链滴

《深入理解 Java 虚拟机》读书笔记：Java 内存模型与线程

作者：jingqueyimu

原文链接：https://ld246.com/article/1585755306099

来源网站：链滴

正文

由于计算机的处理器运算速度与它的存储和通信子系统速度的差距太大了，大量的时间都花费在磁盘 IO、网络通信或者数据库访问上，导致处理器在大部分时间里都处于等待其他资源的状态。因此，为充分利用计算机的处理器运算能力，现代计算机操作系统采用了多任务处理的方式，即让计算机并发理多个任务。

对于计算量相同的任务，程序线程并发协调得越有条不紊，效率自然就会越高；反之，线程之间频繁塞甚至死锁，将会大大降低程序的并发能力。

一、硬件的效率与一致性

1、高速缓存

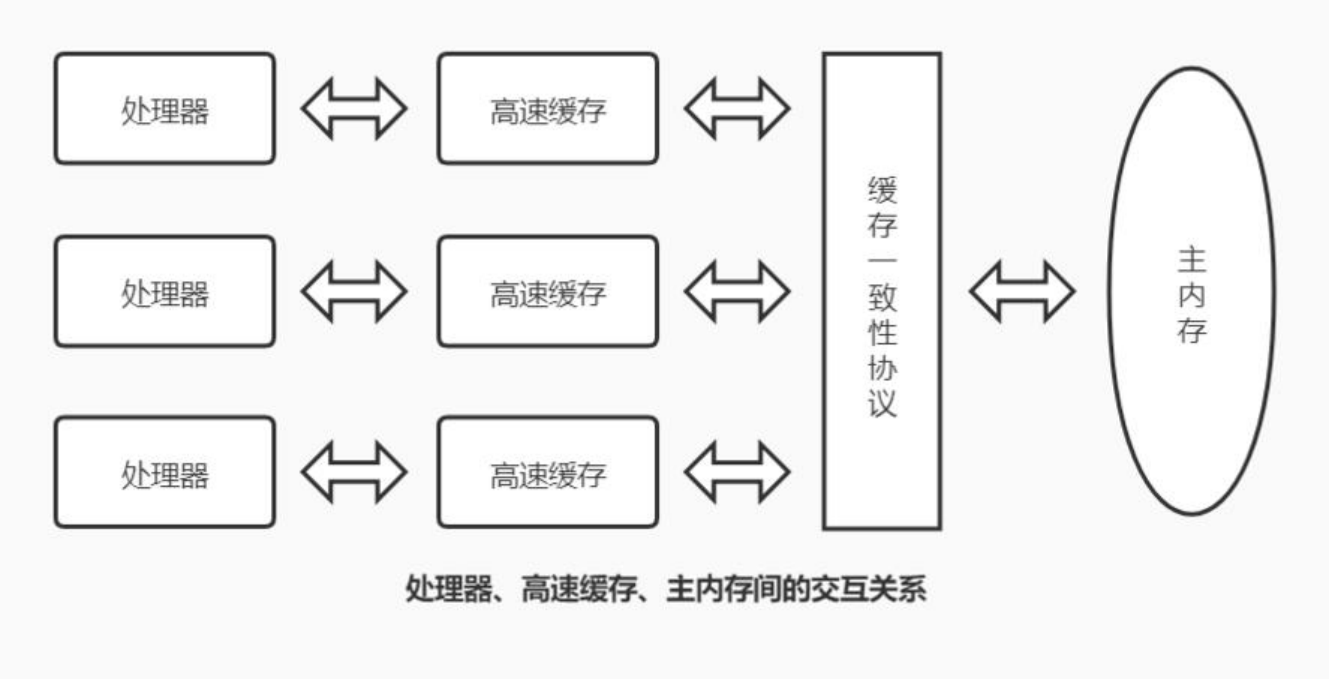
由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统加入了一层读速度尽可能接近处理器运算速度的高速缓存来作为内存与处理器之间的缓冲：将运算需要使用的数据制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存中，这样处理器就无须等待缓慢内存读写了。

2、缓存一致性

基于高速缓存的存储交互解决了处理器与内存的速度矛盾，但也引入了一个新的问题：缓存一致性。

在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存。当多个处理器的运任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致。为了解决一致性的问题，需要个处理器访问缓存时遵循一些协议，在读写时根据协议来进行操作，比如 MSI、MESI 等协议。

处理器、高速缓存、主内存间的交互关系：



处理器、高速缓存、主内存间的交互关系

3、乱序执行

除了增加高速缓存之外，为了使处理器内部的运算单元能尽量被充分利用，处理器可能会对输入代码行乱序执行优化。处理器会在计算之后将乱序执行的结果重组，保证该结果与顺序执行的结果一致。

二、Java 内存模型

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，即在虚拟机中将变量存储到内存和从内中取出变量这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素，但不包括部变量与方法参数，因为后者是线程私有的，不会被共享，不存在竞争问题。

1、主内存与工作内存

Java 内存模型规定了所有的变量都存储在主内存中。每条线程还有自己的工作内存，线程的工作内存保存了被该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完。

从定义上来看，主内存主要对应于 Java 堆中的对象实例数据部分，而工作内存则对应于虚拟机栈中部分区域。

从更低层次上说，主内存直接对应于物理硬件的内存，而为了获取更好的运行速度，虚拟机（甚至是件系统本身的优化措施）可能会让工作内存优先存储于寄存器和高速缓存中，因为程序运行时主要访读写的是工作内存。

2、内存间交互操作

关于主内存与工作内存之间的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存步回主内存之类的实现细节，Java 内存模型中定义了 8 种操作来完成，虚拟机必须保证每一种操作是原子的、不可再分的。

- **lock（锁定）**：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
- **unlock（解锁）**：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才以被其他线程锁定。
- **read（读取）**：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便后的 load 动作使用。
- **load（载入）**：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的量副本中。
- **use（使用）**：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。
- **assign（赋值）**：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- **store（存储）**：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随的 write 操作使用。
- **write（写入）**：作用于主内存的变量，它把 store 操作从工作内存中得到的变量的值放入主内存变量中。

3、对于 volatile 型变量的特殊规则

volatile 的作用：

● 保证变量对所有线程的可见性。即当一条线程修改了某个变量的值，新值对于其他线程来说是可以即得知的。

● 禁止指令重排序优化。

volatile 变量只能保证可见性，不能保证原子性。在以下运算场景中，仍然要通过加锁（使用 synchronized 或 java.util.concurrent 中的原子类）来保证原子性：

● 运算结果依赖于变量的当前值，并且其他线程可能会修改变量的值。

● 变量需要与其他的状态变量共同参与不变约束。

对 volatile 变量的特殊规则：

● 某个线程对 volatile 变量的 read、load、use 操作必须连续一起出现。这条规则要求在工作内存，每次使用变量前都必须先从主内存刷新最新的值，用于保证能看见其他线程对变量所做的修改后的。

● 某个线程对 volatile 变量的 assign、store、write 操作必须连续一起出现。这条规则要求在工作存中，每次修改变量后都必须立刻同步回主内存中，用于保证其他线程可以看到自己对变量所做的修。

● 如果线程 A 的 use、assign 操作先于线程 B，那么线程 A 的 read、write 也必须先于 线程 B。这规则要求 volatile 变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同。

4、对于 long 和 double 型变量的特殊规则

Java 内存模型允许虚拟机将没有被 volatile 修饰的 64 位数据（long 和 double）的读写操作划分为次 32 位的操作来进行，即允许虚拟机实现可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性，这就是所谓的 long 和 double 的非原子性协定。

目前各平台下的商用虚拟机几乎都把 64 位数据的读写操作实现为具有原子性的操作，因此在编写代时一般不需要把 long 和 double 变量专门声明为 volatile。

5、原子性、可见性与有序性

Java 内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这 3 个特征来建立的。

（1）原子性

● 基本数据类型的访问读写具备原子性（不考虑 long、double 的非原子性协定）： Java 内存模型接保证了 read、load、assign、use、store 和 write 操作的原子性。

● synchronized 代码块之间的操作具备原子性：底层通过 lock 和 unlock 操作实现。

（2）可见性

可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。

Java 内存模型是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内存作为传递媒介的方式来实现可见性的。

volatile、synchronized、final 关键字都能实现可见性。

（3）有序性

Java 程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在个线程中观察另一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行的语义”，后半是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。

Java 语言提供了 volatile、synchronized 关键字来保证线程之间操作的有序性。

6、先行发生原则

先行发生是 Java 内存模型中定义的两项操作之间的偏序关系，如果说操作 A 先行发生于操作 B，其就是说在发生操作 B 之前，操作 A 产生的影响能被操作 B 观察到。“影响”包括修改了内存中共享量的值、发送了消息、调用了方法等。它是判断数据是否存在竞争、线程是否安全的主要依据。

Java内存模型的先行发生关系：

- **程序次序规则**：在一个线程内，按照程序代码顺序，书写在前的操作先行发生于书写在后的操作。确地说，是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- **管程锁定规则**：一个 unlock 操作先行发生于后面（时间上的先后顺序）对同一个锁的 lock 操作。
- **volatile 变量规则**：对一个 volatile 变量的写操作先行发生于后面（时间上的先后顺序）对这个变的读操作。
- **线程启动规则**：Thread 对象的 start() 方法先行发生于此线程的每一个动作。
- **线程终止规则**：线程中的所有操作都先行发生于对此线程的终止检测，可以通过 Thread.isAlive() 法检测到线程是否已经终止执行。
- **线程中断规则**：对线程 interrupt() 方法的调用先行发生于被中断线程检测到中断事件的发生，可通过 Thread.interrupted() 方法检测到是否有中断发生。
- **对象终结规则**：一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize() 方法的开。
- **传递性**：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么可以得出操作 A 先行发于操作 C。

如果两个操作之间的关系不满足以上规则，并且无法从以上规则推导出来，那么它们就没有顺序性保，虚拟机可以对它们随意地进行重排序。

三、Java 与线程

1、线程的实现

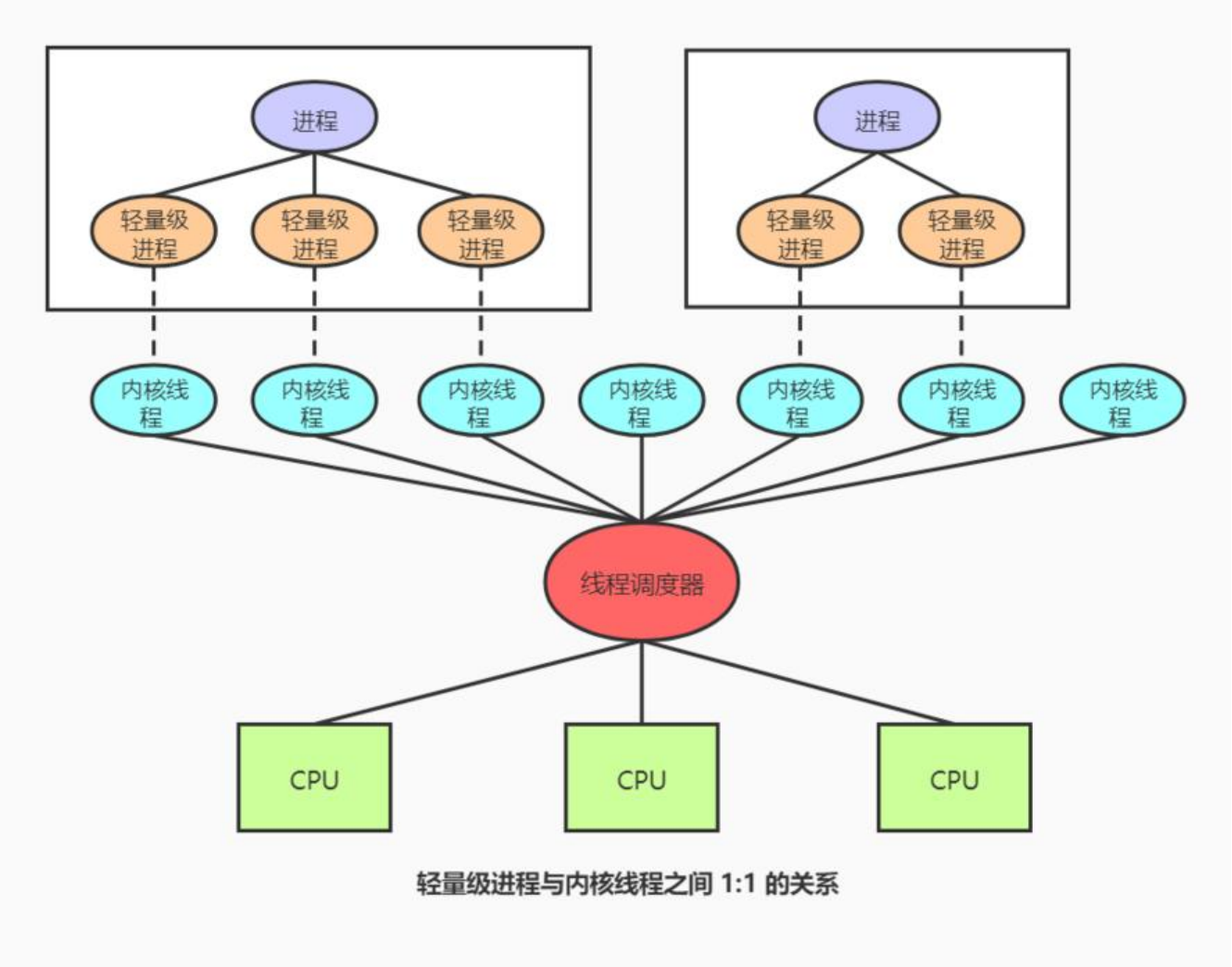
主流的操作系统都提供了线程实现，Java 语言则提供了在不同硬件和操作系统平台下对线程操作的统处理，每个已经执行 start() 且还未结束的 java.lang.Thread 类的实例就代表了一个线程。

（1）使用内核线程实现

内核线程（Kernel-Level Thread，KLT）就是直接由操作系统内核（Kernel）支持的线程，这种线程内核来完成线程切换，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器。

每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内就叫做多线程内核。

程序一般不会直接使用内核线程，而是使用内核线程的一种高级接口——轻量级进程（Light Weight rocess，LWP），轻量级进程就是通常意义上所讲的线程，每个轻量级进程都由一个内核线程支持。种轻量级进程与内核线程之间 1:1 的关系称为**一对一的线程模型**。



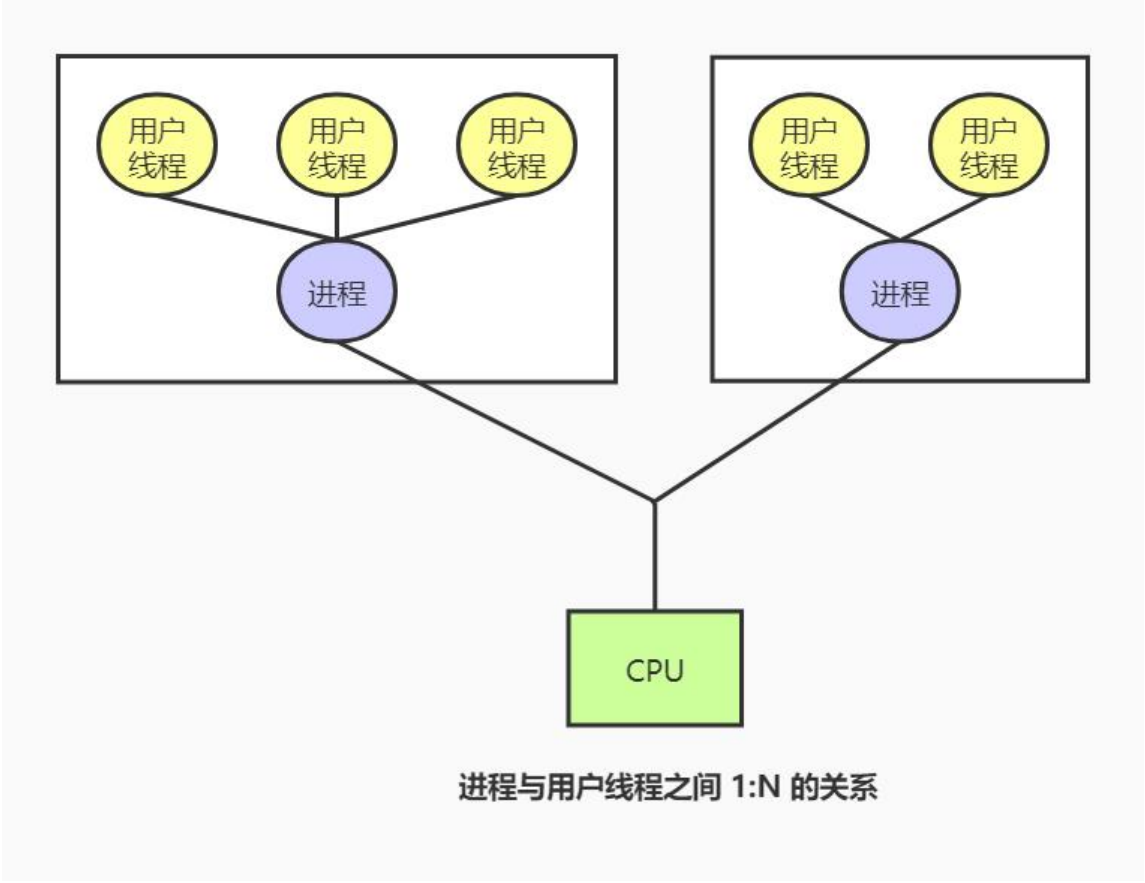
轻量级进程与内核线程之间 1:1 的关系

轻量级进程的局限性：

- 由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而统调用的代价相对较高，需要在用户态和内核态中来回切换。
- 每个轻量级进程都需要有一个内核线程的支持，会消耗一定的内核资源（如内核线程的栈空间），此一个系统支持轻量级进程的数量是有限的。

（2）使用用户线程实现

用户线程（User Thread，UT）完全建立在用户空间的线程库上，系统内核不能感知线程的存在。用线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。因此操作快速且低消耗，可以支持规模更大的线程数量。这种进程与用户线程之间 1:N 的关系称为**一对多的线程模型**。

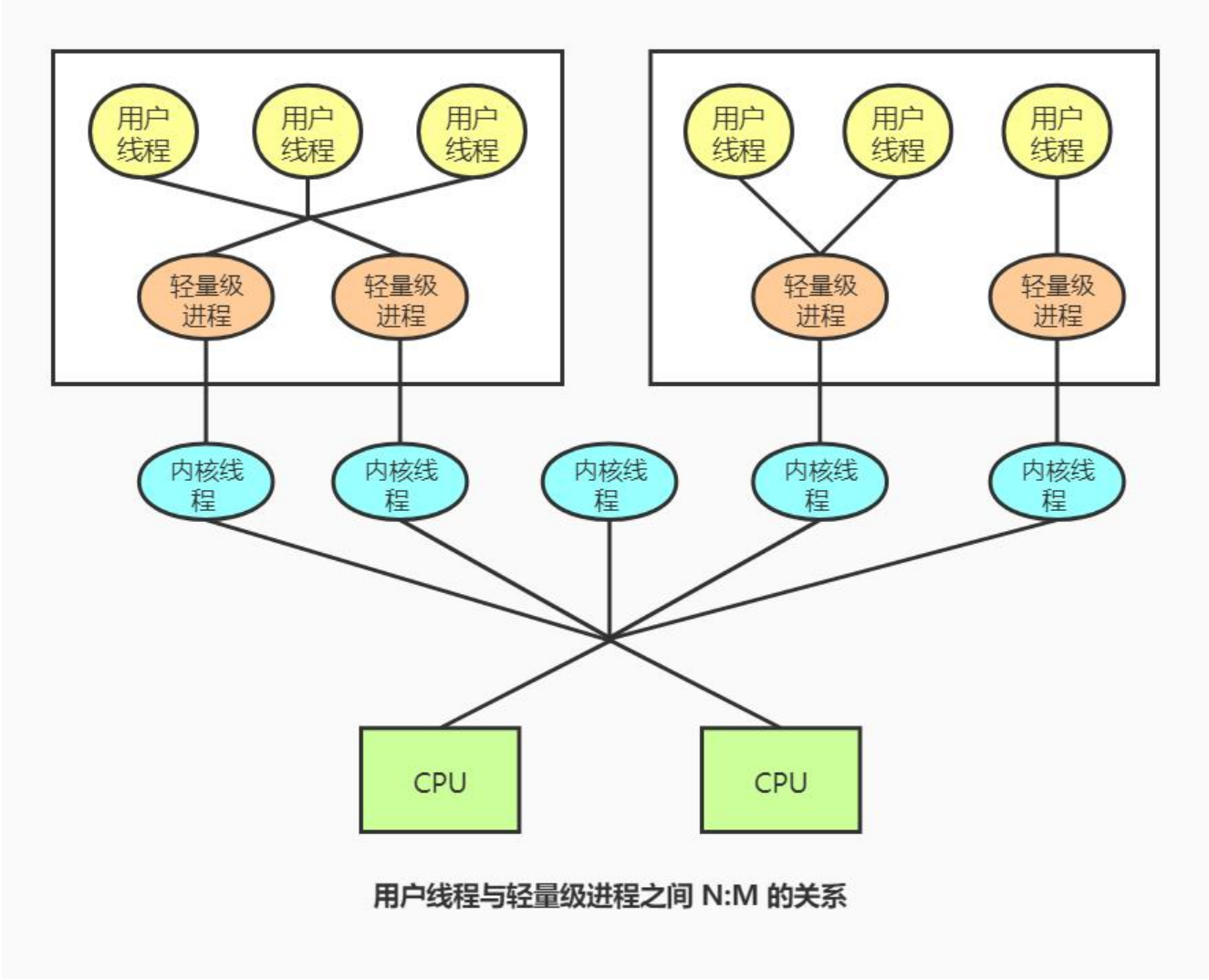


进程与用户线程之间 1:N 的关系

使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都要用户程序自己处理。因此使用用户线程实现的程序一般都比较复杂。

（3）使用用户线程加轻量级进程混合实现

混合实现时，用户线程还是完全建立在用户空间中，而操作系统提供支持的轻量级进程则作为用户线和内核线程之间的桥梁。在这种混合模式中，用户线程与轻量级进程的数量比是不定的，即为 N:M 关系，这种就是**多对多的线程模型。**



用户线程与轻量级进程之间 N:M 的关系

混合实现的好处：

- 用户线程的操作依然廉价，并且可以支持大规模的用户线程并发。
- 可以使用内核提供的线程调度功能及处理器映射。
- 由于用户线程的系统调用要通过轻量级进程来完成，因此大大降低了整个进程被完全阻塞的风险。

2、Java 线程调度

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式有两种：协同式线程调度和抢占式程调度。

（1）协同式线程调度

线程的执行时间由线程本身来控制，线程执行完之后，主动通知系统切换到另外一个线程上。

协同式线程调度最大的好处是实现简单，而且切换线程的操作对线程自己是可知的，所以没有什么线同步的问题。它的坏处就是线程执行时间不可控，如果一个线程编写有问题，一直不告知系统进行线切换，那么程序就会一直阻塞在那里。

（2）抢占式线程调度

每个线程由系统来分配执行时间，线程的切换不由线程本身来决定。

使用抢占式线程调度时，线程的执行时间是系统可控的，不会有一个线程导致整个进程阻塞的问题。

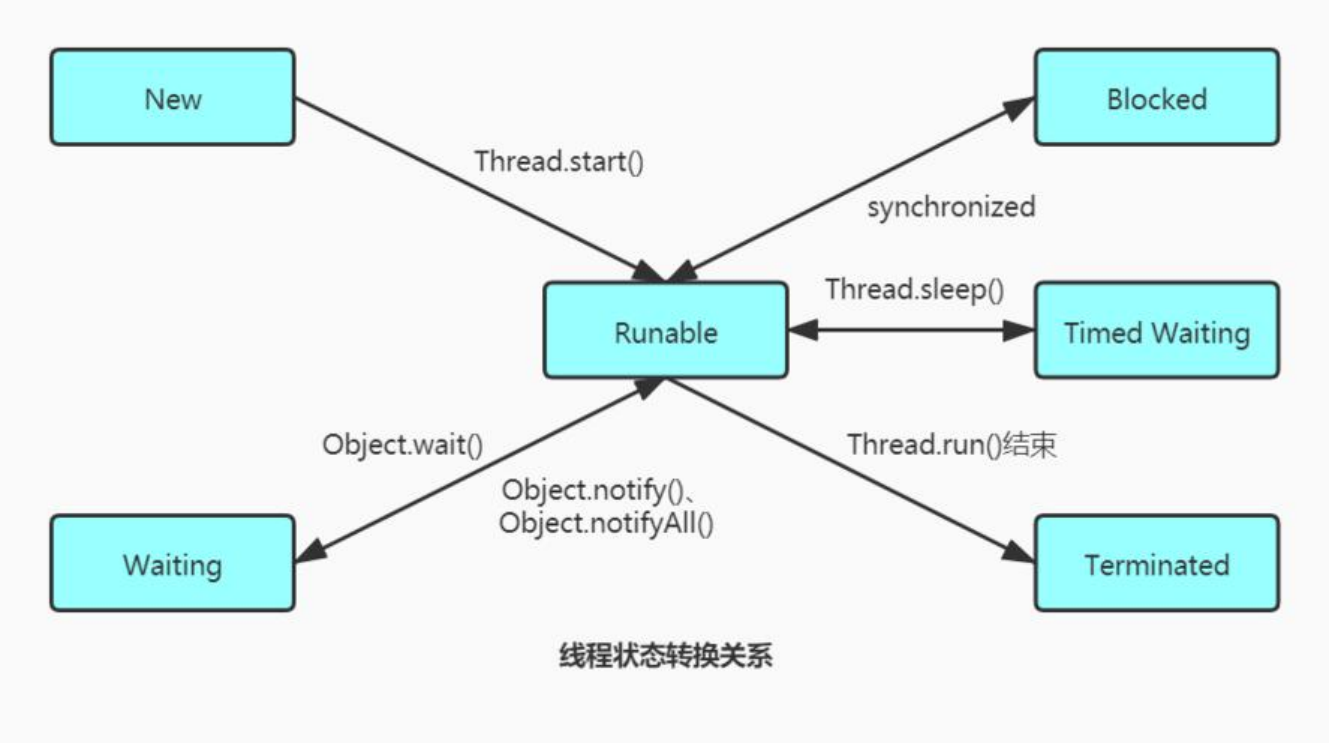
Java 使用的线程调度方式就是抢占式调度。

3、线程状态

（1）6 种线程状态

- **新建（New）**：创建后尚未启动的线程处于这种状态。
- **运行（Runable）**：包括了操作系统线程状态中的 Running 和 Ready，处于此状态的线程有可能在执行，也有可能正在等待着 CPU 为它分配执行时间。
- **无限期等待（Waiting）**：不会被分配 CPU 执行时间，等待着被其他线程显式地唤醒。
- **限期等待（Timed Waiting）**：不会被分配 CPU 执行时间，无须等待被其他线程显式地唤醒，在定时间之后会由系统自动唤醒。
- **阻塞（Blocked）**：线程被阻塞了，在等待着获取到一个排他锁。在程序等待进入同步区域的时候线程将进入这种状态。
- **结束（Terminated）**：已终止线程的线程状态，线程已经结束执行。

（2）线程状态转换



线程状态转换关系