# 线程安全II

2024年4月3日 17:48

## 什么是AQS?

全称是 AbstractQueuedSynchronizer，即抽象队列同步器。它是构建锁或者其他同步组件的基础框架

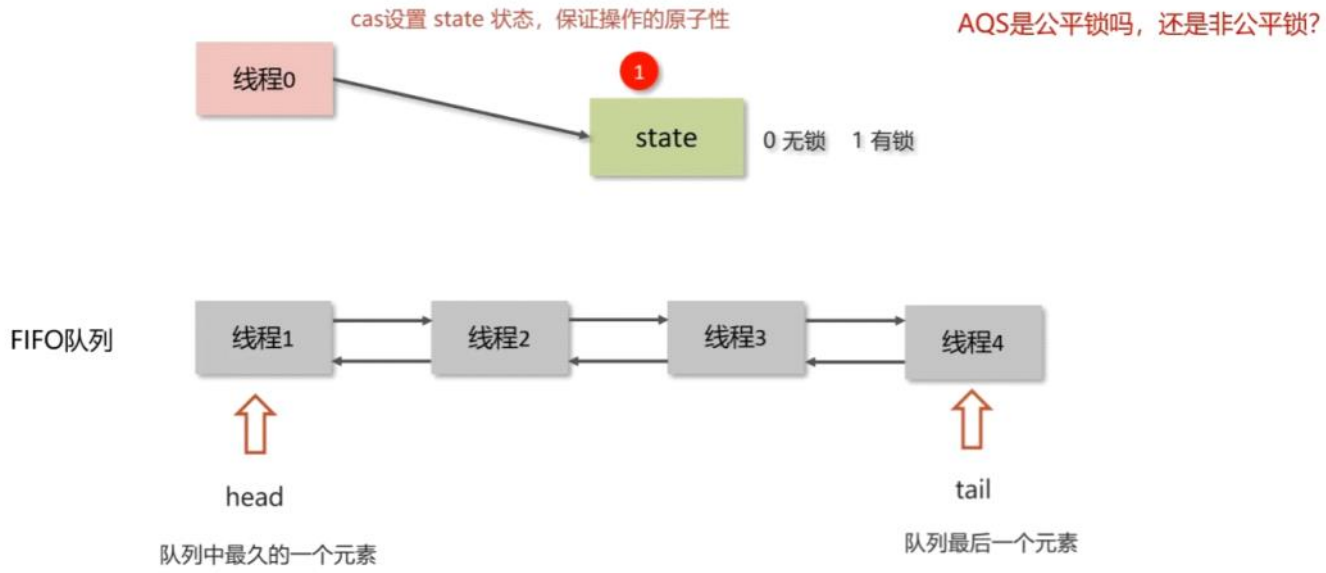
AQS与Synchronized的区别

| synchronized | AQS |
| --- | --- |
| 关键字，c++ 语言实现 | java 语言实现 |
| 悲观锁，自动释放锁 | 悲观锁，手动开启和关闭 |
| 锁竞争激烈都是重量级锁，性能差 | 锁竞争激烈的情况下，提供了多种解决方案 |

AQS常见的实现类

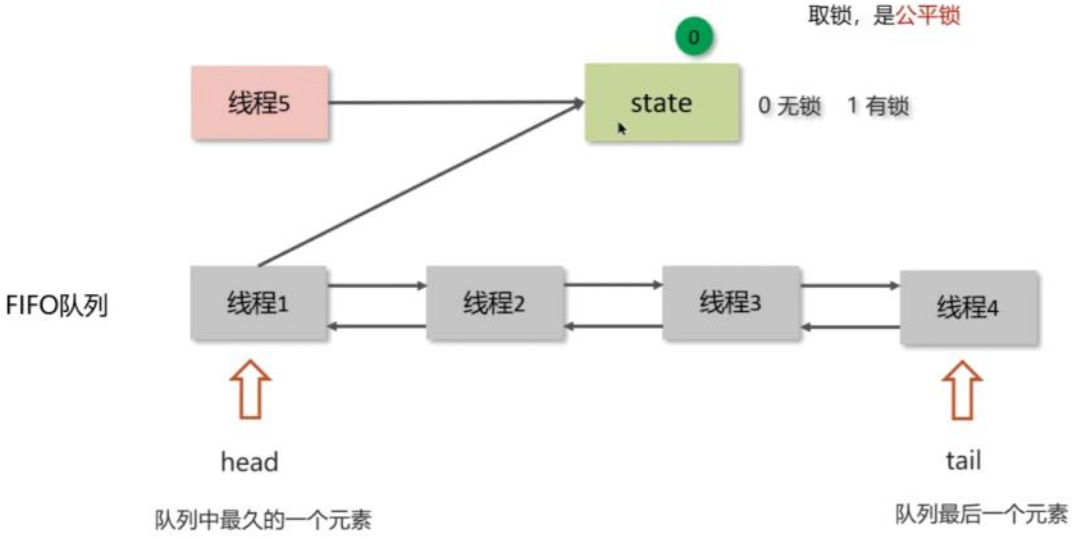
- ReentrantLock 阻塞式锁
- Semaphore 信号量
- CountDownLatch 倒计时锁

## AQS-多个线程共同去抢这个资源是如何保证原子性的呢



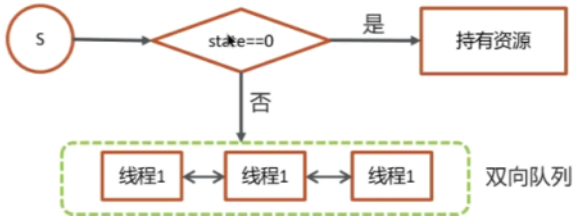
## AQS是公平锁吗，还是非公平锁?

- 新的线程与队列中的线程共同来抢资源，是非公平锁
- 新的线程到队列中等待，只让队列中的head线程获取锁，是公平锁




## 什么是AQS?

- 是多线程中的队列同步器。是一种锁机制，它是做为一个基础框架使用的，像ReentrantLock、Semaphore都是基于AQS实现的
- AQS内部维护了一个先进先出的双向队列，队列中存储的排队的线程
- 在AQS内部还有一个属性state，这个state就相当于是一个资源，默认是0（无锁状态），如果队列中的有一个线程修改成功了state为1，则当前线程就相等于获取了资源
- 在对state修改的时候使用的cas操作，保证多个线程修改的情况下原子性




## ReentrantLock的实现原理

ReentrantLock翻译过来是可重入锁，相对于synchronized它具备以下特点：

- 可中断
- 可以设置超时时间
- 可以设置公平锁
- 支持多个条件变量
- 与synchronized一样，都支持重入

```
//创建锁对象
ReentrantLock lock = new ReentrantLock();
try {
    // 获取锁
    lock.lock();
} finally {
    // 释放锁
    lock.unlock();
}
```

## ReentrantLock的实现原理

ReentrantLock主要利用CAS+AQS队列来实现。它支持公平锁和非公平锁，两者的实现类似

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：

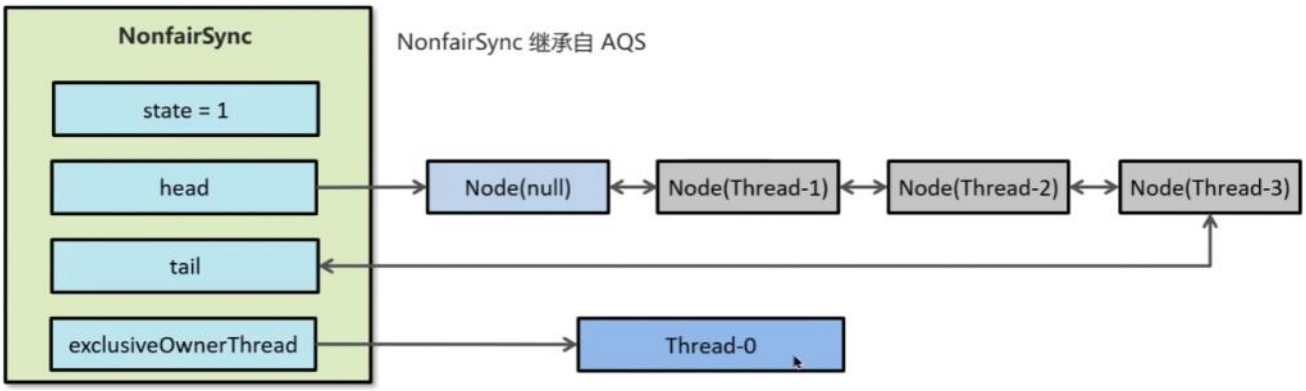
```
public ReentrantLock() {
    sync = new NonfairSync();
}

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

```
abstract static class Sync extends AbstractQueuedSynchronizer {

}
```

## ReentrantLock的实现原理




- 线程来抢锁后使用cas的方式修改state状态，修改状态成功为1，则让exclusiveOwnerThread属性指向当前线程，获取锁成功
- 假如修改状态失败，则会进入双向队列中等待，head指向双向队列头部，tail指向双向队列尾部
- 当exclusiveOwnerThread为null的时候，则会唤醒在双向队列中等待的线程
- 公平锁则体现在按照先后顺序获取锁，非公平体现在不在排队的线程也可以抢锁

## ReentrantLock的实现原理

- ReentrantLock表示支持重新进入的锁，调用 lock 方 法获取了锁之后，再次调用 lock，是不会再阻塞
- ReentrantLock主要利用CAS+AQS队列来实现
- 支持公平锁和非公平锁，在提供的构造器的中无参默认是非公平锁，也可以传参设置为公平锁

## synchronized和Lock有什么区别？

- 语法层面

synchronized 是关键字，源码在 jvm 中，用 c++ 语言实现

Lock 是接口，源码由 jdk 提供，用 java 语言实现

使用 synchronized 时，退出同步代码块锁会自动释放，而使用 Lock 时，需要手动调用 unlock 方法释放锁

- 功能层面

二者均属于悲观锁、都具备基本的互斥、同步、锁重入功能

Lock 提供了许多 synchronized 不具备的功能，例如公平锁、可打断、可超时、多条件变量

Lock 有适合不同场景的实现，如 ReentrantLock， ReentrantReadWriteLock(读写锁)

- 性能层面

在没有竞争时，synchronized 做了很多优化，如偏向锁、轻量级锁，性能不赖

在竞争激烈时，Lock 的实现通常会提供更好的性能

## 死锁产生的条件是什么?

死锁：一个线程需要同时获取多把锁，这时就容易发生死锁

```
Object A = new Object();
Object B = new Object();
Thread t1 = new Thread(() -> {
    synchronized (A) {
        System.out.println("lock A");
        try {
            sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        synchronized (B) {
            System.out.println("lock B");
            System.out.println("操作...");
        }
    }
}, "t1");

Thread t2 = new Thread(() -> {
    synchronized (B) {
        System.out.println("lock B");
        try {
            sleep(500);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        synchronized (A) {
            System.out.println("lock A");
            System.out.println("操作...");
        }
    }
}, "t2");
t1.start();
t2.start();
```


此时程序并没有结束，这种现象就是死锁现象...线程t1持有A的锁等待获取B锁，线程t2持有B的锁等待获取A的锁。

## 如何进行死锁诊断?

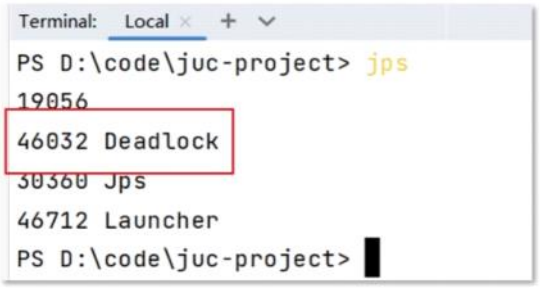
当程序出现了死锁现象，我们可以使用jdk自带的工具：jps和 jstack

- jps：输出JVM中运行的进程状态信息
- jstack：查看java进程内线程的堆栈信息

## 如何进行死锁诊断?

解决步骤如下

第一：查看运行的线程




第二，使用jstack查看线程运行的情况，下图是截图的关键信息

运行命令：jstack -l 46032


## 如何进行死锁诊断?

### 其他解决工具，可视化工具

- jconsole

用于对jvm的内存，线程，类 的监控，是一个基于 jmx 的 GUI 性能监控工具

打开方式：java 安装目录 bin目录下 直接启动 jconsole.exe 就行

- VisualVM：故障处理工具

能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈

打开方式：java 安装目录 bin目录下 直接启动 jvisualvm.exe就行

### 1. 死锁产生的条件是什么?

一个线程需要同时获取多把锁，这时就容易发生死锁

### 2. 如何进行死锁诊断?

- 当程序出现了死锁现象，我们可以使用jdk自带的工具：jps和 jstack
- jps：输出JVM中运行的进程状态信息
- jstack：查看java进程内线程的堆栈信息，查看日志，检查是否有死锁

  如果有死锁现象，需要查看具体代码分析后，可修复
- 可视化工具jconsole、VisualVM也可以检查死锁问题

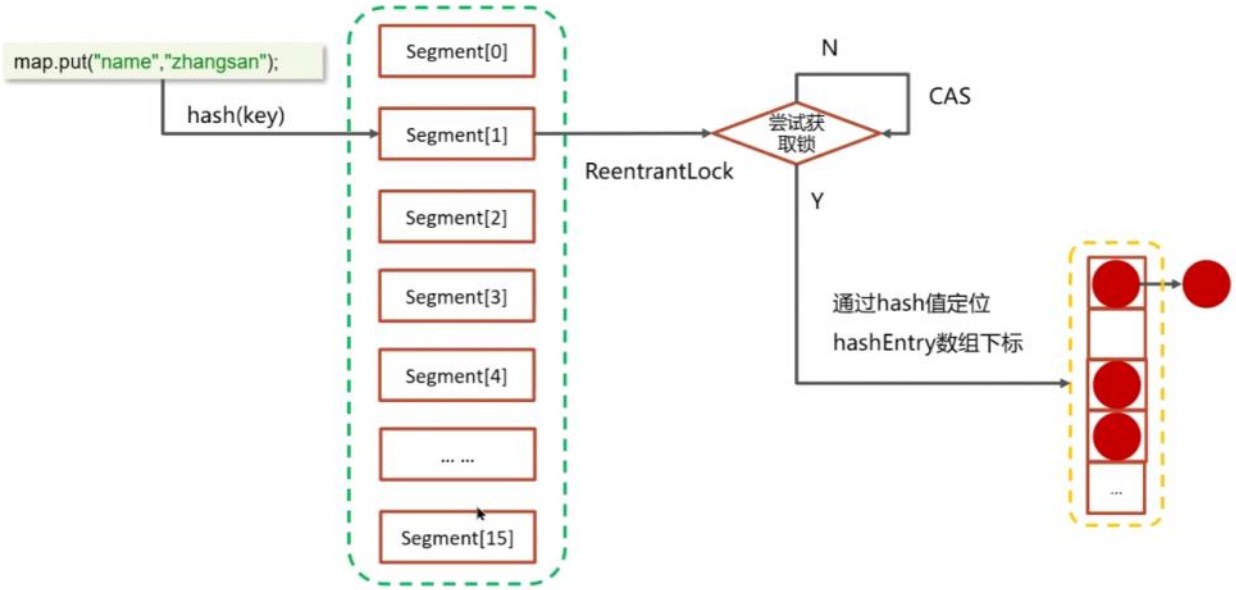
## 聊一下ConcurrentHashMap

ConcurrentHashMap 是一种线程安全的高效Map集合

底层数据结构：

- JDK1.7底层采用分段的数组+链表实现
- JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。

## JDK1.7中ConcurrentHashMap

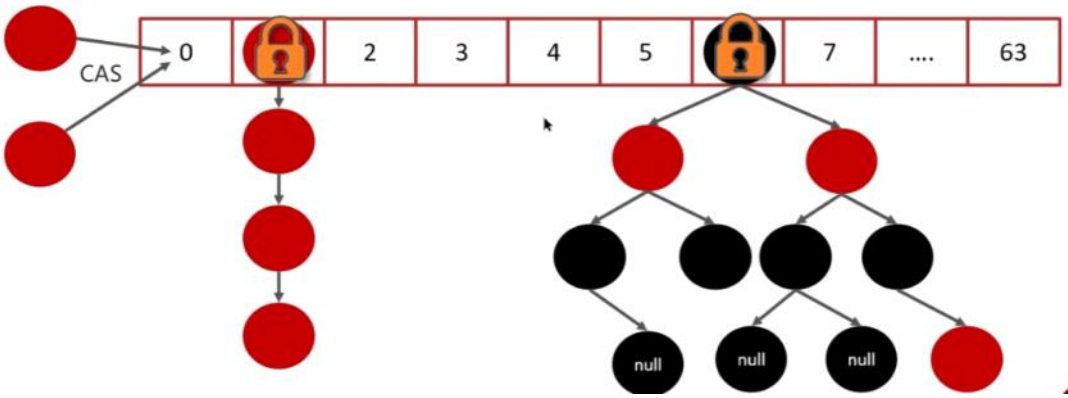



## JDK1.8中ConcurrentHashMap

在JDK1.8中，放弃了Segment臃肿的设计，数据结构跟HashMap的数据结构是一样的：数组+红黑树+链表

采用 CAS + Synchronized来保证并发安全进行实现

- CAS控制数组节点的添加
- synchronized只锁定当前链表或红黑二叉树的首节点，只要hash不冲突，就不会产生并发的问题，效率得到提升



聊一下ConcurrentHashMap

1. 底层数据结构：

- JDK1.7底层采用分段的数组+链表实现
- JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树

2. 加锁的方式

- JDK1.7采用Segment分段锁，底层使用的是ReentrantLock
- JDK1.8采用CAS添加新节点，采用synchronized锁定链表或红黑二叉树的首节点，相对Segment分段锁粒度更细，性能更好

## 导致并发程序出现问题的根本原因是什么

Java并发编程三大特性

- 原子性
- 可见性
- 有序性

## 导致并发程序出现问题的根本原因是什么

原子性：一个线程在CPU中操作不可暂停，也不可中断，要不执行完成，要不不执行

```
int ticketNum = 10;
public void getTicket(){
    if(ticketNum <= 0){
        return ;
    }
    System.out.println(Thread.currentThread().getName()+"抢到一张票,剩余:"+ticketNum);
     // 非原子性操作
       ticketNum--;
}

public static void main(String[] args) {
    TicketDemo demo = new TicketDemo();
    for(int i=0;i<20;i++){
        new Thread(demo::getTicket).start();
    }
}
```

不是原子操作，怎么保证原子操作呢?

## 导致并发程序出现问题的根本原因是什么

1.synchronized：同步加锁

2.JUC里面的lock：加锁

```
int ticketNum = 10;
public synchronized void getTicket(){
    if(ticketNum <= 0){
        return ;
    }
    System.out.println(Thread.currentThread().getName()+"抢到一张票,剩余:"+ticketNum);
    // 非原子性操作
        ticketNum--;
}

public static void main(String[] args) {
    TicketDemo demo = new TicketDemo();
    for(int i=0;i<20;i++){
        new Thread(demo::getTicket).start();
    }
}
```

## 导致并发程序出现问题的根本原因是什么

内存可见性：让一个线程对共享变量的修改对另一个线程可见

```
public class VolatileDemo {

    private static boolean flag = false;
    public static void main(String[] args) throws InterruptedException {
        new Thread(()->{
            while(!flag){
            }
            System.out.println("第一个线程执行完毕...");
        }).start();
        Thread.sleep(100);
        new Thread(()->{
            flag = true;
            System.out.println("第二线程执行完毕...");
        }).start();
    }
}
```


## 导致并发程序出现问题的根本原因是什么

有序性

指令重排：处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的

```
int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

## 导致并发程序出现问题的根本原因是什么

1.原子性　synchronized、lock

2.内存可见性　volatile、synchronized、lock

3.有序性　volatile