Java多线程的发展与实现

摘要

# 当前计算机CPU的性能已经足够支撑同时运行多个线程，当某一线程的处理不需要占用CPU而只需要I/O,其他线程就可以在操作系统的调度下的使用CPU，本文主要讨论了操作系统对于Java多线程的调度，分析了Java从单线程到多线程到纤程的发展历程，通过对比三者之间的优缺点，强调了Java多线程编程在项目开发中的重要性。随后讨论了两种实现Java多线程的方法及在编程中可能遇到的安全性问题，无论线程的执行顺序，最终的执行结果必须是我们想要的，这就要求我们在开发过程中考虑到多线程的并发问题，并重点讨论了Synchronized和Lock两种解决安全性问题的方法。本文通过代码段的方法作为例子，证明了Java多线程的可行性和安全性。

ABSTRACT

The performance of the current computer CPU is sufficient to support multiple threads running at the same time. When a thread does not need to occupy the CPU but only needs I/O, other threads can use the CPU under the scheduling of the operating system. This article mainly discusses the operation For the scheduling of Java multithreading, the system analyzes the development process of Java from single thread to multithread to fiber. By comparing the advantages and disadvantages of the three, the importance of Java multithreading programming in project development is emphasized. Two methods of implementing Java multi-threading and the security issues that may be encountered in programming are then discussed. Regardless of the execution order of threads, the final execution result must be what we want, which requires us to consider in the development process The concurrency problem of multithreading, and two methods to solve the security problem of Synchronized and Lock are discussed. This article uses the code segment method as an example to prove the feasibility and safety of Java multithreading.

目录

[1 线程及其作用 1](#_Toc89193483)

[2 Java多线程的发展历程 2](#_Toc89193484)

[2.1 Java单线程处理任务 2](#_Toc89193485)

[2.2 Java多线程处理任务 2](#_Toc89193486)

[2.3 Java单线程与多线程的优劣 3](#_Toc89193487)

[2.3.1 Java单线程的优劣 3](#_Toc89193488)

[2.3.2 Java多线程的优劣 3](#_Toc89193489)

[2.4 Java多线程的发展——纤程 4](#_Toc89193490)

[3 Java多线程的实现方法 4](#_Toc89193491)

[3.1 Java多线程综述 4](#_Toc89193492)

[3.2 继承Thread类实现多线程 4](#_Toc89193493)

[3.3 实现Runnable接口实现多线程 6](#_Toc89193494)

[3.4 Thread 和 Runnable的区别 7](#_Toc89193495)

[3.5 Java多线程的优化——线程排序 7](#_Toc89193496)

[4 Java多线程的安全性问题 9](#_Toc89193497)

[4.1 产生原因 9](#_Toc89193498)

[4.2 如何解决安全性问题 10](#_Toc89193499)

[4.2.1 使用Synchronized关键字 10](#_Toc89193500)

[4.2.2 使用lock接口 10](#_Toc89193501)

[参考文献 12](#_Toc89193502)

# 1 线程及其作用

现代计算机的核心部件是CPU，整个计算机硬件系统都是围绕CPU的功能和需求所组建的，CPU几乎承担了计算机所有的计算任务（GPU也承担了一部分计算任务）；而操作系统是计算机的管理者，它负责计算任务调度，资源的管理和分配等。进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。

随着时间的发展，日益复杂的进程切换所需要的开销，对CPU的要求越来越高，线程就应运而生。线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。一个标准的线程由线程ID，当前指令指针PC，寄存器和堆栈组成。而进程由内存空间(代码，数据，进程空间，打开的文件)和一个或多个线程组成，进程和线程的关系如下图2所示。总之，线程和进程都是一种抽象的概念，线程是一种比进程还小的抽象，线程和进程都可用于实现并发，图1展示了它们之间的关系。在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位，它相当于一个进程里只有一个线程，进程本身就是线程。在多线程的演变中，延伸出了多线程的相关框架，如Niculescu等人[1]提出的基于JPLF框架的多线程编程模型等。

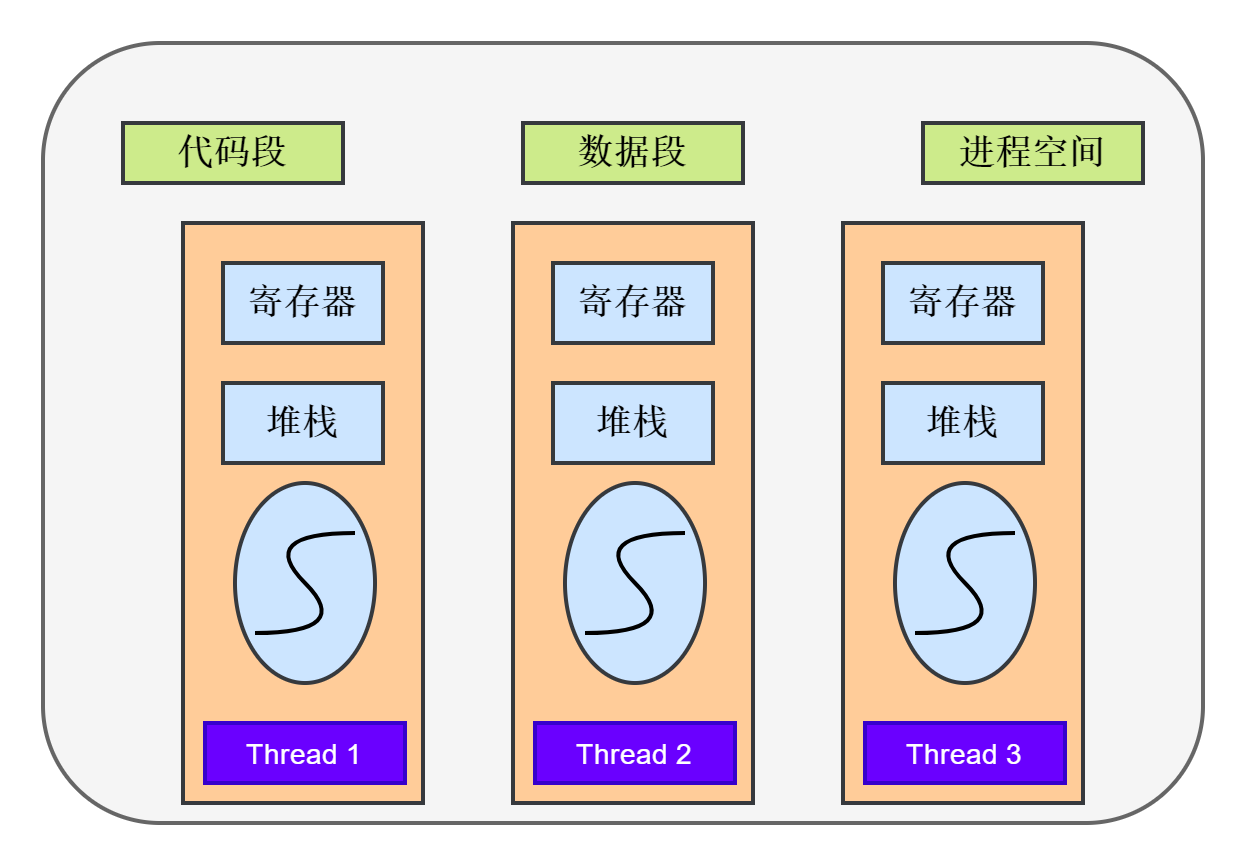


图1 进程与线程的关系

# 2 Java多线程的发展历程

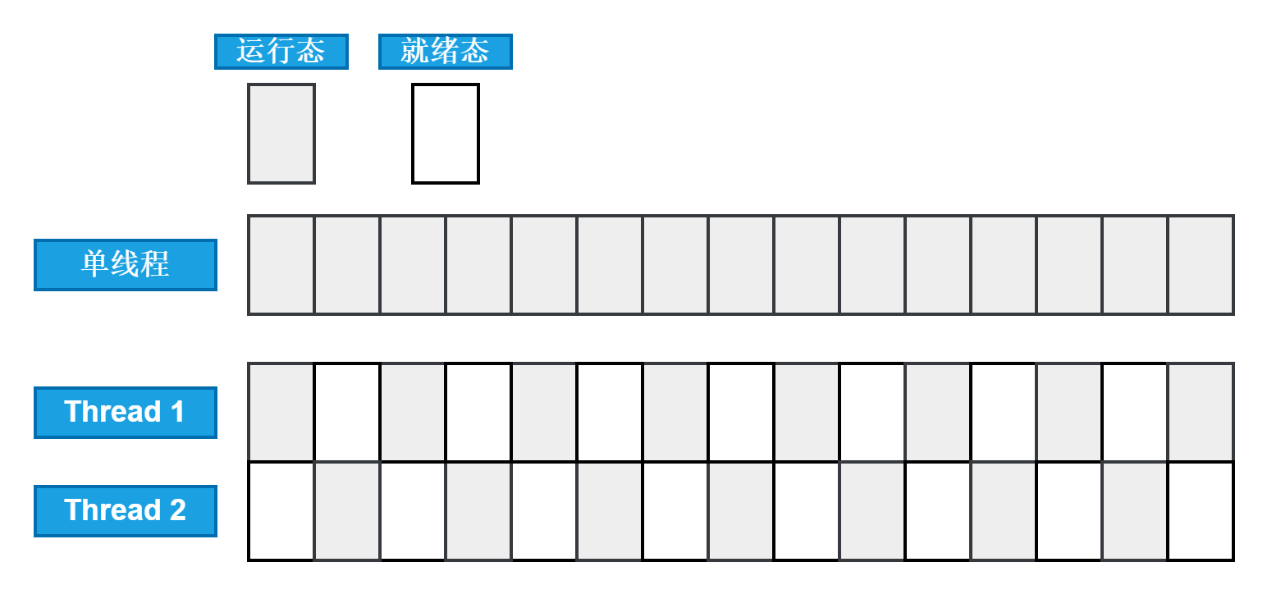
## 2.1 Java单线程处理任务

在计算机CPU性能还比较落后的时代，CPU主要的工作方式是单线程处理任务，所谓单线程，就是同一时刻只有一个线程在工作，一个程序在执行时，它的执行路径是按顺序排列的。例如对于一个C语言程序，默认情况下，在执行时系统为该程序分配一个主线程，该线程执行程序中以Main方法开始和结束的代码。线程具有生命周期，它包含3个状态，分别为出生状态、就绪状态和运行状态。出生状态就是用户在创建线程时处于的状态，在用户使用该线程实例调用Start方法之前，线程都处于出生状态；当用户调用Start方法后，线程处于就绪状态（又被称为可执行状态）；当线程得到系统资源后就进入运行状态。

## 2.2 Java多线程处理任务

随着时间的发展，单进程就是单线程的时代已经过去。CPU的切换单位是基于进程的，在单线程时代，一个线程正在工作，就意味着其他线程无法请求CPU的资源完成计算，这就好比我们在使用浏览器的同时就不能运行微信。另外，如果当前线程正在执行一个需要较长时间的IO操作时，那些不需要执行IO操作的线程也必须等待当前线程执行完成之后才可以运行。所以，慢慢衍生出了多线程的方法来解决这个难题，每个进程都可以生成多个线程来执行一个程序，每个线程之间没有很明确的边界。 CPU 在线程之间来回移动，这样导致了CPU对于时间片的划分更加精细，以满足多线程的切换使用。没有什么明显的。

引入多线程的目的，就是"最大限度地利用CPU资源",随着计算机CPU的发展，性能已经足够支撑同时运行多个线程。当某一线程的处理不需要占用CPU而只和I/O,其他线程就可以“见缝插针”的使用CPU。在多线程中，往往会有一个主线程，每个程序执行时都会至少产生一个进程（有的产生多个进程），而每一个进程至少要有一个主线程。这个线程其实是进程执行的一条线索，除了主线程外你还可以给进程增加其它的线程，也即增加其它的执行线索，由此在某种程度上可以看成是给一个应用程序增加了多任务功能。当程序运行后，您可以根据各种条件挂起或运行这些线程，尤其在多CPU的环境中，这些线程是并发运行的。多线程就是在一个进程内有多个线程，从而使一个应用程序有了多任务的功能。多线程与单线程的执行差异如图2所示。

图2 单线程与多线程的执行

## 2.3 Java单线程与多线程的优劣

### 2.3.1 Java单线程的优劣

单线程有如下优点。

1. 单线程编写的代码逻辑非常清晰明朗，按顺序执行即可。
2. 不会因为任务并行而产生死锁问题，因此可以不用设置锁，减少了操作系统的开销。
3. CPU不会因为切换线程而产生额外的开销，开销全部来自于线程自身的运行。

相比之下，单线程的缺点较为明显，这也是为什么单线程逐渐被淘汰的原因。

1. 效率低下，CPU利用率极地，无法发挥CPU的全部性能。
2. 对于复杂程序的处理表现远远不如多线程，且执行缓慢。
3. 不具有并发性，必须等之前的线程执行完后才可以执行下一个线程。

### 2.3.2 Java多线程的优劣

多线程有如下优点。

1. 极大提高CPU的利用率。在多线程程序中，一个线程必须等待的时候，CPU可以运行其它的线程而不是等待，这样就大大提高了程序的效率。
2. 程序可以并发执行，减少了程序的运行时间。

同时，多线程的并发执行就必然导致了一些问题。

1. 多线程需要协调和管理，所以需要CPU时间跟踪线程。
2. 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题。
3. 线程太多会导致控制太复杂，最终可能造成很多Bug。

## 2.4 Java多线程的发展——纤程

当代并发服务器应用程序通常由较小的独立服务构建，并且经常不得不在过大的负载下执行。传统Java多线程的应用程序不容易扩展，主要原因是由于过于依赖于操作系统内核线程，只能在有限的硬件资源上创建数量有限的线程。另外，重量级操作系统线程的实现非常健壮和通用以支持各种使用场景，因此在解决特定应用程序需求方面浪费资源并且通常效率低下，这些情况催生了纤程的产生[2]。纤程又叫协程，是随着线程发展产生的绿色线程，即“线程中的线程”，它切换和调度不需要经过操作系统，它具有如下优点：占有的资源少，例如通常情况下，操作系统要启动一个线程，需要为该线程预留大约4M的内存空间，而对于同样的纤程只需要预留大约4K的内存空间，显著减小了占用的系统资源；切换简单，由于它占有的资源小，因此切换时非常迅速；在同一时间内可以有非常多的纤程被启动，远多余系统能运行的线程数量。

[Pufek](https://ieeexplore.ieee.org/author/37086887208)等人提出了一种基于Java 虚拟机的新颖的结构化并发模型，该模型引入了 OpenJDK 的 Loom 项目[3]，实现了轻量级纤程的工程化应用，逐渐成为目前主流Java的开发方法。

# 3 Java多线程的实现方法

## 3.1 Java多线程综述

Java实现多线程是Java的核心功能之一，尤其是在后端开发和服务器端开发，Java多线程技术是必不可少的。通常情况下，Java实现多线程有3中方法，第一种是通过继承Java的Thread类实现多线程，第二种方法是实现Runnable接口实现Java多线程，第三种方法是通过与Future和Java线程池配合**实现Callable接口[4][5]**，从而实现多线程，由于第三种实现方法比较复杂且并不常见，因此本文着重论述前两种实现方法。

## 3.2 继承Thread类实现多线程

尽管在这里，我将继承Thread类的方法作为一种实现Java多线程的方法，但是不得不指出，Thread本质上也是实现了Runnable接口的一个实例，在这里它代表的是一个线程的实例，并且，必须通过start（）方法来启动这个实例，这是启动线程的唯一方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法，如下图3所示，运行两次的结果如图4所示。

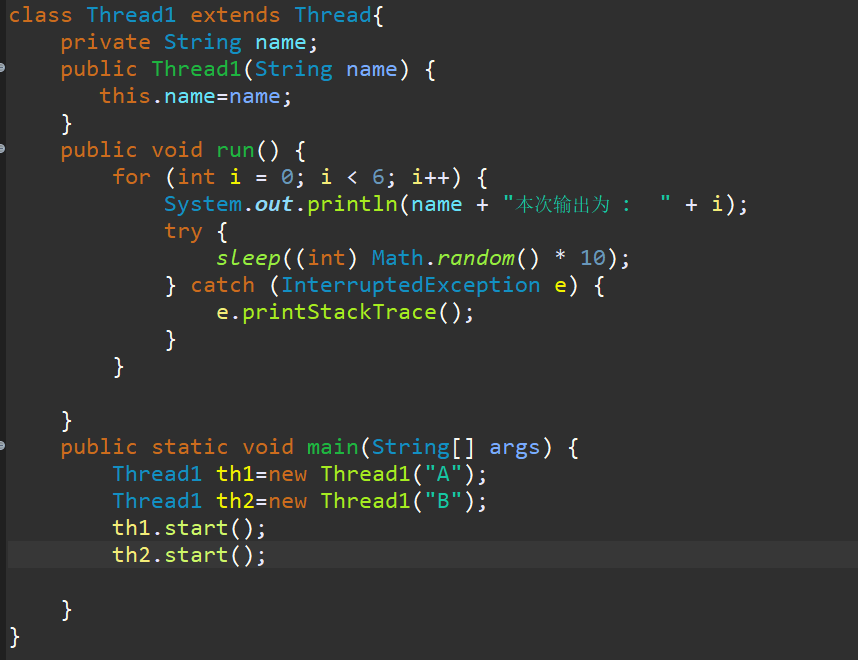


图3 继承Java的Thread类实现多线程

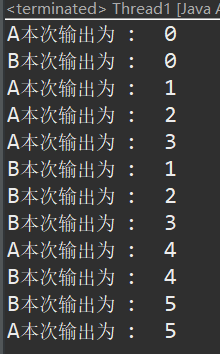
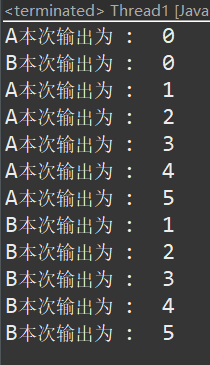


图4 两次的运行结果

从上述所举例子的程序源码中我们可以看到，当执行main（）方法时，Java虚拟机维护了一个进程，随着th1和th2的start（）方法被调用，分别创建了2个线程，因此该程序就处于多线程运行状态。

## 3.3 实现Runnable接口实现多线程

采用Runnable也是非常常见的一种Java多线程方法，需要在程序中重写run（）方法[6]。在编程中，这种方法比较容易上手，图5是该方法的一个实例。



图5 实现Runnable接口实现多线程

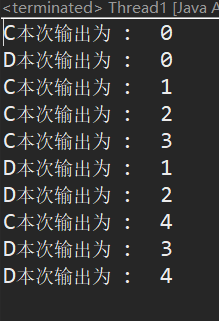
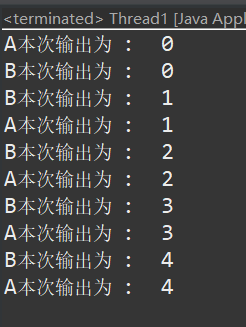


图6 实现Runnable接口实现多线程的运行结果

从上述代码中我们可以看到，实现Runnable接口的步骤为：建立Runnable 对象，使用参数为Runnable 对象的构造方法进行创建Thread 实例，对实例调用start()方法启动线程，这样一来也很好的避免了单线程的局限性。

## 3.4 Thread 和 Runnable的区别

首先，最直观的区别是，Thread是类，而Runnable是接口，但是Thread类事实上实现了Runnable接口。在使用Runnable定义的子类中没有start（）方法，只有Thread类中才有，而在Thread类中，有一个构造方法：public Thread(Runnable targer)，此构造方法接受Runnable的子类实例，也就是说可以通过Thread类来启动Runnable实现的多线程。

其次，从资源共享的角度来说，如果一个类继承了Thread则不方便共享资源，如果实现了Runnable接口，则可以很方便的实现资源共享。因为Runnable**可**以使用多个相同的程序代码的线程去处理同一个资源，这就使得程序具有很强的健壮性，代码可以被多个线程共享，代码和数据独立。但是，如果程序需要使用线程池，那么线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类。

另外，对于run（）方法和start（）方法的具体实现，也存在差异，用start方法来启动线程，真正实现了多线程运行，这时无需等待run方法体代码执行完毕而直接继续执行下面的代码。通过调用Thread类的 start()方法来启动一个线程，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到cpu时间片，就开始执行run()方法，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，Run方法运行结束，此线程随即终止。而run()方法只是类的一个普通方法而已，如果直接调用Run方法，程序中依然只有主线程这一个线程，其程序执行路径还是只有一条，还是要顺序执行，还是要等待run方法体执行完毕后才可继续执行下面的代码，这样就没有达到写线程的目的。实际上，public static void main(String[] args)方法也是一个线程，和继承Thread类的线程一样，都在运行后处于可运行态，等待操作系统的调度。

## 3.5 Java多线程的优化——线程排序

由于当前的CPU可以同时执行多个任务，所以在Java多线程并发运行时，可以利用分治法的原理，让一些不相关的操作同步进行，可以有效提高执行效率。但是，如果线程过多，线程频繁切换，会导致性能下降，因此需要优化Java多线程的实现过程，比较主流的做法是将进程中的数据划分为若干个部分，利用计算机的并行能力，对于不同部分的数据分别放置不同的资源来提高性能。

在目前最新的研究进展中，出现了一些Java多线程的优化方案，Deming Wang等人[7]提出了一种基于Java多线程的快速排序策略，并与Java多线程的顺序执行作了详细对比，并且使用各种性能指标验证结果，例如空闲时间百分比、最大频率百分比、处理器效用百分比、处理器时间百分比和总执行时间，结果显示多线程快速排序显示出比顺序快速排序更高的效率，首先将数据分成若干段，每段使用一个并发线程进行快速排序，然后通过合并算法将所有段合并为一个段，最后所有的数据已经排序完毕，理论上，快速排序的时间复杂度是O(nlog(n))，在此方法中的时间复杂度是O((n/m)log(n/m)，因为数据分为m段。归并排序的时间复杂度是 O(n)，合并排序的时间复杂度由于并发所以小于 O(n/2)，所以我们实现的时间复杂度小于 O((n/m)log(n/m))+ O(n/2)，该过程的流程图如下图6所示。

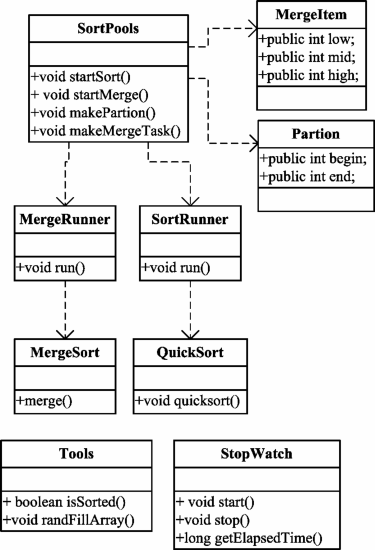


图7 Java多线程快速排序算法

# 4 Java多线程的安全性问题

## 4.1 产生原因

在多道程序设计时，Java多线程的安全性问题是一个必须考虑的因素。何为安全性问题？如果一个进程中含有多个线程，如果不管这些线程的执行顺序是怎样的，操作系统是如何调度的，最终运行的结果都能达到我们预期的结果，那么我们就认为这个多线程系统是安全的。我们考虑下图8的例子。



图8 Java多线程的安全性问题例子

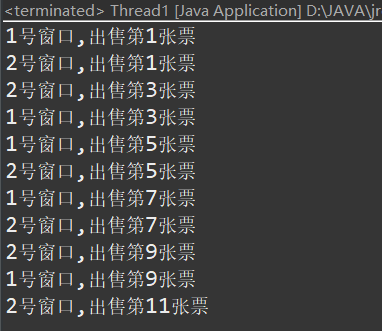


图9 Java多线程的运行结果

可以看到，这里多线程的并发执行出现了严重的错误，一共只有10张票，结果却卖出了11张票，这个结果无疑是荒唐的。出现上述结果的原因，就是当只剩下最后一张票时，两个线程同时进入了while循环，都获取了count的值，让其-1，但是两个线程没有进行同步，导致第一个线程修改了票数显示为10，第二个线程仍然继续修改了票数显示为11。

## 4.2 如何解决安全性问题

### 4.2.1 使用Synchronized关键字

Synchronized是Java的一个自动锁，当Synchronized用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码，当两个并发线程访问同一个对象object中的这个Synchronized (this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块；当一个线程访问object的一个Synchronized (this)同步代码块时，另一个线程仍然可以访问该object中的非Synchronized (this)同步代码块。第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个Synchronized (this)同步代码块时，它就获得了这个object的对象锁，所以将会导致其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。因此，使用synchronized关键字可以有效的避免出现多线程并发带来的安全性问题，但是，它的效率非常低，多个线程需要判断锁，比较消耗资源，抢锁的资源。

### 4.2.2 使用lock接口

Synchronized的使用有时候会带来这样的问题：如果一个获取Synchronized锁的线程由于要等待IO被阻塞了，但是又迟迟不释放锁，其他线程就会产生长时间等待，会极大的影响程序的执行效率。而lock就是解决这个问题的，它可以避免线程长时间的等待下去。

lock可以配置公平策略, 在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照先来先服务的规则从队列中获取自己的锁，保证了线程按一定顺序获取锁。lock接口主要提供了trylock（）、unlock（）、lock（）等方法，其中trylock（）方法可以试图获取锁，获取到或获取不到时，返回不同的返回值，让程序可以灵活处理，不会一直长时间的等待，lock（）和unlock（）可以在不同的方法中执行,可以实现同一个线程在上一个方法中lock（）在后续的其他方法中unlock(),比Synchronized灵活的多。但是，和Synchronized不同的是，lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。

# 参考文献

1. V. Niculescu, F. Loulergue, D. Bufnea and A. Sterca, "A Java Framework for High Level Parallel Programming using Powerlists", Parallel and Distributed Computing Applications and Technologies (PDCAT), pp. 255-262, 2017.
2. M. Toro, How to Implement Lightweight Threads, Universidad EAFIT, Jan 2019.
3. P. Pufek, D. Beronić, B Mihaljević and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine", Proc. of the 43rd International Convention on Information Communication and Electronic Technology MIPRO 2020, 2020.
4. Stream (Java Platform SE 8), 02 2020, [online] Available: https://docs.oracle. com/javase/8/docs/api/java/util/stream/Stream.html.
5. The Java Tutorials: Fork/Join, 02 2020, [online] Available: https://docs.oracle. com/javase/tutorial/essential/concurrency/forkjoin.html.
6. Intel MPI Library Developer Reference for Linux OS: Java Bindings for MPI-2 Routines, 02 2020, [online] Available: <https://software.intel.com/en-us/mpi-developer-reference-linux-java-bindings-for-mpi-2-routines>.
7. Deming Wang et al., "An implementation of sorting algorithm based on Java multi-thread technology", Computer Science and Electronics Engineering (ICCSEE), vol. 1, 2012.