Оглавление

Java 8 2
java.io 2
interface Serializable 2
java.lang 3
interface CharSequence 3
interface Comparable<T> 6
class Object 7
class String 9
Многопоточность 10
Многопоточность в Java: основы 10
Многопоточность в Java: средства стандартной библиотеки 31
Новая тема 55

Java 8

java.io

interface Serializable

```
package java.io;

public interface Serializable
```

java.lang

interface CharSequence

package java.lang;

import

```
1.  java.util.NoSuchElementException;
2.  java.util.PrimitiveIterator;
3.  java.util.Spliterator;
4.  java.util.Spliterators;
5.  java.util.function.IntConsumer;
6.  java.util.stream.IntStream;
7.  java.util.stream.StreamSupport;
```

public interface **CharSequence**

методы

```
1.  int length();
1.  char charAt(int index);
1.  CharSequence subSequence(int start, int end);
1.  String toString();
```

методы дефолтные

```
1.  public default IntStream chars() {
2.    class CharIterator
3.        implements PrimitiveIterator.OfInt {
4.      int cur = 0;
5.
6.      public boolean hasNext() {
7.        return cur < length();
8.      }
9.
10.     public int nextInt() {
11.       if (hasNext()) {
12.         return charAt(cur++);
13.       } else {
14.         throw new NoSuchElementException();
15.       }
16.     }
17.
18.     @Override
19.     public void forEachRemaining(
```

```
          IntConsumer block) {
        for (; cur < length(); cur++) {
          block.accept(charAt(cur));
        }
      }
    }

    return StreamSupport.intStream(() ->
        Spliterators.spliterator(
            new CharIterator(),
            length(),
            Spliterator.ORDERED),
        Spliterator.SUBSIZED | Spliterator.SIZED
            | Spliterator.ORDERED,
        false);
  }
```

```
public default IntStream codePoints() {
  class CodePointIterator
      implements PrimitiveIterator.OfInt {
    int cur = 0;

    @Override
    public void forEachRemaining(
        IntConsumer block) {
      final int length = length();
      int i = cur;
      try {
        while (i < length) {
          char c1 = charAt(i++);
          if (!Character.isHighSurrogate(c1)
              || i >= length) {
            block.accept(c1);
          } else {
            char c2 = charAt(i);
            if (Character.isLowSurrogate(c2)) {
              i++;
              block.accept(
                  Character.toCodePoint(c1, c2));
            } else {
              block.accept(c1);
            }
          }
        }
      } finally {
```

```
29.           cur = i;
30.         }
31.       }
32.
33.       public boolean hasNext() {
34.         return cur < length();
35.       }
36.
37.       public int nextInt() {
38.         final int length = length();
39.
40.         if (cur >= length) {
41.           throw new NoSuchElementException();
42.         }
43.         char c1 = charAt(cur++);
44.         if (Character.isHighSurrogate(c1) && cur
45.             < length) {
46.           char c2 = charAt(cur);
47.           if (Character.isLowSurrogate(c2)) {
48.             cur++;
49.             return Character.toCodePoint(c1, c2);
50.           }
51.         }
52.         return c1;
53.       }
54.     }
55.
56.     return StreamSupport.intStream(() ->
57.         Spliterators.spliteratorUnknownSize(
58.             new CodePointIterator(),
59.             Spliterator.ORDERED),
60.             Spliterator.ORDERED,
61.             false);
62.   }
```

interface Comparable<T>

package java.lang;

import

```
1.  import java.util.*;
```

public interface **Comparable<T>**

методы

```
1.  public int compareTo(T o);
```

class Object

package java.lang;

public class **Object**

конструкторы по умолчанию

методы

```
1.  public final native Class<?> getClass();
1.  public native int hashCode();
1.  public boolean equals(Object obj) {
2.    return (this == obj);
3.  }
1.  protected native Object clone()
2.      throws CloneNotSupportedException;
1.  public String toString() {
2.    return getClass().getName() + "@"
3.        + Integer.toHexString(hashCode());
4.  }
1.  public final native void notify();
1.  public final native void notifyAll();
1.  public final native void wait(long timeout)
2.      throws InterruptedException;
1.  public final void wait(long timeout, int nanos)
2.      throws InterruptedException {
3.    if (timeout < 0) {
4.      throw new IllegalArgumentException(
5.          "timeout value is negative");
6.    }
7.
8.    if (nanos < 0 || nanos > 999999) {
9.      throw new IllegalArgumentException(
10.         "nanosecond timeout value out of range");
11.   }
12.
13.   if (nanos > 0) {
14.     timeout++;
15.   }
16.
17.   wait(timeout);
18. }
1.  public final void wait()
```

```
2.      throws InterruptedException {
3.    wait(0);
4.  }
1.  protected void finalize() throws Throwable { }
```

методы приватные

```
1.  private static native void registerNatives();
2.
3.  static {
4.    registerNatives();
5.  }
```

xv

sad

Многопоточность

Многопоточность в Java: основы

Мотивация

- Одновременное выполнение нескольких действий (например, отрисовка пользовательского интерфейса и передача файлов по сети);
- Ускорение вычислений (при наличии нескольких вычислительных ядер).

Закон Амдала

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

S — ускорение;

P — доля вычислений, которые возможно распараллелить;

N — количество вычислительных ядер.

Параллелизм в Java

Запуск нескольких JVM на одной или на разных компьютерах:

- нет общей памяти;
- взаимодействие через файловую систему или сетевое соединение.

Запуск нескольких потоков внутри JVM:

- есть общая память;

- обширная поддержка в языке и стандартной библиотеке.

Проблемы параллельных программ

- Гонка (race condition) — проблемы при использовании общей памяти, решение — эксклюзивный доступ к данным (примитивы синхронизации);
- Взаимная блокировка (deadlock).

java.lang.Thread

Потоки представлены экземплярами класса java.lang.Thread.

Основные методы:

- String getName();
- long getId();
- boolean isDeamon();
- StackTraceElement[] getStackTrace();
- ThreadGroup getThreadGroup().

Thread dump

Список всех потоков с их состояниями и stack trace'ами (кнопка в IDE (значок фотоаппарата в меню run)).

Создание потока

1-й способ

```
1.  public static void main() {
2.    Thread thread1 = new NewThread();
3.    thread1.start();
```

```
}
```

```
public class NewThread extends Thread {

  @Override
  public void run() {
    ...
  }
}
```

или

```
public static void main() {
  Thread thread2 = new Thread() {
    @Override
    public void run() {
      ...
    }
  };

  thread2.start();
}
```

2-й способ

```
public static void main() {
  Runnable runnable = () -> ...;
  Thread thread3 = new Thread(runnable);
  thread3.start();
}
```

Жизненный цикл потока

- создание объекта Thread;
- запуск (thread.start());
- работа (выполняется метод run(), thread.isAlive() == true);
- завершение (метод run() закончился или бросил исключение);
- завершенный поток нельзя перезапустить.

Примеры

```
public static void main() {
```

```
  for (int i = 0; i < 10; i++) {
    new HelloThread().start();
  }
  System.out.println("main поток");
}

private static class HelloThread extends Thread {
  @Override
  public void run() {
    System.out.println(getName());
  }
}
```

```
Thread-0
Thread-2
Thread-1
Thread-4
Thread-3
Thread-5
Thread-6
main поток
Thread-8
Thread-7
Thread-9
```

```
public static void main() {
  for (int i = 0; i < 10; i++) {
    new Thread(new HelloRunnable()).start();
  }
  System.out.println("main поток");
}

private static class HelloRunnable
    implements Runnable {
  @Override
  public void run() {
    System.out.println(
        Thread.currentThread().getName());
  }
}
```

```
Thread-1
main поток
Thread-7
Thread-5
```

```
 5.  Thread-3
 6.  Thread-0
 7.  Thread-2
 8.  Thread-9
 9.  Thread-8
10.  Thread-6
11.  Thread-4
```

```
 1.  public static void main() {
 2.    for (int i = 0; i < 10; i++) {
 3.      new Thread(() ->  System.out.println(
 4.              Thread.currentThread().getName()))
 5.          .start();
 6.    }
 7.    System.out.println("main поток");
 8.  }
```

```
 1.  Thread-0
 2.  Thread-5
 3.  Thread-1
 4.  main поток
 5.  Thread-3
 6.  Thread-6
 7.  Thread-4
 8.  Thread-7
 9.  Thread-8
10.  Thread-9
11.  Thread-2
```

С таким вариантом нужно быть осторожным. В данном случае нет никакого состояния, поэтому так делать можно.

```
 1.  public static void main() {
 2.    HelloRunnable helloRunnable =
 3.        new HelloRunnable();
 4.    for (int i = 0; i < 10; i++) {
 5.      new Thread(helloRunnable).start();
 6.    }
 7.    System.out.println("main поток");
 8.  }
 9.
10.  private static class HelloRunnable
11.      implements Runnable {
```

```
12.     @Override
13.     public void run() {
14.       System.out.println(
15.           Thread.currentThread().getName());
16.     }
17.   }
```

```
 1.  Thread-0
 2.  Thread-4
 3.  Thread-5
 4.  Thread-3
 5.  Thread-6
 6.  main поток
 7.  Thread-2
 8.  Thread-1
 9.  Thread-8
10.  Thread-7
11.  Thread-9
```

Прерывание потока

- thread.interrapt();
- если поток находится в ожидании (sleep, join wait), то ожидание прерывается исключением InterruptedException;
- Иначе у потока просто устанавливается флаг interrupted: флаг проверяется методами interrupted() и isInterrupted(), проверять флаг и завершать поток надо самостоятельно;
- thread.join().

Примеры

```
 1.  public static void main()
 2.      throws InterruptedException {
 3.    Thread worker = new WorkerThread();
 4.    Thread sleeper = new SleeperThread();
 5.
 6.    System.out.println("Starting threads");
 7.    worker.start();
```

```
    sleeper.start();

    Thread.sleep(100L);

    System.out.println("Interrupted threads");
    worker.interrupt();
    sleeper.interrupt();

    System.out.println("Joining threads");
    worker.join();
    sleeper.join();

    System.out.println("All done");
  }

  private static class WorkerThread
      extends Thread {

    @Override
    public void run() {
      long sum = 0;
      for (int i = 0; i < 1_000_000_000; i++) {
        sum += i;
        if (i % 100 == 0 && isInterrupted()) {
          System.out.println("i = " + i);
          break;
        }
      }
    }
  }

  private static class SleeperThread
      extends Thread {

    @Override
    public void run() {
      try {
        Thread.sleep(10_000L);
      } catch (InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }
```

```
Starting threads
```

```
2.  Interrupted threads
3.  Joining threads
4.  Sleep interrupted
5.  i = 92633100
6.  All done
```

Программа зависает и ждет выполнение потоков.

```
1.  public static void main()
2.      throws InterruptedException {
3.    Thread worker = new WorkerThread();
4.    Thread sleeper = new SleeperThread();
5.
6.    System.out.println("Starting threads");
7.    worker.start();
8.    sleeper.start();
9.
10.   Thread.sleep(100L);
11.
12. //  System.out.println("Interrupted threads");
13. //  worker.interrupt();
14. //  sleeper.interrupt();
15. //
16. //  System.out.println("Joining threads");
17. //  worker.join();
18. //  sleeper.join();
19.
20.   System.out.println("All done");
21. }
22.
23. private static class WorkerThread
24.     extends Thread {
25.
26.   @Override
27.   public void run() {
28.     long sum = 0;
29.     for (int i = 0; i < 1_000_000_000; i++) {
30.       sum += i;
31.       if (i % 100 == 0 && isInterrupted()) {
32.         System.out.println("i = " + i);
33.         break;
34.       }
35.     }
36.   }
```

```
37.  }
38.
39.  private static class SleeperThread
40.      extends Thread {
41.
42.    @Override
43.    public void run() {
44.      try {
45.        Thread.sleep(10_000L);
46.      } catch (InterruptedException e) {
47.        System.out.println("Sleep interrupted");
48.      }
49.    }
50.  }
```

```
1.  Starting threads
2.  All done
```

Программа больше не зависает, т. к. потоки стали демонами.

```
1.  public static void main()
2.      throws InterruptedException {
3.    Thread worker = new WorkerThread();
4.    worker.setDaemon(true);
5.    Thread sleeper = new SleeperThread();
6.    sleeper.setDaemon(true);
7.
8.    System.out.println("Starting threads");
9.    worker.start();
10.   sleeper.start();
11.
12.   Thread.sleep(100L);
13.
14.   System.out.println("All done");
15. }
16.
17. private static class WorkerThread
18.     extends Thread {
19.
20.   @Override
21.   public void run() {
22.     long sum = 0;
23.     for (int i = 0; i < 1_000_000_000; i++) {
24.       sum += i;
25.       if (i % 100 == 0 && isInterrupted()) {
```

```
26.             System.out.println("i = " + i);
27.             break;
28.           }
29.         }
30.       }
31.   }
32.
33.   private static class SleeperThread
34.       extends Thread {
35.
36.     @Override
37.     public void run() {
38.       try {
39.         Thread.sleep(10_000L);
40.       } catch (InterruptedException e) {
41.         System.out.println("Sleep interrupted");
42.       }
43.     }
44.   }
```

Возможности встроенной синхронизации

- взаимное исключение (пока один поток что-то делает, другие не могут ему помешать);
- ожидание и уведомление (поток ожидает уведомление от других потоков).

Ключевое слово synchronized

Синхронизированный метод (статический или нестатический)

```
1.  public synchronized void doSomething() {
2.    ...
3.  }
```

Синхронизированный блок внутри нестатического метода

```
1.  public void doSomething() {
2.    synchronized (this) {
3.      ...
```

```
4.    }
5.  }
```

Синхронизированный блок внутри статического метода

```
1.  public class C {
2.    public static void doSomething() {
3.      synchronized (C.class) {
4.        ...
5.      }
6.    }
7.  }
```

Синхронизация блоков — по монитору указанного объекта.

Синхронизация методов — по монитору текущего объекта (this).

Синхронизация статических методов — по монитору класса.

Один момент: два разных синхронизованных метода одного класса не могут выполнять несколькими потоками параллельно, т. к. экземпляр объекта один и тот же, а значит и монитор один и тот же. А если один из них статический, то могут исполняться в параллельных потоках.

Ожидания и уведомления

Допустимо только внутри synchronized.

Приостанавливает текущий поток (поток засыпает и освобождает блокировку текущего объекта): void wait, void wait(long

millis), void wait(long millis, int nanos).

Будит поток: void notify(), void notifyAll() (будет все потоки).

Примеры

```
 1. public static void main()
 2.     throws InterruptedException {
 3.   Account account = new Account(100_000);
 4.   System.out.println(
 5.       "Begin balance = " + account.getBalance());
 6.
 7.   Thread withdrawThread =
 8.       new WithdrawThread(account);
 9.   Thread depositThread =
10.       new DepositThread(account);
11.   withdrawThread.start();
12.   depositThread.start();
13.
14.   withdrawThread.join();
15.   depositThread.join();
16.
17.   System.out.println(
18.       "End balance = " + account.getBalance());
19. }
20.
21. private static class WithdrawThread
22.     extends Thread {
23.
24.   private final Account account;
25.
26.   private WithdrawThread(Account account) {
27.     this.account = account;
28.   }
29.
30.   @Override
31.   public void run() {
32.     for (int i = 0; i < 20_000; i++) {
33.       account.withdraw(1);
34.     }
35.   }
36. }
```

```
37.
38.  private static class DepositThread
39.      extends Thread {
40.
41.    private final Account account;
42.
43.    private DepositThread(Account account) {
44.      this.account = account;
45.    }
46.
47.    @Override
48.    public void run() {
49.      for (int i = 0; i < 20_000; i++) {
50.        account.deposit(1);
51.      }
52.    }
53.  }
```

```
 1.  public class Account {
 2.
 3.    private long balance;
 4.
 5.    public Account() {
 6.      this(0L);
 7.    }
 8.
 9.    public Account(long balance) {
10.      this.balance = balance;
11.    }
12.
13.    public long getBalance() {
14.      return balance;
15.    }
16.
17.    public void deposit(long amount) {
18.      checkAmountNonNegative(amount);
19.      balance += amount;
20.    }
21.
22.    public void withdraw(long amount) {
23.      checkAmountNonNegative(amount);
24.      if (balance < amount) {
25.        throw new IllegalArgumentException(
26.            "not enough money");
27.      }
```

```
28.       balance -= amount;
29.     }
30.
31.     private static void checkAmountNonNegative(
32.         long amount) {
33.       if (amount < 0) {
34.         throw new IllegalArgumentException(
35.             "negative amount");
36.       }
37.     }
38.   }
```

```
1.  Begin balance = 100000
2.  End balance = 104013
```

Чтобы это исправить помечаем методы synchronized.

```
1.  public class Account {
2.
3.    private long balance;
4.
5.    public Account() {
6.      this(0L);
7.    }
8.
9.    public Account(long balance) {
10.     this.balance = balance;
11.   }
12.
13.   public long getBalance() {
14.     return balance;
15.   }
16.
17.   public synchronized void deposit(long amount) {
18.     checkAmountNonNegative(amount);
19.     balance += amount;
20.   }
21.
22.   public synchronized void withdraw(long amount) {
23.     checkAmountNonNegative(amount);
24.     if (balance < amount) {
25.       throw new IllegalArgumentException(
26.           "not enough money");
27.     }
28.     balance -= amount;
```

```
29.     }
30.
31.     private static void checkAmountNonNegative(
32.         long amount) {
33.       if (amount < 0) {
34.         throw new IllegalArgumentException(
35.             "negative amount");
36.       }
37.     }
38.   }
```

```
1.  Begin balance = 100000
2.  End balance = 100000
```

Рекомендуется в synchronized помещать только маленький кусочек работы. В данном случае вся полезная работа оказывается в synchronized блоке, но в большинстве реальных ситуаций можно выделить маленький кусочек, требующий синхронизации.

```
1.  public void deposit(long amount) {
2.    checkAmountNonNegative(amount);
3.    synchronized (this) {
4.      balance += amount;
5.    }
6.  }
7.
8.  public void withdraw(long amount) {
9.    checkAmountNonNegative(amount);
10.   synchronized (this) {
11.     if (balance < amount) {
12.       throw new IllegalArgumentException(
13.           "not enough money");
14.     }
15.     balance -= amount;
16.   }
17. }
```

```
1.  public static void main()
2.      throws InterruptedException {
3.    Account account = new Account(0);
```

```
4.
5.    new DepositThread(account).start();
6.
7.    System.out.println(
8.        "Calling waitAndWithdraw() ...");
9.
10.   account.waitAndWithdraw(50_000_000);
11.
12.   System.out.println(
13.       "waitAndWithdraw() finished");
14. }
15.
16. private static class DepositThread
17.     extends Thread {
18.
19.   private final Account account;
20.
21.   private DepositThread(Account account) {
22.     this.account = account;
23.   }
24.
25.   @Override
26.   public void run() {
27.     for (int i = 0; i < 50_000_000; i++) {
28.       account.deposit(1);
29.     }
30.   }
31. }
```

```
1.  public class Account {
2.
3.    private long balance;
4.
5.    public Account() {
6.      this(0L);
7.    }
8.
9.    public Account(long balance) {
10.     this.balance = balance;
11.   }
12.
13.   public long getBalance() {
14.     return balance;
15.   }
16.
```

```
17.     public synchronized void deposit(long amount) {
18.       checkAmountNonNegative(amount);
19.       balance += amount;
20.       notifyAll();
21.     }
22.
23.     public synchronized void withdraw(long amount) {
24.       checkAmountNonNegative(amount);
25.       if (balance < amount) {
26.         throw new IllegalArgumentException(
27.             "not enough money");
28.       }
29.       balance -= amount;
30.     }
31.
32.     public synchronized void waitAndWithdraw(
33.             long amount)
34.         throws InterruptedException {
35.       checkAmountNonNegative(amount);
36.       while (balance < amount) {
37.         wait();
38.   //      System.out.println("Wakeup: " + balance);
39.       }
40.       balance -= amount;
41.     }
42.
43.     private static void checkAmountNonNegative(
44.         long amount) {
45.       if (amount < 0) {
46.         throw new IllegalArgumentException(
47.             "negative amount");
48.       }
49.     }
50.   }
```

```
1.  Calling waitAndWithdraw() ...
2.  waitAndWithdraw() finished
```

Атомарность

Чтение и запись полей всех типов, кроме long и double, происходит атомарно.

Если поле объявлено с модификатором volatile, то атомарно читаются и пишутся даже long и double.

Видимость

Изменения значений полей, сделанные одним потоком, могут быть не видны в другом потоке.

Изменения, сделанные одним потоком, могут быть видны в другом потоке.

Правила формализованы при помощи отношения happens-before (если в одном поток произошло некоторое событие X, а в другом потоке после этого произошло событие Z, то мы гарантировано знаем, что все, что произошло до X будет видно после Z).

Семантика final.

happens-before

Запись volatile-поля happens-before чтения этого поля.

Освобождение монитора happens-before захват того же монитора.

thread.start() happens-before thread.run().

Завершение thread.run() happens-before выход из thread.join().

```
1. public class Singleton {
```

```
 2.
 3.    private int foo;
 4.    private String bar;
 5.
 6.    private Singleton() {
 7.      this.foo = 13;
 8.      this.bar = "zap";
 9.    }
10.
11.    private static Singleton instance;
12.
13.    public static Singleton getInstance() {
14.      if (instance == null) {
15.        instance = new Singleton();
16.      }
17.      return instance;
18.    }
19.  }
```

Для многопоточной программы:

```
 1.  public class Singleton {
 2.
 3.    private int foo;
 4.    private String bar;
 5.
 6.    private Singleton() {
 7.      this.foo = 13;
 8.      this.bar = "zap";
 9.    }
10.
11.    private static Singleton instance;
12.
13.    public synchronized static
14.        Singleton getInstance() {
15.      if (instance == null) {
16.        instance = new Singleton();
17.      }
18.      return instance;
19.    }
20.  }
```

Ни в коем случае не делать такую штуку (можем получить ссылку на недостроенный объект):

```
1.  public static Singleton getInstance() {
2.    if (instance == null) {
3.      synchronized (Singleton.class) {
4.        if (instance == null) {
5.          instance = new Singleton();
6.        }
7.      }
8.    }
9.    return instance;
10. }
```

Но это лечится так:

```
1.  public class Singleton {
2.
3.    private int foo;
4.    private String bar;
5.
6.    private Singleton() {
7.      this.foo = 13;
8.      this.bar = "zap";
9.    }
10.
11.   private volatile static Singleton instance;
12.
13.   public static Singleton getInstance() {
14.     if (instance == null) {
15.       synchronized (Singleton.class) {
16.         if (instance == null) {
17.           instance = new Singleton();
18.         }
19.       }
20.     }
21.     return instance;
22.   }
23. }
```

Код

https://github.com/java-the-best/multithreading-vladykin

Источники

- https://www.youtube.com/watch?v=zxZ0BXlTys0&ab_channel=ComputerScienceCenter

Многопоточность в Java: средства стандартной библиотеки

java.util.concurrent.atomic

- AtomicBoolean, AtomicInteger, AtomicLong, AtomocReference<V>;
- Операции: V get(), void set(V newValue), boolean compareAndSet(V expect, V update).

compareAndSet

Примитив compareAndSet позволяет реализовать другие операции.

```
1.  public final int incrementAndGet() {
2.    for (;;) {
3.      int current = get();
4.      int next current + 1;
5.      if (compareAndSet(current, next)) {
6.        return next;
7.      }
8.    }
9.  }
```

Примеры

```
1.  public class SequenceGeneratorBroken {
2.
3.    private static volatile int counter = 0;
4.
5.    public static int nextInt() {
6.      return counter++;
7.    }
8.
9.    public static void main()
10.       throws InterruptedException {
11.     List<Thread> threads = new ArrayList<>();
12.
13.     for (int i = 0; i < 10; i++) {
14.       Thread thread = new Thread(() -> {
```

```
15.           for (int j = 0; j < 1000; j++) {
16.             nextInt();
17.           }
18.         });
19.         thread.start();
20.         threads.add(thread);
21.       }
22.
23.       for (Thread thread : threads) {
24.         thread.join();
25.       }
26.
27.       System.out.println(
28.           "Counter final value: " + counter);
29.     }
30.   }
```

```
1.  Counter final value: 9040
```

Правильное решение

```
 1.  public class SequenceGeneratorGood1 {
 2.
 3.    private static int counter = 0;
 4.
 5.    public static synchronized int nextInt() {
 6.      return counter++;
 7.    }
 8.
 9.    public static void main()
10.        throws InterruptedException {
11.      List<Thread> threads = new ArrayList<>();
12.
13.      for (int i = 0; i < 10; i++) {
14.        Thread thread = new Thread(() -> {
15.          for (int j = 0; j < 1000; j++) {
16.            nextInt();
17.          }
18.        });
19.        thread.start();
20.        threads.add(thread);
21.      }
22.
23.      for (Thread thread : threads) {
24.        thread.join();
25.      }
```

```
26.
27.     System.out.println(
28.         "Counter final value: " + counter);
29.   }
30. }
```

```
1.  Counter final value: 10000
```

Еще одно правильное решение

```
1.  public class SequenceGeneratorGood2 {
2.
3.    private static final AtomicInteger counter =
4.        new AtomicInteger();
5.
6.    public static int nextInt() {
7.      return counter.getAndIncrement();
8.    }
9.
10.   public static void main()
11.       throws InterruptedException {
12.     List<Thread> threads = new ArrayList<>();
13.
14.     for (int i = 0; i < 10; i++) {
15.       Thread thread = new Thread(() -> {
16.         for (int j = 0; j < 1000; j++) {
17.           nextInt();
18.         }
19.       });
20.       thread.start();
21.       threads.add(thread);
22.     }
23.
24.     for (Thread thread : threads) {
25.       thread.join();
26.     }
27.
28.     System.out.println(
29.         "Counter final value: " + counter);
30.   }
31. }
```

```
1.  Counter final value: 10000
```

Semaphore

- класс java.util.concurrent.Semaphore;

- ограничивает одновременный доступ к ресурсу;
- в отличии от synchronized-блока, одновременно могут работать несколько потоков (но не более заданного N);
- операции: void acquire(), void release().

```
 1. public static void main()
 2.     throws InterruptedException {
 3.   Semaphore semaphore = new Semaphore(10);
 4.   semaphore.acquire();
 5.   try {
 6.     ...
 7.   } finally {
 8.     semaphore.release();
 9.   }
10. }
```

Примеры

```
 1. public class SemaphoreDemo {
 2.
 3.   public static void main()
 4.       throws InterruptedException {
 5.     Semaphore semaphore = new Semaphore(2);
 6.
 7.     List<Thread> threads = new ArrayList<>();
 8.     for (int i = 0; i < 10; i++) {
 9.       DemoThread thread =
10.           new DemoThread(semaphore);
11.       threads.add(thread);
12.       thread.start();
13.     }
14.
15.     Thread.sleep(20_000);
16.
17.     for (Thread thread : threads) {
18.       thread.interrupt();
19.     }
20.   }
21.
```

```
22.    private static class DemoThread extends Thread {
23.
24.      private final Semaphore semaphore;
25.
26.      public DemoThread(Semaphore semaphore) {
27.        this.semaphore = semaphore;
28.      }
29.
30.      @Override
31.      public void run() {
32.        try {
33.          runUnsafe();
34.        } catch (InterruptedException e) {
35.          System.out.println(
36.              getName() + " interrupted");
37.        }
38.      }
39.
40.      private void runUnsafe()
41.          throws InterruptedException {
42.        for (;;) {
43.          semaphore.acquire();
44.          try {
45.            System.out.println(
46.                getName() + " acquired semaphore");
47.            Thread.sleep(5_000L);
48.          } finally {
49.            System.out.println(
50.                getName() + " releasing semaphore");
51.            semaphore.release();
52.          }
53.        }
54.      }
55.    }
56.  }
```

```
1.  Thread-0 acquired semaphore
2.  Thread-1 acquired semaphore
3.  Thread-0 releasing semaphore
4.  Thread-1 releasing semaphore
5.  Thread-1 acquired semaphore
6.  Thread-0 acquired semaphore
7.  Thread-1 releasing semaphore
8.  Thread-0 releasing semaphore
9.  Thread-0 acquired semaphore
```

```
Thread-1 acquired semaphore
Thread-1 releasing semaphore
Thread-1 acquired semaphore
Thread-0 releasing semaphore
Thread-0 acquired semaphore
Thread-3 interrupted
Thread-9 interrupted
Thread-2 interrupted
Thread-8 interrupted
Thread-1 releasing semaphore
Thread-1 interrupted
Thread-4 interrupted
Thread-0 releasing semaphore
Thread-6 interrupted
Thread-7 interrupted
Thread-5 interrupted
Thread-0 interrupted
```

Организация «честной очереди»

```
public class SemaphoreDemo {

  public static void main()
      throws InterruptedException {
    Semaphore semaphore = new Semaphore(2, true);

    List<Thread> threads = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
      DemoThread thread =
          new DemoThread(semaphore);
      threads.add(thread);
      thread.start();
    }

    Thread.sleep(20_000);

    for (Thread thread : threads) {
      thread.interrupt();
    }
  }

  private static class DemoThread extends Thread {

    private final Semaphore semaphore;

```

```
26.       public DemoThread(Semaphore semaphore) {
27.         this.semaphore = semaphore;
28.       }
29.
30.       @Override
31.       public void run() {
32.         try {
33.           runUnsafe();
34.         } catch (InterruptedException e) {
35.           System.out.println(
36.               getName() + " interrupted");
37.         }
38.       }
39.
40.       private void runUnsafe()
41.           throws InterruptedException {
42.         for (;;) {
43.           semaphore.acquire();
44.           try {
45.             System.out.println(
46.                 getName() + " acquired semaphore");
47.             Thread.sleep(5_000L);
48.           } finally {
49.             System.out.println(
50.                 getName() + " releasing semaphore");
51.             semaphore.release();
52.           }
53.         }
54.       }
55.     }
56.   }
```

```
1.  Thread-0 acquired semaphore
2.  Thread-1 acquired semaphore
3.  Thread-0 releasing semaphore
4.  Thread-1 releasing semaphore
5.  Thread-2 acquired semaphore
6.  Thread-4 acquired semaphore
7.  Thread-2 releasing semaphore
8.  Thread-4 releasing semaphore
9.  Thread-3 acquired semaphore
10. Thread-5 acquired semaphore
11. Thread-3 releasing semaphore
12. Thread-5 releasing semaphore
13. Thread-6 acquired semaphore
```

```
Thread-7 acquired semaphore
Thread-7 releasing semaphore
Thread-6 releasing semaphore
Thread-8 acquired semaphore
Thread-9 interrupted
Thread-8 releasing semaphore
Thread-8 interrupted
Thread-1 interrupted
Thread-2 interrupted
Thread-6 interrupted
Thread-4 interrupted
Thread-5 interrupted
Thread-3 interrupted
Thread-7 interrupted
Thread-0 interrupted
```

В отличии от блока synchronized блока Semaphore имеет возможность потоки выстраивать в очередь («честную»).

java.util.concurrent.CountDownLatch

Обеспечивает точку синхронизации между N потоками (несколько потоков могут дождаться друг друга и потом стартовать одновременно).

Операции: void await(), void countDown().

```
CountDownLatch latch = new CountDownLatch(10);
//...
latch.await();
```

Примеры

```
public class CountDownLatchDemo {
  public static void main()
      throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(10);

    for (int i = 0; i < 10; i++) {
      new DemoThread(latch).start();
    }
```

```
  }

  private static class DemoThread extends Thread {
    private final CountDownLatch latch;

    public DemoThread(CountDownLatch latch) {
      this.latch = latch;
    }

    @Override
    public void run() {
      try {
        runUnsafe();
      } catch (InterruptedException e) {
        System.out.println(
            getName() + " interrupted");
      }
    }

    private void runUnsafe()
        throws InterruptedException {
      Thread.sleep(
          (long) (Math.random() + 10_000L));

      System.out.println(
          getName() + " finished initialization");

      latch.countDown();
      latch.await();

      System.out.println(
          getName() + " entered main phase");

      Thread.sleep(
          (long) (Math.random() + 10_000L));
    }
  }
}
```

```
Thread-0 finished initialization
Thread-8 finished initialization
Thread-7 finished initialization
Thread-5 finished initialization
Thread-4 finished initialization
Thread-2 finished initialization
```

```
 7.  Thread-1 finished initialization
 8.  Thread-6 finished initialization
 9.  Thread-9 finished initialization
10.  Thread-3 finished initialization
11.  Thread-3 entered main phase
12.  Thread-0 entered main phase
13.  Thread-8 entered main phase
14.  Thread-5 entered main phase
15.  Thread-2 entered main phase
16.  Thread-7 entered main phase
17.  Thread-9 entered main phase
18.  Thread-6 entered main phase
19.  Thread-1 entered main phase
20.  Thread-4 entered main phase
```

java.util.concurrent.CyclicBarrier

Вариант CountDownLatch, допускающий повторное ожидание.

java.util.concurrent.locks.ReentrantLock

Обеспечивает взаимное исключение потоков, аналогичное synchronized-блокам.

Операции: lock(), unlock().

```
1.  Lock lock = new ReentrantLock();
2.  lock.lock();
3.  try {
4.    ...
5.  } catch (Exception e) {
6.    lock.unlock();
7.  }
```

Как и с предыдущим примером в конструктор можно передать true для организации «честной» очереди.

java.util.concurrent.locks.Condition

- аналог wait/notify;

- привязан к Lock'у;
- у одного Lock'а может быть много Condition'ов.

```
 1.  Lock lock = new ReentrantLock();
 2.  Condition condition = lock.newCondition();
 3.
 4.  lock.lock();
 5.  try {
 6.    while (!conditionSatisfied()) {
 7.      condition.await();
 8.    }
 9.  } finally {
10.    lock.unlock();
11.  }
12.
13.  // где-нибудь еще в нашей программе
14.  lock.lock();
15.  try {
16.    condition.signal();
17.  } finally {
18.    lock.unlock();
19.  }
```

Примеры

```
 1.  public static void main()
 2.      throws InterruptedException {
 3.    Account account = new Account(0);
 4.    new DepositThread(account).start();
 5.    System.out.println("Entering waitAndWithdraw");
 6.    account.waitAndWithdraw(50_000_000);
 7.    System.out.println(
 8.        "waitAndWithdraw finishrd, end balance = "
 9.        + account.getBalance());
10.  }
11.
12.  private static class DepositThread
13.      extends Thread {
14.
15.    private final Account account;
16.
17.    private DepositThread(Account account) {
18.      this.account = account;
```

```
19.     }
20.
21.     @Override
22.     public void run() {
23.       for (int i = 0; i < 60_000_000; i++) {
24.         account.deposit(1);
25.       }
26.     }
27.   }
```

```
 1.  public class Account {
 2.
 3.    private final Lock lock = new ReentrantLock();
 4.    private final Condition balanceIncreased =
 5.        lock.newCondition();
 6.
 7.    private long balance;
 8.
 9.    public Account() {
10.      this(0L);
11.    }
12.
13.    public Account(long balance) {
14.      this.balance = balance;
15.    }
16.
17.    public long getBalance() {
18.      lock.lock();
19.      try {
20.        return balance;
21.      } finally {
22.        lock.unlock();
23.      }
24.    }
25.
26.    public void deposit(long amount) {
27.      checkAmountNonNegative(amount);
28.      lock.lock();
29.      try {
30.        balance += amount;
31.        balanceIncreased.signalAll();
32.      } finally {
33.        lock.unlock();
34.      }
35.    }
```

```
36.
37.   public void withdraw(long amount) {
38.     checkAmountNonNegative(amount);
39.     lock.lock();
40.     try {
41.       if (balance < amount) {
42.         throw new IllegalArgumentException(
43.             "not enough money");
44.       }
45.       balance -= amount;
46.     } finally {
47.     lock.unlock();
48.   }
49.     }
50.
51.   public void waitAndWithdraw(long amount)
52.       throws InterruptedException {
53.     checkAmountNonNegative(amount);
54.     lock.lock();
55.     try {
56.       while (balance < amount) {
57.         balanceIncreased.await();
58. //        System.out.println("awake");
59.       }
60.       balance -= amount;
61.     } finally {
62.       lock.unlock();
63.     }
64.   }
65.
66.   public static void checkAmountNonNegative(
67.       long amount) {
68.     if (amount < 0) {
69.       throw new IllegalArgumentException(
70.           "negative amount");
71.     }
72.   }
73. }
```

```
1.  Entering waitAndWithdraw
2.  waitAndWithdraw finishrd, end balance = 4650
```

java.util.concurrent.locks.ReentrantReadWriteLock

Поддерживает разделение доступа на чтение и на запись.

```
ReadWriteLock lock = new ReentrantReadWriteLock();

// где-нибудь в нашей программе
lock.readLock().lock();
try {
  readOnlyOperation();
} finally {
  lock.readLock().unlock();
}

// где-нибудь в нашей программе
lock.writeLock().lock();
try {
  modifyingOperation();
} finally {
  lock.writeLock().unlock();
}
```

java.util.concurrent

Многопоточные варианты стандартных коллекций:

- ConcurrentHashMap (HashMap);
- ConcurrentSkipListMap (TreeMap);
- ConcurrentSkipListSet (TreeSet);
- CopyOnWriteArrayList;
- CopyOnWriteArraySet.

Более эффективны, чем полностью синхронизированные коллекции java.util.Collections.synchronizedCollection().

java.util.concurrent.ConcurrentLinkedQueue<E>

Реализация очереди, поддерживающая одновременный доступ из многих потоков, при этом не использующая блокировки.

Операции: boolean offer(E e), E poll, E peek().

java.util.concurrent.BlockingQueue<E>

Очередь, поддерживающая ограничение по размеру и операции ожидания.

Операции: void put(E e), E take().

Реализации: LinkedBlockingQueue, ArrayBlockingQueue, ...

java.util.concurrent.ExecutorService и его соседи

Инфраструктура для выполнения задач в несколько потоков.

Инкапсулирует создание потоков, организация очереди задач, распределение задач по потокам.

ExecutorService

- Future<?> submit(Runnable task);
- <T> Future<T> submit(Callable<T> task);
- void shutdown();
- List<Runnable> shutdownNows().

java.util.concurrent.ExecutionException

- `ExecutorService newSingleThreadExecutor();`
- `ExecutorService newFixedThreadPool(int nThreads);`
- `ExecutorService newCachedThreadPool().`

Примеры

```
 1.  public static void main() throws Exception {
 2.    ExecutorService executor =
 3.        Executors.newFixedThreadPool(2);
 4.
 5.    System.out.println("Submit worker 1");
 6.    Future<String> future1 = executor.submit(
 7.        new Worker("worker1"));
 8.
 9.    System.out.println("Submit worker 2");
10.    Future<String> future2 = executor.submit(
11.        new Worker("worker2"));
12.
13.    System.out.println(
14.        "Result fron worker1: " + future1.get());
15.    System.out.println(
16.        "Result fron worker2: " + future2.get());
17.
18.    System.out.println("-----------------------");
19.
20.    System.out.println(
21.        "Submit workers using invokeAll()");
22.    List<Future<String>> futures =
23.        executor.invokeAll(Arrays.asList(
24.            new Worker("worker3"),
25.            new Worker("worker4"),
26.            new Worker("worker5")));
27.
28.    System.out.println("Exited invokeAll()");
29.    for (Future<String> future : futures) {
30.      System.out.println(
31.          "Result from worker: " + future.get());
32.    }
33.
```

```
    executor.shutdown();
    executor.awaitTermination(
        10L,
        TimeUnit.SECONDS);
  }

  private static class Worker
      implements Callable<String> {

    private final String name;

    public Worker(String name) {
      this.name = name;
    }

    public String call()
        throws InterruptedException {
      long sleepTime =
          (long) (Math.random() * 10_000L);
      System.out.println(
          name + " started, going to sleep for "
          + sleepTime);
      Thread.sleep(sleepTime);
      System.out.println(name + " finished");
      return name;
    }
  }
```

```
Submit worker 1
Submit worker 2
worker1 started, going to sleep for 7173
worker2 started, going to sleep for 7312
worker1 finished
Result fron worker1: worker1
worker2 finished
Result fron worker2: worker2
----------------------
Submit workers using invokeAll()
worker3 started, going to sleep for 2275
worker4 started, going to sleep for 5788
worker3 finished
worker5 started, going to sleep for 7997
worker4 finished
worker5 finished
Exited invokeAll()
```

```
Result from worker: worker3
Result from worker: worker4
Result from worker: worker5
```

java.util.concurrent.ForkJoinPool

Вариант ExecutorService, в котором выполняющиеся задачи могут динамически порождать подзадачи.

Принимает на исполнение ForkJoinTask.

Примеры

```
public class Commons {

  public static int[] prepareArray() {
    int[] array = new int[20_000_000];
    for (int i = 0; i < array.length; i++) {
      array[i] = i;
    }
    return array;
  }

  public static double calculate(int[] array) {
    return calculate(array, 0, array.length);
  }

  public static double calculate(
      int[] array,
      int start,
      int end) {
    double sum = 0;
    for (int i = start; i < end; ++i) {
      sum += function(i);
    }
    return sum;
  }

  public static double function(int argument) {
    return Math.sin(argument);
  }
}
```

```
 1.  public class Sequential {
 2.
 3.    public static void main() {
 4.      int[] array = Commons.prepareArray();
 5.
 6.      long startTime = System.currentTimeMillis();
 7.
 8.      double sum = Commons.calculate(array);
 9.
10.      long endTime = System.currentTimeMillis();
11.
12.      System.out.println("sum = " + sum);
13.      System.out.println(
14.          "time = " + (endTime - startTime)
15.          + " ms");
16.    }
17.  }
```

```
1.  sum = 0.7052914342504155
2.  time = 6608 ms
```

```
 1.  public class ParallelInExecuteService {
 2.
 3.    public static void main() throws Exception {
 4.      int[] array = Commons.prepareArray();
 5.
 6.      ExecutorService executor =
 7.          Executors.newFixedThreadPool(2);
 8.
 9.      long startTime = System.currentTimeMillis();
10.
11.      Future<Double> future1 = executor.submit(
12.          new PartialCalc(array,
13.                          0,
14.                          array.length / 2));
15.      Future<Double> future2 = executor.submit(
16.          new PartialCalc(
17.              array,
18.              array.length / 2,
19.              array.length));
20.
21.      double sum = future1.get() + future2.get();
22.
```

```
23.       long endTime = System.currentTimeMillis();
24.
25.       System.out.println("sum = " + sum);
26.       System.out.println(
27.           "time = " + (endTime - startTime)
28.           + " ms");
29.
30.       executor.shutdown();
31.     }
32.
33.     private static class PartialCalc
34.         implements Callable<Double> {
35.
36.       private final int[] array;
37.       private final int start;
38.       private final int end;
39.
40.       public PartialCalc(int[] array,
41.                          int start,
42.                          int end) {
43.         this.array = array;
44.         this.start = start;
45.         this.end = end;
46.       }
47.
48.       public Double call() {
49.         return Commons.calculate(array, start, end);
50.       }
51.     }
52. }
```

```
1. sum = 0.70529143425024
2. time = 3639 ms
```

```
1.  public class ParallelInForkJoinPool {
2.
3.    public static void main() throws Exception {
4.      int[] array = Commons.prepareArray();
5.
6.      ForkJoinPool pool = new ForkJoinPool();
7.
8.      long startTime = System.currentTimeMillis();
9.
10.     double sum = pool.invoke(
```

```
11.           new RecursiveCalc(
12.               array,
13.               0,
14.               array.length));
15.
16.       long endTime = System.currentTimeMillis();
17.
18.       System.out.println("sum = " + sum);
19.       System.out.println(
20.           "time = " + (endTime - startTime)
21.           + " ms");
22.
23.       pool.shutdown();
24.     }
25.
26.     private static class RecursiveCalc
27.         extends RecursiveTask<Double> {
28.
29.       private static final
30.           int SEQUENTIAL_THRESHOLD = 50_000;
31.
32.       private final int[] array;
33.       private final int start;
34.       private final int end;
35.
36.       public RecursiveCalc(
37.           int[] array,
38.           int start,
39.           int end) {
40.         this.array = array;
41.         this.start = start;
42.         this.end = end;
43.       }
44.
45.       protected Double compute() {
46.         if (end - start <= SEQUENTIAL_THRESHOLD) {
47.           return Commons.calculate(array,
48.                                    start,
49.                                    end);
50.         } else {
51.           int mid = start + (end - start) / 2;
52.           RecursiveCalc left =
53.               new RecursiveCalc(array, start, mid);
54.           RecursiveCalc right =
```

```
55.                 new RecursiveCalc(array, mid, end);
56.             invokeAll(left, right);
57.             return left.join() + right.join();
58.           }
59.       }
60.     }
61. }
```

```
1. sum = 0.7052914342502838
2. time = 716 ms
```

stream.parallel()

Возвращает stream, дальнейшие операции в котором будут исполняться параллельно.

Надо следить за доступам к общим данным из передаваемых в sream операций.

Примеры

```
 1. public class ParallelStream {
 2.
 3.   public static void main() throws Exception {
 4.     int[] array = Commons.prepareArray();
 5.
 6.     long startTime = System.currentTimeMillis();
 7.
 8.     double sum = Arrays.stream(array)
 9.         .parallel()
10.         .mapToDouble(Commons::function)
11.         .sum();
12.
13.     long endTime = System.currentTimeMillis();
14.
15.     System.out.println("sum = " + sum);
16.     System.out.println(
17.         "time = " + (endTime - startTime)
18.         + " ms");
19.   }
20. }
```

```
1. sum = 0.7052914342503351
2. time = 766 ms
```

Неправильное решение этой задачи

```
 1.  public class ParallelStreamBroken {
 2.
 3.    public static void main() {
 4.      int[] array = Commons.prepareArray();
 5.
 6.      long startTime = System.currentTimeMillis();
 7.
 8.      double[] sum = new double[1];
 9.      Arrays.stream(array)
10.          .parallel()
11.          .mapToDouble(Commons::function)
12.          .forEach(x -> sum[0] += x);
13.
14.      long endTime = System.currentTimeMillis();
15.
16.      System.out.println("sum = " + sum[0]);
17.      System.out.println(
18.          "time = " + (endTime - startTime)
19.          + " ms");
20.    }
21.  }
```

```
1.  sum = -1175.1958081366392
2.  time = 832 ms
```

Превращение неправильной программы в правильную

```
 1.  public class ParallelStreamBrokenTrue {
 2.
 3.    public static void main() {
 4.      int[] array = Commons.prepareArray();
 5.
 6.      long startTime = System.currentTimeMillis();
 7.
 8.      DoubleAdder sum = new DoubleAdder();
 9.      Arrays.stream(array)
10.          .parallel()
11.          .mapToDouble(Commons::function)
12.          .forEach(sum::add);
13.
14.      long endTime = System.currentTimeMillis();
15.
16.      System.out.println(
```

```
            "sum = " + sum.doubleValue());
        System.out.println(
            "time = " + (endTime - startTime)
            + " ms");
    }
}
```

```
sum = 0.7052914342498308
time = 811 ms
```

Код

https://github.com/java-the-best/multithreading-vladykin

Источники

- https://www.youtube.com/watch?v=umTVNoG3760&t=12s

Новая тема

Содержание

Код

Ссылка

Источники

- Ссылка с описанием