


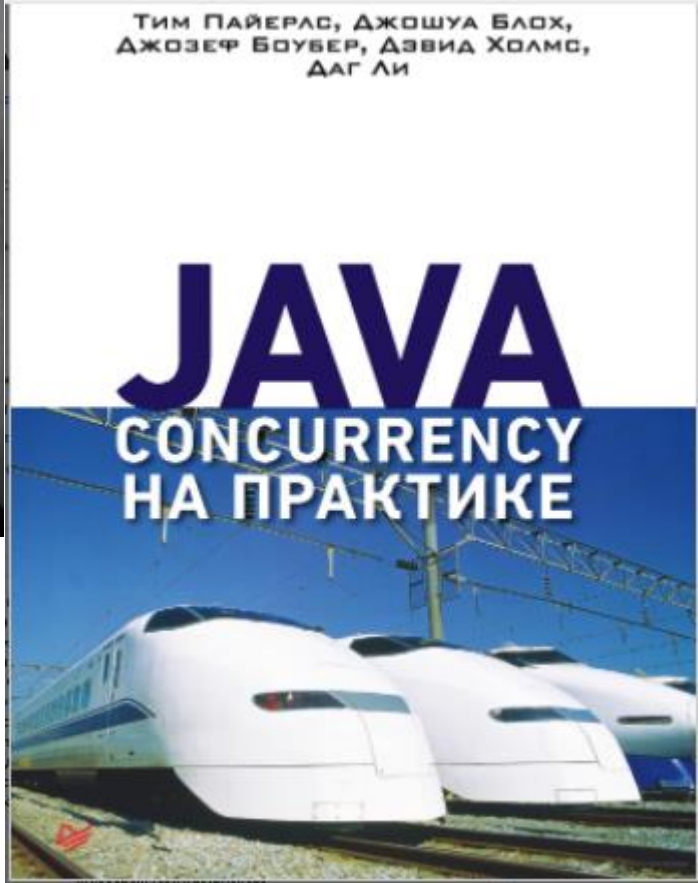
`java.util.concurrent`

Joker<?> 2017

Cay Horstmann

San Jose State University

Concurrency
for Humans



Многопоточное
программирование –
теория и практика

Роман Елизаров
Devexperts

Jpoint Student Day

0:13 / 49:30



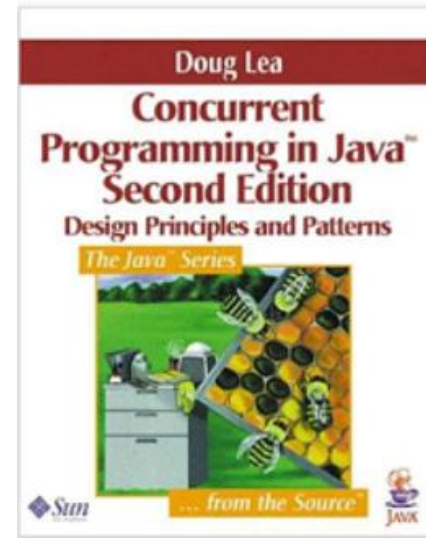
java.util.concurrent

Lea D. Concurrent Programming in Java

Начиная с Java 1.5

Включает

- классы для явного управления блокировкой
- коллекции для безопасной работы с потоками
- классы для управления синхронизацией
- классы, реализующие интерфейс Callable
- классы для реализации механизма ForkJoin
- классы для атомарных операций с переменными



`java.util.concurrent.atomic`

- Реализует семантику `volatile`
- Чаще всего используются для реализации последовательностей значений

Пример –генератор последовательности

```
class NextID {
    private int v;
    public NextID(){v=0;}
    public int getId() {return v++;}
}
class Generate implements Runnable{
    NextID id;
    public Generate (NextID n){    id=n;}
    public void run(){
        try {
            for (. . .){
                System.out.println("new ID "+id.getId());
                Thread.sleep();
            }
        } catch (InterruptedException e ){}
    }
}
```

```
NextID seq = new NextID();

Thread t[] = new Thread[n];
for (int i=0; i< n; i++) {
    Generate r = new Generate (seq);
    t[i] = new Thread(r);
    t[i].start();
}
```

Варианты запуска

4 потока

Каждый поток использует 5
сгенерированных значений

Ожидаем значения от 0 до 19

run:	run:	run:
new ID 1	new ID 0	new ID 1
new ID 3	new ID 2	new ID 3
new ID 2	new ID 1	new ID 0
new ID 0	new ID 3	new ID 2
new ID 4	new ID 4	new ID 4
new ID 5	new ID 5	<u>new ID 5</u>
<u>new ID 6</u>	new ID 6	<u>new ID 5</u>
<u>new ID 6</u>	new ID 7	new ID 6
new ID 7	new ID 8	new ID 7
new ID 8	new ID 9	new ID 8
new ID 9	new ID 10	new ID 9
new ID 10	new ID 11	new ID 10
new ID 11	new ID 12	new ID 11
new ID 12	new ID 13	new ID 12
new ID 13	new ID 15	new ID 13
new ID 14	new ID 14	<u>new ID 14</u>
new ID 15	new ID 16	<u>new ID 14</u>
new ID 16	new ID 17	new ID 15
new ID 17	new ID 18	new ID 16
new ID 18	new ID 19	new ID 17

Решение

```
synchronized public void run(){ // не поможет
    try {
        for (. . .){
            ...
        }
    }
}
```

Изменим класс, хранящий последовательные значения

```
class NextID {
    private final AtomicInteger v = new AtomicInteger(0);
    public int getId() {
        return v.getAndIncrement();
    }
    // private volatile int v;
    // public NextID(){v=0;}
    // public int getId() {return v++;}
}
```

```
run:
new ID 2
new ID 3
new ID 1
new ID 0
new ID 4
new ID 6
new ID 5
new ID 7
new ID 9
new ID 8
new ID 10
new ID 11
new ID 12
new ID 13
new ID 14
new ID 15
new ID 16
new ID 17
new ID 18
new ID 19
```

java.util.concurrent.locks

- Интерфейс Lock
- Классы
 - ReentrantLock
 - ReentrantReadWriteLock
- Методы
 - lock()
 - unlock()
 - tryLock()

Схема, если нет тупиков

```
private Lock myLock = new ReentrantLock();
myLock.lock(); // объект заблокирован, больше никто
                // не может его заблокировать
try {
    // критический фрагмент кода
}
finally {
    myLock.unlock(); // операцию разблокирования
                    //необходимо выполнить даже
                    //при возникновении исключения
}
```

Чтобы избежать тупиков

```
if (myLock.tryLock(1, TimeUnit.SECONDS)) {  
    try {  
        // критический фрагмент кода  
    }  
    finally{  
        myLock.unlock();  
    }  
}  
else { // блокировка не удалась  
    // выполнение других действий  
    // Thread.sleep(waitTime);  
}
```

Блокировки чтения-записи

Создать объект
ReentrantReadWriteLock

```
private ReentrantReadWriteLock rw =  
    new ReentrantReadWriteLock ();
```

Сформировать на его
основе отдельные
объекты для
блокировки чтения и
записи

```
private Lock readLock = rw.readLock();  
private Lock writeLock = rw.writeLock();
```

Блокировки чтения-записи

Использовать блокировку
чтения там, где возможен
совместный доступ

```
readLock.lock();  
  
try { . . . }  
  
finally (  
    readLock.unlock();  
}
```

Использовать блокировку
записи там, где
необходим эксклюзивный
доступ

```
writeLock.lock();  
  
try { . . . }  
  
finally {  
    writeLock.unlock();  
}
```

Объекты условий

Интерфейс Condition

Создается из объекта блокировки

```
Condition myCond = myLock.newCondition();
```

Методы объекта условия, связанного с объектом блокировки

await()

signal()

signalAll()

Пример использования

```
private Lock myLock = new ReentrantLock();
private Condition myCond = myLock.newCondition();
myLock.lock();
try {
    while(. . .)    //условие когда действие невозможно
        myCond.await();
    // действие
    myCond.signalAll();
}
finally {
    myLock.unlock();
}
```

Пример с банком

```
bankLock = new ReentrantLock();
sufficientFunds = bankLock.newCondition();

public void transfer(int from, int to, double amount)    throws InterruptedException
{
    if (bankLock. tryLock(1,TimeUnit.SECONDS)) {
        try {
            while (accounts[from] < amount)
                sufficientFunds.await(10, TimeUnit.MILLISECONDS);
            accounts[from] -= amount;
            accounts[to]    += amount;
            System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
            sufficientFunds.signalAll();
        } finally {    bankLock.unlock();    }
    }
}
```

Синхронизаторы

Синхронизаторы

- Классы, которые помогают управлять набором взаимодействующих потоков
- Используются в ряде типов взаимодействия между потоками («рандеву»)

<https://habr.com/ru/post/277669/>

Классы синхронизаторов

- Semaphore
- CyclicBarrier
- CountdownLatch
- Phaser (начиная с Java 7)
- Exchanger

Семафоры

- Имеет фиксированное количество разрешений
- Используются для ограничения количества потоков, которые могут получить доступ к некоторому (физическому или логическому) ресурсу
- Методы
 - `acquire()` / `tryAcquire()`
 - `availablePermits()`
 - `release()`
 - ...
- Семафор с одним разрешением называется мьютексом

Обратный счетчик

- `CountDownLatch`
- Замок с обратным отсчетом
- Предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках
- Когда все операции завершатся, ожидающие потоки будут «отпущены», чтобы продолжить свою деятельность

Обратный счетчик

```
CountDownLatch gateway = new CountDownLatch(3);
Runnable task = new Runnable ( ){
    public void run() {
        try {
            gateway.countDown();
            System.out.println("Countdown: " + gateway.getCount());
            gateway.await();
            System.out.println("Finished");
        } catch (InterruptedException e) { e.printStackTrace(); } };

for (int i = 0; i < 3; i++) { new Thread(task).start(); }
```

Барьер

- CyclicBarrier
- Является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется.
- Как только все потоки прибыли, выполняется опциональное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются»
- Барьер можно использовать снова, даже после того, как он сломается.

Барьеры

```
Runnable barrierAction = . . . ;// действие барьера  
CyclicBarrier barrier = new CyclicBarrier(nthreads,  
    barrierAction);
```

```
// метод потока  
public void run() {  
    doWork();  
    barrier.await(); // ждать у барьера  
    . . . .  
}
```

Класс Exchanger<V>

- Используется при работе двух потоков с двумя экземплярами одной структуры данных.
- Один поток заполняет свой экземпляр, другой считывает из своего экземпляра
- Когда каждый поток выполнит свою задачу, они обмениваются экземплярами

Phaser

- Этот класс позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия.
- Как и `CyclicBarrier`, `Phaser` является точкой синхронизации, в которой встречаются потоки-участники.
- Когда все стороны прибыли, `Phaser` переходит к следующей фазе и снова ожидает ее завершения.

Phaser

- Каждая фаза (цикл синхронизации) имеет номер;
- Количество сторон-участников жестко не задано и может меняться: поток может регистрироваться в качестве участника и отменять свое участие;
- Участник не обязан ожидать, пока все остальные участники соберутся на барьере. Чтобы продолжить свою работу достаточно сообщить о своем прибытии;
- Случайные свидетели могут следить за активностью в барьере;
- Поток может и не быть стороной-участником барьера, чтобы ожидать его преодоления;
- У фазера нет опционального действия.