Java 多线程同步管程机制研究

王若寒,张新昊,刘　怡,陈　益
(湖北工业大学,湖北武汉　430068)

[摘　要]多线程在生活中有广泛的应用,因此实现多线程同步管程,避免多线程模型中线程之间产生冲突,有十分重要的意义。介绍多线程和线程同步的相关概念,基于 Java 语言设计实验,着重分析了两种主要的同步技术:同步方法与同步代码块,得到 Java 多线程同步管程的原理。给出对应的应用实例,研究线程之间没有同步带来的后果,并根据多线程同步管程的原理,同时使用 ExecutorService 创建线程池,对不安全应用实例进行改进,验证其结果可行性,最后简述了多线程技术的发展趋势。

[关键词]Java;多线程;同步机制;线程安全

[作者简介]王若寒,女,河北廊坊人,湖北工业大学在读本科生,研究方向:集成电路设计与集成系统。张新昊,男,山西太原人,湖北工业大学在读本科生,研究方向:计算机科学与技术。刘怡,女,湖北黄石人,湖北工业大学在读本科生,研究方向:软件工程。通信作者:陈益,女,副教授,研究方向:集成电路设计、芯片专业人才培养中的大思政建设。

[基金项目]本文系 2024 年度大学生创新创业训练计划项目“基于 Java 多线程同步管程机制关键技术研究”(项目编号:20240200075)。

[DOI]https://doi.org/10.62662/kjxk0103001　　[中图分类号]TP312

[本刊网址]www.oacj.net　　[投稿邮箱]kjxk999@163.com

引言

在当今计算机技术迅猛发展的时代,现代应用程序对性能和响应速度的要求日益提高。多线程作为一种能有效提升程序运行效率的技术被广泛关注和应用。在多种编程语言中,Java 语言具有丰富的标准库,成为多线程技术的广泛应用领域。在 Java 语言中,多线程技术的实现有强大的支持机制,如线程池、同步控制和并发运行等。线程池可以有效地管理线程的生命周期,避免频繁创建和销毁线程带来的性能开销;同步控制机制则能够确保多个线程在访问共享资源时不会出现数据不一致的问题;并发运行机制则使得多个线程可以同时执行,充分利用多核处理器的计算能力。这些机制使得 Java 成为开发高效能、可靠性强的多线程应用的不错选择。通过多线程,程序开发者能充分利用多核处理器来提升程序的运行效率。

然而,多线程编程也带来了诸多挑战,如线程安全、死锁和资源竞争等问题,这些都需要开发者在设计和实现过程中加以重视。本文旨在通过同步管程的方法,避免多线程互相影响,使得计算机运行更加协调,从而确保运行结果的正确性、稳定性。同步管程是一种高级的同步机制,它通过提供一种更加灵活和高效的同步方式,使得开发者能够更好地控制线程之间的同步操作。通过使用同步管程,开发者可以避免线程之间的直接通信,从而减少线程之间的冲突和竞争。本文将详细介绍同步管程的原理和实现方法,并通过具体的实例来展示其在多线程编程中的应用效果。

一、概念

(一)进程、线程

首先需要明晰进程的概念,进程是单次运行起来的程序,是一个动态的理论。而线程是系统可以进行运算实时调度的基本单位,可被包含于进程当中,是进程中的具体运行单位及最小的作用流,因此线程又叫轻量级进程。在线程同步程序中,每一个线程能够独立进行,分享进程的资源共享。总之,进程具有操作系统的资源共享(如内存等),是资源配置的基本单位,而线程是功能执行的基本单位。

(二)并行与并发

并行与并发都可以实现同时做两件事情这个效果。并行是几件任务分给多个执行者同时执行;并发是几件任务通过一个执行者在同一时间段内依次发生,快速切换,即在逻辑上实现同时,并非真

正的同时。

如今我们使用的多 CPU 技术是指物理上存在多个 CPU,多个 CPU 之间通过总线进行通讯。可想而知,这种方法带来的弊端是通讯速度慢。而多核技术是指一个 CPU 中有多个核心,多个核心之间通过 cache 进行通讯,外设与 CPU 之间的通讯仍然通过总线进行。

操作系统会将 CPU 分成不同的时间片。当多进程使用多 CPU 时,不同的进程会分配给不同的 CPU 运行,多进程之间并行。如果进程数小于 CPU 数,多余的 CPU 就会空闲。当多进程使用单 CPU 时,不同的进程会分配给不同的时间片,多进程之间并发。

多线程的概念则有两种:用户态多线程和内核态多线程。多核 CPU 可以支持内核态多线程并行。当多线程使用多核时,不同的线程会分配给不同的核,实现多线程并行。若线程数小于核心数,多余的 CPU 核心就会空闲。当多线程使用单核 CPU 时,线程利用时间片完成高并发。

(三)线程创建的方法及原理

1.继承 thread 类

让定义的新类继承原有 thread 类的功能特性,在这一过程中,必须重新写入 run( )方法,即 run( )方法中的功能需要根据新类功能修改。最后在 main( )方法中创建新类所需的对象。

示例如图 1 所示:

```
class MyThread extends Thread
{
String s;
int m,count=0;
MyThread(String ss,int mm){s=ss;m=mm;}
    public void run()
    {
    try{
       while (true){
        System.out.print(s);
        System.out.flush();
        sleep(m);
        if(++count >=20) break;
        }
       }
    catch(InterruptedException e){return;}
    }

 Run | Debug
 public static void main(String args[])
  {
    MyThread you=new MyThread(ss:"you ",mm:50);
    MyThread me =new MyThread(ss:"me ",mm:100);
    you.start();
    me.start();
   }
}
```

图 1　继承 thread 类创建线程实例

2.实现 Runnable 接口创建线程

如图 2 所示,创建一个继承 Runnable 接口的自定义类,自定义类能够实现 Runnable 接口功能,并覆盖了 run( )方法。使用 thread 类创建一个线程对象,将 MyRunnable 实例作为目标任务。

// 定义一个实现 Runnable 接口的类

```
class MyRunnable implements Runnable {
    String s;
    int m, count = 0;

    MyRunnable(String ss, int mm) {
        s = ss;
        m = mm;
    }

    public void run() {
        try {
            while (true) {
                System.out.print(s);
                System.out.flush();

                if (++count >= 20) break;

                Thread.sleep(m); // 延时，防止输出过快
            }
        } catch (InterruptedException e) {
            return;
        }
    }

    Run | Debug
    public static void main(String args[]) {
        Runnable you = new MyRunnable(ss:"you ", mm:50);
        Runnable me = new MyRunnable(ss:"me ", mm:100);

        Thread thread1 = new Thread(you);
        Thread thread2 = new Thread(me);

        thread1.start();
        thread2.start();
    }
}
```

图 2　继承 Runnable 接口创建线程实例

注意,无论采用哪种方法创建线程,最后都需使用关键字 new 进行类中对象(图 1 中 you 和 me,图 2 中 thread1 和 thread2)的创建。

(四)线程的生命周期

Java 线程的生命周期包括新建、就绪、运行、阻塞和死亡等状态,如图 3 所示。理解线程的生命周期对于编写正确的多线程程序至关重要。新建状态,在线程被创建的初始时刻,线程并不能执行功能,只是被分配了一块特定的内存;只有在 main( )方法中调用所创建线程的 start( )方法,线程才真正进入就绪状态,等待 CPU;其后,进入运行状态执行特定功能;运行状态的线程通过一些方法(如 sleep( )、wait( )等),暂停线程在特殊情况下的执行;在终止状态,线程调用 stop( )方法使其无法继续运行。

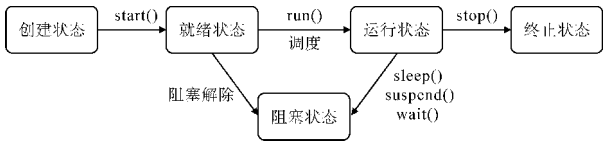

图 3 线程的操作状态

为了减少在创建和销毁线程时所产生的性能开销,Java 中采用线程池这一执行器,用于在一个后台线程中执行任务。通过重用已经创建的线程来执行新的任务,线程池提高了程序的性能和资源利用率。线程池可以通过 Executors 类提供的几个静态方法来创建。

(五)线程同步

线程同步保证了多个线程在访问共享资源时的一致性和完整性。实际上,线程同步可以理解为“排队”:多个线程需要依次对共享数据进行处理,而并非同时处理。例如,如果两个线程同时修改一份数据,修改过程中的动作很有可能冲突重复,以致正确的修改被覆盖。为避免这种情况,需要采用同步机制来管理线程的访问顺序和互相排斥。

Java 提供了多种同步机制,包含 synchronized 关键字、显式锁(例如 ReentrantLock),以及更先进的并发工具类(如 CountDownLatch 和 CyclicBarrier 等)。

本文重点研究 synchronized 锁的工作原理。顾名思义,当一线程访问一资源时,synchronized 可以将资源锁住,阻止另一线程访问。当这一线程访问结束时,锁将被释放,另一线程获得锁,同理仍然禁止其他线程访问。总之,锁使资源具有了专一性。下述几种情况下,锁将被释放:synchronized 方法执行完成、synchronized 代码块执行完成,或者线程在执行过程中抛出异常。

在使用 synchronized 修饰一个方法时,它会锁死调用这一方法的对象实例,这样,这一对象的方法将具有专一性,只能由一个线程执行;但是如果 synchronized 修饰一个代码块,则会锁死括号内选定的对象,这种形式更为灵活。

1.同步方法

使用 synchronized 关键字定义方法,确保该方法的每时刻的专一性。被 synchronized 关键字修饰的一个实例方法,会在方法执行时被锁住当前实例对象(this),从而确保在同一时刻只有一个线程可以执行该实例方法。也就是说,只有一个线程可以访问该实例方法,其他线程需要等待该方法执行完毕并释放锁之后才能执行。

2.同步代码块

应用 synchronized 关键字在方法内部结构界定代码块,可以提供更细致的同步控制,仅对指定的代码片段进行锁死。synchronized 同步块是 Java 中实现线程同步的一种方式,它通过锁死某个对象来确保在多线程环境中对共享的安全访问。与 synchronized 修饰的方法不同,同步块能够灵活地操纵锁的粒度,仅对于需要同步的部分加锁,进而提升程序的性能。

3.线程同步实例

上文中,我们阐述了同步方法与同步代码块各自的含义。我们通过如下的实例,进一步说明二者的区别。

假设我们有一个线程安全的计数器类,需要确保 increment( ) 方法和 getCount( ) 方法在多线程环境中是线程安全的。

(1)同步方法(如图 4 所示):

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

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

    public synchronized int getCount() {
        return count;
    }
}
```

图 4 同步方法示例

(2)同步代码块(如图 5 所示):

```
public class Block {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        synchronized (this) {
            return count;
        }
    }
}
```

图 5 同步代码块示例

二、多线程的应用——售票程序

我们首先给出售票程序的代码如下:

```
public class Ticketseller2 extends Thread {
    //票
    private int ticket_number=100;

    //private Object lock = new Object();

    //继承 thread 类必须用 run 方法，买票
    public   void run(){
            while (true) {
                buy();
            }

        }
    private void buy(){
                //判断是否有票
                if (ticket_number <= 0){
                    return;
                }

            //买票
            System.out.println(Thread.currentThread().getName()+
            "正在出售第"+ticket_number--+"张票");

    }

    public static void main (String[] args){
        Ticketseller2 thread1 = new Ticketseller2();
        //创建 Ticketseller2 的线程实例
        Ticketseller2 thread2 = new Ticketseller2();
        Ticketseller2 thread3 = new Ticketseller2();
        thread1.start();
        thread2.start();
        thread3.start();//start 会自动调用 run 方法中的内容
    }
}
```

此代码结果如图 6 所示：

```
Thread-1正在出售第78张票
Thread-2正在出售第75张票
Thread-0正在出售第76张票
Thread-2正在出售第73张票
Thread-1正在出售第74张票
Thread-2正在出售第71张票
Thread-0正在出售第72张票
Thread-0正在出售第68张票
Thread-2正在出售第69张票
Thread-1正在出售第70张票
Thread-2正在出售第66张票
Thread-0正在出售第67张票
Thread-2正在出售第64张票
Thread-2正在出售第62张票
Thread-1正在出售第65张票
Thread-2正在出售第61张票
Thread-0正在出售第63张票
```

图 6　不安全代码运行结果

从结果中我们可以看到，所售票的票号无序，线程不能安全运行。三个线程并发访问票数据，我们必须对其进行排队操作并且上锁，Java 中我们通过 synchronized 关键字来实现。

(一)使用 synchronized 关键字

更改变量定义如下：

private static int ticket_number = 100；

使其资源为静态，即三个线程共享这一份资源。

更改方法定义如下：

private static synchronized void buy( ) { }

此时结果如图 7 所示：

```
Thread-2正在出售第13张票
Thread-2正在出售第12张票
Thread-2正在出售第11张票
Thread-2正在出售第10张票
Thread-2正在出售第9张票
Thread-2正在出售第8张票
Thread-2正在出售第7张票
Thread-2正在出售第6张票
Thread-2正在出售第5张票
Thread-2正在出售第4张票
Thread-2正在出售第3张票
Thread-2正在出售第2张票
Thread-2正在出售第1张票
```

图 7　安全代码运行结果

本段程序主要运用 synchronized 同步方法实现线程同步，确保程序运行的正确性。本例在 Ticketseller2 类中的 buy( ) 方法上使用 synchronized 关键字，synchronized 关键字确保每次只有一个线程能够执行该方法，即当多个线程同时尝试调用 buy( ) 方法时，只有一个线程能够成功获得锁并进入方法执行，其他线程会被阻塞，直到持有锁的线程完成执行并释放锁，即多个线程竞争获取 Ticketseller2 类实例的锁。

在 Java 中，每个对象都有一个关联的锁(即监视器锁)。当一个线程进入同步方法时，它必须首先获得该对象的锁。如果其他线程已经持有该锁，后续线程将会被阻塞，直到锁被释放。在 buy( ) 方法的执行过程中，线程完成了对票数的更新(即减少 ticket_number)后，方法执行完毕，线程会释放锁，其他被阻塞的线程可以重新竞争这个锁。这种获取和释放锁的机制是 synchronized 的核心，它通过确保同一时间只有一个线程能够执行同步方法(或代码块)，从而保证了共享资源的线程安全性。

(二)使用线程池

为确保线程安全，本研究已用 synchronized 关键字以同步对火车票库存的访问，确保在多线程环境下数据的一致性。同时，本研究通过 ExecutorSer-

vice 创建线程池,使用线程池管理售票请求以复用线程、提高效率。

关键实现代码如下：

```
public class TicketSaleSystem {
    private static int ticketsAvailable = 100; // 初始有 100 张票
    public static synchronized void sellTicket() {
        if (ticketsAvailable > 0) {
            ticketsAvailable--;
         System.out.println(Thread.currentThread().getName() +
          " sold a ticket. " + ticketsAvailable + " tickets left.");
        } else {
            System.out.println("No tickets available.");
        }
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);

        for (int i = 0; i < 200; i++) {
            executor.execute(TicketSaleSystem::sellTicket);
        }

        executor.shutdown();
    }
}
```

实验结果如图 8 所示：

```
pool-1-thread-3 sold a ticket. 13 tickets left.
pool-1-thread-3 sold a ticket. 12 tickets left.
pool-1-thread-2 sold a ticket. 11 tickets left.
pool-1-thread-3 sold a ticket. 10 tickets left.
pool-1-thread-4 sold a ticket. 9 tickets left.
pool-1-thread-5 sold a ticket. 8 tickets left.
pool-1-thread-9 sold a ticket. 7 tickets left.
pool-1-thread-7 sold a ticket. 6 tickets left.
pool-1-thread-8 sold a ticket. 5 tickets left.
pool-1-thread-1 sold a ticket. 4 tickets left.
pool-1-thread-10 sold a ticket. 3 tickets left.
pool-1-thread-6 sold a ticket. 2 tickets left.
pool-1-thread-10 sold a ticket. 1 tickets left.
pool-1-thread-10 sold a ticket. 0 tickets left.
No tickets available.
```

图 8　线程池优化代码运行结果

在用于处理售票逻辑的静态同步方法 sellTicket 中,首先检查 ticketsAvailable 是否大于 0。如果是,表示还有票可售,票数减 1,并打印出当前线程名称、已售出的票信息以及剩余票数。如果票数为 0 或小于 0,打印出“无票可售”的信息。同步的 sellTicket 方法保证了在任何时刻,只有一个线程能够修改 ticketsAvailable 变量,从而确保了售票过程的线程安全

本实验创建线程数为 10 的线程池,使用 10 线程一起执行特定功能。通过设置 200 次任务循环,每一次循环都将调用 sellTicket( ) 方法。在循环结束后,调用 shutdown( ) 方法强制关闭线程池。线程池将不再收到任务。

三、结语

线程同步策略是 Java 语言的显著特点,JVM 虚拟机正是通过线程同步机制来提升程序效率。本文以售票程序为例分析了 Java 多线程同步管程机制的内部逻辑。同步管程机制是多线程有序运行的保障,运用线程同步管程技术的程序能更好地描述和处理现实生活的实际问题,将成为应用程序开发和编程设计的必然结果。

参考文献：

[1]朱金波.Java 编程语言在计算机软件开发中的应用优势分析[J].信息记录材料,2023,24(5):68-70.

[2]郭沛.基于动态线程池的计算机联锁多用户并发访问方法[D].北京:北京交通大学,2022.

[3]鲁法明,郑佳静,包云霞,等.基于锁增广分段图的多线程程序死锁检测[J].软件学报,2021,32(6):1682-1700.

[4]简道红.多线程程序数据竞争静态检测方法研究[D].大连:大连理工大学,2013.

[5]丁宇新,程虎.Java 虚拟机用户级多线程的设计与实现[J].软件学报,2000(5):701-706.

[6]Brian Goetz, Tim Peierls, Joshua Bloch,等.Java 并发编程实战[M].童云兰,译.北京:机械工业出版社,2012.

[7]郑龙.多线程锁同步运行时特征分析与调优机制研究[D].武汉:华中科技大学,2016.

[8]柳晨光.面向多线程机制的软件重构方法研究与实现[D].石家庄:河北科技大学,2016.

Research on Java Multithread Synchronization Mechanism

WANG Ruo-han, ZHANG Xin-hao, LIU Yi, CHEN Yi
(Hubei University of Technology, Wuhan Hubei 430068, China)

Abstract: Multithread has widespread applications in daily life, so implementing multithread synchronization mechanisms to prevent conflicts between threads in a multithread model is of great significance. This paper introduces concepts related to multithread and thread synchronization, designs experiments based on the Java language, and focuses on analyzing two primary synchronization techniques: synchronized methods and synchronized code blocks, thereby elucidating the principles of Java multithread synchronization mechanisms. Corresponding application examples are provided to study the consequences of threads operating without synchronization. Based on the principle of multithread synchronization, ExecutorService is used to create thread pool, and improvements are made to unsafe application examples to verify the feasibility of the results. Finally, the trends in multithread technology development are briefly outlined.

Key words: Java; multithread; synchronization mechanism; thread safety