锁的分类

根据不同的标准锁可以分为以下7个类别：

1. **偏向锁/轻量级锁/重量级锁：**

   偏向锁：如果自始至终，对于这把锁都不存在竞争，那么其实就没必要上锁，只需要打个标记就行了，这就是偏向锁的思想。

   轻量级锁：JVM 开发者发现在很多情况下，synchronized 中的代码是被多个线程交替执行的，而不是同时执行的，也就是说并不存在实际的竞争，或者是只有短时间的锁竞争，用 CAS 就可以解决，这种情况下，用完全互斥的重量级锁是没必要的。轻量级锁是指当锁原来是偏向锁的时候，被另一个线程访问，说明存在竞争，那么偏向锁就会升级为轻量级锁，线程会通过自旋的形式尝试获取锁，而不会陷入阻塞。

   重量级锁：重量级锁是互斥锁，它是利用操作系统的同步机制实现的，所以开销相对比较大。当多个线程直接有实际竞争，且锁竞争时间长的时候，轻量级锁不能满足需求，锁就会膨胀为重量级锁。重量级锁会让其他申请却拿不到锁的线程进入阻塞状态。

   偏向锁性能最好，可以避免执行 CAS 操作。而轻量级锁利用自旋和 CAS 避免了重量级锁带来的线程阻塞和唤醒，性能中等。重量级锁则会把获取不到锁的线程阻塞，性能最差。

2. **可重入锁/非可重入锁：**

   可重入锁：同一个线程可以多次获取同一把锁。

   非可重入锁：指当前线程已经持有这把锁，如果再次获取这把锁，必须要先释放锁后才能再次尝试获取。

   对于可重入锁而言，最典型的就是 ReentrantLock 了，正如它的名字一样，reentrant 的意思就是可重入，它也是 Lock 接口最主要的一个实现类。

   可重入锁原理：

   每个锁关联线程持有者和计数器，某一线程请求成功后，JVM会记下锁的持有线程，计数器+1，请求线程请求该锁，必须等待，该线程再次拿到这个锁，计数器会递增，线程退出同步代码块时，计数器会递减，如果计数器为 0，则释放该锁。

3. **共享锁/独占锁：**

   共享锁：共享锁指的是我们同一把锁可以被多个线程同时获得。

   独占锁：把锁只能同时被一个线程获得。（读写锁中的读锁，是共享锁，而写锁是独占锁。读锁可以被同时读，可以同时被多个线程持有，而写锁最多只能同时被一个线程持有。）

4. **公平锁/非公平锁：**

   公平锁：如果线程现在拿不到这把锁，那么线程就都会进入等待，开始排队，在等待队列里等待时间长的线程会优先拿到这把锁，有先来先得的意思。

   非公平锁：它不那么“完美”了，它会在一定情况下，忽略掉已经在排队的线程，发生插队现象。

5. **悲观锁/乐观锁：**

   悲观锁：在获取资源之前，必须先拿到锁，以便达到“独占”的状态，当前线程在操作资源的时候，其他线程由于不能拿到锁，所以其他线程不能来影响我。

   乐观锁：和悲观锁恰恰相反，它并不要求在获取资源前拿到锁，也不会锁住资源；相反，乐观锁利用 CAS 理念，在不独占资源的情况下，完成了对资源的修改。

6. **自旋锁/非自旋锁：**

   自旋锁：如果线程现在拿不到锁，并不直接陷入阻塞或者释放 CPU 资源，而是开始利用循环，不停地尝试获取锁，这个循环过程被形象地比喻为“自旋”。

   非自旋锁：如果拿不到锁就直接放弃，或者进行其他的处理逻辑，例如去排队、陷入阻塞等。

7. **可中断锁/不可中断锁：**

可中断锁：ReentrantLock 是一种典型的可中断锁，例如使用 lockInterruptibly 方法在获取锁的过程中，突然不想获取了，那么也可以在中断之后去做其他的事情，不需要一直傻等到获取到锁才离开。

不可中断锁：在 Java 中，synchronized 关键字修饰的锁代表的是不可中断锁，一旦线程申请了锁，就没有回头路了，只能等到拿到锁以后才能进行其他的逻辑处理。

什么是悲观锁和乐观锁?

悲观锁：

每次获取并修改数据时，都把数据锁住，让其他线程无法访问该数据（这样读的线程不能操作）。假设线程 A 拿到了锁，并且正在操作同步资源，那么此时线程 B 就必须进行等待当线程 A 执行完毕后，CPU 才会唤醒正在等待这把锁的线程 B 再次尝试获取锁。

乐观锁：

乐观锁的实现一般都是利用 CAS 算法实现的。在同步之前，会先判断这个资源是否已经被其他线程所修改过。如果同步资源没有被其他线程修改更新，此时线程 A 就会去更新同步资源，完成修改的过程；如果同步资源已经被其他线程修改更新了，线程A会根据不同的业务逻辑去选择报错或者重试。(如果线程A一直通知被修改了，拿不到锁，会导致不停重试，开销很大。)

两种锁各自的使用场景：

悲观锁适合用于并发写入多、临界区代码复杂、竞争激烈等场景，这种场景下悲观锁可以避免大量的无用的反复尝试等消耗。

乐观锁适用于大部分是读取，少部分是修改的场景，也适合虽然读写都很多，但是并发并不激烈的场景。在这些场景下，乐观锁不加锁的特点能让性能大幅提高。

乐观锁比悲观锁快的原因：

当线程执行修改的时候，悲观锁不能执行读线程，因为线程已经锁死。而乐观锁可以执行读线程。

乐观锁的思想就是CAS的运用,下面是CAS算法介绍：

CAS有3个操作数，内存值V，预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

值A是之前读取到内存值V，值B是值A基础上计算出来的，当V不等于A，说明在刚才计算B的期间内，内存值V已经被修改了，那么CAS不应该再修改了，可以为了避免多人修改导致出错（保证多人修改的线程安全）。

CAS的缺点：

1. **ABA问题**：如果内存值原来是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的解决办法：使用版本号，在变量前面追加上版本号，比如每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A-2B-3A。

   扩展：**从Java1.5**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方法作用是首先检查当前引用是否等于预期引用，并且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。

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。
3. **只能保证一个共享变量的原子操作**：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。

   扩展：或者有一个取巧的办法，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Java1.5开始JDK提供了**AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。**

synchronized 背后的“monitor 锁”

synchronized介绍：

最简单的同步方式就是利用 synchronized 关键字来修饰代码块或者修饰一个方法，那么这部分被保护的代码，在同一时刻就最多只有一个线程可以运行，而 synchronized 的背后正是利用 monitor 锁实现的。这个锁也被称为内置锁或 monitor 锁，获得 monitor 锁的唯一途径就是进入由这个锁保护的同步代码块或同步方法，线程在进入被 synchronized 保护的代码块之前，会自动获取锁，并且**无论是正常路径退出，还是通过抛出异常退出，在退出的时候都会自动释放锁。**

同步代码块和同步方法获取和释放monitor 原理：

同步代码块：

```
public class SynTest {
    public void synBlock() {
        synchronized (this) {
            System.out.println("lagou");
        }
    }
}
```

同步代码块是利用 monitorenter（加锁） 和 monitorexit（释放锁）指令实现的。

> monitorexit：
> monitorexit 的作用是将 monitor 的计数器减 1，直到减为 0 为止。代表这个 monitor 已经被释放了，已经没有任何线程拥有它了，也就代表着解锁，所以，其他正在等待这个 monitor 的线程，此时便可以再次尝试获取这个 monitor 的所有权。

同步关键字底层加锁

- 同步关键字修饰方法：

```
public class SynchronizedTest {
    public synchronized void lockA() {
        System.out.println("lockA() execute.");
    }
}
```

flags中含有ACC_SYNCHRONIZED标识符，该标识符存在于同步方法的常量池中，表示这是一个同步方法，当线程开始访问方法lockA()时，会查看flags是否含有ACC_SYNCHRONIZED标识符，如果有就会去尝试获取监视器锁，获取成功就执行方法，否则就会被阻塞等待。值得注意的是，如果同步方法在执行过程中出现异常并且在内部没有处理，那么线程会被抛出方法之外时会**自动释放监视器锁。**

- 同步关键字修饰同步代码块：

```
public class SynchronizedTest {
    public void lockA() {
        syschroinzed(this) {
            System.out.println("lockA() execute.");
        }
    }
}
```

看到了monitorenter和monitorexit两条指令，它们分别代表获取和释放监视器锁。两条字节码指令都需要**显式指定**一个reference类型的加锁和解锁对象。在执行monitorenter指令过程中会尝试获取监视器锁，如果获取成功则**锁计数器自增1**，执行monitorexit指令时**锁计数器自减1**，计数器为0时锁会被释放。

总结：

1. synchronized修饰方法时**隐式指定**锁对象进行加锁，标识符为ACC_SYNCHRONIZED，在执行方法前需要获取锁对象。
2. synchronized修饰代码块时需要**显式指定**锁对象进行加锁，在执行monitorenter指令时需要先获取锁对象。
3. synchronized修饰的锁方法和同步代码块是**可重入**的，锁的计数器会**不断自增**，所以对同一个锁对象加锁多少次就要释放多少次。
4. synchronized修饰方法时会根据该方法是否是静态方法去判断获取的是Class对象锁还是**本类的实例对象锁。**

synchronized 和 Lock的相同点与区别?

相同点：

- synchronized 和 Lock 都是用来保护资源线程安全的。
- 都可以保证可见性（后面会讲解）
- synchronized 和 ReentrantLock 都拥有可重入的特点：

  可重入指的是某个线程如果已经获得了一个锁，现在试图再次请求这个它已经获得的锁，如果它无需提前释放这个锁，而是直接可以继续使用持有的这个锁，那么就是可重入的。如果必须释放锁后才能再次申请这个锁，就是不可重入的。

不同点：

- **用法区别：**

  synchronized 关键字可以加在方法上，不需要指定锁对象（此时的锁对象为 this），也可以新建一个同步代码块并且自定义 monitor 锁对象；而 Lock 接口必须显示用 Lock 锁对象开始加锁 lock() 和解锁 unlock()，并且一般会在 finally 块中确保用 unlock() 来解锁，以防发生死锁。

- **加解锁顺序不同：**

  Lock可以不按顺序加解锁，顺序可以自定义。

  synchronized 解锁的顺序和加锁的顺序必须完全相反，不能自定义。

- **synchronized 锁不够灵活：**

synchronized 锁已经被某个线程获得了，此时其他线程如果还想获得，那它只能被阻塞，直到持有锁的线程运行完毕或者发生异常从而释放这个锁。

Lock 类在等锁的过程中，如果使用的是 lockInterruptibly 方法，那么如果觉得等待的时间太长了不想再继续等待，可以中断退出，也可以用 tryLock() 等方法尝试获取锁，如果获取不到锁也可以做别的事，更加灵活。

- **synchronized 锁只能同时被一个线程拥有，但是 Lock 锁没有这个限制**

  例如在读写锁中的读锁，是可以同时被多个线程持有的，可是 synchronized 做不到。

- **原理区别：**

  Lock 根据实现不同，有不同的原理，例如 ReentrantLock 内部是通过 AQS 来获取和释放锁的。

- **是否可以设置公平/非公平：**

  公平锁是指多个线程在等待同一个锁时，根据先来后到的原则依次获得锁。ReentrantLock 等 Lock 实现类可以根据自己的需要来设置公平或非公平，synchronized 则不能设置。

- **性能区别：**

  在 Java 5 以及之前，synchronized 的性能比较低，但是到了 Java 6 以后，发生了变化，因为 JDK 对 synchronized 进行了很多优化，比如自适应自旋、锁消除、锁粗化、轻量级锁、偏向锁等，所以后期的 Java 版本里的 synchronized 的性能并不比 Lock 差。

如何选择Lock和synchronized：

1. 如果能不用最好不使用 Lock 。因为在许多情况下你可以使用 java.util.concurrent 包中的机制，它会为你处理所有的加锁和解锁操作，也就是推荐优先使用工具类来加解锁
2. 如果 synchronized 关键字适合你的程序， 那么请尽量使用它，这样可以减少编写代码的数量，减少出错的概率。因为一旦忘记在 finally 里 unlock，代码可能会出很大的问题，而使用 synchronized 更安全。
3. 如果特别需要 Lock 的特殊功能，比如尝试获取锁、可中断、超时功能等，才使用 Lock。

Lock 有哪几个常用方法？分别有什么用？

Lock 接口的各个方法：

```
1  public interface Lock {
2      //获取锁，线程获取锁时如果锁已被其他线程获取，则进行等待。
3      void lock();
4      //获取锁，除非当前线程在获取锁期间被中断，否则便会一直尝试获取直到获取到为止。
5      void lockInterruptibly() throws InterruptedException;
6      //用来尝试获取锁，如果当前锁没有被其他线程占用，则获取成功，返回 true，否则返回
   false，代表获取锁失败。相比于 lock()，这样的方法显然功能更强大，我们可以根据是否能获取到
   锁来决定后续程序的行为。
7      boolean tryLock();
8      //这个方法和 tryLock() 很类似，区别在于 tryLock(long time, TimeUnit unit) 方法会
   有一个超时时间，在拿不到锁时会等待一定的时间，如果在时间期限结束后，还获取不到锁，就会返回
   false；如果一开始就获取锁或者等待期间内获取到锁，则返回 true。
9      boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
10     //用于解锁，
11     void unlock();
12
13     Condition newCondition();
14 }
```

tryLock 和 lock 和 lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程，lock 不会抛出异常，而 lockInterruptibly 会抛出异常

公平锁和非公平锁

公平和非公平锁的实现代码（还有其他实现了类）：

```
new ReentrantLock(false);//非公平
new ReentrantLock(true); //公平
或
new ReentrantReadWriteLock(true);
new ReentrantReadWriteLock(false);
```

公平锁怎么实现

将当前线程结点加入等待队列之中，公平锁在锁释放后会严格按照等到队列去取后续值。

什么是公平和非公平锁：

公平锁指的是按照线程请求的顺序，来分配锁；而非公平锁指的是不完全按照请求的顺序，在一定情况下，可以允许插队。但需要注意这里的非公平并不是指完全的随机，不是说线程可以任意插队，而是仅仅“在合适的时机”插队。

非公平锁插队的时机：

当前请求获取锁时，恰巧前一个持有锁的线程释放了这把锁，那么当前锁可以不顾已经等待的线程立即插队（有些线程线程等待时间比较长，整体效率考虑，让等待时间短的先进行）。

公平和非公平的优缺点：

	优势	劣势
公平锁	各线程公平平等，每个线程在等待一段时间后，总有执行的机会	更慢，吞吐量更小
不公平锁	更快，吞吐量更大	有可能产生线程饥饿，也就是某些线程在长时间内，始终得不到执行

公平锁和非公平锁的源码对比：

公平锁在获取锁时多了一个限制条件：hasQueuedPredecessors() 为 false，这个方法就是判断在等待队列中是否已经有线程在排队了。

如果是公平锁，那么一旦已经有线程在排队了，当前线程就不再尝试获取锁；对于非公平锁而言，**无论是否已经有线程在排队，都会尝试获取一下锁，获取不到的话，再去排队。**

读写锁 ReadWriteLock 获取锁有哪些规则?

ReadWriteLock 的实现类ReentrantReadWriteLock 最主要的有两个方法：readLock() 和 writeLock() 用来获取读锁和写锁。

读锁和写锁的关系：

只有读锁线程在运行，另外一个线程也是申请读锁时直接获取，其他情况都要等待锁释放才能获取。

读写锁适用场合：

最后我们来看下读写锁的适用场合，相比于 ReentrantLock 适用于一般场合，ReadWriteLock 适用于读多写少的情况，合理使用可以进一步提高并发效率。

读写锁的升降级策略：

只能从写锁降级为读锁（在不释放写锁的情况下，直接获取读锁，这就是读写锁的降级），不能从读锁升级为写锁（在不释放读锁的情况下，直接获取写锁，这就是读写锁的升级）。

为什么不支持锁的升级?

我们知道读写锁的特点是如果线程都申请读锁，是可以多个线程同时持有的，可是如果是写锁，只能有一个线程持有，并且不可能存在读锁和写锁同时持有的情况。

正是因为不可能有读锁和写锁同时持有的情况，所以升级写锁的过程中，需要等到所有的读锁都释放，此时才能进行升级。

ReentrantReadWriteLock 读写锁

ReentrantReadWriteLock 可以设置为公平或者非公平，代码如下：

公平锁：

```
1 ReentrantReadWriteLock reentrantReadWriteLock = new ReentrantReadWriteLock(true);
```

公平锁的底层实现：

```
1 final boolean writerShouldBlock() {
2     return hasQueuedPredecessors();
3 }
4 final boolean readerShouldBlock() {
5     return hasQueuedPredecessors();
6 }
```

只要等待队列中有线程在等待，也就是 hasQueuedPredecessors() 返回 true 的时候，那么 writer 和 reader 都会 block，也就是一律不允许插队，都乖乖去排队，这也符合公平锁的思想。

非公平锁：

```
1 ReentrantReadWriteLock reentrantReadWriteLock = new ReentrantReadWriteLock(false);
```

非公平锁的底层实现：

```
final boolean writerShouldBlock() {
    return false; // writers can always barge
}
final boolean readerShouldBlock() {
    return apparentlyFirstQueuedIsExclusive();
}
```

在 writerShouldBlock() 这个方法中始终返回 false，可以看出，对于想获取写锁的线程而言，由于返回值是 false，所以写锁它是随时可以插队的，这就和我们的 ReentrantLock 的设计思想是一样的，但是读锁却不一样，读锁不允许插队。

允许写锁插队的原因：

因为写锁并不容易插队成功，写锁只有在当前没有任何其他线程持有读锁和写锁的时候，才能插队成功，同时写锁一旦插队失败就会进入等待队列，所以很难造成“饥饿”的情况，允许写锁插队是为了提高效率。

不允许读锁插队的原因：

因为一直有读请求插队的话，写请求就一直等待，为了防止饥饿，在等待队列的头结点是尝试获取写锁的线程的时候，不允许读锁插队。

ReentrantReadWriteLock 的升降级：

升降级策略：只能从写锁降级为读锁，不能从读锁升级为写锁。

写锁降级为读锁案例：

```
final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.writeLock().lock();
data = new Object();
//在不释放写锁的情况下，直接获取读锁，这就是读写锁的降级。（前提是写操作完才加该行代码）
rwl.readLock().lock();
```

为什么要降级：

写的地方只有一两个地方，写完我们就可以读，因为读没有线程安全问题，这样不影响效率。

读锁不能升级为写锁案例：

```
final static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
public static void main(String[] args) {
    upgrade();
}
public static void upgrade() {
    rwl.readLock().lock();
    System.out.println("获取到了读锁");
    rwl.writeLock().lock();
    System.out.println("成功升级");
}
```

这段代码会打印出“获取到了读锁”，但是却不会打印出“成功升级”，因为 ReentrantReadWriteLock 不支持读锁升级到写锁。

为什么不允许升级：

总结：

两个线程都需升级时会产生死锁，因为升级的前提必须其他线程释放读锁。

详解：

因为不可能有读锁和写锁同时持有的情况，所以升级写锁的过程中，需要等到所有的读锁都释放，此时才能进行升级。

假设有 A，B 和 C 三个线程，它们都已持有读锁。假设线程 A 尝试从读锁升级到写锁。那么它必须等待 B 和 C 释放掉已经获取到的读锁。如果随着时间推移，B 和 C 逐渐释放了它们的读锁，此时线程 A 确实是可以成功升级并获取写锁。

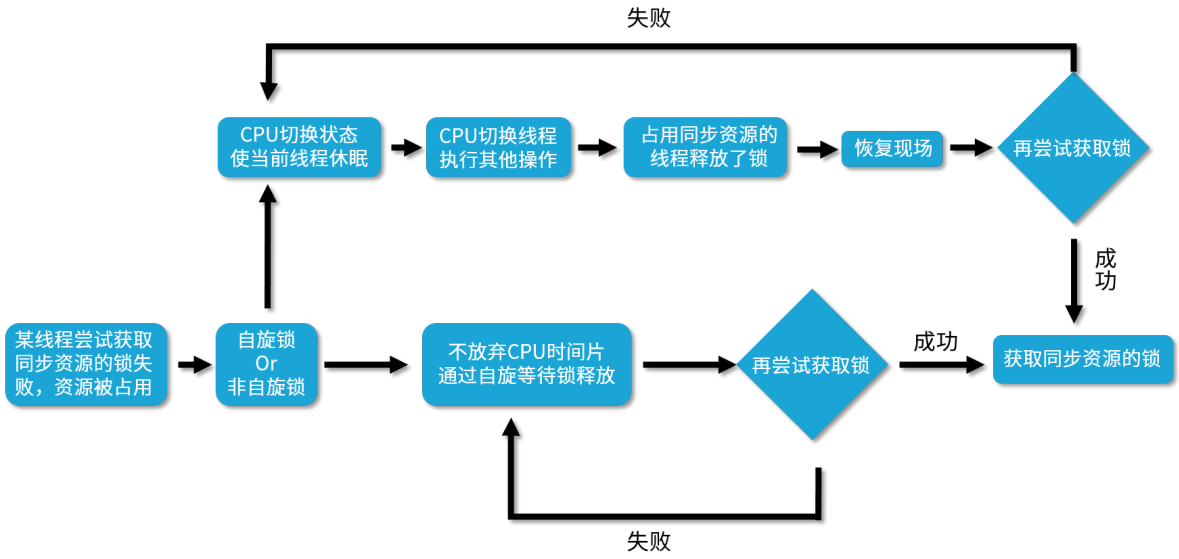
但是我们考虑一种特殊情况。假设线程 A 和 B 都想升级到写锁，那么对于线程 A 而言，它需要等待其他所有线程，包括线程 B 在内释放读锁。而线程 B 也需要等待所有的线程，包括线程 A 释放读锁。这就是一种非常典型的死锁的情况。谁都愿不愿意率先释放掉自己手中的锁。

但是读写锁的升级并不是不可能的，也有可以实现的方案，如果我们保证每次只有一个线程可以升级，那么就可以保证线程安全。只不过最常见的 ReentrantReadWriteLock 对此并不支持。

自旋锁和非自旋锁

自旋”就是自己在这里不停地循环，直到目标达成。而不像普通的锁那样，如果获取不到锁就进入阻塞。

对比自旋和非自旋的获取锁的流程



自己实现可重入自旋锁（其实是使用CAS实现的）：

```
1  public class ReentrantSpinLock  {
2
3      private AtomicReference<Thread> owner = new AtomicReference<>();
4
5      //重入次数
6      private int count = 0;
7
8      public void lock() {
9          Thread t = Thread.currentThread();
10         if (t == owner.get()) {
11             ++count;
```

```
            return;
        }
        //自旋获取锁（使用CAS来实现）
        while (!owner.compareAndSet(null, t)) {
            System.out.println("自旋了");
        }
    }

    public void unlock() {
        Thread t = Thread.currentThread();
        //只有持有锁的线程才能解锁
        if (t == owner.get()) {
            if (count > 0) {
                --count;
            } else {
                //此处无需CAS操作，因为没有竞争，因为只有线程持有者才能解锁
                owner.set(null);
            }
        }
    }
}
```

自旋锁的优缺点：

优点：

首先，阻塞和唤醒线程都是需要高昂的开销的，如果同步代码块中的内容不复杂，那么可能转换线程带来的开销比实际业务代码执行的开销还要大。

在很多场景下，可能我们的同步代码块的内容并不多，所以需要的执行时间也很短，如果我们仅仅为了这点时间就去切换线程状态，那么其实不如让线程不切换状态，而是让它自旋地尝试获取锁，等待其他线程释放锁，有时我只需要稍等一下，就可以避免上下文切换等开销，提高了效率。

用一句话总结自旋锁的好处，那就是自旋锁用循环去不停地尝试获取锁，让线程始终处于 Runnable 状态，节省了线程状态切换带来的开销。

缺点：

虽然避免了线程切换的开销，但是它在避免线程切换开销的同时也带来了新的开销，因为它需要不停得去尝试获取锁。如果这把锁一直不能被释放，那么这种尝试只是无用的尝试，会白白浪费处理器资源。也就是说，虽然一开始自旋锁的开销低于线程切换，但是随着时间的增加，这种开销也是水涨船高，后期甚至会超过线程切换的开销，得不偿失。

自旋锁的适用场景：

自旋锁适用于并发度不是特别高的场景，以及临界区比较短小的情况，这样我们可以利用避免线程切换来提高效率。

可是如果临界区很大，线程一旦拿到锁，很久才会释放的话，那就不合适用自旋锁，因为自旋会一直占用 CPU 却无法拿到锁，白白消耗资源。

JVM对锁的优化

相比于 JDK 1.5，在 JDK 1.6 中 HotSopt 虚拟机对 synchronized 内置锁的性能进行了很多优化，包括自适应的自旋、锁消除、锁粗化、偏向锁、轻量级锁等。有了这些优化措施后，synchronized 锁的性能得到了大幅提高，下面我们分别介绍这些具体的优化。

自适应的自旋锁：

JDK 1.6 中引入了自适应的自旋锁来解决长时间自旋的问题。自旋的时间不再固定，而是会根据最近自旋尝试的成功率、失败率，以及当前锁的拥有者的状态等多种因素来共同决定。

自旋的持续时间是变化的，自旋锁变“聪明”了。比如，如果最近尝试自旋获取某一把锁成功了，那么下一次可能还会继续使用自旋，并且允许自旋更长的时间；

但是如果最近自旋获取某一把锁失败了，那么可能会省略掉自旋的过程，以便减少无用的自旋，提高效率。

锁消除：

举例：如果发现某些对象不可能被其他线程访问到，那么就可以把它们当成栈上数据，栈上数据由于只有本线程可以访问，自然是线程安全的，也就无需加锁，所以会把这样的锁给自动去除掉。

例如，我们的 StringBuffer 的 append 方法如下所示：

```
@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}
```

从代码中可以看出，这个方法是被 synchronized 修饰的同步方法，因为它可能会被多个线程同时使用。

但是在大多数情况下，它只会在一个线程内被使用，如果编译器能确定这个 StringBuffer 对象只会在一个线程内被使用，就代表肯定是线程安全的，那么我们的编译器便会做出优化，把对应的 synchronized 给消除，省去加锁和解锁的操作，以便增加整体的效率。

锁粗化：

如下代码所示：

```java
public void lockCoarsening() {
    synchronized (this) {
        //do something
    }
    synchronized (this) {
        //do something
    }
    synchronized (this) {
        //do something
    }
}
```

上面频繁的创建和释放锁浪费性能，我们可以合为一个synchronized块（同步区域变大）。

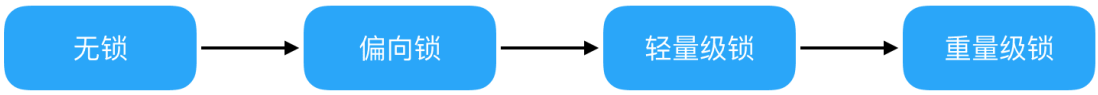
不过同步区域变大在循环不可以使用：

```java
for (int i = 0; i < 1000; i++) {
    synchronized (this) {
        //do something
    }
}
```

如果扩大同步区域的话，从第一次循环开始到最后一次循环结束，才结束代码块释放锁的话，导致其他线程无法获取锁。

锁粗化功能是默认打开的，用 -XX:-EliminateLocks 可以关闭该功能。

偏向锁/轻量级锁/重量级锁



JVM 默认会优先使用偏向锁，如果有必要的话才逐步升级，这大幅提高了锁的性能。

4种常用Java线程锁的特点，性能比较、使用场景：

四种常见的线程锁：synchronized、ReentrantLock、Semaphore、AtomicInteger

1. **synchronized：**

   在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好。

2. **ReentrantLock：**

   在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍，而ReentrantLock确还能维持常态。高并发量情况下使用ReentrantLock。

3. **Atomic：**

和上面的类似，不激烈情况下，性能比synchronized略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic的性能会优于ReentrantLock一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个Atomic的变量，多于一个同步无效。因为他不能在多个Atomic之间同步。所以，我们写同步的时候，优先考虑synchronized，如果有特殊需要，再进一步优化。ReentrantLock和Atomic如果用的不好，不仅不能提高性能，还可能带来灾难。

4. **Semaphore：**

Semaphore基本能完成ReentrantLock的所有工作，使用方法也与之类似，通过acquire()与release()方法来获得和释放临界资源。

经实测，Semaphone.acquire()方法默认为可响应中断锁，与ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被Thread.interrupt()方法中断。

此外，Semaphore也实现了可轮询的锁请求与定时锁的功能，除了方法名tryAcquire与tryLock不同，其使用方法与ReentrantLock几乎一致。Semaphore也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore的锁释放操作也由手动进行，因此与ReentrantLock一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在finally代码块中完成。

Semaphore的主要方法摘要：

void acquire():从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。

void release():释放一个许可，将其返回给信号量。

int availablePermits():返回此信号量中当前可用的许可数。

boolean hasQueuedThreads():查询是否有线程正在等待获取。

```
public class SemaphoreDemo {

    static class TaskThread extends Thread {

        Semaphore semaphore;

        public TaskThread(Semaphore semaphore) {
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                semaphore.acquire();
                System.out.println(getName() + " acquire");
                Thread.sleep(1000);
                semaphore.release();
                System.out.println(getName() + " release ");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        int threadNum = 5;
        Semaphore semaphore = new Semaphore(2);
        for (int i = 0; i < threadNum; i++) {
            new TaskThread(semaphore).start();
        }
    }
```

```
32
33 }
```

多线程有用到锁吗

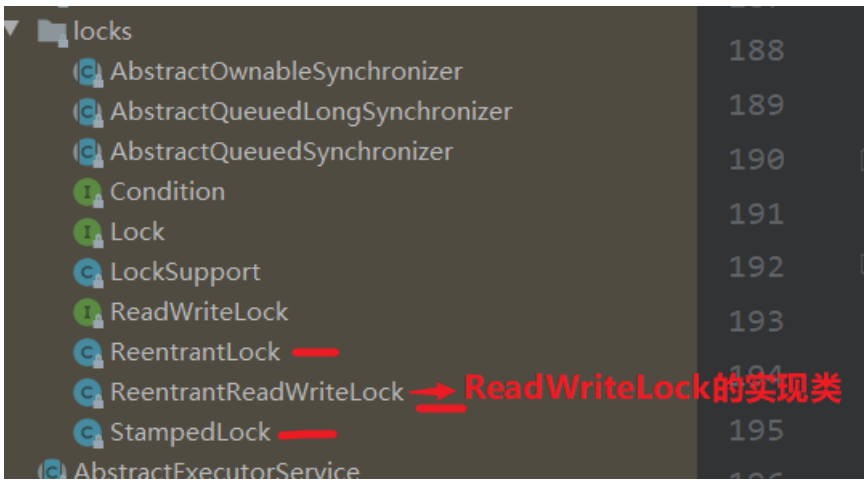
锁的分类：

> 根据不同的标准锁可以分为以下7个类别：
>
> 1. **偏向锁/轻量级锁/重量级锁：**
>
>   偏向锁：如果自始至终，对于这把锁都不存在竞争，那么其实就没必要上锁，只需要打个标记就行了，这就是偏向锁的思想。
>
>   轻量级锁：JVM 开发者发现在很多情况下，synchronized 中的代码是被多个线程交替执行的，而不是同时执行的，也就是说并不存在实际的竞争，或者是只有短时间的锁竞争，用 CAS 就可以解决，这种情况下，用完全互斥的重量级锁是没必要的。轻量级锁是指当锁原来是偏向锁的时候，被另一个线程访问，说明存在竞争，那么偏向锁就会升级为轻量级锁，线程会通过自旋的形式尝试获取锁，而不会陷入阻塞。
>
>   重量级锁：重量级锁是互斥锁，它是利用操作系统的同步机制实现的，所以开销相对比较大。当多个线程直接有实际竞争，且锁竞争时间长的时候，轻量级锁不能满足需求，锁就会膨胀为重量级锁。重量级锁会让其他申请却拿不到锁的线程进入阻塞状态。
>
>   
>
>
>   偏向锁性能最好，可以避免执行 CAS 操作。而轻量级锁利用自旋和 CAS 避免了重量级锁带来的线程阻塞和唤醒，性能中等。重量级锁则会把获取不到锁的线程阻塞，性能最差。
>
> 2. **可重入锁/非可重入锁：**
>
>   可重入锁：同一个线程可以多次获取同一把锁。
>
>   非可重入锁：指当前线程已经持有这把锁，如果再次获取这把锁，必须要先释放锁后才能再次尝试获取。
>
>   对于可重入锁而言，最典型的就是 ReentrantLock 了，正如它的名字一样，reentrant 的意思就是可重入，它也是 Lock 接口最主要的一个实现类。
>
>   **可重入锁原理：**
>
>   每个锁关联线程持有者和计数器，某一线程请求成功后，JVM会记下锁的持有线程，计数器+1，请求线程请求该锁，必须等待，该线程再次拿到这个锁，计数器会递增，线程退出同步代码块时，计数器会递减，如果计数器为 0，则释放该锁。
>
> 3. **共享锁/独占锁：**
>
>   共享锁：共享锁指的是我们同一把锁可以被多个线程同时获得。
>
>   独占锁：把锁只能同时被一个线程获得。（读写锁中的读锁，是共享锁，而写锁是独占锁。读锁可以被同时读，可以同时被多个线程持有，而写锁最多只能同时被一个线程持有。）
>
> 4. **公平锁/非公平锁：**
>
>   公平锁：如果线程现在拿不到这把锁，那么线程就都会进入等待，开始排队，在等待队列里等待时间长的线程会优先拿到这把锁，有先来先得的意思。
>
>   非公平锁：它不那么“完美”了，它会在一定情况下，忽略掉已经在排队的线程，发生插队现象。
>
> 5. **悲观锁/乐观锁：**
>
>   悲观锁：在获取资源之前，必须先拿到锁，以便达到“独占”的状态，当前线程在操作资源的时候，其他线程由于不能拿到锁，所以其他线程不能来影响我。

> 乐观锁：和悲观锁恰恰相反，它并不要求在获取资源前拿到锁，也不会锁住资源；相反，乐观锁利用 CAS 理念，在不独占资源的情况下，完成了对资源的修改。
>
> 6. **自旋锁/非自旋锁：**
>
>   自旋锁：如果线程现在拿不到锁，并不直接陷入阻塞或者释放 CPU 资源，而是开始利用循环，不停地尝试获取锁，这个循环过程被形象地比喻为“自旋”。
>
>   非自旋锁：如果拿不到锁就直接放弃，或者进行其他的处理逻辑，例如去排队、陷入阻塞等。
>
> 7. **可中断锁/不可中断锁：**
>
>   可中断锁：ReentrantLock 是一种典型的可中断锁，例如使用 lockInterruptibly 方法在获取锁的过程中，突然不想获取了，那么也可以在中断之后去做其他的事情，不需要一直傻等到获取到锁才离开。
>
>   不可中断锁：在 Java 中，synchronized 关键字修饰的锁代表的是不可中断锁，一旦线程申请了锁，就没有回头路了，只能等到拿到锁以后才能进行其他的逻辑处理。

```
final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
 rwl.readLock().lock();
 rwl.readLock().unlock();
```

以下几种锁：



- **ReentrantLock：**通过设置构造函数

```
final ReentrantLock lock = new ReentrantLock(false); //非公平锁（默认）
final ReentrantLock lock = new ReentrantLock(true); //公平锁
```

ReentrantLock：又称重入锁、独享锁、互斥锁

- **ReadWriteLock：**又称共享锁、读写锁

```
final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().lock(); //读锁
rwl.writeLock().lock(); //写锁
```

线程进入读锁的前提条件：

没有其他线程的写锁。

没有写请求或者有写请求，但调用线程和持有锁的线程是同一个。

线程进入写锁的前提条件：

没有其他线程的读锁。

没有其他线程的写锁。

而读写锁有以下三个重要的特性：

（1）公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。

（2）重进入：读锁和写锁都支持线程重进入。

（3）锁降级：遵循获取写锁、获取读锁再释放写锁的次序，**写锁能够降级成为读锁。**

写锁什么时候会降级成读锁？

比如事务执行要10秒，写操作占用1秒，其他的都是读操作9秒，那么执行完1秒的写锁后面降级为读锁。

为什么不支持锁的升级？

我们知道读写锁的特点是如果线程都申请读锁，是可以多个线程同时持有的，可是如果是写锁，只能有一个线程持有，并且不可能存在读锁和写锁同时持有的情况。

正是因为不可能有读锁和写锁同时持有的情况，所以升级写锁的过程中，需要等到所有的读锁都释放，此时才能进行升级。

- **StampedLock：**StampedLock是为了优化可重入读写锁性能的一个锁实现工具，jdk8开始引入相比于普通的ReentranReadWriteLock主要多了一种乐观读的功能在API上增加了stamp的入参和返回值。

  StampedLock控制锁有三种模式（写，悲观读，乐观读），比ReadWriteLock多一个乐观读。

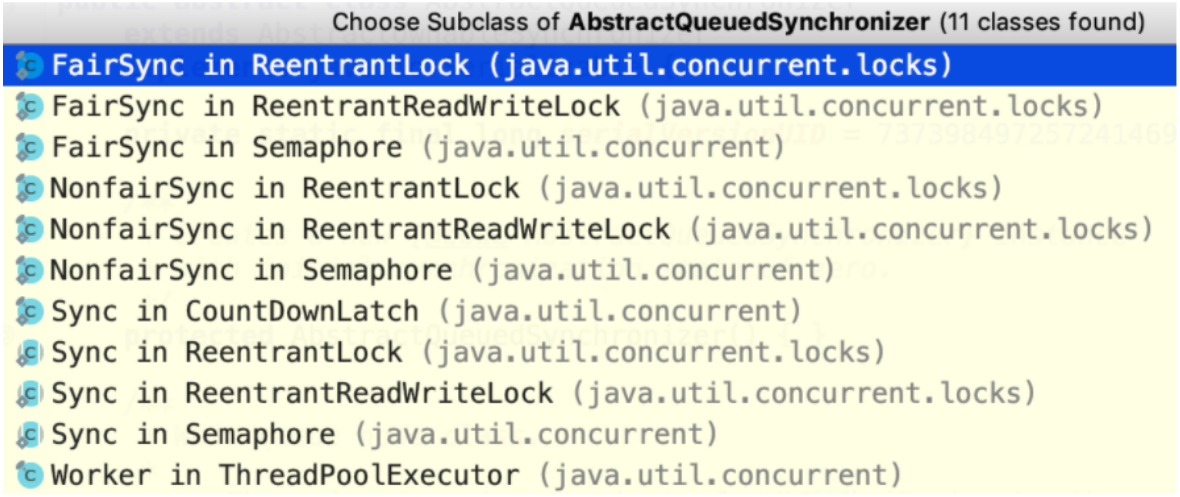
  乐观读：

  相对悲观读不会阻塞写锁。

  乐观读实现：

  先我们通过`tryOptimisticRead()`获取一个乐观读锁，并返回版本号。接着进行读取，读取完成后，我们通过`validate()`去验证**版本号**，如果在读取过程中没有写入，版本号不变，验证成功，我们就可以放心地继续后续操作。如果在读取过程中有写入，版本号会发生变化，验证将失败。在失败的时候，我们再通过获取悲观读锁再次读取。

锁的底层AQS



如图所示，AQS 在 **ReentrantLock**、**ReentrantReadWriteLock**、**Semaphore**、**CountDownLatch**、**ThreadPoolExcutor 的 Worker** 中都有运用（JDK 1.8），AQS 是这些类的底层原理。

而以上这些类，很多都是我们经常使用的类，大部分我们在前面课时中也已经详细介绍过，所以说 JUC 包里很多重要的工具类背后都离不开 AQS 框架，因此 AQS 的重要性不言而喻。

AQS 的作用：

AQS 是一个用于构建锁、同步器等线程协作工具类的框架，有了 AQS 以后，很多用于线程协作的工具类就都可以很方便的被写出来，有了 AQS 之后，可以让更上层的开发极大的减少工作量，避免重复造轮子，同时也避免了上层因处理不当而导致的线程安全问题，因为 AQS 把这些事情都做好了。总之，有了 AQS 之后，我们构建线程协作工具类就容易多了。

AQS 的底层实现：

AQS 最核心的三大部分来实现：

- 状态
- 队列
- 获取/释放方法

1. 状态：

如果我们的 AQS 想要去管理或者想作为协作工具类的一个基础框架，那么它必然要管理一些状态，而这个状态在 AQS 内部就是用 state 变量去表示的。它的定义如下：

```
/**
 * The synchronization state.
 */
private volatile int state;
```

而 state 的含义并不是一成不变的，它会**根据具体实现类的作用不同而表示不同的含义**，下面举几个例子。

比如说在信号量里面，state 表示的是剩余**许可证的数量**。如果我们最开始把 state 设置为 10，这就代表许可证初始一共有 10 个，然后当某一个线程取走一个许可证之后，这个 state 就会变为 9，所以信号量的 state 相当于是一个内部计数器。

再比如，在 CountDownLatch 工具类里面，state 表示的是**需要“倒数”的数量**。一开始我们假设把它设置为 5，当每次调用 CountDown 方法时，state 就会减 1，一直减到 0 的时候就代表这个门闩被放开。

下面我们再来看一下 state 在 ReentrantLock 中是什么含义，在 ReentrantLock 中它表示的是**锁的占有情况**。最开始是 0，表示没有任何线程占有锁；如果 state 变成 1，则就代表这个锁已经被某一个线程所持有了。

那为什么还会变成 2、3、4 呢？为什么会往上加呢？因为 ReentrantLock 是可重入的，同一个线程可以再次拥有这把锁就叫**重入**。如果这个锁被同一个线程多次获取，那么 state 就会逐渐的往上加，state 的值表示重入的次数。在释放的时候也是逐步递减，比如一开始是 4，释放一次就变成了 3，再释放一次变成了 2，这样进行的减操作，即便是减到 2 或者 1 了，都不代表这个锁是没有任何线程持有，只有当它减到 0 的时候，此时恢复到最开始的状态了，则代表现在没有任何线程持有这个锁了。所以，state 等于 0 表示锁不被任何线程所占有，代表这个锁当前是处于释放状态的，其他线程此时就可以来尝试获取了。

这就是 state 在不同类中不同含义的一个具体表现。我们举了三个例子，如果未来有新的工具要利用到 AQS，它一定也需要利用 state，为这个类表示它所需要的业务逻辑和状态。

下面我们再来看一下关于 state 修改的问题，因为 state 是会被多个线程共享的，会被并发地修改，所以所有去修改 state 的方法都必须要保证 state 是线程安全的。可是 state 本身它仅仅是被 volatile 修饰的，volatile 本身并不足以保证线程安全，所以我们就来看一下，AQS 在修改 state 的时候具体利用了什么样的设计来保证并发安全。

我们举两个和 state 相关的方法，分别是 compareAndSetState 及 setState，它们的实现已经由 AQS 去完成了，也就是说，我们直接调用这两个方法就可以对 state 进行线程安全的修改。下面就来看一下这两个方法的源码是怎么实现的。

- 先来看一下 compareAndSetState 方法，这是一个我们非常熟悉的 CAS 操作，这个方法的代码，如下所示：

```
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

方法里面只有一行代码，即 return unsafe.compareAndSwapInt(this, stateOffset, expect, update)，这个方法我们已经非常熟悉了，它利用了 Unsafe 里面的 CAS 操作，利用 CPU 指令的原子性保证了这个操作的原子性，与之前介绍过的原子类去保证线程安全的原理是一致的。

- 接下来看一下 setState 方法的源码，如下所示：

```
protected final void setState(int newState) {
    state = newState;
}
```

我们可以看到，它去修改 state 值的时候非常直截了当，直接把 state = newState，这样就直接赋值了。你可能会感到困惑，这里并没有进行任何的并发安全处理，没有加锁也没有 CAS，那如何能保证线程安全呢?

这里就要说到 volatile 的作用了，前面在学习 volatile 关键字的时候，知道了它适用于两种场景，其中一种场景就是，当**对基本类型的变量进行直接赋值时**，如果加了 volatile 就可以保证它的线程安全。注意，这是 volatile 的非常典型的使用场景。

```
/**
 * The synchronization state.
 */
private volatile int state;
```

可以看出，state 是 int 类型的，属于基本类型，并且这里的 setState 方法内是对 state 直接赋值的，它不涉及读取之前的值，也不涉及在原来值的基础上再修改，所以我们仅仅利用 volatile 就可以保证在这种情况下的并发安全，这就是 setState 方法线程安全的原因。

下面我们对 state 进行总结，在 AQS 中有 state 这样的一个属性，是被 volatile 修饰的，会被并发修改，它代表当前工具类的某种状态，在不同的类中代表不同的含义。

2. **队列：**

下面我们再来看看 AQS 的第二个核心部分，**FIFO 队列**，即先进先出队列，这个队列最主要的作用是存储等待的线程。假设很多线程都想要同时抢锁，那么大部分的线程是抢不到的，那怎么去处理这些抢不到锁的线程呢？就得需要有一个队列来存放、管理它们。所以 AQS 的一大功能就是充当线程的“**排队管理器**”。

当多个线程去竞争同一把锁的时候，就需要用**排队机制**把那些没能拿到锁的线程串在一起；而当前面的线程释放锁之后，这个管理器就会挑选一个合适的线程来尝试抢刚刚释放的那把锁。所以 AQS 就一直在维护这个队列，并把等待的线程都放到队列里面。

这个队列内部是双向链表的形式，其数据结构看似简单，但是要想维护成一个线程安全的双向队列却非常复杂，因为要考虑很多的多线程并发问题。我们来看一下 AQS 作者 Doug Lea 给出的关于这个队列的一个图示：

s

在队列中，分别用 head 和 tail 来表示头节点和尾节点，两者在初始化的时候都指向了一个空节点。头节点可以理解为“当前持有锁的线程”，而在头节点之后的线程就被阻塞了，它们会等待被唤醒，唤醒也是由 AQS 负责操作的。

3. **获取/释放方法：**

下面我们就来看一看 AQS 的第三个核心部分，获取/释放方法。在 AQS 中除了刚才讲过的 state 和队列之外，还有一部分非常重要，那就是**获取和释放相关的重要方法**，这些方法是协作工具类的**逻辑**的**具体体现**，需要每一个协作工具类**自己去实现**，所以在不同的工具类中，它们的实现和含义各不相同。

获取方法

我们首先来看一下获取方法。获取操作通常会依赖 state 变量的值，根据 state 值不同，协作工具类也会有不同的逻辑，并且在获取的时候也经常会阻塞，下面就让我们来看几个具体的例子。

比如 ReentrantLock 中的 lock 方法就是其中一个“获取方法”，执行时，如果发现 state 不等于 0 且当前线程不是持有锁的线程，那么就代表这个锁已经被其他线程所持有了。这个时候，当然就获取不到锁，于是就让该线程进入阻塞状态。

再比如，Semaphore 中的 acquire 方法就是其中一个“获取方法”，作用是获取许可证，此时能不能获取到这个许可证也取决于 state 的值。如果 state 值是正数，那么代表还有剩余的许可证，数量足够的话，就可以成功获取；但如果 state 是 0，则代表已经没有更多的空余许可证了，此时这个线程就获取不到许可证，会进入阻塞状态，所以这里同样也是和 state 的值相关的。

再举个例子，CountDownLatch 获取方法就是 await 方法（包含重载方法），作用是“等待，直到倒数结束”。执行 await 的时候会判断 state 的值，如果 state 不等于 0，线程就陷入阻塞状态，直到其他线程执行倒数方法把 state 减为 0，此时就代表现在这个门闩放开了，所以之前阻塞的线程就会被唤醒。

我们总结一下，“获取方法”在不同的类中代表不同的含义，但往往**和 state 值相关**，也经常会让线程进入**阻塞**状态，这也同样证明了 state 状态在 AQS 类中的重要地位。

释放方法

释放方法是站在获取方法的对立面的，通常和刚才的获取方法配合使用。我们刚才讲的获取方法可能会让线程阻塞，比如说获取不到锁就会让线程进入阻塞状态，但是释放方法通常是**不会阻塞线程**的。

比如在 Semaphore 信号量里面，释放就是 release 方法（包含重载方法），release() 方法的作用是去释放一个许可证，会让 state 加 1；而在 CountDownLatch 里面，释放就是 countDown 方法，作用是倒数一个数，让 state 减 1。所以也可以看出，在不同的实现类里面，他们对于 state 的操作是截然不同的，需要由每一个协作类根据自己的逻辑去具体实现。

总结

本课时我们介绍了 AQS 最重要的三个部分。第一个是 state，它是一个数值，在不同的类中表示不同的含义，往往代表一种状态；第二个是一个队列，该队列用来存放线程；第三个是“获取/释放”的相关方法，需要利用 AQS 的工具类根据自己的逻辑去实现。