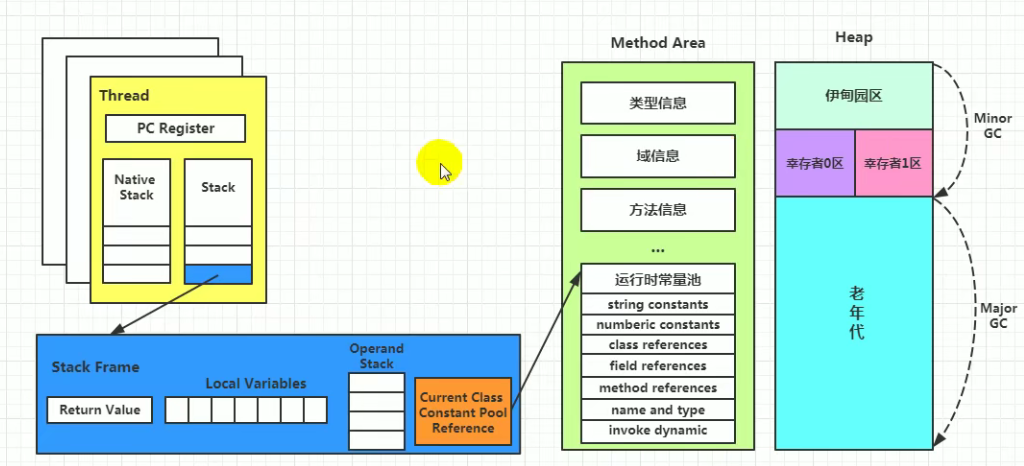
**运行时数据区**



**概述**

Java文档：[java (oracle.com)](https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE)

Running Data Area

[Chapter 2. The Structure of the Java Virtual Machine (oracle.com)](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.1)

不同的JVM对于内存的划分方式和管理机制存在着部分差异

每个线程：程序计数器、栈、本地方法栈

线程间共享：堆空间、方法区

每个JVM只有一个Runtime实例，即运行时环境。

**线程**

线程是一个程序的运行单元

JVM允许一个应用有多个线程并行执行

Hotspot JVM，每个线程与操作系统的本地线程直接映射

操作系统负责所有线程的安排调度到任何一个可用CPU上，一旦本地线程初始化成功，

它就会调用Java线程的run()方法

**守护线程、普通线程**

后台线程：不包括main线程，以及所有这个main线程自己创建的线程

后台线程主要有：虚拟机线程、周期任务线程、GC线程、编译线程、信号调度线程

**一、程序计数器（PC Rejister）**

并行：多个线程在同一时间执行

并发：指两个或多个事件在同一时间间隔发生。

**概述**

JVM中的PC寄存器是对物理PC寄存器的一种抽象模拟

PC计数器生命周期与线程的生命周期保持一致

它是唯一一个再JVM规范中没有规定任何OOM的区域，也没有GC

任何时间一个线程都只有一个方法在执行，也是所谓的当前方法

程序计数器会存储当前线程正在执行的Java方法的JVM指令地址

如果执行native方法，则是未指定值（undefined）

**作用**

用来存储指向下一条指令的地址，由执行引擎读取下一条指令

问题：

**使用PC寄存器存储字节码指令地址有什么用？**

因为CPU会不断切换线程，当切换回来后，需要知道接着从哪开始继续执行

JVM的字节码解释器就需要通过改变PC寄存器的值来明确下一条应该执行什么样

的字节指令

**PC寄存器为什么被设为线程私有？**

因为CPU会不断切换任务，为了能准确地记录各个线程当前的字节码指令地址，

最好的办法就是给每个线程分配一个PC寄存器

**二、虚拟机栈（Java Virtual Machine Stack）**

Java指令都是根据栈来设计的，不同平台CPU架构不同，所以不能基于寄存器设计。

栈是运行时单位，堆是存储的单位

**概述**

Java虚拟机栈，每个线程在创建时都会创建一个虚拟机栈，内部保存一个个栈帧(Stack Frame)，对应一次次的方法调用。（栈顶的方法表示当前方法）

**对线程是私有的**

生命周期：生命周期和线程一致

**作用**

主管Java程序的运行，保存方法的局部变量（8种基本数据类型，对象的引用地址）、部分结果

、并参与方法的调用和返回

**特点**

对栈来说不存在垃圾回收的问题

**栈可能出现的异常**

Java虚拟机规范允许**Java栈的大小是动态的或者固定不变的**

如果采用固定大小的Java虚拟栈，那每一个线程的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虚拟机所允许的最大容量，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，或者在创建新线程的时候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。

**设置栈内大小**

使用参数-Xss，设置线程的最大栈空间，栈的大小决定函数调用的最大可达深度

[java (oracle.com)](https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE)

**栈运行原理**

每个线程都有自己的栈，栈中的数据都是以栈帧的格式存在。

栈帧是一个内存区块，一个数据集

在一条活动线程上，一个时间点上，只会有一个活动的栈帧（当前栈帧，Current Frame）

与当前栈帧对应的是当前方法（Current Method），定义这个方法的类就是当前类（Current Class）

不同线程中所包含的栈帧之间是不能相互引用的

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，之后，虚拟机会丢弃当前栈帧，使得前一个栈帧变为当前栈帧。

Java方法有两种返回函数的方式：一种是正常的函数返回，return；另一种是抛出异常（没有try..catch）。都会导致栈帧被弹出。

**栈帧的内部结构**

局部变量表（Local Variables）

操作数栈（Opeand Stack）（或表达式栈）

动态链接（Dynamic Linking）（或指向运行时常量池的方法引用）

方法返回地址（Return Address）（或方法正常退出或异常退出的定义）

一些附加信息

**1. 局部变量表**

也成为局部变量数组或本地变量表

定义为一个数字数组，主要用于存储方法参数和定义在方法体内的局部变量

（包括基本类型、对象引用、returnAddress类型）

由于局部变量表建立在线程的栈上，不存在数据安全问题

局部变量表所需要的容量的大小是在编译期确定下来的，在方法运行期间不会改变大小

局部变量表中的变量只在当前方法调用中有效，当方法调用结束后，随着方法栈帧的销毁，局部变量表也随之销毁

**Slot**

局部变量最基本的存储单位时Slot（变量槽）

32位以内的类型只占用一个slot（包括returnAddress类型），64位的类型（long、double）占两个槽

局部变量会按照声明的顺序复制到局部变量表中的每一个slot上

对于访问64位的局部变量值时，只需要使用前一个索引（因为有2个slot）

如果当前帧是由构造方法或者实例方法创建的，那么该对象的引用this会存放在index位为0的slot处，其余参数按参数列表顺序排列（这就解释了static方法不能有this引用）

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声明的局部变量就可能会重复使用已国企的局部变量槽位，节省资源

**静态变量与局部变量的对比**

变量分类：按数据类型分：a：基本数据类型；b：引用数据类型

按在类中声明的位置分：

a：成员变量：在使用前都经历过默认初始化赋值

类变量：linking的prepare阶段，给类变量默认赋值——>在initial阶段：给类变显式赋值即静态代码块赋值

实例变量：随着对象的创建，会在对空间中分配实例变量空间，并进行默认赋值

b：局部变量：在使用前，必须进行显式赋值的，否则编译不通过

在栈帧中，与性能调优关系最为密切的部分就是局部变量表，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，只有被局部变量表中直接或间接引用对象都不会被回收

**2. 操作数栈 (Operand Stack)**

栈：可以使用数组、链表来实现，(由于数组创建后是固定长度) 操作数栈也有一个明确的栈深用于存储数值，所需的最大栈深在编译期就定义好了，保存在方法的Code属性中，max\_stack的值

后进先出，也称为表达式栈（Expression Stack）

注意操作数栈并非采用访问索引的方式进行数据访问，而是只能通过push和pop

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

32bit类型占用一个栈深单位

64bit类型占用两个栈深单位

操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈和出栈

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

Java虚拟机的解释引擎就是基于栈的执行引擎，这里的栈就是操作数栈

**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

**3. 动态链接(Dynamic Linking)--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**运行时常量池**中该栈帧**所属方法**的引用

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会作为符号引用 保存在class文件的常量池里

**为什么需要常量池？**

提供一些符号和常量，便于指令的识别

**方法的调用**

符号引用-----》直接引用

**静态链接**

当一个字节码文件被装载进JVM中，如果被调用的目标方法在编译期间可知，切在运行期间保持不变，这种情况下的调用方法的符号引用转换为直接引用的过程，称为静态链接

**动态链接**

如果被调用的方法在编译期间无法被确定下来的，称为动态链接

**对应的方法的绑定机制：**

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 ，只发生一次

**早期绑定：/非虚方法**

被调用的目标方法在编译期可知、且运行期间保持不变

**晚期绑定：/虚方法**

如果被调用的方法在编译期间无法被确定下来，只能在程序运行期根据实际类型绑定的方法

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

子类对象多态性的使用前提：

1. 类的继承关系

2. 方法的重写

**方法调用指令：**

普通调用指令：

（非虚方法）

1. invokestatic：调用静态方法，解析决断确定唯一方法版本

2. invokespecial：调用<init>方法、私有及父类方法，解析阶段确定微一方法版本

3. invokevirtual：调用所有虚方法（注意final修饰的如果没有显示的写super. 会显示invokevirtual，但它还是非虚方法，因为final不能重写）

4. invokeinterface：调用用接口方法

1、2指令调用的方法称为非虚方法，其余的（final修饰的除外）称为虚方法

动态调用指令：

5. invokedynamic：动态解析出所需要调用的方法，然后执行

Java是静态类型语言

动态类型语言：判断变量值的类型信息（对类型的检查在运行期）

静态类型语言：判断变量自身的类型信息（对类型的检查在编译期）

**虚方法表**

**为什么要有虚方法表？**

由于虚方法需要频繁的动态分派，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

每个类都有一个虚方法表，表中存放着各个方法的实际入口（没有非虚方法）

**虚方法什么时候被创建？**

虚方法表会在类加载的链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

**4. 方法返回地址（return adress）**

存放调用该方法的PC寄存器值

方法正常退出，调用者的PC计数器的值作为返回地址，即调用该方法的下一条指令的地址，

异常退出时，返回地址要通过异常表来确定，栈帧中一般不会保存这部分信息。

**正常完成出口和异常完成出口的区别：**

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。

**5. 一些附加信息**

栈帧允许携带与Java虚拟机实现相关的一些附加信息

**虚拟机栈--面试相关**

举例栈溢出的情况（StackOverflowError）

通过-Xss设置栈的大小：OOM

通过调整栈大小，就能保证不出现溢出吗？

不能

垃圾回收是否会涉及到虚拟机栈？

不会

补充：

|  |  |  |
| --- | --- | --- |
|  | ERROR | GC |
| 程序计数器 | × | × |
| 虚拟机栈 | √ | × |
| 本地方法栈 | √ | × |
| 方法区 | √ | √ |
| 堆 | √ | √ |

分配的栈内存越大越好吗？

不会，会挤占其它线程的空间

方法中定义的局部变量是否线程安全？

具体问题具体分析

何为线程安全？如果只有一个线程才可以操作此数据，则必是线程安全的

如果有多个线程操作此数据，则此数据是共享数据。如果不考虑同步机制，则会

存在线程安全

**三、堆（Heap）**

**概述**

一个JVM实例只有一个堆内存，堆也是Java内存管理的核心区

Java的堆在JVM启动时就被创建，其空间大小也就确定了

堆内存的大小可以调节

堆可以处在物理上不连续、但逻辑上应该被视为连续的

所有线程共享Java堆，可以划分为线程私有的缓冲区（TLAB）

在方法结束后，堆中的对象不会被马上移除，仅仅在垃圾收集的时候才会被移除

堆时GC执行垃圾回收的重点区域

现代垃圾收集器大部分都是基于分代垃圾收集理论设计

JDK 8之后 堆内存逻辑上分为三部分：**新生区、养老区、元空间（永久代，JDK 7 之前）**

**1. 设置对空间的大小参数**

“-Xms”：表示堆区（新生代+老年代）的起始内存，等价于-XX:InitialHeapSize

“-Xmx”：表示堆区（新生代+老年代）的最大内存，等价于-XX:MaxHeapSize

一旦堆区中的内存大小超过“-Xmx”所指定的最大内存时，

将会抛出OutOfMemoryError异常

**2. 默认堆空间的大小**

初始内存大小：物理电脑内存 / 64

最大内存大小：物理电脑内存 / 4

**3. 手动设置：-Xms600m -Xmx600m**

开发中建议：将初始堆内存和最大的内存设置成相同的值

**4. 查看设置参数**

方式一：jps / jstat -gc 进程id

方式二：-XX:+PrintGCDetails

在JVM中的Java对象可分为两类：

一类是生命周期较短的对象

一类是生命周期很长的对象

**Java堆区划分**

手动配置新生代和老年代在堆结构中的占比：（一般不会修改）

默认-XX:NewRatio=2，表示新生代占1，老年代占2

可修改-XX:NewRatio=4，新生代占1，老年代占4

在HotSpot，新生代中 Eden：from：to 占比为8:1:1

查看可在jvisualvm或者cmd中的jinfo -flag NewRatio 进程id

-XX:SurvivorRatio=？：设置新生代中Eden区与Survivor区的比例（默认值是8）

-XX:-UseAdaptiveSizePolicy：关闭自适应内存分配策略（暂时用不到）

-Xmn:设置新生代的空间大小（在与-XX:NewRatio=?冲突时，优先按-Xmn的设置）

[java (oracle.com)](https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE)

**堆的划分**

**1. 年轻代**

**Eden区**

**Survivor0区（from区）**

**Survivor1区（to区）**

**2. 老年代：存放新生代中经历多次GC仍然存活的对象**

**对象分配过程**

1. new的对象先放在Eden区，此区有大小限制

2. 当Eden区填满时，程序又需要创建对象，JVM的垃圾回收器将对Eden区进行垃圾回收（Minor GC），将Eden区不再被其它对象所引用的对象进行销毁，再加载新的对象放到Eden区。

3. 然后将Eden区中的剩余对象移动到幸存者0区。

4. 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区的，如果没有回收，就会放到幸存者1区

5. 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

6. 什么时候去养老区？可以设置次数，默认15次

可以通过设置参数改变：-XX:MaxTenuringThreshold=<N>进行设置

7. 在养老区，相对悠闲，养老区内存不足时，再次触发GC，进行养老区的内存清理

8. 如果养老区执行了GC后，仍无法对对象保存，会产生OOM异常

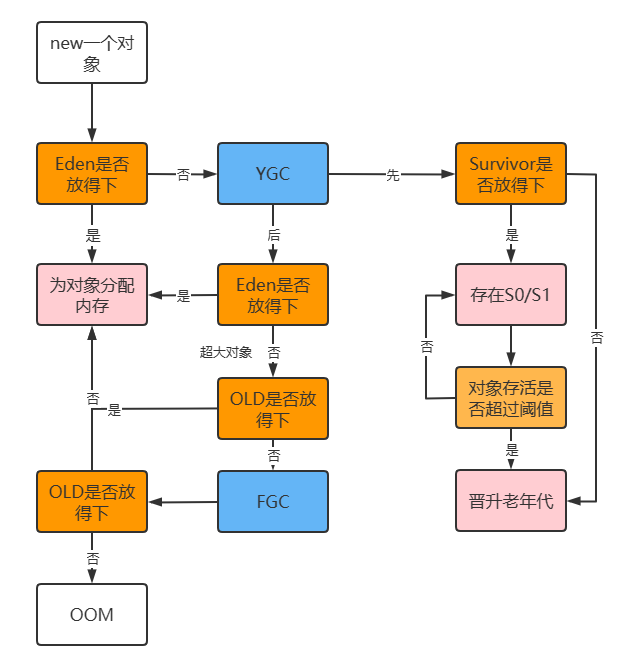
总结：

幸存者某区满的时候不会触发YGC/Minor GC

在这种情况下，只有当Eden区也满的时候，会带着Eden区和幸存者某区一起进行YGC，如果幸存者区这是还是满的，则会将幸存者某区的对象存放致养老区。

针对幸存者s0，s1区的总结，复制之后有交换，谁空谁是to

关于垃圾回收，频繁在新生区收集，很少在养老区回收，几乎不在永久区



**Minor GC、Major GC、Full GC的区别与使用**

Jvm在进行GC是，并非每次都对上面三个内存区域（新生代、老年代、方法区）

一起回收的，大部分时候回收的都是指的是新生代

针对HotSpot VM 的实现，它里里面的GC按照回收区域分为两大类型：

部分收集（Partial GC）、整堆收集（Full GC）

**部分收集（Partial GC）**

不是完整收集整个Java堆的垃圾收集，其中又分为：

1. 新生代收集（Minor GC / Young GC）：只是新生代的垃圾收集

2. 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 会有单独的老年代行为

注意，很多时候Majot GC 和 Full GC 混淆使用，

需要具体分辨是老年代回收还是整堆回收

3. 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堆和方法区的垃圾收集

**新生代GC（Minor GC）触发机制**

当新生代空间不足，就会触发Minor GC

每次Minor GC 会清理新生代的内存

注意（Survivor满不会触发GC，只有Eden代满才会）

Minor GC 会引发 STW，会暂停其它用户线程，垃圾回收结束时，用户线程才会恢复运行

**老年代GC（Major/Full GC）触发机制**

在老年代空间不足时，会先尝试触发Minor GC，如果空间还是不足，就会触发Major GC

当对象从老年代消失时，我们说Major GC 或者 Full GC发生了

出现了Major GC 至少会伴随一次Minor GC （但非绝对，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC 的策略选择过程）

Major GC的速度一般会比Minor GC慢10倍以上，STW时间更长

Major GC后，内存还是不足，就会OOM

**Full GC 触发机制：**

1. 调用System.gc()

2. 老年代空间不足

3. 方法区空间不足

4. 通过Minor GC进入老年代的平均大小大于老年代的可用内存

5. 由Eden区，Survivor0、向Survivor1 复制时，由于太大，需要放到老年代，

但老年代的可用内存小于该对象的大小

Full GC 时开发或者调优过程中尽量要避免的

**Java堆为什么需要分代？不分代就不能正常工作了吗？**

不分代完全可以，分代的唯一理由就是优化GC性能

**内存分配策略**

1. 优先分配到Eden

2. 大对象直接分配到老年代

3. 长期存活的对象分配到老年代

4. 动态对象年龄判断

如果幸存者区中相同年龄的所有对象大小的总和大于幸存者空间的一般，年龄大于或等于该年龄的对象可以直接进入老年代

5. 空间分配担保

-XX:HandlePromotionFailure

**对象分配：TLAB（Thread Local Allocation Buffer）**

堆区是被线程共享的，为了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

对Eden区继续进行划分， JVM给每个线程分配了一个私有缓存区域，包含在Eden区中

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，还能够提高内存分配的吞吐量，我们将这种分配内存的方式称为快速分配策略。

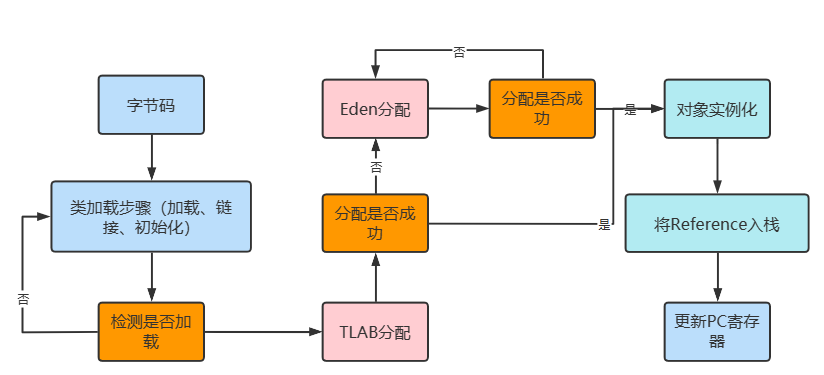
1. JVM将TLAB作为内存分配的首选

2. TLAB空间分配失败时，JVM会通过使用加锁机制确保操作数据的原子性，从而在Eden中分配内存

3. TLAB占比：Eden空间的1%

4. 配置：-XX:+UseTLAB(默认是开启的)

jinfo -flag UseTLAB 进程id //查看是否开启



**空间分配担保**

发生Minor GC之前，虚拟机会**①**检查 老年代最大可用的连续空间 是否 大于 新生代所有对象的总空间

如果大于：此次Minor GC 是安全的

如果小于：则虚拟机会查看-XX:HandlePromotionFailure设置值是否允许担保失败

如果为ture，会**②**检查 老年代最大可用连续空间是否大于历次晋升到老年代的对象的平均大小

如果大于，会进程Minor GC，但仍有风险

如果小于，会进行一次Full GC

如果为false，进行一次Full GC

JDK 6 Update 24 之后，HandlePromotionFailure不会影响担保策略了，**①**或者**②为true** 则会进行Minor GC，否则进行Full GC

**堆是分配对象的唯一选择吗**

不是

经过逃逸分析后（Escape Analysis）发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配

Java中的对象不一定是在堆上分配的，因为JVM通过逃逸分析，能够分析出一个新对象的使用范围，并以此确定是否要将这个对象分配到堆上。

’但，目前Oracle HotSpot JVM（只实现了逃逸分析、标量替换，栈上分配还没有实现）并没有这么做，还是可以明确所有的对象实例都是创建在堆上

JDK目前没有应用逃逸分析

**四、逃逸分析（Escape Analysis）**

**逃逸分析的基本行为就是分析对象动态作用域**

当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发送逃逸

当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。（如作为调用参数传递到其他地方）

没有发送逃逸的对象，则可以分配到栈上，随着方法的执行结束，栈空间被移除。

**如何快速判断是否发生逃逸？**

看new 的对象实体 是否有可能在方法外被调用

参数设置：在JDK6u23之后，HotSpot默认开启了逃逸分析，

-XX:+DoEscapeAnalysis，显示开启逃逸分析

-XX:+PrintEscapeAnalysis，查看逃逸分析的筛选结果

开发中能使用局部变量的，就不要再方法外定义

**代码优化**

**1. 栈上分配**

如果对象没有发生逃逸，会栈上分配，随着方法的执行结束，栈空间被移除。

发生逃逸：变量赋值、方法返回、实例引用传递

**2. 同步省略**

如果一个对象只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步

线程同步的代价很高，会降低并发性和性能

动态编译同步块的时候，JIT编译器借助逃逸分析，判断同步块对象使用的锁对象是否只能被一个线程访问而没有被发布到其他线程。

如果没有，那么JIT编译器在编译这个同步块的时候就会取消对这部分代码的同步。

这个取消同步的过程，就叫同步省略，也叫锁清除

**3. 分离对象或标量替换**

有的对象可能不需要作为一个连续的内存结构也可以被访问到，那么对象的部分可以不存储在内存，

而是存储在CPU寄存器中

标量（Scalar）是指一个无法再分解成更小的数据的数据，Java中的原始数据类型就是标量

可以分解的数据叫做聚合量（Aggregate），Java中的对象就是聚合量

在JIT阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，会把这个对象拆解成若干个其中包含的若干个成员变量来代替，这个过程就是标量替换。

标量替换参数设置：

-XX:+EliminateAllocations：默认开启，允许将对象打散分配在栈上

**本章小结**

1. 年轻代是对象的诞生、成长、消亡的区域，一个对象在这里产生、应用，最后被垃圾回收器收集、结束生命。

2. 老年代放置长生命周期的对象，通常都是从Survivor区域筛选拷贝过来的Java对象。当然也有特殊情况，普通的对象会被分配在TLAB上，如果对象较大，JVM会试图将其分配在Eden其他位置上；如果对象太大，无法再新生代中找到足够长的连续空闲空间，JVM就会直接分配老年代。

3. 当GC只发生在年轻代中，回收年轻代对象的行为被称为Minor GC，当GC发生在老年代时则被称为Major GC 或者 Full GC。一般，Minor GC 的发生频率比Major GC高很多。

**小结堆空间的参数设置**

<https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE>

-XX:+PrintFlagsInitial：查看所有的参数的默认初始值

-XX:+PrintFlagsFinal：查看所有的参数的最终值（可能会有修改后的）

具体查看某个参数的指令： jps ---> jinfo -flag SurvivorRatio 进程id

-Xms：初始堆空间内存（默认物理内存的1/64）

-Xmx：最大堆空间内存（默认物理内存的1/4）

-Xmn：设置新生代的大小（初始值及最大值）

-XX:NewRatio：配置新生代与老年代在堆结构中的占比

-XX:SurvivorRatio：设置新生代Eden和S0/S1空间的比例

-XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄

-XX:+PrintGCDetails：输出详细的GC处理日志

打印GC简要信息，① -XX:+PrintGC ② -verbose:gc

-XX:HandlePromotionFailure：是否设置空间分配担保

-XX:+DoEscapeAnalysis，显示开启逃逸分析（JDK6u23之后默认都开启）

-XX:+PrintEscapeAnalysis，查看逃逸分析的筛选结果

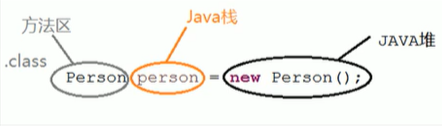
-XX:+EliminateAllocations：默认开启，允许将对象打散分配在栈上

**五、方法区（Method Area）**

**概述**

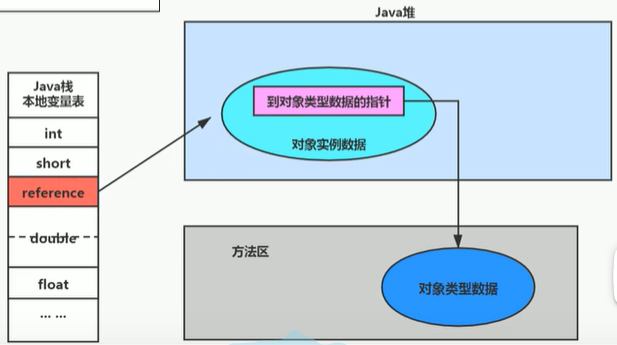
[The Java® Virtual Machine Specification (oracle.com)](https://docs.oracle.com/javase/specs/jvms/se8/html/)

方法区存储对象类型。



尽管所有方法区逻辑上时属于堆的一部分，但对HotSpotJVM，在具体的实现过程中，方法区看作式一块独立于Java堆的内存空间

方法区 与 堆 一样，



被各个线程共享

方法区在JVM启动时被创建，物理内存中和Java堆一样 不连续

方法区的大小可固定、可扩展

方法区的大小决定了系统可以保存多少类，如果定义太多的类，会导致方法区溢出，JVM抛出java.lang.OutOfMemory:Metaspace

关闭JVM就会释放这个区域的内存

元空间与永久代最大的区别：元空间不在虚拟机设置的内存中，而是使用本地内存

元空间与永久代并不只是名字改了，内部结构也调整了

**设置方法区内存大小**

JDK7之前：

-XX:PermSize=100M

-XX:MaxPermSize=100M

JDK8之后：

-XX:MetaspaceSize=100M，设置元空间初始大小，在64位系统下,默认是21M

-XX:MaxMetaspaceSize=100M，设置元空间最大大小，默认-1，即没有限制

MetaspaceSize 是初始的高水位线，一旦超过这个水位线，Full GC 将会触发并卸载没用的类（即这些类对应的类加载器不再存活），然后这个高水位线将会重置，新的高水位线的值取决于GC释放了多少元空间，如果释放的空间少，那么在不超过MaxMetaspaceSize时，适当提高该值，如果释放的空间过多，则适当降低该值。

如果初始化的水位线设置过低，将会频繁地发生GC，建议将-XX:MetaspaceSize设置为较高的值。

**如何解决OOM？**



**方法区的内部结构**

类型信息、常量、静态变量、即时编译器编译后的代码缓存、域信息、方法信息

**1. 类型信息**

这个类型的完整有效名称（全名=包名.类名）

这个类型直接父类的完整有效名（对于interface或java.lang.Object都没有父类）

这个类型的修饰符（public，abstract，final的某个子集）

这个类型直接接口的一个有序列表

**2. 域（Field）信息**

JVM必须在方方法区中保存类型的所有域的相关信息和声明顺序

域的相关信息包括：域名称、域类型、域修饰符（public，private、protected、static、final、volatile、transient的某个子集）

**3. 方法（Method）信息**

方法名称、方法的返回类型（或void）、方法参数的数量和类型（按顺序）、方法的修饰符、方法的字节码、操作数栈、局部变量表及大小（抽象和本地方法除外）、异常表（抽象和本地方法除外）

**4. 静态变量**

**non-final的类变量**

随类的加载而加载，即使某类的实例赋值为null，也能访问类变量

**static final 全局常量**

被声明为final的类变量的处理方法 则不同，每个全局常量在编译的时候就会被分配了

**5. 运行时常量池**

字节码文件中有一个常量池

这个常量池在被加载到内存当中就变成了运行时常量池

方法区内部包含了一个运行时常量池

**（1）class文件中的常量池（Constant Pool Table）**

常量池表包含了各种字面量和对类型、域和方法的符号引用，用存放编译期生成的各种字面量与符号引用，这部分内容在类加载后存放到方法区的运行时常量池中。

补充：字面量

int i = 1;把整数1赋值给int型变量i，整数1就是Java字面量

同样，String s = "abc";中的abc也是字面量。

**为什么需要常量池？**

为了便于链接，jvm 使用常量池来保存跟踪当前类中引用的其他类及其成员变量和成员方法。

**常量池具体包括**：

1. 数量值

2. 字符串值

3. 类引用

4. 字段引用

5. 方法引用

常量池，可以看做是一张表，虚拟机指令根据这张常量表找到要执行的类、方法名、参数类型、字面量等类型。

**（2）运行时常量池**

JVM为每个已加载的类型（类或接口）都会维护一个常量池，其中的数据通过索引访问。

运行时常量池中包含在编译期就已经确定的数值字面量，以及运行期解析后的方法或者字段引用，此时已经不是符号地址了，这里换为真实地址。

运行时常量池，具备动态性，也就是说运行时常量池存储的信息可能会动态增加（编译期常量池没有的信息）

如果构造运行时常量池所需的内存空间超过了方法区所能提供的最大值，JVM会抛出OOM异常

**方法区的演进细节**

只有HotSpot才有永久代（JDK1.8之前）

JDK 1.6 ：有永久代，静态变量存放在永久代上

JDK 1.7 ：有永久代，但已逐步“去永久代”，字符串常量池、静态变量移除，保存在堆中

JDK 1.8 ：无永久代，类型信息、字段、方法、常量保存在本地内存的元空间，但字符串常量池、静态变量仍在堆

**永久代为什么要替换元空间？**

1. 为永久代设置空间大小是很难确定的

2. 对永久代调优很困难：判断类或常量花时间，即Full GC花时间，交由本地内存是更好的选择

**StringTable为什么要调整到堆中？**

因为永久代的回收效率低，在Full GC 的时候才会触发，而Full GC 是老年代空间不足、永久代不足的时候才会触发，这就会导致StringTable的回收效率不高，而开发中还会有大量的字符串被创建，回收效率低，又导致了永久代空间不足。

**静态变量的存放**

class Test {
static ObjectHolder staticObj = new ObjectHolder();
ObjectHolder staticObj = new ObjectHolder();
void foo() {
ObjectHolder staticObj = new ObjectHolder();
System.out.println("done");
}
}

三个New的对象，都存放在Java堆中

staticObj随着Test的类型信息存放在方法区，instanceObj随着Test的对象实例存放在Java堆，localObject则是存放在foo() 方法栈帧的局部变量表中。

**方法区的垃圾收集**

主要回收两部分的内容，

常量池中废弃的常量 和 不再使用的类型

**常量的回收策略：**

方法区中的常量池中的常量：

1. 字面量：比较接近Java语言层次的常量概念，如文本字符串、被声明为final的常量值。

2. 符号量：

1. 类和接口的全限定名

2. 字段的名称和描述符

3. 方法的名称和描述符

常量池中的的回收策略：常量没有被任何地方引用，就被回收

**类型的回收策略：**

同时满足下面的三个条件：

1. 该类所有的实例都已被回收，也就是不存在任何改类以及其派生类的实例

2. 加载该类的类加载器已被回收，这个条件很难达成

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改类的方法

在大量使用反射、动态代理、CGLib等字节码框架，动态生成JSP以及OSGi这类频繁自定义类加载器的场景中，通常都需要Java虚拟机具备类型卸载能力，保证不会对方法区造成过大的内存压力。

**六、本地方法接口&本地方法栈（Native Method Stack）**

**概述**

一个Native Method就是一个Java调用非Java代码的接口

标识符native可以与所有其它的Java标识符连用，但abstract除外

**为什么要使用Native Method**

**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，

**2. 与操作系统的交互**

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的一部分就是用C写的

**3. Sun's Java**

Sun的解释器使用C实现的，者使得它能像一些普通的C一样与外部交互

**本地方法栈（Native Method Stack）**

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

本地方法栈，也是线程私有的

允许被实现成固定或者可扩展的内存大小

StackOverflowError和OOM与虚拟机栈一样

本地方法是用C语言实现的

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在受虚拟机限制的世界，

它和虚拟机拥有同样的权限

本地方法可以通过本地接口来访问虚拟机内部的运行时数据区

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