Java并发

seaboat

目　录

前言

> 原文出处：Java并发
> 作者：汪建

Java并发

> 涉及java并发相关的文章

悲观的并发策略——Synchronized互斥锁

volatile既然不足以保证数据同步，那么就必须要引入锁来确保。互斥锁是最常见的同步手段，在并发过程中，当多条线程对同一个共享数据竞争时，它保证共享数据同一时刻只能被一条线程使用，其他线程只有等到锁释放后才能重新进行竞争。对于java开发人员，我们最熟悉的肯定就是用synchronized关键词完成锁功能，在涉及到多线程并发时，对于一些变量，你应该会毫不犹豫地加上synchronized去保证变量的同步性。

在C/C++可直接使用操作系统提供的互斥锁实现同步和线程的阻塞和唤起，与之不同的是，java要把这些底层封装，而synchronized就是一个典型的互斥锁，同时它也是一个JVM级别的锁，它的实现细节全部封装在JVM中实现，对开发人员只提供了synchronized关键词。根据锁的颗粒度，可以用synchronized对一个变量、一个方法、一个对象和一个类等加锁。被synchronized修饰的程序块经过编译后，会在前后生成monitorenter和monitorexit两个字节码指令，其中涉及到锁定和解锁对象的确定，这就要根据synchronized来确定了，假如明确指定了所对象，例如synchronized(变量)、synchronized(this)等，说明加解锁对象为变量或运行时对象。假如没有明确指定对象，则根据synchronized修饰的方法去找对应的锁对象，如修饰一个非静态方法表示此方法对应的对象为锁对象，如修饰一个静态方法则表示此方法对应的类对象为锁对象。当一个对象被锁住时，对象里面所有用synchronized修饰的方法都将产生堵塞，而对象里非synchronized修饰的方法可正常被调用，不收锁影响。

为了实现互斥锁，JVM的monitorenter和monitorexit字节码依赖底层操作系统的互斥锁来实现，java层面的线程与操作系统的原生线程有映射关系，这时如果要将一个线程进行阻塞或唤起都需要操作系统的协助，这就需要从用户态切换到内核态来执行，这种切换代价十分昂贵，需要消耗很多处理器时间。如果可能，应该减少这样的切换，jvm一般会采取一些措施进行优化，例如在把线程进行阻塞操作之前先让线程自旋等待一段时间，可能在等待期间其他线程已经解锁，这时就无需再让线程执行阻塞操作，避免了用户态到内核态的切换。

Synchronized还有另外一个重要的特性——可重入性。这个特性主要是针对当前线程而言的，可重入即是自己可以再次获得自己的内部锁，在尝试获取对象锁时，如果当前线程已经拥有了此对象的锁，则把锁的计数器加一，在释放锁时则对应地减一，当锁计数器为0时表示锁完全被释放，此时其他线程可对其加锁。可重入特性是为了解决自己锁死自己的情况，如下面伪代码：

```
public class DeadLock{

public synchronized void method1(){}

public synchronized void method2(){

this.method1();

}
```

```
public static void main(String[] args){
  DeadLock deadLock=new DeadLock();
   deadLock.method2();
}
}
```

这种情况其实也并非不常见，一个类中的同步方法调用另一个同步方法，假如synchronized不支持重入，进入method2方法时当前线程将尝试获取deadLock对象的锁，而method2方法里面执行method1方法时，当前线程又要去尝试获取deadLock对象的锁，这时由于不支持重入，它要去等deadLock对象的锁释放，把自己阻塞了，这就是自己锁死自己的现象。所以重入机制的引入，杜绝了这种情况的发生。

synchronized实现的是一个非公平锁，非公平主要表现在获取锁的行为上，并非是按照申请锁的时间前后给等待线程分配锁的，每当锁被释放后，任何一个线程都有机会竞争到锁，这样做的目的是为了提高执行性能，当然也会产生线程饥饿现象。

synchronized最后一个特性（缺点）就是不可中断性，在所有等待的线程中，你们唯一能做的就是等，而实际情况可能是有些任务等了足够久了，我要取消此任务去干别的事情，此时synchronized是无法帮你实现的，它把所有实现机制都交给了JVM，提供了方便的同时也体现出了自己的局限性。

这节主要介绍java的synchronized关键词，包括它的作用及JVM及操作系统的底层实现，它的可重入性和不可中断性，它实现的是一个非公平的互斥锁，同时它也是一个悲观的并发策略，不管是否会产生竞争，任何的数据操作都必须要加锁。对synchronized深入全面的理解对理解tomcat中跟多线程并发相关的模块是很有帮助的。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

乐观的并发策略——基于CAS的自旋

悲观者与乐观者的做事方式完全不一样，悲观者的人生观是一件事情我必须要百分之百完全控制才会去做，否则就认为这件事情一定会出问题；而乐观者的人生观则相反，凡事不管最终结果如何，他都会先尝试去做，大不了最后不成功。这就是悲观锁与乐观锁的区别，悲观锁会把整个对象加锁占为自有后才去做操作，乐观锁不获取锁直接做操作，然后通过一定检测手段决定是否更新数据。这一节将对乐观锁进行深入探讨。

上节讨论的Synchronized互斥锁属于悲观锁，它有一个明显的缺点，它不管数据存不存在竞争都加锁，随着并发量增加，且如果锁的时间比较长，其性能开销将会变得很大。有没有办法解决这个问题？答案是基于冲突检测的乐观锁。这种模式下，已经没有所谓的锁概念了，每条线程都直接先去执行操作，计算完成后检测是否与其他线程存在共享数据竞争，如果没有则让此操作成功，如果存在共享数据竞争则可能不断地重新执行操作和检测，直到成功为止，可叫CAS自旋。

乐观锁的核心算法是CAS（Compareand Swap，比较并交换），它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。如图2-5-4-1，有两个线程可能会差不多同时对某内存操作，线程二先读取某内存值作为预期值，执行到某处时线程二决定将新值设置到内存块中，如果线程一在此期间修改了内存块，则通过CAS即可以检测出来，假如检测没问题则线程二将新值赋予内存块。

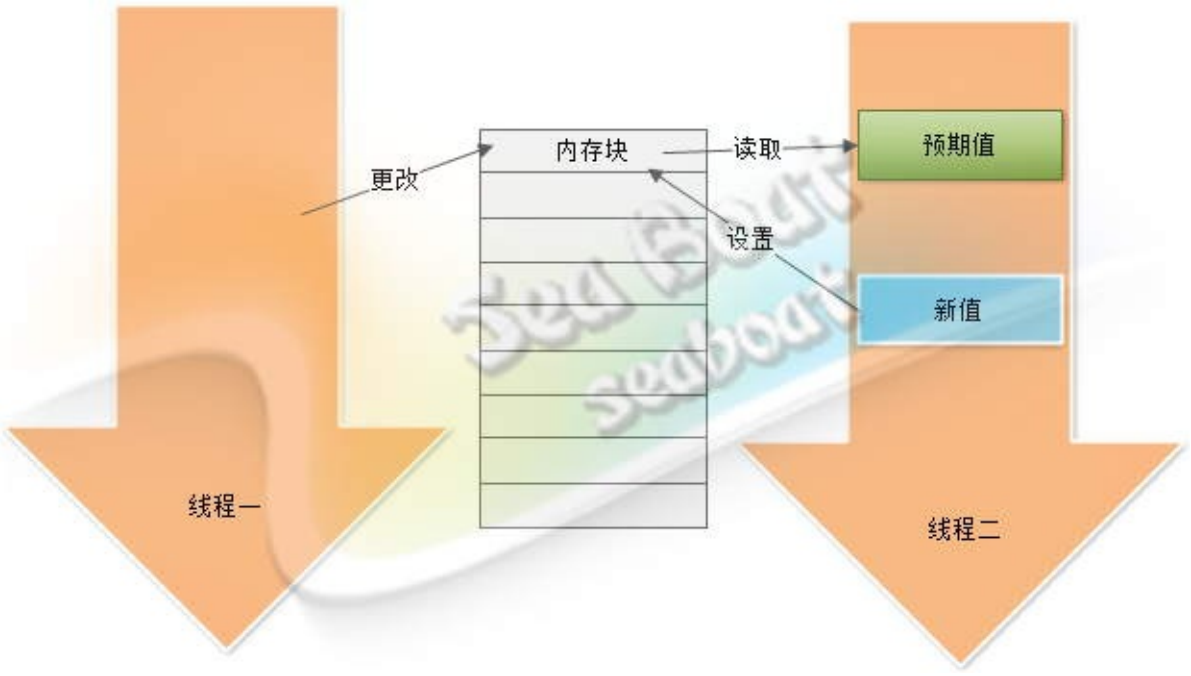


图2-5-4-1

假如你足够细心你可能会发现一个疑问，比较和交换，从字面上就有两个操作了，更别说实际CAS可能会有更多的执行指令，他们是原子性的吗？如果非原子性又怎么保证CAS操作期间出现并发带来的问题？我是不是需要用上节提到的互斥锁来保证他的原子性操作？CAS肯定是具有原子性的，不然就谈不上在并发中使用了，但这个原子性是由CPU硬件指令实现保证的，即使用JNI调用native方法调用由C++编写的硬件级别指令，jdk中提供了Unsafe类执行这些操作。另外，你可能想着CAS是通过互斥锁来实现原子性的，这样确实能实现，但用这种方式来保证原子性显示毫无意义。下面一个伪代码加深对CAS的理解：

```
public class AtomicInt {

  private volatile int value;

  public final int get() {

     return value;

   }

publicfinal int getAndIncrement() {

     for (;;) {

        int current = get();

        int next = current + 1;

        if (compareAndSet(current, next))

            return current;

     }

  }


  public final boolean compareAndSet(int expect, int update) {

    Unsafe类提供的硬件级别的compareAndSwapInt方法;

  }

}
```

其中最重要的方法是getAndIncrement方法，它里面实现了基于CAS的自旋。

现在已经了解乐观锁及CAS相关机制，乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，但

它也有缺点：

① 观锁只能保证一个共享变量的原子操作。如上例子，自旋过程中只能保证value变量的原子性，这时如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。

② 长时间自旋可能导致开销大。假如CAS长时间不成功而一直自旋，会给CPU带来很大的开销。

③ ABA问题。CAS的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是A，后来被一条线程改为B，最后又被改成了A，则CAS认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入版本号，每次变量更新都把版本号加一。

乐观锁是对悲观锁的改进，虽然它也有缺点，但它确实已经成为提高并发性能的主要手段，而且jdk中的并发包也大量使用基于CAS的乐观锁。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS之原子性如何保证？

在研究AQS框架时，会发现这个类很多地方都使用了CAS操作，在并发实现中CAS操作必须具备原子性，而且是硬件级别的原子性，java被隔离在硬件之上，明显力不从心，这时为了能直接操作操作系统层面，肯定要通过用C++编写的native本地方法来扩展实现。JDK提供了一个类来满足CAS的要求，sun.misc.Unsafe，从名字上可以大概知道它用于执行低级别、不安全的操作，AQS就是使用此类完成硬件级别的原子操作。

Unsafe是一个很强大的类，它可以分配内存、释放内存、可以定位对象某字段的位置、可以修改对象的字段值、可以使线程挂起、使线程恢复、可进行硬件级别原子的CAS操作等等，但平时我们没有这么特殊的需求去使用它，而且必须在受信任代码（一般由JVM指定）中调用此类，例如直接Unsafe unsafe = Unsafe.getUnsafe();获取一个Unsafe实例是不会成功的，因为这个类的安全性很重要，设计者对其进行了如下判断，它会检测调用它的类是否由启动类加载器Bootstrap ClassLoader（它的类加载器为null）加载，由此保证此类只能由JVM指定的类使用。

```
public static Unsafe getUnsafe() {

  Class cc = sun.reflect.Reflection.getCallerClass(2);

  if (cc.getClassLoader() != null)

     throw new SecurityException("Unsafe");

  return theUnsafe;

   }
```

当然可以通过反射绕过上面的限制，用下面的getUnsafeInstance方法可以获取Unsafe实例，这段代码演示了如何获取java对象的相对地址偏移量及使用Unsafe完成CAS操作，最终输出的是flag字段的内存偏移量及CAS操作后的值。分别为8和101。另外如果使用开发工具如Eclipse，可能会编译通不过，只要把编译错误提示关掉即可。

```
public class UnsafeTest {

privateint flag = 100;

privatestatic long offset;

privatestatic Unsafe unsafe = null;

static{

    try{
```

```
        unsafe= getUnsafeInstance();
        offset= unsafe.objectFieldOffset(UnsafeTest.class
                .getDeclaredField("flag"));
    }catch (Exception e) {
        e.printStackTrace();
    }
}
publicstatic void main(String[] args) throws Exception {
    intexpect = 100;
    intupdate = 101;
    UnsafeTestunsafeTest = new UnsafeTest();
    System.out.println("unsafeTest对象的flag字段的地址偏移量为："+offset);
    unsafeTest.doSwap(offset,expect, update);
    System.out.println("CAS操作后的flag值为：" +unsafeTest.getFlag());
}
privateboolean doSwap(long offset, int expect, int update) {
    returnunsafe.compareAndSwapInt(this, offset, expect, update);
}
publicint getFlag() {
    returnflag;
}
privatestatic Unsafe getUnsafeInstance() throws SecurityException,
        NoSuchFieldException,IllegalArgumentException,
        IllegalAccessException{
```

```
    FieldtheUnsafeInstance = Unsafe.class.getDeclaredField("theUnsafe");
    theUnsafeInstance.setAccessible(true);
    return(Unsafe) theUnsafeInstance.get(Unsafe.class);
}
}
```

Unsafe类让我们明白了java是如何实现对操作系统操作的，一般我们使用java是不需要在内存中处理java对象及内存地址位置的，但有的时候我们确实需要知道java对象相关的地址，于是我们使用Unsafe类，尽管java对其提供了足够的安全管理。

Java语言的设计者们极力隐藏涉及底层操作系统的相关操作，但此节我们本着对AQS框架实现的目的，不得不剖析了Unsafe类，因为AQS里面即是使用Unsafe获取对象字段的地址偏移量、相关原子操作来实现CAS操作的。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS之如何使用AQS构建同步器

AQS的设计思想是通过继承的方式提供一个模板让大家可以很容易根据不同场景实现一个富有个性化的同步器。同步器的核心是要管理一个共享状态，通过对状态的控制即可以实现不同的锁机制。AQS的设计必须考虑把复杂重复且容易出错的队列管理工作统一抽象出来管理，并且要统一控制好流程，而暴露给子类调用的方法主要就是操作共享状态的方法，以此提供对状态的原子性操作。一般子类的同步器中使用AQS提供的getState、setState、compareAndSetState三个方法，前两个为普通的get和set方法，要使用这两个方法必须要保证不存在数据竞争，compareAndSetState方法提供了CAS方式的硬件级别的原子更新。对于独占模式而言，锁获取与释放的流程的定义则交给acquire和release两个方法，它们定义了锁获取与释放的逻辑，同时也是提供给子类获取和释放锁的接口。它的执行逻辑可以参考前面的“锁的获取与释放”，它提供了一个怎样强大的模板？由下面的伪代码可以清晰展示出来，请注意tryAcquire和tryRelease这两个方法，它就是留给子类实现个性化的方法，通过这两个方法对共享状态的管理可以自定义多种多样的同步器，而队列的管理及流程的控制则不是你需要考虑的问题。

①锁获取模板

```
if(tryAcquire(arg)) {
  创建node
  使用CAS方式把node插入到队列尾部
  while(true){
  if(tryAcquire(arg) 并且 node的前驱节点为头节点){
把当前节点设置为头节点
  跳出循环
}else{
  使用CAS方式修改node前驱节点的waitStatus标识为signal
  if(修改成功)
      挂起当前线程
}
}
```

②锁释放模板

```
  if(tryRelease(arg)){
唤醒后续节点包含的线程
}
```

我们可以认为同步器可实现任何不同锁的语义，一般提供给使用者的锁是用AQS框架封装实现的更高层次的实现，提供一种更加形象的API让使用者使用起来更加方便简洁，而不是让使用者直接接触AQS框架，例如，ReentrantLock、Semphore、CountDownLatch等等，这些不同的形象的锁让你使用起来更好理解更加得心应手，而且不容易混淆。然而这些锁都是由AQS实现，AQS同步器面向的是线程和状态的控制，定义了线程获取状态的机制及线程排队等操作，很好地隔离了两者的关注点，高层关注的是场景的使用，而AQS同步器则关心的是并发的控制。假如你要实现一个自定义同步装置，官方推荐的做法是将集成

AQS同步器的子类作为同步装置的内部类，而同步装置中相关的操作只需代理成子类中对应的方法即可。往下用一个简单的例子看看如何实现自己的锁，由于同步器被分为两种模式，独占模式和共享模式，所以例子也对应给出。

①独占模式，独占模式采取的例子是银行服务窗口，假如某个银行网点只有一个服务窗口，那么此银行服务窗口只能同时服务一个人，其他人必须排队等待，所以这种银行窗口同步装置是一个独占模型。第一个类是银行窗口同步装置类，它按照推荐的做法使用一个继承AQS同步器的子类实现，并作为子类出现。第二个类是测试类，形象一点地说，有三位良民到银行去办理业务，分别是tom、jim和jay，我们使用BankServiceWindow就可以约束他们排队，一个一个轮着办理业务而避免陷入混乱的局面。

```
public class BankServiceWindow {
private final Sync sync;
public BankServiceWindow() {
sync = new Sync();
}
private static class Sync extends AbstractQueuedSynchronizer {
public boolean tryAcquire(int acquires) {
if (compareAndSetState(0, 1)) {
setExclusiveOwnerThread(Thread.currentThread());
return true;
}
return false;
}
protected boolean tryRelease(int releases) {
if (getState() == 0)
throw new IllegalMonitorStateException();
setExclusiveOwnerThread(null);
setState(0);
return true;
}
}
public void handle() {
sync.acquire(1);
}
public void unhandle() {
sync.release(1);
}
}

public class BankServiceWindowTest {
```

```
public static void main(String[] args){
final BankServiceWindow bankServiceWindow=new BankServiceWindow();
Thread tom=new Thread(){
public void run(){
bankServiceWindow.handle();
System.out.println("tom开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("tom结束办理业务");
bankServiceWindow.unhandle();
}
};
Thread jim=new Thread(){
public void run(){
bankServiceWindow.handle();
System.out.println("jim开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("jim结束办理业务");
bankServiceWindow.unhandle();
}
};
Thread jay=new Thread(){
public void run(){
bankServiceWindow.handle();
System.out.println("jay开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("jay结束办理业务");
bankServiceWindow.unhandle();
```

```
 }
 };
 tom.start();
 jim.start();
 jay.start();
  }
}
```

输出结果如下：

```
tom开始办理业务
tom结束办理业务
jim开始办理业务
jim结束办理业务
jay开始办理业务
jay结束办理业务
```

明显tom、jim、jay仨是排队完成的，但是无法保证三者的顺序，可能是tom、jim、jay，也可能是tom、jay、jim，因为在入列以前的执行先后是无法确定的，它的语义是保证一个接一个办理。如果没有同步器限制的情况，输出结果将不可预测，可能为输出如下：

```
jim开始办理业务
jay开始办理业务
tom开始办理业务
jay结束办理业务
jim结束办理业务
tom结束办理业务
```

②共享模式，共享模式采取的例子同样是银行服务窗口，随着此网点的发展，办理业务的人越来越多，一个服务窗口已经无法满足需求，于是又分配了一位员工开了另外一个服务窗口，这时就可以同时服务两个人了，但两个窗口都有人占用时同样也必须排队等待，这种服务窗口同步器装置就是一个共享型。第一个类是共享模式的同步装置类，跟独占模式不同的是它的状态的初始值可以自由定义，获取与释放就是对状态递减和累加操作。第二个类是测试类，tom、jim和jay再次来到银行，一个有两个窗口甚是高兴，他们可以两个人同时办理了，时间缩减了不少。

```
public class BankServiceWindows {
private final Sync sync;
public BankServiceWindows(int count) {
sync = new Sync(count);
}
private static class Sync extends AbstractQueuedSynchronizer {
Sync(int count) {
setState(count);
}
```

```
public int tryAcquireShared(int interval) {
for (;;) {
int current = getState();
int newCount = current - 1;
if (newCount < 0 || compareAndSetState(current, newCount)) {
return newCount;
}
}
}
public boolean tryReleaseShared(int interval) {
for (;;) {
int current = getState();
int newCount = current + 1;
if (compareAndSetState(current, newCount)) {
return true;
}
}
}
}

public void handle() {
sync.acquireShared(1);
}

public void unhandle() {
sync.releaseShared(1);
}

}

public class BankServiceWindowsTest {
public static void main(String[] args){
final BankServiceWindows bankServiceWindows=new BankServiceWindows(2);
Thread tom=new Thread(){
public void run(){
bankServiceWindows.handle();
System.out.println("tom开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
```

```
e.printStackTrace();
}
System.out.println("tom结束办理业务");
bankServiceWindows.unhandle();
}
};
Thread jim=new Thread(){
public void run(){
bankServiceWindows.handle();
System.out.println("jim开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("jim结束办理业务");
bankServiceWindows.unhandle();
}
};
Thread jay=new Thread(){
public void run(){
bankServiceWindows.handle();
System.out.println("jay开始办理业务");
try {
this.sleep(5000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("jay结束办理业务");
bankServiceWindows.unhandle();
}
};
tom.start();
jim.start();
jay.start();
}
}
```

可能的输出结果为：

```
tom开始办理业务
```

jay开始办理业务
jay结束办理业务
tom结束办理业务
jim开始办理业务
jim结束办理业务
tom和jay几乎同时开始办理业务，而jay结束后有空闲的服务窗口jim才过来。
这节主要讲的是如何使用AQS构建自己的同步器，并且剖析了锁获取与释放的模板的逻辑，让你更好理解AQS的实现，最后分别给出了独占模式和共享模式的同步器实现的例子，相信你们搞清楚这两种方式的实现后，要构建更加复杂的同步器就知道力往哪里使了。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java多线程模型

谈到Java多线程就涉及到多线程的模型及Java线程与底层操作系统之间的关系。正如我们熟知，现代机器可以分为硬件和软件两大块，如图2-5-1-1，硬件是基础，软件提供实现不同功能的手段。而且软件可以分为操作系统和应用程序，操作系统专注于对硬件的交互管理并提供一个运行环境给应用程序使用，应用程序则是能实现若干功能的并且运行在操作系统环境中的软件。同样，线程按照操作系统和应用程序两层次可以分为内核线程(Kernel Thread)和用户线程(User Thread)。

图2-5-1-1

所谓内核线程就是直接由操作系统内核支持和管理的线程，线程的建立、启动、同步、销毁、切换等操作都由内核完成。基本所有的现代操作系统都支持内核线程。用户线程指完全建立在用户空间的线程库上，由内核支持而无需内核管理，内核也无法感知用户线程的存在，线程的建立、启动、同步、销毁、切换完全在用户态完成，无需切换到内核。可以把用户线程看成是更高层面的线程，而内核线程则是最底层的支持，那么他们之间必然存在一定的映射关系，一般有三种常用的关系，下面将逐个列出。

① 一对一模型

一对一模型可以说是最简单的映射模型，如图2-5-1-2，KT为内核线程，UT为用户线程，每个用户线程都对应一个内核线程，由于每个用户线程都有各自的内核线程，所以他们互不影响，即使其中一个线程阻塞，也允许另一个线程继续执行，这无疑是此模型的优点，但也存在一个严重的缺陷，由于一对一的关系，有多少个用户线程就代表有多少个内核线程，而内核线程的开销较大，一般操作系统都会有内核线程数量的限制，用户线程的数量也被限制。

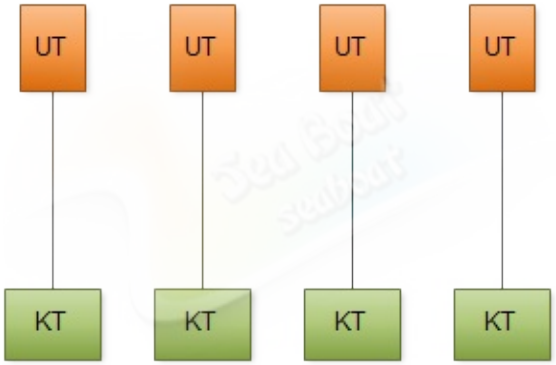


图2-5-1-2

② 多对一模型

第二种是多对一模型，如图2-5-1-3，可以清晰看到多个用户线程映射到同一个内核线程上，可以看成由一条内核线程实现若干个用户线程的并发功能，线程的管理在用户空间中进行，一般不需要切换到内核态，效率较高，而且比起一对一模型，支持的线程数量更大。但此模型有个致命的弱点是如果一个线程执行了阻塞调用，所有线程都将阻塞，并且任意时刻只能有一个线程访问内核。另外，对线程的所有操作都将由用户应用自己处理。所以一般除了在不支持多线程的操作系统被迫使用此模型外，在多线程操作系统中基本不使用。

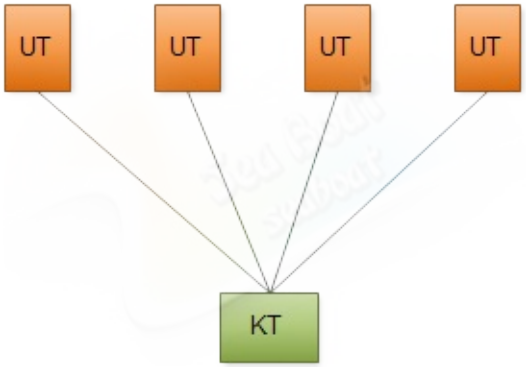


图2-5-1-3

③ 多对多模型

多对多模型的提出是为了解决前面两种模型的缺点，如图2-5-1-4，多个用户线程与多个内核线程映射形成多路复用。前面提到的一对一模型存在受内核线程数量限制的问题，多对一模型虽然解决了数量限制问题，但它存在一个线程阻塞导致所有线程阻塞的风险，而且由于一个内核线程只能调度一个线程导致并发性不强。看看多对多模型如何解决这些问题，由于多对一是多对多的子集，所以多对多具备多对一的优点，线程数不受限制。除此之外，多个内核线程可处理多个用户线程，当某个线程阻塞时，将可以调度另外一个线程执行，这从另一方面看也是增强了并发性。

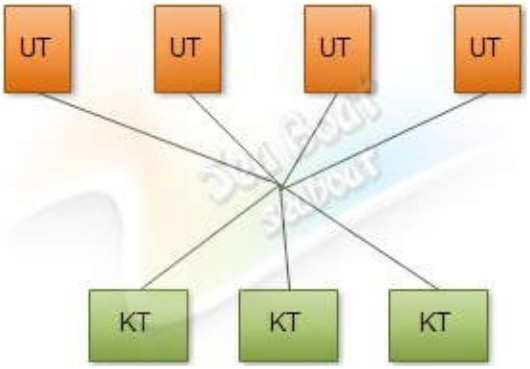


图2-5-1-4

三种模型各自有各自的特点，不同的现代操作系统可能使用不同的线程模型，例如linux和windows可能使用了一对一模型，而solaris和unix某些版本可能使用多对多模型。对于线程的创建和管理主要由线程库提供用户级别和内核级别两种API进行操作。用户级别由于不涉及内核操作，所有代码和数据结构均存放在用户空间，与此相反，内核级别由内核支持，将直接调用内核系统操作，代码和数据结构存在与内核空间中。在实际程序中我们一般不直接使用内核线程，用户线程与内核线程直接需要一种中间数据结构，它由内核支持且是内核线程的高级抽象，这个高级接口被称为轻量级进程（Light Weight Process），下面简称LWP。图2-5-1-5是三种模型增加了轻量级进程的示意图，从某种层面上看，LWP最多算是广义的用户线程，并非狭义定义的用户进程，LWP线程库是以内核为基础，很多操作要进行内核调用，效率不高，如果要快速低消耗的操作则需要一个纯粹的用户线程，线程库完全建立在用户空间。于是可以看到一个进程P里面一般包含若干个用户进程，用户进程以某种关系对应轻量级进程，而轻量级进程则是内核线程的高级体现。如此一来，一个内核线程堵塞将导致LWP也阻塞，与LWP相连的用户线程也将阻塞。

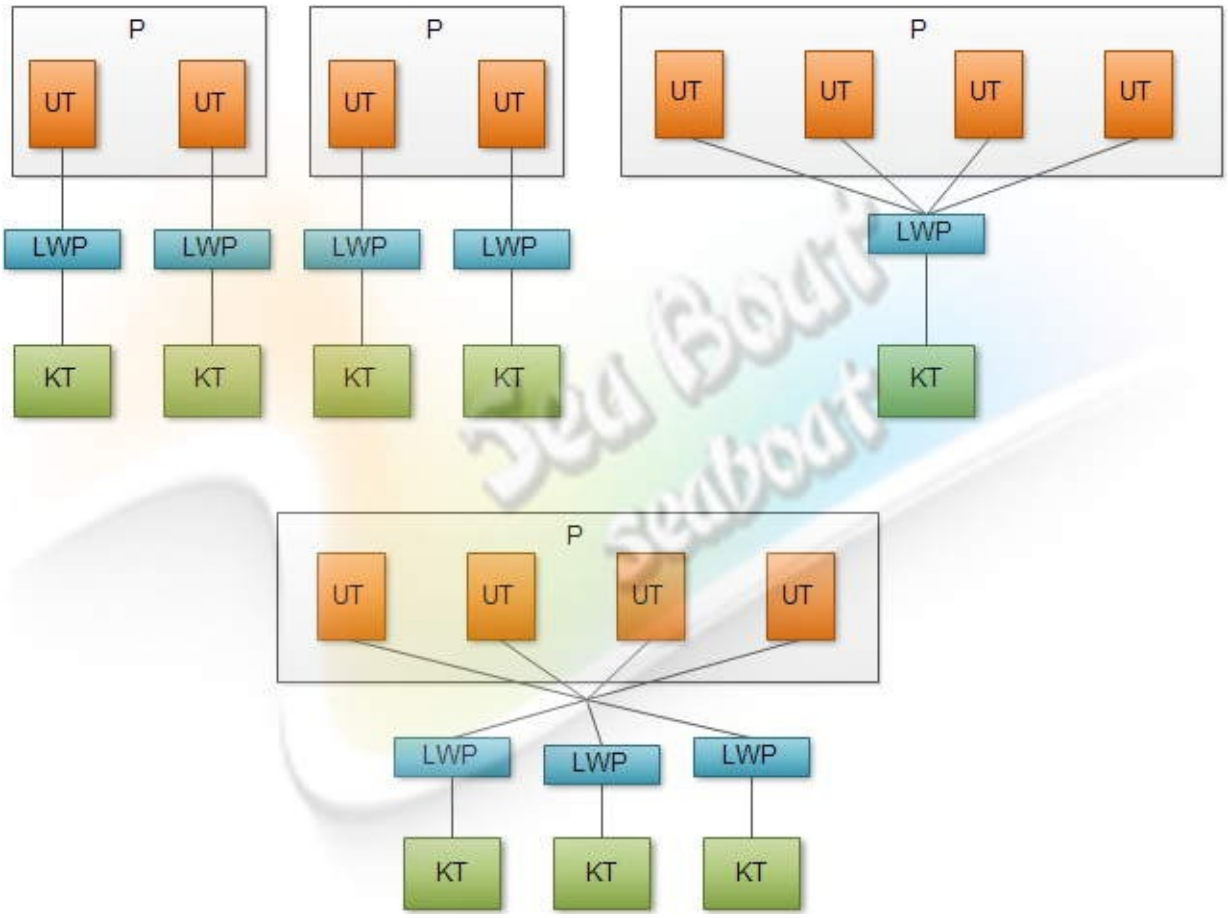


图2-5-1-5

最后要谈谈Java线程与底层操作系统的关系，由于Java通过JVM封装了底层操作系统的差异，所以Java线程也必然是要封装不同操作系统提供一个统一的并发定义，在JDK发展历史上，java语言开发者曾经通过一类叫“绿色线程（Green Threads）”的用户线程进行实现Java线程，但从jdk1.2开始，java线程使用操作系统原生线程模型实现，也就是说Java线程的实现通过不同操作系统提供的线程库分别实现，JVM根据不同操作系统的线程模型对Java线程进行映射，假如Java运行在windows系统上，它通常直接使用Win32 API实现多线程，假如Java运行在linux系统则直接使用Pthread线程库实现多线程。这样一来就顺利隐藏了底层实现细节，提供给开发者就是一个具有统一抽象的线程语义。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——什么是AQS框架

什么是AQS框架
1995年sun公司发布了第一个java语言版本，可以说从jdk1.1到jdk1.4期间java的使用主要是在移动应用和中小型企业应用中，在此类领域中基本不用设计大型并发场景，当然也没有大型互联网公司使用java，因为担心它本身的性能。在互联网及服务器硬件迅猛的发展下，sun公司更加注重企业级应用方面，毫无疑问高并发是一个重要的主题，于是在J2SE5.0（jdk1.5）代号为老虎的版本中增加了更加强大的并发相关的操作包——java.util.concurrent。此后java在高并发中表现优异，很多大型互联网公司都使用java作为主要开发语言，例如阿里巴巴、ebay等，这些公司系统的访问绝对是属于世界级的大型并发场景，反映了java在大型并发场景是可行的。Jdk的并发包提供了各种锁及同步机制，其实现的核心类是AbstractQueuedSynchronizer，我们简称为AQS框架，它为不同场景提供了实现锁及同步机制的基本框架，为同步状态的原子性管理、线程的阻塞、线程的解除阻塞及排队管理提供了一种通用的机制。

Jdk的并发包（juc）的作者是Doug Lea，但其中思想却是结合了多位大师的智慧，如果你想深入理解juc的相关理论可以参考Doug Lea写的《The_java.util.concurrent_Synchronizer_Framework》论文。从这里可以找到AQS的理论基础，包括框架的基本原理、需求、设计、实现思路、用法及性能，由于这些方面篇幅较大，本文不打算涉及所有方面，主要将针对AQS类的结构及相关操作进行分析。

ASQ将线程封装到一个Node里面，并维护一个CHL Node FIFO队列，它是一个非阻塞的FIFO队列，也就是说在并发条件下往此队列做插入或移除操作不会阻塞，是通过自旋锁和CAS保证节点插入和移除的原子性，实现无锁快速插入。

其实AbstractQueuedSynchronizer主要就是维护了一个state属性、一个FIFO队列和线程的阻塞与解除阻塞操作。state表示同步状态，它的类型为32位整型，对state的更新必须要保证原子性。这里的队列是一个双向链表，每个节点里面都有一个prev和next，它们分别是前一个节点和后一个节点的引用。需要注意的是此双向链表除了链头其他每个节点内部都包含一个线程，而链头可以理解为一个空节点。

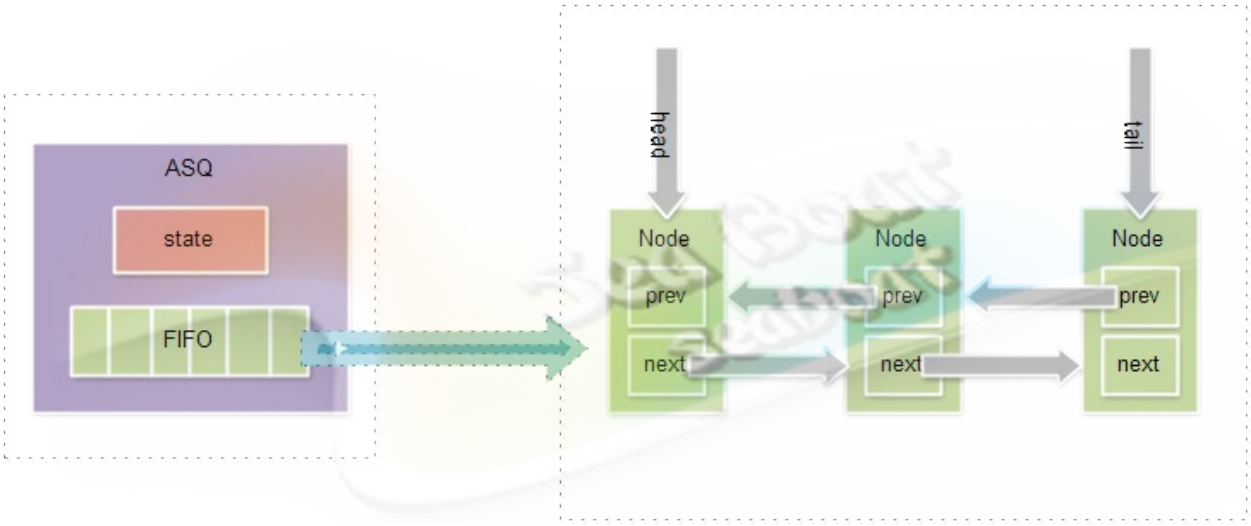


图2-5-5-1

对于队列的结构我们需要深入理解下，图2-5-5-2展示的是组成双向链表其中一个节点的结构,该节点包含五个主要元素，表示的意思如下表，

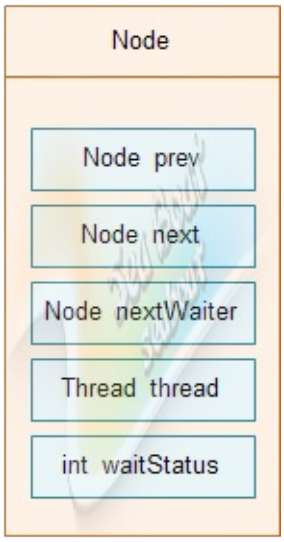


图2-5-5-2

属性	含义
Node prev	前驱节点，指向前一个节点
Node next	后续节点，指向后一个节点
Node nextWaiter	用于存储condition队列的后续节点
Thread thread	入队列时的当前线程
int waitStatus	有五种状态： ① SIGNAL,值为-1，表示当前节点的后续节点中的线程通过park被阻塞了，当前节点在释放或取消时要通过unpark解除它的阻塞。 ② CANCELLED，值为1，表示当前节点的线程因为超时或中断被取消了。 ③ CONDITION，值为-2，表示当前节点在condition队列中。 ④ PROPAGATE，值为-3，共享模式的头结点可能处于此状态，表示无条件往下传播，引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒。 ⑤ 0，除了以上四种状态的第五种状态，一般是节点初始状态。

前驱节点prev的引入主要是为了完成超时及取消语义，前驱节点取消后只需向前找到一个未取消的前驱节点即可；后续节点的引入主要是为了优化后续节点的查找，避免每次从尾部向前查找；nextWaiter用于表示condition队列的后续节点，此时prev和next属性将不再使用，而且节点状态处于Node.CONDITION;waitStatus表示的是后续节点状态，这是因为AQS中使用CLH队列实现线程的结构管理，而CLH结构正是用前一节点某一属性表示当前节点的状态，这样更容易实现取消和超时功能。

上面是对节点及节点组成队列的结构的介绍，接着介绍AQS相关的一些操作，包括锁的获取与释放、队列的管理、同步状态的更新、线程阻塞与唤醒、取消中断与超时中断等等。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java多线程的调度策略

在Java多线程环境中，为保证所有线程的执行能按照一定的规则执行，JVM实现了一个线程调度器，它定义了线程调度的策略，对于CPU运算的分配都进行了规定，按照这些特定的机制为多个线程分配CPU的使用权。这小节关注线程如何进行调度，了解了java线程调度模式有助于后面并发框架的深入探讨。

一般线程调度模式分为两种——抢占式调度和协同式调度。抢占式调度指的是每条线程执行的时间、线程的切换都由系统控制，系统控制指的是在系统某种运行机制下，可能每条线程都分同样的执行时间片，也可能是某些线程执行的时间片较长，甚至某些线程得不到执行的时间片。在这种机制下，一个线程的堵塞不会导致整个进程堵塞。协同式调度指某一线程执行完后主动通知系统切换到另一线程上执行，这种模式就像接力赛一样，一个人跑完自己的路程就把接力棒交接给下一个人，下个人继续往下跑。线程的执行时间由线程本身控制，线程切换可以预知，不存在多线程同步问题，但它有一个致命弱点：如果一个线程编写有问题，运行到一半就一直堵塞，那么可能导致整个系统崩溃。

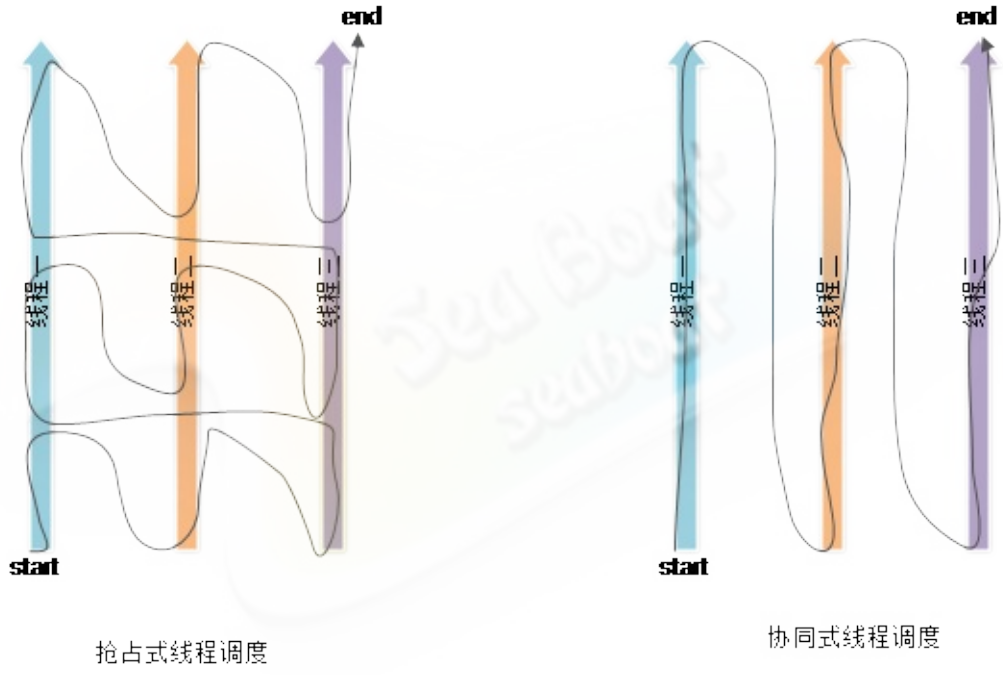


图2-5-6-1

为更加形象说明两种模式的不同，看图2-5-6-1，左边为抢占式线程调度，假如三条线程需要运行，处理器运行的路径是在线程一运行一个时间片后强制切换到线程二运行一个时间片，然后切到线程三，再回到线程一，如此循环直至三条线程都执行完。而协同式线程调度则不这样走，它会先将线程一执行完，线程一再通知线程二执行，线程二再通知线程三，直到线程三执行完。

在了解了两种线程调度模式后，现在考虑Java使用的是哪种线程调度模式。此问题的讨论涉及到JVM的实

现，JVM规范中规定每个线程都有优先级，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。JVM的规范没有严格地给调度策略定义，我想正是因为面对众多不同调度策略，JVM要封装所有细节提供一个统一的策略不太现实，于是给了一个不严谨但足够统一的定义。回到问题上，Java使用的线程调度是抢占式调度，在JVM中体现为让可运行池中优先级高的线程拥有CPU使用权，如果可运行池中线程优先级一样则随机选择线程，但要注意的是实际上一个绝对时间点只有一个线程在运行（这里是相对于一个CPU来说，如果你的机器是多核的还是可能多个线程同时运行的），直到此线程进入非可运行状态或另一个具有更高优先级的线程进入可运行线程池，才会使之让出CPU的使用权，更高优先级的线程抢占了优先级低的线程的CPU。

Java中线程会按优先级分配CPU时间片运行，那么线程什么时候放弃CPU的使用权？可以归类成三种情况：

1. 当前运行线程主动放弃CPU，JVM暂时放弃CPU操作（基于时间片轮转调度的JVM操作系统不会让线程永久放弃CPU，或者说放弃本次时间片的执行权），例如调用yield()方法。
2. 当前运行线程因为某些原因进入阻塞状态，例如阻塞在I/O上。
3. 当前运行线程结束，即运行完run()方法里面的任务。

三种情况中第三种很好理解，任务执行完了自然放弃CPU，前两种情况用两个例子说明，先看使用yield放弃CPU什么情况：

```
public class YeildThread {

publicstatic void main(String[] args) {

  MyThreadmt = new MyThread();

  mt.start();

  while(true) {

      System.out.println("主线程");

  }

}

}

class MyThread extends Thread {

publicvoid run() {
```

```
    while(true) {
        System.out.println("被放弃线程");
        Thread.currentThread().yield();
    }
}
}
```

截取某段输出如下，输出“主线程”比“被放弃线程”运行的机会多，因为mt线程每次循环都把时间片让给主线程，正是因为yield操作并不会永远放弃CPU，仅仅只是放弃了此次时间片，把剩下的时间让给别的线程，

主线程

主线程

主线程

主线程

主线程

主线程

被放弃线程

主线程

主线程

主线程

主线程

主线程

主线程

主线程

第二个例子为节省代码量将使用伪代码表示，例子简单但已能说明问题，运行程序将有两条线程工作，ioThread每次遇到I/O阻塞就放弃当前的时间片，而主线程则按JVM分配的时间片正常运行。

```
public class IOBlockThread {

publicstatic void main(String[] args) {

    IOThread ioThread = new IOThread();

    ioThread.start();

   主线程任务执行

}

}

class IOThread extends Thread {

publicvoid run() {

   while(true) {

       I/O阻塞

   }

}

}
```

Java的线程的调度机制都由JVM实现，假如有若干条线程，你想让某些线程拥有更长的执行时间，或某些线程分配少点执行时间，这时就涉及“线程优先级”，Java把线程优先级分成10个级别，线程被创建时如果没有明确声明则使用默认优先级，JVM将根据每个线程的优先级分配执行时间的概率。有三个常量Thread.MIN_PRIORITY、Thread.NORM_PRIORITY、Thread.MAX_PRIORITY分别表示最小优先级值（1）、默认优先级值（5）、最大优先级值（10）。

由于JVM的实现以宿主操作系统为基础，所以Java优先级值与各种不同操作系统的原生线程优先级必然存在某种映射关系，这样才足以封装所有操作系统的优先级提供统一优先级语义。例如1-10优先级值在linux可能要与0-99优先级值进行映射，而windows系统则有7个优先级要映射。

线程的调度策略决定上层多线程运行机制，JVM的线程调度器实现了抢占式调度，每条线程执行的时间由它分配管理，它将按照线程优先级的建议对线程执行的时间进行分配，优先级越高，可能得到CPU的时间则越长。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS中断的支持

线程的定义给我们提供了并发执行多个任务的方式，大多数情况下我们会让每个任务都自行执行结束，这样能保证事务的一致性，但是有时我们希望在任务执行中取消任务，使线程停止。在java中要让线程安全、快速、可靠地停下来并不是一件容易的事，java也没有提供任何可靠的方法终止线程的执行。回到第六小节，线程调度策略中有抢占式和协作式两个概念，与之类似的是中断机制也有协作式和抢占式。

历史上Java曾经使用stop()方法终止线程的运行，他们属于抢占式中断。但它引来了很多问题，早已被JDK弃用。调用stop()方法则意味着①将释放该线程所持的所有锁，而且锁的释放不可控。②即刻将抛出ThreadDeath异常，不管程序运行到哪里，但它不总是有效，如果存在被终止线程的锁竞争；第一点将导致数据一致性问题，这个很好理解，一般数据加锁就是为了保护数据的一致性，而线程停止伴随所持锁的释放，很可能导致被保护的数据呈现不一致性，最终导致程序运算出现错误。第二点比较模糊，它要说明的问题就是可能存在某种情况stop()方法不能及时终止线程，甚至可能终止不了线程。看如下代码会发生什么情况，看起来线程mt因为执行了stop()方法将停止，按理来说就算execut方法是一个死循环，只要执行了stop()方法线程将结束，无限循环也将结束。其实不然，因为我们在execute方法使用了synchronized修饰，同步方法表示在执行execute时将对mt对象进行加锁，另外，Thread的stop()方法也是同步的，于是在调用mt线程的stop()方法前必须获取mt对象锁，但mt对象锁被execute方法占用，且不释放，于是stop()方法永远获取不了mt对象锁，最后得到一个结论，使用stop()方法停止线程不可靠，它未必总能有效终止线程。

```
public class ThreadStop {

public static voidmain(String[] args) {

        Thread mt= new MyThread();

        mt.start();

        try {

                Thread.currentThread().sleep(100);

        } catch(InterruptedException e) {

                e.printStackTrace();

        }

        mt.stop();

}
```

```
static classMyThread extends Thread {
        publicvoid run() {
                execute();
        }
        privatesynchronized void execute() {
                while(true) {
                }
        }
}
}
```

经历了很长时间的发展，Java最终选择用一种协作式的中断机制实现中断。协作式中断的原理很简单，其核心是先对中断标识进行标记，某线程设置某线程的中断标识位，被标记了中断位的线程在适当的时间节点会抛出异常，捕获异常后做相应的处理。实现协作中断有三个要点需要考虑：①是在Java层面实现轮询中断标识还是在JVM中实现；②轮询的颗粒度的控制，一般颗粒度要尽量小周期尽量短以保证响应的及时性；③轮询的时间节点的选择，其实就是在哪些方法里面轮询，例如JVM将Thread类的wait()、sleep()、join()等方法都实现中断标识的轮询操作。

中断标识放在哪里？中断是针对线程实例而言，从Java层面上看，标识变量放到线程中肯定再合适不过了，但由于由JVM维护，所以中断标识具体由本地方法维护。在Java层面仅仅留下几个API用于操作中断标识，如下，

```
public class Thread{
  public voidinterrupt() {……}
  public BooleanisInterrupted() {……}
  public static Booleaninterrupted() {……}
}
```

上面三个方法依次用于设置线程为中断状态、判断线程状态是否中断、清除当前线程中断状态并返回它之前的值。通过interrupt()方法设置中断标识，假如在非阻塞线程则仅仅只是改变了中断状态，线程将继续往下运行，但假如在可取消阻塞线程中，如正在执行sleep()、wait()、join()等方法的线程则会因为被设置了中断状态而抛出InterruptedException异常，程序对此异常捕获处理。

上面提到的三个要点，第一是轮询在哪个层面实现，这个没有特别的要求，在实际中只要不出现逻辑问题，在Java层面或JVM层面实现都是可以的，例如常用的线程睡眠、等待等操作是通过JVM实现，而AQS框架里面的中断则放到Java实现，不管在哪个层面上去实现，在轮询过程中都一定要能保证不会产生阻塞。第二是要保证轮询的颗粒度尽可能的小周期尽可能短，这关系到中断响应的速度。第三点是关于轮询的时间节点的选取。

针对三要点来看看AQS框架中是如何支持中断的，主要在等待获取锁的过程中提供中断操作，下面是伪代码。只需增加加红加粗部分逻辑即可实现中断支持，在循环体中每次循环都对当前线程中断标识位进行判断，一旦检查到线程被标记为中断则抛出InterruptedException异常，高层代码对此异常捕获处理即完成中断处理。总结起来就是ASQ框架获取锁的中断机制是在Java层面实现的，轮询时间节点选择在不断做尝试获取锁操作过程中，每个循环的颗粒度比较小，响应速度得以保证，且循环过程不存在阻塞风险，保证中断检测不会失效。

if(尝试获取锁失败) {

创建node

使用CAS方式把node插入到队列尾部

while(true){

if(尝试获取锁成功并且 node的前驱节点为头节点){

把当前节点设置为头节点

跳出循环

}else{

使用CAS方式修改node前驱节点的waitStatus标识为signal

if(修改成功){

挂起当前线程

if(当前线程中断位标识为true)

抛出InterruptedException异常

}

}

}

判断线程是否处于中断状态其实很简单，只需使用Thread.interrupted()操作，如果为true则说明线程处

于中断位，并清除中断位。至此AQS实现了支持中断的获取锁操作。

此节从java发展过程分析了抢占式中断及协作式中断，由于抢占式存在一些缺陷现在已不推荐使用，而协作式中断作为推荐做法，尽管在响应时间较长，但其具有无可比拟的优势。协作式中断我们可以在JVM层面实现，同样也可以在Java层面实现，例如AQS框架的中断即是在Java层面实现，不过如果继续深究是因为Java留了几个API供我们操作线程的中断标识位，这才使Java层面实现中断操作得以实现。对于java的协作式中断机制有人肯定有人批评，批评者说java没有抢占式中断机制，且协作式中断机制迫使开发者必须维护中断状态，迫使开发者必须处理InterruptedException。但肯定者则认为，虽然协作式中断机制推迟了中断请求的处理，但它为开发人员提供更灵活的中断处理策略，响应性可能不及抢占式，但程序健壮性更强。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java线程状态

线程跟人类一样拥有自己的生命周期，一条线程从创建到执行完毕的过程即是线程的生命周期，此过程可能在不同时刻处于不同的状态，线程状态正是这小节的主题，线程到底有多少种状态？不同状态之间是如何转化的？

对于线程的状态的分类并没有严格的规定，只要能正确表示状态即可，如图2-5-7-1，先看其中一种状态分类，一个线程从创建到死亡可能会经历若干个状态，但在任意一个时间点线程只能处于其中一种状态，总共包含五个状态：新建（new）、可运行（runnable）、运行（running）、非可运行（not runnable）、死亡（dead）。线程的状态的转化可以由程序控制，通过某些API可以达到转化效果，例如Thread类的start、stop、sleep、suspend、resume、wait、notify等方法（stop、suspend、resume等方法因为容易引起死锁问题而早已被弃用）。

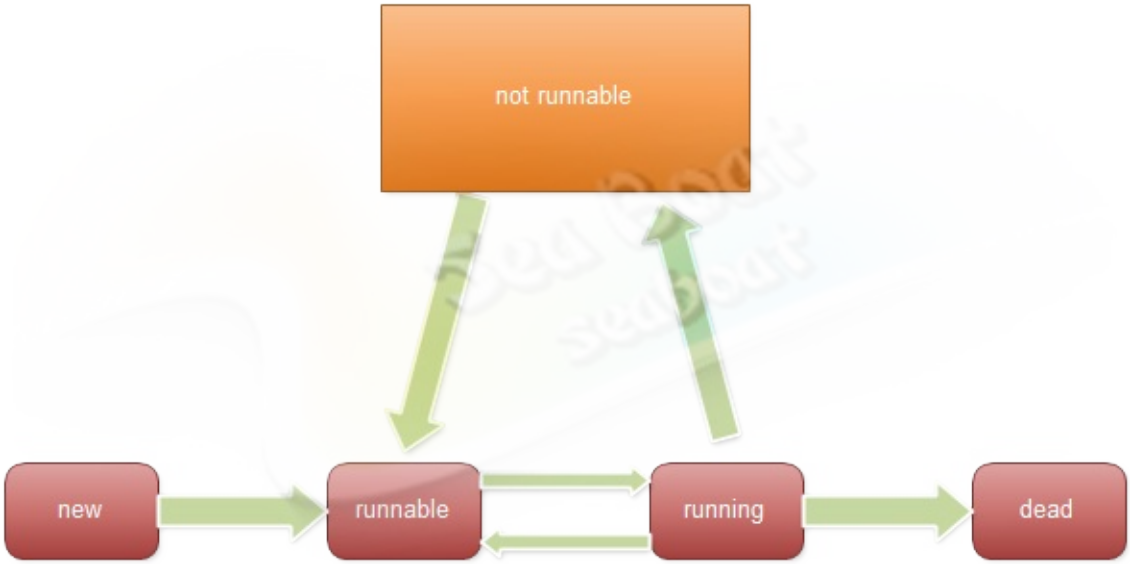


图2-5-7-1

l 新建（new）：一个线程被创建了但未被启动就处于新建状态，即在程序中使用new MyThread();创建的线程实例就处于此状态。

l 可运行（runnable）：创建的线程实例调用start()方法后便进入可运行状态，处于此状态的线程并不是说一定处于运行状态，我们在上一节多线程调度策略了解到Java多线程使用的是抢占式调度，每个可运行线程轮着获取CPU时间片，可以虚拟想象成有一个可运行线程池，start()方法把线程放进可运行线程池中，CPU按一定规则一个个执行池里的线程。

l 运行（running）：当可运行线程获取到CPU执行时间片即进去了运行状态。

l 非可运行（notrunnable）：运行中的线程因某种原因暂时放弃CPU的使用权，可能是因为执行了挂

起、睡眠或等待等操作，在执行I/O操作时由于外部设备速度远低于处理器速度也可能导致线程暂时放弃CPU使用权，在获取对象的同步锁过程中如果同步锁先被别的线程占用同样可能导致线程暂时放弃CPU。

l 死亡（dead）：线程执行完run()方法实现的任务，或因为异常导致退出任务，线程进入死亡状态后将不可再转换成其他状态。

将非可运行（not runnable）状态继续细分，如图2-5-7-2，新建、可运行、运行、死亡四个状态的定义和转化与前面的一样，重点看非可运行状态引申出来的三个状态：阻塞（blocked）、同步锁（locked）、等待（waiting）。

l 阻塞（blocked）：阻塞由阻塞事件触发，线程处于阻塞状态将放弃CPU的使用权，暂时停止运行。一般线程执行了sleep()、join()方法，或发出了I/O请求，线程就将处于阻塞状态，假如sleep()执行的睡眠结束、join()执行的等待中断超时、I/O请求结束，则将重新回到可执行状态，等待分配CPU。

l 同步锁（locked）：假如一个线程准备调用一个同步方法，而同步方法对应的对象正被其他线程占用，此时线程就将进入同步锁状态。实际上，Java中的每个object对象都有一个monitor，此monitor负责对同步域在并发时的独占处理，即一个线程调用某对象的同步方法时，JVM将检测改对象的monitor是否已被占用，如果没有被占用，线程则得到monitor占有权，继续执行该对象的同步方法，否则线程将被扔进一个等待线程队列排队，直到monitor被释放后，所有等待的线程继续竞争monitor占有权，抢到monitor占有权后才进入可执行状态等待CPU的分配，才有资格执行同步方法。

l 等待（waiting）：运行中的线程执行了wait()方法后就进入等待状态。一个对象执行了wait()方法同样将使线程进入该对象的等待线程队列，同时它还将释放对象锁，即放弃monitor的占有权。只有在其他线程中对该对象调用notify()、notifyAll()方法时才会唤醒等待线程队列中的线程，notify是随机唤醒等待队列中的一个线程，而nofityAll则是唤醒所有等待队列中的线程，所有线程被唤醒后将对该对象的monitor占有权竞争，获取到占有权的线程才能转化为可执行状态，等待分配CPU往下执行，其他线程则继续等待。

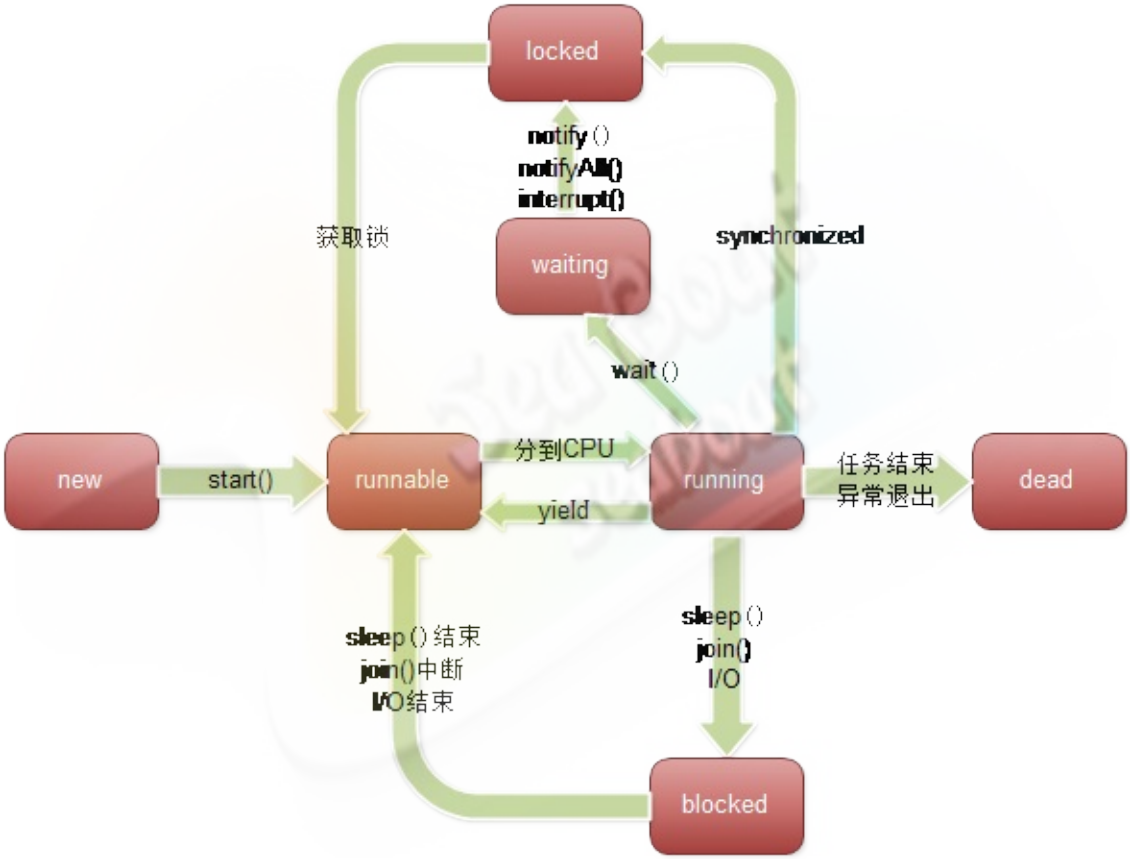


图2-5-7-2

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

多线程之Java线程阻塞与唤醒

线程的阻塞和唤醒在多线程并发过程中是一个关键点，当线程数量达到很大的数量级时，并发可能带来很多隐蔽的问题。如何正确暂停一个线程，暂停后又如何在一个要求的时间点恢复，这些都需要仔细考虑的细节。在Java发展史上曾经使用suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。如下代码，主要的逻辑代码是主线程启动线程mt一段时间后尝试使用suspend()让线程挂起，最后使用resume()恢复线程。但现实并不如愿，执行到suspend()时将一直卡住，你等不来“canyou get here?”的输出。

```
public class ThreadSuspend {

public static voidmain(String[] args) {

    Thread mt = newMyThread();

    mt.start();

    try {

        Thread.currentThread().sleep(100);

    } catch(InterruptedException e) {

        e.printStackTrace();

    }

    mt.suspend();

    System.out.println("canyou get here?");

    mt.resume();

}

static class MyThreadextends Thread {

    public void run() {

        while (true) {

            System.out.println("running....");

        }
```

```
    }
}
}
```

产生上面所述现象其实是由死锁导致，看起来一点问题都没有，线程的任务仅仅只是简单地打印字符串，问题的根源隐藏得较深，主线程启动了线程mt后，线程mt开始执行execute()方法，不断打印字符串，问题正是出现在System.out.println，由于println被声明为一个同步方法，执行时将对System类的out（PrintStream类的一个实例）单例属性加同步锁，而suspend()方法挂起线程但并不释放锁，在线程mt被挂起后主线程调用System.out.println同样需要获取System类out对象的同步锁才能打印“can you get here?”，主线程一直在等待同步锁而mt线程不释放锁，这就导致了死锁的产生。

可见suspend和resume有死锁倾向，一不小心将导致很多问题，甚至导致整个系统崩溃。也许，解决方案可以使用以对象为目标的阻塞，即利用Object类的wait()和notify()方法实现线程阻塞。针对对象的阻塞编程思维需要稍微转化下，它与面向线程阻塞思维有较大差异，如前面的suspend与resume只需在线程内直接调用就能完成挂起恢复操作，这个很好理解，而如果改用wait、notify形式则通过一个object作为信号，可以看成是一堵门，object的wait()方法是锁门的动作，notify()是开门的动作，某一线程一旦关上门后其他线程都将阻塞，直到别的线程打开门。如图2-5-8-4，一个对象object调用wait()方法则像是堵了一扇门，线程一、线程二都将阻塞，线程三调用object的notify()方法打开门（准确说是调用了notifyAll()方法，notify()仅仅能让线程一或线程二其中一条线程通过），线程一、线程二得以通过。

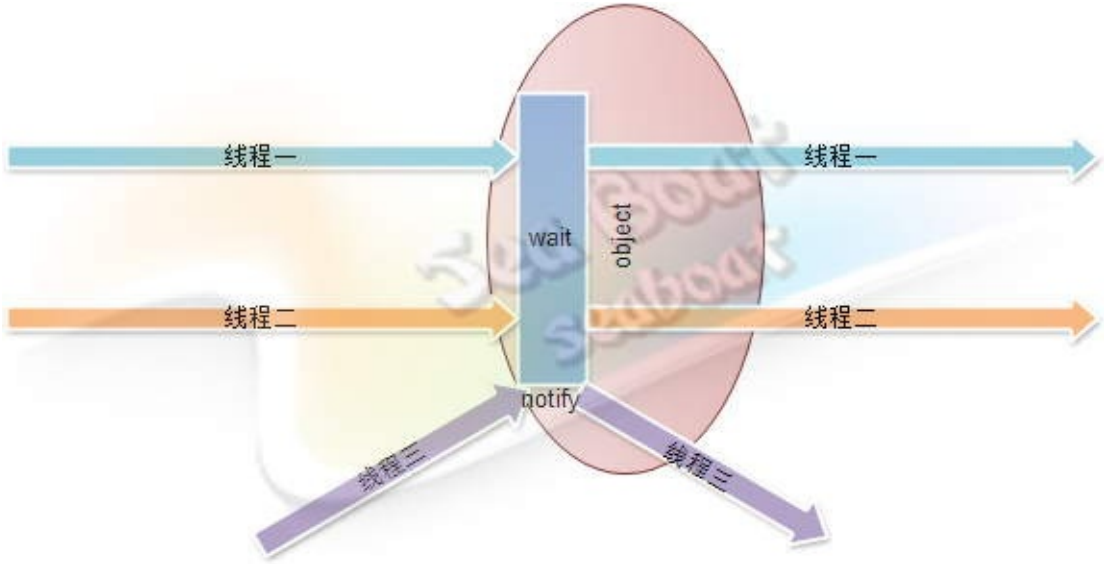


图2-5-8-4

使用wait和notify能规避死锁问题，但并不能完全避免，必须在编程过程中避免死锁。在使用过程中需要注意的几点是：首先，wait、notify方法是针对对象的，调用任意对象的wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行；其次，wait、notify方法必须在

synchronized块或方法中被调用，并且要保证同步块或方法的锁对象与调用wait、notify方法的对象是同一个，如此一来在调用wait之前当前线程就已经成功获取某对象的锁，执行wait阻塞后当前线程就将之前获取的对象锁释放。当然假如你不按照上面规定约束编写，程序一样能通过编译，但运行时将抛出IllegalMonitorStateException异常，必须在编写时保证用法正确；最后，notify是随机唤醒一条阻塞中的线程并让之获取对象锁，进而往下执行，而notifyAll则是唤醒阻塞中的所有线程，让他们去竞争该对象锁，获取到锁的那条线程才能往下执行。

通过wait、notify改造上面的例子，代码如下，改造的思想就是在MyThread中添加一个标识变量，一旦变量改变就相应地调用wait和notify阻塞唤醒线程，由于在执行wait后将释放synchronized (this)锁住的对象锁，此时System.out.println("running....");早已执行完毕，System类out对象不存在死锁问题。

```
publicclass ThreadWait {

    public static void main(String[] args) {

        MyThread mt = new MyThread();

        mt.start();

        try {

            Thread.currentThread().sleep(10);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        mt.suspendThread();

        System.out.println("can you gethere?");

        try {

            Thread.currentThread().sleep(3000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        mt.resumeThread();

    }
```

```
}
classMyThread extends Thread {
    public boolean stop = false;
    public void run() {
        while (true) {
            synchronized (this) {
                System.out.println("running....");
                if (stop)
                    try {
                        wait();
                    } catch(InterruptedException e) {
                        e.printStackTrace();
                    }
            }
        }
    }
    public void suspendThread() {
        this.stop = true;
    }
    public void resumeThread() {
        synchronized (this) {
            this.stop = false;
            notify();
         }
```

```
    }
}
```

wait与notify组合的方式看起来是个不错的解决方式，但其面向的主体是对象object，阻塞的是当前线程，而唤醒的是随机的某个线程或所有线程，偏重于线程之间的通信交互。假如换个角度，面向的主体是线程的话，我就能轻而易举地对指定的线程进行阻塞唤醒，这个时候就需要LockSupport，它提供的park和unpark方法分别用于阻塞和唤醒，而且它提供避免死锁和竞态条件，很好地代替suspend和resume组合。用park和unpark改造上述例子，代码如下：

```
public class ThreadPark {
public static voidmain(String[] args) {
        MyThreadmt = new MyThread();
        mt.start();
        try {
            Thread.currentThread().sleep(10);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
        mt.park();
        System.out.println("canyou get here?");
        try {
            Thread.currentThread().sleep(3000);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
        mt.unPark();
}
static classMyThread extends Thread {
```

```
        privateboolean isPark=false;
        publicvoid run() {
                while(true) {
                        if(isPark)
                                LockSupport.park();
                        System.out.println("running....");
                }
        }
   public void park(){
   isPark=true;
   }
   public void unPark(){
   isPark=false;
LockSupport.unpark(this);
   }
}
}
```

把主体换成线程进行的阻塞看起来貌似比较顺眼，而且由于park与unpark方法控制的颗粒度更加细小，能准确决定线程在某个点停止，进而避免死锁的产生，例如此例中在执行System.out.println前线程就被阻塞了，于是不存在因竞争System类out对象而产生死锁，即便在执行System.out.println后线程才阻塞也不存在死锁问题，因为锁已释放。

LockSupport类为线程阻塞唤醒提供了基础，同时，在竞争条件问题上，它具有wait和notify无可比拟的优势。使用wait和notify组合时，某一线程在被另一线程notify之前必须要保证此线程已经执行到wait等待点，错过notify则可能永远都在等待，另外notify也不能保证唤醒指定的某线程。反观LockSupport，由于park与unpark引入了许可机制，许可逻辑为：①park将许可在等于0的时候阻塞，等于1的时候返回并将许可减为0；②unpark尝试唤醒线程，许可加1。根据这两个逻辑，对于同一条线程，park与unpark先后操作的顺序似乎并不影响程序正确地执行，假如先执行unpark操作，许可则为1，之后再执行park操作，此时因为许可等于1直接返回往下执行，并不执行阻塞操作。

最后，LockSupport的park与unpark组合真正解耦了线程之间的同步，不再需要另外的对象变量存储状态，并且也不需要考虑同步锁，wait与notify要保证必须有锁才能执行，而且执行notify操作释放锁后还要将当前线程扔进该对象锁的等待队列，LockSupport则完全不用考虑对象、锁、等待队列等问题。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS之阻塞与唤醒

根据前面的线程阻塞与唤醒小节知道，目前在Java语言层面能实现阻塞唤醒的方式一共有三种：suspend与resume组合、wait与notify组合、park与unpark组合。其中suspend与resume因为存在无法解决的竟态问题而被Java废弃，同样，wait与notify也存在竟态条件，wait必须在notify之前执行，假如一个线程先执行notify再执行wait将可能导致一个线程永远阻塞，如此一来，必须要提出另外一种解决方案，就是park与unpark组合，它位于juc包下，应该也是因为当时编写juc时发现java现有方式无法解决问题而引入的新阻塞唤醒方式，由于park与unpark使用的是许可机制，许可最大为1，所以unpark与park操作不会累加，而且unpark可以在park之前执行，如unpark先执行，后面park将不阻塞。
Java真正意义上的语言层面上的并发编程应该从并发专家Doug Lea领导的JSR-166开始，此规范请求向JCP提交了向Java语言中添加并发编程工具，即在jdk中添加java.util.concurrent工具包供开发者使用，开发者可以轻松构建自己的同步器，而在此之前并发过程中同步都只能依靠JVM内置的管程。

ASQ框架的阻塞和唤醒显然使用的是LockSupport类的park与unpark方法，分别调用的是Unsafe类的park与unpark本地方法。逻辑如下：

阻塞

```
if(尝试获取锁失败) {
  创建node
  使用CAS方式把node插入到队列尾部
  while(true){
  if(尝试获取锁成功 并且 node的前驱节点为头节点){
把当前节点设置为头节点
  跳出循环
}else{
  使用CAS方式修改node前驱节点的waitStatus标识为signal
  if(修改成功){
      LockSupport.park();
}
}
```

}

唤醒

if(尝试释放锁成功){

LockSupport.unpark(下一节点包含的线程);

}

假如一条线程参与锁竞争，首先先尝试获取锁，失败的话创建节点并插入队列尾部，然后再次尝试获取锁，如若成功则不做其他任务处理直接返回，否则设置节点状态为待运行状态，最后使用LockSupport的park阻塞当前线程。前驱节点运行完后将尝试唤醒后继节点，使用的即是LockSupport的unpark唤醒。

总的来说，java提供的juc并发工具包，在阻塞与唤醒操作方面由于suspend与resume存在各种各样问题，必须使用LockSupport中提供的方法操作。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS阻塞队列管理（一）——自旋锁

我们知道一个线程在尝试获取锁失败后将被阻塞并加入等待队列中，它是一个怎样的队列？又是如何管理此队列？这节聊聊CHL Node FIFO队列。
在谈到CHL Node FIFO队列之前，我们先分析这种队列的几个要素。首先要了解的是自旋锁，所谓自旋锁即是某一线程去尝试获取某个锁时，如果该锁已经被其他线程占用的话，此线程将不断循环检查该锁是否被释放，而不是让此线程挂起或睡眠。它属于为了保证共享资源而提出的一种锁机制，与互斥锁类似，保证了公共资源在任意时刻最多只能由一条线程获取使用，不同的是互斥锁在获取锁失败后将进入睡眠或阻塞状态。下面利用代码实现一个简单的自旋锁，

```
public class SpinLock {
private static Unsafe unsafe = null;
private static final long valueOffset;
private volatile int value = 0;
static {
try {
unsafe=getUnsafeInstance();
valueOffset = unsafe.objectFieldOffset(SpinLock.class
.getDeclaredField("value"));
} catch (Exception ex) {
throw new Error(ex);
}
}
private static Unsafe getUnsafeInstance() throws SecurityException,
NoSuchFieldException, IllegalArgumentException,
IllegalAccessException {
Field theUnsafeInstance = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafeInstance.setAccessible(true);
return (Unsafe) theUnsafeInstance.get(Unsafe.class);
}
public void lock() {
for (;;) {
        int newV = value + 1;
        if (unsafe.compareAndSwapInt(this, valueOffset, 0, newV)){
        return ;
        }
    }
}
```

```
public void unlock() {
unsafe.compareAndSwapInt(this, valueOffset, 1, 0);
}
}
```

这是一个很简单的自旋锁，主要看加粗加红的两个方法lock和unlock，Unsafe仅仅是为操作提供了硬件级别的原子CAS操作，暂时忽略此类，只要知道它的作用即可，我们将在后面的“原子性如何保证”小节中对此进行更加深入的阐述。对于lock方法，假如有若干线程竞争，能成功通过CAS操作修改value值为newV的线程即是成功获取锁的线程，将直接通过，而其他的线程则不断在循环检测value值是否又改回0，而将value改为0的操作就是获取锁的线程执行完后对该锁进行释放，通过unlock方法释放锁，释放后若干线程又对该锁竞争。如此一来，没获取的锁也不会被挂起或阻塞，而是不断循环检查状态。图2-5-9-3可加深自旋锁的理解，五条线程轮询value变量，t1获取成功后将value置为1，此状态时其他线程无法竞争锁，t1使用完锁后将value置为0，剩下的线程继续竞争锁，以此类推。这样就保证了某个区域块的线程安全性。

图2-5-9-3 自旋锁

自旋锁适用于锁占用时间短，即锁保护临界区很小的情景，同时它需要硬件级别操作，也要保证各缓存数据的一致性，另外，无法保证公平性，不保证先到先获得，可能造成线程饥饿。在多处理器机器上，每个

线程对应的处理器都对同一个变量进行读写，而每次读写操作都将要同步每个处理器缓存，导致系统性能严重下降。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS阻塞队列管理（二）——自旋锁优化

看Craig, Landin, and Hagersten发明的CLH锁如何优化同步带来的花销，其核心思想是：通过一定手段将所有线程对某一共享变量轮询竞争转化为一个线程队列且队列中的线程各自轮询自己的本地变量。这个转化过程由两个要点，一是构建怎样的队列&如何构建队列，为了保证公平性，构建的将是一个FIFO队列，构建的时候主要通过移动尾部节点tail实现队列的排队，每个想获取锁的线程创建一个新节点并通过CAS原子操作将新节点赋予tail，然后让当前线程轮询前一节点的某个状态位，如图2-5-9-3，如此就成功构建线程排队队列；二是如何释放队列，执行完线程后只需将当前线程对应的节点状态位置为解锁状态，由于下一节点一直在轮询，可获取到锁。

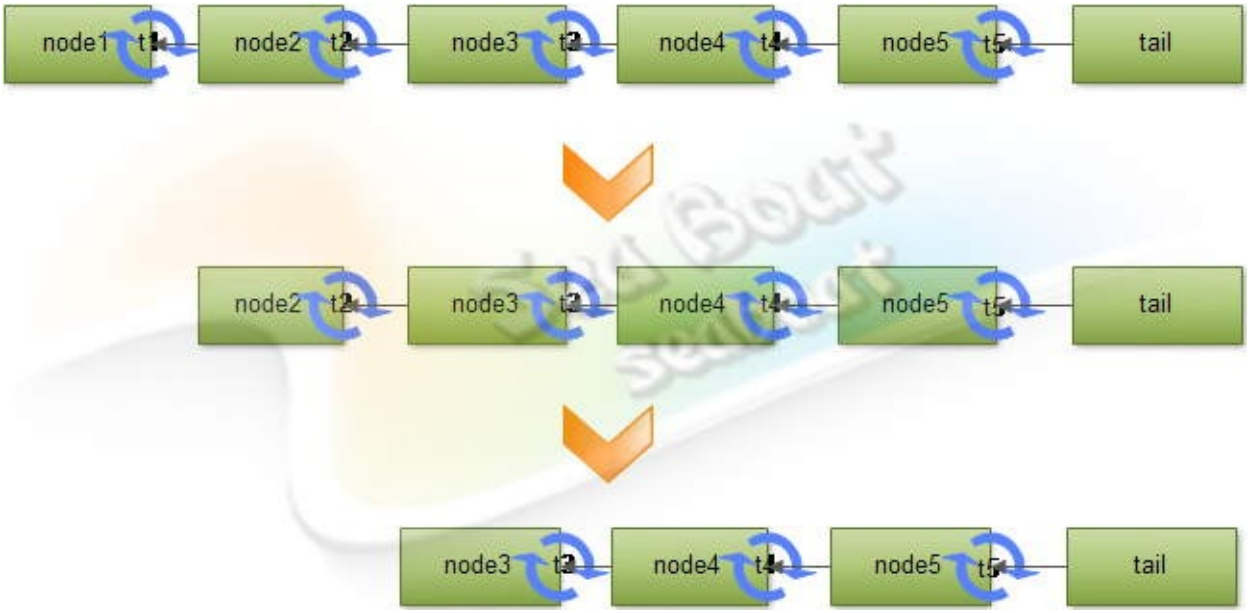


图2-5-9-3 CLH锁

CLH锁的核心思想貌似是将众多线程长时间对某资源的竞争，通过有序化这些线程转化为只需对本地变量检测。唯一存在竞争的地方就是在入队列之前对尾节点tail的竞争，但竞争的线程的数量已经少了很多，且比起所有线程直接对某资源竞争的轮询次数也减少了很多，节省了很多CPU缓存同步操作，大大提升系统性能，利用空间换取性能。下面提供一个简单的CLH锁实现代码，lock与unlock两方法提供加锁解锁操作，每次加锁解锁必须将一个CLHNode对象作为参数传入，lock方法的for循环是通过CAS操作将新节点插入队列，而while循环则是检测前驱节点的锁状态位，一旦前驱节点锁状态位允许则结束检测让线程往下执行。解锁操作先判断当前节点是否为尾节点，如是则直接将尾节点置为空，此时说名仅仅只有一条线程在执行，否则将当前节点的锁状态位置为解锁状态。

```
public class CLHLock {

private static Unsafe unsafe = null;
```

```
private static final long valueOffset;
private volatile CLHNode tail;
public class CLHNode {
private boolean isLocked = true;
}

static {
try {
unsafe = getUnsafeInstance();
valueOffset = unsafe.objectFieldOffset(CLHLock.class
.getDeclaredField("tail"));
} catch (Exception ex) {
throw new Error(ex);
}
}

public void lock(CLHNode currentThreadNode) {
CLHNode preNode = null;
for (;;) {
preNode = tail;
if (unsafe.compareAndSwapObject(this, valueOffset, tail,
currentThreadNode))
break;
}
if (preNode != null)
while (preNode.isLocked) {
}
}

public void unlock(CLHNode currentThreadNode) {
if (!unsafe.compareAndSwapObject(this, valueOffset, currentThreadNode,null))
 currentThreadNode.isLocked = false;
}

private static Unsafe getUnsafeInstance() throws SecurityException,
NoSuchFieldException, IllegalArgumentException,
IllegalAccessException {
Field theUnsafeInstance = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafeInstance.setAccessible(true);
return (Unsafe) theUnsafeInstance.get(Unsafe.class);
```

}

}

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS阻塞队列管理（三）——CLH锁改造

在CLH锁核心思想的影响下，Java并发包的基础框架AQS以CLH锁作为基础而设计，其中主要是考虑到CLH锁更容易实现取消与超时功能。比起原来的CLH锁已经做了很大的改造，主要从两方面进行了改造：节点的结构与节点等待机制。在结构上引入了头结点和尾节点，他们分别指向队列的头和尾，尝试获取锁、入队列、释放锁等实现都与头尾节点相关，并且每个节点都引入前驱节点和后后续节点的引用；在等待机制上由原来的自旋改成阻塞唤醒。如图2-5-9-4，通过前驱后续节点的引用一节节连接起来形成一个链表队列，对于头尾节点的更新必须是原子的。下面详细看看入队、检测挂起、释放出队、超时、取消等操作。

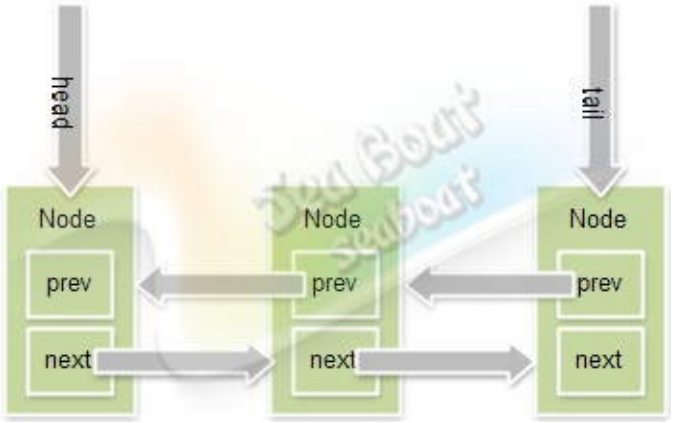


图2-5-9-4 CLH队列

①入队，整块逻辑其实是用一个无限循环进行CAS操作，即用自旋方式竞争直到成功。将尾节点tail的旧值赋予新节点node的前驱节点，并尝试CAS操作将新节点node赋予尾节点tail，原先的尾节点的后续节点指向新建节点node。完成上面步骤就建立起一条如图2-5-9-4所示的链表队列。代码简化如下：

```
for (;;) {
 Node t = tail;
 node.prev = t;
 if (compareAndSetTail(t, node)) {
    t.next = node;
    return node;
 }
}
```

②检测挂起，上面我们说到节点等待机制已经被AQS作者由自旋机制改造成阻塞机制，一个新建的节点完成入队操作后，如果是自旋则直接进入循环检测前驱节点是否为头结点即可，但现在被改为阻塞机制，当前线程将首先检测是否为头结点且尝试获取锁，如果当前节点为头结点并成功获取锁则直接返回，当前线程不进入阻塞，否则将当前线程阻塞。代码简化如下：

```
for (;;) {
```

```
  if (node.prev == head)
if(尝试获取锁成功){
      head=node;
      node.next=null;
      return;
   }
 阻塞线程
}
```

③释放出队，出队的主要工作是负责唤醒等待队列中后续节点，让所有等待节点环环相接，每条线程有序地往下执行。代码简化如下：

```
Node s = node.next;
唤醒节点s包含的线程
```

④超时，在支持超时的模式下需要LockSupport类的parkNanos方法支持，线程在阻塞一段时间后会自动唤醒，每次循环将累加消耗时间，当总消耗时间大于等于自定义的超时时间时就直接分返。代码简化如下：

```
for (;;) {
 尝试获取锁
 if (nanosTimeout <= 总消耗时间)
    return;
 LockSupport.parkNanos(this, nanosTimeout);
}
```

⑤取消，队列中等待锁的队列可能因为中断或超时而涉及到取消操作，这种情况下被取消的节点不再进行锁竞争。此过程主要完成的工作是将取消的节点移除，先将节点的。先将节点node状态设置成取消，再将前驱节点pred的后续节点指向node的后续节点，这里由于涉及到竞争，必须通过CAS进行操作，CAS操作就算失败也不必理会，因为已经改了节点的状态，在尝试获取锁操作中会循环对节点的状态判断。

```
node.waitStatus = Node.CANCELLED;
Node pred = node.prev;
Node predNext = pred.next;
Node next = node.next;

compareAndSetNext(pred, predNext, next);
```

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——AQS超时机制

AQS框架提供的另外一个优秀机制是锁获取超时的支持，当大量线程对某一锁竞争时可能导致某些线程在很长一段时间都获取不了锁，在某些场景下可能希望如果线程在一段时间内不能成功获取锁就取消对该锁的等待以提高性能，这时就需要用到超时机制。在JDK1.5之前还没有juc工具，当时的并发控制职能通过JVM内置的synchronized关键词实现锁，但对一些特殊要求却力不从心，例如超时取消控制。JDK1.5开始引入juc工具完美解决了此问题，而这正得益于并发基础框架AQS提供了超时的支持。
为了更精确地保证时间间隔统计的准确性，实现时使用了System.nanoTime()更为精确的方法，它能精确到纳秒级别。超时机制的思想就是在不断进行锁竞争的同时记录竞争的时间，一旦时间段超过指定的时间则停止轮询直接返回，返回前对等待队列中对应节点进行取消操作。往下看实现的逻辑，

```
if(尝试获取锁失败) {
long lastTime = System.nanoTime();
  创建node
  使用CAS方式把node插入到队列尾部
  while(true){
  if(尝试获取锁成功 并且 node的前驱节点为头节点){
把当前节点设置为头节点
  跳出循环
}else{
  if (nanosTimeout <= 0){
取消等待队列中此节点
跳出循环
}
  使用CAS方式修改node前驱节点的waitStatus标识为signal
  if(修改成功)
      if(nanosTimeout > spinForTimeoutThreshold)
      阻塞当前线程nanosTimeout纳秒
  long now = System.nanoTime();
  nanosTimeout -= now - lastTime;
  lastTime = now;
}
}
```

上面正是在前面章节锁的获取逻辑中添加超时处理，核心逻辑是不断循环减去处理的时间消耗，一旦小于0就取消节点并跳出循环，其中有两点必须要注意，一个是真正的阻塞时间应该是扣除了竞争入队的时间后剩余的时间，保证阻塞事件的准确性，我们可以看到每次循环都会减去相应的处理时间；另外一个是关于spinForTimeoutThreshold变量阀值，它是决定使用自旋方式消耗时间还是使用系统阻塞方式消耗时间的分割线，juc工具包作者通过测试将默认值设置为1000ns，即如果在成功插入等待队列后剩余时间大于

1000ns则调用系统底层阻塞，否则不调用系统底层，取而代之的是仅仅让之在Java应用层不断循环消耗时间，属于优化的措施。

至此AQS框架在获取锁的过程中提供了超时机制，超时的支持让Java在并发方面提供了更完善的机制，更多的并发策略满足开发者更多需求。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——同步状态的管理

整个AQS框架核心功能都是围绕着其32位整型属性state进行，一般可以说它表示锁的数量，对同步状态的控制可以实现不同的同步工具，例如闭锁、信号量、栅栏等等。为了保证可见性此变量被声明为volatile，保证每次的原子更新都将及时反映到每条线程上。而对于同步状态的管理可以大体分为两块，一是独占模式的管理，另外是共享模式的管理。通过对这两种模式的灵活变换可以实现多种不同的同步器，如下图，对state的控制可以看成一个管道，管道的大小决定了同时通过的线程，独占模式好比宽度只容许一个线程通过的管道，在这种模式下线程只能逐一通过管道，任意时刻管内只能存在一条线程，这便形成了互斥效果。而共享模式就是管道宽度大于1的管道，可以同时让n条管道通过，吞吐量增加但可能存在共享数据一致性问题。（注意：两种模式的讨论忽略了队列的管理逻辑，实际上CLH Node的引入是为了优化竞争带来的性能问题，不影响同步状态管理的探讨）

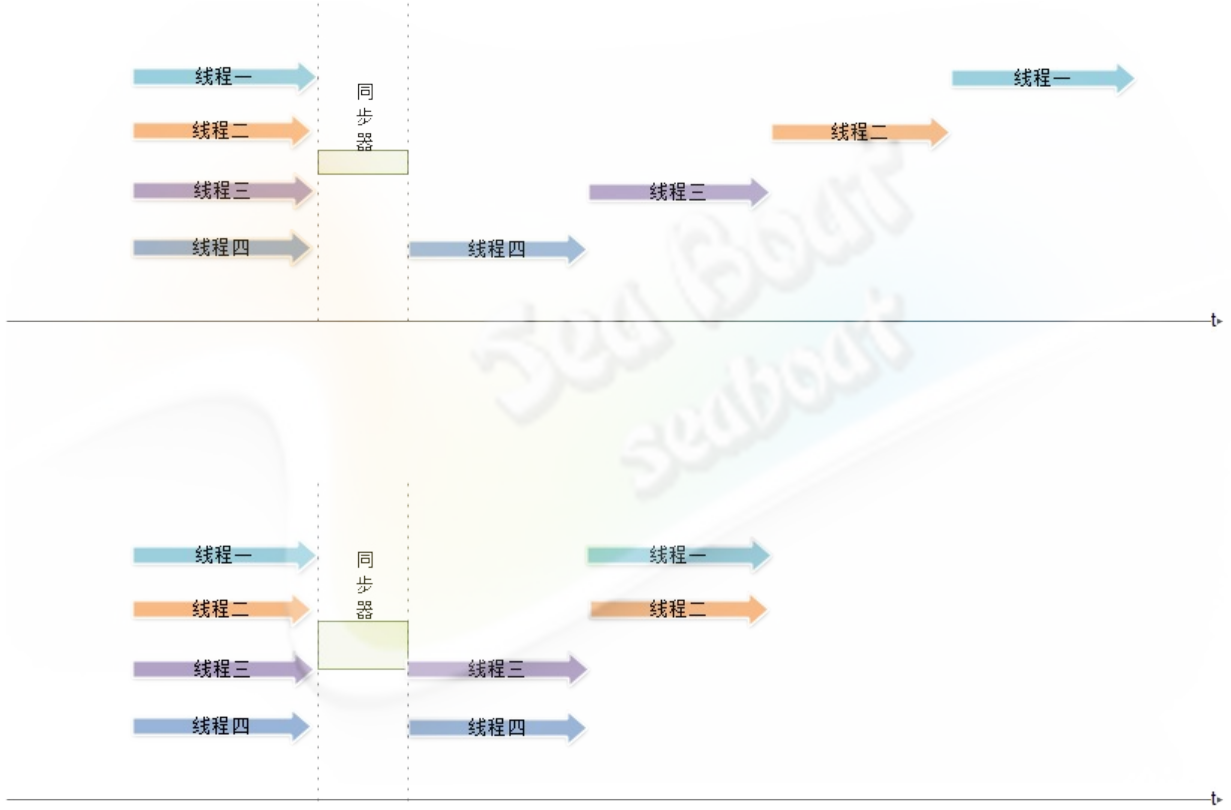


图2-5-9-5 独占模式与共享模式

如何通过state实现独占模式和共享模式？在此之前先了解AQS框架中相关的getState、setState、compareAndSetState三个操作state的基本方法，前两个方法是普通的获取设置方法，其必须保证不存在数据竞争的情况下使用，compareAndSetState方法则提供了CAS方式的硬件级别的原子更新。两种模式就是通过这些方法对state操作实现不同同步模式，下面给出最简单的实现。

独占模式

```
public boolean tryAcquire(int acquires) {
```

```
if (compareAndSetState(0, 1)) {
return true;
}
return false;
}
protected boolean tryRelease(int releases) {
setState(0);
return true;
}
```

多条线程通过tryAcquire尝试把state变量改为1，由于CAS算法的保证，最终有且仅有一条线程成功修改state，修改成功的线程代表获取锁成功，将拥有往下执行的权利，进入管道。当执行完毕退出管道时执行tryRelease尝试把state变量改为0，让出管道，此处由于不存在线程竞争所以可直接使用setState，接着其他未通过的线程继续重复尝试。

共享模式

```
public int tryAcquireShared(int interval) {
for (;;) {
int current = getState();
int newCount = current - 1;
if (newCount < 0 || compareAndSetState(current, newCount)) {
return newCount;
}
}
}
public boolean tryReleaseShared(int interval) {
for (;;) {
int current = getState();
int newCount = current + 1;
if (compareAndSetState(current, newCount)) {
return true;
}
}
}
```

与独占模式不同的是对state的管理及判断条件，独占模式state的值只能为0或1，而共享模式的state是可以被出事换成任意整数，一般初始值表示提供一个同时n条线程通过的管道宽度，这样一来，多条线程通过tryAcquireShared尝试将state的值减去1，成功修改state后就返回新值，只有当新值大于等于0才表示获取锁成功，拥有往下执行的权利，进入管道。在执行完毕时线程将调用tryReleaseShared尝试修改state值使之增加1，表示我已经执行完了并让出管道的通道供后面线程使用，需要说明的是与独占模式不同，由于可能存在多条线程并发释放锁，所以此处必须使用基于CAS算法的修改方法，修改成功后其他线程便可继续竞争锁。

ASQ框架提供了对同步状态state的基本操作，了解了两种模式对state操作开发者可能很自由地自定义自己的同步器。实际中AQS框架在提供state状态管理接口的同时也将维护等待队列的工作，两项工作被封装成一个模板，规定了工作流程，工作流程包括什么条件下加入等待队列、什么条件移除等待节点、如何操作等待队列、需不需要阻塞、支不支持中断等等，对外仅仅提供state状态操作接口供开发者自定义，而队列的维护工作已经绑定在模板中，无需你自己动手。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发框架——公平性

所谓公平性指所有线程对临界资源申请访问权限的成功率都一样，不会让某些线程拥有优先权。通过前面的CLH Node FIFO学习知道了等待队列是一个先进先出的队列，那么是否就可以说每条线程获取锁时就是公平的呢？关于公平性这里分拆成三个点分别阐述：

①准备入队列的节点，此情况讨论的是线程加入等待队列时产生的竞争是否公平，线程在尝试获取锁失败后将被加入等待队列，这时多个线程通过自旋将节点加入队列，所有线程在自旋过程中是无法保证其公平性的，可能后来的线程比早到的先进入队列，所以节点入队列不具公平性。

②等待队列中的节点，情况①中成功加入队列后即成为等待队列中的节点，我们知道此队列是一个先入先出队列，那么很简单能得到，队列中的所有节点是公平的，他们都按照顺序等待自己被前驱节点唤醒并获取锁，所以等待队列中的节点具有公平性。

③闯入的节点，这种情况是指一个新线程到达共享资源边界时不管等待队列中是否存在其他等待节点它都将优先尝试去获取锁，这种称为可闯入策略。可闯入特性破坏了公平性，AQS框架对外体现的公平性主要由此体现，下面将对闯入特性展开分析。

AQS提供的基础获取锁算法是一种可闯入的算法，即如果有新线程到来先进行一次获取尝试，不成功的情况下才将当前线程加入等待队列。如图2-5-9-6所示，等待队列中节点线程按照顺序一个接一个尝试去获取共享资源的使用权，某时刻头结点线程准备尝试获取的同时另外一条线程闯入，此线程并非直接加入等待队列的尾部，而是先跟头结点线程竞争获取资源，闯入线程如果成功获取共享资源则直接执行，头结点线程则继续等待下一次尝试，如此一来闯入线程成功插队，后来的线程比早到的线程先执行，说明AQS基础获取算法是不严格公平的。

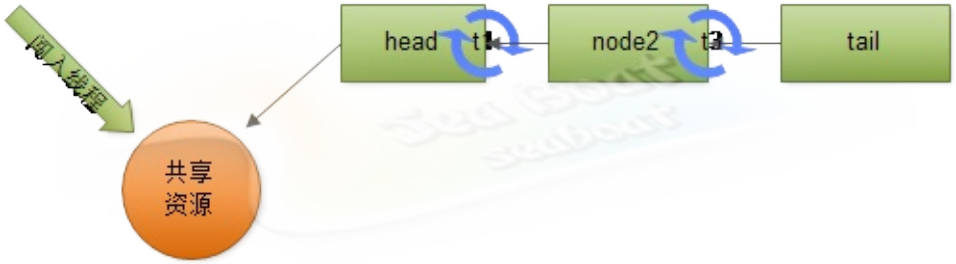


图2-5-9-6 闯入线程

基础获取算法逻辑简化如下：首先尝试获取锁，假如获取失败才创建节点并加入到等待队列的尾部，接着通过不断循环检查是否轮到自己执行，当然此过程为了提高性能可能将线程先挂起，最终由前驱节点唤醒。

```
if(尝试获取锁失败) {
  创建node
  使用CAS方式把node插入到队列尾部
  while(true){
  if(尝试获取锁成功 并且 node的前驱节点为头节点){
把当前节点设置为头节点
  跳出循环
```

```
}else{
  使用CAS方式修改node前驱节点的waitStatus标识为signal
  if(修改成功)
     挂起当前线程
}
}
```

为什么要使用闯入策略？可闯入的策略通常可以提供更高的总吞吐量。由于一般同步器颗粒度比较小，也可以说共享资源的范围较小，而线程从阻塞状态到被唤醒所消耗的时间周期可能是通过共享资源时间周期的几倍甚至几十倍，如此一来线程唤醒过程中将存在一个很大的时间周期空窗期，导致资源没有得到充分利用，为了提高吞吐量，引入这种闯入策略，它可以使在等待队列头结点从阻塞到被唤醒的时间段内闯入的线程直接获取锁并通过同步器，以便充分利用唤醒过程这一空窗期，大大增加了吞吐率。另外，闯入机制的实现对外提供一种竞争调节机制，即开发者可以在自定义同步器中定义闯入尝试获取的次数，假设次数为n则不断重复获取直到n次都获取不成功才把线程加入等待队列中，随着次数n的增加可以增大成功闯入的几率。同时，这种闯入策略可能导致等待队列中的线程饥饿，因为锁可能一直被闯入的线程获取，但由于一般持有同步器的时间很短暂而避免饥饿的发生，反之如果保护的代码体很长并且持有同步器的时间较长，这将大大增加等待队列无限等待的风险。

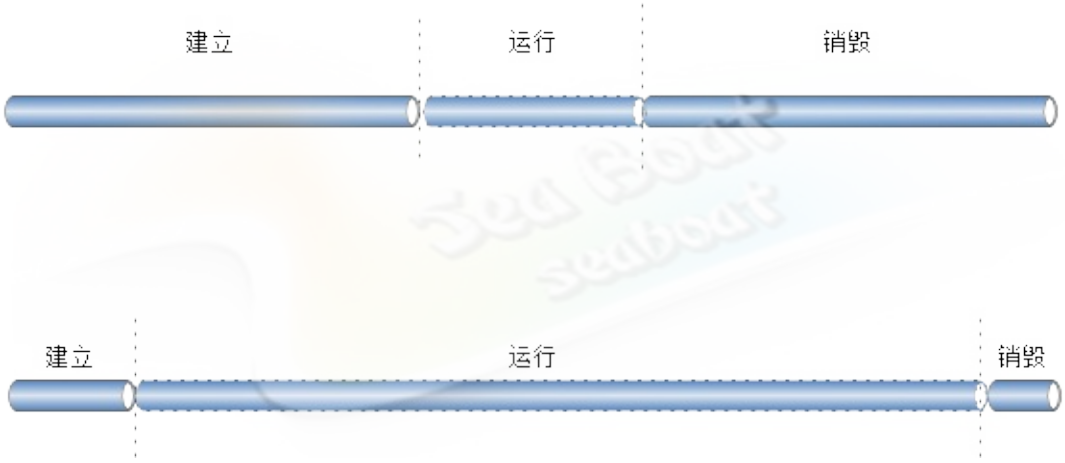
在实际情况中还是要根据用户需求制定策略，在一个公平性要求很高的场景，则可以把闯入策略去除掉以达到公平。在自定义同步器中可以通过AQS预留方法tryAcquire方法实现，只需判断当前线程是否为等待队列中头结点对应的线程，若不是则直接返回false，尝试获取失败。但前面这种公平性是相对Java语法语义层面上的公平性，在现实中JVM的实现会直接影响线程执行的顺序。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：

Java并发——线程池原理

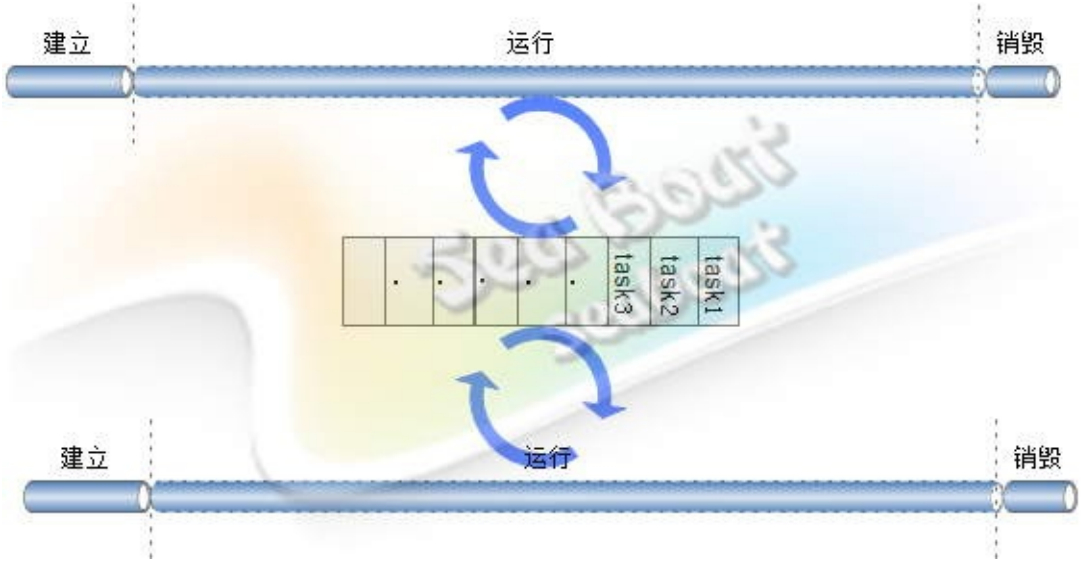
“池”技术对我们来说是非常熟悉的一个概念，它的引入是为了在某些场景下提高系统某些关键节点性能，最典型的例子就是数据库连接池，JDBC是一种服务供应接口（SPI），具体的数据库连接实现类由不同厂商实现，数据库连接的建立和销毁都是很耗时耗资源的操作，为了查询数据库中某条记录，最原始的一个过程是建立连接、发送查询语句、返回查询结果、销毁连接，假如仅仅是一个很简单的查询语句，那么可能建立连接与销毁连接两个步骤就已经占所有资源时间消耗的绝大部分，如此低下的效率显然让人无法接受。针对这个过程是否能通过某些手段提高效率，于是想到的尽可能减少创建和销毁连接操作，因为连接相对于查询是无状态的，不必每次查询都重新生成销毁，我们可以把这些通道维护起来供下一次查询使用，维护这些管道的工作就交给了“池”。

线程池也是类似于数据库连接池的一种池，而仅仅是把池里的对象换成了线程。线程是为多任务而引入的概念，每个线程在任意时刻执行一个任务，假如多个任务要并发执行则要用到多线程技术。每个线程都有自己的生命周期，以创建为始销毁为末。如下图，两个线程运行阶段占整个生命周期的比重不同，运行阶段所占比重小的线程可以认为其运行效率低，反观下面一条线程则认为运行效率高。在大多数场景下都比较符合图上面的线程运行模式，例如我们常见的web服务、数据库服务等等。为了提高运行效率引入线程池，它的核心思想就是把运行阶段尽量拉长，对于每个任务的到来不是重复建立销毁线程，而是重复利用之前建立好的线程执行任务。



其中一种方案是在系统启动事建立好一定数量的线程并做好线程维护工作，一旦有任务到来即从线程池中取出一条空闲的线程执行任务。原理听起来比较清晰，但现实中对于一条线程，一旦调用start方法后就将运行任务直到任务完成，随后JVM将对线程对象进行GC回收，如此一来线程不就销毁了吗？是的，所以需要换种思维角度，让这些线程启动后通过一个无限循环来执行指定的任务，下面将重点讲解如何实现线程池。

一个线程池的属性起码包含初始化线程数量、线程数组、任务队列。初始化线程数量指线程池初始化的线程数，线程数组保存了线程池中所有线程，任务队列指添加到线程池等待处理的所有任务。如下图，线程池里有两条线程，池里线程的工作就是不断循环检测任务队列中是否有需要执行的任务，如果有则处理并移出任务队列。于是可以说线程池中的所有线程的任务就是不断检测任务队列并不断执行队列中的任务。



看一个最简单粗糙的线程池的实现，使用线程池是只需实例化一个对象，构造函数会创建相应数量的线程并启动线程，启动的线程无限循环检测任务队列，执行方法execute()仅仅把任务添加到任务队列中。有一点需要注意的是所有任务都必须实现Runnable接口，这是线程池的任务队列与工作线程的约定，juc工具包作者Doug Lea大神当时如此规定，工作线程检测任务队列并调用队列的run()方法，假如你自己重新写一个线程池是完全可以自己定义一个不一样的任务接口。一个完善的线程池并不像下面例子简单，需要提供启动、销毁、增加工作线程的策略、最大工作线程数、各种状态的获取等等操作，而且工作线程也不可能老是做无用循环，需要对任务队列使用wait、notify优化或任务队列改用阻塞队列。

```
public final class ThreadPool {
private final int worker_num;
private WorkerThread[] workerThrads;
private List taskQueue = new LinkedList();
private static ThreadPool threadPool;

public ThreadPool(int worker_num) {
this.worker_num = worker_num;
workerThrads = new WorkerThread[worker_num];
for (int i = 0; i < worker_num; i++) {
workerThrads[i] = new WorkerThread();
workerThrads[i].start();
}
}

public void execute(Runnable task) {
synchronized (taskQueue) {
taskQueue.add(task);
}
```

```
}

private class WorkerThread extends Thread {
public void run() {
Runnable r = null;
while (true) {
synchronized (taskQueue) {
if (!taskQueue.isEmpty()) {
r = taskQueue.remove(0);
r.run();
}
}
}
}
}
}
```

通过上面已经清楚了线程池原理，但并不提倡重造轮子行为，因为线程池处理不好很容易产生死锁问题，同时线程池内状态同步操作不当也可能导致意想不到的问题，除此之外还有很多其他的并发问题，除非是很有经验的并发程序员才能尽可能减少可能的错误。我们直接使用jdk的juc工具包即可，它由Doug Lea编写的优秀并发程序工具，单线程池就已经提供了好多种类的线程池，实际开发中根据需求选择合适的线程池。

喜欢研究java的同学可以交个朋友，下面是本人的微信号：