# 并发基础

## 什么场景下使用多线程

多线程解决的是等待问题。

1、cpu密集型的计算速度慢的数据，通过并行来提高计算速度。

2、等待网络I/O比较耗时的操作，通过异步减少网络阻塞。

## 实现多线程的方式

1. 继承Thread 类实现

|  |
| --- |
| public class ThreadDemo extends Thread{   public static void main(String[] args) {  ThreadDemo t1 = new ThreadDemo();  ThreadDemo t2 = new ThreadDemo();  ThreadDemo t3 = new ThreadDemo();  t1.start();  t2.start();  t3.start();  }  @Override  public void run() {  System.*out*.println("ThreadDemo run....");  } } |

1. 实现Runnable接口

|  |
| --- |
| public class RunnableDemo implements Runnable {  public static void main(String[] args) {  for(int i=0;i<3;i++){  Thread t = new Thread(new RunnableDemo(),"线程"+i);  t.start();  }  }  @Override  public void run() {  System.*out*.println("线程名称:"+Thread.*currentThread*().getName()+" RunnableDemo is run....");  } } |

1. 实现Callable接口通过futureTask包装器来创建线程

依赖子线程的返回结果

|  |
| --- |
| public class CallableDemo implements Callable<String> {  public static void main(String[] args) throws ExecutionException, InterruptedException {  ExecutorService executorService= Executors.*newFixedThreadPool*(1);  CallableDemo callableDemo = new CallableDemo();  Future<String> futures= executorService.submit(callableDemo);  System.*out*.println(futures.get());  executorService.shutdown();  }  @Override  public String call() throws Exception {  int a =1;  int b=2;  return "执行结果："+(a+b);  } } |

## 线程的状态

线程是java程序中执行的最小单元，有6种状态 NEW 、RUNNABLE、 BLOCKED 、WAITTING 、TIME\_WAITING 、TERMINATED

NEW：初始状态，线程被构建，但是还没有调用start方法

RUNNABLED：运行状态，JAVA线程把操作系统中的就绪和运行两种状态统一称为“运行中”

BLOCKED：阻塞状态，表示线程进入等待状态,也就是线程因为某种原因放弃了CPU使用权，阻塞也分为几种情况

* 等待阻塞：运行的线程执行wait方法，jvm会把当前线程放入到等待队列
* 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被其他线程锁占用了，那么jvm会把当前的线程放入到锁池中
* 其他阻塞：运行的线程执行Thread.sleep或者t.join方法，或者发出了I/O请求时，JVM会把当前线程设置为阻塞状态，当sleep结束、join线程终止、io处理完毕则线程恢复

TIME\_WAITING：超时等待状态，超时以后自动返回

TERMINATED：终止状态，表示当前线程执行完毕



## 线程的停止

虽然jdk 提供了stop方法，但是是一个过期的方法，调用后并不保证线程能停止，资源能够释放。如果要优雅的去中断一个线程，使用interrupt方法。

### interrupt方法

当其他线程通过调用当前线程的interrupt方法，表示向当前线程打个招呼，告诉他可以中断线程的执行了，至于什么时候中断，取决于当前线程自己。

线程通过检查资深是否被中断来进行相应，可以通过isInterrupted()来判断是否被中断。

|  |
| --- |
| public class InterruptDemo {  private static int *i*;  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (!Thread.*currentThread*().isInterrupted()){  *i*++;  }  System.*out*.println("i的值："+*i*);  }  },"interruptDemo");   t.start();  Thread.*sleep*(1000);  t.interrupt();  }  } |

### Thread.interrupted 线程复位

* 当前线程复位

|  |
| --- |
| public class InterruptedDemo {  private static int *i*;  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (true){  boolean ii=Thread.*currentThread*().isInterrupted();  if(ii){  System.*out*.println("before:"+ii);  //对线程进行复位  Thread.*interrupted*();  System.*out*.println("after:"+Thread.*currentThread*().isInterrupted());  }  }  //System.out.println("i的值："+i);  }  },"InterruptedDemo");   t.start();  Thread.*sleep*(1000);  t.interrupt();  }  } |

* 其他线程复位

除了通过Thread.interrupted方法对线程中断标识进行复位以外，还有一种被动复位的场景，就是对抛出InterruptedException异常的方法，在InterruptedException抛出之前，JVM会先把线程的中断标识位清除，然后才会抛出InterruptedException，这个时候如果调用isInterrupted方法，将会返回false

|  |
| --- |
| public class InterruptExceptionDemo {  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (true){   try {  Thread.*sleep*(100000);  } catch (InterruptedException e) {  //抛出该异常，会将复位标识设置为false  e.printStackTrace();  }  }  }  },"InterruptedDemo");  t.start();  TimeUnit.*SECONDS*.sleep(1);  t.interrupt();//设置复位标识为true  TimeUnit.*SECONDS*.sleep(1);  System.*out*.println(t.isInterrupted());//false  }  } |

### 其他的线程停止方法

|  |
| --- |
| public class VolatileDemo {  private static volatile boolean *stop*=false;   public static void main(String[] args) throws InterruptedException {  Thread thread = new Thread(new Runnable() {  int i=0;  @Override  public void run() {  while(!*stop*){  i++;  }  System.*out*.println("i="+i);  }  });  thread.start();  System.*out*.println("开始执行 Thread");  Thread.*sleep*(1000);  *stop*=true;   } } |

## 线程安全方面

### 线程安全性问题

线程安全问题是 可见性，原子性，有序性

### CPU高速缓存



高速缓存从下到上越接近CPU速度越快，同时容量也越小。现在大部分的处理器都有二级或者三级缓存，从下到上依次为 L3 cache, L2 cache, L1 cache. 缓存又可以分为指令缓存和数据缓存，指令缓存用来缓存程序的代码，数据缓存用来缓存程序的数据。

### CPU缓存一致性问题

CPU-0读取主存的数据，缓存到CPU-0的高速缓存中，CPU-1也做了同样的事情，而CPU-1把count的值修改成了2，并且同步到CPU-1的高速缓存，但是这个修改以后的值并没有写入到主存中，CPU-0访问该字节，由于缓存没有更新，所以仍然是之前的值，就会导致数据不一致的问题

引发这个问题的原因是因为多核心CPU情况下存在指令并行执行，而各个CPU核心之间的数据不共享从而导致缓存一致性问题，为了解决这个问题，CPU生产厂商提供了相应的解决方案

1. **总线锁**

当一个CPU对其缓存中的数据进行操作的时候，往总线中发送一个Lock信号。其他处理器的请求将会被阻塞，那么该处理器可以独占共享内存。总线锁相当于把CPU和内存之间的通信锁住了，所以这种方式会导致CPU的性能下降，所以P6系列以后的处理器，出现了另外一种方式，就是缓存锁。

1. **缓存锁**

如果缓存在处理器缓存行中的内存区域在LOCK操作期间被锁定，当它执行锁操作回写内存时，处理不在总线上声明LOCK信号，而是修改内部的缓存地址，然后通过缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域的数据，当其他处理器回写已经被锁定的缓存行的数据时会导致该缓存行无效。

所以如果声明了CPU的锁机制，会生成一个LOCK指令，会产生两个作用

1、 Lock前缀指令会引起引起处理器缓存回写到内存，在P6以后的处理器中，LOCK信号一般不锁总线，而是锁缓存

2. 一个处理器的缓存回写到内存会导致其他处理器的缓存无效

### 缓存一致性协议

比较经典的应该就是MESI协议了，它的方法是在CPU缓存中保存一个标记位，这个标记为有四种状态。

* M(Modified) 修改缓存，当前CPU缓存已经被修改，表示已经和内存中的数据不一致了
* E(Exclusive) 独占缓存，当前cpu的缓存和内存中数据保持一致 ，而且其他处理器没有缓存该数据
* S(Shared) 共享缓存，数据和内存中数据一致，并且该数据存在多个cpu缓存中
* I(Invalid) 失效缓存，说明CPU的缓存已经不能使用了

每个Core的Cache控制器不仅知道自己的读写操作，也监听其它Cache的读写操作，嗅探（snooping）"协议

CPU的读取会遵循几个原则

1. 如果缓存的状态是I，那么就从内存中读取，否则直接从缓存读取

2. 如果缓存处于M或者E的CPU 嗅探到其他CPU有读的操作，就把自己的缓存写入到内存，并把自己的状态设置为S

3. 只有缓存状态是M或E的时候，CPU才可以修改缓存中的数据，修改后，缓存状态变为MC

### CPU的优化执行

为了提高运行效率，cpu会在保证程序语义的前提下，会进行指令重排序。

### 并发编程问题

缓存一致性导致可见性、处理器的乱序执行导致原子性、指令重排序导致有序性。

### JMM内存模型

内存模型定义了共享内存系统中多线程程序读写操作行为的规范，来屏蔽各种硬件和操作系统的内存访问差异，来实现Java程序在各个平台下都能达到一致的内存访问效果。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，也就是在虚拟机中将变量存储到内存以及从内存中取出变量（这里的变量，指的是共享变量，也就是实例对象、静态字段、数组对象等存储在堆内存中的变量。而对于局部变量这类的，属于线程私有，不会被共享）这类的底层细节。通过这些规则来规范对内存的读写操作，从而保证指令执行的正确性。它与处理器有关、与缓存有关、与并发有关、与编译器也有关。他解决了CPU多级缓存、处理器优化、指令重排等导致的内存访问问题，保证了并发场景下的可见性、原子性和有序性，。内存模型解决并发问题主要采用两种方式：限制处理器优化和使用内存屏障

Java内存模型定义了线程和内存的交互方式，在JMM抽象模型中，分为主内存、工作内存。主内存是所有线程共享的，工作内存是每个线程独有的。线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存中的变量。并且不同的线程之间无法访问对方工作内存中的变量，线程间的变量值的传递都需要通过主内存来完成，他们三者的交互关系如下



所以，总的来说，JMM是一种规范，目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性。

## JMM解决线程的安全性问题

### JMM内部的一些技术

提供了一系列关键字来解决，volatile、Synchronized、final、juc等

* 原子性保障

在java中提供了两个高级的字节码指令monitorenter和monitorexit，在Java中对应的Synchronized来保证代码块内的操作是原子的

* 有序性保障

在Java中，可以使用synchronized和volatile来保证多线程之间操作的有序性。实现方式有所区别：volatile关键字会禁止指令重排。synchronized关键字保证同一时刻只允许一条线程操作。

* 可见性保障

Java中的volatile关键字提供了一个功能，那就是被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。因此，可以使用volatile来保证多线程操作时变量的可见性。除了volatile，Java中的synchronized和final两个关键字也可以实现可见性

### volatile如何保证可见性

volatile 可以防止指令重排序，指令重排序的目的是提高程序的运行效率，但是不会改变程序执行的语义。指令重排序必须要遵循的原则是，不影响代码执行的最终结果，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序，(这里所说的数据依赖性仅仅是针对单个处理器中执行的指令和单个线程中执行的操作.)这个语义，实际上就是as-if-serial语义，不管怎么重排序，单线程程序的执行结果不会改变，编译器、处理器都必

须遵守as-if-serial语义

### 内存屏障

CPU的乱序执行，本质还是，由于在多CPU的机器上，每个CPU都存在cache，当一个特定数据第一次被特定一个CPU获取时，由于在该CPU缓存中不存在，就会从内存中去获取，被加载到CPU高速缓存中后就能从缓存中快速访问。当某个CPU进行写操作时，它必须确保其他的CPU已经将这个数据从他们的缓存中移除，这样才能让其他CPU安全的修改数据。显然，存在多个cache时，我们必须通过一个cache一致性协议来避免数据不一致的问题，而这个通讯的过程就可能导致乱序访问的问题，也就是运行时的内存乱序访问。

现在的CPU架构都提供了内存屏障功能，在x86的cpu中，实现了相应的内存屏障

写屏障(store barrier)、读屏障(load barrier)和全屏障(Full Barrier)，主要的作用是

* 防止指令之间的重排序
* 保证数据的可见性

在JMM中把内存屏障指令分为4类，通过在不同的语义下使用不同的内存屏障来进制特定类型的处理器重排序，从而来保证内存的可见性

* LoadLoad Barriers, load1 ; LoadLoad; load2 , 确保load1数据的装载优先于load2及所有后续装载指令的装载
* StoreStore Barriers，store1; storestore;store2 , 确保store1数据对其他处理器可见优先于store2及所有后续存储指令的存储
* LoadStore Barries， load1;loadstore;store2, 确保load1数据装载优先于store2以及后续的存储指令刷新到内存
* StoreLoad Barries， store1; storeload;load2, 确保store1数据对其他处理器变得可见， 优先于load2及所有后续装载指令的装载；这条内存屏障指令是一个全能型的屏障，在前面讲cpu层面的内存屏障的时候有提到。它同时具有其他3条屏障的效果。

## Synchronized

三种应用方式

1. 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
2. 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。
3. 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。

### synchronized括号后面的对象

synchronized扩后后面的对象是一把锁，在java中任意一个对象都可以成为锁，如果括号里的对象不是同一个，那这个锁互不影响。

### synchronized的字节码指令

对于同步块的实现使用了monitorenter和monitorexit指令，他们隐式的执行了Lock和UnLock操作，用于提供原子性保证。monitorenter指令插入到同步代码块开始的位置、monitorexit指令插入到同步代码块结束位置，jvm需要保证每个monitorenter都有一个monitorexit对应。这两个指令，本质上都是对一个对象的监视器(monitor)进行获取，这个过程是排他的，也就是说同一时刻只能有一个线程获取到由synchronized所保护对象的监视器线程执行到monitorenter指令时，会尝试获取对象所对应的monitor所有权，也就是尝试获取对象的锁；而执行monitorexit，就是释放monitor的所有权。

### synchronized的锁的原理

jdk1.6以后对synchronized锁进行了优化，包含偏向锁、轻量级锁、重量级锁;要了解synchronized锁的原理需要了解两个概念，一个是对象头、另一个是monitor。

#### Java对象头

在Hotspot虚拟机中，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；Java对象头是实现synchronized的锁对象的基础，一般而言，synchronized使用的锁对象是存储在Java对象头里。它是轻量级锁和偏向锁的关键。

#### Mark Word

Mark Word用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等。Java对象头一般占有两个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit）

#### synchronized的锁升级和获取过程

synchronized的锁是进行过优化的，引入了偏向锁、轻量级锁；锁的级别从低到高逐步升级， 无锁->偏向锁->轻量级锁->重量级锁.

##### 自旋锁（CAS）

自旋锁就是让不满足条件的线程等待一段时间，而不是立即挂起。看持有锁的线程是否能够很快释放锁。怎么自旋呢？其实就是一段没有任何意义的循环。虽然它通过占用处理器的时间来避免线程切换带来的开销，但是如果持有锁的线程不能在很快释放锁，那么自旋的线程就会浪费处理器的资源，因为它不会做任何有意义的工作。所以，自旋等待的时间或者次数是有一个限度的，如果自旋超过了定义的时间仍然没有获取到锁，则该线程应该被挂起。

##### 偏向锁

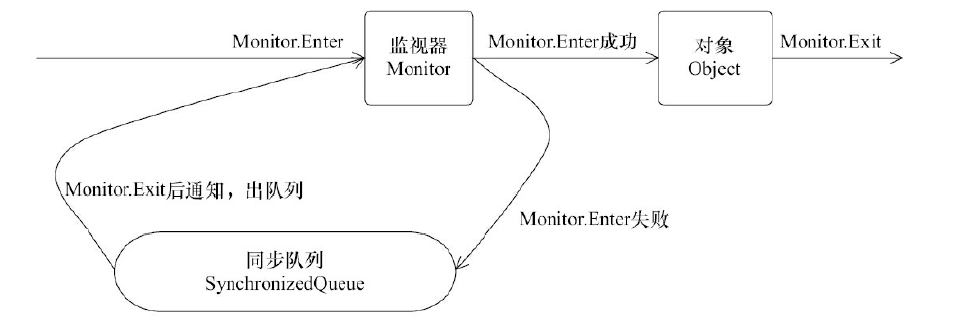
大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁）：如果没有设置，则使用CAS竞争锁；如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程

##### 轻量级锁

引入轻量级锁的主要目的是在多没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁

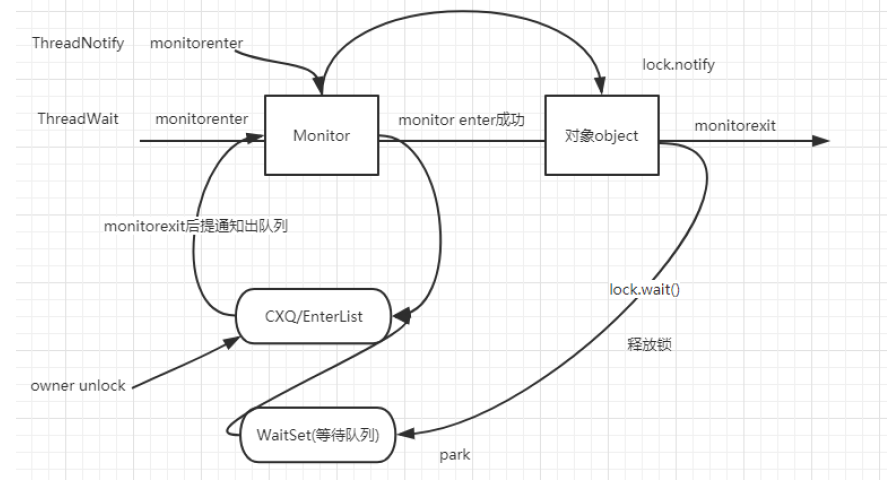
##### 重量级锁

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。

* 

## wait和notify

调用wait方法，首先会获取监视器锁，获得成功以后，会让当前线程进入等待状态进入等待队列并且释放锁；然后当其他线程调用notify或者notifyall以后，会选择从等待队列中唤醒任意一个线程，而执行完notify方法以后，并不会立马唤醒线程，原因是当前的线程仍然持有这把锁，处于等待状态的线程无法获得锁。必须要等到当前的线程执行完monitorexit指令以后，也就是锁被释放以后，处于等待队列中的线程就可以开始竞争锁了。



### wait和notify为什么需要在synchronized里面

wait方法的语义有两个，一个是释放当前的对象锁、另一个是使得当前线程进入阻塞队列， 而这些操作都和监视器是相关的，所以wait必须要获得一个监视器锁而对于notify来说也是一样，它是唤醒一个线程，既然要去唤醒，首先得知道它在哪里？所以就必须要找到这个对象获取到这个对象的锁，然后到这个对象的等待队列中去唤醒一个线程。

## 锁

### 可重入锁 ReentrantLock

|  |
| --- |
| public class ReentrantLockDemo {  private static int *count*=0;  private static Lock *lock* = new ReentrantLock();  public static void inc(){  *lock*.lock();  try {  Thread.*sleep*(1000);  *count*++;  } catch (InterruptedException e) {  e.printStackTrace();  }finally {  *lock*.unlock();  }  }  public static void main(String[] args) throws InterruptedException {  for(int i=0;i<500;i++){  new Thread(new Runnable() {  @Override  public void run() {  ReentrantLockDemo.*inc*();  }  }).start();  }  Thread.*sleep*(4000);  System.*out*.println("result:"+*count*);  } } |

### 读写锁 ReentrantReadWriteLock

我们以前理解的锁，基本都是排他锁，也就是这些锁在同一时刻只允许一个线程进行访问，而读写所在同一时刻可以允许多个线程访问，但是在写线程访问时，所有的读线程和其他写线程都会被阻塞。读写锁维护了一对锁，一个读锁、一个写锁; 一般情况下，读写锁的性能都会比排它锁好，因为大多数场景读是多于写的。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量。

|  |
| --- |
| public class ReadWriteLockDemo {  private static Map<String,Object> *cacheMap* = new HashMap<>();  private static ReentrantReadWriteLock *lock* = new ReentrantReadWriteLock();  static Lock *read* =*lock*.readLock();  static Lock *write* = *lock*.writeLock();  public static final Object get(String key){  System.*out*.println("开始获取读锁内容");  *read*.lock();  try {  return *cacheMap*.get(key);  }finally {  *read*.unlock();  }  }  public static void put(String key ,Object value){  System.*out*.println("开始写内容");  *write*.lock();  try {  *cacheMap*.put(key,value);  }finally {  *write*.unlock();  }  }  public static void main(String[] args) {  ReadWriteLockDemo.*put*("time",new Date());  Object obj = ReadWriteLockDemo.*get*("time");  System.*out*.println(obj);  } } |

在这个案例中，通过hashmap来模拟了一个内存缓存，然后使用读写所来保证这个内存缓存的线程安全性。当执行读操作的时候，需要获取读锁，在并发访问的时候，读锁不会被阻塞，因为读操作不会影响执行结果。

在执行写操作是，线程必须要获取写锁，当已经有线程持有写锁的情况下，当前线程会被阻塞，只有当写锁释放以后，其他读写操作才能继续执行。使用读写锁提升读操作的并发性，也保证每次写操作对所有的读写操作的可见性

* 读锁与读锁可以共享
* 读锁与写锁不可以共享（排他）
* 写锁与写锁不可以共享（排他）

### Lock和synchronized的简单对比

* Lock是一个类，synchronized是一个Java关键字
* Lock使用更灵活，可以自由控制锁的获取和释放。而synchronized的锁的释放是被动的，当出现异常或者同步代码块执行完以后，才会释放锁。
* Lock 可以判断锁的状态，而synchronized做不到。
* Lock 可以实现公平锁和非公平锁，而synchronized只有非公平锁。

## AQS（基础重点）

Lock之所以能实现线程安全的锁，主要的核心是AQS(AbstractQueuedSynchronizer),AbstractQueuedSynchronizer提供了一个FIFO队列，可以看做是一个用来实现锁以及其他需要同步功能的框架。这里简称该类为AQS。AQS的使用依靠继承来完成，子类通过继承自AQS并实现所需的方法来管理同步状态。例如常见的ReentrantLock，CountDownLatch等AQS的两种功能。

### AQS的功能可以分为两种：独占和共享

* 独占锁模式下，每次只能有一个线程持有锁，比如ReentrantLock就是以独占方式实现的互斥锁
* 共享锁模式下，允许多个线程同时获取锁，并发访问共享资源，比如ReentrantReadWriteLock。

独占锁是一种悲观保守的加锁策略，它限制了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。

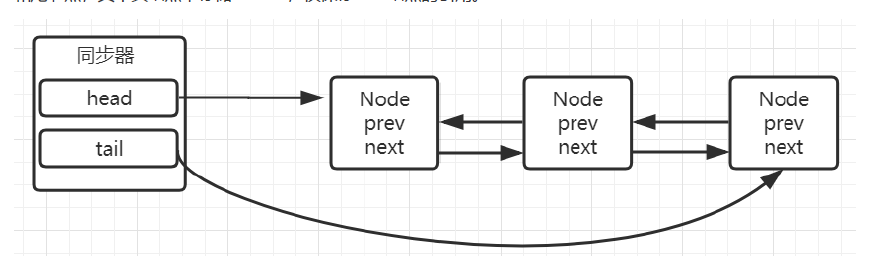
### AQS的内部实现

同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点（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，会把首节点中的线程唤醒，使其再次尝试获取同步状态。

Node的主要属性如下

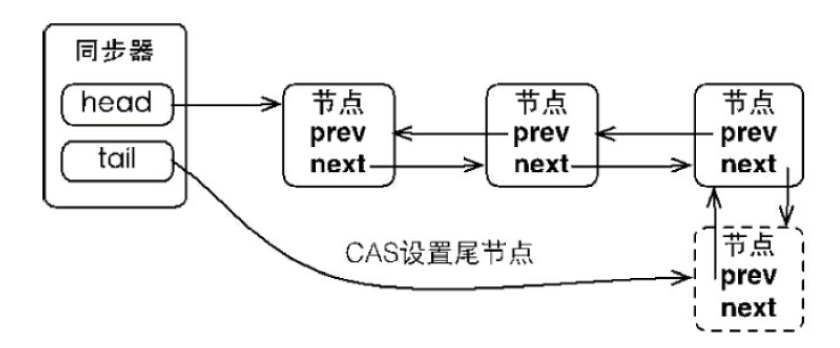
|  |
| --- |
| static final class Node {  int waitStatus; //表示节点的状态，包含cancelled（取消）；condition 表示节点在等待condition  也就是在condition队列中  Node prev; //前继节点  Node next; //后继节点  Node nextWaiter; //存储在condition队列中的后继节点  Thread thread; //当前线程  } |

AQS类底层的数据结构是使用双向链表，是队列的一种实现。包括一个head节点和一个tail节点，分别表示头结点和尾节点，其中头结点不存储Thread，仅保存next结点的引用。

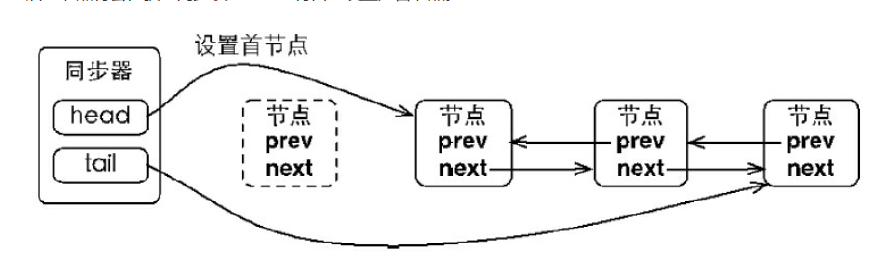


当一个线程成功地获取了同步状态（或者锁），其他线程将无法获取到同步状态，转而被构造成为节点并加入到同步队列中，而这个加入队列的过程必须要保证线程安全，因此

同步器提供了一个基于CAS的设置尾节点的方法：compareAndSetTail(Nodeexpect,Nodeupdate)，它需要传递当前线程“认为”的尾节点和当前节点，只有设置成功后，当前节点才正式与之前的尾节点建立关联。



同步队列遵循FIFO，首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，而后继节点将会在获取同步状态成功时将自己设置为首节点。



设置首节点是通过获取同步状态成功的线程来完成的，由于只有一个线程能够成功获取到同步状态，因此设置头节点的方法并不需要使用CAS来保证，它只需要将首节点设置成为原首节点的后继节点并断开原首节点的next引用即可。