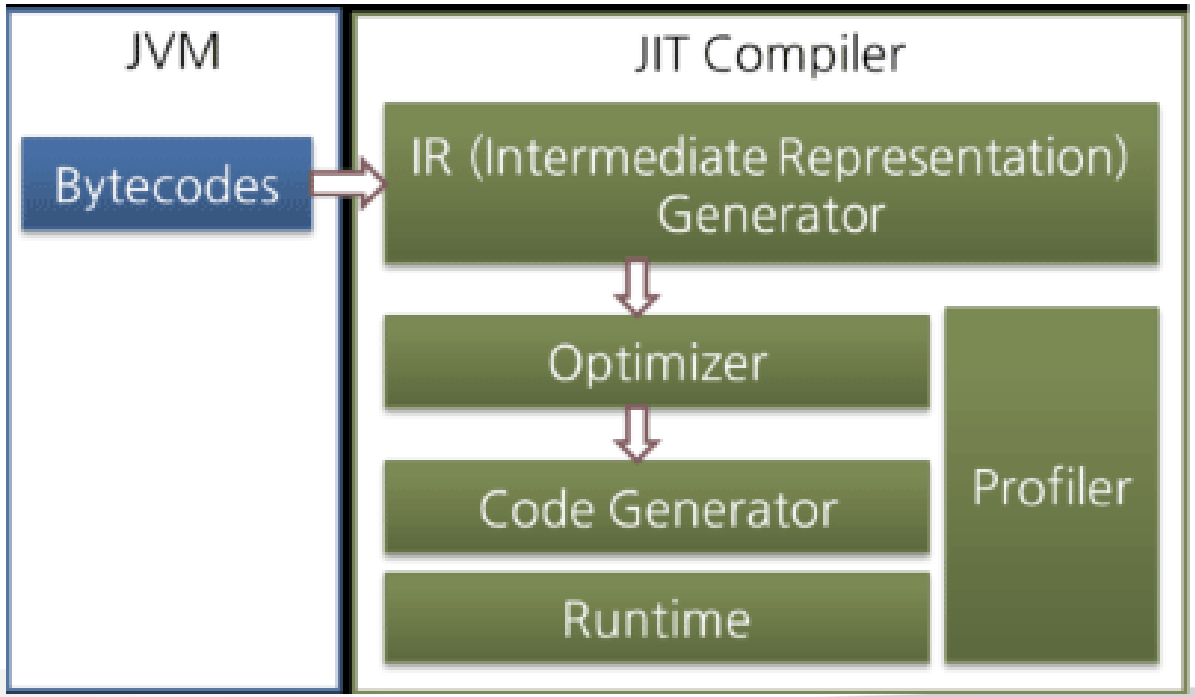
执行引擎

- 通过类装载器装载的，被分配到JVM的运行时数据区的字节码会被执行引擎执行。执行引擎以指令为单位读取Java字节码。它就像一个CPU一样，一条一条地执行机器指令。每个字节码指令都由一个1字节的操作码和附加的操作数组成。执行引擎取得一个操作码，然后根据操作数来执行任务，完成后就继续执行下一条操作码。
- 不过Java字节码是用一种人类可以读懂的语言编写的，而不是用机器可以直接执行的语言。因此，执行引擎必须把字节码转换成可以直接被JVM执行的语言。字节码可以通过以下两种方式转换成合适的语言。
- 解释器：一条一条地读取，解释并且执行字节码指令。因为它一条一条地解释和执行指令，所以它可以很快地解释字节码，但是执行起来会比较慢。这是解释执行的语言的一个缺点。字节码这种“语言”基本来说是解释执行的。
- 即时(Just-In-Time)编译器：即时编译器被引入用来弥补解释器的缺点。执行引擎首先按照解释执行的方式来执行，然后在合适的时候，即时编译器把整段字节码编译成本地代码。然后，执行引擎就没有必要再去解释执行方法了，它可以直接通过本地代码去执行它。执行本地代码比一条一条进行解释执行的速度快很多。编译后的代码可以执行的很快，因为本地代码是保存在缓存里的

执行引擎

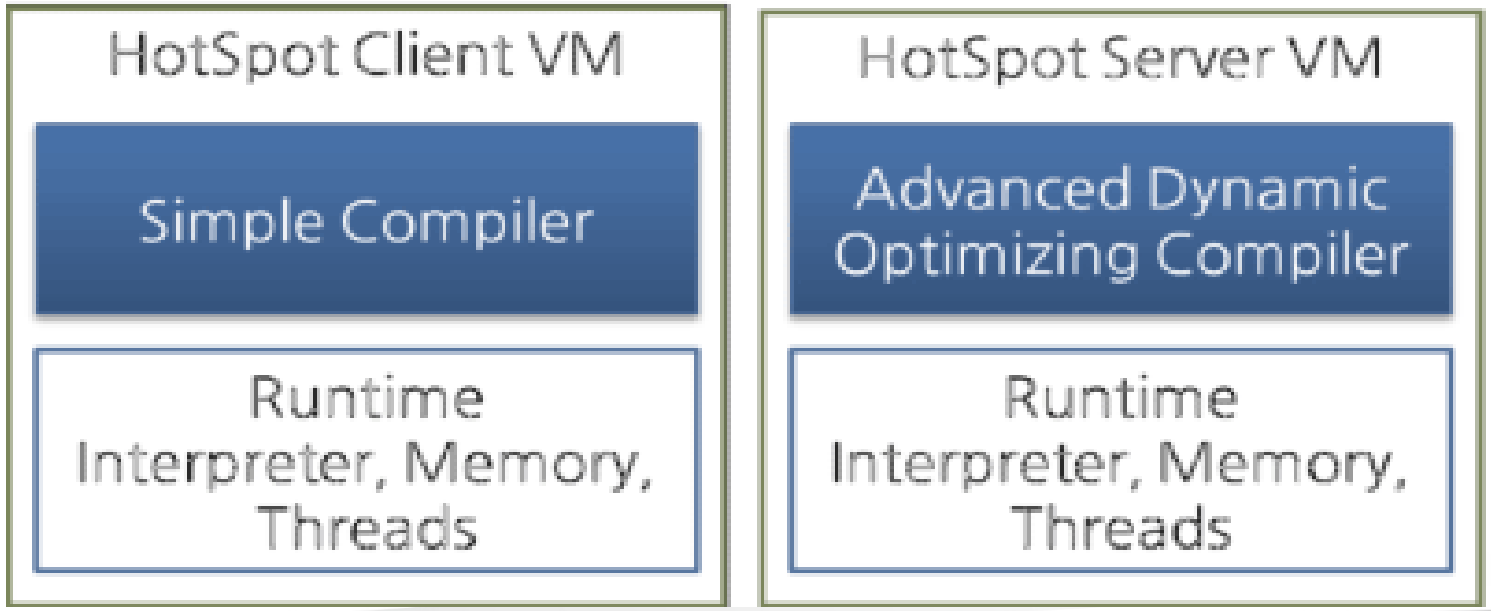
不过，用JIT编译器来编译代码所花的时间要比用解释器去一条条解释执行花的时间要多。因此，如果代码只被执行一次的话，那么最好还是解释执行而不是编译后再执行。因此，内置了JIT编译器的JVM都会检查方法的执行频率，如果一个方法的执行频率超过一个特定的值的话，那么这个方法就会被编译成本地代码。JVM规范没有定义执行引擎该如何去执行。因此，JVM的提供者通过使用不同的技术以及不同类型的JIT编译器来提高执行引擎的效率。





JIT编译器把字节码转换成一个中间层表达式，一种中间层的表示方式，来进行优化，然后再把这种表示转换成本地代码。

Oracle Hotspot VM使用一种叫做热点编译器的JIT编译器。**它之所以被称作”热点“是因为热点编译器通过分析找到最需要编译的“热点”代码，然后把热点代码编译成本地代码。**如果已经被编译成本地代码的字节码不再被频繁调用了，换句话说，这个方法不再是热点了，那么Hotspot VM会把编译过的本地代码从cache里移除，并且重新按照解释的方式来执行它。Hotspot VM分为Server VM和Client VM两种，这两种VM使用不同的JIT编译器。



- -Xms10m 堆最小值
- -Xmx10m 堆最大值
- -Xmn 新生代大小
- -XX:SurvivorRatio 表示新生代eden/from（to）
- -XX:NewRation 表示老年代/新生代
- -XX:+DoEscapeAnalysis 开启逃逸分析
- -XX:+EliminateAllcations开启标量替换
- -XX:+PrintGC 打印gc日志
- -XX:+PrintGCDetails 打印GC详细日志
- -XX:MaxPermSize (方法区最大值，jdk8以前，jdk8：-XX:MaxMetaspaceSize)
- -XX:MaxDirectMemorySize 直接内存最大值，默认为-Xmx值

安全点(safepoint)

BONC

1 Safepoint介绍

2 Safepoint-解释执行

3 Safepoint-JIT执行

4 Safepoint-小结

5 Safepoint-源码分析

安全点(safepoint)

BONC

- safepoint 安全点指一些特定的位置，当线程运行到这些位置时，线程的一些状态可以被确定(the thread's representation of it's Java machine state is well described)，比如记录OopMap的状态，从而确定GC Root的信息，使JVM可以安全的进行一些操作，比如开始GC。
- safepoint指的特定位置主要有:
  - 循环的末尾 (防止大循环的时候一直不进入safepoint，而其他线程在等待它进入safepoint)
  - 方法返回前
  - 调用方法的call之后
  - 抛出异常的位置
- 之所以选择这些位置作为safepoint的插入点，主要的考虑是“避免程序长时间运行而不进入safepoint”，比如GC的时候必须要等到Java线程都进入到safepoint的时候VMThread才能开始执行GC，如果程序长时间运行而没有进入safepoint，那么GC也无法开始，JVM可能进入到Freezen假死状态。在stackoverflow上有人提到过一个问题，由于BigInteger的pow执行时JVM没有插入safepoint,导致大量运算时线程一直无法进入safepoint，而GC线程也在等待这个Java线程进入safepoint才能开始GC，结果JVM就Freezen了。
- http://blog.ragozin.info/2012/10/safepoints-in-hotspot-jvm.html

垃圾回收-安全点(safepoint)

- JVM在很多场景下使用到safepoint, 最常见的场景就是GC的时候。对一个Java线程来说，它要么处在safepoint,要么不在safepoint。
  - Garbage collection pauses
  - Code deoptimization
  - Flushing code cache
  - Class redefinition (e.g. hot swap or instrumentation)
  - Biased lock revocation
  - Various debug operation (e.g. deadlock check or stacktrace dump)

- GC的标记阶段需要stop the world，让所有Java线程挂起，这样JVM才可以安全地来标记对象。safepoint可以用来实现让所有Java线程挂起的需求。这是一种"主动式"(Voluntary Suspension)的实现。JVM有两种执行方式：解释型和编译型(JIT)

- JVM会设置一个2字节的dispatch tables,解释器执行的时候会经常去检查这个dispatch tables，当有safepoint请求的时候，就会让线程去进行safepoint检查

- JIT编译的时候直接把safepoint的检查代码加入了生成的本地代码，当JVM需要让Java线程进入safepoint的时候，只需要设置一个标志位，让Java线程运行到safepoint的时候主动检查这个标志位，如果标志被设置，那么线程停顿，如果没有被设置，那么继续执行。

- 例如hotspot在x86中为轮询safepoint会生成一条类似于“test %eax,0x160100”的指令，JVM需要进入gc前，先把0x160100设置为不可读，那所有线程执行到检查0x160100的test指令后都会停顿下来

```
0x01b6d627: call   0x01b2b210          ; OopMap{[60]=Oop off=460}
                                        ;*invokeinterface size
                                        ; - Client1::main@113 (line 23)
                                        ;   {virtual_call}
 0x01b6d62c: nop                        ; OopMap{[60]=Oop off=461}
                                        ;*if_icmplt
                                        ; - Client1::main@118 (line 23)
 0x01b6d62d: test   %eax,0x160100       ;   {poll}
 0x01b6d633: mov    0x50(%esp),%esi
 0x01b6d637: cmp    %eax,%esi
```

- VMThread会一直等待直到VMOperationQueue中有操作请求出现，比如GC请求。而VMThread要开始工作必须要等到所有的Java线程进入到safepoint。JVM维护了一个数据结构，记录了所有的线程，所以它可以快速检查所有线程的状态。当有GC请求时，所有进入到safepoint的Java线程会在一个Thread_Lock锁阻塞，直到当JVM操作完成后，VM释放Thread_Lock，阻塞的Java线程才能继续运行。
- GC stop the world的时候，所有运行Java code的线程被阻塞，如果运行native code线程不去和Java代码交互，那么这些线程不需要阻塞。VM操作相关的线程也不会被阻塞。
- safepoint只能处理正在运行的线程，它们可以主动运行到safepoint。而一些Sleep或者被blocked的线程不能主动运行到safepoint。这些线程也需要在GC的时候被标记检查，JVM引入了safe region的概念。safe region是指一块区域，这块区域中的引用都不会被修改，比如线程被阻塞了，那么它的线程堆栈中的引用是不会被修改的，JVM可以安全地进行标记。线程进入到safe region的时候先标识自己进入了safe region，等它被唤醒准备离开safe region的时候，先检查能否离开，如果GC已经完成，那么可以离开，否则就在safe region呆着。这可以理解，因为如果GC还没完成，那么这些在safe region中的线程也是被stop the world所影响的线程的一部分，如果让他们可以正常执行了，可能会影响标记的结果
- 可以设置JVM参数 -XX:+PrintSafepointStatistics –XX:PrintSafepointStatisticsCount=1 来输出safepoint的统计信息

- 源码位于
  D:\git\openjdk\hotspot\src\share\vm\runtime\safepoint.cpp

  D:\git\openjdk\hotspot\src\share\vm\runtime\vmThread.cpp

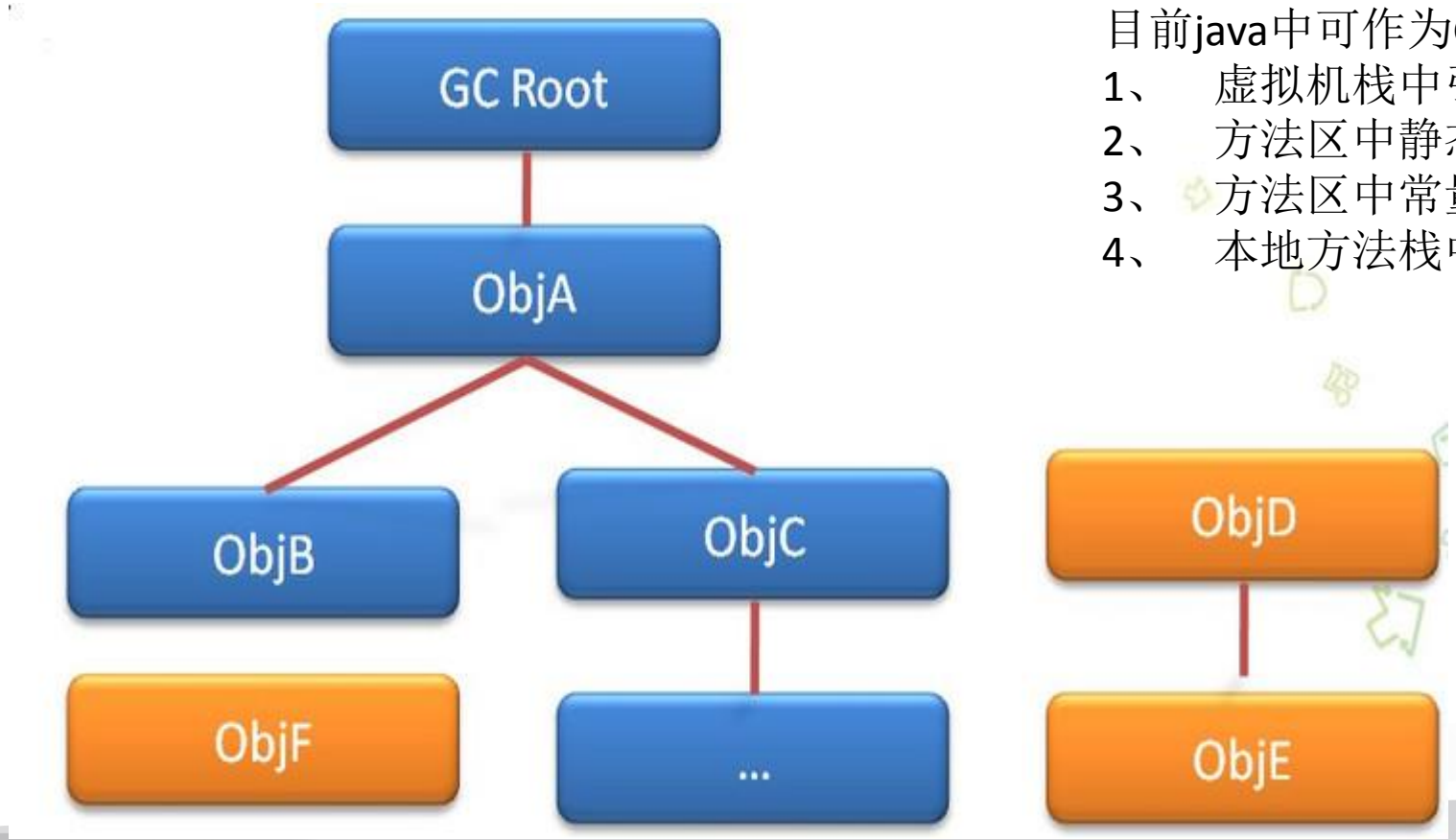
垃圾回收算法

BONC

1. 引用计数法(reference counting)
2. 标记清除(mark-sweep)
3. 复制算法(copying)
4. 标记压缩(mark-compact)
5. 分代算法(Generational collecting)
6. 分区算法(region)

垃圾回收

根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。



目前java中可作为GC Root的对象有

1、 虚拟机栈中引用的对象（本地变量表）
2、 方法区中静态属性引用的对象
3、 方法区中常量引用的对象
4、 本地方法栈中引用的对象（Native对象）

- 强引用

  只要引用存在，垃圾回收器永远不会回收

  Object obj = new Object();

- 软引用

  非必须引用，内存溢出之前进行回收，可以通过以下代码实现

  Object obj = new Object();

  SoftReference<Object> sf = new SoftReference<Object>(obj);

  obj = null;

  sf.get();//有时候会返回null

  这时候sf是对obj的一个软引用，通过sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回null;

  软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

- 弱引用

  第二次垃圾回收时回收，可以通过如下代码实现

  Object obj = new Object();
  WeakReference<Object> wf = new WeakReference<Object>(obj);
  obj = null;
  wf.get();//有时候会返回null
  wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾

  弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。

  弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的isEnQueued方法返回对象是否被垃圾回收器标记为即将回收的垃圾

- 虚引用（幽灵/幻影引用）

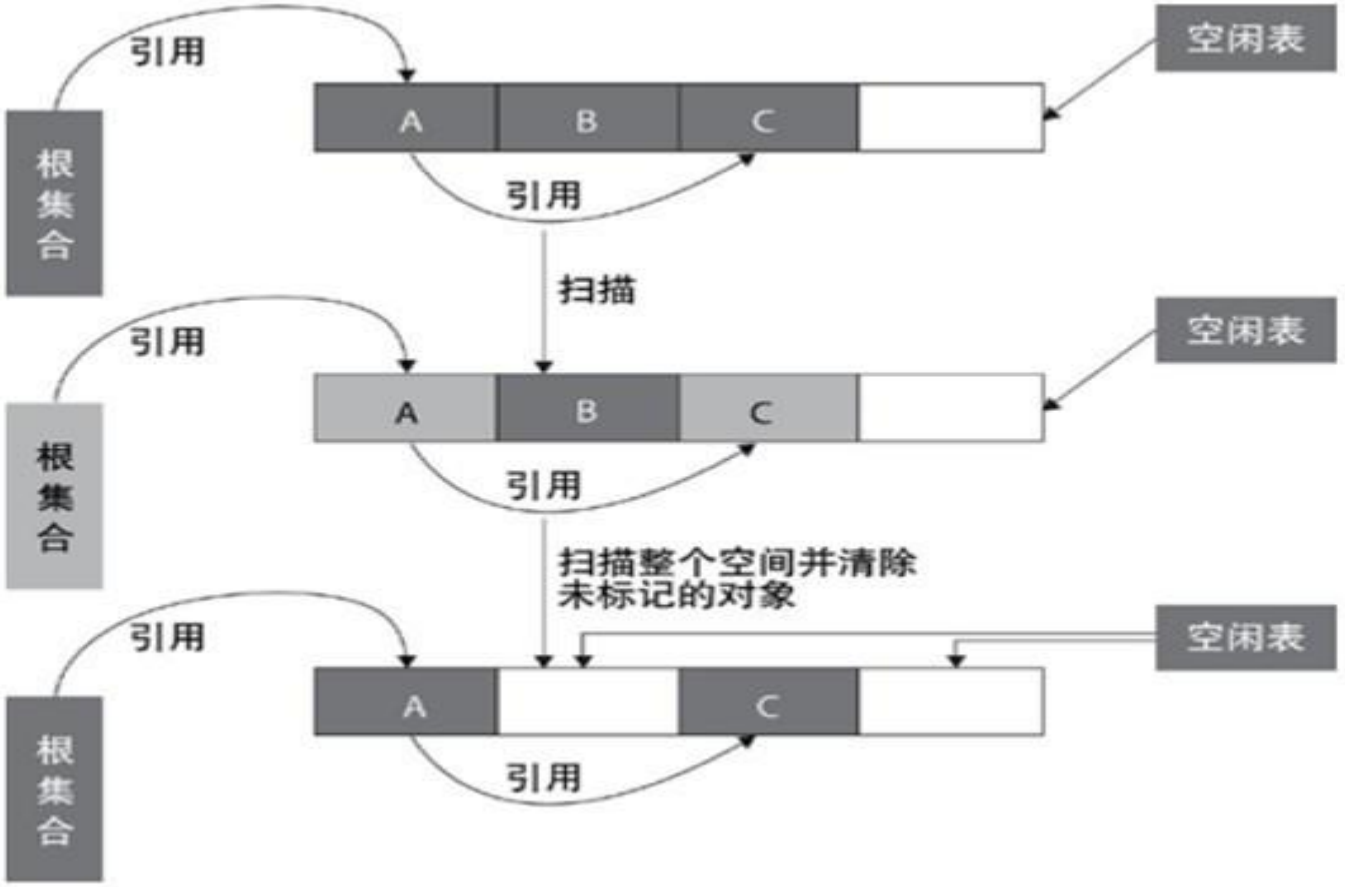
  垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

  Object obj = new Object();
  PhantomReference<Object> pf = new PhantomReference<Object>(obj);
  obj=null;
  pf.get();//永远返回null
  pf.isEnQueued();//返回从内存中已经删除

  虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。

  虚引用主要用于检测对象是否已经从内存中删除。

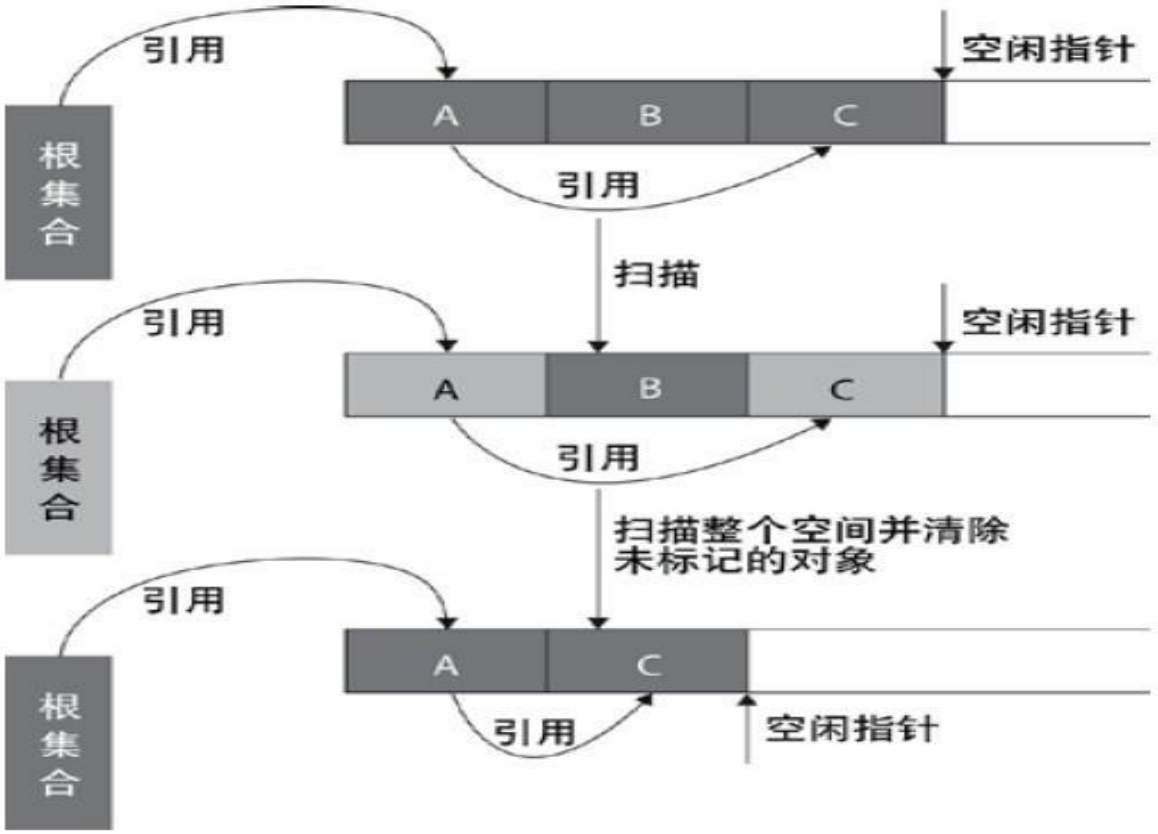
标记清除



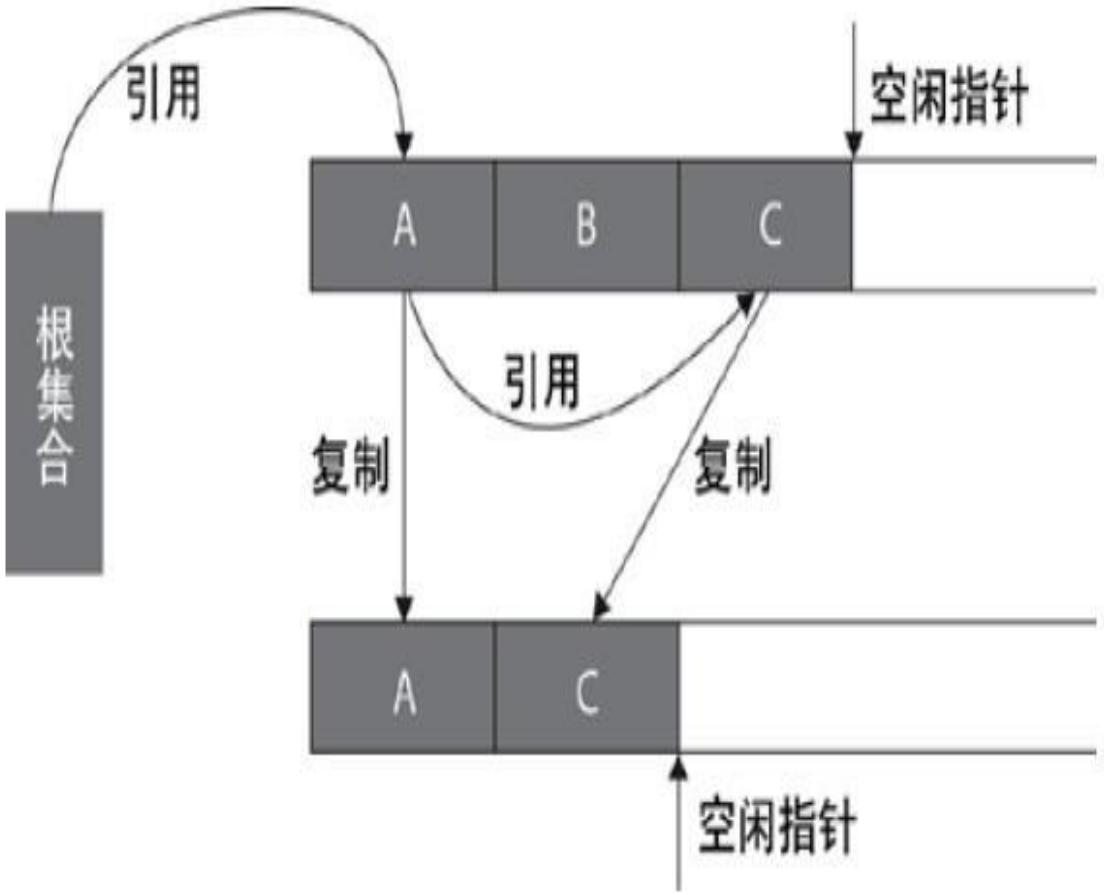
标记-清除算法采用从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，如上图所示。

标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片！

标记整理算法

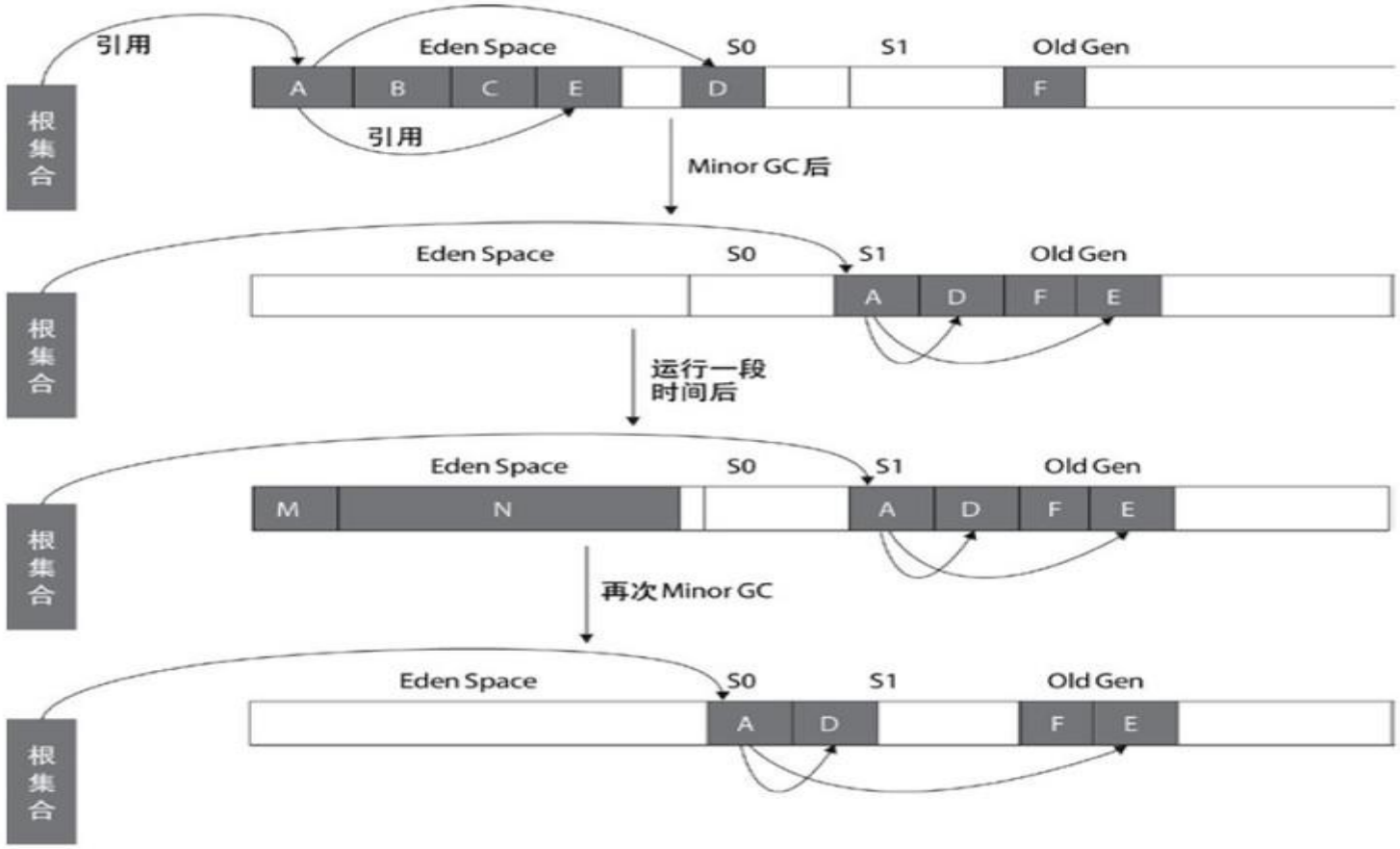


标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题



- 复制算法采用从根集合扫描，并将存活对象复制到一块新的，没有使用过的空间中，这种算法当控件存活的对象比较少时，极为高效，但是带来的成本是需要一块内存交换空间用于进行对象的移动。也就是我们前面提到的

分代算法

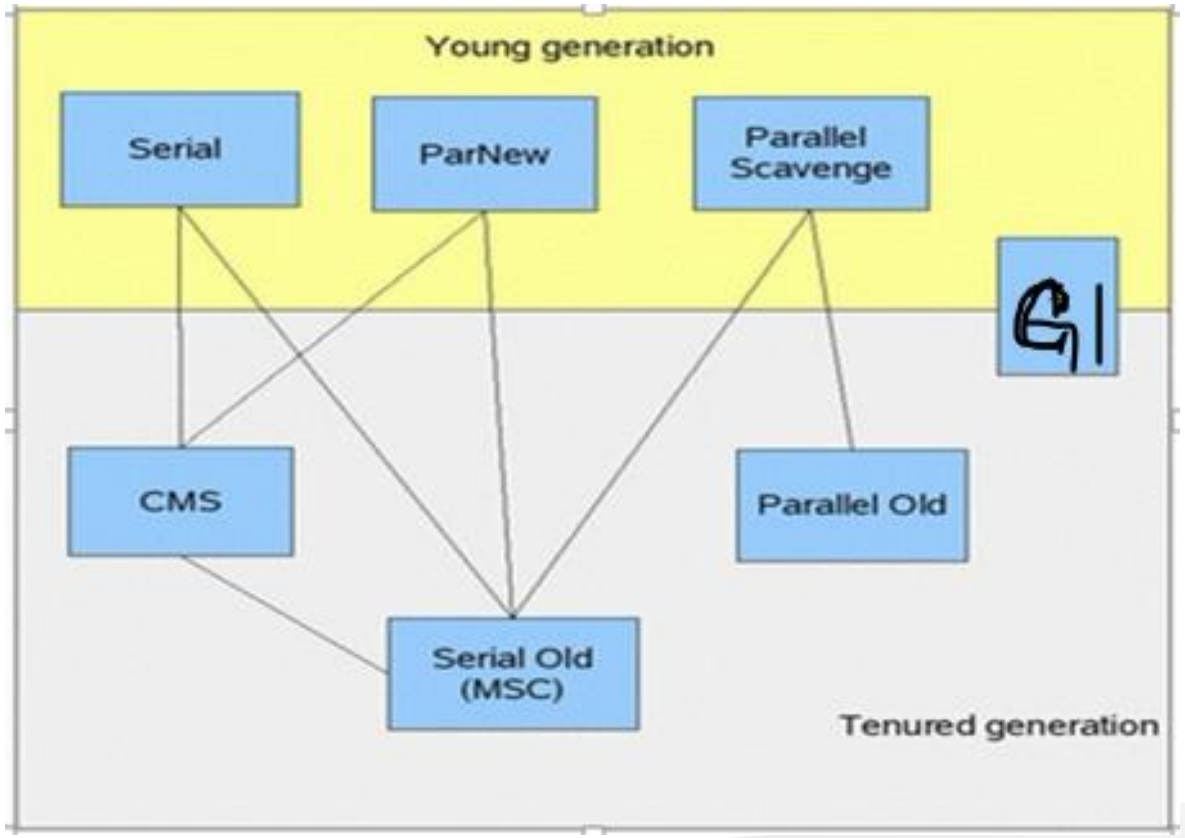


- JVM为了优化内存的回收，进行了分代回收的方式，对于新生代内存的回收（minor GC）主要采用复制算法

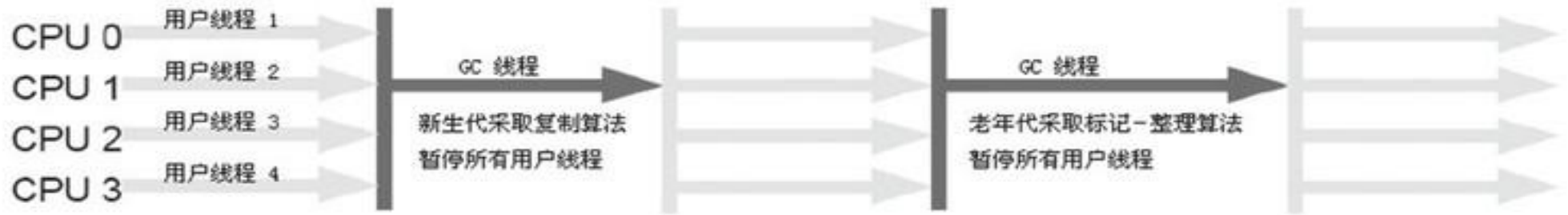
- 将堆空间划分成连续的小空间，每个小空间独立使用，独立回收，该算法的好处是可以控制一次回收多少区间

- Serial收集器
- ParNew收集器
- ParallelScavenge
- SerialOld
- ParallelOld
- CMS
- G1

GC算法整合


Young generation
Serial
ParNew
Parallel Scavenge
G1
CMS
Parallel Old
Serial Old (MSC)
Tenured generation

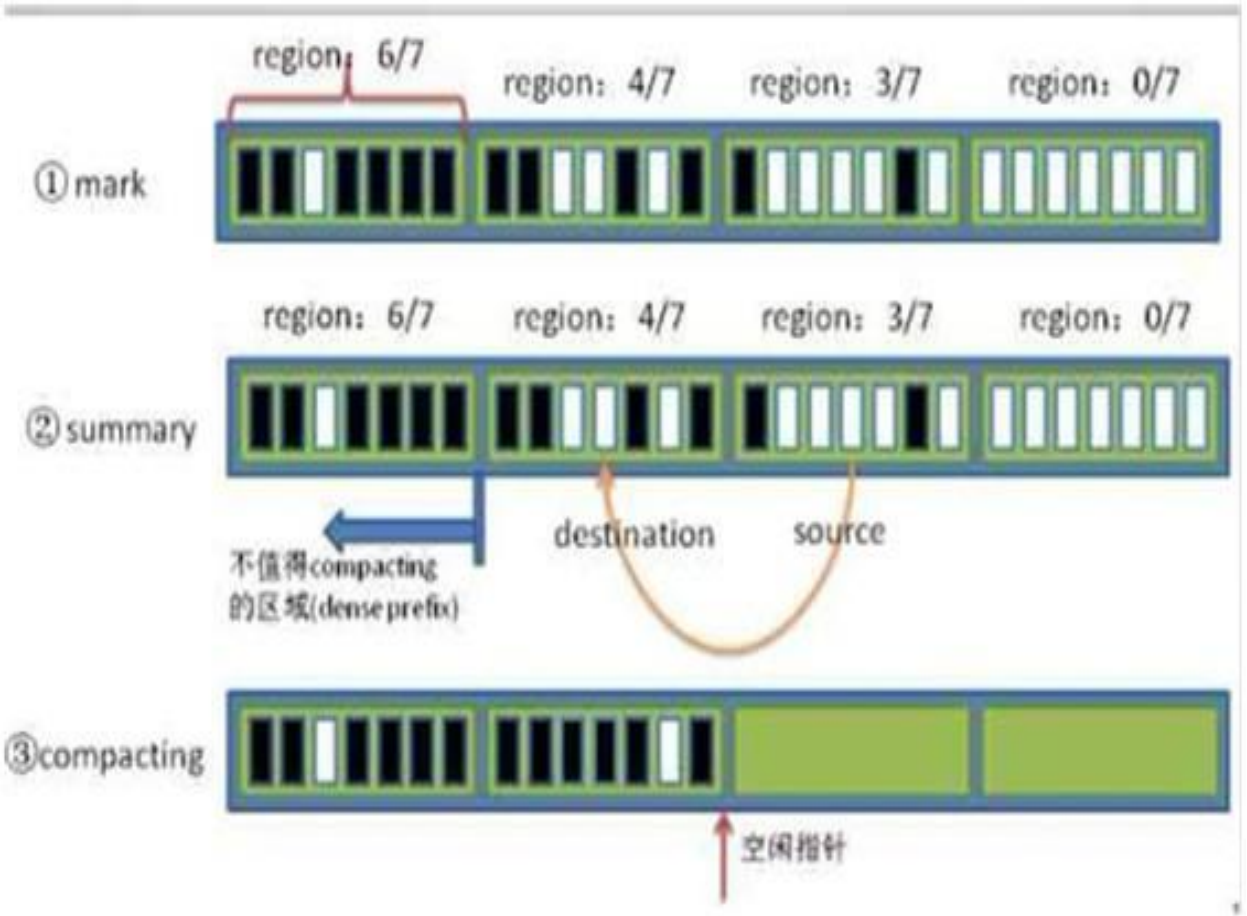

新生代可用的GC
串行GC (Serial Copying)
并行回收GC (Parallel Scavenge)
并行GC (ParNew)
旧生代可用的GC
串行GC (Serial MSC)
并行GC (Parallel MSC)
并发GC (CMS)



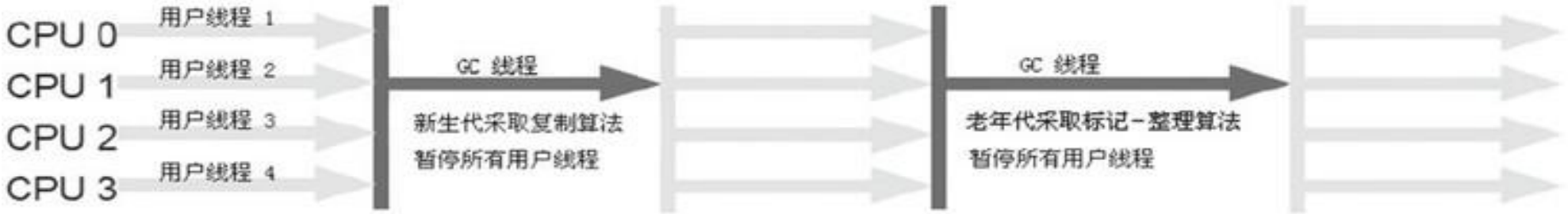
Serial收集器是历史最悠久的一个回收器，JDK1.3之前广泛使用这个收集器，目前也是ClientVM下 ServerVM 4核4GB以下机器的默认垃圾回收器。串行收集器并不是只能使用一个CPU进行收集，而是当JVM需要进行垃圾回收的时候，需要中断所有的用户线程，直到它回收结束为止，因此又号称“Stop The World”的垃圾回收器。注意，JVM中文名称为java虚拟机，因此它就像一台虚拟的电脑一样在工作，而其中的每一个线程就被认为是JVM的一个处理器，因此大家看到图中的CPU0、CPU1实际为用户的线程。

- 串行回收方式适合低端机器，是Client模式下的默认收集器，对CPU和内存的消耗不高，适合用户交互比较少，后台任务较多的系统。Serial收集器默认新旧生代的回收器搭配为Serial+ SerialOld

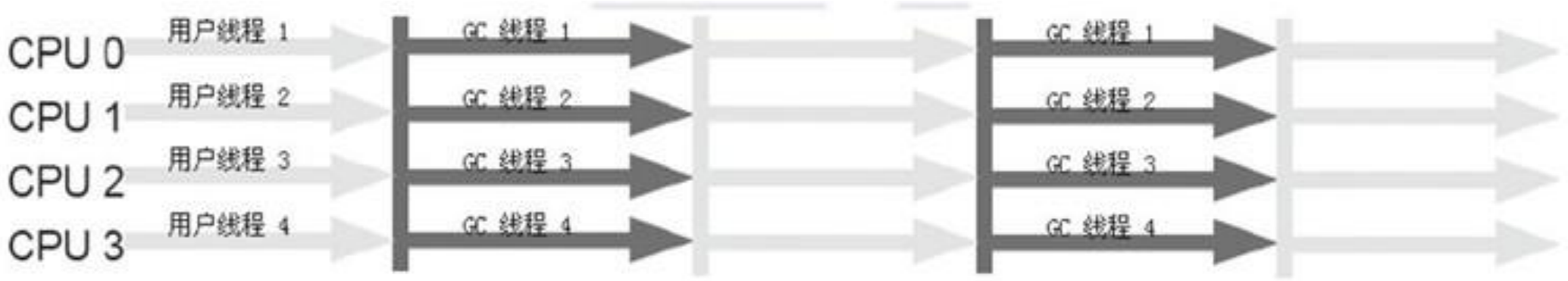
- ParNew收集器其实就是多线程版本的Serial收集器。
- 同样有Stop The World的问题，他是多CPU模式下的首选回收器（该回收器在单CPU的环境下回收效率远远低于Serial收集器），也是Server模式下的默认收集器



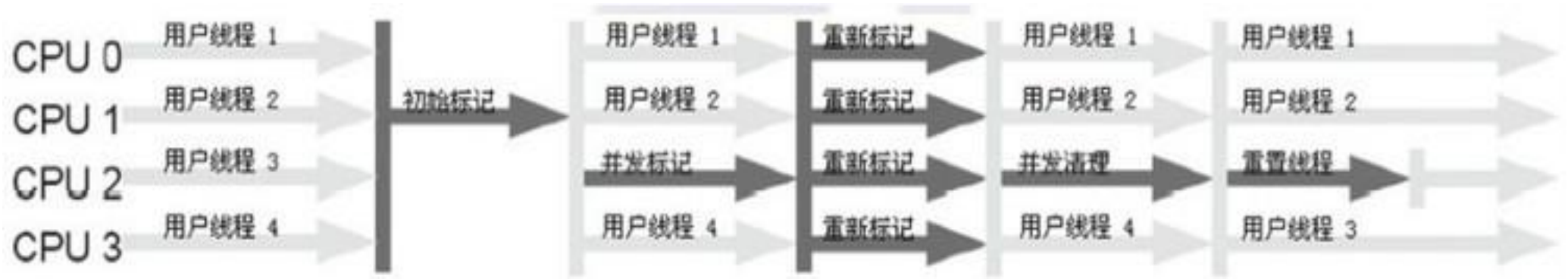
- ParallelScavenge又被称为是吞吐量优先的收集器，ParallelScavenge所提到的吞吐量=程序运行时间/(JVM执行回收的时间+程序运行时间),假设程序运行了100分钟，JVM的垃圾回收占用1分钟，那么吞吐量就是99%。在当今网络告诉发达的今天，良好的响应速度是提升用户体验的一个重要指标，多核并行云计算的发展要求程序尽可能的使用CPU和内存资源，尽快的计算出最终结果，因此在交互不多的云端，比较适合使用该回收器



- SerialOld是旧生代Client模式下的默认收集器，单线程执行；在JDK1.6之前也是ParallelScvenge回收新生代模式下旧生代的默认收集器，同时也是并发收集器CMS回收失败后的备用收集器



- ParallelOld是老生代并行收集器的一种，使用标记整理算法、是老生代吞吐量优先的一个收集器。这个收集器是JDK1.6之后刚引入的一款收集器，我们看之前那个图之间的关联关系可以看到，早期没有ParallelOld之前，吞吐量优先的收集器老生代只能使用串行回收收集器，大大的拖累了吞吐量优先的性能，自从JDK1.6之后，才能真正做到较高效率的吞吐量优先。

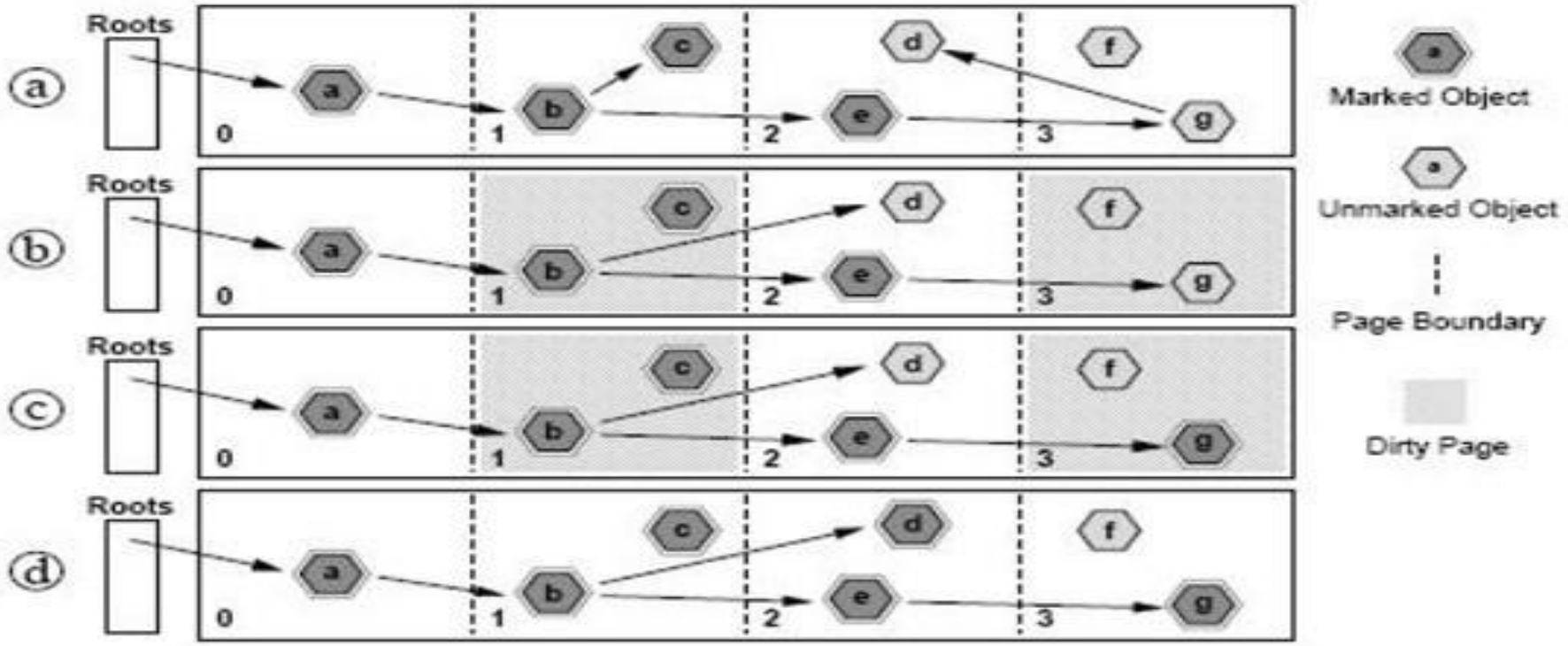


- CMS又称响应时间优先(最短回收停顿)的回收器，使用并发模式回收垃圾，使用标记-清除算法，CMS对CPU是非常敏感的，它的回收线程数=（CPU+3）/4，因此当CPU是2核的实惠，回收线程将占用的CPU资源的50%，而当CPU核心数为4时仅占用25%

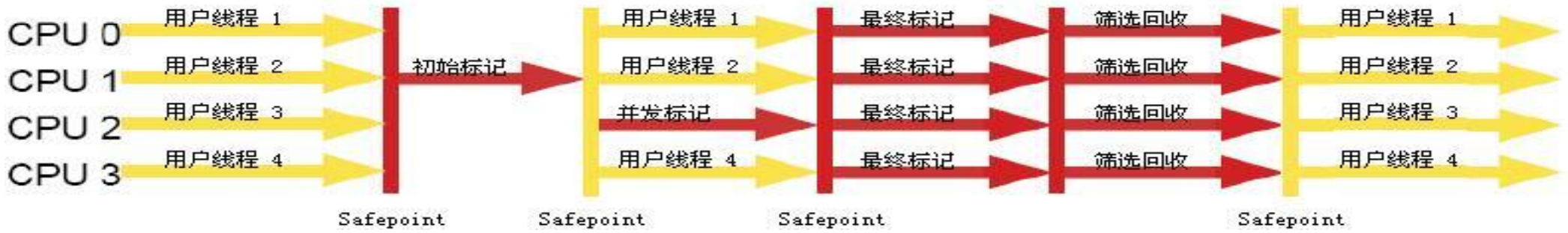
- 初始标记（CMS initial mark）
- 并发标记（CMS concurrent mark）
- 重新标记（CMS remark）
- 并发清除（CMS concurrent sweep）

在初始标记的时候，需要中断所有用户线程，在并发标记阶段，用户线程和标记线程并发执行，而在这个过程中，随着内存引用关系的变化，可能会发生原来标记的对象被释放，进而引发新的垃圾，因此可能会产生一系列的浮动垃圾，不能被回收。

CMS 为了确保能够扫描到所有的对象，避免在Initial Marking 中还有未标识到的对象，采用的方法为找到标记了的对象，并将这些对象放入Stack 中，扫描时寻找此对象依赖的对象，如果依赖的对象的地址在其之前，则将此对象进行标记，并同时放入Stack 中，如依赖的对象地址在其之后，则仅标记该对象。在进行Concurrent Marking 时minor GC 也可能会同时进行，这个时候很容易造成旧生代对象引用关系改变，CMS 为了应对这样的并发现象，提供了一个Mod Union Table 来进行记录，在这个Mod Union Table中记录每次minor GC 后修改了的Card 的信息。这也是ParallelScavenge不能和CMS一起使用的原因。



在运行回收过后，c就变成了浮动垃圾。
由于CMS会产生浮动垃圾，当回收过后，浮动垃圾如果产生过多，同时因为使用标记-清除算法会产生碎片，可能会导致回收过后的连续空间仍然不能容纳新生代移动过来或者新创建的大资源，因此会导致CMS回收失败，进而触发另外一次FULL GC，而这时候则采用SerialOld进行二次回收。
同时CMS因为可能产生浮动垃圾，而CMS在执行回收的同时新生代也有可能在进行回收操作，为了保证旧生代能够存放新生代转移过来的数据，CMS在旧生代内存到达全部容量的68%就触发了CMS的回收！



- 可以回收新生代也可以回收旧生代，SunHotSpot 1.6u14以上EarlyAccess版本加入了这个回收器，sun公司预期SunHotSpot1.7发布正式版，他是商用高性能垃圾回收器，通过重新划分内存区域，整合优化CMS，同时注重吞吐量和响应时间，但是杯具的是被oracle收购之后这个收集器属于商用收费收集器，因此目前基本上没有人使用

- 堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的
- JVM为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间TLAB（Thread Local Allocation Buffer），其大小由JVM根据运行的情况计算而得，在TLAB上分配对象时不需要加锁，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配
- TLAB仅作用于新生代的Eden Space，因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效。

1. 与串行回收器相关的参数

- -XX:+UseSerialGC：在新生代和老年代使用串行收集器。
- -XX:SurvivorRatio：设置 eden 区大小和 survivior 区大小的比例。
- -XX:PretenureSizeThreshold：设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。
- -XX:MaxTenuringThreshold：设置对象进入老年代的年龄的最大值。每一次 Minor GC 后，对象年龄就加 1。任何大于这个年龄的对象，一定会进入老年代。

与并行 GC 相关的参数

-XX:+UseParNewGC：在新生代使用并行收集器。

-XX:+UseParallelOldGC：老年代使用并行回收收集器。

-XX:ParallelGCThreads：设置用于垃圾回收的线程数。通过情况下可以和 CPU 数量相等，但在 CPU 数量比较多的情况下，设置相对较小的数值也是合理的。

-XX:MaxGCPauseMillis：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMillis 以内。

-XX:GCTimeRatio：设置吞吐量大小。它的值是一个 0 到 100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 $1/(1+n)$的时间用于垃圾收集。

-XX:+UseAdaptiveSizePolicy：打开自适应 GC 策略。在这种模式下，新生代的大小、eden 和 survivior 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。

与 CMS 回收器相关的参数

-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用 CMS+串行收集器。

-XX:ParallelCMSThreads：设定 CMS 的线程数量。

-XX:CMSInitiatingOccupancyFraction：设置 CMS 收集器在老年代空间被使用多少后触发，默认为 68%。

-XX:+UseCMSCompactAtFullCollection：设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片的整理。

-XX:CMSFullGCsBeforeCompaction：设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。

-XX:+CMSClassUnloadingEnabled：允许对类元数据区进行回收。

-XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动 CMS 回收（前提是-XX:+CMSClassUnloadingEnabled 激活了）。

-XX:UseCMSInitiatingOccupancyOnly：表示只在到达阈值的时候才进行 CMS 回收。

-XX:+CMSIncrementalMode：使用增量模式，比较适合单 CPU。增量模式在 JDK 8 中标记为废弃，并且将在 JDK 9 中彻底移除。

与 G1 回收器相关的参数

-XX:+UseG1GC：使用 G1 回收器。

-XX:MaxGCPauseMillis：设置最大垃圾收集停顿时间。

-XX:GCPauseIntervalMillis：设置停顿间隔时间。

TLAB 相关

-XX:+UseTLAB：开启 TLAB 分配。

-XX:+PrintTLAB：打印 TLAB 相关分配信息。

-XX:TLABSize：设置 TLAB 大小。

-XX:+ResizeTLAB：自动调整 TLAB 大小。

其他参数

-XX:+DisableExplicitGC：禁用显式 GC。

-XX:+ExplicitGCInvokesConcurrent：使用并发方式处理显式 GC。

- -XX:AutoBoxCacheMax=20000

Integer i = 3;这语句有着 int自动装箱成Integer的过程，JDK默认只缓存 -128 ~ +127的 int 和 long，超出范围的数字就要即时构建新的Integer对象。

- 启动时访问并置零内存页面-XX:+AlwaysPreTouch

启动时就把参数里说好了的内存全部添一遍，可能启动时慢上一点，但后面访问时会更流畅，比如页面会连续分配，比如不会在晋升新生代到老生代时才去访问页面使得GC停顿时间加长。不过这选项对大堆才会更有感觉一点。

- -XX:+PerfDisableSharedMem
- Cassandra家的一个参数，一直没留意，直到发生高IO时的JVM停顿。原来JVM经常会默默的在/tmp/hperf 目录写上一点statistics数据，如果刚好遇到PageCache刷盘，把文件阻塞了，就不能结束这个Stop the World的安全点了。用此参数可以禁止JVM写statistics数据，代价是jps, jstat 用不了，只能用JMX取数据。有时用JMX取新生代老生代使用百分比还真没jstat方便。详见The Four Month Bug: JVM statistics cause garbage collection pauses

-XX:+HeapDumpOnOutOfMemoryError and -XX:HeapDumpPath

可以通过设置-XX:HeapDumpPath=<path>来改变默认的堆内存快照生成路径，<path>可以是相对或者绝对路径。堆内存快照文件有可能很庞大，特别是当内存溢出错误发生的时候。因此，我们推荐将堆内存快照生成路径指定到一个拥有足够磁盘空间的地方。

-XX:OnOutOfMemoryError

当内存溢发生时，我们甚至可以可以执行一些指令，比如发个E-mail通知管理员或者执行一些清理工作。通过-XX:OnOutOfMemoryError 这个参数我们可以做到这一点，这个参数可以接受一串指令和它们的参数。在这里，我们将不会深入它的细节，但我们提供了它的一个例子。在下面的例子中，当内存溢出错误发生的时候，我们会将堆内存快照写到/tmp/heapdump.hprof 文件并且在JVM的运行目录执行脚本cleanup.sh，如：

java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof -XX:OnOutOfMemoryError ="sh ~/cleanup.sh" MyApp

-XX:InitialCodeCacheSize and -XX:ReservedCodeCacheSize

JVM一个有趣的，但往往被忽视的内存区域是“代码缓存”，它是用来存储已编译方法生成的本地代码。代码缓存确实很少引起性能问题，但是一旦发生其影响可能是毁灭性的。如果代码缓存被占满，JVM会打印出一条警告消息，并切换到interpreted-only 模式：JIT编译器被停用，字节码将不再会被编译成机器码。因此，应用程序将继续运行，但运行速度会降低一个数量级，直到有人注意到这个问题。就像其他内存区域一样，我们可以自定义代码缓存的大小。相关的参数是-XX:InitialCodeCacheSize 和-XX:ReservedCodeCacheSize，它们的参数和上面介绍的参数一样，都是字节值

-XX:+UseCodeCacheFlushing

如果代码缓存不断增长，例如，因为热部署引起的内存泄漏，那么提高代码的缓存大小只会延缓其发生溢出。为了避免这种情况的发生，我们可以尝试一个有趣的新参数：当代码缓存被填满时让JVM放弃一些编译代码。通过使用-XX:+UseCodeCacheFlushing 这个参数，我们至少可以避免当代码缓存被填满的时候JVM切换到interpreted-only 模式。不过，我仍建议尽快解决代码缓存问题发生的根本原因，如找出内存泄漏并修复它

BONC

- Linux监控工具
- JDK工具

- Top
- Vmstat
- Iostat
- Pidstat
- Ps
- Netstat
- Sar

- Jps
- Jstat
- Jinfo
- Jmap
- Jhat
- Jstack
- Jstatd
- Jconsole
- Uisual VM

JDK工具

S0C：年轻代中第一个survivor（幸存区）的容量 (字节)
S1C：年轻代中第二个survivor（幸存区）的容量 (字节)
S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (字节)
S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (字节)
EC：年轻代中Eden的容量 (字节)
EU：年轻代中Eden目前已使用空间 (字节)
OC：Old代的容量 (字节)
OU：Old代目前已使用空间 (字节)
PC：Perm(持久代)的容量 (字节)
PU：Perm(持久代)目前已使用空间 (字节)
YGC：从应用程序启动到采样时年轻代中gc次数
YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)
FGC：从应用程序启动到采样时old代(全gc)gc次数
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)
GCT：从应用程序启动到采样时gc用的总时间(s)
NGCMN：年轻代(young)中初始化(最小)的大小 (字节)
NGCMX：年轻代(young)的最大容量 (字节)
NGC：年轻代(young)中当前的容量 (字节)
OGCMN：old代中初始化(最小)的大小 (字节)
OGCMX：old代的最大容量 (字节)
OGC：old代当前新生成的容量 (字节)
PGCMN：perm代中初始化(最小)的大小 (字节)

PGCMX：perm代的最大容量 (字节)
PGC：perm代当前新生成的容量 (字节)
S0：年轻代中第一个survivor（幸存区）已使用的占当前容量百分比
S1：年轻代中第二个survivor（幸存区）已使用的占当前容量百分比
E：年轻代中Eden（伊甸园）已使用的占当前容量百分比
O：old代已使用的占当前容量百分比
P：perm代已使用的占当前容量百分比
S0CMX：年轻代中第一个survivor（幸存区）的最大容量 (字节)
S1CMX：年轻代中第二个survivor（幸存区）的最大容量 (字节)
ECMX：年轻代中Eden（伊甸园）的最大容量 (字节)
DSS：当前需要survivor（幸存区）的容量 (字节)（Eden区已满）
TT：持有次数限制
MTT：最大持有次数限制

- **jps**(Java Virtual Machine Process Status Tool)
  查看所有的jvm进程，包括进程ID，进程启动的路径等等。
- -l main完整路径
- -m 传递给main参数
- -v vm参数
- -q 只输出pid

BONC

• jstack(Java Stack Trace)

- ① 观察jvm中当前所有线程的运行情况和线程当前状态。
- ② 系统崩溃了？如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。
- ③ 系统hung住了？jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。
  - -F当’jstack [-l] pid’没有相应的时候强制打印栈信息
  - -l长列表. 打印关于锁的附加信息,例如属于java.util.concurrent的ownable synchronizers列表.
  - -m打印java和native c/c++框架的所有栈信息.
  - -h | -help打印帮助信息
  - pid 需要被打印配置信息的java进程id,可以用jps查询

-class：统计类装载器的行为
-compiler：统计HotSpot Just-in-Time编译器的行为
-gc：统计堆各个分区的使用情况
-gccapacity：统计新生区，老年区，permanent区的heap容量情况
-gccause：统计最后一次gc和当前gc的原因
-gcnew：统计gc时，新生代的情况
-gcnewcapacity：统计新生代大小和空间
-gcold：统计老年代和永久代的行为
-gcoldcapacity：统计老年代大小
-gcpermcapacity：统计永久代大小
-gcutil：统计gc时，heap情况
-printcompilation：HotSpot编译方法统计

```
#每隔1秒监控一次，一共做10次
  jstat -class 17970 1000 10
##########################################
[root@lq225 conf]# jstat -class 2058 1000 10
Loaded  Bytes  Unloaded  Bytes     Time
  1697  3349.5        0     0.0       1.79
  1697  3349.5        0     0.0       1.79
  1697  3349.5        0     0.0       1.79
  1697  3349.5        0     0.0       1.79
  ..................................................
######################### 术语分隔符 #########################
#Loaded 类加载数量
#Bytes  加载的大小（k）
#Unloaded 类卸载的数量
#Bytes 卸载的大小（k）
#Time 时间花费在执行类加载和卸载操作
```

```
Compiled Failed Invalid   Time   FailedType FailedMethod
     302      0       0     1.27          0
..................................................
########################## 术语分隔符 ##########################
#Compiled 编译任务的执行次数
#Failed   编译任务的失败次数
#Invalid  编译任务无效的次数
#Time     编译任务花费的时间
#FailedType 最后一次编译错误的类型
#FailedMethod 最后一次编译错误的类名和方法
```

```
#每隔2秒监控一次，共20次
  jstat -gc 2058 2000 20
##############################
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC
8704.0 8704.0 805.5   0.0   69952.0  64174.5   174784.0    2644.5   16384.0 10426.7      2    0.034
8704.0 8704.0 805.5   0.0   69952.0  64174.5   174784.0    2644.5   16384.0 10426.7      2    0.034
8704.0 8704.0 805.5   0.0   69952.0  64174.5   174784.0    2644.5   16384.0 10426.7      2    0.034
......................................
###################### 术语分隔符 ######################
#S0C 生还者区0 容量(KB)
#S1C 生还者区1 容量(KB)
#S0U 生还者区0 使用量(KB)
#S1U 生还者区1 使用量(KB)
#EC 伊甸园区容量(KB)
#EU 伊甸园区使用量(KB)
#OC 老年区容量(KB)
#OU 老年区使用量(KB)
#PC 永久区容量(KB)
#PU 永久区使用量(KB)
#YGC 新生代GC次数
#YGCT 新生代GC时间
#FGC full GC 事件的次数
#FGCT full GC的时间
#GCT 总GC时间
```

```
NGCMN    NGCMX     NGC     S0C   S1C       EC       OGCMN      OGCMX      OGC         OC       PGCMN
131072.0 131072.0 131072.0 13056.0 13056.0 104960.0   393216.0   393216.0   393216.0   393216.0  65536
..................................................................................................
######################## 术语分隔符 ########################
#NGCMN 最小新生代容量(KB)
#NGCMX 最大新生代容量(KB)
#NGC 当前新生代容量(KB)
#S0C 当前生存者0区容量(KB)
#S1C 当前生存者1区容量(KB)
#OGCMN 老年代最小容量(KB)
#OGCMX 老年代最大容量(KB)
#OGC 当前老年代容量(KB).
#OC 当前老年代? Current old space capacity (KB).
#PGCMN 永久区最小容量(KB)
#PGCMX 永久区最大容量(KB)
#PGC 当前永久区容量(KB).
#PC 当前永久区? Current Permanent space capacity (KB).
#YGC young GC事件的次数
#FGC Full GC次数
```

-gccause

```
S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT    LGCC                 GCC
  0.00  99.84  12.76   0.92  46.23      1    0.016     0    0.000    0.016 unknown GCCause      No GC
.............................................
######################## 术语分隔符 ########################
#S0 年轻代中第一个survivor（幸存区）已使用的占当前容量百分比
#S1 年轻代中第二个survivor（幸存区）已使用的占当前容量百分比
#E 年轻代中Eden（伊甸园）已使用的占当前容量百分比
#O old代已使用的占当前容量百分比
#P perm代已使用的占当前容量百分比
#YGC 从应用程序启动到采样时年轻代中gc次数
#FGC 从应用程序启动到采样时old代(全gc)gc次数
#FGCT 从应用程序启动到采样时old代(全gc)gc所用时间(s)
#GCT 从应用程序启动到采样时gc用的总时间(s)
#LGCC 最后一次GC的原因
#GCC 当前GC的原因
```

```
#每隔1秒监控一次，共10次
jstat -gcutil 2058 1000 10
###################################
[root@lq225 conf]# jstat -gcutil 2058 1000 10
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  9.25   0.00  96.73   1.51  63.64       2    0.034      0    0.000    0.034
  9.25   0.00  96.73   1.51  63.64       2    0.034      0    0.000    0.034
  9.25   0.00  96.73   1.51  63.64       2    0.034      0    0.000    0.034
  9.25   0.00  96.73   1.51  63.64       2    0.034      0    0.000    0.034
```

- -dump:[live,]format=b,file=<filename> 使用hprof二进制形式,输出jvm的heap内容到文件. live子选项是可选的，假如指定live选项,那么只输出活的对象到文件.
- -finalizerinfo 打印正等候回收的对象的信息.
- -heap 打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况.
- -histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”*”. 如果live子参数加上后,只统计活的对象数量.
- -permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来.
- -F 强迫.在pid没有相应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效.
- -h | -help 打印辅助信息
- -J 传递参数给jmap启动的jvm.
- pid 需要被打印配相信息的java进程id.

```
jmap -histo  2058
##############################
 num     #instances         #bytes  class name
----------------------------------------------
   1:           206        3585312  [I
   2:         19621        2791880  <constMethodKlass>
   3:         19621        2520048  <methodKlass>
   4:         21010        2251616  [C
----------------------------------------------
```

```
#生成的文件可以使用jhat工具进行分析，在OOM（内存溢出）时，分析大对象，非常有用
jmap -dump:live,format=b,file=data.hprof 2058

#通过使用如下参数启动JVM，也可以获取到dump文件：
 -XX:+HeapDumpOnOutOfMemoryError
 -XX:HeapDumpPath=./java_pid<pid>.hprof

#如果在虚拟机中导出的heap信息文件可以拿到WINDOWS上进行分析，可以查找诸如内存方面的问题，可以这么做：
jhat data.hprof
#执行成功后，访问http://localhost:7000即可查看内存信息。（首先把7000端口打开）
```

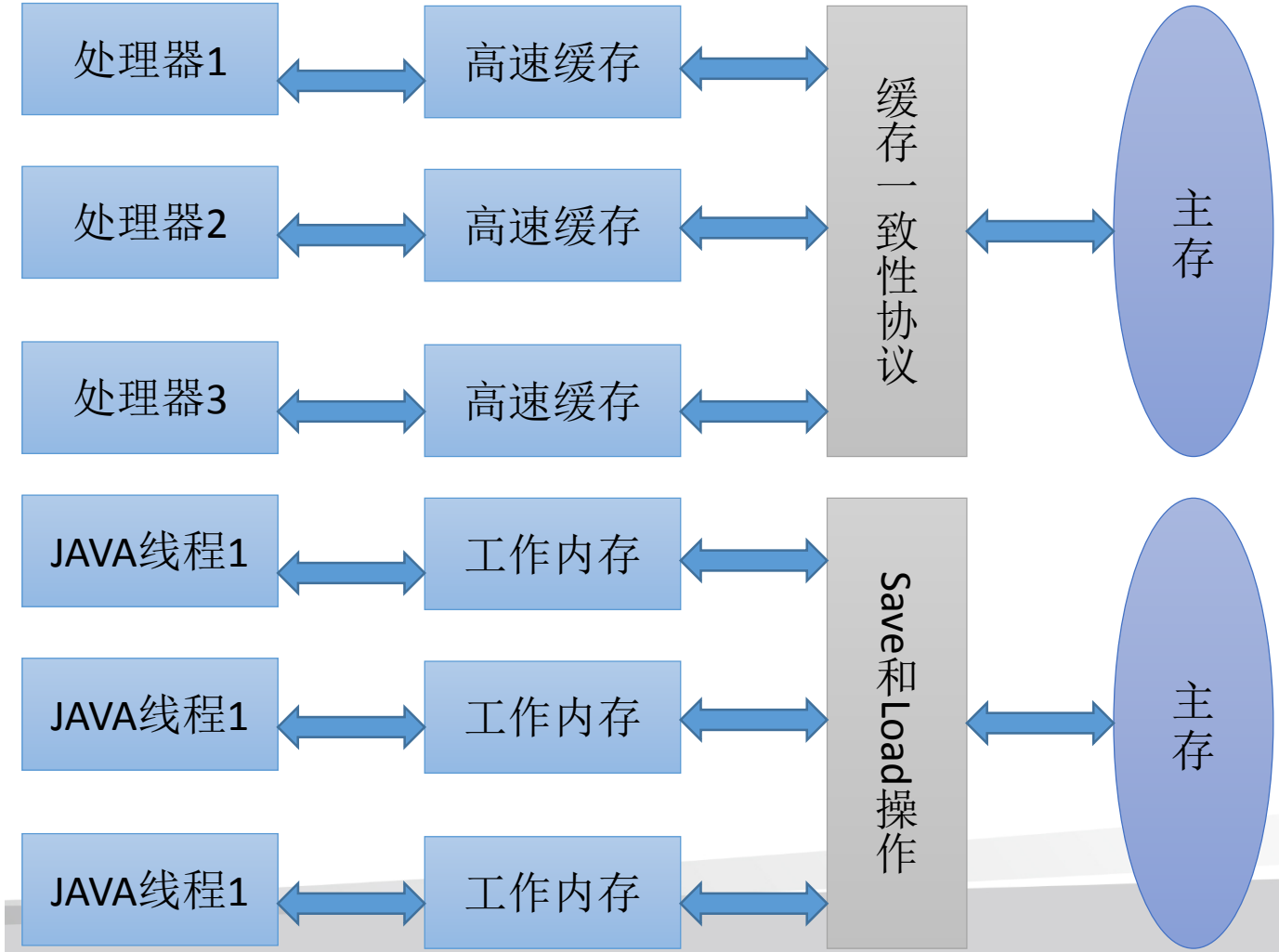
```
jinfo <option> <pid>
```

其中 option 可以为以下信息。

- -flag <name>：打印指定 Java 虚拟机的参数值。
- -flag [+|-]<name>：设置指定 Java 虚拟机参数的布尔值。
- -flag <name>=<value>：设置指定 Java 虚拟机参数的值。

- 远程监控
- 需开启java安全策略
- 如：配置文件jstatd.all.policy
- grant codebase “${JAVA_HOME}/lib/tools.jar” {
- permission java.security.AllPermission;
- }
- Jstatd –J-Djava.security.policy=jstatd.all.policy

JMM(Java Memory Model)：屏蔽硬件与操作系统的内存访问差异，以实现JAVA程序在各种平台下都达到一致的内存访问效果，其工作方式和cpu类似



如果你是3年及其以上，你得知道……

BONC

ArrayList实现原理？
HashSet实现原理？
HashMap实现原理？
TreeMap实现原理？
ConcurrentHashMap实现原理？
Collections.sort实现原理？
AVL?
Volatile
Happen-before
线程。。。。。。