7362

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Ф. УТКИНА

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА JAVA. БЛОКИРОВКИ

Методические указания к лабораторной работе

Рязань 2022

УДК 681.3

Многопоточное программирование на Java. Блокировки: методические указания к лабораторной работе / Рязан. гос. радиотехн. ун-т.; сост.: А.А. Митрошин, В.Г. Псоянц. Рязань, 2022. 16 с.

Содержат описание лабораторной работы, используемой в курсе «Параллельное программирование».

Предназначены для студентов всех форм обучения направления подготовки «Информатика и вычислительная техника».

Табл. 6. Библиогр.: 3 назв.

Java, многопоточное программирование, блокировки

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета им. В.Ф Уткина.

Рецензент: кафедра САПР вычислительных средств Рязанского государственного радиотехнического университета (зав. кафедрой засл. деят. науки и техники РФ В.П. Корячко)

Блокировки

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized могут использоваться блокировки. Блокировки описаны в пакете java.util.concurrent.locks.

Общая схема использования блокировок такова. Вначале поток пытается получить доступ к общему ресурсу. Если ресурс свободен, то он блокируется. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен (заблокирован), то поток ожидает, пока эта блокировка не будет снята.

Основные преимущества или улучшения по сравнению с неявной синхронизацией:

- разделение блокировок на блокировки чтения и записи;
- некоторые блокировки разрешают одновременный доступ к общему ресурсу (например, блокировка ReadWriteLock).

Различные способы получения блокировки:

- блокировка - lock();
- неблокирующая попытка получения блокировки - tryLock();
- прерываемая блокировка: lockInterruptibly().

Объекты блокировки реализуют один из интерфейсов: Lock или ReadWriteLock.

Интерфейс *Lock*

```
public interface Lock
```

Интерфейс Lock определяет основные функции, которые должен реализовать объект блокировки. В отличие от неявных блокировок позволяет получить блокировку неблокирующим или прерываемым способом (в дополнение к блокирующему способу). Методы интерфейса Lock приведены в табл. 1.

Таблица 1. Методы интерфейса Lock

Модификатор и тип	Метод и его описание
`void`	`lock()` Получает блокировку, если она свободна. Если блокировка занята, то текущий поток ожидает ее получения
`void`	`lockInterruptibly()` Получает блокировку, если текущий поток не прерван. Если блокировка доступна, то текущий поток получает ее и немедленно продолжает выполняться. Если блокировка недоступна, то поток ожидает до тех пор, пока не произойдет одно из двух

Окончание таблицы 1

	возможных событий: - текущий поток получает блокировку после того, как она освободилась; - некоторый другой поток прервет текущий поток и поддерживается прерывание получения блокировки. Если текущий поток: - имеет статус прерванного при вхождении в этот метод; - прерван, пока получал блокировку, и поддерживается прерывание получения блокировки. Метод выбрасывает исключение InterruptedException, когда статус прерванного у текущего потока сброшен
`Condition`	`newCondition()` Возвращает новый объект условия (Condition), ограниченный объектом блокировки. До получения условия блокировка должна удерживаться текущим потоком. Вызов Condition.await() будет автоматически снимать блокировку до ожидания и повторно блокировать до возврата из ожидания
`boolean`	`tryLock()` Получает блокировку, если она была свободна во время вызова метода, и возвращает true. Если блокировка недоступна, то возвращает false и немедленно продолжает выполняться
`boolean`	`tryLock(long time, TimeUnit unit)` Получает блокировку, если она свободна в течение заданного интервала времени и текущий поток не был прерван. Параметры: time – максимальное время ожидания получения блокировки; unit – единица измерения времени аргумента time. Возвращает: true, если блокировка получена, и false, если время ожидания получения блокировки истекло. Выбрасывает исключения: InterruptedException – если текущий поток прерван во время получения блокировки (и прерывание получения блокировки поддерживается)
`void`	`unlock()` Освобождает блокировку

Основные реализации интерфейса Lock: ReentrantLock, ReadLock (используется ReentrantReadWriteLock), WriteLock (используется ReentrantReadWriteLock).

Интерфейс ReadWriteLock

```
public interface ReadWriteLock
```

Интерфейс ReadWriteLock определяет пару блокировок, одну для операций только для чтения и другую для записи. Блокировка чтения может быть получена одновременно различными потоками считывателя (если ресурс еще не получен блокировкой записи), в то время как блокировка записи является исключительной. Таким образом, мы можем иметь несколько потоков, считывающих ресурс одновременно, если нет операции записи.

Методы интерфейса Lock приведены в табл. 2.

Таблица 2. Методы интерфейса ReadWriteLock

Модификатор и тип	Метод и его описание
`Lock`	`readLock()` Возвращает блокировку для чтения
`Lock`	`writeLock()` Возвращает блокировку для записи

Основная реализация: ReentrantReadWriteLock.

Интерфейс Condition

```
public interface Condition
```

Применение условий в блокировках позволяет добиться контроля над управлением доступом к потокам. *Условие блокировки* представлет собой объект интерфейса Condition.

Применение объектов Condition во многом аналогично использованию методов `wait/notify/notifyAll` класса Object, в частности, доступно использовать следующие методы интерфейса Condition:

- await - поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы `signal/signalAll`. Во многом аналогичен методу `wait` класса Object;

- signal - сигнализирует, что поток, у которого ранее был вызван метод `await()`, может продолжить работу. Применение аналогично использованию методу `notify` класса Object;

- signalAll - сигнализирует всем потокам, у которых ранее был вызван метод `await()`, что они могут продолжить работу. Аналогичен методу `notifyAll()` класса Object.

Классы блокировок

Класс *ReentrantLock*

```
public class ReentrantLock extends Object implements Lock, Serializable
```

Класс *ReentrantLock* реализует ту же семантику параллелизма, что и неявная блокировка монитора, доступ к которой осуществляется с помощью *synchronized* методов и операторов с расширенными возможностями.

Конструкторы *ReentrantLock*

`ReentrantLock()` – создает блокировку типа ReentrantLock.

`ReentrantLock(Boolean fair)` - создает блокировку типа ReentrantLock с заданной политикой «справедливости» (если fair=true, то потоки будут получать блокировку в той последовательности, в которой ее запрашивали, если fair=false – потоки будут получать блокировку в случайной последовательности).

Методы *ReentrantLock*

Методы класса ReentrantLock описаны в табл. 3.

Таблица 3. Методы класса ReentrantLock

Модификатор и тип	Метод и его описание
`int`	`getHoldCount()` Запрос количества удерживаемых блокировок для текущего потока
`protected Thread`	`getOwner()` Возвращаает поток, который владеет блокировкой в настоящее время, или `null`, если блокировкой никто не владеет
`protected Collection <Thread>`	`getQueuedThreads()` Возвращают коллекцию, содержащую потоки, ожидающие получения блокировки
`int`	`getQueueLength()` Возвращает оценку количества потоков, ожидающих получения блокировки
`protected Collection <Thread>`	`getWaitingThreads(Condition condition)` Возвращает коллекцию, содержащую потоки, которые ожидают на условии, ассоциированном с этой блокировкой

Продолжение таблицы 3

`int`	`getWaitQueueLength(Condition condition)` Возвращает оценку количества потоков, ожидающих на заданном в качестве параметра условии, ассоциированном с этой блокировкой
`boolean`	`hasQueuedThread(Thread thread)` Ожидает ли заданный в качестве параметра поток получения этой блокировки?
`boolean`	`hasQueuedThreads()` Ожидает ли какой-либо поток получения этой блокировки?
`boolean`	`hasWaiters(Condition condition)` Ожидает ли какой-либо поток на условии, переданном в качестве параметра, ассоциированном с этой блокировкой?
`boolean`	`isFair()` Возвращает `true`, если «справедливость» блокировки установлена в `true`
`boolean`	`isHeldByCurrentThread()` Возвращает `true`, если текущий поток удерживает блокировку, и `false` в противном случае
`boolean`	`isLocked()` Возвращает `true`, если блокировка удерживается каким-то потоком, и `false` в противном случае
`void`	`lock()` Получение блокировки
`void`	`lockInterruptibly()` Получает блокировку, если текущий поток не был прерван. Устанавливает счетчик полученных блокировок в 1 и немедленно возвращает результат. Если текущий поток уже удерживает эту блокировку, то значение счетчика блокировки увеличивается на 1 и метод немедленно возвращает результат. Если блокировка удерживается другим потоком, то текущий поток помещается в очередь ожидания и находится в ней до тех пор, пока не произойдет одно из двух событий: - текущий поток получит блокироку; - некоторый другой поток прервет текущий поток.

Окончание таблицы 3

	Если блокировку получит текущий поток, то счетчик блокировки будет установлен в 1. Если у текущего потока установлен статус «прерван» или поток был прерван во время получения блокировки, то выбрасывается исключение `InterruptedException` и статус потока «прерван» очищается
`Condition`	`newCondition()` Возвращает новый объект условия (Condition) для использования с этим экземпляром Lock
`String`	`toString()` Возвращает строковое представление блокировки
`boolean`	`tryLock()` Получает блокировку только в том случае, если никакой другой поток во время вызова не владеет этой блокировкой
`boolean`	`tryLock(long timeout, TimeUnit unit)` Получает блокировку только в том случае, если никакой другой поток в течение времени `timeout` не владеет этой блокировкой и текущий поток не был прерван
`void`	`unlock()` Пытается снять блокировку

Рассмотрим, как можно использовать *ReentrantLock* для синхронизации:

```
public class SharedObject {
    ReentrantLock lock = new ReentrantLock();
    int counter = 0;
    public void perform() {
        lock.lock();
        try {
           // Критическая секция кода
            count++;
        } finally {
            lock.unlock();
        }
    }
   //...
}
```

Обратите внимание на использование вызовов *lock ()* и *unlock ()* в блоке *try-finally*, чтобы избежать тупиковых ситуаций.

В следующем коде поток, вызывающий *tryLock ()*, будет ждать одну секунду и перестанет ждать, если блокировка недоступна.

```
public void performTryLock(){
   //...
    boolean     isLockAcquired     =     lock.tryLock(1,
TimeUnit.SECONDS);

    if(isLockAcquired) {
        try {
           //Критическая секция кода
        } finally {
            lock.unlock();
        }
    }
   //...
}
```

Пример использования *ReentrantLock*

Есть магазин (или склад), в котором могут одновременно быть размещено не более 3 товаров. Производитель должен произвести 10 товаров, а покупатель должен эти товары купить. В то же время покупатель не может купить товар, если на складе нет никаких товаров.

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class ProgramLock {
      public static void main(String[] args) {
      // Создаем объект магазина
      Store store=new Store();
      // Создаем производителя
      Producer producer = new Producer(store);
      // Создаем потребителя
      Consumer consumer = new Consumer(store);
      //   Создаем   и   запускаем   поток,   моделирующий
производство
      new Thread(producer).start();
      //   Создаем   и   запускаем   поток,   моделирующий
потребление
      new Thread(consumer).start();
    }
}
// Класс Магазин (или склад – как кому нравится),
// хранящий произведенные товары
class Store{
   // Продукции в магазине - изначально нет,
```

```
   // будет производиться производителем и потребляться
потребителем
   private int product=0;
   // Блокировка
   ReentrantLock locker;
   // Условие, ассоциированное с блокировкой
   Condition condition;
   // Конструктор для Магазина
   Store(){
      // Создаем блокировку
      locker = new ReentrantLock();
      // Получаем условие, ассоциированное с блокировкой
       condition = locker.newCondition();
   }
   // Метод получения товара из магазина
   public void get() {
      locker.lock();
      try{
          //  Пока  в  магазине  нет  доступных  товаров,
ожидаем
          while (product<1)
              condition.await();
          //  Как  только  товар  появится,  уменьшаем  его
количество на 1
          product--;
          System.out.println("Покупатель     купил     1
товар");
          System.out.println("Товаров  на  складе:  "  +
product);
          // Сигнализируем о том, что количество товара
изменилось
          condition.signalAll();
      }
      catch (InterruptedException e){
          System.out.println(e.getMessage());
      }
      finally{
          locker.unlock();
      }
   }
   // Метод поставки товара в магазин
   public void put() {
        locker.lock();
       try{
          // Пока на складе 3 товара, ждем освобождения
места
          while (product>=3)
              condition.await();
```

```
            // Как только товара стало меньше 3 единиц,
            // добавить на склад 1 единицу товара
            product++;
            System.out.println("Производитель добавил 1
товар");
            System.out.println("Товаров на складе: " +
product);
            // Сигнализируем о том, что количество товара
изменилось
            condition.signalAll();
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock();
        }
    }
}
// Класс Производитель
class Producer implements Runnable{

    Store store;
//Конструктор производителя. Параметр – в какой магазин
будет поставлять товар
    Producer(Store store){this.store=store;}
    public void run(){
        for (int i = 1; i < 1; i++) {
            try {
               Thread.currentThread();
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            store.put();
        }
    }
}
// Класс Потребитель
class Consumer implements Runnable{
    Store store;
//  Конструктор потребителя. Параметр – из какого
магазина будет брать
// товар
    Consumer(Store store){this.store=store;}
    public void run(){
        for (int i = 1; i < 11; i++) {
            try {
```

```
                Thread.currentThread();
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            store.get();
        }
    }
}
```

Класс *ReentrantReadWriteLock*

```
public class ReentrantLock extends Object implements
Lock, Serializable
```

Класс блокировок чтения/записи ReentrantReadWriteLock удобен в том случае, когда существует много потоков, читающих разделяемую структуру данных, и значительно меньшее количество потоков, изменяющих ее. В такой ситуации имеет смысл разрешить разделенный доступ потокам чтения, при том что записывающий поток должен иметь исключительный доступ.

Для использования блокировок чтения и записи необходимо проделать следующие шаги.

1. Создать объект ReentrantReadWriteLock:

```
private ReentrantReadWriteLock rwl=
= ReentrantReadWriteLock()
```

2. Извлечь из созданного объекта блокировки чтения и записи:

```
private Lock readlock=rwl.readLock();
private Lock writelock=rwl.writeLock();
```

3. Использовать блокировку чтения для всех читателей:

```
readlock.lock();
try {…Выполнить необходимое чтение…}
finally(readlock.unlock();)
```

4. Использовать блокировку записи для всех потоков, которые изменяют данные (писателей):

```
writelock.lock();
try {…Записать данные…}
finally(readlock.unlock();)
```

Несколько потоков чтения могут получить доступ к разделяемым данным в том случае, если не выполняется операция записи.

Конструкторы *ReentrantReadWriteLock*

`public ReentrantLock()` - создает объект ReentrantLock. Эквивалентен ReentrantLock(false).

`public ReentrantLock(boolean fair)` - создает объект ReentrantLock с заданной политикой «справедливости» fair. Если fair =

= true, то раньше выдается блокировка тому потоку, который раньше ее запросил. В противном случае последовательность предоставления блокировок недетерминирована.

Методы *ReentrantReadWriteLock*

Методы класса ReentrantReadWriteLock представлены в табл. 4.

Таблица 4. Методы класса ReentrantReadWriteLock

Модификатор и тип	Метод и его описание
`protected Thread`	`getOwner()` Возвращает поток, владеющий в момент вызова метода блокировкой записи, или `null`, если такого потока нет
`protected Collection<Thread>`	`getQueuedReaderThreads()` Возвращает коллекцию, содержащую потоки, ожидающие получения блокировки чтения
`protected Collection<Thread>`	`getQueuedThreads()` Возвращает коллекцию, содержащую потоки, ожидающие получения блокировок чтения или записи
`protected Collection<Thread>`	`getQueuedWriterThreads()` Возвращает коллекцию, содержащую потоки, ожидающие получения блокировки записи
`int`	`getQueueLength()` Возвращает оценку количества потоков, ожидающих получения блокировок чтения или записи
`int`	`getReadHoldCount()` Получение количества повторно входимых удерживаемых текущим потоком блокировок чтения
`int`	`getReadLockCount()` Получение количества блокировок чтения, удерживаемых этой блокировкой

Продолжение таблицы 4

`protected Collection<Thread>`	`getWaitingThreads(Condition condition)` Возвращает коллекцию потоков, ожидающих условие `condition`, ассоциированном с блокировкой записи
`int`	`getWaitQueueLength(Condition condition)` Возвращает оценку количества потоков, ожидающих на заданном условии `condition`, ассоциированном с блокировкой записи
`int`	`getWriteHoldCount()` Возвращает количество повторных вхождений текущего потока в блокировку записи
`boolean`	`hasQueuedThread(Thread thread)` Определяет, ожидает ли переданный в качестве параметра поток получения блокировок чтения или записи
`boolean`	`hasQueuedThreads()` Определяет, ожидает ли какой-либо поток получения блокировки чтения или записи
`boolean`	`hasWaiters(Condition condition)` Определяет, ожидает ли какой-либо поток на передаваемом в качестве параметра условии `condition`, ассоциированном с блокировкой записи
`boolean`	`isFair()` Возвращает `true`, если для блокировки установлен режим «справедливости»

Окончание таблицы 4

`boolean`	`isWriteLocked()` Определяет, захвачена ли блокировка записи каким-либо потоком
`boolean`	`isWriteLockedByCurrentThread()` Определяет, захвачена ли блокировка записи текущим потоком
`ReentrantReadWriteLock.ReadLock`	`readLock()` Возвращает блокировку чтения
`String`	`toString()` Возвращат строковое представление блокировки
`ReentrantReadWriteLock.WriteLock`	`writeLock()` Возвращает блокировку для записи

Конструктор *ReentrantReadWriteLock.ReadLock*

```
protected ReadLock(ReentrantReadWriteLock lock)
```

Конструктор предназначен для использования в подклассах. Параметры: `lock` – внешний объект lock.

Выбрасывает исключения: `NullPointerException` - если lock – null.

Методы *ReentrantReadWriteLock.ReadLock*

Методы класса ReentrantReadWriteLock приведены в табл. 5.

Таблица 5. Методы класса ReentrantReadWriteLock.ReadLock

Модификатор и тип	**Метод и его описание**
`void`	`lock()` Получение блокировки чтения
`void`	`lockInterruptibly()` Получает блокировку чтения, если текущий поток не был прерван
`Condition`	`newCondition()` Выбрасывает исключение `UnsupportedOperationException`, поскольку `ReadLocks` не поддерживает условий
`String`	`toString()` Возвращает строковое представление блокировки

Окончание таблицы 5

`boolean`	`tryLock()` Получает блокировку чтения только в том случае, если никакой другой поток не владеет блокировкой записи во время вызова метода
`boolean`	`tryLock(long timeout, TimeUnit unit)` Получает блокировку чтения, если никакой поток не владеет блокировкой записи и не истекло время тайм-аута, при этом текущий поток не должен быть прерван
`void`	`unlock()` Пытается освободить блокировку

Конструктор *ReentrantReadWriteLock.WriteLock*

`protected WriteLock(ReentrantReadWriteLock lock)` – конструктор для использования в подклассах.

Параметры: lock – внешний объект блокировки.

Выбрасывает исключения: `NullPointerException` - если lock = = null.

Методы *ReentrantReadWriteLock.WriteLock*

Методы класса ReentrantReadWriteLock приведены в табл. 6.

Таблица 6. Методы класса ReentrantReadWriteLock.WriteLock

Модификатор и тип	**Метод и его описание**
`int`	`getHoldCount()` Возвращает количество удерживаемых текущим потоком блокировок записи
`boolean`	`isHeldByCurrentThread()` Возвращает true, если блокировка записи удерживается текущим потоком, и false в противном случае
`void`	`lock()` Получает блокировку чтения
`void`	`lockInterruptibly()` Получает блокировку чтения, если текущий поток не прерван
`Condition`	`newCondition()` Возвращает объект условия для блокировки
`String`	`toString()` Возвращает строковое представление блокировки

Окончание таблицы 6

`boolean`	`tryLock()` Получает блокировку записи только в том случае, если во время вызова метода этой блокировкой не владел никакой поток
`boolean`	`tryLock(long timeout, TimeUnit unit)` Получает блокировку записи, если время тайм-аута ожидания блокировки не истекло и поток не был прерван
`void`	`unlock()` Пытается освободить блокировку

Порядок выполнения работы

1. Изучите теоретический материал.
2. Изучите работу примеров использования блокировок, используя Eclipse (за исключением примеров использования StampedLock).
3. Выполните практическое задание.
4. Ответьте на контрольные вопросы.

Практическое задание

Внесите изменения в пример использования ReentrantLock таким образом, чтобы покупатель мог покупать 1 или 2 товара. Количество покупаемых товаров выбирается случайно.

Контрольные вопросы

1. Для чего используются явные блокировки?
2. Опишите методы интефейса Lock.
3. Опишите методы интефейса ReadWriteLock.
4. Опишите конструкторы и методы класса ReentrantLock.
5. Прокомментируйте пример использования класса Reentrant-Lock.
6. Опишите конструкторы и методы класса ReentrantRead-WriteLock.
7. Опишите конструкторы и методы класса ReentrantRead-WriteLock.ReadLock.

Библиографический список

1. Хорстманн К., Корнелл Г. Java2. Библиотека профессионала, том 1. Основы. – М.: ООО «И.Д. Вильямс», 2008.

2. Эккель Б. Философия Java. – СПб.: Питер, 2009.

3. Гетц Б., Пайерлс Т., Блох Д., Боубер Д., Холмс Д., Ли Д. Java Concurrency на практике. — СПб.: Питер, 2020.

Многопоточное программирование на Java. Блокировки

Составители: М и т р о ш и н Александр Александрович
П с о я н ц Владимир Грикорович

Редактор М.Е. Цветкова
Корректор И.В. Черникова
Подписано в печать 30.06.22. Формат бумаги 60×84 1/16.
Бумага писчая. Печать трафаретная. Усл. печ. л. 1,0.
Тираж 25 экз. Заказ
Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.