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。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，支持多线程的内核就叫做多线程内核（Multi-Threads Kernel）。

   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，如

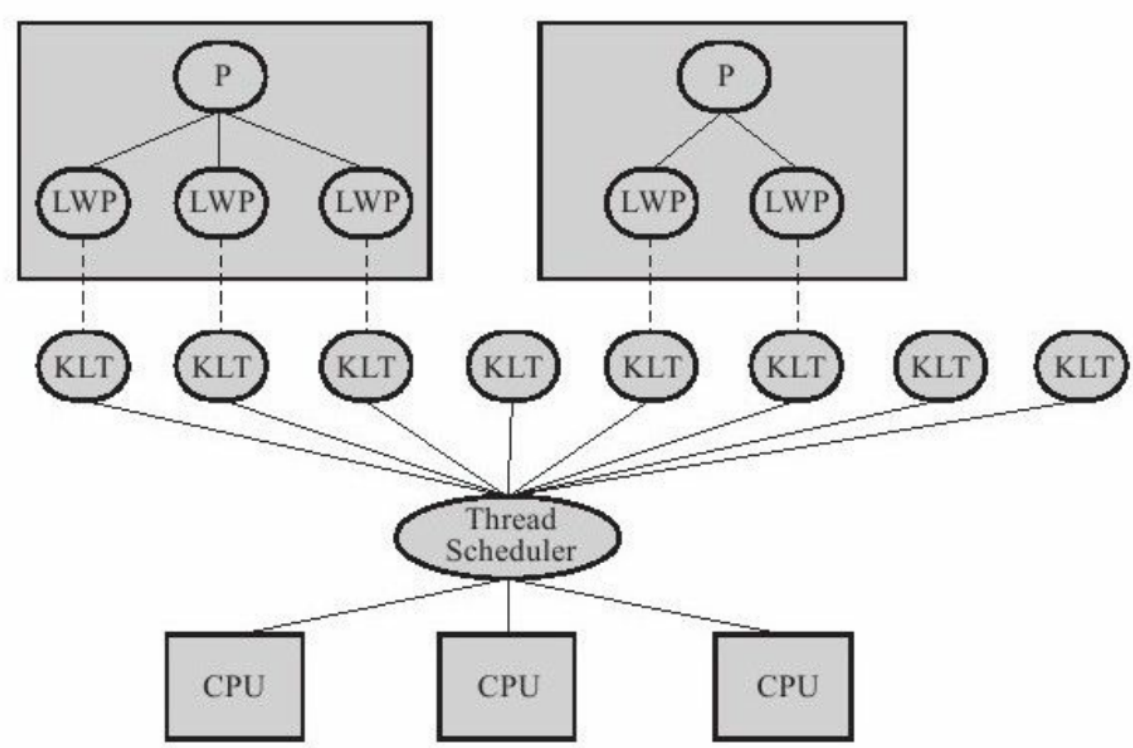


图 12-3 轻量级进程与内核线程之间1:1的关系

2. 使用用户线程实现

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。 如果程序实现得当， 这种线程不需要切换到内核态， 因此操作可以是非常快速且低消耗的， 也可以支持规模更大的线程数量， 部分高性能数据库中的多线程就是由用户线程实现的。 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， 如图12-4所示。

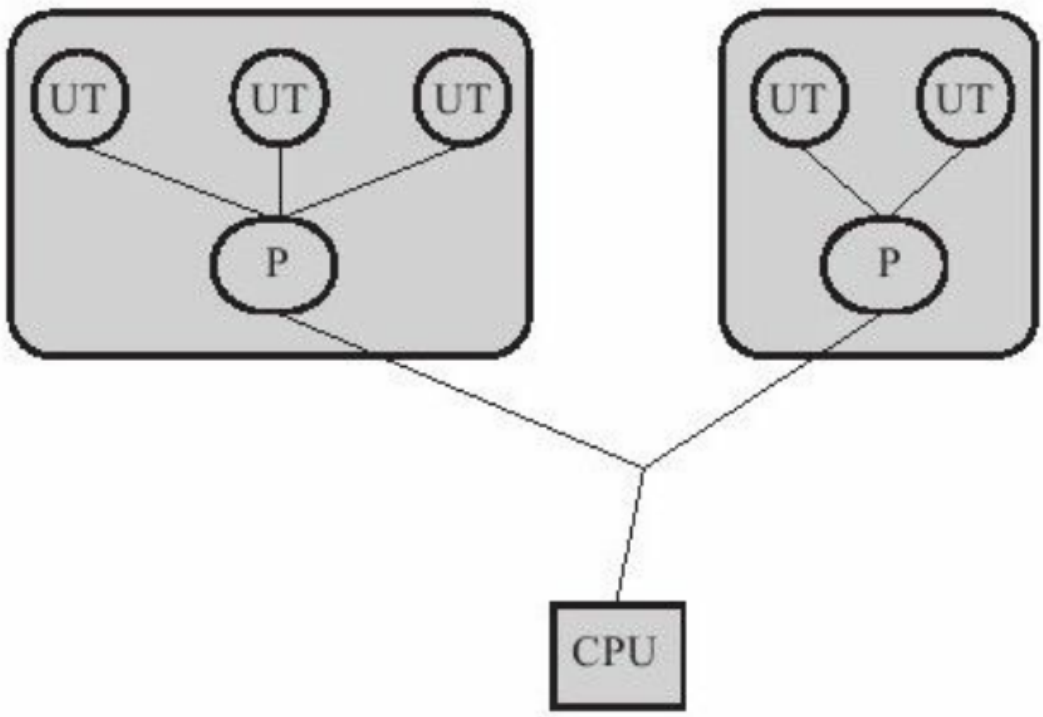


图 12-4 进程与用户线程之间1：N的关系

3. 使用用户线程加轻量级进程混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外， 还有一种将内核线程与用户线程一起使用的实现方式。 在这种混合实现下， 既存在用户线程， 也存在轻量级进程。 用户线程还是完全建立在用户空间中， 因此用户线程的创建、 切换、 析构等操作依然廉价， 并且可以支持大规模的用户线程并发。 而操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁， 这样可以使用内核提供的线程调度功能及处理器映射， 并且用户线程的系统调用要通过轻量级线程来完成， 大大降低了整个进程被完全阻塞的风险。 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的关系， 如图12-5所示， 这种就是多对多的线程模型。

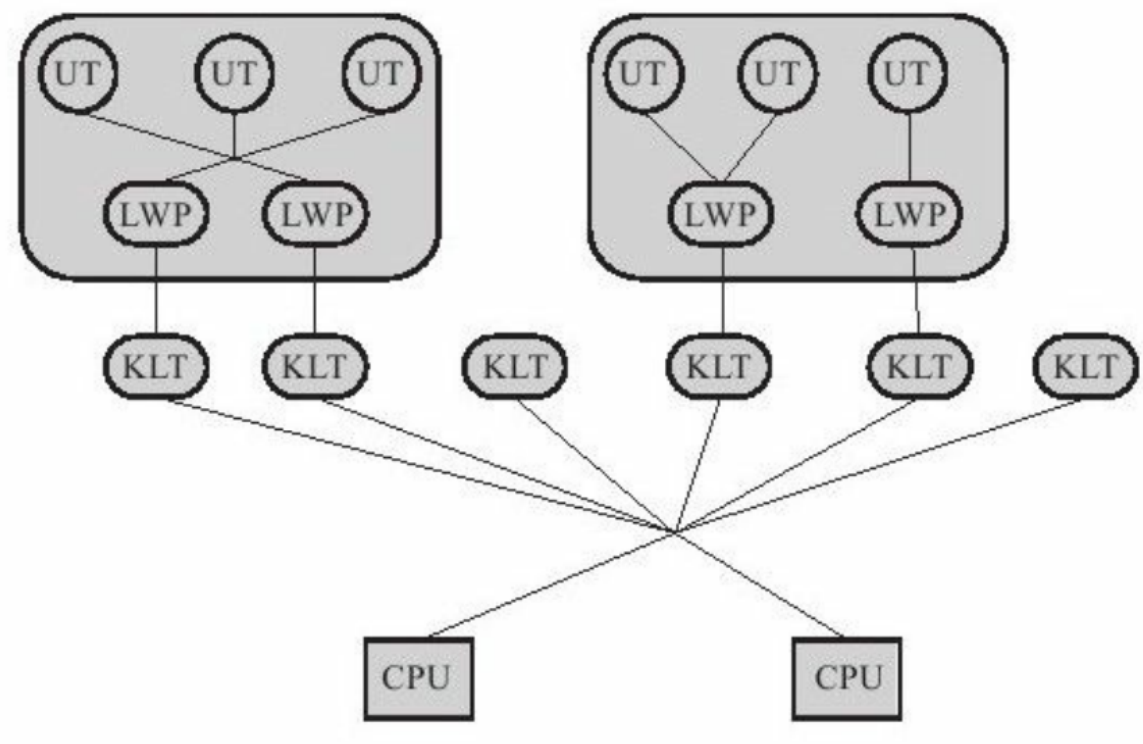
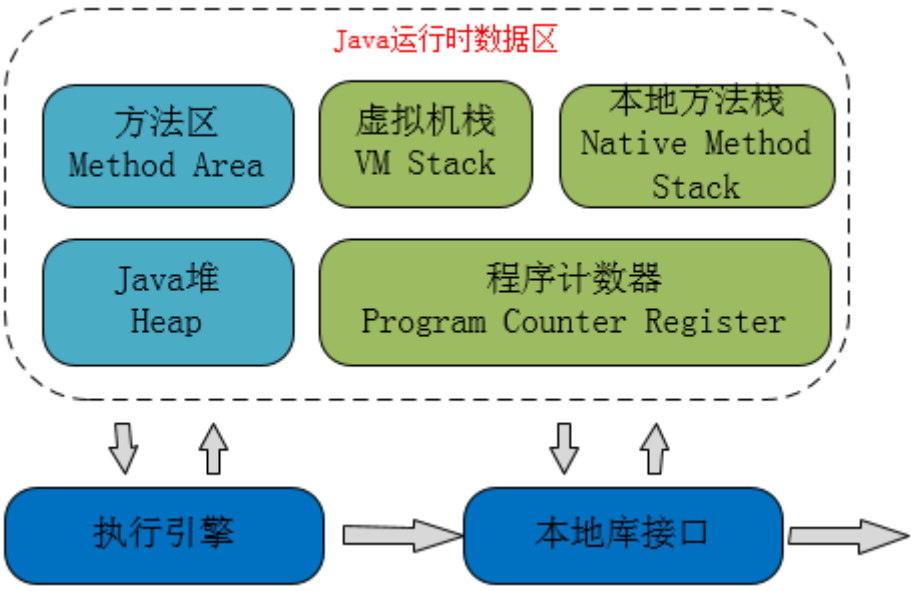


图 12-5 用户线程与轻量级进程之间N：M的关系

Java 内存区域

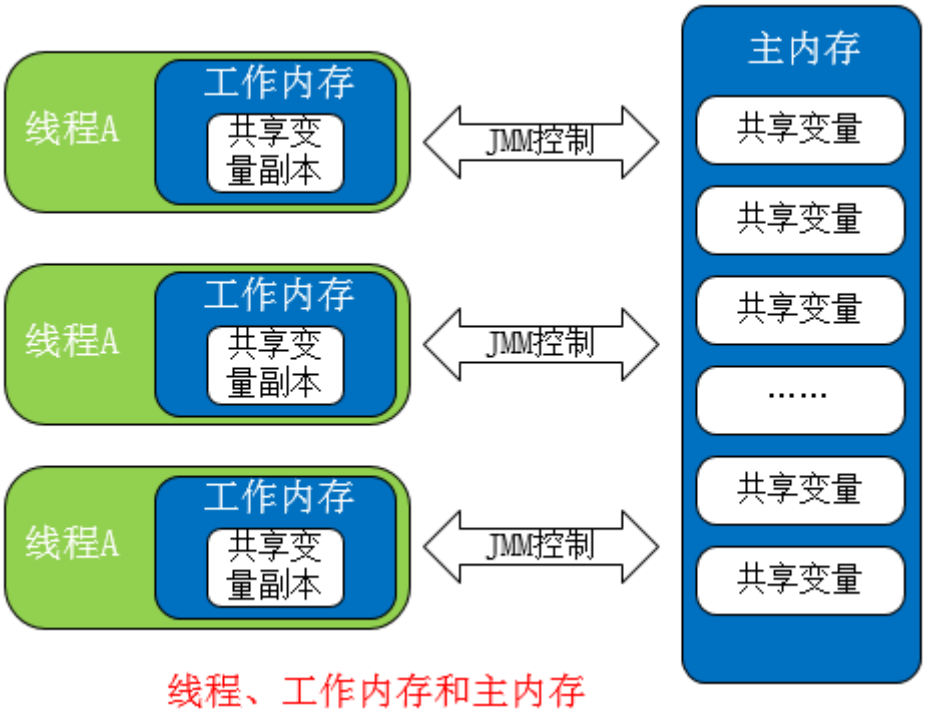


- 方法区（Method Area）：
  方法区属于线程共享的内存区域，又称Non-Heap（非堆），主要用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据，根据Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError 异常。值得注意的是在方法区中存在一个叫运行时常量池(Runtime Constant Pool）的区域，它主要用于存放编译器生成的各种字面量和符号引用，这些内容将在类加载后存放到运行时常量池中，以便后续使用。
- JVM堆（Java Heap）：
  Java 堆也是属于线程共享的内存区域，它在虚拟机启动时创建，是Java 虚拟机所管理的内存中最大的一块，主要用于存放对象实例，几乎所有的对象实例都在这里分配内存，注意 Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做GC 堆，如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError 异常。
- 程序计数器(Program Counter Register)：
  属于线程私有的数据区域，是一小块内存空间，主要代表当前线程所执行的字节码行号指示器。字节码解释器工作时，通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
- 虚拟机栈(Java Virtual Machine Stacks)：
  属于线程私有的数据区域，与线程同时创建，总数与线程关联，代表Java方法执行的内存模型。每个方法执行时都会创建一个栈桢来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。每个方法从调用直结束就对于一个栈桢在虚拟机栈中的入栈和出栈过程，如下：Alt text
- 本地方法栈(Native Method Stacks)：
  本地方法栈属于线程私有的数据区域，这部分主要与虚拟机用到的 Native 方法相关，一般情况下，我们无需关心此区域。

Java内存模型（JMM）

> - 并发程序下需要保证数据访问的一致性和安全性
> - JMM是围绕着多线程的原子性、可见性和有序性建立的

1. Java内存模型概述
   Java内存模型(即Java Memory Model，简称JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝，前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图



线程、工作内存和主内存

- 主内存
  主要存储的是Java实例对象，所有线程创建的实例对象都存放在主内存中，不管该实例对象是成员变量还是方法中的本地变量(也称局部变量)，当然也包括了共享的类信息、常量、静态变量。由于是共享数据区域，多条线程对同一个变量进行访问可能会发现线程安全问题。
- 工作内存
  主要存储当前方法的所有本地变量信息(工作内存中存储着主内存中的变量副本拷贝)，每个线程只能访问自己的工作内存，即线程中的本地变量对其它线程是不可见的，就算是两个线程执行的是同一段代码，它们也会各自在自己的工作内存中创建属于当前线程的本地变量，当然也包括了字节码行号指示器、相关Native方法的信息。注意由于工作内存是每个线程的私有数据，线程间无法相互访问工作内存，因此存储在工作内存的数据不存在线程安全问题。

> 弄清楚主内存和工作内存后，接了解一下主内存与工作内存的数据存储类型以及操作方式，根据虚拟机规范，对于一个实例对象中的成员方法而言，如果方法中包含本地变量是基本数据类型，将直接存储在工作内存的帧栈结构中，但倘若本地变量是引用类型，那么该变量的引用会存储在功能内存的帧栈中，而对象实例将存储在主内存(共享数据区域，堆)中。但对于实例对象的成员变量，不管它是基本数据类型还是引用类型，都会被存储到堆区。至于static变量以及类本身相关信息将会存储在主内存中。需要注意的是，在主内存中的实例对象可以被多线程共享，倘若两个线程同时调用了同一个对象的同一个方法，那么两条线程会将要操作的数据拷贝一份到自己的工作内存中，执行完成操作后才刷新到主内存。

2. 内存间的操作
   关于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、 如何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，虚拟机实现时必须保证下面提及的每一种操作都是原子的、 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，释放后的变量才可以被其他线程锁定。
- 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，以便随后的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变量副本中。
- 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。
- 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，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。
- 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，以便随后的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的变量中。

3. Java内存模型的承诺

- 原子性
  原子性是指一个操作是不可中断的，即使是在多个线程一起执行的时候，一个线程一旦开始，就不会被其他线程干扰。
- 可见性
  可见性是指一个线程修改了一个共享变量的值，其他线程是否能够立即知道这个修改。
- 有序性
  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一个线程，所有的操作都是无序的。

> 指令不能重排的规则：happen-before规则
>
> - 程序顺序性原则：一个线程内保证语义的串行性
> - volatile 规则：volatile变量的写，先发生于读，这保证了volatile变量的可见性
> - 锁规则：解锁（unlock）必然发生在随后加锁（lock）前
> - 传递性：A先于B，B先于C，那么A必然先于C
> - 线程的start（）方法先于它的每一个动作
> - 线程的所有操作先于线程的终结（Thread.join()）
> - 线程的中断（interrupt（））先于被中断的代码
> - 对象的构造函数执行、结束先于finalize（）方法

4. volatile内存语义
   - 保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总数可以被其他线程立即得知。
   - 禁止指令重排序优化。volatile修饰的变量，赋值后多执行了一个“lock” 操作，这个操作相当于一个内存屏障，只有一个CPU访问内存时，并不需要内存屏障；但如果有两个或更多CPU访问同一块内存，且其中有一个在观测另一个，就需要内存屏障来保证一致性了。

> **注意：**
>
> - 由于volatile变量只能保证可见性， 仍然要通过加锁（使用chronized或java.util.concurrent中的原子类） 来保证原子性。

互斥同步锁（synchronized）

synchronized的三种应用方式

> - 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
> - 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁(其锁就是当前类的class对象锁)
> - 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。
>
> **需要注意不是锁同一个对象的问题**

synchronized底层语义原理

> Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现
>
> 1. 同步代码块有明确的 monitorenter 和 monitorexit 指令
> 2. 同步方法是由方法调用指令读取运行时常量池中方法的 ACC_SYNCHRONIZED 标志来隐式实现的

1. 理解Java对象头与Monitor
   在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。
   jvm中采用2个字来存储对象头(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由Mark Word 和 Class Metadata Address 组成，其结构说明如下表：

虚拟机位数	头对象结构	说明
32/64bit	Mark Word	存储对象的hashCode、锁信息或分代年龄或GC标志等信息
32/64bit	Class Metadata Address	类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的实例。

其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构。

锁状态	25bit	4bit	1bit是否是偏向锁	2bit 锁标志位
无锁状态	对象HashCode	对象分代年龄	0	01

重量级锁也就是通常说synchronized的对象锁，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由ObjectMonitor实现的，其主要数据结构如下（位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的）

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0; //记录个数
    _waiters      = 0,
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL; //处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ; //处于等待锁block状态的线程，会被加入到该列表
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
  }
```

ObjectMonitor中有两个队列，_WaitSet 和 _EntryList，用来保存ObjectWaiter对象列表(每个等待锁的线程都会被封装成ObjectWaiter对象)，_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 _EntryList 集合，当线程获取到对象的monitor 后进入 _Owner 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。

2. synchronized代码块底层原理
   同步语句块的实现使用的是monitorenter 和 monitorexit 指令，其中monitorenter指令指向同步代码块的开始位置，monitorexit指令则指明同步代码块的结束位置，当执行monitorenter指令时，当前线程将试图获取 objectref(即对象锁) 所对应的 monitor 的持有权，当 objectref 的 monitor 的进入计数器为 0，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功。
   为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令。从字节码中也可以看出多了一个monitorexit指令，它就是异常结束时被执行的释放monitor 的指令。

3. synchronized方法底层原理
   synchronized修饰的方法并没有monitorenter指令和monitorexit指令，取得代之的确实是ACC_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。
4. Java虚拟机对synchronized的优化

    - 偏向锁
      偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。
    - 轻量级锁
      倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。
    - 自旋锁
      轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。
    - 锁消除
      消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

基于CAS和AQS的锁原理分析

CAS 比较交换，与众不同的并发策略

- CAS算法：包含三个参数CAS（V，E，N），V表示要更新的变量，E表示预期值，N表示新值
- Java中的指针UnSafe类
    - allocateMemory、reallocateMemory、freeMemory分别用于分配内存，扩充内存和释放内存；
    - 可以定位对象某字段的内存位置，也可以修改对象的字段值，即使它是私有的；
    - 线程挂起与恢复，对线程的挂起操作被封装在 LockSupport类中；
    - CAS操作：是通过compareAndSwapXXX方法实现的

AQS工作原理概要

- AbstractQueuedSynchronizer又称为队列同步器(后面简称AQS)，它是用来构建锁或其他同步组件的基础框架，内部通过一个int类型的成员变量state来控制同步状态,当state=0时，则说明没有任何线程占有共享资源的锁，当state=1时，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待，AQS内部通过内部类Node构成FIFO的同步队列来完成线程获取锁的排队工作，同时利用内部类ConditionObject构建等待队列，当Condition调用await()方法后，线程将会加入等待队列中，而当Condition调用signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步队列等待，而另一种则是等待队列(可有多个)，通过Condition调用await()方法释放锁后，将加入等待队列。Alt text

```
/**
 * AQS抽象类
 */
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer{
    //指向同步队列队头
    private transient volatile Node head;

    //指向同步的队尾
    private transient volatile Node tail;

    //同步状态，0代表锁未被占用，1代表锁已被占用
    private volatile int state;

    //省略其他代码......
}

static final class Node {
    //共享模式
    static final Node SHARED = new Node();
    //独占模式
    static final Node EXCLUSIVE = null;

    //标识线程已处于结束状态
    static final int CANCELLED =  1;
    //等待被唤醒状态
    static final int SIGNAL    = -1;
    //条件状态，
    static final int CONDITION = -2;
    //在共享模式中使用表示获得的同步状态会被传播
    static final int PROPAGATE = -3;

    //等待状态,存在CANCELLED、SIGNAL、CONDITION、PROPAGATE 4种
    volatile int waitStatus;

    //同步队列中前驱结点
    volatile Node prev;

    //同步队列中后继结点
    volatile Node next;

    //请求锁的线程
    volatile Thread thread;

    //等待队列中的后继结点，这个与Condition有关，稍后会分析
    Node nextWaiter;

    //判断是否为共享模式
    final boolean isShared() {
        return nextWaiter == SHARED;
    }
```

```
        //获取前驱结点
        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        //.....
    }

    //AQS中提供的主要模板方法，由子类实现。
    public abstract class AbstractQueuedSynchronizer
        extends AbstractOwnableSynchronizer{

            //独占模式下获取锁的方法
            protected boolean tryAcquire(int arg) {
                throw new UnsupportedOperationException();
            }

            //独占模式下解锁的方法
            protected boolean tryRelease(int arg) {
                throw new UnsupportedOperationException();
            }

            //共享模式下获取锁的方法
            protected int tryAcquireShared(int arg) {
                throw new UnsupportedOperationException();
            }

            //共享模式下解锁的方法
            protected boolean tryReleaseShared(int arg) {
                throw new UnsupportedOperationException();
            }
            //判断是否为持有独占锁
            protected boolean isHeldExclusively() {
                throw new UnsupportedOperationException();
            }

        }
```

基于AQS独占模式的ReetrantLock实现过程分析

1. ReetrantLock中非公平锁
   AQS同步器的实现依赖于内部的同步队列(FIFO的双向链表对列)完成对同步状态(state)的管理，当前线程获取锁(同步状态)失败时，AQS会将该线程以及相关等待信息包装成一个节点(Node)并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会将头结点head中的线程唤醒，让其尝试获取同步状态。关于同步队列和Node结点，前面我们已进行了较为详

细的分析，这里重点分析一下获取同步状态和释放同步状态以及如何加入队列的具体操作，这里从ReetrantLock入手分析AQS的具体实现，我们先以非公平锁为例进行分析。

```
//默认构造，创建非公平锁NonfairSync
public ReentrantLock() {
    sync = new NonfairSync();
}
//根据传入参数创建锁类型
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

//加锁操作
public void lock() {
     sync.lock();
}
```

前面说过sync是个抽象类，存在两个不同的实现子类，这里从非公平锁入手，看看其实现

```
/**
 * 非公平锁实现
 */
static final class NonfairSync extends Sync {
    //加锁
    final void lock() {
        //执行CAS操作，获取同步状态
        if (compareAndSetState(0, 1))
       //成功则将独占锁线程设置为当前线程
          setExclusiveOwnerThread(Thread.currentThread());
        else
            //否则再次请求同步状态
            acquire(1);
    }
}
```

这里获取锁时，首先对同步状态执行CAS操作，尝试把state的状态从0设置为1，如果返回true则代表获取同步状态成功，也就是当前线程获取锁成，可操作临界资源，如果返回false，则表示已有线程持有该同步状态(其值为1)，获取锁失败，注意这里存在并发的情景，也就是可能同时存在多个线程设置state变量，因此是CAS操作保证了state变量操作的原子性。返回false后，执行 acquire(1)方法，该方法是AQS中的方法，它对中断不敏感，即使线程获取同步状态失败，进入同步队列，后续对该线程执行中断操作也不会从同步队列中移出，方法如下

```
public final void acquire(int arg) {
    //再次尝试获取同步状态
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

这里传入参数arg表示要获取同步状态后设置的值(即要设置state的值)，因为要获取锁，而status为0时是释放锁，1则是获取锁，所以这里一般传递参数为1，进入方法后首先会执行tryAcquire(arg)方法，在前面分析过该方法在AQS中并没有具体实现，而是交由子类实现，因此该方法是由ReetrantLock类内部实现的

```
//NonfairSync类
static final class NonfairSync extends Sync {

    protected final boolean tryAcquire(int acquires) {
         return nonfairTryAcquire(acquires);
     }
 }

//Sync类
abstract static class Sync extends AbstractQueuedSynchronizer {

  //nonfairTryAcquire方法
  final boolean nonfairTryAcquire(int acquires) {
      final Thread current = Thread.currentThread();
      int c = getState();
      //判断同步状态是否为0，并尝试再次获取同步状态
      if (c == 0) {
          //执行CAS操作
          if (compareAndSetState(0, acquires)) {
              setExclusiveOwnerThread(current);
              return true;
          }
      }
      //如果当前线程已获取锁，属于重入锁，再次获取锁后将status值加1
      else if (current == getExclusiveOwnerThread()) {
          int nextc = c + acquires;
          if (nextc < 0) // overflow
              throw new Error("Maximum lock count exceeded");
          //设置当前同步状态，当前只有一个线程持有锁，因为不会发生线程安全问题，可以直
接执行 setState(nextc);
          setState(nextc);
          return true;
      }
      return false;
  }
  //省略其他代码
}
```

从代码执行流程可以看出，这里做了两件事，一是尝试再次获取同步状态，如果获取成功则将当前线程设置为OwnerThread，否则失败，二是判断当前线程current是否为OwnerThread，如果是则属于重入锁，state自增1，并获取锁成功，返回true，反之失败，返回false，也就是tryAcquire(arg)执行失败，返回false。需要注意的是nonfairTryAcquire(int acquires)内部使用的是CAS原子性操作设置state值，可以保证state的更改是线程安全的，因此只要任意一个线程调用nonfairTryAcquire(int acquires)方法并设置成功即可获取锁，不管该线程是新到来的还是已在同步队列的线程，毕竟这是非公平锁，并不保证同步队列中的线程一定比新到来线程请求先获取到锁。

```
public final void acquire(int arg) {
    //再次尝试获取同步状态
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

如果tryAcquire(arg)返回true，acquireQueued自然不会执行，这是最理想的，因为毕竟当前线程已获取到锁，如果tryAcquire(arg)返回false，则会执行addWaiter(Node.EXCLUSIVE)进行入队操作，由于ReentrantLock属于独占锁，因此结点类型为Node.EXCLUSIVE，下面看看addWaiter方法具体实现

```
private Node addWaiter(Node mode) {
    //将请求同步状态失败的线程封装成结点
    Node node = new Node(Thread.currentThread(), mode);

    Node pred = tail;
    //如果是第一个结点加入肯定为空，跳过。
    //如果非第一个结点则直接执行CAS入队操作，尝试在尾部快速添加
    if (pred != null) {
        node.prev = pred;
        //使用CAS执行尾部结点替换，尝试在尾部快速添加
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    //如果第一次加入或者CAS操作没有成功执行enq入队操作
    enq(node);
    return node;
}
```

创建了一个Node.EXCLUSIVE类型Node结点用于封装线程及其相关信息，其中tail是AQS的成员变量，指向队尾(这点前面的我们分析过AQS维持的是一个双向的链表结构同步队列)，如果是第一个结点，则为tail肯定为空，那么将执行enq(node)操作，如果非第一个结点即tail指向不为null，直接尝试执行CAS操作加入队尾，如果CAS操作失败还是会执行enq(node)，继续看enq(node)：

```
private Node enq(final Node node) {
    //死循环
    for (;;) {
        Node t = tail;
        //如果队列为null，即没有头结点
        if (t == null) { // Must initialize
            //创建并使用CAS设置头结点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {//队尾添加新结点
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
     }
    }
```

这个方法使用一个死循环进行CAS操作，可以解决多线程并发问题。这里做了两件事，一是如果还没有初始同步队列则创建新结点并使用compareAndSetHead设置头结点，tail也指向head，二是队列已存在，则将新结点node添加到队尾。注意这两个步骤都存在同一时间多个线程操作的可能，如果有一个线程修改head和tail成功，那么其他线程将继续循环，直到修改成功，这里使用CAS原子操作进行头结点设置和尾结点tail替换可以保证线程安全，从这里也可以看出head结点本身不存在任何数据，它只是作为一个牵头结点，而tail永远指向尾部结点(前提是队列不为null)。

添加到同步队列后，结点就会进入一个自旋过程，即每个结点都在观察时机待条件满足获取同步状态，然后从同步队列退出并结束自旋，回到之前的acquire()方法，自旋过程是在acquireQueued(addWaiter(Node.EXCLUSIVE), arg))方法中执行的，代码如下

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        //自旋，死循环
        for (;;) {
            //获取前驱结点
            final Node p = node.predecessor();
            当且仅当p为头结点才尝试获取同步状态
            if (p == head && tryAcquire(arg)) {
                //将node设置为头结点
                setHead(node);
                //清空原来头结点的引用便于GC
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            //如果前驱结点不是head，判断是否挂起线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            //最终都没能获取同步状态，结束该线程的请求
            cancelAcquire(node);
    }
}
```

当前线程在自旋(死循环)中获取同步状态，当且仅当前驱结点为头结点才尝试获取同步状态，这符合FIFO的规则，即先进先出，其次head是当前获取同步状态的线程结点，只有当head释放同步状态唤醒后继结点，后继结点才有可能获取到同步状态，因此后继结点在其前继结点为head时，才进行尝试获取同步状态，其他时刻将被挂起。进入if语句后调用setHead(node)方法，将当前线程结点设置为head

```
//设置为头结点
private void setHead(Node node) {
        head = node;
        //清空结点数据
        node.thread = null;
        node.prev = null;
}
```

从图可知更新head结点的指向，将后继结点的线程唤醒并获取同步状态，调用setHead(node)将其替换为head结点，清除相关无用数据。当然如果前驱结点不是head，那么执行如下

```
//如果前驱结点不是head，判断是否挂起线程
if (shouldParkAfterFailedAcquire(p, node) &&parkAndCheckInterrupt())
      interrupted = true;
}

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)
 {
        //获取当前结点的等待状态
        int ws = pred.waitStatus;
        //如果为等待唤醒（SIGNAL）状态则返回true
        if (ws == Node.SIGNAL)
            return true;
        //如果ws>0 则说明是结束状态，
        //遍历前驱结点直到找到没有结束状态的结点
        if (ws > 0) {
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            //如果ws小于0又不是SIGNAL状态，
            //则将其设置为SIGNAL状态，代表该结点的线程正在等待唤醒。
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }

private final boolean parkAndCheckInterrupt() {
        //将当前线程挂起
        LockSupport.park(this);
        //获取线程中断状态,interrupted()是判断当前中断状态，
        //并非中断线程，因此可能true也可能false,并返回
        return Thread.interrupted();
}
```

shouldParkAfterFailedAcquire()方法的作用是判断当前结点的前驱结点是否为SIGNAL状态(即等待唤醒状态)，如果是则返回true。如果结点的ws为CANCELLED状态(值为1>0),即结束状态，则说明该前驱结点已没有用应该从同步队列移除，执行while循环，直到寻找到非CANCELLED状态的结点。倘若前驱结点的ws值不为CANCELLED，也不为SIGNAL(当从Condition的条件等待队列转移到同步队列时，结点状态为CONDITION因此需要转换为SIGNAL)，那么将其转换为SIGNAL状态，等待被唤醒。

若shouldParkAfterFailedAcquire()方法返回true，即前驱结点为SIGNAL状态同时又不是head结点，那么使用parkAndCheckInterrupt()方法挂起当前线程，称为WAITING状态，需要等待一个unpark()操作来唤醒它，到此ReetrantLock内部间接通过AQS的FIFO的同步队列就完成了lock()操作，这里我们总结成逻辑流程图