07-并发编程

刘亚雄

极客时间-Java 讲师

二、线程同步：Synchronized

2.1 synchronized简介

保证方法或代码块在多线程环境运行时，同一个时刻只有一个线程执行代码块。

- JDK1.6之前，synchronized的实现依赖于OS底层互斥锁的MutexLock，存在严重的性能问题
- JDK1.6之后，Java对synchronized进行的了一系列优化，实现方式也改为Monitor（管程）了
- 一句话：有了Synchronized，就线程安全了，保证原子性、可见性、有序性

可以修饰方法（静态和非静态）和代码块：

- 同步代码块的锁：当前对象，字节码对象，其他对象
- 非静态同步方法：锁当前对象
- 静态同步方法：锁是当前类的Class对象

2.2 synchronized原理剖析

01-如何解决可见性问题？Happens-before规则

JMM对于Synchronized的规定：

- **加锁前：**必须把自己本地内存中共享变量的最新值刷到主内存
- **加锁时：**清空本地内存中的共享变量，从主内存中读取共享变量最新的值

使用Synchronized解决可见性案例中的问题

02-Synchronized是如何实现同步的呢？

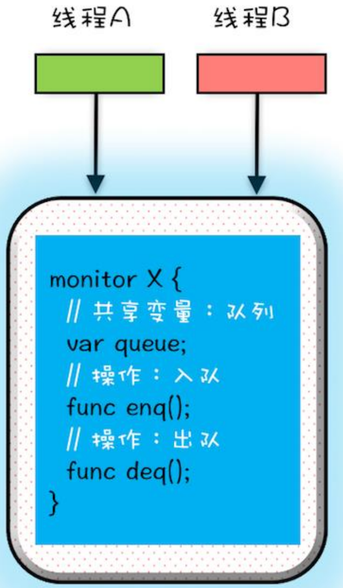
同步操作主要是monitorenter和monitorexit两个jvm指令实现。背后原理是Monitor（管程）

什么是个Monitor呢？

2.2 synchronized原理剖析

03-什么是Monitor呢？

- Monitor意译为管程，直译为监视器。所谓管程，就是**管**理共享变量及对共享变量操作的过**程**。让这个过程可以并发执行。
- **Java所有对象都可以做为锁，为什么？**
- 因为每个对象都都有一个Monitor与之关联。然后线程对monitor执行lock和unlock操作，相当于对对象执行上锁和解锁操作。
- Synchronized里面不可以直接使用lock和unlock方法，但当我们使用了synchroni动加入两个指令monitorenter和monitorexit，对应的就是lock和unlock操作。



04-Monitor的实现原理：将共享变量和对共享变量的操作统一封装起来

使用管程了，问题就解决了吗？

- 极客时间专栏《Java并发编程实战-王宝令》管程：并发编程的万能钥匙文章

2.2 synchronized原理剖析

05-锁优化

- 加了锁之后，不一定就是好的，很多程序员功力不够，盲目使用Synchronized，虽然解决了线程安全问题，但也给系统埋下了迟缓的种子。
- **并发编程的几种情况：**①只有一个线程运行，②两个线程交替执行，③多个线程并发执行
- 经过实践经验总结：前两种情况，可以针对性优化
- JDK1.6基于这两个场景，设计了两种优化方案：**偏向锁和轻量级锁**
- **同步锁一共有四个状态：无锁，偏向锁，轻量级锁，重量级锁**
- **JVM会视情况来逐渐升级锁，而不是上来就加重量级锁，这就是JDK1.6的锁优化**

06-偏向锁：只有一个线程访问锁资源，偏向锁就会把整个同步措施消除

07-轻量级锁：只有两个线程交替竞争锁资源，如果线程竞争锁失败了不立即挂起，而是让它飞一会（自旋），在等待过程中可能锁就会被释放出来，这时尝试重新获取锁

请问，锁信息存储在哪里？

三、volatile关键字

3.1 Volatile简介

01-Java语言对volatile的定义：

- Java语言允许线程访问共享变量，为了确保共享变量能被准确的一致地更新，线程应该确保通过互斥锁单独获取这个变量。Java语言提供了volatile，在某些情况下，它比锁要更方便。如果一个变量被声明成volatile，JMM确保所有线程看到这个变量的值是一致的。

一句话：volatile可以保证多线程场景下**共享变量的可见性、有序性。**

- **可见性：**保证对此共享变量的修改，所有线程的可见性
- **有序性：**禁止指令重排序的优化，遵循JMM的happens-before规则

使用volatile解决可见性案例中的问题

Volatile是如何实现可见性的？

3.2 Volatile实现原理剖析

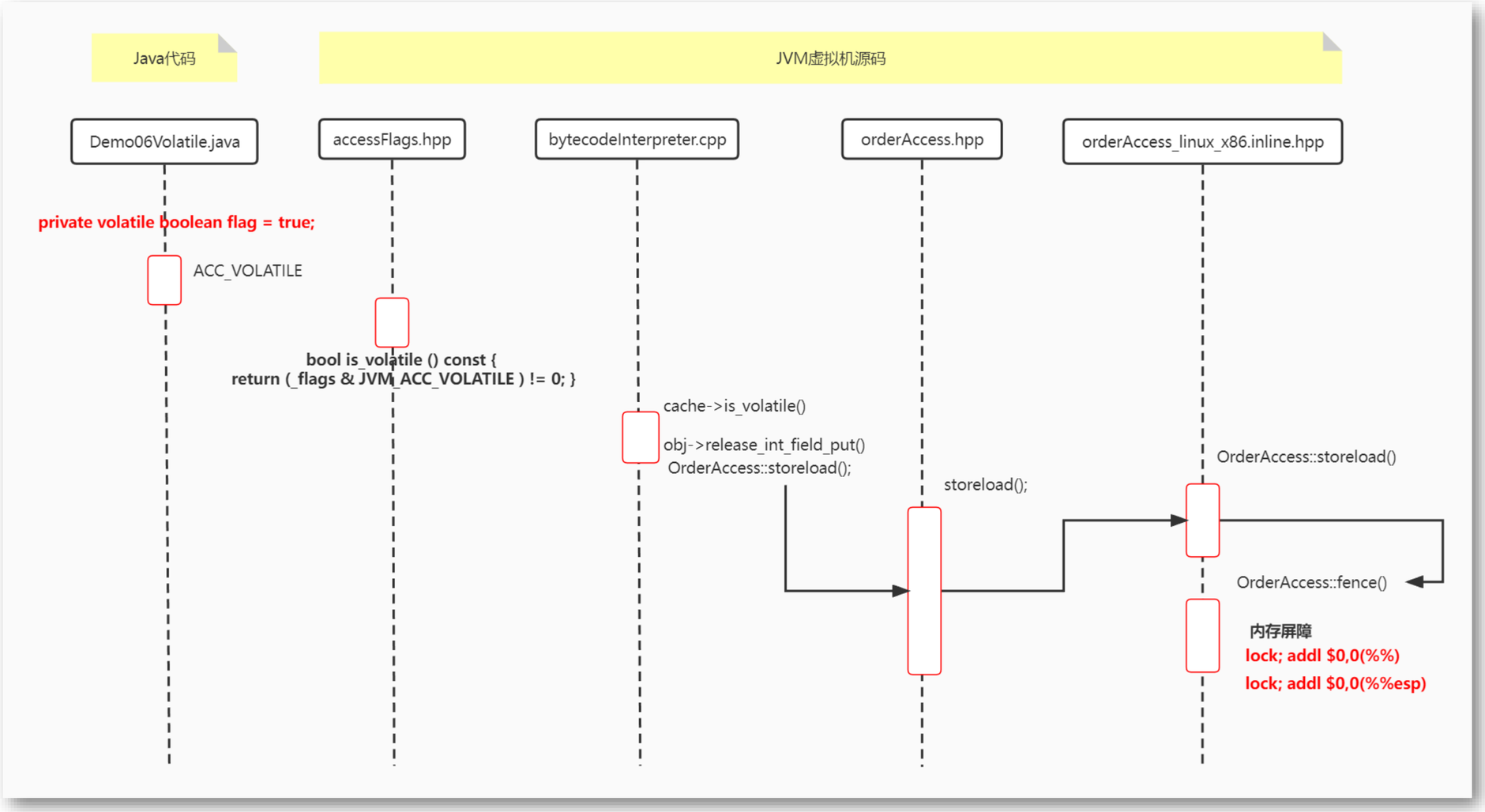
01-volatile实现内存可见性原理：内存屏障

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编译器会根据内存屏障的规则禁止重排序。

- Volatile变量**写操作**时：在**写操作后**加一条**store屏障指令**，让本地内存中变量的值能够刷新到主内存
- Volatile变量**读操作**时：在**读操作前**加一条**load屏障指令**，及时读取到变量在主内存的值

来我们从源码角度看一下Volatile实现的原理！

3.2 Volatile实现原理剖析-图解



3.2 Volatile实现原理剖析

02- JMM 内存屏障插入策略

- 在每个 volatile 写前，插入StoreStore 屏障
- 在每个 volatile 写后，插入StoreLoad 屏障
- 在每个 volatile 读后，插入LoadLoad 屏障
- 在每个 volatile 读后，插入LoadStore 屏障

屏障类型	示例	说明
StoreStore	S01，**StoreStore**，S02	确保S01刷新数据到内存，先于S02及其后所有Store操作，对屏障前后的Load无影响
StoreLoad	S01，**StoreLoad**，L02	**全能型屏障：**会屏蔽屏障前后所有指令的重排
LoadLoad	L01，**LoadLoad**，L02	确保load动作L01，先于L02及其后所有Load操作，对屏障前后Store无影响
LoadStore	L01，**LoadStore**，S02	确保指令前的所有load操作，先于屏障后所有Store操作

3.2 Volatile缺陷

存在原子性的问题：虽然volatile可以保证可见性，但是不能满足原子性

volatile适合使用场景：

共享变量独立于其他变量和自己之前的值，这类变量单独使用的时候适合用volatile

- 对共享变量的写入操作不依赖其当前值：例如++和--，就不行
- 共享变量没有包含在有其他变量的不等式中

Volatile和Synchronized特点比较：

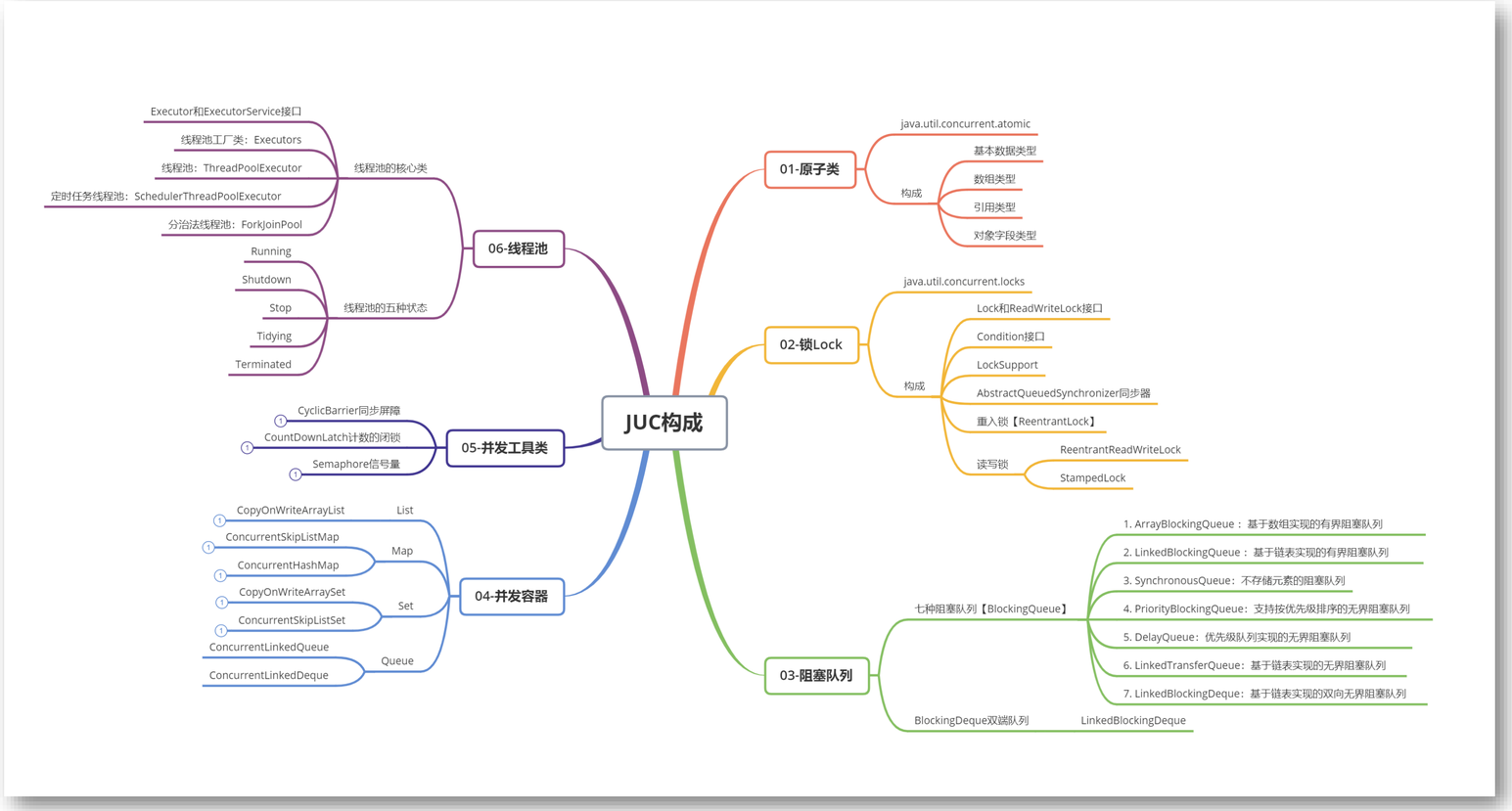
特点	Volatile	Synchronized
加锁	否	是
阻塞线程	否	是
保证原子性	否	是
保证可见性	是	是
性能	很好	很差

四、原子类与CAS

4.1 JUC简介

从JDK1.5起，Java API 中提供了java.util.concurrent（简称JUC）包，在此包中定义了并发编程中很常用的工具。

JUC是 JSR 166 标准规范的一个实现，JSR 166 以及 JUC 包的作者是同一个人 Doug Lea 。



4.2 Atomic包

01-什么是原子类？

JDK1.5之后，JUC的atomic包中，提供了一系列**用法简单、性能高效、线程安全的更新一个变量的类**，这些称之为原子类。

作用：保证共享变量操作的原子性、可见性，可以解决volatile原子性操作变量的BUG

02-AtomicInteger主要API如下：

```
get()                //直接返回值
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
```

用AtomicInteger解决可见性案例中的问题

4.2 Atomic包

03-Atomic包里的类：

- 基本类型：AtomicInteger整形原子类...
- 引用类型：AtomicReference引用类型原子类...
- 数组类型： AtomicIntegerArray整形数组原子类...
- 对象属性修改类型：AtomicIntegerFieldUpdater原子更新整形字段的更新器...
- JDK1.8新增：DoubleAdder双浮点型原子类、LongAdder长整型原子类...

虽然原子类很多，但原理几乎都差不多，其核心是采用CAS进行原子操作

那么，接下来，咱们看一下CAS的原理

4.3 CAS（compare and swap）

01-CAS是什么？

CAS即**compare and swap（比较再替换）**，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、Atomic原子类底层操作，都可以看见CAS。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。

其实，CAS本不难，它只是一个方法而已，这个方法长这样：**执行函数：CAS(V,E,N)**

- V：要读写的内存地址
- E：进行比较的值 （预期值）
- N：拟写入的新值
- 当且仅当 **内存地址的V** 中的值等于 **预期值E** 时，将**内存地址的V**中的值改为N，否则会进行自旋操作，即不断的重试。

CAS本质是一条CPU的原子指令，可以保证共享变量修改的原子性。

究竟是不是这样的呢？来，看源码

4.3 CAS（compare and swap）

02-CAS的缺陷

CAS虽然很好的解决了共享变量的原子操作问题，但还是有一些缺陷：

- **循环时间不可控：**如果CAS一直不成功，那么CAS自旋就是个死循环。会给CPU造成负担
- **只能保证一个共享变量原子操作**
- **ABA问题：**CAS检查操作的值有没有发生改变，如果没有则更新。这就存在一种情况：如果原来的值是A，然后变成了B，然后又变为A了，那么CAS检测不到数据发生了变化，但是其实数据已经改变了。

五、锁与AQS

5.1 Java锁简介

01-JUC包提供了种类丰富的锁，每种锁特性各不相同

- **ReentrantLock重入锁：**它具有与使用 synchronized 相同的一些基本行为和语义，但是它的API功能更强大，重入锁相当于synchronized 的增强版，具有synchronized很多所没有的功能。它是一种**独享锁（互斥锁）**，可以是**公平锁**，也可以是**非公平的锁**。
- **ReentrantReadWriteLock读写锁：**它维护了一对锁，ReadLock读锁和WriteLock写锁。读写锁适合读多写少的场景。基本原则：**读锁可以被多个线程同时持有进行访问，而写锁只能被一个线程持有。**可以这么理解：读写锁是个混合体，它既是一个共享锁，也是一个独享锁。
- **StampedLock重入读写锁**，JDK1.8引入的锁类型，是对读写锁ReentrantReadWriteLock的增强版。

看到这么多锁的类型描述，是不是有点混乱了？

5.2 Java锁分类

01-按上锁方式划分

① **隐式锁：synchronized，**不需要显示加锁和解锁

② **显式锁：JUC包中提供的锁，**需要显示加锁和解锁

02-按特性划分

- **2.1 悲观锁/乐观锁：按照线程在使用共享资源时，要不要锁住同步资源，划分为悲观锁和乐观锁**
  - 悲观锁：JUC锁，synchronized
  - 乐观锁：CAS，关系型数据库的版本号机制
- **2.2 重入锁/不可重入锁：按照同一个线程是否可以重复获取同一把锁，划分为重入锁和不可重入锁**
  - 重入锁：ReentrantLock、synchronized
  - 不可重入锁：不可重入锁，与可重入锁相反，线程获取锁之后不可重复获取锁，重复获取会发生死锁

5.2 Java锁分类

02-按特性划分

- **2.3 公平锁/非公平锁：按照多个线程竞争同一锁时需不需要排队，能不能插队，划分为公平锁和非公平锁。**
  - 公平锁：new ReentrantLock(true)多个线程按照申请锁的顺序获取锁
  - 非公平锁：new ReentrantLock(false)多个线程获取锁的顺序不是按照申请锁的顺序(可以插队) synchronized
- **2.4 独享锁/共享锁：按照多个线程能不能同时共享同一个锁，锁被划分为独享锁和共享锁。**
  - 独享锁：独享锁也叫排他锁，synchronized，ReentrantLock， ReentrantReadWriteLock 的WriteLock写锁
  - 共享锁：ReentrantReadWriteLock的ReadLock读锁

5.2 Java锁分类

03-其他

- **自旋锁：**
  - 实现：CAS、轻量级锁
- **分段锁：**
  - 实现：ConcurrentHashMap
  - ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
- **无锁/偏向锁/轻量级锁/重量级锁**
  - **这四个锁是synchronized独有的四种状态**，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁。
  - 它们是JVM为了提高synchronized锁的获取与释放效率而做的优化
  - 四种状态会随着竞争的情况逐渐升级，而且是不可逆的过程，即不可降级。

用ReentrantLock解决可见性案例中的问题！

5.3 Synchronized和JUC的锁对比

Java已经提供了synchronized，为什么还要使用JUC的锁呢？

Synchronize的缺陷：

- **第一： Synchronized无法控制阻塞时长，阻塞不可中断**
  - 使用Synchronized，假如占有锁的线程被长时间阻塞（IO、sleep、join），由于线程阻塞时没法释放锁，会导致大量线程堆积，轻则影响性能，重则服务雪崩
  - **JUC的锁可以解决这两个缺陷**
- **第二：读多写少的场景中，多个读线程同时操作共享资源时不需要加锁**
  - Synchronized不论是读还是写，均需要同步操作，这种做法并不是最优解
  - **JUC的ReentrantReadWriteLock锁可以解决这个问题**

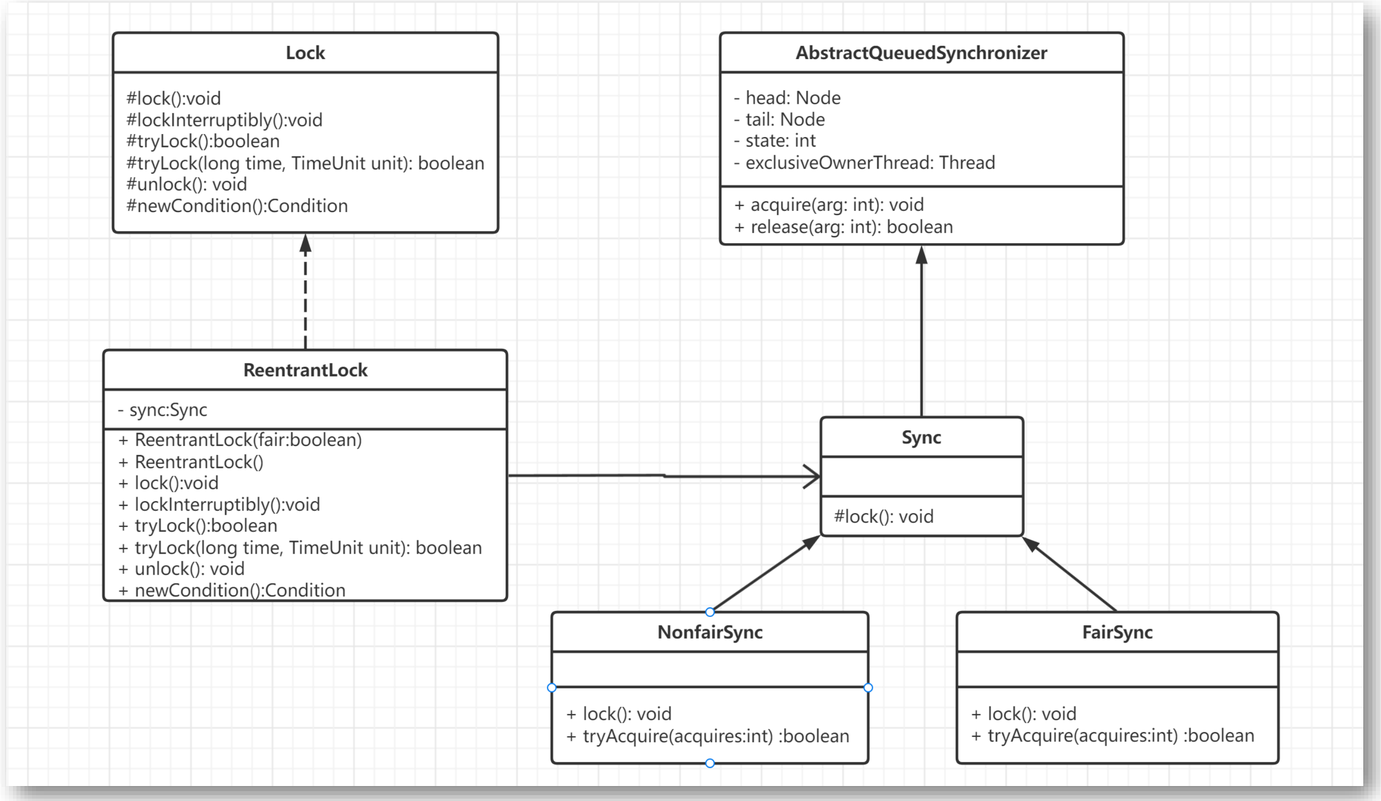
接下来我们看一下ReentrantLock实现原理

5.4 ReentrantLock原理分析之AQS

在重入锁ReentrantLock类关系图中，可以看到NonfairSync和FairSync都继承自抽象类Sync，而**Sync类继承自抽象类AbstractQueuedSynchronizer**（简称AQS）。

如果你看过JUC的源码，发现不仅重入锁用到了AQS， JUC 中绝大部分的同步工具类也都是基于AQS

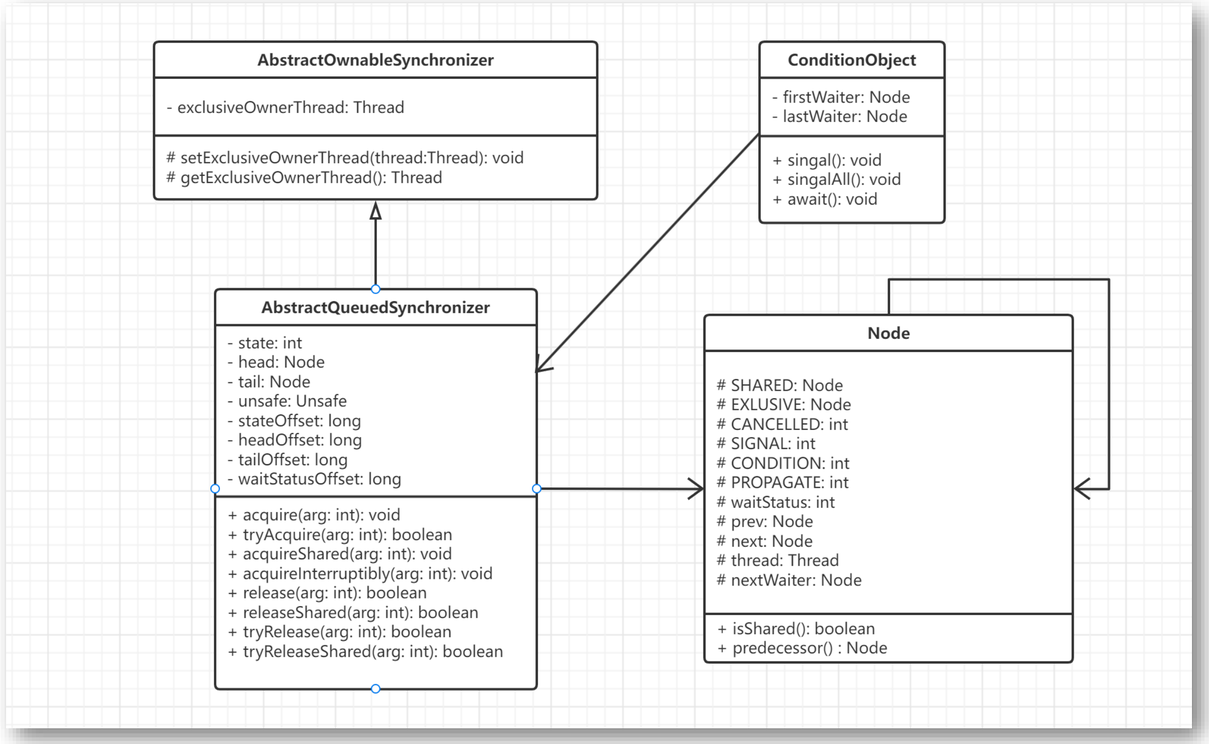
AQS是什么呢？



5.4 ReentrantLock原理分析之AQS

AQS即队列同步器，是JUC并发包中的核心基础组件，其本身只是一个抽象类。其实现原理与前面介绍的**Monitor管程**是一样的，AQS中也用到了CAS和Volatile。

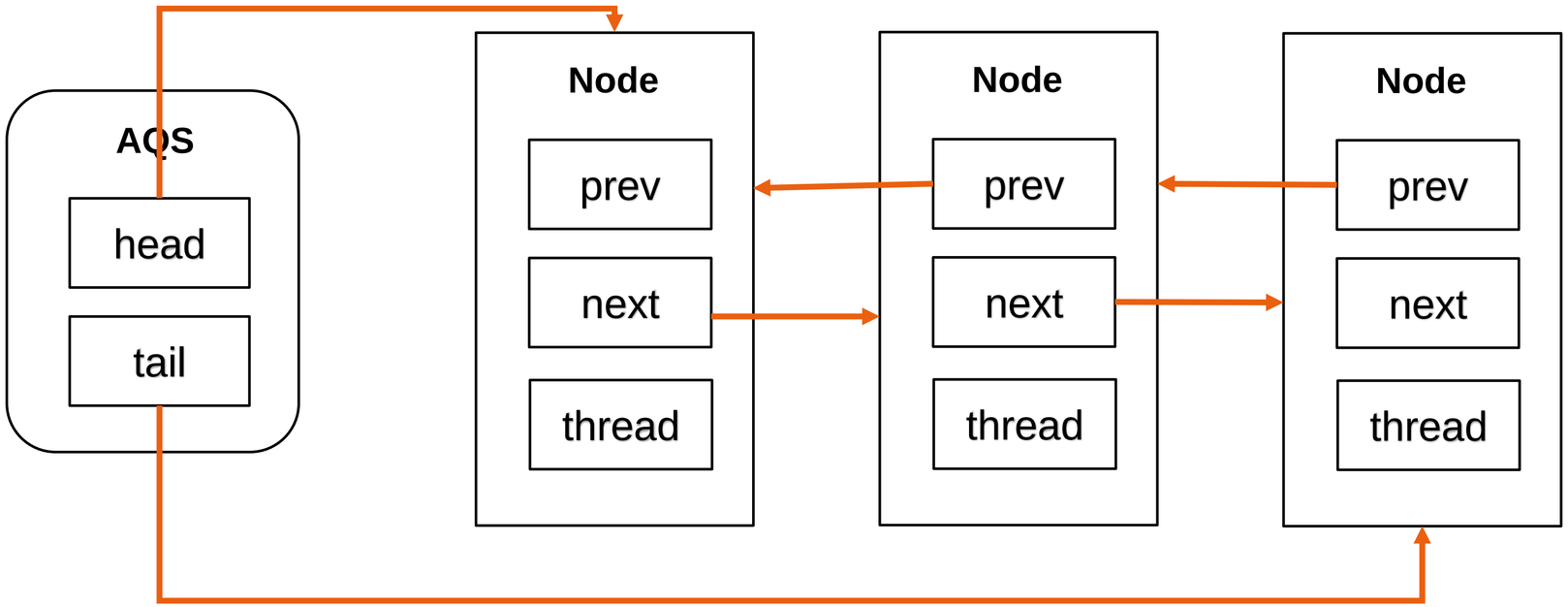
由类图可以看到，AQS是一个**FIFO的双向队列，队列中存储的是thread**，其内部通过节点head和tail记录队首和队尾元素，队列元素的类型为Node。



画个图来看一下

5.5 ReentrantLock原理分析之AQS

AQS中的内部静态类Node为链表节点，**AQS会在线程获取锁失败后，线程会被阻塞并被封装成Node加入到AQS队列中；当获取锁的线程释放锁后，会从AQS队列中的唤醒一个线程（节点）。**



5.5 ReentrantLock原理分析之AQS

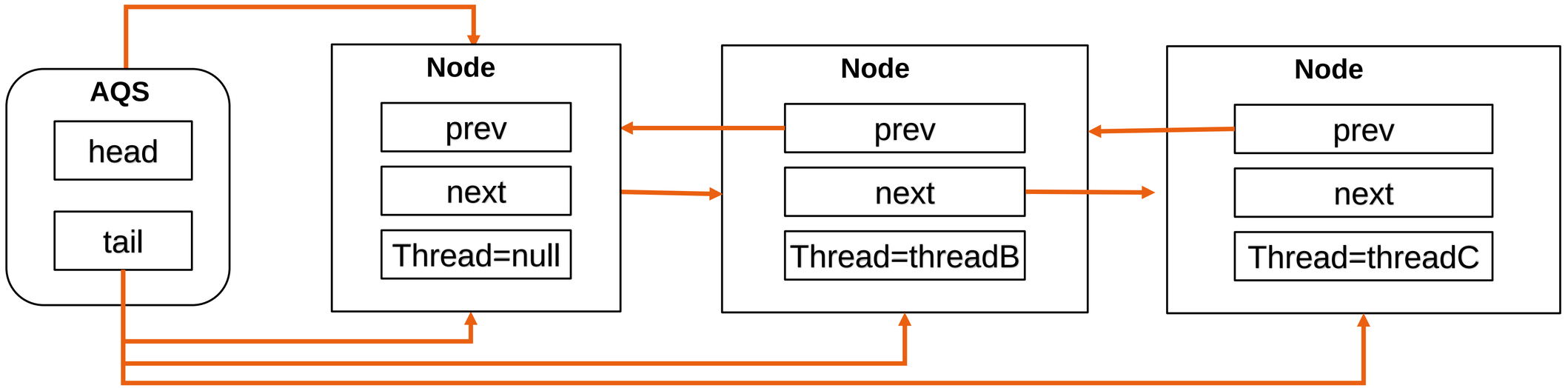
场景01-线程抢夺锁失败时，AQS队列的变化

① AQS的head、tail分别代表同步队列头节点和尾节点指针默认为null

② 当第一个线程抢夺锁失败，同步队列会先初始化，随后线程会被封装成Node节点追加到AQS队列中。

- 假设：当前独占锁的的线程为ThreadA，抢占锁失败的线程为ThreadB。
- 2.1 同步队列初始化，首先在队列中添加Node，thread=null
- 2.2 将ThreadB封装成为Node，追加到AQS队列

③ 当下一个线程抢夺锁失败时，继续重复上面步骤。假设：ThreadC抢占线程失败



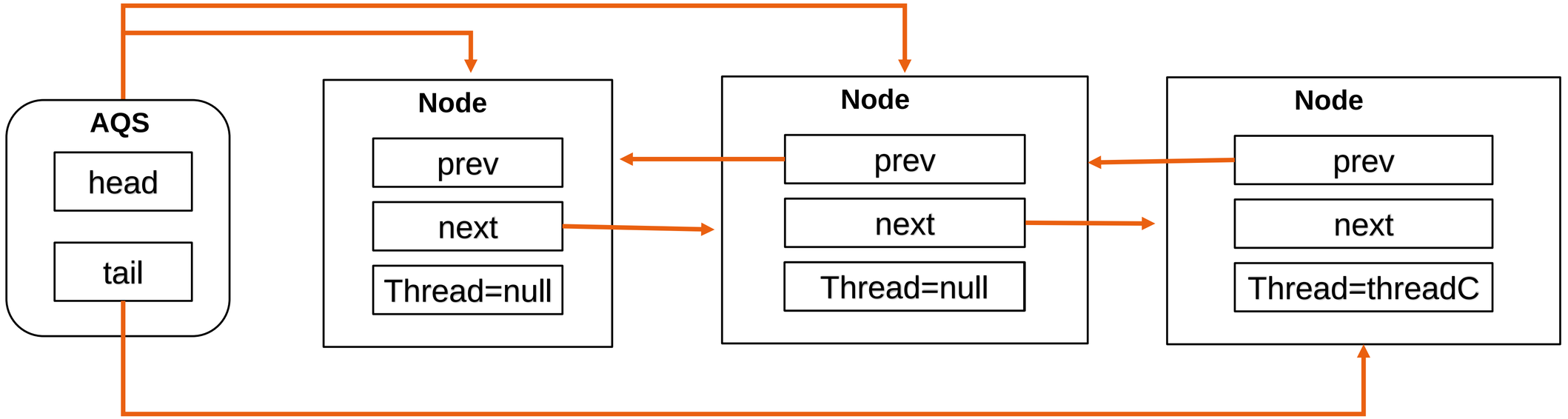
5.5 ReentrantLock原理分析之AQS

场景02-线程被唤醒时，AQS队列的变化

① ReentrantLock唤醒阻塞线程时，会按照**FIFO的原则从AQS中head头部开始唤醒首个节点中线程**。

② head节点表示当前获取锁成功的线程ThreadA节点。

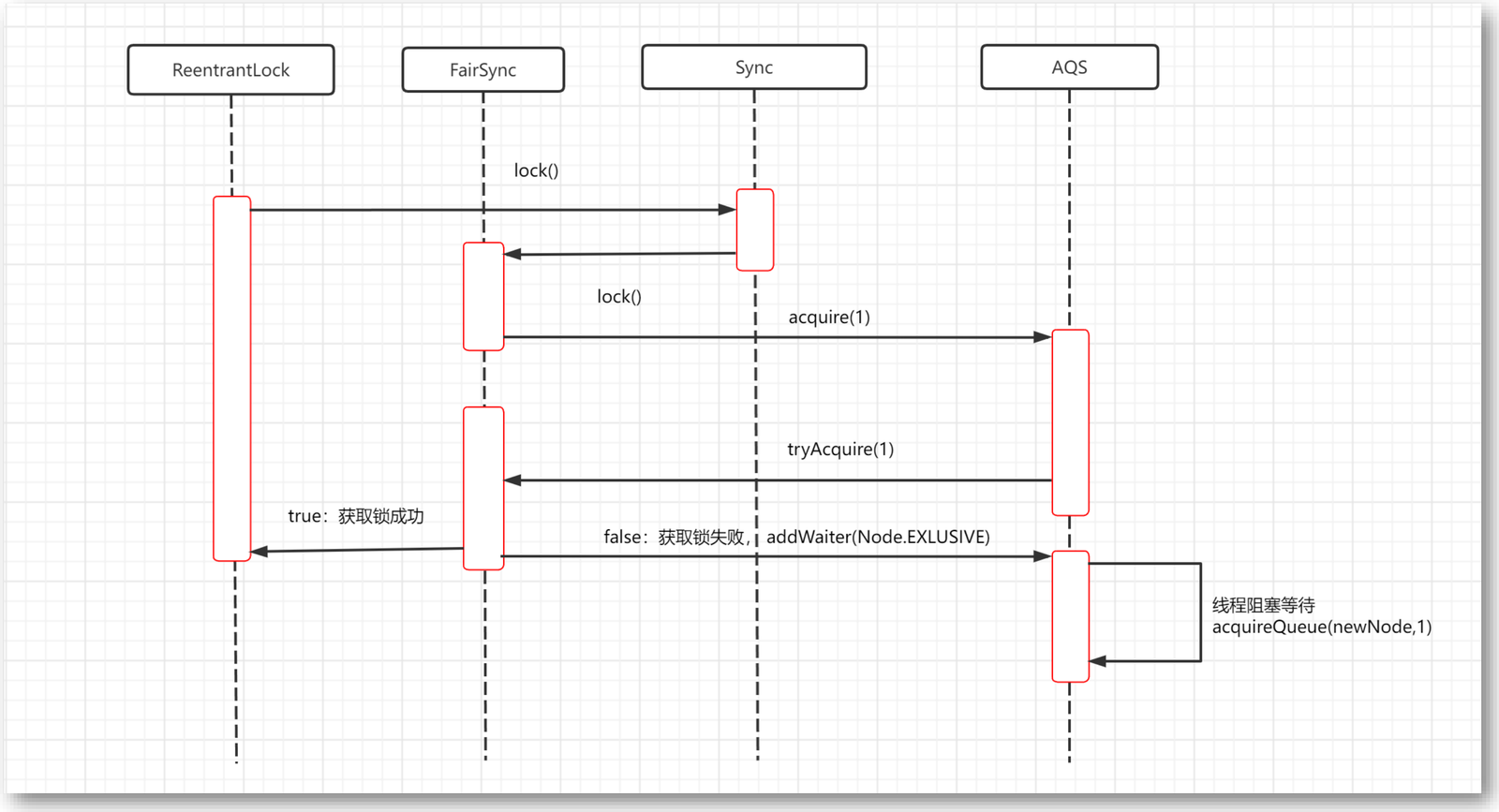
③ 当ThreadA释放锁时，它会唤醒后继节点线程ThreadB，ThreadB开始尝试获得锁，如果ThreadB获得锁成功，会将自己设置为AQS的头节点。ThreadB获取锁成功后，AQS变化如下：



5.6 ReentrantLock源码分析-锁的获取

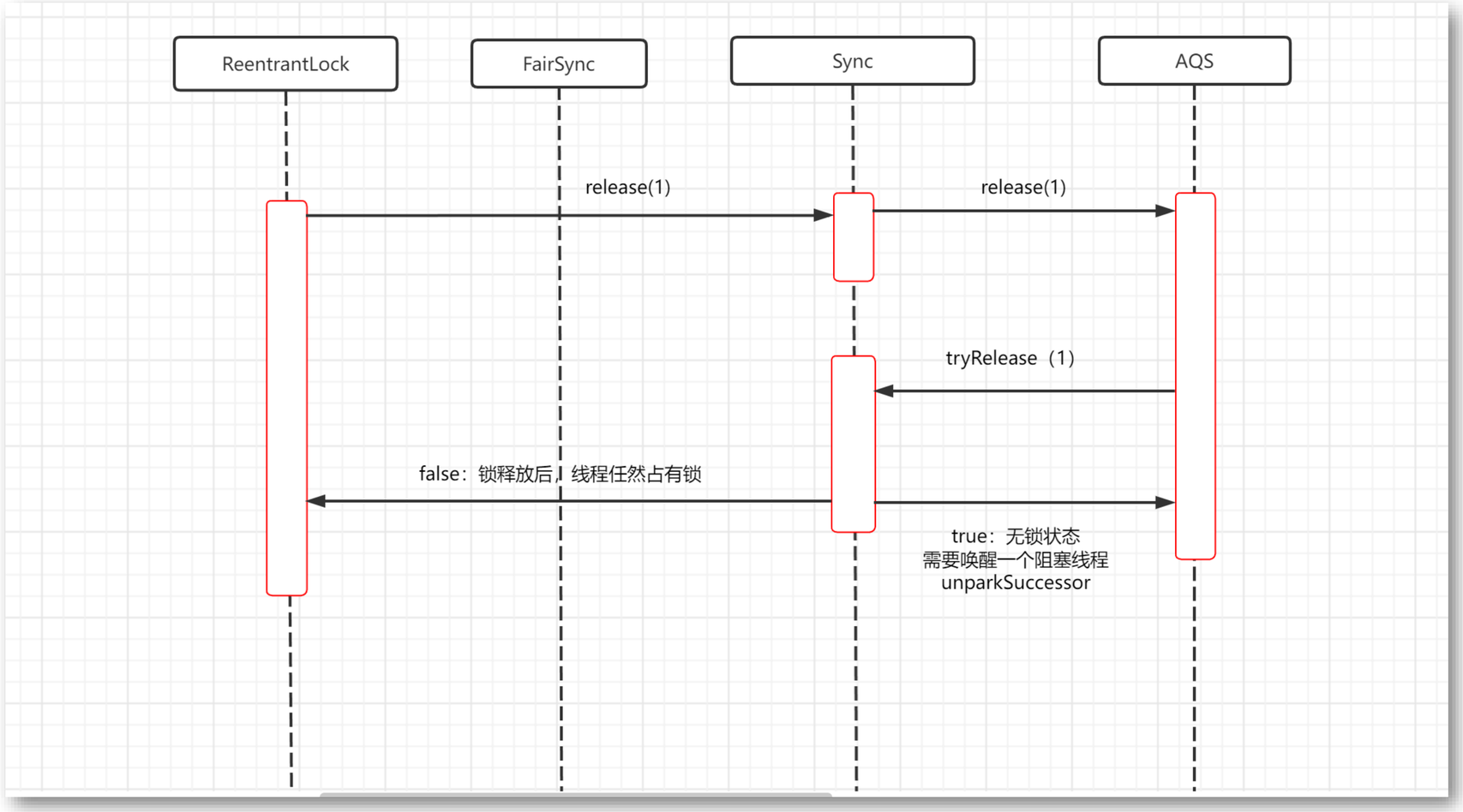
研究任何框架或工具都就要一个入口，我们以重入锁为切入点来理解AQS的作用及实现。下面我们深入ReentrantLock源码来分析AQS是如何实现线程同步的。

ReentrantLock锁获取源码分析：



5.7 ReentrantLock源码分析-锁的释放

ReentrantLock锁释放源码分析：



5.8 公平锁和非公平锁源码实现区别

公平锁/非公平锁：按照多个线程竞争同一锁时需不需要排队，能不能插队

获取锁的两处差异：

① lock方法差异：详情看课堂笔记

② tryAcquire差异：详情看课堂笔记

FairSync.lock：公平锁获取锁

```
final void lock() {
    acquire(1);
}

```

NoFairSync.lock：非公平锁获取锁，lock方法中新线程会先通过CAS操作compareAndSetState(0, 1)，尝试获得锁。

```
final void lock() {
    if (compareAndSetState(0, 1))//新线程，第一次插队
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

NoFairSync.tryAcquire和NoFairSync.nonfairTryAcquire：

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {//非公平锁，入队前，二次插队
            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;
}
```

5.9 读写锁ReentrantReadWriteLock

读写锁：维护着一对锁(读锁和写锁)，通过分离读锁和写锁，使得并发能力比一般的互斥锁有较大提升。**同一时间，可以允许多个读线程同时访问，但在写线程访问时，所有读写线程都会阻塞。**

所以说，读锁是共享的，写锁是排他的。

主要特性：

- 支持公平和非公平锁
- 支持重入
- 锁降级：写锁可以降级为读锁，但是读锁不能升级为写锁

```
1  /** 内部类  读锁 */
2  private final ReentrantReadWriteLock.ReadLock readerLock;
3  /** 内部类  写锁 */
4  private final ReentrantReadWriteLock.WriteLock writerLock;
5
6  final Sync sync;
```

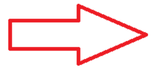
做个案例来看一下

5.10 锁优化

如何优化锁？

- 减少锁的持有时间
- 减少锁粒度
  - 将大对象拆分为小对象，增加并行度，降低锁的竞争
  - 例如：早期ConcurrentHashMap的分段锁
- 锁分离
  - 根据功能场景进行锁分离
  - 例如：读多写少的场景，使用读写锁可以提高性能
- 锁消除：锁消除是编译器自动的一种优化方式
- 锁粗化
  - 增加锁的范围，降低加解锁的频次

```
public synchronized void syncMethodBefore(){
    otherCode1();
    mutextMethod();
    otherCode2();
}
```



```
public void syncMethodAfter(){
    otherCode1();
    synchronized (this){
        mutextMethod();
    }
    otherCode2();
}
```

```
public synchronized void demoMethodBefore(){
    synchronized (lockA){
        mutextMethodA();
    }
    //做其他不需要的同步工作，但能很快执行完毕
  synchronized (lockA){
        mutextMethodB();
    }
}
```

```
public void syncMethodAfter(){
    synchronized (lockA){
        mutextMethodA();
        //做其他不需要的同步工作，但能很快执行完毕
        mutextMethodB();
    }
}
```

```
public synchronized void demoMethodBefore(){
    for (int i = 0; i < circle; i++) {
        synchronized (lock){
            //do sth
        }
    }
}
```

```
public void syncMethodAfter(){
    synchronized (lock){
        for (int i = 0; i < circle; i++) {
            //do sth
        }
    }
}
```

THANKS

教育不是注满一桶水，而是点燃一把火