Многопоточность в Java. Средства стандартной библиотеки (java.util.concurrent)

Сидоров Кирилл

Атомарные классы

Пакет **java.util.concurrent.atomic** содержит 9 классов для выполнения атомарных операций. **Атомарная операция** — операция, которая либо выполняется целиком, либо не выполняется вовсе (другими словами, выполняется за «один шаг»); операция, которая не может быть частично выполнена и частично не выполнена.

Основные классы пакета java.util.concurrent.atomic:

- **AtomicBoolean**
- **AtomicInteger**
- **AtomicLong**
- **AtomicReference<T>**

Основные методы всех Atomic классов:

1) boolean **compareAndSet**(T oldValue, T newValue)

2) T **get()**

Механизмы блокировок потоков

Пессимистический механизм блокировки — это вариант блокировки, когда в данный момент времени работа с общим состоянием разрешается только одному потоку. Использование ключевого слово **synchronized** является пессимистической блокировкой.

Противоположный «механизм блокировки» — **оптимистический**. Одним из вариантов оптимистической блокировки является подход «**Compare And Swap**» (CAS, Сравнение с обменом).

В этом случае блокировки **не происходит**, поток берет общую переменную и выполняет с ней необходимую операцию. Далее поток пытается установить в общую переменную обновлённое значение. Если поток обнаруживает, что значение переменной изменилось другим потоком, то он повторяет операцию снова, но уже с новым значением переменной.

Compare And Swap

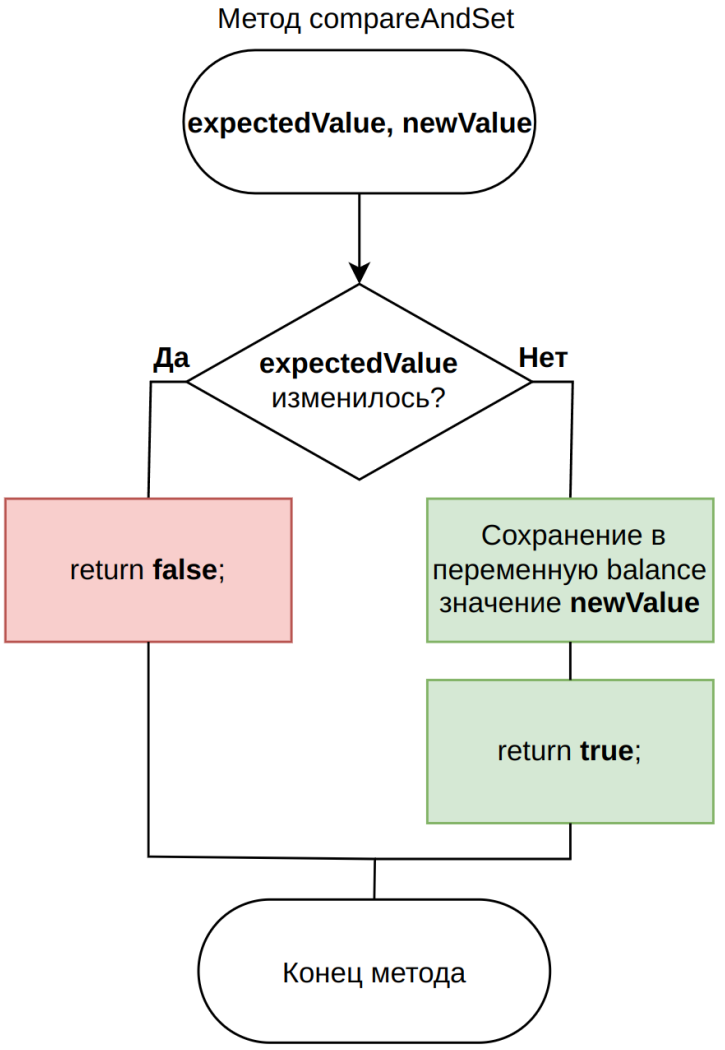
```
AtomicLong balance;

Account(long value) {
    this.balance = new AtomicLong(value);
}

void inc() {

    boolean success = false;
    while(!success) {
        long expValue = balance.get();
        long newValue = expValue + 1;

        success = balance.compareAndSet(expValue, newValue);
    }
}
```



Преимущества подхода Compare And Swap

1. Современные процессоры имеют встроенную реализацию Compare And Swap (т. е. на «машинном уровне» уже есть эффективная реализация данного подхода).

2. Блокировка потока (именно блокировка, когда потоку приходиться ждать) является «дорогой» с точки зрения вычислительных ресурсов операцией. Кроме того, у нас нет никакой гарантии относительно того, когда именно заблокированный поток будет разблокирован, и когда блок synchronized снова станет свободным.

Иллюстрация пессимистического механизма блокировки

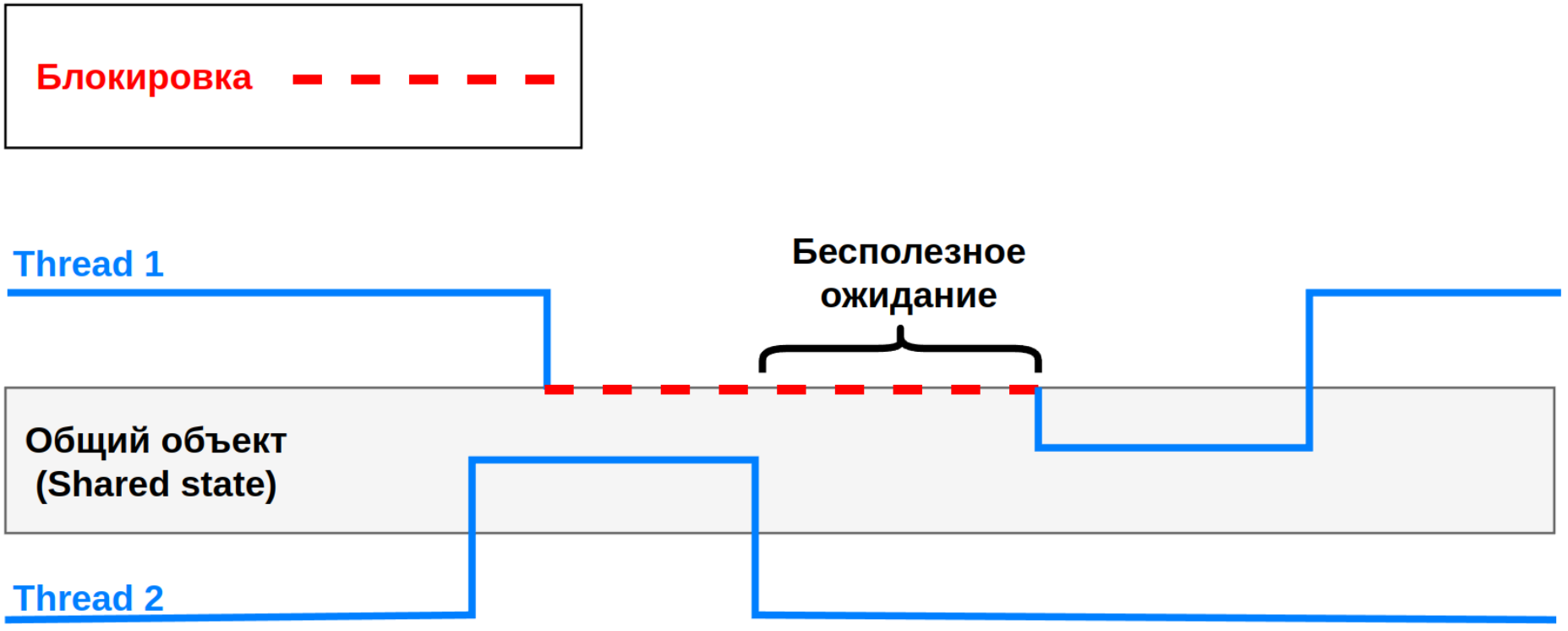
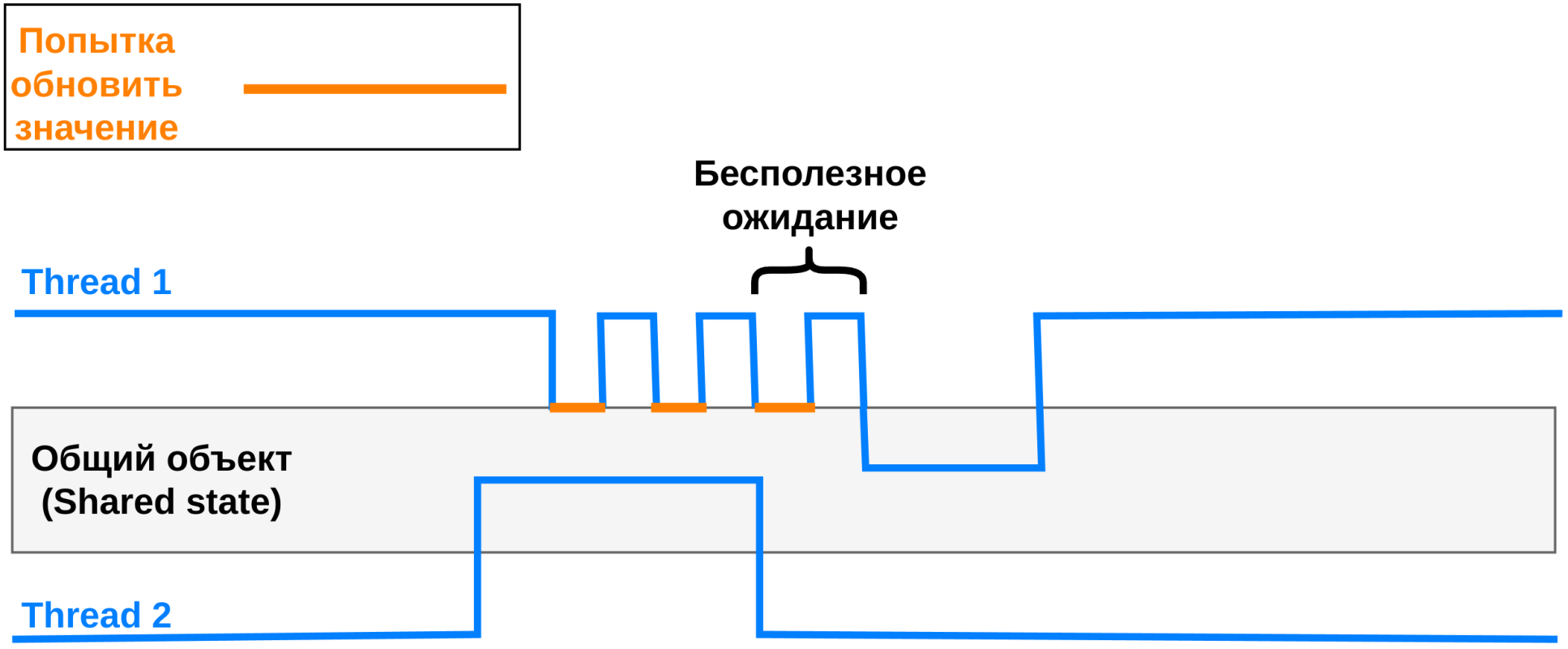


Иллюстрация оптимистического механизма блокировки (Compare And Swap)



Блокировки

Для управления доступом к общему ресурсу в качестве **альтернативы** оператору **synchronized** могут быть использованы блокировки из пакета **java.util.concurrent.locks**.

Основные **классы** пакета **java.util.concurrent.locks:**

1. Интерфейс Lock

2. ReentrantLock

3. ReentrantReadWriteLock

Основные **методы** интерфейса **Lock:**

1. void **lock()**: получение блокировки, поток будет ждать, если блокировка занята.

2. boolean **tryLock()**: попытка получить блокировку без ожидания (вернет true — если блокировка получена, иначе — false).

3. boolean **tryLock**(long timeout, TimeUnit unit): попытка получить блокировку с ожиданием.

4. void **unlock**(): освобождение блокировки.

Блокировки

Реализация с использованием ReentrantLock

```
public class Account {

    private Lock lock = new ReentrantLock();
    private long balance;

    public void add(long value) {

        lock.lock();

        balance += value; // работа с общим состоянием

        lock.unlock();
    }
}
```

Альтернативная реализация с использованием synchronized

```
public class Account {

    private long balance;

    public synchronized void add(long value) {
        balance += value; // работа с общим состоянием
    }
}
```

Преимущества блокировок из пакета **java.util.concurrent.locks** перед **synchronized**

1. Синхронизированный блок не дает никаких гарантий относительно последовательности, в которой потокам, ожидающим входа в него, предоставляется доступ. **Locks могут дать гарантии «справедливости»** — отдаватать блокировку потоку, который дольше других ожидает. Для создания справедливой блокировки нужно в конструкторе класса передать флаг fair = true (например, **new ReentrantLock(true)**).

2. **Locks** позволяют устанавливать **таймаут на получение блокировки**, либо просто сделать попытку захвата блокировки.

3. Синхронизированный блок должен полностью содержаться в одном методе. Lock может иметь вызовы lock() и unlock() в отдельных методах (но при этом не нужно забывать про усложнение кода в таком случае).

ReentantReadWriteLock

Класс **ReentrantReadWriteLock** позволяет **нескольким** потокам **одновременно читать** общий объект, но только **одному записывать** в данный момент времени.

```
// может быть "справедливой" (true)
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

// ReadLock может быть "захвачена" одновременно несколькими потоками:
// если ни один поток не заблокировал ReadWriteLock для записи,
// либо если ни один поток не запросил блокировку записи (но еще не получил ее).
lock.readLock().lock();
// Чтение общей переменной
lock.readLock().unlock();

// WriteLock может быть "захвачена" только ОДНИМ потоком
// если в данный момент нет потоков, которые "захватили" Write или Read блокировку.
lock.writeLock().lock();
// Запись общей переменной
lock.writeLock().unlock();
```

Важные особенности блокировок

1. Слово **Reentrant** (вариант перевода «Повторно используемый», происходит от слова ReEnter) в название блокировки означает, что один и тот же поток может «**захватывать**» (метод **lock()**) блокировку **несколько раз**, соответственно и «**вернуть**» блокировку (метод **unlock()**) поток должен такое же количество раз. Блок **synchronized** также является Reentrant.

Reentrant Lock

```
public class Account {

    private Lock lock = new ReentrantLock();

    public void outer() {
        lock.lock();
        inner();
        lock.unlock();
    }

    public void inner() {
        lock.lock();
        // какая-то работа
        lock.unlock();
    }
}
```

Reentrant synchronized

```
public class Account {

    public synchronized void outer() {
        inner();
    }

    public synchronized void inner() {
        // какая-то работа
    }
}
```

Важные особенности блокировок

2. Правильной практикой считается **вызов** метода **unlock**() **в блоке finally**, чтобы гарантировать «возвращение» блокировки если возникнет исключительная ситуация внутри критической секции (блока синхронизации).

```
lock.lock();
try {
    // критическая секция, в которой может возникнуть исключение
} finally {
    lock.unlock();
}
// без использования finally может быть Deadlock
```

3. Метод **tryLock()** (без параметров) **не учитывает режим «справедливости» ReentrantLock** (new ReentrantLock(true)). Чтобы обеспечить справедливость, вы должны использовать метод tryLock(long timeout, TimeUnit timeUnit).

Объекты синхронизации Synchroniser пакета java.util.concurrent

В пакете java.util.concurrent есть более сложные механизмы синхронизации для различных специфических задач.

Semaphore	Объект синхронизации, ограничивающий количество потоков, которые могут «войти» в заданный участок кода.
CountDownLatch	Объект синхронизации, разрешающий вход в заданный участок кода при выполнении определенных условий.
CyclicBarrier	Объект синхронизации типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков.
Exchanger	Объект синхронизации, позволяющий провести обмен данными между двумя потоками.
Phaser	Объект синхронизации типа «барьер». Phaser позволяет синхронизировать потоки с помощьюю отдельной фазы (стадии) выполнения общего действия.

Коллекции для многопоточности

Коллекции Java для многопоточных программ

Потокобезопасные коллекции
(пакте java.util.concurrent)

Примеры классов:

ConcurrentHashMap
CopyOnWriteArrayList
CopyOnWriteArraySet

Неблокирующие очереди
Примеры классов:

ConcurrentLinkedQueue
ConcurrentLinkedDeque

Блокирующие очереди
Примеры классов:

ArrayBlockingQueue
LinkedBlockingQueue
LinkedBlockingDeque

Коллекции с синхронизированными методами
(пакет java.util)

Создаются с помощью статических методов класса *Collections*:

Collections.synchronizedList(List)
Collections.synchronizedSet(Set)
Collections.synchronizedMap(Map)

Коллекции с синхронизированными методами (пакет java.util)

1. Коллекции с синхронизированными методами создаются с помощью статических методов класса Collections: Collections.synchronizedList(List<T> list) и т. д.

2. В каждом методе выполняется «синхронизация» по объекту коллекции, таким образом **только один поток** в данный момент времени может производить какие-либо действия с коллекцией.

3. **Прохождение по элементам таких коллекций с помощью итератора (Iterator) (или с помощью цикла) не является потокобезопасной операцией!** Если один поток изменяет коллекцию, а другой в этот момент проходит по её элементам может возникнуть ConcurrentModificationException. При использовании итератора синхронизированной коллекции нужно обязательно выполнять синхронизацию по объекту коллекции самостоятельно.

```
List<String> syncList = Collections.synchronizedList(list);
synchronized (syncList) {
    Iterator i = syncList.iterator();

    while (i.hasNext()) {
        // прохождение по элементам списка
    }
}
```

```
public void add(int index, E element) {
    synchronized (this.mutex) {
        this.list.add(index, element);
    }
}

public E remove(int index) {
    synchronized (this.mutex) {
        return this.list.remove(index);
    }
}

public int indexOf(Object o) {
    synchronized (this.mutex) {
        return this.list.indexOf(o);
    }
}
```

Потокобезопасные коллекции (пакет java.util.concurrent)

Основные потокобезопасные коллекции:

1) ConcurrentHashMap — потокобезопасный аналог HashMap.

2) CopyOnWriteArrayList — потокобезопасный аналог ArrayList.

3) CopyOnWriteArraySet — потокобезопасная коллекция, реализующая интерфейс Set.

4) ConcurrentSkipListMap — потокобезопасный аналог TreeMap.

5) CncurrentSkipListSet — потокобезопасный аналог TreeSet.

Неблокирующие очереди

6) ConcurrentLinkedQueue — потокобезопасная односторонняя очередь.

7) ConcurrentLinkedDeque — потокобезопасная двухсторонняя очередь.

Блокирующие очереди

8) ArrayBlockingQueue — потокобезопасная односторонная очередь.

9) LinkedBlockingQueue — потокобезопасная односторонная очердь на связанных узлах.

10) LinkedBlockingDeque — потокобезопасная двухсторонняя очердь на связанных узлах.

Особенности потокобезопасных коллекции (java.util.concurrent)

1) ConcurrentHashMap состоит из нескольких сегментов (нескольких HashMap), благодаря чему при работе с коллекцией **не требуется блокировки всей Map**. При записи нового значения блокируется только часть Map. При одновременной итерации и изменении Map в другом поток, исключение ConcurrentModificationException не происходит. Однако **итератор не предназначен для использования более чем одним потоком**.

2) CopyOnWriteArrayList (и CopyOnWriteArraySet) **создает новую копию** списка при выполнении **модифицирующих операций** и гарантирует, что итератор вернет состояние списка на момент его создания. При одновременной итерации и изменении списка в другом поток, исключение ConcurrentModificationException не происходит. CopyOnWriteArrayList (и CopyOnWriteArraySet) хорошо подходят для задач, где есть нечастые операции вставки и удаления в одних потоках и одновременный перебор в других.

3) CopyOnWriteArraySet **не поддерживает** метод remove() (при вызове метода возникает исключение).

Блокирующие очереди (java.util.concurrent)

Блокирующие очереди пакета java.uitl.concurrent реализуют интерфейс: **BlockingQueue<E>**.

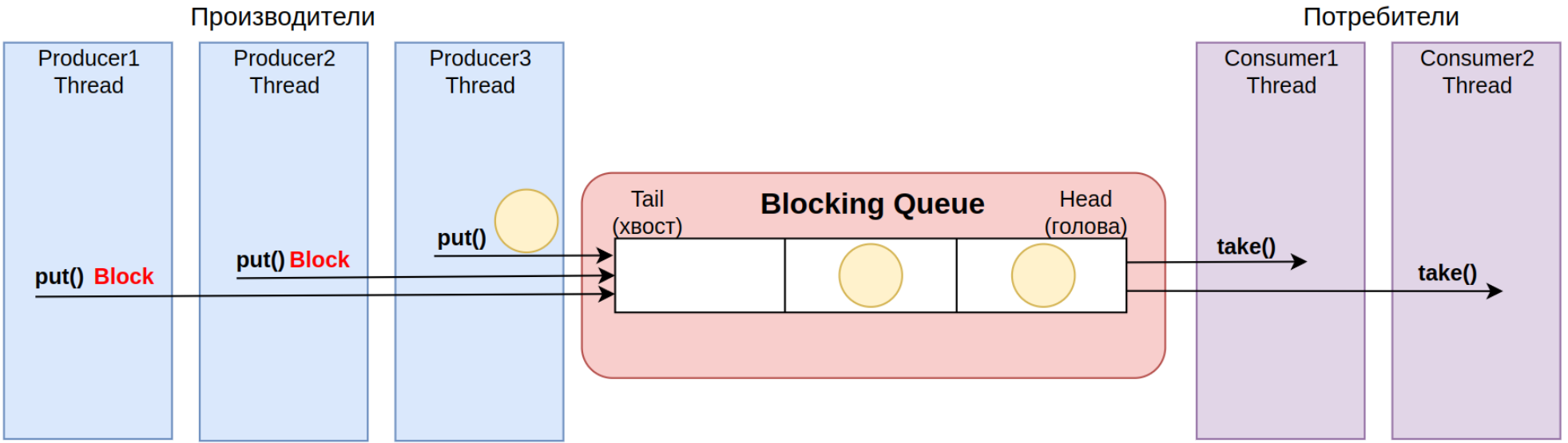
Блокирующая очередь означает, что Java **BlockingQueue способна блокировать потоки**, которые пытаются вставить или извлечь элементы из очереди. Например, если поток пытается взять элемент, а очередь пуста, поток может быть заблокирован до тех пор, пока в очереди не появится новый элемент.

Основные методы интерфейса **BlockingQueue** представлены в таблице:

	Может вызвать исключение	Блокируют поток	Не блокирует поток	Блокировка с ожиданием
Вставка (Insert)	boolean add(E e)	**put(E e)**	boolean offer(E e)	boolean offer(e, time, unit)
Удаление (Remove)	boolean remove(Object o)	**E take()**	E poll()	E poll(time, unit)
Получение значения без удаления	E element()	-	E peek()	-

Блокирующие очереди (java.util.concurrent)

Блокирующие очереди могут быть эффективно использованы в реализации шаблона проектирования параллелизма «Производитель-Потребитель».

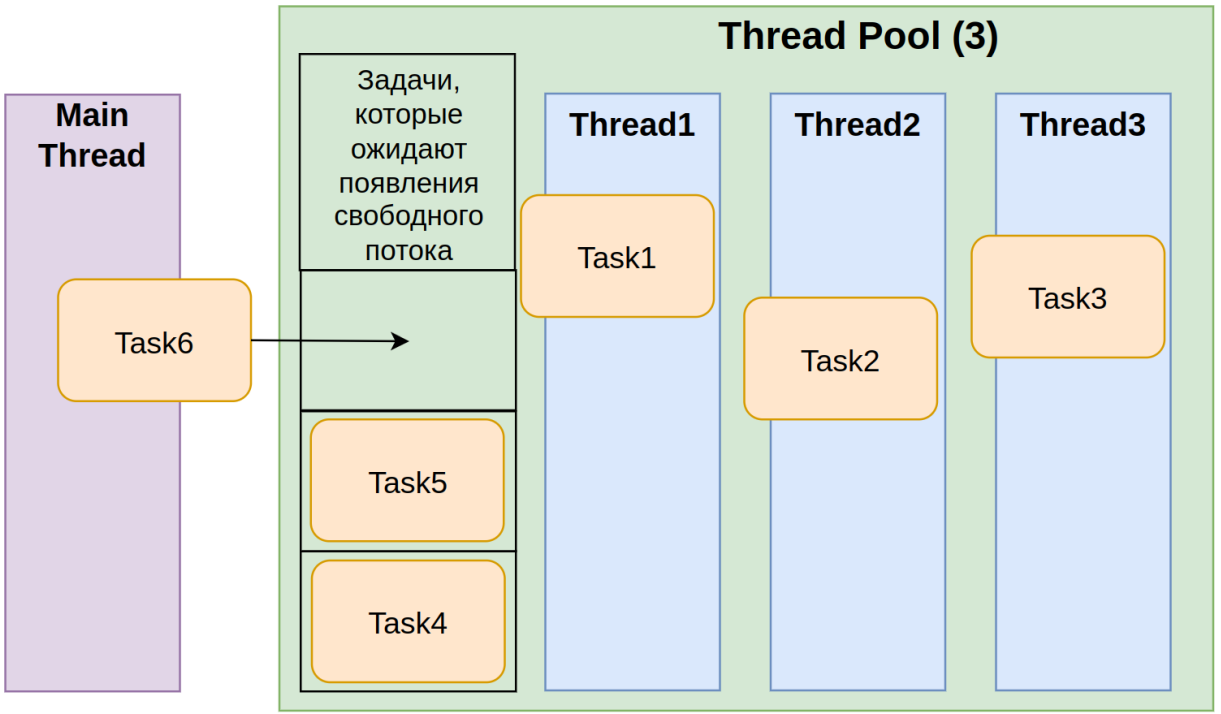


Пул потоков (Thread Pool)

Пул потоков — это набор предварительно инициализированных потоков, размер которого может быть как фиксированным, так и переменным.

Пул потоков позволяет «**повторно использовать**» **потоки** для выполнения задач.

Пул потоков — это альтернатива созданию нового потока для каждой задачи, которую необходимо выполнить.



Преимущества использования Пула потоков.

1. Операция **создания** нового **потока** является **дорогостоящей** операцией с точки зрения вычислительных ресурсов, поэтому повторное использование существующего потока для выполнения задачи может улучшить производительность относительно реализаций, где для каждой задачи создаётся новый поток (особенное если задач много).

2. Пул потоков **позволяет регулировать нагрузку** на программу (и соответственно на компьютер).

ThreadPoolExecutor

В пакета java.util.concurrent есть готовая реализация пула потоков — **ThreadPoolExecutor**.

Создать пул потоков можно самостоятельно с помощью конструктора класса **ThreadPoolExecutor**.

Либо воспользоваться статическими методами класса **Executors**.

Пулы потоков в пакете java.util.concurrent реализуют интерфейс **ExecutorService**.

```
// Пул с фиксированным количеством потоков
ExecutorService fixedThreadPool = Executors.newFixedThreadPool( nThreads: 10);

// Пул с одним потоком
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
```

Callable и Runnable

```java
package java.util.concurrent;

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

```java
package java.lang;

@FunctionalInterface
public interface Runnable {
    void run();
}
```

Про что не нужно забывать

1. Про существование оптимистического «механизма блокировки», алгоритм Compare And Swap.

2. Различия потокобезопасных коллекций и коллекций с синхронизированными методами.

3. Преимущества использования Пула потоков.