1. atomic包
2. 概述

JUC中有一个包java.util.concurrent.atomic中存放着原子操作的类，如AtomicInteger，大致保证阔基本类型，引用类型，数组类型，对象的属性修改器类型，JDK1.8新增类。

1. 基本类型

使用原子的方式更新基本类型，如AtomicInteger，AtomicLong的主要API如下所示：

get() 返回值

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) 仅当get时才会set

compareAndSet(int,int) 尝试新增后对比，如果增加成功则返回true

AtomicBoolean类的主要API如下所示：

compareAndSet(Boolean, Boolean) 参数为原始值和修改的新值，修改成功返回true

getAndSet(Boolean) 尝试设置新的boolean值，返回设置前的值

1. 引用类型

AtomicStampedReference类仅仅是AtomicReference类的再一次封装，增加了一层引用和计数器，计数器的设置是由自己控制的，可以按照自己的方式标识版本号，一般是自增操作。

AtomicMarkableReference和AtomicStampedReference功能差不多，只不过描述的只是两种状态，是与否，而AtomicStampedReference是多种状态。

1. 数组类型

使用原子的方式更新数组里的某个元素，如AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray。很多API用法与基本类型都是相似的，不在赘述。

1. 对象的属性修改器类型

如果需要原子更新某个类中的某个字段时，需要用到对象的属性修改器类型原子类。如AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater。但是修改的对象是有一些限制的，如下所示：

1. 操作的目标不能是static类型，因为CAS使用的Unsafe类的方法提取的都是非static类型的属性偏移量，如果目标是static类型，在获取时没有使用对应的static方法会报错的。
2. 操作的目标不能是final类型，因为final类型无法修改。
3. 必须是volatile类型，即内存可见的。
4. 属性必须对当前的修改器updater所在的区域是可见的，如果是private则必须是当前类中；如果是protected则必须有父子关系；如果是default则必须在同一个包下。
5. JDK1.8新增类

虽然普通的atomic类方法已经通过CAS优化了原子操作，但是对于高并发的场景，多个线程CAS都会失败，并陷入无限的自旋锁中，浪费资源。因此，JDK1.8后，新增了多个类来优化这种缺陷。原有的atomic类由于多个线程同时竞争一个变量而造成CAS失败，因此，如果将该变量分解为多个变量，让每个线程都能够获取到变量进行CAS操作不就可行了吗！是的，新增的类就是这种思路，有LongAdder, DoubleAdder, LongAccumulator, DoubleAccumulator，这些新增的类的性能对于多线程的情况优势十分明显。

1. AQS
2. 概述

AQS（Abastract Queue Synchrinizer），队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentranLock, ReentrantReadWriteLock, Semaphore等），是JJUC并发包中的核心基础组件。

1. 优势

AQS解决了实现同步器时涉及到的大量细节问题，如获取同步状态，FIFO同步队列。基于AQS来构建同步器可以极大地减少工作，也不必处理在多个位置的竞争问题。

1. state同步状态

AQS维护了一个volatile int类型的变量state表示当前的同步状态，当state>0时表示已经获取了锁，当state=0表示释放了锁。有以下三个方法操作state:

getState() //返回state值

setState() //设置当前同步状态

compareAndSetState() //使用CAS设置当前状态，保证状态设置的原子性，依赖于Unsafe类的compareAndSwapInt()方法实现

1. 资源共享方式

AQS定了了两种资源共享方式：

1. Exclusive，独占方式，只有一个线程能够执行，如ReentrantLock
2. Share， 共享方式， 多个线程可以同时执行，如Semaphore / CountDownLatch
3. CHL同步队列

AQS内部维护着一个CHL同步队列，遵循FIFO原则，AQS依赖它来完成同步状态的管理。每个线程排队进入CHL队列中，当前线程如果获取同步状态失败时，AQS会将当前线程已经等待的状态信息构造成一个节点，加入到CHL同步队列中，同时会阻塞当前线程，当同步状态释放时，会把队头的节点去掉，唤醒后面一个节点，使其再次尝试获取同步状态。队列能够保证每个时刻只有一个线程能够获取到同步状态，因此出队列的操作不需要使用CAS来保证原子性，而不同的线程入队列的操作需要保证原子性，因为同时会有不同的线程入队列。

1. 锁
2. 锁的类型
3. 互斥锁

对象互斥保证共享数据操作的原子性，每个对象都对应一个可称为“互斥锁”的标记，用来保证在任一时刻，只能有一个线程访问该对象。

1. 阻塞锁

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，就绪状态的所有线程通过竞争获得对象锁。

1. 自旋锁

让当前线程不停地执行无意义的循环，当循环的条件被其他线程改变时，才能够进入临界区。这种锁由于不进行线程状态的改变，因此响应速度更快，但是如果多线程竞争着锁，就会出现持续自旋的资源浪费。

1. 读写锁

特殊的自旋锁，将共享资源的访问者分为了读者和写着，读者只对共享资源进行读访问，写者需要对共享资源进行写操作。读操作可以并发执行，而写操作是排他的，只能单线程进行，不能同时有读者和写者。

1. 公平锁

公平锁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。非公平锁在加锁前不考虑排队等待问题，直接尝试获取锁，获取不到就到队尾等待。非公平锁的性能搞好，因为公平锁需要在多核的情况下维护一个队列。

1. ReentrantLock
2. 概述

可重入锁，是一种递归无阻塞的同步机制，即可以在锁中嵌套另一个锁，通过同步状态值来判断当前锁的状态。该锁等同于synchronized的使用，但是ReentrantLock更加灵活，强大，能够减少死锁的发生几率。

1. 构造方法

通过在构造方法中传入boolean参数，表示是否是公平锁，默认是非公平锁，传入true表示是公平锁。

构造方法源码如下所示：

public ReentrantLock() {

   //非公平锁

   sync = new NonfairSync();

}

​

public ReentrantLock(boolean fair) {

   //公平锁

   sync = fair ? new FairSync() : new NonfairSync();

}

Sync是ReentrantLock里的一个内部类，它继承了AQS，有两个子类，一个是公平锁FariSync，另一个是非公平锁NonfairSync。

1. 获取锁

创建锁对象，通过lock方法来获取锁。

ReentrantLock lock = new ReentrantLock();

lock.lock();

lock方法也是调用的Sync类的lock方法：

public void lock(){

sync.lock();

}

最后会调用AQS同步队列的方法来加锁。

1. 释放锁

获取同步锁，使用完毕后需要释放锁，调用unlock()方法，该方法中调用了Sync类的release()方法，release方法定义在AQS中，源码如下：

public final boolean release(int arg) {

   if (tryRelease(arg)) {

       Node h = head;

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

           unparkSuccessor(h);

       return true;

  }

   return false;

}

其中使用的tryRelease()方法的源码如下所示：

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;

}

可以看到，由于ReentrantLock锁是可重入锁，可以嵌套锁，因此同步状态可能有好几层，只有每一层嵌套的锁都释放掉后，同步队列的状态state=0，则将锁持有的线程释放。

1. 公平锁和非公平锁

公平锁在获取同步状态时多了一个限制条件，判断当前的线程是否位于CHL同步队列中的第一个，如果是则可以获取锁，这就保证了FIFO的公平性。

1. ReentrantLock对于Synchronized的优势
2. 提供更多的功能，如时间锁等候，可中断锁等候，锁投票。
3. 提供了Condition，能够更加灵活地操作线程的等待，唤醒，比如可以通过Condition和ReentrantLock的绑定，仅仅唤醒部分的线程。
4. 提供了可轮询的锁请求，尝试地去获取锁，如果成功则继续，否则可以等到下次运行时处理，而不像synchronized那样对于锁请求只有成功和阻塞两种结果，这样ReentrantLock不容易产生死锁。
5. 更加灵活的同步代码块，使用synchronized时，只能在同一个synchronized块结构中获取和释放，注意，ReentrantLock的锁释放一定要在finally中。
6. 支持中断处理，性能更好。
7. 读写锁ReentrantReadWriteLock
8. 概述

大多数场景下，大部分时间提供的都是读服务，写服务的时间很少，因此如果使用ReentrantLock这个互斥锁，会对读线程的性能造成很大的影响，读写锁由此而生。它维护着一对锁，一个读锁和一个写锁，允许多个读线程并发，只允许写线程单线程执行，且写线程运行时，所有的读和写线程都被阻塞。

1. 特征
2. 公平性

支持公平锁和非公平锁。

1. 重入性

支持重入，读和写锁都可以嵌套65535个锁。

1. 锁降级

写锁能够降级称为读锁，读锁不能升级为写锁。

1. 实现

ReentrantReadWriteLock实现了接口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：

public interface ReadWriteLock {

   Lock readLock();

   Lock writeLock();

}

ReadWriteLock类定义了两个方法，readLock()返回读锁，writeLock()返回写锁，最终的锁主体还是要依靠Sync来实现的。所以ReentrantReadWriteLock实际上只有一个锁，只是在获取读锁和写锁的方式上不一样而已，都是由Lock类实现。

ReentrantLock中使用state整数来表示同步状态，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，需要用一个变量来维护多种状态。所以读写锁采用按位切割使用的方式维护这个变量，将其切分为两部分，高16位表示读，低16位表示写。

1. 写锁的获取

写锁是一个支持可重入的互斥锁，获取写锁最终会调用Sync类中的tryAcquire(int arg)方法，该方法和ReentrantLock中的tryAcquire(int arg)方法大致一样，不过在判断重入的时候添加了一个条件：是否存在读锁。因为要 确保写锁的操作对读锁是可见的，不能在读锁存在的情况下获取写锁，这样会导致写锁操作的不可见。只有等待所有读锁释放完毕后，写锁才能够被当前线程获取。

1. 写锁的释放

写锁释放锁的整个过程和ReentrantLock相似，由于存在锁的嵌套操作，每次释放都是减少写状态值，当写状态值为0时表示写锁已经完全释放了，从而其他等待的线程可以继续访问读写锁，获取同步状态。

1. 读锁的获取

读锁是一个可重入的共享锁，它能够被多个线程共同持有，在没有其他写线程的访问时，读锁总是获取成功的。

1. 读锁的释放

unlock()方法释放读锁。

1. 锁降级

由于写锁的优先级是高于读锁的，因此锁降级只能由写锁降级为读锁，读锁无法升级为写锁。锁降级遵循以下的顺序：首先获取写锁，然后获取读锁，最后释放掉写锁，就只剩下读锁了，降级成功。

1. Condition
2. 概述

Condition对线程的等待，唤醒操作更加灵活，原来通过synchronized的wait()和notify()方法实现的等待通知模式要么只能唤醒一个线程或者所有线程。Condition能够灵活地唤醒指定的部分线程。

Condition必须配合锁一起使用，因为对共享状态变量的访问发生在多线程环境中，一个Condition的实例必须和一个Lock绑定，因此Condition一般都是作为Lock的内部实现。

1. Condition的实现
2. AQS的内部类

获取一个Condition必须通过Lock的newCondition()方法，该方法定义在接口Lock下，返回的结果是绑定到此Lock实例的Condition实例。Condition是一个接口，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是同步锁的实现基础，所以ConditionObject定义为AQS的内部类，定义如下：

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

}

1. 等待队列

每个Condition对象都包含着一个FIFO队列，队列中每个节点都包含着一个线程引用，该线程就是在Condition对象上等待的线程，当前线程调用await()方法时，将会将当前线程构造成一个节点放入队列的尾部。过程与AQS中的CHL同步队列差不多，使用的都是同一个类(AbstractQueuedSynchronized.Node静态内部类)。

1. 等待状态

调用Condition的await()方法会使当前线程进入等待状态，同时加入到Condition等待队列中并且释放锁。然后不断检测该节点代表的线程是否出现在CLH同步队列中，如果存在则存在说明该线程被唤醒了，参与锁的竞争；如果不存在，则说明还在等待，则继续挂起。当从await()方法返回时，当前线程一定是获取了Condition相关的锁。

1. 通知

调用Condition的signal()方法，首先判断当前线程是否已经获得了锁，只有没有锁的线程才能够唤醒。然后唤醒在等待队列中的头结点，完成队列的头结点修改工作，并且将旧的头结点移动到CHL同步队列中，参与锁的竞争。