JUC多线程 (二)

学习目标：

- 掌握volatile解决内存可见性的使用
- 了解CAS原子操作
- 掌握JUC的atomic原子操作包的使用
- 了解AQS同步队列的作用
- 了解JUC的锁的基本概念
- 掌握ReentrantLock和ReentrantReadWriteLock的使用
- 掌握Condition的使用

5 Volatile

通过前面内容我们了解了synchronized，虽然JVM对它做了很多优化，但是它还是一个重量级的锁。而接下来要介绍的volatile则是轻量级的synchronized。如果一个变量使用volatile，则它比使用synchronized的成本更加低，因为它不会引起线程上下文的切换和调度。

Java语言规范对volatile的定义如下：

> Java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。

通俗点讲就是说一个变量如果用volatile修饰了，则Java可以确保所有线程看到这个变量的值是一致的，如果某个线程对volatile修饰的共享变量进行更新，那么其他线程可以立马看到这个更新，这就是内存可见性。

volatile虽然看起来比较简单，使用起来无非就是在一个变量前面加上volatile即可，但是要用好并不容易。

5.1 解决内存可见性问题

在可见性问题案例中进行如下修改，添加volatile关键词：

```
private volatile boolean flag = true;
```

线程写Volatile变量的过程：

1. 改变线程本地内存中Volatile变量副本的值；
2. 将改变后的副本的值从本地内存刷新到主内存

线程读Volatile变量的过程：

1. 从主内存中读取Volatile变量的最新值到线程的本地内存中
2. 从本地内存中读取Volatile变量的副本

Volatile实现内存可见性原理：

写操作时，通过在写操作指令后加入一条store屏障指令，让本地内存中变量的值能够刷新到主内存中

读操作时，通过在读操作前加入一条load屏障指令，及时读取到变量在主内存的值

PS：内存屏障（Memory Barrier）是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。Java编译器也会根据内存屏障的规则禁止重排序

volatile的底层实现是通过插入内存屏障，但是对于编译器来说，发现一个最优布置来最小化插入内存屏障的总数几乎是不可能的，所以，JMM采用了保守策略。如下：

- StoreStore屏障可以保证在volatile写之前，其前面的所有普通写操作都已经刷新到主内存中。
- StoreLoad屏障的作用是避免volatile写与后面可能有的volatile读/写操作重排序。
- LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。
- LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。

5.2 原子性的问题

虽然Volatile 关键字可以让变量在多个线程之间可见，但是Volatile不具备原子性。

```
public class Demo3Volatile {

    public static void main(String[] args) throws
InterruptedException {
        VolatileDemo demo = new VolatileDemo();
```

```
            Thread t = new Thread(demo);
            t.start();
        }

        Thread.sleep(1000);
        System.out.println(demo.count);
    }

    static class VolatileDemo implements Runnable {
        public volatile int count;
        //public volatile AtomicInteger count = new
AtomicInteger(0);

        public void run() {
            addCount();
        }

        public void addCount() {
            for (int i = 0; i < 10000; i++) {
                count++;
            }
        }
    }
}
```

以上出现原子性问题的原因是count++并不是原子性操作。

count = 5 开始，流程分析：

1. 线程1读取count的值为5
2. 线程2读取count的值为5
3. 线程2加1操作
4. 线程2最新count的值为6
5. 线程2写入值到主内存的最新值为6

这个时候，线程1的count为5，线程2的count为6

如果切换到线程1执行，那么线程1得到的结果是6，写入到主内存的值还是6

现在的情况是对count进行了两次加1操作，但是主内存实际上只是加1一次

1. 使用synchronized
2. 使用ReentrantLock（可重入锁）
3. 使用AtomicInteger（原子操作）

使用synchronized

```
public synchronized void addCount() {
    for (int i = 0; i < 10000; i++) {
        count++;
    }
}
```

使用ReentrantLock（可重入锁）

```
//可重入锁
private Lock lock = new ReentrantLock();

public void addCount() {
    for (int i = 0; i < 10000; i++) {
        lock.lock();
        count++;
        lock.unlock();
    }
}
```

使用AtomicInteger（原子操作）

```
public static AtomicInteger count = new AtomicInteger(0);
public void addCount() {
    for (int i = 0; i < 10000; i++) {
        //count++;
        count.incrementAndGet();
    }
}
```

5.3 Volatile 适合使用场景

a）对变量的写入操作不依赖其当前值

满足：boolean变量、直接赋值的变量等

b）该变量没有包含在具有其他变量的不变式中

不满足：不变式 low<up

总结：变量真正独立于其他变量和自己以前的值，在单独使用的时候，适合用volatile

5.4 synchronized和volatile比较

a）volatile不需要加锁，比synchronized更轻便，不会阻塞线程

b）synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件（**变量真正独立于其他变量和自己以前的值**） 在某些情况下可以使用 volatile 代替 synchronized 来优化代码提升效率。

6 J.U.C之CAS

J.U.C 即 java.util.concurrent，是 JSR 166 标准规范的一个实现； JSR 166 以及 J.U.C 包的作者是 Doug Lea 。

J.U.C 框架是 Java 5 中引入的，而我们最熟悉的线程池机制就在这个包，J.U.C 框架包含的内容有：

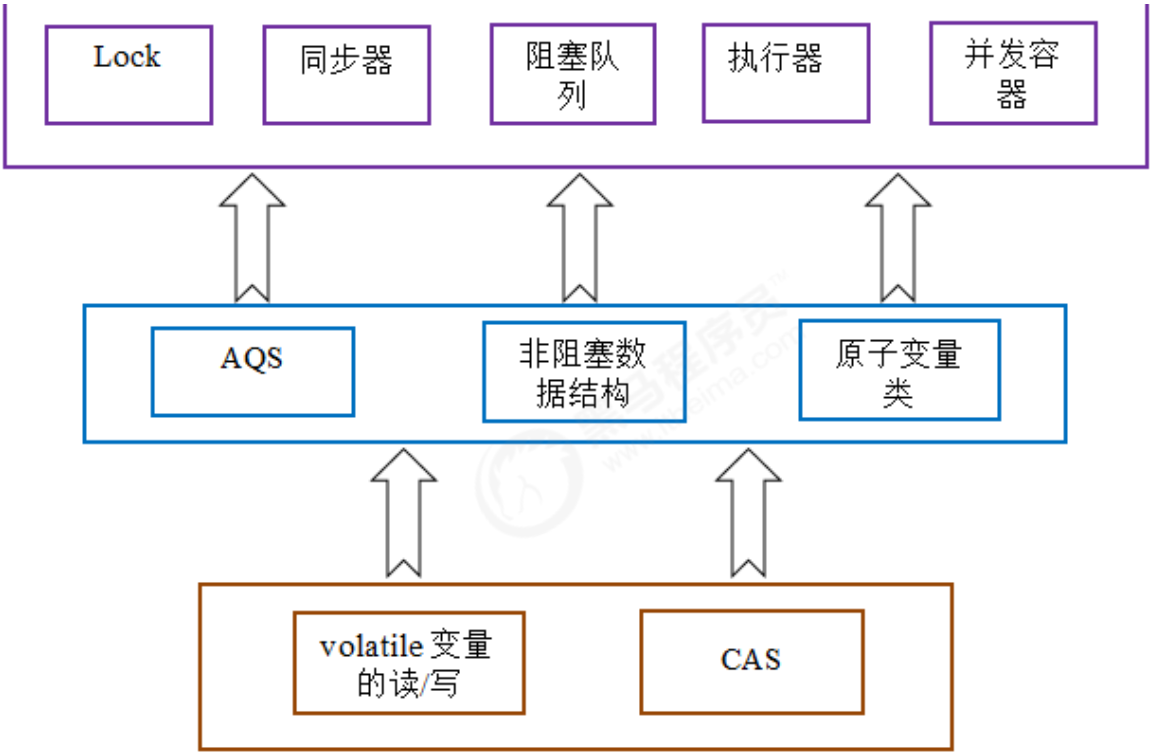
- AbstractQueuedSynchronizer（AQS框架），J.U.C 中实现锁和同步机制的基础；
- Locks & Condition（锁和条件变量），比 synchronized、wait、notify 更细粒度的锁机制；
- Executor 框架（线程池、Callable、Future），任务的执行和调度框架；
- Synchronizers（同步器），主要用于协助线程同步，有 CountDownLatch、CyclicBarrier、Semaphore、Exchanger；
- Atomic Variables（原子变量），方便程序员在多线程环境下，无锁的进行原子操作，核心操作是 CAS 原子操作，所谓的 CAS 操作，即 compare and swap，

(Swap)当前变量，否则不作操作；

- BlockingQueue（阻塞队列），阻塞队列提供了可阻塞的入队和出对操作，如果队列满了，入队操作将阻塞直到有空间可用，如果队列空了，出队操作将阻塞直到有元素可用；
- Concurrent Collections（并发容器），说到并发容器，不得不提同步容器。在JDK1.5 之前，为了线程安全，我们一般都是使用同步容器，同步容器主要的缺点是：对所有容器状态的访问都串行化，严重降低了并发性；某些复合操作，仍然需要加锁来保护；迭代期间，若其它线程并发修改该容器，会抛出ConcurrentModificationException 异常，即快速失败机制；
- Fork/Join 并行计算框架，这块内容是在 JDK1.7 中引入的，可以方便利用多核平台的计算能力，简化并行程序的编写，开发人员仅需关注如何划分任务和组合中间结果；
- TimeUnit 枚举，TimeUnit 是 java.util.concurrent 包下面的一个枚举类，TimeUnit 提供了可读性更好的线程暂停操作，以及方便的时间单位转换方法；

6.1 CAS介绍

CAS，Compare And Swap，即比较并交换。同步组件中大量使用CAS技术实现了Java多线程的并发操作。整个AQS同步组件、Atomic原子类操作等等都是以CAS实现的，甚至ConcurrentHashMap在1.8的版本中也调整为了CAS+Synchronized。可以说CAS是整个JUC的基石。



6.2 CAS原理剖析

再次测试之前Volatile的例子，把循环的次数调整为一亿（保证在一秒之内不能遍历完成，从而测试三种原子操作的性能），我们发现，AtomicInteger原子操作性能最高，他是用的就是CAS。

6.2.2 synchronized同步分析

注意，本小节是解释synchronized性能低效的原因，只要能理解synchronized同步过程其实还需要做很多事，这些逻辑的执行都需要占用资源，从而导致性能较低，是为了对比CAS的高效。这部分分析过于深入JMM底层原理，不适合初级甚至中级程序员学习。

我们之前讲过，synchronized的同步操作主要是monitorenter和monitorexit这两个jvm指令实现的，我们先写一段简单的代码：

```
public class Demo2Synchronized {
    public void test2() {
        synchronized (this) {
        }
    }
}
```

```
javac Demo2Synchronized.java
javap -c Demo2Synchronized.class
```

从结果可以看出，同步代码块是使用monitorenter和monitorexit这两个jvm指令实现的：

```
D:\itcast\class_thread\class_01\src\main\java\com\itheima\demo0525>javap -c SynchronizedDemo.class
Compiled from "SynchronizedDemo.java"
public class com.itheima.demo0525.SynchronizedDemo {
  public com.itheima.demo0525.SynchronizedDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public synchronized void test1();
    Code:
       0: return

  public void test2();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter      monitor进入，获取锁
       4: aload_1
       5: monitorexit       monitor退出，释放锁
       6: goto          14
       9: astore_2
      10: aload_1
      11: monitorexit
      12: aload_2
      13: athrow
      14: return
    Exception table:
       from    to  target type
           4     6     9   any
           9    12     9   any
}
```

monitorenter和monitorexit这两个jvm指令实现锁的使用，主要是基于 Mark Word和、monitor。

Mark Word

Hotspot虚拟机的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。其中Klass Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据，它是synchronized实现轻量级锁和偏向锁的关键。

年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向的时间戳等等。Java对象头一般占有两个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit），但是如果对象是数组类型，则需要三个机器码，因为JVM虚拟机可以通过Java对象的元数据信息确定Java对象的大小，但是无法从数组的元数据来确认数组的大小，所以用一块来记录数组长度。下图是Java对象头的存储结构（32位虚拟机）：

25Bit	4bit	1bit	2bit
对象的hashCode	对象的分代年龄	是否是偏向锁	锁标志位

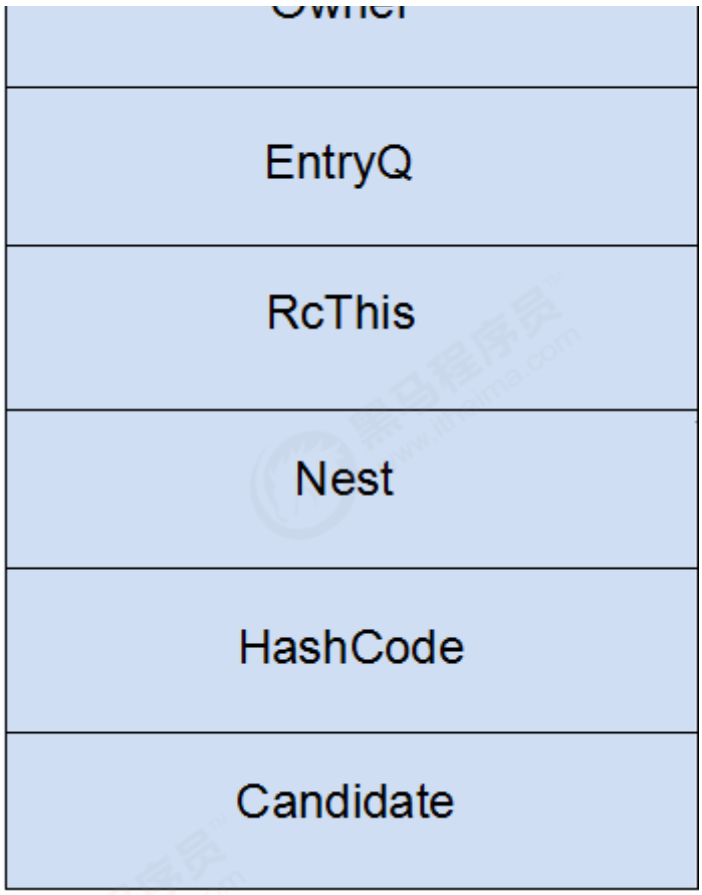
对象头信息是与对象自身定义的数据无关的额外存储成本，但是考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据，它会根据对象的状态复用自己的存储空间，也就是说，Mark Word会随着程序的运行发生变化，变化状态如下（32位虚拟机）：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象hashcode、对象分代年龄				01
轻量级锁	指向锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

- **monitor**

  什么是Monitor？我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象。与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中，每一个Java对象都带了一把看不见的锁，它叫做内部锁或者Monitor锁。

  Monitor 是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联（对象头的MarkWord中的LockWord指向monitor的起始地址），同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。其结构如下：



- **Owner**：初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；
- **EntryQ**:关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程。
- **RcThis**:表示blocked或waiting在该monitor record上的所有线程的个数。
- **Nest**:用来实现重入锁的计数。
- **HashCode**:保存从对象头拷贝过来的HashCode值（可能还包含GC age）。
- **Candidate**:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁。

6.2.3 CAS原理

在上一部分，我们介绍了synchronized底层做了大量的工作，才实现同步，而同步保证了原子操作。但是不可避免的是性能较低。CAS是如何提高性能的呢?

B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。如果CAS操作失败，通过自旋的方式等待并再次尝试，直到成功。

CAS在 **先比较后修改** 这个CAS过程中，根本没有获取锁，释放锁的操作，是硬件层面的原子操作，跟JMM内存模型没有关系。大家可以理解为直接使用其他的语言，在JVM虚拟机之外直接操作计算机硬件，正因为如此，对比synchronized的同步，少了很多的逻辑步骤，使得性能大为提高。

JUC下的atomic类都是通过CAS来实现的，下面就是一个AtomicInteger原子操作类的例子，在其中使用了Unsafe unsafe = Unsafe.getUnsafe()。Unsafe 是CAS的核心类，它提供了硬件级别的原子操作。

```
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}
//操作的值也进行了volatile修饰，保证内存可见性
private volatile int value;
```

继续查看AtomicInteger的addAndGet()方法：

```
public final int addAndGet(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta) + delta;
}

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 +
var4));

    return var5;
}
```

方法为native方法，有四个参数，分别代表：对象、对象的地址、预期值、修改值。：

```
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

Unsafe 是一个比较危险的类，主要是用于执行低级别、不安全的方法集合。尽管这个类和所有的方法都是公开的（public），但是这个类的使用仍然受限，你无法在自己的java程序中直接使用该类，因为只有授信的代码才能获得该类的实例。可是为什么Unsafe的native方法就可以保证是原子操作呢?

6.3 native关键词

前面提到了sun.misc.Unsafe这个类，里面的方法使用native关键词声明本地方法，为什么要用native?

Java无法直接访问底层操作系统，但有能力调用其他语言编写的函数or方法，是通过JNI(Java Native Interfface)实现。使用时，通过native关键字告诉JVM这个方法是在外部定义的。但JVM也不知道去哪找这个原生方法，此时需要通过javah命令生成.h文件。

示例步骤(c语言为例):

1. javac生成.class文件，比如javac NativePeer.java
2. javah生成.h文件，比如javah NativePeer
3. 编写c语言文件，在其中include进上一步生成的.h文件，然后实现其中声明而未实现的函数
4. 生成dll共享库，然后Java程序load库，调用即可

   native可以和任何除abstract外的关键字连用，这也说明了这些方法是有实体的，并且能够和其他Java方法一样，拥有各种Java的特性。

   > native方法有效地扩充了jvm，实际上我们所用的很多代码已经涉及到这种方法了，通过非常简洁的接口帮我们实现Java以外的工作。

native优势:

的很多，纯Java实现可能会导致效率不达标，或者可读性可差。

2. Java毕竟不是一个完整的系统，它经常需要一些底层的支持，通过JNI和native method我们就可以实现jre与底层的交互，得到强大的底层操作系统的支持，使用一些Java本身没有封装的操作系统的特性。

6.4 多CPU的CAS处理

CAS可以保证一次的读-改-写操作是原子操作，在单处理器上该操作容易实现，但是在多处理器上实现就有点儿复杂了。CPU提供了两种方法来实现多处理器的原子操作：总线加锁或者缓存加锁。

- **总线加锁**：总线加锁就是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使用共享内存。但是这种处理方式显得有点儿霸道，不厚道，他把CPU和内存之间的通信锁住了，在锁定期间，其他处理器都不能其他内存地址的数据，其开销有点儿大。
- **缓存加锁**：其实针对于上面那种情况我们只需要保证在同一时刻对某个内存地址的操作是原子性的即可。缓存加锁就是缓存在内存区域的数据如果在加锁期间，当它执行锁操作写回内存时，处理器不在输出LOCK#信号，而是修改内部的内存地址，利用缓存一致性协议来保证原子性。缓存一致性机制可以保证**同一个内存区域的数据仅能被一个处理器修改**，也就是说当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行。

6.4 CAS缺陷

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方法：循环时间太长、只能保证一个共享变量原子操作、ABA问题。

- **循环时间太长**

  如果CAS一直不成功呢？这种情况绝对有可能发生，如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。在JUC中有些地方就限制了CAS自旋的次数，例如BlockingQueue的SynchronousQueue。

- **只能保证一个共享变量原子操作**

  看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了。

CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，这就是所谓的ABA问题。对于ABA问题其解决方案是加上版本号，即在每个变量都加上一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A。

> CAS的ABA隐患问题，Java提供了AtomicStampedReference来解决。AtomicStampedReference通过包装[E,Integer]的元组来对对象标记版本戳stamp，从而避免ABA问题。对于上面的案例应该线程1会失败。

下面我们将通过一个例子可以可以看到AtomicStampedReference和AtomicInteger的区别。我们定义两个线程，线程1负责将100 —> 110 —> 100，线程2执行 100 —>120，看两者之间的区别。

```
public class Demo4ABA {

    private static AtomicInteger ai = new AtomicInteger(100);
    private static AtomicStampedReference air = new
AtomicStampedReference(100, 1);

    //ABA问题演示：
    //1. 线程1先对数据进行修改 A-B-A过程
    //2. 线程2也对数据进行修改 A-C的过程
    public static void main(String[] args) throws
InterruptedException {

        // AtomicInteger可以看到不会有任何限制随便改
        // 线程2修改的时候也不可能知道要A-C 的时候，A是原来的A还是修改之后的A
        Thread at1 = new Thread(new Runnable() {
            public void run() {
                ai.compareAndSet(100, 110);
                ai.compareAndSet(110, 100);
            }
        });

        Thread at2 = new Thread(new Runnable() {
            public void run() {
                try {
                    //为了让线程1先执行完，等一会
                    TimeUnit.MILLISECONDS.sleep(100);
```

```
                }
                System.out.println("AtomicInteger:" +
ai.compareAndSet(100, 120));
                System.out.println("执行结果: " + ai.get());
            }
        });

        at1.start();
        at2.start();

        //顺序执行，AtomicInteger案例先执行
        at1.join();
        at2.join();

        //AtomicStampedReference可以看到每次修改都需要设置标识Stamp，相
当于进行了1A-2B-3A的操作
        //线程2进行操作的时候，虽然数值都一样，但是可以根据标识很容易的知道A
是以前的1A，还是现在的3A
        Thread tsf1 = new Thread(new Runnable() {
            public void run() {
                try {
                    TimeUnit.MILLISECONDS.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 预期引用：100，更新后的引用：110，预期标识getStamp()
更新后的标识getStamp() + 1
                air.compareAndSet(100, 110, air.getStamp(),
air.getStamp() + 1);
                air.compareAndSet(110, 100, air.getStamp(),
air.getStamp() + 1);
            }
        });

        Thread tsf2 = new Thread(new Runnable() {
            public void run() {

                //tsf2先获取stamp，导致预期时间戳不一致
                int stamp = air.getStamp();

                try {
                    TimeUnit.MILLISECONDS.sleep(100);      //线程
tsf1执行完
```

```
                }
                System.out.println("AtomicStampedReference:" +
air.compareAndSet(100, 120, stamp, stamp + 1));
                int[] stampArr = {stamp + 1};
                System.out.println("执行结果: " +
air.get(stampArr));
            }
        });

        tsf1.start();
        tsf2.start();
    }
```

运行结果充分展示了AtomicInteger的ABA问题和AtomicStampedReference解决ABA问题。

7 J.U.C之atomic包

7.1 atomic包介绍

通过前面CAS的学习，我们了解到AtomicInteger的工作原理，它们的内部都维护者一个对应的基本类型的成员变量value，这个变量是被volatile关键字修饰的，保证多线程环境下看见的是同一个（可见性）。

AtomicInteger在进行一些原子操作的时候，依赖Unsafe类里面的CAS方法，原子操作就是通过自旋方式，不断地使用CAS函数进行尝试直到达到自己的目的。

除了AtomicInteger类以外还有很多其他的类也有类似的功能，在JUC中有一个包java.util.concurrent.atomic存放原子操作的类，atomic里的类主要包括：

- 基本类型
  使用原子的方式更新基本类型

  > AtomicInteger：整形原子类
  > AtomicLong：长整型原子类
  > AtomicBoolean ：布尔型原子类

- 引用类型

> AtomicStampedReference：原子更新引用类型里的字段原子类
> AtomicMarkableReference ： 原子更新带有标记位的引用类型

- 数组类型
  使用原子的方式更新数组里的某个元素

> AtomicIntegerArray：整形数组原子类
> AtomicLongArray：长整形数组原子类
> AtomicReferenceArray ： 引用类型数组原子类

- 对象的属性修改类型

> AtomicIntegerFieldUpdater:原子更新整形字段的更新器
> AtomicLongFieldUpdater：原子更新长整形字段的更新器
> AtomicReferenceFieldUpdater ： 原子更新引用类形字段的更新器

- JDK1.8新增类

> DoubleAdder：双浮点型原子类
> LongAdder：长整型原子类
> DoubleAccumulator：类似DoubleAdder，但要更加灵活(要传入一个函数式接口)
> LongAccumulator：类似LongAdder，但要更加灵活(要传入一个函数式接口)

虽然涉及到的类很多，但是原理和AtomicInteger都是一样，使用CAS进行的原子操作，其方法和使用都是大同小异的。

7.2 基本类型

使用原子的方式更新基本类型

> AtomicInteger：整形原子类
> AtomicLong：长整型原子类
> AtomicBoolean ： 布尔型原子类

AtomicInteger主要API如下：

```
getAndAdd(int)        //增加指定的数据，返回变化前的数据
getAndDecrement()     //减少1，返回减少前的数据
getAndIncrement()     //增加1，返回增加前的数据
getAndSet(int)        //设置指定的数据，返回设置前的数据

addAndGet(int)        //增加指定的数据后返回增加后的数据
decrementAndGet()     //减少1，返回减少后的值
incrementAndGet()     //增加1，返回增加后的值
lazySet(int)          //仅仅当get时才会set

compareAndSet(int, int)//尝试新增后对比，若增加成功则返回true否则返回
false
```

AtomicLong主要API和AtomicInteger，只是类型不是int，而是long

AtomicBoolean主要API如下：

```
compareAndSet(boolean, boolean) //参数1为原始值，参数2为修改的新值，若
修改成功返回true，否则返回false
getAndSet(boolean)// 尝试设置新的boolean值，直到成功为止，返回设置前的数
据
```

7.4 引用类型

> AtomicReference：引用类型原子类
> AtomicStampedRefrence：原子更新引用类型里的字段原子类
> AtomicMarkableReference ： 原子更新带有标记位的引用类型

AtomicReference引用类型和基本类型的作用基本一样，例子如下：

```
public class Demo5AtomicReference {

    public static void main(String[] args) throws
InterruptedException {
        User u1 = new User("张三", 22);
        User u2 = new User("李四", 33);

        AtomicReference ar = new AtomicReference(u1);
        ar.compareAndSet(u1, u2);
```

```
    }

    static class User {
        private String name;
        public volatile int age;

        public User(String name, int age) {
            super();
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

AtomicStampedReference其实它仅仅是在AtomicReference类的再一次包装，里面增加了一层引用和计数器，其实是否为计数器完全由自己控制，大多数我们是让他自增的，你也可以按照自己的方式来标示版本号。案例参考前面的ABA例子

AtomicMarkableReference和AtomicStampedReference功能差不多，区别的是：它描述更加简单的是与否的关系。通常ABA问题只有两种状态，而AtomicStampedReference是多种状态。

```
public class Demo6AtomicMrkableReference {

    public static void main(String[] args) throws
InterruptedException {
        User u1 = new User("张三", 22);
        User u2 = new User("李四", 33);

        //和AtomicStampedReference效果一样，用于解决ABA的
        //区别是表示不是用的版本号，而只有true和false两种状态。相当于未修
改和已修改
        AtomicMarkableReference<User> amr = new
AtomicMarkableReference(u1, true);
        amr.compareAndSet(u1, u2, false, true);

        System.out.println(amr.getReference());

    }

    static class User {
        private String name;
        public volatile int age;

        public User(String name, int age) {
            super();
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }
```

```
        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

7.3 数组类型

使用原子的方式更新数组里的某个元素

> AtomicIntegerArray：整形数组原子类
> AtomicLongArray：长整形数组原子类
> AtomicReferenceArray ：引用类型数组原子类

AtomicIntegerArray主要API如下：

```
addAndGet(int, int)//执行加法，第一个参数为数组的下标，第二个参数为增加的数量，返回增加后的结果
compareAndSet(int, int, int)// 对比修改，参1数组下标，参2原始值，参3修改目标值，成功返回true否则false
decrementAndGet(int)// 参数为数组下标，将数组对应数字减少1，返回减少后的数据
incrementAndGet(int)// 参数为数组下标，将数组对应数字增加1，返回增加后的数据

getAndAdd(int, int)// 和addAndGet类似，区别是返回值是变化前的数据
getAndDecrement(int)// 和decrementAndGet类似，区别是返回变化前的数据
getAndIncrement(int)// 和incrementAndGet类似，区别是返回变化前的数据
getAndSet(int, int)// 将对应下标的数字设置为指定值，第二个参数为设置的值，返回是变化前的数据
```

AtomicIntegerArray主要API和AtomicLongArray，只是类型不是int，而是long

```
public class Demo7AtomicIntegerArray {

    public static void main(String[] args) throws
InterruptedException {
        int[] arr = {1, 2, 3, 4, 5};
        AtomicIntegerArray aia = new AtomicIntegerArray(arr);

        aia.compareAndSet(1, 2, 200);

        System.out.println(aia.toString());
    }
}
```

AtomicReferenceArray 主要API：

```
//参数1：数组下标；
//参数2：修改原始值对比；
//参数3：修改目标值
//修改成功返回true，否则返回false
compareAndSet(int, Object, Object)

//参数1：数组下标
//参数2：修改的目标
//修改成功为止，返回修改前的数据
getAndSet(int, Object)
```

AtomicReferenceArray 案例：

```
public class Demo8AtomicReferenceArray {

    public static void main(String[] args) throws
InterruptedException {
        User u1 = new User("张三", 22);
        User u2 = new User("李四", 33);
        User[] arr = {u1, u2};

        AtomicReferenceArray<User> ara = new
AtomicReferenceArray<User>(arr);
        System.out.println(ara.toString());
```

```
        System.out.println(ara.toString());
    }

    static class User {
        private String name;
        public volatile int age;

        public User(String name, int age) {
            super();
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

7.5 对象的属性修改类型

> AtomicIntegerFieldUpdater:原子更新整形字段的更新器
> AtomicLongFieldUpdater：原子更新长整形字段的更新器
> AtomicReferenceFieldUpdater ：原子更新引用类形字段的更新器

但是他们的使用通常有以下几个限制：

- 限制1：操作的目标不能是static类型，前面说到的unsafe提取的是非static类型的属性偏移量，如果是static类型在获取时如果没有使用对应的方法是会报错的，而这个Updater并没有使用对应的方法。
- 限制2：操作的目标不能是final类型的，因为final根本没法修改。
- 限制3：必须是volatile类型的数据，也就是数据本身是读一致的。
- 限制4：属性必须对当前的Updater所在的区域是可见的，也就是private如果不是当前类肯定是不可见的，protected如果不存在父子关系也是不可见的，default如果不是在同一个package下也是不可见的。

实现方式：通过反射找到属性，对属性进行操作。

例子：

```
public class AtomicIntegerFieldUpdaterTest {
    public static void main(String[] args) {
        AtomicIntegerFieldUpdater<User> a =
AtomicIntegerFieldUpdater.newUpdater(User.class, "age");

        User user = new User("Java", 22);
        System.out.println(a.get(user));
        System.out.println(a.getAndAdd(user,10));
        System.out.println(a.get(user));
    }
}

class User {
    private String name;
    public volatile int age;

    public User(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
```

```
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

}
```

7.6 JDK1.8新增类

> LongAdder：长整型原子类
> DoubleAdder：双浮点型原子类
> LongAccumulator：类似LongAdder，但要更加灵活(要传入一个函数式接口)
> DoubleAccumulator：类似DoubleAdder，但要更加灵活(要传入一个函数式接口)

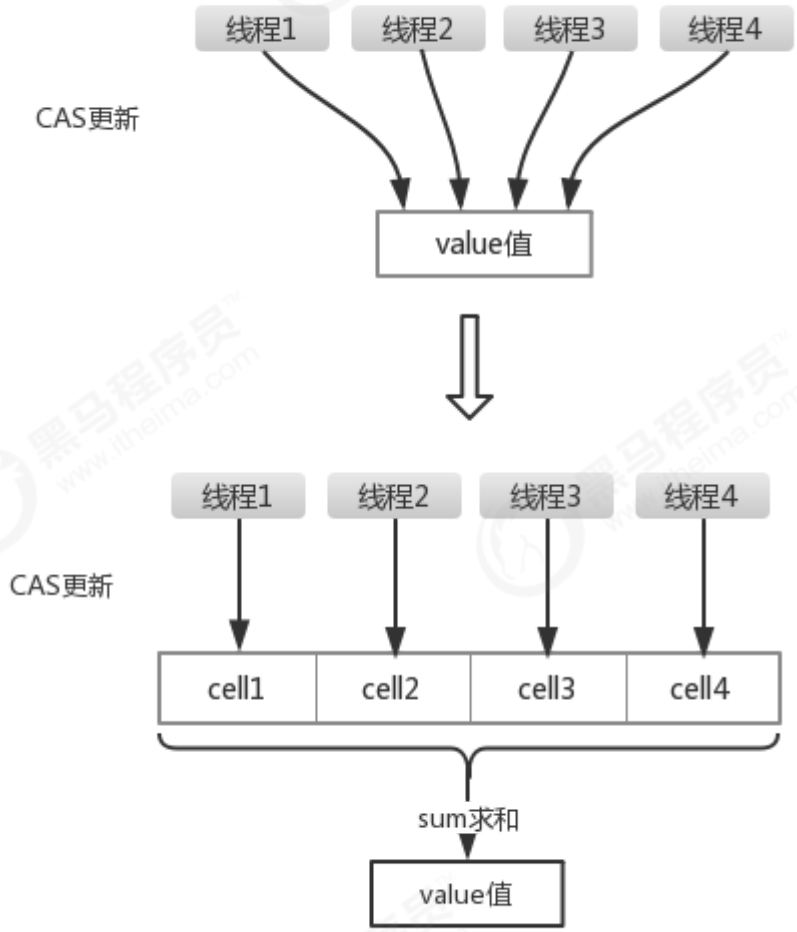
LongAdder是jdk1.8提供的累加器，基于Striped64实现，所提供的API基本上可以替换原先的AtomicLong。

LongAdder类似于AtomicLong是原子性递增或者递减类，AtomicLong已经通过CAS提供了非阻塞的原子性操作，相比使用阻塞算法的同步器来说性能已经很好了，但是JDK开发组并不满足，因为在非常高的并发请求下AtomicLong的性能不能让他们接受，虽然AtomicLong使用CAS但是CAS失败后还是通过无限循环的自旋锁不断尝试。

```
public final long incrementAndGet() {
    for (;;) {
        long current = get();
        long next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

在高并发下N多线程同时去操作一个变量会造成大量线程CAS失败然后处于自旋状态，这大大浪费了cpu资源，降低了并发性。那么既然AtomicLong性能由于过多线程同时去竞争一个变量的更新而降低的，那么如果把一个变量分解为多个变量，让同样多的线程去竞争多个资源那么性能问题不就解决了？是的，JDK8提供的LongAdder就是这个思路。下面通过图形来标示两者不同。

AtomicLong和LongAdder对比：



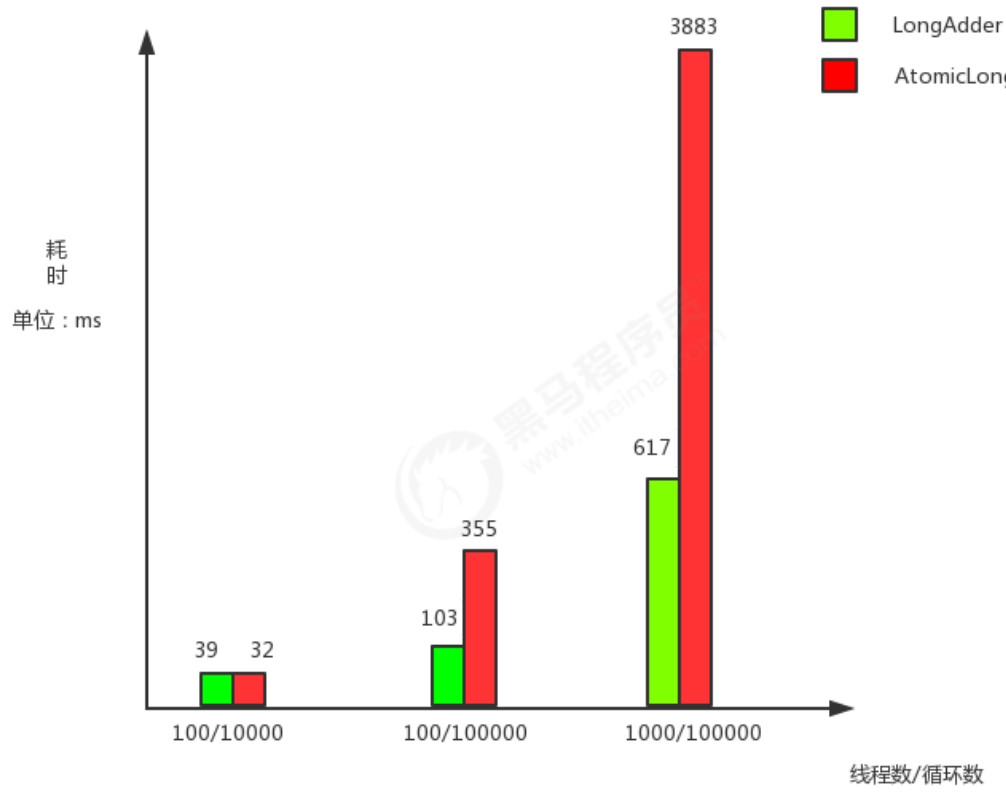
一段LongAdder和Atomic的对比测试代码：

```
public class Demo9Compare {

    public static void main(String[] args) {
        AtomicLong atomicLong = new AtomicLong(0L);
        LongAdder longAdder = new LongAdder();
```

```
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < 1000000; j++) {
                        //atomicLong.incrementAndGet();
                        longAdder.increment();
                    }
                }
            }).start();
        }

        while (Thread.activeCount() > 2) {
        }

        System.out.println(atomicLong.get());
        System.out.println(longAdder.longValue());

        System.out.println("耗时: " + (System.currentTimeMillis()
- start));

    }
}
```

不同计算机因为CPU、内存等硬件不一样，所以测试的数值也不一样，但是得到的结论都是一样的

测试结果：



从上结果图可以看出，在并发比较低的时候，LongAdder和AtomicLong的效果非常接近。但是当并发较高时，两者的差距会越来越大。上图中在线程数为1000，每个线程循环数为100000时，LongAdder的效率是AtomicLong的6倍左右。

8 J.U.C之AQS

8.1 AQS简介

AQS(AbstractQueuedSynchronizer），即队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentrantLock、ReentrantReadWriteLock、Semaphore等），JUC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。它是JUC并发包中的核心基础组件。

在这里我们只是对AQS进行了解，它只是一个抽象类，但是JUC中的很多组件都是基于这个抽象类，也可以说这个AQS是多数JUC组件的基础。

8.1.1 AQS的作用

synchronized还是存在一些缺陷的：它缺少了获取锁与释放锁的可操作性，可中断、超时获取锁，而且独占式在高并发场景下性能大打折扣。

AQS解决了实现同步器时涉及到的大量细节问题，例如获取同步状态、FIFO同步队列。基于AQS来构建同步器可以带来很多好处。它不仅能够极大地减少实现工作，而且也不必处理在多个位置上发生的竞争问题。

8.1.2 state状态

AQS维护了一个volatile int类型的变量state表示当前同步状态。当state>0时表示已经获取了锁，当state = 0时表示释放了锁。

它提供了三个方法来对同步状态state进行操作：

> getState()：返回同步状态的当前值
> setState()：设置当前同步状态
> compareAndSetState()：使用CAS设置当前状态，该方法能够保证状态设置的原子性

这三种操作均是CAS原子操作，其中compareAndSetState的实现依赖于Unsafe的compareAndSwapInt()方法

8.1.3 资源共享方式

AQS定义两种资源共享方式：

- Exclusive（独占，只有一个线程能执行，如ReentrantLock）
- Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

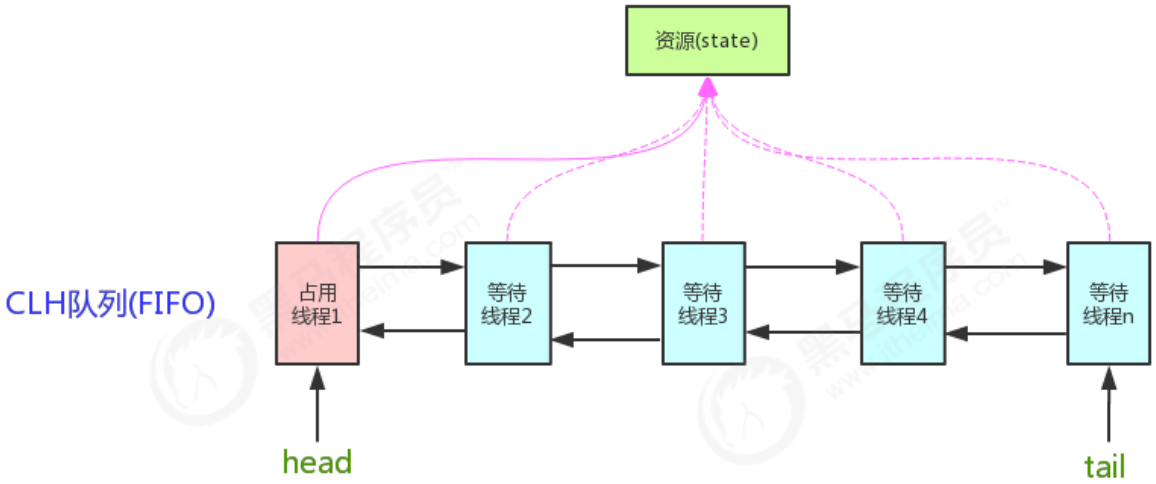
> - isHeldExclusively()：当前同步器是否在独占式模式下被线程占用，一般该方法表示是否被当前线程所独占。只有用到condition才需要去实现它。
> - tryAcquire(int)：独占方式。尝试获取同步状态，成功则返回true，失败则返回false。其他线程需要等待该线程释放同步状态才能获取同步状态。

返回false。

- tryAcquireShared(int)：共享方式。尝试获取同步状态。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放同步状态，如果释放后允许唤醒后续等待结点，返回true，否则返回false。

8.2 CLH同步队列

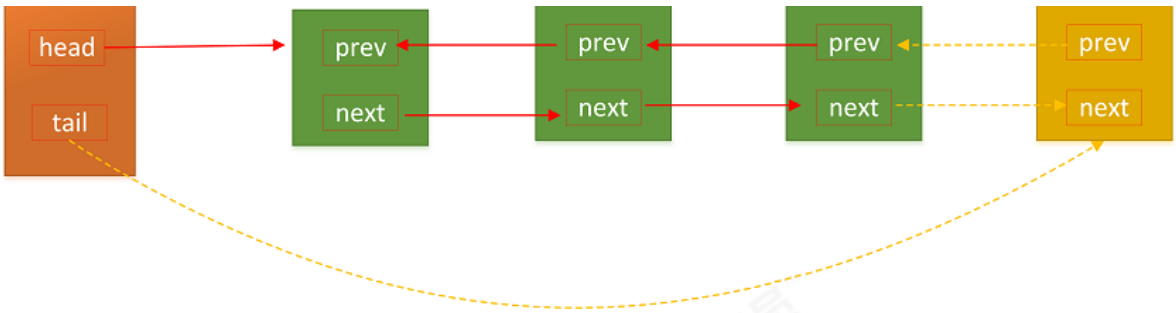
AQS内部维护着一个FIFO队列，该队列就是CLH同步队列，遵循FIFO原则（First Input First Output先进先出）。CLH同步队列是一个FIFO双向队列，AQS依赖它来完成同步状态的管理。



当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。

8.2.3 入列

CLH队列入列非常简单，就是tail指向新节点、新节点的prev指向当前最后的节点，当前最后一个节点的next指向当前节点。



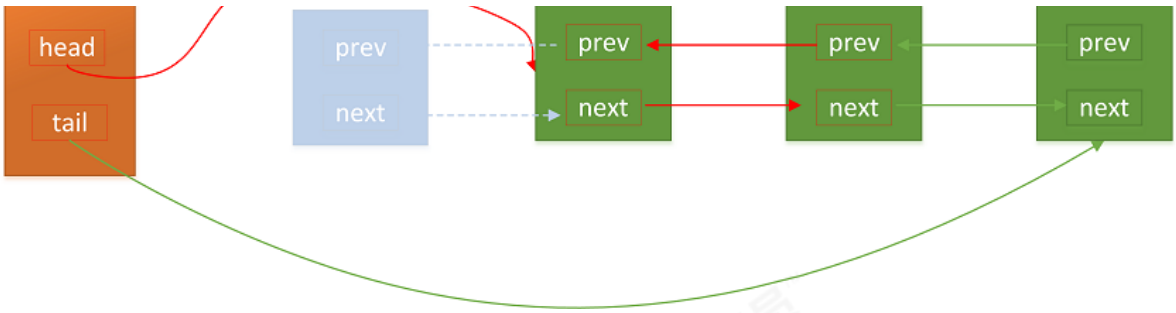
代码我们可以看看addWaiter(Node node)方法：

```
private Node addWaiter(Node mode) {
    //新建Node
    Node node = new Node(Thread.currentThread(), mode);
    //快速尝试添加尾节点
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        //CAS设置尾节点
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    //多次尝试
    enq(node);
    return node;
}
```

在上面代码中，两个方法都是通过一个CAS方法compareAndSetTail(Node expect, Node update)来设置尾节点，该方法可以确保节点是线程安全添加的。在enq(Node node)方法中，AQS通过“死循环”的方式来保证节点可以正确添加，只有成功添加后，当前线程才会从该方法返回，否则会一直执行下去。

8.2.4 出列

CLH同步队列遵循FIFO，首节点的线程释放同步状态后，将会唤醒它的后继节点（next），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。head执行该节点并断开原首节点的next和当前节点的prev即可，注意在这个过程是不需要使用CAS来保证的，因为只有一个线程能够成功获取到同步状态。过程图如下：



9 J.U.C之锁

9.1 锁的基本概念

虽然在前面锁优化的部分已经提到过一些锁的概念，但不完全，这里是对锁的概念补充。

9.1.1 互斥锁

在编程中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。每个对象都对应于一个可称为" 互斥锁" 的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

9.1.2 阻塞锁

阻塞锁，可以说是让线程进入阻塞状态进行等待，当获得相应的信号（唤醒，时间） 时，才可以进入线程的准备就绪状态，准备就绪状态的所有线程，通过竞争，进入运行状态。

9.1.3 自旋锁

自旋锁是采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时，才能进入临界区。

由于自旋锁只是将当前线程不停地执行循环体，不进行线程状态的改变，所以响应速度更快。但当线程数不停增加时，性能下降明显，因为每个线程都需要执行，占用CPU时间。如果线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁。

9.1.4 读写锁

读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。

读写锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。

9.1.5 公平锁

公平锁（Fair）：加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得

非公平锁（Nonfair）：加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待

非公平锁性能比公平锁高，因为公平锁需要在多核的情况下维护一个队列。

9.2 ReentrantLock

ReentrantLock，可重入锁，是一种递归无阻塞的同步机制。它可以等同于synchronized的使用，但是ReentrantLock提供了比synchronized更强大、灵活的锁机制，可以减少死锁发生的概率。

ReentrantLock还提供了公平锁和非公平锁的选择，构造方法接受一个可选的公平参数（默认非公平锁），当设置为true时，表示公平锁，否则为非公平锁。公平锁的效率往往没有非公平锁的效率高，在许多线程访问的情况下，公平锁表现出较低的吞吐量。

查看ReentrantLock源码中的构造方法：

```
public ReentrantLock() {
    //非公平锁
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    //公平锁
    sync = fair ? new FairSync() : new NonfairSync();
}
```

AQS（AbstractQueuedSynchronizer），它有两个子类：公平锁FairSync和非公平锁NonfairSync。

9.2.1 获取锁

一般都是这么使用ReentrantLock获取锁的：（默认非公平锁）

```
//非公平锁
ReentrantLock lock = new ReentrantLock();
lock.lock();
```

lock方法：

```
public void lock() {
    sync.lock();
}
```

加锁最终可以看到会调用方法：

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

其实底层就是使用AQS同步队列。

9.2.2 释放锁

获取同步锁后，使用完毕则需要释放锁，ReentrantLock提供了unlock释放锁：

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

unlock内部使用Sync的release()释放锁，release()是在AQS中定义的：

```
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

释放同步状态的tryRelease()是同步组件自己实现：

```
protected final boolean tryRelease(int releases) {
    //减掉releases
    int c = getState() - releases;
    //如果释放的不是持有锁的线程，抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    //state == 0 表示已经释放完全了，其他线程可以获取同步状态了
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

只有当同步状态彻底释放后该方法才会返回true。当同步队列的状态state == 0时，则将锁持有线程设置为null，free= true，表示释放成功。

9.2.3 公平锁与非公平锁原理

公平锁与非公平锁的区别在于获取锁的时候是否按照FIFO的顺序来。释放锁不存在公平性和非公平性，比较非公平锁和公平锁获取同步状态的过程，会发现两者唯一的区别就在于公平锁在获取同步状态时多了一个限制条件：hasQueuedPredecessors()，定义如下：

```
    Node t = tail;  //尾节点
    Node h = head;  //头节点
    Node s;

    //头节点 != 尾节点
    //同步队列第一个节点不为null
    //当前线程是同步队列第一个节点
    return h != t &&
        ((s = h.next) == null || s.thread !=
Thread.currentThread());
}
```

该方法主要做一件事情：主要是判断当前线程是否位于CLH同步队列中的第一个。如果是则返回true，否则返回false。

9.2.4 ReentrantLock与synchronized的区别

前面提到ReentrantLock提供了比synchronized更加灵活和强大的锁机制，那么它的灵活和强大之处在哪里呢？他们之间又有什么相异之处呢？

1. 与synchronized相比，ReentrantLock提供了更多，更加全面的功能，具备更强的扩展性。例如：时间锁等候，可中断锁等候，锁投票。
2. ReentrantLock还提供了条件Condition，对线程的等待、唤醒操作更加详细和灵活，所以在多个条件变量和高度竞争锁的地方，ReentrantLock更加适合（以后会阐述Condition）。
3. ReentrantLock提供了可轮询的锁请求。它会尝试着去获取锁，如果成功则继续，否则可以等到下次运行时处理，而synchronized则一旦进入锁请求要么成功要么阻塞，所以相比synchronized而言，ReentrantLock会不容易产生死锁些。
4. ReentrantLock支持更加灵活的同步代码块，但是使用synchronized时，只能在同一个synchronized块结构中获取和释放。注：ReentrantLock的锁释放一定要在finally中处理，否则可能会产生严重的后果。
5. ReentrantLock支持中断处理，且性能较synchronized会好些。

9.3 读写锁ReentrantReadWriteLock

问，但是在大多数场景下，大部分的问都是提供读服务，而写服务占有的时间较少。然而读服务不存在数据竞争问题，如果一个线程在读时禁止其他线程读势必会导致性能降低。所以就提供了读写锁。

读写锁维护着一对锁，一个读锁和一个写锁。通过分离读锁和写锁，使得并发性比一般的互斥锁有了较大的提升：在同一时间可以允许多个读线程同时访问，但是在写线程访问时，所有读线程和写线程都会被阻塞。

读写锁的主要特性：

1. 公平性：支持公平性和非公平性。
2. 重入性：支持重入。读写锁最多支持65535个递归写入锁和65535个递归读取锁。
3. 锁降级：写锁能够降级成为读锁，遵循获取写锁、获取读锁在释放写锁的次序。读锁不能升级为写锁。

读写锁ReentrantReadWriteLock实现接口ReadWriteLock，该接口维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

ReadWriteLock定义了两个方法。readLock()返回用于读操作的锁，writeLock()返回用于写操作的锁。ReentrantReadWriteLock定义如下：

```
/** 内部类  读锁 */
private final ReentrantReadWriteLock.ReadLock readerLock;
/** 内部类  写锁 */
private final ReentrantReadWriteLock.WriteLock writerLock;

final Sync sync;

/** 使用默认（非公平）的排序属性创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock() {
    this(false);
}

/** 使用给定的公平策略创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
```

```
}

/** 返回用于写入操作的锁 */
public ReentrantReadWriteLock.WriteLock writeLock() { return
writerLock; }
/** 返回用于读取操作的锁 */
public ReentrantReadWriteLock.ReadLock  readLock()  { return
readerLock; }

abstract static class Sync extends AbstractQueuedSynchronizer {
    //省略其余源代码
}
public static class WriteLock implements Lock,
java.io.Serializable{
    //省略其余源代码
}

public static class ReadLock implements Lock,
java.io.Serializable {
    //省略其余源代码
}
```

ReentrantReadWriteLock与ReentrantLock一样，其锁主体依然是Sync，它的读锁、写锁都是依靠Sync来实现的。所以ReentrantReadWriteLock实际上只有一个锁，只是在获取读取锁和写入锁的方式上不一样而已，它的读写锁其实就是两个类：ReadLock、writeLock，这两个类都是lock实现。

在ReentrantLock中使用一个int类型的state来表示同步状态，该值表示锁被一个线程重复获取的次数。但是读写锁ReentrantReadWriteLock内部维护着一对锁，需要用一个变量维护多种状态。所以读写锁采用"按位切割使用"的方式来维护这个变量，将其切分为两部分，高16为表示读，低16为表示写。分割之后，读写锁是如何迅速确定读锁和写锁的状态呢？通过位运算。假如当前同步状态为S，那么写状态等于 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)。代码如下：

```
static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

9.3.1 写锁的获取

写锁就是一个支持可重入的互斥锁。

写锁的获取最终会调用tryAcquire(int arg)，该方法在内部类Sync中实现：

```
    protected final boolean tryAcquire(int acquires) {
        Thread current = Thread.currentThread();
        //当前锁个数
        int c = getState();
        //写锁
        int w = exclusiveCount(c);
        if (c != 0) {
            //c != 0 && w == 0 表示存在读锁
            //当前线程不是已经获取写锁的线程
            if (w == 0 || current != getExclusiveOwnerThread())
                return false;
            //超出最大范围
            if (w + exclusiveCount(acquires) > MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
            setState(c + acquires);
            return true;
        }
        //是否需要阻塞
        if (writerShouldBlock() ||
                !compareAndSetState(c, c + acquires))
            return false;
        //设置获取锁的线程为当前线程
        setExclusiveOwnerThread(current);
        return true;
    }
```

条件：读锁是否存在。因为要确保写锁的操作对读锁是可见的，如果在存在读锁的情况下允许获取写锁，那么那些已经获取读锁的其他线程可能就无法感知当前写线程的操作。因此只有等读锁完全释放后，写锁才能够被当前线程所获取，一旦写锁开始获取了，所有其他读、写线程均会被阻塞。

9.3.2 写锁的释放

获取了写锁用完了则需要释放，WriteLock提供了unlock()方法释放写锁：

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

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

写锁的释放最终还是会调用AQS的模板方法release(int arg)方法，该方法首先调用tryRelease(int arg)方法尝试释放锁，tryRelease(int arg)方法为读写锁内部类Sync中定义了，如下：

```
protected final boolean tryRelease(int releases) {
    //释放的线程不为锁的持有者
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    int nextc = getState() - releases;
    //若写锁的新线程数为0，则将锁的持有者设置为null
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
    setState(nextc);
    return free;
}
```

当写状态为0的表示写锁已经完全释放了，从而等待的其他线程可以继续访问读写锁，获取同步状态，同时此次写线程的修改对后续的线程可见。

9.3.3 读锁的获取

读锁为一个可重入的共享锁，它能够被多个线程同时持有，在没有其他写线程访问时，读锁总是获取成功。

读锁的获取可以通过ReadLock的lock()方法：

```
public void lock() {
    sync.acquireShared(1);
}
```

Sync的acquireShared(int arg)定义在AQS中：

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

9.3.4 读锁的释放

与写锁相同，读锁也提供了unlock()释放读锁：

```
public void unlock() {
    sync.releaseShared(1);
}
```

unlcok()方法内部使用Sync的releaseShared(int arg)方法，该方法也定义AQS中：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

读写锁有一个特性就是锁降级，锁降级就意味着写锁是可以降级为读锁的。锁降级需要遵循以下顺序：

`获取写锁=>获取读锁=>释放写锁`

9.3.6 读写锁例子

```
public class Demo10ReentrantReadWriteLock {
    private static volatile int count = 0;

    public static void main(String[] args) {
        ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
        WriteDemo writeDemo = new WriteDemo(lock);
        ReadDemo readDemo = new ReadDemo(lock);

        for (int i = 0; i < 3; i++) {
            new Thread(writeDemo).start();
        }
        for (int i = 0; i < 5; i++) {
            new Thread(readDemo).start();
        }
    }

    static class WriteDemo implements Runnable {
        ReentrantReadWriteLock lock;
        public WriteDemo(ReentrantReadWriteLock lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                lock.writeLock().lock();
                count++;
                System.out.println("写锁："+count);
```

```
            }
        }

        static class ReadDemo implements Runnable {
            ReentrantReadWriteLock lock;
            public ReadDemo(ReentrantReadWriteLock lock) {
                this.lock = lock;
            }

            @Override
            public void run() {
                for (int i = 0; i < 5; i++) {
                    try {
                        TimeUnit.MILLISECONDS.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    lock.readLock().lock();
                    System.out.println("读锁："+count);
                    lock.readLock().unlock();
                }
            }
        }
    }
```

10 J.U.C之Condition

10.1 Condition介绍

在没有Lock之前，我们使用synchronized来控制同步，配合Object的wait()、notify()系列方法可以实现等待/通知模式。在JDK5后，Java提供了Lock接口，相对于Synchronized而言，Lock提供了条件Condition，对线程的等待、唤醒操作更加详细和灵活。

下图是Condition与Object的监视器方法的对比：

前置条件	获取对象的锁	调用Lock.newCondtion()获取Condition对象
调用方式	直接调用	直接调用
等待队列个数	一个	多个
当前线程释放锁并进入等待状态	支持	支持
当前线程释放锁并进入等待状态，在等待状态中不响应中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入从等待状态到将来的某个时间	不支持	支持
唤醒等待队列中的某个线程	支持	支持
唤醒等待队列中的全部线程	支持	支持

Condition提供了一系列的方法来对阻塞和唤醒线程：

1. **await()**：造成当前线程在接到信号或被中断之前一直处于等待状态。
2. **await(long time, TimeUnit unit)**：造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。
3. **awaitNanos(long nanosTimeout)**：造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。返回值表示剩余时间，如果在nanosTimesout之前唤醒，那么返回值 = nanosTimeout – 消耗时间，如果返回值 <= 0 ,则可以认定它已经超时了。
4. **awaitUninterruptibly()**：造成当前线程在接到信号之前一直处于等待状态。【注意：该方法对中断不敏感】。
5. **awaitUntil(Date deadline)**：造成当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态。如果没有到指定时间就被通知，则返回true，否则表示到了指定时间，返回返回false。
6. **signal()**：唤醒一个等待线程。该线程从等待方法返回前必须获得与Condition相关的锁。
7. **signal()All**：唤醒所有等待线程。能够从等待方法返回的线程必须获得与Condition相关的锁。

   Condition是一种广义上的条件队列（等待队列）。他为线程提供了一种更为灵活的等待/通知模式，线程在调用await方法后执行挂起操作，直到线程等待的某个条件为真时才会被唤醒。Condition必须要配合锁一起使用，因为对共享状态变量的访问发生在多线程环境下。一个Condition的实例必须与一个Lock绑定，因此Condition一般都是作为Lock的内部实现。

案例：

```
public class Demo11Condition {

    private Lock reentrantLock = new ReentrantLock();
```

```
    public void m1() {
        reentrantLock.lock();
        try {
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已经进入执行等待。。。");
            condition1.await();
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已被唤醒，继续执行。。。");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            reentrantLock.unlock();
        }
    }

    public void m2() {
        reentrantLock.lock();
        try {
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已经进入执行等待。。。");
            condition1.await();
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已被唤醒，继续执行。。。");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            reentrantLock.unlock();
        }
    }

    public void m3() {
        reentrantLock.lock();
        try {
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已经进入执行等待。。。");
            condition2.await();
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已被唤醒，继续执行。。。");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            reentrantLock.unlock();
```

```
    }

    public void m4() {
        reentrantLock.lock();
        try {
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已经进入发出condition1唤醒信
号。。。");
            condition1.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            reentrantLock.unlock();
        }
    }

    public void m5() {
        reentrantLock.lock();
        try {
            System.out.println("线程 " +
Thread.currentThread().getName() + " 已经进入发出condition2唤醒信
号。。。");
            condition2.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            reentrantLock.unlock();
        }
    }

    public static void main(String[] args) throws Exception {
        final Demo11Condition useCondition = new
Demo11Condition();
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                useCondition.m1();
            }
        }, "t1");
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                useCondition.m2();
            }
        }, "t2");
        Thread t3 = new Thread(new Runnable() {
```

```
            }
        }, "t3");
        Thread t4 = new Thread(new Runnable() {
            public void run() {
                useCondition.m4();
            }
        }, "t4");
        Thread t5 = new Thread(new Runnable() {
            public void run() {
                useCondition.m5();
            }
        }, "t5");

        t1.start();
        t2.start();
        t3.start();

        Thread.sleep(2000);
        t4.start();


        Thread.sleep(2000);
        t5.start();
    }
}
```

10.2 Condition的实现

获取一个Condition必须通过Lock的newCondition()方法。该方法定义在接口Lock下面，返回的结果是绑定到此 Lock 实例的新 Condition 实例。Condition为一个接口，其下仅有一个实现类ConditionObject，由于Condition的操作需要获取相关的锁，而AQS则是同步锁的实现基础，所以ConditionObject则定义为AQS的内部类。定义如下：

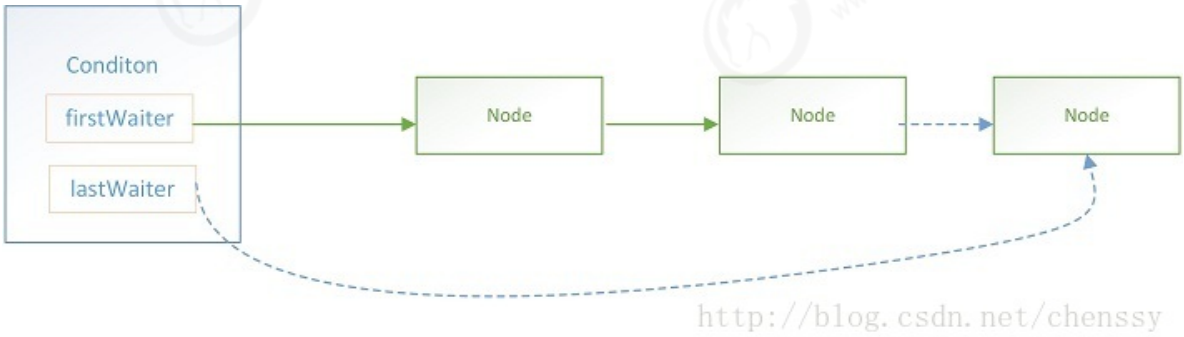
```
public class ConditionObject implements Condition,
java.io.Serializable {
}
```

10.2.1 等待队列

能的关键。在队列中每一个节点都包含有一个线程引用，该线程就是在该Condition对象上等待的线程。源码如下：

```
public class ConditionObject implements Condition,
java.io.Serializable {
    private static final long serialVersionUID =
1173984872572414699L;

    //头节点
    private transient Node firstWaiter;
    //尾节点
    private transient Node lastWaiter;

    public ConditionObject() {
    }

    /** 省略方法 **/
}
```

从上面代码可以看出Condition拥有首节点（firstWaiter），尾节点（lastWaiter）。当前线程调用await()方法，将会以当前线程构造成一个节点（Node），并将节点加入到该队列的尾部。结构如下：



Node里面包含了当前线程的引用。Node定义与AQS的CLH同步队列的节点使用的都是同一个类（AbstractQueuedSynchronized.Node静态内部类）。

Condition的队列结构比CLH同步队列的结构简单些，新增过程较为简单只需要将原尾节点的nextWaiter指向新增节点，然后更新lastWaiter即可。

10.2.2 等待状态

调用Condition的await()方法会使当前线程进入等待状态，同时会加入到Condition等待队列同时释放锁。当从await()方法返回时，当前线程一定是获取了Condition相关连的锁。

```
        //当前线程中断
        if (Thread.interrupted())
            throw new InterruptedException();
        //当前线程加入等待队列
        Node node = addConditionWaiter();
        //释放锁
        long savedState = fullyRelease(node);
        int interruptMode = 0;
        /**
         * 检测此节点的线程是否在同步队上，如果不在，则说明该线程还不具备竞争锁的资格，则继续等待
         * 直到检测到此节点在同步队列上
         */
        while (!isOnSyncQueue(node)) {
            //线程挂起
            LockSupport.park(this);
            //如果已经中断了，则退出
            if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                break;
        }
        //竞争同步状态
        if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
            interruptMode = REINTERRUPT;
        //清理下条件队列中的不是在等待条件的节点
        if (node.nextWaiter != null) // clean up if cancelled
            unlinkCancelledWaiters();
        if (interruptMode != 0)
            reportInterruptAfterWait(interruptMode);
    }
```

此段代码的逻辑是：首先将当前线程新建一个节点同时加入到条件队列中，然后释放当前线程持有的同步状态。然后则是不断检测该节点代表的线程释放出现在CLH同步队列中（收到signal信号之后就会在AQS队列中检测到），如果不存在则一直挂起，否则参与竞争同步状态。

10.2.3 通知

调用Condition的signal()方法，将会唤醒在等待队列中等待最长时间的节点（条件队列里的首节点），在唤醒节点前，会将节点移到CLH同步队列中。

```
    //检测当前线程是否为拥有锁的独
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    //头节点，唤醒条件队列中的第一个节点
    Node first = firstWaiter;
    if (first != null)
        doSignal(first);    //唤醒
}
```

该方法首先会判断当前线程是否已经获得了锁，这是前置条件。然后唤醒等待队列中的头节点。

doSignal(Node first)：唤醒头节点

```
private void doSignal(Node first) {
    do {
        //修改头结点，完成旧头结点的移出工作
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}
```

doSignal(Node first)主要是做两件事：

1.修改头节点，

2.调用transferForSignal(Node first) 方法将节点移动到CLH同步队列中。