猫眼电影

Java锁机制

By 徐荣阳 2017.02.21

目录

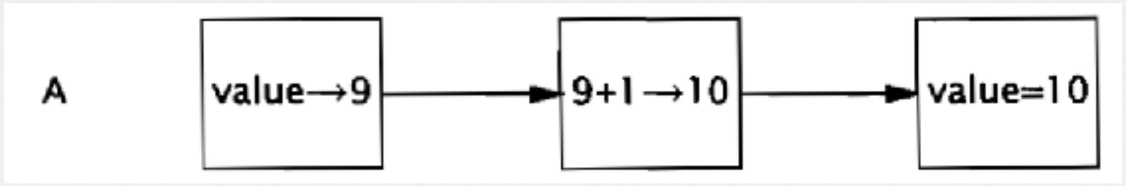
- 常见的并发模型
- 实现Java锁的基础
- Java中锁的实现机制
- QA

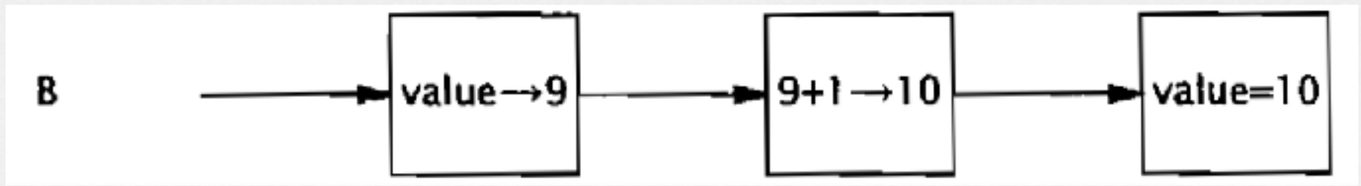
常见的并发模型

- 线程与锁（Java，C，C++）
- 消息传递（Akka，Erlang）
- CSP模型（Go）
- 函数式并发（Clojure）

一个典型的并发问题

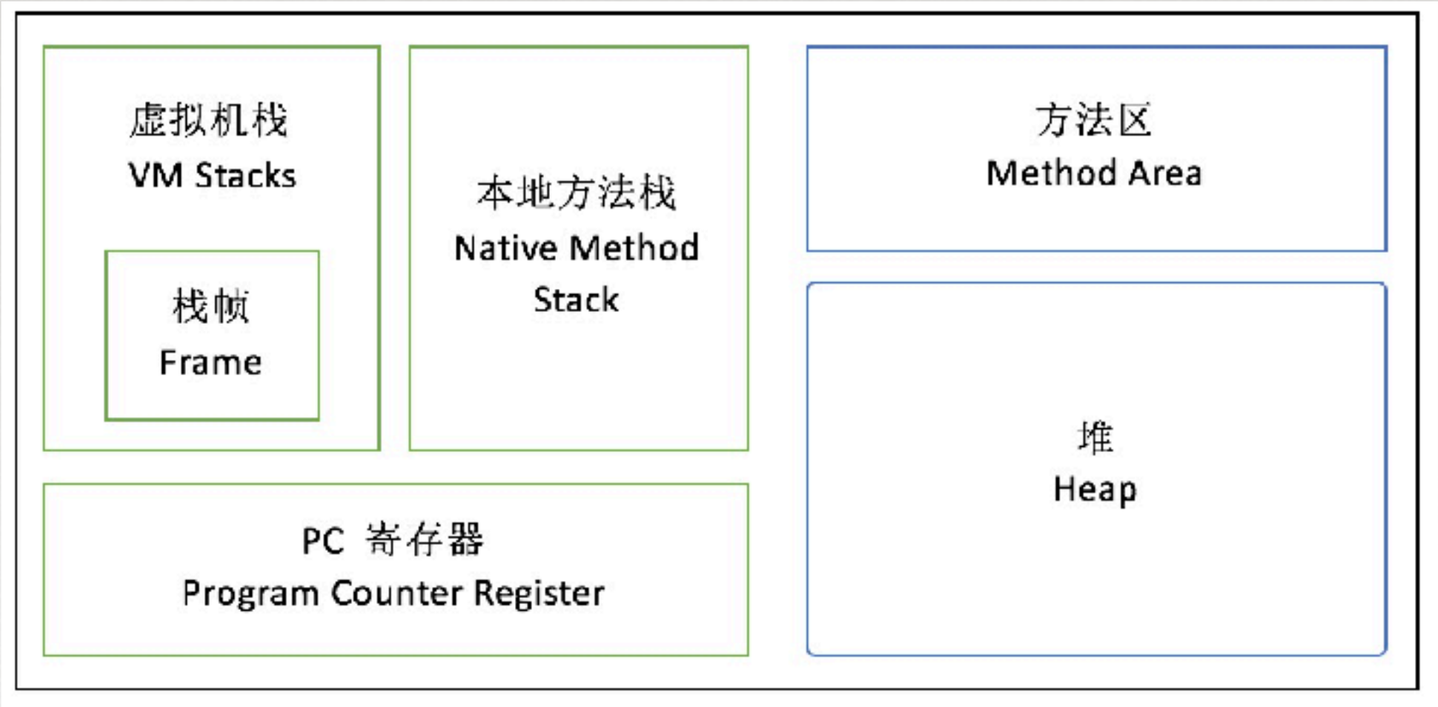
```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** 返回一个唯一的数值。*/
    public int getNext() {
        return value++;
    }
}
```





Java中哪些数据是共享的

- 线程间共享变量
  - 实例域
  - 静态域
  - 数组元素
- 非共享变量
  - 局部变量
  - 方法参数
  - 异常处理参数



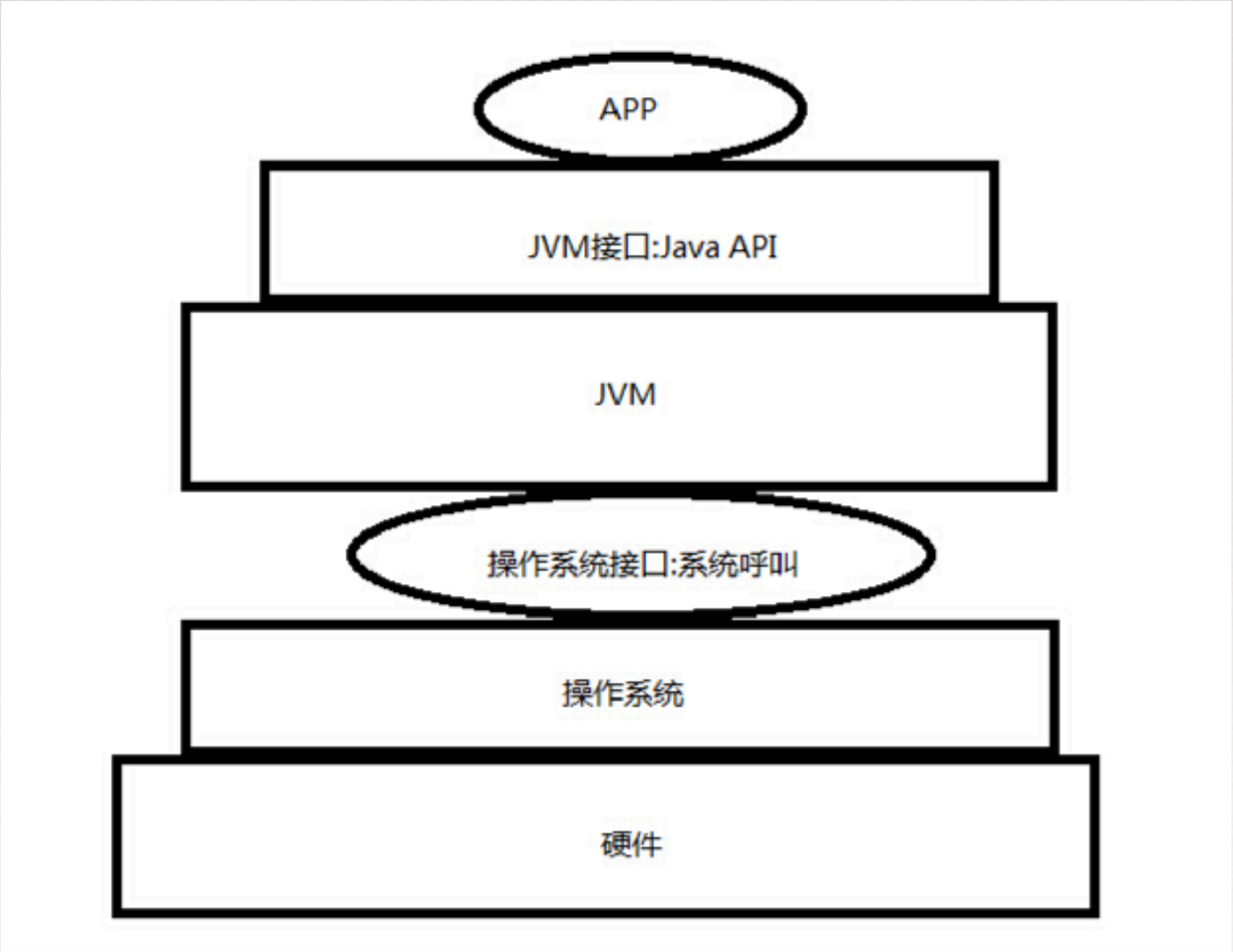
Java虚拟机运行时数据区

Java中有哪些锁

- JDK 1.5之前
  - ✓synchronized
- JDK 1.5 ~ 至今
  - ✓ReentrantLock
  - ✓ReentrantReadWriteLock
- 其它一些并发工具
  - ✓CountDownLatch
  - ✓CyclicBarrier
  - ✓Semaphore
  - ✓AtomicX (Integer, Long, Reference)
  - ✓AtomicXFieldUpdater(Integer, Long, Reference)

实现Java锁的基础

- 特殊指令
  - CPU级别CAS指令
- 中断机制
- 内存模型
  - CPU
  - Java



实现Java锁的基础

- CAS(Compare and swap)
  - 基于CMPXCHG指令（Intel）
  - sun.misc.Unsafe.compareAndSwapX
- LockSupport(park, unpark)
- volatile
  - 共享变量的“可见性”
  - 当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int actual = *ptr;
3      if (actual == expected)
4          *ptr = new;
5      return actual;
6  }
```

评估锁的实现

- 互斥
- 公平性
- 性能

SpinLock实现

- 互斥
- 非公平
- 性能
  - 消耗CPU时间
  - 适用于保持锁时间比较短的情况

```java
public class SpinLock {
    private AtomicReference<Thread> owner = new AtomicReference<~>();

    public void lock() {
        Thread currentThread = Thread.currentThread();

        // 如果锁未被占用，则设置当前线程为锁的拥有者
        while (!owner.compareAndSet(null, currentThread)) {}
    }

    public void unlock() {
        Thread currentThread = Thread.currentThread();

        // 只有锁的拥有者才能释放锁
        owner.compareAndSet(currentThread, null);
    }
}
```

SpinLock实现

- AtomicInteger
- AtomicX

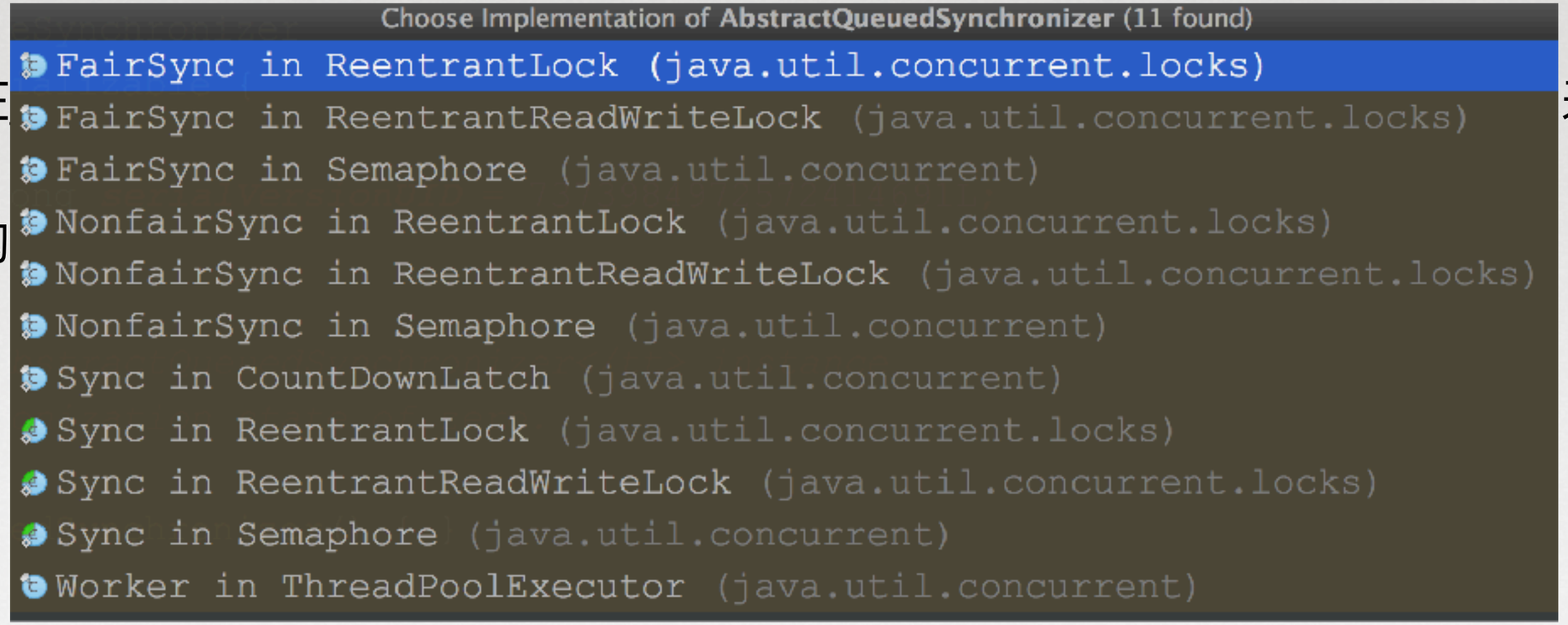
```
/**
 * Atomically increments by one the current value.
 *
 * @return the updated value
 */
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Q: Too much spinning?

A: Using Queues: Sleeping Instead Of Spinning

AQS

- AbstractQueuedSynchronizer
- 在 来
  构



AQS

- 同步状态的原子性管理
- 线程的阻塞与解除阻塞
- 队列的管理

同步状态的原子管理

- AQS类使用state变量（int）来保存同步状态（AQLS）
- 同步状态的含义是在子类中赋予的
- 提供getState、setState以及compareAndSet操作

同步状态在子类中的作用

- ReentrantLock中表示获得锁的线程对锁的重入次数
- CountDownLatch中表示计数器的初始大小
- Semaphore表示初始化的许可数

线程的阻塞与解除阻塞

- Java中的线程对应操作系统中的线程
- j.u.c.locks.LockSupport
- sun.misc.Unsafe

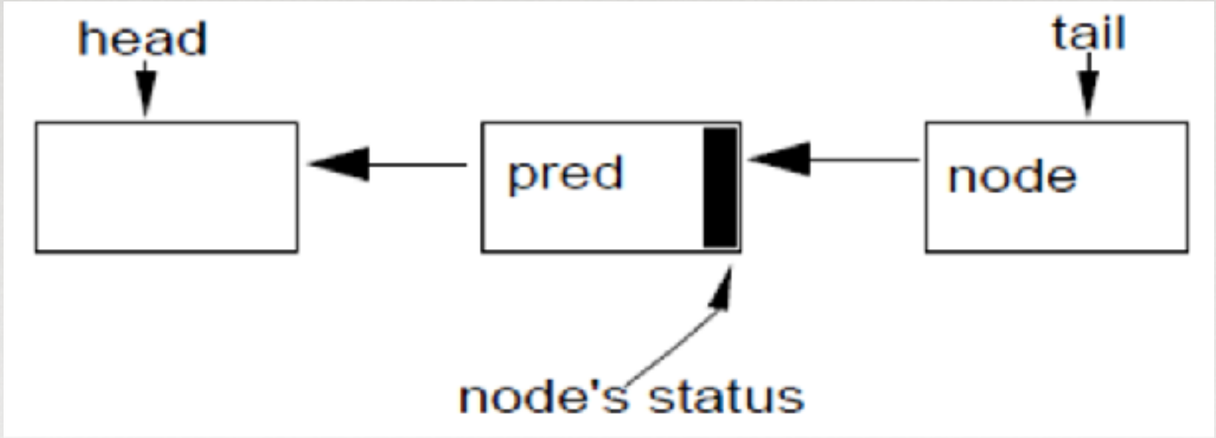
```
public static void park() {
    unsafe.park(false, 0L);
}
```

```
public static void unpark(Thread thread) {
    if (thread != null)
        unsafe.unpark(thread);
}
```

队列管理(CLH队列)

- 基于链表的FIFO队列
- CLH的头节点的线程是锁的持有者
- 没有获得锁的线程会被包装成Node

  放入CLH的尾部进行排队并休眠

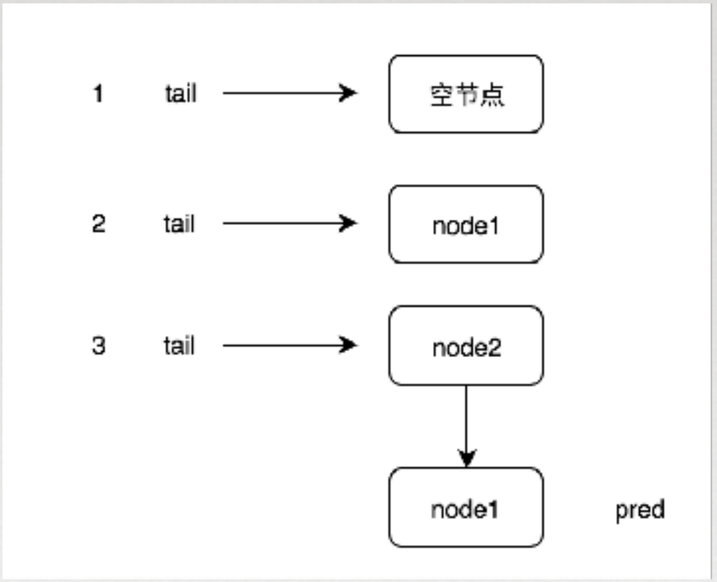


队列管理(CLH队列)

```
public class CLHLock {

    private static class QNode {
        // 是否拿到锁
        volatile boolean locked;
    }

    private final AtomicReference<QNode> tail; // 队列尾
    private final ThreadLocal<QNode> myNode;   // 当前节点
    private final ThreadLocal<QNode> myPred;   // 前置节点

    public CLHLock() {
        tail = new AtomicReference<QNode>(new QNode());
        myNode = new ThreadLocal<QNode>() {
            protected QNode initialValue() {
                return new QNode();
            }
        };

        myPred = new ThreadLocal<QNode>();
    }
}
```

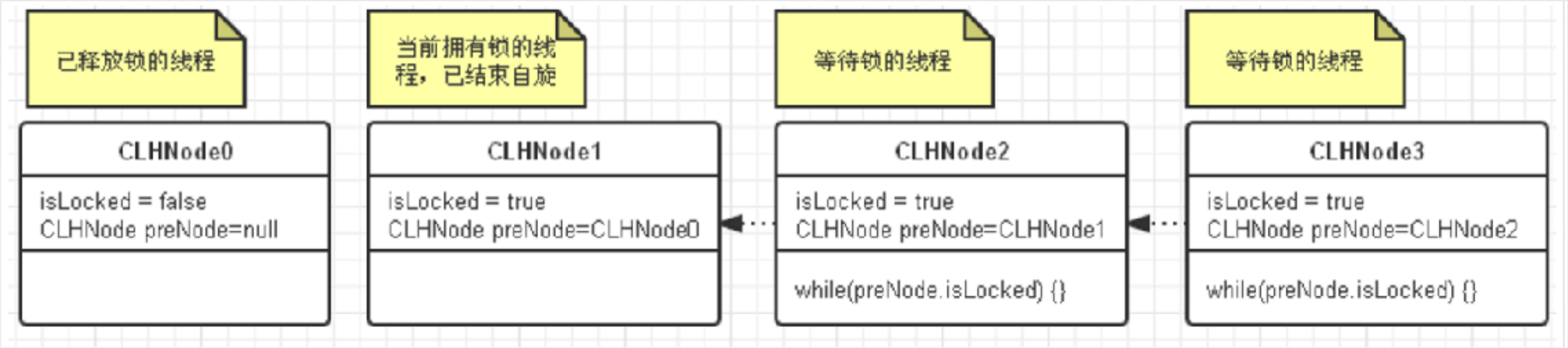
队列管理(CLH队列)

```
public void lock() {
    QNode node = myNode.get();
    node.locked = true; // 设置为已拿到锁
    // CAS获取到前驱节点，并设置tail为当前节点
    QNode pred = tail.getAndSet(node);
    myPred.set(pred); // 设置前驱节点
    while (pred.locked) {} // 没有拿到锁，自旋
}
public final V getAndSet(V newValue) {
    while (true) {
        V x = get();
        if (compareAndSet(x, newValue))
            return x;
    }
}
```



队列管理(CLH队列)

```
public void unlock() {
    QNode node = myNode.get(); // 获取当前线程对应的node
    node.locked = false; // 释放锁
    myNode.set(myPred.get()); // 当前节点设置为空
}
```



队列管理(CLH队列)

- 能够快速入队和出队
- 在前驱节点的属性上自旋
- CLH的队列是隐式的，并不实际持有下个节点
- CLH锁释放时只需要改变自己的属性

AQS要点

- acquire操作会阻塞调用的线程，直到同步状态允许其继续执行
- release操作是通过某种方式改变同步状态，使得一或多个被acquire阻塞的线程继续执行

acquire伪代码

```
while（当前线程没有获得执行许可）
        将当前线程入队，如果当前线程不在等待队列
        可能会阻塞当前线程
}
可能更新同步状态
如果当前线程在等待队列中，将其移除队列
```

release伪代码

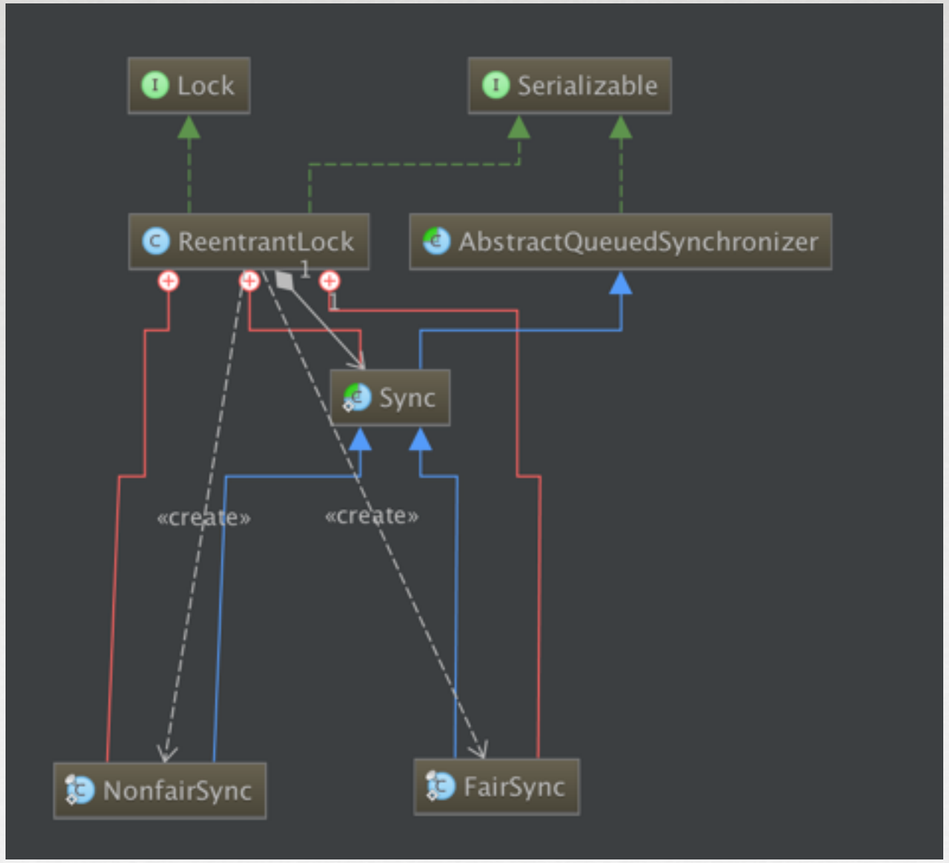
```
更新同步器状态
if（某一被阻塞线程被允许执行acquire操作）
        激活一个或多个等待队列中的线程
```

ReentrantLock

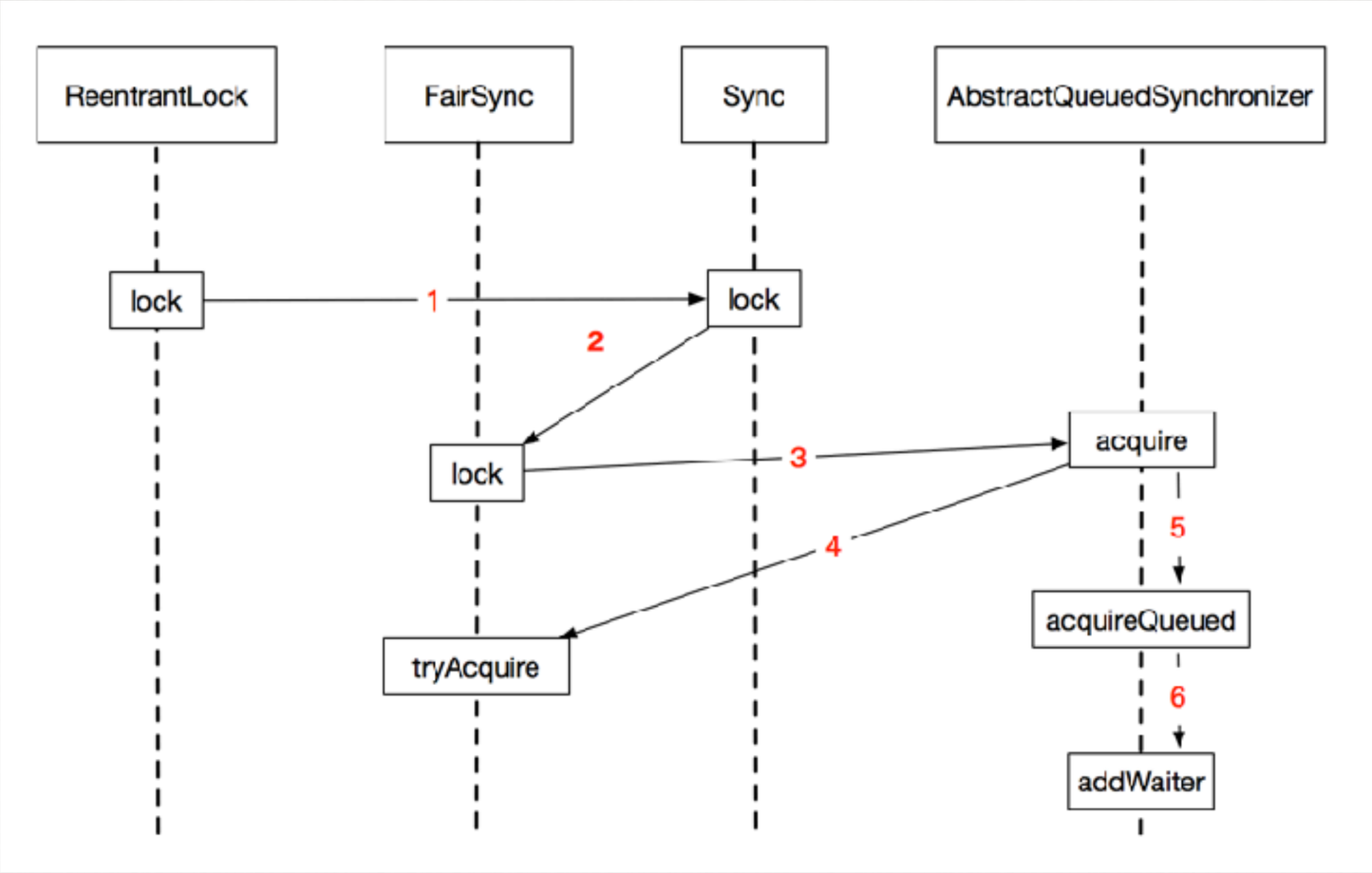
```
public class ReentrantLock implements Lock, java.io.Serializable {

    private final Sync sync;  // API调用都委托给一个内部类 Sync，Sync继承了AQS

    public ReentrantLock(boolean fair) {
        // Sync分为两个子类：公平锁和非公平锁
        sync = fair ? new FairSync() : new NonfairSync();
    }

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

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

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

ReentrantLock

继承关系



调用关系（以公平锁为例）



所以，AQS是基础

ReentrantLock

```
static final class FairSync extends Sync {
    private static final long serialVersionUID = -3000897897090466540L;

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

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

NonFairSync

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

acquire操作尝试获取锁,没有获取成功就加入等待队列

```
if(!tryAcquire(arg)) {
    node = 创建等待队列新节点
    pred = 新节点的前继节点
    while (!(pred是头结点 && tryAcquire(arg))) {
        if (检查前一个节点的状态，判断是否要挂起当前线程) {
            park()
        } else {
            向前遍历，找到合适的前继节点
        }
        设置正确的前继承节点
    }
    head = node
}
```

ReentrantLock

tryAcquire直接尝试获取锁

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&     // 没有等待节点
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current); // 拿到锁
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;  // 可重入，并计数
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

release操作释放锁，并唤醒后继节点

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        //如果waitStatus为0，则表示后面没阻塞线程了，没必要进行唤醒了
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h); // 唤醒线程
        return true;
    }
    return false;
}
```

ReentrantLock

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases; // 更改同步器状态
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

还有哪些

- Shared
- Cancellation
- Timeout
- Condition

Reference

- The java.util.concurrent Synchronizer Framework
- 《Java并发编程实践》
- concurrent包源码

Thanks

By 徐荣阳 2017.02.21