JAVA多线程

1、线程基础

1.1、多线程的实现方式

灵活性：**通常建议实现Runnable接口而不是继承Thread类。**实现Runnable接口允许你的类继承其他类，因为Java不支持多重继承。此外，实现Runnable接口可以让你的设计更加灵活，比如同一个Runnable实例可以被多个Thread`实例包装。

Callable接口是Java中的一个并发编程的工具，它与Runnable接口类似，但提供了更多功能。**Callable的主要特点是它的call()方法可以返回结果并且可以抛出异常。**Callable通常与FutureTask或ExecutorService一起使用，以实现具有返回值的多线程任务。

1.1.1、继承Thread类

自定义一个MyThread类，用来继承Thread类，在MyThread类中重写run（）方法，启动线程。

```
setName(String name):设置线程名
getName():返回字符串形式的线程名
Thread.CurrentThread():返回当前正在执行的线程对象

public class Demo01_extends_thread {
    public static void main(String[] args) {
        //创建线程
        MyThread t01 = new MyThread();
        MyThread t02 = new MyThread();
        MyThread t03 = new MyThread("线程03");

        t01.start();
        t02.start();
        t03.start();
```

```
        //设置线程名（补救的设置线程名的方式）
        t01.setName("线程01");
        t02.setName("线程02");
        //设置主线程名称
        Thread.currentThread().setName("主线程");
        for (int i = 0; i < 50; i++) {
            //Thread.currentThread() 获取当前正在执行线
程的对象

 System.out.println(Thread.currentThread().getName()
+ ":" + i);
        }
    }
}

class MyThread extends Thread{
    public MyThread() {}

    public MyThread(String name) {
        super(name);
    }

    //run方法是每个线程运行过程中都必须执行的方法
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(this.getName() + ":"
+ i);
        }
    }

}
```

继承 Thread 类的优缺点：

- 优点：编码简单

- 缺点：线程类已经继承了 Thread 类无法继承其他类了，功能不能通过继承拓展（单继承的局限性）

1.1.2、实现Runnable接口

自定义一个MyRunnable类来实现Runnable接口，在MyRunnable类中重写run()方法，创建Thread对象，并把MyRunnable对象作为Thread类构造方法的参数传递进去，启动线程。

```
public class Demo02 {

    public static void main(String[] args) {
        MyRunnable myRun = new MyRunnable();//将一个任务提取出来，让多个线程共同去执行
        //封装线程对象
        Thread t01 = new Thread(myRun, "线程01");
        Thread t02 = new Thread(myRun, "线程02");
        Thread t03 = new Thread(myRun, "线程03");
        //开启线程
        t01.start();
        t02.start();
        t03.start();
        //通过匿名内部类的方式创建线程
        new Thread(() -> {
            for (int i = 0; i < 20; i++) {

 System.out.println(Thread.currentThread().getName() + " - " + i);
            }
        },"线程04").start();
    }
}

//自定义线程类，实现Runnable接口
//这并不是一个线程类，是一个可运行的类，它还不是一个线程。
class MyRunnable implements Runnable{
```

```
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {

 System.out.println(Thread.currentThread().getName()
+ " - " + i);
        }
    }
}
```

- 线程任务类只是实现了 Runnable 接口，可以继续继承其他类，避免了单继承的局限性
- 线程池可以放入实现 Runnable 或 Callable 线程任务对象

1.1.3、实现Callable接口+FutureTask

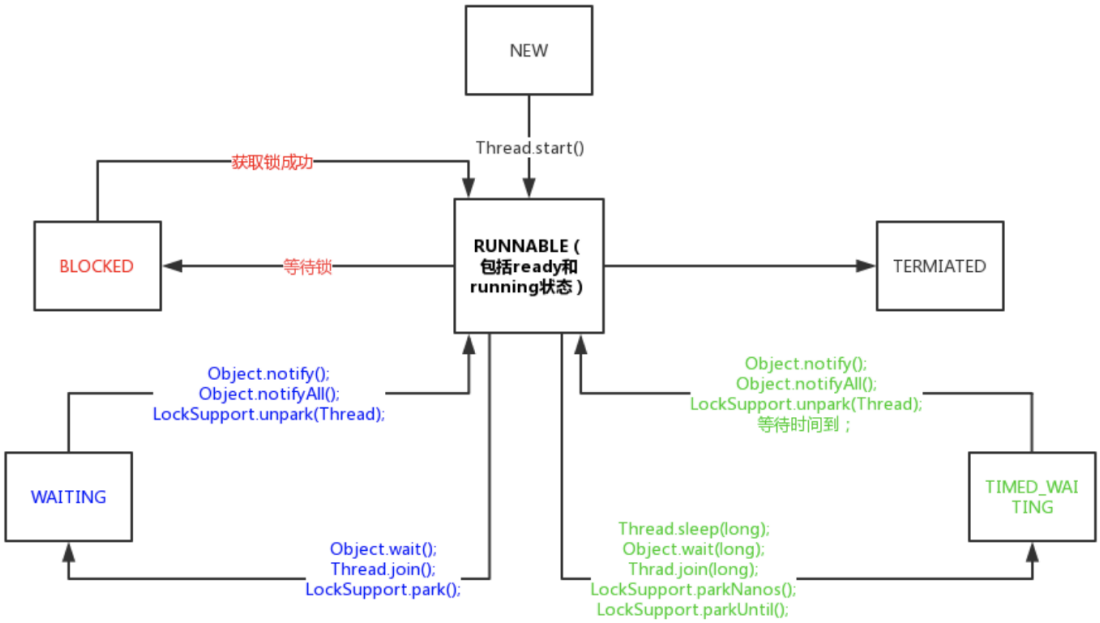
```
public class Demo03_Callable {

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

        FutureTask<Integer> task = new FutureTask<>
(new Callable<Integer>() {
            @Override
            public Integer call() {
                // try with resource ...
                System.out.println("Callable task
completed");
                return 100 + 200; // 返回计算结果
            }
        });

        task.run();
//        Thread t = new Thread(task);
//        t.start();
```

```
        // 在获取FutureTask结果时不适用try-with-
resources，因为这里没有自动关闭的资源
        try {
            Integer result = task.get(); // 这可能会阻
塞调用它的线程
            System.out.println("Thread execution
result: " + result);
        } catch (InterruptedException |
ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread continues
after retrieving the result");
    }
}
```

实现 Callable 接口：

1. 定义一个线程任务类实现 Callable 接口，申明线程执行的结果类型
2. 重写线程任务类的 call 方法，这个方法可以直接返回执行的结果
3. 创建一个 Callable 的线程任务对象
4. 把 Callable 的线程任务对象包装成一个未来任务对象 FutureTask
5. 把未来任务对象包装成线程对象
6. **调用线程的 start() 方法启动线程(直接调用run是直接在主线程中运行)**

1.2、线程控制



Java线程在任何时刻都处于以下几种状态之一：

- NEW：线程已经创建，但还没有开始执行。
- **RUNNABLE**：线程可能正在运行也可能正在等待CPU分配时间片。
- **BLOCKED**：线程**因为尝试访问一个被其他线程锁住的同步块/方法而阻塞。**
- **WAITING**：线程**无限期等待**另一个线程执行特定操作（例如，通过 `Object.wait`、`Thread.join`、`LockSupport.park`）。
- **TIMED_WAITING**：线程**等待**另一个线程执行特定操作，但**等待时间有上限**（例如，`Thread.sleep`、`Object.wait`带有超时参数、`LockSupport.parkNanos`、`LockSupport.parkUntil`）。
- TERMINATED：线程已经结束执行。

1.2.1、sleep()

```
public class DemoInterrupt {

    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable2());
        t.setName("t");
        t.start();
        try {
            Thread.sleep(1000 * 5);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        t.stop(); //强行终止线程 缺点:容易损坏数据   线程没有保存的数据容易丢失
    }
}

class MyRunnable2 implements Runnable {
    @Override
    public void run() {

 System.out.println(Thread.currentThread().getName() + "---> begin");
        try {
            // 睡眠10s
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //10s之后才会执行这里

 System.out.println(Thread.currentThread().getName() + "---> end");

    }

}

子线程的状态1RUNNABLE
子线程开始Sleep
子线程的状态2TIMED_WAITING
子线程结束Sleep
```

- 调用sleep会让当前线程从Running进入Timed Waiting（也是阻塞状态）
- sleep方法要设置睡眠的时间，当时间到了，线程未必立即执行，线程会变成就绪状态，等待CPU时间片

- 其它线程可以使用interrupt方法**打断**正在sleep的线程，被打断的线程**抛出InterruptedException**
- sleep() 方法的过程中，**线程不会释放对象锁**

1.2.2、yield()

```
public class DemoYield {

    public static void main(String[] args) {
        //创建线程
        MyThread5 t01 = new MyThread5("线程01");
        MyThread5 t02 = new MyThread5("线程02");
        MyThread5 t03 = new MyThread5("线程03");

        //开启线程
        t01.start();
        t02.start();
        t03.start();
    }
}

class MyThread5 extends Thread{
    public MyThread5() {}

    public MyThread5(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            if(i ` 1){
                Thread.yield();//当循i环到30时，让线程
让步
                //1、回到抢占队列中，又争夺到了执行权
                //2、回到抢占队列中，没有争夺到执行权
            }
```

```
            System.out.println(this.getName() + ":"
+ i);
        }
    }
}
```

- 调用yield会让线程从Running进入Runnable就绪状态
- yield方法的过程中**不会释放资源锁**

1.2.3、join()

当在某个程序执行流中调用其他线程的join()方法时，**调用线程(主)将被阻塞，直到join线程执行完为止。**

```
public static void main(String[] args) throws
InterruptedException {
    Thread thread = new Thread(() -> {
        try {
            System.out.println("等3s再执行");
            Thread.sleep(3000);
            System.out.println("3s后了可以执行了");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("子线程执行了");
    });
    thread.start();
    thread.join();
    System.out.println("主线程执行了");
}

等3s再执行
3s后了可以执行了
子线程执行了
主线程执行了
```

> **底层使用了保护性暂停原理**

```
public final synchronized void join(long millis)
    throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout
value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay ≤ 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

在`join(long millis)`方法的上下文中，`isAlive()`用于实现等待机制，确保当前线程（**即执行`join`方法的线程**，而非被`join`的线程）**挂起**，直到以下两种情况之一发生：

1. 被`join`的线程（即`this`线程）执行完毕（`isAlive()`返回`false`）。
2. 指定的等待时间`millis`已过（如果提供了非零的`millis`值）。

1.2.4、interrupt()

sleep、wait、join 方法都会让线程进入阻塞状态，打断线程**会清除中断标记**（false）

- **Thread.interrupted()**检查当前线程的中断状态，并且会**清除中断标记。**
- **Thread.currentThread().isInterrupted()只检查当前线程的中断状态。。**
- **public void interrupt()**：打断线程，异常处理机制，**打断sleep、wait、join会清除中断标记，打断正常运行的线程则不会。。**

```
public static void main(String[] args) throws
InterruptedException {
    Thread t1 = new Thread(()->{
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "t1");
    t1.start();
    Thread.sleep(500);
    t1.interrupt();
    System.out.println(" 打断状态: {}" +
t1.isInterrupted());// 打断状态: {}false
}


public static void main(String[] args) throws
Exception {
    Thread t2 = new Thread(()->{
        while(true) {
            Thread current = Thread.currentThread();
            boolean interrupted =
current.isInterrupted();
            if(interrupted) {
```

```
                System.out.println(" 打断状态: {}" +
interrupted);//打断状态: {}true
                break;
            }
        }
    }, "t2");
    t2.start();
    Thread.sleep(500);
    t2.interrupt();
}
```

模拟汽车过桥

```
public static void main(String[] args) {

    Thread carTwo = new Thread(() -> {
        SmallTool.printTimeAndThread("卡丁2号 准备过
桥");
        SmallTool.printTimeAndThread("发现1号在过，开始
等待");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            SmallTool.printTimeAndThread("卡丁2号 开始
过桥");
        }
        SmallTool.printTimeAndThread("卡丁2号 过桥完
毕");
    });


    Thread carOne = new Thread(() -> {
        SmallTool.printTimeAndThread("卡丁1号 开始过
桥");
        int timeSpend = new Random().nextInt(500) +
1000;
```

```
            SmallTool.sleepMillis(timeSpend);
            SmallTool.printTimeAndThread("卡丁1号 过桥完毕
耗时:" + timeSpend);
            //
SmallTool.printTimeAndThread("卡丁2号的状态" +
carTwo.getState());
            carTwo.interrupt();
        });

        carOne.start();
        carTwo.start();
    }

1708346109387   |   21  |   Thread-1    |   卡丁1号
开始过桥
1708346109387   |   20  |   Thread-0    |   卡丁2号
准备过桥
1708346109387   |   20  |   Thread-0    |   发现1号在
过，开始等待
1708346110560   |   21  |   Thread-1    |   卡丁1号
过桥完毕 耗时:1173
1708346110560   |   20  |   Thread-0    |   卡丁2号
开始过桥
1708346110560   |   20  |   Thread-0    |   卡丁2号
过桥完毕
```

线程carTwo的执行流程

1. 打印“卡丁2号 准备过桥”和“发现1号在过，开始等待”。
2. 尝试执行`Thread.sleep(3000)`，模拟等待过程。
3. 如果在睡眠期间被中断（这是由另一个线程`carOne`触发的），则捕获到`InterruptedException`异常，随后打印“卡丁2号 开始过桥”。
4. 不管是否被中断，都会继续执行打印“卡丁2号 过桥完毕”。

> **线程carOne的执行流程**

1. 打印“卡丁1号 开始过桥”。
2. 随机生成一个介于1000到1500毫秒之间的睡眠时间，模拟卡丁1号过桥的时间，并睡眠相应的时间。
3. 睡眠结束后，打印“卡丁1号 过桥完毕 耗时:”和实际耗时。
4. 调用carTwo.interrupt()，尝试中断carTwo线程，即卡丁2号。

- **Thread.sleep方法在被中断时会清除中断标志，并抛出InterruptedException。**
- **在carTwo的catch块中，直接处理了中断情况，模拟了卡丁2号接收到中断信号后立即开始过桥的行为。**

1.2.5、run()与start()

> start()

通过该方法启动线程的同时也**创建了一个线程**，真正实现了多线程。无需等待run()方法中的代码执行完毕，就可以接着执行下面的代码。此时start()的这个线程处于就绪状态，**当得到CPU的时间片后就会执行其中的run()方法**。这个run()方法包含了要执行的这个线程的内容，run()方法运行结束，此线程也就终止了。

> run方法()

通过run方法启动线程其实就是调用一个类中的方法，当作普通的方法的方式调用。**并没有创建一个线程，程序中依旧只有一个主线程，必须等到run()方法里面的代码执行完毕，才会继续执行主线程下面的代码。**

2、共享模型之内存

JMM 即 Java Memory Model，它定义了主存、工作内存抽象概念，底层对应着 CPU 寄存器、缓存、硬件内存、CPU 指令优化等。

JMM 体现在以下几个方面

- 原子性 - 保证一个操作或者多个操作，**要么全部执行，要么都不执行**

- 可见性 - 保证多个线程访问一个资源时，该资源的状态、值信息等**对于其他线程都是可见的**
- 有序性 - 保证程序执行的顺序**按照代码先后执行**

2.1、可见性测试

2.1.1、两阶段终止

停止标记用 **volatile** 是为了保证该变量在多个线程之间的**可见性**。

```
// 停止标记用 volatile 是为了保证该变量在多个线程之间的可见性
// 我们的例子中，即主线程把它修改为 true 对 t1 线程可见
public class TPTVolatile {

    private Thread thread;
    private volatile boolean stop = false;

    public static void main(String[] args) throws InterruptedException {
        TPTVolatile tptVolatile = new TPTVolatile();
        tptVolatile.start();
        Thread.sleep(1500);
        tptVolatile.stop();
    }

    public void start(){
        thread = new Thread(() -> {
            while(true) {
                //Thread current = Thread.currentThread();
                if(stop) {
                    System.out.println(("料理后事"));
                    break;
                }
                try {
```

```
                    Thread.sleep(1000);
                    System.out.println(("将结果保
存"));
                } catch (InterruptedException e) {
                }
                // 执行监控操作
            }
        },"监控线程");
        thread.start();
    }

    public void stop() {
        stop = true;
        // 设置，可以让睡着稍微线程醒来，不然线程只有循环时
才会判断stop
        thread.interrupt();
    }
}
```

2.1.2、犹豫模式

确保了在任何时刻**只有一个线程**可以进入启动逻辑的关键部分。

synchronized+volatile实现单例

```
public class MonitorService {

    // 用来表示是否已经有线程已经在执行启动了
    private volatile boolean starting;

    public void start() {
        log.info("尝试启动监控线程...");
        synchronized (this) {
            if (starting) {
                return;
            }
            starting = true;
        }
```

```
            // 真正启动监控线程...
        }
    }
```

2.2、有序性测试

> Clock Cycle Time 时钟周期时间

主频的概念大家接触的比较多，而 CPU 的 Clock Cycle Time（时钟周期时间），等于**主频的倒数，意思是 CPU 能够识别的最小时间单位**，比如说 4G 主频的 CPU 的 Clock Cycle Time 就是 0.25 ns，作为对比，我们墙上挂钟的Cycle Time 是 1s

例如，运行一条加法指令一般需要一个时钟周期时间

> CPI 平均时钟周期数

有的指令需要更多的时钟周期时间，所以引出了 CPI （Cycles Per Instruction）**指令平均时钟周期数**

> IPC 即 CPI 的倒数

IPC（Instruction Per Clock Cycle）**即 CPI 的倒数，表示每个时钟周期能够运行的指令数**

> CPU 执行时间

程序的 CPU 执行时间，即我们前面提到的 user + system 时间，可以用下面的公式来表示

程序 CPU 执行时间 = 指令数 * CPI * Clock Cycle Time

2.3、volatile原理

volatile 的底层实现原理是内存屏障，Memory Barrier（Memory Fence）

- **对 volatile 变量的写指令后会加入写屏障**
- **对 volatile 变量的读指令前会加入读屏障**

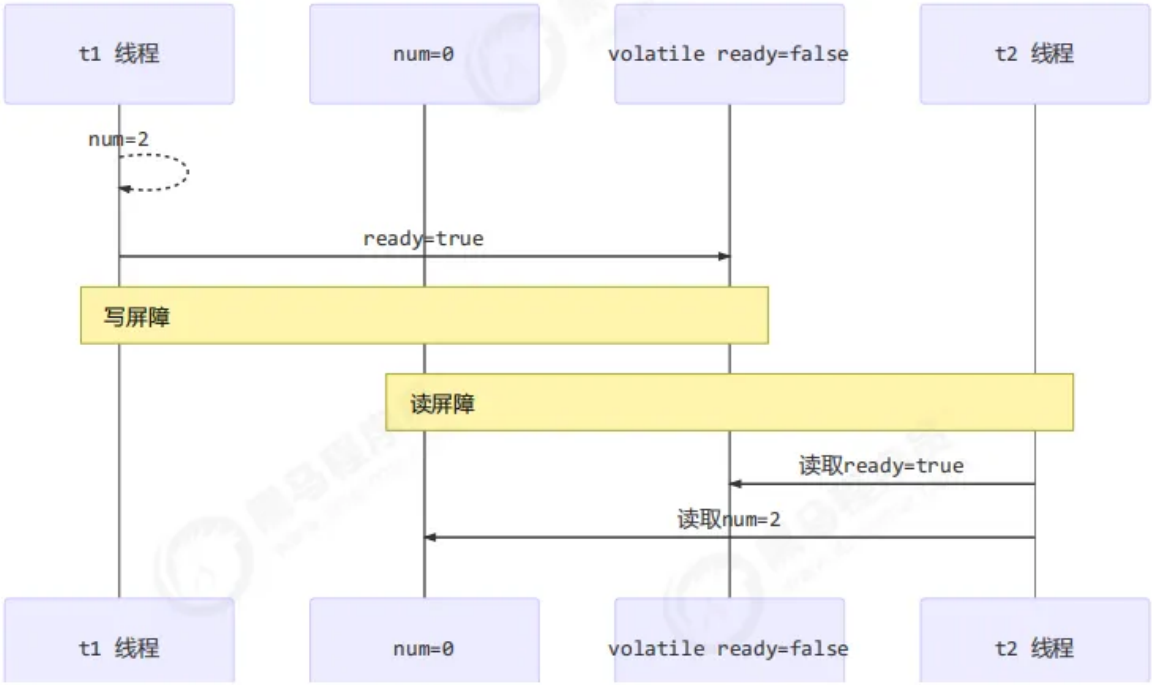
2.3.1、如何保证可见性

写屏障（sfence）保证在该屏障之前的，对所有共享变量(不管有没有加volatile)的改动，都同步到主存当中。

```
public void actor2(I_Result r) {
    num = 2;
    ready = true; // ready 是 volatile 赋值带写屏障
    // 写屏障
}
```

读屏障（lfence）保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据。

```
public void actor1(I_Result r) {
    // 读屏障
    // ready 是 volatile 读取值带读屏障
    if(ready) {
        r.r1 = num + num;
    } else {
        r.r1 = 1;
    }
}
```



2.3.2、如何保证可见性

写屏障会确保指令重排序时，**不会将写屏障之前的代码排在写屏障之后**

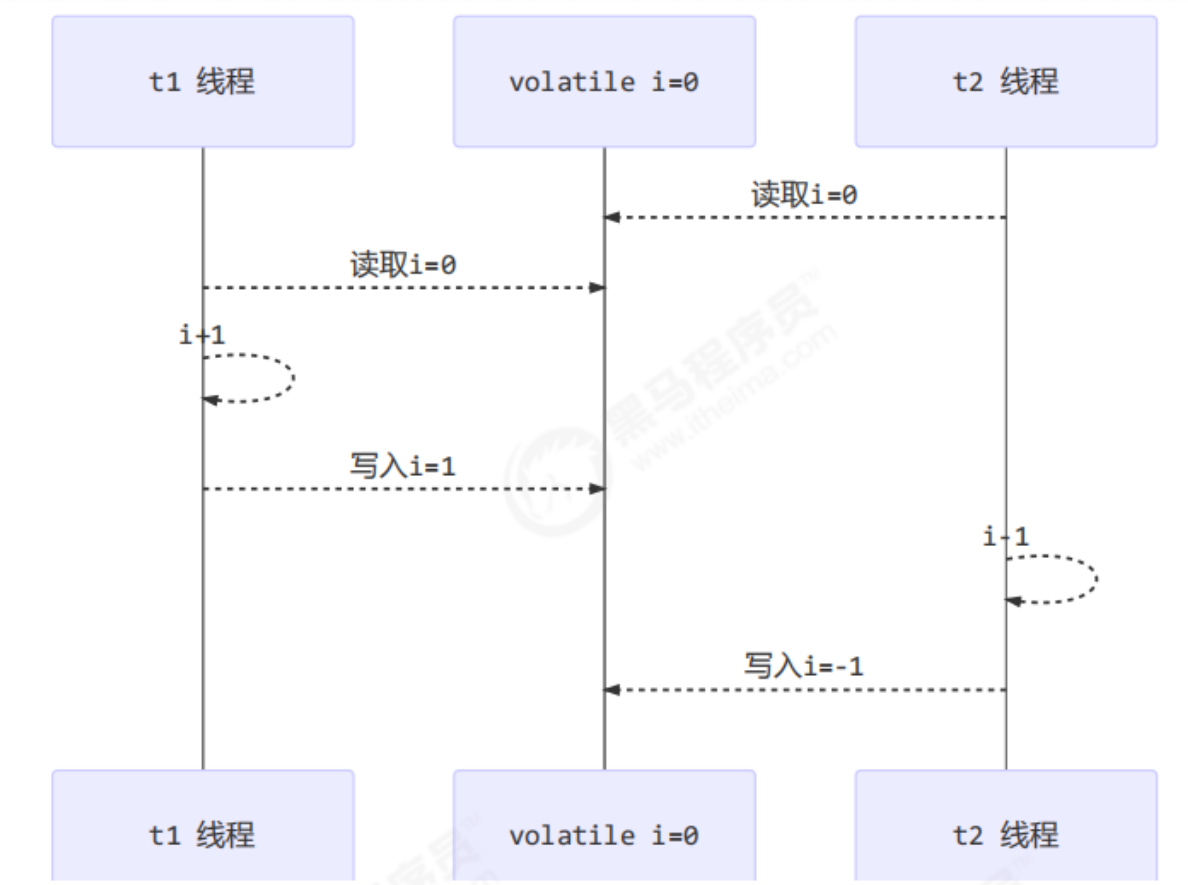
```
public void actor2(I_Result r) {
    num = 2;
    ready = true; // ready 是 volatile 赋值带写屏障
    // 写屏障
}
```

读屏障会确保指令重排序时，**不会将读屏障之后的代码排在读屏障之前**

```
public void actor1(I_Result r) {
    // 读屏障
    // ready 是 volatile 读取值带读屏障
    if(ready) {
        r.r1 = num + num;
    } else {
        r.r1 = 1;
    }
}
```

还是那句话，**不能解决指令交错**：

- 写屏障仅仅是保证之后的读能够读到最新的结果，但不能保证读跑到它前面去
- 而**有序性**的保证也只是**保证了本线程内相关代码不被重排序**



2.4、应用-单例双锁

```
public final class Singleton {
    private Singleton() { }
    private static volatile Singleton INSTANCE =
null;

    public static Singleton getInstance() {
        // 实例没创建，才会进入内部的 synchronized代码块
        if (INSTANCE == null) {
            synchronized (Singleton.class) { // t2
                // 也许有其它线程已经创建实例，所以再判断一
次
                if (INSTANCE == null) { // t1
                    INSTANCE = new Singleton();
```

```
                }
            }
        }
        return INSTANCE;
    }
}
```

- **第一次检查（外部检查）**：首先检查`INSTANCE`是否已经被初始化，以**避免不必要的同步开销**。
- **第二次检查（内部检查）**：即使线程进入了同步块，也需要再次检查`INSTANCE`是否为`null`。这是因为当第一个进入同步块的线程创建了实例后，等待的其他线程仍然会进入同步块（但是一个接一个），所以需要再次检查以**避免重复创建实例**。

> 指令重排序问题-volatile

在`INSTANCE = new Singleton();`这行代码中，实际上包含了三个主要操作：

1. **为`Singleton`对象分配内存空间。**
2. **调用`Singleton`的构造函数，初始化成员变量。**
3. **将`INSTANCE`引用指向分配的内存区域。**

如果没有`volatile`关键字，JVM和编译器出于性能优化的考虑，可能会对这些操作进行重排序。特别是，**操作3可能在操作2之前发生，即先将`INSTANCE`引用指向分配的内存（此时对象尚未完全构造），再进行对象的初始**化。

如果发生了上述的重排序，考虑以下场景：

- **线程A**执行`Singleton`的`getInstance`方法，并进入`INSTANCE = new Singleton();`的执行过程。由于指令重排序，它可能在对象完全构造之前就已经将`INSTANCE`的引用设置为非`null`了。
- **线程B**也调用`Singleton`的`getInstance`方法，**进行第一次的`null`检查时发现`INSTANCE`已经不是`null`了**（因为线程A已经将其设置为非`null`，尽管对象可能尚未完全初始化）。因此，**线程B可能会直接返回这个尚未完全初始化的`Singleton`实例并使用它。**

3、线程的安全

局部变量永远都不会存在线程安全问题。因为局部变量不共享。局部变量在栈中。所以局部变量永远都不会共享。实例变量在堆中，堆只有1个。静态变量在方法区中，方法区只有1个。堆和方法区都是多线程共享的，所以可能存在线程安全问题。

3.1、线程同步的利弊

- 好处：解决了线程同步的数据安全问题
- 弊端：当线程很多的时候，每个线程都会去判断同步上面的这个锁，很耗费资源，降低效率

3.2、synchronized使用

3.2.1、synchronized的作用

`synchronized` 的作用主要有三：

- **原子性**：**所谓原子性就是指一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。**被`synchronized`修饰的类或对象的所有操作都是原子的，因为在执行操作之前必须先获得类或对象的锁，直到执行完才能释放。
- **可见性**：**可见性是指多个线程访问一个资源时，该资源的状态、值信息等对于其他线程都是可见的。** synchronized和volatile都具有可见性，其中synchronized对一个类或对象加锁时，一个线程如果要访问该类或对象必须先获得它的锁，而这个锁的状态对于其他任何线程都是可见的，并且在释放锁之前会将对变量的修改刷新到共享内存当中，保证资源变量的可见性。
- **有序性**：**有序性值程序执行的顺序按照代码先后执行。**synchronized和volatile都具有有序性，Java允许编译器和处理器对指令进行重排，但是指令重排并不会影响单线程的顺序，它影响的是多线程并发执行的顺序性。synchronized保证了每个时刻都只有一个线程访问同步代码块，也就确定了线程执行同步代码块是分先后顺序的，保证了有序性。

3.2.2、synchronized的使用

> 修饰一个代码块

修饰代码块： 指定加锁对象，对给定对象/类加锁。
`synchronized(this|object)` 表示进入同步代码库前要获得**对象实例的锁。** **`synchronized(类.class)`** 表示进入同步代码前要获得 **Java类锁**

```
synchronized(this) {
  //业务代码
}
```

```
public class Demo04_synchronized_codeBlock {

    public static void main(String[] args) throws
InterruptedException {
        System.out.println("使用关键字synchronized");
        SyncThread syncThread = new SyncThread();
        Thread thread1 = new Thread(syncThread,
"SyncThread1");
        Thread thread2 = new Thread(syncThread,
"SyncThread2");
        thread1.start();
        thread2.start();
        thread1.join(); // 主线程等待1完成
        thread2.join(); // 主线程等待2完成
    }
}

class SyncThread implements Runnable {

    private static int count;

    public SyncThread() {
        count = 0;
    }
```

```
    public void run() {
        // 对象实例锁
        synchronized (this){
            for (int i = 0; i < 5; i++) {
                try {
                    System.out.println("线程
名:"+Thread.currentThread().getName() + ":" +
(count++));
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public int getCount() {
        return count;
    }
}
```

> 修饰实例方法

修饰实例方法： 作用于当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁**。

```
public synchronized void run() {
    for (int i = 0; i < 5; i++) {
        try {
            System.out.println("线程
名:"+Thread.currentThread().getName() + ":" +
(count++));
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

> 修饰静态方法

修饰静态方法： 也就是给**当前类加锁**，会作用于类的所有对象实例 ，进入同步代码前要获得 **当前类锁**。

```
synchronized void staic method() {
  //业务代码
}
```

3.3、synchronized的原理

> Mark Word

Mark Word是对象头的一部分，用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID等信息。

Mark Word 结构：最后两位是**锁标志位。**

```
|-------------------------------------------------------|--------------------|
|                 Mark Word (32 bits)                   |       State        |
|-------------------------------------------------------|--------------------|
|  hashcode:25          | age:4 | biased_lock:0 | 01    |       Normal       |
|-------------------------------------------------------|--------------------|
|  thread:23 | epoch:2 | age:4 | biased_lock:1 | 01    |       Biased       |
|-------------------------------------------------------|--------------------|
|              ptr_to_lock_record:30          | 00    | Lightweight Locked |
|-------------------------------------------------------|--------------------|
|              ptr_to_heavyweight_monitor:30  | 10    | Heavyweight Locked |
|-------------------------------------------------------|--------------------|
|                                             | 11    |    Marked for GC   |
|-------------------------------------------------------|--------------------|
```

- 无锁（匿名偏向）状态：
  - 25位的哈希码
  - 4位的GC分代年龄
  - 2位的锁状态标志
- 偏向锁状态：
  - 23位的线程ID
  - 2位的epoch
  - 4位的GC分代年龄
  - 1位的偏向锁标志
  - 2位的锁状态标志
- 轻量级锁状态：
  - 30位的指针到锁记录
  - 2位的锁状态标志
- 重量级锁状态：
  - 30位的指针到监视器
  - 2位的锁状态标志

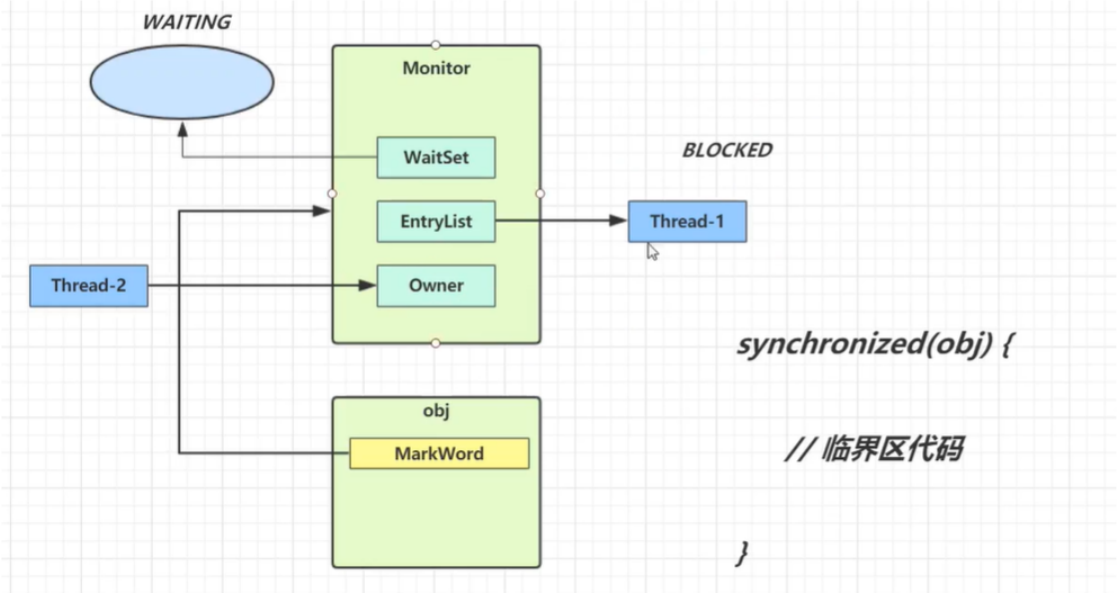
> Monitor

每个 Java 对象都可以关联一个 Monitor 对象。

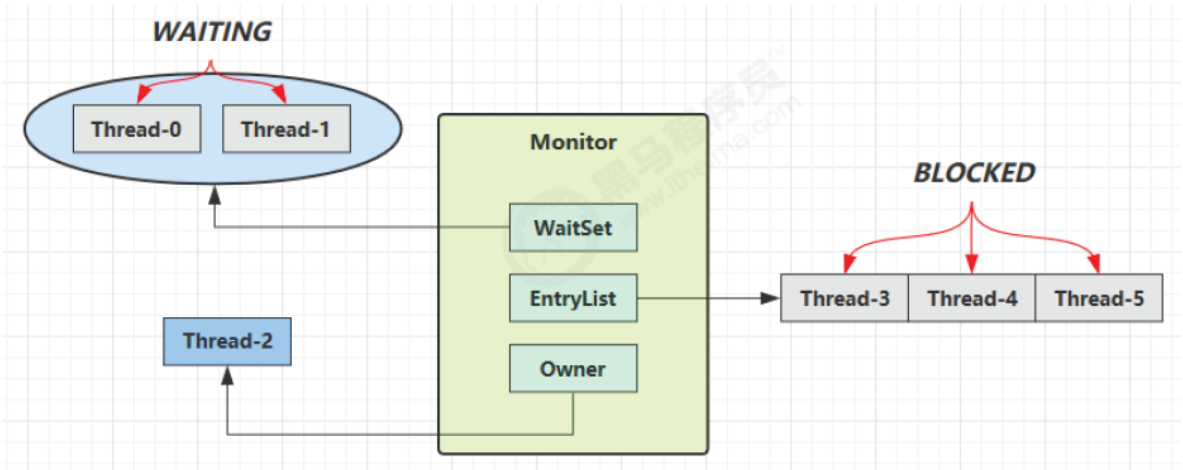
工作流程：

- 开始时 Monitor 中 Owner 为 null
- 当 Thread-2 执行 synchronized(obj) 就会将 Monitor 的所有者 Owner 置为 Thread-2，Monitor 中只能有一个 Owner，

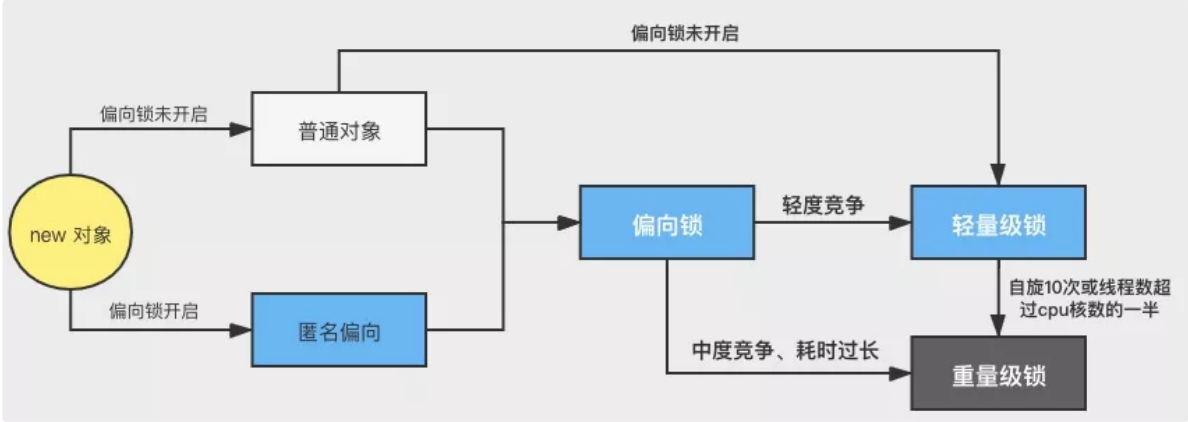
obj 对象的 Mark Word 指向 Monitor，把**对象原有的 MarkWord 存入线程栈中的锁记录**中。



- 在 Thread-2 上锁的过程，Thread-3、Thread-4、Thread-5 也执行 synchronized(obj)，就会进入 **EntryList BLOCKED**（双向链表）
- Thread-2 执行完同步代码块的内容，根据 obj 对象头中 Monitor 地址寻找，设置 Owner 为空，**把线程栈的锁记录中的对象头的值设置回Obj的 MarkWord**
- **唤醒 EntryList 中等待的线程来竞争锁，竞争是非公平的**，如果这时有新的线程想要获取锁，可能直接就抢占到了，阻塞队列的线程就会继续阻塞
- **WaitSet 中的 Thread-0，是以前获得过锁，但条件不满足进入 WAITING 状态的线程**



3.4、synchronized锁升级



（不涉及Monitor的）偏向锁

偏向锁的思想是偏向于让第一个获取锁对象的线程，这个线程之后重新获取该锁不再需要同步操作：

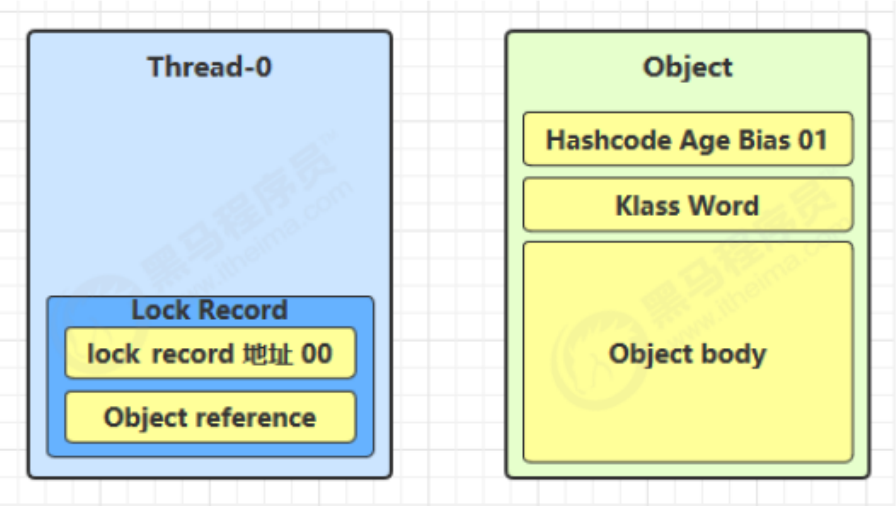
- 当锁对象第一次被线程获得的时候进入偏向状态，标记为 `101`，同时**使用 CAS 操作将线程 ID 记录到 Mark Word。**如果 CAS 操作成功，这个线程以后进入这个锁相关的同步块，查看这个线程 ID 是自己的就表示没有竞争，就不需要再进行任何同步操作
- 当有另外一个线程去尝试获取这个锁对象时，偏向状态就宣告结束，此时撤销偏向（Revoke Bias）后**恢复到未锁定或轻量级锁状态**

Mark Word (64 bits)					State
unused:25	hashcode:31	unused:1	age:4	biased_lock:0	01	Normal
thread:54	epoch:2	unused:1	age:4	biased_lock:1	01	Biased
ptr_to_lock_record:62					00	Lightweight Locked
ptr_to_heavyweight_monitor:62					10	Heavyweight Locked
					11	Marked for GC

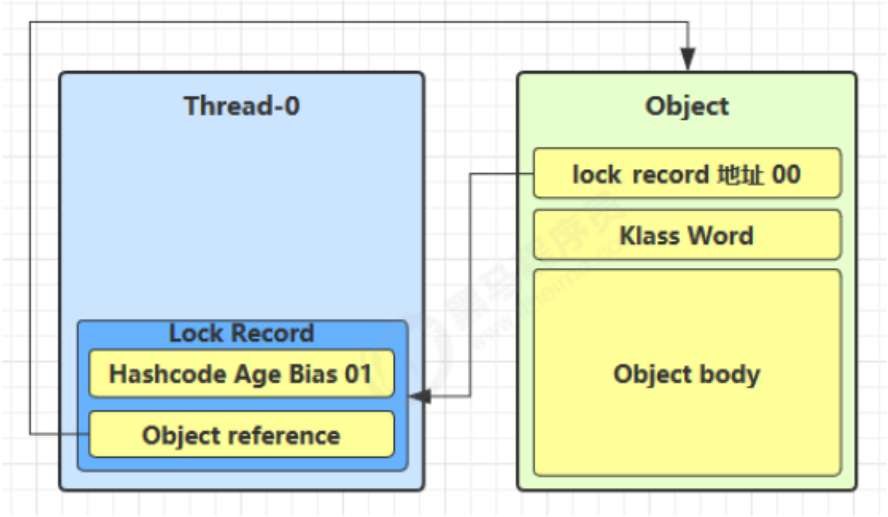
（不涉及Monitor的）轻量级锁

一个对象有**多个线程要加锁，但加锁的时间是错开的（没有竞争）**，可以使用轻量级锁来优化。

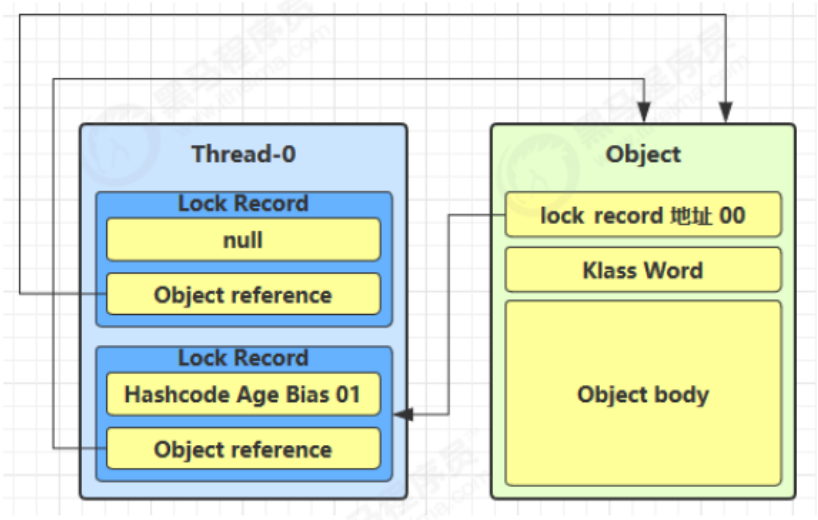
- 创建锁记录（Lock Record）对象，每个线程的**栈帧**都会包含一个锁记录的结构，存储锁定对象的 Mark Word



- 让锁记录中 Object reference 指向锁住的对象，并尝试**用 CAS 将 Object 的 Mark Word和 lock record 的锁记录地址和状态 00（轻量级锁）进行交换**



- 如果 CAS 失败，有两种情况：
  - 如果是其它线程已经持有了该 Object 的轻量级锁，这时表明有竞争，进入**锁膨胀**过程
  - 如果是线程自己执行了 synchronized **锁重入**，就添加一条 Lock Record 作为重入的计数

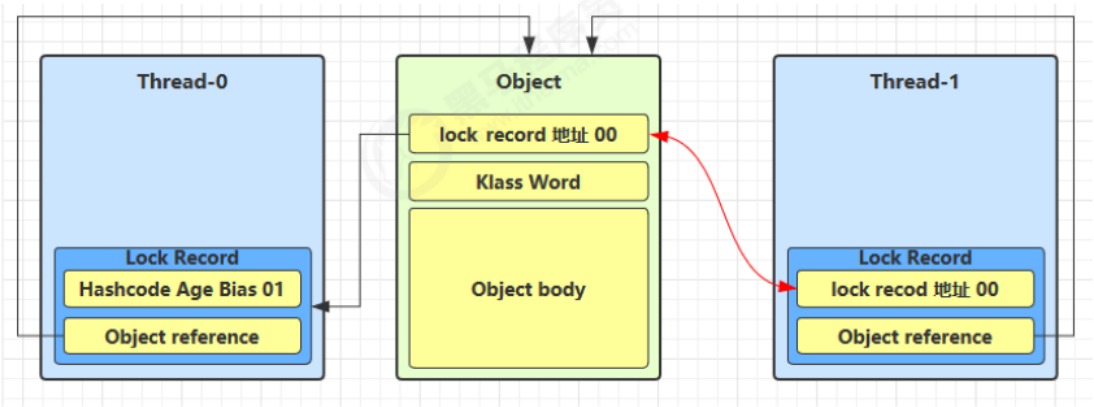


- 当退出 synchronized 代码块（解锁时）
  - 如果有取值为 null 的锁记录，表示有**重入**，这时重置锁记录，表示重入计数减 1
  - 如果锁记录的值不为 null，这时使用 CAS **将 Mark Word 的值恢复给对象头**
    - 成功，则解锁成功
    - 失败，说明轻量级锁进行了锁膨胀或已经升级为重量级锁，进入重量级锁解锁流程

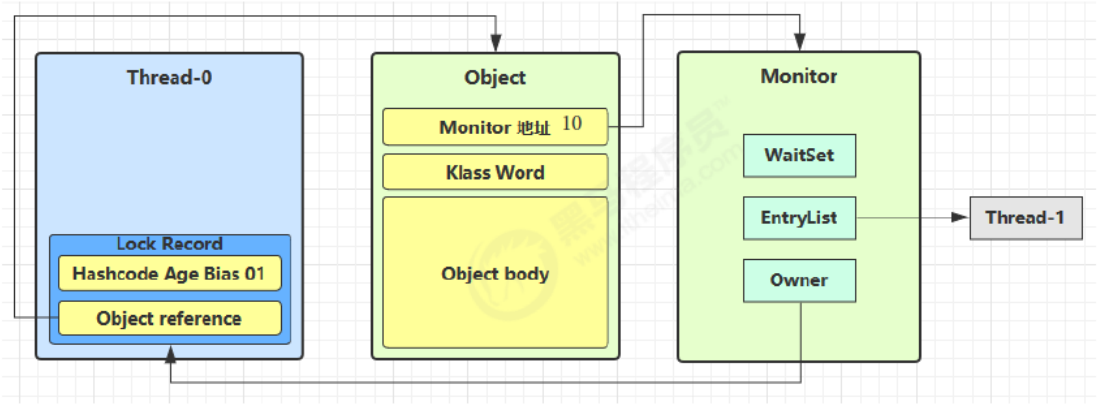
> 锁膨胀以及重量级锁

在尝试加轻量级锁的过程中，CAS 操作无法成功，可能是其它线程为此对象加上了轻量级锁（有竞争），这时需要进行锁膨胀，将轻量级锁变为**重量级锁**

- 当 Thread-1 进行轻量级加锁时，Thread-0 已经对该对象加了轻量级锁



- Thread-1 加轻量级锁失败，进入锁膨胀流程：为 Object 对象申请 Monitor 锁，**通过 Object 对象头获取到持锁线程**，将 Monitor 的 Owner 置为 Thread-0，将 Object 的对象头指向重量级锁地址，然后自己进入 Monitor 的 EntryList BLOCKED



- 当 Thread-0 退出同步块解锁时，使用 CAS 将 Mark Word 的值恢复给对象头失败，这时进入重量级解锁流程，即**按照 Monitor 地址找到 Monitor 对象，设置 Owner 为 null，唤醒 EntryList 中 BLOCKED 线程**

> 自旋锁

重量级锁竞争时，尝试获取锁的线程不会立即阻塞，可以使用**自旋**（默认 10 次）来进行优化，采用循环的方式去尝试获取锁

注意：

- 自旋占用 CPU 时间，单核 CPU 自旋就是浪费时间，因为同一时刻只能运行一个线程，多核 CPU 自旋才能发挥优势
- 自旋失败的线程会进入阻塞状态

优点：不会进入阻塞状态，**减少线程上下文切换的消耗**

缺点：当自旋的线程越来越多时，会不断的消耗 CPU 资源

现代Java版本中，`ThreadLocal`的实现方式发生了变化。每个`Thread`对象内部都有一个`ThreadLocal.ThreadLocalMap`，这个`Map`专门用于存储该线程的所有`ThreadLocal`变量。在这个`Map`中，每个`ThreadLocal`实例是键，而对应的线程特有的数据则作为值。这种设计允许每个线程高效地访问其线程局部变量，而无需全局的同步，同时也避免了内存泄漏问题。

3.5、ThreadLocal

假设我们有一个`ThreadLocal<Integer>`用来存储每个线程的用户ID：

```
public class UserSession {
    private static final ThreadLocal<Integer> userId
= new ThreadLocal<>();

    public static void setUserId(int id) {
        userId.set(id);
    }

    public static Integer getUserId() {
        return userId.get();
    }
}
```

- 当线程A调用`UserSession.setUserId(1);`时，它实际上是**在线程A的`ThreadLocalMap`中添加一个条目，其中键是`userId`（`ThreadLocal`实例），值是`1`。**
- 当线程B调用`UserSession.setUserId(2);`时，**同样的`ThreadLocal`实例（即`userId`）被用作键，但这次是在线程B的`ThreadLocalMap`中添加条目，值是`2`。**

因此，尽管线程A和线程B都使用相同的`ThreadLocal`对象，它们各自能够存取自己独有的数据，互不干扰。这就是现代`ThreadLocal`通过每个线程拥有自己的`ThreadLocalMap`来实现线程局部存储的机制。

4、wait/notify

需要获取对象锁后才可以调用 `锁对象.wait()`，**notify 随机唤醒一个线程，notifyAll 唤醒所有线程去竞争 CPU。**

Object 类 API：

```
public final void notify():唤醒正在等待对象监视器的单个线程。
public final void notifyAll():唤醒正在等待对象监视器的所有线程。
public final void wait():导致当前线程等待，直到另一个线程调用该对象的 notify() 方法或 notifyAll()方法。
public final native void wait(long timeout):有时限的等待，到n毫秒后结束等待，或是被唤醒
```

说明：**wait 是挂起线程，需要唤醒的都是挂起操作**，阻塞线程可以自己去争抢锁，挂起的线程需要唤醒后去争抢锁

对比 sleep()：

- 原理不同：sleep() 方法是属于 Thread 类，是线程用来控制自身流程的，使此线程暂停执行一段时间而把执行机会让给其他线程；wait() 方法属于 Object 类，用于线程间通信
- 对**锁的处理机制**不同：调用 sleep() 方法的过程中，线程不会释放对象锁，当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池（不释放锁其他线程怎么抢占到锁执行唤醒操作），但是都会释放 CPU
- 使用区域不同：wait() 方法必须放在**同步控制方法和同步代码块（先获取锁）**中使用，sleep() 方法则可以放在任何地方使用

wait()和notify()方法的底层实现是依赖于操作系统的原生线程调度和同步机制。Java虚拟机（JVM）通过调用底层操作系统提供的同步原语（如互斥锁、条件变量等）来实现这些方法。

4.1、（同步）设计-保护性暂停

即 Guarded Suspension，用在一个线程等待另一个线程的执行结果。

- 有一个结果需要从一个线程传递到另一个线程，让他们关联同一个 GuardedObject
- 如果有结果不断从一个线程到另一个线程那么可以使用消息队列（见生产者/消费者）
- **JDK 中，join 的实现、Future 的实现，采用的就是此模式**

> 单任务

当前线程在调用 `lock.wait(waitTime)` 时被其他线程调用了 `lock.notify()` 或 `lock.notifyAll()`，从而提前被唤醒。

```
class GuardedObjectV2 {
    private Object response;
    private final Object lock = new Object();

    public Object get(long millis) {
        synchronized (lock) {
            // 1) 记录最初时间
            long begin = System.currentTimeMillis();
            // 2) 已经经历的时间
            long timePassed = 0;

            while (response == null) {
                // 4) 假设 millis 是 1000，结果在 400
时唤醒了，那么还有 600 要等
                long waitTime = millis - timePassed;
                log.debug("waitTime: {}", waitTime);

                if (waitTime <= 0) {
                    log.debug("break...");
                    break;
                }

                try {
                    lock.wait(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // 提前被唤醒
                timePassed =
System.currentTimeMillis() - begin;
```

```
                log.debug("timePassed: {}, object is
null {}",
                          timePassed, response ==
null);
            }
            return response;
        }
    }
    public void complete(Object response) {
        synchronized (lock) {
            // 条件满足，通知等待线程
            this.response = response;
            log.debug("notify...");
            lock.notifyAll();
        }
    }
}
```

一个线程(主)等待另一个线程(Thread)的执行结果

```
public static void main(String[] args) {
    GuardedObjectV2 v2 = new GuardedObjectV2();

    new Thread(() -> {
        sleep(1);
        v2.complete(null);
        sleep(1);
        v2.complete(Arrays.asList("a", "b", "c"));
    }).start();

    Object response = v2.get(2500);
    if (response != null) {
        log.debug("get response: [{}] lines",
((List<String>) response).size());
    } else {
        log.debug("can't get response");
    }
```

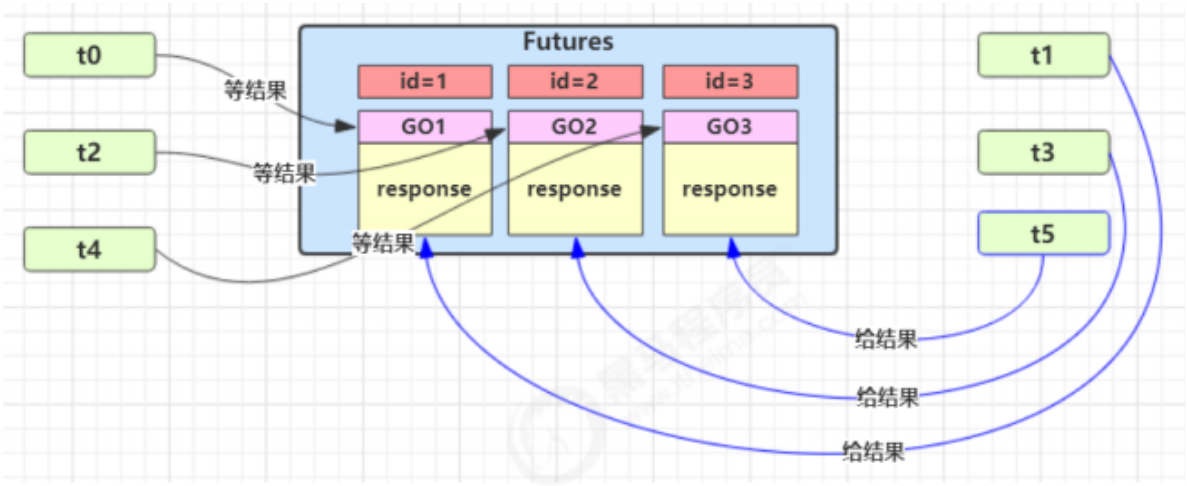
```
}
```

> 多任务

图中 Futures 就好比居民楼一层的信箱（每个信箱有房间编号），左侧的 t0，t2，t4 就好比等待邮件的居民，右 侧的 t1，t3，t5 就好比邮递员。

如果需要在多个类之间使用 GuardedObject 对象，作为参数传递不是很方便，因此设计一个用来解耦的中间类， 这样不仅能够解耦【结果等待者】和【结果生产者】，还能够同时支持多个任务的管理。

多个一对一，区别于生产者，消费者



新增 id 用来标识 Guarded Object

```
class GuardedObject {
    // 标识 Guarded Object
    private int id;
    public GuardedObject(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
```

```
    // 结果
    private Object response;
    // 获取结果
    // timeout 表示要等待多久 2000
    public Object get(long timeout) {
        synchronized (this) {
            // 开始时间 15:00:00
            long begin = System.currentTimeMillis();
            // 经历的时间
            long passedTime = 0;
            while (response == null) {
                // 这一轮循环应该等待的时间
                long waitTime = timeout -
passedTime;
                // 经历的时间超过了最大等待时间时，退出循环
                if (timeout - passedTime ≤ 0) {
                    break;
                }
                try {
                    this.wait(waitTime); // 虚假唤醒
15:00:01
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 求得经历时间
                passedTime =
System.currentTimeMillis() - begin; // 15:00:02 1s
            }
            return response;
        }
    }
    // 产生结果
    public void complete(Object response) {
        synchronized (this) {
            // 给结果成员变量赋值
            this.response = response;
            this.notifyAll();
```

```
        }
    }
}
```

中间解耦类+业务相关类

```
class Mailboxes {
    private static Map<Integer, GuardedObject> boxes
= new Hashtable<>();

    private static int id = 1;
    // 产生唯一 id
    private static synchronized int generateId() {
        return id++;
    }

    public static GuardedObject getGuardedObject(int
id) {
        return boxes.remove(id);
    }

    public static GuardedObject
createGuardedObject() {
        GuardedObject go = new
GuardedObject(generateId());
        boxes.put(go.getId(), go);
        return go;
    }

    public static Set<Integer> getIds() {
        return boxes.keySet();
    }
}

class People extends Thread{
    @Override
    public void run() {
```

```
            // 收信
            GuardedObject guardedObject =
Mailboxes.createGuardedObject();
            log.debug("开始收信 id:{}",
guardedObject.getId());
            Object mail = guardedObject.get(5000);
            log.debug("收到信 id:{}, 内容:{}",
guardedObject.getId(), mail);
        }
    }


    class Postman extends Thread {
        private int id;
        private String mail;
        public Postman(int id, String mail) {
            this.id = id;
            this.mail = mail;
        }
        @Override
        public void run() {
            GuardedObject guardedObject =
Mailboxes.getGuardedObject(id);
            log.debug("送信 id:{}, 内容:{}", id, mail);
            guardedObject.complete(mail);
        }
    }
```

测试类

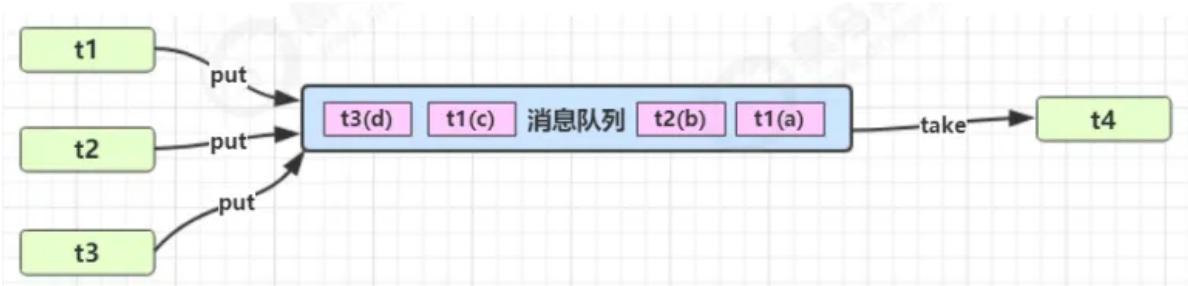
1s后开始送信

```
public static void main(String[] args) throws
InterruptedException {
    for (int i = 0; i < 3; i++) {
        new People().start();
    }
    Sleeper.sleep(1);
    for (Integer id : Mailboxes.getIds()) {
        new Postman(id, "内容" + id).start();
    }
}
```

4.2、（异步）设计-生产者消费者模式

要点

- 消费队列可以用来平衡生产和消费的线程资源
- 生产者仅负责产生结果数据，不关心数据该如何处理，而消费者专心处理结果数据
- **消息队列是有容量限制的**，满时不会再加入数据，空时不会再消耗数据
- JDK 中各种**阻塞队列**，采用的就是这种模式



> 实体类

一个id,一个消息对象

```
public class Message {

    private int id;
    private Object message;
    public Message(int id, Object message) {
```

```
        this.id = id;
        this.message = message;
    }
    public int getId() {
        return id;
    }
    public Object getMessage() {
        return message;
    }
}
```

> 消息队列类

容量+阻塞队列

```
public class MessageQueue {

    private LinkedList<Message> queue;
    private int capacity;

    public MessageQueue(int capacity) {
        this.capacity = capacity;
        queue = new LinkedList<>();
    }

    public Message take() {
        synchronized (queue) {
            while (queue.isEmpty()) {
                System.out.println(("没货了, wait"));
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            Message message = queue.removeFirst();
            queue.notifyAll();
```

```
            return message;
        }
    }

    public void put(Message message) {
        synchronized (queue) {
            while (queue.size() == capacity) {
                System.out.println(("库存已达上限,
wait"));
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            queue.addLast(message);
            System.out.println(("成功放入message" +
message.getId()));
            queue.notifyAll();
        }
    }
}
```

测试类

```
public static void main(String[] args) {
    MessageQueue messageQueue = new MessageQueue(2);

    // 4 个生产者线程, 下载任务
    for (int i = 0; i < 4; i++) {
        int id = i;
        new Thread(() -> {
            System.out.println(("download..."));
            String response = download();
            System.out.println(("try put
message({})" + id));
```

```
            messageQueue.put(new Message(id,
response));
        }, "生产者" + i).start();
    }

    // 1 个消费者线程, 处理结果
    new Thread(() -> {
        while (true) {
            Message message = messageQueue.take();
            String response = (String)
message.getMessage();
            System.out.println(("take message({}):
[{}] lines" + message.getId() + response));;
        }
    }, "消费者").start();
```

5、park与unpark

unpark专门用于控制线程的挂起和继续执行，而interrupt是线程中断机制的一部分，除了能唤醒park的线程外，还可以中断线程的阻塞状态（在sleep、wait、join等操作中）。**使用interrupt唤醒park线程时，通常需要在线程内部处理中断逻辑，但是counter不变。**

LockSupport 是用来创建锁和其他同步类的**线程原语**

LockSupport 类方法：

- LockSupport.park()：暂停当前线程，挂起原语
- LockSupport.unpark(thread)：恢复某个线程的运行
- **先 park 再 unpark 和先 unpark 再 park 效果一样，都会直接恢复线程的运行**

```
public static void main(String[] args) {
    Thread t1 = new Thread(() -> {
        System.out.println("start..."); //1
        Thread.sleep(1000);// Thread.sleep(3000)
```

```
        // 先 park 再 unpark 和先 unpark 再 park 效果一样，都会直接恢复线程的运行
        System.out.println("park...");  //2
        LockSupport.park();
        System.out.println("resume...");//4
    },"t1");
    t1.start();
    Thread.sleep(2000);
    System.out.println("unpark...");    //3
    LockSupport.unpark(t1);
}
```

> 原理

`LockSupport.park()`和`LockSupport.park(Object blocker)`都是用来**挂起当前线程**的，但`park(Object blocker)`版本在挂起前后做了额外的操作：它记录了一个“blocker”对象，**记录导致线程挂起的资源或活动。**

park方法

所以，先多次unpark,只需要一个park就可以抵消。

1. 获取当前线程关联的 Parker 对象。
2. **将计数器置为 0，同时检查计数器的原值是否为 1，如果是则放弃后续操作。**
3. 在互斥量上加锁。
4. 在条件变量上阻塞，同时释放锁并等待被其他线程唤醒，当被唤醒后，将重新获取锁。
5. **当线程恢复至运行状态后，将计数器的值再次置为 0。**
6. 释放锁。

unpark方法

该方法做的事情相对简单，它先是给当前线程加锁，然后**将计数器的值改为 1，接着判断 Parker 对象所关联的线程是否被 park**，如果是，则通过 `pthread_mutex_signal` 函数唤醒该线程，最后释放锁。

1. 获取目标线程关联的 Parker 对象（注意目标线程不是当前线程）。

2. 在互斥量上加锁。
3. **将计数器置为 1。**
4. **唤醒在条件变量上等待着的线程。**
5. 释放锁。

```
public static void main(String[] args) {
    Thread t1 = new Thread(() -> {
        System.out.println("start..."); //1
        try {
            Thread.sleep(1000);// Thread.sleep(3000)
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("park...");  //2
        LockSupport.park();
        //            LockSupport.park();
        System.out.println("resume...");//4
    },"t1");
    t1.start();
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println("unpark...");    //3
    LockSupport.unpark(t1);
}
```

```
start...
park...
unpark...
resume...
```

6、共享模型之无锁

结合 CAS 和 volatile 可以实现无锁并发，适用于线程数少、多核 CPU 的场景下。

- **CAS 是基于乐观锁的思想**：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- **synchronized 是基于悲观锁的思想**：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。
- CAS 体现的是无锁并发、无阻塞并发，请仔细体会这两句话的意思
-
  - 因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一
  - 但如果竞争激烈，可以想到重试必然频繁发生，反而效率会受影响

6.1、原子整数AtomicInteger

```
public static void main(String[] args) {
    AtomicInteger i = new AtomicInteger(0);

    // 获取并自增（i = 0, 结果 i = 1, 返回 0），类似于
i++
    System.out.println(i.getAndIncrement());

    // 自增并获取（i = 1, 结果 i = 2, 返回 2），类似于
++i
    System.out.println(i.incrementAndGet());

    // 自减并获取（i = 2, 结果 i = 1, 返回 1），类似于 --
i
    System.out.println(i.decrementAndGet());

    // 获取并自减（i = 1, 结果 i = 0, 返回 1），类似于 i-
-
```

```
        System.out.println(i.getAndDecrement());

        // 获取并加值（i = 0, 结果 i = 5, 返回 0）
        System.out.println(i.getAndAdd(5));

        // 加值并获取（i = 5, 结果 i = 0, 返回 0）
        System.out.println(i.addAndGet(-5));

        // 获取并更新（i = 0, p 为 i 的当前值, 结果 i = -2,
返回 0）
        // 其中函数中的操作能保证原子，但函数需要无副作用
        System.out.println(i.getAndUpdate(p -> p - 2));

        // 更新并获取（i = -2, p 为 i 的当前值, 结果 i = 0,
返回 0）
        // 其中函数中的操作能保证原子，但函数需要无副作用
        System.out.println(i.updateAndGet(p -> p + 2));

        // 获取并计算（i = 0, p 为 i 的当前值, x 为参数1, 结
果 i = 10, 返回 0）
        // 其中函数中的操作能保证原子，但函数需要无副作用
        // getAndUpdate 如果在 lambda 中引用了外部的局部变
量，要保证该局部变量是 final 的
        // getAndAccumulate 可以通过 参数1 来引用外部的局部变
量，但因为其不在 lambda 中因此不必是 final
        System.out.println(i.getAndAccumulate(10, (p, x)
-> p + x));

        // 计算并获取（i = 10, p 为 i 的当前值, x 为参数1, 结
果 i = 0, 返回 0）
        // 其中函数中的操作能保证原子，但函数需要无副作用
        System.out.println(i.accumulateAndGet(-10,
Integer::sum));
    }
```

`getAndAdd`适用于简单的**加减**操作，而`getAndAccumulate`提供了更高的灵活性（**乘法**），允许进行更复杂的原子更新操作。

6.2、原子引用 AtomicReference

```
public class DecimalAccountSafeCas {

    AtomicReference<BigDecimal> ref;

    public DecimalAccountSafeCas(BigDecimal balance)
{
        ref = new AtomicReference<>(balance);
    }

    public BigDecimal getBalance() {
        return ref.get();
    }

    public void withdraw(BigDecimal amount) {
        ref.accumulateAndGet(amount,
BigDecimal::subtract);
    }
}
```

6.3、ABA问题及解决

```
static AtomicReference<String> ref = new
AtomicReference<>("A");

public static void main(String[] args) throws
InterruptedException {
    System.out.println(("main start..."));
    // 获取值A
    // 这个共享变量被它线程修改过?
    String prev = ref.get();
    other(); // 模拟ABA问题
```

```
    sleep(1000);

    // 尝试改为C
    System.out.println(("change A->C {}" +
ref.compareAndSet(prev, "C")));
}

private static void other() throws
InterruptedException {

    new Thread(() -> {
        System.out.println(("change A->B {}" +
ref.compareAndSet(ref.get(), "B")));
    }, "t1").start();

    sleep(500);

    new Thread(() -> {
        System.out.println(("change B->A {}" +
ref.compareAndSet(ref.get(), "A")));
    }, "t2").start();

}
change A->B {}true
change B->A {}true
change A->C {}true
```

主线程仅能判断出共享变量的值与最初值 A 是否相同，不能感知到这种从 A 改为 B 又 改回 A 的情况，**如果主线程希望： 只要有其它线程【动过了】共享变量，那么自己的 cas 就算失败，这时，仅比较值是不够的，需要再加一个版本号。**

```
static AtomicStampedReference<String> ref = new
AtomicStampedReference<>("A", 0);

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

```
    System.out.println(("main start..."));
    // 获取值 A
    // 这个共享变量被它线程修改过?
    // 获取值 A
    String prev = ref.getReference();
    // 获取版本号
    int stamp = ref.getStamp();
    System.out.println(("版本 {}" + stamp));

    other();

    sleep(1000);
    // 尝试改为 C
    System.out.println("change A->C {}" +
                       ref.compareAndSet(prev, "C",
stamp, stamp + 1));
}

private static void other() throws
InterruptedException {

    new Thread(() -> {
        System.out.println(("change A->B {}" +
ref.compareAndSet(ref.getReference(), "B",

            ref.getStamp(), ref.getStamp() + 1)));
        System.out.println(("更新版本为 {}" +
ref.getStamp()));
    }, "t1").start();

    sleep(500);

    new Thread(() -> {
        System.out.println(("change B->A {}" +
ref.compareAndSet(ref.getReference(), "A",

            ref.getStamp(), ref.getStamp() + 1)));
```

```
        System.out.println(("更新版本为 {}" +
ref.getStamp()));
    }, "t2").start();

}
```

AtomicStampedReference 可以给原子引用加上版本号，追踪原子引用整个的变化过程，如：

A -> B -> A -> C ，通过AtomicStampedReference，我们可以知道，引用变量中途被更改了几次。

-

7、线程池与与异步回调

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

7.1、线程池基本原理

> 线程池参数

- int corePoolSize：常驻核心线程数，创建完线程以后，当有请求任务时，就会安排核心线程去完成任务，即当到达corePoolSize数量以后，就会**放到缓存队列**中。
- maximumPoolSize:当任务到达corePoolSize数量以后，**缓存队列中也满了**，就会开启最大线程数。
- long keepAliveTime：当线程池中的线程数量超过corePoolSize时，这些多余的线程在空闲状态下可以保持存活的时间。**一旦这段时间过去，如果线程仍然是空闲的，那么线程将被终止，直到线程池中的线程数量再次回到corePoolSize。**
- TimeUnit unit： keepAliveTime的单位。
- BlockingQueue workQueue：**阻塞队列，用来存储等待执行的任务。**线程池内的工作线程会从这个队列中取任务来执行。
- ThreadFactory threadFactory：用于创建线程池中工作线程的线程工厂，一般用默认的。

- RejectedExecutionHandler handler：拒绝策略，用于处理ThreadPoolExecutor中因为某些原因无法执行的任务。当任务被提交给线程池，**但线程池无法接受新任务时（线程池已满且工作队列也满）**。

> 线程池运行流程

- 线程池创建，准备好corePoolSize数量的核心线程，准备接受任务。
- 新的任务进来，用 corePoolSize准备好的空闲线程执行；corePoolSize满了，就将再进来的任务放入阻塞队列workQueue中，执行完任务之后的空闲的 corePoolSize就会自己去阻塞队列workQueue获取任务执行；阻塞队列workQueue也满了，就直接开最大线程数maximumPoolSize,**maximumPoolSize都执行好了，maximumPoolSize-corePoolSize数量空闲的线程会在keepAliveTime 指定的时间后自动销毁，终保持到corePoolSize大小；如果线程数开到了maximumPoolSize数量，还有新的任务进来，就会使用RejectedExecutionHandler 指定的拒绝策略进行处理。**

> 常见的4种线程池

newCacheThreadPool

如果线程池的当前规模超出处理需求，那么**将回收空闲的线程，若无可回收，则新建线程**。该类的线程池特别**适合于任务量经常变化的情况**。

```
public static void main(String[] args) {
    ExecutorService cachedThreadPool =
Executors.newCachedThreadPool();
    for (int i = 0; i < 10; i++) {
        final int index = i;
        try {
            Thread.sleep(index * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        cachedThreadPool.execute(() -> {
            System.out.println(index);
```

```
        });
    }
    cachedThreadPool.shutdown();
}
```

newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务时，如果线程池中的线程数没有达到corePoolSize，则创建新线程执行任务，达到corePoolSize后，多出的任务则在队列中等待。
（**corePoolSize=maximumPoolSize**）

```
public static void main(String[] args) {
    ExecutorService fixedThreadPool =
Executors.newFixedThreadPool(5);
    for (int i = 0; i < 10; i++) {
        // lambda表达式中必须是final的变量
        final int index = i;
        fixedThreadPool.execute(() -> {
            try {
                System.out.println(index);
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
    fixedThreadPool.shutdown();
}
```

newScheduleThreadPool

创建一个定长线程池，**支持定时（延时）及周期性任务执行。适用于需要多个后台线程执行周期任务。**

> 延时执行schedule

```
public static void method1() {
    ScheduledExecutorService executor =
Executors.newScheduledThreadPool(2);

    // 添加两个任务，希望它们都在 1s 后执行
    executor.schedule(() -> {
        System.out.println("任务1，执行时间：" + new
Date());
        try { Thread.sleep(2000); } catch
(InterruptedException e) { }
    }, 1000, TimeUnit.MILLISECONDS);

    executor.schedule(() -> {
        System.out.println("任务2，执行时间：" + new
Date());
    }, 1000, TimeUnit.MILLISECONDS);
}
```

> 固定频率执行(从任务执行**开始**算间隔)scheduleAtFixedRate

输出分析：一开始，延时 1s，接下来，由于任务执行时间 > 间隔时间，间隔被『撑』到了 2s

```
public static void method2() {
    ScheduledExecutorService pool =
Executors.newScheduledThreadPool(1);

    System.out.println(("start..."));
    pool.scheduleAtFixedRate(() -> {
        try {
            sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(("running..."));
    }, 1, 1, TimeUnit.SECONDS); // 延时时间 + 执行间隔
}
```

> 固定频率执行(从任务执行**结束**算间隔)scheduleWithFixedDelay

输出分析：一开始，延时 1s，scheduleWithFixedDelay 的间隔是 上一个任务结束 ⟷ 延时 ⟷ 下一个任务开始 所以间隔都是 3s

```
public static void method3() {
    ScheduledExecutorService pool =
Executors.newScheduledThreadPool(1);
    System.out.println(("start..."));
    pool.scheduleWithFixedDelay(()-> {
        System.out.println(("running..."));
        try {
            sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }, 1, 1, TimeUnit.SECONDS); // 延时时间 + 执行间隔
}
```

newSingleThreadExecutor

创建一个单线程的执行器。**这个线程池中的单个线程会按照任务的提交顺序依次执行任务，保证任务按顺序执行。**

```
public static void main(String[] args) {
    ExecutorService singleThreadExecutor =
Executors.newSingleThreadExecutor();
    for (int i = 0; i < 9; i++) {
        final int index = i;
        singleThreadExecutor.execute(() -> {
            try {
                System.out.println(index);
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
    singleThreadExecutor.shutdown();
}
```

> 开发中为什么使用线程池

- **降低资源的消耗**：创建和销毁线程需要时间和系统资源，特别是在需要频繁创建和销毁线程的场景下，这个开销是非常显著的。线程池通过重复利用一组已经创建好的线程来减少这种开销。
- **提高响应速度**：当任务到来时，如果有空闲线程，则可以立即将任务分配给空闲线程执行，无需等待线程的创建。这样就能快速响应任务请求，提高了应用程序的响应速度。

> 创建多少线程池合适

- 过小会导致程序不能充分地利用系统资源、容易导致饥饿
- 过大会导致更多的线程上下文切换，占用更多内存

CPU 密集型运算

通常采用 **cpu 核数 + 1** 能够实现最优的 CPU 利用率，+1 是保证当线程由于页缺失故障（操作系统）或其它原因

导致暂停时，额外的这个线程就能顶上去，保证 CPU 时钟周期不被浪费

I/O 密集型运算

CPU 不总是处于繁忙状态，例如，当你执行业务计算时，这时候会使用 CPU 资源，但当你执行 I/O 操作时、远程

RPC 调用时，包括进行数据库操作时，这时候 CPU 就闲下来了，你可以利用多线程提高它的利用率。

经验公式如下

```
线程数 = 核数 * 期望 CPU 利用率 * 总时间(CPU计算时间 + 等待时间) / CPU 计算时间
```

例如 4 核 CPU 计算时间是 50% ，其它等待时间是 50%，期望 cpu 被 100% 利用，套用公式

```
4 * 100% * 100% / 50% = 8
```

例如 4 核 CPU 计算时间是 10% ，其它等待时间是 90%，期望 cpu 被 100% 利用，套用公式

```
4 * 100% * 100% / 10% = 40
```

7.2、处理线程池异常

> submit方法，主动捉异常

```
public static void main(String[] args) {
    ExecutorService pool =
Executors.newFixedThreadPool(1);
    pool.submit(() -> {
        try {
            System.out.println(("task1"));
            int i = 1 / 0;
        } catch (Exception e) {
            System.out.println(("error:" + e));
        }
    });
}
task1
error:java.lang.ArithmeticException: / by zero
```

使用Future

```
public static void main(String[] args) throws
ExecutionException, InterruptedException {
    ExecutorService pool =
Executors.newFixedThreadPool(1);

    Future<Boolean> f = pool.submit(() -> {
        System.out.println(("task1"));
        int i = 1 / 0;
        return true;
    });
    System.out.println(("result:{}" + f.get()));
}
task1
Exception in thread "main"
java.util.concurrent.ExecutionException:
java.lang.ArithmeticException: / by zero
    at
java.util.concurrent.FutureTask.report(FutureTask.ja
va:122)
```

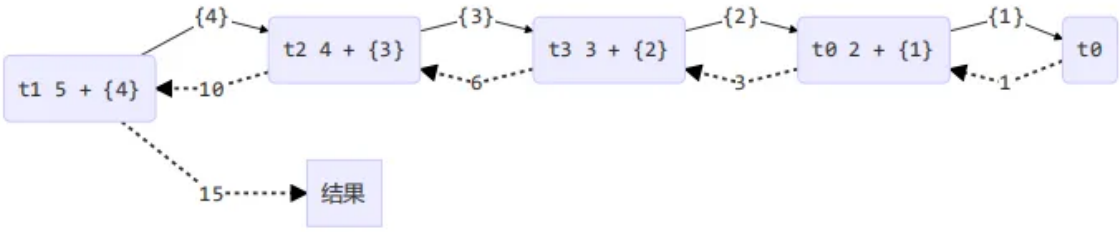
```
    at
java.util.concurrent.FutureTask.get(FutureTask.java:
192)
    at
ScheduledExecutorServiceTest.main(ScheduledExecutorS
erviceTest.java:16)
Caused by: java.lang.ArithmeticException: / by zero
    at
ScheduledExecutorServiceTest.lambda$main$0(Scheduled
ExecutorServiceTest.java:13)
    at
java.util.concurrent.FutureTask.run(FutureTask.java:
266)
    at
java.util.concurrent.ThreadPoolExecutor.runWorker(Th
readPoolExecutor.java:1149)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.run(T
hreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)
```

7.3、Fork/Join（分治思想）

- Fork/Join 是 JDK 1.7 加入的新的线程池实现，它体现的是一种分治思想，**适用于能够进行任务拆分的 cpu 密集型运算。**
- Fork/Join 在分治的基础上加入了多线程，可以**把每个任务的分解和合并交给不同的线程来完成，进一步提升了运算效率。**
- **Fork/Join 默认会创建与 cpu 核心数大小相同的线程池 。**

> 基础使用

提交给 Fork/Join 线程池的任务需要继承 **RecursiveTask（有返回值）或 RecursiveAction（没有返回值）**，例如下面定义了一个对 1~n 之间的整数求和的任务。



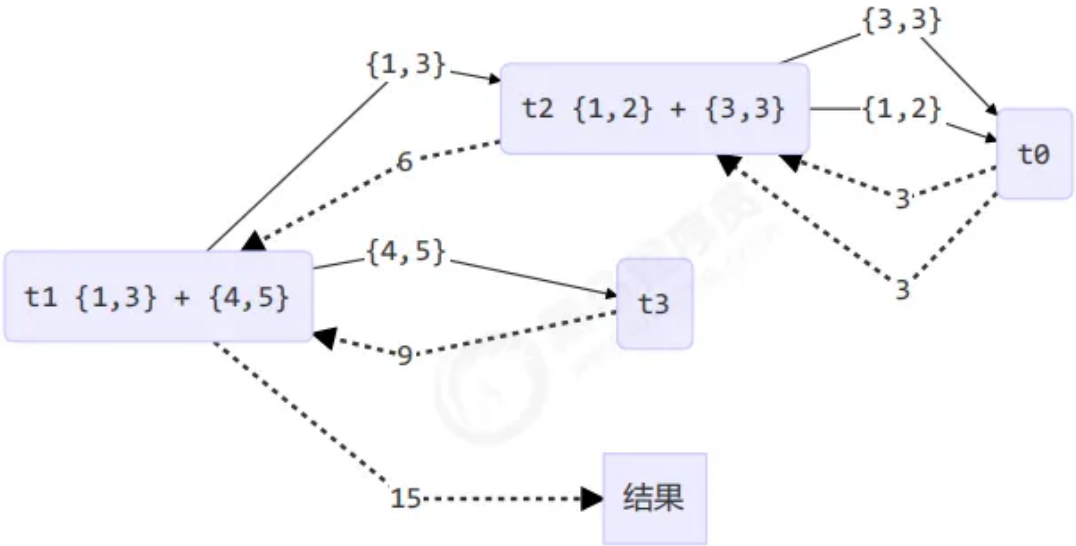
- compute方法李需要自定义递归得逻辑
- fork()执行任务，join获取结果

```
public class AddTask1 extends RecursiveTask<Integer>
{
    int n;

    public AddTask1(int n) {
        this.n = n;
    }

    @Override
    public String toString() {
        return "{" + n + '}';
    }

    @Override
    protected Integer compute() {
        // 终止条件：如果 n 已经为 1，可以求得结果了
        if (n == 1) {
            System.out.println(("join() {}" + n));
            return n;
        }

        // 将任务进行拆分(fork)
        AddTask1 t1 = new AddTask1(n - 1);
        t1.fork(); // 让一个线程去执行
        System.out.println(("fork() {} + {}" + n +
t1));

        // 合并(join)结果
```

```
        int result = n + t1.join();
        System.out.println(("join() {} + {} = {}" +
n + t1 + result));
        return result;
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(new
AddTask1(5)));
    }
}
```

> 也可以使用**二分**来加速

原始得做法**任务之间需要相互依赖等待**，可以减少这方面的消耗。




```
class AddTask3 extends RecursiveTask<Integer> {

    int begin;
    int end;

    public AddTask3(int begin, int end) {
        this.begin = begin;
```

```
        this.end = end;
    }

    @Override
    public String toString() {
        return "{" + begin + "," + end + '}';
    }

    @Override
    protected Integer compute() {
        // 5, 5
        if (begin == end) {
            log.debug("join() {}", begin);
            return begin;
        }
        // 4, 5
        if (end - begin == 1) {
            log.debug("join() {} + {} = {}", begin,
end, end + begin);
            return end + begin;
        }

        // 1 5
        int mid = (end + begin) / 2; // 3
        AddTask3 t1 = new AddTask3(begin, mid); //
1,3
        t1.fork();
        AddTask3 t2 = new AddTask3(mid + 1, end); //
4,5
        t2.fork();
        log.debug("fork() {} + {} = ?", t1, t2);
        int result = t1.join() + t2.join();
        log.debug("join() {} + {} = {}", t1, t2,
result);
        return result;
    }
}
```

7.4、CompletableFuture 异步回调

Futrue 在 Java 里面，通常用来表示一个异步任务的引用，比如我们将任务提交到线程池里面，然后我们会得到一个 Futrue，在 Future 里面有 isDone 方法来 判断任务是否处理结束，还有 get 方法可以一直阻塞直到任务结束然后获取结果，但**整体来说这种方式，还是同步的，因为需要客户端不断阻塞等待或者不断轮询才能知道任务是否完成。**

7.4.1、创建异步对象

```
runAsync(Runnable runnable)
```

此方法使用`ForkJoinPool.commonPool()`作为它的线程池执行一个异步操作，不返回结果。

```
CompletableFuture<Void> future =
CompletableFuture.runAsync(() -> {
    // 异步执行的任务
    System.out.println("Running in a separate thread
than the main thread.");
});
```

```
runAsync(Runnable runnable, Executor executor)
```

与`runAsync(Runnable runnable)`类似，但它允许你指定一个自定义的`Executor`来执行异步任务，不返回结果。

```
Executor executor = Executors.newCachedThreadPool();
CompletableFuture<Void> future_ =
CompletableFuture.runAsync(() -> {
    // 异步执行的任务
    System.out.println("Running with a custom
Executor.");
}, executor);
```

```
supplyAsync(Supplier supplier)
```

此方法使用ForkJoinPool.commonPool()执行一个异步操作，并返回一个CompletableFuture，其中包含了操作的结果。Supplier<U>是一个函数式接口，它返回一个值。

```
CompletableFuture<String> future = 
CompletableFuture.supplyAsync(() -> {
    // 异步执行的任务，并返回结果
    return "Result of the asynchronous computation";
});
```

> supplyAsync(Supplier supplier, Executor executor)

与supplyAsync(Supplier<U> supplier)类似，但它允许你指定一个自定义的Executor来执行异步任务，并返回一个包含操作结果的CompletableFuture。

```
Executor executor = Executors.newFixedThreadPool(2);
CompletableFuture<String> future = 
CompletableFuture.supplyAsync(() -> {
    // 异步执行的任务，并返回结果
    return "Result of the asynchronous computation 
with a custom Executor";
}, executor);
```

CompletableFuture的这四个静态方法提供了创建异步操作的灵活方式，既可以不返回结果（使用runAsync），也可以返回结果（使用supplyAsync）。此外，它们允许使用默认的线程池或指定自定义的Executor，从而给予开发者更大的控制权，以适应不同的需求和优化执行性能。

```
public static void main(String[] args) throws 
ExecutionException, InterruptedException {
    Executor executor = 
Executors.newCachedThreadPool();
    System.out.println("main....start.....");
    CompletableFuture<Void> completableFuture = 
CompletableFuture.runAsync(() -> {
```

```
        System.out.println("当前线程: " +
Thread.currentThread().getId()); // 当前线程: 20
        int i = 10 / 2;
        System.out.println("运行结果: " + i); // 运行结
果: 5
    }, executor);

    CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程: " +
Thread.currentThread().getId()); // 当前线程: 21
        int i = 10 / 2;
        System.out.println("运行结果: " + i); // 运行结
果: 5
        return i;
    }, executor);
    Integer integer = future.get();

    System.out.println("main....stop....." +
integer);
}
```

7.4.2、计算完成时回调Complete方法

whenComplete/whenCompleteAsync 可以处理正常和异常的计算结果，exceptionally 处理异常情况。

whenComplete ：执行当前任务的线程继续执行回调任务。

whenCompleteAsync： 这个回调任务继续提交给线程池来进行执行。

方法不以 Async 结尾，意味着 Action 使用相同的线程执行，而 Async 可能会使用其他线程池执行（如果是使用相同的线程池，也可能会被同一个线程选中执行）。

```
public static void main(String[] args) {
    Executor executor =
Executors.newCachedThreadPool();
```

```
    // 创建异步任务
    CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程: " +
Thread.currentThread().getId()); // 当前线程: 20
        int i = 10 / 0;
        System.out.println("运行结果: " + i);
        return i;
    }, executor).whenComplete((res,exception) ->{
        // 异步任务成功完成了...结果是: null异常是:
java.util.concurrent.CompletionException:
java.lang.ArithmeticException: / by zero
        System.out.println("异步任务成功完成了...结果
是: " +res + "异常是: " + exception);
    }).exceptionally(throwable -> {
        return 10;
    });
}
```

- 使用`CompletableFuture.supplyAsync(Supplier<U> supplier, Executor executor)`方法创建一个异步任务。这个任务在由`Executors.newCachedThreadPool()`提供的线程池中执行。
- 异步任务的内容是打印当前线程的ID，然后尝试执行一个会导致`ArithmeticException`的除以零操作，接着打印运行结果（实际上，由于异常，后面的打印不会被执行）
- 使用`whenComplete(BiConsumer<? super T,? super Throwable> action)`方法来处理任务完成时的情况，无论任务是正常结束还是异常结束。这个回调方法有两个参数：结果`res`和异常`exception`。**如果异步任务正常完成，`exception`将是`null`；如果异步任务抛出异常，`res`将是`null`，而`exception`将包含异常信息。**
- 在这个例子中，由于异步任务中发生了除以零的操作，所以会触发`ArithmeticException`异常。因此，`whenComplete`方法中，`res`是`null`，`exception`是`java.util.concurrent.CompletionException: java.lang.ArithmeticException: / by zero`。

- **使用exceptionally(Function<Throwable, ? extends T> fn)方法来处理任务因异常而结束的情况。**这允许你为异常情况提供一个默认值或者基于异常类型进行恢复，返回一个新的结果。**在这个例子中，当发生异常时，返回了一个默认值10。**

7.4.3、异常处理handle方法

和回调一样，可以对结果做最后的处理（可处理异常），可改变返回值。

```
public static void testHandle() {
    Executor executor =
Executors.newCachedThreadPool();
    CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程: " +
Thread.currentThread().getId()); // 当前线程: 22
        int i = 10 / 2;
        System.out.println("运行结果: " + i); // 运行结
果: 5
        return i;
    }, executor).handle((res,thr) ->{
        if (res != null ) {
            return res * 2;
        }
        if (thr != null) {
            return 0;
        }
        return 0;
    });
}
```

- **结果转换：handle可以基于计算的成功或不同类型的失败来转换结果，而whenComplete仅用于执行一些操作，如记录日志，发送通知等，不影响结果。**
- **异常处理**：两者都可以处理异常，但**handle能够通过返回不同的值来恢复异常，而whenComplete不能修改返回结果**，只能执行一些附加操作。

- whenComplete方法允许你处理正常完成的计算结果或异常，但它不改变CompletableFuture的结果。即使你在whenComplete的回调函数中对结果进行了“修改”，这种“修改”并不会影响CompletableFuture的最终结果。**whenComplete更多的是用来执行一些操作（如记录日志）。exceptionally方法提供了一种异常恢复机制，允许你在遇到异常时提供一个备选的结果。**这意味着如果前面的异步操作失败了，你可以在exceptionally方法中返回一个“默认”值或者基于异常类型进行恢复，从而改变CompletableFuture的最终结果。

7.4.4、线程依赖方法then

> thenApply

thenApply方法：当一个线程依赖另一个线程时，获取上一个任务返回的结果，并**返回当前任务的返回值**。

```
public static void testThenApply() throws
ExecutionException, InterruptedException {
    CompletableFuture<Integer> original =
CompletableFuture.supplyAsync(() -> 5);
    CompletableFuture<Integer> result =
original.thenApply(new Function<Integer, Integer>()
{
        @Override
        public Integer apply(Integer integer) {
            return integer * integer;
        }
    });
    Integer integer = result.get();
    System.out.println(integer); // 25
}
```

> thenAccept

thenAccept方法：消费处理结果，接受任务处理结果，并消费处理，**无返回结果**。

```
public static void testThenAccept() throws
ExecutionException, InterruptedException {
    CompletableFuture<Void> future =
CompletableFuture.supplyAsync(() -> "Hello")
            .thenAccept(result ->
System.out.println(result + " World"));
    // 输出: Hello World
}
```

> thenRun

thenRun方法：**不直接处理前一步的计算结果，而是在计算完成后执行一段代码**。

```
public static void testThenRun() throws
ExecutionException, InterruptedException {
    CompletableFuture<Void> future =
CompletableFuture.supplyAsync(() -> {
            // 模拟一些长时间运行的异步任务
            return "Task completed";
    }).thenRun(() -> System.out.println("任务完成，这
是后续操作"));
    // 输出: 任务完成，这是后续操作
}
```

7.4.5、两任务组合-都要完成

> thenCombine

组合两个CompletableFuture的结果，**并返回一个新的CompletableFuture，其值是两个CompletableFuture结果的组合。**

```
public static void testCombine() throws
ExecutionException, InterruptedException {
    ExecutorService executor =
Executors.newCachedThreadPool();
```

```
    /**
     * 两个都完成
     */
    CompletableFuture<Integer> future01 =
CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1当前线程：" +
Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束：" + i);
        return i;
    }, executor);

    CompletableFuture<String> future02 =
CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2当前线程：" +
Thread.currentThread().getId());
        System.out.println("任务2结束：");
        return "Hello";
    }, executor);
    // f1 和 f2 单独定义返回结果
    CompletableFuture<String> future =
future01.thenCombineAsync(future02, (f1, f2) -> {
        return f1 + ":" + f2 + "-> Haha";
    }, executor);
    System.out.println("main....end....." +
future.get());
    executor.shutdown();
}
```

> thenAcceptBoth

组合两个 future，获取两个 future 任务的返回结果，**然后处理任务，没有返回值**。

```
public static void testAcceptBoth() {
    CompletableFuture<Integer> future1 =
CompletableFuture.supplyAsync(() -> 5);
    CompletableFuture<Integer> future2 =
CompletableFuture.supplyAsync(() -> 10);
    future1.thenAcceptBoth(future2, (num1, num2) ->
System.out.println("Sum: " + (num1 + num2)));
    // 输出: Sum: 15
}
```

runAfterBoth

在两个CompletableFuture任务都执行完成后，执行一个Runnable任务，**这个任务不处理前面CompletableFuture的结果。**

```
public static void testRunAfterBoth() {
    CompletableFuture<Void> future1 =
CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            throw new IllegalStateException(e);
        }
        System.out.println("First task completed.");
    });

    CompletableFuture<Void> future2 =
CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            throw new IllegalStateException(e);
        }
        System.out.println("Second task
completed.");
    });
```

```
    future1.runAfterBoth(future2, () ->
System.out.println("Both tasks completed."));
    // 输出:
    // First task completed.
    // Second task completed.
    // Both tasks completed.
}
```

7.4.6、两任务组合-一个完成

- applyToEither;两个任务有一个执行完成，**获取它的返回值，处理任务并有新的返回值**
- acceptEither: 两个任务有一个执行完成，**获取它的返回值，处理任务，没有新的返回值**
- runAfterEither:两个任务有一个执行完成，**不需要获取它的返回值，处理别的任务**

```
CompletableFuture<String> future =
future01.applyToEitherAsync(future02, res -> {
    System.out.println("任务3开始...之前的结果: " +
res);
    return res.toString() + "->哈哈";
}, executor);
```

7.4.7、多任务组合

allOf：**等待所有任务完成**

anyOf：**只要有一个任务完成**

```
public static void testAny() throws
ExecutionException, InterruptedException {

    CompletableFuture<String> futureImg =
CompletableFuture.supplyAsync(() -> {
        System.out.println("查询商品的图片信息");
```

```
        return "hello.jpg";
    });

    CompletableFuture<String> futureAttr =
CompletableFuture.supplyAsync(() -> {
        System.out.println("查询商品的属性");
        return "黑色+256G";
    });

    CompletableFuture<String> futureDesc =
CompletableFuture.supplyAsync(() -> {
        try { TimeUnit.SECONDS.sleep(3); } catch
(InterruptedException e) { e.printStackTrace(); }
        System.out.println("查询商品介绍");
        return "华为";
    });

    // 等待全部执行完
    //        CompletableFuture<Void> allOf =
CompletableFuture.allOf(futureImg, futureAttr,
futureDesc);
    //        allOf.get();

    // 只需要有一个执行完
    CompletableFuture<Object> anyOf =
CompletableFuture.anyOf(futureImg, futureAttr,
futureDesc);
    anyOf.get();
    System.out.println("main....end....." +
anyOf.get());

}
```

CompletableFuture的操作是异步执行的。如果主线程（例如运行main方法的线程）在异步操作完成之前退出了，那么这些异步操作可能还没有机会产生输出。确保主线程等待足够长的时间，以便异步操作有机会完成。例如，可以在**main方法的末尾使用**Thread.sleep来延迟主线程

的结束。

—

8、AQS线程同步框架

8.1、AQS原理

全称是 AbstractQueuedSynchronizer，是阻塞式锁和相关的同步器工具的框架。

> state属性 独占/共享模式

用 state 属性来表示资源的状态（分独占模式和共享模式），**子类需要定义如何维护这个状态，控制如何获取锁和释放锁**

- - getState - 获取 state 状态
  - setState - 设置 state 状态
  - compareAndSetState - cas 机制设置 state 状态
  - 独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源

> 等待队列

提供了基于 FIFO 的等待队列，**类似于 Monitor 的 EntryList**

> 条件变量

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

> 需要子类实现的方法

子类主要实现这样一些方法（默认抛出 UnsupportedOperationException）

- tryAcquire
- tryRelease

- tryAcquireShared
- tryReleaseShared
- isHeldExclusively

> 获取释放锁的方法

```
// 如果获取锁失败
if (!tryAcquire(arg)) {
    // 入队，可以选择阻塞当前线程 park unpark
}

// 如果释放锁成功
if (tryRelease(arg)) {
    // 让阻塞线程恢复运行
}
```

8.2、AQS设计原理

AQS 要实现的功能目标

- 阻塞版本获取锁 acquire 和非阻塞的版本尝试获取锁 tryAcquire
- 获取锁超时机制
- **通过打断取消机制**
- 独占机制及共享机制
- **条件不满足时的等待机制**

```
// 获取锁的逻辑
while(state 状态不允许获取) {
    if(队列中还没有此线程) {
        入队并阻塞
    }
}
当前线程出队

// 释放锁的逻辑
if(state 状态允许了) {
    恢复阻塞的线程(s)
}
```

> state 设计

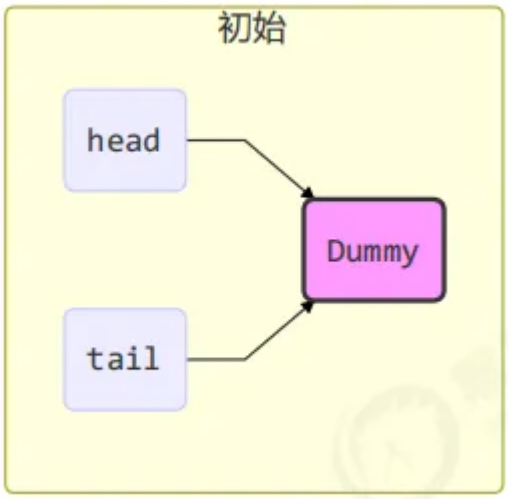
- **state 使用 volatile 配合 cas 保证其修改时的原子性**
- state 使用了 32bit int 来维护同步状态，因为当时使用 long 在很多平台下测试的结果并不理想

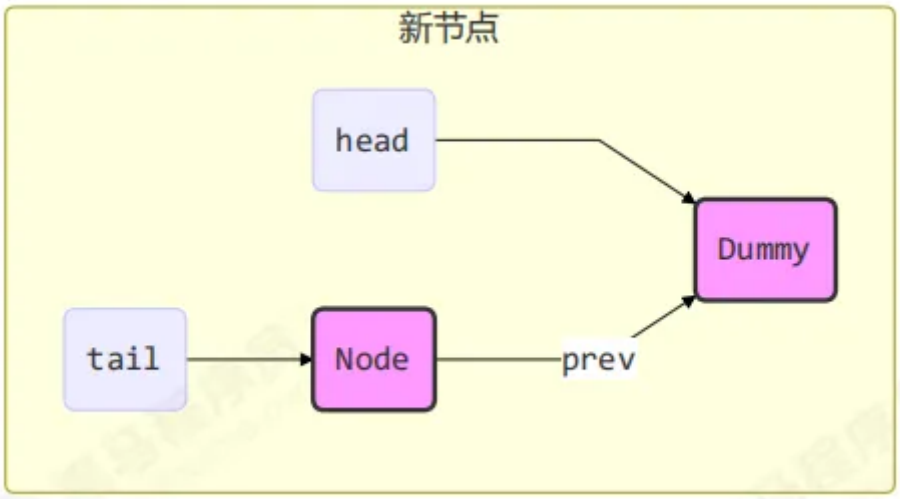
> 阻塞恢复设计

- 早期的控制线程暂停和恢复的 api 有 suspend 和 resume，但它们是不可用的，因为如果先调用的 resume 那么 suspend 将感知不到
- 解决方法是使用 park & unpark 来实现线程的暂停和恢复，具体原理在之前讲过了，先 unpark 再 park 也没问题
- **park & unpark 是针对线程的**，而不是针对同步器的，因此控制粒度更为精细
- **park 线程还可以通过 interrupt 打断**

> 队列设计

- 使用了 FIFO 先入先出队列，并不支持优先级队列
- 设计时借鉴了 CLH 队列，它是一种**单向无锁队列**





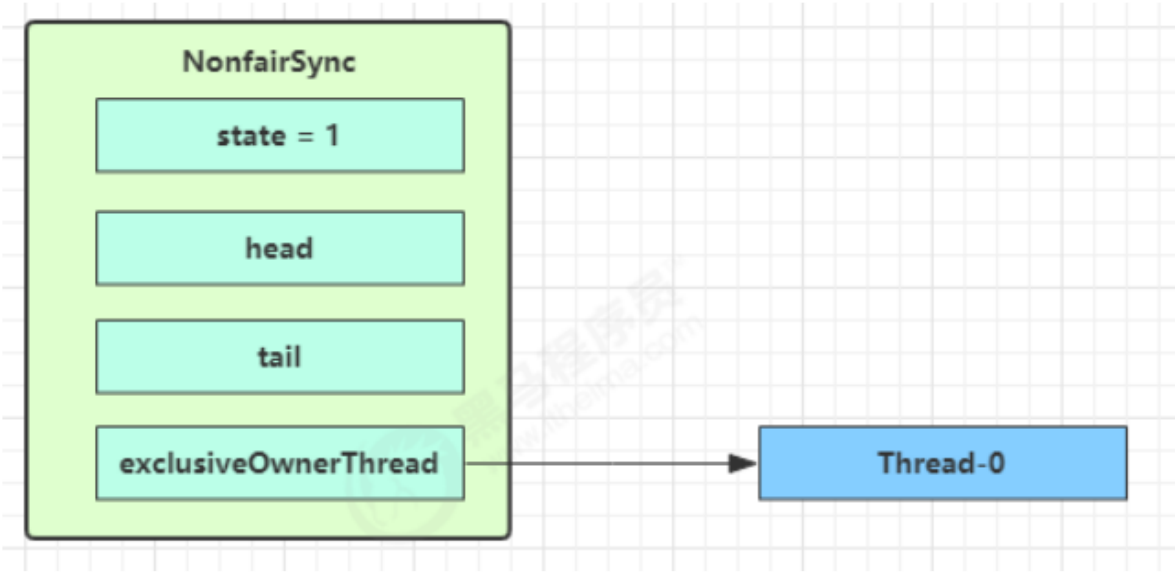
队列中有 head 和 tail 两个指针节点，都用 volatile 修饰配合 cas 使用。

```
// 入队伪代码，只需要考虑 tail 赋值的原子性
do {
    // 原来的 tail
    Node prev = tail;
    // 用 cas 在原来 tail 的基础上改为 node
} while(tail.compareAndSet(prev, node))

// 出队伪代码
// prev 是上一个节点
while((Node prev=node.prev).state ≠ 唤醒状态) {
}
// 设置头节点
head = node;
```

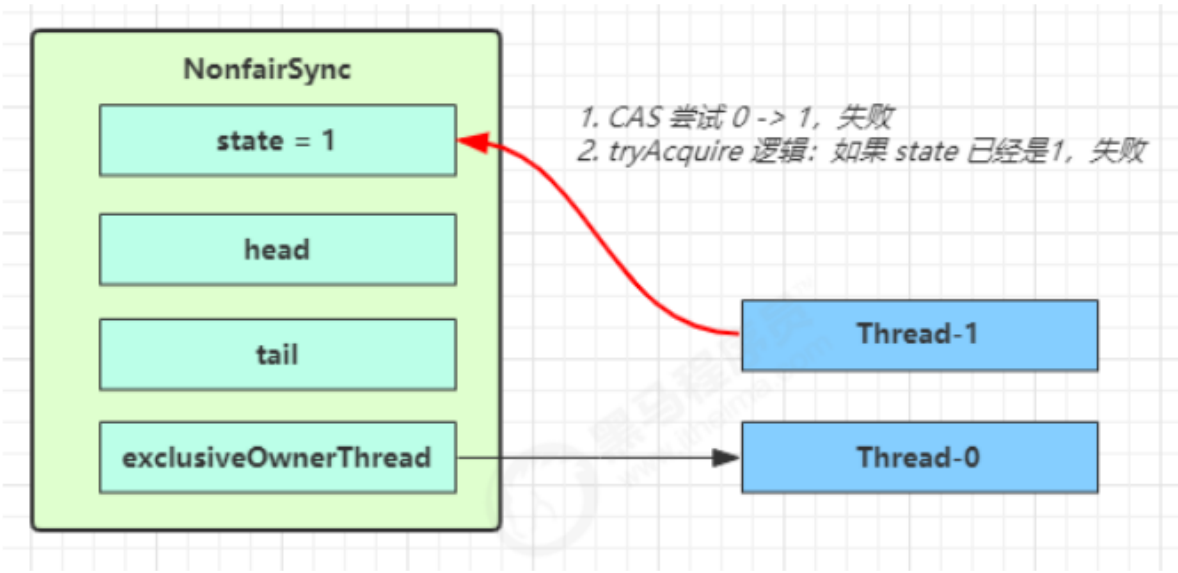
8.3、ReentrantLock原理及使用

8.3.1、加锁解锁流程

> 没有竞争时(占有锁)



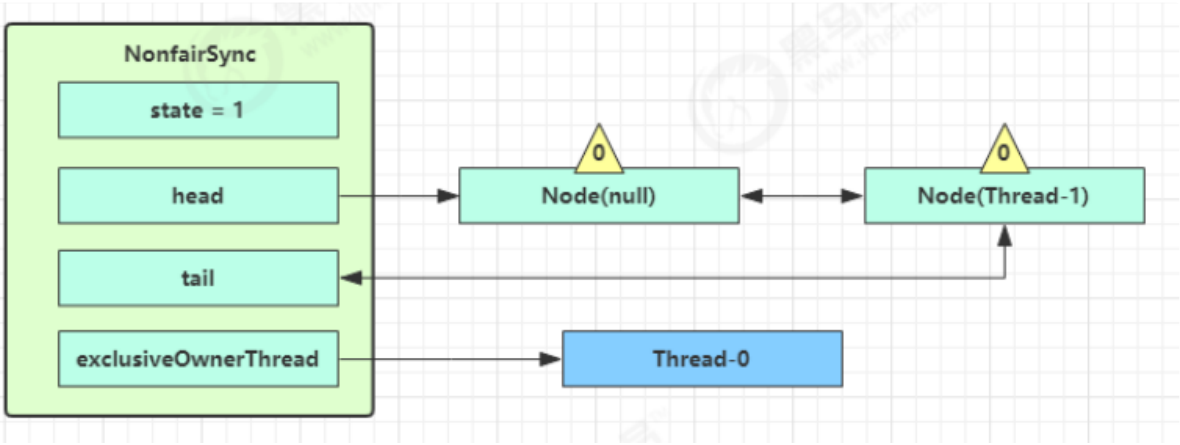
> 第一个竞争出现时(发生排队)



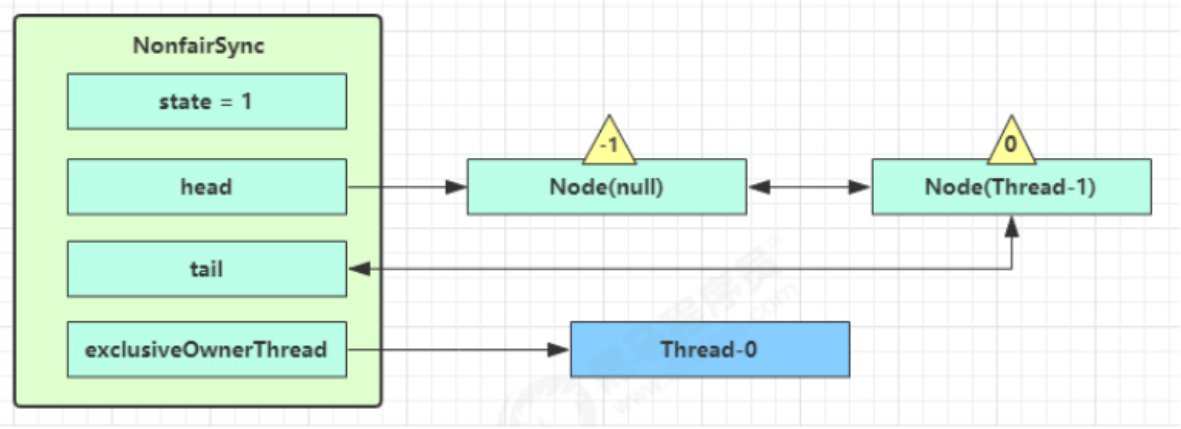
Thread-1 执行了

- CAS 尝试将 state 由 0 改为 1，结果失败
- 进入 tryAcquire 逻辑，这时 state 已经是1，结果仍然失败
- 接下来进入 addWaiter 逻辑，构造 Node 队列

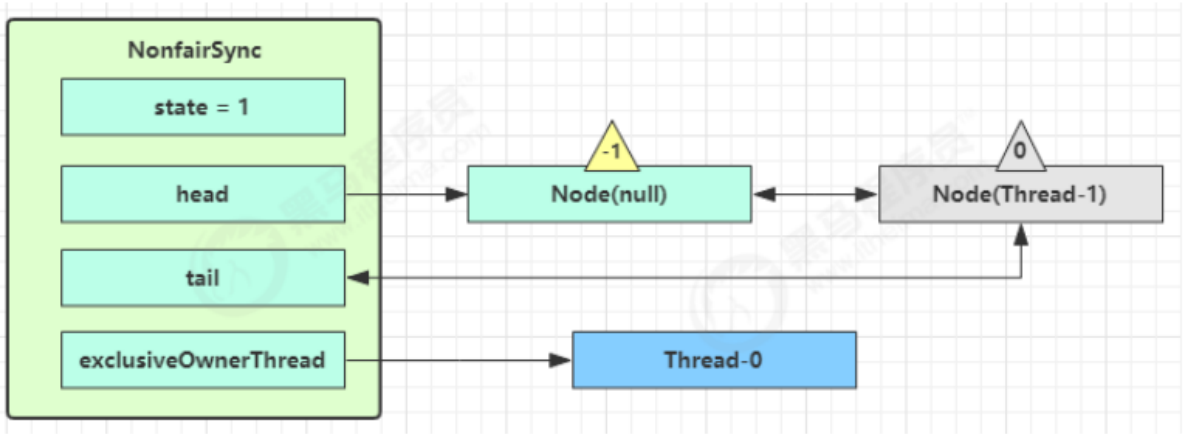
- - **图中黄色三角表示该 Node 的 waitStatus 状态，其中 0 为默认正常状态**
  - **其中第一个 Node 称为 Dummy（哑元）或哨兵，用来占位，并不关联线程**



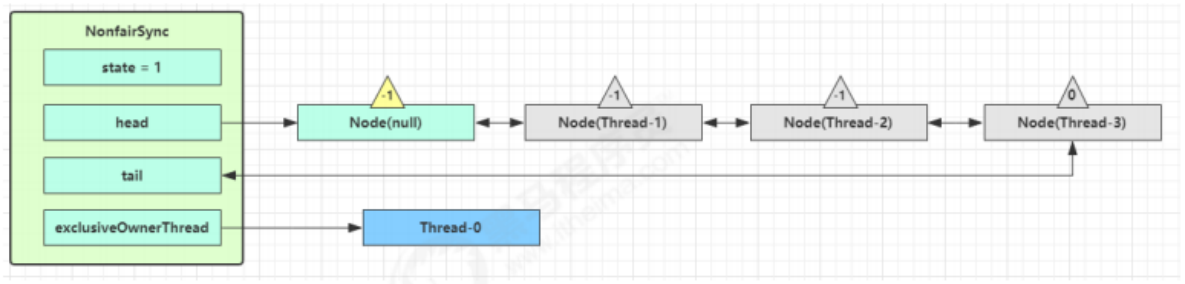
- 当前线程进入 acquireQueued 逻辑
- **acquireQueued 会在一个死循环中不断尝试获得锁，失败后进入 park 阻塞**
- **如果自己是紧邻着 head（排第二位），那么再次 tryAcquire 尝试获取锁，当然这时 state 仍为 1，失败**
- **进入 shouldParkAfterFailedAcquire 逻辑，将前驱 node，即 head 的 waitStatus 改为 -1，这次返回 false**



- shouldParkAfterFailedAcquire 执行完毕回到 acquireQueued ，再次 tryAcquire 尝试获取锁，当然这时 state 仍为 1，失败
- **当再次进入 shouldParkAfterFailedAcquire 时，这时因为其前驱 node 的 waitStatus 已经是 -1，这次返回true**
- 进入 parkAndCheckInterrupt， Thread-1 park（灰色表示）



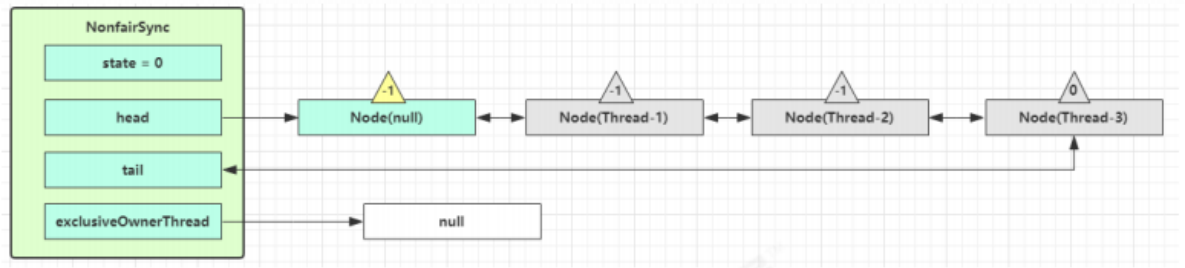
再次有多个线程经历上述过程竞争失败，变成这个样子



> 原OwnerThread释放锁时

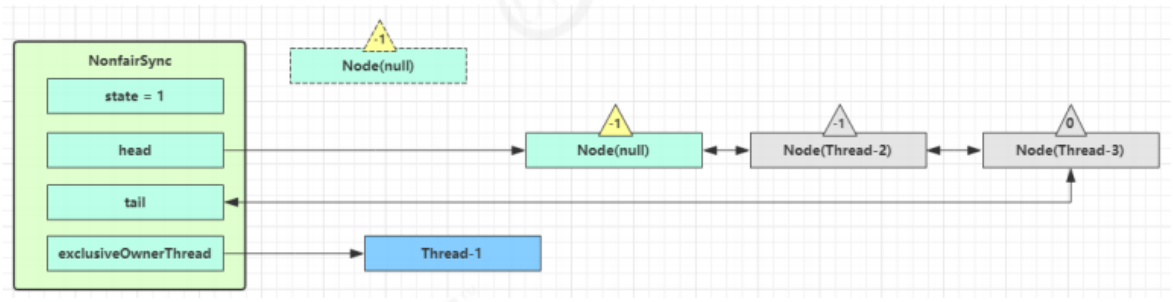
Thread-0 释放锁，进入 tryRelease 流程，**如果成功**

- 设置 exclusiveOwnerThread 为 null
- state = 0



> 队列内线程抢到锁

- 当前队列不为 null，并且 head 的 waitStatus = -1，进入 unparkSuccessor 流程
- 找到队列中离 head 最近的一个 Node（没取消的），unpark 恢复其运行，本例中即为 Thread-1
- 回到 Thread-1 的 acquireQueued 流程

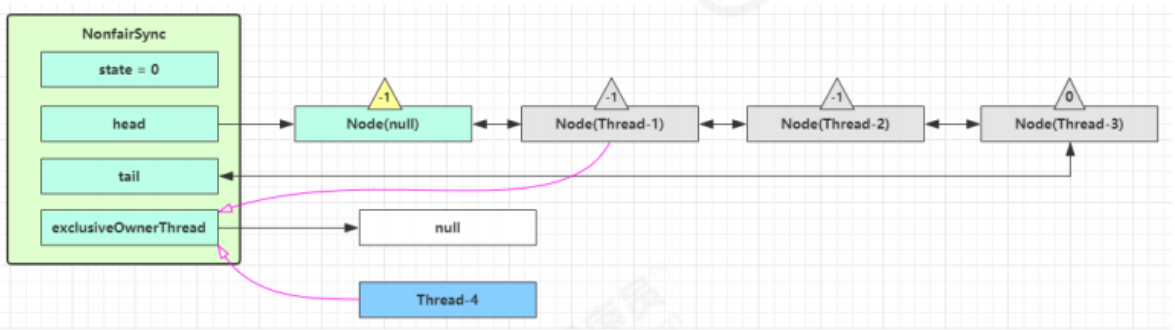


如果加锁成功（没有竞争），会设置

- exclusiveOwnerThread 为 Thread-1，state = 1
- head 指向刚刚 Thread-1 所在的 Node，该 Node 清空 Thread
- 原本的 head 因为从链表断开，而可被垃圾回收

> 队列外线程抢到锁

如果这时候有其它线程来竞争（非公平的体现），例如这时有 Thread-4 来了



如果不巧又被 Thread-4 占了先

- Thread-4 被设置为 exclusiveOwnerThread，state = 1
- Thread-1 再次进入 acquireQueued 流程，获取锁失败，重新进入 park 阻塞

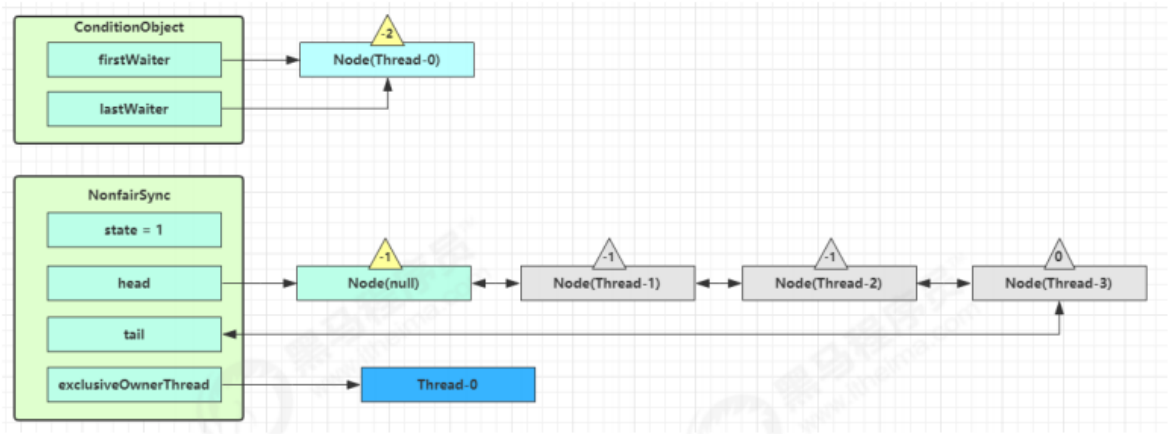
8.3.2、条件变量实现原理

每个条件变量其实就对应着一个等待队列，其实现类是 ConditionObject

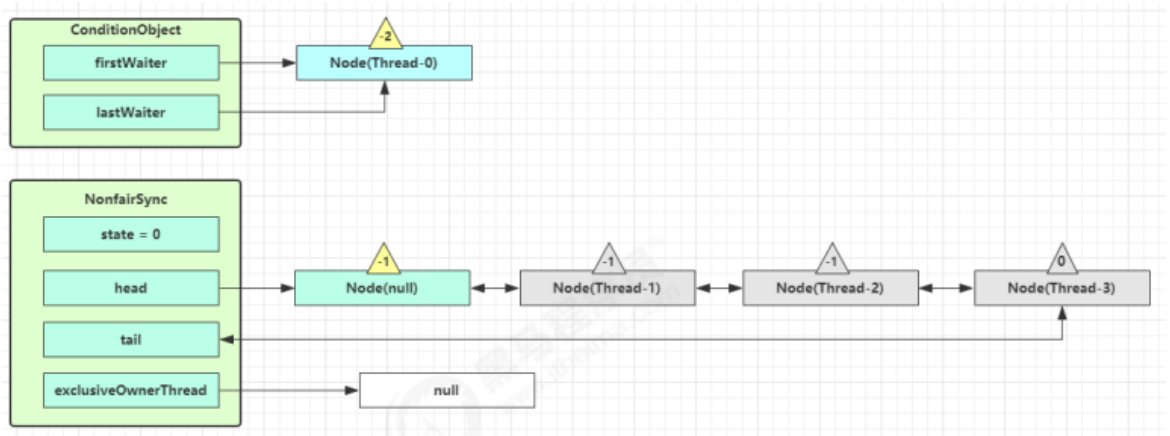
> await 流程

Thread-0 持有锁，调用 await，进入 ConditionObject 的 addConditionWaiter 流程

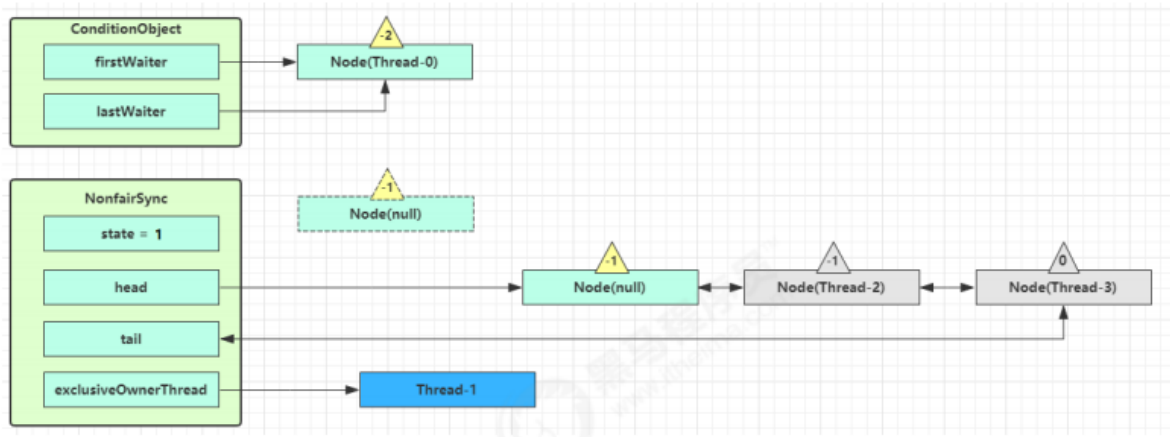
创建新的 Node 状态为 -2 (Node.CONDITION)，关联 Thread-0，加入等待队列尾部



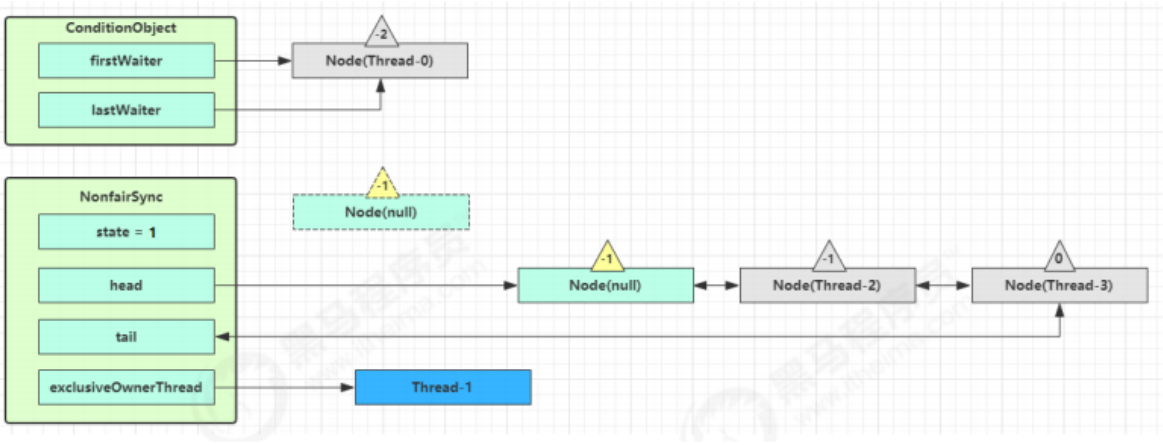
接下来进入 AQS 的 fullyRelease（**有可能重入锁**） 流程，释放同步器上的锁



unpark AQS 队列中的下一个节点，竞争锁，假设没有其他竞争线程，那么 Thread-1 竞争成功

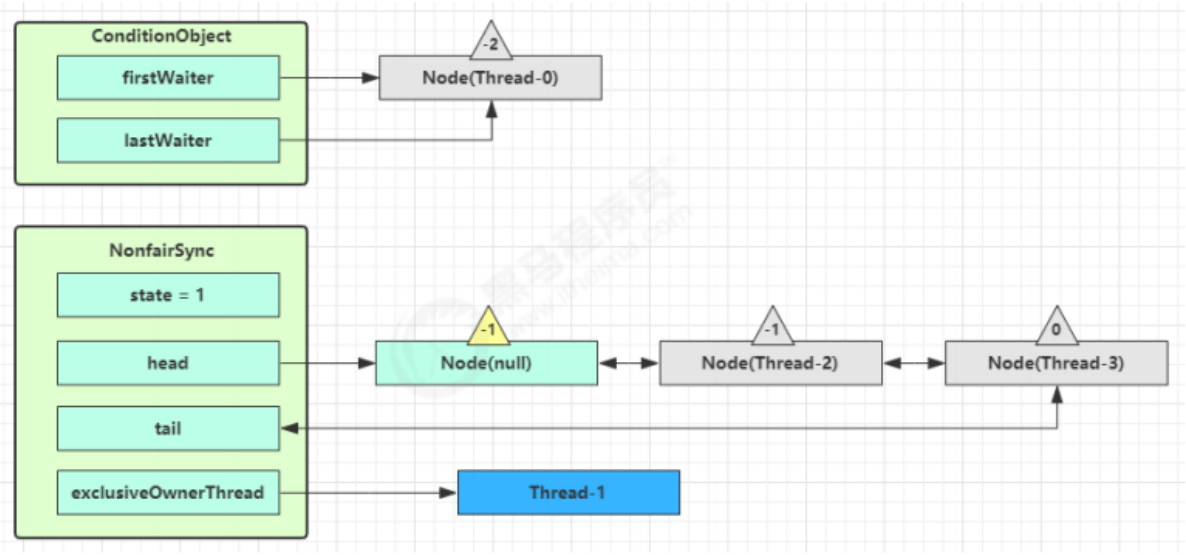


park 阻塞 Thread-0

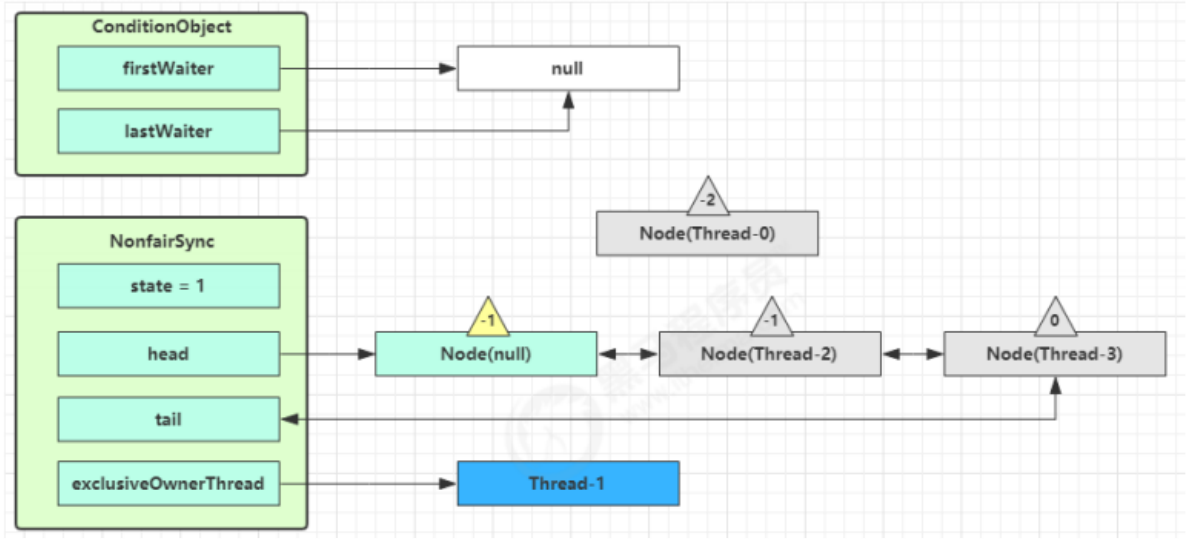


signal 流程

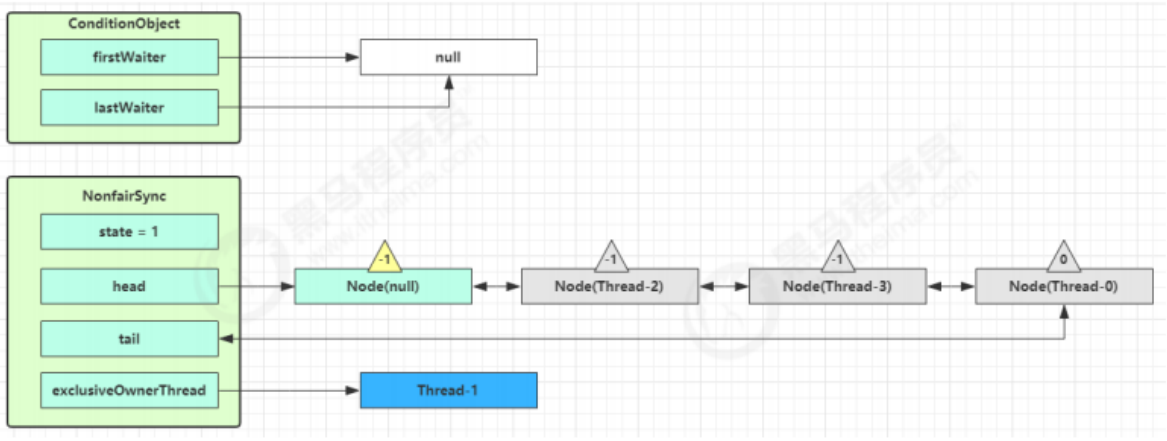
假设 Thread-1 要来唤醒 Thread-0



进入 ConditionObject 的 doSignal 流程，取得等待队列中第一个 Node，即 Thread-0 所在 Node



执行 transferForSignal 流程，**将该 Node 加入 AQS 队列尾部，将 Thread-0 的 waitStatus 改为 0，Thread-3 的waitStatus 改为 -1**



Thread-1 释放锁，进入 unlock 流程，略

8.3.3、ReentrantLock使用实例

```
public class ReentrantLockTest extends Thread {

    // 创建锁对象  true是公平锁
    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(10);
        // 创建多个线程
        for (int i = 0; i < 10; i++) {
            fixedThreadPool.execute(myThread);
        }
        fixedThreadPool.shutdown();
    }


    static class MyThread implements Runnable {
```

```
        @Override
        public void run() {
            // 尝试获取锁
            lock.lock();
            try {
                Thread.sleep(1000);
                // 打印执行线程的名字
                System.out.println("线程:" +
Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            } finally {
                // 释放锁
                lock.unlock();
            }
        }
    }
}
```

- 使用 ReentrantLock 时，需要注意**不要将加锁操作放在 try 中**，这样会导致未加锁成功就执行了释放锁的操作，从而导致程序执行异常。**例如在try代码中先出现异常，再加锁（实际没有执行），然后再释放锁，释放锁的异常会覆盖程序原有的异常。**
- 使用 ReentrantLock 时一定要记得释放锁，否则就会导致该锁一直被占用，其他使用该锁的线程则会永久的等待下去，所以我们在使用 ReentrantLock 时，**一定要在 finally 中释放锁**，这样就可以保证锁一定会被释放。

8.3.4、源码解读

8.3.4.1、非公平锁加锁流程

> 加锁代码

```
// 加锁实现
final void lock() {
```

```
    // 首先用 cas 尝试（仅尝试一次）将 state 从 0 改为 1，如果成功表示获得了独占锁
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // 如果尝试失败，进入 ㈠
        acquire(1);
}

// ㈠ AQS 继承过来的方法，方便阅读，放在此处
public final void acquire(int arg) {
    // ㈡ tryAcquire
    if (
        !tryAcquire(arg) &&
        // 当 tryAcquire 返回为 false 时，先调用 addWaiter ㈣，接着 acquireQueued ㈤
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg)
    ) {
        selfInterrupt();
    }
}
```

> 尝试获取锁

```
// ㈡ 进入 ㈢
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

// ㈢ Sync 继承过来的方法，方便阅读，放在此处
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // 如果当前还没有其他线程获得锁
```

```
    if (c == 0) {
        // 尝试用 cas 获得，这里体现了非公平性：不去检查
AQS 队列
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 如果当前线程已经获取锁，说明发生了锁重入
    else if (current == getExclusiveOwnerThread()) {
        // state++ 锁重入计数
        int nextc = c + acquires;
        if (nextc < 0) // overflow，超出了锁重入的上限
            throw new Error("Maximum lock count
exceeded");
        // 设置新的state，锁重入成功
        setState(nextc);
        return true;
    }
    // 获取失败，回到调用处
    return false;
}
```

> 获取锁失败，尝试将结点加入队列尾部

```
// ㈣ AQS 继承过来的方法，方便阅读，放在此处
private Node addWaiter(Node mode) {
    // 将当前线程关联到一个 Node 对象上，模式为独占模式
    Node node = new Node(Thread.currentThread(),
mode);
    // 如果 tail 不为 null，cas 尝试依次将 Node 对象加入
AQS 队列尾部
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
```

```
                // 双向链表
                pred.next = node;
                return node;
            }
        }
        // 尝试将 Node 对象加入 AQS 队列尾部失败，进入 ㈥，循环尝试
        enq(node);
        return node;
    }

    // ㈥ AQS 继承过来的方法，方便阅读，放在此处
    private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            if (t == null) {
                // 还没有，设置 head 为哨兵节点（不对应线程，状态为 0）
                if (compareAndSetHead(new Node())) {
                    tail = head;
                }
            } else {
                // cas 尝试将 Node 对象加入 AQS 队列尾部
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

最后尝试获取锁

```
    // ㈤ AQS 继承过来的方法，方便阅读，放在此处
```

```
final boolean acquireQueued(final Node node, int
arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            // 前一个结点
            final Node p = node.predecessor();
            // 上一个节点是 head，表示轮到自己（当前线程对
应的 node）了，尝试获取
            if (p == head && tryAcquire(arg)) {
                // 获取成功，设置自己（当前线程对应的
node）为 head
                setHead(node);
                // 上一个节点 help GC
                p.next = null;
                failed = false;
                // 返回中断标记 false
                return interrupted;
            }
            if (
                // 判断是否应当 park线程，进入 ⑦
                shouldParkAfterFailedAcquire(p,
node) &&
                // park 等待，此时 Node 的状态被置为
Node.SIGNAL ⑧
                parkAndCheckInterrupt()
            ) {
                interrupted = true;
            }
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

判断是否应该park

```
// (七) AQS 继承过来的方法, 方便阅读, 放在此处
private static boolean
shouldParkAfterFailedAcquire(Node pred, Node node) {
    // 获取上一个节点的状态
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL) {
        // 上一个节点都在阻塞,并且其有责任唤醒当前线程, 那么
当前线程也阻塞好了
        return true;
    }
    // > 0 表示前一个结点是取消状态
    if (ws > 0) {
        // 前一个结点取消, 那么重构删除前面所有取消的节点,
返回到外层循环重试
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 这次没有park线程, 设置上一个节点状态为
Node.SIGNAL, 下一次获取锁失败就可以成功park
        compareAndSetWaitStatus(pred, ws,
Node.SIGNAL);
    }
    return false;
}

// (八) 阻塞当前线程
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

8.3.4.2、非公平锁解锁流程

> 解锁代码

```
// 解锁实现
public void unlock() {
    sync.release(1);
}

// AQS 继承过来的方法, 方便阅读, 放在此处
public final boolean release(int arg) {
    // 尝试释放锁, 进入 ㈠
    if (tryRelease(arg)) {
        // 队列头节点 unpark
        Node h = head;
        if (
            // 队列不为 null
            h != null &&
            // waitStatus == Node.SIGNAL 才需要 unpark
            h.waitStatus != 0
        ) {
            // unpark AQS 中等待的线程, 进入 ㈡
            unparkSuccessor(h);
        }
        return true;
    }
    return false;
}
```

> 尝试释放锁

```
// ㈠ Sync 继承过来的方法, 方便阅读, 放在此处
protected final boolean tryRelease(int releases) {
    // state--
    int c = getState() - releases;
```

```
        // 当前线程没有获取锁的情况下释放锁，报异常
        if (Thread.currentThread() ≠ getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();

        boolean free = false;
        // 支持锁重入，只有 state 减为 0，才释放成功
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
        setState(c);
        return free;
    }

    // ㈡ AQS 继承过来的方法，方便阅读，放在此处
    private void unparkSuccessor(Node node) {
        // 如果状态为 Node.SIGNAL 尝试重置状态为 0
        // 不成功也可以
        int ws = node.waitStatus;
        if (ws < 0) {
            compareAndSetWaitStatus(node, ws, 0);
        }
        // 找到需要 unpark 的节点，但本节点从 AQS 队列中脱离，是由唤醒节点完成的
        Node s = node.next;

        // 不考虑已取消的节点，从 AQS 队列从后至前找到队列最前面需要 unpark 的节点
        if (s == null || s.waitStatus > 0) {
            s = null;
            for (Node t = tail; t ≠ null && t ≠ node; t = t.prev)
                if (t.waitStatus ≤ 0)
                    s = t;
        }
        // 唤醒下一个线程
```

```
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

8.3.4.3、锁重入原理(获取释放锁中的state值)

```
static final class NonfairSync extends Sync {
    // ...

    // Sync 继承过来的方法, 方便阅读, 放在此处
    final boolean nonfairTryAcquire(int acquires) {
        final Thread current = 
Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        // 如果已经获得了锁, 线程还是当前线程, 表示发生了锁
重入
        else if (current == 
getExclusiveOwnerThread()) {
            // state++
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count 
exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    // Sync 继承过来的方法, 方便阅读, 放在此处
```

```
    protected final boolean tryRelease(int releases)
{
        // state--
        int c = getState() - releases;
        if (Thread.currentThread() ≠
getExclusiveOwnerThread())
            throw new
IllegalMonitorStateException();
        boolean free = false;
        // 支持锁重入，只有 state 减为 0，才释放成功
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
        setState(c);
        return free;
    }
}
```

8.3.4.4、可打断原理 (interrupt)

> 不可打断模式

在此模式下，**即使它被打断，仍会驻留在 AQS 队列中**，一直要等到获得锁后方能得知自己被打断了。

```
// Sync 继承自 AQS
static final class NonfairSync extends Sync {
    // ...

    private final boolean parkAndCheckInterrupt() {
        // 如果打断标记已经是 true，则 park 会失效
        LockSupport.park(this);
        // interrupted 会清除打断标记
        return Thread.interrupted();
    }

```

```
    final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null;
                    failed = false;
                    // 还是需要获得锁后, 才能返回打断状态
                    return interrupted;
                }
                if (
                    shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt()
                ) {
                    // 如果是因为 interrupt 被唤醒, 返回打断状态为 true
                    interrupted = true;
                }
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    public final void acquire(int arg) {
        if (
            !tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg)
        ) {
            // 如果打断状态为 true
```

```
            selfInterrupt();
        }
    }

    static void selfInterrupt() {
        // 重新产生一次中断
        Thread.currentThread().interrupt();
    }
}
```

> 可打断模式

遇到打断之后，抛出异常。

```
static final class NonfairSync extends Sync {
    public final void acquireInterruptibly(int arg)
throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        // 如果没有获得到锁，进入 ㈠
        if (!tryAcquire(arg))
            doAcquireInterruptibly(arg);
    }

    // ㈠ 可打断的获取锁流程
    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()) {
                    // 在 park 过程中如果被 interrupt
会进入此
                    // 这时候抛出异常，而不会再次进入 for
(;;)
                    throw new
InterruptedException();
                }
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
}
```

8.3.4.5、公平锁实现原理

hasQueuedPredecessors()先检查 AQS 队列中是否有前驱节点，没有才去竞争

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

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

    // AQS 继承过来的方法，方便阅读，放在此处
    public final void acquire(int arg) {
        if (
            !tryAcquire(arg) &&
```

```
            acquireQueued(addWaiter(Node.EXCLUSIVE),
arg)
        ) {
            selfInterrupt();
        }
    }

    // 与非公平锁主要区别在于 tryAcquire 方法的实现
    protected final boolean tryAcquire(int acquires)
{
        final Thread current =
Thread.currentThread();
        int c = getState();
        if (c == 0) {
            // 先检查 AQS 队列中是否有前驱节点，没有才去竞
争
            if (!hasQueuedPredecessors() &&
                compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current ==
getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0)
                throw new Error("Maximum lock count
exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    // (二) AQS 继承过来的方法，方便阅读，放在此处
    public final boolean hasQueuedPredecessors() {
        Node t = tail;
```

```
        Node h = head;
        Node s;
        // h != t 时表示队列中有 Node
        return h != t &&
            (
            // (s = h.next) == null 表示队列中还有没有老二
            (s = h.next) == null ||
            // 或者队列中老二线程不是此线程
            s.thread != Thread.currentThread()
        );
    }
}
```

8.4、ReentrantReadWriteLod的使用

- **当读操作远远高于写操作时**，这时候使用 读写锁 让 读-读 可以并发，提高性能。
- 类似于数据库中的 select ... from ... lock in share mode

提供一个 数据容器类 内部分别使用读锁保护数据的 read() 方法，写锁保护数据的 write() 方法

```
private Object data;
private ReentrantReadWriteLock rw = new ReentrantReadWriteLock();
private ReentrantReadWriteLock.ReadLock r = rw.readLock();
private ReentrantReadWriteLock.WriteLock w = rw.writeLock();

public Object read() throws InterruptedException {
    System.out.println(("获取读锁..."));
    r.lock();
    try {
        System.out.println(("读取"));
```

```
            sleep(1);
            return data;
        } finally {
            System.out.println(("释放读锁..."));
            r.unlock();
        }
    }

    public void write() throws InterruptedException {
        System.out.println(("获取写锁..."));
        w.lock();
        try {
            System.out.println(("写入"));
            sleep(1);
        } finally {
            System.out.println(("释放写锁..."));
            w.unlock();
        }
    }
```

测试 读锁-读锁 可以并发

```
    public static void readToRead(String[] args) {
        DataContainer dataContainer = new
    DataContainer();

        new Thread(() -> {
            try {
                dataContainer.read();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }, "t1").start();
```

```
    new Thread(() -> {
        try {
            dataContainer.read();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }, "t2").start();
}
```

测试 读锁(写锁)-写锁 相互阻塞

```
public static void writeToRead() {
    DataContainer dataContainer = new
DataContainer();

    new Thread(() -> {
        try {
            dataContainer.read();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }, "t1").start();

    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    new Thread(() -> {
        try {
            dataContainer.write();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }, "t2").start();
}
```

```
获取读锁...
读取
释放读锁...
获取写锁...
写入
释放写锁...
```

> 读写锁实现一致性缓存

为了避免不必要的数据库查询，实现采用了"双重检查锁定"的模式。**queryOne方法中，当缓存未命中时，第一次获取读锁检查是为了降低获取写锁的竞争；第二次在获取写锁后再检查一次，是为了防止其他线程已经查询过并更新了缓存。**这可以在一定程度上降低不必要的数据库查询。

```
class GenericCachedDao<T> {
    // HashMap 作为缓存非线程安全，需要保护
    HashMap<SqlPair, T> map = new HashMap<>();

    ReentrantReadWriteLock lock = new
ReentrantReadWriteLock(); // 读写锁
    GenericDao genericDao = new GenericDao();

    public int update(String sql, Object... params)
{
        SqlPair key = new SqlPair(sql, params);
        // 加写锁，防止其它线程对缓存读取和更改
        lock.writeLock().lock();
        try {
            int rows = genericDao.update(sql,
params);
            map.clear();
            return rows;
        } finally {
            lock.writeLock().unlock();
        }
    }
```

```
    public T queryOne(Class<T> beanClass, String sql, Object... params) {
        SqlPair key = new SqlPair(sql, params);
        // 加读锁, 防止其它线程对缓存更改
        lock.readLock().lock();
        try {
            T value = map.get(key);
            if (value != null) {
                return value;
            }
        } finally {
            lock.readLock().unlock();
        }
        // 加写锁, 防止其它线程对缓存读取和更改
        lock.writeLock().lock();
        try {
            // get 方法上面部分是可能多个线程进来的, 可能已经向缓存填充了数据
            // 为防止重复查询数据库, 再次验证
            T value = map.get(key);
            if (value == null) {
                // 如果没有, 查询数据库
                value = genericDao.queryOne(beanClass, sql, params);
                map.put(key, value);
            }
            return value;
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

8.5、Semaphore的使用

Semaphore 是用来管理许可证的，线程在调用 acquire() 方法时，如果没有许可证，那么线程就会阻塞等待，直到有许可证时才能继续执行。可以轻松的实现「限流」功能。

> 具体案例

接下来，咱们使用代码的方式来演示 Semaphore 的使用。我们以停车场的限流为例，**假设整个停车场只有 2 个车位（车位虽少，但足矣说明问题），但来停车的却有 4 辆车，显然车位不够用了，此时需要保证停车场最多只能有 2 辆车**，接下来咱们使用 Semaphore 来实现车辆的限流功能，具体实现代码如下：

```
package aqs;

import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class SemaphoreTest {

    // 创建信号量
    static Semaphore semaphore = new Semaphore(2);
    static String red = "\u001B[31m";
    static String green = "\u001B[32m";
    static String yellow = "\u001B[33m";
    public static void main(String[] args) {

        // 创建 5 个固定的线程数
        ExecutorService threadPool =
Executors.newFixedThreadPool(4);

        // 定义执行任务
        MyThread myThread = new MyThread();
```

```
        for (int i = 0; i < 4; i++) {
            threadPool.execute(myThread);
        }

        // 等线程池任务执行完之后关闭
        threadPool.shutdown();
    }

    static class MyThread implements Runnable {
        @Override
        public void run() {
            String tName =
Thread.currentThread().getName();
            System.out.println(yellow +
String.format("老司机：%s，停车场外排队，时间：%s",
tName, new Date()));

            // 模拟在尝试获取资源之前的操作
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }

            // 尝试获取资源
            try {
                semaphore.acquire();
                System.out.println(red +
String.format("老司机：%s，已进入停车场，时间：%s",
tName, new Date()));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

            // 模拟持有资源之后的操作
```

```
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                } finally {
                    // 注意，这里没有检查是否成功获取资源，因为只有成功获取后才会进入try块
                    System.out.println(green + String.format("老司机：%s，离开停车场，时间：%s",
                            tName, new Date()));
                    semaphore.release(); // 释放资源
                }
            }
        }


}
```

从上述的结果我们可以看出，当有 5 辆车同时需要进入停车场时，因为停车场的停车位只有 2 个，所以停车场最多只能容纳 2 辆车。此时我们通过 Semaphore 的 acquire 方法（阻塞等待）和 release 方法（颁发一个证书）顺利的实现了限流的功能，让停车场的车辆数始终控制在 2 辆车以下。

引入 `acquired` 布尔变量来跟踪是否成功获取了锁。只有在成功获取锁之后（即 `acquired` 被设置为 `true`），`finally` 块中的 `release()` 方法才会被调用。这样，就可以避免在未成功获取锁的情况下尝试释放锁，从而避免了潜在的问题。

> 关于公平模式和非公平模式

```
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}
public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new
NonfairSync(permits);
}
```

如果想用公平模式就可以使用第二个构造函数 Semaphore(int permits, **boolean fair**)，将 fair 值设置为 true 就是公平模式来获取证书了。

8.6、CountDownLatch的使用

CountDownLatch 在创建的时候需要传入一个整数，在这个整数“倒数”到 0 之前，主线程需要一直挂起等待，直到其他的线程都执行之后，主线才能继续执行。 countDownLatch.countDown() (线程池) ;countDownLatch.await() (主线程)

```
public class CountDownLatchTest {

    // 创建 CountDownLatch
    static CountDownLatch countDownLatch = new
CountDownLatch(2);
    // 使用 AtomicInteger 保证线程安全的任务序号增加
    static AtomicInteger taskNumber = new
AtomicInteger(1);

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

        // 创建固定线程数的线程池
        ExecutorService executorService =
Executors.newFixedThreadPool(2);
        MyThread myThread = new MyThread();
        for (int i = 0; i < 2; i++) {
```

```
            executorService.execute(myThread);
        }

        // 等待任务执行完成
        countDownLatch.await();
        System.out.println("程序执行完成~");
        executorService.shutdown();
    }

    static class MyThread implements Runnable {

        @Override
        public void run() {
            try {
                // 模拟任务执行时间
                Thread.sleep(1000);
                // 安全地获取并增加任务序号
                int currentTaskNumber =
taskNumber.getAndIncrement();

 System.out.println(Thread.currentThread().getName()
+ "我是任务"
                                        +
currentTaskNumber + ", 我开始执行了~");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                // 通知 CountDownLatch 一个任务已经完成
                countDownLatch.countDown();
            }
        }
    }
}

pool-1-thread-2我是任务1, 我开始执行了~
pool-1-thread-1我是任务2, 我开始执行了~
程序执行完成~
```

CountDownLatch 的实现是在其内部创建并维护了一个 volatile 类型的整数计数器，当调用 countDown() 方法时，会尝试将整数计数器 -1，当调用 wait() 方法时，当前线程就会判断整数计数器是否为 0，如果为 0，则继续往下执行，如果不为 0，则使当前线程进入等待状态，直到某个线程将计数器设置为 0，才会唤醒在 await() 方法中等待的线程继续执行。

> 常用方法

```
// 线程（一般主）被挂起直到 count 值为 0 才继续执行
public void await() throws InterruptedException { };

// 和 await() 类似，只不过等待一定的时间后 count 值还没变为 0 的话就会继续执行
public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };

// 将 count 值减 1
public void countDown() { };
```

> 应用之同步等待多个远程调用结束

```
@RestController
public class TestCountDownlatchController {
    @GetMapping("/order/{id}")
    public Map<String, Object> order(@PathVariable int id) {
        HashMap<String, Object> map = new HashMap<>();
        map.put("id", id);
        map.put("total", "2300.00");
        sleep(2000);
        return map;
    }
```

```
    @GetMapping("/product/{id}")
    public Map<String, Object> product(@PathVariable
int id) {
        HashMap<String, Object> map = new HashMap<>
();
        if (id == 1) {
            map.put("name", "小爱音箱");
            map.put("price", 300);
        } else if (id == 2) {
            map.put("name", "小米手机");
            map.put("price", 2000);
        }
        map.put("id", id);
        sleep(1000);
        return map;
    }

    @GetMapping("/logistics/{id}")
    public Map<String, Object>
logistics(@PathVariable int id) {
        HashMap<String, Object> map = new HashMap<>
();
        map.put("id", id);
        map.put("name", "中通快递");
        sleep(2500);
        return map;
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
RestTemplate restTemplate = new RestTemplate();
log.debug("begin");
ExecutorService service =
Executors.newCachedThreadPool();
CountDownLatch latch = new CountDownLatch(4);
Future<Map<String,Object>> f1 = service.submit(() ->
{
    Map<String, Object> r =
restTemplate.getForObject("http://localhost:8080/order/{1}", Map.class, 1);
    return r;
});
Future<Map<String, Object>> f2 = service.submit(() ->
{
    Map<String, Object> r =
restTemplate.getForObject("http://localhost:8080/product/{1}", Map.class, 1);
    return r;
});
Future<Map<String, Object>> f3 = service.submit(() ->
{
    Map<String, Object> r =
restTemplate.getForObject("http://localhost:8080/product/{1}", Map.class, 2);
    return r;
});
Future<Map<String, Object>> f4 = service.submit(() ->
{
    Map<String, Object> r =
restTemplate.getForObject("http://localhost:8080/logistics/{1}", Map.class, 1);
    return r;
});
```

```
System.out.println(f1.get());
System.out.println(f2.get());
System.out.println(f3.get());
System.out.println(f4.get());
log.debug("执行完毕");
service.shutdown();
```

8.7、CyclicBarrier 的使用

CyclicBarrier 是一个同步辅助类，允许一组线程相互等待，直到所有线程都达到公共屏障点 (Barrier Point) ，然后继续执行。使用 CyclicBarrier 的好处在于它允许线程集合在继续执行前等待对方，这在模拟一些并行处理任务时非常有用，比如并行计算、游戏中的多玩家准备等场景。

barrier.await();

> **作为单一栅栏使用，等待所有线程到齐再往下。**

```
public class CyclicBarrierTest {
    private static CyclicBarrier barrier;
    private static SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss:SSS");
    static class GatherAction implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(new Random().nextInt(4) * 1000);
 System.out.println(formatter.format(new Date()) + " : " +
Thread.currentThread().getName() + " 到达，等待其他线程到齐");
                //开始拦截线程，等待所有线程到齐才往下
```

```
                barrier.await();

                System.out.println(formatter.format(new Date()) + " : " +
                        Thread.currentThread().getName() + " 开始工作");
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        barrier = new CyclicBarrier(4);  // 设定计数器计数为 10
        ExecutorService executorService = Executors.newFixedThreadPool(4); // 创建一个固定大小的线程池
        for (int i = 0; i < 4; i++) {
            executorService.submit(new GatherAction());
        }
        executorService.shutdown(); // 关闭线程池
    }
}
```

> **作为可重复栅栏使用，等待所有线程到齐再往下**

完成屏障动作（如果有的话）或者当最后一个线程到达时，CyclicBarrier 会自动重置其内部计数器到初始状态。

```
public class CyclicBarrierTest_reuse {

    private static CyclicBarrier barrier;
```

```
    private static SimpleDateFormat formatter = new
SimpleDateFormat("HH:mm:ss:SSS");

    static class GatherAction implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(new Random().nextInt(3)
* 1000);

 System.out.println(formatter.format(new Date()) + "
: " +

Thread.currentThread().getName() + " 到达, 等待其他线程
到齐");
                barrier.await(); //开始拦截线程, 等待所
有线程到齐才往下


 System.out.println(formatter.format(new Date()) + "
: " +

Thread.currentThread().getName() + " 开始工作");
                Thread.sleep(new Random().nextInt(3)
* 1000);

                barrier.await();

 System.out.println(formatter.format(new Date()) + "
: " +

Thread.currentThread().getName() + " 工作已完成");
            } catch (InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
```

```
    }

    public static void main(String[] args) throws
InterruptedException, BrokenBarrierException {
        barrier = new CyclicBarrier(3);  // 设定计数器
计数为 10
        ExecutorService executorService =
Executors.newFixedThreadPool(3);
        GatherAction action = new GatherAction(); //
实例化 Runnable 对象
        for (int i = 0; i < 3; i++) {
            executorService.submit(action);
        }
        executorService.shutdown(); // 关闭线程池
        System.out.println("======== 所有任务已完成
========");
    }
}
```

```
18:02:31:453 : pool-1-thread-2 到达，等待其他线程到齐
18:02:33:457 : pool-1-thread-3 到达，等待其他线程到齐
18:02:33:457 : pool-1-thread-1 到达，等待其他线程到齐
18:02:33:457 : pool-1-thread-1 开始工作
18:02:33:457 : pool-1-thread-3 开始工作
18:02:33:457 : pool-1-thread-2 开始工作
18:02:35:465 : pool-1-thread-1 工作已完成
18:02:35:465 : pool-1-thread-3 工作已完成
18:02:35:465 : pool-1-thread-2 工作已完成
```

1. **第一次调用 `await()`**：当所有10个线程都到达第一次调用 `await()` 的地方时，屏障打开，允许所有线程继续执行。这时，如果有一个屏障动作（BarrierAction）定义，它会被执行。在您的代码中，没有定义屏障动作，所以线程继续执行下一行代码。
2. **执行工作**：每个线程模拟工作，通过 `Thread.sleep()` 暂停一段随机时间。
3. **第二次调用 `await()`**：所有线程在完成其“工作”后，再次调用 `await()`。同样地，当所有线程都到达这个点时，`CyclicBarrier`

再次打开，允许所有线程继续执行。这就完成了 CyclicBarrier 的一次复用。

8.8、ConcurrentHashMap使用及原理

8.8.1、单词计数实例

> 构造txt

List subList(int fromIndex, int toIndex): **此列表中指定范围的视图（截取一个范围）。**

```
public static void generateFiles() {
    int length = ALPHA.length();
    int count = 200;
    List<String> list = new ArrayList<>(length * count);

    for (int i = 0; i < length; i++) {
        char ch = ALPHA.charAt(i);
        for (int j = 0; j < count; j++) {
            list.add(String.valueOf(ch));
        }
    }

    Collections.shuffle(list);

    for (int i = 0; i < 5; i++) {
        try (PrintWriter out = new PrintWriter(
            new OutputStreamWriter(
                Files.newOutputStream
(Paths.get("D:\\programmingWorkspace\\JavaSpace\\JavaMultiThread\\src\\aqs\\"
                                                    + (i + 1) + ".txt"))))) {
```

```
            String collect = String.join("\n",
list.subList(i * count, (i + 1) * count));
            out.print(collect);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

> 多线程读取txt文件

你要做的是实现两个参数

- 一是提供一个 map 集合，用来存放每个单词的计数结果，key 为单词，value 为计数
- 二是提供一组操作，保证计数的安全性，会传递 map 集合以及 单词 List

函数式接口

首先，这段代码使用了Java的函数式编程特性，通过Lambda表达式来传递行为。在这个例子中，`Supplier`和`BiConsumer`是Java中的内置函数式接口。

- `Supplier<T>`：这是一个函数式接口，它提供了一个`get()`方法，用于返回一个泛型结果。在这个例子中，它被用来提供一个新的`Map<String,V>`实例。
- `BiConsumer<T, U>`：这是**一个接收两个输入参数但不返回结果的操作**。在这个例子中，它被用来定义如何处理一个`Map<String,V>`和一个`List<String>`的行为。

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
```

```
    T get();
}

@FunctionalInterface
public interface BiConsumer<T, U> {

    /**
     * Performs this operation on the given
arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     */
    void accept(T t, U u);

    /**
     * Returns a composed {@code BiConsumer} that
performs, in sequence, this
     * operation followed by the {@code after}
operation. If performing either
     * operation throws an exception, it is relayed
to the caller of the
     * composed operation.  If performing this
operation throws an exception,
     * the {@code after} operation will not be
performed.
     *
     * @param after the operation to perform after
this operation
     * @return a composed {@code BiConsumer} that
performs in sequence this
     * operation followed by the {@code after}
operation
     * @throws NullPointerException if {@code after}
is null
     */
```

```
    default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {
        Objects.requireNonNull(after);

        return (l, r) -> {
            accept(l, r);
            after.accept(l, r);
        };
    }
}
```

泛型方法，接受两个参数：一个Supplier和一个BiConsumer。这个方法的目的是提供一个灵活的方式来创建一个Map（通过Supplier），然后使用这个Map和一个单词列表来执行一些操作（通过BiConsumer）。

```
public static void main(String[] args) {
    demo(
        // 创建 map 集合
        // 创建 ConcurrentHashMap 对不对?
        new Supplier<Map<String, Integer>>() {
            @Override
            public Map<String, Integer> get() {
                return new ConcurrentHashMap<>(); // HashMap
            }
        },
        // 进行计数
        new BiConsumer<Map<String, Integer>, List<String>>() {
            @Override
            public void accept(Map<String, Integer> map, List<String> words) {
                for (String word : words) {
                    //这里的getter和setter无法保证原子性
                    Integer counter = map.get(word);
                    int newValue = counter == null ? 1 : counter + 1;
```

```
                    map.put(word, newValue);
                }
            }
        }
    );
}

private static <V> void demo(Supplier<Map<String,V>>
supplier,
                             
BiConsumer<Map<String,V>,List<String>> consumer) {
    Map<String, V> counterMap = supplier.get();
    List<Thread> ts = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
        int idx = i;
        Thread thread = new Thread(() -> {
            List<String> words = readFromFile(idx);
            consumer.accept(counterMap, words);
        });
        ts.add(thread);
    }
    ts.forEach(Thread::start);

    ts.forEach(t-> {
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    System.out.println(counterMap);
}

// 读取文件，并加入到列表中
public static List<String> readFromFile(int i) {
    ArrayList<String> words = new ArrayList<>();
```

```
    try (BufferedReader in = new BufferedReader(new
InputStreamReader(
            Files.newInputStream(
                    Paths.get("D:\\programmingWorkspace\\JavaSpace\\JavaMultiThread\\src\\aqs\\" + i
                            + ".txt"))))) {
        while(true) {
            String word = in.readLine();
            if(word == null) {
                break;
            }
            words.add(word);
        }
        return words;
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

在这里包含三个操作，只能说get和put方法是原子的，合起来就不是了，所以即使这里使用concurrentHashMap也不行

```
Integer counter = map.get(word);
int newValue = counter == null ? 1 : counter + 1;
map.put(word, newValue);
```

正确使用concurrentHashMap

```
demo(
    new Supplier<Map<String, LongAdder>>() {
        @Override
        public Map<String, LongAdder> get() {
            return new ConcurrentHashMap<>();
        }
    }
```

```
    ,
    new BiConsumer<Map<String, LongAdder>,
List<String>>() {
        @Override
        public void accept(Map<String, LongAdder>
map, List<String> words) {
            for (String word : words) {
                LongAdder adder =
map.computeIfAbsent(word, new Function<String,
LongAdder>() {
                    @Override
                    public LongAdder apply(String
word) {
                        return new LongAdder();
                    }
                });
                adder.increment();
            }
        }
    }


);
```

- **computeIfAbsent方法**：这个方法是原子的，对于给定的键，如果键不存在，就会使用给定的映射函数计算其值，并将其插入到映射中（保证这个连续操作是原子的）。这保证了即使有多个线程尝试同时计算某个键的值，该键的值也只会被计算一次，并且在整个过程中映射的一致性得到了保护。
- LongAdder提供了线程安全的increment方法（保证计数是原子的），用于累加计数，保证了在并发修改的情况下计数的准确性。

8.8.2、JDK 8 ConcurrentHashMap

分段锁：ConcurrentHashMap被分成多个段（Segment），每个段实质上是一个独立的HashMap，并拥有自己的锁。当线程访问某个段中的元素时，只需要获取这个段的锁，这样允许多线程可以同时访问不同段的数据，从而减少锁的竞争，提高并发性能。

8.8.3、JDK 8 ConcurrentHashMap

Java 8 数组（Node） + （ 链表 Node | 红黑树 TreeNode ） 以下数组简称（table），链表简称（bin）

- 初始化，使用 **cas懒惰初始化 table**
- 树化，**当 table.length < 64 时，先尝试扩容，超过 64 时，并且 bin.length > 8 时，会将链表树化，树化过程会用 synchronized 锁住链表头**
- put，**如果该 bin 尚未创建，只需要使用 cas 创建 bin；如果已经有了，synchronized 锁住链表头进行后续 put 操作，元素添加至 bin 的尾部**
- get，**无锁操作仅需要保证可见性（volatile）**，扩容过程中 get 操作拿到的是 ForwardingNode 它会让 get 操作在新 table 进行搜索
- 扩容，扩容时以 bin 为单位进行，需要**对 bin 进行 synchronized**，但这时妙的是其它竞争线程也不是无事可做，它们会帮助把其它 bin 进行扩容。
- 计数，**没有竞争发生，向 baseCount（volatile） 累加计数**。有竞争发生，新建 counterCells，向其中的一个 cell 累加计数
-
  - counterCells 初始有两个 cell
  - 如果计数竞争比较激烈，会创建新的 cell 来累加计数
  - **最后计算出来的不是精确值，而是大概的值**

8.9、ThreadLocal的使用

ThreadLocal适用于每一个线程需要自己独立实例，而且实例的话需要在多个方法里被使用到，也就是变量在线程之间是隔离的但是在方法或者是类里面是共享的场景。同一个变量，不同线程自己设定值。

```
public class ThreadLocalTest {

    private static final ThreadLocal<String>
localVar = new ThreadLocal<>();

    static void print(String str) {
        //打印当前线程中本地内存中变量的值
        System.out.println(str + " :" +
localVar.get());
        //清除内存中的本地变量
        localVar.remove();
    }
    public static void main(String[] args) throws
InterruptedException {

        new Thread(() -> {
            localVar.set("xdclass_A");
            print("A");
            //打印本地变量
            System.out.println("清除后: " +
localVar.get());
        },"A").start();
        Thread.sleep(1000);

        new Thread(() -> {
            localVar.set("xdclass_B");
            print("B");
            System.out.println("清除后 " +
localVar.get());
        },"B").start();
    }
}
```

> ThreadLocal核心应用的场景介绍

ThreaLocal作用在每个线程内都都需要独立的保存信息，这样就方便同一个线程的其他方法获取到该信息的场景，**由于每一个线程获取到的信息可能都是不一样的，前面执行的方法保存了信息之后，后续方法可以通过ThreadLocal可以直接获取到，避免了传参**，这个类似于全局变量的概念。比如像**用户登录令牌解密后的信息传递、用户权限信息。**

```
#用户微服务配置token解密信息传递例子
public static ThreadLocal<LoginUser> threadLocal =
new ThreadLocal<>();
                LoginUser loginUser = new
LoginUser();
                loginUser.setId(id);
                loginUser.setName(name);
                loginUser.setMail(mail);
                loginUser.setHeadImg(headImg);
                threadLocal.set(loginUser);
#后续想直接获取到直接threadLocal.getxxx就可以了
```

8.10、阻塞队列BlockingQueue

用了两把锁和dummy节点

- 用两把锁，同一时刻，可以允许两个线程同时（一个生产者与一个消费者）执行
- - 消费者与消费者线程仍然串行
  - 生产者与生产者线程仍然串行
- **putLock 保证的是 last 节点的线程安全，takeLock 保证的是 head 节点的线程安全**。两把锁保证了入队和出队没有竞争

```
// 用于 put(阻塞) offer(非阻塞)
private final ReentrantLock putLock = new
ReentrantLock();

// 用户 take(阻塞) poll(非阻塞)
private final ReentrantLock takeLock = new
ReentrantLock();
```

8.10.1、put操作

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // 计算节点数量
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    // count 用来维护元素计数
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        // 满了等待
        while (count.get() == capacity) {
            // 倒过来读就好：等待 notFull
            notFull.await();
        }
        // 有空位，入队且计数加一
        enqueue(node);
        c = count.getAndIncrement();
        // 除了自己 put 以外，队列还有空位，由自己叫醒其他 put 线程
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    // 如果队列中有一个元素，叫醒 take 线程
    if (c == 0)
        // 这里调用的是 notEmpty.signal() 而不是 notEmpty.signalAll() 是为了减少竞争
        signalNotEmpty();
}
```

> 关于c + 1 < capacity

这个条件检查是在元素成功入队之后执行的，用来决定是否需要唤醒其他可能在等待队列非满条件（notFull）的线程。此时，**c是count.getAndIncrement()操作之前的队列元素计数。因此，c + 1实际上是元素入队之后的队列大小。**

> 关于c == 0

c == 0意味着在当前元素入队之前，队列是空的（因为c是入队操作之前的计数）。这是检查在此次操作之前队列是否为空，如果是，这意味着此次操作后队列中至少有一个元素，**因此需要唤醒正在等待队列非空（notEmpty）的消费者线程。**

8.10.2、take操作

```
public E take() throws InterruptedException {
    E x;
    // 计算节点数量
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    // 如果队列中只有一个空位时，叫醒 put 线程
    // 如果有多个线程进行出队，第一个线程满足 c == capacity，但后续线程 c < capacity
    if (c == capacity)
```

```
        // 这里调用的是 notFull.signal()，唤醒正在等待队列没有满（`notFull`）的生产者线程
        signalNotFull()
        return x;
    }
```

notFull.signal()，唤醒正在等待队列没有满（notFull）的生产者线程。

9、实例：若依-异步任务管理器

> SysLoginService.java

```
public String login(String username, String password, String code, String uuid){
    ...
    AsyncManager.me().execute(AsyncFactory.recordLogininfor(username, Constants.LOGIN_FAIL, MessageUtils.message("user.password.not.match")));
    ...
}
```

在登录的业务层方法中，通过**异步任务管理器**来记录登录日志。

- AsyncManager.me()获取一个AsyncManager对象。
- 执行execute方法，执行任务，传入的是一个**task对象，实现了Runnable接口，是一个任务，由线程Thread去执行。**

> AsyncFactory.java

```
/**
 * @return 返回任务对象task，交给线程池处理
```

```
    */
public static TimerTask recordLogininfor(final
String username, final String status,
                                                final
String message,final Object... args)
{
    ...查找信息
        return new TimerTask()
    {
        @Override
        public void run()
        {
            ...封装信息
                // 插入数据,但并不是真的执行，而是把任务交
给线程去执行
                
 SpringUtils.getBean(ISysLogininforService.class).in
sertLogininfor(logininfor);
        }
    };
}
```

异步任务管理器，内部定义了一个线程池，然后根据业务创建添加日志的任务，交给线程池去处理，这样做到了日志和业务的抽象，解耦合，日志全部统一处理。

recordLogininfor()返回的是TimerTask定时任务，交给**定时任务调度线程池scheduledExecutorService**，他通过在**异步任务管理器类AsyncManager**中用getBean()从IOC容器中获取。

AsyncManager.me()是单例模式的应用。

> AsyncManager.java

```
public class AsyncManager
{
    ...
```

```
    /**
     * 异步操作任务调度线程池
     */
    private ScheduledExecutorService executor = SpringUtils.getBean("scheduledExecutorService");

    /**
     * 单例模式
     */
    private AsyncManager(){}
    private static AsyncManager me = new AsyncManager();
    public static AsyncManager me()
    {
        return me;
    }

    /**
     * 执行任务
     *
     * @param task 任务
     */
    public void execute(TimerTask task)
    {
        executor.schedule(task, OPERATE_DELAY_TIME, TimeUnit.MILLISECONDS);
    }

    /**
     * 停止任务线程池
     */
    public void shutdown()
    {
        Threads.shutdownAndAwaitTermination(executor);
    }
}
```

在线程池配置类ThreadPoolConfig定义了**执行周期性或定时任务的线程池。**

> **ThreadPoolConfig.java**

```
/**
     * 执行周期性或定时任务
     */
    @Bean(name = "scheduledExecutorService")
    protected ScheduledExecutorService
scheduledExecutorService()
    {
        return new
ScheduledThreadPoolExecutor(corePoolSize,
                new
BasicThreadFactory.Builder().namingPattern("schedule
-pool-%d").daemon(true).build(),
                new
ThreadPoolExecutor.CallerRunsPolicy())
            // 这个策略会在调用者的线程中直接运行新的任务。
这样做可以降低向线程池提交任务的速率。
        {
            @Override
            protected void afterExecute(Runnable r,
Throwable t)
            {
                super.afterExecute(r, t);
                Threads.printException(r, t);
            }
        };
    }
```

`protected void afterExecute(Runnable r, Throwable t)`：这个方法是`ThreadPoolExecutor`的一个钩子（hook）方法，它在每个任务执行完毕后被调用。在这个重写的`afterExecute`方法中，首先调用了`super.afterExecute(r, t)`以继承父类的行为，然后调用**`Threads.printException(r, t)`来处理执行过程中抛出的异常。**

Threads.java

```
/**
 * 线程相关工具类.
 *
 * @author Manigoat
 */
public class Threads
{
    /**
     * 打印线程异常信息
     */
    public static void printException(Runnable r, Throwable t)
    {
        if (t == null && r instanceof Future<?>)
        {
            try
            {
                Future<?> future = (Future<?>) r;
                if (future.isDone())
                {
                    future.get(); // 这类任务的异常不会在普通的执行流程中直接抛出来，需要显示的获取异常
                }
            }
            catch (CancellationException ce)
            {
                t = ce;
            }
            catch (ExecutionException ee)
            {
                t = ee.getCause();
            }
            catch (InterruptedException ie)
            {
```

```
                Thread.currentThread().interrupt();
            }
        }
        if (t != null)
        {
            logger.error(t.getMessage(), t);
        }
    }
}
```

printException方法的目的是打印出在执行线程或任务时发生的异常信息。这个方法特别处理了实现了Future<?>接口的Runnable任务，因为这类任务的异常不会在普通的执行流程中直接抛出来，而是需要通过Future.get()方法来显式地检查和抛出。

1. 首先检查传入的Throwable t是否为null，并且Runnable r是否是Future<?>的实例。如果是，**尝试通过future.get()获取任务的结果，以此触发可能的异常。**
2. 如果在调用future.get()时捕获到CancellationException、ExecutionException或InterruptedException异常，则分别处理这些异常。特别地，**ExecutionException通常包装了任务执行中的实际异常，因此通过ee.getCause()获取记录这个原始异常。**
3. 如果Throwable t不为null（无论是直接传入的，还是从Future中获取的），则使用日志记录器logger打印异常信息和堆栈跟踪。

—