第零节：小册简介

来介绍一下《二哥的并发编程进阶之路》小册吧。小册主要围绕着 Java 中的并发编程/多线程展开，一共 31 个小节，15 万+字，手绘图 200+张，耗费了巨大的心血，以下是小册的个人信息。

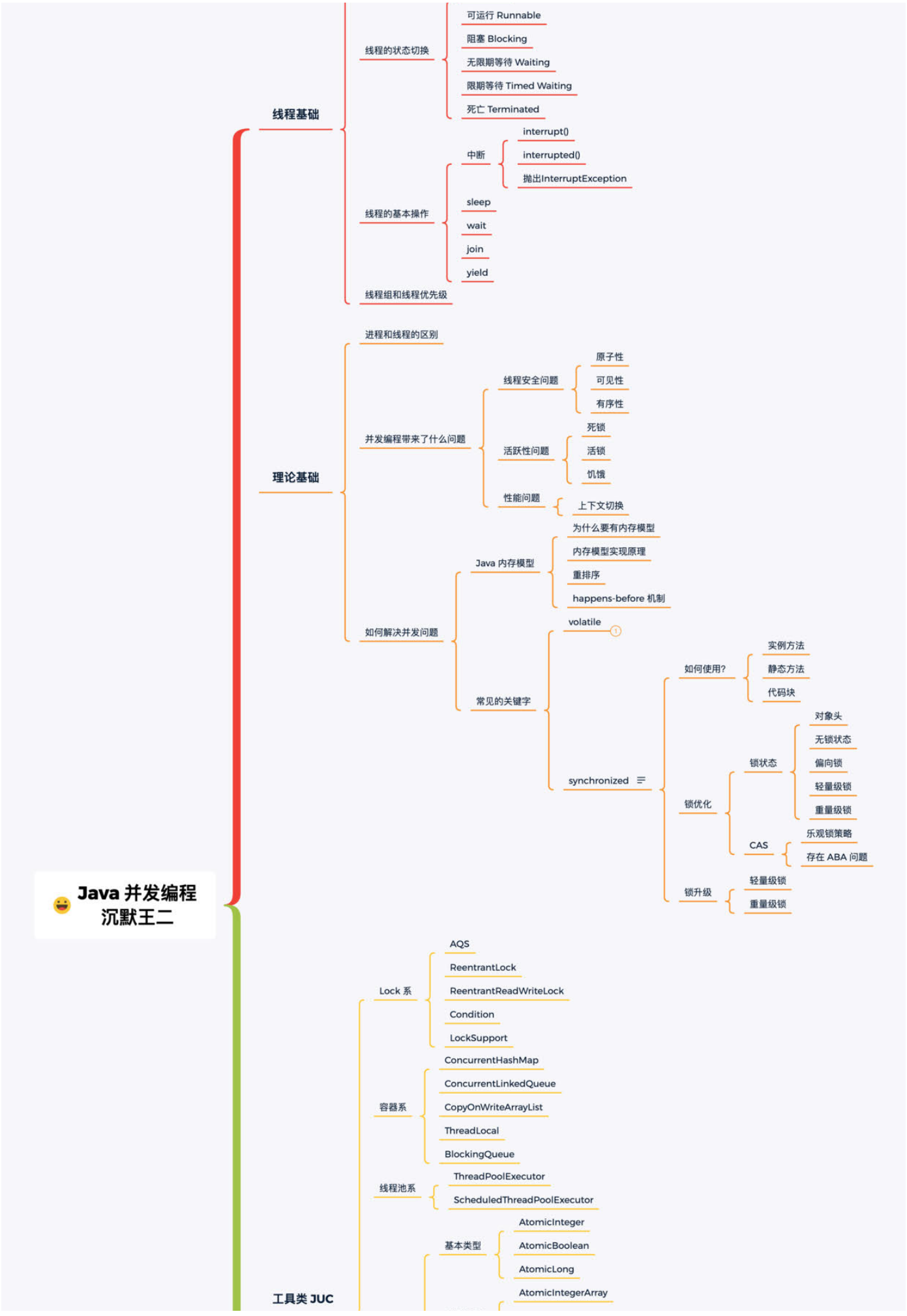
- 小册名字：二哥的并发编程进阶之路
- 小册作者：沉默王二
- 小册品质：该小册的内容来源于二哥在 GitHub 上开源的知识库《Java 进阶之路》，能在 GitHub 取得 9300+ star 可以说品质是有目共睹，尤其是国内还有不少小伙伴在访问 GitHub 的时候很不顺利。
- 小册初衷：面试过小伙伴应该感受比较深，并发编程在 Java 岗的面试中占比挺大，尤其是去一些知名公司的话，像 volatile、sychronized、CAS、AQS、线程池、JUC 包、原子类、ThreadLocal、生产者-消费者模式等内容，都是面试中经常被考察的内容。另外，工作一两年后的初级程序员，如果想进阶为高级程序员，并发编程的内容也是完全绕不开的，二哥之所以花大力气整理《并发编程小册》的原因也在于此，就是希望能帮助大家轻松且深刻地掌握并发编程/多线程方面的知识。
- 小册简介：主要针对并发编程进行讲解，包括并发编程的基础知识、并发编程的高级知识、并发编程的源码分析、并发编程的面试题等，内容涵盖了 Java 并发编程的方方面面，是一份非常适合 Java 并发编程的学习资料。
- 小册说明：小册算是《二哥的 Java 进阶之路基础篇》的姐妹篇，可通过 GitHub 阅读或者二哥的网站在线阅读，同时提供 PDF 版本，15 万+字，手绘图 200+张，有亮白版、暗黑版和 epub 三个版本，前后耗费 3 个多月的时间，很不容易，希望你能好好珍惜。

小册包含哪些内容?

《二哥的并发编程进阶之路》主要包含下面这些内容：

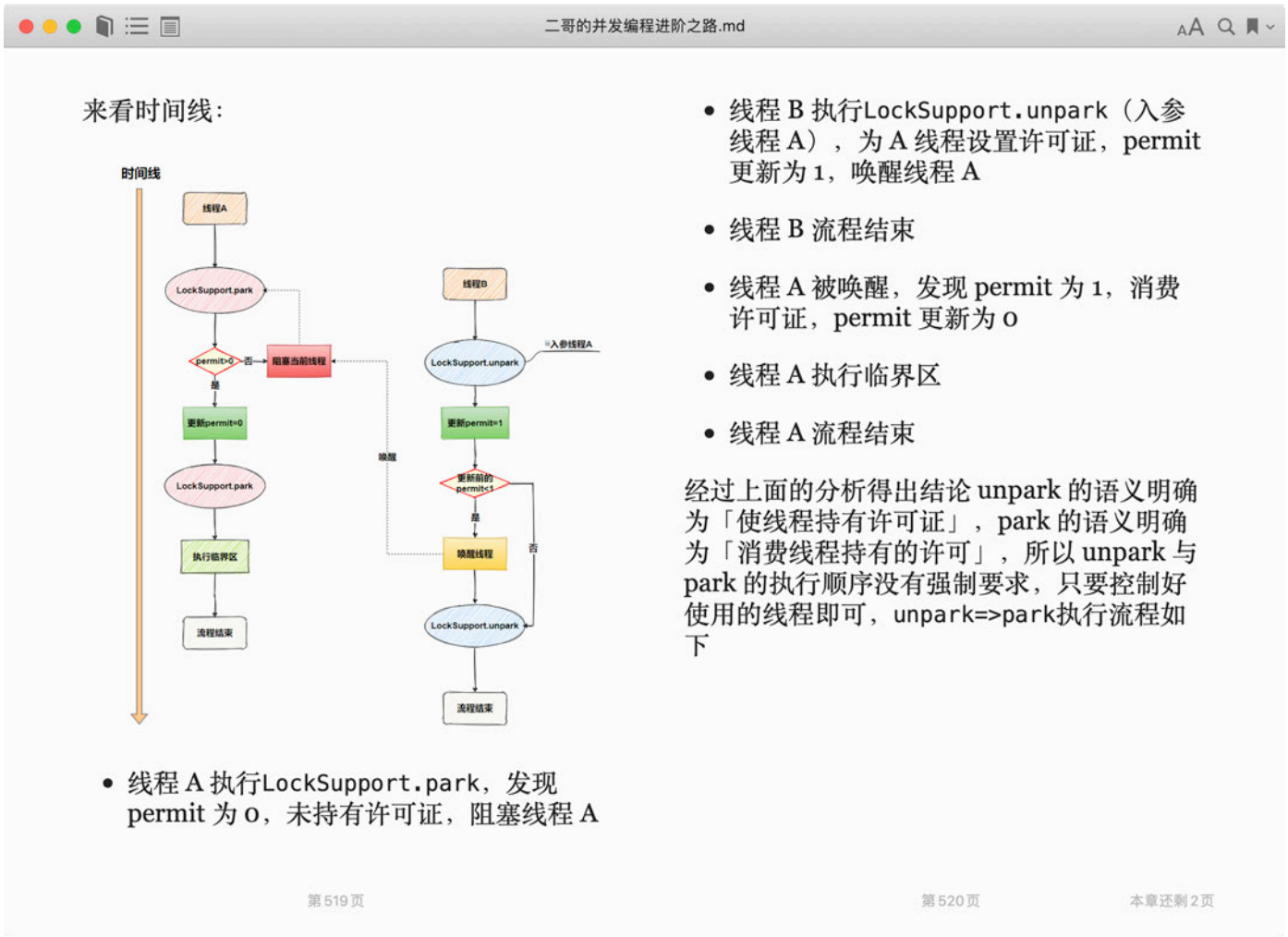
- 线程的基本概念和使用方法
- 进程和线程的区别
- 多线程带来了哪些问题
- Java的内存模型
- sychronized 关键字
- volatile 关键字
- CAS 原理
- AQS 原理
- ReentrantLock
- ReentrantReadWriteLock
- Condition
- CountDownLatch
- 线程池
- 并发容器（ConcurrentHashMap、ConcurrentLinkedQueue、BlockingQueue、CopyOnWriteArrayList）
- 本地变量 ThreadLocal
- 生产者消费者模型

一共 31 篇内容，共计 15 万+ 字，用一张思维导图来做个总结吧。


Java 并发编程
沉默王二
线程基础
线程的状态切换
可运行 Runnable
阻塞 Blocking
无限期等待 Waiting
限期等待 Timed Waiting
死亡 Terminated
线程的基本操作
中断
interrupt()
interrupted()
抛出InterruptException
sleep
wait
join
yield
线程组和线程优先级
理论基础
进程和线程的区别
并发编程带来了什么问题
线程安全问题
原子性
可见性
有序性
活跃性问题
死锁
活锁
饥饿
性能问题
上下文切换
如何解决并发问题
Java 内存模型
为什么要有内存模型
内存模型实现原理
重排序
happens-before 机制
常见的关键字
volatile
synchronized
如何使用?
实例方法
静态方法
代码块
锁优化
锁状态
对象头
无锁状态
偏向锁
轻量级锁
重量级锁
CAS
乐观锁策略
存在 ABA 问题
锁升级
轻量级锁
重量级锁
工具类 JUC
Lock 系
AQS
ReentrantLock
ReentrantReadWriteLock
Condition
LockSupport
容器系
ConcurrentHashMap
ConcurrentLinkedQueue
CopyOnWriteArrayList
ThreadLocal
BlockingQueue
线程池系
ThreadPoolExecutor
ScheduledThreadPoolExecutor
基本类型
AtomicInteger
AtomicBoolean
AtomicLong
AtomicIntegerArray

这里展示一下暗黑版的 PDF 视图，大家先感受一下，手绘图都画得非常用心。

这是 epub 版本的阅读效果，感觉左右翻动的效果好舒服，一次可以看两页，真的就像在读纸质版书籍一样，体验非常棒。



如果你喜欢在线阅读，请戳下面这个网址：

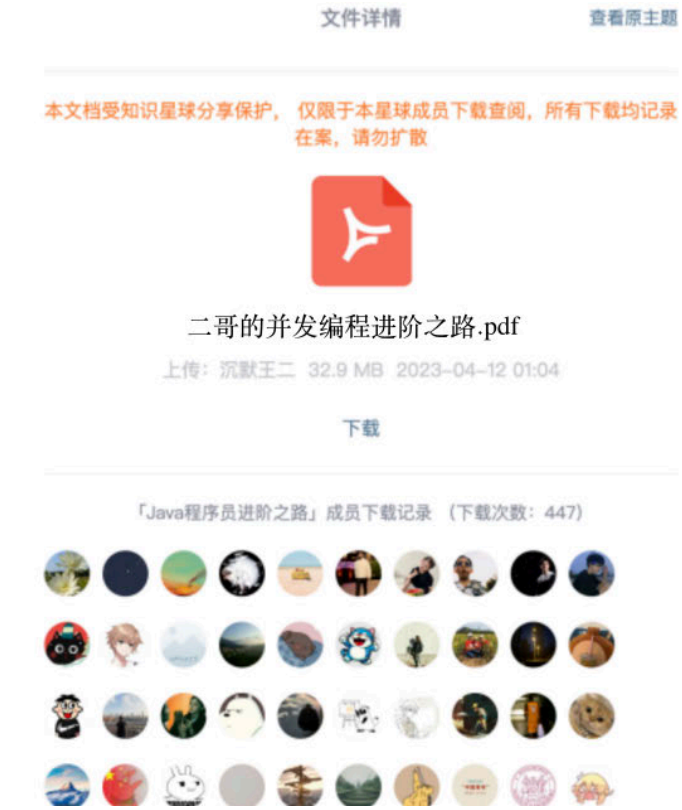
> https://javabetter.cn/xuexiluxian/java/thread.html

如果你在阅读过程中感觉这份小册写的还不错，甚至有亿点点收获，我的虚荣心也会得到恰当的满足😜

如何获取最新版?

小册分为 3 个版本，暗黑版（适合夜服）、亮白版（适合打印）、epub 版，绝对不虚市面上任何一本 Java 并发编程/多线程的实体书！

小册会持续保持**更新**，如果想获得最新版，请加入二哥的编程星球后在星球第二个置顶帖「知识图谱」中获取 PDF 版本，15 万+字，200+ 张手绘图。前后耗费 3 个多月的时间，很不容易，希望你能认真阅读。



第一节：Java多线程入门

对于 Java 初学者来说，多线程的很多概念听起来就很难理解。比方说：

- 进程，是对运行时程序的封装，是系统进行资源调度和分配的基本单位，实现了操作系统的并发。
- 线程，是进程的子任务，是 CPU 调度和分派的基本单位，实现了进程内部的并发。

很抽象，对不对？打个比喻，你在打一把王者（其实我不会玩哈 doge）：

- 进程可以比作是你开的这一把游戏
- 线程可以比作是你所选的英雄或者是游戏中的水晶野怪等之类的。

带着这个比喻来理解进程和线程的一些关系，一个进程可以有多个线程就叫多线程。是不是感觉非常好理解了？

进程和线程

❤1、线程在进程下进行

（单独的英雄角色、野怪、小兵肯定不能运行）

❤2、进程之间不会相互影响，主线程结束将会导致整个进程结束

（两把游戏之间不会有联系和影响。你的水晶被推掉，你这把游戏就结束了）

❤3、不同的进程数据很难共享

（两把游戏之间很难有联系，有联系的情况比如上把的敌人这把又匹配到了）

❤4、同进程下的不同线程之间数据很容易共享

（你开的那一把游戏，你可以看到每个玩家的状态——生死，也可以看到每个玩家的出装等等）

❤5、进程使用内存地址可以限定使用量

（开的房间模式，决定了你可以设置有多少人进，当房间满了后，其他人就进不去了，除非有人退出房间，其他人才能进）

创建线程的三种方式

搞清楚上面这些概念之后，我们来看一下多线程创建的三种方式：

继承 Thread 类

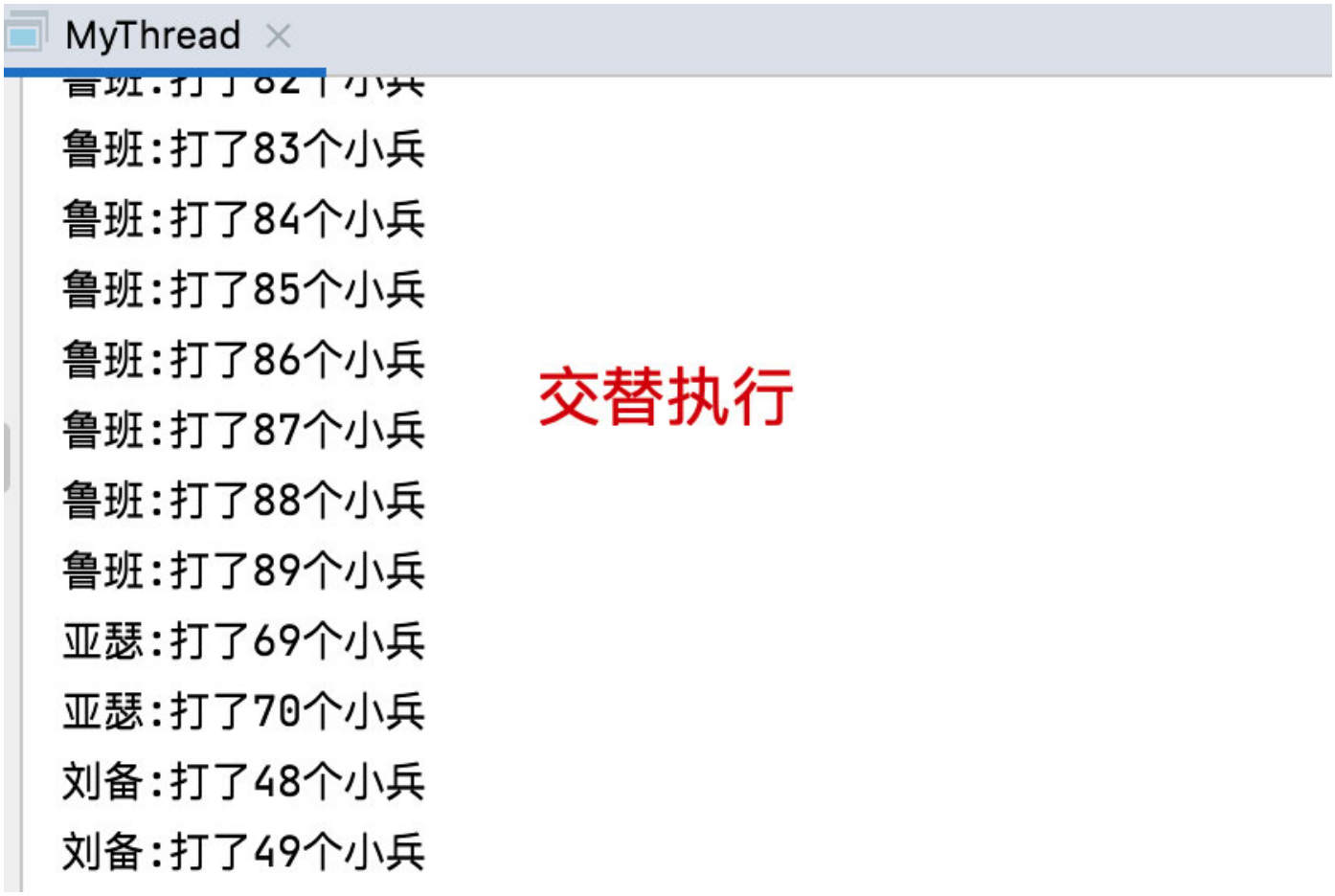
♠①：创建一个类继承 Thread 类，并重写 run 方法。

```
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(getName() + ":打了" + i + "个小兵");
        }
    }
}
```

我们来写个测试方法验证下：

```
//创建MyThread对象
MyThread t1=new  MyThread();
MyThread t2=new  MyThread();
MyThread t3=new  MyThread();
//设置线程的名字
t1.setName("鲁班");
t2.setName("刘备");
t3.setName("亚瑟");
//启动线程
t1.start();
t2.start();
t3.start();
```

来看一下执行后的结果：



实现 Runnable 接口

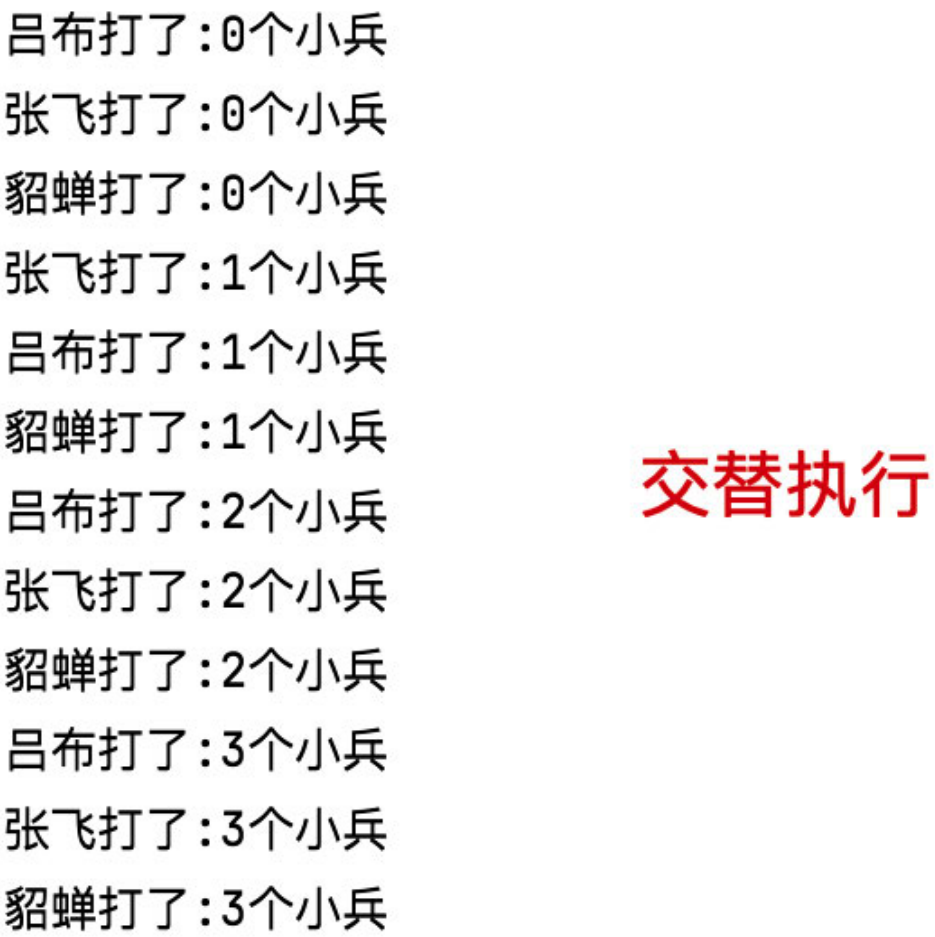
♠②：创建一个类实现 Runnable 接口，并重写 run 方法。

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {//sleep会发生异常要显示处理
                Thread.sleep(20);//暂停20毫秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "打了:" + i + "个小
兵");
        }
    }
}
```

我们来写个测试方法验证下：

```
//创建MyRunnable类
MyRunnable mr = new MyRunnable();
//创建Thread类的有参构造,并设置线程名
Thread t1 = new Thread(mr, "张飞");
Thread t2 = new Thread(mr, "貂蝉");
Thread t3 = new Thread(mr, "吕布");
//启动线程
t1.start();
t2.start();
t3.start();
```

来看一下执行后的结果：



实现 Callable 接口

♠③：实现 Callable 接口，重写 call 方法，这种方式可以通过 FutureTask 获取任务执行的返回值。

```
public class CallerTask implements Callable<String> {
    public String call() throws Exception {
        return "Hello,i am running!";
    }

    public static void main(String[] args) {
        //创建异步任务
        FutureTask<String> task=new FutureTask<String>(new CallerTask());
        //启动线程
        new Thread(task).start();
        try {
            //等待执行完成，并获取返回结果
            String result=task.get();
            System.out.println(result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

关于线程的一些疑问

❤1、为什么要重写 run 方法?

这是因为默认的`run()`方法不会做任何事情。

为了让线程执行一些实际的任务，我们需要提供自己的`run()`方法实现，这就需要重写`run()`方法。

```
public class MyThread extends Thread {
  public void run() {
    System.out.println("MyThread running");
  }
}
```

在这个例子中，我们重写了`run()`方法，使其打印出一条消息。当我们创建并启动这个线程的实例时，它就会打印出这条消息。

❤2、run 方法和 start 方法有什么区别?

- `run()`：封装线程执行的代码，直接调用相当于调用普通方法。
- `start()`：启动线程，然后由 JVM 调用此线程的`run()`方法。

❤3、通过继承 Thread 的方法和实现 Runnable 接口的方式创建多线程，哪个好?

实现 Runable 接口好，原因有两个：

- ♠①、避免了 Java 单继承的局限性，Java 不支持多重继承，因此如果我们的类已经继承了另一个类，就不能再继承 Thread 类了。
- ♠②、适合多个相同的程序代码去处理同一资源的情况，把线程、代码和数据有效的分离，更符合面向对象的设计思想。Callable 接口与 Runnable 非常相似，但可以返回一个结果。

控制线程的其他方法

针对线程控制，大家还会遇到 3 个常见的方法，我们来一一介绍下。

1）sleep()

使当前正在执行的线程暂停指定的毫秒数，也就是进入休眠的状态。

需要注意的是，sleep 的时候要对异常进行处理。

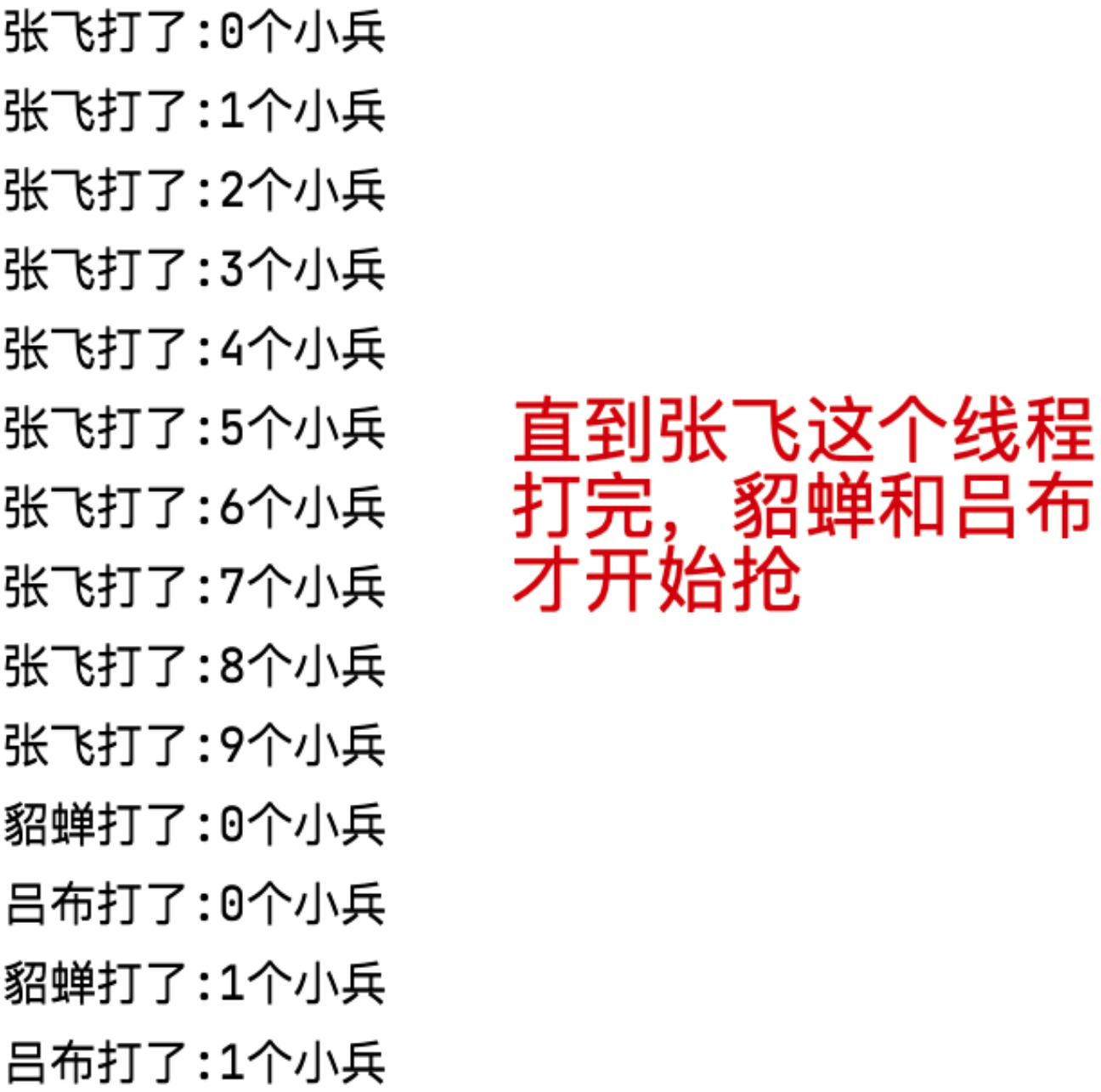
```
try {//sleep会发生异常要显示处理
    Thread.sleep(20);//暂停20毫秒
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

2）join()

等待这个线程执行完才会轮到后续线程得到 cpu 的执行权，使用这个也要捕获异常。

```
//创建MyRunnable类
MyRunnable mr = new MyRunnable();
//创建Thread类的有参构造,并设置线程名
Thread t1 = new Thread(mr, "张飞");
Thread t2 = new Thread(mr, "貂蝉");
Thread t3 = new Thread(mr, "吕布");
//启动线程
t1.start();
try {
    t1.join(); //等待t1执行完才会轮到t2, t3抢
} catch (InterruptedException e) {
    e.printStackTrace();
}
t2.start();
t3.start();
```

来看一下执行后的结果：



3）setDaemon()

将此线程标记为守护线程，准确来说，就是服务其他的线程，像 Java 中的垃圾回收线程，就是典型的守护线程。

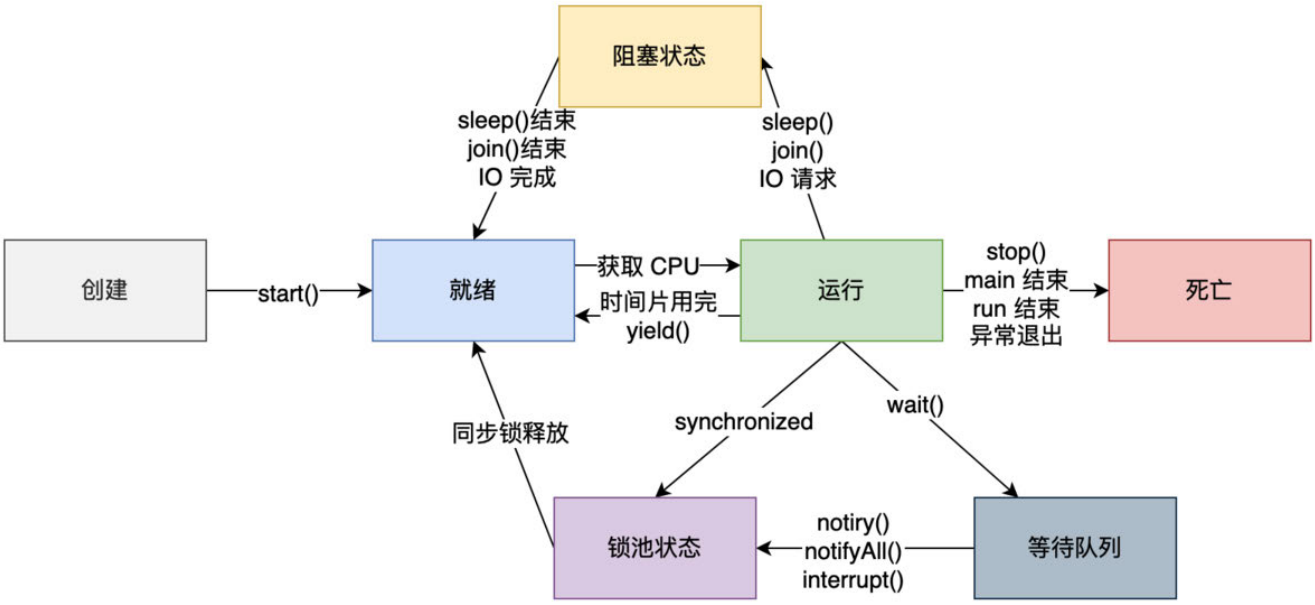
```
//创建MyRunnable类
MyRunnable mr = new MyRunnable();
//创建Thread类的有参构造,并设置线程名
Thread t1 = new Thread(mr, "张飞");
Thread t2 = new Thread(mr, "貂蝉");
Thread t3 = new Thread(mr, "吕布");

t1.setDaemon(true);
t2.setDaemon(true);

//启动线程
t1.start();
```

```
t2.start();
t3.start();
```

如果其他线程都执行完毕，main 方法（主线程）也执行完毕，JVM 就会退出，也就是停止运行。如果 JVM 都停止运行了，守护线程自然也就停止了。

小结

本文主要介绍了 Java 多线程的创建方式，以及线程的一些常用方法。最后再来看一下线程的生命周期吧，一图胜千言。



GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。

第二节：获取线程的执行结果

在第一节：初步掌握 Java 多线程中，我们讲述了创建线程的 3 种方式，一种是直接继承 Thread，一种是实现 Runnable 接口，另外一种是实现 Callable 接口。

前 2 种方式都有一个缺陷：在执行完任务之后无法获取执行结果。

如果需要获取执行结果，就必须通过共享变量或者线程通信的方式来达到目的，这样使用起来就比较麻烦。

Java 1.5 提供了 Callable、Future、FutureTask，它们可以在任务执行完后得到执行结果，今天我们就来详细的了解一下。

无返回值的 Runnable

由于 `Runnable` 的 `run()` 方法的返回值为 void：

```
public interface Runnable {
    public abstract void run();
}
```

所以在执行完任务之后无法返回任何结果。

有返回值的 Callable

Callable 位于 `java.util.concurrent` 包下，也是一个接口，它定义了一个 `call()` 方法：

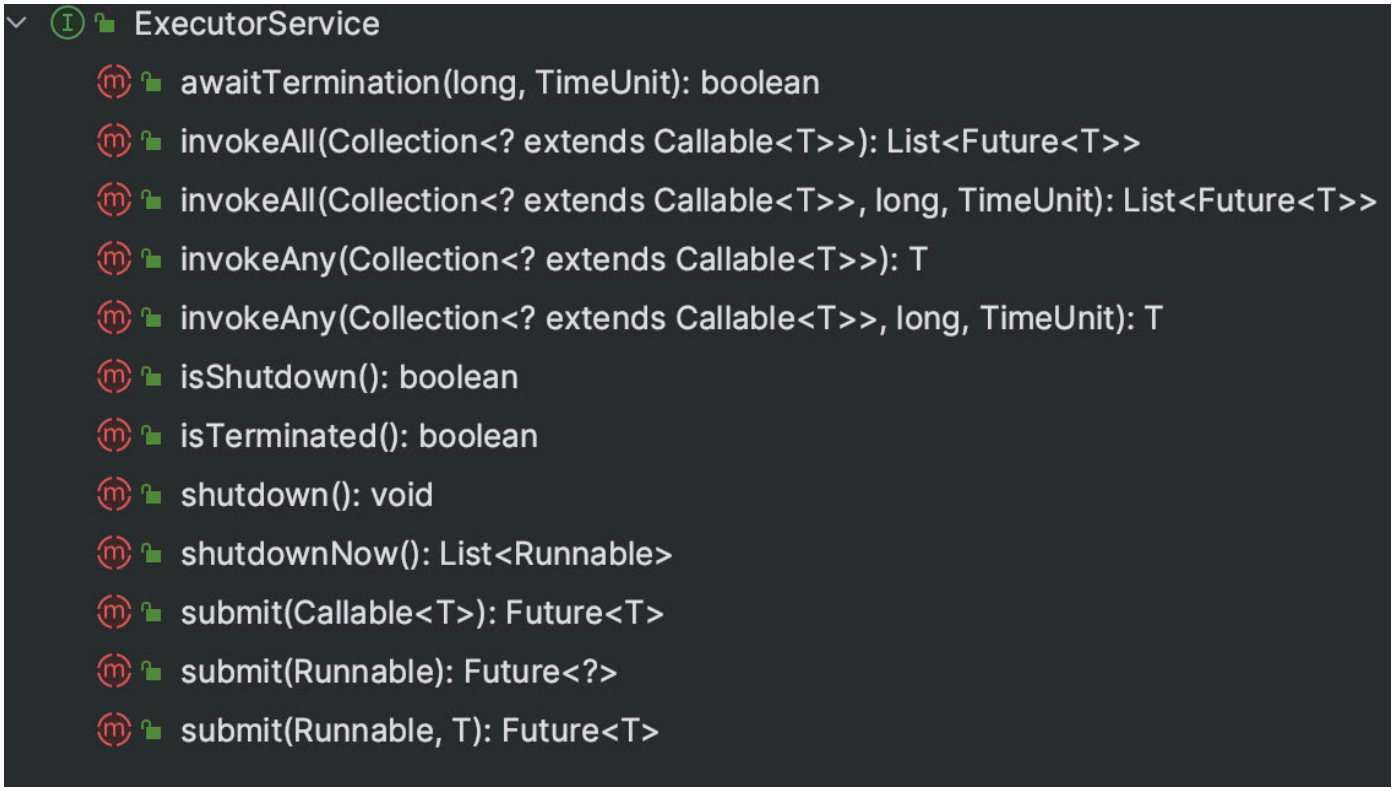
```
public interface Callable<V> {
    V call() throws Exception;
}
```

可以看到，`call()` 方法返回的类型是一个 V 类型的泛型。

那怎么使用 Callable 呢?

一般会配合 ExecutorService（后面在讲线程池的时候会细讲，这里记住就行）来使用。

ExecutorService 是一个接口，位于 `java.util.concurrent` 包下，它是 Java 线程池框架的核心接口，用来异步执行任务。它提供了一些关键方法用来进行线程管理。



下面的例子就用到了 ExecutorService 的 submit 方法。

```
// 创建一个包含5个线程的线程池
ExecutorService executorService = Executors.newFixedThreadPool(5);

// 创建一个Callable任务
Callable<String> task = new Callable<String>() {
    public String call() {
        return "Hello from " + Thread.currentThread().getName();
    }
};

// 提交任务到ExecutorService执行，并获取Future对象
Future[] futures = new Future[10];
for (int i = 0; i < 10; i++) {
    futures[i] = executorService.submit(task);
}
```

```
// 通过Future对象获取任务的结果
for (int i = 0; i < 10; i++) {
    System.out.println(futures[i].get());
}

// 关闭ExecutorService，不再接受新的任务，等待所有已提交的任务完成
executorService.shutdown();
```

我们通过 Executors 工具类来创建一个 ExecutorService，然后向里面提交 Callable 任务，然后通过 Future 来获取执行结果。

为了做对比，我们再来看一下使用 Runnable 的方式：

```
// 创建一个包含5个线程的线程池
ExecutorService executorService = Executors.newFixedThreadPool(5);

// 创建一个Runnable任务
Runnable task = new Runnable() {
    public void run() {
        System.out.println("Hello from " + Thread.currentThread().getName());
    }
};

// 提交任务到ExecutorService执行
for (int i = 0; i < 10; i++) {
    executorService.submit(task);
}

// 关闭ExecutorService，不再接受新的任务，等待所有已提交的任务完成
executorService.shutdown();
```

可以看到，使用 Runnable 的方式要比 Callable 的方式简单一些，但是 Callable 的方式可以获取执行结果，这是 Runnable 做不到的。

异步计算结果 Future 接口

在前面的例子中，我们通过 Future 来获取 Callable 任务的执行结果，那么 Future 是什么呢?

Future 位于 `java.util.concurrent` 包下，它是一个接口：

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

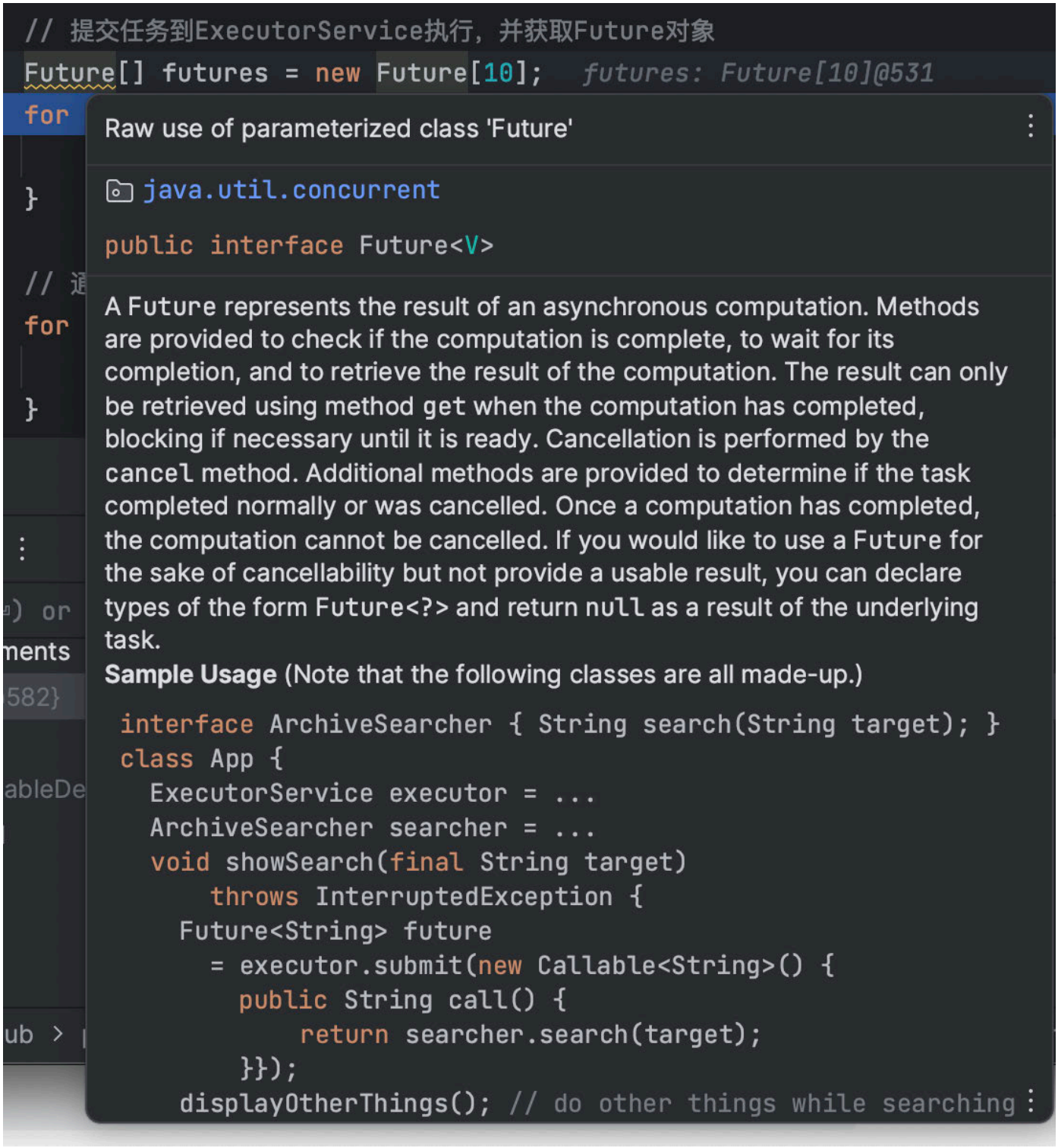
一共声明了 5 个方法：

- `cancel()` 方法用来取消任务，如果取消任务成功则返回 true，如果取消任务失败则返回 false。参数 mayInterruptIfRunning 表示是否允许取消正在执行却没有执行完毕的任务，如果设置 true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论 mayInterruptIfRunning 为 true 还是 false，此方法肯定返回 false，即如果取消已经完成的任务会返回 false；如果任务正在执行，若 mayInterruptIfRunning 设置为 true，则返回 true，若 mayInterruptIfRunning 设置为 false，则返回 false；如果任务还没有执行，则无论 mayInterruptIfRunning 为 true 还是 false，肯定返回 true。
- `isCancelled()` 方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
- `isDone()` 方法表示任务是否已经完成，若任务完成，则返回 true；
- `get()`方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；
- `get(long timeout, TimeUnit unit)`用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回 null。

也就是说 Future 提供了三种功能：

- 1）判断任务是否完成；
- 2）能够中断任务；
- 3）能够获取任务执行结果。

由于 Future 只是一个接口，如果直接 new 的话，编译器是会有一个 ⚠️ 警告的，它会提醒我们最好使用 FutureTask。



实际上，FutureTask 是 Future 接口的一个唯一实现类，我们在前面的例子中 `executorService.submit()` 返回的就是 FutureTask，通过 debug 模式可以观察到。

异步计算结果 FutureTask 实现类

我们来看一下 FutureTask 的实现：

```
public class FutureTask<V> implements RunnableFuture<V>
```

FutureTask 类实现了 RunnableFuture 接口，我们看一下 RunnableFuture 接口的实现：

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

可以看出 RunnableFuture 继承了 Runnable 接口和 Future 接口，而 FutureTask 实现了 RunnableFuture 接口。所以它既可以作为 Runnable 被线程执行，又可以作为 Future 得到 Callable 的返回值。

FutureTask 提供了 2 个构造器：

```
public FutureTask(Callable<V> callable) {
}
public FutureTask(Runnable runnable, V result) {
}
```

当需要异步执行一个计算并在稍后的某个时间点获取其结果时，就可以使用 FutureTask。来个例子

```
// 创建一个固定大小的线程池
ExecutorService executorService = Executors.newFixedThreadPool(3);

// 创建一系列 Callable
Callable<Integer>[] tasks = new Callable[5];
```

```
for (int i = 0; i < tasks.length; i++) {
    final int index = i;
    tasks[i] = new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            TimeUnit.SECONDS.sleep(index + 1);
            return (index + 1) * 100;
        }
    };
}

// 将 Callable 包装为 FutureTask，并提交到线程池
FutureTask<Integer>[] futureTasks = new FutureTask[tasks.length];
for (int i = 0; i < tasks.length; i++) {
    futureTasks[i] = new FutureTask<>(tasks[i]);
    executorService.submit(futureTasks[i]);
}

// 获取任务结果
for (int i = 0; i < futureTasks.length; i++) {
    System.out.println("Result of task" + (i + 1) + ": " + futureTasks[i].get());
}

// 关闭线程池
executorService.shutdown();
```

来看一下输出结果

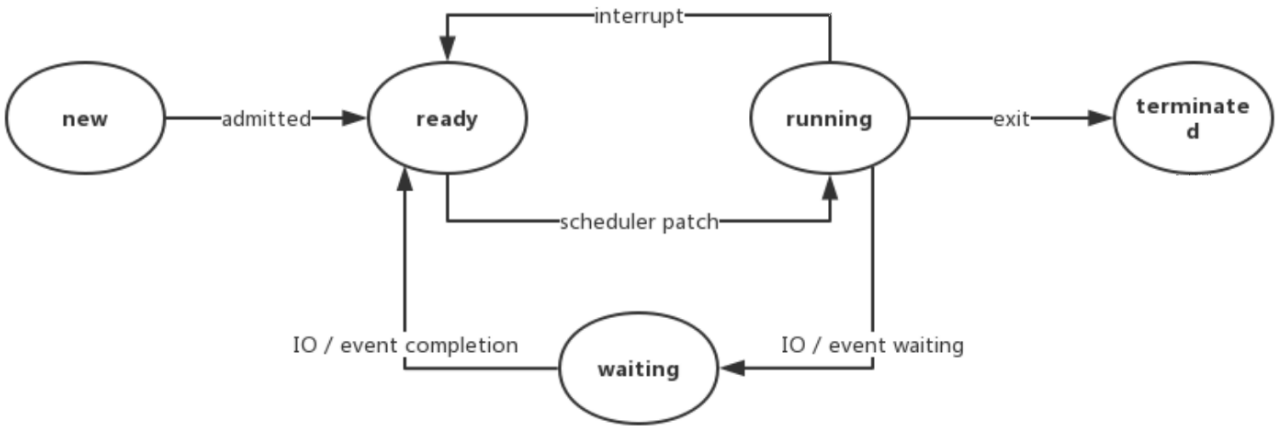
```
Result of task1: 100
Result of task2: 200
Result of task3: 300
Result of task4: 400
Result of task5: 500
```

小结

本文深入解释了如何在 Java 中使用 Callable、Future 和 FutureTask 来获取多线程执行结果。

第三节：Java 线程的 6 种状态

我们先来看看操作系统中的线程状态转换。在操作系统中，线程被视为轻量级的进程，所以**线程状态其实和进程状态是一致的**。



操作系统的线程主要有以下三个状态：

- 就绪状态(ready)：线程正在等待使用 CPU，经调度程序调用之后进入 running 状态。
- 执行状态(running)：线程正在使用 CPU。
- 等待状态(waiting): 线程经过等待事件的调用或者正在等待其他资源（如 I/O）。

然后我们来看 Java 线程的 6 个状态：

```
// Thread.State 源码
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```

NEW

处于 NEW 状态的线程此时尚未启动。这里的尚未启动指的是还没调用 Thread 实例的`start()`方法。

```
private void testStateNew() {
    Thread thread = new Thread(() -> {});
    System.out.println(thread.getState()); // 输出 NEW
}
```

从上面可以看出，只是创建了线程而并没有调用 start 方法，此时线程处于 NEW 状态。

关于 start 的两个引申问题

1. 反复调用同一个线程的 start 方法是否可行?
2. 假如一个线程执行完毕（此时处于 TERMINATED 状态），再次调用这个线程的 start 方法是否可行?

要分析这两个问题，我们先来看看`start()`的源码：

```
// 使用synchronized关键字保证这个方法是线程安全的
public synchronized void start() {
    // threadStatus != 0 表示这个线程已经被启动过或已经结束了
    // 如果试图再次启动这个线程，就会抛出IllegalThreadStateException异常
    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    // 将这个线程添加到当前线程的线程组中
    group.add(this);

    // 声明一个变量，用于记录线程是否启动成功
    boolean started = false;
    try {
        // 使用native方法启动这个线程
        start0();
        // 如果没有抛出异常，那么started被设为true，表示线程启动成功
        started = true;
    } finally {
        // 在finally语句块中，无论try语句块中的代码是否抛出异常，都会执行
        try {
```

```
            // 如果线程没有启动成功，就从线程组中移除这个线程
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            // 如果在移除线程的过程中发生了异常，我们选择忽略这个异常
        }
    }
}
```

可以看到，在`start()`内部，有一个 threadStatus 变量。如果它不等于 0，调用`start()`会直接抛出异常。

接着往下看，有一个 native 的 `start0()` 方法。这个方法并没有对**threadStatus**进行处理。到这里我们仿佛拿这个 threadStatus 没辙了，通过 debug 再看一下:

```
@Test
public void testStartMethod() {
    Thread thread = new Thread(() -> {});
    thread.start(); // 第一次调用
    thread.start(); // 第二次调用
}
```

在 start 方法内部的最开始打断点：

- 第一次调用时 threadStatus 的值是 0。
- 第二次调用时 threadStatus 的值不为 0。

查看当前线程状态的源码：

```
// Thread.getState方法源码：
public State getState() {
    // get current thread state
    return sun.misc.VM.toThreadState(threadStatus);
}

// sun.misc.VM 源码：
// 如果线程的状态值和4做位与操作结果不为0，线程处于RUNNABLE状态。
// 如果线程的状态值和1024做位与操作结果不为0，线程处于BLOCKED状态。
// 如果线程的状态值和16做位与操作结果不为0，线程处于WAITING状态。
// 如果线程的状态值和32做位与操作结果不为0，线程处于TIMED_WAITING状态。
// 如果线程的状态值和2做位与操作结果不为0，线程处于TERMINATED状态。
// 最后，如果线程的状态值和1做位与操作结果为0，线程处于NEW状态，否则线程处于RUNNABLE状态。
public static State toThreadState(int var0) {
    if ((var0 & 4) != 0) {
        return State.RUNNABLE;
    } else if ((var0 & 1024) != 0) {
        return State.BLOCKED;
    } else if ((var0 & 16) != 0) {
        return State.WAITING;
```

```
    } else if ((var0 & 32) != 0) {
        return State.TIMED_WAITING;
    } else if ((var0 & 2) != 0) {
        return State.TERMINATED;
    } else {
        return (var0 & 1) == 0 ? State.NEW : State.RUNNABLE;
    }
}
```

还记得我们引申的两个问题吗?

1. 反复调用同一个线程的 start 方法是否可行?
2. 假如一个线程执行完毕（此时处于 TERMINATED 状态），再次调用这个线程的 start 方法是否可行?

结合上面的源码可以得到的答案是：

1. 都不行，在调用 start 之后，threadStatus 的值会改变（`threadStatus !=0`），再次调用 start 方法会抛出 IllegalThreadStateException 异常。
2. threadStatus 为 2 代表当前线程状态为 TERMINATED（下面会讲）。

RUNNABLE

表示当前线程正在运行中。处于 RUNNABLE 状态的线程在 Java 虚拟机中运行，也有可能在等待 CPU 分配资源。

我们来看看 Thread 源码里对 RUNNABLE 状态的定义：

```
/**
 * Thread state for a runnable thread.  A thread in the runnable
 * state is executing in the Java virtual machine but it may
 * be waiting for other resources from the operating system
 * such as processor.
 */
```

意思大家应该都能看得懂，不懂翻译一下（其实前面已经翻译过了）。

也就是说，Java 线程的**RUNNABLE**状态其实包括了操作系统线程的**ready**和**running**两个状态。

BLOCKED

阻塞状态。处于 BLOCKED 状态的线程正等待锁（锁会在后面细讲）的释放以进入同步区。

我们用 BLOCKED 状态举个生活中的例子：

假如今天你下班后准备去食堂吃饭。你来到食堂仅有的一个窗口，发现前面已经有个人在窗口前了，此时你必须得等前面的人从窗口离开才行。

假设你是线程 t2，你前面的那个人是线程 t1。此时 t1 占有了锁（食堂唯一的窗口），t2 正在等待锁的释放，所以此时 t2 就处于 BLOCKED 状态。

WAITING

等待状态。处于等待状态的线程变成 RUNNABLE 状态需要其他线程唤醒。

调用下面这 3 个方法会使线程进入等待状态：

- `Object.wait()`：使当前线程处于等待状态直到另一个线程唤醒它；
- `Thread.join()`：等待线程执行完毕，底层调用的是 Object 的 wait 方法；
- `LockSupport.park()`：除非获得调用许可，否则禁用当前线程进行线程调度。LockSupport 我们在后面会细讲。

我们延续上面的例子继续解释一下 WAITING 状态：

你等了好几分钟，终于轮到你了，突然你们有一个“不懂事”的经理来了。你看到他你就有一种不祥的预感，果然，他是来找你的。

他把你拉到一旁叫你待会儿再吃饭，说他下午要去作报告，赶紧来找你了解一下项目的情况。你心里虽然有一万个不愿意但是你还是从食堂窗口走开了。

此时，假设你还是线程 t2，你的经理是线程 t1。虽然你此时都占有锁（窗口）了，“不速之客”来了你还是得释放掉锁。此时你 t2 的状态就是 WAITING。然后经理 t1 获得锁，进入 RUNNABLE 状态。

要是经理 t1 不主动唤醒你 t2（notify、notifyAll..），可以说你 t2 只能一直等待了。

TIMED_WAITING

超时等待状态。线程等待一个具体的时间，时间到后会被自动唤醒。

调用如下方法会使线程进入超时等待状态：

- `Thread.sleep(long millis)`：使当前线程睡眠指定时间；
- `Object.wait(long timeout)`：线程休眠指定时间，等待期间可以通过`notify()`/`notifyAll()`唤醒；
- `Thread.join(long millis)`：等待当前线程最多执行 millis 毫秒，如果 millis 为 0，则会一直执行；
- `LockSupport.parkNanos(long nanos)`： 除非获得调用许可，否则禁用当前线程进行线程调度指定时间；LockSupport 我们在后面会细讲；
- `LockSupport.parkUntil(long deadline)`：同上，也是禁止线程进行调度指定时间；

我们继续延续上面的例子来解释一下 TIMED_WAITING 状态：

到了第二天中午，又到了饭点，你还是到了窗口前。

突然间想起你的同事叫你等他一起，他说让你等他十分钟他改个 bug。

好吧，那就等等吧，你就离开了窗口。很快十分钟过去了，你见他还没来，你想都等了这么久了还不来，那你还是先去吃饭好了。

这时你还是线程 t1，你改 bug 的同事是线程 t2。t2 让 t1 等待了指定时间，此时 t1 等待期间就属于 TIMED_WATING 状态。

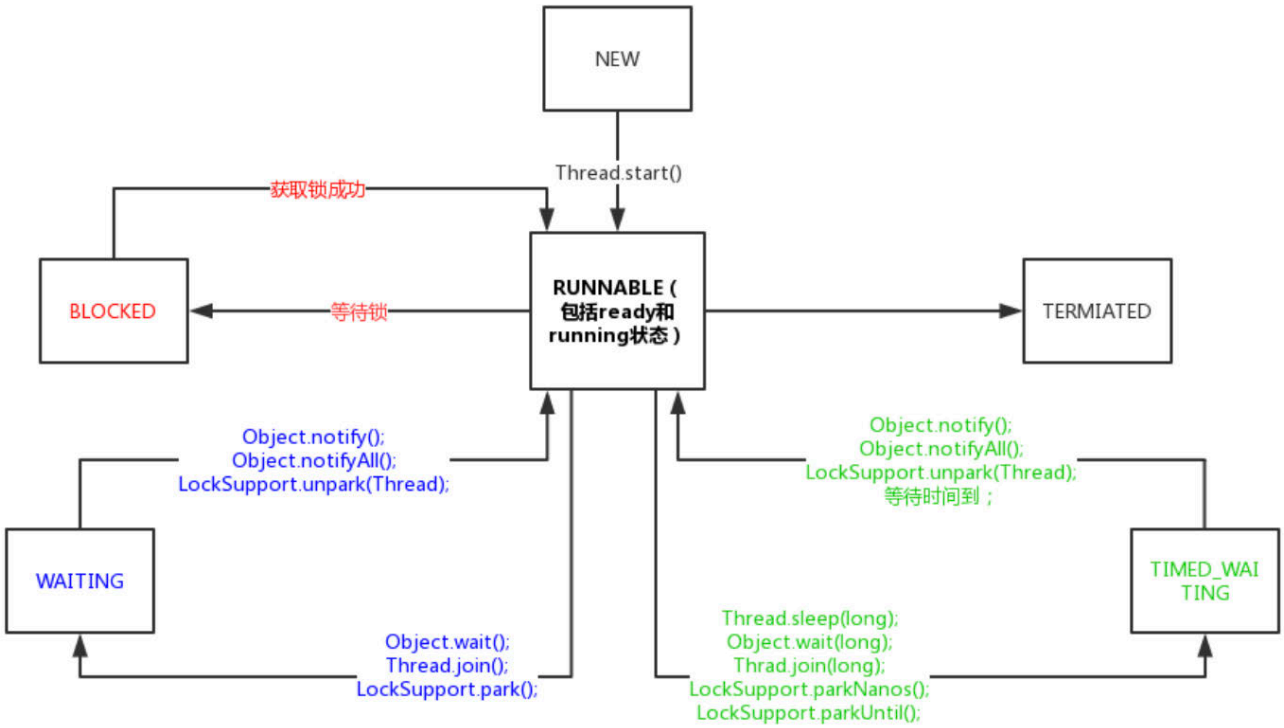
t1 等待 10 分钟后，就自动唤醒，拥有了去争夺锁的资格。

TERMINATED

终止状态。此时线程已执行完毕。

线程状态的转换

根据上面关于线程状态的介绍我们可以得到下面的**线程状态转换图**：



BLOCKED 与 RUNNABLE 状态的转换

我们在上面说过：处于 BLOCKED 状态的线程在等待锁的释放。假如这里有两个线程 a 和 b，a 线程提前获得了锁并暂未释放锁，此时 b 就处于 BLOCKED 状态。我们来看一个例子：

```
@Test
public void blockedTest() {
    Thread a = new Thread(new Runnable() {
        @Override
        public void run() {
            testMethod();
        }
    }, "a");

    Thread b = new Thread(new Runnable() {
        @Override
        public void run() {
            testMethod();
        }
    }, "b");

    a.start();
    b.start();

    System.out.println(a.getName() + ":" + a.getState()); // 输出?
    System.out.println(b.getName() + ":" + b.getState()); // 输出?
}
```

```
// 同步方法争夺锁
private synchronized void testMethod() {
    try {
        Thread.sleep(2000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

初看之下，大家可能会觉得线程 a 会先调用同步方法，同步方法内又调用了`Thread.sleep()`方法，必然会输出 TIMED_WAITING，而线程 b 因为等待线程 a 释放锁所以必然会输出 BLOCKED。

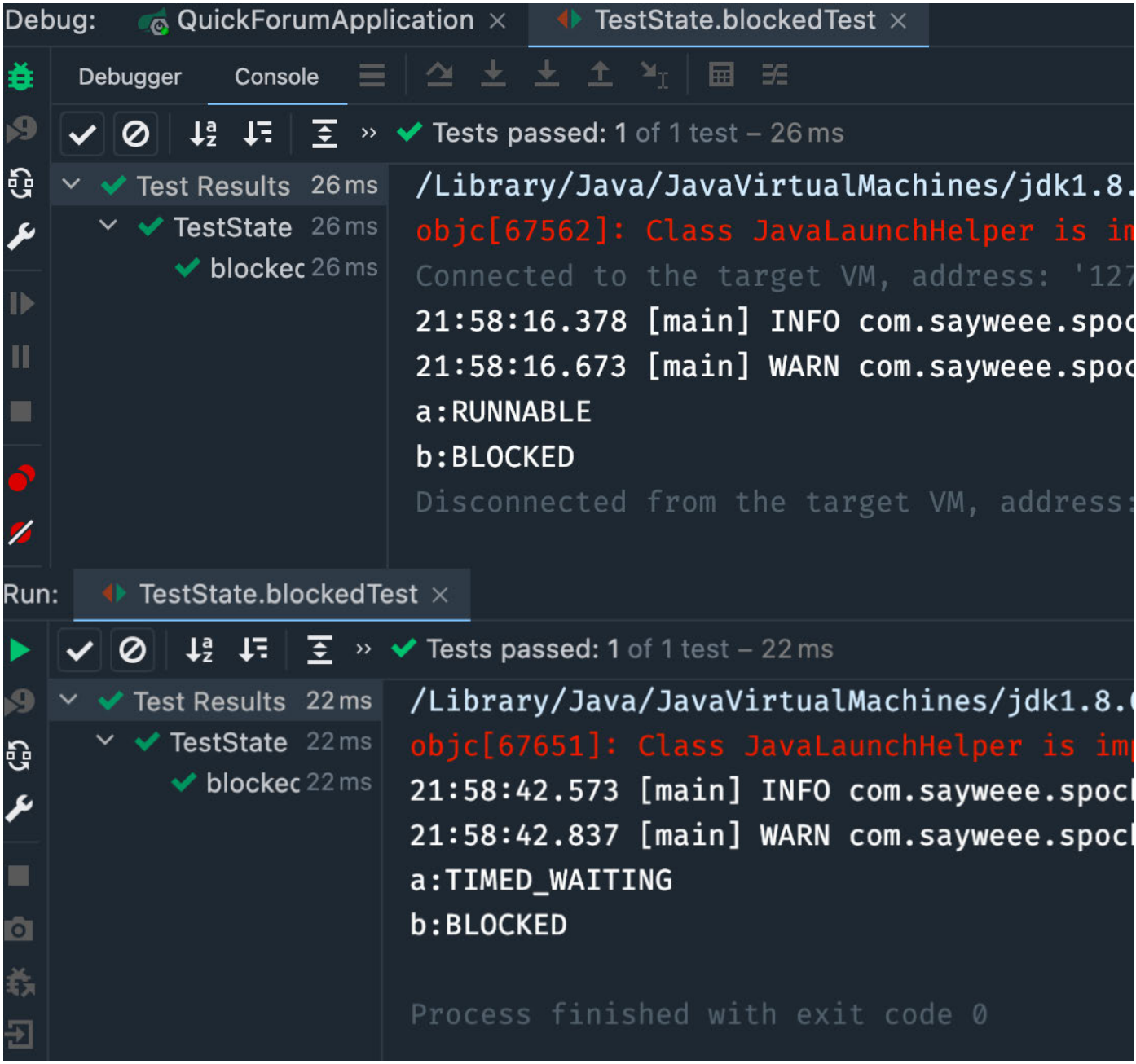
其实不然，有两点需要值得大家注意：

- 一是**在测试方法`blockedTest()`内还有一个 main 线程**
- 二是**启动线程后执行 run 方法还是需要消耗一定时间的**。

测试方法的 main 线程只保证了 a，b 两个线程调用 start 方法（转化为 RUNNABLE 状态），如果 CPU 执行效率高一点，还没等两个线程真正开始争夺锁，就已经打印此时两个线程的状态（RUNNABLE）了。

当然，如果 CPU 执行效率低一点，其中某个线程也是可能打印出 BLOCKED 状态的（此时两个线程已经开始争夺锁了）。

下面是我执行了几次的结果对比：



这时你可能又会问了，要是我想要打印出 BLOCKED 状态我该怎么处理呢?

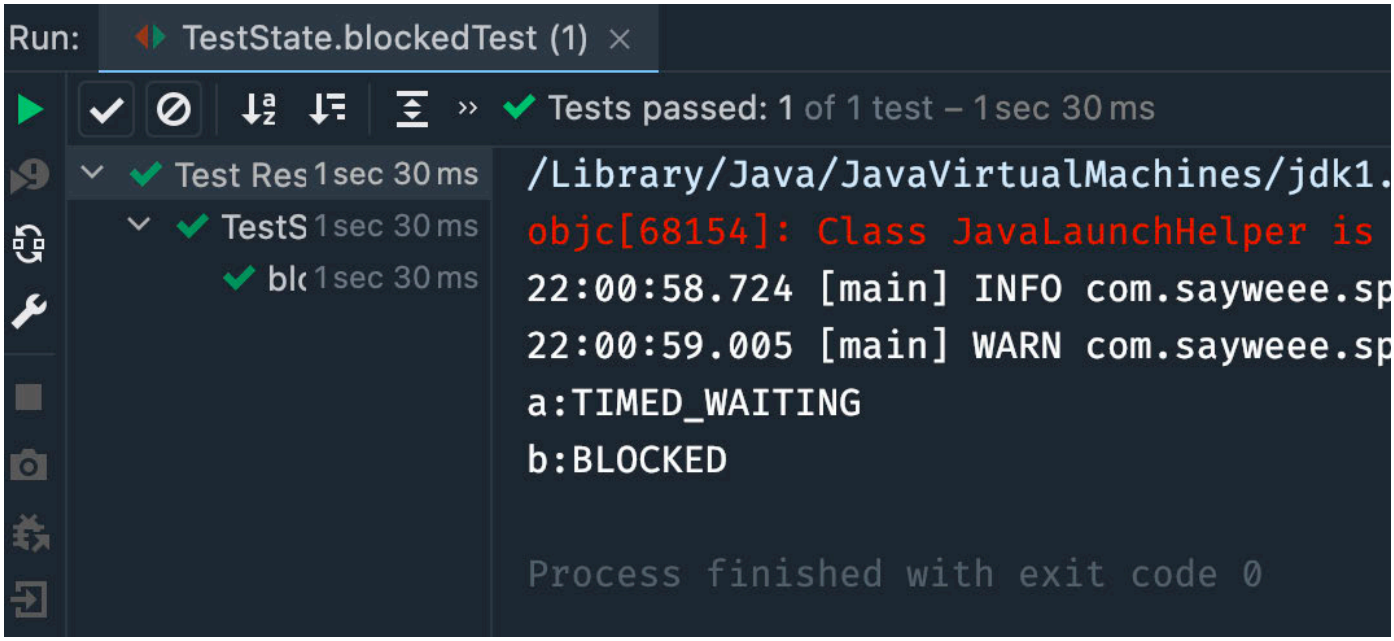
BLOCKED 状态的产生需要两个线程争夺锁才行。那我们处理下测试方法里的 main 线程就可以了，让它“休息一会儿”，调用一下`Thread.sleep()`方法。

这里需要注意的是 main 线程休息的时间，要保证在线程争夺锁的时间内，不要等到前一个线程锁都释放了你再去争夺锁，此时还是得不到 BLOCKED 状态的。

我们把上面的测试方法 blockedTest 改动一下：

```
public void blockedTest() throws InterruptedException {
    ......
    a.start();
    Thread.sleep(1000L); // 需要注意这里main线程休眠了1000毫秒，而testMethod()里休眠了2000毫秒
    b.start();
    System.out.println(a.getName() + ":" + a.getState()); // 输出?
    System.out.println(b.getName() + ":" + b.getState()); // 输出?
}
```

运行结果如下所示：



在这个例子中两个线程的状态转换如下

- a 的状态转换过程：RUNNABLE（`a.start()`） -> TIMED_WATING（`Thread.sleep()`）->RUNABLE（`sleep()`时间到）->*BLOCKED(未抢到锁)* -> TERMINATED
- b 的状态转换过程：RUNNABLE（`b.start()`）-> *BLOCKED(未抢到锁)* ->TERMINATED

斜体表示可能出现的状态， 大家可以在自己的电脑上多试几次看看输出。同样，这里的输出也可能有多钟结果。

WAITING 状态与 RUNNABLE 状态的转换

根据转换图我们知道有 3 个方法可以使线程从 RUNNABLE 状态转为 WAITING 状态。我们主要介绍下**Object.wait()**和**Thread.join()**。

Object.wait()

调用`wait()`方法前线程必须持有对象的锁。

线程调用`wait()`方法时，会释放当前的锁，直到有其他线程调用`notify()`/`notifyAll()`方法唤醒等待锁的线程。

需要注意的是，其他线程调用`notify()`方法只会唤醒单个等待锁的线程，如有有多个线程都在等待这个锁的话不一定会唤醒到之前调用`wait()`方法的线程。

同样，调用`notifyAll()`方法唤醒所有等待锁的线程之后，也不一定会马上把时间片分给刚才放弃锁的那个线程，具体要看系统的调度。

Thread.join()

调用`join()`方法，会一直等待这个线程执行完毕（转换为 TERMINATED 状态）。

我们再把上面的例子线程启动那里改变一下：

```
public void blockedTest() {
    ......
    a.start();
    a.join();
    b.start();
    System.out.println(a.getName() + ":" + a.getState()); // 输出 TERMINATED
    System.out.println(b.getName() + ":" + b.getState());
}
```

要是没有调用 join 方法，main 线程不管 a 线程是否执行完毕都会继续往下走。

a 线程启动之后马上调用了 join 方法，这里 main 线程就会等到 a 线程执行完毕，所以这里 a 线程打印的状态固定是**TERMINATED**。

至于 b 线程的状态，有可能打印 RUNNABLE（尚未进入同步方法），也有可能打印 TIMED_WAITING（进入了同步方法）。

TIMED_WAITING 与 RUNNABLE 状态转换

TIMED_WAITING 与 WAITING 状态类似，只是 TIMED_WAITING 状态等待的时间是指定的。

Thread.sleep(long)

使当前线程睡眠指定时间。需要注意这里的“睡眠”只是暂时使线程停止执行，并不会释放锁。时间到后，线程会重新进入 RUNNABLE 状态。

Object.wait(long)

`wait(long)`方法使线程进入 TIMED_WAITING 状态。这里的`wait(long)`方法与无参方法 wait()相同的地方是，都可以通过其他线程调用`notify()`或`notifyAll()`方法来唤醒。

不同的地方是，有参方法`wait(long)`就算其他线程不来唤醒它，经过指定时间 long 之后它会自动唤醒，拥有去争夺锁的资格。

Thread.join(long)

`join(long)`使当前线程执行指定时间，并且使线程进入 TIMED_WAITING 状态。

我们再来改一改刚才的示例:

```
public void blockedTest() {
    ......
    a.start();
    a.join(1000L);
    b.start();
    System.out.println(a.getName() + ":" + a.getState()); // 输出 TIEMD_WAITING
    System.out.println(b.getName() + ":" + b.getState());
}
```

这里调用`a.join(1000L)`，因为是指定了具体 a 线程执行的时间的，并且执行时间是小于 a 线程 sleep 的时间，所以 a 线程状态输出 TIMED_WAITING。

b 线程状态仍然不固定（RUNNABLE 或 BLOCKED）。

线程中断

在某些情况下，我们在线程启动后发现并不需要它继续执行下去时，需要中断线程。目前在 Java 里还没有安全方法来直接停止线程，但是 Java 提供了线程中断机制来处理需要中断线程的情况。

线程中断机制是一种协作机制。需要注意，通过中断操作并不能直接终止一个线程，而是通知需要被中断的线程自行处理。

简单介绍下 Thread 类里提供的关于线程中断的几个方法：

- `Thread.interrupt()`：中断线程。这里的中断线程并不会立即停止线程，而是设置线程的中断状态为 true（默认是 flase）；
- `Thread.currentThread().isInterrupted()`：测试当前线程是否被中断。线程的中断状态会受这个方法的影响，调用一次可以使线程中断状态变为 true，调用两次会使这个线程的中断状态重新转为 false；
- `Thread.isInterrupted()`：测试当前线程是否被中断。与上面方法不同的是调用这个方法并不会影响线程的中断状态。

在线程中断机制里，当其他线程通知需要被中断的线程后，线程中断的状态被设置为 true，但是具体被要求中断的线程要怎么处理，完全由被中断线程自己决定，可以在合适的时机中断请求，也可以完全不处理继续执行下去。

小结

本文详细解析了 Java 线程的 6 种状态 — 新建、运行、阻塞、等待、定时等待和终止，以及这些状态之间的切换过程。

> 编辑：沉默王二，原文内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。

第四节：线程组和线程优先级

Java 提供了 ThreadGroup 类来创建一组相关的线程，使线程组管理更方便。每个 Java 线程都有一个优先级，这个优先级会影响到操作系统为这个线程分配处理器时间的顺序。

这篇内容将分别来介绍一下线程组和线程优先级。

线程组(ThreadGroup)

Java 用 ThreadGroup 来表示线程组，我们可以通过线程组对线程进行批量控制。

ThreadGroup 和 Thread 的关系就如同他们的字面意思一样简单粗暴，每个 Thread 必然存在于一个 ThreadGroup 中，Thread 不能独立于 ThreadGroup 存在。执行`main()`方法的线程名字是 main，如果在 new Thread 时没有显式指定，那么默认将父线程（当前执行 new Thread 的线程）线程组设置为自己的线程组。

示例代码：

```
Thread testThread = new Thread(() -> {
    System.out.println("testThread当前线程组名字：" +
            Thread.currentThread().getThreadGroup().getName());
    System.out.println("testThread线程名字：" +
            Thread.currentThread().getName());
});

testThread.start();
System.out.println("执行main所在线程的线程组名字：  " +
Thread.currentThread().getThreadGroup().getName());
System.out.println("执行main方法线程名字：" + Thread.currentThread().getName());
```

输出结果：

```
执行main所在线程的线程组名字：  main
testThread当前线程组名字：main
testThread线程名字：Thread-0
执行main方法线程名字：main
```

ThreadGroup 是一个标准的**向下引用**的树状结构，这样设计可以**防止"上级"线程被"下级"线程引用而无法有效地被 GC 回收**。

线程组的常用方法及数据结构

获取当前线程的线程组名字

```
Thread.currentThread().getThreadGroup().getName()
```

复制线程组

```
// 获取当前的线程组
ThreadGroup threadGroup = Thread.currentThread().getThreadGroup();
// 复制一个线程组到一个线程数组（获取Thread信息）
Thread[] threads = new Thread[threadGroup.activeCount()];
threadGroup.enumerate(threads);
```

线程组统一异常处理

```
// 创建一个线程组，并重新定义异常
ThreadGroup group = new ThreadGroup("testGroup") {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println(t.getName() + ": " + e.getMessage());
    }
};

// 测试异常
```

```
Thread thread = new Thread(group, () -> {
    // 抛出 unchecked 异常
    throw new RuntimeException("测试异常");
});

// 启动线程
thread.start();
```

线程组的数据结构

线程组还可以包含其他的线程组，不仅仅是线程。首先看看 `ThreadGroup` 源码中的成员变量。

```
public class ThreadGroup implements Thread.UncaughtExceptionHandler {
    private final ThreadGroup parent; // 父亲ThreadGroup
    String name; // ThreadGroup 的名称
    int maxPriority; // 最大优先级
    boolean destroyed; // 是否被销毁
    boolean daemon; // 是否守护线程
    boolean vmAllowSuspension; // 是否可以中断

    int nUnstartedThreads = 0; // 还未启动的线程
    int nthreads; // ThreadGroup中线程数目
    Thread threads[]; // ThreadGroup中的线程

    int ngroups; // 线程组数目
    ThreadGroup groups[]; // 线程组数组
}
```

然后看看构造方法：

```
// 私有构造方法
private ThreadGroup() {
    this.name = "system";
    this.maxPriority = Thread.MAX_PRIORITY;
    this.parent = null;
}

// 默认是以当前ThreadGroup作为parent ThreadGroup，新线程组的父线程组是目前正在运行线程的线程组。
public ThreadGroup(String name) {
    this(Thread.currentThread().getThreadGroup(), name);
}

// 构造方法
public ThreadGroup(ThreadGroup parent, String name) {
    this(checkParentAccess(parent), parent, name);
}

// 私有构造方法，主要的构造函数
```

```
private ThreadGroup(Void unused, ThreadGroup parent, String name) {
    this.name = name;
    this.maxPriority = parent.maxPriority;
    this.daemon = parent.daemon;
    this.vmAllowSuspension = parent.vmAllowSuspension;
    this.parent = parent;
    parent.add(this);
}
```

第三个构造方法里调用了`checkParentAccess`方法，来看看这个方法的源码：

```
// 检查parent ThreadGroup
private static Void checkParentAccess(ThreadGroup parent) {
    parent.checkAccess();
    return null;
}

// 判断当前运行的线程是否具有修改线程组的权限
public final void checkAccess() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkAccess(this);
    }
}
```

这里涉及到`SecurityManager`这个类，它是 Java 的安全管理器，它允许应用程序在执行一个可能不安全或敏感的操作前确定该操作是什么，以及是否允许在执行该操作的上下文中执行它。

比如引入了第三方类库，但是并不能保证它的安全性。

其实 Thread 类也有一个 checkAccess 方法，不过是用来当前运行的线程是否有权限修改被调用的这个线程实例。（Determines if the currently running thread has permission to modify this thread.）

总结一下，线程组是一个树状的结构，每个线程组下面可以有多个线程或者线程组。线程组可以起到统一控制线程的优先级和检查线程权限的作用。

线程的优先级

线程优先级可以指定，范围是 1~10。但并不是所有的操作系统都支持 10 级优先级的划分（比如有些操作系统只支持 3 级划分：低、中、高），Java 只是给操作系统一个优先级的**参考值**，线程最终**在操作系统中的优先级**还是由操作系统决定。

Java 默认的线程优先级为 5，线程的执行顺序由调度程序来决定，线程的优先级会在线程被调用之前设定。

通常情况下，高优先级的线程将会比低优先级的线程有**更高的概率**得到执行。`Thread`类的`setPriority()`方法可以用来设定线程的优先级。

```
Thread a = new Thread();
System.out.println("我是默认线程优先级："+a.getPriority());
Thread b = new Thread();
b.setPriority(10);
System.out.println("我是设置过的线程优先级："+b.getPriority());
```

输出结果：

```
我是默认线程优先级：5
我是设置过的线程优先级：10
```

既然有 10 个级别来设定线程的优先级，那是不是可以在业务实现的时候，采用这种方法来指定线程执行的先后顺序呢？

对于这个问题，答案是：No!

Java 中的优先级不是特别的可靠，**Java 程序中对线程所设置的优先级只是给操作系统一个建议，操作系统不一定会采纳。而真正的调用顺序，是由操作系统的线程调度算法来决定的。**

我们通过代码来验证一下：

```
static class MyThread extends Thread {
    @Override
    public void run() {
        // 输出当前线程的名字和优先级
        System.out.println("MyThread当前线程：" + Thread.currentThread().getName()
                + ",优先级：" + Thread.currentThread().getPriority());
    }
}

public static void main(String[] args) {
    // 创建 10 个线程，从 1-10 运行，优先级从 1-10
    for (int i = 1; i <= 10; i++) {
        Thread thread = new MyThread();
        thread.setName("线程" + i);
        thread.setPriority(i);
        thread.start();
    }
}
```

运行该程序，有时候可以按照优先级执行，有时却不行，这是某次输出：

```
MyThread当前线程：线程2,优先级：2
MyThread当前线程：线程4,优先级：4
MyThread当前线程：线程3,优先级：3
MyThread当前线程：线程5,优先级：5
MyThread当前线程：线程1,优先级：1
MyThread当前线程：线程6,优先级：6
MyThread当前线程：线程7,优先级：7
MyThread当前线程：线程8,优先级：8
MyThread当前线程：线程9,优先级：9
MyThread当前线程：线程10,优先级：10
```

Java 提供了一个**线程调度器**来监视和控制处于**RUNNABLE 状态**的线程。

- 线程的调度策略采用**抢占式**的方式，优先级高的线程会比优先级低的线程有更大的几率优先执行。
- 在优先级相同的情况下，会按照“先到先得”的原则执行。
- 每个 Java 程序都有一个默认的主线程，就是通过 JVM 启动的第一个线程——main 线程。

还有一种特殊的线程，叫做**守护线程（Daemon）**，守护线程默认的优先级比较低。

- 如果某线程是守护线程，那如果所有的非守护线程都结束了，这个守护线程也会自动结束。
- 当所有的非守护线程结束时，守护线程会自动关闭，这就免去了还要继续关闭子线程的麻烦。
- 线程默认是非守护线程，可以通过 Thread 类的 setDaemon 方法来设置为守护线程。

线程组和线程优先级之间的关系

之前我们谈到一个线程必然存在于一个线程组中，那么当线程和线程组的优先级不一致的时候会怎样呢？我们来验证一下：

```
 // 创建一个线程组
ThreadGroup group = new ThreadGroup("testGroup");
// 将线程组的优先级指定为 7
group.setMaxPriority(7);
// 创建一个线程，将该线程加入到 group 中
Thread thread = new Thread(group, "test-thread");
// 企图将线程的优先级设定为 10
thread.setPriority(10);
// 输出线程组的优先级和线程的优先级
System.out.println("线程组的优先级是：" + group.getMaxPriority());
System.out.println("线程的优先级是：" + thread.getPriority());
```

输出：

```
线程组的优先级是：7
线程的优先级是：7
```

所以，如果某个线程的优先级大于线程所在**线程组的最大优先级**，那么该线程的优先级将会失效，取而代之的是线程组的最大优先级。

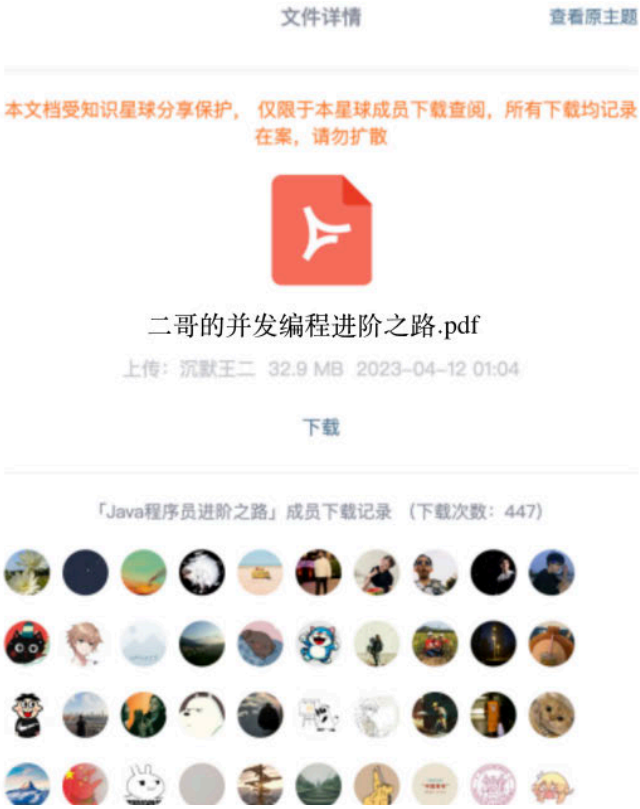
小结

Java 提供了 ThreadGroup 类来创建一组相关的线程，使线程组管理更方便；每个 Java 线程都有一个优先级，这个优先级会影响到操作系统为这个线程分配处理器时间的顺序。

> 编辑：沉默王二，原文内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。



第五节：进程与线程的区别

经过前面几章的学习，我们对线程的基本概念和使用方式已经有了比较充分的了解，那么接下来我们就来分析一下，线程是如何从进程进化而来的，它们之间又有哪些区别，搞清楚两者之间的差别对接下来的学习也是至关重要的，甚至有些公司的面试官也喜欢问这个。

进程

最初的计算机只能接受一些特定的指令，用户每输入一个指令，计算机就做出一个操作。当用户在思考或者输入时，计算机就在等待。这样效率非常低下，在很多时候，计算机都处在等待状态。

批处理操作系统

后来有了**批处理操作系统**，把一系列需要操作的指令写下来，形成一个清单，一次性交给计算机。用户将多个需要执行的程序写在磁带上，然后交由计算机去读取并逐个执行这些程序，并将输出结果写在另一个磁带上。

批处理操作系统在一定程度上提高了计算机的效率，但是由于**批处理操作系统的指令运行方式仍然是串行的，内存中始终只有一个程序在运行**，后面的程序需要等待前面的程序执行完成后才能开始执行，而前面的程序有时会由于 I/O 操作、网络等原因阻塞，所以**批处理操作效率也不高**。

进程的提出

人们对于计算机的性能要求越来越高，现有的批处理操作系统并不能满足人们的需求，而批处理操作系统的瓶颈在于内存中只存在一个程序，那么内存中能不能存在多个程序呢？这是人们亟待解决的问题。

于是，科学家们提出了进程的概念。

进程就是**应用程序在内存中分配的空间，也就是正在运行的程序**，各个进程之间互不干扰。同时进程保存着程序每一个时刻运行的状态。

> 程序：用某种编程语言(Java、Python 等)编写，能够完成一定任务或者功能的代码集合，是指令和数据的有序集合，是**一段静态代码**。

此时，CPU 采用时间片轮转的方式运行进程：CPU 为每个进程分配一个时间段，称作它的时间片。如果在时间片结束时进程还在运行，则暂停这个进程的运行，并且 CPU 分配给另一个进程（这个过程叫做上下文切换）。如果进程在时间片结束前阻塞或结束，则 CPU 立即进行切换，不用等待时间片用完。

> 当进程暂停时，它会保存当前进程的状态（进程标识，进程使用的资源等），在下一次切换回来时根据之前保存的状态进行恢复，接着继续执行。

使用进程+CPU 时间片轮转方式的操作系统，在宏观上看起来同一时间段执行多个任务，换句话说，**进程让操作系统的并发成为了可能**。虽然并发从宏观上看有多个任务在执行，但在事实上，对于**单核 CPU**来说，任意具体时刻都只有一个任务在占用 CPU 资源。

对操作系统的要求进一步提高

虽然进程的出现，使得操作系统的性能大大提升，但是随着时间的推移，人们并不满足一个进程在一段时间只能做一件事情，如果一个进程有多个子任务时，只能逐个得执行这些子任务，很影响效率。

> 比如杀毒软件在检测用户电脑时，如果在某一项检测中卡住了，那么后面的检测项也会受到影响。或者说当你使用杀毒软件中的扫描病毒功能时，在扫描病毒结束之前，无法使用杀毒软件中清理垃圾的功能，这显然无法满足人们的要求。

线程

那么能不能让这些子任务同时执行呢？于是人们又提出了线程的概念，**让一个线程执行一个子任务，这样一个进程就包含了多个线程，每个线程负责一个单独的子任务。**

> - 使用线程之后，事情就变得简单多了。当用户使用扫描病毒功能时，就让扫描病毒这个线程去执行。同时，如果用户又使用清理垃圾功能，那么可以先暂停扫描病毒线程，先响应用户的清理垃圾的操作，让清理垃圾这个线程去执行。响应完后再切换回来，接着执行扫描病毒线程。
> - 注意：操作系统是如何分配时间片给每一个线程的，涉及到线程的调度策略，有兴趣的同学可以看一下《操作系统》相关的内容，这里就不再展开了，涉及的内容比较多。

总之，进程和线程的提出极大的提高了操作系统的性能。**进程让操作系统的并发性成为了可能，而线程让进程的内部并发成为了可能。**

既然多进程的方式可以实现并发，为什么还要使用多线程呢?

多进程方式确实可以实现并发，但使用多线程，有以下几个好处：

- 进程间的通信比较复杂，而线程间的通信比较简单，通常情况下，我们需要使用共享资源，这些资源在线程间的通信很容易。
- 进程是重量级的，而线程是轻量级的，多线程方式的系统开销更小。

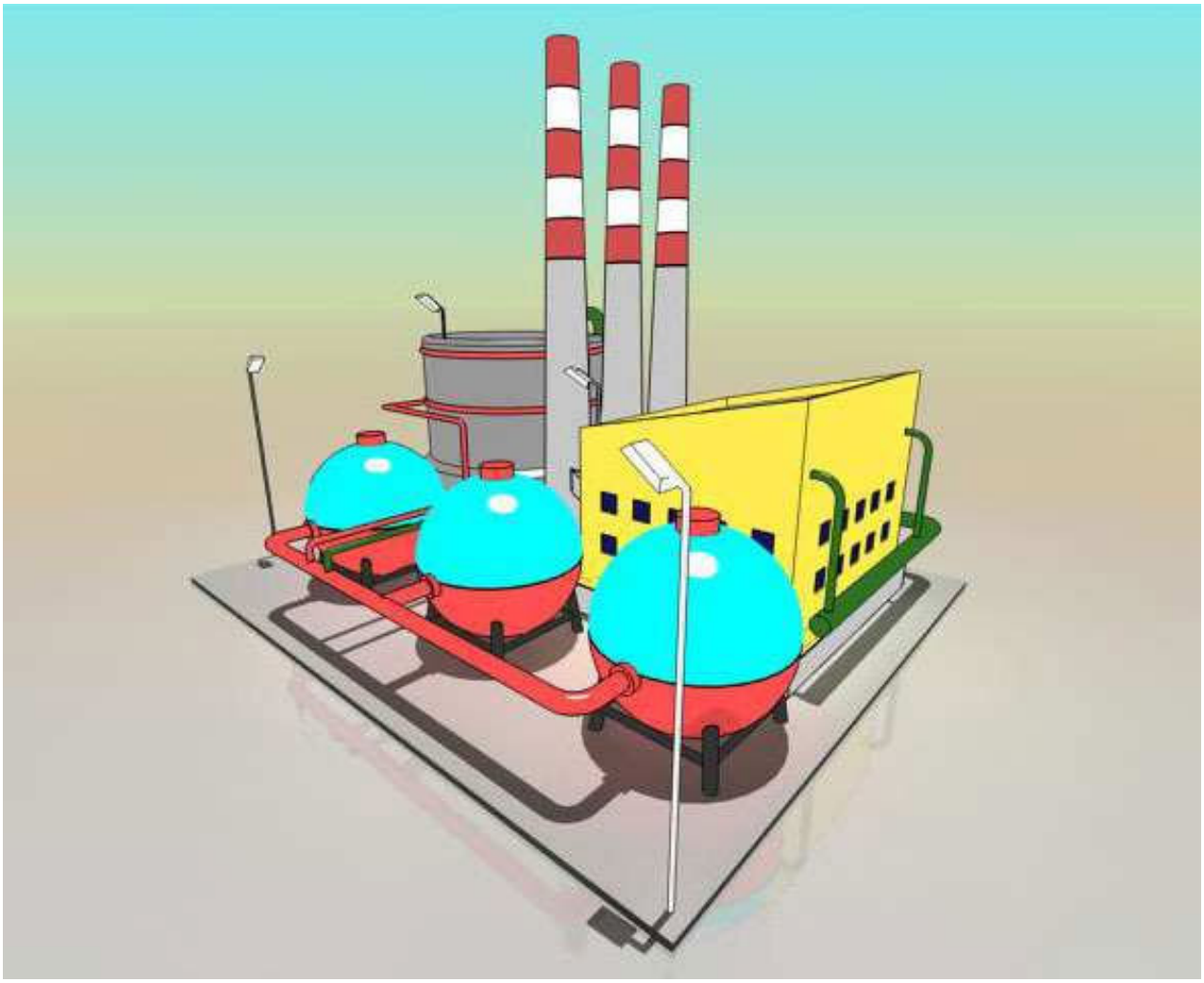
进程和线程的区别

进程是一个独立的运行环境，而线程是在进程中执行的一个任务。他们两个本质的区别是**是否单独占有内存地址空间及其它系统资源（比如 I/O）**：

- 进程单独占有一定的内存地址空间，所以进程间存在内存隔离，数据是分开的，数据共享复杂但是同步简单，各个进程之间互不干扰；而线程共享所属进程占有的内存地址空间和资源，数据共享简单，但是同步复杂。
- 进程单独占有一定的内存地址空间，一个进程出现问题不会影响其他进程，不影响主程序的稳定性，可靠性高；一个线程崩溃可能影响整个程序的稳定性，可靠性较低。
- 进程单独占有一定的内存地址空间，进程的创建和销毁不仅需要保存寄存器和栈信息，还需要资源的分配回收以及页调度，开销较大；线程只需要保存寄存器和栈信息，开销较小。

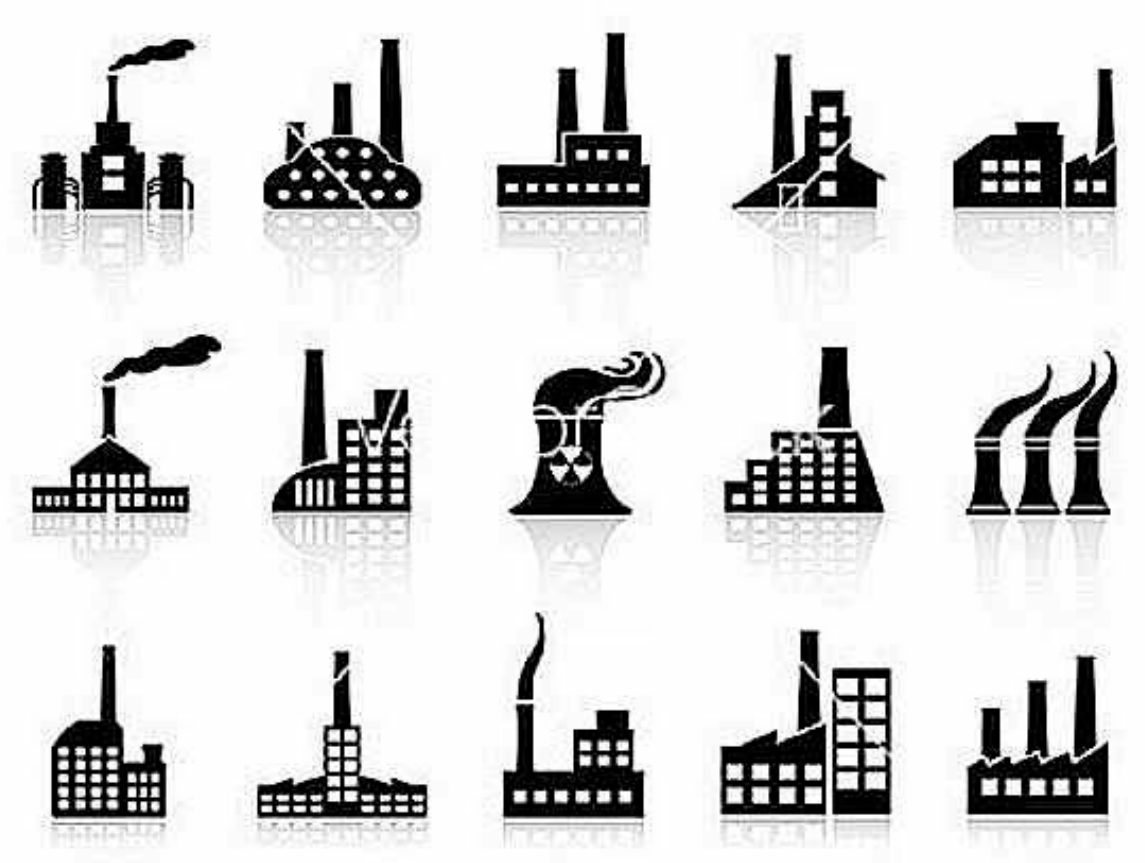
另外一个重要区别是，**进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位**，即 CPU 分配时间的单位 。

再用一些图来表达一下，感官会更清晰一些。

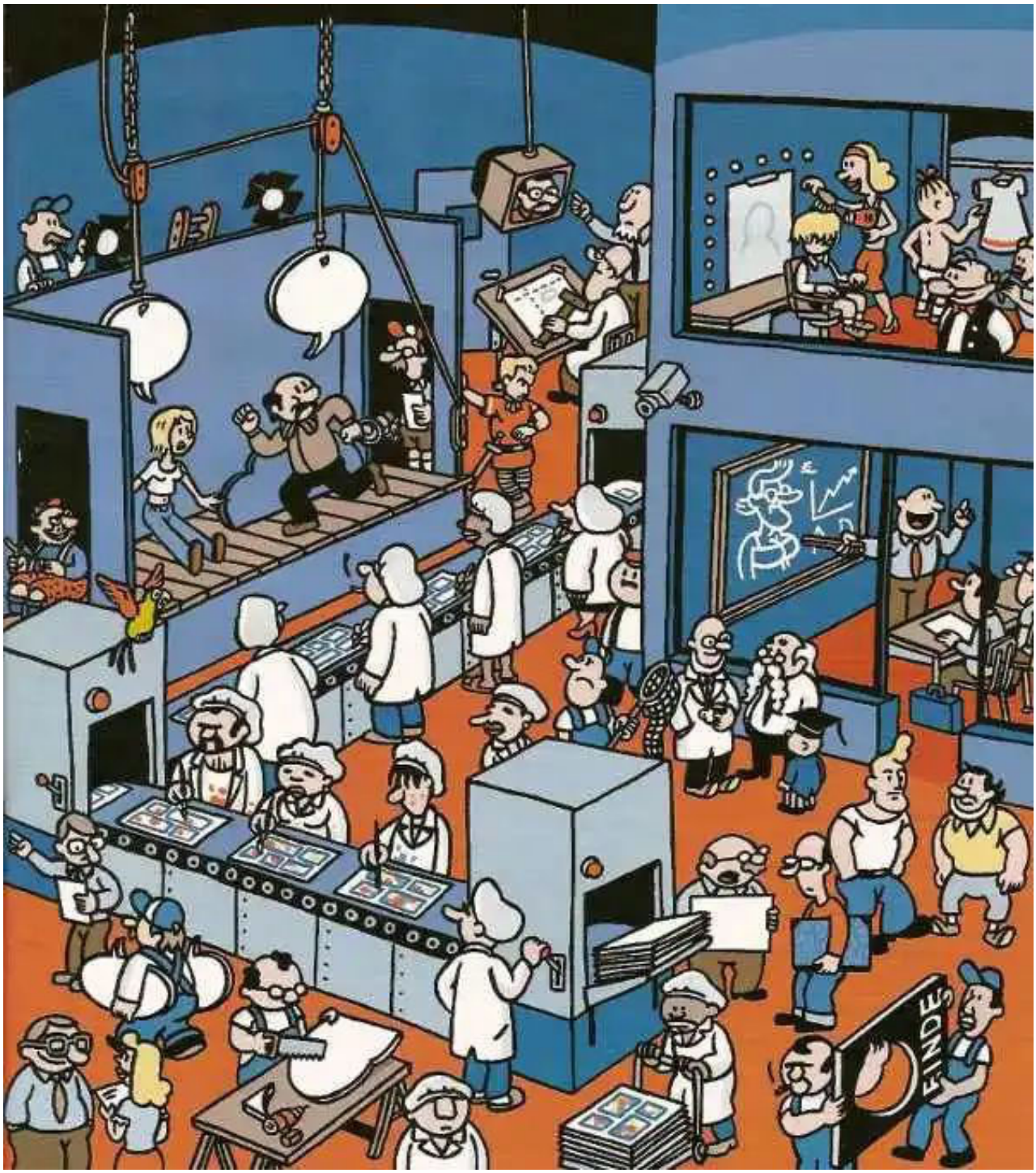


计算机的核心是 CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个 CPU 一次只能运行一个任务。

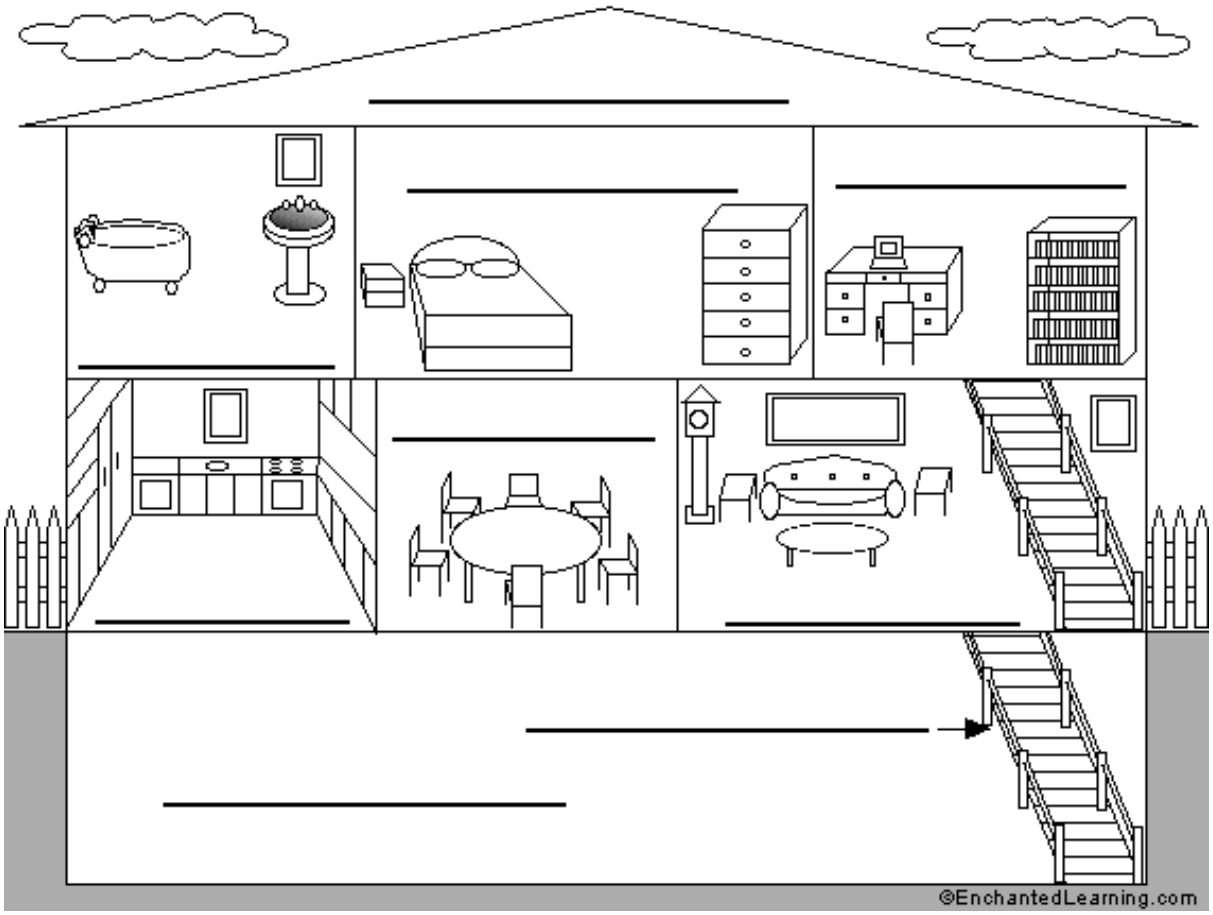


进程就好比工厂的车间，它代表 CPU 所能处理的单个任务。任一时刻，CPU 总是运行一个进程，其他进程处于非运行状态。



一个车间里，可以有很多工人。他们协同完成一个任务。

线程就好比车间里的工人。一个进程可以包括多个线程。



车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。

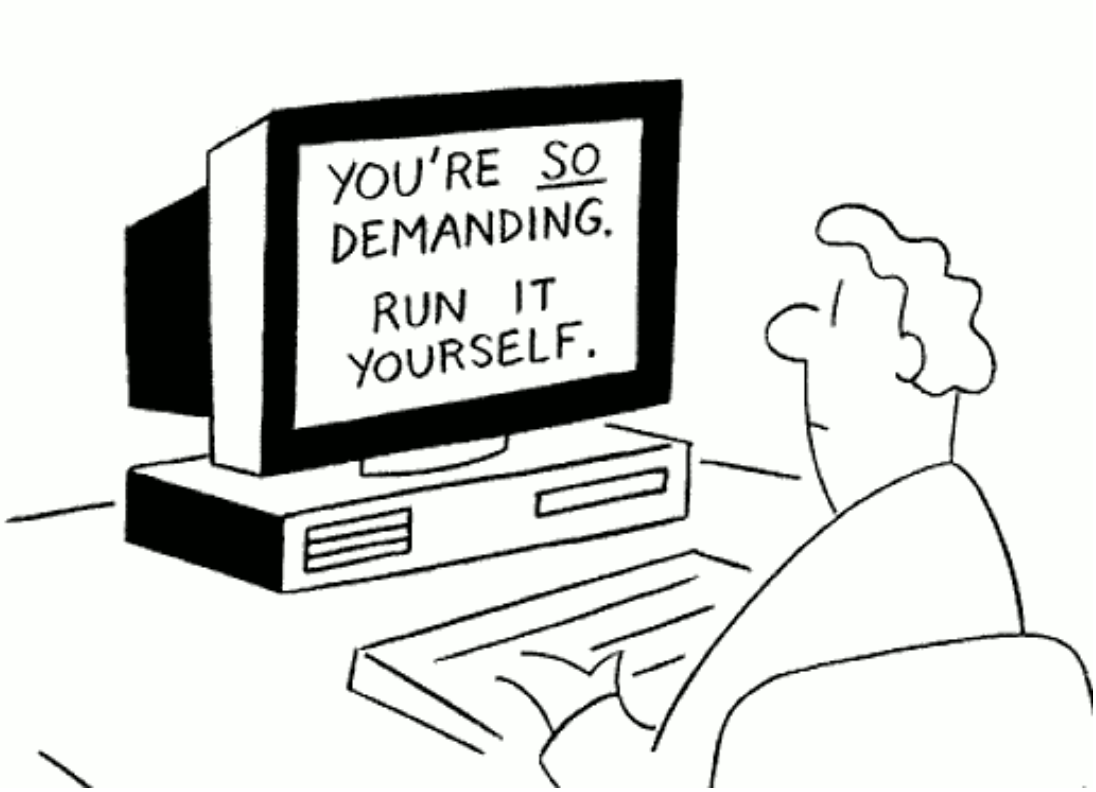
可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。

一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

还有些房间，可以同时容纳 n 个人，比如厨房。也就是说，如果人数大于 n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

这时的解决方法，就是在门口挂 n 把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex 是 semaphore 的一种特殊情况（n=1 时）。也就是说，完全可以用后者替代前者。但是，因为 mutex 较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。



操作系统的设计，因此可以归结为三点：

- 以多进程形式，允许多个任务同时运行；
- 以多线程形式，允许单个任务分成不同的部分运行；
- 提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

线程带来的好处

一直以来，硬件的发展极其迅速，其中有一个很著名的"摩尔定律"。

摩尔定律并不是一种自然法则或者是物理定律，它只是基于一些观测数据后，对未来的一种预测。按照所预测的速度，我们的计算能力会按照指数级别的速度增长，不久以后会拥有超强的计算能力。2004 年，Intel 宣布 4GHz 芯片的计划推迟到 2005 年，然后在 2004 年秋季，Intel 宣布彻底取消 4GHz 的计划，也就是说摩尔定律的有效性超过半个世纪后戛然而止。但是，聪明的硬件工程师并没有停止研发的脚步，他们为了进一步提升计算速度，不再追求单独的计算单元，而是将多个计算单元整合到了一起，于是就出现了多核 CPU。

短短十几年的时间，家用型 CPU，比如 Intel i7 就可以达到 4 核心甚至 8 核心。而专业服务器通常可以达到几个独立的 CPU，每一个 CPU 甚至拥有多达 8 个以上的内核。因此，摩尔定律似乎在 CPU 核心扩展上继续得以验证。因此，多核 CPU 的背景下，并发编程变得越来越受重视，因为通过**并发编程的形式可以将多核 CPU 的计算能力发挥到极致**。

顶级计算机科学家 Donald Ervin Knuth 如此评价这种情况：在我看来，这种现象（并发）或多或少是由于硬件设计者无计可施导致的，他们将摩尔定律的责任推给了开发者。

另外，有些特殊的业务场景下，先天就适合于并发编程。比如在图像处理领域，一张 1024X768 像素的图片，包含达到 78 万 6 千多个像素。要在短时间内将所有的像素遍历一边需要很长的时间，面对如此复杂的计算量就需要充分利用多核计算的能力。

又比如当我们在网上购物时，为了提升响应速度，需要拆分，减库存，生成订单等等这些操作，就可以利用多线程的技术来完成。**面对复杂业务模型，并行程序会比串行程序更适用于业务需求** 。正是因为这些优点，使得多线程技术得到了进一步的重视，Java 开发者也应该掌握并发编程，以便：

- 充分利用多核 CPU 的计算能力；
- 方便进行业务拆分，提升应用性能

怎么样，进程和线程的概念就彻底搞懂了吧？再遇到面试官问这个问题，就直接吊打他吧。

小结

总结来说，进程和线程都是操作系统用于并发执行的方式，但是它们在资源管理、独立性、开销以及影响范围等方面有所不同。

- 进程是操作系统分配资源的基本单位，线程是操作系统调度的基本单位。
- 进程拥有独立的内存空间，线程共享所属进程的内存空间。
- 进程的创建和销毁需要资源的分配和回收，开销较大；线程的创建和销毁只需要保存寄存器和栈信息，开销较小。
- 进程间的通信比较复杂，而线程间的通信比较简单。
- 进程间是相互独立的，一个进程崩溃不会影响其他进程；线程间是相互依赖的，一个线程崩溃可能影响整个程序的稳定性。

> 编辑：沉默王二，部分内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。还有一部分图片来源于阮一峰的博客，地址戳这里。

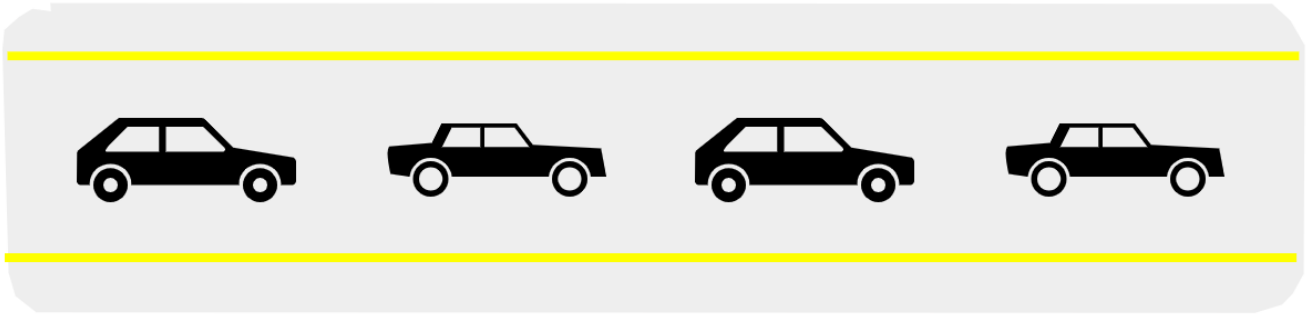
第六节：多线程带来了哪些问题?

前面我们了解到，多线程技术有很多好处，比如说多线程可以充分利用多核 CPU 的计算能力，那多线程难道就没有一点缺点吗?

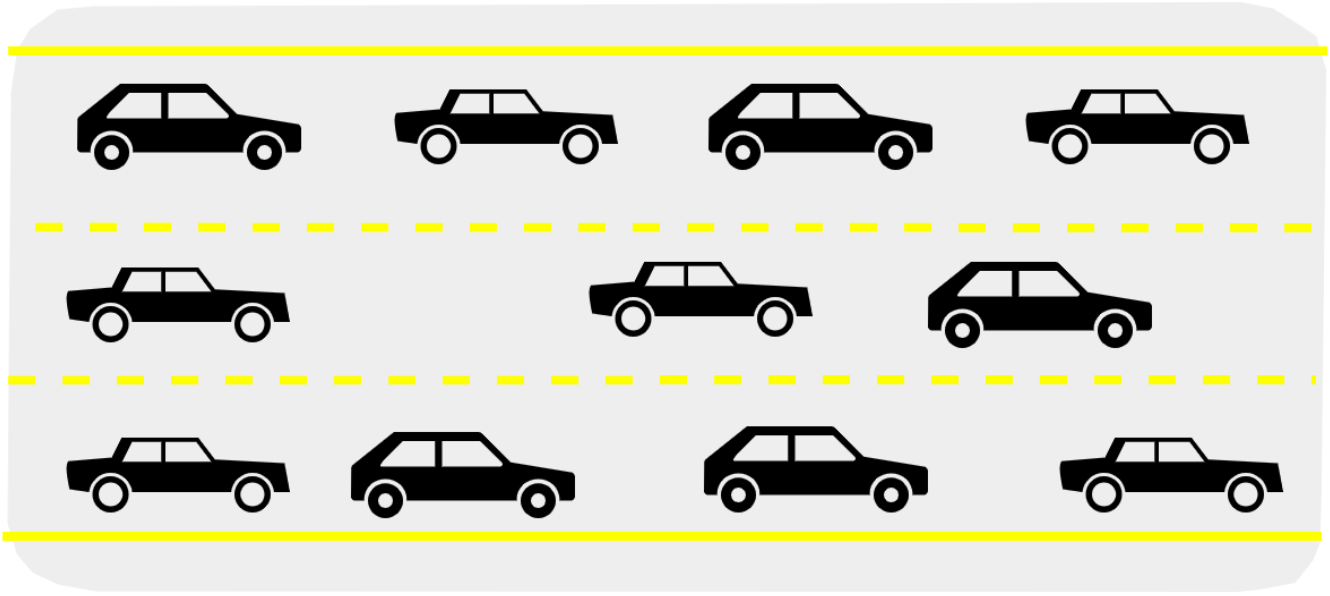
有。

多线程很难掌握，稍不注意，就容易使程序崩溃。我们以在路上开车为例：

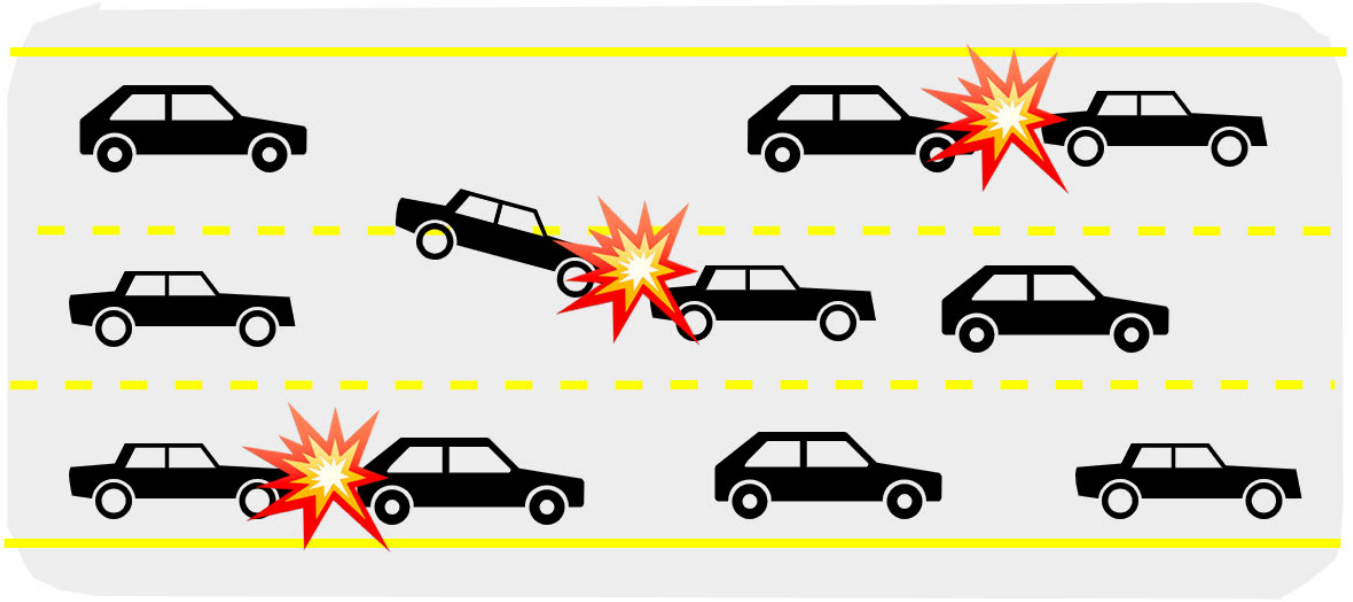
在一个单向行驶的道路上，每辆汽车都遵守交通规则，这时候整体通行是正常的。『单向车道』意味着『一个线程』，『多辆车』意味着『多个 job 任务』。



如果需要提升车辆的同行效率，一般的做法就是扩展车道，对应程序来说就是『加线程池』，增加线程数。这样在同一时间内，通行的车辆数远远大于单车道。



然而车道一旦多起来，『加塞』的场景就会越来越多，出现碰撞后也会影响整条马路的通行效率。这么一对比下来『多车道』就比『单车道』慢多了。



防止汽车频繁变道加塞可以在车道间增加『护栏』，那在程序的世界里该怎么做呢？

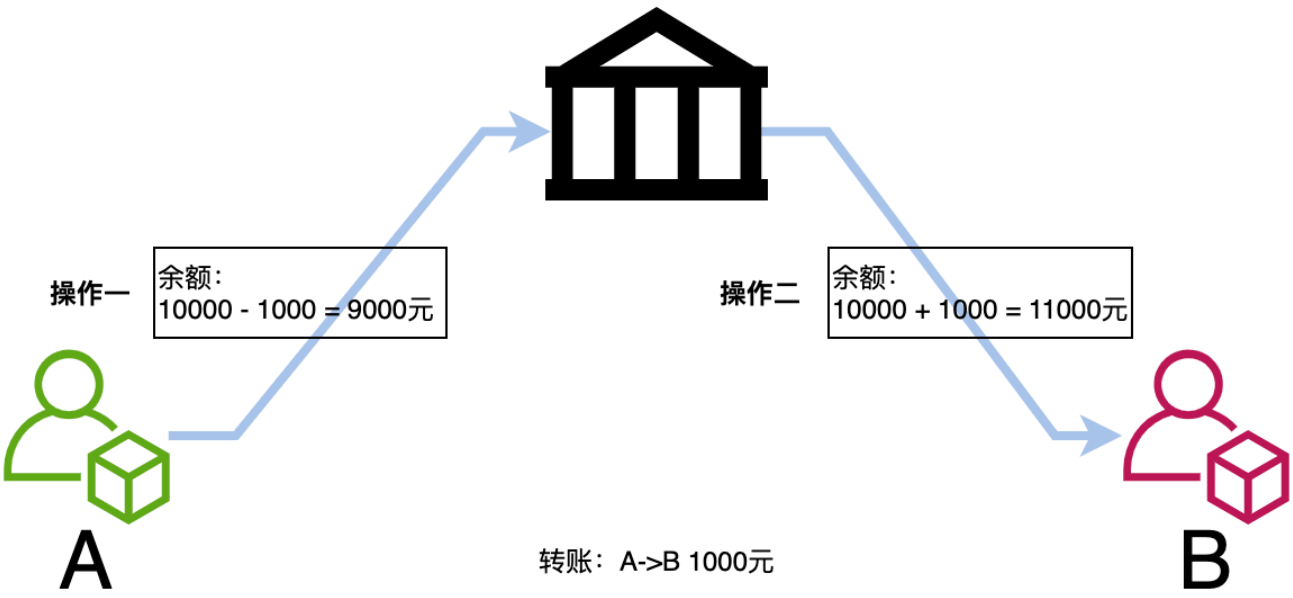
多线程遇到的问题归纳起来就三类：『线程安全问题』、『活跃性问题』、『性能问题』。

线程安全问题

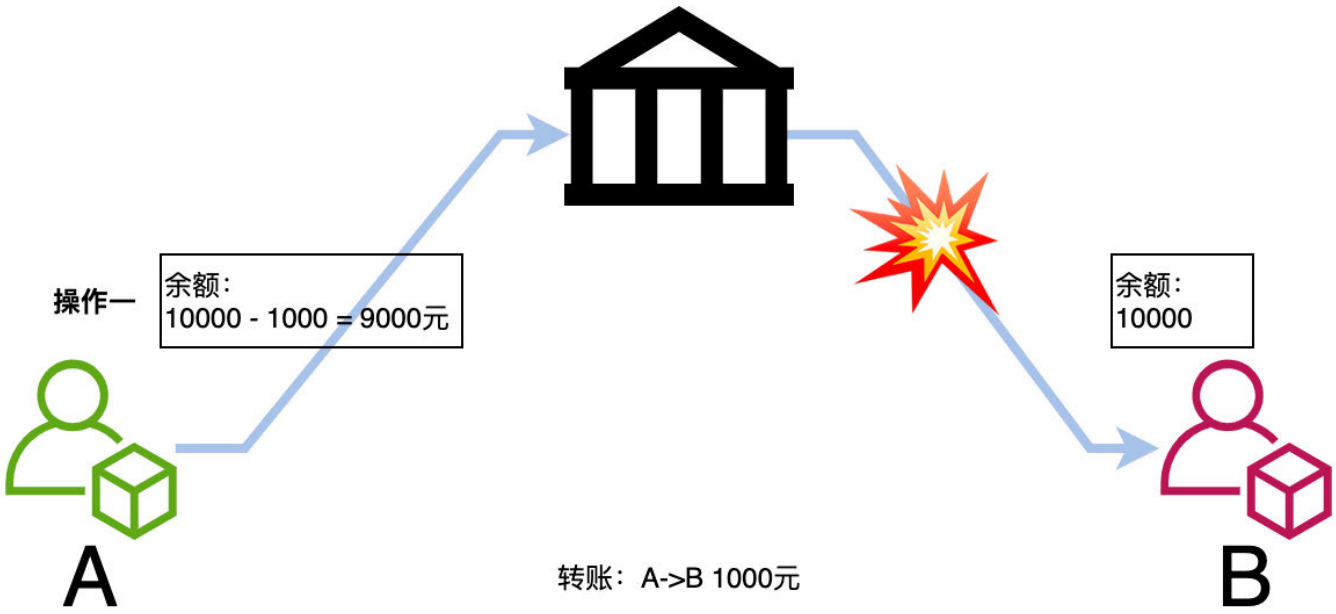
有时候我们会发现，明明在单线程环境中正常运行的代码，在多线程环境中就会出现意料之外的结果，这就是大家常说的『线程不安全』。那到底什么是线程不安全呢？

原子性

举一个银行转账的例子，比如从账户 A 向账户 B 转 1000 元，那么必然包括 2 个操作：从账户 A 减去 1000 元，往账户 B 加上 1000 元，两个操作都成功才意味着一次转账最终成功。



试想一下，如果这两个操作不具备原子性，从 A 的账户扣减了 1000 元之后，操作突然终止了，账户 B 没有增加 1000 元，那问题就大了。



银行转账有两个步骤，出现意外后导致转账失败，说明没有原子性。

> - 原子性：即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
> - 原子操作：即不会被线程调度机制打断的操作，没有上下文切换。

在并发编程中很多操作都不是原子操作，出个小题目：

```
int i = 0; // 操作1
i++;   // 操作2
int j = i; // 操作3
i = i + 1; // 操作4
```

上面这四个操作中哪些是原子操作，哪些不是呢?

有些小伙伴可能认为这些都是原子操作，其实只有操作 1 是原子操作。

- 操作 1：这是原子操作，因为它是一个单一的、不可分割的步骤。
- 操作 2：这不是原子操作。这实际上是一个 "read-modify-write" 操作，它包括了读取 i 的值，增加 i，然后写回 i。
- 操作 3：这是原子操作，因为它是一个单一的、不可分割的步骤。
- 操作 4：这不是原子操作。和 i++ 一样，这也是一个 "read-modify-write" 操作。

在单线程环境下上述四个操作都不会出现问题，但是在多线程环境下，如果不加锁的话，可能会得到意料之外的值。我们来测试一下，看看输出结果。

```
public class YuanziDeo {
    private static int i = 0;

    public static void main(String[] args) throws InterruptedException {
        int numThreads = 2;
        int numIncrementsPerThread = 100000;
```

```
        Thread[] threads = new Thread[numThreads];

        for (int j = 0; j < numThreads; j++) {
            threads[j] = new Thread(() -> {
                for (int k = 0; k < numIncrementsPerThread; k++) {
                    i++;
                }
            });
            threads[j].start();
        }

        for (Thread thread : threads) {
            thread.join();
        }

        System.out.println("Final value of i = " + i);
        System.out.println("Expected value = " + (numThreads *
numIncrementsPerThread));
    }
}
```

输出如下：

```
Final value of i = 102249
Expected value = 200000
```

i 期望的值为 200000，但实际跑出来的是 102249，这证明 i++ 不是一个原子操作，对吧？

可见性

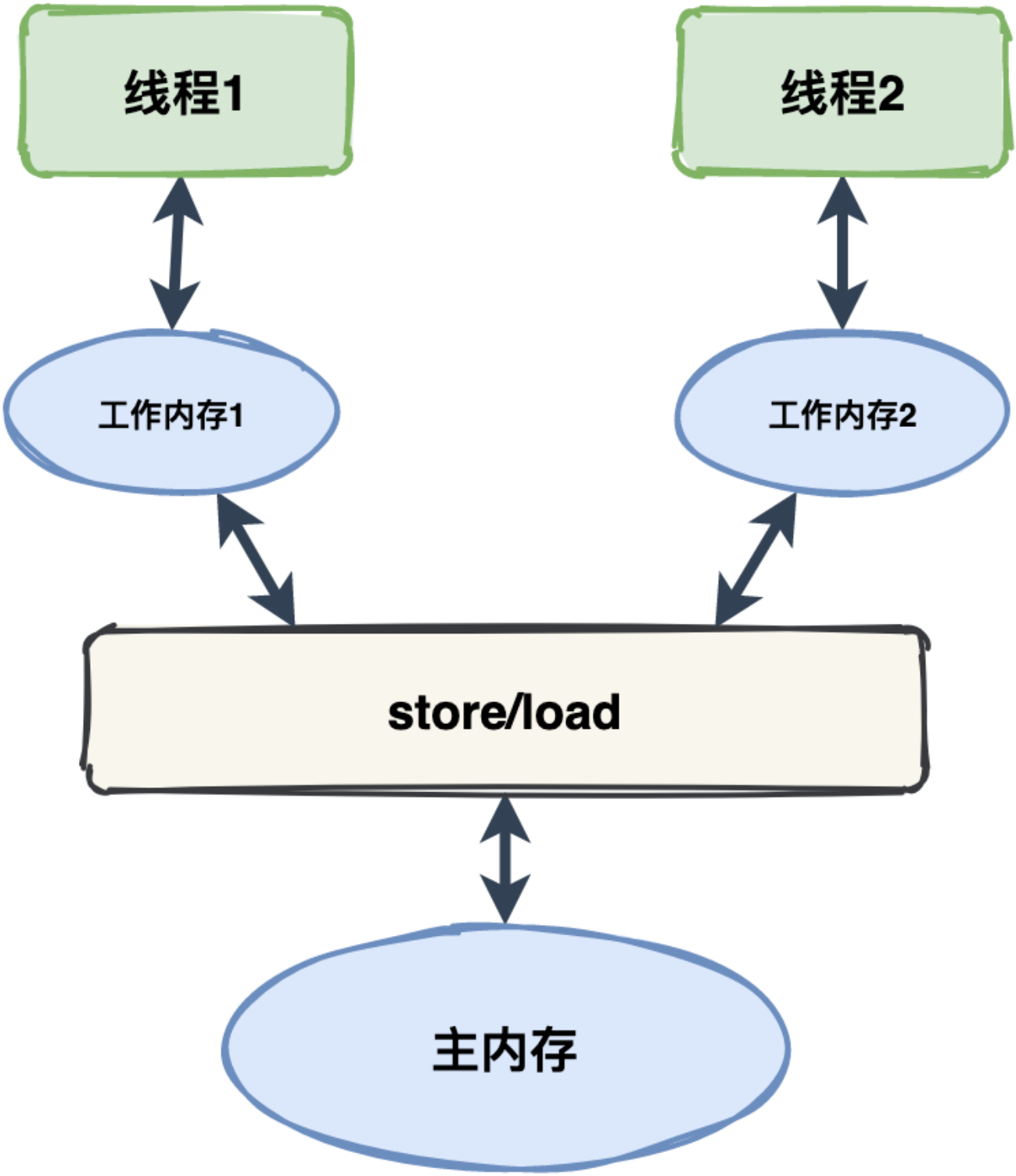
talk is cheap，show me code，来看这段代码：

```
class Test {
  int i = 50;
  int j = 0;

  public void update() {
    // 线程1执行
    i = 100;
  }

  public int get() {
    // 线程2执行
    j = i;
    return j;
  }
}
```

假如有两个线程，线程 1 执行 update 方法将 i 赋值为 100，一般情况下线程 1 会在自己的工作内存中完成赋值操作，但不会及时将新值刷新到主内存中。

这个时候线程 2 执行 get 方法，首先会从主内存中读取 i 的值，然后加载到自己的工作内存中，此时读到 i 的值仍然是 50，再将 50 赋值给 j，最后返回 j 的值就是 50 了。原本期望返回 100，结果返回 50，这就是可见性问题，线程 1 对变量 i 进行了修改，线程 2 并没有立即看到 i 的新值。

> 可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。



如上图，每个线程都有属于自己的工作内存，工作内存和主内存间需要通过 store 和 load 等进行交互。

为了解决多线程的可见性问题，Java 提供了`volatile`这个关键字。当一个共享变量被 volatile 修饰时，它会保证修改的值立即更新到主存当中，这样的话，当有其他线程需要读取时，就会从内存中读到新值。普通的共享变量不能保证可见性，因为变量被修改后什么时候刷回到主存是不确定的，因此另外一个线程读到的可能就是旧值。

当然 Java 的锁机制如 synchronized 和 lock 也是可以保证可见性的。

活跃性问题

上面讲到为了解决`可见性`的问题，我们可以采取加锁的方式来解决，但如果加锁使用不当也容易引入其他问题，比如『死锁』。

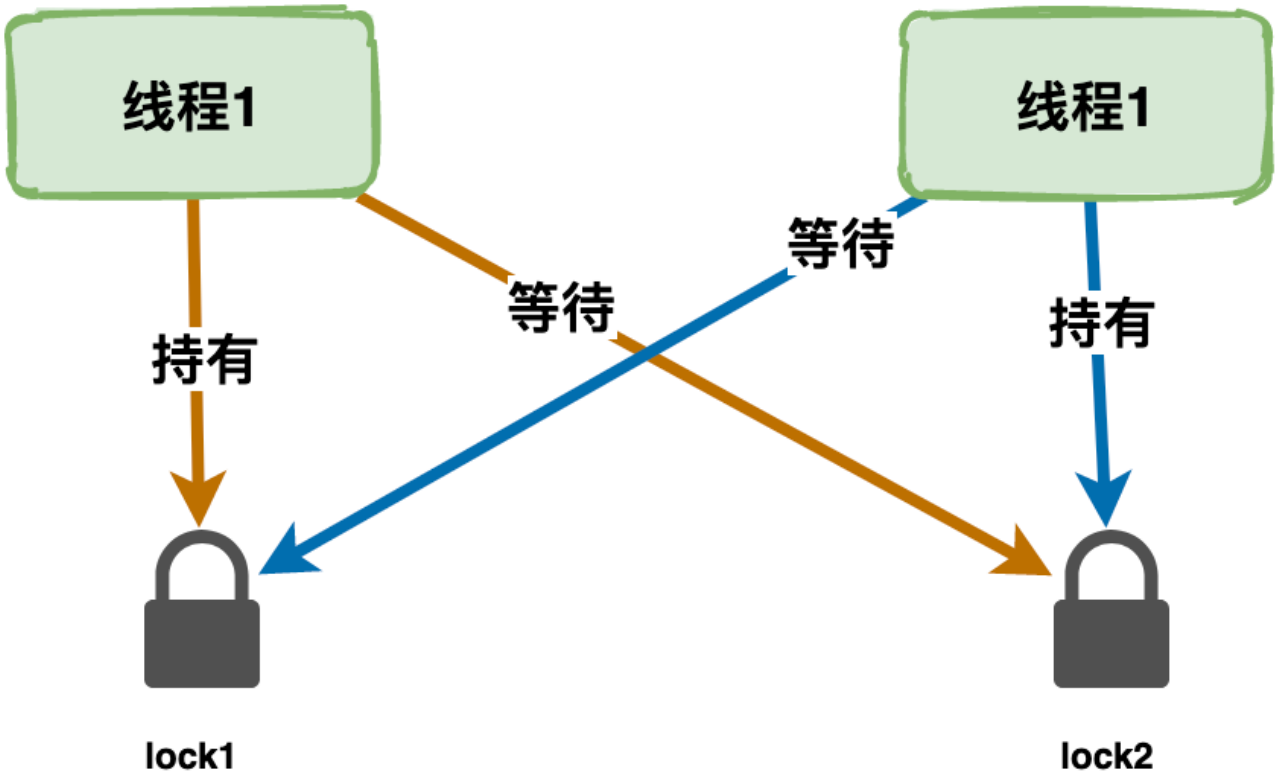
在讲『死锁』之前，我们需要先引入另外一个概念：`活跃性问题`。

> 活跃性是指某件正确的事情最终会发生，但当某个操作无法继续下去的时候，就会发生活跃性问题。

概念可能有点拗口，活跃性问题一般有这样几类：`死锁`，`活锁`，`饥饿问题`。

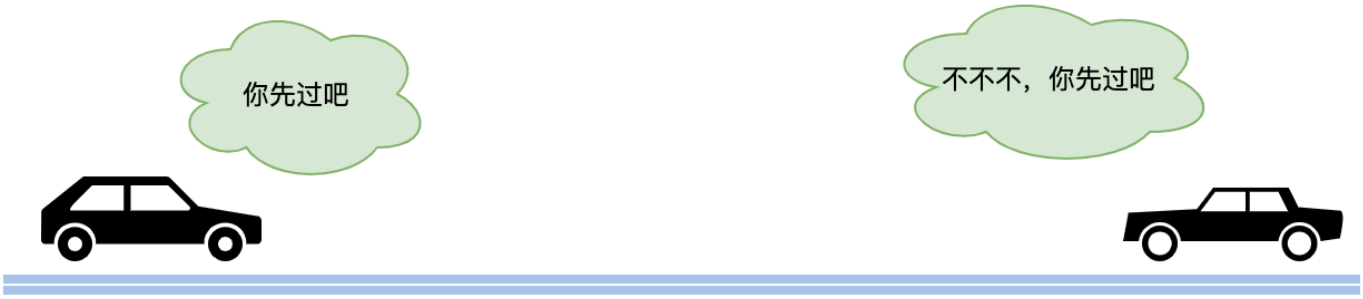
死锁

死锁是指多个线程因为环形等待锁的关系而永远地阻塞下去。



活锁

死锁是两个线程都在等待对方释放锁导致阻塞。而`活锁`的意思是线程没有阻塞，还活着呢。当多个线程都在运行并且都在修改各自的状态，而其他线程又依赖这个状态，就导致任何一个线程都无法继续执行，只能重复着自身的动作，于是就发生了活锁。



举一个生活中的例子，大家平时在走路的时候，迎面走来一个人，两个人互相让路，但是又同时走到了一个方向，如果一直这样重复着避让，这俩人就发生了活锁，学到了吧，嘿嘿。

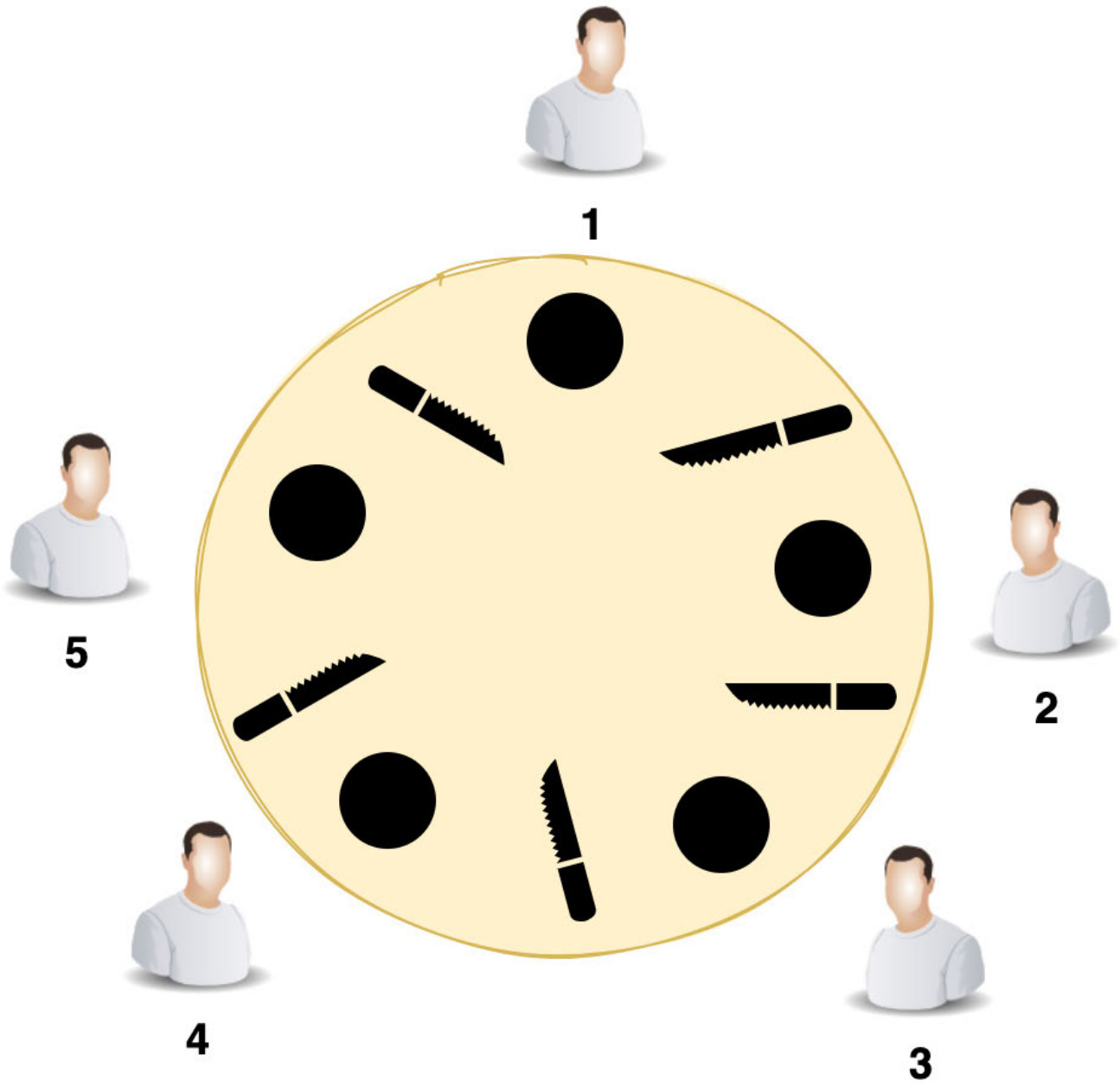
饥饿

如果一个线程无其他异常却迟迟不能继续运行，那基本上是处于饥饿状态了。

常见的有几种场景:

- 高优先级的线程一直在运行消耗 CPU，所有的低优先级线程一直处于等待；
- 一些线程被永久堵塞在一个等待进入同步块的状态，而其他线程总是能在它之前持续地对该同步块进行访问；

有一个非常经典的饥饿问题就是`哲学家用餐问题`，如下图所示，有五个哲学家在用餐，每个人必须要同时拿两把叉子才开始就餐，如果哲学家 1 和哲学家 3 同时开始就餐，那哲学家 2、4、5 就得饿肚子等待了。

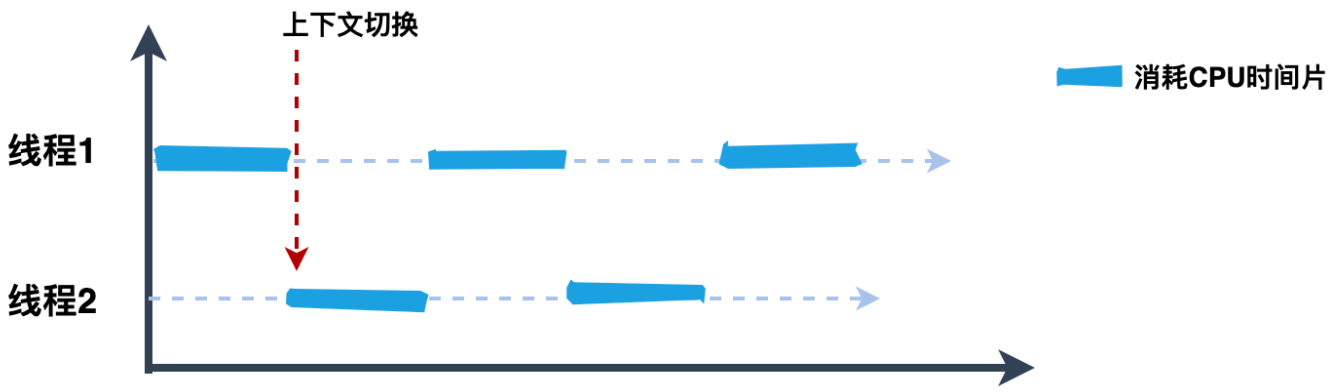


性能问题

前面讲到了线程安全和死锁、活锁这些问题，如果这些都没有发生，多线程并发一定比单线程串行执行快吗？答案是不一定，因为多线程有`创建线程`和`线程上下文切换`的开销。

创建线程是直接向系统申请资源的，对操作系统来说，创建一个线程的代价是十分昂贵的，需要给它分配内存、列入调度等。

线程创建完之后，还会遇到线程`上下文切换`。



CPU 是很宝贵的资源，速度非常快，为了保证雨露均沾，通常会给不同的线程分配`时间片`，当 CPU 从执行一个线程切换到执行另一个线程时，CPU 需要保存当前线程的本地数据，程序指针等状态，并加载下一个要执行线程的本地数据，程序指针等，也就是『上下文切换』。

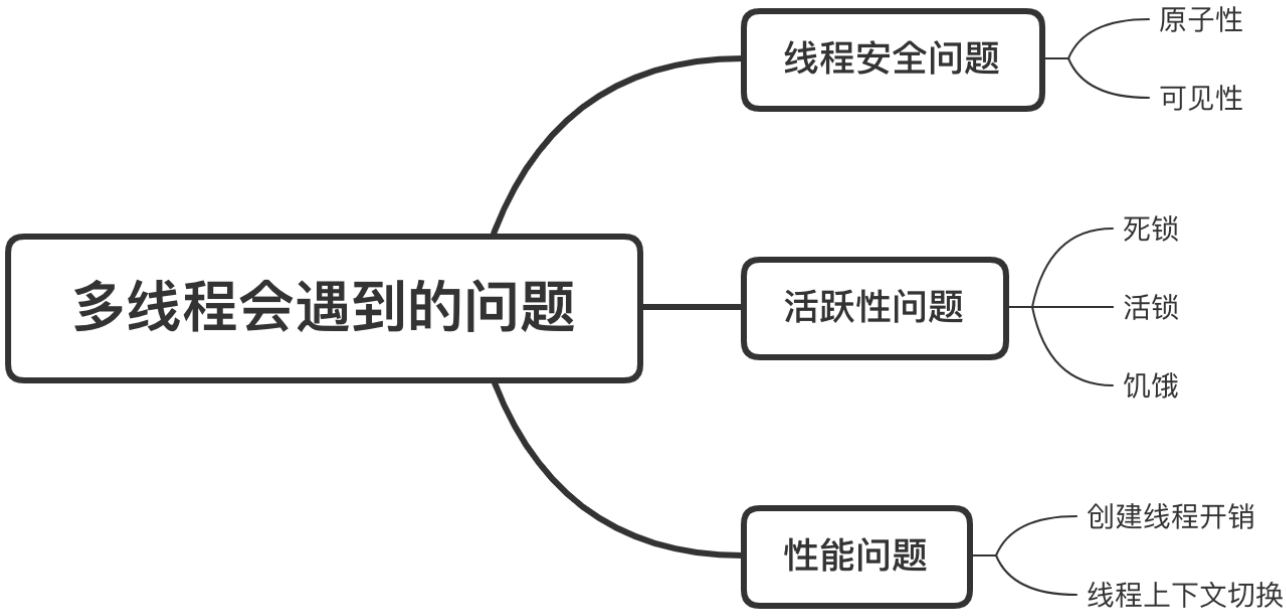
一般减少上下文切换的方法有：

- 无锁并发编程：可以参照 ConcurrentHashMap 锁分段的思想，不同的线程处理不同段的数据，这样在多线程竞争的条件下，可以减少上下文切换的时间。
- CAS 算法，利用 Atomic + CAS 算法来更新数据，采用乐观锁的方式，可以有效减少一部分不必要的锁竞争带来的上下文切换。
- 使用最少线程：避免创建不必要的线程，如果任务很少，但创建了很多的线程，这样就会造成大量的线程都处于等待状态。
- 协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

小结

多线程用好了可以让程序的效率成倍提升，用不好可能比单线程还要慢。

用一张图来总结一下上面讲的：



> 编辑：沉默王二，编辑前的内容来自于朋友雷小帅的开源仓库Java 八股文，内容很不错，强烈推荐。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。

第七节：Java的内存模型（JMM）

某年某月的某一天，小二去蚂蚁金服面试了，一上来，面试官老王就问他：『说说什么是 Java 的内存模型(JMM)吧？』

小二内心狂喜，不假思索地答道：『Java 内存主要分为五大块：堆、方法区、虚拟机栈、本地方法栈、PC 寄存器，balabala……』

老王会心一笑，露出一道光芒，略带轻蔑地打断了小王，说：『好了，今天的面试先到这里了，回去等通知吧』

一听到等通知这句话，小二心里知道，这场面试大概率是凉凉了。为什么呢？小二很是不解。

回到宿舍翻了翻《二哥的 Java 进阶之路并发编程篇》，小二才恍然大悟，原来自己弄错了概念，面试官是想考察 JMM，但是小二一听到`Java内存`这几个关键字就开始背Java 运行时内存区域的八股文了，害，Java 内存模型(JMM)和 Java 运行时内存区域的区别可大着呢。

Java 内存模型（Java Memory Model，JMM）定义了 Java 程序中的变量、线程如何和主存以及工作内存进行交互的规则。它主要涉及到多线程环境下的共享变量可见性、指令重排等问题，是理解并发编程中的关键概念。

并发编程的线程之间存在两个问题：

- 线程间如何通信？即：线程之间以何种机制来交换信息
- 线程间如何同步？即：线程以何种机制来控制不同线程间发生的相对顺序

有两种并发模型可以解决这两个问题：

- 消息传递并发模型
- 共享内存并发模型

这两种模型之间的区别如下图所示：

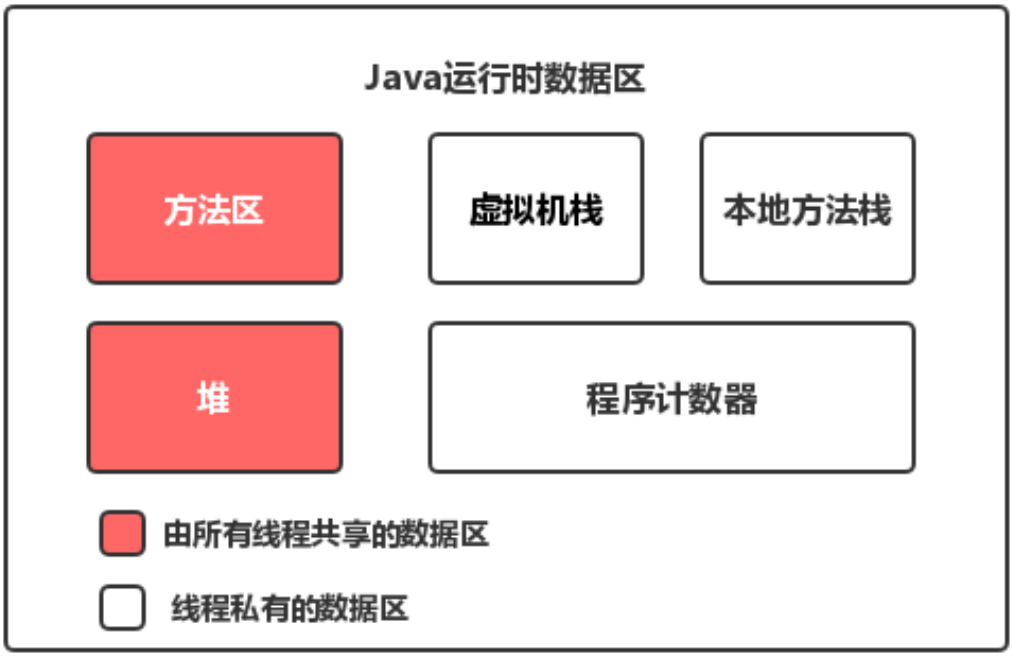
	如何通信	如何同步
消息传递并发模型	线程之间**没有公共状态**，线程间的通信必须通过发送消息来显示进行通信。	发送消息天然同步，因为发送消息总是在接受消息之前，因此同步是隐式的。
共享内存并发模型	线程之间共享程序的公共状态，通过**写-读内存中的公共状态**进行隐式通信。	**必须显式指定某段代码需要在线程之间互斥执行**，同步是显式的。

Java 使用的是共享内存并发模型。

Java 内存模型

什么是共享变量?

先来看一下运行时数据区，相信大家一点都不陌生：



对于每一个线程来说，栈都是私有的，而堆是共有的。

也就是说，在栈中的变量（局部变量、方法定义的参数、异常处理的参数）不会在线程之间共享，也就不会有内存可见性的问题，也不受内存模型的影响。而在堆中的变量是共享的，一般称之为共享变量。

所以，内存可见性针对的是**堆中的共享变量**。

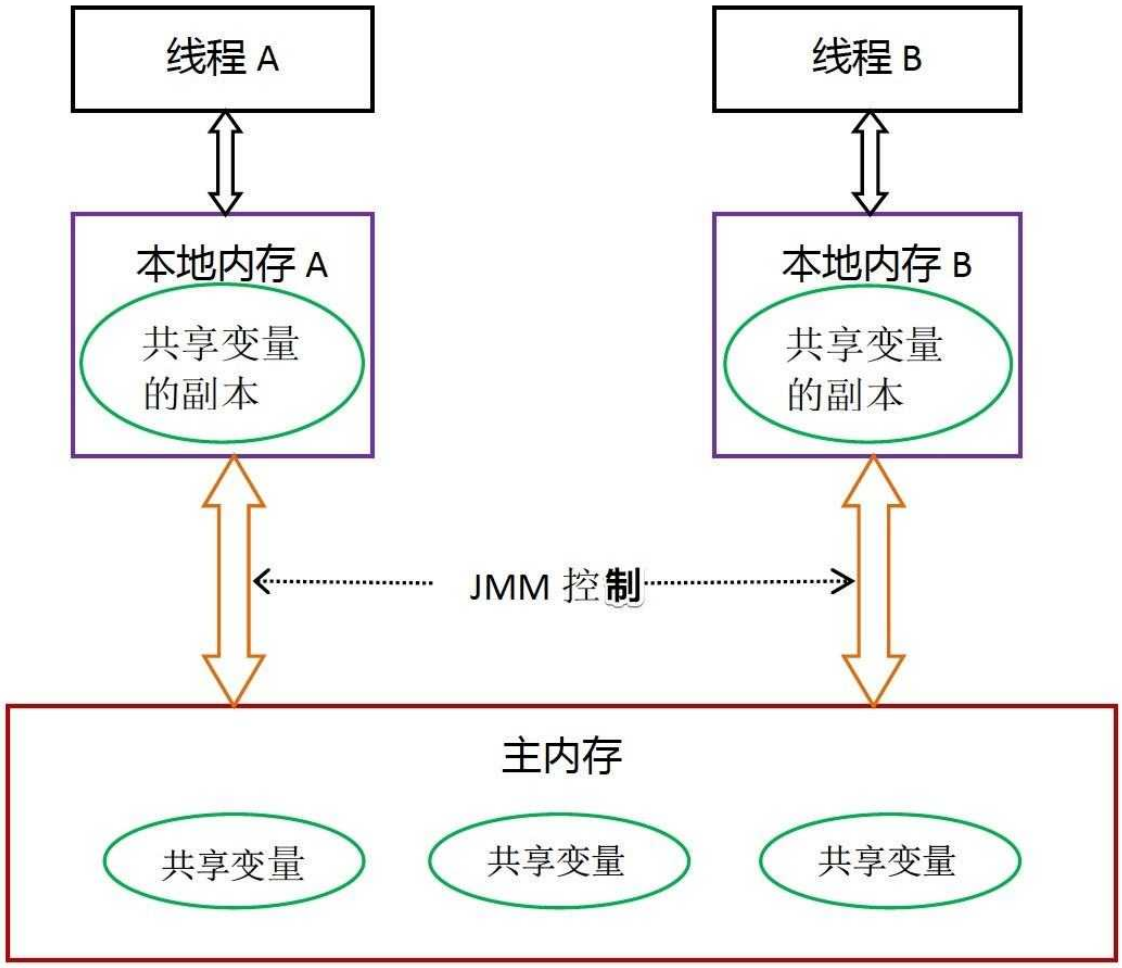
内存可见性问题是如何发生的?

那可能就有小伙伴会问：**既然堆是共享的，为什么在堆中会有内存不可见问题**?

这是因为现代计算机为了高效，往往会在高速缓存区中缓存共享变量，因为 CPU 访问缓存区比访问内存要快得多。

> 线程之间的共享变量存在于主存中，每个线程都有一个私有的本地内存，存储了该线程的读、写共享变量的副本。本地内存是 Java 内存模型的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器等。

Java 线程之间的通信由 Java 内存模型（简称 JMM）控制，从抽象的角度来说，JMM 定义了线程和主存之间的抽象关系。JMM 的抽象示意图如图所示：



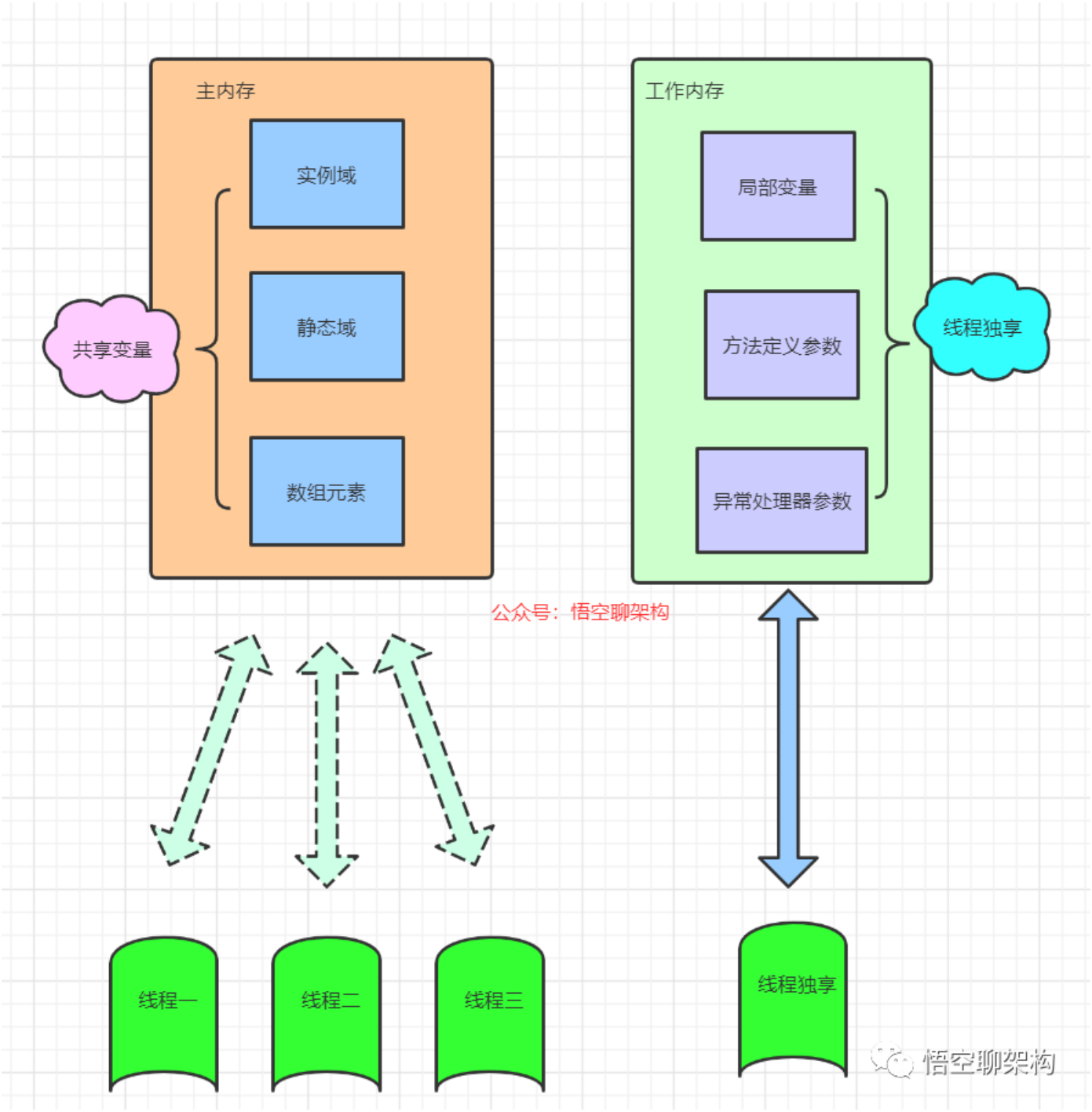
从图中可以看出：

1. 所有的共享变量都存在主存中。
2. 每个线程都保存了一份该线程使用到的共享变量的副本。
3. 如果线程 A 与线程 B 之间要通信的话，必须经历下面 2 个步骤：

1. 线程 A 将本地内存 A 中更新过的共享变量刷新到主存中去。
2. 线程 B 到主存中去读取线程 A 之前已经更新过的共享变量。

所以，线程 A 无法直接访问线程 B 的工作内存，线程间通信必须经过主存。

注意，根据 JMM 的规定，**线程对共享变量的所有操作都必须在自己的本地内存中进行，不能直接从主存中读取。**



- 主内存：Java堆中对象实例数据部分，对应于物理硬件的内存
- 工作内存：Java栈中的部分区域，优先存储于寄存器和高速缓存

所以线程 B 并不是直接去主存中读取共享变量的值，而是先在本地内存 B 中找到这个共享变量，发现这个共享变量已经被更新了，然后本地内存 B 去主存中读取这个共享变量的新值，并拷贝到本地内存 B 中，最后线程 B 再读取本地内存 B 中的新值。

如何保证内存可见性?

那么怎么知道这个共享变量的被其他线程更新了呢？这就是 JMM 的功劳了，也是 JMM 存在的必要性之一。**JMM 通过控制主存与每个线程的本地内存之间的交互，来提供内存可见性保证。**

Java 中的 volatile 关键字可以保证多线程操作共享变量的可见性以及禁止指令重排序，synchronized 关键字不仅保证可见性，同时也保证了原子性（互斥性）。

在更底层，JMM 通过内存屏障来实现内存的可见性以及禁止重排序。为了程序员更方便地理解，设计者提出了 happens-before 的概念（下文会细讲），它更加简单易懂，从而避免了程序员为了理解内存可见性而去学习复杂的重排序规则，以及这些规则的具体实现方法。

JMM 与 Java 运行时内存区域的区别

前面提到了 JMM 和 Java 运行时内存区域的划分，这两者既有差别又有联系：

- 区别

  两者是不同的概念。JMM 是抽象的，他是用来描述一组规则，通过这个规则来控制各个变量的访问方式，围绕原子性、有序性、可见性等展开。而 Java 运行时内存的划分是具体的，是 JVM 运行 Java 程序时必要的内存划分。

- 联系

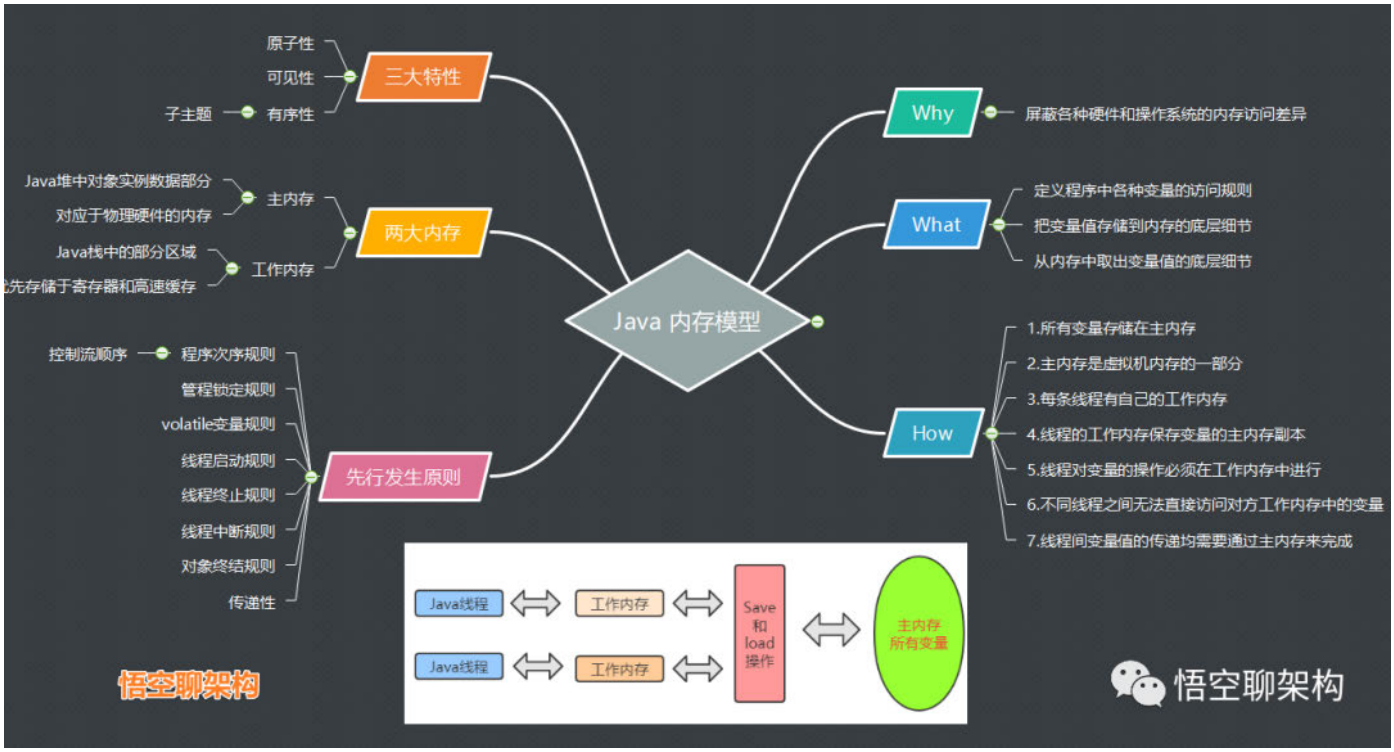
  都存在私有数据区域和共享数据区域。一般来说，JMM 中的主存属于共享数据区域，包含了堆和方法区；同样，JMM 中的本地内存属于私有数据区域，包含了程序计数器、本地方法栈、虚拟机栈。

总结一下：

Java 运行时内存区域描述的是在 JVM 运行时，如何将内存划分为不同的区域，并且每个区域的功能和工作机制。主要包括以下几个部分：

- 方法区：存储了每一个类的结构信息，如运行时常量池、字段和方法数据、构造方法和普通方法的字节码内容。
- 堆：几乎所有的对象实例以及数组都在这里分配内存。这是 Java 内存管理的主要区域。
- 栈：每一个线程有一个私有的栈，每一次方法调用都会创建一个新的栈帧，用于存储局部变量、操作数栈、动态链接、方法出口等信息。所有的栈帧都是在方法调用和方法执行完成之后创建和销毁的。
- 本地方法栈：与栈类似，不过本地方法栈为 JVM 使用到的 native 方法服务。
- 程序计数器：每个线程都有一个独立的程序计数器，用于指示当前线程执行到了字节码的哪一行。

Java 内存模型 (JMM) 主要针对的是多线程环境下，如何在主内存与工作内存之间安全地执行操作。



它涵盖的主题包括变量的可见性、指令重排、原子操作等，旨在解决由于多线程并发编程带来的一些问题。

- 可见性：当一个线程修改了共享变量的值，这个新值对于其他线程来说可以立即知道。
- 原子性：一个或多个操作在整个过程中，不会被其他的线程或者操作所打断，这些操作是一个整体，要么都执行，要么都不执行。
- 有序性：程序执行的顺序按照代码的先后顺序执行的。

JMM 与重排序

前面提到了，JMM 定义了多线程之间如何互相交互的规则，主要目的是为了解决由于编译器优化、处理器优化和缓存系统等导致的可见性、原子性和有序性。

那我们接下来就来聊聊重排序以及它所带来的顺序问题。

为什么指令重排可以提高性能?

大家都知道，计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排。

那可能有小伙伴就要问：**为什么指令重排序可以提高性能?**

简单地说，每一个指令都会包含多个步骤，每个步骤可能使用不同的硬件。因此，**流水线技术**产生了，它的原理是指令 1 还没有执行完，就可以开始执行指令 2，而不用等到指令 1 执行结束后再执行指令 2，这样就大大提高了效率。

但是，流水线技术最害怕**中断**，恢复中断的代价是比较大的，所以我们要想尽办法不让流水线中断。指令重排就是减少中断的一种技术。

我们分析一下下面这段代码的执行情况：

```
a = b + c;
d = e - f ;
```

先加载 b、c（**注意，有可能先加载 b，也有可能先加载 c**），但是在执行 `add(b,c)` 的时候，需要等待 b、c 装载结束才能继续执行，也就是需要增加停顿，那么后面的指令（加载 e 和 f）也会有停顿，这就降低了计算机的执行效率。

为了减少停顿，我们可以在加载完 b 和 c 后把 e 和 f 也加载了，然后再去执行 `add(b,c)`，这样做对程序（串行）是没有影响的，但却减少了停顿。

换句话说，既然 `add(b,c)` 需要停顿，那还不如去做一些有意义的事情（加载 e 和 f）。

综上所述，**指令重排对于提高 CPU 性能十分必要，但也带来了乱序的问题。**

重排序有哪几种?

指令重排一般分为以下三种：

- **编译器优化重排**，编译器在**不改变单线程程序语义**的前提下，重新安排语句的执行顺序。
- **指令并行重排**，现代处理器采用了指令级并行技术来将多条指令重叠执行。如果**不存在数据依赖性**(即后一个执行的语句无需依赖前面执行的语句的结果)，处理器可以改变语句对应的机器指令的执行顺序。
- **内存系统重排**，由于处理器使用缓存和读写缓存冲区，这使得加载(load)和存储(store)操作看上去可能是在乱序执行，因为三级缓存的存在，导致内存与缓存的数据同步存在时间差。

指令重排可以保证串行语义一致，但是没有义务保证多线程间的语义也一致。所以在多线程下，指令重排序可能会导致一些问题。

JMM 与顺序一致性模型

当程序未正确同步的时候，就可能存在数据竞争。

> 数据竞争：在一个线程中写一个变量，在另一个线程读同一个变量，并且写和读没有通过同步来排序。

如果程序中包含了数据竞争，那么运行的结果往往充满了**不确定性**，比如读发生在了写之前，可能就会读到错误的值；如果一个线程能够正确同步，那么就不存在数据竞争。

Java 内存模型（JMM）对于正确同步多线程程序的内存一致性做了以下保证：**如果程序是正确同步的，程序的执行将具有顺序一致性**。即程序的执行结果和该程序在顺序一致性模型中执行的结果相同。

这里的同步包括使用 volatile、final、synchronized 等关键字实现的同步。

如果我们开发者没有正确使用`volatile`、`final`、`synchronized` 等关键字，那么即便是使用了同步，JMM 也不会有内存可见性的保证，很可能会导致程序出错，并且不可重现，很难排查。

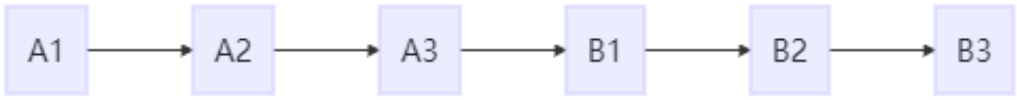
什么是顺序一致性模型?

顺序一致性模型是一个**理想化的理论参考模型**，它为程序提供了极强的内存可见性保证。顺序一致性模型有两大特性：

- 一个线程中的所有操作必须按照程序的顺序（即 Java 代码的顺序）来执行。
- 不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序。即在顺序一致性模型中，每个操作必须是**原子性的，且立刻对所有线程可见**。

为了理解这两个特性，我们举个例子，假设有两个线程 A 和 B 并发执行，线程 A 有 3 个操作，他们在程序中的顺序是 A1->A2->A3，线程 B 也有 3 个操作，B1->B2->B3。

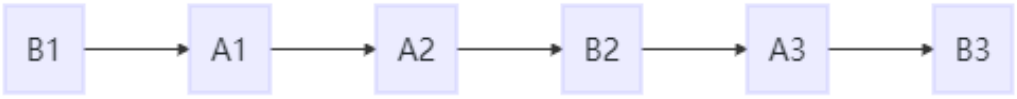
假设**正确使用了同步**，A 线程的 3 个操作执行后释放锁，B 线程获取同一个锁。那么在**顺序一致性模型**中的执行效果如下所示：



操作的执行整体上有序，并且两个线程都只能看到这个执行顺序。

JMM 为什么不保证顺序一致性?

假设**没有使用同步**，那么在**顺序一致性模型**中的执行效果如下所示：



操作的执行整体上无序，但是两个线程都只能看到这个执行顺序。之所以可以得到这个保证，是因为顺序一致性模型中的**每个操作必须立即对任意线程可见**。

但是 JMM 没有这样的保证。

比如，在当前线程把写过的数据缓存在本地内存中，在没有刷新到主存之前，这个写操作仅对当前线程可见；从其他线程的角度来观察，这个写操作根本没有被当前线程所执行。

只有当前线程把本地内存中写过的数据刷新到主存之后，这个写操作才对其他线程可见。在这种情况下，当前线程和其他线程看到的执行顺序是不一样的。

在顺序一致性模型中，所有操作完全按照程序的顺序串行执行。但是 JMM 中，临界区内（同步块或同步方法中）的代码可以发生重排序（但不允许临界区内的代码“逃逸”到临界区之外，因为会破坏锁的内存语义）。

虽然线程 A 在临界区做了重排序，但是因为锁的特性，线程 B 无法观察到线程 A 在临界区的重排序。这种重排序既提高了执行效率，又没有改变程序的执行结果。

同时，JMM 会在退出临界区和进入临界区做特殊的处理，使得在临界区内程序获得与顺序一致性模型相同的内存视图。

由此可见，JMM 的具体实现方针是：在不改变（正确同步的）程序执行结果的前提下，尽量为编译期和处理器的优化打开方便之门。

对于未同步的多线程，JMM 只提供**最小安全性**：线程读取到的值，要么是之前某个线程写入的值，要么是默认值，不会无中生有。

为了实现这个安全性，JVM 在堆上分配对象时，首先会对内存空间清零，然后才会在上面分配对象（这两个操作是同步的）。

JMM 没有保证未同步程序的执行结果与该程序在顺序一致性中执行结果一致。因为如果要保证执行结果一致，那么 JMM 需要禁止大量的优化，对程序的执行性能会产生很大的影响。

未同步程序在 JMM 和顺序一致性内存模型中的执行特性有如下差异：

1. 顺序一致性保证单线程内的操作会按程序的顺序执行；JMM 不保证单线程内的操作会按程序的顺序执行。（因为重排序，但是 JMM 保证单线程下的重排序不影响执行结果）
2. 顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而 JMM 不保证所有线程能看到一致的操作执行顺序。（因为 JMM 不保证所有操作立即可见）
3. 顺序一致性模型保证对所有的内存读写操作都具有原子性，而 JMM 不保证对 64 位的 long 型和 double 型变量的写操作具有原子性。

JMM 与 happens-before

一方面，我们开发者需要 JMM 提供一个强大的内存模型来编写代码；另一方面，编译器和处理器希望 JMM 对它们的束缚越少越好，这样它们就可以尽可能多的做优化来提高性能，希望的是一个弱的内存模型。

JMM 考虑了这两种需求，并且找到了平衡点，对编译器和处理器来说，**只要不改变程序的执行结果（单线程程序和正确同步了的多线程程序），编译器和处理器怎么优化都行。**

对于我们开发者来说，JMM 提供了**happens-before 规则**（JSR-133 规范），满足了我们的诉求——**简单易懂，并且提供了足够强的内存可见性保证。**换言之，我们开发者只要遵循 happens-before 规则，那么我们写的程序就能保证在 JMM 中具有强的内存可见性。

JMM 使用 happens-before 的概念来定制两个操作之间的执行顺序。这两个操作可以在一个线程内，也可以是不同的线程种。

happens-before 关系的定义如下：

1. 如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
2. **两个操作之间存在 happens-before 关系，并不意味着 Java 平台的具体实现必须要按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与按 happens-before 关系来执行的结果一致，那么 JMM 也允许这样的重排序。**

happens-before 关系本质上和 as-if-serial 语义是一回事。

as-if-serial 语义保证单线程内重排序后的执行结果和程序代码本身应有的结果是一致的，happens-before 关系保证正确同步的多线程程序的执行结果不被重排序改变。

总之，**如果操作 A happens-before 操作 B，那么操作 A 在内存上所做的操作对操作 B 都是可见的，不管它们在不在一个线程。**

happens-before 关系有哪些?

在 Java 中，有以下天然的 happens-before 关系：

- 程序顺序规则：一个线程中的每一个操作，happens-before 于该线程中的任意后续操作。
- 监视器锁规则：对一个锁的解锁，happens-before 于随后对这个锁的加锁。
- volatile 变量规则：对一个 volatile 域的写，happens-before 于任意后续对这个 volatile 域的读。
- 传递性：如果 A happens-before B，且 B happens-before C，那么 A happens-before C。
- start 规则：如果线程 A 执行操作 `ThreadB.start()` 启动线程 B，那么 A 线程的 `ThreadB.start()` 操作 happens-before 于线程 B 中的任意操作。
- join 规则：如果线程 A 执行操作 `ThreadB.join()` 并成功返回，那么线程 B 中的任意操作 happens-before 于线程 A 从 `ThreadB.join()` 操作成功返回。

举例：

```
int a = 1; // A操作
int b = 2; // B操作
int sum = a + b;// C 操作
System.out.println(sum);
```

根据以上介绍的 happens-before 规则，假如只有一个线程，那么不难得出：

```
1> A happens-before B
2> B happens-before C
3> A happens-before C
```

注意，真正在执行指令的时候，其实 JVM 有可能对操作 A & B 进行重排序，因为无论先执行 A 还是 B，他们都对对方是可见的，并且不影响执行结果。

如果这里发生了重排序，这在视觉上违背了 happens-before 原则，但是 JMM 是允许这样的重排序的。

所以，我们只关心 happens-before 规则，不用关心 JVM 到底是怎样执行的。只要确定操作 A happens-before 操作 B 就行了。

重排序有两类，JMM 对这两类重排序有不同的策略：

- 会改变程序执行结果的重排序，比如 A -> C，JMM 要求编译器和处理器都禁止这种重排序。
- 不会改变程序执行结果的重排序，比如 A -> B，JMM 对编译器和处理器不做要求，允许这种重排序。

小结

- Java 内存模型（JMM）定义了 Java 程序中的变量、线程如何和主存以及工作内存进行交互的规则。它主要涉及到多线程环境下的共享变量可见性、指令重排等问题，是理解并发编程中的关键概念。
- Java 内存模型（JMM）主要针对的是多线程环境下，如何在主内存与工作内存之间安全地执行操作。
- Java 运行时内存区域描述的是在 JVM 运行时，如何将内存划分为不同的区域，并且每个区域的功能和工作机制。主要包括方法区、堆、栈、本地方法栈、程序计数器。
- 指令重排是为了提高 CPU 性能，但是可能会导致一些问题，比如多线程环境下的内存可见性问题。
- happens-before 规则是 JMM 提供的强大的内存可见性保证，只要遵循 happens-before 规则，那么我们写的程序就能保证在 JMM 中具有强的内存可见性。

> 编辑：沉默王二，编辑前的内容部分来源于朋友雷小帅的开源仓库 Java 八股文，强烈推荐。部分内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。

第八节：volatile 关键字

“三妹啊，这节我们来学习 Java 中的 volatile 关键字吧，以及容易遇到的坑。”看着三妹好学的样子，我倍感欣慰。

“好呀，哥。”三妹愉快的答应了。

> 这是我们在《二哥的 Java 进阶之路基础篇》中常见的对话模式，老读者应该对这种模式不陌生。

在讲并发编程带来了哪些问题的时候，我们提到了可见性和原子性，那我现在可以直接告诉大家了：volatile 可以保证可见性，但不保证原子性：

- 当写一个 volatile 变量时，JMM 会把该线程在本地内存中的变量强制刷新到主内存中去；
- 这个写操作会导致其他线程中的 volatile 变量缓存无效。

volatile 会禁止指令重排

在讲 JMM 的时候，我们提到了指令重排，相信大家都还有印象，我们来回顾一下重排序需要遵守的规则：

- 重排序不会对存在数据依赖关系的操作进行重排序。比如：`a=1;b=a;` 这个指令序列，因为第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
- 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变。比如：`a=1;b=2;c=a+b` 这三个操作，第一步 (a=1) 和第二步 (b=2) 由于不存在数据依赖关系，所以可能会发生重排序，但是 c=a+b 这个操作是不会被重排序的，因为需要保证最终的结果一定是 c=a+b=3。

使用 volatile 关键字修饰共享变量可以禁止这种重排序。怎么做到的呢?

当我们使用 volatile 关键字来修饰一个变量时，Java 内存模型会插入内存屏障（一个处理器指令，可以对 CPU 或编译器重排序做出约束）来确保以下两点：

- 写屏障（Write Barrier）：当一个 volatile 变量被写入时，写屏障确保在该屏障之前的所有变量的写入操作都提交到主内存。
- 读屏障（Read Barrier）：当读取一个 volatile 变量时，读屏障确保在该屏障之后的所有读操作都从主内存中读取。

换句话说：

- 当程序执行到 volatile 变量的读操作或者写操作时，在其前面操作的更改肯定已经全部进行，且结果对后面的操作可见；在其后面的操作肯定还没有进行；
- 在进行指令优化时，不能将 volatile 变量的语句放在其后面执行，也不能把 volatile 变量后面的语句放到其前面执行。

“也就是说，执行到 volatile 变量时，其前面的所有语句都必须执行完，后面所有得语句都未执行。且前面语句的结果对 volatile 变量及其后面语句可见。”我瞅了了三妹一眼继续说。

先看下面未使用 volatile 的代码：

```
class ReorderExample {
  int a = 0;
  boolean flag = false;
  public void writer() {
      a = 1;                    //1
      flag = true;              //2
  }
  Public void reader() {
      if (flag) {                 //3
          int i =  a * a;         //4
          System.out.println(i);
      }
  }
}
```

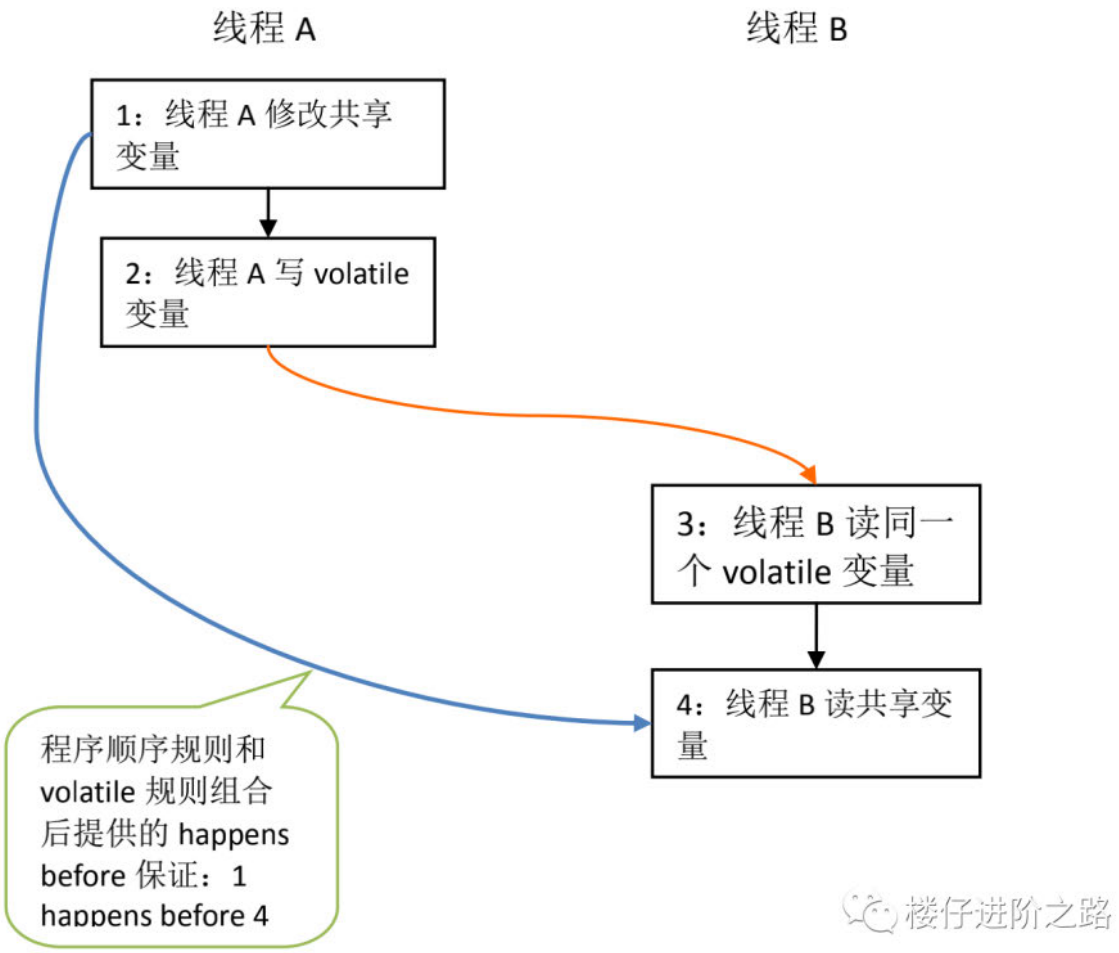
因为重排序影响，所以最终的输出可能是 0，重排序请参考上一篇 JMM 的介绍，如果引入 volatile，我们再看一下代码：

```
class ReorderExample {
  int a = 0;
  boolean volatile flag = false;
  public void writer() {
      a = 1;                    //1
      flag = true;              //2
  }
  Public void reader() {
      if (flag) {                 //3
          int i =  a * a;         //4
          System.out.println(i);
      }
  }
}
```

这时候，volatile 会禁止指令重排序，这个过程建立在 happens before 关系（上一篇介绍过了）的基础上：

1. 根据程序次序规则，1 happens before 2; 3 happens before 4。
2. 根据 volatile 规则，2 happens before 3。
3. 根据 happens before 的传递性规则，1 happens before 4。

上述 happens before 关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个 happens before 关系：

- 黑色箭头表示程序顺序规则；
- 橙色箭头表示 volatile 规则；
- 蓝色箭头表示组合这些规则后提供的 happens before 保证。

这里 A 线程写一个 volatile 变量后，B 线程读同一个 volatile 变量。A 线程在写 volatile 变量之前所有可见的共享变量，在 B 线程读同一个 volatile 变量后，将立即变得对 B 线程可见。

volatile 不适用的场景

下面是变量自加的示例：

```
public class volatileTest {
    public volatile int inc = 0;
    public void increase() {
        inc++;
```

```
    }
    public static void main(String[] args) {
        final volatileTest test = new volatileTest();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("inc output:" + test.inc);
    }
}
```

测试输出：

```
inc output:8182
```

“为什么呀？二哥？”看到这个结果，三妹疑惑地问。

“因为 inc++不是一个原子性操作（前面讲过），由读取、加、赋值 3 步组成，所以结果并不能达到 10000。”我耐心地回答。

“哦，你这样说我就理解了。”三妹点点头。

怎么解决呢？

01、采用 synchronized（下一篇会讲，戳链接直达），把 `inc++` 拎出来单独加 synchronized 关键字：

```
public class volatileTest1 {
    public int inc = 0;
    public synchronized void increase() {
        inc++;
    }
    public static void main(String[] args) {
        final volatileTest1 test = new volatileTest1();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add synchronized, inc output:" + test.inc);
```

```
    }
}
```

02、采用 Lock，通过重入锁 ReentrantLock 对 `inc++` 加锁（后面都会细讲，戳链接直达）：

```
public class volatileTest2 {
    public int inc = 0;
    Lock lock = new ReentrantLock();
    public void increase() {
        lock.lock();
        inc++;
        lock.unlock();
    }
    public static void main(String[] args) {
        final volatileTest2 test = new volatileTest2();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add lock, inc output:" + test.inc);
    }
}
```

03、采用原子类 AtomicInteger（后面也会细讲，戳链接直达）来实现：

```
public class volatileTest3 {
    public AtomicInteger inc = new AtomicInteger();
    public void increase() {
        inc.getAndIncrement();
    }
    public static void main(String[] args) {
        final volatileTest3 test = new volatileTest3();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<100;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
```

```
        System.out.println("add AtomicInteger, inc output:" + test.inc);
    }
}
```

三者输出都是 1000，如下：

```
add synchronized, inc output:1000
add lock, inc output:1000
add AtomicInteger, inc output:1000
```

volatile 实现单例模式的双重锁

这是一个使用"双重检查锁定"（double-checked locking）实现的单例模式（Singleton Pattern）的例子。

```
public class penguin {
    private static volatile penguin m_penguin = null;
    // 避免通过new初始化对象
    private void penguin() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin getInstance() {       //1
        if (null == m_penguin) {                //2
            synchronized(penguin.class) {       //3
                if (null == m_penguin) {        //4
                    m_penguin = new penguin(); //5
                }
            }
        }
        return m_penguin;                       //6
    }
}
```

在这个例子中，penguin 类只能被实例化一次。

首先看代码解释：

- 声明了一个类型为 penguin 的 volatile 变量 m_penguin，它是类的静态变量，用来存储 penguin 类的唯一实例。
- `penguin()` 构造方法被声明为 private，这样就阻止了外部代码使用 new 来创建 penguin 实例，保证了只能通过 `getInstance()` 方法获取实例。
- `getInstance()` 方法是获取 penguin 类唯一实例的公共静态方法。
- `if (null == m_penguin)` 检查是否已经存在实例。如果不存在，才进入同步代码块。
- `synchronized(penguin.class)` 对类的 Class 对象加锁，这是确保在多线程环境下，同时只能有一个线程进入同步代码块。在同步代码块中，再次检查实例是否已经存在，如果不存在，则创建新的实例。这就是所谓的"双重检查锁定"。
- 最后返回 m_penguin，也就是 penguin 的唯一实例。

其中，使用 volatile 关键字是为了防止 `m_penguin = new penguin()` 这一步被指令重排序。实际上，`new penguin()` 这一步分为三个子步骤：

- 分配对象的内存空间。
- 初始化对象。
- 将 m_penguin 指向分配的内存空间。

如果不使用 volatile 关键字，JVM 可能会对这三个子步骤进行指令重排序，如果步骤 2 和步骤 3 被重排序，那么线程 A 可能在对象还没有被初始化完成时，线程 B 已经开始使用这个对象，从而导致问题。而使用 volatile 关键字可以防止这种指令重排序。

伪代码代码如下：

```
a. memory = allocate() //分配内存
b. ctorInstanc(memory) //初始化对象
c. instance = memory   //设置instance指向刚分配的地址
```

上面的代码在编译运行时，可能会出现重排序从 a-b-c 排序为 a-c-b。在多线程的情况下会出现以下问题。

当线程 A 在执行第 5 行代码时，B 线程进来执行到第 2 行代码。假设此时 A 执行的过程中发生了指令重排序，即先执行了 a 和 c，没有执行 b。那么由于 A 线程执行了 c 导致 instance 指向了一段地址，所以 B 线程判断 instance 不为 null，会直接跳到第 6 行并返回一个未初始化的对象。

小结

“好了，三妹，我们来总结一下。”我舒了一口气说。

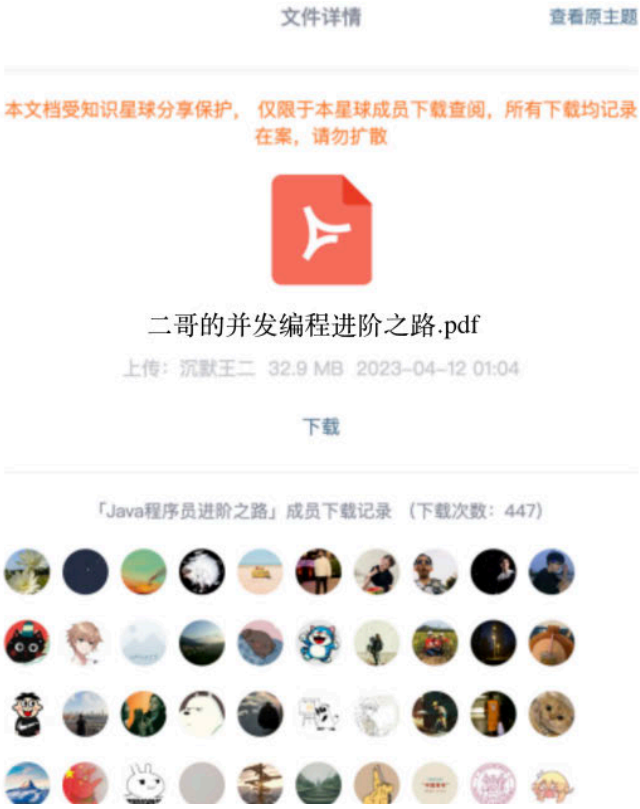
volatile 可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在 JVM 底层 volatile 是采用“内存屏障”来实现的。

观察加入 volatile 关键字和没有加入 volatile 关键字时所生成的汇编代码就能发现，加入 volatile 关键字时，会多出一个 lock 前缀指令，lock 前缀指令实际上相当于一个内存屏障（也称内存栅栏），内存屏障会提供 3 个功能：

- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 它会强制将对缓存的修改操作立即写入主存；
- 如果是写操作，它会导致其他 CPU 中对应的缓存行无效。

最后，我们学习了 volatile 不适用的场景，以及解决的方法，并解释了双重检查锁定实现的单例模式为何需要使用 volatile。

> 编辑：沉默王二，编辑前的内容主要来自于二哥的技术派团队成员楼仔，原文链接戳：volatile。



第九节：synchronized关键字

“三妹，今天我们来学习 synchronized 关键字吧。”我瞅了一眼坐在沙发上的三妹说，她正在王者荣耀呢。

三妹（颜值在线，气质也在线）喝了一口冰可乐接着说：“好的，二哥。马上，打完这把，看我三杀。”

在 Java 中，关键字 synchronized 可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到 synchronized 的另外一个重要的作用，synchronized 可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代 volatile 功能）。

synchronized 关键字最主要有以下 3 种应用方式：

- 同步方法，为当前对象（this）加锁，进入同步代码前要获得当前对象的锁；
- 同步静态方法，为当前类加锁（锁的是 Class 对象），进入同步代码前要获得当前类的锁；
- 同步代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

synchronized同步方法

通过在方法声明中加入 synchronized 关键字，可以保证在任意时刻，只有一个线程能执行该方法。

来看代码：

```
public class AccountingSync implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 同步方法
    public synchronized void increase() {
```

```
        i ++;
    }
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
    }
    public static void main(String args[]) throws InterruptedException {
        AccountingSync instance = new AccountingSync();
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
```

输出结果：

```
/**
 * 输出结果：
 * static, i output:2000000
 */
```

如果在方法 `increase()` 前不加 synchronized，因为 i++ 不具备原子性，所以最终结果会小于 2000000，具体分析可以参考《volatile》的内容。

> 注意：一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他 synchronized 方法，但是其他线程还是可以访问该对象的其他非 synchronized 方法。

但是，如果一个线程 A 需要访问对象 obj1 的 synchronized 方法 f1(当前对象锁是 obj1)，另一个线程 B 需要访问对象 obj2 的 synchronized 方法 f2(当前对象锁是 obj2)，这样是允许的：

```
public class AccountingSyncBad implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 同步方法
    public synchronized void increase() {
        i ++;
    }

    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
```

```
    }

    public static void main(String args[]) throws InterruptedException {
        // new 两个AccountingSync新实例
        Thread t1 = new Thread(new AccountingSyncBad());
        Thread t2 = new Thread(new AccountingSyncBad());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
```

输出结果：

```
/**
 * 输出结果：
 * static, i output:1224617
 */
```

上述代码与前面不同的是，我们创建了两个对象 AccountingSyncBad，然后启动两个不同的线程对共享变量 i 进行操作，但很遗憾，操作结果是 1224617 而不是期望的结果 2000000。

因为上述代码犯了严重的错误，虽然使用了 synchronized 同步 increase 方法，但却 new 了两个不同的对象，这也就意味着存在着两个不同的对象锁，因此 t1 和 t2 都会进入各自的对象锁，也就是说 t1 和 t2 线程使用的是不同的锁，因此线程安全是无法保证的。

> 每个对象都有一个对象锁，不同的对象，他们的锁不会互相影响。

解决这种问题的的方式是将 synchronized 作用于静态的 increase 方法，这样的话，对象锁就锁的是当前的类，由于无论创建多少个对象，类永远只有一个，所有在这样的情况下对象锁就是唯一的。

参考：对象和类

synchronized同步静态方法

当 synchronized 同步静态方法时，锁的是当前类的 Class 对象，不属于某个对象。当前类的 Class 对象锁被获取，不影响实例对象锁的获取，两者互不影响，本质上是 this 和 Class 的不同。

由于静态成员变量不专属于任何一个对象，因此通过 Class 锁可以控制静态成员变量的并发操作。

需要注意的是如果线程 A 调用了一个对象的非静态 synchronized 方法，线程 B 需要调用这个对象所属类的静态 synchronized 方法，是不会发生互斥的，因为访问静态 synchronized 方法占用的锁是当前类的 Class 对象，而访问非静态 synchronized 方法占用的锁是当前对象（this）的锁，看如下代码：

```
public class AccountingSyncClass implements Runnable {
    static int i = 0;
    /**
     * 同步静态方法,锁是当前class对象，也就是
```

```
     * AccountingSyncClass类对应的class对象
     */
    public static synchronized void increase() {
        i++;
    }
    // 非静态,访问时锁不一样不会发生互斥
    public synchronized void increase4Obj() {
        i++;
    }
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        //new新实例
        Thread t1=new Thread(new AccountingSyncClass());
        //new新实例
        Thread t2=new Thread(new AccountingSyncClass());
        //启动线程
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
/**
 * 输出结果：
 * 2000000
 */
```

由于 synchronized 关键字同步的是静态的 increase 方法，与同步实例方法不同的是，其锁对象是当前类的 Class 对象。

注意代码中的 increase4Obj 方法是实例方法，其对象锁是当前实例对象（this），如果别的线程调用该方法，将不会产生互斥现象，毕竟锁的对象不同，这种情况下可能会发生线程安全问题(操作了共享静态变量 i)。

synchronized同步代码块

某些情况下，我们编写的方法代码量比较多，存在一些比较耗时的操作，而需要同步的代码块只有一小部分，如果直接对整个方法进行同步，可能会得不偿失，此时我们可以使用同步代码块的方式对需要同步的代码进行包裹。

示例如下：

```
public class AccountingSync2 implements Runnable {
    static AccountingSync2 instance = new AccountingSync2(); // 饿汉单例模式

    static int i=0;

    @Override
```

```
    public void run() {
        //省略其他耗时操作....
        //使用同步代码块对变量i进行同步操作,锁对象为instance
        synchronized(instance){
            for(int j=0;j<1000000;j++){
                i++;
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1=new Thread(instance);
        Thread t2=new Thread(instance);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
```

输出结果：

```
/**
 * 输出结果：
 * 2000000
 */
```

我们将 synchronized 作用于一个给定的实例对象 instance，即当前实例对象就是锁的对象，当线程进入 synchronized 包裹的代码块时就会要求当前线程持有 instance 实例对象的锁，如果当前有其他线程正持有该对象锁，那么新的线程就必须等待，这样就保证了每次只有一个线程执行 `i++` 操作。

当然除了用 instance 作为对象外，我们还可以使用 this 对象(代表当前实例)或者当前类的 Class 对象作为锁，如下代码：

```
//this,当前实例对象锁
synchronized(this){
    for(int j=0;j<1000000;j++){
        i++;
    }
}
//Class对象锁
synchronized(AccountingSync.class){
    for(int j=0;j<1000000;j++){
        i++;
    }
}
```

synchronized禁止指令重排

指令重排我们前面讲 JMM 的时候讲过， 这里我们再结合 synchronized 关键字来讲一下。

看下面这段代码：

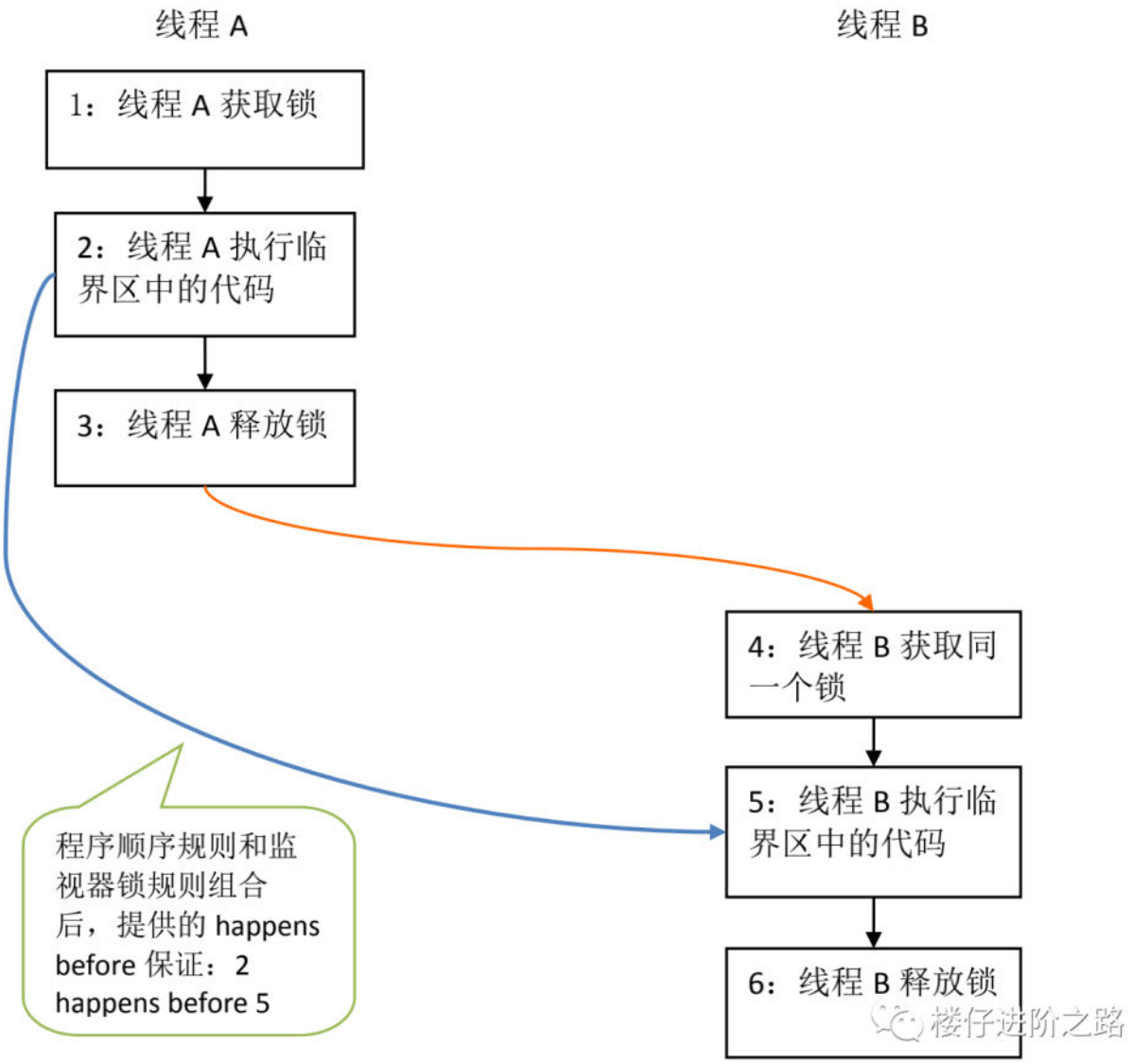
```
class MonitorExample {
    int a = 0;
    public synchronized void writer() {  //1
        a++;                             //2
    }                                    //3
    public synchronized void reader() {  //4
        int i = a;                       //5
        //……
    }                                    //6
}
```

假设线程 A 执行 `writer()` 方法，随后线程 B 执行 `reader()` 方法。根据 happens before 规则，这个过程包含的 happens before 关系可以分为：

- 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
- 根据监视器锁规则，3 happens before 4。
- 根据 happens before 的传递性，2 happens before 5。

> 在 Java 内存模型中，监视器锁规则是一种 happens-before 规则，它规定了对一个监视器锁（monitor lock）或者叫做互斥锁的解锁操作 happens-before 于随后对这个锁的加锁操作。简单来说，这意味着在一个线程释放某个锁之后，另一个线程获得同一把锁的时候，前一个线程在释放锁时所做的所有修改对后一个线程都是可见的。

上述 happens before 关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个 happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的 happens before 保证。

上图表示在线程 A 释放了锁之后，随后线程 B 获取同一个锁。在上图中，2 happens before 5。因此，线程 A 在释放锁之前所有可见的共享变量，在线程 B 获取同一个锁之后，将立刻变得对 B 线程可见。

synchronized属于可重入锁

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。

synchronized 就是可重入锁，因此一个线程调用 synchronized 方法的同时，在其方法体内部调用该对象另一个 synchronized 方法是允许的，如下：

```
public class AccountingSync implements Runnable{
    static AccountingSync instance=new AccountingSync();
    static int i=0;
    static int j=0;

    @Override
```

```
    public void run() {
        for(int j=0;j<1000000;j++){
            //this,当前实例对象锁
            synchronized(this){
                i++;
                increase();//synchronized的可重入性
            }
        }
    }

    public synchronized void increase(){
        j++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1=new Thread(instance);
        Thread t2=new Thread(instance);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
```

1、AccountingSync 类中定义了一个静态的 AccountingSync 实例 instance 和两个静态的整数 i 和 j，静态变量被所有的对象所共享。

2、在 run 方法中，使用了 `synchronized(this)` 来加锁。这里的锁对象是 this，即当前的 AccountingSync 实例。在锁定的代码块中，对静态变量 i 进行增加，并调用了 increase 方法。

3、increase 方法是一个同步方法，它会对 j 进行增加。由于 increase 方法也是同步的，所以它能在已经获取到锁的情况下被 run 方法调用，这就是 synchronized 关键字的可重入性。

4、在 main 方法中，创建了两个线程 t1 和 t2，它们共享同一个 Runnable 对象，也就是共享同一个 AccountingSync 实例。然后启动这两个线程，并使用 join 方法等待它们都执行完成后，打印 i 的值。

此程序中的 `synchronized(this)` 和 synchronized 方法都使用了同一个锁对象（当前的 AccountingSync 实例），并且对静态变量 i 和 j 进行了增加操作，因此，在多线程环境下，也能保证 i 和 j 的操作是线程安全的。

小节

“好了，三妹，今天就学到这吧。”好不容易讲完了，我长吁一口气，扶了扶眼镜对三妹说。

记住 synchronized 的三种应用方式，指令重排情况分析，以及 synchronized 的可重入性，通过今天的学习，你基本可以掌握 synchronized 的使用姿势了。

同步会带来一定的性能开销，因此需要合理使用。不应将整个方法或者更大范围的代码块做同步，而应尽可能地缩小同步范围。

在 JVM 的早期版本中，synchronized 是重量级的，因为线程阻塞和唤醒需要操作系统的介入。但在 JVM 的后续版本中，对 synchronized 进行了大量优化，如偏向锁、轻量级锁和适应性自旋等，所以现在的 synchronized 并不一定是重量级的，其性能在许多情况下都很好，可以大胆地用。

> 编辑：沉默王二，编辑前的内容主要来自于二哥的技术派团队成员楼仔，原文链接戳：volatile。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第一版 PDF 终于来了！包括Java基础语法、数组&字符串、OOP、集合框架、Java IO、异常处理、Java 新特性、网络编程、NIO、并发编程、JVM等等，共计 32 万余字，可以说是通俗易懂、风趣幽默……详情戳：太赞了，GitHub 上标星 9300+ 的 Java 教程

微信搜 **沉默王二** 或扫描下方二维码关注二哥的原创公众号沉默王二，回复 **222** 即可免费领取。

第十节：synchronized的四种锁状态

前面一节我们讲了 synchronized 关键字的基本使用，它能用来同步方法和代码块，那 synchronized 到底锁的是什么呢？随着 JDK 版本的升级，synchronized 又做出了哪些改变呢？“synchronized 性能很差”的谣言真的存在吗？

我想这是很多小伙伴感兴趣的。

首先需要明确的一点是：**Java 多线程的锁都是基于对象的**，Java 中的每一个对象都可以作为一个锁。

还有一点需要注意的是，我们常听到的**类锁**其实也是对象锁，上一节我们也讲到了，应该有不少小伙伴注意到了。

这里再多说几句吧。Class 对象是一种特殊的 Java 对象，代表了程序中的类和接口。Java 中的每个类型（包括类、接口、数组以及基础类型）在 JVM 中都有一个唯一的 Class 对象与之对应。这个 Class 对象被创建的时机是在 JVM 加载类时，由 JVM 自动完成。

Class 对象中包含了与类相关的很多信息，如类的名称、类的父类、类实现的接口、类的构造方法、类的方法、类的字段等等。这些信息通常被称为元数据（metadata）。

可以通过 Class 对象来获取类的元数据，甚至动态地创建类的实例、调用类的方法、访问类的字段等。这就是Java 的反射（Reflection）机制。

所以我们常说的类锁，其实就是 Class 对象的锁。

锁的基本用法

`synchronized` 翻译成中文就是“同步”的意思。

我们通常使用`synchronized`关键字来给一段代码或一个方法上锁，我们上一节已经讲过了，这里简单回顾一下，因为 synchronized 真的非常重要，面试常问，开发常用。它通常有以下三种形式：

```
// 关键字在实例方法上，锁为当前实例
public synchronized void instanceLock() {
    // code
}

// 关键字在静态方法上，锁为当前Class对象
public static synchronized void classLock() {
    // code
}

// 关键字在代码块上，锁为括号里面的对象
public void blockLock() {
    Object o = new Object();
    synchronized (o) {
        // code
    }
}
```

这里介绍一下“临界区”的概念。所谓“临界区”，指的是某一块代码区域，它同一时刻只能由一个线程执行。在上面的例子中，如果`synchronized`关键字在方法上，那临界区就是整个方法内部。而如果是 synchronized 代码块，那临界区就指的是代码块内部的区域。

通过上面的例子我们可以看到，下面这两个写法其实是等价的作用：

```
// 关键字在实例方法上，锁为当前实例
public synchronized void instanceLock() {
    // code
}

// 关键字在代码块上，锁为括号里面的对象
public void blockLock() {
    synchronized (this) {
        // code
    }
}
```

同理，下面这两个方法也应该是等价的：

```
// 关键字在静态方法上，锁为当前Class对象
public static synchronized void classLock() {
    // code
}

// 关键字在代码块上，锁为括号里面的对象
public void blockLock() {
    synchronized (this.getClass()) {
        // code
    }
}
```

锁的四种状态及锁降级

在 JDK 1.6 以前，所有的锁都是”重量级“锁，因为使用的是操作系统的互斥锁，当一个线程持有锁时，其他试图进入synchronized块的线程将被阻塞，直到锁被释放。涉及到了线程上下文切换和用户态与内核态的切换，因此效率较低。

这也是为什么很多开发者会认为 synchronized 性能很差的原因。

那为了减少获得锁和释放锁带来的性能消耗，JDK 1.6 引入了“偏向锁”和“轻量级锁” 的概念，对 synchronized 做了一次重大的升级，升级后的 synchronized 性能可以说上了一个新台阶。

在 JDK 1.6 及其以后，一个对象其实有四种锁状态，它们级别由低到高依次是：

1. 无锁状态
2. 偏向锁状态
3. 轻量级锁状态
4. 重量级锁状态

无锁就是没有对资源进行锁定，任何线程都可以尝试去修改它，很好理解。

几种锁会随着竞争情况逐渐升级，锁的升级很容易发生，但是锁降级发生的条件就比较苛刻了，锁降级发生在 Stop The World（Java 垃圾回收中的一个重要概念，JVM 篇会细讲）期间，当 JVM 进入安全点的时候，会检查是否有闲置的锁，然后进行降级。

关于锁降级有一点需要说明：

不同于大部分文章说的锁不能降级，实际上 HotSpot JVM 是支持锁降级的，这篇帖子里有一个很关键的论述，帖子是 R 大给出的。

> In its current implementation, monitor deflation is performed during every STW pause, while all Java threads are waiting at a safepoint. We have seen safepoint cleanup stalls up to 200ms on monitor-heavy-applications。

大致的意思就是重量级锁降级发生于 STW（Stop The World）阶段，降级对象为仅仅能被 VMThread 访问而没有其他 JavaThread 访问的对象。

各种锁的优缺点对比（来自《Java 并发编程的艺术》）：

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗 CPU。	追求响应时间。同步块执行速度非常快。
重量级锁	线程竞争不使用自旋，不会消耗 CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。同步块执行时间较长。

对象的锁放在什么地方

前面我们提到，Java 的锁都是基于对象的。

首先我们来看看一个对象的“锁”是存放在什么地方的。

每个 Java 对象都有一个对象头。如果是非数组类型，则用 2 个字宽来存储对象头，如果是数组，则会用 3 个字宽来存储对象头。在 32 位处理器中，一个字宽是 32 位；在 64 位虚拟机中，一个字宽是 64 位。对象头的内容如下表所示：

长度	内容	说明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/64bit	Array length	数组的长度（如果是数组）

我们主要来看看 Mark Word 的格式：

锁状态	29 bit 或 61 bit	1 bit 是否是偏向锁?	2 bit 锁标志位
无锁		0	01
偏向锁	线程 ID	1	01
轻量级锁	指向栈中锁记录的指针	此时这一位不用于标识偏向锁	00
重量级锁	指向互斥量（重量级锁）的指针	此时这一位不用于标识偏向锁	10
GC 标记		此时这一位不用于标识偏向锁	11

可以看到，当对象状态为偏向锁时，`Mark Word`存储的是偏向的线程 ID；当状态为轻量级锁时，`Mark Word`存储的是指向线程栈中`Lock Record`的指针；当状态为重量级锁时，`Mark Word`为指向堆中的 monitor（监视器）对象的指针。

> 在 Java 中，监视器（monitor）是一种同步工具，用于保护共享数据，避免多线程并发访问导致数据不一致。在 Java 中，每个对象都有一个内置的监视器。

监视器包括两个重要部分，一个是锁，一个是等待/通知机制，后者是通过 Object 类中的`wait()`，`notify()`，`notifyAll()`等方法实现的（我们会在讲Condition和生产者-消费者模式）详细地讲。

下面分别介绍这几种锁以及它们之间是如何升级的。

偏向锁

Hotspot 的作者经过以往的研究发现大多数情况下**锁不仅不存在多线程竞争，而且总是由同一线程多次获得**，于是引入了偏向锁。

偏向锁会偏向于第一个访问锁的线程，如果在接下来的运行过程中，该锁没有被其他的线程访问，则持有偏向锁的线程将永远不需要触发同步。也就是说，**偏向锁在资源无竞争情况下消除了同步语句**，连 CAS（后面会细讲，戳链接直达） 操作都不做了，着极大地提高了程序的运行性能。

大白话就是对锁设置个变量，如果发现为 true，代表资源无竞争，则无需再走各种加锁/解锁流程。如果为 false，代表存在其他线程竞争资源，那么就会走后面的流程。

偏向锁的实现原理

一个线程在第一次进入同步块时，会在对象头和栈帧中的锁记录里存储锁偏向的线程 ID。当下次该线程进入这个同步块时，会去检查锁的 Mark Word 里面是不是放的自己的线程 ID。

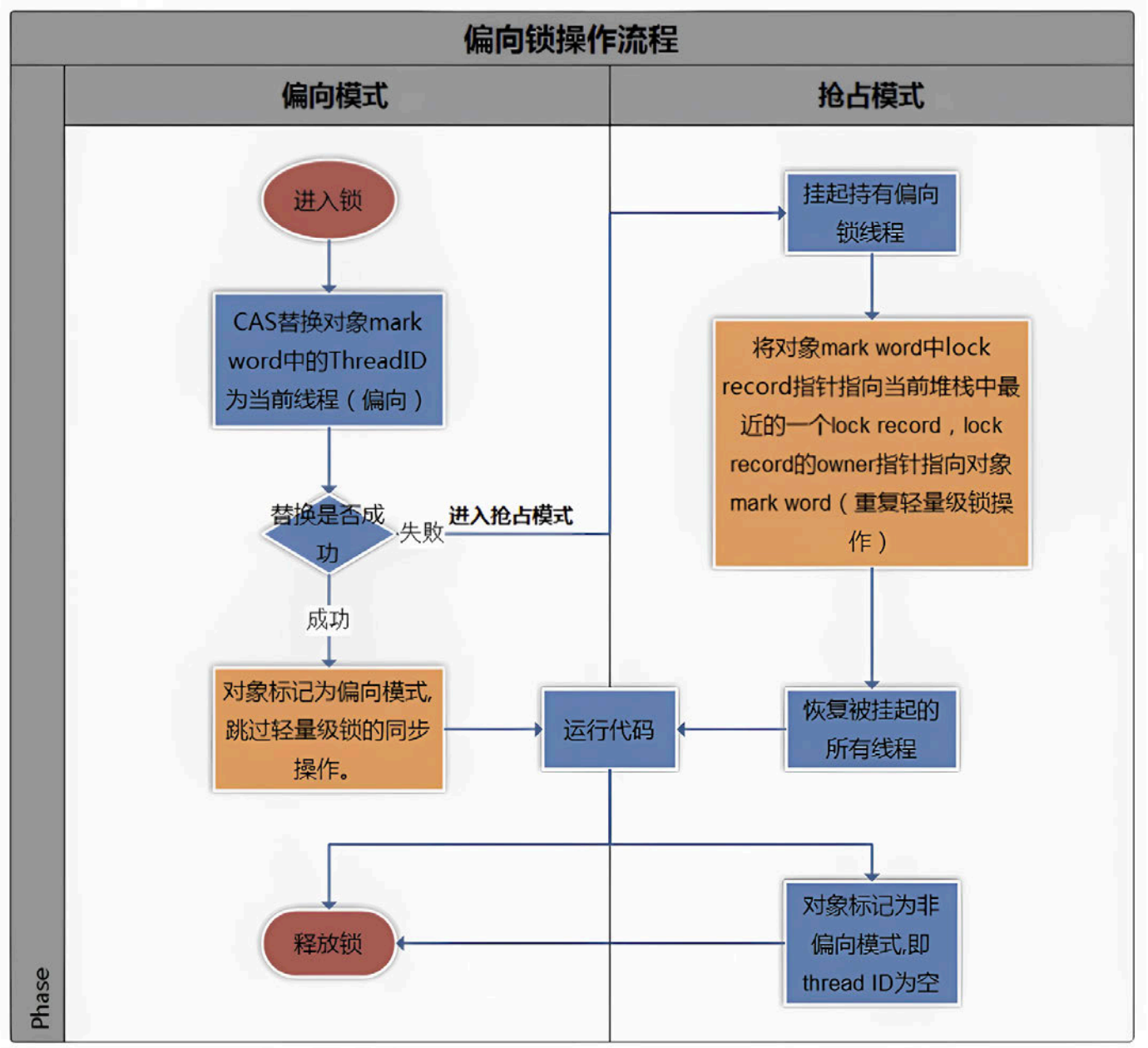
如果是，表明该线程已经获得了锁，以后该线程在进入和退出同步块时不需要花费 CAS 操作来加锁和解锁；如果不是，就代表有另一个线程来竞争这个偏向锁。这个时候会尝试使用 CAS 来替换 Mark Word 里面的线程 ID 为新线程的 ID，这个时候要分两种情况：

- 成功，表示之前的线程不存在了， Mark Word 里面的线程 ID 为新线程的 ID，锁不会升级，仍然为偏向锁；
- 失败，表示之前的线程仍然存在，那么暂停之前的线程，设置偏向锁标识为 0，并设置锁标志位为 00，升级为轻量级锁，会按照轻量级锁的方式进行竞争锁。

CAS: Compare and Swap 会在后面细讲，可戳链接直达，这里简单提一嘴。

CAS 是比较并设置的意思，用于在硬件层面上提供原子性操作。在 在某些处理器架构（如x86）中，比较并交换通过指令 CMPXCHG 实现（（Compare and Exchange），一种原子指令），通过比较是否和给定的数值一致，如果一致则修改，不一致则不修改。

线程竞争偏向锁的过程如下：



图中涉及到了 lock record 指针指向当前堆栈中的最近一个 lock record，是轻量级锁按照先来先服务的模式进行了轻量级锁的加锁。

撤销偏向锁

偏向锁使用了一种**等到竞争出现才释放锁的机制**，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。

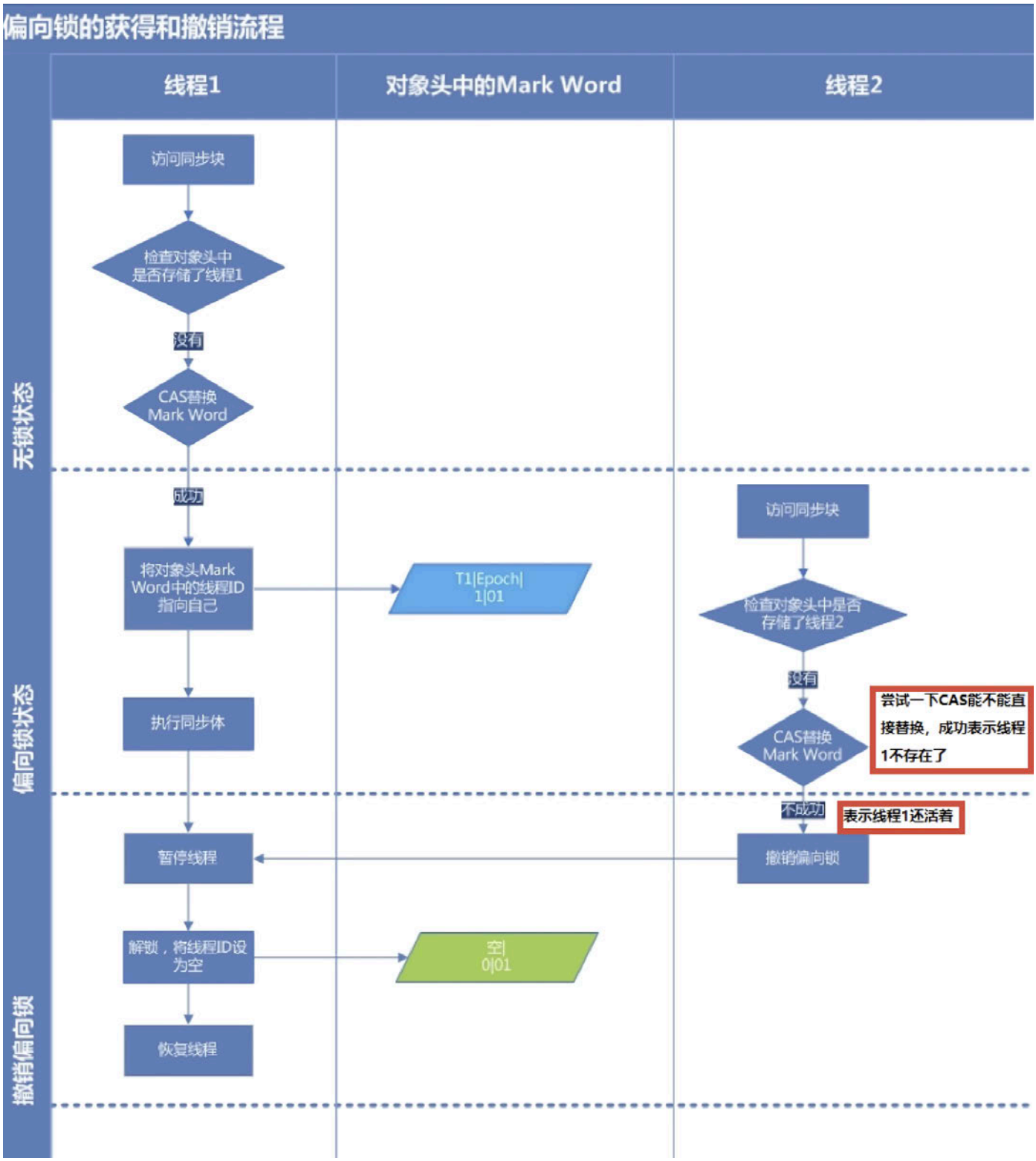
偏向锁升级成轻量级锁时，会暂停拥有偏向锁的线程，重置偏向锁标识，这个过程看起来容易，实则开销还是很大的，大概的过程如下：

1. 在一个安全点（在这个时间点上没有字节码正在执行）停止拥有锁的线程。
2. 遍历线程栈，如果存在锁记录的话，需要修复锁记录和 Mark Word，使其变成无锁状态。
3. 唤醒被停止的线程，将当前锁升级成轻量级锁。

所以，如果应用程序里所有的锁通常处于竞争状态，那么偏向锁就会是一种累赘，对于这种情况，我们可以一开始就把偏向锁这个默认功能给关闭：

```
-XX:UseBiasedLocking=false
```

下面这个经典的图总结了偏向锁的获得和撤销：



轻量级锁

多个线程在不同时段获取同一把锁，即不存在锁竞争的情况，也就没有线程阻塞。针对这种情况，JVM 采用轻量级锁来避免线程的阻塞与唤醒。

JVM 会为每个线程在当前线程的栈帧中创建用于存储锁记录的空间，我们称为 Displaced Mark Word。如果一个线程获得锁的时候发现是轻量级锁，会把锁的 Mark Word 复制到自己的 Displaced Mark Word 里面。

然后线程尝试用 CAS 将锁的 Mark Word 替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示 Mark Word 已经被替换成了其他线程的锁记录，说明在与其它线程竞争锁，当前线程就尝试使用自旋来获取锁。

> 自旋：不断尝试去获取锁，一般用循环来实现。

自旋是需要消耗 CPU 的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费 CPU 资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环 10 次，如果还没获取到锁就进入阻塞状态。

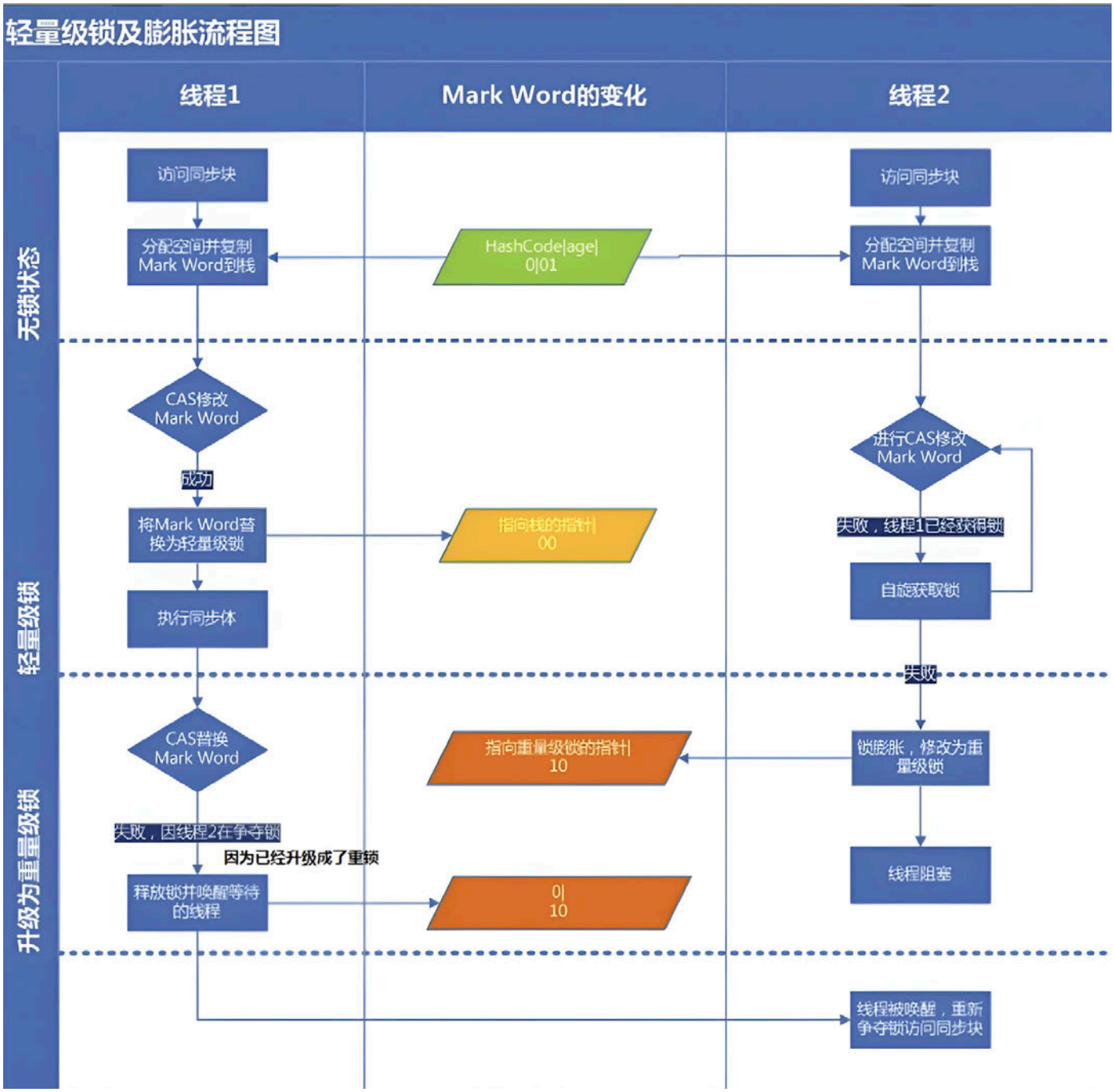
但是 JDK 采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

自旋也不是一直进行下去的，如果自旋到一定程度（和 JVM、操作系统相关），依然没有获取到锁，称为自旋失败，那么这个线程会阻塞。同时这个锁就会**升级成重量级锁**。

轻量级锁的释放

在释放锁时，当前线程会使用 CAS 操作将 Displaced Mark Word 的内容复制回锁的 Mark Word 里面。如果没有发生竞争，那么这个复制的操作会成功。如果有其他线程因为自旋多次导致轻量级锁升级成了重量级锁，那么 CAS 操作会失败，此时会释放锁并唤醒被阻塞的线程。

一张图说明加锁和释放锁的过程：



重量级锁

重量级锁依赖于操作系统的互斥锁（mutex，用于保证任何给定时间内，只有一个线程可以执行某一段特定的代码段） 实现，而操作系统中线程间状态的转换需要相对较长的时间，所以重量级锁效率很低，但被阻塞的线程不会消耗 CPU。

前面说到，每一个对象都可以当做一个锁，当多个线程同时请求某个对象锁时，对象锁会设置几种状态用来区分请求的线程：

- Contention List：所有请求锁的线程将被首先放置到该竞争队列
- Entry List：Contention List 中那些有资格成为候选人的线程被移到 Entry List
- Wait Set：那些调用 wait 方法被阻塞的线程被放置到 Wait Set
- OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为 OnDeck
- Owner：获得锁的线程称为 Owner
- !Owner：释放锁的线程

当一个线程尝试获得锁时，如果该锁已经被占用，则会将该线程封装成一个`ObjectWaiter`对象插入到 Contention List 队列的队首，然后调用`park`方法挂起当前线程。

当线程释放锁时，会从 Contention List 或 EntryList 中挑选一个线程唤醒，被选中的线程叫做`Heir presumptive`即假定继承人，假定继承人被唤醒后会尝试获得锁，但`synchronized`是非公平的，所以假定继承人不一定能获得锁。

这是因为对于重量级锁，线程需要先自旋尝试获得锁，这样做的目的是为了减少执行操作系统同步操作带来的开销。如果自旋不成功再进入等待队列。这对那些已经在等待队列中的线程来说，稍微显得不公平，还有一个不公平的地方是自旋线程可能会抢占了 Ready 线程的锁。

如果线程获得锁后调用`Object.wait`方法，则会将线程加入到 WaitSet 中，当被`Object.notify`唤醒后，会将线程从 WaitSet 移动到 Contention List 或 EntryList 中去。需要注意的是，当调用一个锁对象的`wait`或`notify`方法时，**如当前锁的状态是偏向锁或轻量级锁则会先膨胀成重量级锁**。

锁的升级流程

每一个线程在准备获取共享资源时：
第一步，检查 MarkWord 里面是不是放的自己的 ThreadId ,如果是，表示当前线程是处于 “偏向锁” 。

第二步，如果 MarkWord 不是自己的 ThreadId，锁升级，这时候，用 CAS 来执行切换，新的线程根据 MarkWord 里面现有的 ThreadId，通知之前线程暂停，之前线程将 Markword 的内容置为空。

第三步，两个线程都把锁对象的 HashCode 复制到自己新建的用于存储锁的记录空间，接着开始通过 CAS 操作，把锁对象的 MarKword 的内容修改为自己新建的记录空间的地址的方式竞争 MarkWord。

第四步，第三步中成功执行 CAS 的获得资源，失败的则进入自旋 。

第五步，自旋的线程在自旋过程中，成功获得资源(即之前获的资源的线程执行完成并释放了共享资源)，则整个状态依然处于 轻量级锁的状态，如果自旋失败 。

第六步，进入重量级锁的状态，这个时候，自旋的线程进行阻塞，等待之前线程执行完成并唤醒自己。

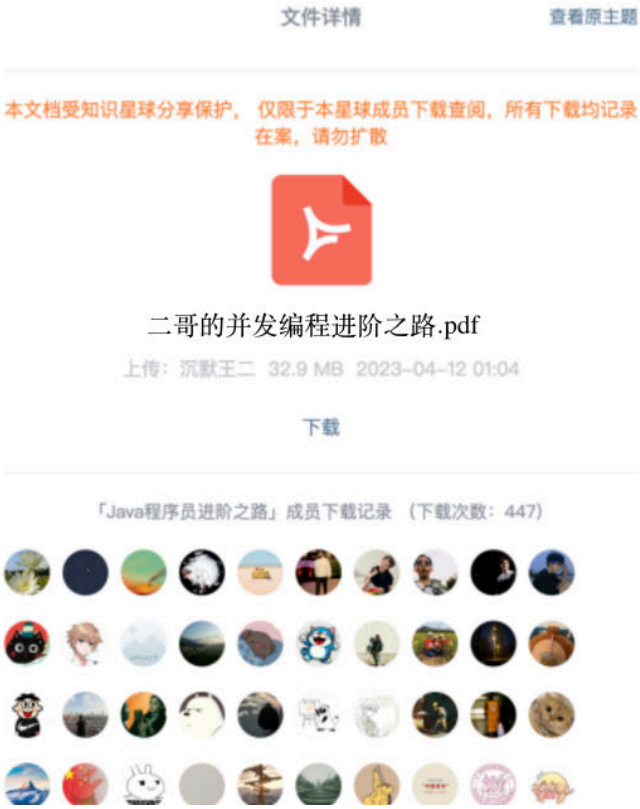
小结

- Java 中的每一个对象都可以作为一个锁，Java 中的锁都是基于对象的。
- synchronized 关键字可以用来修饰方法和代码块，它可以保证在同一时刻最多只有一个线程执行该段代码。
- synchronized 关键字在修饰方法时，锁为当前实例对象；在修饰静态方法时，锁为当前 Class 对象；在修饰代码块时，锁为括号里面的对象。
- Java 6 为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁“。在 Java 6 以前，所有的锁都是”重量级“锁。所以在 Java 6 及其以后，一个对象其实有四种锁状态，它们级别由低到高依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。
- 偏向锁会偏向于第一个访问锁的线程，如果在接下来的运行过程中，该锁没有被其他的线程访问，则持有偏向锁的线程将永远不需要触发同步。也就是说，偏向锁在资源无竞争情况下消除了同步语句，连 CAS 操作都不做了，提高了程序的运行性能。
- 轻量级锁是通过 CAS 操作和自旋来实现的，如果自旋失败，则会升级为重量级锁。
- 重量级锁依赖于操作系统的互斥量（mutex） 实现的，而操作系统中线程间状态的转换需要相对较长的时间，所以重量级锁效率很低，但被阻塞的线程不会消耗 CPU。

> 编辑：沉默王二，原文内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。



第十一节：深入浅出偏向锁

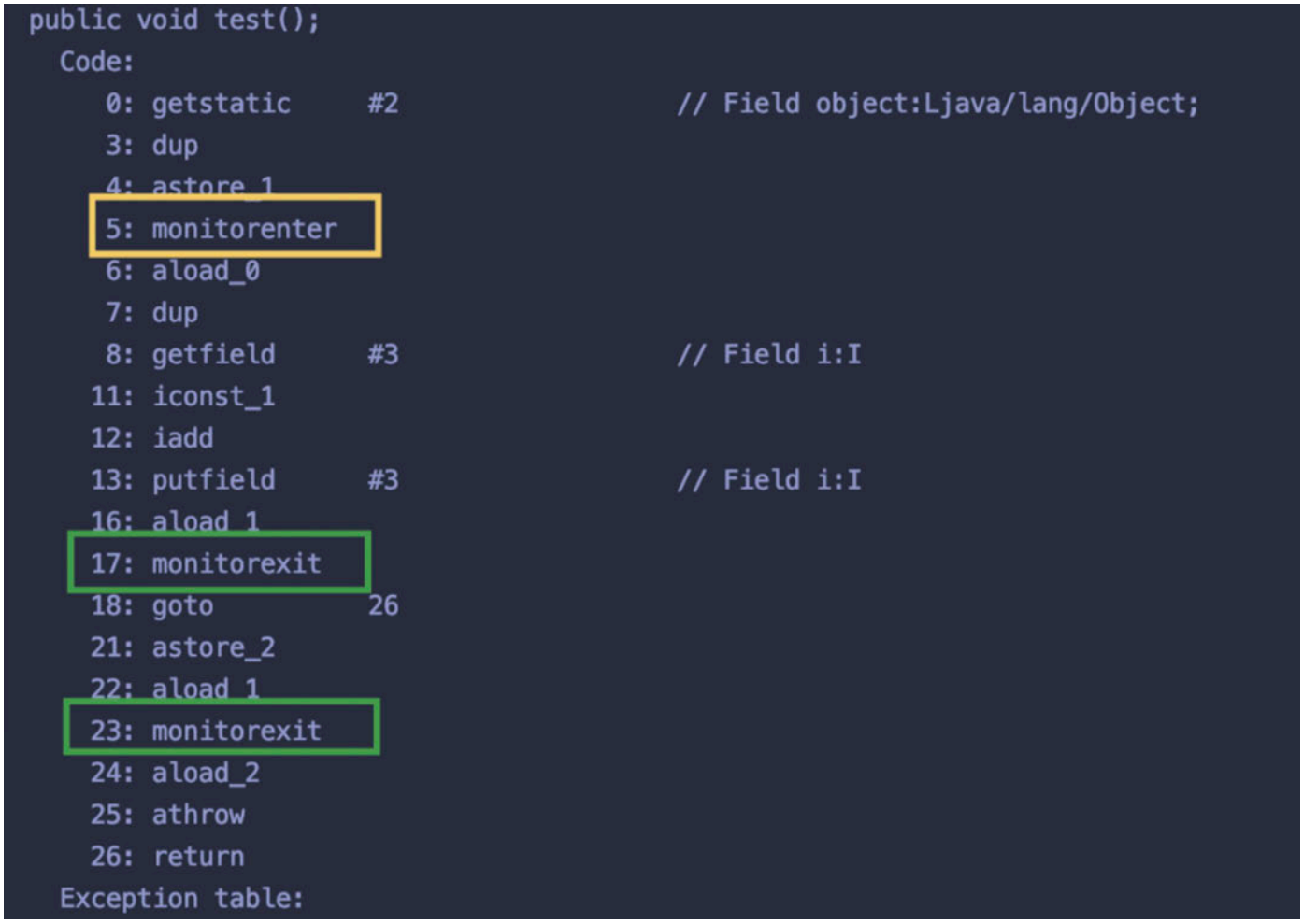
在 JDK 1.5 之前，面对 Java 并发问题， synchronized 是一招鲜的解决方案：

1. 同步方法，锁上当前实例对象
2. 同步静态方法，锁上当前类的 Class 对象
3. 同步块，锁上代码块里面配置的对象

拿同步块来举例:

```
public void test(){
  synchronized (object) {
    i++;
  }
}
```

经过 `javap -v` 编译后的指令如下：



- `monitorenter` 指令在编译后会插入到同步代码块的开始位置；
- `monitorexit` 指令会插入到方法结束和异常的位置（实际隐藏了try-finally）。
- 每个对象都有一个 monitor 与之关联，当一个线程执行到 monitorenter 指令时，就会获得对象所对应 `monitor` 的所有权，也就获得到了对象的锁。

对象监视器 monitor

这里再简单说一下 monitor 的概念。

在 Java 中，monitor 可以被看作是一种守门人或保安，它确保同一时刻只有一个线程可以访问受保护的代码段。你可以将它想象成一个房间的门，门的里面有一些重要的东西，而 monitor 就是那个保护门的保安。

这里是 monitor 的工作方式：

- 进入房间: 当一个线程想要进入受保护的代码区域（房间）时，它必须得到 monitor 的允许。如果房间里没有其他线程，monitor 会让它进入并关闭门。
- 等待其他线程: 如果房间里已经有一个线程，其他线程就必须等待。monitor 会让其他线程排队等候，直到房间里的线程完成工作离开房间。
- 离开房间: 当线程完成它的工作并离开受保护的代码区域时，monitor 会重新打开门，并让等待队列中的下一个线程进入。
- 协调线程: monitor 还可以通过一些特殊的机制（例如 wait 和 notify 方法，讲 Condtion 的时候会细讲）来协调线程之间的合作。线程可以通过 monitor 来发送信号告诉其他线程现在可以执行某些操作了。

重量级锁

当另外一个线程执行到同步块的时候，由于它没有对应 `monitor` 的所有权，就会被阻塞，此时控制权只能交给操作系统，也就会从 `user mode` 切换到 `kernel mode`（在讲 JMM 的时候有提到，戳链接直达），由操作系统来负责线程间的调度和线程的状态变更，这就需要频繁的在这两个模式下切换（**上下文转换**）。

有点竞争就找内核的行为很不好，会引起很大的开销，所以大家都叫它**重量级锁**，自然效率也很低，这也就给很多小伙伴留下了一个根深蒂固的印象 —— synchronized 关键字相比于其他同步机制性能不好，但其实不然，我们前面也讲过了。

轻量级锁

如果 CPU 通过 CAS（后面会细讲，戳链接直达）就能处理好加锁/释放锁，这样就不会有上下文的切换。

但是当竞争很激烈，CAS 尝试再多也是浪费 CPU，权衡一下，不如升级成重量级锁，阻塞线程排队竞争，也就有了轻量级锁升级成重量级锁的过程。

HotSpot 的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由**同一个线程**多次获得，同一个线程反复获取锁，如果还按照 CAS 的方式获取锁，也是有一定代价的，如何让这个代价更小一些呢?

偏向锁

偏向锁实际上就是「锁对象」潜意识「偏向」同一个线程来访问，让锁对象记住这个线程 ID，当线程再次获取锁时，亮出身份，如果是同一个 ID 直接获取锁就好了，是一种 `load-and-test` 的过程，相较 CAS 又轻量级了一些。

可是多线程环境，也不可能只有同一个线程一直获取这个锁，其他线程也是要干活的，如果出现多个线程竞争的情况，就会有偏向锁升级的过程。

这里可以思考一下：**偏向锁可以绕过轻量级锁，直接升级到重量级锁吗?**

都是同一个锁对象，却有多种锁状态，其目的显而易见：

> 占用的资源越少，程序执行的速度越快。

偏向锁和轻量级锁，都不会调用系统互斥量（Mutex Lock），它们只是为了提升性能多出来的两种锁状态，这样可以在不同场景下采取最合适的策略：

- 偏向锁：无竞争的情况下，只有一个线程进入临界区，采用偏向锁
- 轻量级锁：多个线程可以交替进入临界区，采用轻量级锁
- 重量级锁：多线程同时进入临界区，交给操作系统互斥量来处理

关于这部分内容，我们在讲 进击的synchronized也曾讲过，但是这部分内容的确又很需要多花点时间去搞透彻，所以我们这里就从不同的角度切入，多花点时间来盘一下。

到这里，大家应该理解了，但仍然会有很多疑问：

1. 锁对象是在哪存储线程 ID 的?
2. 整个升级过程是如何过渡的?

想理解这些问题，就需要先知道 Java 对象头的结构。

Java 对象头

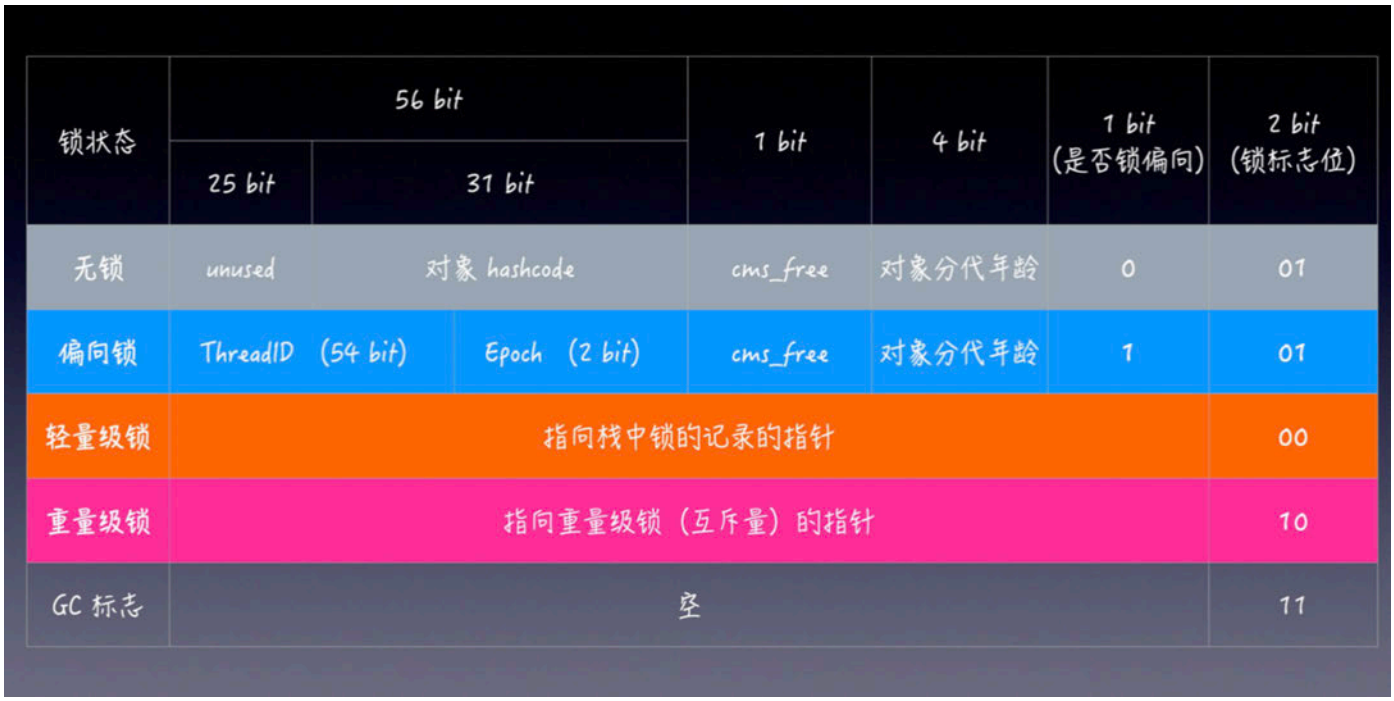
按照常规理解，识别线程 ID 需要一组 mapping 映射关系来搞定，如果单独维护这个 mapping 关系又要考虑线程安全的问题。根据奥卡姆剃刀原理，Java 万物皆是对象，对象皆可用作锁，与其单独维护一个 mapping 关系，不如中心化将锁的信息维护在 Java 对象本身上。

> 奥卡姆剃刀原理是一种问题解决原则，简单来说就是：在解释某事物时，没有必要假设更多的东西，当有多个解释时，应选择假设最少、最简单的那个解释。

Java 对象头最多由三部分构成：

1. MarkWord
2. ClassMetadata Address
3. Array Length （**如果对象是数组才会有这部分**）

其中 `Markword` 是保存锁状态的关键，对象锁状态可以从偏向锁升级到轻量级锁，再升级到重量级锁，加上初始的无锁状态，可以理解为有 4 种状态。想在一个对象中表示这么多信息自然就要用`位`来存储，在 64 位操作系统中，是这样存储的（**注意颜色标记**），想看具体注释的可以看 hotspot(1.8) 源码文件 `path/hotspot/src/share/vm/oops/markOop.hpp` 第 30 行。



有了这些基本信息，接下来我们就只需要弄清楚，MarkWord 中的锁信息是怎么变化的。

单纯看上图，还十分抽象，作为程序员的我们最喜欢用代码说话，贴心的 openjdk 官网提供了可以查看对象内存布局的工具 JOL (java object layout)，我们直接通过 Maven 引入到项目中。

Maven Package

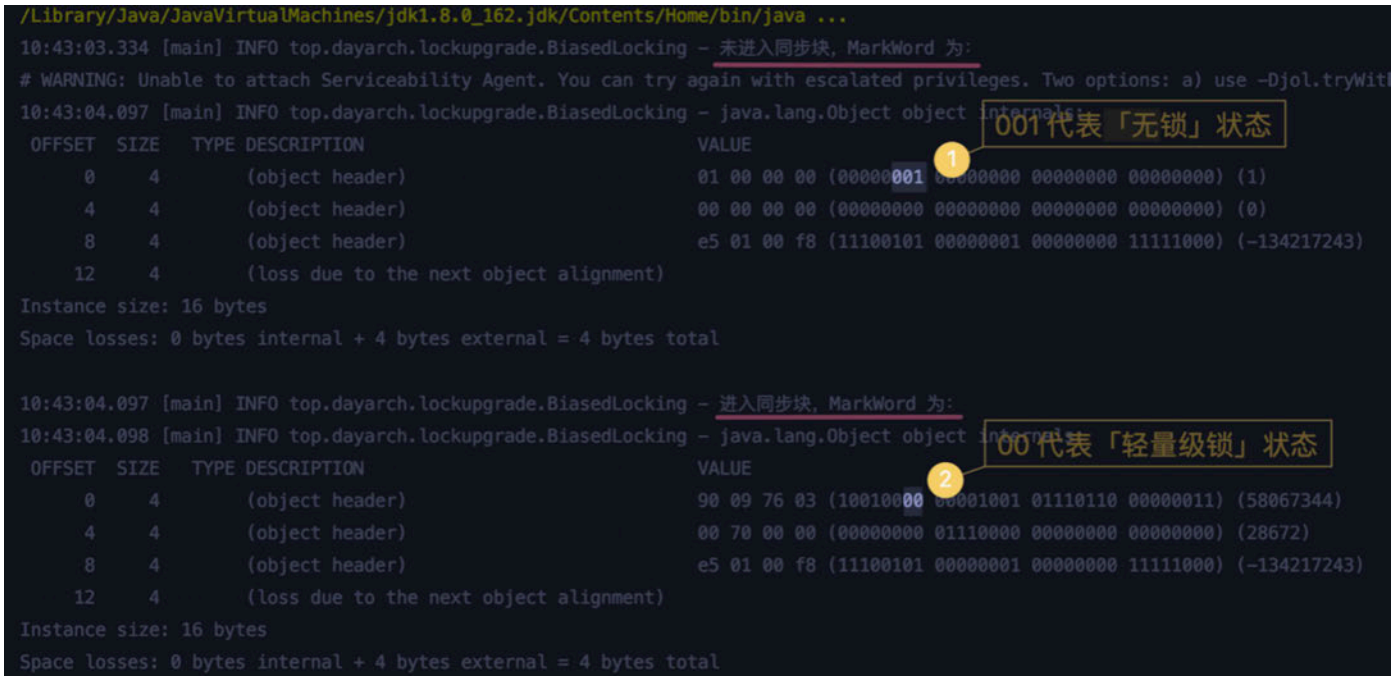
```
<dependency>
  <groupId>org.openjdk.jol</groupId>
  <artifactId>jol-core</artifactId>
  <version>0.14</version>
</dependency>
```

接下来我们就通过代码来深入了解一下偏向锁。

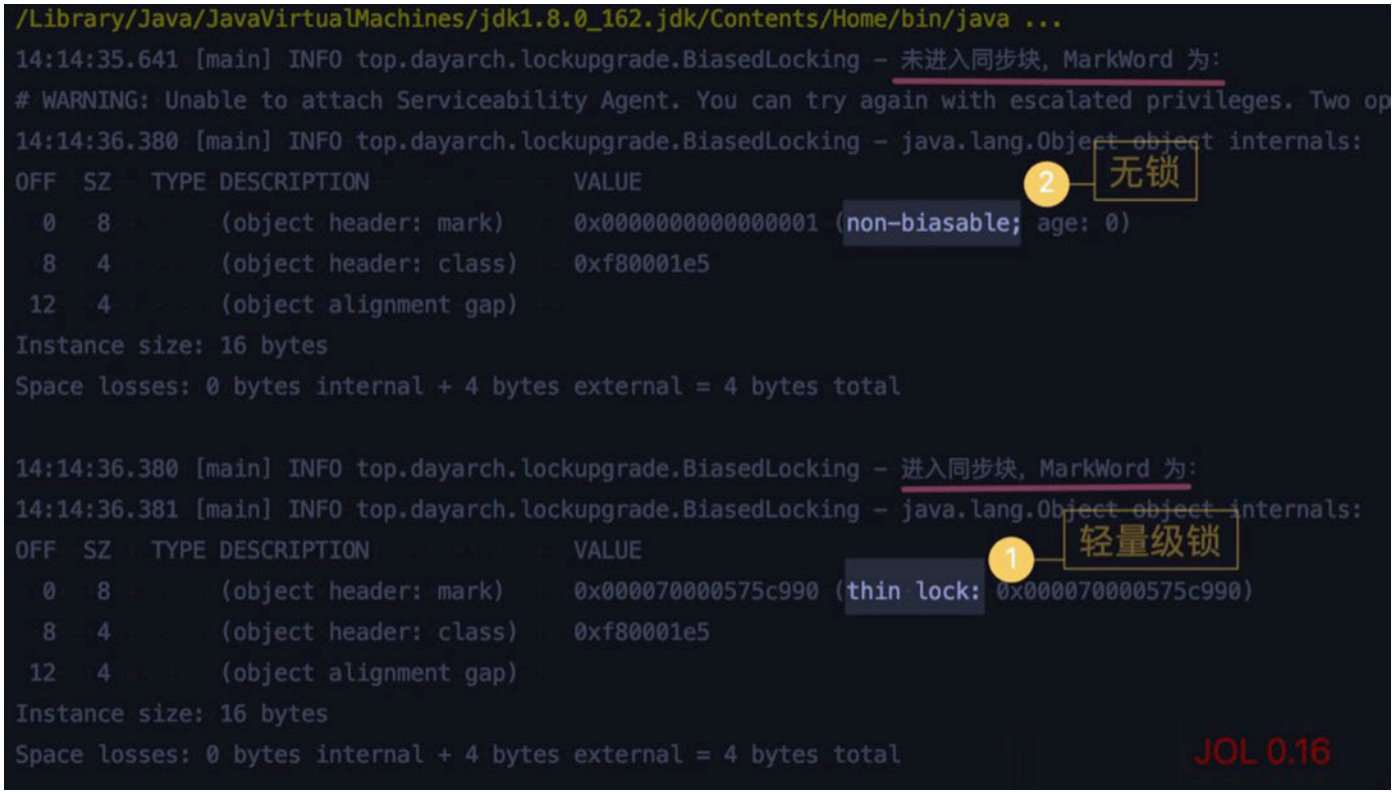
场景 1

```
public static void main(String[] args) {
    Object o = new Object();
    log.info("未进入同步块，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());
    synchronized (o){
        log.info(("进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

来看输出结果：



上面我们用到的 JOL 版本为 `0.14`，接下来我们要用 `0.16` 版本查看输出结果，因为这个版本给了我们更友好的说明，同样的代码，来看输出结果：



看到这个结果，有些小伙伴会有疑问，JDK 1.6 之后默认是开启偏向锁的，为什么初始化的代码是无锁状态，进入同步块产生竞争就绕过偏向锁直接变成轻量级锁了呢?

> 虽然默认开启了偏向锁，但是开启**有延迟**，大概 4s。原因是 JVM 内部的代码有很多地方用到了 synchronized，如果直接开启偏向，产生竞争就要有锁升级，会带来额外的性能损耗，所以就有了延迟策略。

```
void BiasedLocking::init() {
  // If biased locking is enabled, schedule a task to fire a few
  // seconds into the run which turns on biased locking for all
  // currently loaded classes as well as future ones. This is a
  // workaround for startup time regressions due to a large number of
  // safepoints being taken during VM startup for bias revocation.
  // Ideally we would have a lower cost for individual bias revocation
  // and not need a mechanism like this.
  if (UseBiasedLocking) {
    if (BiasedLockingStartupDelay > 0) {
      EnableBiasedLockingTask* task = new EnableBiasedLockingTask(BiasedLockingStartupDelay);
      task->enroll();
    } else {
      VM_EnableBiasedLocking op(false);
      VMThread::execute(&op);
    }
  }
}
```

biasedLocking.cpp

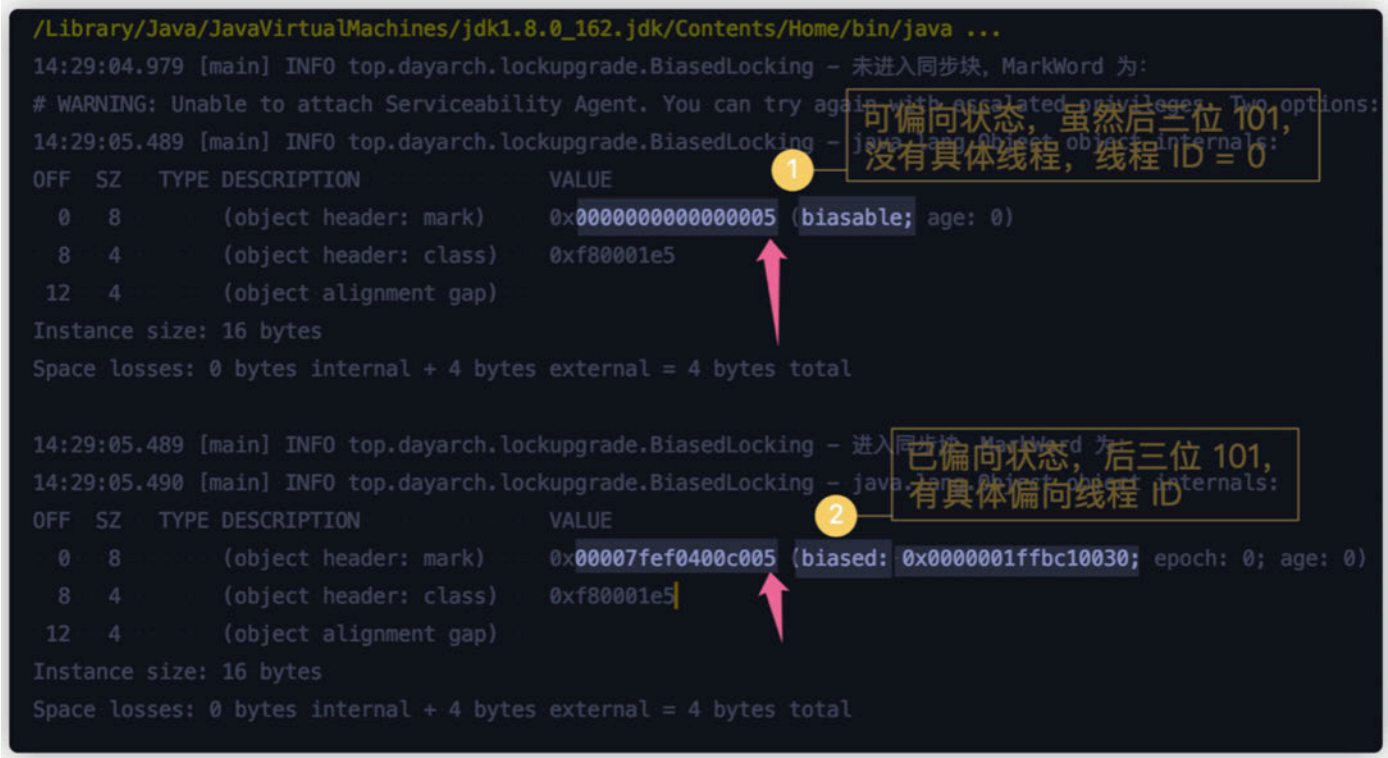
可以通过参数 `-XX:BiasedLockingStartupDelay=0` 将延迟改为 0，但是**不建议**这么做。

场景 2

那我们就代码延迟 5 秒来创建对象，来看看偏向是否生效

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);
    Object o = new Object();
    log.info("未进入同步块，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());
    synchronized (o){
        log.info(("进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

重新查看运行结果：



这样的结果是符合我们预期的，但是结果中的 `biasable` 状态，在 MarkWord 表格中并不存在，其实这是一种**匿名偏向状态**，是对象初始化中，JVM 帮我们做的。这样当有线程进入同步块时：

1. 可偏向状态：直接就 CAS 替换 ThreadID，如果成功，就可以获取偏向锁了
2. 不可偏向状态：就会变成轻量级锁

那问题又来了，现在锁对象有具体偏向的线程，如果新的线程过来执行同步块会偏向新的线程吗？

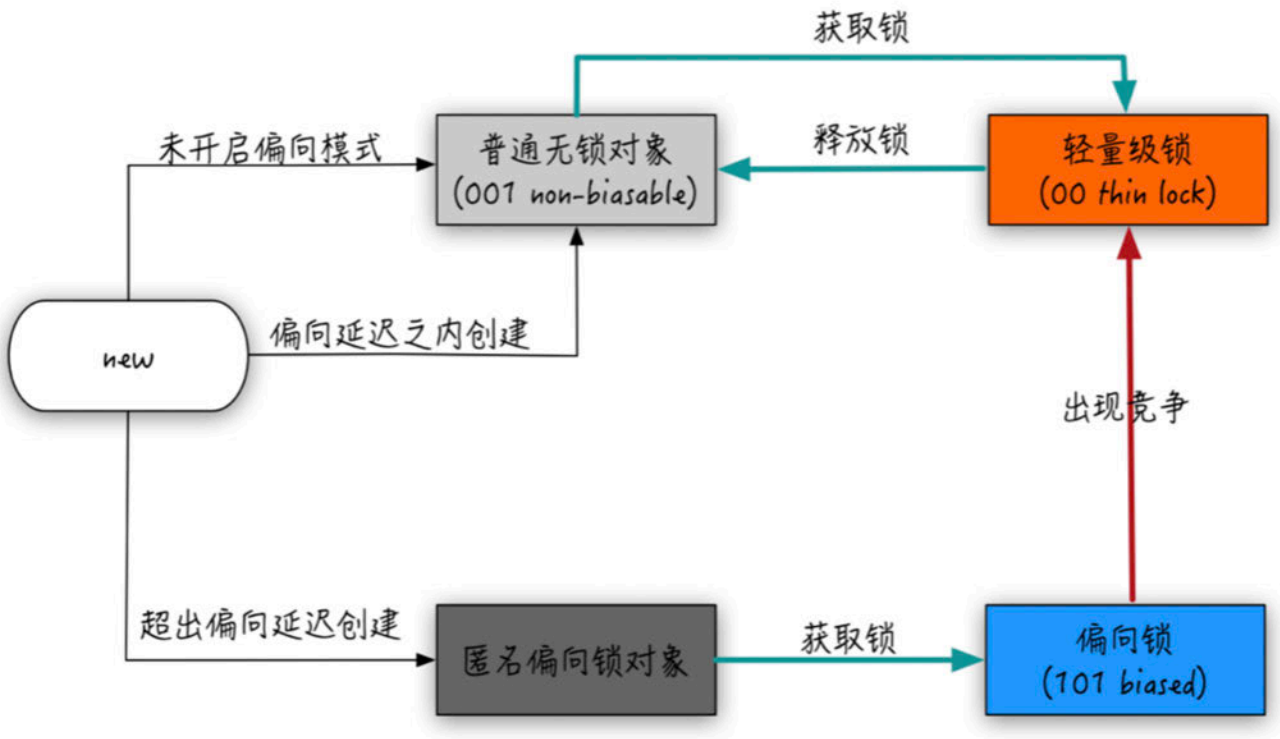
场景 3

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);
    Object o = new Object();
    log.info("未进入同步块，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());
    synchronized (o){
        log.info(("进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }

    Thread t2 = new Thread(() -> {
        synchronized (o) {
            log.info("新线程获取锁，MarkWord为：");
            log.info(ClassLayout.parseInstance(o).toPrintable());
        }
    });

    t2.start();
    t2.join();
    log.info("主线程再次查看锁对象，MarkWord为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());

    synchronized (o){
        log.info(("主线程再次进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

来看运行结果，奇怪的事情发生了：

- 标记1：初始可偏向状态
- 标记2：偏向主线程后，主线程退出同步代码块
- 标记3：**新线程**进入同步代码块，升级成了轻量级锁
- 标记4：新线程的轻量级锁退出同步代码块，主线程查看，变为不可偏向状态
- 标记5：由于对象不可偏向，同**场景 1**主线程再次进入同步块，自然就会用轻量级锁

至此，场景一二三可以总结为一张图：



从这样的运行结果上来看，偏向锁像是“**一锤子买卖**”，只要偏向了某个线程，后续其他线程尝试获取锁，都会变为轻量级锁，这样的偏向非常局限。**事实上并不是这样**，如果你仔细看标记 2（已偏向状态），还有个 epoch 我们没有提及，这个值就是打破这种局限性的关键，在了解 epoch 之前，我们还要了解一个概念——偏向撤销（后面在讲批量撤销的时候会细讲这个陌生的 epoch）。

偏向撤销

在讲偏向撤销之前，需要和大家明确一个概念——偏向锁撤销和偏向锁释放是两码事：

1. 撤销：笼统的说，就是多个线程竞争导致不能再使用偏向模式，主要是告知这个锁对象不能再用偏向模式
2. 释放：和你的常规理解一样，对应的就是 synchronized 方法的退出或 synchronized 块的结束

何为偏向撤销?

> 从偏向状态撤回到原来的状态，也就是将 MarkWord 的第 3 位（是否偏向撤销）的值，`从 1 变回 0`

如果只是一个线程获取锁，再加上「偏心」的机制，是没有理由撤销偏向的，**所以偏向的撤销只能发生在有竞争的情况下**。

想要撤销偏向锁，还不能对持有偏向锁的线程有影响，就要等待持有偏向锁的线程到达一个 `safepoint 安全点` (这里的安全点是 JVM 为了保证在垃圾回收的过程中引用关系不会发生变化设置的一种安全状态，在这个状态上会暂停所有线程工作)， 在这个安全点会挂起获得偏向锁的线程，后续讲 JVM 的时候会详细讲。

在这个安全点，线程可能还是处在不同的状态，先说结论（因为源码就是这么写的）

1. 线程不存活，或者活着的线程退出了同步块，很简单，直接撤销偏向就好了
2. 活着的线程但仍在同步块之内，那就升级成轻量级锁

这个和 epoch 貌似还是没啥关系，因为这还不是全部场景。

偏向锁是特定场景下提升程序效率的方案，可并不代表所有程序都满足这些特定场景，比如这些场景（在开启偏向锁的前提下）：

1. 一个线程创建了大量对象并执行了初始的同步操作，之后在另一个线程中将这些对象作为锁进行之后的操作。这种 case 下，会导致大量的偏向锁撤销操作
2. 明知有多线程竞争（生产者/消费者队列），还要使用偏向锁，也会导致各种撤销

很显然，这两种场景肯定会导致偏向撤销的，一个偏向撤销的成本无所谓，大量偏向撤销的成本是不能忽视的。那怎么办?

既不想禁用偏向锁，还不想忍受大量撤销偏向增加的成本，这种方案就是设计一个**有阶梯的底线**。

批量重偏向（bulk rebias）

这是第一种场景的快速解决方案，以 class 为单位，为每个 class 维护一个偏向锁撤销计数器，只要 class 的对象发生偏向撤销，该计数器 `+1`，当这个值达到重偏向阈值（默认 20）时：

```
BiasedLockingBulkRebiasThreshold = 20
```

JVM 就认为该 class 的偏向锁有问题，因此会进行批量重偏向, 它的实现方式就用到了我们上面说的 `epoch`。

`Epoch`，如其含义「纪元」一样，就是一个时间戳。每个 class 对象会有一个对应的`epoch`字段，每个**处于偏向锁状态对象**的`mark word`中也有该字段，其初始值为创建该对象时 class 中的`epoch`的值（此时二者是相等的）。

每次发生批量重偏向时，就将该值加 1，同时遍历 JVM 中所有线程的栈：

1. 找到该 class 所有**正处于加锁状态**的偏向锁对象，将其`epoch`字段改为新值
2. class 中**不处于加锁状态**的偏向锁对象（没被任何线程持有，但之前是被线程持有过的，这种锁对象的 markword 肯定也是有偏向的），保持 `epoch` 字段值不变

这样下次获得锁时，发现当前对象的`epoch`值和 class 的`epoch`，本着**今朝不问前朝事** 的原则（上一个纪元），就算当前已经偏向了其他线程，也不会执行撤销操作，而是直接通过 CAS 操作将其`mark word`的线程 ID 改成当前线程 ID，这也算是一定程度的优化，毕竟没升级锁；

如果 `epoch` 都一样，说明没有发生过批量重偏向, 如果 `markword` 有线程 ID，还有其他锁来竞争，那锁自然是要升级的(如同前面举的例子 epoch=0)。

批量重偏向是第一阶梯底线，还有第二阶梯底线

批量撤销（bulk revoke）

当达到重偏向阈值后，假设该 class 计数器继续增长，当其达到批量撤销的阈值后（默认 40）时，

```
BiasedLockingBulkRevokeThreshold = 40
```

JVM 就认为该 class 的使用场景存在多线程竞争，会标记该 class 为不可偏向。之后对于该 class 的锁，直接走轻量级锁的逻辑。

这就是第二阶梯底线，但是在第一阶梯到第二阶梯的过渡过程中，也就是在彻底禁用偏向锁之前，还会给一次改过自新的机会，那就是另外一个计时器：

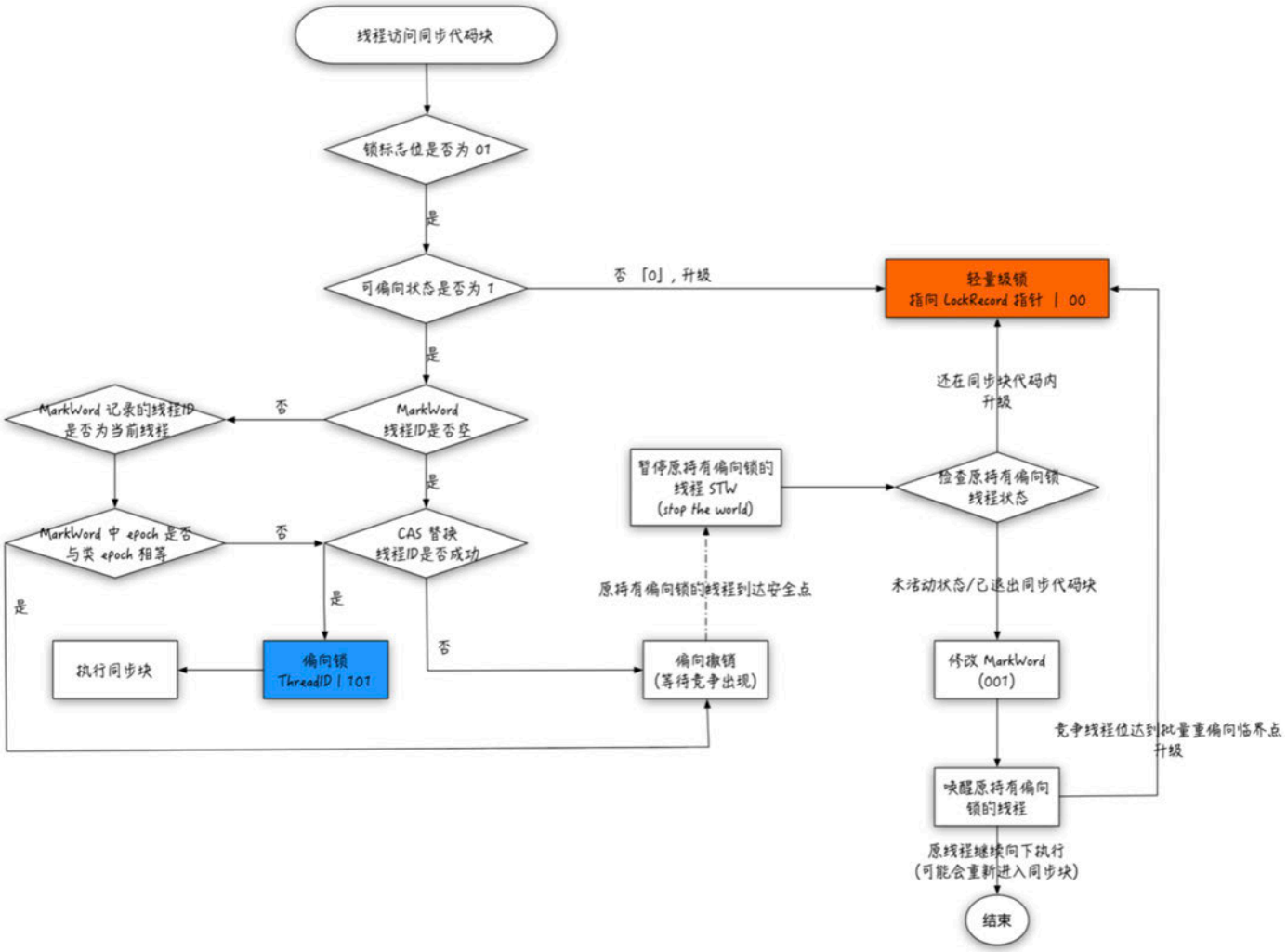
```
BiasedLockingDecayTime = 25000
```

1. 如果在距离上次批量重偏向发生的 25 秒之内，并且累计撤销计数达到 40，就会发生批量撤销（偏向锁彻底

game over）

2. 如果在距离上次批量重偏向发生超过 25 秒之外，就会重置在 `[20, 40)` 内的计数，再给次机会

大家有兴趣可以写代码测试一下临界点，观察锁对象 `markword` 的变化。至此，整个偏向锁的工作流程可以用一张图表示：



到此，你应该对偏向锁有个基本的认识了。

偏向锁与 HashCode

上面场景一，无锁状态，对象头中没有 hashcode；偏向锁状态，对象头还是没有 hashcode，那我们的 hashcode 哪去了？

首先要知道，hashcode 不是创建对象就帮我们写到对象头中的，而是要经过**第一次**调用 `Object::hashCode()` 或者 `System::identityHashCode(Object)` 才会存储在对象头中的。

第一次**生成 hashcode 后，该值应该是一直保持不变的**，但偏向锁又是来回更改锁对象的 markword，必定会对 hashcode 的生成有影响，那怎么办呢？我们来用代码验证：

场景一

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);

    Object o = new Object();
    log.info("未生成 hashcode，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());

    o.hashCode();
    log.info("已生成 hashcode，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());

    synchronized (o){
        log.info(("进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

来看运行结果

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_162.jdk/Contents/Home/bin/java ...
15:45:17.251 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - 未生成 hashcode，MarkWord 为：
# WARNING: Unable to attach Serviceability Agent. You can try again with escalated privileges. Two options: a) use
15:45:17.771 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - java.lang.Object object internals:
OFF  SZ   TYPE DESCRIPTION               VALUE
  0   8        (object header: mark)     0x0000000000000005 (biasable; age: 0)
  8   4        (object header: class)    0xf80001e5
 12   4        (object alignment gap)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

15:45:17.771 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - 已生成 hashcode，MarkWord 为：
15:45:17.771 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - java.lang.Object object internals:
OFF  SZ   TYPE DESCRIPTION               VALUE
  0   8        (object header: mark)     0x00000023223dd801 (hash: 0x23223dd8; age: 0)
  8   4        (object header: class)    0xf80001e5
 12   4        (object alignment gap)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

15:45:17.771 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - 进入同步块，MarkWord 为：
15:45:17.772 [main] INFO top.dayarch.lockupgrade.BIasedLockingAndHashCode - java.lang.Object object internals:
OFF  SZ   TYPE DESCRIPTION               VALUE
  0   8        (object header: mark)     0x0000700007961990 (thin lock: 0x0000700007961990)
  8   4        (object header: class)    0xf80001e5
 12   4        (object alignment gap)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

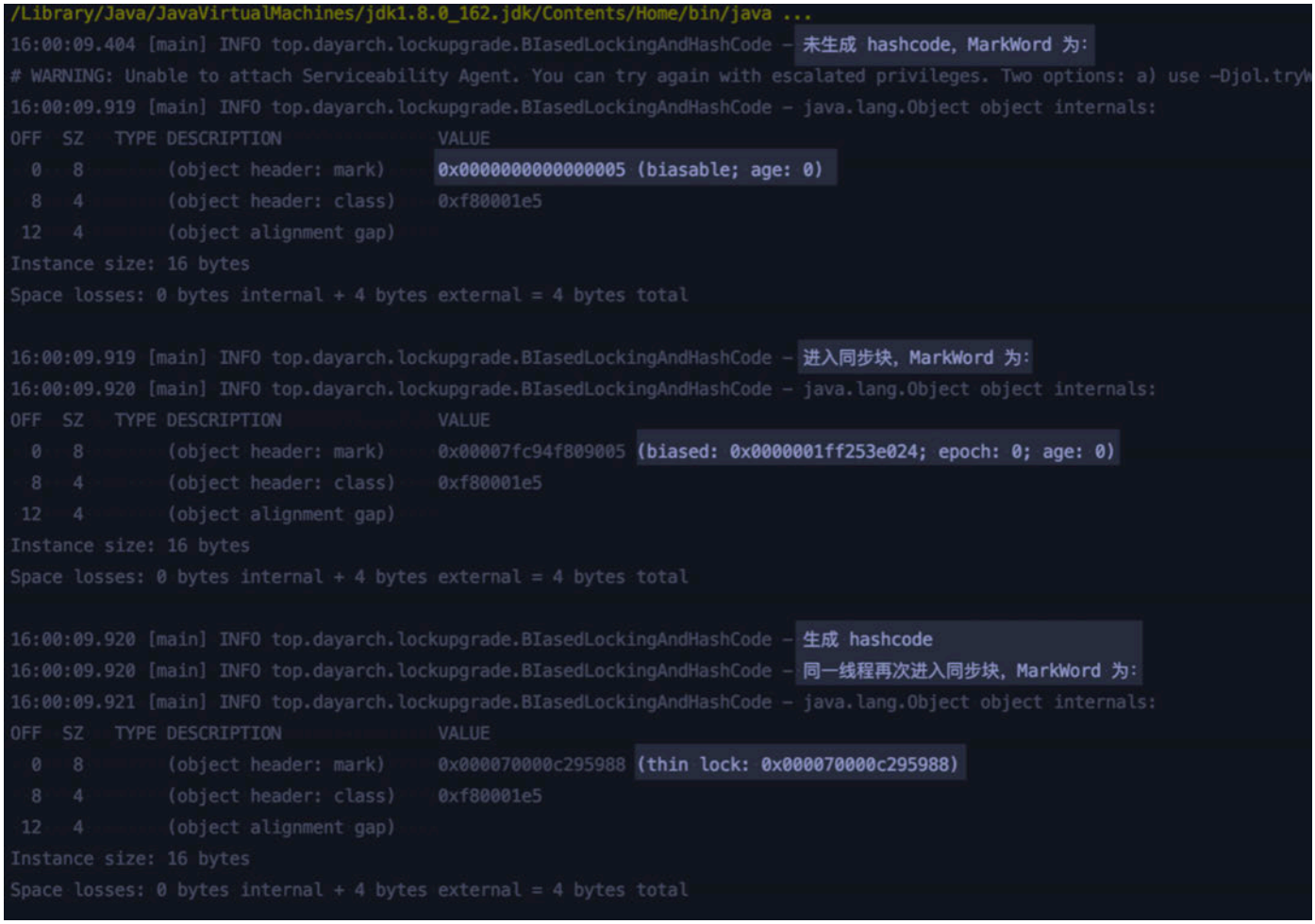
结论就是：即便初始化为可偏向状态的对象，一旦调用 `Object::hashCode()` 或者 `System::identityHashCode(Object)`，进入同步块就会直接使用轻量级锁。

场景二

假如已偏向某一个线程，然后生成了 hashcode，然后同一个线程又进入同步块，会发生什么呢？来看代码：

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);

    Object o = new Object();
    log.info("未生成 hashcode，MarkWord 为：");
    log.info(ClassLayout.parseInstance(o).toPrintable());

    synchronized (o){
        log.info(("进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }

    o.hashCode();
    log.info("生成 hashcode");
    synchronized (o){
        log.info(("同一线程再次进入同步块，MarkWord 为："));
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

查看运行结果：



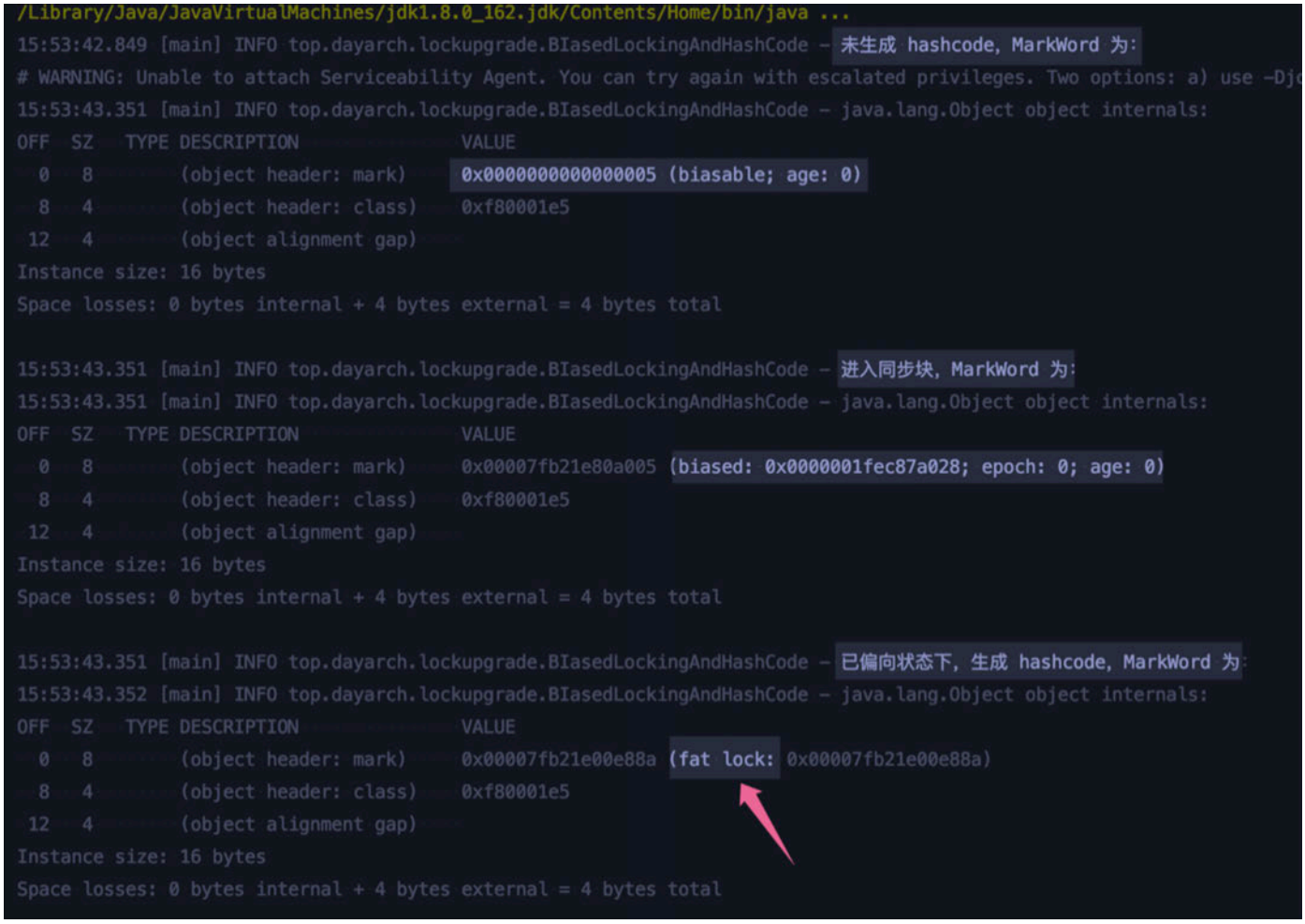
结论就是：同场景一，会直接使用轻量级锁。

场景三

那假如对象处于已偏向状态，在同步块中调用了那两个方法会发生什么呢？继续代码验证：

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);

    Object o = new Object();
    log.info("未生成 hashcode, MarkWord 为: ");
    log.info(ClassLayout.parseInstance(o).toPrintable());

    synchronized (o){
        log.info(("进入同步块, MarkWord 为: "));
        log.info(ClassLayout.parseInstance(o).toPrintable());
        o.hashCode();
        log.info("已偏向状态下, 生成 hashcode, MarkWord 为: ");
        log.info(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

来看运行结果：



结论就是：如果对象处在已偏向状态，生成 hashcode 后，就会直接升级成重量级锁。

最后用书中的一段话来描述锁和 hashcode 之间的关系。

> 在Java语言里面一个对象如果计算过哈希码，就应该一直保持该值不变（强烈推荐但不强制，因为用户可以重载hashCode()方法按自己的意愿返回哈希码），否则很多依赖对象哈希码的API都可能存在出错风险。而作为绝大多数对象哈希码来源的Object::hashCode()方法，返回的是对象的一致性哈希码（Identity Hash Code），这个值是能强制保证不变的，它通过在对象头中存储计算结果来保证第一次计算之后，再次调用该方法取到的哈希码值永远不会再发生改变。因此，当一个对象已经计算过一致性哈希码后，它就再也无法进入偏向锁状态了；而当一个对象当前正处于偏向锁状态，又收到需要计算其一致性哈希码请求[1]时，它的偏向状态会被立即撤销，并且锁会膨胀为重量级锁。在重量级锁的实现中，对象头指向了重量级锁的位置，代表重量级锁的ObjectMonitor类里有字段可以记录非加锁状态（标志位为“01”）下的Mark Word，其中自然可以存储原来的哈希码。

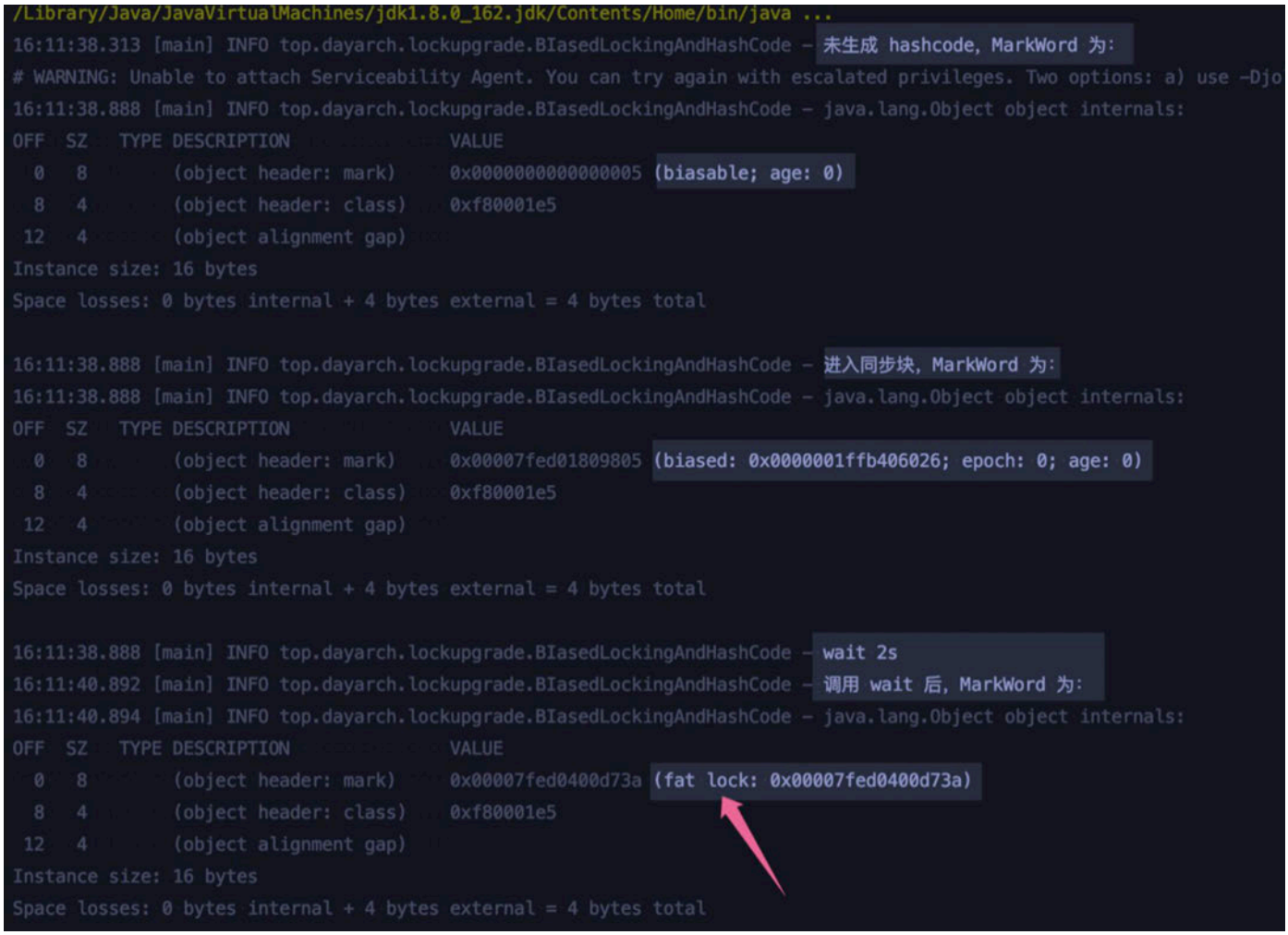
重量级锁和Object.wait

Object 除了提供上述的 hashcode 方法，还有 `wait()` 方法，这也是我们在同步块中常用的，调用 wait 方法会对锁产生哪些影响呢？来看代码：

```
public static void main(String[] args) throws InterruptedException {
    // 睡眠 5s
    Thread.sleep(5000);

    Object o = new Object();
```

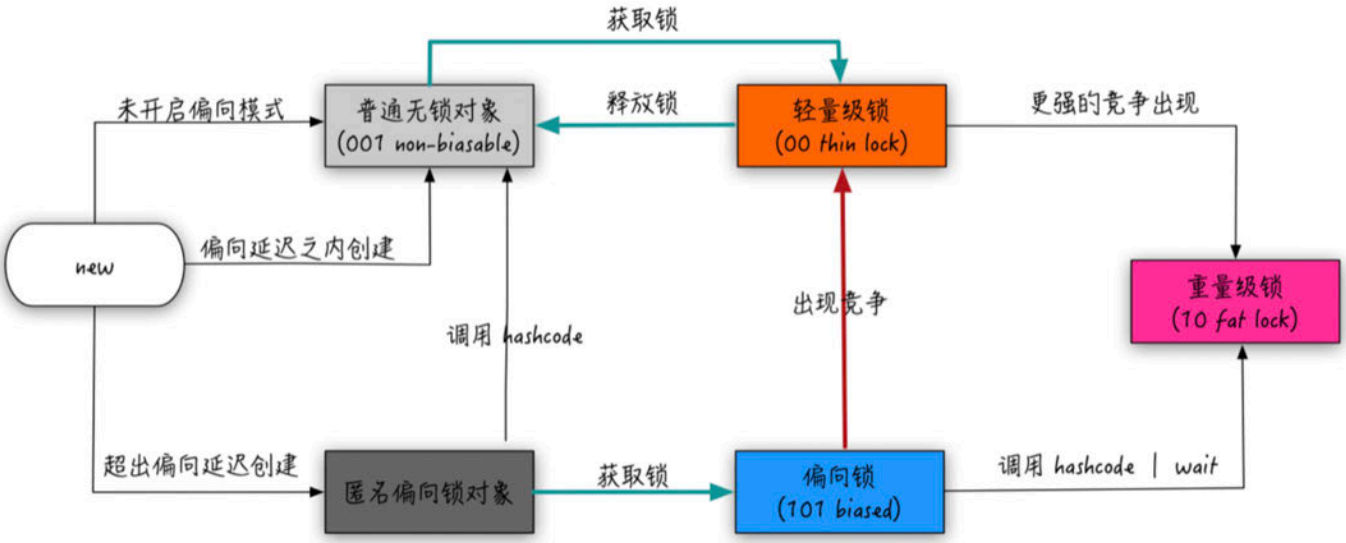
```
        log.info("未生成 hashcode，MarkWord 为：");
        log.info(ClassLayout.parseInstance(o).toPrintable());

        synchronized (o) {
            log.info(("进入同步块，MarkWord 为："));
            log.info(ClassLayout.parseInstance(o).toPrintable());

            log.info("wait 2s");
            o.wait(2000);

            log.info(("调用 wait 后，MarkWord 为："));
            log.info(ClassLayout.parseInstance(o).toPrintable());
        }
    }
```

查看运行结果：



结论就是，wait 方法是互斥量（重量级锁）独有的，一旦调用该方法，就会升级成重量级锁（这个是面试可以说出的亮点内容哦）

最后再继续丰富一下锁对象变化图：



再见偏向锁

看到这个副标题你可能有些慌，为啥要告别偏向锁，因为维护成本有些高了，来看 Open JDK 官方声明，JEP 374: Deprecate and Disable Biased Locking

这个说明的更新时间距离现在很近，在 JDK 15 版本就已经开始了

Goals

Determine the need for continued support of the legacy synchronization optimization of biased locking, which is costly to maintain.

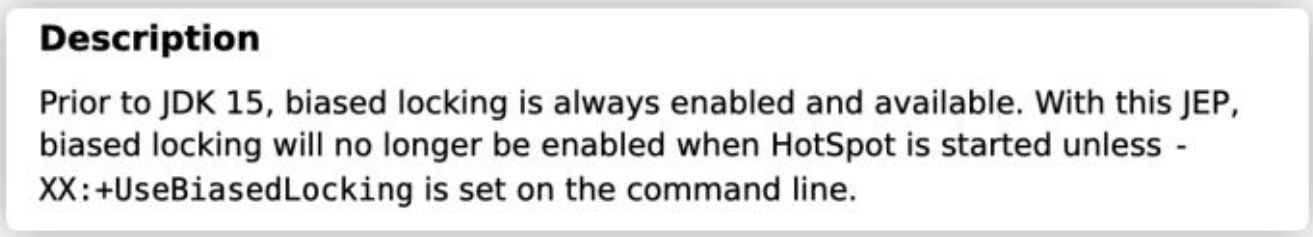
一句话解释就是维护成本太高

Motivation

Biased locking is an optimization technique used in the HotSpot Virtual Machine to reduce the overhead of uncontended locking. It aims to avoid executing a compare-and-swap atomic operation when acquiring a monitor by assuming that a monitor remains owned by a given thread until a different thread tries to acquire it. The initial lock of the monitor *biases* the monitor towards that thread, avoiding the need for atomic instructions in subsequent synchronized operations on the same object. When many threads perform many synchronized operations on objects used in a single-threaded fashion, biasing the locks has historically led to significant performance improvements over regular locking techniques.

The performance gains seen in the past are far less evident today. Many applications that benefited from biased locking are older, legacy applications that use the early Java collection APIs, which synchronize on every access (e.g., `Hashtable` and `Vector`). Newer applications generally use the non-synchronized collections (e.g., `HashMap` and `ArrayList`), introduced in Java 1.2 for single-threaded scenarios, or the even more-performant concurrent data structures, introduced in Java 5, for multi-threaded scenarios. This means that applications that benefit from biased locking due to unnecessary synchronization will likely see a performance improvement if the code is updated to use these newer classes. Furthermore, applications built around a thread-pool queue and worker threads generally perform better with biased locking disabled. (SPECjbb2015 was designed that way, e.g., while SPECjvm98 and SPECjbb2005 were not). Biased locking comes with the cost of requiring an expensive revocation operation in case of contention. Applications that benefit from it are therefore only those that exhibit significant amounts of uncontended synchronized operations, like those mentioned above, so that the cost of executing cheap lock owner checks plus an occasional expensive revocation is still lower than the cost of executing the eluded compare-and-swap atomic instructions. Changes in the cost of atomic instructions since the introduction of biased locking into HotSpot also change the amount of uncontended operations needed for that relation to remain true. Another aspect worth noting is that applications won't have noticeable performance improvements from biased locking even when the previous cost relation is true when the time spend on synchronized operations is still only a small fraction of the total application workload.

Biased locking introduced a lot of complex code into the synchronization subsystem and is invasive to other HotSpot components as well. This complexity is a barrier to understanding various parts of the code and an impediment to making significant design changes within the synchronization subsystem. To that end we would like to disable, deprecate, and eventually remove support for biased locking.



最终就是，JDK 15 之前，偏向锁默认是 enabled，从 JDK 15 开始，默认就是 disabled，除非显示的通过 `UseBiasedLocking 开启`。

其中在 quarkus 上的一篇文章说明的更加直接

偏向锁给 JVM 增加了巨大的复杂性，只有少数非常有经验的程序员才能理解整个过程，维护成本很高，大大阻碍了开发新特性的进程（换个角度理解，你掌握了，是不是就是那少数有经验的程序员了呢？哈哈）

小结

偏向锁可能就这样的走完了它的一生，有些小伙伴可能直接发问，都被 deprecated 了，JDK 都 17 了，还讲这么多干什么？

1. Java 任它发，我用 Java 8，这是很多主流的状态，至少你用的版本没有被 deprecated
2. 面试还是会被经常问到
3. 万一哪天有更好的设计方案，“偏向锁”又以新的形式回来了呢，了解变化才能更好理解背后设计
4. 奥卡姆剃刀原理，我们现实中的优化也一样，如果没有必要不要增加实体，如果增加的内容带来很大的成本，不如大胆的废除掉，接受一点落差

> 编辑：沉默王二，编辑前的内容主要来自于日拱一兵的这篇知乎文章
> https://zhuanlan.zhihu.com/p/451061367

第十二节：乐观锁 CAS

CAS（Compare-and-Swap）是一种乐观锁的实现方式，全称为“比较并交换”，是一种无锁的原子操作。

在并发编程中，我们都知道`i++`操作是非线程安全的，这是因为`i++`操作不是原子操作，我们之前在讲多线程带来了什么问题中有讲到，大家应该还记得吧？

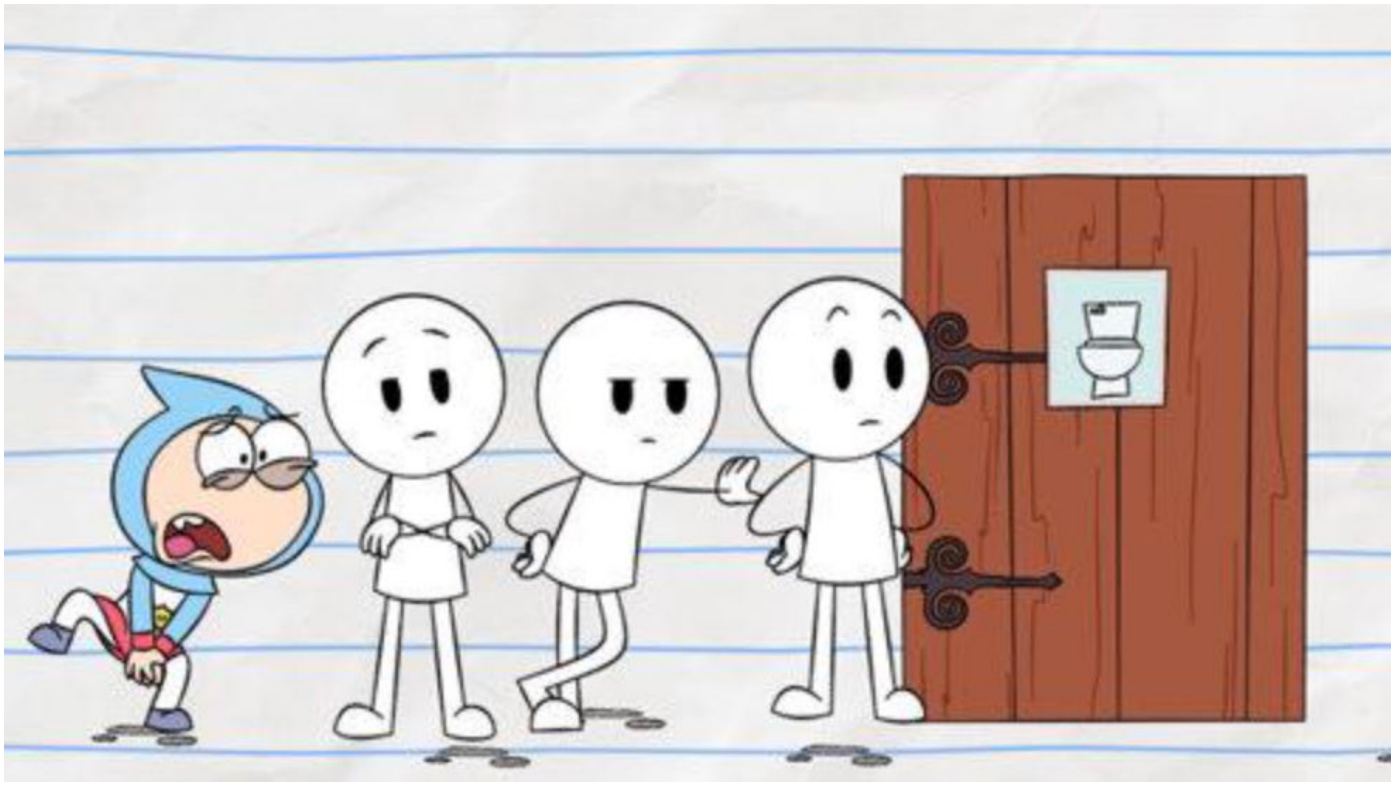
如何保证原子性呢？

常见的做法就是加锁。

在 Java 中，我们可以使用 synchronized关键字 和 `CAS`（Compare-and-Swap）来实现加锁效果。

`synchronized` 是悲观锁，尽管随着 JDK 版本的升级，synchronized 关键字已经“轻量级”了很多（前面有细讲，戳链接回顾），但依然是悲观锁，线程开始执行第一步就要获取锁，一旦获得锁，其他的线程进入后就会阻塞并等待锁。

如果不好理解，我们来举个生活中的例子：一个人进入厕所后首先把门锁上（获取锁），然后开始上厕所，这个时候有其他人来了就只能在外面等（阻塞），就算再急也没用。上完厕所完事后把门打开（解锁），其他人就可以进入了。



`CAS` 是乐观锁，线程执行的时候不会加锁，它会假设此时没有冲突，然后完成某项操作；如果因为冲突失败了就重试，直到成功为止。

乐观锁与悲观锁

锁可以从不同的角度来分类。比如我们在前面讲 synchronized 四种锁状态的时候，提到过偏向锁、轻量级锁、重量级锁，对吧？乐观锁和悲观锁也是一种分类方式。

悲观锁

对于悲观锁来说，它总是认为每次访问共享资源时会发生冲突，所以必须对每次数据操作加上锁，以保证临界区的程序同一时间只能有一个线程在执行。

乐观锁

乐观锁，顾名思义，它是乐观派。乐观锁总是假设对共享资源的访问没有冲突，线程可以不停地执行，无需加锁也无需等待。一旦多个线程发生冲突，乐观锁通常使用一种称为 CAS 的技术来保证线程执行的安全性。

由于乐观锁假想操作中没有锁的存在，因此不太可能出现死锁的情况，换句话说，**乐观锁天生免疫死锁**。

- 乐观锁多用于“读多写少“的环境，避免频繁加锁影响性能；
- 悲观锁多用于”写多读少“的环境，避免频繁失败和重试影响性能。

什么是 CAS

在 CAS 中，有这样三个值：

- V：要更新的变量(var)
- E：预期值(expected)
- N：新值(new)

比较并交换的过程如下：

判断 V 是否等于 E，如果等于，将 V 的值设置为 N；如果不等，说明已经有其它线程更新了 V，于是当前线程放弃更新，什么都不做。

这里的**预期值 E 本质上指的是“旧值”**。

我们以一个简单的例子来解释这个过程：

1. 如果有一个多个线程共享的变量`i`原本等于 5，我现在在线程 A 中，想把它设置为新的值 6;
2. 我们使用 CAS 来做这个事情；
3. 首先我们用 i 去与 5 对比，发现它等于 5，说明没有被其它线程改过，那我就把它设置为新的值 6，此次 CAS 成功，`i`的值被设置成了 6；
4. 如果不等于 5，说明`i`被其它线程改过了（比如现在`i`的值为 2），那么我就什么也不做，此次 CAS 失败，`i`的值仍然为 2。

在这个例子中，`i`就是 V，5 就是 E，6 就是 N。

那有没有可能我在判断了`i`为 5 之后，正准备更新它的新值的时候，被其它线程更改了`i`的值呢?

不会的。因为 CAS 是一种原子操作，它是一种系统原语，是一条 CPU 的原子指令，从 CPU 层面已经保证它的原子性。

当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败，但失败的线程并不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。

CAS 的原理

前面提到，CAS 是一种原子操作。那么 Java 是怎样来使用 CAS 的呢？我们知道，在 Java 中，如果一个方法是 native 的，那 Java 就不负责具体实现它，而是交给底层的 JVM 使用 C 语言 或者 C++ 去实现。

在 Java 中，有一个`Unsafe`类（后面会细讲，戳链接直达），它在`sun.misc`包中。它里面都是一些`native`方法，其中就有几个是关于 CAS 的：

```
boolean compareAndSwapObject(Object o, long offset,Object expected, Object x);
boolean compareAndSwapInt(Object o, long offset,int expected,int x);
boolean compareAndSwapLong(Object o, long offset,long expected,long x);
```

Unsafe 对 CAS 的实现是通过 C++ 实现的，它的具体实现和操作系统、CPU 都有关系。

Linux 的 X86 下主要是通过`cmpxchgl`这个指令在 CPU 上完成 CAS 操作的，但在多处理器情况下，必须使用`lock`指令加锁来完成。当然，不同的操作系统和处理器在实现方式上肯定会有所不同。

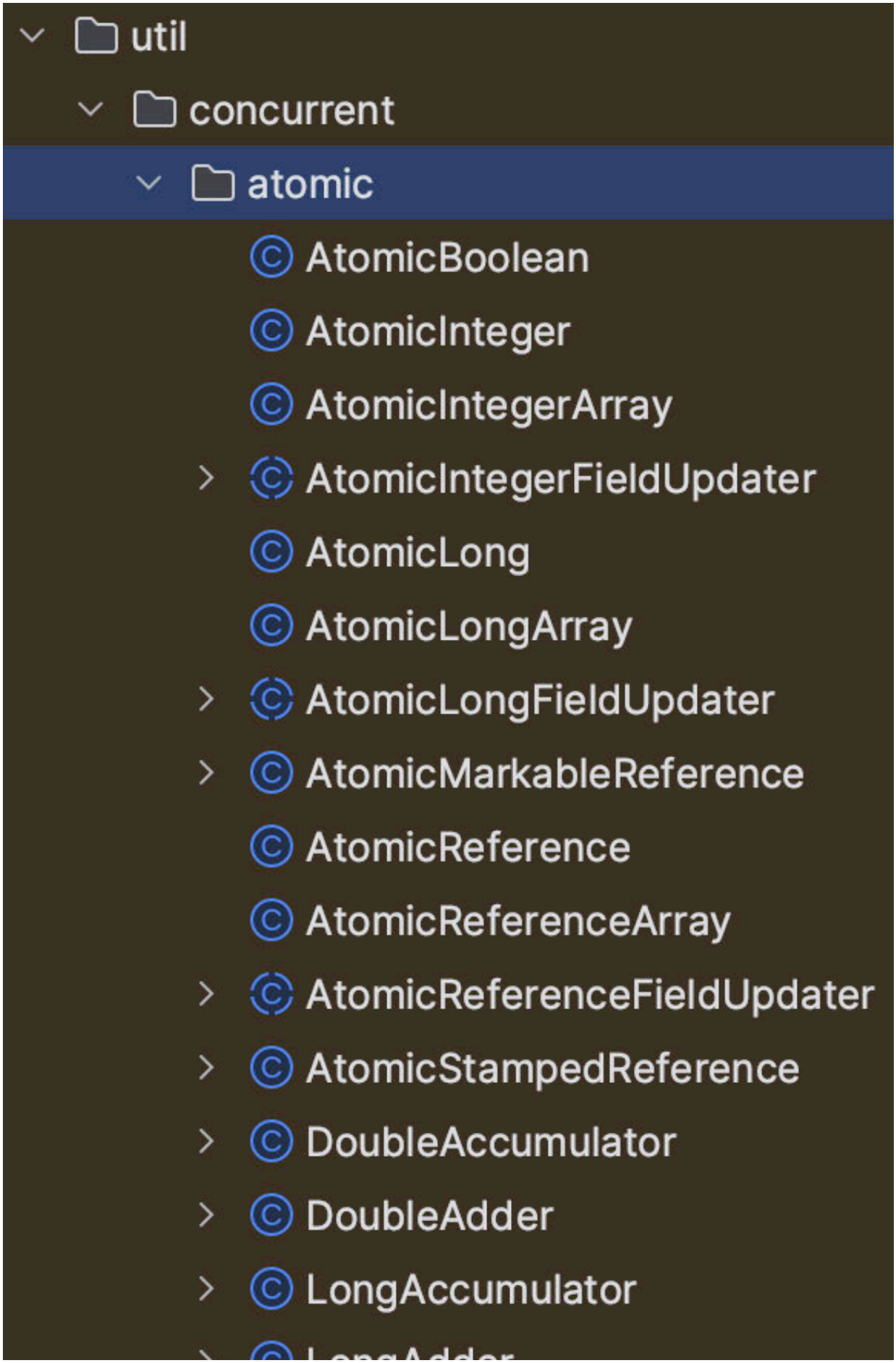
> CMPXCHG是“Compare and Exchange”的缩写，它是一种原子指令，用于在多核/多线程环境中安全地修改共享数据。CMPXCHG在很多现代微处理器体系结构中都有，例如Intel x86/x64体系。对于32位操作数，这个指令通常写作CMPXCHG，而在64位操作数中，它被称为CMPXCHG8B或CMPXCHG16B。

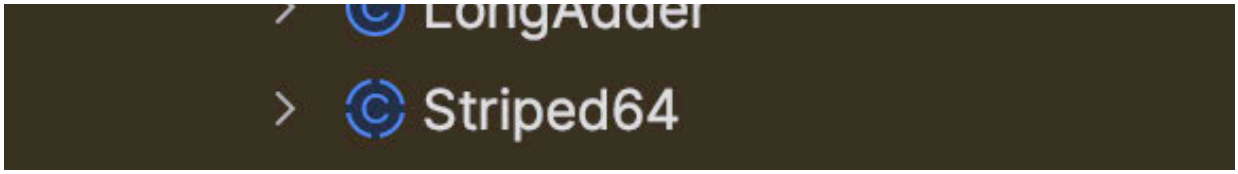
除了上面提到的方法，Unsafe 里面还有其它的方法。比如支持线程挂起和恢复的`park`和`unpark`方法，LockSupport 类（后面会讲）底层就调用了这两个方法。还有支持反射操作的`allocateInstance()`方法。

CAS 如何实现原子操作?

上面介绍了 Unsafe 类的几个支持 CAS 的方法。那 Java 具体是如何通过这几个方法来实现原子操作的呢?

JDK 提供了一些用于原子操作的类，在`java.util.concurrent.atomic`包下面。在 JDK 8 中，有以下这些类：





从名字就可以看出来这些类大概的用途（原子类后面会细讲，戳链接直达）：

- 原子更新基本类型
- 原子更新数组
- 原子更新引用
- 原子更新字段（属性）

这里我们以`AtomicInteger`类的`getAndAdd(int delta)`方法为例，来看看 Java 是如何实现原子操作的。

先来看 getAndAdd 方法的源码：

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}
```

这里的 unsafe 其实就是一个`Unsafe`对象：

```
// setup to use Unsafe.compareAndSwapInt for updates
private static final Unsafe unsafe = Unsafe.getUnsafe();
```

所以，`AtomicInteger`类的`getAndAdd()`方法是通过调用`Unsafe`类的方法实现的：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

让我们详细分析下这段代码，先看参数：

- Object var1，这个参数代表你想要进行操作的对象。
- long var2，这个参数是你想要操作的 var1 对象中的某个字段的偏移量。这个偏移量可以通过 Unsafe 类的 objectFieldOffset 方法获得。
- int var4，这个参数是你想要增加的值。

再来看方法执行的过程：

- 首先，在 do while 循环开始，通过`this.getIntVolatile(var1, var2)`获取当前对象指定字段的值，将其存入临时变量 var5 中。这里的 getIntVolatile 方法能保证读操作的可见性，即读取的结果是最新的写入结果，不会因为 JVM 的优化策略（如指令重排序）或者 CPU 的缓存导致读取到过期的数据。
- 然后，执行`compareAndSwapInt(var1, var2, var5, var5 + var4)`进行 CAS 操作。如果对象 var1 在内存地址 var2 处的值等于预期值 var5，则将该位置的值更新为 var5 + var4，并返回 true；否则，不做任何操

作并返回 false。
- 如果 CAS 操作成功，说明我们成功地将 var1 对象的 var2 偏移量处的字段的值更新为 var5 + var4，并且这个更新操作是原子性的，因此我们跳出循环并返回原来的值 var5。
- 如果 CAS 操作失败，说明在我们尝试更新值的时候，有其他线程修改了该字段的值，所以我们继续循环，重新获取该字段的值，然后再次尝试进行 CAS 操作。

这里使用的是**do-while 循环**。这种循环不多见，它的目的是**保证循环体内的语句至少会被执行一遍**。这样才能保证 return 的值是我们期望的值。

JDK 9 及其以后版本中，getAndAddInt 方法和 JDK 8 中的实现有所不同，我们就拿 JDK 11 的源码来做一个对比吧：

```
@HotSpotIntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

这个方法上面增加了 `@HotSpotIntrinsicCandidate` 注解。这个注解允许 HotSpot VM 自己来写汇编或 IR 编译器来实现该方法以提供更加的性能。

> IR（Intermediate Representation）是一种用于帮助优化编译器的中间代码表示方法。编译器通常将源代码首先转化为 IR，然后对 IR 进行各种优化，最后将优化后的 IR 转化为目标代码。在 JVM（Java Virtual Machine）中，JIT（Just-In-Time）编译器将 Java 字节码（即.class 文件的内容）转化为 IR，然后对 IR 进行优化，最后将 IR 编译为机器码。这个过程在 Java 程序运行时进行，因此被称为“即时编译”。JVM 中的 C1 和 C2 编译器就是 IR 编译器。C1 编译器在编译时进行一些简单的优化，然后快速地将 IR 编译为机器码。C2 编译器在编译时进行更深入的优化，以获得更高的执行效率，但编译的时间也相对更长。

也就是说，虽然表面上看到的是 weakCompareAndSet 和 compareAndSet，但是不排除 HotSpot VM 会手动来实现 weakCompareAndSet 真正功能的可能性。

简单来说，`weakCompareAndSet` 操作仅保留了 `volatile` 自身变量的特性，而除去了 happens-before 规则带来的内存语义。换句话说，`weakCompareAndSet` **无法保证处理操作目标的 volatile 变量外的其他变量的执行顺序（编译器和处理器为了优化程序性能而对指令序列进行重新排序），同时也无法保证这些变量的可见性。** 但这在一定程度上可以提高性能。

再回到循环条件上来，可以看到它是在不断尝试去用 CAS 更新。如果更新失败，就继续重试。

为什么要把获取“旧值”v 的操作放到循环体内呢?

这也好理解。前面我们说了，CAS 如果旧值 V 不等于预期值 E，就会更新失败。说明旧的值发生了变化。那我们当然需要返回的是被其他线程改变之后的旧值了，因此放在了 do 循环体内。

CAS 的三大问题

尽管 CAS 提供了一种有效的同步手段，但也存在一些问题，主要有以下三个：ABA 问题、长时间自旋、多个共享变量的原子操作。

ABA 问题

所谓的 ABA 问题，就是一个值原来是 A，变成了 B，又变回了 A。这个时候使用 CAS 是检查不出变化的，但实际上却被更新了两次。

ABA 问题的解决思路是在变量前面追加上**版本号或者时间戳**。从 JDK 1.5 开始，JDK 的 atomic 包里提供了一个类`AtomicStampedReference`类来解决 ABA 问题。

这个类的`compareAndSet`方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果二者都相等，才使用 CAS 设置为新的值和标志。

```
public boolean compareAndSet(V   expectedReference,
                             V   newReference,
                             int expectedStamp,
                             int newStamp) {
    Pair<V> current = pair;
    return
        expectedReference == current.reference &&
        expectedStamp == current.stamp &&
        ((newReference == current.reference &&
          newStamp == current.stamp) ||
          casPair(current, Pair.of(newReference, newStamp)));
}
```

先来看参数：

- expectedReference：预期引用，也就是你认为原本应该在那个位置的引用。
- newReference：新引用，如果预期引用正确，将被设置到该位置的新引用。
- expectedStamp：预期标记，这是你认为原本应该在那个位置的标记。
- newStamp：新标记，如果预期标记正确，将被设置到该位置的新标记。

执行流程：

①、`Pair<V> current = pair;`这行代码获取当前的 pair 对象，其中包含了引用和标记。

②、接下来的 return 语句做了几个检查：

- `expectedReference == current.reference && expectedStamp == current.stamp`：首先检查当前的引用和标记是否和预期的引用和标记相同。如果二者中有任何一个不同，这个方法就会返回 false。
- 如果上述检查通过，也就是说当前的引用和标记与预期的相同，那么接下来就会检查新的引用和标记是否也与当前的相同。如果相同，那么实际上没有必要做任何改变，这个方法就会返回 true。
- 如果新的引用或者标记与当前的不同，那么就会调用 casPair 方法来尝试更新 pair 对象。casPair 方法会尝试用 newReference 和 newStamp 创建的新的 Pair 对象替换当前的 pair 对象。如果替换成功，casPair 方法会返回 true；如果替换失败（也就是说在尝试替换的过程中，pair 对象已经被其他线程改变了），casPair 方法会返回 false。

长时间自旋

CAS 多与自旋结合。如果自旋 CAS 长时间不成功，会占用大量的 CPU 资源。

解决思路是让 JVM 支持处理器提供的**pause 指令**。

pause 指令能让自旋失败时 cpu 睡眠一小段时间再继续自旋，从而使得读操作的频率降低很多，为解决内存顺序冲突而导致的 CPU 流水线重排的代价也会小很多。

多个共享变量的原子操作

当对一个共享变量执行操作时，CAS 能够保证该变量的原子性。但是对于多个共享变量，CAS 就无法保证操作的原子性，这时通常有两种做法：

1. 使用`AtomicReference`类保证对象之间的原子性，把多个变量放到一个对象里面进行 CAS 操作；
2. 使用锁。锁内的临界区代码可以保证只有当前线程能操作。

小结

CAS（Compare-and-Swap）是一种被广泛应用在并发控制中的算法，它是一种乐观锁的实现方式。CAS 全称为“比较并交换”，是一种无锁的原子操作。

CAS 的全称是：比较并交换（Compare And Swap）。在 CAS 中，有这样三个值：

- V：要更新的变量(var)
- E：预期值(expected)
- N：新值(new)

比较并交换的过程如下：

判断 V 是否等于 E，如果等于，将 V 的值设置为 N；如果不等，说明已经有其它线程更新了 V，于是当前线程放弃更新，什么都不做。

这里的**预期值 E 本质上指的是“旧值”**。

CAS 虽好，但也有一些问题，比如说 ABA 问题、循环时间长开销大、只能保证一个共享变量的原子操作等。在开发中，我们要根据实际情况来选择使用 CAS 还是使用锁。

> 编辑：沉默王二，编辑前的内容来源于朋友开源的这个仓库：深入浅出 Java 多线程，强烈推荐。

第十三节：抽象队列同步器 AQS

AQS是`AbstractQueuedSynchronizer`的简称，即`抽象队列同步器`，从字面上可以这样理解：

- 抽象：抽象类，只实现一些主要逻辑，有些方法由子类实现；
- 队列：使用先进先出（FIFO）的队列存储数据；
- 同步：实现了同步的功能。

那 AQS 有什么用呢?

AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的同步器，比如我们后面会细讲的 ReentrantLock，Semaphore，ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等，都是基于 AQS 的。

当然了，我们也可以利用 AQS 轻松定制专属的同步器，只要实现它的几个`protected`方法就可以了。

AQS 的数据结构

AQS 内部使用了一个 volatile 的变量 state 来作为资源的标识。

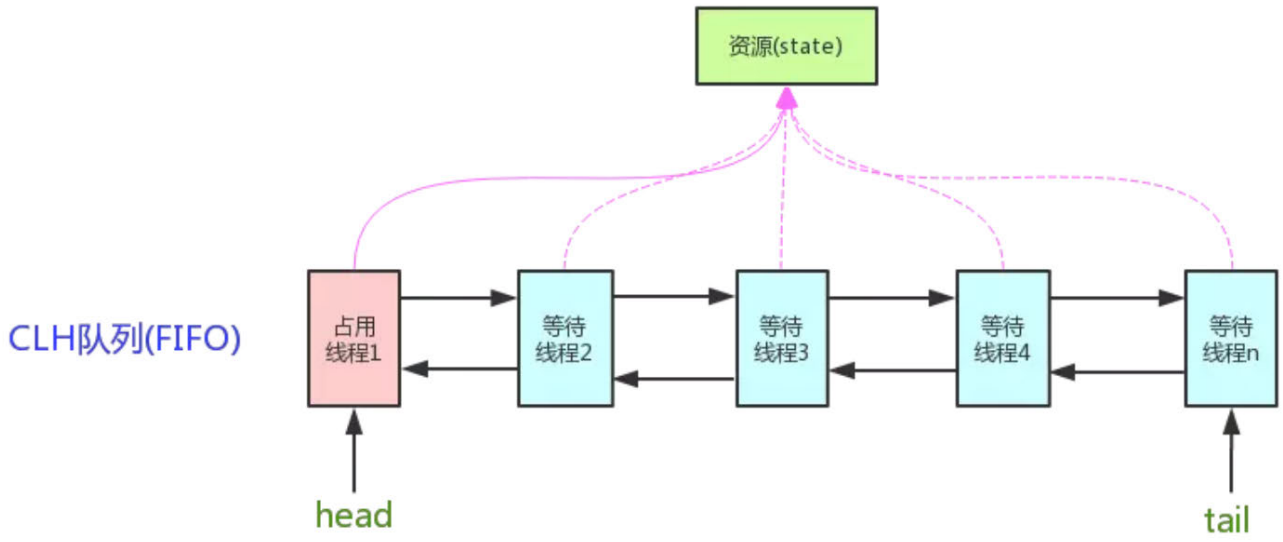
```
/**
 * The synchronization state.
 */
private volatile int state;
```

同时定义了几个获取和改变 state 的 protected 方法，子类可以覆盖这些方法来实现自己的逻辑：

```
getState()
setState()
compareAndSetState()
```

这三种操作均是原子操作，其中 compareAndSetState 的实现依赖于 Unsafe 的 `compareAndSwapInt()` 方法。

AQS 内部使用了一个先进先出（FIFO）的双端队列，并使用了两个引用 head 和 tail 用于标识队列的头部和尾部。其数据结构如下图所示：



但它并不直接储存线程，而是储存拥有线程的 Node 节点。

等待队列节点类。

等待队列是“CLH”（Craig、Landin 和 Hagersten）锁定队列的变体。 CLH 锁通常用于自旋锁。相反，我们将它们用于阻塞同步器，但使用相同的基本策略，即在其节点的前驱中保存有关线程的一些控制信息。每个节点中的“状态”字段跟踪线程是否应该阻塞。当其前任节点释放时，节点会收到信号。否则，队列的每个节点都充当持有单个等待线程的特定通知样式监视器。状态字段不控制线程是否被授予锁等。如果线程位于队列中的第一个，则它可能会尝试获取。但成为第一并不能保证成功；它只赋予竞争的权利。所以当前释放的竞争者线程可能需要重新等待。

要排队到 CLH 锁中，您可以原子地将其拼接为新的尾部。要出队，只需设置头字段即可。

```
      +------+  prev +------+       +------+
 head |      | <---- |      | <---- |      |  tail
      +------+       +------+       +------+
```

插入到 CLH 队列中只需要对“尾部”执行单个原子操作，因此存在一个从未排队到已排队的简单原子分界点。类似地，出队仅涉及更新“头部”。然而，节点需要做更多的工作来确定谁是他们的继任者，部分原因是为了处理由于超时和中断而可能取消的情况。

“prev”链接（在原始 CLH 锁中未使用）主要用于处理取消。如果一个节点被取消，它的后继节点（通常）会重新链接到未取消的前任节点。有关自旋锁情况下类似机制的解释，请参阅 Scott 和 Scherer 的论文，网址为：http://www.cs.rochester.edu/u/scott/synchronization/

我们还使用“下一个”链接来实现阻塞机制。每个节点的线程 ID 保存在其自己的节点中，因此前驱通过遍历下一个链接来确定它是哪个线程，从而向下一个节点发出信号以唤醒。后继者的确定必须避免与新排队的节点竞争以设置其前任者的“下一个”字段。当节点的后继者似乎为空时，必要时可以通过从原子更新的“尾部”向后检查来解决此问题。 （或者，换句话说，下一个链接是一种优化，因此我们通常不需要向后扫描。）

取消给基本算法引入了一些保守性。由于我们必须轮询其他节点的取消，因此我们可能会忽略取消的节点是在我们前面还是后面。这是通过在取消时始终取消后继者来解决的，从而使他们能够稳定在新的前任上，除非我们能够确定一个未取消的前任将承担这一责任。

CLH 队列需要一个虚拟头节点才能启动。但我们不会在构建时创建它们，因为如果不存在争用，那就是浪费精力。相反，在第一次争用时构造节点并设置头指针和尾指针。

等待条件的线程使用相同的节点，但使用附加链接。条件只需要链接简单（非并发）链接队列中的节点，因为它们仅在独占持有时才被访问。在等待时，节点被插入到条件队列中。收到信号后，节点将被转移到主队列。状态字段的特殊值用于标记节点位于哪个队列。

感谢 Dave Dice、Mark Moir、Victor Luchangco、Bill Scherer 和 Michael Scott，以及 JSR-166 专家组的成员，他们对本课程的设计提供了有益的想法、讨论和批评。

```
static final class Node { ... }
```

AQS 的 Node 节点

资源有两种共享模式，或者说两种同步方式：

- 独占模式（Exclusive）：资源是独占的，一次只能有一个线程获取。如 ReentrantLock（后面会细讲，戳链接直达）。
- 共享模式（Share）：同时可以被多个线程获取，具体的资源个数可以通过参数指定。如 Semaphore/CountDownLatch（戳链接直达，后面会细讲）。

一般情况下，子类只需要根据需求实现其中一种模式就可以，当然也有同时实现两种模式的同步类，如 ReadWriteLock。

AQS 中关于这两种资源共享模式的定义源码均在内部类 Node 中。我们来看看 Node 的结构：

```
static final class Node {
    // 标记一个结点（对应的线程）在共享模式下等待
```

```
    static final Node SHARED = new Node();
    // 标记一个结点（对应的线程）在独占模式下等待
    static final Node EXCLUSIVE = null;

    // waitStatus的值，表示该结点（对应的线程）已被取消
    static final int CANCELLED = 1;
    // waitStatus的值，表示后继结点（对应的线程）需要被唤醒
    static final int SIGNAL = -1;
    // waitStatus的值，表示该结点（对应的线程）在等待某一条件
    static final int CONDITION = -2;
    /*waitStatus的值，表示有资源可用，新head结点需要继续唤醒后继结点（共享模式下，多线程并发释放资源，而head唤醒其后继结点后，需要把多出来的资源留给后面的结点；设置新的head结点时，会继续唤醒其后继结点）*/
    static final int PROPAGATE = -3;

    // 等待状态，取值范围，-3，-2，-1，0，1
    volatile int waitStatus;
    volatile Node prev; // 前驱结点
    volatile Node next; // 后继结点
    volatile Thread thread; // 结点对应的线程
    Node nextWaiter; // 等待队列里下一个等待条件的结点


    // 判断共享模式的方法
    final boolean isShared() {
        return nextWaiter == SHARED;
    }

    Node(Thread thread, Node mode) {     // Used by addWaiter
        this.nextWaiter = mode;
        this.thread = thread;
    }

    // 其它方法忽略，可以参考具体的源码
}

// AQS里面的addWaiter私有方法
private Node addWaiter(Node mode) {
    // 使用了Node的这个构造函数
    Node node = new Node(Thread.currentThread(), mode);
    // 其它代码省略
}
```

这里面的 waitStatus 是用来标记当前节点的状态的，它有以下几种状态：

- CANCELLED：表示当前节点（对应的线程）已被取消。当等待超时或被中断，会触发进入为此状态，进入该状态后节点状态不再变化；
- SIGNAL：后面节点等待当前节点唤醒；
- CONDITION：Condition（后面会细讲，戳链接直达）中使用，当前线程阻塞在Condition，如果其他线程调

用了Condition的signal方法，这个节点将从等待队列转移到同步队列队尾，等待获取同步锁；
- PROPAGATE：共享模式，前置节点唤醒后面节点后，唤醒操作无条件传播下去；
- 0：中间状态，当前节点后面的节点已经唤醒，但是当前节点线程还没有执行完成。

通过 Node 我们可以实现两种队列：

1）一是通过 prev 和 next 实现 CLH（Craig, Landin, and Hagersten）队列（线程同步队列、双向队列）。

在 CLH 锁中，每个等待的线程都会有一个关联的 Node，每个 Node 有一个 prev 和 next 指针。当一个线程尝试获取锁并失败时，它会将自己添加到队列的尾部并自旋，等待前一个节点的线程释放锁。类似下面这样。

```
public class CLHLock {
    private volatile Node tail;
    private ThreadLocal<Node> myNode = ThreadLocal.withInitial(Node::new);
    private ThreadLocal<Node> myPred = new ThreadLocal<>();

    public void lock() {
        Node node = myNode.get();
        node.locked = true;
        // 把自己放到队尾，并取出前面的节点
        Node pred = tail;
        myPred.set(pred);
        while (pred.locked) {
            // 自旋等待
        }
    }

    public void unlock() {
        Node node = myNode.get();
        node.locked = false;
        myNode.set(myPred.get());
    }

    private static class Node {
        private volatile boolean locked;
    }
}
```

2）二是通过 nextWaiter 实现 Condition（后面会细讲，戳链接直达）上的等待线程队列（单向队列），这个 Condition 主要用在 ReentrantLock 类中。

AQS 的源码解析

AQS 的设计是基于**模板方法模式**的，它有一些方法必须要子类去实现的，它们主要有：

- `isHeldExclusively()`：该线程是否正在独占资源。只有用到 condition 才需要去实现它。
- `tryAcquire(int)`：独占方式。尝试获取资源，成功则返回 true，失败则返回 false。
- `tryRelease(int)`：独占方式。尝试释放资源，成功则返回 true，失败则返回 false。

- `tryAcquireShared(int)`：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- `tryReleaseShared(int)`：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回 true，否则返回 false。

这些方法虽然都是`protected`的，但是它们并没有在 AQS 具体实现，而是直接抛出异常：

```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
```

这里不使用抽象方法的目的是：避免强迫子类中把所有的抽象方法都实现一遍，减少无用功，这样子类只需要实现自己关心的抽象方法即可，比如 信号 Semaphore 只需要实现 tryAcquire 方法而不用实现其余不需要用到的模版方法：

而 AQS 实现了一系列主要的逻辑。下面我们从源码来分析一下获取和释放资源的主要逻辑：

获取资源

获取资源的入口是`acquire(int arg)`方法。arg 是要获取的资源个数，在独占模式下始终为 1。我们先来看看这个方法的逻辑：

```
public final void acquire(int arg) {
    // tryAcquire 再次尝试获取锁资源，如果尝试成功，返回true，尝试失败返回false
    if (!tryAcquire(arg) &&
        // 走到这，代表获取锁资源失败，需要将当前线程封装成一个Node，追加到AQS的队列中
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        // 线程中断
        selfInterrupt();
}
```

首先调用 tryAcquire 尝试去获取资源。前面提到了这个方法是在子类中具体实现的，可以直接进入 ReentrantLock 中 查看。

如果获取资源失败，就通过 `addWaiter(Node.EXCLUSIVE)` 方法把这个线程插入到等待队列中。其中传入的参数代表要插入的 Node 是独占式的。这个方法的具体实现：

```
private Node addWaiter(Node mode) {
 //创建 Node 类，并且设置 thread 为当前线程，设置为排它锁
 Node node = new Node(Thread.currentThread(), mode);
 // 获取 AQS 中队列的尾部节点
 Node pred = tail;
 // 如果 tail == null，说明是空队列，
 // 不为 null，说明现在队列中有数据，
 if (pred != null) {
  // 将当前节点的 prev 指向刚才的尾部节点，那么当前节点应该设置为尾部节点
  node.prev = pred;
  // CAS 将 tail 节点设置为当前节点
  if (compareAndSetTail(pred, node)) {
   // 将之前尾节点的 next 设置为当前节点
   pred.next = node;
   // 返回当前节点
   return node;
  }
 }
 enq(node);
 return node;
}

// 自旋CAS插入等待队列
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

上面的两个方法比较好理解，就是在队列的尾部插入新的 Node 节点，但是需要注意的是由于 AQS 中会存在多个线程同时争夺资源的情况，因此肯定会出现多个线程同时插入节点的操作，在这里是通过 CAS 自旋的方式保证了操作的线程安全性。

OK，现在回到最开始的 aquire 方法。现在通过 addWaiter 方法，已经把一个 Node 放到等待队列尾部了。而处于等待队列的结点是从头结点一个一个去获取资源的。具体的实现我们来看看 acquireQueued 方法：

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        // interrupted用于记录线程是否被中断过
        boolean interrupted = false;
        for (;;) { // 自旋操作
            // 获取当前节点的前驱节点
            final Node p = node.predecessor();
            // 如果前驱节点是head节点，并且尝试获取同步状态成功
            if (p == head && tryAcquire(arg)) {
                // 设置当前节点为head节点
                setHead(node);
                // 前驱节点的next引用设为null，帮助垃圾回收器回收该节点
                p.next = null;
                // 获取同步状态成功，将failed设为false
                failed = false;
                // 返回线程是否被中断过
                return interrupted;
            }
            // 如果应该让当前线程阻塞并且线程在阻塞时被中断，则将interrupted设为true
            if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        // 如果获取同步状态失败，取消尝试获取同步状态
        if (failed)
            cancelAcquire(node);
    }
}
```

这里 parkAndCheckInterrupt 方法内部使用到了 `LockSupport.park(this)`，顺便简单介绍一下 park 方法。

LockSupport 类是 Java 6 引入的一个类，提供了基本的线程同步原语。LockSupport 实际上是调用了 Unsafe 类里的方法，归结到 Unsafe 里，只有两个：

- `park(boolean isAbsolute, long time)`：阻塞当前线程
- `unpark(Thread jthread)`：使给定的线程停止阻塞

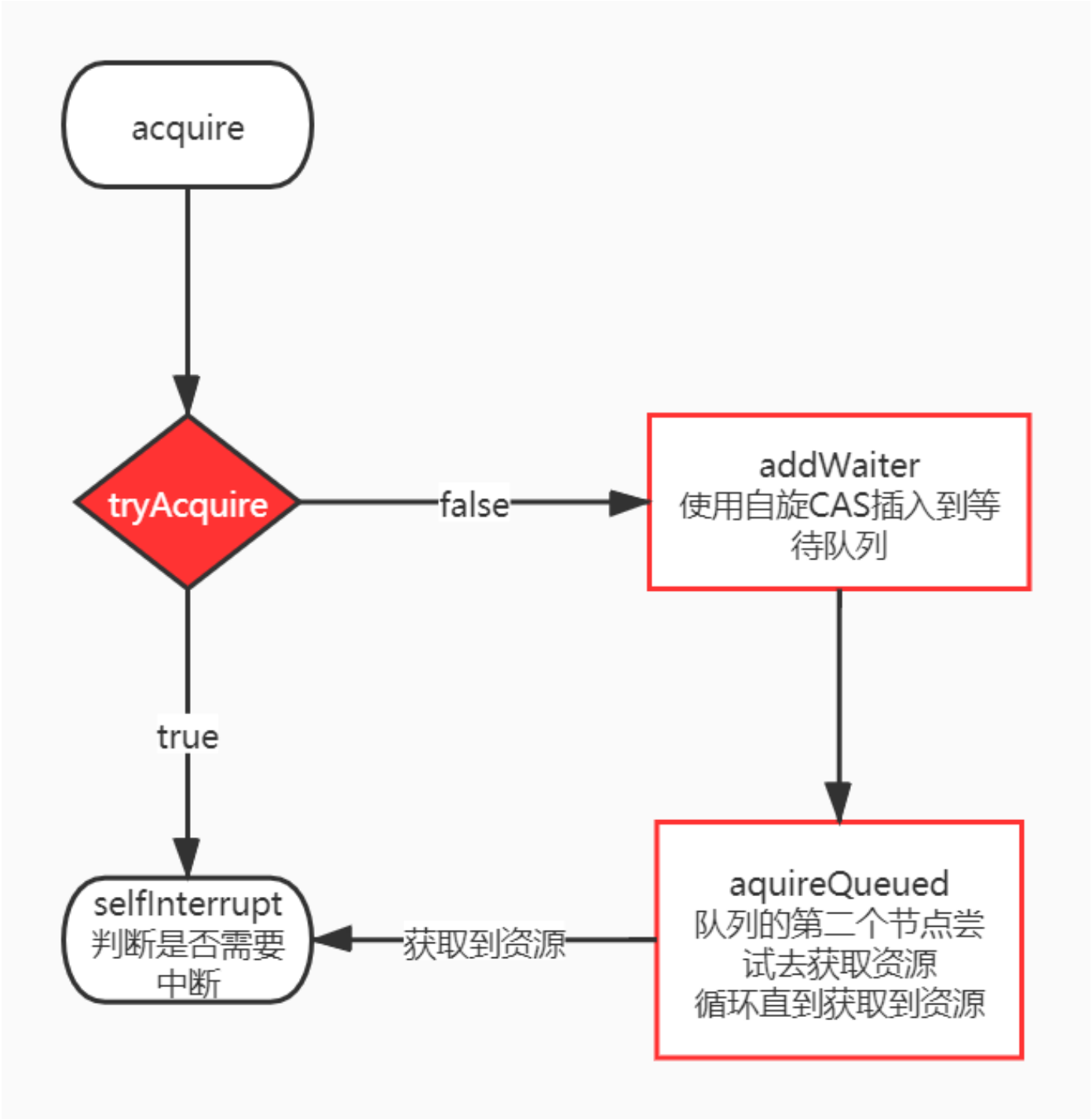
所以**结点进入等待队列后，是调用 park 使它进入阻塞状态的。只有头结点的线程是处于活跃状态的。**

当然，获取资源的方法除了 acquire 外，还有以下三个：

- acquireInterruptibly：申请可中断的资源（独占模式）
- acquireShared：申请共享模式的资源
- acquireSharedInterruptibly：申请可中断的资源（共享模式）

可中断的意思是，在线程中断时可能会抛出 `InterruptedException`

总结起来的一个流程图：



释放资源

释放资源相比于获取资源来说，会简单许多。在 AQS 中只有一小段实现。源码：

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

```
private void unparkSuccessor(Node node) {
    // 如果状态是负数，尝试把它设置为0
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    // 得到头结点的后继结点head.next
    Node s = node.next;
    // 如果这个后继结点为空或者状态大于0
    // 通过前面的定义我们知道，大于0只有一种可能，就是这个结点已被取消（只有 Node.CANCELLED(=1) 这一种状态大于0）
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 从尾部开始倒着寻找第一个还未取消的节点（真正的后继者）
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    // 如果后继结点不为空，
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

在`java.util.concurrent.locks.ReentrantLock`的实现中，`tryRelease(arg)`会减少持有锁的数量，如果持有锁的数量变为0，释放锁并返回true。

如果`tryRelease(arg)`成功释放了锁，那么接下来会检查队列的头结点。如果头结点存在并且waitStatus不为0（这意味着有线程在等待），那么会调用`unparkSuccessor(Node h)`方法来唤醒等待的线程。

小结

AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的同步器，比如我们提到的 ReentrantLock，Semaphore，ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等皆是基于 AQS 的。

当然了，我们也可以利用 AQS 轻松定制专属的同步器，只要实现它的几个`protected`方法就可以了。

来个互斥锁（同一时刻只允许一个线程持有锁）。

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

public class Mutex {

    private static class Sync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int arg) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
```

```
            return false;
        }

        @Override
        protected boolean tryRelease(int arg) {
            if (getState() == 0) {
                throw new IllegalMonitorStateException();
            }
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }

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

    private final Sync sync = new Sync();

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

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

    public boolean isLocked() {
        return sync.isHeldExclusively();
    }
}
```

上面的 Mutex 类是一个互斥锁。它内部使用了一个 Sync 类，该类继承自 AQS。

- tryAcquire：尝试获取资源。如果当前状态为0（未锁定），那么设置为1（锁定），并设置当前线程为独占资源的线程。
- tryRelease：尝试释放资源。设置状态为0并清除持有资源的线程。
- isHeldExclusively：判断当前资源是否被独占。

假设有一个线程不安全的资源，我们需要确保在任何时刻只有一个线程能访问它，那么就可以使用这个 Mutex 锁来确保线程安全。

```
public class Resource {
    private Mutex mutex = new Mutex();

    public void use() {
        mutex.lock();
        try {
            // 对资源的操作
        } finally {
            mutex.unlock();
        }
    }
}
```

在上述场景中，我们为一个不安全的资源添加了一个互斥锁，确保同一时刻只有一个线程可以使用这个资源，从而确保线程安全。

> 编辑：沉默王二，编辑前的内容来源于朋友开源的这个仓库：深入浅出 Java 多线程，强烈推荐。值得参考文章：君哥聊技术：2万字 + 40 张图带你精通 Java AQS，阿 Q 说代码：20张图带你彻底了解加锁和解锁

第十四节：锁分类和 JUC

前面我们介绍了基于对象的原生锁——synchronized，实际上，Java 在`java.util.concurrent`（JUC）包下，还为我们提供了更多的锁类和锁接口（尤其是子包 locks 下），它们有更强大的功能或更牛逼的性能。

来看看`synchronized`的不足之处吧。

- 如果临界区是只读操作，其实可以多线程一起执行，但使用 synchronized 的话，**同一时间只能有一个线程执行**。
- synchronized 无法知道线程有没有成功获取到锁。
- 使用 synchronized，如果临界区因为 IO 或者 sleep 方法等原因阻塞了，而当前线程又没有释放锁，就会导致**所有线程等待**。

> 临界区（Critical Section）是多线程中一个 非常重要的概念，指的是在代码中访问共享资源的那部分，且同一时刻只能有一个线程能访问的代码。多个线程同时访问临界区的资源如果没有任何同步（加锁）操作，会导致资源的状态不可预测和不一致，从而产生所谓的“竞态条件”(Race Condition)。在许多并发控制策略中，例如互斥锁 synchronized，目标就是确保任何时候只有一个线程进入临界区。

不过，synchronized 的这些不足之处都可以通过 JUC 包下的其他锁来弥补，下面先来看一下锁的分类吧。

锁的几种分类

Java 提供了种类丰富的锁，每种锁因其特性的不同，在适当的场景下能够展现出非常高的效率。我们可以通过特性将锁进行分组归类。

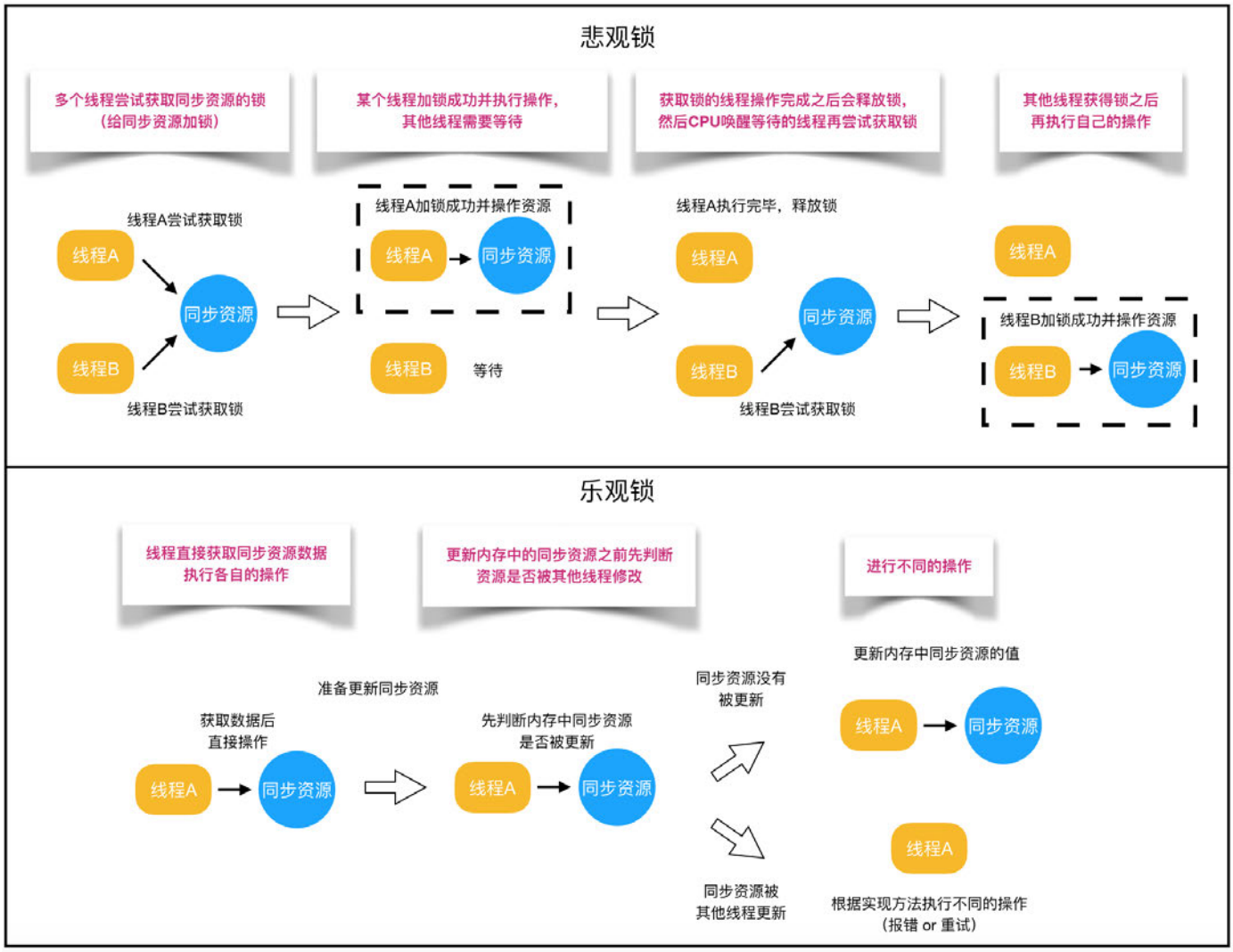
乐观锁 VS 悲观锁

乐观锁与悲观锁是一种广义上的概念，体现了看待线程同步的不同角度。

先说概念。对于同一个数据的并发操作，悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java 中，synchronized 关键字 是最典型的悲观锁。

而乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会加锁，只是在更新数据的时候会去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。

乐观锁在 Java 中是通过无锁编程来实现的，最常采用的是CAS 算法，Java 原子类的递增操作就通过 CAS 自旋实现的。



根据上面的概念描述我们可以发现：

- 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
- 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

光说概念有些抽象，我们来看下乐观锁和悲观锁的调用方式：

```
// --------- 悲观锁的调用方式 -------------------------
// synchronized
public synchronized void testMethod() {
  // 操作同步资源
}
// ReentrantLock
private ReentrantLock lock = new ReentrantLock();
// 需要保证多个线程使用的是同一个锁
public void modifyPublicResources() {
  lock.lock();
  // 操作同步资源
  lock.unlock();
}

// --------- 乐观锁的调用方式 -------------------------
private AtomicInteger atomicInteger = new AtomicInteger();
```

```
// 需要保证多个线程使用的是同一个AtomicInteger
atomicInteger.incrementAndGet(); //执行自增1
```

通过调用方式的举例，我们发现悲观锁基本都是在显式的锁定之后再操作同步资源，而乐观锁则直接去操作同步资源。那么，为何乐观锁能够做到不锁定同步资源也可以正确的实现线程同步呢？我们这里再次来温习一下 “CAS” 的技术原理，之前也讲过，就当是复习了。

CAS 是一种无锁算法，可以在不使用锁（没有线程被阻塞）的情况下实现多线程之间的变量同步。JUC 包中的原子类（后面会细讲，戳链接直达）就是通过 CAS 实现的乐观锁。

CAS 算法涉及到三个操作数：

- 需要读写的内存值 V。
- 进行比较的值 A。
- 要写入的新值 B。

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值（“比较+更新”整体是一个原子操作），否则不会执行任何操作。一般情况下，“更新”是一个不断重试的操作。

之前提到 JUC 包中的原子类，就是通过 CAS 实现的乐观锁，那么我们进入原子类 AtomicInteger 的源码（后面也会细讲，既然讲到了，这里就过一下吧），来看一下 AtomicInteger 的定义：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

根据定义我们可以看出各属性的作用：

- unsafe： 获取并操作内存的数据。
- valueOffset： 存储 value 在 AtomicInteger 中的偏移量。
- value： 存储 AtomicInteger 的 int 值，该属性需要借助 volatile 关键字保证其在线程间是可见的。

接下来，我们查看 AtomicInteger 的自增方法`incrementAndGet()`，发现自增方法底层调用的是`unsafe.getAndAddInt()`。但是由于 JDK 本身只有 Unsafe.class，通过 class 文件中的参数名，并不能很好地了解方法的作用，所以我们通过 OpenJDK 8 来查看 Unsafe（后面也会讲，戳链接直达） 的源码：

```
// ---------------------- JDK 8 ----------------------
```

```
// AtomicInteger 自增方法
public final int incrementAndGet() {
  return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}

// Unsafe.class
public final int getAndAddInt(Object var1, long var2, int var4) {
  int var5;
  do {
      var5 = this.getIntVolatile(var1, var2);
  } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
  return var5;
}

// ------------------------- OpenJDK 8 -------------------------
// Unsafe.java
public final int getAndAddInt(Object o, long offset, int delta) {
   int v;
   do {
       v = getIntVolatile(o, offset);
   } while (!compareAndSwapInt(o, offset, v, v + delta));
   return v;
}
```

根据源码我们可以看出，`getAndAddInt()`循环获取给定对象 o 中偏移量处的值 v，然后判断内存值是否等于 v。如果相等则将内存值设置为 v + delta，否则返回 false，继续循环进行重试，直到设置成功才能退出循环，并且将旧值返回。

整个“比较+更新”操作都封装在`compareAndSwapInt()`中，在 JNI 里是借助于一个 CPU 指令完成的，属于原子操作，可以保证多个线程都能够看到同一个变量的修改值。

> Java Native Interface（JNI）是Java与本地代码（如C、C++）之间的 桥梁。它允许Java代码与原生应用程序的接口（API）和本地库进行交互，并获得一些Java不能轻松完成任务的能力。

后续 JDK 通过 CPU 的 cmpxchg 指令去比较寄存器中的 A 和 内存中的值 V。如果相等，就把要写入的新值 B 存入内存中。如果不相等，就将内存值 V 赋值给寄存器中的值 A。然后通过 Java 代码中的 while 循环再次调用 cmpxchg 指令进行重试，直到设置成功为止。

> CMPXCHG是“Compare and Exchange”的缩写，它是一种原子指令，用于在多核/多线程环境中安全地修改共享数据。CMPXCHG在很多现代微处理器体系结构中都有，例如Intel x86/x64体系。对于32位操作数，这个指令通常写作CMPXCHG，而在64位操作数中，它被称为CMPXCHG8B或CMPXCHG16B。

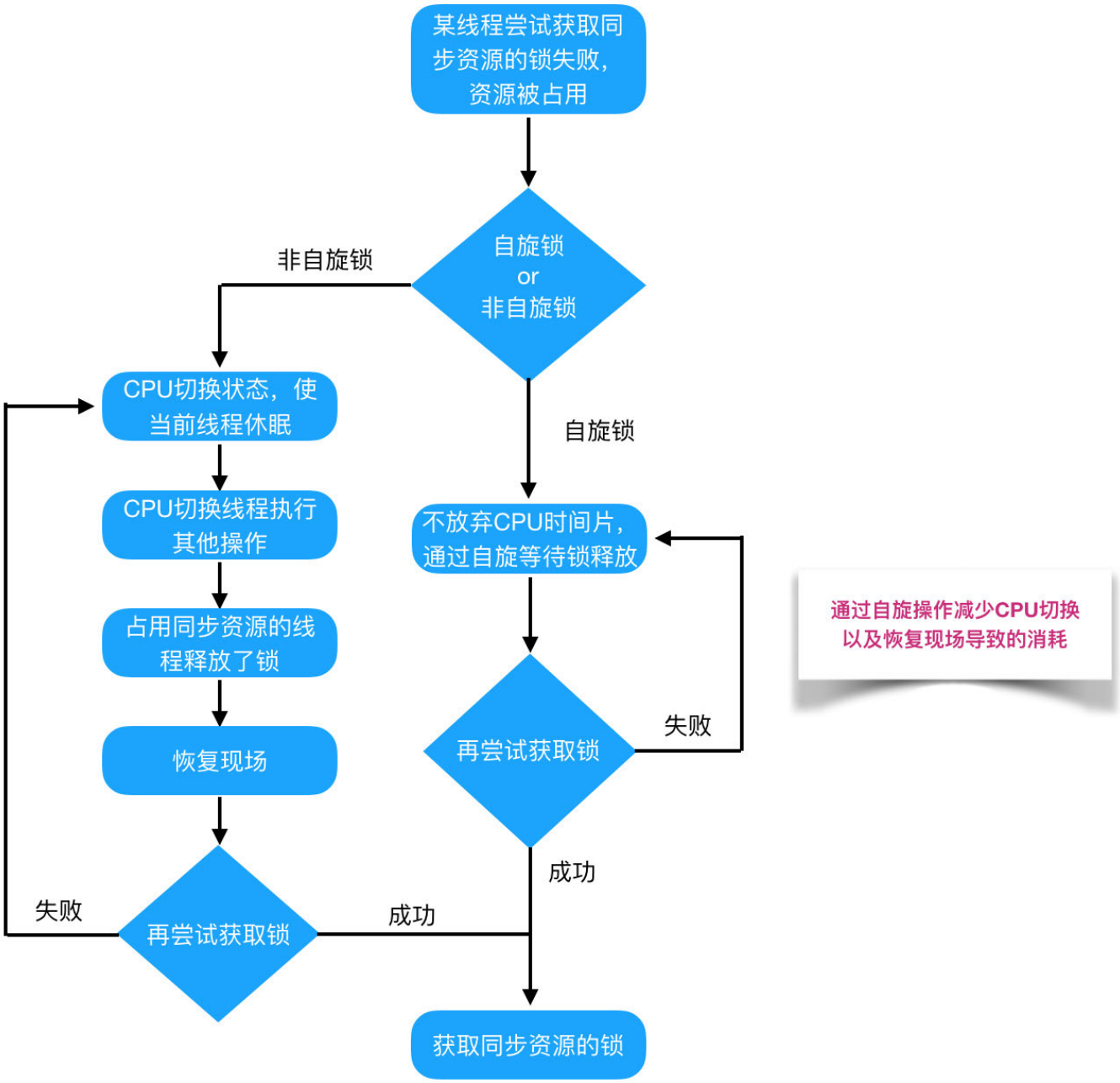
CAS 虽然高效，但也存在三大问题，我们前面也讲过，不知道大家还记得不，如果不记得，可以戳链接去回顾一波（dogdogdog）。

自旋锁 VS 适应性自旋锁

阻塞或唤醒一个 Java 线程需要操作系统切换 CPU 状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。

在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复线程花费的时间可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃 CPU 的执行时间，看看持有锁的线程是否会很快释放锁。

为了让当前线程“稍等一下”，我们需要让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不用阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。



自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是 10 次，可以使用`-XX:PreBlockSpin`来更改）没有成功获得锁，就应当挂起线程。

自旋锁的实现原理同样也是 CAS，AtomicInteger 中调用 unsafe 进行自增操作的源码中的 do-while 循环就是一个自旋操作，如果修改数值失败则通过循环来执行自旋，直至修改成功。

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}
```

自旋锁在 JDK1.4.2 中引入，使用`-XX:+UseSpinning`来开启。JDK 6 中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。

自适应意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功的，进而它将允许自旋等待更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

无锁偏向锁轻量级锁重量级锁

这四种锁是专门针对 synchronized 的，我们在synchronized 锁的到底是什么一文中已经详细地介绍过，这里就不再赘述了。

可重入锁和非可重入锁

可重入锁又名递归锁，是指同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提：锁的是同一个对象或者 class），不会因为之前已经获取过还没释放而阻塞。Java 中ReentrantLock（后面会细讲，戳链接直达）和synchronized都是可重入锁，可重入锁的一个优点就是可以一定程度避免死锁。下面用示例代码来进行分析：

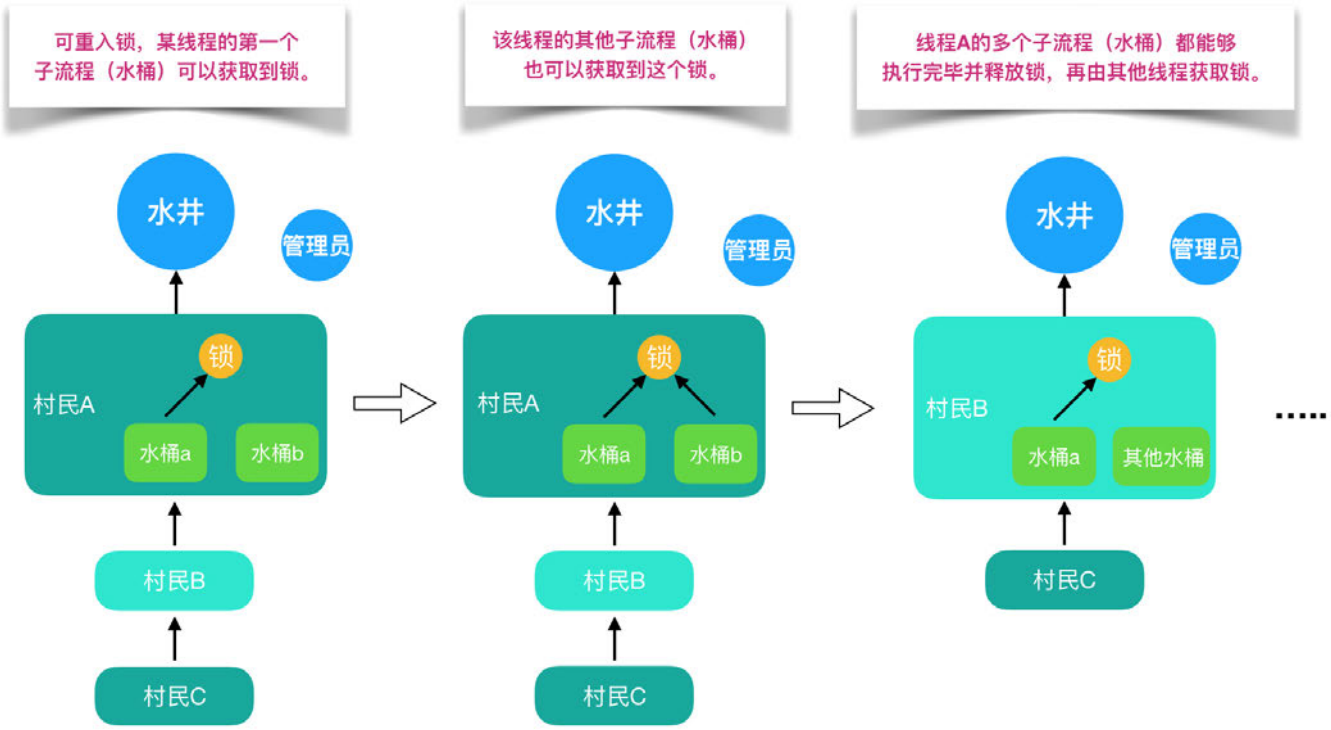
```
public class Widget {
    public synchronized void doSomething() {
        System.out.println("方法1执行...");
        doOthers();
    }

    public synchronized void doOthers() {
        System.out.println("方法2执行...");
    }
}
```

在上面的代码中，类中的两个方法都是被内置锁 synchronized 修饰的，`doSomething()`方法中调用了`doOthers()`方法。因为内置锁是可重入的，所以同一个线程在调用`doOthers()`时可以直接获得当前对象的锁，进入`doOthers()`进行操作。

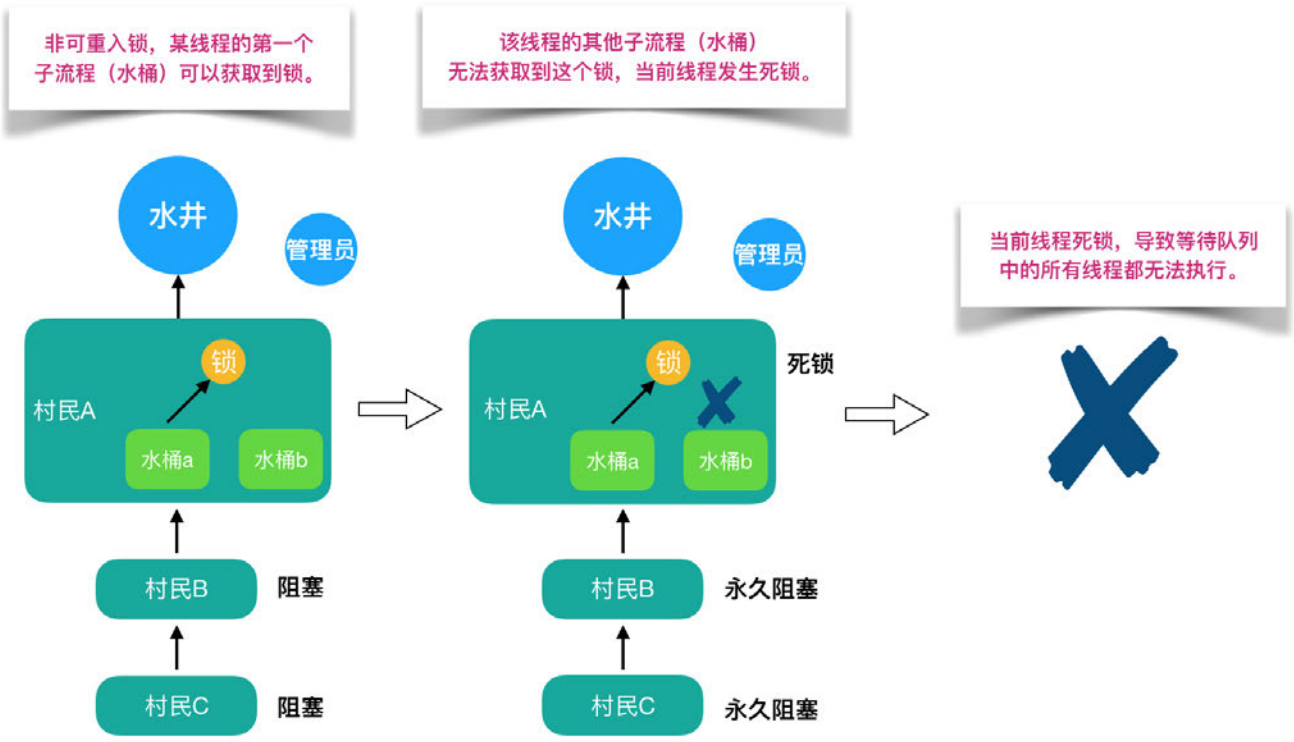
如果是一个不可重入锁，那么当前线程在调用`doOthers()`之前，需要将执行`doSomething()`时获取当前对象的锁释放掉，实际上该对象锁已经被当前线程所持有，且无法释放。所以此时会出现死锁。

那为什么可重入锁就可以在嵌套调用时自动获得锁呢?

还是打水的例子，有多个人在排队打水，此时管理员允许锁和同一个人的多个水桶绑定。这个人用多个水桶打水时，第一个水桶和锁绑定并打完水之后，第二个水桶也可以直接和锁绑定并开始打水，所有的水桶都打完水之后打水人才会将锁还给管理员。这个人的所有打水流程都能够成功执行，后续等待的人也能够打到水。这就是可重入锁。



但如果是非可重入锁的话，此时管理员只允许锁和同一个人的一个水桶绑定。第一个水桶和锁绑定打完水之后并不会释放锁，导致第二个水桶不能和锁绑定也无法打水。当前线程出现死锁，整个等待队列中的所有线程都无法被唤醒。



之前我们说过 ReentrantLock 和 synchronized 都是重入锁，那么我们通过重入锁 ReentrantLock 以及非可重入锁 NonReentrantLock 的源码来对比分析一下为什么非可重入锁在重复调用同步资源时会出现死锁。

首先ReentrantLock和 NonReentrantLock 都继承了父类AQS，其父类 AQS 中维护了一个同步状态 status 来计数重入次数，status 初始值为 0。

当线程尝试获取锁时，可重入锁先尝试获取并更新 status 值，如果`status == 0`表示没有其他线程在执行同步代码，则把 status 置为 1，当前线程开始执行。如果`status != 0`，则判断当前线程是否获取到了这个锁，如果是的话执行`status+1`，且当前线程可以再次获取锁。

而非可重入锁是直接获取并尝试更新当前 status 的值，如果`status != 0`的话会导致其获取锁失败，当前线程阻塞。

释放锁时，可重入锁同样会先获取当前 status 的值，在当前线程是持有锁的线程的前提下。如果`status-1 == 0`，则表示当前线程所有重复获取锁的操作都已经执行完毕，然后该线程才会真正释放锁。而非可重入锁则是在确定当前线程是持有锁的线程之后，直接将 status 置为 0，将锁释放。

可重入锁

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

获取锁时先判断，如果当前线程就是已经占有锁的线程，则status值+1，并返回true。

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

释放锁时也是先判断当前线程是否是已占有锁的线程，然后在判断status。如果status等于0，才真正的释放锁

非可重入锁

```
protected boolean tryAcquire(int acquires) {
    if (this.compareAndSetState( expect: 0, update: 1)) {
        this.owner = Thread.currentThread();
        return true;
    } else {
        return false;
    }
}
```

非重入锁是直接尝试获取锁

```
protected boolean tryRelease(int releases) {
    if (Thread.currentThread() != this.owner) {
        throw new IllegalMonitorStateException();
    } else {
        this.owner = null;
        this.setState(0);
        return true;
    }
}
```

释放锁时也是直接将status置为0

公平锁与非公平锁

这里的“公平”，其实通俗意义来说就是“先来后到”，也就是 FIFO。如果对一个锁来说，先对锁获取请求的线程一定会先被满足，后对锁获取请求的线程后被满足，那这个锁就是公平的。反之，那就是不公平的。

一般情况下，**非公平锁能提升一定的效率。但是非公平锁可能会发生线程饥饿（有一些线程长时间得不到锁）的情况**。所以要根据实际的需求来选择非公平锁和公平锁。

ReentrantLock 支持非公平锁和公平锁两种。

读写锁和排它锁

我们前面讲到的 synchronized 和后面要讲的 ReentrantLock，其实都是“排它锁”。也就是说，这些锁在同一时刻只允许一个线程进行访问。

而读写锁可以在同一时刻允许多个读线程访问。Java 提供了 ReentrantReadWriteLock（后面会细讲，戳链接直达）类作为读写锁的默认实现，内部维护了两个锁：一个读锁，一个写锁。通过分离读锁和写锁，使得在“读多写少”的环境下，大大地提高了性能。

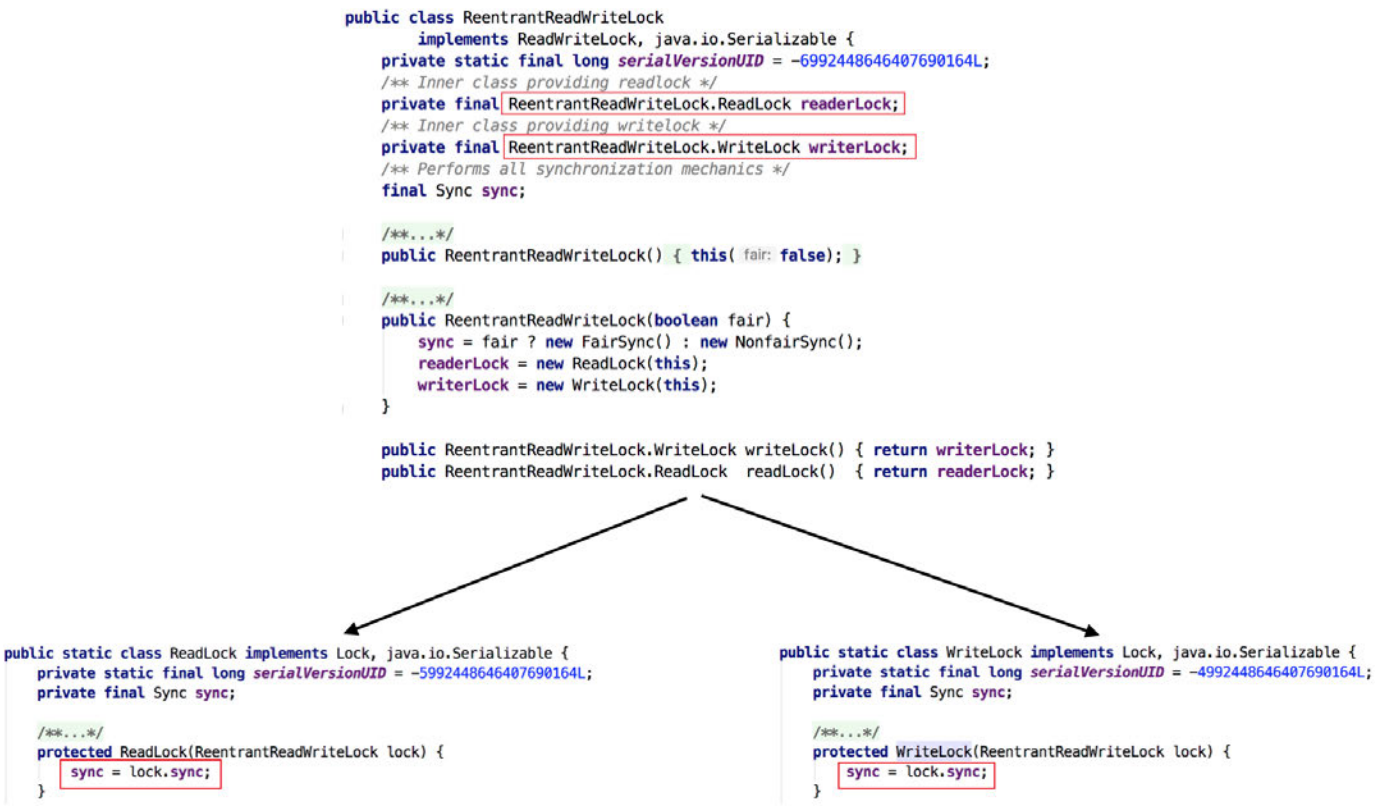
> 注意，即使用读写锁，在写线程访问时，所有的读线程和其它写线程均被阻塞。

排它锁也叫独享锁，如果线程 T 对数据 A 加上排它锁后，则其他线程不能再对 A 加任何类型的锁。获得排它锁的线程既能读数据又能修改数据。

与之对应的，就是共享锁，指该锁可被多个线程所持有。如果线程 T 对数据 A 加上共享锁后，则其他线程只能对 A 再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

下图为 ReentrantReadWriteLock 的部分源码：



我们看到ReentrantReadWriteLock有两把锁：ReadLock 和 WriteLock，由词知意，一个读锁一个写锁，合称“读写锁”。再进一步观察可以发现 ReadLock 和 WriteLock 是靠内部类 Sync 实现的锁。Sync 是 AQS 的一个子类，这种结构在CountDownLatch、Semaphore（后面会细讲，戳链接直达）、ReentrantLock（接下来会讲，戳链接直达）里面也都存在。

在 ReentrantReadWriteLock 里面，读锁和写锁的锁主体都是 Sync，但读锁和写锁的加锁方式不一样。读锁是共享锁，写锁是独享锁。读锁的共享锁可保证并发读非常高效，而读写、写读、写写的过程互斥，因为读锁和写锁是分离的。所以 ReentrantReadWriteLock 的并发性相比一般的互斥锁有了很大提升。

那读锁和写锁的具体加锁方式有什么区别呢?

在了解源码之前我们需要回顾一下其他知识。 在最开始提及 AQS 的时候我们也提到了 state 字段（int 类型，32 位），该字段用来描述有多少线程持有锁。

在独享锁中，这个值通常是 0 或者 1（如果是重入锁的话 state 值就是重入的次数），在共享锁中 state 就是持有锁的数量。但是在 ReentrantReadWriteLock 中有读、写两把锁，所以需要在一个整型变量 state 上分别描述读锁和写锁的数量（或者也可以叫状态）。

于是将 state 变量“按位切割”切分成了两个部分，高 16 位表示读锁状态（读锁个数），低 16 位表示写锁状态（写锁个数）。如下图所示：

32位																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
高十六位，读状态																低十六位，写状态															

了解了概念之后我们再来看代码，先看写锁的加锁源码：

```
protected final boolean tryAcquire(int acquires) {
  Thread current = Thread.currentThread();
```

```
  int c = getState(); // 取到当前锁的个数
  int w = exclusiveCount(c); // 取写锁的个数w
  if (c != 0) { // 如果已经有线程持有了锁(c!=0)
    // (Note: if c != 0 and w == 0 then shared count != 0)
    if (w == 0 || current != getExclusiveOwnerThread()) // 如果写线程数（w）为0（换言之存在读锁） 或者持有锁的线程不是当前线程就返回失败
      return false;
    if (w + exclusiveCount(acquires) > MAX_COUNT)     // 如果写入锁的数量大于最大数（65535，2的16次方-1）就抛出一个Error。
      throw new Error("Maximum lock count exceeded");
    // Reentrant acquire
    setState(c + acquires);
    return true;
  }
  if (writerShouldBlock() || !compareAndSetState(c, c + acquires)) // 如果当且写线程数为0，并且当前线程需要阻塞那么就返回失败；或者如果通过CAS增加写线程数失败也返回失败。
    return false;
  setExclusiveOwnerThread(current); // 如果c=0，w=0或者c>0，w>0（重入），则设置当前线程或锁的拥有者
  return true;
}
```

- 这段代码首先取到当前锁的个数 c，然后再通过 c 来获取写锁的个数 w。因为写锁是低 16 位，所以取低 16 位的最大值与当前的 c 做与运算（`int w = exclusiveCount©;`），高 16 位和 0 与运算后是 0，剩下的就是低位运算的值，同时也是持有写锁的线程数目。
- 在取到写锁线程的数目后，首先判断是否已经有线程持有了锁。如果已经有线程持有了锁(c!=0)，则查看当前写锁线程的数目，如果写线程数为 0（即此时存在读锁）或者持有锁的线程不是当前线程就返回失败（涉及到公平锁和非公平锁的实现）。
- 如果写入锁的数量大于最大数（65535，2 的 16 次方-1）就抛出一个 Error。
- 如果当前写线程数为 0（那么读线程也应该为 0，因为上面已经处理`c!=0`的情况），并且当前线程需要阻塞那么就返回失败；如果通过 CAS 增加写线程数失败也返回失败。
- 如果 c=0,w=0 或者 c>0,w>0（重入），则设置当前线程或锁的拥有者，返回成功！

`tryAcquire()`除了重入条件（当前线程为获取写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取，原因在于：必须确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。

因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。写锁的释放与 ReentrantLock 的释放过程基本类似，每次释放均减少写状态，当写状态为 0 时表示写锁已被释放，然后等待的读写线程才能够继续访问读写锁，同时前次写线程的修改对后续的读写线程可见。

接着是读锁的代码：

```
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
```

```
            return -1;                                          // 如果其他线程已经获取了写锁，则当前线程
获取读锁失败，进入等待状态
        int r = sharedCount(c);
        if (!readerShouldBlock() &&
            r < MAX_COUNT &&
            compareAndSetState(c, c + SHARED_UNIT)) {
            if (r == 0) {
                firstReader = current;
                firstReaderHoldCount = 1;
            } else if (firstReader == current) {
                firstReaderHoldCount++;
            } else {
                HoldCounter rh = cachedHoldCounter;
                if (rh == null || rh.tid != getThreadId(current))
                    cachedHoldCounter = rh = readHolds.get();
                else if (rh.count == 0)
                    readHolds.set(rh);
                rh.count++;
            }
            return 1;
        }
        return fullTryAcquireShared(current);
    }
```

可以看到在`tryAcquireShared(int unused)`方法中，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全，依靠 CAS 保证）增加读状态，成功获取读锁。读锁的每次释放（线程安全的，可能有多个读线程同时释放读锁）均减少读状态，减少的值是“`1<<16`”。所以读写锁才能实现读读的过程共享，而读写、写读、写写的过程互斥。

此时，我们再回头看一下互斥锁 ReentrantLock 中公平锁和非公平锁的加锁源码：

公平锁

```
/**...*/
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

非公平锁

```
/**...*/
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

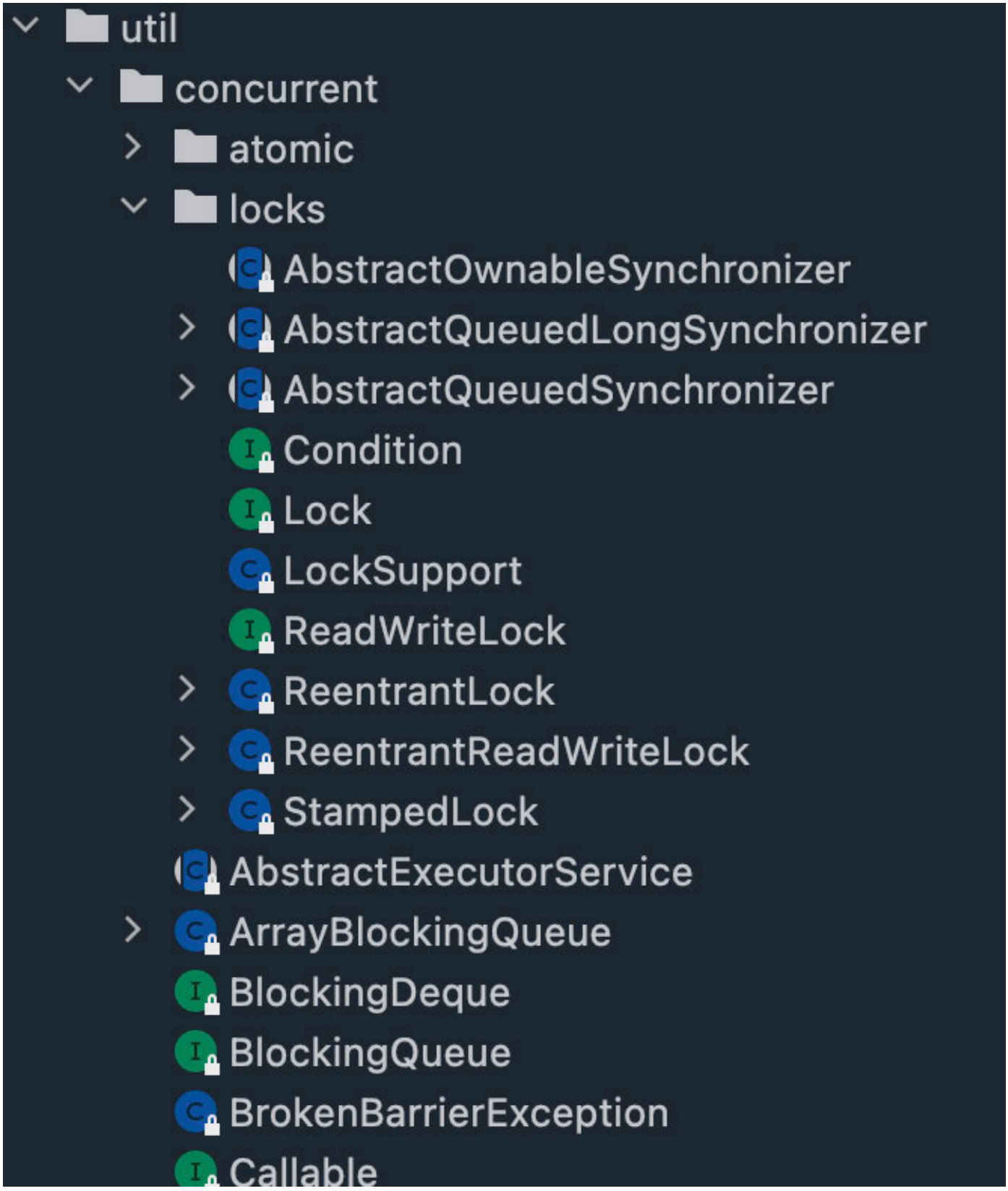
我们发现公平锁和非公平锁如果在当前线程不是拥有锁的线程时，就不能添加锁。
所以它们添加的都是独享锁！

我们发现在 ReentrantLock 虽然有公平锁和非公平锁两种，但是它们添加的都是独享锁。根据源码所示，当某一个线程调用 lock 方法获取锁时，如果同步资源没有被其他线程锁住，那么当前线程在使用 CAS 更新 state 成功后就会成功抢占该资源。而如果公共资源被占用且不是被当前线程占用，那么就会加锁失败。所以可以确定 ReentrantLock 无论读操作还是写操作，添加的锁都是都是独享锁。

综上，只有 synchronized 是远远不能满足多样化业务对锁的要求的。接下来我们介绍一下 JDK 中有关锁的一些接口和类。

JUC 包下的锁

众所周知，JDK 中关于并发的类大多都在 JUC 包下。



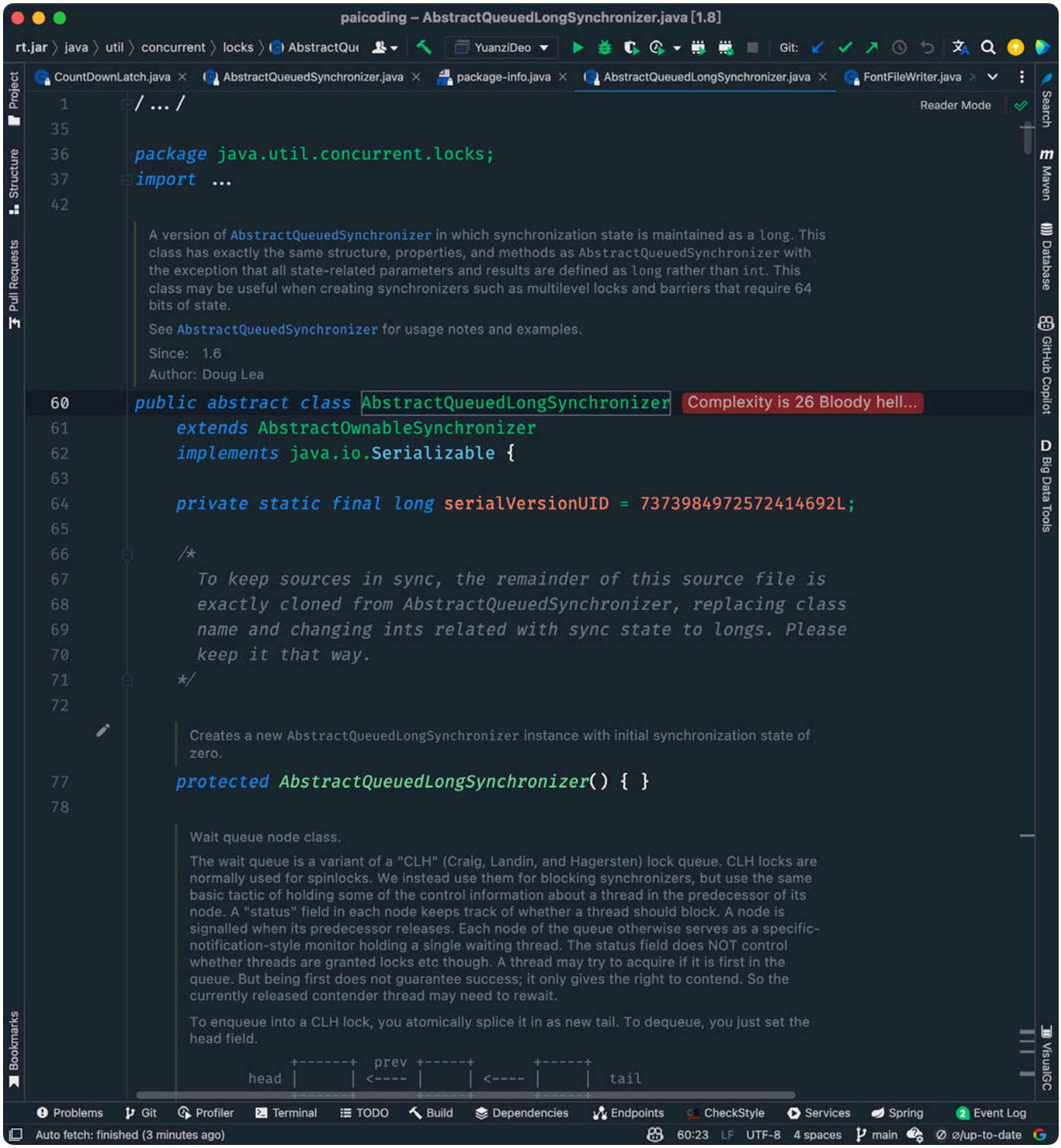
看名字就知道，locks 包是提供一些并发锁的工具类的。前面我们介绍的 AQS（AbstractQueuedSynchronizer）就是在这个包下。

抽象类 AQS/AQLS/AOS

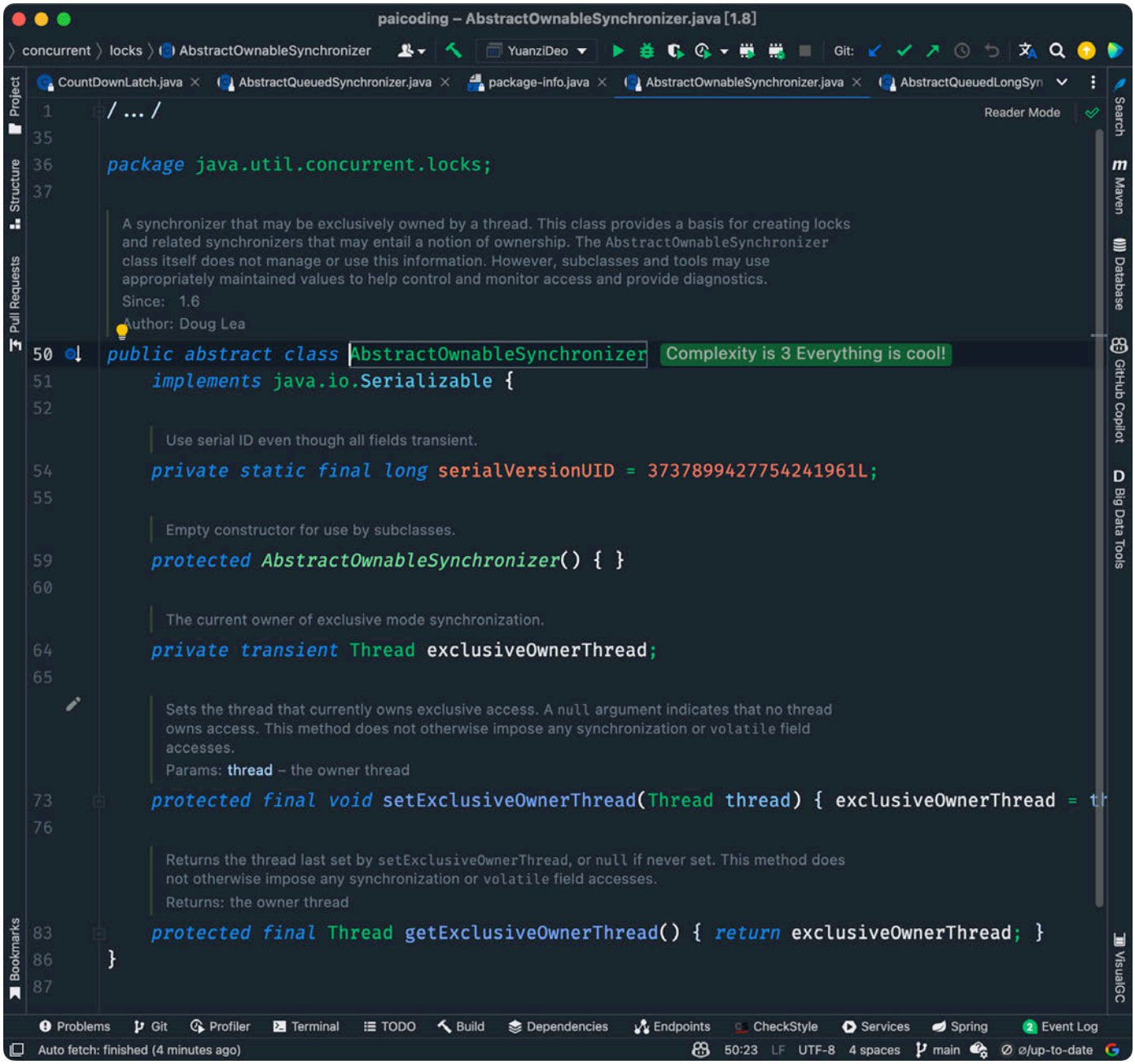
这三个抽象类有一定的关系，所以这里放到一起讲。

首先我们来看**AQS**（AbstractQueuedSynchronizer），它是在 JDK 1.5 发布的，提供了一个“队列同步器”的基本功能实现。

AQS 里面的“资源”是用一个`int`类型的数据来表示的，有时候业务需求的资源数超出了`int`的范围，所以在 JDK 1.6 中，多了一个**AQLS**（AbstractQueuedLongSynchronizer）。它的代码跟 AQS 几乎一样，只是把资源的类型变成了`long`类型。



AQS 和 AQLS 都继承了一个类叫**AOS**（AbstractOwnableSynchronizer）。这个类也是在 JDK 1.6 中出现的。



这个类只有几行简单的代码。从源码类上的注释可以知道，它是用于表示锁与持有者之间的关系（独占模式）。可以看一下它的主要方法：

```
// 独占模式，锁的持有者
private transient Thread exclusiveOwnerThread;

// 设置锁持有者
protected final void setExclusiveOwnerThread(Thread t) {
    exclusiveOwnerThread = t;
}

// 获取锁的持有线程
protected final Thread getExclusiveOwnerThread() {
    return exclusiveOwnerThread;
}
```

接口 Condition/Lock/ReadWriteLock

locks 包下共有三个接口：`Condition`、`Lock`、`ReadWriteLock`。

其中，Lock 和 ReadWriteLock 从名字就可以看得出来，分别是锁和读写锁的意思。Lock 接口里面有一些获取锁和释放锁的方法声明，而 ReadWriteLock 里面只有两个方法，分别返回“读锁”和“写锁”：

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

Lock 接口中有一个方法可以获得一个Condition（后面会细讲，戳链接直达）：

```
Condition newCondition();
```

之前我们提到过每个对象都可以用`Object`的**wait/notify**方法来实现**等待/通知机制**。而 Condition 接口也提供了类似 Object 的方法，可以配合**Lock**来实现等待/通知模式。

既然有 Object 的监视器方法了，为什么还要用 Condition 呢？这里有一个简单的对比：

对比项	Object 监视器	Condition
前置条件	获取对象的锁	调用 Lock.lock 获取锁，调用 Lock.newCondition 获取 Condition 对象
调用方式	直接调用，比如 `object.notify()`	直接调用，比如 `condition.await()`
等待队列的个数	一个	多个
当前线程释放锁进入等待状态	支持	支持
当前线程释放锁进入等待状态，在等待状态中不中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入等待状态直到将来的某个时间	不支持	支持
唤醒等待队列中的一个线程	支持	支持
唤醒等待队列中的全部线程	支持	支持

Condition 和 Object 的 wait/notify 基本相似。其中，Condition 的 await 方法对应的是 Object 的 wait 方法，而 Condition 的**signal/signalAll**方法则对应 Object 的 notify/`notifyAll()`。但 Condition 类似于 Object 的等待/通知机制的加强版。我们来看看主要的方法：

方法名称	描述
`await()`	当前线程进入等待状态直到被通知（signal）或者中断；当前线程进入运行状态并从 `await()` 方法返回的场景包括：（1）其他线程调用相同 Condition 对象的 signal/signalAll 方法，并且当前线程被唤醒；（2）其他线程调用 interrupt 方法中断当前线程；
`awaitUninterruptibly()`	当前线程进入等待状态直到被通知，在此过程中对中断信号不敏感，不支持中断当前线程
awaitNanos(long)	当前线程进入等待状态，直到被通知、中断或者超时。如果返回值小于等于 0，可以认定就是超时了
awaitUntil(Date)	当前线程进入等待状态，直到被通知、中断或者超时。如果没到指定时间被通知，则返回 true，否则返回 false
signal()	唤醒一个等待在 Condition 上的线程，被唤醒的线程在方法返回前必须获得与 Condition 对象关联的锁
signalAll()	唤醒所有等待在 Condition 上的线程，能够从 await()等方法返回的线程必须先获得与 Condition 对象关联的锁

可重入锁ReentrantLock

ReentrantLock（接下来细讲，戳链接直达）是 Lock 接口的默认实现，实现了锁的基本功能。

从名字上看，它是一个“可重入”锁，从源码上看，它内部有一个抽象类`Sync`，继承了 AQS，自己实现了一个同步器。

同时，ReentrantLock 内部有两个非抽象类`NonfairSync`和`FairSync`，它们都继承了 Sync。从名字上可以看得出，分别是”非公平同步器“和”公平同步器“的意思。这意味着 ReentrantLock 可以支持”公平锁“和”非公平锁“。

通过看这两个同步器的源码可以发现，它们的实现都是”独占“的。都调用了 AOS 的`setExclusiveOwnerThread`方法，所以 ReentrantLock 的锁是”独占“的，也就是说，它的锁都是”排他锁“，不能共享。

在 ReentrantLock 的构造方法里，可以传入一个`boolean`类型的参数，来指定它是否是一个公平锁，默认情况下是非公平的。这个参数一旦实例化后就不能修改，只能通过`isFair()`方法来查看。

来看一个 ReentrantLock 的简单示例：

```
public class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock(); // 获取锁
        try {
            count++;
            System.out.println("增量 " + Thread.currentThread().getName() + ": " +
count);
        } finally {
```

```
            lock.unlock(); // 释放锁
        }
    }

    public static void main(String[] args) {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("最终结果：" + counter.count);
    }
}
```

在这个示例中，Counter 类使用了一个 ReentrantLock 来保护 count 变量的访问。increment 方法首先获取锁，然后增加计数，并在 finally 块中释放锁。这确保了即使方法中抛出异常，锁也会被正确释放。

在 main 方法中，我们创建了两个线程来并发执行 increment 操作。由于使用了锁，因此对 count 变量的访问是串行化的，结果是正确的。

这个示例展示了 ReentrantLock 的基本用法。与 synchronized 关键字相比，ReentrantLock 提供了更高的灵活性，例如可中断的锁获取、公平锁选项、锁的定时获取等。

来看一下最终输出结果：

读写锁ReentrantReadWriteLock

ReentrantReadWriteLock（后面会细讲，戳链接直达） 是 ReadWriteLock 接口的默认实现。它与 ReentrantLock 的功能类似，同样是可重入的，支持非公平锁和公平锁。不同的是，它还支持"读写锁"。

ReentrantReadWriteLock 内部的结构大概是这样：

```
// 内部结构
private final ReentrantReadWriteLock.ReadLock readerLock;
private final ReentrantReadWriteLock.WriteLock writerLock;
final Sync sync;
abstract static class Sync extends AbstractQueuedSynchronizer {
    // 具体实现
}
static final class NonfairSync extends Sync {
    // 具体实现
}
static final class FairSync extends Sync {
    // 具体实现
}
public static class ReadLock implements Lock, java.io.Serializable {
```

```
    private final Sync sync;
    protected ReadLock(ReentrantReadWriteLock lock) {
            sync = lock.sync;
    }
    // 具体实现
}
public static class WriteLock implements Lock, java.io.Serializable {
    private final Sync sync;
    protected WriteLock(ReentrantReadWriteLock lock) {
            sync = lock.sync;
    }
    // 具体实现
}

// 构造方法，初始化两个锁
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}

// 获取读锁和写锁的方法
public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }
```

可以看到，它同样是内部维护了两个同步器。且维护了两个 Lock 的实现类 ReadLock 和 WriteLock。从源码可以发现，这两个内部类用的是外部类的同步器。

来看一下 ReentrantReadWriteLock 的使用示例：

```
public class SharedResource {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private int data = 0;

    public void write(int value) {
        lock.writeLock().lock(); // 获取写锁
        try {
            data = value;
            System.out.println("写 " + Thread.currentThread().getName() + ": " + data);
        } finally {
            lock.writeLock().unlock(); // 释放写锁
        }
    }

    public void read() {
        lock.readLock().lock(); // 获取读锁
        try {
            System.out.println("读 " + Thread.currentThread().getName() + ": " + data);
        } finally {
```

```
            lock.readLock().unlock(); // 释放读锁
        }
    }

    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();

        // 创建读线程
        Thread readThread1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                sharedResource.read();
            }
        });

        Thread readThread2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                sharedResource.read();
            }
        });

        // 创建写线程
        Thread writeThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                sharedResource.write(i);
            }
        });

        readThread1.start();
        readThread2.start();
        writeThread.start();

        try {
            readThread1.join();
            readThread2.join();
            writeThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在上述代码中，我们定义了一个 SharedResource 类，该类使用 ReentrantReadWriteLock 来保护其内部数据。write 方法获取写锁，并更新共享数据。read 方法获取读锁，并读取共享数据。

在 main 方法中，我们创建了两个读线程和一个写线程。由于 ReentrantReadWriteLock 允许多个读取操作同时进行，因此读线程可以同时运行。然而，写入操作会被串行化，并且在写入操作进行时，读取操作将被阻塞。

来看一下输出结果：

```
读 Thread-0: 0
```

```
读 Thread-1: 0
写 Thread-2: 0
写 Thread-2: 1
写 Thread-2: 2
写 Thread-2: 3
写 Thread-2: 4
读 Thread-0: 4
读 Thread-1: 4
读 Thread-0: 4
读 Thread-1: 4
读 Thread-0: 4
读 Thread-1: 4
读 Thread-0: 4
读 Thread-1: 4
```

ReentrantReadWriteLock 实现了读写锁，但它有一个小弊端，就是在“写”操作的时候，其它线程不能写也不能读。我们称这种现象为“写饥饿”，将在下文的 StampedLock 类继续讨论这个问题。

锁王StampedLock

`StampedLock` 类是 Java 8 才发布的，也是 Doug Lea 大神所写，有人称它为锁的性能之王。

StampedLock 没有实现 Lock 接口和 ReadWriteLock 接口，但它实现了“读写锁”的功能，并且性能比 ReentrantReadWriteLock 更高。StampedLock 还把读锁分为了“乐观读锁”和“悲观读锁”两种。

前面提到了 ReentrantReadWriteLock 会发生“写饥饿”的现象，但 StampedLock 不会。它是怎么做到的呢?

它的核心思想在于，**在读的时候如果发生了写，应该通过重试的方式来获取新的值，而不应该阻塞写操作。这种模式也就是典型的无锁编程思想，和 CAS 自旋的思想一样**。这种操作方式决定了 StampedLock 在读线程非常多而写线程非常少的场景下非常适用，同时还避免了写饥饿情况的发生。

我们来分析一下官方提供的用法（在 JDK 源码类声明的上方或 Javadoc 里可以找到）。

来看一下。

```
class Point {
   private double x, y;
```

```
private final StampedLock sl = new StampedLock();

// 写锁的使用
void move(double deltaX, double deltaY) {
  long stamp = sl.writeLock(); // 获取写锁
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    sl.unlockWrite(stamp); // 释放写锁
  }
}

// 乐观读锁的使用
double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead(); // 获取乐观读锁
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) { // //检查乐观读锁后是否有其他写锁发生，有则返回false
     stamp = sl.readLock(); // 获取一个悲观读锁
     try {
       currentX = x;
       currentY = y;
     } finally {
        sl.unlockRead(stamp); // 释放悲观读锁
     }
  }
  return Math.sqrt(currentX * currentX + currentY * currentY);
}

// 悲观读锁以及读锁升级写锁的使用
void moveIfAtOrigin(double newX, double newY) {
  long stamp = sl.readLock(); // 悲观读锁
  try {
    while (x == 0.0 && y == 0.0) {
      // 读锁尝试转换为写锁：转换成功后相当于获取了写锁，转换失败相当于有写锁被占用
      long ws = sl.tryConvertToWriteLock(stamp);

      if (ws != 0L) { // 如果转换成功
        stamp = ws; // 读锁的票据更新为写锁的
        x = newX;
        y = newY;
        break;
      }
      else { // 如果转换失败
        sl.unlockRead(stamp); // 释放读锁
        stamp = sl.writeLock(); // 强制获取写锁
      }
    }
  } finally {
```

```
            sl.unlock(stamp); // 释放所有锁
        }
    }
}
```

乐观读锁的意思就是先假定在这个锁获取期间，共享变量不会被改变，既然假定不会被改变，那就不需要上锁。

在获取乐观读锁之后进行了一些操作，然后又调用了 validate 方法，这个方法就是用来验证 tryOptimisticRead 之后，是否有写操作执行过，如果有，则获取一个悲观读锁，这里的悲观读锁和 ReentrantReadWriteLock 中的读锁类似，也是个共享锁。

可以看到，StampedLock 获取锁会返回一个`long`类型的变量，释放锁的时候再把这个变量传进去。简单看看源码：

```
// 用于操作state后获取stamp的值
private static final int LG_READERS = 7;
private static final long RUNIT = 1L;               //0000 0000 0001
private static final long WBIT  = 1L << LG_READERS; //0000 1000 0000
private static final long RBITS = WBIT - 1L;        //0000 0111 1111
private static final long RFULL = RBITS - 1L;       //0000 0111 1110
private static final long ABITS = RBITS | WBIT;     //0000 1111 1111
private static final long SBITS = ~RBITS;           //1111 1000 0000

// 初始化时state的值
private static final long ORIGIN = WBIT << 1;       //0001 0000 0000

// 锁共享变量state
private transient volatile long state;
// 读锁溢出时用来存储多出的读锁
private transient int readerOverflow;
```

StampedLock 用这个 long 类型的变量的前 7 位（LG_READERS）来表示读锁，每获取一个悲观读锁，就加 1（RUNIT），每释放一个悲观读锁，就减 1。而悲观读锁最多只能装 128 个（7 位限制），很容易溢出，所以用一个 int 类型的变量来存储溢出的悲观读锁。

写锁用 state 变量剩下的位来表示，每次获取一个写锁，就加 0000 1000 0000（WBIT）。需要注意的是，**写锁在释放的时候，并不是减 WBIT，而是再加 WBIT**。这是为了**让每次写锁都留下痕迹**，解决 CAS 中的 ABA 问题，也为**乐观锁检查变化**validate 方法提供基础。

乐观读锁就比较简单了，并没有真正改变 state 的值，而是在获取锁的时候记录 state 的写状态，在操作完成后去检查 state 的写状态部分是否发生变化，上文提到了，每次写锁都会留下痕迹，也是为了这里乐观锁检查变化提供方便。

总的来说，StampedLock 的性能是非常优异的，基本上可以取代 ReentrantReadWriteLock。我们来一个 StampedLock 和 ReentrantReadWriteLock 的对比使用示例。

ReentrantReadWriteLock：

```
public class SharedResourceWithReentrantReadWriteLock {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

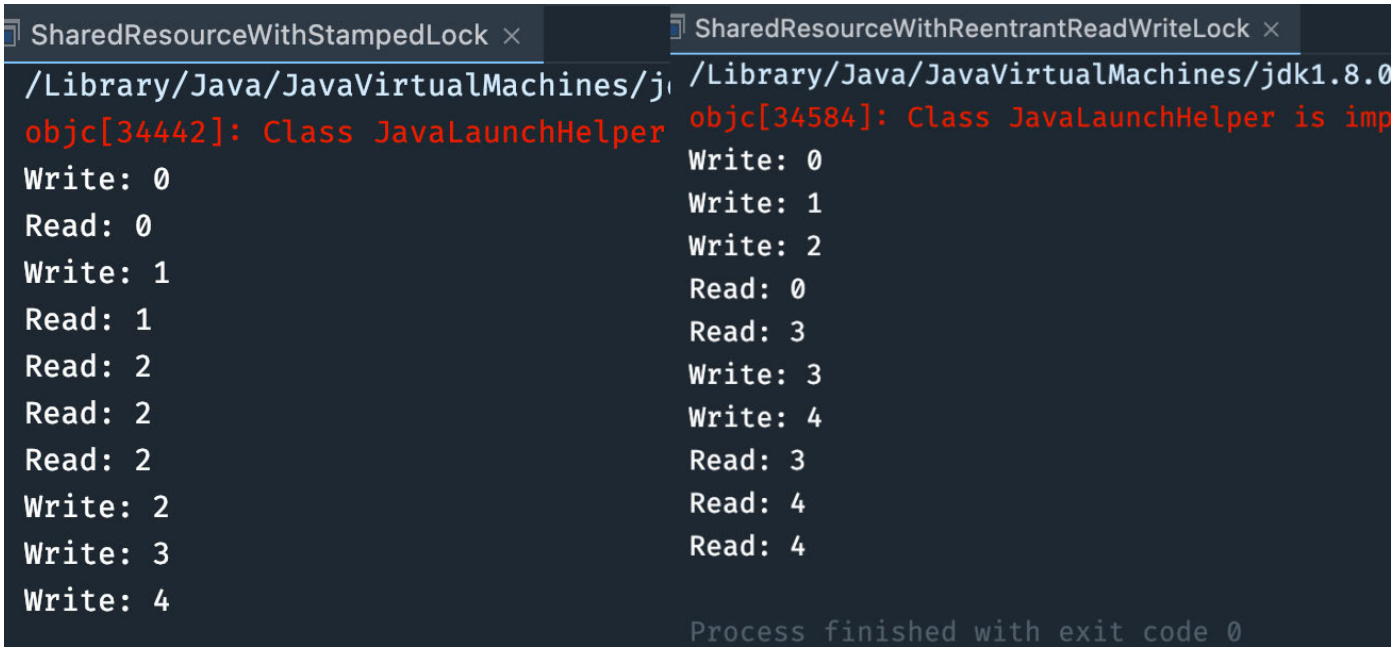
```
    private int data = 0;

    public void write(int value) {
        lock.writeLock().lock();
        try {
            data = value;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int read() {
        lock.readLock().lock();
        try {
            return data;
        } finally {
            lock.readLock().unlock();
        }
    }

    public static void main(String[] args) {
        SharedResourceWithReentrantReadWriteLock sharedResource = new
SharedResourceWithReentrantReadWriteLock();

        Thread writer = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                sharedResource.write(i);
                System.out.println("Write: " + i);
            }
        });

        Thread reader = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                int value = sharedResource.read();
                System.out.println("Read: " + value);
            }
        });

        writer.start();
        reader.start();
    }
}
```

StampedLock：

```
public class SharedResourceWithStampedLock {
    private final StampedLock sl = new StampedLock();
    private int data = 0;
```

```
    public void write(int value) {
        long stamp = sl.writeLock();
        try {
            data = value;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    public int read() {
        long stamp = sl.tryOptimisticRead();
        int currentData = data;
        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                currentData = data;
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return currentData;
    }

    public static void main(String[] args) {
        SharedResourceWithStampedLock sharedResource = new
SharedResourceWithStampedLock();

        Thread writer = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                sharedResource.write(i);
                System.out.println("Write: " + i);
            }
        });

        Thread reader = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                int value = sharedResource.read();
                System.out.println("Read: " + value);
            }
        });

        writer.start();
        reader.start();
    }
}
```

来看一下输出结果的对比。



1、可重入性：ReentrantReadWriteLock 支持可重入，即在一个线程中可以多次获取读锁或写锁。StampedLock 则不支持可重入。

2、乐观读锁：StampedLock 提供了乐观读锁机制，允许一个线程在没有任何写入操作发生的情况下读取数据，从而提高了性能。而 ReentrantReadWriteLock 没有提供这样的机制。

3、锁降级：StampedLock 提供了从写锁到读锁的降级功能，这在某些场景下可以提供额外的灵活性。ReentrantReadWriteLock 不直接提供这样的功能。

4、API 复杂性：由于提供了乐观读锁和锁降级功能，StampedLock 的 API 相对复杂一些，需要更小心地使用以避免死锁和其他问题。ReentrantReadWriteLock 的 API 相对更直观和容易使用。

综上所述，StampedLock 提供了更高的性能和灵活性，但也带来了更复杂的使用方式。ReentrantReadWriteLock 则相对简单和直观，特别适用于没有高并发读的场景。

JUC 包下的其他工具类

locks 包下的锁接口和锁类介绍完了，我们这里再讲一些 JUC 包下的其他工具类，比如 Semaphore、CountDownLatch、CyclicBarrier、Exchanger、Phaser 等（这些在通信工具类中也会细讲）。

Semaphore

Semaphore 是一个计数信号量，它的作用是限制可以访问某些资源（物理或逻辑的）的线程数目。Semaphore 的构造方法可以指定信号量的数目，也可以指定是否是公平的。

Semaphore 有两个主要的方法：`acquire()`和`release()`。`acquire()`方法会尝试获取一个信号量，如果获取不到，就会阻塞当前线程，直到有线程释放信号量。`release()`方法会释放一个信号量，释放之后，会唤醒一个等待的线程。

Semaphore 还有一个`tryAcquire()`方法，它会尝试获取一个信号量，如果获取不到，就会返回 false，不会阻塞当前线程。

Semaphore 用来控制同时访问某个特定资源的操作数量，它并不保证线程安全，所以要保证线程安全，还需要加上同步锁。

来看一个 Semaphore 的使用示例：

```
public class ResourcePool {
    private final Semaphore semaphore;

    public ResourcePool(int limit) {
        this.semaphore = new Semaphore(limit);
    }

    public void useResource() {
        try {
            semaphore.acquire();
            // 使用资源
            System.out.println("资源开始使用了 " + Thread.currentThread().getName());
            Thread.sleep(1000); // 模拟资源使用时间
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
            System.out.println("资源释放了 " + Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        ResourcePool pool = new ResourcePool(3); // 限制3个线程同时访问资源

        for (int i = 0; i < 10; i++) {
            new Thread(pool::useResource).start();
        }
    }
}
```

来看一下输出结果：

```
资源开始使用了 Thread-0
资源开始使用了 Thread-2
资源开始使用了 Thread-1
资源释放了 Thread-0
资源释放了 Thread-2
资源开始使用了 Thread-4
资源开始使用了 Thread-3
资源开始使用了 Thread-5
资源释放了 Thread-1
```

```
资源开始使用了 Thread-6
资源开始使用了 Thread-8
资源开始使用了 Thread-7
资源释放了 Thread-4
资源释放了 Thread-3
资源释放了 Thread-5
资源释放了 Thread-8
资源释放了 Thread-6
资源开始使用了 Thread-9
资源释放了 Thread-7
资源释放了 Thread-9
```

CountDownLatch

CountDownLatch 是一个同步工具类，它允许一个或多个线程一直等待，直到其他线程的操作执行完后再执行。

CountDownLatch 有一个计数器，可以通过`countDown()`方法对计数器的数目进行减一操作，也可以通过`await()`方法来阻塞当前线程，直到计数器的值为 0。

CountDownLatch 一般用来控制线程等待，它可以让某个线程一直等待直到倒计时结束，再开始执行。

来看一个 CountDownLatch 的使用示例：

```
public class InitializationDemo {

    public static void main(String[] args) throws InterruptedException {
        // 创建一个倒计数为 3 的 CountDownLatch
        CountDownLatch latch = new CountDownLatch(3);

        Thread service1 = new Thread(new Service("服务 1", 2000, latch));
        Thread service2 = new Thread(new Service("服务 2", 3000, latch));
        Thread service3 = new Thread(new Service("服务 3", 4000, latch));

        service1.start();
        service2.start();
        service3.start();

        // 等待所有服务初始化完成
        latch.await();
        System.out.println("所有服务都准备好了");
    }

    static class Service implements Runnable {
        private final String name;
        private final int timeToStart;
        private final CountDownLatch latch;

        public Service(String name, int timeToStart, CountDownLatch latch) {
            this.name = name;
            this.timeToStart = timeToStart;
```

```
            this.latch = latch;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(timeToStart);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(name + " 准备好了");
            latch.countDown(); // 减少倒计数
        }
    }
}
```

在这个示例中，我们有三个服务，每个服务都在一个单独的线程中启动，并需要一些时间来初始化。主线程使用 CountDownLatch 等待这三个服务全部启动完成后，再继续执行。每个服务启动完毕后都会调用 `countDown()` 方法。主线程通过调用 `await()` 方法等待，直到倒计数变为零，然后继续执行。

来看运行结果：

```
服务 1 准备好了
服务 2 准备好了
服务 3 准备好了
所有服务都准备好了
```

CyclicBarrier

CyclicBarrier 是一个同步工具类，它允许一组线程互相等待，直到到达某个公共屏障点（common barrier point）。

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 sheet 保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

CyclicBarrier 的计数器可以通过 `reset()` 方法重置，所以它能处理循环使用的场景。比如，我们将一个大任务分成 10 个小任务，用 10 个线程分别执行这 10 个小任务，当 10 个小任务都执行完之后，再合并这 10 个小任务的结果，这个时候就可以用 CyclicBarrier 来实现。

CyclicBarrier 还有一个有参构造方法，可以指定一个 Runnable，这个 Runnable 会在 CyclicBarrier 的计数器为 0 的时候执行，用来完成更复杂的任务。

来看一下使用示例用：

```
public class CyclicBarrierDemo {

    public static void main(String[] args) {
        int numberOfThreads = 3; // 线程数量
```

```
        CyclicBarrier barrier = new CyclicBarrier(numberOfThreads, () -> {
            // 当所有线程都到达障碍点时执行的操作
            System.out.println("所有线程都已到达屏障，进入下一阶段");
        });

        for (int i = 0; i < numberOfThreads; i++) {
            new Thread(new Task(barrier), "Thread " + (i + 1)).start();
        }
    }

    static class Task implements Runnable {
        private final CyclicBarrier barrier;

        public Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " 正在屏障处等待");
                barrier.await(); // 等待所有线程到达障碍点
                System.out.println(Thread.currentThread().getName() + " 已越过屏障.");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

输出结果如下所示：

```
Thread 1 正在屏障处等待
Thread 3 正在屏障处等待
Thread 2 正在屏障处等待
所有线程都已到达屏障，进入下一阶段
Thread 2 已越过屏障.
Thread 1 已越过屏障.
Thread 3 已越过屏障.
```

Exchanger

Exchanger 是一个用于线程间协作的工具类。Exchanger 用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过 exchange 方法交换数据，如果第一个线程先执行 exchange 方法，它会一直等待第二个线程也执行 exchange 方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

Exchanger 可以用于遗传算法、校对工作和数据同步等场景。

来看一个使用示例：

```
public class ExchangerDemo {

    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<>();

        new Thread(() -> {
            try {
                String data1 = "data1";
                System.out.println(Thread.currentThread().getName() + " 正在把 " + data1
+ " 交换出去");
                Thread.sleep(1000); // 模拟线程处理耗时
                String data2 = exchanger.exchange(data1);
                System.out.println(Thread.currentThread().getName() + " 交换到了 " +
data2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread 1").start();

        new Thread(() -> {
            try {
                String data1 = "data2";
                System.out.println(Thread.currentThread().getName() + " 正在把 " + data1
+ " 交换出去");
                Thread.sleep(2000); // 模拟线程处理耗时
                String data2 = exchanger.exchange(data1);
                System.out.println(Thread.currentThread().getName() + " 交换到了 " +
data2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread 2").start();
    }
}
```

输出结果如下所示：

```
Thread 1 正在把 data1 交换出去
Thread 2 正在把 data2 交换出去
Thread 2 交换到了 data1
Thread 1 交换到了 data2
```

Phaser

Phaser 是一个同步工具类，它可以让多个线程在某个时刻一起完成任务。

Phaser 可以理解为一个线程的计数器，它可以将这个计数器加一或减一。当这个计数器的值为 0 的时候，所有调用`await()`方法而在等待的线程就会继续执行。

Phaser 的计数器可以被动态地更新，也可以被动态地增加或减少。Phaser 还提供了一些方法来帮助我们更好地控制线程的到达。

来看一个使用示例：

```
public class PhaserDemo {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(3); // 3 个线程共同完成任务

        new Thread(new Task(phaser), "Thread 1").start();
        new Thread(new Task(phaser), "Thread 2").start();
        new Thread(new Task(phaser), "Thread 3").start();
    }

    static class Task implements Runnable {
        private final Phaser phaser;

        public Task(Phaser phaser) {
            this.phaser = phaser;
        }

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + " 完成了第一步操作");
            phaser.arriveAndAwaitAdvance(); // 等待其他线程完成第一步操作
            System.out.println(Thread.currentThread().getName() + " 完成了第二步操作");
            phaser.arriveAndAwaitAdvance(); // 等待其他线程完成第二步操作
            System.out.println(Thread.currentThread().getName() + " 完成了第三步操作");
            phaser.arriveAndAwaitAdvance(); // 等待其他线程完成第三步操作
        }
    }
}
```

输出结果如下所示：

```
Thread 1 完成了第一步操作
Thread 2 完成了第一步操作
Thread 3 完成了第一步操作
Thread 3 完成了第二步操作
Thread 1 完成了第二步操作
Thread 2 完成了第二步操作
Thread 1 完成了第三步操作
Thread 3 完成了第三步操作
Thread 2 完成了第三步操作
```

小结

本文介绍了 JUC 包下的锁接口和锁类，包括 Lock、ReadWriteLock、Condition、ReentrantLock、ReentrantReadWriteLock、StampedLock 等。还介绍了 JUC 包下的其他工具类，包括 Semaphore、CountDownLatch、CyclicBarrier、Exchanger、Phaser 等。

JUC 包下的锁接口和锁类，可以说是 Java 并发编程的核心，也是面试中经常会问到的知识点。所以，一定要掌握好。

> 编辑：沉默王二，编辑前的内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐；
> 还有一部分内容来源于美团点评后端工程师家琪的这篇文章，强烈推荐。

第十五节：重入锁 ReentrantLock

ReentrantLock 重入锁，是实现Lock 接口的一个类，也是在实际编程中使用频率很高的一个锁，**支持重入性，表示能够对共享资源重复加锁，即当前线程获取该锁后再次获取不会被阻塞。**

要想支持重入性，就要解决两个问题：

1. 在线程获取锁的时候，如果已经获取锁的线程是当前线程的话则直接再次获取成功；
2. 由于锁会被获取 n 次，那么只有锁在被释放同样的 n 次之后，该锁才算是完全释放成功。

我们知道，同步组件主要是通过重写 AQS 的几个 protected 方法来表达自己的同步语义。

ReentrantLock 的源码分析

针对第一个问题，我们来看看 ReentrantLock 是怎样实现的，以非公平锁为例，判断当前线程能否获得锁为例，核心方法为内部类 Sync 的 nonfairTryAcquire 方法：

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    //1. 如果该锁未被任何线程占有，该锁能被当前线程获取
  if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
  //2.若被占有，检查占有线程是否是当前线程
    else if (current == getExclusiveOwnerThread()) {
    // 3. 再次获取，计数加一
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

这段代码的逻辑很简单，具体请看注释。为了支持重入性，在第二步增加了处理逻辑，如果该锁已经被线程占有了，会继续检查占有线程是否为当前线程，如果是的话，同步状态加 1 返回 true，表示可以再次获取成功。每次重新获取都会对同步状态进行加一的操作，那么释放的时候处理思路是怎样的呢？（依然还是以非公平锁为例）核心方法为 tryRelease：

```
protected final boolean tryRelease(int releases) {
  //1. 同步状态减1
    int c = getState() - releases;
```

```
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
    //2. 只有当同步状态为0时，锁成功被释放，返回true
        free = true;
        setExclusiveOwnerThread(null);
    }
  // 3. 锁未被完全释放，返回false
    setState(c);
    return free;
}
```

代码的逻辑请看注释，需要注意的是，重入锁的释放必须得等到同步状态为 0 时锁才算成功释放，否则锁仍未释放。如果锁被获取了 n 次，释放了 n-1 次，该锁未完全释放返回 false，只有被释放 n 次才算成功释放，返回 true。到现在我们可以理清 ReentrantLock 重入性的实现了，也就是理解了同步语义的第一条。

ReentrantLock 支持两种锁：**公平锁**和**非公平锁。何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求上的绝对时间顺序，满足 FIFO**。ReentrantLock 的构造方法无参时是构造非公平锁，源码为：

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

另外还提供了一种方式，可传入一个 boolean 值，true 时为公平锁，false 时为非公平锁，源码为：

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

在非公平锁获取时（nonfairTryAcquire 方法），只是简单的获取了一下当前状态然后做了一些逻辑处理，并没有考虑到当前同步队列中线程等待的情况。

我们来看看公平锁的处理逻辑是怎样的，核心方法为：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
```

```
            if (nextc < 0)
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
```

这段代码的逻辑与 nonfairTryAcquire 基本上一致，唯一的不同在于增加了 hasQueuedPredecessors 的逻辑判断，从方法名就可以知道该方法用来判断当前节点在同步队列中是否有前驱节点的，如果有前驱节点，说明有线程比当前线程更早的请求资源，根据公平性，当前线程请求资源失败。如果当前节点没有前驱节点，才有做后面逻辑判断的必要性。

公平锁每次都是从同步队列中的第一个节点获取到锁，而非公平性锁则不一定，有可能刚释放锁的线程能再次获取到锁。

ReentrantLock 的使用

ReentrantLock 的使用方式与 synchronized 关键字类似，都是通过加锁和释放锁来实现同步的。我们来看看 ReentrantLock 的使用方式，以非公平锁为例：

```
public class ReentrantLockTest {
    private static final ReentrantLock lock = new ReentrantLock();
    private static int count = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                lock.lock();
                try {
                    count++;
                } finally {
                    lock.unlock();
                }
            }
        });
        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                lock.lock();
                try {
                    count++;
                } finally {
                    lock.unlock();
                }
            }
        });
        thread1.start();
        thread2.start();
        thread1.join();
```

```
        thread2.join();
        System.out.println(count);
    }
}
```

代码很简单，两个线程分别对 count 变量进行 10000 次累加操作，最后输出 count 的值。我们来看看运行结果：

```
20000
```

可以看到，两个线程对 count 变量进行了 20000 次累加操作，说明 ReentrantLock 是支持重入性的。我们再来看看公平锁的使用方式，只需要将 ReentrantLock 的构造方法改为公平锁即可：

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

运行结果为：

```
20000
```

可以看到，公平锁的运行结果与非公平锁的运行结果一致，这是因为公平锁的实现方式与非公平锁的实现方式基本一致，只是在获取锁时增加了判断当前节点是否有前驱节点的逻辑判断。

- 公平锁: 按照线程请求锁的顺序获取锁，即先到先得。
- 非公平锁: 线程获取锁的顺序可能与请求锁的顺序不同，可能导致某些线程获取锁的速度较快。

需要注意的是，使用 ReentrantLock 时，锁必须在 try 代码块开始之前获取，并且加锁之前不能有异常抛出，否则在 finally 块中就无法释放锁（ReentrantLock 的锁必须在 finally 中手动释放）。

错误❎示例：

```
Lock lock = new XxxLock();
// ...
try {
    // 如果在此抛出异常，会直接执行 finally 块的代码
    doSomething();
    // 不管锁是否成功，finally 块都会执行
    lock.lock();
    doOthers();

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

正确✅示例：

```
Lock lock = new XxxLock();
// ...
lock.lock();
try {
    doSomething();
    doOthers();
} finally {
    lock.unlock();
}
```

ReentrantLock 与 synchronized

ReentrantLock 与 synchronized 关键字都是用来实现同步的，那么它们之间有什么区别呢？我们来看看它们的对比：

- **ReentrantLock 是一个类，而 synchronized 是 Java 中的关键字**；
- **ReentrantLock 可以实现多路选择通知（可以绑定多个 Condition（后面会细讲，戳链接直达）），而 synchronized 只能通过 wait 和 notify/notifyAll 方法唤醒一个线程或者唤醒全部线程（单路通知）**；
- ReentrantLock 必须手动释放锁。通常需要在 finally 块中调用 unlock 方法以确保锁被正确释放。synchronized 会自动释放锁，当同步块执行完毕时，由 JVM 自动释放，不需要手动操作。
- ReentrantLock: 通常提供更好的性能，特别是在高竞争环境下。synchronized: 在某些情况下，性能可能稍差一些，但随着 JDK 版本的升级，性能差距已经不大了。

以下是一个简单的性能比较demo：

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

public class PerformanceTest {
    private static final int NUM_THREADS = 10;
    private static final int NUM_INCREMENTS = 1_000_000;

    private int count1 = 0;
    private int count2 = 0;

    private final ReentrantLock lock = new ReentrantLock();
    private final Object syncLock = new Object();

    public void increment1() {
        lock.lock();
        try {
            count1++;
        } finally {
            lock.unlock();
        }
    }

    public void increment2() {
        synchronized (syncLock) {
            count2++;
```

```
        }
    }

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

        // Test ReentrantLock
        long startTime = System.nanoTime();
        Thread[] threads = new Thread[NUM_THREADS];
        for (int i = 0; i < NUM_THREADS; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < NUM_INCREMENTS; j++) {
                    test.increment1();
                }
            });
            threads[i].start();
        }
        for (Thread thread : threads) {
            thread.join();
        }
        long endTime = System.nanoTime();
        System.out.println("ReentrantLock time: " + (endTime - startTime) + " ns");

        // Test synchronized
        startTime = System.nanoTime();
        for (int i = 0; i < NUM_THREADS; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < NUM_INCREMENTS; j++) {
                    test.increment2();
                }
            });
            threads[i].start();
        }
        for (Thread thread : threads) {
            thread.join();
        }
        endTime = System.nanoTime();
        System.out.println("synchronized time: " + (endTime - startTime) + " ns");
    }
}
```

来看输出结果：

```
ReentrantLock time: 269913857 ns
synchronized  time: 350595013 ns
```

这个测试在两种锁机制下尝试执行多次增量操作，然后测量所需的时间。

小结

本篇主要介绍了 ReentrantLock 的实现原理，以及与 synchronized 关键字的比较。

> 编辑：沉默王二，编辑前的内容主要来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency

---

第十六节：读写锁 ReentrantReadWriteLock

ReentrantReadWriteLock 是 Java 的一种读写锁，它允许多个读线程同时访问，但只允许一个写线程访问（会阻塞所有的读写线程）。这种锁的设计可以提高性能，特别是在读操作的数量远远超过写操作的情况下。

在并发场景中，为了解决线程安全问题，我们通常会使用关键字 synchronized 或者 JUC 包中实现了 Lock 接口的 ReentrantLock。但它们都是独占式获取锁，也就是在同一时刻只有一个线程能够获取锁。

而在一些业务场景中，大部分只是读数据，写数据很少，如果仅仅是读数据的话并不会影响数据正确性，而如果在这种业务场景下，依然使用独占锁的话，很显然会出现性能瓶颈。针对这种读多写少的情况，Java 提供了另外一个实现 Lock 接口的 ReentrantReadWriteLock——读写锁。

我们在前面讲 Lock 接口的时候，提到过读写锁，不知道大家是否还有印象。

读写锁允许同一时刻被多个读线程访问，但是在写线程访问时，所有的读线程和其他的写线程都会被阻塞。

在分析 WirteLock 和 ReadLock 的互斥性时，我们可以按照 WriteLock 与 WriteLock，WriteLock 与 ReadLock 以及 ReadLock 与 ReadLock 进行对比分析。

这里总结一下读写锁的特性：

1）**公平性选择**：支持非公平性（默认）和公平的锁获取方式，非公平的吞吐量优于公平；

在计算机科学和性能评估中，吞吐量（Throughput）是一个衡量系统处理能力的指标。它描述了单位时间内系统能够处理的事务或操作数量。吞吐量可以用来评估系统的效率和性能，例如，每秒钟完成多少次请求或操作。

非公平锁不保证等待获取锁的线程的顺序。当锁被释放时，哪个线程能够获取该锁并不遵循任何特定的顺序。这种方式通常效率较高，因为线程不需要按照队列顺序等待，从而可以减少上下文切换和调度开销，提高吞吐量。

公平锁则确保等待获取锁的线程将按照它们请求锁的顺序来获取锁。第一个请求锁的线程将是第一个获得锁的线程，以此类推。虽然公平锁的行为更容易预测，但由于需要维护一个明确的队列顺序，可能会增加额外的开销，从而降低吞吐量。

我们在讲重入锁ReentrantLock提到过这一点。

2）**重入性**：支持重入，读锁获取后能再次获取，写锁获取之后能够再次获取写锁，同时也能够获取读锁；

我们前面在讲 Lock 的时候也细致地讲过这一点。

3）**锁降级**：写锁降级是一种允许写锁转换为读锁的过程。通常的顺序是：

- 获取写锁：线程首先获取写锁，确保在修改数据时排它访问。
- 获取读锁：在写锁保持的同时，线程可以再次获取读锁。
- 释放写锁：线程保持读锁的同时释放写锁。
- 释放读锁：最后线程释放读锁。

这样，写锁就降级为读锁，允许其他线程进行并发读取，但仍然排除其他线程的写操作。下面的代码展示了如何使用 ReentrantReadWriteLock 来降级写锁：

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
ReentrantReadWriteLock.WriteLock writeLock = lock.writeLock();
ReentrantReadWriteLock.ReadLock readLock = lock.readLock();

writeLock.lock(); // 获取写锁
try {
    // 执行写操作
    readLock.lock(); // 获取读锁
} finally {
    writeLock.unlock(); // 释放写锁
}

try {
    // 执行读操作
} finally {
    readLock.unlock(); // 释放读锁
}
```

写锁降级为读锁的过程有助于保持数据的一致性，而不影响并发读取的性能。通过这种方式，线程可以继续保持对数据的独占访问权限，直到它准备允许其他线程共享读取访问。这样可以确保在写操作和随后的读操作之间的数据一致性，并且允许其他读取线程并发访问。

要想彻底理解读写锁必须能够理解这几个问题：

- 1. 读写锁是怎样实现分别记录读写状态的?
- 2. 写锁是怎样获取和释放的?
- 3. 读锁是怎样获取和释放的?

我们带着这样的三个问题，再去了解下读写锁。

写锁详解

写锁的获取

同一时刻，ReentrantReadWriteLock 的写锁是不能被多个线程获取的，很显然 ReentrantReadWriteLock 的写锁是独占式锁，而实现写锁的同步语义是通过重写 AQS 中的 tryAcquire 方法实现的。源码为:

```
protected final boolean tryAcquire(int acquires) {
    /*
     * Walkthrough:
     * 1. If read count nonzero or write count nonzero
     *    and owner is a different thread, fail.
     * 2. If count would saturate, fail. (This can only
     *    happen if count is already nonzero.)
     * 3. Otherwise, this thread is eligible for lock if
     *    it is either a reentrant acquire or
     *    queue policy allows it. If so, update state
     *    and set owner.
     */
    Thread current = Thread.currentThread();
  // 1. 获取写锁当前的同步状态
    int c = getState();
  // 2. 获取写锁获取的次数
    int w = exclusiveCount(c);
    if (c != 0) {
        // (Note: if c != 0 and w == 0 then shared count != 0)
    // 3.1 当读锁已被读线程获取或者当前线程不是已经获取写锁的线程的话
    // 当前线程获取写锁失败
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // Reentrant acquire
    // 3.2 当前线程获取写锁，支持可重复加锁
        setState(c + acquires);
        return true;
    }
  // 3.3 写锁未被任何线程获取，当前线程可获取写锁
```

```
    if (writerShouldBlock() ||
        !compareAndSetState(c, c + acquires))
        return false;
    setExclusiveOwnerThread(current);
    return true;
}
```

这段代码的逻辑请看注释，这里有一个地方需要重点关注，`exclusiveCount(c)`方法，该方法源码为：

```
static int exclusiveCount(int c)    {
    return c & EXCLUSIVE_MASK;
}
```

其中**EXCLUSIVE_MASK**为:

```
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
```

EXCLUSIVE_MASK 为 1 左移 16 位然后减 1，即为 0x0000FFFF。而 exclusiveCount 方法是将同步状态（state 为 int 类型）与 0x0000FFFF 相与，即取同步状态的低 16 位。

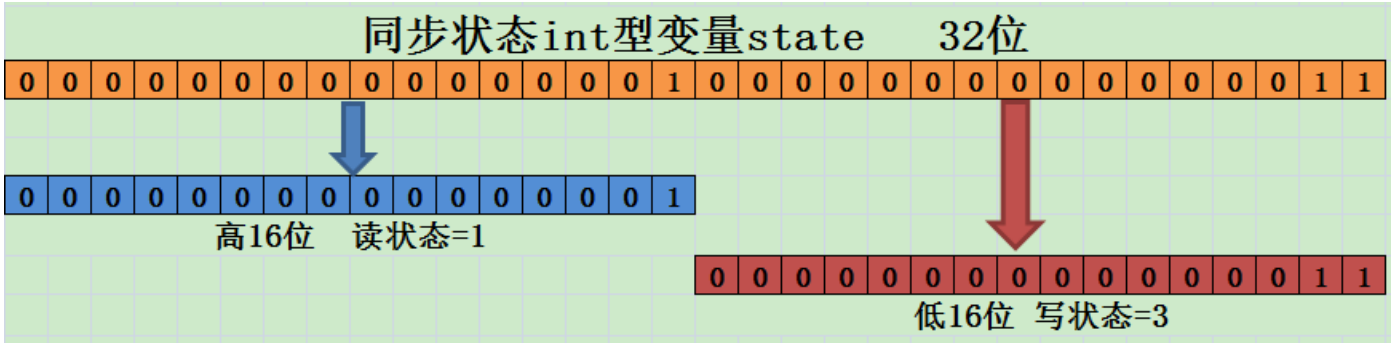
那么低 16 位代表什么呢？根据 exclusiveCount 方法的注释为独占式获取的次数即写锁被获取的次数，现在就可以得出来一个结论**同步状态的低 16 位用来表示写锁的获取次数**。

同时还有一个方法值得我们注意：

```
static int sharedCount(int c)    {
    return c >>> SHARED_SHIFT;
}
```

该方法是获取读锁被获取的次数，是将同步状态（int c）右移 16 次，即取同步状态的高 16 位，现在我们可以得出另外一个结论**同步状态的高 16 位用来表示读锁被获取的次数**。

还记得这个问题“读写锁是怎样实现分别记录读写状态的”吗？其示意图如下图所示：



好，现在我们回过头来看写锁获取方法 tryAcquire，其主要逻辑为：**当读锁已经被读线程获取或者写锁已经被其他写线程获取，则写锁获取失败；否则，获取成功并支持重入，增加写状态。**

写锁的释放

写锁释放通过重写 AQS 的 tryRelease 方法，源码为：

```
protected final boolean tryRelease(int releases) {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
  //1. 同步状态减去写状态
    int nextc = getState() - releases;
  //2. 当前写状态是否为0，为0则释放写锁
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
  //3. 不为0则更新同步状态
    setState(nextc);
    return free;
}
```

源码的实现逻辑请看注释，不难理解，与 ReentrantLock 基本一致，这里需要注意的是，减少写状态 `int nextc = getState() - releases;` 只需要用**当前同步状态直接减去写状态，原因正是我们刚才所说的写状态是由同步状态的低 16 位表示的。**

读锁详解

读锁的获取

看完了写锁，再来看看读锁，读锁不是独占式锁，即同一时刻该锁可以被多个读线程获取，也就是一种共享式锁。按照之前对 AQS 的介绍，实现共享式同步组件的同步语义需要通过重写 AQS 的 tryAcquireShared 方法和 tryReleaseShared 方法。读锁的获取实现方法为：

```
protected final int tryAcquireShared(int unused) {
    /*
     * Walkthrough:
     * 1. If write lock held by another thread, fail.
     * 2. Otherwise, this thread is eligible for
     *    lock wrt state, so ask if it should block
     *    because of queue policy. If not, try
     *    to grant by CASing state and updating count.
     *    Note that step does not check for reentrant
     *    acquires, which is postponed to full version
     *    to avoid having to check hold count in
     *    the more typical non-reentrant case.
     * 3. If step 2 fails either because thread
     *    apparently not eligible or CAS fails or count
     *    saturated, chain to version with full retry loop.
     */
    Thread current = Thread.currentThread();
    int c = getState();
```

```
  //1. 如果写锁已经被获取并且获取写锁的线程不是当前线程的话，当前
  // 线程获取读锁失败返回-1
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    int r = sharedCount(c);
    if (!readerShouldBlock() &&
        r < MAX_COUNT &&
    //2. 当前线程获取读锁
        compareAndSetState(c, c + SHARED_UNIT)) {
    //3. 下面的代码主要是新增的一些功能，比如getReadHoldCount()方法
    //返回当前获取读锁的次数
        if (r == 0) {
            firstReader = current;
            firstReaderHoldCount = 1;
        } else if (firstReader == current) {
            firstReaderHoldCount++;
        } else {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != getThreadId(current))
                cachedHoldCounter = rh = readHolds.get();
            else if (rh.count == 0)
                readHolds.set(rh);
            rh.count++;
        }
        return 1;
    }
  //4. 处理在第二步中CAS操作失败的自旋已经实现重入性
    return fullTryAcquireShared(current);
}
```

代码的逻辑请看注释，需要注意的是 **当写锁被其他线程获取后，读锁获取失败**，否则获取成功，会利用 CAS 更新同步状态。

另外，当前同步状态需要加上 SHARED_UNIT（`(1 << SHARED_SHIFT)`，即 0x00010000）的原因，我们在上面也说过了，同步状态的高 16 位用来表示读锁被获取的次数。

如果 CAS 失败或者已经获取读锁的线程再次获取读锁时，是靠 fullTryAcquireShared 方法实现的，这段代码就不展开说了，有兴趣可以看看。

读锁的释放

读锁释放的实现主要通过方法 tryReleaseShared，源码如下，主要逻辑请看注释：

```
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
  // 前面还是为了实现getReadHoldCount等新功能
    if (firstReader == current) {
        // assert firstReaderHoldCount > 0;
        if (firstReaderHoldCount == 1)
```

```
                firstReader = null;
            else
                firstReaderHoldCount--;
        } else {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != getThreadId(current))
                rh = readHolds.get();
            int count = rh.count;
            if (count <= 1) {
                readHolds.remove();
                if (count <= 0)
                    throw unmatchedUnlockException();
            }
            --rh.count;
        }
        for (;;) {
            int c = getState();
        // 读锁释放 将同步状态减去读状态即可
            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;
        }
    }
```

锁降级

读写锁支持锁降级，**遵循按照获取写锁，获取读锁再释放写锁的次序，写锁能够降级成为读锁**，不支持锁升级，关于锁降级，下面的示例代码摘自 ReentrantWriteReadLock 源码：

```
void processCachedData() {
    rwl.readLock().lock();
    if (!cacheValid) {
        // Must release read lock before acquiring write lock
        rwl.readLock().unlock();
        rwl.writeLock().lock();
        try {
            // Recheck state because another thread might have
            // acquired write lock and changed state before we did.
            if (!cacheValid) {
                data = ...
        cacheValid = true;
      }
      // Downgrade by acquiring read lock before releasing write lock
      rwl.readLock().lock();
    } finally {
      rwl.writeLock().unlock(); // Unlock write, still hold read
```

```
    }
  }

  try {
    use(data);
  } finally {
    rwl.readLock().unlock();
  }
}
```

这里的流程可以解释如下：

- 获取读锁：首先尝试获取读锁来检查某个缓存是否有效。
- 检查缓存：如果缓存无效，则需要释放读锁，因为在获取写锁之前必须释放读锁。
- 获取写锁：获取写锁以便更新缓存。此时，可能还需要重新检查缓存状态，因为在释放读锁和获取写锁之间可能有其他线程修改了状态。
- 更新缓存：如果确认缓存无效，更新缓存并将其标记为有效。
- 写锁降级为读锁：在释放写锁之前，获取读锁，从而实现写锁到读锁的降级。这样，在释放写锁后，其他线程可以并发读取，但不能写入。
- 使用数据：现在可以安全地使用缓存数据了。
- 释放读锁：完成操作后释放读锁。

这个流程结合了读锁和写锁的优点，确保了数据的一致性和可用性，同时允许在可能的情况下进行并发读取。使用读写锁的代码可能看起来比使用简单的互斥锁更复杂，但它提供了更精细的并发控制，可能会提高多线程应用程序的性能。

使用读写锁

ReentrantReadWriteLock 的使用非常简单，下面的代码展示了如何使用 ReentrantReadWriteLock 来实现一个线程安全的计数器：

```
public class Counter {
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();
    private int count = 0;

    public int getCount() {
        r.lock();
        try {
            return count;
        } finally {
            r.unlock();
        }
    }

    public void inc() {
        w.lock();
        try {
```

```
            count++;
        } finally {
            w.unlock();
        }
    }
}
```

我们再来模拟一个稍微复杂一点的例子，如何使用读写锁来实现安全地读取和更新共享数据。

```
public class CachedData {
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private Object data;
    private boolean cacheValid;

    public void processCachedData() {
        // Acquire read lock
        rwl.readLock().lock();
        if (!cacheValid) {
            // Must release read lock before acquiring write lock
            rwl.readLock().unlock();
            rwl.writeLock().lock();
            try {
                // Recheck state because another thread might have
                // acquired write lock and changed state before we did
                if (!cacheValid) {
                    data = fetchDataFromDatabase();
                    cacheValid = true;
                }
                // Downgrade by acquiring read lock before releasing write lock
                rwl.readLock().lock();
            } finally {
                rwl.writeLock().unlock(); // Unlock write, still hold read
            }
        }

        try {
            use(data);
        } finally {
            rwl.readLock().unlock();
        }
    }

    private Object fetchDataFromDatabase() {
        // Simulate fetching data from a database
        return new Object();
    }

    private void use(Object data) {
        // Simulate using the data
```

```
        System.out.println("使用数据： " + data);
    }

    public static void main(String[] args) {
        CachedData cachedData = new CachedData();
        cachedData.processCachedData();
    }
}
```

当缓存无效时，会先释放读锁，然后获取写锁来更新缓存。一旦缓存被更新，就会进行写锁到读锁的降级，允许其他线程并发读取，但仍然排除写入。

这样的结构允许在确保数据一致性的同时，实现并发读取的优势，从而提高多线程环境下的性能。

小结

ReentrantReadWriteLock 是 Java 的一种读写锁，它允许多个读线程同时访问，但只允许一个写线程访问，或者阻塞所有的读写线程。这种锁的设计可以提高性能，特别是在数据结构中，读操作的数量远远超过写操作的情况下。

读写锁的实现主要是通过重写 AQS 的 tryAcquire 方法和 tryRelease 方法实现的，读锁和写锁的获取和释放都是通过这两个方法实现的。

读写锁支持锁降级，遵循按照获取写锁，获取读锁再释放写锁的次序，写锁能够降级成为读锁，不支持锁升级。

> 编辑：沉默王二，编辑前的内容主要来自于 CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency

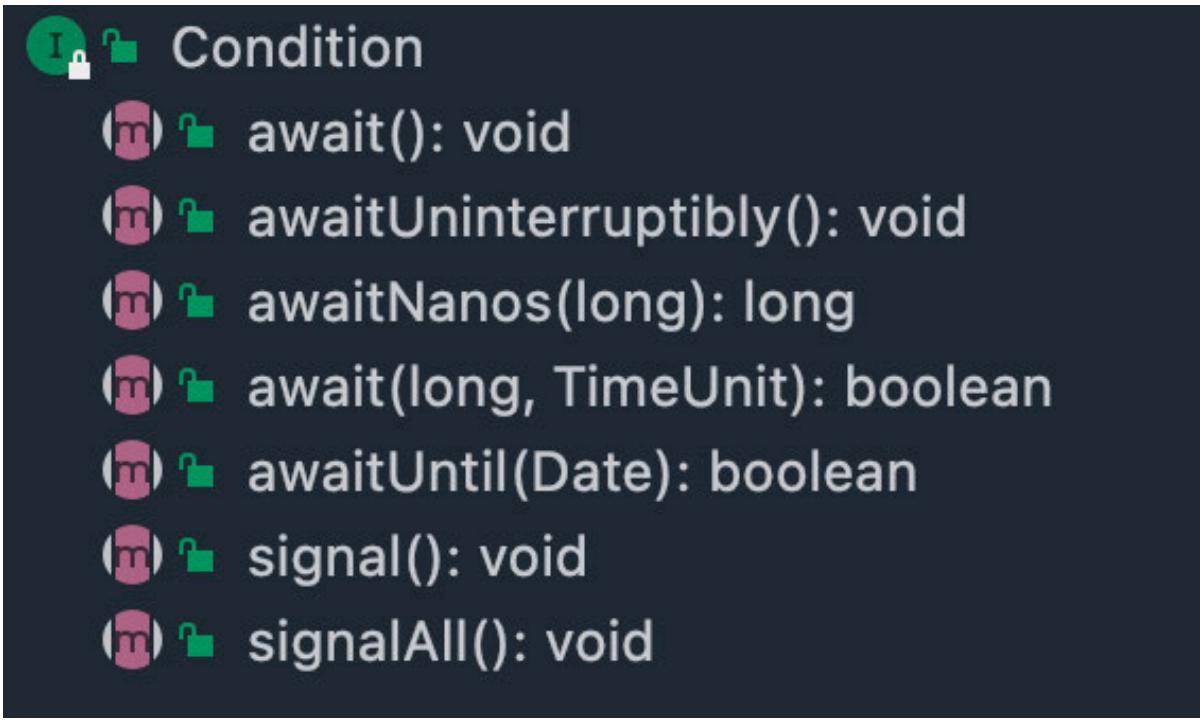
---

第十七节：等待通知条件 Condition

关于等待通知条件 Condition 我们在前面讲 ReentrantLock 的时候提到过，不知道大家还记得不？

每个对象都可以调用 Object 的 wait/notify 方法来实现等待/通知机制。而 Condition 接口也提供了类似的方法。

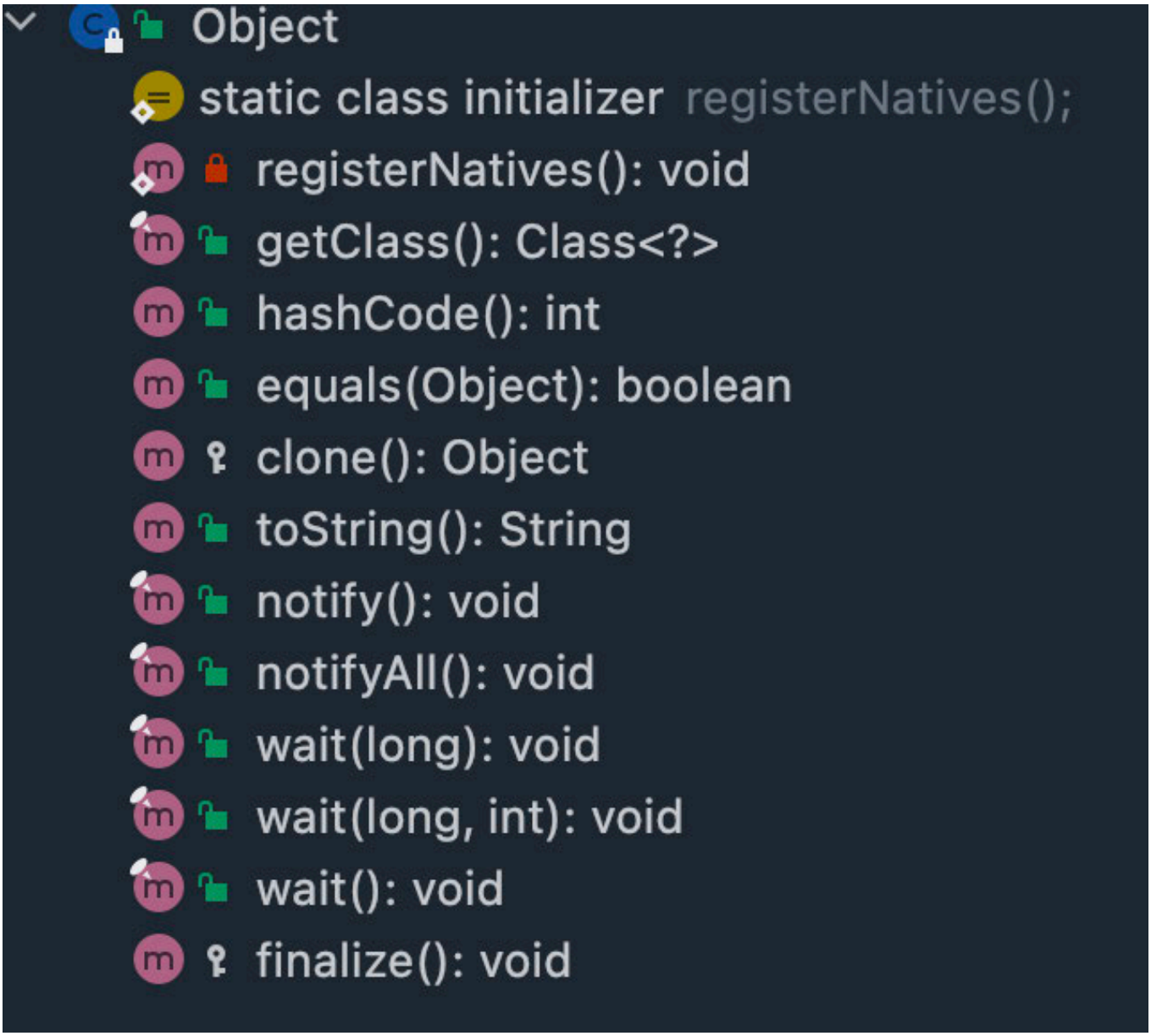
Condition 接口一共提供了以下 7 个方法：



- `await()`：线程等待直到被通知或者中断。类似于 `Object.wait()`。

- `awaitUninterruptibly()`：线程等待直到被通知，即使在等待时被中断也不会返回。没有与之对应的 Object 方法。
- `await(long time, TimeUnit unit)`：线程等待指定的时间，或被通知，或被中断。类似于 `Object.wait(long timeout)`，但提供了更灵活的时间单位。
- `awaitNanos(long nanosTimeout)`：线程等待指定的纳秒时间，或被通知，或被中断。没有与之对应的 Object 方法。
- `awaitUntil(Date deadline)`：线程等待直到指定的截止日期，或被通知，或被中断。没有与之对应的 Object 方法。
- `signal()`：唤醒一个等待的线程。类似于 `Object.notify()`。
- `signalAll()`：唤醒所有等待的线程。类似于 `Object.notifyAll()`。

我们再来回顾一下 Object 类的主要方法：



- `wait()`：线程等待直到被通知或者中断。
- `wait(long timeout)`：线程等待指定的时间，或被通知，或被中断。
- `wait(long timeout, int nanos)`：线程等待指定的时间，或被通知，或被中断。
- `notify()`：唤醒一个等待的线程。
- `notifyAll()`：唤醒所有等待的线程。

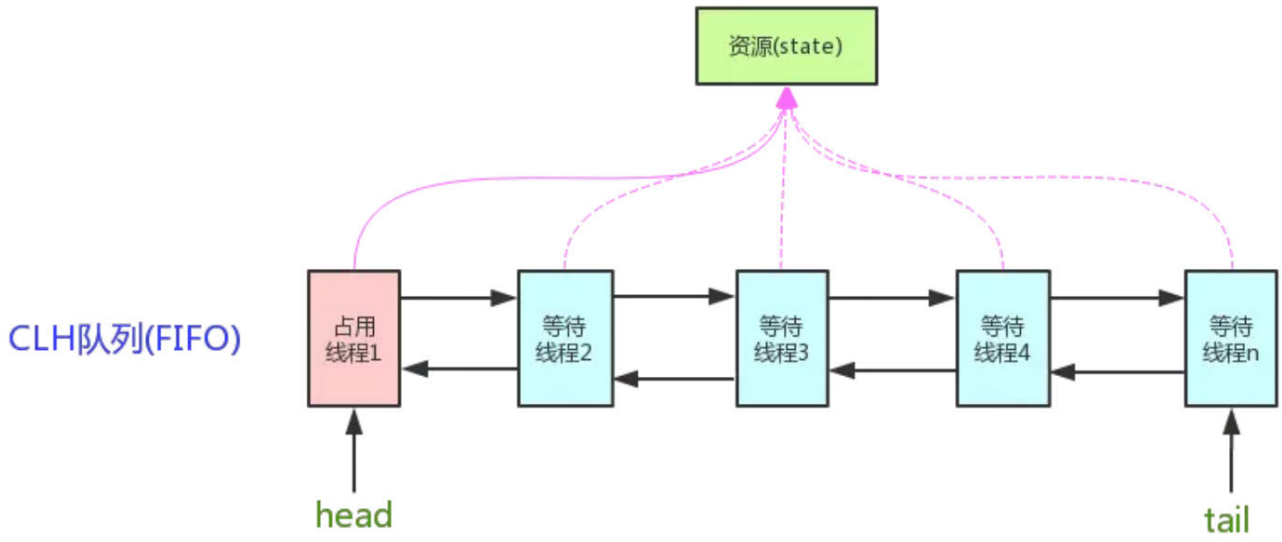
Condition 源码分析

要想深入理解 Condition 的实现原理，就需要挖掘一下 Condiiton 的源码。

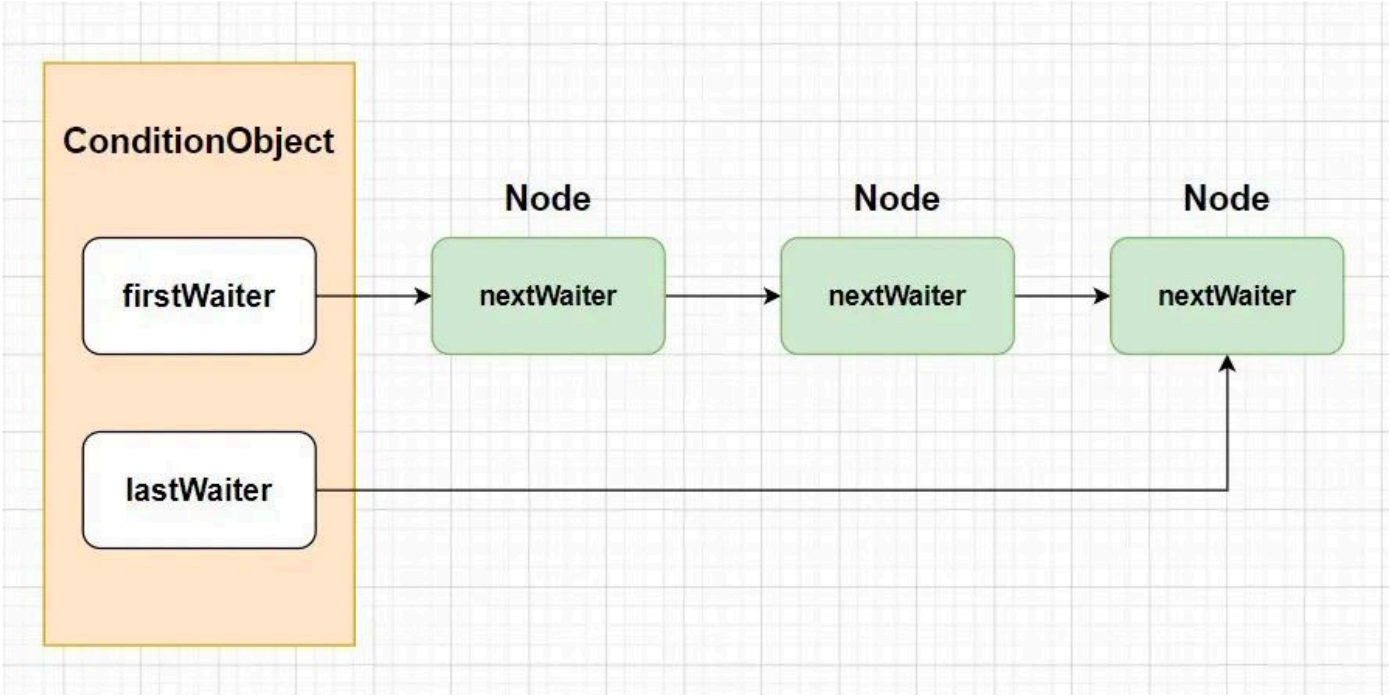
创建一个 Condition 对象可以通过`lock.newCondition()`来创建，这个方法实际上会 new 一个 **ConditionObject** 的对象，ConditionObject 是 AQS 的一个内部类，我们就拿 ReentrantLock 来举例说明吧。

```
public class ReentrantLock implements Lock, java.io.Serializable {
    abstract static class Sync extends AbstractQueuedSynchronizer {
        final ConditionObject newCondition() {
            return new ConditionObject();
        }
    }
    public Condition newCondition() {
        return sync.newCondition();
    }
}
```

前面我们学过，AQS 内部维护了一个先进先出（FIFO）的双端队列，并使用了两个引用 head 和 tail 用于标识队列的头部和尾部。



Condition 内部也使用了同样的方式，内部维护了一个先进先出（FIFO）的单向队列，我们把它称为等待队列。



所有调用 await 方法的线程都会加入到等待队列中，并且线程状态均为等待状态。firstWaiter 指向首节点，lastWaiter 指向尾节点，源码如下：

```
public class ConditionObject implements Condition, java.io.Serializable {
    private static final long serialVersionUID = 1173984872572414699L;
    /** First node of condition queue. */
    private transient Node firstWaiter;
    /** Last node of condition queue. */
    private transient Node lastWaiter;
}
```

Node 中的 nextWaiter 指向队列中的下一个节点。并且进入到等待队列的 Node 节点状态都会被设置为 CONDITION（下面的 demo 中可以看得到）。

```
static final class Node { Complexity is 6 It's time to do something...
    Marker to indicate a node is waiting in shared mode
    static final Node SHARED = new Node();
    Marker to indicate a node is waiting in exclusive mode
    static final Node EXCLUSIVE = null;

    waitStatus value to indicate thread has cancelled
    static final int CANCELLED =  1;
    waitStatus value to indicate successor's thread needs unparking
    static final int SIGNAL    = -1;
    waitStatus value to indicate thread is waiting on condition
    static final int CONDITION = -2;
    waitStatus value to indicate the next acquireShared should unconditionally propagate
```

```
static final int PROPAGATE = -3;
```

状态字段，仅取值： SIGNAL：此节点的后继节点被（或即将）阻止（通过 park），因此当前节点在释放或取消时必须取消其后继节点的停放。为了避免竞争，获取方法必须首先指示它们需要信号，然后重试原子获取，然后在失败时阻止。已取消：此节点因超时或中断而被取消。节点永远不会离开此状态。特别是，具有已取消节点的线程永远不会再阻塞。条件：此节点当前位于条件队列中。在传输之前，它不会用作同步队列节点，此时状态将设置为 0。（此处使用此值与该字段的其他用法无关，但简化了力学。传播：共享版本应传播到其他节点。这是在 doReleaseShared 中设置的（仅适用于头节点），以确保传播继续，即使此后其他操作已经介入。0：以上都不是 这些值按数字排列以简化使用。非负值意味着节点不需要发出信号。因此，大多数代码不需要检查特定值，只需检查符号即可。对于正常同步节点，该字段初始化为 0，对于条件节点，该字段初始化为 CONDITION。它使用 CAS 进行修改（或在可能的情况下，无条件易失性写入）。

```
volatile int waitStatus;
```

链接到当前节点/线程所依赖的前置节点来检查 waitStatus。在排队期间分配，仅在出列时取消（为了 GC）。此外，在取消前置节点后，我们会短路，同时找到一个未取消的节点，该节点将始终存在，因为头节点永远不会被取消：节点只有在成功获取后才会成为头节点。已取消的线程永远不会成功获取，线程只会取消自身，而不会取消任何其他节点。

```
volatile Node prev;
```

链接到当前节点/线程在发布时取消停放的后续节点。在排队期间分配，在绕过取消的前置任务时进行调整，在取消排队时取消（为了 GC）。enq 操作直到附加后才会分配前置任务的下一个字段，因此看到空的下一个字段并不一定意味着该节点位于队列末尾。但是，如果下一个字段显示为空，我们可以从尾部扫描 prev 以仔细检查。已取消节点的下一个字段设置为指向节点本身而不是 null，以使 isOnSyncQueue 的工作更轻松。

```
volatile Node next;
```

The thread that enqueued this node. Initialized on construction and nulled out after use.

```
volatile Thread thread;
```

链接到下一个节点等待条件，或特殊值 SHARED。由于条件队列仅在以独占模式保持时访问，因此我们只需要一个简单的链接队列来在节点等待条件时保留节点。然后将它们传输到队列中以重新获取。由于条件只能是独占的，因此我们使用特殊值来指示共享模式来保存字段。

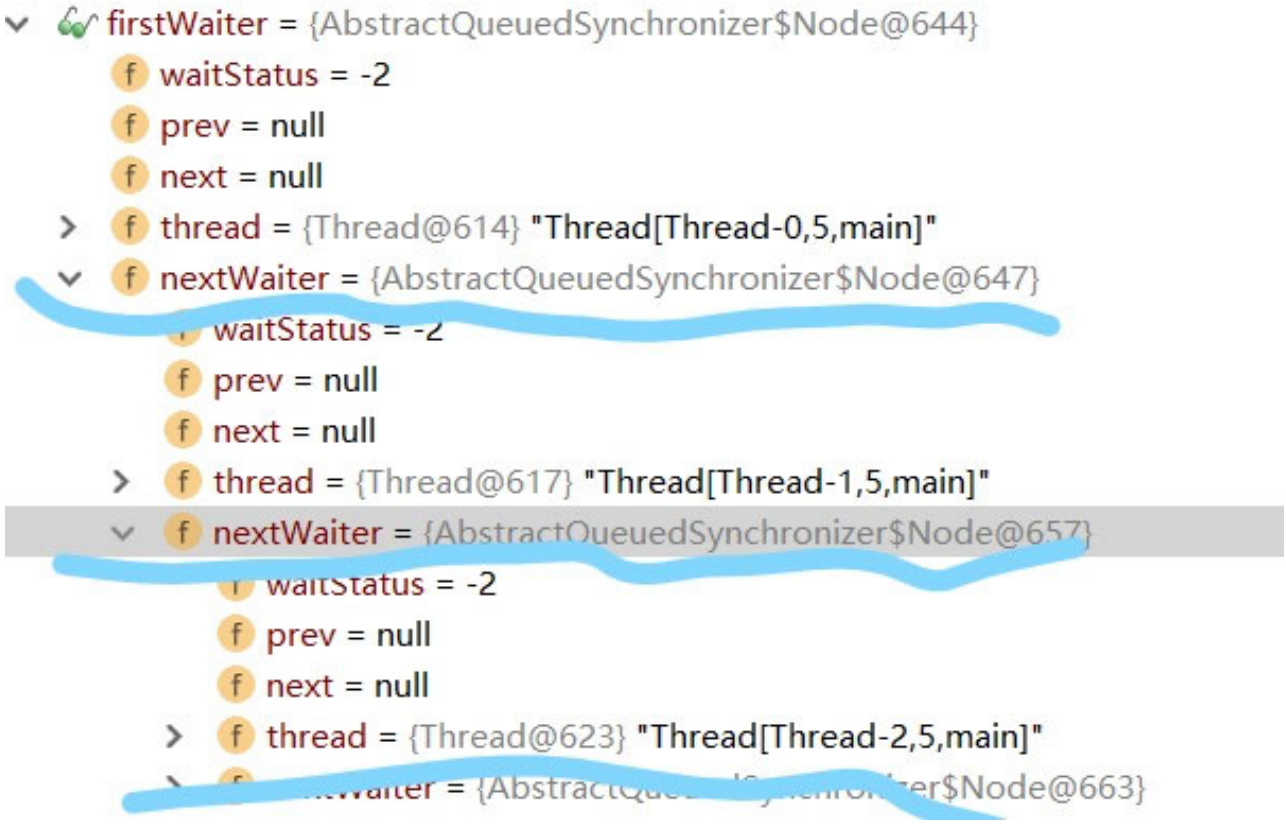
```
Node nextWaiter;
```

上面提到，Condition 的**等待队列是一个单向队列**，我们用一个 demo 通过 debug 的方式验证下。

```
public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(() -> {
            lock.lock();
            try {
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
```

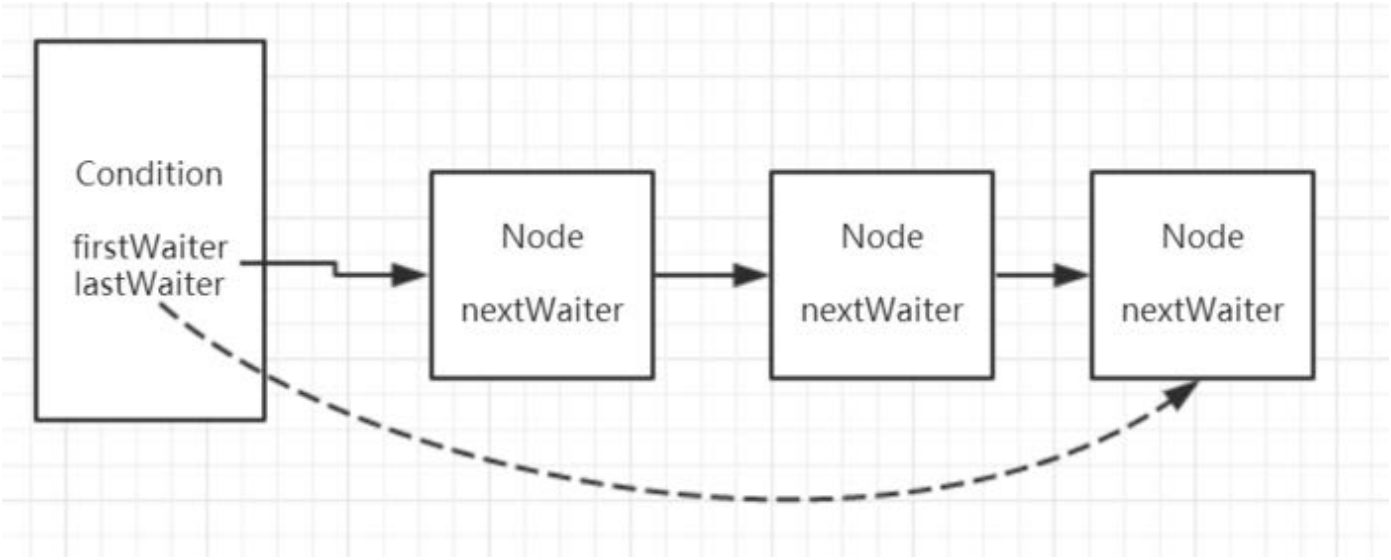
```
        });
        thread.start();
    }
}
```

这段代码没有任何实际意义，甚至很臭。新建了 10 个线程，没有线程先获取锁，然后调用 condition.await 方法释放锁将当前线程加入到等待队列中，通过 debug 走到第 10 个线程的时候，查看`firstWaiter`即等待队列中的头节点：



从这个图我们可以很清楚的看到这样 2 点：

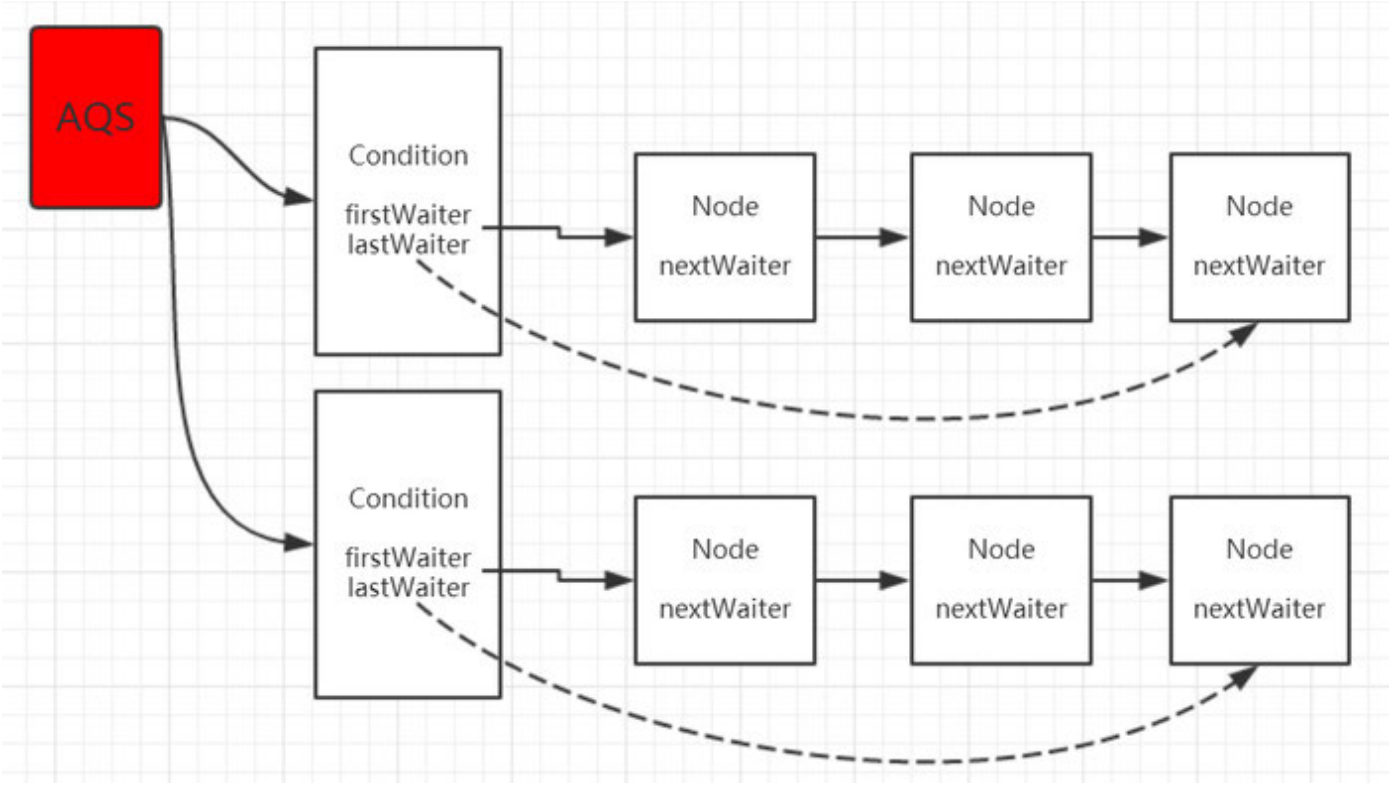
1. 调用 condition.await 方法后线程依次尾插入到了等待队列中，依次为 Thread-0,Thread-1,Thread-2....Thread-8；
2. 等待队列是一个单向队列。示意图如下：



同时还有一点需要注意：我们可以多次调用`newCondition()`方法创建多个 Condition 对象，也就是一个 lock 可以持有多个等待队列。

而如果是 Object 方式的话，就只能有一个同步队列和一个等待队列。

因此，ReentrantLock 等 AQS 是可以持有一个同步队列和多个等待队列的，new 多个 Condition 就行了。示意图如下：



持有多个等待队列的好处是什么呢？我们可以通过下面这个例子来说明：

```
public class BoundedBuffer<T> {
    private final LinkedList<T> buffer;  // 使用 LinkedList 作为缓冲区
    private final int capacity;          // 缓冲区最大容量
    private final ReentrantLock lock;    // 互斥锁
    private final Condition notEmpty;    // 缓冲区非空条件
    private final Condition notFull;     // 缓冲区非满条件

    public BoundedBuffer(int capacity) {
        this.capacity = capacity;
        this.buffer = new LinkedList<>();
        this.lock = new ReentrantLock();
        this.notEmpty = lock.newCondition();
        this.notFull = lock.newCondition();
    }

    // 放入一个元素
    public void put(T item) throws InterruptedException {
        lock.lock();
        try {
            // 如果缓冲区满，等待
```

```
            while (buffer.size() == capacity) {
                notFull.await();
            }
            buffer.add(item);
            // 通知可能正在等待的消费者
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    // 取出一个元素
    public T take() throws InterruptedException {
        lock.lock();
        try {
            // 如果缓冲区空，等待
            while (buffer.isEmpty()) {
                notEmpty.await();
            }
            T item = buffer.removeFirst();
            // 通知可能正在等待的生产者
            notFull.signal();
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

考虑这个简单的有界缓冲区 BoundedBuffer，其中生产者放入元素，消费者取出元素。我们将使用两个 Condition：一个表示缓冲区不为空（用于消费者等待），另一个表示缓冲区不满（用于生产者等待）。

生产者调用 put 方法放入元素，如果缓冲区已满，则等待 notFull 条件。消费者调用 take 方法取出元素，如果缓冲区为空，则等待 notEmpty 条件。当一个元素被放入或取出时，相应的条件会发出信号，唤醒等待的线程。

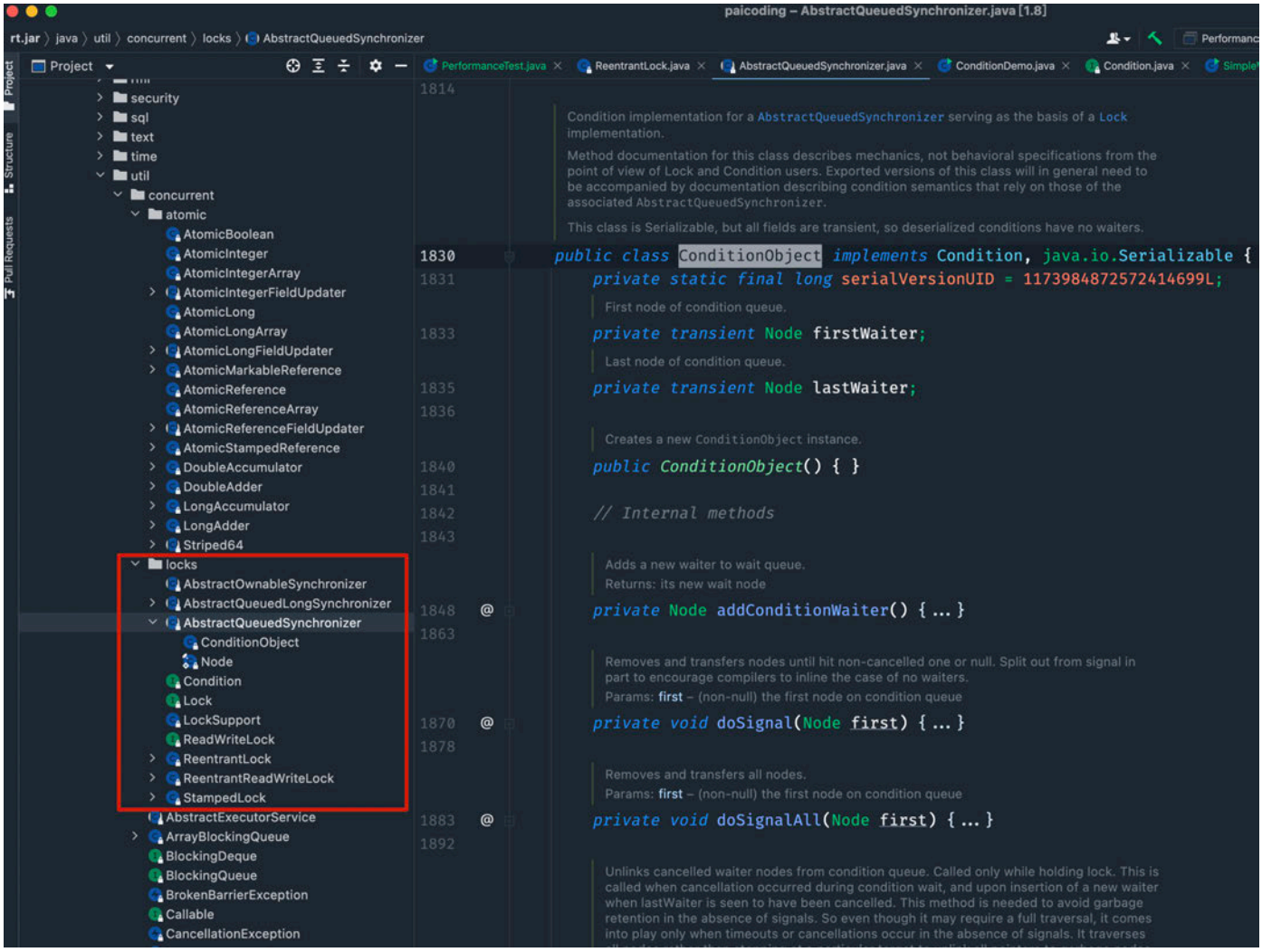
使用多个 Condition 对象的主要优点是为锁提供了更细粒度的控制，可以实现更复杂的同步场景，比如上面提到的有界缓冲区。

好，接下来，我们来继续分析 Condition 的源码。

Condition 的 await 方法

当调用`condition.await()`方法后会使当前获取锁的线程进入到等待队列，如果该线程能够从 `await()` 方法返回的话，一定是该线程获取了与 Condition 相关联的锁。

前面讲过了，Condition 只是一个接口，它的实现类为 ConditionObject，是 AQS 的子类。



ConditionObject 的 await 方法源码如下：

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
  // 1. 将当前线程包装成Node，尾插入到等待队列中
    Node node = addConditionWaiter();
  // 2. 释放当前线程所占用的lock，在释放的过程中会唤醒同步队列中的下一个节点
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
    // 3. 当前线程进入到等待状态
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
  // 4. 自旋等待获取到同步状态（即获取到lock）
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
  // 5. 处理被中断的情况
    if (interruptMode != 0)
```

```
        reportInterruptAfterWait(interruptMode);
}
```

代码的主要逻辑**请看注释**。当前线程调用`condition.await()`方法后，会释放 lock 然后加入到等待队列，直到被`signal/signalAll`方法唤醒。

大家可能会有这样几个问题：

1. 怎样将当前线程添加到等待队列中?
2. 释放锁的过程是?
3. 怎样才能从 await 方法中退出?

上面这段代码就告诉了我们这三个问题的答案。

问题 1 的答案

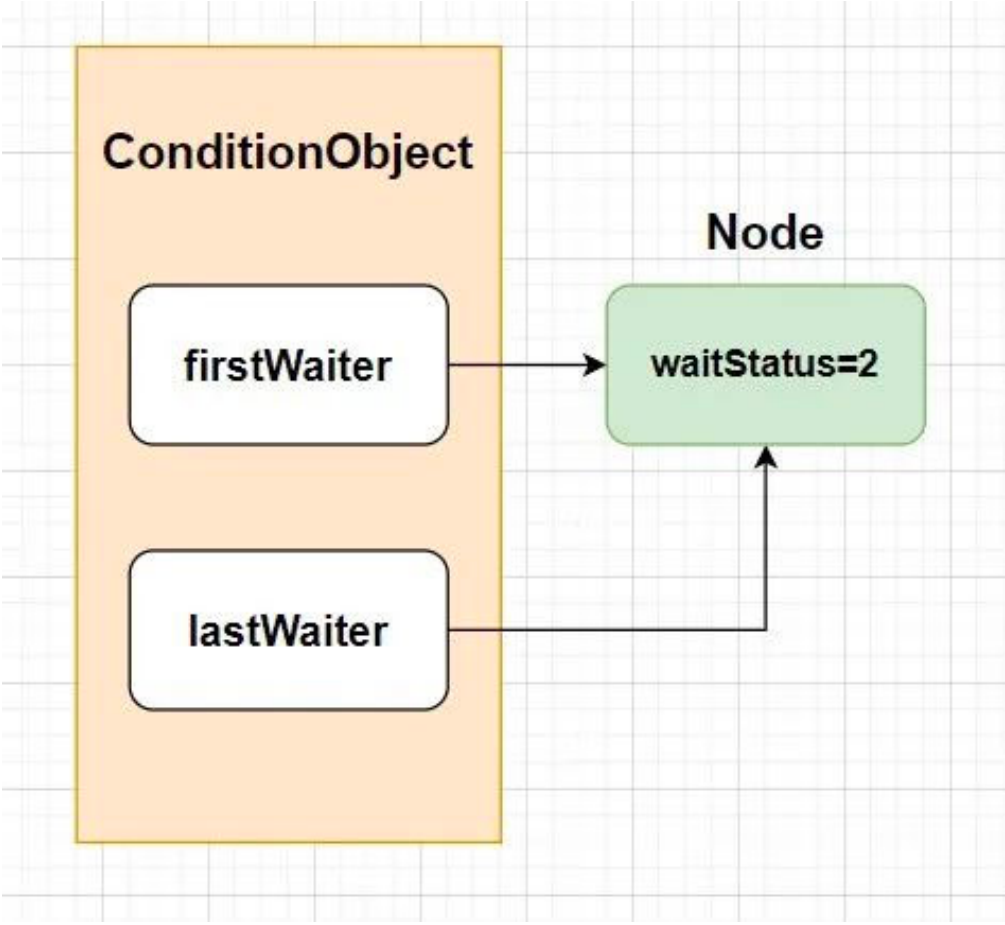
调用 addConditionWaiter 方法会将当前线程添加到等待队列中，源码如下：

```
private Node addConditionWaiter() {
 Node t = lastWaiter;
 if (t != null && t.waitStatus != Node.CONDITION) {
  //将不处于等待状态的节点从等待队列中移除
  unlinkCancelledWaiters();
  t = lastWaiter;
 }
 Node node = new Node(Thread.currentThread(), Node.CONDITION);
 //尾节点为空
 if (t == null)
        //将首节点指向node
  firstWaiter = node;
 else
  //将尾节点的nextWaiter指向node节点
  t.nextWaiter = node;
 //尾节点指向node
 lastWaiter = node;
 return node;
}
```

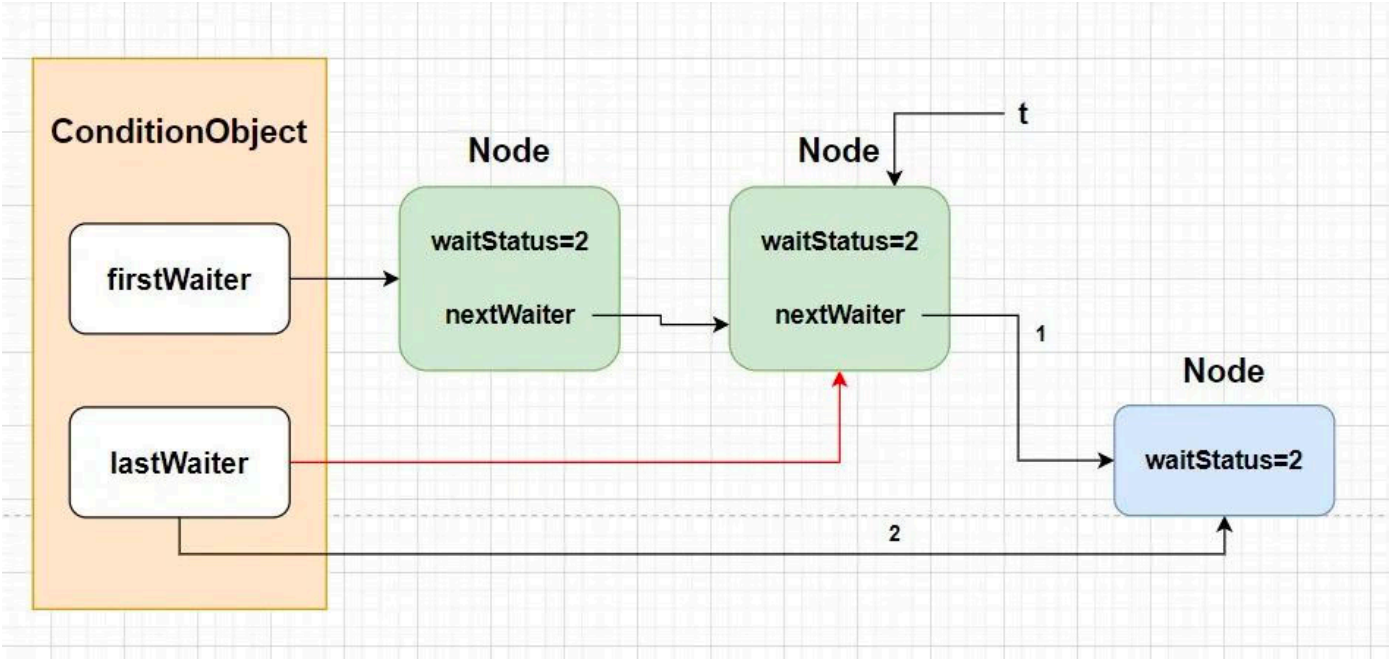
首先将 t 指向尾节点，如果尾节点不为空并且它的`waitStatus!=-2`（-2 为 CONDITION，表示正在等待 Condition 条件），则将不处于等待状态的节点从等待队列中移除，并且将 t 指向新的尾节点。

然后将当前线程封装成 waitStatus 为-2 的节点追加到等待队列尾部。

如果尾节点为空，则表明队列为空，将首尾节点都指向当前节点。



如果尾节点不为空，表明队列中有其他节点，则将当前尾节点的 nextWaiter 指向当前节点，将当前节点置为尾节点。



简单总结一下，这段代码的作用就是**通过尾插入的方式将当前线程封装的 Node 插入到等待队列中**，同时可以看出，Condtion 的等待队列是一个**不带头节点的链式队列**，之前我们学习 AQS 时知道同步队列**是一个带头节点的链式队列**，这是两者的一个区别。

关于头节点的作用，我们这里简单说明一下。

不带头节点是指在链表数据结构中，链表的第一个节点就是实际存储的第一个数据元素，而不是一个特定的"头"节点，该节点不包含实际的数据。

1）不带头节点的链表：

- 链表的第一个节点就是第一个实际的数据节点。
- 当链表为空时，头引用（通常称为 head）指向 null。

2）带头节点的链表：

- 链表有一个特殊的节点作为链表的开头，这个特殊的节点称为头节点。
- 头节点通常不存储任何实际数据，或者它的数据字段不被使用。
- 无论链表是否为空，头节点总是存在的。当链表为空时，头节点的下一个节点指向 null。
- 使用头节点可以简化某些链表操作，因为你不必特殊处理第一个元素的插入和删除。

为了更好地解释这两种链表结构，我将为每种结构提供一个简单的整数链表插入方法的示例。

1）不带头节点的链表

```
public class Node {
    public int data;
    public Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class LinkedListWithoutHead {
    public Node head;

    public void insert(int value) {
        Node newNode = new Node(value);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }
}
```

2）带头节点的链表

```
public class NodeWithHead {
    public int data;
    public NodeWithHead next;
```

```
    public NodeWithHead(int data) {
        this.data = data;
        this.next = null;
    }
}

public class LinkedListWithHead {
    private NodeWithHead head;

    public LinkedListWithHead() {
        head = new NodeWithHead(-1);  // 初始化头节点
    }

    public void insert(int value) {
        NodeWithHead newNode = new NodeWithHead(value);
        NodeWithHead temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }
}
```

这下是不是就彻底明白了？说明白了头节点，我们再回到 Condition 的 await 方法。

问题 2 的答案

将当前节点插入到等待对列之后，会使当前线程释放 lock，由 fullyRelease 方法实现，源码如下：

```
final int fullyRelease(Node node) {
 //释放锁失败为true，释放锁成功为false
 boolean failed = true;
 try {
     //获取当前锁的state
  int savedState = getState();
  //释放锁成功的话
  if (release(savedState)) {
   failed = false;
   return savedState;
  } else {
   throw new IllegalMonitorStateException();
  }
 } finally {
  if (failed)
   //释放锁失败的话将节点状态置为取消
   node.waitStatus = Node.CANCELLED;
 }
}
```

这段代码也很容易理解，**调用 AQS 的模板方法 release 释放 AQS 的同步状态并且唤醒在同步队列中头节点的后继节点引用的线程**，如果释放成功则正常返回，若失败的话就抛出异常。

问题 3 的答案

怎样从 await 方法退出呢？现在回过头再来看 await 方法，其中有这样一段逻辑：

```
while (!isOnSyncQueue(node)) {
  // 3. 当前线程进入到等待状态
    LockSupport.park(this);
    if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
        break;
}
```

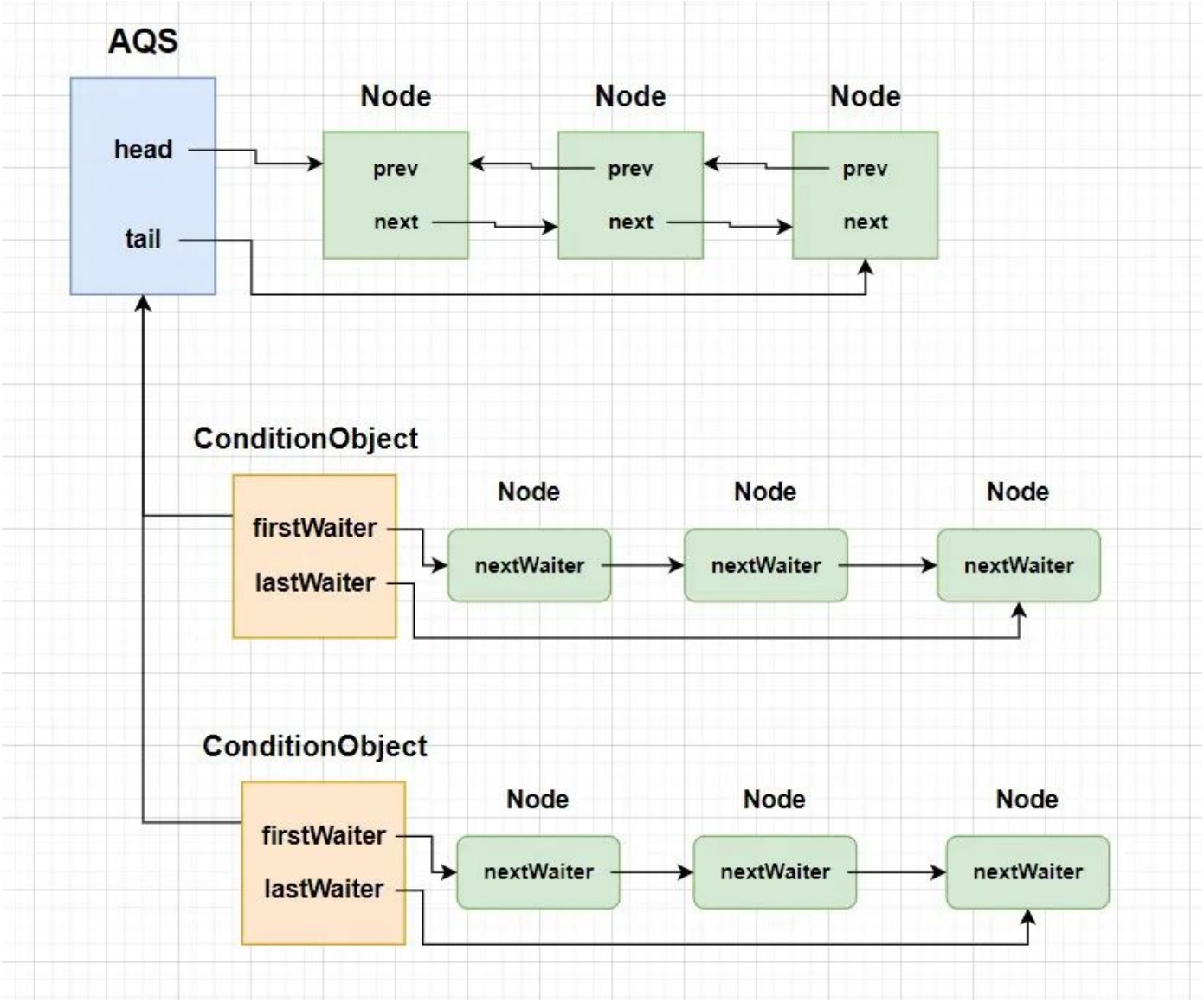
isOnSyncQueue 方法用于判断当前线程所在的 Node 是否在同步队列中。

```
如果节点（始终是最初放置在条件队列中的节点）现在正在等待在同步队列上重新获取，则返回 true。
参数： 节点 – 节点
返回： 如果正在重新获取，则为 true
final boolean isOnSyncQueue(Node node) {  Complexity is 10 It's time to do something...
    if (node.waitStatus == Node.CONDITION || node.prev == null)
        return false;
    if (node.next != null) // If has successor, it must be on queue
        return true;
    /*
     * node.prev can be non-null, but not yet on queue because
     * the CAS to place it on queue can fail. So we have to
     * traverse from tail to make sure it actually made it.  It
     * will always be near the tail in calls to this method, and
     * unless the CAS failed (which is unlikely), it will be
     * there, so we hardly ever traverse much.
     */
    return findNodeFromTail(node);
}
```

如果当前节点的 waitStatus=-2，说明它在等待队列中，返回 false；如果当前节点有前驱节点，则证明它在 AQS 队列中，但是前驱节点为空，说明它是头节点，而头节点是不参与锁竞争的，也返回 false。

如果当前节点既不在等待队列中，又不是 AQS 中的头节点且存在 next 节点，说明它存在于 AQS 中，直接返回 true。

这里有必要给大家看一下同步队列与等待队列的关系图了。



当线程第一次调用 condition.await 方法时，会进入到这个 while 循环，然后通过 `LockSupport.park(this)` 使当前线程进入等待状态，那么要想退出 await，第一个前提条件就是要先退出这个 while 循环，出口就只两个地方：

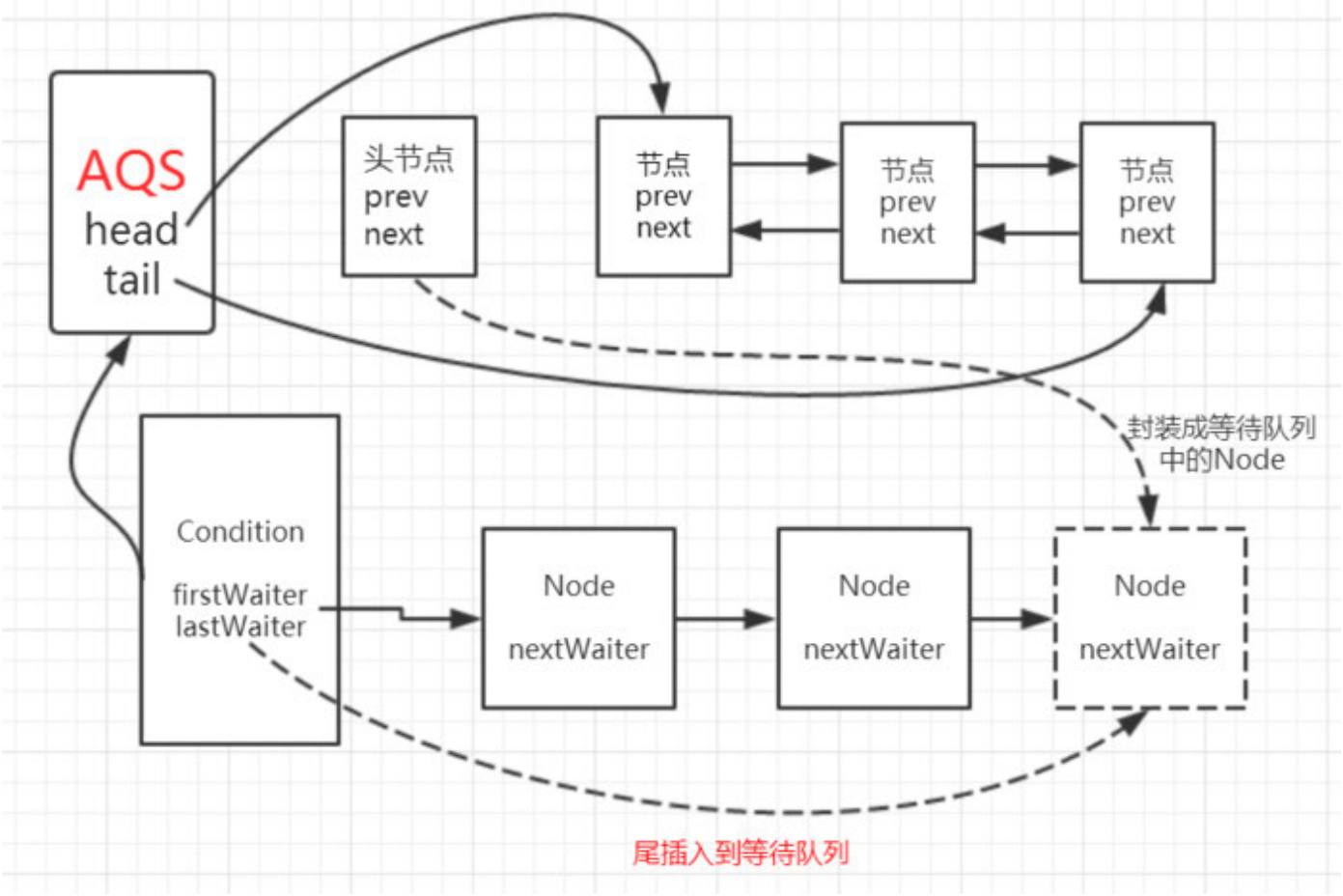
1. 走到 break 退出 while 循环；
2. while 循环中的逻辑判断为 false。

出现第 1 种情况的条件是，当前等待的线程被中断后代码会走到 break 退出，第 2 种情况是当前节点被移动到了同步队列中（即另外一个线程调用了 condition 的 signal 或者 signalAll 方法），while 中逻辑判断为 false 后结束 while 循环。

总结一下，退出 await 方法的前提条件是**当前线程被中断或者调用 condition.signal 或者 condition.signalAll 使当前节点移动到同步队列后**。

当退出 while 循环后会调用`acquireQueued(node, savedState)`，该方法的作用是在**自旋过程中线程不断尝试获取同步状态，直到成功（线程获取到 lock）**。这样也说明了**退出 await 方法必须是已经获得了 condition 引用（关联）的 lock**。

到目前为止，上文提到的三个问题，我们都通过阅读源码的方式找到了答案，也加深了对 await 方法的理解。await 方法示意图如下：



如图，调用 condition.await 方法的线程必须是已经获得了 lock 的线程，也就是当前线程是同步队列中的头节点。调用该方法后会使得当前线程所封装的 Node 尾插入到等待队列中。

> 超时机制的支持

condition 还额外支持超时机制，使用者可调用 awaitNanos、awaitUtil 这两个方法，实现原理基本上与 AQS 中的 tryAcquire 方法如出一辙。

> 不响应中断的支持

要想不响应中断可以调用 `condition.awaitUninterruptibly()` 方法，该方法的源码如下：

```
public final void awaitUninterruptibly() {
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    boolean interrupted = false;
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        if (Thread.interrupted())
            interrupted = true;
    }
    if (acquireQueued(node, savedState) || interrupted)
        selfInterrupt();
}
```

这段方法与上面的 await 方法基本一致，只不过减少了对中断的处理。

signal/signalAll 实现原理

调用 condition 的 signal 或者 signalAll 方法可以将等待队列中等待时间最长的节点移动到同步队列中，使得该节点能够有机会获得 lock。等待队列是先进先出（FIFO）的，所以等待队列的头节点必然会是等待时间最长的节点，也就是每次调用 condition 的 signal 方法都会将头节点移动到同步队列中。

我们来通过看源码的方式验证这个猜想是不是正确的，signal 方法源码如下：

```
public final void signal() {
    //1. 先检测当前线程是否已经获取lock
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    //2. 获取等待队列中第一个节点，之后的操作都是针对这个节点
  Node first = firstWaiter;
    if (first != null)
        doSignal(first);
}
```

signal 方法首先会检测当前线程是否已经获取了 lock，如果没有获取 lock 会直接抛出异常，如果获取的话，再得到等待队列的头节点，之后的 doSignal 方法也是基于该节点。下面我们来看看 doSignal 方法做了些什么事情，doSignal 方法源码如下：

```
private void doSignal(Node first) {
    do {
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        //1. 将头节点从等待队列中移除
        first.nextWaiter = null;
        //2. while中transferForSignal方法对头节点做真正的处理
    } while (!transferForSignal(first) &&
                (first = firstWaiter) != null);
}
```

具体逻辑请看注释，真正对头节点做处理的逻辑在**transferForSignal**方法中，该方法源码为：

```
final boolean transferForSignal(Node node) {
    /*
     * If cannot change waitStatus, the node has been cancelled.
     */
  //1. 更新状态为0
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    /*
     * Splice onto queue and try to set waitStatus of predecessor to
     * indicate that thread is (probably) waiting. If cancelled or
     * attempt to set waitStatus fails, wake up to resync (in which
     * case the waitStatus can be transiently and harmlessly wrong).
```

```
        */
    //2.将该节点移入到同步队列中去
        Node p = enq(node);
        int ws = p.waitStatus;
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
            LockSupport.unpark(node.thread);
        return true;
}
```

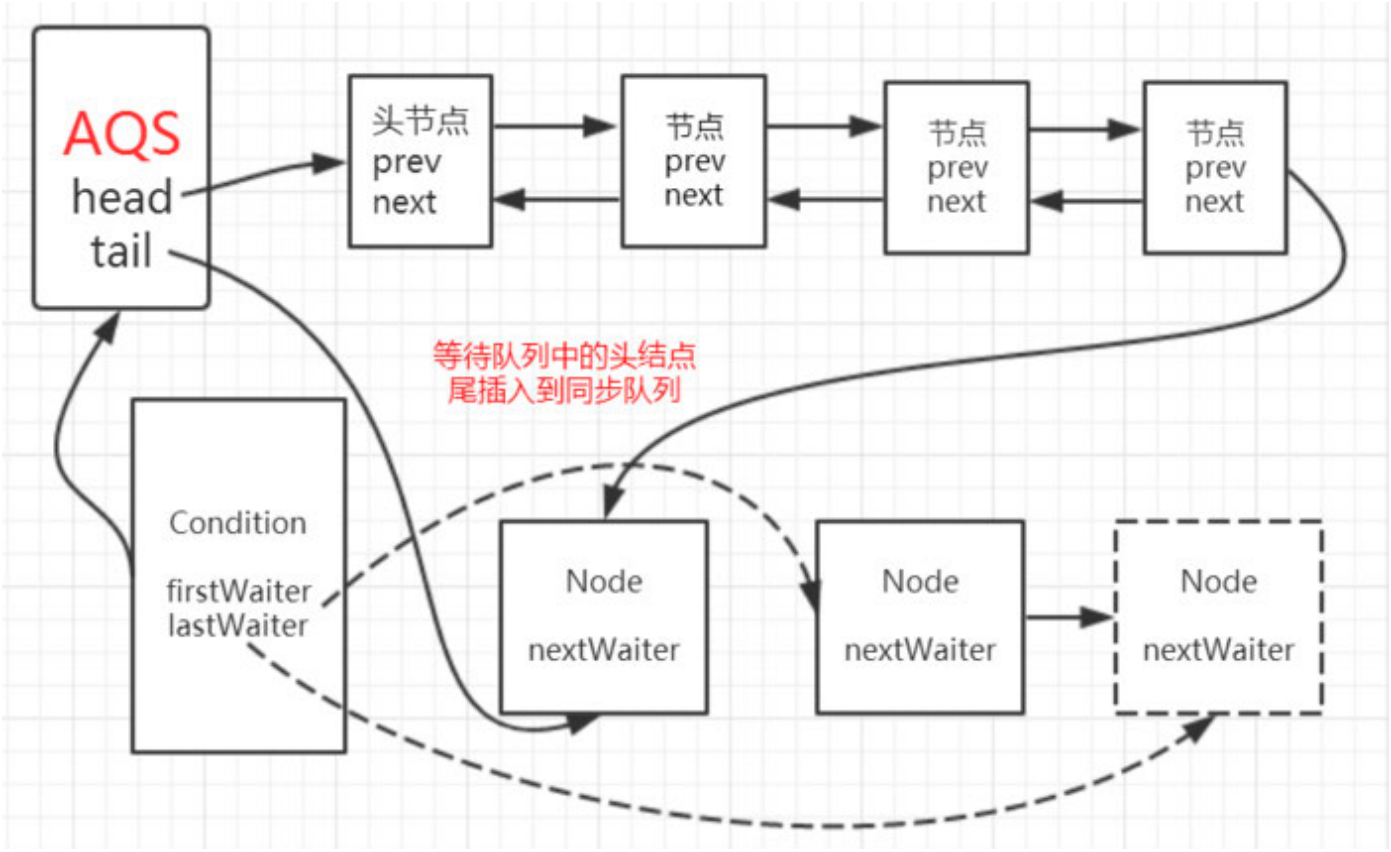
关键逻辑请看注释，这段代码主要做了两件事情：

- 1.将头节点的状态更改为 CONDITION；
- 2.调用 enq 方法，将该节点尾插入到同步队列中，关于 enq 方法请看 AQS 的底层实现这篇文章。

现在我们可以得出如下结论：

调用 condition.signal 方法的前提条件是当前线程已经获取了 lock，该方法会使等待队列中的头节点即等待时间最长的那个节点移入到同步队列，而移入到同步队列后才有机会被唤醒，即从 await 方法中的 `LockSupport.park(this)` 方法中返回，才有机会让调用 await 方法的线程成功退出。

signal 执行示意图如下图：



> signalAll

sigllAll 与 sigal 方法的区别体现在 doSignalAll 方法上，前面我们已经知道 **doSignal 方法只会对等待队列的头节点进行操作**，doSignalAll 的源码如下：

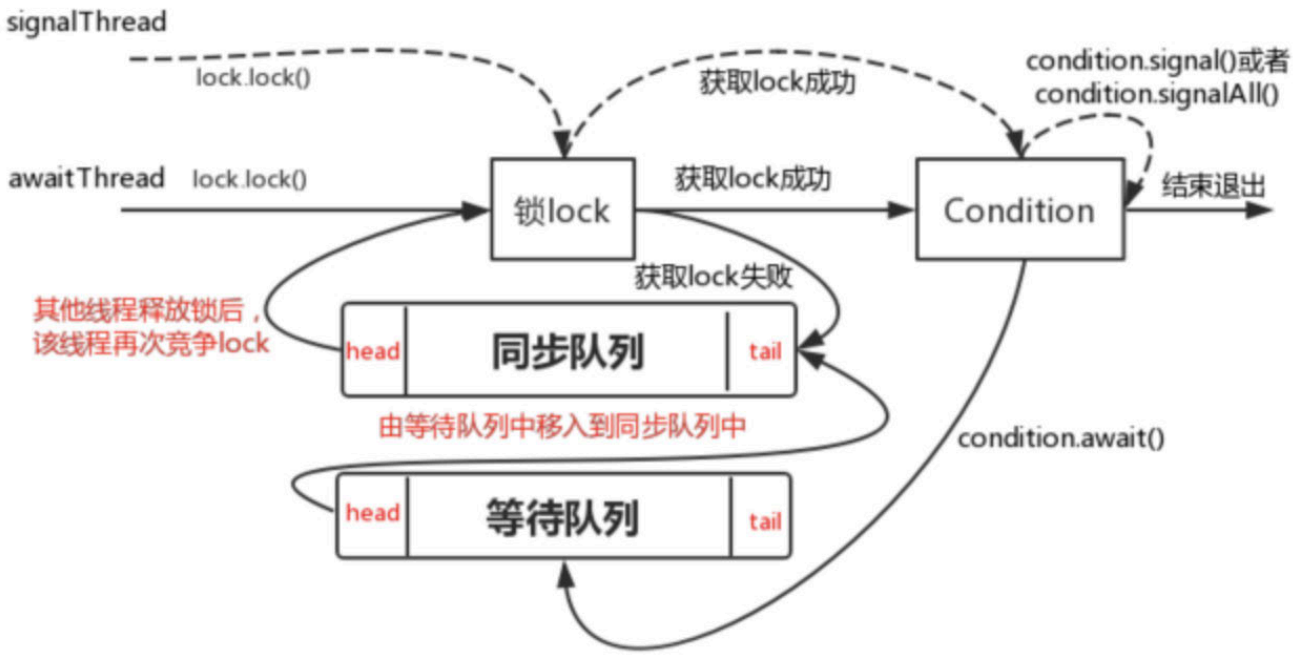
```
private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        Node next = first.nextWaiter;
        first.nextWaiter = null;
        transferForSignal(first);
        first = next;
    } while (first != null);
}
```

该方法会将等待队列中的每一个节点都移入到同步队列中，即“通知”当前调用 `condition.await()` 方法的每一个线程。

await 与 signal/signalAll

文章开篇提到的等待/通知机制，通过 condition 的 await 和 signal/signalAll 方法就可以实现，而这种机制能够解决最经典的问题就是“生产者与消费者问题”，“生产者消费者问题”也是面试的高频考点，后面会细讲，戳链接直达。

await、signal 和 signalAll 方法就像一个开关，控制着线程 A（等待方）和线程 B（通知方）。它们之间的关系可以用下面这幅图来说明，会更贴切：



线程 awaitThread 先通过 `lock.lock()` 方法获取锁，成功后调用 condition.await 方法进入等待队列，而另一个线程 signalThread 通过 `lock.lock()` 方法获取锁成功后调用了 condition.signal 或者 signalAll 方法，使得线程 awaitThread 能够有机会移入到同步队列中，当其他线程释放 lock 后使得线程 awaitThread 能够有机会获取 lock，从而使得线程 awaitThread 能够从 await 方法中退出并执行后续操作。如果 awaitThread 获取 lock 失败会直接进入到同步队列。

Condition使用示例

我们用一个很简单的例子说说 condition 的用法：

```
public class AwaitSignal {
    private static ReentrantLock lock = new ReentrantLock();
    private static Condition condition = lock.newCondition();
    private static volatile boolean flag = false;

    public static void main(String[] args) {
        Thread waiter = new Thread(new waiter());
        waiter.start();
        Thread signaler = new Thread(new signaler());
        signaler.start();
    }

    static class waiter implements Runnable {

        @Override
        public void run() {
            lock.lock();
            try {
                while (!flag) {
                    System.out.println(Thread.currentThread().getName() + "当前条件不满足
等待");
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(Thread.currentThread().getName() + "接收到通知条件满
足");
            } finally {
                lock.unlock();
            }
        }
    }

    static class signaler implements Runnable {

        @Override
        public void run() {
            lock.lock();
            try {
                flag = true;
                condition.signalAll();
            } finally {
                lock.unlock();
            }
        }
```

```
    }
}
```

输出结果为：

```
Thread-0当前条件不满足等待
Thread-0接收到通知，条件满足
```

开启了两个线程 waiter 和 signaler，waiter 线程开始执行的时候由于条件不满足，执行 condition.await 方法使该线程进入等待状态，同时释放锁，signaler 线程获取到锁之后更改条件，并通知所有的等待线程，然后释放锁。这时，waiter 线程获取到锁，由于 signaler 线程更改了条件，此时相对于 waiter 来说，条件满足，继续执行。

小结

Condition 接口是 Java 并发编程中一个重要的组件，用于线程间的协调和通信。它通常与锁（特别是 ReentrantLock）一起使用，为线程提供了一种等待某个条件成真的机制，并允许其他线程在该条件变化时通知等待线程。这为线程间的协调提供了更灵活、更强大的工具。

> 编辑：沉默王二，编辑前的内容主要来自于 CL0610 的 GitHub 仓库https://github.com/CL0610/Java-concurrency，部分内容和图片来自读者阿 Q 说代码的这篇文章终于把Condition的原理讲透彻了，强烈推荐。

第十八节：线程阻塞唤醒类 LockSupport

LockSupprot 用来阻塞和唤醒线程，底层实现依赖于 Unsafe 类（后面会细讲）。

该类包含一组用于阻塞和唤醒线程的静态方法，这些方法主要是围绕 park 和 unpark 展开，话不多说，直接来看一个简单的例子吧。

```
public class LockSupportDemo1 {
    public static void main(String[] args) {
        Thread mainThread = Thread.currentThread();

        // 创建一个线程从1数到1000
        Thread counterThread = new Thread(() -> {
            for (int i = 1; i <= 1000; i++) {
                System.out.println(i);
                if (i == 500) {
                    // 当数到500时，唤醒主线程
                    LockSupport.unpark(mainThread);
                }
            }
        });

        counterThread.start();

        // 主线程调用park
        LockSupport.park();
        System.out.println("Main thread was unparked.");
    }
}
```

上面的代码中，当 counterThread 数到 500 时，它会唤醒 mainThread。而 mainThread 在调用 park 方法时会被阻塞，直到被 unpark。

LockSupport 中的方法不多，这里将这些方法做一个总结：

阻塞线程

1. `void park()`：阻塞当前线程，如果调用 unpark 方法或线程被中断，则该线程将变得可运行。请注意，park 不会抛出 InterruptedException，因此线程必须单独检查其中断状态。
2. `void park(Object blocker)`：功能同方法 1，入参增加一个 Object 对象，用来记录导致线程阻塞的对象，方便问题排查。
3. `void parkNanos(long nanos)`：阻塞当前线程一定的纳秒时间，或直到被 unpark 调用，或线程被中断。
4. `void parkNanos(Object blocker, long nanos)`：功能同方法 3，入参增加一个 Object 对象，用来记录导致线程阻塞的对象，方便问题排查。
5. `void parkUntil(long deadline)`：阻塞当前线程直到某个指定的截止时间（以毫秒为单位），或直到被 unpark 调用，或线程被中断。

6. `void parkUntil(Object blocker, long deadline)`：功能同方法 5，入参增加一个 Object 对象，用来记录导致线程阻塞的对象，方便问题排查。

唤醒线程

`void unpark(Thread thread)`：唤醒一个由 park 方法阻塞的线程。如果该线程未被阻塞，那么下一次调用 park 时将立即返回。这允许“先发制人”式的唤醒机制。

实际上，LockSupport 阻塞和唤醒线程的功能依赖于 `sun.misc.Unsafe`，这是一个很底层的类，后面这篇文章会细讲，比如 LockSupport 的 park 方法是通过 `unsafe.park()` 方法实现的。

Dump 线程

"Dump 线程"通常是指获取线程的当前状态和调用堆栈的详细快照。这可以提供关于线程正在执行什么操作以及线程在代码的哪个部分的重要信息。

下面是线程转储中可能包括的一些信息：

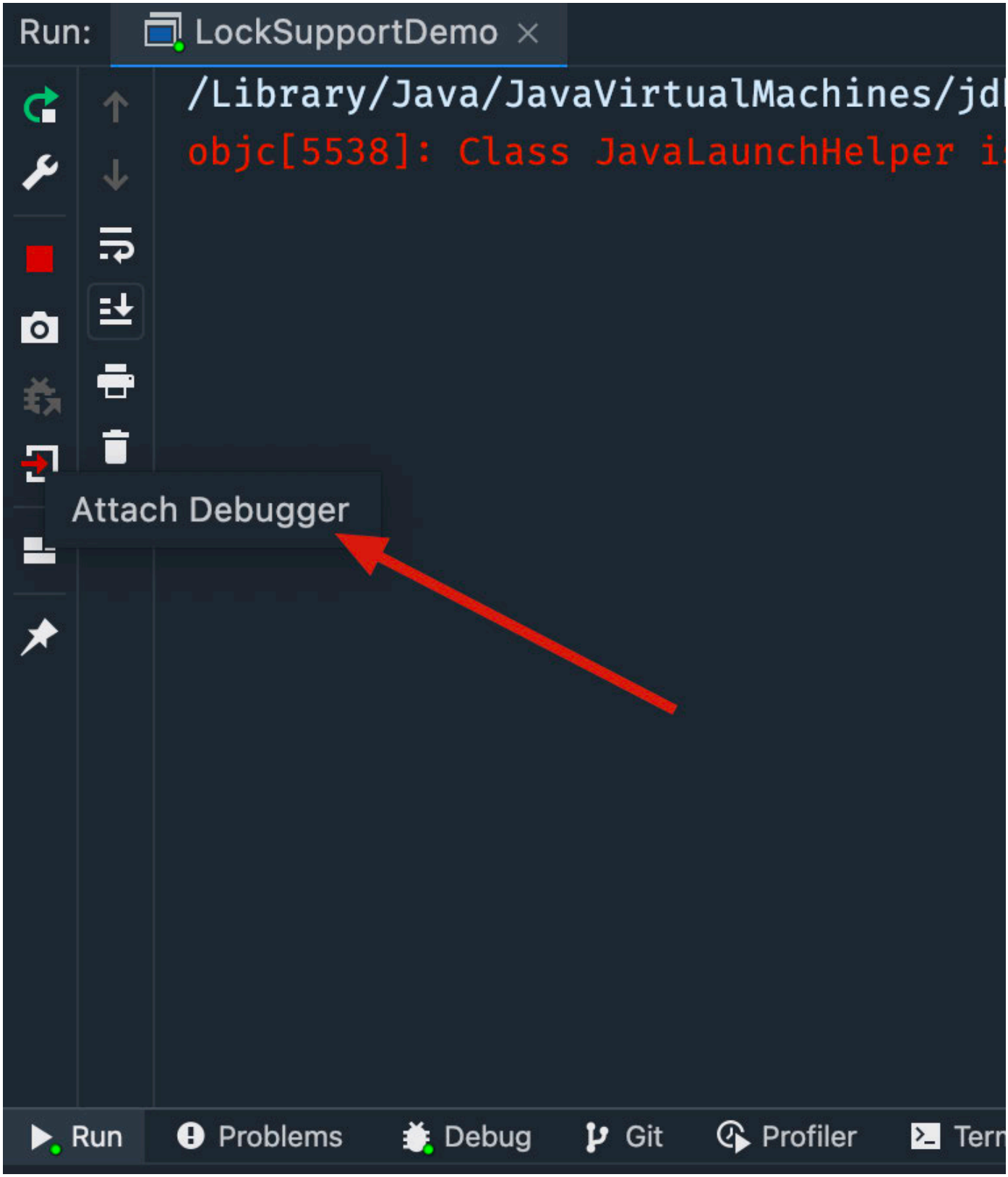
- 线程 ID 和名称：线程的唯一标识符和可读名称。
- 线程状态：线程的当前状态，例如运行（RUNNABLE）、等待（WAITING）、睡眠（TIMED_WAITING）或阻塞（BLOCKED）。
- 调用堆栈：线程的调用堆栈跟踪，显示线程从当前执行点回溯到初始调用的完整方法调用序列。
- 锁信息：如果线程正在等待或持有锁，线程转储通常还包括有关这些锁的信息。

线程转储可以通过各种方式获得，例如使用 Java 的 jstack 工具，或从 Java VisualVM、Java Mission Control 等工具获取。

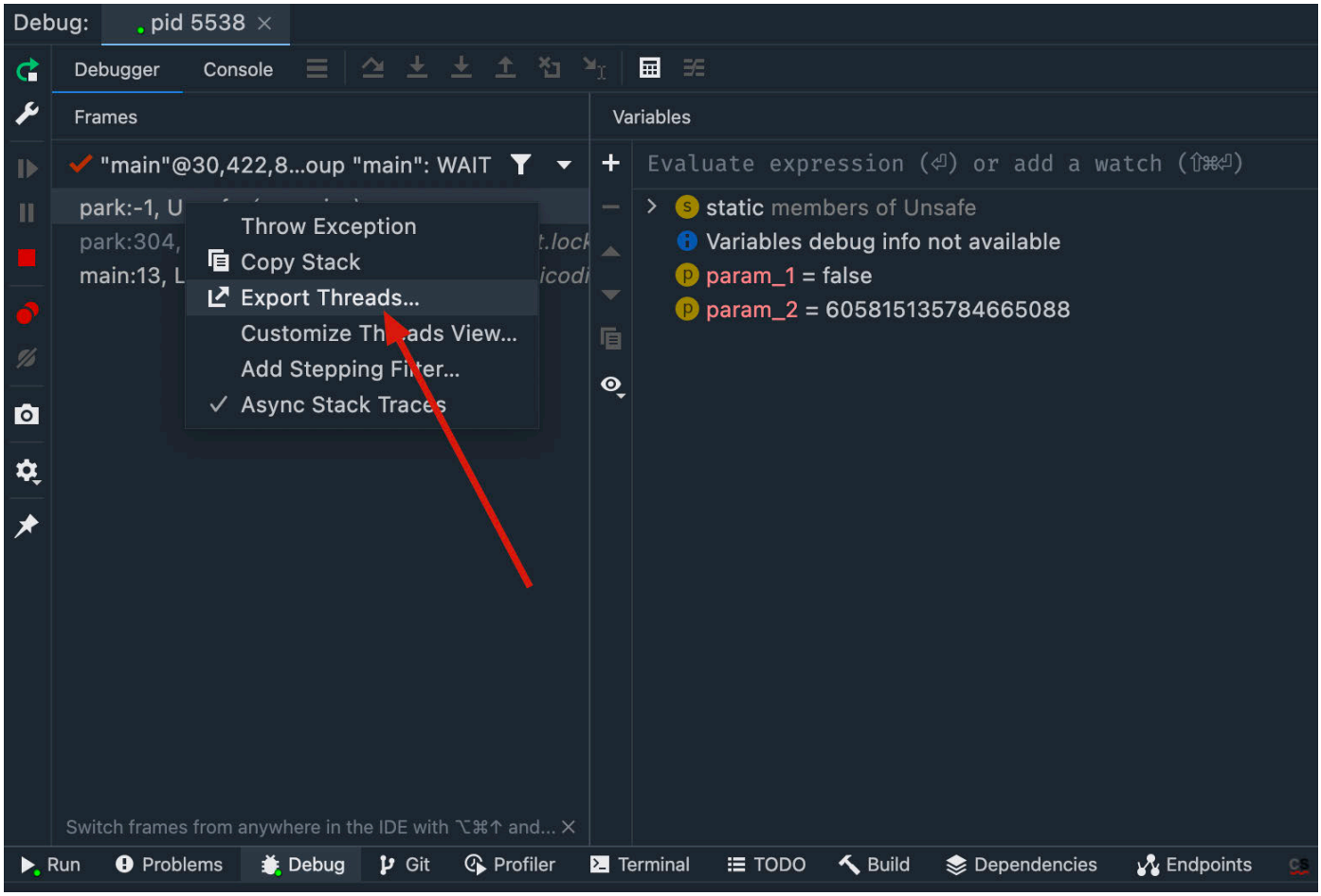
下面是一个简单的例子，通过 LockSupport 阻塞线程，然后通过 Intellij IDEA 查看 dump 线程信息。

```
public class LockSupportDemo {
    public static void main(String[] args) {
        LockSupport.park();
    }
}
```

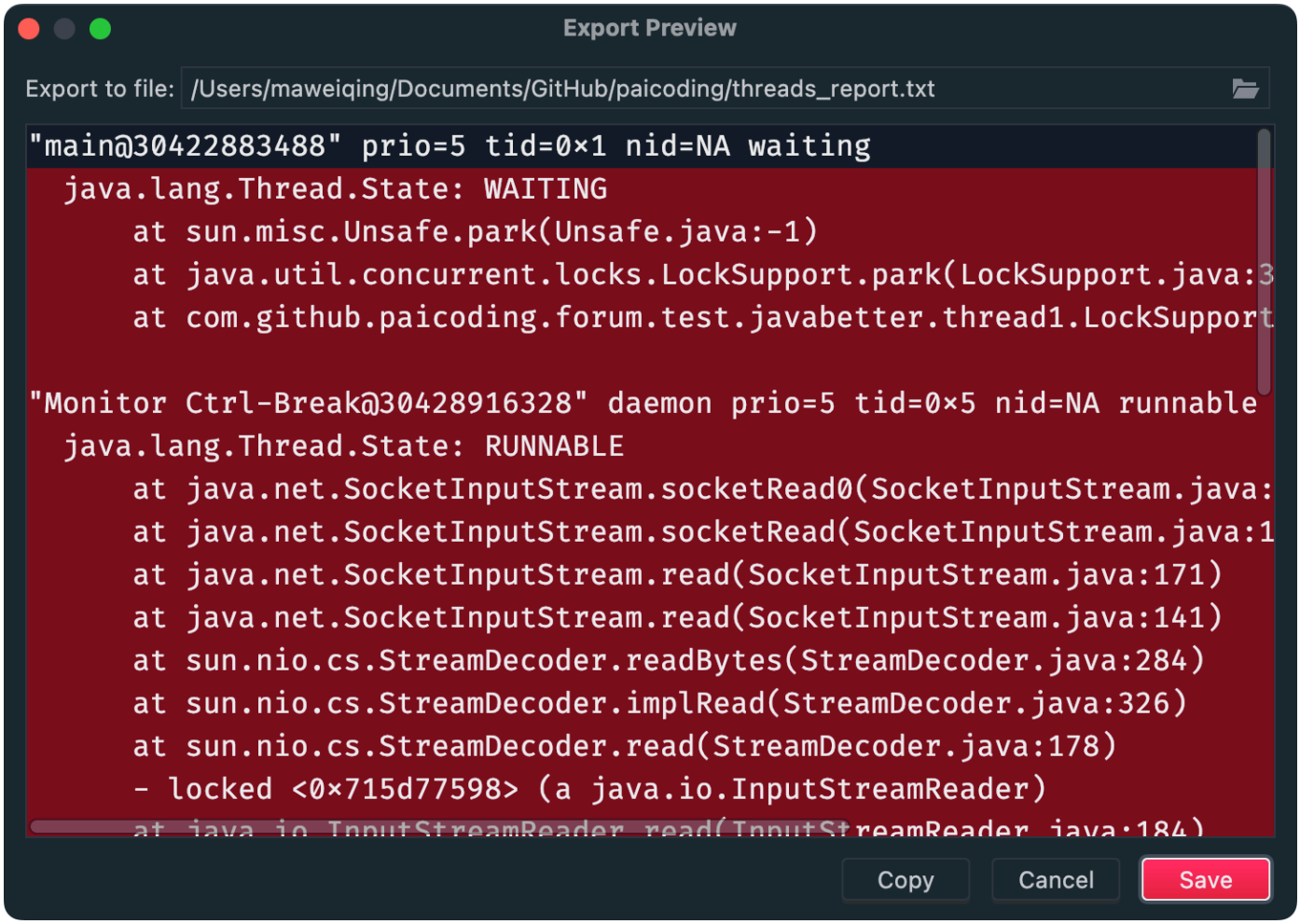
运行，然后再 Run 面板中点击「attach debugger」。



然后在 debugger 面板中右键选择「export thread」。



就可以看了 Dump 线程信息了。



调用 park()方法 dump 线程：

```
"main" #1 prio=5 os_prio=0 tid=0x02cdcc00 nid=0x2b48 waiting on condition [0x00d6f000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:304)
        at learn.LockSupportDemo.main(LockSupportDemo.java:7)
```

调用 park(Object blocker)方法 dump 线程

```
"main" #1 prio=5 os_prio=0 tid=0x0069cc00 nid=0x6c0 waiting on condition [0x00dcf000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x048c2d18> (a java.lang.String)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at learn.LockSupportDemo.main(LockSupportDemo.java:7)
```

分别调用无参和有参的 park 方法，然后通过 dump 线程信息可以看出，带 Object 的 park 方法相较于无参的 park 方法会增加 `parking to wait for  <0x048c2d18> (a java.lang.String)` 的信息，这种信息类似于记录“案发现场”，有助于我们开发者迅速发现问题并解决问题。

有意思的事情是，Java 1.5 推出 LockSupport 时遗漏了阻塞信息的描述，于是在 Java 1.6 的时候进行了补充。

与 synchronzed 的区别

还有一点需要注意的是：**synchronzed 会使线程阻塞，线程会进入 BLOCKED 状态，而调用 LockSupprt 方法阻塞线程会使线程进入到 WAITING 状态。**

来一个简单的例子演示一下该怎么用。

```
public class LockSupportExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is parked now");
            LockSupport.park();
            System.out.println("Thread is unparked now");
        });

        thread.start();

        try {
            Thread.sleep(3000); // 主线程等待3秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        LockSupport.unpark(thread); // 主线程唤醒阻塞的线程
    }
}
```

thread 线程调用 `LockSupport.park()` 使 thread 阻塞，当 mian 线程睡眠 3 秒结束后通过 `LockSupport.unpark(thread)` 方法唤醒 thread 线程，thread 线程被唤醒后会执行后续的操作。另外，**`LockSupport.unpark(thread)` 可以指定线程对象唤醒指定的线程**。

运行结果：

```
Thread is parked now
Thread is unparked now
```

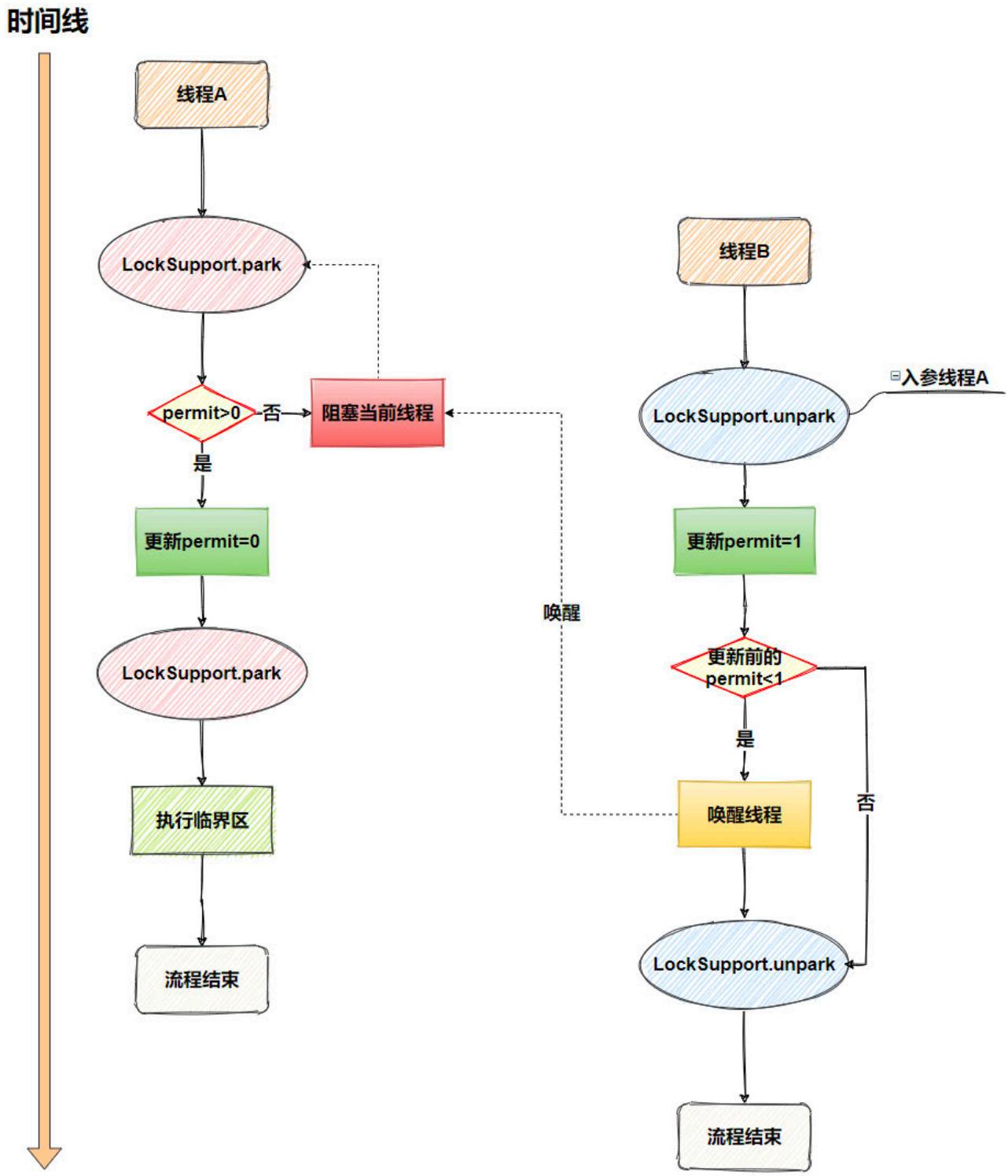
设计思路

LockSupport 的设计思路是通过许可证来实现的，就像汽车上高速公路，入口处要获取通行卡，出口处要交出通行卡，如果没有通行卡你就无法出站，当然你可以选择补一张通行卡。

LockSupport 会为使用它的线程关联一个许可证（permit）状态，permit 的语义「是否拥有许可」，0 代表否，1 代表是，默认是 0。

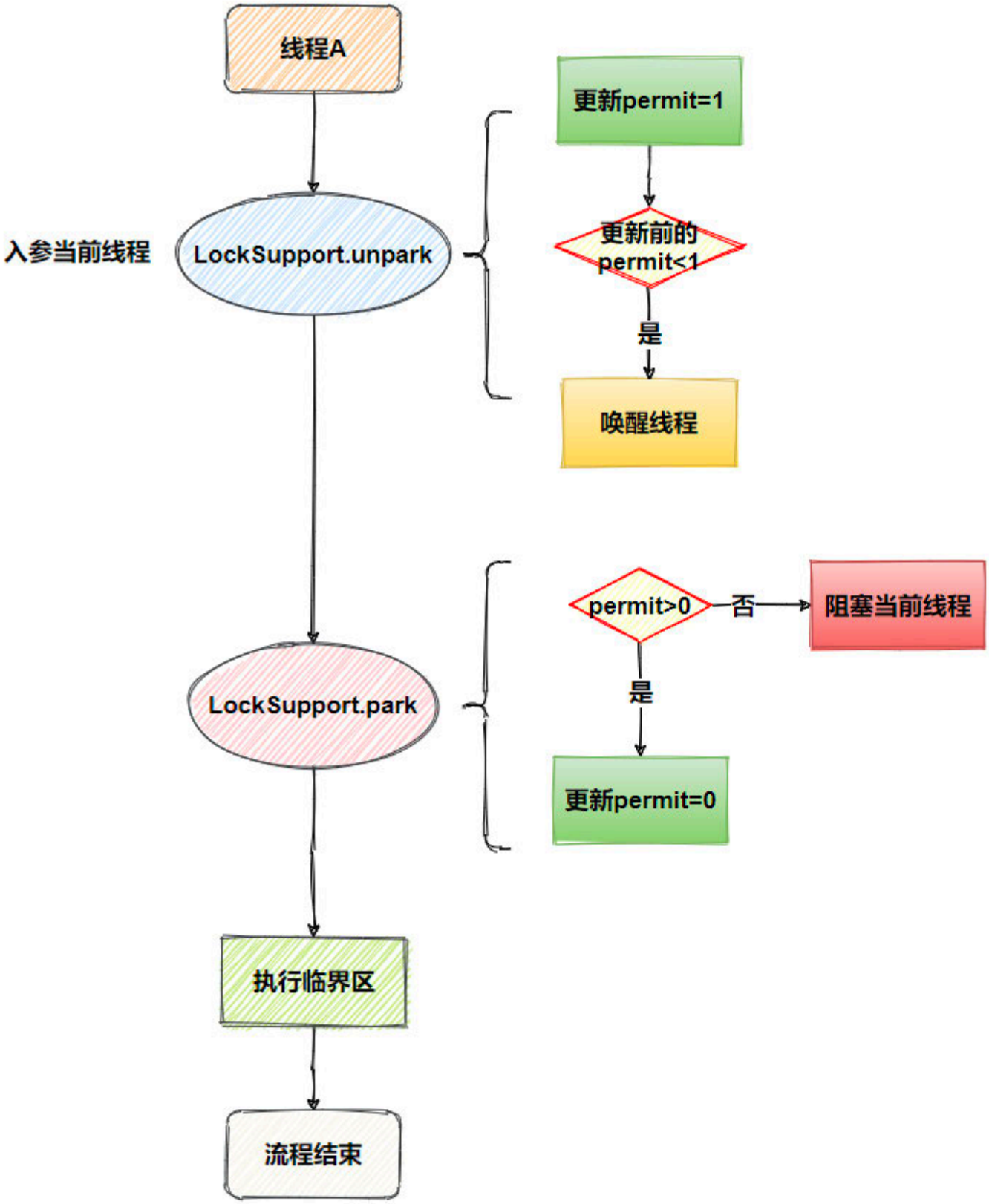
- `LockSupport.unpark`：指定线程关联的 permit 直接更新为 1，如果更新前的`permit<1`，唤醒指定线程
- `LockSupport.park`：当前线程关联的 permit 如果>0，直接把 permit 更新为 0，否则阻塞当前线程

来看时间线：



- 线程 A 执行`LockSupport.park`，发现 permit 为 0，未持有许可证，阻塞线程 A
- 线程 B 执行`LockSupport.unpark`（入参线程 A），为 A 线程设置许可证，permit 更新为 1，唤醒线程 A
- 线程 B 流程结束
- 线程 A 被唤醒，发现 permit 为 1，消费许可证，permit 更新为 0
- 线程 A 执行临界区
- 线程 A 流程结束

经过上面的分析得出结论 unpark 的语义明确为「使线程持有许可证」，park 的语义明确为「消费线程持有的许可」，所以 unpark 与 park 的执行顺序没有强制要求，只要控制好使用的线程即可，`unpark=>park`执行流程如下



- permit 默认是 0，线程 A 执行 LockSupport.unpark，permit 更新为 1，线程 A 持有许可证
- 线程 A 执行 LockSupport.park，此时 permit 是 1，消费许可证，permit 更新为 0
- 执行临界区
- 流程结束

最后再补充下 park 的注意点，因 park 阻塞的线程不仅仅会被 unpark 唤醒，还可能会被线程中断（`Thread.interrupt`）唤醒，而且不会抛出 InterruptedException 异常，所以建议在 park 后自行判断线程中断状态，来做对应的业务处理。

为什么推荐使用 LockSupport 来做线程的阻塞与唤醒（线程间协同工作），因为它具备如下优点：

- 以线程为操作对象更符合阻塞线程的直观语义
- 操作更精准，可以准确地唤醒某一个线程（notify 随机唤醒一个线程，notifyAll 唤醒所有等待的线程）
- 无需竞争锁对象（以线程作为操作对象），不会因竞争锁对象产生死锁问题
- unpark 与 park 没有严格的执行顺序，不会因执行顺序引起死锁问题，比如「Thread.suspend 和 Thread.resume」没按照严格顺序执行，就会产生死锁

面试题

阿里面试官：有 3 个独立的线程，一个只会输出 A，一个只会输出 B，一个只会输出 C，在三个线程启动的情况下，请用合理的方式让他们按顺序打印 ABCABC。

```
public class ABCPrinter {
    private static Thread t1, t2, t3;

    public static void main(String[] args) {
        t1 = new Thread(() -> {
            for (int i = 0; i < 2; i++) {
                LockSupport.park();
                System.out.print("A");
                LockSupport.unpark(t2);
            }
        });

        t2 = new Thread(() -> {
            for (int i = 0; i < 2; i++) {
                LockSupport.park();
                System.out.print("B");
                LockSupport.unpark(t3);
            }
        });

        t3 = new Thread(() -> {
            for (int i = 0; i < 2; i++) {
                LockSupport.park();
                System.out.print("C");
                LockSupport.unpark(t1);
            }
        });

        t1.start();
        t2.start();
        t3.start();
```

```
        // 主线程稍微等待一下，确保其他线程已经启动并且进入park状态。
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 启动整个流程
        LockSupport.unpark(t1);
    }
}
```

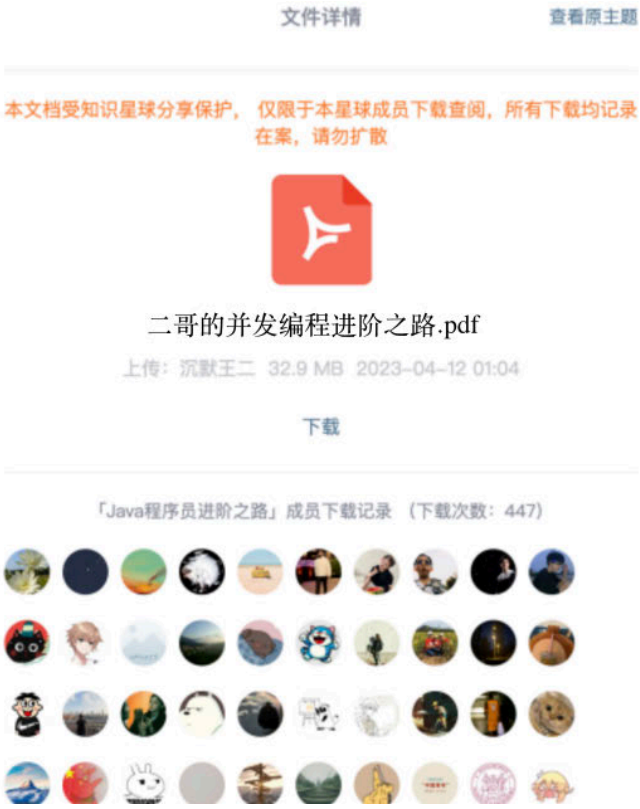
这里的实现方式是：

- 我们首先为每个线程创建一个 Runnable，使其在循环中 park 自身，然后输出其对应的字符，并 unpark 下一个线程。
- 所有线程在启动后会先调用 park 将自己阻塞。
- 主线程稍微延迟后调用 t1 的 unpark，启动整个打印流程。
  这样可以保证每个线程按照预期的顺序进行工作。

小结

LockSupport 提供了一种更底层和灵活的线程调度方式。它不依赖于同步块或特定的锁对象。可以用于构建更复杂的同步结构，例如自定义锁或并发容器。LockSupport.park 与 LockSupport.unpark 的组合使得线程之间的精确控制变得更容易，而不需要复杂的同步逻辑和对象监视。

> 编辑：沉默王二，编辑前的内容主要来自于 CL0610 的 GitHub 仓库https://github.com/CL0610/Java-concurrency，另外一部分内容和图片来自于读者程序猿阿星的写给小白看的 LockSupport，强烈推荐。



第十九节：Java的并发容器

Java 的并发集合容器提供了在多线程环境中高效访问和操作的数据结构。这些容器通过内部的同步机制实现了线程安全，使得开发者无需显式同步代码就能在并发环境下安全使用，比如说：ConcurrentHashMap、阻塞队列和 CopyOnWrite 容器等。

java.util 包下提供了一些容器类（集合框架），其中 Vector 和 Hashtable 是线程安全的，但实现方式比较粗暴，通过在方法上加「sychronized」关键字实现。

但即便是 Vector 这样线程安全的类，在应对多线程的复合操作时也需要在客户端继续加锁以保证原子性。来看下面的例子：

```
public class TestVector {
  private Vector<String> vector;

  //方法一
  public  Object getLast(Vector vector) {
      int lastIndex = vector.size() - 1;
      return vector.get(lastIndex);
  }

  //方法二
  public  void deleteLast(Vector vector) {
      int lastIndex = vector.size() - 1;
      vector.remove(lastIndex);
  }
```

```
    //方法三
    public  Object getLastSysnchronized(Vector vector) {
      synchronized(vector){
        int lastIndex = vector.size() - 1;
        return vector.get(lastIndex);
      }
    }

    //方法四
    public  void deleteLastSysnchronized(Vector vector) {
      synchronized (vector){
        int lastIndex = vector.size() - 1;
        vector.remove(lastIndex);
      }
    }

}
```

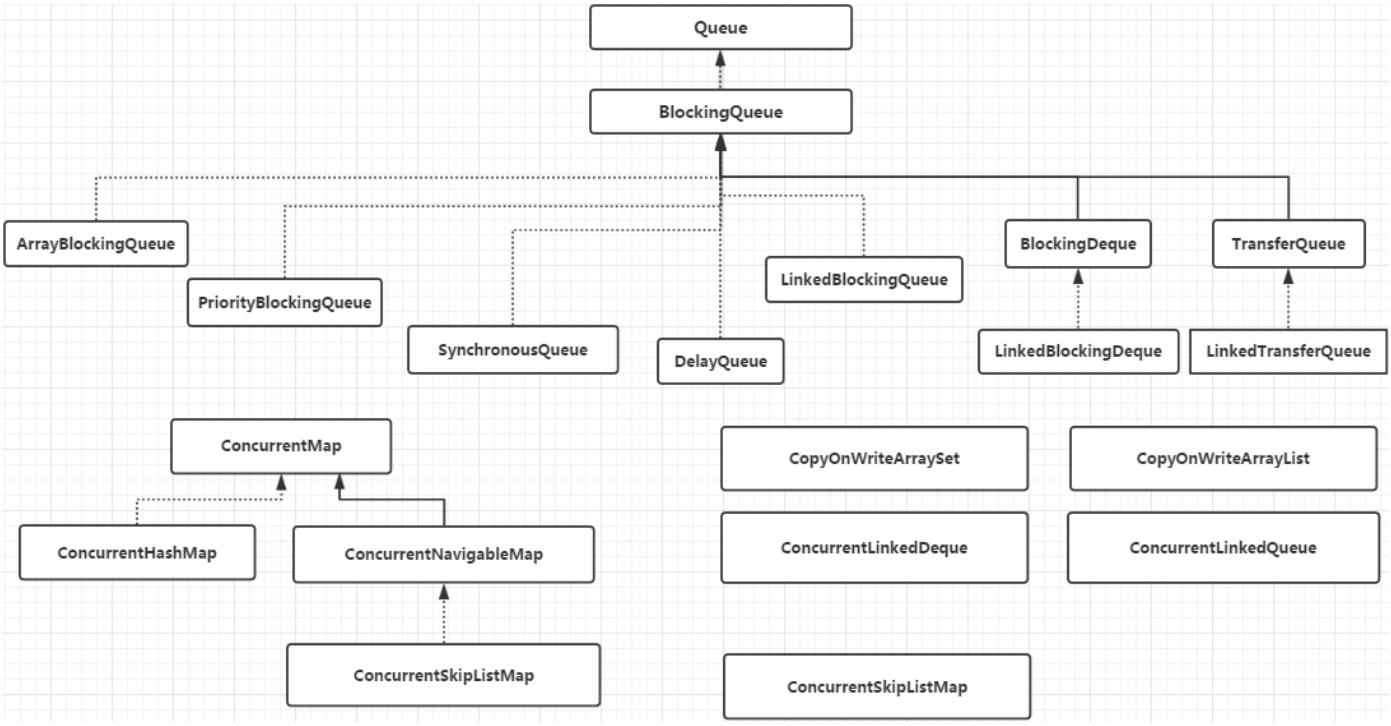
如果方法一和方法二是一个组合的话，那么当方法一获取到了`vector`的 size 之后，方法二已经执行完毕，这样就会导致程序出现错误。

如果方法三与方法四组合的话，就还需在内部加锁来保证`vector`上的原子性操作。

于是并发容器就应用而生了，它们是线程安全的，可以在多线程环境下高效地访问和操作数据，而不需要额外的同步措施。

并发容器类

整体架构如下图所示：



并发 Map

ConcurrentMap 接口

ConcurrentMap 接口继承了 Map 接口，在 Map 接口的基础上又定义了四个方法：

```
public interface ConcurrentMap<K, V> extends Map<K, V> {

    //插入元素
    V putIfAbsent(K key, V value);

    //移除元素
    boolean remove(Object key, Object value);

    //替换元素
    boolean replace(K key, V oldValue, V newValue);

    //替换元素
    V replace(K key, V value);

}
```

putIfAbsent：与原有 put 方法不同的是，putIfAbsent 如果插入的 key 相同，则不替换原有的 value 值；

remove：与原有 remove 方法不同的是，新 remove 方法中增加了对 value 的判断，如果要删除的 key-value 不能与 Map 中原有的 key-value 对应上，则不会删除该元素；

replace(K,V,V)：增加了对 value 值的判断，如果 key-oldValue 能与 Map 中原有的 key-value 对应上，才进行替换操作；

replace(K,V)：与上面的 replace 不同的是，此 replace 不会对 Map 中原有的 key-value 进行比较，如果 key 存在则直接替换；

ConcurrentHashMap

ConcurrentHashMap 同 HashMap 一样，也是基于散列表的 map，但是它提供了一种与 Hashtable 完全不同的加锁策略，提供了更高效的并发性和伸缩性。

后面我们会单独开一篇来详细介绍 ConcurrentHashMap，戳链接直达。

ConcurrentSkipListMap

ConcurrentNavigableMap 接口继承了 NavigableMap 接口，这个接口提供了针对给定搜索目标返回最接近匹配项的导航方法。

ConcurrentNavigableMap 接口的主要实现类是 ConcurrentSkipListMap 类。从名字上来看，它的底层使用的是跳表（SkipList）。跳表是一种"空间换时间"的数据结构，可以使用 CAS 来保证并发安全性。

与 ConcurrentHashMap 的读密集操作相比，ConcurrentSkipListMap 的读和写操作的性能相对较低。这是由其数据结构导致的，因为跳表的插入和删除需要更复杂的指针操作。然而，ConcurrentSkipListMap 提供了有序性，这是 ConcurrentHashMap 所没有的。

ConcurrentSkipListMap 适用于需要线程安全的同时又需要元素有序的场合。如果不需要有序，ConcurrentHashMap 可能是更好的选择，因为它通常具有更高的性能。

并发 Queue

JDK 并没有提供线程安全的 List 类，因为对 List 来说，**很难去开发一个通用并且没有并发瓶颈的线程安全的 List**。因为即使简单的读操作，比如 `contains()`，也需要再搜索的时候锁住整个 list。

所以退一步，JDK 提供了队列和双端队列的线程安全类：ConcurrentLinkedQueue 和 ConcurrentLinkedDeque。因为队列相对于 List 来说，有更多的限制。这两个类是使用 CAS 来实现线程安全的。

我们会在后面单独开一篇来详细介绍ConcurrentLinkedQueue，戳链接直达。

并发 Set

ConcurrentSkipListSet 是线程安全的有序集合。底层是使用 ConcurrentSkipListMap 来实现。

谷歌的 Guava实现了一个线程安全的 ConcurrentHashSet：

```
Set<String> s = Sets.newConcurrentHashSet();
```

Set 日常开发中用的并不多，所以这里就不展开细讲了。

阻塞队列

我们假设一种场景，生产者一直生产资源，消费者一直消费资源（后面会细讲，戳链接直达），资源存储在一个缓冲池中，生产者将生产的资源存进缓冲池中，消费者从缓冲池中拿到资源进行消费，这就是大名鼎鼎的**生产者-消费者模式**。

该模式能够简化开发过程，一方面消除了生产者类与消费者类之间的代码依赖性，另一方面将生产数据的过程与使用数据的过程解耦简化负载。

我们自己 coding 实现这个模式的时候，因为需要让**多个线程操作共享变量**（即资源），所以很容易引发**线程安全问题**，造成**重复消费**和**死锁**，尤其是生产者和消费者存在多个的情况。另外，当缓冲池空了，我们需要阻塞消费者，唤醒生产者；当缓冲池满了，我们需要阻塞生产者，唤醒消费者，这些个**等待-唤醒**逻辑都需要自己实现。

这么容易出错的事情，JDK 当然帮我们做啦，这就是阻塞队列（BlockingQueue），**你只管往里面存、取就行，而不用担心多线程环境下存、取共享变量的线程安全问题。**

> BlockingQueue 是 Java util.concurrent 包下重要的数据结构，区别于普通的队列，BlockingQueue 提供了**线程安全的队列访问方式**，并发包下很多高级同步类的实现都是基于 BlockingQueue 实现的。

BlockingQueue 一般用于生产者-消费者模式，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。**BlockingQueue 就是存放元素的容器。**

BlockingQueue 的操作方法

阻塞队列提供了四组不同的方法用于插入、移除、检查元素：

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	**put(e)**	offer(e,time,unit)
移除方法	remove()	poll()	**take()**	poll(time,unit)
检查方法	element()	peek()	-	-

- 抛出异常：如果操作无法立即执行，会抛异常。当阻塞队列满时候，再往队列里插入元素，会抛出`IllegalStateException("Queue full")`异常。当队列为空时，从队列里获取元素时会抛出 NoSuchElementException 异常 。
- 返回特殊值：如果操作无法立即执行，会返回一个特殊值，通常是 true / false。
- 一直阻塞：如果操作无法立即执行，则一直阻塞或者响应中断。
- 超时退出：如果操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功，通常是 true / false。

注意：

- 不能往阻塞队列中插入 null，会抛出空指针异常。
- 可以访问阻塞队列中的任意元素，调用`remove(o)`可以将队列之中的特定对象移除，但并不高效，尽量避免使用。

我们会在后面单独开一篇BlockingQueue来细讲，戳链接直达。

BlockingQueue 的实现类

ArrayBlockingQueue

由**数组**结构组成的**有界**阻塞队列。内部结构是数组，具有数组的特性。

```
public ArrayBlockingQueue(int capacity, boolean fair){
 //..省略代码
}
```

可以初始化队列大小，一旦初始化将不能改变。构造方法中的 fair 表示控制对象的内部锁是否采用公平锁，默认是**非公平锁**。

LinkedBlockingQueue

由**链表**结构组成的**有界**阻塞队列。内部结构是链表，具有链表的特性。默认队列的大小是`Integer.MAX_VALUE`，也可以指定大小。此队列按照**先进先出**的原则对元素进行排序。

DelayQueue

该队列中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。注入其中的元素必须实现`java.util.concurrent.Delayed`接口。

DelayQueue 是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

PriorityBlockingQueue

基于优先级的无界阻塞队列（优先级的判断通过构造函数传入的 Compator 对象来决定），内部控制线程同步的锁采用的是非公平锁。

> 网上大部分博客上**PriorityBlockingQueue**为公平锁，其实是不对的，查阅源码（感谢github:**ambition0802**同学的指出）：

```
public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator) {
    this.lock = new ReentrantLock(); //默认构造方法-非公平锁
    ...//其余代码略
}
```

SynchronousQueue

这个队列比较特殊，**没有任何内部容量**，甚至连一个队列的容量都没有。并且每个 put 必须等待一个 take，反之亦然。

需要区别容量为 1 的 ArrayBlockingQueue、LinkedBlockingQueue。

以下方法的返回值，可以帮助理解这个队列：

- `iterator()` 永远返回空，因为里面没有东西
- `peek()` 永远返回 null
- `put()` 往 queue 放进去一个 element 以后就一直 wait 直到有其他 thread 进来把这个 element 取走。
- `offer()` 往 queue 里放一个 element 后立即返回，如果碰巧这个 element 被另一个 thread 取走了，offer 方法返回 true，认为 offer 成功；否则返回 false。
- `take()` 取出并且 remove 掉 queue 里的 element，取不到东西他会一直等。
- `poll()` 取出并且 remove 掉 queue 里的 element，只有到碰巧另外一个线程正在往 queue 里 offer 数据或者 put 数据的时候，该方法才会取到东西。否则立即返回 null。
- `isEmpty()` 永远返回 true
- `remove()&removeAll()` 永远返回 false

注意

PriorityBlockingQueue不会阻塞数据生产者（因为队列是无界的），而只会在没有可消费的数据时阻塞数据的消费者。因此使用的时候要特别注意，**生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。**对于使用默认大小的**LinkedBlockingQueue**也是一样的。

CopyOnWrite 容器

在聊 CopyOnWrite 容器之前我们先来谈谈什么是 CopyOnWrite 机制，CopyOnWrite 是计算机设计领域的一种优化策略，也是一种在并发场景下常用的设计思想——写入时复制。

什么是写入时复制呢?

就是当有多个调用者同时去请求一个资源数据的时候，有一个调用者出于某些原因需要对当前的数据源进行修改，这个时候系统将会复制一个当前数据源的副本给调用者修改。

CopyOnWrite 容器即**写时复制的容器**，当我们往一个容器中添加元素的时候，不直接往容器中添加，而是将当前容器进行 copy，复制出来一个新的容器，然后向新容器中添加我们需要的元素，最后将原容器的引用指向新容器。

这样做的好处在于，我们可以在并发的场景下对容器进行"读操作"而不需要"加锁"，从而达到读写分离的目的。从 JDK 1.5 开始 Java 并发包里提供了两个使用 CopyOnWrite 机制实现的并发容器，分别是 CopyOnWriteArrayList（后面会细讲，戳链接直达） 和 CopyOnWriteArraySet（不常用）。

小结

本文主要介绍了并发包中的三个重要的容器类，Map、阻塞队列和 CopyOnWrite 容器，Map 用于存储键值对，阻塞队列用于生产者-消费者模型，而 CopyOnWrite 容器用于“读多写少”的并发场景。

> 编辑：沉默王二，部分内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，推荐阅读：時光以北这篇 ConcurrentSkipListMap 讲的很不错，可以学习。

第二十节：并发容器ConcurrentHashMap

ConcurrentHashMap 是 Java 并发包 (java.util.concurrent) 中的一种线程安全的哈希表实现。

HashMap 在多线程环境下扩容会出现 CPU 接近 100% 的情况，因为 HashMap 并不是线程安全的，我们可以通过 Collections 的`Map<K,V> synchronizedMap(Map<K,V> m)`将 HashMap 包装成一个线程安全的 map。

比如 SynchronzedMap 的 put 方法源码就是加锁过的：

```
public V put(K key, V value) {
    synchronized (mutex) {return m.put(key, value);}
}
```

synchronized 同步代码块的方式我们前面也讲过了，大家应该都还有印象。

不过，这并不是最优雅的方式。Doug Lea 大师不遗余力的为我们创造了一些线程安全的并发容器，让每一个 Java 开发人员都倍感幸福。相对于 HashMap，ConcurrentHashMap 就是线程安全的 map，其中**利用了锁分段的思想大大提高了并发的效率**。

在介绍并发容器的时候，我们也曾提到过 ConcurrentHashMap，它从 JDK 1.8 开始有了较大的变化，光是代码量就足足增加了很多。

1.8 版本舍弃了 segment，并且使用了大量的 synchronized，以及 CAS 无锁操作以保证 ConcurrentHashMap 的线程安全性。

为什么不用 ReentrantLock 而是 synchronzied 呢?

实际上，synchronzied 做了很多的优化，这个我们前面也讲过了，包括偏向锁、轻量级锁、重量级锁，可以依次向上升级锁状态，因此，synchronized 相较于 ReentrantLock 的性能其实差不多，甚至在某些情况更优。

ConcurrentHashMap 的变化

ConcurrentHashMap 在 JDK 1.7 和 JDK 1.8 中有一些区别。这里我们分开介绍一下。

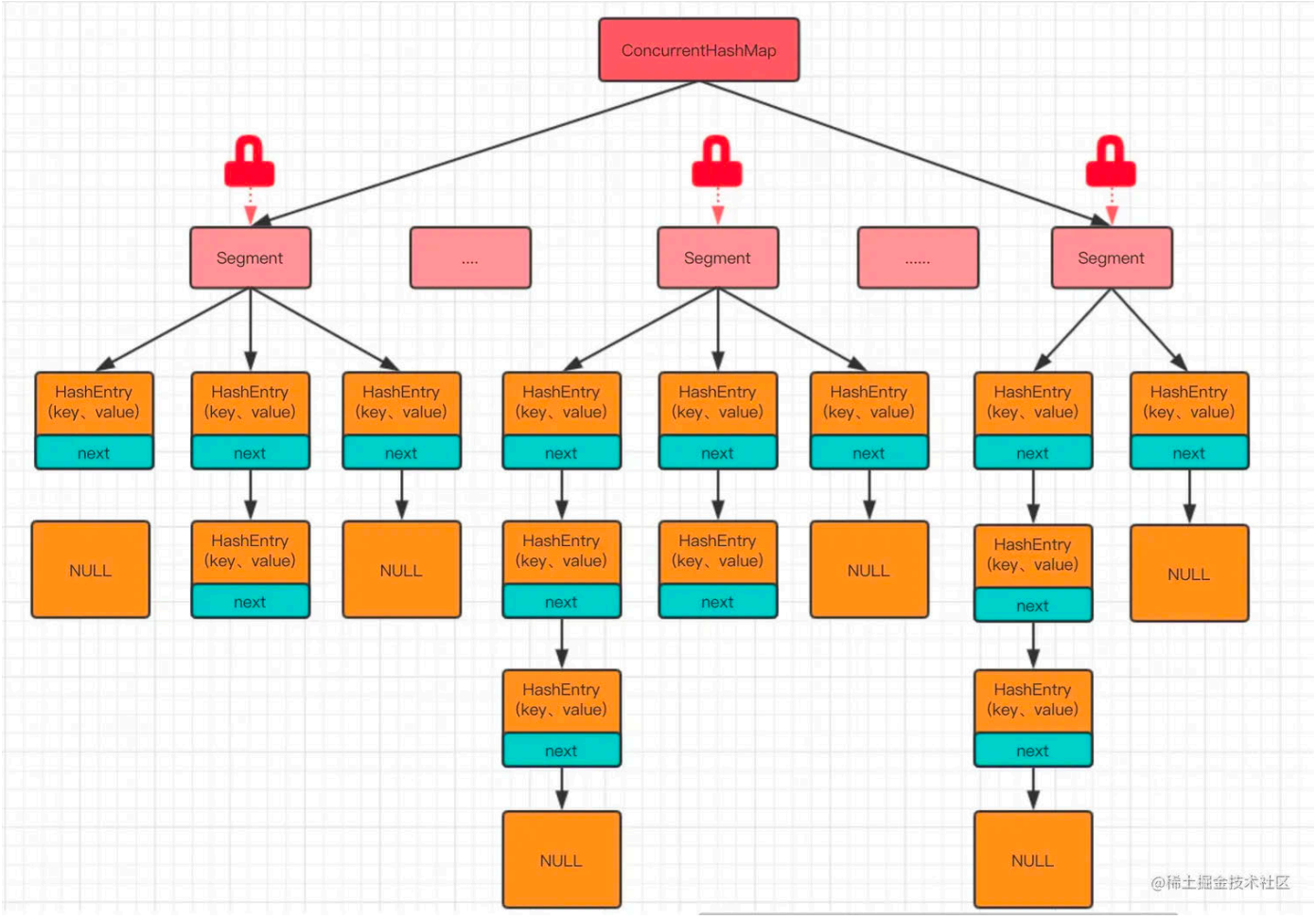
JDK 1.7

ConcurrentHashMap 在 JDK 1.7 中，提供了一种粒度更细的加锁机制，这种机制叫分段锁「Lock Striping」。整个哈希表被分为多个段，每个段都独立锁定。读取操作不需要锁，写入操作仅锁定相关的段。这减小了锁冲突的几率，从而提高了并发性能。

这种机制的优点：在并发环境下将实现更高的吞吐量，而在单线程环境下只损失非常小的性能。

可以这样理解分段锁，就是**将数据分段，对每一段数据分配一把锁**。当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

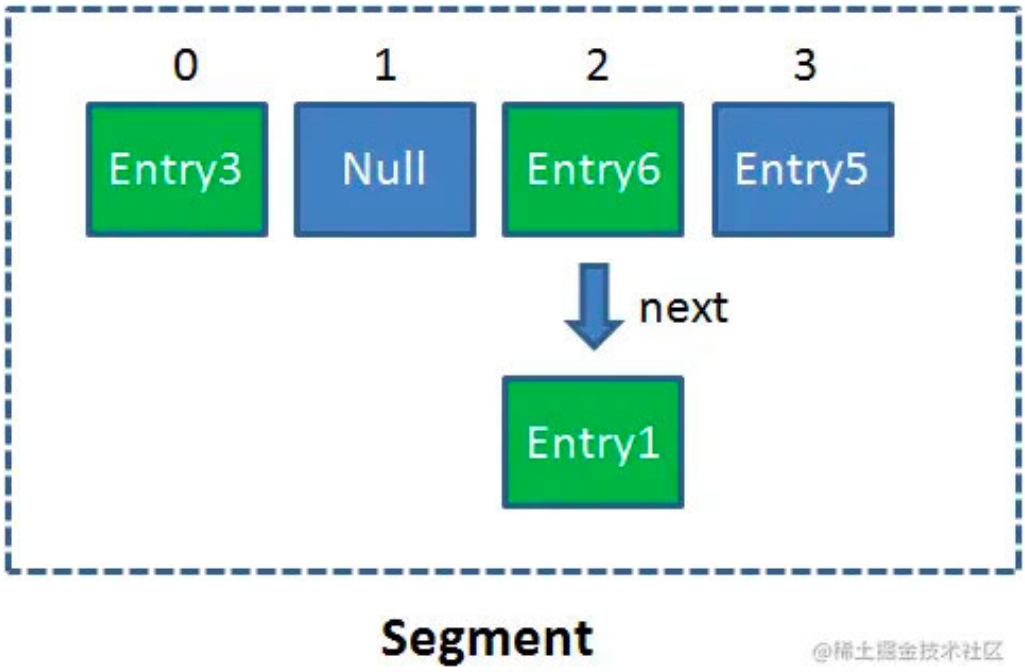
有些方法需要跨段，比如`size()`、`isEmpty()`、`containsValue()`，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完后，再按顺序释放所有段的锁。如下图：



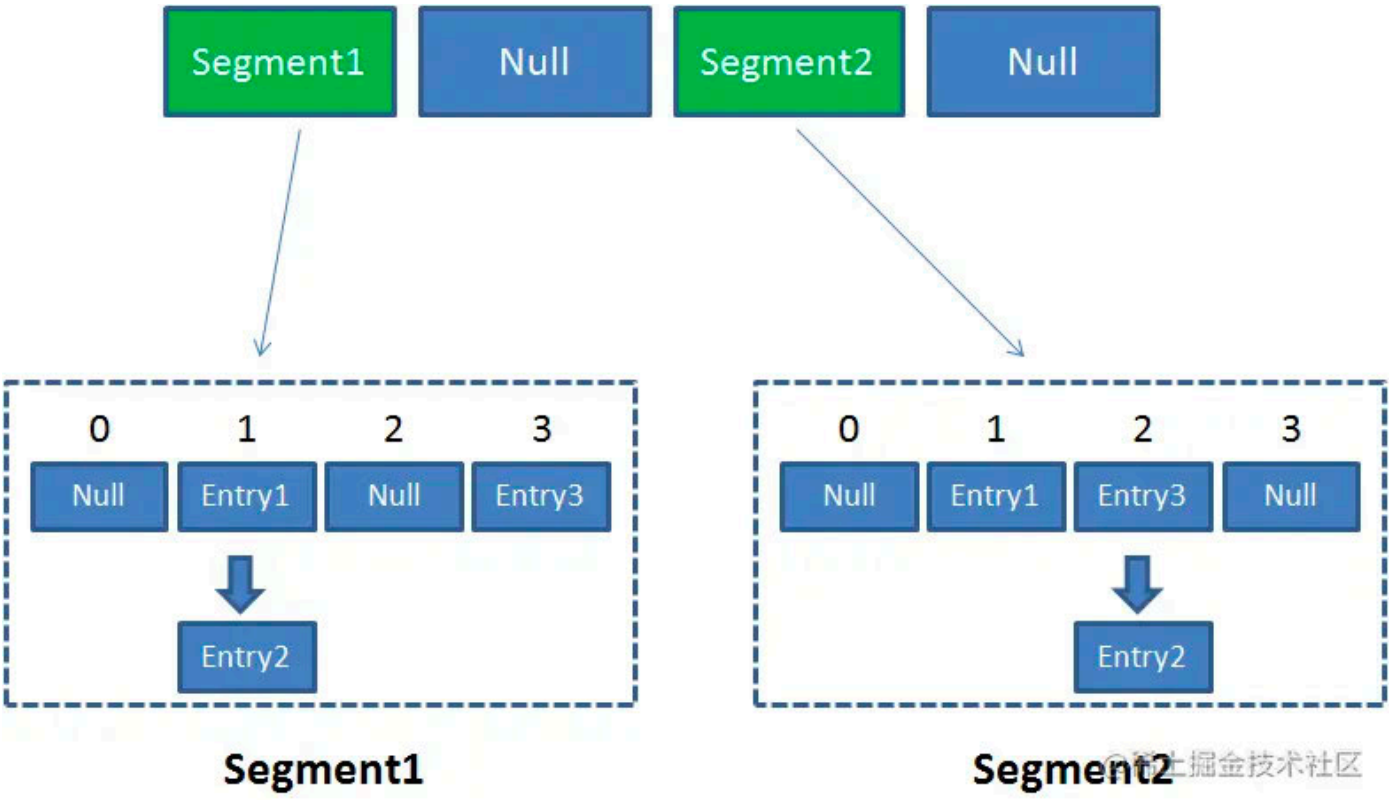
ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组构成的。Segment 是一种可重入的锁 ReentrantLock，HashEntry 则用于存储键值对数据。

一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构， 一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素， 每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

单一的 Segment 结构如下：

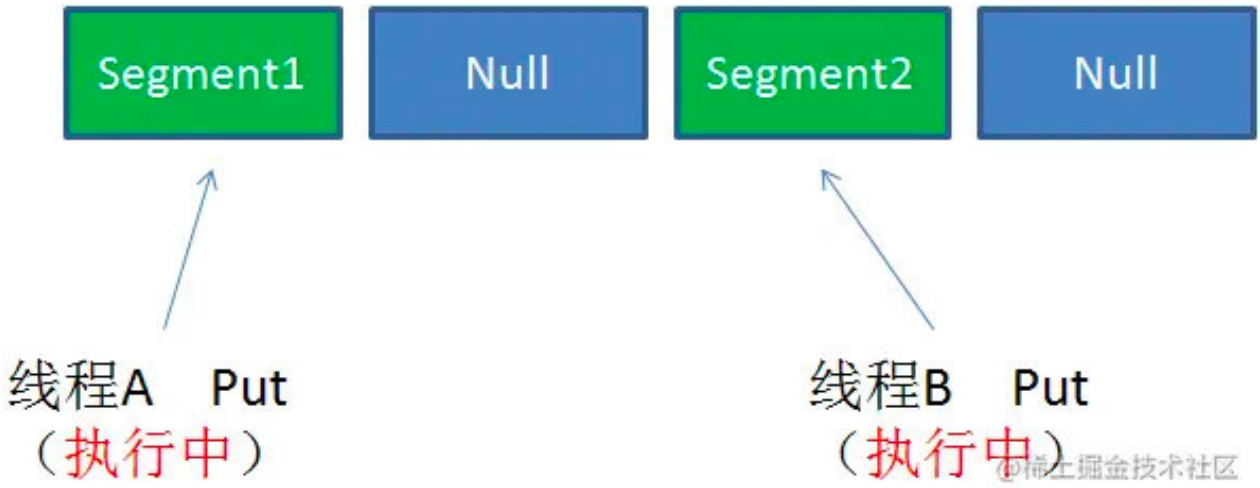


像这样的 Segment 对象，在 ConcurrentHashMap 集合中有多少个呢？有 2 的 N 次方个，共同保存在一个名为 segments 的数组当中。 因此整个 ConcurrentHashMap 的结构如下：

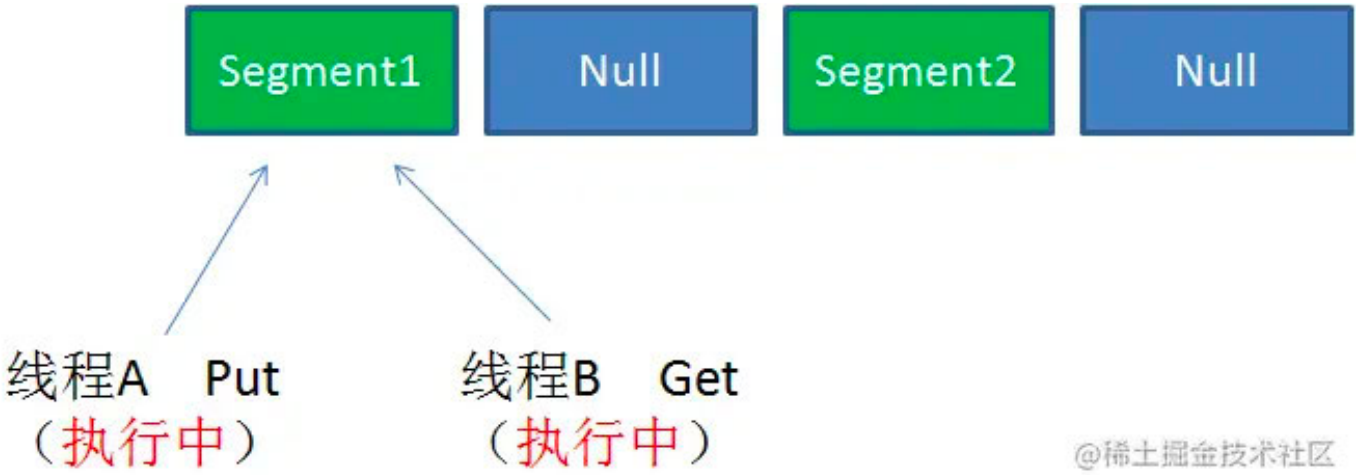


可以说，ConcurrentHashMap 是一个二级哈希表。在一个总的哈希表下面，有若干个子哈希表。

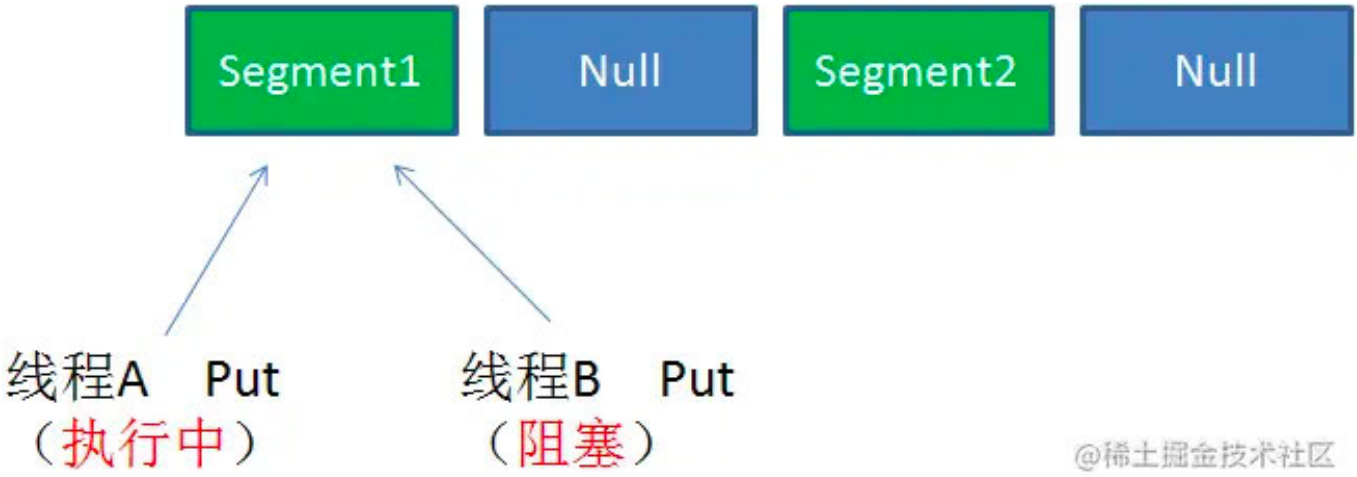
Case1：不同 Segment 的并发写入（可以并发执行）



Case2：同一 Segment 的一写一读（可以并发执行）



Case3：同一 Segment 的并发写入



Segment 的写入是需要上锁的，因此对同一 Segment 的并发写入会被阻塞。

由此可见，ConcurrentHashMap 中每个 Segment 各自持有一把锁。在保证线程安全的同时降低了锁的粒度，让并发操作效率更高。

ConcurrentHashMap 读写过程如下：

get 方法

- 为输入的 Key 做 Hash 运算，得到 hash 值。
- 通过 hash 值，定位到对应的 Segment 对象
- 再次通过 hash 值，定位到 Segment 当中数组的具体位置。

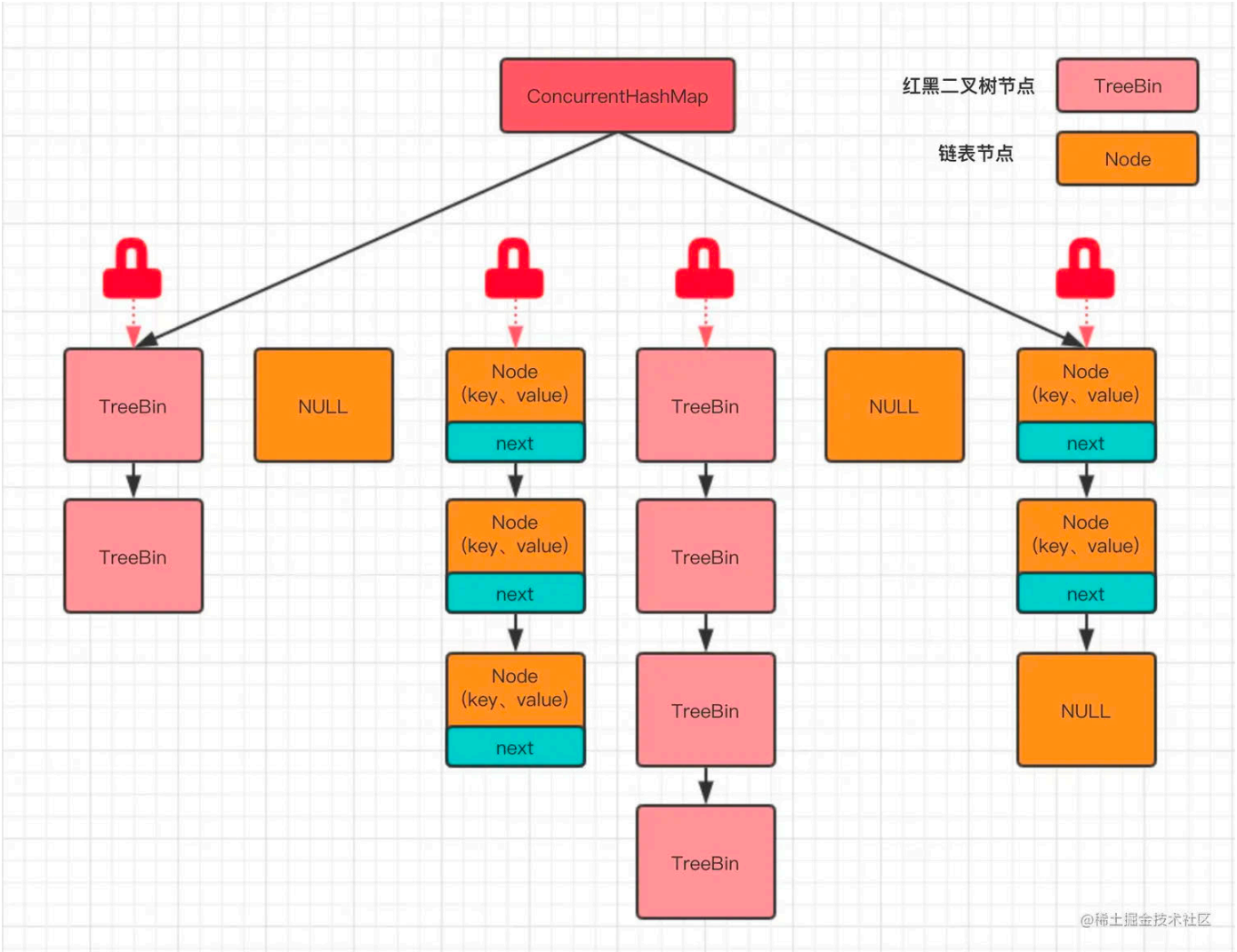
put 方法

- 为输入的 Key 做 Hash 运算，得到 hash 值。
- 通过 hash 值，定位到对应的 Segment 对象
- 获取可重入锁
- 再次通过 hash 值，定位到 Segment 当中数组的具体位置。
- 插入或覆盖 HashEntry 对象。
- 释放锁。

JDK 1.8

而在 JDK 1.8 中，ConcurrentHashMap 主要做了两个优化：

- 同 HashMap 一样，链表也会在长度达到 8 的时候转化为红黑树，这样可以提升大量冲突时候的查询效率；
- 以某个位置的头结点（链表的头结点或红黑树的 root 结点）为锁，配合自旋+ CAS 避免不必要的锁开销，进一步提升并发性能。



相比 JDK1.7 中的 ConcurrentHashMap，JDK1.8 中的 ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS + synchronized 来保证并发安全性，整个容器只分为一个 Segment，即 table 数组。

JDK1.8 中的 ConcurrentHashMap 对节点 Node 类中的共享变量，和 JDK1.7 一样，使用 volatile 关键字，保证多线程操作时，变量的可见性！

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;

    Node(int hash, K key, V val, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.val = val;
        this.next = next;
    }
......
}
```

ConcurrentHashMap的字段

1、**table**，`volatile Node<K,V>[] table`：

装载 Node 的数组，作为 ConcurrentHashMap 的底层容器，采用懒加载的方式，直到第一次插入数据的时候才会进行初始化操作，数组的大小总是为 2 的幂次方，讲 HashMap 的时候讲过。

2、**nextTable**，`volatile Node<K,V>[] nextTable`

扩容时使用，平时为 null，只有在扩容的时候才为非 null

3、**sizeCtl**，`volatile int sizeCtl`

该属性用来控制 table 数组的大小，根据是否初始化和是否正在扩容有几种情况：

- **当值为负数时：** 如果为-1 表示正在初始化，如果为 -N 则表示当前正有 N-1 个线程进行扩容操作；
- **当值为正数时：** 如果当前数组为 null 的话表示 table 在初始化过程中，sizeCtl 表示为需要新建数组的长度；若已经初始化了，表示当前数据容器（table 数组）可用容量，也可以理解成临界值（插入节点数超过了该临界值就需要扩容），具体指为数组的长度 n 乘以 加载因子 loadFactor；
- 当值为 0 时，即数组长度为默认初始值。

4、`sun.misc.Unsafe U`

在 ConcurrentHashMap 的实现中，可以看到用了大量的 `U.compareAndSwapXXXX` 方法去修改 ConcurrentHashMap 的一些属性。

这些方法实际上是利用了 CAS 算法用于保证线程安全性，这是一种乐观策略：假设每一次操作都不会产生冲突，当且仅当冲突发生的时候再去尝试。

我们前面也讲过了，CAS 操作依赖于现代处理器指令集，通过底层的**CMPXCHG**指令实现。`CAS(V,O,N)` 核心思想为：**若当前变量实际值 V 与期望的旧值 O 相同，则表明该变量没被其他线程进行修改，因此可以安全的将新值 N 赋值给变量；若当前变量实际值 V 与期望的旧值 O 不相同，则表明该变量已经被其他线程做了处理，此时将新值 N 赋给变量操作就是不安全的，在进行重试。**

在并发容器中，CAS 是通过`sun.misc.Unsafe`类实现的，该类提供了一些可以直接操控内存和线程的底层操作，可以理解为 Java 中的“指针”。该成员变量的获取是在静态代码块中：

```
static {
    try {
        U = sun.misc.Unsafe.getUnsafe();
    .......
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

ConcurrentHashMap的内部类

1、Node

Node 类实现了 Map.Entry 接口，主要存放 key-value 对，并且具有 next 域

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    ......
}
```

另外可以看出很多属性都是用 volatile 关键字修饰的，也是为了保证内存可见性。

2、TreeNode

树节点，继承于承载数据的 Node 类。红黑树的操作是针对 TreeBin 类的，从该类的注释也可以看出，TreeBin 是对 TreeNode 的再一次封装，下面会提到。

```
**
 * Nodes for use in TreeBins
 */
static final class TreeNode<K,V> extends Node<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;
    ......
}
```

3、TreeBin

这个类并不负责用户的 key、value 信息，而是封装了很多 TreeNode 节点。实际的 ConcurrentHashMap “数组”中，存放的都是 TreeBin 对象，而不是 TreeNode 对象。

```
static final class TreeBin<K,V> extends Node<K,V> {
        TreeNode<K,V> root;
        volatile TreeNode<K,V> first;
        volatile Thread waiter;
        volatile int lockState;
        // values for lockState
        static final int WRITER = 1; // set while holding write lock
        static final int WAITER = 2; // set when waiting for write lock
        static final int READER = 4; // increment value for setting read lock
    ......
}
```

4、ForwardingNode

在扩容时会出现的特殊节点，其 key、value、hash 全部为 null。并拥有 nextTable 引用的新 table 数组。

```
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
   .....
}
```

ConcurrentHashMap的CAS

ConcurrentHashMap 会大量使用 CAS 来修改它的属性和进行一些操作。因此，在理解 ConcurrentHashMap 的方法前，我们需要了解几个常用的利用 CAS 算法来保障线程安全的操作。

1、tabAt

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}
```

该方法用来获取 table 数组中索引为 i 的 Node 元素。

2、casTabAt

```
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

利用 CAS 操作设置 table 数组中索引为 i 的元素

3、setTabAt

```
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

该方法用来设置 table 数组中索引为 i 的元素

ConcurrentHashMap的方法

构造方法

ConcurrentHashMap 一共提供了以下 5 个构造方法：

```
// 1. 构造一个空的map，即table数组还未初始化，初始化放在第一次插入数据时，默认大小为16
ConcurrentHashMap()
// 2. 给定map的大小
ConcurrentHashMap(int initialCapacity)
// 3. 给定一个map
ConcurrentHashMap(Map<? extends K, ? extends V> m)
// 4. 给定map的大小以及加载因子
ConcurrentHashMap(int initialCapacity, float loadFactor)
// 5. 给定map大小，加载因子以及并发度（预计同时操作数据的线程）
ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel)
```

差别请看注释，我们来看看第 2 种构造方法，源码如下：

```
public ConcurrentHashMap(int initialCapacity) {
  //1. 小于0直接抛异常
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
  //2. 判断是否超过了允许的最大值，超过了话则取最大值，否则再对该值进一步处理
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
  //3. 赋值给sizeCtl
    this.sizeCtl = cap;
}
```

这段代码的逻辑请看注释，很容易理解，如果小于 0 就直接抛异常，如果指定值大于所允许的最大值就取最大值，否则再对指定值做进一步处理。最后将 cap 赋值给 sizeCtl。

当调用构造方法之后，sizeCtl 的大小就代表了 ConcurrentHashMap 的大小，即 table 数组的长度。

tableSizeFor 做了哪些事情呢？源码如下：

```
/**
 * Returns a power of two table size for the given desired capacity.
 * See Hackers Delight, sec 3.2
 */
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

注释写的很清楚，该方法会将构造方法指定的大小转换成一个 2 的幂次方数，也就是说 ConcurrentHashMap 的大小一定是 2 的幂次方，比如，当指定大小为 18 时，为了满足 2 的幂次方特性，实际上 ConcurrentHashMap 的大小为 2 的 5 次方（32）。

另外，需要注意的是，**调用构造方法时并初始化 table 数组，而只算出了 table 数组的长度，当第一次向 ConcurrentHashMap 插入数据时才会真正的完成初始化，并创建 table 数组。**

initTable 方法

直接上源码：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
      // 1. 保证只有一个线程正在进行初始化操作
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
          // 2. 得出数组的大小
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
          // 3. 这里才真正的初始化数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
          // 4. 计算数组中可用的大小：实际大小n*0.75（加载因子）
                    sc = n - (n >>> 2);
```

```
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

代码的逻辑请见注释。

可能存在这样一种情况，多个线程同时进入到这个方法，为了保证能够正确地初始化，第 1 步会先通过 if 进行判断，如果当前已经有一个线程正在初始化，这时候其他线程会调用 `Thread.yield()` 让出 CPU 时间片。

正在进行初始化的线程会调用 `U.compareAndSwapInt` 方法将 sizeCtl 改为 -1，即正在初始化的状态。

另外还需要注意，在第四步中会进一步计算数组中可用的大小，即数组的实际大小 n 乘以加载因子 0.75，0.75 就是四分之三，这里`n - (n >>> 2)`刚好是`n-(1/4)n=(3/4)n`，挺有意思的吧?

如果选择是无参的构造方法，这里在 new Node 数组的时候会使用默认大小`DEFAULT_CAPACITY`（16），然后乘以加载因子 0.75，结果为 12，也就是说数组当前的可用大小为 12。

put 方法

调用 put 方法时会调用 putVal 方法，源码如下：

```
/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
  //1. 计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
    //2. 如果当前table还没有初始化先调用initTable方法将tab进行初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
    //3. tab中索引为i的位置的元素为null，则直接使用CAS将值插入即可
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
    //4. 当前正在扩容
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) {
```

```
                if (tabAt(tab, i) == f) {
            //5. 当前为链表，在链表中插入新的键值对
                    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, null);
                                break;
                            }
                        }
                    }
            // 6.当前为红黑树，将新的键值对插入到红黑树中
                    else if (f instanceof TreeBin) {
                        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;
                        }
                    }
                }
            }
        // 7.插入完键值对后再根据实际大小看是否需要转换成红黑树
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
  //8.对当前容量大小进行检查，如果超过了临界值（实际大小*加载因子）就需要扩容
    addCount(1L, binCount);
    return null;
}
```

ConcurrentHashMap 是一个哈希桶数组，如果不出现哈希冲突的时候，每个元素均匀的分布在哈希桶数组中。当出现哈希冲突的时候，采用**拉链法的解决方案**，将 hash 值相同的节点转换成链表的形式，另外，在 JDK 1.8 版本中，为了防止拉链过长，当链表的长度大于 8 的时候会将链表转换成红黑树。

确定好数组的索引 i 后，可以调用 `tabAt()` 方法获取该位置上的元素，如果当前 Node 为 null 的话，可以直接用 casTabAt 方法将新值插入。

拉链法、确定索引 i 的知识在学习 HashMap 的时候就讲过，相信大家都还没有忘。

如果当前节点不为 null，且该节点为特殊节点（forwardingNode），就说明当前 concurrentHashMap 正在进行扩容操作。怎么确定当前这个 Node 是特殊节点呢?

通过判断该节点的 hash 值是不是等于 -1（MOVED）：

```
static final int MOVED     = -1; // hash for forwarding nodes
```

当 `table[i]` 不为 null 并且不是 forwardingNode 时，以及当前 Node 的 hash 值大于 `0（fh >= 0）` 时，说明当前节点为链表的头节点，那么向 ConcurrentHashMap 插入新值就是向这个链表插入新值。通过 `synchronized (f)` 的方式进行加锁以实现线程安全。

往链表中插入节点的部分代码如下：

```
if (fh >= 0) {
    binCount = 1;
    for (Node<K,V> e = f;; ++binCount) {
        K ek;
    // 找到hash值相同的key,覆盖旧值即可
        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) {
      //如果到链表末尾仍未找到，则直接将新值插入到链表末尾即可
            pred.next = new Node<K,V>(hash, key,
                                      value, null);
            break;
        }
    }
}
```

这部分代码很好理解，就两种情况：

1. 如果在链表中找到了与待插入的 key 相同的节点，就直接覆盖；
2. 如果找到链表的末尾都还没找到的话，直接将待插入的键值对追加到链表的末尾。

当链表长度超过 8（默认值）时，链表就转换为红黑树，利用红黑树快速增删改查的特点可以提高 ConcurrentHashMap 的性能：

```
if (f instanceof TreeBin) {
    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;
    }
}
```

这段代码很简单，调用 putTreeVal 方法向红黑树插入新节点，同样的逻辑，**如果在红黑树中存在 Key 相同（hash 值相等并且 equals 方法判断为 true）的节点，就覆盖旧值，否则向红黑树追加新节点。**

当完成数据新节点插入后，会进一步对当前链表大小进行调整：

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

至此，put 方法就分析完了，我们来做个总结：

1. 对每一个放入的值，先用 spread 方法对 key 的 hashcode 进行 hash 计算，由此来确定这个值在 table 中的位置；
2. 如果当前 table 数组还未初始化，进行初始化操作；
3. 如果这个位置是 null，那么使用 CAS 操作直接放入；
4. 如果这个位置存在节点，说明发生了 hash 碰撞，先判断这个节点的类型，如果该节点 `==MOVED` 的话，说明正在进行扩容；
5. 如果是链表节点（`fh>0`），先获取头节点，再依次向后遍历确定这个新加入节点的位置。如果遇到 key 相同的节点，直接覆盖。否则在链表尾插入；
6. 如果这个节点的类型是 TreeBin，直接调用红黑树的插入方法插入新的节点；
7. 插入完节点之后再次检查链表的长度，如果长度大于 8，就把这个链表转换成红黑树；
8. 对当前容量大小进行检查，如果超过了临界值（实际大小*加载因子）就需要扩容。

get 方法

get 方法的源码如下：

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
  // 1. 重hash
```

```
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 2. table[i]桶节点的key与查找的key相同，则直接返回
    if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
    // 3. 当前节点hash小于0说明为树节点，在红黑树中查找即可
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
    //4. 从链表中查找，查找到则返回该节点的value，否则就返回null即可
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

- 哈希: 对传入的键的哈希值进行散列，这有助于减少哈希冲突的可能性。使用 spread 方法可以保证不同的键更均匀地分布在桶数组中。
- 直接查找: 查找的第一步是检查键的哈希值是否位于表的正确位置。如果在该桶的第一个元素中找到了键，则直接返回该元素的值。这里使用了 == 操作符和 equals 方法来比较键，这有助于处理可能的 null 值和确保正确的相等性比较。
- 红黑树查找: 如果第一个节点的哈希值小于0，那么这个桶的数据结构是红黑树（Java 8 引入了树化结构来改进链表在哈希冲突时的性能）。在这种情况下，使用 find 方法在红黑树中查找键。
- 链表查找: 如果前两个条件都不满足，那么代码将遍历该桶中的链表。如果在链表中找到了具有相同哈希值和键的元素，则返回其值。如果遍历完整个链表都未找到，则返回 null。

transfer 方法

当 ConcurrentHashMap 容量不足的时候，需要对 table 进行扩容。这个方法的基本思想跟 HashMap 很像，但由于支持并发扩容，所以要复杂一些。transfer 方法源码如下：

```
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; // subdivide range
  //1. 新建Node数组，容量为之前的两倍
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
```

```
            return;
        }
        nextTable = nextTab;
        transferIndex = n;
    }
    int nextn = nextTab.length;
//2. 新建forwardingNode引用，在之后会用到
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        // 3. 确定遍历中的索引i
    while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ?
                                   nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
    //4.将原数组中的元素复制到新数组中去
    //4.5 for循环退出，扩容结束修改sizeCtl属性
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            if (finishing) {
                nextTable = null;
                table = nextTab;
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
    //4.1 当前数组中第i个元素为null，用CAS设置成特殊节点forwardingNode(可以理解成占位符)
        else if ((f = tabAt(tab, i)) == null)
```

```
            advance = casTabAt(tab, i, null, fwd);
    //4.2 如果遍历到ForwardingNode节点  说明这个点已经被处理过了 直接跳过   这里是控制并发扩容的核心
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    if (fh >= 0) {
            //4.3 处理当前节点为链表的头结点的情况，根据最高位为1还是为0(最高位指数组长度位)，将原链表拆分为两个链表，分别放到新数组的i位置和i+n位置。这里还通过巧妙的处理措施，使得原链表中的一部分能直接平移到新链表(即lastRun及其后面跟着的一串节点)，剩下部分才需要通过new方式克隆移动到新链表中（采用头插法）。
                        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;
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            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); //可以看到是逆序插入新节点的（头插）
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                       //在nextTable的i位置上插入一个链表
                         setTabAt(nextTab, i, ln);
                         //在nextTable的i+n的位置上插入另一个链表
                         setTabAt(nextTab, i + n, hn);
                         //在table的i位置上插入forwardNode节点  表示已经处理过该节点
                         setTabAt(tab, i, fwd);
                         //设置advance为true 返回到上面的while循环中 就可以执行i--操作
                         advance = true;
                    }
        //4.4 处理当前节点是TreeBin时的情况，操作和上面的类似
                    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;
                    }
                }
            }
        }
    }
}
```

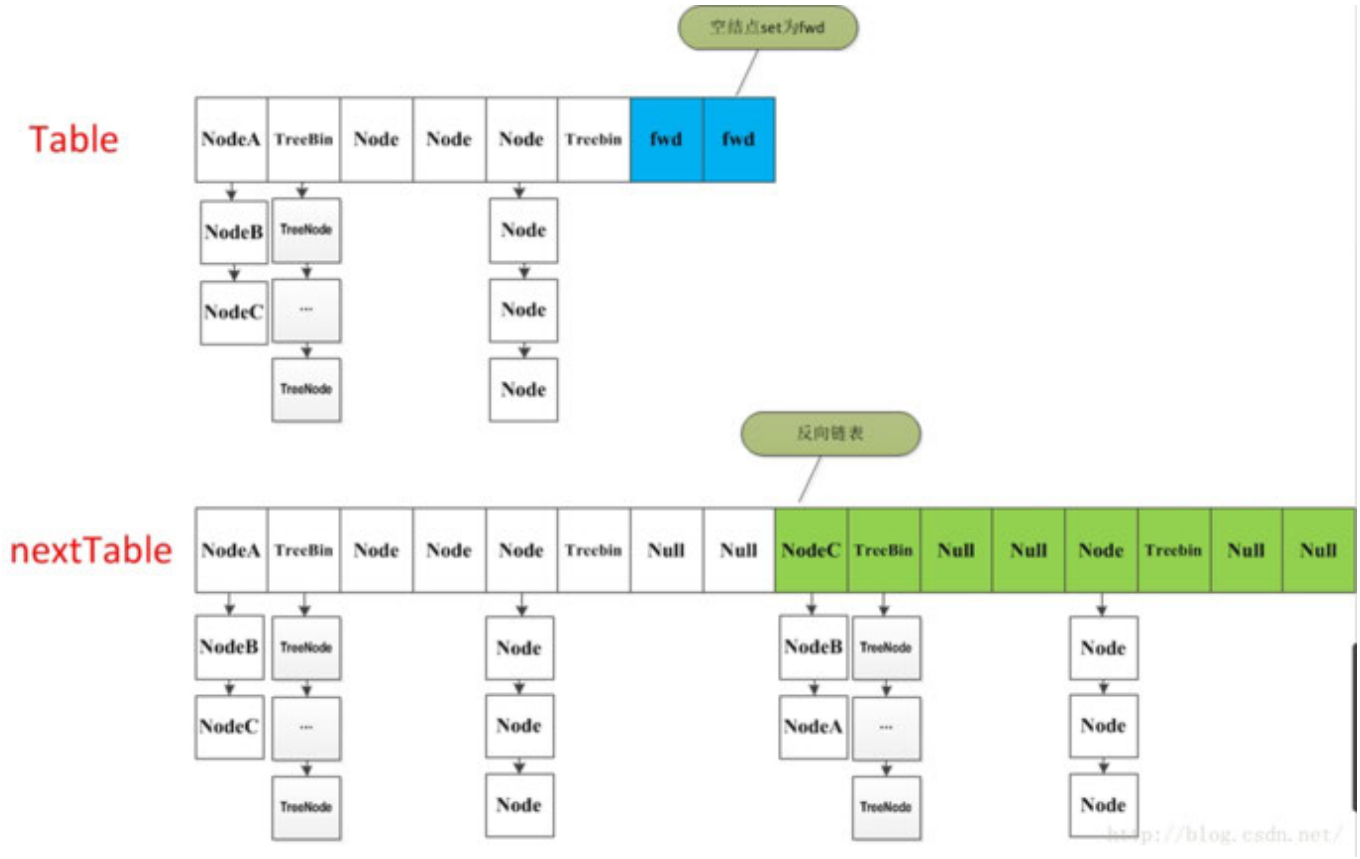
代码逻辑请看注释，整个扩容操作分为**两个部分**：

第一部分是构建一个 nextTable，它的容量是原来的两倍，这个操作是单线程完成的。

第二个部分是将原来 table 中的元素复制到 nextTable 中，主要是遍历复制的过程。
得到当前遍历的数组位置 i，然后利用 tabAt 方法获得 i 位置的元素：

1. 如果这个位置为空，就在原 table 中的 i 位置放入 forwardNode 节点，这个也是触发并发扩容的关键；
2. 如果这个位置是 Node 节点（`fh>=0`），并且是链表的头节点，就把这个链表分裂成两个链表，把它们分别放在 nextTable 的 i 和 i+n 的位置上；

3. 如果这个位置是 TreeBin 节点（`fh<0`），也做一个反序处理，并且判断是否需要 untreefi，把处理的结果分别放在 nextTable 的 i 和 i+n 的位置上；
4. 遍历所有的节点，就完成复制工作，这时让 nextTable 作为新的 table，并且更新 sizeCtl 为新容量的 0.75 倍，完成扩容。



size 相关的方法

对于 ConcurrentHashMap 来说，这个 table 里到底装了多少东西是不确定的，因为**不可能在调用 `size()` 方法的时候“stop the world”让其他线程都停下来去统计**，对于这个不确定的 size，ConcurrentHashMap 仍然花费了大量的力气。

为了统计元素的个数，ConcurrentHashMap 定义了一些变量和一个内部类。

```
/**
 * A padded cell for distributing counts.  Adapted from LongAdder
 * and Striped64.  See their internal docs for explanation.
 */
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}

/*****************************************/

/**
 * 实际上保存的是HashMap中的元素个数  利用CAS锁进行更新
 但它并不用返回当前HashMap的元素个数
```

```
 */
private transient volatile long baseCount;
/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;

/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;
```

再来看如何统计的源码：

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
 /**
 * Returns the number of mappings. This method should be used
 * instead of {@link #size} because a ConcurrentHashMap may
 * contain more mappings than can be represented as an int. The
 * value returned is an estimate; the actual count may differ if
 * there are concurrent insertions or removals.
 *
 * @return the number of mappings
 * @since 1.8
 */
public long mappingCount() {
    long n = sumCount();
    return (n < 0L) ? 0L : n; // ignore transient negative values
}

 final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;//所有counter的值求和
        }
    }
    return sum;
}
```

size 方法返回 Map 中的元素数量，但结果被限制在 Integer.MAX_VALUE 内。如果计算的大小超过这个值，则返回 Integer.MAX_VALUE。如果计算的大小小于0，则返回0。

mappingCount 方法也返回 Map 中的元素数量，但允许返回一个 long 值，因此可以表示大于 Integer.MAX_VALUE 的数量。与 `size()` 方法类似，该方法也会忽略负值，返回0。

sumCount 方法计算 Map 的实际大小。ConcurrentHashMap 使用一个基础计数 baseCount 和一个 CounterCell 数组 counterCells 来跟踪大小。这种结构有助于减少多线程环境中的争用，因为不同的线程可能会更新不同的 CounterCell。

在计算总和时，`sumCount()` 方法将 baseCount 与 counterCells 数组中的所有非空单元的值相加。

在 put 方法结尾处调用了 addCount 方法，把当前 ConcurrentHashMap 的元素个数 +1，这个方法一共做了两件事，更新 baseCount 的值，检测是否进行扩容。

```
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    //利用CAS方法更新baseCount的值
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    //如果check值大于等于0 则需要检验是否需要进行扩容操作
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            //
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                 //如果已经有其他线程在执行扩容操作
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            //当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
```

```
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

ConcurrentHashMap示例

假设我们想要构建一个线程安全的高并发统计用户访问次数的功能。在这里，ConcurrentHashMap是一个很好的选择，因为它提供了高并发性能。

```
import java.util.concurrent.ConcurrentHashMap;

public class UserVisitCounter {

    private final ConcurrentHashMap<String, Integer> visitCountMap;

    public UserVisitCounter() {
        this.visitCountMap = new ConcurrentHashMap<>();
    }

    // 用户访问时调用的方法
    public void userVisited(String userId) {
        visitCountMap.compute(userId, (key, value) -> value == null ? 1 : value + 1);
    }

    // 获取用户的访问次数
    public int getVisitCount(String userId) {
        return visitCountMap.getOrDefault(userId, 0);
    }

    public static void main(String[] args) {
        UserVisitCounter counter = new UserVisitCounter();

        // 模拟用户访问
        counter.userVisited("user1");
        counter.userVisited("user1");
        counter.userVisited("user2");

        System.out.println("User1 visit count: " + counter.getVisitCount("user1")); //
输出: User1 visit count: 2
        System.out.println("User2 visit count: " + counter.getVisitCount("user2")); //
输出: User2 visit count: 1
    }
}
```

在上述示例中：

- 我们使用了ConcurrentHashMap来存储用户的访问次数。
- 当用户访问时，我们通过userVisited方法更新访问次数。
- 使用ConcurrentHashMap的compute方法可以确保原子地更新用户的访问次数。
- 可以通过getVisitCount方法检索任何用户的访问次数。

ConcurrentHashMap使我们能够无需担心并发问题就能构建这样一个高效的统计系统。

小结

ConcurrentHashMap 是线程安全的，支持完全并发的读取，并且有很多线程可以同时执行写入。在早期版本（例如 JDK 1.7）中，ConcurrentHashMap 使用分段锁技术。整个哈希表被分成一些段（Segment），每个段独立加锁。这样，在不同段上的操作可以并发进行。从 JDK 1.8 开始，ConcurrentHashMap 的内部实现有了很大的变化。它放弃了分段锁技术，转而采用了更先进的并发控制策略，如 CAS 操作和红黑树等，进一步提高了并发性能。

由于并发性质，ConcurrentHashMap 的大小计算可能不是精确的，但通常足够接近真实值。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency，部分内容来自于这篇初念初恋-ConcurrentHashMap，图片画的特别漂亮。

第二十一节：非阻塞队列 ConcurrentLinkedQueue

ConcurrentLinkedQueue 是 `java.util.concurrent`（JUC） 包下的一个线程安全的队列实现。基于非阻塞算法（Michael-Scott 非阻塞算法的一种变体），这意味着 ConcurrentLinkedQueue 不再使用传统的锁机制来保护数据安全，而是依靠底层原子的操作（如 CAS）来实现。

Michael-Scott 由 Maged M. Michael 和 Michael L. Scott 在 1996 年提出，在这种算法中，一个线程的失败或挂起不会导致其他线程也失败或挂起。

好，接下来一起来看一下 ConcurrentLinkedQueue 的源码实现。

节点类Node

先从它的节点类 Node 看起，好明白 ConcurrentLinkedQueue 的底层数据结构。Node 类的源码如下：

```
private static class Node<E> {
        volatile E item;
        volatile Node<E> next;
    .......
}
```

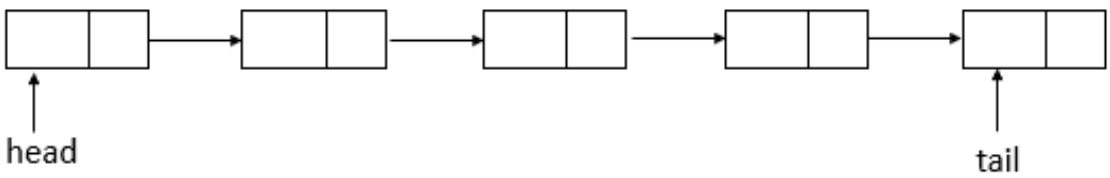
Node 节点包含了两个字段：

- 一个是数据域 item
- 另一个是 next 指针，用于指向下一个节点从而构成链式队列。

两个字段都是用 volatile 修饰的，以保证内存的可见性。

另外，ConcurrentLinkedQueue 还有这样两个成员变量：

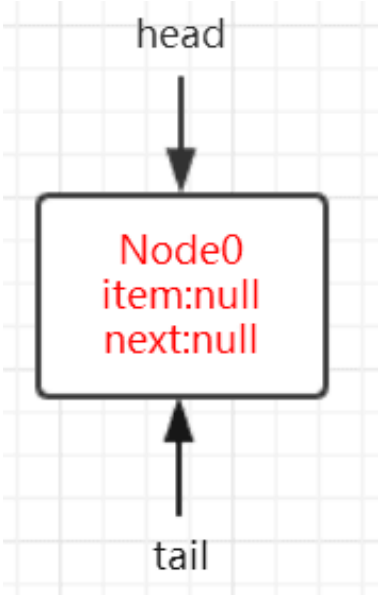
```
private transient volatile Node<E> head;
private transient volatile Node<E> tail;
```



说明 ConcurrentLinkedQueue 通过持有头尾两个引用来进行队列管理。当我们调用无参构造方法时，其源码如下：

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

head 和 tail 会指向同一个节点，此时 ConcurrentLinkedQueue 的状态如下图所示：



head 和 tail 指向同一个节点 Node0，该节点的 item 字段为 null，next 字段也为 null。

在队列进行出队入队的时候，免不了要对节点进行操作，在多线程环境下就很容易出现线程安全问题。ConcurrentLinkedQueue 选择使用 CAS 来保证线程安全：

```
//更改Node中的数据域item
boolean casItem(E cmp, E val) {
    return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
}
//更改Node中的指针域next
void lazySetNext(Node<E> val) {
    UNSAFE.putOrderedObject(this, nextOffset, val);
}
//更改Node中的指针域next
boolean casNext(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
}
```

可以看出，这些方法实际上调用的是 UNSAFE 的方法：

```
// Unsafe mechanics

private static final sun.misc.Unsafe UNSAFE;
private static final long itemOffset;
private static final long nextOffset;

static {
    try {
        UNSAFE = sun.misc.Unsafe.getUnsafe();
        Class<?> k = Node.class;
        itemOffset = UNSAFE.objectFieldOffset
            (k.getDeclaredField( name: "item"));
        nextOffset = UNSAFE.objectFieldOffset
            (k.getDeclaredField( name: "next"));
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

`sun.misc.Unsafe` 是 Java 内部的一个类，它提供了一组可以直接访问底层资源和操作内存的方法。这个类的功能非常强大，因为它允许程序绕过 Java 的访问控制和安全检查，直接执行底层操作。

Unsafe 允许分配、释放和访问本机内存，就像使用 C 语言中的 malloc 和 free 一样。我们在讲 CAS 的时候有详细讲过，相信大家都还有印象。

offer方法

ConcurrentLinkedQueue 是一种先进先出（FIFO，First-In-First-Out）的队列，offer 方法用于在队列尾部插入一个元素。如果成功添加元素，则返回 true。下面是这个方法的一般定义：

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

来看这么一段代码：

```
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
queue.offer(1);
queue.offer(2);
```

我们创建一个 ConcurrentLinkedQueue 对象 queue，先 offer 1，再 offer 2。其中 offer 的源码如下：

```
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);

    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            // p is last node
            if (p.casNext(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
                    casTail(t, newNode);  // Failure is OK.
                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;
    }
}
```

1、参数检查：`checkNotNull(e)` 确保传递的元素不是 null。

2、新节点创建：`final Node<E> newNode = new Node<E>(e)` 创建一个新的节点来保存要添加的元素。

3、尾部节点循环：该循环用于找到队列的尾部节点，并将新节点安全地链接到尾部。

- a. 读取下一个节点：`Node<E> q = p.next` 读取当前节点的下一个节点。
- b. 尾部节点检查：如果 q 是 null，这意味着当前节点 p 是尾部节点。
- c. CAS操作添加新节点：`p.casNext(null, newNode)` 使用 CAS 操作将新节点链接到当前的尾部节点。如果成功，则更新尾部引用，并返回 true。
- d. 双跳尾部更新：`casTail(t, newNode)` 有时尝试更新尾部引用，使其指向新的尾部节点。这有助于其他线程更快地找到尾部。
- e. 掉出列表检查：如果 p == q，这意味着当前线程从列表上掉了下来。此时，代码尝试跳转到头部或新的尾部。
- f. 进一步检查：否则，代码进行进一步的检查并更新 p 的值，可能是当前的尾部或下一个节点。

我把代码注释去掉，并标上行号。

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

```
1.    checkNotNull(e);
2.    final Node<E> newNode = new Node<E>(e);
3.    for (Node<E> t = tail, p = t;;) {
4.        Node<E> q = p.next;
5.        if (q == null) {
6.            // p is last node
7.            if (p.casNext(null, newNode)) {
8.                if (p != t)
9.                    casTail(t, newNode);
10.                return true;
            }
        }
11.        else if (p == q)
12.            p = (t != (t = tail)) ? t : head;
           else
13.            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

单线程执行角度分析

我们再从**单线程的角度**分析 offer 1 的过程。

第 1 行代码检查元素 e 是否为 null，为 null 就直接抛出空指针异常。

第 2 行代码将 e 包装成一个 Node 对象。

第 3 行为 for 循环，只有初始化条件没有循环结束条件，这很符合 CAS 的“套路”，在循环体内，如果 CAS 操作成功会直接 return 返回，如果 CAS 操作失败就在 for 循环中不断重试直至成功。这里实例变量 t 被初始化为 tail，p 被初始化为 t 即 tail。

p 被认为是队列真正的尾节点，tail 不一定是真正的尾节点，因为在 ConcurrentLinkedQueue 中 tail 延迟更新的。

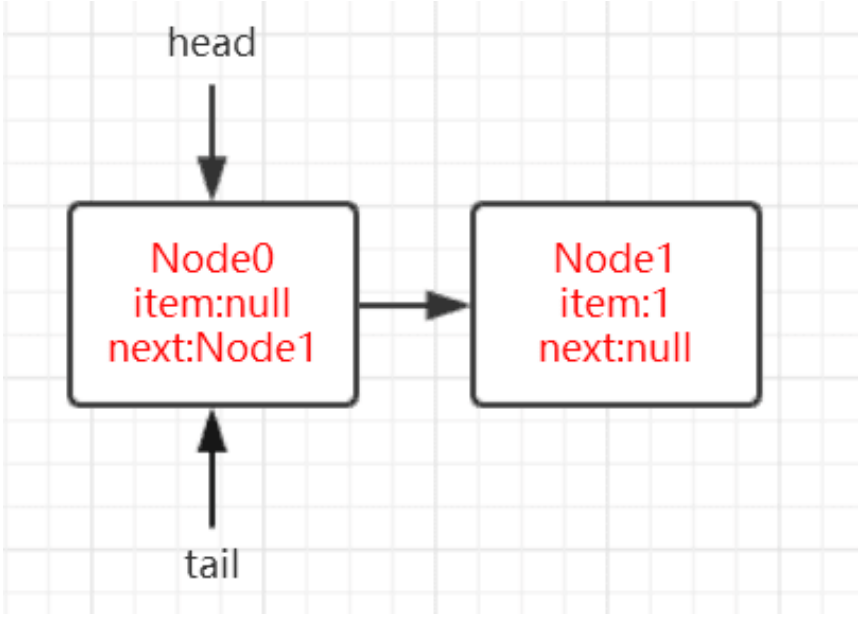
代码走到第 3 行的时候，t 和 p 分别指向初始化时创建的 item（null），next 字段也为 null，即 Node0。

第 4 行变量 q 被赋值为 null。

第 5 行 if 判断结果为 true。

第 7 行使用 casNext 将插入的 Node 设置为当前队列尾节点 p 的 next 节点，如果 CAS 操作失败，此次循环结束，下次循环进行重试。

CAS 操作成功走到第 8 行，此时 p==t，if 判断为 false，直接 return true 返回。如果成功插入 1 的话，此时 ConcurrentLinkedQueue 的状态如下图所示：



此时队列的尾节点应该是 Node1，而 tail 指向的节点依然是 Node0，因此可以说明 tail 是延迟更新的。

那么我们继续看 offer 2，很显然此时第 4 行 q 指向的节点不为 null 了，而是指向 Node1，第 5 行 if 判断为 false，第 11 行 if 判断为 false，代码会走到第 13 行。

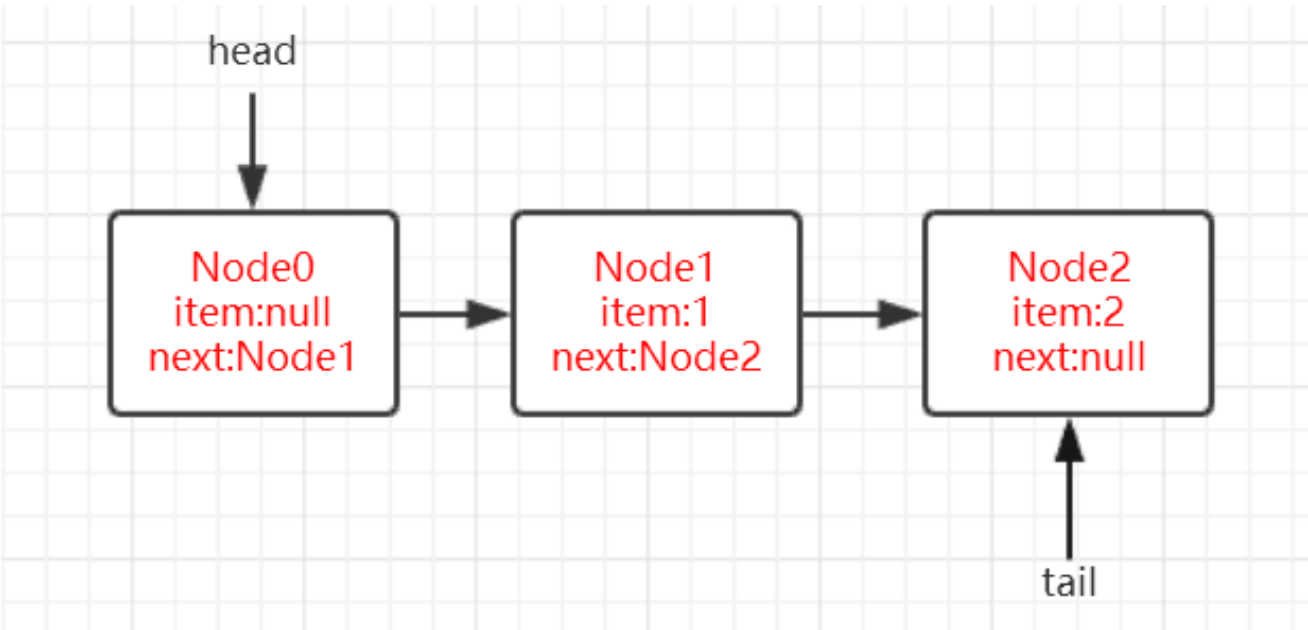
好了，**再插入节点的时候我们来问自己这样一个问题：tail 并不是真正的尾节点，那么在插入节点的时候，我们是不是应该先找到当前的尾节点才能插入？**

第 13 行代码就是**找出队列真正的尾节点**。

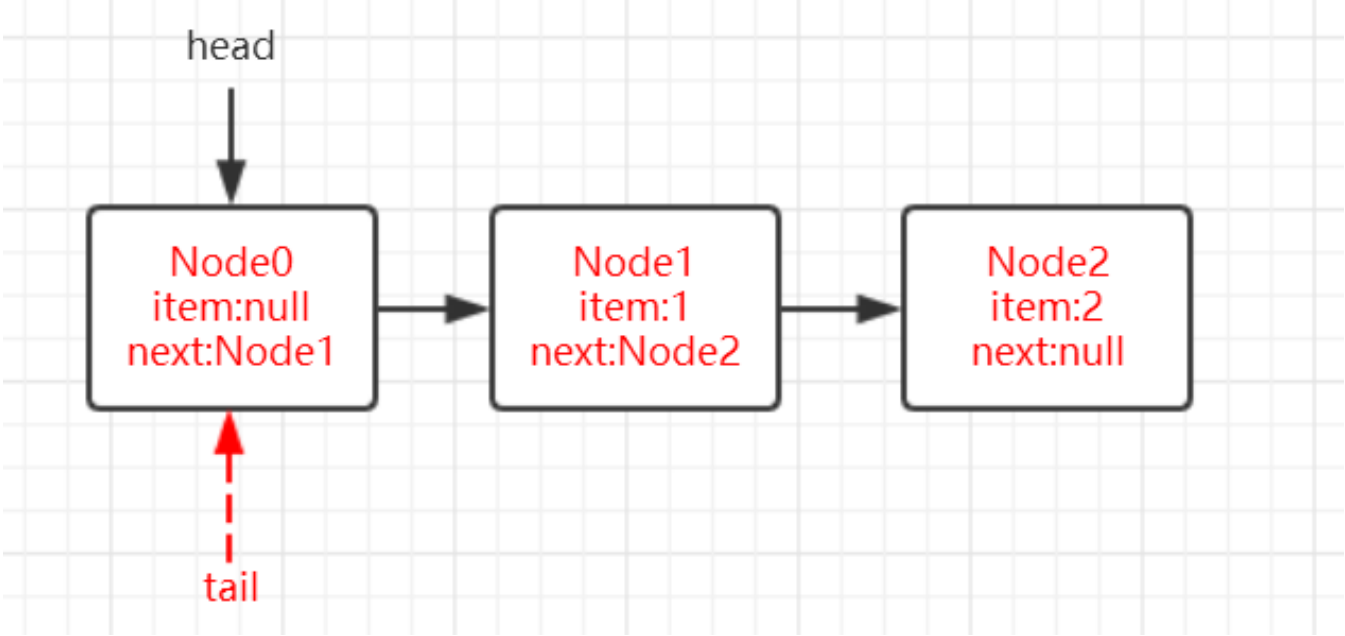
```
p = (p != t && t != (t = tail)) ? t : q;
```

这段代码在**单线程环境**执行时，由于 p==t，此时 p 会被赋值为 q，而 q 等于`Node<E> q = p.next`，即 Node1。

在第一次循环中，p 指向了队列真正的尾节点 Node1，那么在下一次循环中，第 4 行 q 指向的节点为 null，那么第 5 行 if 判断则为 true，第 7 行依然通过 casNext 设置 p 节点的 next 为当前新增的 Node，接下来走到第 8 行，这个时候 p!=t，第 8 行 if 判断为 true，会通过`casTail(t, newNode)`将当前节点 Node 设置为队列的尾节点，此时的队列的状态示意图如下图所示：



tail 指向的节点由 Node0 变为 Node2，这里的 casTail 是不需要重试的，原因是，offer 主要是通过 p 的 next 节点 q（`Node<E> q = p.next`）决定后面的逻辑走向，casTail 失败时状态示意图如下：



如果 casTail 更新 tail 失败，即 tail 还是指向 Node0 节点，无非就是多循环几次，通过第 13 行代码定位到尾节点。

通过单线程执行角度的分析，我们可以了解到 offer 的执行逻辑为：

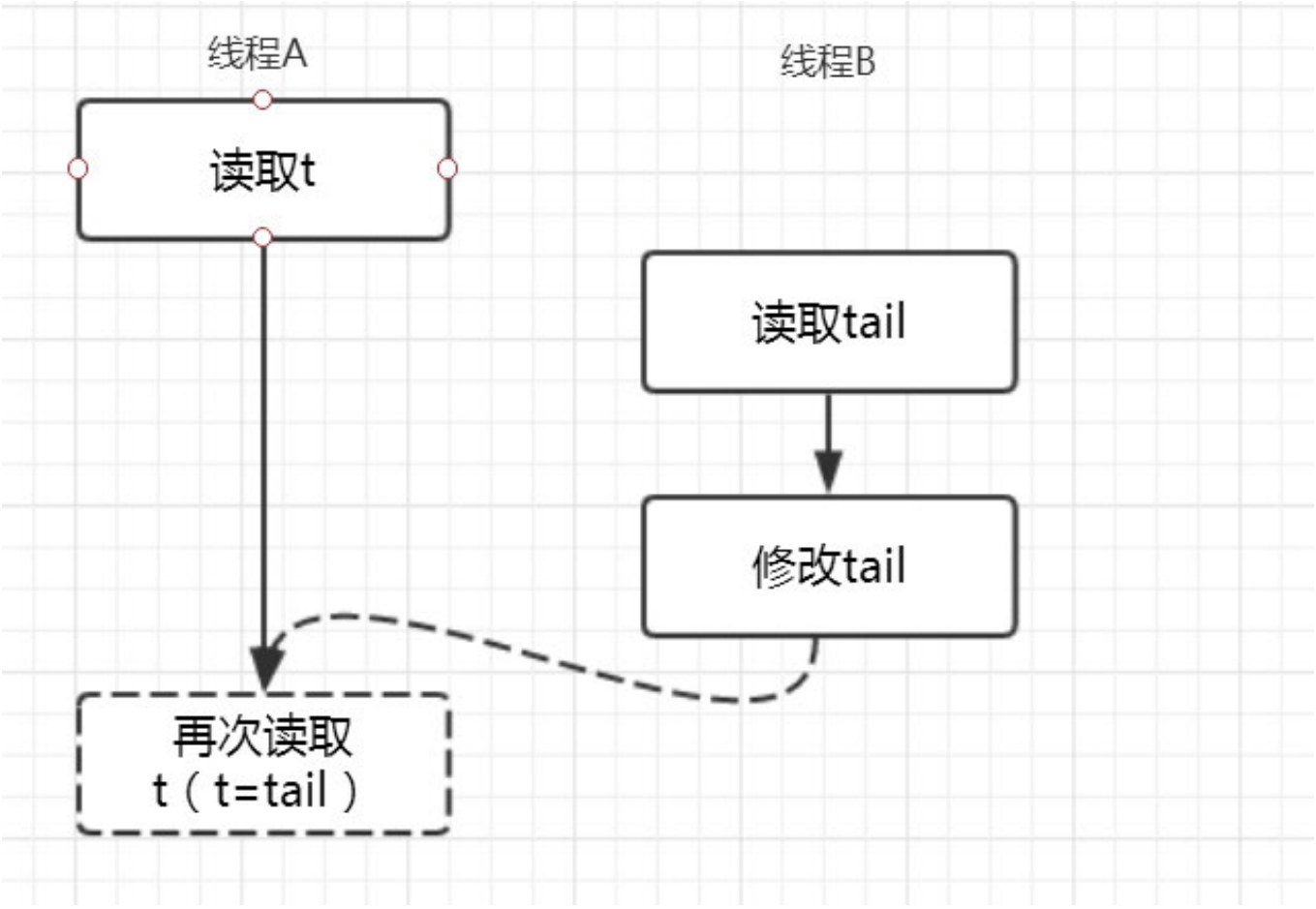
1. **如果 tail 节点的下一个节点（next 字段）为 null 的话，说明 tail 节点即为队列真正的尾节点，因此可以通过 casNext 插入当前待插入的节点，但此时 tail 并未变化**
2. **如果 tail 节点的下一个节点（next 字段）不为 null 的话，说明 tail 节点不是队列的真正尾节点。通过 `q（Node<E> q = p.next）` 往前找到尾节点，然后通过 casNext 插入当前待插入的节点，并通过 casTail 方式更新 tail。**

在单线程环境下，`p = (p != t && t != (t = tail)) ? t : q;`这行代码永远不会将 p 赋值为 t，我们试着在**多线程**的环境下继续分析。

多线程执行角度分析

在**多线程环境**下，`p = (p != t && t != (t = tail)) ? t : q;` 这行代码就有意思了。

由于 `t != (t = tail)` 这个操作**并非一个原子操作**，所以就有这样一种情况：



假设线程 A 此时读取了变量 t，线程 B 刚好在这个时候 offer 一个 Node，此时会修改 tail，那么线程 A 再次执行 t=tail 时，t 会指向另外一个节点，很显然线程 A 前后两次读取的变量 t 指向的节点不同，即`t != (t = tail)`为 true，并且由于 t 节点的变化，`p != t`也为 true，此时该行代码的执行结果是：p 和 t 都指向了同一个节点，并且 t 也是队列真正的尾节点。也就是说，现在已经定位到队列真正的尾节点，可以执行 offer 操作了。

到此为止，还剩下第 11 行的代码没有分析，大家应该可以猜到这种情况：**一部分线程 offer，一部分线程 poll**（下面会讲，用于检索并删除队列的头部元素，和 offer 是相对的）。

当`if (p == q)`为 true 时，说明 p 节点的 next 也指向它自己，这种节点称之为**哨兵节点，这种节点在队列中存在的价值不大，一般表示要删除的节点或者空节点**。为了能够更好地理解这种情况，我们先看看 poll 方法的执行过程，再回过头来看，总之这是一个很有意思的事情。

poll方法

poll 方法的源码如下：

```
public E poll() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            E item = p.item;

            if (item != null && p.casItem(item, null)) {
                // Successful CAS is the linearization point
                // for item to be removed from this queue.
                if (p != h) // hop two nodes at a time
```

```
                    updateHead(h, ((q = p.next) != null) ? q : p);
                return item;
            }
            else if ((q = p.next) == null) {
                updateHead(h, p);
                return null;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}
```

1、无限循环：外部的无限循环是为了确保在高并发环境中能够正确地从队列的头部移除元素。

2、初始化引用：对于当前头节点h和节点p（开始时与头节点相同）的初始化。

3、读取当前节点的项：`E item = p.item` 读取当前节点的元素。

4、检查当前项是否不为null：

- 如果是，并且CAS操作成功将该项设置为null（即`p.casItem(item, null)`），则表示元素已成功移除。
- 如果当前节点不是头节点（p != h），则更新头引用以"跳过"两个节点。
- 返回被移除的元素。

5、检查是否到达队尾：

- 如果q = p.next是null，则表示已到达队列的尾部。更新头引用，并返回null表示队列为空。
- 如果p == q，则表示可能有并发修改造成的异常情况，通过continue restartFromHead跳回外部循环的开始，重新尝试。

6、移动到下一个节点：将p设置为q，即下一个节点，并继续循环。

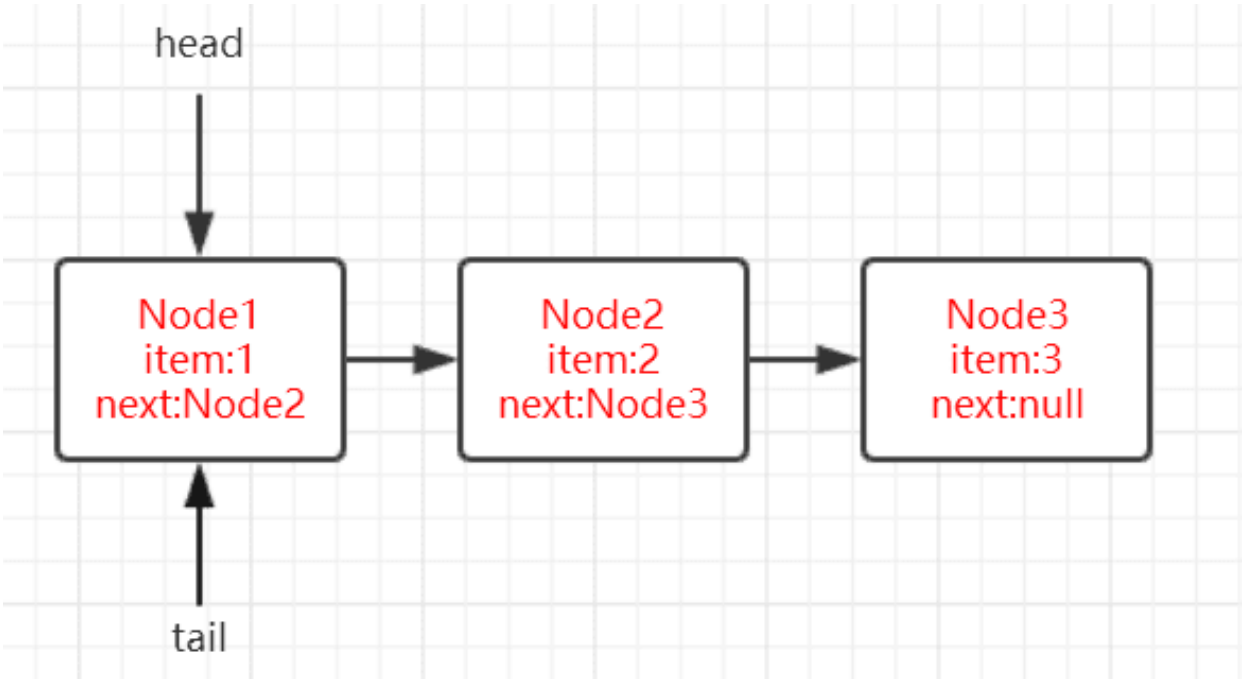
单线程执行角度分析

为了便于分析，我把代码注释删掉了，并标上行号。

```
public E poll() {
    restartFromHead:
    1. for (;;) {
    2.     for (Node<E> h = head, p = h, q;;) {
    3.         E item = p.item;
    4.         if (item != null && p.casItem(item, null)) {
    5.             if (p != h) // hop two nodes at a time
    6.                 updateHead(h, ((q = p.next) != null) ? q : p);
    7.             return item;
            }
    8.         else if ((q = p.next) == null) {
    9.             updateHead(h, p);
```

```
10.             return null;
            }
11.         else if (p == q)
12.             continue restartFromHead;
            else
13.             p = q;
        }
    }
}
```

假设 ConcurrentLinkedQueue 初始状态如下图所示：

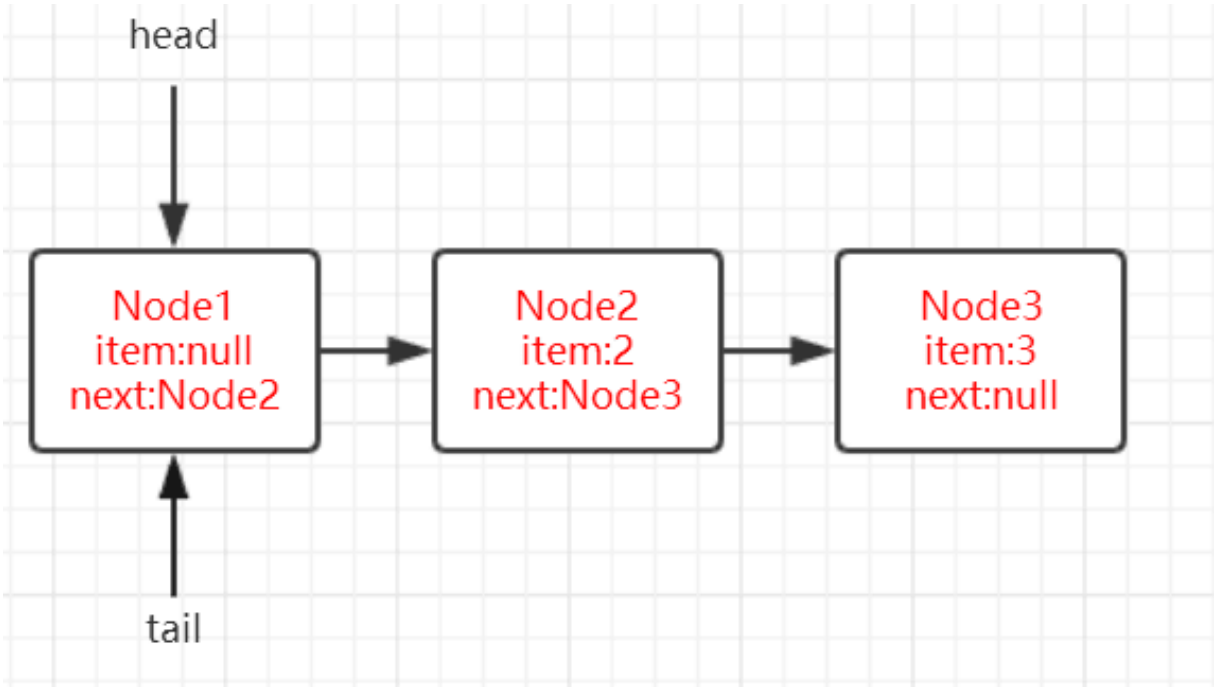


参数 offer 时的定义，我们将**变量 p 作为要删除的头节点，h（head）并不一定是队列的头节点**。

先来看 poll 出 Node1 时的情况，由于`p=h=head`，很显然此时 p 指向的 Node1 的数据不为 null，第 4 行代码`item!=null`的判断为 true，接下来通过`casItem`将 Node1 的数据设置为 null。

如果 CAS 失败则此次循环结束，等待下一次循环进行重试。

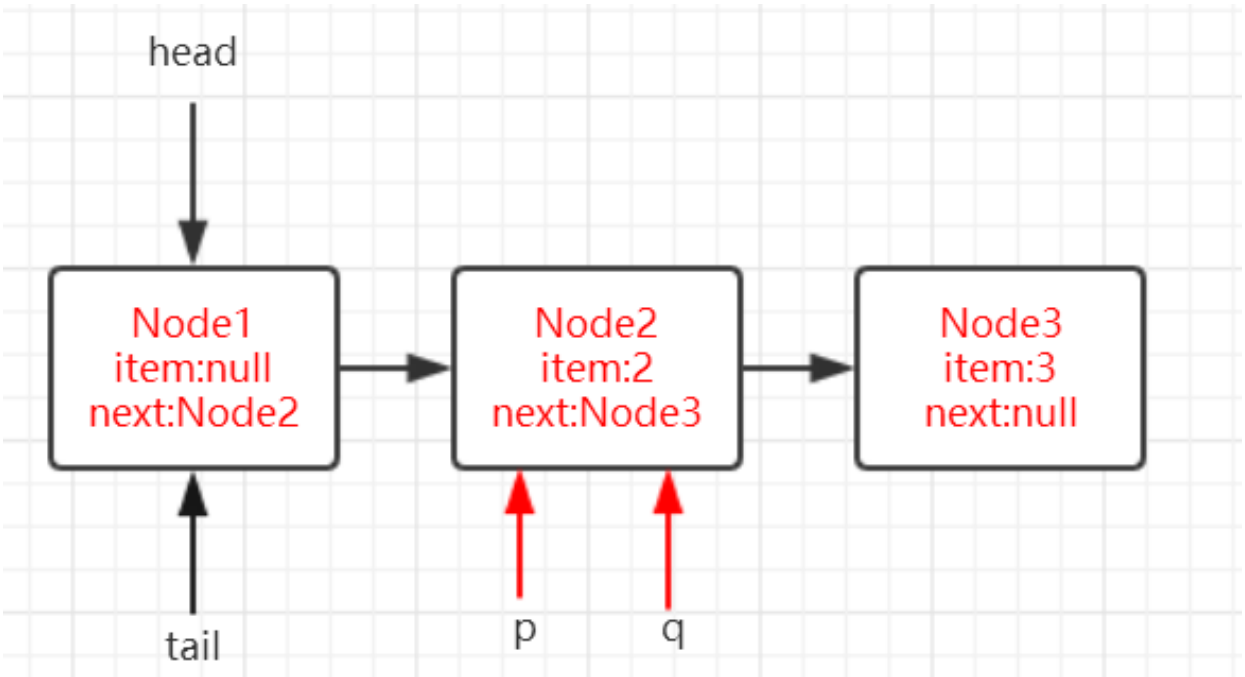
若第 4 行执行成功进入到第 5 行代码，此时 p 和 h 都指向 Node1，第 5 行 if 判断为 false，然后直接到第 7 行 return 回 Node1 的数据域 1，方法结束，此时的队列状态如下图所示。



继续从队列中 poll，很显然当前 h 和 p 指向的 Node1 的数据为 null，那么第一件事就是要**定位准备删除的头节点（找到数据不为 null 的节点）**。

继续看，第三行代码 item 为 null，第 4 行代码 if 判断为 false，走到第 8 行代码（`q = p.next`），if 也为 false，由于 q 指向了 Node2，第 11 行的 if 判断也为 false，因此代码走到了第 13 行，这个时候 p 和 q 共同指向了 Node2，也就找到了要删除的真正的头节点。

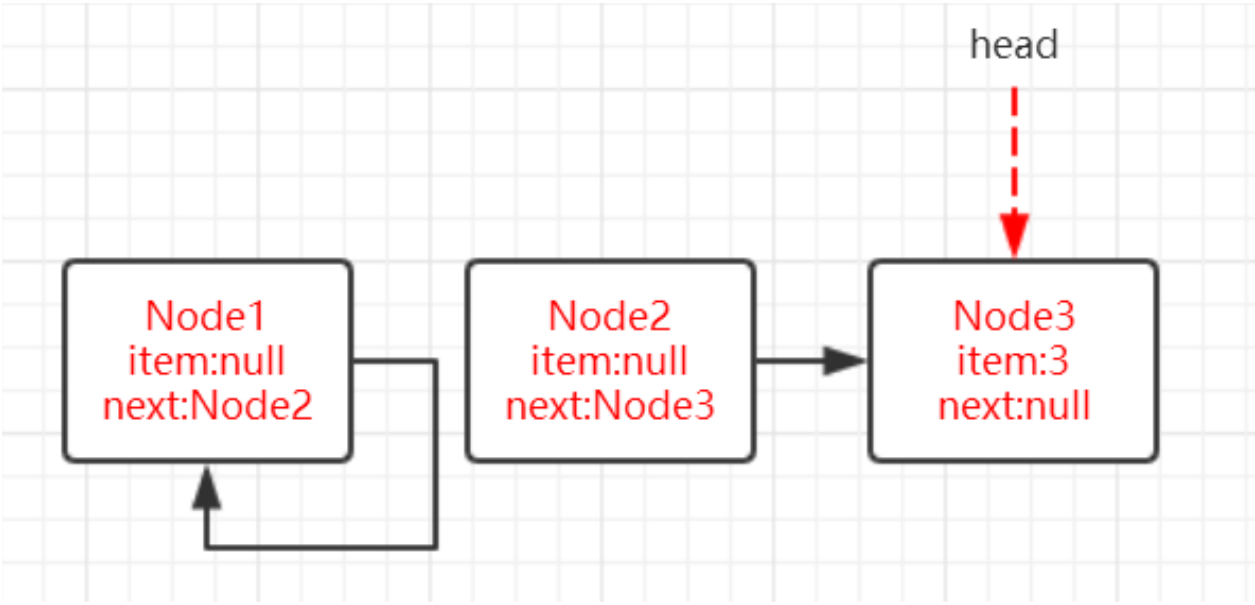
定位待删除的头节点的过程为：**如果当前节点的数据为 null，很显然该节点不是待删除的节点，就用当前节点的下一个节点去试探**。经过第一次循环后，此时状态图为下图所示：



进行下一次循环，第 4 行的操作同上所述，假设第 4 行中 casItem 设置成功，由于 p 已经指向了 Node2，而 h 依旧指向 Node1，此时第 5 行的 if 判断为 true，然后执行`updateHead(h, ((q = p.next) != null) ? q : p)`，此时 q 指向 Node3，updateHead 方法的源码如下：

```
final void updateHead(Node<E> h, Node<E> p) {
    if (h != p && casHead(h, p))
        h.lazySetNext(h);
}
```

该方法主要通过`casHead`将队列的 head 指向 Node3，并且通过`h.lazySetNext`将 Node1 的 next 指向它自己。最后在第 7 行代码返回 Node2 的值。此时队列的状态如下图所示：



Node1 的 next 指向它自己，head 指向了 Node3。

如果队列为空的话，就会执行到第 8 行`(q = p.next) == null`，if 判断为 true，因此在第 10 行中直接返回 null。

来做个总结：

1. **如果当前 head、h 和 p 指向的节点 Item 不为 null，说明该节点为真正的头节点（待删除节点），只需要通过 casItem 方法将 item 设置为 null，然后将原来的 item 返回即可。**
2. **如果当前 head、h 和 p 指向的节点 item 为 null 的话，说明该节点不是真正待删除的节点，那么应该继续寻找 item 不为 null 的节点。通过让 q 指向 p 的下一个节点（q = p.next）进行试探，若找到则通过 updateHead 方法更新 head 节点以及构造哨兵节点（通过`updateHead`方法的`h.lazySetNext(h)`）。**

多线程执行情况分析

现在回过头来看 poll 方法的源码，有这样一部分：

```
else if (p == q)
    continue restartFromHead;
```

这部分就是用来处理多个线程 poll 的，`q = p.next`，也就是说 q 永远指向的是 p 的下一个节点，那什么情况下 p 和 q 会指向同一个节点呢?

根据前面的分析，只有 p 指向的节点在 poll 的时候变成了**哨兵节点**（通过 updateHead 方法中的 h.lazySetNext）。

当线程 A 在判断`p==q`时，线程 B 已经执行完 poll 方法，将 p 节点转换为**哨兵节点**，并且 head 节点已经发生了改变，所以就需要从 restartFromHead 处执行，保证用到的是最新的 head。

试想，还有这样一种情况。如果当前队列为空队列，线程 A 进行 poll 操作，同时线程 B 执行 offer，然后线程 A 再执行 poll，那么此时线程 A 返回的是 null 还是线程 B 刚插入的那个节点呢？我们来写一段 demo：

```
public static void main(String[] args) {
    Thread thread1 = new Thread(() -> {
        Integer value = queue.poll();
        System.out.println(Thread.currentThread().getName() + " poll 的值为：" + value);
        System.out.println("queue当前是否为空队列：" + queue.isEmpty());
    });
    thread1.start();
    Thread thread2 = new Thread(() -> {
        queue.offer(1);
    });
    thread2.start();
}
```

输出结果为：

```
Thread-0 poll 的值为：null
queue当前是否为空队列：false
```

thread1 先执行到第 8 行代码`if ((q = p.next) == null)`，由于队列为空 if 判断为 true，进入 if 块，此时让 thread1 暂停，然后 thread2 进行 offer 插入值为 1 节点，thread2 执行结束。再让 thread1 执行，这时**thread1 并没有进行重试**，而是继续往下走，返回 null，尽管此时队列由于 thread2 已经插入了值为 1 的新节点。

输出结果为`thread0 poll 的为 null`，并且队列不为空。

因此，**在判断队列是否为空的时候，不能通过 poll 返回 null 进行判断，要通过 isEmpty 进行判断。**

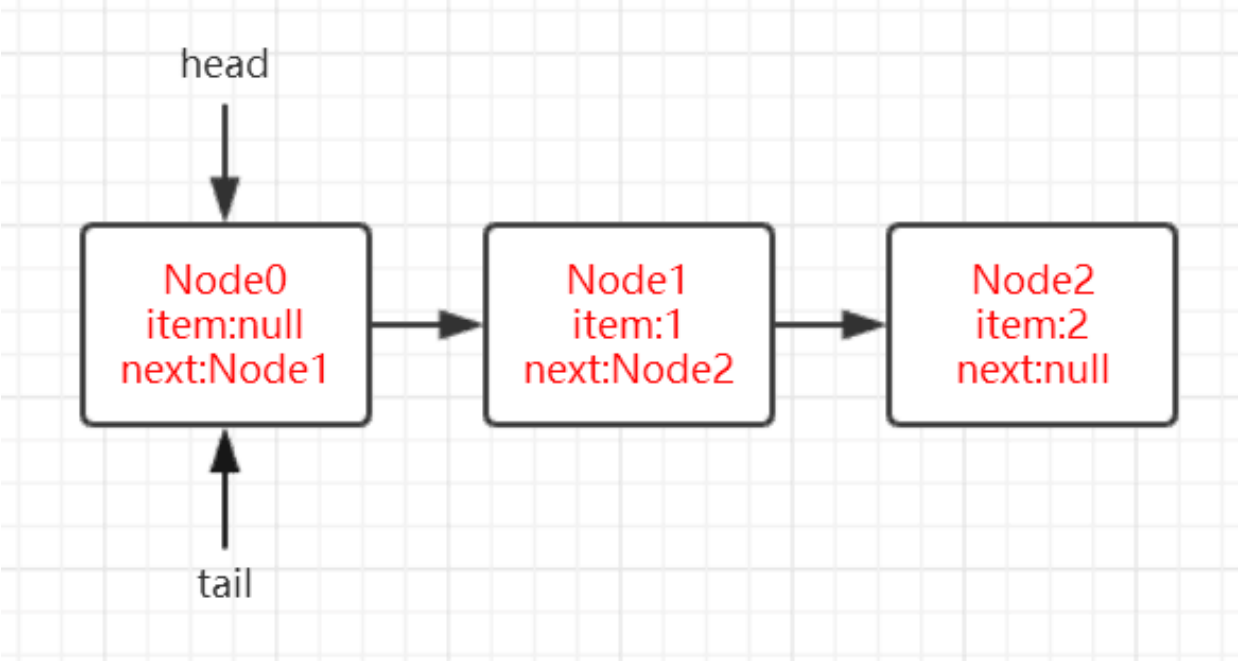
部分线程 offer 部分线程 poll

在分析 offer 方法的时候我们留了一个问题，即对 offer 方法中第 11 行代码的理解。

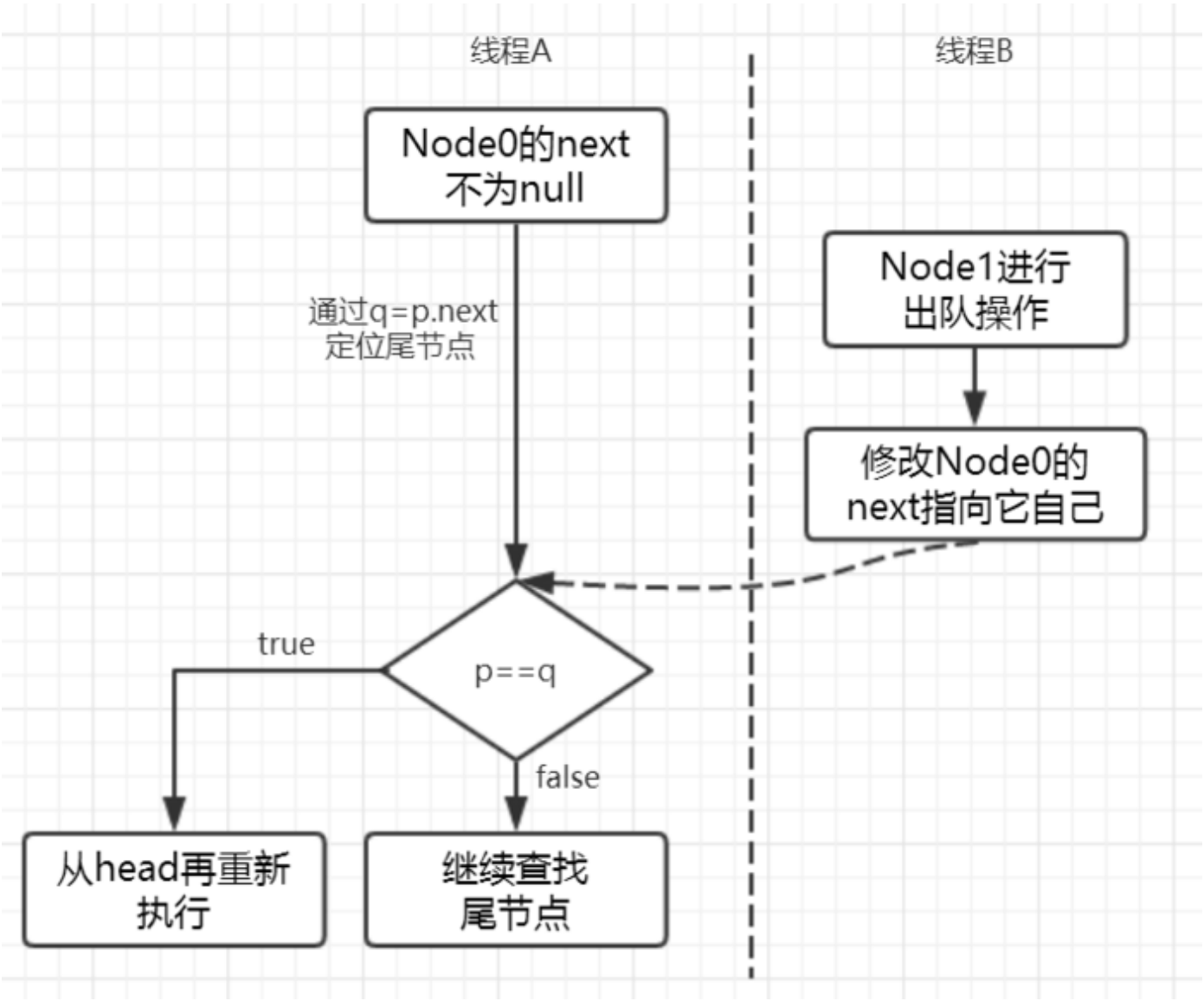
> **offer->poll->offer**

前面我们提到，offer 方法的第 11 行代码`if (p == q)`，能够让 if 条件为 true 的情况只有 p 节点为**哨兵节点**，什么时候会有哨兵节点呢？

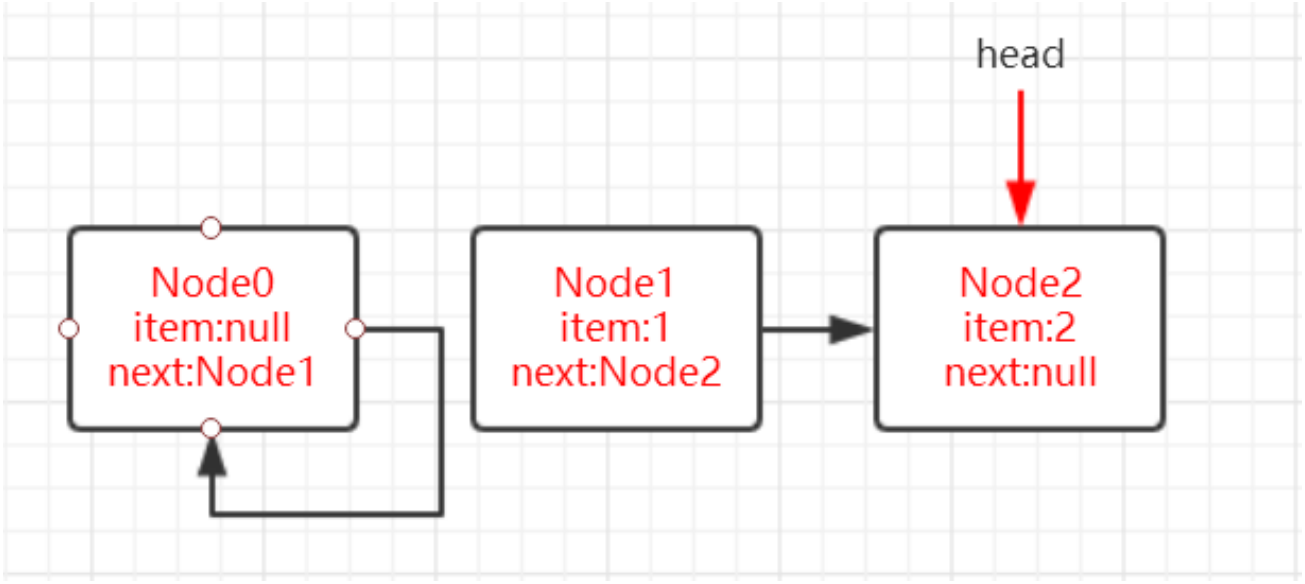
在 poll 方法的分析中，我们找到了答案，即**当 head 节点的 item 字段为 null 时会寻找真正的头节点，等到待插入的节点插入之后，会更新 head，并且将原 head 节点设置为哨兵节点。** 假设队列初始状态如下图所示：



因此在线程 A 执行 offer 时，线程 B 执行 poll 会存在如下一种情况：



线程 A 的 tail 节点存在 next 节点 Node1，因此会通过 q 往前寻找队列真正的尾节点，当执行到 `if (p == q)` 时，线程 B 执行 poll 操作，对线程 B 来说，head 和 p 指向 Node0，由于 Node0 的 item 字段为 null，同样会往前找队列的真正头节点 Node1，在线程 B 执行完 poll 后，Node0 就会转换为**哨兵节点**，也就意味着队列的 head 发生了改变，此时队列状态为下图所示。



线程 A 执行判断 `if (p == q)` 为 true，继续执行 `p = (t != (t = tail)) ? t : head;`，由于 tail 没有发生改变，所以 p 被赋值为 head，重新从 head 开始完成插入操作。

延迟更新策略

通过上面对 offer 和 poll 方法的分析，我们发现 tail 和 head 是延迟更新的，两者更新的触发时机为：

tail 更新的触发时机：当 tail 节点的下一个节点不为 null 的时候，会执行定位队列真正尾节点的操作，找到尾节点后完成插入，之后才会通过 casTail 进行 tail 更新；当 tail 节点的下一个节点为 null 的时候，只插入节点不更新 tail。

head 更新的触发时机：当 head 节点的 item 为 null 的时候，会执行定位队列真正头节点的操作，找到头节点后完成删除，之后才会通过 updateHead 进行 head 更新；当 head 节点的 item 不为 null 的时候，只删除节点不更新 head。

注意，源码中有这样一段注释：**hop two nodes at a time**。

所以这种延迟更新的策略叫做 HOPS，大概原因是这个（猜的），从上面更新时的状态图可以看出，head 和 tail 的更新是“跳着的”，即中间总是隔了一个。这样设计的意图是什么呢?

如果让 tail 永远作为尾节点，实现的代码量会更少，而且逻辑更易懂。

但是，这样做有一个缺点，**如果有大量的入队操作，每次都要执行 CAS 进行 tail 的更新，汇总起来对性能也是非常大的损耗。如果能减少 CAS 更新操作，就可以大大提升入队的操作效率，所以 doug lea 大师每间隔 1 次（tail 和队尾节点的距离为 1）才利用 CAS 更新 tail。**

对 head 的更新也是同样的道理，虽然这样设计会多出在循环中定位尾节点的操作，但总体来说，读的操作效率要远远高于写的效率，因此，多出来的定位尾节点的性能损耗相对就很小了。

使用示例

```
public class ConcurrentLinkedQueueTest {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
        queue.offer(1);
        queue.offer(2);
        queue.offer(3);
        queue.offer(4);
        queue.offer(5);
        System.out.println("queue当前是否为空队列：" + queue.isEmpty());
        System.out.poll();
        System.out.println("queue当前是否为空队列：" + queue.isEmpty());
        System.out.println("queue当前的大小为：" + queue.size());
    }
}
```

输出结果为：

```
queue当前是否为空队列：false
queue当前是否为空队列：false
queue当前的大小为：4
```

小结

ConcurrentLinkedQueue 是一种先进先出（FIFO，First-In-First-Out）的队列，它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先加入的，尾是最近加入的。该队列不允许 null 元素。

ConcurrentLinkedQueue 采用了 HOPS 的设计，即 head 和 tail 是延迟更新的，这种设计的主要目的是减小多线程环境下的争用，并提高性能。

ConcurrentLinkedQueue 的 offer 方法用于在队列尾部插入一个元素。如果成功添加元素，则返回 true。

ConcurrentLinkedQueue 的 poll 方法用于检索并删除队列的头部元素。如果队列为空，则返回 null。

ConcurrentLinkedQueue 的 isEmpty 方法用于检索队列是否为空。

ConcurrentLinkedQueue 的 size 方法用于返回队列的大小。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency。

第二十二节：阻塞队列 BlockingQueue

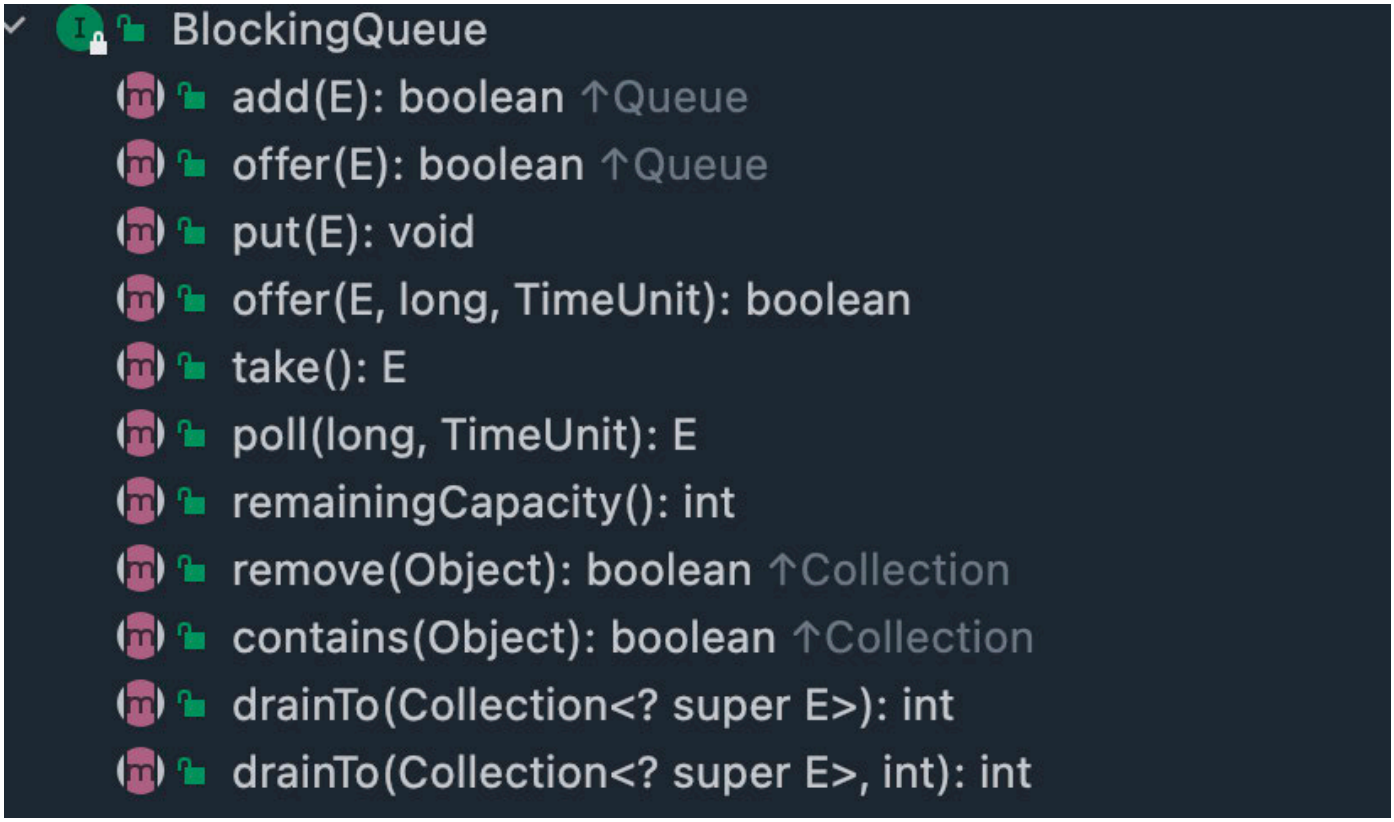
BlockingQueue 是 Java 中的一个接口，它代表了一个线程安全的队列，不仅可以由多个线程并发访问，还添加了等待/通知机制，以便在队列为空时阻塞获取元素的线程，直到队列变得可用，或者在队列满时阻塞插入元素的线程，直到队列变得可用。

最常用的"生产者-消费者"问题中，队列通常被视作线程间的数据容器，生产者将“生产”出来的数据放入数据容器，消费者从“数据容器”中获取数据，这样，生产者线程和消费者线程就解耦了，各自只需要专注自己的业务即可。

阻塞队列（BlockingQueue）被广泛用于“生产者-消费者”问题中，其原因是 BlockingQueue 提供了可阻塞的插入和移除方法。**当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。**

基本操作

BlockingQueue 接口定义的方法如下所示：



由于 BlockingQueue 继承了 Queue 接口，因此，BlockingQueue 也具有 Queue 接口的基本操作，如下所示：

1）插入元素

1. `boolean add(E e)`：将元素添加到队列尾部，如果队列满了，则抛出异常 IllegalStateException。
2. `boolean offer(E e)`：将元素添加到队列尾部，如果队列满了，则返回 false。

2）删除元素

1. `boolean remove(Object o)`：从队列中删除元素，成功返回`true`，失败返回`false`
2. `E poll()`：检索并删除此队列的头部，如果此队列为空，则返回null。

3）查找元素

1. `E element()`：检索但不删除此队列的头部，如果队列为空时则抛出 NoSuchElementException 异常；
2. `peek()`：检索但不删除此队列的头部，如果此队列为空，则返回 null.

除了从 Queue 接口 继承到一些方法，BlockingQueue 自身还定义了一些其他的方法，比如说插入操作：

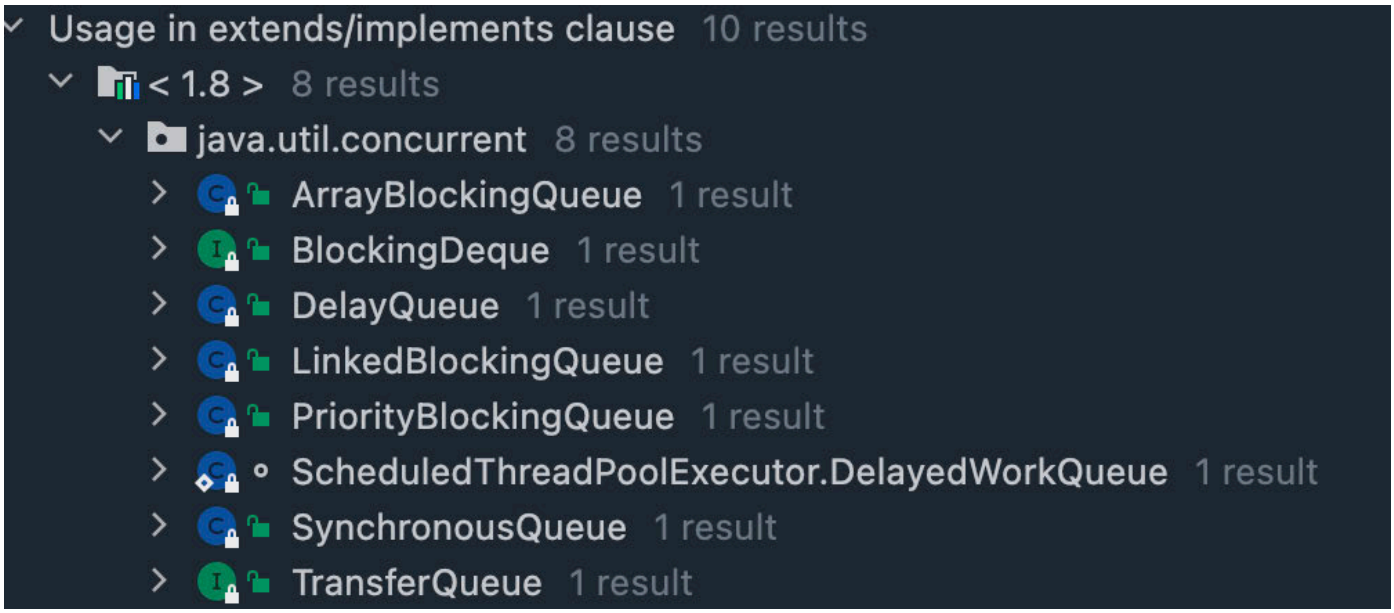
1. `void put(E e)`：将元素添加到队列尾部，如果队列满了，则线程将阻塞直到有空间。
2. `offer(E e, long timeout, TimeUnit unit)`：将指定的元素插入此队列中，如果队列满了，则等待指定的时间，直到队列可用。

比如说删除操作：

1. `take()`：检索并删除此队列的头部，如有必要，则等待直到队列可用；
2. `poll(long timeout, TimeUnit unit)`：检索并删除此队列的头部，如果需要元素变得可用，则等待指定的等待时间。

ArrayBlockingQueue

BlockingQueue 接口的实现类有 ArrayBlockingQueue、DelayQueue、LinkedBlockingDeque、LinkedBlockingQueue、LinkedTransferQueue、PriorityBlockingQueue、SynchronousQueue 等，我们先从 ArrayBlockingQueue 说起。



ArrayBlockingQueue 它是一个基于数组的有界阻塞队列：

- 有界：ArrayBlockingQueue 的大小是在构造时就确定了，并且在之后不能更改。这个界限提供了流量控制，有助于资源的合理使用。
- FIFO：队列操作符合先进先出的原则。
- 当队列容量满时，尝试将元素放入队列将导致阻塞；尝试从一个空的队列取出元素也会阻塞。

需要注意的是，ArrayBlockingQueue 并不能保证绝对的公平，所谓公平是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到 ArrayBlockingQueue。

这是因为还有其他系统级别的因素，如线程调度，可能会影响到实际的执行顺序。如果需要公平的 ArrayBlockingQueue，可在声明的时候设置公平标志为 true：

```
private static ArrayBlockingQueue<Integer> blockingQueue = new
ArrayBlockingQueue<Integer>(10, true);
```

ArrayBlockingQueue 的字段如下：

```
/** The queued items */
final Object[] items;

/** items index for next take, poll, peek or remove */
int takeIndex;

/** items index for next put, offer, or add */
int putIndex;

/** Number of elements in the queue */
```

```
int count;

/*
 * Concurrency control uses the classic two-condition algorithm
 * found in any textbook.
 */

/** Main lock guarding all access */
final ReentrantLock lock;

/** Condition for waiting takes */
private final Condition notEmpty;

/** Condition for waiting puts */
private final Condition notFull;
```

- items: 这是一个用于存储队列元素的数组。队列的大小在构造时定义，并且在生命周期内不会改变。
- takeIndex: 这个索引用于下一个 take、poll、peek 或 remove 操作。它指向当前可被消费的元素位置。
- putIndex: 这个索引用于下一个 put、offer 或 add 操作。它指向新元素将被插入的位置。
- count: 这是队列中当前元素的数量。当达到数组大小时，进一步的 put 操作将被阻塞。
- lock: 这是用于保护队列访问的 ReentrantLock 对象。所有的访问和修改队列的操作都需要通过这个锁来同步。
- notEmpty: 这个条件 Condition 用于等待 take 操作。当队列为空时，尝试从队列中取元素的线程将等待这个条件。
- notFull: 这个条件 Condition 用于等待 put 操作。当队列已满时，尝试向队列中添加元素的线程将等待这个条件。

构造方法如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

1）put 方法详解

`put(E e)`方法源码如下：

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
    //如果当前队列已满，将线程移入到notFull等待队列中
```

```
            while (count == items.length)
                notFull.await();
        //满足插入数据的要求，直接进行入队操作
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }
```

该方法的逻辑很简单，当队列已满时（`count == items.length`）将线程移入到 notFull 等待队列中，如果满足插入数据的条件，直接调用`enqueue(e)`插入元素。enqueue 方法源码如下：

```
private void enqueue(E x) {
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
  //插入数据
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
  //通知消费者线程，当前队列中有数据可供消费
    notEmpty.signal();
}
```

enqueue 方法的逻辑同样很简单，先插入数据（`items[putIndex] = x`），然后通知被阻塞的消费者线程：当前队列中有数据可供消费（`notEmpty.signal()`）了。

2）take 方法详解

take 方法的源码如下：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
    //如果队列为空，没有数据，将消费者线程移入等待队列中
        while (count == 0)
            notEmpty.await();
    //获取数据
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

1. 如果当前队列为空的话，则将获取数据的消费者线程移入到等待队列中；
2. 如果队列不为空则获取数据，即完成出队操作`dequeue`。dequeue 方法源码如下：

```
private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
  //获取数据
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0;
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    //通知被阻塞的生产者线程
  notFull.signal();
    return x;
}
```

dequeue 方法主要做了两件事情：

1. 获取队列中的数据（`(E) items[takeIndex]`）；
2. 通知可能正在等待插入元素的生产者线程队列现在有可用空间，通过调用notFull 条件变量的 signal 方法实现。

从以上分析可以看出，put 和 take 方法主要通过 Condition 的通知机制来完成阻塞式的数据生产和消费。

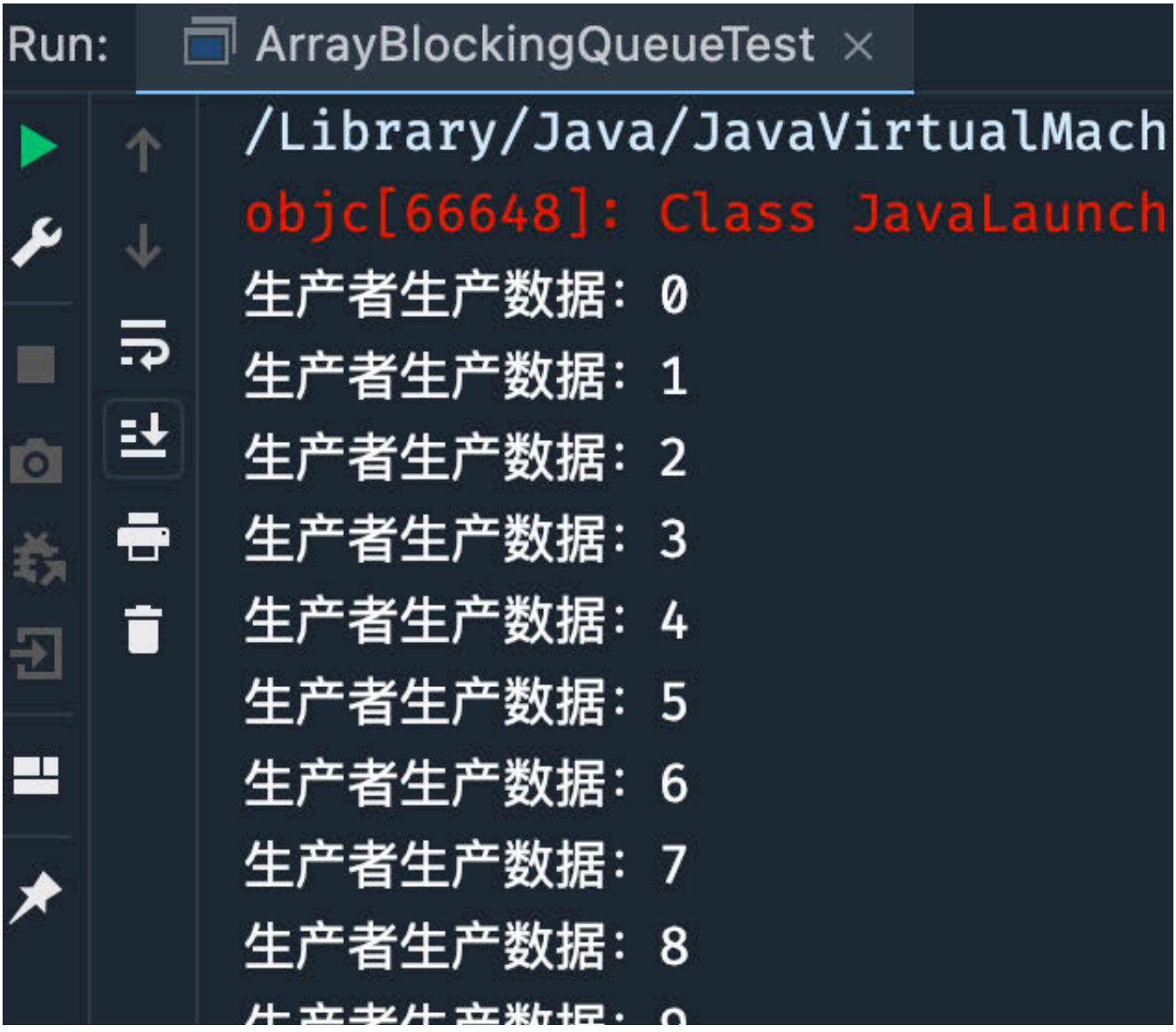
3）使用示例

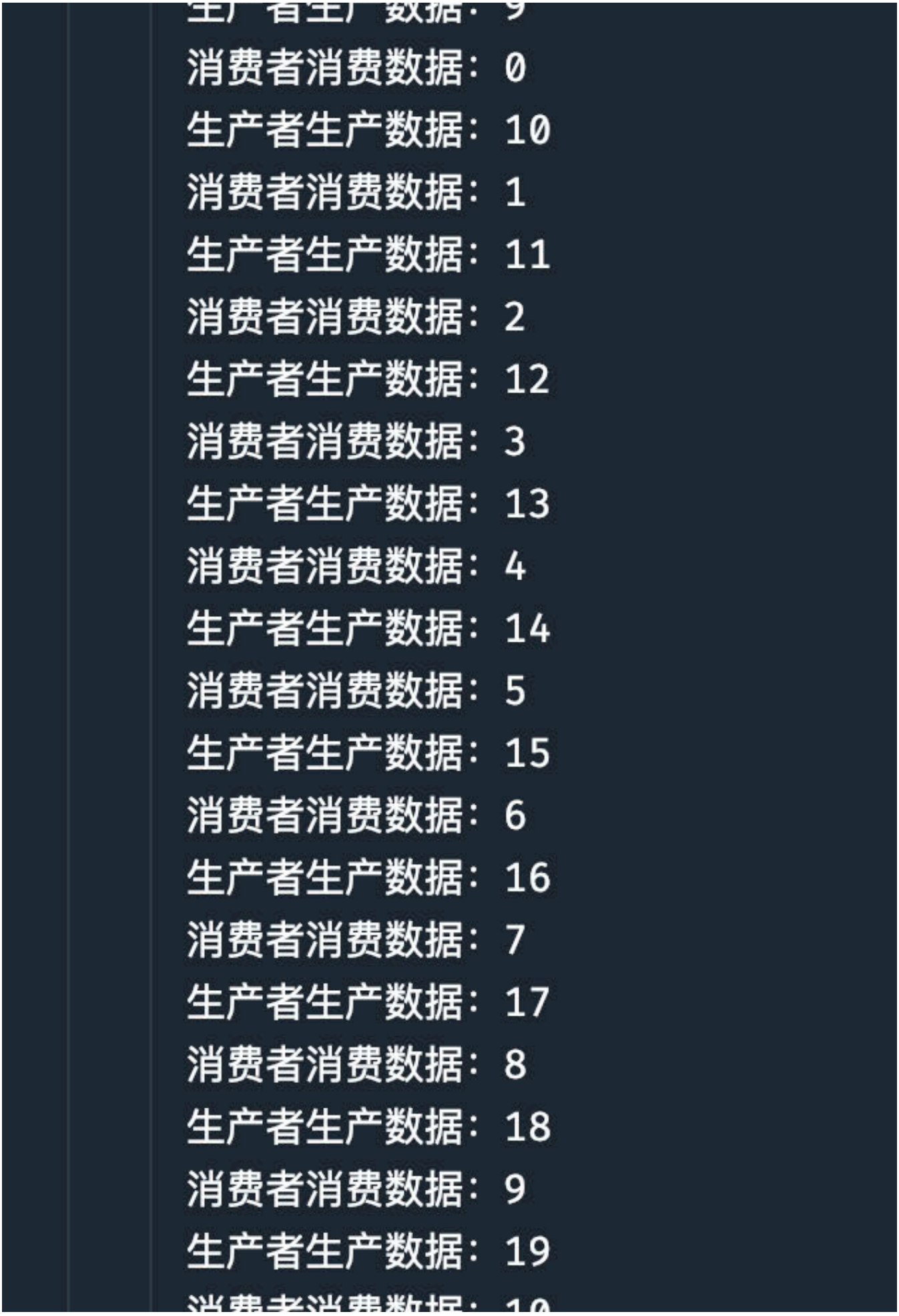
OK，我们再来看一个 ArrayBlockingQueue 的使用示例：

```
public class ArrayBlockingQueueTest {
    private static ArrayBlockingQueue<Integer> blockingQueue = new
ArrayBlockingQueue<Integer>(10, true);

    public static void main(String[] args) {
        new Thread(new Producer()).start();
        new Thread(new Consumer()).start();
    }

    static class Producer implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                try {
                    blockingQueue.put(i);
                    System.out.println("生产者生产数据：" + i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
            }
        }
    }

    static class Consumer implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                try {
                    Integer data = blockingQueue.take();
                    System.out.println("消费者消费数据：" + data);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

运行的部分结果如下图所示：




生产者生产数据：9
消费者消费数据：0
生产者生产数据：10
消费者消费数据：1
生产者生产数据：11
消费者消费数据：2
生产者生产数据：12
消费者消费数据：3
生产者生产数据：13
消费者消费数据：4
生产者生产数据：14
消费者消费数据：5
生产者生产数据：15
消费者消费数据：6
生产者生产数据：16
消费者消费数据：7
生产者生产数据：17
消费者消费数据：8
生产者生产数据：18
消费者消费数据：9
生产者生产数据：19

LinkedBlockingQueue

LinkedBlockingQueue 是一个基于链表的线程安全的阻塞队列：

- 可以在队列头部和尾部进行高效的插入和删除操作。
- 当队列为空时，取操作会被阻塞，直到队列中有新的元素可用。当队列已满时，插入操作会被阻塞，直到队列有可用空间。
- 可以在构造时指定最大容量。如果不指定，默认为 Integer.MAX_VALUE，这意味着队列的大小受限于可用内存。

LinkedBlockingQueue 的字段如下：

```
/** Current number of elements */
private final AtomicInteger count = new AtomicInteger();

/**
 * Head of linked list.
 * Invariant: head.item == null
 */
transient Node<E> head;

/**
 * Tail of linked list.
 * Invariant: last.next == null
 */
private transient Node<E> last;

/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();

/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();

/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();

/** Wait queue for waiting puts */
private final Condition notFull = putLock.newCondition();
```

- count: 一个 AtomicInteger，表示队列中当前元素的数量。通过原子操作保证其线程安全。
- head: 队列的头部节点。由于这是一个 FIFO 队列，所以元素总是从头部移除。头部节点的 item 字段始终为 null，它作为一个虚拟节点，用于帮助管理队列。
- last: 队列的尾部节点。新元素总是插入到尾部。
- takeLock 和 putLock: 这是 LinkedBlockingQueue 中的两把 ReentrantLock 锁。takeLock 用于控制取操作，putLock 用于控制放入操作。这样的设计使得放入和取出操作能够在一定程度上并行执行，从而提高队列的吞吐量。
- notEmpty 和 notFull: 这是两个 Condition 变量，分别与 takeLock 和 putLock 相关联。当队列为空时，尝试从队列中取出元素的线程将会在 notEmpty 上等待。当新元素被放入队列时，这些等待的线程将会被唤醒。同样地，当队列已满时，尝试向队列中放入元素的线程将会在 notFull 上等待，等待队列有可用空间时被唤

醒。

链表的 Node 节点的定义如下：

```
static class Node<E> {
    E item;

    /**
     * One of:
     * - the real successor Node
     * - this Node, meaning the successor is head.next
     * - null, meaning there is no successor (this is the last node)
     */
    Node<E> next;

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

01）item: 这个字段用于存储节点包含的元素。

02）next: 这个字段表示节点在队列中的后继节点。这个字段有三个可能的值：

- 后继节点的实际引用。
- 此节点自身的引用，意味着后继节点是头节点的下一个节点。
- null，表示没有后继节点，也就是说此节点是队列的最后一个节点。

03）`Node(E x)`: 这是节点类的构造方法，它接受一个元素 x 并将其赋值给 item 字段。

1）put 方法详解

put 方法源码如下：

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset local var
    // holding count negative to indicate failure unless set.
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        /*
         * Note that count is used in wait guard even though it is
         * not protected by lock. This works because count can
         * only decrease at this point (all other puts are shut
         * out by lock), and we (or some other waiting put) are
         * signalled if it ever changes from capacity. Similarly
         * for all other uses of count in other wait guards.
         */
```

```
    //如果队列已满，则阻塞当前线程，将其移入等待队列
        while (count.get() == capacity) {
            notFull.await();
        }
    //入队操作，插入数据
        enqueue(node);
        c = count.getAndIncrement();
    //若队列满足插入数据的条件，则通知被阻塞的生产者线程
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
}
```

put 方法的逻辑基本上和 ArrayBlockingQueue 的一样。

01）参数检查：如果传入的元素为 null，则抛出 NullPointerException。LinkedBlockingQueue 不允许插入 null 元素。

02）局部变量初始化：

- `int c = -1;` 用于存储操作前的队列元素数量，预设为 -1 表示失败，除非稍后设置。
- `Node<E> node = new Node<E>(e);` 创建一个新的节点包含要插入的元素 e。
- `final ReentrantLock putLock = this.putLock;` 和 `final AtomicInteger count = this.count;` 获取队列的锁和计数器对象。

03）获取锁：`putLock.lockInterruptibly();` 尝试获取用于插入操作的锁，如果线程被中断，则抛出 InterruptedException。

04）等待队列非满：如果队列已满（`count.get() == capacity`），当前线程将被阻塞，并等待 notFull 条件被满足。一旦有空间可用，线程将被唤醒继续执行。

05）入队操作：调用 `enqueue(node);` 将新节点插入队列的尾部。

06）更新计数：通过 `c = count.getAndIncrement();` 获取并递增队列的元素计数。

07）检查并可能的唤醒其他生产者线程：如果队列没有满（`c + 1 < capacity`），使用 `notFull.signal();` 唤醒可能正在等待插入空间的其他生产者线程。

08）释放锁：finally 块确保锁在操作完成后被释放。

09）可能的唤醒消费者线程：如果插入操作将队列从空变为非空（`c == 0`），则调用 `signalNotEmpty();` 唤醒可能正在等待非空队列的消费者线程。

2）take 方法详解

take 方法的源码如下：

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
    //当前队列为空，则阻塞当前线程，将其移入到等待队列中，直至满足条件
        while (count.get() == 0) {
            notEmpty.await();
        }
    //移除队头元素，获取数据
        x = dequeue();
        c = count.getAndDecrement();
        //如果当前满足移除元素的条件，则通知被阻塞的消费者线程
    if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

01）局部变量初始化：

- `E x;` 用于存储被取出的元素。
- `int c = -1;` 用于存储操作前的队列元素数量，预设为 -1 表示失败，除非稍后设置。
- `final AtomicInteger count = this.count;` 和 `final ReentrantLock takeLock = this.takeLock;` 获取队列的计数器和锁对象。

02）获取锁：`takeLock.lockInterruptibly();` 尝试获取用于取出操作的锁，如果线程被中断，则抛出 InterruptedException。

03）等待队列非空：如果队列为空（`count.get() == 0`），当前线程将被阻塞，并等待 notEmpty 条件被满足。一旦队列非空，线程将被唤醒继续执行。

04）出队操作：调用 `x = dequeue();` 从队列的头部移除元素，并将其赋值给 x。

05）更新计数：通过 `c = count.getAndDecrement();` 获取并递减队列的元素计数。

06）检查并可能的唤醒其他消费者线程：如果队列仍有其他元素（`c > 1`），使用 `notEmpty.signal();` 唤醒可能正在等待非空队列的其他消费者线程。

07）释放锁：finally 块确保锁在操作完成后被释放。

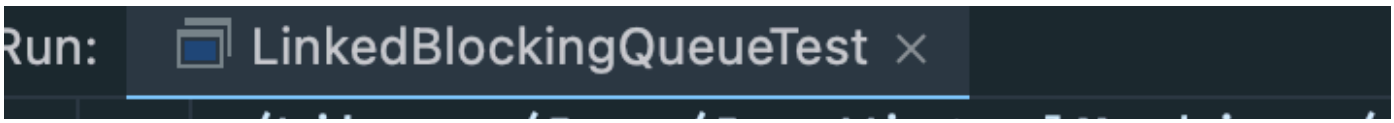
08）可能的唤醒生产者线程：如果取出操作将队列从满变为未满（`c == capacity`），则调用 `signalNotFull();` 唤醒可能正在等待插入空间的生产者线程。

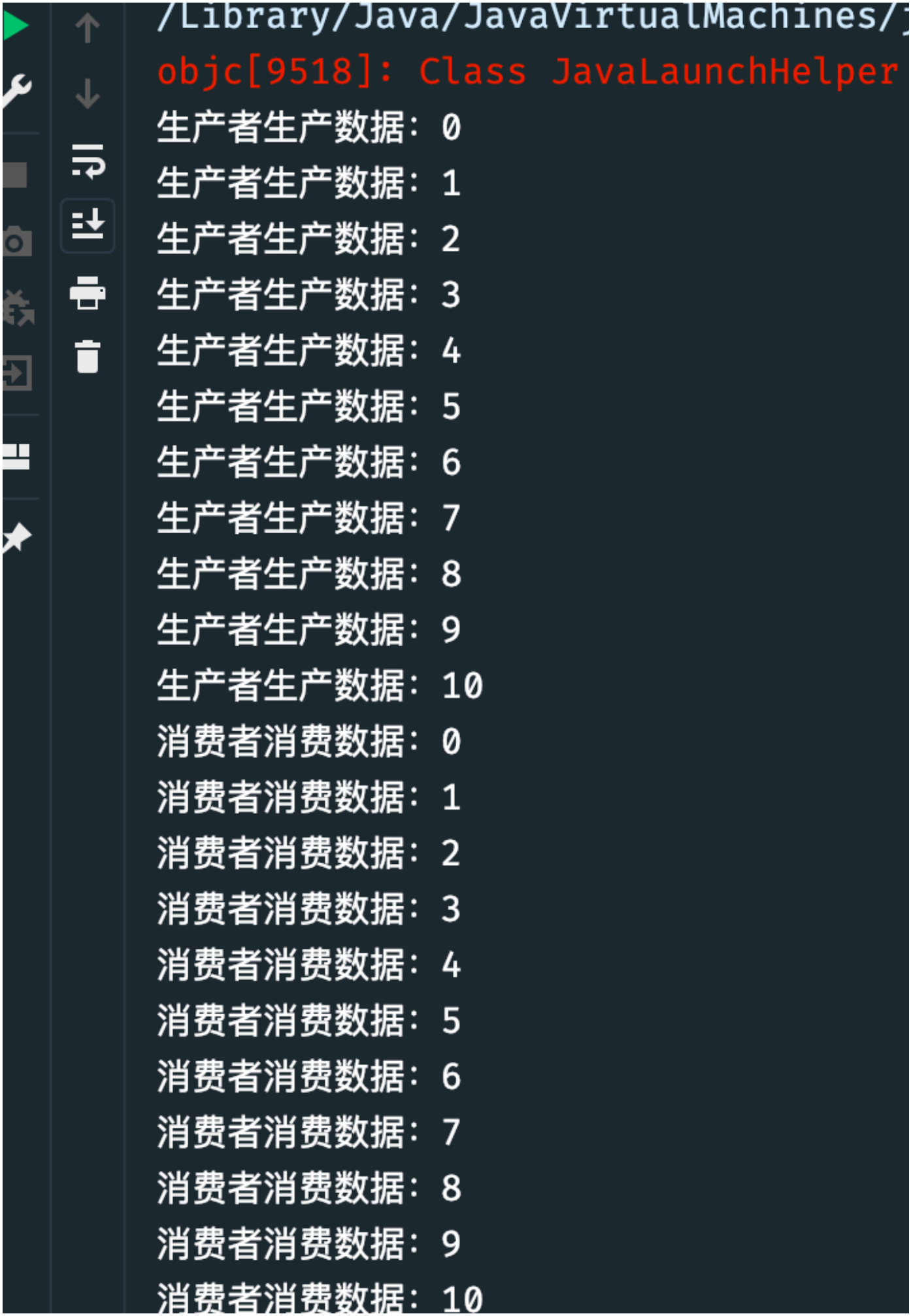
09）返回取出的元素：最后返回被取出的元素 x。

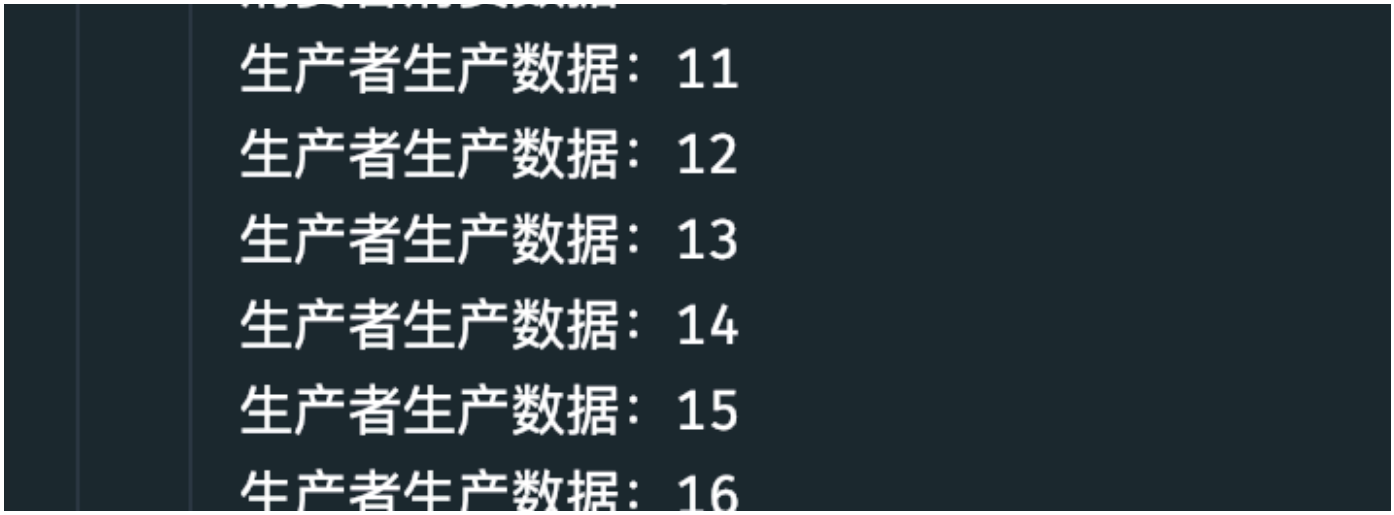
3）使用示例

```
public class LinkedBlockingQueueTest {
    private static LinkedBlockingQueue<Integer> blockingQueue = new
LinkedBlockingQueue<Integer>(10);

    public static void main(String[] args) {
        new Thread(new Producer()).start();
        new Thread(new Consumer()).start();
    }

    static class Producer implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                try {
                    blockingQueue.put(i);
                    System.out.println("生产者生产数据：" + i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static class Consumer implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                try {
                    Integer data = blockingQueue.take();
                    System.out.println("消费者消费数据：" + data);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

运行的部分结果如下图所示：




/Library/Java/JavaVirtualMachines/
objc[9518]: Class JavaLaunchHelper
生产者生产数据：0
生产者生产数据：1
生产者生产数据：2
生产者生产数据：3
生产者生产数据：4
生产者生产数据：5
生产者生产数据：6
生产者生产数据：7
生产者生产数据：8
生产者生产数据：9
生产者生产数据：10
消费者消费数据：0
消费者消费数据：1
消费者消费数据：2
消费者消费数据：3
消费者消费数据：4
消费者消费数据：5
消费者消费数据：6
消费者消费数据：7
消费者消费数据：8
消费者消费数据：9
消费者消费数据：10



ArrayBlockingQueue 与 LinkedBlockingQueue 的比较

相同点：ArrayBlockingQueue 和 LinkedBlockingQueue 都是通过 Condition 通知机制来实现可阻塞的插入和删除。

不同点：

1. ArrayBlockingQueue 基于数组实现，而 LinkedBlockingQueue 基于链表实现；
2. ArrayBlockingQueue 使用一个单独的 ReentrantLock 来控制对队列的访问，而 LinkedBlockingQueue 使用两个锁（putLock 和 takeLock），一个用于放入操作，另一个用于取出操作。这可以提供更细粒度的控制，并可能减少线程之间的竞争。

PriorityBlockingQueue

PriorityBlockingQueue 是一个具有优先级排序特性的无界阻塞队列。元素在队列中的排序遵循自然排序或者通过提供的比较器进行定制排序。你可以通过实现 Comparable 接口来定义自然排序。

当需要根据优先级来执行任务时，PriorityBlockingQueue 会非常有用。下面的代码演示了如何使用 PriorityBlockingQueue 来管理具有不同优先级的任务。

```
class Task implements Comparable<Task> {
    private int priority;
    private String name;

    public Task(int priority, String name) {
        this.priority = priority;
        this.name = name;
    }

    public int compareTo(Task other) {
        return Integer.compare(other.priority, this.priority); // higher values have
higher priority
    }

    public String getName() {
        return name;
```

```
    }
}

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        PriorityBlockingQueue<Task> queue = new PriorityBlockingQueue<>();
        queue.put(new Task(1, "Low priority task"));
        queue.put(new Task(50, "High priority task"));
        queue.put(new Task(10, "Medium priority task"));

        while (!queue.isEmpty()) {
            System.out.println(queue.take().getName());
        }
    }
}
```

上例创建了一个优先级阻塞队列，并添加了三个具有不同优先级的任务。它们会按优先级从高到低的顺序被取出并打印。运行结果如下：

```
High priority task
Medium priority task
Low priority task
```

SynchronousQueue

SynchronousQueue 是一个非常特殊的阻塞队列，它不存储任何元素。每一个插入操作必须等待另一个线程的移除操作，反之亦然。因此，SynchronousQueue 的内部实际上是空的，但它允许一个线程向另一个线程逐个传输元素。

SynchronousQueue 允许线程直接将元素交付给另一个线程。因此，如果一个线程尝试插入一个元素，并且有另一个线程尝试移除一个元素，则插入和移除操作将同时成功。

如果想让一个线程将确切的信息直接发送给另一个线程的情况下，可以使用 SynchronousQueue。下面的代码展示了如何使用 SynchronousQueue 进行线程间的通信：

```
public class SynchronousQueueDemo {
    public static void main(String[] args) {
        SynchronousQueue<String> queue = new SynchronousQueue<>();

        // Producer Thread
        new Thread(() -> {
            try {
                String event = "SYNCHRONOUS_EVENT";
                System.out.println("Putting: " + event);
                queue.put(event);
                System.out.println("Put successfully: " + event);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
```

```
        }).start();

        // Consumer Thread
        new Thread(() -> {
            try {
                String event = queue.take();
                System.out.println("Taken: " + event);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }
}
```

上例创建了一个 SynchronousQueue，并在一个线程中插入了一个元素，另一个线程中移除了这个元素。运行结果如下：

```
Putting: SYNCHRONOUS_EVENT
Put successfully: SYNCHRONOUS_EVENT
Taken: SYNCHRONOUS_EVENT
```

LinkedTransferQueue

LinkedTransferQueue 是一个基于链表结构的无界传输队列，实现了 TransferQueue 接口，它提供了一种强大的线程间交流机制。它的功能与其他阻塞队列类似，但还包括“转移”语义：允许一个元素直接从生产者传输给消费者，如果消费者已经在等待。如果没有等待的消费者，元素将入队。

常用方法有两个：

- `transfer(E e)`，将元素转移到等待的消费者，如果不存在等待的消费者，则元素会入队并阻塞直到该元素被消费。
- `tryTransfer(E e)`，尝试立即转移元素，如果有消费者正在等待，则传输成功；否则，返回 false。

如果想要更紧密地控制生产者和消费者之间的交互，可以使用 LinkedTransferQueue。

```
public class LinkedTransferQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        LinkedTransferQueue<String> queue = new LinkedTransferQueue<>();

        // Consumer Thread
        new Thread(() -> {
            try {
                System.out.println("消费者正在等待获取元素...");
                String element = queue.take();
                System.out.println("消费者收到: " + element);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
```

```
        // Let consumer thread start first
        TimeUnit.SECONDS.sleep(1);

        // Producer Thread
        System.out.println("生产者正在传输元素");
        queue.transfer("Hello, World!");

        System.out.println("生产者已转移元素");
    }
}
```

消费者线程首先启动并等待接收元素。生产者线程调用 transfer 方法将元素直接传输给消费者。

运行结果如下：

```
消费者正在等待获取元素...
生产者正在传输元素
生产者已转移元素
消费者收到：Hello, World!
```

LinkedBlockingDeque

LinkedBlockingDeque 是一个基于链表结构的双端阻塞队列。它同时支持从队列头部插入和移除元素，也支持从队列尾部插入和移除元素。因此，LinkedBlockingDeque 可以作为 FIFO 队列或 LIFO 队列来使用。

常用方法有：

- `addFirst(E e)`, `addLast(E e)`: 在队列的开头/结尾添加元素。
- `takeFirst()`, `takeLast()`: 从队列的开头/结尾移除和返回元素，如果队列为空，则等待。
- `putFirst(E e)`, `putLast(E e)`: 在队列的开头/结尾插入元素，如果队列已满，则等待。
- `pollFirst(long timeout, TimeUnit unit)`, `pollLast(long timeout, TimeUnit unit)`: 在队列的开头/结尾移除和返回元素，如果队列为空，则等待指定的超时时间。

使用示例：

```
public class LinkedBlockingDequeDemo {
    public static void main(String[] args) throws InterruptedException {
        LinkedBlockingDeque<String> deque = new LinkedBlockingDeque<>(10);

        // Adding elements at the end of the deque
        deque.putLast("Item1");
        deque.putLast("Item2");

        // Adding elements at the beginning of the deque
        deque.putFirst("Item3");

        // Removing elements from the beginning
        System.out.println(deque.takeFirst()); // Output: Item3
```

```
        // Removing elements from the end
        System.out.println(deque.takeLast()); // Output: Item2
    }
}
```

运行结果如下：

```
Item3
Item2
```

DelayQueue

DelayQueue 是一个无界阻塞队列，用于存放实现了 Delayed 接口的元素，这些元素只能在其到期时才能从队列中取走。这使得 DelayQueue 成为实现时间基于优先级的调度服务的理想选择。

下面的示例展示了如何使用 DelayQueue。

```
public class DelayQueueDemo {
    public static void main(String[] args) {
        DelayQueue<DelayedElement> queue = new DelayQueue<>();

        // 将带有5秒延迟的元素放入队列
        queue.put(new DelayedElement(5000, "这是一个 5 秒延迟的元素"));

        try {
            System.out.println("取一个元素...");
            // take() 将阻塞，直到延迟到期
            DelayedElement element = queue.take();
            System.out.println(element.getMessage());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class DelayedElement implements Delayed {
        private final long delayUntil;
        private final String message;

        public DelayedElement(long delayInMillis, String message) {
            this.delayUntil = System.currentTimeMillis() + delayInMillis;
            this.message = message;
        }

        public String getMessage() {
            return message;
        }
```

```
        @Override
        public long getDelay(TimeUnit unit) {
            return unit.convert(delayUntil - System.currentTimeMillis(),
TimeUnit.MILLISECONDS);
        }

        @Override
        public int compareTo(Delayed o) {
            return Long.compare(this.delayUntil, ((DelayedElement) o).delayUntil);
        }
    }
}
```

上例创建了一个 DelayQueue，并将一个带有 5 秒延迟的元素放入队列。然后，它调用 `take()` 方法从队列中取出元素。由于元素的延迟时间为 5 秒，因此 `take()` 方法将阻塞 5 秒，直到元素到期。运行结果如下：

```
取一个元素...
这是一个 5 秒延迟的元素
```

小结

本文介绍了 Java 中的阻塞队列，包括 ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue、LinkedTransferQueue、LinkedBlockingDeque 和 DelayQueue。它们都是线程安全的，可以在多线程环境下使用。

阻塞队列是一个非常有用的工具，可以用于实现生产者-消费者模式，或者在多线程环境下进行线程间通信。它们还可以用于实现线程池和其他数据结构，如优先级队列、延迟队列等。

阻塞队列的实现原理是使用 Condition 通知机制，当队列为空时，消费者线程将被阻塞，直到队列中有数据可供消费。当队列已满时，生产者线程将被阻塞，直到队列有可用空间。

阻塞队列是 Java 并发编程中的一个重要概念，它在多线程编程中有着广泛的应用。因此，我们应该熟悉它们的使用方法和实现原理。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency。

第二十三节：并发容器 CopyOnWriteArrayList

学过 ArrayList 的小伙伴应该记得，ArrayList 是一个线程不安全的容器，如果在多线程环境下使用，需要手动加锁，或者使用 `Collections.synchronizedList()` 方法将其转换为线程安全的容器。

否则，将会出现 ConcurrentModificationException 异常。

于是，Doug Lea 大师为我们提供了一个并发版本的 ArrayList——CopyOnWriteArrayList。

CopyOnWriteArrayList 是线程安全的，可以在多线程环境下使用。CopyOnWriteArrayList 遵循写时复制的原则，每当对列表进行修改（例如添加、删除或更改元素）时，都会创建列表的一个新副本，这个新副本会替换旧的列表，而对旧列表的所有读取操作仍然可以继续。

由于在修改时创建了新的副本，所以读取操作不需要锁定。这使得在多读取者和少写入者的情况下读取操作非常高效。当然，由于每次写操作都会创建一个新的数组副本，所以会增加存储和时间的开销。如果写操作非常频繁，性能会受到影响。

什么是 CopyOnWrite

大家应该还记得读写锁 ReentrantReadWriteLock 吧？读写锁是通过读写分离的思想来实现的，即读写锁将读写操作分别加锁，从而实现读写操作的并发执行。

但是，读写锁也存在一些问题，比如说在写锁执行后，读线程会被阻塞，直到写锁被释放后读线程才有机会获取到锁从而读到最新的数据，站在**读线程的角度来看，读线程在任何时候都能获取到最新的数据，满足数据实时性**。

而 CopyOnWriteArrayList 是通过 Copy-On-Write(COW)，即写时复制的思想来通过延时更新的策略实现数据的最终一致性，并且能够保证读线程间不阻塞。当然，**这要牺牲数据的实时性**。

通俗的讲，CopyOnWrite 就是当我们往一个容器添加元素的时候，不直接往容器中添加，而是先复制出一个新的容器，然后在新的容器里添加元素，添加完之后，再将原容器的引用指向新的容器。多个线程在读的时候，不需要加锁，因为当前容器不会添加任何元素。

我们在介绍并发容器的时候，也曾提到过，相信大家都还有印象。

CopyOnWriteArrayList原理

OK，接下来我们来看一下 CopyOnWriteArrayList 的源码。顾名思义，实际上 CopyOnWriteArrayList 内部维护的就是一个数组：

```
/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;
```

该数组被 volatile 修饰，能够保证数据的内存可见性。

get 方法

get 方法的源码如下：

```
public E get(int index) {
    return get(getArray(), index);
}
/**
 * Gets the array.  Non-private so as to also be accessible
 * from CopyOnWriteArraySet class.
 */
final Object[] getArray() {
    return array;
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

get 方法的实现非常简单，几乎就是一个“单线程”，没有添加任何的线程安全控制，没有加锁也没有 CAS 操作，原因就是所有的读线程只会读取容器中的数据，并不会进行修改。

add 方法

add 方法的源码如下：

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    //1. 使用Lock,保证写线程在同一时刻只有一个
    lock.lock();

    try {
        //2. 获取旧数组引用
```

```
            Object[] elements = getArray();
            int len = elements.length;

            //3. 创建新的数组，并将旧数组的数据复制到新数组中
            Object[] newElements = Arrays.copyOf(elements, len + 1);

            //4. 往新数组中添加新的数据
            newElements[len] = e;

            //5. 将旧数组引用指向新的数组
            setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }
```

add 方法的逻辑也比较容易理解，需要注意这么几点：

01、采用 ReentrantLock 保证同一时刻只有一个写线程正在进行数组的复制；

02、通过调用 `getArray()` 方法获取旧的数组。

```
final Object[] getArray() {
    return array;
}
```

03、然后创建一个新的数组，把旧的数组复制过来，然后在新的数组中添加数据，再将新的数组赋值给旧的数组引用。

```
final void setArray(Object[] a) {
    array = a;
}
```

根据 volatile 的 happens-before 规则，所以这个更改对所有线程是立即可见的。

04、最后，在 finally 块中释放锁，以便其他线程可以访问和修改列表。

CopyOnWriteArrayList 的使用

CopyOnWriteArrayList 的使用非常简单，和 ArrayList 的使用几乎一样，只是在创建对象的时候需要使用 CopyOnWriteArrayList 的构造方法，如下所示：

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("element1");
list.add("element2");

for (String element : list) {
    System.out.println(element);
}
```

CopyOnWriteArrayList 的缺点

CopyOnWrite 容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要特别注意。

1. **内存占用问题**：因为 CopyOnWrite 的写时复制机制，在进行写操作的时候，内存里会同时有两个对象，旧的对象和新写入的对象，分析 add 方法的时候大家都看到了。

如果这些对象占用的内存比较大，比如说 200M 左右，那么再写入 100M 数据进去，内存就会占用 600M，那么这时候就会造成频繁的 minor GC 和 major GC。

1. **数据一致性问题**：CopyOnWrite 容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用 CopyOnWrite 容器，最好通过 ReentrantReadWriteLock 自定义一个的列表。

我们来比较一下 CopyOnWrite 和读写锁。

相同点：

1. 两者都是通过读写分离的思想来实现的；
2. 读线程间是互不阻塞的

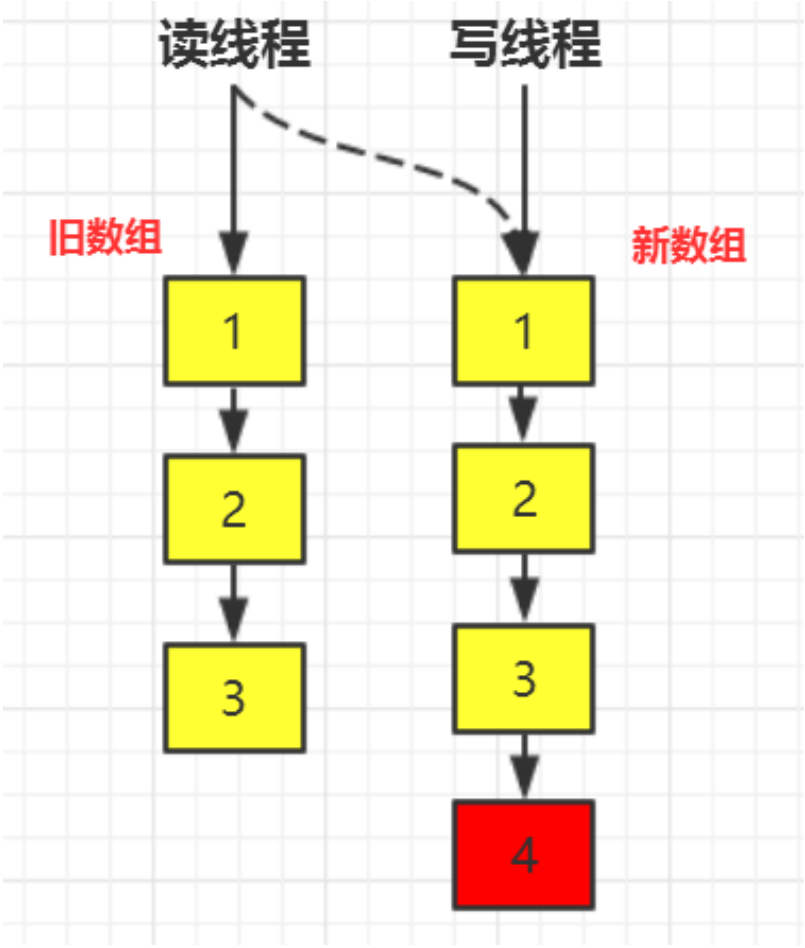
不同点：

为了实现数据实时性，在写锁被获取后，读线程会阻塞；或者当读锁被获取后，写线程会阻塞，从而解决“脏读”的问题。而 CopyOnWrite 对数据的更新是写时复制的，因此读线程是延时感知的，单不会存在阻塞的情况。

对这一点从文字上可能比较难理解，我们通过 debug 来看一下，add 方法核心代码为：

```
1.Object[] elements = getArray();
2.int len = elements.length;
3.Object[] newElements = Arrays.copyOf(elements, len + 1);
4.newElements[len] = e;
5.setArray(newElements);
```

假设 COW 的变化如下图所示：



数组中已有数据 1,2,3，现在写线程想往数组中添加数据 4，我们在第 5 行处打上断点，让写线程暂停。

此时，读线程依然会“不受影响”的从数组中读取数据，可是还是只能读到 1,2,3。

如果读线程能够立即读到新添加的数据就叫数据实时性。当对第 5 行的断点放开后，读线程感知到了数据的变化，所以读到了完整的数据 1,2,3,4，这叫**数据最终一致性**，尽管有可能中间间隔了好几秒才感知到。

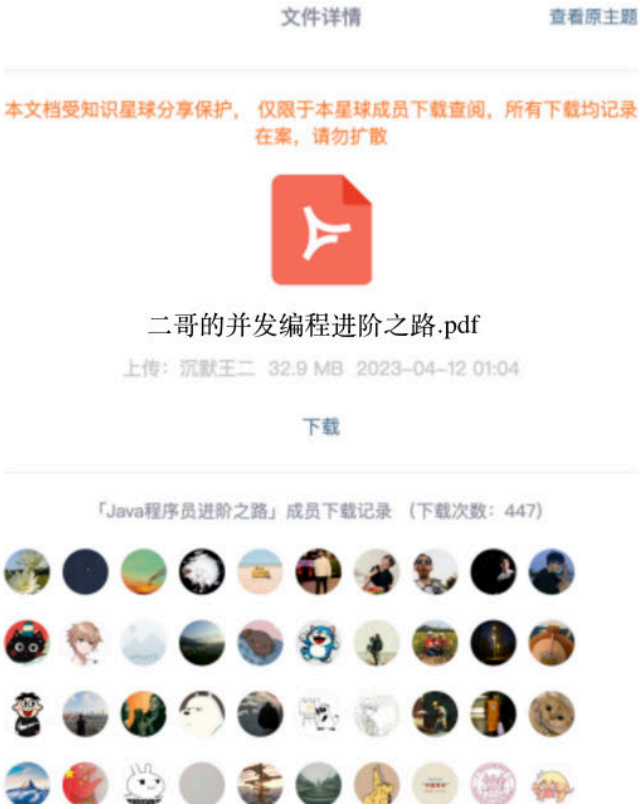
小结

CopyOnWriteArrayList 是一个线程安全的变体，它是 Java 的 ArrayList 类的并发版本。这个类的线程安全是通过一个简单但强大的想法实现的：每当列表修改时，就创建列表的一个新副本。

CopyOnWriteArrayList 适用于读操作远远大于写操作的场景，比如说缓存。因为 CopyOnWriteArrayList 采用写时复制的思想，所以写操作的性能较低，因此不适合写操作频繁的场景。

CopyOnWriteArrayList 也存在一些缺点，比如说内存占用问题和数据一致性问题，所以在开发的时候需要特别注意。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency。



第二十四节：本地变量 ThreadLocal

ThreadLocal 是 Java 中提供的一种用于实现线程局部变量的工具。它允许每个线程都拥有自己的独立副本，从而实现线程隔离。ThreadLocal 可以用于解决多线程中共享对象的线程安全问题。

通常，我们会使用 synchronzed 关键字 或者 lock 来控制线程对临界区资源的同步顺序，但这种加锁的方式会让未获取到锁的线程进行阻塞，很显然，这种方式的时间效率不会特别高。

线程安全问题的核心在于多个线程会对同一个临界区的共享资源进行访问，那如果每个线程都拥有自己的“共享资源”，各用各的，互不影响，这样就不会出现线程安全的问题了，对吧？

事实上，这就是一种“**空间换时间**”的思想，每个线程拥有自己的“共享资源”，虽然内存占用变大了，但由于不需要同步，也就减少了线程可能存在的阻塞问题，从而提高时间上的效率。

不过，ThreadLocal 并不在 java.util.concurrent 并发包下，而是在 java.lang 包下，但我更倾向于把它当作是一种并发容器。

顾名思义，**ThreadLocal 就是线程的“本地变量”，即每个线程都拥有该变量的一个副本，达到人手一份的目的，这样就可以避免共享资源的竞争**。

ThreadLocal 的源码分析

set 方法

set 方法用于设置当前线程中 ThreadLocal 的变量值，该方法的源码如下：

```
public void set(T value) {
  //1. 获取当前线程实例对象
```

```
    Thread t = Thread.currentThread();

  //2. 通过当前线程实例获取到ThreadLocalMap对象
    ThreadLocalMap map = getMap(t);

    if (map != null)
     //3. 如果Map不为null,则以当前ThreadLocal实例为key,值为value进行存入
       map.set(this, value);
    else
    //4.map为null,则新建ThreadLocalMap并存入value
      createMap(t, value);
}
```

- 通过 `Thread.currentThread()` 方法获取当前调用此方法的线程实例。
- 每个线程都有自己的 ThreadLocalMap，这个映射表存储了线程的局部变量，其中键是 ThreadLocal 对象，值为特定于线程的对象。
- 如果 Map 不为 null，则以当前 ThreadLocal 实例为 key，值为 value 进行存入；如果 map 为 null，则新建 ThreadLocalMap 并存入 value。

通过源码我们知道，value 是存放在 ThreadLocalMap 里的。来看下 ThreadLocalMap 是什么，先有个简单的认识，后面会细讲。

ThreadLocalMap 是怎样来的呢？通过 `getMap(t)`：

```
ThreadLocalMap getMap(Thread t) {
    return t.ThreadLocals;
}
```

该方法直接返回当前线程对象 t 的一个成员变量 ThreadLocals：

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap ThreadLocals = null;
```

再来看 set 方法，当 map 为 null 的时候会通过 `createMap(t, value)` 方法 new 出来一个：

```
void createMap(Thread t, T firstValue) {
    t.ThreadLocals = new ThreadLocalMap(this, firstValue);
}
```

该方法 new 了一个 ThreadLocalMap 实例对象，然后以当前 ThreadLocal 实例作为 key，值为 value 存放到 ThreadLocalMap 中，然后将当前线程对象的 ThreadLocals 赋值为 ThreadLocalMap 对象。

set 方法的重要性在于它确保了每个线程都有自己的变量副本。由于这些变量是存储在与线程关联的映射表中的，所以不同的线程之间的这些变量互不影响。

get 方法

get 方法用于获取当前线程中 ThreadLocal 的变量值，同样的还是来看源码：

```
public T get() {
  //1. 获取当前线程的实例对象
  Thread t = Thread.currentThread();

  //2. 获取当前线程的ThreadLocalMap
  ThreadLocalMap map = getMap(t);
  if (map != null) {
  //3. 获取map中当前ThreadLocal实例为key的值的entry
    ThreadLocalMap.Entry e = map.getEntry(this);

    if (e != null) {
      @SuppressWarnings("unchecked")
    //4. 当前entitiy不为null的话，就返回相应的值value
      T result = (T)e.value;
      return result;
    }
  }
  //5. 若map为null或者entry为null的话通过该方法初始化，并返回该方法返回的value
  return setInitialValue();
}
```

代码逻辑请看注释；我们来看下 setInitialValue 主要做了些什么事情?

```
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

该方法的逻辑和 set 方法几乎一样，主要来看下 initialValue 方法:

```
protected T initialValue() {
    return null;
}
```

这个方法是通过 protected 修饰的，也就意味着 ThreadLocal 的子类可以重写该方法给一个合适的初始值。

这里是 initialValue 方法的典型用法：

```
private static ThreadLocal<Integer> myThreadLocal = new ThreadLocal<Integer>() {
    @Override
    protected Integer initialValue() {
        return 0; // 初始值设置为0
    }
};
```

此代码段创建了一个新的 `ThreadLocal<Integer>` 对象，其初始值为 0。任何尝试首次访问此 ThreadLocal 变量的线程都会看到值 0。

整个 setInitialValue 方法的目的是确保每个线程在第一次尝试访问其 ThreadLocal 变量时都有一个合适的值。这种“懒惰”初始化的方法确保了仅在实际需要特定于线程的值时才创建这些值。

remove 方法

```
public void remove() {
  //1. 获取当前线程的ThreadLocalMap
  ThreadLocalMap m = getMap(Thread.currentThread());
  if (m != null)
    //2. 从map中删除以当前ThreadLocal实例为key的键值对
    m.remove(this);
}
```

remove 方法的作用是从当前线程的 ThreadLocalMap 中删除与当前 ThreadLocal 实例关联的条目。这个方法在释放线程局部变量的资源或重置线程局部变量的值时特别有用。

以下是使用 remove 方法的示例代码：

```
ThreadLocal<String> threadLocal = ThreadLocal.withInitial(() -> "Initial Value");

Thread thread = new Thread(() -> {
    System.out.println(threadLocal.get()); // 输出 "Initial Value"
    threadLocal.set("Updated Value");
    System.out.println(threadLocal.get()); // 输出 "Updated Value"
    threadLocal.remove();
    System.out.println(threadLocal.get()); // 输出 "Initial Value"
});
thread.start();
```

输出结果：

```
Initial Value
Updated Value
Initial Value
```

ThreadLocalMap 的源码分析

ThreadLocalMap 是 ThreadLocal 类的静态内部类，它是一个定制的哈希表，专门用于保存每个线程中的线程局部变量。

```
static class ThreadLocalMap {}
```

和大多数容器一样，ThreadLocalMap 内部维护了一个 Entry 类型的数组 类型的数组 table，长度为 2 的幂次方。。

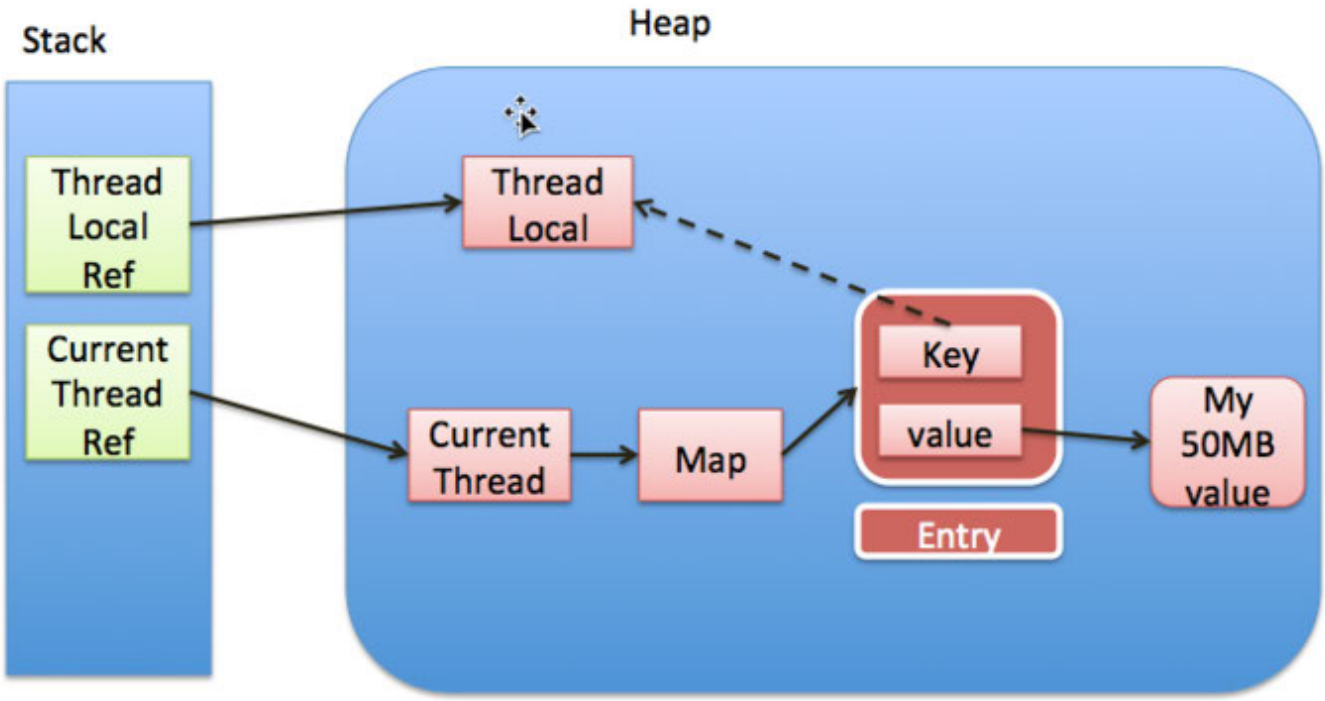
```
/**
 * The table, resized as necessary.
 * table.length MUST always be a power of two.
 */
private Entry[] table;
```

来看下 Entry 是什么：

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

Entry 继承了弱引用 `WeakReference<ThreadLocal<?>>`，它的 value 字段用于存储与特定 ThreadLocal 对象关联的值。使用弱引用作为键允许垃圾收集器在不再需要的情况下回收 ThreadLocal 实例。

这里我们可以用一张图来理解下 Thread、ThreadLocal、ThreadLocalMap、Entry 之间的关系：



上图中的实线表示强引用，虚线表示弱引用。每个线程都可以通过 ThreadLocals 获取到 ThreadLocalMap，而 ThreadLocalMap 实际上就是一个以 ThreadLocal 实例为 key，任意对象为 value 的 Entry 数组。

当我们为 ThreadLocal 变量赋值时，实际上就是以当前 ThreadLocal 实例为 key，值为 Entry 往这个 ThreadLocalMap 中存放。

注意，Entry 的 key 为弱引用，意味着当 ThreadLocal 外部强引用被置为 null（`ThreadLocalInstance=null`）时，根据可达性分析，ThreadLocal 实例此时没有任何一条链路引用它，所以系统 GC 的时候 ThreadLocal 会被回收。

这样一来，ThreadLocalMap 就会出现 key 为 null 的 Entry，也就没办法访问这些 key 对应的 value，如果线程迟迟不结束的话，这些 key 为 null 的 value 就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value，无法回收就会造成内存泄漏。

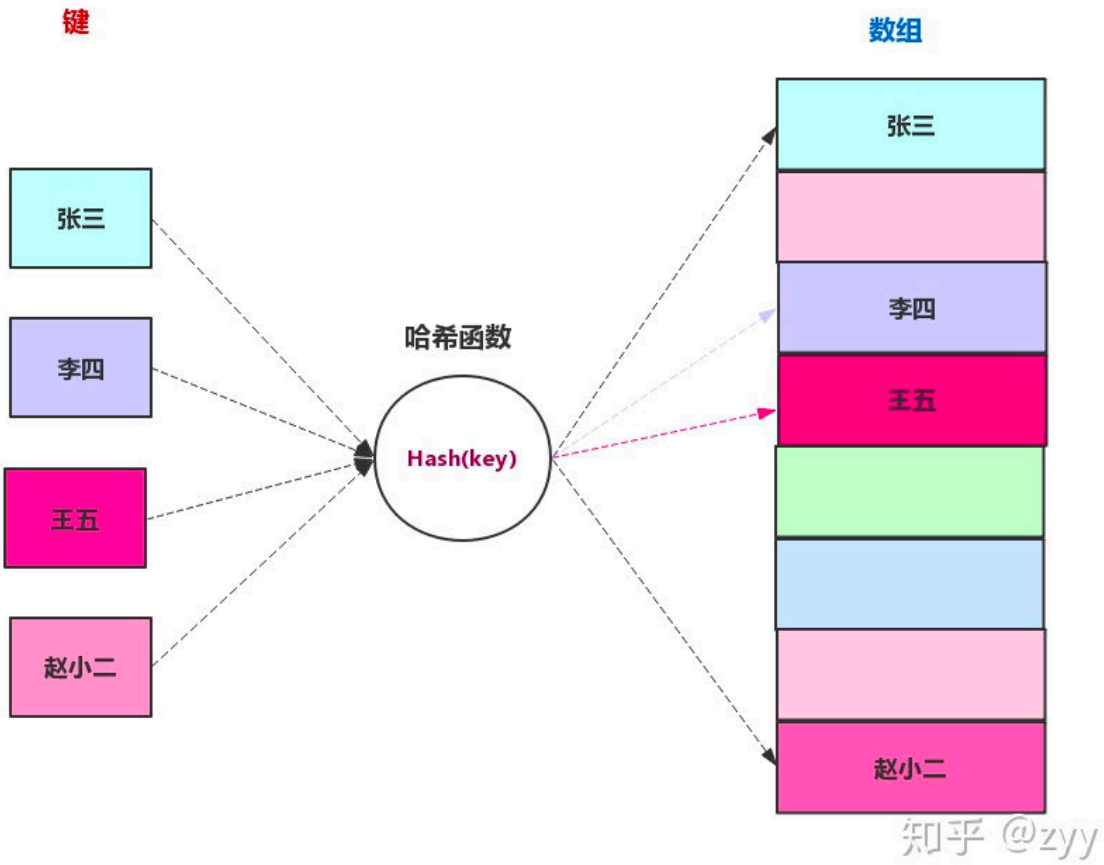
当然，如果 thread 运行结束，ThreadLocal、ThreadLocalMap、Entry 没有引用链可达，在垃圾回收时都会被系统回收。但实际开发中，线程为了复用是不会主动结束的，比如说数据库连接池，过大的线程池可能会增加内存泄漏的风险，因此合理配置线程池的大小和线程的存活时间有助于减轻这个问题。

为了避免这个问题，在每次使用完 ThreadLocal 之后，最好明确调用 ThreadLocal 的 remove 方法来删除与当前线程关联的值。这样可以确保线程再次使用时不会存储旧的、不再需要的值。

与 ConcurrentHashMap、HashMap 等容器一样，ThreadLocalMap 也是通过哈希表实现的。

哈希表

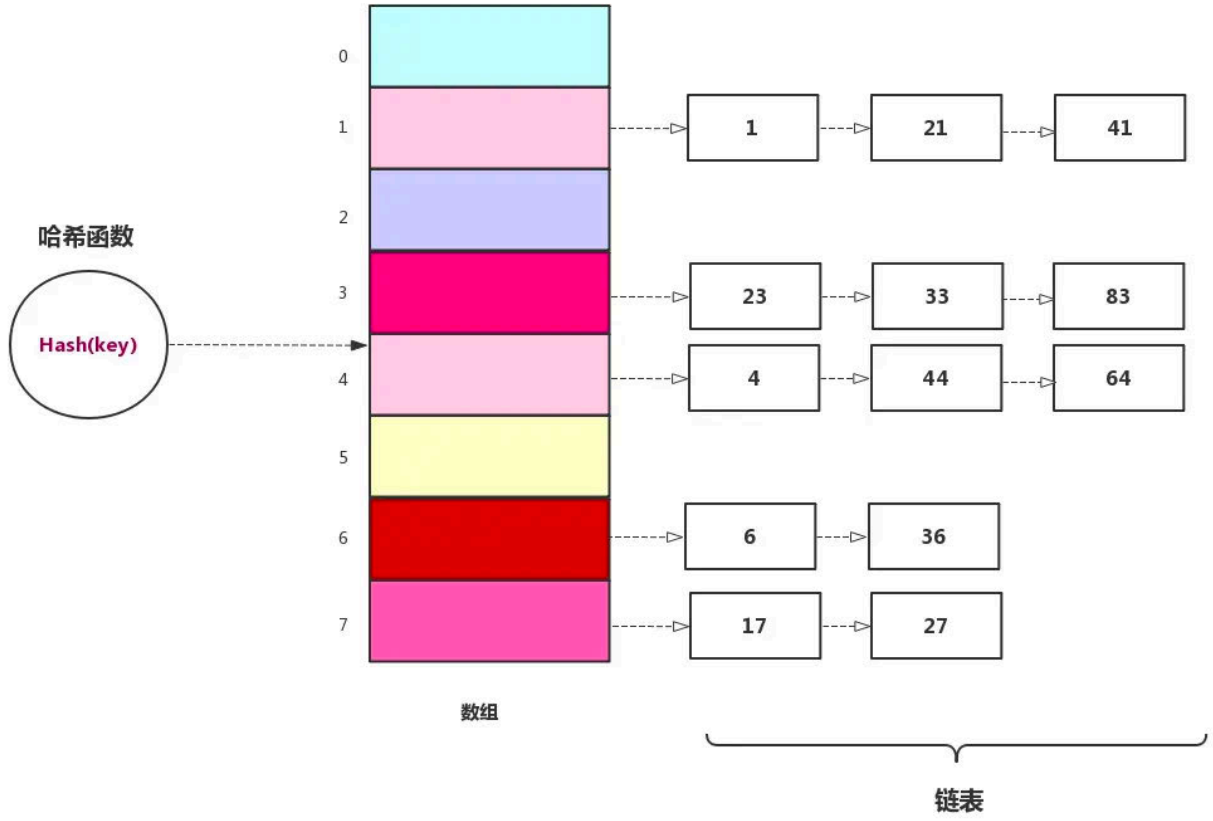
哈希表是基于数组的，每个数组元素被称为一个“桶”（Bucket），桶中存储了键值对（Key-Value Pair），键是通过哈希函数生成的，理想的哈希函数可以均匀分布键，从而最大限度地减少冲突。



理想的哈希函数可以均匀分布键，从而最大限度地减少冲突。当两个或多个键的哈希值相同（即映射到同一个桶）时，称之为哈希冲突。常见的解决策略有拉链法和开放地址法。

拉链法

在讲 HashMap 的时候，我们详细讲过拉链法，相信大家都还有印象，我们这里简单回顾一下：当某项关键字通过哈希后落到哈希表中的某个位置，把该条数据添加到链表中，其他同样映射到这个位置的数据项也只需要添加到链表中。下面是示意图：

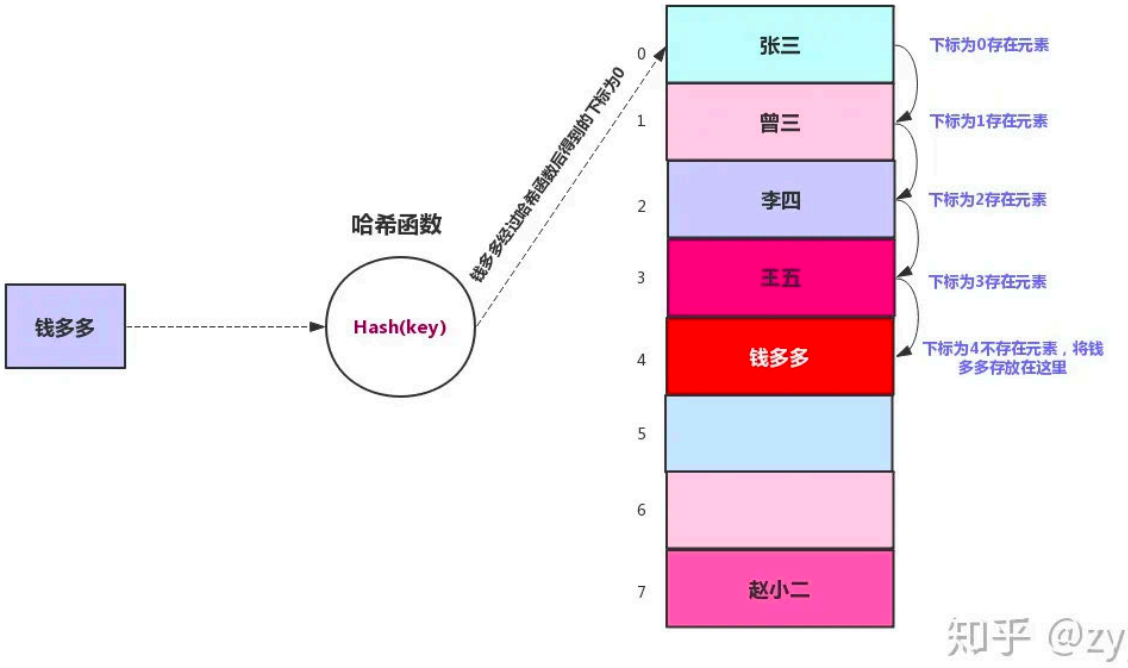


开放地址法

开放地址法中，若数据不能直接存放在哈希函数计算出来的数组下标时，就需要寻找其他位置来存放。在开放地址法中有三种方式来寻找其他的位置，分别是「线性探测」、「二次探测」、「再哈希法」。

01、线性探测：当哈希函数计算出来的数组下标已经被占用时，就顺序往后查找，直到找到一个空闲的位置。

例如我们将数88经过哈希函数后得到的数组下标是16，但是在数组下标为16的地方已经存在元素，那么就找17，17还存在元素就找18，一直往下找，直到找到空白地方存放元素。我们来看下面这张图：

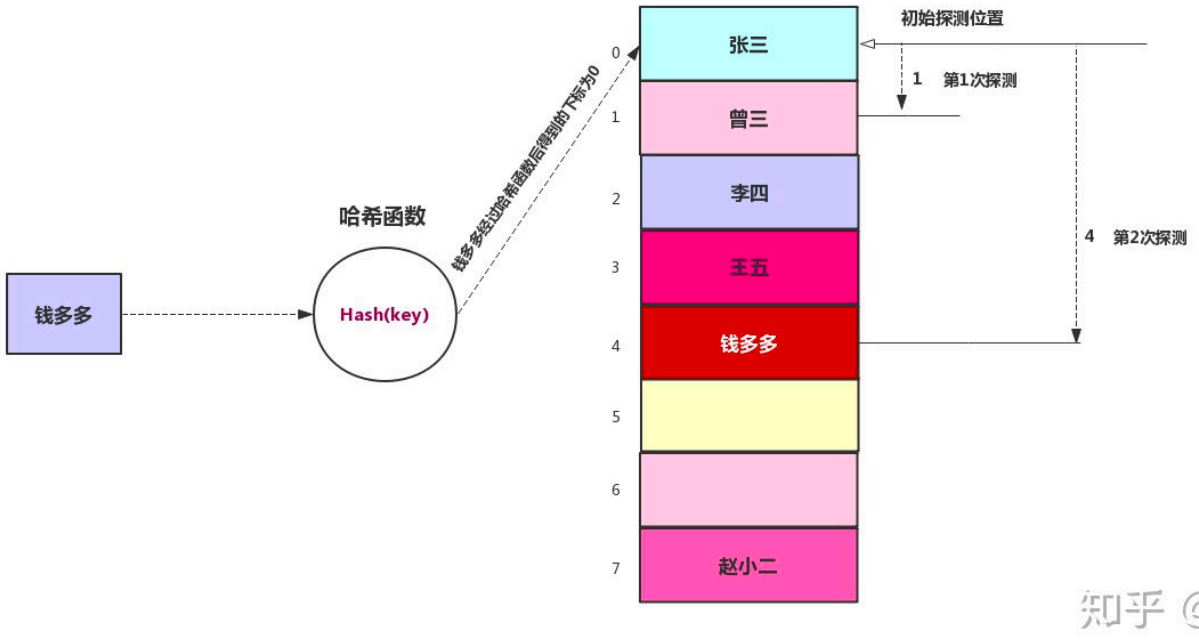


我们向哈希表中添加一个元素钱多多，钱多多经过哈希函数后得到的数组下标为0，但是在0的位置已经有张三了，所以下标往前移，直到下标4才为空，所以就将元素钱多多添加到数组下标为4的地方。

02、二次探测：当哈希函数计算出来的数组下标已经被占用时，就顺序往后查找，直到找到一个空闲的位置。不同的是，二次探测是按照某种规律查找，而不是顺序查找，比如说每次查找的步长是 1，2，4，8，16.....

在线性探测哈希表中，数据会发生聚集，一旦聚集形成，它就会变的越来越大，那些哈希函数后落在聚集范围内的数据项，都需要一步一步往后移动，并且插入到聚集的后面，因此聚集变的越大，聚集增长的越快。这个就像我们在逛超市一样，当某个地方人很多时，人只会越来越多，大家都只是想知道这里在干什么。

二次探测是防止聚集产生的一种尝试，思想是探测相隔较远的单元，而不是和原始位置相邻的单元。在线性探测中，如果哈希函数得到的原始下标是x，线性探测就是x+1,x+2,x+3.....，以此类推，而在二次探测中，探测过程是x+1,x+4,x+9,x+16,x+25.....,以此类推，到原始距离的步数平方，为了方便理解，我们来看下面这张图。



在线性探测中我们找到钱多多的存储位置需要经过4步。在二次探测中，每次是原始距离步数的平方，所以我们只需要两次就找到钱多多的存储位置。

03、再哈希法：当哈希函数计算出来的数组下标已经被占用时，就使用另一个哈希函数计算出来的数组下标。

二次探测消除了线性探测的聚集问题，这种聚集问题叫做原始聚集，然而，二次探测也产生了新的聚集问题，之所以会产生新的聚集问题，是因为所有映射到同一位置的关键字在寻找空位时，探测的位置都是一样的。

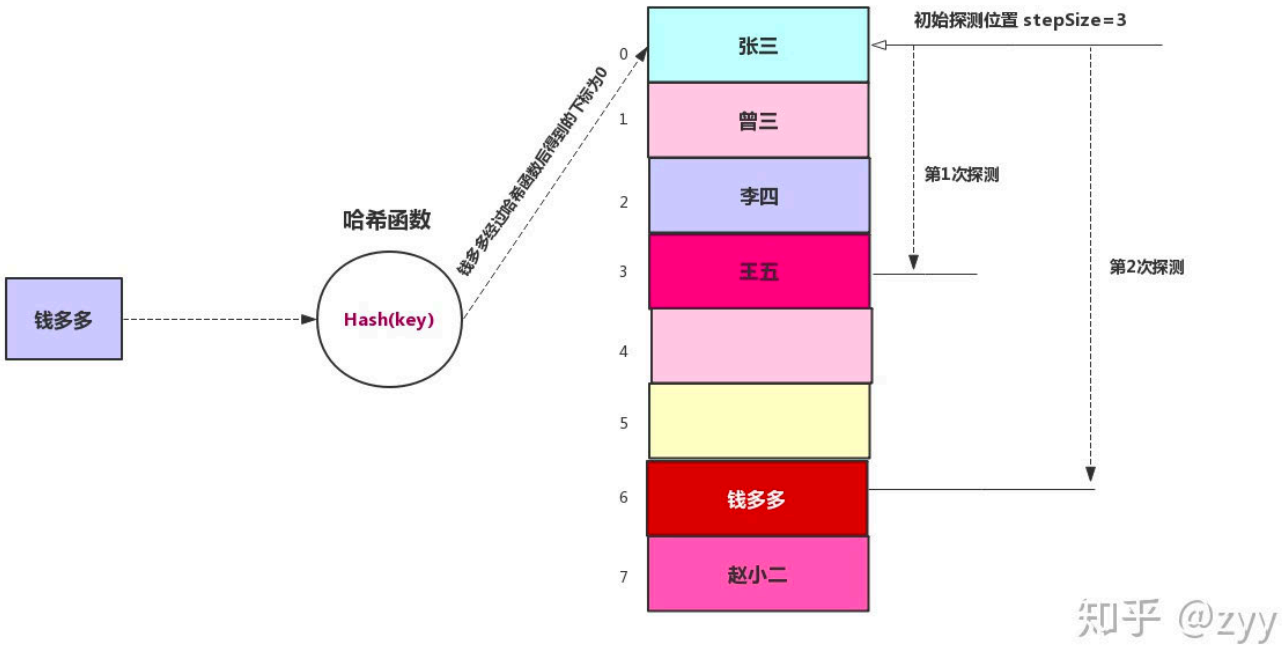
比如讲1、11、21、31、41依次插入到哈希表中，它们映射的位置都是1，那么11需要以一为步长探测，21需要以四为步长探测，31需要为九为步长探测，41需要以十六为步长探测，只要有一项映射到1的位置，就需要更长的步长来探测，这个现象叫做二次聚集。

再哈希法是为了消除原始聚集和二次聚集问题，不管是线性探测还是二次探测，每次的探测步长都是固定的。双哈希是除了第一个哈希函数外再增加一个哈希函数用来根据关键字生成探测步长，这样即使第一个哈希函数映射到了数组的同一下标，但是探测步长不一样，这样就能够解决聚集的问题。

第二个哈希函数必须具备如下特点：

- 和第一个哈希函数不一样
- 不能输出为 0，因为步长为 0，每次探测都是指向同一个位置，将进入死循环，经过试验得出 `stepSize = constant-(key%constant);` 形式的哈希函数效果非常好，constant是一个质数并且小于数组容量。

示意图如下：



ThreadLocalMap 是使用开放地址法来处理哈希冲突的，和 HashMap 不同，之所以采用不同的方式主要是因为：

ThreadLocalMap 中的哈希值分散的比较均匀，很少会出现冲突。并且 ThreadLocalMap 经常需要清除无用的对象，冲突的概率就更小了。

set 方法

好，在了解哈希表的相关知识后，我们再来看一下 set 方法。set 方法的源码如下：

```
private void set(ThreadLocal<?> key, Object value) {

    // We don't use a fast path as with get() because it is at
    // least as common to use set() to create new entries as
    // it is to replace existing ones, in which case, a fast
    // path would fail more often than not.

    Entry[] tab = table;
    int len = tab.length;
  //根据ThreadLocal的hashCode确定Entry应该存放的位置
    int i = key.ThreadLocalHashCode & (len-1);

  //采用开放地址法，hash冲突的时候使用线性探测
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();
```

```
    //覆盖旧Entry
        if (k == key) {
            e.value = value;
            return;
        }
    //当key为null时，说明ThreadLocal强引用已经被释放掉，那么就无法
    //再通过这个key获取ThreadLocalMap中对应的entry，这里就存在内存泄漏的可能性
        if (k == null) {
      //用当前插入的值替换掉这个key为null的“脏”entry
            replaceStaleEntry(key, value, i);
            return;
        }
    }
  //新建entry并插入table中i处
    tab[i] = new Entry(key, value);
    int sz = ++size;
  //插入后再次清除一些key为null的“脏”entry,如果大于阈值就需要扩容
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

set 方法的关键部分**请看注释**，这里有几点需要注意：

01、ThreadLocal 的 hashcode

```
private final int ThreadLocalHashCode = nextHashCode();
private static final int HASH_INCREMENT = 0x61c88647;
private static AtomicInteger nextHashCode =new AtomicInteger();
  /**
   * Returns the next hash code.
   */
  private static int nextHashCode() {
      return nextHashCode.getAndAdd(HASH_INCREMENT);
  }
```

ThreadLocal 的 hashCode 是通过 `nextHashCode()` 方法获取的，该方法实际上是用 AtomicInteger 加上 0x61c88647 来实现的。

0x61c88647 是一个魔数，用于 ThreadLocal 的哈希码递增。这个值的选择并不是随机的，它是一个质数，具有以下特性：

- 质数：它是一个质数，这意味着它不能被除 1 和它本身之外的任何数字整除。
- 黄金比例：这个数字大约等于黄金比例的 32 位浮点表示的一半。黄金比例具有一些有趣的数学特性，其中之一是与斐波那契数列的关系。
- 递增分布：在 ThreadLocal 中，这个数字用于在哈希表中分散不同线程的哈希码，从而减少冲突。每当创建新的 ThreadLocal 对象时，都会将此值添加到上一个 ThreadLocal 的哈希码中。这个递增的步长有助于在哈希表中均匀地分配 ThreadLocal 对象。
- 性能优化：通过使用这个特定的值，算法能够确保哈希码的均匀分布，从而减少哈希冲突的可能性。这对于哈希表的性能至关重要，因为冲突可能会降低查找的效率。

02、怎样确定新值插入的位置?

通过这行代码：`key.ThreadLocalHashCode & (len-1)`。

同 HashMap 一样，通过当前 key 的 hashcode 与哈希表大小相与。原理我们在 HashMap 的时候已经讲过了，不记得的小伙伴可以回去看一遍。

03、怎样解决 hash 冲突?

通过 `nextIndex(i, len)`，该方法中的`((i + 1 < len) ? i + 1 : 0);`能不断往后线性探测，当到哈希表末尾的时候再从 0 开始，成环形。

04、怎样解决“脏”Entry?

我们知道，使用 ThreadLocal 有可能存在内存泄漏的问题，针对这种 key 为 null 的 Entry，我们称之为“stale entry”，直译为不新鲜的 entry，我把它理解为“脏 entry”。

当然了，Josh Bloch 和 Doug Lea 已经替我们考虑了这种情况，源码中提供了这些解决方案：

在向ThreadLocalMap添加新条目时，可以检查是否有“脏”Entry（键为null的Entry），并用新的条目替换它。这就是源码中的replaceStaleEntry方法所做的事情。

```
将在设置操作期间遇到的陈旧条目替换为指定键的条目。无论指定键是否已存在条目，都将存储在
value参数中传递的值。作为一种副作用，此方法会清除包含陈旧条目的“运行”中的所有陈旧条目。
（运行是两个空槽之间的条目序列。）
参数： key - 键
      value - 与键相关联的值
      staleSlot - 在查找键时遇到的第一个陈旧条目的索引。
private void replaceStaleEntry(ThreadLocal<?> key, Object value,
                               int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;
    Entry e;

    // Back up to check for prior stale entry in current run.
    // We clean out whole runs at a time to avoid continual
    // incremental rehashing due to garbage collector freeing
    // up refs in bunches (i.e., whenever the collector runs).
    int slotToExpunge = staleSlot;
    for (int i = prevIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = prevIndex(i, len))
        if (e.get() == null)
```

在某些操作过程中（例如添加、获取等），可以增加额外的清理操作来扫描并移除“脏”Entry。这可以通过遍历哈希表，并删除那些键为null的条目来实现。源码中的cleanSomeSlots方法就是这样一个例子。

```
按照启发式算法扫描一些单元格，寻找过时的条目。当添加新元素或删除其他过时的元素时，会调用此函数。它执行对数次数的扫描，以在不扫描（快速但保留垃圾）和扫描与元素数量成比例的次数之间取得平衡。后者将找到所有的垃圾，但会导致一些插入操作需要 O(n) 的时间。
参数：  i – 一个已知不包含过时条目的位置。扫描从 i 后面的元素开始。
        n – 扫描控制：扫描 log2(n) 个单元格，除非找到一个过时的条目，在这种情况下，还会额外扫描 log2(table.length)-1 个单元格。当从插入操作调用时，此参数是元素的数量，而当从替换过时条目调用时，此参数是表的长度。（注意：通过调整 n 的权重而不仅仅是使用直接的对数 n，可以更改这一切的策略变得更加积极或保守。但是这个版本简单、快速，看起来运行良好。）
返回值：  如果已经删除了任何过时的条目，则为 true。
private boolean cleanSomeSlots(int i, int n) {  Complexity is 8 It's time to do
    boolean removed = false;
    Entry[] tab = table;
    int len = tab.length;
    do {
        i = nextIndex(i, len);
        Entry e = tab[i];
        if (e != null && e.get() == null) {
            n = len;
            removed = true;
            i = expungeStaleEntry(i);
        }
    } while ( (n >>>= 1) != 0);
    return removed;
}
```

05、如何进行扩容?

和 HashMap 一样，ThreadLocalMap 也有扩容机制，那么它的 threshold 又是怎样确定的呢?

```
private int threshold; // Default to 0
/**
 * The initial capacity -- MUST be a power of two.
 */
private static final int INITIAL_CAPACITY = 16;

ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    table = new Entry[INITIAL_CAPACITY];
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}

/**
 * Set the resize threshold to maintain at worst a 2/3 load factor.
 */
```

```
private void setThreshold(int len) {
    threshold = len * 2 / 3;
}
```

在第一次对 ThreadLocal 赋值的时候会创建初始大小为 16 的 ThreadLocalMap，并且通过 setThreshold 方法设置 threshold，其值为当前哈希数组长度乘以（2/3），也就是说加载因子为 2/3。

加载因子（Load Factor）是哈希表的一个重要概念，它表示哈希表中已经存放的条目数量与哈希表容量的比例。加载因子可以用来衡量哈希表的满载程度，影响哈希表的查找、插入和删除操作的性能。相信大家都还记得，HashMap 的加载因子都为 0.75。

这里**ThreadLocalMap 初始大小为 16，加载因子为 2/3**，所以哈希表可用大小为：16*2/3=10，即哈希表可用容量为 10。

当哈希表的 size 大于 threshold 的时候，会通过 resize 方法进行扩容。

```
/**
 * Double the capacity of the table.
 */
private void resize() {
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
  //新数组为原数组的2倍
    int newLen = oldLen * 2;
    Entry[] newTab = new Entry[newLen];
    int count = 0;

    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
      //遍历过程中如果遇到脏entry的话直接另value为null,有助于value能够被回收
            if (k == null) {
                e.value = null; // Help the GC
            } else {
        //重新确定entry在新数组的位置，然后进行插入
                int h = k.ThreadLocalHashCode & (newLen - 1);
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                newTab[h] = e;
                count++;
            }
        }
    }
  //设置新哈希表的threshHold和size属性
    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

方法逻辑**请看注释**，新建的数组为原来数组长度的两倍，然后遍历旧数组中的 entry 并将其插入到新的数组中。注意，这段代码考虑得非常周全，**在扩容的过程中，针对脏 entry 会把 value 设为 null，以便被垃圾回收，解决隐藏的内存泄漏问题**。

getEntry 方法

getEntry 方法的源码如下：

```
private Entry getEntry(ThreadLocal<?> key) {
  //1. 确定在哈希数组中的位置
    int i = key.ThreadLocalHashCode & (table.length - 1);
  //2. 根据索引i获取entry
    Entry e = table[i];
  //3. 满足条件则返回该entry
    if (e != null && e.get() == key)
        return e;
    else
    //4. 未查找到满足条件的entry，额外在做的处理
        return getEntryAfterMiss(key, i, e);
}
```

方法的逻辑很简单，如果当前 entry 的 key 和查找的 key 相同就直接返回这个 entry，否则的就通过 getEntryAfterMiss 做进一步处理：如果索引处的条目为null，或者其键与给定的键不匹配，那么需要调用 getEntryAfterMiss方法来处理可能的哈希冲突。

getEntryAfterMiss 方法如下：

```
private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
    Entry[] tab = table;
    int len = tab.length;

    while (e != null) {
        ThreadLocal<?> k = e.get();
        if (k == key)
      //找到和查询的key相同的entry则返回
            return e;
        if (k == null)
      //解决脏entry的问题
            expungeStaleEntry(i);
        else
      //继续向后环形查找
            i = nextIndex(i, len);
        e = tab[i];
    }
    return null;
}
```

getEntryAfterMiss 方法用于在发生哈希冲突的情况下继续在ThreadLocalMap中查找条目，通过开放寻址的策略，在哈希表中的其他位置查找，并适当地处理“脏”条目。

remove 方法

直接来看源码：

```
/**
 * Remove the entry for key.
 */
private void remove(ThreadLocal<?> key) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.ThreadLocalHashCode & (len-1);
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        if (e.get() == key) {
      //将entry的key置为null
            e.clear();
      //将该entry的value也置为null
            expungeStaleEntry(i);
            return;
        }
    }
}
```

01、通过局部变量tab获取ThreadLocalMap的哈希表数组，len表示其长度。

02、通过`key.ThreadLocalHashCode & (len-1)`计算给定ThreadLocal键的哈希索引。这将决定从哪个索引位置开始搜索。

03、使用开放寻址法遍历哈希表，通过`nextIndex(i, len)`计算下一个索引以处理哈希冲突。

04、如果找到与给定键匹配的条目（即`e.get() == key`），执行以下操作：

- 清除键：通过调用`e.clear()`方法，将条目的键置为null。由于Entry是WeakReference的子类，clear方法将断开对ThreadLocal对象的引用，允许垃圾收集器在需要时回收它。
- 清除值：通过调用`expungeStaleEntry(i)`方法，清除该条目的值并对哈希表进行部分清理。该方法的目的是清除哈希表中的无效条目，即那些其键已被垃圾收集的条目。

05、结束删除操作：一旦找到并删除了匹配的条目，方法返回。如果遍历整个哈希表都没有找到匹配的键，则该方法不执行任何操作并正常返回。

ThreadLocal 的使用场景

ThreadLocal 的使用场景非常多，比如说：

- 用于保存用户登录信息，这样在同一个线程中的任何地方都可以获取到登录信息。
- 用于保存数据库连接、Session 对象等，这样在同一个线程中的任何地方都可以获取到数据库连接、Session 对象等。
- 用于保存事务上下文，这样在同一个线程中的任何地方都可以获取到事务上下文。
- 用于保存线程中的变量，这样在同一个线程中的任何地方都可以获取到线程中的变量。

下面是一个使用ThreadLocal来保存用户登录信息的示例。这个示例适用于像Web服务器这样的多线程环境，其中每个线程处理一个独立的用户请求。

```
public class UserAuthenticationService {

    // 创建一个ThreadLocal实例，用于保存用户登录信息
    private static ThreadLocal<User> currentUser = ThreadLocal.withInitial(() -> null);

    public static void main(String[] args) {
        // 模拟用户登录
        loginUser(new User("Alice", "password123"));
        System.out.println("User logged in: " + getCurrentUser().getUsername());

        // 模拟另一个线程处理另一个用户
        Runnable task = () -> {
            loginUser(new User("Bob", "password456"));
            System.out.println("User logged in: " + getCurrentUser().getUsername());
        };

        Thread thread = new Thread(task);
        thread.start();
    }

    // 模拟用户登录方法
    public static void loginUser(User user) {
        // 这里通常会有一些身份验证逻辑
        currentUser.set(user);
    }

    // 获取当前线程关联的用户信息
    public static User getCurrentUser() {
        return currentUser.get();
    }

    // 用户类
    public static class User {
        private final String username;
        private final String password;

        public User(String username, String password) {
            this.username = username;
            this.password = password;
        }

        public String getUsername() {
            return username;
        }

        // 其他getter和setter...
```

```
    }
}
```

这个示例定义了一个UserAuthenticationService类，该类使用ThreadLocal来保存与当前线程关联的用户登录信息。假设用户已经通过身份验证，将用户对象存储在currentUser ThreadLocal变量中。getCurrentUser方法用于检索与当前线程关联的用户信息。由于使用了ThreadLocal，因此不同的线程可以同时登录不同的用户，而不会相互干扰。

小结

ThreadLocal 是一个非常有用的工具类，它可以用于保存线程中的变量，这样在同一个线程中的任何地方都可以获取到线程中的变量。但是，ThreadLocal 也是一个非常容易被误用的工具类，如果没有使用好，就可能会造成内存泄漏的问题。

ThreadLocalMap 是 ThreadLocal 的核心，它是一个以 ThreadLocal 实例为 key，任意对象为 value 的哈希表。ThreadLocalMap 使用开放地址法来处理哈希冲突，它的初始容量为 16，加载因子为 2/3，扩容时会将容量扩大为原来的两倍。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency，部分图片和内容来资源知乎这篇帖子。

第二十五节：线程池

好，终于到 Java 的线程池了，这是 Java 并发编程中非常重要的一块内容，今天我们就通过图文的方式来彻底弄懂线程池的工作原理，以及在实际项目中该如何自定义适合业务的线程池。

一、什么是线程池

线程池其实是一种池化的技术实现，池化技术的核心思想就是实现资源的复用，避免资源的重复创建和销毁带来的性能开销。线程池可以管理一堆线程，让线程执行完任务之后不进行销毁，而是继续去处理其它线程已经提交的任务。

使用线程池的好处

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

二、线程池的构造

Java 主要是通过构建 ThreadPoolExecutor 来创建线程池的。接下来我们看一下线程池是如何构造出来的

ThreadPoolExecutor 的构造方法：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

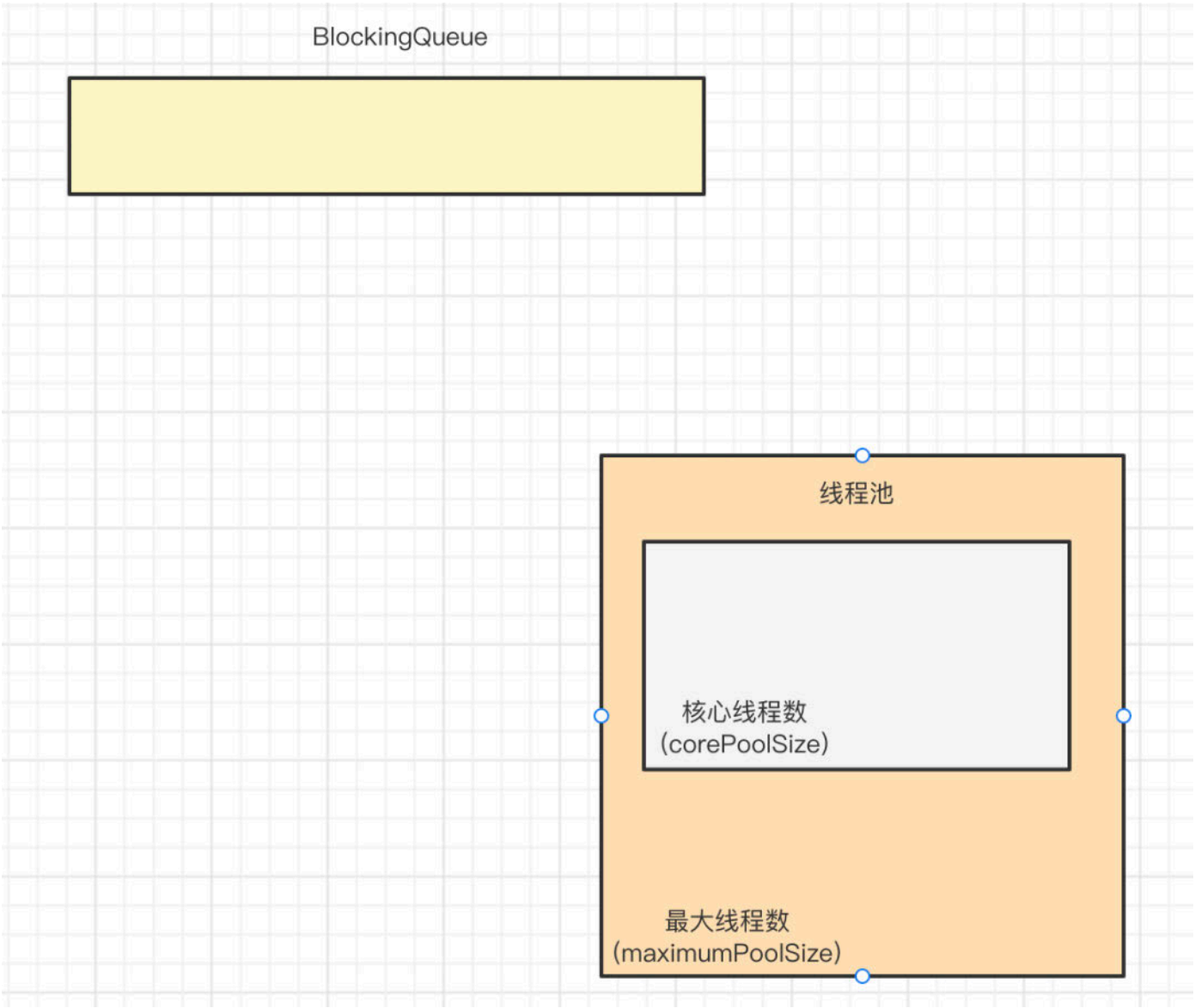
- corePoolSize：线程池中用来工作的核心线程数量。
- maximumPoolSize：最大线程数，线程池允许创建的最大线程数。
- keepAliveTime：超出 corePoolSize 后创建的线程存活时间或者是所有线程最大存活时间，取决于配置。
- unit：keepAliveTime 的时间单位。
- workQueue：任务队列，是一个阻塞队列，当线程数达到核心线程数后，会将任务存储在阻塞队列中。
- threadFactory ：线程池内部创建线程所用的工厂。
- handler：拒绝策略；当队列已满并且线程数量达到最大线程数量时，会调用该方法处理任务。

线程池的构造其实很简单，就是传入一堆参数，然后进行简单的赋值操作。

三、线程池的运行原理

说完线程池的核心构造参数，接下来来讲解这些参数在线程池中是如何工作的。

线程池刚创建出来是什么样子呢，如下图：



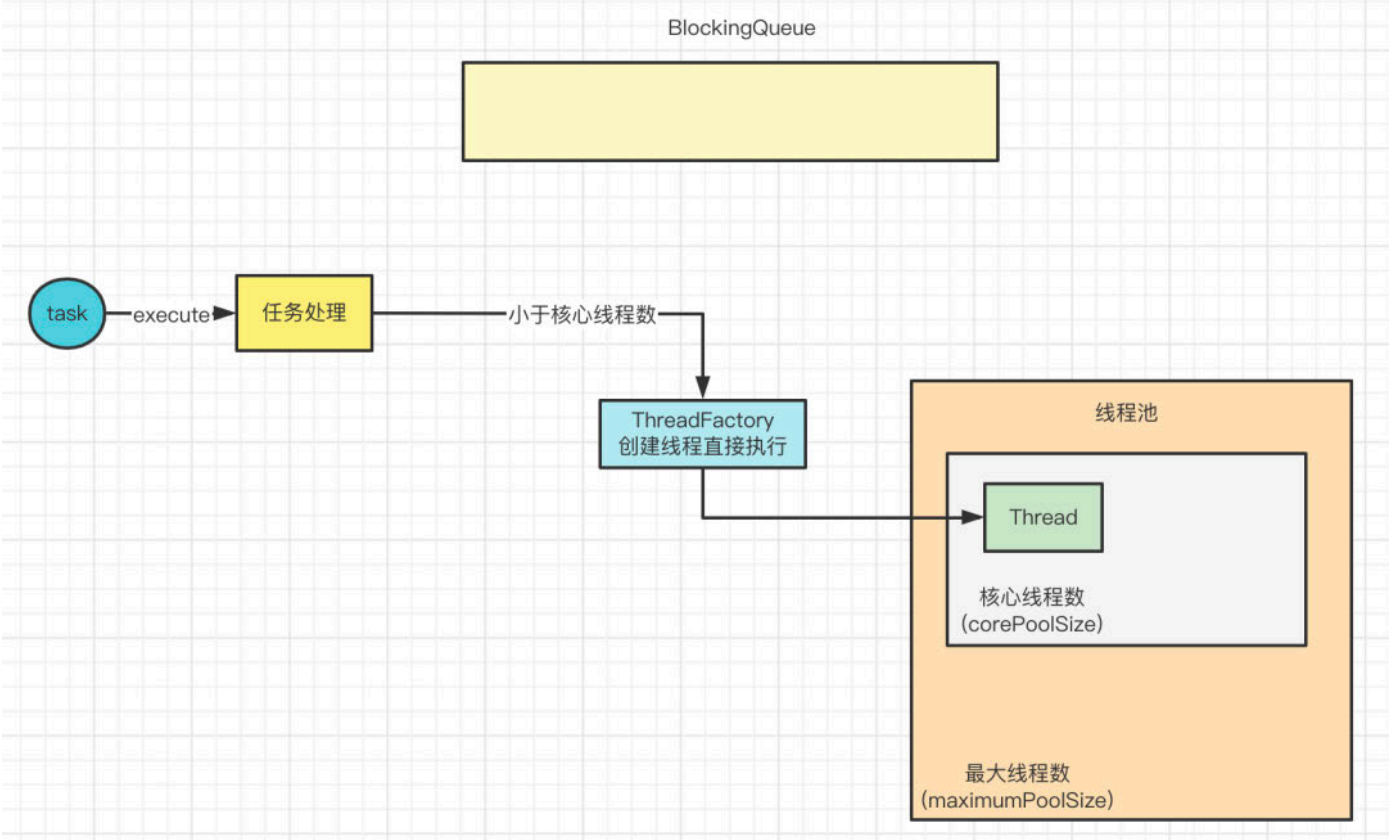
没错，刚创建出来的线程池中只有一个构造时传入的阻塞队列，里面并没有线程，如果想要在执行之前创建好核心线程数，可以调用 prestartAllCoreThreads 方法来实现，默认是没有线程的。

```
启动所有核心线程，使它们空闲等待任务。这会覆盖默认的策略，只有在执行新任务时才启动核心线程。
返回： 启动的线程数
public int prestartAllCoreThreads() {  Complexity is 3 Everything is cool!
    int n = 0;
    while (addWorker( firstTask: null,  core: true))
        ++n;
    return n;
}
```

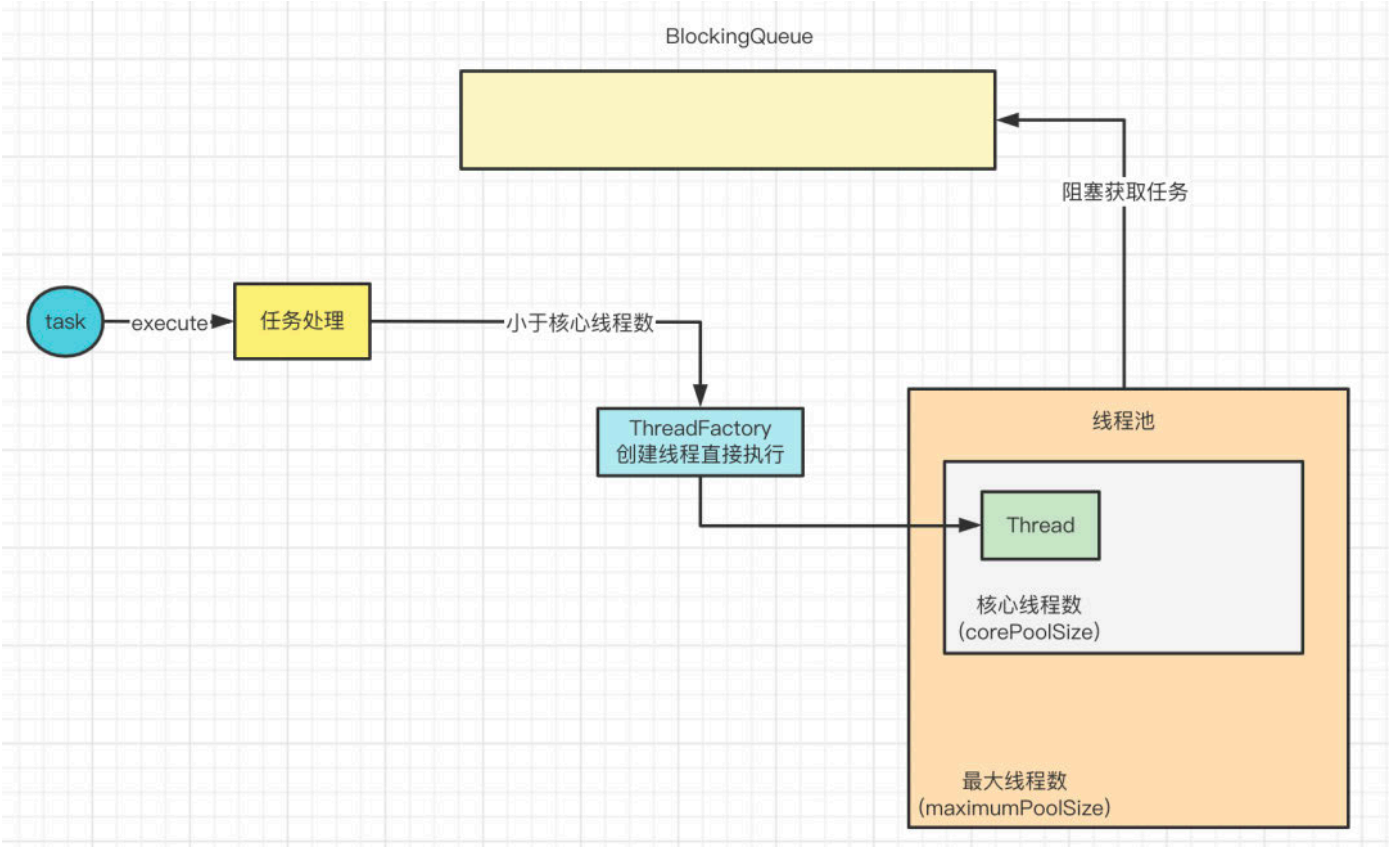
当有线程通过 execute 方法提交了一个任务，会发生什么呢?

首先会去判断当前线程池的线程数是否小于核心线程数，也就是线程池构造时传入的参数 corePoolSize。

如果小于，那么就直接通过 ThreadFactory 创建一个线程来执行这个任务，如图



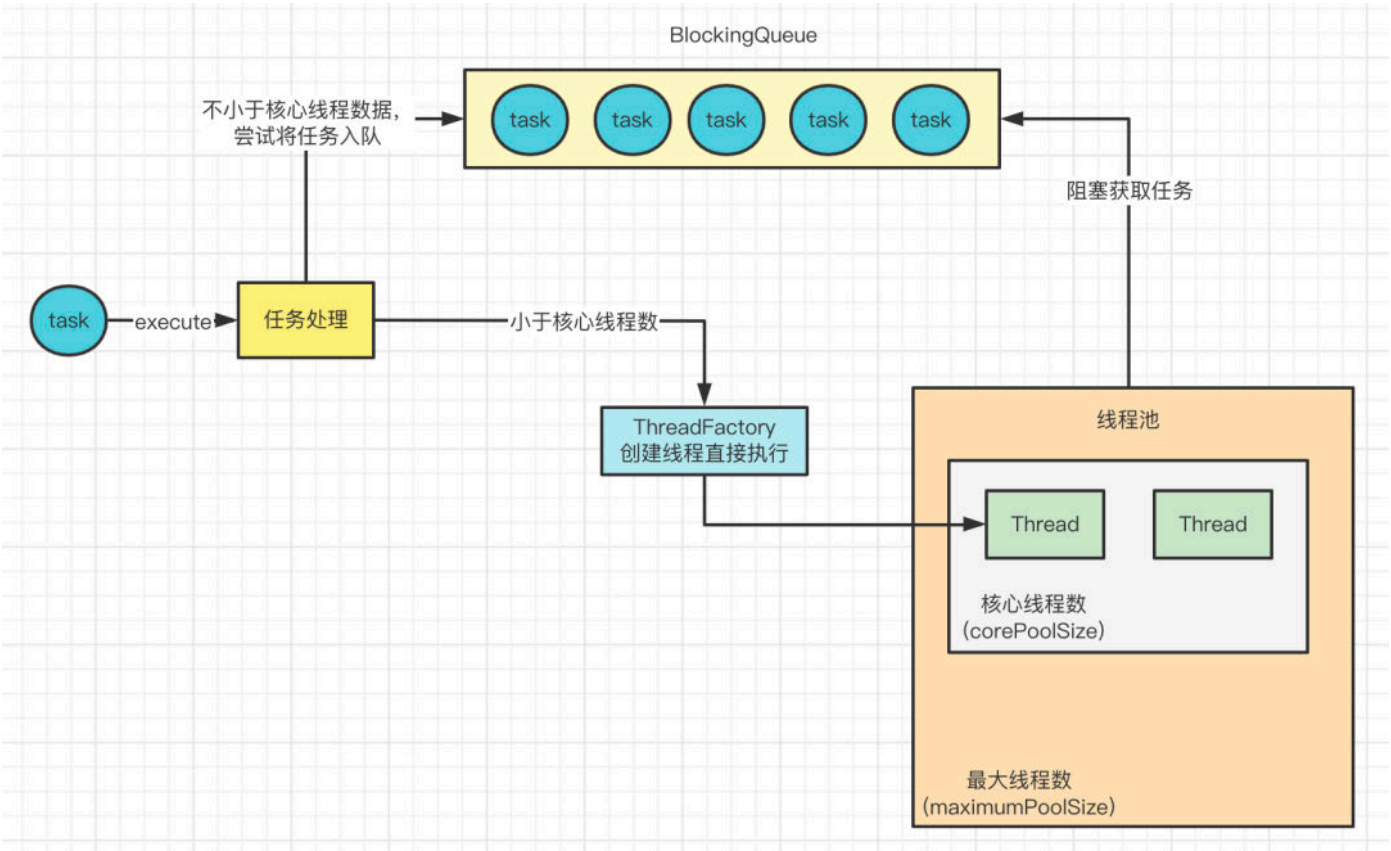
当任务执行完之后，线程不会退出，而是会去阻塞队列中获取任务，如下图



接下来如果又提交了一个任务，也会按照上述的步骤去判断是否小于核心线程数，如果小于，还是会创建线程来执行任务，执行完之后也会从阻塞队列中获取任务。

这里有个细节，就是提交任务的时候，就算有线程池里的线程从阻塞队列中获取不到任务，如果线程池里的线程数还是小于核心线程数，那么依然会继续创建线程，而不是复用已有的线程。

如果线程池里的线程数不再小于核心线程数呢？那么此时就会尝试将任务放入阻塞队列中，入队成功之后，如图

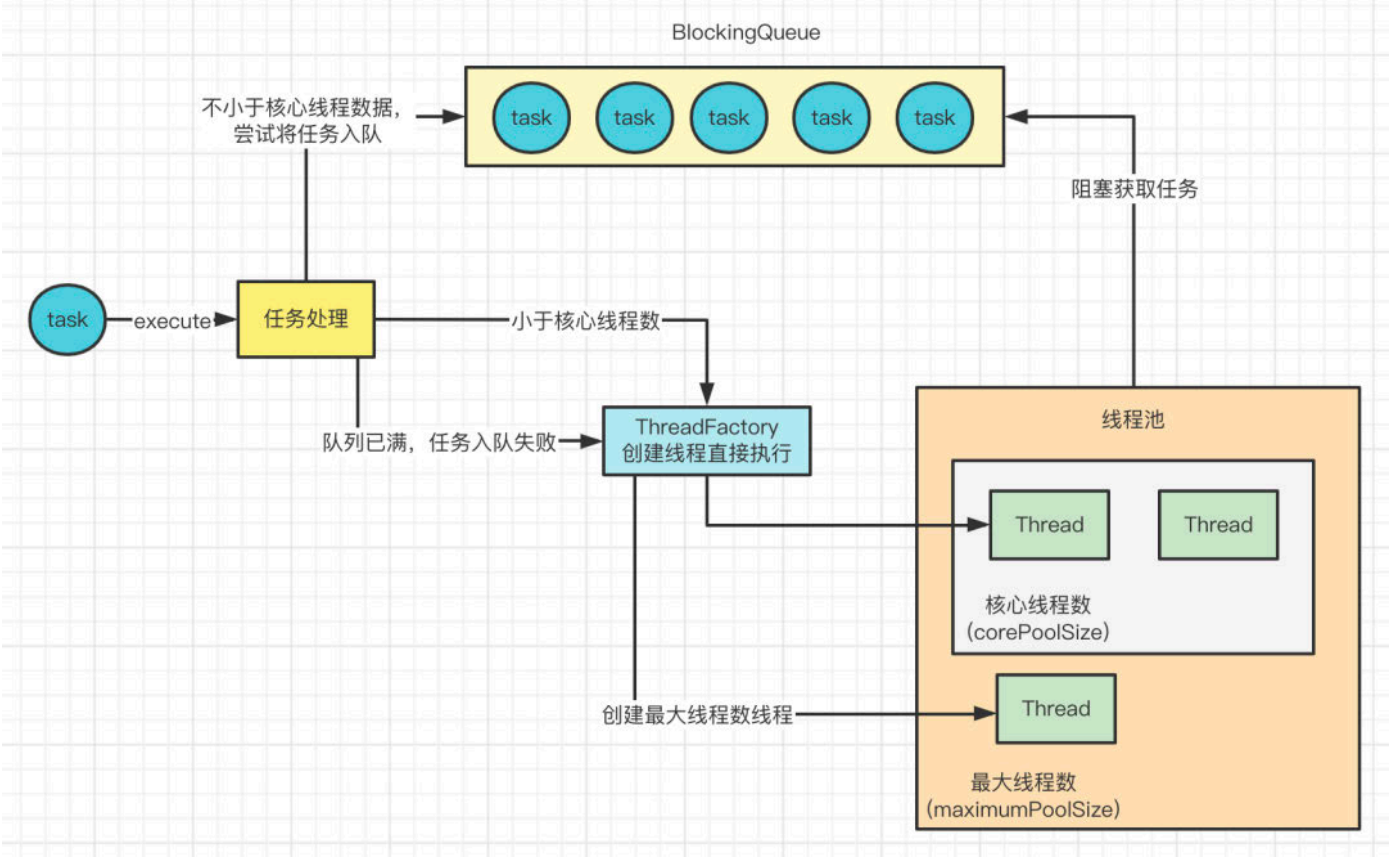


这样，阻塞的线程就可以获取到任务了。

但是，随着任务越来越多，队列已经满了，任务放入失败，怎么办呢?

此时会判断当前线程池里的线程数是否小于最大线程数，也就是入参时的 maximumPoolSize 参数

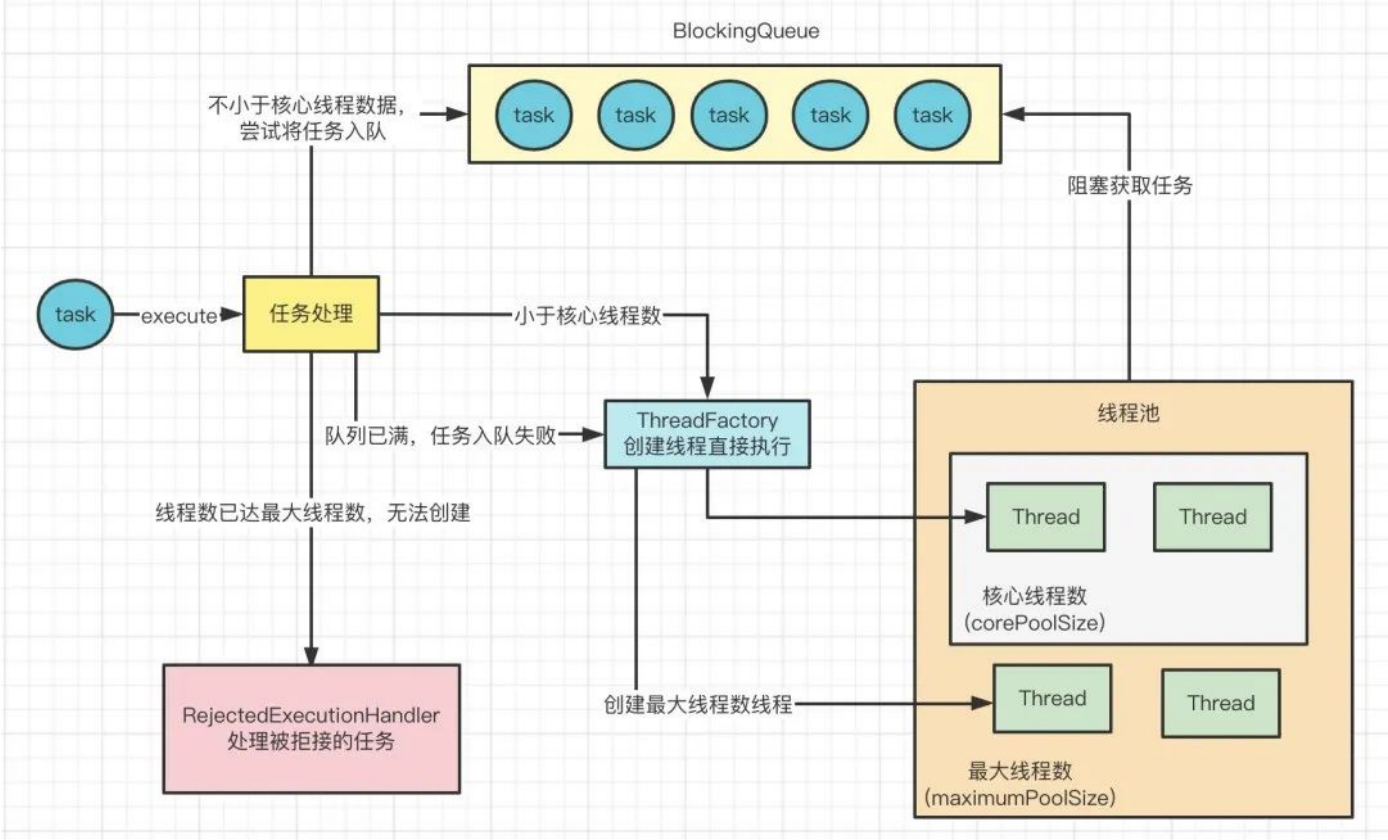
如果小于最大线程数，那么也会创建非核心线程来执行提交的任务，如图



所以，就算队列中有任务，新创建的线程还是会优先处理这个提交的任务，而不是从队列中获取已有的任务执行，**从这可以看出，先提交的任务不一定先执行。**

假如线程数已经达到最大线程数量，怎么办呢?

此时就会执行拒绝策略，也就是构造线程池的时候，传入的 RejectedExecutionHandler 对象，来处理这个任务。



JDK 自带的 RejectedExecutionHandler 实现有 4 种

- AbortPolicy：丢弃任务，抛出运行时异常
- CallerRunsPolicy：由提交任务的线程来执行任务
- DiscardPolicy：丢弃这个任务，但是不抛异常
- DiscardOldestPolicy：从队列中剔除最先进入队列的任务，然后再次提交任务

线程池创建的时候，如果不指定拒绝策略就默认是 AbortPolicy 策略。

当然，你也可以自己实现 RejectedExecutionHandler 接口，比如将任务存在数据库或者缓存中，这样就可以从数据库或者缓存中获取被拒绝掉的任务了。

到这里，我们发现，线程池构造的几个参数 corePoolSize、maximumPoolSize、workQueue、threadFactory、handler 我们都在上述的执行过程中讲到了，那么还差两个参数 keepAliveTime 和 unit（unit 是 keepAliveTime 的时间单位）没讲到，所以 keepAliveTime 是如何起作用的呢，这个问题留到后面分析。

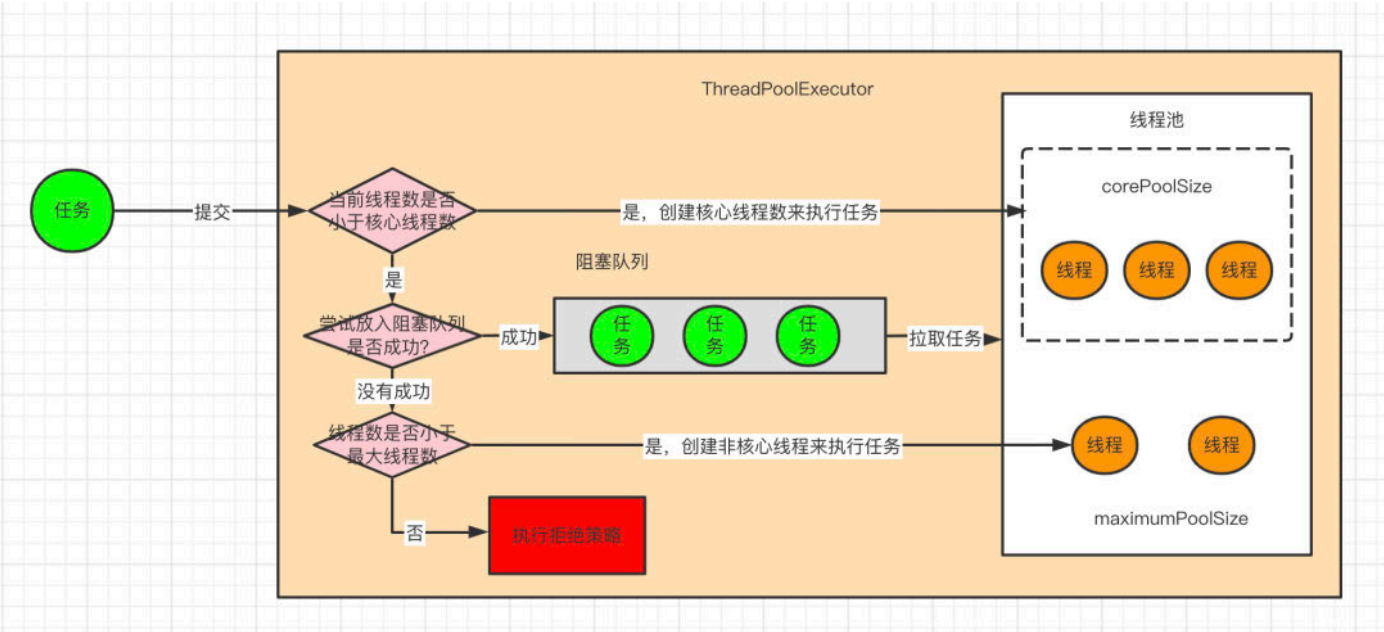
说完整个执行的流程，接下来看看 execute 方法的代码是如何实现的。

```
public void execute(Runnable command) {
    // 首先检查提交的任务是否为null，是的话则抛出NullPointerException。
    if (command == null)
        throw new NullPointerException();

    // 获取线程池的当前状态（ctl是一个AtomicInteger，其中包含了线程池状态和工作线程数）
    int c = ctl.get();

    // 1. 检查当前运行的工作线程数是否少于核心线程数（corePoolSize）
    if (workerCountOf(c) < corePoolSize) {
        // 如果少于核心线程数，尝试添加一个新的工作线程来执行提交的任务
```

```
            // addWorker方法会检查线程池状态和工作线程数，并决定是否真的添加新线程
            if (addWorker(command, true))
                return;
            // 重新获取线程池的状态，因为在尝试添加线程的过程中线程池的状态可能已经发生变化
            c = ctl.get();
        }

        // 2. 尝试将任务添加到任务队列中
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            // 双重检查线程池的状态
            if (! isRunning(recheck) && remove(command))  // 如果线程池已经停止，从队列中移除任务
                reject(command);
            // 如果线程池正在运行，但是工作线程数为0，尝试添加一个新的工作线程
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 3. 如果任务队列满了，尝试添加一个新的非核心工作线程来执行任务
        else if (!addWorker(command, false))
            // 如果无法添加新的工作线程（可能因为线程池已经停止或者达到最大线程数限制），则拒绝任务
            reject(command);
    }
```

- `workerCountOf(c)<corePoolSize`：判断是否小于核心线程数，是的话就通过 addWorker 方法，addWorker 用来添加线程并执行任务。
- `workQueue.offer(command)`：尝试往阻塞队列中添加任务。添加失败就会再次调用 addWorker 尝试添加非核心线程来执行任务；如果还是失败了，就会调用 `reject(command)` 来拒绝这个任务。

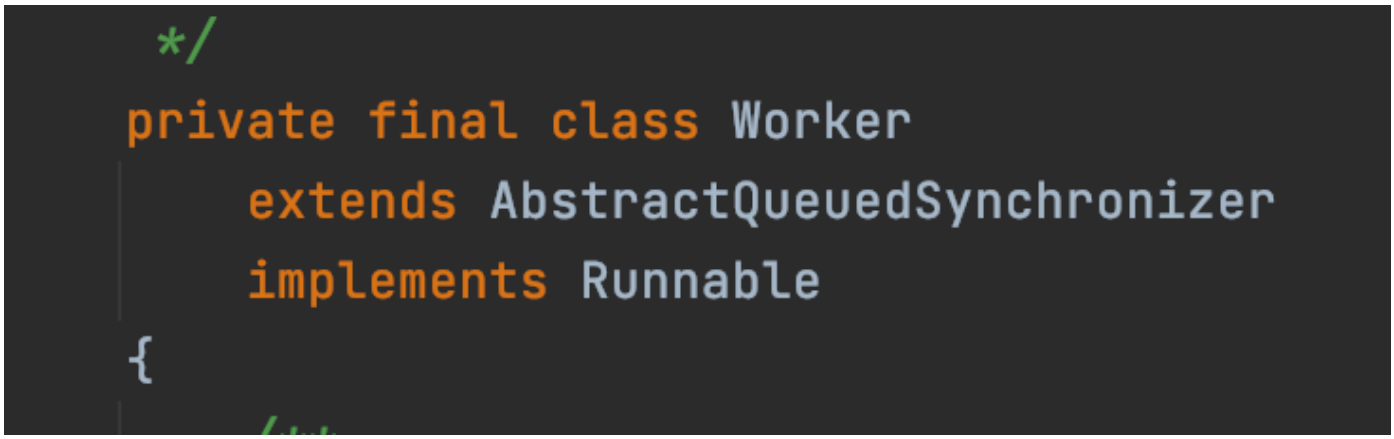
再来另画一张图总结一下 execute 的执行流程



四、线程池中线程实现复用的原理

线程池的核心功能就是实现线程的重复利用，那么线程池是如何实现线程的复用呢?

线程在线程池内部其实被封装成了一个 Worker 对象



Worker 继承了 AQS，也就是具有一定锁的特性。

创建线程来执行任务的方法，上面提到了，是通过 addWorker 方法。在创建 Worker 对象的时候，会把线程和任务一起封装到 Worker 内部，然后调用 runWorker 方法来让线程执行任务，接下来我们就来看一下 runWorker 方法。

```
final void runWorker(Worker w) {
    // 获取当前工作线程
    Thread wt = Thread.currentThread();

    // 从 Worker 中取出第一个任务
    Runnable task = w.firstTask;
    w.firstTask = null;

    // 解锁 Worker（允许中断）
    w.unlock();

    boolean completedAbruptly = true;
    try {
        // 当有任务需要执行或者能够从任务队列中获取到任务时，工作线程就会持续运行
        while (task != null || (task = getTask()) != null) {
            // 锁定 Worker，确保在执行任务期间不会被其他线程干扰
            w.lock();

            // 如果线程池正在停止，并确保线程已经中断
            // 如果线程没有中断并且线程池已经达到停止状态，中断线程
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();

            try {
                // 在执行任务之前，可以插入一些自定义的操作
                beforeExecute(wt, task);
```

```
                Throwable thrown = null;
                try {
                    // 实际执行任务
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    // 执行任务后，可以插入一些自定义的操作
                    afterExecute(task, thrown);
                }
            } finally {
                // 清空任务，并更新完成任务的计数
                task = null;
                w.completedTasks++;
                // 解锁 Worker
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        // 工作线程退出的后续处理
        processWorkerExit(w, completedAbruptly);
    }
}
```

从这里就可以找出线程执行完任务不会退出的原因了，runWorker 内部使用了 while 死循环，当第一个任务执行完之后，会不断地通过 getTask 方法获取任务，只要能获取到任务，就会调用 run 方法继续执行任务，这就是线程能够复用的主要原因。

但是如果从 getTask 获取不到方法的话，就会调用 finally 中的 processWorkerExit 方法，将线程退出。

这里有个一个细节就是，因为 Worker 继承了 AQS，每次在执行任务之前都会调用 Worker 的 lock 方法，执行完任务之后，会调用 unlock 方法，这样做的目的就可以通过 Woker 的加锁状态判断出当前线程是否正在执行任务。

如果想知道线程是否正在执行任务，只需要调用 Woker 的 tryLock 方法，根据是否加锁成功就能判断，加锁成功说明当前线程没有加锁，也就没有执行任务了，在调用 shutdown 方法关闭线程池的时候，就时用这种方式来判断线程有没有在执行任务，如果没有的话，会尝试打断没有执行任务的线程。

五、线程是如何获取任务以及如何实现超时的

前面我们讲到，线程在执行完任务之后，会继续从 getTask 方法中获取任务，获取不到就会退出。接下来我们就来看一看 getTask 方法的实现。

```
private Runnable getTask() {
    // 标志，表示最后一个poll()操作是否超时
    boolean timedOut = false;
```

```
    // 无限循环，直到获取到任务或决定工作线程应该退出
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // 如果线程池状态是SHUTDOWN或更高（如STOP）并且任务队列为空，那么工作线程应该减少并退出
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // 检查工作线程是否应当在没有任务执行时，经过keepAliveTime之后被终止
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        // 如果工作线程数超出最大线程数或者超出核心线程数且上一次poll()超时，并且队列为空或工作线程数
大于1，
        // 则尝试减少工作线程数
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            // 根据timed标志，决定是无限期等待任务，还是等待keepAliveTime时间
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :  // 指定时间内等待
                workQueue.take();  // 无限期等待
            if (r != null)  // 成功获取到任务
                return r;
            // 如果poll()超时，则设置timedOut标志
            timedOut = true;
        } catch (InterruptedException retry) {
            // 如果在等待任务时线程被中断，重置timedOut标志并重新尝试获取任务
            timedOut = false;
        }
    }
}
```

前面就是线程池的一些状态判断，这里有一行代码

```
boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
```

这行代码是用来判断当前过来获取任务的线程是否可以超时退出。如果 allowCoreThreadTimeOut 设置为 true 或者线程池当前的线程数大于核心线程数，也就是 corePoolSize，那么该获取任务的线程就可以超时退出。

怎么做到超时退出呢，就是这行核心代码

```
Runnable r = timed ?
workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
workQueue.take();
```

会根据是否允许超时来选择调用阻塞队列 workQueue 的 poll 方法或者 take 方法。如果允许超时，则调用 poll 方法，传入 keepAliveTime，也就是构造线程池时传入的空闲时间，这个方法的意思就是从队列中阻塞 keepAliveTime 时间来获取任务，获取不到就会返回 null；如果不允许超时，就会调用 take 方法，这个方法会一直阻塞获取任务，直到从队列中获取到任务为止。

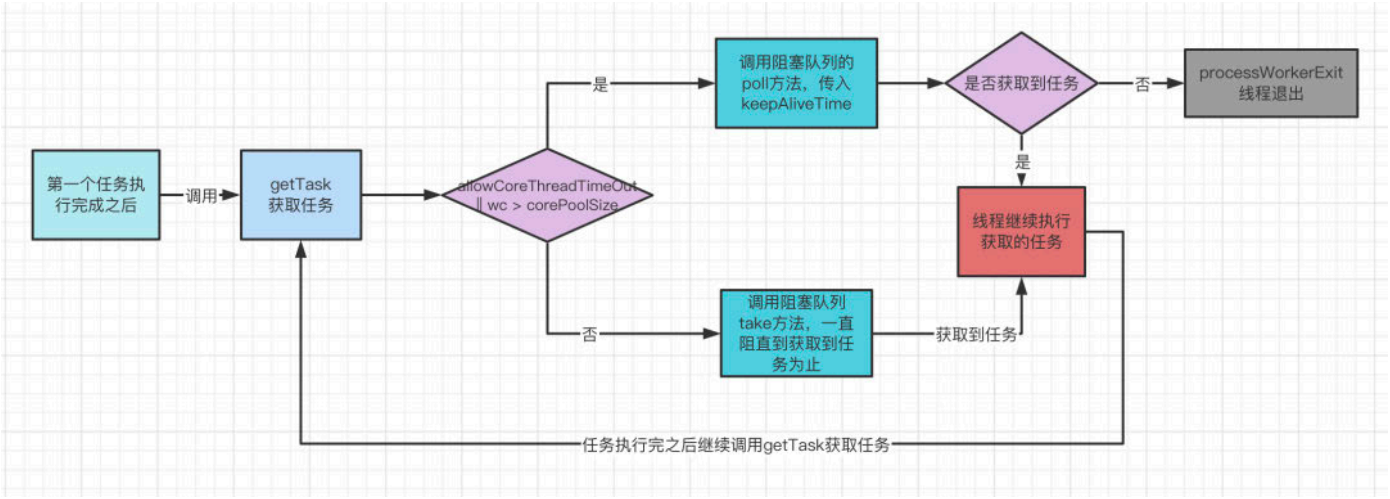
从这里就可以看到 keepAliveTime 是如何使用的了。

所以到这里，大家应该知道线程池中的线程为什么可以做到空闲一定时间就退出了吧?

其实最主要就是利用了阻塞队列的 poll 方法，这个方法可以指定超时时间，一旦线程达到了 keepAliveTime 还没有获取到任务，就会返回 null，一旦 getTask 方法返回 null，线程就会退出。

这里也有一个细节，就是判断当前获取任务的线程是否可以超时退出的时候，如果将 allowCoreThreadTimeOut 设置为 true，那么所有线程走到这个 timed 都是 true，所有线程包括核心线程都可以做到超时退出。如果线程池需要将核心线程超时退出，就可以通过 allowCoreThreadTimeOut 方法将 allowCoreThreadTimeOut 变量设置为 true。

整个 getTask 方法以及线程超时退出的机制如图所示



六、线程池的 5 种状态

线程池内部有 5 个常量来代表线程池的五种状态

```
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;
```

- RUNNING：线程池创建时就是这个状态，能够接收新任务，以及对已添加的任务进行处理。
- SHUTDOWN：调用 shutdown 方法，线程池就会转换成 SHUTDOWN 状态，此时线程池不再接收新任务，

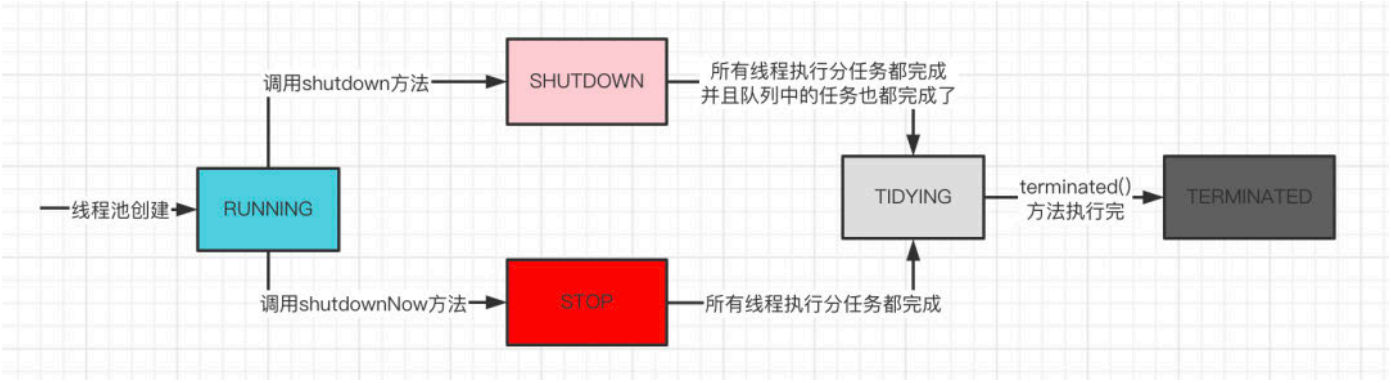
但能继续处理已添加的任务到队列中。

- STOP：调用 shutdownNow 方法，线程池就会转换成 STOP 状态，不接收新任务，也不能继续处理已添加的任务到队列中任务，并且会尝试中断正在处理的任务的线程。
- TIDYING：SHUTDOWN 状态下，任务数为 0， 其他所有任务已终止，线程池会变为 TIDYING 状态；线程池在 SHUTDOWN 状态，任务队列为空且执行中任务为空，线程池会变为 TIDYING 状态；线程池在 STOP 状态，线程池中执行中任务为空时，线程池会变为 TIDYING 状态。
- TERMINATED：线程池彻底终止。线程池在 TIDYING 状态执行完 `terminated()` 方法就会转变为 TERMINATED 状态。

线程池状态具体是存在 ctl 成员变量中的，ctl 中不仅存储了线程池的状态还存储了当前线程池中线程数的大小

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

最后画个图来总结一下这 5 种状态的流转



其实，在线程池运行过程中，绝大多数操作执行前都得判断当前线程池处于哪种状态，再来决定是否继续执行该操作。

七、线程池的关闭

线程池提供了 shutdown 和 shutdownNow 两个方法来关闭线程池。

shutdown 方法

```
/**
 * 启动一次顺序关闭，在这次关闭中，执行器不再接受新任务，但会继续处理队列中的已存在任务。
 * 当所有任务都完成后，线程池中的线程会逐渐退出。
 */
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock; // ThreadPoolExecutor的主锁
    mainLock.lock(); // 加锁以确保独占访问

    try {
        checkShutdownAccess(); // 检查是否有关闭的权限
        advanceRunState(SHUTDOWN); // 将执行器的状态更新为SHUTDOWN
        interruptIdleWorkers(); // 中断所有闲置的工作线程
        onShutdown(); // ScheduledThreadPoolExecutor中的挂钩方法，可供子类重写以进行额外操作
    } finally {
        mainLock.unlock(); // 无论try块如何退出都要释放锁
```

```
    }

    tryTerminate(); // 如果条件允许，尝试终止执行器
}
```

就是将线程池的状态修改为 SHUTDOWN，然后尝试打断空闲的线程（如何判断空闲，上面在说 Worker 继承 AQS 的时候说过），也就是在阻塞等待任务的线程。

shutdownNow 方法

```
/**
 * 尝试停止所有正在执行的任务，停止处理等待的任务，
 * 并返回等待处理的任务列表。
 *
 * @return 从未开始执行的任务列表
 */
public List<Runnable> shutdownNow() {
    List<Runnable> tasks; // 用于存储未执行的任务的列表
    final ReentrantLock mainLock = this.mainLock; // ThreadPoolExecutor的主锁
    mainLock.lock(); // 加锁以确保独占访问

    try {
        checkShutdownAccess(); // 检查是否有关闭的权限
        advanceRunState(STOP); // 将执行器的状态更新为STOP
        interruptWorkers(); // 中断所有工作线程
        tasks = drainQueue(); // 清空队列并将结果放入任务列表中
    } finally {
        mainLock.unlock(); // 无论try块如何退出都要释放锁
    }

    tryTerminate(); // 如果条件允许，尝试终止执行器

    return tasks; // 返回队列中未被执行的任务列表
}
```

就是将线程池的状态修改为 STOP，然后尝试打断所有的线程，从阻塞队列中移除剩余的任务，这也是为什么 shutdownNow 不能执行剩余任务的原因。

所以也可以看出 shutdown 方法和 shutdownNow 方法的主要区别就是，shutdown 之后还能处理在队列中的任务，shutdownNow 直接就将任务从队列中移除，线程池里的线程就不再处理了。

八、线程池的监控

在项目中使用线程池的时候，一般需要对线程池进行监控，方便出问题的时候快速定位。线程池本身提供了一些方法来获取线程池的运行状态。

- getCompletedTaskCount：已经执行完成的任务数量
- getLargestPoolSize：线程池里曾经创建过的最大的线程数量。这个主要是用来判断线程是否满过。
- getActiveCount：获取正在执行任务的线程数据

- getPoolSize：获取当前线程池中线程数量的大小

除了线程池提供的上述已经实现的方法，同时线程池也预留了很多扩展方法。比如在 runWorker 方法里面，执行任务之前会回调 beforeExecute 方法，执行任务之后会回调 afterExecute 方法，而这些方法默认都是空实现，小伙伴们可以自己继承 ThreadPoolExecutor 来重写这些方法，实现自己想要的功能。

九、线程池的使用场景

在 Java 程序中，其实经常需要用到多线程来处理一些业务，但是不建议单纯继承 Thread 或者实现 Runnable 接口来创建线程，这样会导致频繁创建及销毁线程，同时创建过多的线程也可能引发资源耗尽的风险。

所以使用线程池是一种更合理的选择，方便管理任务，同时实现线程的重复利用。所以线程池一般适合需要异步或者多线程处理任务的场景。

以下是几个线程池使用场景的简单示例：

01、Web服务器模拟：

模拟一个简单的Web服务器，接受请求并使用线程池进行处理。

```
import java.util.concurrent.*;

public class SimpleWebServer {
    private static final int NTHREADS = 100;
    private static final ExecutorService exec = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) {
        while (true) {
            // 接收请求
            Runnable request = new Runnable() {
                public void run() {
                    // 处理请求
                    System.out.println("Request handled by " +
Thread.currentThread().getName());
                }
            };

            exec.execute(request);
        }
    }
}
```

02、并行计算：

使用线程池进行并行的数值计算。

```
import java.util.concurrent.*;

public class ParallelCalculation {
```

```
    private static final int NTHREADS = 4;
    private static final ExecutorService exec = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) {
        Callable<Double> task = new Callable<Double>() {
            @Override
            public Double call() {
                // 这里模拟一些数值计算
                return Math.random() * 100;
            }
        };

        List<Future<Double>> results = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            results.add(exec.submit(task));
        }

        for (Future<Double> result : results) {
            try {
                System.out.println(result.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }

        exec.shutdown();
    }
}
```

03、异步任务处理：

模拟处理异步任务。

```
import java.util.concurrent.*;

public class AsynchronousTaskProcessor {

    private static final ExecutorService exec = Executors.newCachedThreadPool();

    public static void main(String[] args) {
        exec.execute(() -> {
            // 执行某些异步任务
            System.out.println("Async task started");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            System.out.println("Async task completed");
        });

        System.out.println("Main thread continues to execute other operations.");
        exec.shutdown();
    }
}
```

十、Executors 构建线程池以及问题分析

在上面的示例中，我们使用了 JDK 内部提供的 Executors 工具类来快速创建线程池。

1）固定线程数量的线程池：核心线程数与最大线程数相等

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

2）单个线程数量的线程池

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

3）接近无限大线程数量的线程池

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

4）带定时调度功能的线程池

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

虽然 JDK 提供了快速创建线程池的方法，但其实不推荐使用 Executors 来创建线程池，因为从上面构造线程池的代码可以看出，newFixedThreadPool 线程池由于使用了 LinkedBlockingQueue，队列的容量默认无限大，实际使用中出现任务过多时会导致内存溢出；newCachedThreadPool 线程池由于核心线程数无限大，当任务过多的时候会导致创建大量的线程，可能机器负载过高导致服务宕机。

这也是面试常问的一道八股文，大家需要注意。

十一、实际项目中如何合理的自定义线程池

通过上面分析提到，通过 Executors 这个工具类来创建的线程池其实都无法满足实际的使用场景，那么在实际的项目中，到底该如何构造线程池呢，该如何合理的设置参数?

线程数

线程数的设置主要取决于业务是 IO 密集型还是 CPU 密集型。

CPU 密集型：指的是任务主要使用来进行大量的计算，没有什么导致线程阻塞。一般这种场景的线程数设置为 CPU 核心数+1。

IO 密集型：当执行任务需要大量的 io，比如磁盘 io，网络 io，可能会存在大量的阻塞，所以在 IO 密集型任务中使用多线程可以大大地加速任务的处理。一般线程数设置为 2*CPU 核心数

Java 中用来获取 CPU 核心数的方法是：`Runtime.getRuntime().availableProcessors();`

线程工厂

一般建议自定义线程工厂，构建线程的时候设置线程的名称，这样在查日志的时候就方便知道是哪个线程执行的代码。

有界队列

一般需要设置有界队列的大小，比如 LinkedBlockingQueue 在构造的时候可以传入参数来限制队列中任务数据的大小，这样就不会因为无限往队列中扔任务导致系统的 oom。

OK，我们来通过自定义 ThreadPoolExecutor 改造一下前面使用 Executors 的例子。

Web服务器模拟：

Web服务器通常需要处理I/O操作，比如网络I/O，因此它们被视为I/O密集型任务。因此，我们将线程数设置为2 * CPU核心数。

```
import java.util.concurrent.*;

public class SimpleWebServer {
    private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
    private static final int CORE_POOL_SIZE = 2 * CPU_COUNT;
    private static final int MAX_POOL_SIZE = 2 * CPU_COUNT + 1;

    private static final ThreadPoolExecutor exec = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            60L,
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(1000)
    );
```

```
    public static void main(String[] args) {
        while (true) {
            Runnable request = () -> System.out.println("Request handled by " +
Thread.currentThread().getName());

            exec.execute(request);
        }
    }
}
```

并行计算：

并行计算任务主要用于计算，没有I/O阻塞，所以它们是CPU密集型的。线程数设置为CPU核心数 + 1。

```
import java.util.*;
import java.util.concurrent.*;

public class ParallelCalculation {
    private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
    private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
    private static final int MAX_POOL_SIZE = CPU_COUNT * 2;

    private static final ThreadPoolExecutor exec = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            10L,
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(1000)
    );

    public static void main(String[] args) {
        Callable<Double> task = () -> Math.random() * 100;

        List<Future<Double>> results = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            results.add(exec.submit(task));
        }

        for (Future<Double> result : results) {
            try {
                System.out.println(result.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }

        exec.shutdown();
    }
}
```

异步任务处理：

异步任务通常涉及到I/O操作，比如数据库查询或文件读写，因此它们被视为I/O密集型任务。因此，我们将线程数设置为2 * CPU核心数。

```
import java.util.concurrent.*;

public class AsynchronousTaskProcessor {
    private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
    private static final int CORE_POOL_SIZE = 2 * CPU_COUNT;
    private static final int MAX_POOL_SIZE = 2 * CPU_COUNT + 2;

    private static final ThreadPoolExecutor exec = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            60L,
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(1000)
    );

    public static void main(String[] args) {
        exec.execute(() -> {
            System.out.println("Async task started");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Async task completed");
        });

        System.out.println("Main thread continues to execute other operations.");
        exec.shutdown();
    }
}
```

十二、总结

本文主要介绍了线程池的原理以及使用场景，线程池主要是通过阻塞队列来实现的，线程池的使用场景主要是异步或者多线程处理任务的场景。线程池的使用可以通过 Executors 来快速创建，但是不推荐使用，因为 Executors 创建的线程池都有一些缺陷，比如无界队列可能导致内存溢出，无限大的线程数可能导致机器负载过高。所以在实际的项目中，建议自定义线程池 ThreadPoolExecutor，根据业务场景来合理的设置线程数，队列大小等参数。

> 编辑：沉默王二，部分内容来自于读者三友的公众号文章 https://mp.weixin.qq.com/s/lVem8mGANea8aUYF3XCO7Q，写得非常不错，强烈推荐。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第一版 PDF 终于来了！包括Java基础语法、数组&字符串、OOP、集合框架、Java IO、异常处理、Java 新特性、网络编程、NIO、并发编程、JVM等等，共计 32 万余字，可以说是通俗易懂、风趣幽默……详情戳：太赞了，GitHub 上标星 9300+ 的 Java 教程

微信搜 **沉默王二** 或扫描下方二维码关注二哥的原创公众号沉默王二，回复 **222** 即可免费领取。

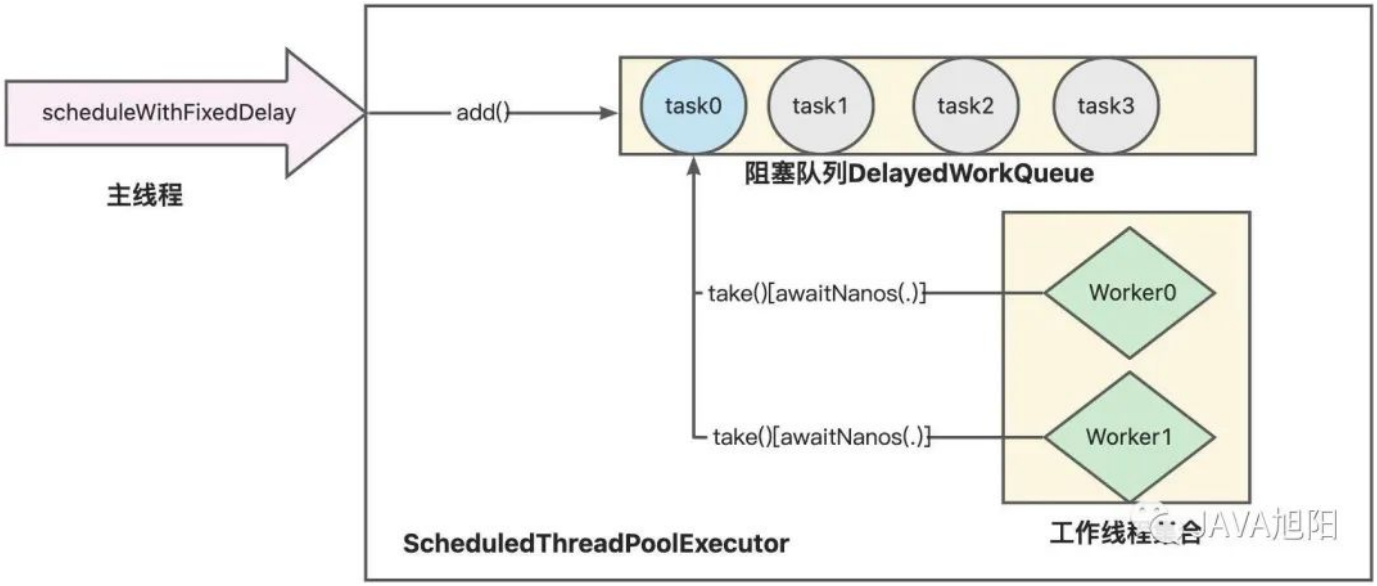
第二十六节：定时任务 ScheduledThreadPoolExecutor

定时任务 `ScheduledThreadPoolExecutor` 类有两个用途：指定时间延迟后执行任务；周期性重复执行任务。

JDK 1.5 之前，主要使用 `Timer` 类来完成定时任务，但是 `Timer` 有以下缺陷：

- Timer 是**单线程**模式；
- 如果在执行任务期间某个 TimerTask 耗时较久，就会影响其它任务的调度；
- Timer 的任务调度是基于绝对时间的，对**系统时间**敏感；
- Timer 不会捕获执行 TimerTask 时所抛出的**异常**，由于 Timer 是单线程的，所以一旦出现异常，线程就会终止，其他任务无法执行。

于是 JDK 1.5 之后，开发者就抛弃了 `Timer`，开始使用 `ScheduledThreadPoolExecutor`。先通过下面这张图感受下。



使用案例

假设我们有这样一个需求，指定时间给其他人发送消息。那么我们会将消息（包含发送时间）存储在数据库中，然后用一个定时任务，每隔 1 秒检查数据库在当前时间有没有需要发送的消息，那这个计划任务怎么完成呢？下面是一个 Demo:

```
public class ThreadPool {

    private static final ScheduledExecutorService executor = new
        ScheduledThreadPoolExecutor(1, Executors.defaultThreadFactory());

    private static SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args){
        // 新建一个固定延迟时间的计划任务
        executor.scheduleWithFixedDelay(new Runnable() {
            @Override
            public void run() {
                if (haveMsgAtCurrentTime()) {
                    System.out.println(df.format(new Date()));
                    System.out.println("大家注意了，我要发消息了");
                }
            }
        }, 1, 1, TimeUnit.SECONDS);
    }

    public static boolean haveMsgAtCurrentTime(){
        //查询数据库，有没有当前时间需要发送的消息
        //这里省略实现，直接返回true
        return true;
    }
}
```

下面截取一段输出（demo 会一直运行下去）：

```
2023-08-24 16:16:48
大家注意了，我要发消息了
2023-08-24 16:16:49
大家注意了，我要发消息了
2023-08-24 16:16:50
大家注意了，我要发消息了
2023-08-24 16:16:51
大家注意了，我要发消息了
2023-08-24 16:16:52
大家注意了，我要发消息了
2023-08-24 16:16:53
大家注意了，我要发消息了
2023-08-24 16:16:54
大家注意了，我要发消息了
2023-08-24 16:16:55
大家注意了，我要发消息了
```

这就是 `ScheduledThreadPoolExecutor` 的一个简单运用，接下来我们来看看它的实现原理。

类结构

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor
  implements ScheduledExecutorService {

    public ScheduledThreadPoolExecutor(int corePoolSize,ThreadFactory threadFactory) {
         super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue(), threadFactory);
    }
    //……
}
```

`ScheduledThreadPoolExecutor` 继承了ThreadPoolExecutor，并实现了`ScheduledExecutorService`接口。线程池 ThreadPoolExecutor 在之前介绍过了，相信大家都还有印象，接下来我们来看看 `ScheduledExecutorService` 接口。

```
public interface ScheduledExecutorService extends ExecutorService {

    /**
     * 安排一个Runnable任务在给定的延迟后执行。
     *
     * @param command 需要执行的任务
     * @param delay 延迟时间
     * @param unit 时间单位
     * @return 可用于提取结果或取消的ScheduledFuture
     */
    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
```

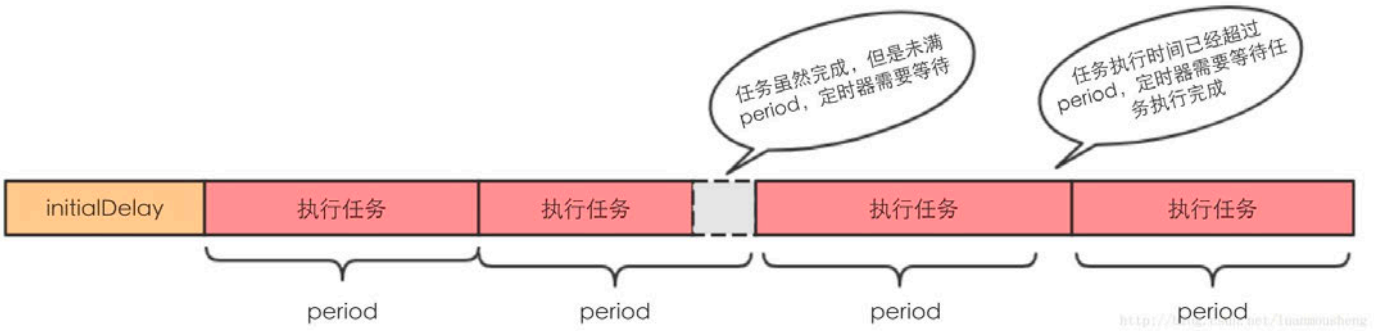
```
    /**
     * 安排一个Callable任务在给定的延迟后执行。
     *
     * @param callable 需要执行的任务
     * @param delay 延迟时间
     * @param unit 时间单位
     * @return 可用于提取结果或取消的ScheduledFuture
     */
    public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit
unit);

    /**
     * 安排一个Runnable任务在给定的初始延迟后首次执行，随后每个period时间间隔执行一次。
     *
     * @param command 需要执行的任务
     * @param initialDelay 首次执行的初始延迟
     * @param period 连续执行之间的时间间隔
     * @param unit 时间单位
     * @return 可用于提取结果或取消的ScheduledFuture
     */
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);

    /**
     * 安排一个Runnable任务在给定的初始延迟后首次执行，随后每次完成任务后等待指定的延迟再次执行。
     *
     * @param command 需要执行的任务
     * @param initialDelay 首次执行的初始延迟
     * @param delay 每次执行结束后的延迟时间
     * @param unit 时间单位
     * @return 可用于提取结果或取消的ScheduledFuture
     */
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit);
}
```

`ScheduledExecutorService` 接口继承了 `ExecutorService` 接口，并增加了几个定时相关的接口方法。前两个方法用于单次调度执行任务，区别是有没有返回值。

重点介绍一下后面两个方法：

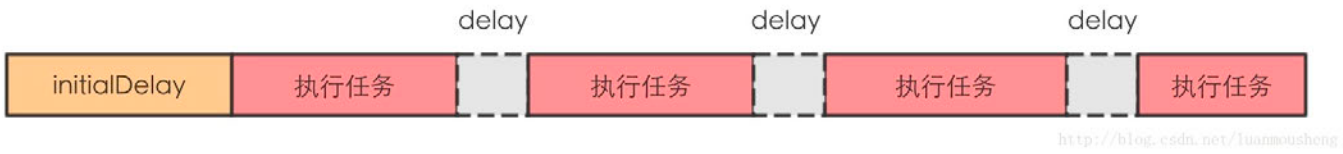
01、scheduleAtFixedRate

scheduleAtFixedRate 方法在`initialDelay`时长后第一次执行任务，以后每隔`period`时长再次执行任务。注意，period 是从**任务开始执行算起**的。开始执行任务后，定时器每隔 period 时长**检查该任务是否完成**，如果完成则再次启动任务，否则等该任务结束后才启动任务。看下图：



02、scheduleWithFixDelay

该方法在`initialDelay`时长后第一次执行任务，以后每当任务执行**完成后**，等待`delay`时长，再次执行任务。看下图。



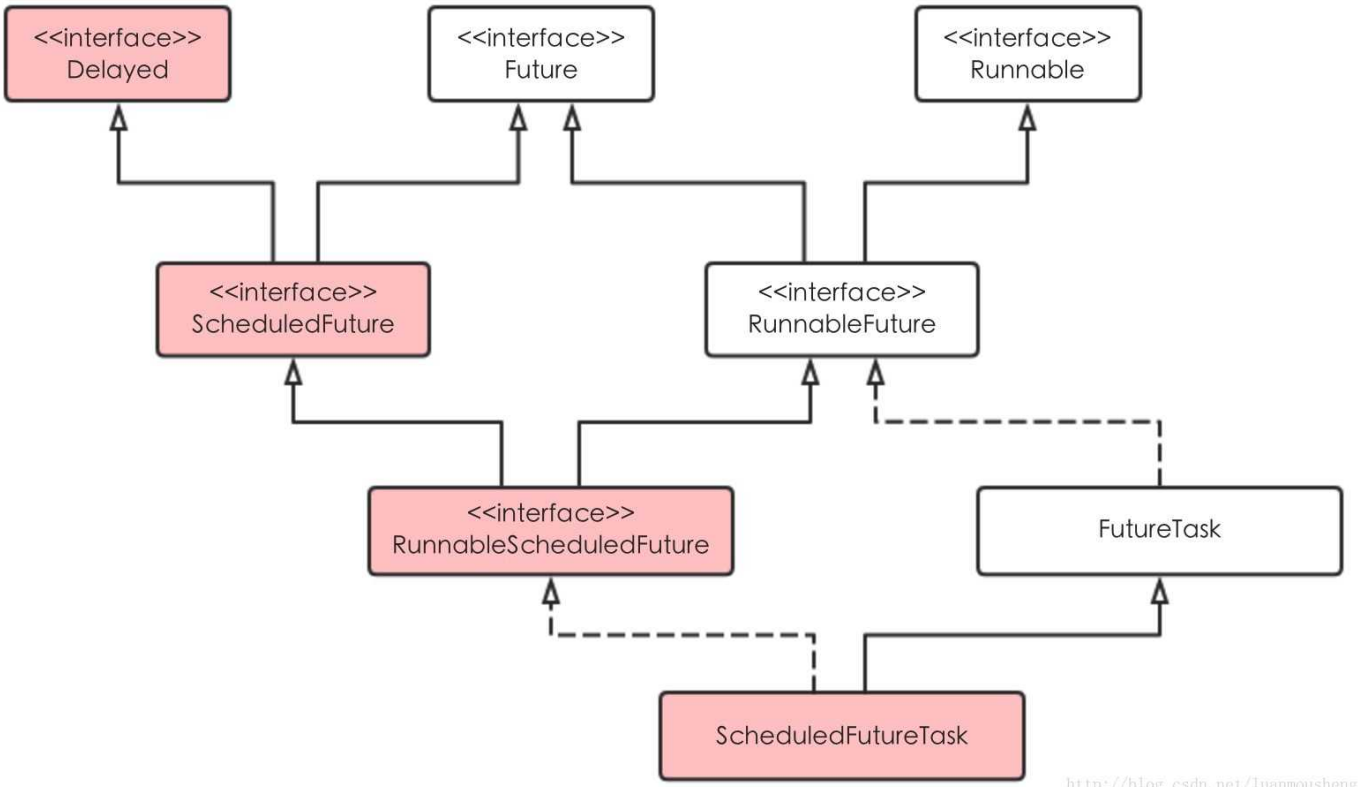
相信大家能体会出来其中的差异。

主要方法

schedule

```
// delay时长后执行任务command，该任务只执行一次
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    // 这里的decorateTask方法仅仅返回第二个参数
    RunnableScheduledFuture<?> t = decorateTask(command,
                                        new ScheduledFutureTask<Void>(command, null,
triggerTime(delay,unit)));
    // 延时或者周期执行任务的主要方法,稍后统一说明
    delayedExecute(t);
    return t;
}
```

我们先看看里面涉及到的几个类和接口的关系图谱：



Delayed 接口

```
// 继承Comparable接口，表示该类对象支持排序
public interface Delayed extends Comparable<Delayed> {
    // 返回该对象剩余时延
    long getDelay(TimeUnit unit);
}
```

`Delayed`接口很简单，继承了`Comparable`接口，表示对象是可以比较排序的。

ScheduledFuture 接口

```
// 仅仅继承了Delayed和Future接口，自己没有任何代码
public interface ScheduledFuture<V> extends Delayed, Future<V> {
}
```

RunnableScheduledFuture 接口

```
public interface RunnableScheduledFuture<V> extends RunnableFuture<V>,
ScheduledFuture<V> {
    // 是否是周期任务，周期任务可被调度运行多次，非周期任务只被运行一次
    boolean isPeriodic();
}
```

ScheduledFutureTask 类

回到`schecule`方法中，它创建了一个`ScheduledFutureTask`对象，由上面的关系图可知，`ScheduledFutureTask`直接或者间接实现了很多接口，一起看看`ScheduledFutureTask`里面的实现方法吧。

构造方法

```
ScheduledFutureTask(Runnable r, V result, long ns, long period) {
    // 调用父类FutureTask的构造方法
    super(r, result);
    // time表示任务下次执行的时间
    this.time = ns;
    // 周期任务，正数表示按照固定速率，负数表示按照固定时延,0表示不是周期任务
    this.period = period;
    // 任务的编号
    this.sequenceNumber = sequencer.getAndIncrement();
}
```

Delayed 接口的实现

```
// 实现Delayed接口的getDelay方法，返回任务开始执行的剩余时间
public long getDelay(TimeUnit unit) {
    return unit.convert(time - now(), TimeUnit.NANOSECONDS);
}
```

Comparable 接口的实现

```
// Comparable接口的compareTo方法，比较两个任务的"大小"。
public int compareTo(Delayed other) {
    if (other == this)
      return 0;
    if (other instanceof ScheduledFutureTask) {
      ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
      long diff = time - x.time;
      // 小于0，说明当前任务的执行时间点早于other，要排在延时队列other的前面
      if (diff < 0)
        return -1;
      // 大于0，说明当前任务的执行时间点晚于other，要排在延时队列other的后面
      else if (diff > 0)
        return 1;
      // 如果两个任务的执行时间点一样，比较两个任务的编号，编号小的排在队列前面，编号大的排在队列后面
      else if (sequenceNumber < x.sequenceNumber)
        return -1;
      else
        return 1;
    }
    // 如果任务类型不是ScheduledFutureTask，通过getDelay方法比较
```

```
    long d = (getDelay(TimeUnit.NANOSECONDS) -
              other.getDelay(TimeUnit.NANOSECONDS));
    return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
}
```

setNextRunTime

```
// 任务执行完后，设置下次执行的时间
private void setNextRunTime() {
    long p = period;
    // p > 0，说明是固定速率运行的任务
    // 在原来任务开始执行时间的基础上加上p即可
    if (p > 0)
      time += p;
    // p < 0，说明是固定时延运行的任务，
    // 下次执行时间在当前时间(任务执行完成的时间)的基础上加上-p的时间
    else
      time = triggerTime(-p);
}
```

Runnable 接口实现

```
public void run() {
    boolean periodic = isPeriodic();
    // 如果当前状态下不能执行任务，则取消任务
    if (!canRunInCurrentRunState(periodic))
      cancel(false);
    // 不是周期性任务，执行一次任务即可，调用父类的run方法
    else if (!periodic)
      ScheduledFutureTask.super.run();
    // 是周期性任务，调用FutureTask的runAndReset方法，方法执行完成后
    // 重新设置任务下一次执行的时间，并将该任务重新入队，等待再次被调度
    else if (ScheduledFutureTask.super.runAndReset()) {
      setNextRunTime();
      reExecutePeriodic(outerTask);
    }
}
```

总结一下 run 方法的执行过程：

1. 如果当前线程池运行状态不运行执行任务，那么就取消该任务，然后直接返回，否则执行步骤 2；
2. 如果不是周期性任务，调用 FutureTask 中的 run 方法执行，会设置执行结果，然后直接返回，否则执行步骤 3；
3. 如果是周期性任务，调用 FutureTask 中的 runAndReset 方法执行，不会设置执行结果，然后直接返回，否则执行步骤 4 和步骤 5；
4. 计算下次执行该任务的具体时间；
5. 重复执行任务。

`runAndReset`方法是为任务多次执行而设计的。`runAndReset`方法执行完任务后不会设置任务的执行结果，也不会去更新任务的状态，以及维持任务的状态为初始状态（**NEW**状态），这也是该方法和 FutureTask `run`方法的区别。

scheduleAtFixedRate

我们看一下代码：

```
// 注意，固定速率和固定时延，传入的参数都是Runnable，也就是说这种定时任务是没有返回值的
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit) {
    if (command == null || unit == null)
      throw new NullPointerException();
    if (period <= 0)
      throw new IllegalArgumentException();
    // 创建一个有初始延时和固定周期的任务
    ScheduledFutureTask<Void> sft =
      new ScheduledFutureTask<Void>(command,
                                    null,
                                    triggerTime(initialDelay, unit),
                                    unit.toNanos(period));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    // outerTask表示将会重新入队的任务
    sft.outerTask = t;
    // 稍后说明
    delayedExecute(t);
    return t;
}
```

`scheduleAtFixedRate`这个方法和`schedule`类似，不同点是`scheduleAtFixedRate`方法内部创建的是`ScheduledFutureTask`，带有初始延时和固定周期的任务。

scheduleWithFixedDelay

`scheduleWithFixedDelay`也是通过`ScheduledFutureTask`体现的，唯一不同的地方在于创建的`ScheduledFutureTask`不同。

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                 long initialDelay,
                                                 long delay,
                                                 TimeUnit unit) {
    if (command == null || unit == null)
      throw new NullPointerException();
    if (delay <= 0)
      throw new IllegalArgumentException();
    // 创建一个有初始延时和固定时延的任务
```

```
    ScheduledFutureTask<Void> sft =
      new ScheduledFutureTask<Void>(command,
                                    null,
                                    triggerTime(initialDelay, unit),
                                    unit.toNanos(-delay));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    // outerTask表示将会重新入队的任务
    sft.outerTask = t;
    // 稍后说明
    delayedExecute(t);
    return t;
}
```

delayedExecute

前面讲到的`schedule`、`scheduleAtFixedRate`和`scheduleWithFixedDelay`最后都调用了`delayedExecute`方法，该方法是定时任务执行的主要方法。一起来看看源码：

```
private void delayedExecute(RunnableScheduledFuture<?> task) {
    // 线程池已经关闭，调用拒绝执行处理器处理
    if (isShutdown())
      reject(task);
    else {
      // 将任务加入到等待队列
      super.getQueue().add(task);
      // 线程池已经关闭，且当前状态不能运行该任务，将该任务从等待队列移除并取消该任务
      if (isShutdown() &&
          !canRunInCurrentRunState(task.isPeriodic()) &&
          remove(task))
        task.cancel(false);
      else
        // 增加一个worker，就算corePoolSize=0也要增加一个worker
        ensurePrestart();
    }
}
```

`delayedExecute`方法的逻辑也很简单，主要就是将任务添加到等待队列，然后调用`ensurePrestart`方法。

```
void ensurePrestart() {
    int wc = workerCountOf(ctl.get());
    if (wc < corePoolSize)
        addWorker(null, true);
    else if (wc == 0)
        addWorker(null, false);
}
```

`ensurePrestart`方法主要是调用了`addWorker`方法，线程池中的工作线程就是通过该方法来启动并执行任务的。相信大家都还有印象。

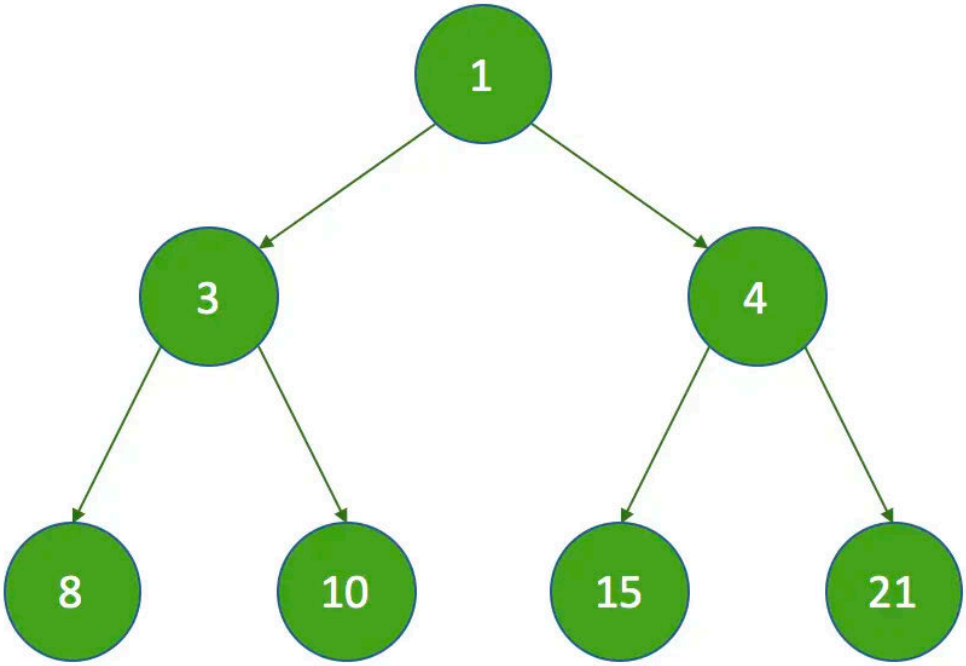
对于`ScheduledThreadPoolExecutor`，`worker`添加到线程池后会在等待队列中等待获取任务，这点是和`ThreadPoolExecutor`是一致的。**但是 worker 是怎么从等待队列取定时任务的呢?**

DelayedWorkQueue

`ScheduledThreadPoolExecutor`使用了`DelayedWorkQueue`来保存等待的任务。

该等待队列的队首应该保存的是**最近将要执行的任务**，所以`worker`只关心队首任务，如果队首任务的开始执行时间还未到，worker 也应该继续等待。

DelayedWorkQueue 是一个无界优先队列，使用数组存储，底层使用堆结构来实现优先队列的功能。



可以转换成如下的数组：

在这种结构中，可以发现有如下特性。假设，索引值从 0 开始，子节点的索引值为 k，父节点的索引值为 p，则：

- 一个节点的左子节点的索引为：k = p * 2 + 1；
- 一个节点的右子节点的索引为：k = (p + 1) * 2；
- 一个节点的父节点的索引为：p = (k - 1) / 2。

我们来看看 DelayedWorkQueue 的声明和成员变量：

```
static class DelayedWorkQueue extends AbstractQueue<Runnable>
implements BlockingQueue<Runnable> {
  // 队列初始容量
  private static final int INITIAL_CAPACITY = 16;
  // 数组用来存储定时任务，通过数组实现堆排序
  private RunnableScheduledFuture[] queue = new
RunnableScheduledFuture[INITIAL_CAPACITY];
  // 当前在队首等待的线程
  private Thread leader = null;
  // 锁和监视器，用于leader线程
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition available = lock.newCondition();
  // 其他代码，略
}
```

当一个线程成为 leader，它只需等待队首任务的 delay 时间即可，其他线程会无条件等待。leader 取到任务返回前要通知其他线程，直到有线程成为新的 leader。每当队首的定时任务被其他更早需要执行的任务替换，leader 就设置为 null，其他等待的线程（被当前 leader 通知）和当前的 leader 重新竞争成为 leader。

> 所有线程都会有三种身份中的一种：leader、follower，以及一个干活中的状态：proccesser。它的基本原则是，永远最多只有一个 leader。所有 follower 都在等待成为 leader。线程池启动时会自动产生一个 Leader 负责等待网络 IO 事件，当有一个事件产生时，Leader 线程首先通知一个 Follower 线程将其提拔为新的 Leader，然后自己就去干活了，去处理这个网络事件，处理完毕后加入 Follower 线程等待队列，等待下次成为 Leader。这种方法可以增强 CPU 高速缓存相似性，及消除动态内存分配和线程间的数据交换。

同时，定义了 ReentrantLock 锁 lock 和 Condition available 用于控制和通知下一个线程竞争成为 leader。

当一个新的任务成为队首，或者需要有新的线程成为 leader 时，available 监视器上的线程将会被通知，然后竞争成为 leader 线程。有些类似于生产者-消费者模式。

DelayedWorkQueue 是一个优先级队列，它可以保证每次出队的任务都是当前队列中执行时间最靠前的，由于它是基于堆结构的队列，堆结构在执行插入和删除操作时的最坏时间复杂度是 `O(logN)`。

接下来看看`DelayedWorkQueue`中几个比较重要的方法。

take

```
public RunnableScheduledFuture take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
      for (;;) {
        // 取堆顶的任务，堆顶是最近要执行的任务
        RunnableScheduledFuture first = queue[0];
        // 堆顶为空，线程要在条件available上等待
        if (first == null)
          available.await();
        else {
          // 堆顶任务还要多长时间才能执行
          long delay = first.getDelay(TimeUnit.NANOSECONDS);
```

```
            // 堆顶任务已经可以执行了，finishPoll会重新调整堆，使其满足最小堆特性，该方法设置任务在
            // 堆中的index为-1并返回该任务
            if (delay <= 0)
              return finishPoll(first);
            // 如果leader不为空，说明已经有线程成为leader并等待堆顶任务
            // 到达执行时间，此时，其他线程都需要在available条件上等待
            else if (leader != null)
              available.await();
            else {
              // leader为空，当前线程成为新的leader
              Thread thisThread = Thread.currentThread();
              leader = thisThread;
              try {
                // 当前线程已经成为leader了，只需要等待堆顶任务到达执行时间即可
                available.awaitNanos(delay);
              } finally {
                // 返回堆顶元素之前将leader设置为空
                if (leader == thisThread)
                  leader = null;
              }
            }
          }
        } finally {
          // 通知其他在available条件等待的线程，这些线程可以去竞争成为新的leader
          if (leader == null && queue[0] != null)
            available.signal();
          lock.unlock();
        }
    }
```

`take`方法是什么时候调用的呢?

在讲解线程池的时候，我们介绍了`getTask`方法，工作线程会循环从`workQueue`中取任务。但计划任务却不同，因为一旦`getTask`方法取出了任务就开始执行了，而这时可能还没有到执行时间，所以在`take`方法中，要保证只有到指定的执行时间，任务才可以被取走。

总结一下流程：

1. 如果堆顶元素为空，在 available 上等待。
2. 如果堆顶任务的执行时间已到，将堆顶元素替换为堆的最后一个元素并调整堆使其满足**最小堆**特性，同时设置任务在堆中索引为-1，返回该任务。
3. 如果 leader 不为空，说明已经有线程成为 leader 了，其他线程都要在 available 监视器上等待。
4. 如果 leader 为空，当前线程成为新的 leader，并等待直到堆顶任务执行时间到达。
5. take 方法返回之前，将 leader 设置为空，并通知其他线程。

再来说一下 leader 的作用，这里的 leader 是**为了减少不必要的定时等待**，当一个线程成为 leader 时，它只等待下一个节点的时间间隔，但其它线程无限期等待。 leader 线程必须在`take()`或`poll()`返回之前 signal 其它线程，除非其他线程成为了 leader。

举例来说，如果没有 leader，那么在执行 take 时，都要执行`available.awaitNanos(delay)`，假设当前线程执行了该段代码，这时还没有 signal，第二个线程也执行了该段代码，则第二个线程也要被阻塞。

但只有一个线程返回队首任务，其他的线程在`awaitNanos(delay)`之后，继续执行 for 循环，因为队首任务已经被返回了，所以这个时候的 for 循环拿到的队首任务是新的，又需要重新判断时间，又要继续阻塞。

所以，为了不让多个线程频繁的做无用的定时等待，这里增加了 leader，如果 leader 不为空，则说明队列中第一个节点已经在等待出队，这时其它的线程会一直阻塞，减少了无用的阻塞（注意，在`finally`中调用了`signal()`来唤醒一个线程，而不是`signalAll()`）。

offer

该方法往队列插入一个值，返回是否成功插入。

```
public boolean offer(Runnable x) {
    if (x == null)
      throw new NullPointerException();
    RunnableScheduledFuture e = (RunnableScheduledFuture)x;
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
      int i = size;
      // 队列元素已经大于等于数组的长度，需要扩容，新堆的容量是原来堆容量的1.5倍
      if (i >= queue.length)
        grow();
      // 堆中元素增加1
      size = i + 1;
      // 调整堆
      if (i == 0) {
        queue[0] = e;
        setIndex(e, 0);
      } else {
          // 调整堆，使的满足最小堆，比较大小的方式就是上文提到的compareTo方法
        siftUp(i, e);
      }
      if (queue[0] == e) {
        leader = null;
        // 通知其他在available条件上等待的线程，这些线程可以竞争成为新的leader
        available.signal();
      }
    } finally {
      lock.unlock();
    }
    return true;
}
```

offer 方法实现了向延迟队列插入一个任务的操作，并保证整个队列仍然满足最小堆的性质。

> 最小堆（Min Heap）是一个完全二叉树，其中每一个父节点的值都小于或等于其子节点的值。换句话说，在最小堆中，根节点（即树的顶部）是所有节点中的最小值。

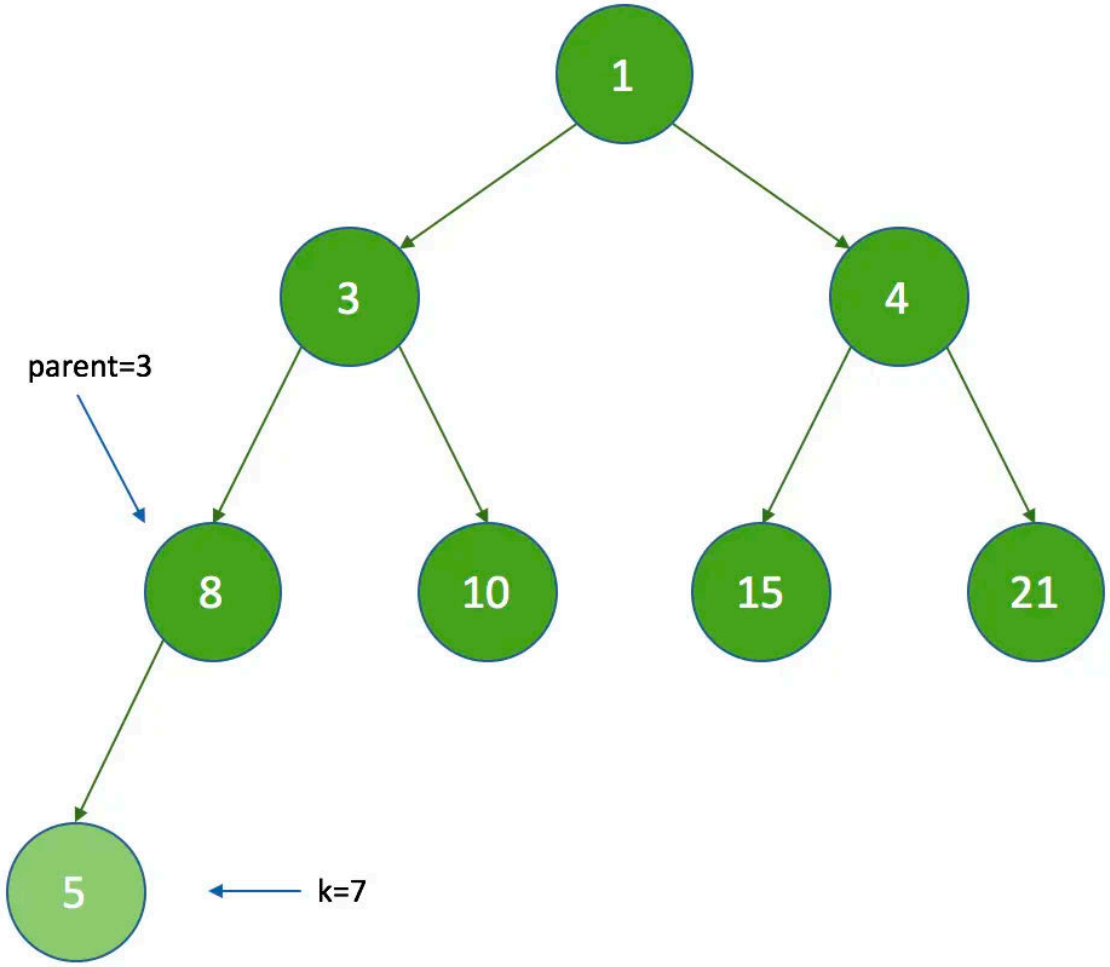
前面我们也提到过最小堆。我们来看一下用于调整堆的 siftUp 方法。

```
private void siftUp(int k, RunnableScheduledFuture<?> key) {
    while (k > 0) {
        // 找到父节点的索引
        int parent = (k - 1) >>> 1;
        // 获取父节点
        RunnableScheduledFuture<?> e = queue[parent];
        // 如果key节点的执行时间大于父节点的执行时间，不需要再排序了
        if (key.compareTo(e) >= 0)
            break;
        // 如果key.compareTo(e) < 0，说明key节点的执行时间小于父节点的执行时间，需要把父节点移到后
面
        queue[k] = e;
        // 设置索引为k
        setIndex(e, k);
        k = parent;
    }
    // key设置为排序后的位置中
    queue[k] = key;
    setIndex(key, k);
}
```

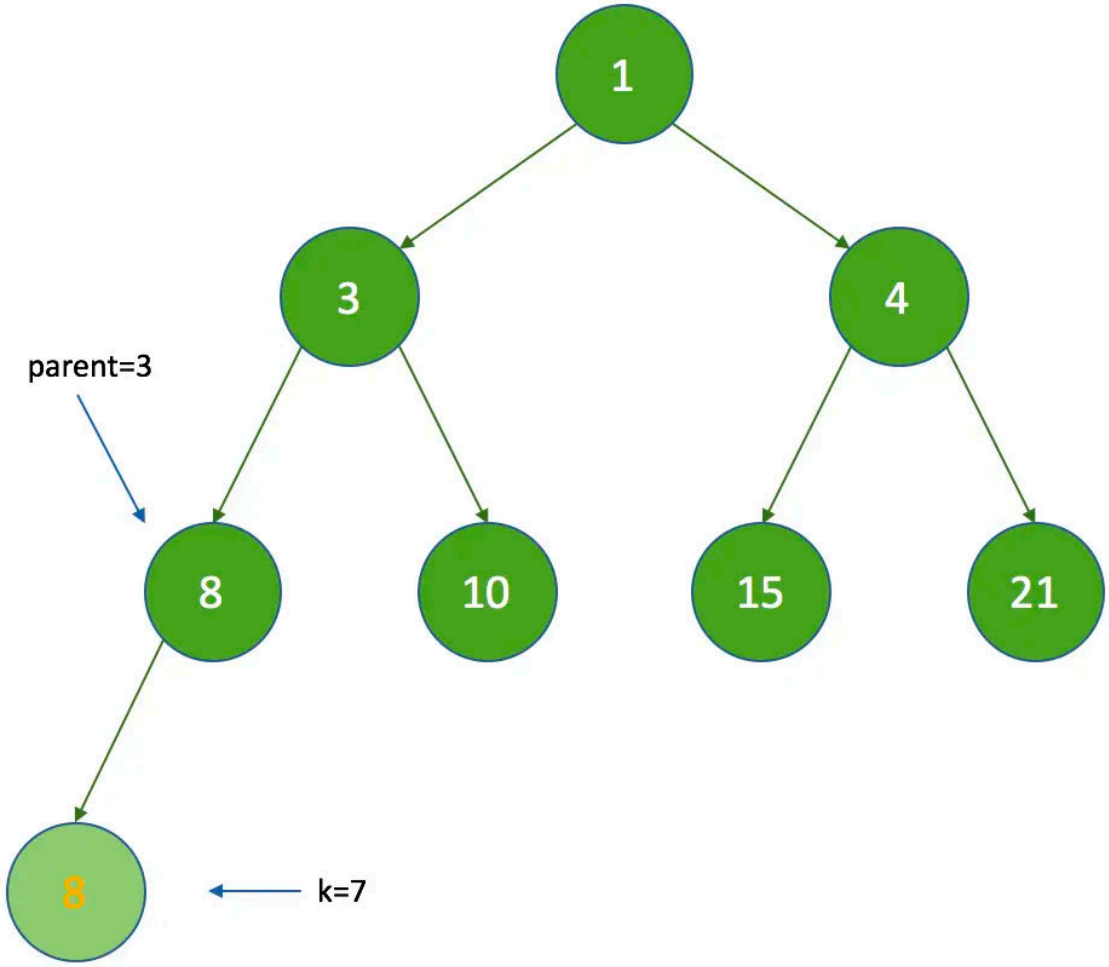
代码很好理解，就是循环的根据key节点与它的父节点来判断，如果key节点的执行时间小于父节点，则将两个节点交换，使执行时间靠前的节点排列在队列的前面。

假设新入队的节点的延迟时间（调用getDelay()方法获得）是5，执行过程如下：

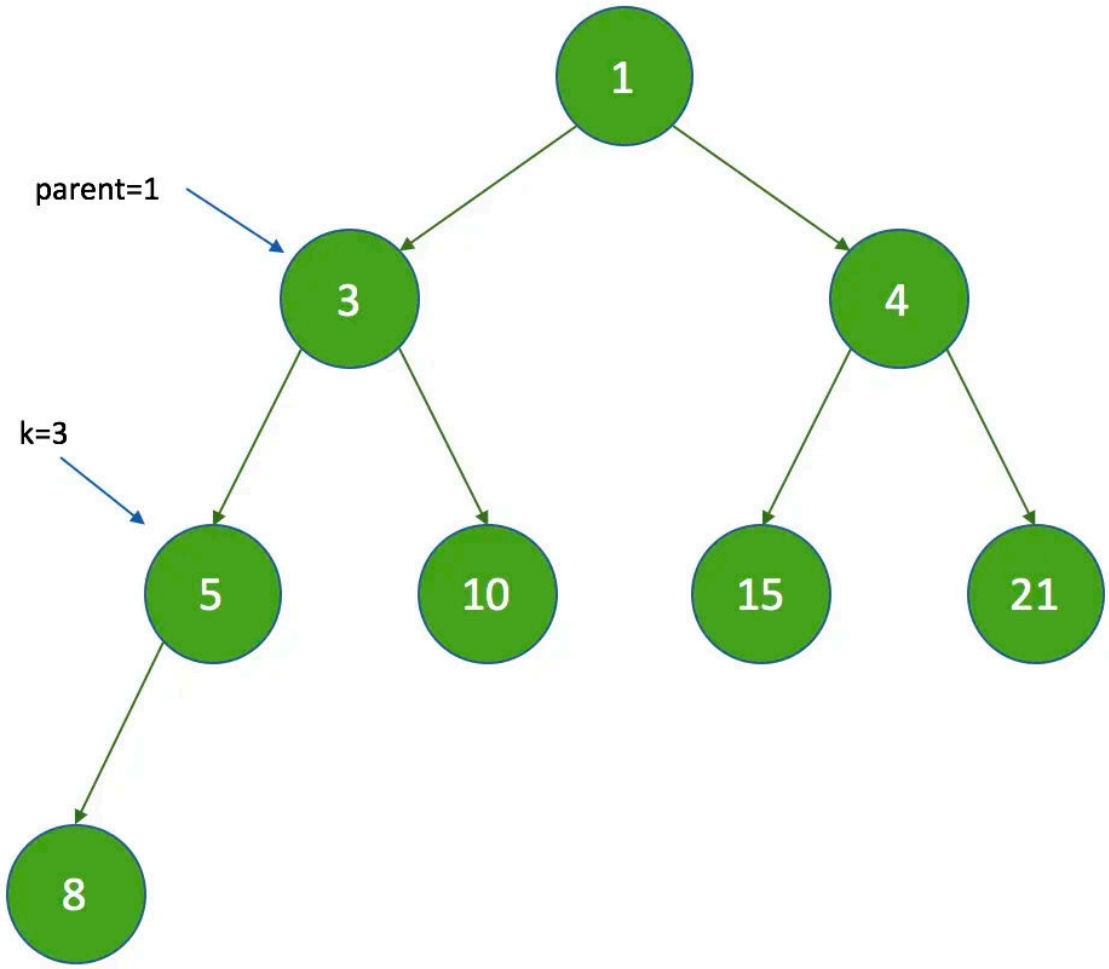
1、先将新的节点添加到数组的尾部，这时新节点的索引k为7：



2、计算新父节点的索引：`parent = (k - 1) >>> 1`，parent = 3，那么`queue[3]`的时间间隔值为8，因为`5 < 8`，将执行`queue[7] = queue[3]`：



3、这时将k设置为3，继续循环，再次计算parent为1，`queue[1]`的时间间隔为3，因为 `5 > 3` ，这时退出循环，最终k为3：



可见，每次新增节点时，只是根据父节点来判断，而不会影响兄弟节点。

小结

`ScheduledThreadPoolExecutor`是一个定时任务的线程池，它的主要作用是周期性的执行任务。它的实现原理是通过`DelayedWorkQueue`来保存等待的任务，`DelayedWorkQueue`是一个无界优先队列，使用数组存储，底层使用堆结构来实现优先队列的功能。

> 编辑：沉默王二，原文内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程，强烈推荐。其他参考链接如下：

- ideabuffer
- 博客园

推荐阅读：读者三友的 11 种延迟任务的实现方式

第二十七节：原子操作类 Atomic

我们前面讲过 CAS，相信大家都还有印象，Java 中的原子操作类，如 AtomicInteger 和 AtomicLong，底层就是利用 CAS 来确保变量更新的原子性的。

像递增运算 count++ 就不是一个原子操作，在多线程环境下并不能得到正确的结果，因为 count++ 操作实际上分为三个步骤：

1. 读取 count 变量的值；
2. 将 count 变量的值加 1；
3. 将 count 变量的值写入到内存中；

假定线程 A 正在修改 count 变量，为了保证线程 B 在使用 count 的时候是线程 A 修改过后的状态，可以用 synchronized 关键字同步一手。

```
private long count = 0;
public synchronized void write() {
    System.out.println("我寻了半生的春天，你一笑便是了。");
    count++;
}
```

但多个线程之间访问 `write()` 方法是互斥的，线程 B 访问的时候必须要等待线程 A 访问结束，有没有更好的办法呢？

AtomicInteger 是 JDK 提供的一个原子操作的 Integer 类，它提供的加减操作是线程安全的。于是我们可以这样：

```
private AtomicInteger count = new AtomicInteger(0);
public void write() {
    System.out.println("我寻了半生的春天，你一笑便是了。");
    count.incrementAndGet();
}
```

你看，这下是不是就舒服多了，不用加锁，也能保证线程安全。OK，接下来，我们来看看原子操作类都有哪些？

原子操作的基本数据类型

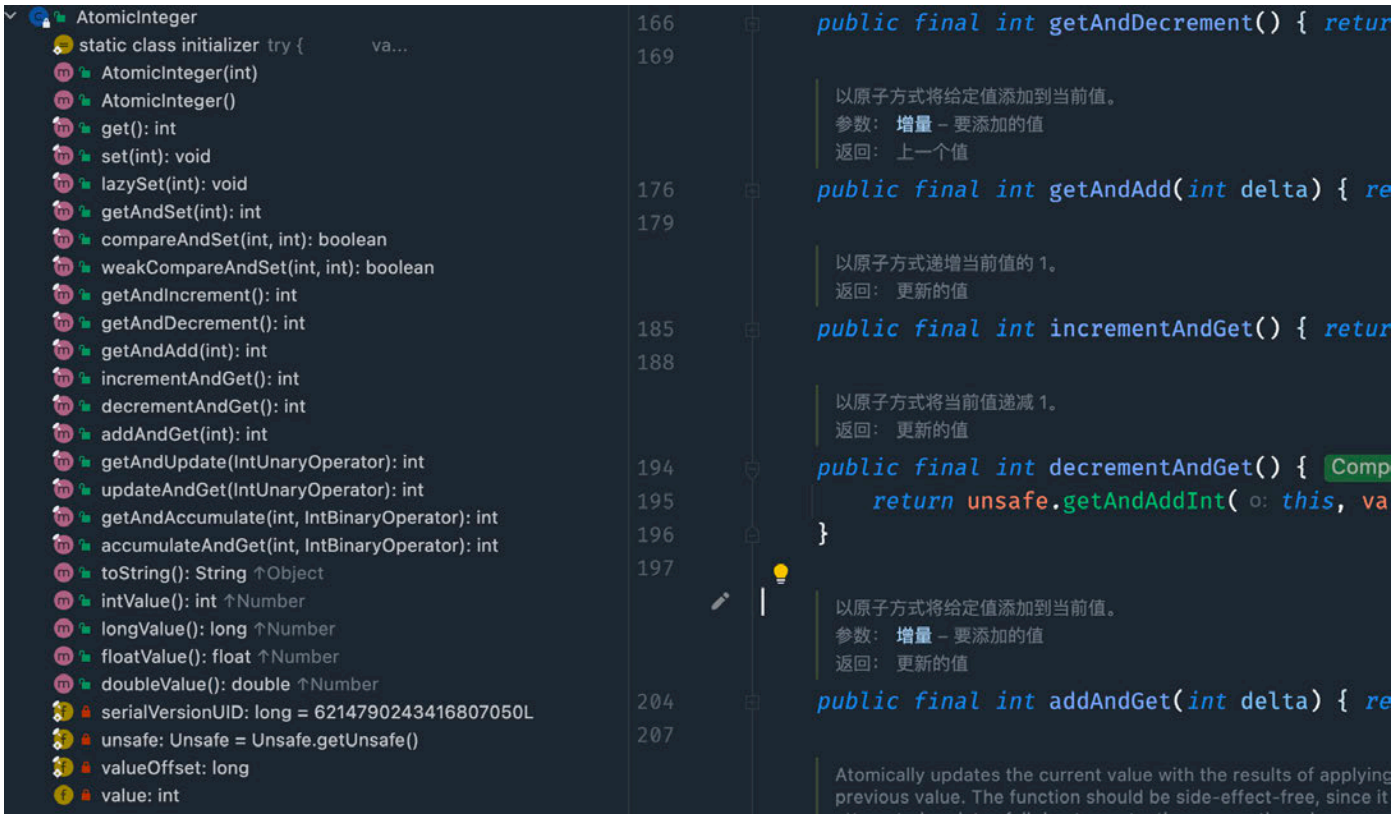
基本类型的原子操作主要有这些：

1. AtomicBoolean：以原子更新的方式更新 boolean；
2. AtomicInteger：以原子更新的方式更新 Integer；
3. AtomicLong：以原子更新的方式更新 Long；

这几个类的用法基本一致，这里以 AtomicInteger 为例。

1. `addAndGet(int delta)`：增加给定的 delta，并获取新值。
2. `incrementAndGet()`：增加 1，并获取新值。
3. `getAndSet(int newValue)`：获取当前值，并将新值设置为 newValue。
4. `getAndIncrement()`：获取当前值，并增加 1。

还有一些方法，可以直接查看 API，都很好理解。



为了能够弄懂 AtomicInteger 的实现原理，以 getAndIncrement 方法为例，来看下源码：

```
public final int getAndIncrement() {
    // 使用Unsafe类中的getAndAddInt方法原子地增加AtomicInteger的当前值
    // 第一个参数this是AtomicInteger的当前实例
    // 第二个参数valueOffset是一个偏移量，它指示在AtomicInteger对象中的哪个位置可以找到实际的int值
    // 第三个参数1表示要加到当前值上的值（即增加的值）
    // 此方法返回的是增加前的原始值
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

可以看出，该方法实际上是调用了 unsafe 对象的 getAndAddInt 方法，unsafe 对象是通过通过 UnSafe 类的静态方法 getUnsafe 获取的：

```
private static final Unsafe unsafe = Unsafe.getUnsafe();
```

Unsafe 类我们在讲 CAS 的时候也讲过，包括 AtomicInteger 类，相信大家还有印象。

Unsafe 类是 Java 中的一个特殊类，用于执行低级、不安全的操作。getAndIncrement 方法就是利用了 Unsafe 类提供的 CAS（Compare-And-Swap）操作来实现原子的 increment 操作。CAS 是一种常用的无锁技术，允许在多线程环境中原子地更新值。

好，下面用一个简单的例子来说明 AtomicInteger 的用法：

```
public class AtomicDemo {
    private static AtomicInteger atomicInteger = new AtomicInteger(1);

    public static void main(String[] args) {
        System.out.println(atomicInteger.getAndIncrement());
        System.out.println(atomicInteger.get());
    }
}
```

输出结果：

```
1
2
```

AtomicLong 和 AtomicInteger 的实现原理基本一致，只不过一个针对的是 long 型，一个针对的是 int 型。

AtomicBoolean 类是怎样实现更新的呢？核心方法是`compareAndSet`方法，其源码如下：

```
public final boolean compareAndSet(boolean expect, boolean update) {
    // 将expect布尔值转化为整数，true为1，false为0
    int e = expect ? 1 : 0;

    // 将update布尔值转化为整数，true为1，false为0
    int u = update ? 1 : 0;

    // 使用Unsafe类中的compareAndSwapInt方法尝试原子地更新AtomicBoolean的当前值
```

```
    // 第一个参数this是AtomicBoolean的当前实例
    // 第二个参数valueOffset是一个偏移量，它指示在AtomicBoolean对象中的哪个位置可以找到实际的int值
    // 第三个参数e是我们期望的当前值（转换为整数后的值）
    // 第四个参数u是我们想要更新的值（转换为整数后的值）
    // 如果当前值与期望值e相等，它会被原子地设置为u，并返回true；否则返回false。
    return unsafe.compareAndSwapInt(this, valueOffset, e, u);
}
```

该方法尝试将当前值从expect设置为update，但这种设置只会在当前值确实为expect时成功。方法返回true表示更新成功，否则返回false。

原子操作的数组类型

如果需要原子更新数组里的某个元素，atomic 也提供了相应的类：

1. AtomicIntegerArray：这个类提供了一些原子更新 int 整数数组的方法。
2. AtomicLongArray：这个类提供了一些原子更新 long 型证书数组的方法。
3. AtomicReferenceArray：这个类提供了一些原子更新引用类型数组的方法。

这几个类的用法一致，就以 AtomicIntegerArray 来总结下常用的方法：

1. `addAndGet(int i, int delta)`：以原子更新的方式将数组中索引为 i 的元素与输入值相加；
2. `getAndIncrement(int i)`：以原子更新的方式将数组中索引为 i 的元素自增加 1；
3. `compareAndSet(int i, int expect, int update)`：将数组中索引为 i 的位置的元素进行更新

可以看出，AtomicIntegerArray 与 AtomicInteger 的方法基本一致，只不过在 AtomicIntegerArray 的方法中会多一个数组索引 i。下面举一个简单的例子：

```
public class AtomicDemo {
    //    private static AtomicInteger atomicInteger = new AtomicInteger(1);
    private static int[] value = new int[]{1, 2, 3};
    private static AtomicIntegerArray integerArray = new AtomicIntegerArray(value);

    public static void main(String[] args) {
        //对数组中索引为1的位置的元素加5
        int result = integerArray.getAndAdd(1, 5);
        System.out.println(integerArray.get(1));
        System.out.println(result);
    }
}
```

输出结果：

```
7
2
```

通过 getAndAdd 方法将位置为 1 的元素加 5，从结果可以看出索引为 1 的元素变成了 7，该方法返回的也是相加之前的数为 2。

原子操作的引用类型

如果需要原子更新引用类型的话，atomic 也提供了相关的类：

1. AtomicReference：原子更新引用类型；
2. AtomicReferenceFieldUpdater：原子更新引用类型里的字段；
3. AtomicMarkableReference：原子更新带有标记位的引用类型；

这几个类的使用方法也是基本一样，以 AtomicReference 为例，来说明这些类的基本用法。下面是一个 demo：

```
public class AtomicDemo {

    private static AtomicReference<User> reference = new AtomicReference<>();

    public static void main(String[] args) {
        User user1 = new User("a", 1);
        reference.set(user1);
        User user2 = new User("b",2);
        User user = reference.getAndSet(user2);
        System.out.println(user);
        System.out.println(reference.get());
    }

    static class User {
        private String userName;
        private int age;

        public User(String userName, int age) {
            this.userName = userName;
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                    "userName='" + userName + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

输出结果：

```
User{userName='a', age=1}
User{userName='b', age=2}
```

首先将对象 User1 用 AtomicReference 进行封装，然后调用 getAndSet 方法进行赋值，从结果可以看出，该方法会原子更新 user 对象，变为 `User{userName='b', age=2}`。

原子更新字段类型

如果需要更新对象的某个字段，atomic 同样也提供了相应的原子操作类：

1. AtomicIntegeFieldUpdater：原子更新整型字段类；
2. AtomicLongFieldUpdater：原子更新长整型字段类；
3. AtomicStampedReference：原子更新引用类型，这种更新方式会带有版本号，是为了解决 CAS 的 ABA 问题，ABA 问题我们前面也讲过。

使用原子更新字段需要两步：

1. 通过静态方法`newUpdater`创建一个更新器，并且设置想要更新的类和字段；
2. 字段必须使用`public volatile`进行修饰；

以 AtomicIntegerFieldUpdater 为例来看看具体的使用：

```
public class AtomicDemo {

    private static AtomicIntegerFieldUpdater updater =
AtomicIntegerFieldUpdater.newUpdater(User.class,"age");
    public static void main(String[] args) {
        User user = new User("a", 1);
        int oldValue = updater.getAndAdd(user, 5);
        System.out.println(oldValue);
        System.out.println(updater.get(user));
    }

    static class User {
        private String userName;
        public volatile int age;

        public User(String userName, int age) {
            this.userName = userName;
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                    "userName='" + userName + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

输出结果：

```
1
6
```

从示例中可以看出，创建`AtomicIntegerFieldUpdater`是通过它提供的静态方法进行创建的，`getAndAdd`方法会将指定的字段加上输入的值，并返回相加之前的值。user 对象中 age 字段原值为 1，加 5 之后变成了 6。

小结

Java 中的 java.util.concurrent.atomic 包提供了一系列类，这些类支持原子操作（即线程安全而无需同步）在单个变量上，这大大减少了并发编程的复杂性。

原子操作类主要有这些：

1. 原子操作的基本数据类型：AtomicBoolean、AtomicInteger、AtomicLong；
2. 原子操作的数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray；
3. 原子操作的引用类型：AtomicReference、AtomicReferenceFieldUpdater、AtomicMarkableReference；

> 编辑：沉默王二，编辑前的内容主要来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency。推荐阅读：码农参上的Unsafe类详解

第二十八节：魔法类 Unsafe

前面我们在讲 CAS 和原子操作 atomic 类的时候，都讲到了 Unsafe。

Unsafe 是 Java 中一个非常特殊的类，它为 Java 提供了一种底层、"不安全"的机制来直接访问和操作内存、线程和对象。正如其名字所暗示的，Unsafe 提供了许多不安全的操作，因此它的使用应该非常小心，并限于那些确实需要使用这些底层操作的场景。

Unsafe 基础

首先我们来尝试获取一个 Unsafe 实例，如果按照`new`的方式去创建，不好意思，编译器会直接报错：

```
Unsafe() has private access in 'sun.misc.Unsafe'
```

查看 Unsafe 类的源码，可以发现它是被 final 修饰的，所以不允许被继承，并且构造方法为`private`类型，即不允许我们直接 new 实例化。不过，Unsafe 在 static 静态代码块中，以单例的方式初始化了一个 Unsafe 对象：

```
public final class Unsafe {
     private static final Unsafe theUnsafe;
     ...
     private Unsafe() {
     }
     ...
     static {
         theUnsafe = new Unsafe();
     }
 }
```

Unsafe 类提供了一个静态方法`getUnsafe`，看上去貌似可以用它来获取 Unsafe 实例：

```
@CallerSensitive
 public static Unsafe getUnsafe() {
     Class var0 = Reflection.getCallerClass();
     if (!VM.isSystemDomainLoader(var0.getClassLoader())) {
         throw new SecurityException("Unsafe");
     } else {
         return theUnsafe;
     }
 }
```

但是如果我们直接调用这个静态方法，也会抛出异常：

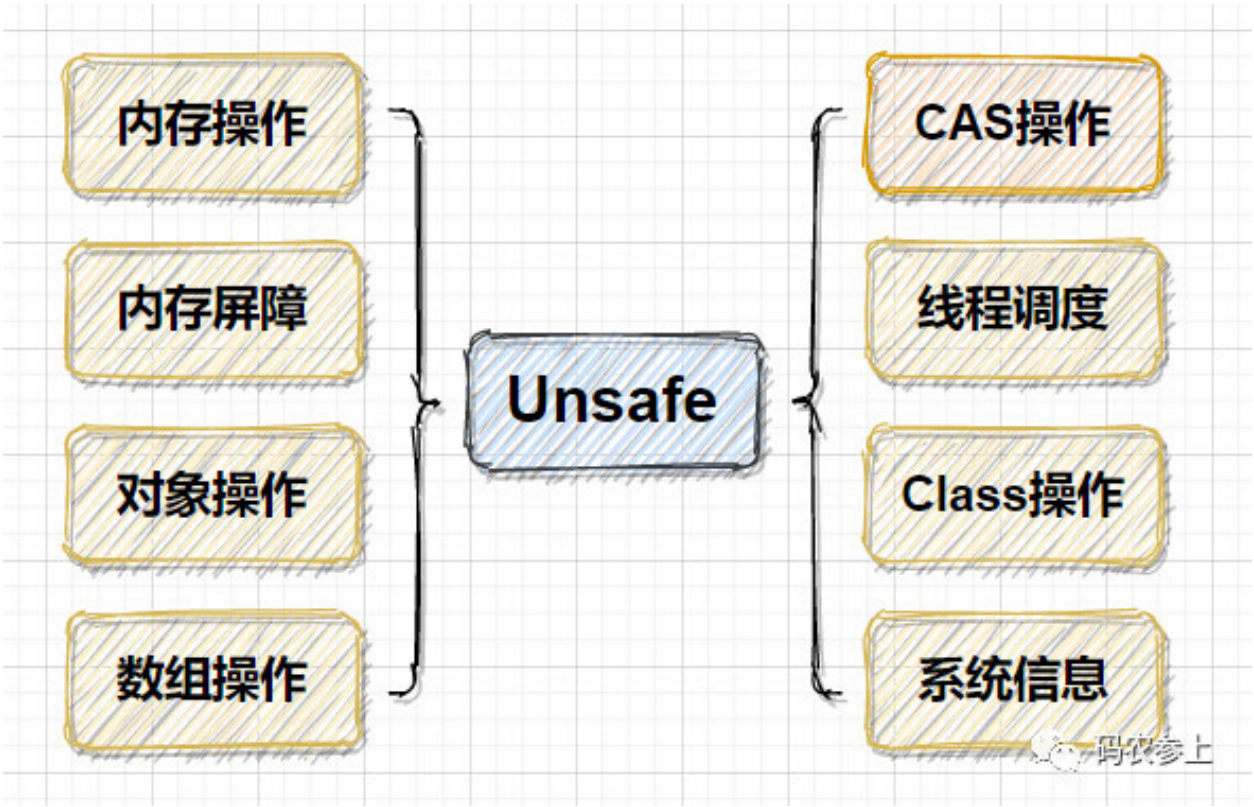
```
Exception in thread "main" java.lang.SecurityException: Unsafe
  at sun.misc.Unsafe.getUnsafe(Unsafe.java:90)
  at com.cn.test.GetUnsafeTest.main(GetUnsafeTest.java:12)
```

这是因为在`getUnsafe`方法中，会对调用者的`classLoader`进行检查，判断当前类是否由`Bootstrap classLoader`加载，如果不是的话就会抛出一个`SecurityException`异常。

也就是说，只有启动类加载器加载的类才能够调用 Unsafe 类中的方法，这是为了防止这些方法在不可信的代码中被调用。

那么，为什么要对 Unsafe 类进行这么谨慎的使用限制呢?

说到底，还是因为它实现的功能过于底层，例如直接进行内存操作、绕过 jvm 的安全检查创建对象等等，概括的来说，Unsafe 类实现的功能可以被分为下面 8 类：



创建实例

看到上面这些功能，你是不是已经有些迫不及待想要试一试了?

那么如果我们执意想要在自己的代码中调用 Unsafe 类的方法，应该怎么获取一个它的实例对象呢?

答案是利用反射获得 Unsafe 类中已经实例化完成的单例对象：

```
public static Unsafe getUnsafe() throws IllegalAccessException {
     Field unsafeField = Unsafe.class.getDeclaredField("theUnsafe");
     //Field unsafeField = Unsafe.class.getDeclaredFields()[0]; //也可以这样，作用相同
     unsafeField.setAccessible(true);
     Unsafe unsafe =(Unsafe) unsafeField.get(null);
     return unsafe;
 }
```

在获取到 Unsafe 的实例对象后，我们就可以使用它来为所欲为了，先来尝试使用它对一个对象的属性进行读写：

```
public void fieldTest(Unsafe unsafe) throws NoSuchFieldException {
     User user=new User();
     long fieldOffset = unsafe.objectFieldOffset(User.class.getDeclaredField("age"));
     System.out.println("offset:"+fieldOffset);
     unsafe.putInt(user,fieldOffset,20);
     System.out.println("age:"+unsafe.getInt(user,fieldOffset));
     System.out.println("age:"+user.getAge());
 }
```

运行代码输出如下：

```
offset:12
 age:20
 age:20
```

可以看到通过 Unsafe 类的`objectFieldOffset`方法获取到了对象中字段的偏移地址，这个偏移地址不是内存中的绝对地址而是一个相对地址，之后再通过这个偏移地址对`int`类型字段的属性值进行读写操作，通过结果也可以看到 Unsafe 的方法和类中的`get`方法获取到的值是相同的。

上面的例子中调用了 Unsafe 类的`putInt`和`getInt`方法，看一下源码中的方法：

```
public native int getInt(Object o, long offset);
 public native void putInt(Object o, long offset, int x);
```

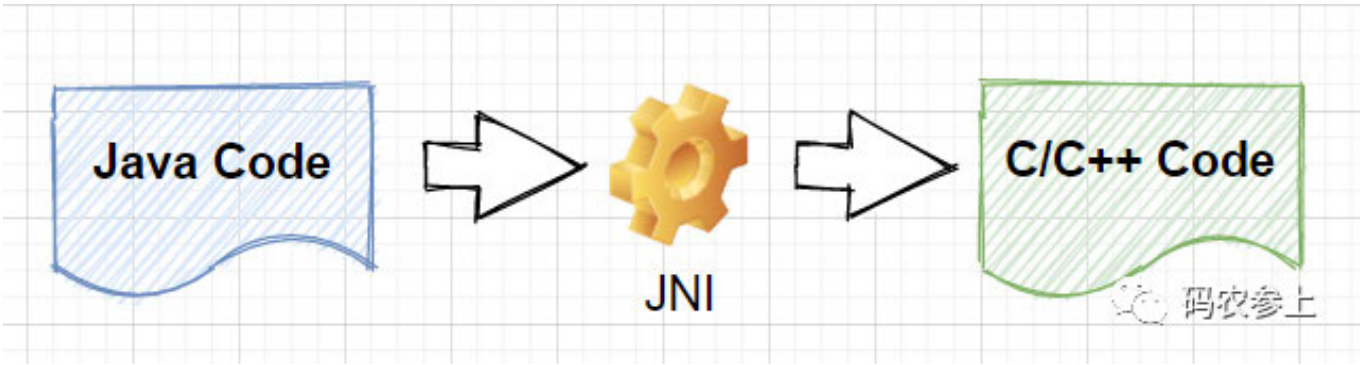
先说作用，`getInt`用于从对象的指定偏移地址处读取一个`int`，`putInt`用于在对象指定偏移地址处写入一个`int`，并且即使类中的这个属性是`private`类型的，也可以对它进行读写。

但是细心的小伙伴可能发现了，这两个方法相对于我们平常写的普通方法，多了一个native关键字修饰，并且没有具体的方法逻辑，那么它是怎么实现的呢?

native 方法

native 方法我们讲过，这里简单回顾下。

`native`方法，简单的说就是由 Java 调用非 Java 代码的接口，被调用的方法是由非 Java 语言实现的，例如它可以由 C 或 C++语言来实现，并编译成 DLL，然后直接供 Java 进行调用。`native`方法是通过 JNI（`Java Native Interface`）实现调用的，从 Java 1.1 开始 JNI 标准就是 Java 平台的一部分，它允许 Java 代码和其他语言的代码进行交互。



Unsafe 类中的很多基础方法都属于`native`方法，那么为什么要使用`native`方法呢？原因可以概括为以下几点：

- 需要用到 Java 中不具备的依赖于操作系统的特性，Java 在实现跨平台的同时要实现对底层的控制，需要借助其他语言发挥作用
- 对于其他语言已经完成的一些现成功能，可以使用 Java 直接调用
- 程序对时间敏感或对性能要求非常高时，有必要使用更加底层的语言，例如 C/C++甚至是汇编

`juc`包的很多并发工具类在实现并发机制时，都调用了`native`方法，通过 native 方法可以打破 Java 运行时的界限，能够接触到操作系统底层的某些功能。

对于同一个`native`方法，不同的操作系统可能会通过不同的方式来实现，但是对于使用者来说是透明的，最终都会得到相同的结果。

Unsafe 应用

在对 Unsafe 的基础有了一定了解后，我们来看一下它的基本应用。

1、内存操作

如果你写过`C`或者`C++`，一定对内存操作不会陌生，而 Java 是不允许直接对内存进行操作的，对象内存的分配和回收都是由`jvm`自己实现。但是在 Unsafe 中，提供的下列接口都可以直接进行内存操作：

```
//分配新的本地空间
 public native long allocateMemory(long bytes);
 //重新调整内存空间的大小
 public native long reallocateMemory(long address, long bytes);
 //将内存设置为指定值
 public native void setMemory(Object o, long offset, long bytes, byte value);
 //内存拷贝
 public native void copyMemory(Object srcBase, long srcOffset,Object destBase, long des
tOffset,long bytes);
 //清除内存
 public native void freeMemory(long address);
```

使用下面的代码进行测试：

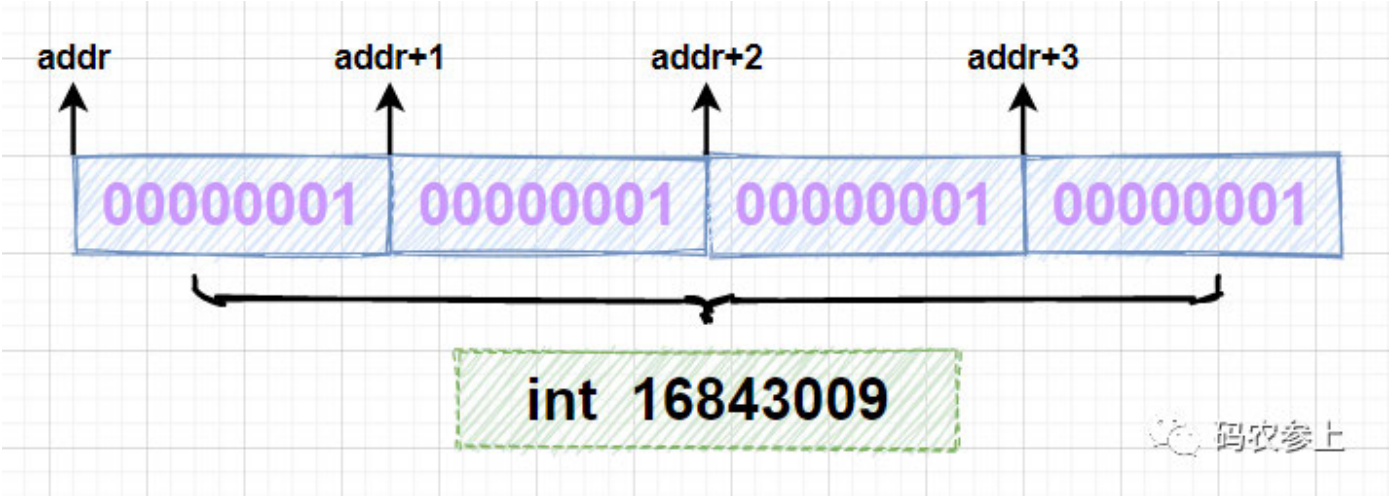
```
private void memoryTest() {
     int size = 4;
     long addr = unsafe.allocateMemory(size);
     long addr3 = unsafe.reallocateMemory(addr, size * 2);
     System.out.println("addr: "+addr);
     System.out.println("addr3: "+addr3);
     try {
         unsafe.setMemory(null,addr ,size,(byte)1);
         for (int i = 0; i < 2; i++) {
             unsafe.copyMemory(null,addr,null,addr3+size*i,4);
         }
         System.out.println(unsafe.getInt(addr));
         System.out.println(unsafe.getLong(addr3));
```

```
        }finally {
            unsafe.freeMemory(addr);
            unsafe.freeMemory(addr3);
        }
    }
```

先看结果输出：

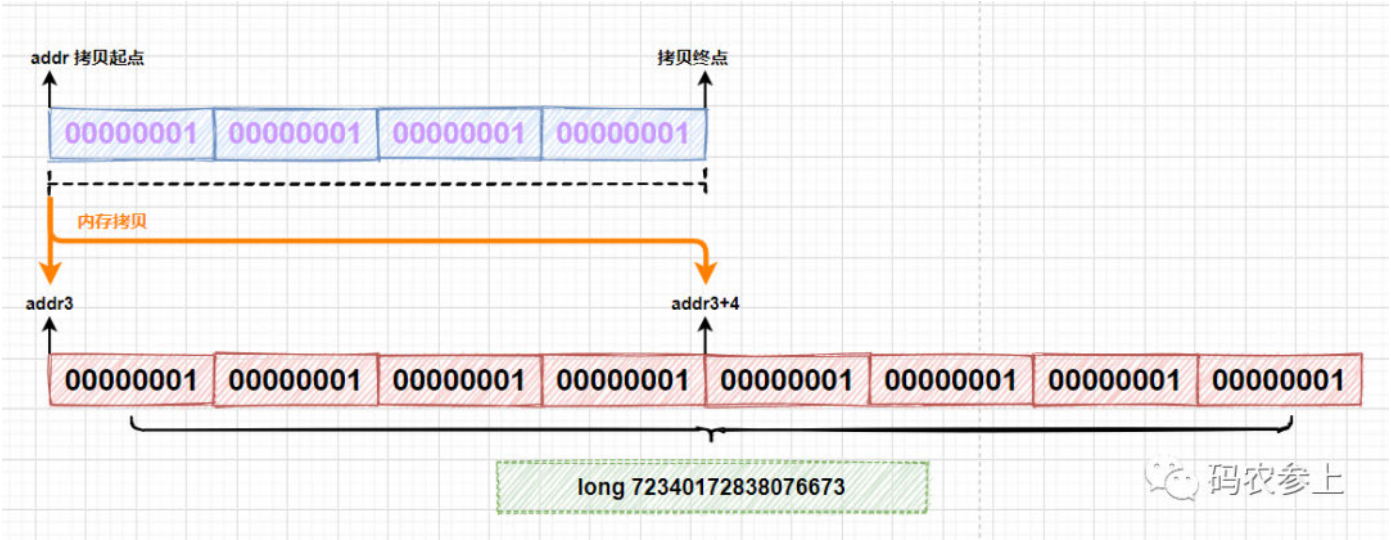
```
addr: 2433733895744
 addr3: 2433733894944
 16843009
 72340172838076673
```

分析一下运行结果，首先使用`allocateMemory`方法申请 4 字节长度的内存空间，在循环中调用`setMemory`方法向每个字节写入内容为`byte`类型的 1，当使用 Unsafe 调用`getInt`方法时，因为一个`int`型变量占 4 个字节，会一次性读取 4 个字节，组成一个`int`的值，对应的十进制结果为 16843009，可以通过图示理解这个过程：



代码中调用`reallocateMemory`方法重新分配了一块 8 字节长度的内存空间，通过比较`addr`和`addr3`可以看到和之前申请的内存地址是不同的。

在代码中的第二个 for 循环里，调用`copyMemory`方法进行了两次内存的拷贝，每次拷贝内存地址`addr`开始的 4 个字节，分别拷贝到以`addr3`和`addr3+4`开始的内存空间上：



拷贝完成后，使用`getLong`方法一次性读取 8 个字节，得到`long`类型的值为 72340172838076673。

需要注意，通过这种方式分配的内存属于堆外内存，是无法进行垃圾回收的，需要我们把这些内存当做一种资源去手动调用`freeMemory`方法进行释放，否则会产生内存泄漏。

通用的操作内存方式是在`try`中执行对内存的操作，最后在`finally`块中进行内存的释放。

2、内存屏障

在介绍内存屏障前，需要知道编译器和 CPU 会在保证程序输出结果一致的情况下，会对代码进行重排序，从指令优化角度提升性能。

而指令重排序可能会带来一个不好的结果，导致 CPU 的高速缓存和内存中数据的不一致，而内存屏障（`Memory Barrier`）就是通过组织屏障两边的指令重排序从而避免编译器和硬件的不正确优化情况。

在硬件层面上，内存屏障是 CPU 为了防止代码进行重排序而提供的指令，不同的硬件平台上实现内存屏障的方法可能并不相同。

在 Java8 中，引入了 3 个内存屏障的方法，它屏蔽了操作系统底层的差异，允许在代码中定义、并统一由 jvm 来生成内存屏障指令，来实现内存屏障的功能。Unsafe 中提供了下面三个内存屏障相关方法：

```
//禁止读操作重排序
 public native void loadFence();
 //禁止写操作重排序
 public native void storeFence();
 //禁止读、写操作重排序
 public native void fullFence();
```

内存屏障可以看做对内存随机访问操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。

以`loadFence`方法为例，它会禁止读操作重排序，保证在这个屏障之前的所有读操作都已经完成，并且将缓存数据设为无效，重新从主存中进行加载。

看到这估计很多小伙伴们会想到 volatile 关键字了，如果在字段上添加了`volatile`关键字，就能够实现字段在多线程下的可见性。

基于读内存屏障，我们也能实现相同的功能。下面定义一个线程方法，在线程中去修改`flag`标志位，注意这里的`flag`是没有被`volatile`修饰的：

```
@Getter
 class ChangeThread implements Runnable{
     /**volatile**/ boolean flag=false;
     @Override
     public void run() {
         try {
             Thread.sleep(3000);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
         System.out.println("subThread change flag to:" + flag);
```

```
            flag = true;
        }
    }
```

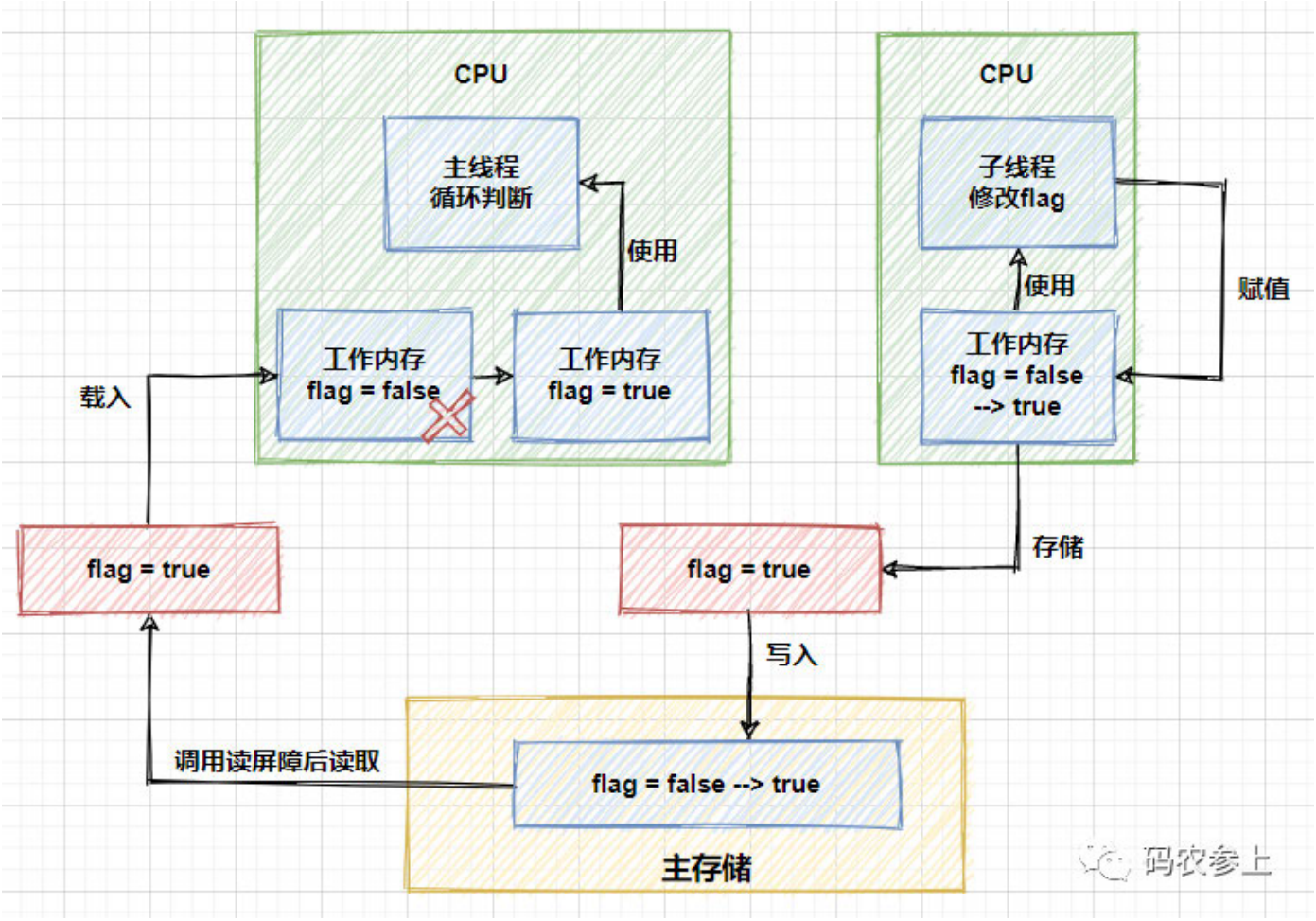
在主线程的`while`循环中，加入内存屏障，测试是否能够感知到`flag`的修改变化：

```
public static void main(String[] args){
        ChangeThread changeThread = new ChangeThread();
        new Thread(changeThread).start();
        while (true) {
            boolean flag = changeThread.isFlag();
            unsafe.loadFence(); //加入读内存屏障
            if (flag){
                System.out.println("detected flag changed");
                break;
            }
        }
        System.out.println("main thread end");
    }
```

运行结果：

```
subThread change flag to:false
 detected flag changed
 main thread end
```

而如果删掉上面代码中的`loadFence`方法，那么主线程将无法感知到`flag`发生的变化，会一直在`while`中循环。可以用图来表示上面的过程：



了解 Java 内存模型（JMM）的小伙伴们应该清楚，运行中的线程不是直接读取主内存中变量的，只能操作自己工作内存中的变量，然后同步到主内存中，并且线程的工作内存是不能共享的。

上图中的流程就是子线程借助于主内存，将修改后的结果同步给了主线程，进而修改主线程中的工作空间，跳出循环。

3、对象操作

01、对象成员属性的内存偏移量获取，以及字段属性值的修改，在上面的例子中我们已经测试过了。

除了前面的`putInt`、`getInt`方法外，Unsafe 提供了 8 种基础数据类型以及`Object`的`put`和`get`方法，并且所有的`put`方法都可以越过访问权限，直接修改内存中的数据。

阅读 openJDK 源码中的注释可以发现，基础数据类型和`Object`的读写稍有不同，基础数据类型是直接操作的属性值（`value`），而`Object`的操作则是基于引用值（`reference value`）。下面是`Object`的读写方法：

```
//在对象的指定偏移地址获取一个对象引用
 public native Object getObject(Object o, long offset);
 //在对象指定偏移地址写入一个对象引用
 public native void putObject(Object o, long offset, Object x);
```

除了对象属性的普通读写外，Unsafe 还提供了**volatile 读写**和**有序写入**方法。`volatile`读写方法的覆盖范围与普通读写相同，包含了全部基础数据类型和`Object`类型，以`int`类型为例：

```
//在对象的指定偏移地址处读取一个int值，支持volatile load语义
 public native int getIntVolatile(Object o, long offset);
 //在对象指定偏移地址处写入一个int，支持volatile store语义
 public native void putIntVolatile(Object o, long offset, int x);
```

相对于普通读写来说，`volatile`读写具有更高的成本，因为它需要保证可见性和有序性。在执行`get`操作时，会强制从主存中获取属性值，在使用`put`方法设置属性值时，会强制将值更新到主存中，从而保证这些变更对其他线程是可见的。

有序写入的方法有以下三个：

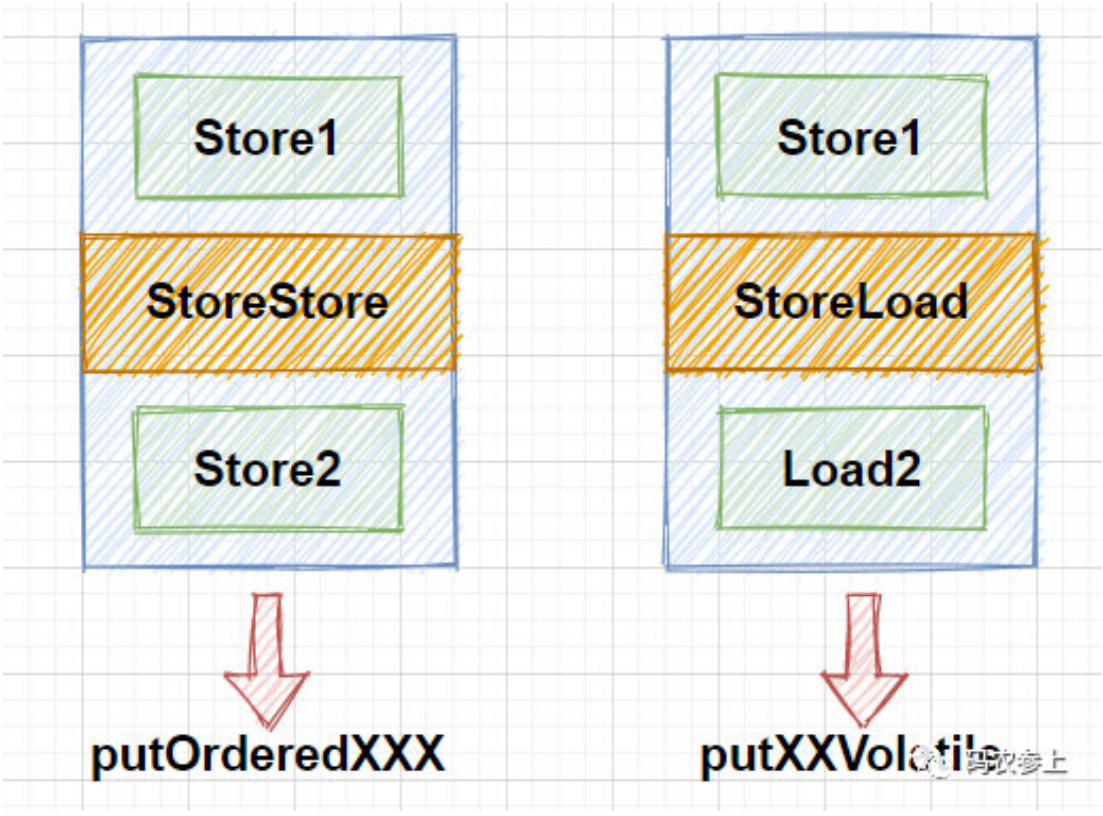
```
public native void putOrderedObject(Object o, long offset, Object x);
 public native void putOrderedInt(Object o, long offset, int x);
 public native void putOrderedLong(Object o, long offset, long x);
```

有序写入的成本相对`volatile`较低，因为它只保证写入时的有序性，而不保证可见性，也就是一个线程写入的值不能保证其他线程立即可见。

为了解决这里的差异性，需要对内存屏障的知识点再进一步进行补充，首先需要了解两个指令的概念：

- `Load`：将主内存中的数据拷贝到处理器的缓存中
- `Store`：将处理器缓存的数据刷新到主内存中

顺序写入与`volatile`写入的差别在于，在顺序写时加入的内存屏障类型为`StoreStore`类型，而在`volatile`写入时加入的内存屏障是`StoreLoad`类型，如下图所示：



在有序写入方法中，使用的是`StoreStore`屏障，该屏障确保`Store1`立刻刷新数据到内存，这一操作先于`Store2`以及后续的存储指令操作。

而在`volatile`写入中，使用的是`StoreLoad`屏障，该屏障确保`Store1`立刻刷新数据到内存，这一操作先于`Load2`及后续的装载指令，并且，`StoreLoad`屏障会使该屏障之前的所有内存访问指令，包括存储指令和访问指令全部完成之后，才执行该屏障之后的内存访问指令。

综上所述，在上面的三类写入方法中，在写入效率方面，按照`put`、`putOrder`、`putVolatile`的顺序效率逐渐降低，

02、使用 Unsafe 的`allocateInstance`方法，允许我们使用非常规的方式进行对象的实例化，首先定义一个实体类，并且在构造方法中对其成员变量进行赋值操作：

```
@Data
 public class A {
     private int b;
     public A(){
         this.b =1;
     }
 }
```

分别基于构造方法、反射以及 Unsafe 方法的不同方式创建对象进行比较：

```
public void objTest() throws Exception{
     A a1=new A();
     System.out.println(a1.getB());
     A a2 = A.class.newInstance();
     System.out.println(a2.getB());
     A a3= (A) unsafe.allocateInstance(A.class);
     System.out.println(a3.getB());
 }
```

打印结果分别为 1、1、0，说明通过`allocateInstance`方法创建对象过程中，不会调用类的构造方法。

使用这种方式创建对象时，只用到了`Class`对象，所以说如果想要跳过对象的初始化阶段或者跳过构造器的安全检查，就可以使用这种方法。

在上面的例子中，如果将 A 类的构造方法改为`private`类型，将无法通过构造方法和反射创建对象，但`allocateInstance`方法仍然有效。

4、数组操作

在 Unsafe 中，可以使用`arrayBaseOffset`方法获取数组中第一个元素的偏移地址，使用`arrayIndexScale`方法可以获取数组中元素间的偏移地址增量。使用下面的代码进行测试：

```
private void arrayTest() {
     String[] array=new String[]{"str1str1str","str2","str3"};
     int baseOffset = unsafe.arrayBaseOffset(String[].class);
     System.out.println(baseOffset);
     int scale = unsafe.arrayIndexScale(String[].class);
     System.out.println(scale);

     for (int i = 0; i < array.length; i++) {
         int offset=baseOffset+scale*i;
         System.out.println(offset+" : "+unsafe.getObject(array,offset));
     }
 }
```

上面代码的输出结果为：

```
16
 4
 16 : str1str1str
 20 : str2
 24 : str3
```

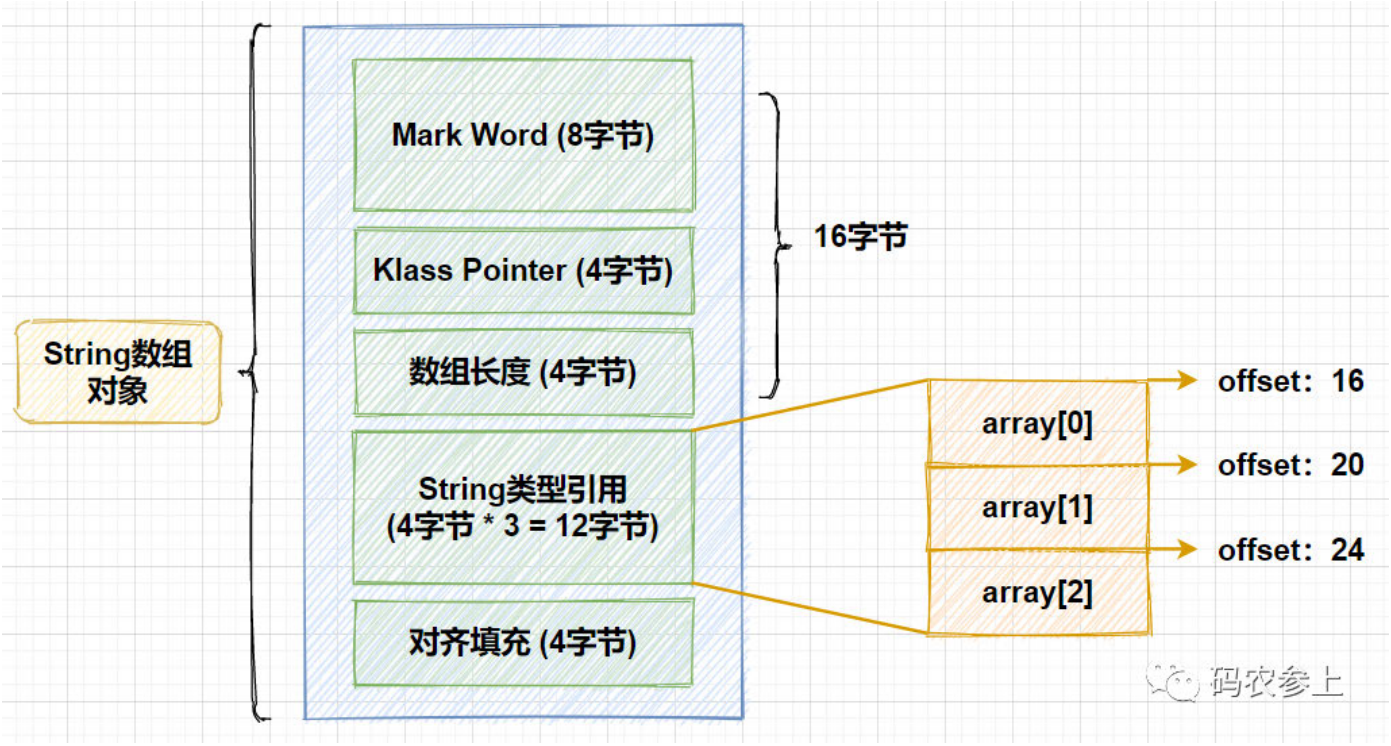
通过配合使用数组偏移首地址和各元素间偏移地址的增量，可以方便的定位到数组中的元素在内存中的位置，进而通过`getObject`方法直接获取任意位置的数组元素。

需要说明的是，`arrayIndexScale`获取的并不是数组中元素占用的大小，而是地址的增量，按照 openJDK 中的注释，可以将它翻译为**元素寻址的转换因子**（`scale factor for addressing elements`）。

在上面的例子中，第一个字符串长度为 11 字节，但其地址增量仍然为 4 字节。

那么，基于这两个值是如何实现寻址和数组元素的访问呢?

我们把上面例子中的 String 数组对象的内存布局画出来，方便大家理解：



在 String 数组对象中，对象头包含 3 部分，`mark word`标记字占用 8 字节，`klass point`类型指针占用 4 字节，数组对象特有的数组长度部分占用 4 字节，总共占用了 16 字节。

第一个 String 的引用类型相对于对象的首地址的偏移量是就 16，之后每个元素在这个基础上加 4，正好对应了我们上面代码中的寻址过程，之后再使用前面说过的`getObject`方法，通过数组对象可以获得对象在堆中的首地址，再配合对象中变量的偏移量，就能获得每一个变量的引用。

5、CAS 操作

在`juc`包的并发工具类中大量地使用了 CAS 操作，像在前面介绍的 synchronized 和 AQS 的文章中也多次提到了 CAS，其作为乐观锁在并发工具类中广泛发挥了作用。

在 Unsafe 类中，提供了`compareAndSwapObject`、`compareAndSwapInt`、`compareAndSwapLong`方法来实现的对`Object`、`int`、`long`类型的 CAS 操作。以`compareAndSwapInt`方法为例：

```
public final native boolean compareAndSwapInt(Object o, long offset,int expected,int x)
;
```

参数中`o`为需要更新的对象，`offset`是对象`o`中整形字段的偏移量，如果这个字段的值与`expected`相同，则将字段的值设为`x`这个新值，并且此更新是不可被中断的，也就是一个原子操作。下面是一个使用`compareAndSwapInt`的例子：

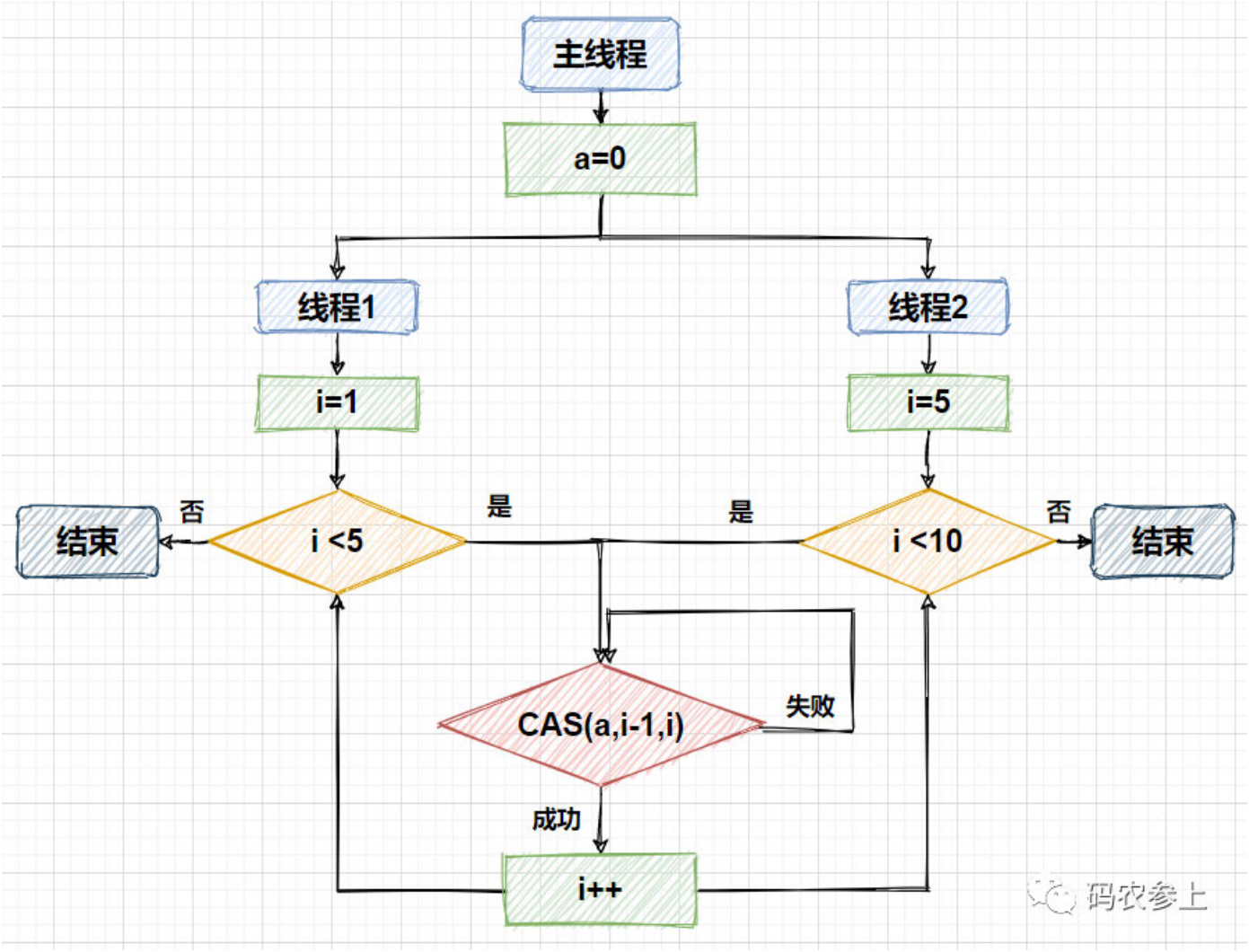
```
private volatile int a;
 public static void main(String[] args){
     CasTest casTest=new CasTest();
     new Thread(()->{
         for (int i = 1; i < 5; i++) {
             casTest.increment(i);
             System.out.print(casTest.a+" ");
         }
```

```
        }).start();
        new Thread(()->{
            for (int i = 5 ; i <10 ; i++) {
                casTest.increment(i);
                System.out.print(casTest.a+" ");
            }
        }).start();
    }

    private void increment(int x){
        while (true){
            try {
                long fieldOffset = unsafe.objectFieldOffset(CasTest.class.getDeclaredField
("a"));
                if (unsafe.compareAndSwapInt(this,fieldOffset,x-1,x))
                    break;
            } catch (NoSuchFieldException e) {
                e.printStackTrace();
            }
        }
    }
```

运行代码会依次输出：

```
1 2 3 4 5 6 7 8 9
```

在上面的例子中，使用两个线程去修改`int`型属性`a`的值，并且只有在`a`的值等于传入的参数`x`减一时，才会将`a`的值变为`x`，也就是实现对`a`的加一的操作。流程如下所示：



需要注意的是，在调用`compareAndSwapInt`方法后，会直接返回`true`或`false`的修改结果，因此需要我们在代码中手动添加自旋的逻辑。

在`AtomicInteger`类的设计中，也是采用了将`compareAndSwapInt`的结果作为循环条件，直至修改成功才退出死循环的方式来实现的原子性的自增操作。

6、线程调度

Unsafe 类中提供了`park`、`unpark`、`monitorEnter`、`monitorExit`、`tryMonitorEnter`方法进行线程调度，在前面介绍 AQS 的文章中我们提到过使用 LockSupport 挂起或唤醒指定线程。这个类我们前面也讲到了，这里再回顾一下。

看一下`LockSupport`的源码，可以看到它也是调用的 Unsafe 类中的方法：

```
public static void park(Object blocker) {
     Thread t = Thread.currentThread();
     setBlocker(t, blocker);
     UNSAFE.park(false, 0L);
     setBlocker(t, null);
 }
 public static void unpark(Thread thread) {
     if (thread != null)
         UNSAFE.unpark(thread);
 }
```

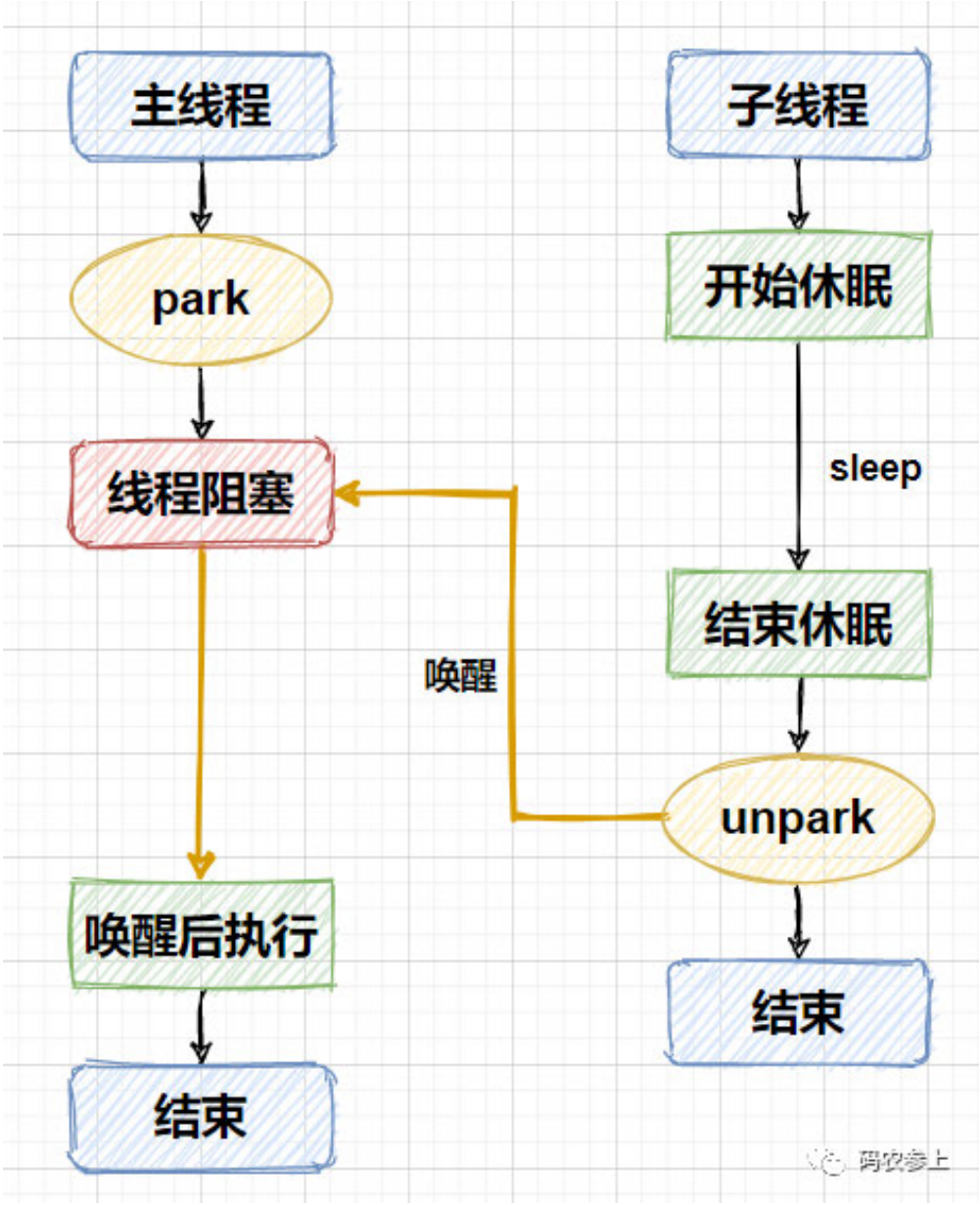
LockSupport 的`park`方法调用了 Unsafe 的`park`方法来阻塞当前线程，此方法将线程阻塞后就不会继续往后执行，直到有其他线程调用`unpark`方法唤醒当前线程。下面的例子对 Unsafe 的这两个方法进行测试：

```
public static void main(String[] args) {
     Thread mainThread = Thread.currentThread();
     new Thread(()->{
         try {
             TimeUnit.SECONDS.sleep(5);
             System.out.println("subThread try to unpark mainThread");
             unsafe.unpark(mainThread);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
     }).start();

     System.out.println("park main mainThread");
     unsafe.park(false,0L);
     System.out.println("unpark mainThread success");
 }
```

程序输出为：

```
park main mainThread
 subThread try to unpark mainThread
 unpark mainThread success
```

程序运行的流程也比较容易看懂，子线程开始运行后先进行睡眠，确保主线程能够调用`park`方法阻塞自己，子线程在睡眠 5 秒后，调用`unpark`方法唤醒主线程，使主线程能继续向下执行。整个流程如下图所示：



此外，Unsafe 源码中`monitor`相关的三个方法已经被标记为`deprecated`，不建议被使用：

```
//获得对象锁
 @Deprecated
 public native void monitorEnter(Object var1);
 //释放对象锁
 @Deprecated
 public native void monitorExit(Object var1);
 //尝试获得对象锁
 @Deprecated
 public native boolean tryMonitorEnter(Object var1);
```

`monitorEnter`方法用于获得对象锁，`monitorExit`用于释放对象锁，如果对一个没有被`monitorEnter`加锁的对象执行此方法，会抛出`IllegalMonitorStateException`异常。`tryMonitorEnter`方法尝试获取对象锁，如果成功则返回`true`，反之返回`false`。

7、Class 操作

Unsafe 对`Class`的相关操作主要包括类加载和静态变量的操作方法。

01、静态属性读取相关的方法：

```
//获取静态属性的偏移量
 public native long staticFieldOffset(Field f);
 //获取静态属性的对象指针
 public native Object staticFieldBase(Field f);
 //判断类是否需要实例化（用于获取类的静态属性前进行检测）
 public native boolean shouldBeInitialized(Class<?> c);
```

创建一个包含静态属性的类，进行测试：

```
@Data
 public class User {
     public static String name="Hydra";
     int age;
 }
 private void staticTest() throws Exception {
     User user=new User();
     System.out.println(unsafe.shouldBeInitialized(User.class));
     Field sexField = User.class.getDeclaredField("name");
     long fieldOffset = unsafe.staticFieldOffset(sexField);
     Object fieldBase = unsafe.staticFieldBase(sexField);
     Object object = unsafe.getObject(fieldBase, fieldOffset);
     System.out.println(object);
 }
```

运行结果：

```
false
 Hydra
```

在 Unsafe 的对象操作中，我们学习了通过`objectFieldOffset`方法获取对象属性偏移量并基于它对变量的值进行存取，但是它不适用于类中的静态属性，这时候就需要使用`staticFieldOffset`方法。

在上面的代码中，获取`Field`对象需要依赖`Class`，而获取静态变量的属性时则不再依赖于`Class`。

在上面的代码中，首先创建一个`User`对象，这是因为如果一个类没有被实例化，那么它的静态属性也不会被初始化，最后获取的字段属性将是`null`。所以在获取静态属性前，需要调用`shouldBeInitialized`方法，判断在获取前是否需要初始化这个类。如果删除创建 User 对象的语句，运行结果会变为：

```
true
 null
```

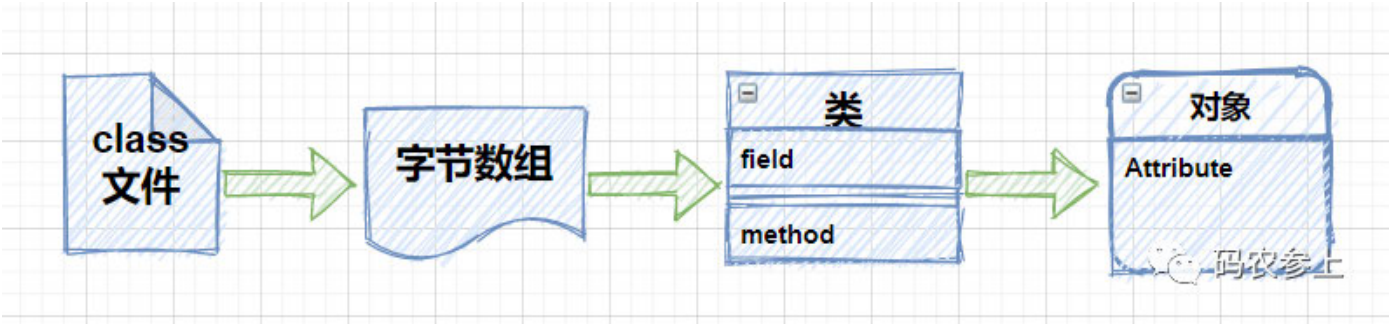
02、使用`defineClass`方法允许程序在运行时动态地创建一个类，方法定义如下：

```
public native Class<?> defineClass(String name, byte[] b, int off, int len,
                                   ClassLoader loader,ProtectionDomain protectionDomai
n);
```

在实际使用过程中，可以只传入字节数组、起始字节的下标以及读取的字节长度，默认情况下，类加载器（`ClassLoader`）和保护域（`ProtectionDomain`）来源于调用此方法的实例。下面的例子中实现了反编译生成后的 class 文件的功能：

```
private static void defineTest() {
     String fileName="F:\\workspace\\unsafe-
test\\target\\classes\\com\\cn\\model\\User.class";
     File file = new File(fileName);
     try(FileInputStream fis = new FileInputStream(file)) {
         byte[] content=new byte[(int)file.length()];
         fis.read(content);
         Class clazz = unsafe.defineClass(null, content, 0, content.length, null, null)
;
         Object o = clazz.newInstance();
         Object age = clazz.getMethod("getAge").invoke(o, null);
         System.out.println(age);
     } catch (Exception e) {
         e.printStackTrace();
     }
 }
```

在上面的代码中，首先读取了一个`class`文件并通过文件流将它转化为字节数组，之后使用`defineClass`方法动态的创建了一个类，并在后续完成了它的实例化工作，流程如下图所示，并且通过这种方式创建的类，会跳过 JVM 的所有安全检查。



除了`defineClass`方法外，Unsafe 还提供了一个`defineAnonymousClass`方法：

```
public native Class<?> defineAnonymousClass(Class<?
> hostClass, byte[] data, Object[] cpPatches);
```

使用该方法可以动态的创建一个匿名类，Lambda表达式中就是使用 ASM 动态生成字节码的，然后利用该方法定义实现相应的函数式接口的匿名类。

在 JDK 15 发布的新特性中，在隐藏类（`Hidden classes`）一条中，指出将在未来的版本中弃用 Unsafe 的`defineAnonymousClass`方法。

8、系统信息

Unsafe 中提供的`addressSize`和`pageSize`方法用于获取系统信息，调用`addressSize`方法会返回系统指针的大小，如果在 64 位系统下默认会返回 8，而 32 位系统则会返回 4。调用 pageSize 方法会返回内存页的大小，值为 2 的整数幂。使用下面的代码可以直接进行打印：

```
private void systemTest() {
     System.out.println(unsafe.addressSize());
     System.out.println(unsafe.pageSize());
 }
```

执行结果：

```
8
 4096
```

这两个方法的应用场景比较少，在`java.nio.Bits`类中，在使用`pageCount`计算所需的内存页的数量时，调用了`pageSize`方法获取内存页的大小。另外，在使用`copySwapMemory`方法拷贝内存时，调用了`addressSize`方法，检测 32 位系统的情况。

小结

在本文中，我们首先介绍了 Unsafe 的基本概念、工作原理，并在此基础上，对它的 API 进行了说明与实践。

相信大家通过这一过程，能够发现 Unsafe 在某些场景下，确实能够为我们提供编程便利。但在使用这些便利时，确实存在着一些安全上的隐患，在我看来，一项技术具有不安全因素并不可怕，可怕的是它在使用过程中被滥用。

尽管之前有传言说会在 Java9 中移除 Unsafe 类，不过它还是照样已经存活到了 JDK 16，按照存在即合理的逻辑，只要使用得当，它还是能给我们带来不少的帮助，因此最后还是建议大家，在使用 Unsafe 的过程中一定要做到使用谨慎使用、避免滥用。

> 编辑：沉默王二，编辑前的内容主要来自于朋友码农参上的公众号文章，写得非常好，推荐关注。

第二十九节：通信工具类

JDK 中提供了一些并发编程中常用的通信工具类以供我们开发者使用，比如说 CountDownLatch,Semaphore,Exchanger,CyclicBarrier,Phaser。

它们都在 JUC 包下。先总体概括一下都有哪些工具类，它们有什么作用，然后再分别介绍它们的主要使用方法和原理。

类	作用
Semaphore	限制线程的数量
Exchanger	两个线程交换数据
CountDownLatch	线程等待直到计数器减为 0 时开始工作
CyclicBarrier	作用跟 CountDownLatch 类似，但是可以重复使用
Phaser	增强的 CyclicBarrier

Semaphore

Semaphore 翻译过来是信号的意思。顾名思义，这个工具类提供的功能就是多个线程彼此“传信号”。而这个“信号”是一个`int`类型的数据，也可以看成是一种“资源”。

可以在构造方法中传入初始资源总数，以及是否使用“公平”的同步器。默认情况下，是非公平的。

```
// 默认情况下使用非公平
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

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

最主要的方法是 acquire 方法和 release 方法。`acquire()`方法会申请一个 permit，而 release 方法会释放一个 permit。当然，你也可以申请多个`acquire(int permits)`或者释放多个`release(int permits)`。

每次 acquire，permits 就会减少一个或者多个。如果减少到了 0，再有其他线程来 acquire，那就要阻塞这个线程直到有其它线程 release permit 为止。

Semaphore 使用案例

Semaphore 往往用于资源有限的场景中，去限制线程的数量。举个例子，我想限制同时只能有 3 个线程在工作：

```
public class SemaphoreDemo {
    static class MyThread implements Runnable {

        private int value;
        private Semaphore semaphore;

        public MyThread(int value, Semaphore semaphore) {
            this.value = value;
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                semaphore.acquire(); // 获取permit
                System.out.println(String.format("当前线程是%d，还剩%d个资源，还有%d个线程在
等待",
                        value, semaphore.availablePermits(),
semaphore.getQueueLength()));
                // 睡眠随机时间，打乱释放顺序
                Random random =new Random();
                Thread.sleep(random.nextInt(1000));
                System.out.println(String.format("线程%d释放了资源", value));
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally{
                semaphore.release(); // 释放permit
            }
        }
    }
```

```
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3);
        for (int i = 0; i < 10; i++) {
            new Thread(new MyThread(i, semaphore)).start();
        }
    }
}
```

输出：

```
当前线程是 1, 还剩 2 个资源，还有 0 个线程在等待
当前线程是 0, 还剩 1 个资源，还有 0 个线程在等待
当前线程是 6, 还剩 0 个资源，还有 0 个线程在等待
线程 6 释放了资源
当前线程是 2, 还剩 0 个资源，还有 6 个线程在等待
线程 2 释放了资源
当前线程是 4, 还剩 0 个资源，还有 5 个线程在等待
线程 0 释放了资源
当前线程是 7, 还剩 0 个资源，还有 4 个线程在等待
线程 1 释放了资源
当前线程是 8, 还剩 0 个资源，还有 3 个线程在等待
线程 7 释放了资源
当前线程是 5, 还剩 0 个资源，还有 2 个线程在等待
线程 4 释放了资源
当前线程是 3, 还剩 0 个资源，还有 1 个线程在等待
线程 8 释放了资源
当前线程是 9, 还剩 0 个资源，还有 0 个线程在等待
线程 9 释放了资源
线程 5 释放了资源
线程 3 释放了资源
```

可以看到，在这次运行中，最开始是 1, 0, 6 这三个线程获得了资源，而其它线程进入了等待队列。然后当某个线程释放资源后，就会有等待队列中的线程获得资源。

当然，Semaphore 默认的 acquire 方法是会让线程进入等待队列，且抛出异常中断。但它还有一些方法可以忽略中断或不进入阻塞队列：

```
// 忽略中断
public void acquireUninterruptibly()
public void acquireUninterruptibly(int permits)

// 不进入等待队列，底层使用CAS
public boolean tryAcquire
public boolean tryAcquire(int permits)
public boolean tryAcquire(int permits, long timeout, TimeUnit unit)
        throws InterruptedException
public boolean tryAcquire(long timeout, TimeUnit unit)
```

Semaphore 原理

Semaphore 内部有一个继承了 AQS 的同步器 Sync，重写了`tryAcquireShared`方法。在这个方法里，会去尝试获取资源。

如果获取失败（想要的资源数量小于目前已有的资源数量），就会返回一个负数（代表尝试获取资源失败）。然后当前线程就会进入 AQS 的等待队列。

Exchanger

Exchanger 类用于两个线程交换数据。它支持泛型，也就是说你可以在两个线程之间传送任何数据。先来一个案例看看如何使用，比如两个线程之间想要传送字符串：

```
public class ExchangerDemo {
    public static void main(String[] args) throws InterruptedException {
        Exchanger<String> exchanger = new Exchanger<>();

        new Thread(() -> {
            try {
                System.out.println("这是线程A，得到了另一个线程的数据："
                        + exchanger.exchange("这是来自线程A的数据"));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        System.out.println("这个时候线程A是阻塞的，在等待线程B的数据");
        Thread.sleep(1000);

        new Thread(() -> {
            try {
                System.out.println("这是线程B，得到了另一个线程的数据："
                        + exchanger.exchange("这是来自线程B的数据"));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

输出：

```
这个时候线程 A 是阻塞的，在等待线程 B 的数据
这是线程 B，得到了另一个线程的数据：这是来自线程 A 的数据
这是线程 A，得到了另一个线程的数据：这是来自线程 B 的数据
```

可以看到，当一个线程调用 exchange 方法后，会处于阻塞状态，只有当另一个线程也调用了 exchange 方法，它才会继续执行。

看源码可以发现它是使用 park/unpark 来实现等待状态切换的，但是在使用 park/unpark 方法之前，使用了 CAS 检查，估计是为了提高性能。

因为 Exchanger 支持泛型，所以我们可以传输任何的数据，比如 IO 流或者 IO 缓存。根据 JDK 里面注释的说法，可以总结为一下特性：

- 此类提供对外的操作是同步的；
- 用于成对出现的线程之间交换数据；
- 可以视作双向的同步队列；
- 可应用于基因算法、流水线设计等场景。

Exchanger 类还有一个有超时参数的方法，如果在指定时间内没有另一个线程调用 exchange，就会抛出一个超时异常。

```
public V exchange(V x, long timeout, TimeUnit unit)
```

那么问题来了，Exchanger 只能是两个线程交换数据吗？那三个调用同一个实例的 exchange 方法会发生什么呢？答案是只有前两个线程会交换数据，第三个线程会进入阻塞状态。

需要注意的是，exchange 是可以重复使用的。也就是说。两个线程可以使用 Exchanger 在内存中不断地再交换数据。

CountDownLatch

先来解读一下 CountDownLatch 这个类名的意义。CountDown 代表计数递减，Latch 是“门闩”的意思。也有人把它称为“屏障”。而 CountDownLatch 这个类的作用也很贴合这个名字的意义，假设某个线程在执行任务之前，需要等待其它线程完成一些前置任务，必须等所有的前置任务都完成，才能开始执行本线程的任务。

CountDownLatch 的方法也很简单，如下：

```
// 构造方法：
public CountDownLatch(int count)

public void await() // 等待
public boolean await(long timeout, TimeUnit unit) // 超时等待
public void countDown() // count - 1
public long getCount() // 获取当前还有多少count
```

CountDownLatch 案例

我们知道，玩游戏的时候，在游戏真正开始之前，一般会等待一些前置任务完成，比如“加载地图数据”，“加载人物模型”，“加载背景音乐”等等。只有当所有的东西都加载完成后，玩家才能真正进入游戏。下面我们就来模拟一下这个 demo。

```
public class CountDownLatchDemo {
    // 定义前置任务线程
    static class PreTaskThread implements Runnable {

        private String task;
```

```
        private CountDownLatch countDownLatch;

        public PreTaskThread(String task, CountDownLatch countDownLatch) {
            this.task = task;
            this.countDownLatch = countDownLatch;
        }

        @Override
        public void run() {
            try {
                Random random = new Random();
                Thread.sleep(random.nextInt(1000));
                System.out.println(task + " - 任务完成");
                countDownLatch.countDown();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        // 假设有三个模块需要加载
        CountDownLatch countDownLatch = new CountDownLatch(3);

        // 主任务
        new Thread(() -> {
            try {
                System.out.println("等待数据加载...");
                System.out.println(String.format("还有%d个前置任务",
countDownLatch.getCount()));
                countDownLatch.await();
                System.out.println("数据加载完成，正式开始游戏！");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        // 前置任务
        new Thread(new PreTaskThread("加载地图数据", countDownLatch)).start();
        new Thread(new PreTaskThread("加载人物模型", countDownLatch)).start();
        new Thread(new PreTaskThread("加载背景音乐", countDownLatch)).start();
    }
}
```

输出：

> 等待数据加载...
> 还有 3 个前置任务
> 加载人物模型 - 任务完成
> 加载背景音乐 - 任务完成
> 加载地图数据 - 任务完成
> 数据加载完成，正式开始游戏！

CountDownLatch 原理

其实 CountDownLatch 类的原理挺简单的，内部同样是一个继承了 AQS 的实现类 Sync，且实现起来还很简单，可能是 JDK 里面 AQS 的子类中最简单的实现了，有兴趣的小伙伴可以去看看这个内部类的源码。

需要注意的是构造器中的**计数值（count）实际上就是闭锁需要等待的线程数量**。这个值只能被设置一次，而且 CountDownLatch**没有提供任何机制去重新设置这个计数值**。

CyclicBarrier

CyclicBarrirer 从名字上来理解是“循环屏障”的意思。前面提到了 CountDownLatch 一旦计数值`count`被降为 0 后，就不能再重新设置了，它只能起一次“屏障”的作用。而 CyclicBarrier 拥有 CountDownLatch 的所有功能，还可以使用`reset()`方法重置屏障。

如果参与者（线程）在等待的过程中，Barrier 被破坏，就会抛出 BrokenBarrierException。可以用`isBroken()`方法检测 Barrier 是否被破坏。

1. 如果有线程已经处于等待状态，调用 reset 方法会导致已经在等待的线程出现 BrokenBarrierException 异常。并且由于出现了 BrokenBarrierException，将会导致始终无法等待。
2. 如果在等待的过程中，线程被中断，会抛出 InterruptedException 异常，并且这个异常会传播到其他所有的线程。
3. 如果在执行屏障操作过程中发生异常，则该异常将传播到当前线程中，其他线程会抛出 BrokenBarrierException，屏障被损坏。
4. 如果超出指定的等待时间，当前线程会抛出 TimeoutException 异常，其他线程会抛出 BrokenBarrierException 异常。

CyclicBarrier 案例

我们同样用玩游戏的例子。如果玩一个游戏有多个“关卡”，那使用 CountDownLatch 显然不太合适，因为需要为每个关卡都创建一个实例。那我们可以使用 CyclicBarrier 来实现每个关卡的数据加载等待功能。

```
public class CyclicBarrierDemo {
    static class PreTaskThread implements Runnable {

        private String task;
        private CyclicBarrier cyclicBarrier;

        public PreTaskThread(String task, CyclicBarrier cyclicBarrier) {
            this.task = task;
            this.cyclicBarrier = cyclicBarrier;
        }
```

```
        @Override
        public void run() {
            // 假设总共三个关卡
            for (int i = 1; i < 4; i++) {
                try {
                    Random random = new Random();
                    Thread.sleep(random.nextInt(1000));
                    System.out.println(String.format("关卡%d的任务%s完成", i, task));
                    cyclicBarrier.await();
                } catch (InterruptedException | BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(3, () -> {
            System.out.println("本关卡所有前置任务完成，开始游戏...");
        });

        new Thread(new PreTaskThread("加载地图数据", cyclicBarrier)).start();
        new Thread(new PreTaskThread("加载人物模型", cyclicBarrier)).start();
        new Thread(new PreTaskThread("加载背景音乐", cyclicBarrier)).start();
    }
}
```

输出：

```
关卡 1 的任务加载地图数据完成
关卡 1 的任务加载背景音乐完成
关卡 1 的任务加载人物模型完成
本关卡所有前置任务完成，开始游戏...
关卡 2 的任务加载地图数据完成
关卡 2 的任务加载背景音乐完成
关卡 2 的任务加载人物模型完成
本关卡所有前置任务完成，开始游戏...
关卡 3 的任务加载人物模型完成
关卡 3 的任务加载地图数据完成
关卡 3 的任务加载背景音乐完成
本关卡所有前置任务完成，开始游戏...
```

注意这里跟 CountDownLatch 的代码有一些不同。CyclicBarrier 没有分为`await()`和`countDown()`，而是只有单独的一个`await()`方法。

一旦调用 await 方法的线程数量等于构造方法中传入的任务总量（这里是 3），就代表达到屏障了。CyclicBarrier 允许我们在达到屏障的时候可以执行一个任务，可以在构造方法传入一个 Runnable 类型的对象。

上述案例就是在达到屏障时，输出“本关卡所有前置任务完成，开始游戏...”。

```
// 构造方法
public CyclicBarrier(int parties) {
    this(parties, null);
}
public CyclicBarrier(int parties, Runnable barrierAction) {
    // 具体实现
}
```

CyclicBarrier 原理

CyclicBarrier 虽说功能与 CountDownLatch 类似，但是实现原理却完全不同，CyclicBarrier 内部使用的是 Lock + Condition 实现的等待/通知模式。详情可以查看这个方法的源码：

```
private int dowait(boolean timed, long nanos)
```

Phaser

Phaser 是 Java 7 中引入的一个并发同步工具，它提供了对动态数量的线程的同步能力，这与 CyclicBarrier 和 CountDownLatch 不同，因为它们都需要预先知道等待的线程数量。Phaser 是多阶段的，意味着它可以同步不同阶段的多个操作。

前面我们介绍了 CyclicBarrier，可以发现它在构造方法里传入了“任务总量”`parties`之后，就不能修改这个值了，并且每次调用`await()`方法也只能消耗一个`parties`计数。但 Phaser 可以动态地调整任务总量！

Phaser 是阶段性的，所以它有一个内部的阶段计数器。每当我们到达一个阶段的结尾时，Phaser 会自动前进到下一个阶段。

名词解释：

- Party：Phaser 的上下文中，一个 party 可以是一个线程，也可以是一个任务。当我们在 Phaser 上注册一个 party 时，Phaser 会递增它的参与者数量。
- arrive：对应一个 party 的状态，初始时是 unarrived，当调用`arriveAndAwaitAdvance()`或者`arriveAndDeregister()`进入 arrive 状态，可以通过`getUnarrivedParties()`获取当前未到达的数量。
- register：注册一个新的 party 到 Phaser。
- deRegister：减少一个 party。
- phase：阶段，当所有注册的 party 都 arrive 之后，将会调用 Phaser 的`onAdvance()`方法来判断是否要进入下一阶段。

Phaser 的终止有两种途径，Phaser 维护的线程执行完毕或者`onAdvance()`返回`true`。

Phaser 案例

还是游戏的案例。假设我们游戏有三个关卡，但只有第一个关卡有新手教程，需要加载新手教程模块。但后面的第二个关卡和第三个关卡都不需要。我们可以用 Phaser 来做这个需求。

代码：

```
public class PhaserDemo {
    static class PreTaskThread implements Runnable {
```

```
        private String task;
        private Phaser phaser;

        public PreTaskThread(String task, Phaser phaser) {
            this.task = task;
            this.phaser = phaser;
        }

        @Override
        public void run() {
            for (int i = 1; i < 4; i++) {
                try {
                    // 第二次关卡起不加载NPC，跳过
                    if (i >= 2 && "加载新手教程".equals(task)) {
                        continue;
                    }
                    Random random = new Random();
                    Thread.sleep(random.nextInt(1000));
                    System.out.println(String.format("关卡%d，需要加载%d个模块，当前模块【%s】",
                            i, phaser.getRegisteredParties(), task));

                    // 从第二个关卡起，不加载NPC
                    if (i == 1 && "加载新手教程".equals(task)) {
                        System.out.println("下次关卡移除加载【新手教程】模块");
                        phaser.arriveAndDeregister(); // 移除一个模块
                    } else {
                        phaser.arriveAndAwaitAdvance();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        Phaser phaser = new Phaser(4) {
            @Override
            protected boolean onAdvance(int phase, int registeredParties) {
                System.out.println(String.format("第%d次关卡准备完成", phase + 1));
                return phase == 3 || registeredParties == 0;
            }
        };

        new Thread(new PreTaskThread("加载地图数据", phaser)).start();
        new Thread(new PreTaskThread("加载人物模型", phaser)).start();
        new Thread(new PreTaskThread("加载背景音乐", phaser)).start();
```

```
            new Thread(new PreTaskThread("加载新手教程", phaser)).start();
        }
    }
```

输出：

```
关卡 1，需要加载 4 个模块，当前模块【加载背景音乐】
关卡 1，需要加载 4 个模块，当前模块【加载新手教程】
下次关卡移除加载【新手教程】模块
关卡 1，需要加载 3 个模块，当前模块【加载地图数据】
关卡 1，需要加载 3 个模块，当前模块【加载人物模型】
第 1 次关卡准备完成
关卡 2，需要加载 3 个模块，当前模块【加载地图数据】
关卡 2，需要加载 3 个模块，当前模块【加载背景音乐】
关卡 2，需要加载 3 个模块，当前模块【加载人物模型】
第 2 次关卡准备完成
关卡 3，需要加载 3 个模块，当前模块【加载人物模型】
关卡 3，需要加载 3 个模块，当前模块【加载地图数据】
关卡 3，需要加载 3 个模块，当前模块【加载背景音乐】
第 3 次关卡准备完成
```

这里要注意关卡 1 的输出，在“加载新手教程”线程中调用了`arriveAndDeregister()`减少一个 party 之后，后面的线程使用`getRegisteredParties()`得到的是已经被修改后的 parties 了。但是当前这个阶段(phase)，仍然是需要 4 个 parties 都 arrive 才触发屏障的。从下一个阶段开始，才需要 3 个 parties 都 arrive 就触发屏障。

Phaser 类用来控制某个阶段的线程数量很有用，但它并不在意这个阶段具体有哪些线程 arrive，只要达到它当前阶段的 parties 值，就触发屏障。所以我这里的案例虽然制定了特定的线程（加载新手教程）来更直观地表述 Phaser 的功能，但其实 Phaser 是没有分辨具体是哪个线程的功能的，它在意的只是数量，这一点需要大家注意。

Phaser 原理

Phaser 类的原理相比起来要复杂得多。它内部使用了两个基于 Fork-Join 框架的原子类辅助：

```
private final AtomicReference<QNode> evenQ;
private final AtomicReference<QNode> oddQ;

static final class QNode implements ForkJoinPool.ManagedBlocker {
  // 实现代码
}
```

有兴趣的小伙伴可以去看看 JDK 源代码，这里不做过多叙述。

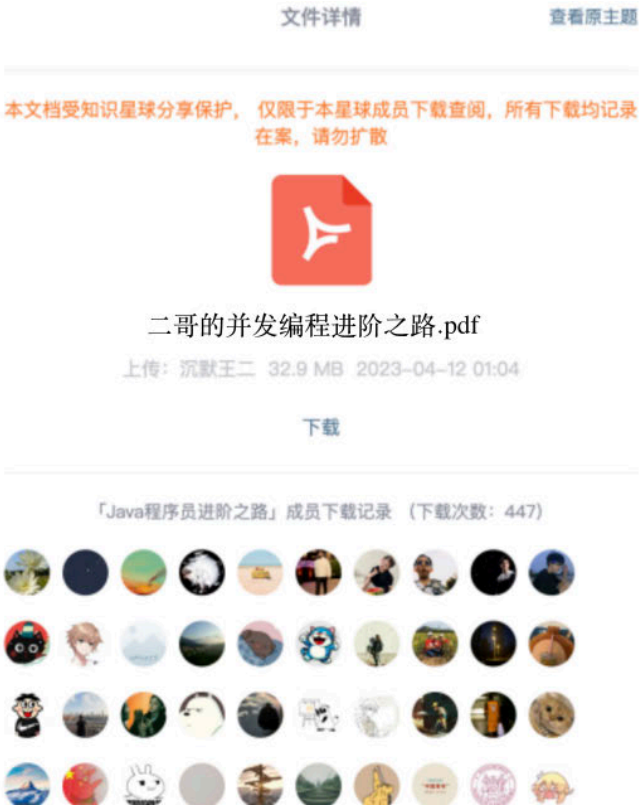
小结

总的来说，CountDownLatch，CyclicBarrier，Phaser 是一个比一个强大，但也一个比一个复杂，需要根据自己的业务需求合理选择。

> 编辑：沉默王二，部分内容来源于朋友小七萤火虫开源的这个仓库：深入浅出 Java 多线程

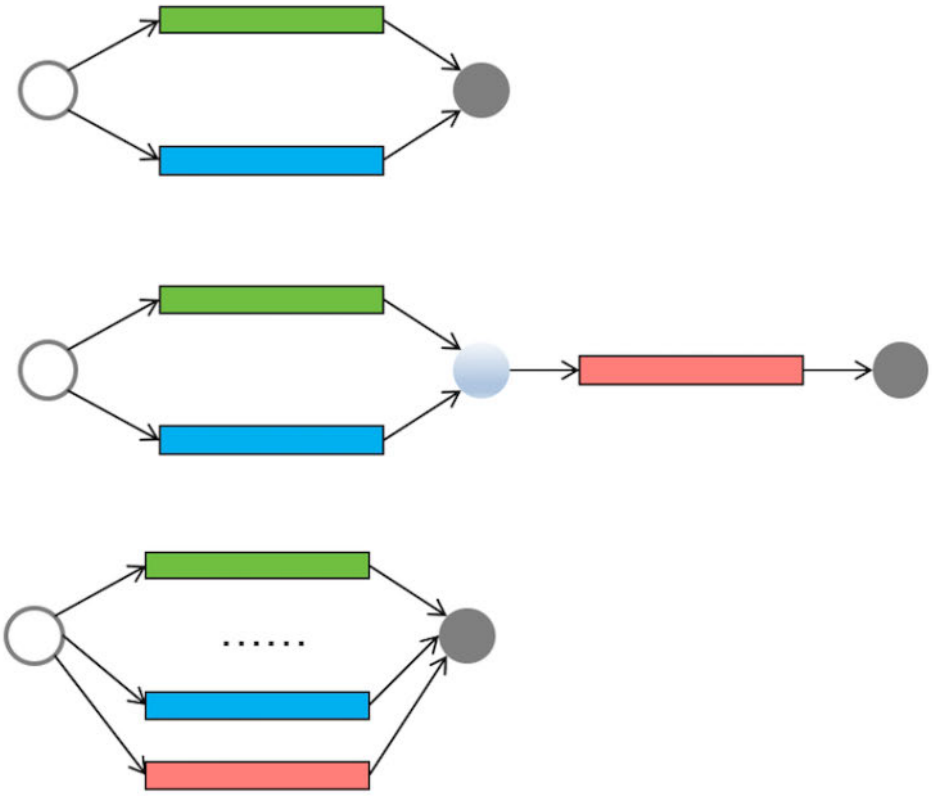
GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。



第三十节：Fork/Join

并发编程领域的任务可以分为三种：简单并行任务、聚合任务和批量并行任务，见下图。



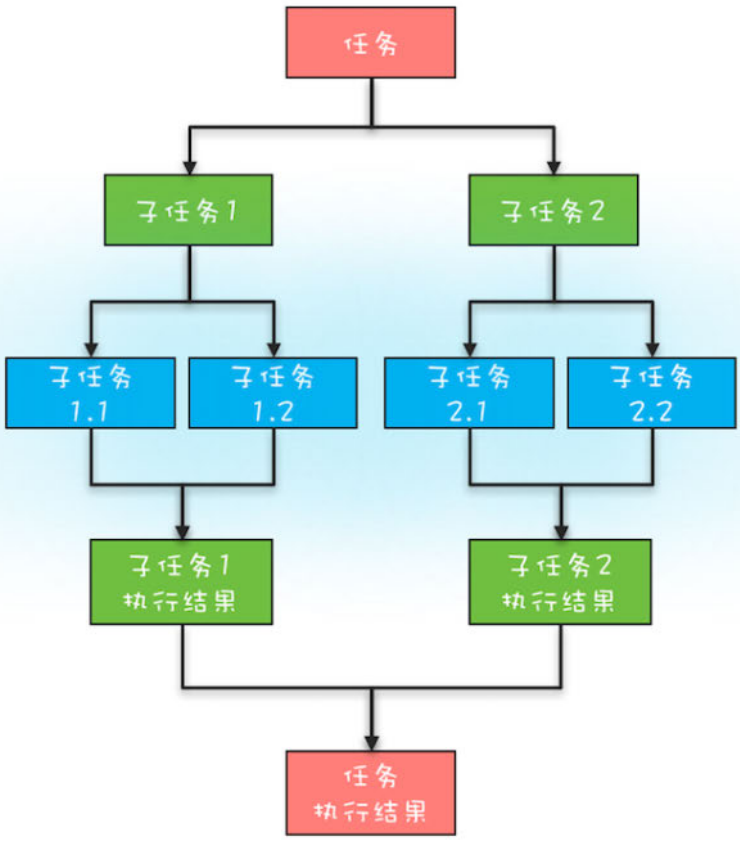
这些模型之外，还有一种任务模型被称为“分治”。分治是一种解决复杂问题的思维方法和模式；具体而言，它将一个复杂的问题分解成多个相似的子问题，然后再将这些子问题进一步分解成更小的子问题，直到每个子问题变得足够简单从而可以直接求解。

从理论上讲，每个问题都对应着一个任务，因此分治实际上就是对任务的划分和组织。分治思想在许多领域都有广泛的应用。例如，在算法领域，我们经常使用分治算法来解决问题（如归并排序和快速排序都属于分治算法，二分查找也是一种分治算法）。在大数据领域，MapReduce 计算框架背后的思想也是基于分治。

由于分治这种任务模型的普遍性，Java 并发包提供了一种名为 Fork/Join 的并行计算框架，专门用于支持分治任务模型的应用。

什么是分治任务模型

分治任务模型可分为两个阶段：一个阶段是 **任务分解**，就是迭代地将任务分解为子任务，直到子任务可以直接计算出结果；另一个阶段是 **结果合并**，即逐层合并子任务的执行结果，直到获得最终结果。下图是一个简化的分治任务模型图，你可以对照着理解。



在这个分治任务模型里，任务和分解后的子任务具有相似性，这种相似性往往体现在任务和子任务的算法是相同的，但是计算的数据规模是不同的。具备这种相似性的问题，我们往往都采用递归算法。

Fork/Join 的使用

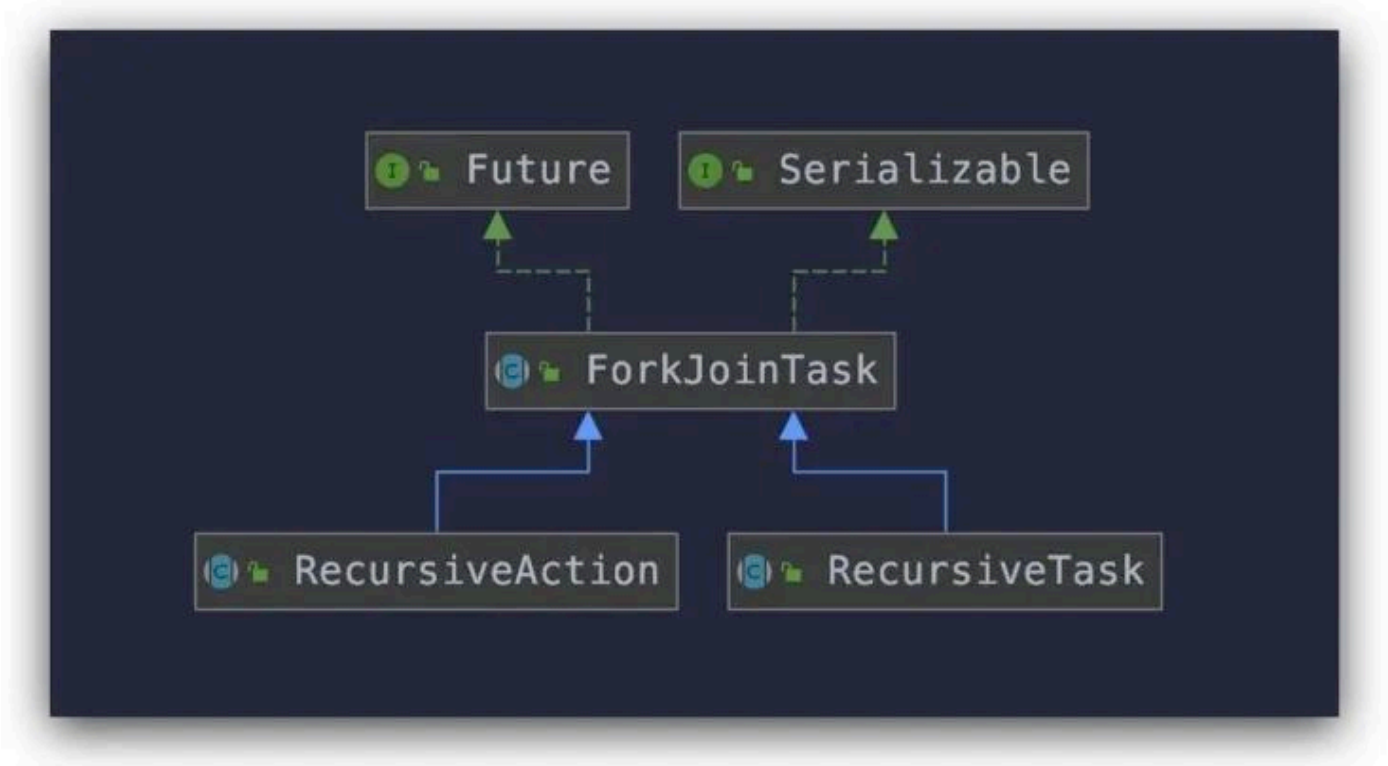
Fork/Join 是一个并行计算框架，主要用于支持分治任务模型。在这个计算框架中，Fork 代表任务的分解，而 Join 代表结果的合并。

Fork/Join 计算框架主要由两部分组成：分治任务的线程池 ForkJoinPool 和分治任务 ForkJoinTask。

这两部分的关系类似于 ThreadPoolExecutor 和 Runnable 之间的关系，都是用于提交任务到线程池的，只不过分治任务有自己独特的类型 ForkJoinTask。

ForkJoinTask 是一个抽象类，其中有许多方法，其中最核心的是 `fork()` 方法和 `join()` 方法。fork 方法用于异步执行一个子任务，而 join 方法通过阻塞当前线程来等待子任务的执行结果。

ForkJoinTask 有两个子类：RecursiveAction 和 RecursiveTask。



从它们的名字就可以看出，都是通过递归的方式来处理分治任务的。这两个子类都定义了一个抽象方法 `compute()`，不同之处在于 RecursiveAction 的 compute 方法没有返回值，而 RecursiveTask 的 compute 方法有返回值。这两个子类也都是抽象类，在使用时需要创建自定义的子类来扩展功能。

接下来，让我们使用 Fork/Join 并行计算框架来计算斐波那契数列（下面的代码示例源自 Java 官方示例）。

首先，我们需要创建一个 ForkJoinPool 线程池以及一个用于计算斐波那契数列的 Fibonacci 分治任务。然后，通过调用 ForkJoinPool 线程池的 `invoke()` 方法来启动分治任务。

由于计算斐波那契数列需要返回结果，所以我们的 Fibonacci 类继承自 RecursiveTask。Fibonacci 分治任务需要实现 compute 方法，在这个方法中，逻辑与普通计算斐波那契数列的方法非常相似，只是在计算 `Fibonacci(n - 1)` 时使用了异步子任务，这是通过 `f1.fork()` 语句来实现。

```
@Slf4j
 public class ForkJoinDemo {
     // 1. 运行入口
     public static void main(String[] args) {
         int n = 20;

         // 为了追踪子线程名称，需要重写 ForkJoinWorkerThreadFactory 的方法
         final ForkJoinPool.ForkJoinWorkerThreadFactory factory = pool -> {
             final ForkJoinWorkerThread worker = ForkJoinPool.defaultForkJoinWorkerThre
adFactory.newThread(pool);
             worker.setName("my-thread" + worker.getPoolIndex());
             return worker;
         };

         //创建分治任务线程池，可以追踪到线程名称
         ForkJoinPool forkJoinPool = new ForkJoinPool(4, factory, null, false);
```

```
        // 快速创建 ForkJoinPool 方法
        // ForkJoinPool forkJoinPool = new ForkJoinPool(4);

        //创建分治任务
        Fibonacci fibonacci = new Fibonacci(n);

        //调用 invoke 方法启动分治任务
        Integer result = forkJoinPool.invoke(fibonacci);
        log.info("Fibonacci {} 的结果是 {}", n, result);
    }
}

// 2. 定义拆分任务，写好拆分逻辑
@Slf4j
class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) {
        this.n = n;
    }

    @Override
    public Integer compute() {
        //和递归类似，定义可计算的最小单元
        if (n <= 1) {
            return n;
        }
        // 想查看子线程名称输出的可以打开下面注释
        //log.info(Thread.currentThread().getName());

        Fibonacci f1 = new Fibonacci(n - 1);
        // 拆分成子任务
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        // f1.join 等待子任务执行结果
        return f2.compute() + f1.join();
    }
}
```

运行程序，我们会得到如下的结果：

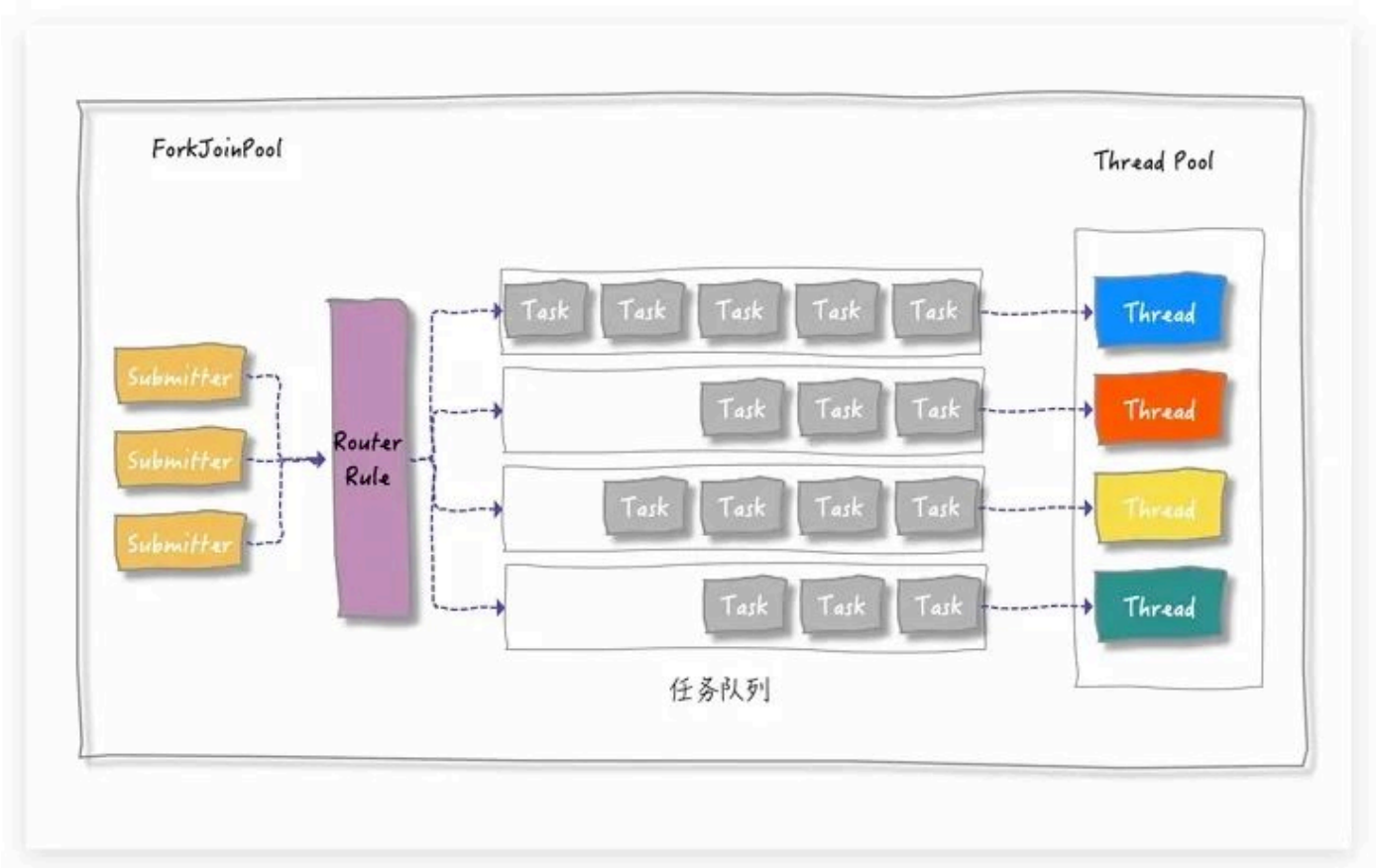
```
17:29:10.336 [main] INFO tech.shuyi.javacodechip.forkjoinpool.ForkJoinDemo - Fibonacci
20 的结果是 6765
```

ForkJoinPool

Fork/Join 并行计算的核心组件是 ForkJoinPool。下面简单介绍一下 ForkJoinPool 的工作原理。

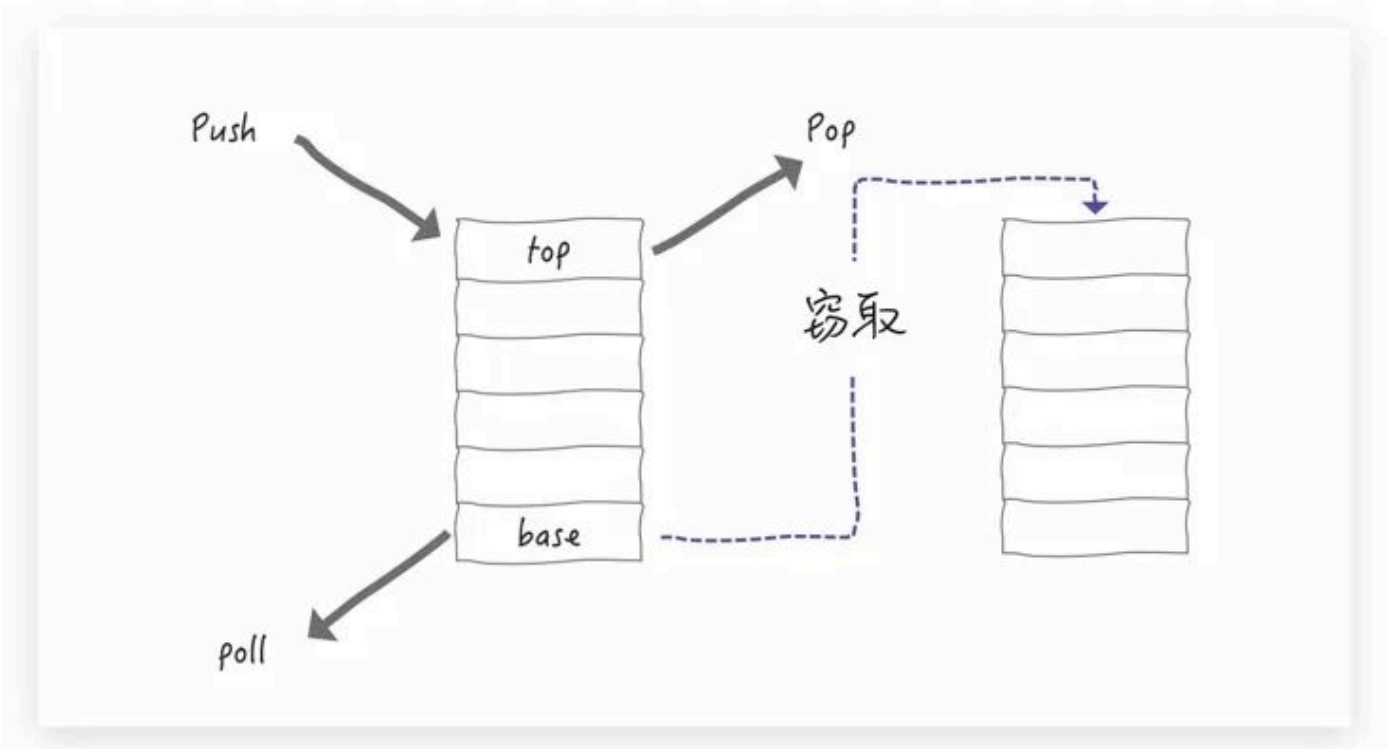
当我们通过 ForkJoinPool 的 invoke 或 submit 方法提交任务时，ForkJoinPool 会根据一定的路由规则将任务分配到一个任务队列中。如果任务执行过程中创建了子任务，那么子任务会被提交到对应工作线程的任务队列中。

ForkJoinPool 中有一个数组形式的成员变量 `workQueue[]`，其对应一个队列数组，每个队列对应一个消费线程。丢入线程池的任务，根据特定规则进行转发。

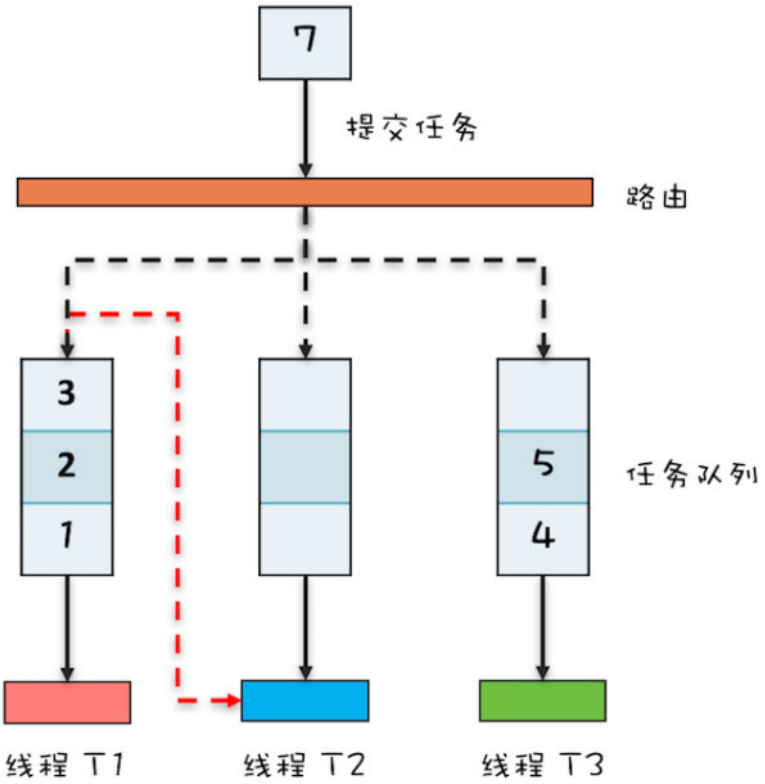


当工作线程的任务队列为空时，它是否无事可做呢?

不是的。ForkJoinPool 引入了一种称为"任务窃取"的机制。当工作线程空闲时，它可以从其他工作线程的任务队列中"窃取"任务。



例如，下图中线程 T2 的任务队列已经为空，它可以窃取线程 T1 任务队列中的任务。这样，所有的工作线程都能保持忙碌的状态。



ForkJoinPool 中的任务队列采用双端队列的形式。工作线程从任务队列的一个端获取任务，而"窃取任务"从另一端进行消费。这种设计能够避免许多不必要的数据竞争。

与ThreadPoolExecutor的比较

ForkJoinPool 与 ThreadPoolExecutor 有很多相似之处，例如都是线程池，都是用于执行任务的。但是，它们之间也有很多不同之处。

首先，ForkJoinPool 采用的是"工作窃取"的机制，而 ThreadPoolExecutor 采用的是"工作复用"的机制。这两种机制各有优劣，ForkJoinPool 的优势在于能够充分利用 CPU 的多核能力，而 ThreadPoolExecutor 的优势在于能够避免线程间的上下文切换。

其次，ForkJoinPool 采用的是分治任务模型，而 ThreadPoolExecutor 采用的是简单并行任务模型。这两种任务模型各有优劣，ForkJoinPool 的优势在于能够处理分治任务，而 ThreadPoolExecutor 的优势在于能够处理简单并行任务。

最后，ForkJoinPool 采用的是 LIFO 的任务队列，而 ThreadPoolExecutor 采用的是 FIFO 的任务队列。这两种任务队列各有优劣，ForkJoinPool 的优势在于能够避免数据竞争，而 ThreadPoolExecutor 的优势在于能够保证任务的顺序性。

假设：我们要计算 1 到 1 亿的和，为了加快计算的速度，我们自然想到算法中的分治原理，将 1 亿个数字分成 1 万个任务，每个任务计算 1 万个数值的综合，利用 CPU 的并发计算性能缩短计算时间。

由于 ThreadPoolExecutor 可以通过 Future 获取到执行结果，因此利用 ThreadPoolExecutor 也是可行的。

当然 ForkJoinPool 实现也是可以的。下面我们将这两种方式都实现一下，看看这两种实现方式有什么不同。

无论哪种实现方式，其大致思路都是：

1. 按照线程池里线程个数 N，将 1 亿个数划分成 N 等份，随后丢入线程池进行计算。
2. 每个计算任务使用 Future 接口获取计算结果，最后积加即可。

我们先使用 ThreadPoolExecutor 实现。

首先，定义一个 Calculator 接口，表示计算数字总和这个动作，如下所示。

```
public interface Calculator {
     /**
      * 把传进来的所有numbers 做求和处理
      *
      * @param numbers
      * @return 总和
      */
     long sumUp(long[] numbers);
 }
```

接着，我们定义一个使用 ThreadPoolExecutor 线程池实现的类，如下所示。

```
 public class ExecutorServiceCalculator implements Calculator {

     private int parallism;
     private ExecutorService pool;

     public ExecutorServiceCalculator() {
         // CPU的核心数 默认就用cpu核心数了
         parallism = Runtime.getRuntime().availableProcessors();
         pool = Executors.newFixedThreadPool(parallism);
     }

     // 1. 处理计算任务的线程
     private static class SumTask implements Callable<Long> {
         private long[] numbers;
         private int from;
         private int to;

         public SumTask(long[] numbers, int from, int to) {
             this.numbers = numbers;
             this.from = from;
             this.to = to;
         }

         @Override
         public Long call() {
             long total = 0;
             for (int i = from; i <= to; i++) {
                 total += numbers[i];
             }
             return total;
```

```
        }
    }

    // 2. 核心业务逻辑实现
    @Override
    public long sumUp(long[] numbers) {
        List<Future<Long>> results = new ArrayList<>();

        // 2.1 数字拆分
        // 把任务分解为 n 份，交给 n 个线程处理    4核心 就等分成4份呗
        // 然后把每一份都扔个一个SumTask线程 进行处理
        int part = numbers.length / parallism;
        for (int i = 0; i < parallism; i++) {
            int from = i * part; //开始位置
            int to = (i == parallism - 1) ? numbers.length - 1 : (i + 1) * part - 1; /
/结束位置

            //扔给线程池计算
            results.add(pool.submit(new SumTask(numbers, from, to)));
        }

        // 2.2 阻塞等待结果
        // 把每个线程的结果相加，得到最终结果 get()方法 是阻塞的
        // 优化方案：可以采用CompletableFuture来优化  JDK1.8的新特性
        long total = 0L;
        for (Future<Long> f : results) {
            try {
                total += f.get();
            } catch (Exception ignore) {
            }
        }

        return total;
    }
}
```

如上面代码所示，我们实现了一个计算单个任务的类 SumTask，在该类中对数值进行累加。其次，我们在 `sumUp()` 方法中，对 1 亿的数字进行拆分，接着扔给线程池计算，最后阻塞等待计算结果，最终累加起来。

我们运行上面的代码，可以得到顺利得到最终结果，如下所示。

```
耗时：10ms
 结果为：50000005000000
```

接着我们使用 ForkJoinPool 来实现。

我们首先实现 SumTask 继承 RecursiveTask 抽象类，并在 compute 方法中定义拆分逻辑及计算。最后在 sumUp 方法中调用 pool 方法进行计算，代码如下所示。

```
public class ForkJoinCalculator implements Calculator {

    private ForkJoinPool pool;

    // 1. 定义计算逻辑
    private static class SumTask extends RecursiveTask<Long> {
        private long[] numbers;
        private int from;
        private int to;

        public SumTask(long[] numbers, int from, int to) {
            this.numbers = numbers;
            this.from = from;
            this.to = to;
        }

        //此方法为ForkJoin的核心方法：对任务进行拆分  拆分的好坏决定了效率的高低
        @Override
        protected Long compute() {

            // 当需要计算的数字个数小于6时，直接采用for loop方式计算结果
            if (to - from < 6) {
                long total = 0;
                for (int i = from; i <= to; i++) {
                    total += numbers[i];
                }
                return total;
            } else {
                // 否则，把任务一分为二，递归拆分(注意此处有递归)到底拆分成多少分 需要根据具体情况
而定
                int middle = (from + to) / 2;
                SumTask taskLeft = new SumTask(numbers, from, middle);
                SumTask taskRight = new SumTask(numbers, middle + 1, to);
                taskLeft.fork();
                taskRight.fork();
                return taskLeft.join() + taskRight.join();
            }
        }
    }

    public ForkJoinCalculator() {
        // 也可以使用公用的线程池 ForkJoinPool.commonPool()：
        // pool = ForkJoinPool.commonPool()
        pool = new ForkJoinPool();
    }

    @Override
    public long sumUp(long[] numbers) {
        Long result = pool.invoke(new SumTask(numbers, 0, numbers.length - 1));
```

```
        pool.shutdown();
        return result;
    }
}
```

运行上面的代码，结果为：

```
耗时：860ms
 结果为：50000005000000
```

对比 ThreadPoolExecutor 和 ForkJoinPool 这两者的实现，可以发现它们都有任务拆分的逻辑，以及最终合并数值的逻辑。但 ForkJoinPool 相比 ThreadPoolExecutor 来说，做了一些实现上的封装，例如：

- 不用手动去获取子任务的结果，而是使用 join 方法直接获取结果。
- 将任务拆分的逻辑，封装到 RecursiveTask 实现类中，而不是裸露在外。

因此对于没有父子任务依赖，但是希望获取到子任务执行结果的并行计算任务，就可以使用 ForkJoinPool 来实现。**在这种情况下，使用 ForkJoinPool 实现更多是代码实现方便，封装做得更加好。**

模拟 MapReduce 统计单词数量

MapReduce 是一个编程模型，同时也是一个处理和生成大数据集的处理框架。它源于 Google，用于支持在大型数据集上的分布式计算。这个框架主要由两个步骤组成：Map 步骤和 Reduce 步骤，这也是它名字的由来。

Fork/Join 并行计算框架通常被用来实现学习 MapReduce 的入门程序，该程序用于统计文件中每个单词的数量。

首先，我们可以使用二分法递归地将文件拆分为更小的部分，直到每个部分只有一行数据。然后，在每个部分中统计单词的数量，并逐级汇总结果。你可以参考之前提到的简化版分治任务模型图来理解该过程。

现在，让我们开始实现。下面的代码使用了字符串数组`String[] fc`来模拟文件内容，其中每个元素与文件中的行数据一一对应。关键代码位于`compute()`方法中，这是一个递归方法。它将前半部分数据 fork 一个递归任务进行处理（关键代码：`mr1.fork()`），而后半部分数据在当前任务中递归处理（`mr2.compute()`）。

```
import java.util.concurrent.RecursiveTask;

public class WordCountTask extends RecursiveTask<Integer> {
    private final String[] fc;
    private final int start, end;

    public WordCountTask(String[] fc, int start, int end) {
        this.fc = fc;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 1) {
            // 对单行数据进行统计
            return countWords(fc[start]);
```

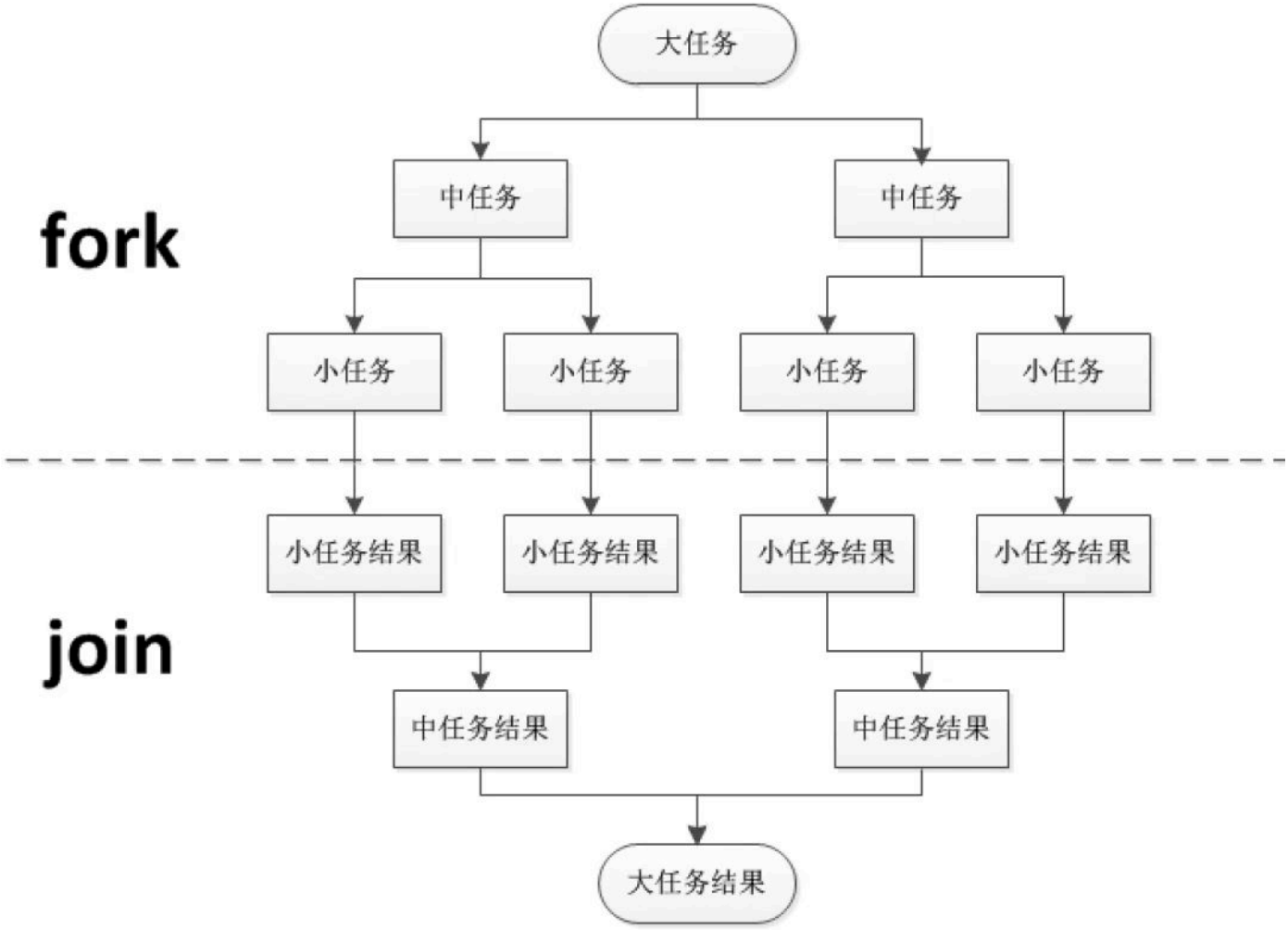
```
        } else {
            int mid = (start + end) / 2;
            WordCountTask mr1 = new WordCountTask(fc, start, mid);
            mr1.fork();
            WordCountTask mr2 = new WordCountTask(fc, mid, end);
            int result2 = mr2.compute();
            int result1 = mr1.join();
            // 汇总结果
            return result1 + result2;
        }
    }

    private int countWords(String line) {
        String[] words = line.split(" ");
        return words.length;
    }
}
```

这个示例程序是对 Fork/Join 模型的简化，实际上在真正的 MapReduce 框架中，还涉及到数据划分、映射阶段、归约阶段等更多的步骤。但是通过此示例，大家可以初步了解如何使用 Fork/Join 并行计算框架来处理类似的任务。

小结

Fork/Join 并行计算框架主要解决的是分治任务。分治的核心思想是“分而治之”：将一个大的任务拆分成小的子任务去解决，然后再把子任务的结果聚合起来从而得到最终结果。这个过程非常类似于大数据处理中的 MapReduce，所以你可以把 Fork/Join 看作单机版的 MapReduce。

Fork/Join 并行计算框架的核心组件是 ForkJoinPool。ForkJoinPool 支持任务窃取机制，能够让所有线程的工作量基本均衡，不会出现有的线程很忙，而有的线程很闲的状况，所以性能很好。



Java 1.8 提供的 Stream API 里面并行流也是以 ForkJoinPool 为基础的。不过需要注意的是，默认情况下所有的并行流计算都共享一个 ForkJoinPool，这个共享的 ForkJoinPool 默认的线程数是 CPU 的核数；如果所有的并行流计算都是 CPU 密集型计算的话，完全没有问题，但是如果存在 I/O 密集型的并行流计算，那么很可能会因为一个很慢的 I/O 计算而拖慢整个系统的性能。

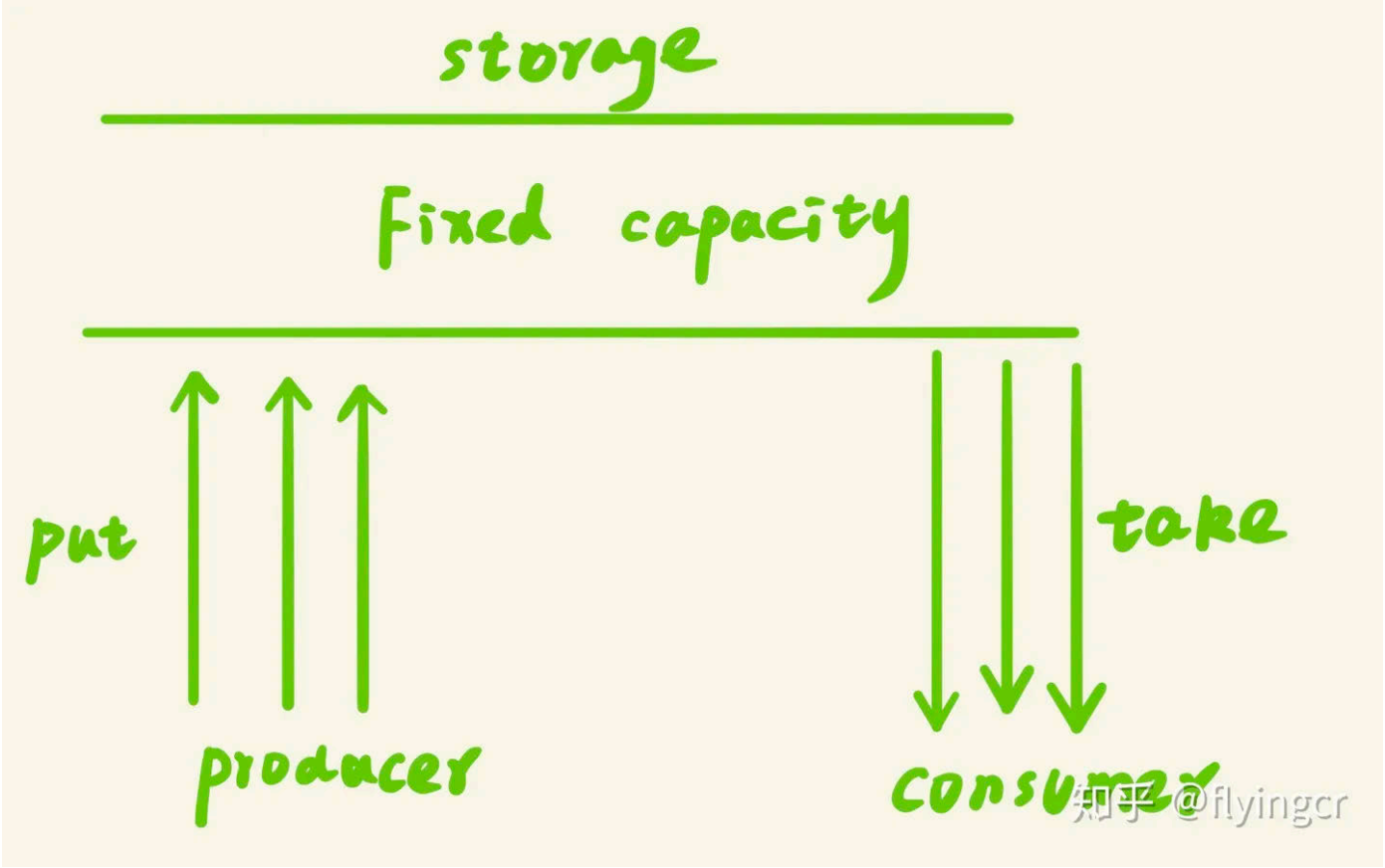
所以 **建议用不同的 ForkJoinPool 执行不同类型的计算任务**。

> 编辑：沉默王二，部分内容来源于这篇文章：分而治之思想Forkjoin，还有一部分内容来源于朋友陈树义的这篇文章，内容很顶，强烈推荐。还有一部分图片来自于朋友「日拱一兵」的文章。

第三十一节：生产者-消费者模式

生产者-消费者模式是一个十分经典的多线程并发协作模式，弄懂生产者-消费者问题能够让我们对并发编程的理解加深。

所谓的生产者-消费者，实际上包含了两类线程，一种是生产者线程用于生产数据，另一种是消费者线程用于消费数据，为了解耦生产者和消费者的关系，通常会采用共享的数据区域，就像是一个仓库，生产者生产数据之后直接放置在共享数据区中，并不需要关心消费者的行为；而消费者只需要从共享数据区中获取数据，不需要关心生产者的行为。



这个共享数据区域中应该具备这样的线程间并发协作功能：

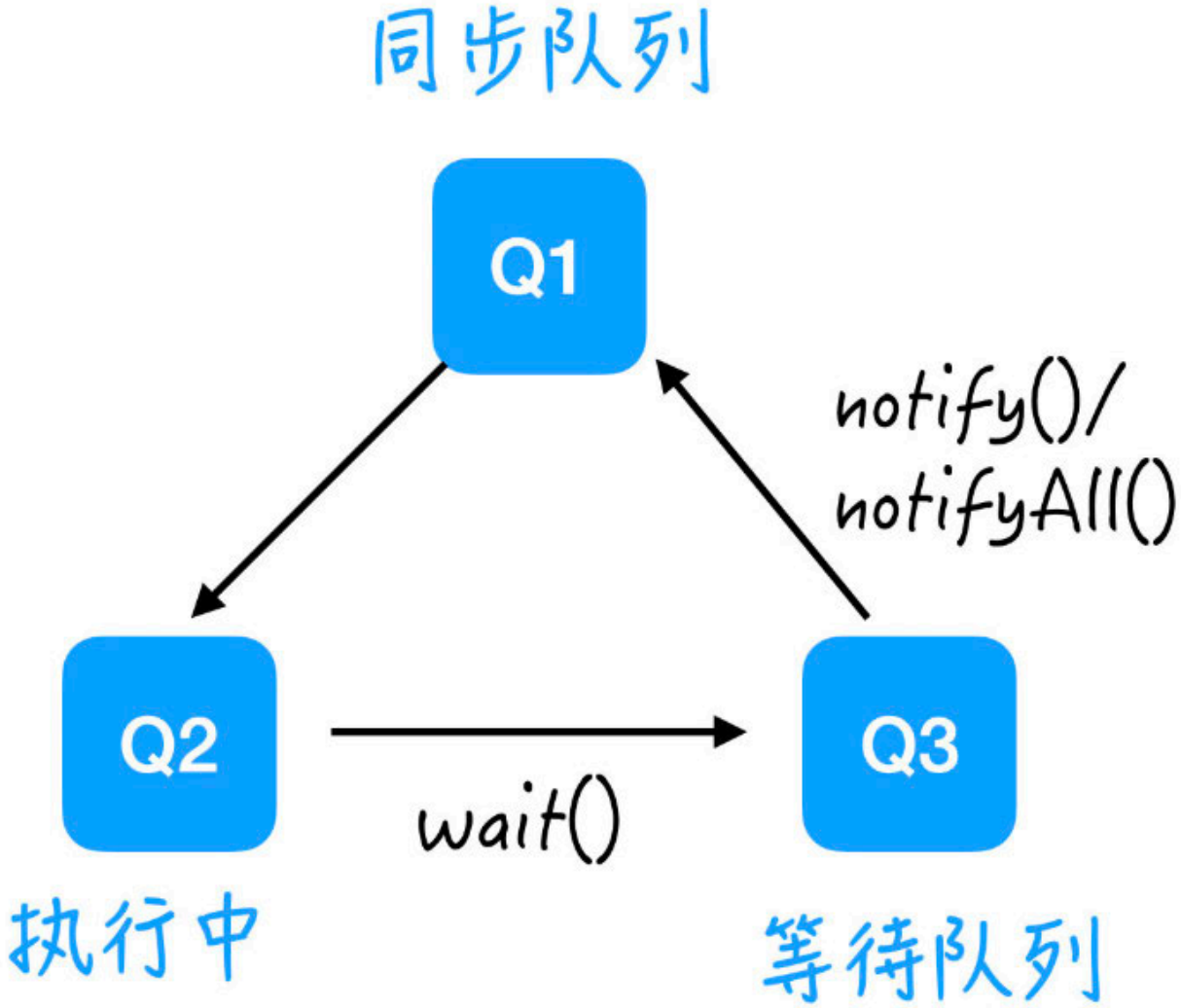
1. 如果共享数据区已满的话，阻塞生产者继续生产数据；
2. 如果共享数据区为空的话，阻塞消费者继续消费数据；

在实现生产者消费者问题时，可以采用三种方式：

1. 使用 Object 的 wait/notify 的消息通知机制；
2. 使用 Lock Condition 的 await/signal 消息通知机制；
3. 使用 BlockingQueue 实现。

wait/notify 的消息通知机制

可以通过 Object 对象的 wait 方法和 notify 方法或 notifyAll 方法来实现线程间的通信。



调用 wait 方法将阻塞当前线程，直到其他线程调用了 notify 方法或 notifyAll 方法进行通知，当前线程才能从 wait 方法处返回，继续执行下面的操作。

这些知识我们在讲 Condition 的时候其实讲到过，相信大家都还有印象。

01、wait

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。

在调用 wait 之前，线程必须获得该对象的监视器锁，即只能在**同步方法或同步块**中调用 wait 方法。调用 wait 方法之后，当前线程会释放锁。如果调用 wait 方法时，线程并未获取到锁的话，则会**抛出 IllegalMonitorStateException**异常。如果再次获取到锁的话，当前线程才能从 wait 方法处成功返回。

02、notify

该方法也需要在同步方法或同步块中调用，即在调用前，线程也必须获得该对象的对象级别锁，如果调用 notify 时没有持有适当的锁，也会抛出 **IllegalMonitorStateException**。

该方法会从 WAITTING 状态的线程中挑选一个进行通知，使得调用 wait 方法的线程从等待队列移入到同步队列中，等待机会再一次获取到锁，从而使得调用 wait 方法的线程能够从 wait 方法处退出。

调用 notify 后，当前线程不会马上释放该对象锁，要等到程序退出同步块后，当前线程才会释放锁。

03、notifyAll

该方法与 notify 方法的工作方式相同，重要的一点差异是：notifyAll 会使所有原来在该对象上 wait 线程统统退出 WAITTING 状态，使得他们全部从等待队列中移入到同步队列中去，等待下一次获取到对象监视器锁的机会。

不过，wait/notify 消息通知存在这样一些问题。

1.notify 早期通知

notify 通知的遗漏，即 threadA 还没开始 wait，threadB 已经 notify 了，这样，threadB 通知是没有任何响应的，当 threadB 退出 synchronized 代码块后，threadA 再开始 wait，便会一直阻塞等待，直到被别的线程打断。

下面的示例代码就模拟出了 notify 早期通知带来的问题：

```
public class EarlyNotify {

    private static String lockObject = "";

    public static void main(String[] args) {
        WaitThread waitThread = new WaitThread(lockObject);
        NotifyThread notifyThread = new NotifyThread(lockObject);
        notifyThread.start();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        waitThread.start();
    }

    static class WaitThread extends Thread {
        private String lock;

        public WaitThread(String lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                try {
                    System.out.println(Thread.currentThread().getName() + "   进去代码
块");
                    System.out.println(Thread.currentThread().getName() + "   开始wait");
                    lock.wait();
                    System.out.println(Thread.currentThread().getName() + "    结束
wait");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }

    static class NotifyThread extends Thread {
        private String lock;

        public NotifyThread(String lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + "   进去代码块");
                System.out.println(Thread.currentThread().getName() + "   开始notify");
                lock.notify();
                System.out.println(Thread.currentThread().getName() + "    结束开始
notify");
            }
        }
    }
}
```

示例中开启了**两个线程，一个是 WaitThread，另一个是 NotifyThread。NotifyThread 会先启动调用 notify 方法。然后 WaitThread 线程才启动，调用 wait 方法，但由于通知过了，wait 方法就无法再获取到相应的通知，因此 WaitThread 会一直在 wait 方法处阻塞，这种现象就是通知过早的现象。**

针对这种问题的解决方法是，添加一个状态标志，让 waitThread 调用 wait 方法前先判断状态是否已经改变了，如果通知已经发出，WaitThread 就不再去 wait。对上面的代码进行优化如下：

```
public class EarlyNotify {

    private static String lockObject = "";
    private static boolean isWait = true;

    public static void main(String[] args) {
        WaitThread waitThread = new WaitThread(lockObject);
        NotifyThread notifyThread = new NotifyThread(lockObject);
        notifyThread.start();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        waitThread.start();
    }

    static class WaitThread extends Thread {
        private String lock;
```

```
        public WaitThread(String lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                try {
                    while (isWait) {
                        System.out.println(Thread.currentThread().getName() + "   进去代码
块");
                        System.out.println(Thread.currentThread().getName() + "   开始
wait");
                        lock.wait();
                        System.out.println(Thread.currentThread().getName() + "    结束
wait");
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static class NotifyThread extends Thread {
        private String lock;

        public NotifyThread(String lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + "   进去代码块");
                System.out.println(Thread.currentThread().getName() + "   开始notify");
                lock.notifyAll();
                isWait = false;
                System.out.println(Thread.currentThread().getName() + "    结束开始
notify");
            }
        }
    }
}
```

这段代码只增加了一个`isWait`状态，NotifyThread 调用 notify 方法后会对状态进行更新，WaitThread 调用 wait 方法之前会先对状态进行判断。

该示例中，调用 notify 后将状态`isWait`改变为 false，因此，在 WaitThread 中 while 对 isWait 判断后就不会执行 wait 方法，从而**避免了 Notify 过早通知造成遗漏的情况。**

总结：在使用线程的等待/通知机制时，一般都要配合一个 boolean 变量值，在 notify 之前改变该 boolean 变量的值，让 wait 返回后能够退出 while 循环，或在通知被遗漏后不会被阻塞在 wait 方法处。

2.等待 wait 的条件发生变化

如果线程在等待时接收到了通知，但是之后等待的条件发生了变化，并没有再次对等待条件进行判断，也会导致程序出现错误。

下面用一个例子来说明这种情况。

```
public class ConditionChange {
  private static List<String> lockObject = new ArrayList();


  public static void main(String[] args) {
      Consumer consumer1 = new Consumer(lockObject);
      Consumer consumer2 = new Consumer(lockObject);
      Productor productor = new Productor(lockObject);
      consumer1.start();
      consumer2.start();
      productor.start();
  }


  static class Consumer extends Thread {
      private List<String> lock;

      public Consumer(List lock) {
          this.lock = lock;
      }

      @Override
      public void run() {
          synchronized (lock) {
              try {
                  //这里使用if的话，就会存在wait条件变化造成程序错误的问题
                  if (lock.isEmpty()) {
                      System.out.println(Thread.currentThread().getName() + " list为
空");
                      System.out.println(Thread.currentThread().getName() + " 调用wait方
法");
                      lock.wait();
                      System.out.println(Thread.currentThread().getName() + "  wait方法
结束");
                  }
                  String element = lock.remove(0);
```

```
                    System.out.println(Thread.currentThread().getName() + " 取出第一个元素为: " + element);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

    }


    static class Productor extends Thread {
        private List<String> lock;

        public Productor(List lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + " 开始添加元素");
                lock.add(Thread.currentThread().getName());
                lock.notifyAll();
            }
        }

    }
}
```

会报异常：

```
Exception in thread "Thread-1" Thread-0 list为空
Thread-0 调用wait方法
Thread-1 list为空
Thread-1 调用wait方法
Thread-2 开始添加元素
Thread-1  wait方法结束
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
```

在这个例子中，一共开启了 3 个线程，Consumer1,Consumer2 以及 Productor。

Consumer1 调用了 wait 方法后，线程处于了 WAITTING 状态，并且将对象锁释放。

此时，Consumer2 获取到对象锁，进入到同步代块中，当执行到 wait 方法时，同样的也会释放对象锁。

然后 productor 获取到对象锁，进入到同步代码块中，向 list 中插入数据，通过 notifyAll 方法通知处于 WAITING 状态的 Consumer1 和 Consumer2 线程。

consumer1 得到对象锁后，从 wait 方法处退出，删除一个元素让 List 为空，方法执行结束，退出同步块，释放掉对象锁。

这个时候 Consumer2 获取到对象锁后，从 wait 方法退出，继续往下执行，这个时候 Consumer2 再执行`lock.remove(0);`就会出错，因为 List 已经为空了。

解决方案： 通过上面的分析，可以看出 Consumer2 报错是因为线程从 wait 方法退出之后没有对 wait 条件进行判断，但此时的 wait 条件已经发生了变化。解决办法就是在 wait 退出之后再对条件进行判断。

```
public class ConditionChange {
  private static List<String> lockObject = new ArrayList();


  public static void main(String[] args) {
      Consumer consumer1 = new Consumer(lockObject);
      Consumer consumer2 = new Consumer(lockObject);
      Productor productor = new Productor(lockObject);
      consumer1.start();
      consumer2.start();
      productor.start();
  }


  static class Consumer extends Thread {
      private List<String> lock;

      public Consumer(List lock) {
          this.lock = lock;
      }

      @Override
      public void run() {
          synchronized (lock) {
              try {
                  //这里使用if的话，就会存在wait条件变化造成程序错误的问题
                  while (lock.isEmpty()) {
                      System.out.println(Thread.currentThread().getName() + " list为
空");
                      System.out.println(Thread.currentThread().getName() + " 调用wait方
法");
                      lock.wait();
                      System.out.println(Thread.currentThread().getName() + "  wait方法
结束");
                  }
                  String element = lock.remove(0);
                  System.out.println(Thread.currentThread().getName() + " 取出第一个元素
为: " + element);
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }
```

```
            }
        }

    }


    static class Productor extends Thread {
        private List<String> lock;

        public Productor(List lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + " 开始添加元素");
                lock.add(Thread.currentThread().getName());
                lock.notifyAll();
            }
        }

    }
}
```

上面的代码与之前的代码相比，仅仅只是将 wait 外围的 if 语句改为了 while 循环，这样当 list 为空时，线程便会继续等待，而不会继续去执行删除 list 中元素中的代码。

总结：在使用线程的等待/通知机制时，一般都要在 while 循环中调用 wait 方法，因此需要配合一个 boolean 变量，满足 while 循环的条件时进入 while 循环，执行 wait 方法，不满足 while 循环条件时，跳出循环，执行后面的代码。

3. “假死”状态

现象：如果是多消费者和多生产者情况，使用 notify 方法可能会出现“假死”的情况，即所有的线程都处于等待状态，无法被唤醒。

原因分析：假设当前有多个生产者线程调用了 wait 方法阻塞等待，其中一个生产者线程获取到对象锁之后使用 notify 通知处于 WAITTING 状态的线程，如果唤醒的仍然是生产者线程，就会造成所有的生产者线程都处于等待状态。

解决办法：将 notify 方法替换成 notifyAll 方法，如果使用的是 lock 的话，就将 signal 方法替换成 signalAll 方法。

总结：Object 提供的消息通知机制应该遵循如下这些条件：

1. 永远在 while 循环中对条件进行判断而不是在 if 语句中进行 wait 条件的判断；
2. 使用 NotifyAll 而不是使用 notify。

基本的使用范式如下：

```
// The standard idiom for calling the wait method in Java
synchronized (sharedObject) {
    while (condition) {
    sharedObject.wait();
        // (Releases lock, and reacquires on wakeup)
    }
    // do action based upon condition e.g. take or put into queue
}
```

wait/notifyAll 实现生产者-消费者

利用 wait/notifyAll 实现生产者和消费者代码如下：

```
public class ProductorConsumer {


    public static void main(String[] args) {

        LinkedList linkedList = new LinkedList();
        ExecutorService service = Executors.newFixedThreadPool(15);
        for (int i = 0; i < 5; i++) {
            service.submit(new Productor(linkedList, 8));
        }

        for (int i = 0; i < 10; i++) {
            service.submit(new Consumer(linkedList));
        }

    }

    static class Productor implements Runnable {

        private List<Integer> list;
        private int maxLength;

        public Productor(List list, int maxLength) {
            this.list = list;
            this.maxLength = maxLength;
        }

        @Override
        public void run() {
            while (true) {
                synchronized (list) {
                    try {
                        while (list.size() == maxLength) {
                            System.out.println("生产者" +
Thread.currentThread().getName() + "  list以达到最大容量，进行wait");
```

```
                            list.wait();
                            System.out.println("生产者" +
Thread.currentThread().getName() + "  退出wait");
                        }
                        Random random = new Random();
                        int i = random.nextInt();
                        System.out.println("生产者" + Thread.currentThread().getName() +
" 生产数据" + i);
                        list.add(i);
                        list.notifyAll();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

            }
        }
    }


    static class Consumer implements Runnable {

        private List<Integer> list;

        public Consumer(List list) {
            this.list = list;
        }

        @Override
        public void run() {
            while (true) {
                synchronized (list) {
                    try {
                        while (list.isEmpty()) {
                            System.out.println("消费者" +
Thread.currentThread().getName() + "  list为空，进行wait");
                            list.wait();
                            System.out.println("消费者" +
Thread.currentThread().getName() + "  退出wait");
                        }
                        Integer element = list.remove(0);
                        System.out.println("消费者" + Thread.currentThread().getName() +
"  消费数据：" + element);
                        list.notifyAll();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
```

```
        }
    }

}
```

输出结果：

```
生产者pool-1-thread-1 生产数据-232820990
生产者pool-1-thread-1 生产数据1432164130
生产者pool-1-thread-1 生产数据1057090222
生产者pool-1-thread-1 生产数据1201395916
生产者pool-1-thread-1 生产数据482766516
生产者pool-1-thread-1  list以达到最大容量，进行wait
消费者pool-1-thread-15  退出wait
消费者pool-1-thread-15  消费数据：1237535349
消费者pool-1-thread-15  消费数据：-1617438932
消费者pool-1-thread-15  消费数据：-535396055
消费者pool-1-thread-15  消费数据：-232820990
消费者pool-1-thread-15  消费数据：1432164130
消费者pool-1-thread-15  消费数据：1057090222
消费者pool-1-thread-15  消费数据：1201395916
消费者pool-1-thread-15  消费数据：482766516
消费者pool-1-thread-15  list为空，进行wait
生产者pool-1-thread-5  退出wait
生产者pool-1-thread-5 生产数据1442969724
生产者pool-1-thread-5 生产数据1177554422
生产者pool-1-thread-5 生产数据-133137235
生产者pool-1-thread-5 生产数据324882560
生产者pool-1-thread-5 生产数据2065211573
生产者pool-1-thread-5 生产数据253569900
生产者pool-1-thread-5 生产数据571277922
生产者pool-1-thread-5 生产数据1622323863
生产者pool-1-thread-5  list以达到最大容量，进行wait
消费者pool-1-thread-10  退出wait
```

await/signalAll 实现生产者-消费者

参照 Object 的 wait 和 notify/notifyAll 方法，Condition 也提供了同样的方法，即 await 方法和 signal/signalAll 方法。这部分知识我们前面在讲 Condition 的时候也讲到过，相信大家都还有印象。

那如果采用 Conditon 的消息通知原理来实现生产者-消费者模型，原理同使用 wait/notifyAll 一样。直接上代码：

```
public class ProductorConsumer {

    private static ReentrantLock lock = new ReentrantLock();
    private static Condition full = lock.newCondition();
    private static Condition empty = lock.newCondition();
```

```
    public static void main(String[] args) {
        LinkedList linkedList = new LinkedList();
        ExecutorService service = Executors.newFixedThreadPool(15);
        for (int i = 0; i < 5; i++) {
            service.submit(new Productor(linkedList, 8, lock));
        }
        for (int i = 0; i < 10; i++) {
            service.submit(new Consumer(linkedList, lock));
        }

    }

    static class Productor implements Runnable {

        private List<Integer> list;
        private int maxLength;
        private Lock lock;

        public Productor(List list, int maxLength, Lock lock) {
            this.list = list;
            this.maxLength = maxLength;
            this.lock = lock;
        }

        @Override
        public void run() {
            while (true) {
                lock.lock();
                try {
                    while (list.size() == maxLength) {
                        System.out.println("生产者" + Thread.currentThread().getName() +
"  list以达到最大容量，进行wait");
                        full.await();
                        System.out.println("生产者" + Thread.currentThread().getName() +
"  退出wait");
                    }
                    Random random = new Random();
                    int i = random.nextInt();
                    System.out.println("生产者" + Thread.currentThread().getName() + " 生
产数据" + i);
                    list.add(i);
                    empty.signalAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }
```

```
    }


    static class Consumer implements Runnable {

        private List<Integer> list;
        private Lock lock;

        public Consumer(List list, Lock lock) {
            this.list = list;
            this.lock = lock;
        }

        @Override
        public void run() {
            while (true) {
                lock.lock();
                try {
                    while (list.isEmpty()) {
                        System.out.println("消费者" + Thread.currentThread().getName() +
"  list为空，进行wait");
                        empty.await();
                        System.out.println("消费者" + Thread.currentThread().getName() +
"  退出wait");
                    }
                    Integer element = list.remove(0);
                    System.out.println("消费者" + Thread.currentThread().getName() + "
消费数据：" + element);
                    full.signalAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }
    }

}
```

输出结果：

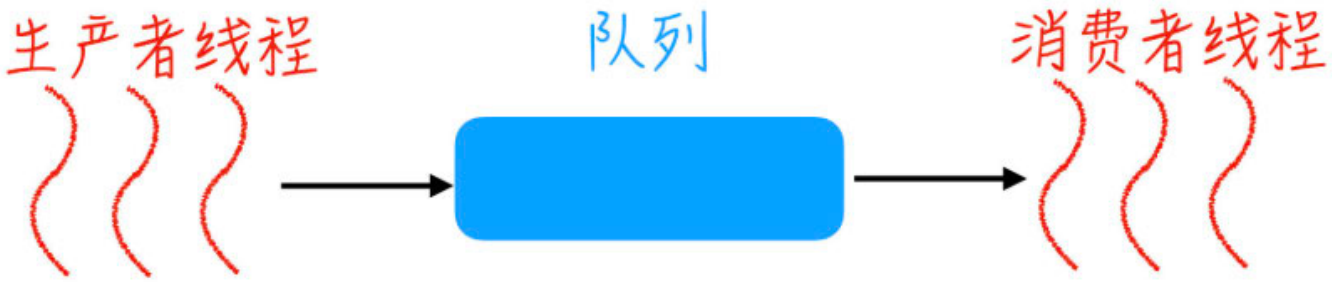
```
消费者pool-1-thread-9  消费数据：1146627506
消费者pool-1-thread-9  消费数据：1508001019
消费者pool-1-thread-9  消费数据：-600080565
消费者pool-1-thread-9  消费数据：-1000305429
消费者pool-1-thread-9  消费数据：-1270658620
消费者pool-1-thread-9  消费数据：1961046169
消费者pool-1-thread-9  消费数据：-307680655
```

```
消费者pool-1-thread-9  list为空，进行wait
消费者pool-1-thread-13  退出wait
消费者pool-1-thread-13  list为空，进行wait
消费者pool-1-thread-10  退出wait
生产者pool-1-thread-5  退出wait
生产者pool-1-thread-5 生产数据-892558288
生产者pool-1-thread-5 生产数据-1917220008
生产者pool-1-thread-5 生产数据2146351766
生产者pool-1-thread-5 生产数据452445380
生产者pool-1-thread-5 生产数据1695168334
生产者pool-1-thread-5 生产数据1979746693
生产者pool-1-thread-5 生产数据-1905436249
生产者pool-1-thread-5 生产数据-101410137
生产者pool-1-thread-5  list以达到最大容量，进行wait
生产者pool-1-thread-1  退出wait
生产者pool-1-thread-1  list以达到最大容量，进行wait
生产者pool-1-thread-4  退出wait
生产者pool-1-thread-4  list以达到最大容量，进行wait
生产者pool-1-thread-2  退出wait
生产者pool-1-thread-2  list以达到最大容量，进行wait
生产者pool-1-thread-3  退出wait
生产者pool-1-thread-3  list以达到最大容量，进行wait
消费者pool-1-thread-9  退出wait
消费者pool-1-thread-9  消费数据：-892558288
```

BlockingQueue 实现生产者-消费者

在讲 BlockingQueue 的时候，我们就讲过，BlockingQueue 非常适合用来实现生产者-消费者模型。

其原因是 BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。



有了这个队列，生产者就只需要关注生产，而不用管消费者的消费行为，更不用等待消费者线程执行完；消费者也只管消费，不用管生产者是怎么生产的，更不用等着生产者生产。

下面直接上代码：

```
public class ProductorConsumer {

    private static LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<>();
```

```
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(15);
        for (int i = 0; i < 5; i++) {
            service.submit(new Productor(queue));
        }
        for (int i = 0; i < 10; i++) {
            service.submit(new Consumer(queue));
        }
    }


    static class Productor implements Runnable {

        private BlockingQueue queue;

        public Productor(BlockingQueue queue) {
            this.queue = queue;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    Random random = new Random();
                    int i = random.nextInt();
                    System.out.println("生产者" + Thread.currentThread().getName() + "生
产数据" + i);
                    queue.put(i);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    static class Consumer implements Runnable {
        private BlockingQueue queue;

        public Consumer(BlockingQueue queue) {
            this.queue = queue;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    Integer element = (Integer) queue.take();
```

```
                    System.out.println("消费者" + Thread.currentThread().getName() + "正在消费数据" + element);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

输出结果：

```
消费者pool-1-thread-7正在消费数据1520577501
生产者pool-1-thread-4生产数据-127809610
消费者pool-1-thread-8正在消费数据504316513
生产者pool-1-thread-2生产数据1994678907
消费者pool-1-thread-11正在消费数据1967302829
生产者pool-1-thread-1生产数据369331507
消费者pool-1-thread-9正在消费数据1994678907
生产者pool-1-thread-2生产数据-919544017
消费者pool-1-thread-12正在消费数据-127809610
生产者pool-1-thread-4生产数据1475197572
消费者pool-1-thread-14正在消费数据-893487914
生产者pool-1-thread-3生产数据906921688
消费者pool-1-thread-6正在消费数据-1292015016
生产者pool-1-thread-5生产数据-652105379
生产者pool-1-thread-5生产数据-1622505717
生产者pool-1-thread-3生产数据-1350268764
消费者pool-1-thread-7正在消费数据906921688
生产者pool-1-thread-4生产数据2091628867
消费者pool-1-thread-13正在消费数据1475197572
消费者pool-1-thread-15正在消费数据-919544017
生产者pool-1-thread-2生产数据564860122
生产者pool-1-thread-2生产数据822954707
消费者pool-1-thread-14正在消费数据564860122
消费者pool-1-thread-10正在消费数据369331507
生产者pool-1-thread-1生产数据-245820912
消费者pool-1-thread-6正在消费数据822954707
生产者pool-1-thread-2生产数据1724595968
生产者pool-1-thread-2生产数据-1151855115
消费者pool-1-thread-12正在消费数据2091628867
生产者pool-1-thread-4生产数据-1774364499
生产者pool-1-thread-4生产数据2006106757
消费者pool-1-thread-14正在消费数据-1774364499
生产者pool-1-thread-3生产数据-1070853639
消费者pool-1-thread-9正在消费数据-1350268764
消费者pool-1-thread-11正在消费数据-1622505717
生产者pool-1-thread-5生产数据355412953
```

可以看出，使用 BlockingQueue 来实现生产者-消费者很简洁，这正是 BlockingQueue 的优势所在。

生产者-消费者模式的应用场景

生产者-消费者模式一般用于将生产数据的一方和消费数据的一方分割开来，将生产数据与消费数据的过程解耦开来。

01、Excutor 任务执行框架：

通过将任务的提交和任务的执行解耦开来，提交任务的操作相当于生产者，执行任务的操作相当于消费者。

例如使用 Excutor 构建 Web 服务器，用于处理线程的请求：生产者将任务提交给线程池，线程池创建线程处理任务，如果需要运行的任务数大于线程池的基本线程数，那么就把任务扔到阻塞队列（通过线程池+阻塞队列的方式比只使用一个阻塞队列的效率高很多，因为消费者能够处理就直接处理掉了，不用每个消费者都要先从阻塞队列中取出任务再执行）

02、消息中间件 MQ:

双十一的时候，会产生大量的订单，那么不可能同时处理那么多的订单，需要将订单放入一个队列里面，然后由专门的线程处理订单。

这里用户下单就是生产者，处理订单的线程就是消费者；再比如 12306 的抢票功能，先由一个容器存储用户提交的订单，然后再由专门处理订单的线程慢慢处理，这样可以在短时间内支持高并发服务。

03、任务的处理时间比较长的情况下：

比如上传附件并处理，那么这个时候可以将用户上传和处理附件分成两个过程，用一个队列暂时存储用户上传的附件，然后立刻返回用户上传成功，然后有专门的线程处理队列中的附件。

生产者-消费者模式的优点：

- 解耦：将生产者类和消费者类进行解耦，消除代码之间的依赖性，简化工作负载的管理
- 复用：通过将生产者类和消费者类独立开来，对生产者类和消费者类进行独立的复用与扩展
- 调整并发数：由于生产者和消费者的处理速度是不一样的，可以调整并发数，给予慢的一方多的并发数，来提高任务的处理速度
- 异步：对于生产者和消费者来说能够各司其职，生产者只需要关心缓冲区是否还有数据，不需要等待消费者处理完；对于消费者来说，也只需要关注缓冲区的内容，不需要关注生产者，通过异步的方式支持高并发，将一个耗时的流程拆成生产和消费两个阶段，这样生产者因为执行 put 的时间比较短，可以支持高并发
- 支持分布式：生产者和消费者通过队列进行通讯，所以不需要运行在同一台机器上，在分布式环境中可以通过 redis 的 list 作为队列，而消费者只需要轮询队列中是否有数据。同时还能支持集群的伸缩性，当某台机器宕掉的时候，不会导致整个集群宕掉

小结

本文主要讲解了线程的等待/通知机制，包括 wait/notify/notifyAll 方法的使用，以及使用 wait/notifyAll 实现生产者-消费者模型的示例代码。

还有 Condition 的 await/signalAll 方法的使用，以及使用 Condition 的 await/signalAll 实现生产者-消费者模型的示例代码。最后还讲解了使用 BlockingQueue 实现生产者-消费者模型的示例代码。

> 编辑：沉默王二，部分内容来自于CL0610的 GitHub 仓库https://github.com/CL0610/Java-concurrency，部分图片和内容来资源知乎这篇帖子。

GitHub 上标星 9300+ 的开源知识库《二哥的 Java 进阶之路》第二份 PDF 《并发编程小册》终于来了！包括线程的基本概念和使用方法、Java的内存模型、sychronized、volatile、CAS、AQS、ReentrantLock、线程池、并发容器、ThreadLocal、生产者消费者模型等面试和开发必须掌握的内容，共计 15 万余字，200+张手绘图，可以说是通俗易懂、风趣幽默……详情戳：太赞了，二哥的并发编程进阶之路.pdf

加入二哥的编程星球，在星球的第二个置顶帖「知识图谱」里就可以获取 PDF 版本。

第三十二节：配套教程

面试指南

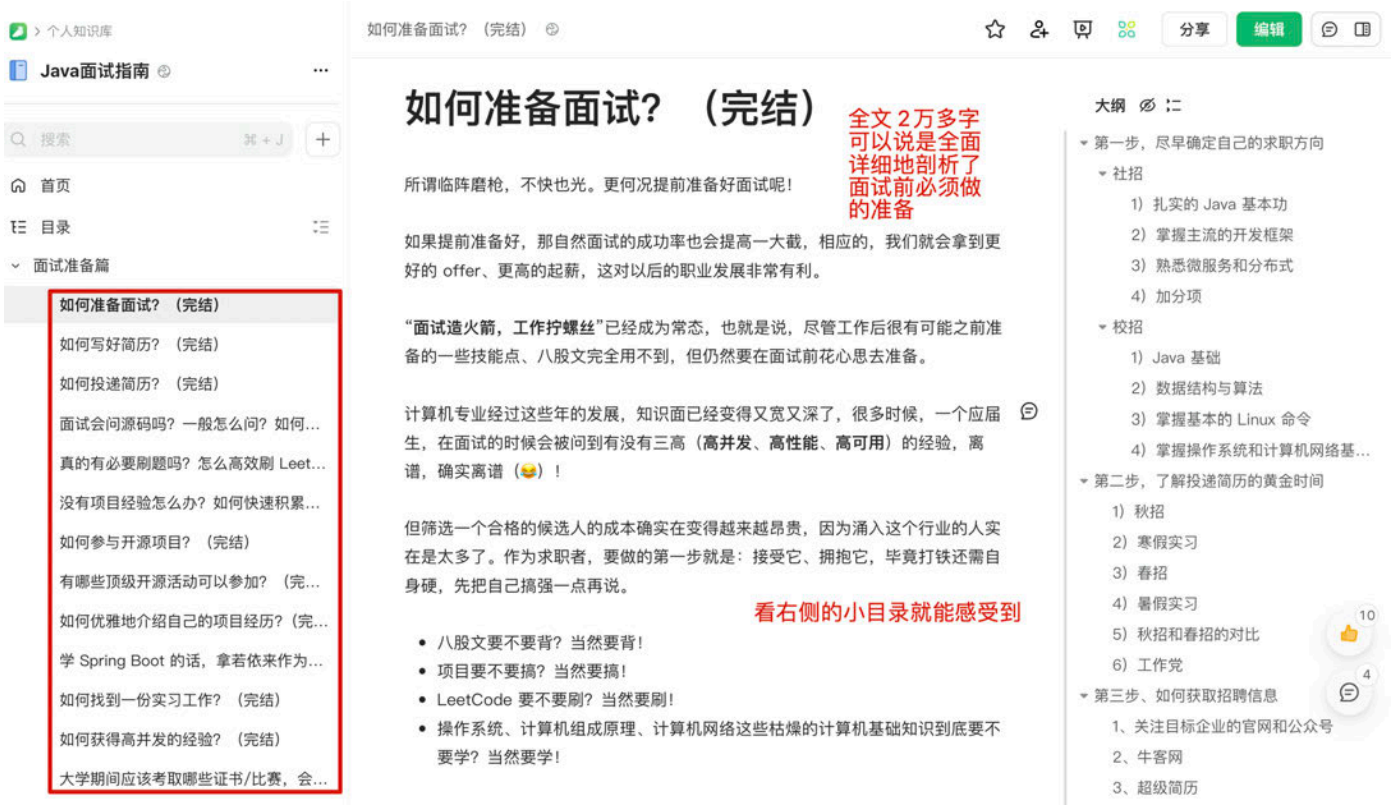
《Java 面试指南》是二哥编程星球的的一个付费专栏，和《Java 进阶之路》上的内容可以形成很好的互补，截止到目前，已经更新 48 万字，可以说是满满的干货和诚意。

一共分为 6 大板块，对面试、职场、技术、学习都会帮助特别大。

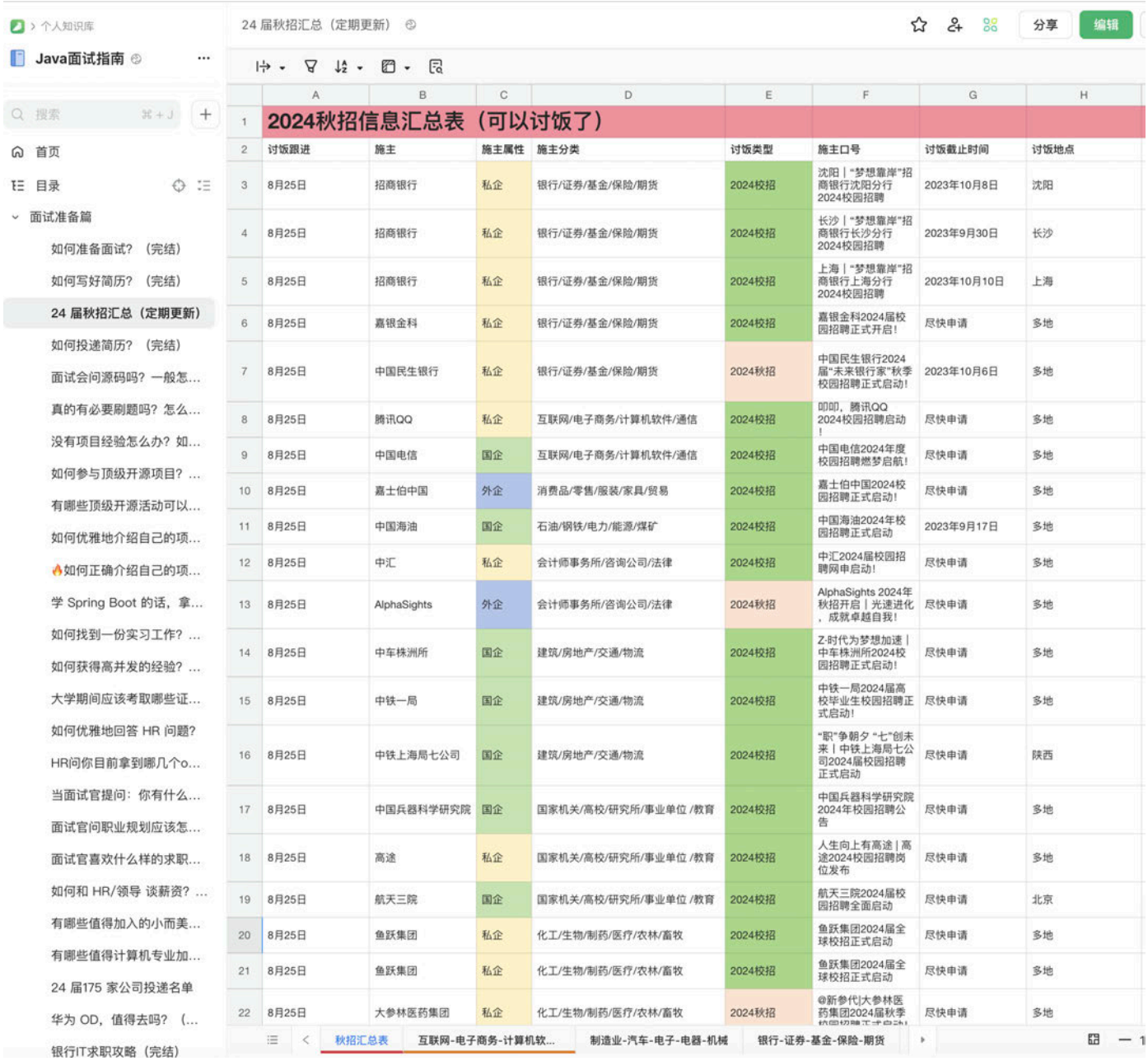
- 面试准备篇（25+篇），手把手教你如何准备面试。
- 职场修炼篇（11+篇），手摸手教你如何在职场中如鱼得水。
- 学习路线篇（13+篇），手勾手教你如何快速学习一门技术栈。
- 技术提升篇（33+篇），手拉手教你如何成为团队不可或缺的技术攻坚小能手。
- 面经分享篇（23+篇），手牵手教你如何在面试中知彼知己，百战不殆。
- 场景设计篇（22+篇），手握手教你如何在面试中脱颖而出。

01、面试准备篇

所谓临阵磨枪，不快也光。更何况提前做好充足的准备呢？这 25+篇内容会系统地引导你该如何做好面试准备。涉及到的主题有：简历、源码、LeetCode、项目经验、开源项目、高并发、证书、和 HR 对线、国企名单、公司投递名单、银行、谈薪等等面试常见问题。


个人知识库
Java面试指南
搜索
首页
目录
面试准备篇
如何准备面试？（完结）
如何写好简历？（完结）
如何投递简历？（完结）
面试会问源码吗？一般怎么问？如何...
真的有必要刷题吗？怎么高效刷 Leet...
没有项目经验怎么办？如何快速积累...
如何参与开源项目？（完结）
有哪些顶级开源活动可以参加？（完...
如何优雅地介绍自己的项目经历？（完...
学 Spring Boot 的话，拿若依来作为...
如何找到一份实习工作？（完结）
如何获得高并发的经验？（完结）
大学期间应该考取哪些证书/比赛，会...
如何准备面试？（完结）
分享
编辑
如何准备面试？（完结）
全文 2万多字可以说是全面详细地剖析了面试前必须做的准备
所谓临阵磨枪，不快也光。更何况提前准备好面试呢！
如果提前准备好，那自然面试的成功率也会提高一大截，相应的，我们就会拿到更好的 offer、更高的起薪，这对以后的职业发展非常有利。
“面试造火箭，工作拧螺丝”已经成为常态，也就是说，尽管工作后很有可能之前准备的一些技能点、八股文完全用不到，但仍然要在面试前花心思去准备。
计算机专业经过这些年的发展，知识面已经变得又宽又深了，很多时候，一个应届生，在面试的时候会被问到有没有三高（高并发、高性能、高可用）的经验，离谱，确实离谱（😂）！
但筛选一个合格的候选人的成本确实在变得越来越昂贵，因为涌入这个行业的人实在是太多了。作为求职者，要做的第一步就是：接受它、拥抱它，毕竟打铁还需自身硬，先把自己搞强一点再说。
看右侧的小目录就能感受到
八股文要不要背？当然要背！
项目要不要搞？当然要搞！
LeetCode 要不要刷？当然要刷！
操作系统、计算机组成原理、计算机网络这些枯燥的计算机基础知识到底要不要学？当然要学！
大纲
第一步，尽早确定自己的求职方向
社招
1）扎实的 Java 基本功
2）掌握主流的开发框架
3）熟悉微服务和分布式
4）加分项
校招
1）Java 基础
2）数据结构与算法
3）掌握基本的 Linux 命令
4）掌握操作系统和计算机网络基...
第二步，了解投递简历的黄金时间
1）秋招
2）寒假实习
3）春招
4）暑假实习
5）秋招和春招的对比
6）工作党
第三步、如何获取招聘信息
1、关注目标企业的官网和公众号
2、牛客网
3、超级简历
10
4


个人知识库
Java面试指南
搜索 ⌘+J
首页
目录
面试准备篇
如何准备面试？（完结）
如何写好简历？（完结）
24 届秋招汇总（定期更新）
如何投递简历？（完结）
面试会问源码吗？一般怎...
真的有必要刷题吗？怎么...
没有项目经验怎么办？如...
如何参与顶级开源项目？...
有哪些顶级开源活动可以...
如何优雅地介绍自己的项...
🔥如何正确介绍自己的项...
学 Spring Boot 的话，拿...
如何找到一份实习工作？...
如何获得高并发的经验？...
大学期间应该考取哪些证...
如何优雅地回答 HR 问题？
HR问你目前拿到哪几个o...
当面试官提问：你有什么...
面试官问职业规划应该怎...
面试官喜欢什么样的求职...
如何和 HR/领导 谈薪资？...
有哪些值得加入的小而美...
有哪些值得计算机专业加...
24 届175 家公司投递名单
华为 OD，值得去吗？（...
银行IT求职攻略（完结）
如何写好简历？（完结）
分享
编辑
社招2-3年模板案例
有球友在微信上问我“有没有社招 2 到 3 年好一点的简历模板啊？”这里就给大家分享一个，我之前的一个读者（工作 2 年多，最近刚面试）的面试情况和他在写简历时的注意事项，那希望能给球友们一些参考和启发。
在整个面试过程中，问的最多的几个问题：
1.Java 本地锁到分布式锁，各种锁的场景，为什么要用，以及不同锁实现方式的底层，优缺点，还有 volitale
2.hashmap，这个就不用多说了，put 过程啊，为什么线程不安全，1.7 和 1.8 的区别，为什么要用红黑树等等，可问的很多
3.多线程实现方式，线程池核心参数，运行过程，有什么问题需要注意的
4.jvm 方面，cms 问的比较多，和 g1 的区别，还有 rootsearching，类加载过程，jvm 内存模型以及各个模块运用
5.redis 哨兵同步，投票选举，集群模式，持久化方式，zset 实现方式
6.dubbo 调用链路， 其 spi 和 java 的有什么区别
7.mysql 索引优化思路，事务 mvcc，日志系统，主从同步， buffer pool，分库分表等
8.zookeeper 脑裂问题，leader 选举过程
9.spring bean 生命周期，循环依赖，ioc 和 aop，事务实现方式等
10.kafka 高吞吐原因，丢失消息的场景，副本维护，leader 选举，消息幂等性保证等
对于 2-3 年工作经历的话，整个面试中问的很多的还是对于「基础、还有各个框架的理解」，这也是最基础的内容，还会有一些「设计选型」，如果你是项目负责人的话，需要知道为何要选某个框架，还有一些兜底，都是需要自己去做的
再来看看读者的这个「简历是如何写」的。
专业技能
给大家看下「专业技能」栏，读者在这里写的基本都是他记得很熟的，不熟悉的框架尽量不要写上去，比如 rabbitmq 虽然你可能看过，但不是很熟，就不要乱写，免得面试中被问到回答不上来。
专业技能
1.具备扎实的 Java 基础以及面向对象编程思想，熟悉 Git、Maven 等进行项目版本管理
大纲
核心注意事项
简历模板
简历内容
基本资料
教育背景
专业技能
项目经历
个人总结
再次强调
1）基本错误尽量减少，比如这样：
2）千万不要 Word，就用 PDF
3）HR/面试官关注的点
4）简历模板
社招2-3年模板案例
专业技能
项目经历
面试过程需要注意什么
1.跟着面试官的节奏回答问题
2.让面试官跟着自己的节奏来问
3.避重就轻，快速逃离
4.学会猜
5.别紧张，放轻松
6.先面小公司
问题回复
学历重要吗？
算法重要吗？
谈薪不敢要怎么办？



讨饭跟进	施主	施主属性	施主分类	讨饭类型	施主口号	讨饭截止时间	讨饭地点
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	沈阳丨“梦想靠岸”招商银行沈阳分行2024校园招聘	2023年10月8日	沈阳
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	长沙丨“梦想靠岸”招商银行长沙分行2024校园招聘	2023年9月30日	长沙
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	上海丨“梦想靠岸”招商银行上海分行2024校园招聘	2023年10月10日	上海
8月25日	嘉银金科	私企	银行/证券/基金/保险/期货	2024校招	嘉银金科2024届校园招聘正式开启！	尽快申请	多地
8月25日	中国民生银行	私企	银行/证券/基金/保险/期货	2024秋招	中国民生银行2024届“未来银行家”秋季校园招聘正式启动！	2023年10月6日	多地
8月25日	腾讯QQ	私企	互联网/电子商务/计算机软件/通信	2024校招	叩叩，腾讯QQ 2024校园招聘启动！	尽快申请	多地
8月25日	中国电信	国企	互联网/电子商务/计算机软件/通信	2024校招	中国电信2024年度校园招聘燃梦启航！	尽快申请	多地
8月25日	嘉士伯中国	外企	消费品/零售/服装/家具/贸易	2024校招	嘉士伯中国2024校园招聘正式启动！	尽快申请	多地
8月25日	中国海油	国企	石油/钢铁/电力/能源/煤矿	2024校招	中国海油2024年校园招聘正式启动	2023年9月17日	多地
8月25日	中汇	私企	会计师事务所/咨询公司/法律	2024校招	中汇2024届校园招聘网申启动！	尽快申请	多地
8月25日	AlphaSights	外企	会计师事务所/咨询公司/法律	2024秋招	AlphaSights 2024年秋招开启丨光速进化，成就卓越自我！	尽快申请	多地
8月25日	中车株洲所	国企	建筑/房地产/交通/物流	2024校招	Z·时代为梦想加速丨中车株洲所2024校园招聘正式启动！	尽快申请	多地
8月25日	中铁一局	国企	建筑/房地产/交通/物流	2024校招	中铁一局2024届高校毕业生校园招聘正式启动！	尽快申请	多地
8月25日	中铁上海局七公司	国企	建筑/房地产/交通/物流	2024校招	“职”争朝夕 “七”创未来丨中铁上海局七公司2024届校园招聘正式启动	尽快申请	陕西
8月25日	中国兵器科学研究院	国企	国家机关/高校/研究所/事业单位 /教育	2024校招	中国兵器科学研究院2024年校园招聘公告	尽快申请	多地
8月25日	高途	私企	国家机关/高校/研究所/事业单位 /教育	2024校招	人生向上有高途丨高途2024校园招聘岗位发布	尽快申请	多地
8月25日	航天三院	国企	国家机关/高校/研究所/事业单位 /教育	2024校招	航天三院2024届校园招聘全面启动	尽快申请	北京
8月25日	鱼跃集团	私企	化工/生物/制药/医疗/农林/畜牧	2024校招	鱼跃集团2024届全球校招正式启动	尽快申请	多地
8月25日	鱼跃集团	私企	化工/生物/制药/医疗/农林/畜牧	2024校招	鱼跃集团2024届全球校招正式启动	尽快申请	多地
8月25日	大参林医药集团	私企	化工/生物/制药/医疗/农林/畜牧	2024秋招	@新参代|大参林医药集团2024届秋季	尽快申请	多地

02、职场修炼篇

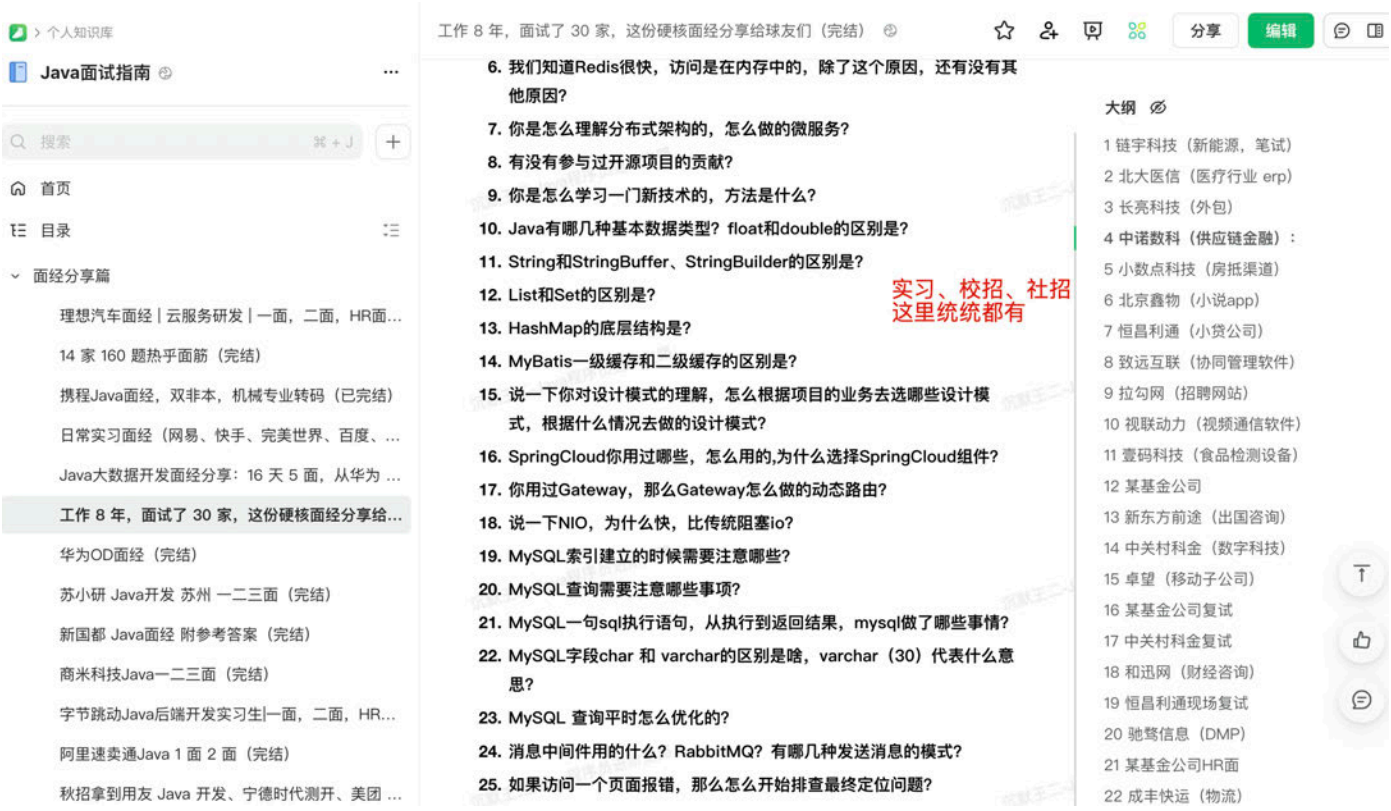
如何平滑度过试用期？如何平滑度过 35 岁程序员危机？如何在繁重的工作中持续成长？如何做副业？如何赚零花钱？如何达到 30 万+年薪等等，都是大家迫切关心的问题，这 11+篇内容会一一为你揭晓答案。

03、技术提升篇

编程能力、技术功底，是我们程序员安身立命之本，是我们求职/工作的最核心的武器。

04、面经分享篇

知彼知己，方能百战不殆，我们必须得站在前辈的肩膀上，才能走得更远更快。他们在面试中遇到过哪些经典的问题，我们能不能提前演练一下，对临场发挥有着至关重要的作用。



05、场景设计题篇

有些面试官不喜欢问八股文，反而更喜欢结合项目问一些非常经典的场景题，这种场景题没有标准的答案，但却很能考察一名求职者的逻辑思维能力。

更多优质专栏

除了《Java 面试指南》专栏，二哥编程星球还提供了：《技术派实战教程》、《编程喵实战笔记》、《二哥的 LeetCode 刷题笔记》、《算法突击 50 题》、《华为 OD 笔试 AB 卷题库》等五个额外的专栏。

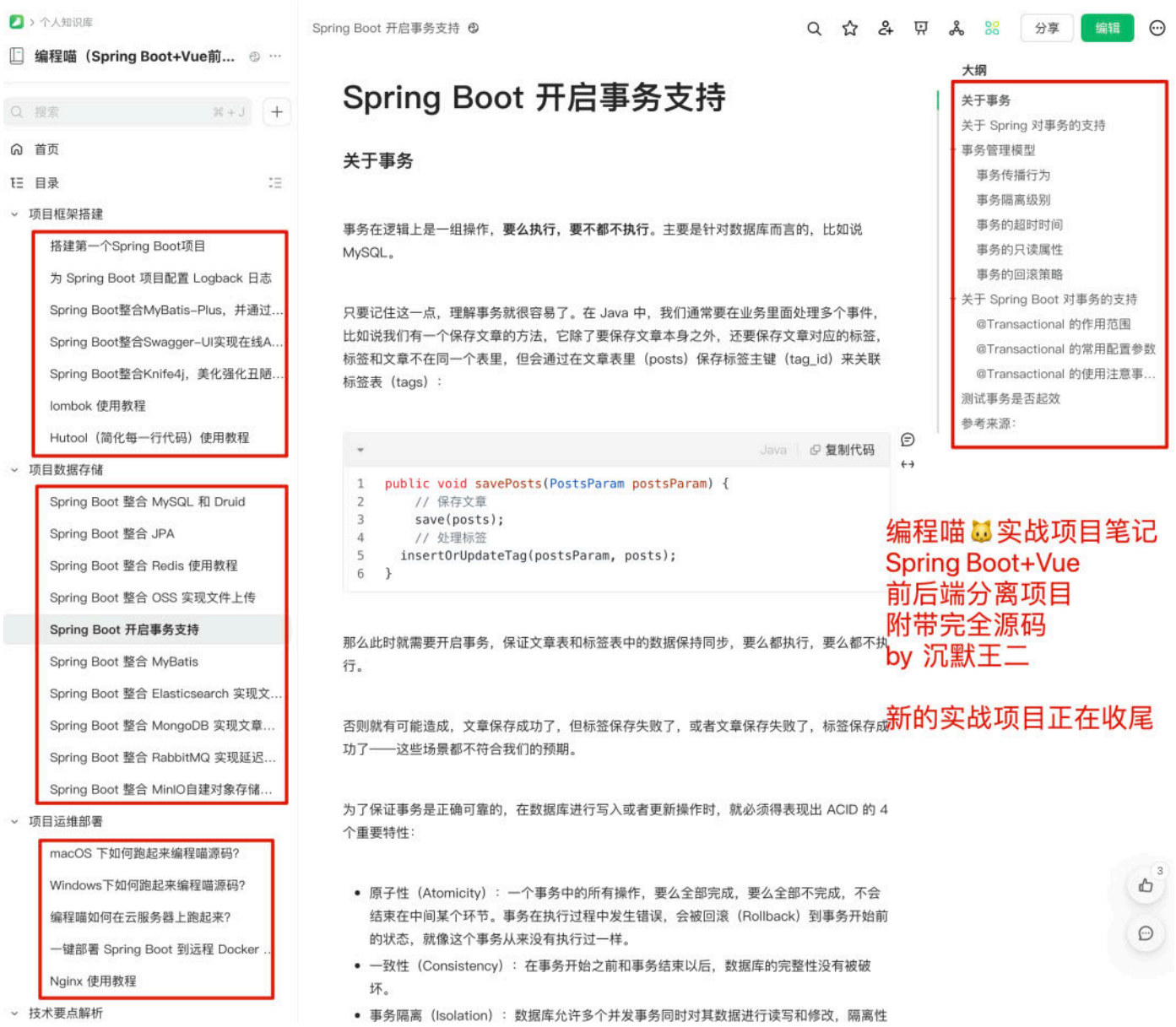
01、技术派实战教程

技术派是一个基于 Spring Boot、MyBatis-Plus、MySQL、Redis、ElasticSearch、MongoDB、Docker、RabbitMQ 等技术栈实现的社区系统，采用主流的互联网技术架构、全新的UI设计、支持一键源码部署，拥有完整的文章&教程发布/搜索/评论/统计流程等，代码完全开源，没有任何二次封装，是一个非常适合二次开发/实战的现代化社区项目👍 。

下面是《技术派教程》部分目录（包括大厂篇、基础篇、进阶篇、工程篇、扩展篇、前端篇，目前已完成 80+篇），很多球友都反馈说光这套教程就值 599 元。

02、编程喵实战笔记

编程喵是一套成熟的学习教程网站，包括前台网站内容展示系统，以及后台网站内容管理系统，采用时下最流行的 Spring Boot + Vue 的前后端分离架构。配套的教程可以带你完成从小白到初级工程师的蜕变。

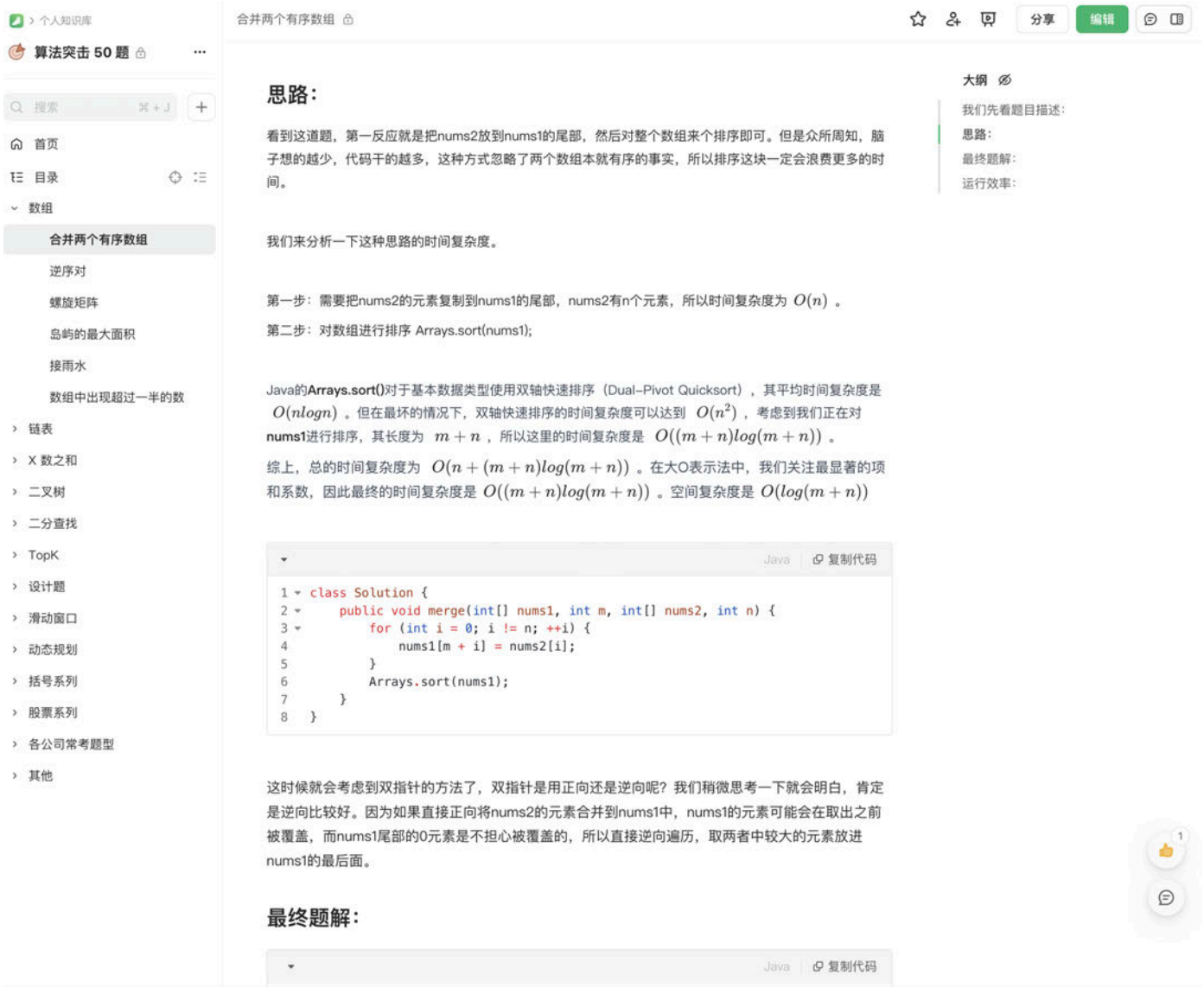


03、二哥的 LeetCode 刷题笔记

《二哥的 LeetCode 刷题笔记》，不仅有详细地解题思路，还有完整的代码示例，力求教会你举一反三的解题能力。

04、算法突击 50 题

准备秋招/春招/社招的小伙伴不少，但往往时间比较紧张，很多小伙伴精力有限，所以我这里精选了 50 道高频算法题，作为笔试的重点突击题型，可以在短时间内最大效率地提升你的笔试通过率。



05、华为 OD 笔试 AB 卷题库

LeetCode 的模式是你只需要输入核心代码就可以了，华为 OD 机考使用的是 ACM 模式，也就是需要手动编写输入输出的模式，更贴近机考的真实场景，所以我这里整理了一套完整的 AB 卷题库，帮助大家在最短的时间内快速提升机考的通过率。

星球限时优惠

一年前，星球的定价还是 99 元一年，第一批优惠券的额度是 30 元，等于说 69 元的低价就可以加入，再扣除掉星球手续费，几乎就是纯粹做公益。

随着时间的推移，星球积累的干货/资源越来越多，我花在星球上的时间也越来越多，星球的知识图谱里沉淀的问题，你可以戳这个链接去感受一下。有学习计划啊、有学生党秋招&春招&offer选择&考研&实习&专升本&培训班的问题啊、有工作党方向选择&转行&求职&职业规划的问题啊，还有大大小小的技术细节，我都竭尽全力去帮助球友，并且得到了球友的认可和尊重。

目前星球已经 3300+ 人了，所以星球也涨价到了 129 元，后续会讲星球的价格调整为 149 元/年，所以想加入的小伙伴一定要趁早。

沉默王二

邀请你加入星球，一起学习

Java程序员进阶之路

星主：沉默王二

3300+	1.5万	554
成员数量	内容数量	运营天数

这是一个编程学习指南+Java项目实战+leetcode刷题的高性价比圈子，欢迎想进步的小伙伴！加入后先阅读星球的两个置顶帖，你会发现物超所值✌。...

知识星球

微信扫码加入星球 ▶

你可以微信扫码或者长按自动识别领取 30 元优惠券，**99/年** 加入，满 4000 人会涨价至 149 元，所以想要加入的话请趁早。

对了，**加入星球后记得花 10 分钟时间看一下星球的两个置顶贴，你会发现物超所值**！

成功没有一蹴而就，没有一飞冲天，但只要你能够一步一个脚印，就能取得你心满意足的好结果，请给自己一个机会！

最后，把二哥的座右铭送给你：**没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟**。

共勉 。

如何贡献?

对了，如果你在阅读的过程中遇到一些错误，欢迎到我的开源仓库提交 issue、PR（审核通过后可成为 Contributor），我会第一时间修正，感谢你为后来者做出的贡献。

> - GitHub：https://github.com/itwanger/toBeBetterJavaer
> - 码云：https://gitee.com/itwanger/toBeBetterJavaer

更新记录

V1.0 版 2023年09月04日

第一版《二哥的并发编程进阶之路》正式完结发布！