# 类加载子系统

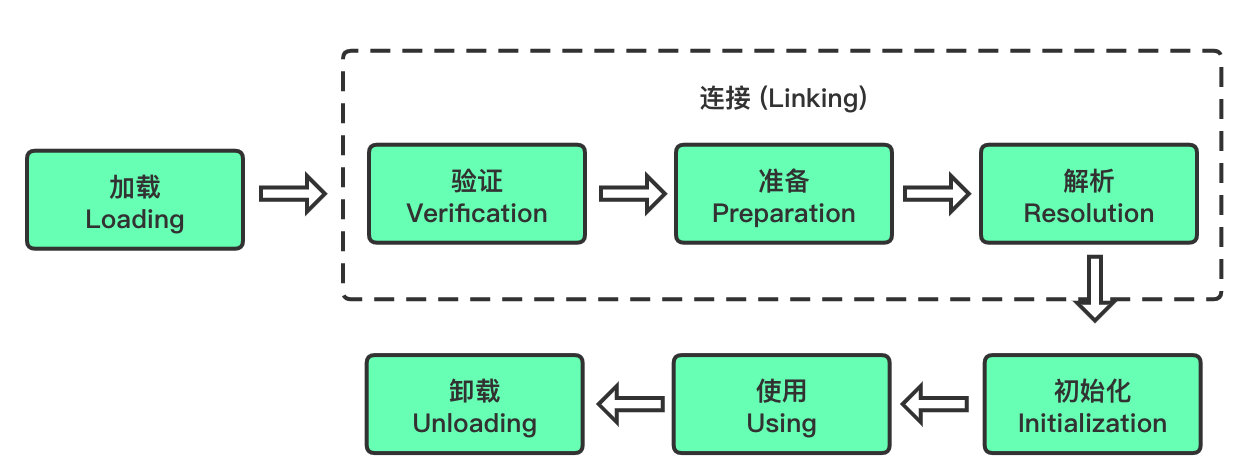
## 类加载机制

Java 虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的加载机制。

把class文件加载到内存，生成java类给虚拟机使用

## 类加载过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：**加载、验证、准备、解析、初始化、使用和卸载**七个阶段。(验证、准备和解析又统称为连接，为了支持 Java 语言的**运行时绑定**，所以解析阶段也可以是在初始化之后进行的。以上顺序都只是说开始的顺序，实际过程中是交叉的混合式进行的，加载过程中可能就已经开始验证了)



### 加载（Loading）

1. 通过一个类的全限定名获取定义此类的二进制字节流
2. 将这个字节流所代表的的静态存储结构转化为方法区的运行时数据结构
3. 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的访问入口

### 连接（Linking）

**验证**（Verify）

1. 目的在于确保 Class 文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全
2. 主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证

**准备**（Prepare）

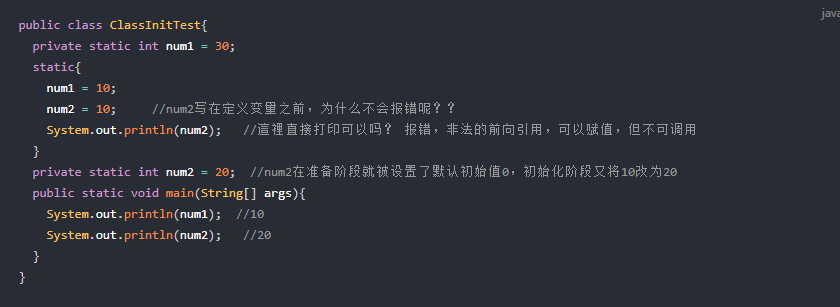
1. 为类变量分配内存并且设置该类变量的默认初始值，即零值
2. 这里不包含用 final 修饰的 static，因为 final 在编译的时候就会分配了，准备阶段会显示初始化
3. 这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到 Java 堆中

**解析**（Resolve）

1. 将常量池内的符号引用转换为直接引用的过程
2. 事实上，解析操作往往会伴随着 JVM 在执行完初始化之后再执行
3. 符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《Java虚拟机规范》的 Class文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄
4. 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等

### 初始化（Initialization）

1. 初始化阶段就是执行类构造器方法 <clinit>() 的过程
2. 此方法不需要定义，是 javac 编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来
3. 构造器方法中指令按语句在源文件中出现的顺序执行
4. <clinit>() 不同于类的构造器（构造器是虚拟机视角下的 <init>()）
5. 若该类具有父类，JVM 会保证子类的 <clinit>() 执行前，父类的 <clinit>() 已经执行完毕
6. 虚拟机必须保证一个类的 <clinit>() 方法在多线程下被同步加锁



## 类加载器

* JVM 支持两种类型的类加载器，分别为引导类加载器（Bootstrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）
* 从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器

### 启动类加载器（引导类加载器，Bootstrap ClassLoader）

这个类加载使用 C/C++ 语言实现，嵌套在 JVM 内部

它用来加载 Java 的核心库（JAVA\_HOME/jre/lib/rt.jar、resource.jar或sun.boot.class.path路径下的内容），用于提供 JVM 自身需要的类

并不继承自 java.lang.ClassLoader，没有父加载器

加载扩展类和应用程序类加载器，并指定为他们的父类加载器

出于安全考虑，Bootstrap 启动类加载器只加载名为 java、Javax、sun 等开头的类

### 扩展类加载器（Extension ClassLoader）

Java 语言编写，由 sun.misc.Launcher$ExtClassLoader 实现

派生于 ClassLoader

父类加载器为启动类加载器

从 java.ext.dirs 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 jre/lib/ext 子目录（扩展目录）下加载类库。如果用户创建的 JAR 放在此目录下，也会自动由扩展类加载器加载

### 应用程序类加载器（也叫系统类加载器，AppClassLoader）

Java 语言编写，由 sun.misc.Lanucher$AppClassLoader 实现

派生于 ClassLoader

父类加载器为扩展类加载器

它负责加载环境变量 classpath 或系统属性 java.class.path 指定路径下的类库

该类加载是程序中默认的类加载器，一般来说，Java 应用的类都是由它来完成加载的

通过 ClassLoader#getSystemClassLoader() 方法可以获取到该类加载器

### ClassLoader

ClassLoader 类，是一个抽象类，其后所有的类加载器都继承自 ClassLoader（不包括启动类加载器）

### 对类加载器的引用

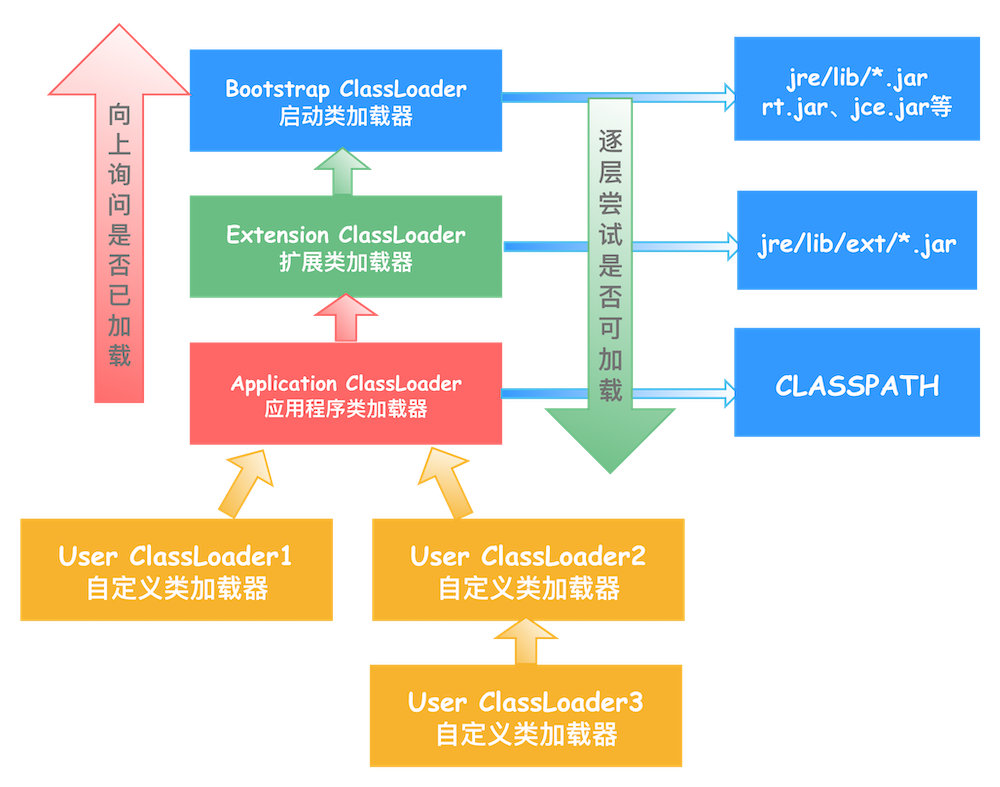
JVM 必须知道一个类型是由启动加载器加载的还是由用户类加载器加载的。如果一个类型是由用户类加载器加载的，那么 JVM 会将这个类加载器的一个引用作为类型信息的一部分保存在方法区中。当解析一个类型到另一个类型的引用的时候，JVM 需要保证这两个类型的类加载器是相同的。

## 双亲委派机制

Java 虚拟机对 class 文件采用的是按需加载的方式，也就是说当需要使用该类的时候才会将它的 class 文件加载到内存生成 class 对象。而且加载某个类的 class 文件时，Java 虚拟机采用的是双亲委派模式，即把请求交给父类处理，它是一种任务委派模式。

### 工作过程

1. 如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行；
2. 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器；
3. 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式



注：

1. 越往上层加载器，加载的类越是底层依赖类，自定义类加载器在最下层（子层），加载自定义类。
2. 这样保证最开始加载最底层的类，上层类在使用能直接调用不用重复加载。

### 优势

避免类的重复加载，JVM 中区分不同类，不仅仅是根据类名，相同的 class 文件被不同的 ClassLoader 加载就属于两个不同的类（比如，Java中的Object类，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，如果不采用双亲委派模型，由各个类加载器自己去加载的话，系统中会存在多种不同的 Object 类）

保护程序安全，防止核心 API 被随意篡改，避免用户自己编写的类动态替换 Java 的一些核心类，比如我们自定义类：java.lang.String

在 JVM 中表示两个 class 对象是否为同一个类存在两个必要条件：

类的完整类名必须一致，包括包名

加载这个类的 ClassLoader（指ClassLoader实例对象）必须相同

### 沙箱安全机制

如果我们自定义 String 类，但是在加载自定义 String 类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载 jdk 自带的文件（rt.jar包中 java\lang\String.class），报错信息说没有 main 方法就是因为加载的是rt.jar包中的 String 类。这样就可以保证对 java 核心源代码的保护，这就是简单的沙箱安全机制。

### 破坏双亲委派模型

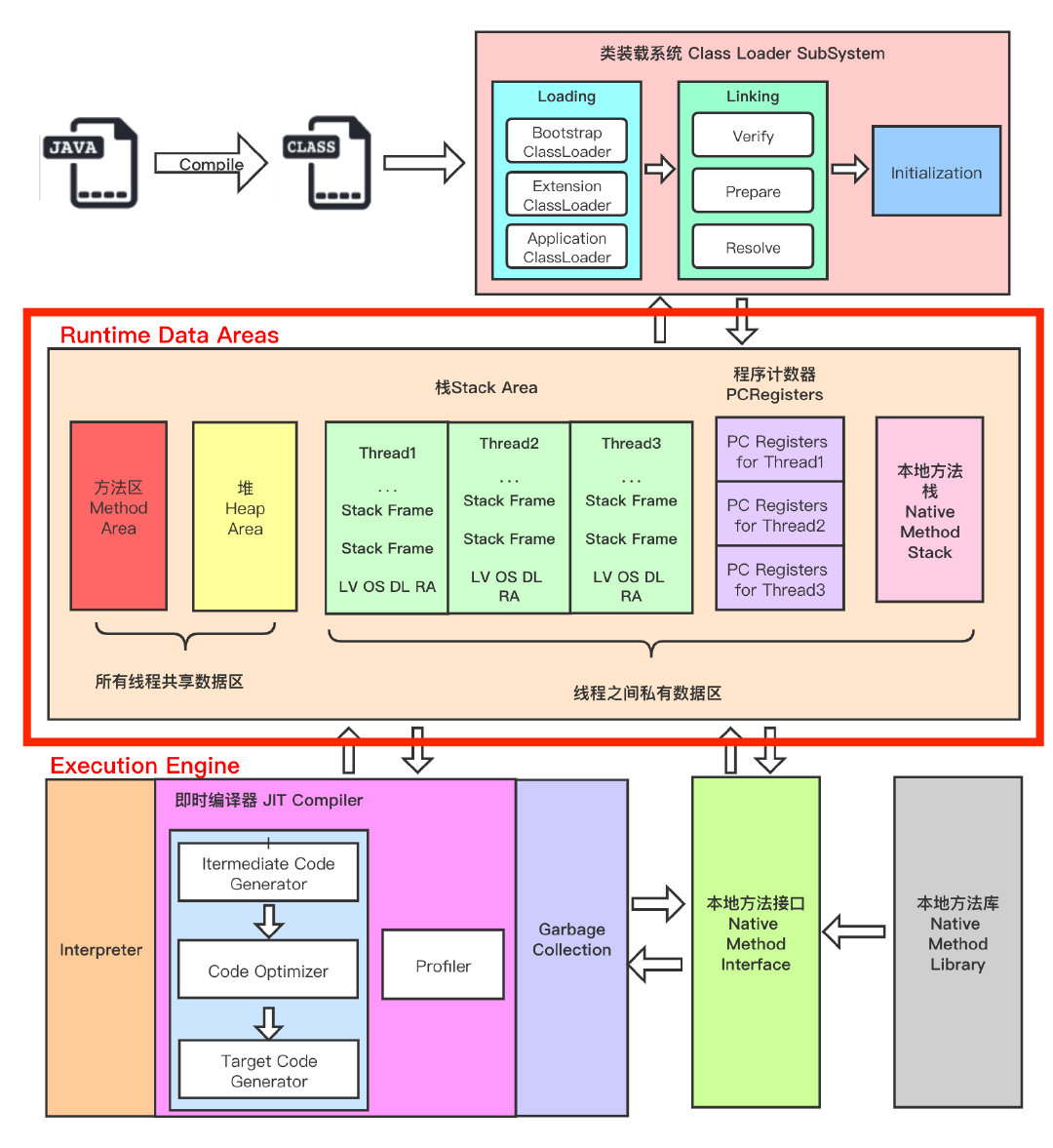
1. 双亲委派模型并不是一个强制性的约束模型，而是 Java 设计者推荐给开发者的类加载器实现方式，可以“被破坏”，只要我们自定义类加载器，重写 loadClass() 方法，指定新的加载逻辑就破坏了，重写 findClass() 方法不会破坏双亲委派。
2. 双亲委派模型有一个问题：顶层 ClassLoader，无法加载底层 ClassLoader 的类。典型例子 JNDI、JDBC，所以加入了线程上下文类加载器（Thread Context ClassLoader），可以通过 Thread.setContextClassLoaser()设置该类加载器，然后顶层 ClassLoader 再使用 Thread.getContextClassLoader() 获得底层的 ClassLoader 进行加载。
3. Tomcat 中使用了自定 ClassLoader，并且也破坏了双亲委托机制。每个应用使用 WebAppClassloader 进行单独加载，他首先使用 WebAppClassloader 进行类加载，如果加载不了再委托父加载器去加载，这样可以保证每个应用中的类不冲突。每个 tomcat 中可以部署多个项目，每个项目中存在很多相同的 class 文件（很多相同的jar包），他们加载到 jvm 中可以做到互不干扰。
4. 利用破坏双亲委派来实现代码热替换（每次修改类文件，不需要重启服务）。因为一个 Class 只能被一个 ClassLoader 加载一次，否则会报 java.lang.LinkageError。当我们想要实现代码热部署时，可以每次都 new 一个自定义的 ClassLoader 来加载新的 Class文件。JSP 的实现动态修改就是使用此特性实现。

# jvm内存结构

Java 虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程一一对应的数据区域会随着线程开始和结束而创建和销毁。

线程私有：程序计数器、栈、本地栈

线程共享：堆、堆外内存（永久代或元空间、代码缓存）



## 程序计数器

程序计数寄存器（Program Counter Register），Register 的命名源于 CPU 的寄存器，寄存器存储指令相关的线程信息，CPU 只有把数据装载到寄存器才能够运行。

这里，并非是广义上所指的物理寄存器，叫程序计数器（或PC计数器或指令计数器）会更加贴切，并且也不容易引起一些不必要的误会。JVM 中的 PC 寄存器是对物理 PC 寄存器的一种抽象模拟。

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。

### 作用

PC 寄存器用来存储指向下一条指令的地址，即将要执行的指令代码。由执行引擎读取下一条指令。

### 概述

它是一块很小的内存空间，几乎可以忽略不计。也是运行速度最快的存储区域

在 JVM 规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期一致

任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。如果当前线程正在执行的是 Java 方法，程序计数器记录的是 JVM 字节码指令地址，如果是执行 natice 方法，则是未指定值（undefined）

它是程序控制流的指示器，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

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

它是唯一一个在 JVM 规范中没有规定任何 OutOfMemoryError 情况的区域

## 虚拟机栈

### 概述

Java 虚拟机栈(Java Virtual Machine Stacks)，早期也叫 Java 栈。每个线程在创建的时候都会创建一个虚拟机栈，其内部保存一个个的栈帧(Stack Frame），对应着一次次 Java 方法调用，是线程私有的，生命周期和线程一致。

作用：主管 Java 程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回。

特点：

* 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器
* JVM 直接对虚拟机栈的操作只有两个：每个方法执行，伴随着入栈（进栈/压栈），方法执行结束出栈
* 栈不存在垃圾回收问题

栈中可能出现的异常：

Java 虚拟机规范允许 Java虚拟机栈的大小是动态的或者是固定不变的

* 如果采用固定大小的 Java 虚拟机栈，那每个线程的 Java 虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过 Java 虚拟机栈允许的最大容量，Java 虚拟机将会抛出一个 StackOverflowError 异常
* 如果 Java 虚拟机栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那 Java 虚拟机将会抛出一个OutOfMemoryError异常

可以通过参数-Xss来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可达深度。

### 栈的存储单位

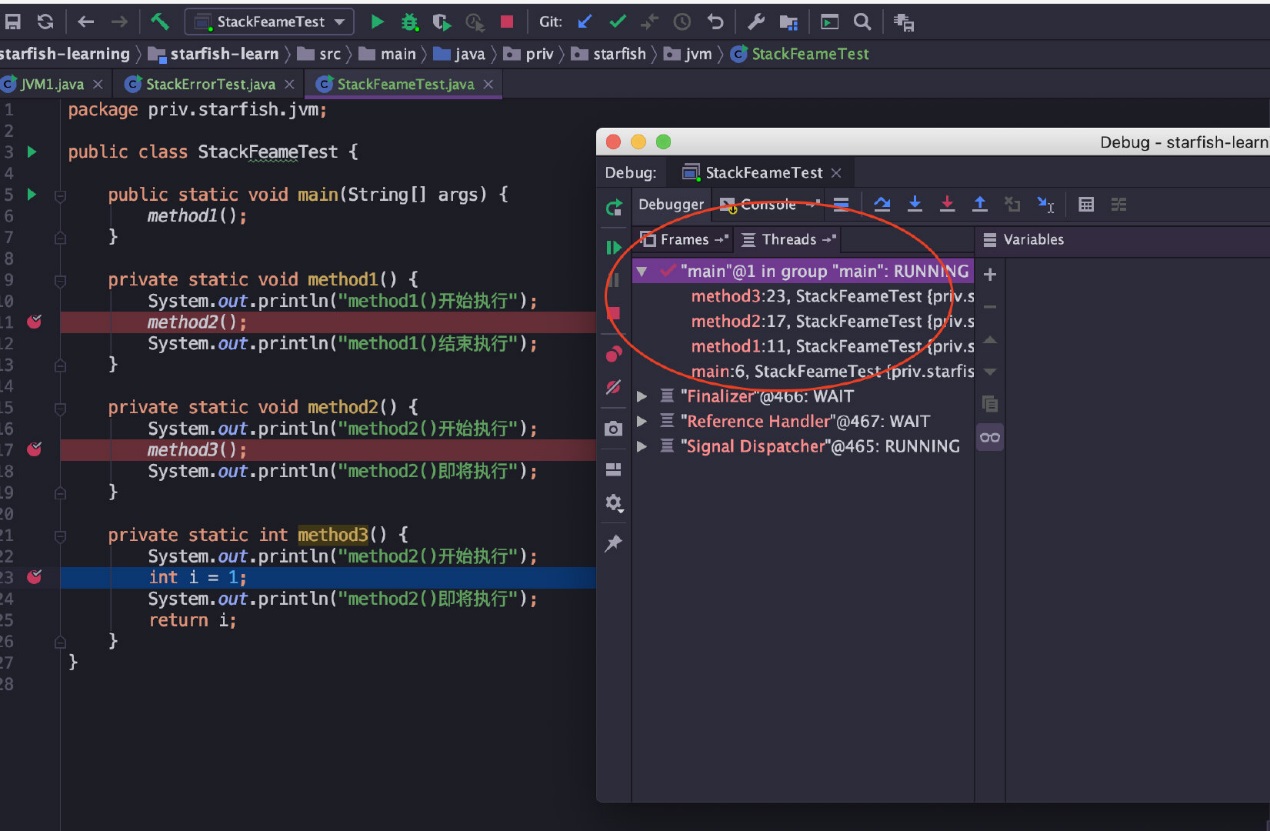
栈中存储什么？

* 每个线程都有自己的栈，栈中的数据都是以栈帧（Stack Frame）的格式存在
* 在这个线程上正在执行的每个方法都各自有对应的一个栈帧
* 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息

### 栈运行原理

* JVM 直接对 Java 栈的操作只有两个，对栈帧的压栈和出栈，遵循“先进后出/后进先出”原则
* 在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为当前栈帧（Current Frame），与当前栈帧对应的方法就是当前方法（Current Method），定义这个方法的类就是当前类（Current Class）
* 执行引擎运行的所有字节码指令只针对当前栈帧进行操作
* 如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，称为新的当前栈帧
* 不同线程中所包含的栈帧是不允许存在相互引用的，即不可能在一个栈帧中引用另外一个线程的栈帧
* 如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧
* Java 方法有两种返回函数的方式，一种是正常的函数返回，使用 return 指令，另一种是抛出异常，不管用哪种方式，都会导致栈帧被弹出

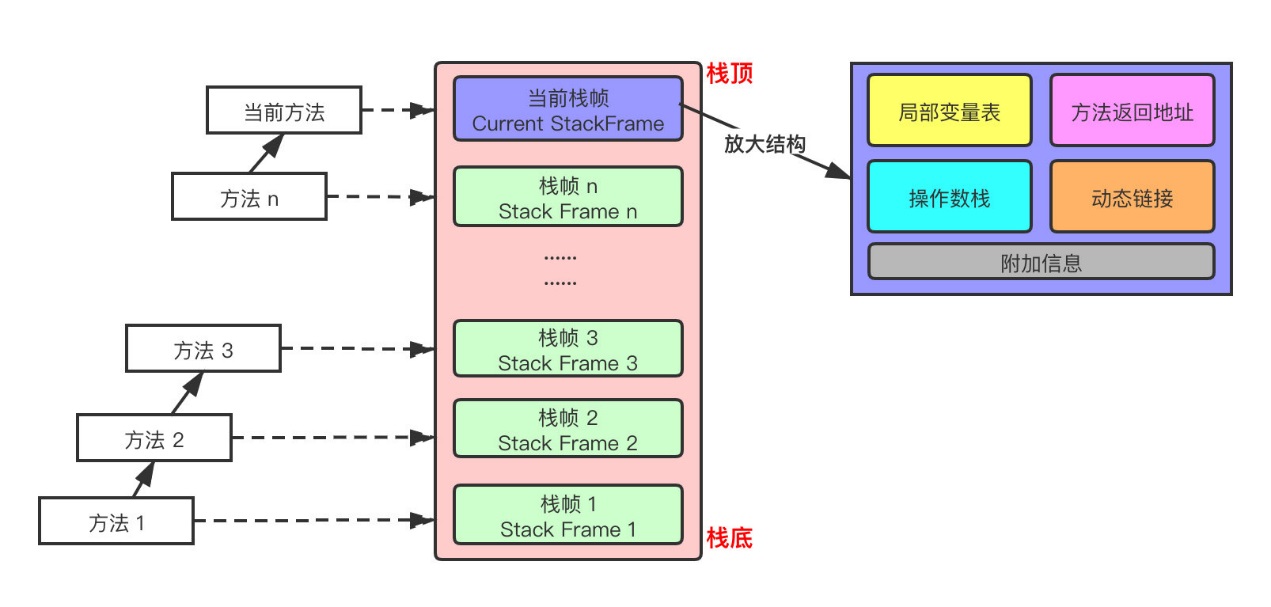
IDEA 在 debug 时候，可以在 debug 窗口看到 Frames 中各种方法的压栈和出栈情况



### 栈帧的内部结构

每个栈帧（Stack Frame）中存储着：

* 局部变量表（Local Variables）
* 操作数栈（Operand Stack）(或称为表达式栈)
* 动态链接（Dynamic Linking）：指向运行时常量池的方法引用
* 方法返回地址（Return Address）：方法正常退出或异常退出的地址
* 一些附加信息



#### 局部变量表

* 局部变量表也被称为局部变量数组或者本地变量表
* 是一组变量值存储空间，主要用于存储方法参数和定义在方法体内的局部变量，包括编译器可知的各种 Java 虚拟机基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它并不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此相关的位置）和 returnAddress 类型（指向了一条字节码指令的地址，已被异常表取代）
* 由于局部变量表是建立在线程的栈上，是线程的私有数据，因此不存在数据安全问题
* 局部变量表所需要的容量大小是编译期确定下来的，并保存在方法的 Code 属性的 maximum local variables 数据项中。在方法运行期间是不会改变局部变量表的大小的
* 方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，它的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越大，以满足方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间，导致其嵌套调用次数就会减少。
* 局部变量表中的变量只在当前方法调用中有效。在方法执行时，虚拟机通过使用局部变量表完成参数值到参数变量列表的传递过程。当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。
* 参数值的存放总是在局部变量数组的 index0 开始，到数组长度 -1 的索引结束

槽 Slot

* 局部变量表最基本的存储单元是 Slot（变量槽）
* 在局部变量表中，32 位以内的类型只占用一个 Slot(包括returnAddress类型)，64 位的类型（long和double）占用两个连续的 Slot
* byte、short、char 在存储前被转换为 int，boolean 也被转换为 int，0 表示 false，非 0 表示 true
* long 和 double 则占据两个 Slot
* JVM 会为局部变量表中的每一个 Slot 都分配一个访问索引，通过这个索引即可成功访问到局部变量表中指定的局部变量值，索引值的范围从 0 开始到局部变量表最大的 Slot 数量
* 当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照顺序被复制到局部变量表中的每一个 Slot 上
* 如果需要访问局部变量表中一个 64bit 的局部变量值时，只需要使用前一个索引即可。（比如：访问 long 或 double 类型变量，不允许采用任何方式单独访问其中的某一个 Slot）
* 如果当前帧是由构造方法或实例方法创建的，那么该对象引用 this 将会存放在 index 为 0 的 Slot 处，其余的参数按照参数表顺序继续排列（这里就引出一个问题：静态方法中为什么不可以引用 this，就是因为 this 变量不存在于当前方法的局部变量表中）
* 栈帧中的局部变量表中的槽位是可以重用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。（下图中，this、a、b、c 理论上应该有 4 个变量，c 复用了 b 的槽）



* 在栈帧中，与性能调优关系最为密切的就是局部变量表。在方法执行时，虚拟机使用局部变量表完成方法的传递
* 局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收

#### 操作数栈

* 每个独立的栈帧中除了包含局部变量表之外，还包含一个后进先出（Last-In-First-Out）的操作数栈，也可以称为表达式栈（Expression Stack）
* 操作数栈，在方法执行过程中，根据字节码指令，往操作数栈中写入数据或提取数据，即入栈（push）、出栈（pop）
* 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈。比如，执行复制、交换、求和等操作

#概述

* 操作数栈，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临时的存储空间
* 操作数栈就是 JVM 执行引擎的一个工作区，当一个方法刚开始执行的时候，一个新的栈帧也会随之被创建出来，此时这个方法的操作数栈是空的
* 每一个操作数栈都会拥有一个明确的栈深度用于存储数值，其所需的最大深度在编译期就定义好了，保存在方法的 Code 属性的 max\_stack 数据项中
* 栈中的任何一个元素都可以是任意的 Java 数据类型

32bit 的类型占用一个栈单位深度

64bit 的类型占用两个栈单位深度

* 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
* 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新 PC 寄存器中下一条需要执行的字节码指令
* 操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译期间进行验证，同时在类加载过程中的类检验阶段的数据流分析阶段要再次验证
* 另外，我们说 Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈

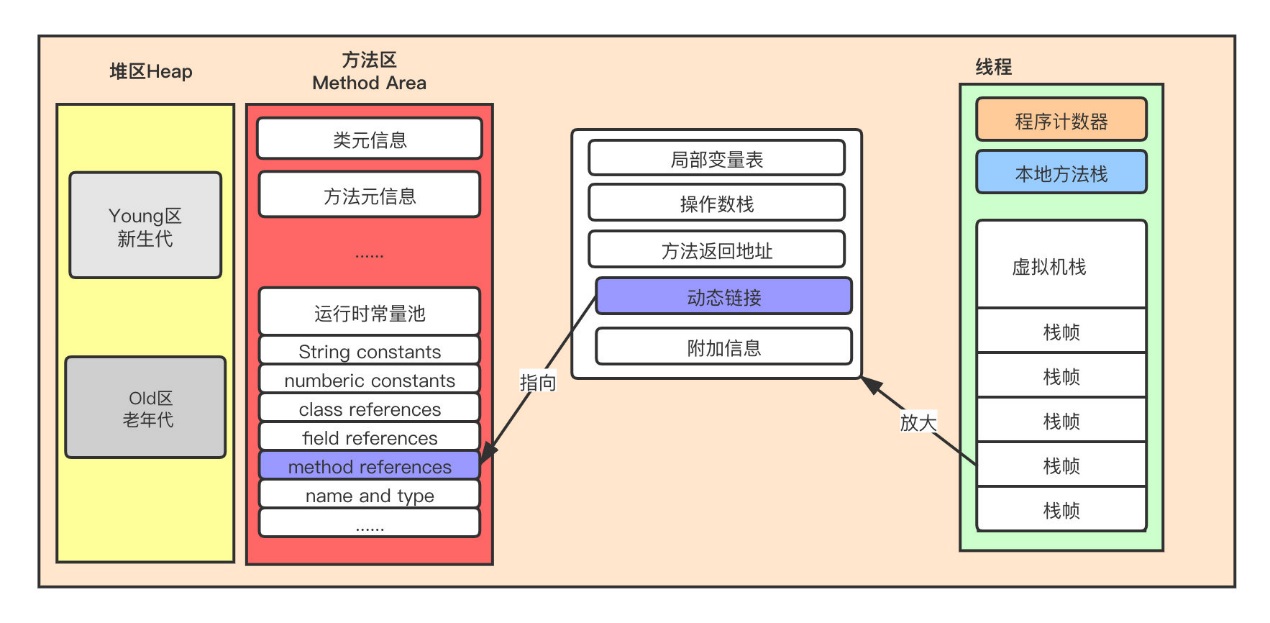
栈顶缓存（Top-of-stack-Cashing）

HotSpot 的执行引擎采用的并非是基于寄存器的架构，但这并不代表 HotSpot VM 的实现并没有间接利用到寄存器资源。寄存器是物理 CPU 中的组成部分之一，它同时也是 CPU 中非常重要的高速存储资源。一般来说，寄存器的读/写速度非常迅速，甚至可以比内存的读/写速度快上几十倍不止，不过寄存器资源却非常有限，不同平台下的 CPU 寄存器数量是不同和不规律的。寄存器主要用于缓存本地机器指令、数值和下一条需要被执行的指令地址等数据。

基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。由于操作数是存储在内存中的，因此频繁的执行内存读/写操作必然会影响执行速度。为了解决这个问题，HotSpot JVM 设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率

#### 动态链接（指向运行时常量池的方法引用）

* 每一个栈帧内部都包含一个指向运行时常量池中该栈帧所属方法的引用。包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接(Dynamic Linking)。
* 在 Java 源文件被编译到字节码文件中时，所有的变量和方法引用都作为符号引用（Symbolic Reference）保存在 Class 文件的常量池中。比如：描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用来表示的，那么动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用



JVM 是如何执行方法调用的

方法调用不同于方法执行，方法调用阶段的唯一任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。Class 文件的编译过程中不包括传统编译器中的连接步骤，一切方法调用在 Class文件里面存储的都是符号引用，而不是方法在实际运行时内存布局中的入口地址（直接引用）。也就是需要在类加载阶段，甚至到运行期才能确定目标方法的直接引用。

【这一块内容，除了方法调用，还包括解析、分派（静态分派、动态分派、单分派与多分派），这里先不介绍，后续再挖】

在 JVM 中，将符号引用转换为调用方法的直接引用与方法的绑定机制有关

**静态链接**：当一个字节码文件被装载进 JVM 内部时，如果被调用的目标方法在编译期可知，且运行期保持不变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接

**动态链接**：如果被调用的方法在编译期无法被确定下来，也就是说，只能在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动态链接

对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）。**绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次**。

早期绑定：**早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时**，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。

晚期绑定：如果被调用的方法在编译器无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式就被称为晚期绑定。

虚方法和非虚方法

如果方法在编译器就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非虚方法，比如静态方法、私有方法、final 方法、实例构造器、父类方法都是非虚方法

其他方法称为虚方法

虚方法表

在面向对象编程中，会频繁的使用到动态分派，如果每次动态分派都要重新在类的方法元数据中搜索合适的目标有可能会影响到执行效率。为了提高性能，JVM 采用在类的方法区建立一个虚方法表（virtual method table），使用索引表来代替查找。非虚方法不会出现在表中。

每个类中都有一个虚方法表，表中存放着各个方法的实际入口。

虚方法表会在类加载的连接阶段被创建并开始初始化，类的变量初始值准备完成之后，JVM 会把该类的方法表也初始化完毕。

#### 方法返回地址（return address）

用来存放调用该方法的 PC 寄存器的值。

一个方法的结束，有两种方式

正常执行完成

出现未处理的异常，非正常退出

无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的 PC 计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出的，返回地址是要通过异常表来确定的，栈帧中一般不会保存这部分信息。

当一个方法开始执行后，只有两种方式可以退出这个方法：

执行引擎遇到任意一个方法返回的字节码指令，会有返回值传递给上层的方法调用者，简称正常完成出口

一个方法的正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类型而定

在字节码指令中，返回指令包含 ireturn(当返回值是 boolean、byte、char、short 和 int 类型时使用)、lreturn、freturn、dreturn 以及 areturn，另外还有一个 return 指令供声明为 void 的方法、实例初始化方法、类和接口的初始化方法使用。

在方法执行的过程中遇到了异常，并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出。简称异常完成出口

方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码。

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、将返回值压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。

正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值

### 本地方法栈

#### 本地方法接口

简单的讲，一个 Native Method 就是一个 Java 调用非 Java 代码的接口。我们知道的 Unsafe 类就有很多本地方法。

为什么要使用本地方法（Native Method）?

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

* 与 Java 环境外交互：有时 Java 应用需要与 Java 外面的环境交互，这就是本地方法存在的原因。
* 与操作系统交互：JVM 支持 Java 语言本身和运行时库，但是有时仍需要依赖一些底层系统的支持。通过本地方法，我们可以实现用 Java 与实现了 jre 的底层系统交互， JVM 的一些部分就是 C 语言写的。
* Sun's Java：Sun的解释器就是C实现的，这使得它能像一些普通的C一样与外部交互。jre大部分都是用 Java 实现的，它也通过一些本地方法与外界交互。比如，类 java.lang.Thread 的 setPriority() 的方法是用Java 实现的，但它实现调用的是该类的本地方法 setPrioruty()，该方法是C实现的，并被植入 JVM 内部。

#### 本地方法栈（Native Method Stack）

1. Java 虚拟机栈用于管理 Java 方法的调用，而本地方法栈用于管理本地方法的调用
2. 本地方法栈也是线程私有的
3. 允许线程固定或者可动态扩展的内存大小
4. 如果线程请求分配的栈容量超过本地方法栈允许的最大容量，Java 虚拟机将会抛出一个 StackOverflowError 异常
5. 如果本地方法栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的本地方法栈，那么 Java虚拟机将会抛出一个OutofMemoryError异常
6. 本地方法是使用 C 语言实现的
7. 它的具体做法是 Mative Method Stack 中登记 native 方法，在 Execution Engine 执行时加载本地方法库当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。它和虚拟机拥有同样的权限。
8. 本地方法可以通过本地方法接口来访问虚拟机内部的运行时数据区，它甚至可以直接使用本地处理器中的寄存器，直接从本地内存的堆中分配任意数量的内存
9. 并不是所有 JVM 都支持本地方法。因为 Java 虚拟机规范并没有明确要求本地方法栈的使用语言、具体实现方式、数据结构等。如果 JVM 产品不打算支持 native 方法，也可以无需实现本地方法栈
10. 在 Hotspot JVM 中，直接将本地方栈和虚拟机栈合二为一

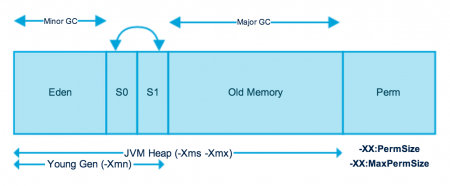
## 堆内存

### 内存划分

对于大多数应用，Java 堆是 Java 虚拟机管理的内存中最大的一块，被所有线程共享。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以及数据都在这里分配内存。

为了进行高效的垃圾回收，虚拟机把堆内存逻辑上划分成三块区域（分代的唯一理由就是优化 GC 性能）：

* **新生带**（年轻代）：新对象和没达到一定年龄的对象都在新生代
* **老年代**（养老区）：被长时间使用的对象，老年代的内存空间应该要比年轻代更大
* **元空间**（JDK1.8 之前叫永久代）：像一些方法中的操作临时对象等，JDK1.8 之前是占用 JVM 内存，JDK1.8 之后直接使用物理内存



Java 虚拟机规范规定，Java 堆可以是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，像磁盘空间一样。实现时，既可以是固定大小，也可以是可扩展的，主流虚拟机都是可扩展的（通过 -Xmx 和 -Xms 控制），如果堆中没有完成实例分配，并且堆无法再扩展时，就会抛出 OutOfMemoryError 异常。

#### 年轻代 (Young Generation)

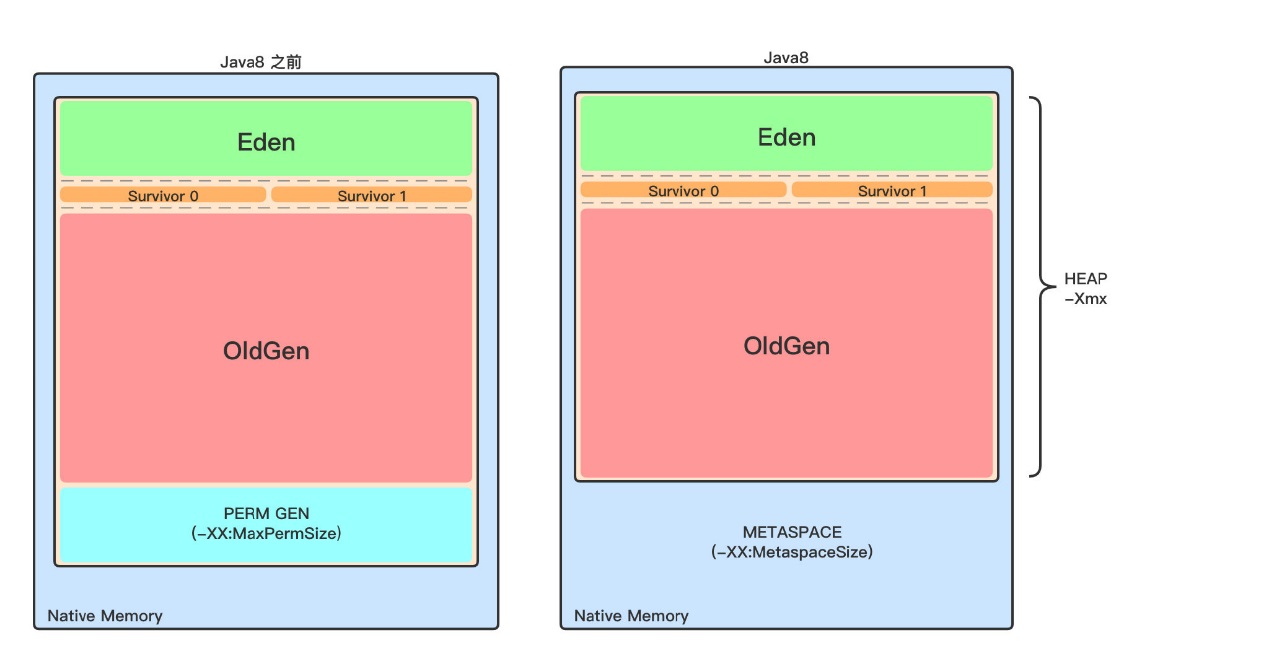
年轻代是所有新对象创建的地方。当填充年轻代时，执行垃圾收集。这种垃圾收集称为 Minor GC。年轻一代被分为三个部分——伊甸园（Eden Memory）和两个幸存区（Survivor Memory，被称为from/to或s0/s1），默认比例是8:1:1

* 大多数新创建的对象都位于 Eden 内存空间中
* 当 Eden 空间被对象填充时，执行Minor GC，并将所有幸存者对象移动到一个幸存者空间中
* Minor GC 检查幸存者对象，并将它们移动到另一个幸存者空间。所以每次，一个幸存者空间总是空的
* 经过多次 GC 循环后存活下来的对象被移动到老年代。通常，这是通过设置年轻一代对象的年龄阈值来实现的，然后他们才有资格提升到老一代

#### 老年代(Old Generation)

旧的一代内存包含那些经过许多轮小型 GC 后仍然存活的对象。通常，垃圾收集是在老年代内存满时执行的。老年代垃圾收集称为 主GC（Major GC），通常需要更长的时间。

大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个Survivor 区之间发生大量的内存拷贝



#### 元空间

不管是 JDK8 之前的永久代，还是 JDK8 及以后的元空间，都可以看作是 Java 虚拟机规范中方法区的实现。

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 Non-Heap（非堆），目的应该是与 Java 堆区分开。

### 设置堆内存大小和 OOM

Java 堆用于存储 Java 对象实例，那么堆的大小在 JVM 启动的时候就确定了，我们可以通过 -Xmx 和 -Xms 来设定

-Xmx 用来表示堆的起始内存，等价于 -XX:InitialHeapSize

-Xms 用来表示堆的最大内存，等价于 -XX:MaxHeapSize

如果堆的内存大小超过 -Xms 设定的最大内存， 就会抛出 OutOfMemoryError 异常。

我们通常会将 -Xmx 和 -Xms 两个参数配置为相同的值，其目的是为了能够在垃圾回收机制清理完堆区后不再需要重新分隔计算堆的大小，从而提高性能

默认情况下，初始堆内存大小为：电脑内存大小/64

默认情况下，最大堆内存大小为：电脑内存大小/4

查看 JVM 堆内存分配

在默认不配置 JVM 堆内存大小的情况下，JVM 根据默认值来配置当前内存大小

默认情况下新生代和老年代的比例是 1:2，可以通过 –XX:NewRatio 来配置

新生代中的 Eden:From Survivor:To Survivor 的比例是 8:1:1，可以通过 -XX:SurvivorRatio 来配置

若在 JDK 7 中开启了 -XX:+UseAdaptiveSizePolicy，JVM 会动态调整 JVM 堆中各个区域的大小以及进入老年代的年龄

此时 –XX:NewRatio 和 -XX:SurvivorRatio 将会失效，而 JDK 8 是默认开启-XX:+UseAdaptiveSizePolicy

在 JDK 8中，不要随意关闭-XX:+UseAdaptiveSizePolicy，除非对堆内存的划分有明确的规划

每次 GC 后都会重新计算 Eden、From Survivor、To Survivor 的大小

### 对象在堆中的生命周期

1. 在 JVM 内存模型的堆中，堆被划分为新生代和老年代

新生代又被进一步划分为 Eden区 和 Survivor区，Survivor 区由 From Survivor 和 To Survivor 组成

1. 当创建一个对象时，对象会被优先分配到新生代的 Eden 区

此时 JVM 会给对象定义一个对象年轻计数器（-XX:MaxTenuringThreshold）

1. 当 Eden 空间不足时，JVM 将执行新生代的垃圾回收（Minor GC）

JVM 会把存活的对象转移到 Survivor 中，并且对象年龄 +1

对象在 Survivor 中同样也会经历 Minor GC，每经历一次 Minor GC，对象年龄都会+1

1. 如果分配的对象超过了-XX:PetenureSizeThreshold，对象会直接被分配到老年代

### 对象的分配过程

为对象分配内存是一件非常严谨和复杂的任务，JVM 的设计者们不仅需要考虑内存如何分配、在哪里分配等问题，并且由于内存分配算法和内存回收算法密切相关，所以还需要考虑 GC 执行完内存回收后是否会在内存空间中产生内存碎片。

* new 的对象先放在伊甸园区，此区有大小限制
* 当伊甸园的空间填满时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收（Minor GC），将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
* 然后将伊甸园中的剩余对象移动到幸存者 0 区
* 如果再次触发垃圾回收，此时上次幸存下来的放到幸存者 0 区，如果没有回收，就会放到幸存者 1 区
* 如果再次经历垃圾回收，此时会重新放回幸存者 0 区，接着再去幸存者 1 区
* 什么时候才会去养老区呢？ 默认是 15 次回收标记
* 在养老区，相对悠闲。当养老区内存不足时，再次触发 Major GC，进行养老区的内存清理
* 若养老区执行了 Major GC 之后发现依然无法进行对象的保存，就会产生 OOM 异常

### GC 垃圾回收简介

Minor GC、Major GC、Full GC

JVM 在进行 GC 时，并非每次都对堆内存（新生代、老年代；方法区）区域一起回收的，大部分时候回收的都是指新生代。

针对 HotSpot VM 的实现，它里面的 GC 按照回收区域又分为两大类：部分收集（Partial GC），整堆收集（Full GC）

部分收集：不是完整收集整个 Java 堆的垃圾收集。其中又分为：

新生代收集（Minor GC/Young GC）：只是新生代的垃圾收集

老年代收集（Major GC/Old GC）：只是老年代的垃圾收集

目前，只有 CMS GC 会有单独收集老年代的行为

很多时候 Major GC 会和 Full GC 混合使用，需要具体分辨是老年代回收还是整堆回收

混合收集（Mixed GC）：收集整个新生代以及部分老年代的垃圾收集

目前只有 G1 GC 会有这种行为

整堆收集（Full GC）：收集整个 Java 堆和方法区的垃圾

### TLAB

Thread Local Allocation Buffer 的简写，基于 CAS 的独享线程（Mutator Threads）可以优先将对象分配在 Eden 中的一块内存，因为是 Java 线程独享的内存区没有锁竞争，所以分配速度更快，每个 TLAB 都是一个线程独享的。

#什么是 TLAB （Thread Local Allocation Buffer）?

从内存模型而不是垃圾回收的角度，对 Eden 区域继续进行划分，JVM 为每个线程分配了一个私有缓存区域，它包含在 Eden 空间内

多线程同时分配内存时，使用 TLAB 可以避免一系列的非线程安全问题，同时还能提升内存分配的吞吐量，因此我们可以将这种内存分配方式称为快速分配策略

OpenJDK 衍生出来的 JVM 大都提供了 TLAB 设计

#为什么要有 TLAB ?

堆区是线程共享的，任何线程都可以访问到堆区中的共享数据

由于对象实例的创建在 JVM 中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的

为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度

尽管不是所有的对象实例都能够在 TLAB 中成功分配内存，但 JVM 确实是将 TLAB 作为内存分配的首选。

在程序中，可以通过 -XX:UseTLAB 设置是否开启 TLAB 空间。

默认情况下，TLAB 空间的内存非常小，仅占有整个 Eden 空间的 1%，我们可以通过 -XX:TLABWasteTargetPercent 设置 TLAB 空间所占用 Eden 空间的百分比大小。

一旦对象在 TLAB 空间分配内存失败时，JVM 就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在 Eden 空间中分配内存。