第二章 线程同步基础 Java 7并发编程实战手册读书笔记

第二章 线程同步基础

1、临界区（Critical Section）

临界区是一段供线程独占式访问的代码，也就是说若有一线程正在访问该代码段，其它线程想要访问，只能等待当前线程离开该代码段方可进入，这样保证了线程安全。

2、Java的同步机制

为了帮助编程人员实现临界区，Java提供了同步机制，当一个线程试图访问一个临界区时，它将提供一中同步机制来查看是不是有其他线程进入了临界区，如果没有其他线程进入了临界区，它就可以进入临界区，如果已经有线程进入了临界区，它就被同步机制挂起，直到进入的线程离开这个临界区，如果等待进入临界区的线程不止一个，JVM会选择其中一个，其余的继续等待。

Java语言的两种基本同步机制

(1)synchronized关键字

(2)Lock接口及其实现机制

3、synchronized的用法

a.synchronized是Java中的关键字，是一种同步锁，被synchronized修饰的地方，一次只允许一个线程访问。它修饰的对象有以下几种：

1)修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；

2)修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；

3)修饰一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；

4)修饰一个类，其作用的范围是synchronized后面括号括起来的部分，作用主要对象是这个类的所有对象。

注意：synchronized修饰静态的方法，同时只能被一个线程访问，但是其他线程可以访问这个对象的非静态方法。必须谨慎这一点，因为两个线程可以同时访问一个对象的不同synchronized方法，其中一个是静态方法，另一个是非静态方法。如果这两个访问改变了相同的数据，将会出现数据不一致的现象。

下面代码是模拟银行取钱存钱的关键代码，存钱和取钱的方法都用synchronized 修饰了，保证一次只允许一个线程访问。

```
public synchronized void addAmount(double amount) {
  double tmp=balance;
  try {
   Thread.sleep(10);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  tmp+=amount;
  balance=tmp;
 }

 public synchronized void subtractAmount(double amount) {
  double tmp=balance;
  try {
   Thread.sleep(10);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  tmp-=amount;
  balance=tmp;
 }
```

b. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。

c. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。

d. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

4、使用非依赖属性实现同步

当使用synchronized关键字保护**代码块**时，必须把对象引用传入参数，通常用this来引用执行方法所属的对象，即synchronized(this)，例如下面的代码：

```
public void method1() {

    //同步代码块
```

```
        synchronized(this) {
            int i = 5;
            while( i-- > 0) {
                System.out.println(Thread.currentThread().getName() + " : " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException ie) {
                }
            }
        }
}
```

但是，如果类中有两个非依赖属性，他们被多个线程共享，但是同一时刻只允许一个线程访问一个属性，其他某个线程访问另一个属性。此时可以不用this来引用。如下Demo：

```
public class Cinema {

 private long vacanciesCinema1;
 private long vacanciesCinema2;
 private final Object controlCinema1, controlCinema2;
 public Cinema(){
  controlCinema1=new Object();
  controlCinema2=new Object();
  vacanciesCinema1=20;
  vacanciesCinema2=20;
 }
 public boolean sellTickets1 (int number) {
  synchronized (controlCinema1) {
   if (number<vacanciesCinema1) {
    vacanciesCinema1-=number;
    return true;
   } else {
    return false;
   }
  }
 }
 public boolean sellTickets2 (int number){
  synchronized (controlCinema2) {
   if (number<vacanciesCinema2) {
    vacanciesCinema2-=number;
    return true;
   } else {
    return false;
   }
  }
 }
 public boolean returnTickets1 (int number) {
  synchronized (controlCinema1) {
   vacanciesCinema1+=number;
   return true;
  }
 }
 public boolean returnTickets2 (int number) {
  synchronized (controlCinema2) {
   vacanciesCinema2+=number;
   return true;
  }
 }
 public long getVacanciesCinema1() {
  return vacanciesCinema1;
 }
 public long getVacanciesCinema2() {
  return vacanciesCinema2;
 }
```

```
}

public class TicketOffice1 implements Runnable {

 private Cinema cinema;
 public TicketOffice1 (Cinema cinema) {
  this.cinema=cinema;
 }
 @Override
 public void run() {
  cinema.sellTickets1(3);
  cinema.sellTickets1(2);
  cinema.sellTickets2(2);
  cinema.returnTickets1(3);
  cinema.sellTickets1(5);
  cinema.sellTickets2(2);
  cinema.sellTickets2(2);
  cinema.sellTickets2(2);
 }
}

public class TicketOffice2 implements Runnable {

 private Cinema cinema;
 public TicketOffice2(Cinema cinema){
  this.cinema=cinema;
 }
 @Override
 public void run() {
  cinema.sellTickets2(2);
  cinema.sellTickets2(4);
  cinema.sellTickets1(2);
  cinema.sellTickets1(1);
  cinema.returnTickets2(2);
  cinema.sellTickets1(3);
  cinema.sellTickets2(2);
  cinema.sellTickets1(2);
 }
}

public class Main {

 public static void main(String[] args) {
  Cinema cinema=new Cinema();
  TicketOffice1 ticketOffice1=new TicketOffice1(cinema);
  Thread thread1=new Thread(ticketOffice1,"TicketOffice1");

  TicketOffice2 ticketOffice2=new TicketOffice2(cinema);
  Thread thread2=new Thread(ticketOffice2,"TicketOffice2");

  thread1.start();
  thread2.start();

  try {
   thread1.join();
   thread2.join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  System.out.printf("Room 1 Vacancies: %d\n",cinema.getVacanciesCinema1());
  System.out.printf("Room 2 Vacancies: %d\n",cinema.getVacanciesCinema2());
 }
}
```

结果

```
<terminated> Main (15) [Java Application] D:\Java\jre\bin\jav
Room 1 Vacancies: 5
Room 2 Vacancies: 6
```

5、在同步代码中使用条件

a.wait()、notify()、notifyAll()是三个定义在Object类里的方法，可以用来控制线程的状态，当同步代码中需要使用条件时，可以使用这几个方法配合完成某些功能。比如并发编程里面典型的例子--生成者、消费者模型。如下Demo：

b. 如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。

如果对象调用了notify方法就会通知某个正在等待这个对象的控制权的线程可以继续运行。

如果对象调用了notifyAll方法就会通知所有等待这个对象控制权的线程继续运行。

c.wait、notify、notifyAll必须在同步方法或块中，否则会抛出异常。

d.obj.notify()会随机唤醒一个等待在obj上的线程，而obj.notifyAll()却会唤醒所有等待在obj上的线程。

```
public class EventStorage {

 private List<Date> storage;
 public EventStorage(){
  maxSize=10;
  storage=new LinkedList<>();
 }
 public synchronized void set(){
   while (storage.size()==maxSize){
    try {
     wait();
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
   storage.add(new Date());
   System.out.printf("Set: %d",storage.size());
   System.out.println();
   notify();
 }
 public synchronized void get(){
   while (storage.size()==0){
    try {
     wait();
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
   System.out.printf("Get: %d: %s",storage.size(),((LinkedList<?>)storage).poll());
   System.out.println();
   notify();
 }
}

public class Producer implements Runnable {
 private EventStorage storage;
 public Producer(EventStorage storage){
  this.storage=storage;
 }
 @Override
 public void run() {
  for (int i=0; i<100; i++){
   storage.set();
  }
 }
}

public class Consumer implements Runnable {
 private EventStorage storage;
 public Consumer(EventStorage storage){
  this.storage=storage;
```

```
 }
 @Override
 public void run() {
  for (int i=0; i<100; i++){
   storage.get();
  }
 }
}
public class Main {

 public static void main(String[] args) {

  EventStorage storage=new EventStorage();
  Producer producer=new Producer(storage);
  Thread thread1=new Thread(producer);

  Consumer consumer=new Consumer(storage);
  Thread thread2=new Thread(consumer);

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

结果

```
<terminated> Main (16) [Java Application] D:\Java\jre\bin\javaw.e
Set: 1
Set: 2
Set: 3
Set: 4
Set: 5
Set: 6
Set: 7
Set: 8
Set: 9
Set: 10
Get: 10: Sun Nov 20 12:58:45 CST 2016
Get: 9: Sun Nov 20 12:58:45 CST 2016
Get: 8: Sun Nov 20 12:58:45 CST 2016
Get: 7: Sun Nov 20 12:58:45 CST 2016
Get: 6: Sun Nov 20 12:58:45 CST 2016
Get: 5: Sun Nov 20 12:58:45 CST 2016
Get: 4: Sun Nov 20 12:58:45 CST 2016
Get: 3: Sun Nov 20 12:58:45 CST 2016
Get: 2: Sun Nov 20 12:58:45 CST 2016
Get: 1: Sun Nov 20 12:58:45 CST 2016
Set: 1
Set: 2
Set: 3
Set: 4
Set: 5
Set: 6
Set: 7
Set: 8
Set: 9
Set: 10
Get: 10: Sun Nov 20 12:58:45 CST 2016
Get: 9: Sun Nov 20 12:58:45 CST 2016
Get: 8: Sun Nov 20 12:58:45 CST 2016
Get: 7: Sun Nov 20 12:58:45 CST 2016
```

6、使用锁实现同步

a.在jdk1.5之后，并发包中新增了Lock接口(以及相关实现类)用来实现锁功能

Lock接口提供了与synchronized关键字类似的同步功能，但**需要在使用时手动获取锁和释放锁**。虽然Lock接口没有synchronized关键字自动获取和释放锁那么便捷，但Lock接口却具有了锁的可操作性，可中断获取以及超时获取锁等多种非常实用的同步特性。

b.Lock接口与synchronized关键字的区别

1）Lock接口可以尝试非阻塞地获取锁 当前线程尝试获取锁。如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁。

2）Lock接口能被中断地获取锁 与synchronized不同，获取到锁的线程能够响应中断，当获取到的锁的线程被中断时，中断异常将会被抛出，同时锁会被释放。

3）Lock接口在指定的截止时间之前获取锁，如果截止时间到了依旧无法获取锁，则返回。

c.Lock接口的API

1）void lock() 获取锁,调用该方法当前线程将会获取锁，当锁获取后，该方法将返回。

2）void lockInterruptibly() throws InterruptedException 可中断获取锁，与lock()方法不同之处在于该方法会响应中断，即在锁的获取过程中可以中断当前线程

3) boolean tryLock() 尝试非阻塞的获取锁，调用该方法立即返回，true表示获取到锁（**意思是，不用像synchronized那样，阻塞其他线程**）
4) boolean tryLock(long time,TimeUnit unit) throws InterruptedException 超时获取锁，以下情况会返回：时间内获取到了锁，时间内被中断，时间到了没有获取到锁。
void unlock() 释放锁

```
public class PrintQueue {

 private final Lock queueLock=new ReentrantLock();
 public void printJob(Object document){
  queueLock.lock();
  try {
   Long duration=(long)(Math.random()*10000);
   System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
   Thread.sleep(duration);
  } catch (InterruptedException e) {
   e.printStackTrace();
  } finally {
   queueLock.unlock();
  }
 }
}
public class Job implements Runnable {

 private PrintQueue printQueue;
 public Job(PrintQueue printQueue){
  this.printQueue=printQueue;
 }
 @Override
 public void run() {
  System.out.printf("%s: Going to print a document\n",Thread.currentThread().getName());
  printQueue.printJob(new Object());
  System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
 }
}

public class Main {
 public static void main (String args[]){
  PrintQueue printQueue=new PrintQueue();
  Thread thread[]=new Thread[10];
  for (int i=0; i<10; i++){
   thread[i]=new Thread(new Job(printQueue),"Thread "+i);
  }
  for (int i=0; i<10; i++){
   thread[i].start();
  }
 }
}
```

结果

```
<terminated> Main (17) [Java Application] D:\Java\jre\bin\javaw.exe (2016年
Thread 1: Going to print a document
Thread 9: Going to print a document
Thread 0: Going to print a document
Thread 9: PrintQueue: Printing a Job during 7 seconds
Thread 7: Going to print a document
Thread 5: Going to print a document
Thread 3: Going to print a document
Thread 8: Going to print a document
Thread 6: Going to print a document
Thread 4: Going to print a document
Thread 2: Going to print a document
Thread 0: PrintQueue: Printing a Job during 0 seconds
Thread 9: The document has been printed
Thread 0: The document has been printed
Thread 7: PrintQueue: Printing a Job during 3 seconds
Thread 7: The document has been printed
Thread 5: PrintQueue: Printing a Job during 6 seconds
Thread 5: The document has been printed
Thread 3: PrintQueue: Printing a Job during 6 seconds
Thread 3: The document has been printed
Thread 1: PrintQueue: Printing a Job during 0 seconds
Thread 8: PrintQueue: Printing a Job during 1 seconds
Thread 1: The document has been printed
Thread 6: PrintQueue: Printing a Job during 7 seconds
Thread 8: The document has been printed
Thread 6: The document has been printed
Thread 4: PrintQueue: Printing a Job during 6 seconds
Thread 2: PrintQueue: Printing a Job during 6 seconds
Thread 4: The document has been printed
Thread 2: The document has been printed
```

7、使用读写锁实现同步数据访问

a.读写锁接口ReadWriteLock和它的实现类ReentrantReadWriteLock读写锁

前面提到的ReentrantLock是排他锁，该锁在同一时刻只允许一个线程来访问，**而读写锁在同一时刻允许可以有多个线程来访问，但在写线程访问时，所有的读线程和其他写线程被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过读写锁分离，使得并发性相比一般的排他锁有了很大的提升**

```
public class PricesInfo {

 private double price1;
 private double price2;

 private ReadWriteLock lock;
 public PricesInfo(){
  price1=1.0;
  price2=2.0;
  lock=new ReentrantReadWriteLock();
 }
 public double getPrice1() {
  lock.readLock().lock();
  double value=price1;
  lock.readLock().unlock();
  return value;
 }
 public double getPrice2() {
  lock.readLock().lock();
  double value=price2;
  lock.readLock().unlock();
  return value;
 }
 public void setPrices(double price1, double price2) {
  lock.writeLock().lock();
  this.price1=price1;
  this.price2=price2;
  lock.writeLock().unlock();
 }
}

public class Reader implements Runnable {

 private PricesInfo pricesInfo;
 public Reader (PricesInfo pricesInfo){
```

```
  this.pricesInfo=pricesInfo;
 }
 @Override
 public void run() {
  for (int i=0; i<10; i++){
   System.out.printf("%s: Price 1: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice1());
   System.out.printf("%s: Price 2: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice2());
  }
 }
}
public class Writer implements Runnable {

 private PricesInfo pricesInfo;
 public Writer(PricesInfo pricesInfo){
  this.pricesInfo=pricesInfo;
 }

 @Override
 public void run() {
  for (int i=0; i<3; i++) {
   System.out.printf("Writer: Attempt to modify the prices.\n");
   pricesInfo.setPrices(Math.random()*10, Math.random()*8);
   System.out.printf("Writer: Prices have been modified.\n");
   try {
    Thread.sleep(2);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
 }
}
public class Main {

 public static void main(String[] args) {

  PricesInfo pricesInfo=new PricesInfo();
  Reader readers[]=new Reader[5];
  Thread threadsReader[]=new Thread[5];
  for (int i=0; i<5; i++){
   readers[i]=new Reader(pricesInfo);
   threadsReader[i]=new Thread(readers[i]);
  }

  Writer writer=new Writer(pricesInfo);
  Thread threadWriter=new Thread(writer);
  for (int i=0; i<5; i++){
   threadsReader[i].start();
  }
  threadWriter.start();
 }
}
```

结果：可以看到 Reader**类的10次循环里面可以允许有多个线程同时存在，而**Writer**类的三次循环则只允许一个线程存在。**

```
<terminated> Main (18) [Java Application] D:\Java\jre\bin\ja
Thread-4: Price 1: 7.760077
Thread-4: Price 2: 0.514121
Thread-4: Price 1: 7.760077
Thread-4: Price 2: 0.514121
Thread-4: Price 1: 7.760077
Thread-4: Price 2: 0.514121
Thread-4: Price 1: 7.760077
Thread-4: Price 2: 0.514121
Thread-2: Price 1: 7.760077
Writer: Prices have been modified.
Thread-0: Price 2: 2.000000
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-0: Price 1: 7.760077
Thread-0: Price 2: 0.514121
Thread-2: Price 2: 0.514121
Thread-2: Price 1: 7.760077
```

8、修改锁的公平性

a.ReentrantLock和ReentrantReadWriteLock类的构造器都有一个布尔参数fair，默认值是false，即非公平模式，在非公平模式下，当很多线程等待锁时，锁将选择其中一个来访问临界区，这种选择没有约束。在公平模式下，当很多线程等待锁时，锁会选择一个等待时间最长的线程获得锁。

注意：这种公平性问题只适用于lock()和unlock()方法，tryLock()方法因为没有将线程置于休眠，所以，fair属性并不影响这个方法

b.新建公平锁方法：private final Lock queueLock=new ReentrantLock(false);

9.在锁中使用多条件

//有待完善，没有看懂例子