目录

RedSpider社区介绍

RedSpider技术社区始于2018年秋的成都。

在本书发布时，社区共五位活跃成员，均参与了本书的撰写及审校工作。他们的代号分别是（排名按拼音排序，不分先后）：灵鹤，毛毛虫，潘帕斯雄鹰，萤火虫，啄木鸟。

活跃

RedSpider是一个非常活跃的技术社区，虽然现在成员人数不多，但是对于社区的各种活动都非常积极和活跃。目前社区内部大多都是Java开发工程师，且都对技术始终保持极高的热情。

RedSpider是一个自我驱动的技术学习社区，社区内部拥有非常丰富的活动形式，比如一起开技术沙龙、一起写博客、一起写书、一起写代码。

除了与技术相关的活动以外，我们还会偶尔线下聚餐和一起出去运动。虽然有一位成员不在成都，但仍然会“云参与”到我们的每次线下活动。

敏捷

RedSpider技术社区是一个敏捷的技术社区。我们通过一些敏捷流程和工具来管理社区的目标和进程。比如在写这本书的时候，我们使用了国内的Teambition软件来追踪每篇文章的初稿和修订状态。

我们每两周一个迭代，我们会在迭代开始时安排下个迭代要做的事情，在迭代结束的时候开迭代会议（Retro）总结这个迭代。

我们使用Git来管理整个书籍的编写工作，托管到GitHub上。

到目前为止总共有两百多次提交记录，地址：
https://github.com/RedSpider1/concurrent。

开源与分享

经社区成员一致决定，本书将发布到GitBook。一方面是觉得出版纸质书比较麻烦，要联系出版商以及排版成Word。另一方面，也算是想免费向大众分享RedSpider社区几个月以来的成果，开源本书也算是社区对中国Java开发者做出一点力所能及的贡献。

交流群

社区创建了微信群，欢迎广大读者及对技术感兴趣的朋友加入交流群：

请先加微信号：**redspider-worker**，备注“**RedSpider技术社区交流群申请**”，我们会尽快通过并拉你进群哦。

作者介绍

本书由五位作者（均为RedSpider社区成员）共同撰写。他们均参与了初稿编写、文章修订等工作。

下面分别是五位作者的简介（以拼音排序）：

灵鹤

招银网络科技高级开发工程师，拥有国外大型企业高并发项目经验，熟悉敏捷流程开发及持续集成，热爱开源，喜欢分享，对于写作始终保持严谨的态度。

毛毛虫

高级Java开发工程师，长期工作在华为一线，专注Java技术体系，熟悉微服务及大数据领域。拥有大数据项目开发经验和运维经验、丰富的企业级项目开发经验。热爱开源，乐于分享。目前专注于微服务，JVM，大数据生态系统。

潘帕斯雄鹰

Java开发工程师，长期位于Java开发一线，拥有丰富的大型企业级项目开发经验，熟悉敏捷开发流程。热衷于探索未知，目前专注于Java高并发，JVM。

萤火虫

阿里巴巴高级Java开发工程师，熟悉微服务及DevOps领域，拥有大型国际高并发项目的开发经验和运维经验。对新技术有强烈的好奇心，热爱钻研技术深度，自我驱动能力和学习能力较强。熟悉敏捷实践。热爱开源和分享，具有丰富的演讲经验和写作经验。

啄木鸟

某外包憨憨Java开发工程师，驻阿里巴巴达摩院一线开发，拥有国内外分布式架构高并发项目开发经验，主要从事微服务架构下API设计与开发。对于敏捷实践与持续集成/交付有丰富经验。拥抱开源，善于钻研技术难点。热爱与人沟通，有丰富的培训团队和新人经验。

文章地址

- 深入浅出Java多线程【强烈推荐！！！】
- 【GithubBook】深入浅出Java多线程
- PDF (PDF生成比较麻烦，不会每个提交都生成，**建议看网页版本**)

本地运行

先安装gitbook

```
npm install gitbook-cli -g
```

然后安装gitbook插件

```
gitbook install
```

运行服务

```
gitbook serve .
```

然后就可以在本地 http://localhost:4000访问了。

本书简介

笔者在读完市面上关于Java并发编程的资料后，感觉有些知识点不是很清晰，于是在RedSpider社区内展开了对Java并发编程原理的讨论。鉴于开源精神，我们决定将我们讨论之后的Java并发编程原理整理成书籍，分享给大家。

站在巨人的肩上，我们可以看得更远。本书内容的主要来源有博客、书籍、论文，对于一些已经叙述得很清晰的知识点我们直接引用在本书中；对于一些没有讲解清楚的知识点，我们加以画图或者编写Demo进行加工；而对于一些模棱两可的知识点，本书在查阅了大量资料的情况下，给出最合理的解释。

写本书的过程也是对自己研究和掌握的技术点进行整理的过程，希望本书能帮助读者快速掌握并发编程技术。

如果您或者您的单位愿意赞助本书或本社区，请发送邮件到RedSpider社区邮件组redspider@qun.mail.163.com或加微信**redspider-worker**进行洽谈。

勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见，可以在我们的github上新建issue，笔者会尽快解答，期待能够得到你的真挚反馈。github地址：https://github.com/RedSpider1/concurrent

公众号

欢迎关注微信公众号“编了个程”，每周会更新一篇Java方面的**原创技术文章**

第一章 进程与线程的基本概念

1.1 进程产生的背景

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。

线程的提出

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》，本文不做深入详解。

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。**

多进程的方式也可以实现并发，为什么我们要使用多线程?

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比较容易。
- 进程是重量级的，而线程是轻量级的，故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 。

1.2 上下文切换

上下文切换（有时也称做进程切换或任务切换）是指 CPU 从一个进程（或线程）切换到另一个进程（或线程）。上下文是指**某一时间点 CPU 寄存器和程序计数器的内容。**

> 寄存器是cpu内部的少量的速度很快的闪存，通常存储和访问计算过程的中间值提高计算机程序的运行速度。
>
> 程序计数器是一个专用的寄存器，用于表明指令序列中 CPU 正在执行的位置，存的值为正在执行的指令的位置或者下一个将要被执行的指令的位置，具体实现依赖于特定的系统。
>
> 举例说明 线程A - B
>
> 1.先挂起线程A，将其在cpu中的状态保存在内存中。
>
> 2.在内存中检索下一个线程B的上下文并将其在 CPU 的寄存器中恢复,执行B线程。
>
> 3.当B执行完，根据程序计数器中指向的位置恢复线程A。

CPU通过为每个线程分配CPU时间片来实现多线程机制。CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。

但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

上下文切换通常是计算密集型的，意味着此操作会**消耗大量的 CPU 时间，故线程也不是越多越好。**如何减少系统中上下文切换次数，是提升多线程性能的一个重点课题。

参考资料

- 线程的几种状态转换
- 进程和线程的由来与别
- 进程、线程、多线程相关总结
- 进程的概念/标识/结构/状态
- 操作系统 - 进程的概念
- 进程管理笔记一、进程的概念及其产生的背景
- 上下文切换
- 进程的概念/标识/结构/状态
- 线程的生命周期及状态转换详解
- 进程与线程

第二章 Java多线程入门类和接口

2.1 Thread类和Runnable接口

上一章我们了解了操作系统中多线程的基本概念。那么在Java中，我们是如何使用多线程的呢?

首先，我们需要有一个“线程”类。JDK提供了 `Thread` 类和 `Runnable` 接口来让我们实现自己的“线程”类。

- 继承 `Thread` 类，并重写 `run` 方法；
- 实现 `Runnable` 接口的 `run` 方法；

2.1.1 继承Thread类

先学会怎么用，再学原理。首先我们来看看怎么用 `Thread` 和 `Runnable` 来写一个Java多线程程序。

首先是继承 `Thread` 类：

```
public class Demo {
    public static class MyThread extends Thread {
        @Override
        public void run() {
            System.out.println("MyThread");
        }
    }

    public static void main(String[] args) {
        Thread myThread = new MyThread();
        myThread.start();
    }
}
```

注意要调用 `start()` 方法后，该线程才算启动!

> 我们在程序里面调用了start()方法后，虚拟机会先为我们创建一个线程，然后等到这个线程第一次得到时间片时再调用run()方法。
>
> 注意不可多次调用start()方法。在第一次调用start()方法后，再次调用start()方法会抛出IllegalThreadStateException异常。

2.1.2 实现Runnable接口

接着我们来看一下 `Runnable` 接口(JDK 1.8 +):

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

可以看到 `Runnable` 是一个函数式接口，这意味着我们可以使用**Java 8的函数式编程**来简化代码。

示例代码:

```
public class Demo {
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            System.out.println("MyThread");
        }
    }

    public static void main(String[] args) {

        new Thread(new MyThread()).start();

        // Java 8 函数式编程，可以省略MyThread类
        new Thread(() -> {
            System.out.println("Java 8 匿名内部类");
        }).start();
    }
}
```

2.1.3 Thread类构造方法

`Thread` 类是一个 `Runnable` 接口的实现类，我们来看看 `Thread` 类的源码。

查看 `Thread` 类的构造方法，发现其实是简单调用一个私有的 `init` 方法来实现初始化。 `init` 的方法签名:

```
// Thread类源码

// 片段1 - init方法
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals)

// 片段2 - 构造函数调用init方法
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}

// 片段3 - 使用在init方法里初始化AccessControlContext类型的私有属性
this.inheritedAccessControlContext =
    acc != null ? acc : AccessController.getContext();

// 片段4 - 两个对用于支持ThreadLocal的私有属性
ThreadLocal.ThreadLocalMap threadLocals = null;
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

我们挨个来解释一下 `init` 方法的这些参数：

- g：线程组，指定这个线程是在哪个线程组下；
- target：指定要执行的任务；
- name：线程的名字，多个线程的名字是可以重复的。如果不指定名字，见片段2；
- acc：见片段3，用于初始化私有变量 `inheritedAccessControlContext` 。

  > 这个变量有点神奇。它是一个私有变量，但是在 `Thread` 类里只有 `init` 方法对它进行初始化，在 `exit` 方法把它设为 `null` 。其它没有任何地方使用它。一般我们是不会使用它的，那什么时候会使用到这个变量呢？可以参考这个stackoverflow的问题：Restrict permissions to threads which execute third party software；

- inheritThreadLocals：可继承的 `ThreadLocal` ，见片段4， `Thread` 类里面有两个私有属性来支持 `ThreadLocal` ，我们会在后面的章节介绍 `ThreadLocal` 的概念。

实际情况下，我们大多是直接调用下面两个构造方法：

```
Thread(Runnable target)
Thread(Runnable target, String name)
```

2.1.4 Thread类的几个常用方法

这里介绍一下Thread类的几个常用的方法：

- currentThread()：静态方法，返回对当前正在执行的线程对象的引用；
- start()：开始执行线程的方法，java虚拟机会调用线程内的run()方法；
- yield()：yield在英语里有放弃的意思，同样，这里的yield()指的是当前线程愿意让出对当前处理器的占用。这里需要注意的是，就算当前线程调用了yield()方法，程序在调度的时候，也还有可能继续运行这个线程的；
- sleep()：静态方法，使当前线程睡眠一段时间；
- join()：使当前线程等待另一个线程执行完毕之后再继续执行，内部调用的是Object类的wait方法实现的；

2.1.5 Thread类与Runnable接口的比较：

实现一个自定义的线程类，可以有继承 `Thread` 类或者实现 `Runnable` 接口这两种方式，它们之间有什么优劣呢？

- 由于Java“单继承，多实现”的特性，Runnable接口使用起来比Thread更灵活。
- Runnable接口出现更符合面向对象，将线程单独进行对象的封装。
- Runnable接口出现，降低了线程对象和线程任务的耦合性。
- 如果使用线程时不需要使用Thread类的诸多方法，显然使用Runnable接口更为轻量。

所以，我们通常优先使用“实现 `Runnable` 接口”这种方式来自定义线程类。

2.2 Callable、Future与FutureTask

通常来说，我们使用 `Runnable` 和 `Thread` 来创建一个新的线程。但是它们有一个弊端，就是 `run` 方法是没有返回值的。而有时候我们希望开启一个线程去执行一个任务，并且这个任务执行完成后有一个返回值。

JDK提供了 `Callable` 接口与 `Future` 接口为我们解决这个问题，这也是所谓的“异步”模型。

2.2.1 Callable接口

`Callable` 与 `Runnable` 类似，同样是只有一个抽象方法的函数式接口。不同的是， `Callable` 提供的方法是**有返回值**的，而且支持**泛型**。

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

那一般是怎么使用 `Callable` 的呢？ `Callable` 一般是配合线程池工具 `ExecutorService` 来使用的。我们会在后续章节解释线程池的使用。这里只介绍 `ExecutorService` 可以使用 `submit` 方法来让一个 `Callable` 接口执行。它会返回一个 `Future` ，我们后续的程序可以通过这个 `Future` 的 `get` 方法得到结果。

这里可以看一个简单的使用demo：

```
// 自定义Callable
class Task implements Callable<Integer>{
    @Override
    public Integer call() throws Exception {
        // 模拟计算需要一秒
        Thread.sleep(1000);
        return 2;
    }
    public static void main(String args[]) throws Exception {
        // 使用
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        Future<Integer> result = executor.submit(task);
        // 注意调用get方法会阻塞当前线程，直到得到结果。
        // 所以实际编码中建议使用可以设置超时时间的重载get方法。
        System.out.println(result.get());
    }
}
```

输出结果：

```
2
```

2.2.2 Future接口

`Future` 接口只有几个比较简单的方法：

```
public abstract interface Future<V> {
    public abstract boolean cancel(boolean paramBoolean);
    public abstract boolean isCancelled();
    public abstract boolean isDone();
    public abstract V get() throws InterruptedException, ExecutionException;
    public abstract V get(long paramLong, TimeUnit paramTimeUnit)
            throws InterruptedException, ExecutionException, TimeoutException;
}
```

`cancel` 方法是试图取消一个线程的执行。

注意是**试图**取消，**并不一定能取消成功**。因为任务可能已完成、已取消、或者一些其它因素不能取消，存在取消失败的可能。 `boolean` 类型的返回值是“是否取消成功”的意思。参数 `paramBoolean` 表示是否采用中断的方式取消线程执行。

所以有时候，为了让任务有能够取消的功能，就使用 `Callable` 来代替 `Runnable` 。如果为了可取消性而使用 `Future` 但又不提供可用的结果，则可以声明 `Future<?>` 形式类型、并返回 `null` 作为底层任务的结果。

2.2.3 FutureTask类

上面介绍了 `Future` 接口。这个接口有一个实现类叫 `FutureTask` 。 `FutureTask` 是实现的 `RunnableFuture` 接口的，而 `RunnableFuture` 接口同时继承了 `Runnable` 接口和 `Future` 接口：

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

那 `FutureTask` 类有什么用？为什么要有一个 `FutureTask` 类？前面说到了 `Future` 只是一个接口，而它里面的 `cancel` ， `get` ， `isDone` 等方法要自己实现起来都是**非常复杂**的。所以JDK提供了一个 `FutureTask` 类来供我们使用。

示例代码：

```
// 自定义Callable，与上面一样
class Task implements Callable<Integer>{
    @Override
    public Integer call() throws Exception {
        // 模拟计算需要一秒
        Thread.sleep(1000);
        return 2;
    }
    public static void main(String args[]) throws Exception {
        // 使用
        ExecutorService executor = Executors.newCachedThreadPool();
        FutureTask<Integer> futureTask = new FutureTask<>(new Task());
        executor.submit(futureTask);
        System.out.println(futureTask.get());
    }
}
```

使用上与第一个Demo有一点小的区别。首先，调用 `submit` 方法是没有返回值的。这里实际上是调用的 `submit(Runnable task)` 方法，而上面的Demo，调用的是 `submit(Callable<T> task)` 方法。

然后，这里是使用 `FutureTask` 直接取 `get` 取值，而上面的Demo是通过 `submit` 方法返回的 `Future` 去取值。

在很多高并发的环境下，有可能Callable和FutureTask会创建多次。FutureTask能够在高并发环境下**确保任务只执行一次**。这块有兴趣的同学可以参看FutureTask源码。

2.2.4 FutureTask的几个状态

```
/**
  *
  * state可能的状态转变路径如下:
  * NEW -> COMPLETING -> NORMAL
  * NEW -> COMPLETING -> EXCEPTIONAL
  * NEW -> CANCELLED
  * NEW -> INTERRUPTING -> INTERRUPTED
  */
private volatile int state;
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

> state表示任务的运行状态，初始状态为NEW。运行状态只会在set、setException、cancel方法中终止。COMPLETING、INTERRUPTING是任务完成后的瞬时状态。

以上就是Java多线程几个基本的类和接口的介绍。可以打开JDK看看源码，体会这几个类的设计思路和用途吧！

参考资料

- Java语言定义的线程状态分析
- Java线程状态分析
- FutureTask源码分析

第三章 线程组和线程优先级

3.1 线程组(ThreadGroup)

Java中用ThreadGroup来表示线程组，我们可以使用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()方法线程的名字是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：

```
public class Demo {
    public static void main(String[] args) {
        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().getThr
        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
执行main方法线程名字：main
testThread当前线程组名字：main
testThread线程名字：Thread-0
```

ThreadGroup管理着它下面的Thread，ThreadGroup是一个标准的**向下引用**的树状结构，这样设计的原因是**防止"上级"线程被"下级"线程引用而无法有效地被GC回收。**

3.2 线程的优先级

Java中线程优先级可以指定，范围是1~10。但是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**在操作系统的优先级**是多少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**更高的几率**得到执行。我们使用方法 `Thread` 类的 `setPriority()` 实例方法来设定线程的优先级。

```
public class Demo {
    public static void main(String[] args) {
        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1-10的级别来设定了线程的优先级，这时候可能有些读者会问，那么我是不是可以在业务实现的时候，采用这种方法来指定一些线程执行的先后顺序?

对于这个问题，我们的答案是:No!

Java中的优先级来说不是特别的可靠，**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决定的。**

我们通过代码来验证一下：

```
public class Demo {
    public static class T1 extends Thread {
        @Override
        public void run() {
            super.run();
            System.out.println(String.format("当前执行的线程是：%s，优先级：%d",
                    Thread.currentThread().getName(),
                    Thread.currentThread().getPriority()));
        }
    }

    public static void main(String[] args) {
        IntStream.range(1, 10).forEach(i -> {
            Thread thread = new Thread(new T1());
            thread.setPriority(i);
            thread.start();
        });
    }
}
```

某次输出：

```
当前执行的线程是：Thread-17，优先级：9
当前执行的线程是：Thread-1，优先级：1
当前执行的线程是：Thread-13，优先级：7
当前执行的线程是：Thread-11，优先级：6
当前执行的线程是：Thread-15，优先级：8
当前执行的线程是：Thread-7，优先级：4
当前执行的线程是：Thread-9，优先级：5
当前执行的线程是：Thread-3，优先级：2
当前执行的线程是：Thread-5，优先级：3
```

Java提供一个**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**，优先级高的线程比优先级低的线程会有更大的几率优先执行。在优先级相同的情况下，按照“先到先得”的原则。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。

还有一种线程称为**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。
>
> 应用场景是：当所有非守护线程结束时，结束其余的子线程（守护线程）自动关闭，就免去了还要继续关闭子线程的麻烦。
>
> 一个线程默认是非守护线程，可以通过Thread类的setDaemon(boolean on)来设置。

在之前，我们有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将会怎样呢？我们用下面的案例来验证一下：

```
public static void main(String[] args) {
    ThreadGroup threadGroup = new ThreadGroup("t1");
    threadGroup.setMaxPriority(6);
    Thread thread = new Thread(threadGroup,"thread");
    thread.setPriority(9);
    System.out.println("我是线程组的优先级"+threadGroup.getMaxPriority());
    System.out.println("我是线程的优先级"+thread.getPriority());
}
```

输出：

> 我是线程组的优先级6
> 我是线程的优先级6

所以，如果某个线程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。

3.3 线程组的常用方法及数据结构

3.3.1 线程组的常用方法

获取当前的线程组名字

```
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

```
package com.func.axc.threadgroup;

public class ThreadGroupDemo {
    public static void main(String[] args) {
        ThreadGroup threadGroup1 = new ThreadGroup("group1") {
            // 继承ThreadGroup并重新定义以下方法
            // 在线程成员抛出unchecked exception
            // 会执行此方法
            public void uncaughtException(Thread t, Throwable e) {
                System.out.println(t.getName() + ": " + e.getMessage());
            }
        };

        // 这个线程是threadGroup1的一员
        Thread thread1 = new Thread(threadGroup1, new Runnable() {
            public void run() {
                // 抛出unchecked异常
                throw new RuntimeException("测试异常");
            }
        });

        thread1.start();
    }
}
```

3.3.2 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; // ThreadGroupr 的名称
    int maxPriority; // 线程最大优先级
    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
}
```

然后看看构造函数：

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

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

// 构造函数
public ThreadGroup(ThreadGroup parent, String name) {
    this(checkParentAccess(parent), parent, name);
}

// 私有构造函数，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);
}
```

第三个构造函数里调用了 `checkParentAccess` 方法，这里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，以及是否是在允许执行该操作的安全上下文中执行它。应用程序可以允许或不允许该操作。
>
> 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.）

总结来说，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的权限的作用。

参考资料

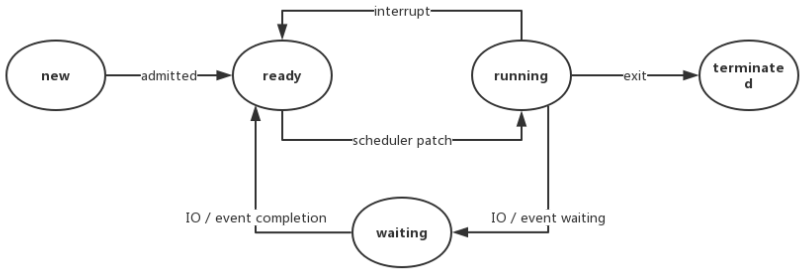
- https://blog.csdn.net/Evankaka/article/details/51627380
- 《Java并发编程实践》
- 《Java多线程编程核心技术》

第四章 Java线程的状态及主要转化方法

4.1 操作系统中的线程状态转换

首先我们来看看操作系统中的线程状态转换。

> 在现在的操作系统中，线程是被视为轻量级进程的，所以**操作系统线程的状态其实和操作系统进程的状态是一致的。**



操作系统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可进入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）。

4.2 Java线程的6个状态

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

4.2.1 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：

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    group.add(this);

    boolean started = false;
    try {
        start0();
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {

        }
    }
}
```

我们可以看到，在start()内部，这里有一个threadStatus的变量。如果它不等于0，调用start()是会直接抛出异常的。

我们接着往下看，有一个native的 `start0()` 方法。这个方法里并没有对**threadStatus**的处理。到了这里我们仿佛就拿这个threadStatus没辙了，我们通过debug的方式再看一下:

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

我是在start()方法内部的最开始打的断点，叙述下在我这里打断点看到的结果：

- 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：
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;
    }
}
```

所以，我们结合上面的源码可以得到引申的两个问题的结果：

> 两个问题的答案都是不可行，在调用一次start()之后，threadStatus的值会改变（threadStatus !=0），此时再次调用start()方法会抛出IllegalThreadStateException异常。
>
> 比如，threadStatus为2代表当前线程状态为TERMINATED。

4.2.2 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。

Java中线程的RUNNABLE状态

看了操作系统线程的几个状态之后我们来看看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.
 */
```

> Java线程的**RUNNABLE**状态其实是包括了传统操作系统线程的**ready**和**running**两个状态的。

4.2.3 BLOCKED

阻塞状态。处于BLOCKED状态的线程正等待锁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。

4.2.4 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。

调用如下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实例的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。

我们延续上面的例子继续解释一下WAITING状态：

> 你等了好几分钟现在终于轮到你了，突然你们有一个“不懂事”的经理突然来了。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。

4.2.5 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.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：

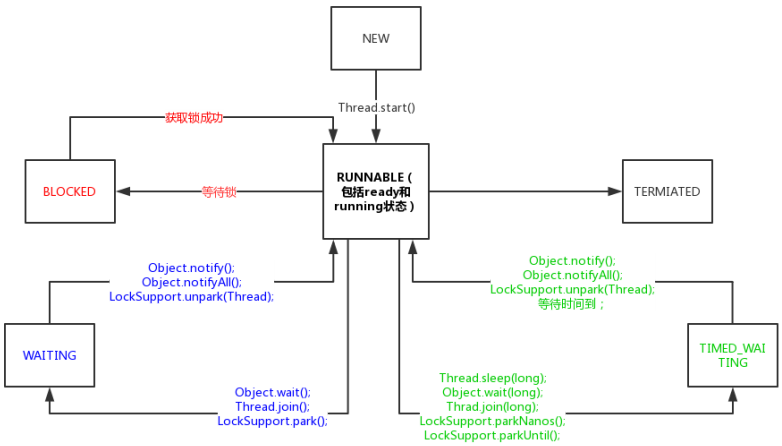
> 到了第二天中午，又到了饭点，你还是到了窗口前。
>
> 突然间想起你的同事叫你等他一起，他说让你等他十分钟他改个bug。
>
> 好吧，你说那你就等等吧，你就离开了窗口。很快十分钟过去了，你见他还没来，你想都等了这么久了还不来，那你还是先去吃饭好了。
>
> 这时你还是线程t1，你改bug的同事是线程t2。t2让t1等待了指定时间，此时t1等待期间就属于TIMED_WATING状态。
>
> t1等待10分钟后，就自动唤醒，拥有了去争夺锁的资格。

4.2.6 TERMINATED

终止状态。此时线程已执行完毕。

4.3 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**：



4.3.1 BLOCKED与RUNNABLE状态的转换

我们在上面说到：处于BLOCKED状态的线程是因为在等待锁的释放。假如这里有两个线程a和b，a线程提前获得了锁并且暂未释放锁，此时b就处于BLOCKED状态。我们先来看一个例子：

```
@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）。

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200
    b.start();
    System.out.println(a.getName() + ":" + a.getState()); // 输出?
    System.out.println(b.getName() + ":" + b.getState()); // 输出?
}
```

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。

4.3.2 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）。

4.3.3 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）。

4.3.4 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里还没有安全直接的方法来停止线程，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。线程的中断状态受这个方法的影响，意思是调用一次使线程中断状态设置为true，连续调用两次会使得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而定，可以在合适的实际处理中断请求，也可以完全不处理继续执行下去。

参考资料

- JDK 1.8 源码
- 深入Thread.sleep
- 不可不说的Java“锁”事
- Java线程状态分析
- Java线程和操作系统线程的关系
- 详细分析 Java 中断机制

第五章 Java线程间的通信

合理的使用Java多线程可以更好地利用服务器资源。一般来讲，线程内部有自己私有的线程上下文，互不干扰。但是当我们需要多个线程之间相互协作的时候，就需要我们掌握Java线程的通信方式。本文将介绍Java线程之间的几种通信原理。

5.1 锁与同步

在Java中，锁的概念都是基于对象的，所以我们又经常称它为对象锁。线程和锁的关系，我们可以用婚姻关系来理解。一个锁同一时间只能被一个线程持有。也就是说，一个锁如果和一个线程“结婚”（持有），那其他线程如果需要得到这个锁，就得等这个线程和这个锁“离婚”（释放）。

在我们的线程之间，有一个同步的概念。什么是同步呢，假如我们现在有2位正在抄暑假作业答案的同学：线程A和线程B。当他们正在抄的时候，老师突然来修改了一些答案，可能A和B最后写出的暑假作业就不一样。我们为了A,B能写出2本相同的暑假作业，我们就需要让老师先修改答案，然后A，B同学再抄。或者A，B同学先抄完，老师再修改答案。这就是线程A，线程B的线程同步。

可以解释为：线程同步是线程之间按照**一定的顺序**执行。

为了达到线程同步，我们可以使用锁来实现它。

我们先来看看一个无锁的程序：

```
public class NoneLock {

    static class ThreadA implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                System.out.println("Thread A " + i);
            }
        }
    }

    static class ThreadB implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                System.out.println("Thread B " + i);
            }
        }
    }

    public static void main(String[] args) {
        new Thread(new ThreadA()).start();
        new Thread(new ThreadB()).start();
    }
}
```

执行这个程序，你会在控制台看到，线程A和线程B各自独立工作，输出自己的打印值。如下是我的电脑上某一次运行的结果。每一次运行结果都会不一样。

```
....
Thread A 48
Thread A 49
Thread B 0
Thread A 50
Thread B 1
Thread A 51
Thread A 52
....
```

那我现在有一个需求，我想等A先执行完之后，再由B去执行，怎么办呢？最简单的方式就是使用一个“对象锁”：

```
public class ObjectLock {
    private static Object lock = new Object();

    static class ThreadA implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                for (int i = 0; i < 100; i++) {
                    System.out.println("Thread A " + i);
                }
            }
        }
    }

    static class ThreadB implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                for (int i = 0; i < 100; i++) {
                    System.out.println("Thread B " + i);
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(new ThreadA()).start();
        Thread.sleep(10);
        new Thread(new ThreadB()).start();
    }
}
```

这里声明了一个名字为 `lock` 的对象锁。我们在 `ThreadA` 和 `ThreadB` 内需要同步的代码块里，都是用 `synchronized` 关键字加上了同一个对象锁 `lock` 。

上文我们说到了，根据线程和锁的关系，同一时间只有一个线程持有一个锁，那么线程B就会等线程A执行完成后释放 `lock` ，线程B才能获得锁 `lock` 。

> 这里在主线程里使用sleep方法睡眠了10毫秒，是为了防止线程B先得到锁。因为如果同时start，线程A和线程B都是出于就绪状态，操作系统可能会先让B运行。这样就会先输出B的内容，然后B执行完成之后自动释放锁，线程A再执行。

5.2 等待/通知机制

上面一种基于“锁”的方式，线程需要不断地去尝试获得锁，如果失败了，再继续尝试。这可能会耗费服务器资源。

而等待/通知机制是另一种方式。

Java多线程的等待/通知机制是基于 `Object` 类的 `wait()` 方法和 `notify()` ，`notifyAll()` 方法来实现的。

> notify()方法会随机叫醒一个正在等待的线程，而notifyAll()会叫醒所有正在等待的线程。

前面我们讲到，一个锁同一时刻只能被一个线程持有。而假如线程A现在持有了一个锁 `lock` 并开始执行，它可以使用 `lock.wait()` 让自己进入等待状态。这个时候， `lock` 这个锁是被释放了的。

这时，线程B获得了 `lock` 这个锁并开始执行，它可以在某一时刻，使用 `lock.notify()` ，通知之前持有 `lock` 锁并进入等待状态的线程A，说“线程A你不用等了，可以往下执行了”。

> 需要注意的是，这个时候线程B并没有释放锁 `lock` ，除非线程B这个时候使用 `lock.wait()` 释放锁，或者线程B执行结束自行释放锁，线程A才能得到 `lock` 锁。

我们用代码来实现一下：

```
public class WaitAndNotify {
    private static Object lock = new Object();

    static class ThreadA implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                for (int i = 0; i < 5; i++) {
                    try {
                        System.out.println("ThreadA: " + i);
                        lock.notify();
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                lock.notify();
            }
        }
    }

    static class ThreadB implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                for (int i = 0; i < 5; i++) {
                    try {
                        System.out.println("ThreadB: " + i);
                        lock.notify();
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                lock.notify();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(new ThreadA()).start();
        Thread.sleep(1000);
        new Thread(new ThreadB()).start();
    }
}

// 输出:
ThreadA: 0
ThreadB: 0
ThreadA: 1
ThreadB: 1
ThreadA: 2
ThreadB: 2
ThreadA: 3
ThreadB: 3
ThreadA: 4
ThreadB: 4
```

在这个Demo里，线程A和线程B首先打印出自己需要的东西，然后使用 `notify()` 方法叫醒另一个正在等待的线程，然后自己使用 `wait()` 方法陷入等待并释放 `lock` 锁。

> 需要注意的是等待/通知机制使用的是使用同一个对象锁，如果你两个线程使用的是不同的对象锁，那它们之间是不能用等待/通知机制通信的。

5.3 信号量

JDK提供了一个类似于“信号量”功能的类 `Semaphore` 。但本文不是要介绍这个类，而是介绍一种基于 `volatile` 关键字的自己实现的信号量通信。

后面会有专门的章节介绍 `volatile` 关键字，这里只是做一个简单的介绍。

> volatile关键字能够保证内存的可见性，如果用volatile关键字声明了一个变量，在一个线程里面改变了这个变量的值，那其它线程是立马可见更改后的值的。

比如我现在有一个需求，我想让线程A输出0，然后线程B输出1，再然后线程A输出2…以此类推。我应该怎样实现呢?

代码：

```
public class Signal {
    private static volatile int signal = 0;

    static class ThreadA implements Runnable {
        @Override
        public void run() {
            while (signal < 5) {
                if (signal % 2 == 0) {
                    System.out.println("threadA: " + signal);
                    signal++;
                }
            }
        }
    }

    static class ThreadB implements Runnable {
        @Override
        public void run() {
            while (signal < 5) {
                if (signal % 2 == 1) {
                    System.out.println("threadB: " + signal);
                    signal = signal + 1;
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(new ThreadA()).start();
        Thread.sleep(1000);
        new Thread(new ThreadB()).start();
    }
}

// 输出:
threadA: 0
threadB: 1
threadA: 2
threadB: 3
threadA: 4
```

我们可以看到，使用了一个 `volatile` 变量 `signal` 来实现了“信号量”的模型。这里需要注意的是， `volatile` 变量需要进行原子操作。

需要注意的是， `signal++` 并不是一个原子操作，所以我们在实际开发中，会根据需要使用 `synchronized` 给它“上锁”，或者是使用 `AtomicInteger` 等原子类。并且上面的程序也**并不是线程安全的**，因为执行 `while` 语句后，可能当前线程就暂停等待时间片了，等线程醒来，可能signal已经大于等于5了。

> 这种实现方式并不一定高效，本例只是演示信号量

信号量的应用场景：

假如在一个停车场中，车位是我们的公共资源，线程就如同车辆，而看门的管理员就是起的“信号量”的作用。

因为在这种场景下，多个线程（超过2个）需要相互合作，我们用简单的“锁”和“等待通知机制”就不那么方便了。这个时候就可以用到信号量。

其实JDK中提供的很多多线程通信工具类都是基于信号量模型的。我们会在后面第三篇的文章中介绍一些常用的通信工具类。

5.4 管道

管道是基于“管道流”的通信方式。JDK提供了 `PipedWriter` 、 `PipedReader` 、 `PipedOutputStream` 、 `PipedInputStream` 。其中，前面两个是基于字符的，后面两个是基于字节流的。

这里的示例代码使用的是基于字符的：

```
public class Pipe {
    static class ReaderThread implements Runnable {
        private PipedReader reader;

        public ReaderThread(PipedReader reader) {
            this.reader = reader;
        }

        @Override
        public void run() {
            System.out.println("this is reader");
            int receive = 0;
            try {
                while ((receive = reader.read()) != -1) {
                    System.out.print((char)receive);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    static class WriterThread implements Runnable {

        private PipedWriter writer;

        public WriterThread(PipedWriter writer) {
            this.writer = writer;
        }

        @Override
        public void run() {
            System.out.println("this is writer");
            int receive = 0;
            try {
                writer.write("test");
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException {
        PipedWriter writer = new PipedWriter();
        PipedReader reader = new PipedReader();
        writer.connect(reader); // 这里注意一定要连接，才能通信

        new Thread(new ReaderThread(reader)).start();
        Thread.sleep(1000);
        new Thread(new WriterThread(writer)).start();
    }
}

// 输出:
this is reader
this is writer
test
```

我们通过线程的构造函数，传入了 `PipedWrite` 和 `PipedReader` 对象。可以简单分析一下这个示例代码的执行流程：

1. 线程ReaderThread开始执行，
2. 线程ReaderThread使用管道reader.read()进入”阻塞“，
3. 线程WriterThread开始执行，
4. 线程WriterThread用writer.write("test")往管道写入字符串，
5. 线程WriterThread使用writer.close()结束管道写入，并执行完毕，
6. 线程ReaderThread接受到管道输出的字符串并打印，
7. 线程ReaderThread执行完毕。

管道通信的应用场景：

这个很好理解。使用管道多半与I/O流相关。当我们一个线程需要先另一个线程发送一个信息（比如字符串）或者文件等等时，就需要使用管道通信了。

5.5 其它通信相关

以上介绍了一些线程间通信的基本原理和方法。除此以外，还有一些与线程通信相关的知识点，这里一并介绍。

5.5.1 join方法

join()方法是Thread类的一个实例方法。它的作用是让当前线程陷入“等待”状态，等join的这个线程执行完成后，再继续执行当前线程。

有时候，主线程创建并启动了子线程，如果子线程中需要进行大量的耗时运算，主线程往往将早于子线程结束之前结束。

如果主线程想等待子线程执行完毕后，获得子线程中的处理完的某个数据，就要用到join方法了。

示例代码：

```
public class Join {
    static class ThreadA implements Runnable {

        @Override
        public void run() {
            try {
                System.out.println("我是子线程，我先睡一秒");
                Thread.sleep(1000);
                System.out.println("我是子线程，我睡完了一秒");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new ThreadA());
        thread.start();
        thread.join();
        System.out.println("如果不加join方法，我会先被打出来，加了就不一样了");
    }
}
```

> 注意join()方法有两个重载方法，一个是join(long)， 一个是join(long, int)。
>
> 实际上，通过源码你会发现，join()方法及其重载方法底层都是利用了wait(long)这个方法。
>
> 对于join(long, int)，通过查看源码(JDK 1.8)发现，底层并没有精确到纳秒，而是对第二个参数做了简单的判断和处理。

5.5.2 sleep方法

sleep方法是Thread类的一个静态方法。它的作用是让当前线程睡眠一段时间。它有这样两个方法：

- Thread.sleep(long)
- Thread.sleep(long, int)

> 同样，查看源码(JDK 1.8)发现，第二个方法貌似只对第二个参数做了简单的处理，没有精确到纳秒。实际上还是调用的第一个方法。

这里需要强调一下：**sleep方法是不会释放当前的锁的，而wait方法会。**这也是最常见的一个多线程面试题。

它们还有这些区别：

- wait可以指定时间，也可以不指定；而sleep必须指定时间。
- wait释放cpu资源，同时释放锁；sleep释放cpu资源，但是不释放锁，所以易死锁。
- wait必须放在同步块或同步方法中，而sleep可以在任意位置。

5.5.3 ThreadLocal类

ThreadLocal是一个本地线程副本变量工具类。内部是一个**弱引用**的Map来维护。这里不详细介绍它的原理，而是只是介绍它的使用，以后有独立章节来介绍ThreadLocal类的原理。

有些朋友称ThreadLocal为**线程本地变量**或**线程本地存储**。严格来说，ThreadLocal类并不属于多线程间的通信，而是让每个线程有自己”独立“的变量，线程之间互不影响。它为每个线程都创建一个**副本**，每个线程可以访问自己内部的副本变量。

ThreadLocal类最常用的就是set方法和get方法。示例代码：

```
public class ThreadLocalDemo {
    static class ThreadA implements Runnable {
        private ThreadLocal<String> threadLocal;

        public ThreadA(ThreadLocal<String> threadLocal) {
            this.threadLocal = threadLocal;
        }

        @Override
        public void run() {
            threadLocal.set("A");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("ThreadA输出: " + threadLocal.get());
        }

        static class ThreadB implements Runnable {
            private ThreadLocal<String> threadLocal;

            public ThreadB(ThreadLocal<String> threadLocal) {
                this.threadLocal = threadLocal;
            }

            @Override
            public void run() {
                threadLocal.set("B");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("ThreadB输出: " + threadLocal.get());
            }
        }

        public static void main(String[] args) {
            ThreadLocal<String> threadLocal = new ThreadLocal<>();
            new Thread(new ThreadA(threadLocal)).start();
            new Thread(new ThreadB(threadLocal)).start();
        }
    }
}

// 输出:
ThreadA输出: A
ThreadB输出: B
```

可以看到，虽然两个线程使用的同一个ThreadLocal实例（通过构造方法传入），但是它们各自可以存取自己当前线程的一个值。

那ThreadLocal有什么作用呢？如果只是单纯的想要线程隔离，在每个线程中声明一个私有变量就好了呀，为什么要使用ThreadLocal？

如果开发者希望将类的某个静态变量（user ID或者transaction ID）与线程状态关联，则可以考虑使用ThreadLocal。

最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等。数据库连接和Session管理涉及多个复杂对象的初始化和关闭。如果在每个线程中声明一些私有变量来进行操作，那这个线程就变得不那么“轻量”了，需要频繁的创建和关闭连接。

5.5.4 InheritableThreadLocal

InheritableThreadLocal类与ThreadLocal类稍有不同，Inheritable是继承的意思。它不仅仅是当前线程可以存取副本值，而且它的子线程也可以存取这个副本值。

参考资料

- JDK 1.8 源码
- 深入理解线程通信
- JAVA多线程之线程间的通信方式
- 线程通信

第六章 Java内存模型基础知识

6.1 并发编程模型的两个关键问题

- 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操作发生的相对顺序

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表所示：

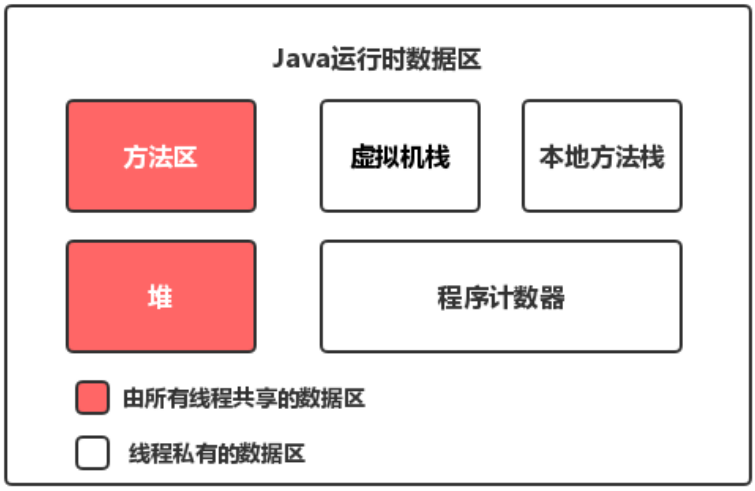
	如何通信	如何同步
消息传递并发模型	线程之间**没有公共状态**，线程间的通信必须通过发送消息来显示进行通信。	发送消息天然同步，因为发送消息总是在接受消息之前，因此同步是隐式的。
共享内存并发模型	线程之间共享程序的公共状态，通过**写-读内存中的公共状态**进行隐式通信。	**必须显式指定某段代码需要在线程之间互斥执行**，同步是显式的。

在Java中，使用的是共享内存并发模型。

6.2 Java内存模型的抽象结构

6.2.1 运行时内存的划分

先谈一下运行时数据区，下面这张图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、方法定义参数、异常处理器参数）不会在线程之间共享，也就不会有内存可见性（下文会说到）的问题，也不受内存模型的影响。而在堆中的变量是共享的，本文称为共享变量。

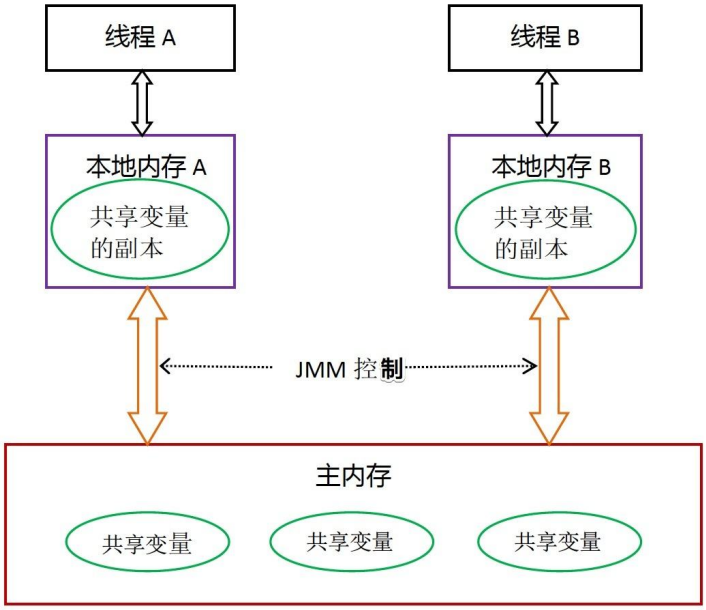
所以，内存可见性是针对的**共享变量**。

6.2.2 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主内存中，每个线程都有一个私有的本地内存，存储了该线程以读、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定义了线程和主内存之间的抽象关系。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内存中。
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：
    i. 线程A将本地内存A中更新过的共享变量刷新到主内存中去。
    ii. 线程B到主内存中去读取线程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内存。

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，不能直接从主内存中读取。**

所以线程B并不是直接去主内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去主内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。

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通过控制主内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。为了程序员的方便理解，提出了happens-before，它更加的简单易懂，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。这里涉及到的所有内容后面都会有专门的章节介绍。

6.2.3 JMM与Java内存区域划分的区别与联系

上面两小节分别提到了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

  两者是不同的概念层次。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、可见性等展开的。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中的主内存属于共享数据区域，他是包含了堆和方法区；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。

实际上，他们表达的是同一种含义，这里不做区分。

参考资料

- 《Java并发编程的艺术》
- 《实战Java高并发程序设计》

第七章 重排序与happens-before

7.1 什么是重排序?

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。

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执行结束之后再执行指令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个代码的执行情况：

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

先加载b、c（**注意，即有可能先加载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，也就是增加了停顿，那么后面的指令也会依次有停顿,这降低了计算机的执行效率。

为了减少这个停顿，我们可以先加载e和f,然后再去加载add(b,c),这样做对程序（串行）是没有影响的,但却减少了停顿。既然add(b,c)需要停顿，那还不如去做一些有意义的事情。

综上所述，**指令重排对于提高CPU处理性能十分必要。虽然由此带来了乱序的问题，但是这点牺牲是值得的。**

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，可以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。

7.2 顺序一致性模型与JMM的保证

顺序一致性模型是一个**理论参考模型**，内存模型在设计的时候都会以顺序一致性内存模型作为参考。

7.2.1 数据竞争与顺序一致性

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；如果一个线程程序能够正确同步，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。

这里的同步包括了使用 `volatile` 、 `final` 、 `synchronized` 等关键字来实现**多线程下的同步。**

如果程序员没有正确使用 `volatile` 、 `final` 、 `synchronized` ，那么即便是使用了同步（单线程下的同步），JMM也不会有内存可见性的保证，可能会导致你的程序出错，并且具有不可重现性，很难排查。

所以如何正确使用 `volatile` 、 `final` 、 `synchronized` ，是程序员应该去了解的。后面会有专门的章节介绍这几个关键字的内存语义及使用。

7.2.2 顺序一致性模型

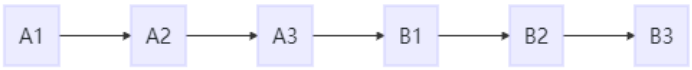
顺序一致性内存模型是一个**理想化的理论参考模型**，它为程序员提供了极强的内存可见性保证。

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。**

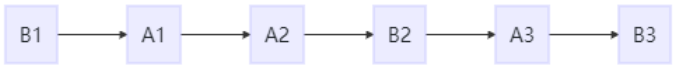
为了理解这两个特性，我们举个例子，假设有两个线程A和B并发执行，线程A有3个操作，他们在程序中的顺序是A1->A2->A3，线程B也有3个操作，B1->B2->B3。

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。

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，在没有刷新到主内存之前，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内存之后，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。

7.2.3 JMM中同步程序的顺序一致性效果

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。

7.2.4 JMM中未同步程序的顺序一致性效果

对于未同步的多线程程序，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。

7.3 happens-before

7.3.1 什么是happens-before?

一方面，程序员需要JMM提供一个强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，这样它们就可以最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。**

而对于程序员，JMM提供了**happens-before规则**（JSR-133规范），满足了程序员的需求——**简单易懂，并且提供了足够强的内存可见性保证。**换言之，程序员只要遵循happens-before规则，那他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以内，也可以是不同的线程之间。因此，JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证。

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。**

7.3.2 天然的happens-before关系

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。

参考资料

- 《Java并发编程的艺术》

第八章 volatile

8.1 几个基本概念

在介绍volatile之前，我们先回顾及介绍几个基本的概念。

8.1.1 内存可见性

在Java内存模型那一章我们介绍了JMM有一个主内存，每个线程有自己私有的工作内存，工作内存中保存了一些变量在主内存的拷贝。

内存可见性，指的是线程之间的可见性，当一个线程修改了共享变量时，另一个线程可以读取到这个修改后的值。

8.1.2 重排序

为优化程序性能，对原有的指令执行顺序进行优化重新排序。重排序可能发生在多个阶段，比如编译重排序、CPU重排序等。

8.1.3 happens-before规则

是一个给程序员使用的规则，只要程序员在写代码的时候遵循happens-before规则，JVM就能保证指令在多线程之间的顺序性符合程序员的预期。

8.2 volatile的内存语义

在Java中，volatile关键字有特殊的内存语义。volatile主要有以下两个功能：

- 保证变量的**内存可见性**
- 禁止volatile变量与普通变量**重排序**（JSR133提出，Java 5 开始才有这个“增强的volatile内存语义”）

8.2.1 内存可见性

以一段示例代码开始：

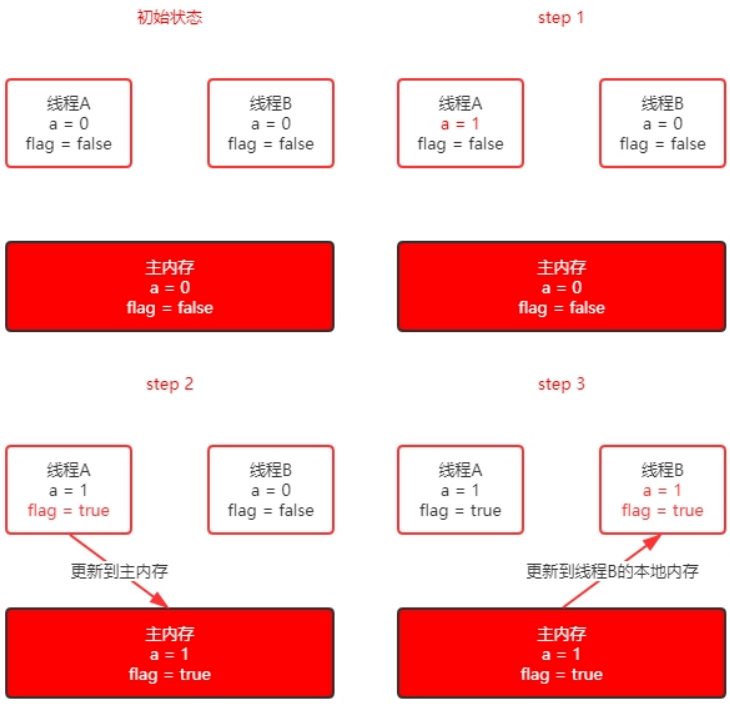
```
public class VolatileExample {
    int a = 0;
    volatile boolean flag = false;

    public void writer() {
        a = 1; // step 1
        flag = true; // step 2
    }

    public void reader() {
        if (flag) { // step 3
            System.out.println(a); // step 4
        }
    }
}
```

在这段代码里，我们使用 `volatile` 关键字修饰了一个 `boolean` 类型的变量 `flag` 。

所谓内存可见性，指的是当一个线程对 `volatile` 修饰的变量进行**写操作**（比如step 2）时，JMM会立即把该线程对应的本地内存中的共享变量的值刷新到主内存；当一个线程对 `volatile` 修饰的变量进行**读操作**（比如step 3）时，JMM会把立即该线程对应的本地内存置为无效，从主内存中读取共享变量的值。

> 在这一点上，volatile与锁具有相同的内存效果，volatile变量的写和锁的释放具有相同的内存语义，volatile变量的读和锁的获取具有相同的内存语义。

假设在时间线上，线程A先执行方法 `writer` 方法，线程B后执行 `reader` 方法。那必然会有下图：



而如果 `flag` 变量**没有**用 `volatile` 修饰，在step 2，线程A的本地内存里面的变量就不会立即更新到主内存，那随后线程B也同样不会去主内存拿最新的值，仍然使用线程B本地内存缓存的变量的值 `a = 0, flag = false` 。

8.2.1 禁止重排序

在JSR-133之前的旧的Java内存模型中，是允许volatile变量与普通变量重排序的。那上面的案例中，可能就会被重排序成下列时序来执行：

1. 线程A写volatile变量，step 2，设置flag为true；
2. 线程B读同一个volatile，step 3，读取到flag为true；
3. 线程B读普通变量，step 4，读取到 a = 0；
4. 线程A修改普通变量，step 1，设置 a = 1；

可见，如果volatile变量与普通变量发生了重排序，虽然volatile变量能保证内存可见性，也可能导致普通变量读取错误。

所以在旧的内存模型中，volatile的写-读就不能与锁的释放-获取具有相同的内存语义了。为了提供一种比锁更轻量级的**线程间的通信机制**，**JSR-133**专家组决定增强volatile的内存语义：严格限制编译器和处理器对volatile变量与普通变量的重排序。

编译器还好说，JVM是怎么还能限制处理器的重排序的呢？它是通过**内存屏障**来实现的。

什么是内存屏障？硬件层面，内存屏障分两种：读屏障（Load Barrier）和写屏障（Store Barrier）。内存屏障有两个作用：

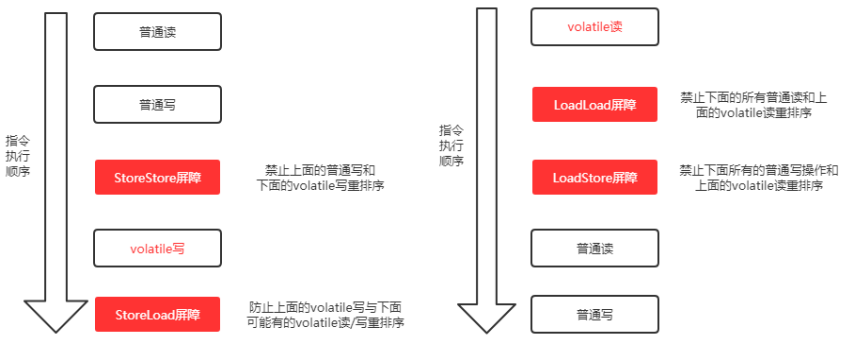
1. 阻止屏障两侧的指令重排序；
2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，或者让缓存中相应的数据失效。

> 注意这里的缓存主要指的是CPU缓存，如L1，L2等

编译器在**生成字节码时**，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。编译器选择了一个**比较保守的JMM内存屏障插入策略**，这样可以保证在任何处理器平台，任何程序中都能得到正确的volatile内存语义。这个策略是：

- 在每个volatile写操作前插入一个StoreStore屏障；
- 在每个volatile写操作后插入一个StoreLoad屏障；
- 在每个volatile读操作后插入一个LoadLoad屏障；
- 在每个volatile读操作后再插入一个LoadStore屏障。

大概示意图是这个样子：



> 再逐个解释一下这几个屏障。注：下述Load代表读操作，Store代表写操作
>
> **LoadLoad屏障**：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
> **StoreStore屏障**：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，这个屏障会把Store1强制刷新到内存，保证Store1的写入操作对其它处理器可见。
> **LoadStore屏障**：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
> **StoreLoad屏障**：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的（冲刷写缓冲器，清空无效化队列）。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能

对于连续多个volatile变量读或者连续多个volatile变量写，编译器做了一定的优化来提高性能，比如：

> 第一个volatile读;
>
> LoadLoad屏障;
>
> 第二个volatile读;
>
> LoadStore屏障

再介绍一下volatile与普通变量的重排序规则:

1. 如果第一个操作是volatile读，那无论第二个操作是什么，都不能重排序;
2. 如果第二个操作是volatile写，那无论第一个操作是什么，都不能重排序;
3. 如果第一个操作是volatile写，第二个操作是volatile读，那不能重排序。

举个例子，我们在案例中step 1，是普通变量的写，step 2是volatile变量的写，那符合第2个规则，这两个steps不能重排序。而step 3是volatile变量读，step 4是普通变量读，符合第1个规则，同样不能重排序。

但如果是下列情况：第一个操作是普通变量读，第二个操作是volatile变量读，那是可以重排序的：

```
// 声明变量
int a = 0; // 声明普通变量
volatile boolean flag = false; // 声明volatile变量

// 以下两个变量的读操作是可以重排序的
int i = a; // 普通变量读
boolean j = flag; // volatile变量读
```

8.3 volatile的用途

从volatile的内存语义上来看，volatile可以保证内存可见性且禁止重排序。

在保证内存可见性这一点上，volatile有着与锁相同的内存语义，所以可以作为一个“轻量级”的锁来使用。但由于volatile仅仅保证对单个volatile变量的读/写具有原子性，而锁可以保证整个**临界区代码**的执行具有原子性。所以**在功能上，锁比**

volatile更强大；在性能上，volatile更有优势。

在禁止重排序这一点上，volatile也是非常有用的。比如我们熟悉的单例模式，其中有一种实现方式是“双重锁检查”，比如这样的代码：

```
public class Singleton {

    private static Singleton instance; // 不使用volatile关键字

    // 双重锁检验
    public static Singleton getInstance() {
        if (instance == null) { // 第7行
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton(); // 第10行
                }
            }
        }
        return instance;
    }
}
```

如果这里的变量声明不使用volatile关键字，是可能会发生错误的。它可能会被重排序：

```
instance = new Singleton(); // 第10行

// 可以分解为以下三个步骤
1 memory=allocate();// 分配内存 相当于c的malloc
2 ctorInstanc(memory) //初始化对象
3 s=memory //设置s指向刚分配的地址

// 上述三个步骤可能会被重排序为 1-3-2，也就是：
1 memory=allocate();// 分配内存 相当于c的malloc
3 s=memory //设置s指向刚分配的地址
2 ctorInstanc(memory) //初始化对象
```

而一旦假设发生了这样的重排序，比如线程A在第10行执行了步骤1和步骤3，但是步骤2还没有执行完。这个时候另一个线程B执行到了第7行，它会判定instance不为空，然后直接返回了一个未初始化完成的instance！

所以JSR-133对volatile做了增强后，volatile的禁止重排序功能还是非常有用的。

参考资料

- happens-before规则和as-if-serial语义
- volatile关键字详解
- Java可见性机制的原理
- Volatile关键字介绍
- Java中Volatile关键字详解
- JVM(十一)Java指令重排序
- 深入理解JVM（二）——内存模型、可见性、指令重排序
- JMM-volatile与内存屏障
- 并发关键字volatile（重排序和内存屏障）

第九章 synchronized与锁

这篇文章我们来聊一聊Java多线程里面的“锁”。

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。

Java类只有一个Class对象（可以有多个实例对象，多个实例共享这个Class对象），而Class对象也是特殊的Java对象。所以我们常说的类锁，其实就是Class对象的锁。

9.1 Synchronized关键字

说到锁，我们通常会谈到 `synchronized` 这个关键字。它翻译成中文就是“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。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
    }
}
```

我们这里介绍一下“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。而如果是使用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
    }
}
```

9.2 几种锁

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：

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，无锁在这里不再细讲。

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会比较苛刻，锁降级发生在Stop The World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。

> 关于锁降级有两点说明：
>
> 1.不同于大部分文章说锁不能降级，实际上HotSpot JVM 是支持锁降级的，文末有链接。
>
> 2.上面提到的Stop The World期间，以及安全点，这些知识是属于JVM的知识范畴，本文不做细讲。

下面分别介绍这几种锁以及它们之间的升级。

9.2.1 Java对象头

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”的信息是存放在什么地方的。

每个Java对象都有对象头。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：

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对象的指针。

9.2.2 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操作都不做了，提高了程序的运行性能。**

> 大白话就是对锁置个变量，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。

实现原理

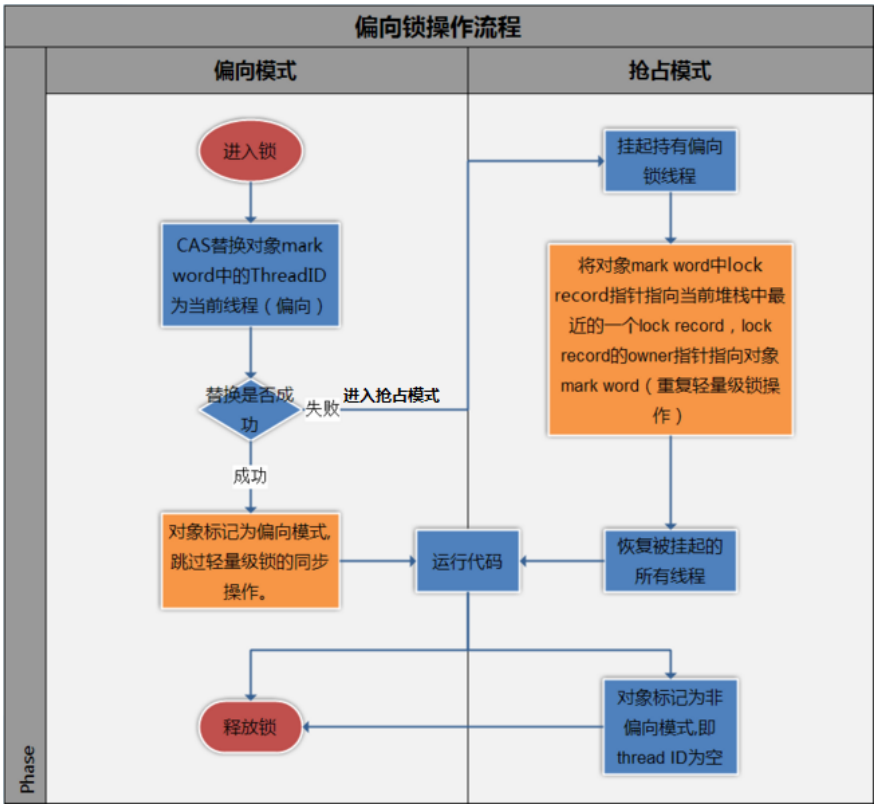
一个线程在第一次进入同步块时，会在对象头和栈帧中的锁记录里存储锁的偏向的线程ID。当下次该线程进入这个同步块时，会去检查锁的Mark Word里面是不是放的自己的线程ID。

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
>
> 比较并设置。用于在硬件层面上提供原子性操作。在 Intel 处理器中，比较并交换通过指令cmpxchg实现。 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。

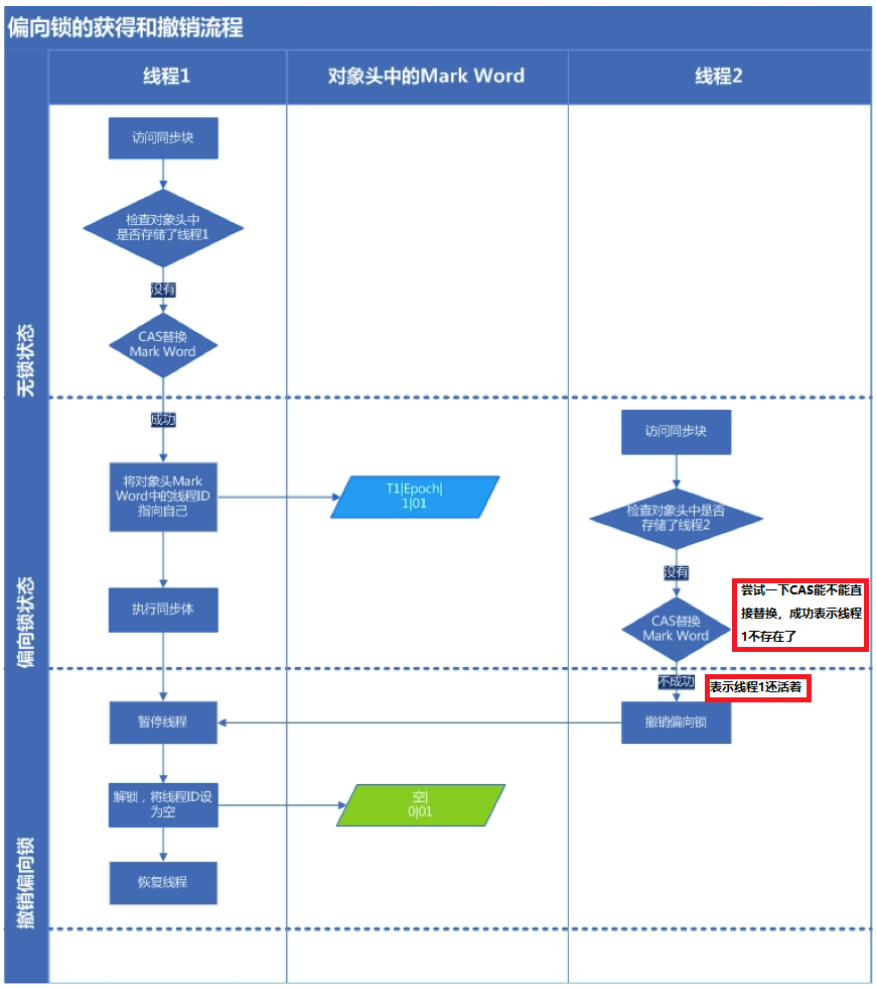
偏向锁升级成轻量级锁时，会暂停拥有偏向锁的线程，重置偏向锁标识，这个过程看起来容易，实则开销还是很大的，大概的过程如下：

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：



9.2.3 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。

轻量级锁的加锁

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。

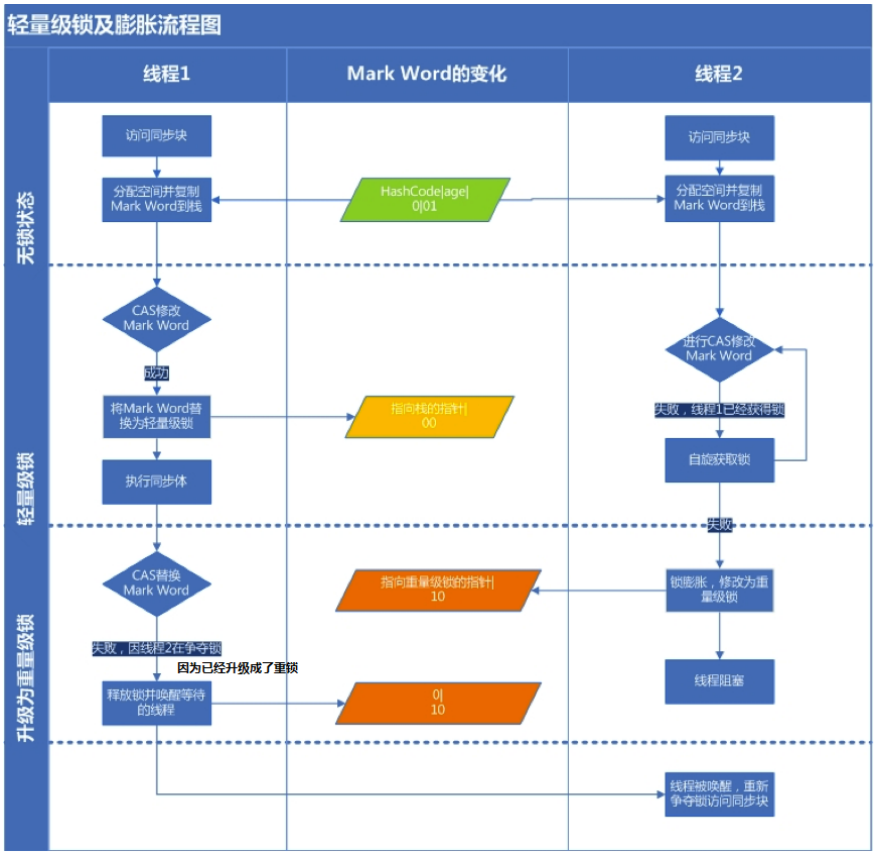
但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

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：



9.2.4 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比较长的时间，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的队列的队首，然后调用 `park` 函数挂起当前线程。

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。**

9.2.5 总结锁的升级流程

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。

9.2.6 各种锁的优缺点对比

下表来自《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。

参考文章

- Java锁优化--JVM锁降级
- Java中的锁机制
- 死磕Synchronized底层实现
- 《Java并发编程的艺术》

第十章 CAS与原子操作

10.1 乐观锁与悲观锁的概念

锁可以从不同的角度分类。其中，乐观锁和悲观锁是一种分类方式。

悲观锁：

悲观锁就是我们常说的锁。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：

乐观锁又称为“无锁”，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。而一旦多个线程发生冲突，乐观锁通常是使用一种称为CAS的技术来保证线程执行的安全性。

由于无锁操作中没有锁的存在，因此不可能出现死锁的情况，也就是说**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；而悲观锁多用于”写多读少“的环境，避免频繁失败和重试影响性能。

10.2 CAS的概念

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，则当前线程放弃更新，什么都不做。

所以这里的**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。

10.3 Java实现CAS的原理 - Unsafe类

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或者c++去实现。

在Java中，有一个 `Unsafe` 类，它在 `sun.misc` 包中。它里面是一些 `native` 方法，其中就有几个关于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);
```

当然，他们都是 `public native` 的。

Unsafe中对CAS的实现是C++写的，它的具体实现和操作系统、CPU都有关系。

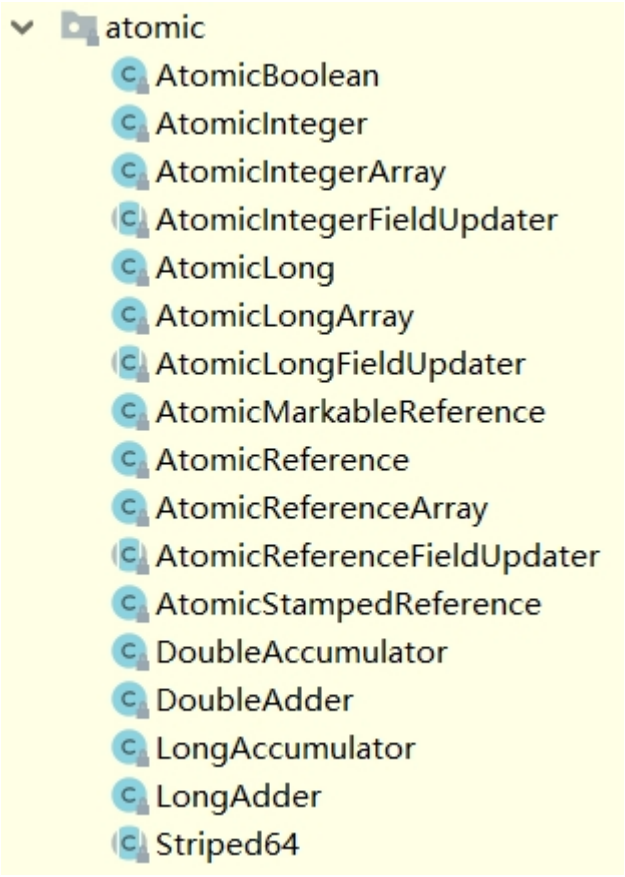
Linux的X86下主要是通过 `cmpxchgl` 这个指令在CPU级完成CAS操作的，但在多处理器情况下必须使用 `lock` 指令加锁来完成。当然不同的操作系统和处理器的实现会有所不同，大家可以自行了解。

当然，Unsafe类里面还有其它方法用于不同的用途。比如支持线程挂起和恢复的 `park` 和 `unpark` ， LockSupport类底层就是调用了这两个方法。还有支持反射操作的 `allocateInstance()` 方法。

10.4 原子操作-AtomicInteger类源码简析

上面介绍了Unsafe类的几个支持CAS的方法。那Java具体是如何使用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 11中，有如下17个类：



从名字就可以看得出来这些类大概的用途：

- 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。

先看看这个方法的源码：

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

这里的U其实就是一个 `Unsafe` 对象：

```
private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe()
```

所以其实 `AtomicInteger` 类的 `getAndAdd(int delta)` 方法是调用 `Unsafe` 类的方法来实现的：

```
@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;
}
```

> 注：这个方法是在JDK 1.8才新增的。在JDK1.8之前，`AtomicInteger`源码实现有所不同，是基于for死循环的，有兴趣的读者可以自行了解一下。

我们来一步步解析这段源码。首先，对象 `o` 是 `this`，也就是一个 `AtomicInteger` 对象。然后 `offset` 是一个常量 `VALUE`。这个常量是在 `AtomicInteger` 类中声明的：

```
private static final long VALUE = U.objectFieldOffset(AtomicInteger.class, "value");
```

同样是调用的 `Unsafe` 的方法。从方法名字上来看，是得到了一个对象字段偏移量。

> 用于获取某个字段相对Java对象的“起始地址”的偏移量。
>
> 一个java对象可以看成是一段内存，各个字段都得按照一定的顺序放在这段内存里，同时考虑到对齐要求，可能这些字段不是连续放置的，
>
> 用这个方法能准确地告诉你某个字段相对于对象的起始内存地址的字节偏移量，因为是相对偏移量，所以它其实跟某个具体对象又没什么太大关系，跟class的定义和虚拟机的内存模型的实现细节更相关。

继续看源码。前面我们讲到，CAS是“无锁”的基础，它允许更新失败。所以经常会与while循环搭配，在失败后不断去重试。

这里声明了一个v，也就是要返回的值。从 `getAndAddInt` 来看，它返回的应该是原来的值，而新的值的 `v + delta`。

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 `v` 是我们期望的值。

循环体的条件是一个CAS方法：

```
public final boolean weakCompareAndSetInt(Object o, long offset,
                                          int expected,
                                          int x) {
    return compareAndSetInt(o, offset, expected, x);
}

public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);
```

可以看到，最终其实是调用的我们之前说到了CAS `native` 方法。那为什么要经过一层 `weakCompareAndSetInt` 呢？从JDK源码上看不出来什么。在JDK 8及之前的版本，这两个方法是一样的。

> 而在JDK 9开始，这两个方法上面增加了@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性能。也就是说虽然外面看到的在JDK9中weakCompareAndSet和compareAndSet底层依旧是调用了一样的代码，但是不排除HotSpot VM会手动来实现weakCompareAndSet真正含义的功能的可能性。

根据本文第一篇参考文章（文末链接），它跟 `volatile` 有关。

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。也就是说， `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。**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。那为什么要把获取“旧值”v的操作放到循环体内呢？其实这也很好理解。前面我们说了，CAS如果旧值V不等于预期值E，它就会更新失败。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。

10.5 CAS实现原子操作的三大问题

这里介绍一下CAS实现原子操作的三大问题及其解决方案。

10.5.1 ABA问题

所谓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)));
}
```

10.5.2 循环时间长开销大

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。

10.5.3 只能保证一个共享变量的原子操作

这个问题你可能已经知道怎么解决了。有两种解决方案：

1. 使用JDK 1.5开始就提供的 `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。

参考资料

1. 对 volatile、compareAndSet、weakCompareAndSet 的一些思考
2. 《Java 并发编程的艺术》

第十一章 AQS

11.1 AQS简介

AQS是 `AbstractQueuedSynchronizer` 的简称，即 `抽象队列同步器` ，从字面意思上理解:

- 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）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，比如我们提到的ReentrantLock，Semaphore，ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。

当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器，只要子类实现它的几个 `protected` 方法就可以了，在下文会有详细的介绍。

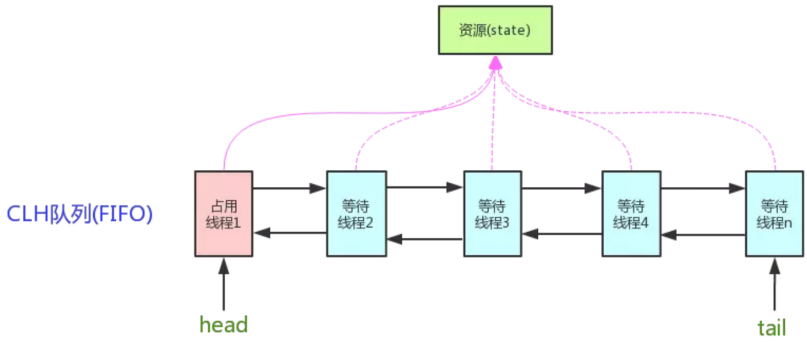
11.2 AQS的数据结构

AQS内部使用了一个volatile的变量state来作为资源的标识。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。

而AQS类本身实现的是一些排队和阻塞的机制，比如具体线程等待队列的维护（如获取资源失败入队/唤醒出队等）。它内部使用了一个先进先出（FIFO）的双端队列，并使用了两个指针head和tail用于标识队列的头部和尾部。其数据结构如图：



但它并不是直接储存线程，而是储存拥有线程的Node节点。

11.3 资源共享模式

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，一次只能一个线程获取。如ReentrantLock。
- 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。

一般情况下，子类只需要根据需求实现其中一种模式，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
    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
}
```

> 注意：通过Node我们可以实现两个队列，一是通过prev和next实现CLH队列(线程同步队列,双向队列)，二是nextWaiter实现Condition条件上的等待线程队列(单向队列)，这个Condition主要用在ReentrantLock类中。

11.4 AQS的主要方法源码解析

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` 方法，但是它们并没有在AQS具体实现，而是直接抛出异常（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）：

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

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：

11.4.1 获取资源

获取资源的入口是acquire(int arg)方法。arg是要获取的资源的个数，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) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

首先调用tryAcquire(arg)尝试去获取资源。前面提到了这个方法是在子类具体实现的。

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) {
    // 生成该线程对应的Node节点
    Node node = new Node(Thread.currentThread(), mode);
    // 将Node插入队列中
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        // 使用CAS尝试，如果成功就返回
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    // 如果等待队列为空或者上述CAS失败，再自旋CAS插入
    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;
            }
        }
    }
}
```

> 上面的两个函数比较好理解，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(int arg)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 {
        boolean interrupted = false;
        // 自旋
        for (;;) {
            final Node p = node.predecessor();
            // 如果node的前驱结点p是head，表示node是第二个结点，就可以尝试去获取资源了
            if (p == head && tryAcquire(arg)) {
                // 拿到资源后，将head指向该结点。
                // 所以head所指的结点，就是当前获取到资源的那个结点或null。
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            // 如果自己可以休息了，就进入waiting状态，直到被unpark()
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        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。
>
> 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类里的函数，归结到Unsafe里，只有两个函数：
>
> - park(boolean isAbsolute, long time)：阻塞当前线程
> - unpark(Thread jthread)：使给定的线程停止阻塞

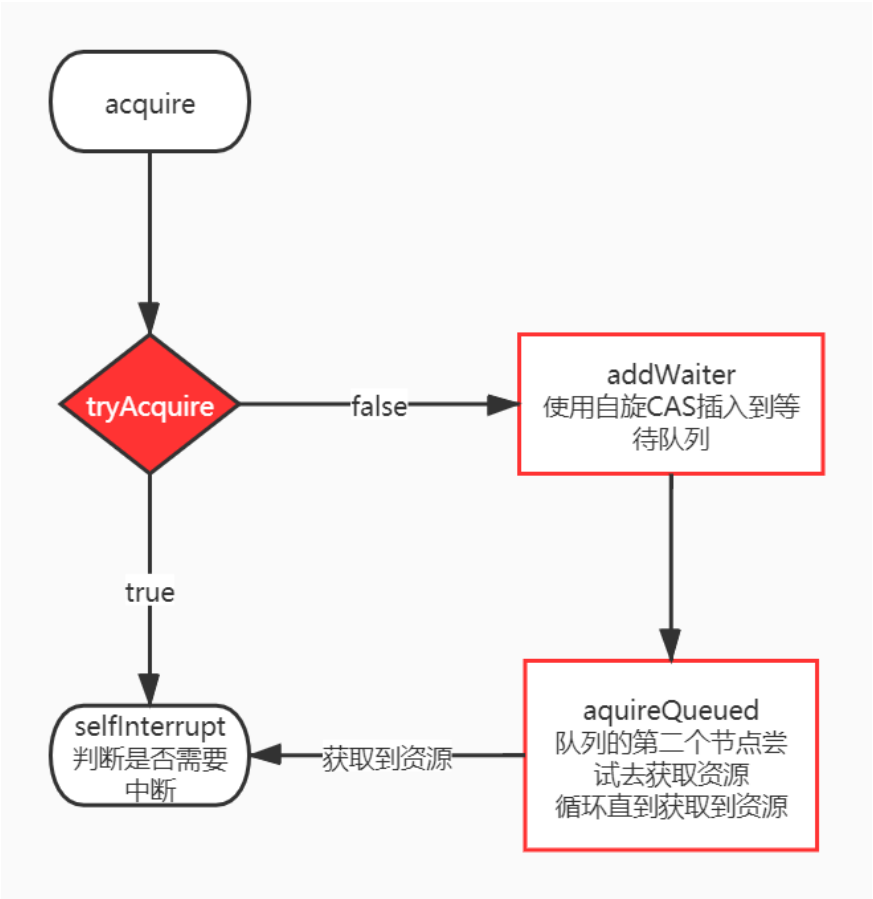
所以**结点进入等待队列后，是调用park使它进入阻塞状态的。只有头结点的线程是处于活跃状态的。**

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：



11.4.2 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
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 等待队列中所有还有用的结点，都向前移动
        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);
}
```

参考资料

- Java技术之AQS详解
- JDK源码之AQS源码剖析
- JUC解析-AQS(1)

第十二章 线程池原理

12.1 为什么要使用线程池

使用线程池主要有以下三个原因：

1. 创建/销毁线程需要消耗系统资源，线程池可以**复用已创建的线程。**
2. **控制并发的数量。**并发数量过多，可能会导致资源消耗过多，从而造成服务器崩溃。（主要原因）
3. **可以对线程做统一管理。**

12.2 线程池的原理

Java中的线程池顶层接口是 `Executor` 接口， `ThreadPoolExecutor` 是这个接口的实现类。

我们先看看 `ThreadPoolExecutor` 类。

12.2.1 ThreadPoolExecutor提供的构造方法

一共有四个构造方法：

```
// 五个参数的构造函数
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)

// 六个参数的构造函数-1
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory)

// 六个参数的构造函数-2
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)

// 七个参数的构造函数
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

涉及到5~7个参数，我们先看看必须的5个参数是什么意思：

- **int corePoolSize**：该线程池中**核心线程数最大值**

  > 核心线程：线程池中有两类线程，核心线程和非核心线程。核心线程默认情况下会一直存在于线程池中，即使这个核心线程什么都不干（铁饭碗），而非核心线程如果长时间的闲置，就会被销毁（临时工）。

- **int maximumPoolSize**：该线程池中**线程总数最大值** 。

  > 该值等于核心线程数量 + 非核心线程数量。

- **long keepAliveTime**：**非核心线程闲置超时时长。**

  > 非核心线程如果处于闲置状态超过该值，就会被销毁。如果设置allowCoreThreadTimeOut(true)，则会也作用于核心线程。

- **TimeUnit unit**：keepAliveTime的单位。

TimeUnit是一个枚举类型 ，包括以下属性：

> NANOSECONDS ： 1微毫秒 = 1微秒 / 1000 MICROSECONDS ： 1微秒 = 1毫秒 / 1000 MILLISECONDS ： 1毫秒 = 1秒 /1000 SECONDS ： 秒 MINUTES ： 分 HOURS ： 小时 DAYS ： 天

- **BlockingQueue workQueue**：阻塞队列，维护着**等待执行的Runnable任务对象。**

  常用的几个阻塞队列：

1. **LinkedBlockingQueue**

   链式阻塞队列，底层数据结构是链表，默认大小是`Integer.MAX_VALUE`，也可以指定大小。

2. **ArrayBlockingQueue**

   数组阻塞队列，底层数据结构是数组，需要指定队列的大小。

3. **SynchronousQueue**

   同步队列，内部容量为0，每个put操作必须等待一个take操作，反之亦然。

4. **DelayQueue**

   延迟队列，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 。

> 我们将在下一章中重点介绍各种阻塞队列

好了，介绍完5个必须的参数之后，还有两个非必须的参数。

- **ThreadFactory threadFactory**

  创建线程的工厂 ，用于批量创建线程，统一在创建线程时设置一些参数，如是否守护线程、线程的优先级等。如果不指定，会新建一个默认的线程工厂。

```
static class DefaultThreadFactory implements ThreadFactory {
    // 省略属性
    // 构造函数
    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() :
        Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" +
            poolNumber.getAndIncrement() +
            "-thread-";
    }

    // 省略
}
```

- **RejectedExecutionHandler handler**

  拒绝处理策略，线程数量大于最大线程数就会采用拒绝处理策略，四种拒绝处理的策略为：

  1. **ThreadPoolExecutor.AbortPolicy**：**默认拒绝处理策略**，丢弃任务并抛出RejectedExecutionException异常。
  2. **ThreadPoolExecutor.DiscardPolicy**：丢弃新来的任务，但是不抛出异常。
  3. **ThreadPoolExecutor.DiscardOldestPolicy**：丢弃队列头部（最旧的）的任务，然后重新尝试执行程序（如果再次失败，重复此过程）。
  4. **ThreadPoolExecutor.CallerRunsPolicy**：由调用线程处理该任务。

12.2.2 ThreadPoolExecutor的策略

线程池本身有一个调度线程，这个线程就是用于管理布控整个线程池里的各种任务和事务，例如创建线程、销毁线程、任务队列管理、线程队列管理等等。

故线程池也有自己的状态。`ThreadPoolExecutor` 类中使用了一些 `final int` 常量变量来表示线程池的状态，分别为RUNNING、SHUTDOWN、STOP、TIDYING 、TERMINATED。

```
// runState is stored in the high-order bits
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;
```

- 线程池创建后处于**RUNNING**状态。
- 调用shutdown()方法后处于**SHUTDOWN**状态，线程池不能接受新的任务，清除一些空闲worker,不会等待阻塞队列的任务完成。
- 调用shutdownNow()方法后处于**STOP**状态，线程池不能接受新的任务，中断所有线程，阻塞队列中没有被执行的任务全部丢弃。此时，poolsize=0,阻塞队列的size也为0。
- 当所有的任务已终止，ctl记录的"任务数量"为0，线程池会变为**TIDYING**状态。接着会执行terminated()函数。

  > ThreadPoolExecutor中有一个控制状态的属性叫 `ctl`，它是一个AtomicInteger类型的变量。线程池状态就是通过AtomicInteger类型的成员变量 `ctl` 来获取的。
  >
  > 获取的 `ctl` 值传入 `runStateOf` 方法，与 `~CAPACITY` 位与运算(`CAPACITY` 是低29位全1的int变量)。
  >
  > `~CAPACITY` 在这里相当于掩码，用来获取ctl的高3位，表示线程池状态；而另外的低29位用于表示工作线程数

- 线程池处在TIDYING状态时，**执行完terminated()方法之后**，就会由 **TIDYING -> TERMINATED**， 线程池被设置为TERMINATED状态。

12.2.3 线程池主要的任务处理流程

处理任务的核心方法是 `execute` ，我们看看 JDK 1.8 源码中 `ThreadPoolExecutor` 是如何处理线程任务的：

```
// JDK 1.8
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    // 1.当前线程数小于corePoolSize,则调用addWorker创建核心线程执行任务
    if (workerCountOf(c) < corePoolSize) {
       if (addWorker(command, true))
           return;
       c = ctl.get();
    }
    // 2.如果不小于corePoolSize，则将任务添加到workQueue队列。
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // 2.1 如果isRunning返回false(状态检查)，则remove这个任务，然后执行拒绝策略。
        if (! isRunning(recheck) && remove(command))
            reject(command);
            // 2.2 线程池处于running状态，但是没有线程，则创建线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 3.如果放入workQueue失败，则创建非核心线程执行任务，
    // 如果这时创建非核心线程失败(当前线程总数不小于maximumPoolSize时)，就会执行拒绝策略。
    else if (!addWorker(command, false))
         reject(command);
}
```

`ctl.get()` 是获取线程池状态，用 `int` 类型表示。第二步中，入队前进行了一次 `isRunning` 判断，入队之后，又进行了一次 `isRunning` 判断。

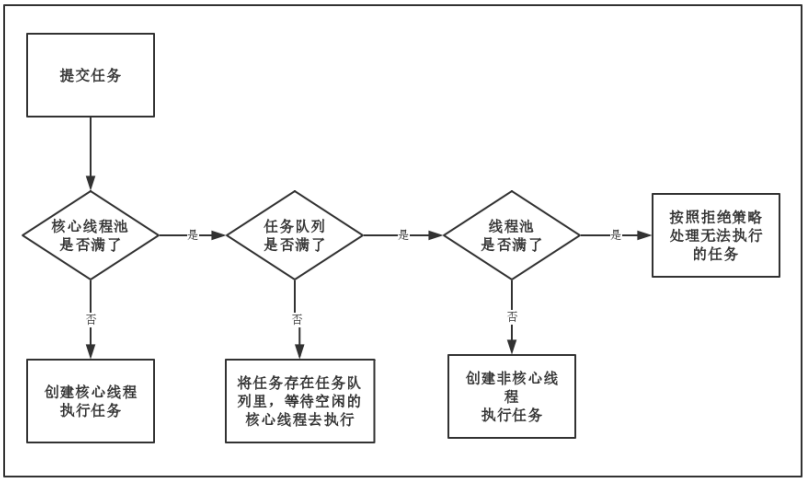
为什么要二次检查线程池的状态?

在多线程的环境下，线程池的状态是时刻发生变化的。很有可能刚获取线程池状态后线程池状态就改变了。判断是否将 `command` 加入 `workqueue` 是线程池之前的状态。倘若没有二次检查，万一线程池处于非**RUNNING**状态（在多线程环境下很有可能发生），那么 `command` 永远不会执行。

总结一下处理流程

1. 线程总数量 < corePoolSize，无论线程是否空闲，都会新建一个核心线程执行任务（让核心线程数量快速达到corePoolSize，在核心线程数量 < corePoolSize时）。**注意，这一步需要获得全局锁。**
2. 线程总数量 >= corePoolSize时，新来的线程任务会进入任务队列中等待，然后空闲的核心线程会依次去缓存队列中取任务来执行（体现了**线程复用**）。
3. 当缓存队列满了，说明这个时候任务已经多到爆棚，需要一些“临时工”来执行这些任务了。于是会创建非核心线程去执行这个任务。**注意，这一步需要获得全局锁。**
4. 缓存队列满了， 且总线程数达到了maximumPoolSize，则会采取上面提到的拒绝策略进行处理。

整个过程如图所示：



12.2.4 ThreadPoolExecutor如何做到线程复用的?

我们知道，一个线程在创建的时候会指定一个线程任务，当执行完这个线程任务之后，线程自动销毁。但是线程池却可以复用线程，即一个线程执行完线程任务后不销毁，继续执行另外的线程任务。**那么，线程池如何做到线程复用呢?**

原来，ThreadPoolExecutor在创建线程时，会将线程封装成**工作线程worker**,并放入**工作线程组**中，然后这个worker反复从阻塞队列中拿任务去执行。话不多说，我们继续看看源码（一定要仔细看，前后有联系）

这里的 `addWorker` 方法是在上面提到的 `execute` 方法里面调用的，先看看上半部分：

```
// ThreadPoolExecutor.addWorker方法源码上半部分
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                // 1.如果core是ture,证明需要创建的线程为核心线程，则先判断当前线程是否大于核
                // 如果core是false,证明需要创建的是非核心线程，则先判断当前线程数是否大于总
                // 如果不小于，则返回false
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
```

上半部分主要是判断线程数量是否超出阈值，超过了就返回false。我们继续看下半部分:

```
    // ThreadPoolExecutor.addWorker方法源码下半部分
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        // 1.创建一个worker对象
        w = new Worker(firstTask);
        // 2.实例化一个Thread对象
        final Thread t = w.thread;
        if (t != null) {
            // 3.线程池全局锁
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                // 4.启动这个线程
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

创建 `worker` 对象，并初始化一个 `Thread` 对象，然后启动这个线程对象。

我们接着看看 `Worker` 类，仅展示部分源码：

```
// Worker类部分源码
private final class Worker extends AbstractQueuedSynchronizer implements Runnable{
    final Thread thread;
    Runnable firstTask;

    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }

    public void run() {
            runWorker(this);
    }
    //其余代码略...
}
```

`Worker` 类实现了 `Runnable` 接口，所以 `Worker` 也是一个线程任务。在构造方法中，创建了一个线程，线程的任务就是自己。故 `addWorker` 方法调用addWorker方法源码下半部分中的第4步 `t.start` ，会触发 `Worker` 类的 `run` 方法被JVM调用。

我们再看看 `runWorker` 的逻辑：

```
// Worker.runWorker方法源代码
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 1.线程启动之后，通过unlock方法释放锁
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        // 2.Worker执行firstTask或从workQueue中获取任务，如果getTask方法不返回null,循环不
        while (task != null || (task = getTask()) != null) {
            // 2.1进行加锁操作，保证thread不被其他线程中断（除非线程池被中断）
            w.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            // 2.2检查线程池状态，倘若线程池处于中断状态，当前线程将中断。
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                // 2.3执行beforeExecute
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    // 2.4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 {
                    // 2.5执行afterExecute方法
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                // 2.6解锁操作
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

首先去执行创建这个worker时就有的任务，当执行完这个任务后，worker的生命周期并没有结束，在 `while` 循环中，worker会不断地调用 `getTask` 方法从**阻塞队列**中获取任务然后调用 `task.run()` 执行任务,从而达到**复用线程**的目的。只要 `getTask` 方法不返回 `null` ,此线程就不会退出。

当然，核心线程池中创建的线程想要拿到阻塞队列中的任务，先要判断线程池的状态，如果**STOP**或者**TERMINATED**，返回 `null` 。

最后看看 `getTask` 方法的实现:

```
// Worker.getTask方法源码
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        // 1.allowCoreThreadTimeOut变量默认是false,核心线程即使空闲也不会被销毁
        // 如果为true,核心线程在keepAliveTime内仍空闲则会被销毁。
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
        // 2.如果运行线程数超过了最大线程数，但是缓存队列已经空了，这时递减worker数量。
        // 如果有设置允许线程超时或者线程数量超过了核心线程数量，
        // 并且线程在规定时间内均未poll到任务且队列为空则递减worker数量
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            // 3.如果timed为true(想想哪些情况下timed为true),则会调用workQueue的poll方法获
            // 超时时间是keepAliveTime。如果超过keepAliveTime时长，
            // poll返回了null，上边提到的while循序就会退出，线程也就执行完了。
            // 如果timed为false（allowCoreThreadTimeOut为falsefalse
            // 且wc > corePoolSize为false），则会调用workQueue的take方法阻塞在当前。
            // 队列中有任务加入时，线程被唤醒，take方法返回任务，并执行。
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

核心线程的会一直卡在 `workQueue.take` 方法，被阻塞并挂起，不会占用CPU资源，直到拿到 `Runnable` 然后返回（当然如果**allowCoreThreadTimeOut**设置为 `true` ,那么核心线程就会去调用 `poll` 方法，因为 `poll` 可能会返回 `null` ,所以这时候核心线程满足超时条件也会被销毁）。

非核心线程会workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)，如果超时还没有拿到，下一次循环判断**compareAndDecrementWorkerCount**就会返回 `null` ,Worker对象的 `run()` 方法循环体的判断为 `null` ,任务结束，然后线程被系统回收 。

源码解析完毕，你理解的源码是否和图中的处理流程一致？如果不一致，那么就多看两遍吧，加油。

12.3 四种常见的线程池

`Executors` 类中提供的几个静态方法来创建线程池。大家到了这一步，如果看懂了前面讲的 `ThreadPoolExecutor` 构造方法中各种参数的意义，那么一看到 `Executors` 类中提供的线程池的源码就应该知道这个线程池是干嘛的。

12.3.1 newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

`CacheThreadPool` 的**运行流程**如下：

1. 提交任务进线程池。
2. 因为**corePoolSize**为0的关系，不创建核心线程，线程池最大为Integer.MAX_VALUE。
3. 尝试将任务添加到**SynchronousQueue**队列。
4. 如果SynchronousQueue入列成功，等待被当前运行的线程空闲后拉取执行。如果当前没有空闲线程，那么就创建一个非核心线程，然后从SynchronousQueue拉取任务并在当前线程执行。
5. 如果SynchronousQueue已有任务在等待，入列操作将会阻塞。

当需要执行很多**短时间**的任务时，CacheThreadPool的线程复用率比较高， 会显著的**提高性能**。而且线程60s后会回收，意味着即使没有任务进来，CacheThreadPool并不会占用很多资源。

12.3.2 newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
}
```

核心线程数量和总线程数量相等，都是传入的参数nThreads，所以只能创建核心线程，不能创建非核心线程。因为LinkedBlockingQueue的默认大小是Integer.MAX_VALUE，故如果核心线程空闲，则交给核心线程处理；如果核心线程不空闲，则入列等待，直到核心线程空闲。

与CachedThreadPool的区别：

- 因为 corePoolSize == maximumPoolSize ，所以FixedThreadPool只会创建核心线程。 而CachedThreadPool因为corePoolSize=0，所以只会创建非核心线程。

- 在 getTask() 方法，如果队列里没有任务可取，线程会一直阻塞在 LinkedBlockingQueue.take()，线程不会被回收。 CachedThreadPool会在60s后收回。
- 由于线程不会被回收，会一直卡在阻塞，所以**没有任务的情况下，FixedThreadPool占用资源更多。**
- 都几乎不会触发拒绝策略，但是原理不同。FixedThreadPool是因为阻塞队列可以很大（最大为Integer最大值），故几乎不会触发拒绝策略；CachedThreadPool是因为线程池很大（最大为Integer最大值），几乎不会导致线程数量大于最大线程数，故几乎不会触发拒绝策略。

12.3.3 newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

有且仅有一个核心线程（ corePoolSize == maximumPoolSize=1），使用了LinkedBlockingQueue（容量很大），所以，**不会创建非核心线程。**所有任务按照**先来先执行**的顺序执行。如果这个唯一的线程不空闲，那么新来的任务会存储在任务队列里等待执行。

12.3.4 newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

//ScheduledThreadPoolExecutor():
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE,
          DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS,
          new DelayedWorkQueue());
}
```

四种常见的线程池基本够我们使用了，但是《阿里巴巴开发手册》不建议我们直接使用Executors类中的线程池，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学需要更加明确线程池的运行规则，规避资源耗尽的风险。

但如果你及团队本身对线程池非常熟悉，又确定业务规模不会大到资源耗尽的程度（比如线程数量或任务队列长度可能达到Integer.MAX_VALUE）时，其实是可以使用JDK提供的这几个接口的，它能让我们的代码具有更强的可读性。

参考资料

1. 线程池，这一篇或许就够了
2. 线程池的使用
3. 线程池原理详解一

4. Java线程池复用的秘密
5. java线程池实现原理与源码分析（jdk1.8）
6. 《并发编程的艺术》

第十三章 阻塞队列

13.1 阻塞队列的由来

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，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。（这块不明白的同学，可以看最下方结语部分的链接）

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。**

13.2 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：如果试图的操作无法立即执行，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：如果试图的操作无法立即执行，返回一个特殊值，通常是true / false。
- 一直阻塞：如果试图的操作无法立即执行，则一直阻塞或者响应中断。
- 超时退出：如果试图的操作无法立即执行，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。

注意之处

- 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。

13.3 BlockingQueue的实现类

13.3.1 ArrayBlockingQueue

由**数组**结构组成的**有界**阻塞队列。内部结构是数组，故具有数组的特性。

```
public ArrayBlockingQueue(int capacity, boolean fair){
    //..省略代码
}
```

可以初始化队列大小， 且一旦初始化不能改变。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**。

13.3.2 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。

13.3.3 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

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
```

13.3.4 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

> 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
    }
```

13.3.5 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。

13.5 阻塞队列的原理

阻塞队列的原理很简单，利用了Lock锁的多条件（Condition）阻塞控制。接下来我们分析ArrayBlockingQueue JDK 1.8 的源码。

首先是构造器，除了初始化队列的大小和是否是公平锁之外，还对同一个锁（lock）初始化了两个监视器，分别是notEmpty和notFull。这两个监视器的作用目前可以简单理解为标记分组，当该线程是put操作时，给他加上监视器notFull,标记

这个线程是一个生产者；当线程是take操作时，给他加上监视器notEmpty，标记这个线程是消费者。

```
//数据元素数组
final Object[] items;
//下一个待取出元素索引
int takeIndex;
//下一个待添加元素索引
int putIndex;
//元素个数
int count;
//内部锁
final ReentrantLock lock;
//消费者监视器
private final Condition notEmpty;
//生产者监视器
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    //..省略其他代码
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

put操作的源码

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    // 1.自旋拿锁
    lock.lockInterruptibly();
    try {
        // 2.判断队列是否满了
        while (count == items.length)
            // 2.1如果满了，阻塞该线程，并标记为notFull线程，
            // 等待notFull的唤醒，唤醒之后继续执行while循环。
            notFull.await();
        // 3.如果没有满，则进入队列
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
private void enqueue(E x) {
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    // 4 唤醒一个等待的线程
    notEmpty.signal();
}
```

总结put的流程：

1. 所有执行put操作的线程竞争lock锁，拿到了lock锁的线程进入下一步，没有拿到lock锁的线程自旋竞争锁。

2. 判断阻塞队列是否满了，如果满了，则调用await方法阻塞这个线程，并标记为notFull（生产者）线程，同时释放lock锁,等待被消费者线程唤醒。
3. 如果没有满，则调用enqueue方法将元素put进阻塞队列。注意这一步的线程还有一种情况是第二步中阻塞的线程被唤醒且又拿到了lock锁的线程。
4. 唤醒一个标记为notEmpty（消费者）的线程。

take操作的源码

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0;
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    notFull.signal();
    return x;
}
```

take操作和put操作的流程是类似的，总结一下take操作的流程：

1. 所有执行take操作的线程竞争lock锁，拿到了lock锁的线程进入下一步，没有拿到lock锁的线程自旋竞争锁。
2. 判断阻塞队列是否为空，如果是空，则调用await方法阻塞这个线程，并标记为notEmpty（消费者）线程，同时释放lock锁,等待被生产者线程唤醒。
3. 如果没有空，则调用dequeue方法。注意这一步的线程还有一种情况是第二步中阻塞的线程被唤醒且又拿到了lock锁的线程。
4. 唤醒一个标记为notFull（生产者）的线程。

注意

1. put和take操作都需要**先获取锁**，没有获取到锁的线程会被挡在第一道大门之外自旋拿锁，直到获取到锁。
2. 就算拿到锁了之后，也**不一定**会顺利进行put/take操作，需要判断**队列是否可用**（是否满/空），如果不可用，则会被阻塞，**并释放锁。**
3. 在第2点被阻塞的线程会被唤醒，但是在唤醒之后，**依然需要拿到锁**才能继续往下执行，否则，自旋拿锁，拿到锁了再while判断队列是否可用（这也是为什么不用if判断，而使用while判断的原因）。

13.6 示例和使用场景

13.6.1 生产者-消费者模型

```
public class Test {
    private int queueSize = 10;
    private ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(queueS

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                try {
                    queue.take();
                    System.out.println("从队列取走一个元素，队列剩余"+queue.size()+"个元
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                try {
                    queue.put(1);
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+(queueSiz
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

下面是这个例子的输出片段：

```
从队列取走一个元素，队列剩余0个元素
从队列取走一个元素，队列剩余0个元素
向队列取中插入一个元素，队列剩余空间：9
向队列取中插入一个元素，队列剩余空间：9
向队列取中插入一个元素，队列剩余空间：9
向队列取中插入一个元素，队列剩余空间：8
向队列取中插入一个元素，队列剩余空间：7
向队列取中插入一个元素，队列剩余空间：6
向队列取中插入一个元素，队列剩余空间：5
向队列取中插入一个元素，队列剩余空间：4
向队列取中插入一个元素，队列剩余空间：3
向队列取中插入一个元素，队列剩余空间：2
向队列取中插入一个元素，队列剩余空间：1
向队列取中插入一个元素，队列剩余空间：0
从队列取走一个元素，队列剩余1个元素
从队列取走一个元素，队列剩余9个元素
```

注意，这个例子中的输出结果看起来可能有问题，比如有几行在插入一个元素之后，队列的剩余空间不变。这是由于**System.out.println语句没有锁。**考虑到这样的情况：线程1在执行完put/take操作后立即失去CPU时间片，然后切换到线程2执行put/take操作，执行完毕后回到线程1的System.out.println语句并输出，发现这个时候阻塞队列的size已经被线程2改变了，所以这个时候输出的size并不是当时线程1执行完put/take操作之后阻塞队列的size，但可以确保的是size不会超过10个。实际上使用阻塞队列是没有问题的。

13.6.2 线程池中使用阻塞队列

```
 public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
}
```

Java中的线程池就是使用阻塞队列实现的，我们在了解阻塞队列之后，无论是使用Executors类中已经提供的线程池，还是自己通过ThreadPoolExecutor实现线程池，都会更加得心应手，想要了解线程池的同学，可以看第十二章：线程池原理。

> 注：上面提到了生产者-消费者模式，大家可以参考生产者-消费者模型，可以更好的理解阻塞队列。

参考资料

- Java中的阻塞队列
- Java并发编程：阻塞队列
- SynchronousQueue应用

第十四章 锁接口和类

前面我们介绍了Java原生的锁——基于对象的锁，它一般是配合synchronized关键字来使用的。实际上，Java在 `java.util.concurrent.locks` 包下，还为我们提供了几个关于锁的类和接口。它们有更强大的功能或更高的性能。

14.1 synchronized的不足之处

我们先来看看 `synchronized` 有什么不足之处。

- 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。**

而这些都是locks包下的锁可以解决的。

14.2 锁的几种分类

锁可以根据以下几种方式来进行分类，下面我们逐一介绍。

14.2.1 可重入锁和非可重入锁

所谓重入锁，顾名思义。就是支持重新进入的锁，也就是说这个锁支持一个**线程对资源重复加锁。**

synchronized关键字就是使用的重入锁。比如说，你在一个synchronized实例方法里面调用另一个本实例的synchronized实例方法，它可以重新进入这个锁，不会出现任何异常。

如果我们自己在继承AQS实现同步器的时候，没有考虑到占有锁的线程再次获取锁的场景，可能就会导致线程阻塞，那这个就是一个“非可重入锁”。

`ReentrantLock` 的中文意思就是可重入锁。也是本文后续要介绍的重点类。

14.2.2 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。

14.2.3 读写锁和排它锁

我们前面讲到的synchronized用的锁和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。

可见，只是synchronized是远远不能满足多样化的业务对锁的要求的。接下来我们介绍一下JDK中有关锁的一些接口和类。

14.3 JDK中有关锁的一些接口和类

众所周知，JDK中关于并发的类大多都在 `java.util.concurrent` （以下简称juc）包下。而juc.locks包看名字就知道，是提供了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。下面分别介绍一下这个包下的类和接口以及它们之间的关系。

14.3.1 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。

首先我们看**AQS**（AbstractQueuedSynchronizer），之前专门有章节介绍这个类，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。而AQS里面的“资源”是用一个 `int` 类型的数据来表示的，有时候我们的业务需求资源的数量超出了 `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;
}
```

14.3.2 接口Condition/Lock/ReadWriteLock

juc.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接口中有一个方法是可以获得一个`Condition`：

```
Condition newCondition();
```

之前我们提到了每个对象都可以用继承自`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监视器的方法，通过与**Lock**配合来实现等待/通知模式。

那为什么既然有Object的监视器方法了，还要用Condition呢？这里有一个二者简单的对比：

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

14.3.3 ReentrantLock

ReentrantLock是一个非抽象类，它是Lock接口的JDK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`，是继承了AQS，自己实现的一个同步器。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。

14.3.4 ReentrantReadWriteLock

这个类也是一个非抽象类，它是ReadWriteLock接口的JDK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。

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”，将在后文的StampedLock类继续讨论这个问题。

14.3.5 StampedLock

`StampedLock` 类是在Java 8 才发布的，也是Doug Lea大神所写，有人号称它为锁的性能之王。它没有实现Lock接口和ReadWriteLock接口，但它其实是实现了“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。

这里篇幅有限，就不介绍StampedLock的源码了，只是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）。

```
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的作用。

参考文档

1. Java并发编程：Lock
2. Java多线程之Lock的使用（一）
3. Java锁之ReentrantReadWriteLock
4. Java锁之ReentrantLock（一）
5. Java并发（8）- 读写锁中的性能之王：StampedLock
6. Java多线程Condition接口原理详解
7. Java中的Condition

第十五章 并发容器集合

15.1 同步容器与并发容器

我们知道在java.util包下提供了一些容器类，而Vector和Hashtable是线程安全的容器类，但是这些容器实现同步的方式是通过对方法加锁(sychronized)方式实现的，这样读写均需要锁操作，导致性能低下。

而即使是Vector这样线程安全的类，在面对多线程下的复合操作的时候也是需要通过客户端加锁的方式保证原子性。如下面例子说明:

```
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);
        }
    }

}
```

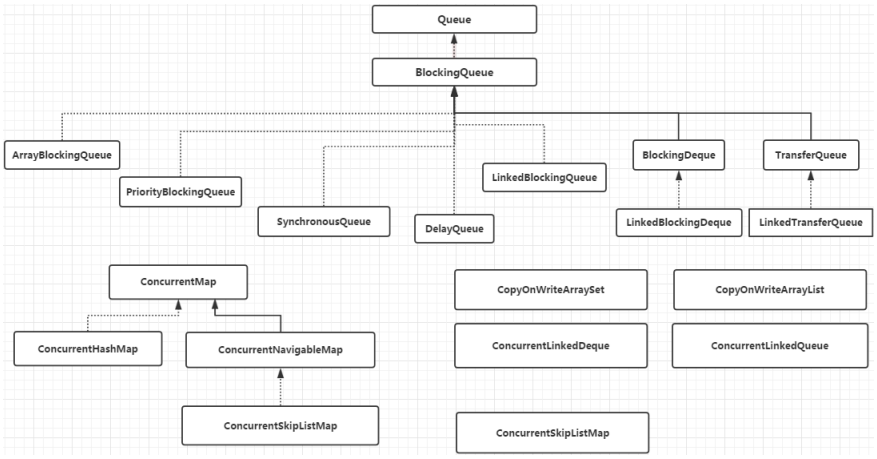
如果方法一和方法二为一个组合的话。那么当方法一获取到了 `vector` 的size之后，方法二已经执行完毕，这样就导致程序的错误。

如果方法三与方法四组合的话。通过锁机制保证了在 `vector` 上的操作的原子性。

并发容器是Java 5 提供的在多线程编程下用于代替同步容器，针对不同的应用场景进行设计，提高容器的并发访问性，同时定义了线程安全的复合操作。

15.2 并发容器类介绍

整体架构(列举常用的容器类)



其中，阻塞队列（BlockingQueue）在第十三章有介绍，CopyOnWrite容器（CopyOnWritexxx）在第十六章有介绍，这里不做过多介绍。

下面分别介绍一些常用的并发容器类和接口，因篇幅原因，这里只介绍这些类的用途和基本的原理，不做过多的源码解析。

15.2.1 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方法中如果插入的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类

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，提供更高效的并发性和伸缩性。

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。

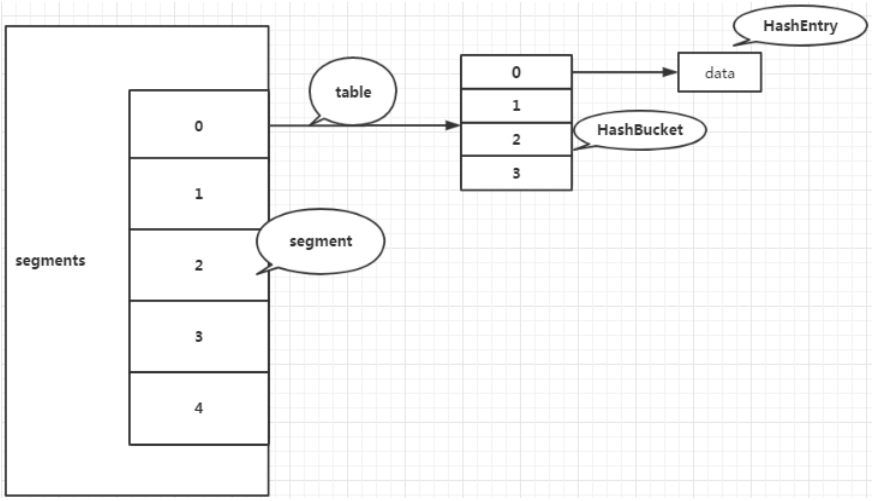
JDK 1.7

ConcurrentHashMap在JDK 1.7中，提供了一种粒度更细的加锁机制来实现在多线程下更高的性能，这种机制叫分段锁(Lock Striping)。

提供的优点是：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，操作完毕后，又按顺序释放所有段的锁。如下图：



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁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。

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。

ConcurrentNavigableMap接口与ConcurrentSkipListMap类

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）的数据结构。关于跳表的数据结构这里不做太多介绍，它是一种”空间换时间“的数据结构，可以使用CAS来保证并发安全性。

15.2.2 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，拿contains() 这样一个操作来说，很难想到搜索的时候如何避免锁住整个list。

所以退一步，JDK提供了对队列和双端队列的线程安全的类：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。

15.2.3 并发Set

JDK提供了ConcurrentSkipListSet，是线程安全的有序的集合。底层是使用ConcurrentSkipListMap实现。

谷歌的guava框架实现了一个线程安全的ConcurrentHashSet：

```
Set<String> s = Sets.newConcurrentHashSet();
```

参考资料

- Java集合-ConcurrentHashMap原理分析
- 同步容器与并发容器类简介
- ConcurrentLinkedQueue的实现原理分析
- ConcurrentHashMap的put源码解析
- 从ConcurrentHashMap能学到哪些并发编程技巧?

第十六章 CopyOnWrite容器

16.1 什么是CopyOnWrite容器

在说到CopyOnWrite容器之前我们先来谈谈什么是CopyOnWrite机制，CopyOnWrite是计算机设计领域中的一种优化策略，也是一种在并发场景下常用的设计思想——写入时复制思想。

那什么是写入时复制思想呢？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和CopyOnWriteArraySet 。我们着重给大家介绍一下CopyOnWriteArrayList。

16.2 CopyOnWriteArrayList

优点： CopyOnWriteArrayList经常被用于“读多写少”的并发场景，是因为CopyOnWriteArrayList无需任何同步措施，大大增强了读的性能。在Java中遍历线程非安全的List(如：ArrayList和 LinkedList)的时候，若中途有别的线程对List容器进行修改，那么会抛出ConcurrentModificationException异常。CopyOnWriteArrayList由于其"读写分离"，遍历和修改操作分别作用在不同的List容器，所以在使用迭代器遍历的时候，则不会抛出异常。

缺点： 第一个缺点是CopyOnWriteArrayList每次执行写操作都会将原容器进行拷贝一份，数据量大的时候，内存会存在较大的压力，可能会引起频繁Full GC（ZGC因为没有使用Full GC）。比如这些对象占用的内存200M左右，那么再写入100M数据进去，内存就会多占用300M。

第二个缺点是CopyOnWriteArrayList由于实现的原因，写和读分别作用在不同新老容器上，在写操作执行过程中，读不会阻塞，但读取到的却是老容器的数据。

现在我们来看一下CopyOnWriteArrayList的add操作源码，它的逻辑很清晰，就是先把原容器进行copy，然后在新的副本上进行“写操作”，最后再切换引用，在此过程中是加了锁的。

```
public boolean add(E e) {

    // ReentrantLock加锁，保证线程安全
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        // 拷贝原容器，长度为原容器长度加一
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 在新副本上执行添加操作
        newElements[len] = e;
        // 将原容器引用指向新副本
        setArray(newElements);
        return true;
    } finally {
        // 解锁
        lock.unlock();
    }
}
```

我们再来看一下remove操作的源码，remove的逻辑是将要remove元素之外的其他元素拷贝到新的副本中，然后再将原容器的引用指向新的副本中，因为remove操作也是“写操作”所以也是要加锁的。

```
public E remove(int index) {

        // 加锁
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            int len = elements.length;
            E oldValue = get(elements, index);
            int numMoved = len - index - 1;
            if (numMoved == 0)
                // 如果要删除的是列表末端数据，拷贝前len-1个数据到新副本上，再切换引用
                setArray(Arrays.copyOf(elements, len - 1));
            else {
                // 否则，将要删除元素之外的其他元素拷贝到新副本中，并切换引用
                Object[] newElements = new Object[len - 1];
                System.arraycopy(elements, 0, newElements, 0, index);
                System.arraycopy(elements, index + 1, newElements, index,
                                 numMoved);
                setArray(newElements);
            }
            return oldValue;
        } finally {
            // 解锁
            lock.unlock();
        }
    }
```

我们再来看看CopyOnWriteArrayList效率最高的读操作的源码

```
public E get(int index) {
    return get(getArray(), index);
}
```

```
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

由上可见“读操作”是没有加锁，直接读取。

16.3 CopyOnWrite的业务中实现

接下来，我们结合具体业务场景来实现一个CopyOnWriteMap的并发容器并且使用它。

```
import java.util.Collection;
import java.util.Map;
import java.util.Set;

public class CopyOnWriteMap<K, V> implements Map<K, V>, Cloneable {
    private volatile Map<K, V> internalMap;

    public CopyOnWriteMap() {
        internalMap = new HashMap<K, V>();
    }

    public V put(K key, V value) {
        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            V val = newMap.put(key, value);
            internalMap = newMap;
            return val;
        }
    }

    public V get(Object key) {
        return internalMap.get(key);
    }

    public void putAll(Map<? extends K, ? extends V> newData) {
        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            newMap.putAll(newData);
            internalMap = newMap;
        }
    }
}
```

上面就是参考CopyOnWriteArrayList实现的CopyOnWriteMap，我们可以用这个容器来做什么呢？结合我们之前说的CopyOnWrite的复制思想，它最适用于“读多写少”的并发场景。

场景：假如我们有一个搜索的网站需要屏蔽一些“关键字”，“黑名单”每晚定时更新，每当用户搜索的时候，“黑名单”中的关键字不会出现在搜索结果当中，并且提示用户敏感字。

```
// 黑名单服务
public class BlackListServiceImpl {
    //  减少扩容开销。根据实际需要，初始化CopyOnWriteMap的大小，避免写时CopyOnWriteMap扩容的
    private static CopyOnWriteMap<String, Boolean> blackListMap =
        new CopyOnWriteMap<String, Boolean>(1000);

    public static boolean isBlackList(String id) {
        return blackListMap.get(id) == null ? false : true;
    }

    public static void addBlackList(String id) {
        blackListMap.put(id, Boolean.TRUE);
    }

    /**
     * 批量添加黑名单
     * (使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制
     * 如使用上面代码里的addBlackList方法)
     * @param ids
     */
    public static void addBlackList(Map<String,Boolean> ids) {
        blackListMap.putAll(ids);
    }

}
```

这里需要各位小伙伴特别特别注意一个问题，此处的场景是每晚凌晨“黑名单”定时更新，原因是CopyOnWrite容器有**数据一致性**的问题，它只能保证**最终数据一致性。**

所以如果我们希望写入的数据马上能准确地读取，请不要使用CopyOnWrite容器。

参考资料

- 《Java并发编程：并发容器之CopyOnWriteArrayList》
- 聊聊并发-Java中的Copy-On-Write容器

第十七章 通信工具类

JDK中提供了一些工具类以供开发者使用。这样的话我们在遇到一些常见的应用场景时就可以使用这些工具类，而不用自己再重复造轮子了。

它们都在java.util.concurrent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

下面分别介绍这几个类。

17.1 Semaphore

17.1.1 Semaphore介绍

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“打信号”。而这个“信号”是一个 `int` 类型的数据，也可以看成是一种“资源”。

可以在构造函数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。

17.1.2 Semaphore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
                        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，且会抛出中断异常。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)
```

17.1.3 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。

17.2 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，它是处于阻塞状态的，只有当另一个线程也调用了exchange方法，它才会继续向下执行。看源码可以发现它是使用park/unpark来实现等待状态的切换的，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。

Exchanger一般用于两个线程之间更方便地在内存中交换数据，因为其支持泛型，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里面的注释的说法，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。

17.3 CountDownLatch

17.3.1 CountDownLatch介绍

先来解读一下CountDownLatch这个类名字的意义。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
```

17.3.2 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.get
                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！

17.3.3 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，有兴趣的读者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。**

17.4 CyclicBarrier

17.4.1 CyclicBarrier介绍

CyclicBarrirer从名字上来理解是“循环的屏障”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。

17.4.2 CyclicBarrier Barrier被破坏

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。

17.4.3 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，那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
}
```

17.4.4 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)
```

17.5 Phaser

17.5.1 Phaser介绍

Phaser这个单词是“移相器，相位器”的意思（好吧，笔者并不懂这是什么玩意，下方资料来自百度百科）。这个类是从JDK 1.7 中出现的。

> 移相器（Phaser）能够对波的相位进行调整的一种装置。任何传输介质对在其中传导的波动都会引入相移，这是早期模拟移相器的原理；现代电子技术发展后利用A/D、D/A转换实现了数字移相，顾名思义，它是一种不连续的移相技术，但特点是移相精度高。 移相器在雷达、导弹姿态控制、加速器、通信、仪器仪表甚至于音乐等领域都有着广泛的应用

Phaser类有点复杂，这里只介绍一些基本的用法和知识点。详情可以查看JDK文档，文档里有这个类非常详尽的介绍。

前面我们介绍了CyclicBarrier，可以发现它在构造方法里传入“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！

名词解释：

- party：对应一个线程，数量可以通过register或者构造参数传入；
- 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party，每一阶段必须所有注册的party都到达才能进入下一阶段；
- 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终止的两种途径，Phaser维护的线程执行完毕或者 `onAdvance()` 返回 `true`
此外Phaser还能维护一个树状的层级关系，构造的时候new Phaser(parentPhaser)，对于Task执行时间短的场景（竞争激烈），也就是说有大量的**party**, 那可以把每个Phaser的任务量设置较小，多个Phaser共同继承一个父Phaser。

> Phasers with large numbers of parties that would otherwise experience heavy synchronization contention costs may instead be set up so that groups of sub-phasers share a common parent. This may greatly increase throughput even though it incurs greater per-operation overhead.
>
> 翻译：如果有大量的party，那许多线程可能同步的竞争成本比较高。所以可以拆分成多个子Phaser共享一个共同的父Phaser。这可能会大大增加吞吐量，即使它会带来更多的每次操作开销。

17.5.2 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
                            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。

另外Phaser类用来控制某个阶段的线程数量很有用，但它并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，但是其实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，这一点需要读者注意。

17.5.3 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
}
```

有兴趣的读者可以去看看JDK源代码，这里不做过多叙述。

总的来说，CountDownLatch，CyclicBarrier，Phaser是一个比一个强大，但也一个比一个复杂。根据自己的业务需求合理选择即可。

参考资料

- JDK 1.8 源码
- 什么时候使用CountDownLatch
- Java 多线程基础 - CyclicBarrier
- Java7:理解Phaser

第十八章 Fork/Join框架

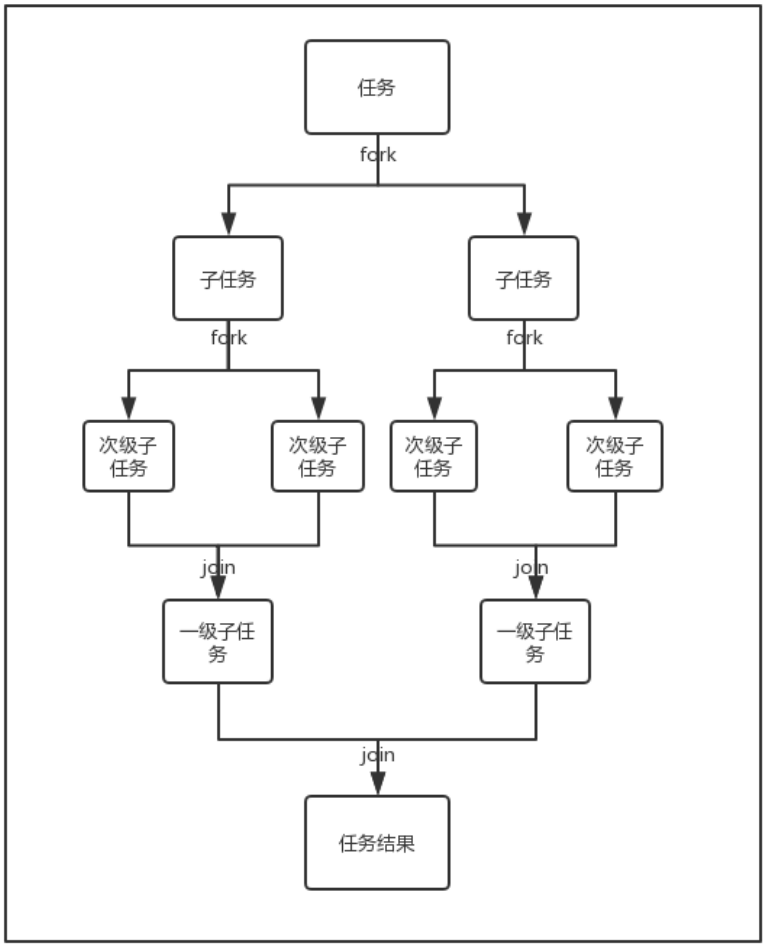
18.1 什么是Fork/Join

Fork/Join框架是一个实现了ExecutorService接口的多线程处理器，它专为那些可以通过递归分解成更细小的任务而设计，最大化的利用多核处理器来提高应用程序的性能。

与其他ExecutorService相关的实现相同的是，Fork/Join框架会将任务分配给线程池中的线程。而与之不同的是，Fork/Join框架在执行任务时使用了**工作窃取算法**。

fork在英文里有分叉的意思，**join**在英文里连接、结合的意思。顾名思义，fork就是要使一个大任务分解成若干个小任务，而join就是最后将各个小任务的结果结合起来得到大任务的结果。

Fork/Join的运行流程大致如下所示：



需要注意的是，图里的次级子任务可以一直分下去，一直分到子任务足够小为止。用伪代码来表示如下：

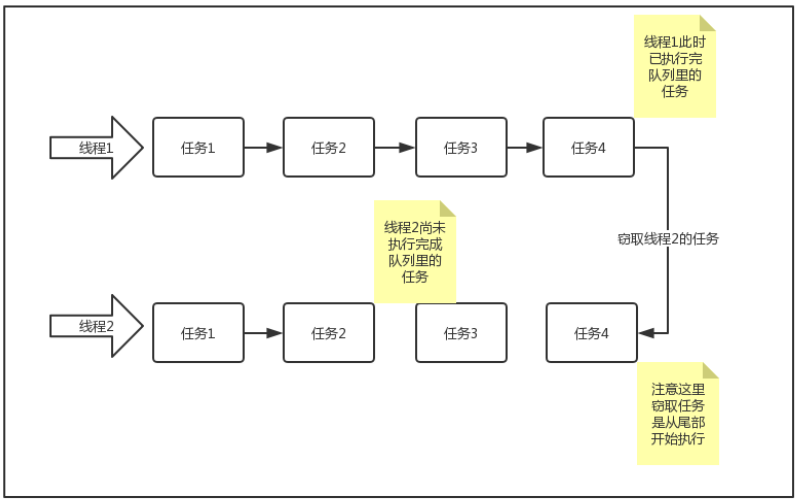
```
solve(任务):
    if(任务已经划分到足够小):
        顺序执行任务
    else:
        for(划分任务得到子任务)
            solve(子任务)
        结合所有子任务的结果到上一层循环
        return 最终结合的结果
```

通过上面伪代码可以看出，我们通过递归嵌套的计算得到最终结果，这里有体现**分而治之(divide and conquer)** 的算法思想。

18.2 工作窃取算法

工作窃取算法指的是在多线程执行不同任务队列的过程中，某个线程执行完自己队列的任务后从其他线程的任务队列里窃取任务来执行。

工作窃取流程如下图所示：



值得注意的是，当一个线程窃取另一个线程的时候，为了减少两个任务线程之间的竞争，我们通常使用**双端队列**来存储任务。被窃取的任务线程都从双端队列的**头部**拿任务执行，而窃取其他任务的线程从双端队列的**尾部**执行任务。

另外，当一个线程在窃取任务时要是没有其他可用的任务了，这个线程会进入**阻塞状态**以等待再次“工作”。

18.3 Fork/Join的具体实现

前面我们说Fork/Join框架简单来讲就是对任务的分割与子任务的合并，所以要实现这个框架，先得有**任务**。在Fork/Join框架里提供了抽象类 `ForkJoinTask` 来实现任务。

18.3.1 ForkJoinTask

ForkJoinTask是一个类似普通线程的实体，但是比普通线程轻量得多。

fork()方法:使用线程池中的空闲线程异步提交任务

```
// 本文所有代码都引自Java 8
public final ForkJoinTask<V> fork() {
    Thread t;
    // ForkJoinWorkerThread是执行ForkJoinTask的专有线程，由ForkJoinPool管理
    // 先判断当前线程是否是ForkJoin专有线程，如果是，则将任务push到当前线程所负责的队列里去
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
        ((ForkJoinWorkerThread)t).workQueue.push(this);
    else
         // 如果不是则将线程加入队列
        // 没有显式创建ForkJoinPool的时候走这里，提交任务到默认的common线程池中
        ForkJoinPool.common.externalPush(this);
    return this;
}
```

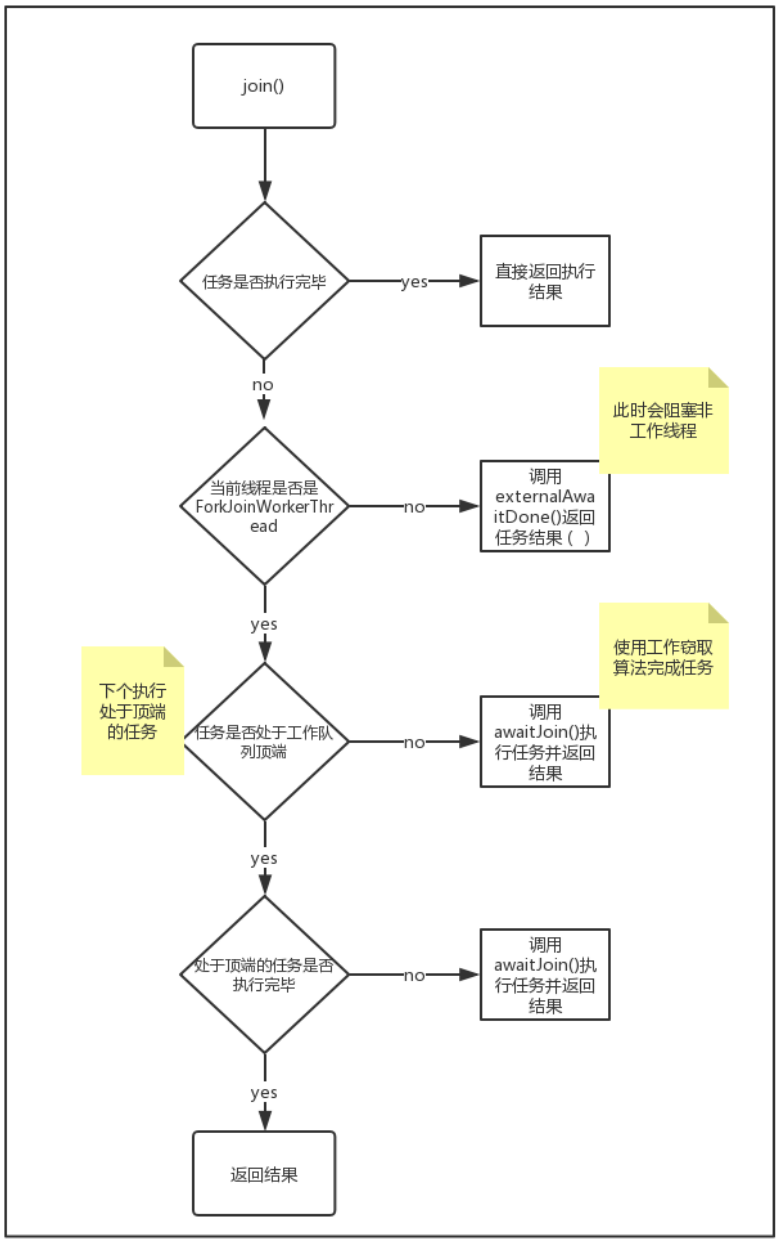
其实fork()只做了一件事，那就是**把任务推入当前工作线程的工作队列里。**

join()方法：等待处理任务的线程处理完毕，获得返回值。

来看下join()的源码：

```
public final V join() {
    int s;
    // doJoin()方法来获取当前任务的执行状态
    if ((s = doJoin() & DONE_MASK) != NORMAL)
        // 任务异常，抛出异常
        reportException(s);
    // 任务正常完成，获取返回值
    return getRawResult();
}

/**
 * doJoin()方法用来返回当前任务的执行状态
 **/
private int doJoin() {
    int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
    // 先判断任务是否执行完毕，执行完毕直接返回结果（执行状态）
    return (s = status) < 0 ? s :
    // 如果没有执行完毕，先判断是否是ForkJoinWorkThread线程
    ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) ?
        // 如果是，先判断任务是否处于工作队列顶端（意味着下一个就执行它）
        // tryUnpush()方法判断任务是否处于当前工作队列顶端，是返回true
        // doExec()方法执行任务
        (w = (wt = (ForkJoinWorkerThread)t).workQueue).
        // 如果是处于顶端并且任务执行完毕，返回结果
        tryUnpush(this) && (s = doExec()) < 0 ? s :
        // 如果不在顶端或者在顶端却没未执行完毕，那就调用awitJoin()执行任务
        // awaitJoin()：使用自旋使任务执行完成，返回结果
        wt.pool.awaitJoin(w, this, 0L) :
    // 如果不是ForkJoinWorkThread线程，执行externalAwaitDone()返回任务结果
    externalAwaitDone();
}
```

我们在之前介绍过说Thread.join()会使线程阻塞，而ForkJoinPool.join()会使线程免于阻塞，下面是ForkJoinPool.join()的流程图：



RecursiveAction和RecursiveTask

通常情况下，在创建任务的时候我们一般不直接继承ForkJoinTask，而是继承它的子类**RecursiveAction**和**RecursiveTask**。

两个都是ForkJoinTask的子类，**RecursiveAction可以看做是无返回值的ForkJoinTask，RecursiveTask是有返回值的ForkJoinTask。**

此外，两个子类都有执行主要计算的方法compute()，当然，RecursiveAction的compute()返回void，RecursiveTask的compute()有具体的返回值。

18.3.2 ForkJoinPool

ForkJoinPool是用于执行ForkJoinTask任务的执行（线程）池。

ForkJoinPool管理着执行池中的线程和任务队列，此外，执行池是否还接受任务，显示线程的运行状态也是在这里处理。

我们来大致看下ForkJoinPool的源码：

```
@sun.misc.Contended
public class ForkJoinPool extends AbstractExecutorService {
    // 任务队列
    volatile WorkQueue[] workQueues;

    // 线程的运行状态
    volatile int runState;

    // 创建ForkJoinWorkerThread的默认工厂，可以通过构造函数重写
    public static final ForkJoinWorkerThreadFactory defaultForkJoinWorkerThreadFactory

    // 公用的线程池，其运行状态不受shutdown()和shutdownNow()的影响
    static final ForkJoinPool common;

    // 私有构造方法，没有任何安全检查和参数校验，由makeCommonPool直接调用
    // 其他构造方法都是源自于此方法
    // parallelism: 并行度，
    // 默认调用java.lang.Runtime.availableProcessors() 方法返回可用处理器的数量
    private ForkJoinPool(int parallelism,
                         ForkJoinWorkerThreadFactory factory, // 工作线程工厂
                         UncaughtExceptionHandler handler, // 拒绝任务的handler
                         int mode, // 同步模式
                         String workerNamePrefix) { // 线程名prefix
        this.workerNamePrefix = workerNamePrefix;
        this.factory = factory;
        this.ueh = handler;
        this.config = (parallelism & SMASK) | mode;
        long np = (long)(-parallelism); // offset ctl counts
        this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_MASK);
    }

}
```

WorkQueue

双端队列，ForkJoinTask存放在这里。

当工作线程在处理自己的工作队列时，会从队列首取任务来执行（FIFO）；如果是窃取其他队列的任务时，窃取的任务位于所属任务队列的队尾（LIFO）。

ForkJoinPool与传统线程池最显著的区别就是它维护了一个**工作队列数组**（volatile WorkQueue[] workQueues，ForkJoinPool中的**每个工作线程都维护着一个工作队列**）。

runState

ForkJoinPool的运行状态。**SHUTDOWN**状态用负数表示，其他用2的幂次表示。

18.4 Fork/Join的使用

上面我们说ForkJoinPool负责管理线程和任务，ForkJoinTask实现fork和join操作，所以要使用Fork/Join框架就离不开这两个类了，只是在实际开发中我们常用ForkJoinTask的子类RecursiveTask 和RecursiveAction来替代ForkJoinTask。

下面我们用一个计算斐波那契数列第n项的例子来看一下Fork/Join的使用：

> 斐波那契数列数列是一个线性递推数列，从第三项开始，每一项的值都等于前两项之和：
>
> 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89……
>
> 如果设f(n）为该数列的第n项（n∈N*），那么有：f(n) = f(n-1) + f(n-2)。

```
public class FibonacciTest {

    class Fibonacci extends RecursiveTask<Integer> {

        int n;

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

        // 主要的实现逻辑都在compute()里
        @Override
        protected Integer compute() {
            // 这里先假设 n >= 0
            if (n <= 1) {
                return n;
            } else {
                // f(n-1)
                Fibonacci f1 = new Fibonacci(n - 1);
                f1.fork();
                // f(n-2)
                Fibonacci f2 = new Fibonacci(n - 2);
                f2.fork();
                // f(n) = f(n-1) + f(n-2)
                return f1.join() + f2.join();
            }
        }
    }

    @Test
    public void testFib() throws ExecutionException, InterruptedException {
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        System.out.println("CPU核数: " + Runtime.getRuntime().availableProcessors());
        long start = System.currentTimeMillis();
        Fibonacci fibonacci = new Fibonacci(40);
        Future<Integer> future = forkJoinPool.submit(fibonacci);
        System.out.println(future.get());
        long end = System.currentTimeMillis();
        System.out.println(String.format("耗时: %d millis", end - start));
    }


}
```

上面例子在本机的输出：

```
CPU核数：4
计算结果：102334155
耗时：9490 millis
```

需要注意的是，上述计算时间复杂度为 `O(2^n)`，随着n的增长计算效率会越来越低，这也是上面的例子中n不敢取太大的原因。

此外，也并不是所有的任务都适合Fork/Join框架，比如上面的例子任务划分过于细小反而体现不出效率，下面我们试试用普通的递归来求f(n)的值，看看是不是要比使用Fork/Join快：

```
// 普通递归，复杂度为O(2^n)
public int plainRecursion(int n) {
    if (n == 1 || n == 2) {
        return 1;
    } else {
        return plainRecursion(n -1) + plainRecursion(n - 2);
    }
}

@Test
public void testPlain() {
    long start = System.currentTimeMillis();
    int result = plainRecursion(40);
    long end = System.currentTimeMillis();
    System.out.println("计算结果:" + result);
    System.out.println(String.format("耗时：%d millis",  end -start));
}
```

普通递归的例子输出：

```
计算结果:102334155
耗时：436 millis
```

通过输出可以很明显的看出来，使用普通递归的效率都要比使用Fork/Join框架要高很多。

这里我们再用另一种思路来计算：

```
// 通过循环来计算，复杂度为O(n)
private int computeFibonacci(int n) {
    // 假设n >= 0
    if (n <= 1) {
        return n;
    } else {
        int first = 1;
        int second = 1;
        int third = 0;
        for (int i = 3; i <= n; i ++) {
            // 第三个数是前两个数之和
            third = first + second;
            // 前两个数右移
            first = second;
            second = third;
        }
        return third;
    }
}

@Test
public void testComputeFibonacci() {
    long start = System.currentTimeMillis();
    int result = computeFibonacci(40);
    long end = System.currentTimeMillis();
    System.out.println("计算结果:" + result);
    System.out.println(String.format("耗时: %d millis",  end -start));
}
```

上面例子在笔者所用电脑的输出为：

```
计算结果:102334155
耗时: 0 millis
```

这里耗时为0不代表没有耗时，是表明这里计算的耗时几乎可以忽略不计，大家可以在自己的电脑试试，即使是n取大很多量级的数据（注意int溢出的问题）耗时也是很短的，或者可以用System.nanoTime()统计纳秒的时间。

为什么在这里普通的递归或循环效率更快呢？因为Fork/Join是使用多个线程协作来计算的，所以会有线程通信和线程切换的开销。

如果要计算的任务比较简单（比如我们案例中的斐波那契数列），那当然是直接使用单线程会更快一些。但如果要计算的东西比较复杂，计算机又是多核的情况下，就可以充分利用多核CPU来提高计算速度。

另外，Java 8 Stream的并行操作底层就是用到了Fork/Join框架，下一章我们将从源码及案例两方面介绍Java 8 Stream的并行操作。

参考资料

- Wikipedia
- 聊聊并发（八）——Fork/Join 框架介绍
- 浅谈Java的Fork/Join并发框架

- Fork/Join 框架-设计与实现（翻译自论文《A Java Fork/Join Framework》原作者 Doug Lea）
- Java 并发编程笔记：如何使用 ForkJoinPool 以及原理
- jdk1.8-ForkJoin框架剖析
- Fork-Join 原理深入分析（二）

第十九章 Java 8 Stream并行计算原理

19.1 Java 8 Stream简介

从Java 8 开始，我们可以使用 `Stream` 接口以及**lambda表达式**进行“流式计算”。它可以让我们对集合的操作更加简洁、更加可读、更加高效。

Stream接口有非常多用于集合计算的方法，比如判空操作empty、过滤操作filter、求最max值、查找操作findFirst和findAny等等。

19.2 Stream单线程串行计算

Stream接口默认是使用串行的方式，也就是说在一个线程里执行。下面举一个例子：

```
public class StreamDemo {
    public static void main(String[] args) {
        Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
                .reduce((a, b) -> {
                    System.out.println(String.format("%s: %d + %d = %d",
                            Thread.currentThread().getName(), a, b, a + b));
                    return a + b;
                })
                .ifPresent(System.out::println);
    }
}
```

我们来理解一下这个方法。首先我们用整数1~9创建了一个 `Stream` 。这里的Stream.of(T... values)方法是Stream接口的一个静态方法，其底层调用的是Arrays.stream(T[] array)方法。

然后我们使用了 `reduce` 方法来计算这个集合的累加和。 `reduce` 方法这里做的是：从前两个元素开始，进行某种操作（我这里进行的是加法操作）后，返回一个结果，然后再拿这个结果跟第三个元素执行同样的操作，以此类推，直到最后的一个元素。

我们来打印一下当前这个reduce操作的线程以及它们被操作的元素和返回的结果以及最后所有reduce方法的结果，也就代表的是数字1到9的累加和。

```
main: 1 + 2 = 3
main: 3 + 3 = 6
main: 6 + 4 = 10
main: 10 + 5 = 15
main: 15 + 6 = 21
main: 21 + 7 = 28
main: 28 + 8 = 36
main: 36 + 9 = 45
45
```

可以看到，默认情况下，它是在一个单线程运行的，也就是**main**线程。然后每次reduce操作都是串行起来的，首先计算前两个数字的和，然后再往后依次计算。

19.3 Stream多线程并行计算

我们思考上面一个例子，是不是一定要在单线程里进行串行地计算呢？假如我的计算机是一个多核计算机，我们在理论上能否利用多核来进行并行计算，提高计算效率呢？

当然可以，比如我们在计算前两个元素1 + 2 = 3的时候，其实我们也可以同时在另一个核计算 3 + 4 = 7。然后等它们都计算完成之后，再计算 3 + 7 = 10的操作。

是不是很熟悉这样的操作手法？没错，它就是ForkJoin框架的思想。下面小小地修改一下上面的代码，增加一行代码，使Stream使用多线程来并行计算：

```
public class StreamParallelDemo {
    public static void main(String[] args) {
        Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
                .parallel()
                .reduce((a, b) -> {
                    System.out.println(String.format("%s: %d + %d = %d",
                            Thread.currentThread().getName(), a, b, a + b));
                    return a + b;
                })
                .ifPresent(System.out::println);
    }
}
```

可以看到，与上一个案例的代码只有一点点区别，就是在reduce方法被调用之前，调用了parallel()方法。下面来看看这个方法的输出：

```
ForkJoinPool.commonPool-worker-1: 3 + 4 = 7
ForkJoinPool.commonPool-worker-4: 8 + 9 = 17
ForkJoinPool.commonPool-worker-2: 5 + 6 = 11
ForkJoinPool.commonPool-worker-3: 1 + 2 = 3
ForkJoinPool.commonPool-worker-4: 7 + 17 = 24
ForkJoinPool.commonPool-worker-4: 11 + 24 = 35
ForkJoinPool.commonPool-worker-3: 3 + 7 = 10
ForkJoinPool.commonPool-worker-3: 10 + 35 = 45
45
```

可以很明显地看到，它使用的线程是 `ForkJoinPool` 里面的 `commonPool` 里面的**worker**线程。并且它们是并行计算的，并不是串行计算的。但由于Fork/Join框架的作用，它最终能很好的协调计算结果，使得计算结果完全正确。

如果我们用Fork/Join代码去实现这样一个功能，那无疑是非常复杂的。但Java8提供了并行式的流式计算，大大简化了我们的代码量，使得我们只需要写很少很简单的代码就可以利用计算机底层的多核资源。

19.4 从源码看Stream并行计算原理

上面我们通过在控制台输出线程的名字，看到了Stream的并行计算底层其实是使用的Fork/Join框架。那它到底是在哪使用Fork/Join的呢？我们从源码上来解析一下上述案例。

`Stream.of` 方法就不说了，它只是生成一个简单的Stream。先来看看 `parallel()` 方法的源码。这里由于我的数据是 `int` 类型的，所以它其实是使用的 `BaseStream` 接口的 `parallel()` 方法。而 `BaseStream` 接口的JDK唯一实现类是一个叫 `AbstractPipeline` 的类。下面我们来看看这个类的 `parallel()` 方法的代码：

```
public final S parallel() {
    sourceStage.parallel = true;
    return (S) this;
}
```

这个方法很简单，就是把一个标识 `sourceStage.parallel` 设置为 `true` 。然后返回实例本身。

接着我们再来看 `reduce` 这个方法的内部实现。

Stream.reduce()方法的具体实现是交给了 `ReferencePipeline` 这个抽象类，它是继承了 `AbstractPipeline` 这个类的:

```
// ReferencePipeline抽象类的reduce方法
@Override
public final Optional<P_OUT> reduce(BinaryOperator<P_OUT> accumulator) {
    // 调用evaluate方法
    return evaluate(ReduceOps.makeRef(accumulator));
}

final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
    assert getOutputShape() == terminalOp.inputShape();
    if (linkedOrConsumed)
        throw new IllegalStateException(MSG_STREAM_LINKED);
    linkedOrConsumed = true;

    return isParallel() // 调用isParallel()判断是否使用并行模式
        ? terminalOp.evaluateParallel(this, sourceSpliterator(terminalOp.getOpFlags())
        : terminalOp.evaluateSequential(this, sourceSpliterator(terminalOp.getOpFlags(
}

@Override
public final boolean isParallel() {
    // 根据之前在parallel()方法设置的那个flag来判断。
    return sourceStage.parallel;
}
```

从它的源码可以知道，reduce方法调用了evaluate方法，而evaluate方法会先去检查当前的flag，是否使用并行模式，如果是则会调用 `evaluateParallel` 方法执行并行计算，否则，会调用 `evaluateSequential` 方法执行串行计算。

这里我们再看看 `TerminalOp` （注意这里是字母I O，而不是数字1 0）接口的 `evaluateParallel` 方法。 `TerminalOp` 接口的实现类有这样几个内部类：

- java.util.stream.FindOps.FindOp
- java.util.stream.ForEachOps.ForEachOp
- java.util.stream.MatchOps.MatchOp
- java.util.stream.ReduceOps.ReduceOp

可以看到，对应的是Stream的几种主要的计算操作。我们这里的示例代码使用的是reduce计算，那我们就看看ReduceOp类的这个方法的源码：

```
// java.util.stream.ReduceOps.ReduceOp.evaluateParallel
@Override
public <P_IN> R evaluateParallel(PipelineHelper<T> helper,
                                 Spliterator<P_IN> spliterator) {
    return new ReduceTask<>(this, helper, spliterator).invoke().get();
}
```

evaluateParallel方法创建了一个新的ReduceTask实例，并且调用了invoke()方法后再调用get()方法，然后返回这个结果。那这个ReduceTask是什么呢？它的invoke方法内部又是什么呢？

追溯源码我们可以发现，ReduceTask类是ReduceOps类的一个内部类，它继承了AbstractTask类，而AbstractTask类又继承了CountedCompleter类，而CountedCompleter类又继承了ForkJoinTask类！

它们的继承关系如下：

> ReduceTask -> AbstractTask -> CountedCompleter -> ForkJoinTask

这里的ReduceTask的invoke方法，其实是调用的ForkJoinTask的invoke方法，中间三层继承并没有覆盖这个方法的实现。

所以这就从源码层面解释了Stream并行的底层原理是使用了Fork/Join框架。

需要注意的是，一个Java进程的Stream并行计算任务默认共享同一个线程池，如果随意的使用并行特性可能会导致方法的吞吐量下降。我们可以通过下面这种方式来让你的某个并行Stream使用自定义的ForkJoin线程池：

```
ForkJoinPool customThreadPool = new ForkJoinPool(4);
long actualTotal = customThreadPool
  .submit(() -> roster.parallelStream().reduce(0, Integer::sum)).get();
```

19.5 Stream并行计算的性能提升

我们可以在本地测试一下如果在多核情况下，Stream并行计算会给我们的程序带来多大的效率上的提升。用以下示例代码来计算一千万个随机数的和：

```
public class StreamParallelDemo {
    public static void main(String[] args) {
        System.out.println(String.format("本计算机的核数：%d", Runtime.getRuntime().ava

        // 产生100w个随机数(1 ~ 100)，组成列表
        Random random = new Random();
        List<Integer> list = new ArrayList<>(1000_0000);

        for (int i = 0; i < 1000_0000; i++) {
            list.add(random.nextInt(100));
        }

        long prevTime = getCurrentTime();
        list.stream().reduce((a, b) -> a + b).ifPresent(System.out::println);
        System.out.println(String.format("单线程计算耗时：%d", getCurrentTime() - prevT

        prevTime = getCurrentTime();
        list.stream().parallel().reduce((a, b) -> a + b).ifPresent(System.out::println
        System.out.println(String.format("多线程计算耗时：%d", getCurrentTime() - prevT

    }

    private static long getCurrentTime() {
        return System.currentTimeMillis();
    }
}
```

输出：

> 本计算机的核数：8
> 495156156
> 单线程计算耗时：223
> 495156156
> 多线程计算耗时：95

所以在多核的情况下，使用Stream的并行计算确实比串行计算能带来很大效率上的提升，并且也能保证结果计算完全准确。

本文一直在强调的“多核”的情况。其实可以看到，我的本地电脑有8核，但并行计算耗时并不是单线程计算耗时除以8，因为线程的创建、销毁以及维护线程上下文的切换等等都有一定的开销。所以如果你的服务器并不是多核服务器，那也没必要用Stream的并行计算。因为在单核的情况下，往往Stream的串行计算比并行计算更快，因为它不需要线程切换的开销。

参考资料

- Java8 Stream 并行计算实现的原理

第二十章 计划任务

自JDK 1.5 开始，JDK提供了 `ScheduledThreadPoolExecutor` 类用于计划任务（又称定时任务），这个类有两个用途：

- 在给定的延迟之后运行任务
- 周期性重复执行任务

在这之前，是使用 `Timer` 类来完成定时任务的，但是 `Timer` 有缺陷：

- Timer是**单线程**模式；
- 如果在执行任务期间某个TimerTask耗时较久，那么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，所以一旦出现异常，则线程就会终止，其他任务也得不到执行。

所以JDK 1.5之后，大家就摒弃 `Timer` ,使用 `ScheduledThreadPoolExecutor` 吧。

20.1 使用案例

假设我有一个需求，指定时间给大家发送消息。那么我们会将消息（包含发送时间）存储在数据库中，然后想用一个定时任务，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写？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;
    }
}
```

下面截取前面的输出（这个demo会一直运行下去）：

```
2019-01-23 16:16:48
大家注意了，我要发消息了
2019-01-23 16:16:49
大家注意了，我要发消息了
2019-01-23 16:16:50
大家注意了，我要发消息了
2019-01-23 16:16:51
大家注意了，我要发消息了
2019-01-23 16:16:52
大家注意了，我要发消息了
2019-01-23 16:16:53
大家注意了，我要发消息了
2019-01-23 16:16:54
大家注意了，我要发消息了
2019-01-23 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，想要知道奥秘，接下来的东西需要仔细的看哦。

20.2 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`，实现了 `ScheduledExecutorService`。 线程池在之前的章节介绍过了，我们先看看 `ScheduledExecutorService`。

```
public interface ScheduledExecutorService extends ExecutorService {

    public ScheduledFuture<?> schedule(Runnable command,long delay, TimeUnit unit);

    public <V> ScheduledFuture<V> schedule(Callable<V> callable,long delay, TimeUnit u

    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit);
}
```

`ScheduledExecutorService` 实现了 `ExecutorService`，并增加若干定时相关的接口。 前两个方法用于单次调度执行任务，区别是有没有返回值。

重点理解一下后面两个方法：

- scheduleAtFixedRate

  该方法在 `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再次启动任务。

- scheduleWithFixDelay

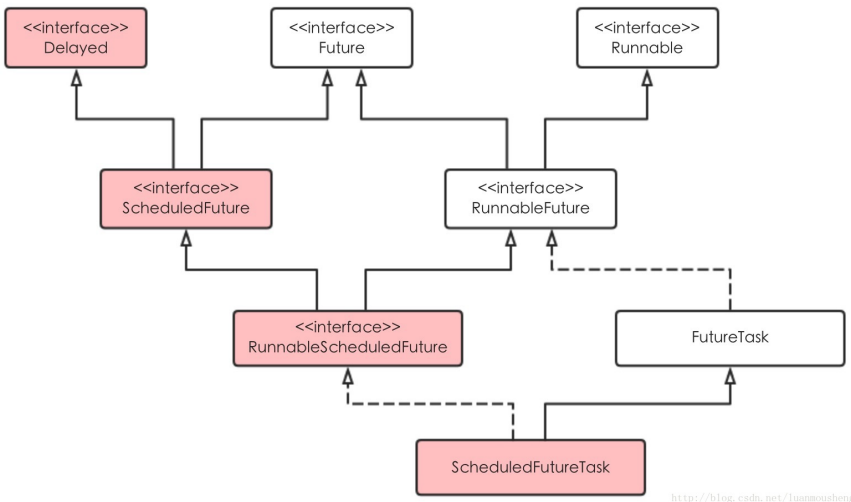
  该方法在 `initialDelay` 时长后第一次执行任务，以后每当任务执行**完成后**，等待 `delay` 时长，再次执行任务。

20.3 主要方法介绍

20.3.1 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
    // 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 `ScheduledFuture`、`RunnableScheduledFuture`、`ScheduledFutureTask` 的关系：



我们先看看这几个接口和类：

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> {
}
```

没有添加其他方法。

RunnableScheduledFuture接口

```
public interface RunnableScheduledFuture<V> extends RunnableFuture<V>, ScheduledFuture
    // 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` 的对象，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
      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可以执行任务，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，维持任务的状态为初始状态（**NEW**状态），这也是该方法和 `FutureTask` 的 `run` 方法的区别。

20.3.2 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 。

20.3.3 scheduleWithFixedDelay

`FixedDelay` 也是通过 `ScheduledFutureTask` 体现的，唯一不同的地方在于创建的 `ScheduledFutureTask` 不同 。这里不再展示源码。

20.3.4 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` ，线程池中的工作线程是通过该方法来启动并执行任务的。 具体可以查看前面讲的线程池章节。

对于 `ScheduledThreadPoolExecutor` ， `worker` 添加到线程池后会在等待队列上等待获取任务，这点是和 `ThreadPoolExecutor` 一致的。**但是worker是怎么从等待队列取定时任务的?**

因为 `ScheduledThreadPoolExecutor` 使用了 `DelayedWorkQueue` 保存等待的任务，该等待队列队首应该保存的是最近将要执行的任务，如果队首任务的开始执行时间还未到， `worker` 也应该继续等待。

20.4 DelayedWorkQueue

`ScheduledThreadPoolExecutor` 使用了 `DelayedWorkQueue` 保存等待的任务。

该等待队列队首应该保存的是**最近将要执行的任务**，所以 `worker` 只关心队首任务即可，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，底层是使用堆结构来实现优先队列的功能。我们先看看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_CAPA
    // 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，它只要等待队首任务的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时，leader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。

同时，定义了锁lock和监视器available用于线程竞争成为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。

接下来看看 `DelayedWorkQueue` 中几个比较重要的方法

20.4.1 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
          // 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？在线程池的章节中，介绍了 `getTask` 方法，工作线程会循环地从 `workQueue` 中取任务。但计划任务却不同，因为如果一旦 `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的时间，所以在 `take` 方法中，要保证只有在到指定的执行时间的时候任务才可以被取走。

总结一下流程：

1. 如果堆顶元素为空，在available条件上等待。
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线程必须在从 `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()` ）。

20.4.2 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;
}
```

在堆中插入了一个节点，这个时候堆有可能不满足最小堆的定义， `siftUp` 用于将堆调整为最小堆，这属于数据结构的基本内容，本文不做介绍。

20.5 总结

内部使用优化的DelayQueue来实现，由于使用队列来实现定时器，有出入队调整堆等操作，所以定时并不是非常非常精确。

参考资料

- 线程池原理（四）
- ScheduleThreadPoolExecutor详解
- 深入理解Java线程池：ScheduledThreadPoolExecutor