文章编号:1673-9469(2011)02-0105-04

基于Java多线程同步的安全性研究

张勇
(宿州职业技术学院 计算机信息系,安徽 宿州 234101)

摘要:解决Java多线程同步的方法是在需要同步的方法签名中加入synchronized关键字,使用synchronized对需要进行同步的代码段进行同步,或使用JDK 5中提供的java.util.concurrent.lock包中的Lock对象。为解决多个线程对同一变量进行访问时可能发生的安全性问题,不仅可以采用同步机制,更可以通过JDK中加入的ThreadLocal来保证更好的并发性。

关键词:线程;同步;有序性;可见性;互斥
中图分类号:TP311.11 **文献标识码**:A

The discussion about Java multithreading synchronization securityand

ZHANG Yong
(Computer Information Department, Suzhou Vocational and Technological College, Anhui Suzhou 234101, China)

Abstract: The solution method of Java multi-thread synchronization is to add synchronization synchronized keyword. The synchronize code is synchronized by using synchronized, or the java.util.concurrent.lock package provided in JDK 5. In addition, the security issues, which perhaps occur during the multiple threads access the same variables, is solved by not only synchronization mechanism, but also adding ThreadLocal in JDK.

Key words: thread; synchronism; orderliness; visibility; exclusiveness

Java多线程是提高程序效能的利器,对于如何开发多线程的程序,已经有了很多的研究。本文并不是告诉您如何编写多线程Java程序,而着重于研究多线程的并发控制以及如何描述线程执行的过程,线程运行的机制,线程同步的必要性,和线程同步的解决方法。因为只有完全掌控Java多线程执行的过程,明白线程运行的机制,才能开发出高安全性的Java应用程序。

1 统一的Java内存模型规范

不同的平台,内存模型是不一样的,但是JVM的内存模型规范是统一的。其实Java的多线程并发问题最终都会反映在Java的内存模型上,所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改。其实Java的内存模型就是要解决两个主要的问题:可见性和有序性。我们都知道计算机有高速缓存的存在,处理器并不是每次处理数据都是取内存的。JVM定义了自己的内存模型,屏蔽了底层平台内存管理细节,对于Java开发人员,要清楚在JVM内存模型的基础上如何解决多线程的可见性和有序性[1]。

在JAVA程序的执行过程中,线程不能直接为主存中的字段赋值,它会将值指定给工作内存中的变量副本(assign),完成后这个变量副本会同步到主存储区(store-write),至于何时同步到主存,根据JVM实现系统决定。有些字段,则会从主内存中将该字段赋值到工作内存中,这个过程为read-load,完成后线程会引用该变量副本,当同一线程多次重复对字段赋值时,如

```
for(int i=0;i<10;i++)
  a++。
```

线程有可能只对工作内存中的副本进行赋值,直到最后一次赋值后才同步到主存储区,所以

收稿日期:2011-01-05
作者简介:张勇(1977-)男,安徽宿州人,硕士,讲师,从事软件工程、电子商务的研究。

assign,store,weite 顺序可以由 JVM 实现系统决定。

假设有一个共享变量 x,线程 A 执行 $x = x + 1$。从上面的描述中可以知道 $x = x + 1$ 并不是一个原子操作,它的执行过程如下:从主存中读取变量 x 副本到工作内存→给 x 加 1→将 x 加 1 后的值写回主存,如果另外一个线程 B 执行 $x = x - 1$,执行过程如下:从主存中读取变量 x 副本到工作内存→给 x 减 1→将 x 减 1 后的值写回主存。那么显然,最终的 x 的值是不可靠的。假设 x 现在为 10,线程 A 加 1,线程 B 减 1,从表面上看,似乎最终 x 还是为 10,但是多线程情况下会有这种情况发生:

1)线程 A 从主存读取 x 副本到工作内存,工作内存中 x 值为 10。

2)线程 B 从主存读取 x 副本到工作内存,工作内存中 x 值为 10。

3)线程 A 将工作内存中 x 加 1,工作内存中 x 值为 11。

4)线程 A 将 x 提交主存中,主存中 x 为 11。

5)线程 B 将工作内存中 x 值减 1,工作内存中 x 值为 9。

6)线程 B 将 x 提交到中主存中,主存中 x 为 9。

同样 x 有可能为 11,如果 x 是一个银行账户,线程 A 存款,线程 B 扣款,显然这样是有严重问题的,要解决这个问题,必须保证线程 A 和线程 B 是有序执行的,并且每个线程执行的加 1 或减 1 是一个原子操作。

2 Synchronized 关键字的使用

上面说了,Java 用 synchronized 关键字做为多线程并发环境的执行有序性的保证手段之一。当一段代码会修改共享变量,这一段代码成为互斥区或临界区,为了保证共享变量的正确性,synchronized 标示了临界区。典型的用法如下:

```
synchronized(锁){
临界区代码
}。
```

为了保证银行账户的安全,可以操作账户的方法如下:

```
public synchronized void add(int num) {
    balance = balance + num;
}
public synchronized void withdraw(int num) {
    balance = balance - num;
}。
```

那么对于 public synchronized void add(int num) 这种情况,意味着什么呢?其实这种情况,锁就是这个方法所在的对象。同理,如果方法是 public static synchronized void add(int num),那么锁就是这个方法所在的 class。理论上,每个对象都可以做为锁,但一个对象做为锁时,应该被多个线程共享,这样才显得有意义,在并发环境下,一个没有共享的对象作为锁是没有意义的。

每个锁对象都有两个队列,一个是就绪队列,一个是阻塞队列,就绪队列存储了将要获得锁的线程,阻塞队列存储了被阻塞的线程,当一个线程被唤醒(notify)后,才会进入到就绪队列,等待 cpu 的调度。当一开始线程 A 第一次执行 account.add 方法时,JVM 会检查锁对象 account 的就绪队列是否已经有线程在等待,如果有则表明 account 的锁已经被占用了,由于是第一次运行,account 的就绪队列为空,所以线程 A 获得了锁,执行 account.add 方法。如果恰好在这个时候,线程 b 要执行 account.withdraw 方法,因为线程 A 已经获得了锁还没有释放,所以线程 B 要进入 account 的就绪队列,等到得到锁后才可以执行。

一个线程执行临界区代码过程如下:获得同步锁→清空工作内存→从主存拷贝变量副本到工作内存→对这些变量计算→将变量从工作内存写回到主存→释放锁,可见,synchronized 既保证了多线程的并发有序性,又保证了多线程的内存可见性[3]。

3 模式问题的解决

生产者/消费者模式其实是一种很经典的线程同步模型,很多时候,并不是光保证多个线程对某共享资源操作的互斥性就够了,往往多个线程之间都是有协作的。

假设有这样一种情况,有一个桌子,桌子上面有一个盘子,盘子里只能放一颗鸡蛋,A 专门往盘子里放鸡蛋,如果盘子里有鸡蛋,则一直等到盘子里没鸡蛋,B 专门从盘子里拿鸡蛋,如果盘子里没鸡蛋,则等待直到盘子里有鸡蛋。其实盘子就是一个互斥区,每次往盘子放鸡蛋应该都是互斥的,A 的等待其实就是主动放弃锁,B 等待时还要提醒 A 放鸡蛋。

如何让线程主动释放锁,很简单,调用锁的 wait()方法就好。wait()方法是从 Object 来的,所以任意对象都有这个方法。

```
Object lock = new Object();//声明了一个对象作为锁
    synchronized (lock) {
      balance = balance - num; //这里放弃了同步锁,好不容易得到,又放弃了
      lock.wait();
    }
```

如果一个线程获得了锁 lock,进入了同步块,执行 lock.wait(),那么这个线程会进入到 lock 的阻塞队列。如果调用 lock.notify()则会通知阻塞队列的某个线程进入就绪队列。

声明一个盘子,只能放一个鸡蛋。

```
package com.jameswxx.synctest;
public class Plate{
    List < Object >  eggs = new ArrayList < Object >();
    public synchronized Object getEgg(){
    if(eggs.size() = =0){
      try{
        wait();
      }catch(InterruptedException e){
      }
    }
    Object egg = eggs.get(0);
    eggs.clear();//清空盘子
    notify();//唤醒阻塞队列的某线程到就绪队列
    return egg;
    }
    public synchronized void putEgg(Object egg){
    If(eggs.size() > 0){
      try{
        wait();
      }catch(InterruptedException e){
      }
    }
    eggs.add(egg);//往盘子里放鸡蛋
    notify();//唤醒阻塞队列的某线程到就绪队列
    }
    }
```

声明一个 Plate 对象为 plate,被线程 *A* 和线程 *B* 共享,*A* 专门放鸡蛋,*B* 专门拿鸡蛋。

假设(1)开始,*A* 调用 plate.putEgg 方法,此时 eggs.size()为 0,因此顺利将鸡蛋放到盘子,还执行了 notify()方法,唤醒锁的阻塞队列的线程,此时阻塞队列还没有线程。(2)又有一个 *A* 线程对象调用 plate.putEgg 方法,此时 eggs.size()不为 0,调用 wait()方法,自己进入了锁对象的阻塞队列。(3)此时,来了一个 *B* 线程对象,调用 plate.getEgg 方法,eggs.size()不为 0,顺利的拿到了一个鸡蛋,还执行了 notify()方法,唤醒锁的阻塞队列的线程,此时阻塞队列有一个 *A* 线程对象,唤醒后,它进入到就绪队列,就绪队列也就它一个,因此马上得到锁,开始往盘子里放鸡蛋,此时盘子是空的,因此放鸡蛋成功。(4)假设接着来了线程 *A*,就重复 2;假设来料线程 *B*,就重复 3。整个过程都保证了放鸡蛋,拿鸡蛋,放鸡蛋,拿鸡蛋。

4 volatile 关键字的使用场景

volatile 是 Java 提供的一种同步手段,只不过它是轻量级的同步,为什么这么说,因为 volatile 只能保证多线程的内存可见性,不能保证多线程的执行有序性。而最彻底的同步要保证有序性和可见性,例如 synchronized。任何被 volatile 修饰的变量,都不拷贝副本到工作内存,任何修改都及时写在主存。因此对于 valatile 修饰的变量的修改,所有线程马上就能看到,但是 volatile 不能保证对变量的修改是有序的[4]。假如:

```
public class VolatileTest{
public volatile int a;
public void add(int count){
a = a + count;
}
}。
```

当一个 VolatileTest 对象被多个线程共享,*a* 的值不一定是正确的,因为 *a* = *a* + count 包含了好几步操作,而此时多个线程的执行是无序的,因为没有任何机制来保证多个线程的执行有序性和原子性。volatile 存在的意义是,任何线程对 *a* 的修改,都会马上被其他线程读取到,因为直接操作主存,没有线程对工作内存和主存的同步。所以,volatile 的使用场景是有限的,在有限的一些情形下可以使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全,必须同时满足下面两个条件[5]。

(1)对变量的写操作不依赖于当前值。

(2)该变量没有包含在具有其他变量的不变式中。

volatile 只保证了可见性,所以 Volatile 适合直接赋值的场景,如:

```
public class VolatileTest{
public volatile int a;
public void setA(int a){
this.a = a;
}
}。
```

在没有 volatile 声明时,多线程环境下,a 的最终值不一定是正确的,因为 this.a = a;涉及到给 a 赋值和将 a 同步回主存的步骤,这个顺序可能被打乱。如果用 volatile 声明了,读取主存副本到工作内存和同步 a 到主存的步骤,相当于是一个原子操作。所以简单来说,volatile 适合这种场景:一个变量被多个线程共享,线程直接给这个变量赋值。这是一种很简单的同步场景,这时候使用 volatile 的开销将会非常小。

5 结束语

使用 synchronized 关键字、volatile 关键字可以为多线程的同步提供基本的安全保障,在开发高安全性的 Java 程序时,为了防止竞争冒险、死锁、活动锁和资源耗等情况的发生,我们必须对线程的等待机制、资源占有机制等作详细的研究与规划,不仅要在线程的运行机制上认真探索,还要在程序的整体构建上作合理的部署,这也是在以后的研究中对这一类问题的从微观到宏观的一个研究转变。

参考文献:

[1] 吴其庆.Java 编程思想与实践[M].北京:冶金工业出版社, 2002.

[2] 包景州.Web 服务中安全身份认证系统的设计和研究[D].上海: 上海交通大学, 2004.

[3] 李尊朝, 苏 军.Java 语言程序设计[M].北京: 中国铁道出版社, 2004.

[4] 沈 袁.实时 Java 平台的研究[D].无锡: 江南大学, 2009.

[5] 金振乾.Java 语言中 read 方法分析[J].科技信息, 2010(27):71.

(责任编辑 刘存英)