# JVM虚拟机

## JVM概述

### 什么是JVM？

JVM是Java Virtual Machine的缩写，通常称为java虚拟机，作为Java可以进行一次编写，到处执行（Write once, run anywhere)的平台基础，由JVM帮工程师屏蔽了不同平台的差异性，提供了一致的编写接口。

### 虚拟机

#### 虚拟机始祖：Classic VM

它是“世界上第一款商用Java虚拟机”。

1996年1月23日，Sun公司发布JDK 1.0，Java语言首次拥有了商用的正式运行环境，这个JDK中所带的虚拟机就是Classic VM。这款虚拟机只能使用纯解释器方式来执行Java代码，如果要使用JIT编译器，就必须进行外挂。但是假如外挂了JIT编译器，JIT编译器就完全接管了虚拟机的执行系统，解释器便不再工作了。

由于解释器和编译器不能配合工作，这就意味着如果要使用编译器执行，编译器就不得不对每一个方法、每一行代码都进行编译，而无论它们执行的频率是否具有编译的价值（不具备热点代码探测的功能，只会全部提前编译）。即使用了JIT编译器输出本地代码，执行效率也和传统的C/C++程序有很大差距，因此就会给用户留下Java运行慢的印象。

#### 英雄气短：Exact VM

Sun的虚拟机团队努力去解决Classic VM所面临的各种问题，提升运行效率。在JDK 1.2时，曾在Solaris平台上发布过一款名为Exact VM的虚拟机，它的执行系统已经具备现代高性能虚拟机的雏形：如编译器与解释器混合工作模式、准确式内存管理（虚拟机可以知道内存中某个位置的数据具体是什么类型）等。

譬如内存中有一个32位的整数123456，它到底是一个reference类型指向123456的内存地址还是一个数值为123456的整数，虚拟机有能力分辨出来。

#### Java界巨擘：HotSpot VM

HotSpot VM最初并非由Sun公司开发，最初并非由Sun公司开发，而是由一家名为“Longview Technologies”的小公司设计。

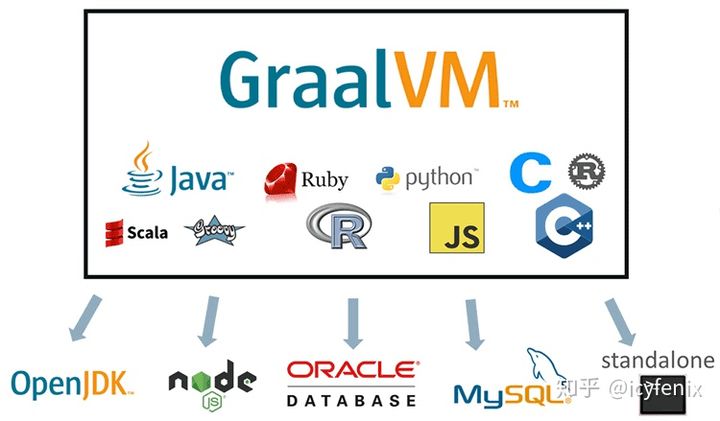
HotSpot VM既继承了Sun之前两款商用虚拟机的优点（如前面提到的准确式内存管理），也有许多自己新的技术优势，如它名称中的HotSpot指的就是它的热点代码探测技术，HotSpot VM的热点代码探测能力可以通过执行计数器找出最具有编译价值的代码，然后通知JIT编译器以方法为单位进行编译。如果一个方法被频繁调用，或方法中有效循环次数很多，将会分别触发标准编译和OSR（栈上替换）编译动作。通过编译器与解释器恰当地协同工作，可以在最优化的程序响应时间与最佳执行性能中取得平衡，而且无须等待本地代码输出才能执行程序，即时编译的时间压力也相对减小，这样有助于引入更多的代码优化技术，输出质量更高的本地代码。

*注：其实两个VM基本上是同时期的独立产品，HotSpot还稍早一些，HotSpot一开始就是准确式GC，而Exact VM之中也有与HotSpot几乎一样的热点探测。为了Exact VM和HotSpot VM哪个成为Sun主要支持的VM产品，在Sun公司内部还有过争论，HotSpot打败Exact并不能算技术上的胜利。*

#### 未来的黑科技：Graal VM

GraalVM 是 Oracle 发布的下世代 Java 虚拟机，2018年4月，Oracle Labs公开，2019年05月才发布了第一个 release 版本，分别有社区版和企业版。

Graal VM被官方称为“Universal VM（通用虚拟机）”和“Polyglot VM（多语言虚拟机）”，这是一个在HotSpot虚拟机基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用，这里“任何语言”包括了Java、Scala、Groovy、Kotlin等基于Java虚拟机之上的语言，还包括了C、C++、Rust等基于LLVM的语言，同时支持其他像JavaScript、Ruby、Python和R语言等等。Graal VM可以无额外开销地混合使用这些编程语言，支持不同语言中混用对方的接口和对象，也能够支持这些语言使用已经编写好的本地库文件。



对于一些本来就不以速度见长的语言运行环境，由于Graal VM本身能够对输入的中间表示进行自动优化，在运行时还能进行即时编译优化，往往使用Graal VM实现能够获得比原生编译器更优秀的执行效率，譬如Graal.js要优于Node.js、Graal.Python要优于CPtyhon，TruffleRuby要优于Ruby MRI，FastR要优于R语言等等。

### JVM性能监控工具

关于性能监控这块的工具有linux的top指令及查看进程相关指令，jinfo，jps，jstat，jmap，jstack，jconsole。

#### top指令

top指令：查看当前所有进程的使用情况，CPU占有率，内存使用情况，服务器负载状态等参数。除此之外它还是个交互命令，使用可参考完全解读top。

#### jps

jps：与linux上的ps类似，用于查看有权访问的虚拟机的进程，可以查看本地运行着几个java程序，并显示他们的进程号。当未指定host id时，默认查看本机jvm进程。

#### jinfo

jinfo：可以输出并修改运行时的java 进程的一些参数。

#### jstat

jstat：可以用来监视jvm内存内的各种堆和非堆的大小及其内存使用量。

#### jstack

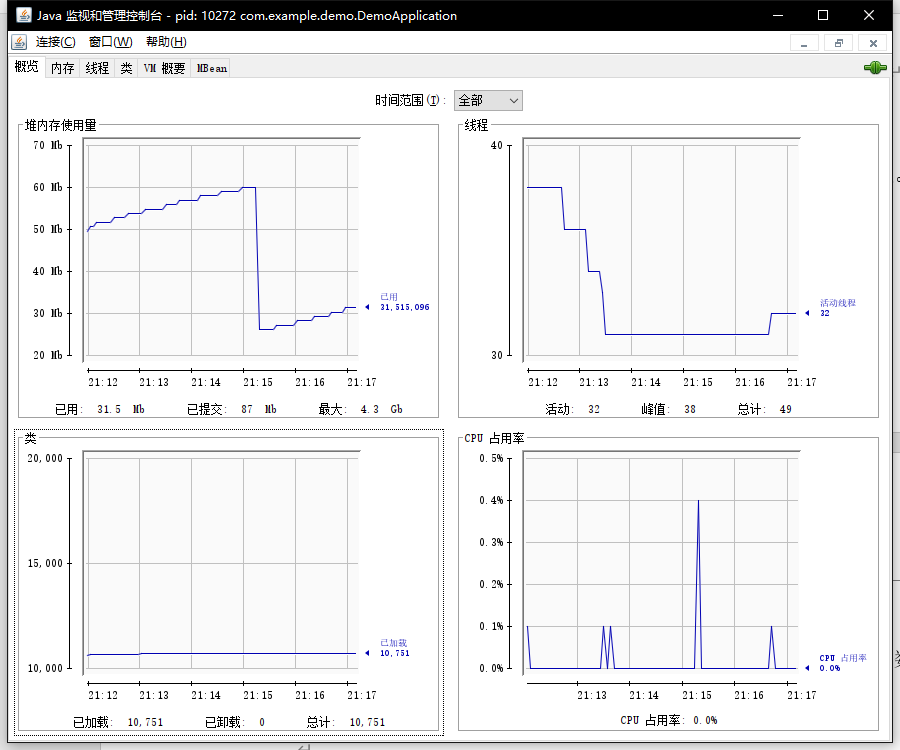
jstack：堆栈跟踪工具，一般用于查看某个进程包含线程的情况。

#### jmap

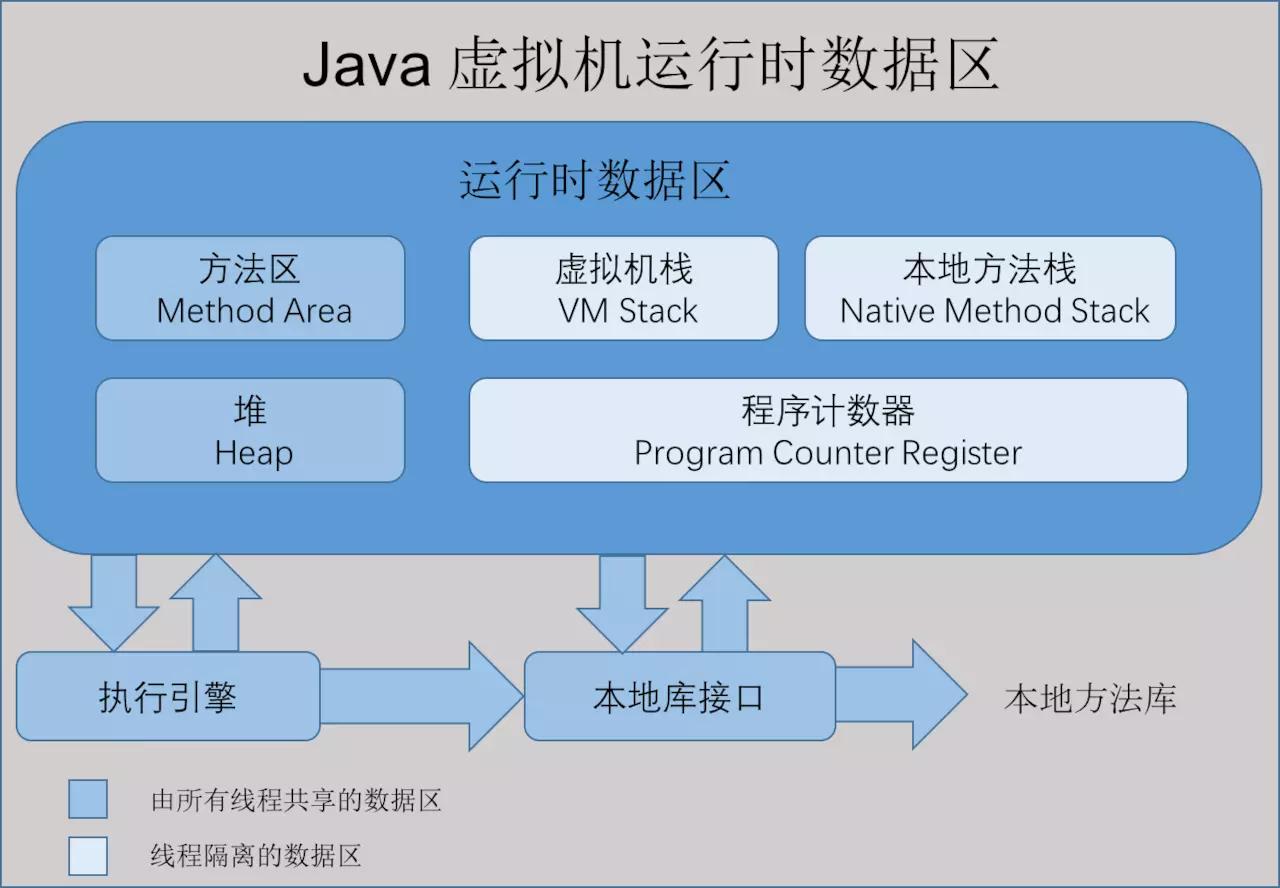
jmap：打印出某个java进程（使用pid）内存内的所有对象的情况。一般用于查看内存占用情况。

#### jconsole

jconsole：一个java GUI监视工具，可以以图表化的形式显示各种数据。并可通过远程连接监视远程的服务器的jvm进程。



## 运行时数据区域概述



### 程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

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

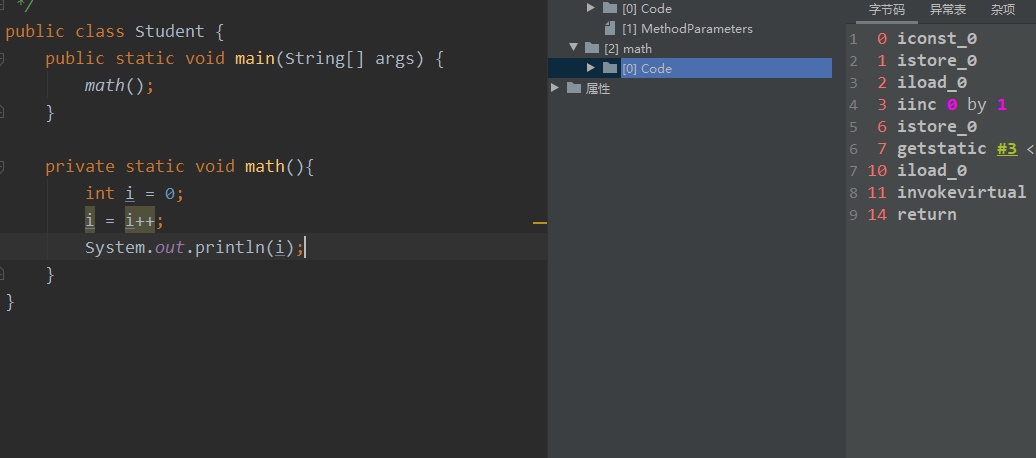
### Java 虚拟机栈

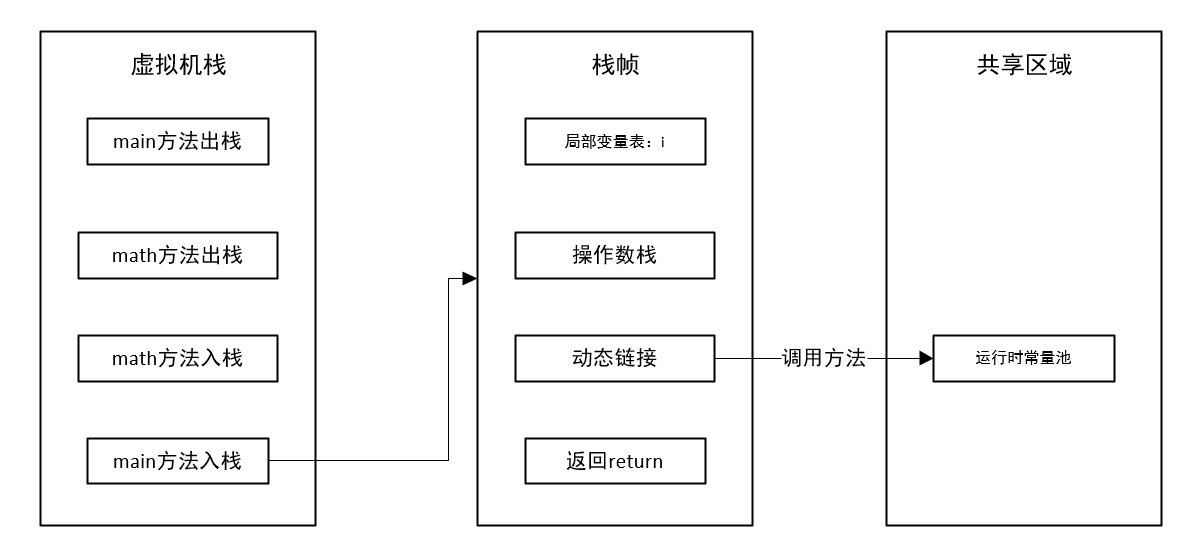
与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

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

在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写人到方法表的Code属性之中，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

代码：





动态链接: 每个栈帧都保存了一个可以指向当前方法所在类的运行时常量池。当前方法中如果需要调用其他方法的时候, 能够从运行时常量池中找到对应的符号引用, 然后将符号引用转换为直接引用,然后就能直接调用对应方法, 这就是动态链接。

静态连接：所有的方法调用在编译好的 Class 文件里面都是一个常量池中的符号引用。运行时，在类加载的解析阶段，会将其中一部分符号引用转化为直接引用。这种在类加载阶段 JVM 就能直接找到真正调用地址，并替换它所对应的符号引用，这个过程我们叫做静态连接，主要包括了静态方法和私有方法两大类。

### 本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其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，而本地方法栈则是为虚拟机使用到的Native 方法服务。

虚拟机规范中对本地方法栈中的方法使用的语言、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

对于大多数应用来说，Java 堆（Java Heap）是Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存**。

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”。

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

### 方法区

方法区（Method Area）与Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。所有的对象在实例化后的整个运行周期内，都被存放在堆内存中。

在JDK1.8以前，HotSpot使用永久代来实现方法区，所以某些场合也认为方法区和永久代是一个概念。

在JDK 6的 时候HotSpot开发团队就有放弃永久代，逐步改为采用本地内存（Native Memory）来实现方法区的计划了，到了JDK 7的HotSpot，已经把原本放在永久代的字符串常量池、静态变量等移出，而到了 JDK 8，终于完全废弃了永久代的概念，改用在本地内存中实现的元空间（Meta- space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

#### 运行时常量池

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

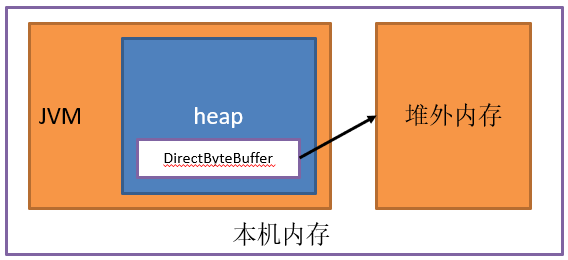
Java 虚拟机对Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于运行时常量池，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

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

既然运行时常量池是方法区的一部分，自然会受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError 异常。

### 直接内存

在 JDK 1.4 中新加入了 NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 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。



直接内存的大小并不受到java堆大小的限制，甚至不受到JVM进程内存大小的限制。它只受限于本机总内存（RAM及SWAP区或者分页文件）大小以及处理器寻址空间的限制（最常见的就是32位/64位CPU的最大寻址空间限制不同）。

直接内存出现OutOfMemoryError的原因是对该区域进行内存分配时，其内存与其他内存加起来超过最大物理内存限制（包括物理的和操作系统级的限制），从而导致OutOfMemoryError。另外，若我们通过参数“-XX:MaxDirectMemorySize”指定了直接内存的最大值，其超过指定的最大值时，也会抛出内存溢出异常。

## 运行时数据区域逻辑

### 详细讲解虚拟机栈

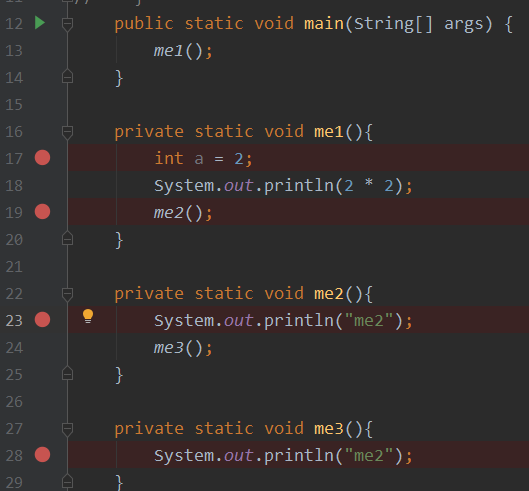
与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法返回等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

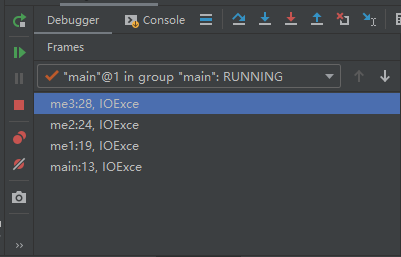
#### 栈帧

栈帧（Frame）是用来存储方法的执行状态和临时数据的数据结构。

特点：

1. 栈帧是线程私有的数据，不可能在一个栈帧之中引用另外一条线程的栈帧。
2. 栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。
3. 正在执行的栈帧被称之为当前栈帧。当方法返回的时候，当前栈帧会传回此方法的执行结果给前一个栈帧，当前栈帧会被遗弃，前一个栈帧变成当前栈帧。



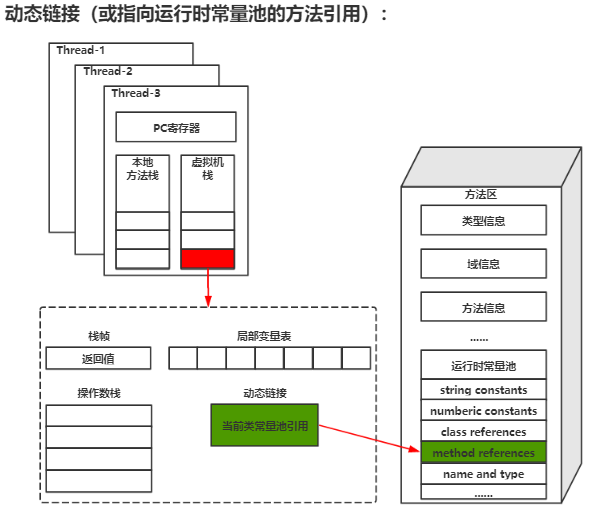


#### 动态链接与静态链接

**概述**

Class文件的常量池中存在有大量的符号引用，字节码中的方法调用指令就以指向常量池的引用作为参数。

1. 部分符号引用在类加载阶段(解析)的时候就转化为直接引用,这种转化为静态链接。
2. 部分符号引用在运行期间转化为直接引用,这种转化为动态链接。



QA：

所以说，什么是符号引用和直接引用呢？

答：

1） 符号引用就是字符串，这个字符串包含足够的信息，以供实际使用时可以找到相应的位置。比如说某个方法的符号引用，如：“java/io/PrintStream.println:(Ljava/lang/String;)V”。里面有类的信息，方法名，方法参数等信息。

为什么要使用符号引用呢？

当第一次运行时，要根据字符串的内容，到该类的方法表中搜索这个方法。运行一次之后，符号引用会被替换为直接引用，下次就不用搜索了。

2） 直接引用在内存中能定位到地址，可以直接定位到要执行的方法。

直接引用可以是

（1）直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）

（2）相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）

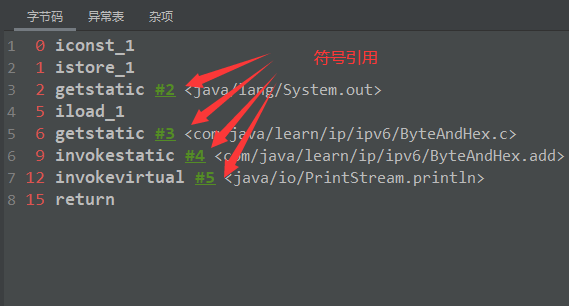
（3）一个能间接定位到目标的句柄

**举个例子：**

private static int *c* = 0;

public static void main(String[] args) {  
 int a = 1;  
 System.*out*.println(*add*(a, *c*));  
}  
  
private static int add(int a, int b){  
 return a + b;  
}

我们通过JClassLib分析它的class文件，下面是main方法的字节码：

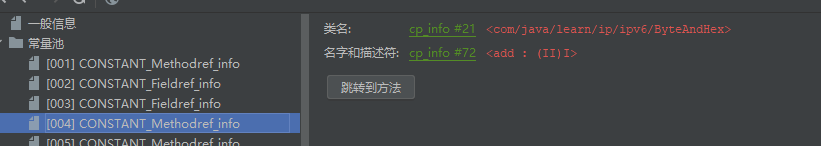


我们就可以通过这些符号索引去常量池里面去获取他们的地址，进而获取他们的直接引用或值。

比如我们的add方法。



获取到这个方法的路径“<com/java/learn/ip/ipv6/ByteAndHex>”



再根据一定的规则，转为这个方法的直接引用（地址），通过它就可以访问add方法了。

#### 局部变量表

局部变量表是一组局部变量值存储空间，用于存放方法参数和方法内部定义的局部变量。

**局部变量槽（slot）**

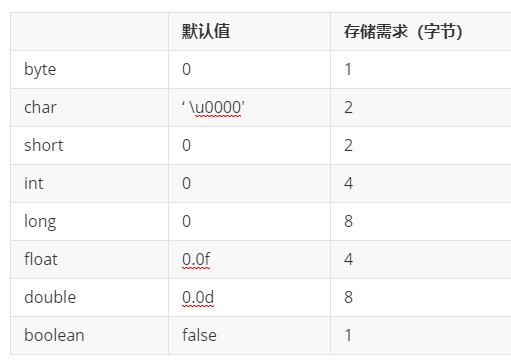
局部变量槽是局部变量表中的基本单位，一个变量槽的大小完全由虚拟机自己定义，在最常使用的HotSpot中，一个局部变量槽是占用32个比特。

基本知识：

1. 一个slot可以保存一个类型为 boolean、byte、char、short、float、reference和 returnAddress的数据，两个slot可以保存一个类型为 long 和 double 的数据。

QA：

1. reference和 returnAddress 数据代表的是什么？

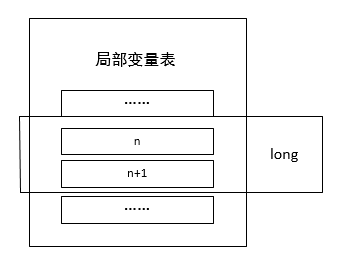


1. 从上面这个表中，明显可以看到boolean、byte、char、short并没有占用4个字节，那么为什么在局部变量表中的时候还是占1个slot呢？
2. 局部变量表在编译期会初始化一个固定长度的数组，用于存储局部变量。从局部变量数组的index 0开始存放参数值，到数组长度-1的索引结束。值得注意的是，如果当前方法是由构造方法或实例方法创建的（非静态方法），那么该对象引用this将会存放在index为0的Slot处。

QA：这也解决了我的一些疑问

1. 为什么static中不能调用this方法？
2. JVM会为局部变量表中的每一个Slot都会分配一个访问索引，通过这个索引可以访问到局部变量表中指定的局部变量值。方法参数和方法体内部的局部变量会按照顺序被复制到局部变量表中的每一个Slot上。
3. 如果需要访问局部变量表的一个64bit的局部变量值时，只需要使用前一个访问索引即可。

注：long 和 double 类型的数据占用两个连续的局部变量，这两种类型的数据值采用两个局部变量之中较小的索引值来定位。例如我们讲一个 double 类型的值存储在索引值为 n 的局部变量中，实际上的意思是索引值为 n 和 n+1 的两个局部变量都用来存储这个值。索引值为 n+1 的局部变量是无法直接读取的。



局部变量表的特点：

1. 局部变量表中的变量只在当前方法调用中有效。
2. 局部变量表的长度由编译期决定，多大的局部变量表，多深的操作数栈都已经完全确定了，在方法运行期间时不会改变局部变量表的大小的。
3. 局部变量表是建立在线程的栈上，是线程私有的数据，因此不存在数据安全问题。

#### 操作数栈

主要用于计算。和局部变量表一样，操作数栈也是一个数组。但是和前者不同的是，操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标准的入栈和出栈操作来完成一次数据访问。

特点：

1. 操作数栈的最大深度也是编译的时候就确定了。
2. 栈中的任何一个元素都是可以任意的Java数据类型，32位的类型占用一个栈单位深度，64位的类型占用两个栈单位深度。
3. 数据运算的地方，大多数指令都在操作数栈先将数据弹出栈，再执行运算，然后将结果压栈。

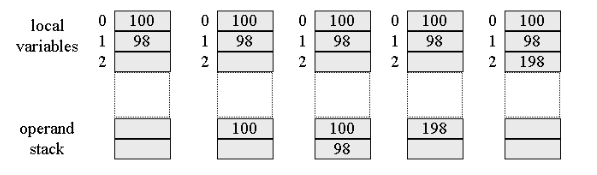
在操作栈中，基本上都是通过字节码指令对栈进行数据的入栈和出栈操作。举例：

iload\_0：将局部变量表中的第0个索引的局部变量压入栈。

iload\_1：将局部变量表中的第1个索引的局部变量压入栈。

iadd：一次弹出栈顶两个元素，然后相加，再压入栈中。

istore\_2：弹出栈顶元素，然后存入到局部变量表第二个索引的局部变量中。



字节码指令地址：

[https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.getstatic](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html%23jvms-6.5.getstatic)

#### 方法返回地址

1. 存放调用该方法的PC寄存器的值。（PC寄存器是用来存储指向下一条指令的地址）
2. 当一个方法开始执行后，只有两种方式可以退出这个方法:
   1. 执行引擎遇到任意一个方法返回的字节码指令（return），会有返回值传递给上层的方法调用者，简称正常完成出口。
   2. 方法执行过程中遇到异常： 无论是java虚拟机内部产生的异常还是代码中throw出的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器（catch），就会导致方法退出，这种退出的方式称为**异常完成出口**
      1. 一个方法若使用该方式退出，是不会给上层调用者任何返回值的。
      2. 无论使用那种方式退出方法，都要返回到方法被调用的位置，程序才能继续执行。

总结：本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值入调用者栈帧的操作数栈、设置PC寄存器值等，让调用者方法继续执行下去。

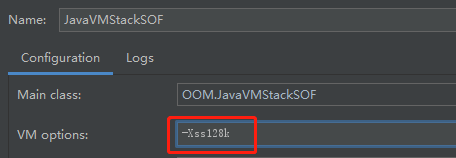
#### 虚拟机栈中的异常

在Java 虚拟机规范中，对这个区域规定了两种异常状况：

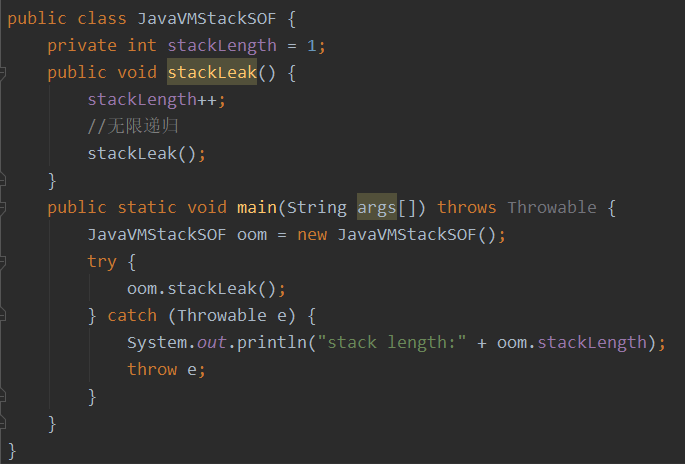
1. 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError 异常；

举个例子：

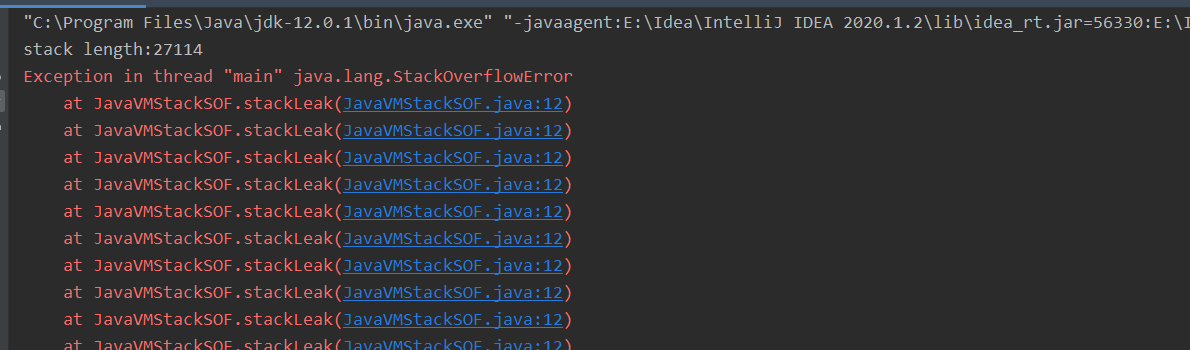
先设置一下栈大小为128K。



执行下面的递归方法，无线递归：



就会出现StackOverflowError 异常：



1. 如果虚拟机栈可以动态扩展（当前大部分的Java 虚拟机都可动态扩展，只不过Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出OutOfMemoryError 异常。

举个例子：

（1）把内存设置为2M，-Xss2M方法和上面是相同的。

（2）跑代码，不断的建立线程。



（3）运行结果

抛出OutOfMemoryError异常。

**总结**

1. 在单线程下，设置了栈内存为128k。无限递归下，导致了栈深度太大，抛出了StackOverflowError异常。
2. 在多线程下，设置了栈内存为2M，由于虚拟机栈是线程独占的，就是说每个线程都有自己的虚拟机栈，那么假设有10000个线程，那么内存占用应该是2M\*10000，约为20G，而可用的总内存大小是有限的，那么总有线程的虚拟机栈分配不到足够的内存，就会报出OOM异常。

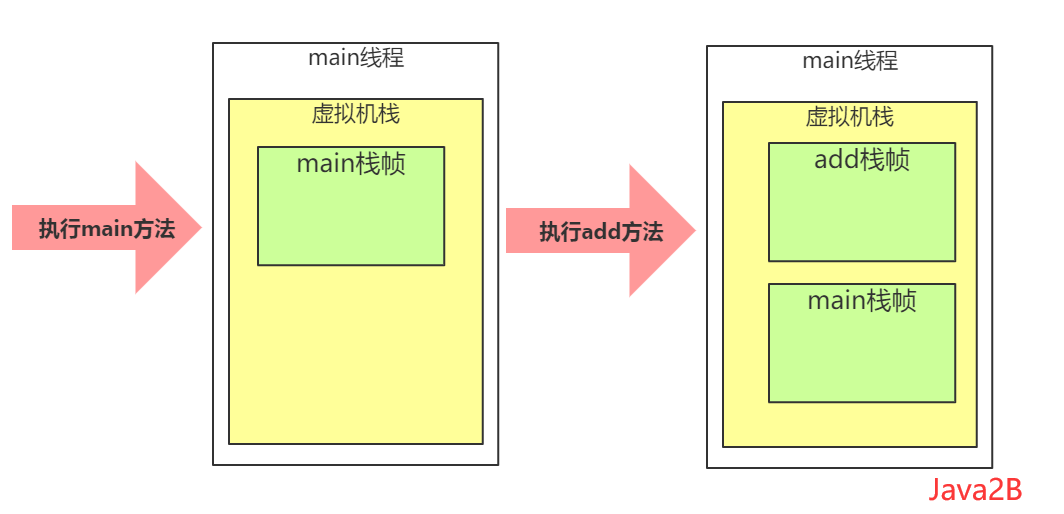
#### 虚拟机栈运行过程

简单的例子：

public class IOExce {  
 public static void main(String[] args) {  
 Math math = new Math();  
 System.*out*.println(math.add());  
 }  
}

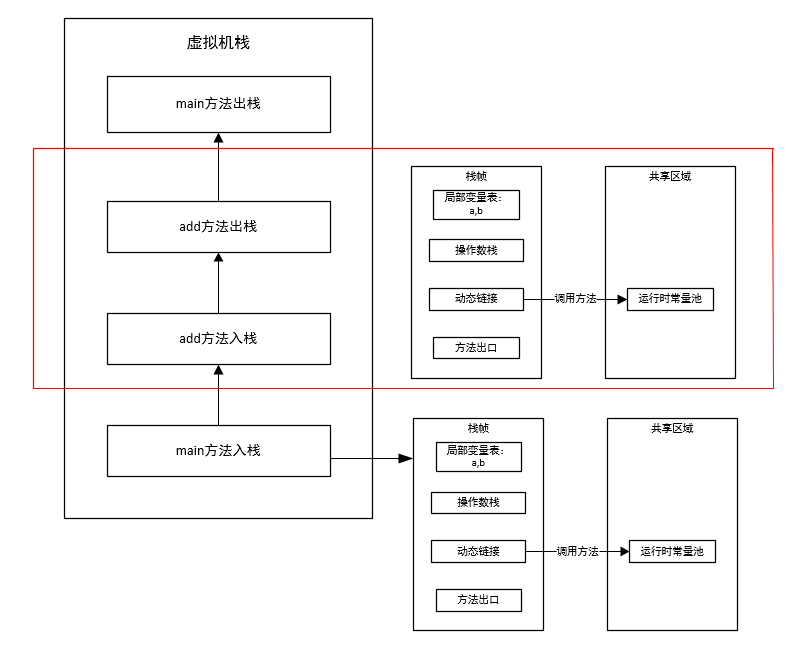
public class Math {  
 int add() {  
 int a = 10;  
 int b = 20;  
 int sum = a + b;  
 return sum \* 3;  
 }  
}

执行过程：

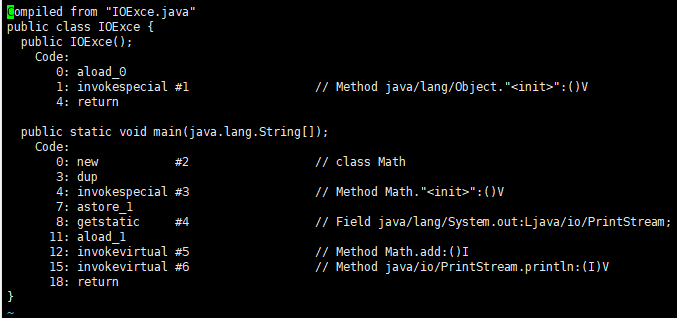


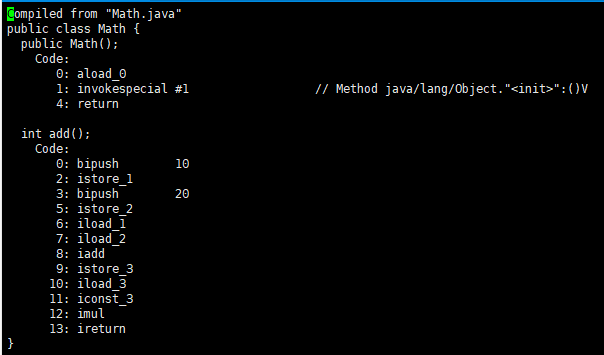
当执行main方法时，栈中会分配main()方法的栈帧，并存储局部变量a和b，紧接着执行相加后存入局部变量表。然后执行add()方法，那么栈又会分配add()的栈帧区域，add栈帧就变成了当前栈帧。

具体的信息如下：



字节码文件：



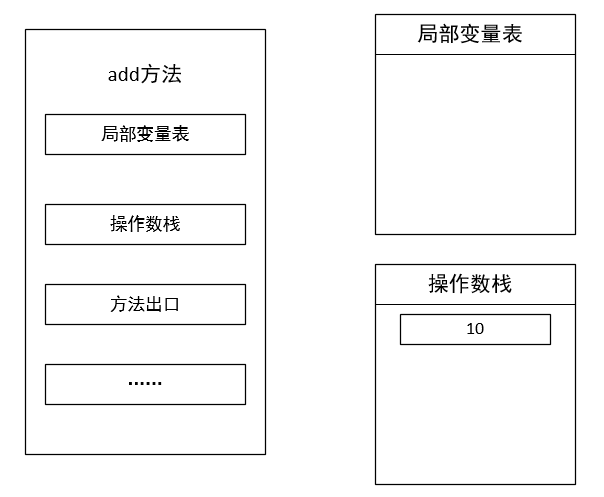


分析add方法执行流程:

1. 压栈

将变量a压入操作数栈中

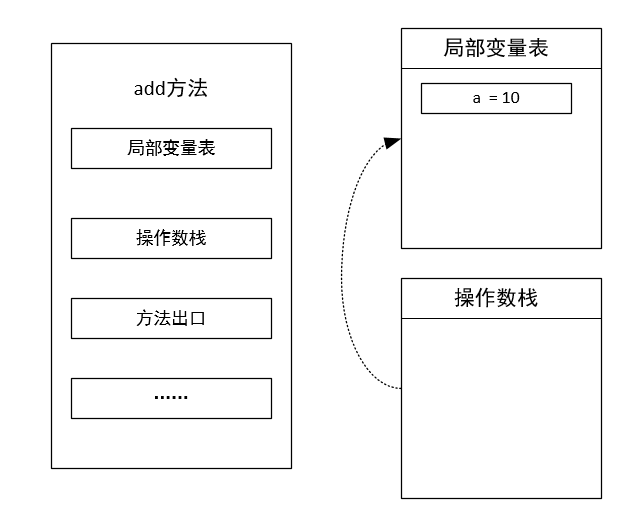
|  |
| --- |
| bipush 10 |



2）存储

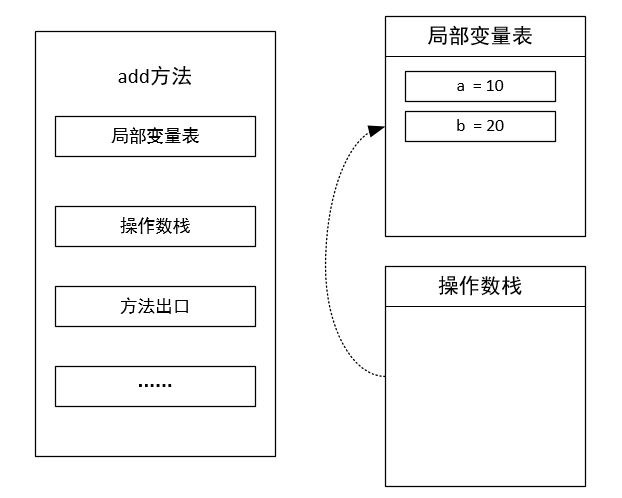
将int类型的a存入局部变量表中

|  |
| --- |
| istore\_0 |



变量b也重复1,2过程。

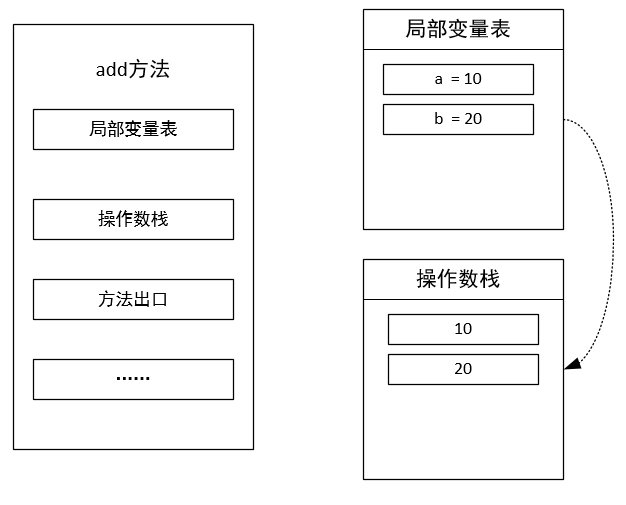
|  |
| --- |
| bipush 20  istore\_1 |



3）装载

从局部变量表中取出变量a和b，压入操作数栈中

|  |
| --- |
| iload\_0  iload\_1 |

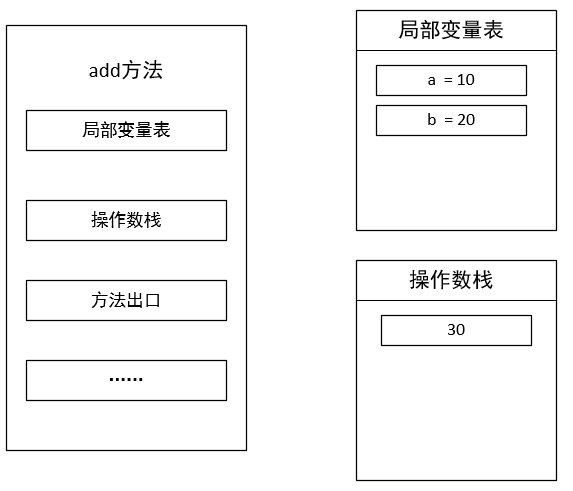


1. 执行运算

执行加法运算

|  |
| --- |
| iadd |

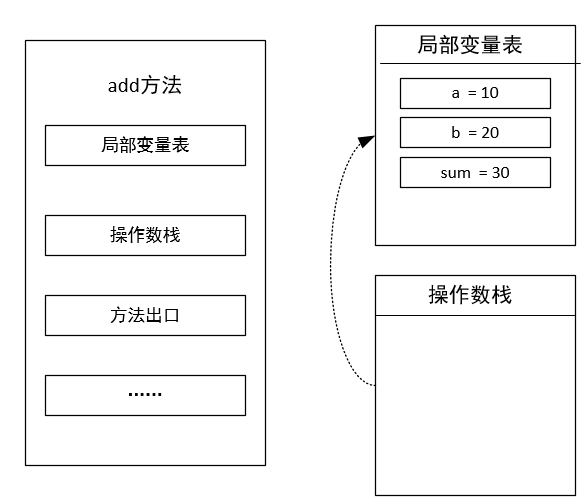
iadd指令一执行，会将操作数栈中的10和20依次从栈底弹出并相加，然后把运算结果30在压入操作数栈底。



1. 存储

将运算结果存入局部变量表中

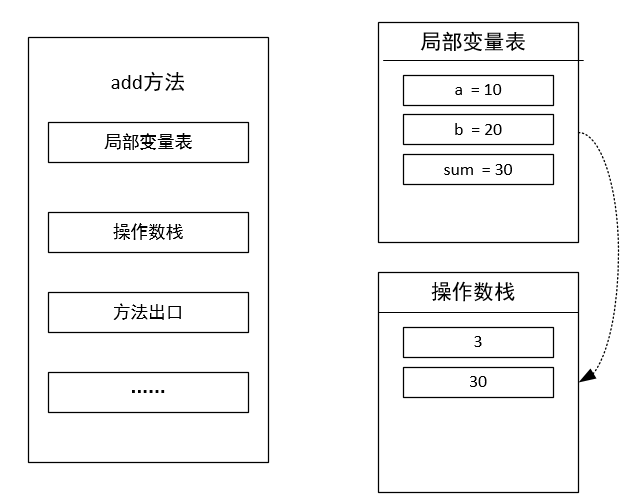
|  |
| --- |
| istore\_2 |



iload\_2装载sum，重复第二步

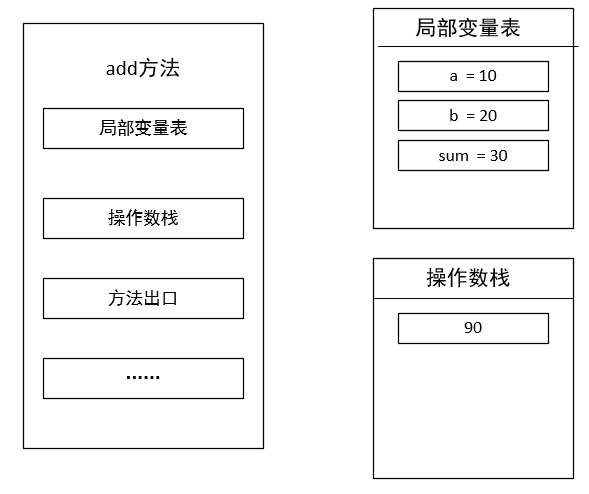
1. 压栈

iconst\_3执行压栈，将3压入栈里面



7) 执行乘法

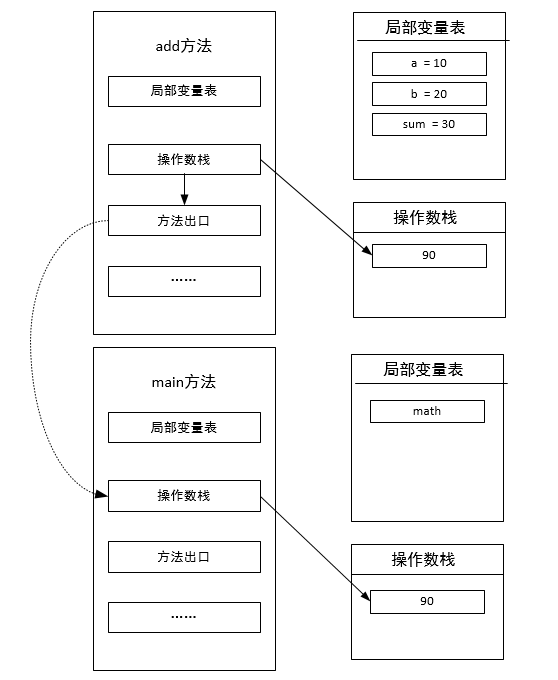
imul 将操作数栈中的3和30依次从栈底弹出并相乘，将结果压入栈中



8）方法返回

ireturn 从方法中返回结果。

add方法是被main方法中调用的，所以通过方法出口返回到mian方法中。add()栈帧中的方法出口就存储了当前要回到的位置，那么当add()方法执行完之后，会根据方法出口中存储的相关信息回到main()方法的相应位置。



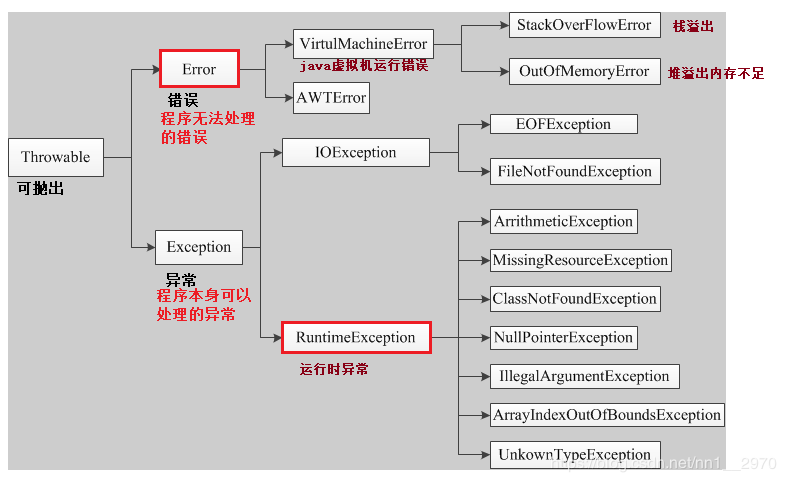
### 异常捕获原理及程序计数器

#### 异常分类

java异常类均继承于java.lang.Throwable。其中可以分为错误和异常：

错误：Error类以及他的子类的实例，代表了JVM本身的错误。错误不能被程序员通过代码处理，Error很少出现。

异常：Exception以及他的子类，代表程序运行时发送的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。



根据Javac对异常的处理要求，将异常类分为2类：

**非检查异常**：Error 和 RuntimeException 以及他们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常，对于这些异常，我们应该修正代码，而不是去通过异常处理器处理 。常见的RuntimeException ：

ArrayIndexOutOfBoundsException（数组下标越界）

NullPointerException（空指针异常）

ArithmeticException（算术异常）

MissingResourceException（丢失资源）

ClassNotFoundException（找不到类）

**检查异常**：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try…catch…finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。如SQLException , IOException等。

下面举个例子：

public class ExceptionTypeTest {  
 public void doSomething() throws ArithmeticException {  
 System.*out*.println();  
 }  
  
 public static void main() {  
 ExceptionTypeTest ett = new ExceptionTypeTest();  
 ett.doSomething();  
 }  
}

问题1:上面的程序能否编译通过？并说明理由。

问题2:上面的程序将ArithmeticException改为IOException能否编译通过？并说明理由。

问题3:那又牵扯出一个问题，非检查异常和检查异常的根本区别是在哪呢？这么划分的作用又是什么呢？

答：

用最为官方的说法，Java最核心设计者之一JOSHUA BLOCH的《Effective Java》异常使用章节，他的主张是：对可恢复的情况使用检查型异常，对编程错误使用运行时异常。我的理解，如果出现这个异常，我们就通过catch捕获进行处理，让他经过处理后能够继续这个线程执行。

#### 导致异常的原因

Java 虚拟机中

异常的出现总是由下面三种原因之一导致的：

1. 虚拟机同步检测到程序发生了非正常的执行情况，这时异常将会紧接着在发生非正常执行情况的字节码指令之后抛出。例如：
   * + 字节码指令所蕴含的操作违反了 Java 语言的语义，如访问一个超出数组边界范围的元素。
     + 类在加载或者链接时出现错误。
     + 使用某些资源的时候产生资源限制，例如使用了太多的内存。
2. athrow 字节码指令被执行。
3. 由于以下原因，导致了异步异常的出现：

* Java 虚拟机实现的内部程序错误。
* 调用了 Thread 或者 ThreadGroup 的 stop 方法。

当某条线程调用了 stop 方法时，将会影响到其他的线程，或者在线程组中的所有线程。 这时候其他线程中出现的异常就是异步异常，因为这些异常可能出现在程序执行过程的任何位置。虚拟机的内部异常也被认为是一种异步异常。

#### 异常捕获流程

接上回我们讲的java虚拟机栈，当一个方法正常执行完毕，Java虚拟机会从调用栈中弹出该方法的栈结构，然后继续处理前一个方法。如果在执行方法的过程中抛出异常，则Java虚拟机必须找到能捕获该异常的catch代码块。它首先查看当前方法是否存在这样的catch代码块，如果存在，那么就执行该catch代码块；否则，Java虚拟机会从调用栈中弹出该方法的栈结构，继续到前一个方法中查找合适的catch代码块。在回溯过程中，如果Java虚拟机在某个方法中找到了处理该异常的代码块，则该方法的栈结构将成为栈顶元素，程序流程将转到该方法的异常处理代码部分继续执行。当Java虚拟机追溯到调用栈的底部的方法时，如果仍然没有找到处理该异常的代码块，按以下步骤处理。

（1）调用异常对象的printStackTrace()方法，打印来自方法调用栈的异常信息。

（2）如果该线程不是主线程，那么终止这个线程，其他线程继续正常运行。如果该线程是主线程（即方法调用栈的底部为main()方法），那么整个应用程序被终止。（比如在项目启动的时候，发生了一些错误，且中途也没被捕获）。

#### Exception Table 异常表

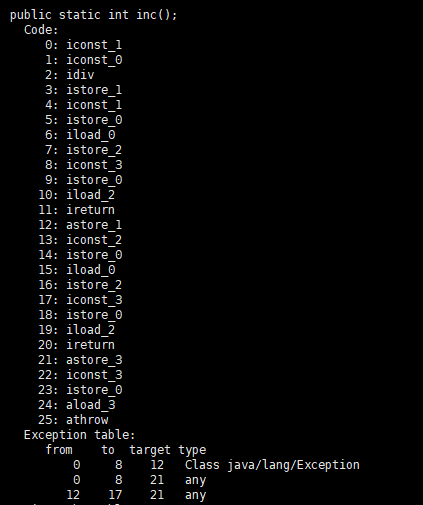
属性表（attribute\_info）可以存在于 Class 文件、字段表、方法表中，用于描述某些场景的专有信息。属性表中有个 Code 属性，该属性在方法表中使用，Java 程序方法体中的代码被编译成的字节码指令存储在 Code 属性中。而异常表（exception\_table）则是存储在 Code 属性表中的一个结构，这个结构是可选的。

在编译生成的字节码中,每个方法都有一个异常表，异常表中的每一个条目代表着一个异常处理器，并且由from指针、to指针、target指针以及所捕获的异常类型构成,这些指针的值都是字节码索引，用以定位字节码。

看下面一段代码

public static int inc() {  
 int x;  
 try {  
 int i = 1 / 0;  
 x = 1;  
 return x;  
 } catch (Exception e) {  
 x = 2;  
 return x;  
 } finally {  
 x = 3;  
 }  
}

使用javap



其中:from和to指针标示了该异常处理器所监控的范围，例如try代码块覆盖的范围，target指针则指异常处理器的开始位置，例如catch代码块的起始位置。

当程序触发异常的时候，java虚拟机会从上到下遍历异常表中的所有条目。当触发异常的字节码索引值在某个异常表条目的监控范围内，java虚拟机会判断所抛出的异常和该条目想要捕获的异常是否匹配。如果匹配，java虚拟机会将控制流转移至该条目target指针指向的字节码。

如果遍历完所有的异常条目，Java虚拟机仍未匹配到异常处理器，那么它会弹出当前方法对应的Java栈帧,并且在调用者中重复上述操作，在最坏的情况下，Java虚拟机需要遍历当前线程Java栈的所有异常表。

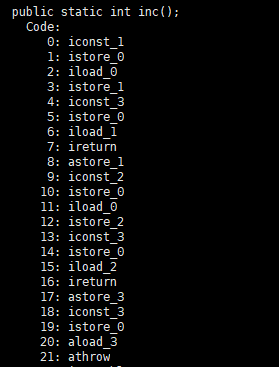
finally代码块的编译比较复杂。当前版本java编译器的做法是复制finally代码块中的内容，分别放在try-catch代码块所有正常执行路径以及异常执行路径的出口中。

#### 异常处理实例

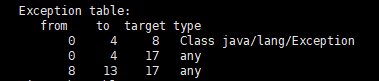
我们来用一段代码来解释，如下所示：

public int inc() {  
 int x;  
 try {  
 x = 1;  
 return x;  
 } catch(Exception e) {  
 x = 2;  
 return x;  
 } finally {  
 x = 3;  
 }  
}

给出上面Java源码编译后的字节码及异常表信息：

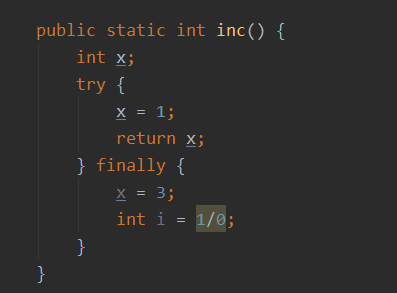


Exception table：

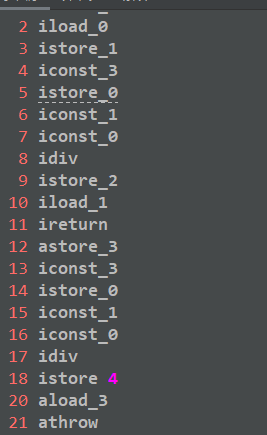


首先，0-3行，就是把整数1赋值给 x，并且将此时 x 的值复制一个副本到局部变量表的 Slot 中暂存，这个 Slot 里面的值在 ireturn 指令执行前会被重新读到栈顶，作为返回值。这时如果没有异常，则执行4-5行，把 x 赋值为3，然后返回前面保存的1，方法结束。如果出现异常，读取异常表发现应该执行第8行，pc 寄存器指针转向8行，8-16行就是把2赋值给 x，然后把 x 暂存起来，再将 x 赋值为3，然后将暂存的2读到操作栈顶返回。第17行开始是把 x 赋值为3并且将栈顶的异常抛出，方法结束。

思考：如果是在finally中抛出了错误，那么字节码指令又是如何执行的？



字节码指令



**小结一下**

从字节码层面分析了虚拟机在处理异常流程的过程，我们可以看出以下几点内容：

异常表实际上是Java代码的一部分，编译器使用异常表而不是简单的跳转命令来实现Java异常及finally处理机制。

当异常处理存在finally语句块时，编译器会自动在每一段可能的分支路径之后都将finally语句块的内容冗余生成一遍来实现finally语义。

在我们Java代码中，finally语句块是在最后的，但编译器在生成字节码时候，其实将finally语句块的执行指令移到了ireturn指令之前，指令重排序了。所以，从字节码层面，我们解释了，为什么finally语句总会执行。

#### CPU中的程序计数器（PC）

当执行一条指令时，首先需要根据PC中存放的指令地址，将指令由内存取到指令寄存器中，此过程称为“取指令”。与此同时，PC中的地址或自动加1或由转移指针给出下一条指令的地址。此后经过分析指令，执行指令。完成第一条指令的执行，而后根据PC取出第二条指令的地址，如此循环，执行每一条指令。

#### JVM中程序计数器

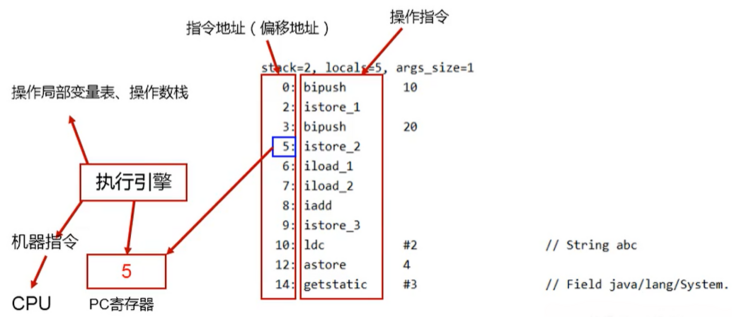
程序计数器（Program Counter Register）又叫PC寄存器，是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器（实质是存储的是指令地址）。

在CPU中PC是一个物理设备，而java中PC则是一个一块比较小的内存空间，它是当前线程字节码执行的行号指示器。在java的概念模型中，字节码解释器就是通过改变这个计数器中的值来选取下一条执行的字节码指令的，分支操作（if语句，switch语句）、循环操作（for，while，do...while）、跳转（break，continue，return）、异常处理和线程恢复等基础模型都需要依赖这个计数器来完成。

#### 字节码的执行原理

编译后的字节码在没有经过JIT(实时编译器)编译前，是通过字节码解释器进行解释执行。其执行原理为：字节码解释器读取内存中的字节码，按照顺序读取字节码指令，读取一个指令就将其翻译成固定的操作。

**举例：**



图中使用红框框起来的就是字节码指令的偏移地址，CPU就根据偏移地址，到相应的内存取出指令然后执行，在计数器中存放执行地址 （红框中的）0 2 3…等等。

执行顺序

->将偏移量赋值给PC寄存器

->执行引擎（解释器）根据PC寄存器中的偏移地址获取字节码指令

->对字节码指令进行解析，解析的过程中会操作操作数栈，局部变量表等

->转换为机器指令

->交给CPU进行处理，处理过程中将新的偏移量赋值给PC寄存器

思考：线程按着指令的顺序执行下去是完全能够保证程序正常执行的，那么为什么还需要程序计数器呢？

我们知道多线程的实现是多个线程轮流占用CPU而实现的，而在线程切换的时候就需要保存当前线程的执行状态，这样在这个线程重新占用CPU的时候才能恢复到之前的状态。

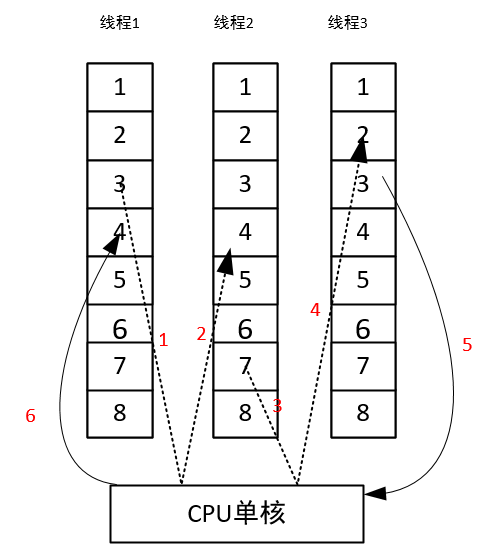
#### 程序计数器的作用

从字节码的执行原理来看，单线程的情况下程序计数器是可有可无的。因为即使没有程序计数器的情况下，程序会按照指令顺序执行下去，即使遇到了分支跳转这样的流程也会按照跳转到指定的指令处继续顺序执行下去，是完全能够保证执行顺序的。

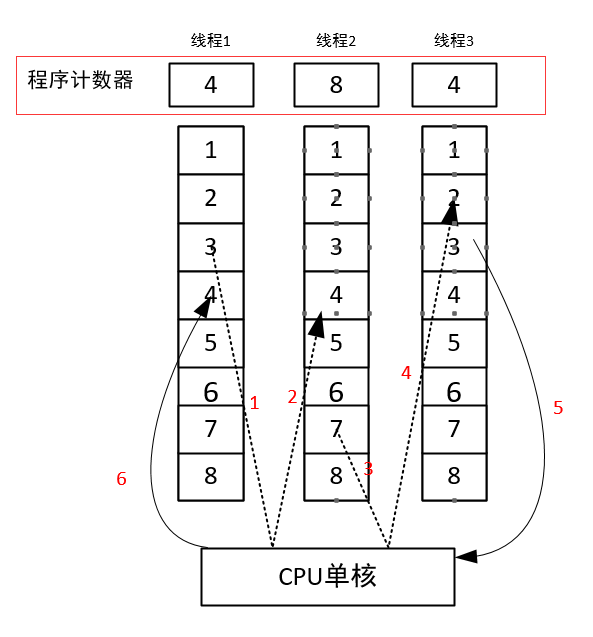
但是现实中程序往往是多线程协作完成任务的。JVM的多线程是通过CPU时间片轮转来实现的，某个线程在执行的过程中可能会因为时间片耗尽而挂起。当它再次获取时间片时，需要从挂起的地方继续执行。在JVM中，通过程序计数器来记录程序的字节码执行位置。

PC寄存器为什么被设置为线程私有的？

我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU会不停地做任务切换，这样必然导致经常中断或恢复，如何保证分毫无差呢？为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个PC寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。



CPU执行线程1的1，2，3的指令，执行3之后，根据时间片的轮转机制，将这个CPU核交给线程2执行4-7步骤的指令，再交给线程3，从第二步的指令开始执行。如果是将程序计数器是属于线程共享的，那么在执行第6步的过程中，就无法恢复线程1是执行到了第几步。



这样就能让我们的程序看着像并行的运行，但其实是在并发运行的。

#### 执行native方法

程序计数器中存储的值，在java方法中是字节码偏移量，而在native方法中则是undefined（native是非java代码编写的，比如C，C++, 它们无法在java编译时生成字节码，即JVM获取不到native实现，只能通过系统指令去调用native方法）。

执行native本地方法时，程序计数器的值为空（Undefined）。因为native方法是java通过JNI直接调用本地C/C++库，可以近似的认为native方法相当于C/C++暴露给java的一个接口，java通过调用这个接口从而调用到C/C++方法。由于该方法是通过C/C++而不是java进行实现。那么自然无法产生相应的字节码，并且C/C++执行时的内存分配是由自己语言决定的，而不是由JVM决定的。

问题：上面这段话，也让我产生了疑问，当native方法执行完成之后，我怎么才能继续回到我们的java方法中继续执行呢？

Java线程以某种形式映射到OS线程上。映射模型可以是1:1（原生线程模型）、n:1（绿色线程 / 用户态线程模型）、m:n（混合模型）。以HotSpot VM的实现为例，它目前在大多数平台上都使用1:1模型，也就是每个Java线程都直接映射到一个OS线程上执行。此时，native方法就由原生平台直接执行，并不需要理会抽象的JVM层面上的“pc寄存器”概念——原生的CPU上真正的PC寄存器是怎样就是怎样。就像一个用C或C++写的多线程程序，它在线程切换的时候是怎样的，Java的native方法也就是怎样的。

当native方法部分执行完成之后，OS线程就会将地址指向我们的java方法编译后的指令位置，那么就又回到了我们的java线程中，程序计数器的值由Undefined改为我们正常的字节码行号。

关于线程模型的讲解

<https://blog.csdn.net/qq_41169749/article/details/102924043>

#### 程序计数器的特点

(1) 程序计数器具有线程隔离性，属于线程私有

(2) 程序计数器占用的内存空间非常小，可以忽略不计，也是运行速度最快的存储区域。

(3) 程序计数器是java虚拟机规范中唯一一个没有规定任何OutofMemeryError的区域（程序计数器中的内容会被不断替换为下一个地址），无GC。

(4) 程序执行的时候，程序计数器是有值的，其记录的是程序正在执行的字节码的地址

(5) 执行native本地方法时，程序计数器的值为undefind。

(6) 生命周期与线程相同，线程启动而产生，线程结束而消亡

### 本地方法栈

#### 什么本地方法？

简单来讲，一个Native Method就是一个java调用非java代码的接口。一个Native Method是这样一个java的方法：该方法的实现由非java语言实现，比如C。这个特征并非java所特有，很多其它的编程语言都有这一机制，比如在C＋＋中，你可以用extern "C"告知C＋＋编译器去调用一个C的函数。

#### 本地方法和Java方法的区别

* JAVA方法是由JAVA编写的，编译成字节码，存储在class文件中
* 本地方法是由其它语言编写的，编译成和处理器相关的机器代码
* 本地方法保存在动态链接库中，即.dll(windows系统)文件中，格式是各个平台专有的
* JAVA方法是与平台无关的，但是本地方法不是。

#### 如何使用本地方法栈

在定义一个native method时，并不提供实现体（有些像定义一个java interface），因为其实现体是由非java语言在外面实现的。，下面给了一个示例：

public class IHaveNatives

{

native public void Native1( int x ) ;

native static public long Native2() ;

native synchronized private float Native3( Object o ) ;

native void Native4( int[] ary ) throws Exception ;

}

* 标识符native可以与所有其它的java标识符连用，但是abstract除外。native与其它java标识符连用时，其意义同非Native Method并无差别，比如native static表明这个方法可以在不产生类的实例时直接调用，这非常方便，比如当你想用一个native method去调用一个C的类库时。上面的第三个方法用到了native synchronized，JVM在进入这个方法的实现体之前会执行同步锁机制（就像java的多线程。）
* 一个native method方法可以返回任何java类型，包括非基本类型，而且同样可以进行异常控制。这些方法的实现体可以制一个异常并且将其抛出，这一点与java的方法非常相似。当一个native method接收到一些非基本类型时如Object或一个整型数组时，这个方法可以访问这非些基本型的内部，但是这将使这个native方法依赖于你所访问的java类的实现。有一点要牢牢记住：我们可以在一个native method的本地实现中访问所有的java特性，但是这要依赖于你所访问的java特性的实现，而且这样做远远不如在java语言中使用那些特性方便和容易。
* native method的存在并不会对其他类调用这些本地方法产生任何影响，实际上调用这些方法的其他类甚至不知道它所调用的是一个本地方法。JVM将控制调用本地方法的所有细节。需要注意当我们将一个本地方法声明为final的情况。用java实现的方法体在被编译时可能会因为内联而产生效率上的提升。但是一个native final方法是否也能获得这样的好处却是值得怀疑的，但是这只是一个代码优化方面的问题，对功能实现没有影响。
* 如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用java语言重写这个方法，同样的如果一个本地方法被fianl标识，它被继承后不能被重写。
* 本地方法非常有用，因为它有效地扩充了jvm.事实上，我们所写的java代码已经用到了本地方法，在sun的java的并发（多线程）的机制实现中，许多与操作系统的接触点都用到了本地方法，这使得java程序能够超越java运行时的界限。有了本地方法，java程序可以做任何应用层次的任务。

#### 为什么要使用本地方法

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

1）与java环境外交互

有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

2）与操作系统交互

JVM支持着java语言本身和运行时库，它是java程序赖以生存的平台，它由一个解释器（解释字节码）和一些连接到本地代码的库组成。然而不管怎样，它毕竟不是一个完整的系统，它经常依赖于一些底层（underneath在下面的）系统的支持。这些底层系统常常是强大的操作系统。通过使用本地方法，我们得以用java实现了jre的与底层系统的交互，甚至JVM的一些部分就是用C写的，还有，如果我们要使用一些java语言本身没有提供封装的操作系统的特性时，我们也需要使用本地方法。

3）Sun's Java

Sun的解释器是用C实现的，这使得它能像一些普通的C一样与外部交互。jre大部分是用java实现的，它也通过一些本地方法与外界交互。例如：类java.lang.Thread 的 setPriority()方法是用java实现的，但是它实现调用的是该类里的本地方法setPriority0()。这个本地方法是用C实现的，并被植入JVM内部，在Windows 95的平台上，这个本地方法最终将调用Win32 SetPriority() API。这是一个本地方法的具体实现由JVM直接提供，更多的情况是本地方法由外部的动态链接库（external dynamic link library）提供，然后被JVM调用。

#### JVM如何调用本地方法

我们知道，当一个类第一次被使用到时，这个类的字节码会被加载到内存，并且只会回载一次。在这个被加载的字节码的入口维持着一个该类所有方法描述符的list，这些方法描述符包含这样一些信息：方法代码存于何处，它有哪些参数，方法的描述符（public之类）等等。

如果一个方法描述符内有native，这个描述符块将有一个指向该方法的实现的指针。这些实现在一些DLL文件内，但是它们会被操作系统加载到java程序的地址空间。当一个带有本地方法的类被加载时，其相关的DLL并未被加载，因此指向方法实现的指针并不会被设置。这些DLL的加载是通过调用System.loadLibrary(String filename)或System.load(String filename)方法实现的。

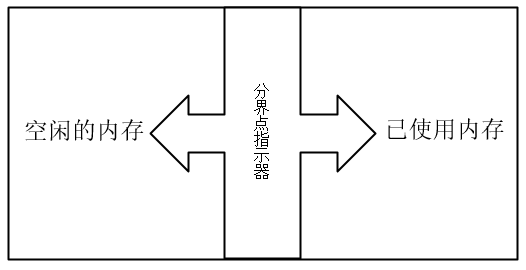
最后需要提示的是，使用本地方法是有开销的，它丧失了java的很多好处。如果别无选择，我们可以选择使用本地方法。

## 对象的一生

### 对象的创建

虚拟机遇到 new 指令时，首先将去检查该指令的参数是否能够在常量池定位到一个类的符号引用，然后判断该类是否已经被加载、解析和初始化过。如果没有，则执行相应的类加载过程，对象的所需要的内存大小在类加载完成后便完全确定了。那么就可以为开辟对象所需要的内存了。

如果堆内存是绝对规整的，也就是用过的内存放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，这种分配方式称之为 指针碰撞（Bump-the-Pointer）。



如果堆内存并不是规整的，也就是说可用内存不是连续的，已用内存和空闲空内存是相互交错的。这时候虚拟机需要维护一个列表，用于记录哪些内存是可以用的，在分配内存的时候从列表找出一个足够大内存空间划分给对象实例。这种分配方式称之为空闲列表（Free List）。

### 对象的内存布局

在HotSpot虚拟机中，对象在堆内存中的存储布局可以划分为三个部分：对象头（Header），实例数据（Instance Data），对齐填充（Padding）。



#### 对象头

对象头又分两部分，Mark Word和类型指针。

Mark Word存储对象运行时数据，如对象的hashcode，对象的分代年龄，锁状态等等。对象的分代年龄完全是为了JVM的Young GC设计的，每经历一次Young GC且存活下来的新生代对象，对象的分代年龄变会加一。直到这个字段增长到阈值，这个对象便会晋升到老年代（tenured generation）。这个字段给的是4个bit，也就是说一个java对象最多经历16次Young GC并且没被回收，那么它将进入老年代。hashcode占25bit，锁标志位占2bit，一个bit固定为0.也就是说整个Mark Word占4个字节。

每种对象实例在方法区都会与一个Class对象与之相对应。对象头的类型指针便是用来定位它的类的元数据。为何要存这个数据？原因之一便是，方法区也要GC(确切的说是Full GC)，如果没有一个类实例指向方法区的Class对象，那么这个Class对象便会被卸载回收。不知道大家有没有经历过java.lang.OutOfMemoryError: PermGen space 这样的错误。如果类一直在加载，永久代必然会被打爆。32位的JDK，类类型指针占4个字节。

#### 实例数据

是对象真正存储的有效信息，就是代码中所定义的各种类型字段的内容。不论是父类继承还是子类中定义的都要记录起来。HotSpot虚拟机默认分配策略：longs/doubles  ints  shorts/chars  bytes/booleans  oops(Ordinary Object  Pointers)，相同的字段总是分配在一起，在满足这个条件的情况下父类中定义的变量会出现在子类前。

#### 对齐填充

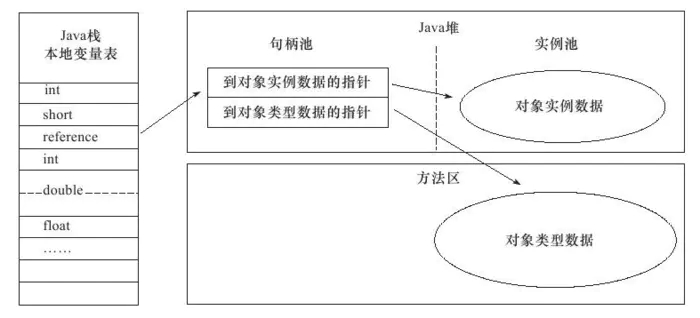
对象可以有对齐数据也可以没有。默认情况下，Java虚拟机堆中对象的起始地址需要对齐至8的倍数。如果一个对象用不到8的倍数个字节则需要对其填充，以此来补齐对象头和实例数据占用内存之后剩余的空间大小。如果对象头和实例数据已经占满了JVM所分配的内存空间，那么就不用再进行对齐填充了。

### 对象的访问定位

Java程序中需要通过栈中的reference数据操作堆上的具体对象。目前主流的访问方式有使用句柄和直接指针。

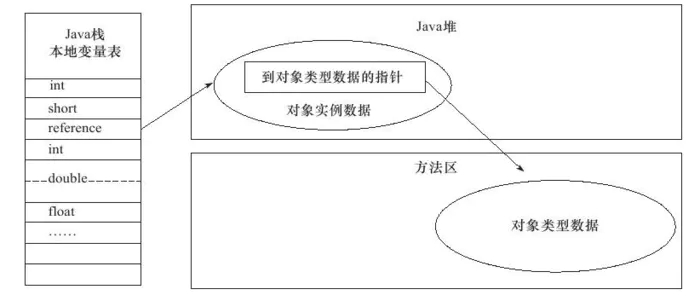
#### 使用句柄

Java堆中划分出一块内存作为句柄池，reference中存储的是对象的句柄地址，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。



#### 直接访问对象

如果使用直接指针访问，那么Java堆对象分布中就必须考虑如何放置访问类型数据的相关信息，reference存储的直接就时对象地址。



## 内存溢出异常

### Java堆溢出

Java堆用于存储对象实例，只要不断创建对象，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，那么在对象数量达到最大堆的容量限制之后就会产生内存溢出异常。

解决思路：先通过内存映像分析工具对dump出来的堆转储快照进行分析，先分清楚是内存泄漏还是内存溢出；如果是内存泄漏，进一步查看泄漏对象到GC Roots的引用链，从而确认为什么无法回收；如果是内存溢出，则应当检查虚拟机堆参数（-Xms与-Xmx）或检查是否存在对象生命周期过长、持有状态时间过长的情况。

### Java虚拟机栈和本地方法栈溢出

HotSpot不区分虚拟机栈和本地方法栈。栈的容量只由-Xss参数设置。

关于虚拟机栈和本地方法栈，在《Java虚拟机规范》中描述了两种异常：

1. 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。
2. 如果虚拟机的栈内存允许动态扩展，当扩展栈容量无法申请到足够的内存时，将判处OutOfMemoryError异常

### 方法区和运行时常量池溢出

方法区用于存放Class的相关信息，对这个区域的测试，基本思路是运行时产生大量的类去填满方法区，直到溢出。使用CGLib实现。

方法区溢出也是一种常见的内存溢出异常，在经常生成大量Class的应用中，需要特别注意类的回收情况，这类场景除了使用了CGLib字节码增强和动态语言外，常见的还有JSP文件的应用(JSP第一次运行时要编译为Java类)、基于OSGI的应用等。

### 本地直接内存溢出

Direct Memory 容量可通过 -XX:MaxDirectMemorySize 指定，如不指定，则默认与Java堆最大值一样。

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，如果发现 OOM 之后 Dump 文件很小，而程序直接或间接使用了NIO，那就可以考虑检查一下是不是这方面的原因。

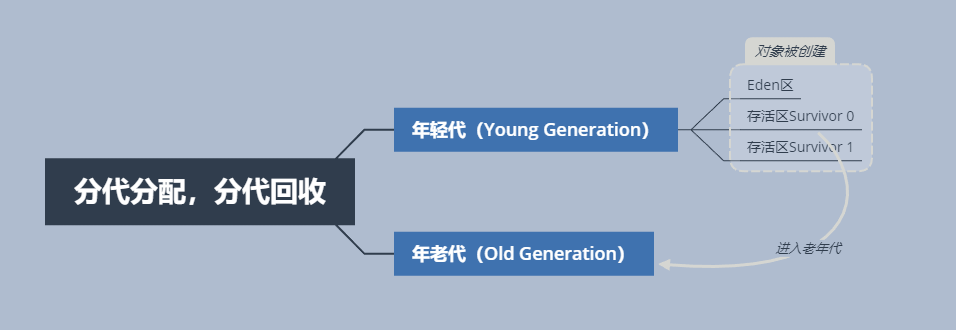
## GC策略及性能调优

年轻代（Young Generation）：对象被创建时，内存的分配首先发生在年轻代（大对象可以直接被创建在年老代），大部分的对象在创建后很快就不再使用，因此很快变得不可达，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消亡的），这个GC机制被称为Minor GC或叫Young GC。注意，Minor GC并不代表年轻代内存不足，它事实上只表示在Eden区上的GC。

年老代（Old Generation）：对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时，将执行Major GC，也叫 Full GC。

### 对象被GC的过程：

1. 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
2. 最初一次，当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
3. 下次Eden区满了，再执行一次Minor GC，将消亡的对象清理掉，将存活的对象复制到Survivor1中，然后清空Eden区；
4. 将Survivor0中消亡的对象清理掉，将其中可以晋级的对象晋级到Old区，将存活的对象也复制到Survivor1区，然后清空Survivor0区；
5. 当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代，但这只是个最大值，并不代表一定是这个值）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。



### Minor GC或叫Young GC

YoungGC的触发时常在发生，当新生代的Eden区满了之后就会触发YoungGC。

新生代的垃圾收集动作，新生代中的对象朝生夕死，所以 Minor GC 非常频繁，回收速度也比较快。

### Major GC或叫 Full GC

1. 调用 System.gc() 方法时，会建议JVM进行Full GC，此方法不建议使用。
2. 新生代使用的是复制算法，为了内存利用率，只使用其中一个 Survivor 空间来做轮换备份，因此如果大量对象在 Minor GC 后仍然存活，导致 Survivor 空间不够用，就会通过分配担保机制，将多出来的对象提前转到老年代，此时如果老年代的可用内存小于该对象的大小，就会触发 Full GC。
3. 当老年代中最大可用的连续空间小于历代晋升到老年代的对象的平均大小时，会触发Full GC 来让老年代腾出更多的空间。
4. 未指定老年代和新生代大小，堆伸缩时会产生fullgc，所以一定要配置-Xmx、-Xms

### 如何优化GC？

1. 尽量不要创建过大的对象或数组。
2. 通过虚拟机的 -Xmn 参数适当调大新生代的大小，让对象尽量在新生代中被回收掉。
3. 通过 -XX:MaxTenuringThreshold 参数调大对象进入老年代的年龄，让对象尽量在新生代中被回收掉。

### 堆大小设置

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制。

典型设置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置JVM最大可用内存为3550M。

-Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g：设置年轻代大小为2G。整个JVM内存大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m:设置持久代大小为16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。