ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М.И. ЧЕРДЫНЦЕВА

СРЕДСТВА ДЛЯ РАЗРАБОТКИ МНОГОПОТОЧНЫХ ПРОГРАММ В ПАКЕТЕ `java.util.concurrent`

(учебно-методическое пособие)

Ростов-на-Дону
2008

Чердынцева М.И.
Средства для разработки многопоточных программ в пакете java.util.concurrent: Учебно-методическое пособие. – Ростов-на-Дону, 2008. – 30 с.

Учебно-методическое пособие содержит материалы, необходимые студентам при изучении курса «Многопоточное программирование на Java». В пособии освещаются новые возможности для поддержки многопоточности, которые появились в Java начиная с версии J2SE JDK 5.0. Они объединены в пакет java.util.concurrent. В методическом пособии приведены описания и примеры использования некоторых из новых возможностей.

При изложении материала предполагается, что студенты знакомы с курсами «Технологии Java», «Многопоточное и распределенное программирование». Студенты должны уметь работать в одной из визуальных сред программирования на Java – NetBeans или Eclipse, или уметь использовать инструменты командной строки пакета разработчика – Java Development Kit.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для расширения возможностей многопоточного программирования, начиная с JDK 5.0, введен пакет `java.util.concurrent`. Использование пакета требует его импорта командой

```
import  java.util.concurrent.*;
```

В методических указаниях рассмотрены те средства, которые предназначены для реализации асинхронных вычислений, организации явных блокировок и условий ожидания потоками освобождения объектов блокировок. Рассмотрены также объекты–синхронизаторы, которые позволяют управлять набором взаимодействующих потоков на основе некоторых шаблонов.

МОДУЛЬ 1 АСИНХРОННЫЕ ВЫЧИСЛЕНИЯ

Цели и задачи модуля: Познакомиться с интерфейсами `Callable` и Future, возможностью создания на их основе асинхронных вычислений. Изучить на примере использование класса–оболочки `FutureTask`. Научиться повышать эффективность использования потоков путем создания пула потоков. Познакомиться с методами класса `Executors`.

1.1. Интерфейсы `Callable` и `Future`

Классы, реализующие интерфейс `Runnable`, представляют задачу, выполняющуюся асинхронно. При необходимости, результаты асинхронно решенной задачи могут быть переданы в вызвавший ее поток через разделяемые объекты. Для того, чтобы облегчить передачу результатов из асинхронной задачи в вызвавший поток, в пакете `java.util.concurrent` введены два интерфейса `Callable` и `Future`.

Интерфейс `Callable` является параметризованным (`generic` – интерфейс), в нем объявлен единственный метод `call()`, возвращающий результат, тип которого совпадает с типом параметра.

```
public interface Callable<V> {
    V call() throws Exception;
}
```

При реализации интерфейса `Callable` в методе `call()` обязательно должен присутствовать оператор return, возвращающий результат параметризованного типа. Так же, как и в случае с классами, реализующими интерфейс `Runnable`, вызов метода `call()` не производится, он вызывается классом, созданным на основании класса, реализующего интерфейс `Callable`, и может быть запущен на выполнение в отдельном потоке или в том же потоке, где и создан. Для управления режимами запуска классов, реализующих интерфейс `Callable`, как и классов, реализующих интерфейс `Runnable` в пакете `java.util.concurrent` реализован набор интерфейсов и классов на базе интерфейса `Executor`.

Поскольку метод `call()` не вызывается явно, невозможно получить результат его выполнения. Для работы с результатами асинхронно выполняющихся задач необходимо реализовать интерфейс `Future`, который описывает набор методов для работы с результатами асинхронных вычислений. Методы, объявленные в интерфейсе `Future` представлены в таблице 1.

Для того, чтобы использовать результаты асинхронно выполняющейся задачи имеется класс `FutureTask<V>`. Объект класса `FutureTask` создается на базе класса, реализующего интерфейс `Callable`. Этот класс реализует интерфейсы `Runnable` и `Future`. То есть его можно запускать на выполнение в отдельном потоке и получать результаты его выполнения по завершению работы.

Объект класса `FutureTask` может быть передан на выполнение любому классу, реализующему интерфейс `Executor` или запущен в отдельном потоке, как класс, реализующий интерфейс `Runnable`. Класс–

оболочка FutureTask представляет собой удобный механизм, превращающий Callable и в Future, и в Runnable.

Таблица 1. Методы интерфейса java.util.concurrent.Future<V>

V get() throws…	Возвращает результат асинхронных вычислений, выполнение блокируется до окончания вычислений.
V get(long timeout, TimeUnit unit) throws…	Возвращает результат асинхронных вычислений, выполнение блокируется до окончания вычислений или до истечения указанного интервала времени.
boolean cancel(boolean mayInterrupt)	Пытается отменить выполнение задачи. Если задача уже запущена и параметр mayInterrupt равен true, она прерывается. Если вычисления еще не начаты, они не начнутся никогда. Если отмена задачи прошла успешно. Метод возвращает значение true.
boolean isCancelled()	Возвращает true, если задача была отменена до ее нормального завершения
boolean isDone()	Возвращает true, если выполнение задачи завершено, если выполнение было прекращено или если в процессе ее выполнения возникло исключение.

Пример. Приводимая ниже программа предназначена взята с незначительными изменениями из /2/. В программе выполняется просмотр всех файлов каталога, включая и все вложенные в него подкаталоги. В процессе просмотра проверяется, содержит ли каждый файл заданную строку символов (ключевое слово). Поиск ключевого слова сопровождается подсчетом количества файлов, в которых оно найдено. Подсчет ведется отдельно для каждого каталога и общим итогом по всем файлам и подкаталогам.

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;
public class FutureTest {
   public static void main (String[] args) {
```

```
    Scanner in = new Scanner(System.in);
    System.out.print("Введите базовую директорию :");
    String directory = in.nextLine();
    System.out.print("Введите искомое слово:");
    String keyword = in.nextLine();
    MatchCounter counter =
        new MatchCounter (new File(directory),keyword);
    FutureTask <Integer> task =
            new FutureTask <Integer> (counter);
    Thread t = new Thread (task);
    t.start();
    try {
        System.out.println(task.get() +
            "  найдено файлов");
    }catch (ExecutionException e) {
        e.printStackTrace();
    }catch (InterruptedException e) {}
}}
/**  Данный класс подсчитывает файлы, находящиеся в каталоге и
его подкаталогах, в которых имеется ключевое слово.
*/
class MatchCounter implements Callable<Integer> {
   public MatchCounter ( File dir, String key) {
      directory = dir;
      keyword = key;
   }
//Метод call() интерфейса Callable аналогичен методу run()
// интерфейса Runnable, но возвращает результат
   public Integer call() {
    count = 0;
    try {
```

```
        File[] files =directory.listFiles();
        ArrayList <Future <Integer>> results =
                new ArrayList <Future <Integer>>();
        for (File file : files)
            if (file.isDirectory()) {
                MatchCounter counter =
                    new MatchCounter (file,keyword);
                FutureTask <Integer> task =
                    new FutureTask Integer>(counter);
        results.add(task);
        Thread t = new Thread(task);
        t.start();
       }else {
        if (search(file)) count++;
       }
  for (Future <Integer> result : results)
  try {
        count += result.get();
  }catch (ExecutionException e) {
        e.printStackTrace();
  }}
  catch (InterruptedException e) {}
  return count;
}
public boolean search (File f) {
  try {
  Scanner in = new Scanner(new
                           FileInputStream(f));
        boolean found = false;
        while (!found && in.hasNextLine()) {
```

```
                String line = in.nextLine();
                if (line.contains(keyword))
                    found = true;
            }
        in.close();
        return found;
        }catch (IOException e) {
            return false;
        }}
    private File directory;
    private String keyword;
    private int count;
}
```

Проектные задания

1. Измените задачу так, чтобы каждый поток считал для файла количество строк, в которых встречается ключевой слово.

2. Разработайте многопоточное приложение, которое подсчитывает, сколько раз встретилось ключевое слово в заданном файле. Потоки должны получать для анализа отдельную строку.

Тесты рубежного контроля

Вопрос 1. Интерфейс `Callable` отличается от `Runnable` тем, что:

Вариант 1	Запускается не в потоке
Вариант 2	Кроме метода run() имеет метод call()
Вариант 3	Запускает поток на выполнение методом call(), который возвращает результат

Вопрос 2. Для того, чтобы получить результаты метода `call()` нужно

Вариант 1	Реализовать интерфейс Future и использовать метод get()
Вариант 2	Вызвать метод call() и указать имя объекта для результата
Вариант 3	В методе call() поместить результат в совместно используемую переменную

МОДУЛЬ 2 СРЕДСТВА БЛОКИРОВКИ

Цели и задачи модуля: Рассмотреть возможности классов блокировки, появившихся в пакете `java.util.concurrent.locks`, ознакомиться с понятиями справедливой блокировки и блокировок чтения и записи. Рассмотреть возможность использовать объекты условий для согласованной работы потоков.

2.1. Объекты блокировки

Вместо использования синхронизованных методов и блоков, в пакете `java.util.concurrent.lock` вводится интерфейс `Lock` и класс `ReentrantLock`, реализующий этот интерфейс. Метод `lock()` позволяет потоку осуществить блокировку. Как только поток заблокирует объект, никакой другой поток не сможет сделать этого до тех пор, пока не будет выполнен метод `unlock()`, снимающий блокировку.

Типичная схема защиты кода с помощью класса `ReentrantLock` выглядит так:

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

Если поток не может заблокировать объект, он переходит в состояние ожидания и остается в нем до разблокирования объекта. Можно использовать более гибкий механизм, используя метод `tryLock()`. Этот метод пытается получить блокировку и возвращает значение `true` при успехе. Если же блокировка невозможна, метод прекращает работу и возвращает значение `false`. При этом поток имеет возможности выполнить какие-нибудь другие действия. В параметрах метода `tryLock()` можно задать время, в течение которого можно ожидать возможность блокировки.

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

2.2. Блокировки чтения – записи

В пакете `java.utils.concurrent.lock`, кроме класса `ReentrantLock` имеется класс `ReentrantReadWriteLock`. Этот класс позволяет сформировать объекты блокировки разных уровней. Блокировка уровня «чтение» позволяет нескольким потокам совместно читать, т.е. использовать блокировку этого же уровня. Блокировка уровня «записи» исключает одновременное обращение других записывающих и читающих потоков. Как следует из названия, такие блокировки чаще всего связаны с выполнение операций чтения/записи.

Для выполнения блокировки чтения и записи необходимо выполнить следующие действия.

1. Создать объект ReentrantReadWriteLock.

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

2. Сформировать на его основе отдельные объекты для блокировки чтения и записи.

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

3. Использовать блокировку чтения там, где возможен совместный доступ.

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

4. Использовать блокировку записи там, где необходим эксклюзивный доступ.

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

2.3. Объекты условий

Использование объектов блокировки является альтернативой по отношению к блокам синхронизации и синхронизованным методам. Понятно, что эти средства не стоит использовать совместно. В тоже время, обмен сообщениями между потоками возможен только внутри синхронизованных методов или блоков. По аналогии с методами класса Object - wait(), notify() и notifyAll() в пакете java.utils.concurrent.lock вводится класс Condition, имеющий методы await(), signal() и signalAll(). Чтобы получить объект условий, связанный с объектом блокировки, используется метод newCondition().

```
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();
}
```

Подробную информацию о пакете `java.utils.concurrent.lock` можно получить из официальной документации фирмы Sun Microsystems /6/.

Пример. Рассматриваемая ниже программа /2/ имитирует работу банка со счетами. В программе случайным образом генерируется операция по переводу денег с одного счета на другой. Каждый счет обслуживается отдельным потоком. В процессе операции произвольная сумма снимается со счета, обслуживаемого данным потоком, и пересылается на другой счет. Если на счете нет указанной суммы, то перевода не производится.

При выполнении этих операций неизвестно, сколько денег содержится на каждом счете в заданный момент. Однако известна общая сумма, которая должна оставаться неизменной, поскольку операции происходят в пределах одного банка.

Запускаемый класс приложения выглядит следующим образом:

```
/* для использования объектов блокировки и объектов условий
   необходимо будет подключить пакет
import java.util.concurrent.locks;
*/
public class SynchBankTest{
    public static void main (String[] args){
```

```
    Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
    int i;
    long t = System.currentTimeMillis();
    for (i=0; i< NACCOUNTS; i++) {
      TransferRunnable r =
            new TransferRunnable(b,i,INITIAL_BALANCE);
      Thread t = new Thread(r);
      t.start();
      t.join();
    }
    long tt = System.currentTimeMillis() - t;
    System.out.println(”Total time =”+tt);
}
public static final int NACCOUNTS = 10;
public static final double INITIAL_BALANCE = 1000;
}
```

В классе SynchBankTest создается объект – банк, для которого определяется количество счетов и начальный баланс (сумма) для каждого счета. После этого, для каждого счета создается обслуживающий его объект класса TransferRunnable,который должен выполнить несколько операций перевода с данного счета на некоторый другой. Для выполнения такого обслуживания запускается отдельный поток.

Рассмотрим теперь описание класса, моделирующего работу банка.

```
class Bank {
/**
    Конструктор объекта, представляющего банк
*/
public Bank (int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i<accounts.length; i++)
```

```
        accounts[i]    = initialBalance;
/** Для обеспечения синхронизации и согласования работы потоков
     необходимо создать объект блокировки
     bankLock = new ReentrantLock();
     и получить связанный с ни объект условия
     sufficientFunds = bankLock.newCondition();
*/
  }
/** Метод осуществляющий перевод денег с одного счета на другой
     В представленном варианте не синхронизован
     Для обеспечения синхронизации необходимо заменить
     на описанный ниже вариант
*/
public void transfer(int from,int to,double amount){
     if (accounts[from]<amount) return;
     System.out.print (Thread.currentThread());
     accounts[from]-=amount;
     System.out.printf(" %10.2f from %d to %d",
          amount,from,to);
     accounts[to]+=amount;
     System.out.printf("Total Balance: %10.2f %n",
                    getTotalBalance());
}
/* измененный вариант метода для перевода денег
public void  transfer(int from, int to, double amount)
                    throws InterruptedException
{
// получаем блокировку
     bankLock.lock();
     try {
// далее идет критичный фрагмент кода
          while (accounts[from] < amount)
```

```
            sufficientFunds.await();//переход в режим ожидания
            System.out.print (Thread.currentThread());
            accounts[from]-=amount;
            System.out.printf(" %10.2f from %d to %d",
                        amount,from,to);
            accounts[to]+=amount;
            System.out.printf("Total Balance: %10.2f %n",
                    getTotalBalance());
            sufficientFunds.signalAll();//разблокировка
                                        //ожидающих потоков
        }
        finally
        {
            bankLock.unlock();// операцию разблокирования нужно
                              // выполнить даже при возникновении
                              // исключения
        }
    }
*/
/** Определение суммы средств на всех счетах
    Для корректной работы также требует синхронизации
    Измененный код метода приведен далее
*/
public double getTotalBalance(){
    double sum=0;
    for (double a:accounts)
        sum+=a;
    return sum;
}
/** Вариант метода определения суммы средств на всех счетах,
    обеспечивающий синхронизацию
```

```
  public double getTotalBalance(){
    bankLock.lock();
    try {
        double sum=0;
        for (double a:accounts)
            sum+=a;
        return sum;
    }
    finally{
        bankLock.unlock();
    }
}
*/
/** Определение количества счетов в банке
*/
public int size(){
    return accounts.length;
}
private final double[] accounts;
/* для выполнения блокировок добавляется объект блокировки
    private Lock bankLock;
   для определения условия согласованности переводов со счета на
   счет добавляется объект условие
    private Condition sufficientFunds;
*/
}
```

Приведенное описание класса Bank изначально не содержит никаких средств, обеспечивающих согласованность работы потоков. Выполнив приложение в таком виде можно обнаружить ситуацию гонки потоков, которая будет выражаться, как в несогласованной выдаче сообщений, приходящих от разных потоков, осуществляющих перевод средств, так и в

неверном балансе суммы всех счетов в банке. Чтобы поучить корректно работающий вариант, следует внести в код исправления в соответствии с приведенными в нем комментариями.

Собственно поток, осуществляющий перевод средств со счета на счет описывается следующим классом:

```
class TransferRunnable implements Runnable{
/**
     Конструктор объекта, реализующего интерфейс Runnable
*/
public TransferRunnable(Bank b, int from, double max){
     bank = b;
     fromAccount = from;
     maxAmount = max;
}
public void run() {
     try {
     for (int j=0; j<MAX_CHANGE; j++){
       int toAccount =(int)(bank.size()*Math.random());
       double amount = maxAmount * Math.random();
       bank.transfer(fromAccount, toAccount, amount);
       Thread.sleep((int) (DELAY * Math.random()));
     }
     } catch (InterruptedException e ){}
}
private Bank bank;
private int fromAccount;
private double maxAmount;
private int DELAY = 100;
  private int MAX_CHANGE=5;
}
```

Проектные задания

3. Модифицируйте код приложения, разделив блокировки в разных методах класса Bank на блокировку чтения и блокировку записи. Проверьте, что это не повлечет возникновения гонки потоков. Оцените время выполнения приложения для этого случая.

Тесты рубежного контроля

Вопрос 1. Чтобы использовать объект блокировки необходимо:

Вариант 1	Создать переменную – объект блокировки
Вариант 2	Использовать синхронизованный метод
Вариант 3	Вызвать метод lock() для переменной – объекта блокировки

Вопрос 2. Отметьте какие из ниже перечисленных методов являются методами класса `Condition`

Вариант 1	await()
Вариант 2	lock()
Вариант 3	notify()
Вариант 4	signal()
Вариант 5	signalAll()
Вариант 6	wait()

МОДУЛЬ 3 СРЕДСТВА СИНХРОНИЗАЦИИ

Цели и задачи модуля: Познакомиться с возможностями появившихся в пакете `java.util.concurrent` классов–синхронизаторов. Научиться использовать синхронизаторы для организации стандартных типов взаимодействия потоков.

3.1. Семафоры

Семафоры (`Semaphore`)организуют взаимодействие потоков на основе набора разрешений. Чтобы пройти семафор (продолжить свою работу), поток запрашивает разрешение, вызывая метод `acquire()`. Для каждого семафора доступно лишь фиксированное число разрешений. Число разрешений семафора определяет количество потоков, которые могут пройти семафор.

Потоки могут не только пытаться пройти семафор, но и выдавать другим потокам разрешения. Для этого вызывается метод `release()`. Любой поток может выдать любое количество разрешений в пределах максимально допустимого для семафора.

Семафор со счетчиком разрешений равным 1 можно использовать в качестве затвора, который открывается и закрывается другим потоком.

Пример. Рассмотрим вариант решения задачи о читателях и писателях. Обмен информацией между читателями и писателями осуществляется через циклический массив заданного размера. В качестве механизма синхронизации используются семафоры.

Один из семафоров управляет писателями, он рассчитан на количество разрешений, равное размеру массива. В дальнейшем, разрешения будут выдавать читатели, после прочтения информации из массива.

Второй семафор управляет читателями. Он рассчитан на одно разрешение. После, того как писатель помещает данные в массив, он выдает очередное разрешение для семафора читателей.

Класс, который организует хранилище данных в виде массива, выглядит так:

```
import java.util.concurrent.*;
public class SmartQ {
    private SmartQRealization impl;
    private Semaphore sWrite;
    private Semaphore sRead;
    public SmartQ(int n) {
        sWrite = new Semaphore(n);
        sRead = new Semaphore(1);
        impl = new SmartQRealization(n);
    try{
//семафор сразу имеет разрешение, заберем его, чтобы читатель не
//смог воспользоваться им раньше, чем писатель запишет хотя бы
//одно значение
        sRead.acquire();
     } catch(InterruptedException e){};
   }
public void push(int i) throws InterruptedException {
        sWrite.acquire();//запрос разрешения на запись
        synchronized(impl) {
            impl.push(i);
            sRead.release();//выдано разрешение на чтение
        }
}
public int pop() throws InterruptedException {
        sRead.acquire();//запрос разрешения на чтение
        synchronized(impl) {
            int tmp = impl.pop();
            sWrite.release();//выдано разрешение на запись.
```

```
            return tmp;
      }

}
```

Классы писателя и читателя выглядят следующим образом:

```
public class Reader implements Runnable {
  private SmartQ q;
  private String name;//для отладки
  private int readTime;
  private boolean stopFlag;
  public Reader(String name, SmartQ q, int readTime) {
        this.name = name;
        this.q = q;
        this.readTime = readTime;
        stopFlag = false;
  }
  public void stop() { stopFlag = true; }
  public void run() {
        try {
            while (!stopFlag) {
                System.out.println("   "+name+
                     "       get ="+q.pop() );
                Thread.sleep(readTime);
            }
        } catch(Exception e) {}
  }
public class Writer implements Runnable {
  private SmartQ q;
  private int writeTime;
```

```
  private boolean stopFlag;
  private int base, step;
  public Writer(SmartQ q,int writeTime,
                int base,int step) {
    this.q = q;
    this.writeTime = writeTime;
    stopFlag = false;
    this.base = base;
        this.step = step;
  }
  public void stop() { stopFlag = true; }
  public void run() {
    try {
      int i = base;
      while (!stopFlag) {
        System.out.println("put="+i);
        q.push(i);
        i += step;
        Thread.sleep(writeTime);
      }
    } catch(Exception e) {}
}}
```

Наконец, рассмотрим класс, который создает хранилище данных, создает и запускает потоки. В этом классе создается несколько писателей и несколько читателей. Они организуются в списки на базе массива.

```
import java.util.*;
public class Main {
    static final int nWriters = 10;
    public static void main(String[] args) {
        SmartQ q = new SmartQ(10);
```

```
        List<Reader> listR = new ArrayList<Reader>();
        List<Writer> listW = new ArrayList<Writer>();
        for ( int i = 0;  i < nWriters;  ++i) {
            Reader r = new Reader("#"+i,q, 10);
            new Thread(r).start();
            listR.add(r);
            Writer w = new Writer(q, 10, i, nWriters);
            new Thread(w).start();
            listW.add(w);
        }
        try{
            Thread.sleep(100);
        } catch(Exception e) {}
        for(Reader r : listR) {
            r.stop();
        }
        for(Writer w : listW) {
            w.stop();
        }
    }
}
```

3.2. Барьеры

Взаимодействие потоков под названием «барьер» реализует класс CyclicBarrier. Пусть несколько потоков решают совместную задачу. Когда все потоки завершатся, полученные ими результаты могут быть объединены. Каждый поток, после завершения своей части работы останавливается перед барьером. После того, как барьер достигнут все потоки, они могут объединить свои результаты и продолжить работу.

Каждый барьер создается с расчетом на то количество потоков, котрые должны его достигнуть.

```
int nthreads = ...;
CyclicBarier br = new CyclicBarier(nthreads);
```

Поток, когда он достигает барьера вызывает метод `await()`.

```
public void run() {
    doWork();
    br.await();
. . .
}
```

Когда метод `await()` вызовут `nthreads` потоков, барьер будет снят. После этого начнет выполняться код в методе `run()` потока, после `await()`.

Барьер называется циклическим, потому что его можно повторно использовать после освобождения ожидающих потоков.

При необходимости можно задать необязательное действие барьера, которое выполнится, когда все потоки достигнут барьера, например, объединить результаты работы всех потоков.

```
Runnable brAction = . . . ;
CyclicBarier br = new CyclicBarier(nthreads, brAction);
```

Если из-за ошибки возможна ситуация, когда нужное количество потоков не смогут достигнуть барьера, можно предусмотреть возможность разрушения барьера. Для этого в методе `await()` хотя бы одного из потоков, достигшего барьера должен быть установлен тайм-аут. По истечении его, барьер разрушается и все достигшие барьера потоки выводятся из состояния ожидания и получают исключение `BrokenBarrierException`.

Пример. Чаще всего для демонстрации использования барьеров рассматривают задачи обработки матриц. При этом потоки обрабатывают

отдельные части матрицы – строки, полосы, блоки. По окончании работы всех потоком выполняется какая–то итоговая обработка. После этого процесс может быть повторен итеративно. В приводимом ниже примере выполнятся параллельное суммирование строк треугольной матрицы. Перед запуском потоков создается циклический барьер, ожидающий их завершения. Для барьера определено действие, которое выполнится, когда все потоки достигнут барьера.

```
import java.util.concurrent.*;
public class Summary {
    private static int matrix[][] = {
        {1},
        {2, 2},
        {3, 3, 3},
        {4, 4, 4, 4},
        {5, 5, 5, 5, 5}
    };
    private static int results[];
    private static class Summer extends Thread {
        int row;
        CyclicBarrier barrier;
        Summer(CyclicBarrier barrier, int row) {
            this.barrier = barrier;
            this.row = row;
        }
        public void run() {
        int columns=matrix[row].length;//размер строки матрицы
        int sum = 0;
        for (int i=0; i<columns; i++) {
            sum += matrix[row][i];
        }
```

```
            results[row] = sum; //готовим для итоговой суммы
            System.out.println(
            "Results for row " + row + " are : " + sum);
// ожидаем завершения остальных потоков
            try {
                barrier.await();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            } catch (BrokenBarrierException ex) {
                ex.printStackTrace();
            }
        }
}
public static void main(String args[]) {
        final int rows = matrix.length;
        results = new int[rows];
//определяем действие барьера, которое выполнится, когда все
//потоки достигнут барьера
        Runnable merger = new Runnable() {
            public void run() {
                int sum = 0;
                for (int i=0; i<rows; i++) {
                    sum += results[i];
                }
                System.out.println("Results are: " + sum);
            }
        };
//циклический барьер создается по числу запускаемых потоков
// в данной задаче число потоков равно числу строк в матрице
CyclicBarrier barrier= new CyclicBarrier(rows, merger);
//запускаем отдельный поток для каждой строки
```

```
    for (int i=0; i<rows; i++) {
        new Summer(barrier, i).start();
    }
    System.out.println("Waiting...");
}}
```

Проектные задания

4. В задаче обработки матрицы измените размер матрицы. Как это отразится на выдаваемых результатах? Чтобы повторно использовать циклический барьер, вызовите для него методы reset() и повторно запустите потоки на выполнение. Что изменится в результатах выдачи?

5. Создайте многопоточное приложение с циклическим барьером, выполняющее любую обработку матрицы итерационно. Определите, какие действия, и с какими частями матрицы можно выполнять параллельно. Определите также, необходима ли какая-то подготовка перед следующей итерацией.

Тесты рубежного контроля

Вопрос 1. Синхронизатор «семафор» работает по принципу:

Вариант 1	Автоматического переключения двух потоков
Вариант 2	Запроса и выдачи разрешений потоками
Вариант 3	Ожидания группы потоков перед семафором, пока количество ожидающих потоков не станет равно заданному

Вопрос 2. Синхронизатор «барьер» используется, когда нужно

Вариант 1	Заставить несколько потоков ожидать, пока завершатся остальные
Вариант 2	Установить точку встречи заданного числа потоков для объединения их результатов
Вариант 3	Прекратить выполнение потоков
Вариант 4	Завершить работу приложения, когда один из потоков достигнет барьера

Вопросы для повторения

1. Что такое объект блокировки, и какие средства имеются для явного выполнения операций блокировки/разблокировки?
2. Что такое блокировки чтения – записи? Для чего они используются?
3. Как объекты условий организуют взаимодействие потоков?
4. В чем преимущества использования наборов данных из пакета `java.util.concurrent`?
5. Что произойдет, если для семафора будет выдано больше разрешений, чем предусмотрено при его создании?
6. Может ли поток, ожидающий разрешения у семафора, сам выдать разрешение?
7. Для чего предназначены классы `Callable` и `Future`?
8. Какие синхронизаторы имеются в пакете `java.util.concurrent`?
9. В чем отличие методов `lock()` и `tryLock()` объектов блокировки?
10. Что произойдет, если поток ожидающий у барьера будет аварийно завершен (прерван)?
11. Для чего используется семафор с одним разрешением?
12. Как организовать взаимодействие потоков, после того как они прошли барьер?
13. Какие задачи решает класс, реализующий интерфейс `Executor`?

ЛИТЕРАТУРА

1. Хабибулин И.Ш. Самоучитель Java 2. – СПб.: БХВ-Петербург, 2005. – 720 с.
2. Хорсман К., Корнелл Г. Java 2. Библиотека профессионала, том II. Тонкости программирования. 7-е изд. – М.: Издательский дом «Вильямс», 2007. – 1168 с.
3. Эккель Брюс. Философия Java. Библиотека программиста. 3-е изд. – Спб.: Питер, 2003. – 971 с.
4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003. – 512 с.
5. Шилдт Г., Холмс Д. Искусство программирования на Java.–М.: Издательсикй дом «Вильямс», 2005.

Ресурсы в сети Интернет

6. http://java.sun.com/docs/
7. http://www.citforum.ru
8. http://www.rusdoc.ru
9. http://www.osp.ru
10. http://javable.com
11. http://javaworld.com