**Блокировки пакета java.util.concurrent.locks**

Обычная синхронизация обеспечивает надежное обновление множественных общих переменных в многопоточном приложении предупреждая гонку или повреждение данных и гарантирует, что параллельные потоки, синхронизируемые должным образом, увидят последние значения этих переменных. Но такая синхронизация не совершена и имеет некоторые функциональные ограничения:

* невозможно прервать поток, который ожидает блокировки;
* невозможно опрашивать или пытаться получить блокировку, не будучи готовым к долгому ожиданию;
* блокировка должна быть снята в том же стековом фрейме, в котором была начата.

Пакет *java.util.concurrent.locks* включает классы, которые можно использовать для блокировки ресурсов с определенными условиями, и которые существенно отличаются от встроенной синхронизации и мониторов. Этот пакет разрешает намного большую гибкость в использовании блокировок без условий и с условием. Классы пакета реализуют следующие интерфейсы :

|  |  |
| --- | --- |
| • [Lock](http://java-online.ru/concurrent-locks.xhtml" \l "lock) | интерфейс поддерживает порядок блокировки и позволяет использовать многократно связанный условный объект *Condition*; |
| • [Condition](http://java-online.ru/concurrent-locks.xhtml" \l "condition) | интерфейс описывает связанные с блокировками переменные, которые могут выполнять функции монитора объекта; |
| • [ReadWriteLock](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html" \t "_blank) | интерфейс поддерживает пару связанных блокировок : одну для чтения и одну для записи. |

На странице наряду с описанием блокировок представлены три примера их использования :

* пример использования блокировки [ReentrantLock](http://java-online.ru/concurrent-locks.xhtml" \l "ereentrantLock);
* пример блокировки методом [lockInterruptibly](http://java-online.ru/concurrent-locks.xhtml" \l "elockInterruptibly);
* пример использования условия блокировки [Condition](http://java-online.ru/concurrent-locks.xhtml" \l "econdition).

**Интерфейс Lock**

Интерфейс **Lock** — это абстракция, допускающая выполнение блокировок, которые реализуются как классы Java, а не как возможность языка (объекта). Это расширяет возможности применения Lock, которые могут иметь различные алгоритмы планирования. Блокировка *Lock* является инструментом для того, чтобы управлять доступом к совместно используемому ресурсу паралельными потоками.

Реализации интерфейса *Lock* существенно расширяют возможности блокировок по сравнению c *synchronized*. Интерфейс *Lock* позволяет осуществлять более гибкое структурирование и поддерживает многократно связанный условный объект *Condition*.

**Методы интерфейса Lock**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| lock() | Получение блокировки ([пример](http://java-online.ru/concurrent-locks.xhtml#ereentrantLock)) |
| lockInterruptibly() | Получение блокировки, если текущий поток не прерывается ([пример](http://java-online.ru/concurrent-locks.xhtml#elockInterruptibly)) |
| newCondition() | Получение нового Condition, связанного с блокировкой Lock ([пример](http://java-online.ru/concurrent-locks.xhtml#econdition)) |
| tryLock() | Получение блокировки, если она свободна во время вызова |
| tryLock(long time, TimeUnit unit) | Получение блокировки в течение заданного времени |
| unlock() | Освобождение блокировки |

Дополнительные возможности, предоставляемые блокировкой *Lock* накладывают определенные обязанности при ее использовании. Так, отсутствие блочно-структурированной блокировки исключает автоматическое ее освобождение, как это происходит с *synchronized*. Поэтому следует использовать следующую структуру кода, включающую блокировку Lock :

Lock lk = ...;

lk.lock();

try {

// доступ к защищенному блокировкой ресурсу

} finally {

// освобождение блокировки

lk.unlock();

}

Подробное описание интерфейса Lock представлено [здесь](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html).

1. **Класс ReentrantLock**

 ReentrantLock - это эксклюзивная блокировка, поэтому только один поток может получить блокировку.

Класс **ReentrantLock**, реализующий интерфейс *Lock*, также, как и *synchronized*, обеспечивает многопоточность, но имеет дополнительные возможности, связанные с опросом о блокировании (lock polling), ожиданием блокирования в течение определенного времени и прерыванием ожидания блокировки одного Треда другим Тредом.

Как прямая замена synchronized (*без использования дополнительных*) стоит использовать только в том случае, если у вас очень часто происходит битва многих потоков за монитор – будет работать быстрее.

В переводе *reentrant* может означать *повторно используемый* (повторный вход). Т.е. один и тот же поток повторно получает одну и ту же блокировку. Но для того, чтобы реально разблокировать необходимо уже будет два раза снять блокировку. Это аналогично использованию *synchronized*; если поток повторно входит в синхронный блок, защищенный монитором, то блокировка не будет снята при выходе потока из второго (или последующего) блока synchronized, блокировка будет снята только когда поток выйдет из первого блока synchronized, в который он вошел под защитой монитора.

Одним из методов интерфейса *Lock* и его реализации *ReentrantLock* является запрос блокировки с возможностью прерывания процесса ожидания. Т.е. если поток запрашивает блокировку методом lockInterruptibly() и не получает ее сразу же, то переходит в процесс ожидания. Методом *interrupt* работу потока можно прервать. Тогда ожидающий блокировки поток просыпается, и генерируется исключительная ситуация InterruptedException. После этого попыток доступа к защищенному ресурсу (получения блокировок) не делается и освобождать блокировку не требуется. Ниже представлен пример использования блокировки [lockInterruptibly](http://java-online.ru/concurrent-locks.xhtml" \l "elockInterruptibly). Структура кода использования блокировки *lockInterruptibly* имеет следующий вид :

Lock l = new ReentrantLock();

try {

l.lockInterruptibly();

try {

// работа с защищенным ресурсом

} finally {

l.unlock();

}

} catch (InterruptedException e) {

System.err.println("Interrupted wait");

}

Внутренний блок try-finally получает блокировку и доступ к защищенным ресурсам; после завершения работы блокировка освобождается. Внешний блок try-catch обрабатывает исключительные ситуации запроса блокировки. Если поток прерван в результате исключительной ситуации, то выполняется перехват *catch (InterruptedException)* и метод снятия блокировки *unlock* не вызывается.

**Пример использования ReentrantLock без использования дополнительных методов**

ReentrantLock lock = new ReentrantLock();

int count = 0;

void increment() {

lock.lock();

try {

count++;

} finally {

lock.unlock();

}

}

**Пример использования ReentrantLock и метода tryLock()**

ExecutorService executor = Executors.newFixedThreadPool(2);

ReentrantLock lock = new ReentrantLock();

executor.submit(() -> {

lock.lock();

try {

sleep(1);

} finally {

lock.unlock();

}

});

executor.submit(() -> {

System.out.println("Locked: " + lock.isLocked());

System.out.println("Held by me: " + lock.isHeldByCurrentThread());

boolean locked = lock.tryLock();

System.out.println("Lock acquired: " + locked);

});

stop(executor);

Пока первый поток удерживает блокировку, второй выведет следующую информацию:

Locked: true  
Held by me: false  
Lock acquired: false

Метод tryLock(), в отличие от обычного lock() не останавливает текущий поток в случае, если ресурс уже занят. Он возвращает булевый результат, который стоит проверить перед тем, как пытаться производить какие-то действия с общими объектами (истина обозначает, что контроль над ресурсами захватить удалось).

**Пример использования прерывания ожидания блокировки Тредом самого себя через 10 сек. tryLock() в ReentrantLock**

package ua.com.prologistic;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ConcurrencyLockExample implements Runnable{

    private Resource resource;

    private Lock lock;

    public ConcurrencyLockExample(Resource r){

        this.resource = r;

        this.lock = new ReentrantLock();

    }

*@Override*

    public void run() {

        try {

            // пытаемся взять лок в течении 10 секунд

            if(lock.tryLock(10, TimeUnit.SECONDS)){

            resource.doSomething();

            }

        } catch (InterruptedException e) {

            e.printStackTrace();

        }finally{

            //убираем лок

            lock.unlock();

        }

        // Для логгирования не требуется потокобезопасность

        resource.doLogging();

    }

}

Здесь используется метод tryLock(), чтобы убедиться в том, что поток ждет только определенное время. Если он не получает блокировку на объект, то просто логгирует и выходит.

**Еще один важный момент**. Мы используем блок try-finally, чтобы убедиться в том, что блокировка будет снята, даже если метод doSomething() бросит исключение.

**Пример использования прерывания Тредом другого Треда методом interrupt() в блоке кода блокированном lockInterruptibly() в ReentrantLock .**

Дальше пойдет пример с трудно читаемым говнокодом, где мудак все действия, которые должны бать в main записал в конструкторе (и запуски потоков и отбор через while) и плюс использует внутренний класс для переопределения метода run() для создания Тредов. А в main он запускает сам себя, создавая объект собственного класса.

Текстовая переменная *resource* используется в качестве общего ресурса, значение которого будет изменяться внутри потоков.

Метод printMessage выводит в консоль сообщения потоков с указанием времени.

А идея этого говнокода в следующем:

Второй и третий потоки стартуют с небольшой задержкой, что однозначно дает первому потоку первому поставить блокировку lock.

Метод *run* класса LockClass вызывает метода lockInterruptibly для получения блокировки. После этого следует небольшая задержка в 2 сек, и далее выполняется проверка, если это первый поток, то он прерывает работу второго потока. После чего первый поток дает возможность второму потоку завершить работу, а сам изменяет и печатает строку общего ресурса, снимает блокировку и завершает работу. Второй же поток не попадает в секцию try...finally, а завершается с исключением, которое перехватывает catch (InterruptedException).

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample

{

String resource = "Hello, World!";

SimpleDateFormat sdf;

sdf = new SimpleDateFormat("HH:mm:ss ");

final int TIME\_WAIT = 7000;

final int TIME\_SLEEP = 5000;

Lock lock;

Thread thread1;

Thread thread2;

Thread thread3;

ReentrantLockExample()

{

lock = new ReentrantLock();

thread1 = new Thread(new LockClass("first",

"Первый поток"))

thread2 = new Thread(new LockClass("second",

"Второй поток"));

thread3 = new Thread(new LockClass("third" ,

"Третий поток"));

thread1.start();

try {

Thread.sleep(400);

thread2.start();

thread3.start();

} catch (InterruptedException e) {}

printMessage(null);

while (thread1.isAlive() || thread2.isAlive()) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("\nЗавершение работы примера");

System.exit(0);

}

//-----------------------------------------------------

void printMessage(final String msg)

{

String text = sdf.format(new Date());

if (msg == null){ text += resource;}

else { text += msg;}

System.out.println(text);

}

//-----------------------------------------------------

class LockClass implements Runnable

{

String name;

String text;

public LockClass(String name, String text) {

this.name = name;

this.text = text;

}

@Override

public void run()

{

try {

printMessage("Wait (" + name + ") ...");

lock.lockInterruptibly();

try {

Thread.sleep(2000);

if (name.equalsIgnoreCase("first")) {

printMessage("Прерывание второго потока");

thread2.interrupt();

thread2.join();

}

// доступ к ресурсу

resource = text;

printMessage(null);

Thread.sleep(TIME\_SLEEP);

} finally {

// Убираем блокировку

lock.unlock();

String text = name + " : завершил работу";

printMessage(text);

}

} catch (InterruptedException e) {

printMessage(name + " : interrupted wait");

}

}

}

//-----------------------------------------------------

public static void main(String[] args) {

new ReentrantLockExample();

}

}

**Сообщения потоков**

Ниже представлены сообщения потоков. Сначала потоки запрашивают блокировку, первый поток, пришедший первым, получает ее, а два потока остаются в ожидании освобождения/получения блокировки. После небольшой задержки первый поток прерывает работу второго потока и ждет его завершения. Далее первый поток освобождает блокировку и завершает работу. Сразу же после этого третий поток получает блокировку и далее по сценарию.

11:17:20 Wait (first) ...

11:17:21 Wait (second) ...

11:17:21 Wait (third) ...

11:17:22 Прерывание второго потока

11:17:22 second : interrupted wait

11:17:22 Первый поток

11:17:29 first : завершил работу

11:17:31 Третий поток

11:17:38 third : завершил работу

Завершение работы примера

**Интерфейс Condition**

Интерфейсное условие **Condition** в сочетании с блокировкой *Lock* позволяет заменить методы монитора/мьютекса (wait, notify и notifyAll) объектом, управляющим ожиданием событий. Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора. **Т.е. тут после выполнения одного потока запускается на выполнение конкретный другой поток, а не просто все или последний ставший на lock. При этом я использую более быстрый (чем Synchronized) многопоточный класс ReentrantLock.**

С помощью Condition мы как-бы даем имя для каждого await(). И потом можем разжимать конкретный await().

Condition это метка, находящаяся в середине кода, после строки блокировки и проверочного while, при срабатывании которой (вызова метода await()) происходит разблокировка ресурса (который перед этим был заблокирован нашей блокировкой) и ожидание от другого блока кода команды signal(). Когда после получения команды signal() от другого блока и проверки while-ом эта метка наконец-то не сработает, то будет выполнен код находящийся ниже ее (т.е. на заблокированном нами ресурсе будет выполнена какая-то операция). В конце этого кода мы сами пишем команду signal() для разжимания какого-то другого конкретного блока кода в это момент стоящего на метке типа Condition.

**Методы интерфейса Condition**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| await() | Переводит поток в состояние ожидания до тех пор, пока не будет выполнено некоторое условие или пока другой поток не вызовет методы signal/signalAll |
| await(long time, TimeUnit unit) | Переводит поток в состояние ожидания на определенное время пока не будет выполнено некоторое условие или пока другой поток не вызовет методы signal/signalAll |
| signal() | Сигнализирует потоку, с конкретно указанным Condition-ном, у которого ранее был вызван метод await(), о возможности продолжения работы. Применение аналогично использованию методу notify класса Object |
| signalAll() | Сигнализирует всем потокам с Condition-ами, у которых ранее был вызван метод await(), о возможности продолжения работы. Применение аналогично использованию методу notifyAll класса Object |

Условие *Condition*, иначе именуемое как очередь условия, предоставляет средство управления для одного потока, чтобы приостановить его выполнение до тех пор, пока он не будет уведомлен другим потоком. Объект Condition связывают с блокировкой. Чтобы получить Condition для блокировки Lock используют метод newCondition().

ReentrantLock locker = new ReentrantLock();

Condition condition = locker.newCondition();

Чтобы перевести поток в ожидание, если определенное условие не выполняется, то используется метод *await* :

while (условие)

condition.await();

После завершения всех действий в потоке (при выходе) подается сигнал об изменении условия другим потокам :

condition.signalAll();

**Пример использования Condition**

class BoundedBuffer {

**final Lock lock = new ReentrantLock();**

final Condition notFull = **lock.newCondition();**

final Condition notEmpty = **lock.newCondition();**

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

**lock.lock();**

**try {**

while (count == items.length)

**notFull.await();**

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

**notEmpty.signal();**

**} finally {**

**lock.unlock();**

**}**

}

public Object take() throws InterruptedException {

**lock.lock();**

**try {**

while (count == 0)

**notEmpty.await();**

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

**notFull.signal();**

return x;

**} finally {**

**lock.unlock();**

**}**

}

}

Дальше идет пример со складом/производителем/покупателем/магазином:

import java.util.concurrent.locks.ReentrantLock;

import java.util.concurrent.locks.Condition;

public class Program {

    public static void main(String[] args) {

        Store store=new Store();

        Producer producer = new Producer(store);

        Consumer consumer = new Consumer(store);

        new Thread(producer).start();

        new Thread(consumer).start();

    }

}

// Класс Магазин, хранящий произведенные товары

class Store{

   private int product=0;

   ReentrantLock locker;

   Condition conditionGet;

   Condition conditionPut;

   Store(){

       locker = new ReentrantLock(); // создаем блокировку

       conditionGet = locker.newCondition();

conditionPut = locker.newCondition(); // получаем условия, связанные с блокировкой

   }

   public void get() {

      locker.lock();

      try{

          // пока нет доступных товаров на складе, ожидаем

          while (product<1)

              conditionGet.await();

          product--;

          System.out.println("Покупатель купил 1 товар");

          System.out.println("Товаров на складе: " + product);

          // сигнализируем

          conditionPut.signal();

      }

      catch (InterruptedException e){

          System.out.println(e.getMessage());

      }

      finally{

          locker.unlock();

      }

   }

   public void put() {

       locker.lock();

       try{

          // пока на складе 3 товара, ждем освобождения места

          while (product>=3)

              conditionPut.await();

          product++;

          System.out.println("Производитель добавил 1 товар");

          System.out.println("Товаров на складе: " + product);

          // сигнализируем

          conditionGet.signal();

      }

      catch (InterruptedException e){

          System.out.println(e.getMessage());

      }

      finally{

          locker.unlock();

      }

   }

}

// класс Производитель

class Producer implements Runnable{

    Store store;

    Producer(Store store){

       this.store=store;

    }

    public void run(){

        for (int i = 1; i < 6; i++) {

            store.put();

        }

    }

}

// Класс Потребитель

class Consumer implements Runnable{

     Store store;

    Consumer(Store store){

       this.store=store;

    }

    public void run(){

        for (int i = 1; i < 6; i++) {

            store.get();

        }

    }

}

На экране получим:

Производитель добавил 1 товар

Товаров на складе: 1

Производитель добавил 1 товар

Товаров на складе: 2

Производитель добавил 1 товар

Товаров на складе: 3

Покупатель купил 1 товар

Товаров на складе: 2

Покупатель купил 1 товар

Товаров на складе: 1

Покупатель купил 1 товар

Товаров на складе: 0

Производитель добавил 1 товар

Товаров на складе: 1

Производитель добавил 1 товар

Товаров на складе: 2

Покупатель купил 1 товар

Товаров на складе: 1

Покупатель купил 1 товар

Товаров на складе: 0

1. **Интерфейс ReadWriteLock + Класс ReentrantReedWriteLock**

Интерфейс ReadWriteLock предлагает другой тип блокировок — отдельную для чтения, и отдельную для записи. Этот интерфейс был добавлен из соображения, что считывать данные (любому количеству потоков) безопасно до тех пор, пока ни один из них не изменяет переменную. Таки образом, блокировку для чтения (*read-lock*) может удерживать любое количество потоков до тех пор, пока не удерживает блокировка для записи (*write-lock*). Такой подход может увеличить производительность в случае, когда чтение используется гораздо чаще, чем запись.

ExecutorService executor = Executors.newFixedThreadPool(2);

Map<String, String> map = new HashMap<>();

ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {

lock.writeLock().lock();

try {

sleep(1);

map.put("foo", "bar");

} finally {

lock.writeLock().unlock();

}

});

В примере выше мы можем видеть, как поток блокирует ресурсы для записи, после чего ждёт одну секунду, записывает данные в HashMap и освобождает ресурсы. Предположим, что в это же время были созданы ещё два потока, которые хотят получить из хэш-таблицы значение:

Runnable readTask = () -> {

lock.readLock().lock();

try {

System.out.println(map.get("foo"));

sleep(1);

} finally {

lock.readLock().unlock();

}

};

executor.submit(readTask);

executor.submit(readTask);

stop(executor);

Если вы попробуете запустить этот пример, то заметите, что оба потока, созданные для чтения, будут простаивать секунду, ожидая завершения работы потока для записи. После снятия блокировки они выполнятся параллельно, и одновременно запишут результат в консоль. Им не нужно ждать завершения работы друг друга, потому что выполнять одновременное чтение вполне безопасно (до тех пор, пока ни один поток не работает параллельно на запись).

1. **Класс StampedLock**

В Java 8 появился новый тип блокировок — StampedLock. Так же, как и в предыдущих примерах, он поддерживает разделение на readLock() и writeLock(). Однако, в отличие от ReadWriteLock, метод блокировки StampedLock возвращает “штамп” — значение типа long. Этот штамп может использоваться в дальнейшем, как для высвобождения ресурсов, так и для проверки состояния блокировки. Вдобавок, у этого класса есть методы, для реализации “оптимистичной” блокировки и для преобразования блокировок из одной в другую.

Вот таким образом следовало бы переписать наш предыдущий пример под использование StampedLock:

ExecutorService executor = Executors.newFixedThreadPool(2);

Map<String, String> map = new HashMap<>();

StampedLock lock = new StampedLock();

executor.submit(() -> {

long stamp = lock.writeLock();

try {

sleep(1);

map.put("foo", "bar");

} finally {

lock.unlockWrite(stamp);

}

});

Runnable readTask = () -> {

long stamp = lock.readLock();

try {

System.out.println(map.get("foo"));

sleep(1);

} finally {

lock.unlockRead(stamp);

}

};

executor.submit(readTask);

executor.submit(readTask);

stop(executor);

Работать этот код будет точно так же, как и его брат-близнец с ReadWriteLock. Тут, правда, стоит упомянуть, что в StampedLock *не* реализована реентерантность. Поэтому особое внимание нужно уделять тому, чтобы не попасть в ситуацию взаимной блокировки (*deadlock*).

В следующем примере демонстрируется “оптимистичная блокировка”:

ExecutorService executor = Executors.newFixedThreadPool(2);

StampedLock lock = new StampedLock();

executor.submit(() -> {

long stamp = lock.tryOptimisticRead();

try {

System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));

sleep(1);

System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));

sleep(2);

System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));

} finally {

lock.unlock(stamp);

}

});

executor.submit(() -> {

long stamp = lock.writeLock();

try {

System.out.println("Write Lock acquired");

sleep(2);

} finally {

lock.unlock(stamp);

System.out.println("Write done");

}

});

stop(executor);

Оптимистичная блокировка для чтения, вызываемая с помощью метода tryOptimisticRead(), отличается тем, что она всегда будет возвращать “штамп” не блокируя текущий поток, вне зависимости от того, занят ли ресурс, к которому она обратилась. В случае, если ресурс был заблокирован блокировкой для записи, возвращённый штамп будет равняться нулю. В любой момент можно проверить, является ли блокировка валидной с помощью lock.validate(stamp). Для приведённого выше кода результат будет таким:

Optimistic Lock Valid: true  
Write Lock acquired  
Optimistic Lock Valid: false  
Write done  
Optimistic Lock Valid: false

Оптимистичная блокировка является валидной с того момента, как ей удалось захватить ресурс. В отличии от обычных блокировок для чтения, оптимистичная не запрещает другим потокам блокировать ресурс для записи. Что же происходит в коде выше? После захвата ресурса блокировка является валидной и оптимистичный поток отправляется спать. В это время другой поток блокирует ресурсы для записи, не дожидаясь окончания работы чтения. Начиная с этого момента, оптимистичная блокировка перестаёт быть валидной (даже после окончания записи).

**Таким образом, при использовании оптимистичных блокировок вам нужно постоянно следить за их валидностью (проверять её нужно уже после того, как выполнены все необходимые операции).**

Иногда может быть полезным преобразовать блокировку для чтения в блокировку для записи не высвобождая ресурсы. В StampedLock это можно сделать с помощью метода tryConvertToWriteLock(), как в этом примере:

ExecutorService executor = Executors.newFixedThreadPool(2);

StampedLock lock = new StampedLock();

executor.submit(() -> {

long stamp = lock.readLock();

try {

if (count == 0) {

stamp = lock.tryConvertToWriteLock(stamp);

if (stamp == 0L) {

System.out.println("Could not convert to write lock");

stamp = lock.writeLock();

}

count = 23;

}

System.out.println(count);

} finally {

lock.unlock(stamp);

}

});

stop(executor);

В этом примере мы хотим прочитать значение переменной count и вывести его в консоль. Однако, если значение равно нулю, мы хотим изменить его на 23. Для этого нужно выполнить преобразования из readLock во writeLock, чтобы не помешать другим потокам обрабатывать переменную. В случае, если вы вызвали tryConvertToWriteLock() в тот момент, когда ресурс занят для записи другим потоком, текущий поток остановлен не будет, однако метод вернёт нулевое значение. В таком случае можно вызвать writeLock() вручную.

Так же есть методы tryConvertToReadLock() и tryConvertToOptimisticRead().