**深入理解Java虚拟机**

虚拟机是棋盘，程序代码是棋谱。

**第一章 走近Java**

**第二部分 自动内存管理机制**

**第二章 Java内存区域与内存溢出异常**

线程私有的概念我的理解是一块大的内存区域里不同线程拥有自己的一小块，互相之间没有访问权限，像军阀割据，比如程序计数器和栈区

线程共享就是所有线程对这块大内存都有访问权限，像是共产主义接班人。

**程序计数器**（program counter register）：是一块较小的内存空间，可以看做是当前线程所执行的字节码的行号指示器。由于多线程是通过线程轮流切换并分配处理器执行时间来实现的，为了能在线程切换后回到正确的执行位置，每条线程都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，称之为“线程私有的”内存。若线程执行的是Java方法则计数器记录正在执行的虚拟机字节码指令的地址，若执行的是Native方法，则计数器值为空。这个内存区域是Java虚拟机中唯一没有规定OutOfMemoryError情况的区域。

Java的本地方法（Native Method）是Java中调用非Java语言方法的接口。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 ;

    }

    这些方法的声明描述了一些非java代码在这些java代码里看起来像什么样子（view）.

**Java的虚拟机栈**：也是线程私有的，生命周期和线程相同，描述的是Java方法的内存模型，每个方法执行的时候会创建一个栈帧用于存放局部变量表、操作数栈、动态链接、方法出口等信息。每个方法的调用到执行完成就对应着栈帧在虚拟机栈中的入栈和出栈。局部变量表存放了编译器可知的各种基本数据类型和对象引用类型，局部内存表所需要的内存空间在编译期间完成分配，当进入一个方法时需要在帧中分配多大的局部变量空间是确定的，在运行期间不会改变。Java的虚拟机栈会抛出StackOverflowError（请求的栈深度超过虚拟机所允许的深度）和OutOfMemoryError异常（虚拟机栈大小可以动态扩展时却无法申请到足够的内存时）

**本地方法栈**：和Java的虚拟机栈基本一样，区别在于本地方法栈为虚拟机中用到的Native方法服务。

**Java堆**：是虚拟机中所管理的最大的一块内存区域，被所有线程共享，在虚拟机启动时创建。堆的唯一目的是存放对象实例，也是垃圾收集器GC管理的主要区域，Java堆可以在物理上不连续，只要在逻辑上连续即可。内存不够完成实例分配时抛出OutOfMemoryError。

**内存泄漏**指由于疏忽或错误造成程序未能释放已经不再使用的[内存](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98)。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

**方法区**（method area）与Java堆一样是线程共享的，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，包括类名、访问修饰符、常量池、字段描述、方法描述等。方法区不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集，这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。**运行时常量池**（runtime constant pool）是方法区的一部分。Class文件除了有类的版本、字段、方法、接口等描述信息外还有一项信息是常量池，用于存放编译器生成的各种字面量和符号引用，这部分内容会在类加载后进入方法区的运行时常量池中存放。运行时常量池具备动态加入的动态性。也会有OutOfMemoryError。

**直接内存**：不是虚拟机运行时数据区的一部分，但是被频繁使用，受制于本机内存。直接内存是堆外内存，不是虚拟机规范中定义的内存区域，也可能导致OutOfMemoryError异常出现。NIO类中引入基于通道和缓冲区的IO方式，使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的应用进行操作，从而避免在Java堆和Native堆中来回复制数据。DirectBuffer比普通Buffer读写更快，但是它的创建、销毁比普通Buffer慢。

**对象的创建过程**首先在方法区中查看能否找到所代表的类已被加载、解析和初始化过，否则执行类加载过程。检查完类加载过程后对象所需的内存大小便可完全确定。为新生的对象分配内存在Java堆区规整的时候可以用指针碰撞的方法，不规整的话可以用空闲列表法分配足够大的空间给对象。为了使创建对象成为线程安全的，可以采用本地线程分配缓冲（TLAB）的方法，在Java堆中为每个线程预先划分出一块内存，哪个线程需要创建对象就在哪个线程的TLAB上分配。虚拟机分配给新生对象的内存空间会初始化为零值，所以对象的实例变量可以不赋初始值使用，执行完new指令后会进行init方法的执行。

对象在内存中存储的布局主要分成三个部分：1对象头2实例数据3对齐填充。其中对象头包含类型指针，也就是所属哪个类，如果是数组则还包括数组大小，对象头的另外一部分是存储对象自身的运行时数据，包括hashcode、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分通常为32bits或64bits。对齐填充不是必须的，只有在虚拟机要求对象起始地址是8字节的整数倍时用于填充满足这一条件。

对象的类型数据指的是类加载的信息，存在方法区中。

访问一个对象的方法有两种。通过句柄访问是对象引用指向句柄，句柄分别指向对象实例数据和对象的类型数据，好处是对象被回收时也无需改变对象引用的指向，只要改变句柄的指向即可。直接指针访问的话对象引用直接指向对象地址（对象地址能指向对象的类型数据），好处是速度更快。

通常操作系统分给一个进程的内存空间是有限制的，除去少部分的程序计数器外，Java堆和方法区可以设置上限值，剩下的内存部分被虚拟机栈和本地方法栈瓜分，如果生成的线程多，那么每个线程允许的栈空间大小会减小，如果每个线程的栈空间分配的大则能允许建立的线程数就越少。

**第三章 垃圾收集器与内存分配策略**

判断一个对象是否存活的方法包括引用计数算法。对象多一个引用，计数就加一，反之减1，计数到0时表示没有被引用，就回收。缺陷是无法解决对象间相互引用的问题（如生成的a对象内引用了b对象，生成的b对象内引用了a对象，则两个对象引用计数都为2，分别去掉生成时的引用，则无法再访问这两个对象，但是引用计数为1，无法GC）。

所以真正应用的是**可达性分析算法**来判断对象是否存活。通过从GC roots对象作为起始点看是否能达到对象，不可达时说明对象是不可用的。GC roots对象包括1虚拟机栈中引用的对象2方法区中静态属性引用的对象3方法区中常量引用的对象4本地方法栈中引用的对象

引用可以分为强引用(不会被回收)>软引用（在内存将要溢出的时候回收）>弱引用（下一次垃圾收集的时候回收）>虚引用（不影响生存时间，只是在回收时会受到系统通知）

方法区的垃圾收集效率较低。主要回收废弃的常量和无用的类。废弃的常量指的是没有任何地方引用这个常量。无用的类要满足三个条件：1该类所有的实例都已经被回收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引用，也就是无法在任何地方通过反射访问该类的方法。

垃圾收集算法是基于”标记-清除“算法的

Java堆中分为新生代和老生代，新生代是新创建的对象，统计表明更容易被废弃和回收，因此采用复制算法回收，老生代则是在多次回收过程中存活的对象，更不容易被废弃和回收，所以采用标记-清除法或者标记-整理法回收。

枚举根节点的时候需要停顿所有的Java线程以免对象引用关系还在变化

虚拟机如HotSpot虚拟机使用OopMap的数据结构来记录哪些地方存放着对象引用。程序执行到“安全点”的时候生成OopMap。

现在都采用**主动式中断的方式来让GC时线程都停下来**。主动式中断：GC要操作时就设置一个标志，然后各个线程运行到安全点或者创建对象需要分配内存的时候主动去轮询这个标志，发现中断标志位真时主动停止。有时线程Sleep状态或者blocked的时候无法执行到安全点，此时线程可愿意标识自己进入了safe region状态，此时的线程不会引起引用状态的改变，这样GC就不用管处于safe region状态的线程了，处于safe region状态的线程要等到根节点枚举完成或者是GC过程结束才能走出safe region。

**Serial收集器**是最基本最古老的收集器，对新生代采用复制方法，对老生代采用标记-整理方法，单线程操作，操作时必须停止其他线程，由于没有线程间的交互所以效率高，是虚拟机在Client模式下默认的新生代收集器。

**ParNew收集器**可以看做是Serial收集器的并行版本，有多条线程完成垃圾收集，是虚拟机在Server模式下默认的新生代收集器，可以与CMS收集器配合使用。

**Parallel Scavenge收集器**也是一个新生代收集器，也采用复制算法、也是多线程并行执行。但是关注吞吐量。（停顿时间参数适用于与用户交互的程序，停顿时间越短用户体验越好，吞吐量指的是CPU用于执行用户代码的时间/(执行用户代码的时间+垃圾收集的时间)，吞吐量指标更适合于后台计算）该收集器可以采取自适应调节策略，让许多具体参数交由虚拟机来管理。

**Serial Old收集器**是Serial收集器的老年代版本，单线程，使用标记-整理算法。

**Parallel Old收集器**是Parallel Scanvenge收集器的老年代版本。

我的理解是单线程版本的在服务器端CPU多个时不如多线程的能发挥出硬件能力。

**CMS(Concurrent Mark Sweep)收集器**是以获取最短回收停顿时间为目标的收集器，分为初始标记（标记能与GC ROOTS直接关联到的对象）、并发标记、重新标记、并发收集四个步骤，其中初始标记和重新标记用时较短，需要停止其他线程，耗时较长的并发标记和并发收集两个步骤可以与其他线程并发执行，以此带来最短的回收停顿时间，缺点是对CPU资源敏感，而且无法处理并行清理时新产生的浮动垃圾，采用的标记-清除方法会带来内存碎片化的问题。

**G1收集器**是一款面向服务器端的垃圾收集器，具有并发和并行特征，G1可以完成老年代和新生代两个区域的收集，采用标记-整理方式实现，停顿可预测。通过将Java堆分成多个大小相等，相互独立的region，跟踪各个region，维护一个优先列表，看在给定的手机时间内收集哪个region的价值最大就收集哪个region。使用remember set来避免回收某个region的时候进行全堆扫描。运作的步骤和CMS收集器有点类似**。**

新生代GC（minor GC）是发生在新生代上的GC，比较频繁也比较快。

老年代GC（major GC/Full GC),速度比较慢，表明这次GC是发生了stop-the-world的

大对象是需要连续内存空间的Java对象，典型的是长字符串和数组，经常出现大对象会需要更多的GC来获得连续空间来安置它们，可以设置参数令大于门限值大小的对象直接进入老年代中。

分配担保机制是在对象无法放入Survivor空间时将对象转移到老年代中，尤其考虑到新生代采用复制方法进行GC，当一个Survivor无法容纳时就借用老年代的空间。

每个对象有一个年龄计数器，对象在Eden出生，并且经过minor GC后能被Survivor空间接纳的话就移入Survivor空间，年龄为1，此后每熬过一次GC年龄加1，可以设置门限在年龄大于某个阈值时移入老年代中。当然也不是这么固定死板，当Survivor空间中同一个年龄的对象占据空间超过Survivor空间的一半时，大于等于该年龄的对象都进入老年代。

HandlePromotionFailure来设置是否允许分配担保失败，因为如果老年代中最大可用连续空间大于新生代对象总空间的话分配担保一定成功，否则有概率不成功，如果分配担保失败就要进行Full GC了。

**第四章 虚拟机性能监控与故障处理工具**

这句话说的太好了。知识、经验是关键基础，数据是依据，工具是运用知识处理数据的手段！

这里说的数据包括运行日志、异常堆栈、GC日志、线程快照、堆转储快照等。

jdk下的bin目录中有很多exe程序都是虚拟机性能监控的工具，要妥善利用

**第三部分 虚拟机执行子系统**

代码编译的结果从本地机器码转变为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

**第六章 类文件结构**

实现语言无关性的基础是虚拟机和字节码存储格式。Java虚拟机不和任何语言绑定，只与Class文件这种特定的二进制文件格式相关联。

Java虚拟机规范要求在Class文件中使用许多强制性的语法和结构化约束。许多语言都可以通过各自语言的编译器得到Class文件，虚拟机不关心Class文件来自何种语言。

Class文件中没有添加任何分隔符，所以文件中的每个字节代表的意义都是严格规定的。遇到占用8位字节以上空间的数据项时，按照高位在前（最高位字节为地址最低位，最低位字节为地址最高位）的方式分割成若干个8位字节进行存储。

Class文件类似C语言中的伪结构体，其中只有无符号数和表两种数据类型。

u1、u2、u4、u8分别表示1、2、4、8个字节长度的无符号数。

**Class文件格式为：**

u4的魔数（CAFEBABY）

u2的次版本

u2的主版本（注意高版本的JDK可以向下兼容以前版本的Class文件，当拒绝执行更高版本的Class文件）

u2的常量池常量数量（以1开始计数！这是非常特殊的！，例如22表示常量池中有21个常量）

接下来为若干个常量池常量的表述，常量池中主要存放的常量一类是字面量（接近Java语言层面的常量，如字符串、声明为final的常量等），一类是符号引用（包括类和接口的全限定名、字段的名称和描述符、方法的名称和描述符）。常量池中常量采用表进行描述，表中的第一位为u1类型的标志位tag，表示这个常量是那种常量，不同类型的常量都具体规定了相应的描述格式。

u2的访问标志有16个标志位可用，分别对是否是public、是否是final等都给予一个标志值，从最后标志值的总和中可以看出具体每个标志位的标志。

u2的类索引

u2的父类索引

u2的接口数量和u2的接口索引（这几个类、父类、接口索引的值对应index的常量池中可以找到如类、父类、接口的全限定名字符串）构成的接口索引集合

u2的字段数量

字段表集合包含字段的访问标志、名称（表示为在常量表中的索引值）、描述符、属性等。字段表集合中不含从超类或者父接口中继承的字段。

（全限定名表示类全名，简单名称表示没有类型和参数修饰的方法或字段名称，描述符为参数列表+返回值的形式，其中参数列表写在（）中）。

方法数量和方法表集合和字段的相类似，只是访问标志不同。

Java语言中的特征签名和Class文件的特征签名不同，Java中只要求方法名称、参数顺序以及参数类型。字节码中的特征签名还包括方法返回值以及受查异常表。

Class文件、字段表、方法表都可以携带自己的属性表集合，来描述某些场景专有的信息。属性表中对数据项目顺序、长度、内容的限制稍微宽松一些。

Java方法体中的代码经过javac编译器处理后变为字节码存储在Code属性内。

字节码指令指的是每个指令是u1类型的单字节，一个方法不允许超过65535条字节码指令。Code属性是Class文件中最重要的一个属性。Java程序可以分为代码（Code，方法体里的Java代码）和元数据（Metadata，包括类、字段、方法定义以及其他信息）两部分。

实例方法中默认存在一个参数this用来指向当前对象实例。

（这里插播一下关于try/catch/finally的执行顺序的说明，有finally块的一定会执行finally块，除非遇到System.exit(0)语句，try/catch中的能被执行到的return语句中的return的值计算后先临时存放，然后执行finally块，如果finally中也有return的值，则用finally中的返回值替换临时存放的值来作为返回，finally块中最好不要有return值）

由于虚拟机采用基于操作数栈而不是寄存器的架构，因此大多数指令都不包含操作数而只有一个操作码。

Java虚拟机中的指令大都不支持boolean、byte、short、char类型的操作，所以这些数据都会转换为int数据来操作，这些类型的数组也会转为int数组处理。

浮点数截断为整形时在不考虑符号的情况下选择最接近但是不大于原值的数字再加上符号，（int）5.6为5，（int）-5.6=-5，（int）-5.4=-5.

Class文件所具有的平台中立（不依赖与特定硬件和操作系统）、紧凑、稳定和可扩展的特点是Java技术体系实现平台无关、语言无关两项特性的重要支柱。

Class文件是Java虚拟机执行引擎的数据入口。

**第七章 虚拟机类加载机制**

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，最终形成可以直接被虚拟机使用的Java类型，这就是虚拟机的类加载机制。Java的类型的加载、连接和初始化过程都是在程序运行期完成的。Java天生可以动态扩展的语言特性就是依赖于运行期动态加载和动态连接这个特点实现的。

类从加载到虚拟机内存开始到卸载出内存的生命周期包括1加载loading、2验证verification、3准备preparation、4解析resolution、5初始化initialization、6使用using、7卸载unloading

其中加载、验证、准备、初始化、卸载这5个节点的开始顺序是确定的，解析则不一定，某些情况下可以在初始化阶段之后再开始。

虚拟机规范严格规定了有且仅有5种情况必须对类立即进行初始化，除此之外所有引用类的方式都不会触发初始化，被称为被动引用。

这五种情况是：

1遇到new、getstatic、putstatic、invokestatic这四条指令时，通常是在Java语言中使用了new关键字实例化对象或者读取或设置一个类的静态字段的时候

2使用java.lang.reflect包的方法对类进行反射调用的时候

3初始化一个类发现其父类还没初始化时要先对父类进行初始化

4虚拟机启动时含有main（）方法的主类要先初始化

5java.lang.invoke.MethodHandle实例最后的解析结构REF\_getStatic、REF\_putStatic、REF\_invokeStaic的方法句柄并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

接口与类不同的是接口不能使用静态初始化块static{ }，同时在情况3中，当接口初始化时，并不要求其父接口都完成了初始化，只有在真正用到父接口时才会进行初始化。

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过子类来引用父类中定义的静态字段时，只会触发父类的初始化而不会触发子类的初始化。

通过数组定义来引用类不会引发类的初始化。例如SuperClass[]superClasses=new SuperClass[10];不会引起SuperClass类的初始化。

常量在编译阶段会存入调用类的常量池中，本质上并没有引用到定义常量的类，因此不会触发定义常量类的初始化。如类A中有public static final int a=1;类B中调用了类A中的常量a，则B会将这个常量在编译时写入Class文件中的常量池中，编译成Class文件之后就和类A没有关系了，所以不会触发A的初始化过程。

**加载阶段：**

1、通过一个类的全限定名来获取定义这个类的二进制字节流（未规定从哪里获取，可以是JAR、EAR等zip包，也可以从网络中获取，如Applet技术，还可以是运行时计算生成，比如动态代理，可可以是从其他文件生成或者从数据库中读取等），这个过程是放在Java虚拟机外部实现的，实现这个过程的代码模块称为类加载器。任意一个类都需要由加载它的类加载器和这个类本身一同确立在Java虚拟机中的唯一性。比较两个类是否相等的前提是两个类的类加载器是同一个。类加载器之间的关系模型是双亲委派模型，在这个模型中，自底向上为自定义类加载器->应用程序类加载器（系统类加载器，负责用户类路径上的类的加载）->扩展类加载器（负责加载<JAVA\_HOME>\lib\ext目录或是java.ext.dirs系统变量所指定的路径中的类库）->启动类加载器（负责加载<JAVA\_HOME>\lib中的可以被虚拟机识别的类库，无法被Java程序直接引用）。除了顶层的启动类加载器外，其余类加载器都要有自己的父亲加载器，这里的父子关系不是用继承来实现，而是使用组合关系来复用父加载器的代码。类加载器收到类加载请求时首先把请求委派给父加载器完成，父亲加载器反馈无法完成时才由子加载器尝试自己去加载，可以理解为父加载器优先级高于子加载器。这种双亲委派模型可以被破坏，比如为SPI代码执行引入的线程上下文类加载器和OSGi中实现模块化热部署的类加载机制等。

2、将这个字节流所代表的静态存储结构转化为方法区中运行时数据结构

3、在内存中生成一个代表这个类的java.lang.Class对象，作为方法区中这个类型的各个数据的访问入口，java.lang.Class对象没有指明是否生成在堆区中，在HotSpot虚拟机中java.lang.Class对象存放在方法区中

对于非数组类的加载，可以使用系统提供的引导类加载器或者自定义类加载器去完成，数组类本身不通过类加载器创建，而是由Java虚拟机直接创建。如果数组的组件类型是引用类型，就递归采用加载过程去加载这个组件类型，数组将会在加载该组件类型的类加载器的类名称空间上被标识，一个类必须与类加载器一起确定唯一性，如果数组的组件类型不是引用类型，虚拟机会把数组标记为与引导类加载器关联。数组类的可见性和它的组件的可见性一致。

**验证阶段：**

验证是连接节点的第一步，目的是保证Class文件中的信息符合当前虚拟机的要求，不会危害虚拟机的安全。验证到输入的字节流不符合Class文件约束的时候会抛出java.lang.VerifyError异常及其子类异常。如果代码确定了很安全可以把验证阶段关掉来提高类加载速度。

验证过程主要包括：

1、文件格式验证，保证输入的字节流能正确的解析并存储于方法区格式之内

2、元数据验证，对字节码描述的信息进行语义的分析，使其符合Java语言规范要求

3、字节码验证，主要对数据流和控制流分析

4、符号引用验证，发生在虚拟机将符号引用转化为直接引用时，这是在解析阶段发生的，符号引用验证是对类自身以外（常量池中各种符号引用）信息进行匹配性校验，看引用的类、字段、方法之类的能不能找到，目的是保证解析动作能正常执行。

**准备阶段：**

准备阶段是正式为类变量（static变量）分配内存并设置初始值的节点，在内存的方法区中，此时尚未执行Java中的任何方法，所以都采用零值（boolean类型默认为false），除了 static final类型的变量在ConstantValue属性上有指定值的会被设定为指定值。

**解析：**

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。符号引用：以一组符号来描述所引用的目标，不必一定已经加载到内存中，与内存布局无关。直接引用：是可以直接指向目标的指针、相对偏移量或是能间接定位到目标的句柄，与内存有关，能直接引用的目标一定是内存中已经存在的。我的理解是符号引用是Class文件中的描述，直接引用是映射到内存中的位置，相当于是在内存中的具体实现。

解析可以对第一次解析的结果进行缓存避免对同一个目标的多次重复解析。解析主要针对类或接口、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。字段、方法的解析都要先解析对应的类或者接口。同名的字段出现在父类和接口中时编译器可能拒绝编译并抛出“the field xxx is ambiguous”的提示。

**初始化：**

准备阶段已经赋过一次系统要求的初始值，而初始化阶段根据程序员制定的主观计划来初始化类变量和其他资源，也就是执行类构造器<clinit>方法的过程。<clinit>由编译器自动收集类中的所有类变量的赋值动作和静态初始化语句块中的语句合并产生的。静态初始化块只能访问定义在它之前的类变量，定义在它之后的类变量，它可以赋值但是不能访问。父类的静态初始化块在子类之前执行。

<clinit>方法不是必需的。如果没有静态语句块或者对变量的赋值操作就不需要。

接口中不能有静态语句块，但仍然有变量初始化的赋值操作，执行接口的<clinit>方法不需要先执行父接口的<clinit>方法。

虚拟机会保证一个类的<clinit>操作能被正确的加锁和同步化。

**第八章 虚拟机字节码执行引擎**

物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的。

虚拟机的执行引擎是由自己实现的，可以自行制定指令集与执行引擎的结构体系，并且执行那些不被硬件直接支持的指令集格式。

栈帧（stack frame）是支持虚拟机进行方法调用和方法执行的数据结构。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。

栈帧中需要多大的局部变量表、多深的操作数栈都已经在编译的时候完全确定，并写入Class文件的Code属性中去。

**局部变量表：**

存放方法参数和方法内部定义的局部变量，以Slot为基本存储空间（要求一个Slot空间能存储32位以内的数据类型，如boolean、byte、char、short、int、float、reference、returnAddress等）。其中reference类型表示对一个对象实例的引用，要求要能从引用中查找到对象在堆中的起始地址索引，还要查找到对象所属数据类型在方法区中存储的类型信息。实例方法中第0个slot表示this对象的引用。变量存储分配的slot可以程序运行到超出这个变量的作用域后将这个slot分配给别的变量使用。

**操作数栈：**

在方法的执行过程中会有各种字节码指令往操作数栈中写入和提取内容。概念模型中，两个栈帧是完全相互独立的，但是实现时让两个栈帧出现一部分重叠来共用一部分数据而不需要额外的参数复制传递。

**动态连接：**

每个栈帧中都包含一个指向运行时常量池中该栈帧所属方法的引用，来支持方法调用过程中的动态连接。

静态连接：Class文件中常量池的大量符号引用在类加载阶段或第一次使用的时候就转化为直接引用

动态连接：这些符号引用在每一次运行期间转化为直接引用

**方法返回地址：**

方法退出有两种情况，一种是遇到任何的返回的字节码指令，这是正常返回出口，另外一种是遇到异常并且没有在方法体内得到处理，这是异常返回出口，这种方式不会产生任何返回值。方法退出后回到方法被调用的位置，这个位置可以用调用者的PC计数器的值。

**方法调用**

方法调用不同于方法执行，方法调用唯一的任务是确定被调用方法的版本。包括解析调用和分派调用。解析和分派二者不是二选一的排他关系，而是在不同的层次上去筛选、确定目标方法的过程。例如静态方法在类加载器完成解析，而静态方法也能有重载版本，选择重载版本的过程由静态分派完成。

**解析调用：**

在类加载时能通过解析将方法的符号引用转变为直接引用的前提是方法在运行前就有一个可以确定的调用版本，并且在运行时不可变。这类方法的调用称为解析，主要包括通过invokestatic调用的静态方法和通过invokespecial调用的私有方法、父类方法和实例构造器<init>方法两大类。这些方法也称为非虚方法，有final修饰的方法也是非虚方法。**解析调用**是静态过程，在编译期间就完全确定，在类装载过程中的解析中就把涉及的符号引用变为确定的直接引用。

面向对象的三个基本特征是封装、继承和多态

**分派调用：**

**分派调用**可以是静态的也可以是动态的

以Human man=new Man（）；为例，Human是引用变量类型，也称为变量的静态类型，后面的Man是变量的实际类型。静态类型的变化只在使用时发生，变量本身的静态类型不会被改变而且最终的静态类型是在编译期可知的。

**静态分派：**

重载时是通过参数的静态类型而不是实际类型来作为判断依据的。所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。静态分派发生在**编译阶段**，此时尚不涉及到Java虚拟机，典型应用是方法重载。字面量不需要定义（比如12.0），所以字面量没有显式的静态类型，此时选择重载方法是有优先级的为字面量选择更加合适的版本。

**动态分派：**

通过invokevirtual指令调用的是虚方法（有final修饰的话则为非虚方法）。这个指令从操作数栈顶第一个元素指向对象的实际类型按继承关系一路往上找是否有符合的方法。

在**运行期间**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。典型应用是方法覆盖（override也称为重写）

方法的接收者（方法的所有者，比如一个对象的方法的接收者就是这个对象）和参数统称为方法的宗量

根据分派基于多少种宗量可以将分派分为单分派和多分派

Java语言中的静态分派属于多分派（在编译期间根据参数和接收者的静态类型），动态分派属于单分派（在静态分派中已经确定了方法的签名，参数不会再改变，这时看接收者的实际类型）。个人理解是静态分派先确定签名，即重载时是那个方法，并且根据静态类型找类（可能是父类），然后动态分派的时候根据类的实际类型找到真正的类，这样类和方法就都确定了。

动态分派可以使用方法表、内联缓存、守护内联等来提高性能。

动态类型语言的关键特征是类型检查的主体过程是在运行期而不是编译期，比如PH、Python、JavaScript，动态类型语言中变量本身无类型，变量值才有类型

静态类型语言是在编译期就进行类型检查过程的语言，比如C++和Java

Java的连接过程不在编译阶段而在类加载阶段

运行时异常如果代码没有运行到这一行就不会有问题，连接时异常即使不会运行到这一行也会有问题

Java语言无法将一个函数作为参数传递，所以像sort方法要将带compare（）方法的Comparator接口作为参数，不过在拥有MethodHandle后，Java语言也能拥有类似函数指针或者委托的方法别名的工具。

反射Reflection和MethodHandle机制都在模拟方法调用，但是Reflection是重量级的，模拟Java代码层次的方法调用，只是站在Java语言的角度看，而MethodHandle是在模拟字节码层次的方法调用，是轻量级的，可以服务于Java虚拟机之上的语言。

每一处含有invokedynamic指令的位置都被称为动态调用点

invokedynamic的分派逻辑是由程序员决定的，而前面的四种invoke指令是由虚拟机决定的

Java语言中从程序源码到指令流之间的编译过程是在虚拟机之外进行的，而解释器是在虚拟机内部的，所以Java程序的编译就是半独立的实现。

基于栈的指令集：指令大部分是零地址指令，优点是可移植性，代码相对紧凑（字节码中一个字节对应一条指令），编译器实现简单（不需要分配空间，所有空间在栈上操作），缺点是慢（虽然代码紧凑但是代码数量大，因为出入栈操作本身也需要代码，而且栈是分配在内存中的，内存速度相对cpu来说是瓶颈）

基于寄存器的指令集：如x86架构的二地址指令集，缺点是程序直接依赖于硬件寄存器，收到硬件约束，优点是快

**第九章 类加载及执行子系统的案例与实战**

**第四部分 程序编译与代码优化**

**第十章 早期（编译期）优化**

编译期因为不同的编译器可能有不同的说法，这里我们采用前端编译器，将.java文件转变为.class文件的过程称为编译期。

局部变量在常量池中没有CONSTANT\_Fieldref\_info引用，所以没有访问标志的信息，因此将局部变量设置为final是没有意义的

语法糖是指对语言功能没有影响，但是更方便程序员使用，增加程序的可读性，比如泛型、自动拆/装箱、变长参数等，它们在编译阶段还原成简单的基础语法结构，这个过程叫做解语法糖。

C#中List<int>和List<String>就是两个不同的类型，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型称为真实泛型。

Java中的泛型是一颗语法糖，ArrayList<Integer>和ArrayList<String>在编译后替换为原生类型，并在相应位置插入了强制转型代码，对于运行期间来说这两个是同一个类，泛型实现方法是类型擦除，基于这种方法实现的泛型称为伪泛型。

返回值不参与重载的选择，但是在Class文件中只要描述符不是完全一致的两个方法就可以共存

擦除法仅仅在Code属性中的字节码进行了擦除，但是元数据中还是保留了泛型信息，因此通过反射手段能取得参数化数据

自动拆装箱编译后使用了Integer.valueOf()方法和Integer.intValue()方法，包装类的“==”操作在不遇到算术运算的情况下不会自动拆箱，遇到算术运算的时候会自动拆箱来比较“==”两边的数值。并且它们的equals（）方法不会处理数据转型的关系。包装类的equals（）方法和“==”区别很重要。

包装类的equals（）方法先判断类型是否相同（如果参数是基本类型会将参数进行装箱），类型相同再判断值是否相同。

“==”是判断是否是同一个地址，基本类型和包装类型通过“==”比较时会自动拆箱，两个包装类型通过“==”比较时判断是否在同一个地址，也就是是否是同一个对象。尤其注意系统中提供了一个缓存功能，-128~127之间的整数自动装箱为一个Integer时，实际上指向缓存中数值对应的Integer，而超出这个范围的整数自动装箱为Integer时总是新创建一个Integer实例。valueOf的JDK源码如下：

public static Integer valueOf(int i) {

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

因此

Integer a = 2;//体会一下这里的自动装箱，等价于 Integer a=Integer。valueOf(2);

Integer b = 2;
System.out.println(a == b); //输出true，指向同一个实例
Integer a1 = 128;
Integer b1 = 128;
System.out.println(a1 == b1); //输出false，两个不同的实例

foreach循环编译后使用了带有迭代器的for循环，所以要求能被遍历的类实现Iterable接口。

变长参数在调用的时候变成了一个数组类型的参数

Java语言中的条件编译也是语法糖，根据布尔常量值的真假，编译器会把分支中不成立的代码块消除掉

**第十一章 晚期（运行期）优化**

编译运行和解释运行都是基于Class文件的，解释运行是将字节码一行一行代码的翻译为本地机器码执行，编译运行是将字节码按方法为单位翻译为本地机器码执行。

**解释运行**一行一行进行，边翻译边运行，不生成目标程序，缺点是再次运行时要重复解释

**编译运行**以方法为单位，先翻译成本地机器码再执行，编译出来的本地机器码质量比解释运行翻译的高，可以保存起来复用，缺点是如果所有代码都编译会很慢。所以我的理解是对于热点代码才有编译运行的价值。

即时（JIT）编译器在运行时把代码（比如热点代码）编译成与本地平台相关的机器码，提高执行效率

程序需要快速启动时，解释器可以首先发挥作用，因为省略了编译过程（理解为已经生成了Class文件），随着时间推移，编译器逐渐发挥作用，越来越多的代码编译为本地代码，获得更高的执行效率。

内存限制大时，采用解释执行节约内存，反之使用编译执行提升效率

会被即时编译器编译的热点代码主要是两类，一类是被多次调用的方法，一类是被多次执行的循环体（这种编译方式发生在方法执行过程中，因而也称为栈上替换（On Stack Replacement）OSR编译。），在JIT编译方式中，这两种情况都是以整个方法作为编译对象。

判断方法是否为热点有两种方法，一种是基于采样的热点探测，JVM周期性的检查各个线程的栈顶，如果一个方法经常出现在栈顶就认为是热点方法，简单高效但是不够精确，另外一种是基于计数器的热点探测，为方法建立计数器，执行次数超过一定阈值就认为是热点方法，精确但是复杂，而且不能获取方法的调用关系。HotSpot中采用基于计数器的热点探测方法。

如果一段时间内方法调用次数不足以交给即时编译器编译，方法的调用计数器就会减少一半，类似半衰的概念

回边计数器没有计数热度衰减的过程，统计的是方法中循环执行的绝对次数，当回边计数器溢出的时候，会把方法计数器的值也调整到溢出状态，这样下次进入该方法的时候就会进入标准编译过程。

默认设置下，在代码编译器还未完成时仍然采用解释方式运行，编译动作在后台的编译程序中进行。

方法内联就是方法a调用方法b时，在编译时直接将b中的代码复制到a中，而避免真正去调用方法b，这样不用再建立b方法的栈帧以及出入栈操作，优化速度。

公共子表达式消除就是如果某个表达式E计算过了并且没有发生变化，那么再次出现E时就无需再重复计算。

对于虚拟机子系统而言，每次数组元素的读写都带有一次隐含的条件判定是否超出数组边界，大量访问数组时是性能负担，因此虚拟机也会聪明的判定某些时候不需要进行数组边界的检查，这就是数组边界检查消除。

许多激进优化都要预留一个逃生门，当优化不能正确进行时切换到解释执行。

比如在进行虚方法内联时，由于多态性，内联没有稳定保证，这时可以有“类型继承关系分析”等方法来提高内联的安全性，但是逃生门还是需要的。

标量就是无法再分解为更小的数据来表示的数据，Java虚拟机中的原始数据类型都可以称为标量，如可以继续分解的则称为聚合量，比如对象。

逃逸分析的基本行为就是分析对象动态作用域。对象在方法中被定义后，如果被外部方法所引用，就是方法逃逸，被外部线程访问就是线程逃逸。

对于不会逃逸的对象可以进行一些高效的优化，比如栈上分配（对象不会被其他方法访问的话可以直接分配在栈上避免以后垃圾回收等）、同步消除（不会线程逃逸的话就不用同步）、标量替换（对象不会被外部访问到的话可以不创建这个对象而改为直接创建他的若干个被这个方法使用到的成员变量来代替，成员变量可以在栈上）。但是逃逸分析也耗时，不逃逸对象少的话应用这项技术会得不偿失。

**第五部分 高效并发**

**第十二章 Java内存模型与线程**

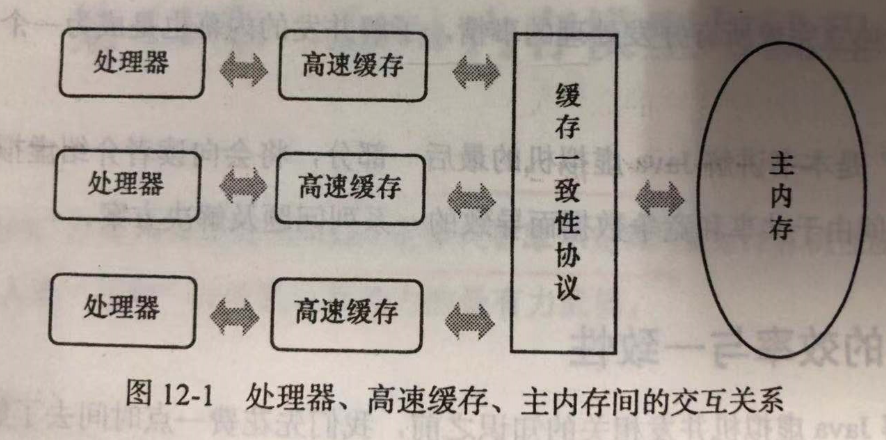
采用多线程并发处理的一个很重要的原因是计算机的计算速度和它的存储以及通信子系统速度差距过大，大量时间花在磁盘IO和网络通信或者数据库访问上，如果不采用多线程并发将会浪费计算机的处理能力在等待这些耗时的操作上。

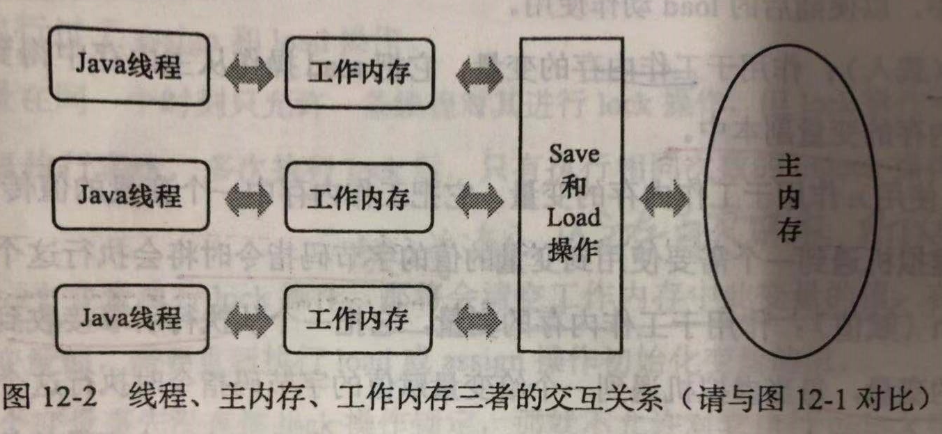
在物理机中，通过高速缓存（Cache）作为内存和处理器之间的缓冲，将运算需要的数据复制到缓存中，运算结束后再从缓存同步回内存中，处理器就不用等待缓慢的内存读写了。

多处理器系统中，每个系统都有自己的高速缓存，但是它们又都共享同一主内存，可能导致同步化问题，因此需要遵循一些缓存一致性协议。

除了高速缓存外，处理器可能会对输入代码进行乱序执行优化。Java虚拟机的JIT编译器中有类似的指令重排优化。

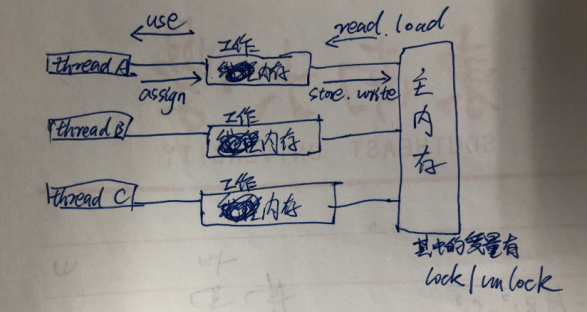
Java内存模型的主要目标是定义程序中各个变量的访问规则。这里指的变量是Java编程中指的变量不同，这里的变量是实例字段、静态字段、构成数组的元素这些会存在竞争问题的数据。





Java内存模型中的工作内存与物理机中的高速缓存类似，线程的工作内存中保存了该线程使用的变量的主内存拷贝。线程对变量的所有操作都在工作内存中完成。

工作内存与主内存的划分与Java内存区域中的Java堆、栈、方法区等不是一个层次的内存划分，二者基本是没有关系的。勉强对应的话，主内存对应于Java堆中的对象实例数据部分，工作内存对应于虚拟机栈中的部分区域。



以上8种操作有一定的规则：

read和load，store和write不能单独出现

assign操作是当执行引擎出现变量赋值的字节码指令时执行，assign操作后变量在工作内存中的改变必须被同步回主内存。

变量只能在主内存中“诞生”，不允许工作内存中直接使用一个未被初始化（load或assign）的变量。

对变量进行lock操作时先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或者assign操作来初始化值。

对一个变量进行unlock时必须先把此变量同步回主内存中（store、write）。

volatile关键字是最轻量级的同步机制，变量定义为volatile之后保证了对所有线程的可见性（当某个线程改变了变量的值时新值对于其他线程是可以立即得知的）。使用volatile变量会禁止指令重排优化。

单例模式有三个要点：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

　　单例模式是结构最简单的设计模式一，在它的核心结构中只包含一个被称为单例类的特殊类。

对于volatile变量，线程要use才在use前read和load变量（保证自己看见这个别的线程对volatile变量的改变），线程assign动作后执行store和write动作（保证别的线程看见这个volatile变量的改变）

volatile变量性能优于锁的理由在于读变量操作几乎与读普通变量一致，写操作要在代码中插入内存屏蔽指令来禁止指令重排优化。

**原子性**：原子操作指的是不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch（切换到另一个线程）。Java内存模型中直接保证的原子性操作包括read、load、assign、use、store、write。lock和unlock提供了更大规模的原子性保证。synchronized块之间的操作也具有原子性。

**可见性**：可见性指的是一个线程修改了共享变量的值后别的变量能立刻得知这个修改。Java内存模型通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式实现可见性。volatile变量保证新值能立刻同步到主内存，以及每次使用前立刻从主内存刷新，保证了多线程操作时变量的可见性，而普通变量无法保证这一点。除了volatile外，synchronized（unlock之前必须把变量同步回主内存中这条规则实现）和final关键字也可以保证可见性。

**有序性**：在本线程中观察，所有操作都是有序的（线程内表现为串行的语义），在一个线程中观察另一个线程，所有操作都是无序的（指令重排序现象和工作内存和主内存同步延迟现象）。

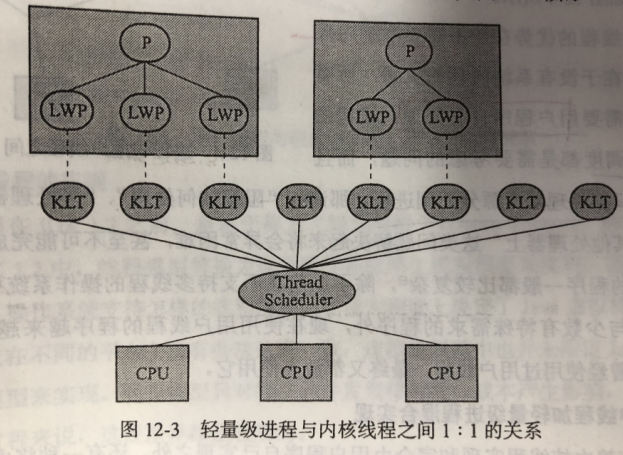
Java中的**先行发生原则**指的就是两项操作的偏序关系，比如有线程启动规则（线程的start（）方法先行发生于线程中的每一个动作）、传递性（a先行发生于b操作，b先行发生于c，则a先行发生于c操作）、程序次序规则（一个线程内在控制流前的操作先行发生于控制流后的操作）等多项规则。无法由这些规则推导出来的操作就没有顺序性保证，虚拟机可以进行任意的指令重排序。

先行发生原则和时间先后顺序基本没有太大的关系！！（比如一个线程中int i=1;int j=2则按照程序次序规则i比j先行发生，但是可能j比i先被处理）！衡量并发安全问题的时候以先行发生原则为准，不要受到时间顺序的干扰。

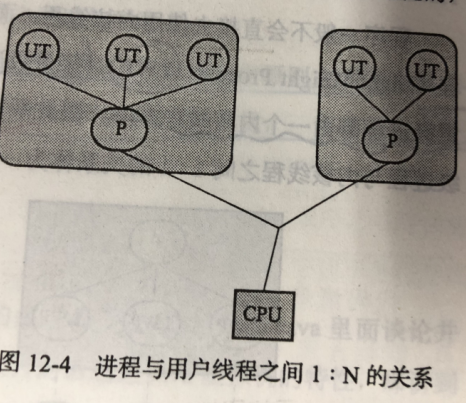
线程是CPU调度的基本单位。各个线程可以共享进程资源（内存地址、文件IO等），又可以独立调度。

**线程的实现主要有3种：**

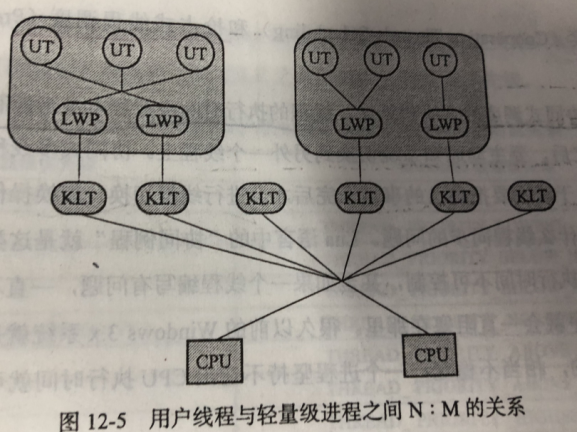
1.使用内核线程实现。一般采用内核线程（Kernel level thread）的高级接口——轻量级进程（light-weight process）来实现，KLT和LWP是1对1的关系，各种线程操作都需要系统调用，要去用户态和内核态中切换



2.使用用户线程（user thread）实现，线程的建立、同步、销毁和调度等都是完全在用户态中完成的，需要用户程序自己处理，很复杂，现在基本抛弃了。



3.使用用户线程加轻量级进程混合实现。用户线程建立在用户空间中，操作廉价，同时还有系统提供的轻量级进程作为与内核的桥梁。



目前Java虚拟机线程使用哪种模型和具体的平台有关，Linux和Windows中都是使用一对一的线程模型实现的。

**线程调度**是指系统为线程分配处理器使用权的过程。线程调度有两种：

1.协同式调度，线程自己控制自己的执行时间，好处是不会有线程同步的问题，坏处是一个线程出问题可能整个系统阻塞在那里崩溃。

2.抢占式调度，系统来分配每个线程的执行时间，Java就是用抢占式调度。

在抢占式调度中可以给线程添加优先级，来让调度过程中线程本身有一丢丢话语权。Java线程的优先级最后会映射到系统原生线程的优先级上去。因此单纯依靠优先级来对线程调度不靠谱。

Java语言定义了线程的5种状态：

1.新建

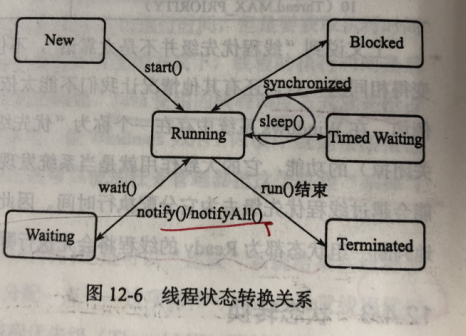
2.运行：包括了running和ready，表明线程可以运行但是不一定在运行

3.1无限期等待（waiting）：等待被其他线程显式唤醒，包括没有设置Timeout参数的Object.wait（）方法和Thread.join（）方法以及LockSupport.park()方法

3.2.限期等待（timed waiting）：在一定时间之后会被系统自动唤醒，包括Thread.sleep（）方法，设置了Timeout参数的Object.wait（）方法和Thread.join（）方法,LockSupport.parkNanos（）方法LockSupport.parkUntil（）方法

4.阻塞：阻塞和等待状态的区别在于阻塞在等待一个排他锁，在程序等待进入同步区域时线程将进入这种状态

5.结束



**第十三章 线程安全与锁优化**

面向过程的编程思想是站在计算机的角度去抽象问题和解决问题，把数据当做问题空间的客体，程序代码则用来处理这些数据。

面向对象的编程思想是站在现实世界的角度去抽象和解决问题，把数据和行为都看做对象的一部分，这样可以让程序员能以符合现实世界的思维方式来编写和组织程序。

线程安全的代码应该封装了所有必要的正确性手段，令调用者无须担心多线程的问题，更无须自己采取任何措施来保证多线程的正确调用。就是要让调用者省心！

我们这里讨论的线程安全都限定于多个线程之间存在共享数据访问这个前提。

**线程安全**按由强到弱分为5个等级：

1.不可变。不可变的对象一定是线程安全的，例如由final关键字修饰，在Java API中不可变的类型包括String和枚举类、java.lang.Number的部分子类

2.绝对线程安全。Java API中标注自己是线程安全的类大多数都不是绝对的线程安全。

3.相对线程安全。需要保证这个对象的单独的操作是线程安全的，但是对于特定顺序的连续调用就可能需要在调用端使用额外的同步手段来保证调用的正确性。大部分线程安全类都是这种类型，比如Vector、HashTable、Collections的synchronizedCollection（）方法包装的集合等。

4.线程兼容。对象本身不是线程安全的，需要调用端使用正确的同步手段，Java API中的大部分的类都是属于线程兼容的，比如和Vector、HashTable对应的ArrayList和HashMap类

5.线程对立。同步措施都不能保证在多线程下的并发使用，比如System.setIn()方法等。

同步是指在多个线程中并发访问共享数据时，保证共享数据在一个时刻只被一个（在使用信号量的时候可以是一些）线程使用

**线程安全的实现方法**：

1.互斥同步：也称为阻塞同步，是悲观的并发策略，总认为不去做同步措施就会出现问题。常用的比如synchronized关键字和ReentrantLock锁。**synchronized**关键字需要一个reference类型参数来指明要锁定和解锁的对象，如果没有明确指定的话就根据synchronized修饰的是实例方法还是类方法去取对应的对象实例或Class对象来作为锁对象。synchronized同步块对同一条线程是可重入的，不会出现自己把自己锁死的情况，synchronized会在同步块的前后生成monitorenter和monitorexit字节码指令。monitorenter尝试获取对象锁的时候，如果对象没有被锁定或者当前线程已经获得了对象锁就把锁的计数器加1，如果获取对象锁失败，当前线程就要阻塞等待。monitorexit把锁的计数器减1，减到0时解锁。Java线程是映射到操作系统的线程上去的，所以要阻塞或者唤醒一个线程都要操作系统完成，这就需要从用户态转到核心态中，耗费很多的处理器时间，因此synchronized是重量级的操作。synchronized表现为原生语法层面的互斥锁，而**ReentrantLock**表现为API层面的互斥锁（lock（）和unlock（）方法配合try/catch块完成）。ReentrantLock增加了等待可中断、可实现公平锁、锁可以绑定多个条件。

2.非阻塞同步：基于冲突检测的乐观并发策略，先操作再看如果产生了冲突就采取补偿措施（比如不断重试），这种策略不需要把线程挂起，因此称为非阻塞同步。

3.无同步方案：比如可重入代码（代码在任意时刻中断转而执行别的代码，然后再转为这个代码，代码不会出错。判断是可重入代码就看给定相同的数据，是不是都能返回同样的结果，也就是结果可预测）和线程本地存储（把需要共享数据的代码放到一个线程中去）。

**锁优化**：

1.自旋锁：阻塞和唤醒线程要在内核态中完成，对性能影响大，通常共享数据的锁定状态很短，所以当线程请求已经被别的线程锁上的数据时，可以让线程再“稍等一等”，执行一个忙循环，不放弃处理器的执行时间，这就是线程的自旋，避免的线程切换的开销。

2.锁消除：由于Java很多API都有锁操作，而这些锁操作可能是没有必要的，因此可以基于逃逸分析判断某些加锁是不是不必要的。

3.锁粗化：通常推荐将同步块的作用范围限制的尽量小，这样等待锁的线程也能尽快拿到锁，但是如果一串代码频繁的加锁和解锁同一个对象，虚拟机就会把锁的作用范围扩展到这串代码中，减少加锁和解锁操作。

4.轻量级锁：在堆中对象的对象头中存储有hashcode、GC分代年龄和存储锁标志位（有未锁定、轻量级锁定、重量级锁定、可偏向等多种状态标志）在没有多线程竞争的前提下减少传统的重量级锁使用，但是如果有多线程竞争，就要膨胀为重量级锁，则轻量级锁比重量级锁更慢。

5.偏向锁：也是为了在无竞争情况下提高运行性能。它的锁会偏向于第一个获得它的线程，这个锁接下来如果没有被其他线程获得，则持有偏向锁的线程将永远不需要同步。当另外的线程尝试获得锁时，偏向模式就结束了。