**java虚拟机**



# 内存模型

## 内存区域

### 程序计数器

概念：当前线程的执行字节码行号指示器。每条线程都需要一个独立的程序计数器。java虚拟机中没有规定程序计数器的任何OutOfMemoryError情况的区域。

### Java虚拟机栈

每个方法只从的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。该区域定义两种异常：1.如果线程请求的栈深度大于虚拟机允许深度，将抛出StackOverflowError；2.如果虚拟机栈允许动态扩展，如果扩展时无法申请到固定的内存抛出OutOfMemoryError

### 本地方法栈

native method:java调用的不是由java实现的方法被称为本地方法本地方法栈和jvm栈发挥的作用十分相似，区别只是一个为java方法服务，一个为native方法服务。本地方法栈也会抛出StackOverflowError和OutOfMemroyError异常

### Java堆

存放对象实例所有线程共享一块内存区域

### 方法区

各个线程共享的内存区域用于存储已经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无法满足内存分配需求时，将抛出OutOfMemroyError异常

### 运行时常量池

运行时常量池是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。字面量：即是这个量本身，例如 int a = 5 中5即是字面量。符号引用：一组符号来表示引用的目标，可以是任何形式的字面量，符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。直接引用：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。直接引用是和虚拟机的布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经被加载入内存中了。运行期间也可以将常量放入常量池中，比如String的intern()方法。和方法区一样也会抛出OutOfMemoryError异常。

### 直接内存

不属于java虚拟机的内存区域NIO类引入了一种给予通道的缓冲区I/O方式，它可以使用Native函数直接分配堆外聂村，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。各个内存区域内存区域总和大于物理内存限制，可能导致动态扩展时出现OutOfMemoryError异常。

## 对象的内存操作

### 对象创建

虚拟机遇到对象创建指令时，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，检查该类是否被加载、解析和初始化过，如果没有先执行类的加载过程。类加载检查通过后，为新生对象进行内存分配。

* 内存分配方式

• 指针碰撞

对于java堆中内存时绝对规整的，使用过的内存在一边空闲的内存在另一边，中间放一个指针作为分界点指示器，分配内存时只需要把指针移动对象大小相等的距离。

• 空闲列表

对于java堆中内存不是规整的，虚拟机用一个列表记录那块儿内存可用，为对象分配内存时，找到一块足够大的空间画风诶对象，同时更新列表。

* 分配内存并发方案

• CAS同步处理

• 本地线程分配缓存

把内存分配的动作按照线程划分到不通的空间上进行，即每个线程在java堆中预先分配一块内存。虚拟机启用本地线程分配缓存可以通过-XX:+/-UserTLAB

### 对象内存布局

* 对象头

• 运行时数据Mark Wrod

存储哈希码、GC分代年龄、所状态标志、线程持有的锁、偏向线程id、偏向时间戳这部分数据官方称为 Mark Word，其为一个非固定的数据结构以便在绩效的空间内存存储尽量多的信息。

• 类型指针

即对象指向它的类元数据的指针，通过这个指针来确定该对象是那个类的实例。

* 实例数据

程序代码中定义的各种类型的字段内容。

* 对齐填充

### 对象的访问定位

* 使用句柄

java堆中划分一块内存作为句柄池，reference中的地址就是句柄地址，句柄包含了对象的实例数据和类型数据的具体地址。优点：reference中存储的是稳定的句柄地址，对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改

* 直接指针

java对象的布局中考虑如何放置访问类型数据的相关信息，reference中存储的直接就是对象地址。优点：访问速度快，节省了一次指针定位的时间开销。

# 垃圾收集

## 判断对象死活策略

### 引用计数法

给对象添加一个引用计数器，一个地方引用，计数器值加1；引用失效时，计数器值减1；任何计数器为0的对象就是不可能在被使用的。缺点：无法解决对象之间相互循环引用的问题

### 可达性分析法

从一系列GC Roots的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时，这证明此对象不可达。GC Roots的对象包括一下：1.虚拟机栈中引用的对象；2.方法区类静态属性引用的对象；3.方法去中常量引用的对象；4本地方法栈JNI引用的对象。

## 垃圾收集算法

### 标记清除法

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。缺点：1.标记和清除两个过程效率不高；2.产生大量不连续的内存碎片。

### 复制算法

将内存划分为大小相等的两块，每次使用其中的一块，当一块内存用完后，就将存货对象复制到另一块上，然后将使用的内存一次清理。实现简单，运行高效。缺点：内存缩小了一半。

### 标记整理算法

标记过程和标记清除算法相同，后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动然后直接清理掉端边界以外的内存。老年代使用。

### 分代收集算法

把java堆分成新生代和老年代，在新生代中，每次有大批对象死去，少量存活所以采用复制算法。老年代中因为对象存活率高/没有额外担保空间，就必须使用标记清除或者标记整理算法。

## 垃圾收集器

### Serial收集器

![desc](data:None;base64,)

单线程收集器，在垃圾收集的过程中必须暂停到其他所有的工作线程。虚拟机在client模式下的默认新生代收集器

### ParNew收集器

![desc](data:None;base64,)

是Serial的多线程版本，是一款并行收集器，可以用-XX:ParallelGCThreads来控制垃圾收集的线程数。

### Parallel Scavenge 收集器

新生代并行收集器，关注垃圾收集事用户线程的停顿时间，目标是达到一个可控制的吞吐量。-XX:MaxGCPauseMillis 参数允许的值是一个大于0的毫秒数，收集器尽可能地保证内存回收花费的时间不超过该值。-XX:GCTimeRatio 参数是一个大于0小于100的整数，即用户线程时间与垃圾收集时间的比率，相当于吞吐量的倒数，默认值是99，那么允许1%的垃圾回收时间。-XX:UseAdaptiveSizePolicy 开关参数，打开后，不需要设置新生代、Eden与Survivor的比例、今生老年代对象大小等参数，虚拟机动态调节。

### Serial Old 收集器

Serial old 收集器是Serial 收集器的老年代版本，是一个但线程收集器，使用标记整理算法。有两个用途：1. JDK1.5之前的版本与Parallel Scavenge收集器搭配使用；2. 作为CMS收集器的后备方案，在并发收集发生Concurrent Mode Failure时使用。

### Paralel Old 收集器

![desc](data:None;base64,)

### CMS 收集器

![desc](data:None;base64,)

老年代，标记清理算法。是一种以获取最短回收停顿为目标的收集器。分四个步骤：初始标记->并发标记->重新标记->并发清除，初始标记和重新标记仍需要 Stop The World。缺点：1. 由于并发，对CPU资源非常敏感；2. 无法收集浮动垃圾；3. 标记清除算法产生大量空间碎片。由于并发清理阶段，用户线程产生的垃圾，CMS无法清除，这部分垃圾称为浮动垃圾。由于并发，CMS不能像其他垃圾收集器那样等到老年代几乎填满在进行收集，需要预留一部分让程序运行。-XX:CMSInitiatingOccupancyFraction 的值来设置触发垃圾收集的百分比，JDK1.5默认68%，JDK1.6默认92%。由于会产生空间碎片，老年代可能无法为大对象提供内存就要触发 Full GC。-XX:+UserCMSCompactAtFullCollection 开关参数，默认开启。用于进行Full GC时开启内存碎片的合并整理。-XX:CMSFullGCsBeforeCompaction 这个参数用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的，默认值时0，表示每次进入Full GC时都进行碎片整理。

### G1 收集器

![desc](data:None;base64,)

特点：1. 并行与并发；2.分代收集；3.空间整合；4.可预测的停顿。化整为零的思路，把java堆分成多个region，使用remebered set避免全堆扫描，如果引用对象在不通的region，通过CarTable把相关引用信息记录到被引用对象所属的Region的Remembered Set中，当进行内存回收时，在GC根结点的枚举范围加入Remembered Set即可保证不对全堆扫描也不会有遗漏。

# 类文件结构

## magic (u4)

java的魔数为：0xCAFEBABE 。使用魔数而不是用扩展名来进行标识是因为据哦站名可以随意改动。

## minor\_version(u2)

次版本号

## major\_version(u2)

主版本号

## constant\_pool\_count(u2)

常量池容量计数值容量池计数从1开始，将第0项常量空出来是为了表示不引用任何常量池项目的情况。

## constant\_pool

常量池主要存放两大类常量：字面量和符号引用。符号引用包含三类常量：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。

### CONSTANT\_Utf8\_info

标志：1，UTF-8编码的字符串。结构：tag(u1)标志位，length(u2)值说明了UTF-8字符串长度是多少个字节。bytes(u1)长度位length。由于length的长度就是java中方法和字段名的最大长度，u2类型就是65535即64KB。

### CONSTANT\_Integer\_info

标志3，整型字面量。

### CONSTANT\_Float\_info

标志4，浮点型字面量。

### CONSTANT\_Long\_info

标志5，长整型字面量。

### CONSTANT\_Double\_info

标志6，双精度浮点型字面量。

### CONSTANT\_Class\_info

标志7，类或接口的符号引用。结构：tag(u1)数量为1，name\_index(u2)。tag是标志位，name\_index是一个索引值，指向常量池中的一个CONSTANT\_Utf8\_info类型常量。

### CONStANT\_String\_info

标志8，字符串类型的字面量

### CONSTANT\_Fieldref\_info

标志9，字段的符号引用。

### CONSTANT\_Methodref\_info

标志10，类中方法的符号引用。

### CONSTANT\_InterfaceMethodref\_info

标志11，接口中方法的符号引用。

### CONSTANT\_NameAndType\_info

标志12，字段或方法的部分符号引用。

### CONSTANT\_MethodHandle\_info

标志15，表示方法句柄。

### CONSTANT\_MethodType\_info

标志16，标识方法类型。

### CONSTANT\_InvokeDynamic\_info

标志18，表示一个动态方法调用点。

## access\_flags(u2)

访问标志位用于识别类或者接口层次的访问信息，包括：这个Class是接口还是类；是否定义为public类型；是否定义为abstract类型；如果是类的话，是否被声明为final等。

## this\_class(u2)

类索引用于确定这个类的全限定名。

## super\_class(u2)

父类索引用于确定这个累的父类的全限定名。除了Object类，所有类的索引都部位0。

## interfaces\_count(u2)

接口计数器，如果没有接口，该计数器的值为0.

## interfaces

接口索引集合

## fields\_count(u2)

## fields

字段表集合，包括类级别变量以及实例级变量。结构：access\_flags(u2)字段修饰符放在其中；name\_index(u2)字段的简单名称；descriptor\_index(u2)字段和方法的描述符；attributes\_count(u2);attributes(attribute\_info);

## methods\_count(u2)

## methods

方法表集合结构：access\_flags(u2);name\_index(u2);descriptor\_index(u2);attributes\_count(u2);attributes(attribute\_info);如果父类方法在子类中没有被重写，方法表中不会出现来自父类的方法信息。

## attributes\_count(u2)

## attributes

属性表集合，在Class文件、字段表、方法表都可以携带自己的属性表集合，用于描述某些场景专有的信息。

# 类加载

![desc](data:None;base64,)

类加载过程包括：加载、验证、准备、解析、初始化、使用、卸载。其中加载、验证、准备、初始化和卸载这五个阶段必须按部就班开始，解析阶段则不一定某些情况下可以在初始化之后开始。类加载时机虚拟机规范并没有定义，但是定义了初始化时机，而加载肯定在初始化之前：1.遇到new，getstatic，putstatic或者invokestatic这4条指令时，如果类没有初始化则需要初始化。即实例化对象的时候、读取或设置一个类的静态字段的时候以及调用一个类的静态方法时；2.使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行初始化则需要先触发初始化；3.当初始化一个类的时候发现父类没有初始化，则需要先触发其父类的初始化；4.当虚拟机虚拟机先触发main方法的主类；5.jdk1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析记过REF\_getStatic、REF\_puStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄对应的类没有进行过初始化则需要进行初始化。

## 类加载过程

### 加载

加载阶段要完成三件事：1.通过一个类的全限定名来获取定义此类的二进制字节流；2.将这个字节流所代表的静态存储结构转化为方法去的运行时数据结构；3.在内存中生成一个代表这个类的Class对象，作为方法去这个类各种数据的访问入口。

### 验证

该阶段的目的是确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

* 文件格式验证

验证内容：1.是否以魔数0xCAFEBABE开头；2.主、次版本号是否在当前虚拟机处理范围内；3.常量池的常量是否又不支持的常量类型（检查常量tag标志）4.CONSTANT\_Utf\_info是否有不符合utf8编码的数据；5.Class文件中各个部分即文件本身是否有被删除的或附加的其他信息；…………

* 元数据验证

验证点：1.这个类是否有父类（除Object类外所有类都需要有父类）；2.这个类的父类是否继承了不允许被挤成的类（被final修饰的类）；3.如果这个类不是抽象类，是否实现了父类或接口中要求实现的所有方法；4.类中的字段、方法是否与父类产生矛盾（例如覆盖了父类的final字段，出现不符合规则的方法重载，如方法参数都一致，返回值类型不通）…………

* 字节码验证

该阶段的目的是通过数据流和控制流分析，确定程序语义是合法的符合逻辑的，该阶段将会对类的方法体进行校验分析，保证被校验类的方法在运行时不会作出危害虚拟机安全的事件。验证点：1.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；2.保证指令不会跳转到方法体以外的字节码指令上；3.保证方法体重的类型转换是有效的…………

* 符号引用验证

发生在虚拟机将符号引用转化为直接引用的时候，在解析阶段发生。验证点：1.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；2.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；3.符号引用中的类、字段、方法的访问性是否可以被当前类访问……

### 准备

准备阶段为类变量分配内存并设置类变量的初始值的阶段。

### 解析

解析阶段是虚拟机将常量池中的符号引用替换为直接饮用的过程。符号引用：以一组符号来描述引用目标，符号可以是任何形式的字面量，只要使用时能无歧义的定位到目标即可。直接引用：可以直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。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、方法句柄和调用点限定符7类符号引用进行，分别对应于常量池的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info、CONSTANT\_MethodType\_info、CONSTANT\_MethodHandle\_info和CONSTANT\_InvokeDynamic\_info 7种常量类型l 。

* 类或者接口的解析

假设当前代码所处类为D，符号引用为N对应的类或接口为C，则解析过程如下：如果C不是一个数组类型，那虚拟机会把全限定名N传递给D的类加载器去加载C，过程中由于元数据校验、字节码校验的需要又可能触发其他相关类的加载过程。如果C是一个数组类型，并且数组原属类型为对象，那先按照第1步加载数组元素类型，然后由虚拟机生成一组代表词数组唯独和元素的数组对象。如前2步无异常，那么C在虚拟机中已成为一个有效的类或接口了，但是在解析完成前还需要进行符号引用验证，确认D是否具备对C的访问权限，如果无权限则抛出java.lang.IllegalAccessError异常。

* 字段解析

字段所属的类或者接口用C表示，步骤如下：如果C本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。否则，如果C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述都与目标匹配的字段，则返回这个字段的直接引用，查找结束。否则，如果C不是Object的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。否则，查找失败，抛出java.lang.NoSuchFieldError异常。在查找过程返回了引用将会对字段权限进行验证，如果不具备访问权限将抛出java.lang.IllegalAccessError异常。

* 类方法解析

类方法解析和字段解析第一步相同，先解析出类方法表的class\_index项中所属的类或接口的符号引用，如果解析成功用C来表示这个类进行如下步骤：类方法和接口方法符号引用的常量类型定义是分开的，如果类方法表中class\_index对应索引C是接口，那直接抛出java.lang.IncompatibleClassChangeError异常。如果通过了第一步，在此类中超着简单名称和描述符都与目标相匹配的方法，如果由返回直接引用查找结束。否则，在父类中查找如果由返回直接引用，查找结束。否则，在C类实现的接口列表以及他们的父类中查找如果找到返回，查找结束。否则宣告失败，抛出java.lang.NoSuchMethodError。最后对方法进行权限校验，如果无权限，抛出java.lang.IllegalAccessError。

* 接口方法解析

根据接口方法表的class\_index，解析出对应接口C，然后进行如下步骤：如果C是个类不是接口，抛出异常java.lang.In compatibleClassChangeError。否则，在接口中找简单名称和描述方法都与目标相匹配的方法，如果由则返回这个方法的直接引用。否则，在接口的父类中递归查找。否则在C的父接口中递归查找。否则查找失败，抛异常java.lang.NoSuchMethodError。

### 初始化

初始化阶段是执行类构造器<clinit>()方法的过程。<clinit>()方法是有编译器自动收集类中的复制动作和静态语句块(static{}块)合并产生的，收集顺序由语句出现的顺序决定，静态块只能访问到静态块之前的变量。<clinit>()方法与累的构造方法不通，不需要显式调用父类构造器，虚拟机会保证子类的<clinit>()方法执行之前先执行父类的。这也意味着父类的静态语句优于子类。<clinit>()方法对于类和接口是非必须的，如果一个类中没有变量赋值和静态代码块，那么编译器可以不为该类生成<clinit>()方法。接口中不使用静态代码块，但是可以由赋值操作，因此也会生成<clinit>()方法。但是接口不需要先执行父类接口的<clinit>()方法，只有父类接口变量被使用的时候才会执行父类<clinit>()方法。虚拟机保证一个类的<clinit>()方法再多线程环境中正确的加锁同步。

## 类加载器

通过一个类的全限定名来获取此类的二进制字节流的代码块被称为类加载器。启动类加载器，负责加载<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数置顶的路径中的并且被虚拟机是表的类库加载到虚拟机内存中。扩展类加载器，由sun.misc.Launcher$ExtClassLoader实现，负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量定指定路径中的所有库类。应用程序类加载器，由sun.misc.Launcher$AppClassLoader实现，负责用户类路径(ClassPath)上所制定的类库。用户自定义类加载器。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都要有自己的父类加载器，但不以继承关系来实现，而是使用组合关系来服用类加载器代码。双亲委派模型工作过程：如果类加载器收到加载类的请求，首先把这个请求委派给父类完成，每层类加载器都是如此，只有父类反馈自己无法加载时，子加载器才会尝试自己加载。

# 字节码执行引擎

## 栈帧

用于支持虚拟机进行方法调用和方法执行的数据结构，他是虚拟机运行时数据曲中的虚拟机栈的栈元素。包含了局部变量表、操作数栈、动态链接、方法返回地址和一些附加信息。

### 局部变量表

用于存放方法参数和方法内部定义的局部变量。

### 操作数栈

### 动态链接

每一个栈帧都包含了一个执行运行时常量池中该栈帧所属方法的引用，该引用是为了支持调用过程中的动态连接。字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数，这些符号引用一部分会在类加载阶段或第一次使用的时候就转化为直接引用，这种转化成为静态解析。另一部分将在每一次运行期间转化为直接引用，这部分成为动态连接。

### 方法返回地址

方法返回方式有两种一种是遇到返回指令另一种是异常退出。

### 附加信息

## 方法调用

方法调用不是方法执行，方法调用的任务是确定调用发放的版本。

### 解析

方法在程序运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期间不可改变，即调用目标在程序代码写好、编译完成时就确定下来，这类方法的调用称为解析。java中编译器可知，运行期不可变的方法主要包括静态方法和私有方法。java中调用方法的字节码指令：invokestatic（调用静态方法）、invokespecial（调用构造器方法、私有方法和父类方法）、invokevirtual（调用所有虚方法）、invokeinterface（调用接口方法）、invoekdynamic（调用动态方法）其中invokestatic和invokespecial调用的方法可以在解析阶段唯一确定，在解析阶段可以把符号引用解析为直接引用的方法称为非虚方法，其他方法称为虚方法。final修饰的方法也是非虚方法。

### 分派

* 静态分派

Human man = new Man();这句代码中Human为静态类型或者叫做外观类型，后面的Man为实际类型，虚拟机在重载时根据静态类型而不是实际类型作为判定依据，所有以来静态类型来定位方法执行版本的分派动作称为静态类型。

* 动态分派

动态分派和重写密切相关。invokevirtual指令步骤：找到栈顶第一个元素所指向的对象的实际类型，计作C。如果C中找到常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过查找结束，不通过，则返回java.lang.IllegalAccessError。否则，按照继承关系递归查找C的各个父类进行第二步操作。如果没有找到，抛出java.lang.AbstractMethodError。由于第一步在运行期确定实际类型，所以方法可被解析道不同的引用上。我们把在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

* 单分派与多分派

方法的接收者和方法的参数统称为宗量。单分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。java目前为止是一个静态多分派，动态单分派的语言。

# 内存模型与线程

## 内存模型

java中所有的变量都存在祝内存中，每个线程有自己的工作内存工作内存中保存了该线程使用的变量的主内存副本。

### 内存操作

lock:作用于主内存的变量，把一个变量标识为线程独占状态。unlock：作用域主内存变量，把一个处于锁定状态的变量释放出来，释放后可以被其他线程锁定。read：作用域主内存变量，把一个变量值从主内存出啊疏导线程的工作内存中，以便随后的load动作。load：作用于工作内存的变量，把read操作从主内存得到的变量放到工作内存的变量副本中。use：作用于工作内存变量，把变量值传递给执行引擎。assign：作用于工作内存的变量，把一个执行引擎的值父给工作内存的变量。store：作用于工作内存的变量，把工作内存的变量的值传递给主内存中。write：作用于主内存的变量，把store操作从国做内存中得到的变量值放到主内存中。java内存模型规定在执行上面8种操作时必须满足如下规则：read和load，store和write不能单独出现。不能废弃assign操作，即变量在工作内存改变了，必须同步到主内存。不允许无原因（没发生任何assign）把数据从线程工作内存同步到主内存。新的变量只能在主内存中诞生。一个变量同一时刻只允许一条线程对其进行 lock，但是同一线程可以被进行多次lock，lock多少次就要unlock多少次。如果对一个变量执行了lock操作，那么竟会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新load或assign操作。如果一个变量未被lock则不允许unlock。执行unlock之前必须先把变量同步回主内存。

### volatile

两种特性：保证此变量对所有线程的可见性。禁止指令重排优化。

## 线程

### 线程实现

使用内核线程实现，直接由操作系统内核支持的线程，有内核来完成线程切换和调度并负责把线程映射到各个处理器上。程序一般不实用内核进程，而是使用内核线程的接口，轻量级进程，轻量级进程和内核线程对应关系为1:1,缺点是调用代价高，用户态和核心态切换，系统支持轻量级进程的数量有限。使用用户线程实现，优势在于不需要系统内核支持，缺点是复杂。使用用户线程和轻量级进程混合实现。线程的建立在用户空间，使用轻量级进程作为内核线程的桥梁，用户闲线程和轻量级进程关系为n:m。

### 线程调度

协同式线程调度：线程执行时间由线程本身控制，线程把自己执行完之后主动通知系统切换到另一个线程上。优点：无线程同步问题。缺点：线程执行时间不可控。抢占式线程调度：每个线程由系统分派执行时间，线程切换不由线程本身决定不会有线程阻塞导致整个进程崩溃。

# 线程安全

## 实现方法

### 互斥同步

多线程并发访问共享数据时，保证共享数据在同一时刻只被一个线程使用。

### 非阻塞同步

基于冲突检测的乐观并发策略，就是先进性操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就采取其他补偿措施（最常见的措施就是不断重试直到成功为止），这种策略不需要把线程挂起因此称为非阻塞同步。例如CAS

### 无同步方案

## 锁优化

### 自旋锁与自适应自旋

当多个线程争共享数据时，可以让为争夺到的线程忙循环，这项技术就是自旋。自旋的默认次数为10，可以用-XX:PreBlockSpin来更改。自适应意味着自旋的时间不再固定，由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定。

### 锁消除

虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除判定依据来源于逃逸分析的数据支持，如果判断一段代码中，堆上所有数据不会逃逸出去从而被其他线程访问，那就可以把他们当作栈上数据对待，同步加锁无需进行。

### 锁粗化

如果虚拟机探测到有一堆零碎操作都对同一个对象加锁，将会把加锁同步的范围扩展到整个操作徐磊的外部。

### 轻量级锁

### 偏向锁