# **1. 并发的挑战**

【并发的目的是为了让程序运行得更快，不要为了并发而并发】

并发带来的问题：

- 上下文切换的性能损耗

- 死锁

- 受限于硬件和软件的资源限制

## **1.1 上下文切换**

在单核处理器中，CPU通过“时间片分配算法”给每个线程分配CPU时间片。时间片一般是几十毫秒（ms）。

在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换。

### 1.1.1 多线程一定快吗

下面的代码：

```java

public class ConcurrencyTest {

    private static final long count = 10000000l;

    public static void main(String[] args) throws InterruptedException {

        concurrency();

        serial();

    }

    private static void concurrency() throws InterruptedException {

        long start = System.nanoTime();

        Thread thread = new Thread(new Runnable() {

            @Override

            public void run() {

                int a = 0;

                for (long i = 0; i < count; i++) {

                    a += 5;

                }

            }

        });

        thread.start();

        int b = 0;

        for (long i = 0; i < count; i++) {

            b--;

        }

        long time = System.nanoTime() - start;

        thread.join();

        System.out.println("concurrency : " + time/1000 + "ms, b = " + b);

    }

    private static void serial() {

        long start = System.nanoTime();

        int a = 0;

        for (long i = 0; i < count; i++) {

            a += 5;

        }

        int b = 0;

        for (long i = 0; i < count; i++) {

            b--;

        }

        long time = System.nanoTime() - start;

        System.out.println("serial : " + time/1000 + "ms, b = " + b);

    }

}

```

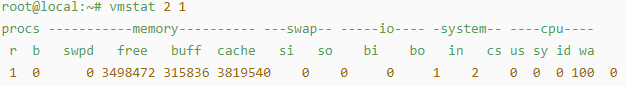
这段代码，当累加操作不超过百万次时，并发速度会比串行执行累加操作要慢。这是因为线程创建和上下文切换的额外开销。

### 1.1.2 测试上下文切换次数和时长

Lmbench3是一个性能分析工具。

使用 Lmbench3 可以测量上下文切换的时长。

使用 vmstat 可以测量上下文切换的次数。



CS（Content Switch）表示上下文切换的次数

### 1.1.3 如何减少上下文切换

1. 无锁并发编程

多线程竞争锁时，会引起上下文切换。可以对数据hash分段，不同的线程处理不同段的数据

2. CAS算法

Java的Atomic包使用CAS算法来更新数据，不需要加锁

3. 少使用线程

避免创建不需要的线程，不要造成大量线程都处于等待状态

4. 使用协程

在单线程里维持多个任务间的切换

### 1.1.4 减少上下文切换实战

减少线上大量WAITING的线程，来减少上下文切换次数

1. 用jstack命令dump线程信息，查看pid为3117的进程里的线程都在做什么

$ sudo -u admin /opt/ifeve/java/bin/jstack 31177 > /home/tengfei.fangtf/dump17

2. 统计所有线程分别处于什么状态，发现300多个线程处于WAITING（onobjectmonitor）状态

$ grep java.lang.Thread.State dump17 | awk '{print $2$3$4$5}'| sort |uniq -c

3. 打开dump文件查看处于WAITING（onobjectmonitor）的线程在做什么。发现这些线程基本全是JBOSS的工作线程，在await。说明JBOSS线程池里线程接收到的任务太少，大量线程都闲着

4. 减少JBOSS的工作线程数，找到JBOSS的线程池配置信息，将maxThreads降到100

5. 重启JBOSS，再dump线程信息，然后统计WAITING（onobjectmonitor）的线程，发现减少了175个。WAITING的线程少了，系统上下文切换的次数就会少，因为每一次从WAITTING到RUNNABLE都会进行一次上下文的切换

6. 使用vmstat命令测试一下

## 1.2 死锁

【一旦死锁，就会造成功能不可用】

简单演示代码：

```java

public class DeadLockDemo {

    private static String A = "A";

    private static String B = "B";

    public static void main(String[] args) {

        new DeadLockDemo().deadLock();

    }

    private static void deadLock() {

        Thread t1 = new Thread(new Runnable() {

            @Override

            public void run() {

                synchronized (A) {

                    try {

                        Thread.currentThread().sleep(2000);

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    }

                    synchronized (B) {

                        System.out.println("t1");

                    }

                }

            }

        });

        Thread t2 = new Thread(new Runnable() {

            @Override

            public void run() {

                synchronized (B) {

                    synchronized (A) {

                        System.out.println("t2");

                    }

                }

            }

        });

        t1.start();

        t2.start();

    }

}

```

$ jps

15668 DeadLockDemo

$ jstack 15668

避免死锁常见方法：

- 避免一个线程同时获取多个锁

- 避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源

- 尝试使用定时锁，使用lock.tryLock(timeout)替代使用内部锁（synchronized）机制

- 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况

## 1.3 资源限制的挑战

并发时，程序的执行速度受限于硬件资源或软件资源

例如，服务器的带宽只有2Mb/s，某个资源的下载速度是1Mb/s每秒，系统启动10个线程下载资源，下载速度不会变成10Mb/s！！！

硬件资源限制有带宽的上传/下载速度、硬盘读写速度和CPU的处理速度。

软件资源限制有数据库的连接数和socket连接数等。

如果将某段串行的代码并发执行，因为受限于资源，仍然在串行执行，这时候程序不仅不会加快执行，反而会更慢，因为增加了上下文切换和资源调度的时间。

对于硬件资源限制，可以考虑使用集群并行执行程序。

既然单机的资源有限制，那么就让程序在多机上运行。

比如使用ODPS、Hadoop或者自己搭建服务器集群，不同的机器处理不同的数据。可以通过“数据ID%机器数”，计算得到一个机器编号，然后由对应编号的机器处理这笔数据。

对于软件资源限制，可以考虑使用资源池将资源复用。

比如使用连接池将数据库和Socket连接复用，或者在调用对方webservice接口获取数据时，只建立一个连接。

在资源限制情况下进行并发编程，需要根据不同的资源限制调整程序的并发度，比如下载文件程序依赖于两个资源——带宽和硬盘读写速度。有数据库操作时，涉及数据库连接数，如果SQL语句执行非常快，而线程的数量比数据库连接数大很多，则某些线程会被阻塞，等待数据库连接。

【强烈建议多使用JDK并发包提供的并发容器和工具类来解决并发问题，因为这些类都已经通过了充分的测试和优化】

# **java并发的底层实现原理**

3个方面：

- CPU

- JVM

- 语言规范和内存模型

- 编译器和处理器

【并发机制实现依赖于JVM的实现和CPU的指令】

源码 --> 编译 --> 字节码 --> 类加载器 --> JVM --> 汇编指令 ---> CPU执行

【Java中的大部分容器和并发框架都依赖于volatile和原子操作的实现原理】

## 2.1 volatile

synchronized和volatile在并发中都扮演着重要的角色。

volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”。

可见性：

当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

volatile变量修饰符使用恰当的话，它比synchronized的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。

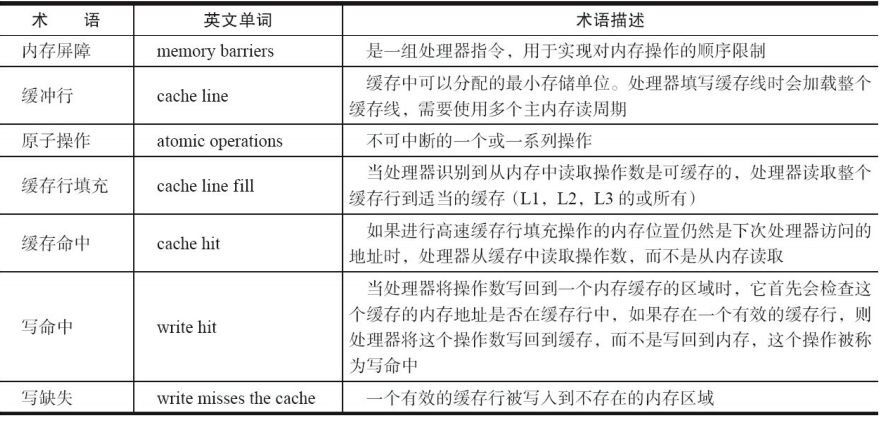
### 2.1.1 volatile的定义与实现原理

Java语言规范第3版中对volatile的定义如下：

Java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。

Java语言提供了volatile，在某些情况下比锁要更加方便。如果一个字段被声明成volatile，Java线程内存模型（JMM）确保所有线程看到这个变量的值是一致的。

与其实现原理相关的CPU术语与说明：



volatile是如何来保证可见性的呢？

在X86处理器下通过工具获取JIT编译器生成的汇编指令来查看对volatile进行写操作时，CPU会做什么事情。

instance = new Singleton(); // instance是volatile变量

转变成汇编代码，

0x01a3de1d: movb $0×0,0×1104800(%esi);0x01a3de24: **lock** addl $0×0,(%esp);

有volatile变量修饰的共享变量进行写操作的时候会多出第二行汇编代码，通过查IA-32架构软件开发者手册可知，Lock前缀的指令在多核处理器下会引发了两件事情：

1）将当前处理器缓存行的数据写回到系统内存。

2）这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部

缓存（L1，L2或其他）后再进行操作，但操作完不知道何时会写到内存。

如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。所以，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现**缓存一致性协议**，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

总结出了volatile的两条实现原则：

1）Lock前缀指令会引起处理器缓存回写到内存。

Lock前缀指令导致在执行指令期间，声言处理器的LOCK#信号。

在多处理器环境中，LOCK#信号确保在声言该信号期间，处理器可以独占任何共享内存（因为它会锁住总线，导致其他CPU不能访问总线，不能访问总线就意味着不能访问系统内存）。但是，在最近的处理器里，LOCK#信号一般不锁总线，而是锁缓存，毕竟锁总线开销的比较大。

在8.1.4节有详细说明锁定操作对处理器缓存的影响，对于Intel486和Pentium处理器，在锁操作时，总是在总线上声言LOCK#信号。但在P6和目前的处理器中，如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反，它会锁定这块内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性，此操作被称为“**缓存锁定**”，缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据。

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。

IA-32处理器和Intel 64处理器使用MESI（修改、独占、共享、无效）控制协议去维护内部缓存和其他处理器缓存的一致性。在多核处理器系统中进行操作的时候，IA-32和Intel 64处理器能嗅探其他处理器访问系统内存和它们的内部缓存。

处理器使用嗅探技术保证它的内部缓存、系统内存和其他处理器的缓存的数据在总线上保持一致。

例如，在Pentium和P6 family处理器中，如果通过嗅探一个处理器来检测其他处理器打算写内存地址，而这个地址当前处于共享状态，那么正在嗅探的处理器将使它的缓存行无效，在下次访问相同内存地址时，强制执行缓存行填充。

### 2.1.2 volatile的使用优化

著名的Java并发编程大师Doug lea在JDK 7的并发包里新增一个队列集合类LinkedTransferQueue.java，它在使用volatile变量时，用一种追加字节的方式来优化队列出队和入队的性能。

LinkedTransferQueue.java的代码如下。

```java

/\*\* 队列中的头部节点 \*/

private transient f?inal PaddedAtomicReference<QNode> head;

/\*\* 队列中的尾部节点 \*/

private transient f?inal PaddedAtomicReference<QNode> tail;

static f?inal class PaddedAtomicReference <T> extends AtomicReference<T> {

        // 使用很多4个字节的引用追加到64个字节

        Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;

        PaddedAtomicReference(T r) {

        super(r);

    }

}

public class AtomicReference <V> implements java.io.Serializable {

    private volatile V value;

    // 省略其他代码

}

```

【java8已经改写了】

追加字节能优化性能？

这种方式看起来很神奇，但如果深入理解处理器架构就能理解其中的奥秘。

LinkedTransferQueue这个类，它使用一个内部类类型来定义队列的头节点（head）和尾节点（tail），而这个内部类PaddedAtomicReference相对于父类

AtomicReference只做了一件事情，就是将共享变量追加到64字节。

我们可以来计算下，一个对象的引用占4个字节，它追加了15个变量（共占60个字节），再加上父类的value变量，一共64个字节。

为什么追加64字节能够提高并发编程的效率呢？

因为对于英特尔酷睿i7、酷睿、Atom和NetBurst，以及Core Solo和Pentium M处理器的L1、L2或L3缓存的高速缓存行是64个字节宽，不支持部分填充缓存行，这意味着，如果队列的头节点和尾节点都不足64字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头、尾节点，当一个处理器试图修改头节点时，会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作则需要不停修改头节点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。

Doug Lea使用追加到64字节的方式来填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行，使头、尾节点在修改时不会互相锁定。

那么是不是在使用volatile变量时都应该追加到64字节呢？

不是的。

在两种场景下不应该使用这种方式。

1. 缓存行非64字节宽的处理器。如P6系列和奔腾处理器，它们的L1和L2高速缓存行是32个字节宽。

2. 共享变量不会被频繁地写。因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定的性能消耗，如果共享变量不被频繁写的话，锁的几率也非常小，就没必要通过追加字节的方式来避免相互锁定。

不过这种追加字节的方式在Java 7下可能不生效，因为Java 7变得更加智慧，它会淘汰或重新排列无用字段，需要使用其他追加字节的方式。

## 2.2 synchronized的实现原理与应用

很多人都会称呼它为重量级锁。

但是，随着Java SE 1.6对synchronized进行了各种优化之后，有些情况下它就并不那么重了。

Java SE 1.6中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁，以及锁的存储结构和升级过程。

synchronized实现同步的基础：

Java中的每一个对象都可以作为锁。

具体表现为以下3种形式：

1. 对于普通同步方法，锁是当前实例对象

2. 对于静态同步方法，锁是当前类的Class对象

3. 对于同步方法块，锁是Synchonized括号里配置的对象

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

JVM规范中可以看到Synchonized在JVM里的实现原理：

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。

代码块同步是使用monitorenter和monitorexit指令实现的，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明。但是，方法的同步同样可以使用这两个指令来实现。

monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。

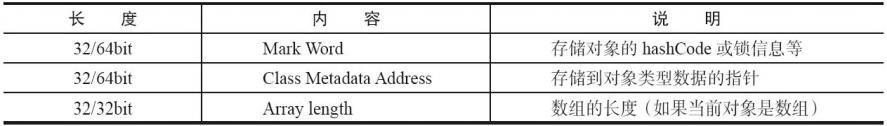
任何对象都有一个monitor与之关联，当且一个monitor被持有后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁。

### 2.2.1 Java对象头

synchronized用的锁是存在Java对象头里的。

如果对象是数组类型，则虚拟机用3个字宽（Word）存储对象头，如果对象是非数组类型，则用2字宽存储对象头。在32位虚拟机中，1字宽等于4字节，即32bit。

Java对象头的长度



Java对象头里的Mark Word里默认存储对象的HashCode、分代年龄和锁标记位。

Java对象头的存储结构

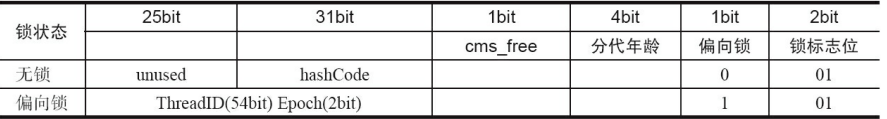


在运行期间，Mark Word里存储的数据会随着锁标志位的变化而变化。

Mark Word可能变化为存储以下4种数据



在64位虚拟机下，Mark Word是64bit大小的



### 2.2.2 锁的升级与对比

Java SE 1.6为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”。

在Java SE 1.6中，锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态。

这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。

#### 2.2.2.1 偏向锁

HotSpot的作者经过研究发现，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。

偏向锁的撤销

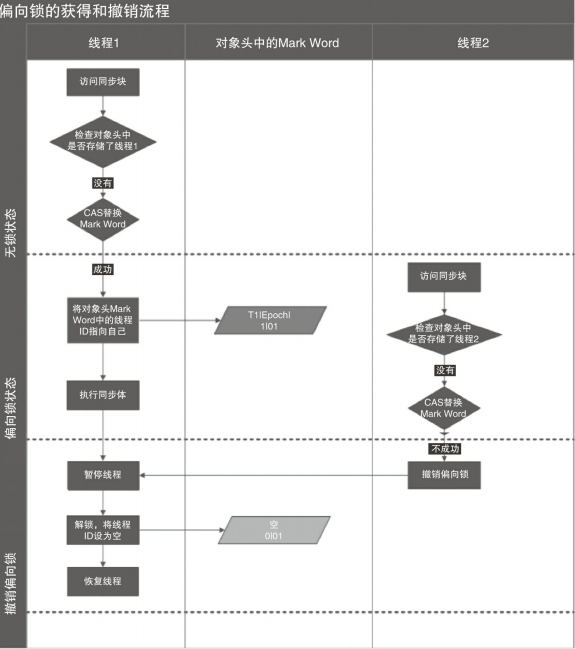
偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。

偏向锁的撤销，需要等待全局安全点（在这个时间点上没有正在执行的字节码）。

它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态；如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他

线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。

偏向锁初始化的流程



【程1演示了偏向锁初始化的流程，线程2演示了偏向锁撤销的流程】

关闭偏向锁

偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟：

-XX:BiasedLockingStartupDelay=0

如果你确定应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁：

-XX:-UseBiasedLocking=false，

那么程序默认会进入轻量级锁状态。

#### 2.2.2.2 轻量级锁

（1）轻量级锁加锁

线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并

将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。

然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。

如果成功，当前线程获得锁。

如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

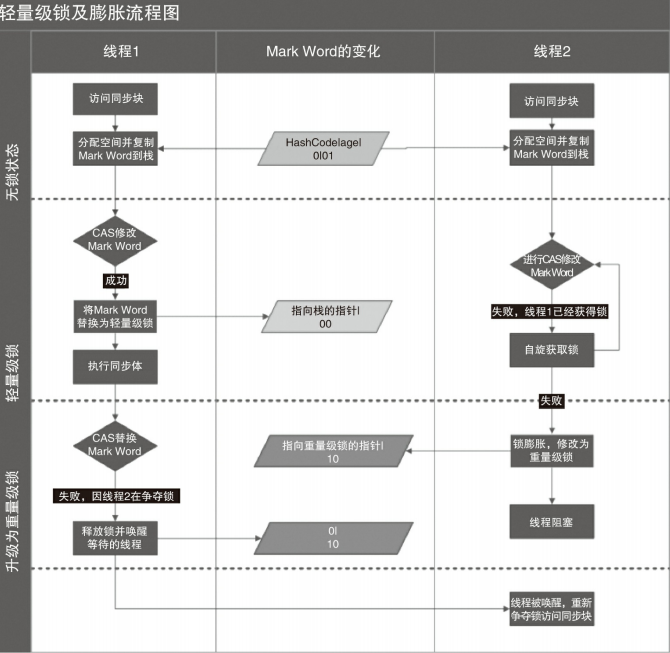
（2）轻量级锁解锁

轻量级解锁时，会使用原子的CAS操作将Displaced Mark Word替换回到对象头。

如果成功，则表示没有竞争发生。

如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

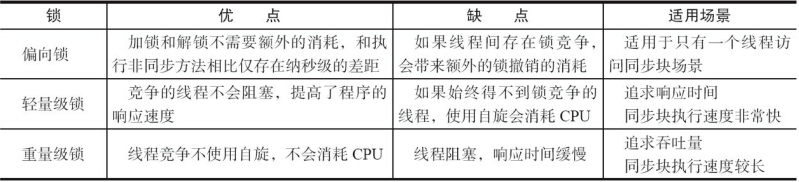
两个线程同时争夺锁，导致锁膨胀的流程图。



因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。

当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

#### 2.2.2.3 锁的优缺点对比



## 2.3 原子操作的实现原理

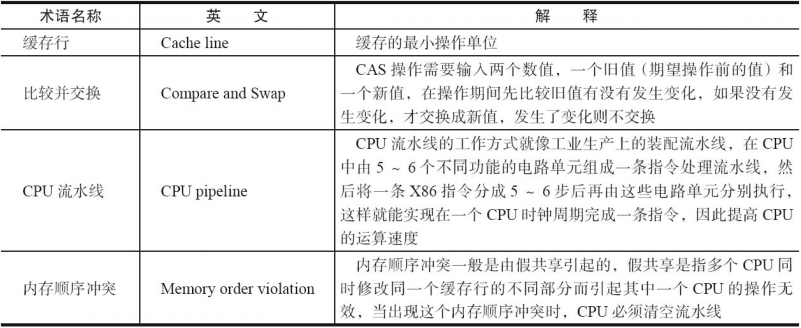
原子（atomic）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为“不可被中断的一个或一系列操作”。

在多处理器上实现原子操作就变得有点复杂。

在Intel处理器和Java里是如何实现原子操作的。

### 2.3.1 术语定义

在了解原子操作的实现原理前，先要了解一下相关的术语



### 2.3.2 处理器如何实现原子操作

32位IA-32处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操

作。

首先处理器会自动保证基本的内存操作的原子性。

处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。

Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。

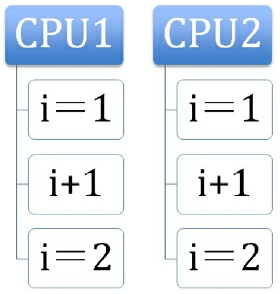
但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

（1）使用总线锁保证原子性

第一个机制是通过总线锁保证原子性。

如果多个处理器同时对共享变量进行读改写操作（i++就是经典的读改写操作），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致。

举个例子，如果i=1，我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2。



原因可能是多个处理器同时从各自的缓存中读取变量i，分别进行加1操作，然后分别写入系统内存中。

那么，想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。

所谓总线锁就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。

（2）使用缓存锁保证原子性

第二个机制是通过缓存锁定来保证原子性。

在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但总线锁定把CPU和内存之间的通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1、L2和L3高速缓存里，那么原子操作就可以直接在

处理器内部缓存中进行，并不需要声明总线锁，在Pentium 6和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。

所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言LOCK#信号，而是修改内部的内存地址，并允许它的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由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如图2-3所示的例子中，当CPU1修改缓存行中的i时使用了缓存锁定，那么CPU2就不能同时缓存i的缓存行。

但是有两种情况下处理器不会使用缓存锁定。

第一种情况是：

当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行

（cache line）时，则处理器会调用总线锁定。

第二种情况是：

有些处理器不支持缓存锁定。对于Intel 486和Pentium处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

针对以上两个机制，我们通过Intel处理器提供了很多Lock前缀的指令来实现。例如，位测试和修改指令：BTS、BTR、BTC；交换指令XADD、CMPXCHG，以及其他一些操作数和逻辑指令（如ADD、OR）等，被这些指令操作的内存区域就会加锁，导致其他处理器不能同时访问它。

### 2.3.3 Java如何实现原子操作

在Java中可以通过锁和循环CAS的方式来实现原子操作。

（1）使用循环CAS实现原子操作

JVM中的CAS操作正是利用了处理器提供的CMPXCHG指令实现的。自旋CAS实现的基本思路就是循环进行CAS操作直到成功为止，以下代码实现了一个基于CAS线程安全的计数器

方法safeCount和一个非线程安全的计数器count。

从Java 1.5开始，JDK的并发包里提供了一些类来支持原子操作，如AtomicBoolean（用原子方式更新的boolean值）、AtomicInteger（用原子方式更新的int值）和AtomicLong（用原子方式更新的long值）。

这些原子包装类还提供了有用的工具方法，比如以原子的方式将当前值自增1和自减1。

（2）CAS实现原子操作的三大问题

在Java并发包中有一些并发框架也使用了自旋CAS的方式来实现原子操作，比如

LinkedTransferQueue类的xfer()方法。

CAS虽然很高效地解决了原子操作，但是CAS仍然存在三大问题。

1. ABA问题

2. 循环时间长开销大

3. 只能保证一个共享变量的原子操作

1）ABA问题。

因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。

ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A→B→A就会变成1A→2B→3A。

从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

```java

package java.util.concurrent.atomic;

public class AtomicStampedReference<V> {

    public boolean compareAndSet(

        V expectedReference, // 预期引用

        V newReference, // 更新后的引用

        int expectedStamp, // 预期标志

        int newStamp // 更新后的标志

    )

}

```

2）循环时间长开销大。

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

如果JVM能支持处理器提供的pause指令，那么效率会有一定的提升。

pause指令有两个作用：

第一，它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零；

第二，它可以避免在退出循环的时候因内存顺序冲突（Memory Order Violation）而引起CPU流水线被清空（CPU Pipeline Flush），从而提高CPU的执行效率。

3）只能保证一个共享变量的原子操作。

当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。

还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。

比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。

从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

（3）使用锁机制实现原子操作

锁机制保证了只有获得锁的线程才能够操作锁定的内存区域。

JVM内部实现了很多种锁机制，有偏向锁、轻量级锁和互斥锁。

有意思的是除了偏向锁，JVM实现锁的方式都用了循环CAS，即当一个线程想进入同步块的时候使用循环CAS的方式来获取锁，当它退出同步块的时候使用循环CAS释放锁。

# 3. Java内存模型（JMM）

【Java线程之间的通信对程序员完全透明，内存可见性问题很容易困扰Java程序员】

关键点：

内存模型的相关概念

内存模型中的顺序一致性

重排序与顺序一致性内存模型

同步原语

3个同步原语（synchronized、volatile和final）的内存语义

重排序规则在处理器中的实现

内存模型的设计

Java内存模型的设计原理

处理器内存模型和顺序一致性内存模型的关系

## 3.1 Java内存模型的基础

### 3.1.1 并发编程模型的两个关键问题

在并发编程中，需要处理两个关键问题：

线程之间如何通信及线程之间如何同步（这里的线程是指并发执行的活动实体）。

通信是指线程之间以何种机制来交换信息。

在命令式编程中，线程之间的通信机制有两种：

共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信。

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过发送消息来显式进行通信。

同步是指程序中用于控制不同线程间操作发生相对顺序的机制。

在共享内存并发模型里，同步是显式进行的。

程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java的并发采用的是共享内存模型，Java线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。

如果编写多线程程序的Java程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

### 3.1.2 Java内存模型的抽象结构

在Java中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享

（“共享变量”这个术语代指实例域，静态域和数组元素）。

局部变量（Local Variables），方法定义参数（Java语言规范称之为Formal Method Parameters）和异常处理器参数（Exception Handler Parameters）不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

Java线程之间的通信由Java内存模型（本文简称为JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。

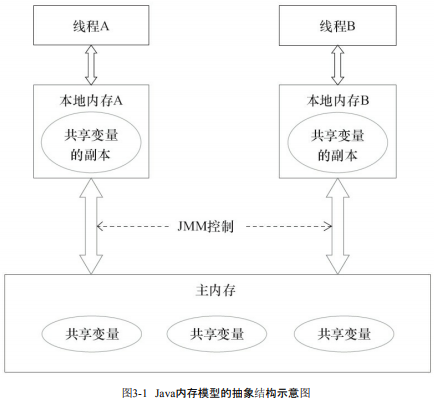
从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：

线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地

内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。

本地内存是JMM的一个抽象概念，并不真实存在。

它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。



如果线程A与线程B之间要通信的话，必须要经历下面2个步骤：

1）线程A把本地内存A中更新过的共享变量刷新到主内存中去

2）线程B到主内存中去读取线程A之前已更新过的共享变量

本地内存A和本地内存B由主内存中共享变量x的副本。

假设初始时，这3个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。

当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内

存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要

经过主内存。

JMM通过控制主内存与每个线程的本地内存之间的交互，来为Java程序员提供内存可见性保证。

### 3.1.3 从源代码到指令序列的重排序

在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。

重排序分3种类型：

1）编译器优化的重排序。

编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2）指令级并行的重排序。

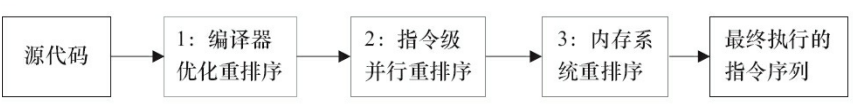
现代处理器采用了指令级并行技术（Instruction-Level Parallelism，ILP）来将多条指令重叠执行。

如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3）内存系统的重排序。

由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会分别经历下面3种重排序



上述的1属于编译器重排序，2和3属于处理器重排序。

这些重排序可能会导致多线程程序出现内存可见性问题。

对于编译器，JMM的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。

对于处理器重排序，JMM的处理器重排序规则会要求Java编译器在生成指令序列时，插入特定类型的内存屏障（Memory Barriers，Intel称之为Memory Fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序。

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

### 3.1.4 并发编程模型的分类

现代的处理器使用写缓冲区临时保存向内存写入的数据。

写缓冲区可以保证指令流水线持续运行，它可以避免由于处理器停顿下来等待向内存写入数据而产生的延迟。

同时，通过以批处理的方式刷新写缓冲区，以及合并写缓冲区中对同一内存地址的多次写，减少对内存总线的占用。

虽然写缓冲区有这么多好处，但每个处理器上的写缓冲区，仅仅对它所在的处理器可见。

这个特性会对内存操作的执行顺序产生重要的影响：处理器对内存的读/写操作的执行顺序，不一定与内存实际发生的读/写操作顺序一致！

由于写缓冲区仅对自己的处理器可见，它会导致处理器执行内存操作的顺序可能会与内存实际的操作执行顺序不一致。

由于现代的处理器都会使用写缓冲区，因此现代的处理器都会允许对写-读操作进行重排序。

为了保证内存可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。JMM把内存屏障指令分为4类：



### 3.1.5 happens-before简介

从JDK 5开始，Java使用新的JSR-133内存模型。

JSR-133使用happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系。

这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

与程序员密切相关的happens-before规则如下：

1. 程序顺序规则：

一个线程中的每个操作，happens-before于该线程中的任意后续操作。

2. 监视器锁规则：

对一个锁的解锁，happens-before于随后对这个锁的加锁。

3. volatile变量规则：

对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4. 传递性：

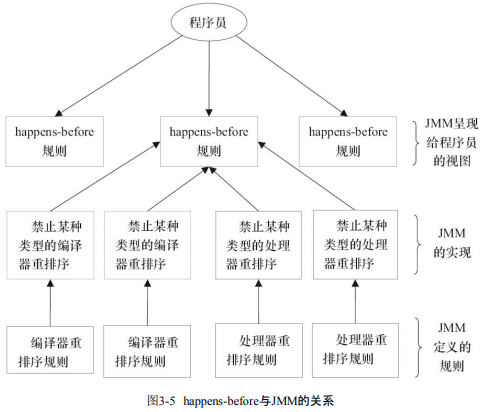
如果A happens-before B，且B happens-before C，那么A happens-before C。

注意：

两个操作之间具有happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行！

happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前（the first is visible to and ordered before the second）。

happens-before与JMM的关系：



一个happens-before规则对应于一个或多个编译器和处理器重排序规则。

对于Java程序员来说，happens-before规则简单易懂，它避免Java程序员为了理解JMM提供的内存可见性保证而去学习复杂的重排序规则以及这些规则的具体实现方法。

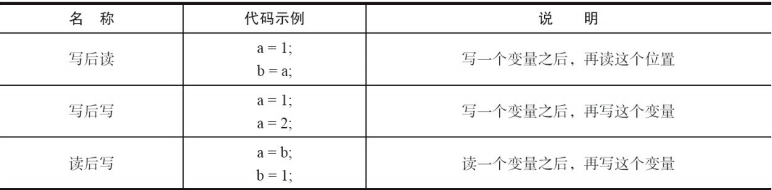
## 3.2 重排序

【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重新排序的一种手段。】

### 3.2.1 数据依赖性

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间

就存在数据依赖性。数据依赖分为下列3种类型：



上面3种情况，只要重排序两个操作的执行顺序，程序的执行结果就会被改变。

前面提到过，编译器和处理器可能会对操作做重排序。

编译器和处理器在重排序时，会遵守数据依赖性，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针对单个处理器中执行的指令序列和单个线程中执行的操作，

不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

### 3.2.2 as-if-serial语义

as-if-serial语义的意思是：

不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。

编译器、runtime和处理器都必须遵守as-if-serial语义。

为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。

但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器、runtime和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。

as-if-serial语义使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。

### 3.2.3 程序顺序规则

在计算机中，软件技术和硬件技术有一个共同的目标：在不改变程序执行结果的前提下，尽可能提高并行度。

编译器和处理器遵从这一目标，从happens-before的定义我们可以看出，JMM同样遵从这一目标。

### 3.2.4 重排序对多线程的影响

示例：

```java

class ReorderExample {

    int a = 0;

    boolean flag = false;

    public void writer() {

        a = 1; // 1

        flag = true; // 2

    }

    Public void reader() {

        if (flag) { // 3

            int i = a \* a; // 4

            // todo

        }

    }

}

```

这2个方法当然不是线程安全的。

在程序中，操作3和操作4存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。

为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。

以处理器的猜测执行为例，执行线程B的处理器可以提前读取并计算a\*a，然后把计算结果临时保存到一个名为重排序缓冲（Reorder Buffer，ROB）的硬件缓存中。

当操作3的条件判断为真时，就把该计算结果写入变量i中。

重排序在这里破坏了多线程程序的语义！

在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是as-if-serial语义允许对存在控制依赖的操作做重排序的原因）。

但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。

## 3.3 顺序一致性

顺序一致性内存模型是一个理论参考模型，在设计的时候，处理器的内存模型和编程语

言的内存模型都会以顺序一致性内存模型作为参照。

### 3.3.1 数据竞争与顺序一致性

当程序未正确同步时，就可能会存在数据竞争。

Java内存模型规范对数据竞争的定义如下：

在一个线程中写一个变量，

在另一个线程读同一个变量，

而且写和读没有通过同步来排序。

当代码中包含数据竞争时，程序的执行往往产生违反直觉的结果。

如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

JMM对正确同步的多线程程序的内存一致性做了如下保证：

如果程序是正确同步的，程序的执行将具有顺序一致性（Sequentially Consistent）——即程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同。

这对于程序员来说是一个极强的保证。

这里的同步是指广义上的同步，包括对常用同步原语（synchronized、volatile和final）的正确使用。

### 3.3.2 顺序一致性内存模型

顺序一致性内存模型是一个被计算机科学家理想化了的理论参考模型，它为程序员提供

了极强的内存可见性保证。

顺序一致性内存模型有两大特性：

1） 一个线程中的所有操作必须按照程序的顺序来执行。

2）（不管程序是否同步）所有线程都只能看到一个单一的操作执行顺序。

在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

在概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关

可以连接到任意一个线程，同时每一个线程必须按照程序的顺序来执行内存读/写操作。

在任意时间点最多只能有一个线程可以连接到内存。

当多个线程并发执行时，内存模型保证所有线程的所有内存读/写操作串行化（即在顺序一致性模型中，所有操作之间具有全序关系）。

当然，这是个理想模型。在JMM中就没有这个保证。

### 3.3.3 同步程序的顺序一致性效果

正确的同步：

```java

class SynchronizedExample {

    int a = 0;

    boolean flag = false;

    public synchronized void writer() { // 获取锁

        a = 1;

        flag = true;

    } // 释放锁

    public synchronized void reader() { // 获取锁

        if (flag) {

            int i = a;

            // todo

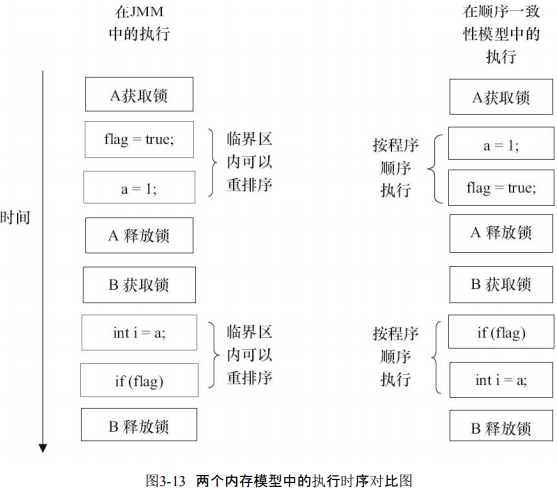
        } // 释放锁

    }

}

```

根据JMM规范，该程序的执行结果将与该程序在顺序一致性模型中的执行结果相同。



顺序一致性模型中，所有操作完全按程序的顺序串行执行。

而在JMM中，临界区内的代码可以重排序（但JMM不允许临界区内的代码“逸出”到临界区之外，那样会破坏监视器的语义）。

JMM会在退出临界区和进入临界区这两个关键时间点做一些特别处理，使得线程在这两

个时间点具有与顺序一致性模型相同的内存视图（具体细节后文会说明）。

虽然线程A在临界区内做了重排序，但由于监视器互斥执行的特性，这里的线程B根本无法“观察”到线程A在临界区内的重排序。

这种重排序既提高了执行效率，又没有改变程序的执行结果。

JMM在具体实现上的基本方针为：

在不改变（正确同步的）程序执行结果的前提下，尽可能地为编译器和处理器的优化打开方便之门。

### 3.3.4 未同步程序的执行特性

对于未同步或未正确同步的多线程程序，JMM只提供最小安全性：

线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0，Null，False），JMM保证线程读操作读取到的值不会无中生有（Out Of Thin Air）的冒出来。

为了实现最小安全性，JVM在堆上分配对象时，首先会对内存空间进行清零，然后才会在上面分配对象（JVM内部会同步这两个操作）。

因此，在已清零的内存空间（Pre-zeroed Memory）分配对象时，域的默认初始化已经完成了。

JMM不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致。

因为如果想要保证执行结果一致，JMM需要禁止大量的处理器和编译器的优化，这对程序的执行性能会产生很大的影响。

而且未同步程序在顺序一致性模型中执行时，整体是无序的，其执行结果往往无法预知。而且，保证未同步程序在这两个模型中的执行结果一致没什么意义。

未同步程序在JMM中的执行时，整体上是无序的，其执行结果无法预知。

未同步程序在两个模型中的执行特性有如下几个差异：

1）顺序一致性模型保证单线程内的操作会按程序的顺序执行，而JMM不保证单线程内的

操作会按程序的顺序执行（比如上面正确同步的多线程程序在临界区内的重排序）。

2）顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而JMM不保证所有线程能看到一致的操作执行顺序。

3）JMM不保证对64位的long型和double型变量的写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。

这个差异与处理器总线的工作机制密切相关。

在计算机中，数据通过总线在处理器和内存之间传递。

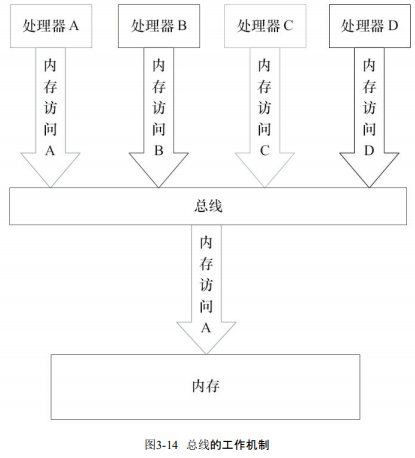
每次处理器和内存之间的数据传递都是通过一系列步骤来完成的，这一系列步骤称之为总线事务（Bus Transaction）。

总线事务包括读事务（Read Transaction）和写事务（Write Transaction）。

读事务从内存传送数据到处理器，写事务从处理器传送数据到内存，每个事务会读/写内存中一个或多个物理上连续的字。

这里的关键是，总线会同步试图并发使用总线的事务。

在一个处理器执行总线事务期间，总线会禁止其他的处理器和I/O设备执行内存的读/写。



假设处理器A，B和C同时向总线发起总线事务，这时总线仲裁（Bus Arbitration）会对竞争做出裁决，这里假设总线在仲裁后判定处理器A在竞争中获胜（总线仲裁会确保所有处理器都能公平的访问内存）。

此时处理器A继续它的总线事务，而其他两个处理器则要等待处理器A的总线事务完成后才能再次执行内存访问。

假设在处理器A执行总线事务期间（不管这个总线事务是读事务还是写事务），处理器D向总线发起了总线事务，此时处理器D的请求会被总线禁止。

总线的这些工作机制可以把所有处理器对内存的访问以串行化的方式来执行。

在任意时间点，最多只能有一个处理器可以访问内存。

这个特性确保了单个总线事务之中的内存读/写操作具有原子性。

在一些32位的处理器上，如果要求对64位数据的写操作具有原子性，会有比较大的开销。

为了照顾这种处理器，Java语言规范鼓励但不强求JVM对64位的long型变量和double型变量的写操作具有原子性。

当JVM在这种处理器上运行时，可能会把一个64位long/double型变量的写操作拆分为两个32位的写操作来执行。

这两个32位的写操作可能会被分配到不同的总线事务中执行，此时对这个64位变量的写操作将不具有原子性。

当单个内存操作不具有原子性时，可能会产生意想不到后果。

## 3.4 volatile的内存语义

【当声明共享变量为volatile后，对这个变量的读/写将会很特别。】

### 3.4.1 volatile的特性

示例：

```java

class VolatileFeaturesExample {

    volatile long vl = 0L; // 使用volatile声明64位的long型变量

    public void set(long l) {

        vl = l; // 单个volatile变量的写

    }

    public void getAndIncrement () {

        vl++; // 复合（多个）volatile变量的读/写

    }

    public long get() {

        return vl; // 单个volatile变量的读

    }

}

```

假设有多个线程分别调用上面程序的3个方法，这个程序在语义上和下面程序等价。

```java

class VolatileFeaturesExample {

    long vl = 0L; // 64位的long型普通变量

    public synchronized void set(long l) { // 对单个的普通变量的写用同一个锁同步

        vl = l;

    }

    public void getAndIncrement () { // 普通方法调用

        long temp = get(); // 调用已同步的读方法

        temp += 1L; // 普通写操作

        set(temp); // 调用已同步的写方法

    }

    public synchronized long get() { // 对单个的普通变量的读用同一个锁同步

        return vl;

    }

}

```

一个volatile变量的单个读/写操作，与一个普通变量的读/写操作都是使用同一个锁来同步，它们之间的执行效果相同。

锁的happens-before规则保证释放锁和获取锁的两个线程之间的内存可见性，这意味着对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

锁的语义决定了临界区代码的执行具有原子性。

这意味着，即使是64位的long型和double型变量，只要它是volatile变量，对该变量的读/写就具有原子性。

如果是多个volatile操作或类似于volatile++这种复合操作，这些操作整体上不具有原子性。

总结：

volatile变量自身具有下列特性：

1. 可见性

对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

2. 原子性

对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

### 3.4.2 volatile写-读建立的happens-before关系

对程序员来说，volatile对线程的内存可见性的影响比volatile自身的特性更为重要，也更需要我们去关注。

从JSR-133开始（即从JDK5开始），volatile变量的写-读可以实现线程之间的通信。

从内存语义的角度来说，volatile的写-读与锁的释放-获取有相同的内存效果：volatile写和锁的释放有相同的内存语义

volatile读与锁的获取有相同的内存语义

```java

class VolatileExample {

    int a = 0;

    volatile boolean flag = false;

    public void writer() {

        a = 1; // 1

        flag = true; // 2

    }

    public void reader() {

        if (flag) { // 3

            int i = a; // 4

            // todo

        }

    }

}

```

这个过程建立的happens-before关系可以分为3类：

1）根据程序次序规则，1 happens-before 2;3 happens-before 4。

2）根据volatile规则，2 happens-before 3。

3）根据happens-before的传递性规则，1 happens-before 4。

所以，对一个volatile变量的写，将立即对所有读可见。

### 3.4.3 volatile写-读的内存语义

volatile写的内存语义：

当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。

volatile读的内存语义：

当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

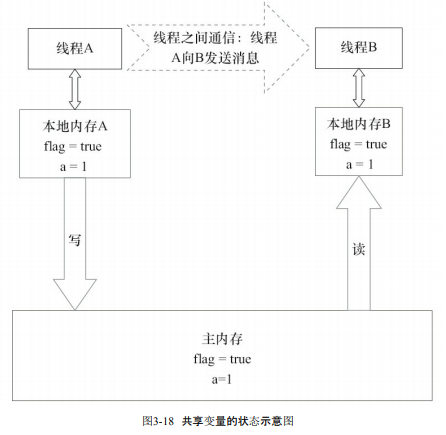
在读线程B读一个volatile变量后，写线程A在写这个volatile变量之前所有可见的共享变量的值都将立即变得对读线程B可见。

总结：

1. 线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所做修改的）消息。

2. 线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。

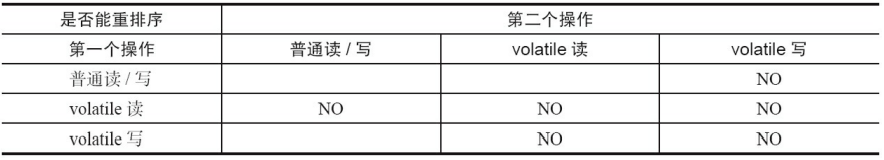
3. 线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。



### 3.4.4 volatile内存语义的实现

为了实现volatile内存语义，JMM会分别限制编译器重排序和处理器重排序这两种类型的重排序类型。

JMM针对编译器制定的volatile重排序规则表。



举例来说，第三行最后一个单元格的意思是：

在程序中，当第一个操作为普通变量的读或写时，如果第二个操作为volatile写，则编译器不能重排序这两个操作。

从表3-5我们可以看出。

1. 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。

这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。

2. 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。

这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。

3. 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能。

为此，JMM采取保守策略。下面是基于保守策略的JMM内存屏障插入策略。

在每个volatile写操作的前面插入一个StoreStore屏障。

在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

上述内存屏障插入策略非常保守，但它可以保证在任意处理器平台，任意的程序中都能得到正确的volatile内存语义。

不同的处理器有不同“松紧度”的处理器内存模型，内存屏障的插入还可以根据具体的处理器内存模型继续优化。

### 3.4.5 JSR-133为什么要增强volatile的内存语义

在旧的内存模型中，volatile的写-读没有锁的释放-获所具有的内存语义。

为了提供一种比锁更轻量级的线程之间通信的机制，JSR-133专家组决定增强volatile的内存语义：

严格限制编译器和处理器对volatile变量与普通变量的重排序，确保volatile的写-读和锁的释放-获取具有相同的内存语义。

从编译器重排序规则和处理器内存屏障插入策略来看，只要volatile变量与普通变量之间的重排序可能会破坏volatile的内存语义，这种重排序就会被编译器重排序规则和处理器内存屏障插入策略禁止。

由于volatile仅仅保证对单个volatile变量的读/写具有原子性，而锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性。

在功能上，锁比volatile更强大；在可伸缩性和执行性能上，volatile更有优势。

如果想在程序中用volatile代替锁，请一定谨慎。

## 3.5 锁的内存语义

【锁可以让临界区互斥执行。】

### 3.5.1 锁的释放-获取建立的happens-before关系

锁是Java并发编程中最重要的同步机制。

锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。

锁释放-获取的示例

```java

class MonitorExample {

    int a = 0;

    public synchronized void writer() { // 1

        a++; // 2

    } // 3

    public synchronized void reader() { // 4

        int i = a; // 5

        // todo

    } // 6

}

```

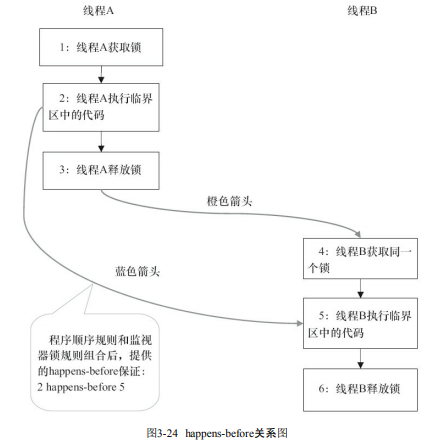
假设线程A执行writer()方法，随后线程B执行reader()方法。

根据happens-before规则，这个过程包含的happens-before关系可以分为3类：

1）根据程序次序规则，1 happens-before 2,2 happens-before 3;4 happens-before 5,5 happensbefore 6。

2）根据监视器锁规则，3 happens-before 4。

3）根据happens-before的传递性，2 happens-before 5。



### 3.5.2 锁的释放和获取的内存语义

当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的

临界区代码必须从主内存中读取共享变量。

对比锁释放-获取的内存语义与volatile写-读的内存语义可以看出：

锁释放与volatile写有相同的内存语义

锁获取与volatile读有相同的内存语义

总结：

1. 线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。

2. 线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。

3. 线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

### 3.5.3 锁内存语义的实现

借助ReentrantLock的源代码，来分析锁内存语义的具体实现机制。

```java

class ReentrantLockExample {

    int a = 0;

    ReentrantLock lock = new ReentrantLock();

    public void writer() {

        lock.lock(); // 获取锁

        try {

            a++;

        } finally {

            lock.unlock(); // 释放锁

        }

    }

    public void reader () {

        lock.lock(); // 获取锁

        try {

            int i = a;

            // todo

        } finally {

            lock.unlock(); // 释放锁

        }

    }

}

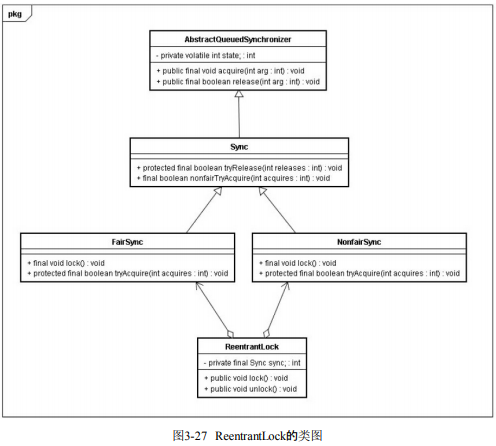
```

在ReentrantLock中，调用lock()方法获取锁；调用unlock()方法释放锁。

ReentrantLock的实现依赖于Java同步器框架AbstractQueuedSynchronizer（AQS）。

AQS使用一个整型的volatile变量（命名为state）来维护同步状态，这个volatile变量是ReentrantLock内存语义实现的关键。

ReentrantLock的类图



ReentrantLock分为公平锁和非公平锁。

```java

package java.util.concurrent.locks;

public class ReentrantLock implements Lock, java.io.Serializable {

    private final Sync sync;

    abstract static class Sync extends AbstractQueuedSynchronizer {

    }

    static final class NonfairSync extends Sync {

    }

    static final class FairSync extends Sync {

    }

}

```

使用公平锁时，加锁方法lock()调用轨迹如下：

1）ReentrantLock:lock()

2）FairSync:lock()

3）AbstractQueuedSynchronizer:acquire(int arg)

4）ReentrantLock:tryAcquire(int acquires)

在第4步真正开始加锁，下面是该方法的源代码。

```java

protected final boolean tryAcquire(int acquires) {

    final Thread current = Thread.currentThread();

    int c = getState(); // 获取锁的开始，首先读volatile变量state

    if (c == 0) {

        if (isFirst(current) &&

            compareAndSetState(0, acquires)) {

            setExclusiveOwnerThread(current);

            return true;

        }

    }

    else if (current == getExclusiveOwnerThread()) {

        int nextc = c + acquires;

        if (nextc < 0)

            throw new Error("Maximum lock count exceeded");

        setState(nextc);

        return true;

    }

    return false;

}

```

在使用公平锁时，解锁方法unlock()调用轨迹如下：

1）ReentrantLock:unlock()

2）AbstractQueuedSynchronizer:release(int arg)

3）Sync:tryRelease(int releases)

在第3步真正开始释放锁，下面是该方法的源代码。

```java

protected final boolean tryRelease(int releases) {

    int c = getState() - releases;

    if (Thread.currentThread() != getExclusiveOwnerThread())

        throw new IllegalMonitorStateException();

    boolean free = false;

    if (c == 0) {

        free = true;

        setExclusiveOwnerThread(null);

    }

    setState(c); // 释放锁的最后，写volatile变量state

    return free;

}

```

公平锁在释放锁的最后写volatile变量state，在获取锁时首先读这个volatile变量。

根据volatile的happens-before规则，释放锁的线程在写volatile变量之前可见的共享变量，在获取锁的线程读取同一个volatile变量后将立即变得对获取锁的线程可见。

非公平锁的释放和公平锁完全一样。

使用非公平锁时，加锁方法lock()调用轨迹如下：

1）ReentrantLock:lock()

2）NonfairSync:lock()

3）AbstractQueuedSynchronizer:compareAndSetState(int expect,int update)

在第3步真正开始加锁，下面是该方法的源代码。

```java

protected final boolean compareAndSetState(int expect, int update) {

    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}

```

该方法以原子操作的方式更新state变量，本文把Java的compareAndSet()方法调用简称为CAS。

JDK文档对该方法的说明如下：

如果当前状态值等于预期值，则以原子方式将同步状态设置为给定的更新值。此操作具有volatile读和写的内存语义。

分别从编译器和处理器的角度来分析，CAS如何同时具有volatile读和volatile写

的内存语义？

编译器不会对volatile读与volatile读后面的任意内存操作重排序；编译器不会对volatile写与volatile写前面的任意内存操作重排序。

组合这两个条件，意味着为了同时实现volatile读和volatile写的内存语义，编译器不能对CAS与CAS前面和后面的任意内存操作重排序。

在编译器c++源码中，CAS本地方法实现上，对于不同处理器有不同实现。但是都实现了内存屏障，达到了指令重排序来实现CAS方法和volatile相同的内存语义。

总结：

1. 公平锁和非公平锁释放时，最后都要写一个volatile变量state

2. 公平锁获取时，首先会去读volatile变量

3. 非公平锁获取时，首先会用CAS更新volatile变量，这个操作同时具有volatile读和volatile写的内存语义

从对ReentrantLock的分析可以看出，锁释放-获取的内存语义的实现至少有下面两种

方式：

1）利用volatile变量的写-读所具有的内存语义。

2）利用CAS所附带的volatile读和volatile写的内存语义。

### 3.5.4 concurrent包的实现

由于Java的CAS同时具有volatile读和volatile写的内存语义，因此Java线程之间的通信现在有了下面4种方式：

1）A线程写volatile变量，随后B线程读这个volatile变量

2）A线程写volatile变量，随后B线程用CAS更新这个volatile变量

3）A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量

4）A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量

Java的CAS会使用现代处理器上提供的高效机器级别的原子指令，这些原子指令以原子

方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键。同时，volatile变量的读/写和CAS可以实现线程之间的通信。把这些特性整合在一起，就形成了整个concurrent包得以实现的基石。

分析concurrent包的源代码实现，会发现一个通用化的实现模式：

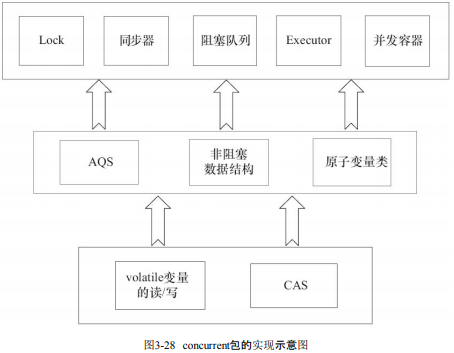
首先，声明共享变量为volatile。

然后，使用CAS的原子条件更新来实现线程之间的同步。

同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

AQS、非阻塞数据结构、原子变量类（java.util.concurrent.atomic包中的类），这些concurrent包中的基础类都是使用这种模式来实现的，而concurrent包中的高层类又是依赖于这些基础类来实现的。

从整体来看，concurrent包的实现示意图如3-28所示。



## 3.6 final域的内存语义

与锁和volatile相比，对final域的读和写更像是普通的变量访问。

### 3.6.1 final域的重排序规则

对于final域，编译器和处理器要遵守两个重排序规则：

1）在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2）初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

示例代码：

```java

public class FinalExample {

    int i; // 普通变量

    final int j; // final变量

    static FinalExample obj;

    public FinalExample() { // 构造函数

        i = 1; // 写普通域

        j = 2; // 写final域

    }

    public static void writer() { // 写线程A执行

        obj = new FinalExample();

    }

    public static void reader() { // 读线程B执行

        FinalExample object = obj; // 读对象引用

        int a = object.i; // 读普通域

        int b = object.j; // 读final域

    }

}

```

这里假设一个线程A执行writer()方法，随后另一个线程B执行reader()方法。

### 3.6.2 写final域的重排序规则

写final域的重排序规则禁止把final域的写重排序到构造函数之外。这个规则的实现包含下面2个方面：

1）JMM禁止编译器把final域的写重排序到构造函数之外

2）编译器会在final域的写之后，构造函数return之前，插入一个StoreStore屏障。这个屏障禁止处理器把final域的写重排序到构造函数之外

写final域的重排序规则可以确保：

在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域不具有这个保障。

### 3.6.3 读final域的重排序规则

读final域的重排序规则：

在一个线程中，初次读对象引用与初次读该对象包含的final域，JMM禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读final域操作的前面插入一个LoadLoad屏障。

初次读对象引用与初次读该对象包含的final域，这两个操作之间存在间接依赖关系。

由于编译器遵守间接依赖关系，因此编译器不会重排序这两个操作。

大多数处理器也会遵守间接依赖，也不会重排序这两个操作。但有少数处理器允许对存在间接依赖关系的操作做重排序（比如alpha处理器），这个规则就是专门用来针对这种处理器的。

读final域的重排序规则可以确保：

在读一个对象的final域之前，一定会先读包含这个final域的对象的引用。

### 3.6.4 final域为引用类型

示例代码：

```java

public class FinalReferenceExample {

    final int[] intArray; // final是引用类型

    static FinalReferenceExample obj;

    public FinalReferenceExample() { // 构造函数

        intArray = new int[1]; // 1

        intArray[0] = 1; // 2

    }

    public static void writerOne() { // 写线程A执行

        obj = new FinalReferenceExample (); // 3

    }

    public static void writerTwo() { // 写线程B执行

        obj.intArray[0] = 2; // 4

    }

    public static void reader() { // 读线程C执行

        if (obj != null) { // 5

            int temp1 = obj.intArray[0]; // 6

        }

    }

}

```

对于引用类型，写final域的重排序规则对编译器和处理器增加了如下约束：

在构造函数内对一个final引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

### 3.6.5 为什么final引用不能从构造函数内“溢出”

写final域的重排序规则可以确保：

在引用变量为任意线程可见之前，该引用变量指向的对象的final域已经在构造函数中被正确初始化过了。

其实，要得到这个效果，还需要一个保证：

在构造函数内部，不能让这个被构造对象的引用为其他线程所见，也就是对象引用不能在构造函数中“逸出”。

### 3.6.6 final语义在处理器中的实现

写final域的重排序规则会要求编译器在final域的写之后，构造函数return之前插入一个StoreStore障屏。

读final域的重排序规则要求编译器在读final域的操作前面插入一个LoadLoad屏障。

由于X86处理器不会对写-写操作做重排序，所以在X86处理器中，写final域需要的

StoreStore障屏会被省略掉。

同样，由于X86处理器不会对存在间接依赖关系的操作做重排序，所以在X86处理器中，读final域需要的LoadLoad屏障也会被省略掉。

也就是说，在X86处理器中，final域的读/写不会插入任何内存屏障！

### 3.6.7 JSR-133为什么要增强final的语义

在旧的Java内存模型中，一个最严重的缺陷就是线程可能看到final域的值会改变。

最常见的例子就是在旧的Java内存模型中，String的值可能会改变。

为了修补这个漏洞，JSR-133专家组增强了final的语义。

通过为final域增加写和读重排序规则，可以为Java程序员提供初始化安全保证：

只要对象是正确构造的（被构造对象的引用在构造函数中没有“逸出”），那么不需要使用同步（指lock和volatile的使用）就可以保证任意线程都能看到这个final域在构造函数中被初始化之后的值。

## 3.7 happens-before

【happens-before是JMM最核心的概念。对应Java程序员来说，理解happens-before是理解JMM的关键】

### 3.7.1 JMM的设计

从JMM设计者的角度，在设计JMM时，需要考虑两个关键因素。

程序员对内存模型的使用。

程序员希望内存模型易于理解、易于编程。程序员希望基于一个强内存模型来编写代码。

编译器和处理器对内存模型的实现。

编译器和处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。编译器和处理器希望实现一个弱内存模型。

由于这两个因素互相矛盾，所以JSR-133专家组在设计JMM时的核心目标就是找到一个好的平衡点：

一方面，要为程序员提供足够强的内存可见性保证；

另一方面，对编译器和处理器的限制要尽可能地放松。

比如：

```java

double pi = 3.14; // A

double r = 1.0; // B

double area = pi \* r \* r; // C

```

JMM把happens-before要求禁止的重排序分为了下面两类。

1. 会改变程序执行结果的重排序

2. 不会改变程序执行结果的重排序

JMM对这两种不同性质的重排序，采取了不同的策略：

1. 对于会改变程序执行结果的重排序，JMM要求编译器和处理器必须禁止这种重排序。

2. 对于不会改变程序执行结果的重排序，JMM对编译器和处理器不做要求（JMM允许这种重排序）。



JMM向程序员提供的happens-before规则能满足程序员的需求。

JMM的happens-before规则不但简单易懂，而且也向程序员提供了足够强的内存可见性保证（有些内存可见性保证其实并不一定真实存在，比如上面的A happens-before B）。

JMM对编译器和处理器的束缚已经尽可能少。

从上面的分析可以看出，JMM其实是在遵循一个基本原则：只要不改变程序的执行结果（指的是单线程程序和正确同步的多线程程序），编译器和处理器怎么优化都行。

例如，如果编译器经过细致的分析后，认定一个锁只会被单个线程访问，那么这个锁可以被消除。

再如，如果编译器经过细致的分析后，认定一个volatile变量只会被单个线程访问，那么编译器可以把这个volatile变量当作一个普通变量来对待。

这些优化既不会改变程序的执行结果，又能提高程序的执行效率。

### 3.7.2 happens-before的定义

happens-before的概念最初由Leslie Lamport在其一篇影响深远的论文（《Time，Clocks and the Ordering of Events in a Distributed System》）中提出。

JSR-133使用happens-before的概念来指定两个操作之间的执行顺序。

由于这两个操作可以在一个线程之内，也可以是在不同线程之间。

因此，JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证（如果A线程的写操作a与B线程的读操作b之间存在happensbefore关系，尽管a操作和b操作在不同的线程中执行，但JMM向程序员保证a操作将对b操作可见）。

《JSR-133:Java Memory Model and Thread Specification》对happens-before关系的定义如下：

1）如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

2）两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序并不非法（也就是说，JMM允许这种重排序）。

上面的1）是JMM对程序员的承诺。

从程序员的角度来说，可以这样理解happens-before关系：如果A happens-before B，那么Java内存模型将向程序员保证——A操作的结果将对B可见，且A的执行顺序排在B之前。注意，这只是Java内存模型向程序员做出的保证！

上面的2）是JMM对编译器和处理器重排序的约束原则。

正如前面所言，JMM其实是在遵循一个基本原则：只要不改变程序的执行结果（指的是单线程程序和正确同步的多线程程序），编译器和处理器怎么优化都行。

JMM这么做的原因是：程序员对于这两个操作是否真的被重排序并不关心，程序员关心的是程序执行时的语义不能被改变（即执行结果不能被改变）。

因此，happens-before关系本质上和as-if-serial语义是一回事。

as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。

as-if-serial语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。

happens-before关系给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按happens-before指定的顺序来执行的。

as-if-serial语义和happens-before这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

### 3.7.3 happens-before规则

《JSR-133:Java Memory Model and Thread Specification》定义了如下happens-before规则。

1）程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

2）监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

3）volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4）传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

5）start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的

ThreadB.start()操作happens-before于线程B中的任意操作。

6）join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

由于编译器和处理器都要遵守as-if-serial语义，也就是说，as-if-serial语义保证了程序顺序规则。

因此，可以把程序顺序规则看成是对as-if-serial语义的“封装”。

传递性是由volatile的内存屏障插入策略和volatile的编译器重排序规则共同来保证的。

## 3.8 双重检查锁定与延迟初始化

在Java多线程程序中，有时候需要采用延迟初始化来降低初始化类和创建对象的开销。

双重检查锁定是常见的延迟初始化技术，但它是一个错误的用法。

### 3.8.1 双重检查锁定的由来

在Java程序中，有时候可能需要推迟一些高开销的对象初始化操作，并且只有在使用这些对象时才进行初始化。此时，程序员可能会采用延迟初始化。

但要正确实现线程安全的延迟初始化需要一些技巧，否则很容易出现问题。

比如，错误的延迟加载：

```java

public class UnsafeLazyInitialization {

    private static Instance instance;

    public static Instance getInstance() {

        if (instance == null) // 1：A线程执行

            instance = new Instance(); // 2：B线程执行

        return instance;

    }

}

```

初步解决：

```java

public class SafeLazyInitialization {

    private static Instance instance;

    public synchronized static Instance getInstance() {

        if (instance == null)

            instance = new Instance();

        return instance;

    }

}

```

由于对getInstance()方法做了同步处理，synchronized将导致性能开销。

如果getInstance()方法被多个线程频繁的调用，将会导致程序执行性能的下降。

反之，如果getInstance()方法不会被多个线程频繁的调用，那么这个延迟初始化方案将能提供令人满意的性能。

在早期的JVM中，synchronized（甚至是无竞争的synchronized）存在巨大的性能开销。因此，引入“双重检查锁定（Double-Checked Locking）”。通过双重检查锁定来降低同步的开销。

```java

public class DoubleCheckedLocking { // 1

    private static Instance instance; // 2

    public static Instance getInstance() { // 3

        if (instance == null) { // 4:第一次检查

            synchronized (DoubleCheckedLocking.class) { // 5:加锁

                if (instance == null) // 6:第二次检查

                    instance = new Instance(); // 7:问题的根源出在这里

            } // 8

        } // 9

        return instance; // 10

    } // 11

}

```

如果第一次检查instance不为null，那么就不需要执行下面的加锁和初始

化操作。因此，可以大幅降低synchronized带来的性能开销。

双重检查锁定看起来似乎很完美，但这是一个错误的优化！

在线程执行到第4行，代码读取到instance不为null时，instance引用的对象有可能还没有完成初始化。

### 3.8.2 问题的根源

双重检查锁定示例代码的第7行（instance=new Singleton();）创建了一个对象。

这一行代码可以分解为如下的3行伪代码。

```java

memory = allocate(); // 1：分配对象的内存空间

ctorInstance(memory); // 2：初始化对象

instance = memory; // 3：设置instance指向刚分配的内存地址

```

上面3行伪代码中的2和3之间，可能会被重排序（在一些JIT编译器上，这种重排序是真实发生的）

2和3之间重排序之后的执行时序如下。

```java

memory = allocate(); // 1：分配对象的内存空间

instance = memory; // 3：设置instance指向刚分配的内存地址

// 注意，此时对象还没有被初始化！

ctorInstance(memory); // 2：初始化对象

```

根据《The Java Language Specification,Java SE 7 Edition》（后文简称为Java语言规范），所有线程在执行Java程序时必须要遵守intra-thread semantics。

intra-thread semantics保证重排序不会改变单线程内的程序执行结果。

换句话说，intra-thread semantics允许那些在单线程内，不会改变单线程程序执行结果的重排序。

上面3行伪代码的2和3之间虽然被重排序了，但这个重排序并不会违反intra-thread semantics。

这个重排序在没有改变单线程程序执行结果的前提下，可以提高程序的执行性能。

可以想出两个办法来实现线程安全的延迟初始化。

1）不允许2和3重排序。

2）允许2和3重排序，但不允许其他线程“看到”这个重排序。

### 3.8.3 基于volatile的解决方案

只需要做一点小的修改（把instance声明为volatile型），就可以实现线程安全的延迟初始化。

```java

public class SafeDoubleCheckedLocking {

    private volatile static Instance instance;

    public static Instance getInstance() {

        if (instance == null) {

            synchronized (SafeDoubleCheckedLocking.class) {

            if (instance == null)

                instance = new Instance(); // instance为volatile，现在没问题了

            }

        }

        return instance;

    }

}

```

这个解决方案需要JDK 5或更高版本（因为从JDK 5开始使用新的JSR-133内存模型规范，这个规范增强了volatile的语义）。

当声明对象的引用为volatile后，3.8.2节中的3行伪代码中的2和3之间的重排序，在多线程环境中将会被禁止。

这个方案本质上是通过禁止图3-39中的2和3之间的重排序，来保证线程安全的延迟初始

化。

### 3.8.4 基于类初始化的解决方案

JVM在类的初始化阶段（即在Class被加载后，且被线程使用之前），会执行类的初始化。

在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

基于这个特性，可以实现另一种线程安全的延迟初始化方案（这个方案被称之为Initialization On Demand Holder idiom）。

```java

public class InstanceFactory {

    private static class InstanceHolder {

        public static Instance instance = new Instance();

    }

    public static Instance getInstance() {

        return InstanceHolder.instance ; // 这里将导致InstanceHolder类被初始化

    }

}

```

初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。

根据Java语言规范，在首次发生下列任意一种情况时，一个类或接口类型T将被立即初始化。

1）T是一个类，而且一个T类型的实例被创建。

2）T是一个类，且T中声明的一个静态方法被调用。

3）T中声明的一个静态字段被赋值。

4）T中声明的一个静态字段被使用，而且这个字段不是一个常量字段。

5）T是一个顶级类（Top Level Class，见Java语言规范的§7.6），而且一个断言语句嵌套在T内部被执行。

由于Java语言是多线程的，多个线程可能在同一时间尝试去初始化同一个类或接口（比如这里多个线程可能在同一时刻调用getInstance()方法来初始化InstanceHolder类）。

因此，在Java中初始化一个类或者接口时，需要做细致的同步处理。

Java语言规范规定，对于每一个类或接口C，都有一个唯一的初始化锁LC与之对应。

从C到LC的映射，由JVM的具体实现去自由实现。

JVM在类初始化期间会获取这个初始化锁，并且每个线程至少获取一次锁来确保这个类已经被初始化过了（事实上，Java语言规范允许JVM的具体实现在这里做一些优化，见后文的说明）。

对于类或接口的初始化，Java语言规范制定了精巧而复杂的类初始化处理过程。

大体上可以分为：

第1阶段：通过在Class对象上同步（即获取Class对象的初始化锁），来控制类或接口的初始化。

这个获取锁的线程会一直等待，直到当前线程能够获取到这个初始化锁。

假设Class对象当前还没有被初始化（初始化状态state，此时被标记为state=noInitialization），且有两个线程A和B试图同时初始化这个Class对象。

第2阶段：线程A执行类的初始化，同时线程B在初始化锁对应的condition上等待。

第3阶段：线程A设置state=initialized，然后唤醒在condition中等待的所有线程。

第4阶段：线程B结束类的初始化处理。

第5阶段：线程C执行类的初始化的处理。

这里的condition和state标记是虚构出来的。Java语言规范并没有硬性规定一定要使用condition和state标记。JVM的具体实现只要实现类似功能即可。

通过对比基于volatile的双重检查锁定的方案和基于类初始化的方案，我们会发现基于类初始化的方案的实现代码更简洁。

但基于volatile的双重检查锁定的方案有一个额外的优势：

除了可以对静态字段实现延迟初始化外，还可以对实例字段实现延迟初始化。

字段延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化的字段

的开销。

在大多数时候，正常的初始化要优于延迟初始化。

如果确实需要对实例字段使用线程安全的延迟初始化，请使用上面介绍的基于volatile的延迟初始化的方案；

如果确实需要对静态字段使用线程安全的延迟初始化，请使用上面介绍的基于类初始化的方案。

## 3.9 Java内存模型综述

### 3.9.1 处理器的内存模型

顺序一致性内存模型是一个理论参考模型，JMM和处理器内存模型在设计时通常会以顺序一致性内存模型为参照。

在设计时，JMM和处理器内存模型会对顺序一致性模型做一些放松，因为如果完全按照顺序一致性模型来实现处理器和JMM，那么很多的处理器和编译器优化都要被禁止，这对执行性能将会有很大的影响。

处理器要遵守as-if-serial语义，处理器不会对存在数据依赖性的两个内存操作做重排序。

越是追求性能的处理器，内存模型设计得会越弱。

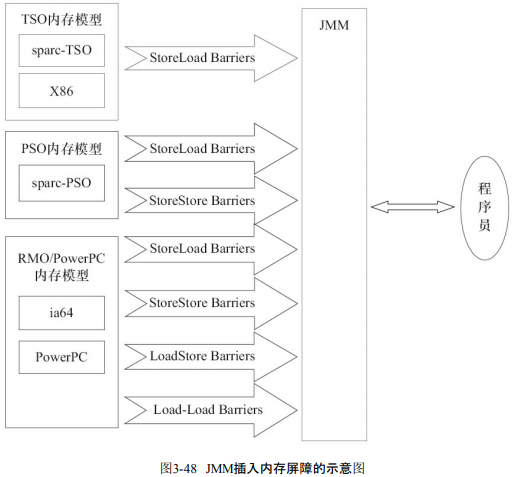
因为这些处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。

由于常见的处理器内存模型比JMM要弱，Java编译器在生成字节码时，会在执行指令序

列的适当位置插入内存屏障来限制处理器的重排序。

同时，由于各种处理器内存模型的强弱不同，为了在不同的处理器平台向程序员展示一个一致的内存模型，JMM在不同的处理器中需要插入的内存屏障的数量和种类也不相同。

JMM屏蔽了不同处理器内存模型的差异，它在不同的处理器平台之上为Java程序员呈现了一个一致的内存模型。



### 3.9.2 各种内存模型之间的关系

JMM是一个语言级的内存模型，处理器内存模型是硬件级的内存模型，顺序一致性内存

模型是一个理论参考模型。

常见的4种处理器内存模型比常用的3种语言内存模型要弱，处理器内存模型和语言内存模型都比顺序一致性内存模型要弱。

同处理器内存模型一样，越是追求执行性能的语言，内存模型设计得会越弱。

### 3.9.3 JMM的内存可见性保证

按程序类型，Java程序的内存可见性保证可以分为下列3类。

1. 单线程程序。

单线程程序不会出现内存可见性问题。

编译器、runtime和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。

2. 正确同步的多线程程序。

正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）。

这是JMM关注的重点，JMM通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。

3. 未同步/未正确同步的多线程程序。

JMM为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0、null、false）。

只要多线程程序是正确同步的，JMM保证该程序在任意的处理器平台上的执行结果，与

该程序在顺序一致性内存模型中的执行结果一致。

### 3.9.4 JSR-133对旧内存模型的修补

JSR-133对JDK 5之前的旧内存模型的修补主要有两个。

1. 增强volatile的内存语义。

旧内存模型允许volatile变量与普通变量重排序。JSR-133严格限制volatile变量与普通变量的重排序，使volatile的写-读和锁的释放-获取具有相同的内存语义。

2. 增强final的内存语义。

在旧内存模型中，多次读取同一个final变量的值可能会不相同。

为此，JSR-133为final增加了两个重排序规则。在保证final引用不会从构造函数内逸出的情况下，final具有了初始化安全性。

## 3.10 小结

Java内存模型

处理器内存模型

顺序一致性内存模型

# 4. 并发基础

线程作为操作系统调度的最小单元，多个线程能够同时执行，这将显著提升程序性能，在多核环境中表现得更加明显。

但是，过多地创建线程和对线程的不当管理也容易造成问题。

## 4.1 线程简介

### 4.1.1 什么是线程

现代操作系统在运行一个程序时，会为其创建一个进程。

现代操作系统调度的最小单元是线程，也叫轻量级进程（LightWeight Process），在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。

```java

import java.lang.management.ManagementFactory;

import java.lang.management.ThreadInfo;

import java.lang.management.ThreadMXBean;

public class MultiThread{

    public static void main(String[] args) {

        // 获取Java线程管理MXBean

        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();

        // 不需要获取同步的monitor和synchronizer信息，仅获取线程和线程堆栈信息

        ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false,false);

        // 遍历线程信息，仅打印线程ID和线程名称信息

        for (ThreadInfo threadInfo : threadInfos) {

            System.out.println("[" + threadInfo.getThreadId() + "] " + threadInfo.getThreadName());

        }

    }

}

```

输出如下所示（输出内容可能不同）:

```

[6] Monitor Ctrl-Break

[5] Attach Listener

[4] Signal Dispatcher

[3] Finalizer

[2] Reference Handler

[1] main

```

### 4.1.2 为什么要使用多线程

（1）更多的处理器核心

一个线程在一个时刻只能运行在一个处理器核心上，如何发挥多个核心的性能称为主要问题。

使用多线程技术，将计算逻辑分配到多个处理器核心上，就会显著减少程序的处理时间，并且随着更多处理器核心的加入而变得更有效率。

（2）更快的响应时间

异步编程，更好的性能。

（3）更好的编程模型

Java为多线程编程提供了良好、考究并且一致的编程模型，使开发人员能够更加专注于问题的解决，即为所遇到的问题建立合适的模型，而不是绞尽脑汁地考虑如何将其多线程化。

一旦开发人员建立好了模型，稍做修改总是能够方便地映射到Java提供的多线程编程模型上。

### 4.1.3 线程优先级

现代操作系统基本采用时分的形式调度运行的线程，操作系统会分出一个个时间片，线程会分配到若干时间片，当线程的时间片用完了就会发生线程调度，并等待着下次分配。

线程分配到的时间片多少也就决定了线程使用处理器资源的多少，而线程优先级就是决定线程需要多或者少分配一些处理器资源的线程属性。

在Java线程中，通过一个整型成员变量priority来控制优先级，优先级的范围从1~10，在线程构建的时候可以通过setPriority(int)方法来修改优先级，默认优先级是5。

优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者I/O操作）的线程需要设置较高优先级，而偏重计算（需要较多CPU时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。

在不同的JVM以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

```java

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.TimeUnit;

public class Priority {

    private static volatile boolean notStart = true;

    private static volatile boolean notEnd = true;

    public static void main(String[] args) throws Exception {

        List<Job> jobs = new ArrayList<Job>();

        for (int i = 0; i < 10; i++) {

            int priority = i < 5 ? Thread.MIN\_PRIORITY : Thread.MAX\_PRIORITY;

            Job job = new Job(priority);

            jobs.add(job);

            Thread thread = new Thread(job, "Thread:" + i);

            thread.setPriority(priority);

            thread.start();

        }

        notStart = false;

        TimeUnit.SECONDS.sleep(10);

        notEnd = false;

        for (Job job : jobs) {

            System.out.println("Job Priority : " + job.priority + ", Count : " + job.jobCount);

        }

    }

    static class Job implements Runnable {

        private int priority;

        private long jobCount;

        public Job(int priority) {

            this.priority = priority;

        }

        public void run() {

            while (notStart) {

                Thread.yield();

            }

            while (notEnd) {

                Thread.yield();

                jobCount++;

            }

        }

    }

}

```

结果：

```

Job Priority : 1, Count : 131641

Job Priority : 1, Count : 131757

Job Priority : 1, Count : 131687

Job Priority : 1, Count : 131741

Job Priority : 1, Count : 131736

Job Priority : 10, Count : 2987130

Job Priority : 10, Count : 2977545

Job Priority : 10, Count : 2987378

Job Priority : 10, Count : 2975449

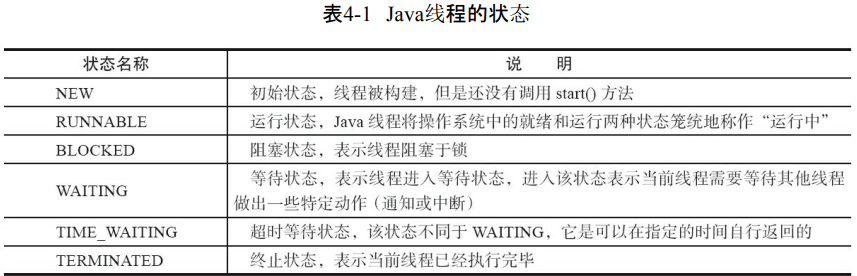
Job Priority : 10, Count : 2968712

```

注意：

线程优先级不能作为程序正确性的依赖，因为操作系统可以完全不用理会Java线程对于优先级的设定。

### 4.1.4 线程的状态



用jstack工具查看堆栈中的线程状态信息。

```java

import java.util.concurrent.TimeUnit;

public class ThreadState {

    public static void main(String[] args) {

        new Thread(new TimeWaiting(), "TimeWaitingThread").start();

        new Thread(new Waiting(), "WaitingThread").start();

        // 使用两个Blocked线程，一个获取锁成功，另一个被阻塞

        new Thread(new Blocked(), "BlockedThread-1").start();

        new Thread(new Blocked(), "BlockedThread-2").start();

    }

    // 该线程不断地进行睡眠

    static class TimeWaiting implements Runnable {

        @Override

        public void run() {

            while (true) {

                SleepUtils.second(100);

            }

        }

    }

    // 该线程在Waiting.class实例上等待

    static class Waiting implements Runnable {

        @Override

        public void run() {

            while (true) {

                synchronized (Waiting.class) {

                    try {

                        Waiting.class.wait();

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    }

                }

            }

        }

    }

    // 该线程在Blocked.class实例上加锁后，不会释放该锁

    static class Blocked implements Runnable {

        public void run() {

            synchronized (Blocked.class) {

                while (true) {

                    SleepUtils.second(100);

                }

            }

        }

    }

    static class SleepUtils {

        public static final void second(long seconds) {

            try {

                TimeUnit.SECONDS.sleep(seconds);

            } catch (InterruptedException e) {

            }

        }

    }

}

```

```

E:\githubRepositorys\xyw21\java\worksp>jps

14192 ThreadState

12596 Jps

7876 Launcher

13900

13916 RemoteMavenServer36

E:\githubRepositorys\xyw21\java\worksp>jstack 14192

2021-08-16 21:59:04

Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.65-b01 mixed mode):

"DestroyJavaVM" #15 prio=5 os\_prio=0 tid=0x00000000031d2800 nid=0x2f58 waiting on condition [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

// BlockedThread-2线程阻塞在获取Blocked.class示例的锁上

"BlockedThread-2" #14 prio=5 os\_prio=0 tid=0x00000000194ff800 nid=0x1ca8 waiting for monitor entry [0x000000001a18f000]

   java.lang.Thread.State: BLOCKED (on object monitor)

        at com.wh.p4.ThreadState$Blocked.run(ThreadState.java:42)

        - waiting to lock <0x00000000d5fc7868> (a java.lang.Class for com.wh.p4.ThreadState$Blocked)

        at java.lang.Thread.run(Thread.java:745)

// BlockedThread-1线程获取到了Blocked.class的锁

"BlockedThread-1" #13 prio=5 os\_prio=0 tid=0x00000000194ff000 nid=0x2b84 waiting on condition [0x000000001a08e000]

   java.lang.Thread.State: TIMED\_WAITING (sleeping)

        at java.lang.Thread.sleep(Native Method)

        at java.lang.Thread.sleep(Thread.java:340)

        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

        at com.wh.p4.ThreadState$SleepUtils.second(ThreadState.java:51)

        at com.wh.p4.ThreadState$Blocked.run(ThreadState.java:42)

        - locked <0x00000000d5fc7868> (a java.lang.Class for com.wh.p4.ThreadState$Blocked)

        at java.lang.Thread.run(Thread.java:745)

// WaitingThread线程在Waiting实例上等待

"WaitingThread" #12 prio=5 os\_prio=0 tid=0x00000000194fc000 nid=0x2888 in Object.wait() [0x0000000019f8e000]

   java.lang.Thread.State: WAITING (on object monitor)

        at java.lang.Object.wait(Native Method)

        - waiting on <0x00000000d5fc4d10> (a java.lang.Class for com.wh.p4.ThreadState$Waiting)

        at java.lang.Object.wait(Object.java:502)

        at com.wh.p4.ThreadState$Waiting.run(ThreadState.java:29)

        - locked <0x00000000d5fc4d10> (a java.lang.Class for com.wh.p4.ThreadState$Waiting)

        at java.lang.Thread.run(Thread.java:745)

// TimeWaitingThread线程处于超时等待

"TimeWaitingThread" #11 prio=5 os\_prio=0 tid=0x00000000194f9000 nid=0x6c waiting on condition [0x0000000019e8f000]

   java.lang.Thread.State: TIMED\_WAITING (sleeping)

        at java.lang.Thread.sleep(Native Method)

        at java.lang.Thread.sleep(Thread.java:340)

        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

        at com.wh.p4.ThreadState$SleepUtils.second(ThreadState.java:51)

        at com.wh.p4.ThreadState$TimeWaiting.run(ThreadState.java:18)

        at java.lang.Thread.run(Thread.java:745)

"Service Thread" #10 daemon prio=9 os\_prio=0 tid=0x00000000194b6000 nid=0x1610 runnable [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #9 daemon prio=9 os\_prio=2 tid=0x000000001943f800 nid=0x7e4 waiting on condition [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os\_prio=2 tid=0x0000000019413800 nid=0x362c waiting on condition [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os\_prio=2 tid=0x000000001940e000 nid=0x524 waiting on condition [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"Monitor Ctrl-Break" #6 daemon prio=5 os\_prio=0 tid=0x00000000193d7000 nid=0x1b74 runnable [0x000000001988e000]

   java.lang.Thread.State: RUNNABLE

        at java.net.SocketInputStream.socketRead0(Native Method)

        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)

        at java.net.SocketInputStream.read(SocketInputStream.java:170)

        at java.net.SocketInputStream.read(SocketInputStream.java:141)

        at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)

        at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)

        at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)

        - locked <0x00000000d5f33468> (a java.io.InputStreamReader)

        at java.io.InputStreamReader.read(InputStreamReader.java:184)

        at java.io.BufferedReader.fill(BufferedReader.java:161)

        at java.io.BufferedReader.readLine(BufferedReader.java:324)

        - locked <0x00000000d5f33468> (a java.io.InputStreamReader)

        at java.io.BufferedReader.readLine(BufferedReader.java:389)

        at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)

"Attach Listener" #5 daemon prio=5 os\_prio=2 tid=0x0000000017e41800 nid=0x291c waiting on condition [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os\_prio=2 tid=0x0000000017e40800 nid=0x286c runnable [0x0000000000000000]

   java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os\_prio=1 tid=0x00000000032c7800 nid=0x580 in Object.wait() [0x000000001918e000]

   java.lang.Thread.State: WAITING (on object monitor)

        at java.lang.Object.wait(Native Method)

        - waiting on <0x00000000d5d070b8> (a java.lang.ref.ReferenceQueue$Lock)

        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)

        - locked <0x00000000d5d070b8> (a java.lang.ref.ReferenceQueue$Lock)

        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)

        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

"Reference Handler" #2 daemon prio=10 os\_prio=2 tid=0x0000000017df9000 nid=0xea8 in Object.wait() [0x000000001908f000]

   java.lang.Thread.State: WAITING (on object monitor)

        at java.lang.Object.wait(Native Method)

        - waiting on <0x00000000d5d06af8> (a java.lang.ref.Reference$Lock)

        at java.lang.Object.wait(Object.java:502)

        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:157)

        - locked <0x00000000d5d06af8> (a java.lang.ref.Reference$Lock)

"VM Thread" os\_prio=2 tid=0x0000000017df7800 nid=0x1880 runnable

"GC task thread#0 (ParallelGC)" os\_prio=0 tid=0x00000000031e8000 nid=0x30ac runnable

"GC task thread#1 (ParallelGC)" os\_prio=0 tid=0x00000000031e9800 nid=0x13fc runnable

"GC task thread#2 (ParallelGC)" os\_prio=0 tid=0x00000000031ec800 nid=0x2c0c runnable

"GC task thread#3 (ParallelGC)" os\_prio=0 tid=0x00000000031ee000 nid=0x2f9c runnable

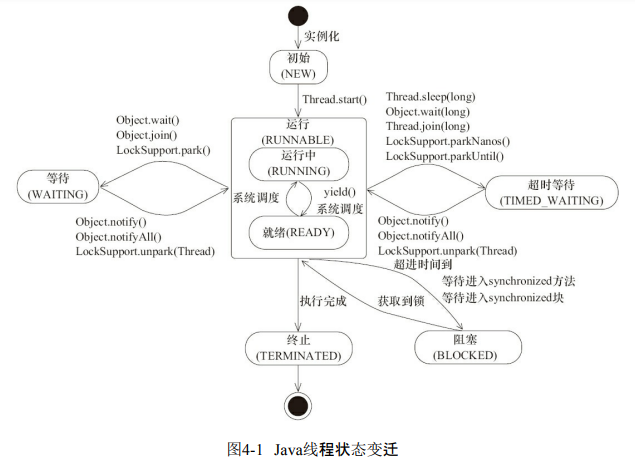
"VM Periodic Task Thread" os\_prio=2 tid=0x00000000194f3000 nid=0x3614 waiting on condition

JNI global references: 33

```

线程在自身的生命周期中，

并不是固定地处于某个状态，而是随着代码的执行在不同的状态之间进行切换：



注意：

Java将操作系统中的运行和就绪两个状态合并称为运行状态。

阻塞状态是线程阻塞在进入synchronized关键字修饰的方法或代码块（获取锁）时的状态。

但是阻塞在java.concurrent包中Lock接口的线程状态却是等待状态，因为java.concurrent包中Lock接口对于阻塞的实现均使用了LockSupport类中的相关方法。

### 4.1.5 Daemon线程

Daemon线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。

这意味着，当一个Java虚拟机中不存在非Daemon线程的时候，Java虚拟机将会退出。

可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程。

注意：

Daemon属性需要在启动线程之前设置，不能在启动线程之后设置。

Daemon线程被用作完成支持性工作，但是在Java虚拟机退出时Daemon线程中的finally块并不一定会执行。

```java

import java.util.concurrent.TimeUnit;

public class Daemon {

    public static void main(String[] args) {

        Thread thread = new Thread(new DaemonRunner(), "DaemonRunner");

        thread.setDaemon(true);

        thread.start();

    }

    static class DaemonRunner implements Runnable {

        @Override

        public void run() {

            try {

                TimeUnit.SECONDS.sleep(10);

            } catch (InterruptedException e) {

                e.printStackTrace();

            } finally {

                System.out.println("DaemonThread finally run.");

            }

        }

    }

}

```

运行Daemon程序，可以看到在终端或者命令提示符上没有任何输出。

main线程（非Daemon线程）在启动了线程DaemonRunner之后随着main方法执行完毕而终止，而此时Java虚拟机中已经没有非Daemon线程，虚拟机需要退出。

Java虚拟机中的所有Daemon线程都需要立即终止，因此DaemonRunner立即终止，但是DaemonRunner中的finally块并没有执行。

注意：

在构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

## 4.2 启动和终止线程

通过调用线程的start()方法进行启动，随着run()方法的执行完毕，线程也随之终止。

### 4.2.1 构造线程

在运行线程之前首先要构造一个线程对象，线程对象在构造的时候需要提供线程所需要的属性，如线程所属的线程组、线程优先级、是否是Daemon线程等信息。

```java

package java.lang;

public

class Thread implements Runnable {

    private void init(ThreadGroup g, Runnable target, String name,long stackSize,

        AccessControlContext acc) {

        if (name == null) {

            throw new NullPointerException("name cannot be null");

        }

        // 当前线程就是该线程的父线程

        Thread parent = currentThread();

        this.group = g;

        // 将daemon、priority属性设置为父线程的对应属性

        this.daemon = parent.isDaemon();

        this.priority = parent.getPriority();

        this.name = name.toCharArray();

        this.target = target;

        setPriority(priority);

        // 将父线程的InheritableThreadLocal复制过来

        if (parent.inheritableThreadLocals != null)

            this.inheritableThreadLocals=ThreadLocal.createInheritedMap(parent.

        inheritableThreadLocals);

        // 分配一个线程ID

        tid = nextThreadID();

    }

}

```

在上述过程中，一个新构造的线程对象是由其parent线程来进行空间分配的。

而child线程继承了parent是否为Daemon、优先级和加载资源的contextClassLoader以及可继承的ThreadLocal，同时还会分配一个唯一的ID来标识这个child线程。

至此，一个能够运行的线程对象就初始化好了，在堆内存中等待着运行。

### 4.2.2 启动线程

线程对象在初始化完成之后，调用start()方法就可以启动这个线程。

线程start()方法的含义是：

当前线程（即parent线程）同步告知Java虚拟机，只要线程规划器空闲，应立即启动调用start()方法的线程。

注意：

启动一个线程前，最好为这个线程设置线程名称，因为这样在使用jstack分析程序或者进行问题排查时，就会给开发人员提供一些提示，自定义的线程最好能够起个名字。

### 4.2.3 理解中断

中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。

中断好比其他线程对该线程打了个招呼，其他线程通过调用该线程的interrupt()方法对其进行中断操作。

线程通过检查自身是否被中断来进行响应，线程通过方法isInterrupted()来进行判断是否被中断，也可以调用静态方法Thread.interrupted()对当前线程的中断标识位进行复位。

如果该线程已经处于终结状态，即使该线程被中断过，在调用该线程对象的isInterrupted()时依旧会返回false。

从Java的API中可以看到，许多声明抛出InterruptedException的方法（例如Thread.sleep(long millis)）这些方法在抛出InterruptedException之前，Java虚拟机会先将该线程的中断标识位清除，然后抛出InterruptedException，此时调用isInterrupted()方法将会返回false。

```java

import java.util.concurrent.TimeUnit;

public class Interrupted {

    public static void main(String[] args) throws Exception {

        // sleepThread不停的尝试睡眠

        Thread sleepThread = new Thread(new SleepRunner(), "SleepThread");

        sleepThread.setDaemon(true);

        // busyThread不停的运行

        Thread busyThread = new Thread(new BusyRunner(), "BusyThread");

        busyThread.setDaemon(true);

        sleepThread.start();

        busyThread.start();

        // 休眠5秒，让sleepThread和busyThread充分运行

        TimeUnit.SECONDS.sleep(5);

        sleepThread.interrupt();

        busyThread.interrupt();

        System.out.println("SleepThread interrupted is " + sleepThread.isInterrupted());

        System.out.println("BusyThread interrupted is " + busyThread.isInterrupted());

        // 防止sleepThread和busyThread立刻退出

        TimeUnit.SECONDS.sleep(2);

    }

    static class SleepRunner implements Runnable {

        @Override

        public void run() {

            while (true) {

                try {

                    TimeUnit.SECONDS.sleep(10);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

    static class BusyRunner implements Runnable {

        @Override

        public void run() {

            while (true) {

            }

        }

    }

}

```

结果：

```

SleepThread interrupted is false

BusyThread interrupted is true

java.lang.InterruptedException: sleep interrupted

    at java.lang.Thread.sleep(Native Method)

    at java.lang.Thread.sleep(Thread.java:340)

    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

    at com.wh.p4.Interrupted$SleepRunner.run(Interrupted.java:29)

    at java.lang.Thread.run(Thread.java:745)

```

从结果可以看出，抛出InterruptedException的线程SleepThread，其中断标识位被清除了，而一直忙碌运作的线程BusyThread，中断标识位没有被清除。

### 4.2.4 过期的suspend()、resume()和stop()

对音乐播放做出的暂停、恢复和停止操作对应在线程Thread的API就是suspend()、resume()和stop()。

```java

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.TimeUnit;

public class Deprecated {

    public static void main(String[] args) throws Exception {

        DateFormat format = new SimpleDateFormat("HH:mm:ss");

        Thread printThread = new Thread(new Runner(), "PrintThread");

        printThread.setDaemon(true);

        printThread.start();

        TimeUnit.SECONDS.sleep(3);

        // 将PrintThread进行暂停，输出内容工作停止

        printThread.suspend();

        System.out.println("main suspend PrintThread at " + format.format(new Date()));

        TimeUnit.SECONDS.sleep(3);

        // 将PrintThread进行恢复，输出内容继续

        printThread.resume();

        System.out.println("main resume PrintThread at " + format.format(new Date()));

        TimeUnit.SECONDS.sleep(3);

        // 将PrintThread进行终止，输出内容停止

        printThread.stop();

        System.out.println("main stop PrintThread at " + format.format(new Date()));

        TimeUnit.SECONDS.sleep(3);

    }

    static class Runner implements Runnable {

        @Override

        public void run() {

            DateFormat format = new SimpleDateFormat("HH:mm:ss");

            while (true) {

                System.out.println(Thread.currentThread().getName() + " Run at " +

                        format.format(new Date()));

                ThreadState.SleepUtils.second(1);

            }

        }

    }

}

```

这些API是过期的，不建议使用。

不建议使用的原因主要有：

以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。

同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。

注意：

正因为suspend()、resume()和stop()方法带来的副作用，这些方法才被标注为不建议使用的过期方法，而暂停和恢复操作可以用后面提到的等待/通知机制来替代。

### 4.2.5 安全地终止线程

中断状态是线程的一个标识位，而中断操作是一种简便的线程间交互方式，而这种交互方式最适合用来取消或停止任务。

除了中断以外，还可以利用一个boolean变量来控制是否需要停止任务并终止该线程。

```java

import java.util.concurrent.TimeUnit;

public class Shutdown {

    public static void main(String[] args) throws Exception {

        Runner one = new Runner();

        Thread countThread = new Thread(one, "CountThread");

        countThread.start();

        // 睡眠1秒，main线程对CountThread进行中断，使CountThread能够感知中断而结束

        TimeUnit.SECONDS.sleep(1);

        countThread.interrupt();

        Runner two = new Runner();

        countThread = new Thread(two, "CountThread");

        countThread.start();

        // 睡眠1秒，main线程对Runner two进行取消，使CountThread能够感知on为false而结束

        TimeUnit.SECONDS.sleep(1);

        two.cancel();

    }

    private static class Runner implements Runnable {

        private long i;

        private volatile boolean on = true;

        @Override

        public void run() {

            while (on && !Thread.currentThread().isInterrupted()){

                i++;

            }

            System.out.println("Count i = " + i);

        }

        public void cancel() {

            on = false;

        }

    }

}

```

结果：

```

Count i = 200290598

Count i = 215280442

```

在执行过程中，main线程通过中断操作和cancel()方法均可使CountThread得以终止。

这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

## 4.3 线程间通信

### 4.3.1 volatile和synchronized关键字

Java支持多个线程同时访问一个对象或者对象的成员变量，由于每个线程可以拥有这个变量的拷贝（虽然对象以及成员变量分配的内存是在共享内存中的，但是每个执行的线程还是可以拥有一份拷贝，这样做的目的是加速程序的执行，这是现代多核处理器的一个显著特性），所以程序在执行过程中，一个线程看到的变量并不一定是最新的。

关键字volatile可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

但是，过多地使用volatile是不必要的，因为它会降低程序执行的效率。(volatile的实现原理，处理器的内存屏障和编译器的指令禁止重排序规则保证，带来了额外的性能损耗)

关键字synchronized可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。

```java

public class Synchronized {

    public static void main(String[] args) {

        // 对Synchronized Class对象进行加锁

        synchronized (Synchronized.class) {

        }

        // 静态同步方法，对Synchronized Class对象进行加锁

        m();

    }

    public static synchronized void m() {}

}

```

使用javap工具查看生成的class文件信息来分析synchronized关键字的实现细节：

```

E:\githubRepositorys\xyw21\java\worksp\target\classes\com\wh\p4>javap -v Synchronized.class

Classfile /E:/githubRepositorys/xyw21/java/worksp/target/classes/com/wh/p4/Synchronized.class

  Last modified 2021-8-16; size 572 bytes

  MD5 checksum 44a26e0536cc4047235e207d99ac1773

  Compiled from "Synchronized.java"

public class com.wh.p4.Synchronized

  minor version: 0

  major version: 52

  flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

   #1 = Methodref          #4.#23         // java/lang/Object."<init>":()V

   #2 = Class              #24            // com/wh/p4/Synchronized

   #3 = Methodref          #2.#25         // com/wh/p4/Synchronized.m:()V

   #4 = Class              #26            // java/lang/Object

   #5 = Utf8               <init>

   #6 = Utf8               ()V

   #7 = Utf8               Code

   #8 = Utf8               LineNumberTable

   #9 = Utf8               LocalVariableTable

  #10 = Utf8               this

  #11 = Utf8               Lcom/wh/p4/Synchronized;

  #12 = Utf8               main

  #13 = Utf8               ([Ljava/lang/String;)V

  #14 = Utf8               args

  #15 = Utf8               [Ljava/lang/String;

  #16 = Utf8               StackMapTable

  #17 = Class              #15            // "[Ljava/lang/String;"

  #18 = Class              #26            // java/lang/Object

  #19 = Class              #27            // java/lang/Throwable

  #20 = Utf8               m

  #21 = Utf8               SourceFile

  #22 = Utf8               Synchronized.java

  #23 = NameAndType        #5:#6          // "<init>":()V

  #24 = Utf8               com/wh/p4/Synchronized

  #25 = NameAndType        #20:#6         // m:()V

  #26 = Utf8               java/lang/Object

  #27 = Utf8               java/lang/Throwable

{

  public com.wh.p4.Synchronized();

    descriptor: ()V

    flags: ACC\_PUBLIC

    Code:

      stack=1, locals=1, args\_size=1

         0: aload\_0

         1: invokespecial #1                  // Method java/lang/Object."<init>":()V

         4: return

      LineNumberTable:

        line 3: 0

      LocalVariableTable:

        Start  Length  Slot  Name   Signature

            0       5     0  this   Lcom/wh/p4/Synchronized;

  public static void main(java.lang.String[]);

    descriptor: ([Ljava/lang/String;)V

    // 方法修饰符，表示：public staticflags: ACC\_PUBLIC, ACC\_STATIC

    flags: ACC\_PUBLIC, ACC\_STATIC

    Code:

      stack=2, locals=3, args\_size=1

         0: ldc           #2                  // class com/wh/p4/Synchronized

         2: dup

         3: astore\_1

         4: monitorenter    // monitorenter：监视器进入，获取锁

         5: aload\_1

         6: monitorexit     // monitorexit：监视器退出，释放锁

         7: goto          15

        10: astore\_2

        11: aload\_1

        12: monitorexit

        13: aload\_2

        14: athrow

        15: invokestatic  #3                  // Method m:()V

        18: return

      Exception table:

         from    to  target type

             5     7    10   any

            10    13    10   any

      LineNumberTable:

        line 6: 0

        line 7: 5

        line 9: 15

        line 10: 18

      LocalVariableTable:

        Start  Length  Slot  Name   Signature

            0      19     0  args   [Ljava/lang/String;

      StackMapTable: number\_of\_entries = 2

        frame\_type = 255 /\* full\_frame \*/

          offset\_delta = 10

          locals = [ class "[Ljava/lang/String;", class java/lang/Object ]

          stack = [ class java/lang/Throwable ]

        frame\_type = 250 /\* chop \*/

          offset\_delta = 4

  public static synchronized void m();

    descriptor: ()V

    flags: ACC\_PUBLIC, ACC\_STATIC, ACC\_SYNCHRONIZED

    Code:

      stack=0, locals=0, args\_size=0

         0: return

      LineNumberTable:

        line 11: 0

}

SourceFile: "Synchronized.java"

```

上面class信息中，对于同步块的实现使用了monitorenter和monitorexit指令，而同步方法则是依靠方法修饰符上的ACC\_SYNCHRONIZED来完成的。

无论采用哪种方式，其本质是对一个对象的监视器（monitor）进行获取，而这个获取过程是排他的，也就是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。

任意一个对象都拥有自己的监视器，当这个对象由同步块或者这个对象的同步方法调用时，执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法，而没有获取到监视器（执行该方法）的线程将会被阻塞在同步块和同步方法的入口处，进入BLOCKED状态。

对象、对象的监视器、同步队列和执行线程之间的关系：



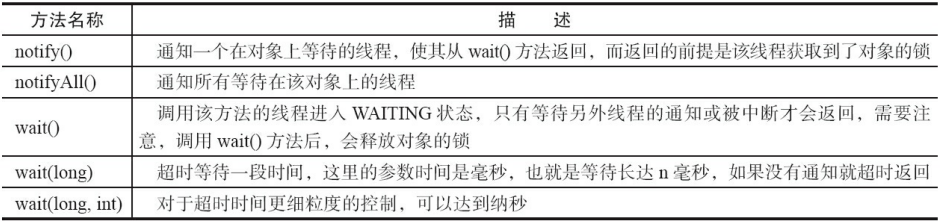
任意线程对Object（Object由synchronized保护）的访问，首先要获得Object的监视器。

如果获取失败，线程进入同步队列，线程状态变为BLOCKED。

当访问Object的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

### 4.3.2 等待/通知机制

等待/通知的相关方法是任意Java对象都具备的，因为这些方法被定义在所有对象的超类java.lang.Object上。



等待/通知机制，是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。

上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

```java

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.TimeUnit;

public class WaitNotify {

    static boolean flag = true;

    static Object lock = new Object();

    public static void main(String[] args) throws Exception {

        Thread waitThread = new Thread(new Wait(), "WaitThread");

        waitThread.start();

        TimeUnit.SECONDS.sleep(1);

        Thread notifyThread = new Thread(new Notify(), "NotifyThread");

        notifyThread.start();

    }

    static class Wait implements Runnable {

        public void run() {

            // 加锁，拥有lock的Monitor

            synchronized (lock) {

                // 当条件不满足时，继续wait，同时释放了lock的锁

                while (flag) {

                    try {

                        System.out.println(Thread.currentThread() + " flag is true. wait @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));

                        lock.wait();

                    } catch (InterruptedException e) {

                    }

                }

                // 条件满足时，完成工作

                System.out.println(Thread.currentThread() + " flag is false. running @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));

            }

        }

    }

    static class Notify implements Runnable {

        public void run() {

            // 加锁，拥有lock的Monitor

            synchronized (lock) {

                // 获取lock的锁，然后进行通知，通知时不会释放lock的锁，

                // 直到当前线程释放了lock后，WaitThread才能从wait方法中返回

                System.out.println(Thread.currentThread() + " hold lock. notify @ " +

                        new SimpleDateFormat("HH:mm:ss").format(new Date()));

                lock.notifyAll();

                flag = false;

                ThreadState.SleepUtils.second(5);

            }

            // 再次加锁

            synchronized (lock) {

                System.out.println(Thread.currentThread() + " hold lock again. sleep @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));

                ThreadState.SleepUtils.second(5);

            }

        }

    }

}

```

输出：

```

Thread[WaitThread,5,main] flag is true. wait @ 22:49:13

Thread[NotifyThread,5,main] hold lock. notify @ 22:49:14

Thread[NotifyThread,5,main] hold lock again. sleep @ 22:49:19

Thread[WaitThread,5,main] flag is false. running @ 22:49:24

```

上述第3行和第4行输出的顺序可能会互换（53行释放锁之后，由于flag状态已变化，55这里有锁竞争）

调用wait()、notify()以及notifyAll()时需要注意的细节：

1）使用wait()、notify()和notifyAll()时需要先对调用对象加锁。

2）调用wait()方法后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列。

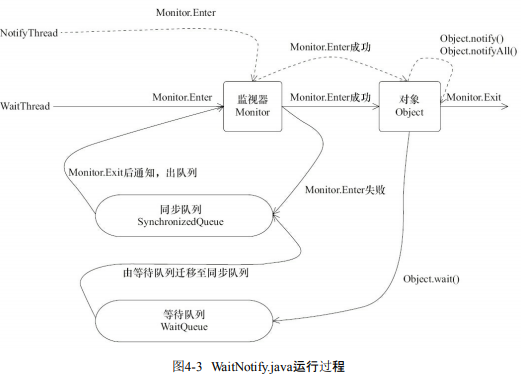
3）notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifAll()的线程释放锁之后，等待线程才有机会从wait()返回。

4）notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll()方法则是将等待队列中所有的线程全部移到同步队列，被移动的线程状态由WAITING变为BLOCKED。

5）从wait()方法返回的前提是获得了调用对象的锁。

从上述细节中可以看到，等待/通知机制依托于同步机制，其目的就是确保等待线程从wait()方法返回时能够感知到通知线程对变量做出的修改。

上述示例的过程：



在图4-3中，WaitThread首先获取了对象的锁，然后调用对象的wait()方法，从而放弃了锁并进入了对象的等待队列WaitQueue中，进入等待状态。

由于WaitThread释放了对象的锁，NotifyThread随后获取了对象的锁，并调用对象的notify()方法，将WaitThread从WaitQueue移到SynchronizedQueue中，此时WaitThread的状态变为阻塞状态。

NotifyThread释放了锁之后，WaitThread再次获取到锁并从wait()方法返回继续执行。

### 4.3.3 等待/通知的经典范式

等待/通知的经典范式，该范式分为两部分，分别针对等待方（消费者）和通知方（生产者）。

等待方遵循如下原则。

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

对应的伪代码如下。

```

synchronized(对象) {

    while(条件不满足) {

        对象.wait();

    }

    对应的处理逻辑

}

```

通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。

对应的伪代码如下。

```

synchronized(对象) {

    改变条件

    对象.notifyAll();

}

```

### 4.3.4 管道输入/输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

管道输入/输出流主要包括了如下4种具体实现：

PipedOutputStream、PipedInputStream、PipedReader和PipedWriter。

前两种面向字节，而后两种面向字符。

示例：

创建了printThread，它用来接受main线程的输入，任何main线程的输入均通过PipedWriter写入，而printThread在另一端通过PipedReader将内容读出并打印。

```java

import java.io.IOException;

import java.io.PipedReader;

import java.io.PipedWriter;

public class Piped {

    public static void main(String[] args) throws Exception {

        PipedWriter out = new PipedWriter();

        PipedReader in = new PipedReader();

        // 将输出流和输入流进行连接，否则在使用时会抛出IOException

        out.connect(in);

        Thread printThread = new Thread(new Print(in), "PrintThread");

        printThread.start();

        int receive = 0;

        try {

            while ((receive = System.in.read()) != -1) {

                out.write(receive);

            }

        } finally {

            out.close();

        }

    }

    static class Print implements Runnable {

        private PipedReader in;

        public Print(PipedReader in) {

            this.in = in;

        }

        public void run() {

            int receive = 0;

            try {

                while ((receive = in.read()) != -1) {

                    System.out.print((char) receive);

                }

            } catch (IOException ex) {

            }

        }

    }

}

```

结果：

运行该示例，输入一组字符串，可以看到被printThread进行了原样输出。

```

asfsdf

asfsdf

```

注意：

对于Piped类型的流，必须先要进行绑定，也就是调用connect()方法，如果没有将输入/输出流绑定起来，对于该流的访问将会抛出异常。

### 4.3.5 Thread.join()的使用

如果一个线程A执行了thread.join()语句，其含义是：

当前线程A等待thread线程终止之后才从thread.join()返回。

线程Thread除了提供join()方法之外，还提供了join(long millis)和join(longmillis,int nanos)两个具备超时特性的方法。

这两个超时方法表示，如果线程thread在给定的超时时间里没有终止，那么将会从该超时方法中返回。

示例：

创建了10个线程，编号0~9，每个线程调用前一个线程的join()方法，也就是线程0结束了，线程1才能从join()方法中返回，而线程0需要等待main线程结束。

```java

import java.util.concurrent.TimeUnit;

public class Join {

    public static void main(String[] args) throws Exception {

        Thread previous = Thread.currentThread();

        for (int i = 0; i < 10; i++) {

            // 每个线程拥有前一个线程的引用，需要等待前一个线程终止，才能从等待中返回

            Thread thread = new Thread(new Domino(previous), String.valueOf(i));

            thread.start();

            previous = thread;

        }

        TimeUnit.SECONDS.sleep(5);

        System.out.println(Thread.currentThread().getName() + " terminate.");

    }

    static class Domino implements Runnable {

        private Thread thread;

        public Domino(Thread thread) {

            this.thread = thread;

        }

        public void run() {

            try {

                thread.join();

            } catch (InterruptedException e) {

            }

            System.out.println(Thread.currentThread().getName() + " terminate.");

        }

    }

}

```

结果：

```

main terminate.

0 terminate.

1 terminate.

2 terminate.

3 terminate.

4 terminate.

5 terminate.

6 terminate.

7 terminate.

8 terminate.

9 terminate.

```

每个线程终止的前提是前驱线程的终止，每个线程等待前驱线程终止后，才从join()方法返回，这里涉及了等待/通知机制（等待前驱线程结束，接收前驱线程结束通知）。

Thread.join()方法的源码

```java

package java.lang;

public

class Thread implements Runnable {

    public final void join() throws InterruptedException {

        join(0);

    }

    public final native boolean isAlive();

    // 加锁当前线程对象

    public final synchronized void join(long millis)

    throws InterruptedException {

        long base = System.currentTimeMillis();

        long now = 0;

        if (millis < 0) {

            throw new IllegalArgumentException("timeout value is negative");

        }

        if (millis == 0) {

            // 条件不满足，继续等待

            while (isAlive()) {

                wait(0);

            }

        } else {

            // 条件符合，方法返回

            while (isAlive()) {

                long delay = millis - now;

                if (delay <= 0) {

                    break;

                }

                wait(delay);

                now = System.currentTimeMillis() - base;

            }

        }

    }

}

```

当线程终止时，会调用线程自身的notifyAll()方法，会通知所有等待在该线程对象上的线程。

可以看到join()方法的逻辑结构与4.3.3节中描述的等待/通知经典范式一致，即加锁、循环和处理逻辑3个步骤。

### 4.3.6 ThreadLocal的使用

ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构。

这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。

可以通过set(T)方法来设置一个值，在当前线程下再通过get()方法获取到原先设置的值。

```java

import java.util.concurrent.TimeUnit;

public class Profiler {

    // 第一次get()方法调用时会进行初始化（如果set方法没有调用），每个线程会调用一次

    private static final ThreadLocal<Long> TIME\_THREADLOCAL = new ThreadLocal<Long>() {

        protected Long initialValue() {

            return System.currentTimeMillis();

        }

    };

    public static final void begin() {

        TIME\_THREADLOCAL.set(System.currentTimeMillis());

    }

    public static final long end() {

        return System.currentTimeMillis() - TIME\_THREADLOCAL.get();

    }

    public static void main(String[] args) throws Exception {

        Profiler.begin();

        TimeUnit.SECONDS.sleep(1);

        System.out.println("Cost: " + Profiler.end() + " mills");

    }

}

```

结果：

Cost: 1013 mills

## 4.4 线程应用实例

### 4.4.1 等待超时模式

调用场景：

调用一个方法时等待一段时间（一般来说是给定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。

只需要对经典范式做出非常小的改动，改动内容如下所示：

假设超时时间段是T，那么可以推断出在当前时间now+T之后就会超时。

定义如下变量：

等待持续时间：REMAINING=T。

超时时间：FUTURE=now+T。

这时仅需要wait(REMAINING)即可，在wait(REMAINING)返回之后会将执行：

REMAINING=FUTURE–now。如果REMAINING小于等于0，表示已经超时，直接退出，否则将继续执行wait(REMAINING)。

上述描述等待超时模式的伪代码如下。

```

// 对当前对象加锁

public synchronized Object get(long mills) throws InterruptedException {

    long future = System.currentTimeMillis() + mills;

    long remaining = mills;

    // 当超时大于0并且result返回值不满足要求

    while ((result == null) && remaining > 0) {

        wait(remaining);

        remaining = future - System.currentTimeMillis();

    }

    return result;

}

```

可以看出，等待超时模式就是在等待/通知范式基础上增加了超时控制，这使得该模式相比原有范式更具有灵活性，因为即使方法执行时间过长，也不会“永久”阻塞调用者，而是会按照调用者的要求“按时”返回。

### 4.4.2 一个简单的数据库连接池示例

使用等待超时模式来构造一个简单的数据库连接池。

在示例中模拟从连接池中获取、使用和释放连接的过程，而客户端获取连接的过程被设定为等待超时的模式，也就是在1000毫秒内如果无法获取到可用连接，将会返回给客户端一个null。

设定连接池的大小为10个，然后通过调节客户端的线程数来模拟无法获取连接的场景。

首先看一下连接池的定义。

通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用fetchConnection(long)方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用releaseConnection(Connection)方法将连接放回线程池。

```java

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.sql.Connection;

import java.util.concurrent.TimeUnit;

public class ConnectionDriver {

    static class ConnectionHandler implements InvocationHandler {

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

            if (method.getName().equals("commit")) {

                TimeUnit.MILLISECONDS.sleep(100);

            }

            return null;

        }

    }

    // 创建一个Connection的代理，在commit时休眠100毫秒

    public static final Connection createConnection() {

        return (Connection) Proxy.newProxyInstance(ConnectionDriver.class.getClassLoader(),

                new Class<?>[] { Connection.class }, new ConnectionHandler());

    }

}

```

```java

import java.sql.Connection;

import java.util.LinkedList;

public class ConnectionPool {

    private LinkedList<Connection> pool = new LinkedList<Connection>();

    public ConnectionPool(int initialSize) {

        if (initialSize > 0) {

            for (int i = 0; i < initialSize; i++) {

                pool.addLast(ConnectionDriver.createConnection());

            }

        }

    }

    public void releaseConnection(Connection connection) {

        if (connection != null) {

            synchronized (pool) {

                // 连接释放后需要进行通知，这样其他消费者能够感知到连接池中已经归还了一个连接

                pool.addLast(connection);

                pool.notifyAll();

            }

        }

    }

    // 在mills内无法获取到连接，将会返回null

    public Connection fetchConnection(long mills) throws InterruptedException {

        synchronized (pool) {

            // 完全超时

            if (mills <= 0) {

                while (pool.isEmpty()) {

                    pool.wait();

                }

                return pool.removeFirst();

            } else {

                long future = System.currentTimeMillis() + mills;

                long remaining = mills;

                while (pool.isEmpty() && remaining > 0) {

                    pool.wait(remaining);

                    remaining = future - System.currentTimeMillis();

                }

                Connection result = null;

                if (!pool.isEmpty()) {

                    result = pool.removeFirst();

                }

                return result;

            }

        }

    }

}

```

```java

import java.sql.Connection;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.atomic.AtomicInteger;

public class ConnectionPoolTest {

    static ConnectionPool pool = new ConnectionPool(10);

    // 保证所有ConnectionRunner能够同时开始

    static CountDownLatch start = new CountDownLatch(1);

    // main线程将会等待所有ConnectionRunner结束后才能继续执行

    static CountDownLatch end;

    public static void main(String[] args) throws Exception {

        // 线程数量，可以修改线程数量进行观察

        int threadCount = 10;

        end = new CountDownLatch(threadCount);

        int count = 20;

        AtomicInteger got = new AtomicInteger();

        AtomicInteger notGot = new AtomicInteger();

        for (int i = 0; i < threadCount; i++) {

            Thread thread = new Thread(new ConnetionRunner(count, got, notGot),

                    "ConnectionRunnerThread");

            thread.start();

        }

        start.countDown();

        end.await();

        System.out.println("total invoke: " + (threadCount \* count));

        System.out.println("got connection: " + got);

        System.out.println("not got connection " + notGot);

    }

    static class ConnetionRunner implements Runnable {

        int count;

        AtomicInteger got;

        AtomicInteger notGot;

        public ConnetionRunner(int count, AtomicInteger got, AtomicInteger notGot) {

            this.count = count;

            this.got = got;

            this.notGot = notGot;

        }

        public void run() {

            try {

                start.await();

            } catch (Exception ex) {

            }

            while (count > 0) {

                try {

                    // 从线程池中获取连接，如果1000ms内无法获取到，将会返回null

                    // 分别统计连接获取的数量got和未获取到的数量notGot

                    Connection connection = pool.fetchConnection(1000);

                    if (connection != null) {

                        try {

                            connection.createStatement();

                            connection.commit();

                        } finally {

                            pool.releaseConnection(connection);

                            got.incrementAndGet();

                        }

                    } else {

                        notGot.incrementAndGet();

                    }

                } catch (Exception ex) {

                } finally {

                    count--;

                }

            }

            end.countDown();

        }

    }

}

```

运行结果：

```

total invoke: 200

got connection: 200

not got connection 0

```

由于java.sql.Connection是一个接口，最终的实现是由数据库驱动提供方来实现的，考虑到只是个示例，我们通过动态代理构造了一个Connection，该Connection的代理实现仅仅是在commit()方法调用时休眠100毫秒。

示例中使用了CountDownLatch来确保ConnectionRunnerThread能够同时开始执行，并且在全部结束之后，才使main线程从等待状态中返回。

当前设定的场景是10个线程同时运行获取连接池（10个连接）中的连接，通过调节线程数量来观察未获取到连接的情况。

线程数、总获取次数、获取到的数量、未获取到的数量以及未获取到的比率



从表中的数据统计可以看出，在资源一定的情况下（连接池中的10个连接），随着客户端线程的逐步增加，客户端出现超时无法获取连接的比率不断升高。

虽然客户端线程在这种超时获取的模式下会出现连接无法获取的情况，但是它能够保证客户端线程不会一直挂在连接获取的操作上，而是“按时”返回，并告知客户端连接获取出现问题，是系统的一种自我保护机制。

数据库连接池的设计也可以复用到其他的资源获取的场景，针对昂贵资源（比如数据库连接）的获取都应该加以超时限制。

### 4.4.3 线程池技术及其示例

线程的创建和消亡都是需要耗费系统资源的

系统频繁的进行线程上下文切换，无故增加系统的负载

线程池技术能够很好地解决这个问题，它预先创建了若干数量的线程，并且不能由用户直接对线程的创建进行控制，在这个前提下重复使用固定或较为固定数目的线程来完成任务的执行。

这样做的好处是，一方面，消除了频繁创建和消亡线程的系统资源开销，另一方面，面对过量任务的提交能够平缓的劣化。

一个简单的线程池接口定义

```

public interface ThreadPool<Job extends Runnable> {

    // 执行一个Job，这个Job需要实现Runnable

    void execute(Job job);

    // 关闭线程池

    void shutdown();

    // 增加工作者线程

    void addWorkers(int num);

    // 减少工作者线程

    void removeWorker(int num);

    // 得到正在等待执行的任务数量

    int getJobSize();

}

```

客户端可以通过execute(Job)方法将Job提交入线程池执行，而客户端自身不用等待Job的执行完成。

除了execute(Job)方法以外，线程池接口提供了增大/减少工作者线程以及关闭线程池的方法。

这里工作者线程代表着一个重复执行Job的线程，而每个由客户端提交的Job都将进入到一个工作队列中等待工作者线程的处理。

实现：

```java

import java.util.ArrayList;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

import java.util.concurrent.atomic.AtomicLong;

public class DefaultThreadPool<Job extends Runnable> implements ThreadPool<Job> {

    // 线程池最大限制数

    private static final int MAX\_WORKER\_NUMBERS = 10;

    // 线程池默认的数量

    private static final int DEFAULT\_WORKER\_NUMBERS = 5;

    // 线程池最小的数量

    private static final int MIN\_WORKER\_NUMBERS = 1;

    // 这是一个工作列表，将会向里面插入工作

    private final LinkedList<Job> jobs = new LinkedList<Job>();

    // 工作者列表

    private final List<Worker> workers = Collections.synchronizedList(new

            ArrayList<Worker>());

    // 工作者线程的数量

    private int workerNum = DEFAULT\_WORKER\_NUMBERS;

    // 线程编号生成

    private AtomicLong threadNum = new AtomicLong();

    public DefaultThreadPool() {

        initializeWokers(DEFAULT\_WORKER\_NUMBERS);

    }

    public DefaultThreadPool(int num) {

        workerNum = num > MAX\_WORKER\_NUMBERS ? MAX\_WORKER\_NUMBERS :

                num < MIN\_WORKER\_NUMBERS ? MIN\_WORKER\_NUMBERS : num;

        initializeWokers(workerNum);

    }

    public void execute(Job job) {

        if (job != null) {

            // 添加一个工作，然后进行通知

            synchronized (jobs) {

                jobs.addLast(job);

                jobs.notify();

            }

        }

    }

    public void shutdown() {

        for (Worker worker : workers) {

            worker.shutdown();

        }

    }

    public void addWorkers(int num) {

        synchronized (jobs) {

            // 限制新增的Worker数量不能超过最大值

            if (num + this.workerNum > MAX\_WORKER\_NUMBERS) {

                num = MAX\_WORKER\_NUMBERS - this.workerNum;

            }

            initializeWokers(num);

            this.workerNum += num;

        }

    }

    public void removeWorker(int num) {

        synchronized (jobs) {

            if (num >= this.workerNum) {

                throw new IllegalArgumentException("beyond workNum");

            }

            // 按照给定的数量停止Worker

            int count = 0;

            while (count < num) {

                Worker worker = workers.get(count);

                if (workers.remove(worker)) {

                    worker.shutdown();

                    count++;

                }

            }

            this.workerNum -= count;

        }

    }

    public int getJobSize() {

        return jobs.size();

    }

    // 初始化线程工作者

    private void initializeWokers(int num) {

        for (int i = 0; i < num; i++) {

            Worker worker = new Worker();

            workers.add(worker);

            Thread thread = new Thread(worker, "ThreadPool-Worker-" + threadNum.

                    incrementAndGet());

            thread.start();

        }

    }

    // 工作者，负责消费任务

    class Worker implements Runnable {

        // 是否工作

        private volatile boolean running = true;

        public void run() {

            while (running) {

                Job job = null;

                synchronized (jobs) {

                    // 如果工作者列表是空的，那么就wait

                    while (jobs.isEmpty()) {

                        try {

                            jobs.wait();

                        } catch (InterruptedException ex) {

                            // 感知到外部对WorkerThread的中断操作，返回

                            Thread.currentThread().interrupt();

                            return;

                        }

                    }

                    // 取出一个Job

                    job = jobs.removeFirst();

                }

                if (job != null) {

                    try {

                        job.run();

                    } catch (Exception ex) {

                        // 忽略Job执行中的Exception

                    }

                }

            }

        }

        public void shutdown() {

            running = false;

        }

    }

}

```

从线程池的实现可以看到，当客户端调用execute(Job)方法时，会不断地向任务列表jobs中添加Job，而每个工作者线程会不断地从jobs上取出一个Job进行执行，当jobs为空时，工作者线程进入等待状态。

添加一个Job后，对工作队列jobs调用了其notify()方法，而不是notifyAll()方法，因为能够确定有工作者线程被唤醒，这时使用notify()方法将会比notifyAll()方法获得更小的开销（避免将等待队列中的线程全部移动到阻塞队列中）。

可以看到，线程池的本质就是使用了一个线程安全的工作队列连接工作者线程和客户端线程，客户端线程将任务放入工作队列后便返回，而工作者线程则不断地从工作队列上取出工作并执行。

当工作队列为空时，所有的工作者线程均等待在工作队列上，当有客户端提交了一个任务之后会通知任意一个工作者线程，随着大量的任务被提交，更多的工作者线程会被唤醒。

### 4.4.4 一个基于线程池技术的简单Web服务器

如果Web服务器是单线程的，多线程的浏览器也没有用武之地，因为服务端还是一个请求一个请求的顺序处理。

因此，大部分Web服务器都是支持并发访问的。常用的Java Web服务器，如Tomcat、Jetty，在其处理请求的过程中都使用到了线程池技术。

通过使用前一节中的线程池来构造一个简单的Web服务器。

这个Web服务器用来处理HTTP请求，目前只能处理简单的文本和JPG图片内容。这个Web服务器使用main线程不断地接受客户端Socket的连接，将连接以及请求提交给线程池处理，这样使得Web服务器能够同时处理多个客户端请求。

```java

import java.io.BufferedReader;

import java.io.ByteArrayOutputStream;

import java.io.Closeable;

import java.io.File;

import java.io.FileInputStream;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class SimpleHttpServer {

    // 处理HttpRequest的线程池

    static ThreadPool<HttpRequestHandler> threadPool = new DefaultThreadPool

            <HttpRequestHandler>(1);

    // SimpleHttpServer的根路径

    static String basePath;

    static ServerSocket serverSocket;

    // 服务监听端口

    static int port = 8080;

    public static void setPort(int port) {

        if (port > 0) {

            SimpleHttpServer.port = port;

        }

    }

    public static void setBasePath(String basePath) {

        if (basePath != null && new File(basePath).exists() && new File(basePath).

                isDirectory()) {

            SimpleHttpServer.basePath = basePath;

        }

    }

    // 启动SimpleHttpServer

    public static void start() throws Exception {

        serverSocket = new ServerSocket(port);

        Socket socket = null;

        while ((socket = serverSocket.accept()) != null) {

            // 接收一个客户端Socket，生成一个HttpRequestHandler，放入线程池执行

            threadPool.execute(new HttpRequestHandler(socket));

        }

        serverSocket.close();

    }

    static class HttpRequestHandler implements Runnable {

        private Socket socket;

        public HttpRequestHandler(Socket socket) {

            this.socket = socket;

        }

        @Override

        public void run() {

            String line = null;

            BufferedReader br = null;

            BufferedReader reader = null;

            PrintWriter out = null;

            InputStream in = null;

            try {

                reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

                String header = reader.readLine();

                // 由相对路径计算出绝对路径

                String filePath = basePath + header.split(" ")[1];

                out = new PrintWriter(socket.getOutputStream());

                // 如果请求资源的后缀为jpg或者ico，则读取资源并输出

                if (filePath.endsWith("jpg") || filePath.endsWith("ico")) {

                    in = new FileInputStream(filePath);

                    ByteArrayOutputStream baos = new ByteArrayOutputStream();

                    int i = 0;

                    while ((i = in.read()) != -1) {

                        baos.write(i);

                    }

                    byte[] array = baos.toByteArray();

                    out.println("HTTP/1.1 200 OK");

                    out.println("Server: Molly");

                    out.println("Content-Type: image/jpeg");

                    out.println("Content-Length: " + array.length);

                    out.println("");

                    socket.getOutputStream().write(array, 0, array.length);

                } else {

                    br = new BufferedReader(new InputStreamReader(new

                            FileInputStream(filePath)));

                    out = new PrintWriter(socket.getOutputStream());

                    out.println("HTTP/1.1 200 OK");

                    out.println("Server: Molly");

                    out.println("Content-Type: text/html; charset=UTF-8");

                    out.println("");

                    while ((line = br.readLine()) != null) {

                        out.println(line);

                    }

                }

                out.flush();

            } catch (Exception ex) {

                out.println("HTTP/1.1 500");

                out.println("");

                out.flush();

            } finally {

                close(br, in, reader, out, socket);

            }

        }

    }

    // 关闭流或者Socket

    private static void close(Closeable... closeables) {

        if (closeables != null) {

            for (Closeable closeable : closeables) {

                try {

                    closeable.close();

                } catch (Exception ex) {

                }

            }

        }

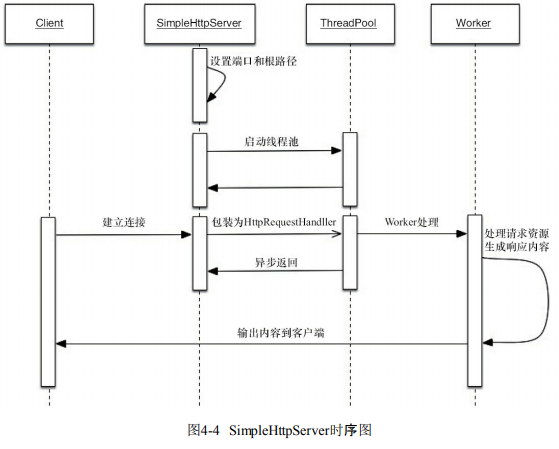
    }

}

```

SimpleHttpServer在建立了与客户端的连接之后，并不会处理客户端的请求，而是将其包装成HttpRequestHandler并交由线程池处理。

在线程池中的Worker处理客户端请求的同时，SimpleHttpServer能够继续完成后续客户端连接的建立，不会阻塞后续客户端的请求。



测试：

Index.html

```

<html>

<head>

    <title>测试页面</title>

</head>

<body >

    <h1>第一张图片</h1>

    <img src="1.jpg" />

    <h1>第二张图片</h1>

    <img src="2.jpg" />

    <h1>第三张图片</h1>

    <img src="3.jpg" />

</body>

</html>

```

将SimpleHttpServer的根目录设定到该HTML页面所在目录，并启动SimpleHttpServer，通过Apache HTTP server benchmarking tool（版本2.3）来测试不同线程数下，SimpleHttpServer的吞吐量表现。

线程池中线程数量并不是越多越好，具体的数量需要评估每个任务的处理时间，以及当前计算机的处理器能力和数量。

使用的线程过少，无法发挥处理器的性能；使用的线程过多，将会增加系统的无故开销，起到相反的作用。

# 5. 锁

【锁相关的API和组件】

Java并发包中与锁相关的API和组件，以及这些API和组件的使用方式和实现细节。

## 5.1 Lock接口

锁是用来控制多个线程访问共享资源的方式。

在Lock接口出现之前，Java程序是靠synchronized关键字实现锁功能的，而Java SE 5之后，并发包中新增了Lock接口（以及相关实现类）用来实现锁功能，它提供了与synchronized关键字类似的同步功能，只是在使用时需要显式地获取和释放锁。

虽然它缺少了（通过synchronized块或者方法所提供的）隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等多种synchronized关键字所不具备的同步特性。

使用synchronized关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放。

当然，这种方式简化了同步的管理，可是扩展性没有显示的锁获取和释放来的好。

Lock的使用的方式：

```java

Lock lock = new ReentrantLock();

lock.lock();

try {

} finally {

    lock.unlock();

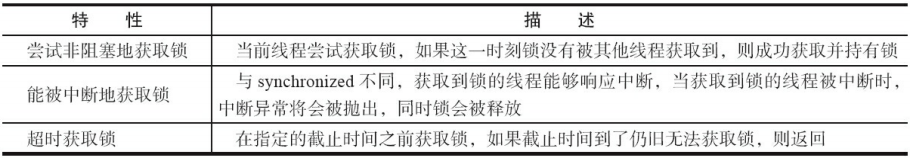
}

```

在finally块中释放锁，目的是保证在获取到锁之后，最终能够被释放。

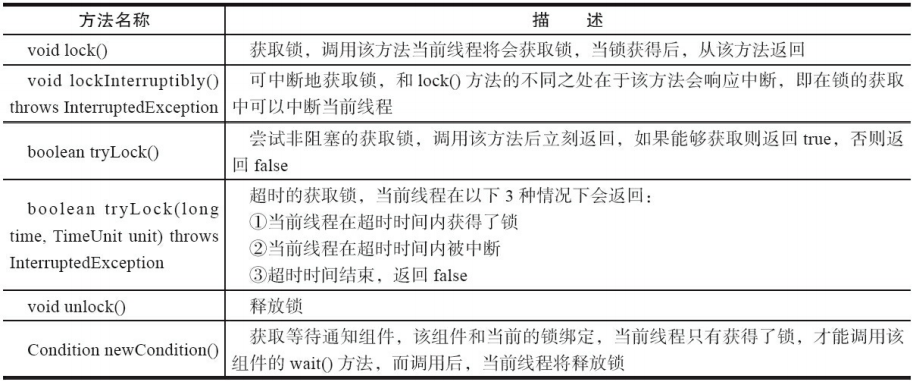
不要将获取锁的过程写在try块中，因为如果在获取锁（自定义锁的实现）时发生了异常，异常抛出的同时，也会导致锁无故释放。

Lock接口提供的synchronized关键字所不具备的主要特性



Lock是一个接口，它定义了锁获取和释放的基本操作。

Lock的API



同步器AbstractQueuedSynchronizer以及常用Lock接口的实现ReentrantLock。

Lock接口的实现基本都是通过聚合了一个同步器的子类来完成线程访问控制的。

## 5.2 队列同步器

队列同步器AbstractQueuedSynchronizer（以下简称同步器）：

是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作，并发包的作者（Doug Lea）期望它能够成为实现大部分同步需求的基础。

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态。

在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法（getState()、setState(int newState)和compareAndSetState(int expect,int update)）来进行操作，因为它们能够保证状态的改变是安全的。

子类推荐被定义为自定义同步组件的静态内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用。

同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型的同步组件（ReentrantLock、ReentrantReadWriteLock和CountDownLatch等）。

同步器是实现锁（也可以是任意同步组件）的关键，在锁的实现中聚合同步器，利用同步器实现锁的语义。

可以这样理解二者之间的关系：

锁是面向使用者的，它定义了使用者与锁交互的接口（比如可以允许两个线程并行访问），隐藏了实现细节；

同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。

锁和同步器很好地隔离了使用者和实现者所需关注的领域。

### 5.2.1 队列同步器的接口与示例

同步器的设计是基于模板方法模式的，也就是说，使用者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。

重写同步器指定的方法时，需要使用同步器提供的如下3个方法来访问或修改同步状态。

getState()：

获取当前同步状态。

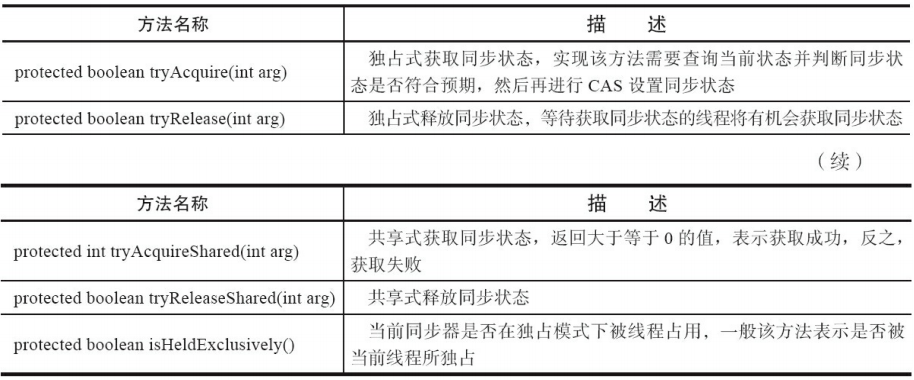
setState(int newState)：

设置当前同步状态。

compareAndSetState(int expect,int update)：

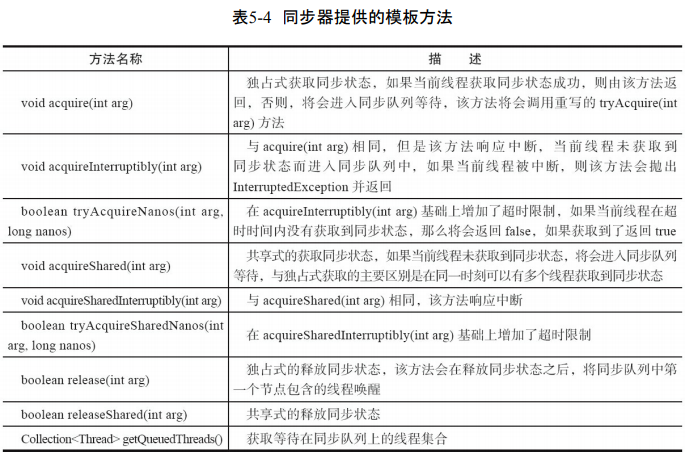
使用CAS设置当前状态，该方法能够保证状态设置的原子性。

同步器可重写的方法



实现自定义同步组件时，将会调用同步器提供的模板方法。

同步器提供的模板方法



同步器提供的模板方法基本上分为3类：

独占式获取与释放同步状态、共享式获取与释放同步状态和查询同步队列中的等待线程情况。

自定义同步组件将使用同步器提供的模板方法来实现自己的同步语义。

只有掌握了同步器的工作原理才能更加深入地理解并发包中其他的并发组件。

一个独占锁的示例

```java

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.AbstractQueuedSynchronizer;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

public class Mutex implements Lock {

    // 静态内部类，自定义同步器

    private static class Sync extends AbstractQueuedSynchronizer {

        // 是否处于占用状态

        protected boolean isHeldExclusively() {

            return getState() == 1;

        }

        // 当状态为0的时候获取锁

        public boolean tryAcquire(int acquires) {

            if (compareAndSetState(0, 1)) {

                setExclusiveOwnerThread(Thread.currentThread());

                return true;

            }

            return false;

        }

        // 释放锁，将状态设置为0

        protected boolean tryRelease(int releases) {

            if (getState() == 0) throw new

                    IllegalMonitorStateException();

            setExclusiveOwnerThread(null);

            setState(0);

            return true;

        }

        // 返回一个Condition，每个condition都包含了一个condition队列

        Condition newCondition() { return new ConditionObject(); }

    }

    // 仅需要将操作代理到Sync上即可

    private final Sync sync = new Sync();

    public void lock() { sync.acquire(1); }

    public boolean tryLock() { return sync.tryAcquire(1); }

    public void unlock() { sync.release(1); }

    public Condition newCondition() { return sync.newCondition(); }

    public boolean isLocked() { return sync.isHeldExclusively(); }

    public boolean hasQueuedThreads() { return sync.hasQueuedThreads(); }

    public void lockInterruptibly() throws InterruptedException {

        sync.acquireInterruptibly(1);

    }

    public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {

        return sync.tryAcquireNanos(1, unit.toNanos(timeout));

    }

}

```

顾名思义，独占锁就是在同一时刻只能有一个线程获取到锁，而其他获取锁的线程只能处于同步队列中等待，只有获取锁的线程释放了锁，后继的线程才能够获取锁。

上述示例中，独占锁Mutex是一个自定义同步组件，它在同一时刻只允许一个线程占有锁。

Mutex中定义了一个静态内部类，该内部类继承了同步器并实现了独占式获取和释放同步状态。

在tryAcquire(int acquires)方法中，如果经过CAS设置成功（同步状态设置为1），则代表获取了同步状态，而在tryRelease(int releases)方法中只是将同步状态重置为0。

用户使用Mutex时并不会直接和内部同步器的实现打交道，而是调用Mutex提供的方法，在Mutex的实现中，以获取锁的lock()方法为例，只需要在方法实现中调用同步器的模板方法acquire(int args)即可，当前线程调用该方法获取同步状态失败后会被加入到同步队列中等待，这样就大大降低了实现一个可靠自定义同步组件的门槛。

### 5.2.2 队列同步器的实现分析

#### 1.同步队列

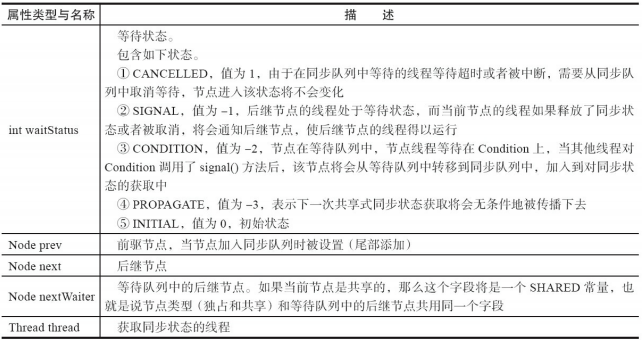
同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理。

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。

同步队列中的节点（Node）用来保存获取同步状态失败的线程引用、等待状态以及前驱和后继节点。

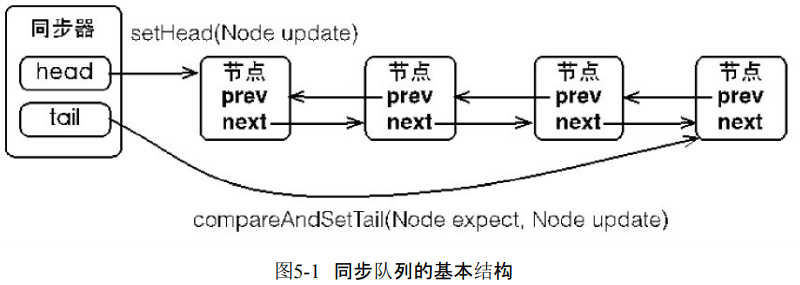
节点的属性类型与名称以及描述



节点是构成同步队列（等待队列，在5.6节中将会介绍）的基础。

同步器拥有首节点（head）和尾节点（tail），没有成功获取同步状态的线程将会成为节点加入该队列的尾部。

同步队列的基本结构

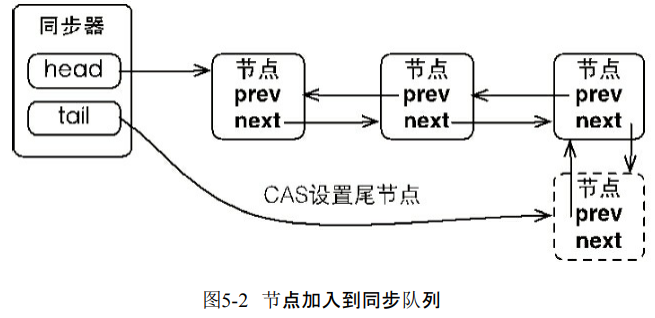


同步器包含了两个节点类型的引用，一个指向头节点，而另一个指向尾节点。

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。

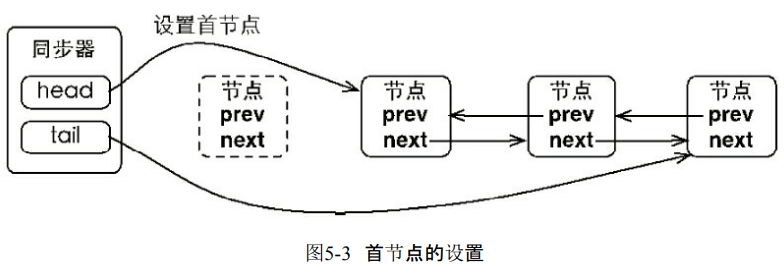
因此同步器提供了一个基于CAS的设置尾节点的方法：compareAndSetTail(Node expect,Node update)，它需要传递当前线程“认为”的尾节点和当前节点，只有设置成功后，当前节点才正式与之前的尾节点建立关联。

同步器将节点加入到同步队列的过程



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。

#### 2.独占式同步状态获取与释放

通过调用同步器的acquire(int arg)方法可以获取同步状态。

该方法对中断不敏感，也就是由于线程获取同步状态失败后进入同步队列中，后续对线程进行中断操作时，线程不会从同步队列中移出。

源码：

```java

package java.util.concurrent.locks;

public abstract class AbstractQueuedSynchronizer

    extends AbstractOwnableSynchronizer

    implements java.io.Serializable {

    // Acquires in exclusive mode, ignoring interrupts.

    public final void acquire(int arg) {

        if (!tryAcquire(arg) &&

            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

            selfInterrupt();

}

protected boolean tryAcquire(int arg) {

    throw new UnsupportedOperationException();

}

}

```

上述代码主要完成了同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的相关工作。

其主要逻辑是：

首先调用自定义同步器实现的tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态。

如果同步状态获取失败，则构造同步节点（独占式Node.EXCLUSIVE，同一时刻只能有一个线程成功获取同步状态）并通过addWaiter(Node node)方法将该节点加入到同步队列的尾部。

最后调用acquireQueued(Node node,int arg)方法，使得该节点以“死循环”的方式获取同步状态。

如果获取不到则阻塞节点中的线程，而被阻塞线程的唤醒主要依靠前驱节点的出队或阻塞线程被中断来实现。

节点的构造以及加入同步队列源码

```java

/\*\*

 \* Creates and enqueues node for current thread and given mode.

 \*

 \* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared

 \* @return the new node

 \*/

private Node addWaiter(Node mode) {

    Node node = new Node(Thread.currentThread(), mode);

    // 快速尝试在尾部添加

    // Try the fast path of enq; backup to full enq on failure

    Node pred = tail;

    if (pred != null) {

        node.prev = pred;

        if (compareAndSetTail(pred, node)) {

            pred.next = node;

            return node;

        }

    }

    enq(node);

    return node;

}

/\*\*

 \* Inserts node into queue, initializing if necessary. See picture above.

 \* @param node the node to insert

 \* @return node's predecessor

 \*/

private Node enq(final Node node) {

    for (;;) {

        Node t = tail;

        if (t == null) { // Must initialize

            if (compareAndSetHead(new Node()))

                tail = head;

        } else {

            node.prev = t;

            if (compareAndSetTail(t, node)) {

                t.next = node;

                return t;

            }

        }

    }

}

```

上述代码通过使用compareAndSetTail(Node expect,Node update)方法来确保节点能够被线程安全添加。

试想一下：

如果使用一个普通的LinkedList来维护节点之间的关系，那么当一个线程获取了同步状态，而其他多个线程由于调用tryAcquire(int arg)方法获取同步状态失败而并发地被添加到LinkedList时，LinkedList将难以保证Node的正确添加，最终的结果可能是节点的数量有偏差，而且顺序也是混乱的。

在enq(final Node node)方法中，同步器通过“死循环”来保证节点的正确添加，在“死循环”中只有通过CAS将节点设置成为尾节点之后，当前线程才能从该方法返回，否则，当前线程不断地尝试设置。

可以看出，enq(final Node node)方法将并发添加节点的请求通过CAS变得“串行化”了。

节点进入同步队列之后，就进入了一个自旋的过程，每个节点（或者说每个线程）都在自省地观察，当条件满足，获取到了同步状态，就可以从这个自旋过程中退出，否则依旧留在这个自旋过程中（并会阻塞节点的线程）。

```java

/\*\*

 \* Acquires in exclusive uninterruptible mode for thread already in

 \* queue. Used by condition wait methods as well as acquire.

 \*

 \* @param node the node

 \* @param arg the acquire argument

 \* @return {@code true} if interrupted while waiting

 \*/

final boolean acquireQueued(final Node node, int arg) {

    boolean failed = true;

    try {

        boolean interrupted = false;

        for (;;) {

            final Node p = node.predecessor();

            if (p == head && tryAcquire(arg)) {

                setHead(node);

                p.next = null; // help GC

                failed = false;

                return interrupted;

            }

            if (shouldParkAfterFailedAcquire(p, node) &&

                parkAndCheckInterrupt())

                interrupted = true;

        }

    } finally {

        if (failed)

            cancelAcquire(node);

    }

}

```

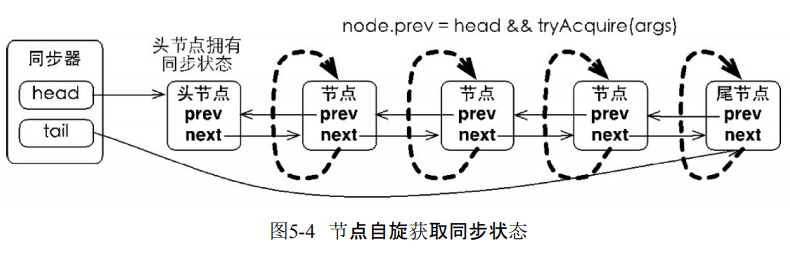
在acquireQueued(final Node node,int arg)方法中，当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头节点才能够尝试获取同步状态，这是为什么？

原因有两个，如下。

第一，头节点是成功获取到同步状态的节点，而头节点的线程释放了同步状态之后，将会唤醒其后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点。

第二，维护同步队列的FIFO原则。

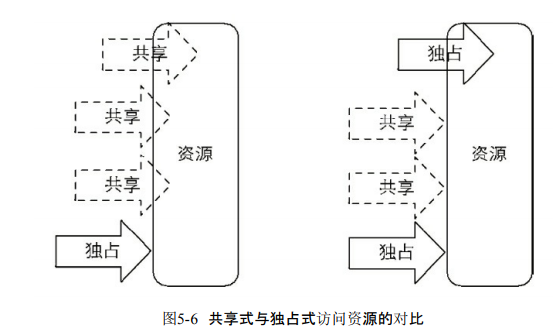
该方法中，节点自旋获取同步状态的行为

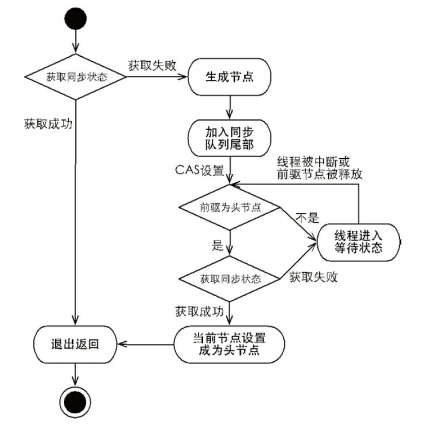


在图5-4中，由于非首节点线程前驱节点出队或者被中断而从等待状态返回，随后检查自己的前驱是否是头节点，如果是则尝试获取同步状态。

可以看到节点和节点之间在循环检查的过程中基本不相互通信，而是简单地判断自己的前驱是否为头节点，这样就使得节点的释放规则符合FIFO，并且也便于对过早通知的处理（过早通知是指前驱节点不是头节点的线程由于中断而被唤醒）。

独占式同步状态获取流程，也就是acquire(int arg)方法调用流程





在图5-5中，前驱节点为头节点且能够获取同步状态的判断条件和线程进入等待状态是获取同步状态的自旋过程。

当同步状态获取成功之后，当前线程从acquire(int arg)方法返回，如果对于锁这种并发组件而言，代表着当前线程获取了锁。

当前线程获取同步状态并执行了相应逻辑之后，就需要释放同步状态，使得后续节点能够继续获取同步状态。

通过调用同步器的release(int arg)方法可以释放同步状态，该方法在释放了同步状态之后，会唤醒其后继节点（进而使后继节点重新尝试获取同步状态）。

```java

public final boolean release(int arg) {

    if (tryRelease(arg)) {

        Node h = head;

        if (h != null && h.waitStatus != 0)

            unparkSuccessor(h);

        return true;

    }

    return false;

}

```

该方法执行时，会唤醒头节点的后继节点线程，unparkSuccessor(Node node)方法使用LockSupport（在后面的章节会专门介绍）来唤醒处于等待状态的线程。

总结：

在获取同步状态时，同步器维护一个同步队列，获取状态失败的线程都会被加入到队列中并在队列中进行自旋；

移出队列（或停止自旋）的条件是前驱节点为头节点且成功获取了同步状态。

在释放同步状态时，同步器调用tryRelease(int arg)方法释放同步状态，然后唤醒头节点的后继节点。

#### 3.共享式同步状态获取与释放

共享式获取与独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。

以文件的读写为例，如果一个程序在对文件进行读操作，那么这一时刻对于该文件的写操作均被阻塞，而读操作能够同时进行。

写操作要求对资源的独占式访问，而读操作可以是共享式访问。

两种不同的访问模式在同一时刻对文件或资源的访问情况



在图5-6中，左半部分，共享式访问资源时，其他共享式的访问均被允许，而独占式访问被阻塞，右半部分是独占式访问资源时，同一时刻其他访问均被阻塞。

通过调用同步器的acquireShared(int arg)方法可以共享式地获取同步状态

```java

public final void acquireShared(int arg) {

    if (tryAcquireShared(arg) < 0)

        doAcquireShared(arg);

}

protected int tryAcquireShared(int arg) {

    throw new UnsupportedOperationException();

}

/\*\*

 \* Acquires in shared uninterruptible mode.

 \* @param arg the acquire argument

 \*/

private void doAcquireShared(int arg) {

    final Node node = addWaiter(Node.SHARED);

    boolean failed = true;

    try {

        boolean interrupted = false;

        for (;;) {

            final Node p = node.predecessor();

            if (p == head) {

                int r = tryAcquireShared(arg);

                if (r >= 0) {

                    setHeadAndPropagate(node, r);

                    p.next = null; // help GC

                    if (interrupted)

                        selfInterrupt();

                    failed = false;

                    return;

                }

            }

            if (shouldParkAfterFailedAcquire(p, node) &&

                parkAndCheckInterrupt())

                interrupted = true;

        }

    } finally {

        if (failed)

            cancelAcquire(node);

    }

}

```

在acquireShared(int arg)方法中，同步器调用tryAcquireShared(int arg)方法尝试获取同步状态，tryAcquireShared(int arg)方法返回值为int类型，当返回值大于等于0时，表示能够获取到同步状态。

因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。

可以看到，在doAcquireShared(int arg)方法的自旋过程中，如果当前节点的前驱为头节点时，尝试获取同步状态，如果返回值大于等于0，表示该次获取同步状态成功并从自旋过程中退出。

与独占式一样，共享式获取也需要释放同步状态，通过调用releaseShared(int arg)方法可以释放同步状态。

```java

public final boolean releaseShared(int arg) {

    if (tryReleaseShared(arg)) {

        doReleaseShared();

        return true;

    }

    return false;

}

```

该方法在释放同步状态之后，将会唤醒后续处于等待状态的节点。

对于能够支持多个线程同时访问的并发组件（比如Semaphore），它和独占式主要区别在于tryReleaseShared(int arg)方法必须确保同步状态（或者资源数）线程安全释放，一般是通过循环和CAS来保证的，因为释放同步状态的操作会同时来自多个线程。

#### 4.独占式超时获取同步状态

通过调用同步器的doAcquireNanos(int arg,long nanosTimeout)方法可以超时获取同步状态，即在指定的时间段内获取同步状态，如果获取到同步状态则返回true，否则，返回false。

该方法提供了传统Java同步操作（比如synchronized关键字）所不具备的特性。

在分析该方法的实现前，先介绍一下响应中断的同步状态获取过程。

在Java 5之前，当一个线程获取不到锁而被阻塞在synchronized之外时，对该线程进行中断操作，此时该线程的中断标志位会被修改，但线程依旧会阻塞在synchronized上，等待着获取锁。

在Java 5中，同步器提供了acquireInterruptibly(int arg)方法，这个方法在等待获取同步状态时，如果当前线程被中断，会立刻返回，并抛出InterruptedException。

超时获取同步状态过程可以被视作响应中断获取同步状态过程的“增强版”。

doAcquireNanos(int arg,long nanosTimeout)方法在支持响应中断的基础上，增加了超时获取的特性。

针对超时获取，主要需要计算出需要睡眠的时间间隔nanosTimeout，为了防止过早通知，nanosTimeout计算公式为：nanosTimeout-=now-lastTime，其中now为当前唤醒时间，lastTime为上次唤醒时间。

如果nanosTimeout大于0则表示超时时间未到，需要继续睡眠nanosTimeout纳秒，反之，表示已经超时。

```java

/\*\*

 \* Acquires in exclusive timed mode.

 \*

 \* @param arg the acquire argument

 \* @param nanosTimeout max wait time

 \* @return {@code true} if acquired

 \*/

private boolean doAcquireNanos(int arg, long nanosTimeout)

        throws InterruptedException {

    if (nanosTimeout <= 0L)

        return false;

    final long deadline = System.nanoTime() + nanosTimeout;

    final Node node = addWaiter(Node.EXCLUSIVE);

    boolean failed = true;

    try {

        for (;;) {

            final Node p = node.predecessor();

            if (p == head && tryAcquire(arg)) {

                setHead(node);

                p.next = null; // help GC

                failed = false;

                return true;

            }

            // 计算时间，当前时间now减去睡眠之前的时间lastTime得到已经睡眠

            // 的时间delta，然后被原有超时时间nanosTimeout减去，得到了

            // 还应该睡眠的时间

            nanosTimeout = deadline - System.nanoTime();

            if (nanosTimeout <= 0L)

                return false;

            if (shouldParkAfterFailedAcquire(p, node) &&

                nanosTimeout > spinForTimeoutThreshold)

                LockSupport.parkNanos(this, nanosTimeout);

            if (Thread.interrupted())

                throw new InterruptedException();

        }

    } finally {

        if (failed)

            cancelAcquire(node);

    }

}

```

该方法在自旋过程中，当节点的前驱节点为头节点时尝试获取同步状态，如果获取成功则从该方法返回。

这个过程和独占式同步获取的过程类似，但是在同步状态获取失败的处理上有所不同。

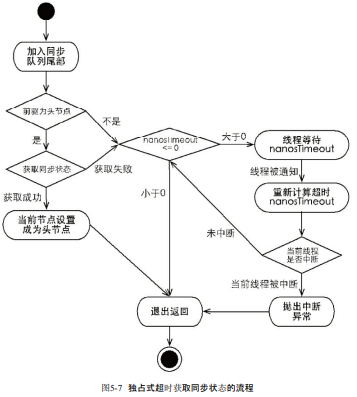
如果当前线程获取同步状态失败，则判断是否超时（nanosTimeout小于等于0表示已经超时），如果没有超时，重新计算超时间隔nanosTimeout，然后使当前线程等待nanosTimeout纳秒（当已到设置的超时时间，该线程会从LockSupport.parkNanos(Object blocker,long nanos)方法返回）。

如果nanosTimeout小于等于spinForTimeoutThreshold（1000纳秒）时，将不会使该线程进行超时等待，而是进入快速的自旋过程。

原因在于，非常短的超时等待无法做到十分精确，如果这时再进行超时等待，相反会让nanosTimeout的超时从整体上表现得反而不精确。

因此，在超时非常短的场景下，同步器会进入无条件的快速自旋。

独占式超时获取同步态的流程



从图5-7中可以看出，独占式超时获取同步状态doAcquireNanos(int arg,long nanosTimeout)和独占式获取同步状态acquire(int args)在流程上非常相似，其主要区别在于未获取到同步状态时的处理逻辑。

acquire(int args)在未获取到同步状态时，将会使当前线程一直处于等待状态，而doAcquireNanos(int arg,long nanosTimeout)会使当前线程等待nanosTimeout纳秒，如果当前线程在nanosTimeout纳秒内没有获取到同步状态，将会从等待逻辑中自动返回。

#### 5.自定义同步组件——TwinsLock

设计一个同步工具：

该工具在同一时刻，只允许至多两个线程同时访问，超过两个线程的访问将被阻塞

TwinsLock能够在同一时刻支持多个线程的访问，这显然是共享式访问，因此，需要使用同步器提供的acquireShared(int args)方法等和Shared相关的方法。

这就要求TwinsLock必须重写tryAcquireShared(int args)方法和tryReleaseShared(int args)方法，这样才能保证同步器的共享式同步状态的获取与释放方法得以执行。

其次，定义资源数。

TwinsLock在同一时刻允许至多两个线程的同时访问，表明同步资源数为2，这样可以设置初始状态status为2，当一个线程进行获取，status减1，该线程释放，则status加1，状态的合法范围为0、1和2，其中0表示当前已经有两个线程获取了同步资源，此时再有其他线程对同步状态进行获取，该线程只能被阻塞。

在同步状态变更时，需要使用compareAndSet(int expect,int update)方法做原子性保障。

```java

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.AbstractQueuedSynchronizer;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

public class TwinsLock implements Lock {

    private final Sync sync = new Sync(2);

    private static final class Sync extends AbstractQueuedSynchronizer {

        Sync(int count) {

            if (count <= 0) {

                throw new IllegalArgumentException("count must large than zero.");

            }

            setState(count);

        }

        public int tryAcquireShared(int reduceCount) {

            for (;;) {

                int current = getState();

                int newCount = current - reduceCount;

                if (newCount < 0 || compareAndSetState(current,

                        newCount)) {

                    return newCount;

                }

            }

        }

        public boolean tryReleaseShared(int returnCount) {

            for (;;) {

                int current = getState();

                int newCount = current + returnCount;

                if (compareAndSetState(current, newCount)) {

                    return true;

                }

            }

        }

    }

    public void lock() {

        sync.acquireShared(1);

    }

    public void unlock() {

        sync.releaseShared(1);

    }

    @Override

    public void lockInterruptibly() throws InterruptedException {

    }

    @Override

    public boolean tryLock() {

        return false;

    }

    @Override

    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {

        return false;

    }

    @Override

    public Condition newCondition() {

        return null;

    }

}

```

在上述示例中，TwinsLock实现了Lock接口，提供了面向使用者的接口，使用者调用lock()方法获取锁，随后调用unlock()方法释放锁，而同一时刻只能有两个线程同时获取到锁。

TwinsLock同时包含了一个自定义同步器Sync，而该同步器面向线程访问和同步状态控制。

以共享式获取同步状态为例：

同步器会先计算出获取后的同步状态，然后通过CAS确保状态的正确设置，当tryAcquireShared(int reduceCount)方法返回值大于等于0时，当前线程才获取同步状态，对于上层的TwinsLock而言，则表示当前线程获得了锁。

同步器作为一个桥梁，连接线程访问以及同步状态控制等底层技术与不同并发组件（比如Lock、CountDownLatch等）的接口语义。

测试：

```java

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

public class TwinsLockTest {

    public static void main(String[] args) throws InterruptedException {

        final Lock lock = new TwinsLock();

        class Worker extends Thread {

            public void run() {

                while (true) {

                    lock.lock();

                    try {

                        TimeUnit.SECONDS.sleep(1);

                        System.out.println(Thread.currentThread().getName());

                        TimeUnit.SECONDS.sleep(1);

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    } finally {

                        lock.unlock();

                    }

                }

            }

        }

        // 启动10个线程

        for (int i = 0; i < 10; i++) {

            Worker w = new Worker();

            w.setDaemon(true);

            w.start();

        }

        // 每隔1秒换行

        for (int i = 0; i < 10; i++) {

            TimeUnit.SECONDS.sleep(1);

            System.out.println();

        }

    }

}

```

运行该测试用例，可以看到线程名称成对输出，也就是在同一时刻只有两个线程能够获取到锁，这表明TwinsLock可以按照预期正确工作。

## 5.3 重入锁

重入锁ReentrantLock，顾名思义，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。

除此之外，该锁的还支持获取锁时的公平和非公平性选择。

在同步器一节中的示例（Mutex），同时考虑如下场景：

当一个线程调用Mutex的lock()方法获取锁之后，如果再次调用lock()方法，则该线程将会被自己所阻塞，原因是Mutex在实现tryAcquire(int acquires)方法时没有考虑占有锁的线程再次获取锁的场景，而在调用tryAcquire(int acquires)方法时返回了false，导致该线程被阻塞。

简单地说，Mutex是一个不支持重进入的锁。

而synchronized关键字隐式的支持重进入，比如一个synchronized修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获得该锁，而不像Mutex由于获取了锁，而在下一次获取锁时出现阻塞自己的情况。

ReentrantLock虽然没能像synchronized关键字一样支持隐式的重进入，但是在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞。

这里提到一个锁获取的公平性问题，如果在绝对时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。

公平的获取锁，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。

ReentrantLock提供了一个构造函数，能够控制锁是否是公平的。

事实上，公平的锁机制往往没有非公平的效率高，但是，并不是任何场景都是以TPS作为唯一的指标，公平锁能够减少“饥饿”发生的概率，等待越久的请求越是能够得到优先满足。

1.实现重进入

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决以下两个问题。

1）线程再次获取锁。

锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。

2）锁的最终释放。

线程重复n次获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。

锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已经成功释放。

ReentrantLock是通过组合自定义同步器来实现锁的获取与释放。

```java

package java.util.concurrent.locks;

public class ReentrantLock implements Lock, java.io.Serializable {

    private final Sync sync;

    abstract static class Sync extends AbstractQueuedSynchronizer {

        final boolean nonfairTryAcquire(int acquires) {

            final Thread current = Thread.currentThread();

            int c = getState();

            if (c == 0) {

                if (compareAndSetState(0, acquires)) {

                    setExclusiveOwnerThread(current);

                    return true;

                }

            }

            else if (current == getExclusiveOwnerThread()) {

                int nextc = c + acquires;

                if (nextc < 0) // overflow

                    throw new Error("Maximum lock count exceeded");

                setState(nextc);

                return true;

            }

            return false;

        }

        protected final boolean tryRelease(int releases) {

            int c = getState() - releases;

            if (Thread.currentThread() != getExclusiveOwnerThread())

                throw new IllegalMonitorStateException();

            boolean free = false;

            if (c == 0) {

                free = true;

                setExclusiveOwnerThread(null);

            }

            setState(c);

            return free;

        }

    }

    protected Collection<Thread> getQueuedThreads() {

        return sync.getQueuedThreads();

    }

    /\*\*

     \* Sync object for non-fair locks

     \*/

    static final class NonfairSync extends Sync {

        private static final long serialVersionUID = 7316153563782823691L;

        /\*\*

         \* Performs lock.  Try immediate barge, backing up to normal

         \* acquire on failure.

         \*/

        final void lock() {

            if (compareAndSetState(0, 1))

                setExclusiveOwnerThread(Thread.currentThread());

            else

                acquire(1);

        }

        protected final boolean tryAcquire(int acquires) {

            return nonfairTryAcquire(acquires);

        }

    }

    /\*\*

     \* Sync object for fair locks

     \*/

    static final class FairSync extends Sync {

        private static final long serialVersionUID = -3000897897090466540L;

        final void lock() {

            acquire(1);

        }

        /\*\*

         \* Fair version of tryAcquire.  Don't grant access unless

         \* recursive call or no waiters or is first.

         \*/

        protected final boolean tryAcquire(int acquires) {

            final Thread current = Thread.currentThread();

            int c = getState();

            if (c == 0) {

                if (!hasQueuedPredecessors() &&

                    compareAndSetState(0, acquires)) {

                    setExclusiveOwnerThread(current);

                    return true;

                }

            }

            else if (current == getExclusiveOwnerThread()) {

                int nextc = c + acquires;

                if (nextc < 0)

                    throw new Error("Maximum lock count exceeded");

                setState(nextc);

                return true;

            }

            return false;

        }

    }

}

```

该方法增加了再次获取同步状态的处理逻辑：

通过判断当前线程是否为获取锁的线程来决定获取操作是否成功，如果是获取锁的线程再次请求，则将同步状态值进行增加并返回true，表示获取同步状态成功。

成功获取锁的线程再次获取锁，只是增加了同步状态值，这也就要求ReentrantLock在释放同步状态时减少同步状态值。

如果该锁被获取了n次，那么前(n-1)次tryRelease(int releases)方法必须返回false，而只有同步状态完全释放了，才能返回true。

可以看到，该方法将同步状态是否为0作为最终释放的条件，当同步状态为0时，将占有线程设置为null，并返回true，表示释放成功。

2.公平与非公平获取锁的区别

公平性与否是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求的绝对时间顺序，也就是FIFO。

对于非公平锁，只要CAS设置同步状态成功，则表示当前线程获取了锁，而公平锁则不同tryAcquire方法：

该方法与nonfairTryAcquire(int acquires)比较，唯一不同的位置为判断条件多了hasQueuedPredecessors()方法，即加入了同步队列中当前节点是否有前驱节点的判断，如果该方法返回true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

测试公平锁、非公平锁：

```java

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.List;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class FairAndUnfairTest {

    private static Lock fairLock = new ReentrantLock2(true);

    private static Lock unfairLock = new ReentrantLock2(false);

    public static void main(String[] args) {

        fair();

//        unfair();

    }

    public static void fair() {

        testLock(fairLock);

    }

    public static void unfair() {

        testLock(unfairLock);

    }

    private static void testLock(Lock lock) {

        for (int i=0;i<5;i++) {

            new Job(lock).start();

        }

    }

    private static class Job extends Thread {

        private Lock lock;

        public Job(Lock lock) {

            this.lock = lock;

        }

        public void run() {

            System.out.println("currentThread："+Thread.currentThread().getName());

            Collection<Thread> list = ((ReentrantLock2)lock).getQueuedThreads();

            for (Thread thread : list) {

                System.out.println("queueThread: "+thread.getName());

            }

        }

    }

    private static class ReentrantLock2 extends ReentrantLock {

        public ReentrantLock2(boolean fair) {

            super(fair);

        }

        public Collection<Thread> getQueuedThreads() {

            List<Thread> arrayList = new ArrayList<Thread>(super.

                    getQueuedThreads());

            Collections.reverse(arrayList);

            return arrayList;

        }

    }

}

```

公平性锁每次都是从同步队列中的第一个节点获取到锁，而非公平性锁出现了一个线程连续获取锁的情况。

为什么会出现线程连续获取锁的情况呢？

回顾nonfairTryAcquire(int acquires)方法，当一个线程请求锁时，只要获取了同步状态即成功获取锁。

在这个前提下，刚释放锁的线程再次获取同步状态的几率会非常大，使得其他线程只能在同步队列中等待。

非公平性锁可能使线程“饥饿”，为什么它又被设定成默认的实现呢？再次观察上表的结果，如果把每次不同线程获取到锁定义为1次切换，公平性锁在测试中进行了10次切换，而非公平性锁只有5次切换，这说明非公平性锁的开销更小。

通过vmstat统计测试运行时系统线程上下文切换的次数。

在测试中公平性锁与非公平性锁相比，总耗时是其94.3倍，总切换次数是其133倍。

可以看出，公平性锁保证了锁的获取按照FIFO原则，而代价是进行大量的线程切换。

非公平性锁虽然可能造成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

## 5.4 读写锁

之前提到锁（如Mutex和ReentrantLock）基本都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。

读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

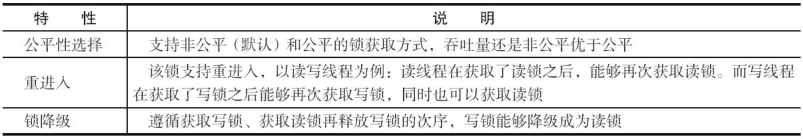
除了保证写操作对读操作的可见性以及并发性的提升之外，读写锁能够简化读写交互场景的编程方式。

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。

Java并发包提供读写锁的实现是ReentrantReadWriteLock。

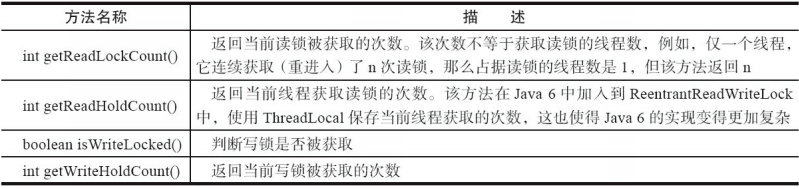
ReentrantReadWriteLock的特性



### 5.4.1 读写锁的接口与示例

ReadWriteLock仅定义了获取读锁和写锁的两个方法，即readLock()方法和writeLock()方法，而其实现ReentrantReadWriteLock，除了接口方法之外，还提供了一些便于外界监控其内部工作状态的方法：

ReentrantReadWriteLock展示内部工作状态的方法



通过一个缓存示例说明读写锁的使用方式：

```java

```

Cache组合一个非线程安全的HashMap作为缓存的实现，同时使用读写锁的读锁和写锁来保证Cache是线程安全的。

在读操作get(String key)方法中，需要获取读锁，这使得并发访问该方法时不会被阻塞。

写操作put(String key,Object value)方法和clear()方法，在更新HashMap时必须提前获取写锁，当获取写锁后，其他线程对于读锁和写锁的获取均被阻塞，而只有写锁被释放之后，其他读写操作才能继续。

Cache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，同时简化了编程方式。

### 5.4.2 读写锁的实现分析

分析ReentrantReadWriteLock的实现，主要包括：

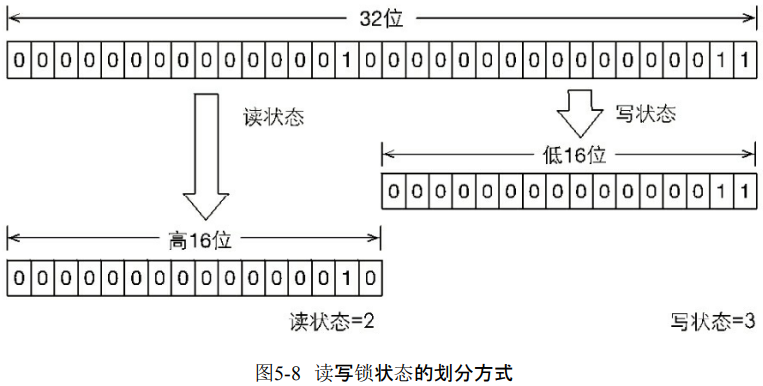
读写状态的设计、写锁的获取与释放、读锁的获取与释放以及锁降级（以下没有特别说明读写锁均可认为是ReentrantReadWriteLock）

1.读写状态的设计

读写锁同样依赖自定义同步器来实现同步功能，而读写状态就是其同步器的同步状态。

回想ReentrantLock中自定义同步器的实现，同步状态表示锁被一个线程重复获取的次数，而读写锁的自定义同步器需要在同步状态（一个整型变量）上维护多个读线程和一个写线程的状态，使得该状态的设计成为读写锁实现的关键。

如果在一个整型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写，划分方式：



当前同步状态表示一个线程已经获取了写锁，且重进入了两次，同时也连续获取了两次读锁。

读写锁是如何迅速确定读和写各自的状态呢？答案是通过位运算。

假设当前同步状态值为S，写状态等于S&0x0000FFFF（将高16位全部抹去），读状态等于S>>>16（无符号补0右移16位）。

当写状态增加1时，等于S+1，当读状态增加1时，等于S+(1<<16)，也就是S+0x00010000。

根据状态的划分能得出一个推论：

S不等于0时，当写状态（S&0x0000FFFF）等于0时，则读状态（S>>>16）大于0，即读锁已被获取。

2.写锁的获取与释放

写锁是一个支持重进入的排它锁。

如果当前线程已经获取了写锁，则增加写状态。

如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程，则当前线程进入等待状态。

```java

package java.util.concurrent.locks;

public class ReentrantReadWriteLock

        implements ReadWriteLock, java.io.Serializable {

    private final ReentrantReadWriteLock.ReadLock readerLock;

    private final ReentrantReadWriteLock.WriteLock writerLock;

    final Sync sync;

    protected final boolean tryAcquire(int acquires) {

        /\*

         \* Walkthrough:

         \* 1. If read count nonzero or write count nonzero

         \*    and owner is a different thread, fail.

         \* 2. If count would saturate, fail. (This can only

         \*    happen if count is already nonzero.)

         \* 3. Otherwise, this thread is eligible for lock if

         \*    it is either a reentrant acquire or

         \*    queue policy allows it. If so, update state

         \*    and set owner.

         \*/

        Thread current = Thread.currentThread();

        int c = getState();

        int w = exclusiveCount(c);

        if (c != 0) {

            // (Note: if c != 0 and w == 0 then shared count != 0)

            if (w == 0 || current != getExclusiveOwnerThread())

                return false;

            if (w + exclusiveCount(acquires) > MAX\_COUNT)

                throw new Error("Maximum lock count exceeded");

            // Reentrant acquire

            setState(c + acquires);

            return true;

        }

        if (writerShouldBlock() ||

            !compareAndSetState(c, c + acquires))

            return false;

        setExclusiveOwnerThread(current);

        return true;

    }

}

```

该方法除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。

如果存在读锁，则写锁不能被获取，原因在于：

读写锁要确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。

因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。

写锁的释放与ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续读写线程可见。

3.读锁的获取与释放

读锁是一个支持重进入的共享锁。

它能够被多个线程同时获取，在没有其他写线程访问（或者写状态为0）时，读锁总会被成功地获取，而所做的也只是（线程安全的）增加读状态。

如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态。

获取读锁的实现从Java 5到Java 6变得复杂许多，主要原因是新增了一些功能，例如getReadHoldCount()方法，作用是返回当前线程获取读锁的次数。

读状态是所有线程获取读锁次数的总和，而每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，由线程自身维护，这使获取读锁的实现变得复杂。

因此，这里将获取读锁的代码做了删减，保留必要的部分：

```java

protected final int tryAcquireShared(int unused) {

    for (;;) {

        int c = getState();

        int nextc = c + (1 << 16);

        if (nextc < c)

            throw new Error("Maximum lock count exceeded");

        if (exclusiveCount(c) != 0 && owner != Thread.currentThread())

            return -1;

        if (compareAndSetState(c, nextc))

            return 1;

    }

}

```

在tryAcquireShared(int unused)方法中，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。

如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全，依靠CAS保证）增加读状态，成功获取读锁。

读锁的每次释放（线程安全的，可能有多个读线程同时释放读锁）均减少读状态，减少的值是（1<<16）。

4.锁降级

锁降级指的是写锁降级成为读锁。

如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。

锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

接下来看一个锁降级的示例。

因为数据不常变化，所以多个线程可以并发地进行数据处理，当数据变更后，如果当前线程感知到数据变化，则进行数据的准备工作，同时其他处理线程被阻塞，直到当前线程完成数据的准备工作：

```java

public void processData() {

    readLock.lock();

    if (!update) {

        // 必须先释放读锁

        readLock.unlock();

        // 锁降级从写锁获取到开始

        writeLock.lock();

        try {

            if (!update) {

                // 准备数据的流程（略）

                update = true;

            }

            readLock.lock();

        } finally {

            writeLock.unlock();

        }

        // 锁降级完成，写锁降级为读锁

    }

    try {

    // 使用数据的流程（略）

    } finally {

        readLock.unlock();

    }

}

```

上述示例中，当数据发生变更后，update变量（布尔类型且volatile修饰）被设置为false，此时所有访问processData()方法的线程都能够感知到变化，但只有一个线程能够获取到写锁，其他线程会被阻塞在读锁和写锁的lock()方法上。

当前线程获取写锁完成数据准备之后，再获取读锁，随后释放写锁，完成锁降级。

锁降级中读锁的获取是否必要呢？答案是必要的。

主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。

如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

RentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。

目的也是保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

## 5.5 LockSupport工具

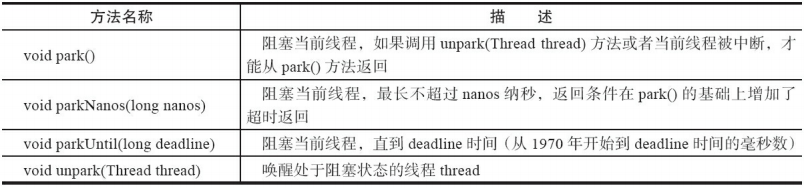
当需要阻塞或唤醒一个线程的时候，都会使用LockSupport工具类来完成相应工作。

LockSupport定义了一组的公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而LockSupport也成为构建同步组件的基础工具。

LockSupport定义了一组以park开头的方法用来阻塞当前线程，以及unpark(Thread thread)方法来唤醒一个被阻塞的线程。

Park有停车的意思，假设线程为车辆，那么park方法代表着停车，而unpark方法则是指车辆启动离开。

LockSupport提供的阻塞和唤醒方法



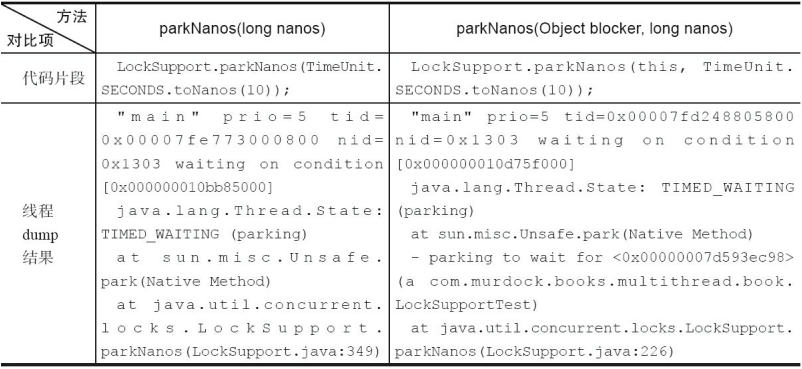
在Java 6中，LockSupport增加了park(Object blocker)、parkNanos(Object blocker,long nanos)和parkUntil(Object blocker,long deadline)3个方法，用于实现阻塞当前线程的功能，其中参数blocker是用来标识当前线程在等待的对象（以下称为阻塞对象），该对象主要用于问题排查和系统监控。

从线程dump结果可以看出，有阻塞对象的parkNanos方法能够传递给开发人员更多的现场信息。

这是由于在Java 5之前，当线程阻塞（使用synchronized关键字）在一个对象上时，通过线程dump能够查看到该线程的阻塞对象，方便问题定位，而Java 5推出的Lock等并发工具时却遗漏了这一点，致使在线程dump时无法提供阻塞对象的信息。

因此，在Java 6中，LockSupport新增了上述3个含有阻塞对象的park方法，用以替代原有的park方法。

Blocker在线程dump中的作用

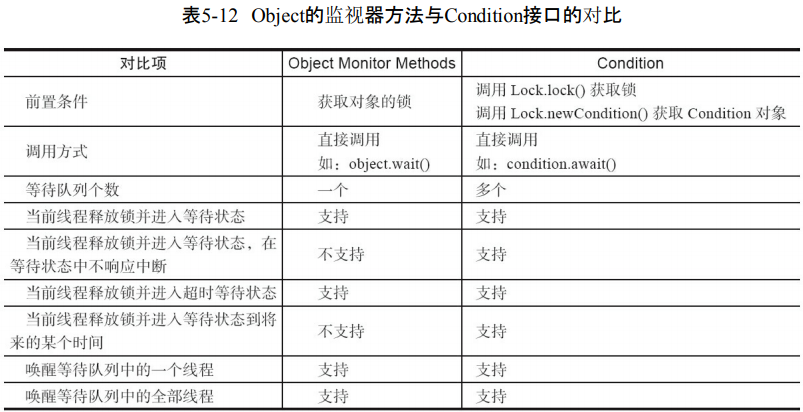


## 5.6 Condition接口

任意一个Java对象，都拥有一组监视器方法（定义在java.lang.Object上），主要包括wait()、wait(long timeout)、notify()以及notifyAll()方法，这些方法与synchronized同步关键字配合，可以实现等待/通知模式。

Condition接口也提供了类似Object的监视器方法，与Lock配合可以实现等待/通知模式，但是这两者在使用方式以及功能特性上还是有差别的。

通过对比Object的监视器方法和Condition接口，可以更详细地了解Condition的特性，对比项与结果：



### 5.6.1 Condition接口与示例

Condition定义了等待/通知两种类型的方法，当前线程调用这些方法时，需要提前获取到Condition对象关联的锁。

Condition对象是由Lock对象（调用Lock对象的newCondition()方法）创建出来的，换句话说，Condition是依赖Lock对象的。

Condition的使用方式比较简单，需要注意在调用方法前获取锁

```java

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ConditionUseCase {

    Lock lock = new ReentrantLock();

    Condition condition = lock.newCondition();

    public void conditionWait() throws InterruptedException {

        lock.lock();

        try {

            condition.await();

        } finally {

            lock.unlock();

        }

    }

    public void conditionSignal() throws InterruptedException {

        lock.lock();

        try {

            condition.signal();

        } finally {

            lock.unlock();

        }

    }

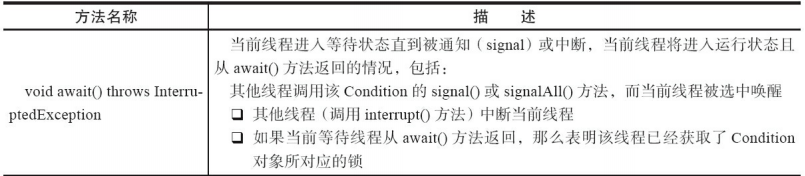
}

```

一般都会将Condition对象作为成员变量。

当调用await()方法后，当前线程会释放锁并在此等待，而其他线程调用Condition对象的signal()方法，通知当前线程后，当前线程才从await()方法返回，并且在返回前已经获取了锁。

Condition的（部分）方法以及描述





获取一个Condition必须通过Lock的newCondition()方法。

下面通过一个有界队列的示例来深入了解Condition的使用方式。

有界队列是一种特殊的队列，当队列为空时，队列的获取操作将会阻塞获取线程，直到队列中有新增元素，当队列已满时，队列的插入操作将会阻塞插入线程，直到队列出现“空位”

```java

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class BoundedQueue<T> {

    private Object[] items;

    // 添加的下标，删除的下标和数组当前数量

    private int addIndex, removeIndex, count;

    private Lock lock = new ReentrantLock();

    private Condition notEmpty = lock.newCondition();

    private Condition notFull = lock.newCondition();

    public BoundedQueue(int size) {

        items = new Object[size];

    }

    // 添加一个元素，如果数组满，则添加线程进入等待状态，直到有"空位"

    public void add(T t) throws InterruptedException {

        lock.lock();

        try {

            while (count == items.length)

                notFull.await();

            items[addIndex] = t;

            if (++addIndex == items.length)

                addIndex = 0;

            ++count;

            notEmpty.signal();

        } finally {

            lock.unlock();

        }

    }

    // 由头部删除一个元素，如果数组空，则删除线程进入等待状态，直到有新添加元素

    @SuppressWarnings("unchecked")

    public T remove() throws InterruptedException {

        lock.lock();

        try {

            while (count == 0)

                notEmpty.await();

            Object x = items[removeIndex];

            if (++removeIndex == items.length)

                removeIndex = 0;

            --count;

            notFull.signal();

            return (T) x;

        } finally {

            lock.unlock();

        }

    }

}

```

上述示例中，BoundedQueue通过add(T t)方法添加一个元素，通过remove()方法移出一个元素。

以添加方法为例。

首先需要获得锁，目的是确保数组修改的可见性和排他性。

当数组数量等于数组长度时，表示数组已满，则调用notFull.await()，当前线程随之释放锁并进入等待状态。

如果数组数量不等于数组长度，表示数组未满，则添加元素到数组中，同时通知等待在notEmpty上的线程，数组中已经有新元素可以获取。

在添加和删除方法中使用while循环而非if判断，目的是防止过早或意外的通知，只有条件符合才能够退出循环。

回想之前提到的等待/通知的经典范式，二者是非常类似的。

### 5.6.2 Condition的实现分析

ConditionObject是同步器AbstractQueuedSynchronizer的内部类，因为Condition的操作需要获取相关联的锁，所以作为同步器的内部类也较为合理。

每个Condition对象都包含着一个队列（以下称为等待队列），该队列是Condition对象实现等待/通知功能的关键。

下面将分析Condition的实现，主要包括：等待队列、等待和通知，下面提到的Condition如果不加说明均指的是ConditionObject。

1.等待队列

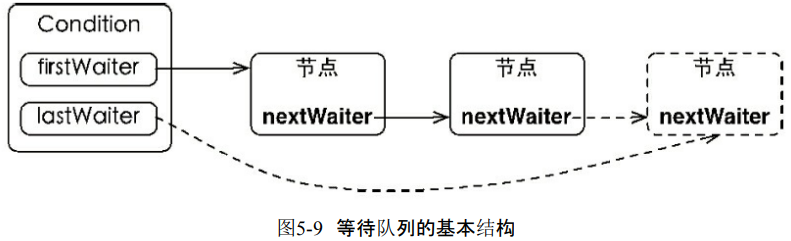
等待队列是一个FIFO的队列，在队列中的每个节点都包含了一个线程引用，该线程就是在Condition对象上等待的线程，如果一个线程调用了Condition.await()方法，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态。

事实上，节点的定义复用了同步器中节点的定义，也就是说，同步队列和等待队列中节点类型都是同步器的静态内部类AbstractQueuedSynchronizer.Node。

一个Condition包含一个等待队列，Condition拥有首节点（firstWaiter）和尾节点（lastWaiter）。

当前线程调用Condition.await()方法，将会以当前线程构造节点，并将节点从尾部加入等待队列。

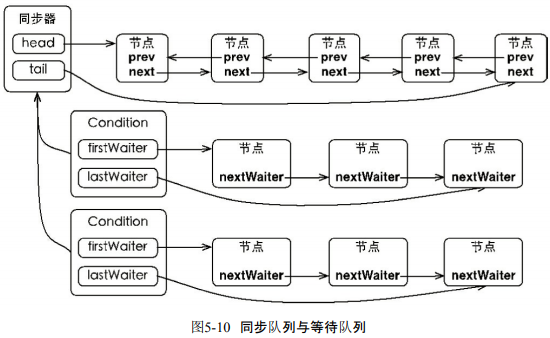
等待队列的基本结构



如图所示，Condition拥有首尾节点的引用，而新增节点只需要将原有的尾节点nextWaiter指向它，并且更新尾节点即可。

上述节点引用更新的过程并没有使用CAS保证，原因在于调用await()方法的线程必定是获取了锁的线程，也就是说该过程是由锁来保证线程安全的。

在Object的监视器模型上，一个对象拥有一个同步队列和等待队列，而并发包中的Lock（更确切地说是同步器）拥有一个同步队列和多个等待队列，其对应关系：



Condition的实现是同步器的内部类，因此每个Condition实例都能够访问同步器提供的方法，相当于每个Condition都拥有所属同步器的引用。

2.等待

调用Condition的await()方法（或者以await开头的方法），会使当前线程进入等待队列并释放锁，同时线程状态变为等待状态。

当从await()方法返回时，当前线程一定获取了Condition相关联的锁。

如果从队列（同步队列和等待队列）的角度看await()方法，当调用await()方法时，相当于同步队列的首节点（获取了锁的节点）移动到Condition的等待队列中。

ConditionObject的await方法

```java

package java.util.concurrent.locks;

public abstract class AbstractQueuedSynchronizer

    extends AbstractOwnableSynchronizer

    implements java.io.Serializable {

    public class ConditionObject implements Condition, java.io.Serializable {

        /\*\* First node of condition queue. \*/

        private transient Node firstWaiter;

        /\*\* Last node of condition queue. \*/

        private transient Node lastWaiter;

        public final void await() throws InterruptedException {

            if (Thread.interrupted())

                throw new InterruptedException();

            // 当前线程加入等待队列

            Node node = addConditionWaiter();

            // 释放同步状态，也就是释放锁

            int savedState = fullyRelease(node);

            int interruptMode = 0;

            while (!isOnSyncQueue(node)) {

                LockSupport.park(this);

                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

                    break;

            }

            if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

                interruptMode = REINTERRUPT;

            if (node.nextWaiter != null) // clean up if cancelled

                unlinkCancelledWaiters();

            if (interruptMode != 0)

                reportInterruptAfterWait(interruptMode);

        }

    }

}

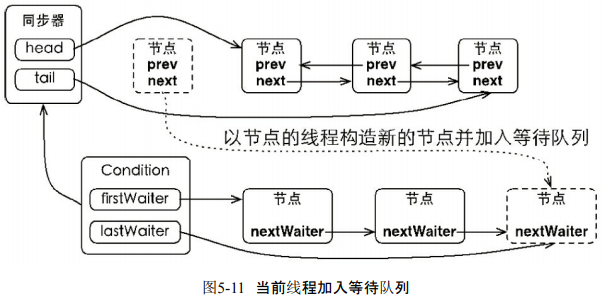
```

调用该方法的线程成功获取了锁的线程，也就是同步队列中的首节点，该方法会将当前线程构造成节点并加入等待队列中，然后释放同步状态，唤醒同步队列中的后继节点，然后当前线程会进入等待状态。

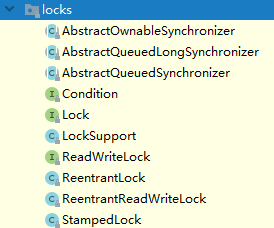
当等待队列中的节点被唤醒，则唤醒节点的线程开始尝试获取同步状态。

如果不是通过其他线程调用Condition.signal()方法唤醒，而是对等待线程进行中断，则会抛出InterruptedException。

如果从队列的角度去看，当前线程加入Condition的等待队列，该过程如图5-11示。



如图所示，同步队列的首节点并不会直接加入等待队列，而是通过addConditionWaiter()方法把当前线程构造成一个新的节点并将其加入等待队列中。



3.通知

调用Condition的signal()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在

唤醒节点之前，会将节点移到同步队列中。

Condition的signal()方法

```java

public final void signal() {

    if (!isHeldExclusively())

        throw new IllegalMonitorStateException();

    Node first = firstWaiter;

    if (first != null)

        doSignal(first);

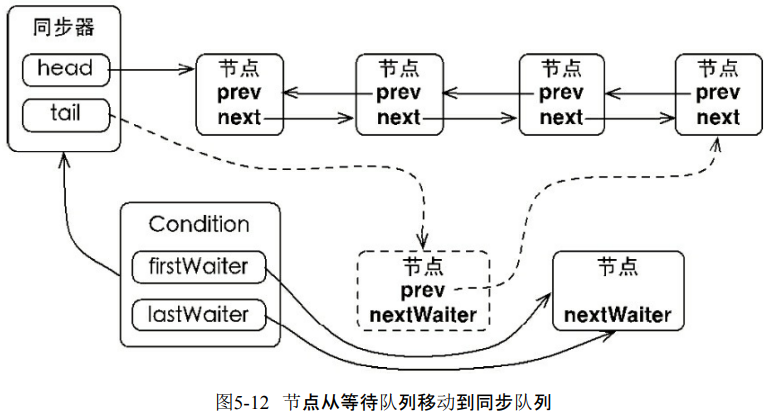
}

```

调用该方法的前置条件是当前线程必须获取了锁，可以看到signal()方法进行了isHeldExclusively()检查，也就是当前线程必须是获取了锁的线程。

接着获取等待队列的首节点，将其移动到同步队列并使用LockSupport唤醒节点中的线程。

节点从等待队列移动到同步队列的过程如图



通过调用同步器的enq(Node node)方法，等待队列中的头节点线程安全地移动到同步队列。

当节点移动到同步队列后，当前线程再使用LockSupport唤醒该节点的线程。

被唤醒后的线程，将从await()方法中的while循环中退出（isOnSyncQueue(Node node)方法返回true，节点已经在同步队列中），进而调用同步器的acquireQueued()方法加入到获取同步状态的竞争中。

成功获取同步状态（或者说锁）之后，被唤醒的线程将从先前调用的await()方法返回，此时该线程已经成功地获取了锁。

Condition的signalAll()方法，相当于对等待队列中的每个节点均执行一次signal()方法，效果就是将等待队列中所有节点全部移动到同步队列中，并唤醒每个节点的线程。

## 5.7 本章小结

本章介绍了Java并发包中与锁相关的API和组件。

通过示例讲述了这些API和组件的使用方式以及需要注意的地方，并在此基础上详细地剖析了队列同步器、重入锁、读写锁以及Condition等API和组件的实现细节，只有理解这些API和组件的实现细节才能够更加准确地运用它们。