# JVM基础知识

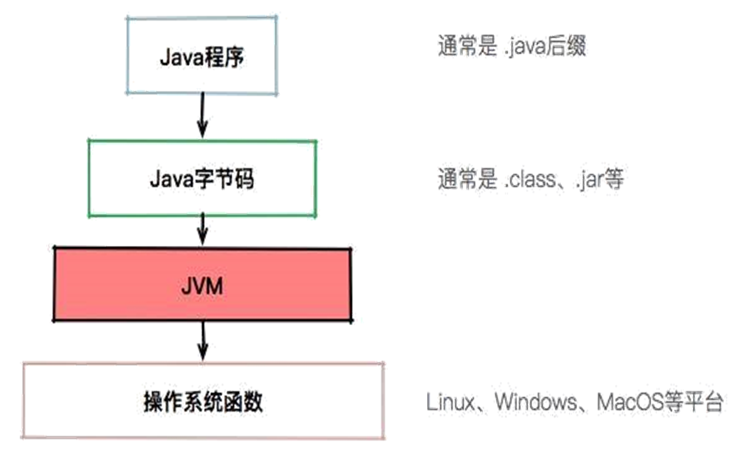
## JVM与操作系统的关系

### Java Virtual Machine

JVM 全称 Java Virtual Machine，也就是我们耳熟能详的 Java 虚拟机。它能识别 .class后缀的文件，并且能够解析它的指令，最终调用操作系统上的函数，完成我们想要的操作。

### 翻译

Java 程序不一样，使用 javac 编译成 .class 文件之后，还需要使用 Java 命令去主动执行它，操作系统并不认识这些 .class 文件。所以JVM就是一个翻译。



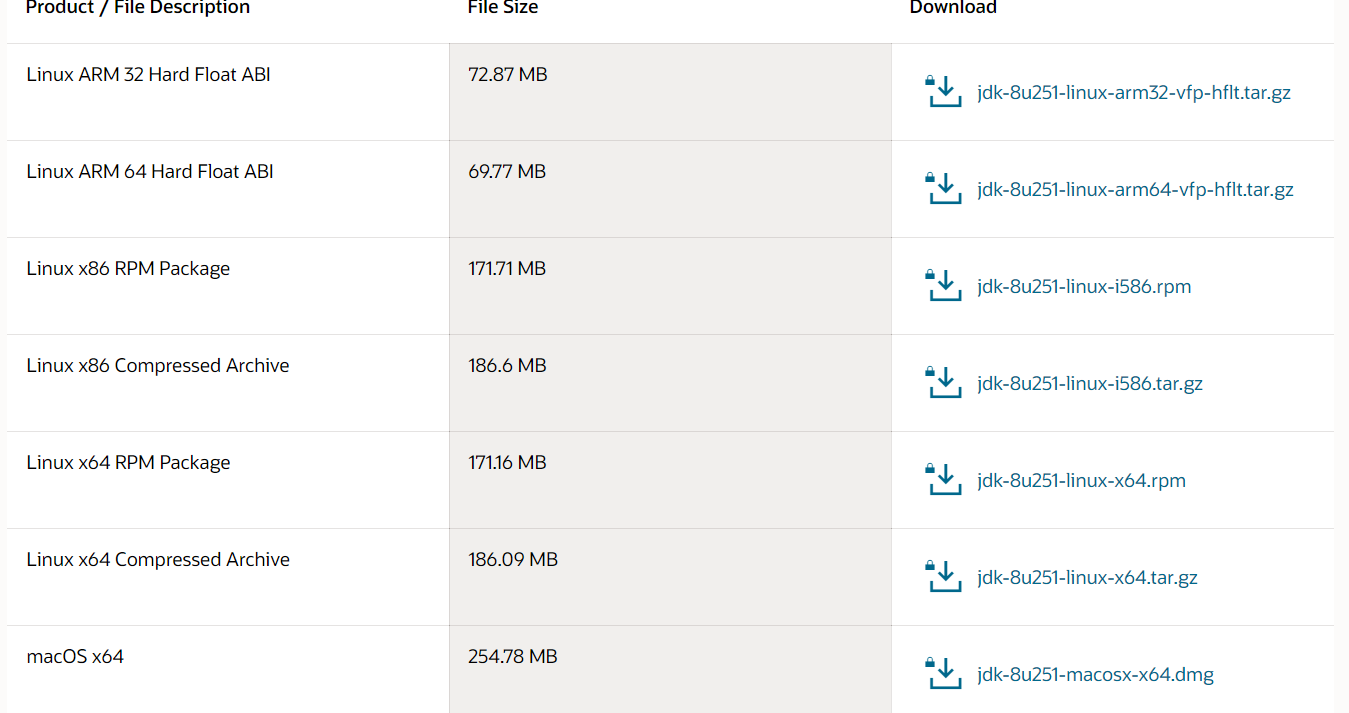
从图中可以看到，有了 JVM 这个抽象层之后，Java 就可以实现跨平台了。JVM 只需要保证能够正确执行 .class 文件，就可以运行在诸如 Linux、Windows、MacOS 等平台上了。

### 从跨平台到跨语言

**跨平台：**我们写的这个类Person这个类，在不同的操作系统上（Linux、Windows、MacOS 等平台）执行，效果是一样，这个就是JVM的跨平台性。

为了实现跨平台型，不同操作系统有不同的JDK的版本

<https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>



**跨语言：**JVM只识别字节码，所以JVM其实跟语言是解耦的，也就是没有直接关联，并不是它翻译Java文件，而是识别class文件，这个一般称之为字节码。还有像Groovy 、Kotlin、Jruby等等语言，它们其实也是编译成字节码，所以也可以在JVM上面跑，这个就是JVM的跨语言特征。

## JVM、JRE、JDK的关系

JVM只是一个翻译，把Class翻译成机器识别的代码，但是需要注意，JVM 不会自己生成代码，需要大家编写代码，同时需要很多依赖类库，这个时候就需要用到JRE。

JRE是什么，它除了包含JVM之外，提供了很多的类库（就是我们说的jar包，它可以提供一些即插即用的功能，比如读取或者操作文件，连接网络，使用I/O等等之类的）这些东西就是JRE提供的基础类库。JVM 标准加上实现的一大堆基础类库，就组成了 Java 的运行时环境，也就是我们常说的 JRE（Java Runtime Environment）。

但对于程序员来说，JRE还不够。我写完要编译代码，还需要调试代码，还需要打包代码、有时候还需要反编译代码。所以我们会使用JDK，因为JDK还提供了一些非常好用的小工具，比如 javac（编译代码）、java、jar （打包代码）、javap（反编译<反汇编>）等。这个就是JDK。

具体可以文档可以通过官网去下载：<https://www.oracle.com/java/technologies/javase-jdk8-doc-downloads.html>

# JVM整体

一个 Java 程序，首先经过 javac 编译成 .class 文件，然后 JVM 将其加载到方法区，执行引擎将会执行这些字节码。执行时，会翻译成操作系统相关的函数。JVM 作为 .class 文件的翻译存在，输入字节码，调用操作系统函数。

过程如下：Java 文件->编译器>字节码->JVM->机器码。

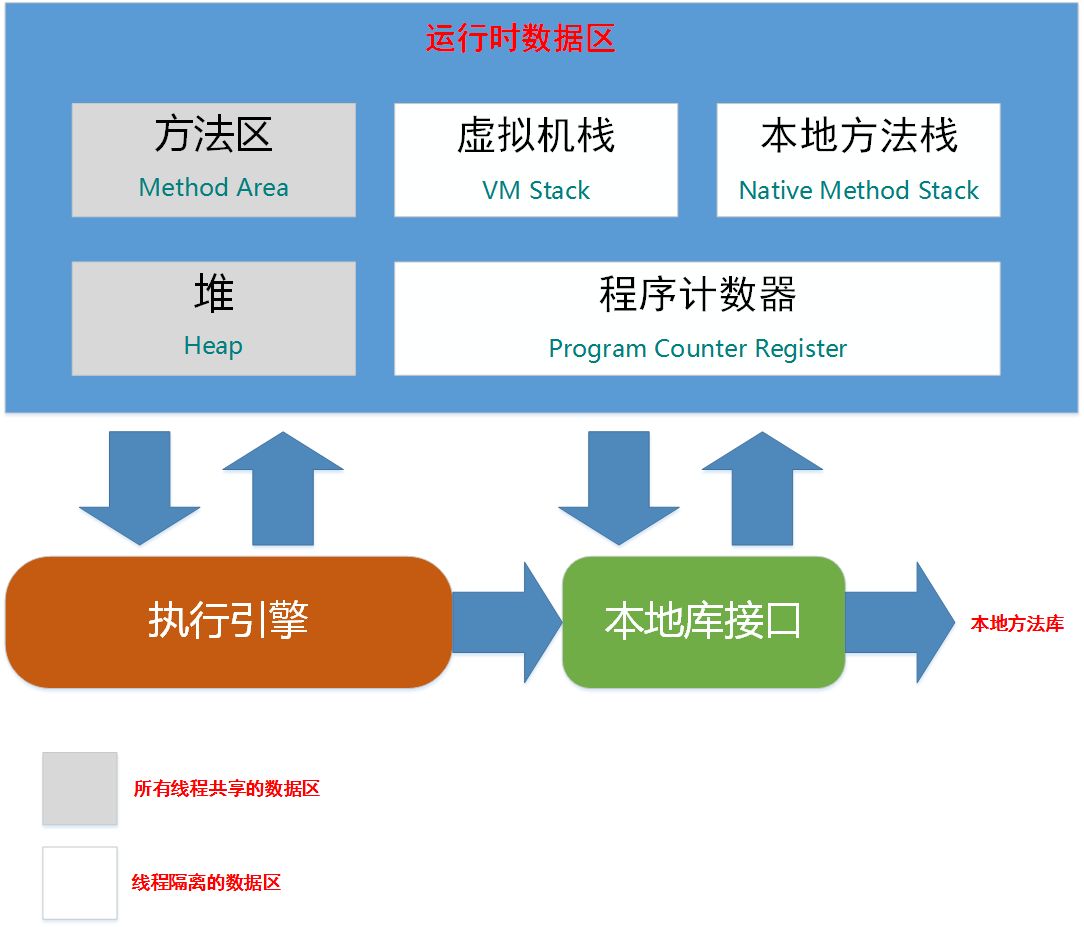
解释执行与GIT：

我们所说的 JVM，狭义上指的就 HotSpot（因为JVM有很多版本，但是使用最多的是HotSpot）。如非特殊说明，我们都以 HotSpot 为准。Java 之所以成为跨平台，就是由于 JVM 的存在。Java 的字节码，是沟通 Java 语言与 JVM 的桥梁，同时也是沟通 JVM 与操作系统的桥梁。

# 运行时数据区域

Java 引以为豪的就是它的自动内存管理机制。相比于 C++的手动内存管理、复杂难以理解的指针等，Java 程序写起来就方便的多。

在 Java 中，JVM 内存主要分为堆、程序计数器、方法区、虚拟机栈和本地方法栈。



# 程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响。

程序计数器是一块很小的内存空间，主要用来记录各个线程执行的字节码的地址，例如，分支、循环、跳转、异常、线程恢复等都依赖于计数器。

由于 Java 是多线程语言，当执行的线程数量超过 CPU 核数时，线程之间会根据时间片轮询争夺 CPU 资源。如果一个线程的时间片用完了，或者是其它原因导致这个线程的 CPU 资源被提前抢夺，那么这个退出的线程就需要单独的一个程序计数器，来记录下一条运行的指令。

程序计数器也是JVM中唯一不会OOM(OutOfMemory)的内存区域

# 虚拟机栈

栈是什么样的数据结构？先进后出(FILO)的数据结构，

虚拟机栈在JVM运行过程中存储当前线程运行方法所需的数据，指令、返回地址**。**

Java 虚拟机栈是基于线程的。哪怕你只有一个 main() 方法，也是以线程的方式运行的。在线程的生命周期中，参与计算的数据会频繁地入栈和出栈，栈的生命周期是和线程一样的。

栈里的每条数据，就是栈帧。在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦完成相应的调用，则出栈。所有的栈帧都出栈后，线程也就结束了。

每个栈帧，都包含四个区域：(局部变量表、操作数栈、动态连接、返回地址)

栈的大小缺省为1M，可用参数 –Xss调整大小，例如-Xss256k

**局部变量表:**顾名思义就是局部变量的表，用于存放我们的局部变量的。首先它是一个32位的长度，主要存放我们的Java的八大基础数据类型，一般32位就可以存放下，如果是64位的就使用高低位占用两个也可以存放下，如果是局部的一些对象，比如我们的Object对象，我们只需要存放它的一个引用地址即可。

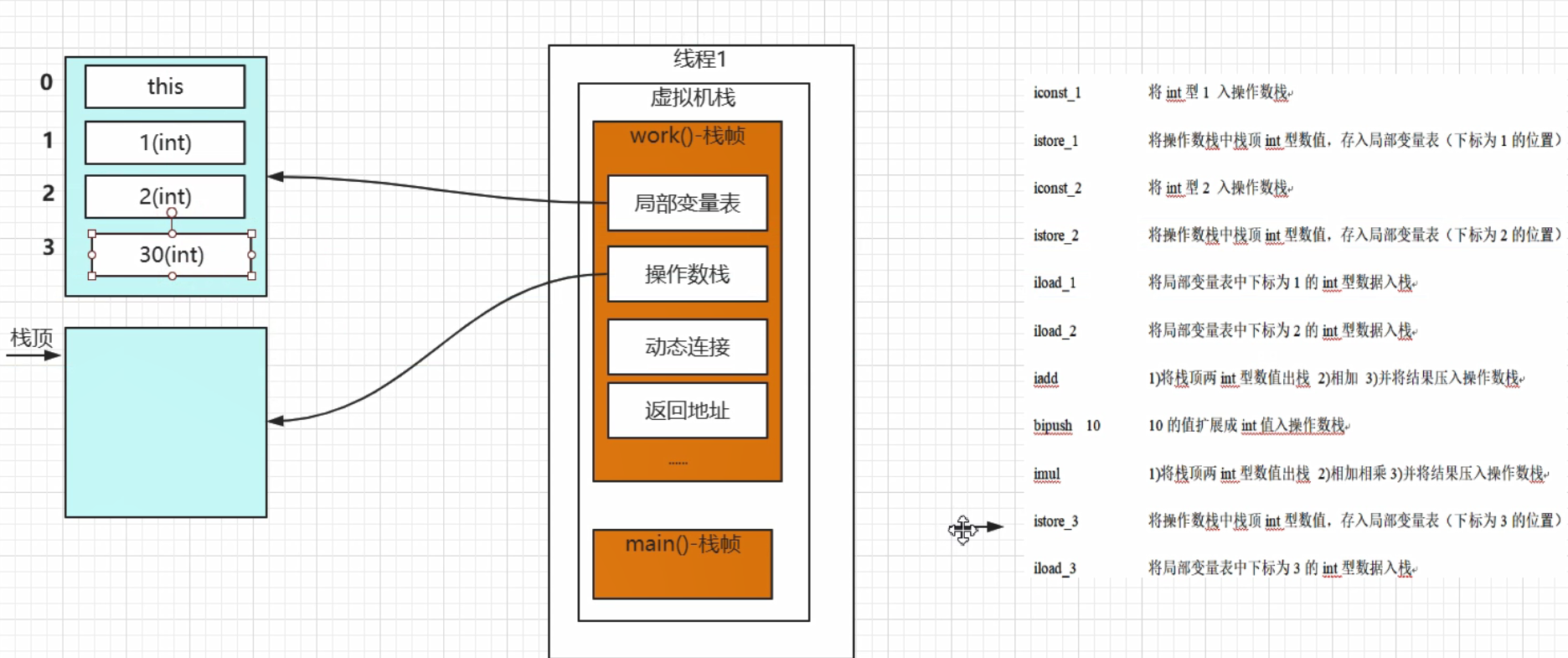
**操作数据栈**：存放我们方法执行的操作数的，它就是一个栈，先进后出的栈结构，操作数栈，就是用来操作的，操作的的元素可以是任意的java数据类型，所以我们知道一个方法刚刚开始的时候，这个方法的操作数栈就是空的，操作数栈运行方法就是JVM一直运行入栈/出栈的操作

**动态连接:**Java语言特性多态（需要类运行时才能确定具体的方法）。

**返回地址:**正常返回（调用程序计数器中的地址作为返回）、异常的话（通过异常处理器表<非栈帧中的>来确定）

# 栈帧执行对内存区域的影响

字节码助记码解释地址：<https://cloud.tencent.com/developer/article/1333540>



在JVM中，基于解释执行的这种方式是基于栈的引擎，这个说的栈，就是操作数栈。

# 本地方法栈

本地方法栈跟 Java 虚拟机栈的功能类似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈则用于管理本地方法的调用。但本地方法并不是用 Java 实现的，而是由 C 语言实现的。

本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域。

虚拟机规范无强制规定，各版本虚拟机自由实现 ，HotSpot直接把本地方法栈和虚拟机栈合二为一 。

# 线程共享的区域

## 方法区/永久代

很多开发者都习惯将方法区称为“永久代”，其实这两者并不是等价的。

HotSpot 虚拟机使用永久代来实现方法区，但在其它虚拟机中，例如，Oracle 的 JRockit、IBM 的 J9 就不存在永久代一说。因此，方法区只是 JVM 中规范的一部分，可以说，在 HotSpot 虚拟机中，设计人员使用了永久代来实现了 JVM 规范的方法区。

方法区主要是用来存放已被虚拟机加载的类相关信息，包括类信息、静态变量、常量、运行时常量池、字符串常量池。

JVM 在执行某个类的时候，必须先加载。在加载类（加载、验证、准备、解析、初始化）的时候，JVM 会先加载 class 文件，而在 class 文件中除了有类的版本、字段、方法和接口等描述信息外，还有一项信息是常量池 (Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。

字面量包括字符串（String a=“b”）、基本类型的常量（final 修饰的变量），符号引用则包括类和方法的全限定名（例如 String 这个类，它的全限定名就是 Java/lang/String）、字段的名称和描述符以及方法的名称和描述符。

而当类加载到内存中后，JVM 就会将 class 文件常量池中的内容存放到运行时的常量池中；在解析阶段，JVM 会把符号引用替换为直接引用（对象的索引值）。

例如，类中的一个字符串常量在 class 文件中时，存放在 class 文件常量池中的；在 JVM 加载完类之后，JVM 会将这个字符串常量放到运行时常量池中，并在解析阶段，指定该字符串对象的索引值。运行时常量池是全局共享的，多个类共用一个运行时常量池，class 文件中常量池多个相同的字符串在运行时常量池只会存在一份。

方法区与堆空间类似，也是一个共享内存区，所以方法区是线程共享的。假如两个线程都试图访问方法区中的同一个类信息，而这个类还没有装入 JVM，那么此时就只允许一个线程去加载它，另一个线程必须等待。在 HotSpot 虚拟机、Java7 版本中已经将永久代的静态变量和运行时常量池转移到了堆中，其余部分则存储在 JVM 的非堆内存中，而 Java8 版本已经将方法区中实现的永久代去掉了，并用元空间（class metadata）代替了之前的永久代，并且元空间的存储位置是本地

**元空间大小参数：**

jdk1.7及以前（初始和最大值）：-XX:PermSize；-XX:MaxPermSize；

jdk1.8以后（初始和最大值）：-XX:MetaspaceSize； -XX:MaxMetaspaceSize

jdk1.8以后大小就只受本机总内存的限制（如果不设置参数的话）

JVM参数参考：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

Java8 为什么使用元空间替代永久代，这样做有什么好处呢？

官方给出的解释是：

移除永久代是为了融合 HotSpot JVM 与 JRockit VM 而做出的努力，因为 JRockit 没有永久代，所以不需要配置永久代。

永久代内存经常不够用或发生内存溢出，抛出异常 java.lang.OutOfMemoryError: PermGen。这是因为在 JDK1.7 版本中，指定的 PermGen 区大小为 8M，由于 PermGen 中类的元数据信息在每次 FullGC 的时候都可能被收集，回收率都偏低，成绩很难令人满意；还有，为 PermGen 分配多大的空间很难确定，PermSize 的大小依赖于很多因素，比如，JVM 加载的 class 总数、常量池的大小和方法的大小等。

## 堆

堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

堆空间一般是程序启动时，就申请了，但是并不一定会全部使用。

随着对象的频繁创建，堆空间占用的越来越多，就需要不定期的对不再使用的对象进行回收。这个在 Java 中，就叫作 GC（Garbage Collection）。

那一个对象创建的时候，到底是在堆上分配，还是在栈上分配呢？这和两个方面有关：对象的类型和在 Java 类中存在的位置。

Java 的对象可以分为基本数据类型和普通对象。

对于普通对象来说，JVM 会首先在堆上创建对象，然后在其他地方使用的其实是它的引用。比如，把这个引用保存在虚拟机栈的局部变量表中。

对于基本数据类型来说（byte、short、int、long、float、double、char)，有两种情况。当你在方法体内声明了基本数据类型的对象，它就会在栈上直接分配。其他情况，都是在堆上分配。

**堆大小参数：**

-Xms：堆的最小值；

-Xmx：堆的最大值；

-Xmn：新生代的大小；

-XX:NewSize；新生代最小值；

-XX:MaxNewSize：新生代最大值；

例如- Xmx256m

# 直接内存

不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域；如果使用了NIO,这块区域会被频繁使用，在java堆内可以用directByteBuffer对象直接引用并操作；

这块内存不受java堆大小限制，但受本机总内存的限制，可以通过-XX:MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常。

# 从底层深入理解运行时数据区

开启HSDB工具

Jdk1.8启动JHSDB的时候必须将sawindbg.dll复制到对应目录的jre下



C:\Program Files\Java\jdk1.8.0\_101\lib

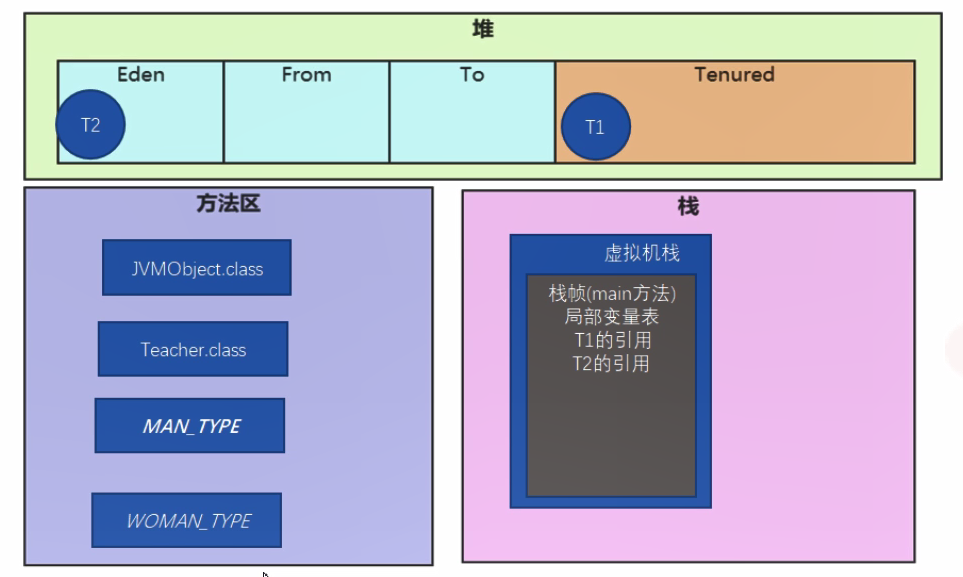
执行 java -cp .\sa-jdi.jar sun.jvm.hotspot.HSDB



当我们通过 Java 运行以上代码时，JVM 的整个处理过程如下：

1. JVM 向操作系统申请内存，JVM 第一步就是通过配置参数或者默认配置参数向操作系统申请内存空间。
2. JVM 获得内存空间后，会根据配置参数分配堆、栈以及方法区的内存大小。
3. 完成上一个步骤后， JVM 首先会执行构造器，编译器会在.java 文件被编译成.class 文件时，收集所有类的初始化代码，包括静态变量赋值语句、静态代码块、静态方法，静态变量和常量放入方法区
4. 执行方法。启动 main 线程，执行 main 方法，开始执行第一行代码。此时堆内存中会创建一个 Teacher 对象，对象引用 student 就存放在栈中。

执行其他方法时，具体的操作：栈帧执行对内存区域的影响。[栈帧执行对内存区域的影响](#_栈帧执行对内存区域的影响)



# 深入辨析堆和栈

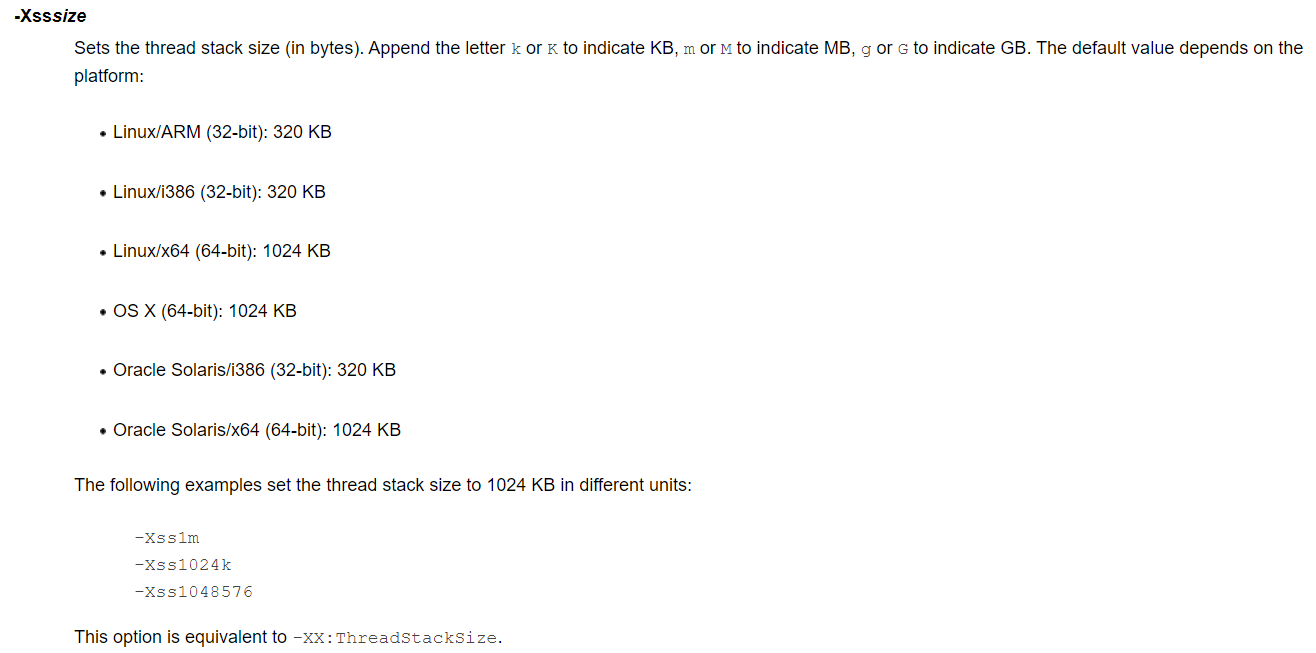
* **功能**
* 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char等）以及对象的引用变量，其内存分配在栈上，变量出了作用域就会自动释放；
* 而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；
* **线程独享还是共享**
* 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
* 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。
* **空间大小**

栈的内存要远远小于堆内存

# 内存溢出

## 栈溢出

参数：-Xss1m， 具体默认值需要查看官网：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BABHDABI>



HotSpot版本中栈的大小是固定的，是不支持拓展的。

java.lang.StackOverflowError 一般的方法调用是很难出现的，如果出现了可能会是无限递归。

虚拟机栈带给我们的启示：方法的执行因为要打包成栈桢，所以天生要比实现同样功能的循环慢，所以树的遍历算法中：递归和非递归(循环来实现)都有存在的意义。递归代码简洁，非递归代码复杂但是速度较快。

OutOfMemoryError：不断建立线程，JVM申请栈内存，机器没有足够的内存。（一般演示不出，演示出来机器也死了）

## 堆溢出

内存溢出：申请内存空间,超出最大堆内存空间。

如果是内存溢出，则通过 调大 -Xms，-Xmx参数。

如果不是内存泄漏，就是说内存中的对象却是都是必须存活的，那么久应该检查JVM的堆参数设置，与机器的内存对比，看是否还有可以调整的空间，再从代码上检查是否存在某些对象生命周期过长、持有状态时间过长、存储结构设计不合理等情况，尽量减少程序运行时的内存消耗。

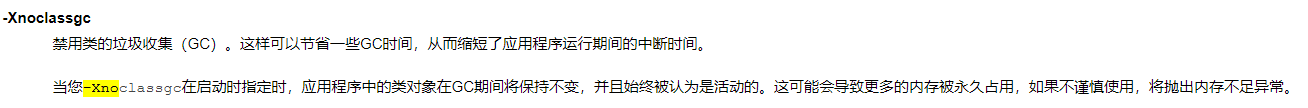
## 方法区溢出

**（1） 运行时常量池溢出**

**（2）方法区中保存的Class对象没有被及时回收掉或者Class信息占用的内存超过了我们配置。**

**注意Class要被回收，条件比较苛刻（仅仅是可以，不代表必然，因为还有一些参数可以进行控制）：**

1. **该类所有的实例都已经被回收，也就是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。



**代码示例：**

**cglib是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。**

**CGLIB包的底层是通过使用一个小而快的**[**字节码**](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)**处理框架ASM，来转换字节码并生成新的类。除了CGLIB包，**[**脚本语言**](https://baike.baidu.com/item/%E8%84%9A%E6%9C%AC%E8%AF%AD%E8%A8%80)**例如Groovy和BeanShell，也是使用ASM来生成java的字节码。当然不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

## 本机直接内存溢出

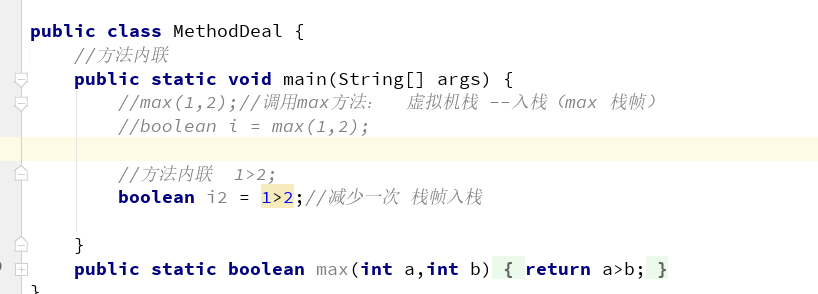
直接内存的容量可以通过MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常；

由直接内存导致的内存溢出，一个比较明显的特征是在HeapDump文件中不会看见有什么明显的异常情况，如果发生了OOM，同时Dump文件很小，可以考虑重点排查下直接内存方面的原因。

# 虚拟机优化技术

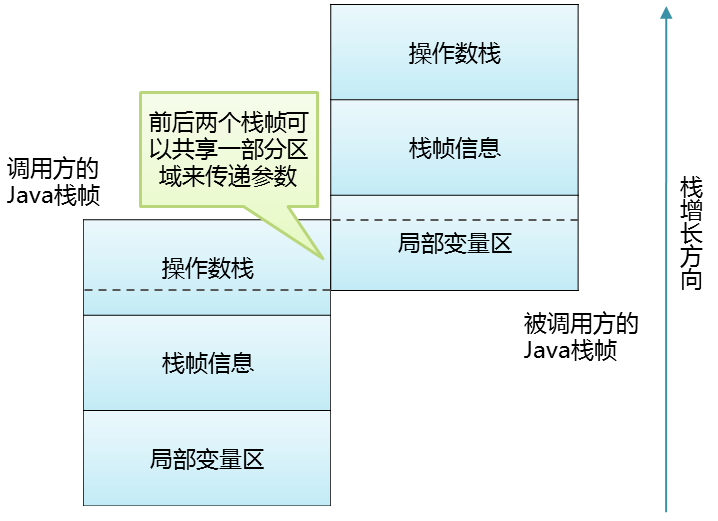
## 编译优化技术——方法内联

方法内联的优化行为，就是把目标方法的代码原封不动的“复制”到调用的方法中，避免真实的方法调用而已。



## 栈的优化技术——栈帧之间数据的共享

在一般的模型中，两个不同的栈帧的内存区域是独立的，但是大部分的JVM在实现中会进行一些优化，使得两个栈帧出现一部分重叠。（主要体现在方法中有参数传递的情况），让下面栈帧的操作数栈和上面栈帧的部分局部变量重叠在一起，这样做不但节约了一部分空间，更加重要的是在进行方法调用时就可以直接公用一部分数据，无需进行额外的参数复制传递了。



使用HSDB工具查看栈空间一样可以看到。

