Java 多线程

面试题

1. 一个线程两次调用start()方法会出现什么情况？为什么

两次调用start()方法会报异常

start()方法执行后，会立刻会线程当前状态进行检查，线程**必须**从NEW状态开始，直到编程TERMINATED状态，因为TERMINATED状态和NEW状态只能出现一次，故第二次调用start()方法会报错

2. 既然start()方法会调用run()方法，为什么我们选择调用start()方法，而不是直接调用run()方法呢?

开启一个线程必须通过start()方法，通过调用start()方法，当前进程会创建一个子线程；直接调用run()方法并不会创建线程，而是当前线程中执行run()方法中的内容，并不会创建一个新的线程

一、基本概念

1.1 进程与线程

1.1.1 进程

- 程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备。进程就是用来加载指令、管理内存、管理 IO 的。
- 当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。
- 进程就可以视为程序的一个实例。大部分程序可以同时运行多个实例进程（例如记事本、画图、浏览器 等），也有的程序只能启动一个实例进程（例如网易云音乐、360 安全卫士等）

1.1.2 线程

- 一个进程之内可以分为一到多个线程。
- 一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给 CPU 执行 。
- Java 中，线程作为小调度单位，进程作为资源分配的最小单位。 在windows 中进程是不活动的，只是作为线程的容器

1.1.3 两者对比

- 进程基本上相互独立的，而线程存在于进程内，是进程的一个子集 进程拥有共享的资源，如内存空间等，供其内部的线程共享
    - 进程间通信较为复杂 同一台计算机的进程通信称为 IPC（Inter-process communication）
    - 不同计算机之间的进程通信，需要通过网络，并遵守共同的协议，例如 HTTP
- 线程通信相对简单，因为它们共享进程内的内存，一个例子是多个线程可以访问同一个共享变量 线程更轻量，线程上下文切换成本一般上要比进程上下文切换低

1.2 并发与并行

并发是一个CPU在不同的时间去不同线程中执行指令。
并行是多个CPU同时处理不同的线程。
引用 Rob Pike 的一段描述：

- 并发（concurrent）是同一时间应对（dealing with）多件事情的能力
- 并行（parallel）是同一时间动手做（doing）多件事情的能力

1.3 应用

1.3.1 同步和异步的概念

以调用方的角度讲，如果

- 需要等待结果返回才能继续运行的话就是**同步**
- 不需要等待就是**异步**

1.3.2 设计

多线程可以使方法的执行变成异步的，比如说读取磁盘文件时，假设读取操作花费了5秒，如果没有线程的调度机制，那么 cpu 只能等 5 秒，啥都不能做。

1.3.3 应用场景

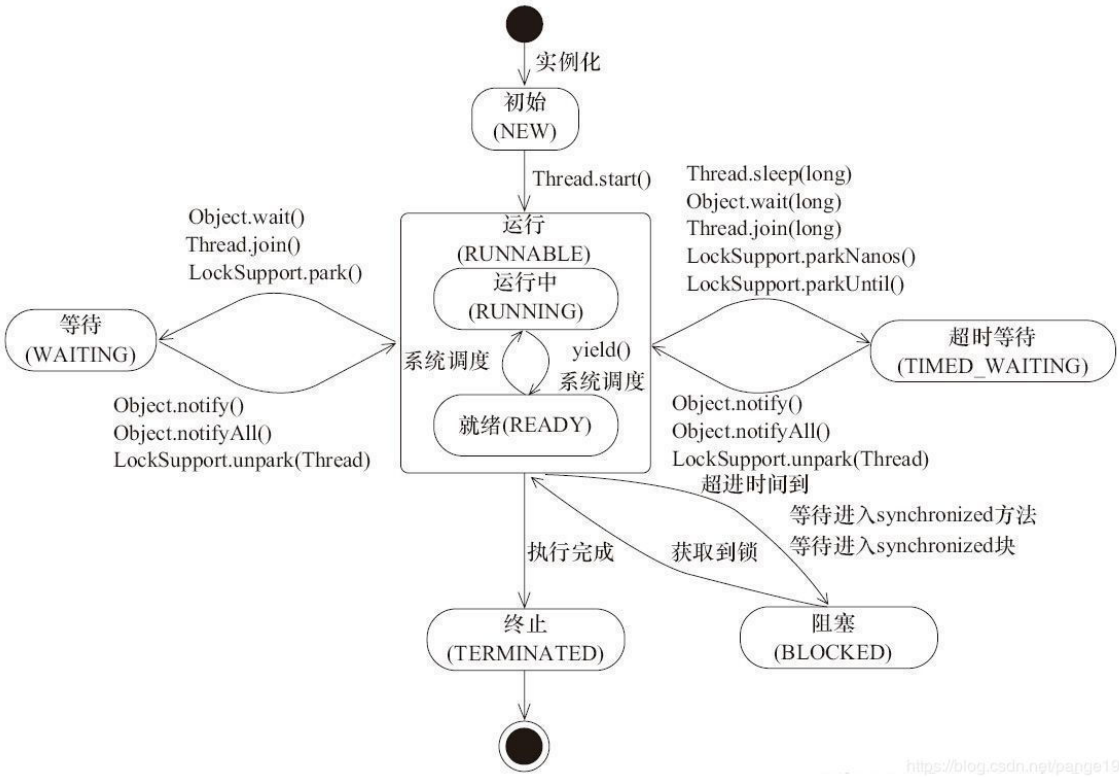
- 比如在项目中，视频文件需要转换格式等操作比较费时，这时开一个新线程处理视频转换，避免阻塞主线程
- tomcat 的异步 servlet 也是类似的目的，让用户线程处理耗时较长的操作，避免阻塞 tomcat 的工作线程
- ui 程序中，开线程进行其他操作，避免阻塞 ui 线程

1.3.4 总结

- 单核 cpu 下，多线程不能实际提高程序运行效率，只是为了能够在不同的任务之间切换，不同线程轮流使用 cpu ，不至于一个线程总占用 cpu，别的线程没法干活
- 多核 cpu 可以并行跑多个线程，但能否提高程序运行效率还是要分情况的
    - 有些任务，经过精心设计，将任务拆分，并行执行，当然可以提高程序的运行效率。但不是所有计算任 务都能拆分（参考后文的【阿姆达尔定律】）
    - 也不是所有任务都需要拆分，任务的目的如果不同，谈拆分和效率没啥意义
- IO 操作不占用 cpu，只是我们一般拷贝文件使用的是**阻塞 IO**，这时相当于线程虽然不用 cpu，但需要一 直等待 IO 结束，没能充分利用线程。所以才有后面的**非阻塞 IO**和**异步 IO**优化

1.4 线程状态

状态名称	状态
New	初始状态，线程被创建，但是还没有调用start()方法
RUNNABLE	运行状态，Java线程将操作系统中的就绪（READY）和运行（RUNNING）两种状态笼统称为“运行中”
BLOCKED	阻塞状态，标示线程阻塞于锁，也就是无法获得锁
WAITING	等待状态，标示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定的动作（通知或者中断）
TIMED_WAITING	超时等待状态，该状态不同于WAITING，它是可以在指定的时间自行返回的（执行了sleep方法）
TERMINATED	终止状态，表示当前线程已经执行完毕



1.5 同步和异步

需要等待结果返回，才能继续运行的是**同步**

不需要等待结果返回，就能继续运行的就是**异步**

二、Java线程

2.1 线程创建

（1）方法一，直接创建Thread子对象，也就是通过继承，重写Thread类中run方法的方式

```
@Test
public void testCreateThreadByExtendsThread() {
    Thread thread = new Thread() {
        @Override
        public void run() {
            log.debug("running");
        }
    };

    thread.setName("t1");
    thread.start();

    log.debug("running");
}
```

（2）方法二，使用 Runnable 配合 Thread

```
@Test
public void testCreateThreadByImplementsRunnable() {
    Thread thread = new Thread(() -> {
        log.debug("running");
    });

    thread.setName("t2");
    thread.start();

    log.debug("running");
}
```

2.2 两种创建线程方式的区别

2.2.1 直接使用Thread类原理

这种创建方式的代码如下（使用了Lambda表达式进行简化）

```
Thread thread = new Thread() {
    @Override
    public void run() {
        log.debug("running");
    }
};
```

其重写了Thread类中的run方法

2.2.2 使用Runnable配合Thread原理

这个方法向Thread构造器传入了一个Runnable对象

```
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

其中init方法代码如下：

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize) {
    init(g, target, name, stackSize, null, true);
}
```

```
private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc,
                      boolean inheritThreadLocals) {
        if (name == null) {
            throw new NullPointerException("name cannot be
null");
        }

        this.name = name;

        Thread parent = currentThread();
        SecurityManager security = System.getSecurityManager();
        if (g == null) {
            /* Determine if it's an applet or not */
```

```
            /* If there is a security manager, ask the security
manager
               what to do. */
            if (security != null) {
                g = security.getThreadGroup();
            }

            /* If the security doesn't have a strong opinion of
the matter
               use the parent thread group. */
            if (g == null) {
                g = parent.getThreadGroup();
            }
        }

        /* checkAccess regardless of whether or not threadgroup
is
           explicitly passed in. */
        g.checkAccess();

        /*
         * Do we have the required permissions?
         */
        if (security != null) {
            if (isCCLOverridden(getClass())) {

 security.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
            }
        }

        g.addUnstarted();

        this.group = g;
        this.daemon = parent.isDaemon();
        this.priority = parent.getPriority();
        if (security == null ||
isCCLOverridden(parent.getClass()))
            this.contextClassLoader =
parent.getContextClassLoader();
        else
            this.contextClassLoader = parent.contextClassLoader;
        this.inheritedAccessControlContext =
                acc != null ? acc :
AccessController.getContext();
        this.target = target;
```

```
        setPriority(priority);
        if (inheritThreadLocals && parent.inheritableThreadLocals
!= null)
            this.inheritableThreadLocals =

 ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
        /* Stash the specified stack size in case the VM cares */
        this.stackSize = stackSize;

        /* Set thread ID */
        tid = nextThreadID();
}
```

其中可以注意到下面代码

```
this.target = target;
```

可以看出，这行代码将我们传入的Runnable对象赋值给了Thread类中的私有成员变量target，其中target成员变量的定义如下：

```
/* What will be run. */
private Runnable target;
```

我们可以看出target成员变量也是Runnable对象，其中run方法代码如下：

```
@Override
public void run() {
    if (target != null) {
        target.run();
    }
}
```

此时调用了成员变量target的run方法，也就是我们自己创建的Runnable对象的run方法

所以第二种方式相对于第一种方式，第一种方式通过子类继承重写父类方法的方式来实现run方法，而第二种是将自定义Runnable对象传入给Thread类，然后调用Runnable自定义类中的自定义方法run

2.3 栈与栈帧

Java Virtual Machine Stacks （Java 虚拟机栈） 我们都知道 JVM 中由堆、栈、方法区所组成，其中栈内存是给谁用的呢？其实就是线程，每个线程启动后，虚拟 机就会为其分配一块栈内存。

- 每个栈由多个栈帧（Frame）组成，对应着每次方法调用时所占用的内存
- 每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法

2.4 线程上下文切换（Thread Context Switch）

因为以下一些原因导致 cpu 不再执行当前的线程，转而执行另一个线程的代码

- 线程的 cpu 时间片用完
- 垃圾回收
- 有更高优先级的线程需要运行
- 线程自己调用了 sleep、yield、wait、join、park、synchronized、lock 等方法

当 Context Switch 发生时，需要由操作系统保存当前线程的状态，并恢复另一个线程的状态，Java 中对应的概念就是程序计数器（Program Counter Register），它的作用是记住下一条 jvm 指令的执行地址，是线程私有的

- 状态包括程序计数器、虚拟机栈中每个栈帧的信息，如局部变量、操作数栈、返回地址等
- Context Switch 频繁发生会影响性能

2.5常见方法

方法名	static	功能说明	注意
start()		启动一个新线程，在新线程中运行 run 方法中的代码	start 方法只是让线程进入就绪状态，里面代码不一定立刻运行，只有当 CPU 将时间片分给线程时，才能进入运行状态，执行代码。每个线程的 start 方法只能调用一次，调用多次就会出现 IllegalThreadStateException
run()		新线程启动会调用的方法	如果在构造 Thread 对象时传递了 Runnable 参数，则线程启动后会调用 Runnable 中的 run 方法，否则默认不执行任何操作。但可以创建 Thread 的子类对象，来覆盖默认行为
join()		等待线程运行结束	
join(long n)		等待线程运行结束,最多等待 n 毫秒	

方法名	static	功能说明	注意
getId()		获取线程长整型的id	id 唯一
getName()		获取线程名	
setName(String)		修改线程名	
getPriority()		获取线程优先级	
setPriority(int)		修改线程优先级	java中规定线程优先级是1~10 的整数，较大的优先级能提高该线程被 CPU 调度的机率
getState()		获取线程状态	Java 中线程状态是用 6 个 enum 表示，分别为：NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
isInterrupted()		判断是否被打断	不会清除 打断标记
isAlive()		线程是否存活（还没有运行完毕）	
interrupt()		打断线程	如果被打断线程正在 sleep，wait，join 会导致被打断的线程抛出 InterruptedException，并清除打断标记（设置为false）；如果打断的正在运行的线程，则会设置打断标记，park 的线程被打断，也会设置打断标记
interrupted()	static	判断当前线程是否被打断	会清除 打断标记

方法名	static	功能说明	注意
currentThread()	static	获取当前正在执行的线程	
sleep(long n)	static	让当前执行的线程休眠n毫秒，休眠时让出cpu 的时间片给其它线程	
yield()	static	提示线程调度器让出当前线程对CPU的使用	主要是为了测试和调试

2.5.1 run方法和start方法

（1）直接调用run方法

```
@Test
public void testRunMethod() {

    new Thread(() -> {
        log.debug("running....");
    }, "t1").run();
}
```

运行结果：

```
08:58:22.693 [main] DEBUG c.Test1 - running....
```

（2）直接调用start方法

```
@Test
public void testRunMethod() {

    new Thread(() -> {
        log.debug("running....");
    }, "t1").start();
}
```

运行结果：

```
08:56:18.106 [t1] DEBUG c.Test1 - running....
```

通过观察上述两个程序的运行结果可以发现，直接调用run方法实际上是**主线程**去执行，并没有创建新的线程

而通过调用start()方法，主线程会创建新的子线程去执行相关的操作

2.5.2 sleep方法

1. 调用 sleep 会让当前线程从 `RUNNING` 进入 `TIME_WAITING` 状态（阻塞）

```
@Test
public void testSleepMethod() throws InterruptedException {

    Thread thread = new Thread(() -> {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "t1");

    thread.start();
    log.debug("t1 current state: {}", thread.getState());

    Thread.sleep(1000);
    log.debug("t1 current state: {}", thread.getState());
}
```

执行结果:

```
09:08:31.540 [main] DEBUG c.Test1 - t1 current state: RUNNABLE
09:08:32.543 [main] DEBUG c.Test1 - t1 current state:
TIMED_WAITING
```

2. 其它线程可以使用 interrupt 方法打断正在睡眠的线程，这时 sleep 方法会抛出 InterruptedException，线程状态由 `TIME_WAITING` 变成了 `RUNNABLE`，睡眠结束后的线程未必会立刻得到执行，需要等待CPU分配时间片

```
@Test
public void testInterruptSleepMethod() throws
InterruptedException {

    Thread t1 = new Thread(() -> {
        log.debug("enter sleep...");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            log.debug("wake up.....");
            e.printStackTrace();
        }
    }, "t1");

    t1.start();

    Thread.sleep(1000);
    log.debug("interrupt...");
    t1.interrupt();
}
```

运行结果：

```
09:13:51.979 [t1] DEBUG c.Test1 - enter sleep...
09:13:52.978 [main] DEBUG c.Test1 - interrupt...
09:13:52.978 [t1] DEBUG c.Test1 - wake up.....
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at
com.ecifics.concurrent.part3.CreateThreadTest.lambda$testInterrup
tSleepMethod$6(CreateThreadTest.java:109)
    at java.lang.Thread.run(Thread.java:748)
```

3. 建议用 TimeUnit 的 sleep 代替 Thread 的 sleep 来获得更好的可读性

```
@Test
public void testTimeUnit() throws InterruptedException {
    log.debug("enter sleep...");
    TimeUnit.SECONDS.sleep(1);
    log.debug("wake up...");
}
```

2.5.3 yield方法

1. 调用 yield 会让当前线程从 `RUNNING` 进入 `READY` 就绪状态，然后调度执行其它线程
2. 具体的实现依赖于操作系统的任务调度器，如果当前没有其他进程需要使用 cpu，那么当前进程即使调用了yield方法，也会继续运行

2.5.4 线程优先级

- 线程优先级会提示（hint）调度器优先调度该线程，**但它仅仅是一个提示，调度器可以忽略它**
- 如果 cpu 比较忙，那么优先级高的线程会获得更多的时间片，但 cpu 闲时，优先级几乎没作用
- 默认优先级为5，优先级范围为1~10

2.5.5 join方法

如果在主线程中，子线程调用join方法，则主线程需要等待子线程执行完毕再继续执行

哪个线程调用join方法，就会等哪个线程先执行

例如下面代码如果没有使用join方法：

```
static int r = 0;
public static void main(String[] args) throws
InterruptedException {
    test1();
}
private static void test1() throws InterruptedException {
    log.debug("开始");
    Thread t1 = new Thread(() -> {
        log.debug("开始");
```

```
        sleep(1);
        log.debug("结束");
        r = 10;
    });
    t1.start();
    log.debug("结果为:{}", r);
    log.debug("结束");
}
```

打印结果为 `r=0` ，而不是期望的 `r=10` ，因为主线程并没有等待t1执行完毕再继续执行，故需要调用join方法，等待t1线程执行完毕

```
static int r = 0;
public static void main(String[] args) throws
InterruptedException {
    test1();
}
private static void test1() throws InterruptedException {
    log.debug("开始");
    Thread t1 = new Thread(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        r = 10;
    });
    t1.start();

    t1.join();

    log.debug("结果为:{}", r);
    log.debug("结束");
}
```

修改后，t1调用join方法，主线程需要等待t1线程执行完毕之后，再继续执行，最后打印结果 `r=10`

join方法可以传入一个参数，设置最长等待时间，单位为毫秒，如果还未到最长等待时间而线程结束，那么主线程也不会在继续等待目标线程，而是继续 执行

2.5.6 Interrupt方法

如果被打断线程正在 sleep，wait，join 会导致被打断的线程抛出 InterruptedException，并清除打断标记（设置为false）；如果打断的正在运行的线程，则会设置打断标记为true，park 的线程被打断，也会设置打断标记，但是并不会停止目标进程的运行

正确打断进程方式

```
@Slf4j
public class TwoPhaseTermination {

    private Thread monitor;

    public void start() {
        monitor = new Thread(() -> {
            Thread currentThread = Thread.currentThread();
            while (true) {
                if (currentThread.isInterrupted()) {
                    log.debug("料理后事");
                    break;
                }

                try {
                    Thread.sleep(1000);
                    log.debug("执行监控记录");
                } catch (InterruptedException e) {
                    currentThread.interrupt();
                    e.printStackTrace();
                }
            }
        });

        monitor.start();
    }

    public void stop() {
        monitor.interrupt();
    }

    public static void main(String[] args) throws
InterruptedException {
        TwoPhaseTermination twoPhaseTermination = new
TwoPhaseTermination();
```

```
        twoPhaseTermination.start();

        Thread.sleep(3500);
        twoPhaseTermination.stop();
    }
}
```

2.5.7 isInterrupted()和interrupted()方法

两个都可以判断当前线程是否被打断，但是isInterrupted不会修改打断标记，而interrupted在返回当前进程状态后会将清除打断标记，也就是将其设置为false

2.5.8 守护线程

默认情况下，Java 进程需要等待所有线程都运行结束，才会结束。有一种特殊的线程叫做守护线程，只要其它非守护线程（比如下面的main进程）运行结束了，即使守护线程的代码没有执行完，也会强制结束

```
@Test
public void testDaemonThread() throws InterruptedException {
    log.debug("开始运行...");
    Thread t1 = new Thread(() -> {
        log.debug("开始运行...");

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        log.debug("运行结束...");
    }, "daemon");

    t1.setDaemon(true);
    t1.start();
    Thread.sleep(2000);
    log.debug("运行结束...");
}
```

> 注：
>
> - 垃圾回收器是一种守护线程

> - Tomcat 中的 Acceptor 和 Poller 线程都是守护线程，所以 Tomcat 接收到 shutdown 命令后，不会等待它们处理完当前请求

三、线程安全

四、 Monitor 概念

八、 park & unpack

8.1 基本使用

- park/unpark都是LockSupport类中的的方法
- 先调用unpark后,再调用park, 此时park不会暂停线程

```
// 暂停当前线程
LockSupport.park();
// 恢复某个线程的运行
LockSupport.unpark(thread);
```

8.2 特点

与Object的wait和notify相比

- wait，notify和notifyAll必须配合Object Monitor一起使用，而park和unpark不需要
- park和unpark是以线程为单位来阻塞和唤醒线程，而notify只能随机唤醒一个等待线程，notifyAll是唤醒所有等待线程，就不那么精确
- park和unpark可以先unpark，而wait和notify不能先notify

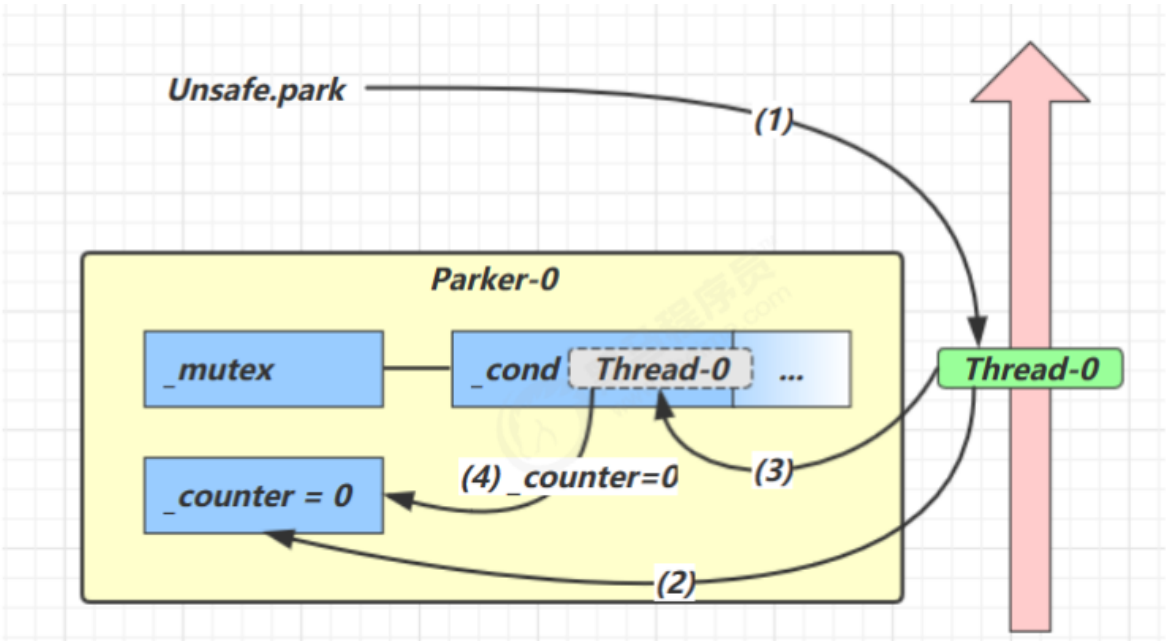
8.3 park、 unpark 原理

每个线程都有自己的一个Parker 对象，由三部分组成 `_counter` ， `_cond` 和 `_mutex`

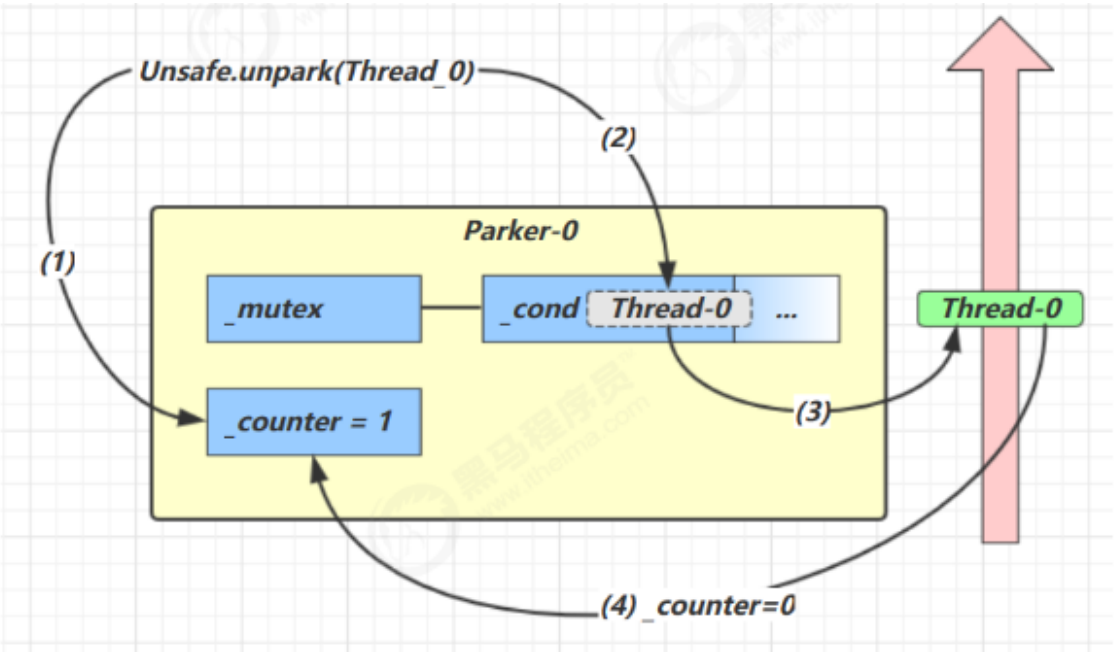
- 打个比喻线程就像一个旅人，Parker 就像他随身携带的背包，条件变量 _ cond 就好比背包中的帐篷。_counter 就好比背包中的备用干粮（0 为耗尽，1 为充足）
- 调用 park 就是要看需不需要停下来歇息
  - 如果备用干粮耗尽，那么钻进帐篷歇息
  - 如果备用干粮充足，那么不需停留，继续前进
- 调用 unpark，就好比令干粮充足
  - 如果这时线程还在帐篷，就唤醒让他继续前进
  - 如果这时线程还在运行，那么下次他调用 park 时，仅是消耗掉备用干粮，不需停留继续前进
  - 因为背包空间有限，**多次调用 unpark 仅会补充一份备用干粮**

8.4 先调用park再调用upark的过程

- 先调用park的情况
  - 当前线程调用 Unsafe.park() 方法
  - 检查 _counter, 本情况为0, 这时, 获得 `_mutex 互斥锁` (mutex对象有个等待队列 _cond)
  - 线程进入 _cond 条件变量阻塞
  - 设置 `_counter = 0` (没干粮了)

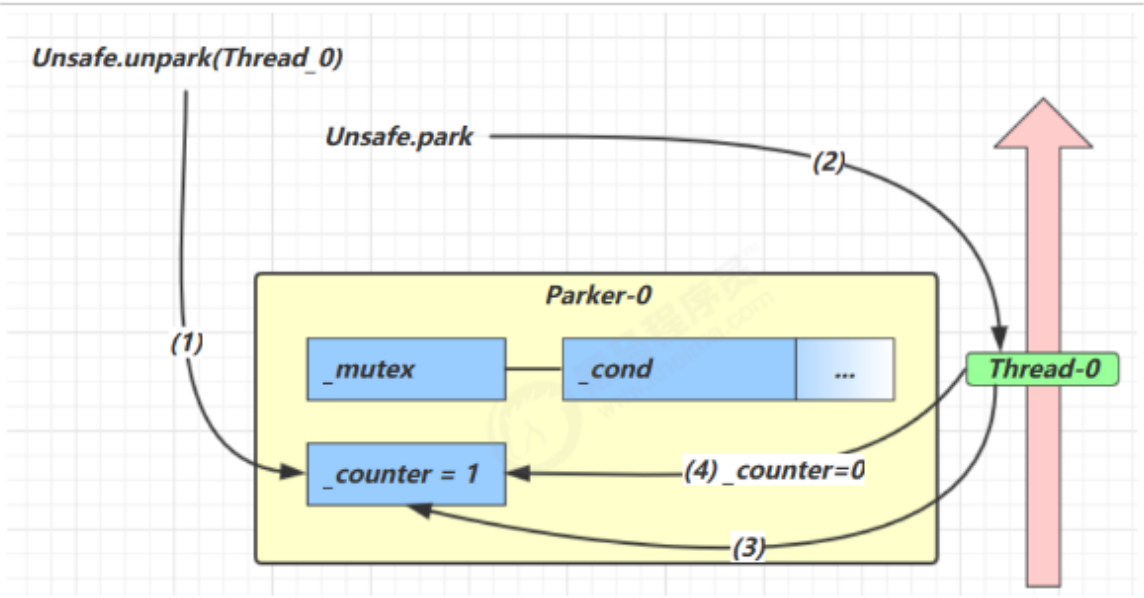


- 调用unpark
  - 调用Unsafe.unpark(Thread_0)方法，设置 `_counter为1`
  - 唤醒 _cond 条件变量中的 Thread_0
  - Thread_0 恢复运行
  - 设置 _counter 为 0



8.5 先调用upark再调用park的过程

- 调用 Unsafe.unpark(Thread_0)方法，设置 `_counter 为 1`
- 当前线程调用 Unsafe.park() 方法
- 检查 `_counter` ，本情况为 1，这时线程 无需阻塞，继续运行
- 设置 _counter 为 0



-

十二、 ReentrantLock

12.1 ReentrantLock的特点 (synchronized不具备的)

- 支持锁重入
  - 可重入锁是指同一个线程如果首次获得了这把锁，那么因为它是这把锁的拥有者，因此 **有权利再次获取这把锁**
- 可中断
  - `lock.lockInterruptibly()` ：可以被其他线程打断的中断锁
- 可以设置超时时间
  - `lock.tryLock(时间)` ：尝试获取锁对象, 如果超过了设置的时间, 还没有获取到锁, 此时就退出阻塞队列, 并释放掉自己拥有的锁
- 可以设置为公平锁
  - (先到先得) 默认是非公平, true为公平 new ReentrantLock(true)
- 支持多个条件变量(**有多个waitset**)
  - (可避免虚假唤醒) - lock.newCondition()创建条件变量对象; 通过条件变量对象调用 await/signal方法, 等待/唤醒

基本用法

```
//获取ReentrantLock对象
private ReentrantLock lock = new ReentrantLock();
//加锁
lock.lock();
try {
    //需要执行的代码
} finally {
    //释放锁
    lock.unlock();
}
```

12.2 ReentrantLock特点详解

12.2.1支持锁重入

- 可重入锁是指同一个线程如果首次获得了这把锁，那么因为它是这把锁的拥有者，因此 **有权利再次获取这把锁**
- 如果是不可重入锁，那么第二次获得锁时，自己也会被锁挡住

```
/**
```

```
 * Description: ReentrantLock 可重入锁，同一个线程可以多次获得锁对象
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.ReentrantTest")
public class ReentrantTest {

    private static ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        // 如果有竞争就进入`阻塞队列`，一直等待着,不能被打断
        lock.lock();
        try {
            log.debug("entry main...");
            m1();
        } finally {
            lock.unlock();
        }
    }

    private static void m1() {
        lock.lock();
        try {
            log.debug("entry m1...");
            m2();
        } finally {
            lock.unlock();
        }
    }

    private static void m2() {
        log.debug("entry m2....");
    }
}

13:54:29.324 guizy.ReentrantTest [main] - entry main...
13:54:29.326 guizy.ReentrantTest [main] - entry m1...
13:54:29.326 guizy.ReentrantTest [main] - entry m2....
```

12.2.2 可中断 (针对于lockInterruptibly()方法获得的中断锁) 直接退出阻塞队列, 获取锁失败

> `synchronized` 和 `reentrantlock.lock()` 的锁, 是不可被打断的; 也就是说别的线程已经获得了锁, 我的线程就需要一直等待下去. 不能中断
>
> - 可被中断的锁, 通过 `lock.lockInterruptibly()` 获取的锁对象, 可以通过调用**阻塞线程的interrupt()方法**

- 如果某个线程处于阻塞状态，可以调用其interrupt方法让其停止阻塞，获得锁失败
  - 处于阻塞状态的线程，被打断了就不用阻塞了，直接停止运行
- 可中断的锁, 在一定程度上可以被动的减少**死锁**的概率, 之所以被动, 是因为我们需要手动调用阻塞线程的interrupt方法;

测试使用 `lock.lockInterruptibly()` 可以从阻塞队列中,打断

```
/**
 * Description: ReentrantLock, 演示RenntrantLock中的可打断锁方法 lock.lockInterruptibly();
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.ReentrantTest")
public class ReentrantTest {

    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            log.debug("t1线程启动...");
            try {
                // lockInterruptibly()是一个可打断的锁，如果有锁竞争在进入阻塞队列后,可以通过interrupt进行打断
                lock.lockInterruptibly();
            } catch (InterruptedException e) {
                e.printStackTrace();
                log.debug("等锁的过程中被打断"); //没有获得锁就被打断跑出的异常
                return;
            }
            try {
```

```
                log.debug("t1线程获得了锁");
            } finally {
                lock.unlock();
            }
        }, "t1");

        // 主线程获得锁(此锁不可打断)
        lock.lock();
        log.debug("main线程获得了锁");
        // 启动t1线程
        t1.start();
        try {
            Sleeper.sleep(1);
            t1.interrupt();             //打断t1线程
            log.debug("执行打断");
        } finally {
            lock.unlock();
        }
    }
}

14:18:09.145 guizy.ReentrantTest [main] - main线程获得了锁
14:18:09.148 guizy.ReentrantTest [t1] - t1线程启动...
14:18:10.149 guizy.ReentrantTest [main] - 执行打断
14:18:10.149 guizy.ReentrantTest [t1] - 等锁的过程中被打断
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireIn
terruptibly(AbstractQueuedSynchronizer.java:898)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInte
rruptibly(AbstractQueuedSynchronizer.java:1222)
    at
java.util.concurrent.locks.ReentrantLock.lockInterruptibly(Reentr
antLock.java:335)
    at
com.guizy.reentrantlock.ReentrantTest.lambda$main$0(ReentrantTest
.java:25)
    at java.lang.Thread.run(Thread.java:748)
```

测试使用 `lock.lock()` 不可以从阻塞队列中打断, 一直等待别的线程释放锁

```
@Slf4j(topic = "guizy.ReentrantTest")
public class ReentrantTest {
```

```
    private static final ReentrantLock lock = new
ReentrantLock();

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            log.debug("t1线程启动...");
                lock.lock();
            try {
                log.debug("t1线程获得了锁");
            } finally {
                lock.unlock();
            }
        }, "t1");

        // 主线程获得锁(此锁不可打断)
        lock.lock();
        log.debug("main线程获得了锁");
        // 启动t1线程
        t1.start();
        try {
            Sleeper.sleep(4);
            t1.interrupt();             //打断t1线程
            log.debug("main线程执行打断");
        } finally {
            lock.unlock();
        }
    }
}
```

- **lock()锁不能被打断**，在主线程中调用t1.interrupt()， 没用，当主线程释放锁之后，t1获得了锁

```
14:21:01.329 guizy.ReentrantTest [main] - main线程获得了锁
14:21:01.331 guizy.ReentrantTest [t1] - t1线程启动...
14:21:05.333 guizy.ReentrantTest [main] - main线程执行打断
14:21:05.333 guizy.ReentrantTest [t1] - t1线程获得了锁
```

12.2.3 锁超时 (lock.tryLock()) 直接退出阻塞队列, 获取锁失败

防止无限制等待, 主动减少死锁

- 使用 `lock.tryLock()` 方法会返回获取锁是否成功。如果成功则返回true，反之则返回false。

- 并且 `tryLock方法` 可以设置**指定等待时间**，参数为： `tryLock(long timeout, TimeUnit unit)` ，其中timeout为最长等待时间，TimeUnit为时间单位

> 获取锁的过程中, 如果超过等待时间, 或者被打断, 就直接从阻塞队列移除, 此时获取锁就失败了, 不会一直阻塞着 ! (可以用来实现死锁问题)

- 不设置等待时间, 立即失败

```
/**
 * Description: ReentrantLock, 演示RenntrantLock中的tryLock(), 获取锁立即失败
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.ReentrantTest")
public class ReentrantTest {

    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            log.debug("尝试获得锁");
            // 此时肯定获取失败, 因为主线程已经获得了锁对象
            if (!lock.tryLock()) {
                log.debug("获取立刻失败, 返回");
                return;
            }
            try {
                log.debug("获得到锁");
            } finally {
                lock.unlock();
            }
        }, "t1");

        lock.lock();
        log.debug("获得到锁");
        t1.start();
        // 主线程2s之后才释放锁
        Sleeper.sleep(2);
        log.debug("释放了锁");
        lock.unlock();
    }
```

```
}

14:52:19.726 guizy.WaitNotifyTest [main] - 获得到锁
14:52:19.728 guizy.WaitNotifyTest [t1] - 尝试获得锁
14:52:19.728 guizy.WaitNotifyTest [t1] - 获取立刻失败，返回
14:52:21.728 guizy.WaitNotifyTest [main] - 释放了锁
```

- **设置等待时间，超过等待时间还没有获得锁，失败，从阻塞队列移除该线程**

```
/**
 * Description: ReentrantLock，演示RenntrantLock中的tryLock(long mills)，超过锁设置的等待时间,就从阻塞队列移除
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.ReentrantTest")
public class ReentrantTest {

    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            log.debug("尝试获得锁");
            try {
                // 设置等待时间，超过等待时间 / 被打断，都会获取锁失败；退出阻塞队列
                if (!lock.tryLock(1, TimeUnit.SECONDS)) {
                    log.debug("获取锁超时，返回");
                    return;
                }
            } catch (InterruptedException e) {
                log.debug("被打断了，获取锁失败，返回");
                e.printStackTrace();
                return;
            }
            try {
                log.debug("获得到锁");
            } finally {
                lock.unlock();
            }
        }, "t1");

        lock.lock();
```

```
            log.debug("获得到锁");
            t1.start();
//            t1.interrupt();
            // 主线程2s之后才释放锁
            Sleeper.sleep(2);
            log.debug("main线程释放了锁");
            lock.unlock();
        }
}

// 超时的打印
14:55:56.647 guizy.WaitNotifyTest [main] - 获得到锁
14:55:56.651 guizy.WaitNotifyTest [t1] - 尝试获得锁
14:55:57.652 guizy.WaitNotifyTest [t1] - 获取锁超时，返回
14:55:58.652 guizy.WaitNotifyTest [main] - main线程释放了锁

// 中断的打印
14:56:41.258 guizy.WaitNotifyTest [main] - 获得到锁
14:56:41.260 guizy.WaitNotifyTest [main] - main线程释放了锁
14:56:41.261 guizy.WaitNotifyTest [t1] - 尝试获得锁
14:56:41.261 guizy.WaitNotifyTest [t1] - 被打断了，获取锁失败，返回
java.lang.InterruptedException
12345678910111213141516171819202122232425262728293031323334353637
38394041424344454647484950515253545 5
```

12.2.4 通过lock.tryLock()来解决, 哲学家就餐问题

`lock.tryLock(时间)` : 尝试获取锁对象, 如果超过了设置的时间, 还没有获取到锁, 此时就退出阻塞队列, 并释放掉自己拥有的锁

```
/**
 * Description: 使用了ReentrantLock锁，该类中有一个tryLock()方法，在指定时间内获取不到锁对象，就从阻塞队列移除,不用一直等待。
 *              当获取了左手边的筷子之后，尝试获取右手边的筷子，如果该筷子被其他哲学家占用，获取失败，此时就先把自己左手边的筷子，
 *              给释放掉. 这样就避免了死锁问题
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.PhilosopherEat")
public class PhilosopherEat {
    public static void main(String[] args) {
        Chopstick c1 = new Chopstick("1");
        Chopstick c2 = new Chopstick("2");
```

```
        Chopstick c3 = new Chopstick("3");
        Chopstick c4 = new Chopstick("4");
        Chopstick c5 = new Chopstick("5");
        new Philosopher("苏格拉底", c1, c2).start();
        new Philosopher("柏拉图", c2, c3).start();
        new Philosopher("亚里士多德", c3, c4).start();
        new Philosopher("赫拉克利特", c4, c5).start();
        new Philosopher("阿基米德", c5, c1).start();
    }
}

@Slf4j(topic = "guizy.Philosopher")
class Philosopher extends Thread {
    final Chopstick left;
    final Chopstick right;

    public Philosopher(String name, Chopstick left, Chopstick
right) {
        super(name);
        this.left = left;
        this.right = right;
    }

    @Override
    public void run() {
        while (true) {
            // 获得了左手边筷子（针对五个哲学家，它们刚开始肯定都可获得左筷
子)
            if (left.tryLock()) {
                try {
                    // 此时发现它的right筷子被占用了，使用tryLock()，
                    // 尝试获取失败，此时它就会将自己左筷子也释放掉
                    // 临界区代码
                    if (right.tryLock()) { //尝试获取右手边筷子，如果
获取失败，则会释放左边的筷子
                        try {
                            eat();
                        } finally {
                            right.unlock();
                        }
                    }
                } finally {
                    left.unlock();
                }
            }
```

```
            }
        }

        private void eat() {
            log.debug("eating...");
            Sleeper.sleep(0.5);
        }
    }

    // 继承ReentrantLock，让筷子类称为锁
    class Chopstick extends ReentrantLock {
        String name;

        public Chopstick(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return "筷子{" + name + '}';
        }
    }
    15:16:01.793 guizy.Philosopher [亚里士多德] - eating...
    15:16:01.795 guizy.Philosopher [苏格拉底] - eating...
    15:16:02.293 guizy.Philosopher [亚里士多德] - eating...
    15:16:02.295 guizy.Philosopher [苏格拉底] - eating...
    15:16:02.794 guizy.Philosopher [赫拉克利特] - eating...
    15:16:02.796 guizy.Philosopher [苏格拉底] - eating...
    15:16:03.294 guizy.Philosopher [赫拉克利特] - eating...
    15:16:03.296 guizy.Philosopher [柏拉图] - eating...
    15:16:03.795 guizy.Philosopher [赫拉克利特] - eating...
    15:16:03.797 guizy.Philosopher [苏格拉底] - eating...
    15:16:04.295 guizy.Philosopher [亚里士多德] - eating...
    15:16:04.297 guizy.Philosopher [苏格拉底] - eating...
    15:16:04.796 guizy.Philosopher [亚里士多德] - eating...
    15:16:04.798 guizy.Philosopher [阿基米德] - eating...
    15:16:05.296 guizy.Philosopher [柏拉图] - eating...
    15:16:05.299 guizy.Philosopher [赫拉克利特] - eating...
```

12.2.5 公平锁 new ReentrantLock(true)

什么是公平锁? 什么是非公平锁?

公平锁 (new ReentrantLock(true))

- 公平锁, 可以把竞争的线程放在一个先进先出的阻塞队列上

- 只要持有锁的线程执行完了, 唤醒阻塞队列中的下一个线程获取锁即可; 此时先进入阻塞队列的线程先获取到锁

非公平锁 (synchronized, new ReentrantLock())

- 非公平锁, 当阻塞队列中已经有等待的线程A了, 此时后到的线程B, 先去尝试看能否获得到锁对象. 如果获取成功, 此时就不需要进入阻塞队列了. 这样以来后来的线程B就先活的到锁了

> 所以公平和非公平的区别： **线程执行同步代码块时， 是否回去尝试获取锁** , 如果会尝试获取锁, 那就是非公平的, 如果不会尝试获取锁, 直接进入阻塞队列, 再等待被唤醒, 那就是公平的
>
> - 如果不进如队列呢? 线程一直尝试获取锁不就行了?
>   - 一直尝试获取锁, 在synchronized轻量级锁升级为重量级锁时, 做的一个优化, 叫做 自旋锁 , 一般很消耗资源, cpu一直空转, 最后获取锁也失败, 所以不推荐使用。在jdk6对于自旋锁有一个机制, 在重试获得锁指定次数就失败等等

- ReentrantLock默认是非公平锁, 可以指定为公平锁。
- 在线程获取锁失败，进入阻塞队列时，**先进入**的会在锁被释放后**先获得**锁。这样的获取方式就是**公平**的。一般不设置 `ReentrantLock` 为公平的, 会降低 并发度
- `Synchronized` 底层的 `Monitor锁` 就是不公平的, 和谁先进入 阻塞队列 是没有关系的。

```
//默认是不公平锁，需要在创建时指定为公平锁
ReentrantLock lock = new ReentrantLock(true);
```

12.2.6 条件变量 (可避免虚假唤醒) - lock.newCondition()创建条件变量对象; 通过条件变量对象调用await/signal方法, 等待/唤醒

- `Synchronized` 中也有条件变量，就是Monitor监视器中的 waitSet等待集合，当条件不满足时进入waitSet 等待
- ReentrantLock的条件变量比 `synchronized` 强大之处在于,它是 支持多个条件变量。

- 这就好比synchronized 是那些不满足条件的线程都在一间休息室等通知; (此时会造成虚假唤醒), 而 ReentrantLock 支持多间休息室，有专门等烟的休息室、专门等早餐的休息室、唤醒时也是按休息室来唤醒; (可以避免虚假唤醒)

使用要点:

- await 前需要 **获得锁**
- await 执行后，会释放锁，进入 `conditionObject` (条件变量)中等待
- await 的线程被唤醒（或打断、或超时）取重新竞争 lock 锁
  - 竞争 lock 锁成功后，从 await 后继续执行
- signal 方法用来唤醒条件变量(等待室)汇总的某一个等待的线程
- signalAll方法, 唤醒条件变量(休息室)中的所有线程

```
/**
 * Description: ReentrantLock可以设置多个条件变量(多个休息室)， 相对于synchronized底层monitor锁中waitSet
 *
 * @author guizy1
 * @date 2020/12/23 13:50
 */
@Slf4j(topic = "guizy.ConditionVariable")
public class ConditionVariable {
    private static boolean hasCigarette = false;
    private static boolean hasTakeout = false;
    private static final ReentrantLock lock = new ReentrantLock();
    // 等待烟的休息室
    static Condition waitCigaretteSet = lock.newCondition();
    // 等外卖的休息室
    static Condition waitTakeoutSet = lock.newCondition();

    public static void main(String[] args) {

        new Thread(() -> {
            lock.lock();
            try {
                log.debug("有烟没? [{}]", hasCigarette);
                while (!hasCigarette) {
                    log.debug("没烟，先歇会! ");
                    try {
                        // 此时小南进入到 等烟的休息室
                        waitCigaretteSet.await();
                    } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
            log.debug("烟来咯，可以开始干活了");
        } finally {
            lock.unlock();
        }
    }, "小南").start();

    new Thread(() -> {
        lock.lock();
        try {
            log.debug("外卖送到没? [{}]", hasTakeout);
            while (!hasTakeout) {
                log.debug("没外卖，先歇会！");
                try {
                    // 此时小女进入到 等外卖的休息室
                    waitTakeoutSet.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            log.debug("外卖来咯，可以开始干活了");
        } finally {
            lock.unlock();
        }
    }, "小女").start();

    Sleeper.sleep(1);
    new Thread(() -> {
        lock.lock();
        try {
            log.debug("送外卖的来咯~");
            hasTakeout = true;
            // 唤醒等外卖的小女线程
            waitTakeoutSet.signal();
        } finally {
            lock.unlock();
        }
    }, "送外卖的").start();

    Sleeper.sleep(1);
    new Thread(() -> {
        lock.lock();
        try {
```

```
                log.debug("送烟的来咯~");
                hasCigarette = true;
                // 唤醒等烟的小南线程
                waitCigaretteSet.signal();
            } finally {
                lock.unlock();
            }
        }, "送烟的").start();
    }
}
```

```
15:08:58.231 guizy.WaitNotifyTest [小南] - 有烟没? [false]
15:08:58.234 guizy.WaitNotifyTest [小南] - 没烟，先歇会!
15:08:58.235 guizy.WaitNotifyTest [小女] - 外卖送到没? [false]
15:08:58.235 guizy.WaitNotifyTest [小女] - 没外卖，先歇会!
15:08:59.232 guizy.WaitNotifyTest [送外卖的] - 送外卖的来咯~
15:08:59.233 guizy.WaitNotifyTest [小女] - 外卖来咯，可以开始干活了
15:09:00.233 guizy.WaitNotifyTest [送烟的] - 送烟的来咯~
15:09:00.234 guizy.WaitNotifyTest [小南] - 烟来咯，可以开始干活了
```

五、本章小结

本章我们需要重点掌握的是

- 分析多线程访问共享资源时，哪些代码片段属于临界区
- 使用 synchronized 互斥解决临界区的线程安全问题
  - 掌握 synchronized 锁对象语法
  - 掌握 synchronzied 加载成员方法和静态方法语法
  - 掌握 wait/notify 同步方法
- 使用 lock 互斥解决临界区的线程安全问题
  - 掌握 lock 的使用细节：可打断、锁超时、公平锁、条件变量
- 学会分析变量的线程安全性、掌握常见线程安全类的使用
- 了解线程活跃性问题：死锁、活锁、饥饿