# java内存模型与线程

## java内存模型

### 主存与工作内存

Java 内存模型主要目标：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存中取出变量这样的底层细节。即与平台无关的缓存一致性协议。

1. Java 内存模型规定所有变量都存储在主存（Main Memory）中（虚拟机内存的一部分）。
2. 每条线程还有自己的工作内存（Working Memory），线程的工作内存保存了被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。
3. 不同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存来完成。
4. 这里的主内存类似于硬件上的内存，工作内存类似于CPU上的高速缓存。


图 线程/主内存/工作内存三者的交互关系



### 内存间交互操作

主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了以下 8 种操作来完成：

1. Lock（锁定）：作用于主内存的变量，将主内存该变量标记成当前线程私有的，其他线程无法访问它把一个变量标识为一条线程独占的状态。
2. Unlock（解锁）：作用于主内存的变量，把一个处于锁定状态的变量释放出来，才能被其他线程锁定。
3. Read（读取）：作用于主内存的变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的 load 动作使用。

4. Load（加载）：作用于工作内存中的变量，把 read 操作从内存中得到的变量值放入工作内存的变量副本中。
5. Use（使用）：作用于工作内存中的变量，把工作内存中一个变量的值传递给执行引擎，每当虚拟机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。
6. Assgin（赋值）：作用于工作内存中的变量，把一个从执行引擎接收到的值赋值给工作内存的变量，每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。
7. Store（存储）：作用于工作内存中的变量，把工作内存中一个变量的值传递到主内存中，以便随后的 write 操作使用。
8. Write（写入）：作用于主内存中的变量，把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

如果把一个变量从主内存复制到工作内存，按顺序执行 read 和 load 操作；如果把变量从工作内存同步回主内存，按顺序执行 store 和 write 操作。

此外java虚拟机定义了关于执行这些操作必须满足的顺序规则，但是过于复杂，直接关注这些规则的一个等效判断原则——**先行发生原则**。

## 对于volatile型变量的特殊规则

1. 在工作内存中，每次使用 V 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 V 修改后的值。
2. 在工作内存中，每次修改 V 后都必须立刻同步回主内存中，用于保证其他线程看到自己对 V 的修改。
3. 这条规则要求 volatile 修饰的变量不会被指令的重排序优化，保证代码的执行顺序与程序的顺序相同。

**保证volatile型变量对所有线程的可见性。**“可见性” 指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。volatile 变量在各个线程中的工作内存中不存在一致性问题（在各个线程的工作内存中 volatile 变量也可以存在不一致的情况，但由于每次使用之前都要先刷新，执行引擎看不到不一致的情况，因此可以认为不存在不一致问题），但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的。

由于 volatile 变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁（使用 synchronized 或 java.util.concurrent 中的原子类）来保证原子性：

1. 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。
2. 变量不需要与其他的状态变量共同参与不变约束。

**volatile 变量的第二个语义是禁止指令重排序优化**
指令重排序：重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。但是同一个线程内，方法的执行过程中无法感知到重排序的影响，这就是所谓的线程内表现为串行的语义。

volatile禁止重排序的规则：

1. 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
2. 在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。
   即执行到volatile变量时，其前面的所有语句都执行完，后面所有语句都未执行。
   且前面语句的结果对volatile变量及其后面语句可见。

```
class Singleton{
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

需要volatile关键字的原因是，在并发情况下，如果没有volatile关键字，在第5行会出现问题。instance = new TestInstance()可以分解为3行伪代码

```
a. memory = allocate() //分配内存

b. ctorInstanc(memory) //初始化对象

c. instance = memory //设置instance指向刚分配的地址
```

上面的代码在编译运行时，可能会出现重排序从a-b-c排序为a-c-b。在多线程的情况下会出现以下问题。当线程A在执行第5行代码时，B线程进来执行到第2行代码。假设此时A执行的过程中发生了指令重排序，即先执行了a和c，没有执行b。那么由于A线程执行了c导致instance指向了一段地址，所以B线程判断instance不为null，会直接跳到第6行并返回一个未初始化的对象。
volatile会禁止这种重排序，从而避免问题。

## 对 long 和 double 型变量的特殊规则

允许虚拟机将没有被 volatile 修饰的 64 位数据类型（long 和 double）的读取操作划分为两次 32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性，就点就是 long 和 double 的非原子协定（Nonatomic Treatment of double and long Variables）。
如果多个线程共享一个为声明为 volatile 的 long 或 double 类型变量，并同时对他们进行读取和修改操作，那么有些线程可能会读取到一个即非原值，也不是其他线程修改值得代表了 “半个变量” 的数值。
不过这种读取带 “半个变量” 的情况非常罕见（在目前商用虚拟机中不会出现），因为 Java 内存模型虽然允许虚拟机不把 long 和 double 变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还 “强烈建议” 虚拟机这样实现。

## 原子性、可见性和有序性

### 原子性

原子性（Atomicity）：由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store 和 write，我们大致可以认为基本数据类型的访问具备原子性（long 和 double 例外）。

如果应用场景需要一个更大范围的原子性保证，Java 内存模型还提供了 lock 和 unlock 操作来满足需求，尽管虚拟机未把 lock 和 unlock 操作直接开放给用户，但是却提供了更高层次的字节码指令 monitorenter 和 monitorexit 来隐式地使用这两个操作，这两个字节码指令反应到 Java 代码中就是同步块——synchronized 关键字，因此在 synchronized 块之间的操作也具备原子性。

### 可见性

可见性（Visibility）：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。
除了 volatile，Java 还有两个关键字能实现可见性，synchronized 和 final。

1. 同步块的可见性是由 “对一个变量执行 unlock 操作之前，必须把此变量同步回主内存中（执行 store 和 write 操作）” 这条规则获得的。
2. 而 final 关键字的可见性是指：被 final 修饰的字段在构造器中一旦被初始化完成，并且构造器没有把 “this” 的引用传递出去（this 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到 “初始化了一半” 的对象），那么其他线程中就能看见 final 字段的值。

### 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；如果在一个线程中观察另外一个线程，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”（Within-Thread As-if-Serial Semantics），后半句是指“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 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 关键字本身就包含了禁止指令重排序的语义，而 synchronized 则是由 “一个变量在同一时刻只允许一条线程对其

进行 lock 操作” 这条规则获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

有序性是指操作的可见性。

## 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，如果操作 A 先行发生于操作 B，其实就是说在发生操作 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变量的值 / 发送了消息 / 调用了方法等。

下面是 Java 内存模型下一些 “天然的” 先行发生关系，无须任何同步器协助就已经存在，可直接在编码中使用。 如果两个操作之间的关系不在此列，并且无法从下列规则推倒出来，它们就没有顺序性的保障，虚拟机可以对它们进行随意地重排序。

1. 程序次序规则（Program Order Rule）：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地来说应该是控制流顺序而不是程序代码顺序，因为要考虑分支 / 循环结构。
2. 管程锁定规则（Monitor Lock Rule）：一个 unlock 操作先行发生于后面对同一锁的 lock 操作。这里必须强调的是同一锁，而 “后面” 是指时间上的先后顺序。
3. volatile 变量规则（Volatile Variable Rule）：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的 “后面” 是指时间上的先后顺序。
4. 线程启动规则（Thread Start Rule）：Thread 对象的 start() 方法先行发生于此线程的每一个动作。
5. 线程终止规则（Thread Termination Rule）：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join() 方法结束 / Thread.isAlive() 的返回值等手段检测到县城已经终止执行。
6. 线程中断规则（Thread Interruption Rule）：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted() 方法检测到是否有中断发生。
7. 对象终结规则（Finalizer Rule）：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() 方法的开始。
8. 传递性（Transitivity）：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C。
   时间上的先后顺序与先行发生原则之间基本没有太大的关系，所以我们衡量并发安全问题时不要受时间顺序的干扰，一切必须以先行发生原则为准。

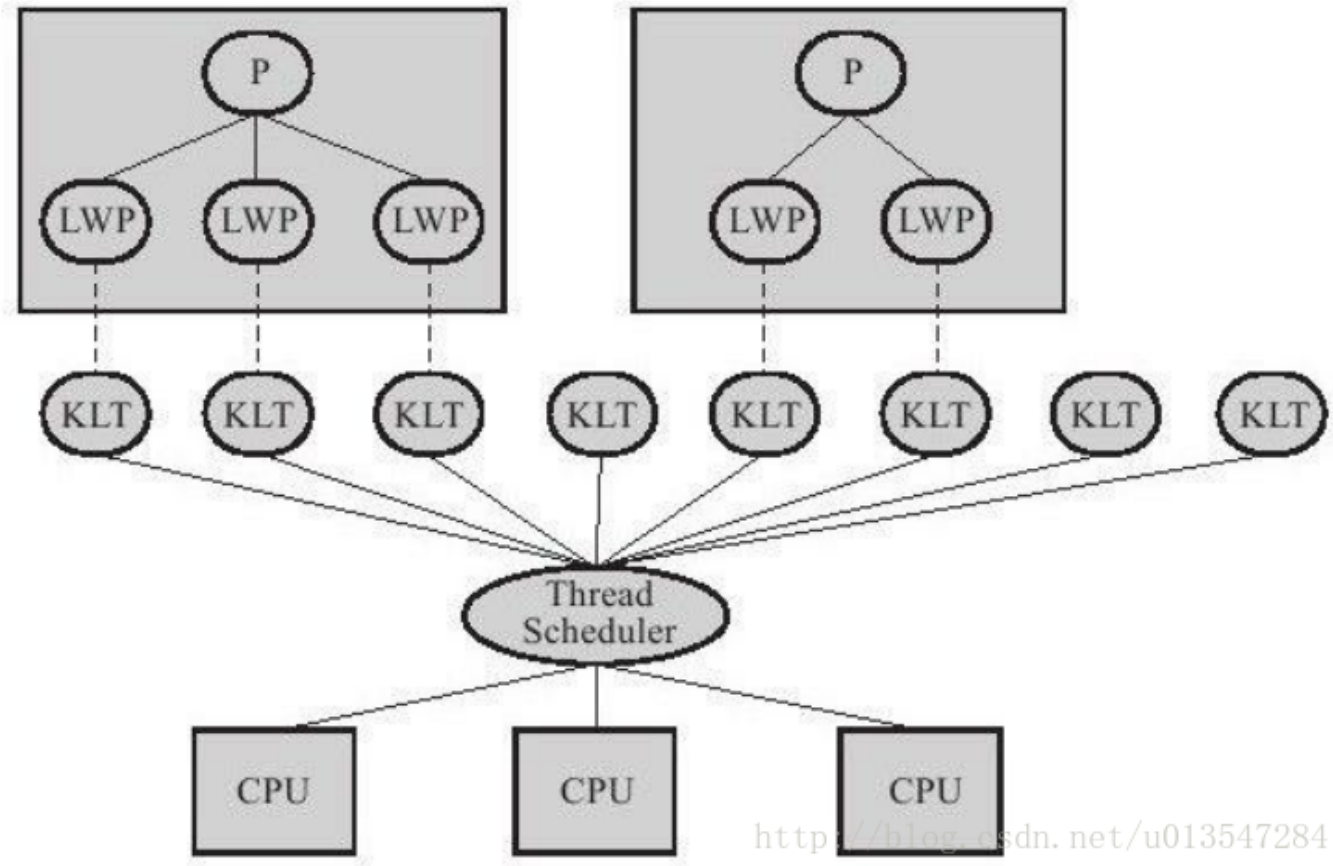
# java与线程

## 线程的实现

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，下称内核）支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器（Scheduler）对线程进行调度，并负责将线程的任务映射到各个处理器上。

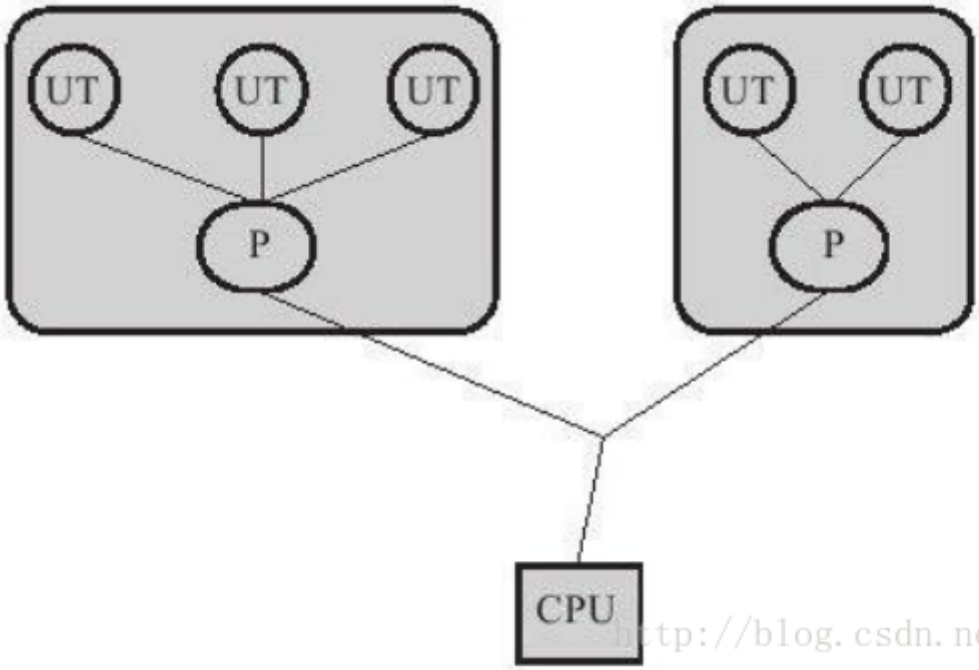
   程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process,LWP），轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。 这种轻量级进程与内核线程之间 1:1 的关系称为一对一的线程模型。





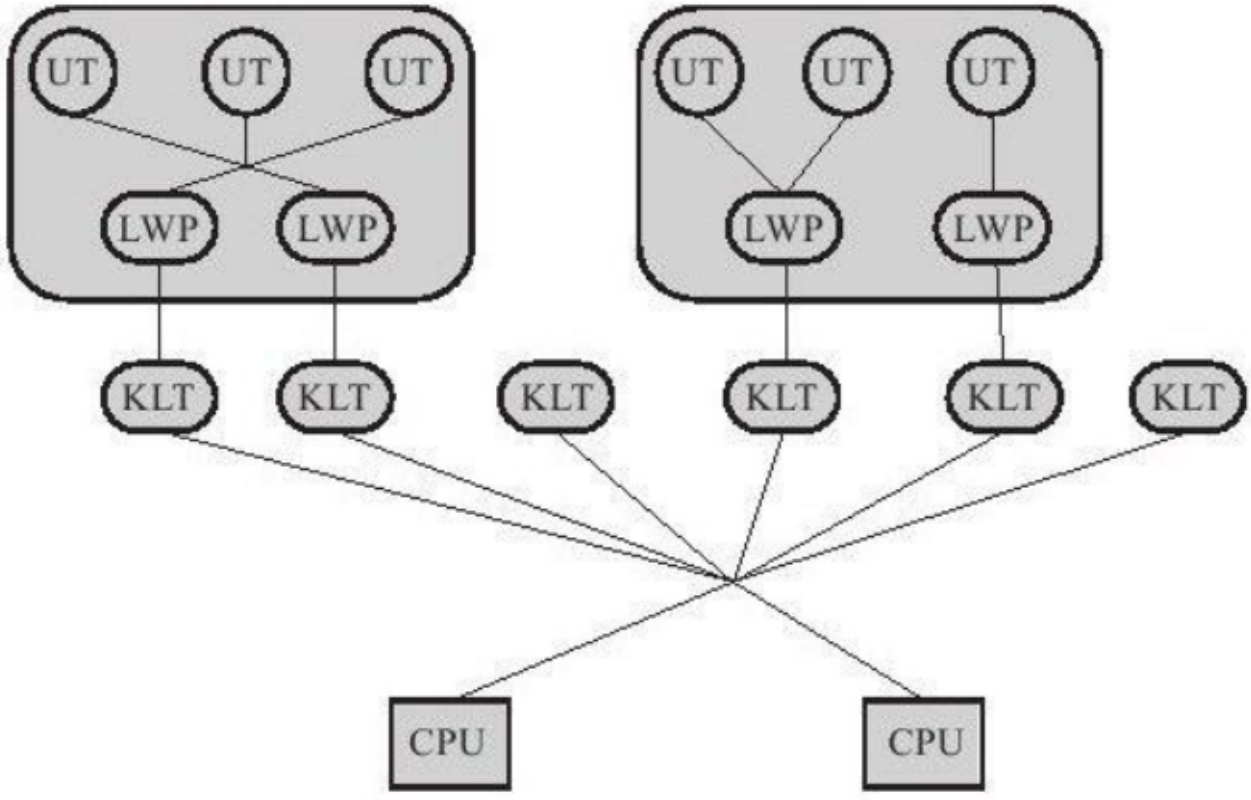
2. 使用用户线程实现

   用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现。 用户线程的建立、 同步、 销毁和调度完全在用户态中完成，不需要内核的帮助。 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





3. 使用用户线程加轻量级进程混合实现




用户线程与轻量级进程之间N：M的关系



## 线程调度

1. 协同式线程调度
   线程的执行时间由线程本身来控制，线程把自己的工作执行完了之后，要主动通知系统切换到另外一个线程上

2. 抢占式线程调度

每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定

3. 线程的状态转换