保护性暂停模式

定义（Guarded Suspension Design Pattern）

1. 某个结果需要在多线程之间传递，则可以让这些线程关联到一个对象 GuardedObject
2. 但是如果这个结果需要不断的从一个线程到另一个线程那么可以使用消息队列（见生产者/消费者）
3. 我们前面前面说的join、future采用的就是这个模式

如何实现

最简单的实现

1、首先编写一个简单的GuardedObject

```
package com.shadow.guarded;

import lombok.extern.slf4j.Slf4j;

@Slf4j(topic = "enjoy")
public class GuardedObject {
    private Object response;

    Object lock = new Object();

    /**
     * 加锁获取 response的值 如果response 没有值则等待
     * @return
     */
    public Object getResponse(){
        synchronized (lock) {
            log.debug("主线程 获取 response 如果为null则wait");
            while (response == null) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        return response;
    }


    /**
     * t1 给response设置值
     * @param response
     */
    public void setResponse(Object response) {
        synchronized (lock) {
```

```
            this.response = response;
            //设置完成之后唤醒主线程
            lock.notifyAll();
        }
    }
}
```

2、编写模拟耗时操作

```
package com.shadow.guarded;

import java.util.concurrent.TimeUnit;

public class Operate {

    public static String dbOprate(){
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "result";
    }


}
```

3、编写测试类

```
package com.shadow.guarded;

import lombok.extern.slf4j.Slf4j;

@Slf4j(topic = "enjoy")
public class Test {
    public static void main(String[] args) {
        GuardedObject guardedObject = new GuardedObject();
        new Thread(() -> {
            String result = Operate.dbOprate();
            log.debug("t1 set完毕...");
            guardedObject.setResponse(result);
        },"t1").start();


        log.debug("主线程等待t1 set");

        Object response = guardedObject.getResponse();
        log.debug("response: [{}] lines",response);
    }
```

```
}
```

结果

```
17:36:15.856 [main] DEBUG enjoy - 主线程等待t1 set
17:36:15.859 [main] DEBUG enjoy - 主线程 获取 response 如果为null则wait
17:36:19.856 [t1] DEBUG enjoy - t1 set完毕...
17:36:19.856 [main] DEBUG enjoy - response: [result] lines

Process finished with exit code 0
```

超时实现

如果想要实现超时，那么在get的时候需要定义一个超时时间

```
public Object getResponse(long millis)
```

然后wait的不能无限的等待

```
lock.wait(millis);
```

继而结束while循环

```
public Object getResponse(long millis){
    synchronized (lock) {
        log.debug("主线程 获取 response 如果为null则wait");
        while (response == null) {
            try {
                lock.wait(millis);
                break;
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    return response;
}
```

分析1

这种做法的问题在于如果主线程被别人叫醒了；就会立马返回；比如超时时间是5s；但是在第2s的时候别人把主线程叫醒了，那么主线程会立马返回没有等足5s

所以需要设计一个经历时间；也就是从他wait到被别人叫醒中间一共经历了多少时间；判断这个时间是否符合超时;如果要计算这个经历时间必须知道开始时间和结束时间；

1、首先定一个开始时间等于当前时间 long begin = System.currentTimeMillis();

2、定一个经历时间 默认为0 long timePassed = 0;

3、判断是否满足条件，满足则返回结果不阻塞；不满足则然后进入while循环 首先计算等待时间（也就是wait的时间） millis-timePassed

4、判断等待时间是否小于0；小于0标识超时了直接结束while循环 返回不等待了

5、如果大于0 进入wait 这样就算提前被别人叫醒 也会在继续wait

最终代码实现

```
package com.shadow.guarded;

import lombok.extern.slf4j.Slf4j;

@Slf4j(topic = "enjoy")
public class GuardedObjectTimeOut {
    private Object response;

    Object lock = new Object();

    /**
     * 加锁获取 response的值 如果response 没有值则等待
     * @return
     */
    public Object getResponse(long millis){
        synchronized (lock) {
            //开始时间
            long begin = System.currentTimeMillis();
            //经历了多少时间 开始肯定是0
            long timePassed = 0;
            while (response == null) {
                long waitTime = millis-timePassed;
                log.debug("主线程  判断如果没有结果则wait{}毫秒",waitTime);
                if (waitTime <= 0) {
                    log.debug("超时了 直接结束while 不等了");
                    break;
                }
                try {
                    lock.wait(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //如果被别人提前唤醒 先不结束 先計算一下经历时间
                timePassed = System.currentTimeMillis() - begin;
                log.debug("经历了: {}", timePassed);
            }
        }
        return response;
    }


    /**
     * t1 给response设置值
     * @param response
     */
    public void setResponse(Object response) {
        synchronized (lock) {
            this.response = response;
            //设置完成之后唤醒主线程
            lock.notifyAll();
        }
    }
```

```
}
```

死锁

如果线程需要获取多把锁那么就很可能会发现死锁

```
package com.shadow.lock;

import jdk.nashorn.internal.ir.Block;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class LockTest {

    //定义两把锁
    static Object x = new Object();
    static Object y = new Object();

    public static void main(String[] args) {
        //线程1启动
        new Thread(()->{
            //获取x的锁
            synchronized (x){
                log.debug("locked x");
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (y){
                    log.debug("locked x");
                    log.debug("t1---------");
                }
            }

        },"t1").start();



        new Thread(()->{
            synchronized (y){
                log.debug("locked y");
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (x){
                    log.debug("locked x");
                    log.debug("t2---------");
                }
            }
```

```
        },"t2").start();
    }


}
```

活锁

不可避免 但是我可以错开他们的执行时间

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class LockTest1 {
    static volatile int count = 10;
    static final Object lock = new Object();


    public static void main(String[] args) {
        //t1线程对count一直做减法 直到减为0才结束
        new Thread(() -> {
            while (count > 0) {
                try {
                    TimeUnit.NANOSECONDS.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                count--;
                log.debug("count: {}", count);
            }
        }, "t1").start();


        //t2线程对count一直做加法 直到加为20才结束
        new Thread(() -> {
            while (count < 20) {
                try {
                    TimeUnit.NANOSECONDS.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                count++;
                log.debug("count: {}", count);
            }
        }, "t2").start();
    }
}
```

Lock--应用

特点:

1. 可打断，可重入
2. 可以设置超时时间
3. 可以设置为公平锁
4. 支持多个条件变量
5. 支持读写锁(单独的篇章来讲)

基本语法

```
// 获取锁
reentrantLock.lock();
try {
// 临界区
} finally {
// 释放锁
reentrantLock.unlock();
}
```

重入

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.locks.ReentrantLock;

@Slf4j(topic = "enjoy")
public class LockTest3 {

    //首先定义一把锁
    static ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        lock1();
    }


    public static void lock1() {
        lock.lock();
        try {
            log.debug("执行lock1");
            //重入
            lock2();
        } finally {
            lock.unlock();
        }
    }
```

```
    public static void lock2() {
        lock.lock();
        try {
            log.debug("执行lock2");
        } finally {
            lock.unlock();
        }
    }

}
```

可打断

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

@Slf4j(topic = "enjoy")
public class LockTest4 {

    public static void main(String[] args) throws InterruptedException {
        ReentrantLock lock = new ReentrantLock();


        //t2首先获取锁 然后阻塞5s
        new Thread(()->{
            try {
                lock.lock();
                log.debug("获取锁----");
                TimeUnit.SECONDS.sleep(5);
                log.debug("t2 5s 之后继续执行");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }

        },"t2").start();


        TimeUnit.SECONDS.sleep(1);

        //t1加锁失败因为被t2持有
        Thread t1 = new Thread(() -> {
            try {
                lock.lockInterruptibly();
                log.debug("获取了锁--执行代码");
            } catch (InterruptedException e) {
                e.printStackTrace();
                log.debug("被打断了没有获取锁");
```

```
                return;
            } finally {
                lock.unlock();
            }
        }, "t1");
        t1.start();



        //由于t1 可以被打断 故而1s之后打断t1 不在等待t2释放锁了
        try {
            log.debug("主线程------1s后打断t1");
            TimeUnit.SECONDS.sleep(2);
            t1.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

```
t---线程
lock.lockInterruptibly();
标识可以打断

怎么打断

t.interrupt();
```

超时

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

@Slf4j(topic = "enjoy")
public class LockTest5 {

    public static void main(String[] args) throws InterruptedException {
        ReentrantLock lock = new ReentrantLock();
        Thread t1 = new Thread(() -> {
            log.debug("t1启动---------");
            try {
                if (!lock.tryLock(2,TimeUnit.SECONDS)) {//嘗試获取锁,如果失败则返回
                    log.debug("拿不到鎖，返回");
                    return;
                }
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                log.debug("获得了锁");
            } finally {
                lock.unlock();
            }
        }, "t1");
        lock.lock();
        log.debug("主綫程获得了锁");
        t1.start();
        try {
            TimeUnit.SECONDS.sleep(3);
        } finally {
            lock.unlock();
        }

    }
}
```

多個條件

synchronized 中也有条件变量，就是以前讲的waitSet 不满足条件的线程进入waitSet；而Lock也有waitSet而且有多个

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

@Slf4j(topic = "enjoy")
public class TestWait5 {

    static final ReentrantLock lock = new ReentrantLock();
    static boolean isPrettyGril = false; // 女人
    static boolean isMoney = false;//工资

    //没有女人的 waitSet
    static Condition waitpg = lock.newCondition();
    // 没有钱的waitSet
    static Condition waitm = lock.newCondition();


    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
           try {
               lock.lock();
                   log.debug("有没有女人[{}]", isPrettyGril);
                   while (!isPrettyGril) {
                       log.debug("没有女人！等女人");
                       try {
                           waitpg.await();
```

```
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                log.debug("男女搭配干活不累，啪啪啪写完了代码");
            }finally {
                lock.unlock();
            }
        }, "jack").start();



        new Thread(() -> {
            try {
                lock.lock();

                log.debug("有没有工资[{}]", isMoney);
                while (!isMoney) {
                    log.debug("没有工资！等发工资");
                    try {
                        waitm.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                log.debug("-----卧槽好多钱，啪啪啪写完了代码");

            }finally {
                lock.unlock();
            }
        }, "rose").start();



       Thread.sleep(1000);
        new Thread(() -> {
            try {
                lock.lock();
                isMoney = true;
                log.debug("钱来哦了");
                waitm.signal();
                isPrettyGril=true;
                waitpg.signal();

            }finally {
                lock.unlock();
            }
        }, "boss").start();
    }



}
```

读写锁

读读并发

读写互斥

写写互斥

读写锁读锁不支持条件

```
读锁的条件直接调用ReentrantReadWriteLock的 newCondition 会直接exception
public Condition newCondition() {
    throw new UnsupportedOperationException();
}
```

读写锁使用的例子

```
package com.shadow.lock;

import com.shadow.aqs.CustomSync;
import lombok.extern.slf4j.Slf4j;


import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.*;


@Slf4j(topic = "enjoy")
public class LockTest10 {

      static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
      static Lock r = rwl.readLock();
      static Lock w = rwl.writeLock();


    public static void main(String[] args) throws InterruptedException {

        //读
        new Thread(()->{
            log.debug("read 获取 锁");
            r.lock();
            try {
                for (int i = 0; i < 10; i++) {
                    m1(i);
                }
            }finally {
                r.unlock();
            }


        },"t1").start();

        //写
        new Thread(()->{
            log.debug("write 获取 锁");
            w.lock();
            try {
```

```
                for (int i = 0; i < 20; i++) {
                    m1(i);
                }
            }finally {
                w.unlock();
            }

        },"t2");


        //读
        new Thread(()->{
            log.debug("write 获取 锁");
            r.lock();
            try {
                for (int i = 0; i < 20; i++) {
                    m1(i);
                }
            }finally {
                r.unlock();
            }

        },"t3").start();

    }


    public static void m1(int i){
            log.debug("exe"+i);
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

读写支持重入但是只支持降级不止升级

```
package com.shadow.lock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;


@Slf4j(topic = "enjoy")
public class LockTest11 {

      static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

```
    static Lock r = rwl.readLock();
    static Lock w = rwl.writeLock();


public static void main(String[] args) throws InterruptedException {

    new Thread(() -> {
        log.debug("read");
        w.lock();
        try {
            log.debug("read 已经获取");
            r.lock();
            log.debug("write 已经获取");
        } finally {
            r.unlock();
            w.unlock();

        }


    }, "t1").start();


}
}
```

```
//缓存
class CachedData {
    Object data;
    //判断缓存是否过期
    volatile boolean cacheValid;
    //定义一把读写锁
    final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    //处理缓存的方法
     void processCachedData() {
      rwl.readLock().lock();
      //如果缓存没有过期则调用 use(data);
      if (!cacheValid) {//要去load真实数据    set给缓存拿到写锁
        //释放读锁 因为不止升级 所以需要先释放
        rwl.readLock().unlock();


        rwl.writeLock().lock();
        try {
          //双重检查
          if (!cacheValid) {
            data = "数据库得到真实数据"
            cacheValid = true;
          }

          //更新缓存之后接着读取 所以先加锁
          rwl.readLock().lock();
        } finally {
          rwl.writeLock().unlock(); // Unlock write, still hold read
        }
      }
```

```
        try {
            //不管上面的if进不进都会执行这里
          //缓存可用
          use(data);
        } finally {
          rwl.readLock().unlock();
        }
      }
    }
}
```

AQS框架

定义

1、全称是 AbstractQueuedSynchronizer

2、阻塞式锁和相关的同步器工具的框架；

3、AQS用一个变量（volatile state） 属性来表示锁的状态，子类去维护这个状态

3、getState、compareAndSetState cas改变这个变量

4、独占模式是只有一个线程能够访问资源

5、而共享模式可以允许多个线程访问资源（读写锁）

6、内部维护了一个FIFO等待队列，类似于 synchronized关键字当中的 Monitor 的 EntryList

7、条件变量来实现等待、唤醒机制，支持多个条件变量，类似于 Monitor 的 WaitSet

8、内部维护了一个Thread exclusiveOwnerThread 来记录当前持有锁的那个线程

功能

1、实现阻塞获取锁 acquire 拿不到锁就去阻塞 等待锁被释放再次获取锁

2、实现非阻塞尝试获取锁 tryAcquire 拿不到锁则直接放弃

3、实现获取锁超时机制

4、实现通过打断来取消

5、实现独占锁及共享锁

6、实现条件不满足的时候等待

自定义实现AQS框架

继承AQS 实现其主要方法

```
package com.shadow.aqs;

import java.util.concurrent.locks.AbstractQueuedLongSynchronizer;
```

```
import java.util.concurrent.locks.Condition;

public class CustomSync extends AbstractQueuedLongSynchronizer {
    @Override
    public boolean tryAcquire(long acquires) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }


    @Override
    protected boolean tryRelease(long arg) {

        if(getState() == 0) {
            throw new IllegalMonitorStateException();
        }
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    @Override
    protected boolean isHeldExclusively() {
        return getState()==1;
    }


    public Condition newCondition() {
        return new ConditionObject();
    }
}
```

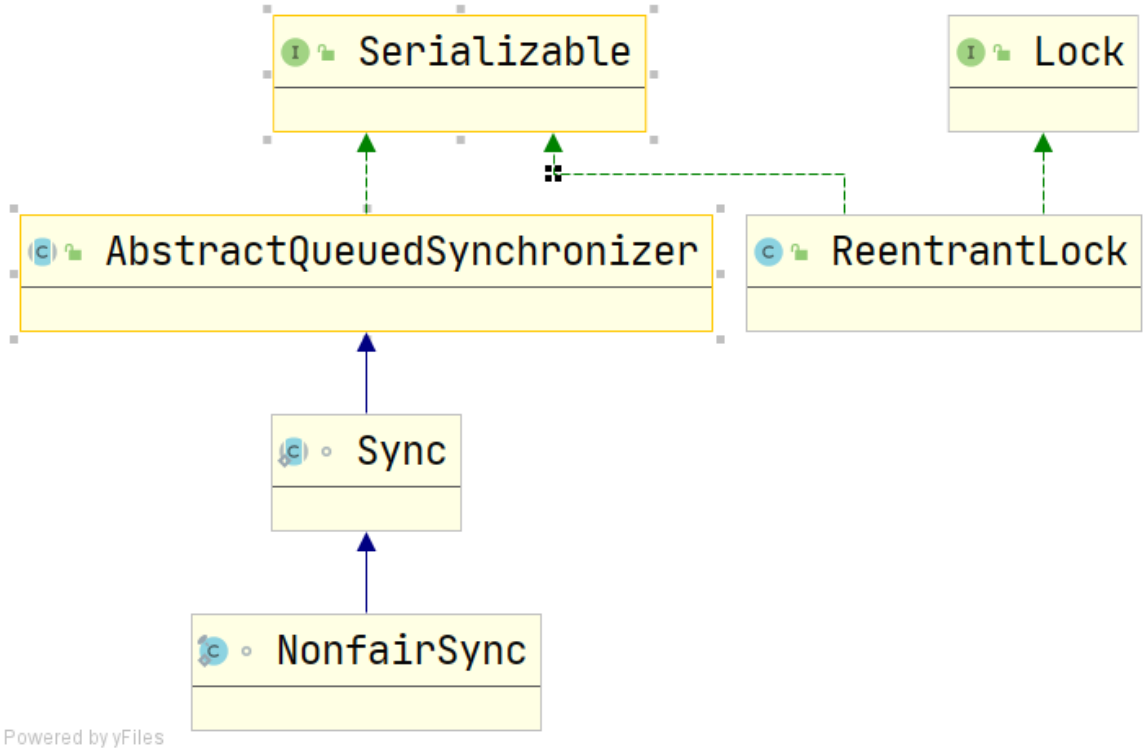
实现Lock接口实现加锁解锁

```
package com.shadow.lock;

import com.shadow.aqs.CustomSync;
import lombok.extern.slf4j.Slf4j;


import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.*;


@Slf4j(topic = "enjoy")
public class LockTest10 implements Lock{
    CustomSync customSync = new CustomSync();


    @Override
    public void lock() {
        customSync.acquire(1);
```

```
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        customSync.acquireInterruptibly(1);
    }

    @Override
    public boolean tryLock() {
        return customSync.tryAcquire(1);
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException
{
        return customSync.tryAcquireNanos(1, unit.toNanos(time));
    }

    @Override
    public void unlock() {
        customSync.release(1);
    }

    @Override
    public Condition newCondition() {
        return customSync.newCondition();
    }


    public static void main(String[] args) throws InterruptedException {
        LockTest10 l = new LockTest10();
        new Thread(()->{
            l.lock();
            log.debug("xxx");
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            l.unlock();
        },"t1").start();

        TimeUnit.SECONDS.sleep(1);
        l.lock();
        log.debug("main");
        l.unlock();
    }
}
```

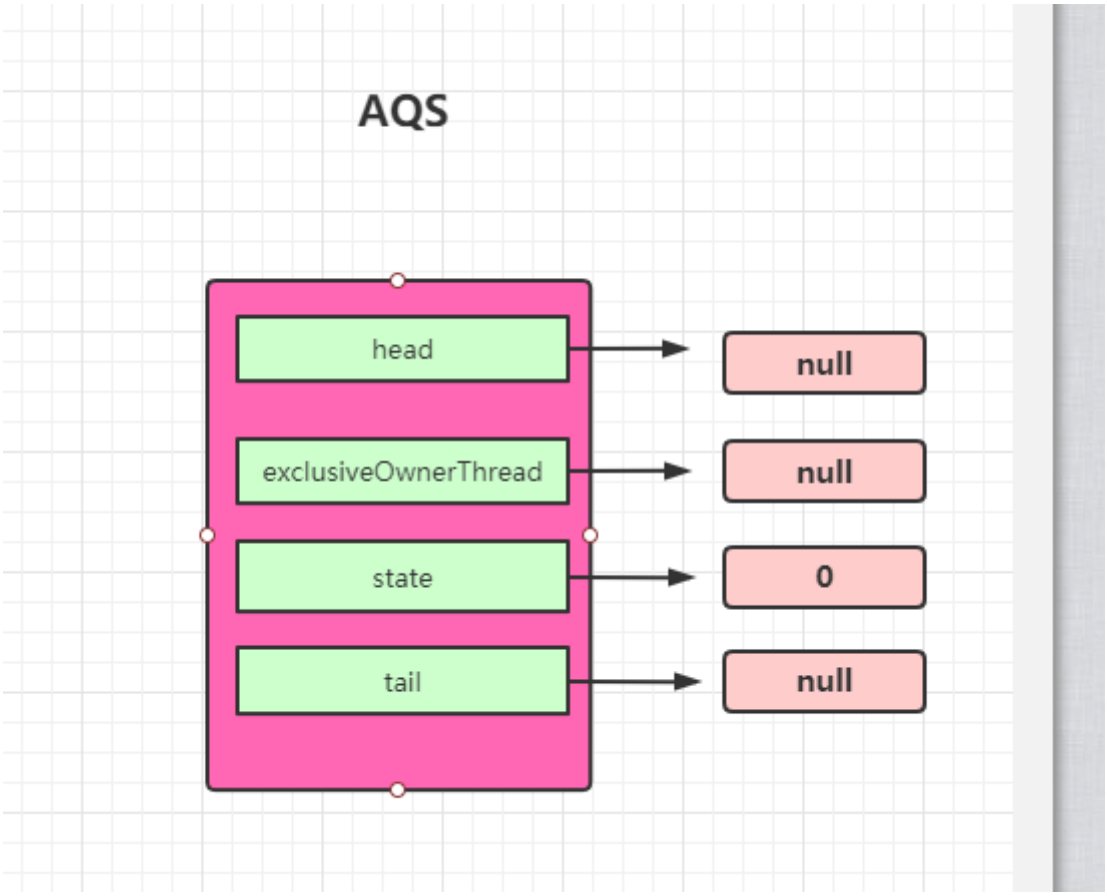
ReentrantLock

和Aqs的关系

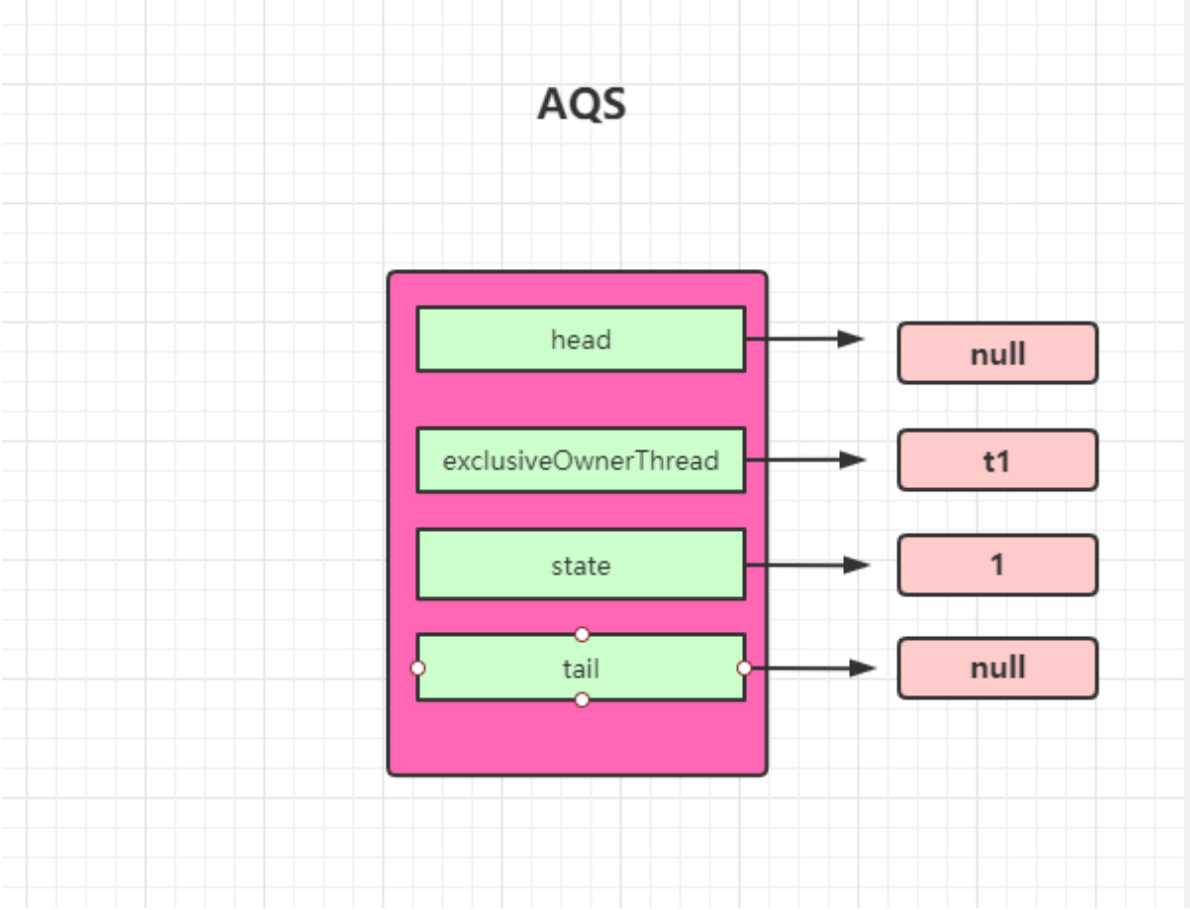


非公平锁加锁流程

1、第一个线程t1、第一次加锁，没有加锁之前 aqs（NonfairSync）的状态



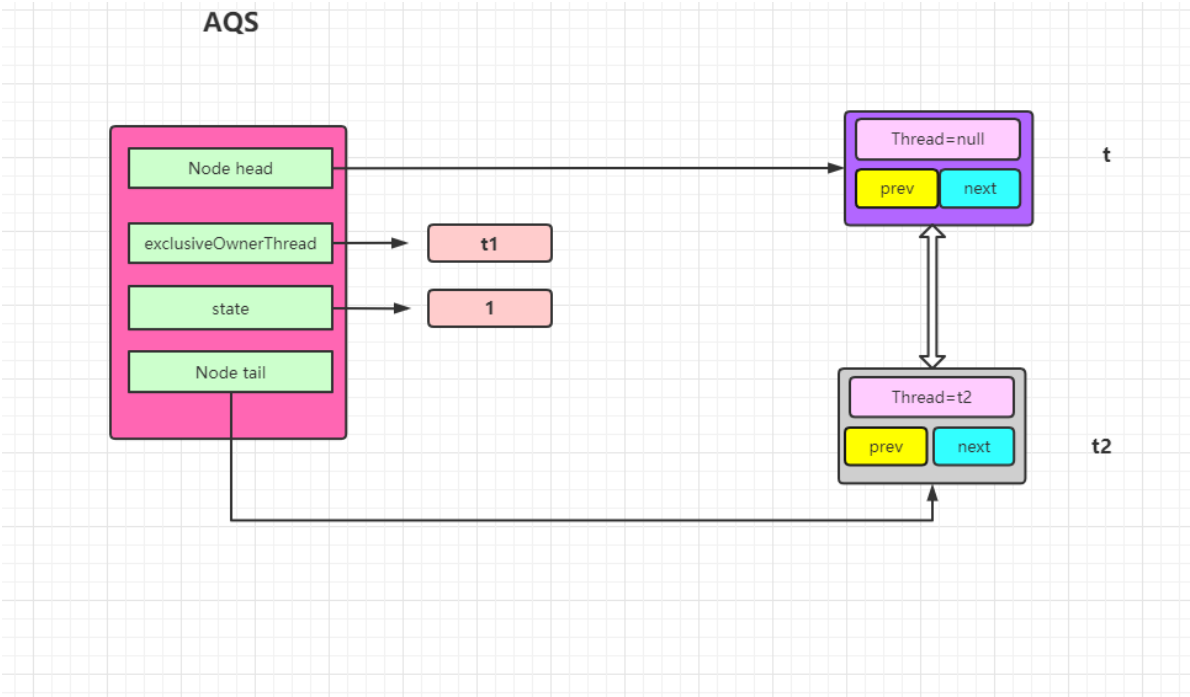
2、t1、加锁之后



、

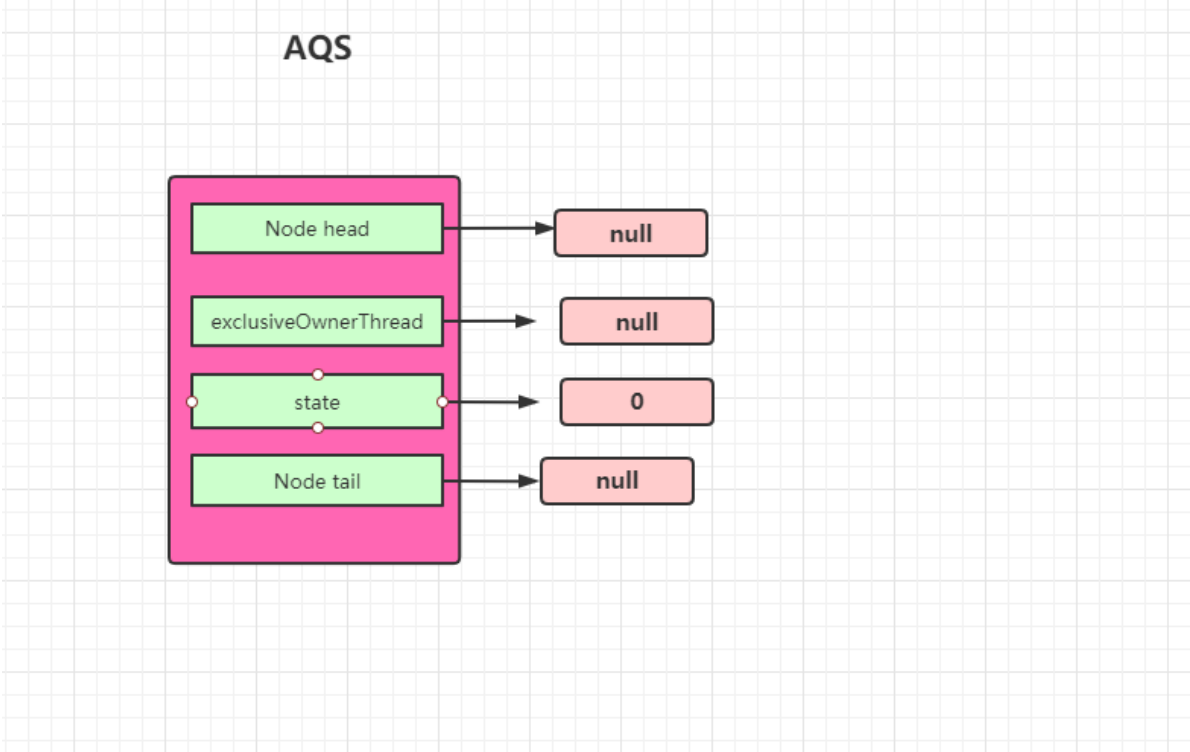
3、第二个线程t2 如果没有释放 AQS状态和2一样

3.1 t2 加锁之后失败之后

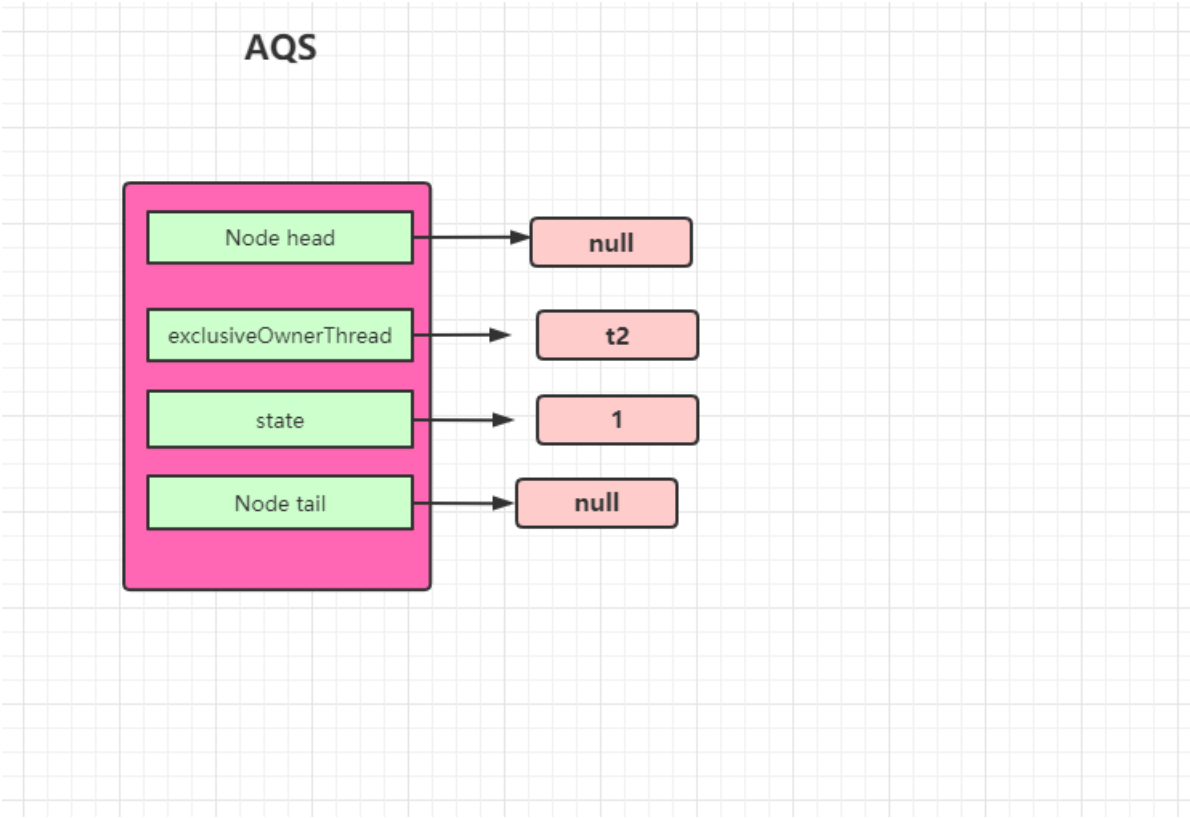


3.2 t2 加锁成功之后

如果需要加锁成功则t1必须释放锁

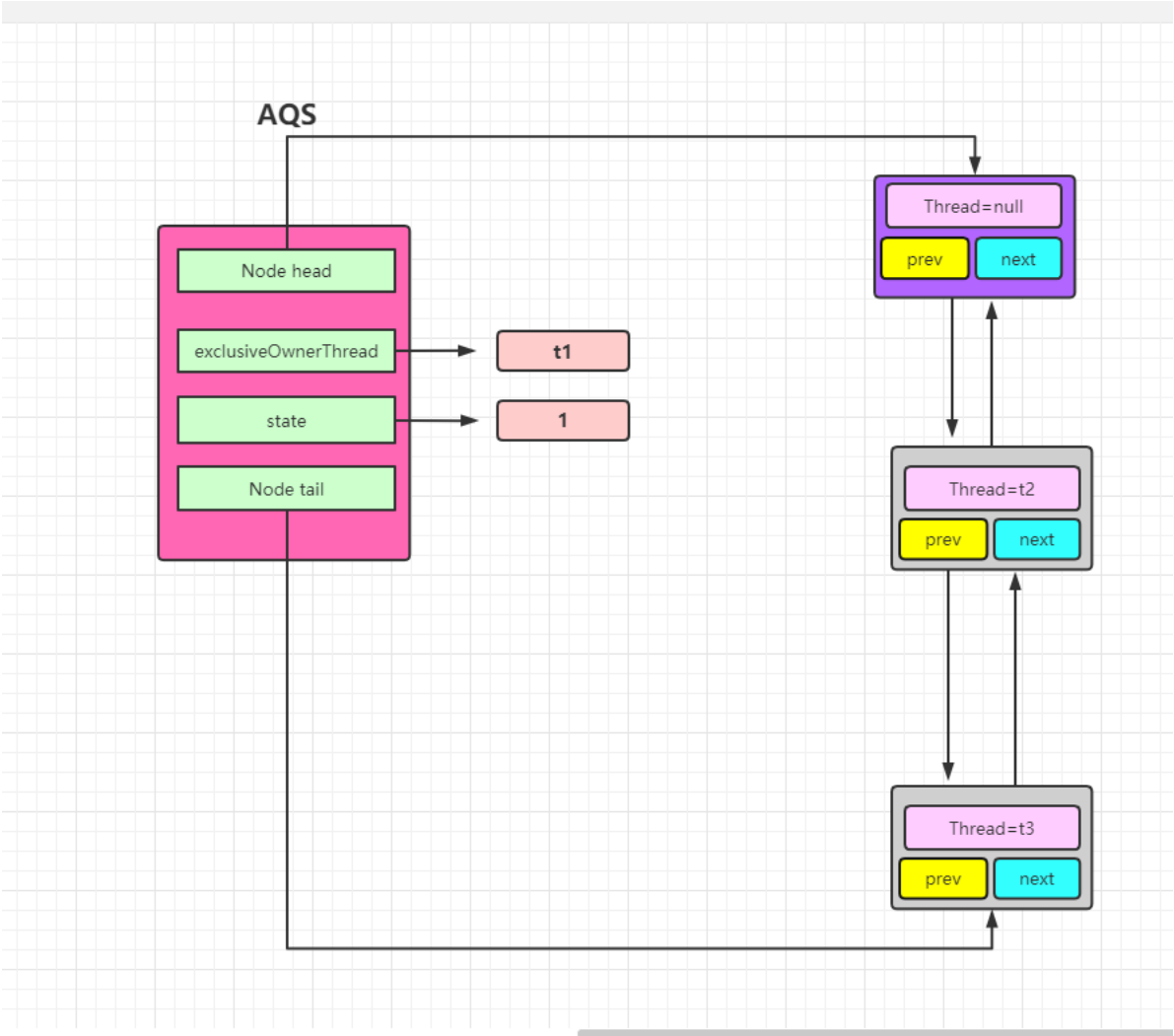


然后t2加锁成功



结论：ReentrantLock如果线程之间没有竞争效率非常高；甚至队列都没有初始化

4、t3来加锁



解锁流程

线程打断

sleep wait join这种线程如果被打断则会直接清除打断标记；什么叫做打断标记 他有什么用?

1、怎么看线程的打断标记 t1.isInterrupted()

2、打断标记顾名思义就是标记自己在这个线程是否被别人打断过了 清除 标记改为false

```
package com.shadow.aqs;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class Lock3 {

    public static void main(String[] args) throws InterruptedException {

        Thread t1 = new Thread(() -> {
            log.debug("t1------");
            try {
                TimeUnit.SECONDS.sleep(100);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }, "t1");
        t1.start();

        //主要为了让子线程 t1先运行
        TimeUnit.SECONDS.sleep(1);
        //本来你这里做了对t1的打断操作 为什么sleep要清除
        t1.interrupt();
        //false
        log.debug("t1的打断标记[{}]",t1.isInterrupted());
    }




}
```

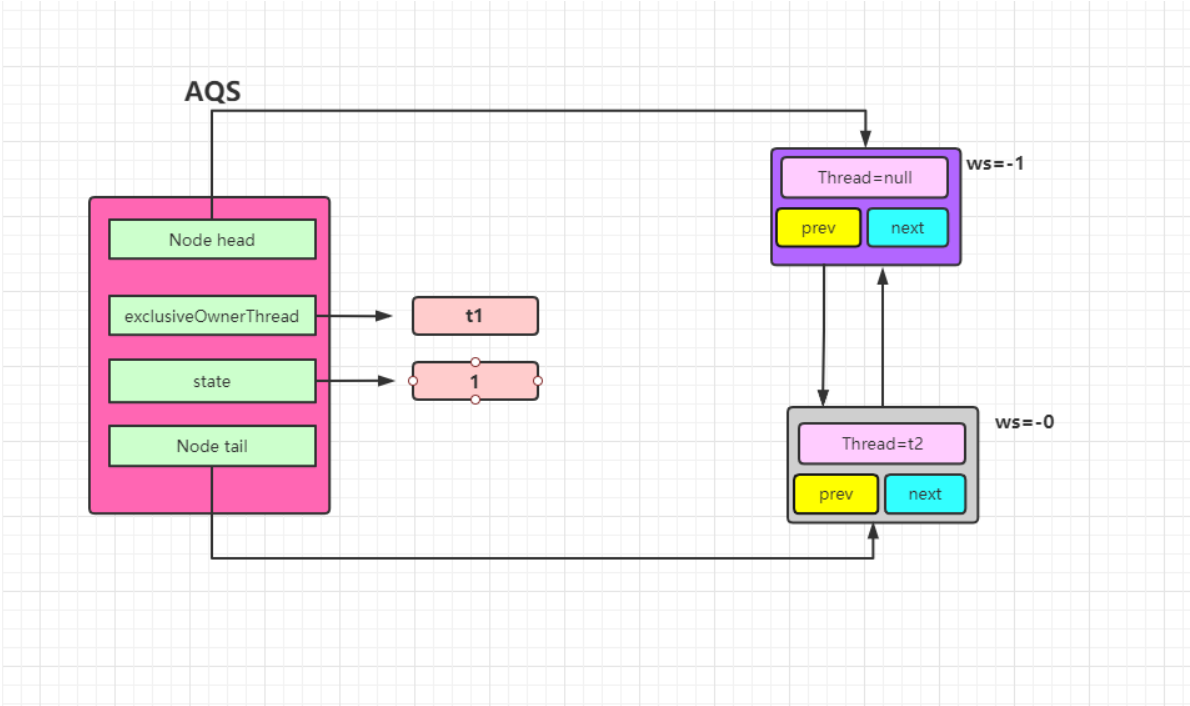
```
package com.shadow.aqs;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class Lock3 {

    public static void main(String[] args) throws InterruptedException {

        Thread t1 = new Thread(() -> {
            log.debug("t1------");
            try {
                TimeUnit.SECONDS.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "t1");
        t1.start();

        //主要为了让子线程 t1先运行
        //TimeUnit.SECONDS.sleep(1);
        //本来你这里做了对t1的打断操作 为什么sleep要清除
        t1.interrupt();
        //true?因为你打断是一个正常的线程（非sleep)
        log.debug("t1的打断标记[{}]",t1.isInterrupted());
    }




}
```

为什么sleep的线程被打断之后会清除打断标记，正常线程调用interrupt()不能被打断 给你一个打断标记 你自己根据标记去判断,如果是sleep则会响应打断所以会清除打断标记;

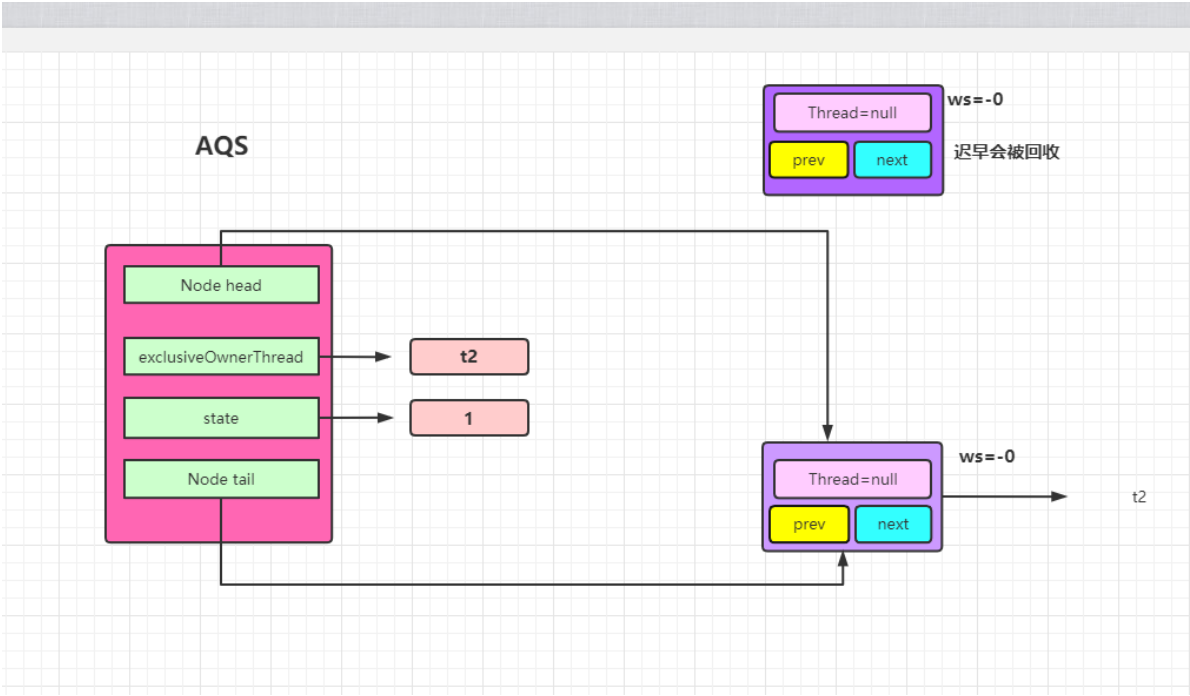
解锁流程

1、t1持有锁t2在park 然后 t1释放锁唤醒t2

before



解锁之后



读写锁的原理

关于读读并发需要注意的

比如先有一个t1写锁拿到锁，后面有一些其他锁或许是读或许是写在park；当t1释放锁之后活安装FIFO的原则唤醒等待的线程；如果第一个被唤醒的是t2写锁则无可厚非；不会再跟着唤醒t3，只有等t2执行完成之后才会去唤醒T3；假设被唤醒的t3是读锁，那么t3会去判断他的下一个t4是不是读锁如果是则把t4唤醒；t4唤醒之后会判断t5是不是读锁；如果t5也是则唤醒t5；依次类推；但是假设t6是写锁则不会唤醒t6了；即使后面的t7是读锁也不会唤醒t7；下面这个代码说明了这个现象

```
package com.shadow.aqs;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

@Slf4j(topic = "enjoy")
public class Lock2 {
    //读写锁
    static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

    static Lock r = rwl.readLock();
    static Lock w = rwl.writeLock();



    public static void main(String[] args) throws InterruptedException {

        /**
         * t1  最先拿到写（w）锁 然后睡眠了5s
         * 之后才会叫醒别人
         */
        Thread t1 = new Thread(() -> {
            w.lock();

            try {
                log.debug("t1 +");
                TimeUnit.SECONDS.sleep(5);
                log.debug("5s 之后");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                w.unlock();
            }
        }, "t1");

        t1.start();

        TimeUnit.SECONDS.sleep(1);
```

```
/**
 * t1在睡眠的过程中 t2不能拿到 读写互斥
 * t2 一直阻塞
 */

Thread t2 = new Thread(() -> {


    try {
        r.lock();
        log.debug("t2----+锁-------");
        TimeUnit.SECONDS.sleep(1);
    } catch (Exception e) {

        e.printStackTrace();
    } finally {
        log.debug("t2-----解锁-------");
        r.unlock();
    }
}, "t2");
t2.start();

TimeUnit.SECONDS.sleep(1);


/**
 * t1在睡眠的过程中 t3不能拿到 读写互斥
 * t3 一直阻塞
 *
 * 当t1释放锁之后 t3和t2 能同时拿到锁
 * 读读并发
 */
Thread t3 = new Thread(() -> {
    try {
        r.lock();
        log.debug("t3----+锁-------");
        TimeUnit.SECONDS.sleep(1);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        log.debug("t3----释放-------");

        r.unlock();
    }
}, "t3");
t3.start();


/**
 * 拿写锁
 * t1睡眠的时候 t4也页阻塞
 * 顺序应该 t2 t3  t4
 */

Thread t4 = new Thread(() -> {
    try {
```

```
                w.lock();
                log.debug("t4--------+---");
                TimeUnit.SECONDS.sleep(10);
                log.debug("t4--------醒来---");
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                log.debug("t4--------解锁---");
                w.unlock();
            }
        }, "t4");

        t4.start();


        /**
         *
         * t5 是读锁
         * 他会不会和t2 t3 一起执行
         */

        Thread t5 = new Thread(() -> {


            try {
                r.lock();
                log.debug("t5--------+锁---");
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                log.debug("t5--------解锁---");
                r.unlock();
            }
        }, "t5");

        t5.start();


    }



}
```

读写锁 写锁上锁流程

```
package com.shadow.rwlock;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 写锁的上锁流程
```

```
 */
@Slf4j(topic = "enjoy")
public class RWLock2 {
    //读写锁
    static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    static Lock r = rwl.readLock();
    static Lock w = rwl.writeLock();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            w.lock();
            try {
                log.debug("t1 w---加锁成功");
            } finally {
                w.unlock();
            }
        }, "t1");
        t1.start();
    }
}
```

```
//写锁在加锁的时候要么锁没有被人持有则会成功，要么锁是重入 否则都失败
protected final boolean tryAcquire(int acquires) {
    /*
     *1、获取当前线程
     */
    Thread current = Thread.currentThread();
    //获取锁的状态----默认是0
    int c = getState();
    //因为读写锁是同步一把锁(同一个对象),所以为了标识读写锁他把锁的前16位标识读锁的状态 后16位标识写锁的状态
    //获取写锁的状态
    int w = exclusiveCount(c);
    //标识有人上了锁(maybe w or r)
    if (c != 0) {
        // (Note: if c != 0 and w == 0 then shared count != 0)
        /*
         *1、判断当前锁是什么锁。如果是只有读锁直接加锁失败
         *为什么呢？因为w==0 标识这把锁从来没有上过写锁，只能是读锁
         *而当前自己是来上写锁的所以只能是升级 故而失败
         * 2、如果需要进到第二个判断|| 标识第一个失败了也就是这把锁有可能上了写锁 也有可能上了读写锁
         * 判断是否重入 如果不是重入失败
         *
         */
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        //重入了把w+1 标识的长度有限 但是这个判断基本没用
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // Reentrant acquire
        //没用超出重入的最大限制 则把w+1
        setState(c + acquires);
```

```
            return true;
        }

        //writerShouldBlock 要不要排队
        //如果正常情况下就是当前这个例子第一次加锁
        //writerShouldBlock 判断队列当中是有有人排队 如果有人排队 如果是公平锁则自己去排队 非公平锁则不排队
        //如果是非公平则不管有没有人排队直接抢锁
        //公平 如果队列当中没人 则不需要排队则（writerShouldBlock（）=false） 加锁
        //公平 如果队列当中有人 则需要排队则（writerShouldBlock（）=true） 加锁  会执行 if快当中的reture false 标识加锁失败
        if (writerShouldBlock() ||
            !compareAndSetState(c, c + acquires)){
             return false;
        }

        //加锁成功则把当前持有锁的线程设置自己
        setExclusiveOwnerThread(current);
        return true;
    }
```

讀鎖上锁流程

```
加锁失败则返回-1  成功则返回>0
tryAcquireShared(arg) < 0
```

```
//这里主要是为了性能 缓存了第一次和最后一次加锁的信息
Thread current = Thread.currentThread();
int c = getState();

//首先判断是否被上了写锁
//exclusiveCount(c) != 0 标识上了写锁
// 但是还会继续判断为什么上了写锁还要继续判断-----》重入降级
//然后再判断是否是重入如果这里是重入则一定是降级 如果不是重入则失败  读写需要互斥
if (exclusiveCount(c) != 0 &&
    getExclusiveOwnerThread() != current)
    return -1;
//如果上面代码没有返回执行到这里有两种情况标识1、没有人上写锁 2、重入降级
int r = sharedCount(c);//得到r的上锁次数
if (!readerShouldBlock() &&
    r < MAX_COUNT &&
    compareAndSetState(c, c + SHARED_UNIT)) {
    //r 是加锁之前的
    //r == 0 标识 这是第一次给这把锁加读锁 之前没有人加锁

    if (r == 0) {
        //如果是第一个线程第一次加锁（之前没有人加过锁）,则把这个线程付给firstReader 局部变量
        firstReader = current;
        //记录一下当前线程的加锁次数
        firstReaderHoldCount = 1;
```

```
    } else if (firstReader == current) {
        firstReaderHoldCount++;
    } else {
        HoldCounter rh = cachedHoldCounter;
        if (rh == null || rh.tid != getThreadId(current))
            cachedHoldCounter = rh = readHolds.get();
        else if (rh.count == 0)
            readHolds.set(rh);
        rh.count++;
    }
    //加锁成功
    return 1;
}
return fullTryAcquireShared(current);
```

高性能读写锁StampedLock

ReentrantReadWriteLock 的性能已经很好了但是他底层还是需要进行一系列的cas操作去加锁；StampedLock如果是读锁上锁是没有这种cas操作的性能比ReentrantReadWriteLock 更好

也称为乐观读锁；即读获取锁的时候 是不加锁 直接返回一个值；然后执行临界区的时候去验证这个值是否有被人修改（写操作加锁）

如果没有被人修改则直接执行临界区的代码；如果被人修改了则需要升级为读写锁(ReentrantReadWriteLock--->readLock);

基本语法：

```
//获取戳 不存在锁
long stamp = lock.tryOptimisticRead();
//验证戳
if(lock.validate(stamp)){
    //成立则执行临界区的代码
    //返回
}
//如果没有返回则表示被人修改了 需要升级成为readLock
lock.readLock();
```

代码示例

```
package com.shadow.stampedLock;

import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;

import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.StampedLock;

/**
 * 一个数据容器
 * 不支持重入
 * 不支持条件
```

```
 */
@Slf4j(topic = "enjoy")
public class DataContainer {
    int i;
    long stampw=0l;

    public void setI(int i) {
        this.i = i;
    }

    private final StampedLock lock = new StampedLock();

    //首先 加 StampedLock
    @SneakyThrows
    public int read() {
        //尝试一次乐观读
        long stamp = lock.tryOptimisticRead();

        log.debug("StampedLock 读锁拿到的戳{}", stamp);
        //1s之后验戳
        TimeUnit.SECONDS.sleep(1);
        //验戳
        if (lock.validate(stamp)) {
            log.debug("StampedLock 验证完毕stamp{}, data.i:{}", stamp, i);
            return i;
        }
        //一定验证失败
        log.debug("验证失败 被写线程给改变了{}", stampw);
        try {
            //锁的升级 也会改戳
            stamp = lock.readLock();
            log.debug("升级之后的加锁成功 {}", stamp);
            TimeUnit.SECONDS.sleep(1);
            log.debug("升级读锁完毕{}, data.i:{}", stamp, i);
            return i;
        } finally {
            log.debug("升级锁解锁 {}", stamp);
            lock.unlockRead(stamp);
        }
    }



    @SneakyThrows
    public void write(int i) {
        //cas 加鎖
        stampw = lock.writeLock();
        log.debug("写锁加锁成功 {}", stampw);
        try {
            TimeUnit.SECONDS.sleep(5);
            this.i = i;
        } finally {
            log.debug("写锁解锁 {},data.i{}", stampw,i);
            lock.unlockWrite(stampw);
        }
    }
}
```

```java
package com.shadow.stampedLock;

import java.util.concurrent.TimeUnit;

public class StampedLockTest {

    public static void main(String[] args) throws InterruptedException {
        //实例化数据容器
        DataContainer dataContainer = new DataContainer();
        //给了一个初始值  不算写 构造方法赋值
        dataContainer.setI(1);

        //读取
        new Thread(() -> {
            dataContainer.read();
        }, "t1").start();


//        new Thread(() -> {
//            dataContainer.read();
//        }, "t2").start();

        TimeUnit.SECONDS.sleep(1);
        new Thread(() -> {
            dataContainer.write(9);
        }, "t2").start();
    }
}
```

那么StampedLock的性能这么好能否替代ReentrantReadWriteLock？

1、他不支持重入

2、不支持条件队列

3、存在一定的并发问题

samephore

来限制对资源访问的线程的上限；好比洗浴店里面的手牌，比如你进去一个洗浴店里，服务生首先会给你一个手牌；如果手牌没有了你则需要去喝茶等待；等他其他问洗完你才可以去享受服务；手牌相当于你一个许可；你去享受服务的时候先要获取手牌，服务完成之后需要归还手牌；

基本语法

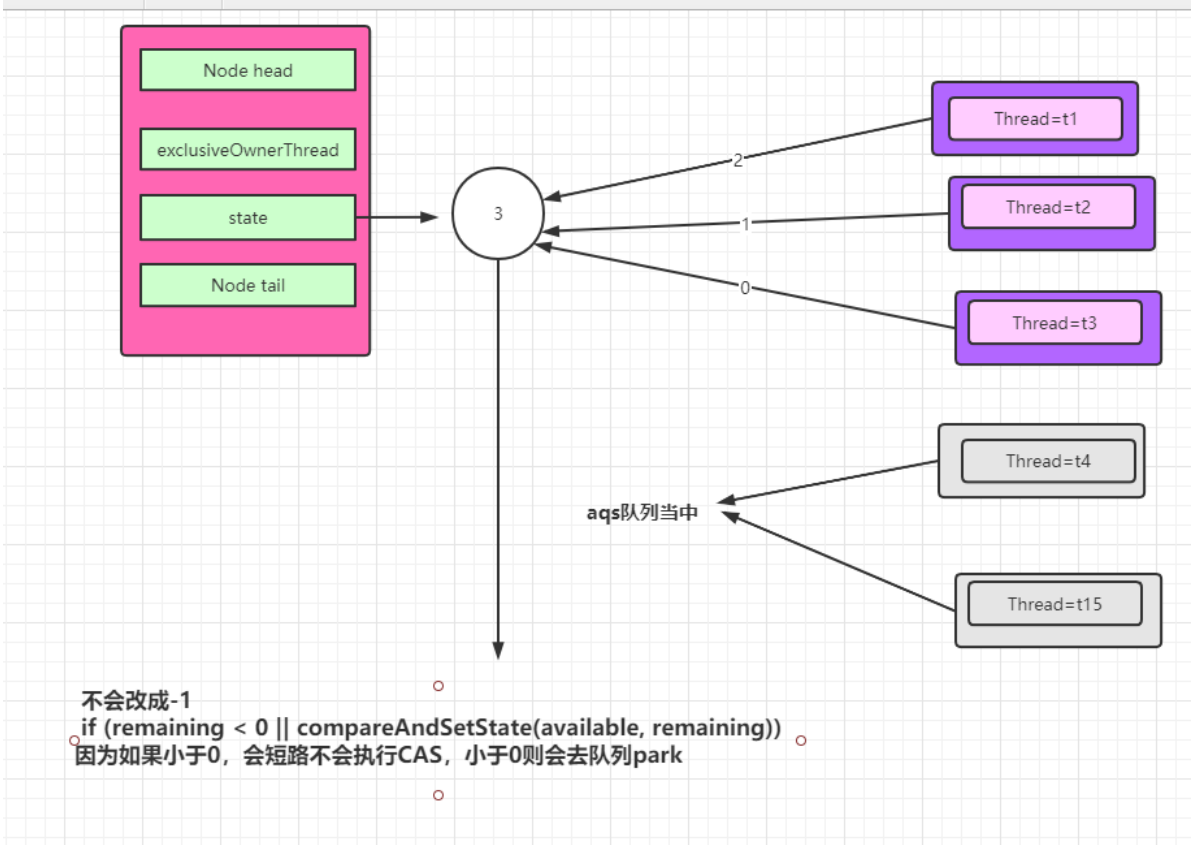
```
//线程的上限
Semaphore semaphore = new Semaphore(3);
//获取一个许可 -
semaphore.acquire();
//释放 +
semaphore.release();
```

代码示例

```
package com.shadow.semaphore;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 来限制对资源访问的线程上线
 */
@Slf4j(topic = "enjoy")
public class SemaphoreTest {
    public static void main(String[] args) {

        //每次访问的线程上限是3
        Semaphore semaphore = new Semaphore(3);

        for (int i = 0; i < 15; i++) {
            new Thread(() -> {
                try {
                    semaphore.acquire();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                try {
                    log.debug("start...");
                    TimeUnit.SECONDS.sleep(1);
                    log.debug("end...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release();
                }
            }).start();
        }
    }
}
```

注意只能限制手牌上限，就是客人的数量，你架不住有的技师今天心情不好不想工作；所以可能店里有100个技师；但是只有60个手牌也就是只能最多服务60个客人；

samephore原理分析



CountDownLatch

倒计时锁；某个线程x等待倒计时为0的时候才执行；所谓的倒计时其实就是一个int类型的变量，在初始化CountDownLatch的时候会给他一个初始值（程序员定的）；在多线程工作的时候可以通过countDown()方法来对计数器-1；当等于0的时候x则会解阻塞运行

基本语法

```
//初始化对象，给一个初始值
CountDownLatch latch = new CountDownLatch(3);

//x线程 调用await阻塞 等待计数器为0的时候才会解阻塞
latch.await();

//其他线程调用countDown();对计数器-1
latch.countDown();
```

代码示例

```
package com.shadow.countDownLatch;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class CountDownLatchTest1 {

    public static void main(String[] args) throws InterruptedException {
        //计数器=3
        CountDownLatch latch = new CountDownLatch(3);

        new Thread(() -> {
            log.debug("t1 thread start");
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //t1 把计数器-1
            latch.countDown();
            log.debug("t1 thread end;count[{}]", latch.getCount());
        },"t1").start();

        new Thread(() -> {
            log.debug("t2 thread start");
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            latch.countDown();
            log.debug("t2 thread end;count[{}]", latch.getCount());
        },"t2").start();

        new Thread(() -> {
            log.debug("t3 thread start");
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            latch.countDown();//主线程可以执行了
            log.debug("t3 thread end;count[{}]", latch.getCount());
        },"t3").start();

        log.debug("main watiing");
        latch.await();

        log.debug("main wait end...");
    }
```

```
}
```

CountDownLatch和join 区别

1、join是一定等待线程执行完成才解阻塞

2、当线程对象Thread对象不明确的时候不能用join ExecutorService

```
package com.shadow.countDownLatch;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

@Slf4j(topic = "enjoy")
public class CountDownLatchTest2 {

    public static void main(String[] args) throws InterruptedException {
        //线程池里面创建4个线程  其中三个是计算的  第四个是汇总的
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        CountDownLatch latch = new CountDownLatch(3);


        executorService.submit(()->{
            log.debug("t1 thread start");
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            latch.countDown();
            log.debug("t1 thread end;count[{}]", latch.getCount());
        });



        executorService.submit(()->{
            log.debug("t2 thread start");
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            latch.countDown();
            log.debug("t2 thread end;count[{}]", latch.getCount());
        });

        executorService.submit(()->{
            log.debug("t3 thread start");
            try {
```

```
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            latch.countDown();
            log.debug("t3 thread end;count[{}]", latch.getCount());
        });


        executorService.submit(()->{
            log.debug("t4 watiing");
            try {
                latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.debug("t4 wait end...");

        });

        executorService.shutdown();
    }
}
```

```
package com.shadow.countDownLatch;

import lombok.extern.slf4j.Slf4j;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

@Slf4j(topic = "enjoy")
public class CountDownLatchTest3 {

    public static void main(String[] args) throws InterruptedException {
        //线程池里面创建3个线程
        List<String> list = new ArrayList<>();
        list.add("Angel");
        list.add("baby");
        list.add("rose");
        list.add("joyce");

        AtomicInteger i= new AtomicInteger();
        ExecutorService executorService = Executors.newFixedThreadPool(4,
(runnable)->{
            //技师的名字
            return new Thread(runnable,list.get(i.getAndIncrement()));
        });

        //让你先去沙发上休息
        CountDownLatch latch = new CountDownLatch(4);
        Random random = new Random();
```

```
        for (int j = 0; j <4 ; j++) {//new 4个线程  并发执行
            int temp =j;
            executorService.submit(()->{
                //k标识的是准备进度 直到准备到100% 才开始服务  这个时间每个技师不固定  因为是random
                for (int k = 0; k <100 ; k++) {
                    try {
                        //模拟每一个技师准备的时间
                        TimeUnit.MILLISECONDS.sleep(random.nextInt(200));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    String name = Thread.currentThread().getName();
                    name=name+"("+k+"%)";//angel(3%) baby(10%) ...
                    list.set(temp,name);
                    System.out.print("\r"+Arrays.toString(list.toArray()));
                }
                //某个人准备好了
                latch.countDown();
            });
        }
        latch.await();
        System.out.println("\n 登上人生巅峰...");
        executorService.shutdown();
    }
}
```

CyclicBarrier

cyclicBarrier 重复栅栏（CountDownLatch），语法和CountDownLatch差不多

基本语法

```
//初始化一个cyclicBarrier 计数器为2
CyclicBarrier cyclicBarrier = new CyclicBarrier(2);
//阻塞 计数器不为0的时候并且会把计数器-1
cyclicBarrier.await();
```

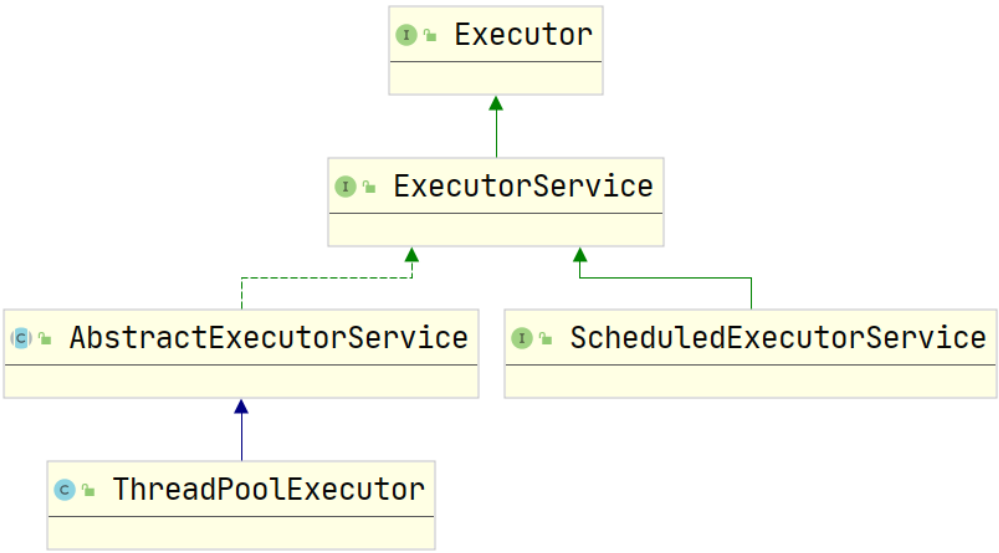
示例代码：

```
package com.shadow.cyclic;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;
@Slf4j(topic = "enjoy")
```

```
public class CyclicBarrierTest {
    public static void main(String[] args) {
        AtomicInteger i= new AtomicInteger();

        CyclicBarrier cyclicBarrier = new CyclicBarrier(2,()->{

            log.debug("t1 t2 end");
        });


        ExecutorService service = Executors.newFixedThreadPool(2);
        for (int j = 0; j <2 ; j++) {
            service.submit(()->{
                log.debug("start");
                try {
                    TimeUnit.SECONDS.sleep(1);
                    log.debug("working");
                    cyclicBarrier.await();
                } catch (Exception e) {
                    e.printStackTrace();
                }


            });

            service.submit(()->{
                log.debug("start");
                try {
                    TimeUnit.SECONDS.sleep(3);
                    log.debug("working");
                    cyclicBarrier.await();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        service.shutdown();
    }
}
```

和CountDownLatch区别 可以重复执行 然后构造方法可以直接提供一个 阻塞的线程等待计数器为0的时候再执行

线程池ExecutorService



线程池状态

ThreadPoolExecutor 使用 int 的高 3 位来表示线程池状态，低 29 位表示线程数量

状态	value	说明
RUNNING（当线程池创建出来的初始状态）	111	能接受任务，能执行阻塞任务
SHUTDOWN(调用shutdown方法)	000	不接受新任务，能执行阻塞任务 肯定可以 執行正在執行的任務
STOP（调用shutDownNow）	001	不接受新任务，打断正在执行的任务，丢弃阻塞任务
TIDYING（中间状态）	010	任务全部执行完，活动线程也没了
TERMINATED（终结状态）	011	线程池终结

构造方法1

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

1、核心线程数

2、最大线程数（应急线程数||空闲线程）

3、针对空闲线程的存活时间 如果超时了则把空闲的线程kill

4、针对3的时间单位

5、任务存放的队列

6、线程工厂，主要是产生线程---给线程起个自定义名字

7、拒绝策略

工作方式

线程池中刚开始没有线程，当一个任务提交给线程池后，线程池会创建一个新线程来执行任务

当线程数达到 核心线程数上限，这时再加入任务，新加的任务会被加入队列当中去

前提是有界队列，任务超过了队列大小时，会创建 maximumPoolSize - corePoolSize 数目的线程数目作为空闲线程来执行任务

如果线程到达 maximumPoolSize 仍然有新任务这时会执行拒绝策略

工厂方法

```
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory);
}
```

```
 ExecutorService executorService = Executors.newFixedThreadPool(n);
```

核心线程数 == 最大线程数（没有救急线程被创建），因此也无需超时时间

阻塞队列是无界的，可以放任意数量的任务

适用于任务量已知，相对耗时的任务

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}

ExecutorService executorService = Executors.newCachedThreadPool()
```

核心线程数是 0， 最大线程数是 Integer.MAX_VALUE，全部都是空闲线程60s后回收

一个可根据需要创建新线程的线程池，如果现有线程没有可用的，则创建一个新线程并添加到池中，如果有被使用完但是还没销毁的线程，就复用该线程。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源

这种线程池比较灵活，对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}

 ExecutorService executorService = Executors.newSingleThreadExecutor();
```

希望多个任务排队执行,线程数固定为 1，任务数多于 1 时，会放入无界队列排队,任务执行完毕，这唯一的线程
也不会被释放。
区别于自己创建一个单线程串行执行任务，如果任务执行失败而终止那么没有任何补救措施，而线程池还会新建一
个线程，保证池的正常工作
Executors.newSingleThreadExecutor() 线程个数始终为1，不能修改，
Executors.newFixedThreadPool(1) 初始时为1，以后还可以修改，对外暴露的是 ThreadPoolExecutor 对象，可以强转后调用 setCorePoolSize 等方法进行修改;

提交任务

```
void execute(Runnable command);
```

提交一个任务

```
<T> Future<T> submit(Callable<T> task);

Future<?> submit(Runnable task);

<T> Future<T> submit(Runnable task, T result);
```

提交一个任务有返回值

```
List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws
InterruptedException;
```

提交所有的任务

```
T invokeAny(Collection<? extends Callable<T>> tasks) throws
InterruptedException, ExecutionException;
```

提交 tasks 中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消

shutdown

```
线程池状态变为SHUTDOWN
不会接收新任务
但已提交任务会执行完（包含了在队列当中）
不会阻塞调用线程的执行
void shutdown();



线程池状态变为 STOP
不会接收新任务
会将队列中的任务返回
并用 interrupt 的方式中断正在执行的任务
List<Runnable>      会将队列中的任务返回
List<Runnable> shutdownNow();


调用 shutdown后，调用线程并不会等待所有任务运行结束,可以利用此方法等待
boolean awaitTermination(long timeout, TimeUnit unit) throws
InterruptedException;
```

JMM java内存模型

关于CPU 内存

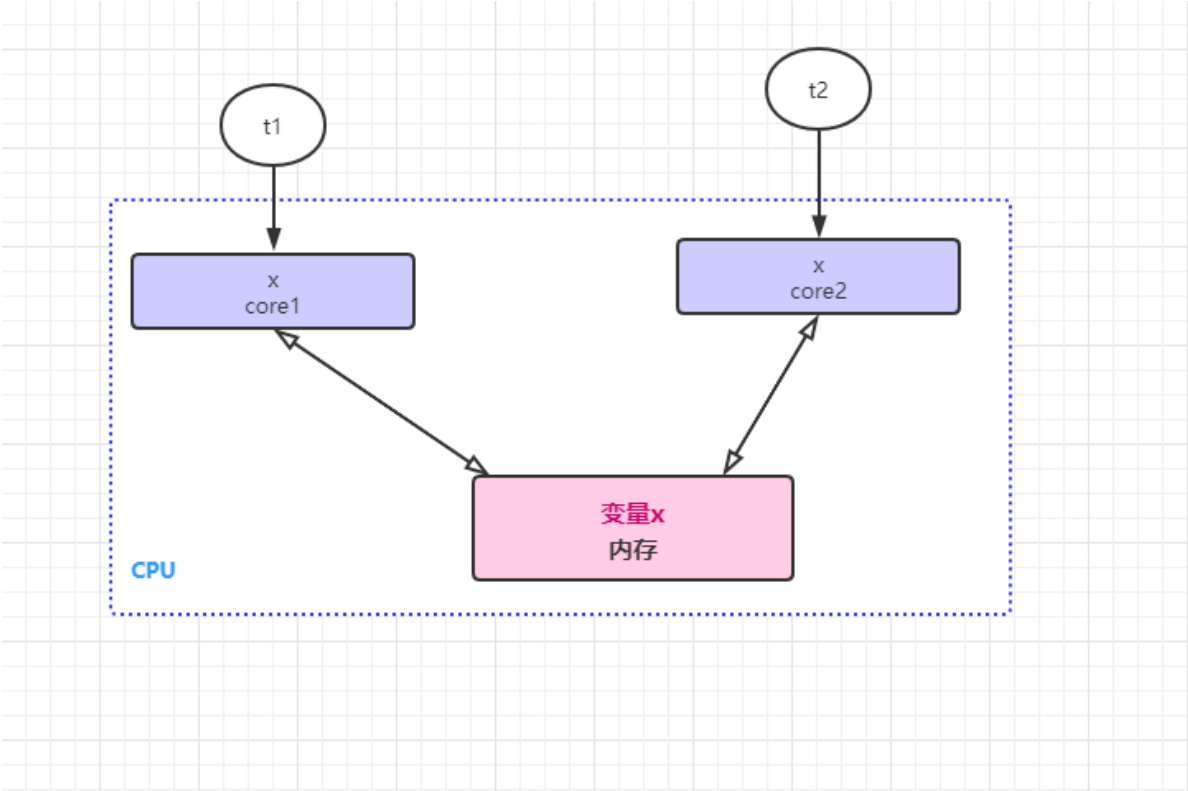
CPU、内存、I/O 设备都在不断迭代，在这个快速发展的过程中，有一个核心矛盾一直存在，就是这三者的速度差异。CPU 和内存的速度差异可以形象地描述为：CPU快于内存快于 I/O 设备，程序里大部分语句都要访问内存，有些还要访问 I/O所以程序整体的性能取决于最慢的操作——读写 I/O 设备，也就是说单方面提高 CPU 性能是无效的。为了合理利用 CPU 的高性能，平衡这三者的速度差异，计算机体系结构、操作系统、编译程序都做出了贡献，主要体现为：CPU 增加了缓存，以均衡与内存的速度差异；操作系统增加了进程、线程，以分时复用 CPU，进而均衡 CPU 与 I/O 设备的速度差异；编译程序优化指令执行次序，使得缓存能够得到更加合理地利用

可见性问题

单核pc，所有的线程都是在一颗 CPU 上执行，CPU 缓存与内存的数据一致性容易解决。因为所有线程都是操作同一个 CPU 的缓存，一个线程对缓存的写，对另外一个线程来说一定是可见的

一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称为可见性

多核时代，每个core都有自己的cache，这时 core的缓存与内存的数据一致性就没那么容易解决了，当多个线程在不同的 core 上执行时，这些线程操作的是不同的 core缓存。比如下图中，线程 A 操作的是 core1 上的缓存，而线程 B 操作的是 core2上的缓存，这个时候线程 A 对变量 x 的操作如果没有及时写会主存那么对于线程 B 而言就不具备可见性了



线程切换

编译优化

重排序可以提高处理的速度

x =3; load x   set x store x

y=4; load y   sety store y

x=x+6; load x   set x store x

x =3; load x   set x

x=x+6; set x store x

y=4; load y   sety store y

案例1

```
package com.jmm;

/**
 * @Author 钢牌讲师--子路
 **/
public class Test1 {

    // 新建几个静态变量
    public static int a = 0 , b = 0;
    public static int x = 0 , y = 0;

    public static void main(String[] args) throws Exception {
        int count = 0;
        while(true){
            count++;
            a = 0 ;
            b = 0 ;
            x = 0 ;
            y = 0 ;
            // 定义两个线程。
            // 线程A
            Thread t1 = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    x = b;
                }
            });

            // 线程B
```

```
            Thread t2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    y = a;
                }
            });

            t1.start();
            t2.start();
            t1.join(); // 让t1线程优先执行完毕
            t2.join(); // 让t2线程优先执行完毕

            // 得到线程执行完毕以后 变量的结果。
            System.out.println("第"+count+"次输出结果: i = " +x +" , j = "+y);
            if(x == 0 && y == 0){
                break;
            }
        }
    }
}
```

如果不发生指令重排的三种情况

x=0; y=1

x=1;y=0

x=1;y=1

案例2

```
public class Test5 {
    private static Test5 INSTANCE ;
    private Test5(){

    }
public static Test5 getInstance(){
    if(INSTANCE == null){
        synchronized (Test5.class){
            INSTANCE = new Test5();
        }
    }
    return INSTANCE;
}
}
```

JAVA 内存模型

你已经知道，导致可见性的原因是缓存，导致有序性的原因是编译优化，那解决可见性、有序性最直接的办法就是禁用缓存和编译优化，但是这样问题虽然解决了，我们程序的性能可就堪忧了。合理的方案应该是按需禁用缓存以及编译优化。那么，如何做到“按需禁用”呢？对于并发程序，何时禁用缓存以及编译优化只有程序员知道，那所谓“按需禁用”其实就是指按照程序员的要求来禁用。所以，为了解决可见性和有序性问题，只需要提供给程序员按需禁用缓存和编译优化的方法即可。Java 内存模型是个很复杂的规范，可以从不同的视角来解读，站在我们这些程序员的视角，本质上可以理解为，Java 内存模型规范了 JVM 如何提供按需禁用缓存和编译优化的方法。具体来说，这些方法包括 volatile、synchronized 和 final 三个关键字，以及六项 Happens-Before 规则

为什么定义Java内存模型？现代计算机体系大部是采用的对称多处理器的体系架构。每个处理器均有独立的寄存器组和缓存，多个处理器可同时执行同一进程中的不同线程，这里称为处理器的乱序执行。在Java中，不同的线程可能访问同一个共享或共享变量。如果任由编译器或处理器对这些访问进行优化的话，很有可能出现无法想象的问题，这里称为编译器的重排序。除了处理器的乱序执行、编译器的重排序，还有内存系统的重排序。因此Java语言规范引入了Java内存模型，通过定义多项规则对编译器和处理器进行限制，主要是针对可见性和有序性

Java内存模型涉及的几个关键词：锁、volatile字段、final修饰符与对象的安全发布。其中：第一是锁，锁操作是具备happens-before关系的，解锁操作happens-before之后对同一把锁的加锁操作。实际上，在解锁的时候，JVM需要强制刷新缓存，使得当前线程所修改的内存对其他线程可见。第二是volatile字段，volatile字段可以看成是一种不保证原子性的同步但保证可见性的特性，其性能往往是优于锁操作的。但是，频繁地访问 volatile字段也会出现因为不断地强制刷新缓存而影响程序的性能的问题。第三是final修饰符，final修饰的实例字段则是涉及到新建对象的发布问题。当一个对象包含final修饰的实例字段时，其他线程能够看到已经初始化的final实例字段，这是安全的

关于java的hb原则

1、程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作hb书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构；但是这个规则是对结果负责

2、一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须强调的是同一个锁，而"后面"是指时间上的先后顺序

3、volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作，这里的"后面"同样是指时间上的先后顺序

4、线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

5、线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join（）方法结束、Thread.isAlive（）的返回值等手段检测到线程已经终止执行

6、Happens-Before具备传递性