THE ROAD TO THE KING

王者并发课

从青铜到王者，结构化认知Java并发体系

秦二爷 著

关于这份文集

《王者并发课》的全称是 **《王者并发课：结构化认知Java中的并发》**，是首发于**掘金专栏**的系列文章。在这份文集中，我们将按照理解难度的不同循序渐进带你**结构化**认知Java中的并发。

该系列文章以不同的段位进行组织，不同的段位代表着不同的**认知难度**。之所以使用王者中的**段位**概念，不仅是因为有调和文章趣味性的考虑，更为重要的是，随着认知难度的增加，我们坚持下去的难度也同样在增加，甚至后者来得更为强烈。**具备攻坚克难的决心和定力，是一名优秀程序员的基本品质**。

所谓“**行百里者半九十**”，对读者如此，对我来说更是如此。这份文集从2021年5月底开始，至2022年6月初版完，共**29**篇，约**10万**余字。历时一年有余，几经周折但最终坚持了下来。当你觉得这份文集对你有所帮助时，希望你也能坚持下来，我们一起做**长期主义者，从青铜到王者**。

你可以学到什么

- 认知Java中并发基础实践和理论；
- 认知并发编程中所面临的问题以及对应的解决策略；
- 认知并发背后的底层原理；
- 理解并掌握Java中的常用并发工具；
- 注：该系列文章并非面向求职面试而写，但可以作为面试辅助材料。

适合的读者

- 已经在Java编程方面具有一定经验和基础知识的学生和职场新人；
- 想要在短时间内获得多线程，并行编程和并发技能的学生和职场新人；
- 希望结构化认知Java并发编程的专业开发者；
- 正在求职路上并希望结构化梳理Java并发知识的专业开发者。

如何获取最新版本

我将会不定期对小册进行版本修订，比如错误更正、内容扩充等。如果你有兴趣，可以关注公众号，及时获取最新版本推送，也可以在公众号菜单里手动获取最新版本。

关注公众号，及时获取最新版本

无广告，无推销，无骚扰。

协作与建议反馈

在文集成稿的过程，特别感谢以下同学对文集的贡献和反馈（非全部名单，排名不分先后）：

- **WinsonWu**
- Ddot
- 越河之卒
- **ClyingDeng**
- 南风的夜
- 庞亮
- gdx
- 半岛铁板
- **程序员小杰**
- **LBJ**
- **Fly**
- **花哥编程**

如何提交错误

该文集非学术性著作，仅供同行学习交流。出于作者水平有限，且面向博客输出，难免会存在严谨性、理论错误、代码错误等问题。你可以通过以下方式提交你的**意见**、**批评**和**建议**：

- 访问github提交issue：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**
- 访问掘金专栏找到文章提交评论：**https://juejin.cn/column/6963590682602635294**
- 关注公众号【MetaThoughts】给我留言

在收到反馈后，我会在1~3个工作日给你回复。**如果你的反馈被采纳，你的名字会加到文集最新版本的贡献者名单上**。

目录与结构

- 铂金8：峡谷幽会-看CyclicBarrier如何跨越重峦叠嶂
- 铂金9：互通有无-Exchanger如何完成线程间的数据交换

四、抬首：驾驭线程池与并发集合 | 砖石⭐⭐⭐⭐

- 钻石1：明心见性-如何由表及里精通线程池设计与原理
- 钻石2：分而治之-如何从原理深入理解ForkJoinPool的快与慢
- 钻石3：琳琅满目-细数CompletableFuture的那些花式玩法

五、低眉：洞见并发底层的原理 | 星耀⭐⭐⭐⭐⭐

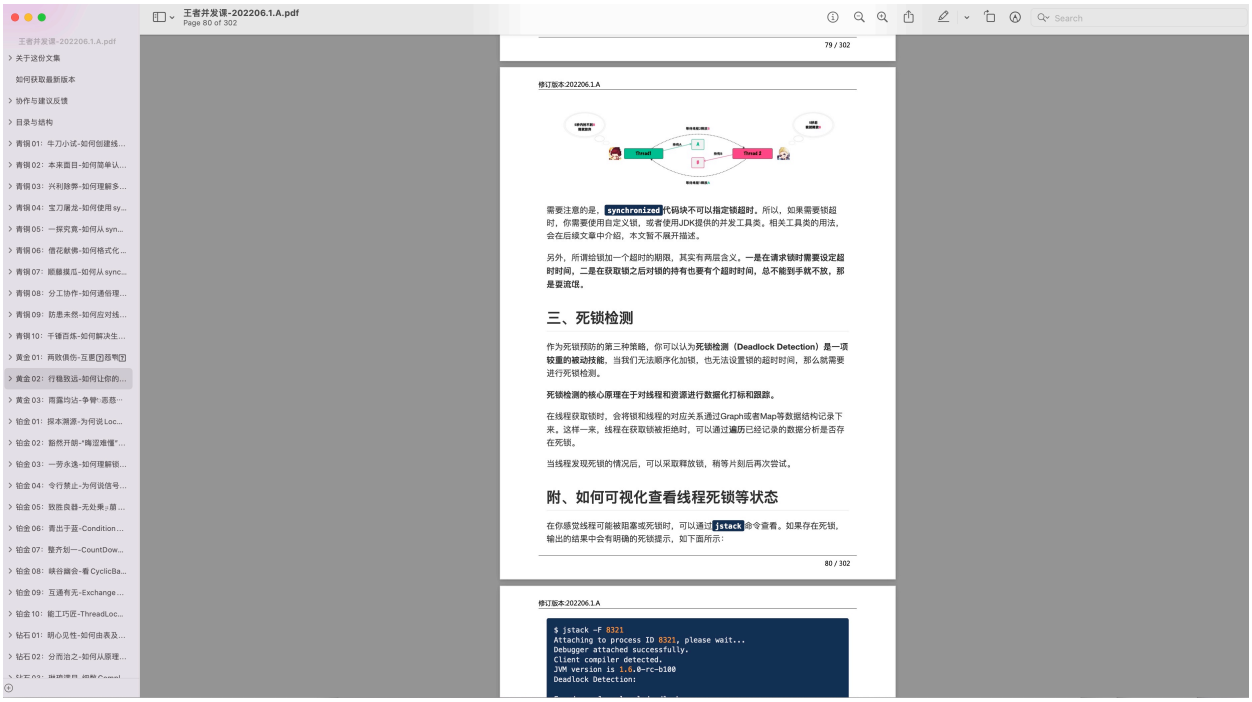
- 星耀1：群雄逐鹿-从鹿死谁手深入理解Java内存模型
- 星耀2：穷理尽妙-解构同步器设计原理与AQS浅析
- 星耀3：自在不羁-领会非阻塞的同步机制和算法

六、觉醒：领略并发中的高级主题| 王者⭐⭐⭐⭐⭐⭐

特别说明：限于作者的认知和写作水平，第六部分的**王者段位**暂不更新。如果你希望了解真实的架构中如何处理高并发问题，可以考虑阅读我的掘金小册：《高并发秒杀的设计精要与实现》。

PDF阅读提示

阅读提示：该文集支持PDF目录浏览，阅读时可以通过PDF目录跳转到不同的章节。



青铜01：牛刀小试-如何创建线程之初体验

欢迎来到《**王者并发课**》，本文是该系列文章中的**第1篇**。

从本文开始，我将基于王者中的段位和场景，从**青铜**、**黄金**、**铂金**、**砖石**、**星耀**到**王者**，不同的段位对应不同的难易程度，由浅入深逐步介绍JAVA中的并发编程，并在每**周二**、**四**、**六**持续更新。

在文章的知识体系方面，主要以实践为主，并在实践中穿插理论知识的讲解，而本文将从最简单的线程创建开始。

一、一个游戏场景

在本局游戏中，将有3位玩家出场，他们分别是哪吒、苏烈和安其拉。根据玩家不同的角色定位，在王者峡谷中，他们会有不同的游戏路线：

- 作为战士的哪吒将走上路的对抗路线；
- 法师安其拉则去镇守中路；
- 战坦苏烈则决定去下路。

二、代码实现

显而易见，你已经发现这几个玩家肯定不是单线程。接下来，我们将通过简单的多线程模拟出他们的路线。当然，真实的游戏引擎中绝不会是几个简单的线程，情况会复杂很多。

```
public static void main(String[] args) {
        Thread neZhaPlayer = new Thread() {
            public void run() {
                System.out.println("我是哪吒，我去上路");
```

```
            }
        };
        Thread anQiLaPlayer = new Thread() {
            public void run() {
                System.out.println("我是安其拉，我去中路");
            }
        };
        Thread suLiePlayer = new Thread() {
            public void run() {
                System.out.println("我是苏烈，我去下路");
            }
        };
        neZhaPlayer.start();
        anQiLaPlayer.start();
        suLiePlayer.start();
}
```

代码的运行结果：

```
我是哪吒，我去上路
我是苏烈，我去下路
我是安其拉，我去中路

Process finished with exit code 0
```

以上，就是游戏中简单的代码片段。我们创建了3个线程表示3个玩家，并通过`run()`方法实现他们的路线动作，随后通过`start()`启动线程。它足够简单，然而这里有3个知识点需要你留意。

1. 创建线程

```
Thread neZhaPlayer = new Thread();
```

2. 执行代码片段

```
public void run() {
    System.out.println("我是哪吒，我去上路");
}
```

3. 启动线程

```
neZhaPlayer.start();
```

对于我们来说，创建线程并不是我们的目标，**我们的目标是运行我们期望的代码（比如玩家的游戏路线或某个动作）**，而线程只是我们实现这一目标的方式。因此，在编写多线程代码时，运行指定的代码片段无疑是极其重要的。在Java中，我们主要有2种方式来指定：

- 继承`Thread`并重写`run`方法；
- 实现`Runnable`接口并将其传递给`Thread`的构造器。

三、线程创建的两种方式

1. 继承Thread创建线程

在上面的示例代码中，我们所使用的正是这种方式，只不过是匿名实现，你也可以通过显示继承：

```
public class NeZhaPlayer extends Thread {
    public void run() {
        System.out.println("我是哪吒，我去上路");
    }
}
```

此外，在Java以及更高的JDK版本中，你还可以通过lambda表达式简化代码：

```
Thread anQiLaPlayer = new Thread(() -> System.out.println("我是哪吒，我去上路"));
```

2. 实现Runnable接口创建线程

创建线程的第2种方法是实现`Runnable`接口。我们创建了`NeZhaRunnable`类并实现`Runnable`接口中的`run`方法，如下面代码所示。

```
public class NeZhaRunnable implements Runnable {
```

```
    public void run() {
        System.out.println("我是哪吒，我去上路");
    }
}

Thread neZhaPlayer = new Thread(new NeZhaRunnable());
neZhaPlayer.start();
```

从效果上看，两种方式创建出来的线程效果是一样的。那么，我们应该怎么选择?

建议你使用Runnable

对于这两种方法，孰优孰劣并没有明确的规定。**但是，从面向对象设计的角度来说，推荐你用第二种方式：实现Runnable接口**。

这是因为，在面向对象设计中，有一条约定俗成的规则，**组合优于继承（Prefer composition over inheritance）**，如果没有特别的目的需要重写父类方法，尽量不要使用继承。在Java中所有的类都只能是单继承，一旦继承Thread之后将不能继承其他类，严重影响类的扩展和灵活性。另外，实现Runnable接口也可以与后面的更高级的并发工具结合使用。

所以，相较于继承Thread，实现Runnable接口可以降低代码之间的耦合，保持更好的灵活性。关于这一原则的更多描述，你可以参考《Effective Java》。

当然，如果你对Thread情有独钟，当我没说。此外，在Java中我们还可以通过`ThreadFactory`等工具类创建线程，不过本质上仍是对这两种方法的封装。

四、注意，别踩坑!

线程的启动固然简单，然而对于一些新手来说，在启动线程的时候，一不小心就会使用`run()`而不是`start()`，就像下面这样:

```
Thread neZhaPlayer = new Thread(new NeZhaRunnable());
neZhaPlayer.run();
```

如果你这么调用的话，**你仍然可以看到你期望的输出，然而这正是陷阱所在**！这是

因为，Runnable中的`run()`方法并不是你所创建的线程调用的，而是调用你这个线程的线程调用的，也就是**主线程**。那为什么直接调用`run()`方法也能看到输出呢？这是因为Thread中的`run()`会直接调用**target**中的`run()`:

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

所以你看，如果你直接调用`run()`的话，并不会创建新的线程。关于这两个方法的执行细节，会在后面的线程状态中分析，**这里你要记住的就是启动线程调用的是`start()`，而不是`run()`** 。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

- 用两种不同的方式，创建出两个线程，交叉打印1~100之间的奇数和偶数，并断点调试。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜02：本来面目-如何简单认识Java中的线程

欢迎来到《王者并发课》，本文是该系列文章中的**第2篇**。

在前面的《兵分三路：如何创建多线程》文章中，我们已经通过Thread和Runnable直观地了解**如何在Java中创建一个线程**，相信你已经有了一定的体感。在本篇文章中，我们将基于前面的示例代码，对线程做些必要的说明，以帮助你从更基础的层面认知线程，并为后续的学习打下基础。

一、从进程认知线程

在上世纪的80年代中期之前，**进程**一直都是操作系统中**拥有资源**和**独立运行**的基本单位。可是，随着计算机的发展，人们对操作系统的吞吐量要求越来越高，并且多处理器也逐渐发展起来，进程作为基本的调度单位已经越来越不合时宜，因为它太重了。

想想看，我们在创建进程的时候，需要给它们创建PCB，并且还要分配需要的所有资源，如内存空间、I/O设备等等。然而，在进程切换时，系统还需要保留当前进程的CPU环境并设置新的进程CPU环境，这些都是需要花费时间的。也就是说，在**进程的创建、切换以及销毁的过程中，系统要花费巨大的时空开销**。如此，在一个操作系统中，就不能设置过多数量的进程，并且还不能频繁切换。显然，这不符合时代的发展需要了。

因此，进程**切换的巨大开销**和**多核CPU**的发展，线程便顺势而生。

从概念上，线程可以理解为**它是操作系统中独立运行的基本单元，一个进程可以拥有多个线程**。并且，**和进程相比，线程拥有进程很多相似属性，因此线程有时候也被称为轻量级进程（Light-Weight Process）**。

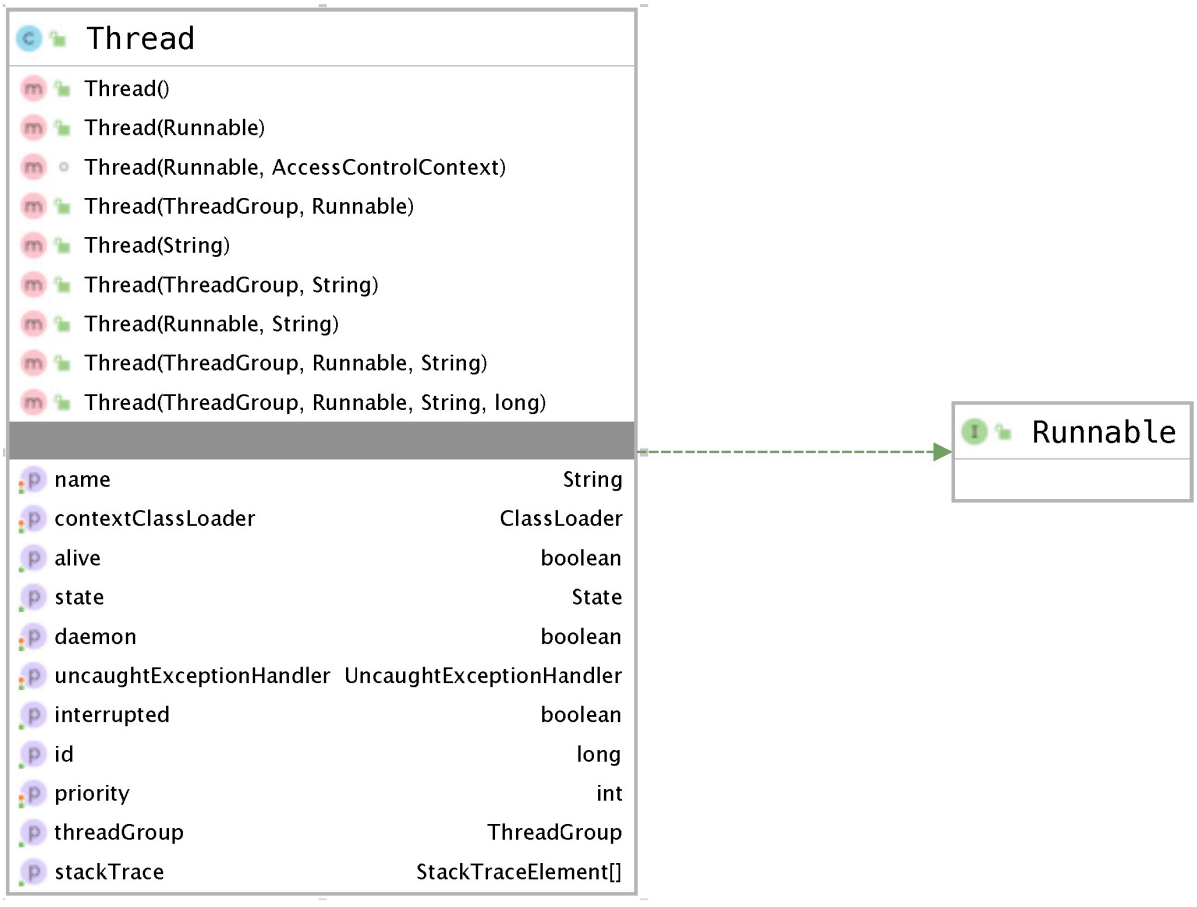
为什么线程相对轻量

在线程之前，系统切换任务时需要切换进程，开销巨大。然而，引入线程后，线程隶属于进程，**进程仍是资源的拥有者，线程只占据少量的资源**。同时，线程的切换并不会导致进程的切换，因此开销较小。此外，进程和线程都可以并发执行，操作系统也因此获得了更好的并发性，也能有效地提高系统资源的利用率和系统吞吐量。

二、Thread类-Java中的线程基础

在本文的第一部分，我们从**操作系统**层面认识了**线程**。而在Java中，我们则需要通过**Thread**类认知线程，包括它的一些基本属性和基本方法。Thread是Java多线程基础中的基础，请不要因为它简单就忽略这部分的内容。

下面的这幅图概括展示了Thread类的核心属性和方法：



1. Thread中如何构造线程

Thread中共有9个公共构造器，当然我们不用掌握全部的构造，熟悉其中几个比较常用的即可：

- `Thread()`，这个构造器会默认生成一个`Thread-`+`n`的名字，`n`是由内部方法`nextThreadNum()`生成的一个整型数字；
- `Thread(String name)`，在构建线程时指定线程名，是一个很不错的实践；
- `Thread(Runnable target)`，传入`Runnable`的实例，这个我们在上一篇文章中已经展示过；
- `Thread(Runnable target, String name)`，在传入`Runnable`实例时指定线程名。

2. Thread中的关键属性

从`init()`方法理解Thread的构造

虽然Thread有9个构造函数，但最终都是通过下面的这个`init()`方法进行构造，所以了解了这个方法，就了解了Thread的构造过程。

```
    /**
     * Initializes a Thread.
     *
     * @param g the Thread group
     * @param target the object whose run() method gets called
     * @param name the name of the new Thread
     * @param stackSize the desired stack size for the new
thread, or
     *        zero to indicate that this parameter is to be
ignored.
     * @param acc the AccessControlContext to inherit, or
     *            AccessController.getContext() if null
     * @param inheritThreadLocals if {@code true}, inherit
initial values for
     *            inheritable thread-locals from the constructing
thread
     */
    private void init(ThreadGroup g, Runnable target, String
```

```
name,
                      long stackSize, AccessControlContext acc,
                      boolean inheritThreadLocals)
```

`init()`方法几十行的代码，不过为了节省篇幅，我们此处只贴出了方法的签名，具体的方法体内容可以自行去看。在方法的签名中，有几个重要的参数：

- `g`：线程所属的线程组。线程组是一组具有相似行为的线程集合；
- `target`：继承`Runnable`的对象实例；
- `name`：线程的名字；

其他几个参数通常不需要自定义，保持默认即可。如果你翻看`init()`的方法体代码，可以看到`init()`对下面几个属性做了初始化：

- `name`：线程的名字；
- `group`：线程所属的线程组；
- `daemon`：是否为守护进程；
- `priority`：线程的优先级；
- `tid`：线程的ID；

关于名字

虽然Thread默认会生成一个线程名，但为了方便日志输出和问题排查，比较建议你在创建线程时自己手动设置名称，比如`anQiLaPlayer`的线程名可以设置为`Thread-anQiLa`。

关于线程ID

和线程名一样，每个线程都有自己的ID，如果你没有指定的话，Thread会自动生成。确切地说，线程的ID是根据`threadSeqNumber()`对Thread的静态变量`threadSeqNumber`进行累加得到：

```
private static synchronized long nextThreadID() {
    return ++threadSeqNumber;
}
```

关于线程优先级

在创建新的线程时，线程的优先级默认和当前父线程的优先级一致，当然我们也可以通过`setPriority(int newPriority)`方法来设置。不过，在设置线程优先级时需要注意两点：

- **Thread线程的优先级设置是不可靠的**：我们可以通过数字来指定线程调度时的优先级，然而最终执行时的调度顺序将由操作系统决定，因为Thread中的优先级设置并不是和所有的操作系统一一对应；
- **线程组的优先级高于线程优先级**：每个线程都会有一个线程组，我们所设置的线程优先级数字不能高于线程组的优先级。如果高于，将会直接使用线程组的优先级。

3. 线程中的关键方法

Thread中几个重要的方法，
如`start()`、`join()`、`sleep()`、`yield()`、`interrupted()`等，关于这几种方法的用法，我们会在下一篇文章中结合线程的状态进行讲解。**需要注意的是，`notify()`、`wait()`等并不是Thread类的方法，它们是Object的方法。**

三、多线程的应用场景

通过前面的分析，我们已经从操作系统层面和Java中认知了线程。那么，什么样的场景需要考虑使用多线程?

总的来说，当你遇到以下两类场景时，需要考虑多线程：

1. 异步

当两个**独立逻辑单元**不需要同步顺序完成时，可以通过多线程异步处理。

比如，用户注册后发送邮件消息。很显然，**注册**和**发送消息**是两个独立逻辑单元，在注册完成后，我们可以另起线程完成消息的发送，从而实现逻辑解耦并缩短注册单元的响应时间。

2. 并发
现在的计算机基本都是多核处理器，在处理批量任务时，可以通过多线程提高处理速度。

比如，假设系统需要向100万的用户发送消息。可以想象，如果单线程处理不知道猴年马月才能完成。而此时，我们便可以通过线程池创建多线程大幅提高效率。

注意，对于一些同学来说，你可能还没有接触过多线程的应用场景。但是，请不要因为工作中的场景简单或数据量较低就忽视多线程的应用，多线程在你身边的各类中间件和互联网大厂中都有着极为广泛的应用。

应用多线程时的风险提示

虽然多线程有很多的好处，但仍然要根据场景客观分析，对多线程不合理的使用会增加系统的**安全风险**和**维护风险**。所以，在使用多线程时，请务必确认**场景的合理性**，以及它在你技术能力掌控之中。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

- 使用不同的构造方式，编写两个线程并打印出线程的关键信息；
- 检索资料，详细比对进程与线程的区别。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜03：兴利除弊-如何理解多线程带来的安全问题

欢迎来到《**王者并发课**》，本文是该系列文章中的**第3篇**。

在前面的两篇文章中，我们体验了线程的创建，并从OS进程层面认识了线程。现在，我们已经知晓多线程在解决一些场景问题时有特效。

然而，不知你可曾想过，多线程虽然效率很高，但是它却有着你**无法回避**的**并发问题**。举个王者中常见的场景，双方10人同时进攻主宰，最后击败主宰的玩家才是真正的赢家，而且只能有一位。所以问题来了，**假如这10位玩家代表10个线程，它们在并发访问同一个资源时，如何保证数据的安全性**？总不至于，主宰只有一条命，可是却有多位玩家获得主宰，这显然不符合逻辑。

这个简单例子的背后，是计算机系统中一个普遍且基本的问题，即**多线程的安全问题**。在设计多线程时，我们追求它的优点，也务必要理解它存在的安全隐患，并为之设计合理的解决方案。否则，多线程这把双刃剑必将给我们以教训。

本文将从**并发**与**并行**的概念出发，帮助你**入门**这些概念并**理解竞态**相关问题。

一、理解并发（Concurrency）和并行（Parallelism）

在并发编程中，并发与并行像一对孪生兄弟，不仅长相相似，又容易让人混淆。但是，它们又有着本质的区别。所以，理解并行与并发，不要尝试去死记硬背概念，在你未能从本质上认识它们之前，你无法欺骗你的大脑去记住它。

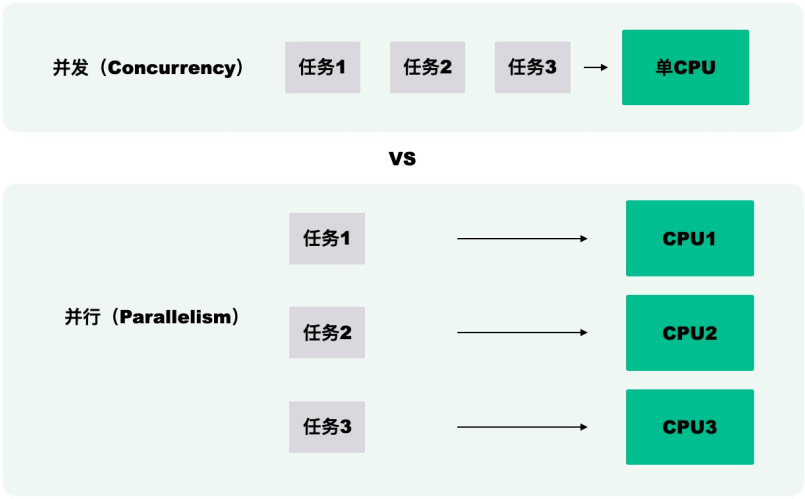
简而言之，**并行与并发的区别的核心在于所竞争的 *资源* 不同**。举个通俗的例子：

- 蓝方5个人一起打**主宰**，是**并发（Concurrency）**，因为竞争的目标资源只有**一个**；

- 蓝方2人去打**主宰**，3人去打**暴君**，是**并行（Parallelism）**，因为竞争的目标资源是**两个**。

类似的，从CPU计算的角度看， 并发和并行的概念可以理解为：

- 如果1个CPU同时执行5个任务，就是**并发**；
- 如果5个CPU同时执行5个任务，并且是每个CPU执行一个，那么就是**并行**。



以上是对并行和并发的通俗概述，如果你有兴趣，可以通过检索资料详细了解**单CPU**下是如何**模拟**并发的。

二、理解竞态（Race Condition）下的安全问题

显然易见，无论是并发还是并行，都有助于**提高计算效率**。然而，效率是一方面，**安全则是更重要的一方面**。比如上面进攻主宰的案例中，一定要能知道是谁给予了最后一击，也就是**数据不能出错**。所以，我们就需要理解多线程下的 **竞态（Race Condition）** 和解决策略。

所谓竞态，你可以理解为多个线程试图在同一时刻修改共享数据的情况。你看，从字面上理解的话，**Race**这个词就是**比赛**的意思。比赛的目标是什么？是看谁先获得共享资源，即进入**临界区（Critical Section）** 。

常见的竞态有下面这两种模式：

- **Read-modify-write**
- **Check-then-act**

1. Read-modify-write

先看下面这段代码，玩家每次进攻，主宰的血量都会减少：

```
public class Master {
    //主宰的初始血量
    private int blood = 100;

    //每次被击打后血量减5
    public int decreaseBlood() throws Exception {
        if(blood <= 0){
            throw new Exception("主宰已经被击败！");
        }
        blood = blood - 5;
        return blood;
    }
}
```

当线程执行`decreaseBlood()`方法调用时，事情是这样发展的：

- 第一步：从内存中读取`blood`的值到寄存器（**Read**）；
- 第二步：修改寄存器中的`blood`值（**Modify**）；
- 第三步：将寄存器的值写回内存（**Write**）。

这就是**Read-modify-write**模式。整个过程看起来一气呵成，实则祸根已经种下。想想看，如果在第一步时，两个线程同时都读取到了值（比如100），随后两个线程同时做了修改，此时在第三步，无论是哪个线程率先将值写回内存，后面的线程都会**覆盖**内存中的值。换句话说，主宰承受了两次攻击，血量应该降低到90，可结果却是95，不是它耐操，而是你代码写错了！

2. Check-then-act

```
//每次被击打后血量减5
    public int decreaseBlood() throws Exception {
        if(blood <= 0){
            throw new Exception("主宰已经被击败！");
        }
        blood = blood - 5;
        return blood;
    }
```

我们再近距离观察下`decreaseBlood()`方法，你会发现，它不仅会让主宰出现攻击两次但血量却只减少一次的情况，还会出现血量为**负值**的情况！这是为什么？

注意`decreaseBlood()`中有一行`if(blood <= 0)`，也就是说如果此时主宰已经被击败，那就不要再往下继续运行，直接抛出异常。但是，问题来了。假设此时主宰的血量是 *5* ，就差**最后一击**了！然后，线程A和线程B两个线程同时进来：

- 第一步：线程A和线程B检查血量是否为 *0* （**Check**）；
- 第二步：线程A和线程B都通过了检查；
- 第三步：线程A和线程B执行血量扣减动作，但顺序未知（**Act**）。

问题是，如果线程A在执行`blood = blood - 5`时，`blood`的值不再是 *5* ，而是已经被线程B更改为 *0* 了呢？那么结果就是主宰最后的血量是 *-5* ！很显然，这样的结果就扯淡了。

以上就是两种常见的竞态情况。简单来说， **Read-modify-write**是在写入时因并发导致值被**覆盖**，而**Check-then-act**则是因并发导致**条件判断失效**。

3. 如何预发竞态

既然多线程是不安全的，那如何预防竞态的发生？其核心在于**锁+原子操作**，即**对临界区进行加锁，让临界区每次有且只能有一个线程访问**，在当前线程未离开临界区时，其他线程不得进入，且线程在临界区的操作必须保证原子性。

在Java中，最简单的加锁方式是使用`synchronized`关键字，我们会在下一篇中对它详细讲解。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

- 写一段多线程并发代码，体验并发时的数据错误。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜04：宝刀屠龙-如何使用synchronized之初体验

欢迎来到《王者并发课》，本文是该系列文章中的**第4篇**。

在前面的文章《双刃剑-理解多线程带来的安全问题》中，我们提到了多线程情况下存在的线程安全问题。本文将以这个问题为背景，介绍如何通过使用`synchronized`关键字解这一问题。当然，在青铜阶段，我们仍不会过多地描述其背后的原理，**重点还是先体验并理解它的用法**。

一、从场景中体验synchronized

是谁击败了主宰

在峡谷中，击败主宰可以获得高额的经济收益。因此，在条件允许的情况下，大家都会争相击败主宰。于是，哪吒和敌方的兰陵王开始争夺主宰。按规矩，**谁是击败主宰的最后一击，谁便是胜利的一方**。

假设主宰的初始血量是100，我们通过代码来模拟下：

```
public class Master {
    //主宰的初始血量
    private int blood = 100;

    //每次被击打后血量减5
    public int decreaseBlood() {
        blood = blood - 5;
        return blood;
    }

    //通过血量判断主宰是否还存活
    public boolean isAlive() {
        return blood > 0;
    }
```

```
}
```

我们定义了哪吒和兰陵王两个线程，让他们同时攻击主宰：

```
public static void main(String[] args) {
        final Master master = new Master();
        Thread neZhaAttachThread = new Thread() {
            public void run() {
                while (master.isAlive()) {
                    try {
                        int remainBlood = master.decreaseBlood();
                        if (remainBlood == 0) {
                            System.out.println("哪吒击败了主宰！");
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        Thread lanLingWangThread = new Thread() {
            public void run() {
                while (master.isAlive()) {
                    try {
                        int remainBlood = master.decreaseBlood();
                        if (remainBlood == 0) {
                            System.out.println("兰陵王击败了主宰！");
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        neZhaAttachThread.start();
        lanLingWangThread.start();
    }
```

下面是运行的结果：

```
兰陵王击败了主宰!
哪吒击败了主宰!
```

```
Process finished with exit code 0
```

两人竟然都获得了主宰！很显然，我们不可能接受这样的结果。然而，细看代码，你会发现这个神奇的结果其实一点也不意外，两个线程在对`blood`做并发减法时出了错误，因为代码中压根没有必要的并发安全控制。

当然，解决办法也比较简单，在`decreaseBlood`方法上添加`synchronized`关键字即可：

```
public synchronized int decreaseBlood() {
        blood = blood - 5;
        return blood;
}
```

为什么加上`synchronized`关键字就可以了呢？这就需要往下看了解Java中的**锁**和**同步**了。

二、认识synchronized

1. 理解Java对象中的锁

在理解`synchronized`之前，我们先简单理解下**锁**的概念。在Java中，每个对象都会有一把锁。当多个线程都需要访问对象时，那么就需要通过获得锁来获得许可，只有获得锁的线程才能访问对象，并且其他线程将进入等待状态，等待其他线程释放锁。如下图所示：

```
public synchronized int decreaseBlood() throws Exception {
      if(blood<=0){
          throw new Exception("主宰已经被击败！");
      }
      blood = blood - 5;
      return blood;
  }
```

synchronized method

2. 理解synchronized关键字

根据Sun官文文档的描述，`synchronized`关键字提供了一种预防**线程干扰**和**内存一致性错误**的简单策略，即如果一个对象对多个线程可见，那么该对象变量（`final`修饰的除外）的读写都需要通过`synchronized`来完成。

你可能已经注意到其中的两个关键名词：

- **线程干扰（Thread Interference）**：不同线程中运行但作用于相同数据的两个操作交错时，就会发生干扰。这意味着这两个操作由多个步骤组成，并且步骤顺序重叠；
- **内存一致性错误（Memory Consistency Errors）**：当不同的线程对应为相同数据的视图不一致时，将发生内存一致性错误。内存一致性错误的原因很复杂，幸运的是，我们不需要详细了解这些原因，所需要的只是避免它们的策略。

从竞态的角度讲，线程干扰对应的是**Read-modify-write**，而内存一致性错误对应的则是**Check-then-act**。

结合**锁**和**synchronized**的概念可以理解为，锁是多线程安全的基础机制，而**synchronized**是锁机制的一种实现。

三、synchronized的四种用法

1. 在实例方法中使用synchronized

```
public synchronized int decreaseBlood() {
        blood = blood - 5;
        return blood;
}
```

注意这段代码中的`synchronized`字段，它表示**当前方法每次能且仅能有一个线程访问**。另外，由于当前方法是实例方法，所以如果该对象存在多个实例的话，不同的实例可以由不同的线程访问，它们之间并无协作关系。

然而，你可能已经想到了，如果当前线程中有两个`synchronized`方法，不同的线程是否可以访问不同的`synchronized`方法呢?

答案是：**不能**。

这是因为每个**实例内的同步方法，能且仅能有一个线程访问**。

2. 在静态方法中使用synchronized

```
public static synchronized int decreaseBlood() {
        blood = blood - 5;
        return blood;
}
```

与实例方法的`synchronized`不同，静态方法的`synchronized`是基于当前方法所属的类，即`Master.class`，而每个类在虚拟机上有且只有一个类对象。所以，对于同一类而言，每次有且只能有一个线程能访问静态`synchronized`方法。

当类中包含有多个静态的`synchronized`方法时，每次也仍然有且只能有一个线程可以访问其中的方法。

注意： 从`synchronized`在实例方法和静态方法中的应用可以看

出，`synchronized`方法是否能允许其他线程的进入，取决于`synchronized`的参数。每个不同的参数，在同一时刻都只允许一个线程访问。基于这样的认知，下面的两种用法就很容易理解了。

3. 在实例方法的代码块中使用synchronized

```
public int decreaseBlood() {
    synchronized(this) {
       blood = blood - 5;
       return blood;
    }
}
```

在某些情况下，你不需要在整个方法层面使用`synchronized`，毕竟这样的方式粒度较大，容易产生阻塞。此时，在代码块中使用`synchronized`就是非常不错的选择，如上面代码所示。

刚才已经提到，`synchronized`的并发限制取决于其参数，在上面这段代码中的参数是`this`，即当前类的实例对象。而在前面的`public synchronized int decreaseBlood()`中，`synchronized`的参数也是当前类的实例对象。因此，下面这两段代码是等同的：

```
public int decreaseBlood() {
    synchronized(this) {
       blood = blood - 5;
       return blood;
    }
}

public synchronized int decreaseBlood() {
       blood = blood - 5;
       return blood;
}
```

4. 在静态方法的代码块中使用synchronized

同理，下面这两个方法的效果也是等同的。

```
public static int decreaseBlood() {
    synchronized(Master.class) {
       blood = blood - 5;
       return blood;
    }
}

public static synchronized int decreaseBlood() {
       blood = blood - 5;
       return blood;
}
```

四、synchronized小结

前面，我们已经介绍了`synchronized`的几种常见用法，不必死记硬背，你只要记住`synchronized`可以接受任何**非null**对象作为参数，而每个参数在同一时刻能且只能允许一个线程访问即可。此外，还有一些具有实际指导意义的Tips你可以注意下：

1. Java中的`synchronized`关键字用于解决多线程访问共享资源时的同步，以解决**线程干扰**和**内存一致性**问题；
2. 你可以通过 **代码块（code block）** 或者 **方法（method）** 来使用`synchronized`关键字；
3. `synchronized`的原理基于**对象中的锁**，当线程需要进入`synchronized`修饰的方法或代码块时，它需要先**获得**锁并在执行结束后**释放**它；
4. 当线程进入**非静态（non-static）同步方法时，它获得的是对象实例（Object level）的锁。而线程进入静态**同步方法时，它所获得的是类实例（Class level）的锁，两者没有必然关系；
5. 如果`synchronized`中使用的对象是**null**，将会抛出`NullPointerException`错误；
6. `synchronized`**对方法的性能有一定影响**，因为线程要等待获取锁；
7. 使用`synchronized`时**尽量使用代码块**，而不是整个方法，以免阻塞整个方法；

8. **尽量不要使用*String*类型和*原始类型*作为参数**。这是因为，JVM在处理字符串、原始类型时会对它们进行优化。比如，你原本是想对不同的字符串进行加锁，然而JVM认为它们是同一个，很显然这不是你想要的结果。

关于`synchronized`的可见性、指令排序等底层原理，我们会在后面的阶段中详细介绍。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

夫子的试炼

- 手写代码体验`synchronized`的不同用法。

延伸阅读与参考资料

- https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
- https://javagoal.com/synchronization-in-java/
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

青铜05：一探究竟-如何从synchronized理解Java对象头中的锁

欢迎来到《王者并发课》，本文是该系列文章中的**第5篇**。

在前面的文章《青铜4：synchronized用法初体验》中，我们已经提到**锁**的概念，并指出`synchronized`是锁机制的一种实现。可是，这么说未免太过抽象，你可能无法直观地理解**锁究竟是什么**？所以，本文会粗略地介绍`synchronized`背后的一些基本原理，让你对Java中的锁有个粗略但直观的印象。

本文将分两个部分，首先你要从Mark Word中认识锁，因为对象锁的信息存在于Mark Word中，其次通过JOL工具实际体验Mark Word的变化。

一、从Mark Word认识锁

我们知道，在HotSpot虚拟机中，一个对象的存储分布由3个部分组成：

- **对象头（Header）**：由**Mark Word**和**Klass Pointer**组成；
- **实例数据（Instance Data）**：对象的成员变量及数据；
- **对齐填充（Padding）**：对齐填充的字节，暂时不必理会。

在这3个部分中，对象头中的**Mark Word**是本文的重点，也是理解Java锁的关键。Mark Word记录的是对象运行时的数据，其中包括：

- **哈希码（identity_hashcode）**
- **GC分代年龄（age）**
- **锁状态标志**
- **线程持有的锁**
- **偏向线程ID（thread）**

所以，从对象头中的Mark Word看，**Java中的锁就是对象头中的一种数据**。在JVM中，**每个对象都有这样的锁**，并且用于多线程访问对象时的并发控制。

如果一个线程想访问某个对象的实例，那么这个线程必须拥有该对象的**锁**。首先，它需要通过对象头中的Mark Word判断该对象的实例是否已经被线程锁定。如果没有锁定，**那么线程会在Mark Word中写入一些标记数据**，就是告诉别人：这个对象是我的啦！如果其他线程想访问这个实例的话，就需要进入等待队列，直到当前的线程释放对象的锁，也就是把Mark Word中的数据擦除。

当一个线程拥有了锁之后，它便可以多次进入。当然，在这个线程释放锁的时候，那么也需要执行相同次数的释放动作。比如，一个线程先后3次获得了锁，那么它也需要释放3次，其他线程才可以继续访问。

下面的表格展示的是64位计算机中的对象头信息：

```
|--------------------------------------------------------------------
----------------------------------------|--------------------
|
|                                                Object Header (128
bits)                                          |        State
|
|--------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
|                                    Mark Word (64 bits)
|    Klass Word (64 bits)     |                     |
|--------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
| unused:25 | identity_hashcode:31 | unused:1 | age:4 |
biased_lock:1 | lock:2 |    OOP to metadata object   |
Normal       |
|--------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
| thread:54 |       epoch:2        | unused:1 | age:4 |
biased_lock:1 | lock:2 |    OOP to metadata object   |
Biased       |
|--------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
```

```
|                       ptr_to_lock_record:62
| lock:2 |    OOP to metadata object    | Lightweight Locked |
|-------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
|                     ptr_to_heavyweight_monitor:62
| lock:2 |    OOP to metadata object    | Heavyweight Locked |
|-------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
|
| lock:2 |    OOP to metadata object    |    Marked for GC   |
|-------------------------------------------------------------------
--------------|-----------------------------|--------------------
|
```

从表格中，你可以看到Object Header中的三部分信息：Mark Word、Klass Word、State.

二、通过JOL体验Mark Word的变化

为了直观感受对象头中Mark Word的变化，我们可以通过 **JOL（Java Object Layout）** 工具演示一遍。JOL是一个不错的Java内存布局查看工具，希望你能记住它。

首先，在工程中引入依赖：

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.10</version>
</dependency>
```

在下面的代码中，`master`是我们创建的对象实例，方法`decreaseBlood()`中会执行加锁动作。所以，在调用`decreaseBlood()`加锁后，**对象头信息应该会发生变化**。

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

```
        Master master = new Master();
        System.out.println("====加锁前====");

System.out.println(ClassLayout.parseInstance(master).toPrintable(
));
        System.out.println("====加锁后====");
        synchronized (master) {

System.out.println(ClassLayout.parseInstance(master).toPrintable(
));
        }
    }
```

结果输出如下:

```
====加锁前====
cn.tao.king.juc.execises1.Master object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
VALUE
      0     4        (object header)                           01
00 00 00 (00000001 00000000 00000000 00000000) (1)
      4     4        (object header)                           00
00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           43
c1 00 f8 (01000011 11000001 00000000 11111000) (-134168253)
     12     4    int Master.blood
100
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

====加锁后====
cn.tao.king.juc.execises1.Master object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
VALUE
      0     4        (object header)                           48
f9 d6 00 (01001000 11111001 11010110 00000000) (14088520)
      4     4        (object header)                           00
70 00 00 (00000000 01110000 00000000 00000000) (28672)
      8     4        (object header)                           43
c1 00 f8 (01000011 11000001 00000000 11111000) (-134168253)
     12     4    int Master.blood                              95
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

```
Process finished with exit code 0
```

从结果中可以看到，代码在执行`synchronized`方法后，所打印出的`object header`信息由`01 00 00 00`、`00 00 00 00`变成了`48 f9 d6 00`、`00 70 00 00`等等，不出意外的话，相信你应该看不明白这些内容的含义。

所以，为了方便阅读，我们在青铜系列文章《**借花献佛-JOL格式化工具**》中提供了一个工具类，让输出更具可读性。借助工具类，我们把代码调整为：

```
 public static void main(String[] args) {
        Master master = new Master();
        System.out.println("====加锁前====");
        printObjectHeader(master);
        System.out.println("====加锁后====");
        synchronized (master) {
            printObjectHeader(master);
        }
    }
```

输出的结果如下：

```
====加锁前====
# WARNING: Unable to attach Serviceability Agent. You can try
again with escalated privileges. Two options: a) use -
Djol.tryWithSudo=true to try with sudo; b) echo 0 | sudo tee
/proc/sys/kernel/yama/ptrace_scope
Class Pointer: 11111000 00000000 11000001 01000011
Mark Word:
  hashcode (31bit): 0000000 00000000 00000000 00000000
  age (4bit): 0000
  biasedLockFlag (1bit): 0
  LockFlag (2bit): 01

====加锁后====
Class Pointer: 11111000 00000000 11000001 01000011
Mark Word:
  javaThread*(62bit,include zero padding): 00000000 00000000
01110000 00000000 00000100 11100100 11101001 100100
  LockFlag (2bit): 00
```

你看，这样一来，输出的结果的结果就一目了然。从**加锁后**的结果中可以看到，Mark Word已经发生变化，当前线程已经获得对象的锁。

至此，你应该明白，原来synchronized的背后的原理是这么回事。当然，本文所讲述只是其中的部分。出于篇幅考虑和难度控制，**本文暂且不会对Java对象头中锁的含义和锁的升级等问题展开描述**，这部分内容会在后面的文章中详细介绍。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

- 下载JOL工具，在代码中体验工具的使用和对象信息的变化。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜06：借花献佛-如何格式化Java内存工具JOL输出

欢迎来到《王者并发课》，本文是该系列文章中的**第6篇**。

在前面的文章《一探究竟-如何从synchronized理解Java对象头中的锁》中，我们介绍并体验了JOL工具。虽然JOL很赞，但它的输出对我们不是很友好，如果不借助工具，我们很难直观理解其中的含义。

下面这段代码是对JOL输出的翻译，建议你收藏。**代码非我所写，文末已经注明出处**。

```
import org.openjdk.jol.info.ClassLayout;

import java.nio.ByteOrder;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PrintObjectHeader {
    /**
     * Get binary data
     *
     * @param o
     * @return
     */
    public static String getObjectHeader(Object o) {
        ByteOrder order = ByteOrder.nativeOrder();//Byte order
        String table =
ClassLayout.parseInstance(o).toPrintable();
        Pattern p = Pattern.compile("(0|1){8}");
        Matcher matcher = p.matcher(table);
        List<String> header = new ArrayList<String>();
        while (matcher.find()) {
```

```
            header.add(matcher.group());
        }
        //Little-endian machines, need to traverse in reverse
        StringBuilder sb = new StringBuilder();
        if (order.equals(ByteOrder.LITTLE_ENDIAN)) {
            Collections.reverse(header);
        }
        for (String s : header) {
            sb.append(s).append(" ");
        }
        return sb.toString().trim();
    }

    /**
     * Parsing object header function for 64bit jvm
     * In 64bit jvm, the object header has two parts: Mark Word
and Class Pointer, Mark Word takes 8 bytes, Class Pointer takes 4
bytes
     *
     * @param s Binary string of object header (each 8 bits,
separated by a space)
     */
    public static void parseObjectHeader(String s) {
        String[] tmp = s.split(" ");
        System.out.print("Class Pointer: ");
        for (int i = 0; i < 4; ++i) {
            System.out.print(tmp[i] + " ");
        }
        System.out.println("\nMark Word:");
        if (tmp[11].charAt(5) == '0' &&
tmp[11].substring(6).equals("01")) {//0 01 lock-free state,
regardless of GC mark
            //notice: Mark word structure without lock:
unused(25bit) + hashcode(31bit) + unused(1bit) + age(4bit) +
biased_lock_flag(1bit) + lock_type(2bit)
            // The reason why hashcode only needs 31bit is:
hashcode can only be greater than or equal to 0, eliminating the
negative range, so you can use 31bit to store
            System.out.print("\thashcode (31bit): ");
            System.out.print(tmp[7].substring(1) + " ");
            for (int i = 8; i < 11; ++i) System.out.print(tmp[i]
+ " ");
            System.out.println();
        } else if (tmp[11].charAt(5) == '1' &&
tmp[11].substring(6).equals("01")) {//1 01, which is the case of
```

```
biased lock
            //notice: The object is in a biased lock, its
structure is: ThreadID(54bit) + epoch(2bit) + unused(1bit) +
age(4bit) + biased_lock_flag(1bit) + lock_type(2bit)
            // ThreadID here is the thread ID holding the biased
lock, epoch: a timestamp of the biased lock, used for
optimization of the biased lock
            System.out.print("\tThreadID(54bit): ");
            for (int i = 4; i < 10; ++i) System.out.print(tmp[i]
+ " ");
            System.out.println(tmp[10].substring(0, 6));
            System.out.println("\tepoch: " +
tmp[10].substring(6));
        } else {//In the case of lightweight locks or heavyweight
locks, regardless of the GC mark
            //notice: JavaThread*(62bit,include zero padding) +
lock_type(2bit)
            // At this point, JavaThread* points to the monitor
of the lock record/heavyweight lock in the stack
            System.out.print("\tjavaThread*(62bit,include zero
padding): ");
            for (int i = 4; i < 11; ++i) System.out.print(tmp[i]
+ " ");
            System.out.println(tmp[11].substring(0, 6));
            System.out.println("\tLockFlag (2bit): " +
tmp[11].substring(6));
            System.out.println();
            return;
        }
        System.out.println("\tage (4bit): " +
tmp[11].substring(1, 5));
        System.out.println("\tbiasedLockFlag (1bit): " +
tmp[11].charAt(5));
        System.out.println("\tLockFlag (2bit): " +
tmp[11].substring(6));

        System.out.println();
    }

    public static void printObjectHeader(Object o) {
        if (o == null) {
            System.out.println("null object.");
            return;
        }
        parseObjectHeader(getObjectHeader(o));
```

```
    }
}
```

延伸阅读与参考资料

- https://www.programmersought.com/article/21094532407/
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

青铜07：顺藤摸瓜-如何从synchronized中的锁认识Monitor

欢迎来到《王者并发课》，本文是该系列文章中的**第7篇**。

在前面的文章中，我们已经体验过synchronized的用法，并对锁的概念和原理做了简单的介绍。然而，你可能已经察觉到，**有一个概念**似乎总是和**synchronized**、**锁**这两个概念**如影相随**，很多人也比较喜欢问它们之间的区别，这个概念就是**Monitor**，也叫**监视器**。

所以，在讲解完synchronized、锁之后，文本将为你讲解Monitor，揭示它们之间那些公开的秘密，希望你不再迷惑。

首先，你要明白的是，**Monitor作为一种同步机制**，它并非Java所特有，但Java实现了这一机制。

为了具象地理解Monitor这一抽象概念，我们先来分析身边的一个常见场景。

一、从医院排队就诊机制理解Monitor

相信你一定有过去医院就诊的经历。我们去医院时，情况一般是这样的：

- 首先，我们在**门诊大厅**前台或自助挂号机**进行挂号**；
- 随后，挂号结束后我们找到对应的**诊室就诊**：
  - 诊室每次只能有一个患者就诊；
  - 如果此时诊室空闲，直接进入就诊；
  - 如果此时诊室内有患者正在就诊，那么我们进入**候诊室**，等待叫号；
- 就诊结束后，**走出就诊室**，候诊室的**下一位候诊患者**进入就诊室。

这个就诊过程你一定耳熟能详，理解起来必然毫不费力气。我们做了一张图展示图下：

仔细看这幅图中的就诊过程，**如果你理解了这个过程，你就理解了Monitor**. 这么简单吗？不要怀疑自己，是的。**你竟然早已理解Monitor机制**！

不要小看这个机制，它可是生活中的智慧体现。在这个就诊机制中，它起到了**两个关键性的作用**：

- **互斥（mutual exclusion ）**： 每次只允许一个患者进入候诊室就诊；
- **协作（cooperation）**： 就诊室中的患者就诊结束后，可以通知候诊区的下一位患者。

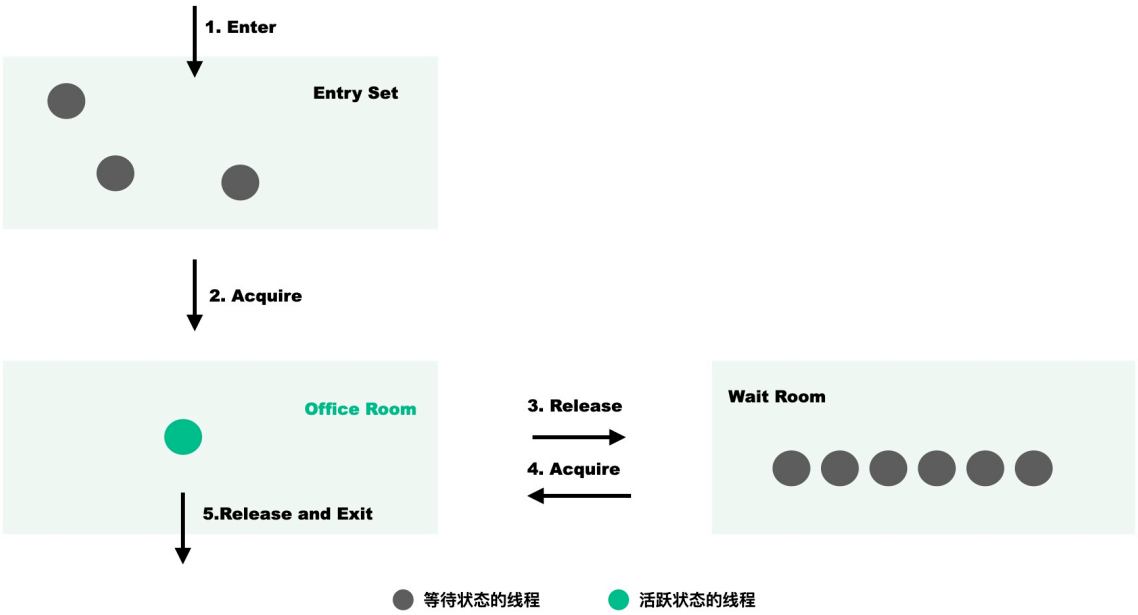
明白了吗？你在医院就诊的过程竟然和Monitor的机制几乎一模一样。我们换个方式来描述Monitor在计算机科学中的作用：

- **互斥（mutual exclusion ）**： 每次只允许一个线程进入临界区；
- **协作（cooperation）**： 当临界区的线程执行结束后满足特定条件时，可以通知其他的等待线程进入。

而就诊过程中的**门诊大厅**、**就诊室**、**候诊室**则恰好对应着Monitor中的三个关键概念。其中：

- **门诊大厅**：所有待进入的线程都必须先在**入口（Entry Set）**挂号才有资格；
- **就诊室**：一个每次只能有一个线程进入的**特殊房间（Special Room）**；
- **候诊室**：就诊室繁忙时，进入**等待区（Wait Set）**；

我们把上面的图稍作调整，就可以看到Monitor在Java中的模样：



对比来看，相信你已经很直观地理解Monitor机制。再一回味，你会发现**synchronized正是对Monitor机制的一种实现**。而在Java中，**每一个对象都会关联一个监视器**。

二、从synchronized源码感受Monitor

既然synchronized是对Monitor机制的一种实现，为了让你更有体感，我们可以写一段**极简代码**一探究竟。

这段代码极为简单，但是够用，我们在代码中使用了`synchronized`关键字：

```
public class SyncMonitorDemo {
    public static void main(String[] args) {
        Object o = new Object();
        synchronized (o) {
            System.out.println("locking...");
        }
    }
}
```

代码写好后，分别执行`javac SyncMonitorDemo.java`和`javap -v`

`SyncMonitorDemo.class`，随后你就能得到下面这样的字节码：

```
Classfile SyncMonitorDemo.class
  Last modified May 26, 2021; size 684 bytes
  MD5 checksum e366920f22845e98c45f26531596d6cf
  Compiled from "SyncMonitorDemo.java"
public class cn.tao.king.juc.execises1.SyncMonitorDemo
  minor version: 0
  major version: 49
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #2.#22         // java/lang/Object."
<init>":()V
   #2 = Class              #23            // java/lang/Object
   #3 = Fieldref           #24.#25        //
java/lang/System.out:Ljava/io/PrintStream;
   #4 = String             #26            // locking...
   #5 = Methodref          #27.#28        //
java/io/PrintStream.println:(Ljava/lang/String;)V
   #6 = Class              #29            //
cn/tao/king/juc/execises1/SyncMonitorDemo
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8
Lcn/tao/king/juc/execises1/SyncMonitorDemo;
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
  #16 = Utf8               args
  #17 = Utf8               [Ljava/lang/String;
  #18 = Utf8               o
  #19 = Utf8               Ljava/lang/Object;
  #20 = Utf8               SourceFile
  #21 = Utf8               SyncMonitorDemo.java
  #22 = NameAndType        #7:#8          // "<init>":()V
  #23 = Utf8               java/lang/Object
  #24 = Class              #30            // java/lang/System
  #25 = NameAndType        #31:#32        //
out:Ljava/io/PrintStream;
  #26 = Utf8               locking...
  #27 = Class              #33            // java/io/PrintStream
```

```
  #28 = NameAndType        #34:#35        // println:
(Ljava/lang/String;)V
  #29 = Utf8
cn/tao/king/juc/execises1/SyncMonitorDemo
  #30 = Utf8               java/lang/System
  #31 = Utf8               out
  #32 = Utf8               Ljava/io/PrintStream;
  #33 = Utf8               java/io/PrintStream
  #34 = Utf8               println
  #35 = Utf8               (Ljava/lang/String;)V
{
  public cn.tao.king.juc.execises1.SyncMonitorDemo();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this
Lcn/tao/king/juc/execises1/SyncMonitorDemo;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=4, args_size=1
         0: new           #2                  // class
java/lang/Object
         3: dup
         4: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         7: astore_1
         8: aload_1
         9: dup
        10: astore_2
        11: monitorenter
        12: getstatic     #3                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        15: ldc           #4                  // String
```

```
locking...
        17: invokevirtual #5                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
        20: aload_2
        21: monitorexit
        22: goto          30
        25: astore_3
        26: aload_2
        27: monitorexit
        28: aload_3
        29: athrow
        30: return
      Exception table:
         from    to  target type
            12    22    25   any
            25    28    25   any
      LineNumberTable:
        line 5: 0
        line 6: 8
        line 7: 12
        line 8: 20
        line 9: 30
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      31     0  args   [Ljava/lang/String;
            8      23     1     o   Ljava/lang/Object;
}
SourceFile: "SyncMonitorDemo.java"
```

javap是JDK自带的一个反汇编命令。你可以忽略其他不必要的信息，直接在结果中找到下面这段代码：

```
11: monitorenter
12: getstatic     #3                  // Field
java/lang/System.out:Ljava/io/PrintStream;
15: ldc           #4                  // String locking...
17: invokevirtual #5                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
20: aload_2
21: monitorexit
```

看到**monitorenter**和**monitorexit**指令，相信智慧的你已经看穿一切。

以上就是文本的全部内容，恭喜你又上了一颗星✨

夫子的试炼

- 写一段包含`synchronized`关键字的代码，使用`javap`命令观察结果。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜08：分工协作-如何通俗理解线程的状态和协作

欢迎来到《**王者并发课**》，本文是该系列文章中的**第8篇**。

在本篇文章中，**我将从多线程的本质出发，为你介绍线程相关的状态和它们的变迁方式，并帮助你掌握这块知识点**。

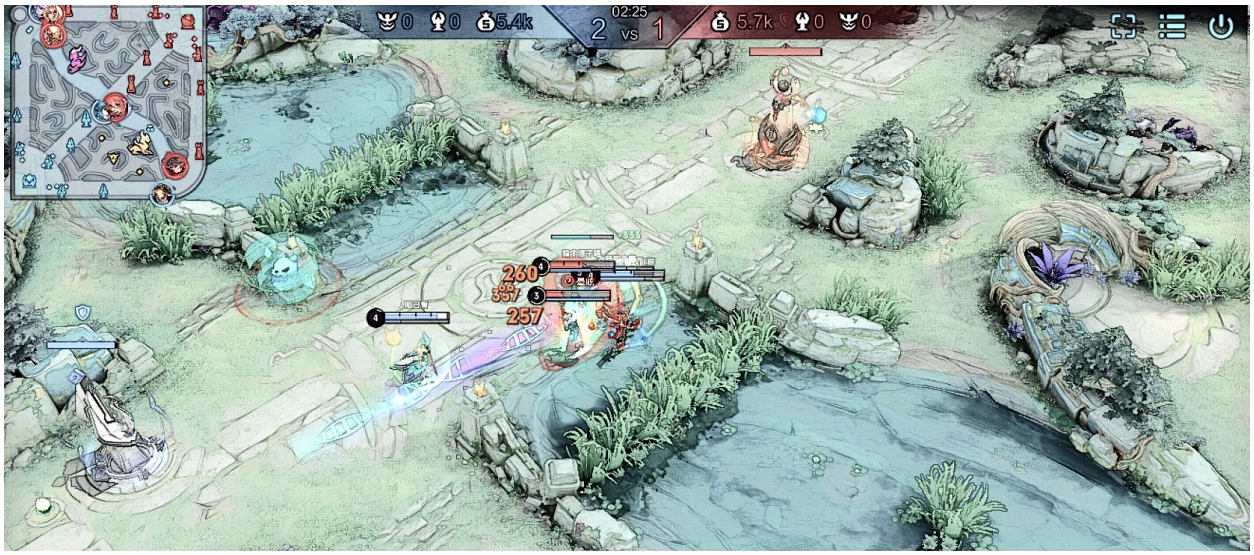
一、多线程的本质是分工协作

如果你是王者的玩家，那么你一定知道王者中的众多英雄分为主要分为几类，比如**法师**、**战士**、**坦克**、**辅助**等等。一些玩家对这些分类可能并不了解，甚至会觉得，干嘛要搞得这么复杂，干不完了嘛。**这...当然不可以**!

抱此想法的如果不是**青铜玩家**，想必就是战场上的那些**个人英雄主义**玩家，在他们眼里没有团队。然而，只有王者知道，比赛胜利的关键，在于**团队的分工协作**。**各自为战**必将**一团乱麻、溃不成军**，正所谓**单丝不成线，独木难成林**。

分工协作无处不在，峡谷中需要分工协作，现实中我们的工作更是社会化分工的结果，因为**社会的本质就是分工协作**。

而我要告诉你的是，在并发编程里，**多线程的本质也是分工协作**，每个线程恰似一个英雄，有着自己的职责、状态和技能（动作方法）。**所谓线程的状态、方法实现不过都是为了完成线程间的分工协作**。换句话说，线程状态的存在不是目的，而是实现分工协作的方式。所以，**理解线程的线程状态和驱动方法，首先要理解它们为什么而存在**。



二、从协作认知线程的状态

线程的状态是线程在协作过程中的**瞬时特征**。根据协作的需要，线程总共有六种状态，分别是**NEW**、**RUNNABLE**、**WAITING**、**TIMED_WAITING**、**BLOCKED**和**TERMINATED**等。比如，我们创建一个英雄哪吒的线程`neZhaPlayer`：

```
Thread neZhaPlayer = new Thread(new NeZhaRunnable());
```

那么，线程创建之后，接下来它将在下图所示的六种状态中变迁。刚创建的线程处于**NEW**的状态，而如果我们调用`neZhaPlayer.start()`，那它将会进入**RUNNABLE**状态。

六种不同状态的含义是这样的：

- **NEW**：线程新建但**尚未启动**时所处的状态，比如上面的`neZhaPlayer`；
- **RUNNABLE**：在 Java 虚拟机中执行的线程所处状态。需要注意的是，虽然线程当前正在被执行，但可能正在等待其他线程释放资源；
- **WAITING**：**无限期**等待**另一个线程**执行特定操作来解除自己的等待状态；
- **TIMED_WAITING**：**限时**等待另一个线程执行或自我解除等待状态；
- **BLOCKED**：**被阻塞**等待其他线程释放Monitor Lock；
- **TERMINATED**：线程执行结束。

在任意特定时刻，一个线程都只能处于上述六种状态中的一种。需要你注意的是**RUNNABLE**这个状态，它有些特殊。确切地说，它包含**READY**和**RUNNING**两个细分状态，下一章节的图示中有明确标示。

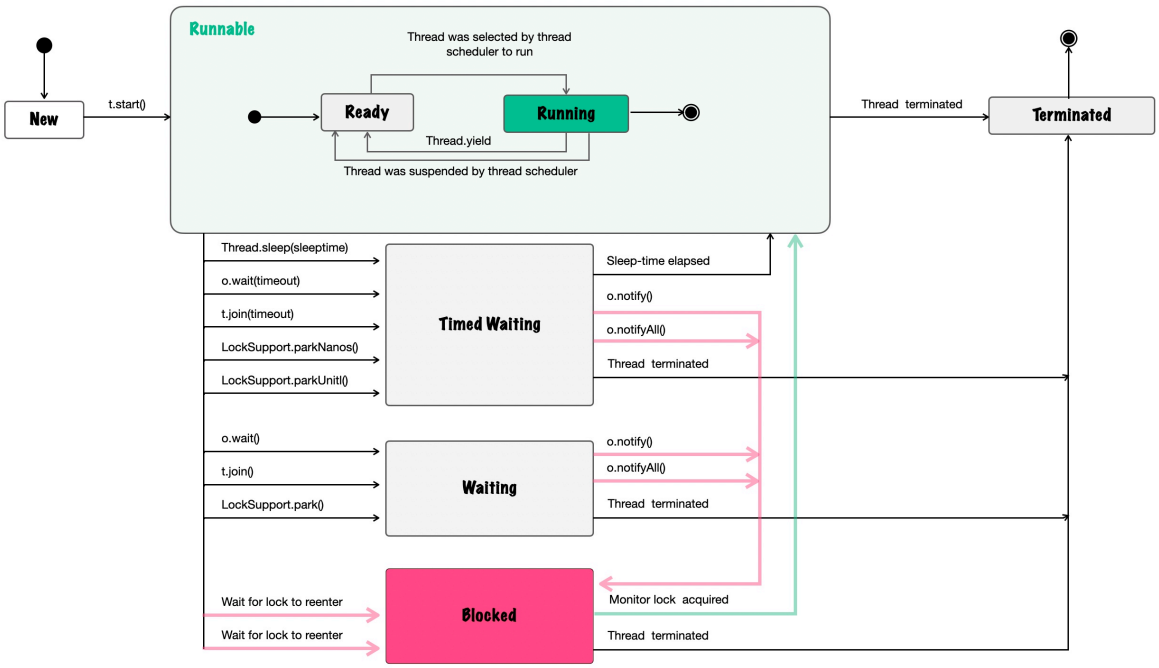
另外，前面我们已经介绍过**Thread**类，对于线程各状态的表述，你可以直接阅读JDK中的`Thread.State`枚举，并可以通过`Thread.getState()`查看当前线程的瞬时状态。

三、从线程状态变迁看背后的方法驱动

和人类的交流类似，在多线程的协作时，它们也需要交流。所以，线程**状态的变迁**就需要不同的方法来实现交流，比如刚创建的线程需要通过调用`start()`将线程状

态由**NEW**变迁为**RUNNABLE**。

下图所展示的正是线程间的状态变迁以及相关的驱动方法，你可以**先大概浏览一遍**，随后再结合下文的各关键方法的表述深入理解。



需要注意的是，本文不会详细介绍线程状态相关的所有方法，这既不现实也毫无必要。**上面这幅宝藏图示是理解本文所述知识的核心，下面所介绍的几个主要方法也并非为了你记忆，而是为了让你更好理解上面这幅图。**

在你理解了这幅宝图之后，你便可以完全自行去了解其他更多的方法。

1. start：对战开始，敌军还有5秒到达战场

```
public class NeZhaRunnable implements Runnable {
    public void run() {
        System.out.println("我是哪吒，我去上路");
    }
}

Thread neZhaPlayer = new Thread(new NeZhaRunnable());
neZhaPlayer.start();
```

`start()`方法主要将完成线程状态从**NEW**到**RUNNABLE**的变迁，这里有两个点：

- 创建新的线程；
- 由新的线程执行其中的`run()`方法。

需要注意的是，**你不可以重复调用`start()`方法**，否则会抛出`IllegalThreadStateException`异常。

2. wait和notify：我在等你，好了请告诉我

哪吒每次在使用完大招后，都需要经历几十秒的冷却时间才可以再次使用，接下来我们通过代码片段来模拟这个过程。

我们先定义一个`Player`类，这个类中包含了`fight()`和`refreshSkills()`两个方法，分别用于进攻和技能刷新，代码片段如下。

```
public class Player {
    public void fight() {
        System.out.println("大招未就绪，冷却中...");
        synchronized (this) {
            try {
                this.wait();
                System.out.println("大招已就绪，发起进攻！");
            } catch (InterruptedException e) {
                System.out.println("大招冷却被中断！");
            }
        }
    }
    public void refreshSkills() {
        System.out.println("技能刷新中...");
        synchronized (this) {
            this.notifyAll();
            System.out.println("技能已刷新！");
        }
    }
}
```

随后，我们写一段`main()`方法使用刚才创建的`Player`。注意，**这里我们创建了两个线程分别调用`Player`中的不同方法**。

```
public static void main(String[] args) throws
InterruptedException {
        final Player neZha = new Player();
        Thread neZhaFightThread = new Thread() {
            public void run() {
                neZha.fight();
            }
        };
        Thread skillRefreshThread = new Thread() {
            public void run() {
                neZha.refreshSkills();
            }
        };
        neZhaFightThread.start();
        skillRefreshThread.start();
    }
```

代码运行结果如下：

```
大招未就绪，冷却中...
技能刷新中...
技能已刷新！
大招已就绪，发起进攻！

Process finished with exit code 0
```

从运行的结果看，符合预期。相信你已经看到了，在上面的代码中我们使用了`wait()`和`notify()`两个函数。这两个线程是如何协作的呢？往下看。

首先，neZhaAttachThread调用了`neZha.fight()`这个方法。可是，**当哪吒想发起进攻的时候，竟然大招还没有冷却**！于是，这个线程不得不通过`wait()`方法进入**等待队列**。

紧接着，skillRefreshThread调用了`neZha.refreshSkills()`这个方法。并且，在执行结束后又调用了`notify()`方法。有趣的事情发生了，前面处于**等待队列**中的neZhaAttachThread竟然又“复活”了，并且大喊了一声：大招已经就绪，发起进攻！

这是怎么回事？理解这块逻辑，你需要了解以下几个知识点：

- `wait()`：看到`wait()`时，你可以简单粗暴地认为每个对象都有一个类似于休息室的**等待队列**，而`wait()`正是把当前线程送进了**等待队列**并暂停继续执行；
- `notify()`：如果说`wait()`是把当前线程送进了等待队列，那么`notify()`则是从等待队列中取出线程。此外，和`notify()`具有相似功能的还有个`notifyAll()`。与`notify()`不同的是，`notifyAll()`会取出等待队列中的所有线程；

看到这，你是不是觉得`wait()`和`notify()`简直是完美的一对？其实不然。真相不仅不完美，还很不靠谱！

`wait()`和`notify()`在执行时都必须先获得锁，这也是你在代码中看到`synchronized`的原因。`notify()`在释放锁的时候，会从等待队列中取出线程，**此时的线程必须获得锁之后才能继续运行**。那么，问题来了。**如果队列中有多个线程时，`notify()`能取出指定的线程吗？答案是不能！**

换句话说，如果队列中有多个线程，你将无法预料后续的执行结果！`notifyAll()`虽然可以取出所有的线程，但最终也只能有一个线程能获得锁。

是不是有点懵？懵就对了。所以你看，`wait()`和`notify()`是不是很不靠谱？因此，如果你需要在项目代码中使用它们，请务必要小心谨慎！

此外，如果你阅读过《Effective Java》，可以看到在这本书里作者Josh Bloch也是强烈建议不要随便使用这对组合。因为它们就像Java中的“汇编语言”，确实复杂且不容易控制，如果有相似的并发场景需要处理，可以考虑使用Java中的其他高级的并发工具。

3. interrupt：做完这一单，我就退隐江湖

在王者的游戏中，如果英雄血量没了，可以回城补血。回城大概需要5秒左右，如果在回城的过程中，突然被攻击或需要移位，那么回城就会中断。接下来，下面我们看看怎么模拟回城中的中断。

现在`Player`中定义`backHome()`方法用于回城。

```
 public void backHome() {
        System.out.println("回城中...");
        synchronized (this) {
            try {
                this.wait();
                System.out.println("已回城");
            } catch (InterruptedException e) {
                System.out.println("回城被中断! ");
            }
        }
    }
```

接下来启动新的线程调用`backHome()`回城补血。

```
public static void main(String[] args) throws
InterruptedException {
        final Player neZha = new Player();
        Thread neZhaBackHomeThread = new Thread() {
            public void run() {
                neZha.backHome();
            }
        };
        neZhaBackHomeThread.start();
        neZhaBackHomeThread.interrupt();
    }
```

运行结果如下:

```
回城中...
回城被中断!

Process finished with exit code 0
```

可以看到，回城被中断了，因为我们调用了`interrupt()`方法！那么，在线程中的**中断**是怎么回事？往下看。

在Thread中，我们可以通过`interrupt()`中断线程。然而，如果你细心的话，还会发现Thread中除了`interrupt()`方法之外，竟然还有两个长相酷似的方法：`interrupted()`和`isInterrupted()`。这就要小心了。

- `interrupt()`：将线程设置为中断状态；

- `interrupted()`：取消线程的中断状态；
- `isInterrupted()`：判断线程是否处于中断状态，而不会变更线程状态。

不得不说，`interrupt()`和`interrupted()`这两个方法的命名实在糟糕，你在编码时可不要学习它，**方法的名字应该清晰明了表达出其意图**。

那么，当我们调用`interrupt()`时，所调用对象的线程会立即抛出`InterruptedException`异常吗？**其实不然，这里容易产生误解**。

`interrupt()`方法只是改变了线程中的中断状态而已，并不会直接抛出中断异常。中断异常的抛出必须是当前线程在执行`wait()`、`sleep()`、`join()`时才会抛出。换句话说，**如果当前线程正在处理其他的逻辑运算，不会被中断，直到下次运行`wait()`、`sleep()`、`join()`时**！

4. join：稍等，等我结束你再开始

在前面的示例中，哪吒发起进攻和技能刷新两个线程是同时开始的。然而，我们在前面已经说了`wait()`和`notify()`并不靠谱，所以我们想在技能刷新结束后再执行后续动作。

```
public static void main(String[] args) throws
InterruptedException {
        final Player neZha = new Player();
        Thread neZhaFightThread = new Thread() {
            public void run() {
                neZha.fight();
            }
        };
        Thread skillRefreshThread = new Thread() {
            public void run() {
                neZha.refreshSkills();
            }
        };

        skillRefreshThread.start();
        skillRefreshThread.join(); //这里是重点
        neZhaFightThread.start();
    }
```

主线程调用`join()`时，会阻塞当前线程继续运行，直到目标线程中的任务执行完毕。此外，在调用`join()`方法时，也可以设置超时时间。

小结

以上就是关于线程状态及变迁的全部内容。在本文中，我们介绍了多线程的本质是协作，而状态和动作方法是实现协作的方式。无论是面试还是其他的资料中，**线程的状态**和**方法**都是重点。然而，我希望你明白了的是，**对于本文知识点的掌握，不要从静态的角度死记硬背，而是要动静结合，从动态的方法认知静态的状态**。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

在本文中，我们并没有提到`yield()`、`Thread.sleep()`和`Thread.current()`等方法。不过，如果你感兴趣的话，不妨检索资料：

- 了解`yield()`并对比它和`join()`的不同；
- 了解`wait()`并对比它和`Thread.sleep()`的不同；
- 了解`Thread.current()`的主要用法和它的实现。

延伸阅读与参考资料

- Life Cycle of a Thread in Java
- Enum Thread.State
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

青铜09：防患未然-如何应对线程中的异常

欢迎来到《王者并发课》，本文是该系列文章中的**第9篇**。

在本篇文章中，**我将为你介绍线程中异常的处理方式以及uncaughtExceptionHandler用法**。

一、新线程中的异常去哪了

应用程序在执行过程中，难免会出现各种意外错误，如果我们没有对错误进行捕获处理，会直接影响应用的运行结果，甚至导致应用崩溃。而在应用异常处理中，多线程的异常处理是比较重要又容易犯错的地方。

接下来，我们通过一段代码模拟一种常见的多线程异常处理方式。

在下面的代码中，我们在主线程中创建了新线程nezhaThread，并期望在主线程中捕获新线程中抛出的异常：

```
public static void main(String[] args) {
        Thread neZhaThread = new Thread() {
            public void run() {
                throw new RuntimeException("我是哪吒，我被围攻了！");
            }
        };
        // 尝试捕获线程抛出的异常
        try {
            neZhaThread.start();
        } catch (Exception e) {
            System.out.println("接收英雄异常：" + e.getMessage());
        }
    }
```

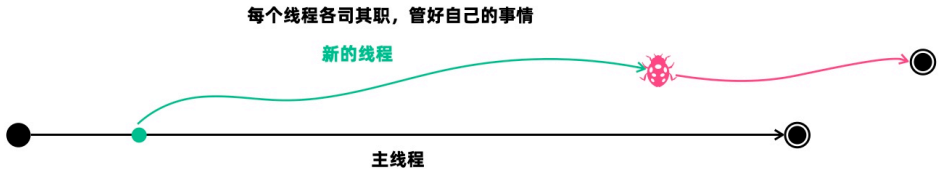
运行结果如下：

```
Exception in thread "Thread-0" java.lang.RuntimeException: 我是哪吒，我被围攻了!
  at cn.tao.king.juc.execises1.ExceptionDemo$1.run(ExceptionDemo.java:7)

Process finished with exit code 0
```

对于多线程新手来说，可能并不能直接看出其中的不合理。然而，从运行的结果中可以看到，没有输出“接收英雄异常”关键字。**也就是说，主线程并未能捕获新线程的异常。** 那这是为什么呢?

理解这一现象，首先要从线程的本质出发。在Java中，**每个线程所运行的都是独立运行的代码片段**，如果我们没有主动提供线程间通信和协作的机制，那么它们彼此之间是隔离的。

换句话说，**每个线程都要在自己的闭环内完成全部的任务处理，包括对异常的处理，如果出错了但你没有主动处理异常，那么它们会按照既定的流程自我了结。**



二、多线程中的异常处理方式

1. 从主线程看异常的处理

为了理解多线程中的错误处理方式，我们先看常见的主线程是如何处理错误的，毕竟相对于多线程，单一的主线程更容易让人理解。

```
 public static void main(String[] args) {
        throw new NullPointerException();
    }
```

很明显，上面这段代码将会抛出下面错误信息：

```
Exception in thread "main" java.lang.NullPointerException
  at
cn.tao.king.juc.execises1.ExceptionDemo.main(ExceptionDemo.java:2
1)
```

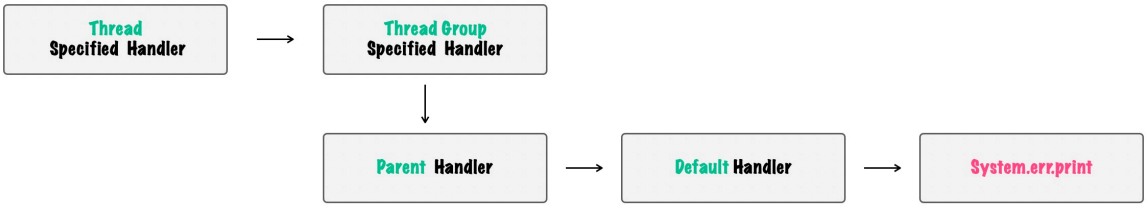
对于类似于空指针错误的堆栈信息，相信你一定并不陌生。在主线程中处理这样的异常很简单，通过编写`try`、`catch`代码块即可。但其实，除了这种方式外，我们还可以通过定义`uncaughtExceptionHandler`来处理主线程中的异常。

```
 public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(new
MyUncaughtExceptionHandler());
        throw new NullPointerException();
    }

    // 自定义错误处理
    static class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
        public void uncaughtException(Thread t, Throwable e) {
            System.out.println("出错了！线程名：" + t.getName() + ",
错误信息：" + e.getMessage());
        }
    }
```

输出结果如下：

```
出错了！线程名：main，错误信息：null

Process finished with exit code 1
```

你看，我们已经地捕获了异常。然而，你可能会疑惑为什么Thread.UncaughtExceptionHandler可以自定义错误处理？说到这，就不得不提Java中的异常处理方式，如下图所示：



在Java中，我们经常可以看到空指针那样的错误的堆栈信息，然而这个堆栈实则是线程在出错的情况下 **“不得已”** 才输出来的。从图中我们可以看到：

- 当线程出错时，首先会检查当前线程是否指定了错误处理器；
- 如果当前线程没有指定错误处理器，则继续检查其所在的线程组是否指定（注意，前面我们已经说过，每个线程都是有线程组的）；
- 如果当前线程的线程组也没有指定，则继续检查其父线程是否指定；
- 如果父线程同样没有指定错误处理器，则最后检查默认处理是否设置；
- **如果默认处理器也没有设置，那么将不得不输出错误的堆栈信息。**

2. 多线程间的异常处理

不要忘记，**主线程也是线程**，所以当你理解了主线程的错误处理方式后，你也就理解了子线程中的异常处理方式，它们和主线程是相同的。在主线程中，我们可以通过`Thread.setDefaultUncaughtExceptionHandler`来设置自定义异常处理器。而在新的子线程中，则可以通过**线程对象**直接指定异常处理器，比如我们给前面的neZhaThread线程设置异常处理器：

```
neZhaThread.setName("哪吒");
neZhaThread.setUncaughtExceptionHandler(new
MyUncaughtExceptionHandler());
```

那么，设置处理器后的线程异常信息则输出如下：

```
出错了！线程名：哪吒，错误信息：我是哪吒，我被围攻了！

Process finished with exit code 0
```

你看，通过定义`uncaughtExceptionHandler`，我们已经捕获并处理了新线程抛出

的异常。

3. 理解UncaughtExceptionHandler

从上面的代码中，相信你已经直观地理解UncaughtExceptionHandler用法。在Java中，**UncaughtExceptionHandler用于处理线程突然异常终止的情况**。当线程因某种原因抛出未处理的异常时，JVM虚拟机将会通过线程中的`getUncaughtExceptionHandler`查询该线程的错误处理器，并将该线程和异常信息作为参数传递过去。如果该线程没有指定错误处理器，将会按照上图所示的流程继续查找。

三、 定义uncaughtExceptionHandler的三个层面

1. 定义默认异常处理器

默认的错误处理器可以作为线程异常的兜底处理器，在线程和线程组未指定异常处理器时，可以使用默认的异常处理器。

```
 Thread.setDefaultUncaughtExceptionHandler(new
MyUncaughtExceptionHandler());
```

2. 自定义特定的异常处理器

如果某个线程需要特定的处理器时，通过线程对象指定异常处理器是个很不错的选择。当然，这种异常处理器不可以与其他线程共享。

```
neZhaThread.setUncaughtExceptionHandler(new
MyUncaughtExceptionHandler());
```

3. 继承ThreadGroup

通过继承ThreadGroup并覆写`uncaughtException`可以重设当前线程组的异常处理器逻辑。不过要注意的是，覆写线程组的行为并不常见，使用时需要慎重。

```
public class MyThreadGroupDemo  extends ThreadGroup{
    public MyThreadGroupDemo(String name) {
        super(name);
    }

    @Override
    public void uncaughtException(Thread t, Throwable e) {
        // 在这里重写线程组的异常处理逻辑
        System.out.println("出错了！线程名：" + t.getName() + "，错误
信息：" + e.getMessage());
    }
}
```

小结

以上就是关于线程异常处理的全部内容，在本文中我们介绍了多线程异常的处理方式以及`uncaughtExceptionHandler`的用法。对于多线程的异常处理应该记住：

- **线程内部的异常应尽可能在其内部解决**；
- 如果主线程需要捕获子线程异常，不可以使用`try`、`catch`，而是要使用`uncaughtExceptionHandler`。当然，已经在子线程内部捕获的异常，主线程将无法捕获。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码了解并体验uncaughtExceptionHandler用法。

延伸阅读与参考资料

- 掘金专栏：https://juejin.cn/column/6963590682602635294

- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

青铜10：千锤百炼-如何解决生产者与消费者经典问题

欢迎来到《王者并发课》，本文是该系列文章中的**第10篇**。

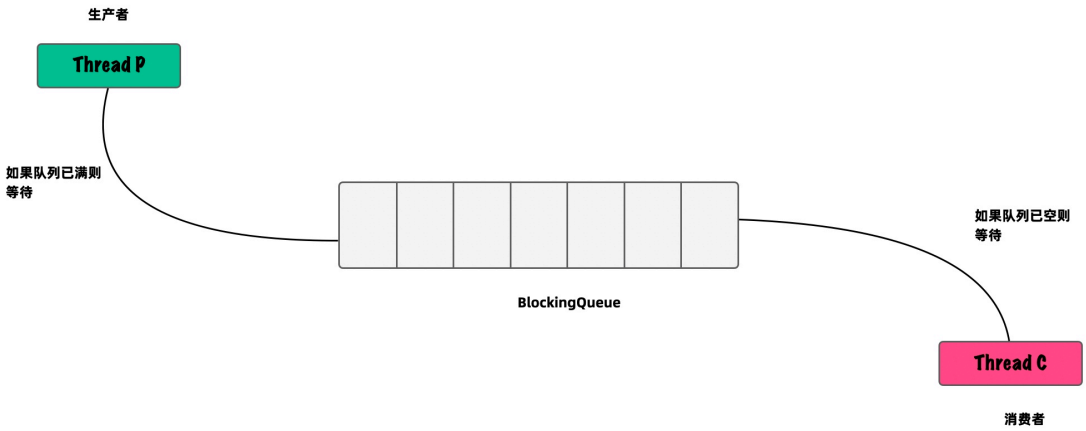
在本篇文章中，**我将为你介绍并发中的经典问题-生产者与消费者问题，并基于前面系列文章的知识点，通过wait、notify实现这一问题的简版方案。**

一、生产者与消费者问题

生产者消费者问题（Producer-consumer problem），也称有限缓冲问题（Bounded-buffer problem），是一个多进程、线程同步问题的经典案例。

这个问题描述了共享固定大小缓冲区的两个进程——即所谓的“**生产者**”和“**消费者**”——在实际运行时会发生的问题。**生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程**。与此同时，消费者也在缓冲区消耗这些数据。

生产者与消费者问题的关键在于**要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据**。

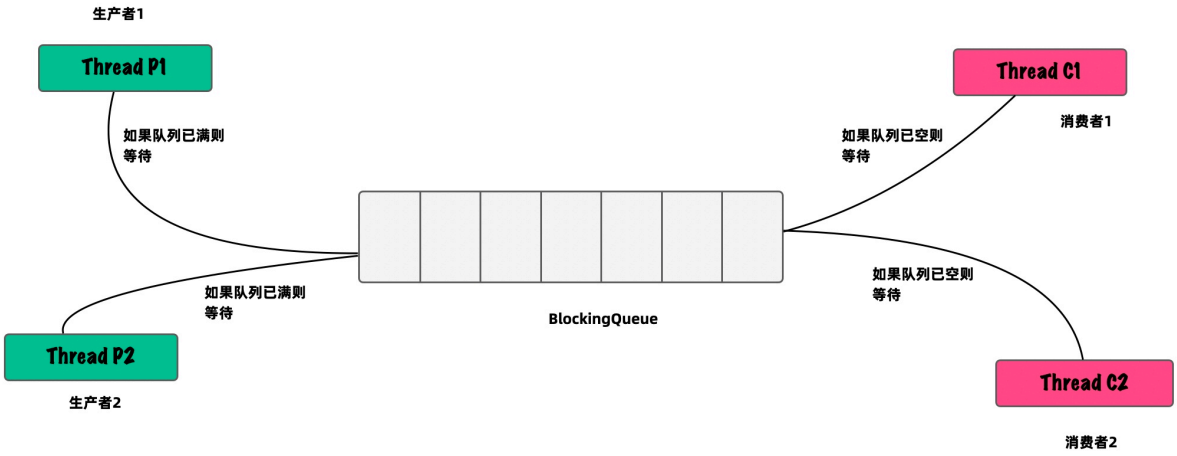


要解决该问题，**就必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到**

下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。

同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用线程间通信的方法解决该问题，常用的方法有信号量等。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。

当然，**生产者与消费者问题并不是局限于单个生产者与消费者，在实际工作中，遇到更多的是多个生产者和消费者的情形。**



生产者与消费者模式在软件开发与设计中有着非常广泛的应用。在这一模式中，生产者与消费者相互独立，它们仅通过缓冲区传递数据，因此可以用于程序间的**解耦**、**异步削峰**等。

生产者与消费者问题的要点：

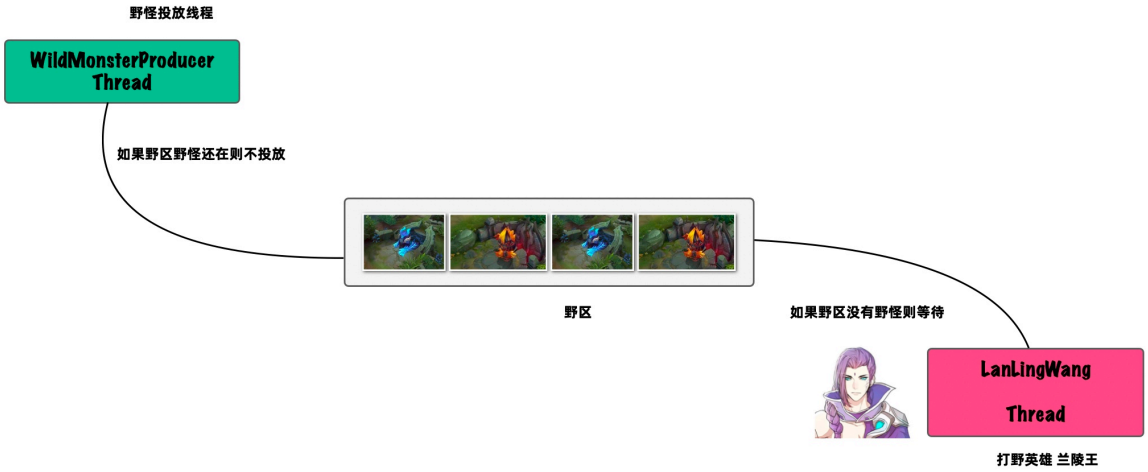
- **生产者与消费者解耦，两者通过缓冲区传递数据**；
- **缓冲区数据装满了之后，生产者停止数据生产或丢弃数据**；
- **缓冲区数据为空后，消费者停止消费并进入等待状态，等待生产者通知。**

二、实现生产者与消费者方案

本节中，我们通过王者中的一个场景来模拟生产者与消费者问题。

在王者中，英雄兰陵王需要通过打野来发育，但是野区的野怪在被打完之后，需要隔一段时间再投放。

所以，我们创建两个线程，一个作为**生产者**向野区投放野怪，一个作为**消费者**打怪。



生产者：每秒检查一次野区，如果野区没有野怪，则进行投放。**野怪投放后，通知打野英雄**。

```
// 野怪投放【生产者】
public static class WildMonsterProducer implements Runnable {
  public void run() {
    try {
      createWildMonster();
    } catch (InterruptedException e) {
      System.out.println("野怪投放被中断");
    }
  }

  //投放野怪，每1秒检查一次
  public void createWildMonster() throws InterruptedException {
    for (int i = 0;; i++) {
      synchronized(wildMonsterArea) {
        if (wildMonsterArea.size() == 0) {
          wildMonsterArea.add("野怪" + i);
          System.out.println(wildMonsterArea.getLast());
          wildMonsterArea.notify();
        }
      }
```

```
      Thread.sleep(1000);
    }
  }
}
```

消费者：打野英雄兰陵王作为消费者，在野区打怪发育。如果野区有野怪，则打掉野怪。 如果没有，会进行**等待野区新的野怪产生**。

```
// 兰陵王，打野英雄
public static class LanLingWang implements Runnable {
  public void run() {
    try {
      attackWildMonster();
    } catch (InterruptedException e) {
      System.out.println("兰陵王打野被中断");
    }
  }

  // 打野，如果没有则进行等待
  public void attackWildMonster() throws InterruptedException {
    while (true) {
      synchronized(wildMonsterArea) {
        if (wildMonsterArea.size() == 0) {
          wildMonsterArea.wait();
        }
        String wildMonster = wildMonsterArea.getLast();
        wildMonsterArea.remove(wildMonster);
        System.out.println("收获野怪：" + wildMonster);
      }
    }
  }
}
```

创建野区，并启动生产者与消费者线程。

```
public class ProducerConsumerProblemDemo {

    // 野怪活动的野区
    private static final LinkedList<String> wildMonsterArea = new
LinkedList<String>();

    public static void main(String[] args) {
        Thread wildMonsterProducerThread = new Thread(new
```

```
WildMonsterProducer());
        Thread lanLingWangThread = new Thread(new LanLingWang());

        wildMonsterProducerThread.start();
        lanLingWangThread.start();
    }
}
```

在上面几段代码中，你需要重点注意的是`synchronized`、`wait`和`notify`用法，它们是本次方案的关键。运行结果如下：

```
野怪0
收获野怪：野怪0
野怪1
收获野怪：野怪1
野怪2
收获野怪：野怪2
野怪3
收获野怪：野怪3
野怪4
收获野怪：野怪4
野怪5
收获野怪：野怪5
野怪6
收获野怪：野怪6
```

从结果可以看到，生产者在创建野怪后，打野英雄兰陵王会进行打野，实现了生产者与消费者的问题。

小结

以上就是关于线程异常处理的全部内容，在本文中我们基于**wait**、**notify**来解决生产者与消费者问题。对于本文内容，你需要理解生产者与消费者问题的核心是什么。另外，**本文所提供的方案仅仅是这一问题多种解决方案中的一种**，在后面的文章中，我们会根据新的知识点提供其他的解法。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码实现生产者与消费者问题。

延伸阅读与参考资料

- **Producer–consumer problem**
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

黄金01：两败俱伤-互不相让的线程如何导致了死锁僵局

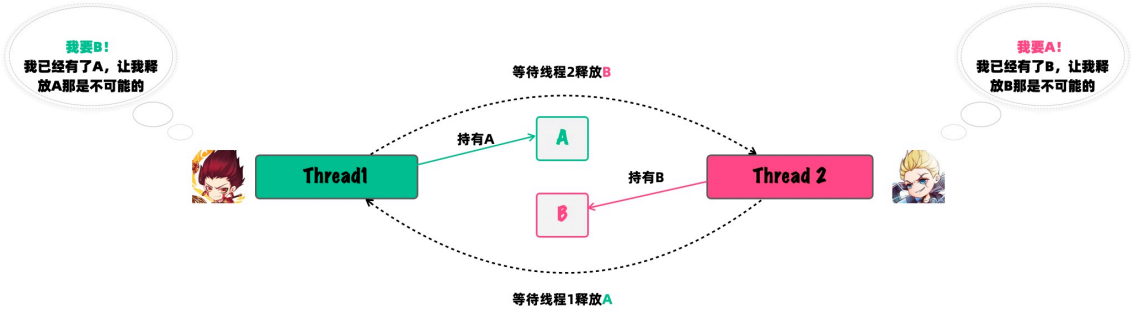
欢迎来到《王者并发课》，本文是该系列文章中的**第11篇**。

在本篇文章中，**我将为你介绍多线程中的经典问题-死锁，以及死锁的产生原因、处理方式和预防措施等。**

一、死锁的产生

观察下面这幅图，线程1持有了A，但它需要B；而线程2持有了B，但是它需要A。

你看，问题就来了，A、B都在等待对方已经持有的资源，并且都不释放，这就让事情陷入了僵局，也就是产生了**死锁**。



在并发编程中，死锁表示的是一种状态。在这种状态下，各方都在等待另一方释放所持有的资源，但是它们之间又缺乏必要的通信机制，导致彼此存在环路依赖而永远地等待下去。

死锁不仅存在于Java程序中，在诸如数据库等其他中间件及分布式架构中都会存在。在数据的设计中，会考虑到死锁的监测和恢复。当数据库中发生死锁时，将选择一个牺牲者并放弃对应的事务，同时释放锁定的资源。在它的竞争者执行结束

后，应用程序可以重新运行这个事务，因为它的竞争者此前已经完成事务。

然而，在JVM中，处理死锁并没有数据库中那么优雅。**当一组线程发生死锁时，“游戏”将到此结束，这些线程将不能再使用，而这可能会直接导致应用程序崩溃、性能降低或者部分功能停止**。

所以，和其他并发问题一样，**死锁是危险的，死锁造成的影响会立即表现出来，而如果在高负载情况下，这将是一场灾难**。

二、死锁产生的必要条件

从第一小节的图示中，我们可以看到死锁产生的一些必要条件：

1. **互斥**：一个资源每次只能被一个线程使用。比如，上图中的A和B同时只能被线程1和线程2其中一个使用；
2. **请求与保持条件**：一个线程在请求其他资源被阻塞时，对已经持有的资源保持不释放。比如，上图中的线程1在请求B时，并不会释放A；
3. **不剥夺条件**：对于线程已经获得的资源，在它主动释放前，不可以主动剥夺。比如，上图中线程1和线程2已经获得的资源，除非自己释放，否则不可以被强制剥夺；
4. **循环等待条件**：多个线程之间形成环状等待。上图中的线程1和线程2所形成的就是循环等待。

三、模拟并体验死锁

在了解什么是死锁及其产生的条件后，我们根据上图中的死锁情景，通过一段代码来模拟体验死锁的发生。

根据上图所示，定义哪吒线程，在运行时将持有**A**并请求**B**：

```
private static class NeZha implements Runnable {
  public void run() {
    synchronized(lockA) {
      System.out.println("哪吒: 持有A!");
```

```
      try {
        Thread.sleep(10);
      } catch (InterruptedException ignored) {}
      System.out.println("哪吒: 等待B...");

      synchronized(lockB) {
        System.out.println("哪吒: 已经同时持有A和B...");
      }
    }
  }
}
```

定义兰陵王线程，在运行时持有**B**并请求**A**:

```
private static class LanLingWang implements Runnable {
  public void run() {
    synchronized(lockB) {
      System.out.println("兰陵王: 持有B!");

      try {
        Thread.sleep(10);
      } catch (InterruptedException ignored) {}
      System.out.println("兰陵王: 等待A...");

      synchronized(lockA) {
        System.out.println("兰陵王: 已经同时持有A和B...");
      }
    }
  }
}
```

启动两个线程:

```
public class DeadLockDemo {
    public static final Object lockA = new Object();
    public static final Object lockB = new Object();

    public static void main(String args[]) {
        Thread thread1 = new Thread(new NeZha());
        Thread thread2 = new Thread(new LanLingWang());
        thread1.start();
        thread2.start();
```

```
    }
}
```

两个线程的输出结果如下：

```
哪吒: 持有A!
兰陵王: 持有B!
哪吒: 等待B...
兰陵王: 等待A...
```

从结果中可以看到，哪吒和兰陵王分别持有了A和B，但他们又相互请求对方持有的资源，**最终导致死锁，两个线程进入了无限地等待**。

四、死锁的处理

1. 忽略死锁

忽略死锁是一种鸵鸟政策，它假设永远不会发生死锁。**这种策略适用于死锁发生概率较低且影响可容忍的场景，如果死锁被证明永远不会发生也可以采用这种策略**。

2. 检测

在这种策略下，死锁是允许发生的。如果系统检测到死锁，也会对其进行纠正，比如跟踪线程状态和资源分配。在死锁时，可以通过一些方法进行纠正：

- **线程终止**：选择其中一个或多个线程进行终止，释放资源，打破死锁状态；
- **资源抢占**：重新分配各线程已经抢占的资源，直到打破死锁。

3. 预防

对待死锁问题，预防是关键。本文第二小节已经列举死锁产生的一些必要条件，**所以如果要预防死锁，只要打破其中任一条件即可**，Java中具体的死锁预发方式我们会在后面的文章中介绍。

小结

以上就是关于死锁的全部内容。在本文中，我们介绍了什么是死锁，以及死锁产生的必要条件和应对策略。对待开发中的死锁问题，既要保持敬畏之心，也不必闻之色变，审慎分析死锁的可能并设计合理策略可以有效预防死锁。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 运行本文的示例代码，尝试找到破解其死锁的方法。

延伸阅读与参考资料

- **死锁**
- 《Java Concurrency in Practice》
- **死锁预防算法**
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

黄金02：行稳致远-如何让你的线程免于死锁

欢迎来到《王者并发课》，本文是该系列文章中的**第12篇**。

在上篇文章中，我们介绍了死锁的概念及其原因，**本文将为你介绍的是几种常见的死锁预防策略**。

简单来说，预防死锁主要有三种策略：

- 顺序化加锁；
- 给锁一个超时期限；
- 检测死锁。

一、顺序化加锁

通常，死锁的产生是由于多个线程**无序请求**资源造成的。资源是有限的，不可能同时满足所有线程的请求。**然而，如果能按照一定的顺序分别满足各个线程的请求，那么死锁也就不再存在，也就是所谓的顺序化加锁（Lock Ordering）**。

举个通俗点的例子，烦人的路口堵车你一定遇到过。路口之所以堵车，是因为车太多了，大家都争相往自己的方向去，互不相让，不堵才怪。这时候，就需要交警在中间进行协调指挥，疏解拥堵。交警之所以可疏解拥堵，其根本原因在于，交警让原本处于**无序竞争**的车流变成了**井然有序**的队列。

车还是那么多的车，路口还是那个路口，可是道路通畅了，这和线程的锁竞争是类似的道理。

在上篇文章的死锁代码中，线程1和线程2分别先占有了A和B，导致了死锁。按照刚才的思路，我们把顺序调整下，线程1和线程2都先占有A，然后再同时争夺B，那么死锁就不会发生。

定义哪吒线程1，先抢占**A**再争夺**B**：

```
private static class NeZha implements Runnable {
  public void run() {
    synchronized(lockA) {
      System.out.println("哪吒: 持有A!");

      try {
        Thread.sleep(10);
      } catch (InterruptedException ignored) {}
      System.out.println("哪吒: 等待B...");

      synchronized(lockB) {
        System.out.println("哪吒: 已经同时持有A和B...");
      }
    }
  }
}
```

定义兰陵王线程2，也是先抢占**A**再抢占**B**，**这与此前就不同了**：

```
private static class LanLingWang implements Runnable {
  public void run() {
    synchronized(lockA) {
      System.out.println("兰陵王: 持有A!");

      try {
        Thread.sleep(10);
      } catch (InterruptedException ignored) {}
      System.out.println("兰陵王: 等待B...");

      synchronized(lockB) {
        System.out.println("兰陵王: 已经同时持有A和B...");
      }
    }
  }
}
```

启动两个线程：

```
public class DeadLockDemo {
```

```
    public static final Object lockA = new Object();
    public static final Object lockB = new Object();

    public static void main(String args[]) {
        Thread thread1 = new Thread(new NeZha());
        Thread thread2 = new Thread(new LanLingWang());
        thread1.start();
        thread2.start();
    }
}
```

两个线程的输出结果如下：

```
哪吒: 持有A!
哪吒: 等待B...
哪吒: 已经同时持有A和B...
兰陵王: 持有A!
兰陵王: 等待B...
兰陵王: 已经同时持有A和B...
```

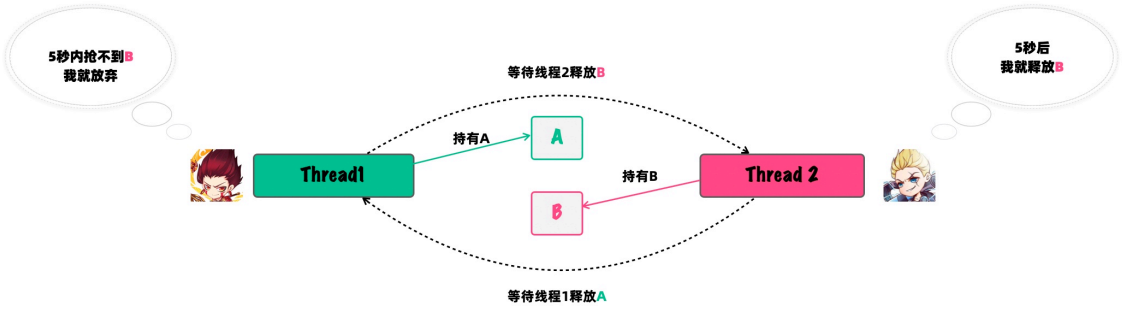
从运行的结果中可以看到，**两个线程都先后获得了自己所需要的资源，而没有导致死锁**。

调整加锁顺序是一种简单但有效的死锁预防策略。但是，这一策略并不是万能的，它仅适用于你在编码时已经知晓加锁的顺序。

二、给锁一个超时期限

在上篇文章中，我们说过死锁的产生有一些必要的条件，其中一个是**无限等待**。设定锁超时时间正是为了打破这一条件，**让无限等待变成有限等待**。

仍然以前面的代码为例，哪吒和兰陵王两个线程在争夺资源时，对方都互不相让导致了无限等待的僵局。而此时，如果其中任何一方给等待设定一个期限，那么时间一到，僵局将不攻自破，而线程仍可以再稍等片刻后继续尝试。



需要注意的是，`synchronized`**代码块不可以指定锁超时**。所以，如果需要锁超时，你需要使用自定义锁，或者使用JDK提供的并发工具类。相关工具类的用法，会在后续文章中介绍，本文暂不展开描述。

另外，所谓给锁加一个超时的期限，其实有两层含义。**一是在请求锁时需要设定超时时间，二是在获取锁之后对锁的持有也要有个超时时间，总不能到手就不放，那是耍流氓**。

三、死锁检测

作为死锁预防的第三种策略，你可以认为**死锁检测（Deadlock Detection）是一项较重的被动技能**，当我们无法顺序化加锁，也无法设置锁的超时时间，那么就需要进行死锁检测。

死锁检测的核心原理在于对线程和资源进行数据化打标和跟踪。

在线程获取锁时，会将锁和线程的对应关系通过Graph或者Map等数据结构记录下来。这样一来，线程在获取锁被拒绝时，可以通过**遍历**已经记录的数据分析是否存在死锁。

当线程发现死锁的情况后，可以采取释放锁，稍等片刻后再次尝试。

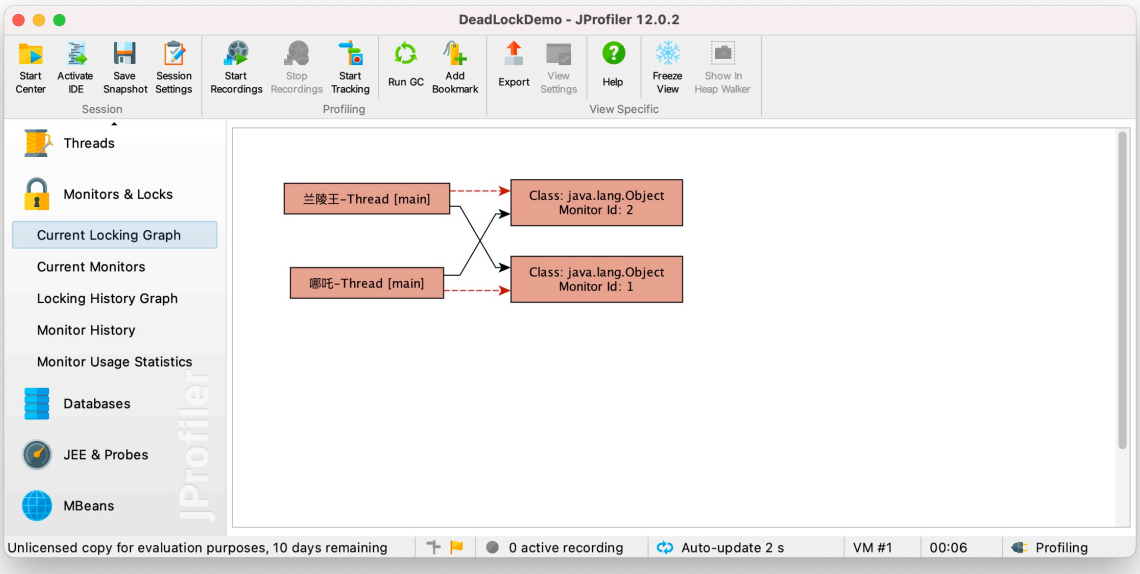
附、如何可视化查看线程死锁等状态

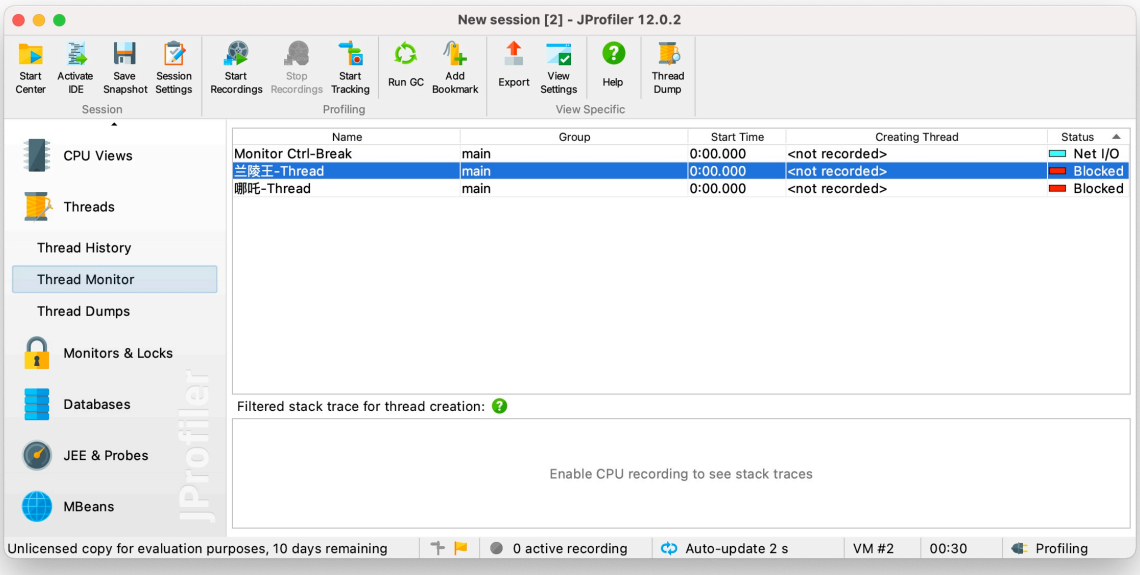
在你感觉线程可能被阻塞或死锁时，可以通过`jstack`命令查看。如果存在死锁，输出的结果中会有明确的死锁提示，如下面所示：

```
$ jstack -F 8321
Attaching to process ID 8321, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
============================

"Thread2":
  waiting to lock Monitor@0x000af398 (Object@0xf819aa10, a
java/lang/String),
  which is held by "Thread1"
"Thread1":
  waiting to lock Monitor@0x000af400 (Object@0xf819aa48, a
java/lang/String),
  which is held by "Thread2"

Found a total of 1 deadlock.
```

除了**`jstack`**之外，JProfiler也是一款非常强大的线程与堆栈分析工具，并可以和IDEA等IDE完美结合。

借助于JProfiler，我们可以非常直观地看到上述示例代码中的死锁，也可以在Thread Monitor中看到两个线程的状态为**blocked**.





需要注意的是，**JProfiler是一款付费软件**，它提供了十天的免费试用时间。如果没有常规的使用需求，而是仅用于学习的话，十天也是够用的。当然，你也可以考虑使用jConsole、jVisualvm等。

小结

以上就是关于死锁预防策略的全部内容。在本文中，我们介绍了三种死锁预发策

略。三种策略各有利弊，就实际工作中的应用而言，**第二种给锁设定超时期限是更为常用的一种做法**，而第一种和第三种具有一定的逻辑难度和技术难度，更侧重于理解而非实际应用。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 通过`jstack`命令查看死锁并解决。

延伸阅读与参考资料

- Deadlock Prevention
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

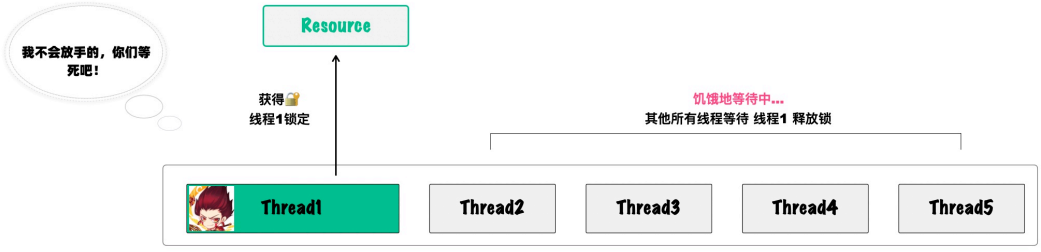
黄金03：雨露均沾-不要让你的线程在竞争中被”饿死“

欢迎来到《王者并发课》，本文是该系列文章中的**第13篇**。

在上篇文章中，我们介绍了避免死锁的几种策略。虽然死锁臭名昭著，然而在并发编程中，除了死锁之外，还有一些同样重要的线程活跃性问题值得关注。它们的知名度不高，但破坏性极强，本文将介绍的正是其中的**线程饥饿**和**活锁问题**。

一、饥饿的产生

所谓线程 **饥饿（Starvation）** 指的是在多线程的资源竞争中，**存在贪婪的线程一直锁定资源不释放，其他的线程则始终处于等待状态，然而这个等待是没有结果的，它们会被活活地饿死。**



独占者的贪婪是饥饿产生的原因之一，概括来说，饥饿一般由下面三种原因导致：

（1）线程被无限阻塞

当获得锁的线程需要执行无限时间长的操作时（比如IO或者无限循环），那么后面的线程将会被**无限阻塞**，导致被饿死。

（2） 线程优先级降低没有获得CPU时间

当多个竞争的线程被设置优先级之后，优先级越高，线程被给予的CPU时间越多。在某些极端情况下，**低优先级的线程可能永远无法被授予充足的CPU时间**，从而导致被饿死。

（3） 线程永远在等待资源

在青铜系列文章中，我们说过`notify`在发送通知时，是无法唤醒指定线程的。当多个线程都处于`wait`时，**那么部分线程可能始终无法被通知到**，以至于挨饿。

二、饥饿与公平

为了直观体验线程的饥饿，我们创建了下面的代码。

创建哪吒、兰陵王等四个英雄玩家，他们以竞争的方式打野，杀死野怪可以获得经济收益。

```
public class StarvationExample {

  public static void main(String[] args) {
    final WildMonster wildMonster = new WildMonster();

    String[] players = {
      "哪吒",
      "兰陵王",
      "铠",
      "典韦"
    };
    for (String player: players) {
      Thread playerThread = new Thread(new Runnable() {
        public void run() {
          wildMonster.killWildMonster();
        }
      });
      playerThread.setName(player);
      playerThread.start();
    }
  }
}
```

```
public class WildMonster {
  public synchronized void killWildMonster() {
    while (true) {
      String playerName = Thread.currentThread().getName();
      System.out.println(playerName + "斩获野怪! ");
      try {
        Thread.sleep(500);
      } catch (InterruptedException e) {
        System.out.println("打野中断");
      }
    }
  }
}
```

运行结果如下:

```
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!
哪吒斩获野怪!

Process finished with exit code 130 (interrupted by signal 2:
SIGINT)
```

从结果中可以看到，在几个线程的运行中，**始终只有哪吒可以斩获野怪，其他英雄束手无策等着被饿死**。为什么会发生这样的事?

仔细看WildMonster类中的代码，问题出在`killWildMonster`同步方法中。**一旦某个英雄进入该方法后，将一直持有对象锁，其他线程被阻塞而无法再进入。**

当然，解决的方法也很简单，只要打破**独占**即可。比如，我们在下面的代码中把`Thread.sleep`改成`wait`，那么问题将迎刃而解。

```
public static class WildMonster {
```

```
    public synchronized void killWildMonster() {
      while (true) {
        String playerName = Thread.currentThread().getName();
        System.out.println(playerName + "斩获野怪！");
        try {
          wait(500);
        } catch (InterruptedException e) {
          System.out.println("打野中断");
        }
      }
    }
  }
```

运行结果如下：

```
哪吒斩获野怪!
铠斩获野怪!
兰陵王斩获野怪!
典韦斩获野怪!
兰陵王斩获野怪!
典韦斩获野怪!

Process finished with exit code 130 (interrupted by signal 2:
SIGINT)
```

从结果中可以看到，四个英雄都获得了打野的机会，在一定程度上实现了**公平**。（备注：`wait`会释放锁，但`sleep`不会，对此不理解的可以查看青铜系列文章。）

如何让线程之间公平竞争，是线程问题中的重要话题。虽然我们无法保证百分百的公平，但我们仍然要通过设计一定的数据结构和使用相应的工具类来增加线程之间的公平性。

关于线程之间的公平性，在本文中重要的是理解它的存在和重要性，关于如何优雅地解决，我们会在后续的文章中介绍相关的并发工具类。

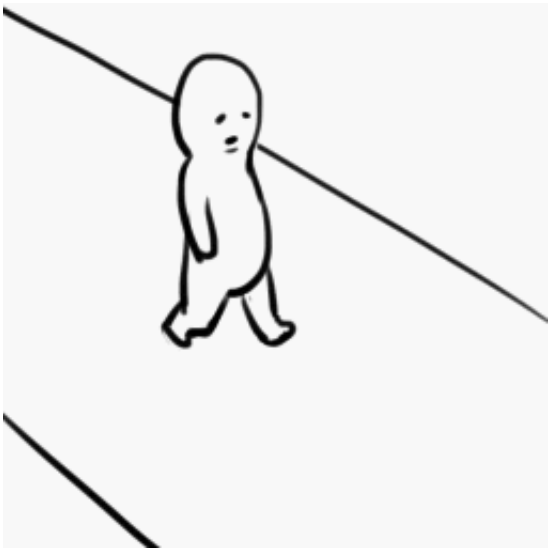
三、活锁的麻烦

相对于死锁，你可能对活锁没有那么熟悉。然而，活锁所造成的负面影响并不亚于死锁。**在结果上，活锁和死锁都是灾难性的，都将会造成应用程序无法提供正常的服务能力**。

所谓**活锁（LiveLock）**，指的是**两个线程都忙于响应对方的请求，但却不干自己的事。它们不断地重复特定的代码，却一事无成**。

不同于死锁，活锁并不会造成线程进入阻塞状态，但它们会原地打转，所以在影响上和死锁相似，程序会进入无线死循环，无法继续进行。

如果你无法直观理解活锁是什么，相信你在走路时一定遇到过下面这种情况。两人相向而行，出于礼貌两人互相让行，让来让去，结果两人仍然无法通行。活锁，也是这个意思。



小结

以上就是关于线程饥饿与活锁的全部内容。在本文中，我们介绍了线程产生饥饿的原因。对待线程饥饿，没有百分百的方案，但可以尽可能地实现公平竞争。我们没有在本文列举线程公平性的一些工具类，因为我认为对问题的理解要比解决方案更重要。如果没有对问题的理解，方案在落地时也会出现知其然而不知其所以然的情况。另外，虽然活锁并不像死锁那样知名度，但是对活锁的恰当理解仍然非常必要，它是并发知识体系中的一部分。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码设置不同线程的优先级，体验线程饥饿并给出解决方案。

延伸阅读与参考资料

- 动态图片引用
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

铂金01：探本溯源-为何说Lock接口是Java中锁的基础

欢迎来到《王者并发课》，本文是该系列文章中的**第14篇**。

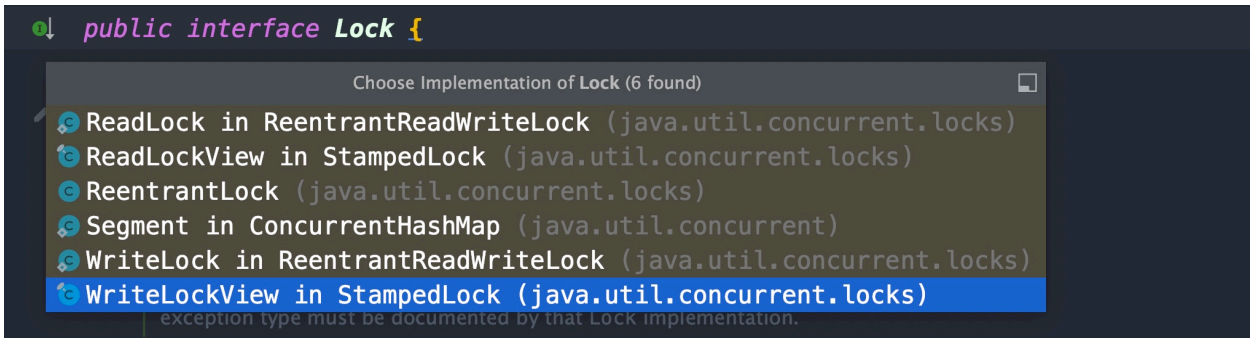
在**黄金**系列中，我们介绍了并发中一些问题，比如死锁、活锁、线程饥饿等问题。在并发编程中，这些问题无疑都是需要解决的。所以，在铂金系列文章中，我们会从并发中的问题出发，探索Java所提供的**锁的能力**以及它们是如何解决这些问题的。

作为铂金系列文章的第一篇，我们将从**Lock接口**开始介绍，因为它是Java中锁的基础，也是并发能力的基础。

一、理解Java中锁的基础：Lock接口

在青铜系列文章中，我们介绍了通过`synchronized`关键字实现对方法和代码块加锁的用法。**然而，虽然`synchronized`非常好用、易用，但是它的灵活度却十分有限，不能灵活地控制加锁和释放锁的时机**。所以，为了更灵活地使用锁，并满足更多的场景需要，就需要我们能够自主地定义锁。于是，就有了**Lock接口**。

理解Lock最直观的方式，莫过于直接在JDK所提供的并发工具类中找到它，如下图所示：



可以看到，Lock接口提供了一些能力API，并有一些具体的实现，如

ReentrantLock、ReentrantReadWriteLock等。

1. Lock的五个核心能力API

- `void lock()`：获取锁。**如果当前锁不可用，则会被阻塞直至锁释放**；
- `void lockInterruptibly()`：获取锁并允许被中断。**这个方法和`lock()`类似，不同的是，它允许被中断并抛出中断异常**。
- `boolean tryLock()`：尝试获取锁。**会立即返回结果，而不会被阻塞**。
- `boolean tryLock(long timeout, TimeUnit timeUnit)`：尝试获取锁并等待一段时间。这个方法和`tryLock()`，但是它会根据参数等待一会，**如果在规定的时间内未能获取到锁就会放弃**；
- `void unlock()`：释放锁。

2. Lock的常见实现

在Java并发工具类中，Lock接口有一些实现，比如：

- ReentrantLock：可重入锁；
- ReentrantReadWriteLock：可重入读写锁；

除了列举的两个实现外，还有一些其他实现类。对于这些实现，暂且不必详细了解，后面会详细介绍。**在目前阶段，你需要理解的是Lock是它们的基础**。

二、自定义Lock

接下来，我们基于前面的示例代码，看看如何将`synchronized`版本的锁用Lock来实现。

```
public static class WildMonster {
  private boolean isWildMonsterBeenKilled;

  public synchronized void killWildMonster() {
    String playerName = Thread.currentThread().getName();
    if (isWildMonsterBeenKilled) {
      System.out.println(playerName + "未斩杀野怪失败...");
```

```
            return;
        }
        isWildMonsterBeenKilled = true;
        System.out.println(playerName + "斩获野怪! ");
    }
}
```

1. 实现一把简单的锁

创建类WildMonsterLock并实现Lock接口，WildMonsterLock将是取代`synchronized`的关键：

```
// 自定义锁
public class WildMonsterLock implements Lock {
    private boolean isLocked = false;

    // 实现lock方法
    public void lock() {
        synchronized (this) {
            while (isLocked) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            isLocked = true;
        }
    }

    // 实现unlock方法
    public void unlock() {
        synchronized (this) {
            isLocked = false;
            this.notify();
        }
    }
}
```

在实现Lock接口时，你需要实现它上述的所有方法。不过，为了简化代码方便展示，我们移除了WildMonsterLock类中的`tryLock`等方法。

对于`wait`和`notify`方法的时候，如果你不熟悉的话，可以查看青铜系列的文章。**这里需要提醒的是，`notify`在使用时务必要和`wait`是同一个监视器。**

基于刚才定义的WildMonsterLock，创建WildMonster类，并在方法killWildMonster中使用WildMonsterLock对象，从而取代synchronized.

```
// 使用刚才自定义的锁
 public static class WildMonster {
   private boolean isWildMonsterBeenKilled;

   public void killWildMonster() {
     // 创建锁对象
     Lock lock = new WildMonsterLock();
     // 获取锁
     lock.lock();
     try {
       String playerName = Thread.currentThread().getName();
       if (isWildMonsterBeenKilled) {
         System.out.println(playerName + "未斩杀野怪失败...");
         return;
       }
       isWildMonsterBeenKilled = true;
       System.out.println(playerName + "斩获野怪！");
     } finally {
       // 执行结束后，无论如何不要忘记释放锁
       lock.unlock();
     }
   }
}
```

输出结果如下：

```
哪吒斩获野怪！
典韦未斩杀野怪失败...
兰陵王未斩杀野怪失败...
铠未斩杀野怪失败...

Process finished with exit code 0
```

从结果中可以看到：**只有哪吒一人斩获了野怪，其他几个英雄均以失败告终，结果符合预期**。这说明，WildMonsterLock达到了和`synchronized`一致的效果。

不过，这里有细节需要注意。在使用`synchronized`时我们无需关心锁的释放，JVM会帮助我们自动完成。然而，**在使用自定义的锁时，一定要使用`try...finally`来确保锁最终一定会被释放**，否则将造成后续线程被阻塞的严重后果。

2. 实现可重入的锁

在`synchronized`中，**锁是可以重入的。所谓锁的可重入，指的是锁可以被线程重复或递归调用**。比如，加锁对象中存在多个加锁方法时，当线程在获取到锁进入其中任一方法后，线程应该可以同时进入其他的加锁方法，而不会出现被阻塞的情况。当然，前提条件是这个加锁的方法用的是同一个对象的锁（监视器）。

在下面这段代码中，方法A和B都是同步方法，并且A中调用B. 那么，线程在调用A时已经获得了当前对象的锁，那么线程在A中调用B时可以直接调用，这就是锁的可重入性。

```
public class WildMonster {
    public synchronized void A() {
        B();
    }

    public synchronized void B() {
       doSomething...
    }
}
```

所以，为了让我们自定义的WildMonsterLock也支持可重入，我们需要对代码进行一点改动。

```
public class WildMonsterLock implements Lock {
    private boolean isLocked = false;

    // 重点：增加字段保存当前获得锁的线程
```

```
    private Thread lockedBy = null;
    // 重点：增加字段记录上锁次数
    private int lockedCount = 0;

    public void lock() {
        synchronized (this) {
            Thread callingThread = Thread.currentThread();
            // 重点：判断是否为当前线程
            while (isLocked && lockedBy != callingThread) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            isLocked = true;
            lockedBy = callingThread;
            lockedCount++;
        }
    }

    public void unlock() {
        synchronized (this) {
            // 重点：判断是否为当前线程
            if (Thread.currentThread() == this.lockedBy) {
                lockedCount--;
                if (lockedCount == 0) {
                    isLocked = false;
                    this.notify();
                }
            }
        }
    }
}
```

在新的WildMonsterLock中，我们增加了`this.lockedBy`和`lockedCount`字段，并在加锁和解锁时增加对线程的判断。**在加锁时，如果当前线程已经获得锁，那么将不必进入等待。而在解锁时，只有当前线程能解锁。**

`lockedCount`字段则是为了保证解锁的次数和加锁的次数是匹配的，比如加锁了3次，那么相应的也要3次解锁。

3. 关注锁的公平性

在黄金系列文章中，我们提到了线程在竞争中可能被饿死，因为竞争并不是公平的。**所以，我们在自定义锁的时候，也应当考虑锁的公平性**。

三、小结

以上就是关于Lock的全部内容。在本文中，我们介绍了Lock是Java中各类锁的基础。它是一个接口，提供了一些能力API，并有着完整的实现。并且，我们也可以根据需要自定义实现锁的逻辑。所以，在学习Java中各种锁的时候，最好先从Lock接口开始。同时，在替代synchronized的过程中，我们也能感受到Lock有一些synchronized所不具备的优势：

- **synchronized用于方法体或代码块，而Lock可以灵活使用，甚至可以跨越方法**；
- **synchronized没有公平性，任何线程都可以获取并长期持有，从而可能饿死其他线程。而基于Lock接口，我们可以实现公平锁，从而避免一些线程活跃性问题**；
- **synchronized被阻塞时只有等待，而Lock则提供了`tryLock`方法，可以快速试错，并可以设定时间限制，使用时更加灵活**；
- **synchronized不可以被中断，而Lock提供了`lockInterruptibly`方法，可以实现中断。**

另外，在自定义锁的时候，要考虑锁的公平性。而在使用锁的时候，则需要考虑锁的安全释放。

夫子的试炼

- 基于Lock接口，自定义实现一把锁。

延伸阅读与参考资料

- **Locks in Java**
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金02：豁然开朗-“晦涩难懂”的ReadWriteLock竟如此妙不可言

欢迎来到《王者并发课》，本文是该系列文章中的**第15篇**。

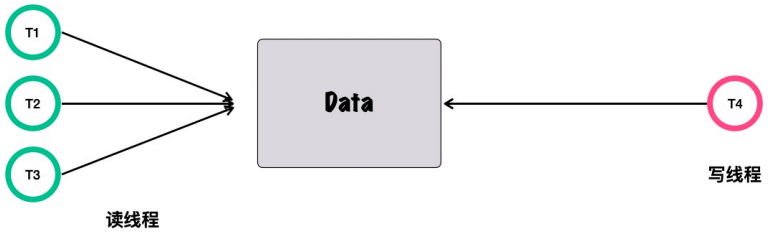
在上篇文章中，我们介绍了Java中锁的基础Lock接口。在本文中，我们将介绍Java中锁的另外一个重要的基本型接口，即**ReadWriteLock**接口。

在探索Java中的并发时，ReadWriteLock无疑是重要的，然而理解它却并不容易。如果你此前曾经检索资料，应该会发现大部分的文章对它的描述都比较晦涩难懂，或连篇累牍的源码陈列，或隔靴搔痒的三言两语，既说不到重点，也说不清来龙去脉。

所以，在本文中我们会将介绍的重点放在对思路的理解上，而不是对源码的解读上。对于源码以及其背后的知识，我们将在后面的更高级的系列中进行讲解。

一、理解ReadWriteLock存在的价值

理解ReadWriteLock，首先要理解它存在的意义是什么。**换言之，它要解决什么问题**。为此，我们不妨从下图着手一探究竟。



不知你看明白了没有，这幅图所表达的有三层含义：

- 大量线程在竞争同一份资源；
- 这些线程中有的是**读请求**，有的是**写请求**；
- 在多个线程的请求中，**读请求明显高于写请求**。

这样的场景是否似曾相识？**没错，它就是典型的缓存应用场景**。

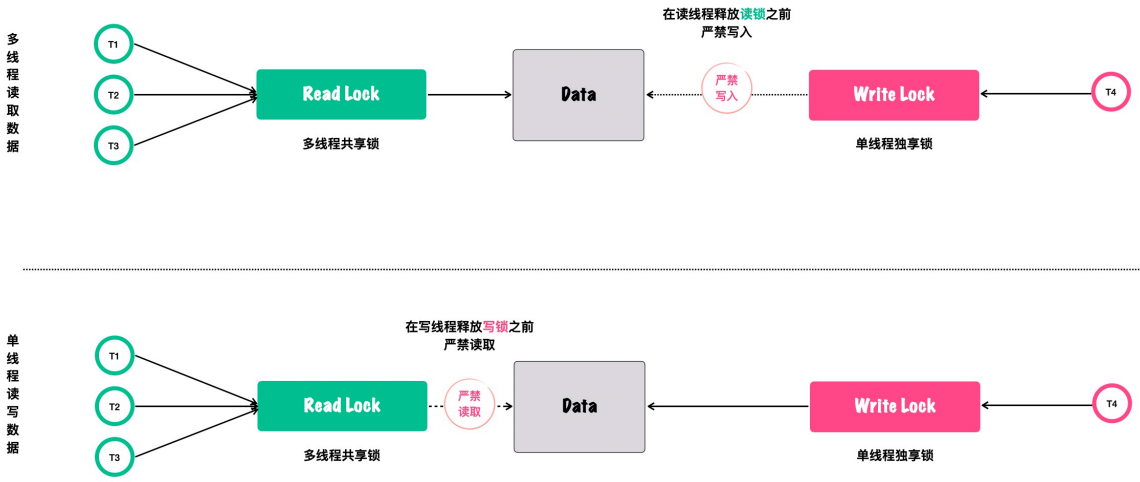
众所周知，缓存的存在是为了提高应用的读写性能。一方面，我们需要通过缓存拦截大量的读数据的请求。另一方面，我们也需要不定期地更新缓存。**但总体而言，更新缓存的次数远远小于读缓存的次数**。

在这个过程中，**关键问题在于，为了保持数据一致性，我们在读写缓存的时候，不能让读请求拿到脏数据，这就需要用到锁**。然而，**更关键的问题在于，虽然读写之间需要互斥，但读与读之间不可以互斥**。

总结来说，这个问题主要有下面这几个要点：

- **数据允许多个线程同时读取，但只允许一个线程进行写入**；
- **在读取数据的时候，不可以存在写操作或者写请求**；
- **在写数据的时候，不可以存在读请求**。

如果你对此仍然有些迷茫，**那么下面这张图建议你收藏**，这张图正是ReadWriteLock对问题的概述和它的解决方案，也是诠释ReadWriteLock最好的一幅图。



在你没有理解ReadWriteLock之前，你会觉得它十分晦涩且源码枯燥。**然而，一旦**

你理解它要解决的问题，以及它所提供的方案后，你会发现它的设计竟然如此巧妙。它竟然设计了两种截然不同的锁，**其中一把正如我们此前认知的那样是线程互斥的，而另一把锁竟然可以为多个线程所共享**！两把锁的完美配合，解决了并发读写的场景问题。

在恍然大悟后，所谓源码不过是队列与共享，它们是ReadWriteLock的一种实现方式，而不是阻挡你理解的绊脚石。

二、自主实现ReadWriteLock

在理解了ReadWriteLock背后的问题和它的解决思路之后，我们就可以完全抛开JDK中的源码自己实现一把读写锁。

```
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;

  public synchronized void lockRead() throws
InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws
InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
      wait();
    }
    writeRequests--;
```

```
    writers++;
  }

  public synchronized void unlockWrite() throws
InterruptedException{
    writers--;
    notifyAll();
  }
}
```

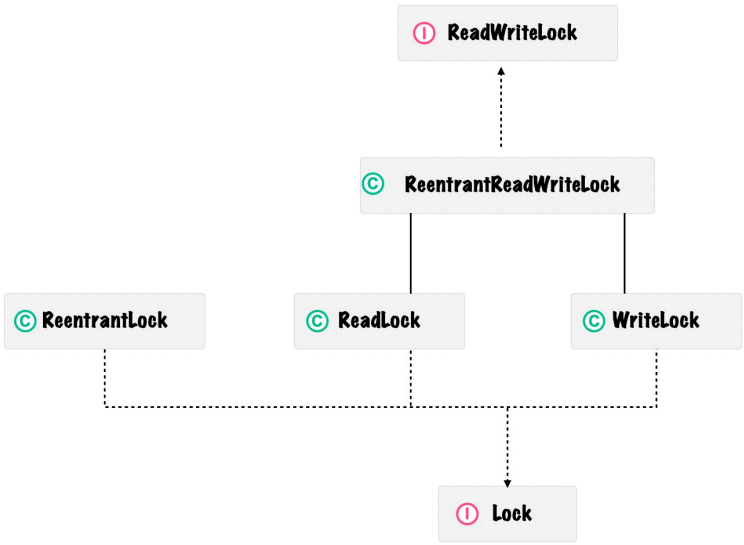
在读锁`lockRead()`中，是不允许有**写请求**或**写操作**的。如果有，那么读请求将进入等待。

而在`lockWrite()`中，**同时不允许读请求和其他写操作的存在，此时只允许有一个写请求**。

以上就是读写锁简单的自主实现方式。**当然，它是不完善的，只是基本的示例**。它没有考虑到基本的**线程重入**问题，真实情况也比它复杂很多，但你理解它的意思就好。

三、Java中的ReadWriteLock是如何实现的

最后，我们再来看JDK中的ReadWriteLock实现的一些基本思路。ReadWriteLock和我们上篇所说的Lock接口以及其他类的基本关系如下图所示：



可以看到，JDK中的读写锁的实现是在**ReentrantReadWriteLock**这个类中。ReentrantReadWriteLock包含了两个内部类：ReadLock和WriteLock，而这两个类又实现了Lock接口。

读写锁的升级与降级

读写锁的升级与降级是ReentrantReadWriteLock中的一个重要知识点，也是高频的面试题。

从读锁到写锁，称之为锁的升级，反之为锁的降级。理解读写锁的升级和降级，最直观的方式是写代码验证。

代码片段1，先获取读锁，再获取写锁。

```
public class ReadWriteLockDemo {
    public static void main(String[] args) {
        ReadWriteLock readWriteLock = new
ReentrantReadWriteLock();
        readWriteLock.readLock().lock();
        System.out.println("已经获取读锁...");
        readWriteLock.writeLock().lock();
        System.out.println("已经获取写锁...");
    }
}
```

输出结果如下：

```
已经获取读锁...
```

代码片段2，先获取写锁，再获取读锁：

```
public class ReadWriteLockDemo {
    public static void main(String[] args) {
        ReadWriteLock readWriteLock = new
ReentrantReadWriteLock();
        readWriteLock.writeLock().lock();
        System.out.println("已经获取写锁...");
        readWriteLock.readLock().lock();
        System.out.println("已经获取读锁...");
    }
```

```
}
```

输出结果如下：

```
已经获取写锁...
已经获取读锁...

Process finished with exit code 0
```

这样一来，结果已经十分明了。**ReentrantReadWriteLock支持锁的降级，但不支持锁的升级**。

读写锁中的公平性

在前面的文章中，我们讲过线程饥饿的由来和后果，所以良好的并发工具类在设计时都会考虑到公平性，ReentrantReadWriteLock也是如此。

在ReentrantReadWriteLock中，同时提供了公平和非公平两种模式，且默认为非公平模式。从下面摘取的源码片段中，可以清晰地看到。

```
 public ReentrantReadWriteLock() {
        this(false);
 }

    /**
   /**
 * Creates a new {@code ReentrantReadWriteLock} with
 * default (nonfair) ordering properties.
 */
public ReentrantReadWriteLock() {
  this(false);
}

/**
 * Creates a new {@code ReentrantReadWriteLock} with
 * the given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair
ordering policy
 */
public ReentrantReadWriteLock(boolean fair) {
```

```
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}
```

小结

以上就是关于读写锁的全部内容。在本文中，我们从缓存问题出发，接着从ReadWriteLock中寻找答案，以便能从更轻松的角度理解ReadWriteLock的来龙去脉。

理解ReadWriteLock的关键不在于对源码的剖析，而在于对其思路的理解。

另外，我们简单地介绍了ReentrantReadWriteLock中的一些关键知识点，但诸如其背后的AQS等并没有展开陈述。对此也不必着急，我们会在后面有详细的分析介绍。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 尝试在示例代码中增加对读写线程的重入支持。

延伸阅读与参考资料

- **示例代码参考**
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金03：一劳永逸-如何理解锁的可重入问题

欢迎来到《王者并发课》，本文是该系列文章中的**第16篇**。

在前面的文章《铂金1：探本溯源-为何说Lock接口是Java中锁的基础》中，我们提到了锁的可重入问题，并作了简单介绍。鉴于锁的可重入是一个重要概念，所以本文把拿出来做一次单独讲解，以帮助你彻底理解它。

一、锁的可重入所造成的问题

首先，我们通过一段示例代码看锁的可重入是如何导致问题发生，以理解它的重要性。

```
public class ReentrantWildArea {
    // 野区锁定
    private boolean isAreaLocked = false;

    // 进入野区A
    public synchronized void enterAreaA() throws
InterruptedException {
        isAreaLocked = true;
        System.out.println("已经进入野区A...");
        enterAreaB();
    }
    // 进入野区B
    public synchronized void enterAreaB() throws
InterruptedException {
        while (isAreaLocked) {
            System.out.println("野区B方法进入等待中...");
            wait();
        }
        System.out.println("已经进入野区B...");
    }
```

```
    public synchronized void unlock() {
        isAreaLocked = false;
        notify();
    }
}
```

在上面这段代码中，我们创建了一片野区，包含了**野区A**和**野区B**。接着，我们再创建一个**打野英雄铠**，让他进去野区打野，看看会发生什么事情。

```
public static void main(String[] args) {
  // 打野英雄铠进入野区
  Thread kaiThread = new Thread(() -> {
    ReentrantWildArea wildArea = new ReentrantWildArea();
    try {
      wildArea.enterAreaA();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  });
  kaiThread.start();
}
```

输出结果如下：

```
已经进入野区A...
野区B方法进入等待中...
```

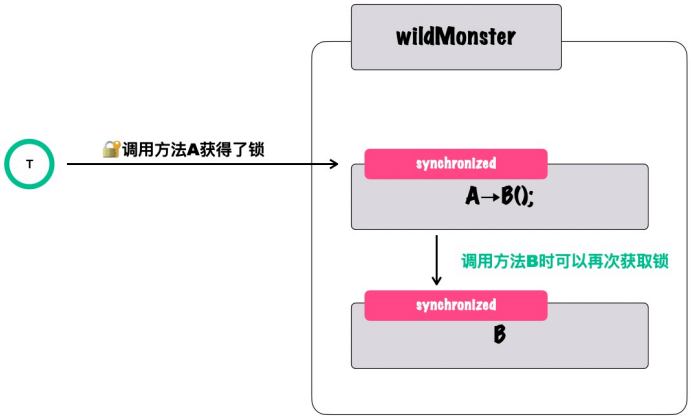
从结果中可以看到，**虽然在同一块野区，但是铠只进入野区A，却没能进入野区B，被阻塞在半道上了**。从代码分析上看，野区的两个方法都声明了`synchronized`，但铠在进入野区A之后，野区进行了锁定`isAreaLocked = true`，导致铠进入野区B时失败。

这就是典型的锁的可重入所造成的问题。**在并发编程时，如果未能处理好这一问题，将会造成线程的无限阻塞，其后果和死锁相当**。

二、理解锁的可重入

所谓锁的可重入，指的是锁可以被线程 *重复* 或 *递归* 调用，也可以理解为对同一把

锁的重复获取。 如果未能处理好锁的可重入问题，将会导致和死锁类似的问题。



三、如何避免锁的可重入问题

避免锁的可重入问题，需要注意两个方面：

- **尽量避免编写需要重入获取锁的代码**；
- **如果需要，使用可重入锁。**

在Java中，`synchronized`是可以重入的，下面的这段代码在调用时不会产生重入问题。

```
public class WildMonster {
    public synchronized void A() {
        B();
    }

    public synchronized void B() {
       doSomething...
    }
}
```

但是，基于Lock接口所实现的各种锁并不总是支持可重入的。在前面的文章中，我们已经展示过不支持重入的Lock接口实现。在具体的场景中使用时，需要务必注意

这点。如果需要可重入锁，可以使用Java中的**ReentrantLock**类。

小结

在本文中，我们再次介绍了锁的可重入问题，并介绍了其产生的原因及避免方式。Java中的`synchronized`关键字支持锁的可重入，但是其他显示锁并非总是支持这一特性，在使用时需要注意。

此外，需要注意的是，锁的可重入对锁的性能有一定的影响，而且实现起来更为复杂。所以，我们不能说锁的可重入与不可重入哪个好，这要取决于具体的问题。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 查看ReentrantLock源码，了解其支持可重入的原理。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金04：令行禁止-为何说信号量是线程间的同步利器

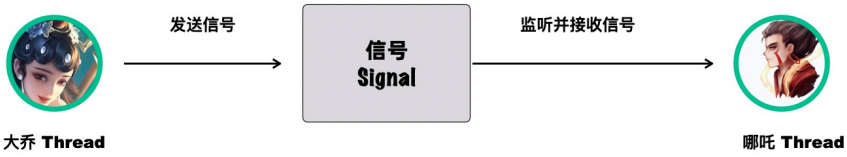
欢迎来到《王者并发课》，本文是该系列文章中的**第17篇**。

在并发编程中，信号量是线程同步的重要工具。在本文中，我将带你认识信号量的概念、用法、种类以及Java中的信号量。

信号量（Semaphore） 是线程间的同步结构，**主要用于多线程协作时的信号传递，以及对共享资源的保护、防止竞态的发生等**。信号量这一概念听起来比较抽象，然而读完本文你会发现它竟然也是如此通俗易懂且挺有用。

一、认识简单的信号量

虽然信号量的概念很抽象，但理解起来可以很简单。比如下面这幅图，**在峡谷对局中，大乔使用大招向哪吒发起了救援，而哪吒在接收到求救信号后前往救援**。



在救援的过程中，**信号**无疑是关键的。如果把大乔和哪吒看作两个线程，那么他们在求救、救援过程中的信号就可以看作是**信号量，用于线程间的同步和通信**。

接下来，我们写一个简单的信号量，模拟还原刚才的求救和施救的过程。

定义一个求救的信号量，里面包含**信号**、**信号发送**和**信号接收**。w m

```
// 求救信号
public class ForHelpSemaphore {
    private boolean signal = false;

    public synchronized void sendSignal() {
        this.signal = true;
        this.notify();
        System.out.println("呼救信号已经发送！");
    }

    public synchronized void receiveSignal() throws
InterruptedException {
        System.out.println("已经就绪，等待求救信号...");
        while (!this.signal) {
            wait();
        }
        this.signal = false;
        System.out.println("求救信号已经收到，正在前往救援！");
    }
}
```

再创建两个线程，分别代表大乔和哪吒。

```
 public static void main(String[] args) {
   ForHelpSemaphore helpSemaphore = new ForHelpSemaphore();

   Thread 大乔 = new Thread(helpSemaphore::sendSignal);
   Thread 哪吒 = new Thread(() -> {
     try {
       helpSemaphore.receiveSignal();
     } catch (InterruptedException e) {
       e.printStackTrace();
     }
   });

   大乔.start();
   哪吒.start();
 }
```

从运行结果中可以看到，他们通过信号量的机制完成了救援行动。

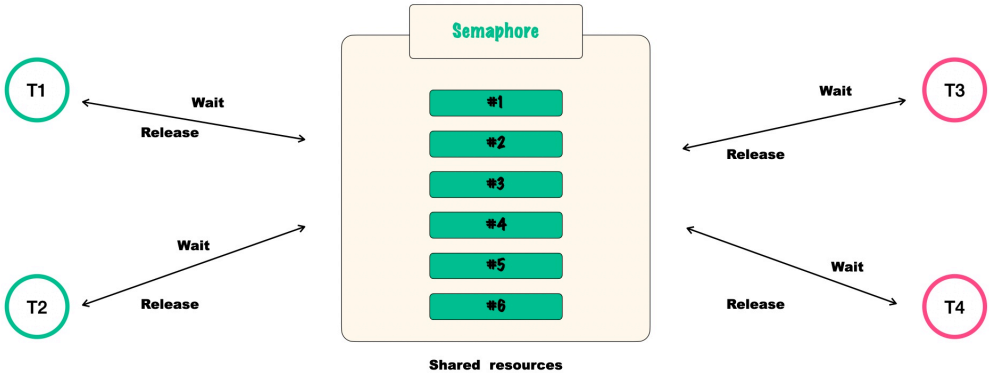
你看，最简单的信号量就是这样的简单。

二、理解宽泛意义上的的信号量

如果把上面大乔和哪吒救援的例子做个梳理的话，可以发展信号量中的一些关键信息：

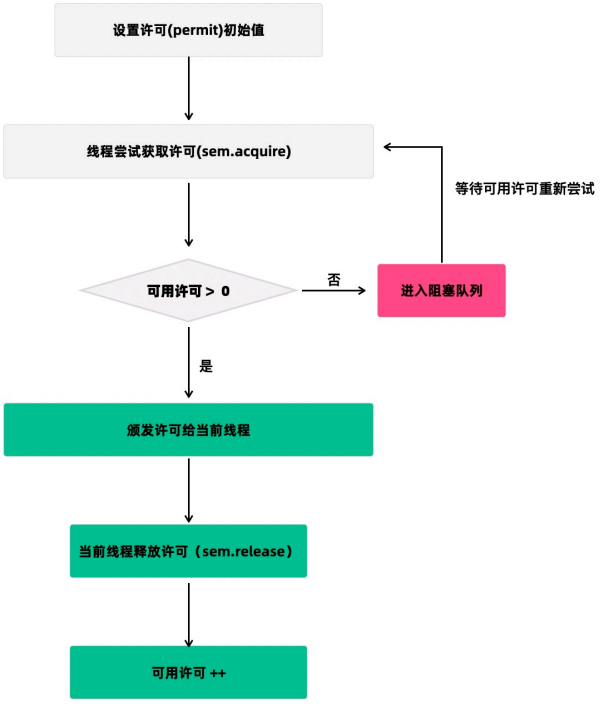
- **共享的资源**。比如`signal`字段是两个线程共享的，它是两个线程协同的基础；
- **多个线程访问相同的共享资源，并根据资源状态采取行动**。比如大乔和哪吒都会读写`signal`字段，然后采取行动。

基于上面的两点理解，我们可以把信号量抽象为下面这张图所示：



从图中可以看到，**多个线程共享一份资源列表，但是资源是有限的**。所以，**线程之间必然要按照一定的顺序有序地访问资源，并在访问结束后释放资源。没有获得资源的线程，只能等待其他线程释放资源后再次尝试获取**。

多线程对共享资源的访问过程，也可以用下面这张流程图表示：



如果你能把这两幅图理解了，那么你也就把信号量的机制理解了。而一旦理解了机制，所谓的源码不过只是某种具体的实现。

三、认识不同类型的信号量

根据信号量的机制和应用场景，一般有下面几种不同类型的信号量。

1. 计数型信号量

```
public class CountingSemaphore {
  private int signals = 0;
  public synchronized void take() {
    this.signals++;
    this.notify();
  }
  public synchronized void release() throws InterruptedException
{
    while (this.signals == 0)
      wait();
    This.signals--;
```

```
  }
}
```

2. 边界型信号量

在计数型信号量中，信号的数量是没有限制的。换句话说，所有的线程都可以发送信号。与此不同的是，在边界型信号量中，通过`bound`字段增加了信号量的限制。

```
public class BoundedSemaphore {
  private int signal = 0;
  private int bound = 0;

  public BoundedSemaphore(int upperBound) {
    this.bound = upperBound;
  }
  public void synchronized take() throws InterruptedException {
    while (this.signal == bound)
      wait();
    this.signal++;
    this.notify++;
  }
  public void synchronized release() throws InterruptedException
{
    while (this.signal == 0)
      wait();
    this.signal--;
  }
}
```

3. 定时型信号量

定时型（timed）信号量指的是允许线程在指定的时间周期内才能执行任务。时间周期结束后，定时器将会重置，所有的许可也都会被回收。

4. 二进制型信号量

二进制信号量和计数型信号量类似，但许可的值只有0和1两种。实现二进制型信号

量相对也是比较容易的，如果是1就是成功，否则是0就是失败。

四、Java中的信号量

在理解了信号量机制并且也理解它很有用之后，先不用着急实现它。在Java中，已经提供了相应的信号量工具类，即`java.util.concurrent.Semaphore`。并且，Java中的信号量实现已经比较全面，你不需要再重写它。

1. Semaphore的核心构造

Semaphore类有两个核心构造：

1. `Semaphore(int num)`
2. `Semaphore(int num, boolean fair)`

其中，`num`表示的是允许访问共享资源的线程数量，而布尔类型的`fair`则表示线程等待时是否需要考虑公平。

2. Semaphore的核心方法

1. `acquire()`: 获取许可，如果当前没有可用的许可，将进入阻塞等待状态；
2. `tryAcquire()`：尝试获取许可，无论有没有可用的许可，都会立即返回；
3. `release()`: 释放许可；
4. `availablePermits()`:返回可用的许可数量。

五、如何通过信号量实现锁的能力

在上面的示例中，由于信号量可以用于保护多线程对共享资源的访问，所以直觉你可能会觉得它像一把锁，而事实上信号量确实可以用于实现锁的能力。

比如，借助于边界信号量，我们把线程访问的上限设置为1，那么此时将只有1个线程可以访问共享资源，而这不就是锁的能力嘛！

下面是通过信号量实现锁的一个示例：

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);
...
semaphore.take();
try {
  //临界区
} finally {
  semaphore.release();
}
```

我们把信号量中的信号数量上限设置为1，代码中的`take()`就相当于`lock()`，而`release()`则相当于`unlock()`。如此，**信号量摇身一变就成了名副其实的锁**。

小结

以上就是关于信号量的全部内容。在本文中，我们介绍了信号量的概念、运行机制、信号量的几种类型、Java中的信号量实现，以及如果通过信号量实现一把锁。

理解信号量的关键在于理解它的概念，也就是它所要解决的问题和它的方案。在理解概念和机制之后，再去看Java中的源码时，就会发现原来如此，又是队列...

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 基于对信号量的理解，尝试自己实现一个简单的信号量。

延伸阅读与参考资料

- Semaphores
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

铂金05：致胜良器-无处不在的“阻塞队列”究竟是何面目

欢迎来到《王者并发课》，本文是该系列文章中的**第18篇**。

在线程的同步中，**阻塞队列**是一个绕不过去的话题，它是同步器底层的关键。所以，我们在本文中将为你介绍阻塞队列的基本原理，以了解它的工作机制和它在Java中的实现。本文稍微有点长，建议先了解大纲再细看章节。

一、阻塞队列介绍

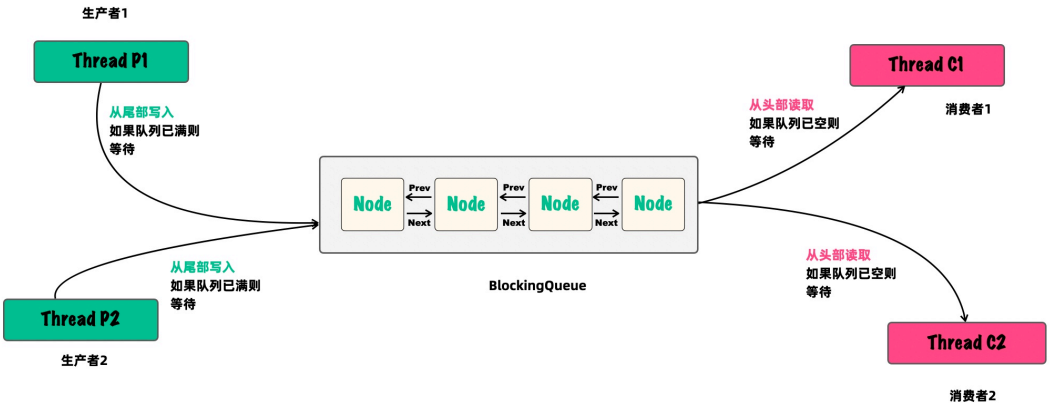
在生活中，相信你一定见过下图的人山人海，也见过其中的秩序井然。**混乱，是失控的开始**。想想看，在没有秩序的情况下，拥挤的人流蜂拥而上十分危险，轻则挤出一身臭汗，重则造成踩踏事故。**而秩序，则让情况免于混乱，排好队大家都舒服**。

面对人流，我们通过排队解决混乱。而面对多线程，我们也通过队列让线程间免于混乱，这就是阻塞队列为何而存在。

所谓阻塞队列，你可以理解它是这样的一种队列：

- **当线程试着往队列里放数据时，如果它已经满了，那么线程将进入等待；**
- **而当线程试着从队列里取数据时，如果它已经空了，那么线程将进入等待。**

下面这张图展示了多线程是如何通过阻塞队列进行协作的：



从图中可以看到，对于阻塞队列数据的读写并不局限于单个线程，往往存在多个线程的竞争。

二、实现简单的阻塞队列

接下来我们先抛开JUC中复杂的阻塞队列，来设计一个简单的阻塞队列，以了解它的核心思想。

在下面的阻塞队列中，我们设计一个队列`queue`，并通过`limit`字段限定它的容量。`enqueue()`方法用于向队列中放入数据，如果队列已满则等待；
而`dequeue()`方法则用于从数据中取出数据，如果队列为空则等待。

```
public class BlockingQueue {
    private final List<Object> queue = new LinkedList<>();
    private final int limit;

    public BlockingQueue(int limit) {
        this.limit = limit;
    }

    public synchronized void enqueue(Object item) throws
InterruptedException {
        while (this.queue.size() == this.limit) {
```

```
            print("队列已满，等待中...");
            wait();
        }
        this.queue.add(item);
        if (this.queue.size() == 1) {
            notifyAll();
        }
        print(item, "已经放入！");
    }


    public synchronized Object dequeue() throws
InterruptedException {
        while (this.queue.size() == 0) {
            print("队列空的，等待中...");
            wait();
        }
        if (this.queue.size() == this.limit) {
            notifyAll();
        }
        Object item = this.queue.get(0);
        print(item, "已经拿到！");
        return this.queue.remove(0);
    }

    public static void print(Object... args) {
        StringBuilder message = new StringBuilder(getThreadName()
+ ":");
        for (Object arg : args) {
            message.append(arg);
        }
        System.out.println(message);
    }

    public static String getThreadName() {
        return Thread.currentThread().getName();
    }
}
```

定义**lanLingWang**线程向队列中放入数据，**niumo**线程从队列中取出数据。

```
  public static void main(String[] args) {
    BlockingQueue blockingQueue = new BlockingQueue(1);
    Thread lanLingWang = new Thread(() -> {
```

```
        try {
          String[] items = { "A", "B", "C", "D", "E" };
          for (String item: items) {
            Thread.sleep(500);
            blockingQueue.enqueue(item);
          }
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      });
      lanLingWang.setName("兰陵王");
      Thread niumo = new Thread(() -> {
        try {
          while (true) {
            blockingQueue.dequeue();
            Thread.sleep(1000);
          }
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      });
      lanLingWang.setName("兰陵王");
      niumo.setName("牛魔王");

      lanLingWang.start();
      niumo.start();
    }
```

运行结果如下:

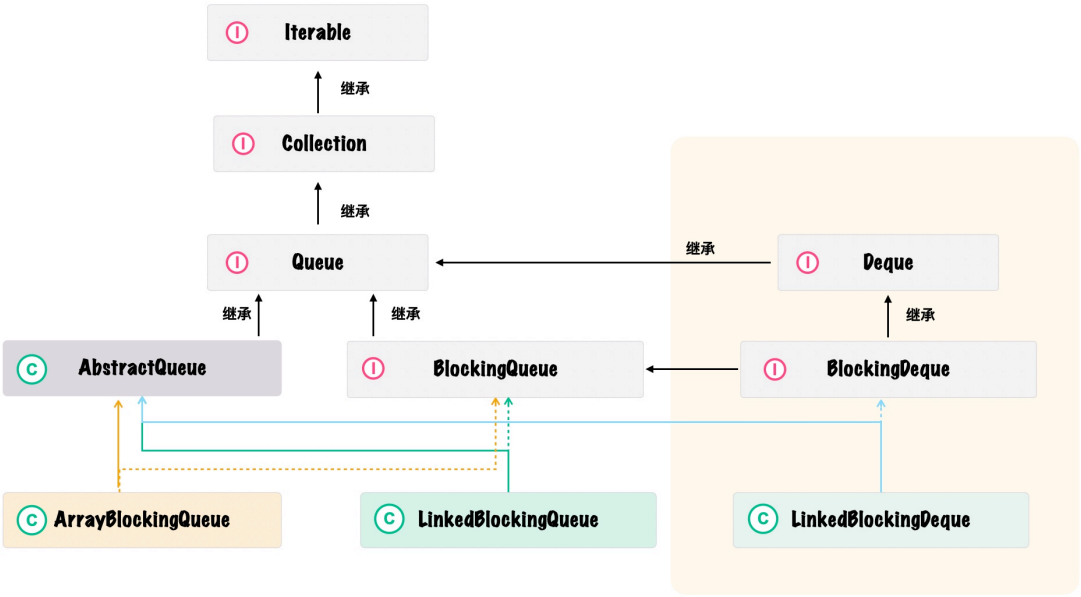
```
牛魔王:队列空的，等待中...
兰陵王:A已经放入!
牛魔王:A已经拿到!
兰陵王:B已经放入!
牛魔王:B已经拿到!
兰陵王:C已经放入!
兰陵王:队列已满，等待中...
牛魔王:C已经拿到!
兰陵王:D已经放入!
兰陵王:队列已满，等待中...
牛魔王:D已经拿到!
兰陵王:E已经放入!
牛魔王:E已经拿到!
牛魔王:队列空的，等待中...
```

从结果中可以看到，设计的阻塞队列已经可以有效工作，你可以仔细地品一品输出的结果。当然，这个阻塞是极其简单的，在下面一节中，我们将介绍Java中的阻塞队列设计。

三、Java中的BlockingQueue

Java中的阻塞队列有两个核心接口：**BlockingQueue**和**BlockingDeque**，相关的接口实现设继承关系如下图所示。相比于上一节中我们自定义的阻塞队列，Java中的实现要复杂很多。不过，你不必为此担心，**理解阻塞队列最重要的是理解它的思想和实现的思路，况且Java中的实现其实很有意思，读起来也比较轻松。**

从图中可以看出，BlockingQueue接口继承了**Queue**接口和**Collection**接口，并有LinkedBlockingQueue和ArrayBlockingQueue两种实现。这里有个有意思的地方，**继承Queue接口很容易理解，可以为什么要继承Collection接口？先卖个关子，你可以思考一会，稍后会给出答案。**



1. 核心方法

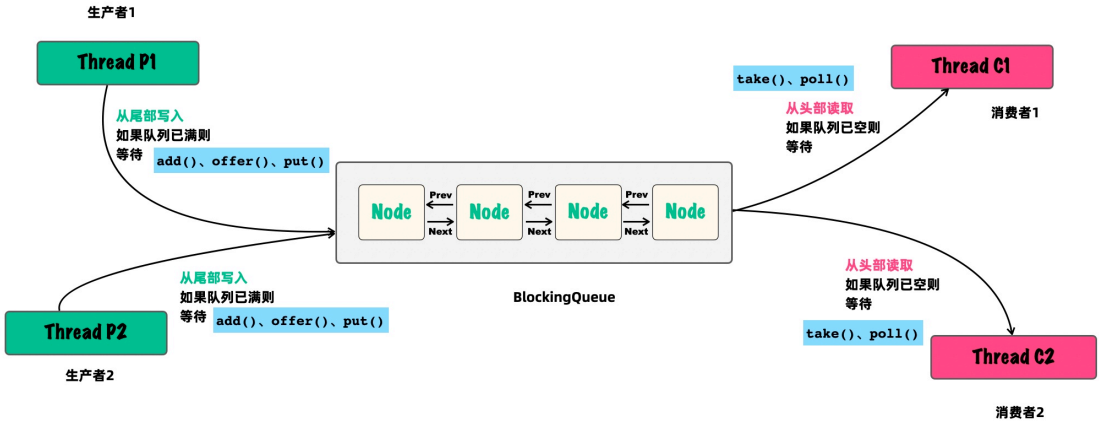
BlockingQueue中义了关于阻塞队列所需要的一系列方法，它们彼此之间看起来很像，从表面上看不出明显的差别。对于这些方法，你不必死记硬背，下图的表格中将这些方法分为了**A**、**B**、**C**、**D**这四种类型，分类之后再去理解它们会容易很多：

类型	A 抛出异常	B 返回特定值	C 阻塞	D 超时限定
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`Element()`	`peek()`	--	--

其中部分关键方法的解释如下：

- `add(E e)`：在不违反容量限制的前提下，向队列中插入数据。**如果成功，返回true，否则抛出异常**；
- `offer(E e)`：在不违反容量限制的前提下，向队列中插入数据。**如果成功，返回`true`，否则返回`false`**；
- `offer(E e, long timeout, TimeUnit unit)`：如果队列中没有足够的空间，将等待一段时间；
- `put(E e)`：在不违反容量限制的前提下，向队列中插入数据。**如果没有足够的空间，将进入等待**；
- `poll(long timeout, TimeUnit unit)`：从队列的头部获取数据，并移除数据。如果没有数据的话，将会等待指定的时间；
- `take()`：从队列的头部获取数据并移除。如果没有可用数据，将进入等待

将这些方法填入前面的那张图，它应该长这样：



2. LinkedBlockingQueue

LinkedBlockingQueue实现了BlockingQueue接口，遵从先进先出（FIFO）的原则，提供了可选的有界阻塞队列（ Optionally Bounded ）的能力，并且是线程安全的。

- **核心数据结构**
  - `int capacity`: 设定队列容量；
  - `Node<E> head`: 队列的头部元素；
  - `Node<E> last`: 队列的尾部元素；
  - `AtomicInteger count`: 队列中元素的总数统计。

LinkedBlockingQueue的数据结构并不复杂，不过需要注意的是，数据结构中并不包含List，仅有`head`和`last`两个Node，设计上比较巧妙。

- **核心构造**
  - `LinkedBlockingQueue()`: 空构造；
  - `LinkedBlockingQueue(int capacity)`: 指定容量构造。
- **线程安全性**
  - `ReentrantLock takeLock`: 获取元素时的锁；
  - `ReentrantLock putLock`: 写入元素时的锁。

注意，LinkedBlockingQueue有两把锁，读取和写入的锁是分离的！这和下面的ArrayBlockingQueue并不相同。

下面截取了LinkedBlockingQueue中读写的部分代码，值得你仔细品一品。**品的时**

候，要重点关注两把锁的使用和读写时数据结构是如何变化的。

- 队列插入示例代码分析

```
 public boolean add(E e) {
        addLast(e);
        return true;
    }

    public void addLast(E e) {
        if (!offerLast(e))
            throw new IllegalStateException("Deque full");
    }

    public boolean offerFirst(E e) {
        if (e == null) throw new NullPointerException();
        Node<E> node = new Node<E>(e);
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return linkFirst(node);
        } finally {
            lock.unlock();
        }
    }
```

- 队列读取示例代码分析

```
 public E poll(long timeout, TimeUnit unit) throws
InterruptedException {
        return pollFirst(timeout, unit);
    }
public E pollFirst(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            E x;
            while ( (x = unlinkFirst()) == null) {
                if (nanos <= 0)
                    return null;
                nanos = notEmpty.awaitNanos(nanos);
```

```
            }
            return x;
        } finally {
            lock.unlock();
        }
    }
```

最后说下LinkedBlockingQueue为什么要继承Collection接口。我们知道，Collection接口有`remove()`这样的移除方法，而这些方法在队列中也是有使用场景的。比如，你把一个数据错误地放入了队列，或者你需要移除已经失效的数据，那么Collection的一些方法就派上了用场。

3. ArrayBlockingQueue

ArrayBlockingQueue是BlockingQueue接口的另外一种实现，**它与LinkedBlockingQueue在设计目标上的的关键不同，在于它是有界的**。

- **核心数据结构**
  - `Object[] items`: 队列元素集合；
  - `int takeIndex`: 下次获取数据时的索引位置；
  - `int putIndex`: 下次写入数据时的索引位置；
  - `int count`: 队列总量计数。

从数据结构中可以看出，ArrayBlockingQueue使用的是数组，而数组是有界的。

- **核心构造**
  - `ArrayBlockingQueue(int capacity)`： 限定容量的构造；
  - `ArrayBlockingQueue(int capacity, boolean fair)`： 限定容量和公平性，默认是不公平的；
  - `ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)`：带有初始化队列元素的构造。
- **线程安全性**
  - `ReentrantLock lock`：队列读取和写入的锁。

在读写锁方面，前面已经说过，LinkedBlockingQueue和ArrayBlockingQueue是不同的，ArrayBlockingQueue只有一把锁，读写用的都是它。

- **队列写入示例代码分析**

```
 public boolean offer(E e) {
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            if (count == items.length)
                return false;
            else {
                enqueue(e);
                return true;
            }
        } 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++;
        notEmpty.signal();
    }
```

下面截取了ArrayBlockingQueue中读写的部分代码，值得你仔细品一品。**品的时候，要重点关注读写锁的使用和读写时数据结构是如何变化的。**

- **队列读取示例代码分析**

```
 public E poll(long timeout, TimeUnit unit) throws
InterruptedException {
     long nanos = unit.toNanos(timeout);
     final ReentrantLock lock = this.lock;
     lock.lockInterruptibly();
```

```
        try {
            while (count == 0) {
                if (nanos <= 0)
                    return null;
                nanos = notEmpty.awaitNanos(nanos);
            }
            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;
    }
```

四、Java中的BlockingDeque

在Java中，BlockingDeque与BlockingQueue是一对孪生兄弟似的存在，它们长得实在太像了，不注意的话很容易混淆。

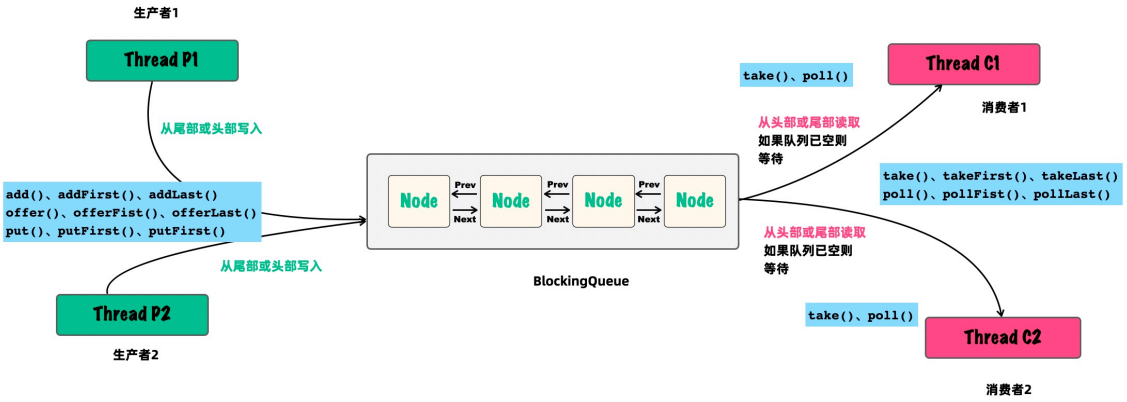
但是，BlockingDeque与BlockingQueue核心不同在于，BlockingQueue只能够从尾部写入、从头部读取，使用上很有限制。而BlockingDeque则支持从任意端读写，在读写时可以指定头部和尾部，丰富了阻塞队列的使用场景。

1. 核心方法

相较于BlockingQueue，BlockingDeque的方法显然要更丰富一些，毕竟它支持了双端的读写。**但是，丰富归丰富，在类型上仍然和BlockingQueue是一致的，你仍然可以参考上面的A、B、C、D四种类型来分类理解**。为了节约篇幅，我们这里就不再罗列，只选取了其中的部分方法作了解释：

- `add(E e)`：在不违反容量限制的前提下，在队列的尾部插入数据；
- `addFirst(E e)`：从头部插入数据，容量不够就抛错；
- `addLast(E e)`：从尾部插入数据，容量不够就抛错；
- `getFirst()`：从头部读取数据；
- `getLast()`：从尾部读取数据，但不会移除数据；
- `offer(E e)`：写入数据；
- `offerFirst(E e)`：从头部写入数据。

将BlockingDeue放入前面的那张图，就是这样：



2. LinkedBlockingDeue

LinkedBlockingDeue是BlockingDeque的核心实现。

- **核心数据结构**
  - `int capacity`：容量设置；
  - `Node<E> head`：队列头部；
  - `Node<E> last`：队列尾部；
  - `int count`：队列计数。

- **核心构造**
  - `LinkedBlockingDeque()`： 空的构造；
  - `LinkedBlockingDeque(int capacity)`： 指定容量的构造；
  - `LinkedBlockingDeque(Collection<? extends E> c)`：构造时初始化队列。
- **线程安全性**
  - `ReentrantLock lock`：读写锁。**注意，读写用的是同一把锁。**

下面截取了LinkedBlockingDeue中读写的部分代码，值得你仔细品一品。**品的时候，要重点关注读写锁的使用和读写时数据结构是如何变化的**

- **队列插入示例代码分析**

```
public void addFirst(E e) {
    if (!offerFirst(e))
        throw new IllegalStateException("Deque full");
}
public boolean offerFirst(E e) {
    if (e == null) throw new NullPointerException();
    Node < E > node = new Node < E > (e);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return linkFirst(node);
    } finally {
        lock.unlock();
    }
}
```

- **队列读取示例代码分析**

```
public E pollFirst() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return unlinkFirst();
    } finally {
        lock.unlock();
```

```
    }
}
```

小结

以上就是关于阻塞队列的全部内容，相较于前面的系列文章，这次的内容明显增加了很多。**看起来很简单，但是不要小瞧它**。理解阻塞队列，首先要理解它所要解决的问题，以及它的接口设计。**接口的设计往往表示的是它所提供的核心能力，所以理解了接口的设计，就成功了一半**。

在Java中，从接口层面，阻塞队列分为BlockingQueue和BlockingDeque的两大类，其主要差异在于双端读写的限制不同。其中，BlockingQueue有LinkedBlockingDeue和ArrayBlockingQueue两种关键实现，而BlockingDeque则有LinkedBlockingDeue实现。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 从数据机构、队列的初始化、锁、性能等方面比较LinkedBlockingDeue和ArrayBlockingQueue的不同。

延伸阅读与参考资料

- **Talk about LinkedBlockingQueue**
- **Blocking Queues**
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金06：青出于蓝-Condition如何把等待与通知玩出新花样

欢迎来到《王者并发课》，本文是该系列文章中的**第19篇**。

在上一篇文章中，我们介绍了阻塞队列。如果你阅读过它的源码，那么你一定会注意到源码有两个Condition类型的变量：`notEmpty`和`notFull`，在读写队列时你也会注意到它们是如何被使用的。事实上，在使用JUC中的各种锁时，Condition都很有用场，你很有必要了解它。所以，本文就为你介绍它的来龙去脉和用法。

在前面的系列文章中，我们多次提到过`synchronized`关键字，相信你已经对它的用法了如于心。在多线程协作时，有两个另外的关键字经常和`synchronized`一同出现，它们相互配合，就是`wait`和`notify`，比如下面的这段代码：

```
public class CountingSemaphore {
  private int signals = 0;
  public synchronized void take() {
    this.signals++;
    this.notify();  // 发送通知
  }
  public synchronized void release() throws InterruptedException
{
    while (this.signals == 0)
      this.wait(); // 释放锁，进入等待
      this.signals--;
  }
}
```

`synchronized`是Java的原生同步工具，`wait`和`notify`是它的原生搭档。然而，在铂金系列中，我们已经介绍了Lock接口和它的一些实现，比如可重入锁ReentrantLock等。相比于`synchronized`，JUC所封装的这些锁工具在功能上要丰富得多，也更加容易使用。所以，相应的配套自然也要跟上，于是Condition就应运而生。

比如在上文的阻塞队列中，Condition就已经闪亮登场：

```
public class LinkedBlockingQueue < E > extends AbstractQueue < E
>
    implements BlockingQueue < E > , java.io.Serializable {

        ...省略源码若干

        // 定义Condition
        // 注意，这里定义两个Condition对象，用于唤醒不同的线程
        private final Condition notEmpty =
takeLock.newCondition();
        private final Condition notFull = putLock.newCondition();

        public E take() throws InterruptedException {
            E x;
            int c = -1;
            final AtomicInteger count = this.count;
            final ReentrantLock takeLock = this.takeLock;
            takeLock.lockInterruptibly();
            try {
                while (count.get() == 0) {
                    // 进入等待
                    notEmpty.await();
                }
                x = dequeue();
                c = count.getAndDecrement();
                if (c > 1)
                    notEmpty.signal();
            } finally {
                takeLock.unlock();
            }
            if (c == capacity)
                signalNotFull();
            return x;
        }

        private void signalNotEmpty() {
            final ReentrantLock takeLock = this.takeLock;
            takeLock.lock();
            try {
                // 发送唤醒信号
                notEmpty.signal();
            } finally {
                takeLock.unlock();
            }
```

```
        }
        ...省略源码若干

    }
```

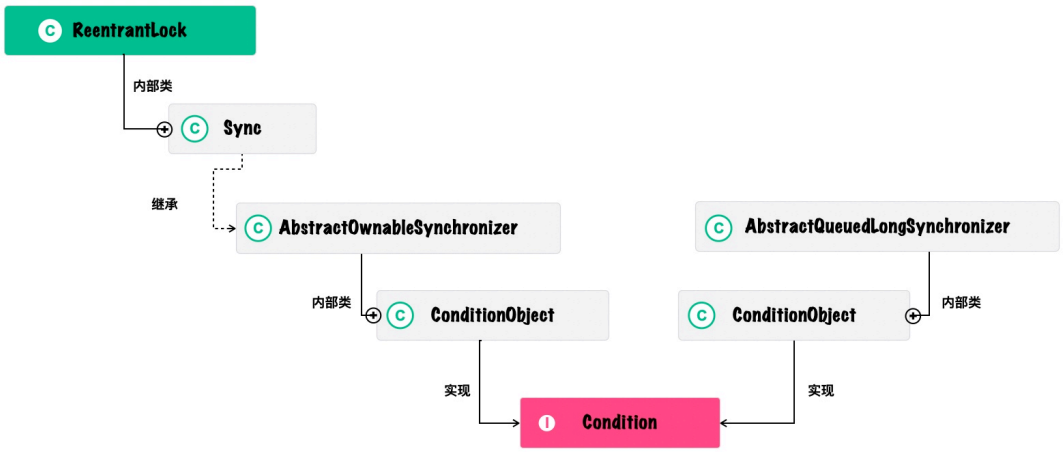
从功能定位上说，作为Lock的配套工具，Condition是`wait`、`notify`和`notifyAll`增强版本，`wait`和`notify`有的能力它都有，`wait`和`notify`没有的能力它也有。

JUC中的Condition是以接口的形式出现，并定义了一些核心方法：

- `await()`：让当前线程进入等待，直到收到信号或者被中断；
- `await(long time, TimeUnit unit)`：让当前线程进入等待，直到收到信号或者被中断，或者到达指定的等待超时时间；
- `awaitNanos(long nanosTimeout)`：让当前线程进入等待，直到收到信号或者被中断，或者到达指定的等待超时时间，只是在时间单位上和上一个方法有所区别；
- `awaitUninterruptibly()`：**让当前线程进入等待，直到收到信号。注意，这个方法对中断是不敏感的**；
- `awaitUntil(Date deadline)`：**让当前线程进入等待，直到收到信号或者被中断，或者到达截止时间**；
- `signal()`：随机唤醒一个线程；
- `signalAll()`：唤醒所有等待的线程。

从Condition的核心方法中可以看到，相较于原生的通知与等待，它的能力明显增强了很多，比如`awaitUninterruptibly()`和`awaitUntil()`。另外，Condition竟然是可以唤醒指定线程的，这就很有意思。

作为接口，我们并不需要手动实现Condition，JUC已经提供了相关的实现，你可以在ReentrantLock中直接使用它。相关的类、接口之间的关系如下所示：



小结

以上就是关于Condition的全部内容。Condition并不复杂，它是JUC中Lock的配套，在理解时要结合原生的`wait`和`notify`去理解。关于Condition与它们之间的详细区别，已经都在下面的表格里：

对比项	Object's Monitor methods	Condition
前置条件	获取对象的锁	调用Lock获取锁，调用lock.newCondition()获取Condition对象
调用方式	直接调用，如object.wait()	直接调用，如condition.await()
等待队列个数	一个	**多个**
当前线程释放锁并进入等待状态	✔	✔
当前线程释放锁并进入等		

待状态，在等待时不响应中断	✘	✔
当前线程释放锁并进入超时等待	✔	✔
当前线程释放锁并进入等待到未来某个时刻	✘	✔
唤醒等待队列中的某一个线程	✔	✔
唤醒等待队列中的全部线程	✔	✔

理解表格中的各项差异，不要死记硬背，而是要基于Condition接口中定义的方法，从关键处理解它们的不同。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码使用Condition唤醒指定线程。

延伸阅读与参考资料

- 小结表格中的内容由网络图片提取，未能找到原始出处，知道的请评论告知，感谢！
- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金07：整齐划一--CountDownLatch如何协调多线程的开始和结束

欢迎来到《王者并发课》，本文是该系列文章中的**第20篇**。

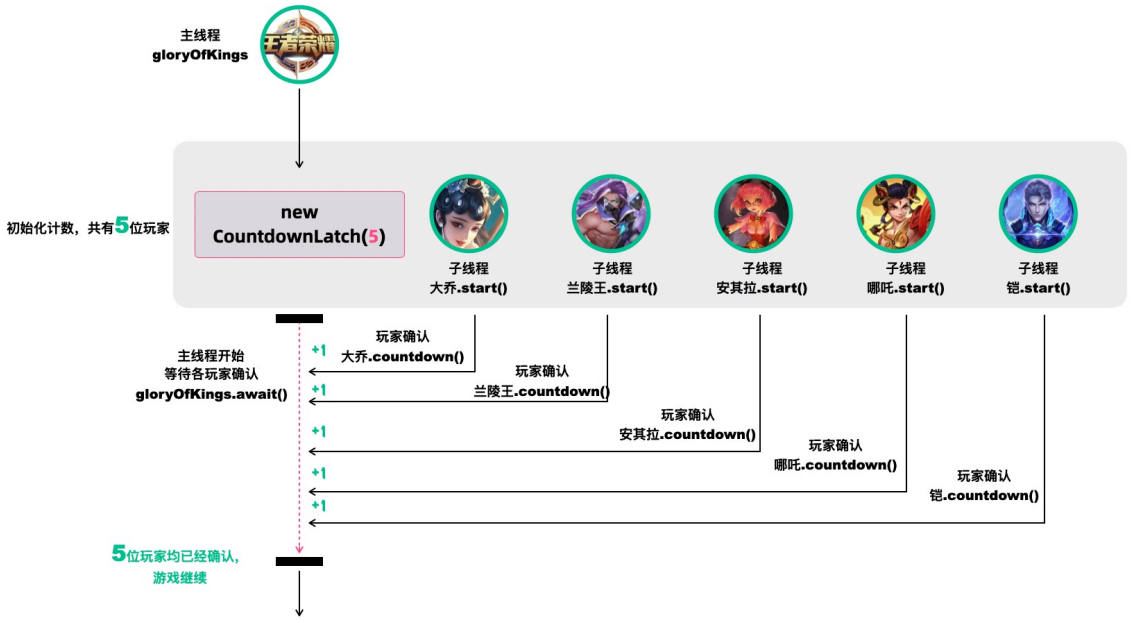
在上一篇文章中，我们介绍了Condition的用法。在本文中，将为你介绍CountDownLatch的用法。CountDownLatch是JUC中的一款常用工具类，当你在编写多线程代码时，**如果你需要协调多个线程的开始和结束动作时，它可能会是你的不错选择**。

一、CountDownLatch适用的两个典型应用场景

场景1. 协调子线程结束动作：等待所有子线程运行结束

对于资深的王者来说，下面这幅图一定再熟悉不过了。在王者开局匹配队友时，所有的玩家都必须进行确认操作，只有全部确认后才可以进入游戏，否则将进行重新匹配。**如果我们把各玩家看作是子线程的话，那么就需要各子线程完成确认动作，游戏才能继续**。其实，此类场景不止于王者，在生活中类似的场景还有很多。比如，所有乘客登机后飞机才能关舱门，除非超时后他们抛弃了你。

对于上图所示的玩家确认界面，如果用多线程模拟的话，那么它应该是下面的样子：



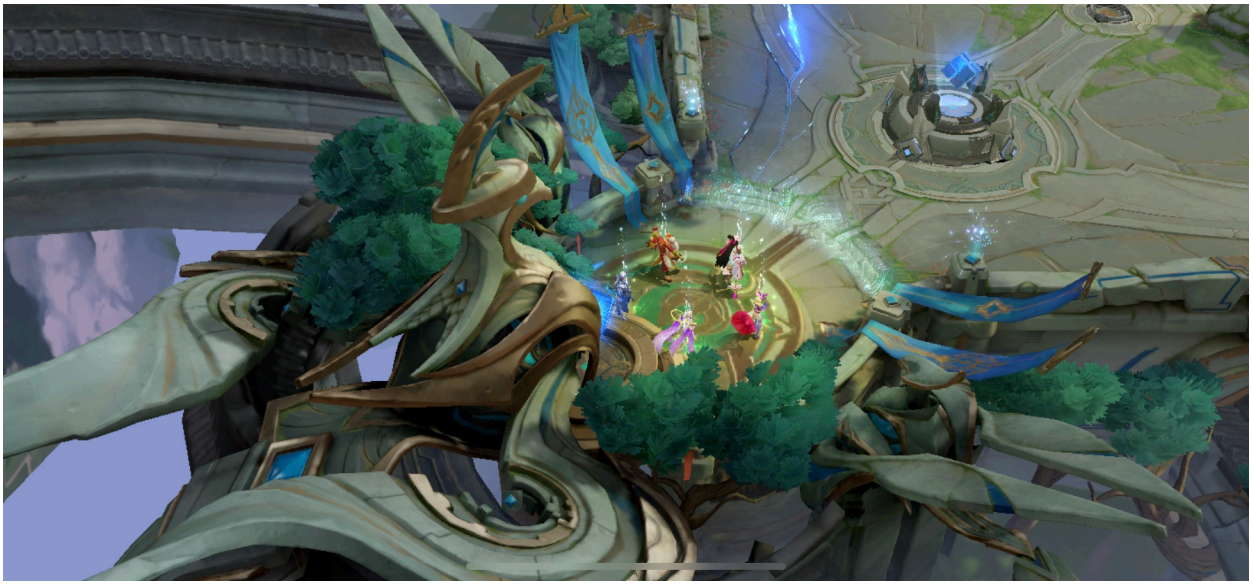
主线程创建了5个子线程，各子任务执行确认动作，**期间主线程进入等待状态，直到各子线程的任务均已经完成，主线程恢复继续执行**，也就是游戏继续。而如果其中某个玩家超时未执行确认的话，那么主线程将结束本次匹配，重新开始新一轮的匹配。

这个场景，就是CountDownLatch适用的第一个经典场景。

场景2. 协调子线程开始动作：统一各线程动作开始的时机

这个场景的例子也十分常见。比如，田径场上，各选手各就各位等待发令枪。在发令枪响之前，选手只能原地就位，否则就是违规。如果从多线程的角度看，这恰似你创建了一些多线程，但是你需要统一管理它们的任务开始时间。**因为，如果你不对此做干预的话，线程调用`start()`之后的具体时间是不确定的，这个知识点我们早在青铜系列文章中就已经讲过。**

在王者中也有类似的场景，游戏开始时，各玩家的初始状态必须一致。**总不能，你刚降生，对方已经打到你家门口了。**



上述的两个场景的问题，正是CountDownLatch所要解决的问题。理解了这两个问题，你也就理解了CountDownLatch存在的价值。

二、Java中的CountDownLatch设计

JUC中CountDownLatch的实现，是以类的形式存在，而不是接口，你可以直接拿过来使用。并且，它的实现还很简单。在数据结构上，CountDownLatch基于一个同步器实现，你可以看它的`final Sync sync`变量。

而在构造函数上，CountDownLatch有且只有`CountDownLatch(int count)`一个构造器，并且你需要指定数量，**并且你不得在中途修改它，这点务必牢记**!

核心函数

- `await()`：等待latch降为0；
- `boolean await(long timeout, TimeUnit unit)`：等待latch降为0，但是可以设置超时时间。比如有玩家超时未确认，那就重新匹配，总不能为了某个玩家等到天荒地老吧。
- `countDown()`：latch数量减1；
- `getCount()`：获取当前的latch数量。

从CountDownLatch的方法上看，还是比较简单易懂的，重点要理解`await()`和`countDown()`.

三、CountDownLatch如何解决场景问题

接下来，我们将第一小节的两个场景问题用代码实现一遍，让你对CountDownLatch的用法有个直观的理解。

场景1. CountDownLatch实现对各子线程的等待

创建大乔、兰陵王、安其拉、哪吒和铠等五个玩家，主线程必须在他们都完成确认后，才可以继续运行。

在这段代码中，`new CountDownLatch(5)`用户创建初始的latch数量，各玩家通

过`countDownLatch.countDown()`完成状态确认。

```
public static void main(String[] args) throws
InterruptedException {
    CountDownLatch countDownLatch = new CountDownLatch(5);

    Thread 大乔 = new Thread(countDownLatch::countDown);
    Thread 兰陵王 = new Thread(countDownLatch::countDown);
    Thread 安其拉 = new Thread(countDownLatch::countDown);
    Thread 哪吒 = new Thread(countDownLatch::countDown);
    Thread 铠 = new Thread(() -> {
        try {
            // 稍等，上个卫生间，马上到...
            Thread.sleep(1500);
            countDownLatch.countDown();
        } catch (InterruptedException ignored) {}
    });

    大乔.start();
    兰陵王.start();
    安其拉.start();
    哪吒.start();
    铠.start();
    countDownLatch.await();
    System.out.println("所有玩家已经就位！");
}
```

场景2. CountDownLatch实现对多线程的统一管理

在这个场景中，我们仍然用五个线程代表大乔、兰陵王、安其拉、哪吒和铠等五个玩家。**需要注意的是，各玩家虽然都调用了`start()`线程，但是它们在运行时都在等待countDownLatch的信号，在信号未收到前，它们不会往下执行。**

```
public static void main(String[] args) throws
InterruptedException {
    CountDownLatch countDownLatch = new CountDownLatch(1);

    Thread 大乔 = new Thread(() -> waitToFight(countDownLatch));
    Thread 兰陵王 = new Thread(() -> waitToFight(countDownLatch));
    Thread 安其拉 = new Thread(() -> waitToFight(countDownLatch));
    Thread 哪吒 = new Thread(() -> waitToFight(countDownLatch));
    Thread 铠 = new Thread(() -> waitToFight(countDownLatch));
```

```
        大乔.start();
        兰陵王.start();
        安其拉.start();
        哪吒.start();
        铠.start();
        Thread.sleep(1000);
        countDownLatch.countDown();
        System.out.println("敌方还有5秒达到战场，全军出击！");
    }

    private static void waitToFight(CountDownLatch countDownLatch) {
        try {
            countDownLatch.await(); // 在此等待信号再继续
            System.out.println("收到，发起进攻！");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

运行结果如下，各玩家在收到出击信号发起了进攻：

```
敌方还有5秒达到战场，全军出击!
收到，发起进攻!
收到，发起进攻!
收到，发起进攻!
收到，发起进攻!
收到，发起进攻!

Process finished with exit code 0
```

小结

以上就是关于CountDownLatch的全部内容。总体上，CountDownLatch比较简单且易于理解。在学习时，先了解其设计意图，再写个Demo基本就能流畅掌握。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码体验CountDownLatch用法。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金08：峡谷幽会-看CyclicBarrier如何跨越重峦叠嶂

欢迎来到《王者并发课》，本文是该系列文章中的**第21篇**，铂金中的**第8篇**。

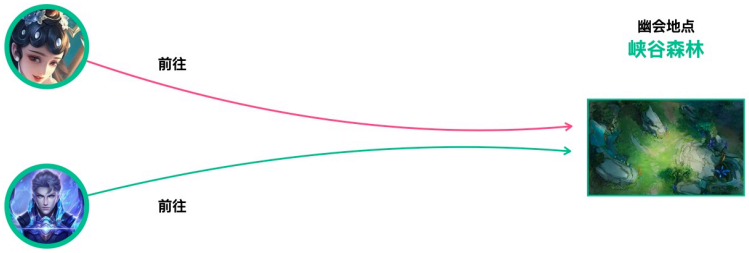
在上一篇文章中，我们介绍了CountDownLatch的用法。在协调多线程的开始和结束时，CountDownLatch是个非常不错的选择。而本文即将给你介绍的CyclicBarrier则更加有趣，它在能力上和CountDownLatch既有相似之处，又有着明显的不同，值得你一览究竟。本文会先从场景上带你理解问题，再去理解CyclicBarrier提供的方案。

一、CyclicBarrier初体验

1. 峡谷森林里的爱情

在峡谷的江湖中，不仅有生杀予夺和刀光剑影，还有着美妙的爱情故事。

峡谷战神铠曾经在危急关头救了大乔，这一出英雄救美让他们擦除了爱情的火花，有事没事两人就在峡谷中的各个角落幽会。其中，**峡谷森林**就是他们常去的地方，**谁先到就等另一个，两人都到齐后，再一起玩耍、切磋武艺**。



这里头，有两个重点。一是他们要**相互等待**，二是都到齐后再**玩耍**。现在，我们试

想一下，如果用代码来模拟这个场景的话，你打算怎么做。有的同学可能会说，两个人（线程）的等待很好处理。**可是，如果是三人呢**?

所以，这个场景问题可以概括为：**多个线程相互等待，到齐后再执行特定动作**。

接下来，我们就通过CyclicBarrier来模拟解决这个场景的问题，直观感受CyclicBarrier的用法。

在下面这段代码中，我们定义了一个**幽会地点（appointmentPlace）**，以及**大乔**和**铠**这两个主人公。在他们都达到幽会地点后，我们输出一句包含三朵玫瑰🌹🌹🌹的话来予以确认，给他们送上祝福。

```
  private static String appointmentPlace = "峡谷森林";

  public static void main(String[] args) {
    CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () ->
print("🌹🌹🌹到达约会地点：大乔和铠都来到了👉" + appointmentPlace));
    Thread 大乔 = newThread("大乔", () -> {
      say("铠，你在哪里...");
      try {
        cyclicBarrier.await(); //到达幽会地点
        say("铠，你终于来了...");
      } catch (Exception e) {
        e.printStackTrace();
      }
    });

    Thread 铠 = newThread("铠", () -> {
      try {
        Thread.sleep(500); //铠打野中
        say("我打个野，马上就到!");
        cyclicBarrier.await(); //到达幽会地点
        say("乔，不好意思，刚打野遇上兰陵王了，你还好吗？！");
      } catch (Exception e) {
        e.printStackTrace();
      }
    });

    大乔.start();
    铠.start();
  }
```

输出结果如下：

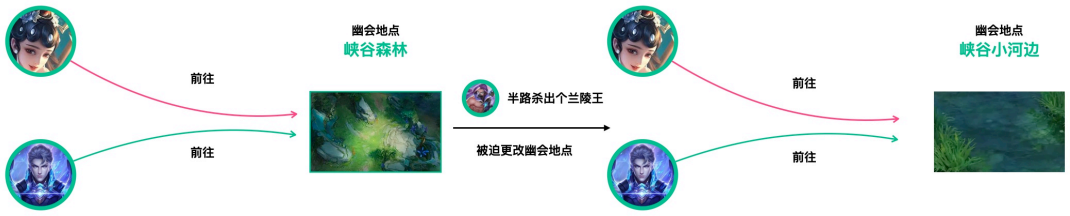
```
大乔:铠，你在哪里...
铠:我打个野，马上就到!
🌹🌹🌹到达约会地点：大乔和铠都来到了👉峡谷森林
铠:乔，不好意思，刚打野遇上兰陵王了，你还好吗？！
大乔:铠，你终于来了...

Process finished with exit code 0
```

对于代码的细节暂且不必深究，本文后面对CyclicBarrier的内部细节会有详解，先感受它的基本用法。

从结果中可以看到，**CyclicBarrier可以像CountDownLatch一样，协调多线程的执行结束动作，在它们都结束后执行特定动作**。从这点上来说，这是CyclicBarrier与CountDownLatch相似之处。然而，接下来的这个场景，所体现的则是它们一个明显的不同之处。

2. 小河边的幽会

在上面的场景中，铠已经提到他在打野时遇到了兰陵王。而在铠与大乔的约会中，兰陵王竟然又撞见了他们，真是冤家路窄。于是，在兰陵王的搅局下，铠和大乔不得不转移阵地，**他们同样约定到新的约定地点后等待对方。（铠一直以为兰陵王也喜欢大乔，要和他横刀夺爱，其实兰陵王在乎的只是铠打了它的野，他的心里只有野怪，对任何女人毫无兴趣）**。



此时，如果继续用代码模拟这一场景的话，那么CountDownLatch就无能为力了，**因为CountDownLatch的使用是一次性的**，无法重复利用。而此时，**你就会发现CyclicBarrier的神奇之处，它竟然可以重复利用**。似乎，你可能已经大概明白它为

什么叫**Cyclic**的原因了。

接下来，我们再走一段代码，模拟大乔和铠的第二次幽会。在代码中，我们仍然定义幽会地点、大乔和铠两个主人公。**但是与此前不同的是，我们还增加了兰陵王这个搅局者，以及中途变更了幽会地点。**

```
private static String appointmentPlace = "峡谷森林";

public static void main(String[] args) {
  CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () ->
System.out.println("🌹🌹🌹到达约会地点：大乔和铠都来到了👉" +
appointmentPlace));
  Thread 大乔 = newThread("大乔", () -> {
    say("铠，你在哪里...");
    try {
      cyclicBarrier.await();
      say("铠，你终于来了...");
      Thread.sleep(2600); //约会中...
      say("好的，你要小心！");
      cyclicBarrier.await(); // 注意这里是第二次调用await
      Thread.sleep(100);
      say("真好！");
    } catch (Exception e) {
      e.printStackTrace();
    }
  });

  Thread 铠 = newThread("铠", () -> {
    try {
      Thread.sleep(500); //铠打野中
      say("我打个野，马上就到!");
      cyclicBarrier.await(); //到达幽会地点
      say("乔，不好意思，刚打野遇上兰陵王了，你还好吗？！");
      Thread.sleep(1500); //幽会中...

      note("幽会中...\n");

      Thread.sleep(1000); //幽会中...
      say("这个该死的兰陵王！乔，你先走，小河边见！"); //铠突然看到了兰陵王
      appointmentPlace = "小河边"; // 铠把地点改成了小河边

      Thread.sleep(1500); //和兰陵王对决中...
      note("\uD83D\uDDE1\uD83D\uDD2A铠和兰陵王决战开始，最终铠杀死了兰陵
```

```
王，并前往小河边...\n");
      cyclicBarrier.await(); // 杀了兰陵王后，铠到了小河边 ！！！ 注意这
里是第二次调用await

      say("乔，我已经解决了兰陵王，你看今晚夜色多美，我陪你看星星到天
明...");
    } catch (Exception ignored) {}
  });

  Thread 兰陵王 = newThread("兰陵王", () -> {
    try {
      Thread.sleep(2500);
      note("兰陵王出场...");
      say("铠打了我的野，不杀他誓不罢休！ ");

      say("铠，原来你和大乔在这里！\uD83D\uDDE1 \uD83D\uDDE1 ");
    } catch (Exception ignored) {}
  });

  兰陵王.start();
  大乔.start();
  铠.start();
}
```

输出结果如下所示。**铠峡谷森林的好事被兰陵王搅局后，铠怒火中烧，让大乔先走，并约定在小河边碰面。随后，铠斩杀了兰陵王（可怜的钢铁直男），并前往小河边，完成他和大乔的第二次幽会。**

```
大乔:铠，你在哪里...
铠:我打个野，马上就到!
🌹🌹🌹到达约会地点：大乔和铠都来到了👉峡谷森林
铠:乔，不好意思，刚打野遇上兰陵王了，你还好吗？！
大乔:铠，你终于来了...
幽会中...

兰陵王出场...
兰陵王:铠打了我的野，不杀他誓不罢休!
兰陵王:铠，原来你和大乔在这里！🗡️🗡️
铠:这个该死的兰陵王！乔，你先走，小河边见!
大乔:好的，你要小心!
□🗡️🔪铠和兰陵王决战开始，最终铠杀死了兰陵王，并前往小河边...
```

```
🌹🌹🌹到达约会地点：大乔和铠都来到了👉小河边
铠:乔，我已经解决了兰陵王，你看今晚夜色多美，我陪你看星星到天明...
大乔:真好!

Process finished with exit code 0
```

同样的，你暂时不要理会代码的细节，**但是你要注意到其中铠和大乔对`await()`的两次调用**。在你没有理解它的原理之前，可能会惊讶于它的神奇，这是正常现象。

二、CyclicBarrier是如何实现的

CyclicBarrier是Java中提供的一个线程同步工具，与CountDownLatch相似，但又并不完全相同，**最核心的区别在于CyclicBarrier是可以循环使用的，这一点在它的名字中也已经有所体现**。

接下来，我们来分析下它具体的源码实现。

1. 核心数据结构

- `private final ReentrantLock lock = new ReentrantLock()`：进入屏障的锁，只有一把；
- `private final Condition trip = lock.newCondition()`：和上面的lock配套使用；
- `private final int parties`：参与方的数量，本文上述的例子只有铠和大乔，所以数量是2；
- `private final Runnable barrierCommand`：在本轮结束时运行的特定代码。本文上述例子用到了它，可以上翻查看；
- `private Generation generation = new Generation()`：当前屏障的代次。比如本文上述的两个场景中，generation是不同的，在铠和大乔将幽会地点改成小河边后，会生成新的generation；
- `private int count`：正在等待的参与方数量。在每个代次中，count会从最初的参与数量（即parties）降至0，到0时本代次结束，而在新的代次或本代次被拆除（broken）时，count的值会恢复为parties的值。

2. 核心构造

- `public CyclicBarrier(int parties)`：指定参与方的数量；
- `public CyclicBarrier(int parties, Runnable barrierAction)`：指定参与方的数量，并指定在本代次结束时运行的代码。

3. 核心方法

- `public int await()`：如果当前线程不是第一个到达屏障的话，它将会进入等待，直到其他线程都到达，除非发生**被中断**、**屏障被拆除**、**屏障被重设**等情况；
- `public int await(long timeout, TimeUnit unit)`：和await()类似，但是加上了时间限制；
- `public boolean isBroken()`：当前屏障是否被拆除；
- `public void reset()`：重设当前屏障。会先拆除屏障再设置新的屏障；
- `public int getNumberWaiting()`：正在等待的线程数量。

在CyclicBarrier的各方法中，最为核心的就是`dowait()`，两个`await()`的内部都是调用这个方法。所以，理解了`dowait()`，基本上就理解了CyclicBarrier的实现关键。

`dowait()`方法略长，稍微需要点耐心，我已经对其中部分做了注释。当然，如果你想看源码的话，还是建议直接从JDK中看它的全部，这里的源码只是为了辅助你理解上下文。

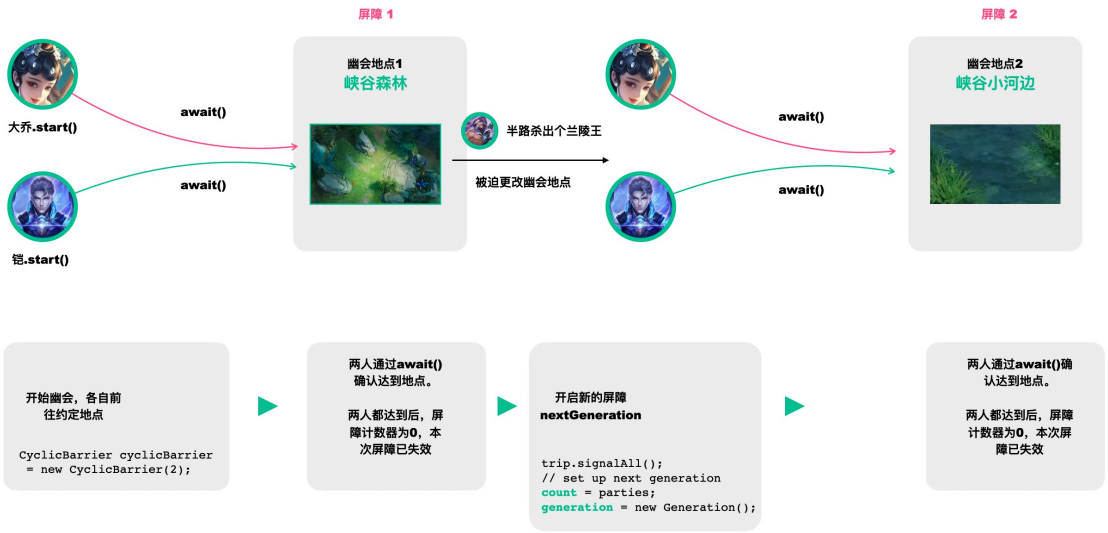
```
private int dowait(boolean timed, long nanos)
throws InterruptedException,
BrokenBarrierException,TimeoutException {
    final ReentrantLock lock = this.lock;
    lock.lock(); // 注意，这里是一定要加锁的
    try {
        final Generation g = generation;

        if (g.broken) // 如果当前屏障被拆除，则抛出异常
            throw new BrokenBarrierException();

        if (Thread.interrupted()) {
            breakBarrier(); // 如果当前线程被中断，则拆除屏障并抛出异常
            throw new InterruptedException();
        }
```

```
        int index = --count; // 当线程调用await后，count减1
        if (index == 0) { // tripped // 如果count为0，接下来将尝试结束
屏障，并开启新的屏障
            boolean ranAction = false;
            try {
                final Runnable command = barrierCommand;
                if (command != null)
                    command.run();
                ranAction = true;
                nextGeneration();
                return 0;
            } finally {
                if (!ranAction)
                    breakBarrier();
            }
        }

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

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

            if (g != generation)
                return index;

            if (timed && nanos <= 0 L) {
```

```
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

对于CyclicBarrier的核心数据结构、构造和方法，都在上面，它们很重要。但是，更为重要的是，要理解CyclicBarrier的思想，**也就是下面这幅值得你收藏的图。理解了这幅图，也就理解了CyclicBarrier**.

此时，从这幅图再回头看第一节的两个场景，铠和大乔先后在**峡谷森林**、**小河边**两个地点幽会。那么，如果也用一幅图来表示的话，它应该是下面这样：



三、CyclicBarrier与CountDownLatch有何不同

前面两节已经提到了两者的核心不同：

- **CountDownLatch是一次性的，而CyclicBarrier则可以多次设置屏障，实现重复利用**；
- **CountDownLatch中的各个子线程不可以等待其他线程，只能完成自己的任务；而CyclicBarrier中的各个线程可以等待其他线程。**

除此之外，它们俩还有着一些其他的不同，整体汇总后如下面的表格所示：

CyclicBarrier	CountDownLatch
CyclicBarrier是可重用的，其中的线程会等待所有的线程完成任务。届时，屏障将被拆除，并可以选择性地做一些特定的动作。	CountDownLatch是一次性的，不同的线程在同一个计数器上工作，直到计数器为0.
CyclicBarrier面向的是线程数	CountDownLatch面向的是任务数
在使用CyclicBarrier时，你必须在构造中指定参与协作的线程数，这些线程必须调用await()方法	使用CountDownLatch时，则必须要指定任务数，至于这些任务由哪些线程完成无关紧要
CyclicBarrier可以在所有的线程释放后重新使用	CountDownLatch在计数器为0时不能再使用
在CyclicBarrier中，如果某个线程遇到了中断、超时等问题时，则处于await的线程都会出现问题	在CountDownLatch中，如果某个线程出现问题，其他线程不受影响

小结

以上就是关于CyclicBarrier的全部内容。在学习CyclicBarrier时，要侧重理解它所要

解决的问题场景，以及它与CountDownLatch的不同，然后再去看源码，这也是为什么我们没有上来就放源码而是绕弯讲了个故事的原因，虽然那个故事挺“狗血”。当然，如果这个狗血的故事能让你记住这个知识点，狗血也值得了。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 编写代码体验CyclicBarrier用法。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

铂金09：互通有无-Exchange如何完成线程间的数据交换

欢迎来到《王者并发课》，本文是该系列文章中的**第22篇**，铂金中的**第9篇**。

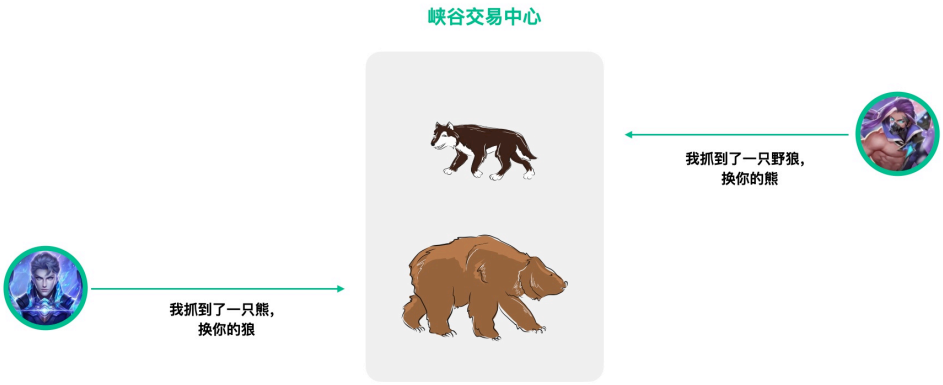
在前面的文章中，我们已经介绍了**ReentrantLock**，**CountDownLatch**，**CyclicBarrier**，**Semaphore**等同步工具。在本文中，将为你介绍最后一个同步工具，即**Exchanger**.

Exchanger用于两个线程在某个节点时进行数据交换。在用法上，Exchanger并不复杂，但是实现上会稍微有点费解。所以，考虑到Exchanger在平时使用的场景并不多，况且多数读者对一些“枯燥”的源码的耐受度有限（可能引起不适或烦躁等不良情绪，阻碍学习），本文将侧重讲它的使用和思想，对于源码不会过多展开，点到为止。

一、Exchanger的使用场景

在峡谷中，铠和兰陵王都是擅长打野的英雄，各自对野怪的偏好也不完全相同。所以，为了能得到自己想要的野怪，他们经常会在峡谷的交易中心**交换**各自的猎物。

这一天，铠打到了一只**棕熊**，而兰陵王则收获了一只**野狼**，并且彼此都想要对方的野怪。于是，他们约定在峡谷交易中心交换双方的野怪，谁先到了就先等会。这个过程，可以用下面这幅图来表示：



在铠和兰陵王交换猎物的过程中，有三个点需要你留意：

- **交换的双方有明确的交易地点**（峡谷交易中心）；
- **交换的双方具有明确的交易对象**（比如棕熊和野狼）；
- **谁先到了就等会儿**（他们中总会有先来后到）。

如果用代码来实现的话，也是有多种方式可以选择，比如前面所学过的同步方法等。不过，虽然做也是可以做的，只是没那么方便。所以，接下来我们就用**Exchanger**来实现这一过程。

在下面的代码中，我们定义了一个`exchanger`，它就类似于峡谷交易中心，而它的类型`Exchanger<WildMonster>`则明确表示交换的对象是**野怪**。

接着，我们再定义两个线程，分别代表**铠**和**兰陵王**。在其线程的内部，会通过前面定义的`exchanger`对象来和对方进行交换数据。**交换完成后，他们彼此将获得对方的物品**。

```
 public static void main(String[] args) {
     Exchanger<WildMonster> exchanger = new Exchanger<> (); // 定
义交换地点和交换类型

     Thread 铠 = newThread("铠", () -> {
         try {
             WildMonster wildMonster = new Bear("棕熊");
             say("我手里有一只：" + wildMonster.getName());
             WildMonster exchanged =
exchanger.exchange(wildMonster); // 交换后将获得对方的物品
             say("交易完成，我获得了：", wildMonster.getName(), "->",
exchanged.getName());
```

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

        Thread 兰陵王 = newThread("兰陵王", () -> {
            try {
                WildMonster wildMonster = new Wolf("野狼");
                say("我手里有一只：" + wildMonster.getName());
                WildMonster exchanged =
exchanger.exchange(wildMonster);
                say("交易完成，我获得了：", wildMonster.getName(), "->",
exchanged.getName());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        铠.start();
        兰陵王.start();
    }
```

下面是上面代码用到的内部类：

```
    @Data
    private static class WildMonster {
        protected String name;
    }

    private static class Wolf extends WildMonster {
        public Wolf(String name) {
            this.name = name;
        }
    }

    private static class Bear extends WildMonster {
        public Bear(String name) {
            this.name = name;
        }
    }
```

示例代码运行结果如下：

```
铠:我手里有一只：棕熊
兰陵王:我手里有一只：野狼
兰陵王:交易完成，我获得了：野狼->棕熊
铠:交易完成，我获得了：棕熊->野狼

Process finished with exit code 0
```

从结果中可以看到，**铠用棕熊换到了野狼，而兰陵王则用野狼换到了棕熊**，他们完成了交换。

以上就是Exchanger的用法，看起来还是非常简单的，事实上也确实很简单。在使用Exchanger的时候要注意下面几点：

- **定义Exchanger对象，各线程通过这个对象完成交换**；
- **在Exchanger对象中要定义类型，也就是这两个线程要交换什么**；
- **线程在调用Exchanger进行交换时，要特别注意的是，先到的那个线程会原地等待另外一个线程的出现**。比如，铠先到交换地点，可这时候兰陵王还没有到，那么铠会等待兰陵王的出现，除非超过设置的时间限制，比如兰陵王中途被妲己蹲了草丛。反之亦然，兰陵王先到也到等铠的出现。

二、Exchanger的源码与实现

虽然理解Exchanger的思想很容易，了解其用法也很简单，但是若要理清它几百余行的源码却并非易事。其原因在于，**槽是Exchanger中的核心概念和属性**，Exchanger中的数据交换分为**单槽交换**和**多槽交换**，其中单槽交换源码简单，**但多槽交换却很复杂**。所以，下文对Exchanger源码的阐述以概括为主，不会对源码深究。如果你有兴趣，可以参考阅读这篇文章，作者对其源码的解读较为详细。

1. 核心构造

与其他同步工具不同的是，**Exchanger有且仅有一个构造函数**。在这个构造中，也只初始化了一个对象`participant`.

```
public Exchanger() {
    participant = new Participant();
```

```
}
```

从继承关系看，Participant本质上是一个ThreadLocal，而其中的Node则是线程的本地变量。

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

2. 核心属性

Exchanger有四个核心变量，如下所示。当然，除此之外，还有一些用以计算的其他变量。不过，为避免引入不必要的复杂度，本文暂不提及。

```
//ThreadLocal变量，每个线程都有自己的一个副本
private final Participant participant;

//多槽位，高并发下使用，保存待匹配的Node实例
private volatile Node[] arena;

//单槽位，arena未初始化时使用的保存待匹配的Node实例
private volatile Node slot;

//初始值为0，当创建arena后会被赋值成SEQ，用来记录arena数组的可用最大索引，会随着并发的增大而增大直到等于最大值FULL，会随着并行的线程逐一匹配成功而减少恢复成初始值
private volatile int bound;
```

Node的具体细节，注意其中的item和match.

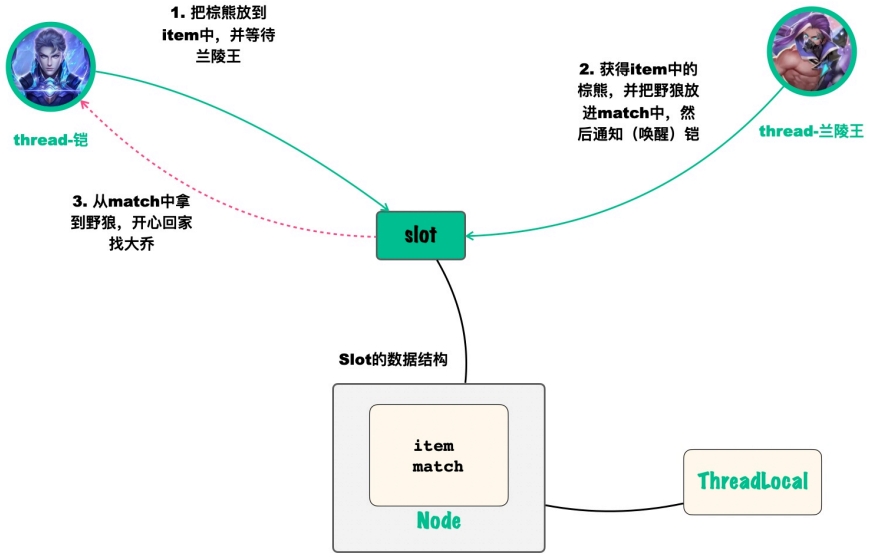
```
 @sun.misc.Contended static final class Node {
        int index;              //arena的下标，多个槽位的时候使用
        int bound;              // 上一次记录的Exchanger.bound
        int collides;           // 记录的 CAS 失败数
        int hash;               // 用于自旋
        Object item;            //   这个线程的数据项
```

```
        volatile Object match;  // 交换的数据
        volatile Thread parked; //  当阻塞时，设置此线程，不阻塞的话会自旋
    }
```

3. 核心方法

```
// 交换数据
// 如果一个线程达到后，会等待其他线程的到达（除非自己被中断）。然后，该线程会和到达的线程交换数据。
// 如果线程在到达后，已经有其他线程在等待。那么，将会唤起该线程并交换数据。
public V exchange(V x) throws InterruptedException {...}

//带有超时限制的交换
public V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException {...}
```

所以，从源码上看上文的示例，那么铠和兰陵王交换数据的过程应该是下面这样的：



小结

以上就是关于Exchanger的全部内容。在学习Exchanger时，要侧重理解它所要解

决的问题场景，以及它的基本用法。对于其源码，当前阶段可以选择“**不求甚解**”，以降维的方式降低学习难度，日后再循序渐进理解。我在写本文时，也曾多次考虑是否要讲清楚源码，最终还是决定暂缓，毕竟现阶段理解它、学会它才是重点。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 使用Exchanger实现生产者与消费者。

延伸阅读与参考资料

- https://www.cnblogs.com/54chensongxia/p/12877843.html
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

铂金10：能工巧匠-ThreadLocal如何为线程打造私有数据空间

欢迎来到《王者并发课》，本文是该系列文章中的**第23篇**，铂金中的**第10篇**。

说起**ThreadLocal**，相信你对它的名字一定不陌生。在并发编程中，它有着较高的出场率，并且也是面试中的高频面试题之一，所以**其重要性不言而喻**。当然，它也可能曾经让你在夜里辗转反侧，或让你在面试时闪烁其词。**因为，ThreadLocal虽然使用简单，但要理解它的原理又似乎并不容易**。

然而，正所谓**明知山有虎，偏向虎山行**。在本文中，我将和你一起学习ThreadLocal的用法及其原理，啃下这块硬骨头。

关于ThreadLocal的用法和原理，网上也有着非常多的资料可以查阅。遗憾的是，这其中的大部分资料在讲解时都不够透彻。有的是蜻蜓点水，没有把必要的细节讲清楚，有的则比较片面，只讲了其中的某个点。

所以，当王者并发课系列写到这篇文章的时候，**如何才能简明扼要地把ThreadLocal介绍清楚，让读者能在一篇文章中透彻地理解它，但同时又要避免万字长文读不下去，是我最近一直在思考的问题**。为此，在综合现有资料的基础上，我精心设计了一些配图，尽可能地让文章图文并茂，以帮助你相对轻松地理解ThreadLocal中的精要。然而，每个读者的背景不同，理解也就不同。所以，对于你认为的并没有讲清楚的地方，希望你在评论区留言反馈，我会尽量调整完善，争取让你“**一文读懂**”。

一、ThreadLocal使用场景初体验

夫子的疑惑：在什么场景下需要使用ThreadLocal?

在王者峡谷中，每个英雄都有着自己的领地和庄园。在庄园里，按照功能职责的不同又划分为不同的区域，比如有**圈养野怪的区域**，还有**存放金币**以及**武器**等不同区

域。**当然，这些区域都是英雄私有的，不能混淆错乱。**

所以，铠在打野和获得金币时，可以把他打的野怪放进自己庄园里，那是他的**私有空间**。同样，兰陵王和其他英雄也是如此。这个设计如下图所示：

现在，我们就来编写一段代码模拟上述的场景：

- **铠在打野和获得金币时，放进他的私有空间里**；
- **兰陵王在打野和获得金币时，放进他的私有空间里**；
- **他们的空间都位于王者峡谷中**。

以下是我们编写的一段代码。在代码中，我们定义了一个`wildMonsterLocal`变量，用于存放英雄们打野时获得的野怪；而`coinLocal`则用于存放英雄们所获得的金币。于是，铠将他所打的**棕熊**放进了圈养区，并将获得的**500金币**放进了**金币存放区**；而兰陵王则将他所打的野狼放进了圈养区，并将获得的**100金币**放进了**金币存放区**。

过了一阵子之后，他们分别取走他们存放的野怪和金币。

主要示例如下所示。在阅读下面示例代码时，要着重注意对ThreadLocal的`get`和`set`方法的调用。

```
//私人野怪圈养区
private static final ThreadLocal < WildMonster > wildMonsterLocal
= new ThreadLocal < > ();
//私人金币存放区
private static final ThreadLocal < Coin > coinLocal = new
ThreadLocal < > ();
```

```
public static void main(String[] args) {
  Thread 铠 = newThread("铠", () -> {
    try {
      say("今天打到了一只棕熊，先把它放进圈养区，改天再享用！");
      wildMonsterLocal.set(new Bear("棕熊"));
      say("路上杀了一些兵线，获得了一些金币，也先存起来！");
      coinLocal.set(new Coin(500));

      Thread.sleep(2000);
      note("\n过了一阵子...\n");
      say("从圈养区拿到了一只：", wildMonsterLocal.get().getName());
      say("金币存放区现有金额：", coinLocal.get().getAmount());
    } catch (InterruptedException e) {}
  });

  Thread 兰陵王 = newThread("兰陵王", () -> {
    try {
      Thread.sleep(1000);
      say("今天打到了一只野狼，先把它放进圈养区，改天再享用！");
      wildMonsterLocal.set(new Wolf("野狼"));
      say("路上杀了一些兵线，获得了一些金币，也先存起来！");
      coinLocal.set(new Coin(100));

      Thread.sleep(2000);
      say("从圈养区拿到了一只：", wildMonsterLocal.get().getName());
      say("金币存放区现有金额：", coinLocal.get().getAmount());
    } catch (InterruptedException e) {}
  });
  铠.start();
  兰陵王.start();
}
```

示例代码中用到的类如下所示：

```
@Data
private static class WildMonster {
  protected String name;
}

private static class Wolf extends WildMonster {
  public Wolf(String name) {
    this.name = name;
```

```
    }
}

private static class Bear extends WildMonster {
    public Bear(String name) {
        this.name = name;
    }
}

@Data
private static class Coin {
    private int amount;

    public Coin(int amount) {
        this.amount = amount;
    }
}
```

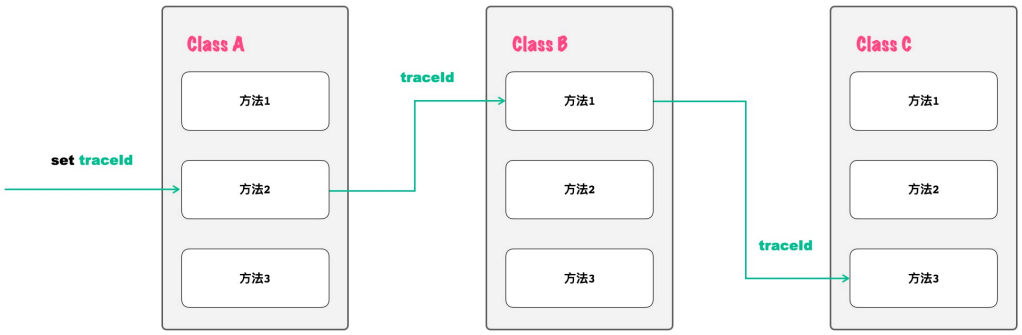
示例代码运行结果如下：

```
铠:今天打到了一只棕熊，先把它放进圈养区，改天再享用!
铠:路上杀了一些兵线，获得了一些金币，也先存起来!
兰陵王:今天打到了一只野狼，先把它放进圈养区，改天再享用!
兰陵王:路上杀了一些兵线，获得了一些金币，也先存起来!

过了一阵子...

铠:从圈养区拿到了一只: 棕熊
铠:金币存放区现有金额: 500
兰陵王:从圈养区拿到了一只: 野狼
兰陵王:金币存放区现有金额: 100

Process finished with exit code 0
```

从运行的结果中，可以清楚地看到，在过了一阵子之后，**铠和兰陵王分别取到了他们之前存放的野怪和金币，并且丝毫不差**。

以上，就是ThreadLocal应用的典型。在多线程并发场景中，如果你需要为**每个线程**设置可以**跨越类和方法**层面的私有变量，那么你就需要考虑使用ThreadLocal了。注意，这里有两个要点，**一是变量为某个线程独享，二是变量可以在不同方法甚至不同的类中共享**。

ThreadLocal在软件设计中的应用场景非常多。举个简单的例子，在一次请求中，如果你需要设置一个**traceId**来跟踪请求的完整调用链路，那么你就需要一个能跨越类和方法的变量，这个变量可以让线程在不同的类中自由获取，且不会出错，其过程如下图所示：



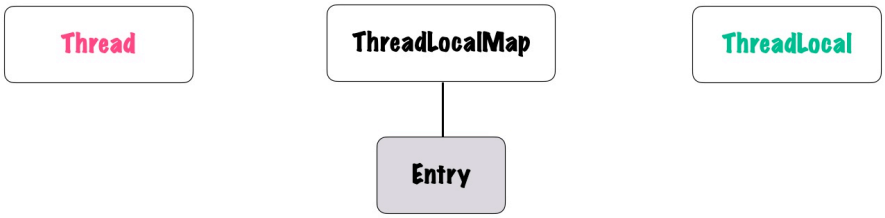
二、ThreadLocal原理解析

对于ThreadLocal，一般来说被提及最多的可能就是那个经典的面试问题：**谈谈你对ThreadLocal内存泄露的理解**。这个问题看起来很简单，但要回答到点子上的话，就必须对其源码有足够理解。当然，背诵面试题的答案扯一通“**软引用**”、“**内存回收**”巴拉巴拉也是可以的，毕竟大部分的面试官也是半吊子。

接下来，我们会结合上文的场景，以及它的示例代码来讲解ThreadLocal的原理，让你找到关于这个问题的真正答案。

1. 源码分析

如果对ThreadLocal理解有困难的话，很大的可能是：**你没有理清不同概念之间的关系**。所以，**理解ThreadLocal源码的第一步是找出它的相关概念，并理清它们之间的关系，即Thread、ThreadLocalMap和ThreadLocal**。正是这三个关键概念，唱出了一台好戏。当然，如果细分的话，你也可以把**Entry**单独拎出来。



关键概念1：Thread类

为什么Thread在关键概念中排名第一，因为ThreadLocal就是为它而生的。那Thread和ThreadLocal是什么关系呢？我们这就来看看Thread的源码：

```
class Thread implements Runnable {
    ...
    /* ThreadLocal values pertaining to this thread. This map is
maintained
     * by the ThreadLocal class. */
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ...
}
```

没有什么比源码展现更清晰的了。你可以非常直观地看到，Thread中有一个变量：`threadLocals`. **通俗地说，这个变量就是用来存放当前线程的一些私有数据的，并且可以存放多个私有数据**，毕竟线程是可以携带多个私有数据的，比如它可以携带`traceId`，也自然可以携带`userId`等数据。理解了这个变量的用途之后，再看看它的类型，也就是`ThreadLocal.ThreadLocalMap`.你看，Thread就这样和ThreadLocal扯上了关系，所以接下来我们来看另外一个关键概念。

关键概念2：ThreadLocalMap类

从Thread的源码中你已经看到，Thread是用ThreadLocalMap来存放线程私有数据的。这里，我们先暂且撇开ThreadLocal，来直接看ThreadLocalMap的源码：

```
static class ThreadLocalMap {

        ...
```

```
        /**
         * The entries in this hash map extend WeakReference,
using
         * its main ref field as the key (which is always a
         * ThreadLocal object).  Note that null keys (i.e.
entry.get()
         * == null) mean that the key is no longer referenced, so
the
         * entry can be expunged from table.  Such entries are
referred to
         * as "stale entries" in the code that follows.
         */
        static class Entry extends WeakReference<ThreadLocal<?>>
{
            /** The value associated with this ThreadLocal. */
            Object value;

            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }

        /**
         * The table, resized as necessary.
         * table.length MUST always be a power of two.
         */
        private Entry[] table;
        ...

}
```

ThreadLocalMap中最关键的属性就是`Entry[] table`，正是它实现了线程私有多数据的存储。而Entry则是继承了WeakReference，并且Entry的Key类型是ThreadLocal. 看到这里，先不要想着ThreadLocalMap的其他源码，你现在应当理解的是，**`table`是线程私有数据存储的地方，而ThreadLocalMap的其他源码不过都是为了`table`数据的存与取而存在的**。这是你对ThreadLocalMap理解的关键，不要把自己迷失在错综复杂的其他源码中。

关键概念3：ThreadLocal类

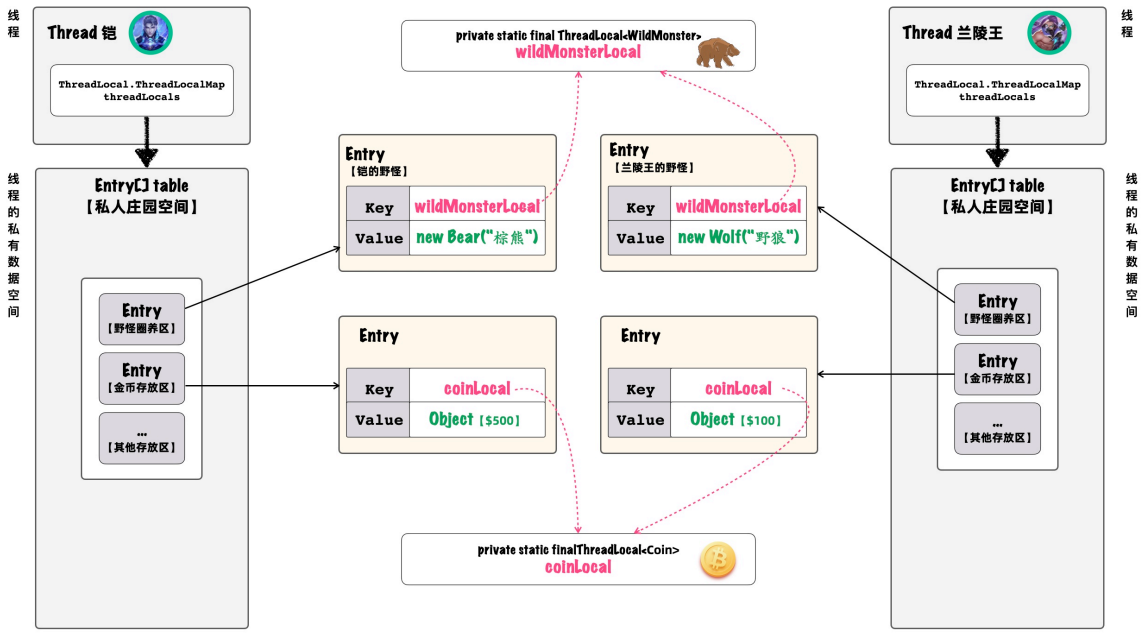
现在，目光终于到了ThreadLocal这个类上。Thread中使用到了ThreadLocalMap，

而接下来你会发现ThreadLocal不过是封装了一些对ThreadLocalMap的操作。你看，ThreadLocal中的`get()`、`set()`、`remove()`等方法都是在操作ThreadLocalMap. 在各种操作之前，都会通过`getMap()`方法拿到当前线程的ThreadLocalMap.

```
public class ThreadLocal<T> {

    ...

    // 获取当前线程的数据
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
    // 初始化数据
    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }

    // 设置当前线程的数据
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
```

```
    }

     public void remove() {
         ThreadLocalMap m = getMap(Thread.currentThread());
         if (m != null)
             m.remove(this);
     }

    // 获取线程私有数据存储的关键，虽然操作在ThreadLocal中，但是实际操作的
是Thread中的threadLocals变量
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }

    // 初始化线程的t.threadLocals变量，设置为空值
    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }
    ...
}
```

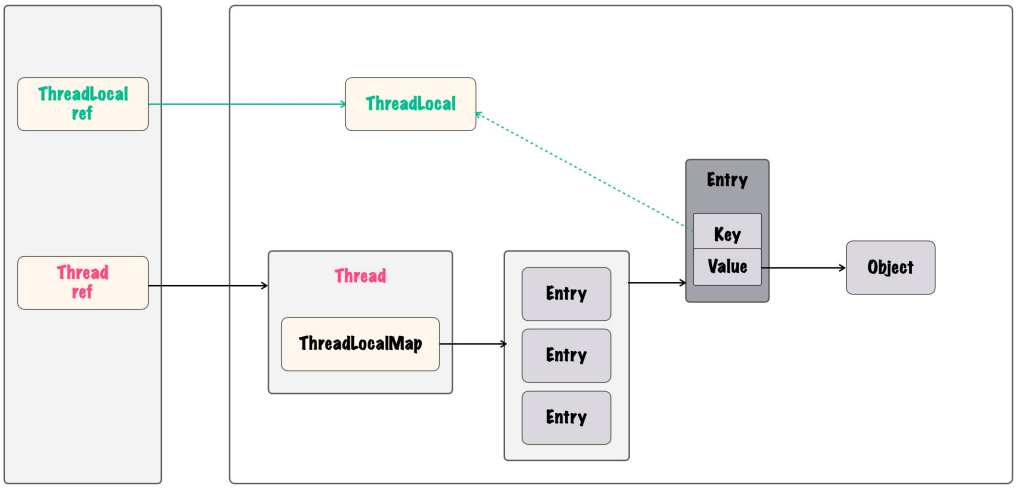
如果，此时你对相关概念及其源码的理解仍然感到困惑，**那就对了**。下面这幅图，将结合相关概念和示例代码，来还原这其中的相关概念和它们之间的关系，这幅图值得你反复细品。



在上面这幅图中，你需要如下这些细节：

- 有两个线程：**铠**和**兰陵王**；
- 有两个ThreadLocal对象，它们分别用于存放线程的私有数据，即英雄们的野怪和金币；
- **线程铠**和**线程兰陵王**都有一个ThreadLocal.ThreadLocalMap的变量，用来存放不同的ThreadLocal，即`wildMonsterLocal`和`coinLocal`这两个变量都会放进ThreadLocalMap的`table`里，也就是entry数组中；
- 当线程向ThreadLocalMap中放入数据时，它的key会指向ThreadLocal对象，而value则是ThreadLocal中的值。比如，当**铠**将**棕熊**放入`wildMonsterLocal`中时，对应Entry的key是`wildMonsterLocal`，而value则是`new Bear()`，即**棕熊**。**当兰陵王放入野怪时，同理；当铠放入金币时，也是同理**；
- 当Entry的key指向ThreadLocal对象时，比如指向`wildMonsterLocal`或`coinLocal`时，注意，**是虚引用，是虚引用，是虚引用，是虚引用**！重要的事情，**说四遍**。看图中的红线虚线，或ThreadLocalMap源码中的`WeakReference`.

如果你已经看明白上面这幅图，那么下面这幅图中的关系也就应该一目了既然。否则，如果你似乎看不明白它，请回到上面继续品上面那幅图，直到你对下图一目了然。



2. 使用指南

接下来，将为你简单介绍ThreadLocal的一些常见高频用法。

（1）创建ThreadLocal

像创建其他对象一样创建即可，没有什么特别之处。

```
ThreadLocal < WildMonster > wildMonsterLocal = new ThreadLocal <
> ();
```

在对象创建完成之后，每个线程便可以向其中读写数据。当然，每个线程都只能看到它们自己的数据。

（2）设置ThreadLocal的值

```
wildMonsterLocal.set(new Bear("棕熊"));
```

（3）取出ThreadLocal的值

```
wildMonsterLocal.get();
```

在读取数据时需要注意的是，如果此时还没有数据设置进来，那么将会调用`setInitialValue`方法来设置初始值并返回给调用方。

（4）移除ThreadLocal的值

```
wildMonsterLocal.remove();
```

（5）初始化ThreadLocal的值

```
private ThreadLocal wildMonsterLocal = new
ThreadLocal<WildMonster>() {
    @Override
    protected WildMonster initialValue() {
        return new WildMonster();
    }
};
```

在对ThreadLocal进行`get`操作时，如果当前尚未进行过数据设置，那么会执行初

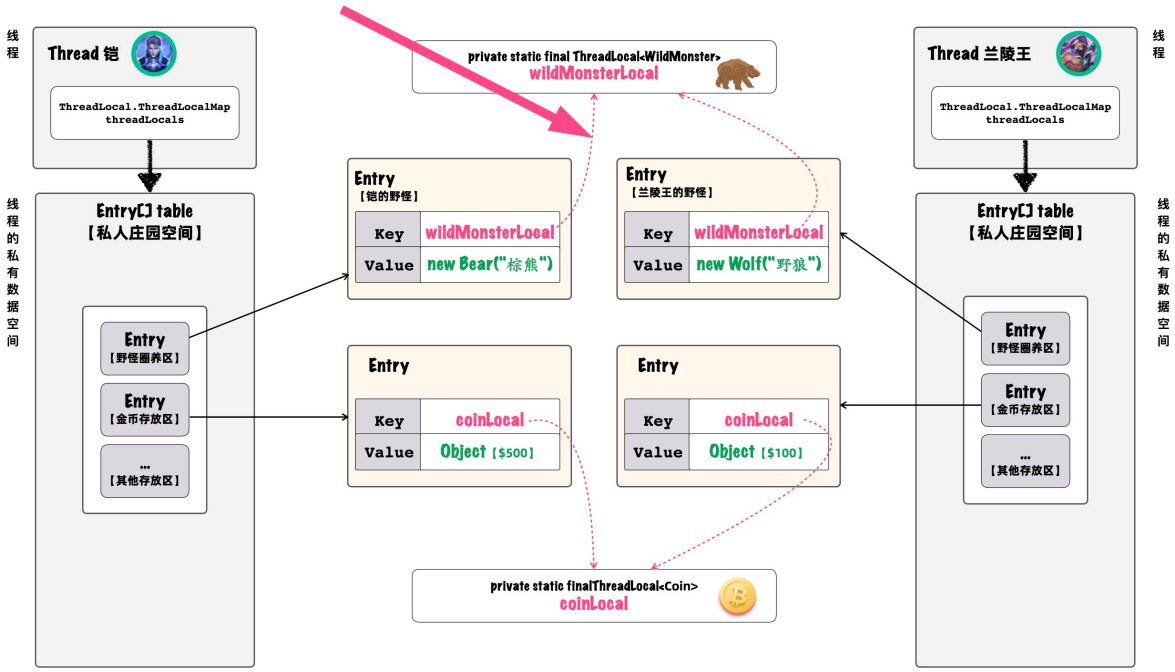
始化动作，如果你此时希望设置初始值，可以重写它的`initialValue`方法。

3. 如何理解ThreadLocal的内存泄露问题

首先，你要理解**弱引用**这个概念。在Java中，引用分为强引用、弱引用、软引用、虚幻引用等不同的引用类型，**而不同的引用类型对应的则是不同的垃圾回收策略**。如果你对此不熟的话，建议可以去检索相关资料，也可以看这篇。

对于弱引用，在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，**不管当前内存空间足够与否，都会回收它的内存**。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。但是，即便是偶尔发生，也足够造成问题。

当你理解了弱引用和对应的垃圾回收策略之后，此刻，请回到上面的那幅图：



在这幅图里，Entry的key指向ThreadLocal对象时，用的正是弱引用，图中已经红色箭头标注。这里的红色虚线会造成上面问题呢？你想想看，如果此时ThreadLocal对象被回收时，那么Entry中的key就编程了null. **可是，虽然key（wildMonsterLocal）变成了null，value的值（new Bear("棕熊")）还是强引用，它还会继续存在，但实际已经没有用了，所以会造成这个Entry就废了，但是因为value的存在却不能被回收**。于是，内存泄露就这样产生了。

那既然如此，为什么要使用弱引用?

相信你一定有这个疑问，如果没有，这篇文章你可能需要再读一遍。明知这里会产生内存泄露的风险，却仍然使用弱引用的原因在于：**当ThreadLocal对象没有强引用时，它们需要被清理，否则它们长期存在于ThreadLocalMap中，也是一种内存泄露**。你看，问题就是这样的一环扣着一环。

最佳实践：如何避免内存泄露

那么，既然事已如此，如何避免内存泄露呢？这里给出一个可行的最佳实践：**在调用完成后，手动执行remove()方法**。

```
private static final ThreadLocal<WildMonster> wildMonsterLocal =
new ThreadLocal<>();

try{
    wildMonsterLocal.get();
    ...
}finally{
    wildMonsterLocal.remove();
}
```

除此之外，ThreadLocal也给出一个方案：在调用`set`方法设置时，会调用`replaceStaleEntry`方法来检查key为null的Entry。如果发现有key为null的Entry，那么会将它的value也设置为null，这样Entry便可以被回收。**当然，如果你没有再调用`set`方法，那么这个方案就是无效的**。

```
private void set(ThreadLocal < ? > key, Object value) {

     ...

     for (Entry e = tab[i]; e != null; e = tab[i = nextIndex(i,
len)]) {
         ThreadLocal < ? > k = e.get();

         if (k == key) {
             e.value = value;
             return;
         }
```

```
            if (k == null) {
                replaceStaleEntry(key, value, i); //看这里
                return;
            }
        }

        ...
    }
```

小结

以上就是关于ThreadLocal的全部内容。在学习ThreadLocal时，首先要理解的是它的应用场景，即它所要解决的问题。其次，对它的源码要有一定的了解。在了解源码时，要注意从Thread、ThreadLocal和ThreadLocalMap三个概念出发，理解他们之间的关系。如此，你才能完全理解常见的内存泄露问题是怎么一回事。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 尝试向你的朋友解释ThreadLocal内存泄露是如何发生的。

延伸阅读与参考资料

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

钻石01：明心见性-如何由表及里深精通线程池设计与原理

欢迎来到《王者并发课》，本文是该系列文章中的**第24篇**，砖石中的**第1篇**。

在钻石系列中，我们将学习线程池相关的框架和工具类。作为铂金系列的第一篇，我们将在这篇文章中深入讲解线程池的应用及原理。

关于线程池，无论是在实际的项目开发还是面试，它都是并发编程中当之无愧的重中之重。因此，掌握线程池是每个Java开发者的必备技能。

本文将从线程池的应用场景和设计原理出发，先带大家手撸一个线程池，在理解线程池的内部构造后，再深入剖析Java中的线程池。全文大约**2.5万字**，篇幅较长，**在阅读时建议先看目录再看内容**。

一、为什么要使用线程池

在前面系列文章的学习中，你已然知道多线程可以加速任务的处理、提高系统的吞吐量。那么，是否我们因此就可以频繁地创建新的线程呢？答案是否定的。频繁地繁创建和启用新的线程不仅代价昂贵，而且无限增加的线程势必也会造成管理成本的急剧上升。**因此，为了平衡多线程的收益和成本，线程池诞生了**。

1. 线程池的使用场景

生产者与消费者问题是线程池的典型应用场景。当你有源源不断的任务需要处理时，为了提高任务的处理速度，你需要创建多个线程。那么，问题来了，如何管理这些任务和多线程呢？答案是：**线程池**。

线程池的池化（Pooling）原理的应用并不局限于Java中，在MySQL和诸多的分布式中间件系统中都有着广泛的应用。当我们链接数据库的时候，对链接的管理用的是线程池；当我们使用Tomcat时，对请求链接的管理用的也是线程池。所以，**当你**

有批量的任务需要多线程处理时，那么基本上你就需要使用线程池。

2. 线程池的使用好处

线程池的好处主要体现在三个方面：**系统资源**、**任务处理速度**和**相关的复杂度管理**，主要表现在：

- **降低系统的资源开销**：通过复用线程池中的工作线程，避免频繁创建新的线程，可以有效降低系统资源的开销；
- **提高任务的执行速度**：新任务达到时，无需创建新的线程，直接将任务交由已经存在的线程进行处理，可以有效提高任务的执行速度；
- **有效管理任务和工作线程**：线程池内提供了任务管理和工作线程管理的机制。

为什么说创建线程是昂贵的

现在你已经知道，频繁地创建新线程需要付出额外的代价，所以我们使用了线程池。那么，创建一个新的线程的代价究竟是怎样的呢？可以参考以下几点：

- 创建线程时，JVM必须为线程堆栈分配和初始化一大块内存。每个线程方法的调用栈帧都会存储到这里，包括局部变量、返回值和常量池等；
- 在创建和注册本机线程时，需要和宿主机发生系统调用；
- 需要创建、初始化描述符，并将其添加到 JVM 内部数据结构中。

另外，从某种意义上说，**只要线程还活着，它就会占用资源，这不仅昂贵，而且浪费**。 例如 ，线程堆栈、访问堆栈的可达对象、JVM 线程描述符、操作系统本机线程描述符等等，在线程活着的时候，这些资源都会持续占据。

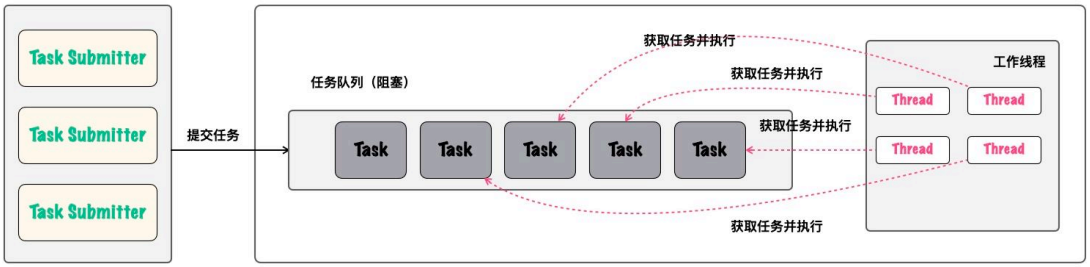
虽然不同的Java平台在创建线程时的代价可能有所差异，但总体来说，都不便宜。

3. 线程池的核心组成

一个完整的线程池，应该包含以下几个核心部分：

- **任务提交**：提供接口接收任务的提交；
- **任务管理**：选择合适的队列对提交的任务进行管理，包括对拒绝策略的设置；
- **任务执行**：由工作线程来执行提交的任务；

- **线程池管理**：包括基本参数设置、任务监控、工作线程管理等。



二、如何手工制作线程池

通过第一部分的阅读，现在你已经了解了线程池的作用及它的核心组成。为了更深刻地理解线程池的组成，在这一部分我们通过简单的**四步**来手工制作一个简单的线程池。当然，**麻雀虽小，五脏俱全**。如果你能手工自制线程池之后，那么在理解后续的Java中的线程池时，将会易如反掌。

1. 线程池设计和制作

第一步：定义一个王者线程池：TheKingThreadPool，它是这次手工制作中名副其实的**主角儿**。在这个线程池中，包含了任务队列管理、工作线程管理，并提供了可以指定队列类型的构造参数，以及任务提交入口和线程池关闭接口。你看，虽然它看起来似乎很迷你，但是线程池的核心组件都已经具备了，甚至在它的基础上，你完全可以把它扩展成更为成熟的线程池。

```
/**
 * 王者线程池
 */
public class TheKingThreadPool {
    private final BlockingQueue<Task> taskQueue;
    private final List<Worker> workers = new ArrayList<>();
    private ThreadPoolStatus status;

    /**
     * 初始化构建线程池
```

```
     *
     * @param worksNumber 线程池中的工作线程数量
     * @param taskQueue   任务队列
     */
    public TheKingThreadPool(int worksNumber, BlockingQueue<Task>
taskQueue) {
        this.taskQueue = taskQueue;
        status = ThreadPoolStatus.RUNNING;
        for (int i = 0; i < worksNumber; i++) {
            workers.add(new Worker("Worker" + i, taskQueue));
        }
        for (Worker worker : workers) {
            Thread workThread = new Thread(worker);
            workThread.setName(worker.getName());
            workThread.start();
        }
    }

    /**
     * 提交任务
     *
     * @param task 待执行的任务
     */
    public synchronized void execute(Task task) {
        if (!this.status.isRunning()) {
            throw new IllegalStateException("线程池非运行状态，停止接
单啦~");
        }
        this.taskQueue.offer(task);
    }

    /**
     * 等待所有任务执行结束
     */
    public synchronized void waitUntilAllTasksFinished() {
        while (this.taskQueue.size() > 0) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    /**
```

```
     * 关闭线程池
     */
    public synchronized void shutdown() {
        this.status = ThreadPoolStatus.SHUTDOWN;
    }

    /**
     * 停止线程池
     */
    public synchronized void stop() {
        this.status = ThreadPoolStatus.SHUTDOWN;
        for (Worker worker : workers) {
            worker.doStop();
        }
    }
}
```

第二步：设计并制作工作线程。工作线程是干活的线程，将负责处理提交到线程池中的任务，我们把它叫做**Worker**。其实，这里的Worker的定义和Java线程池中的Worker已经很像了，它继承了**Runnable**接口并封装了Thread. 在构造Worker时，可以设定它的名字，并传入任务队列。当Worker启动后，它将会从任务队列中获取任务并执行。此外，它还提供了`Stop`方法，用以响应线程池的状态变化。

```
/**
 * 线程池中用于执行任务的线程
 */
public class Worker implements Runnable {
    private final String name;
    private Thread thread = null;
    private final BlockingQueue<Task> taskQueue;
    private boolean isStopped = false;
    private AtomicInteger counter = new AtomicInteger();

    public Worker(String name, BlockingQueue<Task> queue) {
        this.name = name;
        taskQueue = queue;
    }

    public void run() {
        this.thread = Thread.currentThread();
        while (!isStopped()) {
            try {
```

```
                Task task = taskQueue.poll(5L, TimeUnit.SECONDS);
                if (task != null) {
                    note(this.thread.getName(), ":获取到新的任务->",
task.getTaskDesc());
                    task.run();
                    counter.getAndIncrement();
                }
            } catch (Exception ignored) {
            }
        }
        note(this.thread.getName(), ":已结束工作，执行任务数量: " +
counter.get());
    }

    public synchronized void doStop() {
        isStopped = true;
        if (thread != null) {
            this.thread.interrupt();
        }
    }

    public synchronized boolean isStopped() {
        return isStopped;
    }

    public String getName() {
        return name;
    }
}
```

第三步：设计并制作任务。任务是可以**可执行**的对象，因此我们直接继承Runnable接口就行。其实，直接使用Runnable接口也是可以的，只不过为了让示例更加清楚，我们给**Task**加了任务描述的方法。

```
/**
 * 任务
 */
public interface Task extends Runnable {
    String getTaskDesc();
}
```

第四步：设计线程池的状态。线程池作为一个运行框架，它必然会有一系列的状

态，比如运行中、停止、关闭等。

```
public enum ThreadPoolStatus {
    RUNNING(),
    SHUTDOWN(),
    STOP(),
    TIDYING(),
    TERMINATED();

    ThreadPoolStatus() {
    }

    public boolean isRunning() {
        return ThreadPoolStatus.RUNNING.equals(this);
    }
}
```

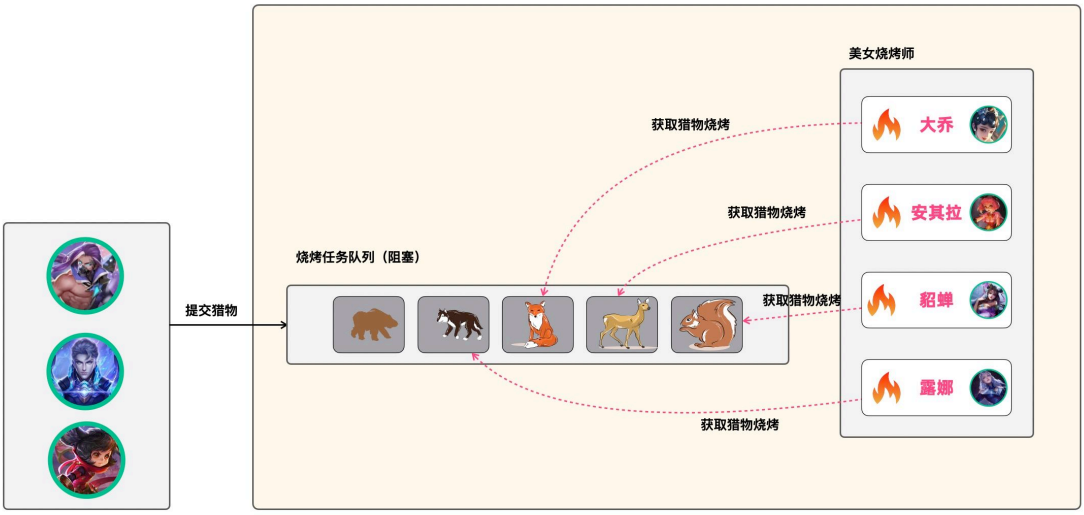
以上四个步骤完成后，一个简易的线程池就已经制作完毕。你看，如果你从以上几点入手来理解线程池的源码的话，是不是要简单多了？Java中的线程池的核心组成也是如此，只不过在细节处理等方面更多全面且丰富。

2. 运行线程池

现在，我们的王者线程池已经制作好。接下来，我们通过一个场景来运行它，看看它的效果如何。

试验场景：**峡谷森林中，铠、兰陵王和典韦等负责打野，而安其拉、貂蝉和大乔等美女负责对狩猎到的野怪进行烧烤，一场欢快的峡谷烧烤节正在进行中**。

在这个场景中，铠和兰陵王他们负责提交任务，而貂蝉和大乔她们则负责处理任务。



在下面的实现代码中，我们通过上述设计的TheKingThreadPool来定义个线程池，`wildMonsters`中的野怪表示待提交的任务，并安排3个工作线程来执行任务。在示例代码的末尾，当所有任务执行结束后，关闭线程池。

```
 public static void main(String[] args) {
        TheKingThreadPool theKingThreadPool = new
TheKingThreadPool(3, new ArrayBlockingQueue<>(10));

        String[] wildMonsters = {"棕熊", "野鸡", "灰狼", "野兔", "狐
狸", "小鹿", "小花豹", "野猪"};
        for (String wildMonsterName : wildMonsters) {
            theKingThreadPool.execute(new Task() {
                public String getTaskDesc() {
                    return wildMonsterName;
                }

                public void run() {

System.out.println(Thread.currentThread().getName() + ":" +
wildMonsterName + "已经烤好");
                }
            });
        }

        theKingThreadPool.waitUntilAllTasksFinished();
        theKingThreadPool.stop();
    }
```

王者线程池运行结果如下：

```
Worker0:获取到新的任务->灰狼
Worker1:获取到新的任务->野鸡
Worker1:野鸡已经烤好
Worker2:获取到新的任务->棕熊
Worker2:棕熊已经烤好
Worker1:获取到新的任务->野兔
Worker1:野兔已经烤好
Worker0:灰狼已经烤好
Worker1:获取到新的任务->小鹿
Worker1:小鹿已经烤好
Worker2:获取到新的任务->狐狸
Worker2:狐狸已经烤好
Worker1:获取到新的任务->野猪
Worker1:野猪已经烤好
Worker0:获取到新的任务->小花豹
Worker0:小花豹已经烤好
Worker0:已结束工作，执行任务数量: 2
Worker2:已结束工作，执行任务数量: 2
Worker1:已结束工作，执行任务数量: 4

Process finished with exit code 0
```

从结果中可以看到，效果完全符合预期。**所有的任务都已经提交完毕，并且都被正确执行**。此外，通过线程池的任务统计，可以看到任务并不是均匀分配，Worker1执行了4个任务，而Worker0和Worker2均只执行了2个任务，这也是线程池中的正常现象。

三、透彻理解Java中的线程池

在手工制作线程线程池之后，再来理解Java中的线程池就相对要容易很多。当然，相比于王者线程池，Java中的线程池（ThreadPoolExecutor）的实现要复杂很多。所以，**理解时应当遵循一定的结构和脉络，把握住线程池的核心要点**，眉毛胡子一把抓、理不清层次会导致你无法有效理解它的设计内涵，进而导致你无法正确掌握它。

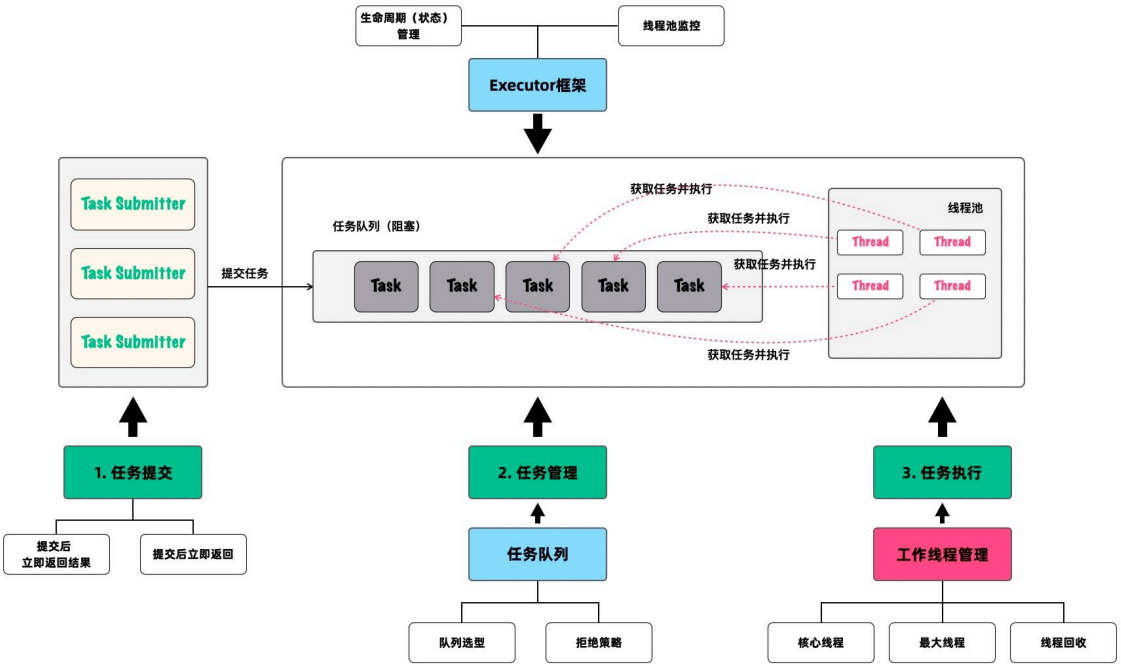
总体来说，Java中的线程池的设计核心都是围绕“**任务**”进行，可以通过**一个框架**、

两大核心、**三大过程**概括。理解了这三个重要概念，基本上你已经能从相对抽象的层面理解了线程池。

- **一个框架**：即线程池的整体设计存在一个框架，而不是杂乱无章的组成。所以，在学习线程池时，首先要能从**立体上感知**到这个框架的存在，而不要陷于凌乱的细节中；
- **两大核心**：在线程池的整个框架中，围绕任务执行这件事，存在两大核心：**任务的管理**和**任务的执行**，对应的也就是**任务队列**和用于执行任务的**工作线程**。**任务队列**和**工作线程**是框架得以有效运转的关键部件；
- **三大过程**：前面说过，线程池的整体设计都是围绕**任务**展开，所以框架内可以分为**任务提交**、**任务管理**和**任务执行**三大过程。

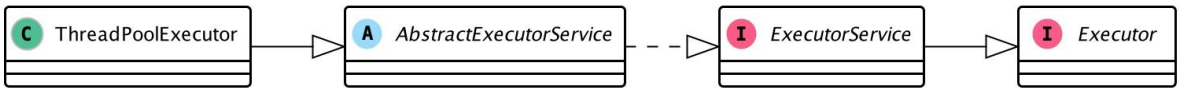
从类比的角度讲，你可以把**框架**看作是一个生产车间。在这个车间里，有一条流水线，**任务队列**和**工作线程**是这条流水线的两大关键组成。而在流水线运作的过程中，就会涉及**任务提交**、**任务管理**和**任务执行**等不同的过程。

下面这幅图，将帮助你立体地感知线程池的整体设计，**建议你收藏**。在这幅图中，清楚地展示了线程池整个框架的工作流程和核心部件，接下来的文章也将围绕这幅图展开。



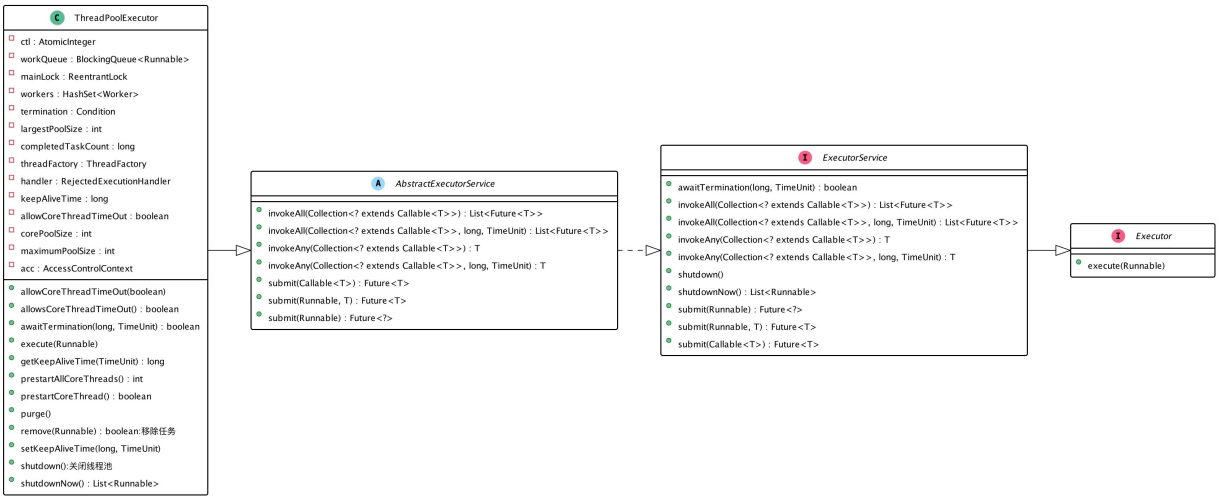
1. 线程池框架设计概览

从源码层面看，理解Java中的线程池，要从下面这**四兄弟**的概念和关系入手，这四个概念务必了然于心。

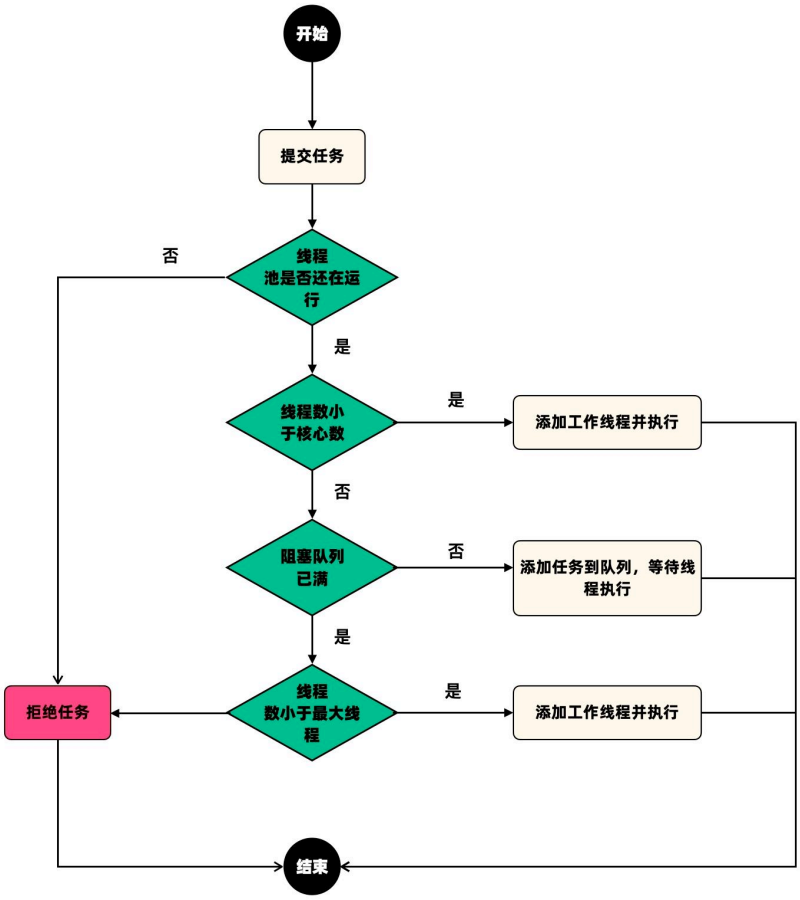


- **Executor**：作为线程池的最顶层接口，Executor的接口在设计上，实现了**任务提交**与**任务执行**之间的解耦，**这是它存在的意义**。在Executor中，只定义了一个方法`void execute(Runnable command)`，用于执行提交的**可运行**的任务。注意，你看它这个方法的参数干脆就叫`command`，也就是“**命令**”，意在表明所提交的不是一个静止的对象，而是**可运行**的命令。并且，这个命令将在未来的某一时刻执行，具体由哪个线程来执行也是不确定的；
- **ExecutorService**：继承了Executor的接口，并在此基础上提供可以**管理服务**和**执行结果（Futrue）** 的能力。ExecutorService所提供的`submit`方法可以返回任务的执行结果，而`shutdown`方法则可以用于关闭服务。相比起来，**Executor只具备单一的执行能力，而ExecutorService则不仅具有执行能力，还提供了简单的服务管理能力**；
- **AbstractExecutorService**：作为ExecutorService的简单实现，该类通过RunnableFuture和newTaskFor实现了`submit`、`invokeAny`和`invokeAll`等方法；
- **ThreadPoolExecutor**：该类是线程池的最终实现类，实现了Executor和ExecutorService中定义的能力，并丰富了AbstractExecutorService中的实现。在ThreadPoolExecutor中，定义了任务管理策略和线程池管理能力，相关能力的实现细节将是我们下文所要讲解的核心所在。

如果你觉得还是不太能直观地感受四兄弟的差异，那么你可以放大查看下面这幅高清图示。看的时候，**要格外注意它们各自方法的不同，方法的不同意味着它们的能力不同**。



而对于线程池总体的执行过程，下面这幅图也建议你收藏。这幅图虽然简明，但完整展示了从任务提交到任务执行的整个过程。这个执行过程往往也是面试中的高频面试题，务必掌握。



（1）线程池的核心属性

线程池中的一些核心属性选取如下，对于其中个别属性会做特别说明。

```
// 线程池控制相关的主要变量
// 这个变量很神奇，下文后专门陈述，请特别留意
private final AtomicInteger ctl = new
AtomicInteger(ctlOf(RUNNING, 0));

// 待处理的任务队列
private final BlockingQueue < Runnable > workQueue;
// 工作线程集合
private final HashSet < Worker > workers = new HashSet < Worker >
();
// 创建线程所用到的线程工厂
private volatile ThreadFactory threadFactory;
// 拒绝策略
private volatile RejectedExecutionHandler handler;
// 核心线程数
private volatile int corePoolSize;
// 最大线程数
private volatile int maximumPoolSize;
// 空闲线程的保活时长
private volatile long keepAliveTime;
// 线程池变更的主要控制锁，在工作线程数、变更线程池状态等场景下都会用到
private final ReentrantLock mainLock = new ReentrantLock();
```

关于ctl字段的特别说明

在ThreadPoolExecutor的多个核心字段中，其他字段可能都比较好理解，但是`ctl`要单独拎出来做些解释。

顾名思义，`ctl`这个字段用于对线程池的控制。它的设计比较有趣，用一个字段却表示了两层含义，也就是这个字段实际是两个字段的合体：

- **runState**：线程池的运行状态（高3位）；
- **workerCount**：工作线程数量（低29位）。

这两个字段的值相互独立，互不影响。那为何要用这种设计呢？这是因为，在线程池中这两个字段几乎总是如影相随，如果不用一个字段来表示的话，那么就需要通

过锁的机制来控制两个字段的一致性。不得不说，这个字段设计上还是比较巧妙的。

在线程池中，也提供了一些方法可以方便地获取线程池的状态和工作线程数量，它们都是通过对`ctl`进行位运算得来。

```
/**
    计算当前线程池的状态
*/
private static int runStateOf(int c) {
    return c & ~CAPACITY;
}
/**
    计算当前工作线程数
*/
private static int workerCountOf(int c) {
    return c & CAPACITY;
}
/**
    初始化ctl变量
*/
private static int ctlOf(int rs, int wc) {
    return rs | wc;
}
```

关于位运算，这里补充一点说明，如果你对位运算有点迷糊的话可以看看，如果你对它比较熟悉则可以直接跳过。

> 假设A=15，二进制是1111；B=6，二进制是110.

运算符	名称	描述	示例
&	按位与	如果相对应位都是1，则结果为1，否则为0	（A&B），得到6，即110
		按位取	

~	按位非	反运算符翻转操作数的每一位，即0变成1，1变成0。	（~A）得到-16，即11111111111111111111111111110000
|	按位或	如果相对应位都是0，则结果为0，否则为 1	（A | B）得到15，即 1111

（2）线程池的核心构造器

ThreadPoolExecutor有四个构造器，其中一个是核心构造器。你可以根据需要，**按需**使用这些构造器。

- **核心构造器之一**：相对较为常用的一个构造器，你可以指定核心线程数、最大线程数、线程保活时间和任务队列类型。

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

- **核心构造器之二**：相比于第一个构造器，你可以在这个构造器中指定ThreadFactory. 通过ThreadFactory，你可以指定线程名称、分组等个性化信

息。

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

- **核心构造器之三**：这个构造器的要点在于，你可以指定拒绝策略。关于任务队列的拒绝策略，下文有详细介绍。

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

- **核心构造器之四**：这个构造器是ThreadPoolExecutor的核心构造器，提供了较为全面的参数设置，上述的三个构造器都是基于它实现。

```
public ThreadPoolExecutor(int corePoolSize,
      int maximumPoolSize,
      long keepAliveTime,
      TimeUnit unit,
      BlockingQueue < Runnable > workQueue,
      ThreadFactory threadFactory,
      RejectedExecutionHandler handler) {
      if (corePoolSize < 0 ||
          maximumPoolSize <= 0 ||
          maximumPoolSize < corePoolSize ||
          keepAliveTime < 0)
```

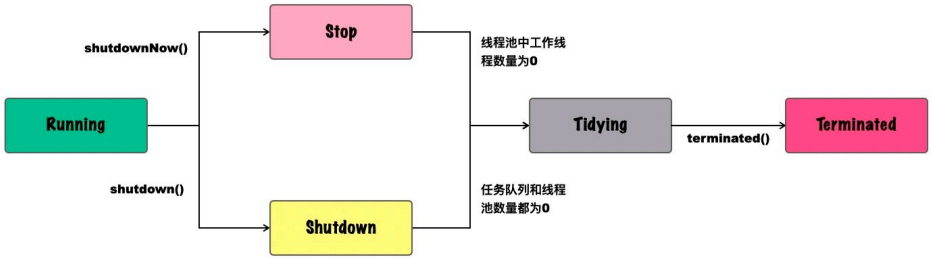
```
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler
== null)
            throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
}
```

（3）线程池中的核心方法

```
/**
* 提交Runnable类型的任务并执行，但不返回结果
*/
public void execute(Runnable command){...}
/**
* 提交Runnable类型的任务，并返回结果
*/
public Future<?> submit(Runnable task){...}
/**
* 提交Runnable类型的任务，并返回结果，支持指定默认结果
*/
public <T> Future<T> submit(Runnable task, T result){...}
/**
* 提交Callable类型的任务并执行
*/
public <T> Future<T> submit(Callable<T> task) {...}
/**
* 关闭线程池，继续执行队列中未完成的任务，但不会接收新的任务
*/
public void shutdown() {...}
/**
* 立即关闭线程池，同时放弃未执行的任务，并不再接收新的任务
*/
public List<Runnable> shutdownNow(){...}
```

（4）线程池的状态与生命周期管理

前文说过，线程池恰似一个生产车间，而从生产车间的角度看，生产车间有运行、停产等不同状态，所以线程池也是有一定的状态和使用周期的。



- **Running**：运行中，该状态下可以继续向线程池中增加任务，并正常处理队列中的任务；
- **Shutdown**：关闭中，该状态下线程池不会立即停止，但不能继续向线程池中增加任务，直到任务执行结束；
- **Stop**：停止，该状态下将不再接收新的任务，**同时不再处理队列中的任务，并中断工作中的线程**；
- **Tidying**：相对短暂的中间状态，所有任务都已经结束，并且所有的工作线程都不再存在（workerCount==0），并运行`terminated()`钩子方法；
- **Terminated**：`terminated()`运行结束。

2. 如何向线程池中提交任务

向线程池提交任务有两种比较常见的方式，**一种是需要返回执行结果的，一种则是不需要返回结果的**。

（1）不关注任务执行结果：execute

通过`execute()`提交任务到线程池后，任务将在未来某个时刻执行，**执行的任务的线程可能是当前线程池中的线程，也可能是新创建的线程**。当然，如果此时线程池应关闭，或者任务队列已满，那么该任务将交由**RejectedExecutionHandler**处

理。

（2）关注任务执行结果：submit

通过`submit()`提交任务到线程池后，运行机制和`execute`类似，其核心不同在于，由`submit()`提交任务时将等待任务执行结束并返回结果。

3. 如何管理提交的任务

（1）任务队列选型策略

- **SynchronousQueue**：无缝传递（Direct handoffs）。当新的任务到达时，将直接交由线程处理，而不是放入缓存队列。因此，如果任务达到时却没有可用线程，那么将会创建新的线程。所以，为了避免任务丢失，在使用SynchronousQueue时，将会需要创建无数的线程，在使用时需要谨慎评估。
- **LinkedBlockingQueue**：无界队列，新提交的任务都会缓存到该队列中。使用无界队列时，只有corePoolSize中的线程来处理队列中的任务，这时候和maximumPoolSize是没有关系的，它不会创建新的线程。当然，你需要注意的是，如果任务的处理速度远低于任务的产生速度，那么LinkedBlockingQueue的无限增长可能会导致内存容量等问题。
- **ArrayBlockingQueue**：有界队列，可能会触发创建新的工作线程，maximumPoolSize参数设置在有界队列中将发挥作用。在使用有界队列时，要特别注意任务队列大小和工作线程数量之间的权衡。如果任务队列大但是线程数量少，那么结果会是系统资源（主要是CPU）占用率较低，但同时系统的吞吐量也会降低。反之，如果缩小任务队列并扩大工作线程数量，那么结果则是系统吞吐量增大，但同时系统资源占用也会增加。所以，使用有界队列时，要考虑到平衡的艺术，并配置相应的拒绝策略。

（2）如何选择合适的拒绝策略

在使用线程池时，拒绝策略是必须要确认的地方，因为它可能会造成任务丢失。

当**线程池已经关闭**或**任务队列已满且无法再创建新的工作线程**时，那么新提交的任务将会被拒绝，拒绝时将调用RejectedExecutionHandler中

的`rejectedExecution(Runnable r, ThreadPoolExecutor executor)`来执行具体的拒绝动作。

```
final void reject(Runnable command) {
    handler.rejectedExecution(command, this);
}
```

以execute方法为例，当线程池状态异常或无法新增工作线程时，将会执行任务拒绝策略。

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
               int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        else if (!addWorker(command, false))
            reject(command);
}
```

ThreadPoolExecutor的默认拒绝策略是AbortPolicy，这一点在属性定义中已经确定。在大部分场景中，直接拒绝任务都是不合适的。

```
private static final RejectedExecutionHandler defaultHandler =
new AbortPolicy();
```

- **AbortPolicy**：默认策略，直接抛出RejectedExecutionException异常；
- **CallerRunsPolicy**：交由当前线程自己来执行。这种策略这提供了一个简单的反馈控制机制，可以减慢提交新任务的速度；
- **DiscardPolicy**：直接丢弃任务，不会抛出异常；

- **DiscardOldestPolicy**：如果此时线程池没有关闭，将从队列的头部取出第一个任务并丢弃，并再次尝试执行。如果执行失败，那么将重复这个过程。

如果上述四种策略均不满足，你也可以通过RejectedExecutionHandler接口定制个性化的拒绝策略。事实上，**为了兼顾任务不丢失和系统负载，建议你自己实现拒绝策略**。

（3）队列维护

对于任务队列的维护，线程池也提供了一些方法。

- **获取当前任务队列**

```
public BlockingQueue<Runnable> getQueue() {
    return workQueue;
}
```

- **从队列中移除任务**

```
public boolean remove(Runnable task) {
    boolean removed = workQueue.remove(task);
    tryTerminate(); // In case SHUTDOWN and now empty
    return removed;
}
```

4. 如何管理执行任务的工作线程

（1）核心工作线程

核心线程（corePoolSize）是指最小数量的工作线程，此类线程不允许超时回收。当然，如果你设置了`allowCoreThreadTimeOut`，那么核心线程也是会超时的，这可能会导致核心线程数为零。核心线程的数量可以通过线程池的构造参数指定。

（2）最大工作线程

最大工作线程指的是线程池为了处理现有任务，所能创建的最大工作线程数量。

最大工作线程可以通过构造函数的`maximumPoolSize`变量设定。当然，如果你所使用的任务队列是无界队列，那么这个参数将形同虚设。

（3）如何创建新的工作线程

在线程池中，新线程的创建是通过`ThreadFactory`完成。你可以通过线程池的构造函数指定特定的ThreadFactory，如未指定将使用默认的`Executors.defaultThreadFactory()`，该工厂所创建的线程具有相同的ThreadGroup和优先级（NORM_PRIORITY），并且都不是守护（Non-Daemon）线程。

通过设定ThreadFactory，你可以自定义线程的名字、线程组以及守护状态等。

在Java的线程池ThreadPoolExecutor中，addWorker方法负责新线程的具体创建工作。

```
  private boolean addWorker(Runnable firstTask, boolean core)
{...}
```

（4）保活时间

保活时间指的是非核心线程在空闲时所能存活的时间。

如果线程池中的线程数量超过了`corePoolSize`中的设定，那么空闲线程的空闲时间在超过`keepAliveTime`中设定的时间后，线程将被回收终止。在线程被回收后，如果需要新的线程时，将继续创建新的线程。

需要注意的是，`keepAliveTime`仅对非核心线程有效，如果需要设置核心线程的保活时间，需要使用`allowCoreThreadTimeOut`参数。

（5）钩子方法

- **设定任务执行前动作：beforeExecute**

如果你希望提交的任务在执行前执行特定的动作，比如写入日志或设定ThreadLocal等。那么，你可以通过重写beforeExecute来实现这一目的。

```
protected void beforeExecute(Thread t, Runnable r) { }
```

- **设定任务执行后动作：afterExecute**
  如果你希望提交的任务在执行后执行特定的动作，比如写入日志或捕获异常等。那么，你可以通过重写afterExecute来实现这一目的。

```
protected void afterExecute(Runnable r, Throwable t) { }
```

- **设定线程池终止动作：terminated**

```
protected void terminated() { }
```

（6）线程池的预热

默认情况下，在设置核心线程数之后，也不会立即创建相关线程，而是任务到达后再创建。

如果你需要预先就启动核心线程，那么你可以通过调用`prestartCoreThread`或`prestartAllCoreThreads`来提前启动，以达到线程池预热目的，并且可以通过`ensurePrestart`方法来验证效果。

（7）线程回收机制

当线程池中的工作线程数量大于corePoolSize设置的数量时，并且存在空闲线程，并且这个空闲线程的空闲时长超过了keepAliveTime所设置的时长，那么这样的空闲线程将会被回收，以降低不必要的资源浪费。

```
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            while (task != null || (task = getTask()) != null) {
           ...
        } finally {
```

```
            processWorkerExit(w, completedAbruptly); // 主动回收自
己
        }
    }
```

（8）线程数调整策略

线程池的工作线程的设置是否合理，关系到系统负载和任务处理速度之间的平衡。这里要明确的是，**如何设置核心线程并没有放之四海而皆准的公式**。每个业务场景都有着它独特的地方，CPU密集型和IO密集型任务存在较大差异。因此，在使用线程池的时候，要具体问题具体分析，但是你可以运行结果持续调整来优化线程池。

5. 线程池使用示例

我们仍以手工制作线程池部分的场景为例，通过ThreadPoolExecutor实现来展示线程池的使用示例。从代码中看，ThreadPoolExecutor的使用和王者线程池TheKingThreadPool的用法基本一致。

```
public static void main(String[] args) {
    ThreadPoolExecutor threadPoolExecutor = new
ThreadPoolExecutor(3, 20, 1000, TimeUnit.MILLISECONDS, new
ArrayBlockingQueue < > (10));

    String[] wildMonsters = {"棕熊", "野鸡", "灰狼", "野兔", "狐狸",
"小鹿", "小花豹", "野猪"};
    for (String wildMonsterName: wildMonsters) {
        threadPoolExecutor.execute(new RunnableTask() {
            public String getTaskDesc() {
                return wildMonsterName;
            }

            public void run() {

System.out.println(Thread.currentThread().getName() + ":" +
wildMonsterName + "已经烤好");
            }
        });
    }
```

```
    threadPoolExecutor.shutdown();
}
```

6. Executors类

Executors是JUC中一个针对ThreadPoolExecutor和ThreadFactory等设计的一个工具类。通过Executors，可以方便地创建不同类型的线程池。当然，其内部主要是通过给ThreadPoolExecutor的构造传递特定的参数实现，并无玄机可言。常用的几个工具如下所示：

- **创建固定线程数的线程池**

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

- **创建只有1个线程的线程池**

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

- **创建缓存线程池**：这种线程池不设定核心线程数，根据任务的数据动态创建线程。当任务执行结束后，线程会被逐步回收，也就是所有的线程都是临时的。

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

```
    }
```

7. 线程池监控

作为一个运行框架，ThreadPoolExecutor既简单也复杂。因此，对其内部的监控和管理是十分必要的。ThreadPoolExecutor也提供了一些方法，通过这些方法，我们可以获取到线程池的一些重要状态和数据。

- **获取线程池大小**

```
public int getPoolSize() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // Remove rare and surprising possibility of
        // isTerminated() && getPoolSize() > 0
        return runStateAtLeast(ctl.get(), TIDYING) ? 0 :
            workers.size();
    } finally {
        mainLock.unlock();
    }
}
```

- **获取活跃工作线程数量**

```
public int getActiveCount() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        int n = 0;
        for (Worker w: workers)
            if (w.isLocked())
                ++n;
        return n;
    } finally {
        mainLock.unlock();
    }
}
```

- **获取最大线程池**

```
 public int getLargestPoolSize() {
     final ReentrantLock mainLock = this.mainLock;
     mainLock.lock();
     try {
         return largestPoolSize;
     } finally {
         mainLock.unlock();
     }
 }
```

- 获取线程池中的任务总数

```
 public long getTaskCount() {
     final ReentrantLock mainLock = this.mainLock;
     mainLock.lock();
     try {
         long n = completedTaskCount;
         for (Worker w: workers) {
             n += w.completedTasks;
             if (w.isLocked())
                 ++n;
         }
         return n + workQueue.size();
     } finally {
         mainLock.unlock();
     }
 }
```

- 获取线程池中已完成的任务总数

```
public long getCompletedTaskCount() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        long n = completedTaskCount;
        for (Worker w: workers)
            n += w.completedTasks;
        return n;
    } finally {
        mainLock.unlock();
    }
}
```

四、如何养成正确使用线程池的良好习惯

1. 线程池的使用风险提示

虽然线程池的使用有诸多的好处，然而天下没有免费的午餐，线程池在给我们带来便利的同时，也有一些避免踩坑的注意事项：

- **线程池设置过大或过小都不合适**。如果线程池的线程数量过多，虽然局部处理速度增加，但将会影响应用系统的整体性能。而如果线程池的线程数量过少，线程池可能无法带来预期的性能的提升；
- **和其他多线程类似，线程池中也可能会发生死锁**。比如，某个任务等待另外一个任务结束，但却没有线程来执行等待的那个任务，这也是为什么要避免任务间存在依赖；
- **添加任务到队列时耗时过长**。如果任务队列已满，外部线程向队列添加任务将会受阻。所以，为了避免外部线程阻塞时间过长，你可以设定最大等待时间；

为了降低这些风险的发生，你在设置线程池的类型和参数时，应当格外小心。在正式上线前，最好能做一次压力测试。

2. 创建线程池的推荐姿势

虽然通过Executors创建线程比较方便，但是Executors的封装屏蔽了一些重要的参数细节，而这些参数对于线程池至关重要，**所以为了避免因对Executors不了解而错误地使用线程池，建议还是通过ThreadPoolExecutor的构造参数直接创建**。

3. 尽量避免使用无界队列

如果再认真点说的话，你应该在任何时候都避免使用无界队列来管理任务。注意，Executors的newFixedThreadPool所使用的是LinkedBlockingQueue，上文有它的源码。

小结

以上就是关于Java线程池的全部内容。在这篇文章中，我们讲解了线程池的应用场景、核心组成及原理，并手工制作了一个线程池，而且在此基础上深入讲解了Java中的线程池ThreadPoolExecutor的实现。虽然文章整体篇幅较大，但是由于线程池涉及的内容十分广泛，难以在一篇文章中全部提及，仍有部分重要内容未能覆盖，比如如何处理线程池中的异常、如何优雅关闭线程池等。

熟练掌握线程池并不是一件容易的事，建议按照本文开篇的建议，先理解其要解决的问题，再理解其核心组成原理，最后再深入到Java中的源码中。如此一来，带着已知的概念去看源码，会更容易理解源码的设计之道。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 思考：如何确保线程池不丢失任务。

延伸阅读与参考资料

- https://stackoverflow.com/questions/5483047/why-is-creating-a-thread-said-to-be-expensive
- http://tutorials.jenkov.com/java-concurrency/thread-pools.html
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

钻石02：分而治之-如何从原理深入理解ForkJoinPool的快与慢

欢迎来到《王者并发课》，本文是该系列文章中的**第25篇**，砖石中的**第2篇**。

在上一篇文章中，我们学习了线程池ThreadPoolExecutor，它通过对任务队列和线程的有效管理实现了对并发任务的处理。然而，ThreadPoolExecutor有两个明显的缺点：**一是无法对大任务进行拆分，对于某个任务只能由单线程执行；二是工作线程从队列中获取任务时存在竞争情况**。这两个缺点都会影响任务的执行效率，要知道高并发场景中的每一毫秒都弥足珍贵。

针对这两个问题，本文即将介绍的**ForkJoinPool**给出了可选的答案。在本文中，我们将首先从分治算法开始介绍，接着体验ForkJoinPool中自定义任务的实现，最后再深入到Java中去理解ForkJoinPool的原理和用法。

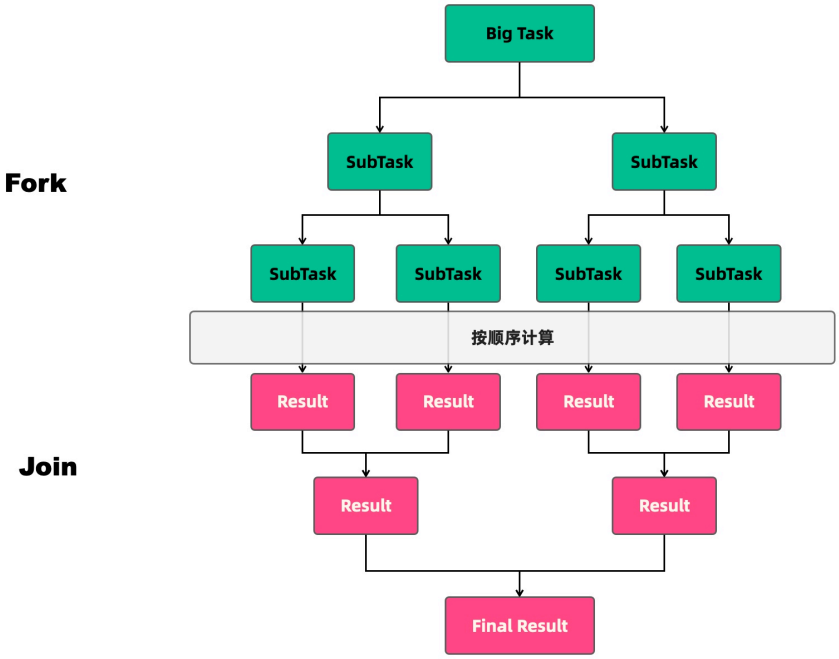
本文大约2万字，篇幅较长，在阅读时建议先看目录再看内容或先收藏。

一、分治算法与Fork/Join模式

在并发计算中，Fork/Join模式往往用于对大任务的并行计算，它通过递归的方式对任务不断地拆解，再将结果进行合并。如果从其思想上看，Fork/Join并不复杂，其本质是**分治算法（Divide-and-Conquer）** 的应用。

分治算法的**基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。** 分治算法的步骤如下：

- **（1）分解**：将要解决的问题划分成若干规模较小的同类问题；
- **（2）求解**：当子问题划分得足够小时，用较简单的方法解决；
- **（3）合并**：按原问题的要求，将子问题的解逐层合并构成原问题的解。



Fork/Join对任务的拆分和对结果合并过程也是如此，可以用下面伪代码来表示：

```
solve(problem):
    if problem is small enough:
        // 如果任务足够小，执行任务
        solve problem directly (sequential algorithm)
    else:
        // 拆分任务
        for part in subdivide(problem)
            fork subtask to solve(part)
        // 合并结果
        join all subtasks spawned in previous loop
        return combined results
```

所以，理解Fork/Join模型和ForkJoinPool线程池，首先要理解其背后的算法的目的和思想，因为后文所要详述的ForkJoinPool不过只是这种算法的一种的实现和应用。

二、Fork/Join应用场景与体验

按照王者并发课所提倡的**思想->实现->源码**的思路，在了解了Fork/Join思想之后，

我们先通过一个场景手工实现一个RecursiveTask，这样可以更好地体验Fork/Join的用法。

> 场景：给定两个自然数，计算两个两个数之间的总和。比如1~n之间的和：1+2+3+...+n

为了解决这个问题，我们创建了TheKingRecursiveSumTask这个核心类，它继承于RecursiveTask. RecursiveTask是ForkJoinPool中的一种任务类型，你暂且不必深入了解它，后文会有详细描述。

TheKingRecursiveSumTask中定义了任务计算的起止范围（`sumBegin`和`sumEnd`）和拆分阈值（`threshold`），以及核心计算逻辑`compute()`.

```
public class TheKingRecursiveSumTask extends RecursiveTask<Long>
{
    private static final AtomicInteger taskCount = new
AtomicInteger();
    private final int sumBegin;
    private final int sumEnd;
    /**
     * 任务拆分阈值，当任务尺寸大于该值时，进行拆分
     */
    private final int threshold;

    public TheKingRecursiveSumTask(int sumBegin, int sumEnd, int
threshold) {
        this.sumBegin = sumBegin;
        this.sumEnd = sumEnd;
        this.threshold = threshold;
    }

    @Override
    protected Long compute() {
        if ((sumEnd - sumBegin) > threshold) {
            // 两个数之间的差值大于阈值，拆分任务
            TheKingRecursiveSumTask subTask1 = new
TheKingRecursiveSumTask(sumBegin, (sumBegin + sumEnd) / 2,
threshold);
            TheKingRecursiveSumTask subTask2 = new
TheKingRecursiveSumTask((sumBegin + sumEnd) / 2, sumEnd,
threshold);
```

```
            subTask1.fork();
            subTask2.fork();
            taskCount.incrementAndGet();
            return subTask1.join() + subTask2.join();
        }
        // 直接执行结果
        long result = 0L;
        for (int i = sumBegin; i < sumEnd; i++) {
            result += i;
        }
        return result;
    }

    public static AtomicInteger getTaskCount() {
        return taskCount;
    }
}
```

在下面的代码中，我们设置的计算区间值0~10000000，**当计算的个数超过100时，将对任务进行拆分**，最大并发数设置为**16**.

```
 public static void main(String[] args) {
     int sumBegin = 0, sumEnd = 10000000;
     computeByForkJoin(sumBegin, sumEnd);
     computeBySingleThread(sumBegin, sumEnd);
 }

 private static void computeByForkJoin(int sumBegin, int sumEnd)
{
     ForkJoinPool forkJoinPool = new ForkJoinPool(16);
     long forkJoinStartTime = System.nanoTime();
     TheKingRecursiveSumTask theKingRecursiveSumTask = new
TheKingRecursiveSumTask(sumBegin, sumEnd, 100);
     long forkJoinResult =
forkJoinPool.invoke(theKingRecursiveSumTask);
     System.out.println("======");
     System.out.println("ForkJoin任务拆分: " +
TheKingRecursiveSumTask.getTaskCount());
     System.out.println("ForkJoin计算结果: " + forkJoinResult);
     System.out.println("ForkJoin计算耗时: " + (System.nanoTime() -
forkJoinStartTime) / 1000000);
 }
```

```
 private static void computeBySingleThread(int sumBegin, int
sumEnd) {
     long computeResult = 0 L;
     long startTime = System.nanoTime();
     for (int i = sumBegin; i < sumEnd; i++) {
         computeResult += i;
     }
     System.out.println("======");
     System.out.println("单线程计算结果: " + computeResult);
     System.out.println("单线程计算耗时: " + (System.nanoTime() -
startTime) / 1000000);
 }
```

运行结果如下:

```
======
ForkJoin任务拆分: 131071
ForkJoin计算结果: 49999995000000
ForkJoin计算耗时: 207
======
单线程计算结果: 49999995000000
单线程计算耗时: 40

Process finished with exit code 0
```

从计算结果中可以看到，ForkJoinPool总共进行了131071次的任务拆分，最终的计算结果是49999995000000，耗时207毫秒。

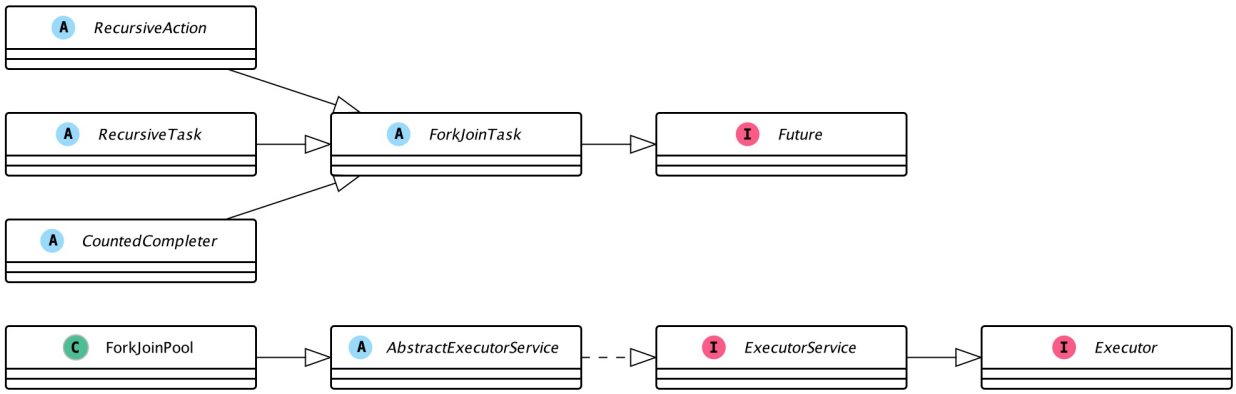
不过，细心的你可能已经发现了，**ForkJoin的并行计算的耗时竟然比单程程还慢?并且足足慢了近5倍**！先别慌，关于ForkJoin的性能问题，我们会在后文有讲解。

三、ForkJoinPool设计与源码分析

在Java中，ForkJoinPool是Fork/Join模型的实现，于Java7引入并在Java8中广泛应用。ForkJoinPool允许其他线程向它提交任务，并根据设定将这些任务拆分为粒度更细的子任务，这些子任务将由ForkJoinPool内部的工作线程来并行执行，并且工作线程之间可以窃取彼此之间的任务。

在接口实现和继承关系上，ForkJoinPool和ThreadPoolExecutor类似，都实现了

Executor和ExecutorService接口，并继承了AbstractExecutorService抽类。而在任务类型上，ForkJoinPool主要有两种任务类型：**RecursiveAction**和**RecursiveTask**，它们继承于ForkJoinTask. 相关关系如下图所示：



解读ForkJoinPool的源码并不容易，虽然它的思想较为简单，但在实现上要考虑的显然更多，加上部分代码可读性一般，所以讲解它的全部源码是不现实的，当然也是没必要的。在下文中，我们将主要介绍其核心的**任务提交和执行**相关的部分源码，其他源码有兴趣的可以自行阅读。

1. 构造ForkJoinPool的几种不同方式

ForkJoinPool中有四个核心参数，用于控制线程池的**并行数**、**工作线程的创建**、**异常处理**和**模式指定**等。各参数解释如下：

- `int parallelism`：指定并行级别（parallelism level）。ForkJoinPool将根据这个设定，决定工作线程的数量。如果未设置的话，将使用`Runtime.getRuntime().availableProcessors()`来设置并行级别；
- `ForkJoinWorkerThreadFactory factory`：ForkJoinPool在创建线程时，会通过factory来创建。注意，这里需要实现的是ForkJoinWorkerThreadFactory，而不是ThreadFactory. 如果你不指定factory，那么将由默认的DefaultForkJoinWorkerThreadFactory负责线程的创建工作；
- `UncaughtExceptionHandler handler`：指定异常处理器，当任务在运行中出错时，将由设定的处理器处理；
- `boolean asyncMode`：从名字上看，你可能会觉得它是**异步模式**设置，但其实是设置队列的工作模式：`asyncMode ? FIFO_QUEUE : LIFO_QUEUE`. 当

asyncMode为true时，将使用先进先出队列，而为false时则使用后进先出的模式。

围绕上面的四个核心参数，ForkJoinPool提供了三种构造方式，使用时你可以根据需要选择其中的一种。

（1）方式一：默认无参构造

在该构造方式中，你无需设定任何参数。ForkJoinPool将根据当前处理器数量来设置并行数量，并使用默认的线程构造工厂。**不推荐**。

```
 public ForkJoinPool() {
        this(Math.min(MAX_CAP,
Runtime.getRuntime().availableProcessors()),
             defaultForkJoinWorkerThreadFactory, null, false);
 }
```

（2）方式二：通过并行数构造

在该构造方式中，你可以指定并行数量，以更有效地平衡处理器数量和负载。**建议在设置时，并行级别应低于当前处理器的数量**。

```
 public ForkJoinPool(int parallelism) {
        this(parallelism, defaultForkJoinWorkerThreadFactory,
null, false);
 }
```

（2）方式三：自定义全部参数构造

以上两种构造方式都是基于这种构造，它允许你配置所有的核心参数。为了更有效地管理ForkJoinPool，**建议你使用这种构造方式**。

```
public ForkJoinPool(int parallelism,
                    ForkJoinWorkerThreadFactory factory,
                    UncaughtExceptionHandler handler,
                    boolean asyncMode) {
        this(checkParallelism(parallelism),
             checkFactory(factory),
             handler,
```

```
            asyncMode ? FIFO_QUEUE : LIFO_QUEUE,
            "ForkJoinPool-" + nextPoolId() + "-worker-");
        checkPermission();
}
```

2. 按类型提交不同任务

任务提交是ForkJoinPool的核心能力之一，在提交任务时你有三种选择，如下面表格所示：

	从非fork/join线程调用	从fork/join调用
提交异步执行	execute(ForkJoinTask)	ForkJoinTask.fork()
等待并获取结果	invoke(ForkJoinTask)	ForkJoinTask.invoke()
提交执行获取Future结果	submit(ForkJoinTask)	ForkJoinTask.fork() (ForkJoinTasks are Futures)

（1）第一类核心方法：invoke

invoke类型的方法接受ForkJoinTask类型的任务，并在任务执行结束后，返回泛型结果。如果提交的任务是null，将抛出空指针异常。

```
public <T> T invoke(ForkJoinTask<T> task) {
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
        return task.join();
}
```

（2）第二类核心方法：execute

execute类型的方法在提交任务后，不会返回结果。另外要注意的是，ForkJoinPool不仅允许提交ForkJoinTask类型任务，还允许提交**Callable**或**Runnable**任务，因此你可以像使用现有Executors一样使用ForkJoinPool。

当然，Callable或Runnable类型任务时，将会转换为ForkJoinTask类型，具体可以查看任务提交的相关源码。那么，这类任务和直接提交ForkJoinTask任务有什么区别呢？还是有的。**区别在于，由于任务是不可切分的，所以这类任务无法获得任务拆分这方面的效益，不过仍然可以获得任务窃取带来的好处和性能提升。**

```
 public void execute(ForkJoinTask<?> task) {
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
 }

public void execute(Runnable task) {
        if (task == null)
            throw new NullPointerException();
        ForkJoinTask<?> job;
        if (task instanceof ForkJoinTask<?>) // avoid re-wrap
            job = (ForkJoinTask<?>) task;
        else
            job = new ForkJoinTask.RunnableExecuteAction(task);
        externalPush(job);
 }
```

（3）第三类核心方法：submit

submit类型的方法支持三种类型的任务提交：ForkJoinTask类型、Callable类型和Runnable类型。在提交任务后，将返回ForkJoinTask类型的结果。如果提交的任务是null，将抛出空指针异常，并且当任务不能按计划执行的话，将抛出任务拒绝异常。

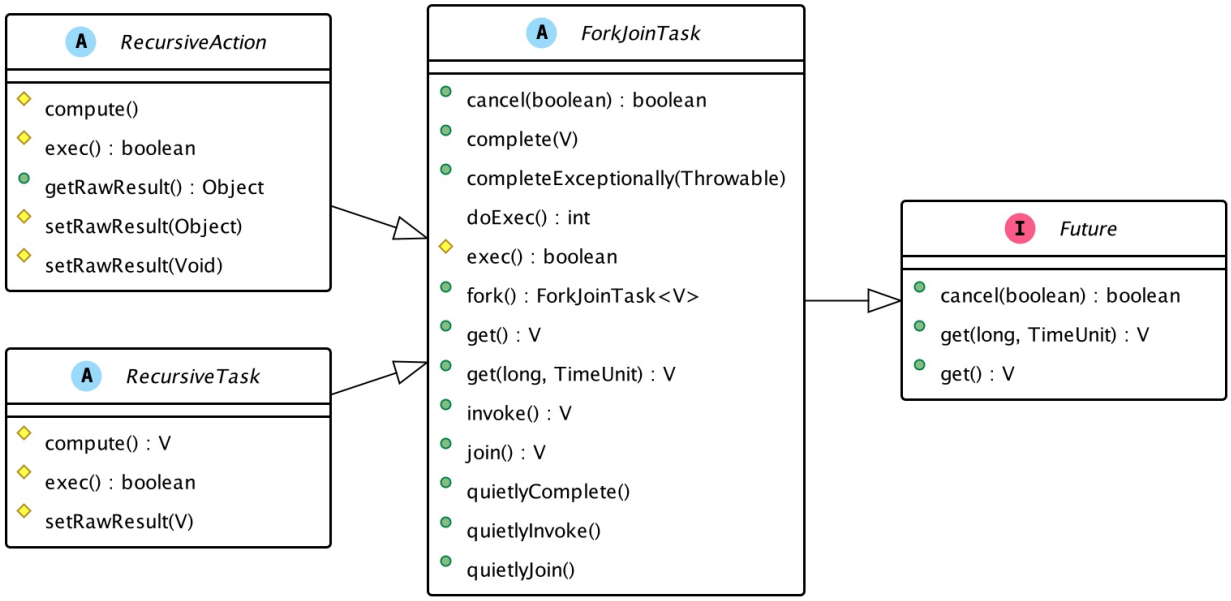
```
   public < T > ForkJoinTask < T > submit(ForkJoinTask < T >
task) {
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
        return task;
   }

   public < T > ForkJoinTask < T > submit(Callable < T > task) {
        ForkJoinTask < T > job = new ForkJoinTask.AdaptedCallable
< T > (task);
        externalPush(job);
```

```
        return job;
    }

    public < T > ForkJoinTask < T > submit(Runnable task, T
result) {
        ForkJoinTask < T > job = new ForkJoinTask.AdaptedRunnable
< T > (task, result);
        externalPush(job);
        return job;
    }

    public ForkJoinTask < ? > submit(Runnable task) {
        if (task == null)
            throw new NullPointerException();
        ForkJoinTask < ? > job;
        if (task instanceof ForkJoinTask < ? > ) // avoid re-wrap
            job = (ForkJoinTask < ? > ) task;
        else
            job = new ForkJoinTask.AdaptedRunnableAction(task);
        externalPush(job);
        return job;
    }
```

3. ForkJoinTask

ForkJoinTask是ForkJoinPool的核心之一，它是任务的实际载体，定义了任务执行时的具体逻辑和拆分逻辑，本文前面的示例代码就是通过继承它实现。作为一个抽象类，ForkJoinTask的行为有点类似于线程，但它更为轻量，因为它不维护自己的运行时堆栈或程序计数器等。

在类的设计上，ForkJoinTask继承了Future接口，所以也可以将其看作是轻量级的Future，它们之间的关系如下图所示。



（1）fork与join

`fork()`/`join()`是ForkJoinTask甚至是ForkJoinPool的核心方法，承载着主要的任务协调作用，一个用于任务提交，一个用于结果获取。

fork-提交任务

`fork()`方法用于向**当前任务所运行的线程池**中提交任务，比如上文示例代码中的`subTask1.fork()`. 注意，不同于其他线程池的写法，任务提交由任务自己通过调用`fork()`完成，对此不要感觉诧异，`fork()`内部会将任务与当前线程进行关联。

从源码中看，如果当前线程是ForkJoinWorkerThread类型，将会放入该线程的任务队列，否则放入common线程池的任务队列中。**关于common线程池，后续会有介绍。**

```
    public final ForkJoinTask<V> fork() {
        Thread t;
        if ((t = Thread.currentThread()) instanceof
ForkJoinWorkerThread)
            ((ForkJoinWorkerThread)t).workQueue.push(this);
        else
            ForkJoinPool.common.externalPush(this);
        return this;
```

```
    }
```

join-获取任务执行结果

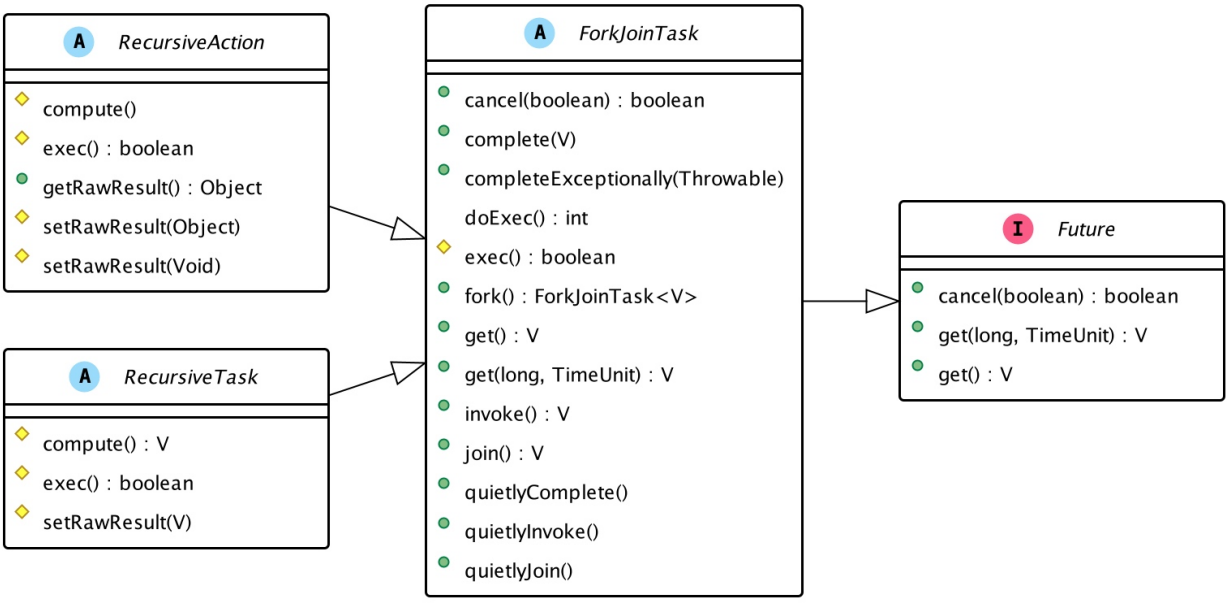
前面，你已经知道可以通过`fork()`提交任务。那么现在，你则可以通过`join()`方法获取任务的执行结果。

调用`join()`时，将阻塞当前线程直到对应的子任务完成运行并返回结果。从源码看，`join()`的核心逻辑由`doJoin()`负责。`doJoin()`虽然很短，但可读性较差，阅读时稍微忍一下。

```
public final V join() {
    int s;
    // 如果调用doJoin返回的非NORMAL状态，将报告异常
    if ((s = doJoin() & DONE_MASK) != NORMAL)
        reportException(s);
    // 正常执行结束，返回原始结果
    return getRawResult();
}

private int doJoin() {
    int s;
    Thread t;
    ForkJoinWorkerThread wt;
    ForkJoinPool.WorkQueue w;
    //如果已完成，返回状态
    return (s = status) < 0 ? s :
      //如果未完成且当前线程是ForkJoinWorkerThread，则从该线程中取出
workQueue，并尝试将当前task取出执行。如果执行的结果是完成，则返回状态；否
则，使用当前线程池awaitJoin方法进行等待
        ((t = Thread.currentThread()) instanceof
ForkJoinWorkerThread) ?
        (w = (wt = (ForkJoinWorkerThread) t).workQueue).
    tryUnpush(this) && (s = doExec()) < 0 ? s :
        wt.pool.awaitJoin(w, this, 0 L):
     //当前线程非ForkJoinWorkerThread，调用externalAwaitDone方法等待
        externalAwaitDone();
}

final int doExec() {
    int s;
    boolean completed;
    if ((s = status) >= 0) {
```

```
            try {
                completed = exec();
            } catch (Throwable rex) {
                return setExceptionalCompletion(rex);
            }
            // 执行完成后，将状态设置为NORMAL
            if (completed)
                s = setCompletion(NORMAL);
        }
        return s;
    }
```

（2）RecursiveAction与RecursiveTask



在ForkJoinPool中，常用的有两种任务类型：**返回结果**的和**不返回结果**的，这方面和ThreadPoolExecutor等线程池是一致的，对应的两个类分别是：**RecursiveAction**和**RecursiveTask**. 从类图中可以看到，它们均继承于ForkJoinTask.

RecursiveAction：无结果返回

RecursiveAction用于递归执行但不需要返回结果的任务，比如下面的排序就是它的典型应用场景。在使用RecursiveAction时，你需要继承并实现它的核心方法`compute()`.

```
static class SortTask extends RecursiveAction {
    final long[] array;
    final int lo, hi;
    SortTask(long[] array, int lo, int hi) {
        this.array = array;
        this.lo = lo;
        this.hi = hi;
    }
    SortTask(long[] array) {
        this(array, 0, array.length);
    }
    // 核心计算方法
    protected void compute() {
        if (hi - lo < THRESHOLD)
            // 直接执行
            sortSequentially(lo, hi);
        else {
            // 拆分任务
            int mid = (lo + hi) >>> 1;
            invokeAll(new SortTask(array, lo, mid),
                new SortTask(array, mid, hi));
            merge(lo, mid, hi);
        }
    }
    // implementation details follow:
    static final int THRESHOLD = 1000;
    void sortSequentially(int lo, int hi) {
        Arrays.sort(array, lo, hi);
    }
    void merge(int lo, int mid, int hi) {
        long[] buf = Arrays.copyOfRange(array, lo, mid);
        for (int i = 0, j = lo, k = mid; i < buf.length; j++)
            array[j] = (k == hi || buf[i] < array[k]) ?
            buf[i++] : array[k++];
    }
}
```

RecursiveTask：返回结果

RecursiveTask用于递归执行需要返回结果的任务，比如前面示例代码中的求和或下面这段求斐波拉契数列求和都是它的典型应用场景。在使用RecursiveTask时，你也需要继承并实现它的核心方法`compute()`.

```
class Fibonacci extends RecursiveTask<Integer> {
  final int n;
  Fibonacci(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1)
      return n;
    Fibonacci f1 = new Fibonacci(n - 1);
    f1.fork();
    Fibonacci f2 = new Fibonacci(n - 2);
    return f2.compute() + f1.join();
  }
}
```

（3）ForkJoinTask使用限制

虽然在某些场景下，ForkJoinTask可以通过任务拆解的方式提高执行效率，但是需要注意的是它并非适合所有的场景。**ForkJoinTask在使用时需要谨记一些限制，违背这些限制可能会适得其反甚至引来灾难**。

为什么这么说呢?

这是因为，ForkJoinTask最适合用于纯粹的**计算任务**，也就是**纯函数计算**，计算过程中的对象都是独立的，对外部没有依赖。你可以想象，如果大量的任务或被拆分的子任务之间彼此依赖或对外部存在严重阻塞依赖，那将是怎样的画面...用千丝万缕来形容也不为过，外部依赖会带来**任务执行**和**问题排查**方面的双重不确定性。

所以，在理想情况下，提交到ForkJoinPool中的任务**应避免执行阻塞I/O**，以免出现不可控的意外情况。当然，这也并非是绝对的，在必要时你也可以定义和使用可阻塞的ForkJoinTask，只不过你需要付出更多的代价和考虑，使用时应当慎之又慎，本文对此不作叙述。

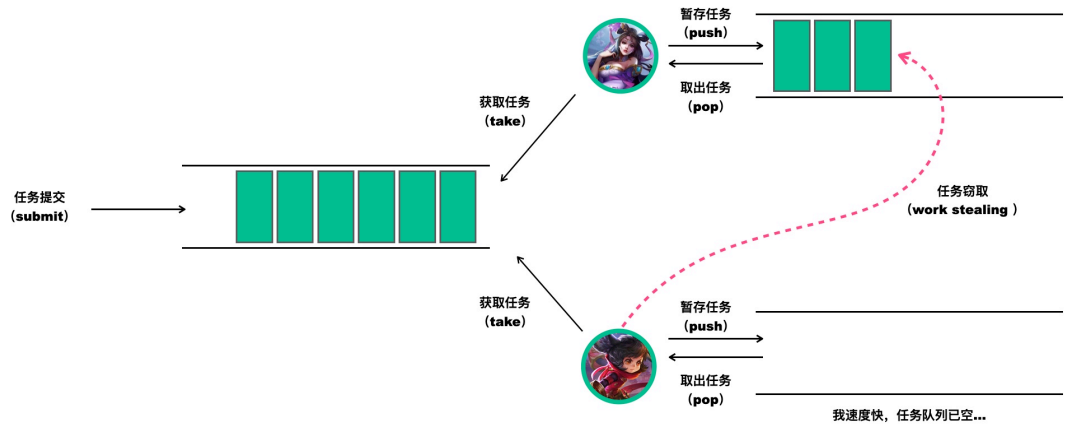
4. 工作队列与任务窃取

前面已经说到，ForkJoinPool与ThreadPoolExecutor有个很大的不同之处在于，ForkJoinPool存在引入了**任务窃取**设计，它是其性能保证的关键之一。

关于任务窃取，简单来说，**就是允许空闲线程从繁忙线程的双端队列中窃取任务**。

默认情况下，工作线程从它自己的双端队列的**头部**获取任务。但是，当自己的任务为空时，线程会从其他繁忙线程双端队列的**尾部**中获取任务。这种方法，最大限度地减少了线程竞争任务的可能性。

ForkJoinPool的大部分操作都发生在**工作窃取队列（work-stealing queues）**中，该队列由内部类WorkQueue实现。其实，这个队列也不是什么神奇之物，它是Deques的特殊形式，但仅支持三种操作方式：**push**、**pop**和**poll**（也称为窃取）。当然，在ForkJoinPool中，队列的读取有着严格的约束，`push`和`pop`仅能从其所属线程调用，而`poll`则可以从其他线程调用。换句话说，前两个方法是留给自己用的，而第三种方法则是为了方便别人来窃取任务用的。**任务窃取的相关过程，可以用下面这幅图来表示，这幅图建议你收藏**：



看到这里，不知你是否会有疑问：**为什么工作线程总是从自己的头部获取任务？为什么要这样设计？首先处理队列中等待时间较长的任务难道不是更有意义吗**？

答案当然不会是“**更有意义**”。**这样做的主要原因是为了提高性能，通过始终选择最近提交的任务，可以增加资源仍分配在CPU缓存中的机会，这样CPU处理起来要快一些**。而窃取者之所以从尾部获取任务，**则是为了降低线程之间的竞争可能，毕竟大家都从一个部分拿任务，竞争的可能要大很多**。

此外，这样的设计还有一种考虑。**由于任务是可分割的，那队列中较旧的任务最有可能粒度较大，因为它们可能还没有被分割，而空闲的线程则相对更有“精力”来完成这些粒度较大的任务**。

5. ForkJoinPool监控

对于一个复杂框架来说，实时地了解ForkJoinPool的内部状态是十分必要的。因此，ForkJoinPool提供了一些常用方法。通过这些方法，你可以了解当前的工作线程、任务处理等情况。

（1）获取运行状态的线程总数

```
public int getRunningThreadCount() {
    int rc = 0;
    WorkQueue[] ws;
    WorkQueue w;
    if ((ws = workQueues) != null) {
        for (int i = 1; i < ws.length; i += 2) {
            if ((w = ws[i]) != null && w.isApparentlyUnblocked())
                ++rc;
        }
    }
    return rc;
}
```

（2）获取活跃线程数量

```
public int getActiveThreadCount() {
        int r = (config & SMASK) + (int)(ctl >> AC_SHIFT);
        return (r <= 0) ? 0 : r; // suppress momentarily negative
values
}
```

（3）判断ForkJoinPool是否空闲

```
public boolean isQuiescent() {
    return (config & SMASK) + (int)(ctl >> AC_SHIFT) <= 0;
}
```

（4）获取任务窃取数量

```
public long getStealCount() {
```

```
    AtomicLong sc = stealCounter;
    long count = (sc == null) ? 0 L : sc.get();
    WorkQueue[] ws;
    WorkQueue w;
    if ((ws = workQueues) != null) {
        for (int i = 1; i < ws.length; i += 2) {
            if ((w = ws[i]) != null)
                count += w.nsteals;
        }
    }
    return count;
}
```

（5）获取队列中的任务数量

```
public long getQueuedTaskCount() {
    long count = 0;
    WorkQueue[] ws;
    WorkQueue w;
    if ((ws = workQueues) != null) {
        for (int i = 1; i < ws.length; i += 2) {
            if ((w = ws[i]) != null)
                count += w.queueSize();
        }
    }
    return count;
}
```

（6）获取已提交的任务数量

```
public int getQueuedSubmissionCount() {
    int count = 0;
    WorkQueue[] ws;
    WorkQueue w;
    if ((ws = workQueues) != null) {
        for (int i = 0; i < ws.length; i += 2) {
            if ((w = ws[i]) != null)
                count += w.queueSize();
        }
    }
    return count;
}
```

四、警惕ForkJoinPool#commonPool

在上文中所示的源码中，你可能已经在多处注意到commonPool的存在。在ForkJoinPool中，commonPool是一个**共享的、静态的**线程池，并且在实际使用时才会进行懒加载，**Java8中的CompletableFuture和并行流（Parallel Streams）用的就是它**。不过，**使用CompletableFuture时你可以指定自己的线程池，但是并行流在使用时却不可以，这也是我们要警惕的地方**。为什么这么说呢?

ForkJoinPool中的commonPool设计初衷是为了降低线程池的重复创建，让一些任务共用同一个线程池，毕竟创建线程池和创建线程都是昂贵的。然而，凡事都有两面性，commonPool在某些场景下确实可以达到线程池复用的目的，但是，**如果你决定与别人分享自己空间，那么当你想使用它的时候，它可能不再完全属于你**。也就是说，当你想用commonPool时，它可能已经其他任务填满了。

提交到ForkJoinPool中的任务一般有两类：**计算类型**和**阻塞类型**。考虑一个场景，应用中多处都在使用这个共享线程池，有人在某处做了个不当操作，比如往池子里丢入了阻塞型任务，那么结果会怎样？结果当然是，**整个线程池都有可能被阻塞**！如此，**整个应用都面临着被拖垮的风险**。看到这里，对于Java8中的并行流的使用，你就应该高度警惕了。

那怎么避免这种情况发生呢？答案是尽量避免使用commonPool，并且在需要运行阻塞任务时，应当创建独立的线程池，和系统的其他部分保持隔离，以免风险扩散。

五、ForkJoinPool性能评估

为了测试ForkJoinPool的性能，我做了一组**简单的**、**非正式**实验。实验分三组进行，为了尽可能让每组的数据客观，每组实验均运行5次，取最后的平均数。

- **实验代码**：本文第一部分的示例代码；
- **实验环境**：Mac；
- **JDK版本**：8；

- **任务分隔阈值**：100

实验结果如下方表格所示：

实验次数	1000量级耗时（毫秒）		1000000量级耗时（毫秒）		1000000000量级（毫秒）	
	Fork/Join	单线程	Fork/Join	单线程	Fork/Join	单
1	4	0	34	5	1157	313
2	3	0	34	6	848	344
3	5	0	16	9	1069	325
4	4	0	35	8	955	307
5	5	0	30	22	922	385
平均	4.2	0	29.8	10	990.2	334

从实验结果（0表示不到1毫秒）来看，**ForkJoinPool的性能竟然不如单线程的效率高**！这样的结果，似乎很惊喜、很意外...**然而，为什么会这样**?

不要惊讶，之所以会出现这个令你匪夷所思的结果，**其原因在于任务拆分的粒度过小**！在上面的测试中，任务拆分阈值仅为100，导致Fork/Join在计算时出现大量的任务拆分动作，也就是任务分的太细，大量的任务拆分和管理也是需要额外成本的。

以0~1000000求和为例，当把阈值从**100**调整为**100000**时，其结果结果如下。可以看到，Fork/Join的优势就体现出来了。

```
======
ForkJoin任务拆分: 16383
ForkJoin计算结果: 499999999500000000
ForkJoin计算耗时: 143
======
单线程计算结果: 499999999500000000
单线程计算耗时: 410
```

那么，问题又来了，哪些因素会影响Fork/Join的性能呢?

根据经验和实验，**任务总数**、**单任务执行耗时**以及**并行数**都会影响到性能。所以，**当你使用Fork/Join框架时，你需要谨慎评估这三个指标，最好能通过模拟对比评估，不要凭感觉冒然在生产环境使用**。

小结

以上就是关于ForkJoinPool的全部内容。Fork/Join是一种基于分治算法的模型，在并发处理计算型任务时有着显著的优势。其效率的提升主要得益于两个方面：

- **任务切分**：将大的任务分割成更小粒度的小任务，让更多的线程参与执行；
- **任务窃取**：通过任务窃取，充分地利用空闲线程，并减少竞争。

在使用ForkJoinPool时，需要特别注意任务的类型是否为**纯函数计算类型**，也就是这些任务不应该关心状态或者外界的变化，这样才是最安全的做法。如果是阻塞类型任务，那么你需要谨慎评估技术方案。虽然ForkJoinPool也能处理阻塞类型任务，但可能会带来复杂的管理成本。

而在性能方面，要认识到Fork/Join的性能并不是开箱即来，而是需要你去评估和验证一些重要指标，通过数据对比得出最佳结论。

此外，ForkJoinPool虽然提供了commonPool，但出于潜在的风险考虑，不推荐使用或谨慎使用。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 动手：使用ForkJoinPool实现List数组排序。

延伸阅读与参考资料

- A Java Fork/Join Framework（Doug Lea）：**http://gee.cs.oswego.edu/dl/**

papers/fj.pdf

- Java Fork and Join using ForkJoinPool：http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html
- Introduction to the Fork/Join Framework：https://www.pluralsight.com/guides/introduction-to-the-fork-join-framework
- The Unfairly Unknown ForkJoinPool：https://medium.com/swlh/the-unfairly-unknown-forkjoinpool-c262777def6a
- A Java? Fork-Join Calamity：http://coopsoft.com/ar/CalamityArticle.html
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

钻石03：琳琅满目-细数CompletableFuture的那些花式玩法

欢迎来到《王者并发课》，本文是该系列文章中的**第26篇**，砖石中的**第3篇**。

从Java8开始，JDK引入了很多新的特性，包括lambda表达式、流式计算等，以及本文所要详述的**CompletableFuture**. 关于CompletableFuture，你可能首先会联想到**Future**接口，对于它我们并不陌生，在ThreadPoolExecutor和ForkJoinPool中都见过它的身影。如果你对此感到困惑的话，不妨先阅读我们的前两篇文章。

Future的接口定义本身并不复杂，使用起来也较为简单，它的核心是`get()`和`isDone()`方法。然而，**Future的简单也导致了它在某些方面会存在先天性的不足**。在某些场景下，Future可能无法满足我们的需求，比如我们无法通过Future实现对并发任务的编排。不过，幸运的是，本文所要介绍的CompletableFuture弥补了Future多方面的不足之处，它可能成为你的最佳之选，这也是本文为什么要谈CompletableFuture的原因。

在这篇文章中，我们将结合Future和线程池，探讨CompletableFuture与Future的不同之处，以及它的核心能力和最佳实践。

一、理解CompletableFuture

1. Future的局限性

从本质上说，**Future表示一个异步计算的结果**。它提供了`isDone()`来检测计算是否已经完成，并且在计算结束后，可以通过`get()`方法来获取计算结果。在异步计算中，Future确实是个非常优秀的接口。但是，它的本身也确实存在着许多限制：

- **并发执行多任务**：Future只提供了get()方法来获取结果，并且是阻塞的。所以，除了等待你别无他法；

- **无法对多个任务进行链式调用**：如果你希望在计算任务完成后执行特定动作，比如发邮件，但Future却没有提供这样的能力；
- **无法组合多个任务**：如果你运行了10个任务，并期望在它们全部执行结束后执行特定动作，那么在Future中这是无能为力的；
- **没有异常处理**：Future接口中没有关于异常处理的方法；

2. CompletableFuture与Future的不同

简单地说，CompletableFuture是Future接口的扩展和增强。CompletableFuture完整地继承了Future接口，并在此基础上进行了丰富地扩展，完美地弥补了Future上述的种种问题。更为重要的是，CompletableFuture**实现了对任务的编排能力**。借助这项能力，我们可以轻松地组织不同任务的运行顺序、规则以及方式。**从某种程度上说，这项能力是它的核心能力**。而在以往，虽然通过CountDownLatch等工具类也可以实现任务的编排，但需要复杂的逻辑处理，不仅耗费精力且难以维护。

3. CompletableFuture初体验

当然，百闻不如一见，既然CompletableFuture如此神乎其神，我们不妨通过一个特定的场景来体验CompletableFuture的用法。

众所周知，王者中有注明的“**草丛三杰（B）**”，妲己就是其中之一，她蹲草丛的本领可谓一绝。话说这天，妲己远远看见地方小鲁班蹦蹦跳跳地走来，对付这样的脆皮最适合在草丛中来一波操作。于是，妲己侧身躲进了草丛，在小鲁班欢快地路过时，妲己一套熟练的**231连招**秒杀了小鲁班。小鲁班死不瞑目，因为他甚至还没看清对手的模样，很快啊！

在这个过程中，包含几组动作：捉拿鲁班、打出技能2、打出技能3以及打出技能1.我们可以通过**CompletableFuture**的链式调用来表达这些动作：

```
CompletableFuture.supplyAsync(CompletableFutureDemo::活捉鲁班)
    .thenAccept(player -> note(player.getName())) // 接收
supplyAsync的结果，获得对方名字
    .thenRun(() -> attack("2技能–偶像魅力：鲁班受到妲己285点法术伤害，并
眩晕1.5秒..."))
    .thenRun(() -> attack("3技能–女王崇拜：妲己放出5团狐火，鲁班受到
```

```
325++点法术伤害..."))
    .thenRun(() -> attack("1技能-灵魂冲击：鲁班受到妲己520点点法术伤
害..."))
    .thenRunAsync(() -> note("鲁班，卒...")); // 使用线程池的其他线程
```

你看，使用CompletableFuture编排动作是不是很容易？在这段只有6行的代码中，我们已经使用了supplyAsync()和thenAccept()等4中不同的方法，并且同时使用了同步和异步。在以往，如果手工实现的话，至少需要洋洋洒洒几十行代码。那CompletableFuture是如何做到如此神功的呢？接着往下看。

二、CompletableFuture的核心设计

总体而言，CompletableFuture实现了**Future**和**CompletionStage**两个接口，并且只有少量的属性。但是，它有近2400余行的代码，并且关系复杂。所以，在核心设计方面，我们不会展开讨论。

现在，你已经知道，Future接口仅提供了`get()`和`isDone`这样的简单方法，仅凭Future无法为CompletableFuture提供丰富的能力。那么，CompletableFuture又是如何扩展自己的能力的呢？这就不得不说**CompletionStage**接口了，它是CompletableFuture核心，也是我们要关注的重点。

顾名思义，根据CompletionStage名字中的“**Stage**”，你可以把它理解为任务编排中的**步骤**。所谓步骤，即任务编排的基本单元，它可以是**一次纯粹的计算**或者是**一个特定的动作**。在一次编排中，会包含多个步骤，这些步骤之间会存在**依赖**、**链式**和**组合**等不同的关系，也存在**并行**和**串行**的关系。这种关系，类似于Pipeline或者流式计算。

既然是编排，就需要维护任务的创建、建立计算关系。为此，CompletableFuture提供了多达**50**多个方法，**在数量上确实庞大且令人瞠目结舌**，想要全部理解显然不太可能，当然也没有必要。虽然CompletableFuture的方法数量众多，但是在理解时仍有规律可循，我们**可以通过分类的方式简化对方法的理解**，理解了**类型**和**变种**，基本上我们也就掌握了CompletableFuture的核心能力。

根据类型，这些方法可以总结为以下**四类**，其他大部分方法都是基于这四种类型的变种：

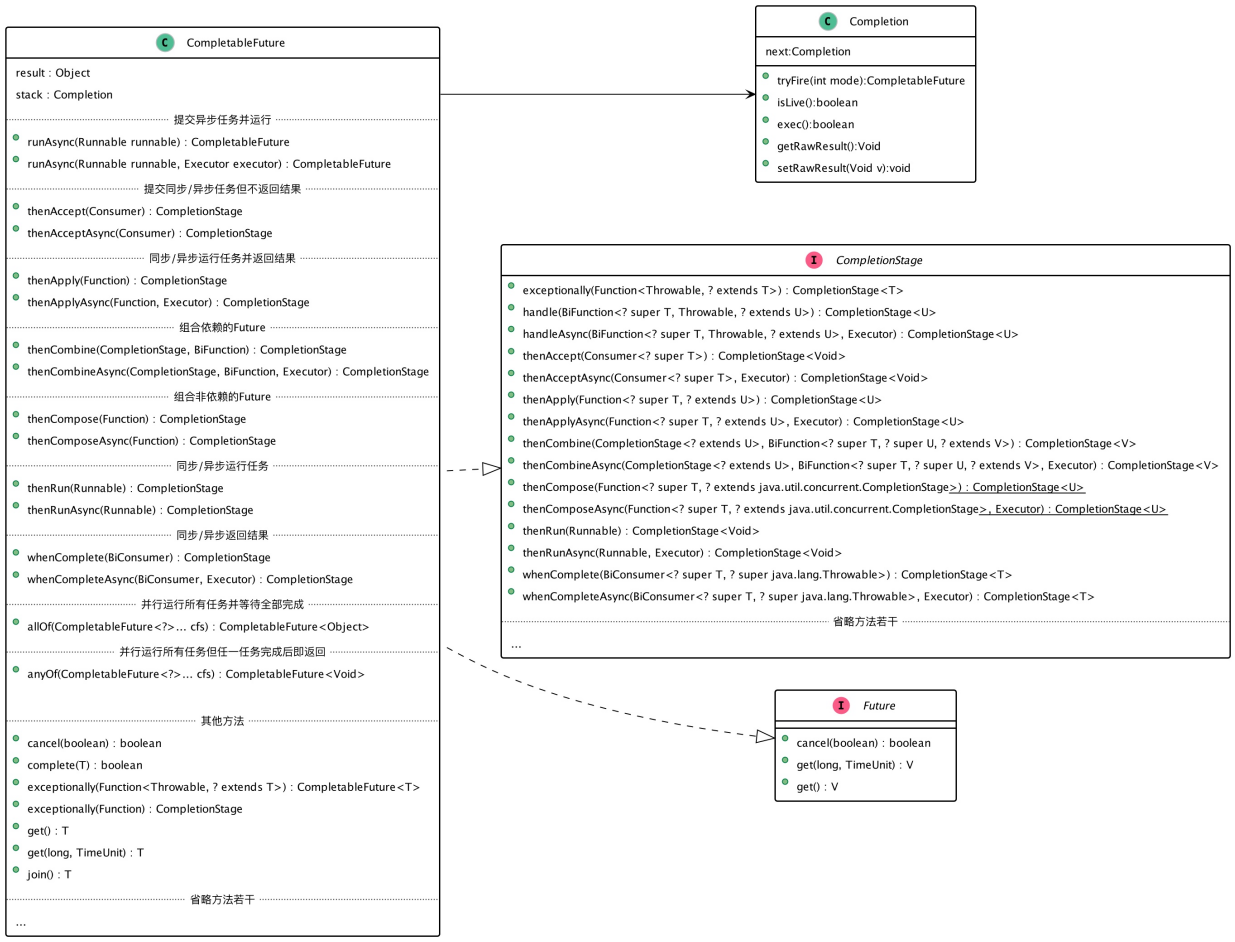
类型	接收参数	返回结果	支持异步
`Supply`	✘	✔	✔
`Apply`	✔	✔	✔
`Accept`	✔	✘	✔
`Run`	✘	✘	✔

关于方法的变种

上述接种类型的方法一般都有三个变种方法：**同步**、**异步**和**指定线程池**。比如，`thenApply()`的三个变种方法如下所示：

```
<U> CompletableFuture<U> thenApply(Function<? super T,? extends
U> fn)
<U> CompletableFuture<U> thenApplyAsync(Function<? super T,?
extends U> fn)
<U> CompletableFuture<U> thenApplyAsync(Function<? super T,?
extends U> fn, Executor executor)
```

下面这幅类图，展示了CompletableFuture和Future、CompletionStage以及Completion之间的关系。当然，由于方法众多，这幅图中并没有全部呈现，而是仅选取了部分重要的方法。



三、CompletableFuture的核心用法

前面已经说过，CompletableFuture的核心方法总共分为四类，而这四类方法又分为两种模式：**同步和异步**。所以，我们从这四类方法中选取了部分核心的API，它们都是我们经常用到的API。

- **同步**：使用当前线程运行任务；
- **异步**：使用CompletableFuture线程池其他线程运行任务，异步方法的名字中带有`Async`.

1. runAsync

`runAsync()`是CompletableFuture最常用的方法之一，它可以接收一个待运行的任务并返回一个CompletableFuture .

当我们想异步运行某个任务时，在以往需要手动实现Thread或者借助Executor实现。而通过runAsync()`就简单多了。比如，我们可以直接传入**Runnable**类型的任务：

```
CompletableFuture.runAsync(new Runnable() {
    @Override
    public void run() {
        note("妲己进入草丛蹲点...等待小鲁班出现");
    }
});
```

此外，在Java8及之后的JDK版本中，我们还可以使用lambda表达式进一步简化代码：

```
CompletableFuture.runAsync(() -> note("妲己进入草丛蹲点...等待小鲁班出现"));
```

这样看起来是不是很简单？相信很多同学也是采用这样的方式来使用`runAsync()`. **不过，如果你也这么用，那么你就要小心了**，这里有陷阱。先卖个关子，文末尾会对CompletableFuture线程池做简要的讲解，帮助你如何避免采坑。

2. supply与supplyAsync

对于`supply()`这个方法，很多人第一印象可能会比较懵，不知道它是做什么的。但其实，它的名字已经说明了一切：所谓“**supply**”当然是提供结果的！换句话说，当我们使用`supply()`时，就**表明我们会返回一个结果，并且这个结果可以被后续的任务所使用**。

举个例子，在下面的示例代码中，我们通过`supplyAsync()`返回了结果，而这个结果在后续的`thenApply()`被使用。

```
// 创建nameFuture，返回姓名
CompletableFuture <String> nameFuture =
CompletableFuture.supplyAsync(() -> {
    return "妲己";
});
```

```
 // 使用thenApply()接收nameFuture的结果，并执行回调动作
CompletableFuture <String> sayLoveFuture =
nameFuture.thenApply(name -> {
    return "爱你，" + name;
});

//阻塞获得表白的结果
System.out.println(sayLoveFuture.get()); // 爱你，妲己
```

你看，一旦理解了`supply()`的含义，它也就如此简单了。如果你希望用新的线程运行任务，可以使用`supplyAsync()`.

3. thenApply与thenApplyAsync

刚才，在前面我们已经介绍了`supply()`，已经知道它是用于提供结果的，并且顺带提了`thenApply()`. 很明显，不用说你可能已经知道`thenApply()`是`supply()`的搭档，用于接收`supply()`的执行结果，并执行特定的代码逻辑，最后返回CompletableFuture结果。

```
 // 使用thenApply()接收nameFuture的结果，并执行回调动作
CompletableFuture <String> sayLoveFuture =
nameFuture.thenApply(name -> {
    return "爱你，" + name;
});

public <U> CompletableFuture <U> thenApplyAsync(
    Function <? super T, ? extends U> fn) {
    return uniApplyStage(null, fn);
}
```

4. thenAccept与thenAcceptAsync

作为`supply()`的档案，`thenApply()`并不是唯一的存在，`thenAccept()`也是。但与`thenApply()`不同，`thenAccept()`**只接收数据，但不会返回，它的返回类型是Void**.

```
CompletableFuture<Void> sayLoveFuture =
nameFuture.thenAccept(name -> {
     System.out.println("爱你, " + name);
});

public CompletableFuture < Void > thenAccept(Consumer < ? super T
> action) {
    return uniAcceptStage(null, action);
}
```

5. thenRun

`thenRun()`就比较简单了，**不接收任务的结果，只运行特定的任务，并且也不返回结果。**

```
public CompletableFuture < Void > thenRun(Runnable action) {
   return uniRunStage(null, action);
}
```

所以，如果你在回调中不想返回任何的结果，只运行特定的逻辑，那么你可以考虑使用`thenAccept`和`thenRun`. 一般来说，这两个方法会在调用链的最后面使用。.

6. thenCompose与 thenCombine

以上几种方法都是各玩各的，但`thenCompose()`与`thenCombine()`就不同了，它们可以实现对**依赖**和**非依赖**两种类型的任务的编排。

编排两个存在依赖关系的任务

在前面的例子中，在接收前面任务的结果时，我们使用的是thenApply(). 也就是说，sayLoveFuture在执行时必须依赖nameFuture的完成，否则执行个锤子。

```
// 创建Future
CompletableFuture <String> nameFuture =
CompletableFuture.supplyAsync(() -> {
    return "妲己";
```

```
});

 // 使用thenApply()接收nameFuture的结果，并执行回调动作
CompletableFuture <String> sayLoveFuture =
nameFuture.thenApply(name -> {
    return "爱你，" + name;
});
```

但其实，除了thenApply()之外，我们还可以使用`thenCompose()`来编排两个存在依赖关系的任务。比如，上面的示例代码可以写成：

```
// 创建Future
CompletableFuture <String> nameFuture =
CompletableFuture.supplyAsync(() -> {
    return "妲己";
});

CompletableFuture<String> sayLoveFuture2 =
nameFuture.thenCompose(name -> {
    return CompletableFuture.supplyAsync(() -> "爱你，" + name);
});
```

可以看到，`thenCompose()`和`thenApply()`的核心不同之处**在于它们的返回值类型**：

- `thenApply()`：返回计算结果的原始类型，比如返回String;
- `thenCompose()`：返回CompletableFuture类型，比如返回CompletableFuture.

组合两个相互独立的任务

考虑一个场景，当我们在执行某个任务时，需要其他任务就绪才可以，应该怎么做？这样的场景并不少见，我们可以使用前面学过的并发工具类实现，也可以使用`thenCombine()`实现。

举个例子，当我们计算某个英雄（比如妲己）的胜率时，我们需要获取她参与的**总场次（rounds）**，以及她**获胜的场次（winRounds）**，然后再通过`winRounds / rounds`来计算。对于这个计算，我们可以这么做：

```
CompletableFuture < Integer > roundsFuture =
```

```
CompletableFuture.supplyAsync(() -> 500);
CompletableFuture < Integer > winRoundsFuture =
CompletableFuture.supplyAsync(() -> 365);

CompletableFuture < Object > winRateFuture = roundsFuture
    .thenCombine(winRoundsFuture, (rounds, winRounds) -> {
        if (rounds == 0) {
            return 0.0;
        }
        DecimalFormat df = new DecimalFormat("0.00");
        return df.format((float) winRounds / rounds);
    });
System.out.println(winRateFuture.get());
```

`thenCombine()`将另外两个任务的结果同时作为参数，参与到自己的计算逻辑中。在另外两个参数未就绪时，它将会处于等待状态。

7. allOf与anyOf

`allOf()`与`anyOf()`也是一对孪生兄弟，当我们需要对**多个Future**的运行进行组织时，就可以考虑使用它们：

- `allOf()`：给定一组任务，等待**所有任务**执行结束；
- `anyOf()`：给定一组任务，等待**其中任一**任务执行结束。

`allOf()`与`anyOf()`的方法签名如下：

```
static CompletableFuture<Void>   allOf(CompletableFuture<?>...
cfs)
static CompletableFuture<Object> anyOf(CompletableFuture<?>...
cfs)
```

需要注意的是，`anyOf()`将返回完任务的执行结果，但是`allOf()`不会返回任何结果，它的返回值是**Void**.

`allOf()`与`anyOf()`的示例代码如下所示。我们创建了roundsFuture和winRoundsFuture，并通过`sleep`模拟它们的执行时间。在执行时，winRoundsFuture将会先返回结果，所以当我们调用 CompletableFuture.anyOf时

也会发现输出的是**365**.

```
 CompletableFuture < Integer > roundsFuture =
CompletableFuture.supplyAsync(() -> {
   try {
     Thread.sleep(200);
     return 500;
   } catch (InterruptedException e) {
     return null;
   }
 });
 CompletableFuture < Integer > winRoundsFuture =
CompletableFuture.supplyAsync(() -> {
   try {
     Thread.sleep(100);
     return 365;
   } catch (InterruptedException e) {
     return null;
   }
 });

 CompletableFuture < Object > completedFuture =
CompletableFuture.anyOf(winRoundsFuture, roundsFuture);
 System.out.println(completedFuture.get()); // 返回365

 CompletableFuture < Void > completedFutures =
CompletableFuture.allOf(winRoundsFuture, roundsFuture);
```

在CompletableFuture之前，如果要实现所有任务结束后执行特定的动作，我们可以考虑CountDownLatch等工具类。现在，则多了一选项，我们也可以考虑使用`CompletableFuture.allOf`.

四、CompletableFuture中的异常处理

对于任何框架来说，对异常的处理都是必不可少的，CompletableFuture当然也不会例外。前面，我们已经了解了CompletableFuture的核心方法。现在，我们再来看如何处理计算过程中的异常。

考虑下面的情况，当`rounds=0`时，将抛出运行时异常。此时，我们应该如何处理?

```
CompletableFuture < ? extends Serializable > winRateFuture =
roundsFuture
    .thenCombine(winRoundsFuture, (rounds, winRounds) -> {
        if (rounds == 0) {
            throw new RuntimeException("总场次错误");
        }
        DecimalFormat df = new DecimalFormat("0.00");
        return df.format((float) winRounds / rounds);
    });
System.out.println(winRateFuture.get());
```

在CompletableFuture链式调用中，**如果某个任务发生了异常，那么后续的任务将都不会再执行**。对于异常，我们有两种处理方式：`exceptionally()`和`handle()`.

1. 使用exceptionally()回调处理异常

在链式调用的尾部使用`exceptionally()`，捕获异常并返回错误情况下的默认值。需要注意的是，`exceptionally()`**仅在发生异常时才会调用**。

```
CompletableFuture < ? extends Serializable > winRateFuture =
roundsFuture
    .thenCombine(winRoundsFuture, (rounds, winRounds) -> {
        if (rounds == 0) {
            throw new RuntimeException("总场次错误");
        }
        DecimalFormat df = new DecimalFormat("0.00");
        return df.format((float) winRounds / rounds);
    }).exceptionally(ex -> {
        System.out.println("出错: " + ex.getMessage());
        return "";
    });
System.out.println(winRateFuture.get());
```

2. 使用handle()处理异常

除了`exceptionally()`，CompletableFuture也提供了`handle()`来处理异常。不过，与`exceptionally()`不同的是，当我们在调用链中使用了`handle()`，**那么**

无论是否发生异常，都会调用它。所以，在`handle()`方法的内部，我们需要通过`if (ex != null)`来判断是否发生了异常。

```
CompletableFuture < ? extends Serializable > winRateFuture =
roundsFuture
    .thenCombine(winRoundsFuture, (rounds, winRounds) -> {
        if (rounds == 0) {
            throw new RuntimeException("总场次错误");
        }
        DecimalFormat df = new DecimalFormat("0.00");
        return df.format((float) winRounds / rounds);
    }).handle((res, ex) -> {
        if (ex != null) {
            System.out.println("出错: " + ex.getMessage());
            return "";
        }
        return res;
    });
System.out.println(winRateFuture.get());
```

当然，如果我们允许某个任务发生异常而不中断整个调用链路，那么可以在其内部通过`try-catch`消化掉。

五、CompletableFuture中的线程池

在前面我们已经说过CompletableFuture中的任务有**同步**、**异步**和**指定线程池**三个变种。比如，当我们调用`thenAccept()`时，将不会使用新的线程，而是使用当前线程。而当我们使用`thenAcceptAsync()`时，则会创建新的线程。**那么，在前面的所有示例中，我们都从未创建过线程，CompletableFuture又是如何创建新线程的**?

答案是**ForkJoinPool.commonPool()**，我们熟悉的老朋友又回来了，还是它。当需要新的线程时，CompletableFuture会从commonPool中获取线程，相关源码如下:

```
public static CompletableFuture<Void> runAsync(Runnable runnable)
{
    return asyncRunStage(asyncPool, runnable);
```

```
}
private static final Executor asyncPool = useCommonPool ?
ForkJoinPool.commonPool() : new ThreadPerTaskExecutor();
```

可问题是，我们已经知道了commonPool的潜在风险，**在生产环境中使用无异于给自己挖坑**。那怎么办呢？当然是**自定义线程池**，如此重要的东西务必要掌握在自己的手上。**换句话说，当我决定使用CompletableFuture的时候，默认就是我们要创建自己的线程池**。不要偷懒，更不要存在侥幸心理。

CompletableFuture中每个核心类型的方法都提供了自定义线程池的重载，使用起来也较为简单：

```
// supplyAsync中可以指定线程池的方法
public static < U > CompletableFuture < U > supplyAsync(Supplier
< U > supplier,
    Executor executor) {
    return asyncSupplyStage(screenExecutor(executor), supplier);
}

// 自定义线程池示例
Executor executor = Executors.newFixedThreadPool(10);

CompletableFuture < Integer > roundsFuture =
CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(200);
        return 500;
    } catch (InterruptedException e) {
        return null;
    }
}, executor);
```

小结

至此，关于CompletableFuture的讲解已经全部结束。我们已经知道，CompletableFuture是Future的扩展和增强，并提供了大量强大且好玩的优秀特性。这些特性可以帮助我们优雅地解决一些场景问题，而在此之前我们要实现相同的方

案可能要花费很大的代价。

当然，CompletableFuture这朵玫瑰虽然很漂亮，但它的刺也同样尖锐，它并不是天生完美。因此，在使用CompletableFuture时仍要遵循一些必要的约束：

- **自定义线程池**：当你决定在生产环境使用CompletableFuture的时候，你应该同时准备好对应的线程池策略，而不是偷懒地使用默认的线程池；
- **团队共识**：技术就是这样，好与不好总是会有不同的标准。当你说好的时候，你的队友可能并不这么认为，反之你也可能也会反对某种技术观点。因此，当你决定采用CompletableFuture的时候，最好和团队同步你的策略，让大家都了解它的优点和潜在的风险，各行其是绝对不是好的策略。

最后，CompletableFuture的源码有近**2400**行，并且有大量的API. 说实话，在王者系列所分析的源码文章中，CompletableFuture的源码是截止目前最难以理解的。如果将源码展开讲解的话，大概需要数万字，这将直接劝退百分之九十以上的读者。所以，我们也不建议你硬啃所有的源码，而是**建议在归纳分类的基础上，有针对性地掌握它的重点部分**。当然，如果你饶有兴趣地读完了它所有的源码，在此给你点赞。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 动手：编写代码体验`runAsync()`的用法，并指定线程池。

延伸阅读与参考资料

- https://thepracticaldeveloper.com/differences-between-completablefuture-future-and-streams/#conclusions-pros-and-cons-of-completablefuture
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

星耀01：群雄逐鹿-从鹿死谁手深入理解Java内存模型

欢迎来到《王者并发课》，本文是该系列文章中的**第27篇**，星耀中的**第1篇**。

在前面青铜、黄金、铂金和砖石系列文章中，我们已经介绍了JAVA并发编程中的常见问题和基本的解决思路。然而，这些文章更多的是侧重于如何理解并使用JAVA既有的工具，并没有谈及工具的原理。所以，在星耀系列中，我们的主题将会侧重于JAVA并发原理的介绍，比如JAVA并发模型、AQS的设计原理等。

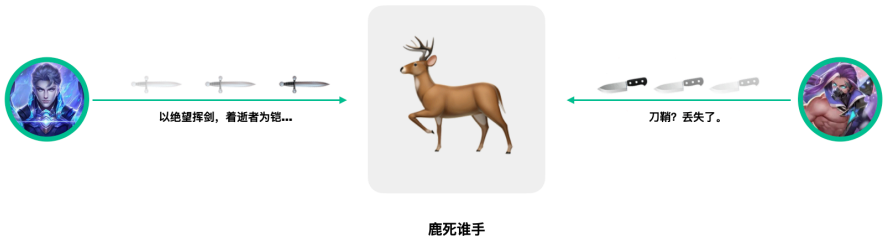
在本篇文章中，我们将首先介绍JAVA内存模型。提到JAVA内存模型，我们可能会首先想到那幅经典的内存模型图，不怎么好看但似乎还挺复杂，让人有种莫名的望而却步的感觉，如果再扯上本文要讲的**指令重排**、**Happens-before**、**volatile**和**synchronized**，似乎难上加难。但其实，我们可以换个思路来理解JAVA内存模型，或许能轻松点，不妨试着往下看。

一、群雄逐鹿，鹿死谁手?

这一切，还得从王者峡谷中跌宕起伏的纷争说起。

月黑风高夜，铠爷拖着利剑在峡谷游荡，山谷里的生活毕竟枯燥且乏味，夜晚打野算是消遣。溪水潺潺处，一只小鹿在悠闲地喝水。要说这小鹿也是倒霉，它哪里知道喝口水竟然遇见了打野， 它背后的男人正挥舞着利剑向它走来，已注定在劫难逃。然而，就在铠即将斩获这只小鹿时，好巧不巧，他的老对手兰陵王从草丛里蹦了出来，真是冤家路窄，哪哪都有这人。于是，这俩打野痴汉开启了对小鹿的你争我夺。

那么，问题来了。按照峡谷的规矩，这鹿最终定然不可能同属于二人，谁补了最后一刀，这鹿就算是谁的。可是，如何界定是谁补的最后一刀?



鹿死谁手

这个问题的背后，就是个典型的并发读写问题。接下来，我们写段程序来感受下这个血腥的场面，并试着找出解决的办法。

我们定义一个`DeerGame`类来模拟这次的竞争。在这个场景中，有一只待宰的小鹿，它有**100**个单位的血量，每次被攻击时都掉部分的血量，血量为**0**时，小鹿将丢掉性命，攻击者获胜。

```
/**
 * 群雄逐鹿
 */
public class DeerGame {
    /**
     * 待宰的小鹿
     */
    private final Deer deer = new Deer();

    /**
     * 物理攻击，一次攻击掉血10个单位
     */
    public boolean physicalAttack() {
        return deer.reduceBlood(10) == 0;
    }

    /**
     * 魔法攻击，一次攻击掉血5个单位
     */
    public boolean magicAttack() {
        return deer.reduceBlood(5) == 0;
    }

    @Data
    private static class Deer {
        private int blood = 100;
```

```
        public int reduceBlood(int bloodToReduce) {
            int remainBlood = blood - bloodToReduce;
            blood = Math.max(remainBlood, 0);
            return blood;
        }
    }
}
```

接着，我们创建两个线程来模型**兰陵王**和**铠**对小鹿的争夺，他们将各自对小鹿进行多达**100次**的物理攻击（有点毫无人性）。谁是最后一击，谁就是获胜的一方。

```
public static void main(String[] args) {
    DeerGame deerGame = new DeerGame();
    Thread 兰陵王 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            if (deerGame.physicalAttack()) {
                System.out.println("兰陵王胜出！");
            }
        }
    });

    Thread 铠 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            if (deerGame.physicalAttack()) {
                System.out.println("铠胜出！");
            }
        }
    });
    兰陵王.start();
    铠.start();
}
```

看到这里，如果你阅读过前面的系列文章，或者有一定的并发编程基础，那么你一定会发现上述代码片段存在严重的缺陷：**它没有并发控制**。换句话说，**两个线程都在读写Deer中的`blood`字段，但这个字段却没有任何的并发处理**，结果就是程序故障，两人可能都获胜。也许你会说，解决这个问题很简单，我们可以在`reduceBlood`方法前面加上`synchronized`关键字来实现多线程同步。

你说的很对，但我们要讨论的问题正在于此。为什么要加上`synchronized`关键字？**加与不加，内存层面发生了什么事？** 虽然解决的办法很简单，但要理解这个办法的内涵却并不容易，这个过程将会涉及到JAVA内存的并发模型设计、硬件架构设

计、CPU指令重排等多个问题，请稍作镇定继续往下看，我们一起来啃下这块硬骨头。

二、内存世界里的软硬件架构差异

（一）从软件层面理解JAVA内存模型设计

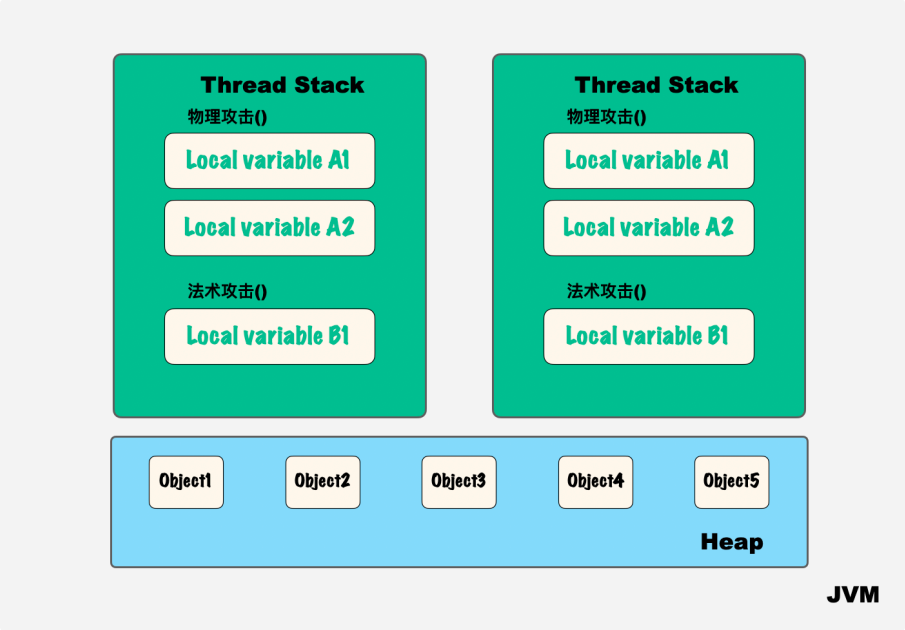
在上述的代码片段中，我们定义了`Deer`对象，在Deer中有个`blood`字段。此外，还有个`reduceBlood`方法，注意在这个方法中的`remainBlood`变量。

```
public class DeerGame {
    private final Deer deer = new Deer();
    @Data
    private static class Deer {
        private int blood = 100;

        public int reduceBlood(int bloodToReduce) {
            int remainBlood = blood - bloodToReduce;
            blood = Math.max(remainBlood, 0);
            return blood;
        }
    }
}
```

在这小小的代码片段中，已经包含了线程的**堆栈**、**本地变量**和**共享对象**等内容。所以，我们现在有必要回顾下JAVA的内存模型设计。当然，我们在此不会展开描述，如果你需要了解更多，可以参考这篇文章。**在此，我们要理解的是，从JAVA模型角度来说，对象与线程的数据放哪里了。**

在JAVA虚拟机中，内存主要分为**堆（Heap）与栈（Stack）两个部分。其中，堆存储的是共享数据，而栈存储的则是线程的专属数据，每个线程都有自己的线程栈（thread stack）**。线程栈包含了当前线程所访问的方法以及当前的访问点，所以线程栈也可以称之为**调用栈（call stack）**。此外，线程栈还包含了所执行方法的本地变量，比如上述示例代码中的`remainBlood`，线程中的本地变量存储在线程栈中，所以它们仅对当前线程可见，其他线程想看是不可能的，若想访问那更是痴

心妄想。常见的 原始类型数据都会存储在线程栈中，比如`boolean`、`byte`、`short`、`char`、`int`、`long`、`float`和`double`等。原始变量在传递时，传递是的是拷贝的副本，并不会传递本身。

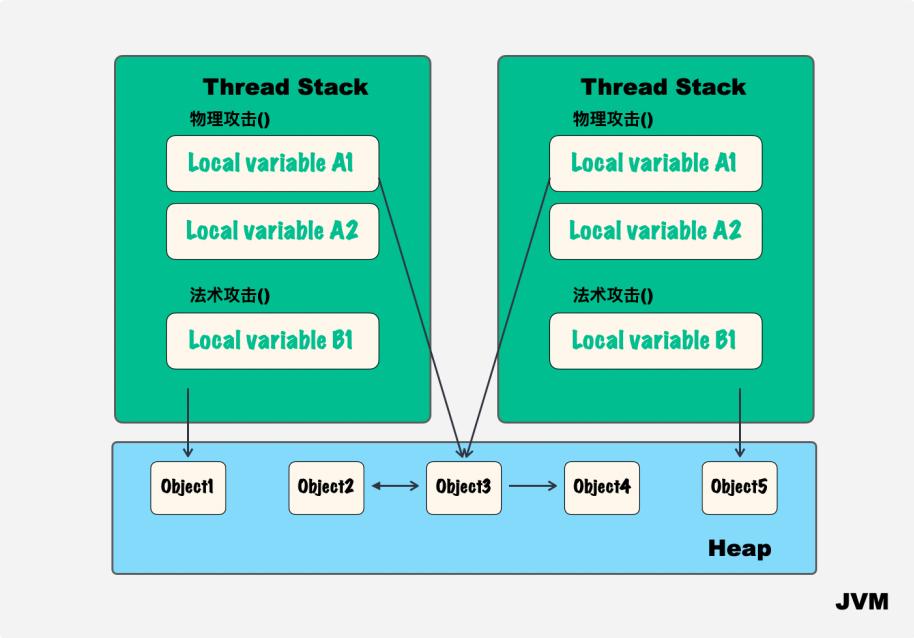


到这里，我们理解每个线程都有自己的小房间来存储自己的数据，在传递原始类型时，传递的也只是副本，看起来安全且祥和。**但是，假如线程间传递的是引用类型呢**？比如，我们传递的不是`int`和`boolean`，而是`Integer`和`Boolean.`..情况会怎么样？又比如，我们把上述示例代码调整，两个线程中分别有自己的本地变量`deerGame1`和`deerGame2`，但是这两个变量的引用都是`deerGame`，所以它们两个注定要纠缠不清。

```
private static final DeerGame deerGame = new DeerGame();

public static void main(String[] args) {
    Thread 兰陵王 = new Thread(() -> {
        DeerGame deerGame1 = deerGame;
        for (int i = 0; i < 100; i++) {
            if (deerGame1.physicalAttack()) {
                System.out.println("兰陵王胜出！");
            }
        }
    });
    Thread 铠 = new Thread(() -> {
```

```
        DeerGame deerGame2 = deerGame;
        for (int i = 0; i < 100; i++) {
            if (deerGame2.physicalAttack()) {
                System.out.println("铠胜出！");
            }
        }
    });
    兰陵王.start();
    铠.start();
}
```

从内存模型理解的话，就是本地变量`deerGame1`和`deerGame2`都引用了`deerGame`，虽然本地变量存储在线程栈中，但是`deerGame`却位于堆上，属于共享数据。**对于共享数据，无论是谁修改了它，如果引用它的线程数据未能及时感知，那么注定是一场悲剧。**

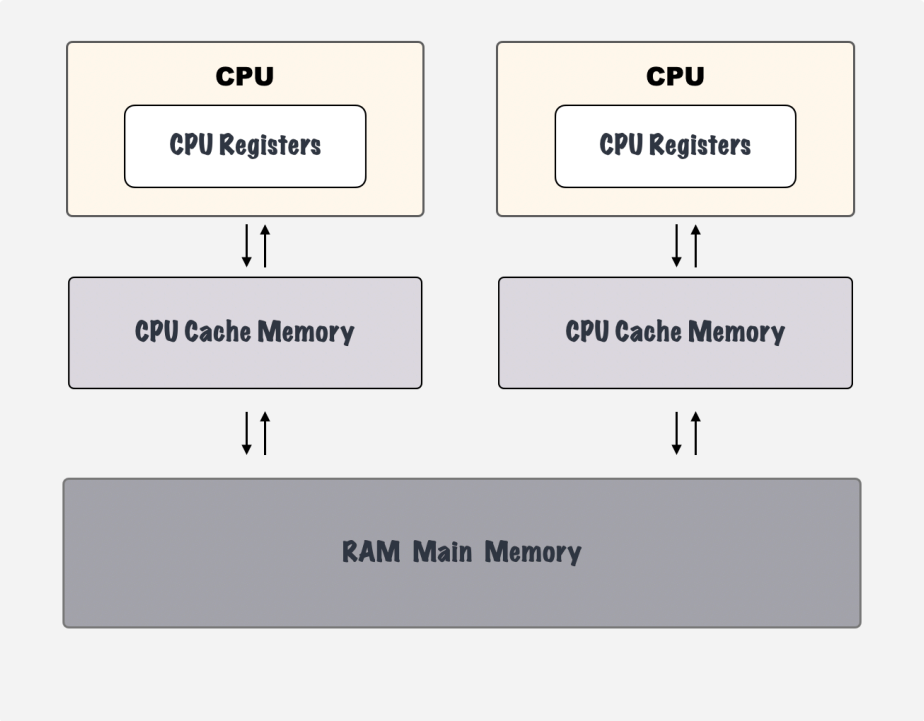


那么，为什么共享的数据在变化时不能被所有的线程很好的感知呢？ 这是个好问题。接下来，我们就要从硬件架构方面来理解这个问题为什么会存在。

（二）从硬件层面理解内存架构

从硬件层面来说，现代计算机的内存架构可以用下面这幅图来简要说明，包含了**主内存**、**CPU高速缓存**和**CPU寄存器**等。从数据读取的速度上来说，很明显离CPU越

近其速度越快，所以最快的是CPU寄存器，其次是CPU高速缓存，最后是主存。



凡是涉及缓存的地方，就必然涉及数据一致性的问题。所以，理解硬件层面的架构最重要的是要能理解多级缓存的存在。正是因为多级缓存的存在，CPU的运算才变得复杂和多变。

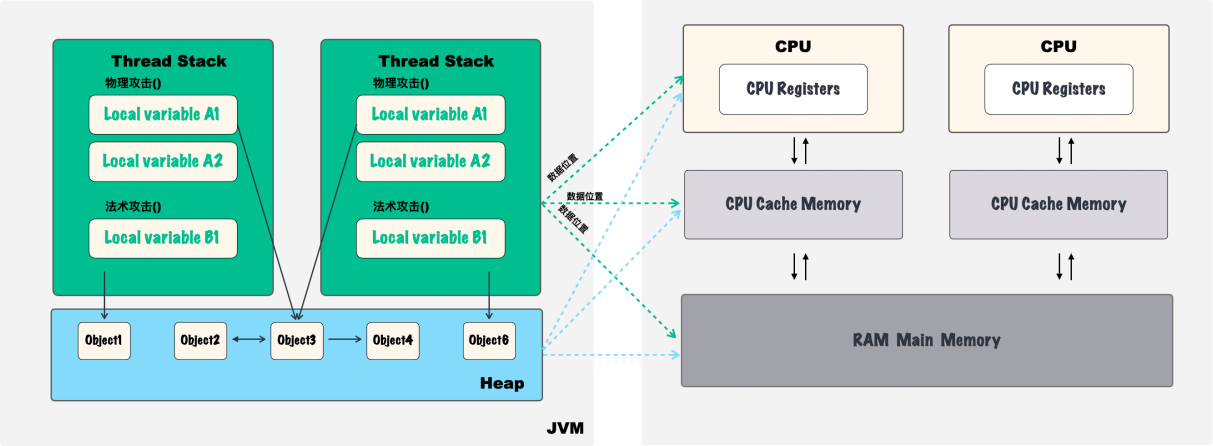
所有的CPU在运算时，都能访问寄存器、高速缓存和主存。**通常，当 CPU 需要访问主存时，它会将部分主存读入其 CPU 缓存。它甚至可以将部分缓存读入其内部寄存器，然后对其执行操作。当 CPU 需要将结果写回主存时，它会将值从其内部寄存器刷新到高速缓存，并在某个时候将值刷新回主存**。而当 CPU 需要在高速缓存中存储其他内容时，通常会将存储在高速缓存中的值刷新回主内存。

看到这里，可能你已经产生了疑问：**不同的线程运行在不同的CPU中，每个CPU又有自己的缓存，那线程之间如何共享数据？** 好问题，这正是我们接下来要讨论的。

（三）理解软硬件的架构差异

从上述软硬件两部分的内容，我们可以看到，JVM的内存模型设计与硬件的缓存体系设计是不同的。在JVM中，我们将存储分为堆和栈两个部分，然而在硬件内存的

设计却没有这种划分，这就导致了下图所示的问题。JVM中大部分的堆栈数据都存储在主存，但是有部分线程的数据存储在寄存器或高速缓存中。**如此，麻烦就来了：不同线程对共享变量的读与写怎么搞**？我更新的变量别人怎么看得见？别人把对象更新了我还怎么用？所以，这就产生了**共享对象可见性**和**竞态条件**两个关键问题。

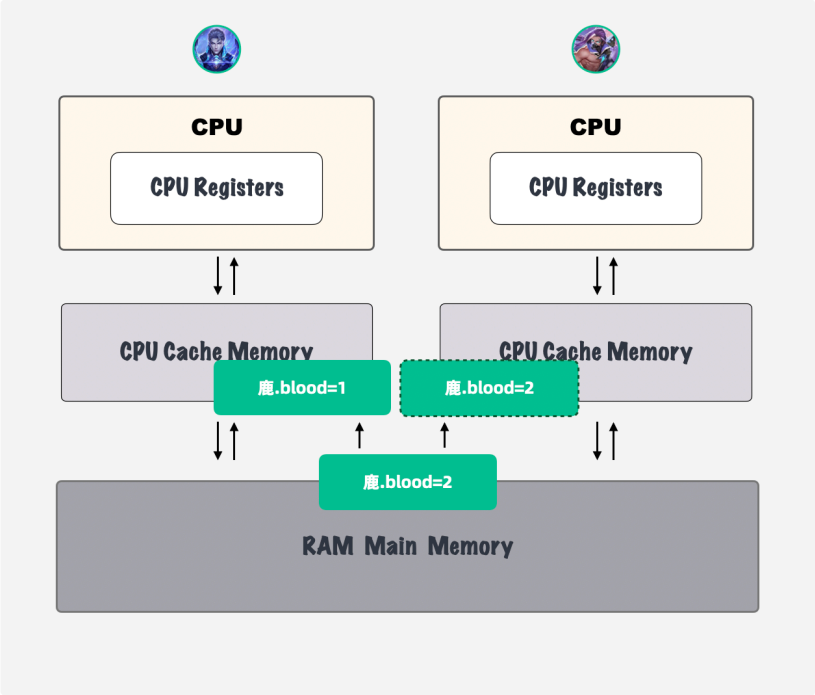


1. 共享对象的可见性

前面我们已经讲过不同线程都有自己的空间来存储自己的数据，不同的CPU也有自己的寄存器和高速缓存来存储数据。对于单个线程来说，线程更新了主存中的变量，在没有其他线程更改的情况下，它总是能读取到之前所更新的值，不会有任何问题。但是，对于多线程来说，这就会产生所谓的 **共享对象的可见性（Visibility of Shared Objects）** 问题。

比如，在上述示例代码中，铠从主存中读取的`blood`的值为2，随后将值改成了1，在他还未将1刷回主存的时候，兰陵王也从存主存中读取到了2. 那么，这个时候数据的不一致性就产生了，兰陵王无法看到铠对数据的修改，很显然他所拿到的**数据已经失效**。

当然，解决可见性的简单而有效的办法就是使用`volatile`关键字。`volatile`**可以确保不同线程在读取变量时，总是从主存读取，在更新缓存时也更新到主存**。关于`volatile`的更多内容，我们稍后再讲。

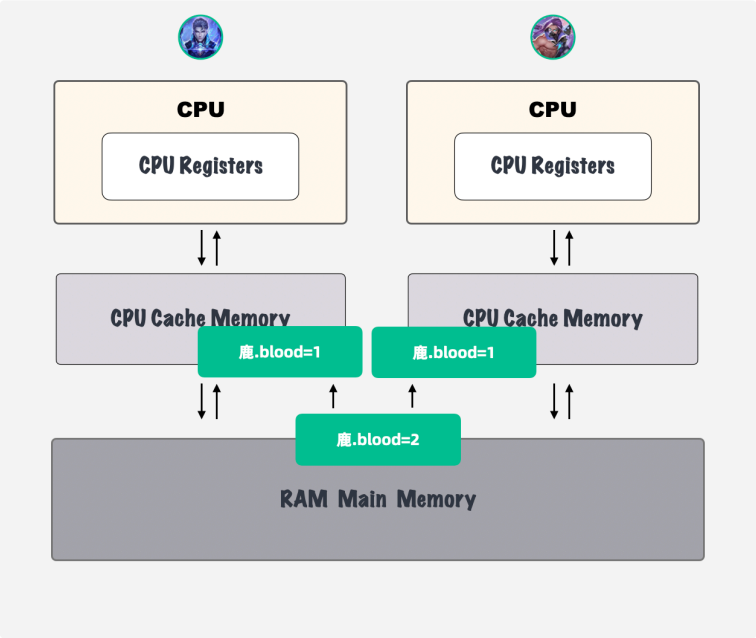


2. 竞态条件

现在，我们已经理解所谓共享对象的可见性是指线程修改数据后，其他线程能否感知看见的问题。那么，如果修改共享对象的不是某个线程，而是多个线程同时修改会怎么样？这就产生了另外一个问题，即**竞态条件（Race Conditions）**。

如下图所示，铠和兰陵王同时从主存中获取了`blood=2`，随后他们在各自的缓存中将`blood`的值更改为1，并相继将更新后的值刷回主存。**但是，此时无论谁先执行刷回主存的动作，主存最后的结果都是1**。很显然，我们期望的结果应该是0，数据已经出错了，这就是竞态条件下的并发问题。

其实，要解决竞态条件的问题并不难，我们自然能想到使用同步，比如`synchronized`可以让同一时刻仅有一个线程可以访问临界区域。同时，线程在同步块内读取变量时，都会从主存中读取，而在线程退出同步块时，则会将变量刷回主存。



你看，可见性和竞态条件这两个问题都不是凭空而来，而是由问题催生了这两个问题。所以，下面的内容就是围绕着两个问题来寻求答案。

三、调和软硬件架构差异

在上文我们讲述了JAVA内存模型的设计和硬件内存的架构，以及它们之间的差异所导致的问题。那如何调和它们之间的差异？这就需要分别从CPU和JAVA模型设计上找到一些答案，也就是我们要谈论的**指令重排**、**Happens-before**、**volatile**和**synchronized**等概念。

但是，为什么是这几个概念而是不是其他?

说到这里，我们要理解的是JAVA内存模型中最重要的就是对**各种操作**的处理。要编排好这些操作，就需要按照**Happens-before**偏序关系对它们进行排序，而这种操作是基于内存操作和同步操作等级别来定义的。如果缺少充足的同步，当不同的线程访问共享数据时，就会发生很多奇怪的问题，所以就需要**volatile**和**synchronized**等措施来规避这些问题。

你看，原本零散的概念是不是就衔接起来了？接下来，我们试着给它们多点描述。

(一)理解Happens-before关系

1. 理解CPU下的指令重排

首先不得不提的就是**指令重排（Instruction Reordering）**，你可能在其他地方也看见过这个概念。**为什么要先说指令重排？因为顺序决定着CPU的运算结果**。无论是共享对象的可见性问题，还是竞态条件问题，**本质上都是顺序的问题**。并且，正是因为顺序的问题，才有了后面的Happens-before关系问题。

从设计初衷上讲，**指令重排是为了提高CPU的处理性能，让CPU并发运算地更快**。比如，10个人去野外打野烧烤，如果这10人总是一起打野、一起捡柴火、一起烤制，那就不如分组后同时分头行动更有效率，打野的回来时，柴火也捡好了。开干，很麻溜。CPU的指令重排要做的就是按照合理的工序，给这10个人分组，然后分头行动。**当然，分工的时候，要考虑到上下游的依赖问题。如果调度器采用不恰当的方式来交替执行不同线程的操作，将会导致不正确的结果**。

看下面这两个简单的语句，语句1和语句2之间没有任何的依赖的依赖关系。如此，CPU在运算时可以并发地执行这两个语句，这样要比逐条执行快很多。

```
statement1: a = a1 + a2
statement2: b = b1 + b2
```

但是，如果我们把上面两个语句调整为下面这样，两个语句不再是独立的语句，语句2依赖于语句1先执行，此时如果并发执行会发生什么？很显然，如果不加以控制，这两条语句的执行顺序可能很随机，每次执行的结果都不一样。于是，问题就来了。

```
statement1: a = a1 + a2
statement2: b = a + b1
```

CPU在执行程序指令时，为了提高处理的效率，并不会傻傻地按顺序执行。CPU会对需要执行的指令进行分析、调整执行的顺序，使得指令集既能并发执行也不会影响最终的结果。

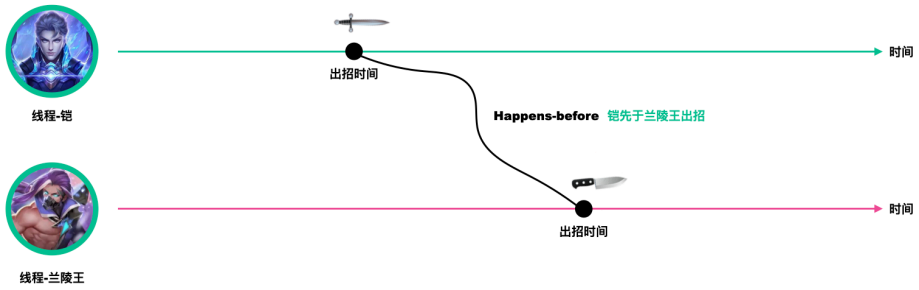
2. 指令重排下的Happens-before关系约定

从静态的视角看，JAVA内存模型可以分为堆栈两大块。然而，如果从动态的视角看，JAVA内存模型其实是通过**各种操作**来定义的，比如对变量的读/写操作、对监视器的加锁/释放操作，以及线程的启动/协调等操作。**为了正确编排、执行这些操作，就需要在它们之间建立一种偏序关系，也就是Happens-before**。

抽象地讲，所谓Happens-before指的是两个**事件的结果**之间的关系。**如果一个事件应该在另外一个事件之前发生，则结果必须反映这一点**。举个例子，我们排序接种新冠疫苗，接种点制定的规则是谁先到谁先接种。那么，如果我比你先到，我自然就一定要在你前面接种。

当铠和兰陵王同时猎杀小鹿时，如果是铠先动的手，那么在程序执行的结果上一定是铠获胜，这和兰陵王后面蹦出来跳得多高没有关系。**所以，这是约定的基本原则，也就是Happens-before关系。如果线程之间建立了这样的关系，那么不同线程的操作应当被其他线程所看见，否则就会出现内存一致性错误（Memory Consistency Errors）**。

至于如何建立这样的关系，那就是我们程序设计上要考虑的事情了。



（二）如何建立Happens-before关系

通常，我们有几种方式来建立Happens-before关系，比如使用单线程、Thread中的start/join等，**但是最重要的两种方式还是`synchronized`和`volatile`**，有两句英文对此有很好的概括：

- “A volatile write will happen before another volatile read。”
- “An unlock on the synchronized block will happen before another lock。”

1. 使用synchronized同步块

当不同的线程访问临界区时，使用`synchronized`关键字是个简单有效的方案。在同一时刻，只有获得锁的线程才能执行临界区的代码块，而未获得锁的线程自然只能在后面等待。**所以，获得锁的线程和其他未获得锁的线程之间，就具有Happens-before关系**。

2. 使用volatile关键字

为了解决指令重新排序的问题，Java提供了`volatile`关键字，它不仅可以保证共享对象的可见性，还提供了Happens-before关系保证：

- **变量写入的可见性保证**：当写入`volatile`变量时，该值保证会直接写入主内存 (RAM)。此外，**写入`volatile`变量的线程可见的所有变量也将同步到主存储器**；
- **变量读取的可见性保证**：当读取`volatile`的值时，可以保证直接从内存中读取该值。此外，**读取 volatile 变量的线程可见的所有变量也将从主存储器中刷新它们的值。**

volatile关键字的局限性

虽然`volatile`关键字保证对`volatile`变量的所有读取都直接从主存读取，并且对`volatile`变量的所有写入都直接写入主存，但应对并发场景时，单单依靠`volatile`仍然是不够的。比如，前面所说的多个线程同时更新某个变量时，就会发生竞态条件，彼此会覆盖他人已更新的值。

什么情况下使用volatile是可靠的

如果两个线程都在读取和写入共享变量，那么使用`volatile`关键字是不够的。在这种情况下，我们需要使用**同步**来保证变量的读取和写入是原子的，所以`volatile`往往和`synchronized`或其他JUC中的并发工具搭配使用。

虽然多线程读写的场景下使用`volatile`并不可靠，但是如果只有一个线程读写，而其他线程只读的话，那么将会保证其他线程所读取的都是最新值。如果我们不使用`volatile`的话，则无法做到这点。

volatile的性能考量

由于`volatile`变量的读写都是发生在主存，那么很显然CPU对此类变量的读取会影响到性能。毕竟，`volatile`变量无法再享受CPU寄存器和高速缓存的性能优势。所以，我们在使用`volatile`时需要考虑到这点。

小结

至此，关于JAVA内存并发模型的介绍到此结束。在本文中，我们通过铠和兰陵王争夺小鹿的例子来引出内存中的并发问题。接着，我们通过对硬件架构的描述来进一步说明内存模型中的数据一致性问题的产生及原因。最后，根据问题我们去寻找答案，并陆续提到了指令重排、Happens-before、volatile和synchronized等概念及作用。

本文的内存较为枯燥不易理解，涉及到了软硬件的架构和很多的概念。在学习时，建议不要把它们看作是独立的知识点，这样理解起来会比较生硬，而是要把它们和要解决的问题联系起来，完成知识的串联，形成整体的结构化认知。**换句话说，我们要理解它是什么，更要理解为什么是它。**

比如，我们通过小鹿的例子感知到了内存模型中的问题，从问题我们又追溯到硬件架构的设计，至此应理解软硬件架构的差异是问题的所在。那如何调和差异呢？于是我们又从硬件架构的角度去理解指令重排和Happens-before的概念，为了约束指令重排和构建Happens-before，volatile和synchronized闪亮登场。这样，我们就可以把本文的很多概念串联起来。

正文到此结束，恭喜你又上了一颗星✨

夫子的试炼

- 动手：编写多线程代码体验`volatile`变量和非`volatile`变量读写的不同效果。

延伸阅读与参考资料

- 《王者并发课》大纲与更新进度总览：https://juejin.cn/post/6967277362455150628
- https://www.baeldung.com/java-volatile
- https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/happens-before.html
- http://tutorials.jenkov.com/java-concurrency/java-memory-model.html
- https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- Github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

星耀02：穷理尽妙-解构同步器设计原理与AQS浅析

欢迎来到《王者并发课》，本文是该系列文章中的**第28篇**，星耀中的**第2篇**。

说起JUC中大名鼎鼎的**AQS（AbstractQueuedSynchronizer）**，相信大家对其并不陌生，说是如雷贯耳也不为过。然而，当我们需要走近它的时候，大部分人或望而却步，或绕道而行。原因在于，虽然我们有了解它的欲望，但想把它搞清楚又似乎总是不容易。**正所谓撼山易，撼AQS难**。

作为星耀系列的文章，解决的正是此类难解之题。所以，我们将在这篇文章中邀你一起，用庖丁解牛的方式来逐步剖析AQS的表象与内核，理解AQS的原理及应用。在本文中，我们不会开篇就直捣黄龙、解读源码，而是**从最基本的同步说起，并由同步循序渐进地讲解同步器、CLH队列和AQS的基本构成，以及AQS在ReentrantLock和CountDownLatch中的应用，最终实现解构AQS的目的**。

一、何为同步

在讨论所谓的**同步**概念之前，我们再来回顾经典的医院就诊的场景。

> 话说冤家路窄的开和赵子龙在林间相遇，话不投机三句开打。数百个来也不见胜负，但两人都已经披红挂彩，只好休战同去医院看大夫。

众所周知，医院向来是医生少患者多，经常人满为患，就算武功高强也得讲究先来后到。于是，赵子龙和铠先在挂号处取了号，一看就诊室有人就诊，外面还有长队，只好面面相觑加入到等候的队伍中。

说到这，相信你对医院这个机制并不陌生，是不是很简单？但我们要说的是，这个场景体现的就是经典的 **同步机制** ，理解这个机制你对 AQS就已经理解了一半。不信？往下看。

在并发编程中，说起“**同步**”，有些人可能总觉得这个词有些别扭。没错，所谓同步是`synchronize`的翻译。但是，对于这个英文单词，我们的理解往往又不是很准确，于是听起来甚至有些别扭。因此，我们首先要能正确理解这个词的含义，才能进一步明白**同步器**是个啥玩意。

在中文语境中，我们说同步，往往指的是**两个或两个以上随时间变化的量在变化过程中保持一定的相对关系**。比如，把本地的数据同步到云端，或者是从手机同步到电脑。看到没有，中文语境中的**同步**往往指的是同一事物**在不同位置的相对变化，比如手机中的照片同步到云端，照片是不是还是那个照片？注意，它说的是同一事物。

但是，在英文中`synchronize`的含义则完全不同于我们日常所理解的**同步**，即使我们把它翻译为**“同步”**。请看英文中对`synchronize`的解释：

synchronize

verb (UK usually **synchronise**)

UK /ˈsɪŋ.krə.naɪz/ US /ˈsɪŋ.krə.naɪz/

[I or T]

to (cause to) happen at the same time:

• *The show was designed so that the lights synchronized **with** the music.*

看到英文版释义中的`at the same time`，相信你对`synchronize`这个词的准确含义已经豁然开朗，并了然于心。**原来，所谓同步就是搞定并发的意思，是两个事件同时发生的关系**。而在计算机科学中，同步器则是用于处理同步关系的结构和算法。

基于对同步的理解，可以发现前述的医生就诊问题也是一种同步机制的设计问题，即如何处理**多名患者**同时看**一个医生**，这里就需要考虑两个关键点：医生的**就诊状态**和**患者的队列**。下面我们通过ReentrantLock来模拟这个就诊问题：

- 如果患者通过`qualificationLock.lock()`获得了就诊资格（比如此刻没有就诊），那么他可以直接就诊；
- 如果患者未能通过`qualificationLock.lock()`获得就诊资格，则患者需要进入`qualificationLock`中的排队机制进行等待。

你看，基于AQS实现的ReentrantLock用起来很简单，其解决问题的思路也很朴素，无非是我们日常也能遇到的问题，所以我们先从朴素的问题上认识AQS。

```
/**
 * 门诊就诊（就诊+排队）同步示例
 */
public class OutpatientDemo {
    /**
     * 当前就诊资格
     */
    private final ReentrantLock qualificationLock = new
ReentrantLock();

    public void 就诊() {
        qualificationLock.lock();  // 如果门诊没有人，则获取当前就诊资
格，直接就诊。否则，将进入大厅队列排队等候。
        try {
            // ... 就诊中
        } finally {
            qualificationLock.unlock(); // 就诊结束离开后，释放资格。
        }
    }
}
```

二、同步器的设计思路

对于普遍存在的同步问题，我们可以根据具体问题设计不同的同步机制，也就是所谓的**同步器**。比如，医生通过取号机、叫号器、显示屏和等候区来解决多人排队问题，去银行办理业务也是类似的机制。当然，这些都是我们生活中的同步机制。而

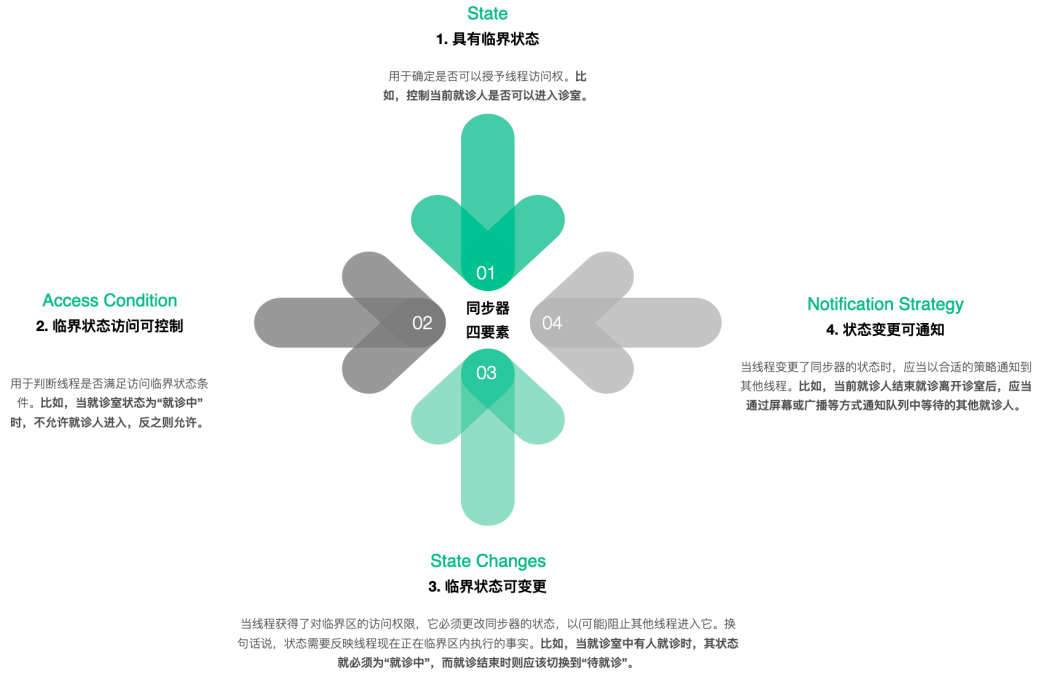
在Java中，我们则可以通过AQS和`synchronize`关键字来实现同步。所以，理解同步器的设计思路是后续理解AQS的关键，而当我们从现实世界的角度看程序世界时，问题则相对更容易理解。

（一）同步器的基本组成

无论是现实世界还是程序世界，在同步器设计的设计上总体是相似的。比如，都有**状态**和**队列**，以及围绕这两个点再增加一些其他辅助机制。所以，从抽象的角度看，一个同步器应该具备以下四要素：

- **临界状态（State）**：用于确定是否可以授予访问对象（比如线程、患者等）的访问权；
- **临界状态访问控制（Access Condition）**：用于判断访问对象是否满足访问临界状态的条件。比如，当就诊室状态为“就诊中”时，则不允许就诊人进入，反之则允许；
- **临界状态可变更（State Changes）**：当线程获得了对临界区的访问权限，它必须更改同步器的状态，以(可能)阻止其他线程进入它。换句话说，状态需要反映线程现在正在临界区内执行的事实。比如，当就诊室中有人就诊时，其状态就必须为“就诊中”，而就诊结束时则应该切换到“待就诊”；
- **状态变更可通知（Notification Strategy）**：当线程变更了同步器的状态时，应当以合适的策略通知到其他线程。比如，当前就诊人结束就诊离开诊室后，应当通过屏幕或广播等方式通知队列中等待的其他就诊人。

当然，这四要素只是在设计同步器时的基本思路和原则，具体同步器的设计并不局限于以上的四要素，可能没有其中的部分组成，也可能会有其他的组成等。



（二）队列的选型

现在，我们已经理解设计同步器的要素，以及其必要的组成，比如状态和队列。对于状态，相对比较简单也好理解，比如AQS中的状态只是一个int类型的字段。而对于队列的理解，则是关键所在，如何设计有效的排队机制，关乎到数据的安全性、同步的效率以及扩展性等多方面的考量。所以，这部分我们要讲的就是如何设计队列，包含无队列和CLH队列等。

1. 无队列

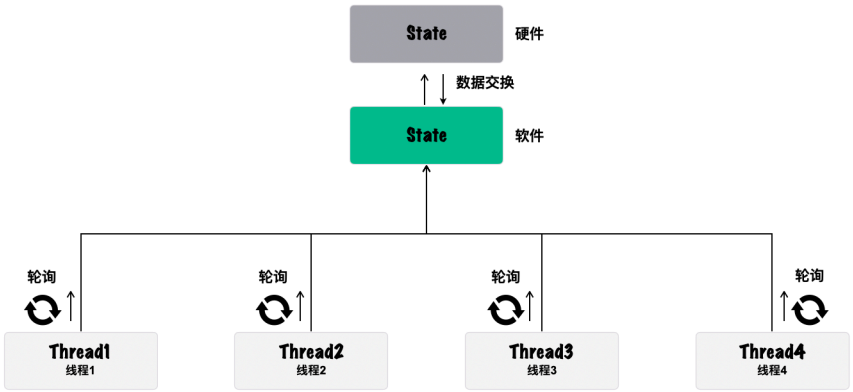
所谓无队列设计，即下面的左图所示，不同的线程为了争取到自己需要的资源会一拥而上，毫无秩序可言。在无队列设计中，力气大、脸皮厚的线程可能会优先获得资源，而素质高的线程则可能一直在礼让，导致始终无法获得资源，也就是我们常说的“**饥饿**”，因为它不公平。同时，无队列也会导致大量的线程阻塞，对于系统来说易引发灾难。

无队列下的线程竞争情况可以用下面这幅图来说明，每个线程通过**自旋锁**来请求资源。所谓自旋，并不是蒙上眼睛原地转圈的意思，而是它会主动地、时不时地去询问资源是否就绪，就像我们时不时跑到门口询问“**到我了吗**”。

自旋锁的优点在于实现简单，也不需要复杂的调度和队列管理。但是，它的缺点则在于：

- **锁饥饿**：在锁的竞争中，有的线程可能会始终被插队，导致饥饿；
- **性能堪忧**：多个线程同时轮询状态时，一个是消耗线程所在的CPU资源，一个是导致多个CPU的高速缓存频繁同步，影响CPU效率。

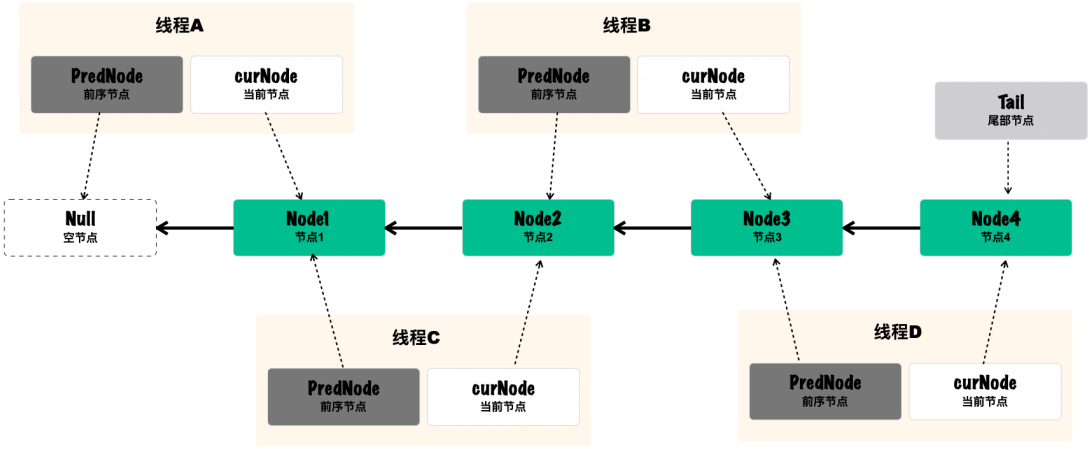
所以，**自旋锁适用于竞争不激烈、锁持有时间短的场景**，像AQS这种为各场景提供基础同步能力的同步器，自然不适合采用这种方式。



既然无队列会存在无序竞争的情况，那如何解决这个问题？如果是我们来设计，如何设计这个队列，如何保证线程的公平性并兼顾性能？这就要说说CHL队列了。

2. 理解CLH队列锁

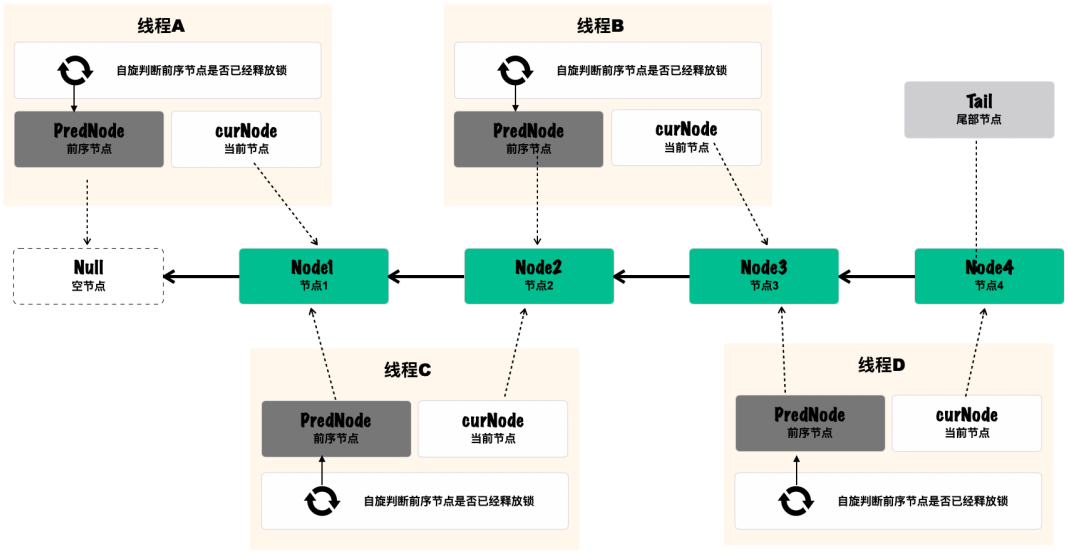
CLH队列锁是由Craig, Landin, and Hagersten三人共同设计的一种队列锁机制，我们先来看看CLH队列的基本模样，如下图所示。



从图中可以看到，在CLH队列中每个线程都是一个节点（Node），在Node中同时还有节点的前驱指针。从结构上看，CLH队列和普通队列似乎没有区别。**但是，这个队列需要解决一个核心问题：如何解决锁的竞争和释放中的性能问题。**所有节点都要自主去竞争？显然不可能。这好比我们在候诊区等待时，时不时都去问问医生”到我了吗“，如果这么做我们没累瘫，医生已经被烦死。

解决这个问题有两个思路：**一个是广播通知下一位患者，比如医院的大屏和喇叭通知；另一个则是我们可以看排在我们前面的人，比如排在我们前面的人已经进去就诊了，那下一位必然是轮到我，不然还有谁？**这两种思路，医院信息化系统通常采用的是第一种，而计算机中的CLH队列锁采用的则是第二种：**既然大家都轮询同一把锁效率低，那我们为何不能只轮询各自的前序节点状态呢**？这就是CLH队列锁的精髓所在。

现在，我们结合下面这幅图用计算机的语言来描述它的原理和精髓：**所谓CLH锁是通过移动尾部节点实现的FIFO队列，每个节点包含了线程、前序节点信息以及各自的状态。CLH队列中的各节点不会轮询同一个共享变量，而是仅轮询各自的本地变量，从而解决效率的问题。此外，FIFO属性也保障了CLH队列的公平性。**



当然，要实现CLH队列锁的高效和公平性，我们在构建队列时，就需要考虑如何设计队列、如何入队、如何出队和如何唤醒节点等一系列问题，而这些问题正是AQS中的一些关键问题，我们将在后文中逐渐展开叙述。

> **补充信息**：现代计算机通常是多CPU架构，各自CPU有着自己的高速缓存。当不同的线程位于不同的CPU时，线程间交换数据时就需要特别考虑性能问题。CLH队列锁针对某些架构是高效的，但换了其他架构则未必，因此我们要了解这部分知识和它的局限性。这里没有说具体的哪种架构，是为避免在本文将问题扩大化，引入太多额外知识会增加不必要的负担，有兴趣的同学可以自行检索这方面的知识。
>
> - CLH队列锁扩展阅读：**https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/CLH.pdf**

2.3 AQS中的CLH队列锁

现在，我们知道CLH队列锁有着效率高、公平和实现简单等优点，那它有没有缺点呢？当然有。CLH队列的主要缺点在于：**一是节点的自旋影响CPU的效率；二是它的实现过于简单，不能满足AQS中复杂的场景需要**，比如AQS中节点的阻塞和唤醒等。**因此，AQS采用了CLH锁，但是对它进行了一些改造和扩展，主要是通过节点状态`waitStatus`来丰富队列的操作性。**

所以，关键的问题就来了：**如何设计重新CLH队列、如何解决入队出队等一系列问**

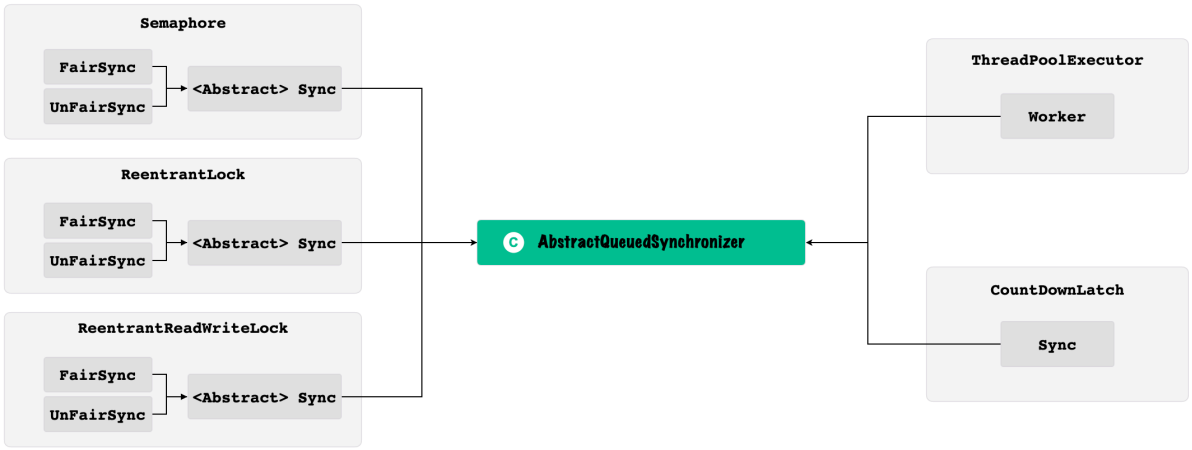
题?

三、初始AQS

（一）基本用途

AQS的全称是**AbstractQueuedSynchronizer**，即抽象队列同步器。这个名字有三个单词，其中**Abstract**表示它是一个**抽象类**，AQS源码中定义了一些具体的方法，也定义了一些抽象的方法。**换句话说，我们不能单独地直接AQS，而是需要继承它并实现部分能力**。

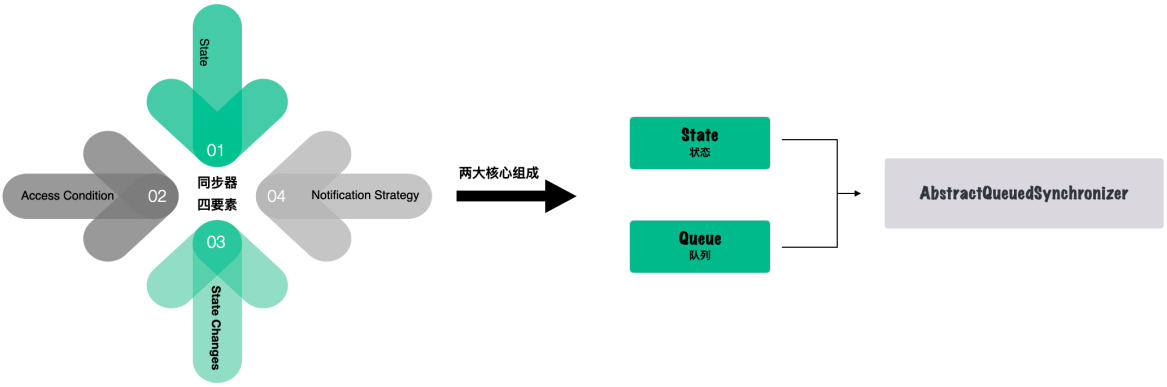
在JAVA的JUC中，AQS的使用非常广泛，在我们前面的系列文章都有提到，比如**Semaphore**、**ReentrantLock**、**ReentrantReadWriteLock**、**ThreadPoolExecutor**和**CountDownLatch**等，相关类对AQS的使用如下图所示。不夸张地说，在我们需要使用同步的时候，其背后几乎都有AQS的影子，只不过我们在前面都没有展开说，后文我们会选择其中两个来展开详述。



（二）总体结构

在前面的系列文章中，我们已经了解设计同步器时的四大核心要素。在AQS中，也仍然遵循这几要素。因此虽然AQS看齐来很复杂，源码洋洋洒洒几千行，但是如果我们分析它的数据结构和方法会发现，其核心就是**State和Queue(Node)**这两个，其他都是围绕它们俩展开。所以，我们理解AQS务必要抓住其核心所在，理解

其内核和外围，不要眉毛胡子一把抓，更不要慌。



接着上面的分析，我们来对AQS的源码和设计进行归类和总结。为了更好的结构化理解AQS，我们可以将其拆解为下面的5层。很显然，AQS中的方法众多，因此我们将相关的方法进行归类，这样在理解时会有侧重点。其中，**绿色部分**为需要重点关注的方法，**波浪线打底**的方法是需要子类实现的方法，而**实线打底**的方法则是AQS中提供的重要方法。

1. 核心部件之状态位

作为AQS的核心之一，同步状态字段`state`的重要性不言而喻。注意，`state`字段有两点需要注意：

- `voliatile`修饰：不同线程在读写该字段时，字段的最新值对各线程可见；

- `int`类型：采用int类型是比较巧妙的设计，用于表示当前同步的状态情况。在AQS中，同步有独占和共享两种模式。其中，在独占模式下，`state`的值为`0`和`1`即可；但是，在共享模式下，`state`的值则会大于1，比如某个具有超能力加成的大夫同时可以看10个病人，那么`state`的值就为`10`。

另外，还需要注意的是，`state`**表示的是同步的状态，而不是线程的状态，线程的状态在队列的节点中，对此不要搞混淆**。

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

2. 核心部件之同步队列

AQS中有三个核心的字段，除了上述的`state`之外，还有两个分别是**Node**类型的`head`和`tail`. 注意，AQS并没有所谓的`queue`字段，而是用`head`和`tail`来表示队列，有头有尾，不就是队列么...那么，`head`和`tail`所构成的队列是怎样的？我们通过源码和图示来说明。

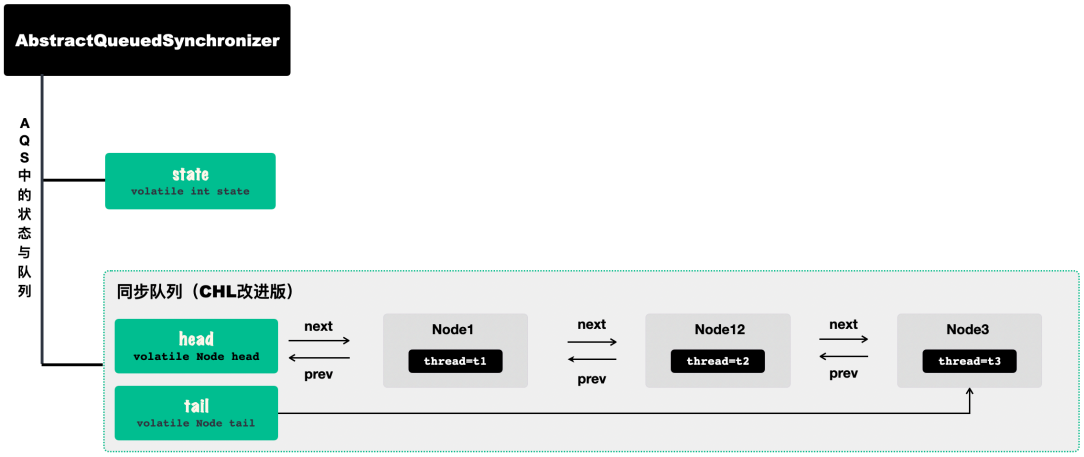
在AQS的源码中，`head`和`tail`均由`transient volatile`来修饰，这点和`state`是一致的，表示对其他线程可见。另外还需要注意的是：

- **延迟初始化（lazily initialized）**：`head`和`tail`均是在需要时才会初始化，而不是在AQS实例化即初始化。在后面我们会提到，并不是在所有情况下都需要队列，在某些情况下AQS是不需要创建的，所以它们都是延迟初始化；
- **由方法提供赋值入口**：任何线程均不可以直接修改`head`和`tail`的值，而必须通过AQS提供的方法入口来完成；

```
/** 队列头部
 * 1. 延迟初始化；
 * 2. 经由方法提供赋值入口；
 * 3. 头部节点状态不可以为Canceled.
 */
private transient volatile Node head;

/** 队列尾部
```

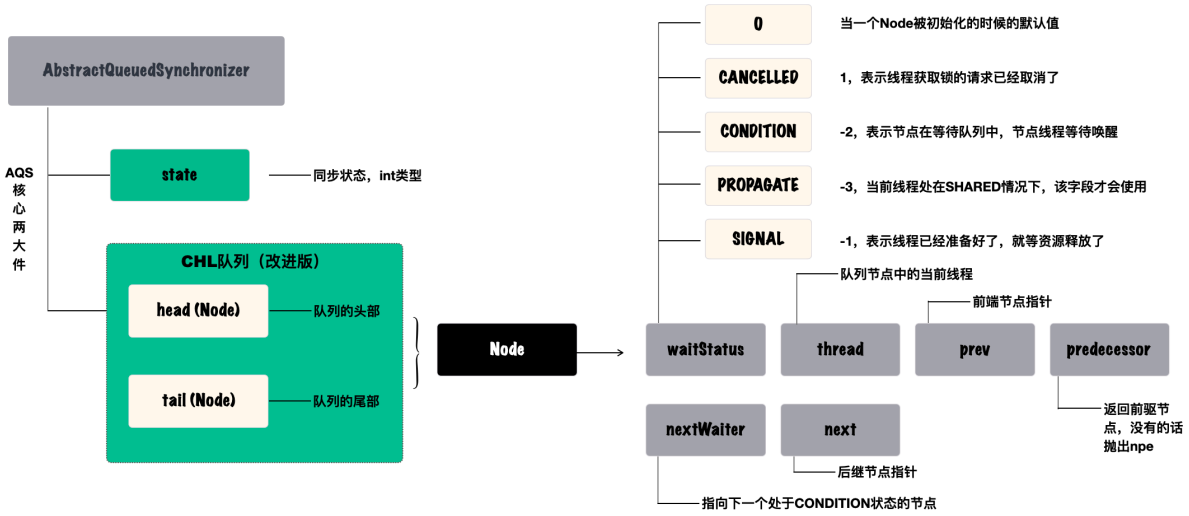
```
     * 1. 延迟初始化;
     * 2. 经由方法提供赋值入口;
     */
    private transient volatile Node tail;
```

AQS中由`head`和`tail`构成的队列图示如下所示。图中可以看到，`state`、`head`和`tail`构成了AQS中三个关键字段。其中，`head`和`tail`又是CHL队列的关键字段，队列中的节点通过前驱和后继的方式完成节点间关系的连接。



3. 核心节点结构

作为AQS核心的数据结构之一，Node的组成如下图所示，包含了`waitStatus`、`thread`、`prev`、`next`和`nextWaiter`几个关键属性。其中，`waitStatus`表示当前节点线程的状态，`thread`表示当前线程，而`prev`和`next`分别代表节点的前驱和后继。



Node中的`waitStatus`是个重要且容易误解的属性，它有5个枚举值，为了方便理解，我们把可以这5个值以`0`为界分为三个区间来理解：

区间	状态	说明
`waitStatus`=0	初始化状态	通过`new Node()`创建节点时，此时状态为0.
`waitStatus`>0	取消状态	线程状态无效，该线程被中断或者等待超时，需要移除该线程节点。
`waitStatus`<0	有效状态	包括-1、-2、-3等值。其中，-1表示该线程处于同步队列且可以被唤醒的状态，-2表示节点在条件队列中，-3用于共享模式，表示可以传播到下个节点。

另外，关于`nextWaiter`属性，这里要先补充说明的是AQS中其实有两种类型的队列：同步队列和条件队列。我们目前主要讨论的都是同步队列，条件队列会在本文末尾讨论，而`nextWaiter`主要用于条件队列。在前面图示的同步队列中，队列是双向队列，由`prev`和`next`表示当前节点的前驱和后继，而条件队列则是单向队列，通过`nextWaiter`指向下一个节点，限于篇幅更多细节不在此处描述。

Node核心源码如下所示：

```
static final class Node {
    static final Node SHARED = new Node();
    static final Node EXCLUSIVE = null;

    /** 线程已经取消 */
    static final int CANCELLED =  1;
    /** 线程需要唤醒 */
    static final int SIGNAL    = -1;
    /** 线程条件等待中*/
    static final int CONDITION = -2;
    /**
     * 用于共享模式，表示可以传播到下个节点
     */
    static final int PROPAGATE = -3;

    /** 节点状态字段 */
    volatile int waitStatus;

    /** 当前节点的前驱节点 */
    volatile Node prev;

    /** 当前节点的后继节点 */
    volatile Node next;

    /** 节点中的线程 */
    volatile Thread thread;

    /**
      用于条件队列，指向下一个等待节点
     */
    Node nextWaiter;

    Node() {
    }
}
```

四、理解AQS的独占模式-以ReentrantLock为例

前面，我们讲述了AQS的基本组成和和核心数据结构。在这部分内容中，我们以

ReentrantLock为例来分析AQS的内部机制，通过具体的示例有助于我们理解抽象的AQS。

（一）基本用法

下面是前文所述的就诊示例代码。在这段代码中，只有当患者获得锁之后才能进入诊室就诊，否则需要排队等待，而就诊结束后则需要将锁释放，让其他等待的患者进入。示例代码只有几行，可以说是相当精简，非常容易理解。但是，其关键就在于`lock()`和`unlock()`两个方法中，**这两个方法则是由ReentrantLock和AQS协作完成**。所以，接下来我们在分析源码时，将会分别分析这两个部分。

```
/**
 * 门诊就诊（就诊+排队）同步示例
 */
public class OutpatientDemo {
    /**
     * 当前就诊资格
     */
    private final static ReentrantLock qualificationLock = new
ReentrantLock();

    public void 就诊() {
        qualificationLock.lock();  // 如果门诊没有人，则获取当前就诊资
格，直接就诊。否则，将进入大厅队列排队等候。
        try {
            // ... 就诊中
        } finally {
            qualificationLock.unlock(); // 就诊结束离开后，释放资格。
        }
    }
}
```

（二）加锁解析

1. ReentrantLock部分源码解析

下面是精简后的ReentrantLock的`lock()`源码，我们移除了和`lock`无关的注释和

其他源码，以减少对理解的干扰。在源码中，我们可以看到一些关键信息：

- ReentrantLock实现了`Lock`接口；
- ReentrantLock中的核心属性是`sync`，**而Sync类是其内部类，并继承了AQS**；

此外，我们还注意到ReentrantLock提供了**公平**和**非公平**两种模式，**默认是非公平模式**，所以前述示例代码使用的是非公平模式。那如何区别所谓的公平和非公平？从源码中，我们可以清晰地看到在非公平模式下，线程在请求锁时并不是立即排队，而是通过`compareAndSetState`尝试加锁，如果失败再去排队。这就像有人总爱找关系直接去找医生，但是被拒绝之后又乖乖去排队，**没有直接排队就是不公平**。

注意，在这个过程中，线程尝试加锁是ReentrantLock中的源码，而失败后通过`acquire`排队时则将进入AQS的源码。

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

    abstract static class Sync extends AbstractQueuedSynchronizer
{
        abstract void lock();
    }

    static final class NonfairSync extends Sync {
        final void lock() {
            if (compareAndSetState(0, 1)) // 自己先处理
                setExclusiveOwnerThread(Thread.currentThread());
            else
                acquire(1); // 不行再调用AQS方法
        }
    }

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

    public void lock() {
        sync.lock();
```

```
    }
}
```

以上就是ReentrantLock加锁时的内部关键源码，比较简单，主要是融合**Lock接口**和**AQS同步器**。接下来，我们再顺着`acquire`方法进入AQS的源码中一探究竟。

2. AQS部分源码解析

`acquire`是AQS资源抢占的重要方法入口，但它源码也很简短，只有三行。然而，通过这三行代码，我们可以看出其中的重要过程：

- 排队前不死心，通过`tryAcquire`尝试再次抢占锁。虽然成功的概率有限，**但是万一成功就没队列和排队什么事了，岂不痛快？！**
- `tryAcquire`抢占失败后，乖乖去排队；
- 排队时，先将当前线程通过`addWaiter`方法封装成一个节点，再通过`acquireQueued`方法将自己加入到队列中；
- 如果抢占失败，排队也失败，**则彻底死心，就地中断自己**。

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

以上这几行便是`acquire`的核心要义，理解要义之后我们再来分析`addWaiter`等其中的细节。**这里再次提醒，通过`acquire`抢占锁时并不是总要排队的，只有当抢占失败后才会排队。换句话说，并不是AQS中的所有场景都需要使用到队列**。

通过addWaiter源码理解入队逻辑

`addWaiter`的主要职责是根据指定的模式将当前线程封装成节点并**入队**。这里的关键在于：

- **封装节点前要指定模式**：独占或者共享。我们示例中使用的是**独占模式（exclusive）**。

```
private Node addWaiter(Node mode) {
```

```
    Node node = new Node(Thread.currentThread(), mode);
    // 先尝试快速入队，如果失败再通过enq走常规入队模式
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

从寥寥无几的源码中，我们可以看到入队的核心方法是`enq`，但是`addWaiter`在调用`enq`之前，**会先尝试直接加入到队尾的方式直接入队，如果失败再执行`enq`**。那为什么要这么做呢?

原因是为了提高入队效率。因为`enq`入队使用的是`for`循环的方式，所以为了避免进入循环，那自然是能直接入队最好。

比如，当赵子龙准备排队时，赵子龙看到排在队尾的是**孙尚香**，如果子龙走到队伍时队尾仍然是她，则子龙可以快速加入队伍。**但是，假如子龙走到队伍时队尾是妲己，妲己的前面才是孙尚香**。很显然，几步路的功夫队伍已经发生了变化。这个时候，如果子龙强行插队排到孙尚香的后面，可能会享受来自妲己的三连魔法攻击。**那怎么办呢?**

当然是遵守江湖规矩，按规矩排队到队尾，拒绝插队。而这，便是`enq`和`acquireQueued`的核心要义：**坚持按规矩排到队列的末尾，没有队列那就创建队列**。

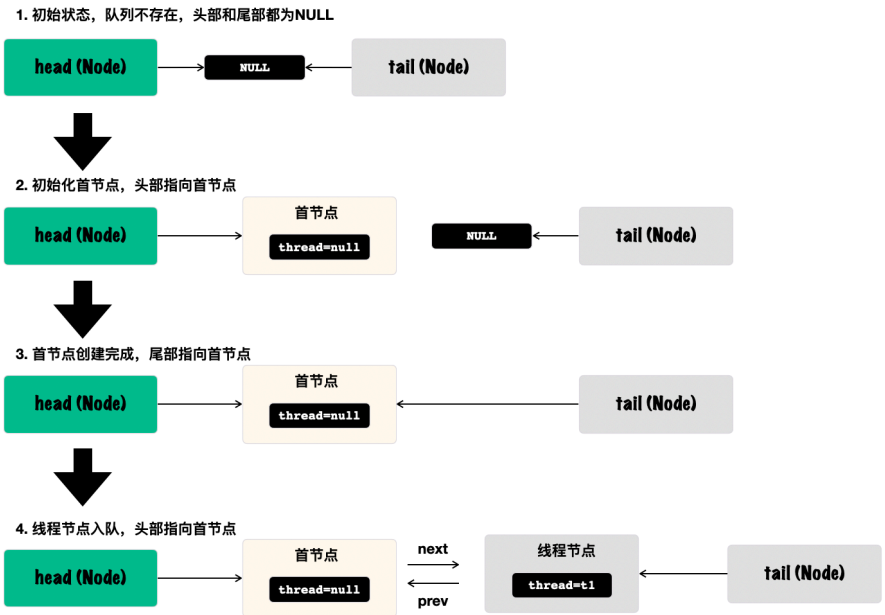
```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // 队列不存在，需要初始化
            // 思考：这里为什么要设置一个空的节点作为头结点？稍后解释
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
```

```
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

3. 队列变化解析

关于`enq`和`acquireQueued`入队时的核心队列变化，为了便于你的理解，我们制作了下面这幅图，图中的四个步骤反映的是队列的变化过程，我们以线程`t1`入队来分析这个过程：

1. 初始状态下，AQS队列中的`head`和`tail`指向的都是NULL;
2. 构建首节点，这个节点是个空节点，队列的头部指向首节点；
3. 队列的尾部指向首节点；
4. 构建线程`t1`节点，并将t1节点的前驱指向首节点，而尾部节点则指向t1节点。



此处需要特别注意的是，在AQS同步队列中有一个所谓的 首节点，在初始化队列时

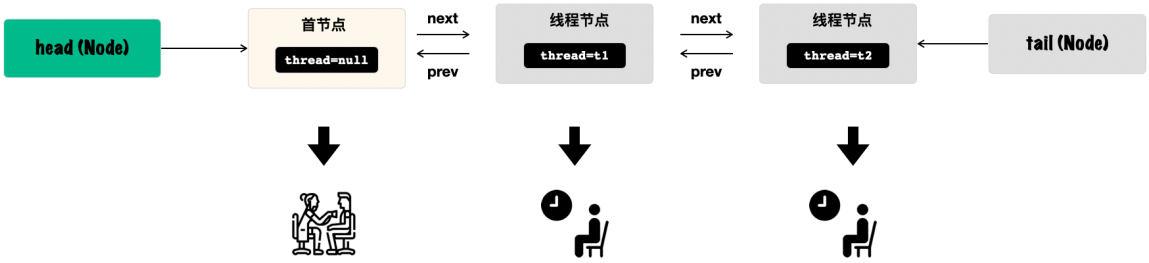
会首先创建这个首节点。这个节点的存在非常有意思，因为它其实是个空节点（thread=null），但又极其容易被误解却又经常被问起。

看到这里，我们不禁要问既然是空节点，那它存在的意义是什么？说起这个问题，我们还是要回到前述的就诊排队问题。

理解AQA同步队列中的首节点【重要】

我们首先来思考一个问题：**在排队就诊的队列中，真的是所有人都在排队吗**？如果是，那医生在干嘛呢？如果不是，那队伍最前面的那个人是什么状态？那究竟是还是不是？**当然不是，进入首节点意味着出队**。

在排队的就诊队列中，最前面的那个人当然不是在排队，**他在医生那里就诊**。对不对？**所以，在AQS中，这个正在就诊的人就是那个空的首节点，是他锁定了资源，他的线程是正在运行中的，而不是等待中**。这就是首节点，它不是排队中的节点，这点非常容易误解。



接下来，我们从`enq`和`acquireQueued`的源码中理解这点。在`enq`源码中，我们可以看到到当队列不存在（`t==null`）时，会执行`compareAndSetHead(new Node())`来设置空的首节点。

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) {
            if (compareAndSetHead(new Node()))// 设置空的首节点
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
```

```
            }
        }
    }
}
```

而在`acquireQueued`入队时，会执行`tryAcquire(arg)`来尝试抢占资源，如果成功则会执行`setHead(node)`将自己设置为首节点。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                // 设置当前节点抢占资源成功，设置为首节点
                setHead(node);
                p.next = null;
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

那么，这个`setHead(node)`做了什么呢？下面的源码已经写得清清楚楚。源码只有三行，但是我们要特别注意的是，**不要认为`setHead(node)`意味着只是排到了队列的头部，它其实意味着出队、出队和出队**！所以，这个方法通常是由`acquire`成功后调用。

而一旦抢占资源成功后，则不需要再排队，所以在当前节点变为首节点后，会通过第二行的`node.thread = null`将其线程置为空。如此，**当该线程执行结束后，当前节点不再有其他引用，可以辅助垃圾回收**。

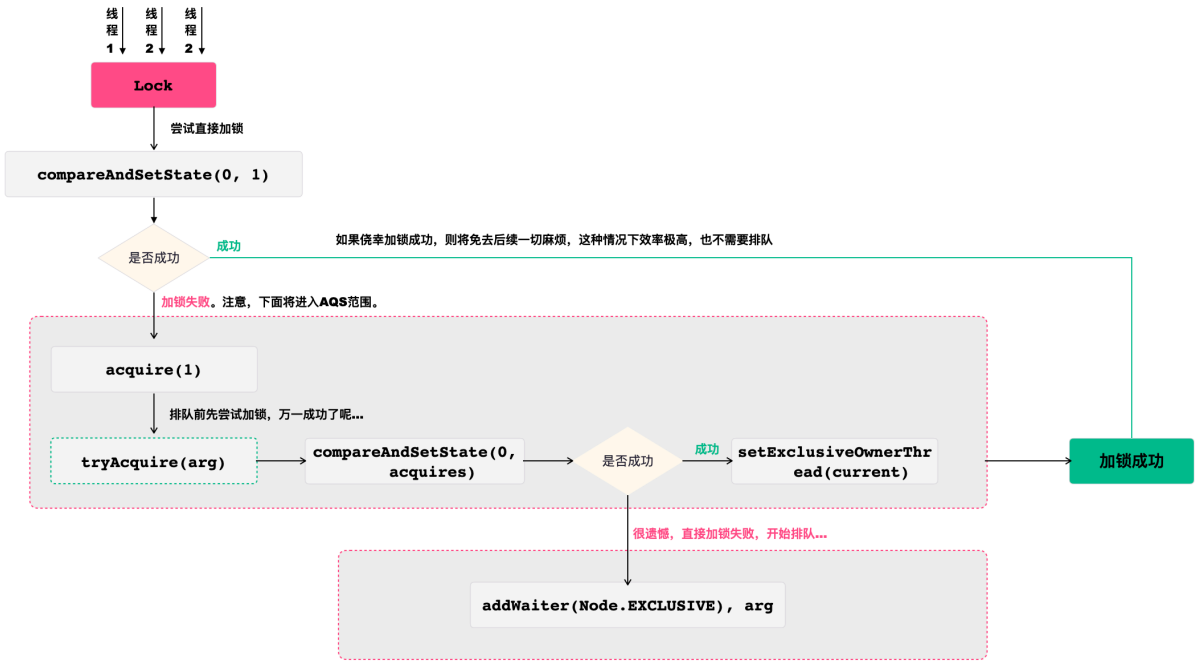
```
private void setHead(Node node) {
```

```
        head = node;
        node.thread = null;
        node.prev = null;
    }
```

4. 总体流程图示

以上就是ReentrantLock中关于加锁的整体过程。在这个过程中，由ReentrantLock和AQS两部分代码共同完成。理解加锁的过程，重点在于理解其中的核心思想和步骤，比如哪些是由ReentrantLock完成的、哪些是AQS完成的、队列是如何设计的、首节点的意义是什么等等。

为了方便你理解加锁的过程，我们制作了下面这幅图，图中展示了一些过程中的一些关键步骤。



（三）锁的释放解析

在上部分内容中，我们通过ReentrantLock分析了AQS的加锁过程，在这部分我们仍然结合两者再来探索AQS的解锁过程，即当我们执行`qualificationLock.unlock()`时发生了什么。

1. ReentrantLock部分源码解析

ReentrantLock中的解锁入口是其`unlock`方法，在这个方法中我们可以看到ReentrantLock本身没有其他的处理逻辑，而是直接调用了AQS的`release`方法。但是，AQS的`release`会调用`tryRelease`，而`tryRelease`的实现则是由ReentrantLock实现，所以ReentrantLock源码中我们要重点关注的只有`tryRelease`这个方法。

`tryRelease`在实现上并不复杂，关键点在于当状态为`0`时则视为释放成功，而且非当前抢占线程不允许释放。

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

    abstract static class Sync extends AbstractQueuedSynchronizer
{
        protected final boolean tryRelease(int releases) {
            int c = getState() - releases;
            if (Thread.currentThread() !=
getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
            if (c == 0) { // 释放成功
                free = true;
                setExclusiveOwnerThread(null);
            }
            setState(c);
            return free;
        }
    }

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

2. AQS部分源码解析

AQS源码部分的重点则在于`release`方法，它主要调用子类的`tryRelease`，当子

类判定释放成功后，则唤醒后继节点线程。

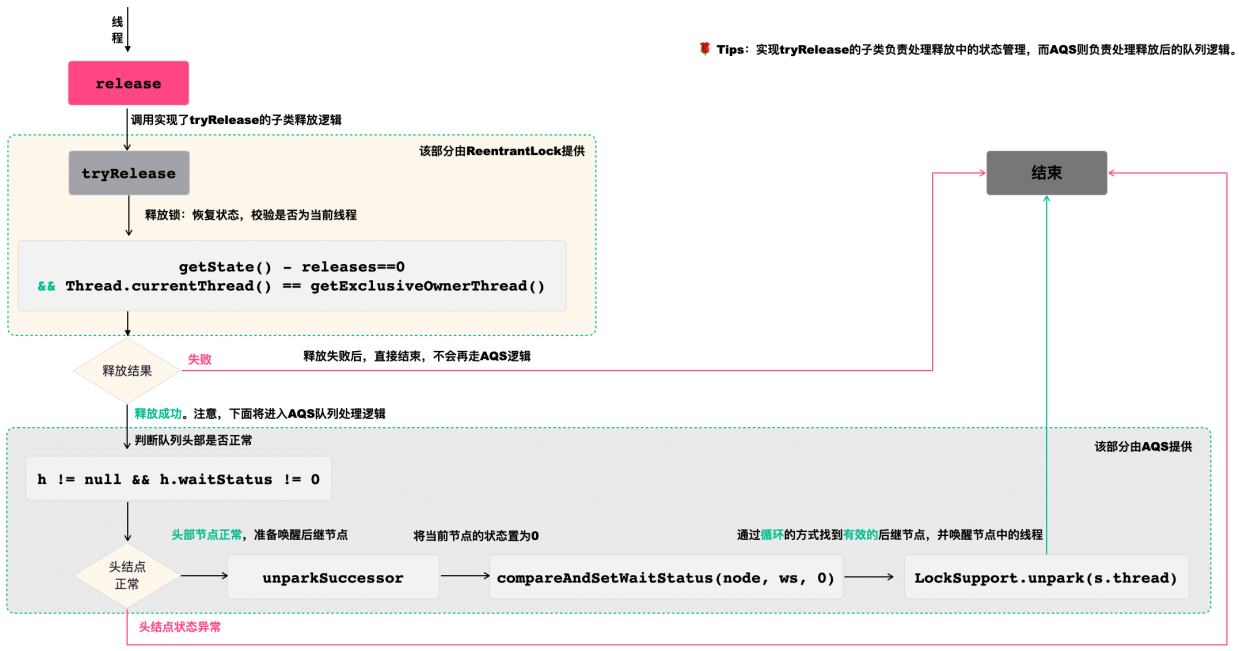
```
public final boolean release(int arg) {
    if (tryRelease(arg)) { // 调用子类实现
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h); // 唤醒后继节点
        return true;
    }
    return false;
}
```

`unparkSuccessor`方法主要用于唤醒后继节点。**需要注意的是，在唤醒过程中可能有的节点已经取消或者为null，那么这个时候会依次向后寻找有效的节点来唤醒。**

```
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    Node s = node.next; // 找到待唤醒的后继节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
// 后继节点可能为null或状态错误，则循环向后查找
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null) // 找到有效的后继节点，唤醒它
        LockSupport.unpark(s.thread);
}
```

3. 总体流程图示

下面是ReentrantLock释放锁时的流程图，相对于加锁而言，锁的释放要简单些。流程中的重点在于要区分哪些是ReentrantLock完成的，哪些是AQS完成的，**以及如何唤醒后继节点**。



五、理解AQS的共享模式-以Semaphore为例

现在，我们已经知道，AQS中的核心属性之一`state`是个`int`类型的变量，并且AQS的同步状态支持**独占模式**和共享模式，而在ReentrantLock中我们已经理解AQS在独占模式下的工作原理，**所以在这部分我们将借助Semaphore来理解AQS中的共享模式**。（如果你对Semaphore不甚了解，可以查阅王者并发课系列的相关专题文章）

（一）基本用法

通常，我们在医院就诊时往往是一个诊室同时仅问诊一个患者，所以我们可以通过ReentrantLock来模拟这一场景。**但是，假如门诊里有两名医生，可以同时问诊两名患者**。那么，这种场景下我们就可以使用Semaphore来模拟，而Semaphore的背后正是AQS的共享模式。相关示例代码如下所示，诊室同时允许两名患者进入。

```
/**
 * 门诊就诊（就诊+排队）同步示例
 */
public class OutpatientDemo {
    /**
```

```
     * 当前就诊资格，允许2人同时就诊
     */
    private final static Semaphore permits = new Semaphore(2);

    public void 就诊() {
        permits.acquire();  // 如果当前有可用就诊资格，则获取当前就诊资格，直接就诊。否则，将进入大厅队列排队等候。
        try {
            // ... 就诊中
        } finally {
            permits.release(); // 就诊结束离开后，释放资格。
        }
    }
}
```

那么，当代码执行`permits.acquire()`时发生了什么？我们接着往下看。

（二）资源抢占解析

Semaphore在执行`permits.acquire()`时分两部分完成，一部分在Semaphore中完成，一部分则由AQS完成。

1. Semaphore部分源码解析

以下是Semaphore中关于`acquire()`的核心源码，为了减少其他代码对你的影响，我们已经删除了不必要的注释和其他代码，仅保留和`acquire()`相关的代码。

在代码中，我们可以看到Semaphore支持公平和非公平两种模式，在ReentrantLock的示例中我们使用的非公平模式，而在此我们将示例Semaphore的公平模式，但其实两者差别并不大。关于下面的源码部分，有几个关键点需要我们注意：

- 默认情况下Semaphore使用的是非公平模式；
- `permits.acquire()`运行时，调用的是AQS中的`sync.acquireSharedInterruptibly(1)`方法，而这个方法又会反过来调用Semaphore中的`tryAcquireShared`方法；
- 源码中`acquire()`、`acquire(1)`、`tryAcquire()`和`acquireUninterruptibly`

()等都只不过是中断、超时等不同的变种，其核心思路不变，不要有“乱花渐欲迷人眼”的错觉。

```
public class Semaphore implements java.io.Serializable {
    private final Sync sync;

    abstract static class Sync extends AbstractQueuedSynchronizer
{
        Sync(int permits) {
            setState(permits);
        }
    }

    static final class FairSync extends Sync {
        FairSync(int permits) {
            super(permits);
        }
        // 公平模式下获取共享资格
        protected int tryAcquireShared(int acquires) {
            for (;;) {
                if (hasQueuedPredecessors())
                    return -1;
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }
    }

    // 构建信号量，并初始化资格总数
    public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }
    // 获取资格，获取失败后等待
    public void acquire() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }
    // 释放资格
    public void release() {
        sync.releaseShared(1);
    }
}
```

2. AQS部分源码解析

AQS在处理共享资源申请时，也有很多不同的变种方法，比如中断、超时等，但其核心思路一致，所以这里使用`acquireSharedInterruptibly`来讲解，这个方法也是Semaphore调用的方法。

`acquireSharedInterruptibly`方法的源码如下所示，主要表达了两层含义：

- 如果当前线程中断，则抛出异常；
- 如果Semaphore中的`tryAcquireShared`返回的结果小于0，则意味着可以继续，将进入AQS队列处理逻辑。

```
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        // 存在可用资源，进入共享获取逻辑
        doAcquireSharedInterruptibly(arg);
}
```

AQS共享模式下的队列处理逻辑和独占模式下的处理逻辑总体相似，其核心差异在于`addWaiter(Node.SHARED)`和`setHeadAndPropagate(node, r)`两行代码，前者标记了当前节点为共享模式，而后者则在将自己设置为头部后，同时唤醒后继节点。那么，唤醒后继节点是什么意思?

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    // 新节点设置为共享模式
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                // 调用子类的方法
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    // 设置头部和传播状态
```

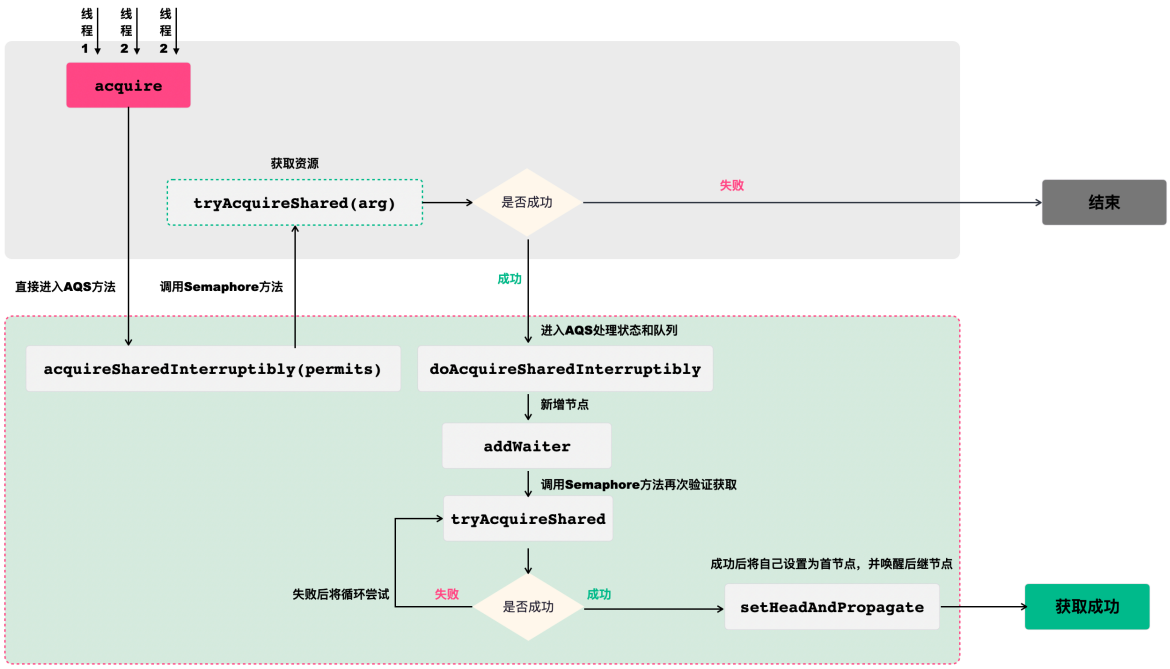
```
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

这是因为，在AQS的独占模式中，资源`state`同时仅允许一个线程抢占，所以除了抢占成功的节点，其他节点均处理等待状态。**但是，在AQS的共享模式中，虽然当前线程抢占了资源，但它抢占的仅是部分资源，还可能有剩余资源可被其他线程抢占，所以它要通过`setHeadAndPropagate(node, r)`唤醒其他节点线程。**

```
private void doReleaseShared() {
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 如果首节点状态是SIGNAL，则需要unpark唤醒下个节点线程
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;            // 循环检查，多次确认
                // 唤醒首节点下个节点线程
                unparkSuccessor(h);
            }
            // ws==0时，将会尝试将ws设置为Node.PROPAGATE，这样
setHeadAndPropagate读到ws<0时，就会唤醒后继节点
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0,
Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        if (h == head)                   // 循环检查，防止头部发生变化
            break;
    }
```

```
}
```

3. 总体流程图示



（三）资源释放解析

1. Semaphore部分源码解析

Semaphore通过`release()`执行资源释放，如下源码所示。`release()`直接调用AQS中的`releaseShared()`，当然这并不是说释放资源都是AQS的事而与Semaphore无关，因为`releaseShared()`中会调用Semaphore中的`tryReleaseShared()`。

```
public class Semaphore implements java.io.Serializable {
    private final Sync sync;

    abstract static class Sync extends AbstractQueuedSynchronizer
{
        Sync(int permits) {
            setState(permits);
        }
```

```
        protected final boolean tryReleaseShared(int releases) {
            for (;;) {
                int current = getState();
                int next = current + releases;
                if (next < current) // 如果传入的releases数值为负，则
抛出异常
                    throw new Error("Maximum permit count
exceeded");
                // 更新最新值
                if (compareAndSetState(current, next))
                    return true;
            }
        }
    }

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

2. AQS部分源码解析

下面是`releaseShared()`方法的源码，可以看到它只做了两件事：一是调用模板方法`tryReleaseShared()`，二是调用`doReleaseShared()`，前者由子类Semaphore提供，后者由AQS提供。

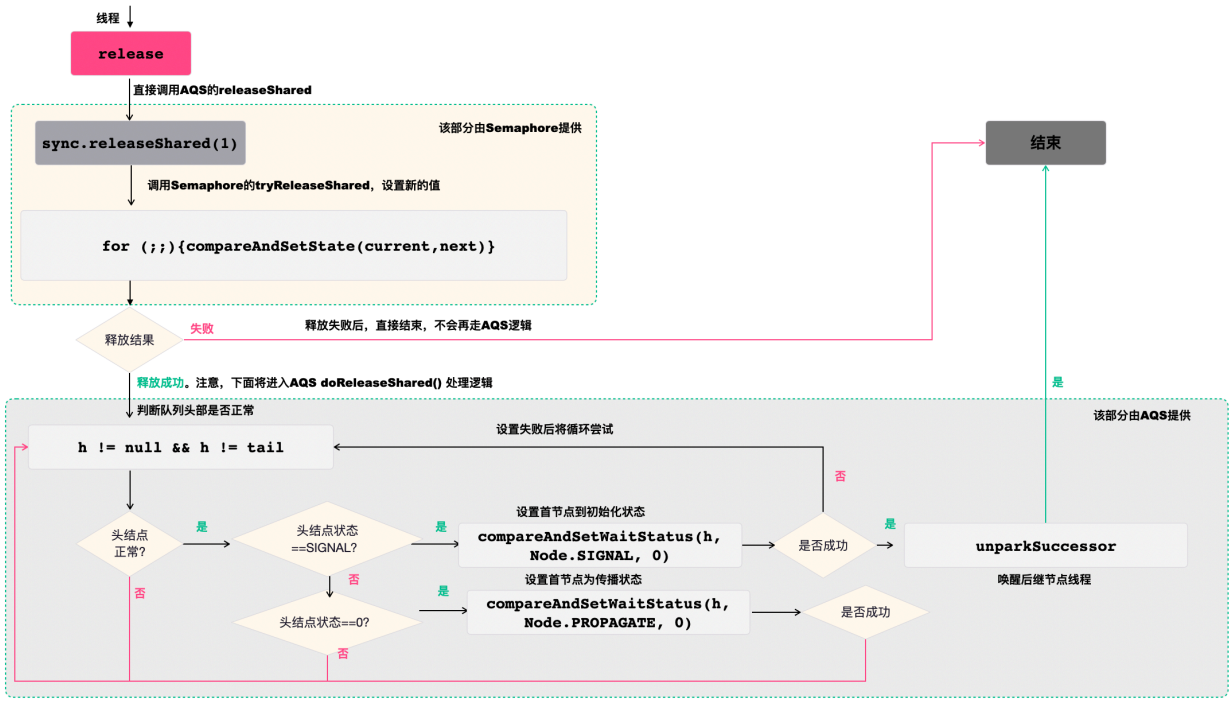
```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

`doReleaseShared()`方法是AQS共享模式下的关键方法，其实它在前面的`setHeadAndPropagate()`方法中也有引用，其关键源码如下。

```
private void doReleaseShared() {
    for (;;) {
        Node h = head;
```

```
            if (h != null && h != tail) {
                int ws = h.waitStatus;
                // 如果首节点状态是SIGNAL，则需要unpark唤醒下个节点线程
                if (ws == Node.SIGNAL) {
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                        continue;            // 循环检查，多次确认
                    // 唤醒首节点下个节点线程
                    unparkSuccessor(h);
                }
                // ws==0时，将会尝试将ws设置为Node.PROPAGATE，这样
setHeadAndPropagate读到ws<0时，就会唤醒后继节点
                else if (ws == 0 &&
                         !compareAndSetWaitStatus(h, 0,
Node.PROPAGATE))
                    continue;                // loop on failed CAS
            }
            if (h == head)                   // 循环检查，防止头部发生变化
                break;
        }
    }
```

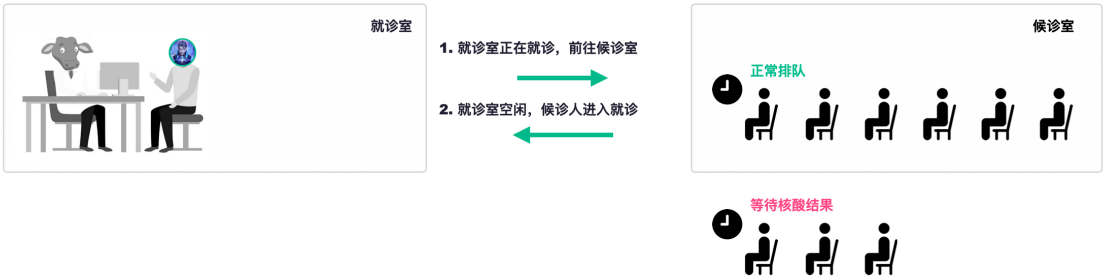
3. 总体流程图示



六、理解AQS中的条件同列

前述内容所讲的都是AQS中的同步队列，接下来我们再来探讨AQS中的条件队列。为了便于理解，我们仍然以医院就诊为例。

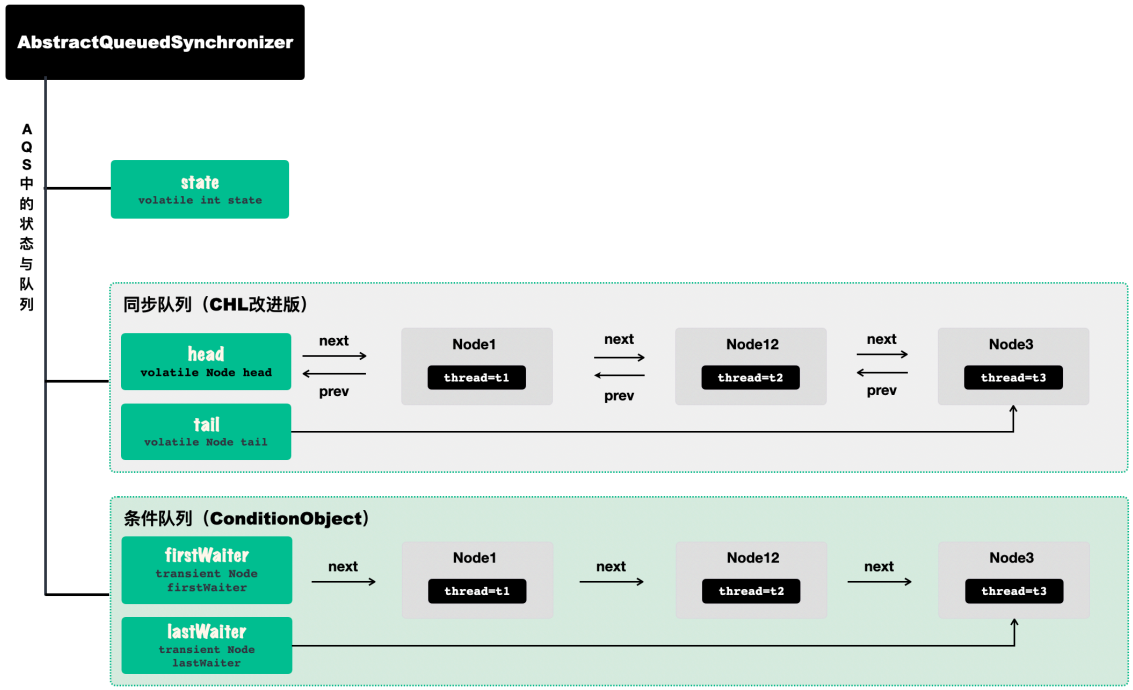
在医院候诊排队时，正常情况下按照队伍排队即可。然而，凡事都有例外，**比如防疫规定没有做核酸的不能到大厅排队，要现在外面做核酸并在结果出来后才能排队**。此时，就会出现两个队列：**一是按先来后到的正常候诊队列，二是核酸结果等待队列**。



这样的医院排队机制相信你可以理解。其实，这种机制不仅在现实中存在，在软件中也存在。作为强大的同步器，AQS所提供的条件队列正是为了解决此类问题。我们将可以将正常候诊的队列称为**同步队列**，而需要等待核酸结果出来后才能进入排队的称为**条件队列**。

所以，基于目前的理解，我们可以进一步完善AQS的整体结构：**同步状态+同步队列+条件队列**，如下图所示。

AQS的条件队列是**单向队列**，由`ConditionObject+Node`组成，ConditionObject中定义了队列的**头部（firstWaiter）和尾部（lastWaiter）**，并且头部和尾部都是Node类型。和AQS中的同步队列类似，条件队列也有着广泛的使用。



比如，ReentrantLock中的`newCondition()`方法所调用的就是AQS中的方法，而ArrayBlockingQueue则使用了ReentrantLock这一方法，相关使用如下所示。

- ReentrantLock中的方法

```
public Condition newCondition() {
    return sync.newCondition(); // 创建新的等待条件
}
```

- ReentrantLock所调用的AQS方法：

```
final ConditionObject newCondition() {
    return new ConditionObject(); // 构建等待队列
}
```

- ArrayBlockingQueue中所使用的ReentrantLock中的方法，比如`lock.newCondition()`：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
```

```
    notEmpty = lock.newCondition(); // 等待队列的具体场景应用
    notFull =  lock.newCondition();
}
```

- ArrayBlockingQueue中通过`notFull.await()`将当前线程放入条件队列，等待唤醒。

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await(); // 条件队列中的等待
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

需要注意的是，**同一线程不可以同时处于同步队列和条件队列中。当线程进入条件队列时，将自动从原来的同步队列中出队**。限于篇幅，关于条件队列的更多源码和流程在此不做更多解释，读者可以自行探索。

小结

正文到此结束，恭喜你又上了一颗星✨

在本文中，我们首先厘清了同步的概念，无论是现实生活还是软件设计，同步都是广泛的存在，而从生活中理解软件中的设计相对较为容易。对于同步问题的解决，队列是常被采用的方案。但是在软件设计中，队列的设计需要考虑到**公平性、性能和扩展性**等多个维度，所以虽然队列是AQS的核心组件之一，但是对CLH队列进行了适当的改造，以更好地适配AQS的设计理念和需求。因此，理解AQS的核心在于理解变种的CLH队列，包括它的设计理念、数据结构组成，以及出队和入队等完整过程，所以我们在开篇引入并介绍了CLH队列。

在本文的第四和第五部分，我们以ReentrantLock和Semaphore为例，介绍了AQS独占模式和共享模式下的入队和队列的变化形态，重点还是在于帮助理解CLH队

列。而在第六部分，我们介绍了**似乎鲜为人知但同样重要**的条件队列。

本文整体篇幅较长，内容较多。然而，在理解AQS时，我们不要深陷冗长的文章和源码中。首先要清楚的并非AQS是什么和它的工作原理，而是要**先搞清楚AQS所解决的是什么问题，**针对问题AQS提出了怎样的方案。如此，才能抓住AQS的核心脉络，理解它的本质。

另外，作为成熟的同步器，AQS提供了完善的各种同步机制，JDK中也提供了多样的同步实现，比如ReentrantLock、Semaphore和CountDownLatch等。**因此，在编码中需要使用同步机制时，应首先考虑现有的稳定的同步方案，其次再考虑自由地自主实现。**

夫子的试炼

- 基于AQS，设计自己的同步器：实现一个队列，三个窗口同时核酸采样。

延伸阅读与参考资料

- http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html
- https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html
- https://blog.51cto.com/u_12302616/3230929
- https://www.cnblogs.com/xijiu/p/14396061.html
- https://www.javarticles.com/2012/10/abstractqueuedsynchronizer-aqs.html
- https://www.infoq.cn/article/bvpvyvxjkm8zstspti0l
- https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/CLH.pdf
- 掘金专栏：https://juejin.cn/column/6963590682602635294
- Github：https://github.com/ThoughtsBeta/TheKingOfConcurrency

常见面试题

- 说说自己对 AQS 的理解?
- 多个线程通过锁请求共享资源，获取不到锁的线程怎么办?
- 独占模式和共享模式有哪些区别?
- 同步队列中，线程入、出同步队列的时机和过程?
- 为什么 AQS 有了同步队列之后，还需要条件队列?
- 如果一个线程需要等待一组线程全部执行完之后再继续执行，有什么好的办法么？是如何实现的?

星耀03：自在不羁-领会非阻塞的同步机制和算法

欢迎来到《王者并发课》，本文是该系列文章中的**第29篇**，星耀中的**第3篇**。

众所周知，在驾车经过拥堵路段时，我们会经常面对：**排队等待或者绕道而行**。前者可以少走弯路，而后者则可以节约时间，它们各有利弊。同样的，在软件设计中，我们也会面临类似的并发问题。因此，**阻塞还是非阻塞**，就成了我们在处理这类问题时的两种常见方案。

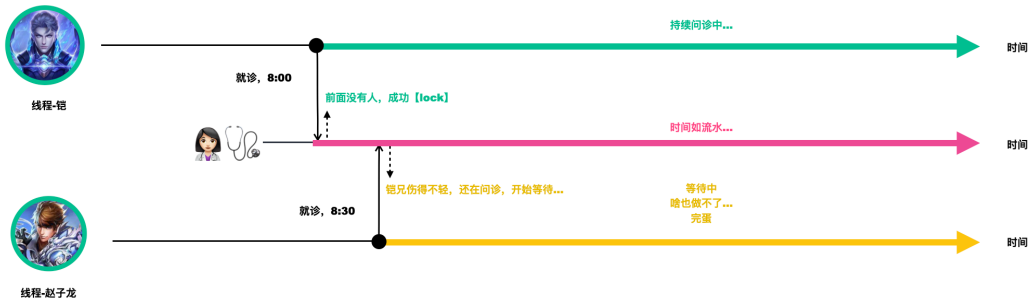
在前面的系列文章中，我们主要介绍的多是阻塞方案，比如同步队列就是典型的阻塞解决方案。然而，阻塞方案虽然可以让整体更为有序，但会降低整体的性能，不利于最大程度地使用资源。毕竟，等待是对时间的浪费。所以，在本文中，我们将通过示例来讨论处理并发的另外一种方案：**非阻塞的机制和算法实现**。

一、阻塞带来的麻烦

我们仍然以峡谷医院的就诊为例来说明阻塞带来的麻烦。

> 早晨八点整，峡谷的牛大夫开始上班。刚一落座，铠捷足先登成了她今天的第一个病人。随后，子龙在八点半到达医院，可是这时候铠正在就诊，所以他只能等待。于是，铠磨磨唧唧和医生从八点聊到了九点，子龙也就从八点半等到了九点。**注意，在这半个小时中，子龙除了等待无法做其他的事**。

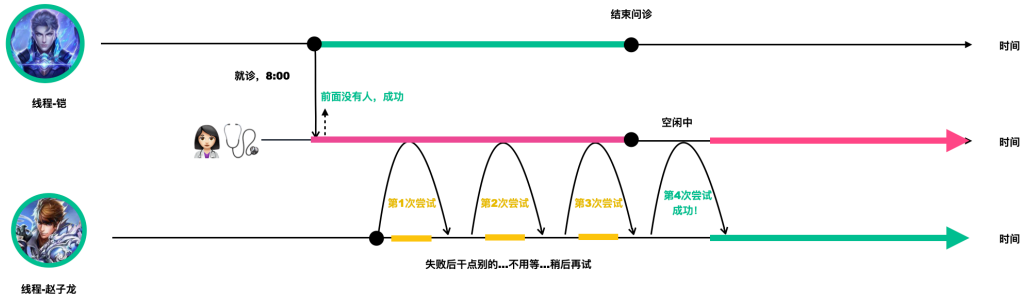
在这个过程中，我们可以理解由于铠没有及时释放资源，子龙被阻塞了。我们试想下，如果此情此景出现在软件设计中会发生什么情况？**小部分线程对资源长时间占据，将导致大量线程被阻塞，从而导致系统陷入瘫痪的状态**。如果你对此感到陌生，那也许是你还没有遇到过线程池被打满的场景。



二、非阻塞的利与弊

既然，在某些场景下，阻塞将导致系统瘫痪，那有没有办法解决呢？当然有，并且我们会自然而然地会想到**非阻塞**。比如，在上面的示例中，假如子龙并没有始终在等待，而是他每隔几分钟去了解下情况。如果医生恰好有空，那他可以直接去就诊，否则他可以做些其他的事情，比如掏出电脑写两行代码。

这就是非阻塞，当前线程在获取资源失败时，不会原地等待，而是直接返回并通过轮询等方式不断尝试。这样的好处显而易见，可以降低系统的负载，并提高线程资源的利用情况。



非阻塞算法是软件设计中的常见算法，也是一种能高性能解决高并发方案，它主要通过使用底层的原子机器指令来代替锁，从而保证数据在并发中的一致性。作为无锁方案，非阻塞方案在可伸缩性和线程的调度上拥有较大的优势，由于没有阻塞所以没有复杂的调度开销。同时，非阻塞算法也不存在死锁和其他线程状态管理问题。

当然，凡事都有两面性，有一利必有一弊，而**非阻塞算法的弊端则在于设计和实现起来很复杂**。

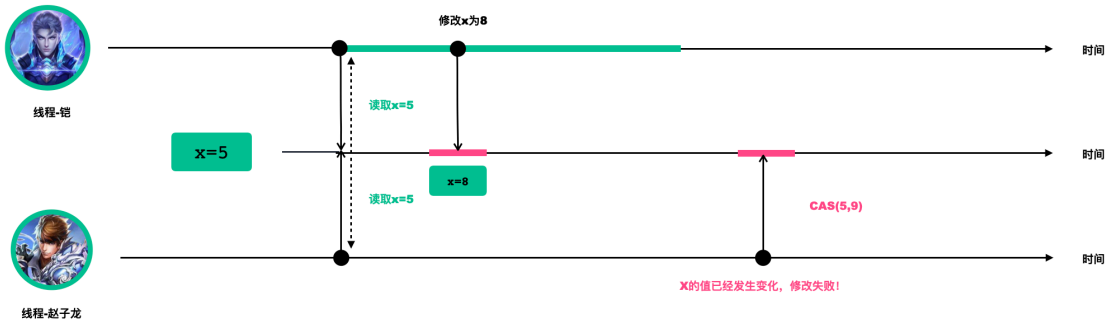
三、如何实现非阻塞设计

（一）非阻塞的基础：CAS

在设计和实现非阻塞算法时，通常会根据CAS来实现，也就是**Compare and Swap（简称CAS）**，这是一种CPU底层提供的计算能力。 CAS的核心在于，**当更新一个变量时，只有这个变量的旧值和内存中的值相同时，才会执行更新**。它是一个原子操作，也就是说数据的读取和更新是在一起的。

举个例子，子龙和铠都从内存中读取`x=5`，随后他们俩分别对x进行了更新：铠将值从5变更为8，即`CAS(5,8)`；而子龙则将试图将x从5变更为9，即`CAS(5,9)`。那么，子龙能成功吗？**当然不能**。

因为x的值已经发生了变化。**当子龙拿着旧值5去试图将x设置为9时，x的值已经不在等于5！这就是CAS的要义，要更新可以，但要和以前一样才行。**



（二）CAS的基础：volatile变量

在前面的JAVA内存模型文章中，我们详细讲述了volatile变量的作用，如果你对其不甚了解可以查阅相关章节。**简而言之，volatile可以让变量的值在变化时对其他线程可见**。也就是说，线程在读取变量时，始终从主存读取而不是缓存，从而保障读取的数据都是最新的。

我们知道，CAS的核心在于更新变量时会比较当前变量的最新值，所以CAS读取的变量必然需要最新的，所以这个变量需要是volatile类型。比如，AtomicInteger是

JAVA非阻塞设计的典型，它的内部用于计数的`value`字段便是volatile类型，相关核心源码如下所示。

根据下面的源码，我们可以清楚地看到AtomicInteger内部有个`compareAndSet`方法，这是它所提供的CAS方法。注意看，`compareAndSet`内部调用的则是**Unsafe**类所提供的`compareAndSwapInt`方法。`sun.misc.Unsafe`是个比较底层的方法，它提供了一些列的和硬件层面交互的能力，关于Unsafe我们不需要做深入的了解，在工作中也应尽量避免对它的直接使用。当然，如果你对它有兴趣，可以参考这篇文章了解更多。

```
public class AtomicInteger extends Number implements
java.io.Serializable {

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

    private volatile int value;

    public AtomicInteger(int initialValue) {
        value = initialValue;
    }

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset,
expect, update);
    }

    public final int incrementAndGet() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }
}
```

`sun.misc.Unsafe`中对底层方法的调用：

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

```
    return var5;
}
```

（三）非阻塞算法的应用

为了直观感受非阻塞方法和阻塞方法在使用时的异同，我们仍然以前面文章的就诊作为示例。在就诊时，每个医生同时只允许一个病人前往就诊，其他的病人需要排队等候。于是，我们通过`synchronized`来模拟这个场景，相关源码如下所示。由于`synchronized`的修饰，`diagnosis`方法是阻塞的，未获得同步锁的线程将处于阻塞状态。

```
/**
 * 当前是否可以就诊
 */
private volatile boolean isAvailable;

public synchronized void diagnosis() {
    try {
        isAvailable = true;
        // ... 就诊中
    } finally {
        isAvailable = false;// 就诊结束离开后，释放资格。
    }
}
```

现在，我们将上述示例代码由阻塞改为非阻塞，如下源码所示。注意，我们将控制就诊状态的变量由`volatile boolean isAvailable`变更为`AtomicBoolean isAtomicAvailable`，并且`diagnosis`方法没有再使用`synchronized`修饰。

重点在于`while`循环中的条件控制逻辑。和阻塞算法明显不同的是，非阻塞算法在抢占失败时，不会进入等待状态，而是不断地尝试直至成功。

```
/**
 * 当前是否可以就诊
 */
private final AtomicBoolean isAtomicAvailable = new
AtomicBoolean();
```

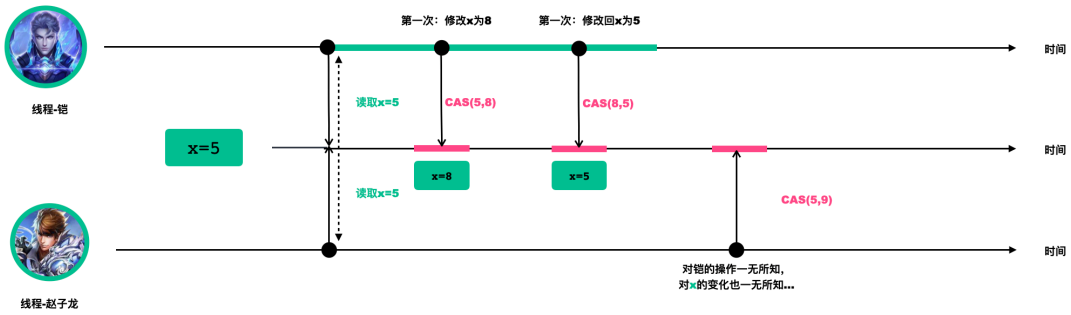
```
public void diagnosis() {
    try {
        while (isAtomicAvailable.compareAndSet(false, true)) {
            // ... 就诊中
        }
    } finally {
        isAtomicAvailable.set(false);// 就诊结束离开后，释放资格。
    }
}
```

AtomicInteger只是一个典型的非阻塞算法的示例。在Java中的`java.util.concurrent.atomic`包中，有大量类似的AtomicXXX工具，它们长相略有不同但原理类似，比如**AtomicBoolean**、**AtomicLong**和**AtomicIntegerArray**等。借助于这些工具，可以帮助我们很方便地实现各种非阻塞的原子性操作。

四、ABA问题与破解

虽然CAS足够强大且易用，但并不意味着它完美无缺。对于老道的程序员来说，ABA问题便是其瑕疵之一。那什么是ABA问题?

我们知道，CAS在计算时，会计算传入的期望值和现有的内存值是否一致，如果不一致则拒绝计算。那么，假如内存值从A变成B再变回A时，其他线程是否知道？比如，下面的图中，铠闲得无聊对`x`进行了一同修改：从5改成8，又从8改成了5. 铠这么牛逼，子龙知道吗?



五、非阻塞算法在JAVA中的应用

小结

正文到此结束，恭喜你又上了一颗星✨

在本文中，我们首先厘清了同步的概念，无论是现实生活还是软件设计，同步都是广泛的存在，而从生活中理解软件中的设计相对较为容易。对于同步问题的解决，队列是常被采用的方案。但是在软件设计中，队列的设计需要考虑到**公平性、性能和扩展性**等多个维度，所以虽然队列是AQS的核心组件之一，但是对CLH队列进行了适当的改造，以更好地适配AQS的设计理念和需求。因此，理解AQS的核心在于理解变种的CLH队列，包括它的设计理念、数据结构组成，以及出队和入队等完整过程，所以我们在开篇引入并介绍了CLH队列。

在本文的第四和第五部分，我们以ReentrantLock和Semaphore为例，介绍了AQS独占模式和共享模式下的入队和队列的变化形态，重点还是在于帮助理解CLH队列。而在第六部分，我们介绍了**似乎鲜为人知但同样重要**的条件队列。

本文整体篇幅较长，内容较多。然而，在理解AQS时，我们不要深陷冗长的文章和源码中。首先要清楚的并非AQS是什么和它的工作原理，而是要**先搞清楚AQS所解决的是什么问题，**针对问题AQS提出了怎样的方案。如此，才能抓住AQS的核心脉络，理解它的本质。

另外，作为成熟的同步器，AQS提供了完善的各种同步机制，JDK中也提供了多样的同步实现，比如ReentrantLock、Semaphore和CountDownLatch等。**因此，在编码中需要使用同步机制时，应首先考虑现有的稳定的同步方案，其次再考虑自由地自主实现**。

夫子的试炼

- 基于AQS，设计自己的同步器：实现一个队列，三个窗口同时核酸采样。

延伸阅读与参考资料

- http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/

- 掘金专栏：**https://juejin.cn/column/6963590682602635294**
- Github：**https://github.com/ThoughtsBeta/TheKingOfConcurrency**

常见面试题

- 说说自己对 AQS 的理解?
- 多个线程通过锁请求共享资源，获取不到锁的线程怎么办?
- 独占模式和共享模式有哪些区别?
- 同步队列中，线程入、出同步队列的时机和过程?
- 为什么 AQS 有了同步队列之后，还需要条件队列?
- 如果一个线程需要等待一组线程全部执行完之后再继续执行，有什么好的办法么？是如何实现的?

关于作者

秦二爷

知名互联网公司非知名程序员。

从业近十年，先后从事敏捷与DevOps咨询、Tech Leader和管理等工作，对分布式
高并发架构有丰富的实战经验，热衷于技术分享和特定领域书籍翻译。

联系方式

- 掘金主页：**https://juejin.cn/user/4081843403490984**
- Github：**https://github.com/ThoughtsBeta**
- 邮件：**thoughts.beta@gmail.com**
- 公众号私信：MetaThoughts

高并发姊妹篇推荐

《高并发秒杀的设计精要与实现》是该文集的姊妹篇，也是JAVA并发领域的补充和延伸，专注于从架构层面解决实际工作中的高并发问题。

高并发秒杀的设计精要与实现

以源码解构高并发秒杀核心要义，进阶与面试手册（Java版）。

秦二爷 专注高并发领域创作，人气专栏《王者并发课》作者。 加入学习群

继续学习

1405 人加入学习 | 18 小节 · 约 487分12秒

小册地址：https://juejin.cn/book/7008372989179723787