并发编程

> @Author 郑金维

一、线程的基础概念

一、基础概念

1.1 进程与线程A

什么是进程？

进程是指运行中的程序。 比如我们使用钉钉，浏览器，需要启动这个程序，操作系统会给这个程序分配一定的资源（占用内存资源）。

什么线程？

线程是CPU调度的基本单位，每个线程执行的都是某一个进程的代码的某个片段。

举个栗子：房子与人

比如现在有一个100平的房子，这个方式可以看做是一个进程

房子里有人，人就可以看做成一个线程。

人在房子中做一个事情，比如吃饭，学习，睡觉。这个就好像线程在执行某个功能的代码。

所谓进程就是线程的容器，需要线程利用进程中的一些资源，处理一个代码、指令。最终实现进程锁预期的结果。

进程和线程的区别：

- 根本不同：进程是操作系统分配的资源，而线程是CPU调度的基本单位。
- 资源方面：同一个进程下的线程共享进程中的一些资源。线程同时拥有自身的独立存储空间。进程之间的资源通常是独立的。
- 数量不同：进程一般指的就是一个进程。而线程是依附于某个进程的，而且一个进程中至少会有一个或多个线程。
- 开销不同：毕竟进程和线程不是一个级别的内容，线程的创建和终止的时间是比较短的。而且线程之间的切换比进程之间的切换速度要快很多。而且进程之间的通讯很麻烦，一般要借助内核才可以实现，而线程之间通讯，相当方面。
- ……

1.2 多线程

什么是多线程？

多线程是指：**单个进程中同时运行多个线程。**

多线程的不低是为了提高CPU的利用率。

可以通过避免一些网络IO或者磁盘IO等需要等待的操作，让CPU去调度其他线程。

这样可以大幅度的提升程序的效率，提高用户的体验。

比如Tomcat可以做并行处理，提升处理的效率，而不是一个一个排队。

不如要处理一个网络等待的操作，开启一个线程去处理需要网络等待的任务，让当前业务线程可以继续往下执行逻辑，效率是可以得到大幅度提升的。

多线程的局限

- 如果线程数量特别多，CPU在切换线程上下文时，会额外造成很大的消耗。
- 任务的拆分需要依赖业务场景，有一些异构化的任务，很难对任务拆分，还有很多业务并不是多线程处理更好。
- **线程安全问题**：虽然多线程带来了一定的性能提升，但是再做一些操作时，多线程如果操作临界资源，可能会发生一些数据不一致的安全问题，甚至涉及到锁操作时，会造成死锁问题。

1.3 串行、并行、并发

什么是串行：

串行就是一个一个排队，第一个做完，第二个才能上。

什么是并行：

并行就是同时处理。（一起上！！！）

什么是并发：

这里的并发并不是三高中的高并发问题，这里是多线程中的并发概念（CPU调度线程的概念）。
CPU在极短的时间内，反复切换执行不同的线程，看似好像是并行，但是只是CPU高速的切换。

并行囊括并发。

并行就是多核CPU同时调度多个线程，是真正的多个线程同时执行。

单核CPU无法实现并行效果，单核CPU是并发。

1.4 同步异步、阻塞非阻塞

同步与异步：执行某个功能后，被调用者是否会**主动反馈**信息

阻塞和非阻塞：执行某个功能后，调用者是否需要**一直等待结果**的反馈。

两个概念看似相似，但是侧重点是完全不一样的。

同步阻塞：比如用锅烧水，水开后，不会主动通知你。烧水开始执行后，需要一直等待水烧开。

同步非阻塞：比如用锅烧水，水开后，不会主动通知你。烧水开始执行后，不需要一直等待水烧开，可以去执行其他功能，但是需要时不时的查看水开了没。

异步阻塞：比如用水壶烧水，水开后，会主动通知你水烧开了。烧水开始执行后，需要一直等待水烧开。

异步非阻塞：比如用水壶烧水，水开后，会主动通知你水烧开了。烧水开始执行后，不需要一直等待水烧开，可以去执行其他功能。

异步非阻塞这个效果是最好的，平时开发时，提升效率最好的方式就是采用异步非阻塞的方式处理一些多线程的任务。

二、线程的创建

线程的创建分为三种方式：

2.1 继承Thread类 重写run方法

启动线程是调用start方法，这样会创建一个新的线程，并执行线程的任务。

如果直接调用run方法，这样会让当前线程执行run方法中的业务逻辑。

```
public class MiTest {

    public static void main(String[] args) {
        MyJob t1 = new MyJob();
        t1.start();
        for (int i = 0; i < 100; i++) {
            System.out.println("main:" + i);
        }
    }

}
class MyJob extends Thread{
```

```
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("MyJob:" + i);
        }
    }
}
```

2.2 实现Runnable接口 重写run方法

```
public class MiTest {

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);
        t1.start();
        for (int i = 0; i < 1000; i++) {
            System.out.println("main:" + i);
        }
    }

}

class MyRunnable implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("MyRunnable:" + i);
        }

    }
}
```

最常用的方式：

- 匿名内部类方式：

```
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("匿名内部类:" + i);
        }
```

```
    }
});
```

- lambda方式：

```
Thread t2 = new Thread(() -> {
    for (int i = 0; i < 100; i++) {
        System.out.println("lambda:" + i);
    }
});
```

2.3 实现Callable 重写call方法，配合FutureTask

Callable一般用于有返回结果的非阻塞的执行方法

同步非阻塞。

```
public class MiTest {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        //1. 创建MyCallable
        MyCallable myCallable = new MyCallable();
        //2. 创建FutureTask，传入Callable
        FutureTask futureTask = new FutureTask(myCallable);
        //3. 创建Thread线程
        Thread t1 = new Thread(futureTask);
        //4. 启动线程
        t1.start();
        //5. 做一些操作
        //6. 要结果
        Object count = futureTask.get();
        System.out.println("总和为：" + count);
    }
}

class MyCallable implements Callable{

    @Override
    public Object call() throws Exception {
        int count = 0;
        for (int i = 0; i < 100; i++) {
            count += i;
        }
        return count;
    }
}
```

2.4 基于线程池构建线程

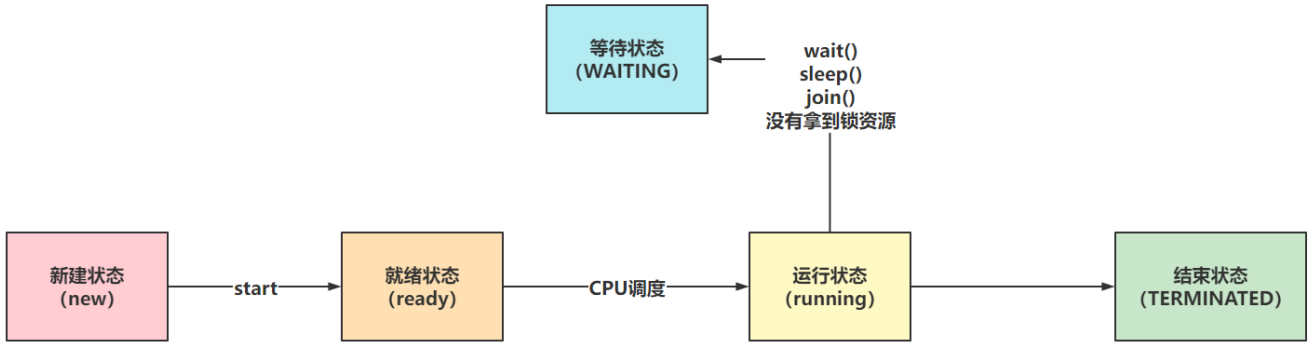
追其底层，其实只有一种，实现Runnble

二、线程的使用

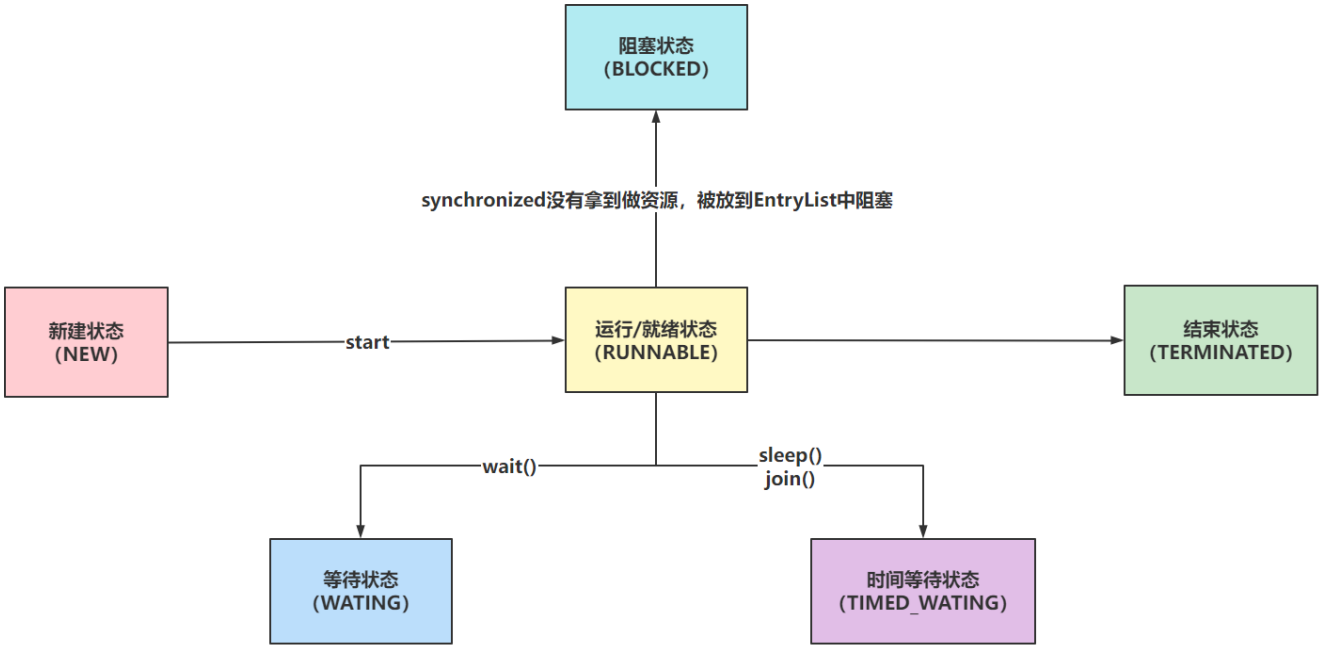
2.1 线程的状态

网上对线程状态的描述很多，有5种，6种，7种，都可以接受

5中状态一般是针对传统的线程状态来说（操作系统层面）



Java中给线程准备的6种状态



NEW：Thread对象被创建出来了，但是还没有执行start方法。

RUNNABLE：Thread对象调用了start方法，就为RUNNABLE状态（CPU调度/没有调度）

BLOCKED、WAITING、TIME_WAITING：都可以理解为是阻塞、等待状态，因为处在这三种状态下，CPU不会调度当前线程

BLOCKED：synchronized没有拿到同步锁，被阻塞的情况

WAITING：调用wait方法就会处于WAITING状态，需要被手动唤醒

TIME_WAITING：调用sleep方法或者join方法，会被自动唤醒，无需手动唤醒

TERMINATED：run方法执行完毕，线程生命周期到头了

在Java代码中验证一下效果

NEW：

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {

    });
    System.out.println(t1.getState());
}
```

RUNNABLE：

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while(true){

        }
    });
    t1.start();
    Thread.sleep(500);
    System.out.println(t1.getState());
}
```

BLOCKED：

```
public static void main(String[] args) throws InterruptedException {
    Object obj = new Object();
    Thread t1 = new Thread(() -> {
        // t1线程拿不到锁资源，导致变为BLOCKED状态
        synchronized (obj){

        }
```

```
    });
    // main线程拿到obj的锁资源
    synchronized (obj) {
        t1.start();
        Thread.sleep(500);
        System.out.println(t1.getState());
    }
}
```

WAITING：

```
public static void main(String[] args) throws InterruptedException {
    Object obj = new Object();
    Thread t1 = new Thread(() -> {
        synchronized (obj){
            try {
                obj.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t1.start();
    Thread.sleep(500);
    System.out.println(t1.getState());
}
```

TIMED_WAITING：

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    t1.start();
    Thread.sleep(500);
    System.out.println(t1.getState());
}
```

TERMINATED：

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    t1.start();
    Thread.sleep(1000);
    System.out.println(t1.getState());
}
```

2.2 线程的常用方法

2.2.1 获取当前线程

Thread的静态方法获取当前线程对象

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
        // 获取当前线程的方法
    Thread main = Thread.currentThread();
    System.out.println(main);
    // "Thread[" + getName() + "," + getPriority() + "," +  group.getName() + "]";
    // Thread[main,5,main]
}
```

2.2.2 线程的名字

在构建Thread对象完毕后，一定要设置一个有意义的名称，方面后期排查错误

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    Thread t1 = new Thread(() -> {
        System.out.println(Thread.currentThread().getName());
    });
    t1.setName("模块-功能-计数器");
    t1.start();
}
```

2.2.3 线程的优先级

其实就是CPU调度线程的优先级、

java中给线程设置的优先级别有10个级别，从1~10任取一个整数。

如果超出这个范围，会排除参数异常的错误

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            System.out.println("t1:" + i);
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            System.out.println("t2:" + i);
        }
    });
    t1.setPriority(1);
    t2.setPriority(10);
    t2.start();
    t1.start();
}
```

2.2.4 线程的让步

可以通过Thread的静态方法yield，让当前线程从运行状态转变为就绪状态。

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            if(i == 50){
                Thread.yield();
            }
            System.out.println("t1:" + i);
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            System.out.println("t2:" + i);
        }
    });
    t2.start();
    t1.start();
}
```

2.2.5 线程的休眠

Thread的静态方法，让线程从运行状态转变为等待状态

sleep有两个方法重载：

- 第一个就是native修饰的，让线程转为等待状态的效果
- 第二个是可以传入毫秒和一个纳秒的方法（如果纳秒值大于等于0.5毫秒，就给休眠的毫秒值+1。如果传入的毫秒值是0，纳秒值不为0，就休眠1毫秒）

sleep会抛出一个InterruptedException

```
public static void main(String[] args) throws InterruptedException {
    System.out.println(System.currentTimeMillis());
    Thread.sleep(1000);
    System.out.println(System.currentTimeMillis());
}
```

2.2.6 线程的强占

Thread的非静态方法join方法

需要在某一个线程下去调用这个方法

如果在main线程中调用了t1.join()，那么main线程会进入到等待状态，需要等待t1线程全部执行完毕，在恢复到就绪状态等待CPU调度。

如果在main线程中调用了t1.join(2000)，那么main线程会进入到等待状态，需要等待t1执行2s后，在恢复到就绪状态等待CPU调度。如果在等待期间，t1已经结束了，那么main线程自动变为就绪状态等待CPU调度。

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
            System.out.println("t1:" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t1.start();
    for (int i = 0; i < 10; i++) {
        System.out.println("main:" + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        if (i == 1){
            try {
                t1.join(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

2.2.7 守护线程

默认情况下，线程都是非守护线程

JVM会在程序中没有非守护线程时，结束掉当前JVM

主线程默认是非守护线程，如果主线程执行结束，需要查看当前JVM内是否还有非守护线程，如果没有JVM直接停止

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
            System.out.println("t1:" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t1.setDaemon(true);
    t1.start();
}
```

2.2.8 线程的等待和唤醒

可以让获取synchronized锁资源的线程通过wait方法进去到锁的**等待池**，并且会释放锁资源

可以让获取synchronized锁资源的线程，通过notify或者notifyAll方法，将等待池中的线程唤醒，添加到**锁池**中

notify随机的唤醒等待池中的一个线程到锁池

notifyAll将等待池中的全部线程都唤醒，并且添加到锁池

在调用wait方法和notify以及norifyAll方法时，必须在synchronized修饰的代码块或者方法内部才可以，因为要操作基于某个对象的锁的信息维护。

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        sync();
    },"t1");

    Thread t2 = new Thread(() -> {
        sync();
    },"t2");
    t1.start();
    t2.start();
    Thread.sleep(12000);
    synchronized (MiTest.class) {
        MiTest.class.notifyAll();
    }
}

public static synchronized void sync()  {
    try {
        for (int i = 0; i < 10; i++) {
            if(i == 5) {
                MiTest.class.wait();
            }
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName());
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

2.3 线程的结束方式

线程结束方式很多，最常用就是让线程的run方法结束，无论是return结束，还是抛出异常结束，都可以

2.3.1 stop方法（不用）

强制让线程结束，无论你在干嘛，不推荐使用当然当然方式，但是，他确实可以把线程干掉

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        try {
```

```
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    t1.start();
    Thread.sleep(500);
    t1.stop();
    System.out.println(t1.getState());
}
```

2.3.2 使用共享变量（很少会用）

这种方式用的也不多，有的线程可能会通过死循环来保证一直运行。

咱们可以通过修改共享变量在破坏死循环，让线程退出循环，结束run方法

```
static volatile boolean flag = true;

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while(flag){
            // 处理任务
        }
        System.out.println("任务结束");
    });
    t1.start();
    Thread.sleep(500);
    flag = false;
}
```

2.3.3 interrupt方式

共享变量方式

```
public static void main(String[] args) throws InterruptedException {
    // 线程默认情况下，    interrupt标记位：false
    System.out.println(Thread.currentThread().isInterrupted());
    // 执行interrupt之后，再次查看打断信息
    Thread.currentThread().interrupt();
    // interrupt标记位：ture
    System.out.println(Thread.currentThread().isInterrupted());
    // 返回当前线程，并归位为false interrupt标记位：ture
    System.out.println(Thread.interrupted());
    // 已经归位了
```

```
    System.out.println(Thread.interrupted());

    // ==========================================================
    Thread t1 = new Thread(() -> {
        while(!Thread.currentThread().isInterrupted()){
            // 处理业务
        }
        System.out.println("t1结束");
    });
    t1.start();
    Thread.sleep(500);
    t1.interrupt();
}
```

通过打断WAITING或者TIMED_WAITING状态的线程，从而抛出异常自行处理

这种停止线程方式是最常用的一种，在框架和JUC中也是最常见的

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while(true){
            // 获取任务
            // 拿到任务，执行任务
            // 没有任务了，让线程休眠
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                System.out.println("基于打断形式结束当前线程");
                return;
            }
        }
    });
    t1.start();
    Thread.sleep(500);
    t1.interrupt();
}
```

wait和sleep的区别？

- 单词不一样。
- sleep属于Thread类中的static方法、wait属于Object类的方法
- sleep属于TIMED_WAITING，自动被唤醒、wait属于WAITING，需要手动唤醒。
- sleep方法在持有锁时，执行，不会释放锁资源、wait在执行后，会释放锁资源。

- sleep可以在持有锁或者不持有锁时，执行。 wait方法必须在只有锁时才可以执行。

wait方法会将持有锁的线程从owner扔到WaitSet集合中，这个操作是在修改ObjectMonitor对象，如果没有持有synchronized锁的话，是无法操作ObjectMonitor对象的。

二、并发编程的三大特性

一、原子性

1.1 什么是并发编程的原子性

JMM（Java Memory Model）。不同的硬件和不同的操作系统在内存上的操作有一定差异的。Java为了解决相同代码在不同操作系统上出现的各种问题，用JMM屏蔽掉各种硬件和操作系统带来的差异。

让Java的并发编程可以做到跨平台。

JMM规定所有变量都会存储在主内存中，在操作的时候，需要从主内存中复制一份到线程内存（CPU内存），在线程内部做计算。**然后再写回主内存中（不一定！）。**

原子性的定义：原子性指一个操作是不可分割的，不可中断的，一个线程在执行时，另一个线程不会影响到他。

并发编程的原子性用代码阐述：

```
private static int count;

public static void increment(){
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    count++;
}

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
          increment();
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
```

```
            increment();
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(count);
}
```

当前程序：多线程操作共享数据时，预期的结果，与最终的结果不符。

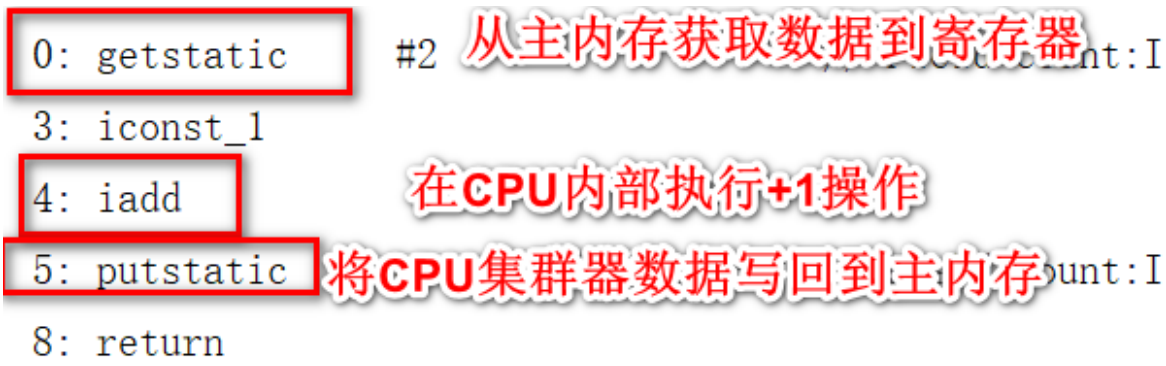
原子性：多线程操作临界资源，预期的结果与最终结果一致。

通过对这个程序的分析，可以查看出，++的操作，一共分为了三部，首先是线程从主内存拿到数据保存到CPU的寄存器中，然后在寄存器中进行+1操作，最终将结果写回到主内存当中。

1.2 保证并发编程的原子性

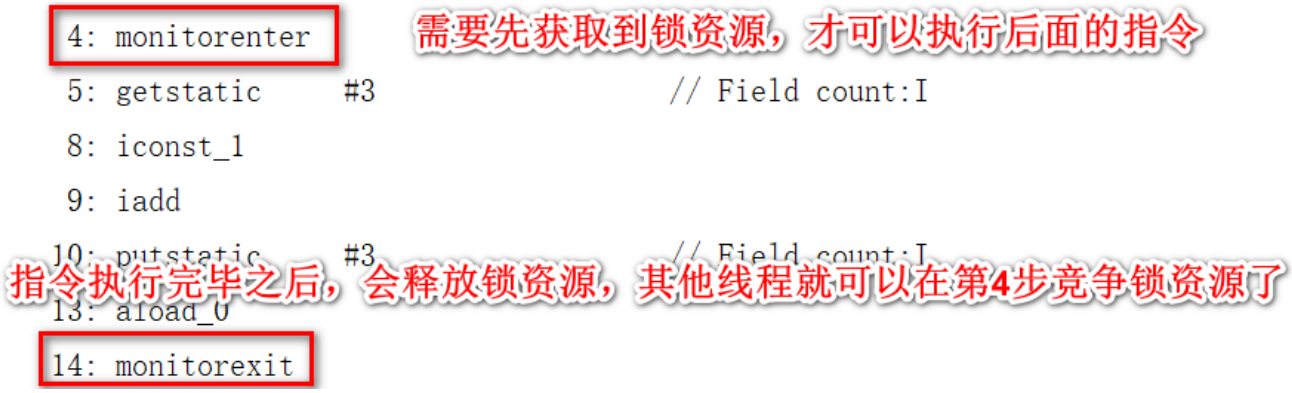
1.2.1 synchronized

因为++操作可以从指令中查看到



可以在方法上追加synchronized关键字或者采用同步代码块的形式来保证原子性

synchronized可以让避免多线程同时操作临街资源，同一时间点，只会有一个线程正在操作临界资源



1.2.2 CAS

到底什么是CAS

compare and swap也就是比较和交换，他是一条CPU的并发原语。

他在替换内存的某个位置的值时，首先查看内存中的值与预期值是否一致，如果一致，执行替换操作。这个操作是一个原子性操作。

Java中基于Unsafe的类提供了对CAS的操作的方法，JVM会帮助我们将方法实现CAS汇编指令。

但是要清楚CAS只是比较和交换，在获取原值的这个操作上，需要你自己实现。

```
private static AtomicInteger count = new AtomicInteger(0);

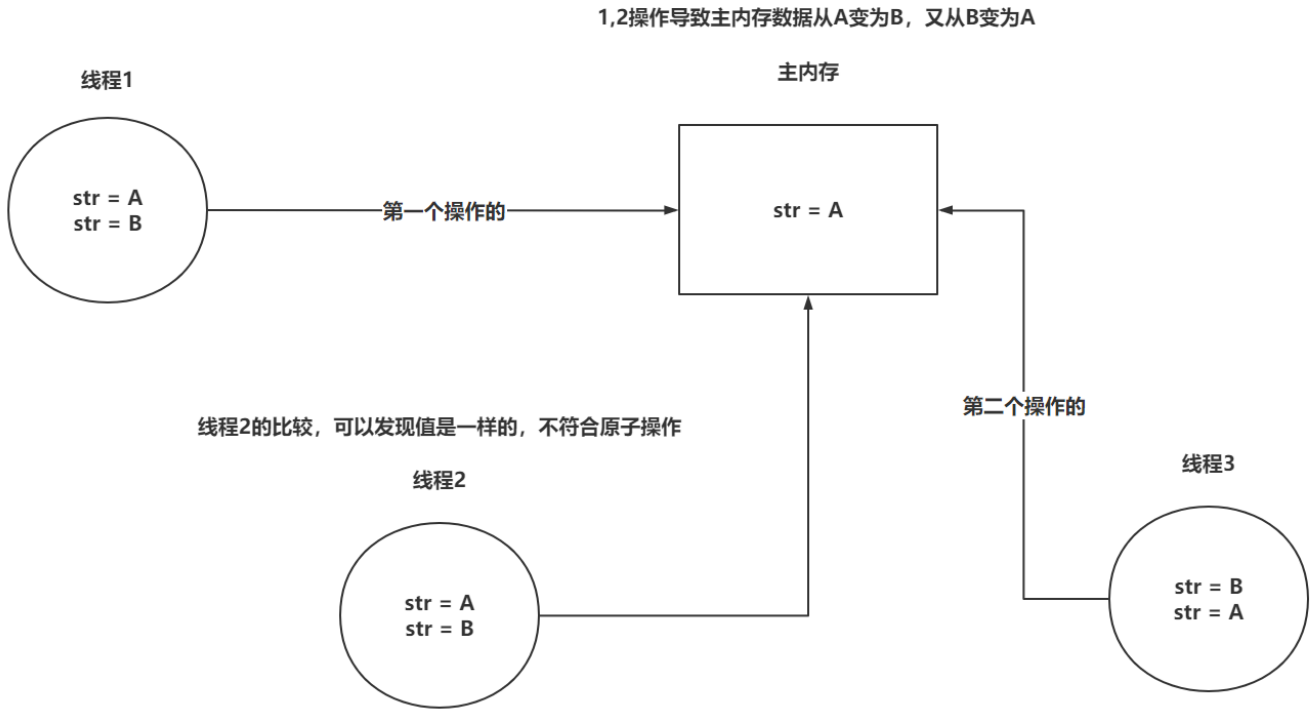
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            count.incrementAndGet();
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            count.incrementAndGet();
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(count);
}
```

Doug Lea在CAS的基础上帮助我们实现了一些原子类，其中就包括现在看到的AtomicInteger，还有其他很多原子类……

CAS的缺点：CAS只能保证对一个变量的操作是原子性的，无法实现对多行代码实现原子性。

CAS的问题：

- **ABA问题**：问题如下，可以引入版本号的方式，来解决ABA的问题。Java中提供了一个类在CAS时，针对各个版本追加版本号的操作。 AtomicStampeReference



- AtomicStampedReference在CAS时，不但会判断原值，还会比较版本信息。

- ```
  public static void main(String[] args) {
      AtomicStampedReference<String> reference = new AtomicStampedReference<>("AAA",1);

      String oldValue = reference.getReference();
      int oldVersion = reference.getStamp();

      boolean b = reference.compareAndSet(oldValue, "B", oldVersion, oldVersion + 1);
      System.out.println("修改1版本的：" + b);

      boolean c = reference.compareAndSet("B", "C", 1, 1 + 1);
      System.out.println("修改2版本的：" + c);
  }
  ```

- **自旋时间过长问题**：
  - 可以指定CAS一共循环多少次，如果超过这个次数，直接失败/或者挂起线程。（自旋锁、自适应自旋锁）
  - 可以在CAS一次失败后，将这个操作暂存起来，后面需要获取结果时，将暂存的操作全部执行，再返回最后的结果。
  ```

1.2.3 Lock锁

Lock锁是在JDK1.5由Doug Lea研发的，他的性能相比synchronized在JDK1.5的时期，性能好了很多多，但是在JDK1.6对synchronized优化之后，性能相差不大，但是如果涉及并发比较多时，推荐ReentrantLock锁，性能会更好。

实现方式：

```
private static int count;

private static ReentrantLock lock = new ReentrantLock();

public static void increment()  {
    lock.lock();
    try {
        count++;
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    } finally {
        lock.unlock();
    }



}

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            increment();
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            increment();
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(count);
}
```

ReentrantLock可以直接对比synchronized，在功能上来说，都是锁。

但是ReentrantLock的功能性相比synchronized更丰富。

ReentrantLock底层是基于AQS实现的，有一个基于CAS维护的state变量来实现锁的操作。

1.2.4 ThreadLocal

Java中的四种引用类型

Java中的使用引用类型分别是**强，软，弱，虚。**

User user = new User（ ）；

在 Java 中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它始终处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。

```
SoftReference
```

其次是软引用，对于只有软引用的对象来说，当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中，作为缓存使用。

然后是弱引用，它比软引用的生存期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。可以解决内存泄漏问题，ThreadLocal就是基于弱引用解决内存泄漏的问题。

最后是虚引用，它不能单独使用，必须和引用队列联合使用。虚引用的主要作用是跟踪对象被垃圾回收的状态。不过在开发中，我们用的更多的还是强引用。

ThreadLocal保证原子性的方式，是不让多线程去操作**临界资源**，让每个线程去操作属于自己的数据

代码实现

```
static ThreadLocal tl1 = new ThreadLocal();
static ThreadLocal tl2 = new ThreadLocal();

public static void main(String[] args) {
    tl1.set("123");
    tl2.set("456");
    Thread t1 = new Thread(() -> {
        System.out.println("t1:" + tl1.get());
        System.out.println("t1:" + tl2.get());
    });
    t1.start();
```

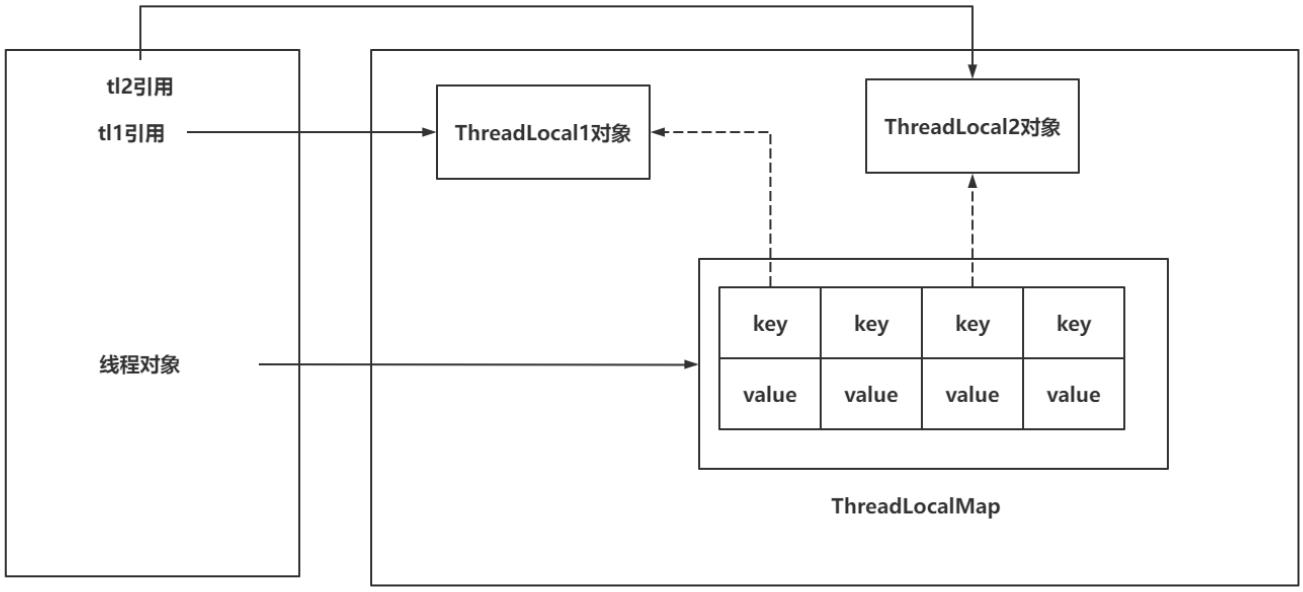
```
    System.out.println("main:" + tl1.get());
    System.out.println("main:" + tl2.get());
}
```

ThreadLocal实现原理：

- 每个Thread中都存储着一个成员变量，ThreadLocalMap
- ThreadLocal本身不存储数据，像是一个工具类，基于ThreadLocal去操作ThreadLocalMap
- ThreadLocalMap本身就是基于Entry[]实现的，因为一个线程可以绑定多个ThreadLocal，这样一来，可能需要存储多个数据，所以采用Entry[]的形式实现。
- 每一个现有都自己独立的ThreadLocalMap，再基于ThreadLocal对象本身作为key，对value进行存取
- ThreadLocalMap的key是一个弱引用，弱引用的特点是，即便有弱引用，在GC时，也必须被回收。这里是为了在ThreadLocal对象失去引用后，如果key的引用是强引用，会导致ThreadLocal对象无法被回收

ThreadLocal内存泄漏问题：

- 如果ThreadLocal引用丢失，key因为弱引用会被GC回收掉，如果同时线程还没有被回收，就会导致内存泄漏，内存中的value无法被回收，同时也无法被获取到。
- 只需要在使用完毕ThreadLocal对象之后，及时的调用remove方法，移除Entry即可

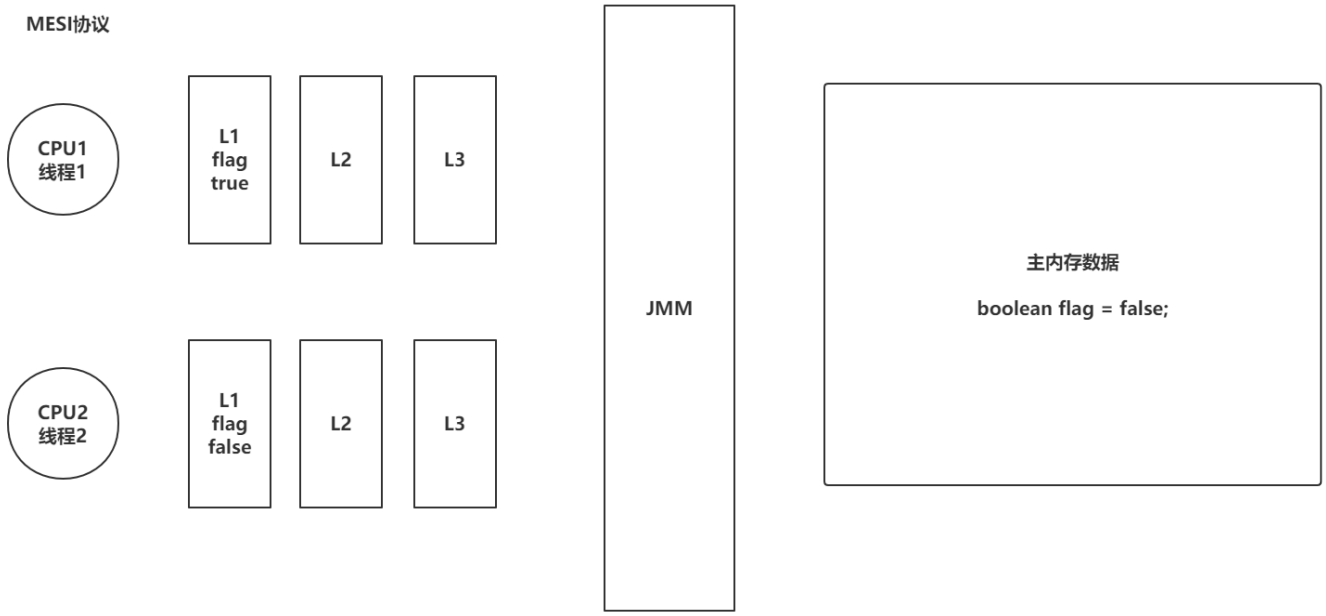


二、可见性

2.1 什么是可见性

可见性问题是基于CPU位置出现的，CPU处理速度非常快，相对CPU来说，去主内存获取数据这个事情太慢了，CPU就提供了L1，L2，L3的三级缓存，每次去主内存拿完数据后，就会存储到CPU的三级缓存，每次去三级缓存拿数据，效率肯定会提升。

这就带来了问题，现在CPU都是多核，每个线程的工作内存（CPU三级缓存）都是独立的，会告知每个线程中做修改时，只改自己的工作内存，没有及时的同步到主内存，导致数据不一致问题。



可见性问题的代码逻辑

```
private static boolean flag = true;

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while (flag) {
            // ....
        }
        System.out.println("t1线程结束");
    });

    t1.start();
    Thread.sleep(10);
    flag = false;
    System.out.println("主线程将flag改为false");
}
```

2.2 解决可见性的方式

2.2.1 volatile

volatile是一个关键字，用来修饰成员变量。

如果属性被volatile修饰，相当于会告诉CPU，对当前属性的操作，不允许使用CPU的缓存，必须去和主内存操作

volatile的内存语义：

- volatile属性被写：当写一个volatile变量，JMM会将当前线程对应的CPU缓存及时的刷新到主内存中
- volatile属性被读：当读一个volatile变量，JMM会将对应的CPU缓存中的内存设置为无效，必须去主内存中重新读取共享变量

其实加了volatile就是告知CPU，对当前属性的读写操作，不允许使用CPU缓存，加了volatile修饰的属性，会在转为汇编之后后，追加一个lock的前缀，CPU执行这个指令时，如果带有lock前缀会做两个事情：

- 将当前处理器缓存行的数据写回到主内存
- 这个写回的数据，在其他的CPU内核的缓存中，直接无效。

总结：volatile就是让CPU每次操作这个数据时，必须立即同步到主内存，以及从主内存读取数据。

```
private volatile static boolean flag = true;

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while (flag) {
            // ....
        }
        System.out.println("t1线程结束");
    });

    t1.start();
    Thread.sleep(10);
    flag = false;
    System.out.println("主线程将flag改为false");
}
```

2.2.2 synchronized

synchronized也是可以解决可见性问题的，synchronized的内存语义。

如果涉及到了synchronized的同步代码块或者是同步方法，获取锁资源之后，将内部涉及到的变量从CPU缓存中移除，必须去主内存中重新拿数据，而且在释放锁之后，会立即将CPU缓存中的数据同步到主内存。

```
private static boolean flag = true;

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while (flag) {
            synchronized (MiTest.class){
                //...
            }
            System.out.println(111);
        }
        System.out.println("t1线程结束");

    });

    t1.start();
    Thread.sleep(10);
    flag = false;
    System.out.println("主线程将flag改为false");
}
```

2.2.3 Lock

Lock锁保证可见性的方式和synchronized完全不同，synchronized基于他的内存语义，在获取锁和释放锁时，对CPU缓存做一个同步到主内存的操作。

Lock锁是基于volatile实现的。Lock锁内部再进行加锁和释放锁时，会对一个由volatile修饰的state属性进行加减操作。

如果对volatile修饰的属性进行写操作，CPU会执行带有lock前缀的指令，CPU会将修改的数据，从CPU缓存立即同步到主内存，同时也会将其他的属性也立即同步到主内存中。还会将其他CPU缓存行中的这个数据设置为无效，必须重新从主内存中拉取。

```
private static boolean flag = true;
private static Lock lock = new ReentrantLock();

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while (flag) {
            lock.lock();
            try{
```

```
                //...
            }finally {
                lock.unlock();
            }
        }
        System.out.println("t1线程结束");

    });

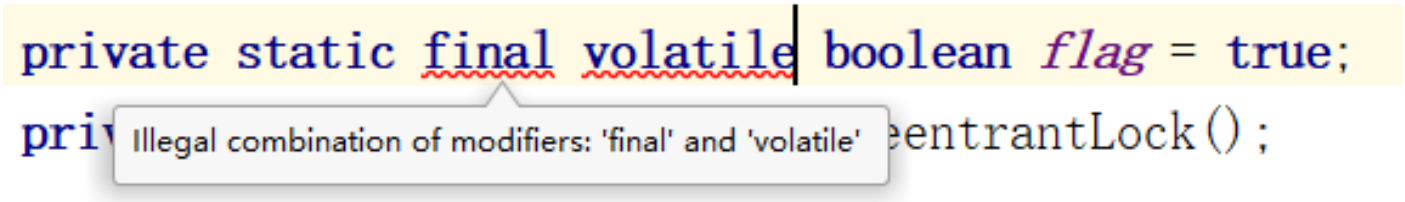
    t1.start();
    Thread.sleep(10);
    flag = false;
    System.out.println("主线程将flag改为false");
}
```

2.2.4 final

final修饰的属性，在运行期间是不允许修改的，这样一来，就间接的保证了可见性，所有多线程读取final属性，值肯定是一样。

final并不是说每次取数据从主内存读取，他没有这个必要，而且final和volatile是不允许同时修饰一个属性的

final修饰的内容已经不允许再次被写了，而volatile是保证每次读写数据去主内存读取，并且volatile会影响一定的性能，就不需要同时修饰。



三、有序性

3.1 什么是有序性

在Java中，.java文件中的内容会被编译，在执行前需要再次转为CPU可以识别的指令，CPU在执行这些指令时，为了提升执行效率，在不影响最终结果的前提下（满足一些要求），会对指令进行重排。

指令乱序执行的原因，是为了尽可能的发挥CPU的性能。

Java中的程序是乱序执行的。

Java程序验证乱序执行效果：

```
static int a,b,x,y;

public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < Integer.MAX_VALUE; i++) {
        a = 0;
        b = 0;
        x = 0;
        y = 0;

        Thread t1 = new Thread(() -> {
            a = 1;
            x = b;
        });
        Thread t2 = new Thread(() -> {
            b = 1;
            y = a;
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        if(x == 0 && y == 0){
            System.out.println("第" + i + "次，x = "+ x + ",y = " + y);
        }
    }
}
```

单例模式由于指令重排序可能会出现问题：

线程可能会拿到没有初始化的对象，导致在使用时，可能由于内部属性为默认值，导致出现一些不必要的问题

```
private static volatile MiTest test;

private MiTest(){}

public static MiTest getInstance(){
    // B
    if(test  == null){
        synchronized (MiTest.class){

            if(test == null){
```

```
                // A ， 开辟空间，test指向地址，初始化
                test = new MiTest();
            }
        }
    }
    return test;
}
```

3.2 as-if-serial

as-if-serial语义：

不论指定如何重排序，需要保证单线程的程序执行结果是不变的。

而且如果存在依赖的关系，那么也不可以做指令重排。

```
// 这种情况肯定不能做指令重排序
int i = 0;
i++;

// 这种情况肯定不能做指令重排序
int j = 200;
j * 100;
j + 100;
// 这里即便出现了指令重排，也不可以影响最终的结果，20100
```

3.3 happens-before

具体规则：

1. 单线程happen-before原则：在同一个线程中，书写在前面的操作happen-before后面的操作。

2. 锁的happen-before原则：同一个锁的unlock操作happen-before此锁的lock操作。

3. volatile的happen-before原则： 对一个volatile变量的写操作happen-before对此变量的任意操作。

4. happen-before的传递性原则： 如果A操作 happen-before B操作，B操作happen-before C操作，那么A操作happen-before C操作。

5. 线程启动的happen-before原则：同一个线程的start方法happen-before此线程的其它方法。

6. 线程中断的happen-before原则：对线程interrupt方法的调用happen-before被中断线程的检测到中断发送的代码。

7. 线程终结的happen-before原则：线程中的所有操作都happen-before线程的终止检测。
8. 对象创建的happen-before原则：一个对象的初始化完成先于他的finalize方法调用。

JMM只有在不出现上述8中情况时，才不会触发指令重排效果。

不需要过分的关注happens-before原则，只需要可以写出线程安全的代码就可以了。

3.4 volatile

如果需要让程序对某一个属性的操作不出现指令重排，除了满足happens-before原则之外，还可以基于volatile修饰属性，从而对这个属性的操作，就不会出现指令重排的问题了。

volatile如何实现的禁止指令重排？

内存屏障概念。将内存屏障看成一条指令。

会在两个操作之间，添加上一道指令，这个指令就可以避免上下执行的其他指令进行重排序。

三、锁

一、锁的分类

1.1 可重入锁、不可重入锁

Java中提供的synchronized，ReentrantLock，ReentrantReadWriteLock都是可重入锁。

重入：当前线程获取到A锁，在获取之后尝试再次获取A锁是可以直接拿到的。

不可重入：当前线程获取到A锁，在获取之后尝试再次获取A锁，无法获取到的，因为A锁被当前线程占用着，需要等待自己释放锁再获取锁。

1.2 乐观锁、悲观锁

Java中提供的synchronized，ReentrantLock，ReentrantReadWriteLock都是悲观锁。

Java中提供的CAS操作，就是乐观锁的一种实现。

悲观锁：获取不到锁资源时，会将当前线程挂起（进入BLOCKED、WAITING），线程挂起会涉及到用户态和内核的太的切换，而这种切换是比较消耗资源的。

- 用户态：JVM可以自行执行的指令，不需要借助操作系统执行。
- 内核态：JVM不可以自行执行，需要操作系统才可以执行。

乐观锁：获取不到锁资源，可以再次让CPU调度，重新尝试获取锁资源。

Atomic原子性类中，就是基于CAS乐观锁实现的。

1.3 公平锁、非公平锁

Java中提供的synchronized只能是非公平锁。

Java中提供的ReentrantLock，ReentrantReadWriteLock可以实现公平锁和非公平锁

公平锁：线程A获取到了锁资源，线程B没有拿到，线程B去排队，线程C来了，锁被A持有，同时线程B在排队。直接排到B的后面，等待B拿到锁资源或者是B取消后，才可以尝试去竞争锁资源。

非公平锁：线程A获取到了锁资源，线程B没有拿到，线程B去排队，线程C来了，先尝试竞争一波

- 拿到锁资源：开心，插队成功。
- 没有拿到锁资源：依然要排到B的后面，等待B拿到锁资源或者是B取消后，才可以尝试去竞争锁资源。

1.4 互斥锁、共享锁

Java中提供的synchronized、ReentrantLock是互斥锁。

Java中提供的ReentrantReadWriteLock，有互斥锁也有共享锁。

互斥锁：同一时间点，只会有一个线程持有者当前互斥锁。

共享锁：同一时间点，当前共享锁可以被多个线程同时持有。

二、深入synchronized

2.1 类锁、对象锁

synchronized的使用一般就是同步方法和同步代码块。

synchronized的锁是基于对象实现的。

如果使用同步方法

- static：此时使用的是当前类.class作为锁（类锁）
- 非static：此时使用的是当前对象做为锁（对象锁）

```
public class MiTest {

    public static void main(String[] args) {
        // 锁的是，当前Test.class
        Test.a();

        Test test = new Test();
        // 锁的是new出来的test对象
        test.b();
    }

}

class Test{
    public static synchronized void a(){
        System.out.println("1111");
    }

    public synchronized void b(){
        System.out.println("2222");
    }
}
```

2.2 synchronized的优化

在JDK1.5的时候，Doug Lee推出了ReentrantLock，lock的性能远高于synchronized，所以JDK团队就在JDK1.6中，对synchronized做了大量的优化。

锁消除：在synchronized修饰的代码中，如果不存在操作临界资源的情况，会触发锁消除，你即便写了synchronized，他也不会触发。

```
public synchronized void method(){
    // 没有操作临界资源
    // 此时这个方法的synchronized你可以认为木有~~
}
```

锁膨胀：如果在一个循环中，频繁的获取和释放做资源，这样带来的消耗很大，锁膨胀就是将锁的范围扩大，避免频繁的竞争和获取锁资源带来不必要的消耗。

```
public void method(){
    for(int i = 0;i < 999999;i++){
        synchronized(对象){

        }
```

```
    }
    // 这是上面的代码会触发锁膨胀
    synchronized(对象){
        for(int i = 0;i < 999999;i++){

        }
    }
}
```

锁升级：ReentrantLock的实现，是先基于乐观锁的CAS尝试获取锁资源，如果拿不到锁资源，才会挂起线程。synchronized在JDK1.6之前，完全就是获取不到锁，立即挂起当前线程，所以synchronized性能比较差。

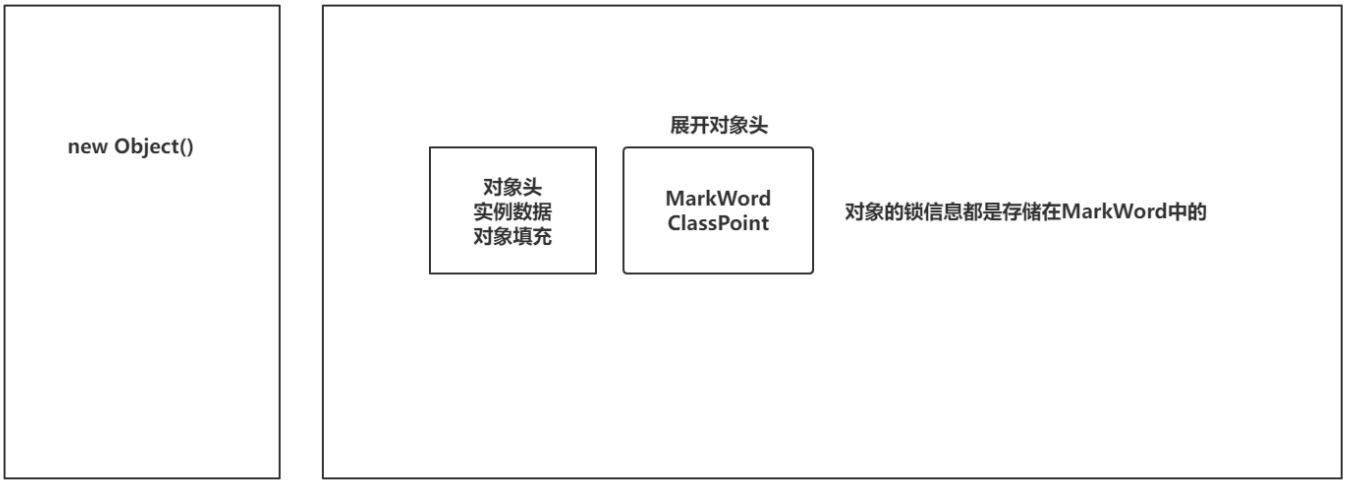
synchronized就在JDK1.6做了锁升级的优化

- **无锁、匿名偏向**：当前对象没有作为锁存在。
- **偏向锁**：如果当前锁资源，只有一个线程在频繁的获取和释放，那么这个线程过来，只需要判断，当前指向的线程是否是当前线程 。
  - 如果是，直接拿着锁资源走。
  - 如果当前线程不是我，基于CAS的方式，尝试将偏向锁指向当前线程。如果获取不到，触发锁升级，升级为轻量级锁。（偏向锁状态出现了锁竞争的情况）
- **轻量级锁**：会采用自旋锁的方式去频繁的以CAS的形式获取锁资源（采用的是**自适应自旋锁**）
  - 如果成功获取到，拿着锁资源走
  - 如果自旋了一定次数，没拿到锁资源，锁升级。
- **重量级锁**：就是最传统的synchronized方式，拿不到锁资源，就挂起当前线程。（用户态&内核态）

2.3 synchronized实现原理

synchronized是基于对象实现的。

先要对Java中对象在堆内存的存储有一个了解。



展开MarkWord

HotSpot的实现

锁状态	25bit	31bit	1bit	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态 (new)	unused	hashcode (如果有调用)	unused	分代年龄	0	0	1

锁状态	54bit	2bit	1bit	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0	1

锁状态	62bit	2bit 锁标志位	
轻量级锁（自旋）	指向线程栈中 Lock Record 的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

MarkWord中标记着四种锁的信息：无锁、偏向锁、轻量级锁、

2.4 synchronized的锁升级

为了可以在Java中看到对象头的MarkWord信息，需要导入依赖

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.9</version>
</dependency>
```

锁默认情况下，开启了偏向锁延迟。

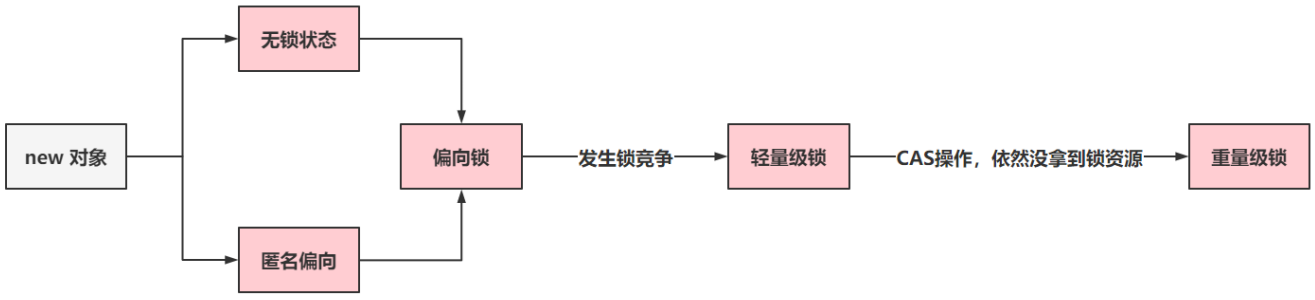
> 偏向锁在升级为轻量级锁时，会涉及到偏向锁撤销，需要等到一个安全点（STW），才可以做偏向锁撤销，在明知道有并发情况，就可以选择不开启偏向锁，或者是设置偏向锁延迟开启
>
> 因为JVM在启动时，需要加载大量的.class文件到内存中，这个操作会涉及到synchronized的使用，为了避免出现偏向锁撤销操作，JVM启动初期，有一个延迟4s开启偏向锁的操作
>
> 如果正常开启偏向锁了，那么不会出现无锁状态，对象会直接变为匿名偏向

```
public static void main(String[] args) throws InterruptedException {
    Thread.sleep(5000);
    Object o = new Object();
    System.out.println(ClassLayout.parseInstance(o).toPrintable());

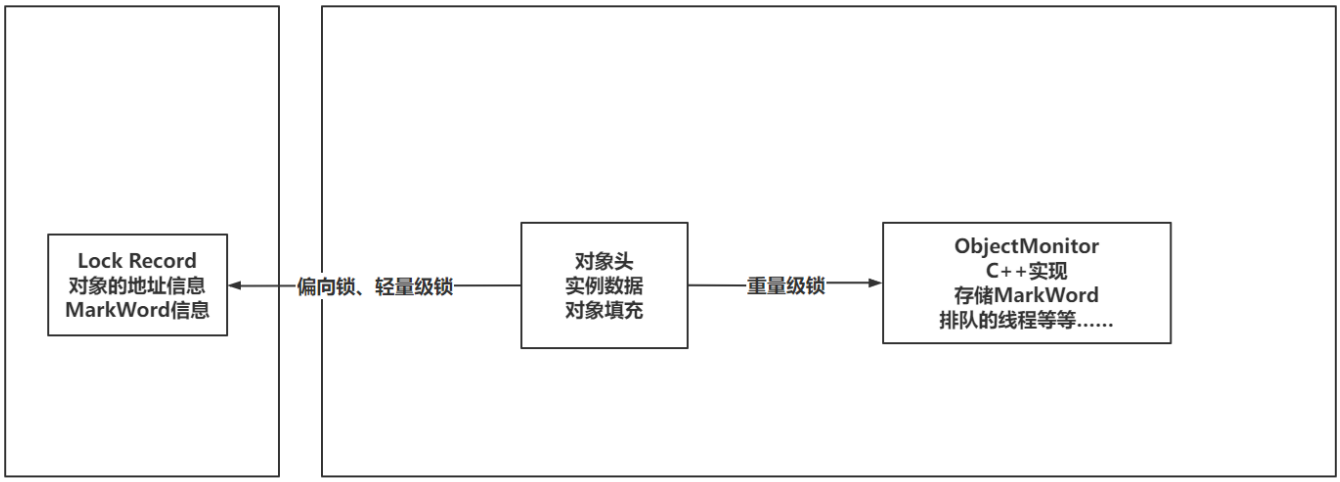
    new Thread(() -> {

        synchronized (o){
            //t1  - 偏向锁
            System.out.println("t1:" + ClassLayout.parseInstance(o).toPrintable());
        }
    }).start();
    //main - 偏向锁 - 轻量级锁CAS - 重量级锁
    synchronized (o){
        System.out.println("main:" + ClassLayout.parseInstance(o).toPrintable());
    }
}
```

整个锁升级状态的转变：



Lock Record以及ObjectMonitor存储的内容



2.5 重量锁底层ObjectMonitor

需要去找到openjdk，在百度中直接搜索openjdk，第一个链接就是

找到ObjectMonitor的两个文件，hpp，cpp

先查看核心属性：http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/69087d08d473/src/share/vm/runtime/objectMonitor.hpp

```
ObjectMonitor() {
    _header       = NULL;  // header存储着MarkWord
    _count        = 0;     // 竞争锁的线程个数
    _waiters      = 0,     // wait的线程个数
    _recursions   = 0;     // 标识当前synchronized锁重入的次数
    _object       = NULL;
    _owner        = NULL;  // 持有锁的线程
    _WaitSet      = NULL;  // 保存wait的线程信息，双向链表
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ; // 获取锁资源失败后，线程要放到当前的单向链表中
    FreeNext      = NULL ;
    _EntryList    = NULL ; // _cxq以及被唤醒的WaitSet中的线程，在一定机制下，会放到EntryList中
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
  }
```

适当的查看几个C++中实现的加锁流程

http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/69087d08d473/src/share/vm/runtime/objectMonitor.cpp

TryLock

```
int ObjectMonitor::TryLock (Thread * Self) {
   for (;;) {
         // 拿到持有锁的线程
      void * own = _owner ;
      // 如果有线程持有锁，告辞
      if (own != NULL) return 0 ;
      // 说明没有线程持有锁，own是null，cmpxchg指令就是底层的CAS实现。
      if (Atomic::cmpxchg_ptr (Self, &_owner, NULL) == NULL) {
               // 成功获取锁资源
         return 1 ;
      }
      // 这里其实重试操作没什么意义，直接返回-1
      if (true) return -1 ;
   }
}
```

try_entry

```
bool ObjectMonitor::try_enter(Thread* THREAD) {
  // 在判断_owner是不是当前线程
  if (THREAD != _owner) {
    // 判断当前持有锁的线程是否是当前线程，说明轻量级锁刚刚升级过来的情况
    if (THREAD->is_lock_owned ((address)_owner)) {
      _owner = THREAD ;
      _recursions = 1 ;
      OwnerIsThread = 1 ;
      return true;
    }
    // CAS操作，尝试获取锁资源
    if (Atomic::cmpxchg_ptr (THREAD, &_owner, NULL) != NULL) {
      // 没拿到锁资源，告辞
      return false;
    }
    // 拿到锁资源
    return true;
  } else {
    // 将_recursions + 1，代表锁重入操作。
    _recursions++;
    return true;
  }
}
```

enter（想方设法拿到锁资源，如果没拿到，挂起扔到_cxq单向链表中）

```
void ATTR ObjectMonitor::enter(TRAPS) {
  // 拿到当前线程
  Thread * const Self = THREAD ;
  void * cur ;
  // CAS走你，
  cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;
  if (cur == NULL) {
     // 拿锁成功
     return ;
  }
  // 锁重入操作
  if (cur == Self) {
     // TODO-FIXME: check for integer overflow!  BUGID 6557169.
     _recursions ++ ;
     return ;
  }
  //轻量级锁过来的。
  if (Self->is_lock_owned ((address)cur)) {
    _recursions = 1 ;
    _owner = Self ;
    OwnerIsThread = 1 ;
    return ;
  }


  // 走到这了，没拿到锁资源，count++
  Atomic::inc_ptr(&_count);


    for (;;) {
      jt->set_suspend_equivalent();
      // 入队操作，进到cxq中
      EnterI (THREAD) ;
      if (!ExitSuspendEquivalent(jt)) break ;
      _recursions = 0 ;
      _succ = NULL ;
      exit (false, Self) ;
      jt->java_suspend_self();
    }
  }
  // count--
  Atomic::dec_ptr(&_count);
```

```
}
```

EnterI

```
for (;;) {
    // 入队
    node._next = nxt = _cxq ;
    // CAS的方式入队。
    if (Atomic::cmpxchg_ptr (&node, &_cxq, nxt) == nxt) break ;

    // 重新尝试获取锁资源
    if (TryLock (Self) > 0) {
        assert (_succ != Self         , "invariant") ;
        assert (_owner == Self        , "invariant") ;
        assert (_Responsible != Self  , "invariant") ;
        return ;
    }
}
```

三、深入ReentrantLock

3.1 ReentrantLock和synchronized的区别

废话区别：单词不一样。。。

核心区别：

- ReentrantLock是个类，synchronized是关键字，当然都是在JVM层面实现互斥锁的方式

效率区别：

- 如果竞争比较激烈，推荐ReentrantLock去实现，不存在锁升级概念。而synchronized是存在锁升级概念的，如果升级到重量级锁，是不存在锁降级的。

底层实现区别：

- 实现原理是不一样，ReentrantLock基于AQS实现的，synchronized是基于ObjectMonitor

功能向的区别：

- ReentrantLock的功能比synchronized更全面。
  - ReentrantLock支持公平锁和非公平锁
  - ReentrantLock可以指定等待锁资源的时间。

选择哪个：如果你对并发编程特别熟练，推荐使用ReentrantLock，功能更丰富。如果掌握的一般般，使用synchronized会更好

3.2 AQS概述

AQS就是AbstractQueuedSynchronizer抽象类，AQS其实就是JUC包下的一个基类，JUC下的很多内容都是基于AQS实现了部分功能，比如ReentrantLock，ThreadPoolExecutor，阻塞队列，CountDownLatch，Semaphore，CyclicBarrier等等都是基于AQS实现。

首先AQS中提供了一个由volatile修饰，并且采用CAS方式修改的int类型的state变量。

其次AQS中维护了一个双向链表，有head，有tail，并且每个节点都是Node对象

```
static final class Node {
        static final Node SHARED = new Node();
        static final Node EXCLUSIVE = null;

        static final int CANCELLED =  1;
        static final int SIGNAL    = -1;
        static final int CONDITION = -2;

        static final int PROPAGATE = -3;

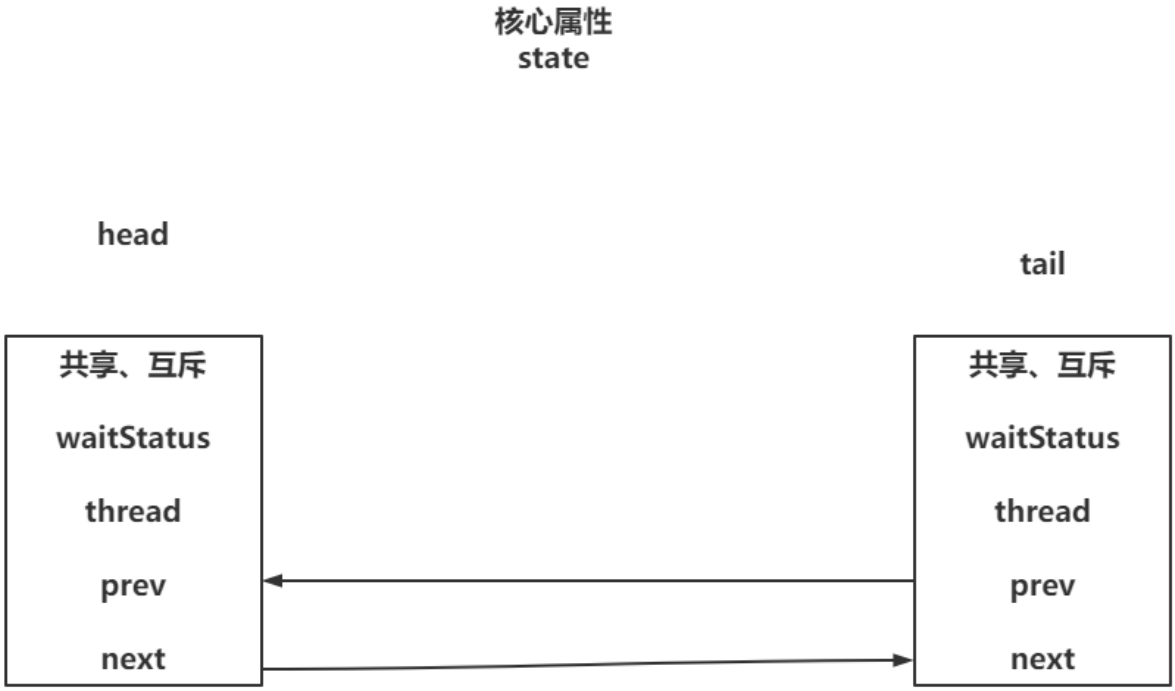

        volatile int waitStatus;


        volatile Node prev;


        volatile Node next;


        volatile Thread thread;
}
```

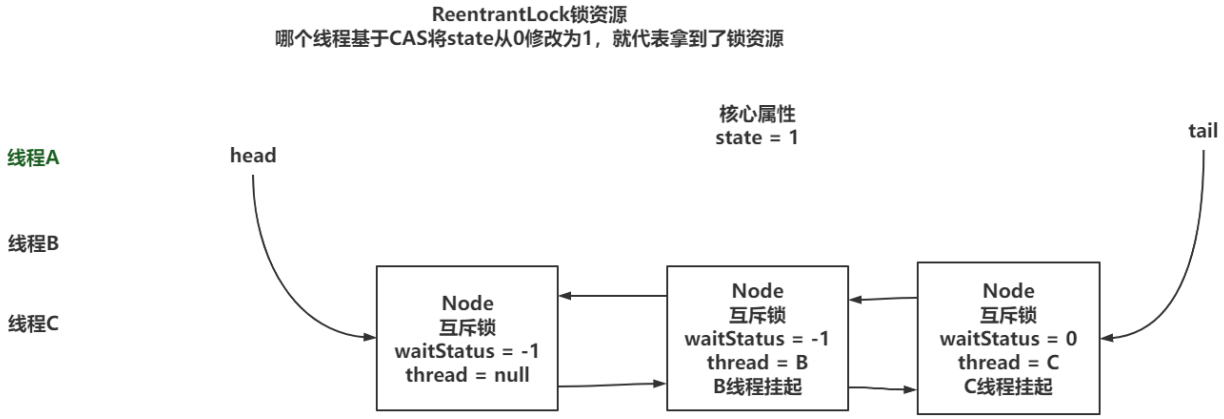
AQS内部结构和属性



3.3 加锁流程源码剖析

3.3.1 加锁流程概述

这个是非公平锁的流程



1. 线程A先执行CAS，将state从0修改为1，线程A就获取到了锁资源，去执行业务代码即可
2. 线程B再执行CAS，发现state已经是1了，无法获取到锁资源
3. 线程B需要去排队，将自己封装为Node对象
4. 需要将当前B线程的Node放到双向队列保存，排队
   4.1 但是双向链表中，必须先有一个伪节点作为头结点，并且放到双向队列中
   4.2 将B线程的Node挂在tail的后面，并且将上一个节点的状态修改为-1，再挂起B线程

3.3.2 三种加锁源码分析

3.3.2.1 lock方法

1. 执行lock方法后，公平锁和非公平锁的执行套路不一样

```
// 非公平锁
final void lock() {
    // 上来就先基于CAS的方式，尝试将state从0改为1
    if (compareAndSetState(0, 1))
        // 获取锁资源成功，会将当前线程设置到exclusiveOwnerThread属性，代表是当前线程持有着锁资源
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // 执行acquire，尝试获取锁资源
        acquire(1);
}

// 公平锁
final void lock() {
    //  执行acquire，尝试获取锁资源
    acquire(1);
}
```

2. acquire方法，是公平锁和非公平锁的逻辑一样

```
public final void acquire(int arg) {
    // tryAcquire：再次查看，当前线程是否可以尝试获取锁资源
    if (!tryAcquire(arg) &&
        // 没有拿到锁资源
        // addWaiter(Node.EXCLUSIVE)：将当前线程封装为Node节点，插入到AQS的双向链表的结尾
        // acquireQueued：查看我是否是第一个排队的节点，如果是可以再次尝试获取锁资源，如果长时间拿不到
        // 如果不是第一个排队的额节点，就尝试挂起线程即可
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        // 中断线程的操作
        selfInterrupt();
}
```

3. tryAcquire方法竞争锁最资源的逻辑，分为公平锁和非公平锁

```
// 非公平锁实现
final boolean nonfairTryAcquire(int acquires) {
    // 获取当前线程
    final Thread current = Thread.currentThread();
    // 获取了state熟属性
    int c = getState();
    // 判断state当前是否为0,之前持有锁的线程释放了锁资源
    if (c == 0) {
        // 再次抢一波锁资源
        if (compareAndSetState(0, acquires)) {
```

```
            setExclusiveOwnerThread(current);
            // 拿锁成功返回true
            return true;
        }
    }
    // 不是0，有线程持有着锁资源，如果是，证明是锁重入操作
    else if (current == getExclusiveOwnerThread()) {
        // 将state + 1
        int nextc = c + acquires;
        if (nextc < 0) // 说明对重入次数+1后，超过了int正数的取值范围
            // 01111111 11111111 11111111 11111111
            // 10000000 00000000 00000000 00000000
            // 说明重入的次数超过界限了。
            throw new Error("Maximum lock count exceeded");
        // 正常的将计算结果，复制给state
        setState(nextc);
        // 锁重入成功
        return true;
    }
    // 返回false
    return false;
}


// 公平锁实现
protected final boolean tryAcquire(int acquires) {
    // 获取当前线程
    final Thread current = Thread.currentThread();
    // ....
    int c = getState();
    if (c == 0) {
        // 查看AQS中是否有排队的Node
        // 没人排队抢一手 。有人排队，如果我是第一个，也抢一手
        if (!hasQueuedPredecessors() &&
            // 抢一手~
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 锁重入~~~
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
```

```
    }
    return false;
}

// 查看是否有线程在AQS的双向队列中排队
// 返回false，代表没人排队
public final boolean hasQueuedPredecessors() {
    // 头尾节点
    Node t = tail;
    Node h = head;
    // s为头结点的next节点
    Node s;
    // 如果头尾节点相等，证明没有线程排队，直接去抢占锁资源
    return h != t &&
        // s节点不为null，并且s节点的线程为当前线程（排在第一名的是不是我）
        (s == null || s.thread != Thread.currentThread());
}
```

4. addWaite方法，将没有拿到锁资源的线程扔到AQS队列中去排队

```
// 没有拿到锁资源，过来排队，  mode：代表互斥锁
private Node addWaiter(Node mode) {
    // 将当前线程封装为Node，
    Node node = new Node(Thread.currentThread(), mode);
    // 拿到尾结点
    Node pred = tail;
    // 如果尾结点不为null
    if (pred != null) {
        // 当前节点的prev指向尾结点
        node.prev = pred;
        // 以CAS的方式，将当前线程设置为tail节点
        if (compareAndSetTail(pred, node)) {
            // 将之前的尾结点的next指向当前节点
            pred.next = node;
            return node;
        }
    }
    // 如果CAS失败，以死循环的方式，保证当前线程的Node一定可以放到AQS队列的末尾
    enq(node);
    return node;
}

private Node enq(final Node node) {
    for (;;) {
        // 拿到尾结点
        Node t = tail;
        // 如果尾结点为空，AQS中一个节点都没有，构建一个伪节点，作为head和tail
```

```
        if (t == null) {
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            // 比较熟悉了，以CAS的方式，在AQS中有节点后，插入到AQS队列的末尾
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

5. acquireQueued方法，判断当前线程是否还能再次尝试获取锁资源，如果不能再次获取锁资源，或者又没获取到，尝试将当前线程挂起

```
// 当前没有拿到锁资源后，并且到AQS排队了之后触发的方法。 中断操作这里不用考虑
final boolean acquireQueued(final Node node, int arg) {
    // 不考虑中断
    // failed：获取锁资源是否失败（这里简单掌握落地，真正触发的，还是tryLock和lockInterruptibly）
    boolean failed = true;
    try {
        boolean interrupted = false;
        // 死循环.....
        for (;;) {
            // 拿到当前节点的前继节点
            final Node p = node.predecessor();
            // 前继节点是否是head，如果是head，再次执行tryAcquire尝试获取锁资源。
            if (p == head && tryAcquire(arg)) {
                // 获取锁资源成功
                // 设置头结点为当前获取锁资源成功Node，并且取消thread信息
                setHead(node);
                // help GC
                p.next = null;
                // 获取锁失败标识为false
                failed = false;
                return interrupted;
            }
            // 没拿到锁资源.....
            // shouldParkAfterFailedAcquire：基于上一个节点转改来判断当前节点是否能够挂起线程，如果可以返
            // 如果不能，就返回false，继续下次循环
            if (shouldParkAfterFailedAcquire(p, node) &&
                // 这里基于Unsafe类的park方法，将当前线程挂起
                parkAndCheckInterrupt())
                interrupted = true;
        }
```

```
    } finally {
        if (failed)
            // 在lock方法中，基本不会执行。
            cancelAcquire(node);
    }
}
// 获取锁资源成功后，先执行setHead
private void setHead(Node node) {
    // 当前节点作为头结点  伪
    head = node;
    // 头结点不需要线程信息
    node.thread = null;
    node.prev = null;
}

// 当前Node没有拿到锁资源，或者没有资格竞争锁资源，看一下能否挂起当前线程
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    // -1，SIGNAL状态：代表当前节点的后继节点，可以挂起线程，后续我会唤醒我的后继节点
    // 1，CANCELLED状态：代表当前节点以及取消了
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        // 上一个节点为-1之后，当前节点才可以安心的挂起线程
        return true;
    if (ws > 0) {
        // 如果当前节点的上一个节点是取消状态，我需要往前找到一个状态不为1的Node，作为他的next节点
        // 找到状态不为1的节点后，设置一下next和prev
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 上一个节点的状态不是1或者-1，那就代表节点状态正常，将上一个节点的状态改为-1
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

3.3.2.2 tryLock方法

- tryLock();

```
// tryLock方法，无论公平锁还有非公平锁。都会走非公平锁抢占锁资源的操作
// 就是拿到state的值， 如果是0，直接CAS浅尝一下
// state 不是0，那就看下是不是锁重入操作
// 如果没抢到，或者不是锁重入操作，告辞，返回false
public boolean tryLock() {
```

```
    // 非公平锁的竞争锁操作
    return sync.nonfairTryAcquire(1);
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

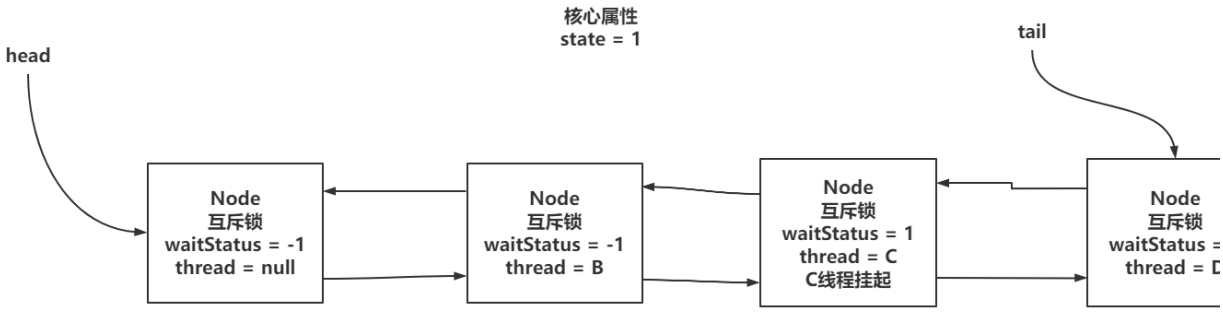
- tryLock(time,unit);
  - 第一波分析，类似的代码：

```
// tryLock(time,unit)执行的方法
public final boolean tryAcquireNanos(int arg, long nanosTimeout)throws InterruptedException {
    // 线程的中断标记位，是不是从false，别改为了true，如果是，直接抛异常
    if (Thread.interrupted())
        throw new InterruptedException();
    // tryAcquire分为公平和非公平锁两种执行方式，如果拿锁成功， 直接告辞，
    return tryAcquire(arg) ||
        // 如果拿锁失败，在这要等待指定时间
        doAcquireNanos(arg, nanosTimeout);
}

private boolean doAcquireNanos(int arg, long nanosTimeout)
        throws InterruptedException {
    // 如果等待时间是0秒，直接告辞，拿锁失败
    if (nanosTimeout <= 0L)
        return false;
    // 设置结束时间。
    final long deadline = System.nanoTime() + nanosTimeout;
    // 先扔到AQS队列
    final Node node = addWaiter(Node.EXCLUSIVE);
    // 拿锁失败，默认true
    boolean failed = true;
```

```
    try {
        for (;;) {
            // 如果在AQS中，当前node是head的next，直接抢锁
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return true;
            }
            // 结算剩余的可用时间
            nanosTimeout = deadline - System.nanoTime();
            // 判断是否是否用尽的位置
            if (nanosTimeout <= 0L)
                return false;
            // shouldParkAfterFailedAcquire：根据上一个节点来确定现在是否可以挂起线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                // 避免剩余时间太少，如果剩余时间少就不用挂起线程
                nanosTimeout > spinForTimeoutThreshold)
                // 如果剩余时间足够，将线程挂起剩余时间
                LockSupport.parkNanos(this, nanosTimeout);
            // 如果线程醒了，查看是中断唤醒的，还是时间到了唤醒的。
            if (Thread.interrupted())
                // 是中断唤醒的！
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

- 取消节点分析：



取消节点整体操作流程：
1、线程设置为null
2、往前找到有效节点作为当前节点的prev
3、将waitStatus设置为1，代表取消
4、脱离整个AQS队列：
4.1：当前Node是tail
4.2：当前节点是head的后继节点
4.3：不是tail节点，也不是hea的后继节点

```
// 取消在AQS中排队的Node
private void cancelAcquire(Node node) {
    // 如果当前节点为null，直接忽略。
    if (node == null)
        return;
    //1. 线程设置为null
    node.thread = null;

    //2. 往前跳过被取消的节点，找到一个有效节点
    Node pred = node.prev;
    while (pred.waitStatus > 0)
        node.prev = pred = pred.prev;

    //3. 拿到了上一个节点之前的next
    Node predNext = pred.next;

    //4. 当前节点状态设置为1，代表节点取消
    node.waitStatus = Node.CANCELLED;

    // 脱离AQS队列的操作
    // 当前Node是尾结点，将tail从当前节点替换为上一个节点
    if (node == tail && compareAndSetTail(node, pred)) {
        compareAndSetNext(pred, predNext, null);
    } else {
        // 到这，上面的操作CAS操作失败
        int ws = pred.waitStatus;
        // 不是head的后继节点
        if (pred != head &&
            // 拿到上一个节点的状态，只要上一个节点的状态不是取消状态，就改为-1
            (ws == Node.SIGNAL || (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL)))
            && pred.thread != null) {
            // 上面的判断都是为了避免后面节点无法被唤醒。
            // 前继节点是有效节点，可以唤醒后面的节点
            Node next = node.next;
            if (next != null && next.waitStatus <= 0)
                compareAndSetNext(pred, predNext, next);
        } else {
            // 当前节点是head的后继节点
            unparkSuccessor(node);
        }

        node.next = node; // help GC
    }
}
```

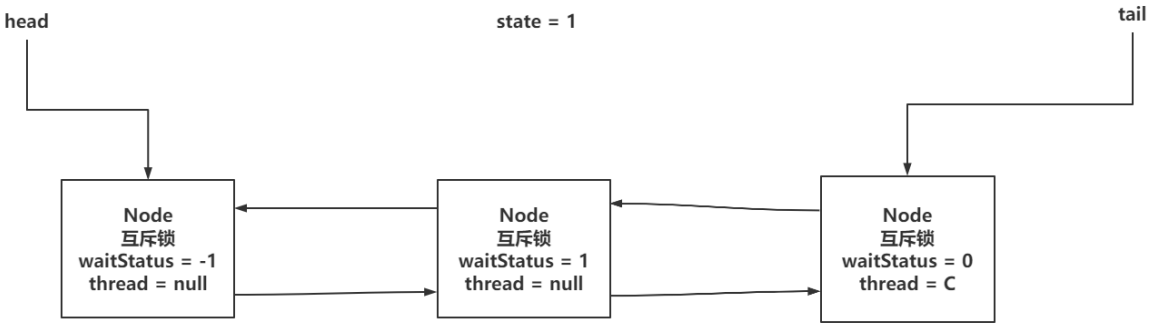
3.3.2.3 lockInterruptibly方法

```
// 这个是lockInterruptibly和tryLock(time,unit)唯一的区别
// lockInterruptibly，拿不到锁资源，就死等，等到锁资源释放后，被唤醒，或者是被中断唤醒
private void doAcquireInterruptibly(int arg) throws InterruptedException {
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return;
            }
            if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                // 中断唤醒抛异常！
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}

private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    // 这个方法可以确认，当前挂起的线程，是被中断唤醒的，还是被正常唤醒的。
    // 中断唤醒，返回true，如果是正常唤醒，返回false
    return Thread.interrupted();
}
```

3.4 释放锁流程源码剖析

3.4.1 释放锁流程概述



线程A持有当前锁，重入了一次，state = 2
线程B和线程C获取锁资源失败，在AQS中排队

线程A释放锁资源调用unlock方法，就是执行了tryRelease方法
首先判断是不是线程A持有着锁资源，如果不是就抛异常。
如果是线程A持有者锁资源，对state - 1.
- 1成功后，会判断state是否为0。如果不是，方法结束。
如果为0，证明当前锁资源释放干净。
查看头结点的状态是否不为0（判断是否为-1）。如果为0，代表后面没挂起的线程
如果不为0，后续链表中有挂起的线程，需要唤醒。
在唤醒线程时，需要先将当前的-1，改为0，找到有效节点唤醒
找到之后，唤醒线程即可

3.4.2 释放锁源码分析

```
public void unlock() {
    // 释放锁资源不分为公平锁和非公平锁，都是一个sync对象
    sync.release(1);
}

// 释放锁的核心流程
public final boolean release(int arg) {
    // 核心释放锁资源的操作之一
    if (tryRelease(arg)) {
        // 如果锁已经释放掉了，走这个逻辑
        Node h = head;
        // h不为null，说明有排队的（录课时估计脑袋蒙圈圈。）
        // 如果h的状态不为0（为-1），说明后面有排队的Node，并且线程已经挂起了。
        if (h != null && h.waitStatus != 0)
            // 唤醒排队的线程
            unparkSuccessor(h);
        return true;
    }
    return false;
}
// ReentrantLock释放锁资源操作
protected final boolean tryRelease(int releases) {
    // 拿到state - 1（并没有赋值给state）
    int c = getState() - releases;
    // 判断当前持有锁的线程是否是当前线程，如果不是，直接抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
```

```
    // free，代表当前锁资源是否释放干净了。
    boolean free = false;
    if (c == 0) {
        // 如果state - 1后的值为0，代表释放干净了。
        free = true;
        // 将持有锁的线程置位null
        setExclusiveOwnerThread(null);
    }
    // 将c设置给state
    setState(c);
    // 锁资源释放干净返回true，否则返回false
    return free;
}

// 唤醒后面排队的Node
private void unparkSuccessor(Node node) {
    // 拿到头节点状态
    int ws = node.waitStatus;
    if (ws < 0)
        // 先基于CAS，将节点状态从-1，改为0
        compareAndSetWaitStatus(node, ws, 0);
    // 拿到头节点的后续节点。
    Node s = node.next;
    // 如果后续节点为null或者，后续节点的状态为1，代表节点取消了。
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 如果后续节点为null，或者后续节点状态为取消状态，从后往前找到一个有效节点环境
        for (Node t = tail; t != null && t != node; t = t.prev)
            // 从后往前找到状态小于等于0的节点
            // 找到离head最新的有效节点，并赋值给s
            if (t.waitStatus <= 0)
                s = t;
    }
    // 只要找到了这个需要被唤醒的节点，执行unpark唤醒
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

3.5 AQS常见的问题

3.5.1 AQS中为什么要有一个虚拟的head节点

因为AQS提供了ReentrantLock的基本实现，而在ReentrantLock释放锁资源时，需要去考虑是否需要执行unparkSuccessor方法，去唤醒后继节点。

因为Node中存在waitStatus的状态，默认情况下状态为0，如果当前节点的后继节点线程挂起了，那么就将当前节点的状态设置为-1。这个-1状态的出现是为了避免重复唤醒或者释放资源的问题。

因为AQS中排队的Node中的线程如果挂起了，是无法自动唤醒的。需要释放锁或者释放资源后，再被释放的线程去唤醒挂起的线程。 因为唤醒节点需要从整个AQS双向链表中找到离head最近的有效节点去唤醒。而这个找离head最近的Node可能需要遍历整个双向链表。如果AQS中，没有挂起的线程，代表不需要去遍历AQS双向链表去找离head最近的有效节点。

为了避免出现不必要的循环链表操作，提供了一个-1的状态。如果只有一个Node进入到AQS中排队，所以发现如果是第一个Node进来，他必须先初始化一个虚拟的head节点作为头，来监控后继节点中是否有挂起的线程。

3.5.2 AQS中为什么选择使用双向链表，而不是单向链表

首先AQS中一般是存放没有获取到资源的Node，而在竞争锁资源时，ReentrantLock提供了一个方法，lockInterruptibly方法，也就是线程在竞争锁资源的排队途中，允许中断。中断后会执行cancelAcquire方法，从而将当前节点状态置位1，并且从AQS队列中移除掉。如果采用单向链表，当前节点只能按到后继或者前继节点，这样是无法将前继节点指向后继节点的，需要遍历整个AQS从头或者从尾去找。单向链表在移除AQS中排队的Node时，成本很高。

当前在唤醒后继节点时，如果是单向链表也会出问题，因为节点插入方式的问题，导致只能单向的去找有效节点去唤醒，从而造成很多次无效的遍历操作，如果是双向链表就可以解决这个问题。

3.6 ConditionObject

3.6.1 ConditionObject的介绍&应用

像synchronized提供了wait和notify的方法实现线程在持有锁时，可以实现挂起，已经唤醒的操作。

ReentrantLock也拥有这个功能。

ReentrantLock提供了await和signal方法去实现类似wait和notify的功能。

想执行await或者是signal就必须先持有lock锁的资源。

先look一下Condition的应用

```
public static void main(String[] args) throws InterruptedException, IOException {
    ReentrantLock lock = new ReentrantLock();
    Condition condition = lock.newCondition();

    new Thread(() -> {
```

```
        lock.lock();
        System.out.println("子线程获取锁资源并await挂起线程");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            condition.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("子线程挂起后被唤醒！持有锁资源");

    }).start();
    Thread.sleep(100);
    // ==================main======================
    lock.lock();
    System.out.println("主线程等待5s拿到锁资源，子线程执行了await方法");
    condition.signal();
    System.out.println("主线程唤醒了await挂起的子线程");
    lock.unlock();
}
```

3.6.2 Condition的构建方式&核心属性

发现在通过lock锁对象执行newCondition方法时，本质就是直接new的AQS提供的ConditionObject对象

```
final ConditionObject newCondition() {
    return new ConditionObject();
}
```

其实lock锁中可以有多个Condition对象。

在对Condition1进行操作时，不会影响到Condition2的单向链表。

其次可以发现ConditionObject中，只有两个核心属性：

```
/** First node of condition queue. */
private transient Node firstWaiter;
/** Last node of condition queue. */
private transient Node lastWaiter;
```

虽然Node对象有prev和next，但是在ConditionObject中是不会使用这两个属性的，只要在Condition队列中，这两个属性都是null。在ConditionObject中只会使用nextWaiter的属性实现单向链表的效果。

3.6.3 Condition的await方法分析（前置分析）

持有锁的线程在执行await方法后会做几个操作：

- 判断线程是否中断，如果中断了，什么都不做。
- 没有中断，就讲当前线程封装为Node添加到Condition的单向链表中
- 一次性释放掉锁资源。
- 如果当前线程没有在AQS队列，就正常执行LockSupport.park(this)挂起线程。

```
// await方法的前置分析，只分析到线程挂起
public final void await() throws InterruptedException {
    // 先判断线程的中断标记位是否是true
    if (Thread.interrupted())
        // 如果是true，就没必要执行后续操作挂起了。
        throw new InterruptedException();
    // 在线程挂起之前，先将当前线程封装为Node，并且添加到Condition队列中
    Node node = addConditionWaiter();
    // fullyRelease在释放锁资源，一次性将锁资源全部释放，并且保留重入的次数
    int savedState = fullyRelease(node);
    // 省略一行代码……
    // 当前Node是否在AQS队列中？
    // 执行fullyRelease方法后，线程就释放锁资源了，如果线程刚刚释放锁资源，其他线程就立即执行了signal方法，
    // 此时当前线程就被放到了AQS的队列中，这样一来线程就不需要执行LockSupport.park(this);去挂起线程了
    while (!isOnSyncQueue(node)) {
        // 如果没有在AQS队列中，正常在Condition单向链表里，正常挂起线程。
        LockSupport.park(this);
        // 省略部分代码……
    }
    // 省略部分代码……
}

// 线程挂起先，添加到Condition单向链表的业务~~
private Node addConditionWaiter() {
    // 拿到尾节点。
    Node t = lastWaiter;
    // 如果尾节点有值，并且尾节点的状态不正常，不是-2，尾节点可能要拜拜了~
    if (t != null && t.waitStatus != Node.CONDITION) {
        // 如果尾节点已经取消了，需要干掉取消的尾节点~
        unlinkCancelledWaiters();
        // 重新获取lastWaiter
```

```
        t = lastWaiter;
    }
    // 构建当前线程的Node，并且状态设置为-2
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    // 如果last节点为null。直接将当前节点设置为firstWaiter
    if (t == null)
        firstWaiter = node;
    else
        // 如果last节点不为null，说明有值，就排在lastWaiter的后面
        t.nextWaiter = node;
    // 把当前节点设置为最后一个节点
    lastWaiter = node;
    // 返回当前节点
    return node;
}

// 干掉取消的尾节点。
private void unlinkCancelledWaiters() {
    // 拿到头节点
    Node t = firstWaiter;
    // 声明一个节点，爱啥啥~~~
    Node trail = null;
    // 如果t不为null，就正常执行~~
    while (t != null) {
        // 拿到t的next节点
        Node next = t.nextWaiter;
        // 如果t的状态不为-2，说明有问题
        if (t.waitStatus != Node.CONDITION) {
            // t节点的next为null
            t.nextWaiter = null;
            // 如果trail为null，代表头结点状态就是1，
            if (trail == null)
                // 将头结点指向next节点
                firstWaiter = next;
            else
                // 如果trail有值，说明不是头结点位置
                trail.nextWaiter = next;
            // 如果next为null，说明单向链表遍历到最后了，直接结束
            if (next == null)
                lastWaiter = trail;
        }
        // 如果t的状态是-2，一切正常
        else {
            // 临时存储t
            trail = t;
        }
        // t指向之前的next
```

```
            t = next;
        }
    }

    // 一次性释放锁资源
    final int fullyRelease(Node node) {
        // 标记位，释放锁资源默认失败！
        boolean failed = true;
        try {
            // 拿到现在state的值
            int savedState = getState();
            // 一次性释放干净全部锁资源
            if (release(savedState)) {
                // 释放锁资源失败了么？ 没有！
                failed = false;
                // 返回对应的锁资源信息
                return savedState;
            } else {
                throw new IllegalMonitorStateException();
            }
        } finally {
            if (failed)
                // 如果释放锁资源失败，将节点状态设置为取消
                node.waitStatus = Node.CANCELLED;
        }
    }
```

3.6.4 Condition的signal方法分析

分为了几个部分：

- 确保执行signal方法的是持有锁的线程
- 脱离Condition的队列
- 将Node状态从-2改为0
- 将Node添加到AQS队列
- 为了避免当前Node无法在AQS队列正常唤醒做了一些判断和操作

```
    // 线程挂起后，可以基于signal唤醒~
    public final void signal() {
        // 在ReentrantLock中，如果执行signal的线程没有持有锁资源，直接扔异常
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        // 拿到排在Condition首位的Node
        Node first = firstWaiter;
```

```
    // 有Node在排队，才需要唤醒，如果没有，直接告辞~~
    if (first != null)
        doSignal(first);
}

// 开始唤醒Condition中的Node中的线程
private void doSignal(Node first) {
    // 先一波do-while走你~~~
    do {
        // 获取到第二个节点，并且将第二个节点设置为firstWaiter
        if ( (firstWaiter = first.nextWaiter) == null)
            // 说明就一个节点在Condition队列中，那么直接将firstWaiter和lastWaiter置位null
            lastWaiter = null;
        // 如果还有nextWaiter节点，因为当前节点要被唤醒了，脱离整个Condition队列。将nextWaiter置位null
        first.nextWaiter = null;
        // 如果transferForSignal返回true，一切正常，退出while循环
    } while (!transferForSignal(first) &&
            // 如果后续节点还有，往后面继续唤醒，如果没有，退出while循环
             (first = firstWaiter) != null);
}

// 准备开始唤醒在Condition中排队的Node
final boolean transferForSignal(Node node) {
    // 将在Condition队列中的Node的状态从-2，改为0，代表要扔到AQS队列了。
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        // 如果失败了，说明在signal之前应当是线程被中断了，从而被唤醒了。
        return false;
    // 如果正常的将Node的状态从-2改为0，这是就要将Condition中的这个Node扔到AQS的队列。
    // 将当前Node扔到AQS队列，返回的p是当前Node的prev
    Node p = enq(node);
    // 获取上一个Node的状态
    int ws = p.waitStatus;
    // 如果ws > 0 ，说明这个Node已经被取消了。
    // 如果ws状态不是取消，将prev节点的状态改为-1,。
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        // 如果prev节点已经取消了，可能会导致当前节点永远无法被唤醒。立即唤醒当前节点，基于acquireQueued方
            // 让当前节点找到一个正常的prev节点，并挂起线程
        // 如果prev节点正常，但是CAS修改prev节点失败了。证明prev节点因为并发原因导致状态改变。还是为了避免
            // 节点无法被正常唤醒，提前唤醒当前线程，基于acquireQueued方法，让当前节点找到一个正常的prev节点
        LockSupport.unpark(node.thread);
    // 返回true
    return true;
}
```

3.6.5 Conditiond的await方法分析（后置分析）

分为了几个部分：

- 唤醒之后，要先确认是中断唤醒还是signal唤醒，还是signal唤醒后被中断
- 确保当前线程的Node已经在AQS队列中
- 执行acquireQueued方法，等待锁资源。
- 在获取锁资源后，要确认是否在获取锁资源的阶段被中断过，如果被中断过，并且不是THROW_IE，那就确保interruptMode是REINTERRUPT。
- 确认当前Node已经不在Condition队列中了
- 最终根据interruptMode来决定具体做的事情
  - 0：嘛也不做。
  - THROW_IE：抛出异常
  - REINTERRUPT：执行线程的interrupt方法

```
// 现在分析await方法的后半部分
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    // 中断模式~
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        // 如果线程执行到这，说明现在被唤醒了。
        // 线程可以被signal唤醒。（如果是signal唤醒，可以确认线程已经在AQS队列中）
        // 线程可以被interrupt唤醒，线程被唤醒后，没有在AQS队列中。
        // 如果线程先被signal唤醒，然后线程中断了。。。。（做一些额外处理）
        // checkInterruptWhileWaiting可以确认当前中如何唤醒的。
        // 返回的值，有三种
        // 0：正常signal唤醒，没别的事（不知道Node是否在AQS队列）
        // THROW_IE（-1）：中断唤醒，并且可以确保在AQS队列
        // REINTERRUPT（1）：signal唤醒，但是线程被中断了，并且可以确保在AQS队列
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // Node一定在AQS队列
    // 执行acquireQueued，尝试在ReentrantLock中获取锁资源。
    // acquireQueued方法返回true：代表线程在AQS队列中挂起时，被中断过
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        // 如果线程在AQS队列排队时，被中断了，并且不是THROW_IE状态，确保线程的interruptMode是REINTERRU
        // REINTERRUPT：await不是中断唤醒，但是后续被中断过！！！
        interruptMode = REINTERRUPT;
    // 如果当前Node还在condition的单向链表中，脱离Condition的单向链表
```

```
    if (node.nextWaiter != null)
        unlinkCancelledWaiters();
    // 如果interruptMode是0，说明线程在signal后以及持有锁的过程中，没被中断过，什么事都不做！
    if (interruptMode != 0)
        // 如果不是0~
        reportInterruptAfterWait(interruptMode);
}
// 判断当前线程被唤醒的模式，确认interruptMode的值。
private int checkInterruptWhileWaiting(Node node) {
    // 判断线程是否中断了。
    return Thread.interrupted() ?
        // THROW_IE：代表线程是被interrupt唤醒的，需要向上排除异常
        // REINTERRUPT：代表线程是signal唤醒的，但是在唤醒之后，被中断了。

        (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
        // 线程是正常的被signal唤醒，并且线程没有中断过。
        0;
}

// 判断线程到底是中断唤醒的，还是signal唤醒的！
final boolean transferAfterCancelledWait(Node node) {
    // 基于CAS将Node的状态从-2改为0
    if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
        // 说明是中断唤醒的线程。因为CAS成功了。
        // 将Node添加到AQS队列中~（如果是中断唤醒的，当前线程同时存在Condition的单向链表以及AQS的队列中
        enq(node);
        // 返回true
        return true;
    }
    // 判断当前的Node是否在AQS队列（signal唤醒的，但是可能线程还没放到AQS队列）
    // 等到signal方法将线程的Node扔到AQS队列后，再做后续操作
    while (!isOnSyncQueue(node))
        // 如果没在AQS队列上，那就线程让步，稍等一会，Node放到AQS队列再处理（看CPU）
        Thread.yield();
    // signal唤醒的，返回false
    return false;
}

// 确认Node是否在AQS队列上
final boolean isOnSyncQueue(Node node) {
    // 如果线程状态为-2，肯定没在AQS队列
    // 如果prev节点的值为null，肯定没在AQS队列
    if (node.waitStatus == Node.CONDITION || node.prev == null)
        // 返回false
        return false;
    // 如果节点的next不为null。说明已经在AQS队列上。、
    if (node.next != null)
```

```
            // 确定AQS队列上有！
            return true;
        // 如果上述判断都没有确认节点在AQS队列上，在AQS队列中寻找一波
        return findNodeFromTail(node);
    }
    // 在AQS队列中找当前节点
    private boolean findNodeFromTail(Node node) {
        // 拿到尾节点
        Node t = tail;
        for (;;) {
            // tail是否是当前节点，如果是，说明在AQS队列
            if (t == node)
                // 可以跳出while循环
                return true;
            // 如果节点为null，AQS队列中没有当前节点
            if (t == null)
                // 进入while，让步一手
                return false;
            // t向前引用
            t = t.prev;
        }
    }

    private void reportInterruptAfterWait(int interruptMode) throws InterruptedException {
        // 如果是中断唤醒的await，直接抛出异常！
        if (interruptMode == THROW_IE)
            throw new InterruptedException();
        // 如果是REINTERRUPT，signal后被中断过
        else if (interruptMode == REINTERRUPT)
            // 确认线程的中断标记位是true
            // Thread.currentThread().interrupt();
            selfInterrupt();
    }
```

3.6.6 Condition的awaitNanos&signalAll方法分析

awaitNanos：仅仅是在await方法的基础上，做了一内内的改变，整体的逻辑思想都是一样的。

挂起线程时，传入要阻塞的时间，时间到了，自动唤醒，走添加到AQS队列的逻辑

```
    // await指定时间，多了个时间到了自动醒。
    public final long awaitNanos(long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        Node node = addConditionWaiter();
```

```
    int savedState = fullyRelease(node);
    // deadline：当前线程最多挂起到什么时间点
    final long deadline = System.nanoTime() + nanosTimeout;
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        // nanosTimeout的时间小于等于0，直接告辞！！
        if (nanosTimeout <= 0L) {
            // 正常扔到AQS队列
            transferAfterCancelledWait(node);
            break;
        }
        // nanosTimeout的时间大于1000纳秒时，才可以挂起线程
        if (nanosTimeout >= spinForTimeoutThreshold)
            // 如果大于，正常挂起
            LockSupport.parkNanos(this, nanosTimeout);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
        // 计算剩余的挂起时间，可能需要重新的走while循环，再次挂起线程
        nanosTimeout = deadline - System.nanoTime();
    }
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null)
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
    // 剩余的挂起时间
    return deadline - System.nanoTime();
}
```

signalAll方法。这个方法一看就懂，之前signal是唤醒1个，这个是全部唤醒

```
// 以do-while的形式，将Condition单向链表中的所有Node，全部唤醒并扔到AQS队列
private void doSignalAll(Node first) {
    // 将头尾都置位null~
    lastWaiter = firstWaiter = null;
    do {
        // 拿到next节点的引用
        Node next = first.nextWaiter;
        // 断开当前Node的nextWaiter
        first.nextWaiter = null;
        // 修改Node状态，扔AQS队列，是否唤醒！
        transferForSignal(first);
        // 指向下一个节点
        first = next;
```

```
    } while (first != null);
}
```

四、深入ReentrantReadWriteLock

一、为什么要出现读写锁

synchronized和ReentrantLock都是互斥锁。

如果说有一个操作是读多写少的，还要保证线程安全的话。如果采用上述的两种互斥锁，效率方面很定是很低的。

在这种情况下，咱们就可以使用ReentrantReadWriteLock读写锁去实现。

读读之间是不互斥的，可以读和读操作并发执行。

但是如果涉及到了写操作，那么还得是互斥的操作。

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

static ReentrantReadWriteLock.WriteLock writeLock = lock.writeLock();

static ReentrantReadWriteLock.ReadLock readLock = lock.readLock();

public static void main(String[] args) throws InterruptedException {
    new Thread(() -> {
        readLock.lock();
        try {
            System.out.println("子线程！");
            try {
                Thread.sleep(500000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
            readLock.unlock();
        }
    }).start();

    Thread.sleep(1000);
    writeLock.lock();
    try {
        System.out.println("主线程！");
    } finally {
        writeLock.unlock();
```

```
    }
}
```

二、读写锁的实现原理

ReentrantReadWriteLock还是基于AQS实现的，还是对state进行操作，拿到锁资源就去干活，如果没有拿到，依然去AQS队列中排队。

读锁操作：基于state的高16位进行操作。

写锁操作：基于state的低16为进行操作。

ReentrantReadWriteLock依然是可重入锁。

写锁重入：读写锁中的写锁的重入方式，基本和ReentrantLock一致，没有什么区别，依然是对state进行+1操作即可，只要确认持有锁资源的线程，是当前写锁线程即可。只不过之前ReentrantLock的重入次数是state的正数取值范围，但是读写锁中写锁范围就变小了。

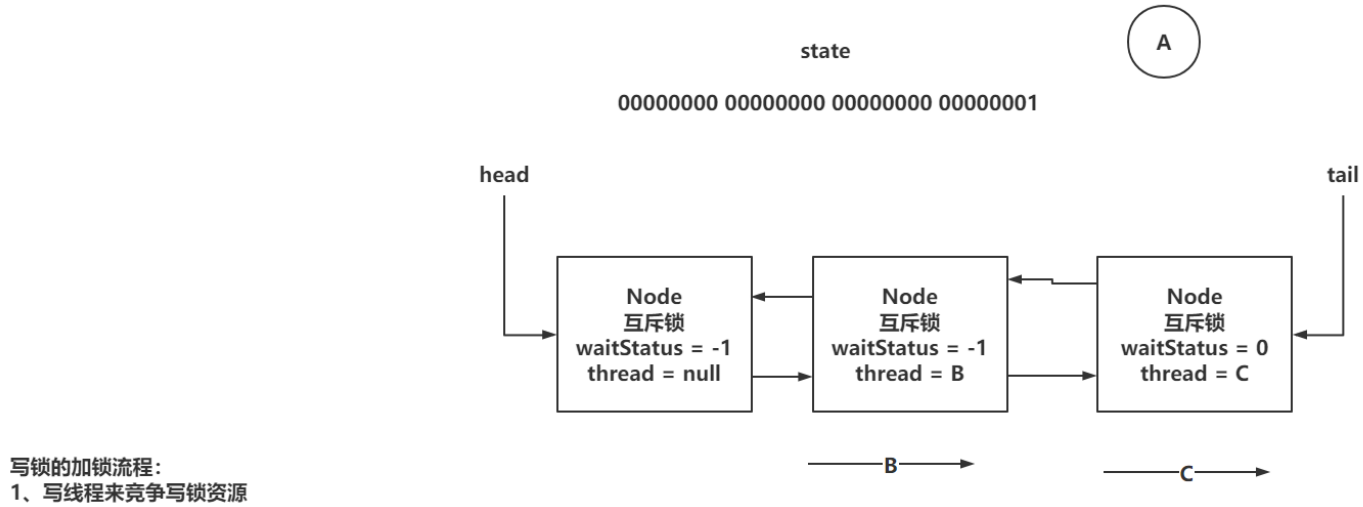
读锁重入：因为读锁是共享锁。读锁在获取锁资源操作时，是要对state的高16位进行 + 1操作。因为读锁是共享锁，所以同一时间会有多个读线程持有读锁资源。这样一来，多个读操作在持有读锁时，无法确认每个线程读锁重入的次数。为了去记录读锁重入的次数，每个读操作的线程，都会有一个ThreadLocal记录锁重入的次数。

写锁的饥饿问题：读锁是共享锁，当有线程持有读锁资源时，再来一个线程想要获取读锁，直接对state修改即可。在读锁资源先被占用后，来了一个写锁资源，此时，大量的需要获取读锁的线程来请求锁资源，如果可以绕过写锁，直接拿资源，会造成写锁长时间无法获取到写锁资源。

读锁在拿到锁资源后，如果再有读线程需要获取读锁资源，需要去AQS队列排队。如果队列的前面需要写锁资源的线程，那么后续读线程是无法拿到锁资源的。持有读锁的线程，只会让写锁线程之前的读线程拿到锁资源

三、写锁分析

3.1 写锁加锁流程概述



写锁的加锁流程：
1、写线程来竞争写锁资源
2、写线程会直接通过tryAcquire获取写锁资源（公平锁&非公平锁）
3、获取state值，并且拿到低16位的值。
4、如果state值为不为0，判断是否是锁重入操作，判断当前持有写锁的线程是否是当前线程。
5、如果state值为0：
5.1、判断是否是公平锁：查看队列是否有排队的，有就直接告辞，没有就抢一手
5.2、判断是否是非公平锁： 直接抢一手
6、如果拿到锁资源，直接告辞，如果没有拿到去排队，而排队的逻辑和ReentrantLock一样

3.2 写锁加锁源码分析

写锁加锁流程

```
// 写锁加锁的入口
public void lock() {
    sync.acquire(1);
}

// 阿巴阿巴！！
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}

// 读写锁的写锁实现tryAcquire
protected final boolean tryAcquire(int acquires) {
    // 拿到当前线程
    Thread current = Thread.currentThread();
    // 拿到state的值
    int c = getState();
    // 得到state低16位的值
    int w = exclusiveCount(c);
    // 判断是否有线程持有着锁资源
    if (c != 0) {
        // 当前没有线程持有写锁，读写互斥，告辞。
```

```
        // 有线程持有写锁，持有写锁的线程不是当前线程，不是锁重入，告辞。
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        // 当前线程持有写锁。 锁重入。
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // 没有超过锁重入的次数，正常 + 1
        setState(c + acquires);
        return true;
    }
    // 尝试获取锁资源
    if (writerShouldBlock() ||
        // CAS拿锁
        !compareAndSetState(c, c + acquires))
        return false;
    // 拿锁成功，设置占有互斥锁的线程
    setExclusiveOwnerThread(current);
    // 返回true
    return true;
}

// ================================================================
// 这个方法是将state的低16位的值拿到
int w = exclusiveCount(c);
state & ((1 << 16) - 1)
00000000 00000000 00000000 00000001    ==   1
00000000 00000001 00000000 00000000    ==   1 << 16
00000000 00000000 11111111 11111111    ==   (1 << 16) - 1
&运算，一个为0，必然为0，都为1，才为1
// ================================================================
// writerShouldBlock方法查看公平锁和非公平锁的效果
// 非公平锁直接返回false执行CAS尝试获取锁资源
// 公平锁需要查看是否有排队的，如果有排队的，我是否是head的next
```

3.3 写锁释放锁流程概述&释放锁源码

释放的流程和ReentrantLock一致，只是在判断释放是否干净时，判断低16位的值

```
// 写锁释放锁的tryRelease方法
protected final boolean tryRelease(int releases) {
    // 判断当前持有写锁的线程是否是当前线程
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    // 获取state - 1
    int nextc = getState() - releases;
    // 判断低16位结果是否为0，如果为0，free设置为true
```

```
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        // 将持有锁的线程设置为null
        setExclusiveOwnerThread(null);
    // 设置给state
    setState(nextc);
    // 释放干净，返回true。 写锁有冲入，这里需要返回false，不去释放排队的Node
    return free;
}
```

四、读锁分析

4.1 读锁加锁流程概述

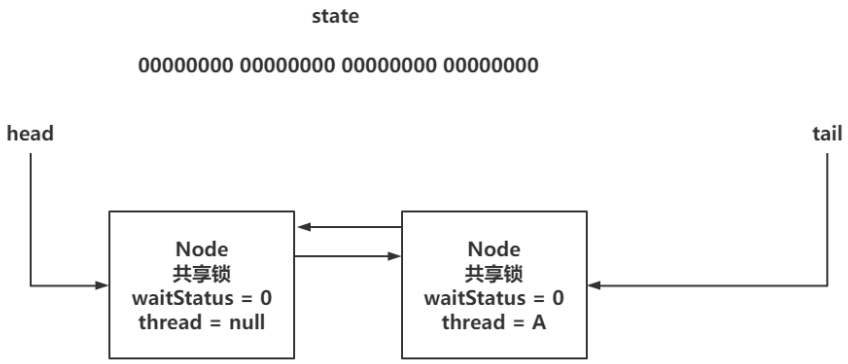
1、分析读锁加速的基本流程

2、分析读锁的可重入锁实现以及优化

3、解决ThreadLocal内存泄漏问题

4、读锁获取锁自后，如果唤醒AQS中排队的读线程

4.1.1 基础读锁流程



读锁加锁流程：
1、读操作线程，竞争读锁资源
2、会竞争共享锁的资源
3、拿到state
4、判断state中的低16位是否为0
如果不为0，代表有写锁占用着资源
如果有资源占用着写锁，但是不是当前线程，告辞。
（写锁 - 读锁的降级）
5、拿到state高16位的值。
公平锁：如果有人排队，直接拜拜，排队去。
非公平锁：查看AQS的队列中，是否有写线程在排队，如果有，排队去。
6、 CAS对state的高16位 + 1
如果成功，拿到读锁资源。
后续操作.....

针对上述简单逻辑的源码分析

```
// 读锁加锁的方法入口
public final void acquireShared(int arg) {
    // 竞争锁资源滴干活
    if (tryAcquireShared(arg) < 0)
        // 没拿到锁资源，去排队
        doAcquireShared(arg);
}

// 读锁竞争锁资源的操作
protected final int tryAcquireShared(int unused) {
    // 拿到当前线程
    Thread current = Thread.currentThread();
    // 拿到state
    int c = getState();
    // 拿到state的低16位，判断 != 0，有写锁占用着锁资源
    // 并且，当前占用锁资源的线程不是当前线程
    if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current)
        // 写锁被其他线程占用，无法获取读锁，直接返回 -1，去排队
        return -1;
    // 没有线程持有写锁、当前线程持有写锁
    // 获取读锁的信息，state的高16位。
    int r = sharedCount(c);
    // 公平锁：就查看队列是由有排队的，有排队的，直接告辞，进不去if，后面也不用判断（没人排队继续走）
    // 非公平锁：没有排队的，直接抢。 有排队的，但是读锁其实不需要排队，如果出现这个情况，大部分是写锁资源
    // 后续Node还没有来记得拿到读锁资源，当前竞争的读线程，可以直接获取
    if (!readerShouldBlock() &&
        // 判断持有读锁的临界值是否达到
        r < MAX_COUNT &&
        // CAS修改state，对高16位进行 + 1
        compareAndSetState(c, c + SHARED_UNIT)) {
        // 省略部分代码！！！！
        return 1;
    }
    return fullTryAcquireShared(current);
}
// 非公平锁的判断
final boolean apparentlyFirstQueuedIsExclusive() {
    Node h, s;
    return (h = head) != null &&    // head为null，可以直接抢占锁资源
        (s = h.next)  != null &&    // head的next为null，可以直接抢占锁资源
        !s.isShared()         &&    // 如果排在head后面的Node，是共享锁，可以直接抢占锁资源。
        s.thread != null;           // 后面排队的thread为null，可以直接抢占锁资源
}
```

4.1.2 读锁重入流程

==========重入操作==========

前面阐述过，读锁为了记录锁重入的次数，需要让每个读线程用ThreadLocal存储重入次数

ReentrantReadWriteLock对读锁重入做了一些优化操作

==========记录重入次数的核心==========

ReentrantReadWriteLock在内部对ThreadLocal做了封装，基于HoldCount的对象存储重入次数，在内部有个count属性记录，而且每个线程都是自己的ThreadLocalHoldCounter，所以可以直接对内部的count进行++操作。

==========第一个获取读锁资源的重入次数记录方式==========

第一个拿到读锁资源的线程，不需要通过ThreadLocal存储，内部提供了两个属性来记录第一个拿到读锁资源线程的信息

内部提供了firstReader记录第一个拿到读锁资源的线程，firstReaderHoldCount记录firstReader的锁重入次数

==========最后一个获取读锁资源的重入次数记录方式==========

最后一个拿到读锁资源的线程，也会缓存他的重入次数，这样++起来更方便

基于cachedHoldCounter缓存最后一个拿到锁资源现成的重入次数

==========最后一个获取读锁资源的重入次数记录方式==========

重入次数的流程执行方式：

1、判断当前线程是否是第一个拿到读锁资源的：如果是，直接将firstReader以及firstReaderHoldCount设置为当前线程的信息

2、判断当前线程是否是firstReader：如果是，直接对firstReaderHoldCount++即可。

3、跟firstReader没关系了，先获取cachedHoldCounter，判断是否是当前线程。

3.1、如果不是，获取当前线程的重入次数，将cachedHoldCounter设置为当前线程。

3.2、如果是，判断当前重入次数是否为0，重新设置当前线程的锁从入信息到readHolds（ThreadLocal）中，算是初始化操作，重入次数是0

> 3.3、前面两者最后都做count++

上述逻辑源码分析

```
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    int r = sharedCount(c);
    if (!readerShouldBlock() &&
        r < MAX_COUNT &&
        compareAndSetState(c, c + SHARED_UNIT)) {
        // ==================================================================
        // 判断r == 0，当前是第一个拿到读锁资源的线程
        if (r == 0) {
            // 将firstReader设置为当前线程
            firstReader = current;
            // 将count设置为1
            firstReaderHoldCount = 1;
        }
        // 判断当前线程是否是第一个获取读锁资源的线程
        else if (firstReader == current) {
            // 直接++。
            firstReaderHoldCount++;
        }
        // 到这，就说明不是第一个获取读锁资源的线程
        else {
            // 那获取最后一个拿到读锁资源的线程
            HoldCounter rh = cachedHoldCounter;
            // 判断当前线程是否是最后一个拿到读锁资源的线程
            if (rh == null || rh.tid != getThreadId(current))
                // 如果不是，设置当前线程为cachedHoldCounter
                cachedHoldCounter = rh = readHolds.get();
            // 当前线程是之前的cacheHoldCounter
            else if (rh.count == 0)
                // 将当前的重入信息设置到ThreadLocal中
                readHolds.set(rh);
            // 重入的++
            rh.count++;
        }
        // ==================================================================
        return 1;
    }
```

```
    return fullTryAcquireShared(current);
}
```

4.1.3 读锁加锁的后续逻辑fullTryAcquireShared

```
// tryAcquireShard方法中，如果没有拿到锁资源，走这个方法，尝试再次获取，逻辑跟上面基本一致。
final int fullTryAcquireShared(Thread current) {
    // 声明当前线程的锁重入次数
    HoldCounter rh = null;
    // 死循环
    for (;;) {
        // 再次拿到state
        int c = getState();
        // 当前如果有写锁在占用锁资源，并且不是当前线程，返回-1，走排队策略
        if (exclusiveCount(c) != 0) {
            if (getExclusiveOwnerThread() != current)
                return -1;

        }
        // 查看当前是否可以尝试竞争锁资源（公平锁和非公平锁的逻辑）
        else if (readerShouldBlock()) {
            // 无论公平还是非公平，只要进来，就代表要放到AQS队列中了，先做一波准备
            // 在处理ThreadLocal的内存泄漏问题
            if (firstReader == current) {
                // 如果当前当前线程是之前的firstReader，什么都不用做
            } else {
                // 第一次进来是null。
                if (rh == null) {
                    // 拿到最后一个获取读锁的线程
                    rh = cachedHoldCounter;
                    // 当前线程并不是cachedHoldCounter，没到拿到
                    if (rh == null || rh.tid != getThreadId(current)) {
                        // 从自己的ThreadLocal中拿到重入计数器
                        rh = readHolds.get();
                        // 如果计数器为0，说明之前没拿到过读锁资源
                        if (rh.count == 0)
                            // remove，避免内存泄漏
                            readHolds.remove();
                    }
                }
                // 前面处理完之后，直接返回-1
                if (rh.count == 0)
                    return -1;
            }
        }
        // 判断重入次数，是否超出阈值
```

```
            if (sharedCount(c) == MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
            // CAS尝试获取锁资源
            if (compareAndSetState(c, c + SHARED_UNIT)) {
                if (sharedCount(c) == 0) {
                    firstReader = current;
                    firstReaderHoldCount = 1;
                } else if (firstReader == current) {
                    firstReaderHoldCount++;
                } else {
                    if (rh == null)
                        rh = cachedHoldCounter;
                    if (rh == null || rh.tid != getThreadId(current))
                        rh = readHolds.get();
                    else if (rh.count == 0)
                        readHolds.set(rh);
                    rh.count++;
                    cachedHoldCounter = rh; // cache for release
                }
                return 1;
            }
        }
    }
```

4.1.4 读线程在AQS队列获取锁资源的后续操作

> 1、正常如果都是读线程来获取读锁资源，不需要使用到AQS队列的，直接CAS操作即可
>
> 2、如果写线程持有着写锁，这是读线程就需要进入到AQS队列排队，可能会有多个读线程在AQS中。
>
> 当写锁释放资源后，会唤醒head后面的读线程，当head后面的读线程拿到锁资源后，还需要查看next节点是否也是读线程在阻塞，如果是，直接唤醒

源码分析

```
// 读锁需要排队的操作
private void doAcquireShared(int arg) {
    // 声明Node，类型是共享锁，并且扔到AQS中排队
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            // 拿到上一个节点
            final Node p = node.predecessor();
```

```
            // 如果prev节点是head，直接可以执行tryAcquireShared
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    // 拿到读锁资源后，需要做的后续处理
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
            // 找到prev有效节点，将状态设置为-1，挂起当前线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}

private void setHeadAndPropagate(Node node, int propagate) {
    // 拿到head节点
    Node h = head;
    // 将当前节点设置为head节点
    setHead(node);
    // 第一个判断更多的是在信号量有处理JDK1.5 BUG的操作。
    if (propagate > 0 || h == null || h.waitStatus < 0 || (h = head) == null || h.waitStatus < 0) {
        // 拿到当前Node的next节点
        Node s = node.next;
        // 如果next节点是共享锁，直接唤醒next节点
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

4.2 读锁的释放锁流程

1、处理重入以及state的值

2、唤醒后续排队的Node

源码分析

```
// 读锁释放锁流程
public final boolean releaseShared(int arg) {
    // tryReleaseShared：处理state的值，以及可重入的内容
    if (tryReleaseShared(arg)) {
        // AQS队列的事！
        doReleaseShared();
        return true;
    }
    return false;
}

// 1、 处理重入问题  2、 处理state
protected final boolean tryReleaseShared(int unused) {
    // 拿到当前线程
    Thread current = Thread.currentThread();
    // 如果是firstReader，直接干活，不需要ThreadLocal
    if (firstReader == current) {
        // assert firstReaderHoldCount > 0;
        if (firstReaderHoldCount == 1)
            firstReader = null;
        else
            firstReaderHoldCount--;
    }
    // 不是firstReader，从cachedHoldCounter以及ThreadLocal处理
    else {
        // 如果是cachedHoldCounter，正常--
        HoldCounter rh = cachedHoldCounter;
        // 如果不是cachedHoldCounter，从自己的ThreadLocal中拿
        if (rh == null || rh.tid != getThreadId(current))
            rh = readHolds.get();
        int count = rh.count;
        // 如果为1或者更小，当前线程就释放干净了，直接remove，避免value内存泄漏
        if (count <= 1) {
            readHolds.remove();
            // 如果已经是0，没必要再unlock，扔个异常
            if (count <= 0)
                throw unmatchedUnlockException();
        }
        // -- 走你。
        --rh.count;
    }
    for (;;) {
        // 拿到state，高16位，-1，成功后，返回state是否为0
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))
```

```
            return nextc == 0;
        }
    }

    // 唤醒AQS中排队的线程
    private void doReleaseShared() {
        // 死循环
        for (;;) {
            // 拿到头
            Node h = head;
            // 说明有排队的
            if (h != null && h != tail) {
                // 拿到head的状态
                int ws = h.waitStatus;
                // 判断是否为 -1
                if (ws == Node.SIGNAL) {
                    // 到这，说明后面有挂起的线程，先基于CAS将head的状态从-1，改为0
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                        continue;
                    // 唤醒后续节点
                    unparkSuccessor(h);
                }
                // 这里不是给读写锁准备的，在信号量里说。。。
                else if (ws == 0 && !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                    continue;
            }
            // 这里是出口
            if (h == head)
                break;
        }
    }
```

五、死锁问题

在咱们的操作系统2022版本有，已经有最新的死锁课程了，这里就不做过多讲解

查看这个课程：

https://www.mashibing.com/course/1368

四、阻塞队列

一、基础概念

1.1 生产者消费者概念

生产者消费者是设计模式的一种。让生产者和消费者基于一个容器来解决强耦合问题。

生产者 消费者彼此之间不会直接通讯的，而是通过一个容器（队列）进行通讯。

所以生产者生产完数据后扔到容器中，不通用等待消费者来处理。

消费者不需要去找生产者要数据，直接从容器中获取即可。

而这种容器最常用的结构就是队列。

1.2 JUC阻塞队列的存取方法

常用的存取方法都是来自于JUC包下的BlockingQueue

生产者存储方法

```
add(E)          // 添加数据到队列，如果队列满了，无法存储，抛出异常
offer(E)    // 添加数据到队列，如果队列满了，返回false
offer(E,timeout,unit)   // 添加数据到队列，如果队列满了，阻塞timeout时间，如果阻塞一段时间，依然没添加进入，
put(E)      // 添加数据到队列，如果队列满了，挂起线程，等到队列中有位置，再扔数据进去，死等！
```

消费者取数据方法

```
remove()    // 从队列中移除数据，如果队列为空，抛出异常
poll()      // 从队列中移除数据，如果队列为空，返回null，么的数据
poll(timeout,unit)   // 从队列中移除数据，如果队列为空，挂起线程timeout时间，等生产者扔数据，再获取
take()     // 从队列中移除数据，如果队列为空，线程挂起，一直等到生产者扔数据，再获取
```

二、ArrayBlockingQueue

2.1 ArrayBlockingQueue的基本使用

ArrayBlockingQueue在初始化的时候，必须指定当前队列的长度。

因为**ArrayBlockingQueue**是基于数组实现的队列结构，数组长度不可变，必须提前设置数组长度信息。

```
public static void main(String[] args) throws ExecutionException, InterruptedException, IOException {
    // 必须设置队列的长度
    ArrayBlockingQueue queue = new ArrayBlockingQueue(4);

    // 生产者扔数据
    queue.add("1");
    queue.offer("2");
    queue.offer("3",2,TimeUnit.SECONDS);
    queue.put("2");

    // 消费者取数据
    System.out.println(queue.remove());
    System.out.println(queue.poll());
    System.out.println(queue.poll(2,TimeUnit.SECONDS));
    System.out.println(queue.take());
}
```

2.2 生产者方法实现原理

生产者添加数据到队列的方法比较多，需要一个一个查看

2.2.1 ArrayBlockingQueue的常见属性

ArrayBlockingQueue中的成员变量

```
lock = 就是一个ReentrantLock
count = 就是当前数组中元素的个数
iterms = 就是数组本身
# 基于putIndex和takeIndex将数组结构实现为了队列结构
```

```
putIndex = 存储数据时的下标
takeIndex = 去数据时的下标
notEmpty = 消费者挂起线程和唤醒线程用到的Condition（看成sync的wait和notify）
notFull = 生产者挂起线程和唤醒线程用到的Condition（看成sync的wait和notify）
```

2.2.2 add方法实现

add方法本身就是调用了offer方法，如果offer方法返回false，直接抛出异常

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        // 抛出的异常
        throw new IllegalStateException("Queue full");
}
```

2.2.3 offer方法实现

```
public boolean offer(E e) {
    // 要求存储的数据不允许为null，为null就抛出空指针
    checkNotNull(e);
    // 当前阻塞队列的lock锁
    final ReentrantLock lock = this.lock;
    // 为了保证线程安全，加锁
    lock.lock();
    try {
        // 如果队列中的元素已经存满了，
        if (count == items.length)
            // 返回false
            return false;
        else {
            // 队列没满，执行enqueue将元素添加到队列中
            enqueue(e);
            // 返回true
            return true;
        }
    } finally {
        // 操作完释放锁
        lock.unlock();
    }
}

//==========================================================
private void enqueue(E x) {
```

```
    // 拿到数组的引用
    final Object[] items = this.items;
    // 将元素放到指定位置
    items[putIndex] = x;
    // 对inputIndex进行++操作，并且判断是否已经等于数组长度，需要归位
    if (++putIndex == items.length)
        // 将索引设置为0
        putIndex = 0;
    // 元素添加成功，进行++操作。
    count++;
    // 将一个Condition中阻塞的线程唤醒。
    notEmpty.signal();
}
```

2.2.4 offer(time,unit)方法

生产者在添加数据时，如果队列已经满了，阻塞一会。

- 阻塞到消费者消费了消息，然后唤醒当前阻塞线程
- 阻塞到了time时间，再次判断是否可以添加，不能，直接告辞。

```
// 如果线程在挂起的时候，如果对当前阻塞线程的中断标记位进行设置，此时会抛出异常直接结束
public boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException {
        // 非空检验
    checkNotNull(e);
    // 将时间单位转换为纳秒
    long nanos = unit.toNanos(timeout);
    // 加锁
    final ReentrantLock lock = this.lock;
    // 允许线程中断并排除异常的加锁方式
    lock.lockInterruptibly();
    try {
        // 为什么是while（虚假唤醒）
        // 如果元素个数和数组长度一致，队列慢了
        while (count == items.length) {
            // 判断等待的时间是否还充裕
            if (nanos <= 0)
                // 不充裕，直接添加失败
                return false;
            // 挂起等待，会同时释放锁资源（对标sync的wait方法）
            // awaitNanos会挂起线程，并且返回剩余的阻塞时间
            // 恢复执行时，需要重新获取锁资源
            nanos = notFull.awaitNanos(nanos);
        }
        // 说明队列有空间了，enqueue将数据扔到阻塞队列中
```

```
            enqueue(e);
            return true;
        } finally {
            // 释放锁资源
            lock.unlock();
        }
    }
```

2.2.5 put方法

如果队列是满的， 就一直挂起，直到被唤醒，或者被中断

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            // await方法一直阻塞，直到被唤醒或者中断标记位
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

2.3 消费者方法实现原理

2.3.1 remove方法

```
// remove方法就是调用了poll
public E remove() {
    E x = poll();
    // 如果有数据，直接返回
    if (x != null)
        return x;
    // 没数据抛出异常
    else
        throw new NoSuchElementException();
}
```

2.4.2 poll方法

```
// 拉取数据
public E poll() {
    // 加锁操作
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 如果没有数据，直接返回null，如果有数据，执行dequeue，取出数据并返回
        return (count == 0) ? null : dequeue();
    } finally {
        lock.unlock();
    }
}

//============================================================
// 取出数据
private E dequeue() {
    // 将成员变量引用到局部变量
    final Object[] items = this.items;
    // 直接获取指定索引位置的数据
    E x = (E) items[takeIndex];
    // 将数组上指定索引位置设置为null
    items[takeIndex] = null;
    // 设置下次取数据时的索引位置
    if (++takeIndex == items.length)
        takeIndex = 0;
    // 对count进行--操作
    count--;
    // 迭代器内容，先跳过
    if (itrs != null)
        itrs.elementDequeued();
    // signal方法，会唤醒当前Condition中排队的一个Node。
    // signalAll方法，会将Condition中所有的Node，全都唤醒
    notFull.signal();
    // 返回数据。
    return x;
}
```

2.4.3 poll(time,unit)方法

```
public E poll(long timeout, TimeUnit unit) throws InterruptedException {
    // 转换时间单位
    long nanos = unit.toNanos(timeout);
    // 竞争锁
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
```

```
    try {
        // 如果没有数据
        while (count == 0) {
            if (nanos <= 0)
                // 没数据，也无法阻塞了，返回null
                return null;
            // 没数据，挂起消费者线程
            nanos = notEmpty.awaitNanos(nanos);
        }
        // 取数据
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

2.4.4 take方法

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        // 虚假唤醒
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

2.4.5 虚假唤醒

阻塞队列中，如果需要线程挂起操作，判断有无数据的位置采用的是while循环 ，为什么不能换成if

肯定是不能换成if逻辑判断

线程A，线程B，线程E，线程C。 其中ABE生产者，C属于消费者

假如线程的队列是满的

```
// E，拿到锁资源，还没有走while判断
while (count == items.length)
    // A醒了
    // B挂起
```

```
    notFull.await();
enqueue(e)；
```

C此时消费一条数据，执行notFull.signal()唤醒一个线程，A线程被唤醒

E走判断，发现有空余位置，可以添加数据到队列，E添加数据，走enqueue

如果判断是if，A在E释放锁资源后，拿到锁资源，直接走enqueue方法。

此时A线程就是在putIndex的位置，覆盖掉之前的数据，造成数据安全问题

三、LinkedBlockingQueue

3.1 LinkedBlockingQueue的底层实现

查看LinkedBlockingQueue是如何存储数据，并且实现链表结构的。

```
// Node对象就是存储数据的单位
static class Node<E> {
    // 存储的数据
    E item;
        // 指向下一个数据的指针
    Node<E> next;
        // 有参构造
    Node(E x) { item = x; }
}
```

查看LinkedBlockingQueue的有参构造

```
// 可以手动指定LinkedBlockingQueue的长度，如果没有指定，默认为Integer.MAX_VALUE
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    // 在初始化时，构建一个item为null的节点，作为head和last
        // 这种node可以成为哨兵Node，
    // 如果没有哨兵节点，那么在获取数据时，需要判断head是否为null，才能找next
    // 如果没有哨兵节点，那么在添加数据时，需要判断last是否为null，才能找next
    last = head = new Node<E>(null);
}
```

查看LinkedBlockingQueue的其他属性

```
// 因为是链表，没有想数组的length属性，基于AtomicInteger来记录长度
private final AtomicInteger count = new AtomicInteger();
```

```
// 链表的头，取
transient Node<E> head;
// 链表的尾，存
private transient Node<E> last;
// 消费者的锁
private final ReentrantLock takeLock = new ReentrantLock();
// 消费者的挂起操作，以及唤醒用的condition
private final Condition notEmpty = takeLock.newCondition();
// 生产者的锁
private final ReentrantLock putLock = new ReentrantLock();
// 生产者的挂起操作，以及唤醒用的condition
private final Condition notFull = putLock.newCondition();
```

3.2 生产者方法实现原理

3.2.1 add方法

你懂得，还是走offer方法

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}
```

3.2.2 offer方法

```
public boolean offer(E e) {
    // 非空校验
    if (e == null) throw new NullPointerException();
    // 拿到存储数据条数的count
    final AtomicInteger count = this.count;
    // 查看当前数据条数，是否等于队列限制长度，达到了这个长度，直接返回false
    if (count.get() == capacity)
        return false;
    // 声明c，作为标记存在
    int c = -1;
    // 将存储的数据封装为Node对象
    Node<E> node = new Node<E>(e);
    // 获取生产者的锁。
    final ReentrantLock putLock = this.putLock;
    // 竞争锁资源
    putLock.lock();
```

```
    try {
        // 再次做一个判断，查看是否还有空间
        if (count.get() < capacity) {
            // enqueue，扔数据
            enqueue(node);
            // 将数据个数 + 1
            c = count.getAndIncrement();
            // 拿到count的值 小于 长度限制
            // 有生产者在基于await挂起，这里添加完数据后，发现还有空间可以存储数据，
            // 唤醒前面可能已经挂起的生产者
            // 因为这里生产者和消费者不是互斥的，写操作进行的同时，可能也有消费者在消费数据。
            if (c + 1 < capacity)
                // 唤醒生产者
                notFull.signal();
        }
    } finally {
        // 释放锁资源
        putLock.unlock();
    }
    // 如果c == 0，代表添加数据之前，队列元素个数是0个。
    // 如果有消费者在队列没有数据的时候，来消费，此时消费者一定会挂起线程
    if (c == 0)
        // 唤醒消费者
        signalNotEmpty();
    // 添加成功返回true，失败返回-1
    return c >= 0;
}

//=================================================
private void enqueue(Node<E> node) {
    // 将当前Node设置为last的next，并且再将当前Node作为last
    last = last.next = node;
}
//=================================================
private void signalNotEmpty() {
    // 获取读锁
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        // 唤醒。
        notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
}
sync -> wait / notify
```

3.2.3 offer(time,unit)方法

```
public boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException {
        // 非空检验
    if (e == null) throw new NullPointerException();
    // 将时间转换为纳秒
    long nanos = unit.toNanos(timeout);
    // 标记
    int c = -1;
    // 写锁，数据条数
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    // 允许中断的加锁方式
    putLock.lockInterruptibly();
    try {
        // 如果元素个数和限制个数一致，直接准备挂起
        while (count.get() == capacity) {
            // 挂起的时间是不是已经没了
            if (nanos <= 0)
                // 添加失败，返回false
                return false;
            // 挂起线程
            nanos = notFull.awaitNanos(nanos);
        }
        // 有空余位置，enqueue添加数据
        enqueue(new Node<E>(e));
        // 元素个数 + 1
        c = count.getAndIncrement();
        // 当前添加完数据，还有位置可以添加数据，唤醒可能阻塞的生产者
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        // 释放锁
        putLock.unlock();
    }
    // 如果之前元素个数是0，唤醒可能等待的消费者
    if (c == 0)
        signalNotEmpty();
    return true;
}
```

3.2.4 put方法

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
```

```
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        while (count.get() == capacity) {
            // 一直挂起线程，等待被唤醒
            notFull.await();
        }
        enqueue(node);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
}
```

3.3 消费者方法实现原理

从remove方法开始，查看消费者获取数据的方式

3.3.1 remove方法

```
public E remove() {
    E x = poll();
    if (x != null)
        return x;
    else
        throw new NoSuchElementException();
}
```

3.3.2 poll方法

```
public E poll() {
    // 拿到队列数据个数的计数器
    final AtomicInteger count = this.count;
    // 当前队列中数据是否0
    if (count.get() == 0)
        // 说明队列没数据，直接返回null即可
        return null;
```

```
    // 声明返回结果
    E x = null;
    // 标记
    int c = -1;
    // 获取消费者的takeLock
    final ReentrantLock takeLock = this.takeLock;
    // 加锁
    takeLock.lock();
    try {
        // 基于DCL，确保当前队列中依然有元素
        if (count.get() > 0) {
            // 从队列中移除数据
            x = dequeue();
            // 将之前的元素个数获取，并--
            c = count.getAndDecrement();
            if (c > 1)
                // 如果依然有数据，继续唤醒await的消费者。
                notEmpty.signal();
        }
    } finally {
        // 释放锁资源
        takeLock.unlock();
    }
    // 如果之前的元素个数为当前队列的限制长度，
    // 现在消费者消费了一个数据，多了一个空位可以添加
    if (c == capacity)
        // 唤醒阻塞的生产者
        signalNotFull();
    return x;
}

//============================================

private E dequeue() {
    // 拿到队列的head位置数据
    Node<E> h = head;
    // 拿到了head的next，因为这个是哨兵Node，需要拿到的head.next的数据
    Node<E> first = h.next;
    // 将之前的哨兵Node.next置位null。help GC。
    h.next = h;
    // 将first置位新的head
    head = first;
    // 拿到返回结果first节点的item数据，也就是之前head.next.item
    E x = first.item;
    // 将first数据置位null，作为新的head
    first.item = null;
    // 返回数据
```

```
    return x;
}

//==================================================

private void signalNotFull() {
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        // 唤醒生产者。
        notFull.signal();
    } finally {
        putLock.unlock();
    }
}
```

3.3.3 poll(time,unit)方法

```
public E poll(long timeout, TimeUnit unit) throws InterruptedException {
    // 返回结果
    E x = null;
    // 标识
    int c = -1;
    // 将挂起实现设置为纳秒级别
    long nanos = unit.toNanos(timeout);
    // 拿到计数器
    final AtomicInteger count = this.count;
    // take锁加锁
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        // 如果没数据，进到while
        while (count.get() == 0) {
            if (nanos <= 0)
                return null;
            // 挂起当前线程
            nanos = notEmpty.awaitNanos(nanos);
        }
        // 剩下内容，和之前一样。
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
```

```
    if (c == capacity)
        signalNotFull();
    return x;
}
```

3.3.4 take方法

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        // 相比poll(time,unit)方法，这里的出口只有一个，就是中断标记位，抛出异常，否则一直等待
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

四、PriorityBlockingQueue概念

4.1 PriorityBlockingQueue介绍

首先PriorityBlockingQueue是一个优先级队列，他不满足先进先出的概念。

会将查询的数据进行排序，排序的方式就是基于插入数据值的本身。

如果是自定义对象必须要实现Comparable接口才可以添加到优先级队列

排序的方式是基于**二叉堆**实现的。底层是采用数据结构实现的二叉堆。

```java
public static void main(String[] args) throws InterruptedException {
    PriorityBlockingQueue queue = new PriorityBlockingQueue();
    queue.add("234");
    queue.add("123");
    queue.add("456");
    queue.add("345");
    System.out.println(queue.poll());
    System.out.println(queue.poll());
    System.out.println(queue.poll());
    System.out.println(queue.poll());
}
}
```

CompanyTest

CompanyTest

```
D:\jdk\bin\java.exe ...
123
234
345
456
```

4.2 二叉堆结构介绍

优先级队列PriorityBlockingQueue基于二叉堆实现的。

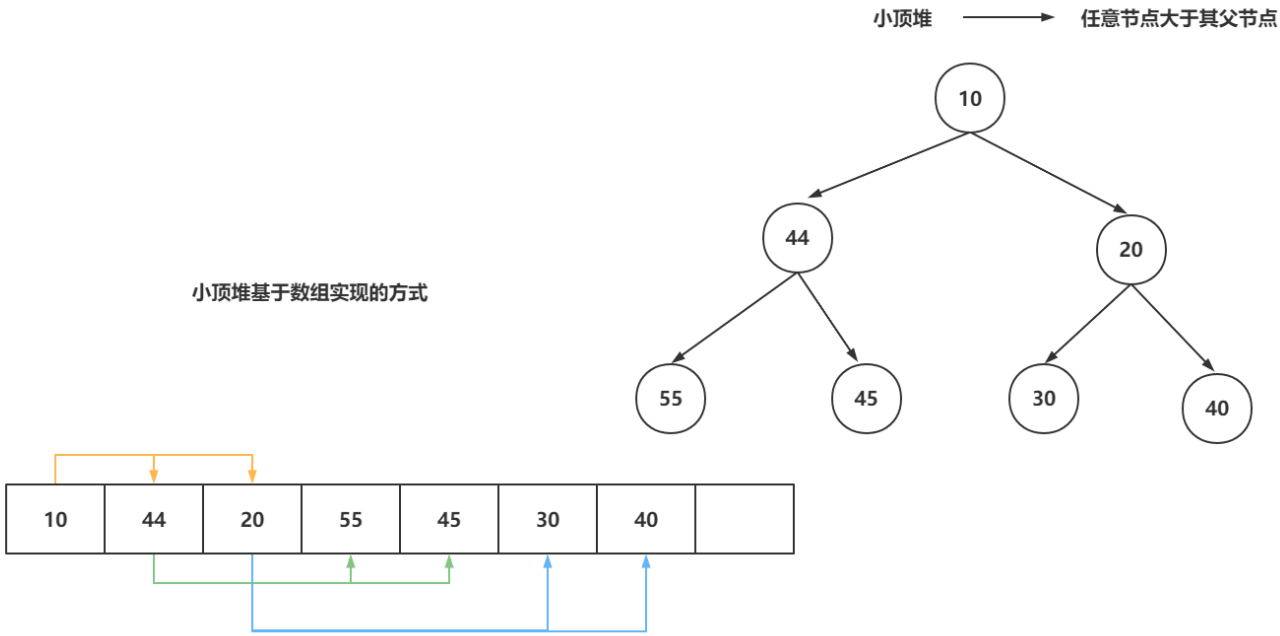
```
private transient Object[] queue;
```

PriorityBlockingQueue是基于数组实现的二叉堆。

二叉堆是什么？

- 二叉堆就是一个完整的二叉树。
- 任意一个节点大于父节点或者小于父节点
- 基于同步的方式，可以定义出小顶堆和大顶堆

小顶堆以及小顶堆基于数据实现的方式。



4.3 PriorityBlockingQueue核心属性

```
// 数组的初始长度
private static final int DEFAULT_INITIAL_CAPACITY = 11;

// 数组的最大长度
// -8的目的是为了适配各个版本的虚拟机
// 默认当前使用的hotspot虚拟机最大支持Integer.MAX_VALUE - 2，但是其他版本的虚拟机不一定。
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

// 存储数据的数组，也是基于这个数组实现的二叉堆。
private transient Object[] queue;

// size记录当前阻塞队列中元素的个数
private transient int size;

// 要求使用的对象要实现Comparable比较器。基于comparator做对象之间的比较
private transient Comparator<? super E> comparator;

// 实现阻塞队列的lock锁
private final ReentrantLock lock;

// 挂起线程操作。
private final Condition notEmpty;

// 因为PriorityBlockingQueue的底层是基于二叉堆的，而二叉堆又是基于数组实现的，数组长度是固定的，如果需要
private transient volatile int allocationSpinLock;
```

```
// 阻塞队列中用到的原理，其实就是普通的优先级队列。
private PriorityQueue<E> q;
```

4.4 PriorityBlockingQueue的写入操作

毕竟是阻塞队列，添加数据的操作，咱们是很了解，无法还是add，offer，offer(time,unit)，put。但是因为优先级队列中，数组是可以扩容的，虽然有长度限制，但是依然属于无界队列的概念，所以生产者不会阻塞，所以只有offer方法可以查看。

这次核心的内容并不是添加数据的区别。主要关注的是如何保证二叉堆中小顶堆的结构的，并且还要查看数组扩容的一个过程是怎样的。

4.4.1 offer基本流程

因为add方法依然调用的是offer方法，直接查看offer方法即可

```
public boolean offer(E e) {
    // 非空判断。
    if (e == null)
        throw new NullPointerException();
    // 拿到锁，直接上锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    // n：size，元素的个数
    // cap：当前数组的长度
    // array：就是存储数据的数组
    int n, cap;
    Object[] array;
    while ((n = size) >= (cap = (array = queue).length))
        // 如果元素个数大于等于数组的长度，需要尝试扩容。
        tryGrow(array, cap);
    try {
        // 拿到了比较器
        Comparator<? super E> cmp = comparator;
        // 比较数据大小，存储数据，是否需要做上移操作，保证平衡的
        if (cmp == null)
            siftUpComparable(n, e, array);
        else
            siftUpUsingComparator(n, e, array, cmp);
        // 元素个数 + 1
        size = n + 1;
        // 如果有挂起的线程，需要去唤醒挂起的消费者。
        notEmpty.signal();
    } finally {
        // 释放锁
```

```
            lock.unlock();
        }
        // 返回true
        return true;
    }
```

4.4.2 offer扩容操作

在添加数据之前，会采用while循环的方式，来判断当前元素个数是否大于等于数组长度。如果满足，需要执行tryGrow方法，对数组进行扩容

如果两个线程同时执行tryGrow，只会有一个线程在扩容，另一个线程可能多次走while循环，多次走tryGrow方法，但是依然需要等待前面的线程扩容完毕。

```
private void tryGrow(Object[] array, int oldCap) {
    // 释放锁资源。
    lock.unlock();
    // 声明新数组。
    Object[] newArray = null;
    // 如果allocationSpinLock属性值为0，说明当前没有线程正在扩容的。
    if (allocationSpinLock == 0 &&
        // 基于CAS的方式，将allocationSpinLock从0修改为1，代表当前线程可以开始扩容
        UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,0, 1)) {
        try {
            // 计算新数组长度
            int newCap = oldCap + ((oldCap < 64) ?
                                   // 如果数组长度比较小，这里加快扩容长度速度。
                                   (oldCap + 2) :
                                   // 如果长度大于等于64了，每次扩容到1.5倍即可。
                                   (oldCap >> 1));
            // 如果新数组长度大于MAX_ARRAY_SIZE，需要做点事了。
            if (newCap - MAX_ARRAY_SIZE > 0) {
                // 声明minCap，长度为老数组 + 1
                int minCap = oldCap + 1;
                // 老数组+1变为负数，或者老数组长度已经大于MAX_ARRAY_SIZE了，无法扩容了。
                if (minCap < 0 || minCap > MAX_ARRAY_SIZE)
                    // 告辞，凉凉~~~~
                    throw new OutOfMemoryError();
                // 如果没有超过限制，直接设置为最大长度即可
                newCap = MAX_ARRAY_SIZE;
            }
            // 新数组长度，得大于老数组长度，
            // 第二个判断确保没有并发扩容的出现。
            if (newCap > oldCap && queue == array)
                // 构建出新数组
```

```
                newArray = new Object[newCap];
            } finally {
                // 新数组有了，标记位归0~~
                allocationSpinLock = 0;
            }
        }
        // 如果到了这，newArray依然为null，说明这个线程没有进到if方法中，去构建新数组
        if (newArray == null)
            // 稍微等一手。
            Thread.yield();
        // 拿锁资源，
        lock.lock();
        // 拿到锁资源后，确认是构建了新数组的线程，这里就需要将新数组复制给queue，并且导入数据
        if (newArray != null && queue == array) {
            // 将新数组赋值给queue
            queue = newArray;
            // 将老数组的数据全部导入到新数组中。
            System.arraycopy(array, 0, newArray, 0, oldCap);
        }
    }
```

4.4.3 offer添加数据

这里是数据如何放到数组上，并且如何保证的二叉堆结构

```
    // k：当前元素的个数（其实就是要放的索引位置）
    // x：需要添加的数据
    // array：数组。。
    private static <T> void siftUpComparable(int k, T x, Object[] array) {
        // 将插入的元素直接强转为Comparable（com.mashibing.User cannot be cast to java.lang.Comparable）
        // 这行强转，会导致添加没有实现Comparable的元素，直接报错。
        Comparable<? super T> key = (Comparable<? super T>) x;
        // k大于0，走while逻辑。（原来有数据）
        while (k > 0) {
            // 获取父节点的索引位置。
            int parent = (k - 1) >>> 1;
            // 拿到父节点的元素。
            Object e = array[parent];
            // 用子节点compareTo父节点，如果 >= 0，说明当前son节点比parent要大。
            if (key.compareTo((T) e) >= 0)
                // 直接break，完事，
                break;
            // 将son节点的位置设置上之前的parent节点
            array[k] = e;
            // 重新设置x节点需要放置的位置。
            k = parent;
```

```
    }
    // k == 0，当前元素是第一个元素，直接插入进去。
    array[k] = key;
}
```

4.5 PriorityBlockingQueue的读取操作

读取操作是存储现在挂起的情况的，因为如果数组中元素个数为0，当前线程如果执行了take方法，必然需要挂起。

其次获取数据，因为是优先级队列，所以需要从二叉堆栈顶拿数据，直接拿索引为0的数据即可，但是拿完之后，需要保持二叉堆结构，所以会有下移操作。

4.5.1 查看获取方法流程

poll：

```
public E poll() {
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        // 拿到返回数据，没拿到，返回null
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

poll(time,unit)：

```
public E poll(long timeout, TimeUnit unit) throws InterruptedException {
    // 将挂起的时间转换为纳秒
    long nanos = unit.toNanos(timeout);
    final ReentrantLock lock = this.lock;
    // 允许线程中断抛异常的加锁
    lock.lockInterruptibly();
    // 声明结果
    E result;
    try {
        // dequeue是去拿数据的，可能会出现拿到的数据为null，如果为null，同时挂起时间还有剩余，这边就直接通过
        while ( (result = dequeue()) == null && nanos > 0)
            nanos = notEmpty.awaitNanos(nanos);
    } finally {
        lock.unlock();
```

```
    }
    // 有数据正常返回，没数据，告辞~
    return result;
}
```

take：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    E result;
    try {
        while ( (result = dequeue()) == null)
            // 无线等，要么有数据，要么中断线程
            notEmpty.await();
    } finally {
        lock.unlock();
    }
    return result;
}
```

4.5.2 查看dequeue获取数据

获取数据主要就是从数组中拿到0索引位置数据，然后保持二叉堆结构

```
private E dequeue() {
    // 将元素个数-1，拿到了索引位置。
    int n = size - 1;
    // 判断是不是木有数据了，没数据直接返回null即可
    if (n < 0)
        return null;
    // 说明有数据
    else {
        // 拿到数组，array
        Object[] array = queue;
        // 拿到0索引位置的数据
        E result = (E) array[0];
        // 拿到最后一个数据
        E x = (E) array[n];
        // 将最后一个位置置位null
        array[n] = null;
        Comparator<? super E> cmp = comparator;
        if (cmp == null)
            siftDownComparable(0, x, array, n);
        else
```

```
            siftDownUsingComparator(0, x, array, n, cmp);
        // 元素个数-1，赋值size
        size = n;
        // 返回result
        return result;
    }
}
```

4.6.3 下移做平衡操作

一定要以局部的方式去查看树结构的变化，他是从跟节点往下找较小的一个子节点，将较小的子节点挪动到父节点位置，再将循环往下走，如果一来，整个二叉堆的结构就可以保证了。

```
// k：默认进来是0
// x：代表二叉堆的最后一个数据
// array：数组
// n：最后一个索引
private static <T> void siftDownComparable(int k, T x, Object[] array,int n) {
    // 健壮性校验，取完第一个数据，已经没数据了，那就不需要做平衡操作
    if (n > 0) {
        // 拿到最后一个数据的比较器
        Comparable<? super T> key = (Comparable<? super T>)x;
        // 因为二叉堆是一个二叉满树，所以在保证二叉堆结构时，只需要做一半就可以
        int half = n >>> 1;
        // 做了超过一半，就不需要再往下找了。
        while (k < half) {
            // 找左子节点索引，一个公式，可以找到当前节点的左子节点
            int child = (k << 1) + 1;
            // 拿到左子节点的数据
            Object c = array[child];
            // 拿到右子节点索引
            int right = child + 1;
            // 确认有右子节点
            // 判断左节点是否大于右节点
            if (right < n && c.compareTo(array[right]) > 0)
                // 如果左大于右，那么c就执行右
                c = array[child = right];
            // 比较最后一个节点是否小于当前的较小的子节点
            if (key.compareTo((T) c) <= 0)
                break;
            // 将左右子节点较小的放到之前的父节点位置
            array[k] = c;
            // k重置到之前的子节点位置
            k = child;
        }
        // 上面while循环搞定后，可以确认整个二叉堆中，数据已经移动ok了，只差当前k的位置数据是null
```

```
        // 将最后一个索引的数据放到k的位置
        array[k] = key;
    }
}
```

五、DelayQueue

5.1 DelayQueue介绍&应用

DelayQueue就是一个延迟队列，生产者写入一个消息，这个消息还有直接被消费的延迟时间。

需要让消息具有延迟的特性。

DelayQueue也是基于二叉堆结构实现的，甚至本事就是基于PriorityQueue实现的功能。二叉堆结构每次获取的是栈顶的数据，需要让DelayQueue中的数据，在比较时，跟根据延迟时间做比较，剩余时间最短的要放在栈顶。

查看DelayQueue类信息：

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E> implements BlockingQueue<E> {
    // 发现DelayQueue中的元素，需要继承Delayed接口。
}
// ==========================================
// 接口继承了Comparable，这样就具备了比较的能力。
public interface Delayed extends Comparable<Delayed> {
    // 抽象方法，就是咱们需要设置的延迟时间
    long getDelay(TimeUnit unit);

    // Comparable接口提供的：public int compareTo(T o);
}
```

基于上述特点，声明一个可以写入DelayQueue的元素类

```
public class Task implements Delayed {

    /** 任务的名称 */
    private String name;

    /** 什么时间点执行 */
    private Long time;

    /**
     *
     * @param name
```

```
     * @param delay  单位毫秒。
     */
    public Task(String name, Long delay) {
        // 任务名称
        this.name = name;
        this.time = System.currentTimeMillis() + delay;
    }

    /**
     * 设置任务什么时候可以出延迟队列
     * @param unit
     * @return
     */
    @Override
    public long getDelay(TimeUnit unit) {
                // 单位是毫秒，视频里写错了，写成了纳秒，
        return unit.convert(time - System.currentTimeMillis(),TimeUnit.MILLISECONDS);
    }

    /**
     * 两个任务在插入到延迟队列时的比较方式
     * @param o
     * @return
     */
    @Override
    public int compareTo(Delayed o) {
        return (int) (this.time - ((Task)o).getTime());
    }
}
```

在使用时，查看到DelayQueue底层用了PriorityQueue，在一定程度上，DelayQueue也是无界队列。

测试效果

```
public static void main(String[] args) throws InterruptedException {
    // 声明元素
    Task task1 = new Task("A",1000L);
    Task task2 = new Task("B",5000L);
    Task task3 = new Task("C",3000L);
    Task task4 = new Task("D",2000L);
    // 声明阻塞队列
    DelayQueue<Task> queue = new DelayQueue<>();
    // 将元素添加到延迟队列中
    queue.put(task1);
    queue.put(task2);
```

```
    queue.put(task3);
    queue.put(task4);
    // 获取元素
    System.out.println(queue.take());
    System.out.println(queue.take());
    System.out.println(queue.take());
    System.out.println(queue.take());
    // A,D,C,B
}
```

在应用时，外卖，15分钟商家需要节点，如果不节点，这个订单自动取消。

可以每下一个订单，就放到延迟队列中，如果规定时间内，商家没有节点，直接通过消费者获取元素，然后取消订单。

只要是有需要延迟一定时间后，再执行的任务，就可以通过延迟队列去实现。

5.2、DelayQueue核心属性

可以查看到DelayQueue就四个核心属性

```
// 因为DelayQueue依然属于阻塞队列，需要保证线程安全。看到只有一把锁，生产者和消费者使用的是一个lock
private final transient ReentrantLock lock = new ReentrantLock();
// 因为DelayQueue还是基于二叉堆结构实现的，没有必要重新搞一个二叉堆，直接使用的PriorityQueue
private final PriorityQueue<E> q = new PriorityQueue<E>();
// leader一般会存储等待栈顶数据的消费者，在整体写入和消费的过程中，会设置的leader的一些判断。
private Thread leader = null;
// 生产者在插入数据时，不会阻塞的。当前的Condition就是给消费者用的
// 比如消费者在获取数据时，发现栈顶的数据还又没到延迟时间。
// 这个时候，咱们就需要将消费者线程挂起，阻塞一会，阻塞到元素到了延迟时间，或者是，生产者插入的元素到了栈
private final Condition available = lock.newCondition();
```

5.3、DelayQueue写入流程分析

Delay是无界的，数组可以动态的扩容，不需要关注生产者的阻塞问题，他就没有阻塞问题。

这里只需要查看offer方法即可。

```
public boolean offer(E e) {
    // 直接获取lock，加锁。
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 直接调用PriorityQueue的插入方法，这里会根据之前重写Delayed接口中的compareTo方法做排序，然后调整
        q.offer(e);
```

```
            // 调用优先级队列的peek方法，拿到堆顶的数据
            // 拿到堆顶数据后，判断是否是刚刚插入的元素
            if (q.peek() == e) {
                // leader赋值为null。在消费者的位置再提一嘴
                leader = null;
                // 唤醒消费者，避免刚刚插入的数据的延迟时间出现问题。
                available.signal();
            }
            // 插入成功，
            return true;
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
```

5.4、DelayQueue读取流程分析

消费者依然还是存在阻塞的情况，因为有两个情况

- 消费者要拿到栈顶数据，但是延迟时间还没到，此时消费者需要等待一会。
- 消费者要来拿数据，但是发现已经有消费者在等待栈顶数据了，这个后来的消费者也需要等待一会。

依然需要查看四个方法的实现

5.4.1 remove方法

```
// 依然是AbstractQueue提供的方法，有结果就返回，没结果扔异常
public E remove() {
    E x = poll();
    if (x != null)
        return x;
    else
        throw new NoSuchElementException();
}
```

5.4.2 poll方法

```
// poll是浅尝一下，不会阻塞消费者，能拿就拿，拿不到就拉倒
public E poll() {
    // 消费者和生产者是一把锁，先拿锁，加锁。
    final ReentrantLock lock = this.lock;
    lock.lock();
```

```
    try {
            // 拿到栈顶数据。
        E first = q.peek();
        // 如果元素为null，直接返回null
        // 如果getDelay方法返回的结果是大于0的，那说明当前元素还每到延迟时间，元素无法返回，返回null
        if (first == null || first.getDelay(NANOSECONDS) > 0)
            return null;
        else
            // 到这说明元素不为null，并且已经达到了延迟时间，直接调用优先级队列的poll方法
            return q.poll();
    } finally {
        // 释放锁。
        lock.unlock();
    }
}
```

5.4.3 poll(time,unit)方法

这个是允许阻塞的，并且指定一定的时间

```
public E poll(long timeout, TimeUnit unit) throws InterruptedException {
    // 先将时间转为纳秒
    long nanos = unit.toNanos(timeout);
    // 拿锁，加锁。
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        // 死循环。
        for (;;) {
            // 拿到堆顶数据
            E first = q.peek();
            // 如果元素为null
            if (first == null) {
                // 并且等待的时间小于等于0。不能等了，直接返回null
                if (nanos <= 0)
                    return null;
                // 说明当前线程还有可以阻塞的时间，阻塞指定时间即可。
                else
                    // 这里挂起线程后，说明队列没有元素，在生产者添加数据之后，会唤醒
                    nanos = available.awaitNanos(nanos);
            // 到这说明，有数据
            } else {
                // 有数据的话，先获取数据现在是否可以执行，延迟时间是否已经到了指定时间
                long delay = first.getDelay(NANOSECONDS);
                // 延迟时间是否已经到了，
                if (delay <= 0)
```

```
                // 时间到了，直接执行优先级队列的poll方法，返回元素
                return q.poll();
            // ====================延迟时间没到，消费者需要等一会====================
            // 这个是查看消费者可以等待的时间，
            if (nanos <= 0)
                // 直接返回nulll
                return null;
            // ====================延迟时间没到，消费者可以等一会====================
            // 把first赋值为null
            first = null;
            // 如果等待的时间，小于元素剩余的延迟时间，消费者直接挂起。反正暂时拿不到，但是不能保证后续是否
            // 如果已经有一个消费者在等待堆顶数据了，我这边不做额外操作，直接挂起即可。
            if (nanos < delay || leader != null)
                nanos = available.awaitNanos(nanos);
            // 当前消费者的阻塞时间可以拿到数据，并且没有其他消费者在等待堆顶数据
            else {
                // 拿到当前消费者的线程对象
                Thread thisThread = Thread.currentThread();
                // 将leader设置为当前线程
                leader = thisThread;
                try {
                    // 会让当前消费者，阻塞这个元素的延迟时间
                    long timeLeft = available.awaitNanos(delay);
                    // 重新计算当前消费者剩余的可阻塞时间，。
                    nanos -= delay - timeLeft;
                } finally {
                    // 到了时间，将leader设置为null
                    if (leader == thisThread)
                        leader = null;
                }
            }
        }
    } finally {
        // 没有消费者在等待元素，队列中的元素不为null
        if (leader == null && q.peek() != null)
            // 只要当前没有leader在等，并且队列有元素，就需要再次唤醒消费者。、
            // 避免队列有元素，但是没有消费者处理的问题
            available.signal();
        // 释放锁
        lock.unlock();
    }
}
```

5.4.4 take方法

这个是允许阻塞的，但是可以一直等，要么等到元素，要么等到被中断。

```
public E take() throws InterruptedException {
    // 正常加锁，并且允许中断
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            // 拿到元素
            E first = q.peek();
            if (first == null)
                // 没有元素挂起。
                available.await();
            else {
                // 有元素，获取延迟时间。
                long delay = first.getDelay(NANOSECONDS);
                // 判断延迟时间是不是已经到了
                if (delay <= 0)
                    // 基于优先级队列的poll方法返回
                    return q.poll();
                first = null;
                // 如果有消费者在等，就正常await挂起
                if (leader != null)
                    available.await();
                // 如果没有消费者在等的堆顶数据，我来等
                else {
                    // 获取当前线程
                    Thread thisThread = Thread.currentThread();
                    // 设置为leader，代表等待堆顶的数据
                    leader = thisThread;
                    try {
                        // 等待指定（堆顶元素的延迟时间）时长，
                        available.awaitNanos(delay);
                    } finally {
                        if (leader == thisThread)
                            // leader赋值null
                            leader = null;
                    }
                }
            }
        }
    } finally {
        // 避免消费者无线等，来一个唤醒消费者的方法，一般是其他消费者拿到元素走了之后，并且延迟队列还有元素
        if (leader == null && q.peek() != null)
            available.signal();
        // 释放锁
```

```
            lock.unlock();
        }
    }
```

六、SynchronousQueue

6.1 SynchronousQueue介绍

SynchronousQueue这个阻塞队列和其他的阻塞队列有很大的区别

在咱们的概念中，队列肯定是要存储数据的，但是SynchronousQueue不会存储数据的

SynchronousQueue队列中，他不存储数据，存储生产者或者是消费者

当存储一个生产者到SynchronousQueue队列中之后，生产者会阻塞（看你调用的方法）

生产者最终会有几种结果：

- 如果在阻塞期间有消费者来匹配，生产者就会将绑定的消息交给消费者
- 生产者得等阻塞结果，或者不允许阻塞，那么就直接失败
- 生产者在阻塞期间，如果线程中断，直接告辞。

同理，消费者和生产者的效果是一样。

生产者和消费者的数据是直接传递的，不会经过SynchronousQueue。

SynchronousQueue是不会存储数据的。

经过阻塞队列的学习：

生产者：

- offer()：生产者在放到SynchronousQueue的同时，如果有消费者在等待消息，直接配对。如果没有消费者在等待消息，这里直接返回，告辞。
- offer(time,unit)：生产者在放到SynchronousQueue的同时，如果有消费者在等待消息，直接配对。如果没有消费者在等待消息，阻塞time时间，如果还没有，告辞。
- put()：生产者在放到SynchronousQueue的同时，如果有消费者在等待消息，直接配对。如果没有，死等。

消费者：poll()，poll(time,unit)，take()。道理和上面的生产者一致。

测试效果：

```
public static void main(String[] args) throws InterruptedException {
    // 因为当前队列不存在数据，没有长度的概念。
    SynchronousQueue queue = new SynchronousQueue();

    String msg = "消息！";
    /*new Thread(() -> {
        // b = false：代表没有消费者来拿
        boolean b = false;
        try {
            b = queue.offer(msg,1, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(b);
    }).start();

    Thread.sleep(100);

    new Thread(() -> {
        System.out.println(queue.poll());
    }).start();*/
    new Thread(() -> {
        try {
            System.out.println(queue.poll(1, TimeUnit.SECONDS));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }).start();

    Thread.sleep(100);

    new Thread(() -> {
        queue.offer(msg);
    }).start();
}
```

6.2 SynchronousQueue核心属性

进到SynchronousQueue类的内部后，发现了一个内部类，Transferer，内部提供了一个transfer的方法

```
abstract static class Transferer<E> {
    abstract E transfer(E e, boolean timed, long nanos);
}
```

当前这个类中提供的transfer方法，就是生产者和消费者在调用读写数据时要用到的核心方法。

生产者在调用上述的transfer方法时，第一个参数e会正常传递数据

消费者在调用上述的transfer方法时，第一个参数e会传递null

SynchronousQueue针对抽象类Transferer做了几种实现。

一共看到了两种实现方式：

- TransferStack
- TransferQueue

这两种类继承了Transferer抽象类，在构建SynchronousQueue时，会指定使用哪种子类

```
// 到底采用哪种实现，需要把对应的对象存放到这个属性中
private transient volatile Transferer<E> transferer;
// 采用无参时，会调用下述方法，再次调用有参构造传入false
public SynchronousQueue() {
    this(false);
}
// 调用的是当前的有参构造，fair代表公平还是不公平
public SynchronousQueue(boolean fair) {
    // 如果是公平，采用Queue，如果是不公平，采用Stack
    transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
}
```

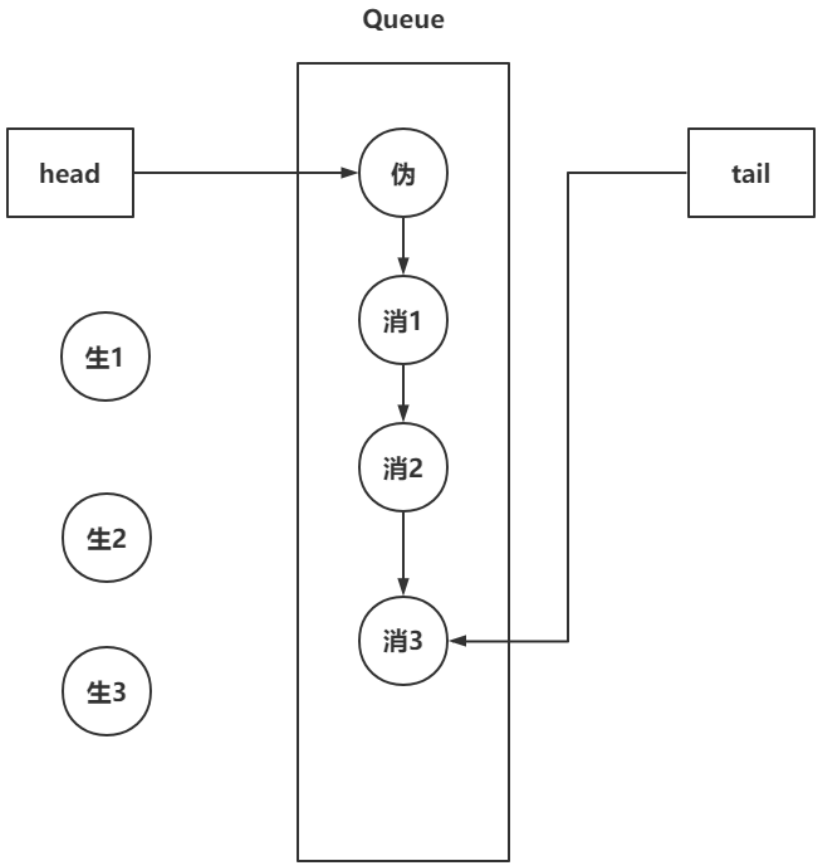
TransferQueue的特点

TransferQueue

看名字就知道，这个东西是一个队列结构。
队列就是先进先出的套路。

基于整个结构可以发现，先入队的消费者或者生产者，先进入的，先匹配。

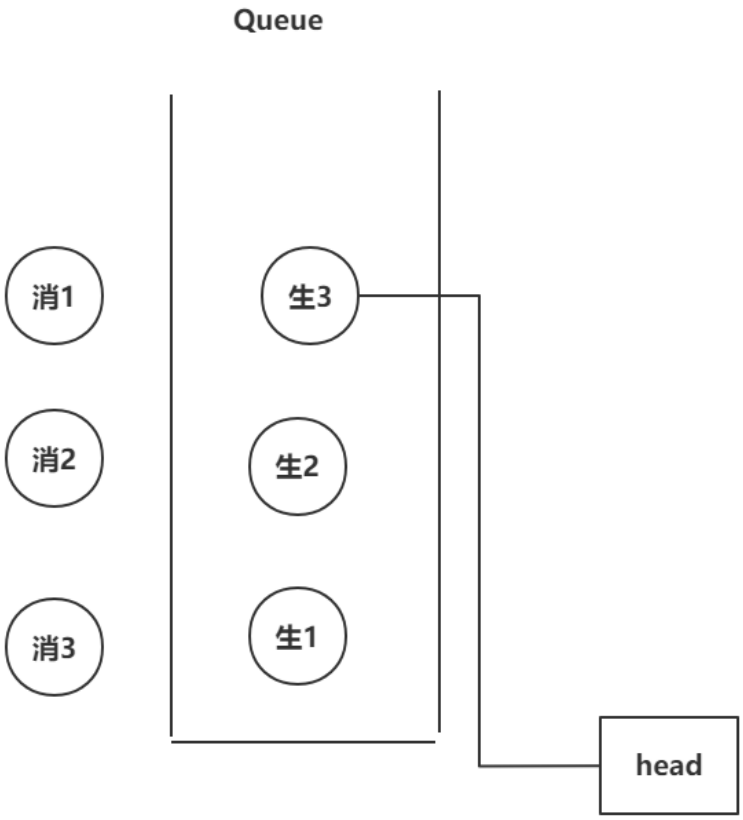
所以这种称为公平



TransferStack

看名字就知道，这个东西是一个栈结构
栈结构是先进后出，先获取栈顶的数据

先入栈的生产者或者是消费者，不是先出栈的。
反而后如栈的才是先匹配上的



代码查看效果

```
public static void main(String[] args) throws InterruptedException {
    // 因为当前队列不存在数据，没有长度的概念。
    SynchronousQueue queue = new SynchronousQueue(true);
    SynchronousQueue queue = new SynchronousQueue(false);

    new Thread(() -> {
        try {
            queue.put("生1");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }).start();
    new Thread(() -> {
        try {
            queue.put("生2");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }).start();
    new Thread(() -> {
        try {
            queue.put("生3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }).start();

    Thread.sleep(100);
    new Thread(() -> {
        System.out.println("消1：" + queue.poll());
    }).start();
    Thread.sleep(100);
    new Thread(() -> {
        System.out.println("消2：" + queue.poll());
    }).start();
    Thread.sleep(100);
    new Thread(() -> {
        System.out.println("消3：" + queue.poll());
    }).start();
}
```

6.3 SynchronousQueue的TransferQueue源码

为了查看清除SynchronousQueue的TransferQueue源码，需要从两点开始查看源码信息

6.3.1 QNode源码信息

```
static final class QNode {
    // 当前节点可以获取到next节点
    volatile QNode next;
    // item在不同情况下效果不同
    // 生产者：有数据
    // 消费者：为null
    volatile Object item;
    // 当前线程
    volatile Thread waiter;
    // 当前属性是永磊区分消费者和生产者的属性
    final boolean isData;
    // 最终生产者需要将item交给消费者
    // 最终消费者需要获取生产者的item

    // 省略了大量提供的CAS操作
    ....
}
```

6.3.2 transfer方法实现

```
// 当前方法是TransferQueue的核心内容
// e：传递的数据
// timed：false，代表无限阻塞，true，代表阻塞nacos时间
E transfer(E e, boolean timed, long nanos) {
    // 当前QNode是要封装当前生产者或者消费者的信息
    QNode s = null;
    // isData == true：代表是生产者
    // isData == false：代表是消费者
    boolean isData = (e != null);
    // 死循环
    for (;;) {
        // 获取尾节点和头结点
        QNode t = tail;
        QNode h = head;
        // 为了避免TransferQueue还没有初始化，这边做一个健壮性判断
        if (t == null || h == null)
            continue;

        // 如果满足h == t 条件，说明当前队列没有生产者或者消费者，为空
        // 如果有节点，同时当前节点和队列节点属于同一种角色。
        // if中的逻辑是进到队列
        if (h == t || t.isData == isData) {
            // ====================在判断并发问题==========================
```

```
// 拿到尾节点的next
QNode tn = t.next;
// 如果t不为尾节点，进来说明有其他线程并发修改了tail
if (t != tail)
    // 重新走for循环
    continue;
// tn如果为不null，说明前面有线程并发，添加了一个节点
if (tn != null) {
    // 直接帮助那个并发线程修改tail的指向
    advanceTail(t, tn);
    // 重新走for循环
    continue;
}
// 获取当前线程是否可以阻塞
// 如果timed为true，并且阻塞的时间小于等于0
// 不需要匹配，直接告辞！！！
if (timed && nanos <= 0)
    return null;
// 如果可以阻塞，将当前需要插入到队列的QNode构建出来
if (s == null)
    s = new QNode(e, isData);
// 基于CAS操作，将tail节点的next设置为当前线程
if (!t.casNext(null, s))
    // 如果进到if，说明修改失败，重新执行for循环修改
    continue;
// CAS操作成功，直接替换tail的指向
advanceTail(t, s);
// 如果进到队列中了，挂起线程，要么等生产者，要么等消费者。
// x是返回替换后的数据
Object x = awaitFulfill(s, e, timed, nanos);
// 如果元素和节点相等，说明节点取消了
if (x == s) {
    // 清空当前节点，将上一个节点的next指向当前节点的next，直接告辞
    clean(t, s);
    return null;
}
// 判断当前节点是否还在队列中
if (!s.isOffList()) {
    // 将当前节点设置为head
    advanceHead(t, s);
    // 如果 x != null， 如果拿到了数据，说明我是消费者
    if (x != null)
        // 将当前节点的item设置为自己
        s.item = s;
    // 线程置位null
    s.waiter = null;
}
```

```
            // 返回数据
            return (x != null) ? (E)x : e;
        }
        // 匹配队列中的橘色
        else {
            // 拿到head的next，作为要匹配的节点
            QNode m = h.next;
            // 做并发判断，如果头节点，尾节点，或者head.next发生了变化，这边要重新走for循环
            if (t != tail || m == null || h != head)
                continue;
            // 没并发问题，可以拿数据
            // 拿到m节点的item作为x。
            Object x = m.item;
            // 如果isData == (x != null)满足，说明当前出现了并发问题，消费者去匹配队列的消费者不合理
            if (isData == (x != null) ||
                // 如果排队的节点取消，就会讲当前QNode中的item指向QNode
                x == m ||
                // 如果前面两个都没满足，可以交换数据了。
                // 如果交换失败，说明有并发问题，
                !m.casItem(x, e)) {
                // 重新设置head节点，并且再走一次循环
                advanceHead(h, m);
                continue;
            }
            // 替换head
            advanceHead(h, m);
            // 唤醒head.next中的线程
            LockSupport.unpark(m.waiter);
            // 这边匹配好了，数据也交换了，直接返回
            // 如果 x != null，说明队列中是生产者，当前是消费者，这边直接返回x具体数据
            // 反之，队列中是消费者，当前是生产者，直接返回自己的数据
            return (x != null) ? (E)x : e;
        }
    }
}
```

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1