# 并发编程（讲师：Old Jia）

## 并发编程简介

java是一个支持多线程的开发语言。多线程可以在包含多个CPU核心的机器上同时处理多个不同的任务，优化资源的使用率，提升程序的效率。在一些对性能要求比较高场合，多线程是java程序调优的重要方面。

大厂面试中比较重要的内容除了算法，就是并发编程。并发编程是最能体现一个程序员功底的方面之一。

并发编程也是在面试中很能加分的部分。

Java并发编程主要涉及以下几个部分：

1. 并发编程三要素
   原子性：即一个不可再被分割的颗粒。在Java中原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。
   有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）
   可见性：当多个线程访问同一个变量时，如果其中一个线程对其作了修改，其他线程能立即获取到最新的值。
2. 线程的五大状态
   创建状态：当用 new 操作符创建一个线程的时候
   就绪状态：调用 start 方法，处于就绪状态的线程并不一定马上就会执行 run 方法，还需要等待CPU的调度
   运行状态：CPU 开始调度线程，并开始执行 run 方法
   阻塞状态：线程的执行过程中由于一些原因进入阻塞状态比如：调用 sleep 方法、尝试去得到一个锁等等
   死亡状态：run 方法执行完 或者 执行过程中遇到了一个异常
3. 悲观锁与乐观锁
   悲观锁：每次操作都会加锁，会造成线程阻塞。
   乐观锁：每次操作不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止，不会造成线程阻塞。
4. 线程之间的协作
   线程间的协作有：wait/notify/notifyAll等
5. synchronized 关键字
   synchronized是Java中的关键字，是一种同步锁。它修饰的对象有以下几种：
   - 修饰一个代码块：被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象
   - 修饰一个方法：被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象
   - 修饰一个静态的方法：其作用的范围是整个静态方法，作用的对象是这个类的所有对象
   - 修饰一个类：其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

6. CAS
   CAS全称是Compare And Swap，即比较替换，是实现并发应用到的一种技术。操作包含三个操作数—内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。
   CAS存在三大问题：ABA问题，循环时间长开销大，以及只能保证一个共享变量的原子操作。
7. 线程池
   如果我们使用线程的时候就去创建一个线程，虽然简单，但是存在很大的问题。如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。线程池通过复用可以大大减少线程频繁创建与销毁带来的性能上的损耗。

常见的多线程面试题有：

1. 重排序有哪些分类？如何避免？
2. Java 中新的Lock接口相对于同步代码块（synchronized block）有什么优势？如果让你实现一个高性能缓存，支持并发读取和单一写入，你如何保证数据完整性。
3. 如何在Java中实现一个阻塞队列。
4. 写一段死锁代码。说说你在Java中如何解决死锁。
5. volatile变量和atomic变量有什么不同？
6. 为什么要用线程池？
7. 实现Runnable接口和Callable接口的区别
8. 执行execute()方法和submit()方法的区别是什么呢？
9. AQS的实现原理是什么？
10. java API中哪些类中使用了AQS？
11. ...

并发编程课程很多内容会从JDK源码解析相关原理。

主要内容包括：

1. 多线程&并发设计原理

   并发核心概念

   并发的问题

   JMM内存模型

2. JUC

   并发容器

   同步工具类

   Atomic类

   Lock与Condition

3. 线程池与Future

   线程池的实现原理

   线程池的类继承体系

   ThreadPoolExecutor

   Executors工具类

   ScheduledThreadPool Executor

   CompletableFuture用法

4. ForkJoinPool

   ForkJoinPool用法

   核心数据结构

   工作窃取队列

   ForkJoinPool状态控制

   Worker线程的阻塞-唤醒机制

   任务的提交过程分析

   工作窃取算法：任务的执行过程分析

   ForkJoinTask的fork/join

   ForkJoinPool的优雅关闭

5. 多线程设计模式

   Single Threaded Execution模式

   Immutable模式

   Guarded Suspension模式

   Balking模式

   Producer-Consumer模式

   Read-Write Lock模式

   Thread-Per-Message模式

   Worker Thread模式

   Future模式

# 第一部分：多线程&并发设计原理

## 1 多线程回顾

### 1.1 Thread和Runnable

#### 1.1.1 Java中的线程

创建执行线程有两种方法：

- 扩展Thread 类。
- 实现Runnable 接口。

扩展Thread类的方式创建新线程：

```
package com.lagou.concurrent.demo;

public class MyThread extends Thread {

    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName() + " 运行了");
            try {
```

```
                Thread.sleep(800);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

实现Runnable接口的方式创建线程：

```
package com.lagou.concurrent.demo;

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        while (true) {

            System.out.println(Thread.currentThread().getName() + " 运行了");

            try {
                Thread.sleep(800);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }
}
```

### 1.1.2 Java中的线程：特征和状态

1. 所有的Java 程序，不论并发与否，都有一个名为主线程的Thread 对象。执行该程序时， Java虚拟机（ JVM ）将创建一个新Thread 并在该线程中执行main()方法。这是非并发应用程序中唯一的线程，也是并发应用程序中的第一个线程。

2. Java中的线程共享应用程序中的所有资源，包括内存和打开的文件，快速而简单地共享信息。但是必须使用同步**避免数据竞争**。

3. Java中的所有线程都有一个优先级，这个整数值介于Thread.MIN_PRIORITY（1）和Thread.MAX_PRIORITY（10）之间，默认优先级是Thread.NORM_PRIORITY（5）。线程的执行顺序并没有保证，通常，较高优先级的线程将在较低优先级的钱程之前执行。

4. 在Java 中，可以创建两种线程：

- 守护线程。
- 非守护线程。

区别在于它们如何影响程序的结束。

Java程序结束执行过程的情形：

- 程序执行Runtime类的exit()方法， 而且用户有权执行该方法。
- 应用程序的所有非守护线程均已结束执行，无论是否有**正在运行的守护线程**。

守护线程通常用在作为垃圾收集器或缓存管理器的应用程序中，执行辅助任务。在线程start之前调用isDaemon()方法检查线程是否为守护线程，也可以使用setDaemon()方法将某个线程确立为守护线程。

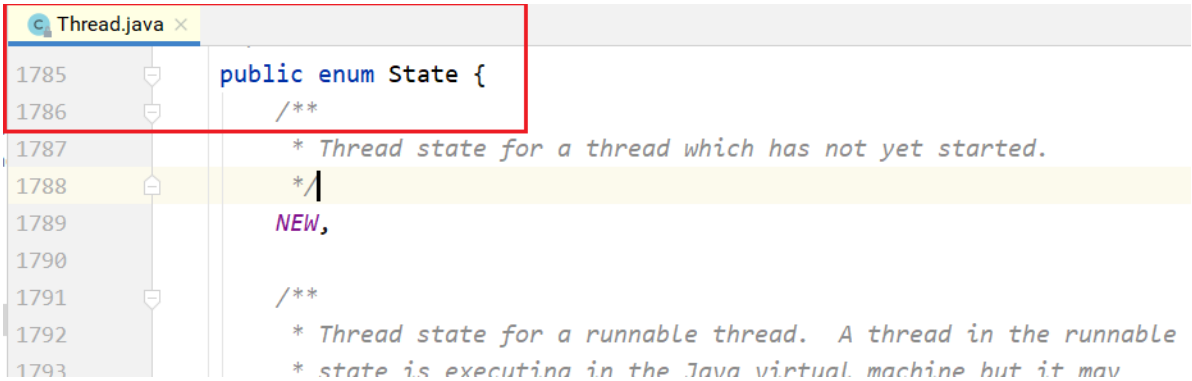
5. Thread.States类中定义线程的状态如下：

- NEW：Thread对象已经创建，但是还没有开始执行。
- RUNNABLE：Thread对象正在Java虚拟机中运行。
- BLOCKED : Thread对象正在等待锁定。
- WAITING：Thread 对象正在等待另一个线程的动作。
- TIME_WAITING：Thread对象正在等待另一个线程的操作，但是有时间限制。
- TERMINATED：Thread对象已经完成了执行。

getState()方法获取Thread对象的状态，可以直接更改线程的状态。

在给定时间内， 线程只能处于一个状态。这些状态是JVM使用的状态，不能映射到操作系统的线程状态。

线程状态的源码：




### 1.1.3 Thread类和Runnable 接口

Runnable接口只定义了一种方法：run()方法。这是每个线程的主方法。当执行start()方法启动新线程时，它将调用run()方法。

Thread类其他常用方法：

- 获取和设置Thread对象信息的方法。
    - getId()：该方法返回Thread对象的标识符。该标识符是在钱程创建时分配的一个正整数。在线程的整个生命周期中是唯一且无法改变的。
    - getName()/setName()：这两种方法允许你获取或设置Thread对象的名称。这个名称是一个String对象，也可以在Thread类的构造函数中建立。
    - getPriority()/setPriority()：你可以使用这两种方法来获取或设置Thread对象的优先级。
    - isDaemon()/setDaemon()：这两种方法允许你获取或建立Thread对象的守护条件。
    - getState()：该方法返回Thread对象的状态。
- interrupt()：中断目标线程，给目标线程发送一个中断信号，线程被打上中断标记。
- interrupted()：判断目标线程是否被中断，但是将清除线程的中断标记。
- isinterrupted()：判断目标线程是否被中断，不会清除中断标记。
- sleep(long ms)：该方法将线程的执行暂停ms时间。
- join()：暂停线程的执行，直到调用该方法的线程执行结束为止。可以使用该方法等待另一个Thread对象结束。
- setUncaughtExceptionHandler()：当线程执行出现未校验异常时，该方法用于建立未校验异常的控制器。
- currentThread()：Thread类的静态方法，返回实际执行该代码的Thread对象。

join示例程序：

```java
package com.lagou.concurrent.demo;

public class MyThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("MyThread线程: " + i);
        }
    }
}
```

```java
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        myThread.start();
        myThread.join();
        System.out.println("main线程 - 执行完成");
    }
}
```

### 1.1.4 Callable

Callable 接口是一个与Runnable 接口非常相似的接口。Callable 接口的主要特征如下。

- 接口。有简单类型参数，与call()方法的返回类型相对应。
- 声明了call()方法。执行器运行任务时，该方法会被执行器执行。它必须返回声明中指定类型的对象。
- call()方法可以抛出任何一种校验异常。可以实现自己的执行器并重载afterExecute()方法来处理这些异常。

```java
package com.lagou.concurrent.demo;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class Main {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        MyCallable myCallable = new MyCallable();

        // 设置Callable对象，泛型表示Callable的返回类型
        FutureTask<String> futureTask = new FutureTask<String>(myCallable);
        // 启动处理线程
        new Thread(futureTask).start();

```

```
        // 同步等待线程运行的结果
        String result = futureTask.get();
        // 5s后得到结果
        System.out.println(result);

    }
}
```

```
package com.lagou.concurrent.demo;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        Thread.sleep(5000);
        return "hello world call() invoked!";
    }
}
```

```
package com.lagou.concurrent.demo;

import java.util.concurrent.*;

public class Main2 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                5, 5, 1, TimeUnit.SECONDS, new ArrayBlockingQueue<>(10)
        ) {
            @Override
            protected void afterExecute(Runnable r, Throwable t) {
                super.afterExecute(r, t);
            }
        };

        Future<String> future = executor.submit(new MyCallable());

        String s = future.get();
        System.out.println(s);

        executor.shutdown();

    }
}
```

## 1.2 synchronized关键字

### 1.2.1 锁的对象

synchronized关键字“给某个对象加锁”，示例代码：

```
public Class MyClass {
    public void synchronized method1() {
        // ...
    }

    public static void synchronized method2() {
        // ...
    }
}
```

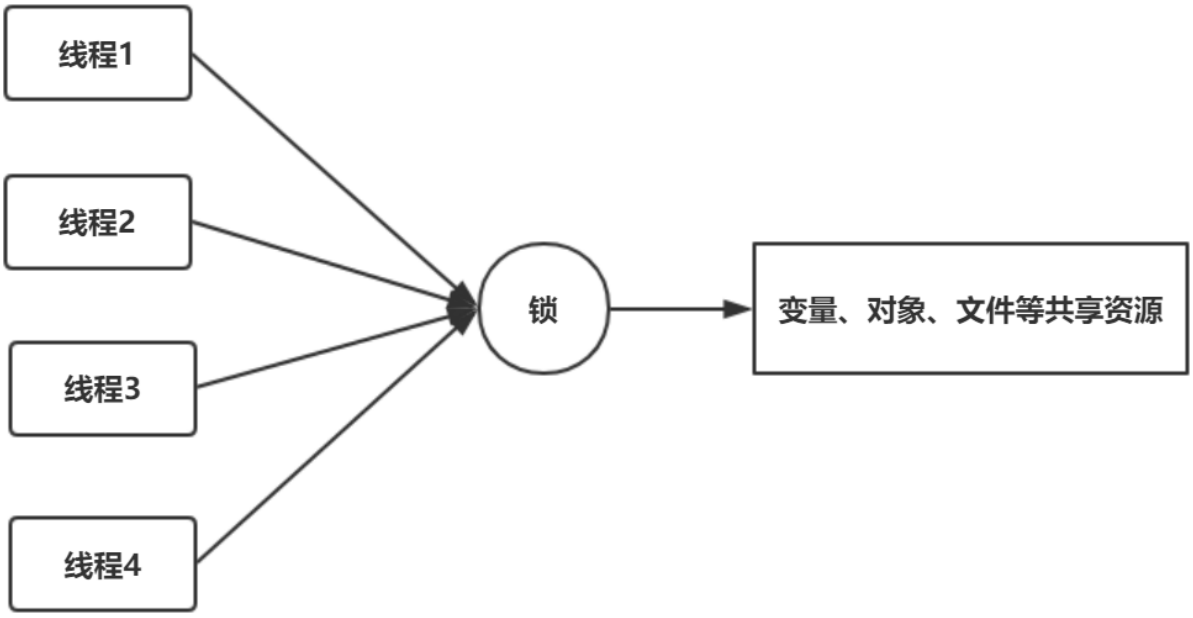
等价于：

```
public class MyClass {
    public void method1() {
        synchronized(this) {
            // ...
        }
    }

    public static void method2() {
        synchronized(MyClass.class) {
            // ...
        }
    }
}
```

实例方法的锁加在对象myClass上；静态方法的锁加在MyClass.class上。

### 1.2.2 锁的本质

如果一份资源需要多个线程同时访问，需要给该资源加锁。加锁之后，可以保证同一时间只能有一个线程访问该资源。资源可以是一个变量、一个对象或一个文件等。




锁是一个“对象”，作用如下：

1. 这个对象内部得有一个标志位（state变量），记录自己有没有被某个线程占用。最简单的情况是这个state有0、1两个取值，0表示没有线程占用这个锁，1表示有某个线程占用了这个锁。
2. 如果这个对象被某个线程占用，记录这个线程的thread ID。
3. 这个对象维护一个thread id list，记录其他所有阻塞的、等待获取拿这个锁的线程。在当前线程释放锁之后从这个thread id list里面取一个线程唤醒。

要访问的共享资源本身也是一个对象，例如前面的对象myClass，这两个对象可以合成一个对象。代码就变成synchronized(this) {...}，要访问的共享资源是对象a，锁加在对象a上。当然，也可以另外新建一个对象，代码变成synchronized(obj1) {...}。这个时候，访问的共享资源是对象a，而锁加在新建的对象obj1上。

资源和锁合二为一，使得在Java里面，synchronized关键字可以加在任何对象的成员上面。这意味着，这个对象既是共享资源，同时也具备“锁”的功能！

#### 1.2.3 实现原理

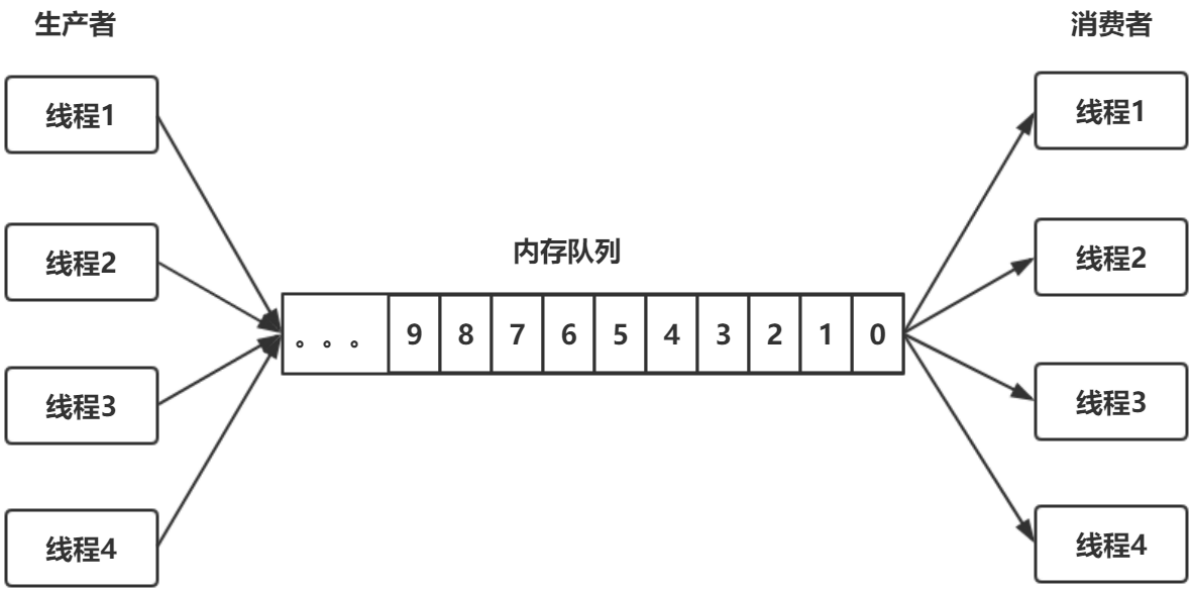
锁如何实现？

在对象头里，有一块数据叫Mark Word。在64位机器上，Mark Word是8字节（64位）的，这64位中有2个重要字段：锁标志位和占用该锁的thread ID。因为不同版本的JVM实现，对象头的数据结构会有各种差异。

## 1.3 wait与notify

#### 1.3.1 生产者–消费者模型

生产者-消费者模型是一个常见的多线程编程模型，如下图所示：




一个内存队列，多个生产者线程往内存队列中放数据；多个消费者线程从内存队列中取数据。要实现这样一个编程模型，需要做下面几件事情：

1. 内存队列本身要加锁，才能实现线程安全。
2. 阻塞。当内存队列满了，生产者放不进去时，会被阻塞；当内存队列是空的时候，消费者无事可做，会被阻塞。
3. 双向通知。消费者被阻塞之后，生产者放入新数据，要notify()消费者；反之，生产者被阻塞之后，消费者消费了数据，要notify()生产者。

第1件事情必须要做，第2件和第3件事情不一定要做。例如，可以采取一个简单的办法，生产者放不进去之后，睡眠几百毫秒再重试，消费者取不到数据之后，睡眠几百毫秒再重试。但这个办法效率低下，也不实时。所以，我们只讨论如何阻塞、如何通知的问题。

**1.如何阻塞?**

办法1：线程自己阻塞自己，也就是生产者、消费者线程各自调用wait()和notify()。

办法2：用一个阻塞队列，当取不到或者放不进去数据的时候，入队/出队函数本身就是阻塞的。

**2.如何双向通知?**

办法1：wait()与notify()机制。

办法2：Condition机制。

单个生产者单个消费者线程的情形：

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) {

        MyQueue myQueue = new MyQueue();

```

```
        ProducerThread producerThread = new ProducerThread(myQueue);
        ConsumerThread consumerThread = new ConsumerThread(myQueue);

        producerThread.start();
        consumerThread.start();

    }
}
```

```
package com.lagou.concurrent.demo;

public class MyQueue {

    private String[] data = new String[10];
    private int getIndex = 0;
    private int putIndex = 0;
    private int size = 0;

    public synchronized void put(String element) {
        if (size == data.length) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data[putIndex] = element;
        ++putIndex;
        if (putIndex == data.length) putIndex = 0;
        ++size;
        notify();
    }

    public synchronized String get() {
        if (size == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        String result = data[getIndex];
        ++getIndex;
        if (getIndex == data.length) getIndex = 0;
        --size;
        notify();
        return result;
    }

}
```

```
package com.lagou.concurrent.demo;
```

```java

import java.util.Random;

public class ProducerThread extends Thread {

    private final MyQueue myQueue;
    private final Random random = new Random();
    private int index = 0;

    public ProducerThread(MyQueue myQueue) {
        this.myQueue = myQueue;
    }

    @Override
    public void run() {
        while (true) {
            String tmp = "ele-" + index;
            myQueue.put(tmp);
            System.out.println("添加元素: " + tmp);
            index++;
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
package com.lagou.concurrent.demo;

import java.util.Random;

public class ConsumerThread extends Thread {

    private final MyQueue myQueue;
    private final Random random = new Random();

    public ConsumerThread(MyQueue myQueue) {
        this.myQueue = myQueue;
    }

    @Override
    public void run() {
        while (true) {
            String s = myQueue.get();
            System.out.println("\t\t消费元素: " + s);
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

多个生产者多个消费者的情形：

```
package com.lagou.concurrent.demo;

public class Main2 {
    public static void main(String[] args) {
        MyQueue2 myQueue = new MyQueue2();
        for (int i = 0; i < 3; i++) {
            new ConsumerThread(myQueue).start();
        }
        for (int i = 0; i < 5; i++) {
            new ProducerThread(myQueue).start();
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class MyQueue2 extends MyQueue {

    private String[] data = new String[10];
    private int getIndex = 0;
    private int putIndex = 0;
    private int size = 0;

    @Override
    public synchronized void put(String element) {
        if (size == data.length) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            put(element);
        } else {
            put0(element);
            notify();
        }
    }

    private void put0(String element) {
        data[putIndex] = element;
        ++putIndex;
        if (putIndex == data.length) putIndex = 0;
        ++size;
    }

    @Override
    public synchronized String get() {
        if (size == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            return get();
        } else {
            String result = get0();
            notify();
            return result;
        }
    }

    private String get0() {
        String result = data[getIndex];
        ++getIndex;
        if (getIndex == data.length) getIndex = 0;
        --size;
        return result;
    }

}
```

### 1.3.2 为什么必须和synchronized一起使用

在Java里面，wait()和notify()是Object的成员函数，是基础中的基础。为什么Java要把wait()和notify()放在如此基础的类里面，而不是作为像Thread一类的成员函数，或者其他类的成员函数呢?

先看为什么wait()和notify()必须和synchronized一起使用？请看下面的代码：

```
class MyClass1 {
    private Object obj1 = new Object();

    public void method1() {
        synchronized(obj1) {
            //...
            obj1.wait();
            //...
        }
    }

    public void method2() {
        synchronized(obj1) {
            //...
            obj1.notify();
            //...
        }
    }
}
```

或者下面的代码：

```
public class MyClass1 {
    public void synchronized method1() {
        //...
        this.wait();
        //...
    }

    public void synchronized method2() {
        //...
        this.notify();
        //...
    }
}
```

然后，开两个线程，线程A调用method1()，线程B调用method2()。答案已经很明显：两个线程之间要通信，对于同一个对象来说，一个线程调用该对象的wait()，另一个线程调用该对象的notify()，该对象本身就需要同步！所以，在调用wait()、notify()之前，要先通过synchronized关键字同步给对象，也就是给该对象加锁。

synchronized关键字可以加在任何对象的实例方法上面，任何对象都可能成为锁。因此，wait()和notify()只能放在Object里面了。

#### 1.3.3 为什么wait()的时候必须释放锁

当线程A进入synchronized(obj1)中之后，也就是对obj1上了锁。此时，调用wait()进入阻塞状态，一直不能退出synchronized代码块；那么，线程B永远无法进入synchronized(obj1)同步块里，永远没有机会调用notify()，发生死锁。

这就涉及一个关键的问题：在wait()的内部，会先释放锁obj1，然后进入阻塞状态，之后，它被另外一个线程用notify()唤醒，重新获取锁！其次，wait()调用完成后，执行后面的业务逻辑代码，然后退出synchronized同步块，再次释放锁。

wait()内部的伪代码如下：

```
wait() {
    // 释放锁
    // 阻塞，等待被其他线程notify
    // 重新获取锁
}
```

如此则可以避免死锁。

#### 1.3.4 wait()与notify()的问题

以上述的生产者-消费者模型来看，其伪代码大致如下：

```
public void enqueue() {
    synchronized(queue) {
        while (queue.full()) {
            queue.wait();
```

```
        }
        //... 数据入列
        queue.notify();  // 通知消费者，队列中有数据了。
    }
}

public void dequeue() {
    synchronized(queue) {
        while (queue.empty()) {
            queue.wait();
        }
        // 数据出队列
        queue.notify(); // 通知生产者，队列中有空间了，可以继续放数据了。
    }
}
```

生产者在通知消费者的同时，也通知了其他的生产者；消费者在通知生产者的同时，也通知了其他消费者。原因在于wait()和notify()所作用的对象和synchronized所作用的对象是同一个，只能有一个对象，无法区分队列空和列队满两个条件。这正是Condition要解决的问题。

## 1.4 InterruptedException与interrupt()方法

### 1.4.1 Interrupted异常

**什么情况下会抛出Interrupted异常**

假设while循环中没有调用任何的阻塞函数，就是通常的算术运算，或者打印一行日志，如下所示。

```
package com.lagou.concurrent.demo;

public class MyThread extends Thread {
    @Override
    public void run() {
        while (true) {
            boolean interrupted = isInterrupted();
            System.out.println("中断标记：" + interrupted);
        }
    }
}
```

这个时候，在主线程中调用一句thread.interrupt()，请问该线程是否会抛出异常？不会。

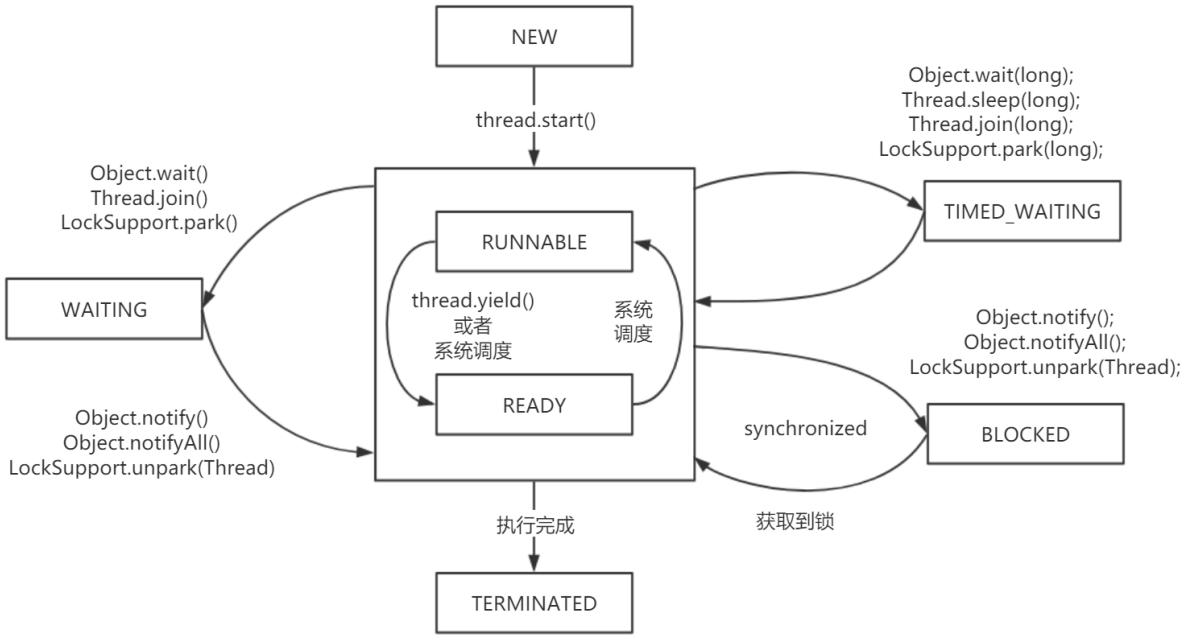
```java
package com.lagou.concurrent.demo;

public class Main42 {
    public static void main(String[] args) throws InterruptedException {
        MyThread42 myThread = new MyThread42();
        myThread.start();
        Thread.sleep(10);
        myThread.interrupt();
        Thread.sleep(100);
        System.exit(0);
    }
}
```

只有那些声明了会抛出InterruptedException的函数才会抛出异常，也就是下面这些常用的函数：

```java
public static native void sleep(long millis) throws InterruptedException {...}
public final void wait() throws InterruptedException {...}
public final void join() throws InterruptedException {...}
```

### 1.4.2 轻量级阻塞与重量级阻塞

能够被中断的阻塞称为轻量级阻塞，对应的线程状态是WAITING或者TIMED_WAITING；而像synchronized 这种不能被中断的阻塞称为重量级阻塞，对应的状态是 BLOCKED。如图所示：调用不同的方法后，一个线程的状态迁移过程。




初始线程处于NEW状态，调用start()开始执行后，进入RUNNING或者READY状态。如果没有调用任何的阻塞函数，线程只会在RUNNING和READY之间切换，也就是系统的时间片调度。这两种状态的切换是操作系统完成的，除非手动调用yield()函数，放弃对CPU的占用。

一旦调用了图中的任何阻塞函数，线程就会进入WAITING或者TIMED_WAITING状态，两者的区别只是前者为无限期阻塞，后者则传入了一个时间参数，阻塞一个有限的时间。如果使用了synchronized关键字或者synchronized块，则会进入BLOCKED状态。

不太常见的阻塞/唤醒函数，LockSupport.park()/unpark()。这对函数非常关键，Concurrent包中Lock的实现即依赖这一对操作原语。

因此thread.interrupted()的精确含义是“唤醒轻量级阻塞”，而不是字面意思“中断一个线程”。

**thread.isInterrupted()与Thread.interrupted()的区别**

因为 thread.interrupted()相当于给线程发送了一个唤醒的信号，所以如果线程此时恰好处于WAITING或者TIMED_WAITING状态，就会抛出一个InterruptedException，并且线程被唤醒。而如果线程此时并没有被阻塞，则线程什么都不会做。但在后续，线程可以判断自己是否收到过其他线程发来的中断信号，然后做一些对应的处理。

这两个方法都是线程用来判断自己是否收到过中断信号的，前者是实例方法，后者是静态方法。二者的区别在于，前者只是读取中断状态，不修改状态；后者不仅读取中断状态，还会重置中断标志位。

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        myThread.start();
        Thread.sleep(10);
        myThread.interrupt();
        Thread.sleep(7);
        System.out.println("main中断状态检查-1：" + myThread.isInterrupted());
        System.out.println("main中断状态检查-2：" + myThread.isInterrupted());

    }
}
```

```
package com.lagou.concurrent.demo;

public class MyThread extends Thread {
    @Override
    public void run() {
        int i = 0;
        while (true) {
            boolean interrupted = isInterrupted();
            System.out.println("中断标记：" + interrupted);

            ++i;
            if (i > 200) {
                // 检查并重置中断标志。
                boolean interrupted1 = Thread.interrupted();
                System.out.println("重置中断状态：" + interrupted1);
                interrupted1 = Thread.interrupted();
                System.out.println("重置中断状态：" + interrupted1);
                interrupted = isInterrupted();
                System.out.println("中断标记：" + interrupted);
                break;
            }

```

```
        }
    }
}
```

## 1.5 线程的优雅关闭

### 1.5.1 stop与destory函数

线程是“一段运行中的代码”，一个运行中的方法。运行到一半的线程能否强制杀死?

不能。在Java中，有stop()、destory()等方法，但这些方法官方明确不建议使用。原因很简单，如果强制杀死线程，则线程中所使用的资源，例如文件描述符、网络连接等无法正常关闭。

因此，一个线程一旦运行起来，不要强行关闭，合理的做法是让其运行完（也就是方法执行完毕），干净地释放掉所有资源，然后退出。如果是一个不断循环运行的线程，就需要用到线程间的通信机制，让主线程通知其退出。

### 1.5.2 守护线程

daemon线程和非daemon线程的对比：

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) {
        MyDaemonThread myDaemonThread = new MyDaemonThread();
        // 设置为daemon线程
        myDaemonThread.setDaemon(true);
        myDaemonThread.start();
        // 启动非daemon线程，当非daemon线程结束，不管daemon线程是否结束，都结束JVM进程。
        new MyThread().start();

    }
}
```

```
package com.lagou.concurrent.demo;

public class MyDaemonThread extends Thread {
    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            try {
                Thread.sleep(500);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class MyThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("非Daemon线程");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

对于上面的程序，在thread.start()前面加一行代码thread.setDaemon(true)。当main(...)函数退出后，线程thread就会退出，整个进程也会退出。

当在一个JVM进程里面开多个线程时，这些线程被分成两类：守护线程和非守护线程。默认都是非守护线程。

在Java中有一个规定：当所有的非守护线程退出后，整个JVM进程就会退出。意思就是守护线程"不算作数"，守护线程不影响整个 JVM 进程的退出。

例如，垃圾回收线程就是守护线程，它们在后台默默工作，当开发者的所有前台线程（非守护线程）都退出之后，整个JVM进程就退出了。

#### 1.5.3 设置关闭的标志位

开发中一般通过设置标志位的方式，停止循环运行的线程。

**代码1**

```
package com.lagou.concurrent.demo;

public class MyThread extends Thread{
    private boolean running = true;

    @Override
    public void run() {
        while (running) {
            System.out.println("线程正在运行。。。");
            try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void stopRunning() {
        this.running = false;
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        myThread.start();
        Thread.sleep(5000);
        myThread.stopRunning();
        myThread.join();
    }
}
```

但上面的代码有一个问题：如果MyThread t在while循环中阻塞在某个地方，例如里面调用了object.wait()函数，那它可能永远没有机会再执行 while( ! stopped)代码，也就一直无法退出循环。

此时，就要用到InterruptedException()与interrupt()函数。

# 2 并发核心概念

## 2.1 并发与并行

在单个处理器上采用单核执行多个任务即为并发。在这种情况下，操作系统的任务调度程序会很快从一个任务切换到另一个任务，因此看起来所有的任务都是同时运行的。

同一时间内在不同计算机、处理器或处理器核心上同时运行多个任务，就是所谓的“并行”。

另一个关于并发的定义是，在系统上同时运行多个任务（不同的任务）就是并发。而另一个关于并行的定义是：同时在某个数据集的不同部分上运行同一任务的不同实例就是并行。

关于并行的最后一个定义是，系统中同时运行了多个任务。关于并发的最后一个定义是，一种解释程序员将任务和它们对共享资源的访问同步的不同技术和机制的方法。

这两个概念非常相似，而且这种相似性随着多核处理器的发展也在不断增强。

## 2.2 同步

在并发中，我们可以将同步定义为一种协调两个或更多任务以获得预期结果的机制。同步的方式有两种：

- 控制同步：例如，当一个任务的开始依赖于另一个任务的结束时，第二个任务不能再第一个任务 完成之前开始。
- 数据访问同步：当两个或更多任务访问共享变量时，再任意时间里，只有一个任务可以访问该变量。

与同步密切相关的一个概念时临界段。临界段是一段代码，由于它可以访问共享资源，因此再任何给定时间内，只能被一个任务执行。**互斥**是用来保证这一要求的机制，而且可以采用不同的方式来实现。

同步可以帮助你在完成并发任务的同时避免一些错误，但是它也为你的算法引入了一些开销。你必须非常仔细地计算任务的数量，这些任务可以独立执行，而无需并行算法中的互通信。这就涉及并发算法的**粒度**。如果算法有着粗粒度（低互通信的大型任务），同步方面的开销就会较低。然而，也许你不会用到系统所有的核心。如果算法有者细粒度（高互通信的小型任务），同步方面的开销就会很高，而且该算法的吞吐量可能不会很好。

并发系统中有不同的同步机制。从理论角度看，最流行的机制如下：

- 信号量（semaphore）：一种用于控制对一个或多个单位资源进行访问的机制。它有一个用于存放可用资源数量的变量，而且可以采用两种原子操作来管理该变量。**互斥**（mutex，mutual exclusion的简写形式）是一种特殊类型的信号量，它只能取两个值（即**资源空闲**和**资源忙**），而且只有将互斥设置为**忙**的那个进程才可以释放它。互斥可以通过保护临界段来帮助你避免出现竞争条件。
- 监视器：一种在共享资源上实现互斥的机制。它有一个互斥、一个条件变量、两种操作（等待条件和通报条件）。一旦你通报了该条件，在等待它的任务中只有一个会继续执行。

如果共享数据的所有用户都受到同步机制的保护，那么代码（或方法、对象）就是**线程安全**的。数据的非阻塞的CAS（compare-and-swap，比较和交换）原语是不可变的，这样就可以在并发应用程序中使用该代码而不会出任何问题。

## 2.3 不可变对象

**不可变对象**是一种非常特殊的对象。在其初始化后，不能修改其可视状态（其属性值）。如果想修改一个不可变对象，那么你就必须创建一个新的对象。

不可变对象的主要优点在于它是线程安全的。你可以在并发应用程序中使用它而不会出现任何问题。

不可变对象的一个例子就是java中的String类。当你给一个String对象赋新值时，会创建一个新的String对象。

## 2.4 原子操作和原子变量

与应用程序的其他任务相比，**原子操作**是一种发生在瞬间的操作。在并发应用程序中，可以通过一个临界段来实现原子操作，以便对整个操作采用同步机制。

**原子变量**是一种通过原子操作来设置和获取其值的变量。可以使用某种同步机制来实现一个原子变量，或者也可以使用CAS以无锁方式来实现一个原子变量，而这种方式并不需要任何同步机制。

## 2.5 共享内存与消息传递

任务可以通过两种不同的方式来相互通信。第一种方法是**共享内存**，通常用于在同一台计算机上运行多任务的情况。任务在读取和写入值的时候使用相同的内存区域。为了避免出现问题，对该共享内存的访问必须在一个由同步机制保护的临界段内完成。

另一种同步机制是**消息传递**，通常用于在不同计算机上运行多任务的情形。当一个任务需要与另一个任务通信时，它会发送一个遵循预定义协议的消息。如果发送方保持阻塞并等待响应，那么该通信就是同步的；如果发送方在发送消息后继续执行自己的流程，那么该通信就是异步的。

# 3 并发的问题

## 3.1 数据竞争

如果有两个或者多个任务在临界段之外对一个共享变量进行写入操作，也就是说没有使用任何同步机制，那么应用程序可能存在**数据竞争**（也叫做**竞争条件**）。

在这些情况下，应用程序的最终结果可能取决于任务的执行顺序。

```
package com.lagou.concurrent.demo;

public class ConcurrentDemo {

    private float myFloat;

    public void modify(float difference) {
        float value = this.myFloat;
        this.myFloat = value + difference;
    }

    public static void main(String[] args) {

    }
}
```

假设有两个不同的任务执行了同一个modify方法。由于任务中语句的执行顺序不同，最终结果也会不同。

modify方法不是原子的，`ConcurrentDemo`也不是线程安全的。

## 3.2 死锁

当两个（或多个）任务正在等待必须由另一线程释放的某个共享资源，而该线程又正在等待必须由前述任务之一释放的另一共享资惊时，并发应用程序就出现了死锁。当系统中同时出现如下四种条件时，就会导致这种情形。我们将其称为Coffman 条件。

- 互斥： 死锁中涉及的资师、必须是不可共享的。一次只有一个任务可以使用该资源。
- 占有并等待条件： 一个任务在占有某一互斥的资源时又请求另一互斥的资源。当它在等待时，不会释放任何资源。
- 不可剥夺：资源只能被那些持有它们的任务释放。
- 循环等待：任务1正等待任务2 所占有的资源， 而任务2 又正在等待任务3 所占有的资源，以此类推，最终任务n又在等待由任务1所占有的资源，这样就出现了循环等待。

有一些机制可以用来避免死锁。

- 忽略它们：这是最常用的机制。你可以假设自己的系统绝不会出现死锁，而如果发生死锁，结果就是你可以停止应用程序并且重新执行它。
- 检测：系统中有一项专门分析系统状态的任务，可以检测是否发生了死锁。如果它检测到了死锁，可以采取一些措施来修复该问题，例如，结束某个任务或者强制释放某一资源。
- 预防：如果你想防止系统出现死锁，就必须预防Coffman 条件中的一条或多条出现。

- 规避：如果你可以在某一任务执行之前得到该任务所使用资源的相关信息，那么死锁是可以规避的。当一个任务要开始执行时，你可以对系统中空闲的资源和任务所需的资源进行分析，这样就可以判断任务是否能够开始执行。

### 3.3 活锁

如果系统中有两个任务，它们总是因对方的行为而改变自己的状态， 那么就出现了活锁。最终结果是它们陷入了状态变更的循环而无法继续向下执行。

例如，有两个任务：任务1和任务2 ，它们都需要用到两个资源：资源1和资源2 。假设任务1对资源1加了一个锁，而任务2 对资源2 加了一个锁。当它们无法访问所需的资源时，就会释放自己的资源并且重新开始循环。这种情况可以无限地持续下去，所以这两个任务都不会结束自己的执行过程。

### 3.4 资源不足

当某个任务在系统中无法获取维持其继续执行所需的资源时，就会出现资源不足。当有多个任务在等待某一资源且该资源被释放时，系统需要选择下一个可以使用该资源的任务。如果你的系统中没有设计良好的算法，那么系统中有些线程很可能要为获取该资源而等待很长时间。

要解决这一问题就要确保公平原则。所有等待某一资源的任务必须在某一给定时间之内占有该资源。可选方案之一就是实现一个算法，在选择下一个将占有某一资源的任务时，对任务已等待该资源的时间因素加以考虑。然而，实现锁的公平需要增加额外的开销，这可能会降低程序的吞吐量。

### 3.5 优先权反转

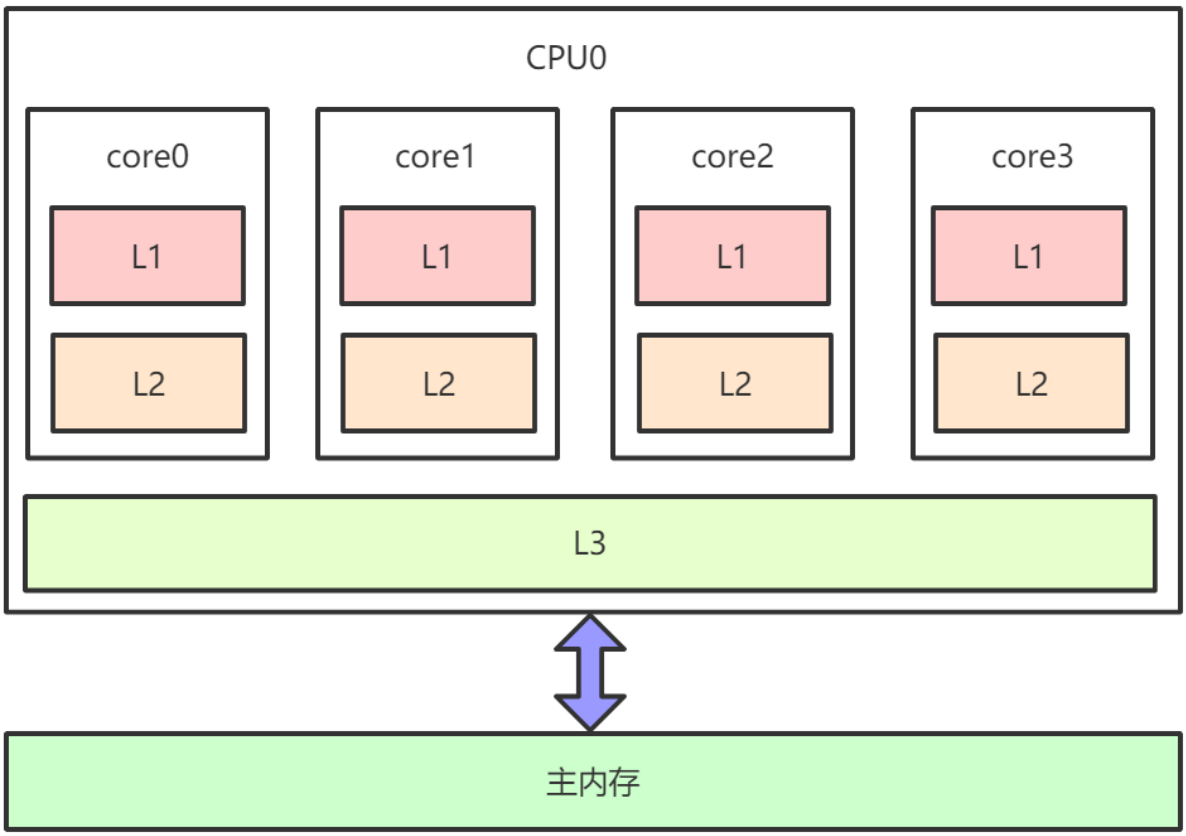
当一个低优先权的任务持有了一个高优先级任务所需的资源时，就会发生优先权反转。这样的话，低优先权的任务就会在高优先权的任务之前执行。

## 4 JMM内存模型

### 4.1 JMM与happen-before

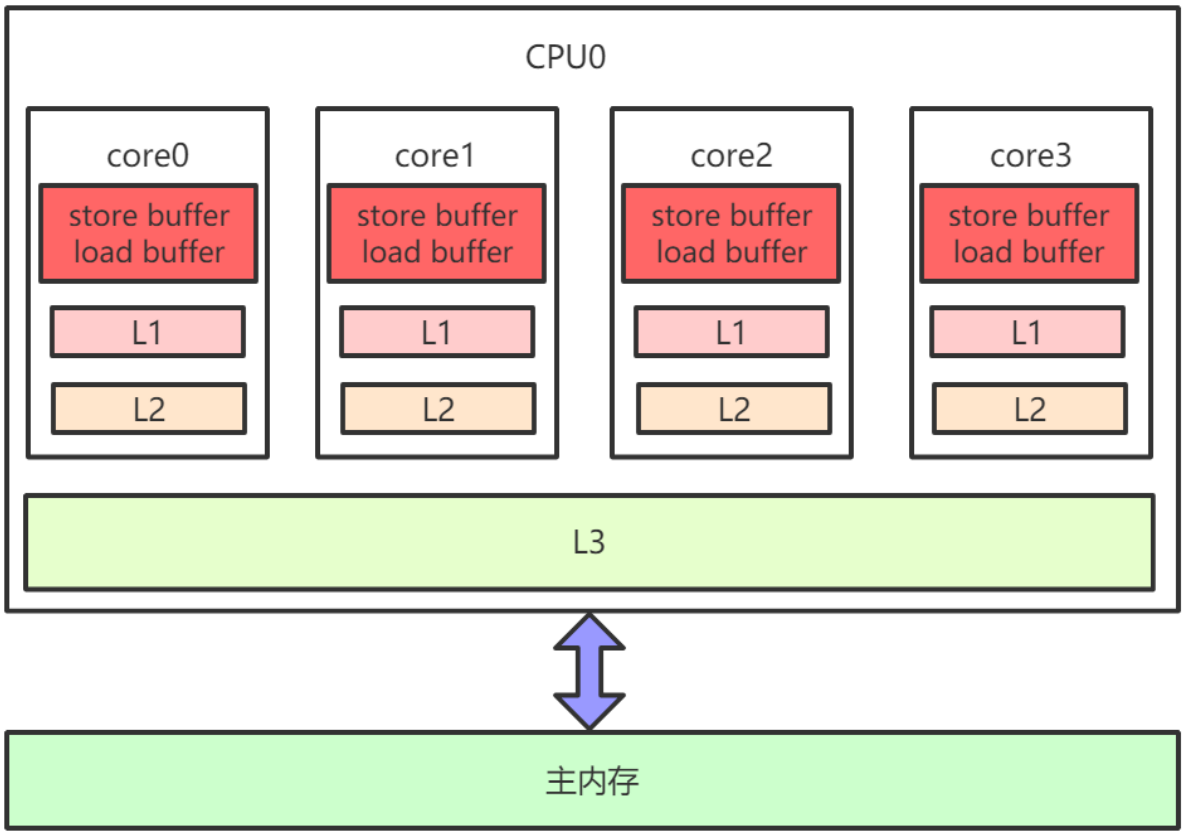
#### 4.1.1 为什么会存在“内存可见性”问题

下图为x86架构下CPU缓存的布局，即在一个CPU 4核下，L1、L2、L3三级缓存与主内存的布局。每个核上面有L1、L2缓存，L3缓存为所有核共用。



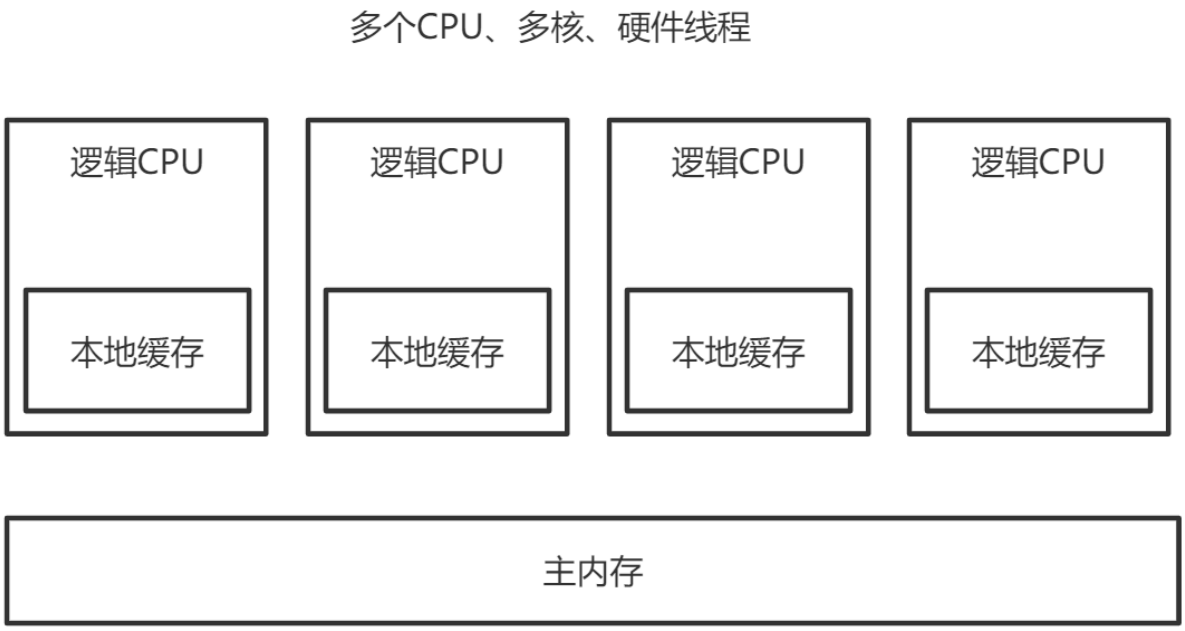

因为存在CPU缓存一致性协议，例如MESI，多个CPU核心之间缓存不会出现不同步的问题，不会有“内存可见性”问题。

缓存一致性协议对性能有很大损耗，为了解决这个问题，又进行了各种优化。例如，在计算单元和L1之间加了Store Buffer、Load Buffer（还有其他各种Buffer），如下图：



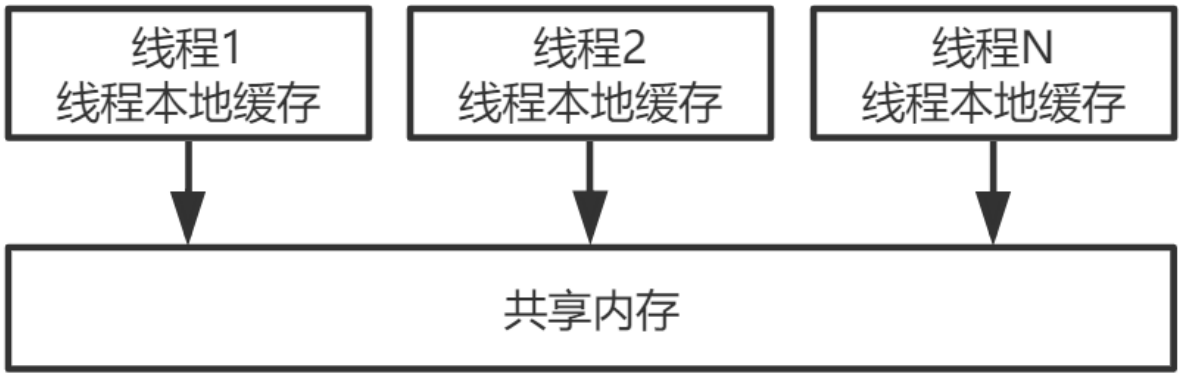
L1、L2、L3和主内存之间是同步的，有缓存一致性协议的保证，但是Store Buffer、Load Buffer和L1之间却是异步的。向内存中写入一个变量，这个变量会保存在Store Buffer里面，稍后才异步地写入L1中，同时同步写入主内存中。

操作系统内核视角下的CPU缓存模型：




多CPU，每个CPU多核，每个核上面可能还有多个硬件线程，对于操作系统来讲，就相当于一个个的逻辑CPU。每个逻辑CPU都有自己的缓存，这些缓存和主内存之间不是完全同步的。

对应到Java里，就是JVM抽象内存模型，如下图所示：




#### 4.1.2 重排序与内存可见性的关系

Store Buffer的延迟写入是重排序的一种，称为内存重排序（Memory Ordering）。除此之外，还有编译器和CPU的指令重排序。

重排序类型：

1. 编译器重排序。

   对于没有先后依赖关系的语句，编译器可以重新调整语句的执行顺序。

2. CPU指令重排序。

   在指令级别，让没有依赖关系的多条指令并行。

3. CPU内存重排序。

   CPU有自己的缓存，指令的执行顺序和写入主内存的顺序不完全一致。

在三种重排序中，第三类就是造成“内存可见性”问题的主因，如下案例：

线程1：

X=1

a=Y

线程2：

Y=1

b=X

假设X、Y是两个全局变量，初始的时候，X=0，Y=0。请问，这两个线程执行完毕之后，a、b的正确结果应该是什么？

很显然，线程1和线程2的执行先后顺序是不确定的，可能顺序执行，也可能交叉执行，最终正确的结果可能是：

1. a=0,b=1
2. a=1,b=0
3. a=1,b=1

也就是不管谁先谁后，执行结果应该是这三种场景中的一种。但实际可能是a=0，b=0。

两个线程的指令都没有重排序，执行顺序就是代码的顺序，但仍然可能出现a=0，b=0。原因是线程1先执行X=1，后执行a=Y，但此时X=1还在自己的Store Buffer里面，没有及时写入主内存中。所以，线程2看到的X还是0。线程2的道理与此相同。

虽然线程1觉得自己是按代码顺序正常执行的，但在线程2看来，a=Y和X=1顺序却是颠倒的。指令没有重排序，是写入内存的操作被延迟了，也就是内存被重排序了，这就造成内存可见性问题。

#### 4.1.3 内存屏障

为了禁止编译器重排序和 CPU 重排序，在编译器和 CPU 层面都有对应的指令，也就是内存屏障（Memory Barrier）。这也正是JMM和happen-before规则的底层实现原理。

编译器的内存屏障，只是为了告诉编译器不要对指令进行重排序。当编译完成之后，这种内存屏障就消失了，CPU并不会感知到编译器中内存屏障的存在。

而CPU的内存屏障是CPU提供的指令，可以由开发者显示调用。

内存屏障是很底层的概念，对于 Java 开发者来说，一般用 volatile 关键字就足够了。但从JDK 8开始，Java在Unsafe类中提供了三个内存屏障函数，如下所示。

```java
public final class Unsafe {
    // ...
    public native void loadFence();
    public native void storeFence();
    public native void fullFence();
    // ...
}
```

在理论层面，可以把基本的CPU内存屏障分成四种：

1. LoadLoad：禁止读和读的重排序。
2. StoreStore：禁止写和写的重排序。
3. LoadStore：禁止读和写的重排序。
4. StoreLoad：禁止写和读的重排序。

Unsafe中的方法：

1. loadFence=LoadLoad+LoadStore
2. storeFence=StoreStore+LoadStore
3. fullFence=loadFence+storeFence+StoreLoad

#### 4.1.4 as-if-serial语义

重排序的原则是什么？什么场景下可以重排序，什么场景下不能重排序呢？

**1.单线程程序的重排序规则**

无论什么语言，站在编译器和CPU的角度来说，不管怎么重排序，单线程程序的执行结果不能改变，这就是单线程程序的重排序规则。

即只要操作之间没有数据依赖性，编译器和CPU都可以任意重排序，因为执行结果不会改变，代码看起来就像是完全串行地一行行从头执行到尾，这也就是as-if-serial语义。

对于单线程程序来说，编译器和CPU可能做了重排序，但开发者感知不到，也不存在内存可见性问题。

**2.多线程程序的重排序规则**

编译器和CPU的这一行为对于单线程程序没有影响，但对多线程程序却有影响。

对于多线程程序来说，线程之间的数据依赖性太复杂，编译器和CPU没有办法完全理解这种依赖性并据此做出最合理的优化。

编译器和CPU只能保证**每个线程的as-if-serial语义**。

线程之间的数据依赖和相互影响，需要编译器和CPU的上层来确定。

上层要告知编译器和CPU在多线程场景下什么时候可以重排序，什么时候不能重排序。

#### 4.1.5 happen-before是什么

使用happen-before描述两个操作之间的内存可见性。

java内存模型（JMM）是一套规范，在多线程中，一方面，要让编译器和CPU可以灵活地重排序；另一方面，要对开发者做一些承诺，明确告知开发者不需要感知什么样的重排序，需要感知什么样的重排序。然后，根据需要决定这种重排序对程序是否有影响。如果有影响，就需要开发者显示地通过volatile、synchronized等线程同步机制来禁止重排序。

关于happen-before：

如果A happen-before B，意味着A的执行结果必须对B可见，也就是保证跨线程的内存可见性。A happen before B不代表A一定在B之前执行。因为，对于多线程程序而言，两个操作的执行顺序是不确定的。happen-before只确保如果A在B之前执行，则A的执行结果必须对B可见。定义了内存可见性的约束，也就定义了一系列重排序的约束。

基于happen-before的这种描述方法，JMM对开发者做出了一系列承诺：

1. 单线程中的每个操作，happen-before 对应该线程中任意后续操作（也就是 as-if-serial语义保证）。
2. 对volatile变量的写入，happen-before对应后续对这个变量的读取。
3. 对synchronized的解锁，happen-before对应后续对这个锁的加锁。

……

JMM对编译器和CPU 来说，volatile 变量不能重排序；非 volatile 变量可以任意重排序。

#### 4.1.6 happen-before的传递性

除了这些基本的happen-before规则，happen-before还具有**传递性**，即若A happen-before B，B happen-before C，则A happen-before C。

如果一个变量不是volatile变量，当一个线程读取、一个线程写入时可能有问题。那岂不是说，在多线程程序中，我们要么加锁，要么必须把所有变量都声明为volatile变量？这显然不可能，而这就得归功于happen-before的传递性。

```
class A {
    private int a = 0;
    private volatile int c = 0;
    public void set() {
        a = 5; // 操作1
        c = 1; // 操作2
    }

    public int get() {
        int d = c; // 操作3
        return a; // 操作4
    }
}
```

假设线程A先调用了set，设置了a=5；之后线程B调用了get，返回值一定是a=5。为什么呢？

操作1和操作2是在同一个线程内存中执行的，操作1 happen-before 操作2，同理，操作3 happen-before操作4。又因为c是volatile变量，对c的写入happen-before对c的读取，所以操作2 happen-before操作3。利用happen-before的传递性，就得到：

操作1 happen-before 操作2 happen-before 操作3 happen-before操作4。

所以，操作1的结果，一定对操作4可见。

```
class A {
    private int a = 0;
    private int c = 0;
    public synchronized void set() {
        a = 5; // 操作1
        c = 1; // 操作2
    }

    public synchronized int get() {
        return a;
    }
}
```

假设线程A先调用了set，设置了a=5；之后线程B调用了get，返回值也一定是a=5。

因为与volatile一样，synchronized同样具有happen-before语义。展开上面的代码可得到类似于下面的伪代码：

```
线程A:
    加锁;   // 操作1
    a = 5; // 操作2
    c = 1; // 操作3
    解锁;   // 操作4
线程B:
    加锁;   // 操作5
    读取a;  // 操作6
    解锁;   // 操作7
```

根据synchronized的happen-before语义，操作4 happen-before 操作5，再结合传递性，最终就会得到：

操作1 happen-before 操作2.....happen-before 操作7。所以，a、c都不是volatile变量，但仍然有内存可见性。

## 4.2 volatile关键字

### 4.2.1 64位写入的原子性（Half Write）

如，对于一个long型变量的赋值和取值操作而言，在多线程场景下，线程A调用set(100)，线程B调用get()，在某些场景下，返回值可能不是100。

```
public class MyClass {
    private long a = 0;
    // 线程A调用set(100)
    public void set(long a) {
        this.a = a;
    }

    // 线程B调用get()，返回值一定是100吗？
    public long get() {
        return this.a;
    }
}
```

因为JVM的规范并没有要求64位的long或者double的写入是原子的。在32位的机器上，一个64位变量的写入可能被拆分成两个32位的写操作来执行。这样一来，读取的线程就可能读到“一半的值”。解决办法也很简单，在long前面加上volatile关键字。

#### 4.2.2 重排序：DCL问题

单例模式的线程安全的写法不止一种，常用写法为DCL（Double Checking Locking），如下所示：

```
public class Singleton {
    private static Singleton instance;
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    // 此处代码有问题
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

上述的`instance = new Singleton();`代码有问题：其底层会分为三个操作：

1. 分配一块内存。
2. 在内存上初始化成员变量。
3. 把instance引用指向内存。

在这三个操作中，操作2和操作3可能重排序，即先把instance指向内存，再初始化成员变量，因为二者并没有先后的依赖关系。此时，另外一个线程可能拿到一个未完全初始化的对象。这时，直接访问里面的成员变量，就可能出错。这就是典型的“构造方法溢出”问题。

解决办法也很简单，就是为instance变量加上volatile修饰。

volatile的三重功效：64位写入的原子性、内存可见性和禁止重排序。

#### 4.2.3 volatile实现原理

由于不同的CPU架构的缓存体系不一样，重排序的策略不一样，所提供的内存屏障指令也就有差异。

这里只探讨为了实现volatile关键字的语义的一种参考做法：

1. 在volatile写操作的前面插入一个StoreStore屏障。保证volatile写操作不会和之前的写操作重排序。
2. 在volatile写操作的后面插入一个StoreLoad屏障。保证volatile写操作不会和之后的读操作重排序。
3. 在volatile读操作的后面插入一个LoadLoad屏障+LoadStore屏障。保证volatile读操作不会和之后的读操作、写操作重排序。

具体到x86平台上，其实不会有LoadLoad、LoadStore和StoreStore重排序，只有StoreLoad一种重排序（内存屏障），也就是只需要在volatile写操作后面加上StoreLoad屏障。

#### 4.2.4 JSR-133对volatile语义的增强

在JSR -133之前的旧内存模型中，一个64位long/ double型变量的读/ 写操作可以被拆分为两个32位的读/写操作来执行。从JSR -133内存模型开始 （即从JDK5开始），仅仅只允许把一个64位long/ double型变量的**写操作拆分**为两个32位的写操作来执行，任意的**读操作**在JSR -133中都**必须具有原子性**（即 任意读操作必须要在单个读事务中执行）。

这也正体现了Java对happen-before规则的严格遵守。

### 4.3 final关键字

#### 4.3.1 构造方法溢出问题

考虑下面的代码：

```
public class MyClass {
    private int num1;
    private int num2;
    private static MyClass myClass;

    public MyClass() {
        num1 = 1;
        num2 = 2;
    }

    /**
      * 线程A先执行write()
      */
    public static void write() {
        myClass = new MyClass();
    }

    /**
      * 线程B接着执行write()
      */
    public static void read() {
        if (myClass != null) {
            int num3 = myClass.num1;
```

```
            int num4 = myClass.num2;
        }
    }
}
```

num3和num4的值是否一定是1和2?

num3、num4不见得一定等于1，2。和DCL的例子类似，也就是构造方法溢出问题。

myClass = new MyClass()这行代码，分解成三个操作：

1. 分配一块内存；
2. 在内存上初始化i=1，j=2；
3. 把myClass指向这块内存。

操作2和操作3可能重排序，因此线程B可能看到未正确初始化的值。对于构造方法溢出，就是一个对象的构造并不是“原子的”，当一个线程正在构造对象时，另外一个线程却可以读到未构造好的“一半对象”。

#### 4.3.2 final的happen-before语义

要解决这个问题，不止有一种办法。

办法1：给num1，num2加上volatile关键字。

办法2：为read/write方法都加上synchronized关键字。

如果num1，num2只需要初始化一次，还可以使用final关键字。

之所以能解决问题，是因为同volatile一样，final关键字也有相应的happen-before语义：

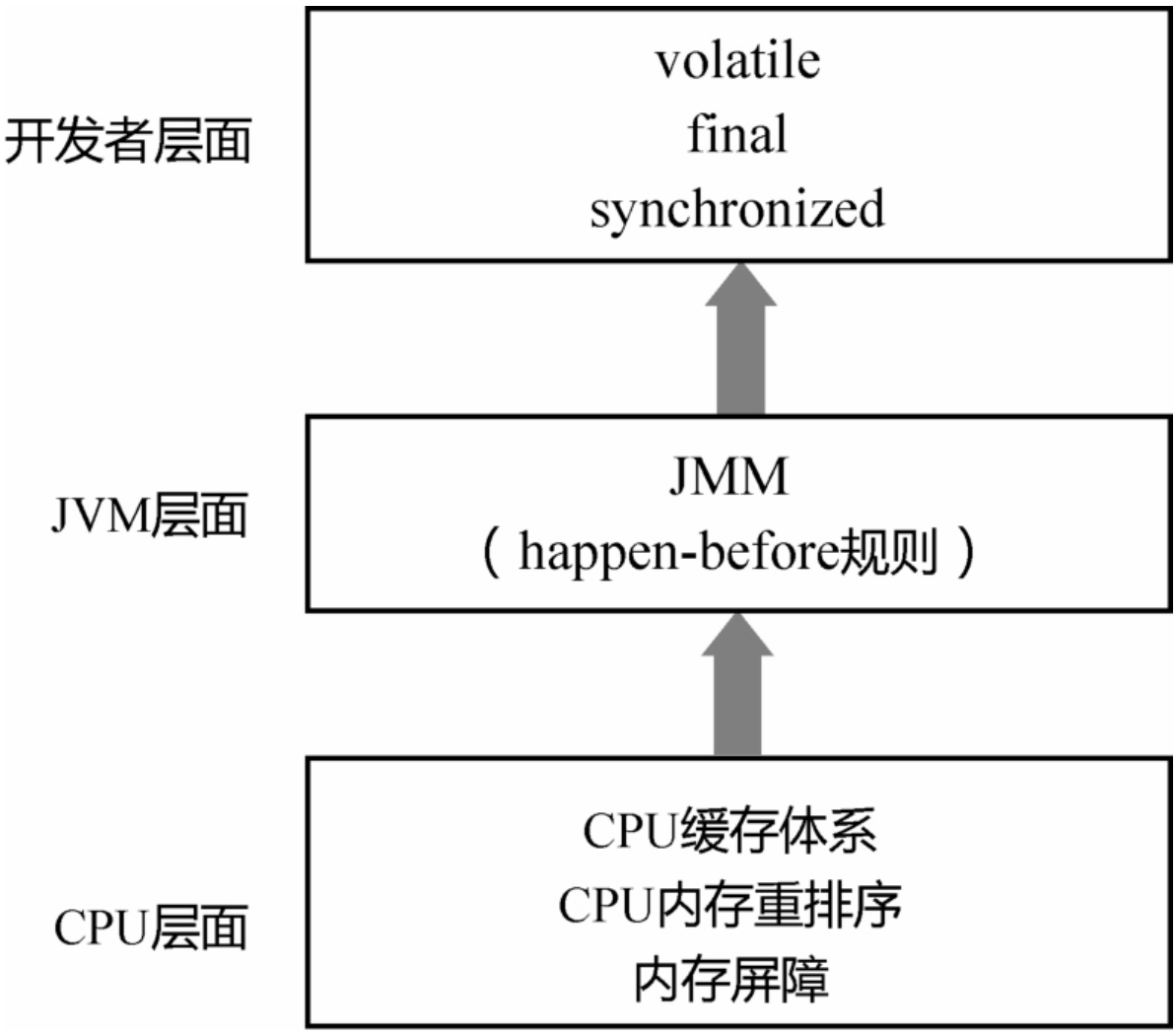
1. 对final域的写（构造方法内部），happen-before于后续对final域所在对象的读。
2. 对final域所在对象的读，happen-before于后续对final域的读。

通过这种happen-before语义的限定，保证了final域的赋值，一定在构造方法之前完成，不会出现另外一个线程读取到了对象，但对象里面的变量却还没有初始化的情形，避免出现构造方法溢出的问题。

#### 4.3.3 happen-before规则总结

1. 单线程中的每个操作，happen-before于该线程中任意后续操作。
2. 对volatile变量的写，happen-before于后续对这个变量的读。
3. 对synchronized的解锁，happen-before于后续对这个锁的加锁。
4. 对final变量的写，happen-before于final域对象的读，happen-before于后续对final变量的读。

四个基本规则再加上happen-before的传递性，就构成JMM对开发者的整个承诺。在这个承诺以外的部分，程序都可能被重排序，都需要开发者小心地处理内存可见性问题。



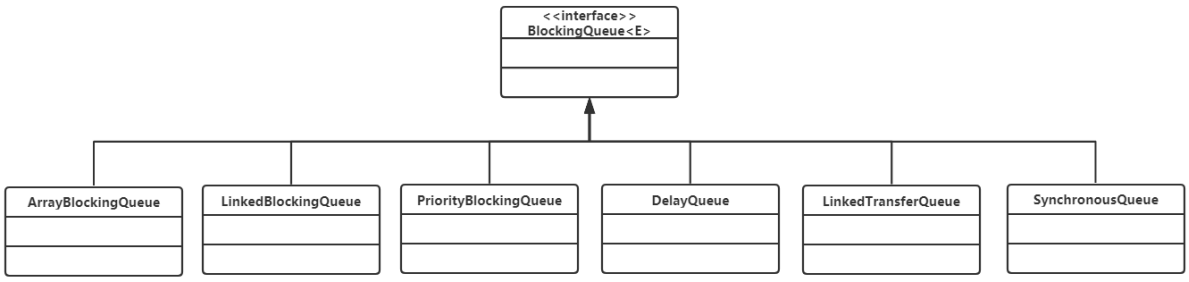

# 第二部分：JUC

## 5 并发容器

### 5.1 BlockingQueue

在所有的并发容器中，BlockingQueue是最常见的一种。BlockingQueue是一个带阻塞功能的队列，当入队列时，若队列已满，则阻塞调用者；当出队列时，若队列为空，则阻塞调用者。

在Concurrent包中，BlockingQueue是一个接口，有许多个不同的实现类，如图所示。




该接口的定义如下：

```
public interface BlockingQueue<E> extends Queue<E> {
    //...
    boolean add(E e);
    boolean offer(E e);
    void put(E e) throws InterruptedException;
    boolean remove(Object o);
    E take() throws InterruptedException;
    E poll(long timeout, TimeUnit unit) throws InterruptedException;
    //...
}
```

该接口和JDK集合包中的Queue接口是兼容的，同时在其基础上增加了阻塞功能。在这里，入队提供了add(...)、offer(..)、put(...)3个方法，有什么区别呢？从上面的定义可以看到，add(...)和offer(..)的返回值是布尔类型，而put无返回值，还会抛出中断异常，所以add(...)和offer(..)是无阻塞的，也是Queue本身定义的接口，而put(..)是阻塞的。add(...)和offer(..)的区别不大，当队列为满的时候，前者会抛出异常，后者则直接返回false。

出队列与之类似，提供了remove()、poll()、take()等方法，remove()是非阻塞式的，take()和poll()是阻塞式的。

### 5.1.1 ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的环形队列，在构造方法中，会要求传入数组的容量。

```
public ArrayBlockingQueue(int capacity) {
    this(capacity, false);
}

public ArrayBlockingQueue(int capacity, boolean fair) {
    // ...
}

public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends
E> c) {
    this(capacity, fair);
    // ...
}
```

其核心数据结构如下：

```java
public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements
BlockingQueue<E>, java.io.Serializable {
    //...
    final Object[] items;
    // 队头指针
    int takeIndex;
    // 队尾指针
    int putIndex;
    int count;

    // 核心为1个锁外加两个条件
    final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;
    //...
}
```

其put/take方法也很简单，如下所示。

put方法：

ArrayBlockingQueue.java

```java
     * @throws NullPointerException {@inheritDoc}
     */
    public void put(E e) throws InterruptedException {
        Objects.requireNonNull(e);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly(); 可中断的Lock
        try {
            while (count == items.length)
                notFull.await(); 若队列满，则阻塞
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }
```

ArrayBlockingQueue.java

```java
    private void enqueue(E e) {
        // assert lock.isHeldByCurrentThread();
        // assert lock.getHoldCount() == 1;
        // assert items[putIndex] == null;
        final Object[] items = this.items;
        items[putIndex] = e;
        if (++putIndex == items.length) putIndex = 0;
        count++;
        notEmpty.signal(); 当将数据put到queue中之后，通知非空条件
    }
```

take方法：

ArrayBlockingQueue.java

```
412     public E take() throws InterruptedException {
413         final ReentrantLock lock = this.lock;
414         lock.lockInterruptibly();
415         try {
416             while (count == 0)
417                 notEmpty.await(); take的时候，如果队列为空，则阻塞
418             return dequeue();
419         } finally {
420             lock.unlock();
421         }
422     }
```

ArrayBlockingQueue.java

```
191     private E dequeue() {
192         // assert lock.isHeldByCurrentThread();
193         // assert lock.getHoldCount() == 1;
194         // assert items[takeIndex] != null;
195         final Object[] items = this.items;
196         /unchecked/
197         E e = (E) items[takeIndex];
198         items[takeIndex] = null;
199         if (++takeIndex == items.length) takeIndex = 0;
200         count--;
201         if (itrs != null)
202             itrs.elementDequeued();
203         notFull.signal(); take结束，通知非满条件
204         return e;
205     }
```

### 5.1.2 LinkedBlockingQueue

LinkedBlockingQueue是一种基于**单向链表**的阻塞队列。因为队头和队尾是2个指针分开操作的，所以用了2把锁+2个条件，同时有1个AtomicInteger的原子变量记录count数。

```
1  public class LinkedBlockingQueue<E> extends AbstractQueue<E> implements
   BlockingQueue<E>, java.io.Serializable {
2      // ...
3      private final int capacity;
4      // 原子变量
5      private final AtomicInteger count = new AtomicInteger(0);
6      // 单向链表的头部
7      private transient Node<E> head;
8      // 单向链表的尾部
9      private transient Node<E> last;
10
11     // 两把锁，两个条件
12     private final ReentrantLock takeLock = new ReentrantLock();
13     private final Condition notEmpty = takeLock.newCondition();
14     private final ReentrantLock putLock = new ReentrantLock();
15     private final Condition notFull = putLock.newCondition();
16     // ...
```

```
17  }
```

在其构造方法中，也可以指定队列的总容量。如果不指定，默认为Integer.MAX_VALUE。

LinkedBlockingQueue.java

```
241         */
242        public LinkedBlockingQueue() {
243            this(Integer.MAX_VALUE);
244        }
```

LinkedBlockingQueue.java

```
252         */
253        public LinkedBlockingQueue(int capacity) {
254            if (capacity <= 0) throw new IllegalArgumentException();
255            this.capacity = capacity;
256            last = head = new Node<E>( x: null);
257        }
```

put/take实现。

LinkedBlockingQueue.java

```
424
425        public E take() throws InterruptedException {
426            final E x;
427            final int c;
428            final AtomicInteger count = this.count;
429            final ReentrantLock takeLock = this.takeLock;
430            takeLock.lockInterruptibly();
431            try {
432                while (count.get() == 0) {
433                    notEmpty.await();
434                }
435                x = dequeue();
436                c = count.getAndDecrement();
437                if (c > 1)
438                    notEmpty.signal(); 如果还有元素，则通知其他take线程
439            } finally {
440                takeLock.unlock();
441            }
442            if (c == capacity)
443                signalNotFull();
444            return x;
445        }
```

LinkedBlockingQueue.java

```
323         */
324     public void put(E e) throws InterruptedException {
325         if (e == null) throw new NullPointerException();
326         final int c;
327         final Node<E> node = new Node<E>(e);
328         final ReentrantLock putLock = this.putLock;
329         final AtomicInteger count = this.count;
330         putLock.lockInterruptibly();
331         try {
332             /*
333              * Note that count is used in wait guard even though it is
334              * not protected by lock. This works because count can
335              * only decrease at this point (all other puts are shut
336              * out by lock), and we (or some other waiting put) are
337              * signalled if it ever changes from capacity. Similarly
338              * for all other uses of count in other wait guards.
339              */
340             while (count.get() == capacity) {
341                 notFull.await();
342             }
343             enqueue(node);
344             c = count.getAndIncrement();
345             if (c + 1 < capacity)
346                 notFull.signal(); 如果队列还有剩余空间则通知其他put线程
347         } finally {
348             putLock.unlock();
349         }
350         if (c == 0)
351             signalNotEmpty();
352     }
```

LinkedBlockingQueue和ArrayBlockingQueue的差异：

1. 为了提高并发度，用2把锁，分别控制队头、队尾的操作。意味着在put(...)和put(...)之间、take()与take()之间是互斥的，put(...)和take()之间并不互斥。但对于count变量，双方都需要操作，所以必须是原子类型。
2. 因为各自拿了一把锁，所以当需要调用对方的condition的signal时，还必须再加上对方的锁，就是signalNotEmpty()和signalNotFull()方法。示例如下所示。

LinkedBlockingQueue.java

```
171     private void signalNotEmpty() {
172         final ReentrantLock takeLock = this.takeLock;
173         takeLock.lock();
174         try {
175             notEmpty.signal();
176         } finally {
177             takeLock.unlock();
178         }
179     }
180
181     /**
182      * Signals a waiting put. Called only from take/poll.
183      */
184     private void signalNotFull() {
185         final ReentrantLock putLock = this.putLock;
186         putLock.lock();
187         try {
188             notFull.signal();
189         } finally {
190             putLock.unlock();
191         }
```

必须先获取takeLock才可以调用notEmpty.signal()方法

必须先获取putLock，才可以调用notFull.signal()方法

3. 不仅put会通知 take，take 也会通知 put。当put 发现非满的时候，也会通知其他 put线程；当take发现非空的时候，也会通知其他take线程。

### 5.1.3 PriorityBlockingQueue

队列通常是先进先出的，而PriorityQueue是按照元素的优先级从小到大出队列的。正因为如此，PriorityQueue中的2个元素之间需要可以比较大小，并实现Comparable接口。

其核心数据结构如下：

```
public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements
BlockingQueue<E>, java.io.Serializable {
    //...
    // 用数组实现的二插小根堆
    private transient Object[] queue;
    private transient int size;

    private transient Comparator<? super E> comparator;
    // 1个锁+一个条件，没有非满条件
    private final ReentrantLock lock;
    private final Condition notEmpty;
    //...
}
```

其构造方法如下所示，如果不指定初始大小，内部会设定一个默认值11，当元素个数超过这个大小之后，会自动扩容。

PriorityBlockingQueue.java

```
193          * their {@linkplain Comparable natural ordering}.
194          */
195         public PriorityBlockingQueue() {
196             this(DEFAULT_INITIAL_CAPACITY, comparator: null);
197         }
```

11

PriorityBlockingQueue.java

```
137     private static final int DEFAULT_INITIAL_CAPACITY = 11;
138
```

PriorityBlockingQueue.java

```
222          *                  than 1
223          */
224         public PriorityBlockingQueue(int initialCapacity,
225                                      Comparator<? super E> comparator) {
226             if (initialCapacity < 1)
227                 throw new IllegalArgumentException();
228             this.comparator = comparator;
229             this.queue = new Object[Math.max(1, initialCapacity)];
230         }
```

下面是对应的put/take方法的实现。

put方法的实现：

PriorityBlockingQueue.java

```
506          * @throws NullPointerException if the spe
507          */
508         public void put(E e) {
509             offer(e); // never need to block
510         }
```

PriorityBlockingQueue.java

```
475    public boolean offer(E e) {
476        if (e == null)
477            throw new NullPointerException();
478        final ReentrantLock lock = this.lock;
479        lock.lock();
480        int n, cap;
481        Object[] es;
482        while ((n = size) >= (cap = (es = queue).length))
483            tryGrow(es, cap); 元素数超过了数据的长度，则扩容
484        try {
485            final Comparator<? super E> cmp;
486            if ((cmp = comparator) == null) 如果没有定义比较操作，则使用元素
487                siftUpComparable(n, e, es); 自带的比较功能
488            else      元素入堆，即执行siftUp操作
489                siftUpUsingComparator(n, e, es, cmp);
490            size = n + 1;
491            notEmpty.signal();
492        } finally {
493            lock.unlock();
494        }
495        return true;
496    }
```

take的实现：

PriorityBlockingQueue.java

```
541    public E take() throws InterruptedException {
542        final ReentrantLock lock = this.lock;
543        lock.lockInterruptibly();
544        E result;
545        try {              出队列
546            while ( (result = dequeue()) == null)
547                notEmpty.await();
548        } finally {
549            lock.unlock();
550        }
551        return result;
552    }
```

PriorityBlockingQueue.java

```
326         */
327        private E dequeue() {
328            // assert lock.isHeldByCurrentThread();
329            final Object[] es;
330            final E result;
331             因为是最小二叉堆，堆顶就是要出队的元素
332            if ((result = (E) ((es = queue)[0])) != null) {
333                final int n;
334                final E x = (E) es[(n = --size)];
335                es[n] = null;
336                if (n > 0) {
337                    final Comparator<? super E> cmp;
338                    if ((cmp = comparator) == null)
339    调整堆，执行siftDown操作 siftDownComparable( k: 0, x, es, n);
340                    else
341                        siftDownUsingComparator( k: 0, x, es, n, cmp);
342                }
343            }
344            return result;
345        }
```

从上面可以看到，在阻塞的实现方面，和ArrayBlockingQueue的机制相似，主要区别是用数组实现了一个二叉堆，从而实现按优先级从小到大出队列。另一个区别是没有notFull条件，当元素个数超出数组长度时，执行扩容操作。

### 5.1.4 DelayQueue

DelayQueue即延迟队列，也就是一个按延迟时间从小到大出队的PriorityQueue。所谓延迟时间，就是“未来将要执行的时间”减去“当前时间”。为此，放入DelayQueue中的元素，必须实现Delayed接口，如下所示。

src.zip 〉 java.base 〉 java 〉 util 〉 concurrent 〉 Delayed　　Main

Delayed.java

```
49  public interface Delayed extends Comparable<Delayed> {
50
51      /**
52       * Returns the remaining delay associated with this object, in the
53       * given time unit.
54       *
55       * @param unit the time unit
56       * @return the remaining delay; zero or negative values indicate
57       * that the delay has already elapsed
58       */
59      long getDelay( @NotNull() TimeUnit unit);
60  }
```

关于该接口：

1. 如果getDelay的返回值小于或等于0，则说明该元素到期，需要从队列中拿出来执行。

2. 该接口首先继承了 Comparable 接口，所以要实现该接口，必须实现 Comparable 接口。具体来说，就是基于getDelay()的返回值比较两个元素的大小。

下面看一下DelayQueue的核心数据结构。

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
implements BlockingQueue<E> {
    // ...
    // 一把锁和一个非空条件
    private final transient ReentrantLock lock = new ReentrantLock();
    private final Condition available = lock.newCondition();
    // 优先级队列
    private final PriorityQueue<E> q = new PriorityQueue<E>();
    // ...
}
```

下面介绍put/take的实现，先从take说起，因为这样更能看出DelayQueue的特性。

DelayQueue.java

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                E first = q.peek(); 取出二叉堆的堆顶元素，即延迟时间最小的
                if (first == null)
                    available.await(); 队列为空，take线程阻塞
                else {
                    long delay = first.getDelay(NANOSECONDS);
                    if (delay <= 0L) 堆顶元素的延迟时间小于等于0，出队列返回
                        return q.poll();
                    first = null; // don't retain ref while waiting
                    if (leader != null) 如果有其他线程也在等待该元素，则无限期
                        available.await(); 等待。
                    else {
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;
                        try {
                            available.awaitNanos(delay); 否则阻塞有限的时间
                        } finally {
                            if (leader == thisThread)
                                leader = null;
                        }
                    }
                }
            }
        } finally {
            if (leader == null && q.peek() != null)
                available.signal();当前线程是leader，已经获取了堆顶元素，唤醒其他线程
            lock.unlock();
        }
    }
```

关于take()方法：

1. 不同于一般的阻塞队列，只在队列为空的时候，才阻塞。如果堆顶元素的延迟时间没到，也会阻塞。
2. 在上面的代码中使用了一个优化技术，用一个Thread leader变量记录了等待堆顶元素的第1个线程。为什么这样做呢？通过 getDelay(..)可以知道堆顶元素何时到期，不必无限期等待，可以使用condition.awaitNanos()等待一个有限的时间；只有当发现还有其他线程也在等待堆顶元素（leader！=NULL）时，才需要无限期等待。

put的实现：

DelayQueue.java

```
165    public void put(E e) {
166        offer(e);
167    }
```

DelayQueue.java

```
143    public boolean offer(E e) {
144        final ReentrantLock lock = this.lock;
145        lock.lock();
146        try {
147            q.offer(e);元素放入二叉堆
148            if (q.peek() == e) {
149                leader = null;
150                available.signal();
151            }
152            return true;
153        } finally {
154            lock.unlock();
155        }
156    }
```

如果放进去的元素刚好在堆顶，说明放入的元素延迟时间最小，需要通知等待的线程；
否则放入的元素不在堆顶，没有必要通知等待的线程。

注意：不是每放入一个元素，都需要通知等待的线程。放入的元素，如果其延迟时间大于当前堆顶的元素延迟时间，就没必要通知等待的线程；只有当延迟时间是最小的，在堆顶时，才有必要通知等待的线程，也就是上面代码中的`if (q.peek() == e) {` 部分。

### 5.1.5 SynchronousQueue

SynchronousQueue是一种特殊的BlockingQueue，它本身没有容量。先调put(...)，线程会阻塞；直到另外一个线程调用了take()，两个线程才同时解锁，反之亦然。对于多个线程而言，例如3个线程，调用3次put(...)，3个线程都会阻塞；直到另外的线程调用3次take()，6个线程才同时解锁，反之亦然。

接下来看SynchronousQueue的实现。

构造方法：

SynchronousQueue.java

```
860    public SynchronousQueue(boolean fair) {
861        transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
862    }
```

和锁一样，也有公平和非公平模式。如果是公平模式，则用TransferQueue实现；如果是非公平模式，则用TransferStack实现。这两个类分别是什么呢？先看一下put/take的实现。

SynchronousQueue.java

```
871    public void put(E e) throws InterruptedException {
872        if (e == null) throw new NullPointerException();
873        if (transferer.transfer(e, timed: false, nanos: 0) == null) {
874            Thread.interrupted();
875            throw new InterruptedException();
876        }
877    }
```

SynchronousQueue.java

```
919    public E take() throws InterruptedException {
920        E e = transferer.transfer( e: null, timed: false, nanos: 0);
921        if (e != null)
922            return e;
923        Thread.interrupted();
924        throw new InterruptedException();
925    }
```

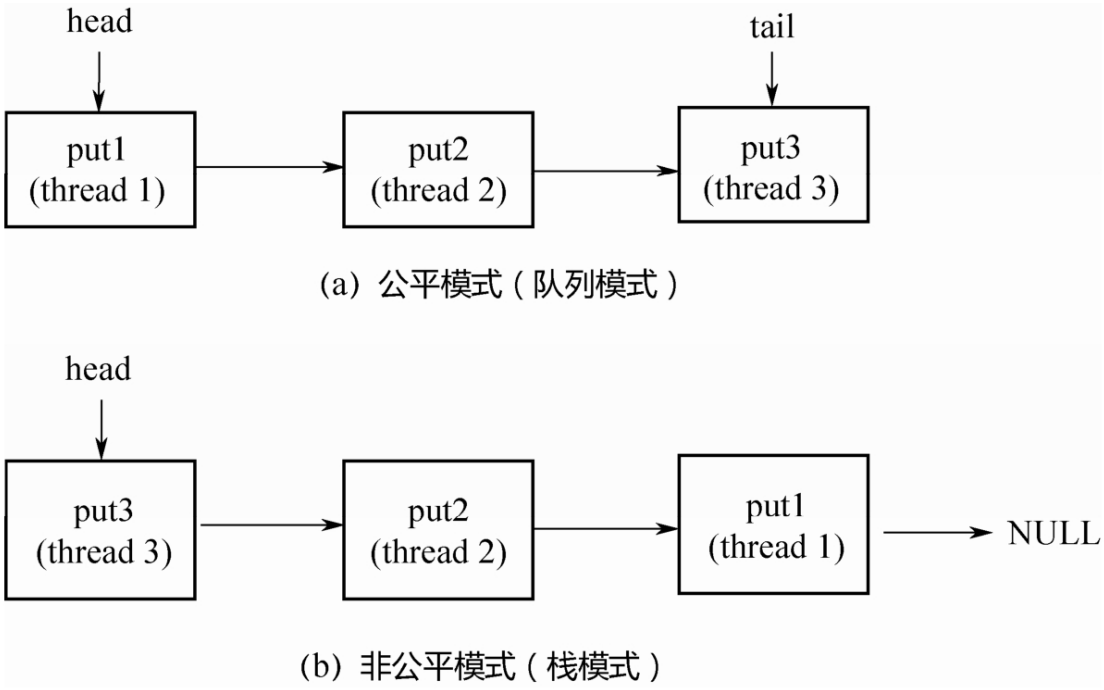
可以看到，put/take都调用了transfer(...)接口。而TransferQueue和TransferStack分别实现了这个接口。该接口在SynchronousQueue内部，如下所示。如果是put(...)，则第1个参数就是对应的元素；如果是take()，则第1个参数为null。后2个参数分别为是否设置超时和对应的超时时间。

SynchronousQueue.java

```
172     * Shared internal API for dual stacks and queues.
173     */
174    abstract static class Transferer<E> {
175        /**
176         * Performs a put or take.
177         *
178         * @param e if non-null, the item to be handed to a consumer;
179         *          if null, requests that transfer return an item
180         *          offered by producer.
181         * @param timed if this operation should timeout
182         * @param nanos the timeout, in nanoseconds
183         * @return if non-null, the item provided or received; if null,
184         *         the operation failed due to timeout or interrupt --
185         *         the caller can distinguish which of these occurred
186         *         by checking Thread.interrupted.
187         */
188        abstract E transfer(E e, boolean timed, long nanos);
189    }
```

接下来看一下什么是公平模式和非公平模式。假设3个线程分别调用了put(...)，3个线程会进入阻塞状态，直到其他线程调用3次take()，和3个put(...)一一配对。

如果是公平模式（队列模式），则第1个调用put(...)的线程1会在队列头部，第1个到来的take()线程和它进行配对，遵循先到先配对的原则，所以是公平的；如果是非公平模式（栈模式），则第3个调用put(...)的线程3会在栈顶，第1个到来的take()线程和它进行配对，遵循的是后到先配对的原则，所以是非公平的。




下面分别看一下TransferQueue和TransferStack的实现。

**1.TransferQueue**

```
public class SynchronousQueue<E> extends AbstractQueue<E> implements
BlockingQueue<E>, java.io.Serializable {
    // ...
    static final class TransferQueue<E> extends Transferer<E> {
        static final class QNode {
            volatile QNode next;
            volatile Object item;
            volatile Thread waiter;
            final boolean isData;
            //...
        }
        transient volatile QNode head;
        transient volatile QNode tail;
        // ...
    }
}
```

从上面的代码可以看出，TransferQueue是一个基于单向链表而实现的队列，通过head和tail 2个指针记录头部和尾部。初始的时候，head和tail会指向一个空节点，构造方法如下所示。

SynchronousQueue.java

```
601
602         TransferQueue() {
603             QNode h = new QNode( item: null,  isData: false); /
604             head = h;
605             tail = h;
606         }
```

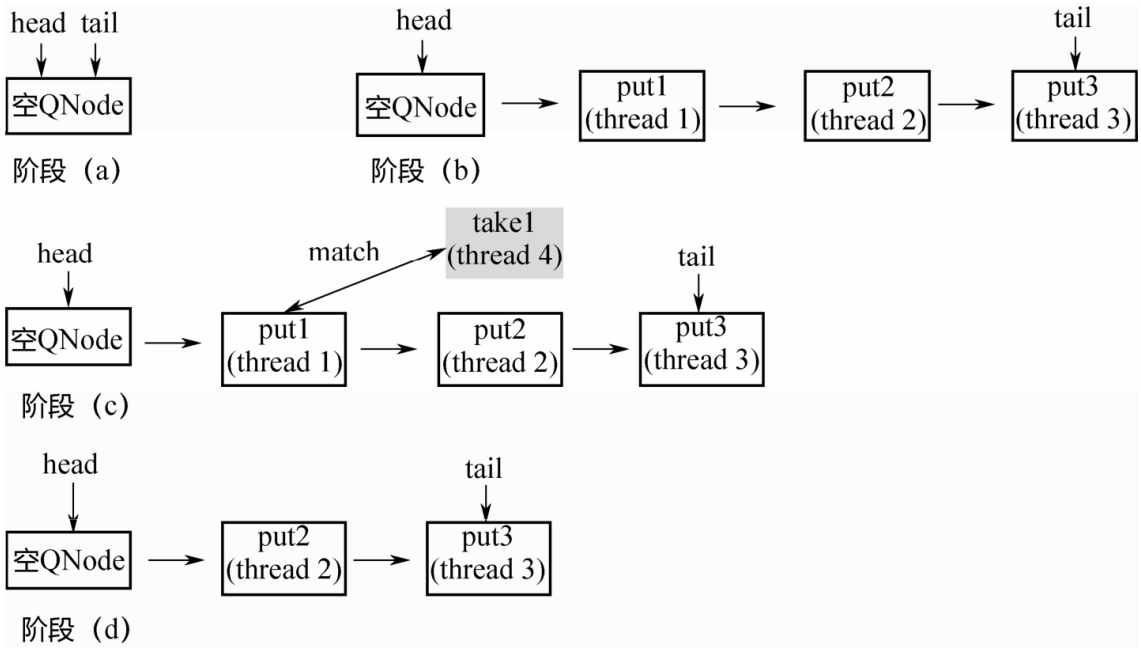
空节点

阶段（a）：队列中是一个空的节点，head/tail都指向这个空节点。

阶段（b）：3个线程分别调用put，生成3个QNode，进入队列。

阶段（c）：来了一个线程调用take，会和队列头部的第1个QNode进行配对。

阶段（d）：第1个QNode出队列。




这里有一个关键点：put节点和take节点一旦相遇，就会配对出队列，所以在队列中不可能同时存在put节点和take节点，要么所有节点都是put节点，要么所有节点都是take节点。

接下来看一下TransferQueue的代码实现。

SynchronousQueue.java

```
637         @SuppressWarnings("unchecked")
638         E transfer(E e, boolean timed, long nanos) {
```

```
664             QNode s = null; // constructed/reused as needed
665             boolean isData = (e != null);
666
667             for (;;) {
668                 QNode t = tail;
669                 QNode h = head;          队列还未初始化，自旋等待
670                 if (t == null || h == null)         // saw uninitialized val
671                     continue;                       // spin
672                                                  队列为空或者当前线程和队列中元素为同一种模式
673                 if (h == t || t.isData == isData) { // empty or same-mode
674                     QNode tn = t.next;
675                     if (t != tail) 不一致读，重新执行for循环inconsistent read
676                         continue;
677                     if (tn != null) {               // lagging tail
678                         advanceTail(t, tn);
679                         continue;
680                     }
681                     if (timed && nanos <= 0L)       // can't wait
682                         return null;
683                     if (s == null)
684                         s = new QNode(e, isData);新建一个节点
685       加入尾部      if (!t.casNext( cmp: null, s))        // failed to link i
686                         continue;
687
688       后移tail指针  advanceTail(t, s);              // swing tail and wait
689       进入阻塞状态  Object x = awaitFulfill(s, e, timed, nanos);
690                     if (x == s) {                   // wait was cancelled
691                         clean(t, s);
692                         return null;
693                     }
694                       从阻塞中唤醒，确定已经处于队列中的第1个元素
695                     if (!s.isOffList()) {           // not already unlinked
696                         advanceHead(t, s);          // unlink if head
697                         if (x != null)              // and forget fields
698                             s.item = s;
699                         s.waiter = null;
700                     }
701                     return (x != null) ? (E)x : e;
702                                                  当前线程可以和队列中的第1个元素
703                 } else {                        进行配对 // complementary-mode
704       取队列中第1个元素 QNode m = h.next;               // node to fulfill
705       不一致读，重新执行for循环 if (t != tail || m == null || h != head)
706                         continue;                   // inconsistent read
```

```
707
708                     Object x = m.item;
709                 已经配对 if (isData == (x != null) ||    // m already fulfilled
710                         x == m ||                        // m cancelled
711                     尝试配对 !m.casItem(x, e)) {         // lost CAS
712             已经配对，直接出队列 advanceHead(h, m);          // dequeue and retry
713                         continue;
714                     }
715
716             配对成功，出队列 advanceHead(h, m);              // successfully fulfille
717 唤醒队列中与第1个元素对应的线程 LockSupport.unpark(m.waiter);
718                     return (x != null) ? (E)x : e;
719                 }
720             }
721         }
```

整个 for 循环有两个大的 if-else 分支，如果当前线程和队列中的元素是同一种模式（都是put节点或者take节点），则与当前线程对应的节点被加入队列尾部并且阻塞；如果不是同一种模式，则选取队列头部的第1个元素进行配对。

这里的配对就是m.casItem（x，e），把自己的item x换成对方的item e，如果CAS操作成功，则配对成功。如果是put节点，则isData=true，item！=null；如果是take节点，则isData=false，item=null。如果CAS操作不成功，则isData和item之间将不一致，也就是isData！=（x！=null），通过这个条件可以判断节点是否已经被匹配过了。

**2.TransferStack**

TransferStack的定义如下所示，首先，它也是一个单向链表。不同于队列，只需要head指针就能实现入栈和出栈操作。

```
static final class TransferStack extends Transferer {
    static final int REQUEST = 0;
    static final int DATA = 1;
    static final int FULFILLING = 2;
    static final class SNode {
        volatile SNode next;     // 单向链表
        volatile SNode match;    // 配对的节点
        volatile Thread waiter;  // 对应的阻塞线程
        Object item;
        int mode;                // 三种模式
        //...
    }
    volatile SNode head;
}
```

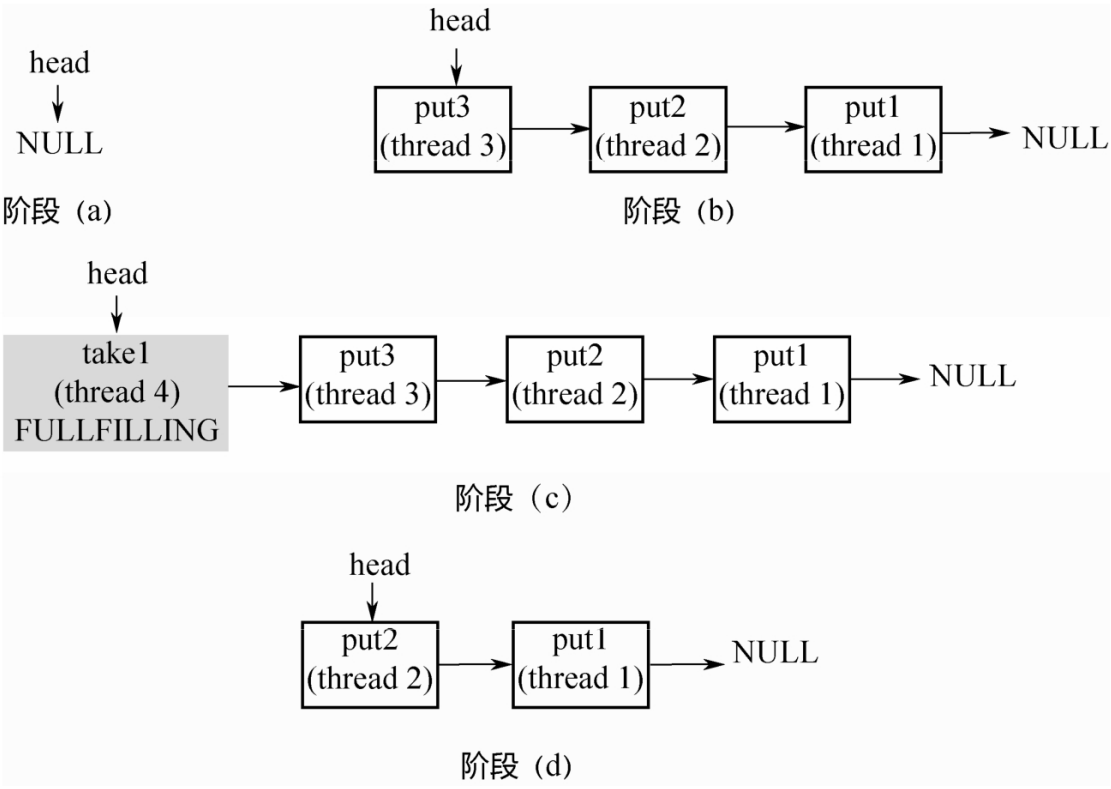
链表中的节点有三种状态，REQUEST对应take节点，DATA对应put节点，二者配对之后，会生成一个FULFILLING节点，入栈，然后FULLING节点和被配对的节点一起出栈。

阶段（a）：head指向NULL。不同于TransferQueue，这里没有空的头节点。

阶段（b）：3个线程调用3次put，依次入栈。

阶段（c）：线程4调用take，和栈顶的第1个元素配对，生成FULLFILLING节点，入栈。

阶段（d）：栈顶的2个元素同时入栈。




下面看一下具体的代码实现。

SynchronousQueue.java

```
327     E transfer(E e, boolean timed, long nanos) {

349             SNode s = null; // constructed/reused as needed
350             int mode = (e == null) ? REQUEST : DATA;
351
352             for (;;) {
353                 SNode h = head;
354     同一种模式  if (h == null || h.mode == mode) {  // empty or same-mode
355                     if (timed && nanos <= 0L) {     // can't wait
356                         if (h != null && h.isCancelled())
357                             casHead(h, h.next);     // pop cancelled node
358                         else
359                             return null;
360                     入栈 } else if (casHead(h, s = snode(s, e, h, mode))) {
361                      阻塞等待 SNode m = awaitFulfill(s, timed, nanos);
362                         if (m == s) {               // wait was cancelled
363                             clean(s);
364                             return null;
365                         }
366                         if ((h = head) != null && h.next == s)
367                             casHead(h, s.next);     // help s's fulfiller
368                         return (E) ((mode == REQUEST) ? m.item : s.item);
369                     }
370     非同一种模式，待匹配 } else if (!isFulfilling(h.mode)) { // try to fulfill
```

```
371                         if (h.isCancelled())            // already cancelled
372                             casHead(h, h.next);         // pop and retry
373 生成一个FULFILLING节点，入栈 else if (casHead(h, s=snode(s, e, h,  mode: FULFILLING|mode))) {
374                             for (;;) { // loop until matched or waiters disappear
375                                 SNode m = s.next;       // m is s's match
376                                 if (m == null) {        // all waiters are gone
377                                     casHead(s,  nh: null);   // pop fulfill node
378                                     s = null;           // use new node next time
379                                     break;              // restart main loop
380                                 }
381                                 SNode mn = m.next;
382                                 if (m.tryMatch(s)) {
383                         两个节点一起出栈 casHead(s, mn);     // pop both s and m
384                                     return (E) ((mode == REQUEST) ? m.item : s.item);
385                                 } else                  // lost match
386                                     s.casNext(m, mn);   // help unlink
387                             }
388                         }
389                     } else {   已经匹配过了，出栈          // help a fulfiller
390                         SNode m = h.next;               // m is h's match
391                         if (m == null)                  // waiter is gone
392                             casHead(h,  nh: null);           // pop fulfilling node
393                         else {
394                             SNode mn = m.next;
395                             if (m.tryMatch(h))          // help match
396                     配对，一起出栈 casHead(h, mn);         // pop both h and m
397                             else                        // lost match
398                                 h.casNext(m, mn);       // help unlink
399                         }
400                     }
401                 }
402             }
```

## 5.2 BlockingDeque

BlockingDeque定义了一个阻塞的双端队列接口，如下所示。

```
1  public interface BlockingDeque<E> extends BlockingQueue<E>, Deque<E> {
2      void putFirst(E e) throws InterruptedException;
3      void putLast(E e) throws InterruptedException;
4      E takeFirst() throws InterruptedException;
5      E takeLast() throws InterruptedException;
6      // ...
7  }
```

该接口继承了BlockingQueue接口，同时增加了对应的双端队列操作接口。该接口只有一个实现，就是LinkedBlockingDeque。

其核心数据结构如下所示，是一个双向链表。

```
1  public class LinkedBlockingDeque<E> extends AbstractQueue<E> implements
   BlockingDeque<E>, java.io.Serializable {
2      static final class Node<E> {
3          E item;
4          Node<E> prev;  // 双向链表的Node
5          Node<E> next;
```

```
        Node(E x) {
            item = x;
        }
    }

    transient Node<E> first;  // 队列的头和尾
    transient Node<E> last;
    private transient int count; // 元素个数
    private final int capacity;  // 容量
    // 一把锁+两个条件
    final ReentrantLock lock = new ReentrantLock();
    private final Condition notEmpty = lock.netCondition();
    private final Condition notFull = lock.newCondition();
    // ...
}
```

对应的实现原理，和LinkedBlockingQueue基本一样，只是LinkedBlockingQueue是单向链表，而LinkedBlockingDeque是双向链表。

LinkedBlockingDeque.java

```
    public E takeFirst() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            E x;
            while ( (x = unlinkFirst()) == null)
                notEmpty.await();
            return x;
        } finally {
            lock.unlock();
        }
    }
```

LinkedBlockingDeque.java

```
    public E takeLast() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            E x;
            while ( (x = unlinkLast()) == null)
                notEmpty.await();
            return x;
        } finally {
            lock.unlock();
        }
    }
```

LinkedBlockingDeque.java

```
363 @    public void putFirst(E e) throws InterruptedException {
364          if (e == null) throw new NullPointerException();
365          Node<E> node = new Node<E>(e);
366          final ReentrantLock lock = this.lock;
367          lock.lock();
368          try {
369              while (!linkFirst(node))
370                  notFull.await();
371          } finally {
372              lock.unlock();
373          }
374      }
```

LinkedBlockingDeque.java

```
380 @    public void putLast(E e) throws InterruptedException {
381          if (e == null) throw new NullPointerException();
382          Node<E> node = new Node<E>(e);
383          final ReentrantLock lock = this.lock;
384          lock.lock();
385          try {
386              while (!linkLast(node))
387                  notFull.await();
388          } finally {
389              lock.unlock();
390          }
391      }
```

## 5.3 CopyOnWrite

CopyOnWrite指在“写”的时候，不是直接“写”源数据，而是把数据拷贝一份进行修改，再通过悲观锁或者乐观锁的方式写回。

那为什么不直接修改，而是要拷贝一份修改呢?

这是为了在“读”的时候不加锁。

### 5.3.1 CopyOnWriteArrayList

和ArrayList一样，CopyOnWriteArrayList的核心数据结构也是一个数组，代码如下：

```
1 public class CopyOnWriteArrayList<E> implements List<E>, RandomAccess,
Cloneable, java.io.Serializable {
2     // ...
3     private volatile transient Object[] array;
4 }
```

下面是CopyOnArrayList的几个“读”方法：

```
    final Object[] getArray() {
        return array;
    }
    //
    public E get(int index) {
        return elementAt(getArray(), index);
    }
    public boolean isEmpty() {
        return size() == 0;
    }
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }
    public int indexOf(Object o) {
        Object[] es = getArray();
        return indexOfRange(o, es, 0, es.length);
    }
    private static int indexOfRange(Object o, Object[] es, int from, int to)
{
        if (o == null) {
            for (int i = from; i < to; i++)
                if (es[i] == null)
                    return i;
        } else {
            for (int i = from; i < to; i++)
                if (o.equals(es[i]))
                    return i;
        }
        return -1;
    }
```

既然这些“读”方法都没有加锁，那么是如何保证“线程安全”呢？答案在“写”方法里面。

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    // 锁对象
    final transient Object lock = new Object();

    public boolean add(E e) {
        synchronized (lock) { // 同步锁对象
            Object[] es = getArray();
            int len = es.length;
            es = Arrays.copyOf(es, len + 1); // CopyOnWrite，写的时候，先拷贝一
份之前的数组。
            es[len] = e;
            setArray(es);
            return true;
        }
    }

    public void add(int index, E element) {
        synchronized (lock) { // 同步锁对象
            Object[] es = getArray();
            int len = es.length;
            if (index > len || index < 0)
```

```
                throw new IndexOutOfBoundsException(outOfBounds(index,
len));
            Object[] newElements;
            int numMoved = len - index;
            if (numMoved == 0)
                newElements = Arrays.copyOf(es, len + 1);
            else {
                newElements = new Object[len + 1];
                System.arraycopy(es, 0, newElements, 0, index); //
CopyOnWrite，写的时候，先拷贝一份之前的数组。
                System.arraycopy(es, index, newElements, index + 1,
numMoved);
            }
            newElements[index] = element;
            setArray(newElements); // 把新数组赋值给老数组
        }
    }
}
```

其他"写"方法，例如remove和add类似，此处不再详述。

#### 5.3.2 CopyOnWriteArraySet

CopyOnWriteArraySet 就是用 Array 实现的一个 Set，保证所有元素都不重复。其内部是封装的一个CopyOnWriteArrayList。

```
public class CopyOnWriteArraySet<E> extends AbstractSet<E> implements
java.io.Serializable {
    // 新封装的CopyOnWriteArrayList
    private final CopyOnWriteArrayList<E> al;

    public CopyOnWriteArraySet() {
        al = new CopyOnWriteArrayList<E>();
    }

    public boolean add(E e) {
        return al.addIfAbsent(e); // 不重复的加进去
    }
}
```

### 5.4 ConcurrentLinkedQueue/Deque

AQS内部的阻塞队列实现原理：基于双向链表，通过对head/tail进行CAS操作，实现入队和出队。

ConcurrentLinkedQueue 的实现原理和AQS 内部的阻塞队列类似：同样是基于 CAS，同样是通过head/tail指针记录队列头部和尾部，但还是有稍许差别。

首先，它是一个单向链表，定义如下：

```
public class ConcurrentLinkedQueue<E> extends AbstractQueue<E> implements
Queue<E>, java.io.Serializable {
    private static class Node<E> {
        volatile E item;
        volatile Node<E> next;
        //...
    }
    private transient volatile Node<E> head;
    private transient volatile Node<E> tail;
    //...
}
```

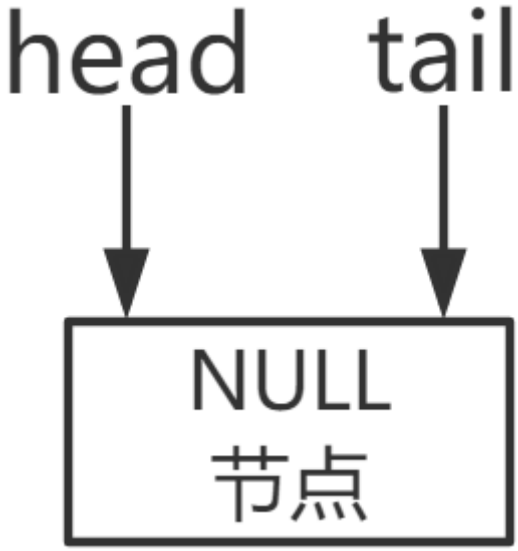
其次，在AQS的阻塞队列中，每次入队后，tail一定后移一个位置；每次出队，head一定后移一个位置，以保证head指向队列头部，tail指向链表尾部。

但在ConcurrentLinkedQueue中，head/tail的更新可能落后于节点的入队和出队，因为它不是直接对 head/tail指针进行 CAS操作的，而是对 Node中的 item进行操作。下面进行详细分析：

**1.初始化**

初始的时候，`head`和`tail`都指向一个`null`节点。对应的代码如下。

```
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>(null);
}
```




**2.入队列**

代码如下所示。

```
public boolean offer(E e) {
```

```
    final Node<E> newNode = new Node<E>(Objects.requireNonNull(e));

    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            if (NEXT.compareAndSet(p, null, newNode)) {
                if (p != t)
                    TAIL.weakCompareAndSet(this, t, newNode);
                return true
            }
        }
        else if (p == q)
            p = (t != (t = tail)) ? t : head;
        else
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

ConcurrentLinkedQueue.java

```
     */
    public boolean offer(E e) {
        final Node<E> newNode = new Node<E>(Objects.requireNonNull(e));

        for (Node<E> t = tail, p = t;;) {
            Node<E> q = p.next;
            if (q == null) {
                // p is last node
                if (NEXT.compareAndSet( ...args: p, null, newNode)) {
                    // Successful CAS is the linearization point
                    // for e to become an element of this queue,
                    // and for newNode to become "live".
                    if (p != t) // hop two nodes at a time; failure is OK
                        TAIL.weakCompareAndSet( ...args: this, t, newNode);
                    return true;
                }
                // Lost CAS race to another thread; re-read next
            }
            else if (p == q)
                // We have fallen off list.  If tail is unchanged, it
                // will also be off-list, in which case we need to
                // jump to head, from which all live nodes are always
                // reachable.  Else the new tail is a better bet.
                p = (t != (t = tail)) ? t : head;
            else
                // Check for tail updates after two hops.
                p = (p != t && t != (t = tail)) ? t : q;
        }
    }
```

对tail的next指针而不是对tail指针执行CAS操作。

每入列两个节点，后移一次tail指针失败也无所谓了。

已经到达队列尾部

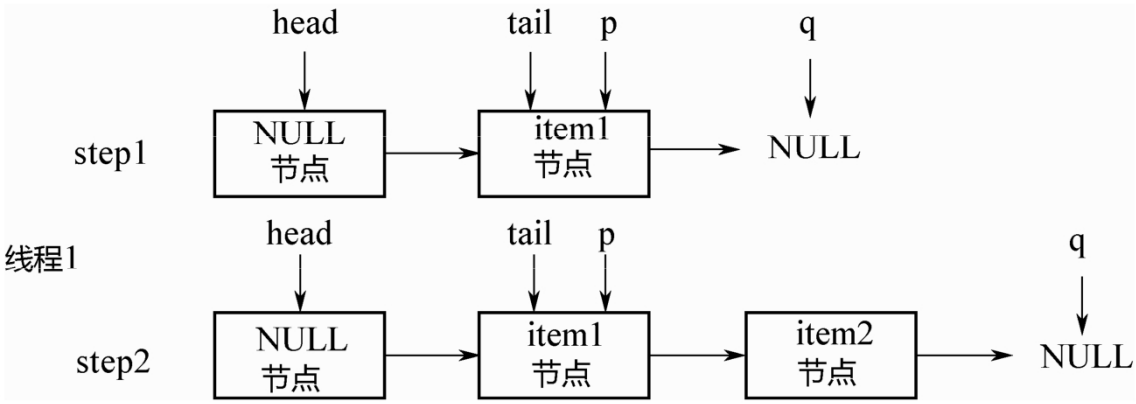
后移p指针

上面的入队其实是每次在队尾追加2个节点时，才移动一次tail节点。如下图所示：

初始的时候，队列中有1个节点item1，tail指向该节点，假设线程1要入队item2节点：

step1:p=tail,q=p.next=NULL.

step2：对p的next执行CAS操作，追加item2，成功之后，p=tail。所以上面的casTail方法不会执行，直接返回。此时tail指针没有变化。



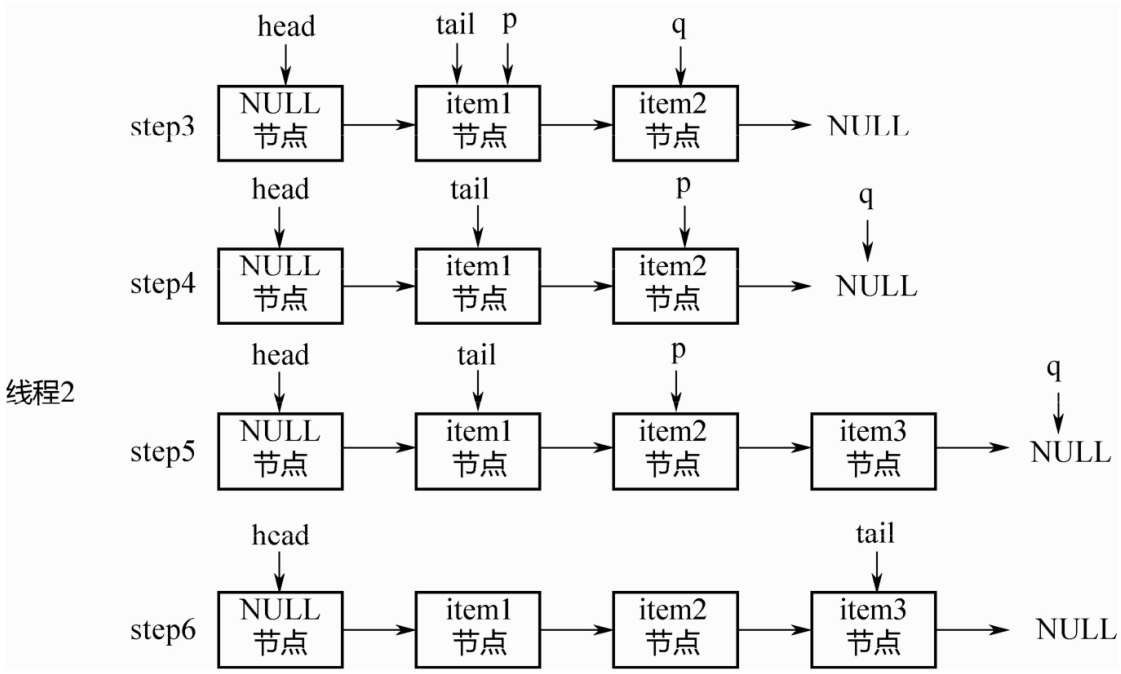

之后，假设线程2要入队item3节点，如下图所示：

step3:p=tail,q=p.next.

step4：q！=NULL，因此不会入队新节点。p，q都后移1位。

step5：q=NULL，对p的next执行CAS操作，入队item3节点。

step6：p！=t，满足条件，执行上面的casTail操作，tail后移2个位置，到达队列尾部。




最后总结一下入队列的两个关键点：

1. 即使tail指针没有移动，只要对p的next指针成功进行CAS操作，就算成功入队列。
2. 只有当 p != tail的时候，才会后移tail指针。也就是说，每连续追加2个节点，才后移1次tail指针。即使CAS失败也没关系，可以由下1个线程来移动tail指针。

**3.出队列**

上面说了入队列之后，tail指针不变化，那是否会出现入队列之后，要出队列却没有元素可出的情况呢?

ConcurrentLinkedQueue.java

```
383     public E poll() {
384         restartFromHead: for (;;) {
385             for (Node<E> h = head, p = h, q;; p = q) {
386                 final E item;
387                 if ((item = p.item) != null && p.casItem(item, val: null)) {
388                     // Successful CAS is the linearization point
389                     // for item to be removed from this queue.
390                     if (p != h) // hop two nodes at a time
391                         updateHead(h, ((q = p.next) != null) ? q : p);
392                     return item;
393                 }
394                 else if ((q = p.next) == null) {
395                     updateHead(h, p);
396                     return null;
397                 }
398                 else if (p == q)
399                     continue restartFromHead;
400             }
401         }
402     }
```

每产生2个NULL节点，才把head指针后移2位

注意：在出队列的时候，并没有移动head指针，而是把item置为null

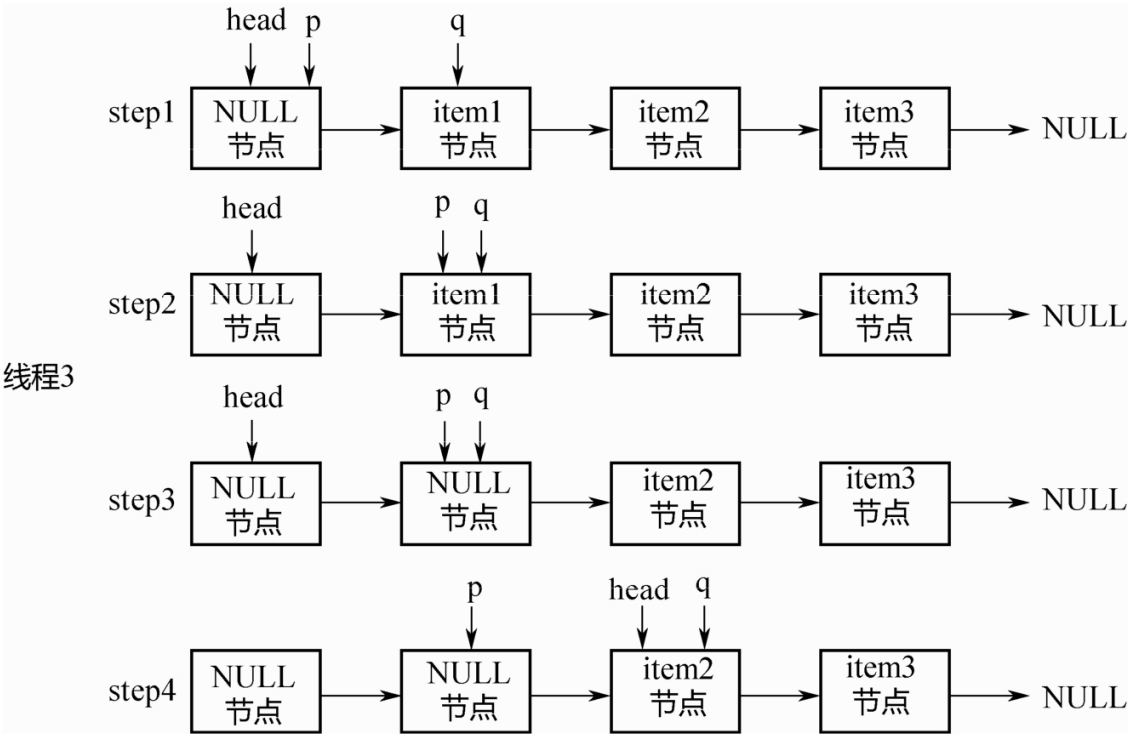
出队列的代码和入队列类似，也有p、q2个指针，整个变化过程如图5-8所示。假设初始的时候head指向空节点，队列中有item1、item2、item3 三个节点。

step1:p=head,q=p.next.p!=q.

step2：后移p指针，使得p=q。

step3：出队列。关键点：此处并没有直接删除item1节点，只是把该节点的item通过CAS操作置为了NULL。

step4：p！=head，此时队列中有了2个 NULL 节点，再前移1次head指针，对其执行updateHead操作。




最后总结一下出队列的关键点：

1. 出队列的判断并非观察 tail 指针的位置，而是依赖于 head 指针后续的节点是否为NULL这一条件。

2. 只要对节点的item执行CAS操作，置为NULL成功，则出队列成功。即使head指针没有成功移动，也可以由下1个线程继续完成。

#### 4.队列判空

因为head/tail 并不是精确地指向队列头部和尾部，所以不能简单地通过比较 head/tail 指针来判断队列是否为空，而是需要从head指针开始遍历，找第1个不为NULL的节点。如果找到，则队列不为空；如果找不到，则队列为空。代码如下所示：

ConcurrentLinkedQueue.java

```
446     public boolean isEmpty() {
447         return first() == null;
448     }
```

寻找第一个不是NULL的节点

ConcurrentLinkedQueue.java

```
427     Node<E> first() {
428         restartFromHead: for (;;) {
429             for (Node<E> h = head, p = h, q;; p = q) {
430                 boolean hasItem = (p.item != null);
431                 if (hasItem || (q = p.next) == null) {
432                     updateHead(h, p);
433                     return hasItem ? p : null;
434                 }
435                 else if (p == q)
436                     continue restartFromHead;
437             }
438         }
439     }
```

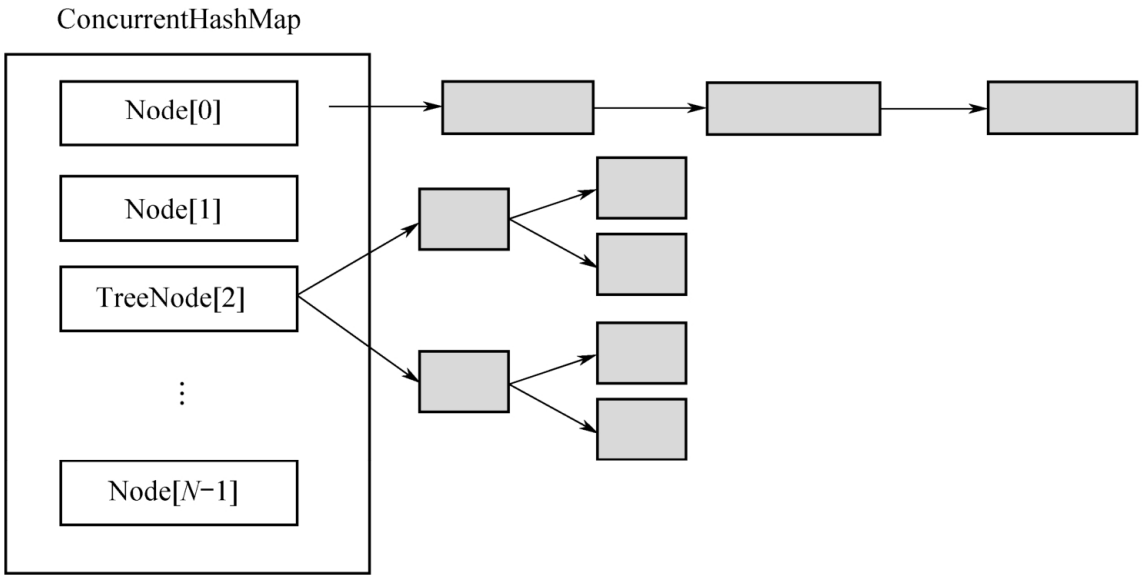
从head指针开始遍历，查找第一个不是null的节点

## 5.5 ConcurrentHashMap

HashMap通常的实现方式是“数组+链表”，这种方式被称为“拉链法”。ConcurrentHashMap在这个基本原理之上进行了各种优化。

首先是所有数据都放在一个大的HashMap中；其次是引入了红黑树。

其原理如下图所示：




如果头节点是Node类型，则尾随它的就是一个普通的链表；如果头节点是TreeNode类型，它的后面就是一颗红黑树，TreeNode是Node的子类。

链表和红黑树之间可以相互转换：初始的时候是链表，当链表中的元素超过某个阈值时，把链表转换成红黑树；反之，当红黑树中的元素个数小于某个阈值时，再转换为链表。

那为什么要做这种设计呢?

1. 使用红黑树，当一个槽里有很多元素时，其查询和更新速度会比链表快很多，Hash冲突的问题由此得到较好的解决。
2. 加锁的粒度，并非整个ConcurrentHashMap，而是对每个头节点分别加锁，即并发度，就是Node数组的长度，初始长度为16。
3. 并发扩容，这是难度最大的。当一个线程要扩容Node数组的时候，其他线程还要读写，因此处理过程很复杂，后面会详细分析。

由上述对比可以总结出来：这种设计一方面降低了Hash冲突，另一方面也提升了并发度。

下面从构造方法开始，一步步深入分析其实现过程。

**1.构造方法分析**

ConcurrentHashMap.java

```
841    public ConcurrentHashMap(int initialCapacity) {
842        this(initialCapacity, LOAD_FACTOR, concurrencyLevel: 1);
843    }
```

ConcurrentHashMap.java

```
892     public ConcurrentHashMap(int initialCapacity,
893                              float loadFactor, int concurrencyLevel) {
894         if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
895             throw new IllegalArgumentException();
896         if (initialCapacity < concurrencyLevel)   // Use at least as many bins
897             initialCapacity = concurrencyLevel;   // as estimated threads
898         long size = (long)(1.0 + (long)initialCapacity / loadFactor);
899         int cap = (size >= (long)MAXIMUM_CAPACITY) ?
900             MAXIMUM_CAPACITY : tableSizeFor((int)size);
901         this.sizeCtl = cap;
902     }
```

在上面的代码中，变量cap就是Node数组的长度，保持为2的整数次方。tableSizeFor(...)方法是根据传入的初始容量，计算出一个合适的数组长度。具体而言：1.5倍的初始容量+1，再往上取最接近的2的整数次方，作为数组长度cap的初始值。

这里的 sizeCtl，其含义是用于控制在初始化或者并发扩容时候的线程数，只不过其初始值设置成cap。

### 2.初始化

在上面的构造方法里只计算了数组的初始大小，并没有对数组进行初始化。当多个线程都往里面放入元素的时候，再进行初始化。这就存在一个问题：多个线程重复初始化。下面看一下是如何处理的。

```
1  private final Node<K,V>[] initTable() {
2      Node<K,V>[] tab; int sc;
3      while ((tab = table) == null || tab.length == 0) {
4          if ((sc = sizeCtl) < 0)
5              Thread.yield(); // 自旋等待
6          else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {   // 重点：将
   sizeCtl设置为-1
7              try {
8                  if ((tab = table) == null || tab.length == 0) {
9                      int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
10                     @SuppressWarnings("unchecked")
11                     Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n]; // 初始化
12                     table = tab = nt;
13                     // sizeCtl不是数组长度，因此初始化成功后，就不再等于数组长度
14                     // 而是n-(n>>>2)=0.75n，表示下一次扩容的阈值：n-n/4
15                     sc = n - (n >>> 2);
16                 }
17             } finally {
18                 sizeCtl = sc;  // 设置sizeCtl的值为sc。
19             }
20             break;
21         }
22     }
23     return tab;
24 }
```

通过上面的代码可以看到，多个线程的竞争是通过对sizeCtl进行CAS操作实现的。如果某个线程成功地把 sizeCtl 设置为-1，它就拥有了初始化的权利，进入初始化的代码模块，等到初始化完成，再把sizeCtl设置回去；其他线程则一直执行while循环，自旋等待，直到数组不为null，即当初始化结束时，退出整个方法。

因为初始化的工作量很小，所以此处选择的策略是让其他线程一直等待，而没有帮助其初始化。

### 3.put（..）实现分析

ConcurrentHashMap.java

```
1005 @    public V put( @NotNull K key,  @NotNull V value) {
1006          return putVal(key, value,  onlyIfAbsent: false);
1007      }
```

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh; K fk; V fv;
        // 分支1：整个数组初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 分支2：第i个元素初始化
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
                break;                   // no lock when adding to empty bin
        }
        // 分支3：扩容
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        //
        else if (onlyIfAbsent // check first node without acquiring lock
                 && fh == hash
                 && ((fk = f.key) == key || (fk != null && key.equals(fk)))
                 && (fv = f.val) != null)
            return fv;
        // 分支4：放入元素
        else {
            V oldVal = null;
            // 重点：加锁
            synchronized (f) {
                // 链表
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
```

```
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key, value);
                                break;
                            }
                        }
                    }
                    else if (f instanceof TreeBin) { // 红黑树
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                    else if (f instanceof ReservationNode)
                        throw new IllegalStateException("Recursive update");
                }
            }
            // 如果是链表，上面的binCount会一直累加
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i); // 超出阈值，转换为红黑树
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    addCount(1L, binCount); // 总元素个数累加1
    return null;
}
```

上面的for循环有4个大的分支：

第1个分支，是整个数组的初始化，前面已讲；

第2个分支，是所在的槽为空，说明该元素是该槽的第一个元素，直接新建一个头节点，然后返回；

第3个分支，说明该槽正在进行扩容，帮助其扩容；

第4个分支，就是把元素放入槽内。槽内可能是一个链表，也可能是一棵红黑树，通过头节点的类型可以判断是哪一种。第4个分支是包裹在synchronized （f）里面的，f对应的数组下标位置的头节点，意味着每个数组元素有一把锁，并发度等于数组的长度。

上面的binCount表示链表的元素个数，当这个数目超过TREEIFY_THRESHOLD=8时，把链表转换成红黑树，也就是 treeifyBin（tab，i）方法。但在这个方法内部，不一定需要进行红黑树转换，可能只做扩容操作，所以接下来从扩容讲起。

#### 4.扩容

扩容的实现是最复杂的，下面从treeifyBin(Node<K,V>[] tab, int index)讲起。

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            // 数组长度小于阈值64，不做红黑树转换，直接扩容
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            // 链表转换为红黑树
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    // 遍历链表，初始化红黑树
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

在上面的代码中，MIN_TREEIFY_CAPACITY=64，意味着当数组的长度没有超过64的时候，数组的每个节点里都是链表，只会扩容，不会转换成红黑树。只有当数组长度大于或等于64时，才考虑把链表转换成红黑树。

ConcurrentHashMap.java

```
559         */
560        static final int MIN_TREEIFY_CAPACITY = 64;
```

在 tryPresize（int size）内部调用了一个核心方法 transfer（Node < K，V > [] tab，Node < K，V > [] nextTab），先从这个方法的分析说起。

```
private final void tryPresize(int size) {
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;
        if (tab == null || (n = tab.length) == 0) {
```

```
            n = (sc > c) ? sc : c;
            if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2);
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        else if (tab == table) {
            int rs = resizeStamp(n);
            if (U.compareAndSetInt(this, SIZECTL, sc,
                                   (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
        }
    }
}
```

```
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // 计算步长
    if (nextTab == null) {            // 初始化新的HashMap
        try {
            @SuppressWarnings("unchecked")
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1]; // 扩容两倍
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        // 初始的transferIndex为旧HashMap的数组长度
        transferIndex = n;
    }
    int nextn = nextTab.length;
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    // 此处，i为遍历下标，bound为边界。
    // 如果成功获取一个任务，则i=nextIndex-1
    // bound=nextIndex-stride；
    // 如果获取不到，则i=0，bound=0
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        // advance表示在从i=transferIndex-1遍历到bound位置的过程中，是否一直继续
        while (advance) {
            int nextIndex, nextBound;
```

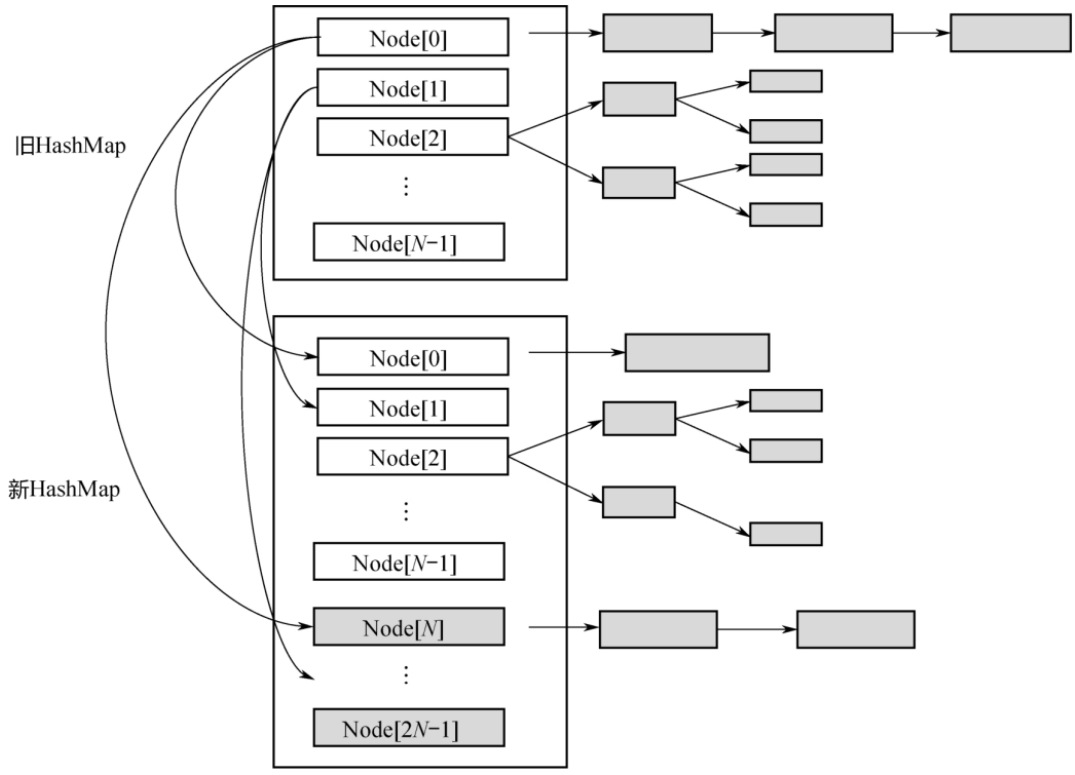
```
31
32              // 以下是哪个分支中的advance都是false，表示如果三个分支都不执行，才可以一直while循环
33              // 目的在于当对transferIndex执行CAS操作不成功的时候，需要自旋，以期获取一个stride的迁移任务。
34              if (--i >= bound || finishing)
35                  // 对数组遍历，通过这里的--i进行。如果成功执行了--i，就不需要继续while循环了，因为advance只能进一步。
36                  advance = false;
37              else if ((nextIndex = transferIndex) <= 0) {
38                  // transferIndex <= 0，整个HashMap完成
39                  i = -1;
40                  advance = false;
41              }
42              // 对transferIndex执行CAS操作，即为当前线程分配1个stride。
43              // CAS操作成功，线程成功获取到一个stride的迁移任务；
44              // CAS操作不成功，线程没有抢到任务，会继续执行while循环，自旋。
45              else if (U.compareAndSetInt
46                       (this, TRANSFERINDEX, nextIndex,
47                        nextBound = (nextIndex > stride ?
48                                     nextIndex - stride : 0))) {
49                  bound = nextBound;
50                  i = nextIndex - 1;
51                  advance = false;
52              }
53          }
54          // i越界，整个HashMap遍历完成
55          if (i < 0 || i >= n || i + n >= nextn) {
56              int sc;
57              // finishing表示整个HashMap扩容完成
58              if (finishing) {
59                  nextTable = null;
60                  // 将nextTab赋值给当前table
61                  table = nextTab;
62                  sizeCtl = (n << 1) - (n >>> 1);
63                  return;
64              }
65              if (U.compareAndSetInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
66                  if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
67                      return;
68                  finishing = advance = true;
69                  i = n; // recheck before commit
70              }
71          }
72          // tab[i]迁移完毕，赋值一个ForwardingNode
73          else if ((f = tabAt(tab, i)) == null)
74              advance = casTabAt(tab, i, null, fwd);
75          // tab[i]的位置已经在迁移过程中
76          else if ((fh = f.hash) == MOVED)
77              advance = true; // already processed
78          else {
79              // 对tab[i]进行迁移操作，tab[i]可能是一个链表或者红黑树
80              synchronized (f) {
81                  if (tabAt(tab, i) == f) {
82                      Node<K,V> ln, hn;
83                      // 链表
84                      if (fh >= 0) {
85                          int runBit = fh & n;
```

```
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                // 表示lastRun之后的所有元素，hash值都是一样的
                                // 记录下这个最后的位置
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            // 链表迁移的优化做法
                            ln = lastRun;
                            hn = null;
                        }
                        else {
                            hn = lastRun;
                            ln = null;
                        }
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                    // 红黑树，迁移做法和链表类似
                    else if (f instanceof TreeBin) {
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> lo = null, loTail = null;
                        TreeNode<K,V> hi = null, hiTail = null;
                        int lc = 0, hc = 0;
                        for (Node<K,V> e = t.first; e != null; e = e.next)
{
                            int h = e.hash;
                            TreeNode<K,V> p = new TreeNode<K,V>
                                (h, e.key, e.val, null, null);
                            if ((h & n) == 0) {
                                if ((p.prev = loTail) == null)
                                    lo = p;
                                else
                                    loTail.next = p;
                                loTail = p;
                                ++lc;
                            }
                            else {
                                if ((p.prev = hiTail) == null)
                                    hi = p;
                                else
                                    hiTail.next = p;
                                hiTail = p;
                                ++hc;
                            }
```

```
                        }
                        ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                            (hc != 0) ? new TreeBin<K,V>(lo) : t;
                        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                            (lc != 0) ? new TreeBin<K,V>(hi) : t;
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                }
            }
        }
    }
}
```

该方法非常复杂，下面一步步分析：

1. 扩容的基本原理如下图，首先建一个新的HashMap，其数组长度是旧数组长度的2倍，然后把旧的元素逐个迁移过来。所以，上面的方法参数有2个，第1个参数tab是扩容之前的HashMap，第2个参数nextTab是扩容之后的HashMap。当nextTab=null的时候，方法最初会对nextTab进行初始化。这里有一个关键点要说明：该方法会被多个线程调用，所以每个线程只是扩容旧的HashMap部分，这就涉及如何划分任务的问题。




2. 上图为多个线程并行扩容-任务划分示意图。旧数组的长度是N，每个线程扩容一段，一段的长度用变量stride（步长）来表示，transferIndex表示了整个数组扩容的进度。

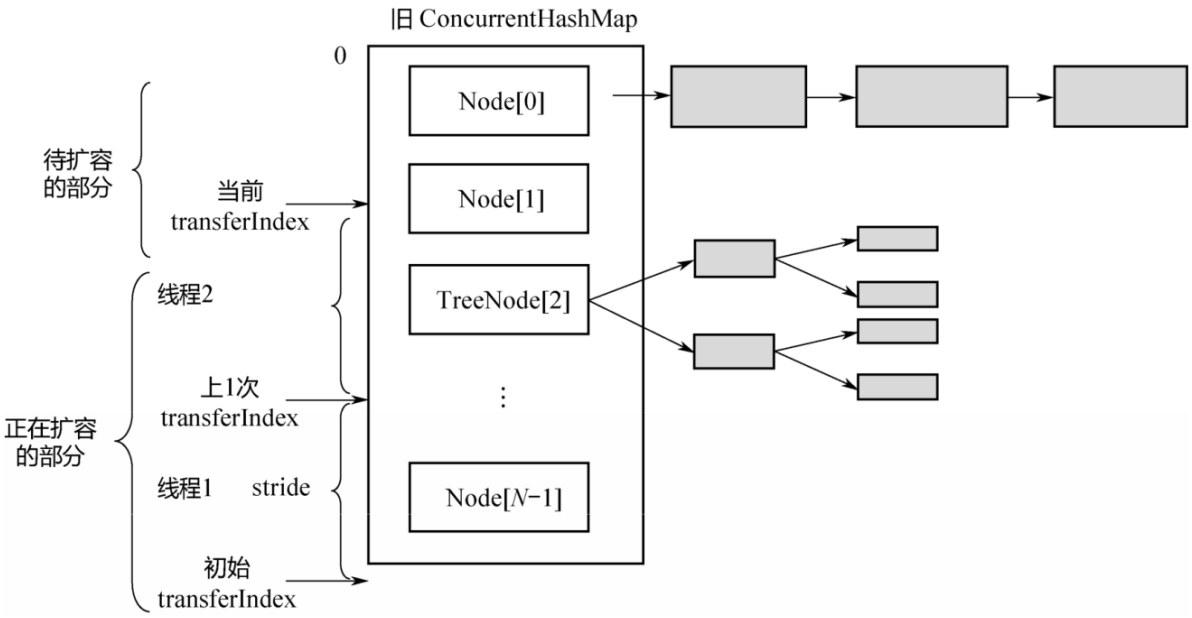
   stride的计算公式如上面的代码所示，即：在单核模式下直接等于n，因为在单核模式下没有办法多个线程并行扩容，只需要1个线程来扩容整个数组；在多核模式下为 （n > > > 3）/NCPU，并且保证步长的最小值是 16。显然，需要的线程个数约为n/stride。

```
ConcurrentHashMap.java
2419        int n = tab.length, stride;
2420        if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
2421            stride = MIN_TRANSFER_STRIDE; // subdivide range
```

transferIndex是ConcurrentHashMap的一个成员变量，记录了扩容的进度。初始值为n，从大到小扩容，每次减stride个位置，最终减至n < =0，表示整个扩容完成。因此，从[0，transferIndex-1]的位置表示还没有分配到线程扩容的部分，从[transfexIndex，n-1]的位置表示已经分配给某个线程进行扩容，当前正在扩容中，或者已经扩容成功。

因为transferIndex会被多个线程并发修改，每次减stride，所以需要通过CAS进行操作，如下面的代码所示。

```
ConcurrentHashMap.java
2446            advance = false;
2447        }
2448        else if (U.compareAndSetInt
2449                 ( o: this, TRANSFERINDEX, nextIndex,
2450                  nextBound = (nextIndex > stride ?
2451                               nextIndex - stride : 0))) {
```

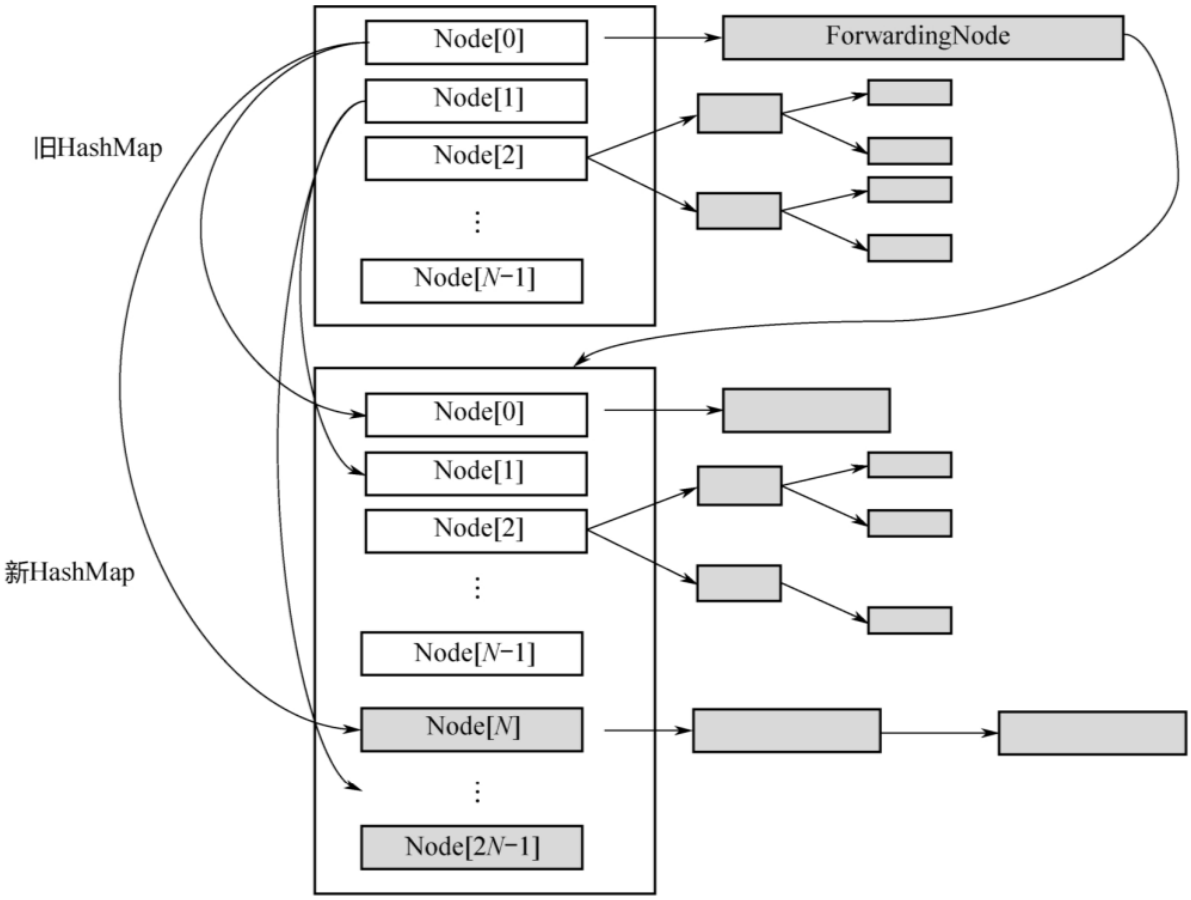



3. 在扩容未完成之前，有的数组下标对应的槽已经迁移到了新的HashMap里面，有的还在旧的HashMap 里面。这个时候，所有调用 get（k，v）的线程还是会访问旧 HashMap，怎么处理呢?

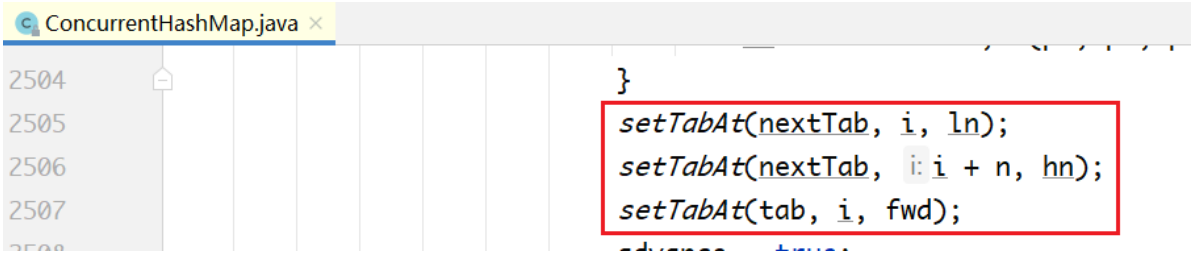
   下图为扩容过程中的转发示意图：当Node[0]已经迁移成功，而其他Node还在迁移过程中时，如果有线程要读取Node[0]的数据，就会访问失败。为此，新建一个ForwardingNode，即转发节点，在这个节点里面记录的是新的 ConcurrentHashMap 的引用。这样，当线程访问到ForwardingNode之后，会去查询新的ConcurrentHashMap。

4. 因为数组的长度 tab.length 是2的整数次方，每次扩容又是2倍。而 Hash 函数是 hashCode%tab.length，等价于hashCode&（tab.length-1）。这意味着：处于第i个位置的元素，在新的Hash表的数组中一定处于第i个或者第i+n个位置，如下图所示。举个简单的例子：假设数组长度是8，扩容之后是16：
   若hashCode=5，5%8=0，扩容后，5%16=0，位置保持不变；

若hashCode=24，24%8=0，扩容后，24%16=8，后移8个位置；
若hashCode=25，25%8=1，扩容后，25%16=9，后移8个位置；
若hashCode=39，39%8=7，扩容后，39%8=7，位置保持不变；
……




正因为有这样的规律，所以如下有代码：




也就是把tab[i]位置的链表或红黑树重新组装成两部分，一部分链接到nextTab[i]的位置，一部分链接到nextTab[i+n]的位置，如上图所示。然后把tab[i]的位置指向一个ForwardingNode节点。

同时，当tab[i]后面是链表时，使用类似于JDK 7中在扩容时的优化方法，从lastRun往后的所有节点，不需依次拷贝，而是直接链接到新的链表头部。从lastRun往前的所有节点，需要依次拷贝。

了解了核心的迁移函数transfer（tab，nextTab），再回头看tryPresize（int size）函数。这个函数的输入是整个Hash表的元素个数，在函数里面，根据需要对整个Hash表进行扩容。想要看明白这个函数，需要透彻地理解sizeCtl变量，下面这段注释摘自源码。

ConcurrentHashMap.java

```
790         private transient volatile long baseCount;
791
792         /**
793          * Table initialization and resizing control.  When negative, the
794          * table is being initialized or resized: -1 for initialization,
795          * else -(1 + the number of active resizing threads).  Otherwise,
796          * when table is null, holds the initial table size to use upon
797          * creation, or 0 for default. After initialization, holds the
798          * next element count value upon which to resize the table.
799          */
800         private transient volatile int sizeCtl;
```

当sizeCtl=-1时，表示整个HashMap正在初始化；

当sizeCtl=某个其他负数时，表示多个线程在对HashMap做并发扩容；

当sizeCtl=cap时，tab=null，表示未初始之前的初始容量（如上面的构造函数所示）；

扩容成功之后，sizeCtl存储的是下一次要扩容的阈值，即上面初始化代码中的n-（n>>>2）=0.75n。

所以，sizeCtl变量在Hash表处于不同状态时，表达不同的含义。明白了这个道理，再来看上面的tryPresize（int size）函数。

```java
private final void tryPresize(int size) {
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
            if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2);
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        else if (tab == table) {
            int rs = resizeStamp(n);
            if (U.compareAndSetInt(this, SIZECTL, sc,
                                   (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
        }
    }
}
```

tryPresize（int size）是根据期望的元素个数对整个Hash表进行扩容，核心是调用transfer函数。在第一次扩容的时候，sizeCtl会被设置成一个很大的负数U.compareAndSwapInt（this，SIZECTL，sc，（rs < < RESIZE_STAMP_SHIFT）+2）；之后每一个线程扩容的时候，sizeCtl 就加 1，U.compareAndSwapInt（this，SIZECTL，sc，sc+1），待扩容完成之后，sizeCtl减1。

## 5.6 ConcurrentSkipListMap/Set

ConcurrentHashMap 是一种 key 无序的 HashMap，ConcurrentSkipListMap则是 key 有序的，实现了NavigableMap接口，此接口又继承了SortedMap接口。

### 5.6.1 ConcurrentSkipListMap

**1.为什么要使用SkipList实现Map?**

在Java的util包中，有一个非线程安全的HashMap，也就是TreeMap，是key有序的，基于红黑树实现。

而在Concurrent包中，提供的key有序的HashMap，也就是ConcurrentSkipListMap，是基于SkipList（跳查表）来实现的。这里为什么不用红黑树，而用跳查表来实现呢?

借用Doug Lea的原话：

```
The reason is that there are no known efficient lockOfree insertion and
deletion algorithms for search trees.
```

也就是目前计算机领域还未找到一种高效的、作用在树上的、无锁的、增加和删除节点的办法。

那为什么SkipList可以无锁地实现节点的增加、删除呢？这要从无锁链表的实现说起。

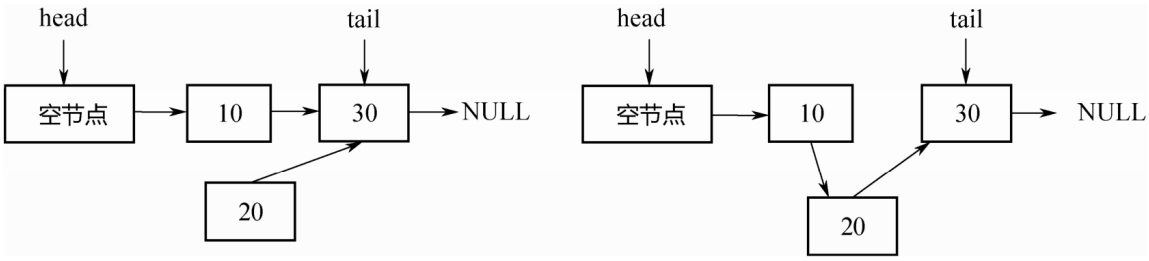
**2.无锁链表**

在前面讲解AQS时，曾反复用到无锁队列，其实现也是链表。究竟二者的区别在哪呢?

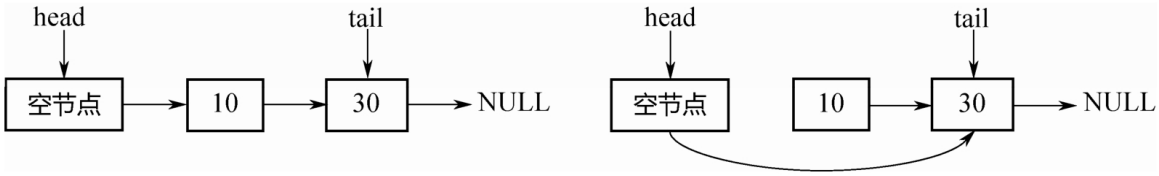
前面讲的无锁队列、栈，都是只在队头、队尾进行CAS操作，通常不会有问题。如果在链表的中间进行插入或删除操作，按照通常的CAS做法，就会出现问题!

关于这个问题，Doug Lea的论文中有清晰的论述，此处引用如下：

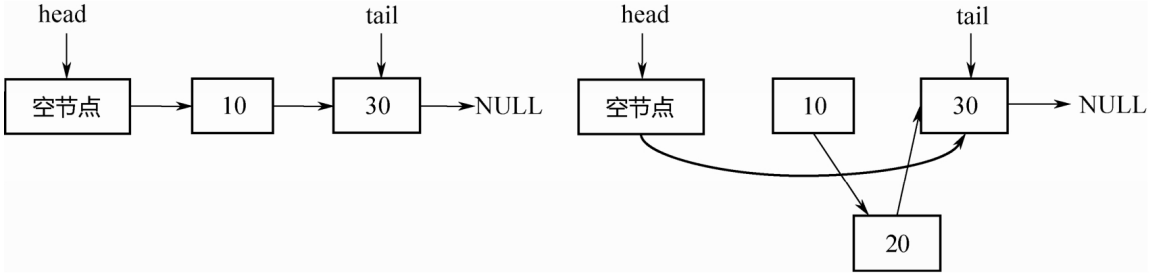
操作1：在节点10后面插入节点20。如下图所示，首先把节点20的next指针指向节点30，然后对节点10的next指针执行CAS操作，使其指向节点20即可。




操作2：删除节点10。如下图所示，只需把头节点的next指针，进行CAS操作到节点30即可。




但是，如果两个线程同时操作，一个删除节点10，一个要在节点10后面插入节点20。并且这两个操作都各自是CAS的，此时就会出现问题。如下图所示，删除节点10，会同时把新插入的节点20也删除掉！这个问题超出了CAS的解决范围。




为什么会出现这个问题呢?

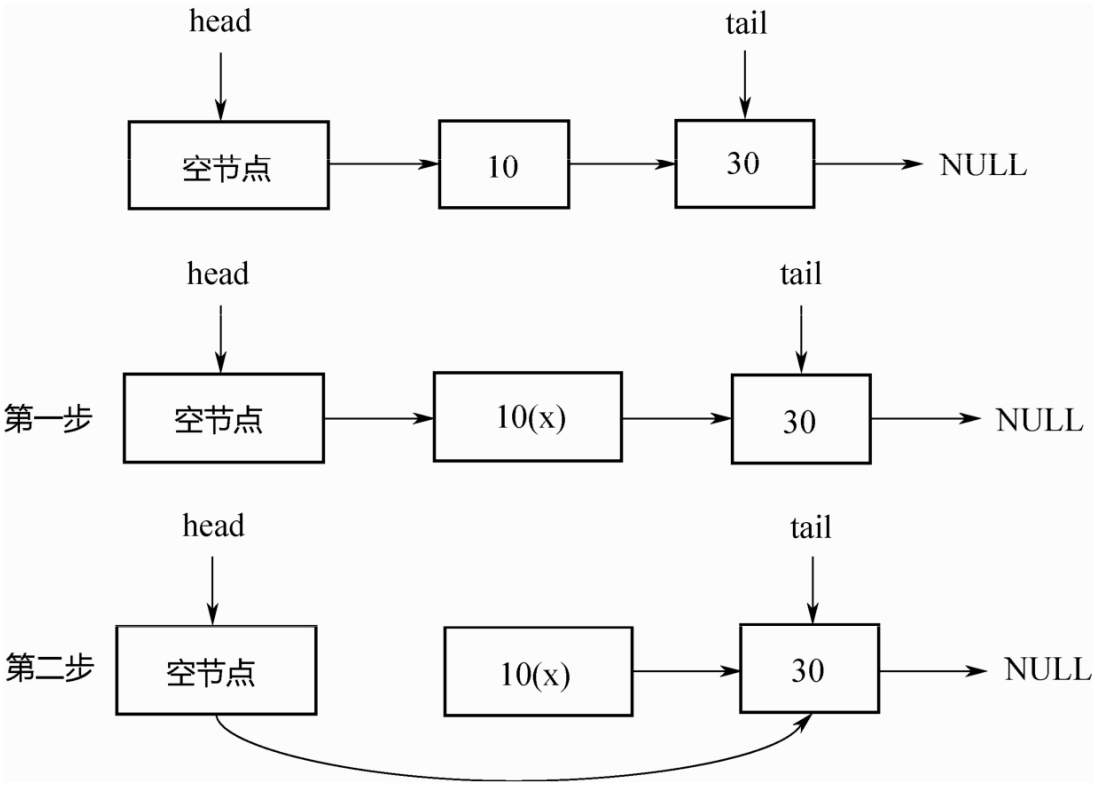
究其原因：在删除节点10的时候，实际受到操作的是节点10的前驱，也就是头节点。节点10本身没有任何变化。故而，再往节点10后插入节点20的线程，并不知道节点10已经被删除了！

针对这个问题，在论文中提出了如下的解决办法，如下图所示，把节点 10 的删除分为两2步：

第一步，把节点10的next指针，mark成删除，即软删除；

第二步，找机会，物理删除。

做标记之后，当线程再往节点10后面插入节点20的时候，便可以先进行判断，节点10是否已经被删除，从而避免在一个删除的节点10后面插入节点20。**这个解决方法有一个关键点："把节点10的next指针指向节点20（插入操作）"和"判断节点10本身是否已经删除（判断操作）"，必须是原子的，必须在1个CAS操作里面完成！**



具体的实现有两个办法：

办法一：AtomicMarkableReference

保证每个 next 是 AtomicMarkableReference 类型。但这个办法不够高效，Doug Lea 在ConcurrentSkipListMap的实现中用了另一种办法。

办法2：Mark节点

我们的目的是标记节点10已经删除，也就是标记它的next字段。那么可以新造一个marker节点，使节点10的next指针指向该Marker节点。这样，当向节点10的后面插入节点20的时候，就可以在插入的同时判断节点10的next指针是否指向了一个Marker节点，这两个操作可以在一个CAS操作里面完成。

**3.跳查表**

解决了无锁链表的插入或删除问题，也就解决了跳查表的一个关键问题。因为跳查表就是多层链表叠起来的。

下面先看一下跳查表的数据结构（下面所用代码都引用自JDK 7，JDK 8中的代码略有差异，但不影响下面的原理分析）。

ConcurrentSkipListMap.java

```
359     static final class Node<K,V> {
360         final K key; // currently, never detached
361         V val;
362         Node<K,V> next;
363         Node(K key, V value, Node<K,V> next) {
364             this.key = key;
365             this.val = value;
366             this.next = next;
367         }
368     }
```

上图中的Node就是跳查表底层节点类型。所有的<K, V>对都是由这个单向链表串起来的。

上面的Index层的节点：

ConcurrentSkipListMap.java

```
373     static final class Index<K,V> {
374         final Node<K,V> node;  // currently, never detached
375         final Index<K,V> down;
376         Index<K,V> right;
377         Index(Node<K,V> node, Index<K,V> down, Index<K,V> right) {
378             this.node = node;
379             this.down = down;
380             this.right = right;
381         }
382     }
```

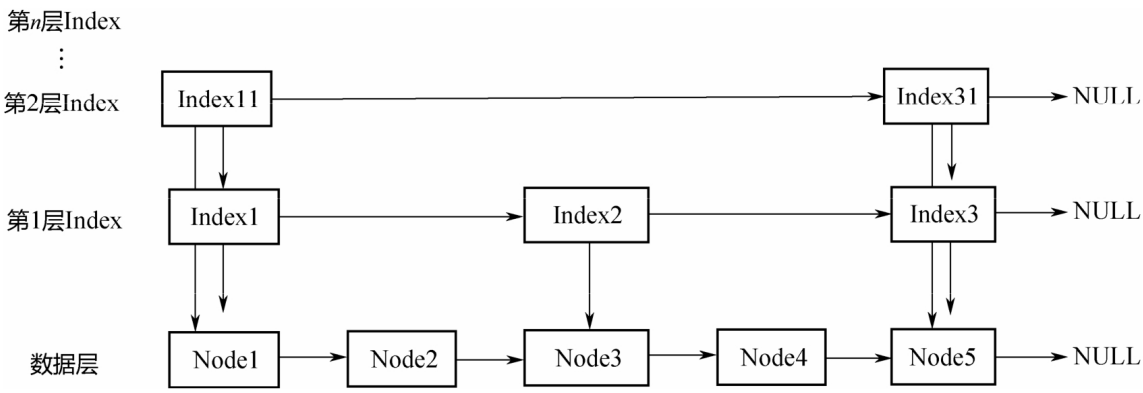
上图中的node属性不存储实际数据，指向Node节点。

down属性：每个Index节点，必须有一个指针，指向其下一个Level对应的节点。

right属性：Index也组成单向链表。

整个ConcurrentSkipListMap就只需要记录顶层的head节点即可：

```
public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable {
    // ...
    private transient Index<K,V> head;
    // ...
}
```




下面详细分析如何从跳查表上查找、插入和删除元素。

1. put实现分析

ConcurrentSkipListMap.java

```
1342    public V put(K key, V value) {
1343        if (value == null)
1344            throw new NullPointerException();
1345        return doPut(key, value, onlyIfAbsent: false);
1346    }
```

```
private V doPut(K key, V value, boolean onlyIfAbsent) {
    if (key == null)
        throw new NullPointerException();
    Comparator<? super K> cmp = comparator;
    for (;;) {
        Index<K,V> h; Node<K,V> b;
        VarHandle.acquireFence();
        int levels = 0;                    // number of levels descended
        if ((h = head) == null) {          // 初始化
            Node<K,V> base = new Node<K,V>(null, null, null);
            h = new Index<K,V>(base, null, null);
            b = (HEAD.compareAndSet(this, null, h)) ? base : null;
        }
        else {
            for (Index<K,V> q = h, r, d;;) { // count while descending
```

```
                while ((r = q.right) != null) {
                    Node<K,V> p; K k;
                    if ((p = r.node) == null || (k = p.key) == null ||
                        p.val == null)
                        RIGHT.compareAndSet(q, r, r.right);
                    else if (cpr(cmp, key, k) > 0)
                        q = r;
                    else
                        break;
                }
                if ((d = q.down) != null) {
                    ++levels;
                    q = d;
                }
                else {
                    b = q.node;
                    break;
                }
            }
        }
        if (b != null) {
            Node<K,V> z = null;              // new node, if inserted
            for (;;) {                       // find insertion point
                Node<K,V> n, p; K k; V v; int c;
                if ((n = b.next) == null) {
                    if (b.key == null)       // if empty, type check key now
                        cpr(cmp, key, key);
                    c = -1;
                }
                else if ((k = n.key) == null)
                    break;                   // can't append; restart
                else if ((v = n.val) == null) {
                    unlinkNode(b, n);
                    c = 1;
                }
                else if ((c = cpr(cmp, key, k)) > 0)
                    b = n;
                else if (c == 0 &&
                         (onlyIfAbsent || VAL.compareAndSet(n, v, value)))
                    return v;
                if (c < 0 &&
                    NEXT.compareAndSet(b, n,
                                       p = new Node<K,V>(key, value, n))) {
                    z = p;
                    break;
                }
            }
            if (z != null) {
                int lr = ThreadLocalRandom.nextSecondarySeed();
                if ((lr & 0x3) == 0) {       // add indices with 1/4 prob
                    int hr = ThreadLocalRandom.nextSecondarySeed();
                    long rnd = ((long)hr << 32) | ((long)lr & 0xffffffffL);
                    int skips = levels;      // levels to descend before add
                    Index<K,V> x = null;
                    for (;;) {               // create at most 62 indices
                        x = new Index<K,V>(z, x, null);
                        if (rnd >= 0L || --skips < 0)
                            break;
```

```
                        else
                            rnd <<= 1;
                    }
                    if (addIndices(h, skips, x, cmp) && skips < 0 &&
                        head == h) {         // try to add new level
                        Index<K,V> hx = new Index<K,V>(z, x, null);
                        Index<K,V> nh = new Index<K,V>(h.node, h, hx);
                        HEAD.compareAndSet(this, h, nh);
                    }
                    if (z.val == null)       // deleted while adding indices
                        findPredecessor(key, cmp); // clean
                }
                addCount(1L);
                return null;
            }
        }
    }
}
```

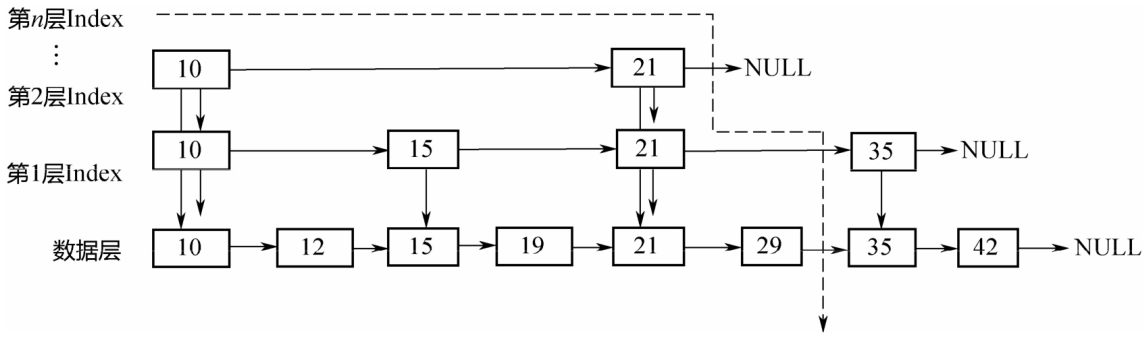
在底层，节点按照从小到大的顺序排列，上面的index层间隔地串在一起，因为从小到大排列。查找的时候，从顶层index开始，自左往右、自上往下，形成图示的遍历曲线。假设要查找的元素是32，遍历过程如下：

先遍历第2层Index，发现在21的后面；

从21下降到第1层Index，从21往后遍历，发现在21和35之间；

从21下降到底层，从21往后遍历，最终发现在29和35之间。

在整个的查找过程中，范围不断缩小，最终定位到底层的两个元素之间。




关于上面的put(...)方法，有一个关键点需要说明：在通过findPredecessor找到了待插入的元素在[b，n]之间之后，并不能马上插入。因为其他线程也在操作这个链表，b、n都有可能被删除，所以在插入之前执行了一系列的检查逻辑，而这也正是无锁链表的复杂之处。

2. remove(...)分析

ConcurrentSkipListMap.java

```
1358    public V remove(Object key) {
1359        return doRemove(key, null);
1360    }
```

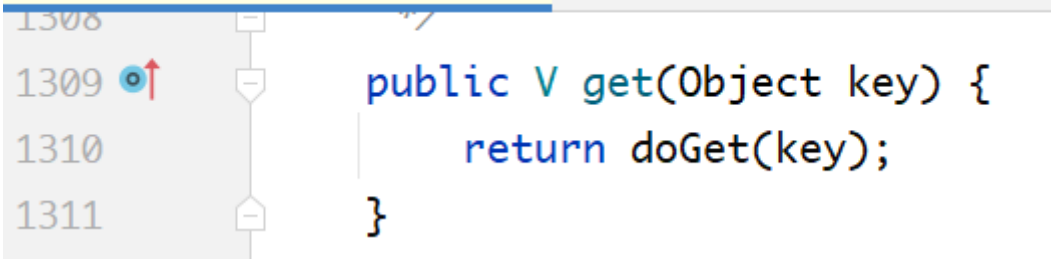
```java
// 若找到了(key, value)就删除，并返回value；找不到就返回null
final V doRemove(Object key, Object value) {
    if (key == null)
        throw new NullPointerException();
    Comparator<? super K> cmp = comparator;
    V result = null;
    Node<K,V> b;
    outer: while ((b = findPredecessor(key, cmp)) != null &&
                  result == null) {
        for (;;) {
            Node<K,V> n; K k; V v; int c;
            if ((n = b.next) == null)
                break outer;
            else if ((k = n.key) == null)
                break;
            else if ((v = n.val) == null)
                unlinkNode(b, n);
            else if ((c = cpr(cmp, key, k)) > 0)
                b = n;
            else if (c < 0)
                break outer;
            else if (value != null && !value.equals(v))
                break outer;
            else if (VAL.compareAndSet(n, v, null)) {
                result = v;
                unlinkNode(b, n);
                break; // loop to clean up
            }
        }
    }
    if (result != null) {
        tryReduceLevel();
        addCount(-1L);
    }
    return result;
}
```

上面的删除方法和插入方法的逻辑非常类似，因为无论是插入，还是删除，都要先找到元素的前驱，也就是定位到元素所在的区间[b，n]。在定位之后，执行下面几个步骤：

1. 如果发现b、n已经被删除了，则执行对应的删除清理逻辑；
2. 否则，如果没有找到待删除的(k, v)，返回null；
3. 如果找到了待删除的元素，也就是节点n，则把n的value置为null，同时在n的后面加上Marker节点，同时检查是否需要降低Index的层次。

3. get分析

ConcurrentSkipListMap.java




```
private V doGet(Object key) {
    Index<K,V> q;
    VarHandle.acquireFence();
    if (key == null)
        throw new NullPointerException();
    Comparator<? super K> cmp = comparator;
    V result = null;
    if ((q = head) != null) {
        outer: for (Index<K,V> r, d;;) {
            while ((r = q.right) != null) {
                Node<K,V> p; K k; V v; int c;
                if ((p = r.node) == null || (k = p.key) == null ||
                    (v = p.val) == null)
                    RIGHT.compareAndSet(q, r, r.right);
                else if ((c = cpr(cmp, key, k)) > 0)
                    q = r;
                else if (c == 0) {
                    result = v;
                    break outer;
                }
                else
                    break;
            }
            if ((d = q.down) != null)
                q = d;
            else {
                Node<K,V> b, n;
                if ((b = q.node) != null) {
                    while ((n = b.next) != null) {
                        V v; int c;
                        K k = n.key;
                        if ((v = n.val) == null || k == null ||
                            (c = cpr(cmp, key, k)) > 0)
                            b = n;
                        else {
                            if (c == 0)
                                result = v;
                            break;
                        }
                    }
                }
                break;
            }
        }
    }
    return result;
```

```
47 }
```

无论是插入、删除，还是查找，都有相似的逻辑，都需要先定位到元素位置[b，n]，然后判断b、n是否已经被删除，如果是，则需要执行相应的删除清理逻辑。这也正是无锁链表复杂的地方。

#### 5.6.2 ConcurrentSkipListSet

如下面代码所示，ConcurrentSkipListSet只是对ConcurrentSkipListMap的简单封装，此处不再进一步展开叙述。

```
1  public class ConcurrentSkipListSet<E>
2      extends AbstractSet<E>
3      implements NavigableSet<E>, Cloneable, java.io.Serializable {
4      // 封装了一个ConcurrentSkipListMap
5      private final ConcurrentNavigableMap<E,Object> m;
6
7      public ConcurrentSkipListSet() {
8          m = new ConcurrentSkipListMap<E,Object>();
9      }
10
11     public boolean add(E e) {
12         return m.putIfAbsent(e, Boolean.TRUE) == null;
13     }
14     // ...
15 }
```

# 6 同步工具类

## 6.1 Semaphore

Semaphore也就是信号量，提供了资源数量的并发访问控制，其使用代码很简单，如下所示：

```
1  // 一开始有5份共享资源。第二个参数表示是否是公平
2  Semaphore myResources = new Semaphore(5, true);
3
4  // 工作线程每获取一份资源，就在该对象上记下来
5  // 在获取的时候是按照公平的方式还是非公平的方式，就要看上一行代码的第二个参数了。
6  // 一般非公平抢占效率较高。
7  myResources.acquire();
8
9  // 工作线程每归还一份资源，就在该对象上记下来
10 // 此时资源可以被其他线程使用
11 myResources.release();
12
13 /*
14 释放指定数目的许可，并将它们归还给信标。
15 可用许可数加上该指定数目。
16 如果线程需要获取N个许可，在有N个许可可用之前，该线程阻塞。
```

```
如果线程获取了N个许可，还有可用的许可，则依次将这些许可赋予等待获取许可的其他线程。
*/
semaphore.release(2);

/*
从信标获取指定数目的许可。如果可用许可数目不够，则线程阻塞，直到被中断。

该方法效果与循环相同，
for (int i = 0; i < permits; i++) acquire();
只不过该方法是原子操作。

如果可用许可数不够，则当前线程阻塞，直到：（二选一）
1. 如果其他线程释放了许可，并且可用的许可数满足当前线程的请求数字；
2. 其他线程中断了当前线程。

permits - 要获取的许可数
*/
semaphore.acquire(3);
```

案例：

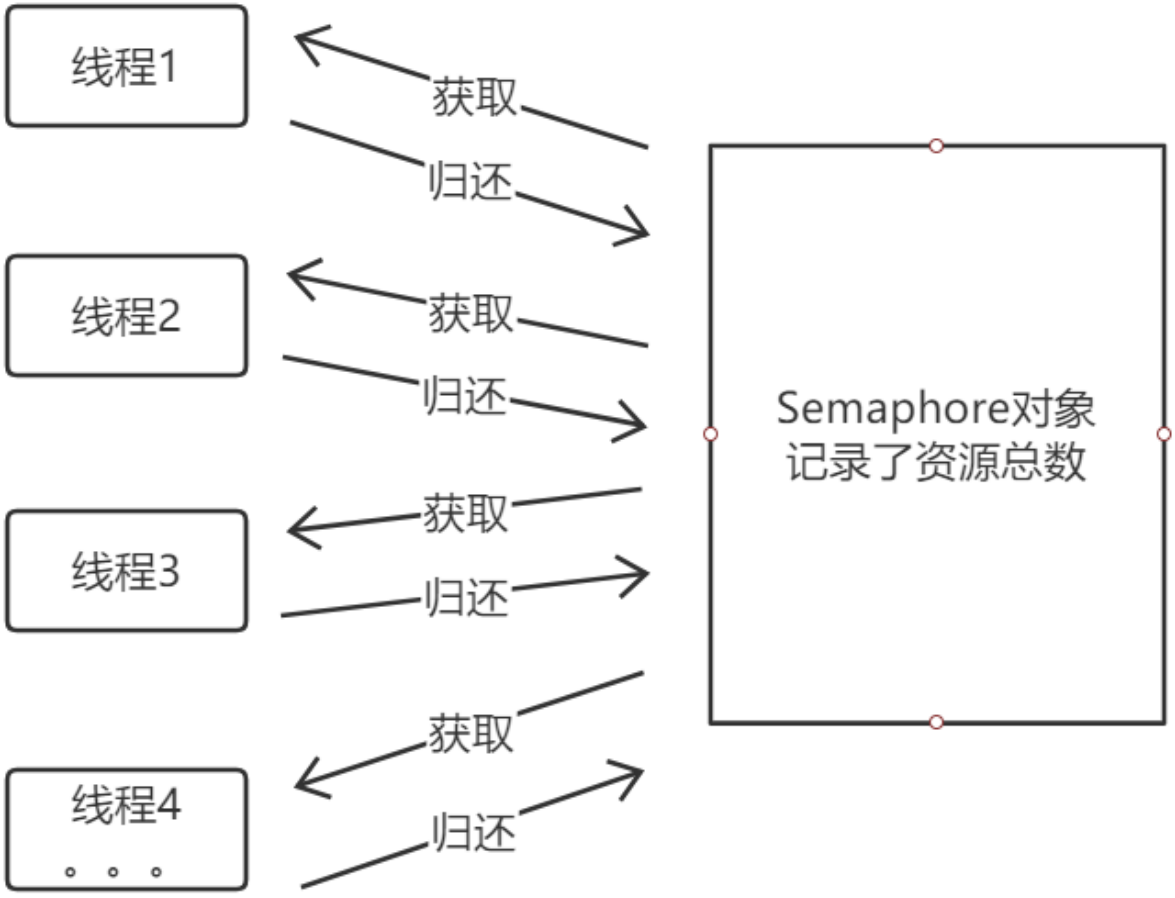
大学生到自习室抢座，写作业：

```
package com.lagou.concurrent.demo;

import java.util.Random;
import java.util.concurrent.Semaphore;

public class MyThread extends Thread {

    private final Semaphore semaphore;
    private final Random random = new Random();

    public MyThread(String name, Semaphore semaphore) {
        super(name);
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        try {
            semaphore.acquire();

            System.out.println(Thread.currentThread().getName() + " - 抢座成功，开始写作业");

            Thread.sleep(random.nextInt(1000));

            System.out.println(Thread.currentThread().getName() + " - 作业完成，腾出座位");

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        semaphore.release();
```

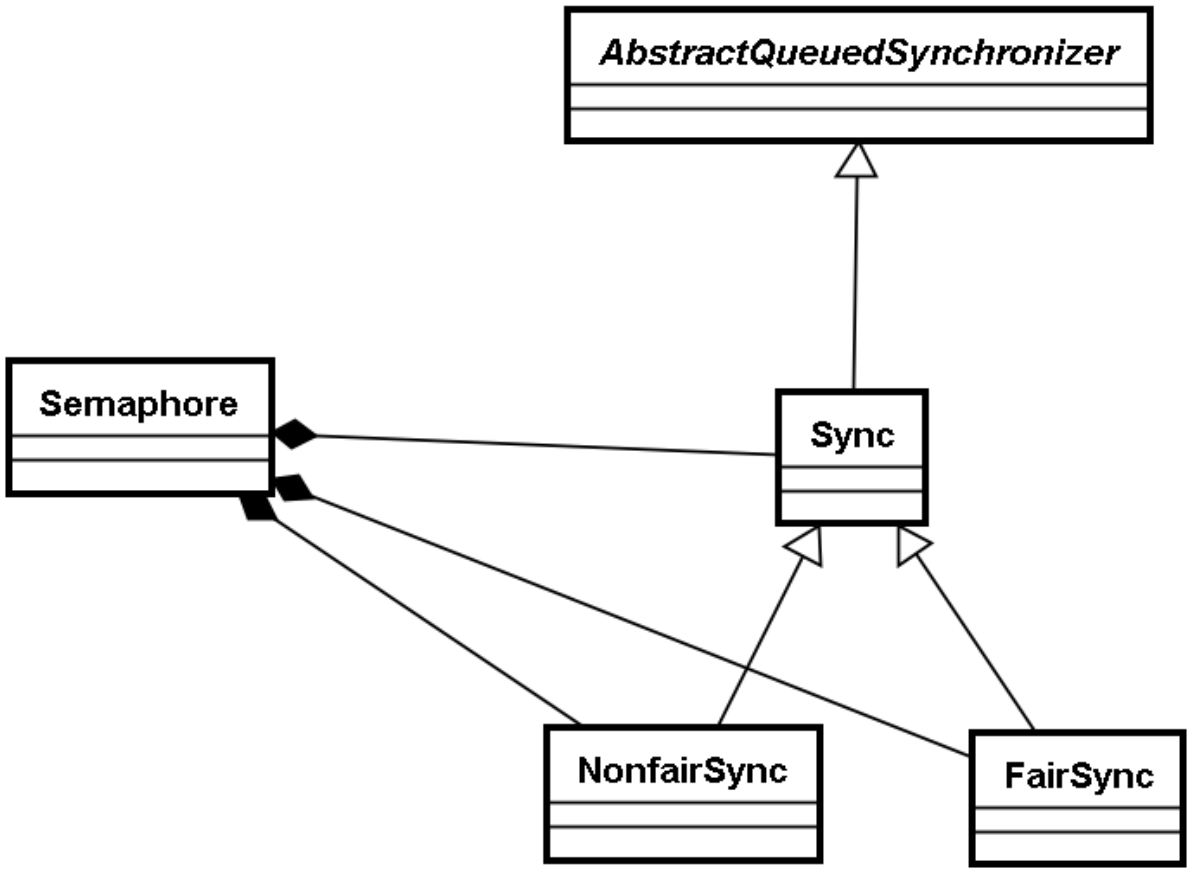
```
    }
}
```

```
package com.lagou.concurrent.demo;

import java.util.concurrent.Semaphore;

public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Semaphore semaphore = new Semaphore(2);

        for (int i = 0; i < 5; i++) {
            new MyThread("学生-" + (i + 1), semaphore).start();
        }
    }
}
```

如下图所示，假设有*n*个线程来获取Semaphore里面的10份资源（*n* > 10），*n*个线程中只有10个线程能获取到，其他线程都会阻塞。直到有线程释放了资源，其他线程才能获取到。




当初始的资源个数为1的时候，Semaphore退化为排他锁。正因为如此，Semaphone的实现原理和锁十分类似，是基于AQS，有公平和非公平之分。Semaphore相关类的继承体系如下图所示：




```
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}

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

由于Semaphore和锁的实现原理基本相同，上面的代码不再展开解释。资源总数即state的初始值，在acquire里对state变量进行CAS减操作，减到0之后，线程阻塞；在release里对state变量进行CAS加操作。

```
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer implements java.io.Serializable {

    // ...
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }

    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
```

```
        }
        return false;
    }
    // ...
}

public class Semaphore {

       protected final boolean tryReleaseShared(int releases) {
           for (;;) {
               int current = getState();
               int next = current + releases;
               if (next < current) // overflow
                   throw new Error("Maximum permit count exceeded");
               if (compareAndSetState(current, next))
                   return true;
           }
       }

    static final class FairSync extends Sync {

        // ...

        FairSync(int permits) {
            super(permits);
        }

        protected int tryAcquireShared(int acquires) {
            for (;;) {
                if (hasQueuedPredecessors())
                    return -1;
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 || compareAndSetState(available,
remaining))
                    return remaining;
            }
        }
    }
}
```

```
package java.lang.invoke;

public abstract class VarHandle {
    // ...
    // CAS，原子操作
    public final native
    @MethodHandle.PolymorphicSignature
    @HotSpotIntrinsicCandidate
    boolean compareAndSet(Object... args);
    // ...
}
```

## 6.2 CountDownLatch

### 6.2.1 CountDownLatch使用场景

假设一个主线程要等待5个 Worker 线程执行完才能退出，可以使用CountDownLatch来实现：

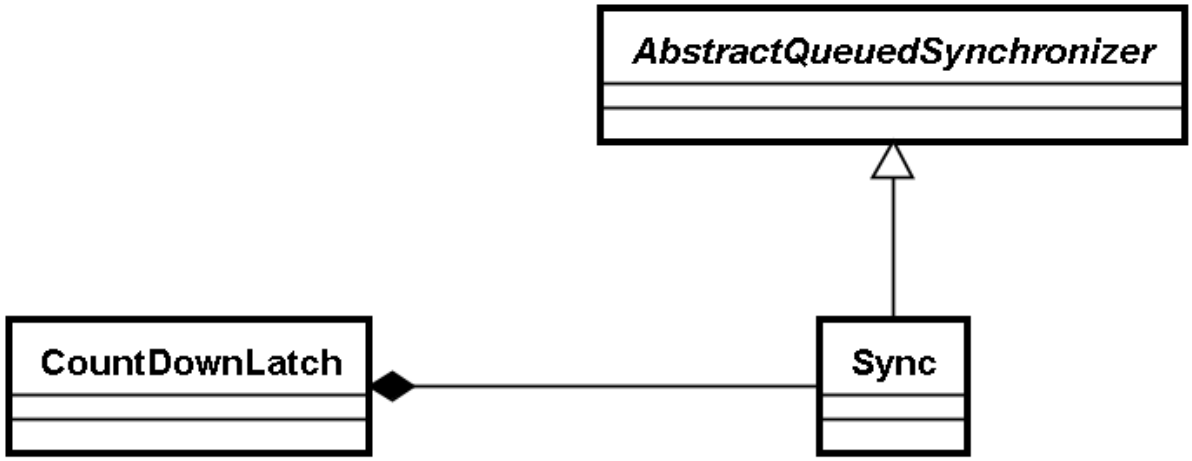
线程：

```
package com.lagou.concurrent.demo;

import java.util.Random;
import java.util.concurrent.CountDownLatch;

public class MyThread extends Thread {

    private final CountDownLatch latch;
    private final Random random = new Random();

    public MyThread(String name, CountDownLatch latch) {
        super(name);
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(random.nextInt(2000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + "运行结束");
        latch.countDown();
    }
}
```

Main类：

```
package com.lagou.concurrent.demo;

import java.util.concurrent.CountDownLatch;

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

        CountDownLatch latch = new CountDownLatch(5);

        new MyThread("线程1", latch).start();
        new MyThread("线程2", latch).start();
        new MyThread("线程3", latch).start();
        new MyThread("线程4", latch).start();
//        new MyThread("线程5", latch).start();

        // 当前线程等待
        latch.await();
        System.out.println("程序运行结束");
    }
}
```

下图为CountDownLatch相关类的继承层次，CountDownLatch原理和Semaphore原理类似，同样是基于AQS，不过没有公平和非公平之分。




### 6.2.2 await()实现分析

如下所示，await()调用的是AQS 的模板方法，这个方法在前面已经介绍过。CountDownLatch.Sync重新实现了tryAccuqireShared方法：

```
public void await() throws InterruptedException {
    // AQS的模板方法
    sync.acquireSharedInterruptibly(1);
}

public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // 被CountDownLatch.Sync实现
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}

    protected int tryAcquireShared(int acquires) {
        return (getState() == 0) ? 1 : -1;
    }
```

从tryAcquireShared(...)方法的实现来看，只要state != 0，调用await()方法的线程便会被放入AQS的阻塞队列，进入阻塞状态。

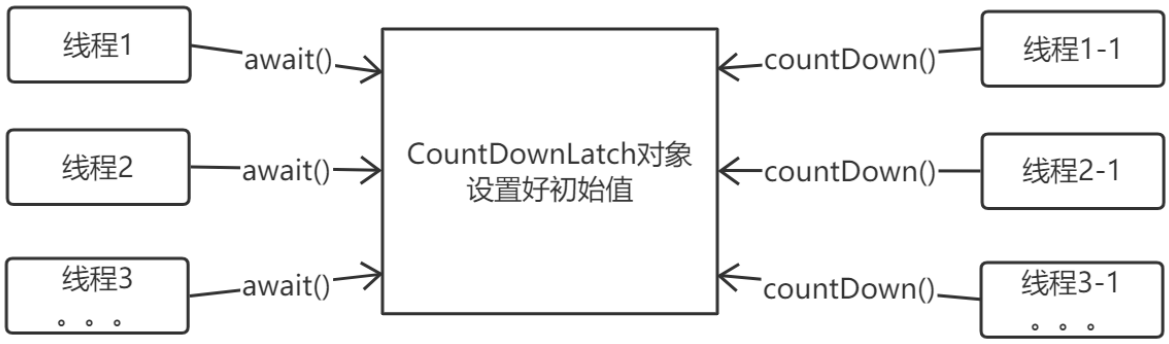
### 6.2.3 countDown()实现分析

```
public void countDown() {
    sync.releaseShared(1);
}

// AQS的模板方法
public final boolean releaseShared(int arg) {
```

```
    // 由CountDownLatch.Sync实现
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}

    protected boolean tryReleaseShared(int releases) {
        // Decrement count; signal when transition to zero
        for (;;) {
            int c = getState();
            if (c == 0)
                return false;
            int nextc = c - 1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
```

countDown()调用的AQS的模板方法releaseShared()，里面的tryReleaseShared(...)由CountDownLatch.Sync实现。从上面的代码可以看出，只有state=0，tryReleaseShared(...)才会返回true，然后执行doReleaseShared(...)，一次性唤醒队列中所有阻塞的线程。

总结：由于是基于AQS阻塞队列来实现的，所以可以让多个线程都阻塞在state=0条件上，通过countDown()一直减state，减到0后一次性唤醒所有线程。如下图所示，假设初始总数为*M*，*N*个线程await()，*M*个线程countDown()，减到0之后，*N*个线程被唤醒。




## 6.3 CyclicBarrier

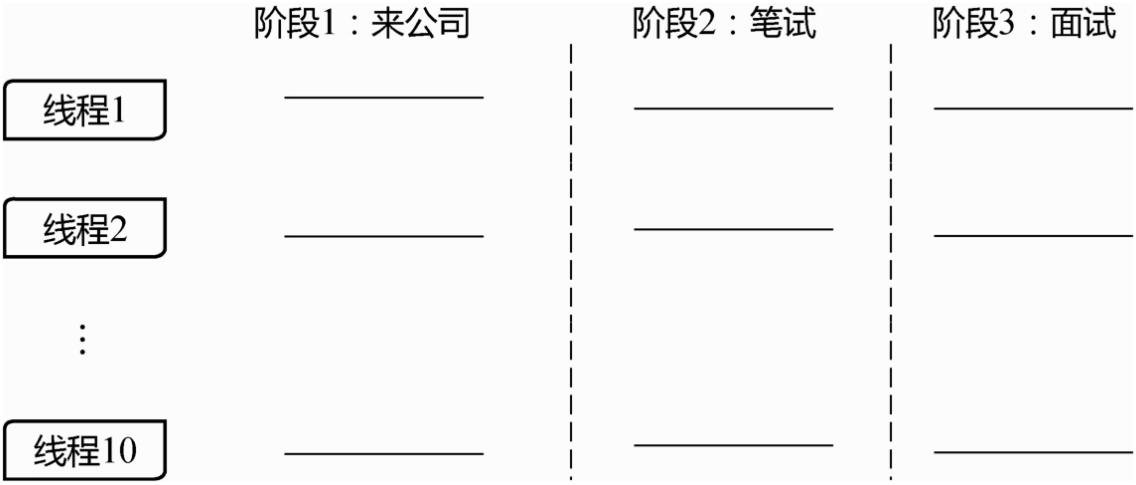
### 6.3.1 CyclicBarrier使用场景

CyclicBarrier使用方式比较简单：

```
CyclicBarrier barrier = new CyclicBarrier(5);
barrier.await();
```

该类用于协调多个线程同步执行操作的场合。

使用场景：10个工程师一起来公司应聘，招聘方式分为笔试和面试。首先，要等人到齐后，开始笔试；笔试结束之后，再一起参加面试。把10个人看作10个线程，10个线程之间的同步过程如下图所示：




Main类：

```
package com.lagou.concurrent.cyclicbarrier;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Main {
    public static void main(String[] args) throws BrokenBarrierException,
InterruptedException {
        CyclicBarrier barrier = new CyclicBarrier(5);

        for (int i = 0; i < 5; i++) {
            new MyThread("线程-" + (i + 1), barrier).start();
        }

    }
}
```

MyThread类：

```
package com.lagou.concurrent.cyclicbarrier;

import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyThread extends Thread {

    private final CyclicBarrier barrier;
    private final Random random = new Random();

    public MyThread(String name, CyclicBarrier barrier) {
        super(name);
        this.barrier = barrier;
    }

    @Override
    public void run() {
```

```
        try {
            Thread.sleep(random.nextInt(2000));
            System.out.println(Thread.currentThread().getName() + " - 已经到达公司");
            barrier.await();

            Thread.sleep(random.nextInt(2000));
            System.out.println(Thread.currentThread().getName() + " - 已经笔试结束");
            barrier.await();

            Thread.sleep(random.nextInt(2000));
            System.out.println(Thread.currentThread().getName() + " - 已经面试结束");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
        super.run();
    }
}
```

在整个过程中，有2个同步点：第1个同步点，要等所有应聘者都到达公司，再一起开始笔试；第2个同步点，要等所有应聘者都结束笔试，之后一起进入面试环节。

### 6.3.2 CyclicBarrier实现原理

CyclicBarrier基于ReentrantLock+Condition实现。

```
public class CyclicBarrier {
    private final ReentrantLock lock = new ReentrantLock();
    // 用于线程之间相互唤醒
    private final Condition trip = lock.newCondition();
    // 线程总数
    private final int parties;
    private int count;
    private Generation generation = new Generation();
    // ...
}
```

下面详细介绍 CyclicBarrier 的实现原理。先看构造方法：

```
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    // 参与方数量
    this.parties = parties;
    this.count = parties;
    // 当所有线程被唤醒时，执行barrierCommand表示的Runnable。
    this.barrierCommand = barrierAction;
}
```

接下来看一下await()方法的实现过程。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}

private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        final Generation g = generation;

        if (g.broken)
            throw new BrokenBarrierException();
        // 响应中断
        if (Thread.interrupted()) {
            // 唤醒所有阻塞的线程
            breakBarrier();
            throw new InterruptedException();
        }

        // 每个线程调用一次await()，count都要减1
        int index = --count;
        // 当count减到0的时候，此线程唤醒其他所有线程
        if (index == 0) {  // tripped
            boolean ranAction = false;
            try {
                final Runnable command = barrierCommand;
                if (command != null)
                    command.run();
                ranAction = true;
                nextGeneration();
                return 0;
            } finally {
                if (!ranAction)
                    breakBarrier();
            }
        }

        // loop until tripped, broken, interrupted, or timed out
```

```
        for (;;) {
            try {
                if (!timed)
                    trip.await();
                else if (nanos > 0L)
                    nanos = trip.awaitNanos(nanos);
            } catch (InterruptedException ie) {
                if (g == generation && ! g.broken) {
                    breakBarrier();
                    throw ie;
                } else {
                    // We're about to finish waiting even if we had not
                    // been interrupted, so this interrupt is deemed to
                    // "belong" to subsequent execution.
                    Thread.currentThread().interrupt();
                }
            }

            if (g.broken)
                throw new BrokenBarrierException();

            if (g != generation)
                return index;

            if (timed && nanos <= 0L) {
                breakBarrier();
                throw new TimeoutException();
            }
        }
    } finally {
        lock.unlock();
    }
}

private void breakBarrier() {
    generation.broken = true;
    count = parties;
    trip.signalAll();
}

private void nextGeneration() {
    // signal completion of last generation
    trip.signalAll();
    // set up next generation
    count = parties;
    generation = new Generation();
}
```

关于上面的方法，有几点说明：

1. CyclicBarrier是可以被重用的。以上一节的应聘场景为例，来了10个线程，这10个线程互相等待，到齐后一起被唤醒，各自执行接下来的逻辑；然后，这10个线程继续互相等待，到齐后再一起被唤醒。每一轮被称为一个Generation，就是一次同步点。
2. CyclicBarrier 会响应中断。10 个线程没有到齐，如果有线程收到了中断信号，所有阻塞的线程也会被唤醒，就是上面的breakBarrier()方法。然后count被重置为初始值（parties），重

新开始。

3. 上面的回调方法，barrierAction只会被第10个线程执行1次（在唤醒其他9个线程之前），而不是10个线程每个都执行1次。

## 6.4 Exchanger

### 6.4.1 使用场景

Exchanger用于线程之间交换数据，其使用代码很简单，是一个exchange(...)方法，使用示例如下：

```
package com.lagou.concurrent.demo;

import java.util.Random;
import java.util.concurrent.Exchanger;

public class Main {
    private static final Random random = new Random();
    public static void main(String[] args) {
        // 建一个多线程共用的exchange对象
        // 把exchange对象传给3个线程对象。每个线程在自己的run方法中调用exchange，把自己的数据作为参数
        // 传递进去，返回值是另外一个线程调用exchange传进去的参数
        Exchanger<String> exchanger = new Exchanger<>();

        new Thread("线程1") {
            @Override
            public void run() {
                while (true) {
                    try {
                        // 如果没有其他线程调用exchange，线程阻塞，直到有其他线程调用exchange为止。
                        String otherData = exchanger.exchange("交换数据1");
                        System.out.println(Thread.currentThread().getName() + "得到<==" + otherData);
                        Thread.sleep(random.nextInt(2000));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();

        new Thread("线程2") {
            @Override
            public void run() {
                while (true) {
                    try {
                        String otherData = exchanger.exchange("交换数据2");
                        System.out.println(Thread.currentThread().getName() + "得到<==" + otherData);
                        Thread.sleep(random.nextInt(2000));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
```

```
                    }
                }
            }
        }.start();

        new Thread("线程3") {
            @Override
            public void run() {
                while (true) {
                    try {
                        String otherData = exchanger.exchange("交换数据3");
                        System.out.println(Thread.currentThread().getName()
+ "得到<==" + otherData);
                        Thread.sleep(random.nextInt(2000));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }
}
```

在上面的例子中，3个线程并发地调用exchange(...)，会两两交互数据，如1/2、1/3和2/3。

### 6.4.2 实现原理

Exchanger的核心机制和Lock一样，也是CAS+park/unpark。

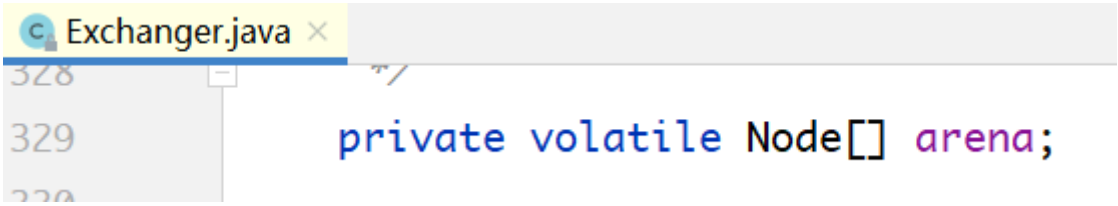
首先，在Exchanger内部，有两个内部类：Participant和Node，代码如下：

```
public class Exchanger<V> {
    // ...
    // 添加了Contended注解，表示伪共享与缓存行填充
    @jdk.internal.vm.annotation.Contended static final class Node {
        int index;              // Arena index
        int bound;              // Last recorded value of Exchanger.bound
        int collides;           // 本次绑定中，CAS操作失败次数
        int hash;               // 自旋伪随机
        Object item;            // 本线程要交换的数据
        volatile Object match;  // 对方线程交换来的数据
        // 当前线程
        volatile Thread parked; // 当前线程阻塞的时候设置该属性，不阻塞为null。
    }

    static final class Participant extends ThreadLocal<Node> {
        public Node initialValue() { return new Node(); }
    }
    // ...
}
```

每个线程在调用exchange(...)方法交换数据的时候，会先创建一个Node对象。

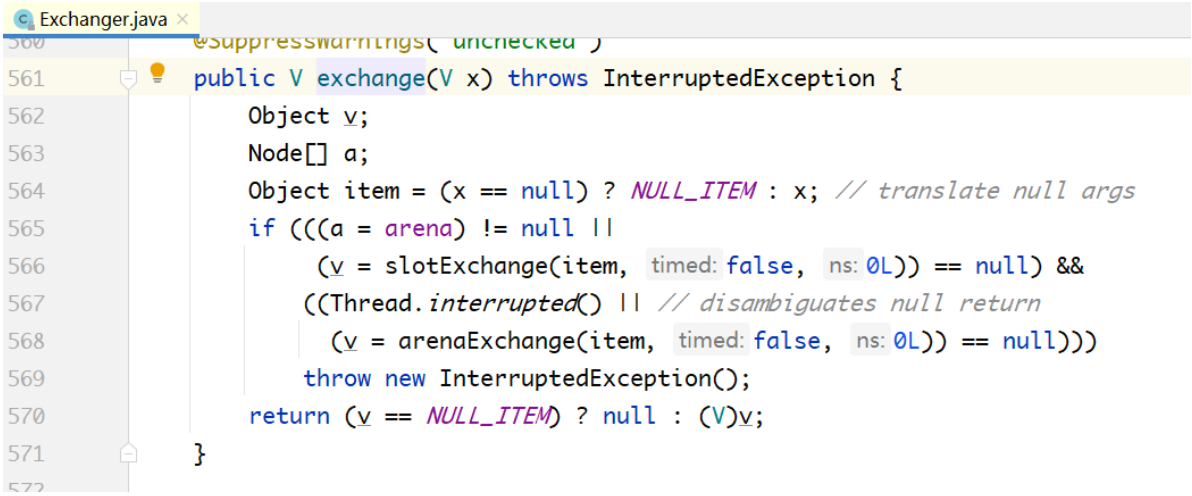
这个Node对象就是对该线程的包装，里面包含了3个重要字段：第一个是该线程要交互的数据，第二个是对方线程交换来的数据，最后一个是该线程自身。

一个Node只能支持2个线程之间交换数据，要实现多个线程并行地交换数据，需要多个Node，因此在Exchanger里面定义了Node数组：




### 6.4.3 exchange(V x)实现分析

明白了大致思路，下面来看exchange(V x)方法的详细实现：




上面方法中，如果arena不是null，表示启用了arena方式交换数据。如果arena不是null，并且线程被中断，则抛异常

如果arena不是null，并且arenaExchange的返回值为null，则抛异常。对方线程交换来的null值是封装为NULL_ITEM对象的，而不是null。

如果slotExchange的返回值是null，并且线程被中断，则抛异常。

如果slotExchange的返回值是null，并且areaExchange的返回值是null，则抛异常。

slotExchange的实现：

```
1  package java.util.concurrent;
2
3  public class Exchanger<V> {
4      // ...
5      /**
6       * 如果不启用arenas，则使用该方法进行线程间数据交换。
7       *
8       * @param item 需要交换的数据
9       * @param timed 是否是计时等待，true表示是计时等待
10      * @param ns 如果是计时等待，该值表示最大等待的时长。
11      * @return 对方线程交换来的数据；如果等待超时或线程中断，或者启用了arena，则返回
    null。
12      */
```

```
    private final Object slotExchange(Object item, boolean timed, long ns)
{
        // participant在初始化的时候设置初始值为new Node()
        // 获取本线程要交换的数据节点
        Node p = participant.get();
        // 获取当前线程
        Thread t = Thread.currentThread();
        // 如果线程被中断，则返回null。
        if (t.isInterrupted())
            return null;

        for (Node q;;) {
            // 如果slot非空，表明有其他线程在等待该线程交换数据
            if ((q = slot) != null) {
                // CAS操作，将当前线程的slot由slot设置为null
                // 如果操作成功，则执行if中的语句
                if (SLOT.compareAndSet(this, q, null)) {
                    // 获取对方线程交换来的数据
                    Object v = q.item;
                    // 设置要交换的数据
                    q.match = item;
                    // 获取q中阻塞的线程对象
                    Thread w = q.parked;
                    if (w != null)
                        // 如果对方阻塞的线程非空，则唤醒阻塞的线程
                        LockSupport.unpark(w);
                    return v;
                }
                // create arena on contention, but continue until slot null
                // 创建arena用于处理多个线程需要交换数据的场合，防止slot冲突
                if (NCPU > 1 && bound == 0 &&
                    BOUND.compareAndSet(this, 0, SEQ)) {
                        arena = new Node[(FULL + 2) << ASHIFT];
                    }
            }
            // 如果arena不是null，需要调用者调用arenaExchange方法接着获取对方线程交
换来的数据
            else if (arena != null)
                return null;
            else {
                // 如果slot为null，表示对方没有线程等待该线程交换数据
                // 设置要交换的本方数据
                p.item = item;
                // 设置当前线程要交换的数据到slot
                // CAS操作，如果设置失败，则进入下一轮for循环
                if (SLOT.compareAndSet(this, null, p))
                    break;
                p.item = null;
            }
        }

        // 没有对方线程等待交换数据，将当前线程要交换的数据放到slot中，是一个Node对象
        // 然后阻塞，等待唤醒
        int h = p.hash;
        // 如果是计时等待交换，则计算超时时间；否则设置为0。
        long end = timed ? System.nanoTime() + ns : 0L;
        // 如果CPU核心数大于1，则使用SPINS数，自旋；否则为1，没必要自旋。
        int spins = (NCPU > 1) ? SPINS : 1;
```

```
69
70          // 记录对方线程交换来的数据
71          Object v;
72          // 如果p.match==null，表示还没有线程交换来数据
73          while ((v = p.match) == null) {
74              // 如果自旋次数大于0，计算hash随机数
75              if (spins > 0) {
76                  // 生成随机数，用于自旋次数控制
77                  h ^= h << 1; h ^= h >>> 3; h ^= h << 10;
78                  if (h == 0)
79                      h = SPINS | (int)t.getId();
80                  else if (h < 0 && (--spins & ((SPINS >>> 1) - 1)) == 0)
81                      Thread.yield();
82              // p是ThreadLocal记录的当前线程的Node。
83              // 如果slot不是p表示slot是别的线程放进去的
84              } else if (slot != p) {
85                  spins = SPINS;
86              } else if (!t.isInterrupted() && arena == null &&
87                         (!timed || (ns = end - System.nanoTime()) > 0L)) {
88                  p.parked = t;
89                  if (slot == p) {
90                      if (ns == 0L)
91                          // 阻塞当前线程
92                          LockSupport.park(this);
93                      else
94                          // 如果是计时等待，则阻塞当前线程指定时间
95                          LockSupport.parkNanos(this, ns);
96                  }
97                  p.parked = null;
98              } else if (SLOT.compareAndSet(this, p, null)) {
99                  // 没有被中断但是超时了，返回TIMED_OUT，否则返回null
100                 v = timed && ns <= 0L && !t.isInterrupted() ? TIMED_OUT :
null;
101                 break;
102             }
103         }
104         // match设置为null值  CAS
105         MATCH.setRelease(p, null);
106         p.item = null;
107         p.hash = h;
108         // 返回获取的对方线程交换来的数据
109         return v;
110     }
111     // ...
112 }
```

arenaExchange的实现：

```
1  package java.util.concurrent;
2
3  public class Exchanger<V> {
4      // ...
5      /**
6       * 当启用arenas的时候，使用该方法进行线程间的数据交换。
7       *
```

```
     * @param item 本线程要交换的非null数据。
     * @param timed 如果需要计时等待，则设置为true。
     * @param ns 表示计时等待的最大时长。
     * @return 对方线程交换来的数据。如果线程被中断，或者等待超时，则返回null。
     */
    private final Object arenaExchange(Object item, boolean timed, long ns)
{
        Node[] a = arena;
        int alen = a.length;
        Node p = participant.get();
        // 访问下标为i处的slot数据
        for (int i = p.index;;) {                      // access slot at i
            int b, m, c;
            int j = (i << ASHIFT) + ((1 << ASHIFT) - 1);
            if (j < 0 || j >= alen)
                j = alen - 1;

            // 取出arena数组的第j个Node元素
            Node q = (Node)AA.getAcquire(a, j);
            // 如果q不是null，则将数组的第j个元素由q设置为null
            if (q != null && AA.compareAndSet(a, j, q, null)) {
                // 获取对方线程交换来的数据
                Object v = q.item;                     // release
                // 设置本方线程交换的数据
                q.match = item;
                // 获取对方线程对象
                Thread w = q.parked;
                if (w != null)
                    // 如果对方线程非空，则唤醒对方线程
                    LockSupport.unpark(w);
                return v;
            }
            // 如果自旋次数没达到边界，且q为null
            else if (i <= (m = (b = bound) & MMASK) && q == null) {
                // 提供本方数据
                p.item = item;                         // offer
                // 将arena的第j个元素由null设置为p
                if (AA.compareAndSet(a, j, null, p)) {
                    long end = (timed && m == 0) ? System.nanoTime() + ns :
0L;
                    Thread t = Thread.currentThread(); // wait
                    // 自旋等待
                    for (int h = p.hash, spins = SPINS;;) {
                        // 获取对方交换来的数据
                        Object v = p.match;
                        // 如果对方交换来的数据非空
                        if (v != null) {
                            // 将p设置为null，CAS操作
                            MATCH.setRelease(p, null);
                            // 清空
                            p.item = null;             // clear for next
use
                            p.hash = h;
                            // 返回交换来的数据
                            return v;
                        }
                        // 产生随机数，用于限制自旋次数
                        else if (spins > 0) {
```

```
                            h ^= h << 1; h ^= h >>> 3; h ^= h << 10; // xorshift
                            if (h == 0)                // initialize hash
                                h = SPINS | (int)t.getId();
                            else if (h < 0 &&          // approx 50% true
                                     (--spins & ((SPINS >>> 1) - 1)) == 0)
                                Thread.yield();        // two yields per wait
                        }
                        // 如果arena的第j个元素不是p
                        else if (AA.getAcquire(a, j) != p)
                            spins = SPINS;       // releaser hasn't set match yet
                        else if (!t.isInterrupted() && m == 0 &&
                                 (!timed ||
                                  (ns = end - System.nanoTime()) > 0L)) {
                            p.parked = t;              // minimize window
                            if (AA.getAcquire(a, j) == p) {
                                if (ns == 0L)
                                    // 当前线程阻塞，等待交换数据
                                    LockSupport.park(this);
                                else
                                    LockSupport.parkNanos(this, ns);
                            }
                            p.parked = null;
                        }
                        // arena的第j个元素是p并且CAS设置arena的第j个元素由p设置为null成功
                        else if (AA.getAcquire(a, j) == p &&
                                 AA.compareAndSet(a, j, p, null)) {
                            if (m != 0)                // try to shrink
                                BOUND.compareAndSet(this, b, b + SEQ - 1);
                            p.item = null;
                            p.hash = h;
                            i = p.index >>>= 1;        // descend
                            // 如果线程被中断，则返回null值
                            if (Thread.interrupted())
                                return null;
                            if (timed && m == 0 && ns <= 0L)
                                // 如果超时，返回TIMED_OUT。
                                return TIMED_OUT;
                            break;                     // expired; restart
                        }
                    }
                }
                else
                    p.item = null;                     // clear offer
            }
            //
            else {
                if (p.bound != b) {                    // stale; reset
                    p.bound = b;
                    p.collides = 0;
                    i = (i != m || m == 0) ? m : m - 1;
                }
                else if ((c = p.collides) < m || m == FULL ||
                         !BOUND.compareAndSet(this, b, b + SEQ + 1)) {
                    p.collides = c + 1;
```

```
                    i = (i == 0) ? m : i - 1;          // cyclically
traverse
                }
                else
                    i = m + 1;                         // grow
                p.index = i;
            }
        }
    }
    // ...
}
```

## 6.5 Phaser

### 6.5.1 用Phaser替代CyclicBarrier和CountDownLatch

从JDK7开始，新增了一个同步工具类Phaser，其功能比CyclicBarrier和CountDownLatch更加强大。

#### 1.用Phaser替代CountDownLatch

考虑讲CountDownLatch时的例子，1个主线程要等10个worker线程完成之后，才能做接下来的事情，也可以用Phaser来实现此功能。在CountDownLatch中，主要是2个方法：await()和countDown()，在Phaser中，与之相对应的方法是awaitAdance(int n)和arrive()。

```
package com.lagou.concurrent.demo;

import java.util.Random;
import java.util.concurrent.Phaser;

public class Main {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(5);

        for (int i = 0; i < 5; i++) {
            new Thread("线程-" + (i + 1)) {
                private final Random random = new Random();
                @Override
                public void run() {
                    System.out.println(getName() + " - 开始运行");
                    try {
                        Thread.sleep(random.nextInt(1000));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(getName() + " - 运行结束");
                    phaser.arrive();
                }
            }.start();
        }
        System.out.println("线程启动完毕");
        phaser.awaitAdvance(phaser.getPhase());
```

```
        System.out.println("线程运行结束");
    }
}
```

### 2.用Phaser替代CyclicBarrier

考虑前面讲CyclicBarrier时，10个工程师去公司应聘的例子，也可以用Phaser实现，代码基本类似。

```
package com.lagou.concurrent.demo;

import java.util.concurrent.Phaser;

public class Main {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(5);

        for (int i = 0; i < 5; i++) {
            new MyThread("线程-" + (i + 1), phaser).start();
        }

        phaser.awaitAdvance(0);

    }
}

package com.lagou.concurrent.demo;

import java.util.Random;
import java.util.concurrent.Phaser;

public class MyThread extends Thread {

    private final Phaser phaser;
    private final Random random = new Random();

    public MyThread(String name, Phaser phaser) {
        super(name);
        this.phaser = phaser;
    }

    @Override
    public void run() {
        System.out.println(getName() + " - 开始向公司出发");
        slowly();
        System.out.println(getName() + " - 已经到达公司");
        // 到达同步点，等待其他线程
        phaser.arriveAndAwaitAdvance();

        System.out.println(getName() + " - 开始笔试");
        slowly();
        System.out.println(getName() + " - 笔试结束");
        // 到达同步点，等待其他线程
        phaser.arriveAndAwaitAdvance();
```

```
46
47          System.out.println(getName() + " - 开始面试");
48          slowly();
49          System.out.println(getName() + " - 面试结束");
50      }
51
52      private void slowly() {
53          try {
54              Thread.sleep(random.nextInt(1000));
55          } catch (InterruptedException e) {
56              e.printStackTrace();
57          }
58      }
59  }
```

arriveAndAwaitAdance()就是 arrive()与 awaitAdvance(int)的组合，表示“我自己已到达这个同步点，同时要等待所有人都到达这个同步点，然后再一起前行”。

### 6.5.2 Phaser新特性

特性1：动态调整线程个数

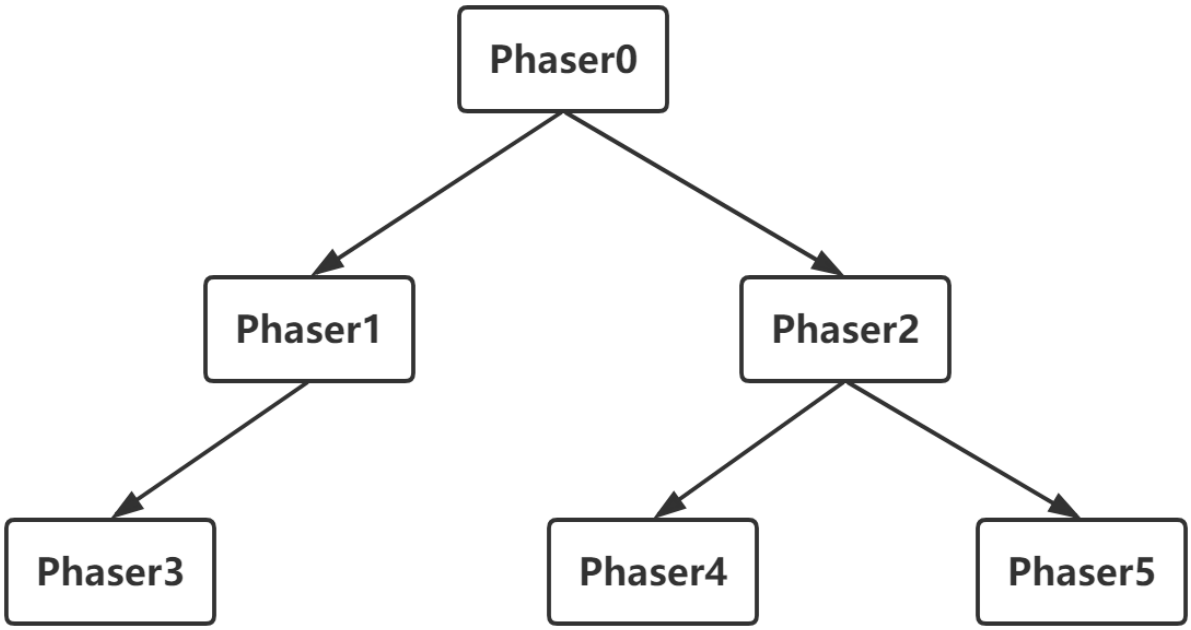
CyclicBarrier 所要同步的线程个数是在构造方法中指定的，之后不能更改，而 Phaser 可以在运行期间动态地调整要同步的线程个数。Phaser 提供了下面这些方法来增加、减少所要同步的线程个数。

```
1  register() // 注册一个
2  bulkRegister(int parties) // 注册多个
3  arriveAndDeregister()  // 解除注册
```

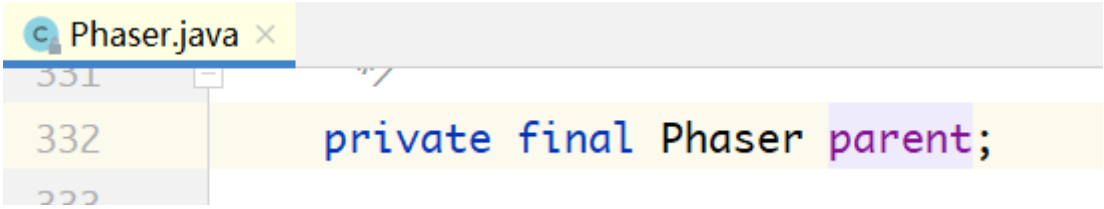
特性2：层次Phaser

多个Phaser可以组成如下图所示的树状结构，可以通过在构造方法中传入父Phaser来实现。

```
1  public Phaser(Phaser parent, int parties) {
2      // ...
3  }
```



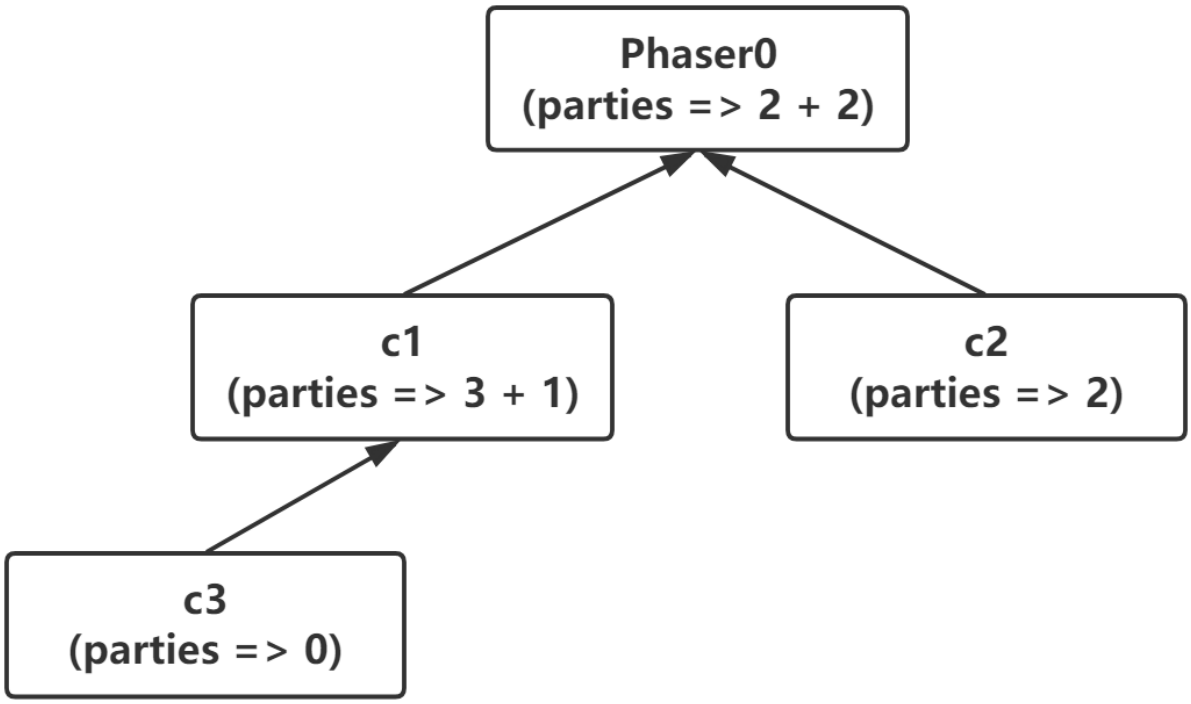

先简单看一下Phaser内部关于树状结构的存储，如下所示：




可以发现，在Phaser的内部结构中，每个Phaser记录了自己的父节点，但并没有记录自己的子节点列表。所以，每个 Phaser 知道自己的父节点是谁，但父节点并不知道自己有多少个子节点，对父节点的操作，是通过子节点来实现的。

树状的Phaser怎么使用呢？考虑如下代码，会组成下图的树状Phaser。

```
Phaser root = new Phaser(2);
Phaser c1 = new Phaser(root, 3);
Phaser c2 = new Phaser(root, 2);
Phaser c3 = new Phaser(c1, 0);
```



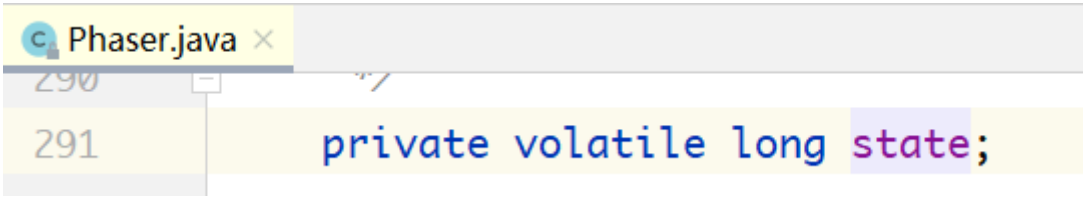

本来root有两个参与者，然后为其加入了两个子Phaser（c1，c2），每个子Phaser会算作1个参与者，root的参与者就变成2+2=4个。c1本来有3个参与者，为其加入了一个子Phaser c3，参与者数量变成3+1=4个。c3的参与者初始为0，后续可以通过调用register()方法加入。

对于树状Phaser上的每个节点来说，可以当作一个独立的Phaser来看待，其运作机制和一个单独的Phaser是一样的。

父Phaser并不用感知子Phaser的存在，当子Phaser中注册的参与者数量大于0时，会把自己向父节点注册；当子Phaser中注册的参与者数量等于0时，会自动向父节点解除注册。父Phaser把子Phaser当作一个正常参与的线程就即可。

#### 6.5.3 state变量解析

大致了解了Phaser的用法和新特性之后，下面仔细剖析其实现原理。Phaser没有基于AQS来实现，但具备AQS的核心特性：state变量、CAS操作、阻塞队列。先从state变量说起。




这个64位的state变量被拆成4部分，下图为state变量各部分：


最高位0表示未同步完成，1表示同步完成，初始最高位为0。

Phaser提供了一系列的成员方法来从state中获取上图中的几个数字，如下所示：

Phaser.java

获取当前的轮数。当前轮数同步完成，返回值是一个负数（最高位为1）

```
826    public final int getPhase() {
827        return (int)(root.state >>> PHASE_SHIFT);
828    }
```

当前phase未完成，返回值是一个负数（最高位为1）

Phaser.java

```
296    private static final int  PHASE_SHIFT     = 32;
```

Phaser.java

```
885    public boolean isTerminated() {
886        return root.state < 0L;
887    }
```

当前轮数同步完成，最高位为1

Phaser.java

获取总注册线程数

```
835    public int getRegisteredParties() {
836        return partiesOf(state);
837    }
```

Phaser.java

```
315    private static int partiesOf(long s) {
316        return (int)s >>> PARTIES_SHIFT;
317    }
```

先把state转为32位int，再右移16位

Phaser.java

```
295    private static final int  PARTIES_SHIFT   = 16;
```

Phaser.java

获取未到达的线程数

```
857    public int getUnarrivedParties() {
858        return unarrivedOf(reconcileState());
859    }
```

Phaser.java

```
310     private static int unarrivedOf(long s) {
311         int counts = (int)s;                    截取低16位
312         return (counts == EMPTY) ? 0 : (counts & UNARRIVED_MASK);
313     }
```

Phaser.java

```
297     private static final int  UNARRIVED_MASK  = 0xffff;       // to mask ints
```

下面再看一下state变量在构造方法中是如何被赋值的：

```
1  public Phaser(Phaser parent, int parties) {
2      if (parties >>> PARTIES_SHIFT != 0)
3          // 如果parties数超出了最大个数（2的16次方），抛异常
4          throw new IllegalArgumentException("Illegal number of parties");
5      // 初始化轮数为0
6      int phase = 0;
7      this.parent = parent;
8      if (parent != null) {
9          final Phaser root = parent.root;
10         // 父节点的根节点就是自己的根节点
11         this.root = root;
12         // 父节点的evenQ就是自己的evenQ
13         this.evenQ = root.evenQ;
14         // 父节点的oddQ就是自己的oddQ
15         this.oddQ = root.oddQ;
16         // 如果参与者不是0，则向父节点注册自己
17         if (parties != 0)
18             phase = parent.doRegister(1);
19     }
20     else {
21         // 如果父节点为null，则自己就是root节点
22         this.root = this;
23         // 创建奇数节点
24         this.evenQ = new AtomicReference<QNode>();
25         // 创建偶数节点
26         this.oddQ = new AtomicReference<QNode>();
27     }
28     this.state = (parties == 0) ? (long)EMPTY :
29         ((long)phase << PHASE_SHIFT) |            // 位或操作，赋值state。最高位
   为0，表示同步未完成
30         ((long)parties << PARTIES_SHIFT) |
31         ((long)parties);
32 }
```

Phaser.java

```
306     private static final int EMPTY          = 1;
```

Phaser.java

```
296     private static final int PHASE_SHIFT    = 32;
```

Phaser.java

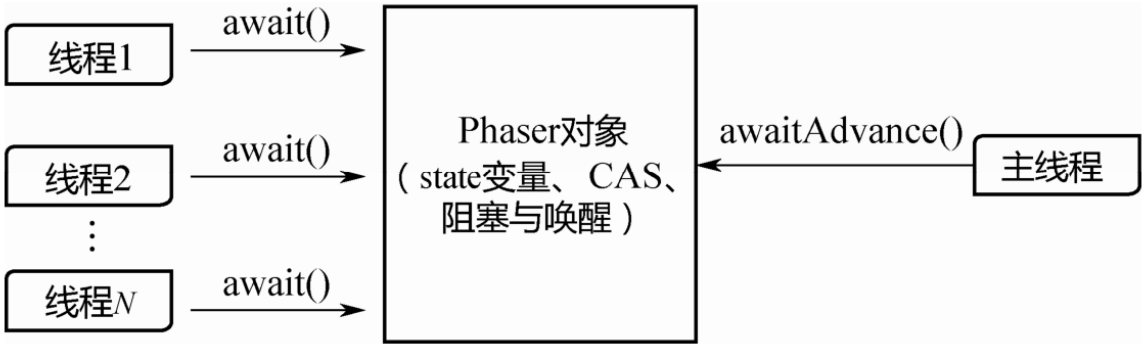
```
295     private static final int PARTIES_SHIFT   = 16;
```

当parties=0时，state被赋予一个EMPTY常量，常量为1；

当parties != 0时，把phase值左移32位；把parties左移16位；然后parties也作为最低的16位，3个值做或操作，赋值给state。

### 6.5.4 阻塞与唤醒（Treiber Stack）

基于上述的state变量，对其执行CAS操作，并进行相应的阻塞与唤醒。如下图所示，右边的主线程会调用awaitAdvance()进行阻塞；左边的arrive()会对state进行CAS的累减操作，当未到达的线程数减到0时，唤醒右边阻塞的主线程。




在这里，阻塞使用的是一个称为Treiber Stack的数据结构，而不是AQS的双向链表。Treiber Stack是一个无锁的栈，它是一个单向链表，出栈、入栈都在链表头部，所以只需要一个head指针，而不需要tail指针，如下的实现：

Phaser.java

```
1079    static final class QNode implements ForkJoinPool.ManagedBlocker {
1080        final Phaser phaser;
1081        final int phase;
1082        final boolean interruptible;
1083        final boolean timed;
1084        boolean wasInterrupted;
1085        long nanos;
1086        final long deadline;   // 每个Node节点对应一个线程
1087        volatile Thread thread; // nulled to cancel wait
1088        QNode next;  // 下一个节点的引用
```

Phaser.java

```
340     * Heads of Treiber stacks for waiting threads. To eliminate
341     * contention when releasing some threads while adding others, we
342     * use two of them, alternating across even and odd phases.
343     * Subphasers share queues with root to speed up releases.
344     */          // 两个引用表示链表的头部，可以避免线程冲突
345    private final AtomicReference<QNode> evenQ;   // 链表的头部指针
346    private final AtomicReference<QNode> oddQ;
```

为了减少并发冲突，这里定义了2个链表，也就是2个Treiber Stack。当phase为奇数轮的时候，阻塞线程放在oddQ里面；当phase为偶数轮的时候，阻塞线程放在evenQ里面。代码如下所示。

Phaser.java

```
961     private void releaseWaiters(int phase) {
962         QNode q;   // first element of queue
963         Thread t;  // its thread
964         AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
965         while ((q = head.get()) != null &&
966                 q.phase != (int)(root.state >>> PHASE_SHIFT)) {
967             if (head.compareAndSet(q, q.next) &&
968                 (t = q.thread) != null) {
```

### 6.5.5 arrive()方法分析

下面看arrive()方法是如何对state变量进行操作，又是如何唤醒线程的。

Phaser.java

```
623     public int arrive() {
624         return doArrive(ONE_ARRIVAL);
625     }
```

Phaser.java

```
643     public int arriveAndDeregister() {
644         return doArrive(ONE_DEREGISTER);
645     }
```

Phaser.java

```
303     private static final int ONE_ARRIVAL     = 1;
304     private static final int ONE_PARTY       = 1 << PARTIES_SHIFT;
305     private static final int ONE_DEREGISTER  = ONE_ARRIVAL|ONE_PARTY;
```

Phaser.java

```
295     private static final int PARTIES_SHIFT   = 16;
```

arrive()和 arriveAndDeregister()内部调用的都是 doArrive(boolean)方法。

区别在于前者只是把“未达到线程数”减1；后者则把“未到达线程数”和“下一轮的总线程数”都减1。下面看一下doArrive(boolean)方法的实现。

```
1  private int doArrive(int adjust) {
2      final Phaser root = this.root;
3      for (;;) {
```

```
        long s = (root == this) ? state : reconcileState();
        int phase = (int)(s >>> PHASE_SHIFT);
        if (phase < 0)
            return phase;
        int counts = (int)s;
        // 获取未到达线程数
        int unarrived = (counts == EMPTY) ? 0 : (counts & UNARRIVED_MASK);
        // 如果未到达线程数小于等于0，抛异常。
        if (unarrived <= 0)
            throw new IllegalStateException(badArrive(s));
        // CAS操作，将state的值减去adjust
        if (STATE.compareAndSet(this, s, s-=adjust)) {
            // 如果未到达线程数为1
            if (unarrived == 1) {
                long n = s & PARTIES_MASK;  // base of next state
                int nextUnarrived = (int)n >>> PARTIES_SHIFT;
                if (root == this) {
                    if (onAdvance(phase, nextUnarrived))
                        n |= TERMINATION_BIT;
                    else if (nextUnarrived == 0)
                        n |= EMPTY;
                    else
                        n |= nextUnarrived;
                    int nextPhase = (phase + 1) & MAX_PHASE;
                    n |= (long)nextPhase << PHASE_SHIFT;
                    STATE.compareAndSet(this, s, n);
                    releaseWaiters(phase);
                }
                // 如果下一轮的未到达线程数为0
                else if (nextUnarrived == 0) { // propagate deregistration
                    phase = parent.doArrive(ONE_DEREGISTER);
                    STATE.compareAndSet(this, s, s | EMPTY);
                }
                else
                    // 否则调用父节点的doArrive方法，传递参数1，表示当前节点已完成
                    phase = parent.doArrive(ONE_ARRIVAL);
            }
            return phase;
        }
    }
}
```

关于上面的方法，有以下几点说明：

1. 定义了2个常量如下。

   当 deregister=false 时，只最低的16位需要减 1，adj=ONE_ARRIVAL；当deregister=true 时，低32位中的2个16位都需要减1，adj=ONE_ARRIVAL|ONE_PARTY。

   Phaser.java

```
303     private static final int  ONE_ARRIVAL     = 1;
304     private static final int  ONE_PARTY       = 1 << PARTIES_SHIFT;
```

2. 把未到达线程数减1。减了之后，如果还未到0，什么都不做，直接返回。如果到0，会做2件事情：第1，重置state，把state的未到达线程个数重置到总的注册的线程数中，同时phase加1；第2，唤醒队列中的线程。

下面看一下唤醒方法：

Phaser.java

```
961     private void releaseWaiters(int phase) {
962         QNode q;   // first element of queue
963         Thread t;  // its thread  根据phase是奇数还是偶数，使用evenQ或oddQ
964         AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
965         while ((q = head.get()) != null &&
966                q.phase != (int)(root.state >>> PHASE_SHIFT)) {  遍历栈
967             if (head.compareAndSet(q, q.next) &&
968                 (t = q.thread) != null) {
969                 q.thread = null;
970                 LockSupport.unpark(t);
971             }
972         }
973     }
```

遍历整个栈，只要栈当中节点的phase不等于当前Phaser的phase，说明该节点不是当前轮的，而是前一轮的，应该被释放并唤醒。

### 6.5.6 awaitAdvance()方法分析

Phaser.java

```
712     public int awaitAdvance(int phase) {
713         final Phaser root = this.root;
714         long s = (root == this) ? state : reconcileState();  当只有一个Phaser，没有树状结构时，root就是this
715         int p = (int)(s >>> PHASE_SHIFT);
716         if (phase < 0)  phase已经结束，无需阻塞，直接返回
717             return phase;
718         if (p == phase)
719             return root.internalAwaitAdvance(phase, node: null);
720         return p;                                   阻塞在phase这一轮上
721     }
```

下面的while循环中有4个分支：

初始的时候，node==null，进入第1个分支进行自旋，自旋次数满足之后，会新建一个QNode节点；

之后执行第3、第4个分支，分别把该节点入栈并阻塞。

```
1  private int internalAwaitAdvance(int phase, QNode node) {
2      // assert root == this;
3      releaseWaiters(phase-1);          // ensure old queue clean
4      boolean queued = false;           // true when node is enqueued
5      int lastUnarrived = 0;            // to increase spins upon change
6      int spins = SPINS_PER_ARRIVAL;
7      long s;
8      int p;
9      while ((p = (int)((s = state) >>> PHASE_SHIFT)) == phase) {
10         if (node == null) {           // 不可中断模式的自旋
11             int unarrived = (int)s & UNARRIVED_MASK;
```

```
            if (unarrived != lastUnarrived &&
                (lastUnarrived = unarrived) < NCPU)
                spins += SPINS_PER_ARRIVAL;
            boolean interrupted = Thread.interrupted();
            if (interrupted || --spins < 0) { // 自旋结束，建一个节点，之后进入阻塞
                node = new QNode(this, phase, false, false, 0L);
                node.wasInterrupted = interrupted;
            }
            else
                Thread.onSpinWait();
        }
        else if (node.isReleasable()) // 从阻塞唤醒，退出while循环
            break;
        else if (!queued) {           // push onto queue
            AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
            QNode q = node.next = head.get();
            if ((q == null || q.phase == phase) &&
                (int)(state >>> PHASE_SHIFT) == phase) // avoid stale enq
                queued = head.compareAndSet(q, node); // 节点入栈
        }
        else {
            try {
                ForkJoinPool.managedBlock(node);  // 调用node.block()阻塞
            } catch (InterruptedException cantHappen) {
                node.wasInterrupted = true;
            }
        }
    }

    if (node != null) {
        if (node.thread != null)
            node.thread = null;       // avoid need for unpark()
        if (node.wasInterrupted && !node.interruptible)
            Thread.currentThread().interrupt();
        if (p == phase && (p = (int)(state >>> PHASE_SHIFT)) == phase)
            return abortWait(phase); // possibly clean up on abort
    }
    releaseWaiters(phase);
    return p;
}
```

这里调用了ForkJoinPool.managedBlock(ManagedBlocker blocker)方法，目的是把node对应的线程阻塞。ManagerdBlocker是ForkJoinPool里面的一个接口，定义如下：

```
public static interface ManagedBlocker {
    boolean block() throws InterruptedException;
    boolean isReleasable();
}
```

QNode实现了该接口，实现原理还是park()，如下所示。之所以没有直接使用park()/unpark()来实现阻塞、唤醒，而是封装了ManagedBlocker这一层，主要是出于使用上的方便考虑。一方面是park()可能被中断唤醒，另一方面是带超时时间的park()，把这二者都封装在一起。

```
static final class QNode implements ForkJoinPool.ManagedBlocker {
    final Phaser phaser;
    final int phase;
    final boolean interruptible;
    final boolean timed;
    boolean wasInterrupted;
    long nanos;
    final long deadline;
    volatile Thread thread; // nulled to cancel wait
    QNode next;
    QNode(Phaser phaser, int phase, boolean interruptible,
          boolean timed, long nanos) {
        this.phaser = phaser;
        this.phase = phase;
        this.interruptible = interruptible;
        this.nanos = nanos;
        this.timed = timed;
        this.deadline = timed ? System.nanoTime() + nanos : 0L;
        thread = Thread.currentThread();
    }
    public boolean isReleasable() {
        if (thread == null)
            return true;
        if (phaser.getPhase() != phase) {
            thread = null;
            return true;
        }
        if (Thread.interrupted())
            wasInterrupted = true;
        if (wasInterrupted && interruptible) {
            thread = null;
            return true;
        }
        if (timed &&
            (nanos <= 0L || (nanos = deadline - System.nanoTime()) <= 0L)) {
            thread = null;
            return true;
        }
        return false;
    }
    public boolean block() {
        while (!isReleasable()) {
            if (timed)
                LockSupport.parkNanos(this, nanos);
            else
                LockSupport.park(this);
        }
        return true;
    }
}
```

理解了arrive()和awaitAdvance()，arriveAndAwaitAdvance()就是二者的一个组合版本。

# 7 Atomic类

## 7.1 AtomicInteger和AtomicLong

如下面代码所示，对于一个整数的加减操作，要保证线程安全，需要加锁，也就是加synchronized关键字。

```
public class MyClass {
    private int count = 0;

    public void synchronized increment() {
        count++;
    }

    public void synchronized decrement() {
        count--;
    }
}
```

但有了Concurrent包的Atomic相关的类之后，synchronized关键字可以用AtomicInteger代替，其性能更好，对应的代码变为：

```
public class MyClass {
    private AtomicInteger count = new AtomicInteger(0);

    public void add() {
        count.getAndIncrement();
    }

    public long minus() {
        count.getAndDecrement();
    }
}
```

其对应的源码如下：

AtomicInteger.java

```
179         */
180        public final int getAndIncrement() {
181            return U.getAndAddInt( o: this, VALUE,  delta: 1);
182        }
183
```

AtomicInteger.java

```
191         */
192        public final int getAndDecrement() {
193            return U.getAndAddInt( o: this, VALUE,  delta: -1);
194        }
```

上图中的U是Unsafe的对象：

AtomicInteger.java

```
59      * are unresolved cyclic startup dependencies.
60      */
61     private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();
```

AtomicInteger的`getAndIncrement()`方法和`getAndDecrement()`方法都调用了一个方法：`U.getAndAddInt(…)`方法，该方法基于CAS实现：

AtomicInteger.java | Unsafe.java

```
2332     */
2333    @HotSpotIntrinsicCandidate
2334    public final int getAndAddInt(Object o, long offset, int delta) {
2335        int v;
2336        do {
2337            v = getIntVolatile(o, offset);
2338        } while (!weakCompareAndSetInt(o, offset, v, x: v + delta));
2339        return v;
2340    }
```

do-while循环直到判断条件返回true为止。该操作称为**自旋**。

AtomicInteger.java | Unsafe.java

```
1973    /** Volatile version of {@link #getInt(Object, long)}  */
1974    @HotSpotIntrinsicCandidate
1975    public native int     getIntVolatile(Object o, long offset);
```

`getAndAddInt`方法具有volatile的语义，也就是对所有线程都是同时可见的。

而`weakCompareAndSetInt`方法的实现：

AtomicInteger.java | Unsafe.java

```
1405    @HotSpotIntrinsicCandidate
1406    public final boolean weakCompareAndSetInt(Object o, long offset,
1407                                              int expected,
1408                                              int x) {
1409        return compareAndSetInt(o, offset, expected, x);
1410    }
```

调用了`compareAndSetInt`方法，该方法的实现：

AtomicInteger.java | Unsafe.java

```
1350
1351        /**
1352         * Atomically updates Java variable to {@code x} if it is currently
1353         * holding {@code expected}.
1354         *
1355         * <p>This operation has memory semantics of a {@code volatile} read
1356         * and write.  Corresponds to C11 atomic_compare_exchange_strong.
1357         *
1358         * @return {@code true} if successful
1359         */
1360        @HotSpotIntrinsicCandidate
1361        public final native boolean compareAndSetInt(Object o, long offset,
1362                                                     int expected,
1363                                                     int x);
```

上图中的方法中，

- 第一个参数表示要修改哪个对象的属性值；
- 第二个参数是该对象属性在内存的偏移量；
- 第三个参数表示期望值；
- 第四个参数表示要设置为的目标值。

源码比较简单，重要的是其中的设计思想。

#### 7.1.1 悲观锁与乐观锁

对于悲观锁，认为数据发生并发冲突的概率很大，读操作之前就上锁。synchronized关键字，后面要讲的ReentrantLock都是悲观锁的典型。

对于乐观锁，认为数据发生并发冲突的概率比较小，读操作之前不上锁。等到写操作的时候，再判断数据在此期间是否被其他线程修改了。如果被其他线程修改了，就把数据重新读出来，重复该过程；如果没有被修改，就写回去。判断数据是否被修改，同时写回新值，这两个操作要合成一个原子操作，也就是CAS （ Compare And Set ）。

AtomicInteger的实现就是典型的乐观锁。

#### 7.1.2 Unsafe 的CAS详解

Unsafe类是整个Concurrent包的基础，里面所有方法都是native的。具体到上面提到的compareAndSetInt方法，即：

Unsafe.java

```
1359         */
1360        @HotSpotIntrinsicCandidate
1361        public final native boolean compareAndSetInt(Object o, long offset,
1362                                                     int expected,
1363                                                     int x);
```

要特别说明一下第二个参数，它是一个long型的整数，经常被称为xxxOffset，意思是某个成员变量在对应的类中的内存偏移量（该变量在内存中的位置），表示该成员变量本身。

第二个参数的值为AtomicInteger中的属性VALUE：

AtomicInteger.java | Unsafe.java

```
191         */
192        public final int getAndDecrement() {
193            return U.getAndAddInt( o: this, VALUE, delta: -1);
194        }
```

VALUE的值：

AtomicInteger.java | Unsafe.java

```
60         */
61        private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();
62        private static final long VALUE = U.objectFieldOffset(AtomicInteger.class, name: "value");
63
```

而Unsafe的`objectFieldOffset(...)`方法调用，就是为了找到AtomicInteger类中value属性所在的内存偏移量。

`objectFieldOffset`方法的实现：

AtomicInteger.java | Unsafe.java

```
965         * @see #objectFieldOffset(Field)
966         */
967 @      public long objectFieldOffset(Class<?> c, String name) {
968            if (c == null || name == null) {
969                throw new NullPointerException();
970            }
971
972            return objectFieldOffset1(c, name);
973        }
```

其中objectFieldOffset1的实现为：

AtomicInteger.java | Unsafe.java

```
3714        private native long objectFieldOffset0(Field f);
3715        private native long objectFieldOffset1(Class<?> c, String name);
```

所有调用CAS的地方，都会先通过这个方法把成员变量转换成一个Offset。以AtomicInteger为例：

```
1  package java.util.concurrent.atomic;
2  public class AtomicInteger extends Number implements java.io.Serializable {
3      private static final jdk.internal.misc.Unsafe U =
   jdk.internal.misc.Unsafe.getUnsafe();
4      private static final long VALUE =
   U.objectFieldOffset(AtomicInteger.class, "value");
5  }
```

从上面代码可以看到，无论是Unsafe还是VALUE，都是静态的，也就是类级别的，所有对象共用的。

此处的VALUE就代表了value变量本身，后面执行CAS操作的时候，不是直接操作value，而是操作VALUE。

#### 7.1.3 自旋与阻塞

当一个线程拿不到锁的时候，有以下两种基本的等待策略：

- 策略1：放弃CPU，进入阻塞状态，等待后续被唤醒，再重新被操作系统调度。
- 策略2：不放弃CPU，空转，不断重试，也就是所谓的“**自旋**”。

很显然，如果是单核的CPU，只能用策略1。因为如果不放弃CPU，那么其他线程无法运行，也就无法释放锁。但对于多CPU或者多核，策略2就很有用了，因为没有线程切换的开销。

**AtomicInteger的实现就用的是“自旋”策略，如果拿不到锁，就会一直重试。**

注意：以上两种策略并不互斥，可以结合使用。如果获取不到锁，先自旋；如果自旋还拿不到锁，再阻塞，synchronized关键字就是这样的实现策略。

除了AtomicInteger，AtomicLong也是同样的原理。

### 7.2 AtomicBoolean和AtomicReference

#### 7.2.1 为什么需要AtomicBoolean

对于int或者long型变量，需要进行加减操作，所以要加锁；但对于一个boolean类型来说，true或false的赋值和取值操作，加上volatile关键字就够了，为什么还需要AtomicBoolean呢？

这是因为往往要实现下面这种功能：

```
if (!flag) {
    flag = true;
    // ...
}

// 或者更清晰一点的：
if (flag == false) {
    flag = true;
    // ...
}
```

也就是要实现 compare和set两个操作合在一起的原子性，而这也正是CAS提供的功能。上面的代码，就变成：

```
if (compareAndSet(false, true)) {
    // ...
}
```

同样地，AtomicReference也需要同样的功能，对应的方法如下：

AtomicReference.java

```
120      */
121     public final boolean compareAndSet(V expectedValue, V newValue) {
122         return VALUE.compareAndSet( ...args: this, expectedValue, newValue);
123     }
```

AtomicReference.java | VarHandle.java

```
725          * @see #getVolatile(Object...)
726          */
727         public final native
728         @MethodHandle.PolymorphicSignature
729         @HotSpotIntrinsicCandidate
730         boolean compareAndSet(Object... args);
731
```

其中，expect是旧的引用，update为新的引用。

### 7.2.2 如何支持boolean和double类型

在Unsafe类中，只提供了三种类型的CAS操作：int、long、Object（也就是引用类型）。如下所示：

Unsafe.java

```
1359      */
1360     @HotSpotIntrinsicCandidate
1361     public final native boolean compareAndSetInt(Object o, long offset,
1362                                                  int expected,
1363                                                  int x);
```

Unsafe.java

```
1907     @HotSpotIntrinsicCandidate
1908     public final native boolean compareAndSetLong(Object o, long offset,
1909                                                   long expected,
1910                                                   long x);
```

Unsafe.java

```
1299    @HotSpotIntrinsicCandidate
1300    public final native boolean compareAndSetObject(Object o, long offset,
1301                                                     Object expected,
1302                                                     Object x);
```

即，在jdk的实现中，这三种CAS操作都是由底层实现的，其他类型的CAS操作都要转换为这三种之一进行操作。

其中的参数：

1. 第一个参数是要修改的对象
2. 第二个参数是对象的成员变量在内存中的位置（一个long型的整数）
3. 第三个参数是该变量的旧值
4. 第四个参数是该变量的新值。

AtomicBoolean类型如何支持?

对于用int型来代替的，在入参的时候，将boolean类型转换成int类型；在返回值的时候，将int类型转换成boolean类型。如下所示：

Unsafe.java | AtomicBoolean.java

```
99      */
100     public final boolean compareAndSet(boolean expectedValue, boolean newValue) {
101         return VALUE.compareAndSet( ...args: this,
102                                     (expectedValue ? 1 : 0),
103                                     (newValue ? 1 : 0));
104     }
```

如果是double类型，又如何支持呢?

这依赖double类型提供的一对double类型和long类型互转的方法：

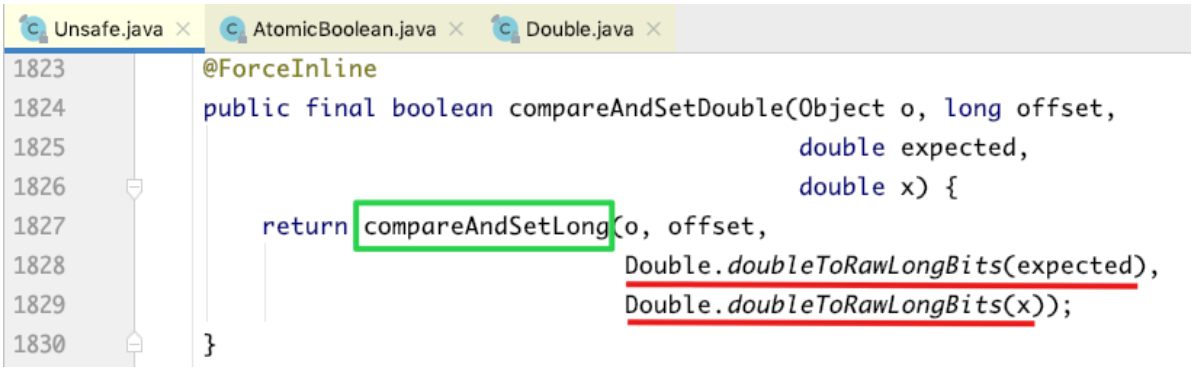
Unsafe.java | AtomicBoolean.java | Double.java

```
959         *          bit pattern.
960         */
961        @HotSpotIntrinsicCandidate
962        public static native double longBitsToDouble(long bits);
```

Unsafe.java | AtomicBoolean.java | Double.java

```
896         */
897        @HotSpotIntrinsicCandidate
898        public static native long doubleToRawLongBits(double value);
```

Unsafe类中的方法实现：




## 7.3 AtomicStampedReference和AtomicMarkableReference

### 7.3.1 ABA问题与解决办法

到目前为止，CAS都是基于“值”来做比较的。但如果另外一个线程把变量的值从A改为B，再从B改回到A，那么尽管修改过两次，可是在当前线程做CAS操作的时候，却会因为值没变而认为数据没有被其他线程修改过，这就是所谓的ABA问题。

举例来说：

小张欠小李100块，约定今天还，给打到网银。

小李家的网银余额是0，打过来之后应该是100块。

小张今天还钱这个事小李知道，小李还告诉了自己媳妇。

小张还钱，小李媳妇看到了，就取出来花掉了。

小李恰好在他媳妇取出之后检查账户，一看余额还是0。

然后找小张，要账。

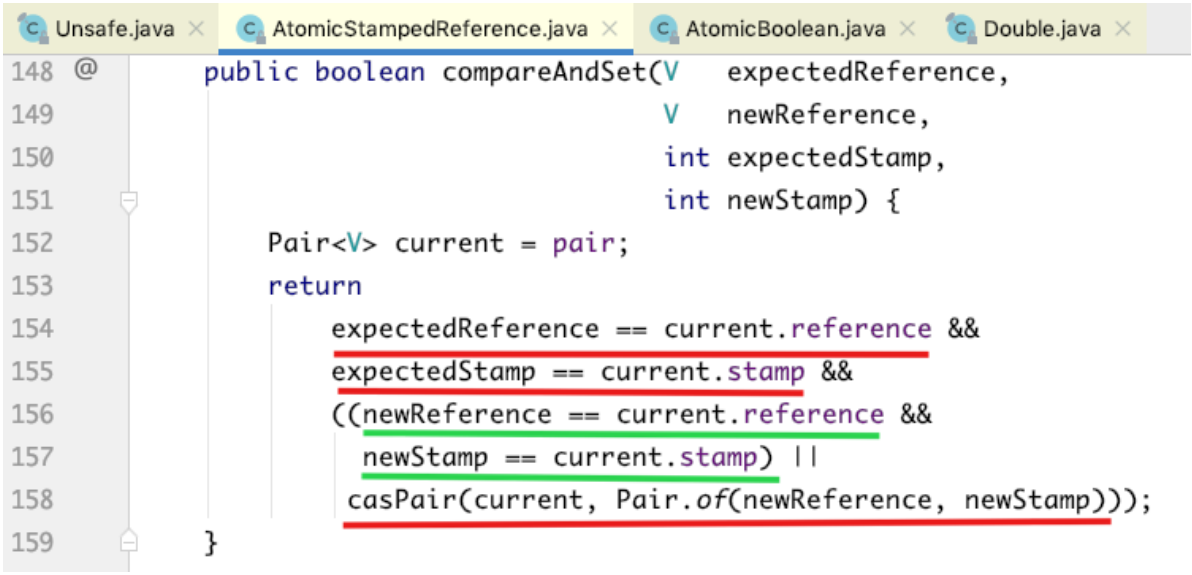
这其中，小李家的账户余额从0到100，再从100到0，小李一开始检查是0，第二次检查还是0，就认为小张没还钱。

实际上小李媳妇花掉了。

ABA问题。

其实小李可以查看账户的收支记录。

要解决 ABA 问题，不仅要比较“值”，还要比较“版本号”，而这正是 AtomicStampedReference做的事情，其对应的CAS方法如下：




之前的 CAS只有两个参数，这里的 CAS有四个参数，后两个参数就是版本号的旧值和新值。

当expectedReference != 对象当前的reference时，说明该数据肯定被其他线程修改过；

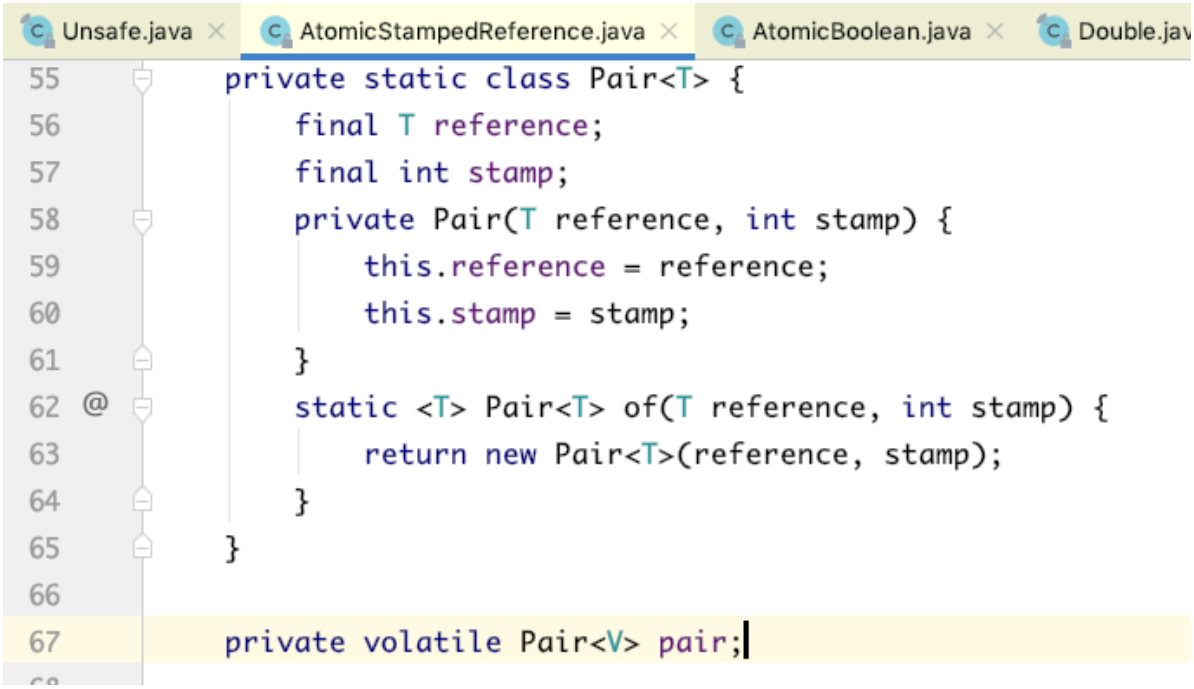
当expectedReference == 对象当前的reference时，再进一步比较expectedStamp是否等于对象当前的版本号，以此判断数据是否被其他线程修改过。

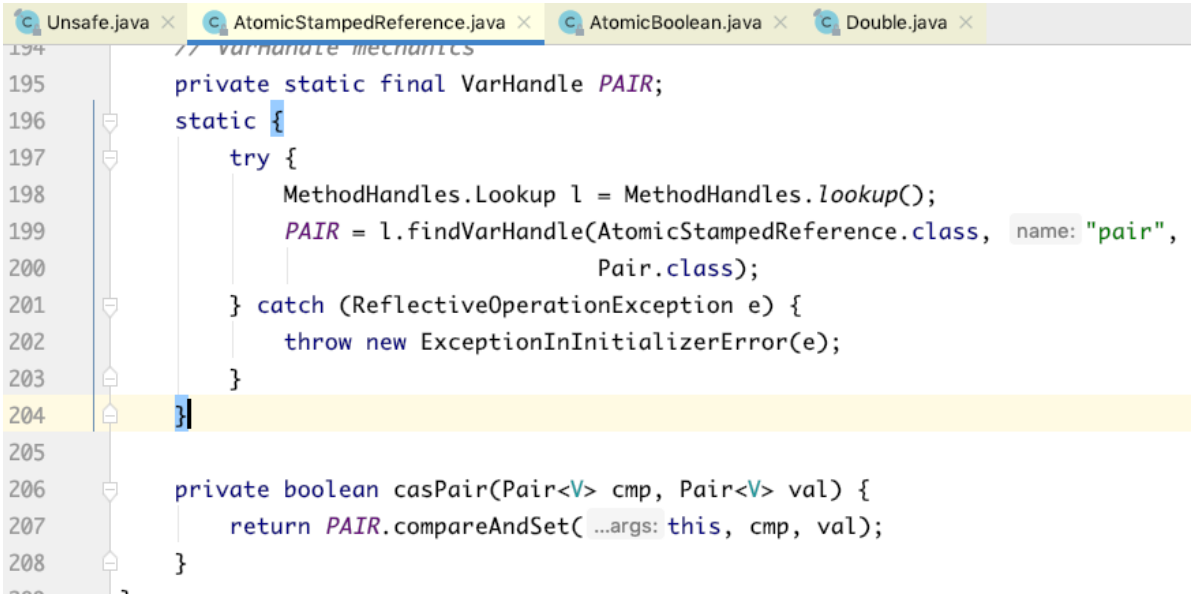
### 7.3.2 为什么没有AtomicStampedInteger或AtomictStampedLong

要解决Integer或者Long型变量的ABA问题，为什么只有AtomicStampedReference，而没有AtomicStampedInteger或者AtomictStampedLong呢?

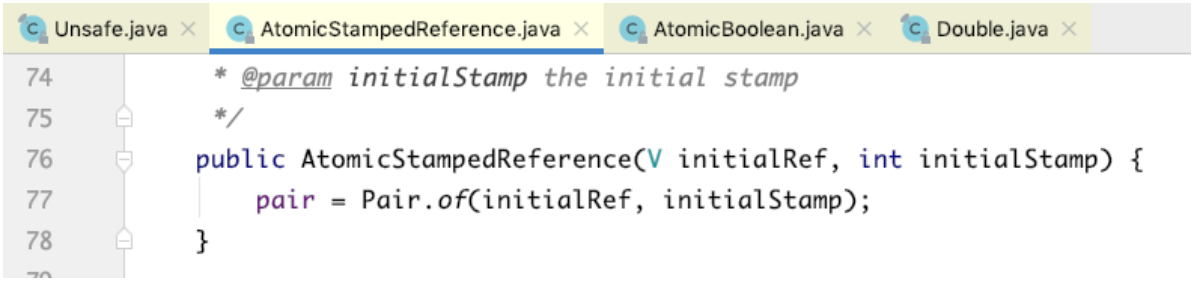
因为这里要同时比较数据的“值”和“版本号”，而Integer型或者Long型的CAS没有办法同时比较两个变量。

于是只能把值和版本号封装成一个对象，也就是这里面的Pair内部类，然后通过对象引用的CAS来实现。代码如下所示：






当使用的时候，在构造方法里面传入值和版本号两个参数，应用程序对版本号进行累加操作，然后调用上面的CAS。如下所示：




### 7.3.3 AtomicMarkableReference

AtomicMarkableReference与AtomicStampedReference原理类似，只是Pair里面的版本号是boolean类型的，而不是整型的累加变量，如下所示：

AtomicMarkableReference.java

```
52      */
53     public class AtomicMarkableReference<V> {
54
55         private static class Pair<T> {
56             final T reference;
57             final boolean mark;
58             private Pair(T reference, boolean mark) {
59                 this.reference = reference;
60                 this.mark = mark;
61             }
62 @           static <T> Pair<T> of(T reference, boolean mark) {
63                 return new Pair<T>(reference, mark);
64             }
65         }
66
67         private volatile Pair<V> pair;
68
```

因为是boolean类型，只能有true、false 两个版本号，所以并不能完全避免ABA问题，只是降低了ABA发生的概率。

## 7.4 AtomicIntegerFieldUpdater、AtomicLongFieldUpdater和AtomicReferenceFieldUpdater

### 7.4.1 为什么需要AtomicXXXFieldUpdater

如果一个类是自己编写的，则可以在编写的时候把成员变量定义为Atomic类型。但如果是一个已经有的类，在不能更改其源代码的情况下，要想实现对其成员变量的原子操作，就需要AtomicIntegerFieldUpdater、AtomicLongFieldUpdater 和 AtomicReferenceFieldUpdater。

通过AtomicIntegerFieldUpdater理解它们的实现原理。

AtomicIntegerFieldUpdater是一个抽象类。

首先，其构造方法是protected，不能直接构造其对象，必须通过它提供的一个静态方法来创建，如下所示：

AtomicIntegerFieldUpdater.java

```
97          /**
98           * Protected do-nothing constructor for use by subclasses.
99           */
100         protected AtomicIntegerFieldUpdater() {
101         }
102
```

方法`newUpdater`用于创建AtomicIntegerFieldUpdater类对象：

AtomicIntegerFieldUpdater.java

```
87          * or the field is inaccessible to the caller according to Java language
88          * access control
89          */
90         @CallerSensitive
91 @       public static <U> AtomicIntegerFieldUpdater<U> newUpdater(Class<U> tclass,
92                                                                   String fieldName) {
93             return new AtomicIntegerFieldUpdaterImpl<U>
94                 (tclass, fieldName, Reflection.getCallerClass());
95         }
96
```

newUpdater(...)静态方法传入的是要修改的类（不是对象）和对应的成员变量的名字，内部通过反射拿到这个类的成员变量，然后包装成一个AtomicIntegerFieldUpdater对象。所以，这个对象表示的是**类**的某个成员，而不是对象的成员变量。

若要修改某个对象的成员变量的值，再传入相应的对象，如下所示：

AtomicIntegerFieldUpdater.java

```
187     public int getAndIncrement(T obj) {
188         int prev, next;
189         do {
190             prev = get(obj);
191             next = prev + 1;
192         } while (!compareAndSet(obj, prev, next));
193         return prev;
194     }
```

AtomicIntegerFieldUpdater.java

```
487     }
488
489     public final boolean compareAndSet(T obj, int expect, int update) {
490         accessCheck(obj);
491         return U.compareAndSetInt(obj, offset, expect, update);
492     }
```

accecssCheck方法的作用是检查该obj是不是tclass类型，如果不是，则拒绝修改，抛出异常。

从代码可以看到，其 CAS 原理和 AtomictInteger 是一样的，底层都调用了 Unsafe 的compareAndSetInt(...)方法。

### 7.4.2 限制条件

要想使用AtomicIntegerFieldUpdater修改成员变量，成员变量必须是volatile的int类型（不能是Integer包装类），该限制从其构造方法中可以看到：

AtomicIntegerFieldUpdater.java

```
415
416             if (field.getType() != int.class)
417                 throw new IllegalArgumentException("Must be integer type");
418
419             if (!Modifier.isVolatile(modifiers))
420                 throw new IllegalArgumentException("Must be volatile type");
421
```

至于 AtomicLongFieldUpdater、AtomicReferenceFieldUpdater，也有类似的限制条件。其底层的CAS原理，也和AtomicLong、AtomicReference一样。

## 7.5 AtomicIntegerArray、AtomicLongArray和AtomicReferenceArray

Concurrent包提供了AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray三个数组元素的原子操作。注意，这里并不是说对整个数组的操作是原子的，而是针对数组中一个元素的原子操作而言。

### 7.5.1 使用方式

以AtomicIntegerArray为例，其使用方式如下：

AtomicIntegerArray.java

```
195     public final int getAndIncrement(int i) {
196         return (int)AA.getAndAdd( ...args: array, i, 1);
197     }
```

相比于AtomicInteger的getAndIncrement()方法，这里只是多了一个传入参数：数组的下标**i**。

其他方法也与此类似，相比于 AtomicInteger 的各种加减方法，也都是多一个下标 **i**，如下所示。

AtomicIntegerArray.java

```
208     public final int getAndDecrement(int i) {
209         return (int)AA.getAndAdd( ...args: array, i, -1);
210     }
```

AtomicIntegerArray.java

```
130     public final int getAndSet(int i, int newValue) {
131         return (int)AA.getAndSet(array, i, newValue);
132     }
```

AtomicIntegerArray.java

```
145    public final boolean compareAndSet(int i, int expectedValue, int newValue) {
146        return AA.compareAndSet(array, i, expectedValue, newValue);
147    }
```

AtomicIntegerArray.java

```
52    private static final VarHandle AA
53        = MethodHandles.arrayElementVarHandle(int[].class);
```

#### 7.5.2 实现原理

其底层的CAS方法直接调用VarHandle中native的getAndAdd方法。如下所示：

AtomicIntegerArray.java

```
195    public final int getAndIncrement(int i) {
196        return (int)AA.getAndAdd( ...args: array, i, 1);
197    }
```

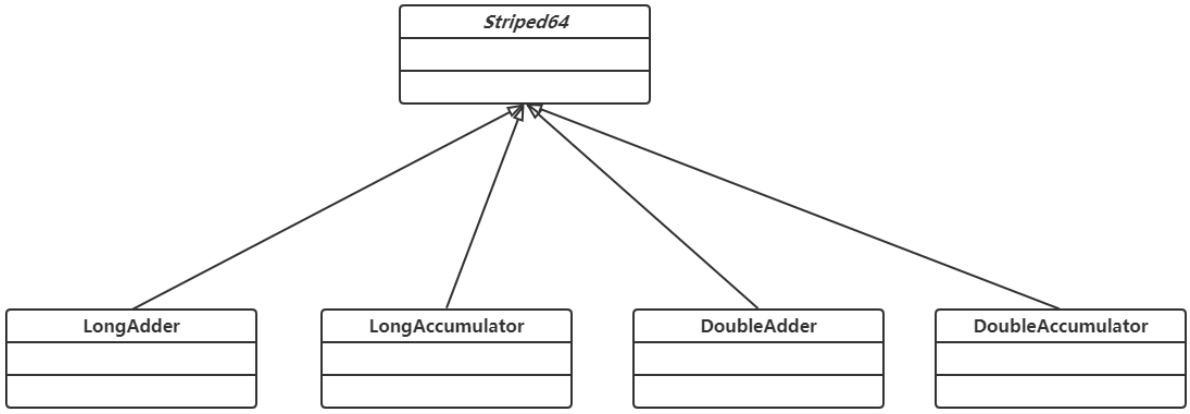
AtomicIntegerArray.java | VarHandle.java

```
1119    public final native
1120    @MethodHandle.PolymorphicSignature
1121    @HotSpotIntrinsicCandidate
1122    Object getAndAdd(Object... args);
```

明白了AtomicIntegerArray的实现原理，另外两个数组的原子类实现原理与之类似。

## 7.6 Striped64与LongAdder

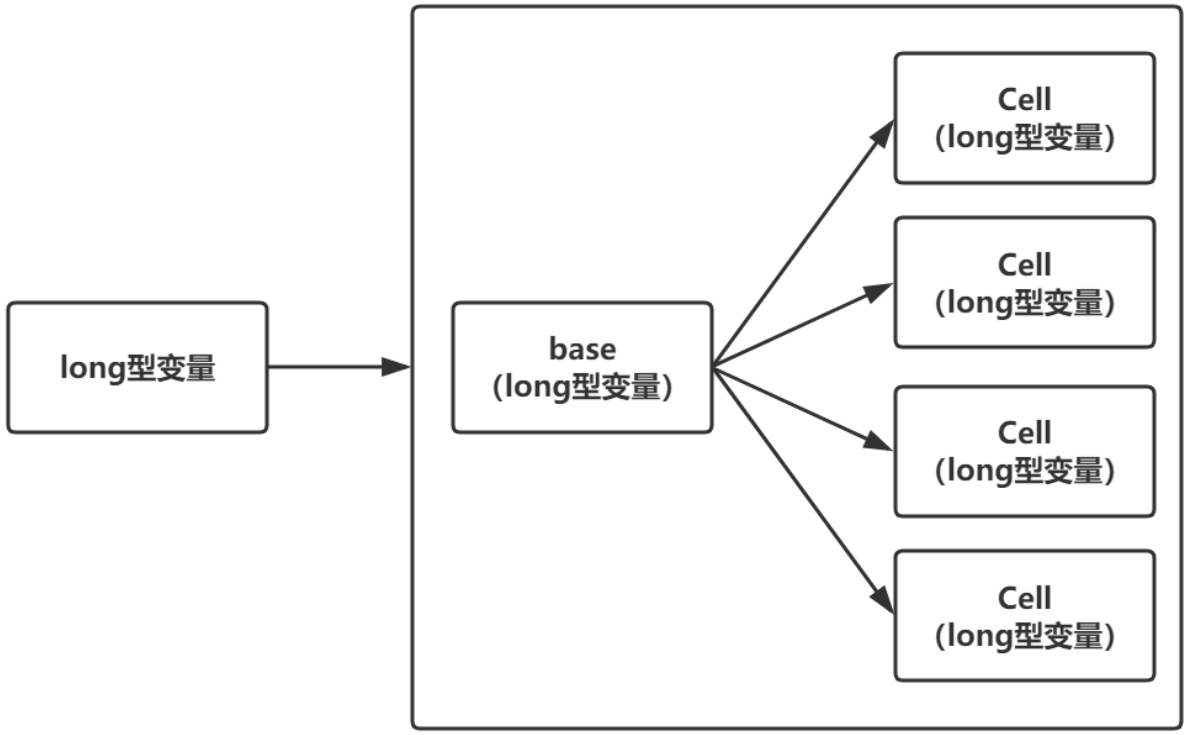
从JDK 8开始，针对Long型的原子操作，Java又提供了LongAdder、LongAccumulator；针对Double类型，Java提供了DoubleAdder、DoubleAccumulator。Striped64相关的类的继承层次如下图所示。




### 7.6.1 LongAdder原理

AtomicLong内部是一个volatile long型变量，由多个线程对这个变量进行CAS操作。多个线程同时对一个变量进行CAS操作，在高并发的场景下仍不够快，如果再要提高性能，该怎么做呢?

把一个变量拆成多份，变为多个变量，有些类似于 ConcurrentHashMap 的分段锁的例子。如下图所示，把一个Long型拆成一个base变量外加多个Cell，每个Cell包装了一个Long型变量。当多个线程并发累加的时候，如果并发度低，就直接加到base变量上；如果并发度高，冲突大，平摊到这些Cell上。在最后取值的时候，再把base和这些Cell求sum运算。




以LongAdder的sum()方法为例，如下所示。

LongAdder.java

```
119     public long sum() {
120         Cell[] cs = cells;
121         long sum = base;
122         if (cs != null) {
123             for (Cell c : cs)
124                 if (c != null)
125                     sum += c.value;
126         }
127         return sum;
128     }
```

由于无论是long，还是double，都是64位的。但因为没有double型的CAS操作，所以是通过把double型转化成long型来实现的。所以，上面的base和cell[]变量，是位于基类Striped64当中的。英文Striped意为“条带”，也就是分片。

```
1  abstract class Striped64 extends Number {
2      transient volatile Cell[] cells;
3      transient volatile long base;
4      @jdk.internal.vm.annotation.Contended static final class Cell {
5          // ...
6          volatile long value;
7          Cell(long x) { value = x; }
8          // ...
9      }
10 }
```

#### 7.6.2 最终一致性

在sum求和方法中，并没有对cells[]数组加锁。也就是说，一边有线程对其执行求和操作，一边还有线程修改数组里的值，也就是最终一致性，而不是强一致性。这也类似于ConcurrentHashMap 中的clear()方法，一边执行清空操作，一边还有线程放入数据，clear()方法调用完毕后再读取，hash map里面可能还有元素。因此，在LongAdder适合高并发的统计场景，而不适合要对某个 Long 型变量进行严格同步的场景。

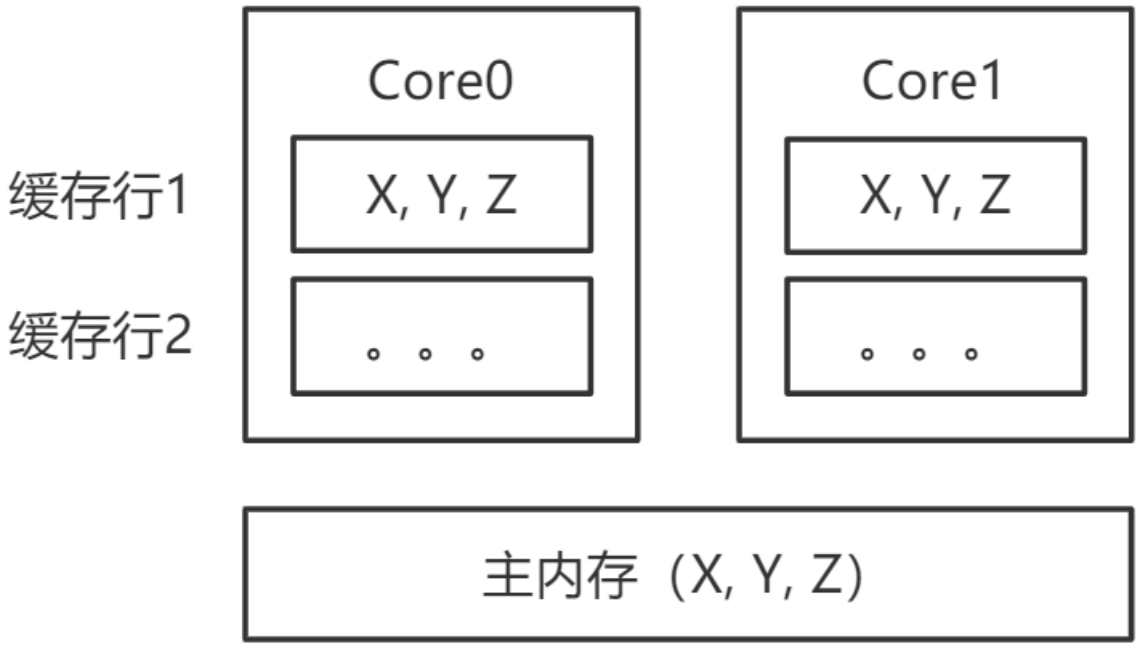
#### 7.6.3 伪共享与缓存行填充

在Cell类的定义中，用了一个独特的注解@sun.misc.Contended，这是JDK 8之后才有的，背后涉及一个很重要的优化原理：伪共享与缓存行填充。

```
LongAdder.java | Striped64.java | DoubleAccumulator.java | DoubleAdder.java | LongAccumulator
123
124     @jdk.internal.vm.annotation.Contended static final class Cell {
125         volatile long value;
```

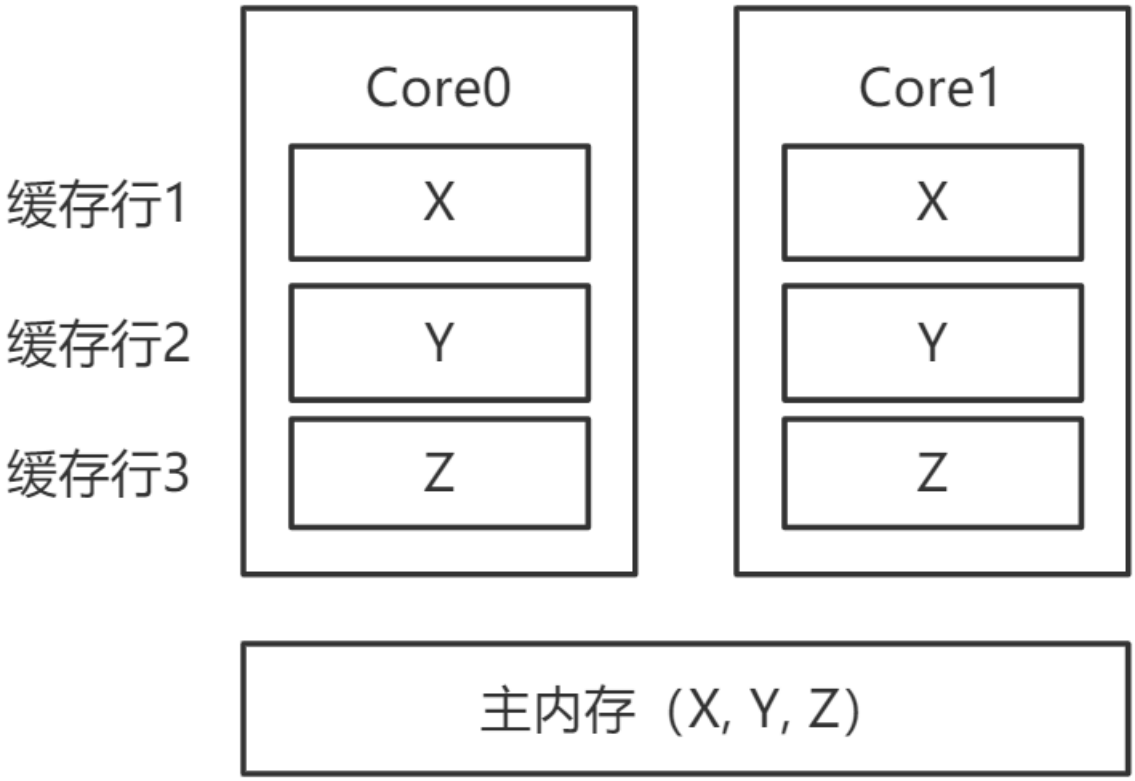
每个 CPU 都有自己的缓存。缓存与主内存进行数据交换的基本单位叫Cache Line（缓存行）。在64位x86架构中，缓存行是64字节，也就是8个Long型的大小。这也意味着当缓存失效，要刷新到主内存的时候，最少要刷新64字节。

如下图所示，主内存中有变量*X*、*Y*、*Z*（假设每个变量都是一个Long型），被CPU1和CPU2分别读入自己的缓存，放在了同一行Cache Line里面。当CPU1修改了*X*变量，它要失效整行Cache Line，也就是往总线上发消息，通知CPU 2对应的Cache Line失效。由于Cache Line是数据交换的基本单位，无法只失效*X*，要失效就会失效整行的Cache Line，这会导致*Y*、*Z*变量的缓存也失效。




虽然只修改了*X*变量，本应该只失效*X*变量的缓存，但*Y*、*Z*变量也随之失效。*Y*、*Z*变量的数据没有修改，本应该很好地被 CPU1 和 CPU2 共享，却没做到，这就是所谓的“伪共享问题”。

问题的原因是，*Y*、*Z*和*X*变量处在了同一行Cache Line里面。要解决这个问题，需要用到所谓的“缓存行填充”，分别在*X*、*Y*、*Z*后面加上7个无用的Long型，填充整个缓存行，让*X*、*Y*、*Z*处在三行不同的缓存行中，如下图所示：




声明一个@jdk.internal.vm.annotation.Contended即可实现缓存行的填充。之所以这个地方要用缓存行填充，是为了不让Cell[]数组中相邻的元素落到同一个缓存行里。

#### 7.6.4 LongAdder核心实现

下面来看LongAdder最核心的累加方法add(long x)，自增、自减操作都是通过调用该方法实现的。

LongAdder.java

```
99      public void increment() {
100         add(1L);
101     }
```

LongAdder.java

```
106     public void decrement() {
107         add(-1L);
108     }
```

LongAdder.java

```
85      public void add(long x) {
86          Cell[] cs; long b, v; int m; Cell c;
87          if ((cs = cells) != null || !casBase(b = base, val: b + x)) { 第一次尝试
88              boolean uncontended = true;
89              if (cs == null || (m = cs.length - 1) < 0 ||
90                  (c = cs[getProbe() & m]) == null ||
91                  !(uncontended = c.cas(v = c.value, val: v + x))) 第二次尝试
92                  longAccumulate(x, fn: null, uncontended);
93          }
94      }
```

当一个线程调用add(x)的时候，首先会尝试使用casBase把x加到base变量上。如果不成功，则再用c.cas(...)方法尝试把 x 加到 Cell 数组的某个元素上。如果还不成功，最后再调用longAccumulate(...)方法。

**注意：**Cell[]数组的大小始终是2的整数次方，在运行中会不断扩容，每次扩容都是增长2倍。上面代码中的`cs[getProbe() & m]`其实就是对数组的大小取模。因为m=cs.length–1，getProbe()为该线程生成一个随机数，用该随机数对数组的长度取模。因为数组长度是2的整数次方，所以可以用&操作来优化取模运算。

对于一个线程来说，它并不在意到底是把x累加到base上面，还是累加到Cell[]数组上面，只要累加成功就可以。因此，这里使用随机数来实现Cell的长度取模。

如果两次尝试都不成功，则调用 longAccumulate(...)方法，该方法在 Striped64 里面LongAccumulator也会用到，如下所示。

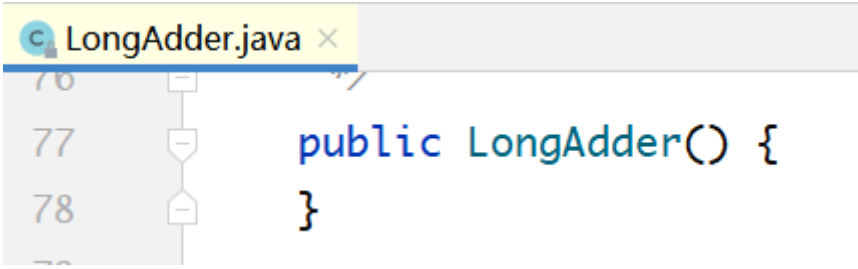
```
final void longAccumulate(long x, LongBinaryOperator fn,
                          boolean wasUncontended) {
    int h;
    if ((h = getProbe()) == 0) {
        ThreadLocalRandom.current(); // force initialization
        h = getProbe();
        wasUncontended = true;
    }
    // true表示最后一个slot非空
    boolean collide = false;
    done: for (;;) {
        Cell[] cs; Cell c; int n; long v;
        // 如果cells不是null，且cells长度大于0
        if ((cs = cells) != null && (n = cs.length) > 0) {
            // cells最大下标对随机数取模，得到新下标。
            // 如果此新下标处的元素是null
            if ((c = cs[(n - 1) & h]) == null) {
                // 自旋锁标识，用于创建cells或扩容cells
                if (cellsBusy == 0) {       // 尝试添加新的Cell
                    Cell r = new Cell(x);   // Optimistically create
                    // 如果cellsBusy为0，则CAS操作cellsBusy为1，获取锁
                    if (cellsBusy == 0 && casCellsBusy()) {
                        try {               // 获取锁之后，再次检查
                            Cell[] rs; int m, j;
                            if ((rs = cells) != null &&
```

```
                                (m = rs.length) > 0 &&
                                rs[j = (m - 1) & h] == null) {
                                // 赋值成功，返回
                                rs[j] = r;
                                break done;
                            }
                        } finally {
                            // 重置标志位，释放锁
                            cellsBusy = 0;
                        }
                        continue;           // 如果slot非空，则进入下一次循环
                    }
                }
                collide = false;
            }
            else if (!wasUncontended)       // CAS操作失败
                wasUncontended = true;      // rehash之后继续
            else if (c.cas(v = c.value,
                           (fn == null) ? v + x : fn.applyAsLong(v, x)))
                break;
            else if (n >= NCPU || cells != cs)
                collide = false;            // At max size or stale
            else if (!collide)
                collide = true;
            else if (cellsBusy == 0 && casCellsBusy()) {
                try {
                    if (cells == cs)        // 扩容，每次都是上次的两倍长度
                        cells = Arrays.copyOf(cs, n << 1);
                } finally {
                    cellsBusy = 0;
                }
                collide = false;
                continue;                   // Retry with expanded table
            }
            h = advanceProbe(h);
        }
        // 如果cells为null或者cells的长度为0，则需要初始化cells数组
        // 此时需要加锁，进行CAS操作
        else if (cellsBusy == 0 && cells == cs && casCellsBusy()) {
            try {                           // Initialize table
                if (cells == cs) {
                    // 实例化Cell数组，实例化Cell，保存x值
                    Cell[] rs = new Cell[2];
                    // h为随机数，对Cells数组取模，赋值新的Cell对象。
                    rs[h & 1] = new Cell(x);
                    cells = rs;
                    break done;
                }
            } finally {
                // 释放CAS锁
                cellsBusy = 0;
            }
        }
        // 如果CAS操作失败，最后回到对base的操作
        // 判断fn是否为null，如果是null则执行加操作，否则执行fn提供的操作
        // 如果操作失败，则重试for循环流程，成功就退出循环
        else if (casBase(v = base,
                         (fn == null) ? v + x : fn.applyAsLong(v, x)))
```

```
            break done;
        }
    }
```

### 7.6.5 LongAccumulator

LongAccumulator的原理和LongAdder类似，只是**功能更强大**，下面为两者构造方法的对比：




LongAdder.java | LongAccumulator.java

```
94  public LongAccumulator(LongBinaryOperator accumulatorFunction,
95                         long identity) {
96      this.function = accumulatorFunction;
97      base = this.identity = identity;
98  }
```

LongAdder只能进行累加操作，并且初始值默认为0；LongAccumulator可以自己定义一个二元操作符，并且可以传入一个初始值。

LongAdder.java | LongAccumulator.java | LongBinaryOperator.java

```
39  @FunctionalInterface
40  public interface LongBinaryOperator {
41
42      /**
43       * Applies this operator to the given operands.
44       *
45       * @param left the first operand
46       * @param right the second operand
47       * @return the operator result
48       */
49      long applyAsLong(long left, long right);
50  }
```

操作符的左值，就是base变量或者Cells[]中元素的当前值；右值，就是add()方法传入的参数x。

下面是LongAccumulator的accumulate(x)方法，与LongAdder的add(x)方法类似，最后都是调用的Striped64的LongAccumulate(...)方法。

唯一的差别就是LongAdder的add(x)方法调用的是casBase(b, b+x)，这里调用的是casBase(b, r)，其中，r=function.applyAsLong(b=base, x)。

LongAdder.java | LongAccumulator.java | LongBinaryOperator.java

```
105     public void accumulate(long x) {
106         Cell[] cs; long b, v, r; int m; Cell c;
107         if ((cs = cells) != null
108             || ((r = function.applyAsLong(b = base, x)) != b
109                 && !casBase(b, r))) {
110             boolean uncontended = true;
111             if (cs == null
112                 || (m = cs.length - 1) < 0
113                 || (c = cs[getProbe() & m]) == null
114                 || !(uncontended =
115                      (r = function.applyAsLong(v = c.value, x)) == v
116                      || c.cas(v, r)))
117                 longAccumulate(x, function, uncontended);
118         }
119     }
```

### 7.6.6 DoubleAdder与DoubleAccumulator

DoubleAdder 其实也是用 long 型实现的，因为没有 double 类型的 CAS 方法。下面是DoubleAdder的add(x)方法，和LongAdder的add(x)方法基本一样，只是多了long和double类型的相互转换。

DoubleAdder.java

```
89  public void add(double x) {
90      Cell[] cs; long b, v; int m; Cell c;
91      if ((cs = cells) != null ||
92          !casBase(b = base,
93                   Double.doubleToRawLongBits
94                   ( value: Double.longBitsToDouble(b) + x))) {
95          boolean uncontended = true;
96          if (cs == null || (m = cs.length - 1) < 0 ||
97              (c = cs[getProbe() & m]) == null ||
98              !(uncontended = c.cas(v = c.value,
99                                    Double.doubleToRawLongBits
100                                   ( value: Double.longBitsToDouble(v) + x))))
101             doubleAccumulate(x,  fn: null, uncontended);
102     }
103 }
```

其中的关键Double.doubleToRawLongBits(Double.longBitsToDouble(b) + x)，在读出来的时候，它把 long 类型转换成 double 类型，然后进行累加，累加的结果再转换成 long 类型，通过CAS写回去。

DoubleAccumulate也是Striped64的成员方法，和longAccumulate类似，也是多了long类型和double类型的互相转换。

DoubleAccumulator和DoubleAdder的关系，与LongAccumulator和LongAdder的关系类似，只是多了一个二元操作符。

# 8 Lock与Condition

## 8.1 互斥锁

### 8.1.1 锁的可重入性

“可重入锁”是指当一个线程调用 object.lock()获取到锁，进入临界区后，再次调用object.lock()，仍然可以获取到该锁。显然，通常的锁都要设计成可重入的，否则就会发生死锁。

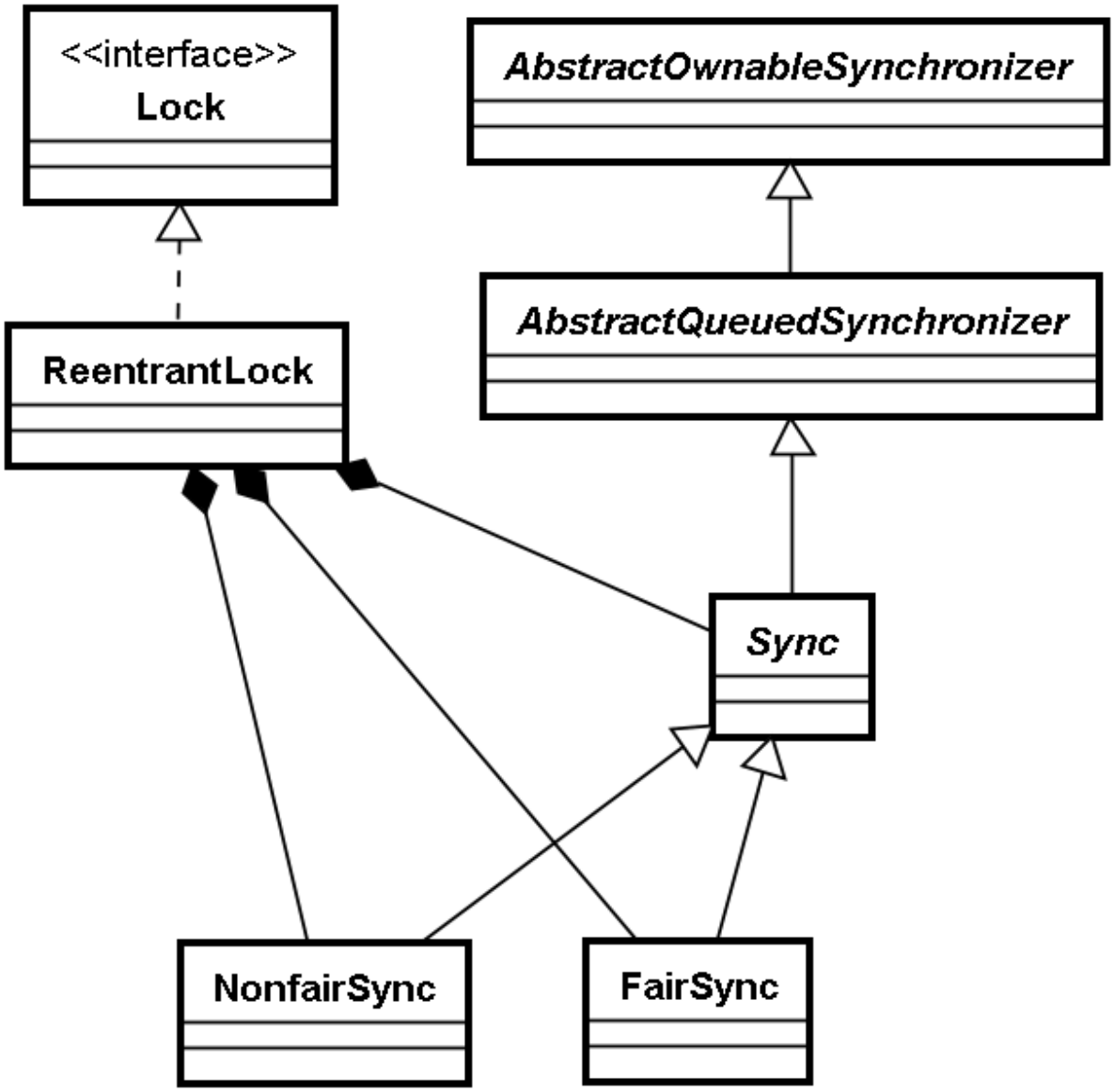
synchronized关键字，就是可重入锁。如下所示：

在一个synchronized方法method1()里面调用另外一个synchronized方法method2()。如果synchronized关键字不可重入，那么在method2()处就会发生阻塞，这显然不可行。

```
public void synchronized method1() {
    // ...
    method2();
    // ...
}

public void synchronized method2() {
    // ...
}
```

### 8.1.2 类继承层次

在正式介绍锁的实现原理之前，先看一下 Concurrent 包中的与互斥锁（ReentrantLock）相关类之间的继承层次，如下图所示：




Lock是一个接口，其定义如下：

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

常用的方法是lock()/unlock()。lock()不能被中断，对应的lockInterruptibly()可以被中断。

ReentrantLock本身没有代码逻辑，实现都在其内部类Sync中：

```
public class ReentrantLock implements Lock, java.io.Serializable {
    private final Sync sync;

    public void lock() {
        sync.acquire(1);
    }

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

#### 8.1.3 锁的公平性vs.非公平性

Sync是一个抽象类，它有两个子类FairSync与NonfairSync，分别对应公平锁和非公平锁。从下面的ReentrantLock构造方法可以看出，会传入一个布尔类型的变量fair指定锁是公平的还是非公平的，默认为非公平的。

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

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

什么叫公平锁和非公平锁呢？先举个现实生活中的例子，一个人去火车站售票窗口买票，发现现场有人排队，于是他排在队伍末尾，遵循先到者优先服务的规则，这叫公平；如果他去了不排队，直接冲到窗口买票，这叫作不公平。

对应到锁的例子，一个新的线程来了之后，看到有很多线程在排队，自己排到队伍末尾，这叫公平；线程来了之后直接去抢锁，这叫作不公平。默认设置的是非公平锁，其实是为了提高效率，减少线程切换。

**锁实现的基本原理**

Sync的父类AbstractQueuedSynchronizer经常被称作队列同步器（**AQS**），这个类非常**重要**，该类的父类是AbstractOwnableSynchronizer。

此处的锁具备synchronized功能，即可以阻塞一个线程。为了实现一把具有阻塞或唤醒功能的锁，需要几个核心要素：

1. 需要一个state变量，标记该锁的状态。state变量至少有两个值：0、1。对state变量的操作，使用CAS保证线程安全。
2. 需要记录当前是哪个线程持有锁。
3. 需要底层支持对一个线程进行**阻塞**或**唤醒**操作。
4. 需要有一个**队列**维护所有阻塞的线程。这个队列也必须是线程安全的无锁队列，也需要使用CAS。

针对要素1和2，在上面两个类中有对应的体现：

```
public abstract class AbstractOwnableSynchronizer implements
java.io.Serializable {
    // ...
    private transient Thread exclusiveOwnerThread;  // 记录持有锁的线程
}

public abstract class AbstractQueuedSynchronizer extends
AbstractOwnableSynchronizer implements java.io.Serializable {

    private volatile int state; // 记录锁的状态，通过CAS修改state的值。
    // ...

}
```

state取值不仅可以是0、1，还可以大于1，就是为了支持锁的可重入性。例如，同样一个线程，调用5次lock，state会变成5；然后调用5次unlock，state减为0。

当state=0时，没有线程持有锁，exclusiveOwnerThread=null；

当state=1时，有一个线程持有锁，exclusiveOwnerThread=该线程；

当state > 1时，说明该线程重入了该锁。

对于要素3，Unsafe类提供了阻塞或唤醒线程的一对操作原语，也就是park/unpark。

```
public native void unpark(Object thread);
public native void park(boolean isAbsolute, long time);
```

有一个LockSupport的工具类，对这一对原语做了简单封装：

```
public class LockSupport {
    // ...

    private static final Unsafe U = Unsafe.getUnsafe();

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

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

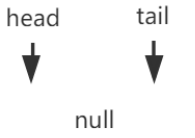
**在当前线程中调用park()，该线程就会被阻塞；在另外一个线程中，调用unpark(Thread thread)，传入一个被阻塞的线程，就可以唤醒阻塞在park()地方的线程。**

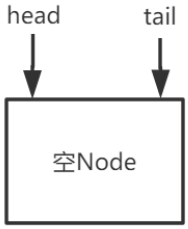
unpark(Thread thread)，它实现了一个线程对另外一个线程的“精准唤醒”。notify也只是唤醒某一个线程，但无法指定具体唤醒哪个线程。

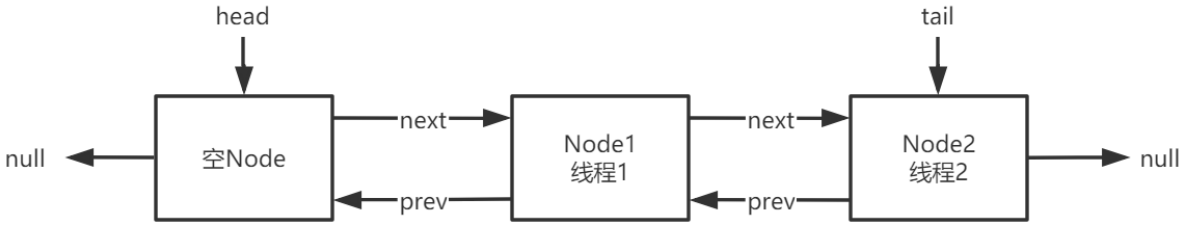
针对要素4，在AQS中利用双向链表和CAS实现了一个阻塞队列。如下所示：

```
public abstract class AbstractQueuedSynchronizer {
    // ...
    static final class Node {
        volatile Thread thread; // 每个Node对应一个被阻塞的线程
        volatile Node prev;
        volatile Node next;
        // ...
    }

    private transient volatile Node head;
    private transient volatile Node tail;
    // ...
}
```

**阻塞队列是整个AQS核心中的核心。**如下图所示，head指向双向链表头部，tail指向双向链表尾部。入队就是把新的Node加到tail后面，然后对tail进行CAS操作；出队就是对head进行CAS操作，把head向后移一个位置。




初始的时候，head=tail=NULL；然后，在往队列中加入阻塞的线程时，会新建一个空的Node，让head和tail都指向这个空Node；之后，在后面加入被阻塞的线程对象。所以，当head=tail的时候，说明队列为空。

### 8.1.4 公平与非公平的lock()实现差异

下面分析基于AQS，ReentrantLock在公平性和非公平性上的实现差异。

ReentrantLock.java

```
196     static final class NonfairSync extends Sync {
197         private static final long serialVersionUID = 7316153563782823691L;
198         protected final boolean tryAcquire(int acquires) {
199             return nonfairTryAcquire(acquires);
200         }
201     }
```

ReentrantLock.java

```
126     final boolean nonfairTryAcquire(int acquires) {
127         final Thread current = Thread.currentThread();
128         int c = getState();
129         if (c == 0) {  如果state为0，直接将当前线程设置为排他线程，同时设置state的值
130             if (compareAndSetState( expect: 0, acquires)) {
131                 setExclusiveOwnerThread(current);
132                 return true;
133             }
134         }    如果state不是0，但是排他线程就是当前线程，则直接设置state的值
135         else if (current == getExclusiveOwnerThread()) {
136             int nextc = c + acquires;
137             if (nextc < 0) // overflow
138                 throw new Error("Maximum lock count exceeded");
139             setState(nextc);
140             return true;
141         }
142         return false;    否则返回false，获取失败
143     }
```

此处没有考虑队列中有没有其他线程
直接使用当前线程获取锁，不排队，不公平

ReentrantLock.java

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

ReentrantLock.java

```
212         @ReservedStackAccess
213         protected final boolean tryAcquire(int acquires) {
214             final Thread current = Thread.currentThread();
215             int c = getState();
216             if (c == 0) {    如果state为0，且队列中没有等待的线程，则设置当前线程
217                 if (!hasQueuedPredecessors() && 为排他线程，同时设置state的值
218                     compareAndSetState( expect: 0, acquires)) {
219                     setExclusiveOwnerThread(current);
220                     return true;
221                 }
222             }    如果排他线程就是当前线程，才直接设置state的值
223             else if (current == getExclusiveOwnerThread()) {
224                 int nextc = c + acquires;
225                 if (nextc < 0)
226                     throw new Error("Maximum lock count exceeded");
227                 setState(nextc);
228                 return true;
229             }
230             return false;
231         }
232     }
```

#### 8.1.5 阻塞队列与唤醒机制

下面进入锁的最为关键的部分，即acquireQueued(...)方法内部一探究竟。

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
1238     public final void acquire(int arg) {
1239         if (!tryAcquire(arg) &&
1240             acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
1241             selfInterrupt();
1242     }
```

先说addWaiter(...)方法，就是为当前线程生成一个Node，然后把Node放入双向链表的尾部。要注意的是，这只是把Thread对象放入了一个队列中而已，线程本身并未阻塞。

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
649         */
650     private Node addWaiter(Node mode) {
651         Node node = new Node(mode);
652 
653         for (;;) {
654             Node oldTail = tail;
655             if (oldTail != null) {
656                 node.setPrevRelaxed(oldTail);
657                 if (compareAndSetTail(oldTail, node)) {
658                     oldTail.next = node;
659                     return node;
660                 }
661             } else {
662                 initializeSyncQueue();
663             }
664         }
665     }
```

创建节点，尝试将节点追加到队列尾部。获取tail节点，将tail节点的next设置为当前节点。

如果tail不存在，就初始化队列。

在addWaiter(...)方法把Thread对象加入阻塞队列之后的工作就要靠acquireQueued(...)方法完成。线程一旦进入acquireQueued(...)就会被无限期阻塞，即使有其他线程调用interrupt()方法也不能将其唤醒，除非有其他线程释放了锁，并且该线程拿到了锁，才会从accquireQueued(...)返回。

进入acquireQueued(...)，该线程被阻塞。在该方法返回的一刻，就是拿到锁的那一刻，也就是被唤醒的那一刻，此时会删除队列的第一个元素（head指针前移1个节点）。

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
906    final boolean acquireQueued(final Node node, int arg) {
907        boolean interrupted = false;
908        try {
909            for (;;) {
910                final Node p = node.predecessor();
911                if (p == head && tryAcquire(arg)) {
912                    setHead(node);
913                    p.next = null; // help GC
914                    return interrupted;
915                }
916                if (shouldParkAfterFailedAcquire(p, node))
917                    interrupted |= parkAndCheckInterrupt();
918            }
919        } catch (Throwable t) {
920            cancelAcquire(node);
921            if (interrupted)
922                selfInterrupt();
923            throw t;
924        }
925    }
```

首先，acquireQueued(...)方法有一个返回值，表示什么意思呢？虽然该方法不会中断响应，但它会记录被阻塞期间有没有其他线程向它发送过中断信号。如果有，则该方法会返回true；否则，返回false。

基于这个返回值，才有了下面的代码：

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
1238    public final void acquire(int arg) {
1239        if (!tryAcquire(arg) &&
1240            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
1241            selfInterrupt();
1242    }
```

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
875    static void selfInterrupt() {
876        Thread.currentThread().interrupt();
877    }
```

当 acquireQueued(...)返回 true 时，会调用 selfInterrupt()，自己给自己发送中断信号，也就是自己把自己的中断标志位设为true。之所以要这么做，是因为自己在阻塞期间，收到其他线程中断信号没有及时响应，现在要进行补偿。这样一来，如果该线程在lock代码块内部有调用sleep()之类的阻塞方法，就可以抛出异常，响应该中断信号。

阻塞就发生在下面这个方法中：

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
884    private final boolean parkAndCheckInterrupt() {
885        LockSupport.park( blocker: this);
886        return Thread.interrupted();
887    }
```

线程调用 park()方法，自己把自己阻塞起来，直到被其他线程唤醒，该方法返回。

park()方法返回有两种情况。

1. 其他线程调用了unpark(Thread t)。
2. 其他线程调用了t.interrupt()。这里要注意的是，lock()不能响应中断，但LockSupport.park()会响应中断。

也正因为LockSupport.park()可能被中断唤醒，acquireQueued(...)方法才写了一个for死循环。唤醒之后，如果发现自己排在队列头部，就去拿锁；如果拿不到锁，则再次自己阻塞自己。不断重复此过程，直到拿到锁。

被唤醒之后，通过Thread.interrupted()来判断是否被中断唤醒。如果是情况1，会返回false；如果是情况2，则返回true。

#### 8.1.6 unlock()实现分析

说完了lock，下面分析unlock的实现。unlock不区分公平还是非公平。

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
438    public void unlock() {
439        sync.release( arg: 1);
440    }
```

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
1300
1301    public final boolean release(int arg) {
1302        if (tryRelease(arg)) {
1303            Node h = head;
1304            if (h != null && h.waitStatus != 0)
1305                unparkSuccessor(h);
1306            return true;
1307        }
1308        return false;
1309    }
```

上图中，当前线程要释放锁，先调用tryRelease(arg)方法，如果返回true，则取出head，让head获取锁。

对于tryRelease方法：

ReentrantLock.java × AbstractQueuedSynchronizer.java ×

```
145         @ReservedStackAccess
146     protected final boolean tryRelease(int releases) {
147         int c = getState() - releases;
148         if (Thread.currentThread() != getExclusiveOwnerThread())
149             throw new IllegalMonitorStateException();
150         boolean free = false;
151         if (c == 0) {
152             free = true;
153             setExclusiveOwnerThread(null);
154         }
155         setState(c);
156         return free;
157     }
```

首先计算当前线程释放锁后的state值。

如果当前线程不是排他线程，则抛异常，因为只有获取锁的线程才可以进行释放锁的操作。

此时设置state，没有使用CAS，因为是单线程操作。

再看unparkSuccessor方法：

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
685    private void unparkSuccessor(Node node) {
686        /*
687         * If status is negative (i.e., possibly needing signal) try
688         * to clear in anticipation of signalling.  It is OK if this
689         * fails or if status is changed by waiting thread.
690         */
691        int ws = node.waitStatus;
692        if (ws < 0)
693            node.compareAndSetWaitStatus(ws, update: 0);
```

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
701        Node s = node.next;
702        if (s == null || s.waitStatus > 0) {
703            s = null;
704            for (Node p = tail; p != node && p != null; p = p.prev)
705                if (p.waitStatus <= 0)
706                    s = p;
707        }
708        if (s != null)
709            LockSupport.unpark(s.thread); 唤醒head节点的线程
710    }
```

release()里面做了两件事：tryRelease(...)方法释放锁；unparkSuccessor(...)方法唤醒队列中的后继者。

### 8.1.7 lockInterruptibly()实现分析

上面的 lock 不能被中断，这里的 lockInterruptibly()可以被中断：

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
316    public void lockInterruptibly() throws InterruptedException {
317        sync.acquireInterruptibly( arg: 1);
318    }
```

可中断地获取锁

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
1258    public final void acquireInterruptibly(int arg)
1259            throws InterruptedException {
1260        if (Thread.interrupted())
1261            throw new InterruptedException();
1262        if (!tryAcquire(arg))
1263            doAcquireInterruptibly(arg);
1264    }
```

这里的 acquireInterruptibly(...)也是 AQS 的模板方法，里面的 tryAcquire(...)分别被 FairSync和NonfairSync实现。

主要看doAcquireInterruptibly(...)方法：

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
931     private void doAcquireInterruptibly(int arg)
932         throws InterruptedException {
933         final Node node = addWaiter(Node.EXCLUSIVE);
934         try {
935             for (;;) {
936                 final Node p = node.predecessor();
937                 if (p == head && tryAcquire(arg)) {
938                     setHead(node);
939                     p.next = null; // help GC
940                     return;
941                 }
942                 if (shouldParkAfterFailedAcquire(p, node) &&
943                     parkAndCheckInterrupt())
944                     throw new InterruptedException();
945             }                    收到中断信号，直接抛异常，不阻塞，直接返回
946         } catch (Throwable t) {
947             cancelAcquire(node);
948             throw t;
949         }
950     }
```

当parkAndCheckInterrupt()返回true的时候，说明有其他线程发送中断信号，直接抛出InterruptedException，跳出for循环，整个方法返回。

#### 8.1.8 tryLock()实现分析

ReentrantLock.java | AbstractQueuedSynchronizer.java

```
346     public boolean tryLock() {
347         return sync.nonfairTryAcquire( acquires: 1);
348     }
```

tryLock()实现基于调用非公平锁的tryAcquire(...)，对state进行CAS操作，如果操作成功就拿到锁；如果操作不成功则直接返回false，也不阻塞。

## 8.2 读写锁

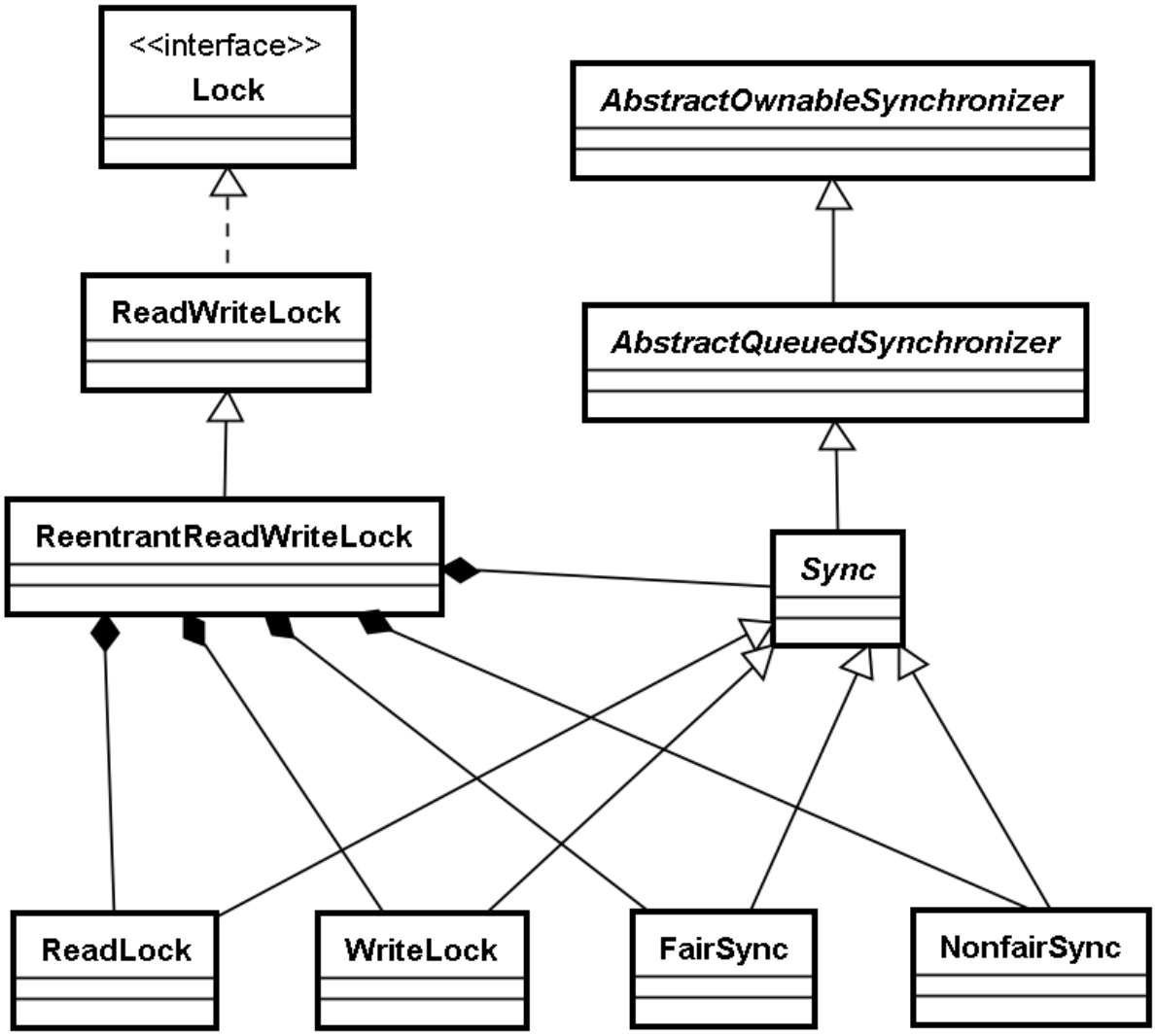
和互斥锁相比，读写锁（ReentrantReadWriteLock）就是读线程和读线程之间不互斥。

读读不互斥，读写互斥，写写互斥

### 8.2.1 类继承层次

ReadWriteLock是一个接口，内部由两个Lock接口组成。

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```




ReentrantReadWriteLock实现了该接口，使用方式如下：

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
Lock readLock = readWriteLock.readLock();
readLock.lock();
// 进行读取操作
readLock.unlock();

Lock writeLock = readWriteLock.writeLock();
writeLock.lock();
// 进行写操作
writeLock.unlock();
```

也就是说，当使用 ReadWriteLock 的时候，并不是直接使用，而是获得其内部的读锁和写锁，然后分别调用lock/unlock。

### 8.2.2 读写锁实现的基本原理

从表面来看，ReadLock和WriteLock是两把锁，实际上它只是同一把锁的两个视图而已。什么叫两个视图呢？可以理解为是一把锁，线程分成两类：读线程和写线程。读线程和写线程之间不互斥（可以同时拿到这把锁），读线程之间不互斥，写线程之间互斥。

从下面的构造方法也可以看出，readerLock和writerLock实际共用同一个sync对象。sync对象同互斥锁一样，分为非公平和公平两种策略，并继承自AQS。

```
public ReentrantReadWriteLock() {
    this(false);
}

public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}
```

同互斥锁一样，读写锁也是用state变量来表示锁状态的。只是state变量在这里的含义和互斥锁完全不同。在内部类Sync中，对state变量进行了重新定义，如下所示：

```
abstract static class Sync extends AbstractQueuedSynchronizer {
    // ...
    static final int SHARED_SHIFT   = 16;
    static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
    static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
    static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
    // 持有读锁的线程的重入次数
    static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
    // 持有写锁的线程的重入次数
    static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
    // ...
}
```

也就是把 state 变量拆成两半，低16位，用来记录写锁。但同一时间既然只能有一个线程写，为什么还需要16位呢？这是因为一个写线程可能多次重入。例如，低16位的值等于5，表示一个写线程重入了5次。

高16位，用来“读”锁。例如，高16位的值等于5，既可以表示5个读线程都拿到了该锁；也可以表示一个读线程重入了5次。

为什么要把一个int类型变量拆成两半，而不是用两个int型变量分别表示读锁和写锁的状态呢？

这是因为无法用一次CAS 同时操作两个int变量，所以用了一个int型的高16位和低16位分别表示读锁和写锁的状态。

当state=0时，说明既没有线程持有读锁，也没有线程持有写锁；当state != 0时，要么有线程持有读锁，要么有线程持有写锁，两者不能同时成立，因为读和写互斥。这时再进一步通过sharedCount(state)和exclusiveCount(state)判断到底是读线程还是写线程持有了该锁。

#### 8.2.3 AQS的两对模板方法

下面介绍在ReentrantReadWriteLock的两个内部类ReadLock和WriteLock中，是如何使用state变量的。

```
public static class ReadLock implements Lock, java.io.Serializable {
    // ...
    public void lock() {
        sync.acquireShared(1);
    }

    public void unlock() {
        sync.releaseShared(1);
    }
    // ...
}

public static class WriteLock implements Lock, java.io.Serializable {
    // ...
    public void lock() {
        sync.acquire(1);
    }

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

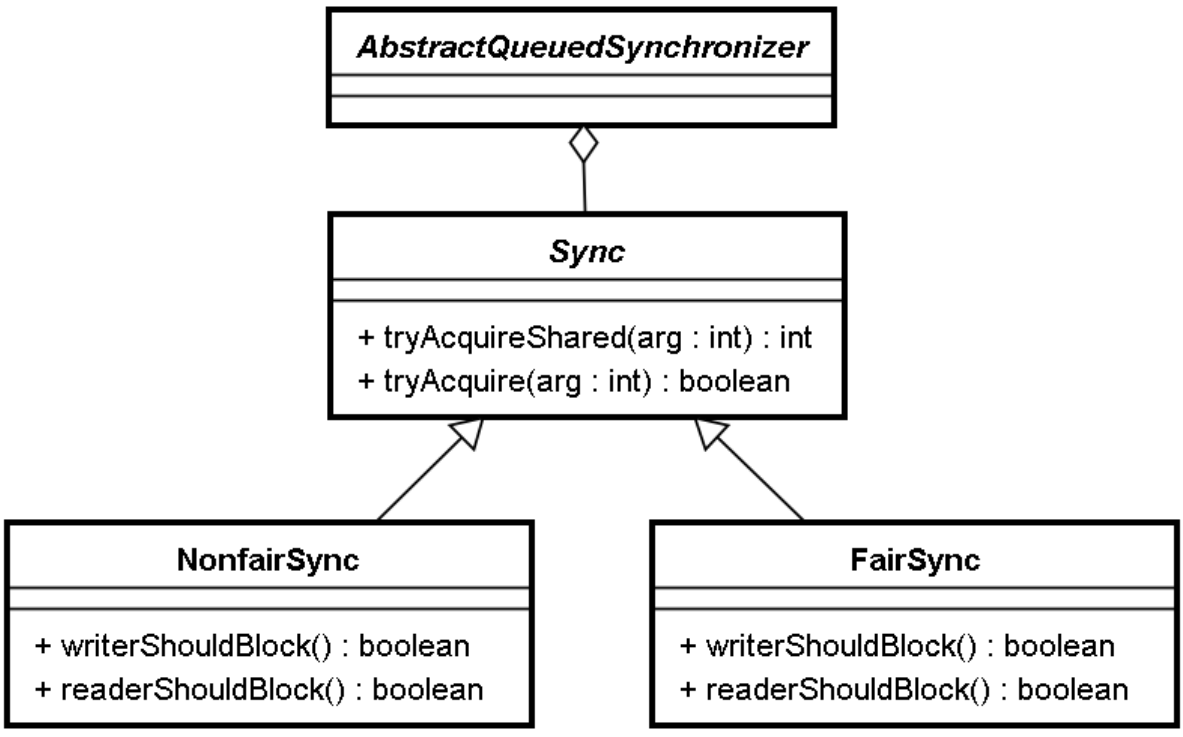
acquire/release、acquireShared/releaseShared 是AQS里面的两对模板方法。互斥锁和读写锁的写锁都是基于acquire/release模板方法来实现的。读写锁的读锁是基于acquireShared/releaseShared这对模板方法来实现的。这两对模板方法的代码如下：

```
public abstract class AbstractQueuedSynchronizer extends
AbstractOwnableSynchronizer implements java.io.Serializable {
    // ...
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&                 // tryAcquire方法由多个Sync子类实现
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }

    public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0) // tryAcquireShared方法由多个Sync子类实现
            doAcquireShared(arg);
```

```
    }

    public final boolean release(int arg) {
        if (tryRelease(arg)) {  // tryRelease方法由多个Sync子类实现
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }

    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {  // tryReleaseShared方法由多个Sync子类实现
            doReleaseShared();
            return true;
        }
        return false;
    }
    // ...
}
```

将读/写、公平/非公平进行排列组合，就有4种组合。如下图所示，上面的两个方法都是在Sync中实现的。Sync中的两个方法又是模板方法，在NonfairSync和FairSync中分别有实现。最终的对应关系如下：

1. 读锁的公平实现：Sync.tryAccquireShared()+FairSync中的两个重写的子方法。
2. 读锁的非公平实现：Sync.tryAccquireShared()+NonfairSync中的两个重写的子方法。
3. 写锁的公平实现：Sync.tryAccquire()+FairSync中的两个重写的子方法。
4. 写锁的非公平实现：Sync.tryAccquire()+NonfairSync中的两个重写的子方法。



```java
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = -8159625535654395037L;
    // 写线程抢锁的时候是否应该阻塞
    final boolean writerShouldBlock() {
        // 写线程在抢锁之前永远不被阻塞，非公平锁
        return false;
    }
    // 读线程抢锁的时候是否应该阻塞
    final boolean readerShouldBlock() {
        // 读线程抢锁的时候，当队列中第一个元素是写线程的时候要阻塞
        return apparentlyFirstQueuedIsExclusive();
    }
}

static final class FairSync extends Sync {
    private static final long serialVersionUID = -2274990926593161451L;
    // 写线程抢锁的时候是否应该阻塞
    final boolean writerShouldBlock() {
        // 写线程在抢锁之前，如果队列中有其他线程在排队，则阻塞。公平锁
        return hasQueuedPredecessors();
    }
    // 读线程抢锁的时候是否应该阻塞
    final boolean readerShouldBlock() {
        // 读线程在抢锁之前，如果队列中有其他线程在排队，阻塞。公平锁
        return hasQueuedPredecessors();
    }
}
```

对于公平，比较容易理解，不论是读锁，还是写锁，只要队列中有其他线程在排队（排队等读锁，或者排队等写锁），就不能直接去抢锁，要排在队列尾部。

对于非公平，读锁和写锁的实现策略略有差异。

写线程能抢锁，前提是state=0，只有在没有其他线程持有读锁或写锁的情况下，它才有机会去抢锁。或者state != 0，但那个持有写锁的线程是它自己，再次重入。写线程是非公平的，即writerShouldBlock()方法一直返回false。

对于读线程，假设当前线程被读线程持有，然后其他读线程还非公平地一直去抢，可能导致写线程永远拿不到锁，所以对于读线程的非公平，要做一些“约束”。当发现队列的第1个元素是写线程的时候，读线程也要阻塞，不能直接去抢。即偏向写线程。

#### 8.2.4 WriteLock公平vs.非公平实现

写锁是排他锁，实现策略类似于互斥锁。

**1.tryLock()实现分析**

ReentrantReadWriteLock.java | AbstractQueuedSynchronizer.java

```
1046    public boolean tryLock() {
1047        return sync.tryWriteLock();
1048    }
```

ReentrantReadWriteLock.java | AbstractQueuedSynchronizer.java

```
564     final boolean tryWriteLock() {
565         Thread current = Thread.currentThread();
566         int c = getState();
567         if (c != 0) {
568             int w = exclusiveCount(c);
569             if (w == 0 || current != getExclusiveOwnerThread())
570                 return false;
571             if (w == MAX_COUNT)
572                 throw new Error("Maximum lock count exceeded");
573         }
574         if (!compareAndSetState(c, update: c + 1))
575             return false;
576         setExclusiveOwnerThread(current);
577         return true;
578     }
```

当state不是0的时候，如果写线程获取锁的个数是0，或者写线程不是当前线程则返回抢锁失败

只要不是上面的情况，则通过CAS设置state的值如果设置成功，就将排他线程设置为当前线程，并返回true

lock()方法：

ReentrantReadWriteLock.java | AbstractQueuedSynchronizer.java

```
958     public void lock() {
959         sync.acquire( arg: 1);
960     }
```

ReentrantReadWriteLock.java | AbstractQueuedSynchronizer.java

```
1238    public final void acquire(int arg) {
1239        if (!tryAcquire(arg) &&
1240            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
1241            selfInterrupt();
1242    }
```

在互斥锁部分讲过了。

tryLock和lock方法不区分公平/非公平。

**2.unlock()实现分析**

ReentrantReadWriteLock.java AbstractQueuedSynchronizer.java

```
1146        public void unlock() {
1147            sync.release( arg: 1);
1148        }
```

ReentrantReadWriteLock.java AbstractQueuedSynchronizer.java

```
1301    public final boolean release(int arg) {
1302        if (tryRelease(arg)) {
1303            Node h = head;
1304            if (h != null && h.waitStatus != 0)
1305                unparkSuccessor(h);
1306            return true;
1307        }
1308        return false;
1309    }
```

unlock()方法不区分公平/非公平。

#### 8.2.5 ReadLock公平vs.非公平实现

读锁是共享锁，其实现策略和排他锁有很大的差异。

##### 1.tryLock()实现分析

ReentrantReadWriteLock.java

```
809         public boolean tryLock() {
810             return sync.tryReadLock();
811         }
```

```
final boolean tryReadLock() {
    // 获取当前线程
    Thread current = Thread.currentThread();
    for (;;) {
        // 获取state值
        int c = getState();
        // 如果是写线程占用锁或者当前线程不是排他线程，则抢锁失败
        if (exclusiveCount(c) != 0 &&
            getExclusiveOwnerThread() != current)
            return false;
        // 获取读锁state值
```

```
        int r = sharedCount(c);
        // 如果获取锁的值达到极限，则抛异常
        if (r == MAX_COUNT)
            throw new Error("Maximum lock count exceeded");

        // 使用CAS设置读线程锁state值
        if (compareAndSetState(c, c + SHARED_UNIT)) {
            // 如果r=0，则当前线程就是第一个读线程
            if (r == 0) {
                firstReader = current;
                // 读线程个数为1
                firstReaderHoldCount = 1;
            // 如果写线程是当前线程
            } else if (firstReader == current) {
                // 如果第一个读线程就是当前线程，表示读线程重入读锁
                firstReaderHoldCount++;
            } else {
                // 如果firstReader不是当前线程，则从ThreadLocal中获取当前线程的读锁个数，并设置当前线程持有的读锁个数
                HoldCounter rh = cachedHoldCounter;
                if (rh == null ||
                    rh.tid != LockSupport.getThreadId(current))
                    cachedHoldCounter = rh = readHolds.get();
                else if (rh.count == 0)
                    readHolds.set(rh);
                rh.count++;
            }
            return true;
        }
    }
}
```

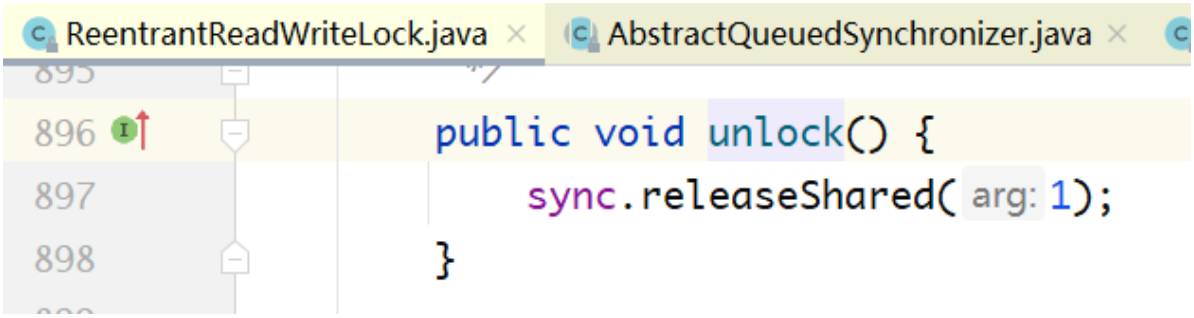
ReentrantReadWriteLock.java

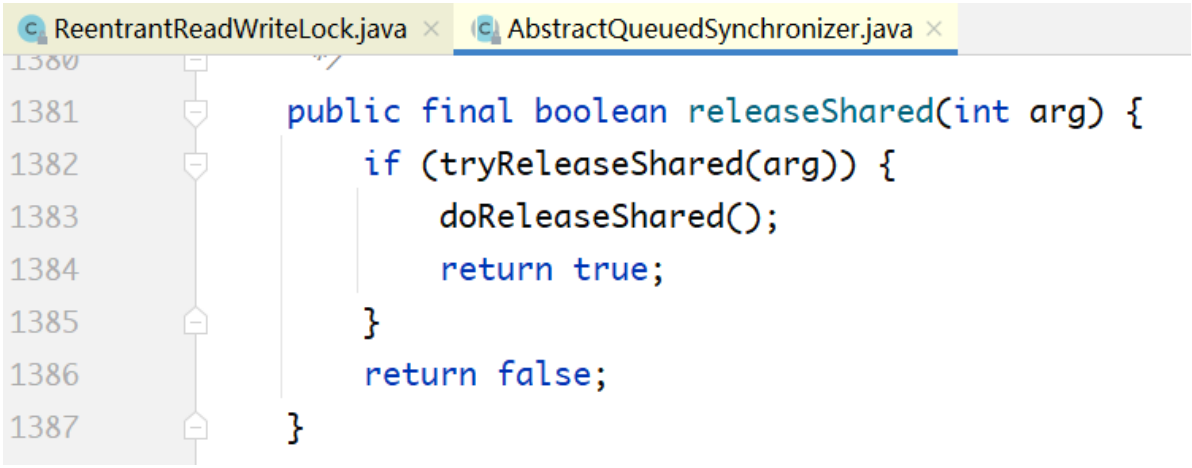
```
737     public void lock() {
738         sync.acquireShared( arg: 1);
739     }
```

ReentrantReadWriteLock.java | AbstractQueuedSynchronizer.java

```
1322    public final void acquireShared(int arg) {
1323        if (tryAcquireShared(arg) < 0)
1324            doAcquireShared(arg);
1325    }
```

**2.unlock()实现分析**




tryReleaseShared()的实现：

```
@ReservedStackAccess
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    // ...
    for (;;) {
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))
            // Releasing the read lock has no effect on readers,
            // but it may allow waiting writers to proceed if
            // both read and write locks are now free.
            return nextc == 0;
    }
}
```

因为读锁是共享锁，多个线程会同时持有读锁，所以对读锁的释放不能直接减1，而是需要通过一个for循环+CAS操作不断重试。这是tryReleaseShared和tryRelease的根本差异所在。

## 8.3 Condition

### 8.3.1 Condition与Lock的关系

Condition本身也是一个接口，其功能和wait/notify类似，如下所示：

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit) throws InterruptedException;
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    void awaitUninterruptibly();
    boolean awaitUntil(Date deadline) throws InterruptedException;
    void signal();
    void signalAll();
}
```

wait()/notify()必须和synchronized一起使用，Condition也必须和Lock一起使用。因此，在Lock的接口中，有一个与Condition相关的接口：

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    // 所有的Condition都是从Lock中构造出来的
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
}
```

### 8.3.2 Condition的使用场景

以ArrayBlockingQueue为例。如下所示为一个用数组实现的阻塞队列，执行put(...)操作的时候，队列满了，生产者线程被阻塞；执行take()操作的时候，队列为空，消费者线程被阻塞。

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
        implements BlockingQueue<E>, java.io.Serializable {
    //...
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    // 一把锁+两个条件
    final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;

    public ArrayBlockingQueue(int capacity, boolean fair) {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
        // 构造器中创建一把锁加两个条件
        lock = new ReentrantLock(fair);
        // 构造器中创建一把锁加两个条件
        notEmpty = lock.newCondition();
        // 构造器中创建一把锁加两个条件
        notFull =  lock.newCondition();
    }

    public void put(E e) throws InterruptedException {
        Objects.requireNonNull(e);
```

```
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                // 非满条件阻塞，队列容量已满
                notFull.await();
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }

    private void enqueue(E e) {
        // assert lock.isHeldByCurrentThread();
        // assert lock.getHoldCount() == 1;
        // assert items[putIndex] == null;
        final Object[] items = this.items;
        items[putIndex] = e;
        if (++putIndex == items.length) putIndex = 0;
        count++;
        // put数据结束，通知消费者非空条件
        notEmpty.signal();
    }

    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                // 阻塞于非空条件，队列元素个数为0，无法消费
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
    }

    private E dequeue() {
        // assert lock.isHeldByCurrentThread();
        // assert lock.getHoldCount() == 1;
        // assert items[takeIndex] != null;
        final Object[] items = this.items;
        @SuppressWarnings("unchecked")
        E e = (E) items[takeIndex];
        items[takeIndex] = null;
        if (++takeIndex == items.length) takeIndex = 0;
        count--;
        if (itrs != null)
            itrs.elementDequeued();
        // 消费成功，通知非满条件，队列中有空间，可以生产元素了。
        notFull.signal();
        return e;
    }
    // ...
}
```

#### 8.3.3 Condition实现原理

可以发现，Condition的使用很方便，避免了wait/notify的生产者通知生产者、消费者通知消费者的问题。具体实现如下：

由于Condition必须和Lock一起使用，所以Condition的实现也是Lock的一部分。首先查看互斥锁和读写锁中Condition的构造方法：

```
public class ReentrantLock implements Lock, java.io.Serializable {
    // ...
    public Condition newCondition() {
        return sync.newCondition();
    }
}

public class ReentrantReadWriteLock
        implements ReadWriteLock, java.io.Serializable {
    // ...
    private final ReentrantReadWriteLock.ReadLock readerLock;
    private final ReentrantReadWriteLock.WriteLock writerLock;
    // ...
    public static class ReadLock implements Lock, java.io.Serializable {
        // 读锁不支持Condition
        public Condition newCondition() {
            // 抛异常
            throw new UnsupportedOperationException();
        }
    }

    public static class WriteLock implements Lock, java.io.Serializable {
        // ...
        public Condition newCondition() {
            return sync.newCondition();
        }
        // ...
    }
    // ...
}
```

首先，读写锁中的 ReadLock 是不支持 Condition 的，读写锁的写锁和互斥锁都支持Condition。虽然它们各自调用的是自己的内部类Sync，但内部类Sync都继承自AQS。因此，上面的代码sync.newCondition最终都调用了AQS中的newCondition：

```
public abstract class AbstractQueuedSynchronizer extends
AbstractOwnableSynchronizer
    implements java.io.Serializable {

    public class ConditionObject implements Condition, java.io.Serializable
{
        // Condition的所有实现，都在ConditionObject类中
    }
}

abstract static class Sync extends AbstractQueuedSynchronizer {
   final ConditionObject newCondition() {
       return new ConditionObject();
   }
}
```

每一个Condition对象上面，都阻塞了多个线程。因此，在ConditionObject内部也有一个双向链表组成的队列，如下所示：

```
public class ConditionObject implements Condition, java.io.Serializable {
    private transient Node firstWaiter;
    private transient Node lastWaiter;
}

static final class Node {
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;
    Node nextWaiter;
}
```

下面来看一下在await()/notify()方法中，是如何使用这个队列的。

#### 8.3.4 await()实现分析

```
public final void await() throws InterruptedException {
    // 刚要执行await()操作，收到中断信号，抛异常
    if (Thread.interrupted())
        throw new InterruptedException();
    // 加入Condition的等待队列
    Node node = addConditionWaiter();
    // 阻塞在Condition之前必须先释放锁，否则会死锁
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        // 阻塞当前线程
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // 重新获取锁
```

```
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        // 被中断唤醒，抛中断异常
        reportInterruptAfterWait(interruptMode);
}
```

关于await，有几个关键点要说明：

1. 线程调用 await()的时候，肯定已经先拿到了锁。所以，在 addConditionWaiter()内部，对这个双向链表的操作不需要执行CAS操作，线程天生是安全的，代码如下：

```
private Node addConditionWaiter() {
    // ...
    Node t = lastWaiter;
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    if (t == null)
        firstWaiter = node;
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```

2. 在线程执行wait操作之前，必须先释放锁。也就是fullyRelease(node)，否则会发生死锁。这个和wait/notify与synchronized的配合机制一样。
3. 线程从wait中被唤醒后，必须用acquireQueued(node, savedState)方法重新拿锁。
4. checkInterruptWhileWaiting(node)代码在park(this)代码之后，是为了检测在park期间是否收到过中断信号。当线程从park中醒来时，有两种可能：一种是其他线程调用了unpark，另一种是收到中断信号。这里的await()方法是可以响应中断的，所以当发现自己是被中断唤醒的，而不是被unpark唤醒的时，会直接退出while循环，await()方法也会返回。
5. isOnSyncQueue(node)用于判断该Node是否在AQS的同步队列里面。初始的时候，Node只在Condition的队列里，而不在AQS的队列里。但执行notity操作的时候，会放进AQS的同步队列。

#### 8.3.5 awaitUninterruptibly()实现分析

与await()不同，awaitUninterruptibly()不会响应中断，其方法的定义中不会有中断异常抛出，下面分析其实现和await()的区别。

AbstractQueuedSynchronizer.java

```
2013    public final void awaitUninterruptibly() {
2014        Node node = addConditionWaiter();
2015        int savedState = fullyRelease(node);
2016        boolean interrupted = false;
2017        while (!isOnSyncQueue(node)) {
2018            LockSupport.park( blocker: this);
2019            if (Thread.interrupted())   当线程唤醒后，如果被中断过，仅记录，不处理
2020                interrupted = true;     继续进行while循环。
2021        }
2022        if (acquireQueued(node, savedState) || interrupted)
2023            selfInterrupt();
2024    }
```

可以看出，整体代码和 await()类似，区别在于收到异常后，不会抛出异常，而是继续执行while循环。

### 8.3.6 notify()实现分析

```
1  public final void signal() {
2      // 只有持有锁的线程，才有资格调用signal()方法
3      if (!isHeldExclusively())
4          throw new IllegalMonitorStateException();
5      Node first = firstWaiter;
6      if (first != null)
7          // 发起通知
8          doSignal(first);
9  }
10
11 // 唤醒队列中的第1个线程
12 private void doSignal(Node first) {
13     do {
14         if ( (firstWaiter = first.nextWaiter) == null)
15             lastWaiter = null;
16         first.nextWaiter = null;
17     } while (!transferForSignal(first) && (first = firstWaiter) != null);
18 }
19
20 final boolean transferForSignal(Node node) {
21     if (!node.compareAndSetWaitStatus(Node.CONDITION, 0))
22         return false;
23     // 先把Node放入互斥锁的同步队列中，再调用unpark方法
24     Node p = enq(node);
25     int ws = p.waitStatus;
26     if (ws > 0 || !p.compareAndSetWaitStatus(ws, Node.SIGNAL))
27         LockSupport.unpark(node.thread);
28     return true;
29 }
```

同 await()一样，在调用 notify()的时候，必须先拿到锁（否则就会抛出上面的异常），是因为前面执行await()的时候，把锁释放了。

然后，从队列中取出firstWaiter，唤醒它。在通过调用unpark唤醒它之前，先用enq(node)方法把这个Node放入AQS的锁对应的阻塞队列中。也正因为如此，才有了await()方法里面的判断条件：

while( ! isOnSyncQueue(node))

这个判断条件满足，说明await线程不是被中断，而是被unpark唤醒的。

notifyAll()与此类似。

## 8.4 StampedLock

### 8.4.1 为什么引入StampedLock

StampedLock是在JDK8中新增的，有了读写锁，为什么还要引入StampedLock呢?

| 锁 | 并发度 |
| --- | --- |
| ReentrantLock | 读读互斥，读写互斥，写写互斥 |
| ReentrantReadWriteLock | 读读不互斥，读写互斥，写写互斥 |
| StampedLock | 读读不互斥，读写不互斥，写写互斥 |

可以看到，从ReentrantLock到StampedLock，并发度依次提高。

另一方面，因为ReentrantReadWriteLock采用的是“悲观读”的策略，当第一个读线程拿到锁之后，第二个、第三个读线程还可以拿到锁，使得写线程一直拿不到锁，可能导致写线程“饿死”。虽然在其公平或非公平的实现中，都尽量避免这种情形，但还有可能发生。

StampedLock引入了“乐观读”策略，读的时候不加读锁，读出来发现数据被修改了，再升级为“悲观读”，相当于降低了“读”的地位，把抢锁的天平往“写”的一方倾斜了一下，避免写线程被饿死。

### 8.4.2 使用场景

在剖析其原理之前，下面先以官方的一个例子来看一下StampedLock如何使用。

```
class Point {
    private double x, y;
    private final StampedLock sl = new StampedLock();
    // 多个线程调用该方法，修改x和y的值
    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    // 多个线程调用该方法，求距离
    double distenceFromOrigin() {
```

```
        // 使用“乐观读”
        long stamp = sl.tryOptimisticRead();
        // 将共享变量拷贝到线程栈
        double currentX = x, currentY = y;
        // 读期间有其他线程修改数据
        if (!sl.validate(stamp)) {
            // 读到的是脏数据，丢弃。
            // 重新使用“悲观读”
            stamp = sl.readLock();
            try {
                currentX = x;
                currentY = y;
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
}
```

如上面代码所示，有一个Point类，多个线程调用move()方法，修改坐标；还有多个线程调用distanceFromOrigin()方法，求距离。

首先，执行move操作的时候，要加写锁。这个用法和ReadWriteLock的用法没有区别，写操作和写操作也是互斥的。

关键在于读的时候，用了一个“乐观读”sl.tryOptimisticRead()，相当于在读之前给数据的状态做了一个“快照”。然后，把数据拷贝到内存里面，在用之前，再比对一次版本号。如果版本号变了，则说明在读的期间有其他线程修改了数据。读出来的数据废弃，重新获取读锁。关键代码就是下面这三行：

```
// 读取之前，获取数据的版本号
long stamp = sl.tryOptimisticRead();
// 读：将一份数据拷贝到线程的栈内存中
double currentX = x, currentY = y;
// 读取之后，对比读之前的版本号和当前的版本号，判断数据是否可用。
// 根据stamp判断在读取数据和使用数据期间，有没有其他线程修改数据
if (!sl.validate(stamp)) {
    // ...
}
```

**要说明的是，这三行关键代码对顺序非常敏感，不能有重排序。因为 state 变量已经是volatile，所以可以禁止重排序，但stamp并不是volatile的。为此，在validate(stamp)方法里面插入内存屏障。**

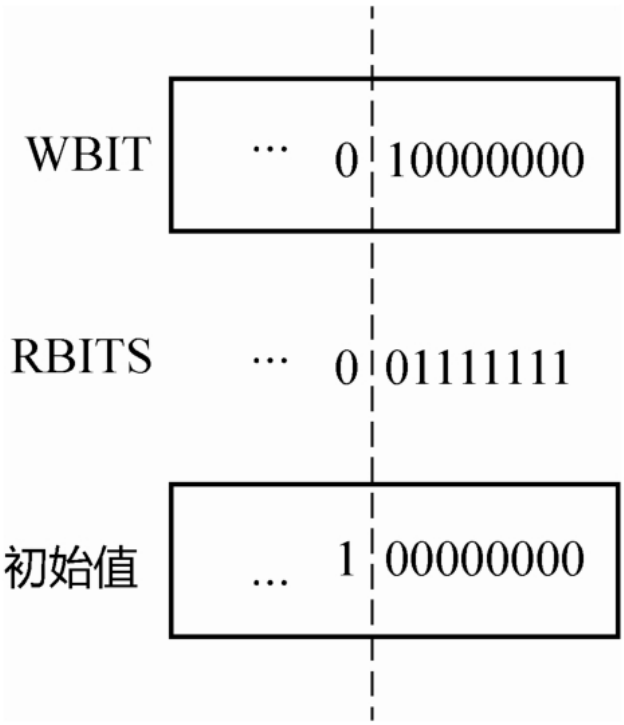
```
public boolean validate(long stamp) {
    VarHandle.acquireFence();
    return (stamp & SBITS) == (state & SBITS);
}
```

### 8.4.3 “乐观读”的实现原理

首先，StampedLock是一个读写锁，因此也会像读写锁那样，把一个state变量分成两半，分别表示读锁和写锁的状态。同时，它还需要一个数据的version。但是，一次CAS没有办法操作两个变量，所以这个state变量本身同时也表示了数据的version。下面先分析state变量。

```
public class StampedLock implements java.io.Serializable {
    private static final int LG_READERS = 7;

    private static final long RUNIT = 1L;
    private static final long WBIT = 1L << LG_READERS; // 第8位表示写锁
    private static final long RBITS = WBIT - 1L;  // 最低的7位表示读锁
    private static final long RFULL = RBITS - 1L; // 读锁的数目
    private static final long ABITS = RBITS | WBIT; // 读锁和写锁状态合二为一
    private static final long SBITS = ~RBITS;

    //
    private static final long ORIGIN = WBIT << 1;  // state的初始值
    private transient volatile long state;
    // ...
}
```

如下图：用最低的8位表示读和写的状态，其中第8位表示写锁的状态，最低的7位表示读锁的状态。因为写锁只有一个bit位，所以写锁是不可重入的。




初始值不为0，而是把WBIT 向左移动了一位，也就是上面的ORIGIN 常量，构造方法如下所示。

StampedLock.java

```
434    public StampedLock() {
435        state = ORIGIN;
436    }
```

为什么state的初始值不设为0呢？看乐观锁的实现：

```
public long tryOptimisticRead() {
    long s;
    return (((s = state) & WBIT) == 0L) ? (s & SBITS) : 0L;
}

public boolean validate(long stamp) {
    VarHandle.acquireFence();
    return (stamp & SBITS) == (state & SBITS); // 当stamp=0时，validate永远返回
false
}
```

上面两个方法必须结合起来看：当state&WBIT != 0的时候，说明有线程持有写锁，上面的tryOptimisticRead会永远返回0。这样，再调用validate（stamp），也就是validate（0）也会永远返回false。这正是我们想要的逻辑：当有线程持有写锁的时候，validate永远返回false，无论写线程是否释放了写锁。因为无论是否释放了（state回到初始值）写锁，state值都不为0，所以validate（0）永远为false。

为什么上面的validate(...)方法不直接比较stamp=state，而要比较state&SBITS=state&SBITS 呢?

因为读锁和读锁是不互斥的!

所以，即使在“乐观读”的时候，state 值被修改了，但如果它改的是第7位，validate(...)还是会返回true。

另外要说明的一点是，上面使用了内存屏障VarHandle.acquireFence();，是因为在这行代码的下一行里面的stamp、SBITS变量不是volatile的，由此可以禁止其和前面的currentX=X，currentY=Y进行重排序。

通过上面的分析，可以发现state的设计非常巧妙。只通过一个变量，既实现了读锁、写锁的状态记录，还实现了数据的版本号的记录。

#### 8.4.4 悲观读/写：“阻塞”与“自旋”策略实现差异

同ReadWriteLock一样，StampedLock也要进行悲观的读锁和写锁操作。不过，它不是基于AQS实现的，而是内部重新实现了一个**阻塞队列**。如下所示。

```
public class StampedLock implements java.io.Serializable {
    // ...
    static final class WNode {
        volatile WNode prev;
        volatile WNode next;
        volatile WNode cowait;
```

```
        volatile Thread thread;
        volatile int status;  // 取值：0，WAITING或CANCELLED
        final int mode;    // 取值：RMODE或WMODE
        WNode(int m, WNode p) {
            mode = m;
            prev = p;
        }
    }
    // ...

    private transient volatile WNode whead;
    private transient volatile WNode wtail;
    // ...
}
```

这个阻塞队列和 AQS 里面的很像。

刚开始的时候，whead=wtail=NULL，然后初始化，建一个空节点，whead和wtail都指向这个空节点，之后往里面加入一个个读线程或写线程节点。

但基于这个阻塞队列实现的锁的调度策略和AQS很不一样，也就是“自旋”。

在AQS里面，当一个线程CAS state失败之后，会立即加入阻塞队列，并且进入阻塞状态。

但在StampedLock中，CAS state失败之后，会不断自旋，自旋足够多的次数之后，如果还拿不到锁，才进入阻塞状态。

为此，根据CPU的核数，定义了自旋次数的常量值。如果是单核的CPU，肯定不能自旋，在多核情况下，才采用自旋策略。

```
private static final int NCPU = Runtime.getRuntime().availableProcessors();
// 自旋的次数，超过这个数字，进入阻塞。
private static final int SPINS = (NCPU > 1) ? 1 << 6 : 0;
```

下面以写锁的加锁，也就是StampedLock的writeLock()方法为例，来看一下自旋的实现。

```
public long writeLock() {
    long next;
    return ((next = tryWriteLock()) != 0L) ? next : acquireWrite(false, 0L);
}

public long tryWriteLock() {
    long s;
    return (((s = state) & ABITS) == 0L) ? tryWriteLock(s) : 0L;
}
```

如上面代码所示，当state&ABITS==0的时候，说明既没有线程持有读锁，也没有线程持有写锁，此时当前线程才有资格通过CAS操作state。若操作不成功，则调用acquireWrite()方法进入阻塞队列，并进行自旋，这个方法是整个加锁操作的核心，代码如下：

```
private long acquireWrite(boolean interruptible, long deadline) {
    WNode node = null, p;
```

```
    for (int spins = -1;;) { // 入列时自旋
        long m, s, ns;
        if ((m = (s = state) & ABITS) == 0L) {
            if ((ns = tryWriteLock(s)) != 0L)
                return ns;  // 自旋的时候获取到锁，返回
        }
        else if (spins < 0)
            // 计算自旋值
            spins = (m == WBIT && wtail == whead) ? SPINS : 0;
        else if (spins > 0) {
            --spins; // 每次自旋获取锁，spins值减一
            Thread.onSpinWait();
        }
        else if ((p = wtail) == null) { // 如果尾部节点是null，初始化队列
            WNode hd = new WNode(WMODE, null);
            // 头部和尾部指向一个节点
            if (WHEAD.weakCompareAndSet(this, null, hd))
                wtail = hd;
        }
        else if (node == null)
            node = new WNode(WMODE, p);
        else if (node.prev != p)
            // p节点作为前置节点
            node.prev = p;
        // for循环唯一的break，成功将节点node添加到队列尾部，才会退出for循环
        else if (WTAIL.weakCompareAndSet(this, p, node)) {
            // 设置p的后置节点为node
            p.next = node;
            break;
        }
    }

    boolean wasInterrupted = false;
    for (int spins = -1;;) {
        WNode h, np, pp; int ps;
        if ((h = whead) == p) {
            if (spins < 0)
                spins = HEAD_SPINS;
            else if (spins < MAX_HEAD_SPINS)
                spins <<= 1;
            for (int k = spins; k > 0; --k) { // spin at head
                long s, ns;
                if (((s = state) & ABITS) == 0L) {
                    if ((ns = tryWriteLock(s)) != 0L) {
                        whead = node;
                        node.prev = null;
                        if (wasInterrupted)
                            Thread.currentThread().interrupt();
                        return ns;
                    }
                }
                else
                    Thread.onSpinWait();
            }
        }
        else if (h != null) { // 唤醒读取的线程
            WNode c; Thread w;
            while ((c = h.cowait) != null) {
```

```
                if (WCOWAIT.weakCompareAndSet(h, c, c.cowait) &&
                    (w = c.thread) != null)
                    LockSupport.unpark(w);
            }
        }
        if (whead == h) {
            if ((np = node.prev) != p) {
                if (np != null)
                    (p = np).next = node;   // stale
            }
            else if ((ps = p.status) == 0)
                WSTATUS.compareAndSet(p, 0, WAITING);
            else if (ps == CANCELLED) {
                if ((pp = p.prev) != null) {
                    node.prev = pp;
                    pp.next = node;
                }
            }
            else {
                long time; // 0 argument to park means no timeout
                if (deadline == 0L)
                    time = 0L;
                else if ((time = deadline - System.nanoTime()) <= 0L)
                    return cancelWaiter(node, node, false);
                Thread wt = Thread.currentThread();
                node.thread = wt;
                if (p.status < 0 && (p != h || (state & ABITS) != 0L) &&
                    whead == h && node.prev == p) {
                    if (time == 0L)
                        // 阻塞，直到被唤醒
                        LockSupport.park(this);
                    else
                        // 计时阻塞
                        LockSupport.parkNanos(this, time);
                }
                node.thread = null;
                if (Thread.interrupted()) {
                    if (interruptible)
                        // 如果被中断了，则取消等待
                        return cancelWaiter(node, node, true);
                    wasInterrupted = true;
                }
            }
        }
    }
}


```

整个acquireWrite(...)方法是两个大的for循环，内部实现了非常复杂的自旋策略。在第一个大的for循环里面，目的就是把该Node加入队列的尾部，一边加入，一边通过CAS操作尝试获得锁。如果获得了，整个方法就会返回；如果不能获得锁，会一直自旋，直到加入队列尾部。

在第二个大的for循环里，也就是该Node已经在队列尾部了。这个时候，如果发现自己刚好也在队列头部，说明队列中除了空的Head节点，就是当前线程了。此时，再进行新一轮的自旋，直到达到MAX_HEAD_SPINS次数，然后进入阻塞。这里有一个关键点要说明：当release(...)方法被调用之后，会唤醒队列头部的第1个元素，此时会执行第二个大的for循环里面的逻辑，也就是接着for循环里面park()方法后面的代码往下执行。

另外一个不同于AQS的阻塞队列的地方是，在每个WNode里面有一个cowait指针，用于串联起所有的读线程。例如，队列尾部阻塞的是一个读线程 1，现在又来了读线程 2、3，那么会通过cowait指针，把1、2、3串联起来。1被唤醒之后，2、3也随之一起被唤醒，因为读和读之间不互斥。

明白加锁的自旋策略后，下面来看锁的释放操作。和读写锁的实现类似，也是做了两件事情：一是把state变量置回原位，二是唤醒阻塞队列中的第一个节点。

StampedLock.java

```
677     @ReservedStackAccess
678     public void unlockWrite(long stamp) {
679         if (state != stamp || (stamp & WBIT) == 0L)
680             throw new IllegalMonitorStateException();
681         unlockWriteInternal(stamp);
682     }
```

StampedLock.java

```
660
661     private long unlockWriteInternal(long s) {
662         long next; WNode h;
663         STATE.setVolatile(this, next = unlockWriteState(s));
664         if ((h = whead) != null && h.status != 0)
665             release(h);
666         return next;
667     }
```

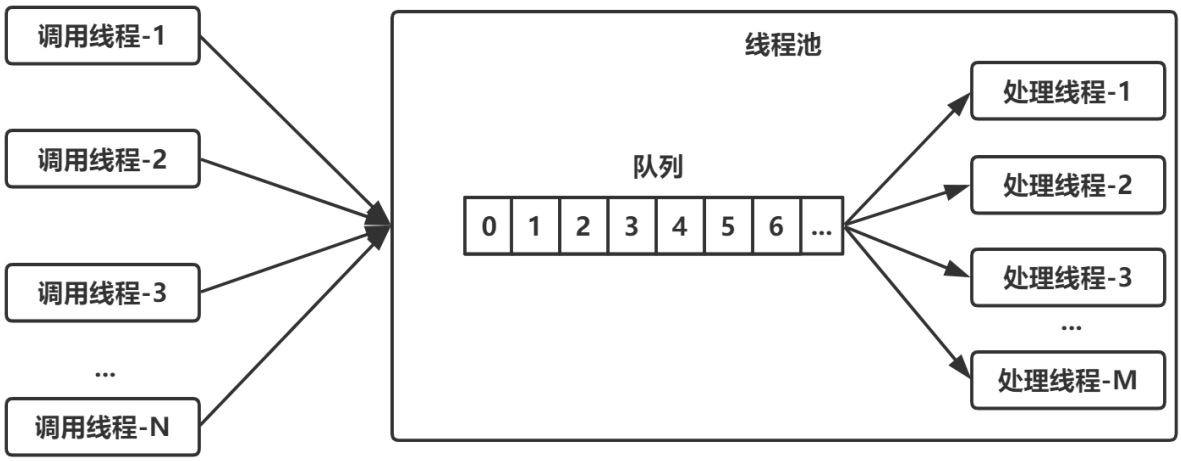
StampedLock.java

```
1207     */
1208     private void release(WNode h) {
1209         if (h != null) {
1210             WNode q; Thread w;
1211             WSTATUS.compareAndSet( ...args: h, WAITING, 0);
1212             if ((q = h.next) == null || q.status == CANCELLED) {
1213                 for (WNode t = wtail; t != null && t != h; t = t.prev)
1214                     if (t.status <= 0)
1215                         q = t;
1216             }
1217             if (q != null && (w = q.thread) != null)
1218                 LockSupport.unpark(w);
1219         }
1220     }
```

# 第三部分：线程池与Future

## 9 线程池的实现原理

下图所示为线程池的实现原理：调用方不断地向线程池中提交任务；线程池中有一组线程，不断地从队列中取任务，这是一个典型的生产者—消费者模型。




要实现这样一个线程池，有几个问题需要考虑：

1. 队列设置多长？如果是无界的，调用方不断地往队列中放任务，可能导致内存耗尽。如果是有界的，当队列满了之后，调用方如何处理？
2. 线程池中的线程个数是固定的，还是动态变化的？
3. 每次提交新任务，是放入队列？还是开新线程？
4. 当没有任务的时候，线程是睡眠一小段时间？还是进入阻塞？如果进入阻塞，如何唤醒？

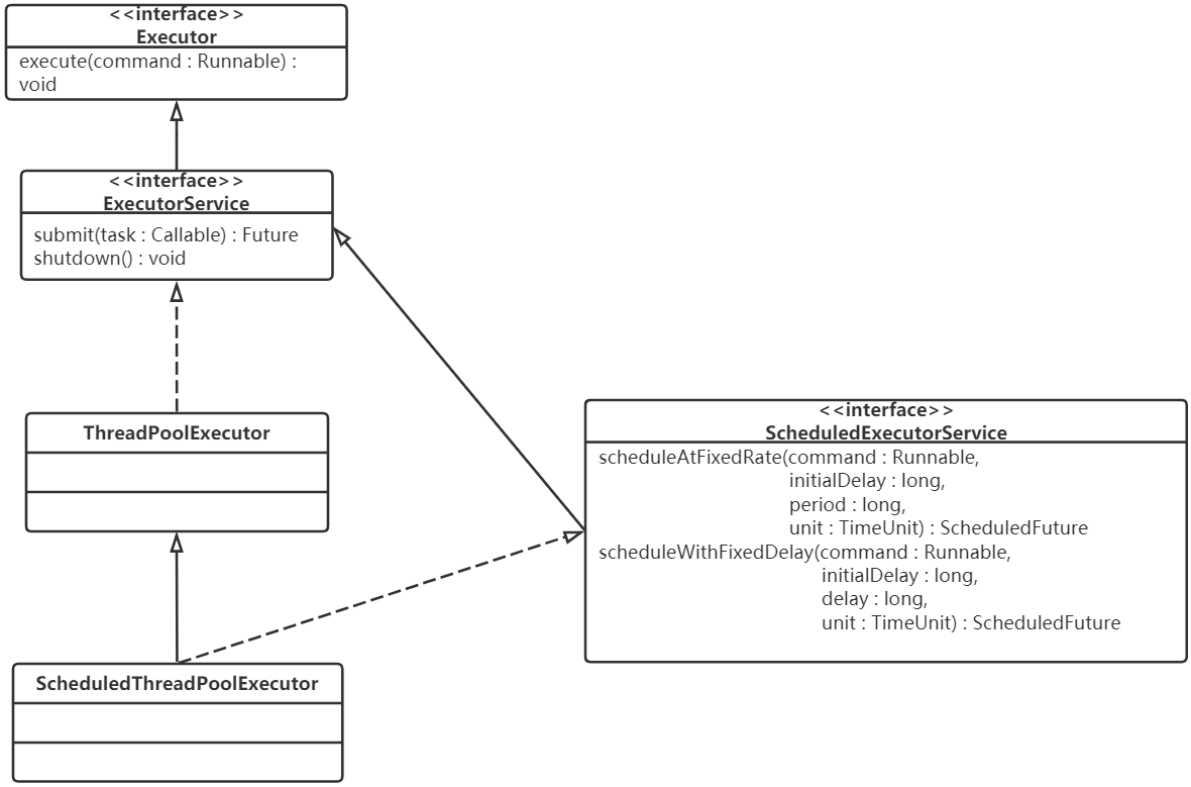
针对问题4，有3种做法：

1. 不使用阻塞队列，只使用一般的线程安全的队列，也无阻塞/唤醒机制。当队列为空时，线程池中的线程只能睡眠一会儿，然后醒来去看队列中有没有新任务到来，如此不断轮询。
2. 不使用阻塞队列，但在队列外部、线程池内部实现了阻塞/唤醒机制。
3. 使用阻塞队列。

很显然，做法3最完善，既避免了线程池内部自己实现阻塞/唤醒机制的麻烦，也避免了做法1的睡眠/轮询带来的资源消耗和延迟。正因为如此，接下来要讲的ThreadPoolExector/ScheduledThreadPoolExecutor都是基于阻塞队列来实现的，而不是一般的队列，至此，各式各样的阻塞队列就要派上用场了。

## 10 线程池的类继承体系

线程池的类继承体系如下图所示：




在这里，有两个核心的类：`ThreadPoolExector`和`ScheduledThreadPoolExecutor`，后者不仅可以执行某个任务，还可以周期性地执行任务。

向线程池中提交的每个任务，都必须实现`Runnable`接口，通过最上面的`Executor`接口中的`execute(Runnable command)`向线程池提交任务。

然后，在`ExecutorService`中，定义了线程池的关闭接口`shutdown()`，还定义了可以有返回值的任务，也就是`Callable`，后面会详细介绍。

# 11 ThreadPoolExecutor

## 11.1 核心数据结构

基于线程池的实现原理，下面看一下ThreadPoolExector的核心数据结构。

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    //...
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    // 存放任务的阻塞队列
    private final BlockingQueue<Runnable> workQueue;
    // 对线程池内部各种变量进行互斥访问控制
    private final ReentrantLock mainLock = new ReentrantLock();
    // 线程集合
    private final HashSet<Worker> workers = new HashSet<Worker>();
    //...
}
```

每一个线程是一个Worker对象。Worker是ThreadPoolExector的内部类，核心数据结构如下：

```java
private final class Worker extends AbstractQueuedSynchronizer implements
Runnable {
    // ...
    final Thread thread; // Worker封装的线程
    Runnable firstTask; // Worker接收到的第1个任务
    volatile long completedTasks; // Worker执行完毕的任务个数
    // ...
}
```

由定义会发现，Worker继承于AQS，也就是说Worker本身就是一把锁。这把锁有什么用处呢？用于线程池的关闭、线程执行任务的过程中。

## 11.2 核心配置参数解释

ThreadPoolExecutor在其构造方法中提供了几个核心配置参数，来配置不同策略的线程池。

ThreadPoolExecutor.java

```java
1282    public ThreadPoolExecutor(int corePoolSize,
1283                              int maximumPoolSize,
1284                              long keepAliveTime,
1285                              TimeUnit unit,
1286                              BlockingQueue<Runnable> workQueue,
1287                              ThreadFactory threadFactory,
1288                              RejectedExecutionHandler handler) {
1289        if (corePoolSize < 0 ||
1290            maximumPoolSize <= 0 ||
1291            maximumPoolSize < corePoolSize ||
1292            keepAliveTime < 0)
1293            throw new IllegalArgumentException();
1294        if (workQueue == null || threadFactory == null || handler == null)
1295            throw new NullPointerException();
1296        this.corePoolSize = corePoolSize;
1297        this.maximumPoolSize = maximumPoolSize;
1298        this.workQueue = workQueue;
1299        this.keepAliveTime = unit.toNanos(keepAliveTime);
1300        this.threadFactory = threadFactory;
1301        this.handler = handler;
1302    }
```

上面的各个参数，解释如下：

1. corePoolSize：在线程池中始终维护的线程个数。
2. maxPoolSize：在corePooSize已满、队列也满的情况下，扩充线程至此值。
3. keepAliveTime/TimeUnit：maxPoolSize 中的空闲线程，销毁所需要的时间，总线程数收缩回corePoolSize。
4. blockingQueue：线程池所用的队列类型。
5. threadFactory：线程创建工厂，可以自定义，有默认值`Executors.defaultThreadFactory()`。
6. RejectedExecutionHandler：corePoolSize已满，队列已满，maxPoolSize 已满，最后的拒绝策略。

下面来看这6个配置参数在任务的提交过程中是怎么运作的。在每次往线程池中提交任务的时候，有如下的处理流程：

步骤一：判断当前线程数是否大于或等于corePoolSize。如果小于，则新建线程执行；如果大于，则进入步骤二。

步骤二：判断队列是否已满。如未满，则放入；如已满，则进入步骤三。

步骤三：判断当前线程数是否大于或等于maxPoolSize。如果小于，则新建线程执行；如果大于，则进入步骤四。

步骤四：根据拒绝策略，拒绝任务。

总结一下：首先判断corePoolSize，其次判断blockingQueue是否已满，接着判断maxPoolSize，最后使用拒绝策略。

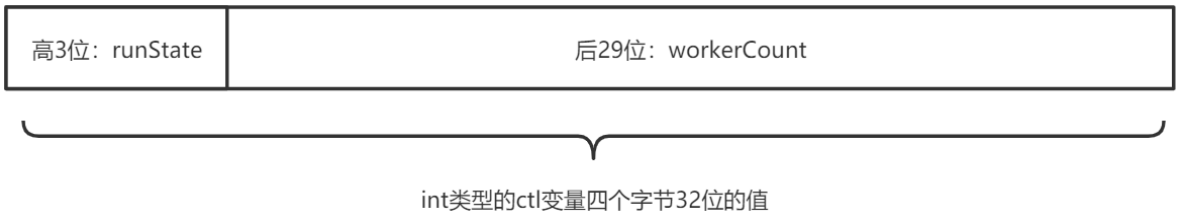
很显然，基于这种流程，如果队列是无界的，将永远没有机会走到步骤三，也即maxPoolSize没有使用，也一定不会走到步骤四。

## 11.3 线程池的优雅关闭

线程池的关闭，较之线程的关闭更加复杂。当关闭一个线程池的时候，有的线程还正在执行某个任务，有的调用者正在向线程池提交任务，并且队列中可能还有未执行的任务。因此，关闭过程不可能是瞬时的，而是需要一个平滑的过渡，这就涉及线程池的完整生命周期管理。

### 1.线程池的生命周期

在JDK 7中，把线程数量（workerCount）和线程池状态（runState）这两个变量打包存储在一个字段里面，即ctl变量。如下图所示，最高的3位存储线程池状态，其余29位存储线程个数。而在JDK 6中，这两个变量是分开存储的。




ThreadPoolExecutor.java

```
380     private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, wc: 0));
381     private static final int COUNT_BITS = Integer.SIZE - 3;
382     private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;
383
384     // runState is stored in the high-order bits
385     private static final int RUNNING    = -1 << COUNT_BITS;
386     private static final int SHUTDOWN   =  0 << COUNT_BITS;
387     private static final int STOP       =  1 << COUNT_BITS;
388     private static final int TIDYING    =  2 << COUNT_BITS;
389     private static final int TERMINATED =  3 << COUNT_BITS;
390
391     // Packing and unpacking ctl
392     private static int runStateOf(int c)     { return c & ~COUNT_MASK; }
393     private static int workerCountOf(int c)  { return c & COUNT_MASK; }
394     private static int ctlOf(int rs, int wc) { return rs | wc; }
395
```

初始线程池状态为RUNNING，线程数为0

高3位表示线程池状态

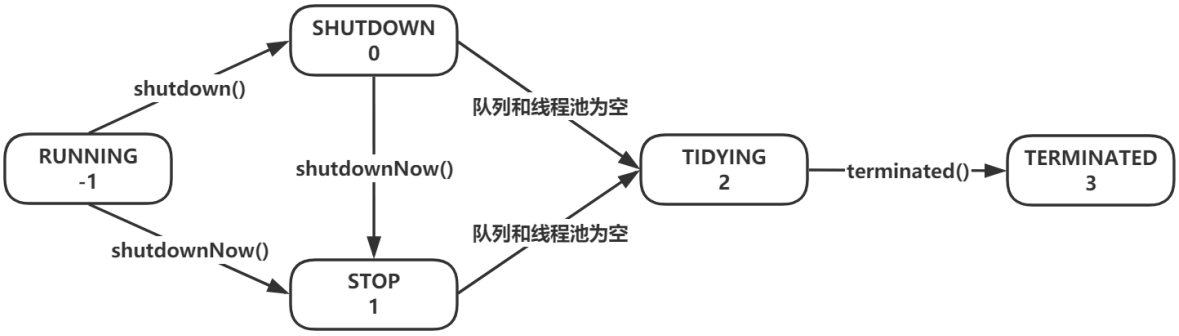
线程池的五种状态

从ctl中解析出runState和workerCount

将runState和workerCount组成一个变量

由上面的代码可以看到，ctl变量被拆成两半，最高的3位用来表示线程池的状态，低的29位表示线程的个数。线程池的状态有五种，分别是RUNNING、SHUTDOWN、STOP、TIDYING和TERMINATED。

下面分析状态之间的迁移过程，如图所示：




线程池有两个关闭方法，shutdown()和shutdownNow()，这两个方法会让线程池切换到不同的状态。在队列为空，线程池也为空之后，进入TIDYING 状态；最后执行一个钩子方法terminated()，进入TERMINATED状态，线程池才真正关闭。

这里的状态迁移有一个非常关键的特征：从小到大迁移，-1，0，1，2，3，只会从小的状态值往大的状态值迁移，不会逆向迁移。例如，当线程池的状态在TIDYING=2时，接下来只可能迁移到TERMINATED=3，不可能迁移回STOP=1或者其他状态。

除 terminated()之外，线程池还提供了其他几个钩子方法，这些方法的实现都是空的。如果想实现自己的线程池，可以重写这几个方法：

```
protected void beforeExecute(Thread t, Runnable r) { }
protected void afterExecute(Runnable r, Throwable t) { }
protected void terminated() { }
```

**2.正确关闭线程池的步骤**

关闭线程池的过程为：在调用 shutdown()或者shutdownNow()之后，线程池并不会立即关闭，接下来需要调用 awaitTermination() 来等待线程池关闭。关闭线程池的正确步骤如下：

```
// executor.shutdownNow();
executor.shutdown();
try {
    boolean flag = true;
    do {
        flag = ! executor.awaitTermination(500, TimeUnit.MILLISECONDS);
    } while (flag);
} catch (InterruptedException e) {
    // ...
}
```

awaitTermination(...)方法的内部实现很简单，如下所示。不断循环判断线程池是否到达了最终状态TERMINATED，如果是，就返回；如果不是，则通过termination条件变量阻塞一段时间，之后继续判断。

ThreadPoolExecutor.java

```
1445  public boolean awaitTermination(long timeout, TimeUnit unit)
1446      throws InterruptedException {
1447      long nanos = unit.toNanos(timeout);
1448      final ReentrantLock mainLock = this.mainLock;
1449      mainLock.lock();
1450      try {
1451          while (runStateLessThan(ctl.get(), TERMINATED)) {
1452              if (nanos <= 0L)
1453                  return false;
1454              nanos = termination.awaitNanos(nanos);
1455          }
1456          return true;
1457      } finally {
1458          mainLock.unlock();
1459      }
1460  }
```

判断线程池状态，是否是TERMINATED

### 3.shutdown()与shutdownNow()的区别

1. shutdown()不会清空任务队列，会等所有任务执行完成，shutdownNow()清空任务队列。
2. shutdown()只会中断空闲的线程，shutdownNow()会中断所有线程。

ThreadPoolExecutor.java

```
1369  public void shutdown() {
1370      final ReentrantLock mainLock = this.mainLock;
1371      mainLock.lock();
1372      try {
1373          checkShutdownAccess();
1374          advanceRunState(SHUTDOWN);
1375          interruptIdleWorkers();
1376          onShutdown(); // hook for ScheduledThreadPoolExecutor
1377      } finally {
1378          mainLock.unlock();
1379      }
1380      tryTerminate();
1381  }
```

加锁，确保线程安全

检查是否有关闭线程池的权限

将线程池状态修改位SHUTDOWN

中断空闲的线程

具有空方法体的钩子方法

ThreadPoolExecutor.java

```
1400    public List<Runnable> shutdownNow() {
1401        List<Runnable> tasks;
1402        final ReentrantLock mainLock = this.mainLock;
1403        mainLock.lock();   加锁，确保线程安全
1404        try {
1405            checkShutdownAccess(); 检查是否有关闭线程池的权限
1406            advanceRunState(STOP); 将线程池状态设置为STOP
1407            interruptWorkers();  中断所有线程
1408            tasks = drainQueue(); 任务队列清空
1409        } finally {
1410            mainLock.unlock();
1411        }
1412        tryTerminate();
1413        return tasks;
1414    }
```

下面看一下在上面的代码里中断空闲线程和中断所有线程的区别。

shutdown()方法中的interruptIdleWorkers()方法的实现：

ThreadPoolExecutor.java

```
809    private void interruptIdleWorkers() {
810        interruptIdleWorkers( onlyOne: false);
811    }
```

ThreadPoolExecutor.java

```
783    private void interruptIdleWorkers(boolean onlyOne) {
784        final ReentrantLock mainLock = this.mainLock;
785        mainLock.lock();
786        try {                                  如果tryLock成功，表示线程处于空闲状态；
787            for (Worker w : workers) {         如果不成功，表示线程持有锁，正在执行某个任务
788                Thread t = w.thread;
789                if (!t.isInterrupted() && w.tryLock()) {
790                    try {
791                        t.interrupt();
792                    } catch (SecurityException ignore) {
793                    } finally {
794                        w.unlock();
795                    }
796                }
797                if (onlyOne)
798                    break;
799            }
800        } finally {
801            mainLock.unlock();
802        }
803    }
```

关键区别点在tryLock()：一个线程在执行一个任务之前，会先加锁，这意味着通过是否持有锁，可以判断出线程是否处于空闲状态。tryLock()如果调用成功，说明线程处于空闲状态，向其发送中断信号；否则不发送。

tryLock()方法

ThreadPoolExecutor.java

```
655    public boolean tryLock()  { return tryAcquire( unused: 1); }
```

tryAcquire方法：

ThreadPoolExecutor.java

```
640    protected boolean tryAcquire(int unused) {
641        if (compareAndSetState( expect: 0,  update: 1)) {
642            setExclusiveOwnerThread(Thread.currentThread());
643            return true;
644        }
645        return false;
646    }
```

shutdownNow()调用了`interruptWorkers();`方法：

ThreadPoolExecutor.java

```
758    private void interruptWorkers() {
759        // assert mainLock.isHeldByCurrentThread();
760        for (Worker w : workers)
761            w.interruptIfStarted();
762    }
```

`interruptIfStarted()`方法的实现：

ThreadPoolExecutor.java

```
659    void interruptIfStarted() {
660        Thread t;
661        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
662            try {
663                t.interrupt();
664            } catch (SecurityException ignore) {
665            }
666        }
667    }
668}
```

只要启动了，并且没有被中断，则一律中断

在上面的代码中，shutdown() 和shutdownNow()都调用了tryTerminate()方法，如下所示：

```
1  final void tryTerminate() {
```

```
    for (;;) {
        int c = ctl.get();
        if (isRunning(c) || runStateAtLeast(c, TIDYING) ||
            (runStateLessThan(c, STOP) && ! workQueue.isEmpty()))
            return;
        if (workerCountOf(c) != 0) { // Eligible to terminate
            interruptIdleWorkers(ONLY_ONE);
            return;
        }
        // 当workQueue为空，wordCount为0时，执行下述代码。
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 将状态切换到到TIDYING状态
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    terminated(); // 调用钩子函数
                } finally {
                    ctl.set(ctlOf(TERMINATED, 0)); // 将状态由TIDYING改为
TERMINATED
                    termination.signalAll(); // 通知awaitTermination(...)
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
    }
}
```

tryTerminate()不会强行终止线程池，只是做了一下检测：当workerCount为0，workerQueue为空时，先把状态切换到TIDYING，然后调用钩子方法terminated()。当钩子方法执行完成时，把状态从TIDYING 改为 TERMINATED，接着调用termination.sinaglAll()，通知前面阻塞在awaitTermination的所有调用者线程。

所以，TIDYING和TREMINATED的区别是在二者之间执行了一个钩子方法terminated()，目前是一个空实现。

## 11.4 任务的提交过程分析

提交任务的方法如下：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    // 如果当前线程数小于corePoolSize，则启动新线程
    if (workerCountOf(c) < corePoolSize) {
```

```
        // 添加worker，并将command设置为worker线程的第一个任务开始执行。
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 如果当前的线程数大于或等于corePoolSize，则调用workQueue.offer放入队列
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // 如果线程池正在停止，则将command任务从队列移除，并拒绝command任务请求。
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // 放入队列中后发现没有线程执行任务，开启新线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 线程数大于maxPoolSize，并且队列已满，调用拒绝策略
    else if (!addWorker(command, false))
        reject(command);
}

// 该方法用于启动新线程。如果第二个参数为true，则使用corePoolSize作为上限，否则使用maxPoolSize作为上限。
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (int c = ctl.get();;) {
        // 如果线程池状态值起码是SHUTDOWN和STOP，或则第一个任务不是null，或者工作队列为空
        // 则添加worker失败，返回false
        if (runStateAtLeast(c, SHUTDOWN)
            && (runStateAtLeast(c, STOP)
                || firstTask != null
                || workQueue.isEmpty()))
            return false;

        for (;;) {
            // 工作线程数达到上限，要么是corePoolSize要么是maximumPoolSize，启动线程失败
            if (workerCountOf(c)
                >= ((core ? corePoolSize : maximumPoolSize) & COUNT_MASK))
                return false;
            // 增加worker数量成功，返回到retry语句
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            // 如果线程池运行状态起码是SHUTDOWN，则重试retry标签语句，CAS
            if (runStateAtLeast(c, SHUTDOWN))
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
    // worker数量加1成功后，接着运行：
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        // 新建worker对象
        w = new Worker(firstTask);
        // 获取线程对象
```

```
        final Thread t = w.thread;
        if (t != null) {
            final ReentrantLock mainLock = this.mainLock;
            // 加锁
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int c = ctl.get();

                if (isRunning(c) ||
                    (runStateLessThan(c, STOP) && firstTask == null)) {
                    // 由于线程已经在运行中，无法启动，抛异常
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    // 将线程对应的worker加入worker集合
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                // 释放锁
                mainLock.unlock();
            }
            // 如果添加worker成功，则启动该worker对应的线程
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        // 如果启动新线程失败
        if (! workerStarted)
            // workCount - 1
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

## 11.5 任务的执行过程分析

在上面的任务提交过程中，可能会开启一个新的Worker，并把任务本身作为firstTask赋给该Worker。但对于一个Worker来说，不是只执行一个任务，而是源源不断地从队列中取任务执行，这是一个不断循环的过程。

下面来看Woker的run()方法的实现过程。

```
private final class Worker extends AbstractQueuedSynchronizer implements
Runnable {
    // 当前worker对象封装的线程
    final Thread thread;
```

```
    // 线程需要运行的第一个任务。可以是null，如果是null，则线程从队列获取任务
    Runnable firstTask;
    // 记录线程执行完成的任务数量，每个线程一个计数器
    volatile long completedTasks;

    /**
        * 使用给定的第一个任务并利用线程工厂创建Worker实例
        * @param firstTask 线程的第一个任务，如果没有，就设置为null，此时线程会从队列获取任务。
        */
    Worker(Runnable firstTask) {
        setState(-1); // 线程处于阻塞状态，调用runWorker的时候中断
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }

    // 调用ThreadPoolExecutor的runWorker方法执行线程的运行
    public void run() {
        runWorker(this);
    }
}

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 中断Worker封装的线程
    w.unlock();
    boolean completedAbruptly = true;
    try {
        // 如果线程初始任务不是null，或者从队列获取的任务不是null，表示该线程应该执行任务。
        while (task != null || (task = getTask()) != null) {
            // 获取线程锁
            w.lock();
            // 如果线程池停止了，确保线程被中断
            // 如果线程池正在运行，确保线程不被中断
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                // 获取到任务后，再次检查线程池状态，如果发现线程池已经停止，则给自己发中断信号
                wt.interrupt();
            try {
                // 任务执行之前的钩子方法，实现为空
                beforeExecute(wt, task);
                try {
                    task.run();
                    // 任务执行结束后的钩子方法，实现为空
                    afterExecute(task, null);
                } catch (Throwable ex) {
                    afterExecute(task, ex);
                    throw ex;
                }
            } finally {
                // 任务执行完成，将task设置为null
                task = null;
```

```
                // 线程已完成的任务数加1
                w.completedTasks++;
                // 释放线程锁
                w.unlock();
            }
        }
        // 判断线程是否是正常退出
        completedAbruptly = false;
    } finally {
        // Worker退出
        processWorkerExit(w, completedAbruptly);
    }
}
```

**1.shutdown()与任务执行过程综合分析**

把任务的执行过程和上面的线程池的关闭过程结合起来进行分析，当调用 shutdown()的时候，可能出现以下几种场景：

1. 当调用shutdown()的时候，所有线程都处于空闲状态。

   这意味着任务队列一定是空的。此时，所有线程都会阻塞在 getTask()方法的地方。然后，所有线程都会收到interruptIdleWorkers()发来的中断信号，getTask()返回null，所有Worker都会退出while循环，之后执行processWorkerExit。

2. 当调用shutdown()的时候，所有线程都处于忙碌状态。

   此时，队列可能是空的，也可能是非空的。interruptIdleWorkers()内部的tryLock调用失败，什么都不会做，所有线程会继续执行自己当前的任务。之后所有线程会执行完队列中的任务，直到队列为空，getTask()才会返回null。之后，就和场景1一样了，退出while循环。

3. 当调用shutdown()的时候，部分线程忙碌，部分线程空闲。

   有部分线程空闲，说明队列一定是空的，这些线程肯定阻塞在 getTask()方法的地方。空闲的这些线程会和场景1一样处理，不空闲的线程会和场景2一样处理。

下面看一下getTask()方法的内部细节：

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();

        // 如果线程池调用了shutdownNow()，返回null
        // 如果线程池调用了shutdown()，并且任务队列为空，也返回null
        if (runStateAtLeast(c, SHUTDOWN)
            && (runStateAtLeast(c, STOP) || workQueue.isEmpty())) {
            // 工作线程数减一
            decrementWorkerCount();
            return null;
```

```
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            // 如果队列为空，就会阻塞pool或者take，前者有超时时间，后者没有超时时间
            // 一旦中断，此处抛异常，对应上文场景1。
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

**2.shutdownNow() 与任务执行过程综合分析**

和上面的 shutdown()类似，只是多了一个环节，即清空任务队列。如果一个线程正在执行某个业务代码，即使向它发送中断信号，也没有用，只能等它把代码执行完成。因此，中断空闲线程和中断所有线程的区别并不是很大，除非线程当前刚好阻塞在某个地方。

当一个Worker最终退出的时候，会执行清理工作：

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果线程正常退出，不会执行if的语句，这里一般是非正常退出，需要将worker数量减一
    if (completedAbruptly)
        decrementWorkerCount();
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        completedTaskCount += w.completedTasks;
        // 将自己的worker从集合移除
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }
    // 每个线程在结束的时候都会调用该方法，看是否可以停止线程池
    tryTerminate();
```

```
    int c = ctl.get();
    // 如果在线程退出前，发现线程池还没有关闭
    if (runStateLessThan(c, STOP)) {
        if (!completedAbruptly) {
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            // 如果线程池中没有其他线程了，并且任务队列非空
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;
            // 如果工作线程数大于min，表示队列中的任务可以由其他线程执行，退出当前线程
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        // 如果当前线程退出前发现线程池没有结束，任务队列不是空的，也没有其他线程来执行
        // 就再启动一个线程来处理。
        addWorker(null, false);
    }
}
```

## 11.6 线程池的4种拒绝策略

在execute(Runnable command)的最后，调用了reject(command)执行拒绝策略，代码如下所示：

ThreadPoolExecutor.java

```
            }
            else if (!addWorker(command, core: false))
                reject(command);
        }
```

ThreadPoolExecutor.java

```
    final void reject(Runnable command) {
        handler.rejectedExecution(command, executor: this);
    }
```

handler就是我们可以设置的拒绝策略管理器：

ThreadPoolExecutor.java

```
    private volatile RejectedExecutionHandler handler;
```

RejectedExecutionHandler 是一个接口，定义了四种实现，分别对应四种不同的拒绝策略，默认是AbortPolicy。

```java
package java.util.concurrent;
public interface RejectedExecutionHandler {
    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

ThreadPoolExecutor类中默认的实现是：

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```java
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }
```

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```java
    private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();
```

四种策略的实现代码如下：

策略1：调用者直接在自己的线程里执行，线程池不处理，比如到医院打点滴，医院没地方了，到你家自己操作吧：

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```java
    public static class CallerRunsPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code CallerRunsPolicy}.
         */
        public CallerRunsPolicy() { }

        /**
         * Executes task r in the caller's thread, unless the executor
         * has been shut down, in which case the task is discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }
```

策略2：线程池抛异常：

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```
2039    public static class AbortPolicy implements RejectedExecutionHandler {
2040        /**
2041         * Creates an {@code AbortPolicy}.
2042         */
2043        public AbortPolicy() { }
2044
2045        /**
2046         * Always throws RejectedExecutionException.
2047         *
2048         * @param r the runnable task requested to be executed
2049         * @param e the executor attempting to execute this task
2050         * @throws RejectedExecutionException always
2051         */
2052        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2053            throw new RejectedExecutionException("Task " + r.toString() +
2054                                                 " rejected from " +
2055                                                 e.toString());
2056        }
2057    }
```

策略3：线程池直接丢掉任务，神不知鬼不觉：

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```
2063    public static class DiscardPolicy implements RejectedExecutionHandler {
2064        /**
2065         * Creates a {@code DiscardPolicy}.
2066         */
2067        public DiscardPolicy() { }
2068
2069        /**
2070         * Does nothing, which has the effect of discarding task r.
2071         *
2072         * @param r the runnable task requested to be executed
2073         * @param e the executor attempting to execute this task
2074         */
2075        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2076        }
2077    }
```

策略4：删除队列中最早的任务，将当前任务入队列：

ThreadPoolExecutor.java × RejectedExecutionHandler.java ×

```
2083            */
2084        public static class DiscardOldestPolicy implements RejectedExecutionHandler {
2085            /**
2086             * Creates a {@code DiscardOldestPolicy} for the given executor.
2087             */
2088            public DiscardOldestPolicy() { }
2089
2090            /**
2091             * Obtains and ignores the next task that the executor
2092             * would otherwise execute, if one is immediately available,
2093             * and then retries execution of task r, unless the executor
2094             * is shut down, in which case task r is instead discarded.
2095             *
2096             * @param r the runnable task requested to be executed
2097             * @param e the executor attempting to execute this task
2098             */
2099            public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2100                if (!e.isShutdown()) {
2101                    e.getQueue().poll();
2102                    e.execute(r);
2103                }
2104            }
2105        }
2106    }
```

**示例程序：**

```
1  package com.lagou.concurrent.demo;
2
3  import java.util.concurrent.ArrayBlockingQueue;
4  import java.util.concurrent.ThreadPoolExecutor;
5  import java.util.concurrent.TimeUnit;
6
7  public class ThreadPoolExecutorDemo {
8      public static void main(String[] args) {
9          ThreadPoolExecutor executor = new ThreadPoolExecutor(
10                 3,
11                 5,
12                 1,
13                 TimeUnit.SECONDS,
14                 new ArrayBlockingQueue<>(3),
15 //                 new ThreadPoolExecutor.AbortPolicy()
16 //                 new ThreadPoolExecutor.CallerRunsPolicy()
17 //                 new ThreadPoolExecutor.DiscardOldestPolicy()
18                 new ThreadPoolExecutor.DiscardPolicy()
19         );
20
21         for (int i = 0; i < 20; i++) {
22             int finalI = i;
23             executor.execute(new Runnable() {
24                 @Override
25                 public void run() {
26                     System.out.println(Thread.currentThread().getId() + "["
   + finalI + "] -- 开始");
27                     try {
28                         Thread.sleep(5000);
```

```
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getId()  + "["
+ finalI + "] -- 结束");
                }
            });
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        executor.shutdown();
        boolean flag = true;

        try {
            do {
                flag = !executor.awaitTermination(1, TimeUnit.SECONDS);
                System.out.println(flag);
            } while (flag);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("线程池关闭成功。。。");
        System.out.println(Thread.currentThread().getId());
    }
}
```

# 12 Executors工具类

concurrent包提供了Executors工具类，利用它可以创建各种不同类型的线程池。

## 12.1 四种对比

单线程的线程池：

Executors.java

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

固定数目线程的线程池：

```java
Executors.java
91    public static ExecutorService newFixedThreadPool(int nThreads) {
92        return new ThreadPoolExecutor(nThreads, nThreads,
93                                      keepAliveTime: 0L, TimeUnit.MILLISECONDS,
94                                      new LinkedBlockingQueue<Runnable>());
95    }
```

每接收一个请求，就创建一个线程来执行：

```java
Executors.java
217    public static ExecutorService newCachedThreadPool() {
218        return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
219                                      keepAliveTime: 60L, TimeUnit.SECONDS,
220                                      new SynchronousQueue<Runnable>());
221    }
```

单线程具有周期调度功能的线程池：

```java
Executors.java
254    public static ScheduledExecutorService newSingleThreadScheduledExecutor() {
255        return new DelegatedScheduledExecutorService
256            (new ScheduledThreadPoolExecutor( corePoolSize: 1));
257    }
```

多线程，有调度功能的线程池：

```java
Executors.java
288    public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
289        return new ScheduledThreadPoolExecutor(corePoolSize);
290    }
```

### 12.2 最佳实践

不同类型的线程池，其实都是由前面的几个关键配置参数配置而成的。

在《阿里巴巴Java开发手册》中，明确禁止使用Executors创建线程池，并要求开发者直接使用ThreadPoolExector或ScheduledThreadPoolExecutor进行创建。这样做是为了强制开发者明确线程池的运行策略，使其对线程池的每个配置参数皆做到心中有数，以规避因使用不当而造成资源耗尽的风险。

## 13 ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor实现了按时间调度来执行任务：

1. 延迟执行任务

ScheduledThreadPoolExecutor.java

```
570     public <V> ScheduledFuture<V> schedule(Callable<V> callable,
571                                            long delay,
572                                            TimeUnit unit) {
```

ScheduledThreadPoolExecutor.java

```
553     public ScheduledFuture<?> schedule(Runnable command,
554                                        long delay,
555                                        TimeUnit unit) {
```

2. 周期执行任务

ScheduledThreadPoolExecutor.java

```
616     public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
617                                                   long initialDelay,
618                                                   long period,
619                                                   TimeUnit unit) {
```

ScheduledThreadPoolExecutor.java

```
664     public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
665                                                      long initialDelay,
666                                                      long delay,
667                                                      TimeUnit unit) {
```

区别如下：

AtFixedRate：按固定频率执行，与任务本身执行时间无关。但有个前提条件，任务执行时间必须小于间隔时间，例如间隔时间是5s，每5s执行一次任务，任务的执行时间必须小于5s。

WithFixedDelay：按固定间隔执行，与任务本身执行时间有关。例如，任务本身执行时间是10s，间隔2s，则下一次开始执行的时间就是12s。

## 13.1 延迟执行和周期性执行的原理

ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，这意味着其内部的数据结构和ThreadPoolExecutor是基本一样的，那它是如何实现延迟执行任务和周期性执行任务的呢？

延迟执行任务依靠的是DelayQueue。DelayQueue是 BlockingQueue的一种，其实现原理是二叉堆。

而周期性执行任务是执行完一个任务之后，再把该任务扔回到任务队列中，如此就可以对一个任务反复执行。

不过这里并没有使用DelayQueue，而是在ScheduledThreadPoolExecutor内部又实现了一个**特定的DelayQueue**。

ScheduledThreadPoolExecutor.java

```
899     static class DelayedWorkQueue extends AbstractQueue<Runnable>
900         implements BlockingQueue<Runnable> {
```

其原理和DelayQueue一样，但针对任务的取消进行了优化。下面主要讲延迟执行和周期性执行的实现过程。

## 13.2 延迟执行

ScheduledThreadPoolExecutor.java

```
553     public ScheduledFuture<?> schedule(Runnable command,
554                                        long delay,
555                                        TimeUnit unit) {
556         if (command == null || unit == null)
557             throw new NullPointerException();
558         RunnableScheduledFuture<Void> t = decorateTask(command,
559             new ScheduledFutureTask<Void>(command, result: null,
560                                           triggerTime(delay, unit),
561                                           sequencer.getAndIncrement()));
562         delayedExecute(t);
563         return t;
564     }
```

传进去的是一个Runnable，外加延迟时间delay。在内部通过decorateTask(...)方法把Runnable包装成一个ScheduleFutureTask对象，而DelayedWorkQueue中存放的正是这种类型的对象，这种类型的对象一定实现了Delayed接口。

ScheduledThreadPoolExecutor.java

```
338     private void delayedExecute(RunnableScheduledFuture<?> task) {
339         if (isShutdown())
340             reject(task);
341         else {
342             super.getQueue().add(task); 将ScheduledFutureTask放入队列中
343             if (!canRunInCurrentRunState(task) && remove(task))
344                 task.cancel( mayInterruptIfRunning: false);
345             else
346                 ensurePrestart();
347         }
348     }
```

ScheduledThreadPoolExecutor.java　ThreadPoolExecutor.java

```
1580    void ensurePrestart() {
1581        int wc = workerCountOf(ctl.get());
1582        if (wc < corePoolSize)
1583            addWorker( firstTask: null,  core: true);
1584        else if (wc == 0)
1585            addWorker( firstTask: null,  core: false);
1586    }
1587
```

如果wc小于corePoolSize，则addWork，开新线程，否则什么也不做

从上面的代码中可以看出，schedule()方法本身很简单，就是把提交的Runnable任务加上delay时间，转换成ScheduledFutureTask对象，放入DelayedWorkerQueue中。任务的执行过程还是复用的ThreadPoolExecutor，延迟的控制是在DelayedWorkerQueue内部完成的。

## 13.3 周期性执行

ScheduledThreadPoolExecutor.java

```
664     public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
665                                                      long initialDelay,
666                                                      long delay,
667                                                      TimeUnit unit) {
668         if (command == null || unit == null)
669             throw new NullPointerException();
670         if (delay <= 0L)
671             throw new IllegalArgumentException();
672         ScheduledFutureTask<Void> sft =
673             new ScheduledFutureTask<Void>(command,
674                                           result: null,
675                                           triggerTime(initialDelay, unit),
676                                           -unit.toNanos(delay), 负数
677                                           sequencer.getAndIncrement());
678         RunnableScheduledFuture<Void> t = decorateTask(command, sft);
679         sft.outerTask = t;
680         delayedExecute(t);
681         return t;
682     }
```

ScheduledThreadPoolExecutor.java

```
616     public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
617                                                   long initialDelay,
618                                                   long period,
619                                                   TimeUnit unit) {
620         if (command == null || unit == null)
621             throw new NullPointerException();
622         if (period <= 0L)
623             throw new IllegalArgumentException();
624         ScheduledFutureTask<Void> sft =
625             new ScheduledFutureTask<Void>(command,
626                                           result: null,
627                                           triggerTime(initialDelay, unit),
628                                           unit.toNanos(period), 正数
629                                           sequencer.getAndIncrement());
630         RunnableScheduledFuture<Void> t = decorateTask(command, sft);
631         sft.outerTask = t;
632         delayedExecute(t);
633         return t;
634     }
```

和schedule(...)方法的框架基本一样，也是包装一个ScheduledFutureTask对象，只是在延迟时间参数之外多了一个周期参数，然后放入DelayedWorkerQueue就结束了。

两个方法的区别在于一个传入的周期是一个负数，另一个传入的周期是一个正数，为什么要这样做呢?

用于生成任务序列号的sequencer，创建ScheduledFutureTask的时候使用：

ScheduledThreadPoolExecutor.java

```
183        private static final AtomicLong sequencer = new AtomicLong();
```

```
1  private class ScheduledFutureTask<V>
2              extends FutureTask<V> implements RunnableScheduledFuture<V> {
3      private final long sequenceNumber;
4      private volatile long time;
5      private final long period;
6
7      ScheduledFutureTask(Runnable r, V result, long triggerTime,
8                          long period, long sequenceNumber) {
9          super(r, result);
10         this.time = triggerTime; // 延迟时间
11         this.period = period; // 周期
12         this.sequenceNumber = sequenceNumber;
13     }
14
15     // 实现Delayed接口
16     public long getDelay(TimeUnit unit) {
17         return unit.convert(time - System.nanoTime(), NANOSECONDS);
18     }
19
20     // 实现Comparable接口
21     public int compareTo(Delayed other) {
22         if (other == this) // compare zero if same object
23             return 0;
24         if (other instanceof ScheduledFutureTask) {
25             ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
26             long diff = time - x.time;
27             if (diff < 0)
28                 return -1;
29             else if (diff > 0)
30                 return 1;
31             // 延迟时间相等，进一步比较序列号
32             else if (sequenceNumber < x.sequenceNumber)
33                 return -1;
34             else
35                 return 1;
36         }
37         long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECONDS);
38         return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;
39     }
40
41     // 实现Runnable接口
42     public void run() {
43         if (!canRunInCurrentRunState(this))
44             cancel(false);
45         // 如果不是周期执行，则执行一次
46         else if (!isPeriodic())
47             super.run();
```

```
            // 如果是周期执行，则重新设置下一次运行的时间，重新入队列
            else if (super.runAndReset()) {
                setNextRunTime();
                reExecutePeriodic(outerTask);
            }
        }

        // 下一次执行时间
        private void setNextRunTime() {
            long p = period;
            if (p > 0)
                time += p;
            else
                time = triggerTime(-p);
        }
    }

    // 下一次触发时间
    long triggerTime(long delay) {
        return System.nanoTime() +
            ((delay < (Long.MAX_VALUE >> 1)) ? delay : overflowFree(delay));
    }

    // 放到队列中，等待下一次执行
    void reExecutePeriodic(RunnableScheduledFuture<?> task) {
        if (canRunInCurrentRunState(task)) {
            super.getQueue().add(task);
            if (canRunInCurrentRunState(task) || !remove(task)) {
                ensurePrestart();
                return;
            }
        }
        task.cancel(false);
    }
```

withFixedDelay和atFixedRate的区别就体现在setNextRunTime里面。

如果是atFixedRate，period > 0，下一次开始执行时间等于上一次开始执行时间+period；

如果是withFixedDelay，period < 0，下一次开始执行时间等于triggerTime(-p)，为now+(-period)，now即上一次执行的结束时间。

## 14 CompletableFuture用法

从JDK 8开始，在Concurrent包中提供了一个强大的异步编程工具CompletableFuture。在JDK8之前，异步编程可以通过线程池和Future来实现，但功能还不够强大。

示例代码：

```
package com.lagou.concurrent.demo;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
```

```
5
6  public class CompletableFutureDemo {
7      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
8          CompletableFuture<String> future = new CompletableFuture<>();
9
10         new Thread(() -> {
11             try {
12                 Thread.sleep(1000);
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16             future.complete("hello world");
17         }).start();
18
19         System.out.println("获取结果中。。。");
20         String result = future.get();
21         System.out.println("获取的结果: " + result);
22     }
23 }
```

CompletableFuture实现了Future接口，所以它也具有Future的特性：调用get()方法会阻塞在那，直到结果返回。

另外1个线程调用complete方法完成该Future，则所有阻塞在get()方法的线程都将获得返回结果。

## 14.1 runAsync与supplyAsync

上面的例子是一个空的任务，下面尝试提交一个真的任务，然后等待结果返回。

**例1：runAsync(Runnable)**

```
1  package com.lagou.concurrent.demo;
2
3  import java.util.concurrent.CompletableFuture;
4  import java.util.concurrent.ExecutionException;
5
6  public class CompletableFutureDemo2 {
7      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
8          CompletableFuture<Void> voidCompletableFuture =
   CompletableFuture.runAsync(() -> {
9              try {
10                 Thread.sleep(2000);
11                 System.out.println("任务执行完成");
12             } catch (InterruptedException e) {
13                 e.printStackTrace();
14             }
15         });
16         // 阻塞，等待任务执行完成
17         voidCompletableFuture.get();
18         System.out.println("程序运行结束");
19
```

```
    }
}
```

CompletableFuture.runAsync(...)传入的是一个Runnable接口。

**例2：supplyAsync(Supplier)**

```
package com.lagou.concurrent.demo;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.function.Supplier;

public class CompletableFutureDemo3 {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        CompletableFuture<String> future = CompletableFuture.supplyAsync(new Supplier<String>() {
            @Override
            public String get() {
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return "这是结果";
            }
        });

        String result = future.get();
        System.out.println("任务执行结果：" + result);
    }
}
```

例2和例1的区别在于，例2的任务有返回值。没有返回值的任务，提交的是Runnable，返回的是CompletableFuture<Void>；有返回值的任务，提交的是 Supplier，返回的是CompletableFuture<String>。Supplier和前面的Callable很相似。

通过上面两个例子可以看出，在基本的用法上，CompletableFuture和Future很相似，都可以提交两类任务：一类是无返回值的，另一类是有返回值的。

## 14.2 thenRun、thenAccept和thenApply

对于 Future，在提交任务之后，只能调用 get()等结果返回；但对于 CompletableFuture，可以在结果上面再加一个callback，当得到结果之后，再接着执行callback。

**例1：thenRun(Runnable)**

```
package com.lagou.concurrent.demo;
```

```
2
3  import java.util.concurrent.CompletableFuture;
4  import java.util.concurrent.ExecutionException;
5  import java.util.concurrent.TimeUnit;
6
7  public class CompletableFutureDemo4 {
8      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
9          CompletableFuture voidCompletableFuture =
   CompletableFuture.supplyAsync(() -> {
10             try {
11                 Thread.sleep(5);
12             } catch (InterruptedException e) {
13                 e.printStackTrace();
14             }
15             return "这是结果";
16         }).thenRun(() ->  {
17             try {
18                 Thread.sleep(2);
19             } catch (InterruptedException e) {
20                 e.printStackTrace();
21             }
22
23             System.out.println("任务执行结束之后执行的语句");
24         });
25
26         // 阻塞等待任务执行完成
27         voidCompletableFuture.get();
28         System.out.println("任务执行结束");
29     }
30 }
```

该案例最后不能获取到结果，只会得到一个null。

**例2：thenAccept(Consumer)**

```
1  package com.lagou.concurrent.demo;
2
3  import java.util.concurrent.CompletableFuture;
4  import java.util.concurrent.ExecutionException;
5  import java.util.function.Consumer;
6
7  public class CompletableFutureDemo5 {
8      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
9          CompletableFuture<Void> future = CompletableFuture.supplyAsync(() ->
   {
10             try {
11                 Thread.sleep(5000);
12             } catch (InterruptedException e) {
13                 e.printStackTrace();
14             }
15             System.out.println("返回中间结果");
16             return "这是中间结果";
17         }).thenAccept((param) -> {
18             try {
```

```
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("任务执行后获得前面的中间结果：" + param);
        });
        // 阻塞等待任务执行完成
        future.get();
        System.out.println("任务执行完成");
    }
}
```

上述代码在thenAccept中可以获取任务的执行结果，接着进行处理。

**例3：thenApply(Function)**

```
package com.lagou.concurrent.demo;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.Function;

public class CompletableFutureDemo6 {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("返回中间结果");
            return "abcdefg";
        }).thenApply(new Function<String, Integer>() {
            @Override
            public Integer apply(String middle) {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("获取中间结果，再次计算返回");

                return middle.length();
            }
        });

        Integer integer = future.get();
        System.out.println("最终的结果为：" + integer);

    }
}
```

三个例子都是在任务执行完成之后，接着执行回调，只是回调的形式不同：

1. thenRun后面跟的是一个无参数、无返回值的方法，即Runnable，所以最终的返回值是CompletableFuture<Void>类型。
2. thenAccept后面跟的是一个有参数、无返回值的方法，称为Consumer，返回值也是CompletableFuture<Void>类型。顾名思义，只进不出，所以称为Consumer；前面的Supplier，是无参数，有返回值，只出不进，和Consumer刚好相反。
3. thenApply 后面跟的是一个有参数、有返回值的方法，称为Function。返回值是CompletableFuture<String>类型。

而参数接收的是前一个任务，即 supplyAsync(...)这个任务的返回值。因此这里只能用supplyAsync，不能用runAsync。因为runAsync没有返回值，不能为下一个链式方法传入参数。

## 14.3 thenCompose与thenCombine

**例1：thenCompose**

在上面的例子中，thenApply接收的是一个Function，但是这个Function的返回值是一个通常的基本数据类型或一个对象，而不是另外一个 CompletableFuture。如果 Function 的返回值也是一个CompletableFuture，就会出现嵌套的CompletableFuture。考虑下面的例子：

```
1 CompletableFuture<CompletableFuture<Integer>> future =
  CompletableFuture.supplyAsync(new Supplier<String>() {
2     @Override
3     public String get() {
```

```
        return "hello world";
    }
}).thenApply(new Function<String, CompletableFuture<Integer>>() {
    @Override
    public CompletableFuture<Integer> apply(String s) {
        return CompletableFuture.supplyAsync(new Supplier<Integer>() {
            @Override
            public Integer get() {
                return s.length();
            }
        });
    }
});

CompletableFuture<Integer> future1 = future.get();
Integer result = future1.get();
System.out.println(result);
```

如果希望返回值是一个非嵌套的CompletableFuture，可以使用thenCompose：

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(new
Supplier<String>() {
    @Override
    public String get() {
        return "hello world";
    }
}).thenCompose(new Function<String, CompletionStage<Integer>>() {
    @Override
    public CompletionStage<Integer> apply(String s) {
        return CompletableFuture.supplyAsync(new Supplier<Integer>() {
            @Override
            public Integer get() {
                return s.length();
            }
        });
    }
});
Integer integer = future.get();
System.out.println(integer);
```

下面是thenCompose方法的接口定义：

CompletableFuture.java × CompletionStage.java ×

```
647     public <U> CompletionStage<U> thenCompose
648         (Function<? super T, ? extends CompletionStage<U>> fn);
```

CompletableFuture中的实现：

CompletableFuture.java

```
2233    public <U> CompletableFuture<U> thenCompose(
2234        Function<? super T, ? extends CompletionStage<U>> fn) {
2235        return uniComposeStage( e: null, fn);
2236    }
```

从该方法的定义可以看出，它传入的参数是一个Function类型，并且Function的返回值必须是CompletionStage的子类，也就是CompletableFuture类型。

**例2：thenCombine**

thenCombine方法的接口定义如下，从传入的参数可以看出，它不同于thenCompose。

CompletableFuture.java | CompletionStage.java

```
308    public <U,V> CompletionStage<V> thenCombine
309        (CompletionStage<? extends U> other,
310         BiFunction<? super T,? super U,? extends V> fn);
311
```

第1个参数是一个CompletableFuture类型，第2个参数是一个方法，并且是一个BiFunction，也就是该方法有2个输入参数，1个返回值。

从该接口的定义可以大致推测，它是要在2个 CompletableFuture 完成之后，把2个CompletableFuture的返回值传进去，再额外做一些事情。实例如下：

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(new
Supplier<String>() {
    @Override
    public String get() {
        return "hello";
    }
}).thenCombine(CompletableFuture.supplyAsync(new Supplier<String>() {
    @Override
    public String get() {
        return "lagou";
    }
}), new BiFunction<String, String, Integer>() {
    @Override
    public Integer apply(String s, String s2) {
        return s.length() + s2.length();
    }
});
Integer result = future.get();
System.out.println(result);
```

## 14.4 任意个CompletableFuture的组合

上面的thenCompose和thenCombine只能组合2个CompletableFuture，而接下来的allOf 和 anyOf 可以组合任意多个CompletableFuture。方法接口定义如下所示。

CompletableFuture.java

```
2336    public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) {
2337        return andTree(cfs, lo: 0, hi: cfs.length - 1);
2338    }
```

CompletableFuture.java

```
2355    public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) {
2356        int n; Object r;
2357        if ((n = cfs.length) <= 1)
2358            return (n == 0)
2359                ? new CompletableFuture<Object>()
```

首先，这两个方法都是静态方法，参数是变长的CompletableFuture的集合。其次，allOf和anyOf的区别，前者是“与”，后者是“或”。

allOf的返回值是CompletableFuture<Void>类型，这是因为每个传入的CompletableFuture的返回值都可能不同，所以组合的结果是无法用某种类型来表示的，索性返回Void类型。

anyOf 的含义是只要有任意一个 CompletableFuture 结束，就可以做接下来的事情，而无须像AllOf那样，等待所有的CompletableFuture结束。

但由于每个CompletableFuture的返回值类型都可能不同，任意一个，意味着无法判断是什么类型，所以anyOf的返回值是CompletableFuture<Object>类型。

```
1  public class CompletableFutureDemo11 {
2
3      private static final Random RANDOM = new Random();
4      private static volatile int result = 0;
5
6      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
7          CompletableFuture[] futures = new CompletableFuture[10];
8
9          for (int i = 0; i < 10; i++) {
10             int finalI = i;
11             CompletableFuture<Void> future = CompletableFuture.runAsync(new
   Runnable() {
12                 @Override
13                 public void run() {
14                     try {
15                         Thread.sleep(1000 + RANDOM.nextInt(1000));
16                     } catch (InterruptedException e) {
17                         e.printStackTrace();
18                     }
19                     result++;
```

```
                }
            });
            futures[i] = future;
        }
        System.out.println(result);

//        for (int i = 0; i < 10; i++) {
//            futures[i].get();
//            System.out.println(result);
//        }

//        Integer allResult = CompletableFuture.allOf(futures).thenApply(new Function<Void, Integer>() {
//            @Override
//            public Integer apply(Void unused) {
//                return result;
//            }
//        }).get();
//
//        System.out.println(allResult);

        Integer anyResult = CompletableFuture.anyOf(futures).thenApply(new Function<Object, Integer>() {
            @Override
            public Integer apply(Object o) {
                return result;
            }
        }).get();
        System.out.println(anyResult);

    }
}
```

## 14.5 四种任务原型

通过上面的例子可以总结出，提交给CompletableFuture执行的任务有四种类型：Runnable、Consumer、Supplier、Function。下面是这四种任务原型的对比。

| 四种任务原型 | 无参数 | 有参数 |
| --- | --- | --- |
| 无返回值 | Runnable接口<br>对应的提交方法：runAsync，thenRun | Consumer接口<br>对应的提交方法：thenAccept |
| 有返回值 | Supplier接口：<br>对应的提交方法：supplierAsync | Function接口<br>对应的提交方法：thenApply |

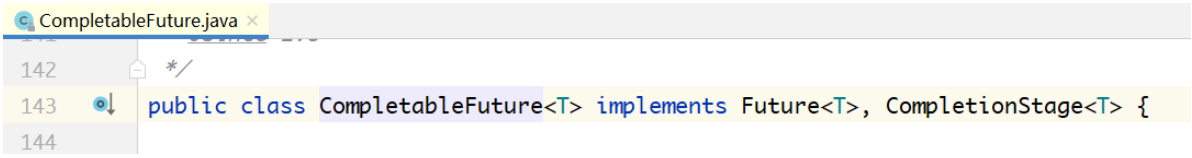
runAsync 与 supplierAsync 是 CompletableFuture 的静态方法；而 thenAccept、thenAsync、thenApply是CompletableFutre的成员方法。

因为初始的时候没有CompletableFuture对象，也没有参数可传，所以提交的只能是Runnable或者Supplier，只能是静态方法；

通过静态方法生成CompletableFuture对象之后，便可以链式地提交其他任务了，这个时候就可以提交Runnable、Consumer、Function，且都是成员方法。

## 14.6 CompletionStage接口

CompletableFuture不仅实现了Future接口，还实现了CompletableStage接口。




CompletionStage接口定义的正是前面的各种链式方法、组合方法，如下所示。

```
package java.util.concurrent;

public interface CompletionStage<T> {
    //
    public CompletionStage<Void> thenRun(Runnable action);
    public CompletionStage<Void> thenAccept(Consumer<? super T> action);
    public <U> CompletionStage<U> thenApply(Function<? super T,? extends U>
fn);
    public <U> CompletionStage<U> thenCompose
        (Function<? super T, ? extends CompletionStage<U>> fn);
    public <U,V> CompletionStage<V> thenCombine
        (CompletionStage<? extends U> other,
         BiFunction<? super T,? super U,? extends V> fn);
    // ...
}
```

关于CompletionStage接口，有几个关键点要说明：

1. 所有方法的返回值都是CompletionStage类型，也就是它自己。正因为如此，才能实现如下的链式调用：future1.thenApply(...).thenApply(...).thenCompose(...).thenRun(...)。
2. thenApply接收的是一个有输入参数、返回值的Function。这个Function的输入参数，必须是？Super T 类型，也就是T或者T的父类型，而T必须是调用thenApplycompletableFuture对象的类型；返回值则必须是？Extends U类型，也就是U或者U的子类型，而U恰好是thenApply的返回值的CompletionStage对应的类型。

其他方法，诸如thenCompose、thenCombine也是类似的原理。

## 14.7 CompletableFuture内部原理

### 14.7.1 CompletableFuture的构造：ForkJoinPool

CompletableFuture中任务的执行依靠ForkJoinPool：

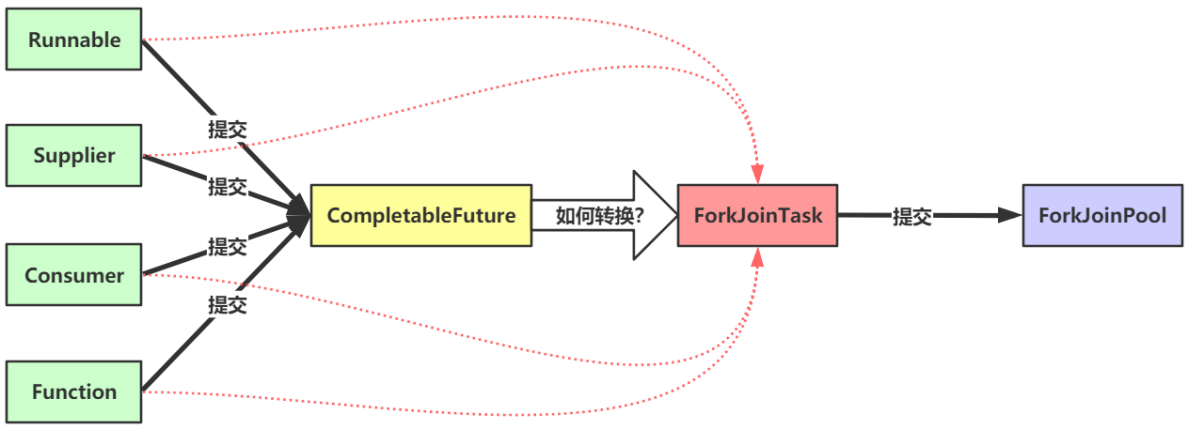
```
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {
    private static final Executor asyncPool = useCommonPool ?
ForkJoinPool.commonPool() : new ThreadPerTaskExecutor();
```

```
    public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
{
        return asyncSupplyStage(asyncPool, supplier);
    }
    static <U> CompletableFuture<U> asyncSupplyStage(Executor e, Supplier<U>
f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<U> d = new CompletableFuture<U>();
        // Supplier转换为ForkJoinTask
        e.execute(new AsyncSupply<U>(d, f));
        return d;
    }
    // ...
}
```

通过上面的代码可以看到，asyncPool是一个static类型，supplierAsync、asyncSupplyStage也都是static方法。Static方法会返回一个CompletableFuture类型对象，之后就可以链式调用，CompletionStage里面的各个方法。

### 14.7.2 任务类型的适配

ForkJoinPool接受的任务是ForkJoinTask 类型，而我们向CompletableFuture提交的任务是`Runnable/Supplier/Consumer/Function`。因此，肯定需要一个适配机制，把这四种类型的任务转换成ForkJoinTask，然后提交给ForkJoinPool，如下图所示：



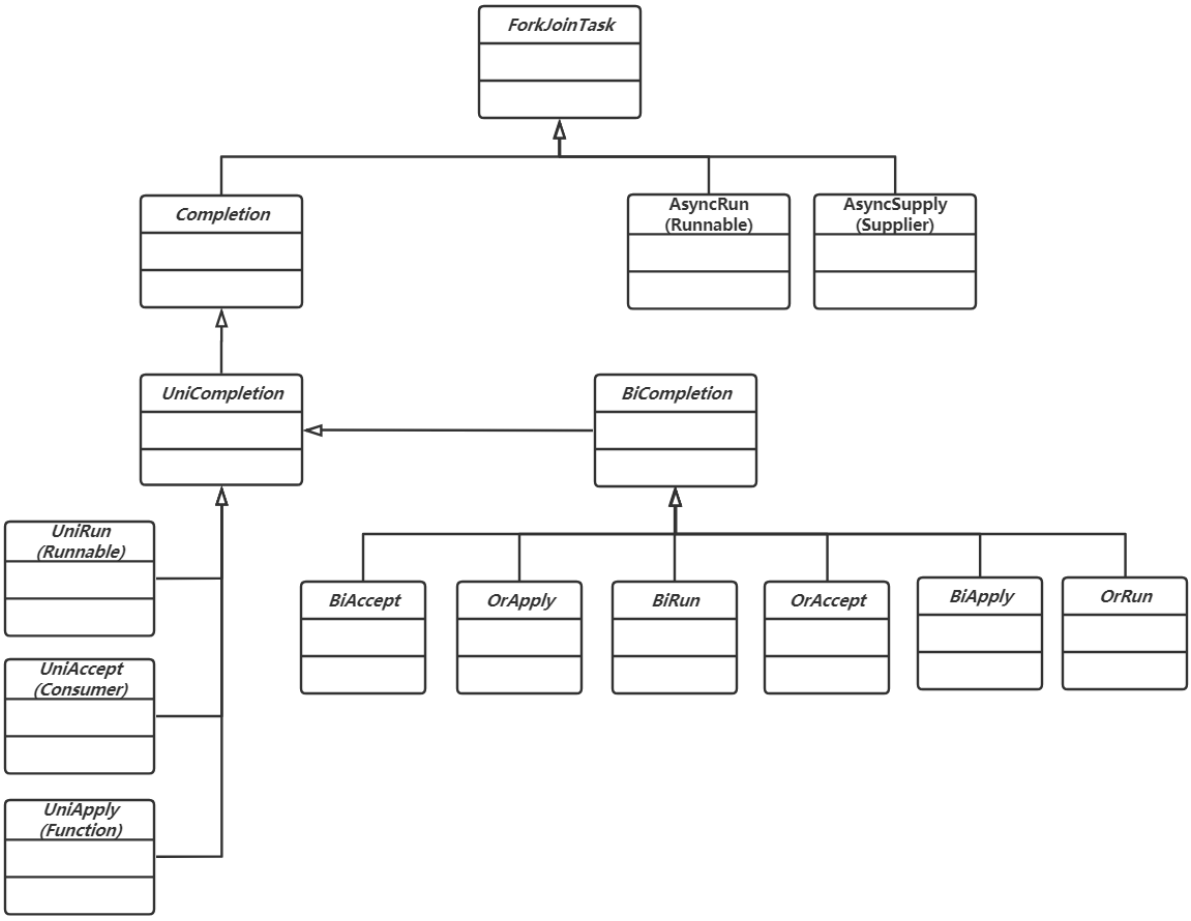

为了完成这种转换，在CompletableFuture内部定义了一系列的内部类，下图是CompletableFuture的各种内部类的继承体系。

在 supplyAsync(...)方法内部，会把一个 Supplier 转换成一个 AsyncSupply，然后提交给ForkJoinPool执行；

在runAsync(...)方法内部，会把一个Runnable转换成一个AsyncRun，然后提交给ForkJoinPool执行；

在 thenRun/thenAccept/thenApply 内部，会分别把 Runnable/Consumer/Function 转换成UniRun/UniAccept/UniApply对象，然后提交给ForkJoinPool执行；

除此之外，还有两种 CompletableFuture 组合的情况，分为“与”和“或”，所以有对应的Bi和Or类型的Completion类型。




下面的代码分别为 UniRun、UniApply、UniAccept 的定义，可以看到，其内部分别封装了 Runnable、Function、Consumer。

CompletableFuture.java

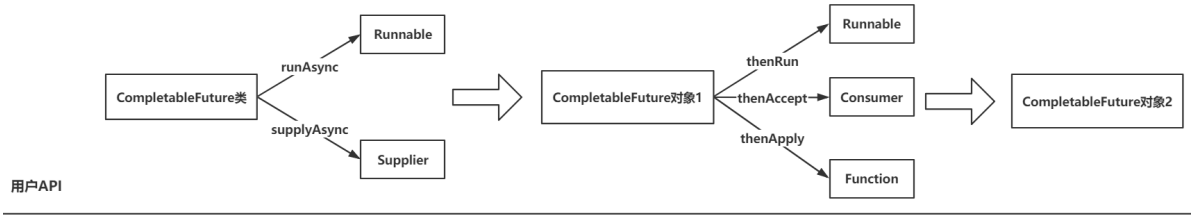
```
763     static final class UniRun<T> extends UniCompletion<T,Void> {
764         Runnable fn;
765         UniRun(Executor executor, CompletableFuture<Void> dep,
766                CompletableFuture<T> src, Runnable fn) {
767             super(executor, dep, src); this.fn = fn;
```

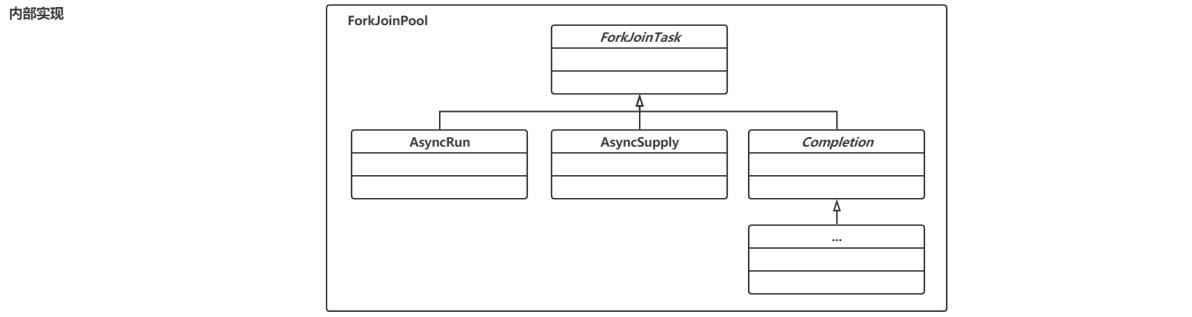
CompletableFuture.java

```
616     static final class UniApply<T,V> extends UniCompletion<T,V> {
617         Function<? super T,? extends V> fn;
618         UniApply(Executor executor, CompletableFuture<V> dep,
619                  CompletableFuture<T> src,
```

CompletableFuture.java

```
689     static final class UniAccept<T> extends UniCompletion<T,Void> {
690         Consumer<? super T> fn;
691         UniAccept(Executor executor, CompletableFuture<Void> dep,
692                   CompletableFuture<T> src, Consumer<? super T> fn) {
```




### 14.7.3 任务的链式执行过程分析

下面以CompletableFuture.supplyAsync(...).thenApply(...).thenRun(...)链式代码为例，分析整个执行过程。

第1步：CompletableFuture future1=CompletableFuture.supplyAsync(...)

CompletableFuture.java

```
1914    public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
1915        return asyncSupplyStage(ASYNC_POOL, supplier);
1916    }
```

CompletableFuture.java

```
433     * Default executor -- ForkJoinPool.commonPool() unless it cannot
434     * support parallelism.
435     */
436    private static final Executor ASYNC_POOL = USE_COMMON_POOL ?
437        ForkJoinPool.commonPool() : new ThreadPerTaskExecutor();
```

CompletableFuture.java

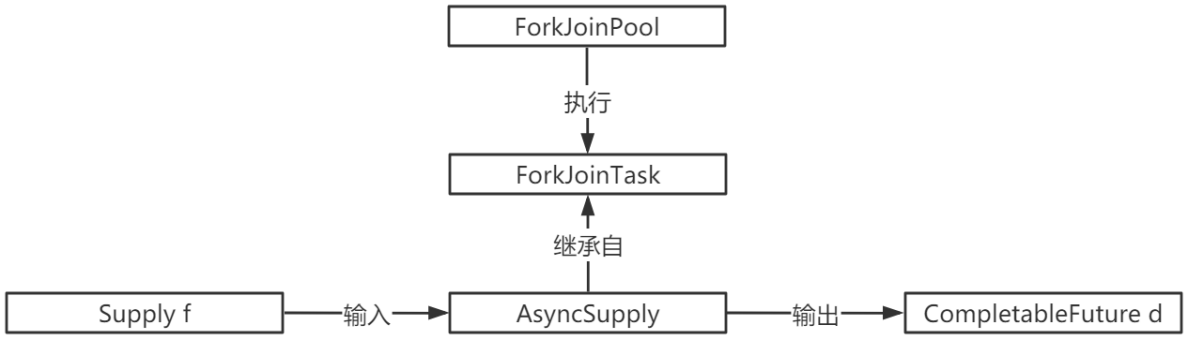
```
1710    static <U> CompletableFuture<U> asyncSupplyStage(Executor e,
1711                                                     Supplier<U> f) {
1712        if (f == null) throw new NullPointerException();
1713        CompletableFuture<U> d = new CompletableFuture<U>();
1714        e.execute(new AsyncSupply<U>(d, f));
1715        return d;
1716    }
```

在上面的代码中，关键是构造了一个AsyncSupply对象，该对象有三个关键点：

1. 它继承自ForkJoinTask，所以能够提交ForkJoinPool来执行。

2. 它封装了Supplier f，即它所执行任务的具体内容。
3. 该任务的返回值，即CompletableFuture d，也被封装在里面。

ForkJoinPool执行一个ForkJoinTask类型的任务，即AsyncSupply。该任务的输入就是Supply，输出结果存放在CompletableFuture中。




第2步：CompletableFuture future2=future1.thenApply(...)

第1步的返回值，也就是上面代码中的 CompletableFuture d，紧接着调用其成员方法thenApply：

CompletableFuture.java

```
2092    public <U> CompletableFuture<U> thenApply(
2093        Function<? super T,? extends U> fn) {
2094        return uniApplyStage( e: null, fn);
2095    }
```

CompletableFuture.java

```
653    private <V> CompletableFuture<V> uniApplyStage(
654        Executor e, Function<? super T,? extends V> f) {
655        if (f == null) throw new NullPointerException();
656        Object r;
657        if ((r = result) != null)  如果上一个任务执行结束，立即执行该任务
658            return uniApplyNow(r, e, f);
659        CompletableFuture<V> d = newIncompleteFuture();
660        unipush(new UniApply<T,V>(e, d,  src: this, f));  如果上一个任务没有执行完，则将
661        return d;                                          当前任务压入任务栈
662    }
```

我们知道，必须等第1步的任务执行完毕，第2步的任务才可以执行。因此，这里提交的任务不可能立即执行，在此处构建了一个UniApply对象，也就是一个ForkJoinTask类型的任务，这个任务放入了第1个任务的栈当中。

CompletableFuture.java

```
579         */
580     @   final void unipush(Completion c) {
581             if (c != null) {
582                 while (!tryPushStack(c)) { 尝试将任务入栈，如果失败，则重试
583                     if (result != null) {
584                         NEXT.set(c, null);
585                         break;
586                     }
587                 }
588                 if (result != null)  如果上一个任务执行结束，则触发该任务的执行
589                     c.tryFire(SYNC);
590             }
591         }
```

每一个CompletableFuture对象内部都有一个栈，存储着是后续依赖它的任务，如下面代码所示。这个栈也就是Treiber Stack，这里的stack存储的就是栈顶指针。

CompletableFuture.java

```
264         volatile Object result;       // Either the result or boxed AltResult
265         volatile Completion stack;    // Top of Treiber stack of dependent actions
```

上面的UniApply对象类似于第1步里面的AsyncSupply，它的构造方法传入了4个参数：

1. 第1个参数是执行它的ForkJoinPool；
2. 第2个参数是输出一个CompletableFuture对象。这个参数，也是thenApply方法的返回值，用来链式执行下一个任务；
3. 第3个参数是其依赖的前置任务，也就是第1步里面提交的任务；
4. 第4个参数是输入（也就是一个Function对象）。

CompletableFuture.java

```
615         /serial/
616         static final class UniApply<T,V> extends UniCompletion<T,V> {
617             Function<? super T,? extends V> fn;
618             UniApply(Executor executor, CompletableFuture<V> dep,
619                      CompletableFuture<T> src,
620                      Function<? super T,? extends V> fn) {
621                 super(executor, dep, src); this.fn = fn;
622             }
```

UniApply对象被放入了第1步的CompletableFuture的栈中，在第1步的任务执行完成之后，就会从栈中弹出并执行。如下代码：

CompletableFuture.java

```
1682        /serial/
1683        static final class AsyncSupply<T> extends ForkJoinTask<Void>
1684            implements Runnable, AsynchronousCompletionTask {
1685            CompletableFuture<T> dep; Supplier<? extends T> fn;
1686            AsyncSupply(CompletableFuture<T> dep, Supplier<? extends T> fn) {
1687                this.dep = dep; this.fn = fn;
1688            }
1689
1690            public final Void getRawResult() { return null; }
1691 @          public final void setRawResult(Void v) {}
1692            public final boolean exec() { run(); return false; }
1693
1694            public void run() {
1695                CompletableFuture<T> d; Supplier<? extends T> f;
1696                if ((d = dep) != null && (f = fn) != null) {
1697                    dep = null; fn = null;
1698                    if (d.result == null) {
1699                        try {
1700                            d.completeValue(f.get()); Supplier的get方法
1701                        } catch (Throwable ex) {
1702                            d.completeThrowable(ex);
1703                        }
1704                    }
1705                    d.postComplete();
1706                }
1707            }
1708    }
```

ForkJoinPool执行上面的AsyncSupply对象的run()方法，实质就是执行Supplier的get()方法。执行结果被塞入了 CompletableFuture d 当中，也就是赋值给了 CompletableFuture 内部的Object result 变量。

调用d.postComplete()，也正是在这个方法里面，把第2步压入的UniApply对象弹出来执行，代码如下所示。

CompletableFuture.java

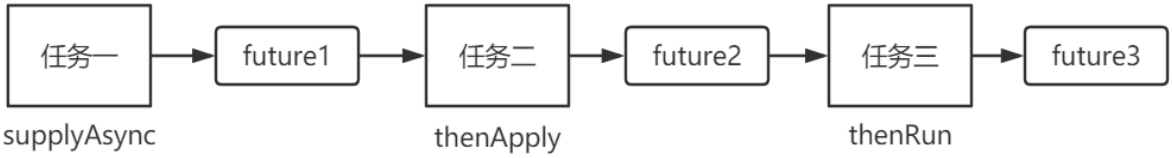
```
487         */
488     final void postComplete() {
489         /*
490          * On each step, variable f holds current dependents to pop
491          * and run.  It is extended along only one path at a time,
492          * pushing others to avoid unbounded recursion.
493          */
494         CompletableFuture<?> f = this; Completion h;
495         while ((h = f.stack) != null ||
496                (f != this && (h = (f = this).stack) != null)) {
497             CompletableFuture<?> d; Completion t;
498             if (STACK.compareAndSet(f, h, t = h.next)) {  出栈
499                 if (t != null) {
500                     if (f != this) {
501                         pushStack(h);
502                         continue;
503                     }
504                     NEXT.compareAndSet( ...args: h, t, null); // try to detach
505                 }
506                 f = (d = h.tryFire(NESTED)) == null ? this : d;  执行出栈任务
507             }
508         }
509     }
```

第3步：CompletableFuture future3=future2.thenRun()

第3步和第2步的过程类似，构建了一个 UniRun 对象，这个对象被压入第2步的CompletableFuture所在的栈中。第2步的任务，当执行完成时，从自己的栈中弹出UniRun对象并执行。

综上所述：

通过supplyAsync/thenApply/thenRun，分别提交了3个任务，每1个任务都有1个返回值对象，也就是1个CompletableFuture。这3个任务通过2个CompletableFuture完成串联。后1个任务，被放入了前1个任务的CompletableFuture里面，前1个任务在执行完成时，会从自己的栈中，弹出下1个任务执行。如此向后传递，完成任务的链式执行。




#### 14.7.4 thenApply与thenApplyAsync的区别

在上面的代码中，我们分析了thenApply，还有一个与之对应的方法是thenApplyAsync。这两个方法调用的是同一个方法，只不过传入的参数不同。

CompletableFuture.java

```
2092    public <U> CompletableFuture<U> thenApply(
2093        Function<? super T,? extends U> fn) {
2094        return uniApplyStage( e: null, fn);
2095    }
```

CompletableFuture.java

```
2097    public <U> CompletableFuture<U> thenApplyAsync(
2098        Function<? super T,? extends U> fn) {
2099        return uniApplyStage(defaultExecutor(), fn);
2100    }
```

CompletableFuture.java

```
653 @       private <V> CompletableFuture<V> uniApplyStage(
654             Executor e, Function<? super T,? extends V> f) {
655             if (f == null) throw new NullPointerException();
656             Object r;
657             if ((r = result) != null)
658                 return uniApplyNow(r, e, f);
659             CompletableFuture<V> d = newIncompleteFuture();
660             unipush(new UniApply<T,V>(e, d, src: this, f));
661             return d;
662         }
```

对于上一个任务已经得出结果的情况：

CompletableFuture.java

```
664 @   private <V> CompletableFuture<V> uniApplyNow(
665         Object r, Executor e, Function<? super T,? extends V> f) {
666         Throwable x;
667         CompletableFuture<V> d = newIncompleteFuture();
668         if (r instanceof AltResult) {
669             if ((x = ((AltResult)r).ex) != null) {
670                 d.result = encodeThrowable(x, r);
671                 return d;
672             }
673             r = null;
674         }
675         try {
676             if (e != null) {    e != null表示是thenApplyAsync方法
677                 e.execute(new UniApply<T,V>( executor: null, d, src: this, f));
678             } else {    e == null表示是thenApply方法
679                 @SuppressWarnings("unchecked") T t = (T) r;
680                 d.result = d.encodeValue(f.apply(t));
681             }
682         } catch (Throwable ex) {
683             d.result = encodeThrowable(ex);
684         }
685         return d;
686     }
```

首先将第一个任务的结果转换为T类型
然后直接调用Funtion的apply方法
而apply方法就是我们传值的Function对象
中实现的apply方法，接收上一个任务的结果
作为参数

如果e != null表示是thenApplyAsync，需要调用ForkJoinPool的execute方法，该方法：

CompletableFuture.java | ForkJoinPool.java

```
2445    public void execute(Runnable task) {
2446        if (task == null)
2447            throw new NullPointerException();
2448        ForkJoinTask<?> job;
2449        if (task instanceof ForkJoinTask<?>) // avoid re-wrap
2450            job = (ForkJoinTask<?>) task;
2451        else
2452            job = new ForkJoinTask.RunnableExecuteAction(task);
2453        externalSubmit(job); 提交任务
2454    }
```

CompletableFuture.java | ForkJoinPool.java

```
1911        */
1912    @   private <T> ForkJoinTask<T> externalSubmit(ForkJoinTask<T> task) {
1913            Thread t; ForkJoinWorkerThread w; WorkQueue q;
1914            if (task == null)
1915                throw new NullPointerException();
1916            if (((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) &&
1917                (w = (ForkJoinWorkerThread)t).pool == this &&
1918                (q = w.workQueue) != null)
1919                q.push(task); 将任务入栈上一个任务的栈，之后取出执行
1920            else
1921                externalPush(task);
1922            return task;
1923        }
```

通过上面的代码可以看到：

1. 如果前置任务没有完成，即a.result=null，thenApply和thenApplyAsync都会将当前任务的下一个任务入栈；然后再出栈执行；
2. 只有在当前任务已经完成的情况下，thenApply才会立即执行，不会入栈，再出栈，不会交给ForkJoinPool；thenApplyAsync还是将下一个任务封装为ForkJoinTask，入栈，之后出栈再执行。

同理，thenRun与thenRunAsync、thenAccept与thenAcceptAsync的区别与此类似。

## 14.8 任务的网状执行：有向无环图

如果任务只是链式执行，便不需要在每个CompletableFuture里面设1个栈了，用1个指针使所有任务组成链表即可。

但实际上，任务不只是链式执行，而是网状执行，组成 1 张图。如下图所示，所有任务组成一个有向无环图：

任务一执行完成之后，任务二、任务三可以并行，在代码层面可以写为：future1.thenApply（任务二），future1.thenApply（任务三）；

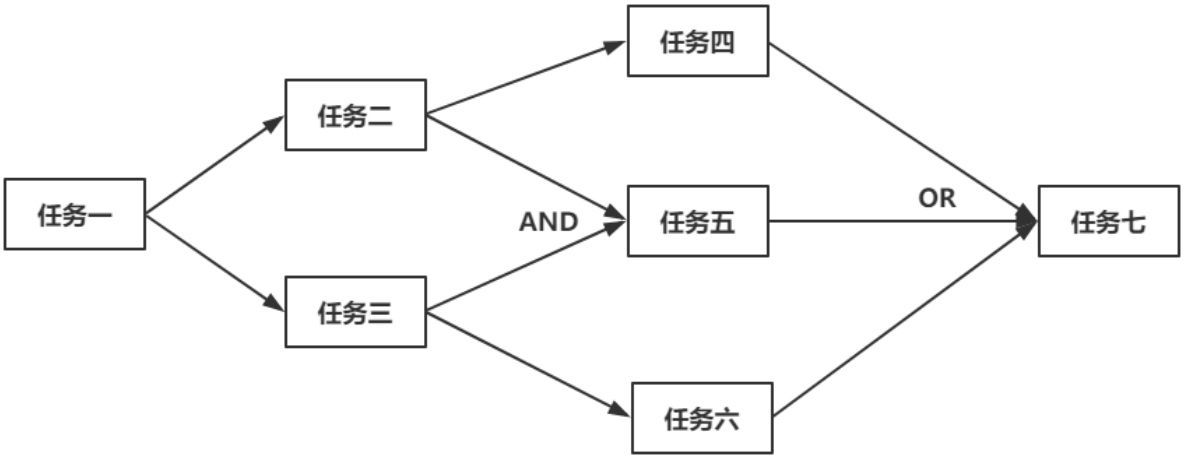
任务四在任务二执行完成时可开始执行；

任务五要等待任务二、任务三都执行完成，才能开始，这里是AND关系；

任务六在任务三执行完成时可以开始执行；

对于任务七，只要任务四、任务五、任务六中任意一个任务结束，就可以开始执行。

总而言之，任务之间是多对多的关系：1个任务有*n*个依赖它的后继任务；1个任务也有*n*个它依赖的前驱任务。




这样一个有向无环图，用什么样的数据结构表达呢？AND和OR的关系又如何表达呢？

有几个关键点：

1. 在每个任务的返回值里面，存储了依赖它的接下来要执行的任务。所以在上图中，任务一的CompletableFuture的栈中存储了任务二、任务三；任务二的CompletableFuutre中存储了任务四、任务五；任务三的CompletableFuture中存储了任务五、任务六。即每个任务的CompletableFuture对象的栈里面，其实存储了该节点的出边对应的任务集合。
2. 任务二、任务三的CompletableFuture里面，都存储了任务五，那么任务五是不是会被触发两次，执行两次呢？

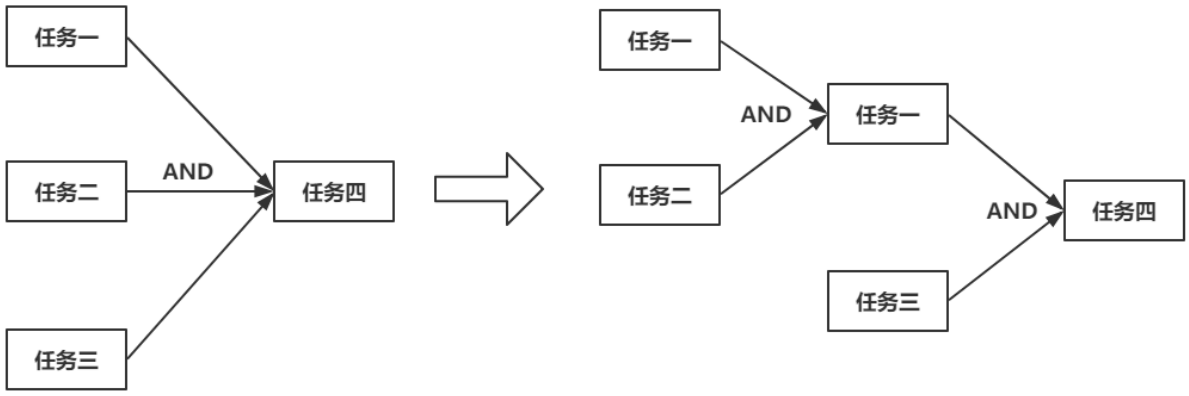
   任务五的确会被触发二次，但它会判断任务二、任务三的结果是不是都完成，如果只完成其中一个，它就不会执行。
3. 任务七存在于任务四、任务五、任务六的CompletableFuture的栈里面，因此会被触发三次。但它只会执行一次，只要其中1个任务执行完成，就可以执行任务七了。
4. 正因为有AND和OR两种不同的关系，因此对应BiApply和OrApply两个对象，这两个对象的构造方法几乎一样，只是在内部执行的时候，一个是AND的逻辑，一个是OR的逻辑。

CompletableFuture.java

```
1189        /serial/
1190        static final class BiApply<T,U,V> extends BiCompletion<T,U,V> {
1191            BiFunction<? super T,? super U,? extends V> fn;
1192            BiApply(Executor executor, CompletableFuture<V> dep,
1193                    CompletableFuture<T> src, CompletableFuture<U> snd,
1194                    BiFunction<? super T,? super U,? extends V> fn) {
1195                super(executor, dep, src, snd); this.fn = fn;
1196            }
```

CompletableFuture.java

```
1480        /serial/
1481        static final class OrApply<T,U extends T,V> extends BiCompletion<T,U,V> {
1482            Function<? super T,? extends V> fn;
1483            OrApply(Executor executor, CompletableFuture<V> dep,
1484                    CompletableFuture<T> src, CompletableFuture<U> snd,
1485                    Function<? super T,? extends V> fn) {
1486                super(executor, dep, src, snd); this.fn = fn;
1487            }
```




5. BiApply和OrApply都是二元操作符，也就是说，只能传入二个被依赖的任务。但上面的任务七同时依赖于任务四、任务五、任务六，这怎么处理呢?

   任何一个多元操作，都能被转换为多个二元操作的叠加。如上图所示，假如任务一AND任务二AND任务三 ==> 任务四，那么它可以被转换为右边的形式。新建了一个AND任务，这个AND任务和任务三再作为参数，构造任务四。OR的关系，与此类似。

此时，thenCombine的内部实现原理也就可以解释了。thenCombine用于任务一、任务二执行完成，再执行任务三。

## 14.9 allOf内部的计算图分析

下面以allOf方法为例，看一下有向无环计算图的内部运作过程：

CompletableFuture.java

```
2336    @   public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) {
2337            return andTree(cfs, lo: 0, hi: cfs.length - 1);
2338        }
```

CompletableFuture.java

```
1432    /** Recursively constructs a tree of completions. */
1433    static CompletableFuture<Void> andTree(CompletableFuture<?>[] cfs,
1434                                           int lo, int hi) {
1435        CompletableFuture<Void> d = new CompletableFuture<~>();
1436        if (lo > hi) // empty
1437            d.result = NIL;
1438        else {
1439            CompletableFuture<?> a, b; Object r, s, z; Throwable x;
1440            int mid = (lo + hi) >>> 1;
1441            if ((a = (lo == mid ? cfs[lo] :
1442                      andTree(cfs, lo, mid))) == null ||
1443                (b = (lo == hi ? a : (hi == mid+1) ? cfs[hi] :
1444                      andTree(cfs,  lo: mid+1, hi))) == null)
1445                throw new NullPointerException();
1446            if ((r = a.result) == null || (s = b.result) == null)
1447                a.bipush(b, new BiRelay<>(d, a, b));
1448            else if ((r instanceof AltResult
1449                      && (x = ((AltResult)(z = r)).ex) != null) ||
1450                     (s instanceof AltResult
1451                      && (x = ((AltResult)(z = s)).ex) != null))
1452                d.result = encodeThrowable(x, z);
1453            else
1454                d.result = NIL;
1455        }
1456        return d;
1457    }
```

任务d被分别压入a和b所在的栈
当a或者b执行结束，就可以接着执行d了

上面的方法是一个递归方法，输入是一个CompletableFuture对象的列表，输出是一个具有AND关系的复合CompletableFuture对象。

最关键的代码如上面加注释部分所示，因为d要等a，b都执行完成之后才能执行，因此d会被分别压入a，b所在的栈中。

CompletableFuture.java

```
1157    /**
1158     * Pushes completion to this and b unless both done.
1159     * Caller should first check that either result or b.result is null.
1160     */
1161    final void bipush(CompletableFuture<?> b, BiCompletion<?,?,?> c) {
1162        if (c != null) {
1163            while (result == null) {
1164                if (tryPushStack(c)) { 将c入栈a任务
1165                    if (b.result == null)
1166                        b.unipush(new CoCompletion(c)); 将c入栈b任务
1167                    else if (result != null)
1168                        c.tryFire(SYNC);
1169                    return;
1170                }
1171            }
1172            b.unipush(c);
1173        }
1174    }
```

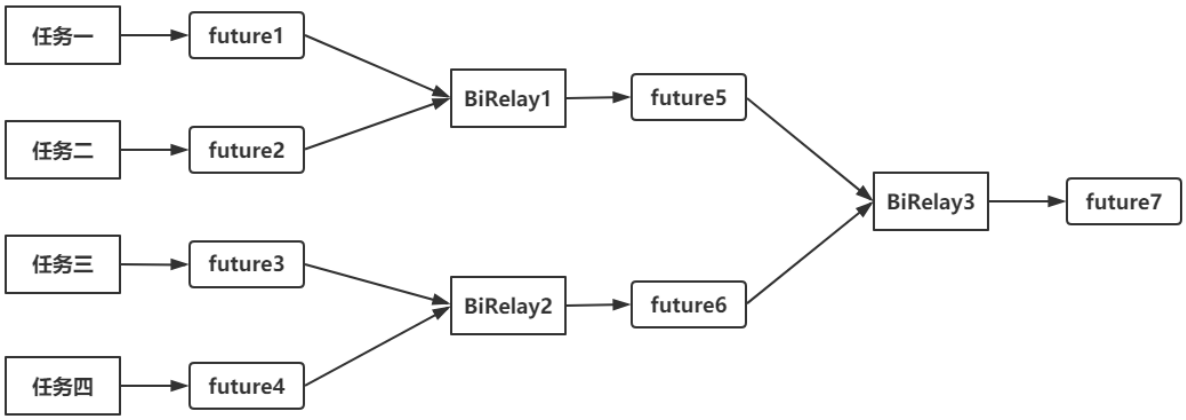
CompletableFuture.java

```
580    final void unipush(Completion c) {
581        if (c != null) {
582            while (!tryPushStack(c)) {
583                if (result != null) {
584                    NEXT.set(c, null);
585                    break;
586                }
587            }
588            if (result != null)
589                c.tryFire(SYNC);
590        }
591    }
```

下图为allOf内部的运作过程。假设allof的参数传入了future1、future2、future3、future4，则对应四个原始任务。

生成BiRelay1、BiRelay2任务，分别压入future1/future2、future3/future4的栈中。无论future1或future2完成，都会触发BiRelay1；无论future3或future4完成，都会触发BiRelay2；

生成BiRelay3任务，压入future5/future6的栈中，无论future5或future6完成，都会触发BiRelay3任务。




BiRelay只是一个中转任务，它本身没有任务代码，只是参照输入的两个future是否完成。如果完成，就从自己的栈中弹出依赖它的BiRelay任务，然后执行。

# 第四部分：ForkJoinPool

# 15 ForkJoinPool用法

ForkJoinPool就是JDK7提供的一种“分治算法”的多线程并行计算框架。Fork意为分叉，Join意为合并，一分一合，相互配合，形成分治算法。此外，也可以将ForkJoinPool看作一个单机版的Map/Reduce，多个线程并行计算。

相比于ThreadPoolExecutor，ForkJoinPool可以更好地实现计算的负载均衡，提高资源利用率。

假设有5个任务，在ThreadPoolExecutor中有5个线程并行执行，其中一个任务的计算量很大，其余4个任务的计算量很小，这会导致1个线程很忙，其他4个线程则处于空闲状态。

利用ForkJoinPool，可以把大的任务拆分成很多小任务，然后这些小任务被所有的线程执行，从而实现任务计算的负载均衡。

**例子1：快排**

快排有2个步骤：

1. 利用数组的第1个元素把数组划分成两半，左边数组里面的元素小于或等于该元素，右边数组里面的元素比该元素大；
2. 对左右的两个子数组分别排序。

左右两个子数组相互独立可以并行计算。利用ForkJoinPool，代码如下：

```
package com.lagou.concurrent.demo;

import java.util.Arrays;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.TimeUnit;

public class ForkJoinPoolDemo01 {

    static class SortTask extends RecursiveAction {
        final long[] array;
        final int lo;
        final int hi;

        public SortTask(long[] array) {
            this.array = array;
            this.lo = 0;
            this.hi = array.length - 1;
        }

        public SortTask(long[] array, int lo, int hi) {
            this.array = array;
            this.lo = lo;
            this.hi = hi;
        }

        private int partition(long[] array, int lo, int hi) {
            long x = array[hi];
```

```
            int i = lo - 1;
            for (int j = lo; j < hi; j++) {
                if (array[j] <= x) {
                    i++;
                    swap(array, i, j);
                }
            }
            swap(array, i + 1, hi);
            return i + 1;
        }

        private void swap(long[] array, int i, int j) {
            if (i != j) {
                long temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        @Override
        protected void compute() {
            if (lo < hi) {
                // 找到分区的元素下标
                int pivot = partition(array, lo, hi);
                // 将数组分为两部分
                SortTask left = new SortTask(array, lo, pivot - 1);
                SortTask right = new SortTask(array, pivot + 1, hi);

                left.fork();
                right.fork();
                left.join();
                right.join();
            }
        }

    }

    public static void main(String[] args) throws InterruptedException {
        long[] array = {5, 3, 7, 9, 2, 4, 1, 8, 10};
        // 一个任务
        ForkJoinTask sort = new SortTask(array);
        // 一个pool
        ForkJoinPool pool = new ForkJoinPool();
        // ForkJoinPool开启多个线程，同时执行上面的子任务
        pool.submit(sort);
        // 结束ForkJoinPool
        pool.shutdown();
        // 等待结束Pool
        pool.awaitTermination(10, TimeUnit.SECONDS);

        System.out.println(Arrays.toString(array));
    }
}
```

**例子2：求1到n个数的和**

```java
package com.lagou.concurrent.demo;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

public class ForkJoinPoolDemo02 {

    static class SumTask extends RecursiveTask<Long> {
        private static final int THRESHOLD = 10;
        private long start;
        private long end;

        public SumTask(long n) {
            this(1, n);
        }

        public SumTask(long start, long end) {
            this.start = start;
            this.end = end;
        }

        @Override
        protected Long compute() {
            long sum = 0;
            // 如果计算的范围在threshold之内，则直接进行计算
            if ((end - start) <= THRESHOLD) {
                for (long l = start; l <= end; l++) {
                    sum += l;
                }
            } else {
                // 否则找出起始和结束的中间值，分割任务
                long mid = (start + end) >>> 1;
                SumTask left = new SumTask(start, mid);
                SumTask right = new SumTask(mid + 1, end);
                left.fork();
                right.fork();
                // 收集子任务计算结果
                sum = left.join() + right.join();
            }
            return sum;
        }
    }

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        SumTask sum = new SumTask(100);
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinTask<Long> future = pool.submit(sum);
        Long aLong = future.get();
        System.out.println(aLong);
        pool.shutdown();
    }
}
```

上面的代码用到了 RecursiveAction 和 RecursiveTask 两个类，它们都继承自抽象类 ForkJoinTask，用到了其中关键的接口 fork()、join()。二者的区别是一个有返回值，一个没有返回值。

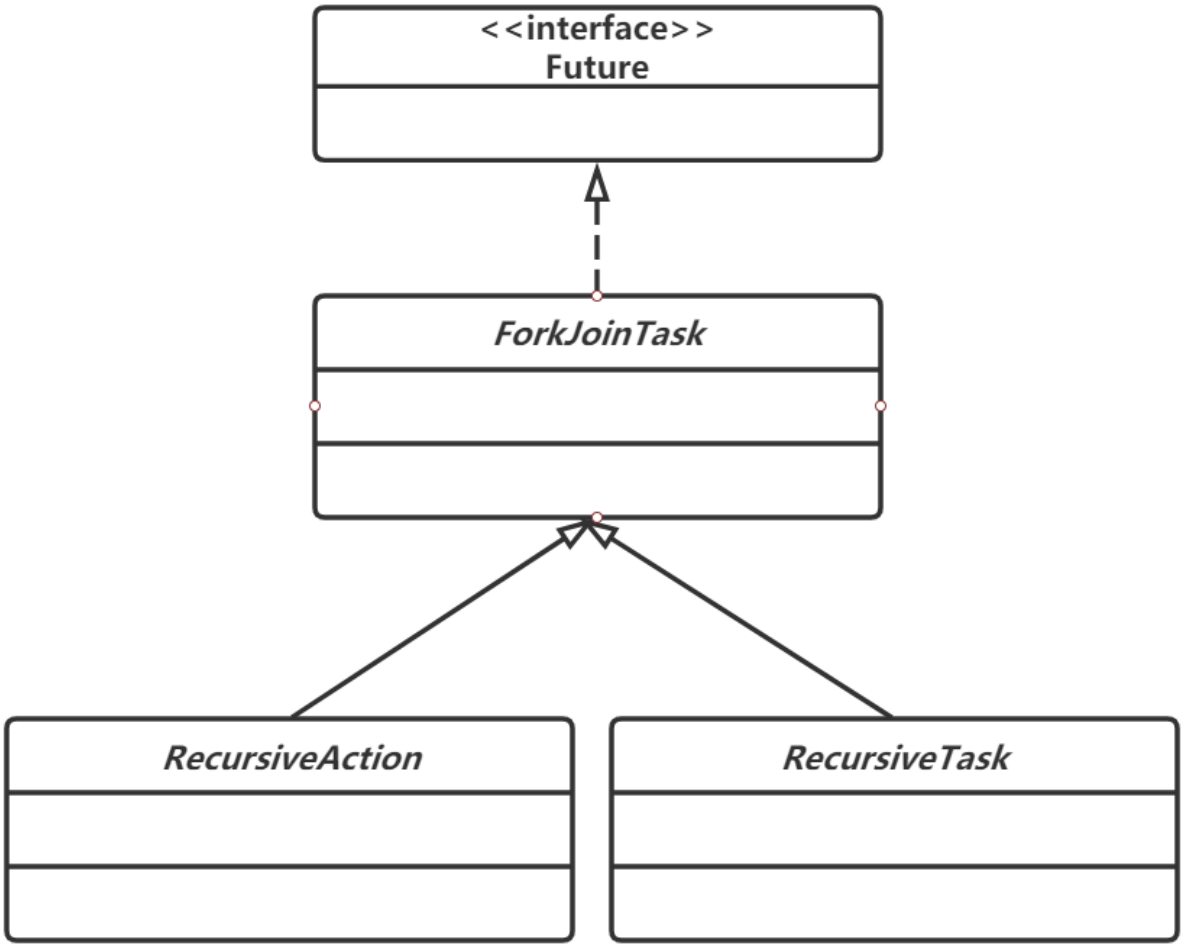
RecursiveAction.java

```
165  public abstract class RecursiveAction extends ForkJoinTask<Void> {
```

RecursiveAction.java | RecursiveTask.java

```
68  public abstract class RecursiveTask<V> extends ForkJoinTask<V> {
```

RecursiveAction/RecursiveTask类继承关系：



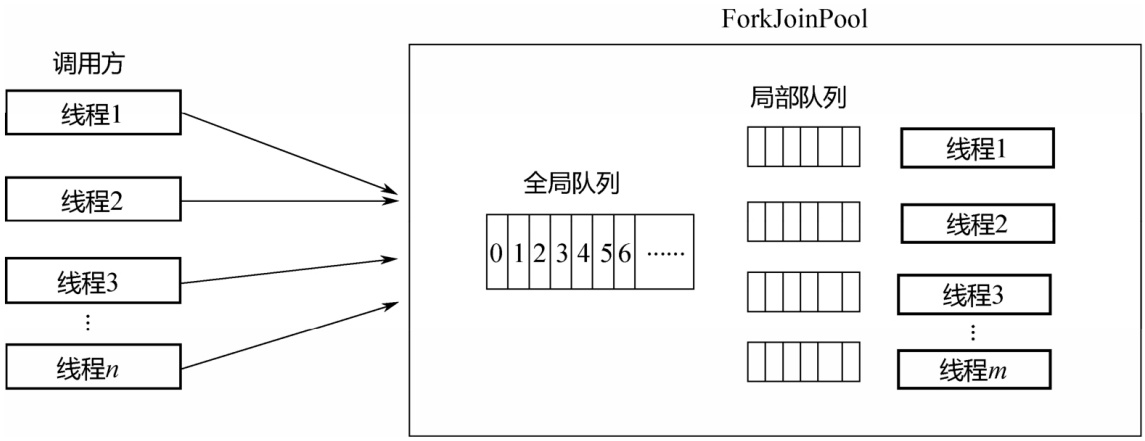

在ForkJoinPool中，对应的接口如下：

RecursiveAction.java | RecursiveTask.java | ForkJoinPool.java

```
2466      public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) {
2467          return externalSubmit(task);
2468      }
```

# 16 核心数据结构

与ThreadPoolExector不同的是，除一个全局的任务队列之外，每个线程还有一个自己的局部队列。




核心数据结构如下所示：

```
public class ForkJoinPool extends AbstractExecutorService {

    // 状态变量，类似于ThreadPoolExecutor中的ctl变量。
    volatile long ctl;
    // 工作线程队列
    WorkQueue[] workQueues;
    // 工作线程工厂
    final ForkJoinWorkerThreadFactory factory;
    // 下一个worker的下标
    int indexSeed;

    static final class WorkQueue {
        volatile int source;       // source queue id, or sentinel
        int id;                    // 在ForkJoinPool的workQueues数组中的下标
        int base;                  // 队列尾部指针
        int top;                   // 队列头指针
        volatile int phase;        // versioned, negative: queued, 1: locked
        int stackPred;             // pool stack (ctl) predecessor link
        int nsteals;               // number of steals
        ForkJoinTask<?>[] array;   // 工作线程的局部队列
        final ForkJoinPool pool;   // the containing pool (may be null)
        final ForkJoinWorkerThread owner; // 该工作队列的所有者线程，null表示共享的
    }
}

public class ForkJoinWorkerThread extends Thread {
    // 当前工作线程所在的线程池，反向引用
    final ForkJoinPool pool;
    // 工作队列
    final ForkJoinPool.WorkQueue workQueue;
}


```

下面看一下这些核心数据结构的构造过程。

ForkJoinPool.java

```
2143    public ForkJoinPool() { 如果不指定并发数，默认为CPU核心个数
2144        this(Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors()),
2145             defaultForkJoinWorkerThreadFactory, handler: null, asyncMode: false,
2146             corePoolSize: 0, MAX_CAP, minimumRunnable: 1, saturate: null, DEFAULT_KEEPALIVE, TimeUnit.MILLISECONDS);
2147    }
```

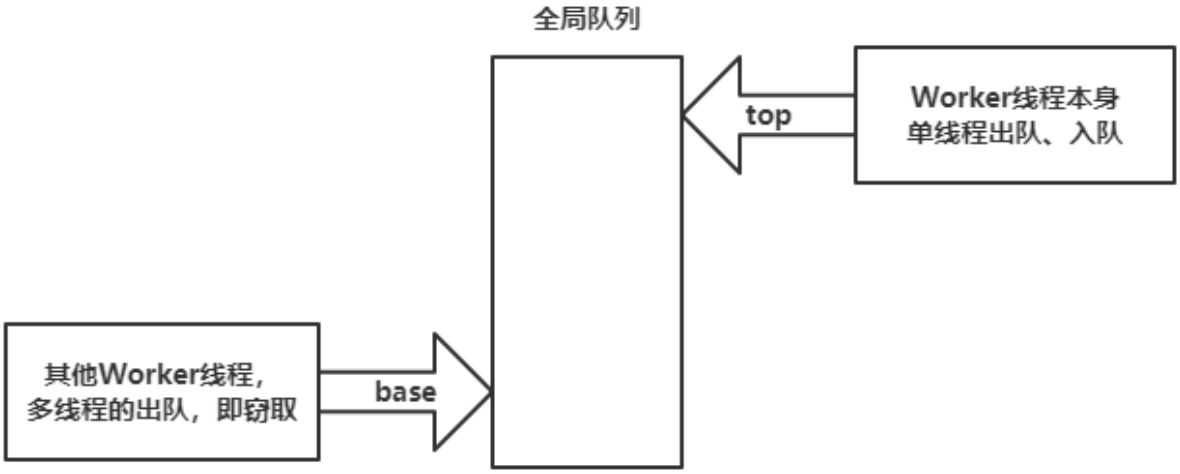
```java
public ForkJoinPool(int parallelism,
                    ForkJoinWorkerThreadFactory factory,
                    UncaughtExceptionHandler handler,
                    boolean asyncMode,
                    int corePoolSize,
                    int maximumPoolSize,
                    int minimumRunnable,
                    Predicate<? super ForkJoinPool> saturate,
                    long keepAliveTime,
                    TimeUnit unit) {
    // check, encode, pack parameters
    if (parallelism <= 0 || parallelism > MAX_CAP ||
        maximumPoolSize < parallelism || keepAliveTime <= 0L)
        throw new IllegalArgumentException();
    if (factory == null)
        throw new NullPointerException();
    long ms = Math.max(unit.toMillis(keepAliveTime), TIMEOUT_SLOP);

    int corep = Math.min(Math.max(corePoolSize, parallelism), MAX_CAP);
    long c = ((((long)(-corep)       << TC_SHIFT) & TC_MASK) |
              (((long)(-parallelism) << RC_SHIFT) & RC_MASK));
    int m = parallelism | (asyncMode ? FIFO : 0);
    int maxSpares = Math.min(maximumPoolSize, MAX_CAP) - parallelism;
    int minAvail = Math.min(Math.max(minimumRunnable, 0), MAX_CAP);
    int b = ((minAvail - parallelism) & SMASK) | (maxSpares << SWIDTH);
    //
    int n = (parallelism > 1) ? parallelism - 1 : 1; // at least 2 slots
    n |= n >>> 1; n |= n >>> 2; n |= n >>> 4; n |= n >>> 8; n |= n >>> 16;
    n = (n + 1) << 1; // power of two, including space for submission queues

    // 工作线程名称前缀
    this.workerNamePrefix = "ForkJoinPool-" + nextPoolId() + "-worker-";
    // 初始化工作线程数组为n，2的幂次方
    this.workQueues = new WorkQueue[n];
    // worker线程工厂，有默认值
    this.factory = factory;
    this.ueh = handler;
    this.saturate = saturate;
    this.keepAlive = ms;
    this.bounds = b;
    this.mode = m;
    // ForkJoinPool的状态
    this.ctl = c;
    checkPermission();
}
```

# 17 工作窃取队列

关于上面的全局队列，有一个关键点需要说明：它并非使用BlockingQueue，而是基于一个普通的数组得以实现。

这个队列又名工作窃取队列，为 ForkJoinPool 的工作窃取算法提供服务。在 ForkJoinPool开篇的注释中，Doug Lea 特别提到了工作窃取队列的实现，其陈述来自如下两篇论文：＂Dynamic Circular Work-Stealing Deque＂ by Chase and Lev，SPAA 2005与＂Idempotent work stealing＂ by Michael，Saraswat，and Vechev，PPoPP 2009。读者可以在网上查阅相应论文。

所谓工作窃取算法，是指一个Worker线程在执行完毕自己队列中的任务之后，可以窃取其他线程队列中的任务来执行，从而实现负载均衡，以防有的线程很空闲，有的线程很忙。这个过程要用到工作窃取队列。




这个队列只有如下几个操作：

1. Worker线程自己，在队列头部，通过对top指针执行加、减操作，实现入队或出队，这是单线程的。
2. 其他Worker线程，在队列尾部，通过对base进行累加，实现出队操作，也就是窃取，这是多线程的，需要通过CAS操作。

这个队列，在*Dynamic Circular Work-Stealing Deque*这篇论文中被称为dynamic-cyclic-array。之所以这样命名，是因为有两个关键点：

1. 整个队列是环形的，也就是一个数组实现的RingBuffer。并且base会一直累加，不会减小；top会累加、减小。最后，base、top的值都会大于整个数组的长度，只是计算数组下标的时候，会取top&（queue.length-1），base&（queue.length-1）。因为queue.length是2的整数次方，这里也就是对queue.length进行取模操作。

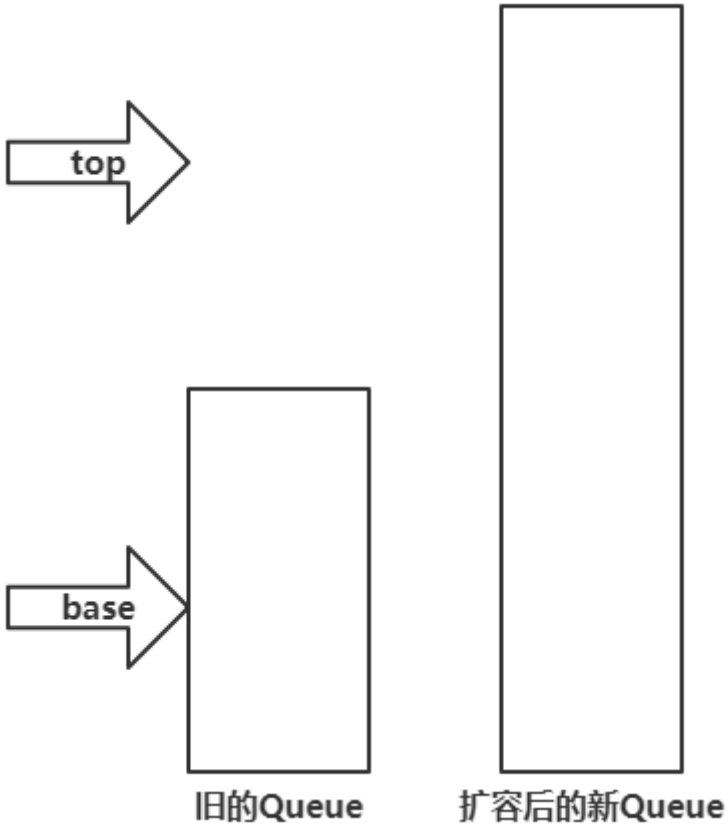
   当top-base=queue.length-1 的时候，队列为满，此时需要扩容；

   当top=base的时候，队列为空，Worker线程即将进入阻塞状态。

2. 当队列满了之后会扩容，所以被称为是动态的。但这就涉及一个棘手的问题：多个线程同时在读写这个队列，如何实现在不加锁的情况下一边读写、一边扩容呢？

通过分析工作窃取队列的特性，我们会发现：在 base 一端，是多线程访问的，但它们只会使base变大，也就是使队列中的元素变少。所以队列为满，一定发生在top一端，对top进行累加的时候，这一端却是单线程的！队列的扩容恰好利用了这个单线程的特性！即在扩容过程中，不可能有其他线程对top进行修改，只有线程对base进行修改！

下图为工作窃取队列扩容示意图。扩容之后，数组长度变成之前的二倍，但top、base的值是不变的！通过top、base对新的数组长度取模，仍然可以定位到元素在新数组中的位置。




下面结合WorkQueue扩容的代码进一步分析。

ForkJoinPool.java × ForkJoinWorkerThread.java ×

```
2466        public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) {
2467            return externalSubmit(task);
2468        }
```

ForkJoinPool.java × ForkJoinWorkerThread.java ×

```
1912    private <T> ForkJoinTask<T> externalSubmit(ForkJoinTask<T> task) {
1913        Thread t; ForkJoinWorkerThread w; WorkQueue q;
1914        if (task == null)
1915            throw new NullPointerException();
1916        if (((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) &&
1917            (w = (ForkJoinWorkerThread)t).pool == this &&
1918            (q = w.workQueue) != null)
1919            q.push(task);
1920        else
1921            externalPush(task);
1922        return task;
1923    }
```

ForkJoinPool.java | ForkJoinWorkerThread.java

```
842         final void push(ForkJoinTask<?> task) {
843             ForkJoinTask<?>[] a;
844             int s = top, d, cap, m;
845             ForkJoinPool p = pool;
846             if ((a = array) != null && (cap = a.length) > 0) {
847                 QA.setRelease(a, (m = cap - 1) & s, task);
848                 top = s + 1;
849                 if (((d = s - (int)BASE.getAcquire( ...args: this)) & ~1) == 0 &&
850                     p != null) {                    // size 0 or 1
851                     VarHandle.fullFence();
852                     p.signalWork();
853                 }
854                 else if (d == m)
855                     growArray( locked: false);
856             }
857         }
```

```
1   final void growArray(boolean locked) {
2       ForkJoinTask<?>[] newA = null;
3       try {
4           ForkJoinTask<?>[] oldA; int oldSize, newSize;
5           // 当旧的array不是null，旧的array包含元素
6           // 并且新的数组长度小于队列最大长度，并且新的长度大于0
7           if ((oldA = array) != null && (oldSize = oldA.length) > 0 &&
8               (newSize = oldSize << 1) <= MAXIMUM_QUEUE_CAPACITY &&
9               newSize > 0) {
10              try {
11                  // 创建新数组
12                  newA = new ForkJoinTask<?>[newSize];
13              } catch (OutOfMemoryError ex) {
14              }
15              if (newA != null) { // poll from old array, push to new
16                  int oldMask = oldSize - 1, newMask = newSize - 1;
17                  for (int s = top - 1, k = oldMask; k >= 0; --k) {
18                      // 逐个复制
19                      ForkJoinTask<?> x = (ForkJoinTask<?>)
20                          // 获取旧的值，将原来的设置为null
21                          QA.getAndSet(oldA, s & oldMask, null);
22                      if (x != null)
23                          newA[s-- & newMask] = x;
24                      else
25                          break;
26                  }
27                  array = newA;
28                  VarHandle.releaseFence();
29              }
30          }
31      } finally {
32          if (locked)
33              phase = 0;
34      }
35      if (newA == null)
36          throw new RejectedExecutionException("Queue capacity exceeded");
```

```
37 }
```

# 18 ForkJoinPool状态控制

## 18.1 状态变量ctl解析

类似于ThreadPoolExecutor，在ForkJoinPool中也有一个ctl变量负责表达ForkJoinPool的整个生命周期和相关的各种状态。不过ctl变量更加复杂，是一个long型变量，代码如下所示。

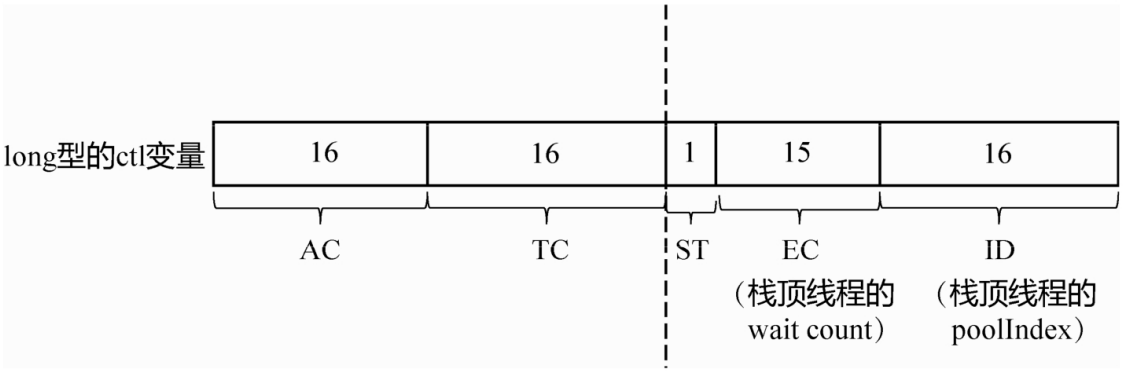
```
1  public class ForkJoinPool extends AbstractExecutorService {
2      // ...
3      // 线程池状态变量
4      volatile long ctl;
5      private static final long SP_MASK    = 0xffffffffL;
6      private static final long UC_MASK    = ~SP_MASK;
7      private static final int  RC_SHIFT   = 48;
8      private static final long RC_UNIT    = 0x0001L << RC_SHIFT;
9      private static final long RC_MASK    = 0xffffL << RC_SHIFT;
10     private static final int  TC_SHIFT   = 32;
11     private static final long TC_UNIT    = 0x0001L << TC_SHIFT;
12     private static final long TC_MASK    = 0xffffL << TC_SHIFT;
13     private static final long ADD_WORKER = 0x0001L << (TC_SHIFT + 15); //
   sign
14
15     // ...
16 }
```

ForkJoinPool.java × ForkJoinWorkerThread.java ×

```
2300        int corep = Math.min(Math.max(corePoolSize, parallelism), MAX_CAP);
2301        long c = ((((long)(-corep)       << TC_SHIFT) & TC_MASK) |
2302                  (((long)(-parallelism) << RC_SHIFT) & RC_MASK));
2303        int m = parallelism | (asyncMode ? FIFO : 0);
```

ctl变量的64个比特位被分成五部分：

1. AC：最高的16个比特位，表示Active线程数-parallelism，parallelism是上面的构造方法传进去的参数；
2. TC：次高的16个比特位，表示Total线程数-parallelism；
3. ST：1个比特位，如果是1，表示整个ForkJoinPool正在关闭；
4. EC：15个比特位，表示阻塞栈的栈顶线程的wait count（关于什么是wait count，接下来解释）；
5. ID：16个比特位，表示阻塞栈的栈顶线程对应的id。



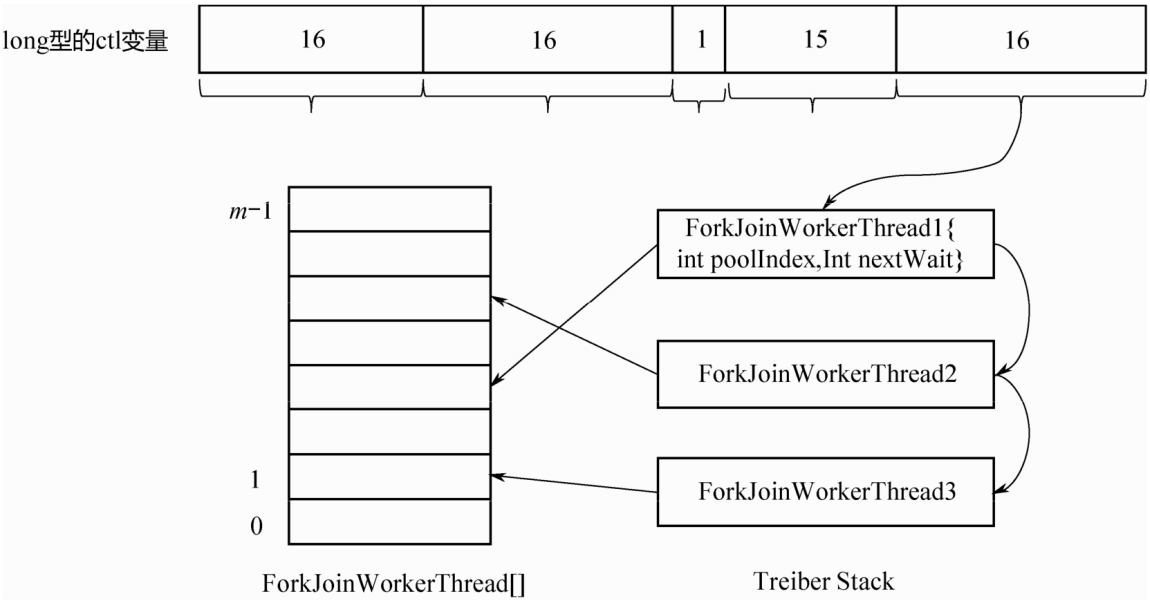

## 18.2 阻塞栈Treiber Stack

什么叫阻塞栈呢?

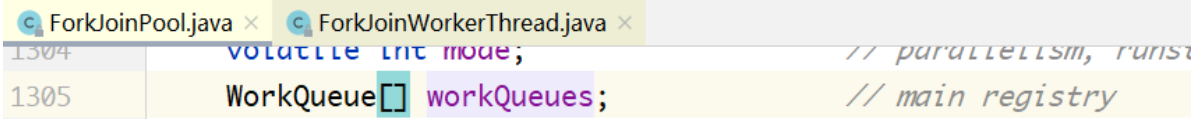
要实现多个线程的阻塞、唤醒，除了park/unpark这一对操作原语，还需要一个**无锁链表**实现的阻塞队列，把所有阻塞的线程串在一起。

在ForkJoinPool中，没有使用阻塞队列，而是使用了阻塞栈。把所有空闲的Worker线程放在一个栈里面，这个栈同样通过链表来实现，名为Treiber Stack。前面讲解Phaser的实现原理的时候，也用过这个数据结构。

下图为所有阻塞的Worker线程组成的Treiber Stack。




首先，WorkQueue有一个id变量，记录了自己在WorkQueue[]数组中的下标位置，id变量就相当于每个WorkQueue或ForkJoinWorkerThread对象的地址；




其次，ForkJoinWorkerThread还有一个stackPred变量，记录了前一个阻塞线程的id，这个stackPred变量就相当于链表的next指针，把所有的阻塞线程串联在一起，组成一个Treiber Stack。

最后，ctl变量的最低16位，记录了栈的栈顶线程的id；中间的15位，记录了栈顶线程被阻塞的次数，也称为wait count。

## 18.3 ctl变量的初始值

构造方法中，有如下的代码：

ForkJoinPool.java | ForkJoinWorkerThread.java

```
2300        int corep = Math.min(Math.max(corePoolSize, parallelism), MAX_CAP);
2301        long c = ((((long)(-corep)       << TC_SHIFT) & TC_MASK) |
2302                  (((long)(-parallelism) << RC_SHIFT) & RC_MASK));
2303        int m = parallelism | (asyncMode ? FIFO : 0);
```

因为在初始的时候，ForkJoinPool 中的线程个数为 0，所以 AC=0-parallelism，TC=0-parallelism。这意味着只有高32位的AC、TC 两个部分填充了值，低32位都是0填充。

## 18.4 ForkJoinWorkerThread状态与个数分析

在ThreadPoolExecutor中，有corePoolSize和maxmiumPoolSize 两个参数联合控制总的线程数，而在ForkJoinPool中只传入了一个parallelism参数，且这个参数并不是实际的线程数。那么，ForkJoinPool在实际的运行过程中，线程数究竟是由哪些因素决定的呢?

要回答这个问题，先得明白ForkJoinPool中的线程都可能有哪几种状态？可能的状态有三种：

1. 空闲状态（放在Treiber Stack里面）。
2. 活跃状态（正在执行某个ForkJoinTask，未阻塞）。
3. 阻塞状态（正在执行某个ForkJoinTask，但阻塞了，于是调用join，等待另外一个任务的结果返回）。

ctl变量很好地反映出了三种状态：

高32位：u=(int) (ctl >>> 32)，然后u又拆分成tc、ac 两个16位；

低32位：c=(int) ctl。

1. c > 0，说明Treiber Stack不为空，有空闲线程；c=0，说明没有空闲线程；
2. ac > 0，说明有活跃线程；ac < =0，说明没有空闲线程，并且还未超出parallelism；
3. tc > 0，说明总线程数 > parallelism。

在提交任务的时候：

ForkJoinPool.java | ThreadLocalRandom.java | ForkJoinWorkerThread.java

```
2466    public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) {
2467        return externalSubmit(task);
2468    }
```

ForkJoinPool.java | ThreadLocalRandom.java | ForkJoinWorkerThread.java

```
1912 @    private <T> ForkJoinTask<T> externalSubmit(ForkJoinTask<T> task) {
1913          Thread t; ForkJoinWorkerThread w; WorkQueue q;
1914          if (task == null)
1915              throw new NullPointerException();
1916          if (((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) &&
1917              (w = (ForkJoinWorkerThread)t).pool == this &&
1918              (q = w.workQueue) != null)
1919              q.push(task);
1920          else
1921              externalPush(task);
1922          return task;
1923      }
```

ForkJoinPool.java | ThreadLocalRandom.java | ForkJoinWorkerThread.java

```
1869      final void externalPush(ForkJoinTask<?> task) {
1870          int r;                                    // initialize caller's
1871          if ((r = ThreadLocalRandom.getProbe()) == 0) {
1872              ThreadLocalRandom.localInit();
```

ForkJoinPool.java | ThreadLocalRandom.java | ForkJoinWorkerThread.java

```
1902                  if (q.lockedPush(task))
1903                      signalWork();
1904                  return;
```

在通知工作线程的时候，需要判断ctl的状态，如果没有闲置的线程，则开启新线程：

ForkJoinPool.java | ThreadLocalRandom.java | ForkJoinWorkerThread.java

```
1469      final void signalWork() {
1470          for (;;) {
1471              long c; int sp; WorkQueue[] ws; int i; WorkQueue v;
1472              if ((c = ctl) >= 0L)                    // enough workers
1473                  break;
1474              else if ((sp = (int)c) == 0) {          // no idle workers
1475                  if ((c & ADD_WORKER) != 0L)         // too few workers
1476                      tryAddWorker(c);
1477                  break;
1478              }
```

# 19 Worker线程的阻塞-唤醒机制

ForkerJoinPool 没有使用 BlockingQueue，也就不利用其阻塞/唤醒机制，而是利用了park/unpark原语，并自行实现了Treiber Stack。

下面进行详细分析ForkerJoinPool，在阻塞和唤醒的时候，分别是如何入栈的。

## 19.1 阻塞–入栈

当一个线程窃取不到任何任务，也就是处于空闲状态时就会阻塞入栈。

ForkJoinPool.java × ForkJoinWorkerThread.java ×

```
172     public void run() {
173         if (workQueue.array == null) { // only run once
174             Throwable exception = null;
175             try {
176                 onStart();
177                 pool.runWorker(workQueue); 调用runWorker启动线程
178             } catch (Throwable ex) {
179                 exception = ex;
180             } finally {
181                 try {
182                     onTermination(exception);
183                 } catch (Throwable ex) {
184                     if (exception == null)
185                         exception = ex;
186                 } finally {
187                     pool.deregisterWorker( wt: this, exception);
188                 }
189             }
190         }
191     }
```

```
1  final void runWorker(WorkQueue w) {
2      // 随机数
3      int r = (w.id ^ ThreadLocalRandom.nextSecondarySeed()) | FIFO; // rng
4      // 初始化任务数组
5      w.array = new ForkJoinTask<?>[INITIAL_QUEUE_CAPACITY];
6      for (;;) {
7          int phase;
8          // 扫描是否有需要执行的一个或多个顶级任务
9          // 其中包含了窃取的任务执行，以及线程局部队列中任务的执行
10         // 如果发现了就执行，返回true
11         // 如果获取不到任务，就需要将该线程入队列，阻塞
12         if (scan(w, r)) {
13             // 随机数
14             r ^= r << 13; r ^= r >>> 17; r ^= r << 5;
15         }
16         // 如果是已经入队列阻塞的，因为phase大于0表示加锁
17         else if ((phase = w.phase) >= 0) {    // enqueue, then rescan
18             long np = (w.phase = (phase + SS_SEQ) | UNSIGNALLED) & SP_MASK;
19             long c, nc;
```

```
            do {
                w.stackPred = (int)(c = ctl);
                // ForkJoinPool中status表示运行中的线程的，数字减一，因为入队列了。
                nc = ((c - RC_UNIT) & UC_MASK) | np;
                // CAS操作，自旋，直到操作成功
            } while (!CTL.weakCompareAndSet(this, c, nc));
        }
        else {                                  // already queued
            int pred = w.stackPred;
            Thread.interrupted();               // clear before park
            w.source = DORMANT;                 // enable signal
            long c = ctl;
            int md = mode, rc = (md & SMASK) + (int)(c >> RC_SHIFT);
            // 如果ForkJoinPool停止，则break，跳出循环
            if (md < 0)
                break;
            // 优雅关闭
            else if (rc <= 0 && (md & SHUTDOWN) != 0 &&
                     tryTerminate(false, false))
                break;
            else if (rc <= 0 && pred != 0 && phase == (int)c) {
                long nc = (UC_MASK & (c - TC_UNIT)) | (SP_MASK & pred);
                long d = keepAlive + System.currentTimeMillis();
                // 线程阻塞，计时等待
                LockSupport.parkUntil(this, d);
                //
                if (ctl == c &&                 // drop on timeout if all idle
                    d - System.currentTimeMillis() <= TIMEOUT_SLOP &&
                    CTL.compareAndSet(this, c, nc)) {
                    // 不再扫描，需要入队列
                    w.phase = QUIET;
                    break;
                }
            }
            // phase为1，表示加锁，phase为负数表示入队列
            else if (w.phase < 0)
                // 如果phase小于0，表示阻塞，排队中
                LockSupport.park(this);
            w.source = 0;
        }
    }
}
```

```
// 从一个队列中扫描一个或多个顶级任务，如果有，就执行
// 对于非空队列，执行任务，返回true
private boolean scan(WorkQueue w, int r) {
    WorkQueue[] ws; int n;
    // 如果workQueues不是null，并且workQueue的长度大于0，并且w非空，w是线程的workQueue
    if ((ws = workQueues) != null && (n = ws.length) > 0 && w != null) {
        // m是ws长度减一，获取ws顶部workQueue
        for (int m = n - 1, j = r & m;;) {
            WorkQueue q; int b;
            // 随机获取workQueue，如果该workQueue的顶指针和底指针不相等，表示有需要执行的任务
```

```
            if ((q = ws[j]) != null && q.top != (b = q.base)) {
                int qid = q.id;
                ForkJoinTask<?>[] a; int cap, k; ForkJoinTask<?> t;
                // 如果workQueue的任务队列不是null，并且元素非空
                if ((a = q.array) != null && (cap = a.length) > 0) {
                    // 获取队列顶部任务
                    t = (ForkJoinTask<?>)QA.getAcquire(a, k = (cap - 1) &
b);
                    // 如果q的base值没有被别的线程修改过，t不是null，并且将t从数组中
移除成功
                    // 即可在当前工作线程执行该任务
                    if (q.base == b++ && t != null &&
                        QA.compareAndSet(a, k, t, null)) {
                        // base+1
                        q.base = b;
                        // 更改source为当前id
                        w.source = qid;
                        // 如果还有任务需要执行，通知其他闲置的线程执行
                        if (q.top - b > 0)
                            signalWork();
                        // 让workQueue中的工作线程来执行不管是窃取来的，还是本地的任
务，还是从queue中获取的其他任务
                        // 公平起见，添加一个随机的边界；剩下的让别的线程来执行。
                        w.topLevelExec(t, q,  // random fairness bound
                                       r & ((n << TOP_BOUND_SHIFT) - 1));
                    }
                }
                return true;
            }
            else if (--n > 0)
                j = (j + 1) & m;
            else
                break;
        }
    }
    return false;
}
```

## 19.2 唤醒-出栈

在新的任务到来之后，空闲的线程被唤醒，其核心逻辑在signalWork方法里面。

```
final void signalWork() {
    for (;;) {
        long c; int sp; WorkQueue[] ws; int i; WorkQueue v;
        if ((c = ctl) >= 0L)                       // 足够的worker线程
            break;
        else if ((sp = (int)c) == 0) {             // 没有闲置的worker线程
            if ((c & ADD_WORKER) != 0L)            // worker线程太少
                tryAddWorker(c); // 尝试添加新的worker线程
            break;
        }
        else if ((ws = workQueues) == null)
            break;                                 // 线程池没有启动或已经停止了
        else if (ws.length <= (i = sp & SMASK))
```

```
            break;                                  // 线程池停止了
        else if ((v = ws[i]) == null)
            break;                                  // 线程池正在停止中
        else {
            int np = sp & ~UNSIGNALLED;
            int vp = v.phase;
            long nc = (v.stackPred & SP_MASK) | (UC_MASK & (c + RC_UNIT));
            Thread vt = v.owner;
            if (sp == vp && CTL.compareAndSet(this, c, nc)) {
                v.phase = np;
                // 如果栈顶元素存在，并且
                if (vt != null && v.source < 0)
                    // 唤醒线程vt
                    LockSupport.unpark(vt);
                break;
            }
        }
    }
}
```

## 20 任务的提交过程分析

在明白了工作窃取队列、ctl变量的各种状态、Worker的各种状态，以及线程阻塞—唤醒机制之后，接下来综合这些知识，详细分析任务的提交和执行过程。

关于任务的提交，ForkJoinPool最外层的接口如下所示。

ForkJoinPool.java

```
2465         */
2466        public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) {
2467            return externalSubmit(task);
2468        }
```

```
/**
 * 将一个可能是外部任务的子任务入队列
 */
private <T> ForkJoinTask<T> externalSubmit(ForkJoinTask<T> task) {
    Thread t; ForkJoinWorkerThread w; WorkQueue q;
    // 任务为null，抛异常
    if (task == null)
        throw new NullPointerException();
    // 如果当前线程是ForkJoinWorkerThread类型的，并且该线程的pool就是当前对象
    // 并且当前pool的workQueue不是null，则将当前任务入队列。
    if (((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) &&
        (w = (ForkJoinWorkerThread)t).pool == this &&
        (q = w.workQueue) != null)
        // 当前任务入队局部队列
        q.push(task);
```

```
    else
        // 否则该任务不是当前线程的子任务，调用外部入队方法，加入全局队列
        externalPush(task);
    return task;
}
```

如何区分一个任务是内部任务，还是外部任务呢?

可以通过调用该方法的线程类型判断。

如果线程类型是ForkJoinWorkerThread，说明是线程池内部的某个线程在调用该方法，则把该任务放入该线程的局部队列;

否则，是外部线程在调用该方法，则将该任务加入全局队列。

## 20.1 内部提交任务push

内部提交任务，即上面的q.push(task)，会放入该线程的工作窃取队列中，代码如下所示。

ForkJoinPool.java

```
842     final void push(ForkJoinTask<?> task) {
843         ForkJoinTask<?>[] a;
844         int s = top, d, cap, m;
845         ForkJoinPool p = pool;
846         if ((a = array) != null && (cap = a.length) > 0) {
847             QA.setRelease(a, (m = cap - 1) & s, task);
848             top = s + 1; 由于是单线程，不需要加锁，直接累加top即可
849             if (((d = s - (int)BASE.getAcquire( ...args: this)) & ~1) == 0 &&
850                 p != null) {                // size 0 or 1
851                 VarHandle.fullFence();
852                 p.signalWork(); 通知空闲线程
853             }
854             else if (d == m)
855                 growArray( locked: false); 队列扩容
856         }
857     }
```

由于工作窃取队列的特性，操作是单线程的，所以此处不需要执行CAS操作。

## 20.2 外部提交任务

```
final void externalPush(ForkJoinTask<?> task) {
    int r;
    // 生成随机数
    if ((r = ThreadLocalRandom.getProbe()) == 0) {
        ThreadLocalRandom.localInit();
        r = ThreadLocalRandom.getProbe();
    }
    for (;;) {
        WorkQueue q;
        int md = mode, n;
        WorkQueue[] ws = workQueues;
        // 如果ForkJoinPool关闭，或者任务队列是null，或者ws的长度小于等于0，拒收任务
```

```
        if ((md & SHUTDOWN) != 0 || ws == null || (n = ws.length) <= 0)
            throw new RejectedExecutionException();

        // 如果随机数计算的workQueues索引处的元素为null，则添加队列
        // 即提交任务的时候，是随机向workQueue中添加workQueue，负载均衡的考虑。
        else if ((q = ws[(n - 1) & r & SQMASK]) == null) { // add queue
            // 计算新workQueue对象的id值
            int qid = (r | QUIET) & ~(FIFO | OWNED);
            // worker线程名称前缀
            Object lock = workerNamePrefix;
            // 创建任务数组
            ForkJoinTask<?>[] qa =
                new ForkJoinTask<?>[INITIAL_QUEUE_CAPACITY];
            // 创建WorkQueue，将当前线程作为
            q = new WorkQueue(this, null);
            // 将任务数组赋值给workQueue
            q.array = qa;
            // 设置workQueue的id值
            q.id = qid;
            // 由于是通过客户端线程添加的workQueue，没有前置workQueue
            // 内部提交任务有源workQueue，表示子任务
            q.source = QUIET;
            if (lock != null) {     // unless disabled, lock pool to install
                synchronized (lock) {
                    WorkQueue[] vs; int i, vn;
                    // 如果workQueues数组不是null，其中有元素，
                    // 并且qid对应的workQueues中的元素为null，则赋值
                    // 因为有可能其他线程将qid对应的workQueues处的元素设置了，
                    // 所以需要加锁，并判断元素是否为null
                    if ((vs = workQueues) != null && (vn = vs.length) > 0 &&
                        vs[i = qid & (vn - 1) & SQMASK] == null)
                        //
                        vs[i] = q;
                }
            }
        }
        // CAS操作，使用随机数
        else if (!q.tryLockPhase()) // move if busy
            r = ThreadLocalRandom.advanceProbe(r);
        else {
            // 如果任务添加成功，通知线程池调度，执行。
            if (q.lockedPush(task))
                signalWork();
            return;
        }
    }
}
```

lockedPush(task)方法的实现：

ForkJoinPool.java | ForkJoinWorkerThread.java

```
862         */
863         final boolean lockedPush(ForkJoinTask<?> task) {
864             ForkJoinTask<?>[] a;
865             boolean signal = false;
866             int s = top, b = base, cap, d;
867             if ((a = array) != null && (cap = a.length) > 0) {
868                 a[(cap - 1) & s] = task;
869                 top = s + 1;
870                 if (b - s + cap - 1 == 0)
871                     growArray( locked: true);
872                 else {
873                     phase = 0; // full volatile unlock
874                     if (((s - base) & ~1) == 0) // size 0 or 1
875                         signal = true;
876                 }
877             }
878             return signal;
879         }
```

外部多个线程会调用该方法，所以要加锁，入队列和扩容的逻辑和线程内部的队列基本相同。最后，调用signalWork()，通知一个空闲线程来取。

## 21 工作窃取算法：任务的执行过程分析

全局队列有任务，局部队列也有任务，每一个Worker线程都会不间断地扫描这些队列，窃取任务来执行。下面从Worker线程的run方法开始分析：

ForkJoinPool.java | ForkJoinTask.java | ForkJoinWorkerThread.java

```
172     public void run() {
173         if (workQueue.array == null) { // only run once
174             Throwable exception = null;
175             try {
176                 onStart();
177                 pool.runWorker(workQueue);
178             } catch (Throwable ex) {
179                 exception = ex;
180             } finally {
181                 try {
182                     onTermination(exception);
183                 } catch (Throwable ex) {
184                     if (exception == null)
185                         exception = ex;
186                 } finally {
187                     pool.deregisterWorker( wt: this, exception);
188                 }
189             }
190         }
191     }
```

调用ForkJoinPoll的runWorker方法

run()方法调用的是所在ForkJoinPool的runWorker方法，如下所示。

```
final void runWorker(WorkQueue w) {
    int r = (w.id ^ ThreadLocalRandom.nextSecondarySeed()) | FIFO; // rng
    w.array = new ForkJoinTask<?>[INITIAL_QUEUE_CAPACITY]; // initialize
    for (;;) {
        int phase;
        if (scan(w, r)) {                     // scan until apparently empty
            r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // move (xorshift)
        }
        else if ((phase = w.phase) >= 0) {    // enqueue, then rescan
            long np = (w.phase = (phase + SS_SEQ) | UNSIGNALLED) & SP_MASK;
            long c, nc;
            do {
                w.stackPred = (int)(c = ctl);
                nc = ((c - RC_UNIT) & UC_MASK) | np;
            } while (!CTL.weakCompareAndSet(this, c, nc));
        }
        else {                                // already queued
            int pred = w.stackPred;
            Thread.interrupted();             // clear before park
            w.source = DORMANT;               // enable signal
            long c = ctl;
            int md = mode, rc = (md & SMASK) + (int)(c >> RC_SHIFT);
            if (md < 0)                       // terminating
                break;
            else if (rc <= 0 && (md & SHUTDOWN) != 0 &&
                     tryTerminate(false, false))
                break;                        // quiescent shutdown
            else if (rc <= 0 && pred != 0 && phase == (int)c) {
                long nc = (UC_MASK & (c - TC_UNIT)) | (SP_MASK & pred);
                long d = keepAlive + System.currentTimeMillis();
                LockSupport.parkUntil(this, d);
                if (ctl == c &&               // drop on timeout if all idle
                    d - System.currentTimeMillis() <= TIMEOUT_SLOP &&
                    CTL.compareAndSet(this, c, nc)) {
                    w.phase = QUIET;
                    break;
                }
            }
            else if (w.phase < 0)
                LockSupport.park(this);       // OK if spuriously woken
            w.source = 0;                     // disable signal
        }
    }
}
```

下面详细看扫描过程scan(w, a)。

```
private boolean scan(WorkQueue w, int r) {
    WorkQueue[] ws; int n;
    if ((ws = workQueues) != null && (n = ws.length) > 0 && w != null) {
        for (int m = n - 1, j = r & m;;) {
```

```
            WorkQueue q; int b;
            if ((q = ws[j]) != null && q.top != (b = q.base)) {
                int qid = q.id;
                ForkJoinTask<?>[] a; int cap, k; ForkJoinTask<?> t;
                if ((a = q.array) != null && (cap = a.length) > 0) {
                    t = (ForkJoinTask<?>)QA.getAcquire(a, k = (cap - 1) &
b);
                    if (q.base == b++ && t != null &&
                        QA.compareAndSet(a, k, t, null)) {
                        q.base = b;
                        w.source = qid;
                        if (q.top - b > 0)
                            signalWork();
                        w.topLevelExec(t, q,  // random fairness bound
                                       r & ((n << TOP_BOUND_SHIFT) - 1));
                    }
                }
                return true;
            }
            else if (--n > 0)
                j = (j + 1) & m;
            else
                break;
        }
    }
    return false;
}
```

## 22 ForkJoinTask的fork/join

如果局部队列、全局中的任务全部是相互独立的，就很简单了。但问题是，对于分治算法来说，分解出来的一个个任务并不是独立的，而是相互依赖，一个任务的完成要依赖另一个前置任务的完成。

这种依赖关系是通过ForkJoinTask中的join()来体现的。且看前面的代码：

```
protected void compute() {
    if (lo < hi) {
        // 分区
        int pivot = partition(array, lo, hi);
        SortTask left = new SortTask(array, lo, pivot - 1);
        SortTask right = new SortTask(array, pivot + 1, hi);
        left.fork();
        right.fork();
        left.join();
        right.join();
    }
}
```

线程在执行当前ForkJoinTask的时候，产生了left、right 两个子Task。

fork是指把这两个子Task放入队列里面；

join则是要等待2个子Task完成。

而子Task在执行过程中，会再次产生两个子Task。如此层层嵌套，类似于递归调用，直到最底层的Task计算完成，再一级级返回。

## 22.1 fork

fork()的代码很简单，就是把自己放入当前线程所在的局部队列中。

如果是外部线程调用fork方法，则直接将任务添加到共享队列中。

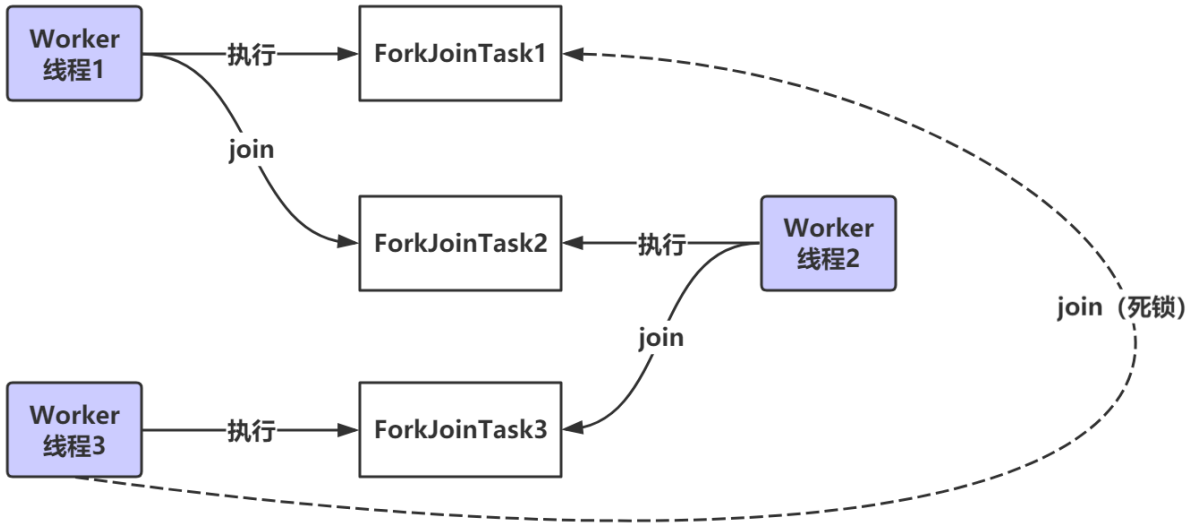
ForkJoinPool.java | ForkJoinTask.java

```
699    public final ForkJoinTask<V> fork() {
700        Thread t;
701        if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
702            ((ForkJoinWorkerThread)t).workQueue.push( task: this);
703        else                                    将任务添加到任务的局部队列中
704            ForkJoinPool.common.externalPush( task: this);
705        return this;
706    }
```

## 22.2 join的嵌套

**1.join的层层嵌套阻塞原理**

join会导致线程的层层嵌套阻塞，如图所示：




线程1在执行 ForkJoinTask1，在执行过程中调用了 forkJoinTask2.join()，所以要等ForkJoinTask2完成，线程1才能返回；

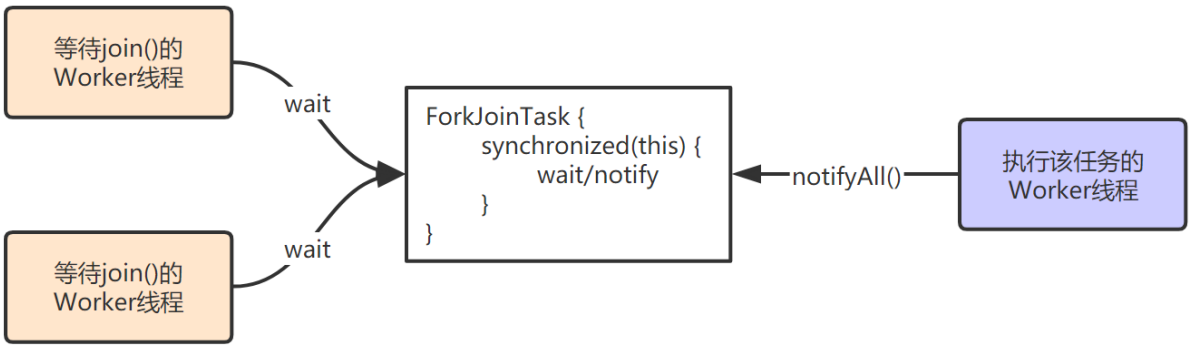
线程2在执行ForkJoinTask2，但由于调用了forkJoinTask3.join()，只有等ForkJoinTask3完成后，线程2才能返回；

线程3在执行ForkJoinTask3。

结果是：线程3首先执行完，然后线程2才能执行完，最后线程1再执行完。所有的任务其实组成一个有向无环图DAG。如果线程3调用了forkJoinTask1.join()，那么会形成环，造成死锁。

那么，这种层次依赖、层次通知的 DAG，在 ForkJoinTask 内部是如何实现的呢？站在ForkJoinTask的角度来看，每个ForkJoinTask，都可能有多个线程在等待它完成，有1个线程在执行它。所以每个ForkJoinTask就是一个同步对象，线程在调用join()的时候，阻塞在这个同步对象上面，执行完成之后，再通过这个同步对象通知所有等待的线程。

利用synchronized关键字和Java原生的wait()/notify()机制，实现了线程的等待-唤醒机制。调用join()的这些线程，内部其实是调用ForkJoinTask这个对象的wait()；执行该任务的Worker线程，在任务执行完毕之后，顺便调用notifyAll()。




**2.ForkJoinTask的状态解析**

要实现fork()/join()的这种线程间的同步，对应的ForkJoinTask一定是有各种状态的，这个状态变量是实现fork/join的基础。

```
public abstract class ForkJoinTask<V> implements Future<V>, Serializable {
    // ...
    // 由ForkJoinPool和workers直接调用
    // 需要是volatile的
    volatile int status;
    private static final int DONE      = 1 << 31; // 负值
    private static final int ABNORMAL  = 1 << 18; // 设置DONE的时候自动设置
    private static final int THROWN    = 1 << 17; // 设置ABNORMAL的时候自动设置
    private static final int SIGNAL    = 1 << 16; // 如果在调用join的线程正在等待，则为true
    private static final int SMASK     = 0xffff;  // short bits for tags
    // ...
}
```

初始时，status=0。共有五种状态，可以分为两大类：

1. 未完成：status > =0。
2. 已完成：status < 0。

所以，通过判断是status > =0，还是status < 0，就可知道任务是否完成，进而决定调用join()的线程是否需要被阻塞。

**3.join的详细实现**

下面看一下代码的详细实现。

```
ForkJoinPool.java | ForkJoinTask.java | ForkJoinWorkerThread.java
719     public final V join() {
720         int s;
721         if (((s = doJoin()) & ABNORMAL) != 0)非正常结束，抛异常
722             reportException(s);
723         return getRawResult(); 正常结束，返回结果
724     }
```

getRawResult()是ForkJoinTask中的一个模板方法，分别被RecursiveAction和RecursiveTask实现，前者没有返回值，所以返回null，后者返回一个类型为V的result变量。

```
ForkJoinPool.java | ForkJoinTask.java | RecursiveAction.java | ForkJoinWorkerThread.java
178     public final Void getRawResult() { return null; }
179
```

```
ForkJoinPool.java | RecursiveTask.java | ForkJoinTask.java | RecursiveAction.java
81
82      public final V getRawResult() {
83          return result;
84      }
85
86      protected final void setRawResult(V value) {
87          result = value;
88      }
```

阻塞主要发生在上面的doJoin()方法里面。在dojoin()里调用t.join()的线程会阻塞，然后等待任务t执行完成，再唤醒该阻塞线程，doJoin()返回。

**注意：**当 doJoin()返回的时候，就是该任务执行完成的时候，doJoin()的返回值就是任务的完成状态，也就是上面的几种状态。

```java
public abstract class ForkJoinTask<V> implements Future<V>, Serializable {
    // ...
    private int doJoin() {
        int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
        // 如果status < 0表示任务已经完成，不用阻塞，直接返回。
        return (s = status) < 0 ? s :
            // 否则判断线程是否是工作线程
            ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) ?
            (w = (wt = (ForkJoinWorkerThread)t).workQueue).
            tryUnpush(this) && (s = doExec()) < 0 ? s :
            wt.pool.awaitJoin(w, this, 0L) :
            externalAwaitDone();
    }
    // ...
```

```
}
```

上面的返回值可读性比较差，变形之后：

```
// 如果status < 0，直接返回s值
if ((s = status) < 0) {
    return s;
} else {
    // 如果线程是工作线程
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
        // 将任务能够从局部队列弹出，并调用doExec()方法执行成功
        if (w = (wt = (ForkJoinWorkerThread)t).workQueue).tryUnpush(this) &&
(s = doExec()) < 0) {
            // 返回s值
            return s;
        } else {
            // 否则等待，线程阻塞
            wt.pool.awaitJoin(w, this, 0L)
        }
    } else {
        // 如果线程不是工作线程，则外部等待任务完成，线程阻塞
        externalAwaitDone();
    }
}
```

先看一下externalAwaitDone()，即外部线程的阻塞过程，相对简单。

```
private int externalAwaitDone() {
    int s = tryExternalHelp();
    if (s >= 0 && (s = (int)STATUS.getAndBitwiseOr(this, SIGNAL)) >= 0) {
        boolean interrupted = false;
        synchronized (this) {
            for (;;) {
                if ((s = status) >= 0) {
                    try {
                        // 如果任务还没有完成，阻塞
                        wait(0L);
                    } catch (InterruptedException ie) {
                        interrupted = true;
                    }
                }
                else {
                    // 唤醒线程，开始执行
                    notifyAll();
                    break;
                }
            }
        }
        if (interrupted)
            Thread.currentThread().interrupt();
    }
    return s;
}
```

内部Worker线程的阻塞，即上面的wt.pool.awaitJoin(w, this, 0L)，相比外部线程的阻塞要做更多工作。它现不在ForkJoinTask里面，而是在ForkJoinWorkerThread里面。

```
final int awaitJoin(WorkQueue w, ForkJoinTask<?> task, long deadline) {
    int s = 0;
    int seed = ThreadLocalRandom.nextSecondarySeed();
    if (w != null && task != null &&
        (!(task instanceof CountedCompleter) ||
         (s = w.helpCC((CountedCompleter<?>)task, 0, false)) >= 0)) {
        // 尝试执行该任务
        w.tryRemoveAndExec(task);
        int src = w.source, id = w.id;
        int r = (seed >>> 16) | 1, step = (seed & ~1) | 2;
        s = task.status;
        while (s >= 0) {
            WorkQueue[] ws;
            int n = (ws = workQueues) == null ? 0 : ws.length, m = n - 1;
            while (n > 0) {
                WorkQueue q; int b;
                if ((q = ws[r & m]) != null && q.source == id &&
                    q.top != (b = q.base)) {
                    ForkJoinTask<?>[] a; int cap, k;
                    int qid = q.id;
                    if ((a = q.array) != null && (cap = a.length) > 0) {
                        ForkJoinTask<?> t = (ForkJoinTask<?>)
                            QA.getAcquire(a, k = (cap - 1) & b);
                        if (q.source == id && q.base == b++ &&
                            t != null && QA.compareAndSet(a, k, t, null)) {
                            q.base = b;
                            w.source = qid;
                            // 执行该任务
                            t.doExec();
                            w.source = src;
                        }
                    }
                    break;
                }
                else {
                    r += step;
                    --n;
                }
            }
            // 如果任务的status < 0，任务执行完成，则退出循环，返回s的值
            if ((s = task.status) < 0)
                break;
            else if (n == 0) { // empty scan
                long ms, ns; int block;
                if (deadline == 0L)
                    ms = 0L;                       // untimed
                else if ((ns = deadline - System.nanoTime()) <= 0L)
                    break;                         // timeout
                else if ((ms = TimeUnit.NANOSECONDS.toMillis(ns)) <= 0L)
                    ms = 1L;                       // avoid 0 for timed wait
                if ((block = tryCompensate(w)) != 0) {
                    task.internalWait(ms);
```

```
                    CTL.getAndAdd(this, (block > 0) ? RC_UNIT : 0L);
                }
                s = task.status;
            }
        }
    }
    return s;
}
```

上面的方法有个关键点：for里面是死循环，并且只有一个返回点，即只有在task.status < 0，任务完成之后才可能返回。否则会不断自旋；若自旋之后还不行，就会调用task.internalWait(ms);阻塞。

task.internalWait(ms);的代码如下。

ForkJoinPool.java | ForkJoinTask.java | ForkJoinWorkerThread.java

```
306
307     final void internalWait(long timeout) {
308         if ((int)STATUS.getAndBitwiseOr( ...args: this, SIGNAL) >= 0) {
309             synchronized (this) {
310                 if (status >= 0)
311                     try { wait(timeout); } catch (InterruptedException ie) { }
312                 else
313                     notifyAll();
314             }
315         }
316     }
```

**4.join的唤醒**

调用t.join()之后，线程会被阻塞。接下来看另外一个线程在任务**t**执行完毕后如何唤醒阻塞的线程。

ForkJoinPool.java | ForkJoinTask.java

```
285
286     final int doExec() {
287         int s; boolean completed;
288         if ((s = status) >= 0) {
289             try {
290                 completed = exec();
291             } catch (Throwable rex) {
292                 completed = false;
293                 s = setExceptionalCompletion(rex);
294             }
295             if (completed)  任务执行完更新status，同时通知其他阻塞的线程
296                 s = setDone();
297         }
298         return s;
299     }
```

```
ForkJoinPool.java  ForkJoinTask.java
254     private int setDone() {
255         int s;
256         if (((s = (int)STATUS.getAndBitwiseOr( ...args: this, DONE)) & SIGNAL) != 0)
257             synchronized (this) { notifyAll(); } 通知所有被阻塞的线程
258         return s | DONE;
259     }
```

任务的执行发生在doExec()方法里面，任务执行完成后，调用一个setDone()通知所有等待的线程。这里也做了两件事：

1. 把status置为完成状态。
2. 如果s != 0，即 s = SIGNAL，说明有线程正在等待这个任务执行完成。调用Java原生的notifyAll()通知所有线程。如果s = 0，说明没有线程等待这个任务，不需要通知。

# 23 ForkJoinPool的优雅关闭

同ThreadPoolExecutor一样，ForkJoinPool的关闭也不可能是“瞬时的”，而是需要一个平滑的过渡过程。

## 23.1 工作线程的退出

对于一个Worker线程来说，它会在一个for循环里面不断轮询队列中的任务，如果有任务，则执行，处在活跃状态；如果没有任务，则进入空闲等待状态。

这个线程如何退出呢?

```
/**
 * 工作线程的顶级循环，通过ForkJoinWorkerThread.run调用
 */
final void runWorker(WorkQueue w) {
    int r = (w.id ^ ThreadLocalRandom.nextSecondarySeed()) | FIFO; // rng
    w.array = new ForkJoinTask<?>[INITIAL_QUEUE_CAPACITY]; // 初始化任务数组。
    for (;;) {
        int phase;
        if (scan(w, r)) {                     // scan until apparently empty
            r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // move (xorshift)
        }
        else if ((phase = w.phase) >= 0) {    // enqueue, then rescan
            long np = (w.phase = (phase + SS_SEQ) | UNSIGNALLED) & SP_MASK;
            long c, nc;
            do {
                w.stackPred = (int)(c = ctl);
                nc = ((c - RC_UNIT) & UC_MASK) | np;
            } while (!CTL.weakCompareAndSet(this, c, nc));
        }
        else {                                // already queued
            int pred = w.stackPred;
            Thread.interrupted();             // clear before park
            w.source = DORMANT;               // enable signal
            long c = ctl;
            int md = mode, rc = (md & SMASK) + (int)(c >> RC_SHIFT);
```

```
            if (md < 0)                         // terminating
                break;
            // 优雅退出
            else if (rc <= 0 && (md & SHUTDOWN) != 0 &&
                     tryTerminate(false, false))
                break;
            else if (rc <= 0 && pred != 0 && phase == (int)c) {
                long nc = (UC_MASK & (c - TC_UNIT)) | (SP_MASK & pred);
                long d = keepAlive + System.currentTimeMillis();
                LockSupport.parkUntil(this, d);
                if (ctl == c &&                 // drop on timeout if all idle
                    d - System.currentTimeMillis() <= TIMEOUT_SLOP &&
                    CTL.compareAndSet(this, c, nc)) {
                    w.phase = QUIET;
                    break;
                }
            }
            else if (w.phase < 0)
                LockSupport.park(this);         // OK if spuriously woken
            w.source = 0;                       // disable signal
        }
    }
}
```

(int) (c = ctl) < 0，即低32位的最高位为1，说明线程池已经进入了关闭状态。但线程池进入关闭状态，不代表所有的线程都会立马关闭。

## 23.2 shutdown()与shutdownNow()的区别

```
public void shutdown() {
    checkPermission();
    tryTerminate(false, true);
}

public List<Runnable> shutdownNow() {
    checkPermission();
    tryTerminate(true, true);
    return Collections.emptyList();
}
```

二者的代码基本相同，都是调用tryTerminate(boolean, boolean)方法，其中一个传入的是false，另一个传入的是true。tryTerminate意为试图关闭ForkJoinPool，并不保证一定可以关闭成功：

```
private boolean tryTerminate(boolean now, boolean enable) {
    int md; // 三个阶段：尝试设置为SHUTDOWN，之后STOP，最后TERMINATED

    while (((md = mode) & SHUTDOWN) == 0) {
        if (!enable || this == common)          // cannot shutdown
            return false;
        else
            // 将mode变量CAS操作设置为SHUTDOWN
```

```
            MODE.compareAndSet(this, md, md | SHUTDOWN);
    }

    while (((md = mode) & STOP) == 0) {       // try to initiate termination
        if (!now) {                           // check if quiescent & empty
            for (long oldSum = 0L;;) {        // repeat until stable
                boolean running = false;
                long checkSum = ctl;
                WorkQueue[] ws = workQueues;
                if ((md & SMASK) + (int)(checkSum >> RC_SHIFT) > 0)
                    // 还有正在运行的线程
                    running = true;
                else if (ws != null) {
                    WorkQueue w;
                    for (int i = 0; i < ws.length; ++i) {
                        if ((w = ws[i]) != null) {
                            int s = w.source, p = w.phase;
                            int d = w.id, b = w.base;
                            if (b != w.top ||
                                ((d & 1) == 1 && (s >= 0 || p >= 0))) {
                                running = true;
                                // 还正在运行
                                break;
                            }
                            checkSum += (((long)s << 48) + ((long)p << 32) +
                                         ((long)b << 16) + (long)d);
                        }
                    }
                }
                if (((md = mode) & STOP) != 0)
                    break;                    // already triggered
                else if (running)
                    return false;
                else if (workQueues == ws && oldSum == (oldSum = checkSum))
                    break;
            }
        }
        if ((md & STOP) == 0)
            // 如果需要立即停止，同时md没有设置为STOP，则设置为STOP
            MODE.compareAndSet(this, md, md | STOP);
    }

    // 如果mode还没有设置为TERMINATED，则进行循环
    while (((md = mode) & TERMINATED) == 0) { // help terminate others
        for (long oldSum = 0L;;) {            // repeat until stable
            WorkQueue[] ws; WorkQueue w;
            long checkSum = ctl;
            if ((ws = workQueues) != null) {
                for (int i = 0; i < ws.length; ++i) {
                    if ((w = ws[i]) != null) {
                        ForkJoinWorkerThread wt = w.owner;
                        // 清空任务队列
                        w.cancelAll();
                        if (wt != null) {
                            try {
                                // 中断join或park的线程
                                wt.interrupt();
                            } catch (Throwable ignore) {
```

```
                                }
                            }
                            checkSum += ((long)w.phase << 32) + w.base;
                        }
                    }
                }
                // 如果已经设置了TERMINATED，则跳出for循环，while循环条件为false，整个方法返回true，停止
                if (((md = mode) & TERMINATED) != 0 ||
                    (workQueues == ws && oldSum == (oldSum = checkSum)))
                    break;
            }
            if ((md & TERMINATED) != 0)
                break;
            else if ((md & SMASK) + (short)(ctl >>> TC_SHIFT) > 0)
                break;
            else if (MODE.compareAndSet(this, md, md | TERMINATED)) {
                synchronized (this) {
                    // 通知调用awaitTermination的线程，关闭ForkJoinPool了
                    notifyAll();
                }
                break;
            }
        }
        return true;
    }
```

总结：shutdown()只拒绝新提交的任务；shutdownNow()会取消现有的全局队列和局部队列中的任务，同时唤醒所有空闲的线程，让这些线程自动退出。

# 第五部分：多线程设计模式

## 24 Single Threaded Execution模式

所谓Single Threaded Execution模式，指的是“以一个线程执行”。该模式用于设置限制，以确保同一时间只能让一个线程执行处理。

Single Threaded Execution有时也称为临界区（critical section）或临界域（critical region）。Single Threaded Execution名称侧重于执行处理的线程，临界区或临界域侧重于执行范围。

**示例程序**

```java
package com.lagou.concurrent.demo;

public class ConcurrentDemo {

    public static void main(String[] args) {
        NumberResource resource = new NumberResource();
        new UserThread(resource).start();
        new UserThread(resource).start();
    }

}
```

```java
package com.lagou.concurrent.demo;

public class UserThread extends Thread {

    private NumberResource resource;

    public UserThread(NumberResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        while (true) {
            resource.showNumber();
        }
    }
}
```

```java
package com.lagou.concurrent.demo;

public class NumberResource {
    private Integer number = 0;
    private Integer printIndex = 0;

    private void add() {
        this.number++;
    }

    private Integer get() {
        return this.number;
    }

    public void showNumber() throws InterruptedException {
        // 大概打印100次，退出虚拟机
        if (printIndex > 100) System.exit(0);

        Integer number1 = this.get();
        this.add();
        Thread.sleep(5);

```

```
        Integer number2 = this.get();

        if ((number1 + 1) == number2) {
            System.out.println(Thread.currentThread().getName() + " => 递增--正确--: " + number1 + " ***** " + number2);
        } else {
            System.out.println(Thread.currentThread().getName() + " => 递增**异常**: " + number1 + " ***** " + number2);
        }
        // 增加计数
        printIndex++;
    }
}

```

运行效果：

```
Run:  ConcurrentDemo
D:\RunningApps\Java\jdk-11.0.5\bin\java.exe "-javaagent:
Thread-0 => 递增**异常**: 118 ***** 121
Thread-0 => 递增**异常**: 166789361 ***** 166789364
Thread-0 => 递增**异常**: 166804958 ***** 166804966
Thread-0 => 递增**异常**: 166814497 ***** 166814531
Thread-0 => 递增**异常**: 166819037 ***** 166819040
Thread-0 => 递增**异常**: 166823717 ***** 166823725
Thread-0 => 递增**异常**: 166827681 ***** 166827687
Thread-1 => 递增**异常**: 166827688 ***** 166827687
Thread-1 => 递增**异常**: 166828187 ***** 166828189
Thread-1 => 递增**异常**: 166828508 ***** 166828510
Thread-1 => 递增**异常**: 166828876 ***** 166828878
Thread-0 => 递增**异常**: 166828877 ***** 166828879
Thread-0 => 递增**异常**: 166829415 ***** 166829417
Thread-1 => 递增**异常**: 166829821 ***** 166829823
Thread-1 => 递增**异常**: 166830245 ***** 166830247
Thread-0 => 递增**异常**: 166830244 ***** 166830246
Thread-0 => 递增**异常**: 166830829 ***** 166830831
```

上述代码之所以递增异常，是因为showNumber方法是一个临界区，其中对数字加一，但又不能保证原子性，在多线程执行的时候，就会出现问题。

线程安全的NumberResource类：

```
package com.lagou.concurrent.demo;

public class NumberResource {
    private Integer number = 0;
    private Integer printIndex = 0;
```

```
6
7       private void add() {
8           this.number++;
9       }
10
11      private Integer get() {
12          return this.number;
13      }
14
15      public synchronized void showNumber() throws InterruptedException {
16          // 大概打印100次，退出虚拟机
17          if (printIndex > 100) System.exit(0);
18
19          Integer number1 = this.get();
20          this.add();
21          Thread.sleep(5);
22
23          Integer number2 = this.get();
24
25          if ((number1 + 1) == number2) {
26              System.out.println(Thread.currentThread().getName() + " => 递增--
    正确--: " + number1 + " ***** " + number2);
27          } else {
28              System.out.println(Thread.currentThread().getName() + " => 递增**
    异常**: " + number1 + " ***** " + number2);
29          }
30          // 增加计数
31          printIndex++;
32      }
33  }
34
```

**Single Threaded Execution模式总结**

1. SharedResource（共享资源）

Single Threaded Execution模式中出现了一个发挥SharedResource（共享资源）作用的类。在示例程序中，由NumberResource类扮演SharedResource角色。

SharedResource角色是可以被多个线程访问的类，包含很多方法，但这些方法主要分为如下两类：

- safeMethod：多个线程同时调用也不会发生问题的方法。
- unsafeMethod：多个线程同时访问会发生问题，因此必须加以保护的方法。

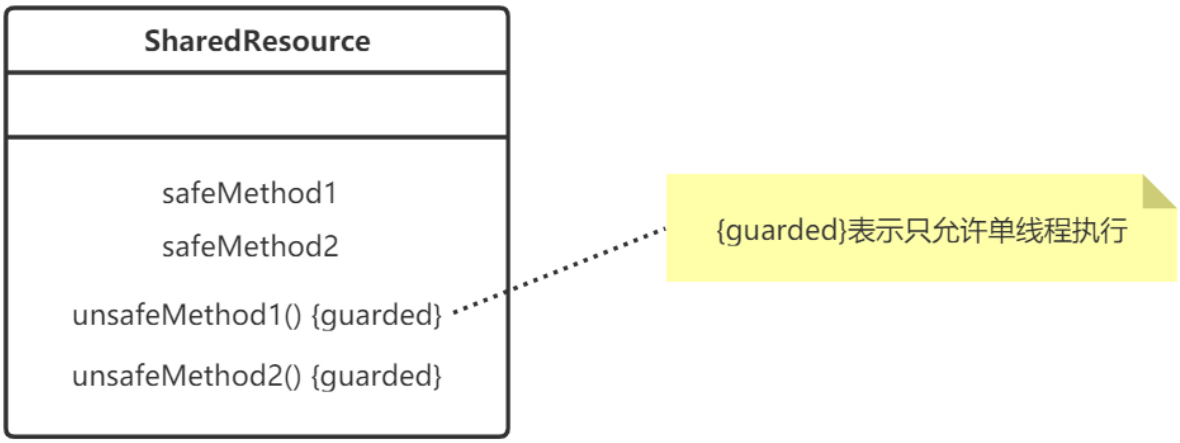
safeMethod，无需考虑。

对于unsafeMethod，在被多个线程同时执行时，实例状态有可能发生分歧。这时就需要保护该方法，使其不被多个线程同时访问。

Single Threaded Execution模式会保护unsafeMethod，使其同时只能由一个线程访问。java则是通过unsafeMethod声明为synchronized方法来进行保护。

我们将只允许单个线程执行的程序范围称为临界区。

**Single Threaded Execution类图**




**何时使用Single Threaded Execution模式**

1. 多线程时

在单线程程序中使用synchronized关键字并不会破坏程序的安全性，但是调用synchronized方法要比调用一般方法花费时间，稍微降低程序性能。

2. 多个线程访问时

当SharedResource角色的实例有可能被多个线程同时访问时，就需要使用Single Threaded Execution模式。

即便是多线程程序，如果所有线程都是完全独立操作的，也无需使用Single Threaded Execution模式。这种状态称为线程互不干涉。

在某些处理多个线程的框架中，有时线程的独立性是由框架控制的。此时，框架的使用者就无需考虑是否使用Single Threaded Execution模式。

3. 状态有可能变化时

之所以需要使用Single Threaded Execution模式，是因为SharedResource角色的状态会发生变化。

如果在创建实例后，实例的状态再也不发生变化，就无需使用Single Threaded Execution模式。

4. 需要确保安全性时

只有在需要确保安全性时，才需要使用Single Threaded Execution模式。

Java的集合类大多数都是非线程安全的。这是为了在不需要考虑安全性的时候提高程序运行速度。

用户在使用类时，需要考虑自己要用的类是否时线程安全的。

**死锁**

使用Single Threaded Execution模式时，存在发生死锁的危险。

死锁是指两个线程分别持有锁，并相互等待对方释放锁的现象。发生死锁的线程都无法再继续运行，程序卡死。

两个人吃饭，都需要刀和叉，但刀叉又只有一套。某时刻，其中一个人拿了刀，另一个拿了叉，而且两人都在等待对方让出自己需要的叉或刀。这种情形下，两个人都只能一直等待下去，这就是发生了死锁。

在Single Threaded Execution模式中，满足下列条件时，会发生死锁：

- 存在多个SharedResource角色
- 线程在持有某个SharedResource角色锁的同时，还想获取其他SharedResource角色的锁
- 获取SharedResource角色的锁的顺序不固定（SharedResource角色是对称的）

**临界区的大小和性能**

一般情况下，Single Threaded Execution模式会降低程序性能：

1. 获取锁花费时间

进入synchronized方法时，线程需要获取对象的锁，该处理会花费时间。

如果SharedResource角色的数量减少了，那么要获取的锁的数量也会相应地减少，从而就能够抑制性能的下降了。

2. 线程冲突引起的等待

当线程执行临界区内的处理时，其他想要进入临界区的线程会阻塞。这种状况称为线程冲突。发生冲突时，程序的整体性能会随着线程等待时间的增加而下降。

# 25 Immutable模式

Immutable就是不变的、不发生改变。Immutable模式中存在着确保实例状态不发生改变的类。在访问这些实例时不需要执行耗时的互斥处理。如果能用好该模式，就可以提高程序性能。

如String就是一个不可变类，immutable的。

**示例程序**

```
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        User user = new User(1001, "张三", "张三是一个好人");
        new UserThread(user).start();
        new UserThread(user).start();
        new UserThread(user).start();
        new UserThread(user).start();
        new UserThread(user).start();
    }
}
```

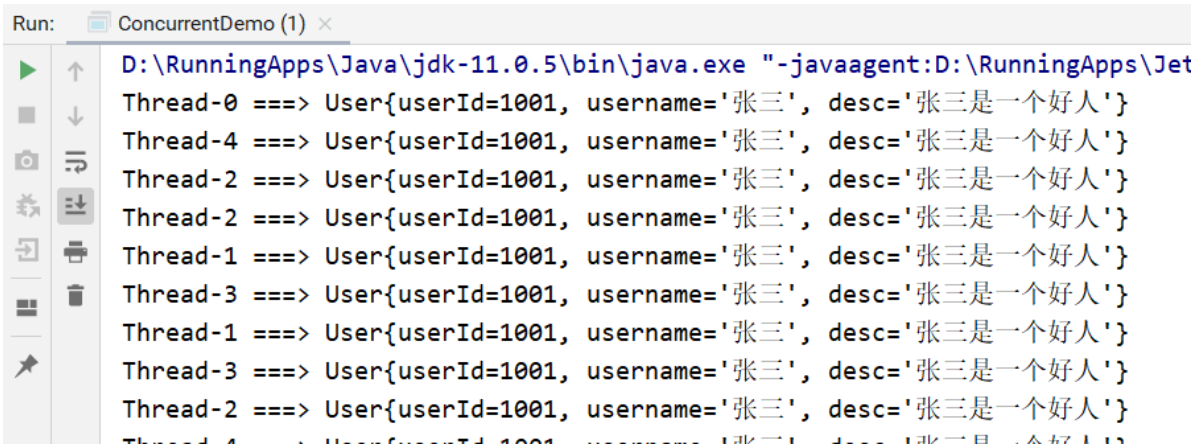
```
package com.lagou.concurrent.demo;
```

```
2
3  public class User {
4      private final Integer userId;
5      private final String username;
6      private final String desc;
7
8      public User(Integer userId, String username, String desc) {
9          this.userId = userId;
10         this.username = username;
11         this.desc = desc;
12     }
13
14     public Integer getUserId() {
15         return userId;
16     }
17
18     public String getUsername() {
19         return username;
20     }
21
22     public String getDesc() {
23         return desc;
24     }
25
26     @Override
27     public String toString() {
28         return "User{" +
29                 "userId=" + userId +
30                 ", username='" + username + '\'' +
31                 ", desc='" + desc + '\'' +
32                 '}';
33     }
34 }
```

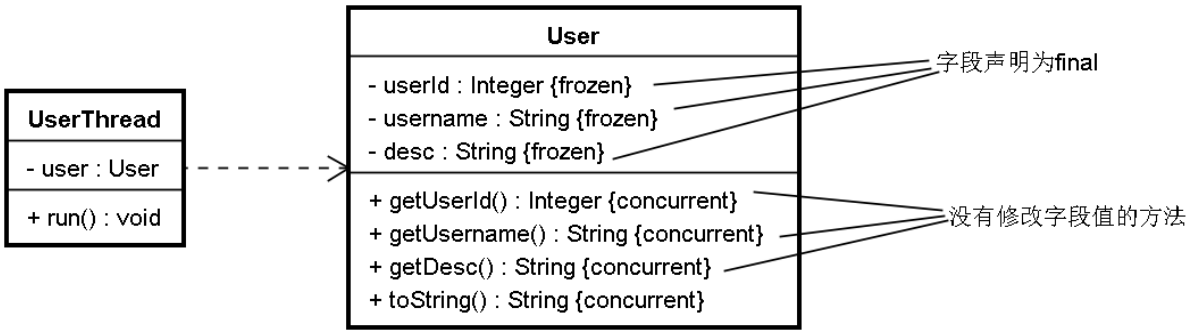
```
1  package com.lagou.concurrent.demo;
2
3  public class UserThread extends Thread {
4
5      private Integer index = 0;
6
7      private User user;
8
9      public UserThread(User user) {
10         this.user = user;
11     }
12
13     @Override
14     public void run() {
15         while (true) {
16
17             if (index >= 100) {
18                 System.exit(0);
19             }
20
```

```
21              System.out.println(Thread.currentThread().getName() + " ===> " +
user);
22              index++;
23          }
24      }
25  }
```

执行效果：




**类图**



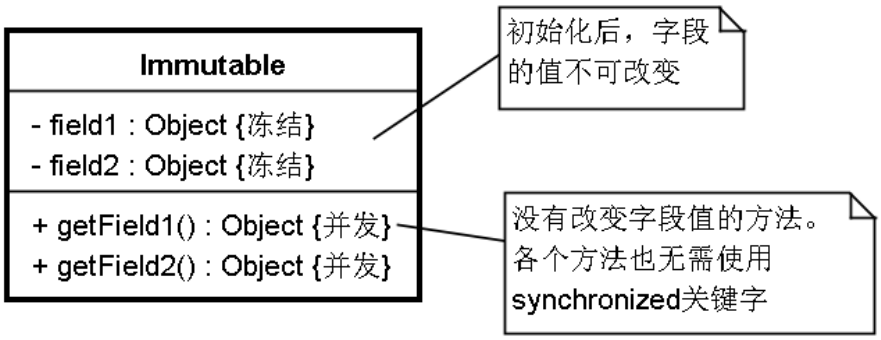

在Single Threaded Execution模式，将修改或引用实例状态的地方设置为临界区，该区只能由一个线程执行。对于本案例的User类，实例的状态绝对不会发生改变，即使多个线程同时对该实例执行处理，实例也不会出错，因为实例的状态不变。如此也无需使用synchronized关键字来保护实例。

**Immutable模式中的角色**

1. Immutable

Immutable角色是一个类，该角色中的字段值不可修改，也不存在修改字段内容的方法。无需对Immutable角色应用Single Threaded Execution模式。无需使用synchronized关键字。就是本案例的User类。

外部可以获取的字段都设置为不可变的




**何时使用Immutable模式**

1. 创建实例后，状态不再发生改变

必须是实例创建后，状态不再发生变化的。实例的状态由字段的值决定。即使字段是final的且不存在setter，也有可能不是不可变的。因为字段引用的实例有可能发生变化。

2. 实例是共享的，且被频繁访问时

Immutable模式的优点是不需要使用synchronized关键字进行保护。意味着在不失去安全性和生存性的前提下提高性能。当实例被多个线程共享，且有可能被频繁访问时，Immutable模式优点明显。

**注意：**

StringBuffer类表示字符串的可变类，String类表示字符串的不可变类。String实例表示的字符串不可以修改，执行操作的方法都不是synchronized修饰的，引用速度更快。

如果需要频繁修改字符串内容，则使用StringBuffer；如果不需要修改字符串内容，只是引用内容，则使用String。

**JDK中的不可变模式**

- java.lang.String
- java.math.BigInteger
- java.math.Decimal
- java.util.regex.Pattern
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Double
- java.lang.Float
- java.lang.Integer
- java.lang.Long
- java.lang.Short
- java.lang.Void

# 26 Guarded Suspension模式

Guarded表示被守护、被保卫、被保护。Suspension表示暂停。如果执行现在的处理会造成问题，就让执行处理的线程进行等待——这就是Guarded Suspension模式。

Guarded Suspension模式通过让线程等待来保证实例的安全型。

Guarded Suspension也称为guarded wait、spin lock等名称。

**示例程序：**

```
package com.lagou.concurrent.demo;

public class Request {

    private final String name;

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

    @Override
    public String toString() {
        return "Request{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

```
package com.lagou.concurrent.demo;

import java.util.LinkedList;
import java.util.Queue;

public class RequestQueue {

    private final Queue<Request> queue = new LinkedList<>();

    public synchronized Request getRequest() {
        while (queue.peek() == null) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return queue.remove();
    }

    public synchronized void putRequest(Request request) {
        queue.offer(request);
        notifyAll();
    }
```

```
}
```

```
package com.lagou.concurrent.demo;

import java.util.Random;

public class ServerThread extends Thread {
    private final Random random;
    private final RequestQueue requestQueue;

    public ServerThread(RequestQueue requestQueue, String name, long seed) {
        super(name);
        this.requestQueue = requestQueue;
        random = new Random(seed);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Request request = requestQueue.getRequest();
            System.out.println(Thread.currentThread().getName() + " 处理 " +
request);
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

import java.util.Random;

public class ClientThread extends Thread {
    private final Random random;
    private final RequestQueue requestQueue;

    public ClientThread(RequestQueue requestQueue, String name, long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Request request = new Request("请求: " + i);
            System.out.println(Thread.currentThread().getName() + " 请求 " +
request);
            requestQueue.putRequest(request);
```

```
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        RequestQueue requestQueue = new RequestQueue();
        new ClientThread(requestQueue, "client-1", 432432L).start();
        new ServerThread(requestQueue, "server-1", 9988766L).start();
    }
}
```

**执行效果：**


**应用保护条件进行保护：**

```java
public class RequestQueue {

    private final Queue<Request> queue = new LinkedList<>();

    public synchronized Request getRequest() {
        while (queue.peek() == null) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return queue.remove();
    }
```

上图中，getRequest方法执行的逻辑是从queue中取出一个Request实例，即`queue.remove()`，但是要获取Request实例，必须满足条件：`queue.peek() != null`。该条件就是Guarded Suspension模式的守护条件（guard condition）。

当线程执行到while语句时：

- 若守护条件成立，线程不进入while语句块，直接执行queue.remove()方法，线程不会等待。
- 若守护条件不成立，线程进入while语句块，执行wait，开始等待。

```java
    public synchronized void putRequest(Request request) {
        queue.offer(request);
        notifyAll();
    }
```

若守护条件不成立，则线程等待。等待什么？等待notifyAll()唤醒该线程。

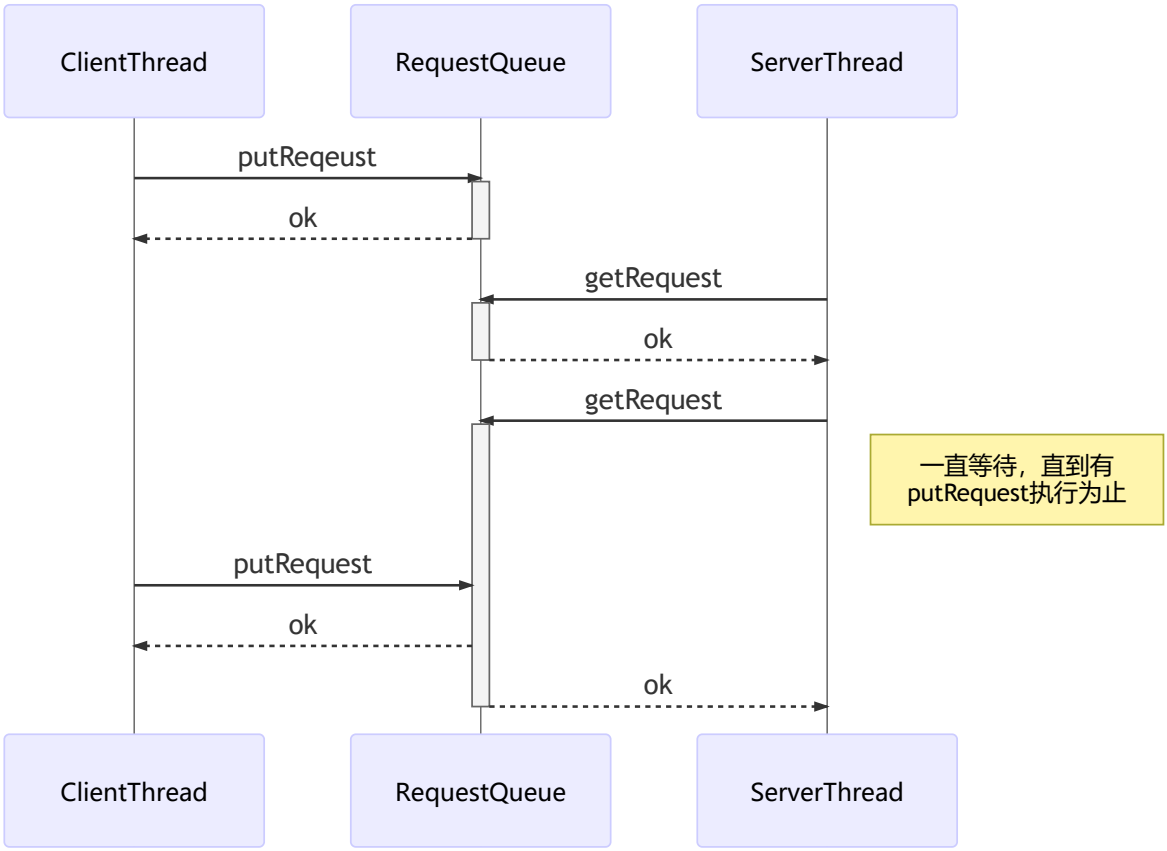
守护条件阻止了线程继续向前执行，除非实例状态发生改变，守护条件成立，被另一个线程唤醒。

该类中的synchronized关键字保护的是queue字段，getRequest方法的synchronized保护该方法只能由一个线程执行。

线程执行this.wait之后，进入this的等待队列，并释放持有的this锁。

notify、notifyAll或interrupt会让线程退出等待队列，实际继续执行之前还必须再次获取this的锁线程才可以继续执行。

**时序图**




**Guarded Suspension模式中的角色**

- GuardedObject（被保护的对象）

GuardedObject角色是一个持有被保护（guardedMethod）的方法的类。当线程执行guardedMethod方法时，若守护条件成立，立即执行；当守护条件不成立，等待。守护条件随着GuardedObject角色的状态不同而变。

除了guardedMethod之外，GuardedObject角色也可以持有其他改变实例状态（stateChangingMethod）的方法。

java中，guardedMethod通过while语句和wait方法来实现，stateChangingMethod通过notify/notifyAll方法实现。

在本案例中，RequestQueue为GuardedObject，getRequest方法为guardedMethod，putRequest为stateChangingMethod。

可以将Guarded Suspension理解为多线程版本的if。

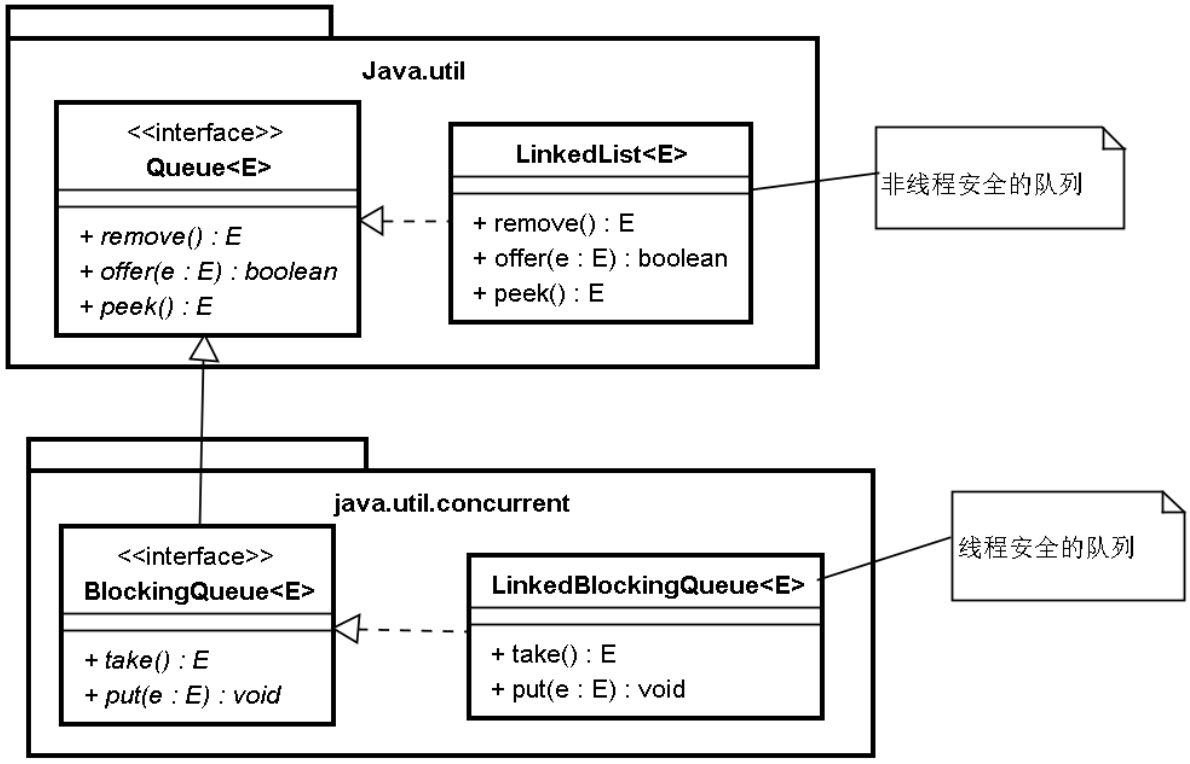
**LinkedBlockingQueue**

可以使用LinkedBlockingQueue替代RequestQueue。

```
1 package com.lagou.concurrent.demo;
2
```

```java
import java.util.concurrent.LinkedBlockingQueue;

public class LinkedBlockingQueueRequestQueue {

    //    private final Queue<Request> queue = new LinkedList<>();
    private final LinkedBlockingQueue<Request> queue = new
LinkedBlockingQueue<>();

    public Request getRequest() {
        Request request = null;
        try {
            request = queue.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return request;
    }

    public void putRequest(Request request) {
        try {
            queue.put(request);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

**类图**



# 27 Balking模式

所谓Balk，就是停止并返回的意思。

Balking模式与Guarded Suspension模式一样，也存在守护条件。在Balking模式中，如果守护条件不成立，则立即中断处理。而Guarded Suspension模式一直等待直到可以运行。

**示例程序**

两个线程，一个是修改线程，修改之后，等待随机时长，保存文件内容。

另一个是保存线程，固定时长进行文件内容的保存。

如果文件需要保存，则执行保存动作

如果文件不需要保存，则不执行保存动作。

```
package com.lagou.concurrent.demo;

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class Data {
    private final String filename;
    private String content;
    private boolean changed;

    public Data(String filename, String content) {
        this.filename = filename;
        this.content = content;
    }

    public synchronized void change(String newContent) {
        this.content = newContent;
        this.changed = true;
    }

    public synchronized void save() throws IOException {
        if (changed) {
            doSave();
            changed = false;
        } else {
            System.out.println(Thread.currentThread().getName() + "不需要保存");
        }
    }

    private void doSave() throws IOException {
        System.out.println(Thread.currentThread().getName() + " 调用doSave，内容为：" + content);
        Writer writer = new FileWriter(filename);
        writer.write(content);
        writer.close();
    }
```

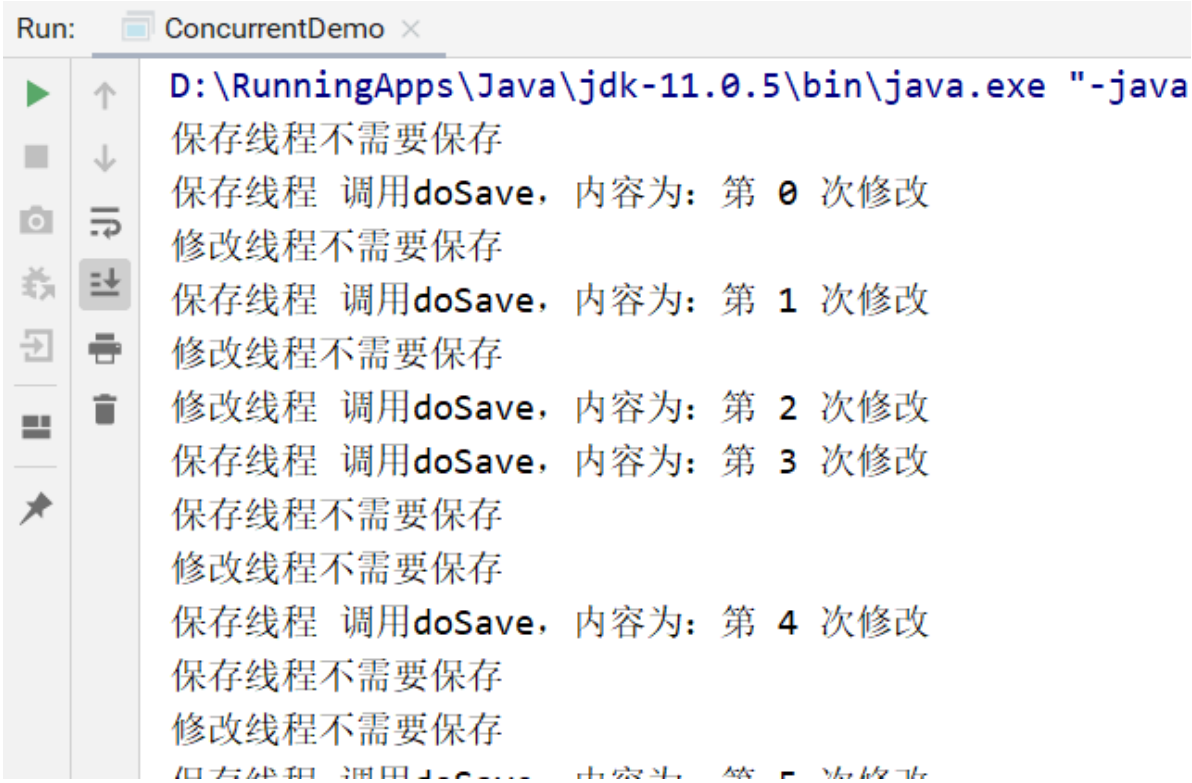
```
}
```

```
package com.lagou.concurrent.demo;

import java.io.IOException;
import java.util.Random;

public class ChangerThread extends Thread {
    private final Data data;
    private final Random random = new Random();

    public ChangerThread(String name, Data data) {
        super(name);
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; true; i++) {
            data.change("第 " + i + " 次修改");
            try {
                Thread.sleep(random.nextInt(2000));
                data.save();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.lagou.concurrent.demo;

import java.io.IOException;

public class SaverThread extends Thread {
    private final Data data;

    public SaverThread(String name, Data data) {
        super(name);
        this.data = data;
    }

    @Override
    public void run() {
        while (true) {
            try {
                data.save();
                Thread.sleep(1000);
            } catch (IOException e) {
```

```
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

**执行效果：**




**Balking模式中的角色**

- GuardedObject（受保护对象）

GuardedObject角色是一个拥有被保护的方法（guardedMethod）的类。当线程执行guardedMethod时，若保护条件成立，则执行实际的处理，若不成立，则不执行实际的处理，直接返回。

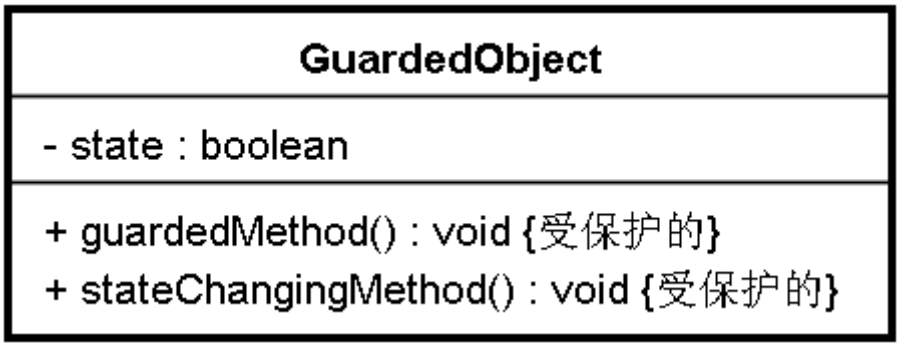
保护条件的成立与否随着GuardedObject角色状态的改变而变动。

除了guardedMethod之外，GuardedObject角色还有可能有其他改变状态的方法（stateChangingMethod）。

在此案例中，Data类对应于GuardedObject，save方法对应guardedMethod，change方法对应stateChangingMethod方法。

保护条件是changed字段为true。

**类图**




**何时使用Balking模式**

- 不需要执行时

在此示例程序中，content字段的内容如果没有修改，就将save方法balk。之所以要balk，是因为content已经写文件了，无需再写了。如果并不需要执行，就可以使用Balking模式。此时可以提高程序性能。

- 不需要等待守护条件成立时

Balking模式的特点就是不等待。若条件成立，就执行，若不成立，就不执行，立即进入下一个操作。

- 守护条件仅在第一次成立时

当“守护条件仅在第一次成立”时，可以使用Balking模式。

比如各种类的初始化操作，检查一次是否初始化了，如果初始化了，就不用执行了。如果没有初始化，则进行初始化。

**balk结果的表示**

1. 忽略balk

最简单的方式就是不通知调用端“发生了balk”。示例程序采用的就是这种方式。

2. 通过返回值表示balk

通过boolean值表示balk。若返回true，表示未发生balk，需要执行并执行了处理。若false，则表示发生了balk，处理已执行，不再需要执行。

有时也会使用null来表示“发生了balk”。

3. 通过异常表示balk

有时也通过异常表示“发生了balk”。即，当balk时，程序并不从方法return，而是抛异常。

**Balking和Guarded Suspension模式之间**

介于“直接balk并返回”和“等待到守护条件成立为止“这两种极端之间的还有一种”在守护条件成立之前等待一段时间“。在守护条件成立之前等待一段时间，如果到时条件还未成立，则直接balk。

这种操作称为计时守护（guarded timed）或超时（timeout）。

**java.util.concurrent中的超时**

1. 通过异常通知超时

当发生超时抛出异常时，不适合使用返回值表示超时，需要使用java.util.concurrent.TimeoutException异常。

如：

java.util.concurrent.Future的get方法；

java.util.concurrent.Exchanger的exchange方法；

java.util.concurrent.Cyclicarrier的await方法

java.util.concurrent.CountDownLatch的await方法。

2. 通过返回值通知超时

当执行多次try时，则不使用异常，而使用返回值表示超时。

如：

java.util.concurrent.BlockingQueue接口，当offer方法的返回值为false，或poll方法的返回值为null，表示发生了超时。

java.util.concurrent.Semaphore类，当tryAcquire方法的返回值为false时，表示发生了超时。

java.util.concurrent.locks.Lock接口，当tryLock方法的返回值为false时，表示发生了超时。

## 28 Producer-Consumer模式

生产者安全地将数据交给消费者。

当生产者和消费者以不同的线程运行时，两者之间的处理速度差异会有问题。

生产者消费者模式用于消除线程间处理速度的差异带来的问题。

在该模式中，生产者和消费者都有多个，当生产者和消费者只有一个时，我们称为管道（Pipe）模式。

**示例程序**

```
package com.lagou.concurrent.demo;

public class Table {
    private final String[] buffer;
    private int tail;
    private int head;
    private int count;

    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
```

```
15
16      public synchronized void put(String steamedBread) throws
    InterruptedException {
17          System.out.println(Thread.currentThread().getName() + " 蒸出来 " +
    steamedBread);
18          while (count >= buffer.length) {
19              wait();
20          }
21          buffer[tail] = steamedBread;
22          tail = (tail + 1) % buffer.length;
23          count++;
24          notifyAll();
25      }
26
27      public synchronized String take() throws InterruptedException {
28          while (count <= 0) {
29              wait();
30          }
31          String steamedBreak = buffer[head];
32          head = (head + 1) % buffer.length;
33          count--;
34          notifyAll();
35          System.out.println(Thread.currentThread().getName() + " 取走 " +
    steamedBreak);
36          return steamedBreak;
37      }
38
39  }
```

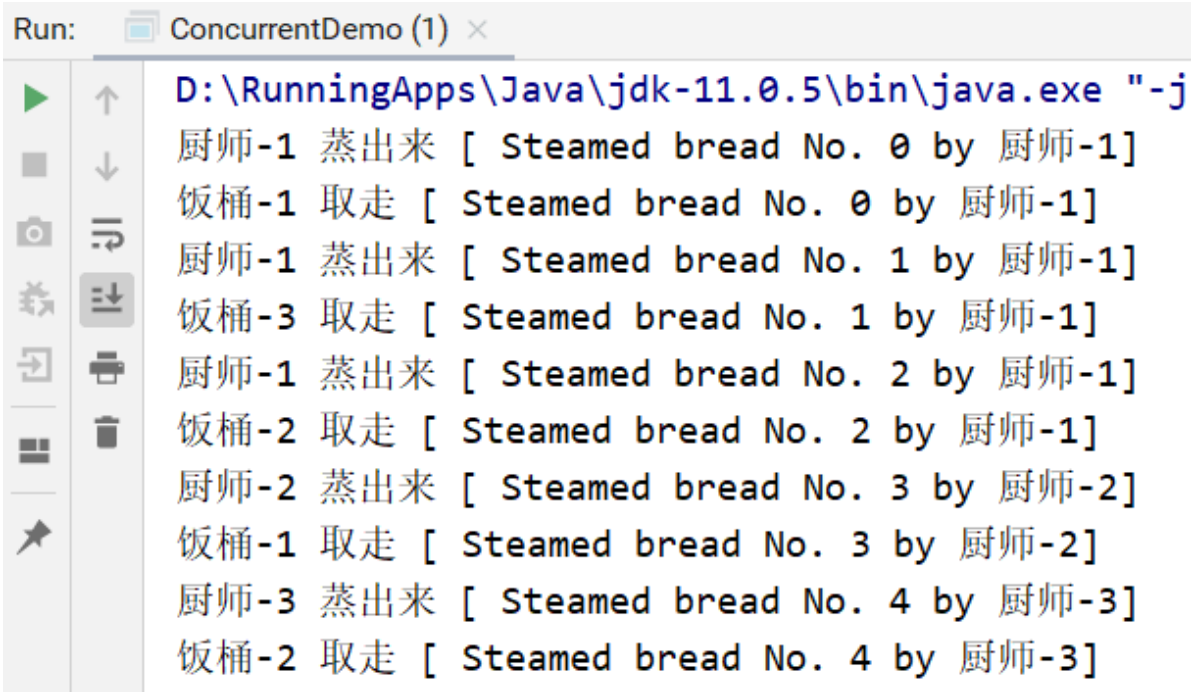
```
1   package com.lagou.concurrent.demo;
2
3   import java.util.Random;
4
5   public class CookerThread extends Thread {
6       private final Random random;
7       private final Table table;
8       private static int id = 0;
9
10      public CookerThread(String name, Table table, long seed) {
11          super(name);
12          this.table = table;
13          this.random = new Random(seed);
14      }
15
16      @Override
17      public void run() {
18          while (true) {
19              try {
20                  Thread.sleep(random.nextInt(1000));
21                  String steamedBread = "[ Steamed bread No. " + nextId() + "
    by " + getName() + "]";
22                  table.put(steamedBread);
23              } catch (InterruptedException e) {
24                  e.printStackTrace();
25              }
```

```
        }

    }

    private static synchronized int nextId() {
        return id++;
    }

}
```

```
package com.lagou.concurrent.demo;

import java.util.Random;

public class EaterThread extends Thread {

    private final Random random;
    private final Table table;

    public EaterThread(String name, Table table, long seed) {
        super(name);
        this.table = table;
        this.random = new Random(seed);
    }

    @Override
    public void run() {
        try {
            while (true) {
                String steamedBread = table.take();
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        Table table = new Table(3);
        new CookerThread("厨师-1", table, 12345L).start();
        new CookerThread("厨师-2", table, 23456L).start();
        new CookerThread("厨师-3", table, 34567L).start();

        new EaterThread("饭桶-1", table, 45678L).start();
        new EaterThread("饭桶-2", table, 56789L).start();
        new EaterThread("饭桶-3", table, 67890L).start();
    }
}
```

**执行效果：**




**关于put方法**

put方法会抛出InterruptedException异常。如果抛出，可以理解为"该操作已取消"。

put方法使用了Guarded Suspension模式。

tail和count的更新采取buffer环的形式。

notifyAll方法唤醒正在等待馒头的线程来吃。

**关于take方法**

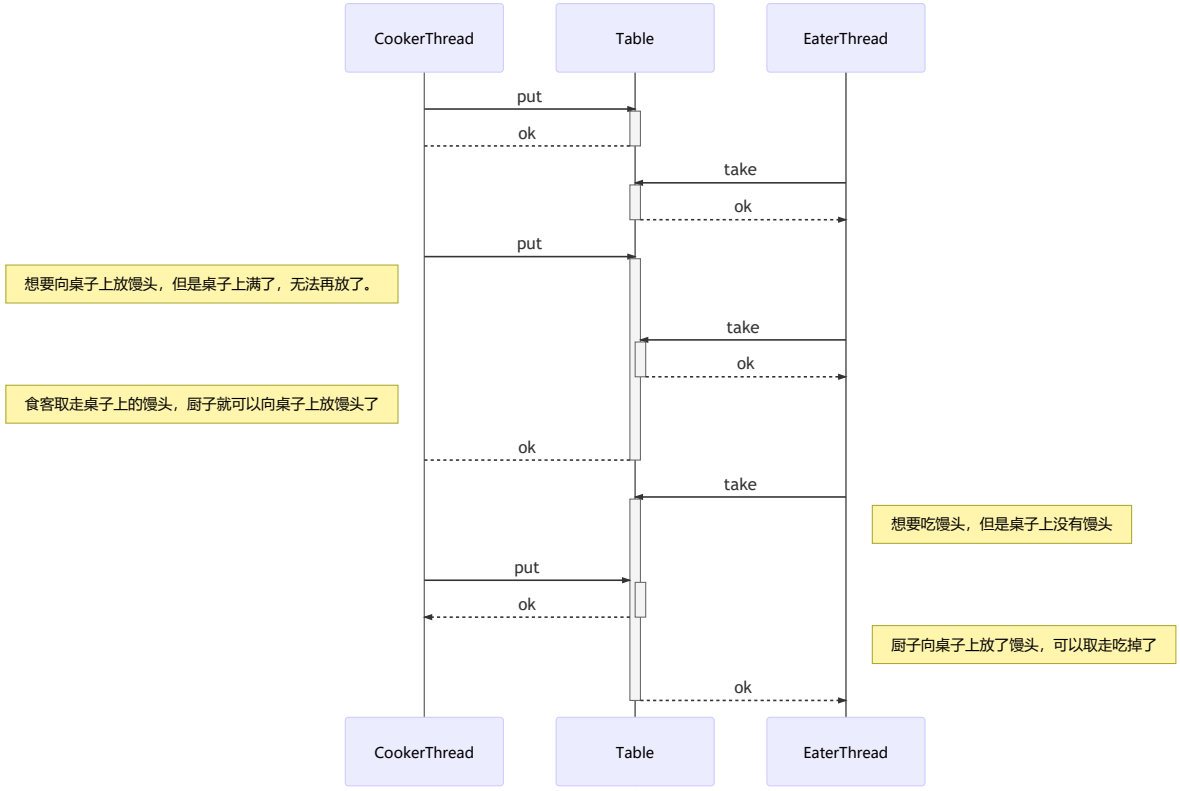
take方法会抛出InterruptedException异常，表示该操作已取消。

take方法采用了Guarded Suspension模式。

head和count的更新采用了buffer环的形式。

notifyAll唤醒等待的厨子线程开始蒸馒头。

**时序图：**




**Producer-Consumer模式中的角色**

1. Data

Data角色由Producer角色生成，供Consumer角色使用。在本案例中，String类的馒头对应于Data角色。

2. Producer

Producer角色生成Data角色，并将其传递给Channel角色。本案例中，CookerThread对应于Producer角色。

3. Consumer

Consumer角色从Channel角色获取Data角色并使用。本案例中，EaterThread对应于Consumer角色。

4. Channel角色

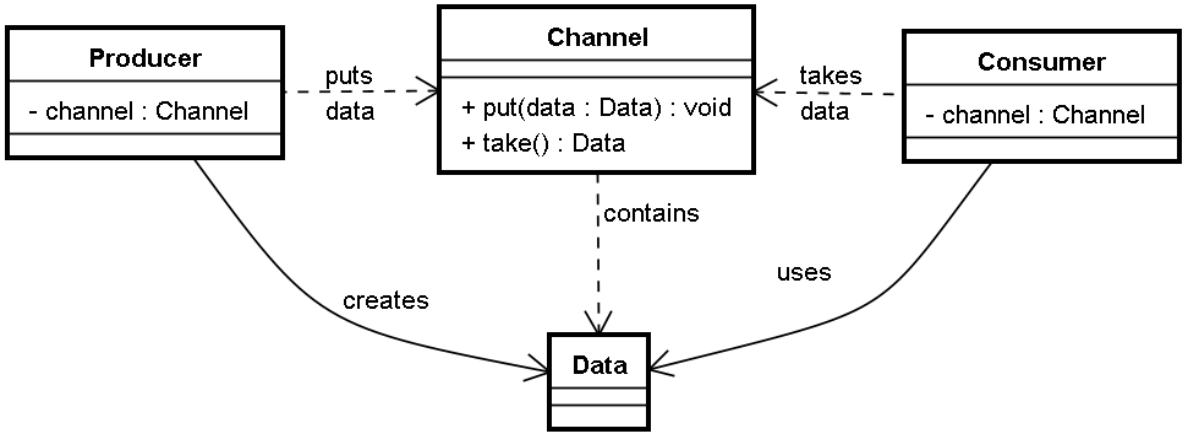
Channel角色管理从Producer角色获取的Data角色，还负责响应Consumer角色的请求，传递Data角色。为了安全，Channel角色会对Producer角色和Consumer角色进行互斥处理。

当producer角色将Data角色传递给Channel角色时，如果Channel角色状态不能接收Data角色，则Producer角色将一直等待，直到Channel可以接收Data角色为止。

当Consumer角色从Channel角色获取Data角色时，如果Channel角色状态没有可以传递的Data角色，则Consumer角色将一直等待，直到Channel角色状态转变为可以传递Data角色为止。

当存在多个Producer角色和Consumer角色时，Channel角色需要对它们做互斥处理。

**类图：**




**守护安全性的Channel角色（可复用）**

在生产者消费者模型中，承担安全守护责任的是Channel角色。Channel角色执行线程间的互斥处理，确保Producer角色正确地将Data角色传递给Consumer角色。

**不要直接传递**

Consumer角色想要获取Data角色，通常是因为想使用这些Data角色来执行某些处理。如果Producer角色直接调用Consumer的方法，执行处理的就不是Consumer的线程，而是Producer角色的线程了。这样一来，异步处理变同步处理，会发生不同Data间的延迟，降低程序的性能。

**传递Data角色的顺序**

1. 队列——先生产先消费
2. 栈——先生产后消费
3. 优先队列——”优先“的先消费

**Channel意义**

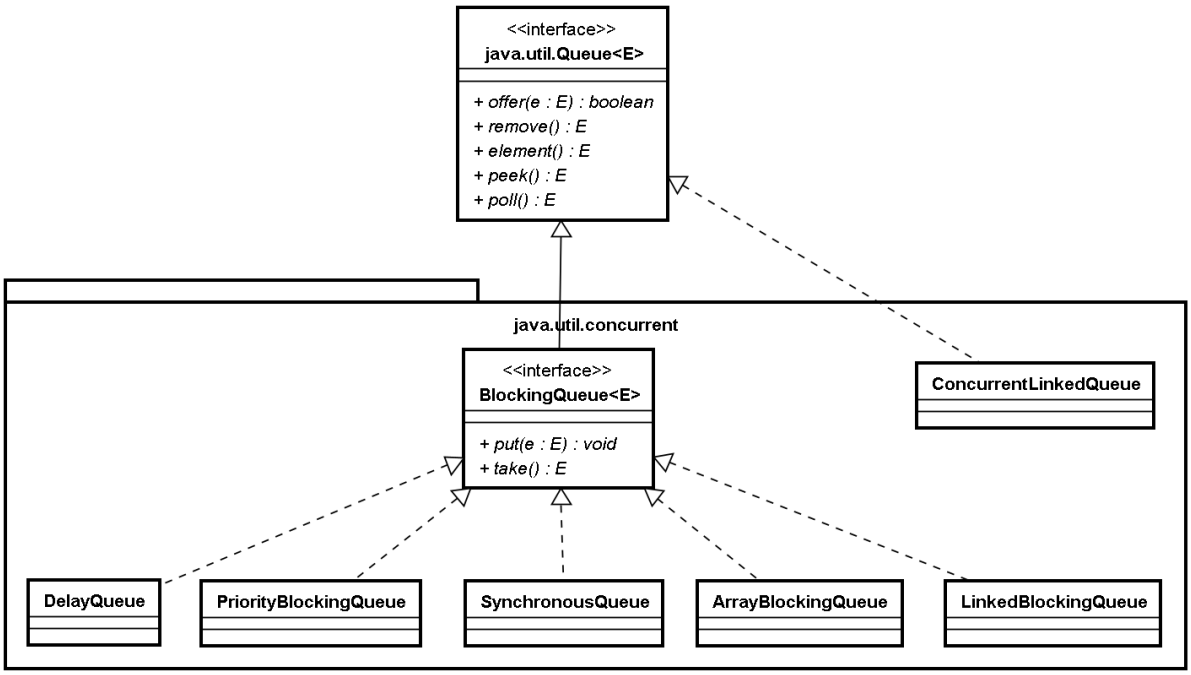
线程的协调要考虑”放在中间的东西“

线程的互斥要考虑”应该保护的东西“

为了让线程协调运行，必须执行互斥处理，以防止共享的内容被破坏。线程的互斥处理时为了线程的协调运行而执行的。

**JUC包和Producer-Consumer模式**

JUC中提供了BlockingQueue接口及其实现类，相当于Producer-Consumer模式中的Channel角色。




- BlockingQueue接口——阻塞队列
- ArrayBlockingQueue——基于数组的BlockingQueue
- LinkedBlockingQueue——基于链表的BlockingQueue
- PriorityBlockingQueue——带有优先级的BlockingQueue
- DelayQueue——一定时间之后才可以take的BlockingQueue
- SynchronousQueue——直接传递的BlockingQueue
- ConcurrentLinkedQueue——元素个数没有最大限制的线程安全队列

# 29 Read-Write Lock模式

当线程读取实例的状态时，实例的状态不会发生变化。实例的状态仅在线程执行写入操作时才会发生变化。

从实例状态变化来看，读取和写入有本质的区别。

在本模式中，读取操作和写入操作分开考虑。在执行读取操作之前，线程必须获取用于读取的锁。在执行写入操作之前，线程必须获取用于写入的锁。

可以多个线程同时读取，读取时不可写入。

当线程正在写入时，其他线程不可以读取或写入。

一般来说，执行互斥会降低程序性能。如果把写入的互斥和读取的互斥分开考虑，则可以提高性能。

## 示例程序

### 入口程序

```java
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        Data data = new Data(10);
        // 启动读取线程，读取同一个对象中的数据
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        // 启动写线程，两个线程一个写大写，另一个写小写
        new WriterThread(data, "ABCDEFGHIJKLMNOPQRSTUVWXYZ").start();
        new WriterThread(data, "abcdefghijklmnopqrstuvwxyz").start();
    }
}
```

### 数据对象

```java
package com.lagou.concurrent.demo;

public class Data {
    private final char[] buffer;
    // Data在线程之间共享
    // Data保有读写锁
    private final ReadWriteLock lock = new ReadWriteLock();

    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }

    public char[] read() throws InterruptedException {
        lock.readLock();
        try {
            return doRead();
        } finally {
            lock.readUnlock();
        }
    }

    public void write(char c) throws InterruptedException {
```

```
        // 加锁，当前线程执行写操作。
        lock.writeLock();
        try {
            // 写操作
            doWrite(c);
        } finally {
            // 写入完成，释放锁，同时唤醒其他等待的线程
            lock.writeUnlock();
        }
    }

    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {
            newbuf[i] = buffer[i];
        }
        slowly();
        return newbuf;
    }

    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = c;
            slowly();
        }
    }

    private void slowly() {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

**读写锁**

```
package com.lagou.concurrent.demo;

public class ReadWriteLock {
    private int readingReaders = 0;
    private int waitingWriters = 0;
    private int writingWriters = 0;
    private boolean preferWriter = true;

    /**
     * 对于读操作：
     * 1. 如果有正在写入的writer，则当前线程等待writer写入完成
     * 2. 如果没有正在写入的writer，但是有正在等待写入的writer，同时偏向写操作
     *    则当前线程等待writer写入完成
     * @throws InterruptedException
```

```
     */
    public synchronized void readLock() throws InterruptedException {
        while (writingWriters > 0 || (preferWriter && waitingWriters > 0)) {
            // 当前线程等待
            wait();
        }
        readingReaders++;
    }

    public synchronized void readUnlock() {
        readingReaders--;
        preferWriter = true;
        // 唤醒在当前对象上等待的其他线程
        notifyAll();
    }

    public synchronized void writeLock() throws InterruptedException {
        waitingWriters++;
        try {
            // 如果有正在读取的线程，或者有正在写入的线程，则当前线程等待被唤醒
            while (readingReaders > 0 || writingWriters > 0) {
                wait();
            }
        } finally {
            waitingWriters--;
        }
        writingWriters++;
    }

    public synchronized void writeUnlock() {
        writingWriters--;
        preferWriter = false;
        // 当前线程写入完成，唤醒其他等待的线程进行读写操作
        notifyAll();
    }

}
```

**写线程**

```
package com.lagou.concurrent.demo;

import java.util.Random;

public class WriterThread extends Thread {
    private static final Random RANDOM = new Random();
    private final Data data;
    private final String filler;
    private int index = 0;

    public WriterThread(Data data, String filler) {
        this.data = data;
        this.filler = filler;
```

```
    }

    @Override
    public void run() {
        try {
            while (true) {
                // 使用轮询的方法，读取当前一个字节
                char c = nextchar();
                // 写字节
                data.write(c);
                Thread.sleep(RANDOM.nextInt(3000));
            }
        } catch (InterruptedException e) {
        }
    }

    private char nextchar() {
        char c = filler.charAt(index);
        index++;
        if (index >= filler.length()) {
            index = 0;
        }
        return c;
    }
}
```

**读取线程**

```
package com.lagou.concurrent.demo;

public class ReaderThread extends Thread {
    private final Data data;

    public ReaderThread(Data data) {
        this.data = data;
    }

    @Override
    public void run() {
        try {
            while (true) {
                char[] readbuf = data.read();
                System.out.println(Thread.currentThread().getName() + " 读取了 " + String.valueOf(readbuf));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

## 守护条件

readLock方法和writeLock方法都是用了Guarded Suspension模式。Guarded Suspension模式的重点是守护条件。

**readLock方法：**

读取线程首先调用readLock方法。当线程从该方法返回，就可以执行实际的读取操作。

当线程开始执行实际的读取操作时，只需要判断是否存在正在写入的线程，以及是否存在正在等待的写入线程。

不考虑读取线程。

如果存在正在写入的线程或者存在正在等待的写线程，则等待。

**writeLock方法：**

在线程开始写入之前，调用writeLock方法。当线程从该方法返回后，就可以执行实际的写入操作。

开始执行写入的条件：如果有线程正在执行读取操作，出现读写冲突；或者如果有线程正在执行写入的操作，引起写冲突，当前线程等待。

## Read-Write Lock模式中的角色：

**Reader**

该角色对共享资源角色执行读取操作。

**Writer**

该角色对共享资源角色执行写操作。

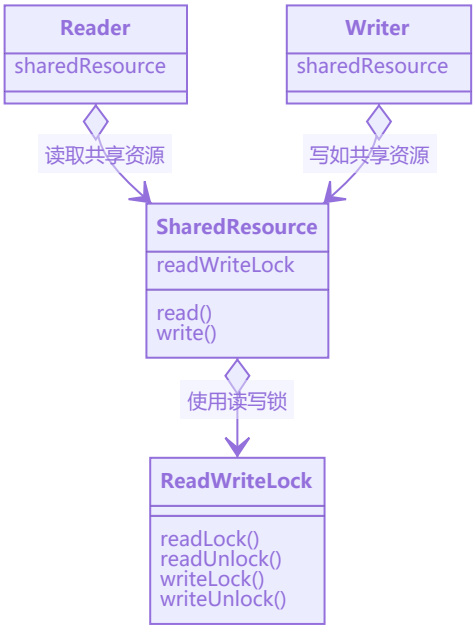
**SharedResource**

共享资源角色表示Reader角色和Writer角色共享的资源。共享资源角色提供不修改内部状态的操作（读取）和修改内部状态的操作（写）。

当前案例中对应于Data类。

**ReadWriteLock**

读写锁角色提供了共享资源角色实现读操作和写操作时需要的锁，即当前案例中的readLock和readUnlock，以及writeLock和writeUnlock。对应于当前案例中ReadWriteLock类。




## 要点

1. 利用读取操作的线程之间不会冲突的特性来提高程序性能

   Read-Write Lock模式利用了读操作的线程之间不会冲突的特性。由于读取操作不会修改共享资源的状态，所以彼此之间无需加锁。因此，多个Reader角色同时执行读取操作，从而提高程序性能。

2. 适合读取操作负载较大的情况

   如果单纯使用Single Threaded Execution模式，则read也只能运行一个线程。如果read负载很重，可以使用Read-Write Lock模式。

3. 适合少写多读

   Read-Write Lock模式优点是Reader之间不会冲突。如果写入很频繁，Writer会频繁停止Reader的处理，也就无法体现出Read-Write Lock模式的优势了。

## 锁的含义

synchronized可以用于获取实例的锁。java中同一个对象锁不能由两个以上的线程同时获取。

用于读取的锁和用于写入的锁与使用synchronized获取的锁是不一样的。开发人员可以通过修改ReadWriteLock类来改变锁的运行。

ReadWriteLock类提供了用于读取的锁和用于写入的锁两个逻辑锁，但是实现这两个逻辑锁的物理锁只有一个，就是ReadWriteLock实例持有的锁。

## JUC包和Read-Write Lock模式

java.util.concurrent.locks包提供了已实现Read-Write Lock模式的ReadWriteLock接口和ReentrantReadWriteLock类。

java.util.concurrent.locks.ReadWriteLock接口的功能和当前案例中的ReadWriteLock类类似。不同之处在于该接口用于读取的锁和用于写入的锁是通过其他对象来实现的。

| 当前案例的ReadWriteLock | java.util.concurrent.locks中的ReadWriteLock接口 |
| --- | --- |
| readLock() | readLock().lock() |
| readUnlock() | readLock().unlock() |
| writeLock() | writeLock().lock() |
| writeUnlock() | writeLock().unlock() |

java.util.concurrent.locks.ReentrantReadWriteLock类实现了ReadWriteLock接口。其特征如下：

- 公平性

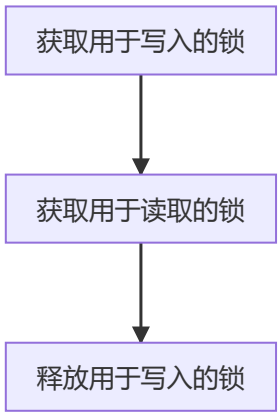
  当创建ReentrantReadWriteLock类的实例时，可以选择锁的获取顺序是否要设置为fair的。如果创建的实例是公平的，那么等待时间久的线程将可以优先获取锁。

- 可重入性

  ReentrantReadWriteLock类的锁是可重入的。Reader角色的线程可以获取用于写入的锁，Writer角色的线程可以获取用于读取的锁。

- 锁降级

  ReentrantReadWriteLock类可以按如下顺序将用于写入的锁降级为用于读取的锁：

  用于读取的锁不能升级为用于写入的锁。

- 快捷方法

  ReentrantReadWriteLock类提供了获取等待中的线程个数的方法`getQueueLength`，以及检查是否获取了用于写入锁的方法`isWriteLocked`等方法。

# 30 Thread-Per-Message模式

该模式可以理解为“每个消息一个线程”。消息这里可以理解为命令或请求。每个命令或请求分配一个线程，由这个线程来处理。

这就是Thread-Per-Message模式。

在Thread-Per-Message模式中，消息的委托方和执行方是不同的线程。

## 示例程序

在此示例程序中，ConcurrentDemo类委托Host来显示字符。Host类会创建一个线程，来处理委托。启动的线程使用Helper类来执行实际的显示。

### 主入口类

```
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        System.out.println("主线程 -- 开始执行");
        Host host = new Host();
        host.request(10, 'A');
        host.request(20, 'B');
        host.request(30, 'C');
        System.out.println("主线程 -- 执行结束");
    }
}
```

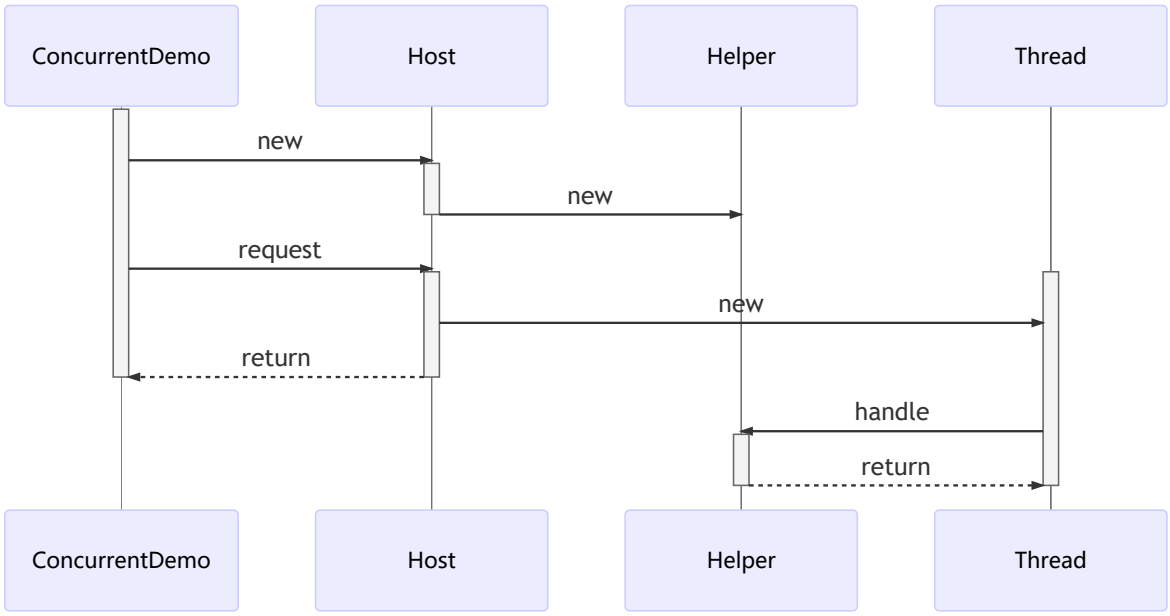
### 处理器类

```
package com.lagou.concurrent.demo;

public class Host {

    private final Helper helper = new Helper();

    public void request(final int count, final char c) {
        System.out.println("\t请求：【" + count + ", " + c + "】开始。。。");
        new Thread() {
            @Override
            public void run() {
                helper.handle(count, c);
            }
        }.start();
        System.out.println("\t请求：【" + count + ", " + c + "】结束！！！");
    }

```

```
}
```

**工具类**

```
package com.lagou.concurrent.demo;

public class Helper {

    public void handle(int count, char c) {
        System.out.println("\t\t处理：【" + count + "，" + c + "】开始。。。");
        for (int i = 0; i < count; i++) {
            slowly();
            System.out.print(c);
        }
        System.out.println("");
        System.out.println("\t\t处理：【" + count + "，" + c + "】结束！！！");

    }

    private void slowly() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```




## Thread-Per-Message模式中的角色

**Client（委托方）**

Client角色向Host角色发起请求，而不用关心Host角色如何实现该请求处理。

当前案例中对应于ConcurrentDemo类。

**Host**

Host角色收到Client角色请求后，创建并启用一个线程。新建的线程使用Helper角色来处理请求。

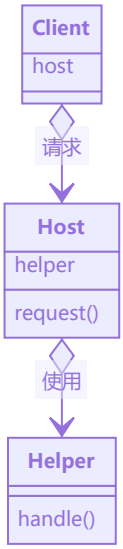
当前案例中对应于Host类。

**Helper**

Helper角色为Host角色提供请求处理的功能。Host角色创建的新线程调用Helper角色。

当前案例中对应于Helper类。

## 类图1




## 要点

1. 提高响应性，缩短延迟时间

   Thread-Per-Message模式能够提高与Client角色对应的Host角色的响应性，降低延迟时间。尤其是当handle操作非常耗时或者handle操作需要等待输入/输出时，效果很明显。

   为了缩短线程启动花费的时间，可以使用Worker Thread模式。

2. 适用于操作顺序没有要求时

   在Thread-Per-Message模式中，handle方法并不一定按照request方法的调用顺序来执行。

3. 适用于不需要返回值时

   在Thread-Per-Message模式中，request方法并不会等待handle方法的执行结束。request得不到handle的结果。

   当需要获取操作结果时，可以使用Future模式。

4. 应用于服务器

## JUC包和Thread-Per-Message模式

**java.lang.Thread类**

最基本的创建、启动线程的类

**java.lang.Runnable接口**

线程锁执行的任务接口

**java.util.concurrent.ThreadFactory接口**

将线程创建抽象化的接口

**java.util.concurrent.Executors**

用于创建实例的工具类

**java.util.concurrent.Executor接口**

将线程执行抽象化的接口

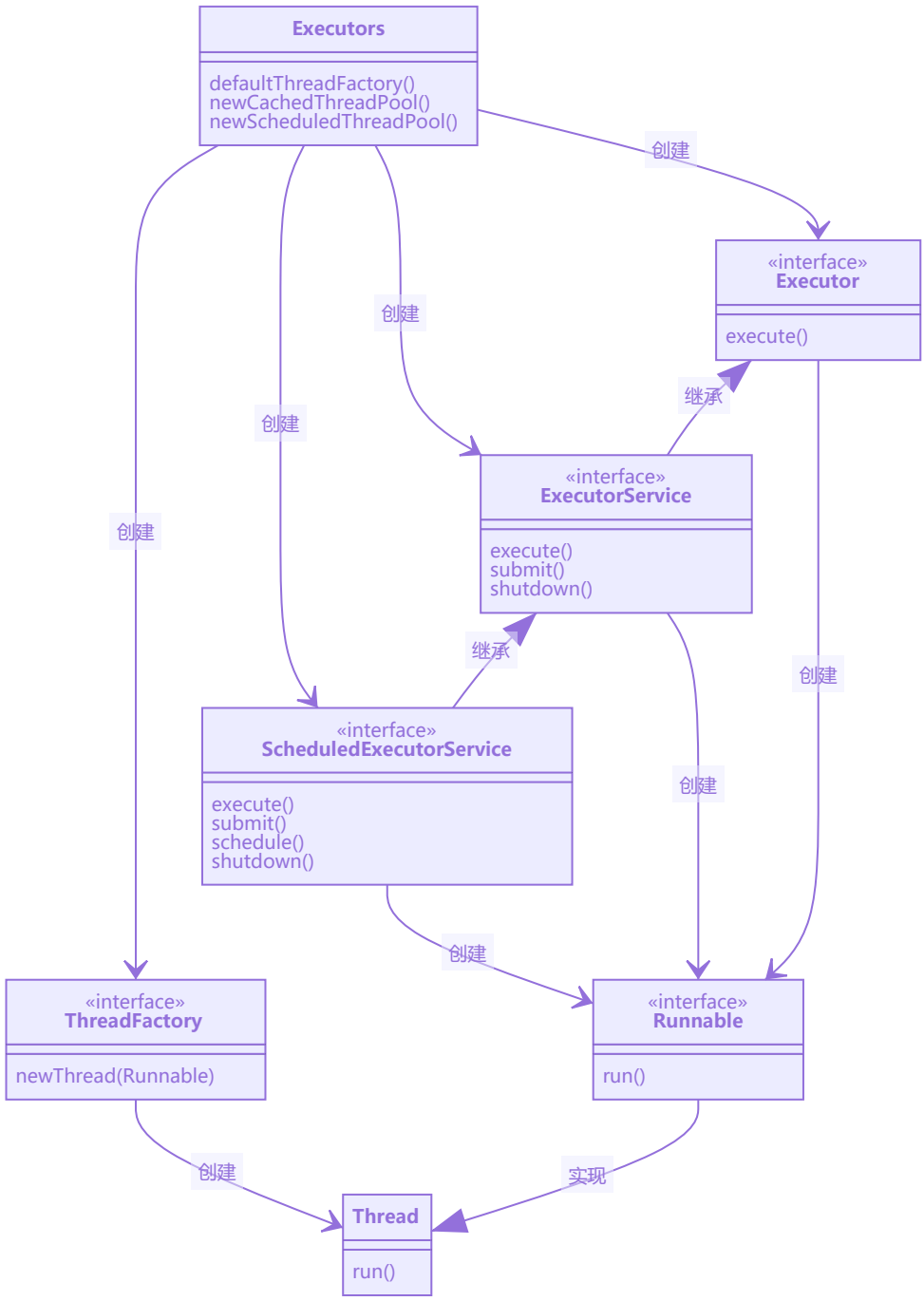
**java.util.concurrent.ExecutorService接口**

将被复用的线程抽象化的接口

**java.util.concurrent.ScheduledExecutorService类**

将被调度线程的执行抽象化的接口

## 类图2



# 31 Worker Thread模式

在Worker Thread模式中，工人线程（worker thread）会逐个取回工作并进行处理。当所有工作全部完成后，工人线程会等待新的工作到来。

Worker Thread模式也被称为Background Thread模式。有时也称为Thread Pool模式。

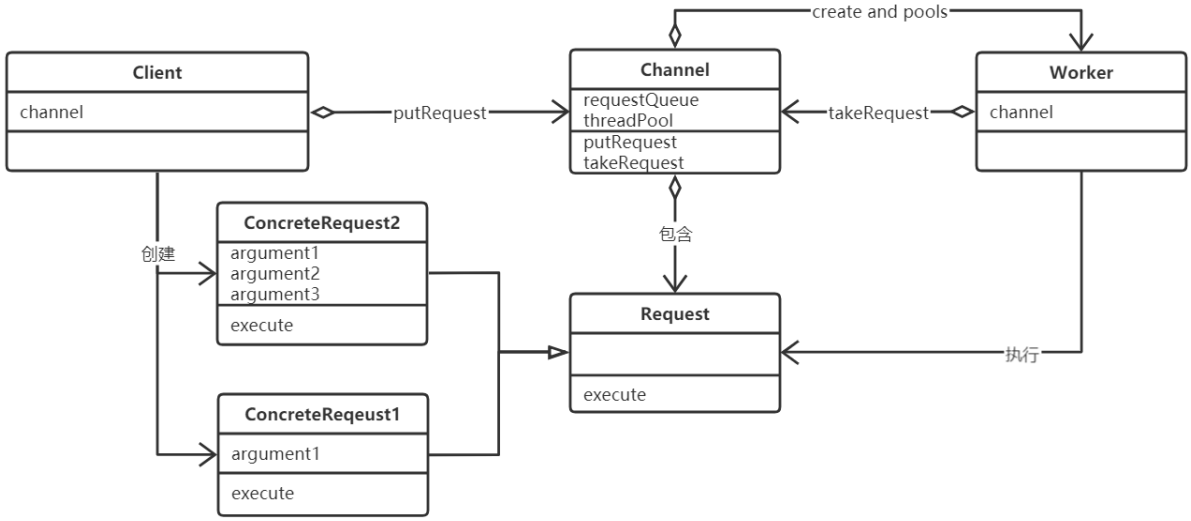
## 示例程序

ClientThread类的线程会向Channel类发送工作请求（委托）。

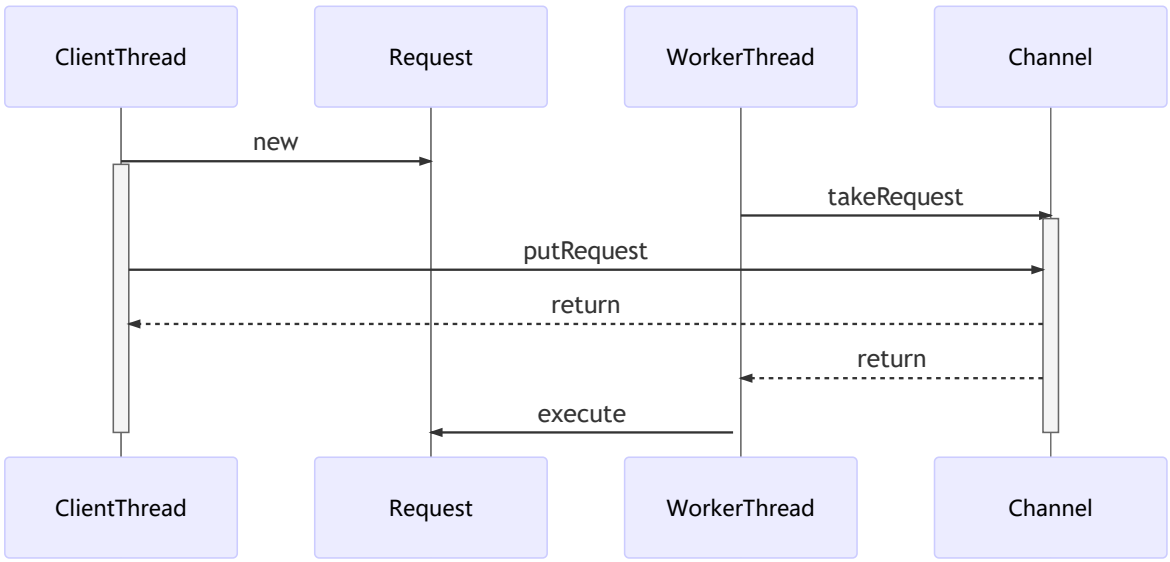
Channel类的实例有五个工人线程进行工作。所有工人线程都在等待工作请求的到来。

当收到工作请求后，工人线程会从Channel获取一项工作请求并开始工作。工作完成后，工人线程回到Channel那里等待下一项工作请求。

### 类图




### 时序图




```java
package com.lagou.concurrent.demo;

public class ConcurrentDemo {
    public static void main(String[] args) {
        Channel channel = new Channel(5);
        channel.startWorkers();
        new ClientThread("张三", channel).start();
        new ClientThread("李四", channel).start();
        new ClientThread("王五", channel).start();
    }
}
```

```java
package com.lagou.concurrent.demo;

public class Channel {

    private static final int MAX_REQUEST = 100;
    private final Request[] requestQueue;
    private int tail;
    private int head;
    private int count;

    private final WorkerThread[] threadPool;

    public Channel(int threads) {
        this.requestQueue = new Request[MAX_REQUEST];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
```

```
18
19          threadPool = new WorkerThread[threads];
20          for (int i = 0; i < threadPool.length; i++) {
21              threadPool[i] = new WorkerThread("Worker-" + i, this);
22          }
23      }
24
25      public void startWorkers() {
26          for (int i = 0; i < threadPool.length; i++) {
27              threadPool[i].start();
28          }
29      }
30
31      public synchronized void putRequest(Request request) {
32          while (count >= requestQueue.length) {
33              try {
34                  wait();
35              } catch (InterruptedException e) {
36
37              }
38          }
39
40          requestQueue[tail] = request;
41          tail = (tail + 1) % requestQueue.length;
42          count++;
43          notifyAll();
44      }
45
46      public synchronized Request takeRequest() {
47          while (count <= 0) {
48              try {
49                  wait();
50              } catch (InterruptedException e) {
51
52              }
53          }
54
55          Request request = requestQueue[head];
56          head = (head + 1) % requestQueue.length;
57          count--;
58          notifyAll();
59          return request;
60      }
61
62  }
```

```
1  package com.lagou.concurrent.demo;
2
3  import java.util.Random;
4
5  public class ClientThread extends Thread {
6      private final Channel channel;
7      private static final Random RANDOM = new Random();
```

```
8
9       public ClientThread(String name, Channel channel) {
10          super(name);
11          this.channel = channel;
12      }
13
14      @Override
15      public void run() {
16          try {
17              for (int i = 0; true; i++) {
18                  Request request = new Request(getName(), i);
19                  channel.putRequest(request);
20                  Thread.sleep(RANDOM.nextInt(1000));
21              }
22          } catch (InterruptedException e) {
23
24          }
25      }
26  }
```

```
1   package com.lagou.concurrent.demo;
2
3   import java.util.Random;
4
5   public class Request {
6       private final String name;
7       private final int number;
8       private static final Random RANDOM = new Random();
9
10      public Request(String name, int number) {
11          this.name = name;
12          this.number = number;
13      }
14
15      public void execute() {
16          System.out.println(Thread.currentThread().getName() + " 执行 " +
    this);
17          try {
18              Thread.sleep(RANDOM.nextInt(1000));
19          } catch (InterruptedException e) {
20
21          }
22      }
23
24      @Override
25      public String toString() {
26          return "Request{" +
27                  "name='" + name + '\'' +
28                  ", number=" + number +
29                  '}';
30      }
31  }
```

```java
package com.lagou.concurrent.demo;

public class WorkerThread extends Thread {

    private final Channel channel;

    public WorkerThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }

    @Override
    public void run() {
        while (true) {
            Request reqeust = channel.takeRequest();
            reqeust.execute();
        }
    }
}
```

## Worker Thread模式中的角色

- Client（委托者）

  Client角色创建Request角色并将其传递给Channel角色。在本例中，ClientThread对应Client角色。

- Channel

  Channel角色接收来自Client角色的Request角色，并将其传递给Worker角色。在本例中，Channel类对应Channel角色。

- Worker

  Worker角色从Channel角色中获取Request角色，并执行其逻辑。当一项工作结束后，继续从Channel获取另外的Request角色。本例中，WorkerThread类对应Worker角色。

- Request

  Request角色表示工作。Request角色中保存了工作的逻辑。本例中，Request类对应Request角色。

## Worker Thread模式的优点

1. 提高吞吐量

   如果将工作交给其他线程，当前线程就可以处理下一项工作，称为Thread Per Message模式。

   由于启动新线程消耗时间，可以通过Worker Thread模式轮流和反复地使用线程来提高吞吐量。

2. 容量控制

   Worker角色的数量在本例中可以传递参数指定。

Worker角色越多，可以并发处理的逻辑越多。同时增加Worker角色会增加消耗的资源。必须根据程序实际运行环境调整Worker角色的数量。

3. 调用与执行的分离

   Worker Thread模式和Thread Per Message模式一样，方法的调用和执行是分开的。方法的调用是invocation，方法的执行是execution。

   这样，可以：

   - 提高响应速度；
   - 控制执行顺序，因为执行不受调用顺序的制约；
   - 可以取消和反复执行；
   - 进行分布式部署，通过网络将Request角色发送到其他Woker计算节点进行处理。

4. Runnable接口的意义

   `java.lang.Runnable`接口有时用作Worker Thread模式的Request角色。即可以创建Runnable接口的实现类对象表示业务逻辑，然后传递给Channel角色。

   Runnable对象可以作为方法参数，可以放到队列中，可以跨网络传输，也可以保存到文件中。如此则Runnable对象不论传输到哪个计算节点，都可以执行。

5. 多态的Request角色

   本案例中，ClientThread传递给Channel的只是Request实例。但是WorkerThread并不知道Request类的详细信息。

   即使我们传递的是Request的子类给Channel，WorkerThread也可以正常执行execute方法。

   通过Request的多态，可以增加任务的种类，而无需修改Channel角色和Worker角色。

### JUC包和Worker Thread模式

1. ThreadPoolExecutor类

   `java.util.concurrent.ThreadPoolExecutor`类是管理Worker线程的类。可以轻松实现Worker Thread模式。

2. 通过`java.util.concurrent`包创建线程池

   `java.util.concurrent.Executors`类就是创建线程池的工具类。

## 32 Future模式

Future的意思是未来。假设由一个方法需要长时间执行才能获取结果，则一般不会让调用的程序等待，而是先返回给它一张“提货卡”。获取提货卡并不消耗很多时间。该“提货卡”就是Future角色。

获取Future角色的线程稍后使用Future角色来获取运行结果。

### 示例程序

| 类名 | 说明 |
| --- | --- |
| Main | 向Host发出请求并获取数据 |
| Host | 接收请求，并向客户端返回FutureData |
| Data | 访问数据方法的接口。 |
| FutureData | RealData的“提货单”。 |
| RealData | 实际数据的类。构造函数会花费很长时间。 |

**Host类**

```
package com.lagou.concurrent.demo;

public class Host {
    public Data request(final int count, final char c) {
        System.out.println("\trequest(" + count + ", " + c + ") 开始");

        // 创建FutureData对象
        final FutureData future = new FutureData();
        // 启动新线程，创建RealData对象
        new Thread() {
            @Override
            public void run() {
                RealData realData = new RealData(count, c);
                future.setRealData(realData);
            }
        }.start();
        System.out.println("\trequest(" + count + ", " + c + ") 结束");
        // 返回提货单
        return future;
    }
}
```

**Data接口：**

```
package com.lagou.concurrent.demo;

public interface Data {
    String getContent();
}
```

**FutureData类：**

```
package com.lagou.concurrent.demo;

public class FutureData implements Data {

    private RealData realData = null;
    private boolean ready = false;

    public synchronized void setRealData(RealData realData) {
        // balking，如果已经准备好，就返回
        if (ready) {
            return;
        }
        this.realData = realData;
        this.ready = true;
        notifyAll();
    }

    @Override
    public synchronized String getContent() {
        // guarded suspension
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return realData.getContent();
    }
}
```

**RealData类：**

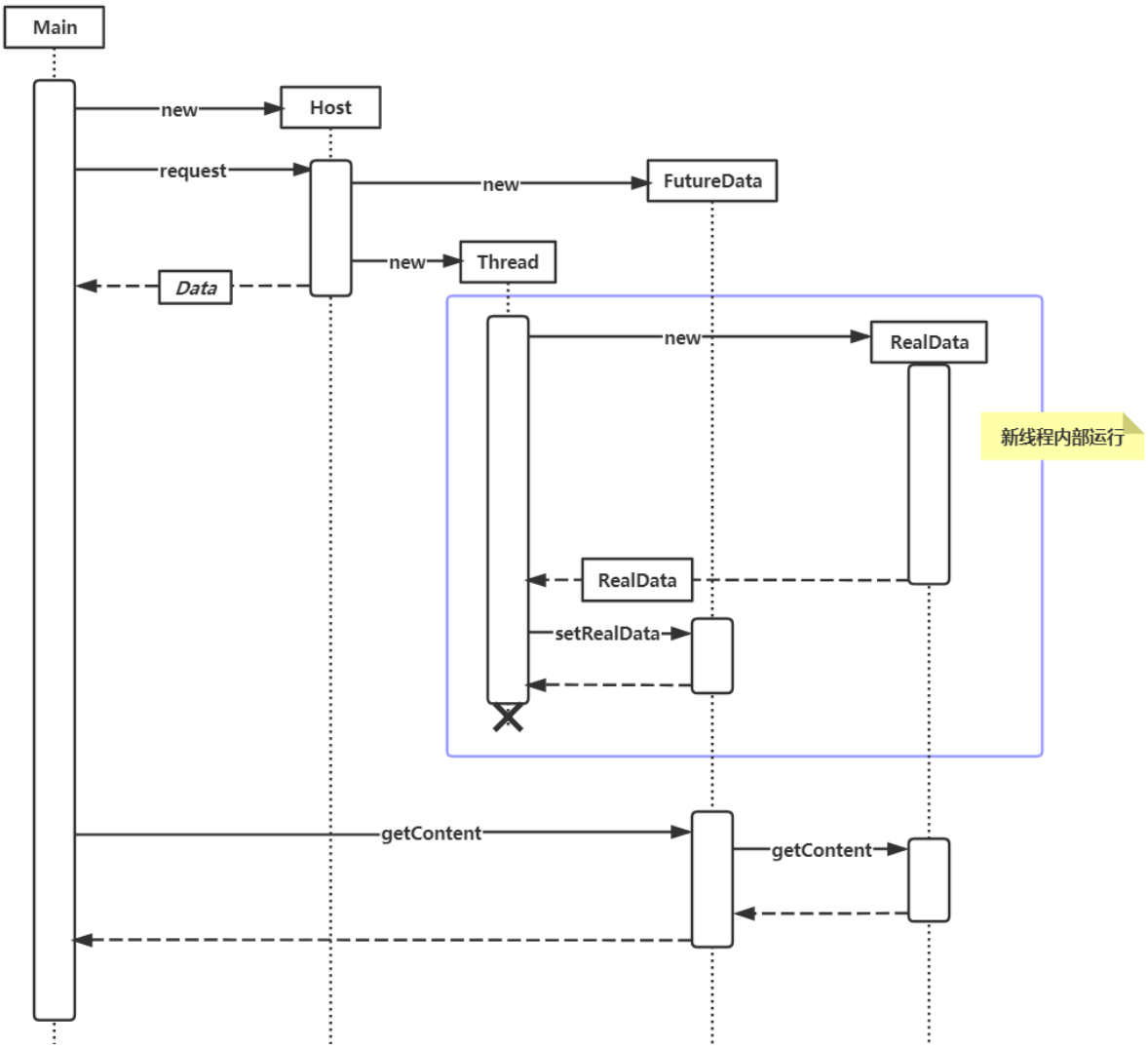
```
package com.lagou.concurrent.demo;

public class RealData implements Data {

    private final String content;

    public RealData(int count, char c) {
        System.out.println("\t组装RealData(" + count + ", " + c + ") 开始");
        char[] buffer = new char[count];
        for (int i = 0; i < count; i++) {
            buffer[i] = c;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
        System.out.println("\t\t组装RealData(" + count + ", " + c + ") 结束");
        this.content = new String(buffer);
    }

    @Override
    public String getContent() {
        return content;
    }
}
```

**Main类：**

```
package com.lagou.concurrent.demo;

public class Main {
    public static void main(String[] args) {
        Host host = new Host();
        Data data1 = host.request(10, 'A');
        Data data2 = host.request(20, 'B');
        Data data3 = host.request(30, 'C');

        System.out.println("等待一会儿再获取结果");

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

        System.out.println("data1 = " + data1.getContent());
        System.out.println("data2 = " + data2.getContent());
        System.out.println("data3 = " + data3.getContent());
    }
}
```

## 流程图




## Future模式中的角色

- Client（请求者）

  Client角色向Host角色发出请求，并立即接收到请求的处理结果——VirtualData角色，也就是Future角色。

  Client角色不必知道返回值是RealData还是Future角色。稍后通过VirtualData角色来操作。

  本案例中，对应Main类。

- Host

  Host角色创建新的线程，由新线程创建RealData角色。同时，Host角色将Future角色（当做VirtualData角色）返回给Client角色。本案例中对应Host类。

- VirtualData（虚拟数据）

  VirtualData角色是让Future角色与RealData角色具有一致性的角色。本案例中对应Data接口。

- RealData（真实数据）

  RealData角色是表示真实数据的角色。创建该对象需要花费很多时间。本案例中对应RealData类。

- Future

  Future角色是RealData角色的“提货单”，由Host角色传递给Client角色。对Client而言，Future角色就是VirtualData角色。当Client角色操作Future角色时线程会wait，直到RealData角色创建完成。

Future角色将Client角色的操作委托给RealData角色。

本案例中，对应于FutureData类。

## 要点：

1. 使用Thread Per Message模式，可以提高程序响应性，但是不能获取结果。Future模式也可以提高程序响应性，还可以获取处理结果。
2. 利用Future模式异步处理特性，可以提高程序吞吐量。虽然并没有减少业务处理的时长，但是如果考虑到I/O，当程序进行磁盘操作时，CPU只是处于等待状态。CPU有空闲时间处理其他的任务。
3. “准备返回值”和“使用返回值”的分离。
4. 如果想等待处理完成后获取返回值，还可以考虑采用回调处理方式。即，当处理完成后，由Host角色启动的线程调用Client角色的方法，进行结果的处理。此时Client角色中的方法需要线程安全地传递返回值。

## JUC包与Future模式

java.util.concurrent包提供了用于支持Future模式的类和接口。

java.util.concurrent.Callable接口将“返回值的某种处理调用”抽象化了。Callable接口声明了call方法。call方法类似于Runnable的run方法，但是call方法有返回值。Callable<String>表示Callable接口的call方法返回值类型为String类型。

java.util.concurrent.Future接口相当于本案例中的Future角色。Future接口声明了get方法来获取结果，但是没有声明设置值的方法。设置值的方法需要在Future接口的实现类中声明。Future<String>表示“Future接口的get方法返回值类型是String类型”。除了get方法，Future接口还声明了用于中断运行的cancel方法。

java.util.concurrent.FutureTask类是实现了Future接口的标准类。FutureTask类声明了用于获取值的get方法、用于中断运行的cancel方法、用于设置值的set方法，以及用于设置异常的setException方法。由于FutureTask类实现了Runnable接口，还声明了run方法。

**Callable、Future、FutureTask的类图**