带你一次搞定 Java 多线程(III)

3 线程同步

3.1 线程同步机制简介

线程同步机制是一套用于协调线程之间的数据访问的机制.该机制可以保障线程安全.

Java 平台提供的线程同步机制包括：锁, volatile 关键字, final 关键字,static 关键字,以及相关的 API,如 Object.wait()/Object.notify()等

3.2 锁概述

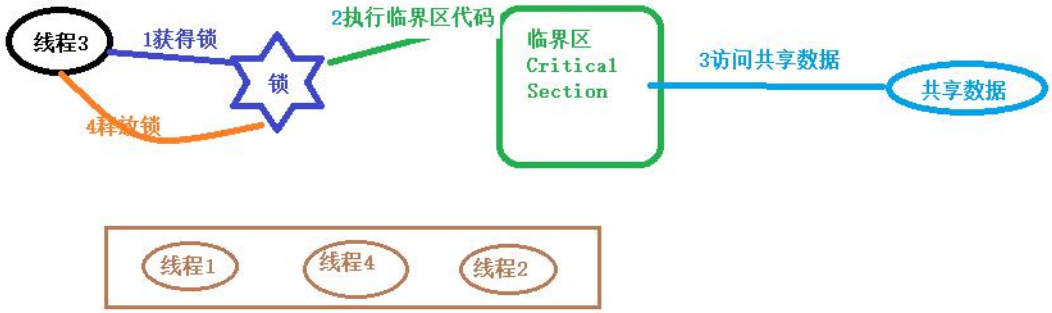
线程安全问题的产生前提是多个线程并发访问共享数据.

将多个线程对共享数据的并发访问转换为串行访问,即一个共享数据一次只能被一个线程访问.锁就是复用这种思路来保障线程安全的

锁(Lock)可以理解为对共享数据进行保护的一个许可证. 对于同一个许可证保护的共享数据来说,任何线程想要访问这些共享数据必须先持有该许可证. 一个线程只有在持有许可证的情况下才能对这些共享数据进行访问; 并且一个许可证一次只能被一个线程持有; 许可证线程在结束对共享数据的访问后必须释放其持有的许可证.

一线程在访问共享数据前必须先获得锁；获得锁的线程称为锁的持有线程； 一个锁一次只能被一个线程持有．锁的持有线程在获得锁之后 和释放锁之前这段时间所执行的代码称为临界区(Critical Section).

锁具有排他性(Exclusive)，即一个锁一次只能被一个线程持有.这种锁称为排它锁或互斥锁(Mutex).



JVM 把锁分为内部锁和显示锁两种．内部锁通过 synchronized 关键字实现；显示锁通过 java.concurrent.locks.Lock 接口的实现类实现的

3.2.1 锁的作用

锁可以实现对共享数据的安全访问．保障线程的原子性,可见性与有序性.

锁是通过互斥保障原子性．一个锁只能被一个线程持有，这就保证临界区的代码一次只能被一个线程执行.使得临界区代码所执行的操作自然而然的具有不可分割的特性,即具备了原子性.

可见性的保障是通过写线程冲刷处理器的缓存和读线程刷新处理器缓存这两个 动作实现的. 在 java 平台中,锁的获得隐含着刷新处理器缓存的动作, 锁的释放隐含着冲刷处理器缓存的动作.

锁能够保障有序性.写线程在临界区所执行的在读线程所执行的临界区看来像是完全按照源码顺序执行的.

注意:

使用锁保障线程的安全性,必须满足以下条件:

这些线程在访问共享数据时必须使用同一个锁

即使是读取共享数据的线程也需要使用同步锁

3.2.2 锁相关的概念

1)可重入性

可重入性(Reentrancy)描述这样一个问题: 一个线程持有该锁的时候能再次(多次)申请该锁

```
void     methodA(){
    申请 a 锁
        methodB();
        释放 a 锁
}
```

```
void    methodB(){
        申请 a 锁
        ....
        释放 a 锁
}
```

如果一个线程持有一个锁的时候还能够继续成功申请该锁,称该锁是可重入的，否则就称该锁为不可重入的

2)锁的争用与调度

Java 平台中内部锁属于非公平锁，显示 Lock 锁既支持公平锁又支持非公平锁

3)锁的粒度

一个锁可以保护的共享数据的数量大小称为锁的粒度.

锁保护共享数据量大,称该锁的粒度粗，否则就称该锁的粒度细.

锁的粒度过粗会导致线程在申请锁时会进行不必要的等待.锁的粒度过细会增加锁调度的开销.

3.3 内部锁:synchronized 关键字

Java 中的每个对象都有一个与之关联的内部锁(Intrinsic lock). 这种锁也称为监视器(Monitor)，这种内部锁是一种排他锁,可以保障原子性,可见性与有序性.

内部锁是通过 synchronized 关键字实现的.synchronized 关键字修饰代码块,修饰该方法.

修饰代码块的语法:

```
synchronized(  对象锁  ) {
    同步代码块,可以在同步代码块中访问共享数据
}
```

修饰实例方法就称为同步实例方法

修饰静态方法称称为同步静态方法

3.3.1 synchronized 同步代码块

```
package com.wkcto.intrinsiclock;

/**
 * synchronized 同步代码块
 *    this 锁对象
 * Author:  老崔
 */
public class Test01 {
    public static void main(String[] args) {
        //创建两个线程,分别调用 mm()方法
        //先创建 Test01 对象,通过对象名调用 mm()方法
        Test01 obj = new Test01();
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm();           //使用的锁对象 this 就是 obj 对象
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm();           //使用的锁对象 this 也是 obj 对象
            }
        }).start();

    }

    //定义方法,打印 100 行字符串
    public void mm(){
        synchronized ( this ) {        //经常使用 this 当前对象作为锁对象
            for (int i = 1; i <= 100; i++) {
```

```
System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
            }
        }
}
```

```
package com.wkcto.intrinsiclock;

/**
 * synchronized 同步代码块
 *    如果线程的锁不同，不能实现同步
 *    想要同步必须使用同一个锁对象
 * Author: 老崔
 */
public class Test02 {
    public static void main(String[] args) {
        //创建两个线程,分别调用 mm()方法
        //先创建 Test01 对象,通过对象名调用 mm()方法
        Test02 obj = new Test02();
        Test02 obj2 = new Test02();
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm();           //使用的锁对象 this 就是 obj 对象
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                obj2.mm();          //使用的锁对象 this 也是 obj2 对象
            }
        }).start();

    }

    //定义方法,打印 100 行字符串
    public void mm(){
        synchronized ( this ) {     //经常使用 this 当前对象作为锁对
```

```
象
                    for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
                    }
                }
            }
}
```

```
package com.wkcto.intrinsiclock;


/**
 * synchronized 同步代码块
 *  使用一个常量对象作为锁对象
 * Author:  老崔
 */
public class Test03 {
    public static void main(String[] args) {
        //创建两个线程,分别调用 mm()方法
        //先创建 Test01 对象,通过对象名调用 mm()方法
        Test03 obj = new Test03();
```

```
        Test03 obj2 = new Test03();


        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm();           //使用的锁对象 OBJ 常量
            }
        }).start();


        new Thread(new Runnable() {
            @Override
            public void run() {
                obj2.mm();          //使用的锁对象 OBJ 常量
            }
        }).start();


    }


    public static final    Object OBJ = new Object();        //定义一个常量,
    //定义方法,打印 100 行字符串
```

```
        public void mm(){
            synchronized ( OBJ ) {        //使用一个常量对象作为锁对象
                for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
            }
        }
}
```

```
package com.wkcto.intrinsiclock;


/**
 * synchronized 同步代码块
 *  使用一个常量对象作为锁对象,不同方法中 的同步代码块也可以同步
 * Author:  老崔
 */
public class Test04 {
        public static void main(String[] args) {
            //创建两个线程,分别调用 mm()方法
```

```
//先创建 Test01 对象,通过对象名调用 mm()方法
Test04 obj = new Test04();
Test04 obj2 = new Test04();


new Thread(new Runnable() {
    @Override
    public void run() {
        obj.mm();           //使用的锁对象 OBJ 常量
    }
}).start();


new Thread(new Runnable() {
    @Override
    public void run() {
        obj2.mm();          //使用的锁对象 OBJ 常量
    }
}).start();


//第三个线程调用静态方法
new Thread(new Runnable() {
    @Override
```

```
                public void run() {
                        sm();           //使用的锁对象 OBJ 常量
                }
        }).start();


    }

    public static final    Object OBJ = new Object();        //定义一个常量,
    //定义方法,打印 100 行字符串
    public void mm(){
        synchronized ( OBJ ) {        //使用一个常量对象作为锁对象
            for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
            }
        }
    }

    //定义方法,打印 100 行字符串
    public static void sm(){
```

```
            synchronized ( OBJ ) {        //使用一个常量对象作为锁对象
                for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
            }
        }
}
```

3.3.2 同步方法

```
package com.wkcto.intrinsiclock;


/**
 * synchronized 同步实例方法
 *        把整个方法体作为同步代码块
 *        默认的锁对象是 this 对象
 * Author: 老崔
 */
public class Test05 {
```

```
    public static void main(String[] args) {
        //先创建 Test01 对象,通过对象名调用 mm()方法
        Test05 obj = new Test05();

        //一个线程调用 mm()方法
        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm();           //使用的锁对象 this 就是 obj 对象
            }
        }).start();

        //另一个线程调用 mm22()方法
        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.mm22();         //使用的锁对象 this 也是 obj 对象，可以同步
//                  new Test05().mm22();        //使用的锁对象 this 是刚刚 new 创建的一个新对象,不是同一个锁对象不能同步
            }
```

```
        }).start();


    }

    //定义方法,打印 100 行字符串
    public void mm(){
        synchronized ( this ) {        //经常使用 this 当前对象作为锁对象
            for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
            }
        }
    }


    //使用 synchronized 修饰实例方法,同步实例方法, 默认 this 作为锁对象
    public synchronized void mm22(){
            for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
```

```
                }
        }
}
```

```
package com.wkcto.intrinsiclock;


/**
 * synchronized 同步静态方法
 *          把整个方法体作为同步代码块
 *          默认的锁对象是当前类的运行时类对象, Test06.class,  有人称它为类锁
 * Author:  老崔
 */
public class Test06 {
        public static void main(String[] args) {
                //先创建 Test01 对象,通过对象名调用 mm()方法
                Test06 obj = new Test06();


                //一个线程调用 m1()方法
                new Thread(new Runnable() {
                        @Override
```

```
            public void run() {
                obj.m1();           //使用的锁对象是 Test06.class
            }
        }).start();

        //另一个线程调用 sm2()方法
        new Thread(new Runnable() {
            @Override
            public void run() {
                Test06.sm2();     //使用的锁对象是 Test06.class
            }
        }).start();

    }

    //定义方法,打印 100 行字符串
    public void m1(){
        //使用当前类的运行时类对象作为锁对象,可以简单的理解为把 Test06 类的字节码文件作为锁对象
        synchronized ( Test06.class ) {
            for (int i = 1; i <= 100; i++) {
```

```
System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
            }
        }

        //使用 synchronized 修饰静态方法,同步静态方法，默认运行时类 Test06.class 作为锁对象
        public synchronized static    void sm2(){
                for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
        }
}
```

```
package com.wkcto.intrinsiclock;

/**
 * 同步方法与同步代码块如何选择
 *   同步方法锁的粒度粗，执行效率低，同步代码块执行效率高
```

```
 *
 * Author: 老崔
 */
public class Test07 {
    public static void main(String[] args) {
        Test07 obj = new Test07();

        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.doLongTimeTask();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                obj.doLongTimeTask();
            }
        }).start();
    }
```

```
    //同步方法，执行效率低
    public synchronized void doLongTimeTask(){
        try {
            System.out.println("Task Begin");
            Thread.sleep(3000);             //模拟任务需要准备 3 秒钟

                System.out.println("开始同步");
                for(int i = 1; i <= 100; i++){

System.out.println(Thread.currentThread().getName() + "-->" + i);
                }

            System.out.println("Task end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    //同步代码块,锁的粒度细，执行效率高
    public void doLongTimeTask2(){
        try {
            System.out.println("Task Begin");
            Thread.sleep(3000);             //模拟任务需要准备 3 秒钟

            synchronized (this){
                System.out.println("开始同步");
                for(int i = 1; i <= 100; i++){

System.out.println(Thread.currentThread().getName() + "-->" + i);
                }
            }
            System.out.println("Task end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3.3.3 脏读

```
package com.wkcto.intrinsiclock;


/**
 * 脏读
 *   出现读取属性值出现了一些意外，读取的是中间值,而不是修改之后 的值
 *   出现脏读的原因是 对共享数据的修改 与对共享数据的读取不同步
 *   解决方法:
 *         不仅对修改数据的代码块进行同步,还要对读取数据的代码块同步
 * Author: 老崔
 */
public class Test08 {
    public static void main(String[] args) throws InterruptedException {
        //开启子线程设置用户名和密码
        PublicValue publicValue = new PublicValue();
        SubThread t1 = new SubThread(publicValue);
```

```
        t1.start();


        //为了确定设置成功
        Thread.sleep(100);
        //在 main 线程中读取用户名,密码
        publicValue.getValue();
    }


    //定义线程,设置用户名和密码
    static class    SubThread extends    Thread{
        private PublicValue publicValue;
        public SubThread( PublicValue publicValue){
            this.publicValue = publicValue;
        }


        @Override
        public void run() {
            publicValue.setValue("bjpowernode", "123");
        }
    }
```

```
    static class PublicValue{
        private String name = "wkcto";
        private String pwd = "666";

        public synchronized void getValue(){
            System.out.println(Thread.currentThread().getName() + ",
getter -- name: " + name + ",--pwd: " + pwd);
        }

        public synchronized void setValue(String name, String pwd){
            this.name = name;
            try {
                Thread.sleep(1000);                    //模拟操作
name 属性需要一定时间
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            this.pwd = pwd;
            System.out.println(Thread.currentThread().getName() + ",
setter --name:" + name + ", --pwd: " + pwd );
        }
```

```
    }
}
```

3.3.4 线程出现异常会自动释放锁

```
package com.wkcto.intrinsiclock;


/**
 * 同步过程中线程出现异常，会自动释放锁对象
 *
 * Author: 老崔
 */
public class Test09 {
    public static void main(String[] args) {
        //先创建 Test01 对象,通过对象名调用 mm()方法
        Test09 obj = new Test09();


        //一个线程调用 m1()方法
        new Thread(new Runnable() {
            @Override
```

```
            public void run() {
                obj.m1();           //使用的锁对象是 Test06.class
            }
        }).start();

        //另一个线程调用 sm2()方法
        new Thread(new Runnable() {
            @Override
            public void run() {
                Test09.sm2();     //使用的锁对象是 Test06.class
            }
        }).start();

    }

    //定义方法,打印 100 行字符串
    public void m1(){
        //使用当前类的运行时类对象作为锁对象,可以简单的理解为把 Test06 类的字节码文件作为锁对象
        synchronized ( Test09.class ) {
            for (int i = 1; i <= 100; i++) {
```

```
System.out.println(Thread.currentThread().getName() + " --> " + i);
                    if ( i == 50){
                        Integer.parseInt("abc");       //把字符串转换为
int 类型时,如果字符串不符合 数字格式会产生异常
                    }
                }
            }
        }
        //使用 synchronized 修饰静态方法,同步静态方法, 默认运行时类
Test06.class 作为锁对象
        public synchronized static    void sm2(){
                for (int i = 1; i <= 100; i++) {

System.out.println(Thread.currentThread().getName() + " --> " + i);
                }
        }
}
```

3.3.5 死锁

```
package com.wkcto.intrinsiclock;

/**
 * 死锁
 *    在多线程程序中,同步时可能需要使用多个锁,如果获得锁的顺序不一致,可能会导致死锁
 *    如何避免死锁?
 *        当需要获得多个锁时,所有线程获得锁的顺序保持一致即可
 * Author: 老崔
 */
public class Test10 {
    public static void main(String[] args) {
        SubThread t1 = new SubThread();
        t1.setName("a");
        t1.start();

        SubThread t2 = new SubThread();
        t2.setName("b");
        t2.start();
```

```
    }

    static class    SubThread    extends Thread{
        private static final    Object lock1 = new Object();
        private static final    Object lock2 = new Object();

        @Override
        public void run() {
            if ("a".equals(Thread.currentThread().getName())){
                synchronized (lock1){
                    System.out.println("a 线程获得了 lock1 锁,还需要获得 lock2 锁");
                    synchronized (lock2){
                        System.out.println("a 线程获得 lock1 后又获得了 lock2,可以想干任何想干的事");
                    }
                }
            }
            if ("b".equals(Thread.currentThread().getName())){
                synchronized (lock2){
                    System.out.println("b 线程获得了 lock2 锁,还需
```

```
要获得 lock1 锁");
                                synchronized (lock1){
                                        System.out.println("b  线程获得 lock2 后又
获得了 lock1,可以想干任何想干的事");
                                }
                        }
                }
        }
    }
}
```

3.4 轻量级同步机制:volative 关键字

3.4.1 volatile 的作用

volatile 关键的作用使变量在多个线程之间可见.

```
package com.wkcto.volatilekw;

/**
 * volatile 的作用可以强制线程从公共内存中读取变量的值,而不是从工作内存中读取
 * Author:  老崔
 */
public class Test02 {
    public static void main(String[] args) {
```

```
        //创建 PrintString 对象
        PrintString printString = new PrintString();

        //开启子线程,让子线程执行 printString 对象的 printStringMethod()方法
        new    Thread(new Runnable() {
            @Override
            public void run() {
                printString.printStringMethod();
            }
        }).start();

        //main 线程睡眠 1000 毫秒
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("在 main 线程中修改打印标志");
        printString.setContinuePrint(false);
       //程序运行,查看在 main 线程中修改了打印标志之后 ,子线程打印是否可以结束打印

        //程序运行后，可能会出现死循环情况

        //分析原因:    main 线程修改了 printString 对象的打印标志后，子线程读不到

        //解决办法: 使用 volatile 关键字修饰 printString 对象的打印标志.

        //          volatile 的作用可以强制线程从公共内存中读取变量的值,而不是从工作内存中读取
    }

    //定义类打印字符串
    static class    PrintString{
```

```
        private volatile boolean continuePrint = true;

        public PrintString setContinuePrint(boolean continuePrint) {
            this.continuePrint = continuePrint;
            return this;
        }

        public void printStringMethod(){
            System.out.println(Thread.currentThread().getName() + "开始....");
            while ( continuePrint ){

            }
            System.out.println(Thread.currentThread().getName() + "结束+++++++++++++++");
        }
    }
}
```

volatile 与 synchronized 比较

1) volatile 关键字是线程同步的轻量级实现,所以 volatile 性能肯定比 synchronized 要好; volatile 只能修饰变量,而 synchronized 可以修饰方法,代码块. 随着 JDK 新版本的发布,synchronized 的执行效率也有较大的提升,在开发中使用 sychronized 的比率还是很大的.

2) 多线程访问 volatile 变量不会发生阻塞,而 synchronized 可能会阻塞

3) volatile 能保证数据的可见性,但是不能保证原子性; 而 synchronized 可以保证原子性,也可以保证可见性

4) 关键字 volatile 解决的是变量在多个线程之间的可见性;

synchronized 关键字解决多个线程之间访问公共资源的同步性.

3.4.2 volatile 非原子特性

volatile 关键字增加了实例变量在多个 线程之间的可见性,但是不具备原子性.

```
package com.wkcto.volatilekw;

/**
 * volatile 不是具备原子性

 * Author:  老崔
 */
public class Test03 {
    public static void main(String[] args) {
        //在 main 线程中创建 10 个子线程
        for (int i = 0; i < 100; i++) {
            new MyThread().start();
        }
    }

    static class MyThread extends Thread{
        //volatile 关键仅仅是表示所有线程从主内存读取 count 变量的值
        public static    int count;

    /*    //这段代码运行后不是线程安全的,想要线程安全,需要使用 synchronized 进行同步,如果使用 synchronized 同时,也就不需要 volatile 关键了
        public static void addCount(){
            for (int i = 0; i < 1000; i++) {
                //count++不是原子操作
                count++;
            }
            System.out.println(Thread.currentThread().getName() + " count=" + count);
```

```
        }*/

        public synchronized static void addCount(){
            for (int i = 0; i < 1000; i++) {
                //count++不是原子操作
                count++;
            }
            System.out.println(Thread.currentThread().getName() + " count=" + count);
        }

        @Override
        public void run() {
            addCount();
        }
    }
}
```

3.4.3 常用原子类进行自增自减操作

我们知道 i++操作不是原子操作，除了使用 Synchronized 进行同步外,也可以使用 AtomicInteger/AtomicLong 原子类进行实现

```
package com.wkcto.volatilekw;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * 使用原子类进行自增
 * Author: 老崔
 */
public class Test04 {
    public static void main(String[] args) throws InterruptedException {
        //在 main 线程中创建 10 个子线程
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();
```

```
            }
            Thread.sleep(1000);
            System.out.println( MyThread.count.get());
        }

        static class MyThread extends Thread{

            //使用 AtomicInteger 对象

            private static AtomicInteger count = new AtomicInteger();

            public    static void addCount(){
                for (int i = 0; i < 10000; i++) {

                    //自增的后缀形式

                    count.getAndIncrement();
                }
                System.out.println(Thread.currentThread().getName() + " count=" + count.get());
            }

            @Override
            public void run() {
                addCount();
            }
        }
}
```