Java, второй семестр

Василий Купоросов

25 мая 2018 г.

Содержание

1. Многопоточное программирование в Java

1.1. Введение

Однопоточную программу можно представить как стек вызовов. Выполнение происходит последовательно, всё хорошо.

В многопоточной программе каждый поток имеет свой стек вызовов, причем все эти стеки имеют общую память (heap).

Важно отличать поток от процесса. У процесса изолированная память, то есть он не может напрямую читать память другого процесса.

Потоки – это термин операционных систем. Они реализованы на уровне ОС. Потоков может быть сколько угодно, но одновременно исполняется столько потоков, сколько ядер у процессора. Какой поток сейчас будет выполняться, определяет менеджер потоков. Он может запускать потоки, останавливать, переключать контекст. Но переключение контекста – дорогая операция. Поэтому, если потоков в программе больше чем ядер, то программа может замедлиться, а не ускориться.

Java поток `!=` поток в ОС. В стандартной реализации Java (например, в Windows, Linux) они точь-в-точь такие же, но в других случаях могут и отличаться. Например, в DOS'е потоки очень странные, поэтому один к одному не сопоставляются с джававскими. У JVM есть свой планировщик, который не зависит от планировщика ОС, на которой джава-машина работает.

В Java есть два вида потоков – потоки-демоны (daemon threads) и пользовательские (user threads). Программа работает, пока хотя бы один пользовательский поток работает (первым запускается поток Main). Когда программа завершается, демоны убиваются насильно. Это их единственное отличие.

Менеджер переключает потоки и отправляет их выполняться на процессор. Менеджер руководствуется в том числе приоритетами. Чем выше приоритет потока, тем вероятнее планировщик отправит поток на процессор. Вообще критериев выбора потока очень много. Некоторые планировщики перебирают потоки в случайном порядке (например, чтобы избежать дедлоков).

1.2. Создание потоков в Java

Основные используемые для многопоточности классы:

- `java.lang.Thread` – класс потока. Позволяет создавать потоки и производить операции с ними.
- `java.lang.Runnable` – интерфейс для сущности, которую можно выполнить. Содержит единственный метод `public void run()`. В нем должна содержаться задача, которую будет выполнять поток.

Класс `Thread` реализует интерфейс `Runnable`, значит, можно создать наследника `Thread`, переопределить у него метод `run()` и использовать в качестве `Runnable`.

Рассмотрим простой пример. Мы создаем экземпляр безымянного класса-наследника `Thread`, переопределяем метод `run()` и запускаем получившийся поток.

```
// Создание потока
Thread t = new Thread() {
  public void run() {
    system.out.println(''Hello'');
  }
};
// Запуск потока
t.start();
```

Важно отметить, что для запуска потока надо писать именно `t.start();` а не `t.run();` потому что второе просто вызовет метод, который выполнится в том потоке, в котором мы находимся сейчас. Метод `start()` просит операционную систему создать поток и выполнить `run()` уже в нем. Стартовать поток можно **не больше одного раза**. Повторный вызов `start()` ни к чему хорошему не приведет.

Рассмотрим другой пример, делающий то же самое.

```
// Создание потока
Thread t = new Thread(new Runnable() {
  public void run() {
    system.out.println(''Hello'');
  }
});
// Запуск потока
t.start();
```

Его отличие в том, что мы не наследуем `Thread`, а передаем ему `Runnable` в конструкторе. Этот способ предпочтительнее, потому что можно создать отдельно объект `Runnable` и передать его нескольким потокам. Еще одна причина – отсутствие множественного наследования. Так как `Thread` – это класс, то отнаследовавшись от него, ни от какого другого класса мы наследоваться не сможем.

Создание потока – дорогая по времени операция, поэтому, если бы была возможность в один поток передавать разные `Runnable`, было бы полезно. Но **в Java такой возможности нет**.

Есть также конструктор `Thread(Runnable runnable, String name)`. В него можно передать имя создаваемого потока, а потом из `run()` спросить, в каком потоке сейчас происходит выполнение, или спросить, какие потоки сейчас выполняются.

Рассмотрим еще один пример.

```
public class Starter {
  public static void main(String[] args) {
    NameRunnable nr = new NameRunnable();
    Thread one = new Thread(nr);
    Thread two = new Thread(nr);
    Thread three = new Thread(nr);

    one.setName(''Первый'');
    two.setName(''Второй'');
    three.setName(''Третий'');

    one.start();
    two.start();
    three.start();
  }
}
```

```
17 class NameRunnable implements Runnable {
18   public void run() {
19     for (int x = 1; x <= 3; x++) {
20       System.out.println(''Запущен'' + Thread.currentThread().getName() + '', x равен'' + x);
21     }
22   }
23 }
```

Здесь мы запускаем три потока, каждый из которых три раза печатает свое имя и чему равен x. В этом примере мы учимся получать текущий поток и его имя: `Thread.currentThread().getName()`. Важно обратить внимание, что `currentThread()` – это **статический** метод класса `Thread`. Он в разных потоках вернет разное, а именно – в каком потоке он вызван. Как он реализован мы не знаем, можно считать, что это магия.

В этом примере мы последовательно запускаем три потока. Тем не менее, поскольку потоки выполняются параллельно, мы получим выведенные строчки в каком-то перемешанном порядке. Однако части строк друг с другом перемешаться не могут, потому что `println()` – блокирует поток вывода, и пока он что-то выводит, другие потоки не могут ничего выводить.

Любой поток обладает следующими свойствами:

- `long id` – идентификатор потока
- `String name` – имя
- `int priority` – приоритет. Чем он больше, тем больше процессорного времени потоку будут стараться давать.
- `boolean daemon` – является ли поток демоном

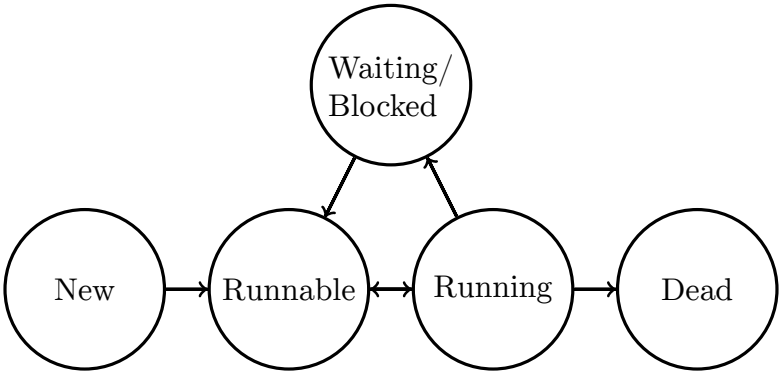
Эти свойства не могут меняться после его запуска.

1.3. Состояния потока

Состояния потока возвращаются методами `int getState()` и `boolean isAlive()`. Когда поток только создан, но еще не вызван, он считается неживым. После вызова `start()` он становится живым, и умирает после окончания выполнения (см таблицу).

getState()	isAlive()
NEW	false
RUNNABLE	true
BLOCKED	true
WAITING	true
TIMED_WAITING	true
TERMINATED	false

Рассмотрим подробнее состояния, в которых может находиться поток, пока он жив.



Только что созданный поток находится в состоянии New, пока его не запустили.

При вызове метода `start()`, он переходит в состояние Runnable. То есть он не начинает сразу выполняться, а ждет, когда планировщик потоков выделит ему процессорное время и даст поработать.

Когда время, отведенное потоку, заканчивается, он переходит обратно в состояние Runnable и ждет, когда его продолжат выполнять.

Также из состояния Running можно попасть в состояние Waiting. В этом состоянии поток либо заблокирован, либо чего-то ждет. Ждать можно какого-то условия или окончания какого-нибудь таймера.

Когда поток сделал всё что хотел, он переходит в состояние Dead и умирает. После этого его нельзя будет запустить снова. То есть поток может выполниться **только один раз**.

1.4. Взаимодействие потоков

1.4.1. Отправка в сон

Поток можно отправить в сон, вызвав метод `Thread.sleep(long millis)`. Самое главное, что надо помнить про этот метод – то что он **статический**. Поэтому следующий код отправит спать не поток `t`, а текущий поток.

```
Thread t = new Thread(...); // создали поток
t.start(); // запустили поток
t.sleep(100); // пошли спать сами, а t не пошел
```

Когда в потоке вызывается `sleep()`, он переходит в состояние Waiting и ждет переданное количество времени. По истечении этого времени в идеальном мире поток перейдет в состояние Runnable, а потом когда-нибудь в состояние Running. В реальности же планировщик потоков постоянно перебирает потоки, которые у него есть, и может еще долго не дойти до спящего потока, так что в состояние Runnable он может перейти еще не скоро. В итоге получается, что поток может спать сколько угодно времени, но не меньше того, сколько ему сказали.

Не всегда удобно задавать время в миллисекундах. Для этого есть перечисление `java.util.concurrent.TimeUnit`, позволяющее задавать время сна в разных единицах:

```
TimeUnit.SECONDS.sleep(long secs);
TimeUnit.MILLISECONDS.sleep(long millis);
TimeUnit.MICROSECONDS.sleep(long micros);
TimeUnit.NANOSECONDS.sleep(long nanos);
TimeUnit.DAYS.sleep(long days);
TimeUnit.HOURS.sleep(long hours);
TimeUnit.MINUTES.sleep(long mins);
```

1.4.2. Приоритеты и yield

Чем больше приоритет у потока, тем чаще ему будут давать работать. Задать приоритет можно методом `setPriority(int priority)`. Приоритет – это целое число от 1 до 10. Однако, не рекомендуется задавать приоритет, отличный от следующих:

```
Thread.MIN_PRIORITY // 1
Thread.NORM_PRIORITY // 5
Thread.MAX_PRIORITY // 10
```

Дело в том, что в старых версиях винды всего три приоритета, и, так как мы пишем кроссплатформенное приложение, надо ориентироваться на худший случай.

Статический метод `yield()` предлагает планировщику переключить текущий поток в состояние Runnable, чтобы можно было выполнить какой-нибудь другой поток. Планировщик может проигнорировать эту просьбу. Такое может быть нужно, если есть много потоков, и мы хотим, чтобы они все завершились максимально одновременно. Для этого мы хотим, чтобы переключение происходило почаще. Другой вариант – это когда наш поток зависит от того, что делают другие, и мы хотим им тоже давать поработать.

Получается, что `yield()` надо использовать только когда приложение уже написано, для увеличения многопоточной производительности.

1.4.3. Ожидание окончания потока

Поток может завершиться до того, как завершились его дочерние потоки. Например, поток Main может запустить несколько потоков и завершиться. Но иногда мы хотим дождаться завершение какого-либо потока. Для этого есть метод `join()`.

Метод `join()` не статический, он вызывается от другого потока, но действует на текущий. Он заставляет текущий поток ждать, пока другой завершится. Есть так же вариант `join(long millis)`, позволяющий ждать либо когда поток завершится, либо когда истечет время.

Например, мы хотим многопоточно вычислить интеграл. Для этого мы разбиваем функцию на много частей, запускаем их вычисления в потоках и от каждого вызываем `join()`, чтобы сложить и получить результат.

1.4.4. Прерывание потока

Метод `interrupt` рекомендует другому потоку завершиться. Если в этот момент в том потоке выполняется метод ожидания (`sleep()`, `join()`, ...), то бросается `InterruptedException`. Дальше его можно поймать и, например, проигнорировать или завершить свою работу. Если же в момент прерывания другой поток что-то активно делал, никакого исключения не будет. В этом случае можно вызвать **нестатический** метод `isInterrupted()`, который вернет, установлен ли у текущего потока флаг прерывания. **Статический** метод `interrupted()` проверяет флаг и сбрасывает его.

То есть код, делающий что-то в цикле, должен выглядеть как-то так:

```
class Worker implements Runnable {
  public void run() {
    try {
      while (!Thread.interrupted()) { ... }
    } catch (InterruptedException e) {}
    // Нас прервали, завершаемся.
  }
}
```

Здесь `try-catch` ловит исключение, которое может броситься от функции ожидания в цикле, а сам цикл выполняется, пока поток не захотят прервать. Такой поток точно прервется, когда его прервут.

1.5. Синхронизация кода

Плюс многопоточного программирования в том, что у всех потоков общие данные. В этом же есть и проблема. Самый простой пример – есть переменная `int i = 0;`, два потока делают `i++;`. В результате можно получить 2, а можно и 1, если первый поток прочитал значение, потом второй прочитал значение, затем каждый из них увеличил значение на 1 и записал. Можно даже получить 0, но об этом в конце семестра.

Чтобы решить эту проблему, надо ввести блокировки и сказать, что эти два куска кода одновременно исполняться не могут. В Java для этого есть `synchronized`-секции. Внутрь такой секции можно войти только когда текущий поток успешно захватил блокировку на объект.

```
synchronized (obj) { // захватили блокировку
  // сделали что хотели
} // сняли блокировку
```

Блокировочным элементом может быть объект любого ссылочного типа. По примитивным типам синхронизироваться нельзя. Поэтому в описанном выше случае придется завести дополнительный объект для синхронизации.

Ниже приведен корректный код, который можно запускать многопоточно.

```
class Counter implements Runnable {
  int i = 0;
  Object obj = new Object();
  public void run() {
    synchronized (obj) {
      i++;
    }
  }
}
```

Обратите внимание, что синхронизация происходит не по самой ссылке, а по тому объекту, на который она ссылается, поэтому в `synchronized ()` можно передавать только ненулевые ссылки, иначе случится `NullPointerException`.

Поскольку каждый объект хранит ссылку на себя (`this`), можно синхронизироваться по ней.

```
class Counter implements Runnable {
  int i = 0;
  public void run() {
    synchronized (this) {
      i++;
    }
  }
}
```

Проговорим подробнее, как происходит синхронизация. Когда поток доходит до секции синхронизации, он обращается к планировщику потоков и говорит, что хочет захватить объект. Если этот объект уже захвачен, планировщик переводит наш поток в состояние Blocked. Как только поток, захвативший объект выходит из секции синхронизации, он сообщает планировщику, что объект ему больше не нужен, и планировщик, когда в следующий раз дойдет до потока, который ждет освободившийся объект, разблокирует его и дает захватить объект.

Здесь есть три важных момента.

Во-первых, предположим, что есть захваченный объект и два потока, которые его ждут. Никто не может предсказать, какой из них проснется первым. То есть потоки просыпаются не в том порядке, в котором они просили блокировку, а в каком-то произвольном. Это называется **нечестная блокировка**.

Во-вторых, `synchronized ()` не бросает `InterruptedException`. Поэтому прервать поток, ожидающий блокировку, нельзя. Это увеличивает шанс получить дедлок. Например, следующий код с большой вероятностью зависнет.

```
Thread t1 = new Thread(() -> {
  synchronized (a) {
    synchronized (b) {
      ...
    }
  }
});
Thread t2 = new Thread(() -> {
  synchronized (b) {
    synchronized (a) {
      ...
    }
  }
});
t1.start();
t2.start();
```

В-третьих, поток может владеть блокировками сразу по нескольким объектам или даже по одному объекту несколько раз. Это полезно, когда, например, есть код, который синхронизируется по объекту и внутри себя вызывает метод, который тоже синхронизируется по этому объекту. Дедлока при этом не произойдет. Блокировка освободится, когда поток отпустит объект столько же раз, сколько захватил.

В Java также есть `synchronized`-методы. Это (почти) то же самое, что метод, в котором всё содержимое обернуто в `synchronized (this){ ... }`. То есть многопоточный счетчик можно переписать так

```
class Counter implements Runnable {
  int i = 0;
  public synchronized void run() {
      i++;
  }
}
```

Заметим, что статические методы не имеют `this`. Поэтому статические `synchronized`-методы синхронизируются по своему классу, то есть

```
class Example {
  public static synchronized void foo() { ... }
}
// эквивалентно
class Example {
  public static void foo() {
    synchronized (Example.class) { ... }
  }
}
```

Важный момент. При вызове `yield()`, `sleep()`, `join()` поток удерживает все свои блокировки. То есть когда вы отправляетесь спать, вы отправляетесь спать со всем багажем блокировок, которые вы захватили.

Рассмотрим пример. У нас есть очередь, которая может хранить не больше одного элемента. И есть два потока – производитель и потребитель. Производитель складывает объекты в очередь, потребитель достает их. Производитель не может ничего положить в очередь, если там занято, а потребитель не может ничего достать, если там ничего нет. У очереди есть объект `private Object data` и методы `public void set(Object data){...}` и `public Object get(){...}`. Посмотрим на пример, как можно написать `set()` и `get()`.

```
public void set(Object data) {
  while (true) { // Активное ожидание
    synchronized (this) {
      if (this.data == null) {
        this.data = data;
        break;
      }
    }
  }
}

public Object get() {
  while (true) { // Активное ожидание
    synchronized (this) {
      if (data != null) {
        Object d = data;
        data = null;
        return d;
      }
    }
  }
}
```

В `set()` мы активно ждем, постоянно проверяя, не освободилась ли очередь, и в итоге кладем объект и выходим. В `get()` мы ждем, пока что-то появится, забираем и выходим.

Вопрос первый. Зачем здесь брать блокировку? Потому что могло так случиться, что два производителя одновременно посмотрели, что место в очереди есть, потом каждый стал складывать туда свой объект, и тот объект, который записался чуть позже, победил. А другой потерялся.

Вопрос второй. Почему нельзя сделать `synchronized`-методы? Потому что как только мы в него войдем, мы заблокируем всем остальным потокам доступ к очереди, и место никогда не освободится (или объект никогда не добавится).

Вопрос третий. Чем плох этот код? Тем, что мы внутри цикла постоянно берем блокировку. Эта операция довольно дорогая, поэтому производительность от этого пострадает. Попробуем с этим побороться.

1.6. Мониторы и условия

В классе `Object` есть три метода: `wait()`, `notify()`, `notifyAll()`. Когда текущий поток вызывает у объекта метод `obj.wait();`, он переходит в состояние Waiting и ждет, что кто-нибудь вызовет у этого же объекта метод `obj.notify();`. Метод `notifyAll()` будит все потоки, которые ждут данный объект, метод `notify()` будит только один поток. Такая блокировка, опять же, не честная, то есть мы не знаем, какой поток проснется первым.

Важно, что все эти методы можно вызывать от объекта, только если текущий поток владеет блокировкой на этот объект. Если это не так, бросится `IllegalMonitorStateException`.

```
Thread t1 = new Thread(() -> {
  . . .
  synchronized (x) {
    try {
      x.wait();
    } catch (InterruptedException e) {} // wait() кидает InterruptedException!
  }
});
Thread t2 = new Thread(() -> {
  . . .
  synchronized (x) {
    x.notify();
  }
});
```

На первый взгляд кажется, что так не работает, потому что первый поток захватывает блокировку по `x` и уходит с ней спать, а второй поток получить блокировку не сможет. На самом же деле метод `wait()` освобождает блокировку в начале ожидания и снова захватывает в конце. Так что у других потоков будет возможность захватить блокировку, пока все ожидающие спят.

Зачем тогда методу `wait()` нужна блокировка? Есть две причины. Во-первых, у каждого потока есть некое множество объектов, которые он заблокировал. И у каждого объекта такого объекта есть флажок, ждем ли мы его. Эта информация нужна планировщику потоков, чтобы он мог нас потом разбудить. Во-вторых, в примере с очередью мы делали какие-то синхронизированные действия с очередью, затем ждали, и продолжали что-то делать с очередью. В таком случае `wait()` автоматически окажется внутри синхронизированного блока, и ему не придется снова ждать блокировку.

Есть также вариант `wait(long time)`, позволяющий ждать только в течение какого-то времени.

Вернемся к примеру с очередью. Посмотрим, как можно реализовать методы `set()` и `get()`.

```
public synchronized void set(Object data) throws InterruptedException {
  if (this.data != null) {
    wait(); // Пассивное ожидание, wait() вызван от this
  }
  this.data = data;
  notify(); // notify() вызван от this
}
public synchronized Object get() throws InterruptedException {
  if (data == null) {
    wait(); // Пассивное ожидание, wait() вызван от this
  }
  Object d = data;
  data = null;
  notify(); // notify() вызван от this
  return d;
}
```

Мы избавились от проблемы с активным ожиданием, однако приобрели несколько других.

Пусть есть два производителя, и они спят. Пришел потребитель, забрал элемент и разбудил одного производителя. Этот производитель добавил элемент и вызвал `notify()`. После этого мог проснуться не потребитель, а второй производитель. Он проснулся, записал свой элемент поверх имеющегося, и всё сломалось.

Но даже с одним производителем есть проблема. Эта проблема называется "внезапное/спонтанное пробуждение". Дело в том, что поток при ожидании `wait()` может проснуться до того, как его уведомили. Это сделали для защиты от дедлоков. Допустим, в кривой программе `notify()` вызвался до вызова `wait()`. Планировщик потоков ленивый. Он один раз проверяет, если ожидающий поток, а потом забывает об этом. Если поток начинает ждать после этого, то он никогда не проснется, если `notify()` не вызовется еще раз. Помимо этого, планировщику выгодно периодически будить все потоки (об этом нам расскажут на ОС). Поэтому иногда случается спонтанное пробуждение.

При `sleep()` и `join()` спонтанное пробуждение случиться не может.

Заменим в нашем коде `if` на `while`.

```
public synchronized void set(Object data) throws InterruptedException {
  while (this.data != null) {
    wait();
  }
  this.data = data;
  notify();
}
public synchronized Object get() throws InterruptedException {
  while (data == null) {
    wait();
  }
  Object d = data;
  data = null;
  notify();
  return d;
}
```

В нормальной ситуации цикл выполнится всего один раз. Но если что-то пошло не так, мы пойдем на второй круг, и ничего не сломается. Так мы избавились от проблем с несколькими производителями и с внезапными пробуждениями. Но проблемы еще остались.

Предположим, есть потребитель и два производителя. Потребитель и первый производитель спят, а второй производитель что-то положил в очередь и вызвал `notify()`. Проснулся второй производитель, увидел, что очередь занята и дальше уснул. В итоге потребителя больше никто не разбудит.

Это можно починить, если производители и потребители будут блокироваться по разным объектам. Тогда код будет выглядеть как-то так.

```
public class Queue {
  Object supplierMonitor = new Object(); // монитор производителя
  Object consumerMonitor = new Object(); // монитор потребителя

  public synchronized void set(Object data) throws InterruptedException {
    synchronized(supplierMonitor) { // захватили монитор производителя
      while (this.data != null) {
        supplierMonitor.wait(); // ждем монитор производителя
      }
    } // отпустили монитор производителя
    this.data = data;
    synchronized(consumerMonitor) { // захватили монитор потребителя
      consumerMonitor.notify(); // уведомили потребителя
    } // отпустили монитор потребителя
  }


```

```
  public synchronized Object get() throws InterruptedException {
    synchronized(consumerMonitor) { // захватили монитор потребителя
      while (data == null) {
        consumerMonitor.wait(); // ждем монитор потребителя
      } // отпустили монитор потребителя
    }
    Object d = data;
    data = null;
    synchronized(supplierMonitor) { // захватили монитор производителя
      supplierMonitor.notify(); // уведомили производителя
    } // отпустили монитор производителя
    return d;
  }
}
```

На собеседованиях любят задавать вопрос, когда может понадобиться создавать `Object`. Это хороший пример такой ситуации.

Рассмотрим еще одну задачу – пул потоков. Есть очередь неограниченного размера, Заказчики складывают в нее задания и ждут, когда их задания выполнятся. Рабочие забирают задания из очереди и выполняют их.

Заказчик кладет свой таск в очередь, уведомляет какого-то работника, что таск появился (в качестве монитора – очередь), и ждет, пока таск выполняется (в качестве монитора – этот таск).

```
Task task = ... ;
queue.add(task);
queue.notify();
while (!task.isOver()) { task.wait(); }
```

Работник ждет, пока очередь станет непустой, забирает задание, после окончания выполнения уведомляет заказчика.

```
while (queue.isEmpty()) { queue.wait(); }
Task task = queue.get();
task.run();
task.setIsOver(true);
task.notify();
```

Пример, когда надо использовать `notifyAll()`, – это инициализация приложения. То есть когда перед началом работы надо сделать какие-то действия. Во время инициализации запускаются много потоков, но они все ждут, когда закончится инициализация. Когда можно начинать работать, надо разбудить сразу все потоки, используя `notifyAll()`.

Пример для самостоятельного изучения (на лекции был показан, но не разбирался).

```

class Calculator extends Thread {
  int total;
  public void run() {
    synchronized (this) {
      for (int i = 0; i < 100; i++) {
        total += i;
      }
      notifyAll();
    }
  }
}

```

```
14 public class Reader extends Thread {
15   Calculator c;
16
17   public Reader(Calculator calc) {
18     c = calc;
19   }
20
21   public void run() {
22     synchronized (c) {
23       try {
24         System.out.println(”Вычисление...”);
25         c.wait();
26       } catch (InterruptedException e) { }
27       System.out.println(”Total: ” + c.total);
28     }
29   }
30
31   public static void main(String[] args) {
32     Calculator calculator = new Calculator();
33     new Reader(calculator).start();
34     new Reader(calculator).start();
35     new Reader(calculator).start();
36     calculator.start();
37   }
38 }
```

1.7. Java Memory Model для чайников

Java Memory Model – это набор соглашений, касающихся поведения программы в многопоточном режиме.

1.7.1. Атомарность

Атомарная операция выполняется как единое целое. То есть пока поток выполняет эту операцию, никакой другой поток ничего делать не может. В Java операции (в том числе и логические) над всеми **примитивными и ссылочными** типами кроме `long` и `double` я вляются атомарными. Это **почти** правда (например, `i++` не является атомарной операцией). Типы `long` и `double` занимают по 8 байтов, поэтому не на всех архитектурах помещаются в машинное слово. Так что атомарность операций с этими типами не гарантируется. Над ссылочными типами есть только операции присваивания и сравнения (==). Они тоже гарантированно атомарные.

Пример. Есть переменные `int a = 0; long b = 0;`. Один поток делает `a = 1; b = −1;`. Другой поток смотрит значения этих переменных. В переменной `a` он может увидеть только 0 или 1. Атомарность гарантирует, что никакого мусора там лежать не будет. В переменной `b` поток может увидеть 0, -1, `0xffffffff00000000` или `0x00000000ffffffff`.

1.7.2. Видимость

Есть переменные `int a = 0; int b = 0;`. Один поток делает `a = 1; b = 2;`. Другой поток выводит значения этих переменных на экран `System.out.println(a, b);` Что может вывестись? Рассмотрим самый нетривиальный вариант: ”0, 2”.

Потоки исполняются на процессоре, а переменные хранятся в общей памяти. Чтобы каждый раз туда не лезть, процессор кеширует значения у себя, производит действия, а потом выгружает обратно. В каком порядке он это делает, мы не знаем. Могло так случиться, что значение `b` уже

выгружено, а значение `a` еще нет. Другой поток смотрит значения в общей памяти, поэтому в данном случае он увидит "0, 2".

Видимость – это когда изменения, сделанные одним потоком, видны другому потоку.

Поток 2 гарантированно увидит изменения, которые сделал поток 1, в следующих случаях:

- Если после изменений поток 1 освободил блокировку, которую захватил поток 2. То есть если в примере обернуть код обоих потоков в `synchronized`-блок **от одного и того же объекта**, то вариантов вывода останется только два: "0, 0" и "1, 2". При этом не обязательно брать блокировку именно по объекту, который будет изменен. Видны будут **все** изменения, произошедшие внутри `synchronized`-блока.
- Если после изменений поток 1 создал поток 2.
- Если поток 2 дождался окончания потока 1 (`t1.join()`).

В остальных случаях правильная видимость не гарантирована.

1.7.3. Упорядоченность

Программа внутри потока выполняется так, как если бы она была написана последовательно. Но с точки зрения других потоков выполнение программы может производиться в произвольном порядке.

Пример. Есть одна переменная `int a = 0;`. Один поток последовательно выполняет над ней два действия: `a = 1; a = 2;`. Другой поток два раза выводит значение этой переменной `print(a); print(a);`. Что он может вывести? Рассмотрим нетривиальные варианты.

Вариант "2 1". Процессор, компилятор и JIT-компилятор имеют право переставлять местами действия при условии, что это не влияет на работу текущего потока. Про другие потоки при этом ничего не гарантируется.

Вариант "2 0". Обращения к общей памяти происходят не напрямую, а через менеджер памяти. У него есть очередь изменений, которые он должен записать в память, а еще он помнит несколько последних изменений. Допустим, первый поток только что сообщил менеджеру, что `a = 2`. Тут же пришел второй поток и спросил, чему равно `a`. Менеджер помнит, что `a = 2` и сообщает об этом потоку. Через некоторое время второй поток снова приходит и спрашивает значение `a`. Менеджер уже забыл, что `a` недавно поменялось, но еще не загрузил значение из очереди в память. Тогда он идет в память, видит там 0 и возвращает.

1.8. volatile-переменные

Операции чтения и записи и простые арифметические операции с `volatile`-переменными, включая `long` и `double`, атомарны (но `i++` всё еще не атомарно).

Чтение и запись `volatile`-переменных происходит не через менеджер памяти, а напрямую. Это не совсем верно, но пока можно считать так. Прямая работа с памятью гарантирует, что изменения переменной видны из других потоков.

Если `volatile`-ссылка изменилась, то данные доступные по ней могли не измениться. Пусть есть `volatile`-переменная `a`, и у неё есть поле `a.x`. Если мы присвоим что-то в `a`, то изменения запишутся сразу в память, но если я поменяю `a.x`, то запить будет происходить через менеджер.

1.9. Singleton

Singleton – это сущность, находящаяся в программе единственном экземпляре. Рассмотрим однопоточную реализацию.

```
1  class Foo {
2    private static Helper helper = null; // собственно экземпляр
3    public static Helper getHelper() { // получение экземпляра
4      if (helper == null) { // если экземпляр еще не создан, создаем
5        helper = new Helper();
6      }
7      return helper; // возвращаем экземпляр
8    }
9  }
```

Если несколько потоков вместе придут просить экземпляр первый раз, то каждый поток создаст себе свой объект, и это плохо.

Можно сделать метод `getHelper()` синхронизированным. Это будет гарантировать, что ровно один поток создаст экземпляр, а остальные сразу после снятия блокировки это увидят. Эта реализация очень дорогая, ведь синхронизацию достаточно делать только при первом обращении.

Рассмотрим реализацию с шаблоном Double-Checking Locking.

```
1  public static Helper getHelper() {
2    if (helper == null) { // проверяем, что еще не создали
3      synchronized(Foo.class) { // захватываем блокировку
4        if (helper == null) { // проверяем еще раз
5          helper = new Helper(); // создаем
6        }
7      }
8    }
9    return helper;
10 }
```

Здесь есть проблема. Порядок выполнения операций может меняться. Поэтому в 5 строчке может сначала выделиться память, ссылке присвоится адрес этой памяти, и только потом вызовется конструктор. В это время другой поток выполнил 2 строчку, понял, что объект создан и вернул его. Поскольку этот поток не делал синхронизацию, он может не увидеть то, что случилось в конструкторе, и ссылка будет указывать на пустую память, поэтому при обращении вылетит `NullPointerException`. Это решается тем, что `helper` делается `volatile`-переменной. Это гарантирует нам атомарность выполнения конструктора.

Но вообще писать `Singleton` надо так:

```
1  public class Singleton {
2    private Singleton() {}
3
4    private class SingletonHolder {
5      public static final Singleton instance = new Singleton();
6    }
7
8    public static Singleton getInstance() {
9      return SingletonHolder.instance;
10   }
11 }
```

Здесь нам всё гарантирует статическая инициализация. То есть нам не дадут класс до того, как завершилась статическая инициализация.

2. Пакет java.util.concurrent

2.1. Блокировки и условия

У synchronized есть несколько проблем – мы не можем проверить, взята ли блокировка, мы отпускаем блокировку в том же куске кода, где захватили, итд.

В пакете `java.util.concurrent.locks` есть интерфейс Lock, который позволяет сделать следующее:

- `lock()` – захватить блокировку
- `lockInterruptibly()` – захватить блокировку с возможностью прерваться (бросает `InterruptedException`).
- `tryLock()` – захватить блокировку, если она не захвачена другим потоком
- `tryLock(long time, TimeUnit unit)` – пытаться захватить блокировку в течение определенного времени (бросает `InterruptedException`)
- `unlock()` – отпустить блокировку
- `newCondition()` – создать условие

Условие – объект интерфейса `Condition`. У него есть следующие методы:

- `await()` – ждать выполнения условия (бросает `InterruptedException`)
- `awaitUntil(Date deadline)` – ждать выполнения условия до наступления дедлайна (бросает `InterruptedException`)
- `awaitUninterruptibly()` – ждать выполнения условия без возможности прерваться
- `signal()` – подать сигнал, что условие выполнилось
- `signalAll()` – подать сигнал всем ожидающим

Чтобы ожидать условие или подать сигнал, надо владеть блокировкой на родительский `Lock` (от которого создан данный `Condition`).

Рассмотрим пример, как можно реализовать производителя.

```
1  void set(Object data) throws InterruptedException {
2    lock.lock(); // захватили блокировку
3    try {
4      while (data != null) {
5        notFull.await(); // notFull создан от lock
6      }
7      this.data = data;
8      notEmpty.signal(); // notEmpty создан от lock
9    } finally {
10     lock.unlock(); // отпустили блокировку
11   }
12 }
```

Теперь производители будят только потребителей, а потребители только производителей.

При использовании локов, в конце необходимо вручную отпускать блокировку. В том числе и если в коде бросилось исключение. Поэтому используют `try-finally`.

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

Есть тонкий момент. Если поток, владеющий блокировкой, вдруг завершается (например вылетает необработанное исключение), то никто не знает, отпустится ли эта блокировка (это зависит от ОС). Чтобы избежать неопределенного поведения, блокировку надо всегда отпускать, когда она не нужна.

Стандартная реализация интерфейса `Lock` – класс `ReentrantLock`. У него есть дополнительные методы:

- `isFair()` – проверить, является ли блокировка честной
- `isLocked()` – проверить, захвачена ли блокировка

Честность блокировки можно передать в конструктор. Далее менять её нельзя. Важно помнить, что честная блокировка работает медленнее обычной. Одна из причин – потому что планировщик будет перебирать ожидающие потоки, пока не найдет тот, который начал ждать раньше всех. Есть более серьезная причина, но о ней позже.

Также у `ReentrantLock` можно спросить информацию о потоках, которые его ждут.

- `getQueuedThreads()` – коллекция потоков, ожидающих лок
- `getQueueLength()` – количество потоков, ожидающих лок
- `hasQueuedThread(Thread thread)` – ждет ли данный поток этот лок
- `hasQueuedThreads()` – ждут ли какие-нибудь потоки этот лок
- `getWaitingThreads(Condition condition)` – коллекция потоков, ожидающих данное условие
- `getWaitQueueLength(Condition condition)` – количество потоков, ожидающих данное условие

Пример очереди для производителей/потребителей.

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {...}
  public Object take() throws InterruptedException {...}
}
```

Методы `take()` и `put()` бросают `InterruptedException`, потому что они внутри ожидают соответствующие кондишены.

Рассмотрим еще один шаблон – читатели и писатели. Есть один ресурс, который некоторые читают, а некоторые пишут. Писать одновременно может только один, читать могут сколько угодно, и нельзя писать, пока кто-то читает.

Типичный пример – когда есть файл конфигурации, все могут что-то в нем исправлять, а потом все могут прочитать изменения.

Для этой задачи есть интерфейс `ReadWriteLock` с методами `readLock()` и `writeLock()`, которые возвращают объекты `Lock` для читателей и писателей соответственно. Реализация этого интерфейса – `ReentrantReadWriteLock`.

2.2. Управление заданиями

Раньше, когда нам надо было распараллелить какую-то задачу, мы разбивали её на куски и для каждого создавали новый поток, который этот кусок делал и завершался. Это плохо с точки зрения производительности, потому что много времени потратится на создание потоков и переключение контекста.

Обычно люди создают пул потоков и передают эти куски выполняться в нем. Это позволяет переиспользовать и не плодить их слишком много.

`Executor` – это интерфейс, имеющий метод `execute(Runnable task)`, который может выполнять задания

- В том же потоке
- В новом потоке
- В фиксированном пуле потоков
- В наращиваемом пуле потоков

В основном используют последние два варианта. При добавлении задачи в фиксированный пул, если все потоки заняты, задача ждет, пока кто-то освободится, а в наращиваемом для неё создается новый поток, который потом либо добавляется в пул и переиспользуется, либо умирает.

От `Executor` наследуется интерфейс `ExecutorService` с методами

- `Future<?> submit(Runnable task)` – выполнить задание
- `Future<V> submit(Callable<V> foo)` – выполнить функцию
- `List<Future> invokeAll(Collection<? extends Callable<V>> tasks)` – выполнить все функции
- `V invokeAny(Collection<? extends Callable<V>> tasks)` – успешно выполнить любую функцию

Недостаток `Runnable` в том, что он не может ничего вернуть после выполнения. Поэтому есть интерфейс `Callable<V>` с методом `V call()`. По сути это как `Supplier`, но используется для многопоточных задач.

Методы `submit()` и `invokeAll()` не могут сразу вернуть результат выполнения, потому что функция может быть выполнена не сразу. Поэтому они возвращают объект интерфейса `Future<V>`.

`Future<V>` имеет методы

- `V get()` – получить результат (бросает `InterruptedException`)
- `V get(long timeout, TimeUnit unit)` – подождать результат определенное время и, если не дождались, бросить `TimeoutException`
- `isDone()` – проверить, закончилось ли выполнение
- `cancel(boolean mayInterruptWhenRunning)` – прервать выполнение
- `isCancelled()` – проверить, прервано ли выполнение

Внутри `ExecutorService` помимо рабочих потоков есть главный поток, занимающийся распределением заданий. И этот поток пользовательский (не демон). Это значит, что пока он не завершился, программа будет работать. В связи с этим есть специальные методы:

- `shutdown()` – прекратить прием заданий и, когда все принятые задания выполнятся, завершить основной поток
- `List<Runnable> shutdownNow()` – прекратить прием заданий и, вернуть все принятые, но невыполненные задания
- `isShutdown()` – проверить, что прием заданий завершен
- `isTerminated()` – проверить, что все задания завершились
- `awaitTermination(long timeout, TimeUnit unit)` – ждать, пока всё завершится (бросает `InterruptedException`)

Чтобы создать пул потоков, надо у класса `java.util.concurrent.Executors` вызвать один из статических методов:

- `newCachedThreadPool()` – расширяемый пул потоков.
- `newFixedThreadPool(int n)` – фиксированный пул потоков.
- `newSingleThreadExecutor()` – однопоточный пул потоков.

Однопоточный пул потоков в основном используется для работы с устройствами ввода-вывода или с сетью. В сокет может одновременно писать только один поток, иначе всё смешается. Но хочется делать это не в главном потоке, а в тред пуле. По сути это то же самое, что фиксированный пул на один поток, но его выделили в отдельный класс из-за частого использования.

Примеры использования тредпулов есть здесь, здесь и здесь.

Интерфейс `ScheduledExecutorService` умеет выполнять задания отложенно или с некоторой периодичностью:

- `schedule(Callable task, long delay, TimeUnit unit)` – выполнить задание через время `delay`.
- `schedule(Runnable task, long delay, TimeUnit unit)` – выполнить задание через время `delay`.

- `scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)` – выполнять каждые `period` единиц времени.
- `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)` – выполнять с разницей `delay` между окончанием одного выполнения и началом следующего.

Все методы возвращают `ScheduledFuture`.

2.3. Примитивы синхронизации

Рассмотрим основные примитивы синхронизации в `java.util.concurrent`.

2.3.1. Semaphore

Семафор – это штука, похожая на `Lock`, но позволяет нескольким (не более чем n) потокам себя захватывать. Число n передается в конструктор.

Например, можно сделать пул на 10 потоков, в котором поступающие задания будут захватывать семафор, и если заданий больше 10, они сидят и ждут, когда им отдадут блокировку.

Попросить блокировку можно методом `acquire()`, отдать блокировку – `release()`. Можно попросить или отдать блокировку сразу несколько раз методами `acquire(int permits)` и `release(int permits)` соответственно.

2.3.2. CyclicBarier

Циклический (многоразовый) барьер. Потоки подбегают к барьеру, упираются в него и ждут. Когда потоков становится n, барьер падает, и все потоки продолжают выполняться.

Типичным примером является многопользовательская игра. Надо дождаться, пока все игроки сделают ход и после этого начать следующий раунд.

Метод `await()` или `await(long timeout, TimeUnit unit)` позволяет сообщить, что поток прибыл к барьеру и начал ждать. Методы `isBroken()` и `reset()` позволяют проверить. сломан ли барьер и восстановить его соответственно. В конструктор можно передать `Runnable`, который будет срабатывать, когда барьер сломался.

Пример использования циклического барьера – скачки.

2.3.3. CountDownLatch

Защелка чем-то похожа на барьер. Сначала она задвинута на n отметок, потоки её выдвигают на одну отметку, и на нулевой отметке она открывается. Отличие от барьера состоит в том, что здесь разделены команды ожидания и выдвигания. Это значит, что одни потоки могут ждать, а другие выдвигать.

Пример использования защелки – инициализация. При старте приложения мы хотим в нескольких потоках выполнить какие-то начальные действия, дождаться их завершение и стартовать все основные потоки.

Метод `await()` или `await(long timeout, TimeUnit unit)` ждёт открывания двери, `countDown()` выдвигает защелку на единичку, `getCount()` проверяет, на сколько защелка выдвинута.

Пример использования защелки.

2.4. Атомарные операции

Помимо стандартных видов атомарных операций (чтение, запись, ...) люди вводят дополнительные виды: `get` – прочитать, `set` – записать, `getAndSet` – прочитать старое значение и записать новое, `compareAndSet` – если старое значение равно тому что надо, записать новое.

Метод `compareAndSet()` можно использовать в следующей идиоме:

```
do {
  oldVal = v.get();
  newVal = process(oldVal);
} while (!v.compareAndSet(oldVal, newVal));
```

Условие в цикле выполнится, если во время наших действий другой поток успел поменять значение переменной. То есть цикл будет крутиться до тех пор, пока ему не удастся атомарно считать значение и записать новое. При этом работает этот код без блокировок.

Можно реализовать неблокирующий `synchronized`:

```
while (!v.compareAndSet(0, 1)); // получаем доступ к ресурсу
    // производим действия
v.set(0);                       // освобождаем ресурс
```

Именно таким образом реализован класс `Lock`, но написан от не на джаве, а на нативном коде. А `synchronized` реализован как-то по-другому.

Вышеперечисленные методы (и некоторые другие) есть у классов `AtomicBoolean`, `AtomicInteger`, `AtomicReference`, итд.

2.5. Немного о скорости работы

Есть много потоков, и каждый из них 1000000 раз делает `x++`. Сравним скорость работы такой программы при разных подходах к синхронизации.

Самым медленным будет подход с честной блокировкой. Пусть один поток захватил блокировку, другой стал ждать. Первый поток выполнил работу, и у него есть время сделать это еще раз. Он снова просит блокировку, но не получает её, так как другой поток запросил её раньше. Получается, что после каждого инкремента происходит переключение контекста, что очень сильно замедляет работу программы.

На втором месте идёт `synchronized`. Он нечестный, поэтому более быстрый.

`Lock` работает еще немного лучше.

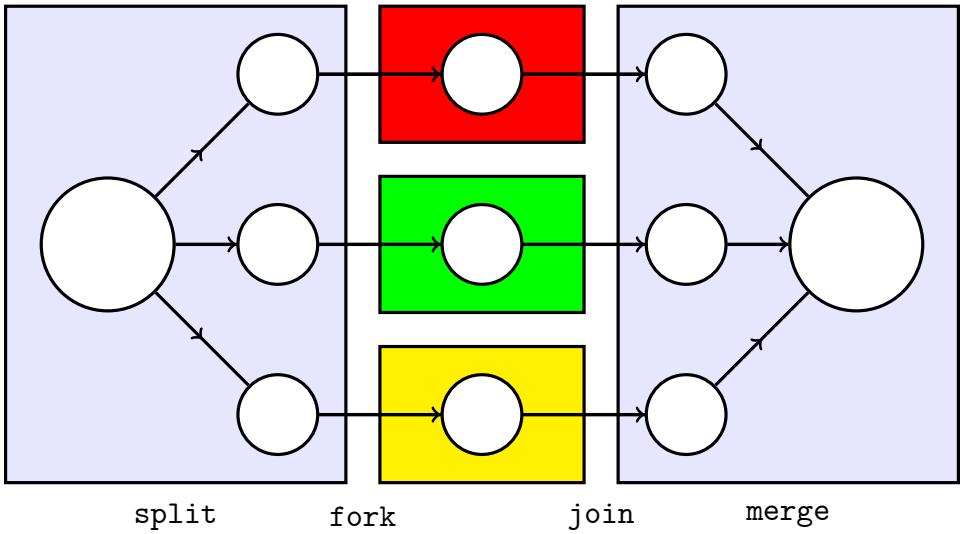
Самым быстрым будет использование `AtomicInt` без всяких блокировок, потому что он реализован нативно и супер оптимизирован.

Какой можно сделать вывод? В данной задаче есть много потоков, которые выполняют много маленьких действий, требующих синхронизации. В этом случае активное ожидание, через которое реализован `Lock`, работает лучше. В общем же случае результаты очень сильно зависят от процессора, версии джавы, итд. Что можно сказать точно, так это то, что **быстрый код не надо писать на джаве**. Для этого есть нормальные быстрые языки, например C++.

3. Fork/Join

3.1. Идеология

Идея Fork/Join основана на принципе "разделяй и властвуй". Берем большую задачу, разбиваем на маленькие подзадачи и решаем их параллельно. Хороший пример – сортировка слиянием. Делим массив на две части, сортируем каждую по отдельности и сливаем результаты воедино. То есть в этой идеологии задача делится на аналогичные ей подзадачи меньшего размера. Чтобы такой процесс можно было распараллелить, задачи должны быть независимы друг от друга.



На схеме показаны этапы решения задачи в идеологии Fork/Join. Сначала задача разбивается на подзадачи (`split`), затем каждая подзадача отправляется в отдельный поток для выполнения (`fork`). Далее главный поток ожидает, пока все подзадачи завершатся (`join`). Наконец, главный поток собирает результаты подзадач и вычисляет ответ на основную задачу (`merge`).

Рассмотрим план реализации, предложенный г-ном Дугласом Ли.

```
Result solve(Problem problem) {
  if (problem.isSmall()) { // Если подзадача маленькая
    // решаем напрямую, в текущем потоке
  } else {
    // делим на части
    // форкаем все части
    // джойним все части
    // собираем результат
  }
}
```

Здесь мы рекурсивно делим подзадачи на еще меньшие подзадачи, пока они не станут слишком малы. Маленькие задачи выгоднее решить сразу, не тратя время на создание потоков. К тому же, надо когда-то закончить разбивать подзадачи, иначе рекурсия не остановится. В такой реализации есть ряд проблем. Рассмотрим некоторые из них.

Проблема первая. Потоков получается слишком много. Это плохо сказывается на производительности.

Вторая проблема в том, что поток, который сделал `fork` и ждет завершения подзадач, в это время ничего не делает. То есть большинство потоков будут тупить вместо выполнения полезной работы.

Решение заключается в том, чтобы при вызове `join()` поток не ждал, а начинал выполнять другую задачу. Для этого создадим Fork/Join пул. В нем будут сидеть несколько потоков. Первый поток получает задачу, разбивает её на подзадачи, раздаёт их другим потокам и начинает ждать. Другие потоки тоже разбивают свои подзадачи, и раздают друг другу (в том числе и первому). Первый поток получает маленькую подзадачу, выполняет её, итд. Таким образом, если есть еще не выполненные подзадачи, все имеющиеся потоки заняты их решением.

Задачи для Fork/Join пула – это объекты **абстрактного** класса `ForkJoinTask`, который является в каком-то смысле легковесным аналогом потока. У этого класса есть наследники:

- `RecursiveAction` – задача, которая ничего не возвращает, является аналогом `Runnable`
- `RecursiveTask` – задача, которая что-то возвращает, является аналогом `Callable`

Наконец, сам пул реализован в классе `ForkJoinPool`, который снаружи выглядит как обычный пул потоков. Его основные методы:

- `void execute(ForkJoinTask<?> task)` – ничего не возвращает
- `T invoke(ForkJoinTask<T> task)` – возвращает результат выполнения
- `ForkJoinTask<T> submit(ForkJoinTask<T> task)` – отправляет задачу на выполнение и возвращает задачу, принятую к исполнению (`ForkJoinTask` реализует интерфейс `Future`)

3.2. Балансировка задач

Важная и сложная часть реализации – это как распределять задачи между потоками. В нашей схеме есть задачи разных размеров, и время выполнения этих задач разное. Более того, в один `ForkJoinPool` можно одновременно передавать задачи разных типов, которые не понятно, как сравнивать по размеру.

Для максимальной эффективности хочется все потоки занять одинаково. Для этого есть несколько подходов:

- **Арбитраж задач**. Есть общая очередь задач, потоки забирают из неё задачи, разбивают и складывают подзадачи обратно в очередь.

  Такой подход не очень хорош, потому что потокам очень часто приходится брать блокировку на очередь, и это тормозит процесс.

  Можно, помимо общей очереди, для каждого потока сделать собственный буфер ограниченного размера. Поток будет хранить подзадачи у себя, пока их не станет слишком много, а потом начнет складывать в очередь.

  Это приведет к неравномерному распределению задач, то есть может так получиться, что один поток решил всю задачу, пока другие ничего не делали.

- **Work dealing**. У каждого потока своя очередь. Когда поток понимает, что ему тяжело, он просит других забрать у него часть задач.

  Минус этого подхода в том, что, во-первых, надо как-то понимать, когда потоку стало тяжело, а во-вторых, опрашивать другие потоки, могут ли они помочь. Это занимает довольно много дополнительного времени.

- **Work stealing**. У каждого потока своя очередь. Когда поток понимает, что ему нечего делать, он идет воровать задачи у перегруженных.

  В отличие от Work dealing, здесь дополнительной работой занимается разгруженный поток, что хорошо.

Как эффективнее реализовать подход Work stealing?

У каждого потока есть очередь задач. Он забирает задачи из головы, и подзадачи тоже кладет в голову. А свободные потоки воруют задачи из хвоста очереди. Таким образом в голове всегда будут маленькие задачи, которые поток решает сам, а в хвосте – большие, и другие потоки их воруют. А воровать выгодно именно большие подзадачи.

Еще одно преимущество брать задачи с разных концов в том, что ворующий поток не заблокируется, когда будет брать блокировку на задачу, потому что поток-владелец возьмет блокировку на другую задачу. Поток может заблокироваться, если в очереди один элемент, или если два потока одновременно пытаются своровать у третьего. Но такие ситуации происходят только при низкой загруженности пула, так что не жалко.

Однако владельцу иногда выгодно брать задачи из своего хвоста. Во-первых, если брать только из головы, то старые большие задачи могут долго лежать и не выполняться. Во-вторых, эффективнее иметь много больших задач а не маленьких, чтобы не тратить время на переключение между ними.

Куда добавлять внешние задачи? Можно сложить в общую очередь, и самый не загруженный поток, когда захочет, заберет её себе. Но так мы опять получаем точку синхронизации, которая замедляет все потоки.

Поэтому у каждого потока есть своя очередь для внешних задач. Когда задачу добавляют, она уходит произвольному потоку, потому что потоки загружены более-менее одинаково. В определенный момент поток просмотрит свою очередь внешних задач, заберет оттуда задачу и добавит себе в хвост, потому что большие задачи должны быть в хвосте.

3.3. Еще немного о скорости работы

Рассмотрим пример. Задача – есть полное бинарное дерево, в каждой вершине записано число. Надо посчитать сумму всех чисел. Хотим распараллелить. Для этого в каждой вершине создадим две подзадачи – левое и правое поддерево, а потом сложим.

Чтобы создать задачу для `ForkJoinPool`, надо наследоваться от `RecursiveTask` и переопределить метод `compute()`.

```
class ValueSumCounter extends RecursiveTask<Long> {
  private final Node node;
  public ValueSumCounter(Node node) {
    this.node = node;
  }
  @Override protected Long compute() {
    if (node.depth() < 15) { // Маленькое дерево. Решаем в лоб.
      return Node.countSum(node);
    }
    long sum = node.getValue();
    List<ValueSumCounter> subTasks = new LinkedList<>();
    for (Node child : node.getChildren()) {
      ValueSumCounter task = new ValueSumCounter(child);
      task.fork(); // Запускаем подзадачу.
      subTasks.add(task);
    }
```

```
    for (ValueSumCounter task : subTasks) {
      sum += task.join(); // Дожидаться выполнения надо, когда все подзадачи уже запущены!
    }
    return sum;
  }
}
```

Отправляем нашу задачу в пул:

```
Node root = BasicNode.createTree(22); // полное дерево глубины 22
ForkJoinPool pool = new ForkJoinPool(4); // пул на 4 потока
long t = System.currentTimeMillis(); // время начала выполнения
pool.invoke(new ValueSumCounter(root)); // запускаем
System.out.println(System.currentTimeMillis() - t); // выводим, сколько времени выполнялся код
```

Если запускать одну и ту же программу много раз, можно заметить, что работает она всегда разное время, причем показатели могут отличаться аж в 30 раз. На самом деле, если несколько раз запустить однопоточный код, его время работы тоже будет меняться. Сравнив результаты, можно увидеть, что в лучшем случае многопоточное решение действительно работает быстрее.

Скопируем наш код два раза в одной программе и запустим. Довольно часто второй запуск будет работать быстрее первого. Это происходит потому, что включается JIT, оптимизирует код, и второй запуск работает оптимально.

Делаем вывод, что запускать небольшие задачи в `ForkJoinPool` бессмысленно, так как JIT включиться не успеет, и Java будет тратить много времени на неоптимальное управление потоками и задачами.

3.4. ParallelStream в действии

Параллельные стримы внутри себя используют `ForkJoinPool`. Стрим разбивается на две части, рекурсивно параллельно производит вычисления, и соединяет результат.

Пример. В цикле 100 раз берется стрим чисел, удаляются нечетные, остальные сортируются и собираются в список. Каждый раз выводится количество элементов, время выполнения и количество задействованных потоков.

```
for (int i = 0; i < 100; i++) {
  long start = System.currentTimeMillis();
  List<Integer> even = numbers.stream()
  .filter(n -> n % 2 == 0).sorted().collect(Collectors.toList());
  System.out.printf("%d elements computed in %5d msecs with %d threads\n",
         even.size(), System.currentTimeMillis() - start, Thread.activeCount());
}
```

Смотрим, что вывелось.

```
4999022 elements computed in 9793 msecs with 1 threads
4999022 elements computed in 913 msecs with 1 threads
4999022 elements computed in 4618 msecs with 1 threads
4999022 elements computed in 877 msecs with 1 threads
. . .
```

Второй раз получилось быстрее чем первый, потому что сработал JIT. Далее снова видим замедление, потому что периодически приходит сборщик мусора и замедляет программу.

Поменяем в третьей строчке `numbers.stream()` на `numbers.parallelstream()`. Результат:

```
5000749 elements computed in 663 msecs with 4 threads
5000749 elements computed in 450 msecs with 4 threads
5000749 elements computed in 715 msecs with 4 threads
5000749 elements computed in 466 msecs with 4 threads
. . .
```

По-прежнему видим скачки, но в целом стало работать намного быстрее. Количество потоков выбирается автоматически исходя из количества ядер на процессоре.

Теперь запустим параллельный стрим один раз, насильно попросив `ForkJoinPool` использовать 4 потока. Для этого запустим java-машину с ключом `-Djava.util.concurrent.ForkJoinPool.common.parallelism=4`. И выведем количество и названия работавших потоков.

```
5002528 elements computed in 563 msecs with 5 threads
ForkJoinPool.commonPool-worker-0
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-3
main
```

Видим, что помимо вновь созданных, основной поток тоже участвует в вычислении стрима. Во-первых, это сделано, чтобы поток, вызвавший вычисление, не простаивал. Во-вторых, это защищает от плохо написанного кода.

Дело в том, что **все** параллельные стримы выполняются в одном пуле. Если какая-то криво написанная задача заблокирует все потоки, то другие задачи в этом пуле выполняться не будут. Чтобы этого избежать, каждый поток, который запускает стрим, берет на себя часть работы, чтобы задача гарантированно выполнилась, даже если все общие потоки заблокировались.

`ParallelStream` можно заставить использовать нужный нам `ForkJoinPool`, просто запустив вычисление стрима в нем. В таком случае стрим будет вычисляться не в общем пуле, а в нашем.

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
forkJoinPool.submit(() -> {
  return numbers.parallelStream().filter(n -> n % 2 == 0).sorted()
  .collect(Collectors.toList());
});
```

Как это работает? Параллельный стрим добавляет себя в пул. После этого каждая подзадача добавляет свои подзадачи в тот пул, в котором сидит. А сидит она как раз в нашем пуле.

Еще один плюс такого подхода в том, что стрим не задействует вызывающий поток, то есть пока он вычисляется, наш поток может сделать еще что-нибудь.

3.5. Проблема общего `ForkJoinPool`

Есть список `url`'ов, которые надо пропинговать. Используем параллельные стримы.

```
Stream<String> urls = Files.lines(Paths.get("urlsToCheck.txt"));
List<String> errors = urls.parallel().filter(url -> {
  // Отправляем запрос и ждем ответа 200 мс. Если не дождались, возвращаем false.
}).collect(toList());
```

Проблема в том, что если все url не отвечают, то потоки общего пула долго ждут и не могут вычислять другие стримы. Поэтому такой код надо запускать в собственном пуле.

3.6. Проблема вложенных параллельных стримов

Следующая задача. Надо заполнить большую матрицу случайными числами. Решение:

```
IntStream.range(0, 10000).parallel().forEach(i -> { // параллелимся по строкам
  results[i][0] = (int) Math.round(Math.random() * 100);
  IntStream.range(1, 9999).parallel().forEach((int j) -> { // параллелимся по столбцам
    results[i][j] = (int) Math.round(Math.random() * 1000);
  });
});
```

Как ни странно, такой код будет работать медленнее, чем если мы будем параллелиться только по строкам, но не по столбцам. На это есть несколько причин.

Во-первых, мы наплодили много слишком задач. Наш пул уже занят параллельным заполнением строк, и распараллеливание по столбцам не будет эффективным.

Во-вторых, теперь в пуле задачи разных типов. Пул пытается отдавать им примерно поровну времени. Но в этой задаче надо больше времени отдавать задачам, непосредственно заполняющим строчки. В результате эффективность страдает.

3.7. Проблема Boxing/Unboxing

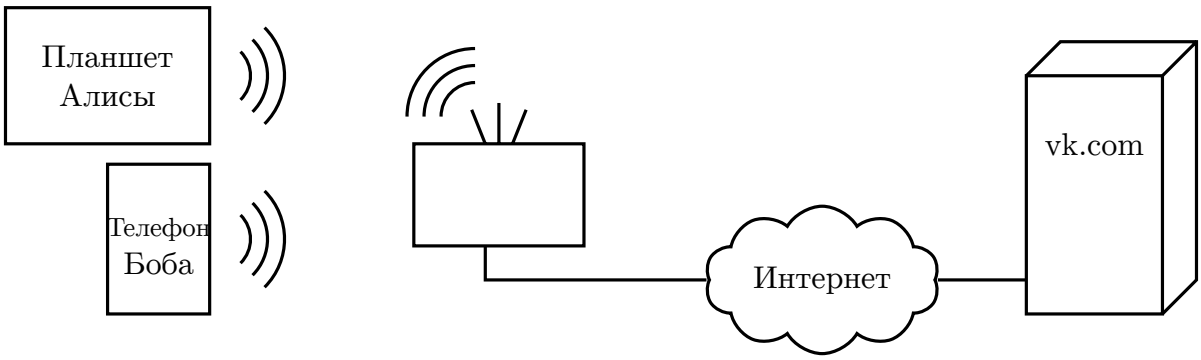
В коде ниже постоянно происходит Boxing и Unboxing. Это сильно замедляет работу.

```
List<Integer> even = numbers.parallelStream()
.filter(n -> n % 2 == 0)
.sorted()
.collect(Collectors.toList());
```

Следующий вариант будет работать быстрее:

```
List<Integer> even = numbers.parallelStream()
.mapToInt(n -> n) // переводим всё в int.
.filter(n -> n % 2 == 0)
.sorted()
.boxed() // переводим в Integer.
.collect(Collectors.toList());
```

4. Стек протоколов TCP/IP



Взаимодействие устройств в сети можно рассматривать на разных уровнях абстракции. На каждом уровне используются специальные протоколы – соглашения о том, как интерпретировать передаваемые данные.

Уровень	**Данные**	**Протоколы**
`Application`	Непосредственно запрос	`http, ftp, ssh...`
`Transport`	`Port1, Port2`	`TCP, UDP...`
`Network`	`IP1, IP2`	`IP, ICMP, IC-MP...`
`Link`	`MAC1, MAC2`	`ARP, Ethernet, Wi-Fi...`

Посылаемый запрос формируется следующим образом: Есть данные для отправки. Это пакет уровня `Application`. Он оборачивается в пакет уровня `Transport`, итд. Пакет уровня `Link` отправляется в сеть. Получатель проверяет, что пакет действительно отправлен ему, разворачивает и отдает на уровень выше, итд.

То есть пакет, передающийся по сети – это пакет с пакетами.

4.1. Уровень `Network`

Чтобы можно было обмениваться данными по сети, нужна адресация. У каждого устройства в сети есть `IP`-адрес (Internet Protocol). Он занимает 4 байта. Люди обычно записывают его четырьмя десятичными числами, например 194.85.248.10.

Данные можно посылать разными способами. Алиса может послать что-то Бобу, а может в ВК. Разница заключается в том, что Бобу можно передать данные напрямую, а в ВК – через коробочку с антеннами, потом через облачко, итд. Чтобы различать такие случаи, у каждого устройства есть также **маска** (`Mask`). Она тоже занимает 4 байта, но в ней сначала идет сколько-то единиц, а потом нули. Маску записывают либо в виде четырех чисел (255.255.255.0), либо к IP-адресу в конец приписывают количество единиц (194.85.248.10/24).

Маску используют чтобы определить, лежат ли два устройства в одной сети. Для этого Считают побитовые `&` адресов с маской и сравнивают результаты.

Алиса хочет отправить данные в ВК. Она проверяет, что `IP`-адреса её планшета и ВК не лежат в одной сети. Тогда она отправляет данные через облачко. Для этого есть **шлюз** (`Gateway`). Это устройство, имеющее доступ к облачку (на картинке – коробочка с антеннами), и его `IP`-адрес.

Всевозможных IP-адресов меньше, чем устройств в интернете. Есть несколько групп IP-адресов, которые запрещается использовать в интернете. Вот некоторые из них:

$$192.168.*.*$$

$$127.*.*.*$$

$$10.*.*.*$$

Такие адреса используются только в локальных сетях. Пакеты с разных устройств поступают на коробочку с антеннами, после чего адреса их отправителей подменяются на один общий. Когда приходит ответ на запрос, адрес подменяется обратно и ответ приходит тому, кому надо. Этот механизм называется NAT (Network Address Translation).

В реальности пользователь подключен к провайдеру, который может быть подключен к более общему провайдеру, итд. Тогда при передаче пакета, IP-адрес может заменяться несколько раз. Можно у провайдера купить "белый" IP-адрес, который можно использовать в интернете, и он подменяться не будет.

Наряду с 4-байтовым протоколом IPv4, есть протокол IPv6. В нем адрес занимает 16 байтов, и таких адресов хватит на всех. Но этот протокол не поддерживается почти никакими провайдерами.

Рассмотрим подробнее, как устроено облачко. От коробочки с антеннами идет провод к маршрутизатору, общему на один дом. От него идет провод к центральному маршрутизатору в районе, итд. У каждого маршрутизатора в этой цепочке есть IP-адрес, маска и шлюз. Когда приходит пакет, маршрутизатор смотрит адрес получателя. Если этот адрес лежит с ним в одной сети, пакет отправляется по адресу. Иначе пакет через шлюз отправляется на следующий маршрутизатор более глобальной сети.

Когда запрос Алисы пришел на сервер ВК, тот формирует ответ и хочет его отправить обратно Алисе. Для этого ему нужен её IP-адрес. Этот адрес был отправлен вместе с запросом. Не забываем, что при передаче запроса, IP-адрес отправителя может меняться.

Таким образом, на уровне Network внешняя обертка содержит IP1 – адрес отправителя и IP2 – адрес получателя.

4.2. Уровень Link

На этом уровне происходит взаимодействие устройств в физической среде, где передается информация. Например, Wi-Fi, Ethernet, ... Для определенности будем рассматривать Ethernet.

На физическом уровне устройствам тоже нужны адреса. У сетевой карты каждого устройства есть (физический) MAC-адрес (Media Access Control). Он занимает 6 байтов и записывается в 16-ричном виде через двоеточие, например 8F:E2:07:54:12:20. Изначально планировалось, что у каждого устройства в мире будет уникальный MAC-адрес. Сейчас это не так. Более того, на некоторых устройствах MAC-адрес можно менять.

Как вообще работает Ethernet? Есть провод. По нему идут 1 и 0. Они группируются в пакеты, между пакетами есть паузы, чтобы определять границы пакетов. Допустим, устройство получило некий пакет. Оно смотрит на MAC-адрес получателя, и если этот адрес не совпадает с адресом этого устройства, выкидывает полученный пакет. Если же адрес совпал, пакет разворачивается и передается выше (на уровень Network). На уровне Network получатель снова разворачивает этот пакет и передает еще выше.

4.2.1. Протокол ARP

Пусть Алиса знает IP-адрес получателя. Чтобы через Ethernet отправить ему данные, необходимо как-то узнать его MAC-адрес. Для этого используется протокол ARP (Address Resolution Protocol). В локальной сети есть выделенные широковещательные адреса. Широковещательный MAC-адрес всегда равен FF:FF:FF:FF:FF:FF, IP-адрес – самый последний адрес, который может быть в данной сети. То есть в сети с маской 255.255.254.0, содержащей адрес 194.85.248.10, широковещательный IP-адрес будет 194.85.249.255.

Широковещательные пакеты, отправленные по этим адресам, обязаны обработать все устройства. В качестве содержимого такой пакет имеет IP-адрес устройства, у которого мы хотим узнать MAC-адрес. Когда все устройства в сети получат запросы, они сравнят этот IP-адрес со своим и ответят только если совпадет. Когда ответ придет Алисе, в нем будет содержаться MAC-адрес устройства, которое послало ответ. Так Алиса узнает нужный ей MAC-адрес.

Этот процесс можно взломать. Такой взлом называется ARP-spoofing. В сеть внедряется устройство, которое посылает ответы на все широковещательные запросы. В результате все дальнейшие запросы приходят на его MAC-адрес.

Что делать, если пришло несколько ответов на широковещательный запрос? Есть два подхода – верить первому пришедшему или последнему пришедшему ответу. При ARP-spoofing обычно посылают очень много ответов, чтобы повысить шанс, что одному из них поверят.

Чтобы каждый раз не посылать ARP-запросы и снизить нагрузку на сеть, MAC-адреса кешируются в памяти устройства. Также, когда устройство включается, оно отправляет широковещательный пакет, похожий на ARP-ответ, только широковещательный. Пакет содержит IP- и MAC-адрес устройства, чтобы другие могли их закешировать.

Итого, на уровне Link внешняя обертка содержит MAC1 и MAC2 – адреса получателя и отправителя, соответственно. Обычный компьютер, получив пакет, проверяет, что MAC-адрес получателя – его и разворачивает. Далее он проверяет. что IP-адрес – его. Если нет, то пакет выкидывается. Маршрутизатор действует иначе. Если MAC-адрес совпал, а IP-адрес – нет, то он передает его в более внешнюю сеть.

4.3. Уровень Application

На сервере ВК крутится много приложений. Когда пакет приходит на сервер, надо как-то понять, какое именно приложение должно его обработать. Для этого вводят понятие **порт**. Это 2-байтное число (от 1 до $2^{16} - 1$). Оно, условно говоря, является идентификатором приложения-обработчика.

Порт можно указать вручную в адресной строке браузера (http://vk.com:3876). По умолчанию браузер отправляет HTTP-запросы на 80 порт, HTTPS-запросы – на 443 порт, итп.

На сервере есть программы, которые слушают определенные порты (Веб-сервер случает 80 порт). Когда сервер получает запрос, он проверяет, слушает ли кто-то данный порт. Если да, то пакет передается слушающему приложению, а иначе выбрасывается.

Разные приложения не могут слушать один порт. Чтобы приложение начало слушать порт, надо либо попросить конкретный порт у ОС (если порт будет занят, вернется ошибка), либо попросить выделить любой порт. Тогда ОС вернет номер выданного порта. В Linux порты с 1 по 1024 зарезервированы, и для их прослушивания необходимы root права. Бóльшая часть портов до 16000 уже расписано. Их можно использовать, но может возникнуть конфликт.

Одно приложение может слушать несколько портов. Например, у браузера отдельный порт на каждую вкладку.

Когда сервер отправляет ответ, он тоже должен указать порт, на который ответ должен прийти. Поэтому в запросе указывается порт отправителя и порт получателя.

4.4. Механизм NAT

Изучим подробнее работу NAT. Когда шлюз хочет отправить пакет во внешнюю сеть, он подменяет IP-адрес **и порт** отправителя и сохраняет у себя, какому старому порту и IP-адресу соответствует новый порт, и отправляет. Когда на новый порт приходит ответ от сервера, шлюз смотрит в табличку и отправляет ответ по нужному адресу и на нужный порт. Такой NAT называется **динамическим**.

Если в сети, где много пользователей (например, в аэропорте), начать качать торренты, у маршрутизатора могут закончиться порты, потому что торрент для скачивания каждого кусочка файла заводит новое соединение. Когда порты заканчиваются, маршрутизатор начинает либо отвергать новые соединения, либо удалять информацию о старых. В результате у всех остальных пользователей интернет либо не работает, либо работает очень медленно.

В нормальных ситуациях у шлюза не один IP-адрес, который он может давать в процессе NAT, а много. Поэтому количество шлюзов умножается еще на количество IP-адресов.

Из-за подмены IP-адресов, сервер не может отправить запрос на устройство, пока не получит запрос от этого устройства. Чтобы такая возможность была, надо купить у провайдера белый IP-адрес для шлюза, и настроить шлюз так, чтобы он все запросы на какой-то определенный порт перенаправлял на определенное устройство (это называется **статический** NAT).

Записи в таблице при динамическом NAT живут, пока по соединению идут пакеты. Если пакеты ходят редко, то запись быстро умирает. Это плохо, потому что ВК, например, предоставляет сервис push-уведомлений. Для этого серверу необходимо отправлять запросы на устройство. Значит надо, чтобы соединение было живо постоянно. Чтобы поддерживать соединение, приложение постоянно отправляет маленькие пакеты на сервер.

4.5. DNS-сервер

Чтобы отправить запрос, надо знать IP-адрес получателя. Но как его узнать? Для этого есть DNS-серверы (Domain Name System) с фиксированными, известными всем устройствам IP-адресами. DNS-сервер получает текстовый адрес сайта и возвращает его IP-адрес. DNS-сервер, в отличие от шлюза, не обязан лежать в одной локальной сети с отправителем запроса.

Прежде чем отправить запрос на vk.com, Алиса отправляет запрос на DNS-сервер (обычным способом), чтобы узнать IP-адрес, записывает этот адрес в запрос для ВК и отправляет.

Как работает DNS-сервер? Внутри него есть записи. Прежде всего, в нем записано, за какие **зоны** он отвечает. Зоны – это кусочки текстового адреса, разделенные точками (ru, spbau.ru, mit.spbau.ru, ...). В рамках каждой зоны сервер хранит записи нескольких типов:

- Запись типа A (Address) хранит текстовый адрес и соответствующий ему IP-адрес:

  site.ru A 194.85.23.13

- Запись типа CNAME (Canonical Name) говорит, что вместо одного адреса надо спрашивать про другой:

  www.site.ru CNAME site.ru

  Если есть записи ''X CNAME Y'' и ''Y CNAME X'', то запрос к Y или к X, зациклится.

- Запись типа MX (Mail Exchanger) позволяет поставить в соответствие домену почтовый сервер, чтобы можно было писать user@mit.spbau.ru вместо user@mail.ru:

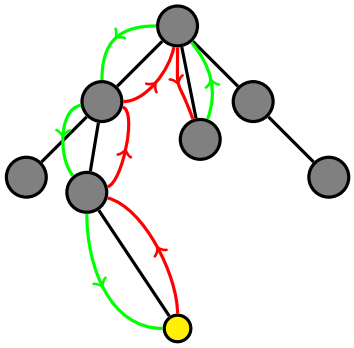
```
.           MX          mail.ru
students    MX          yandex.ru
```

- Запись типа NS (Authoritative Name Server) Возвращает IP-адрес сервера, ответственного за указанную зону. То есть если сервер отвечает за зону spbau.ru, но не отвечает за mit.spbau.ru, то он вернет адрес нижестоящего сервера:

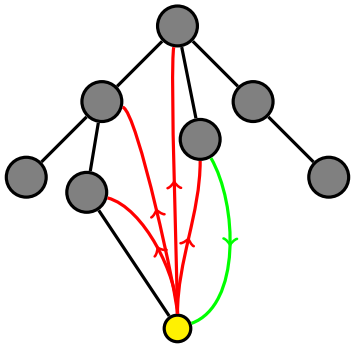
```
mit         NS          193.87.10.20
```

Есть DNS-серверы, которые ни за какую зону не отвечают. Они перенаправляют запросы вышестоящему серверу, но когда они получают ответ, они его запоминают.

Таким образом, DNS-серверы образуют иерархию. DNS-запросы бывают рекурсивные и нерекурсивные. При рекурсивном запросе мы спрашиваем что-то у сервера, он спрашивает у другого сервера, итд. При этом, когда мы получили ответ, все серверы в рекурсивной цепочке запомнили его, и следующие ответы будут происходить быстро. При нерекурсивном запросе мы спрашиваем у первого сервера, он возвращает нам адрес второго, мы спрашиваем у второго, итд.



Рекурсивный DNS-запрос



Нерекурсивный DNS-запрос

Если много компьютеров начнут в бесконечном цикле посылать запросы с несуществующими доменными именами, они все будут доходить до корневого сервера, и тот может не выдержать. Поэтому корневых DNS-серверов всего 13. Они расположены в разных частях света, чтобы запросам было удобнее до них добираться.

Допустим, Алиса хочет сделать свой сайт с именем alice.ru. Для начала надо купить белый IP-адрес, чтобы можно было подключаться снаружи. Далее надо DNS-серверу, ответственному за зону ru, сообщить, что alice.ru находится по нашему IP-адресу. Для этого надо добавить запись типа A или NS. В случае NS, надо купить еще один DNS-сервер, который будет отвечать за нашу зону. Чтобы добавить запись на DNS-сервер, надо обратиться к регистраторам. Они могут взимать за это плату.

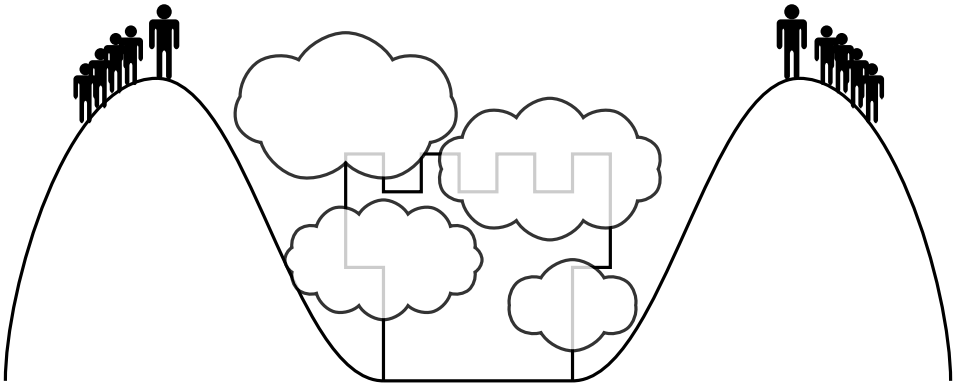
Как мы видим, по сети передается огромное количество служебных данных. Можно считать, что полезная информация занимает только половину всего трафика.

Зачем нужна такая сложная многоуровневая система передачи пакетов? В первую очередь для того, чтобы отделить пользовательское приложение (уровень Application) от физического слоя. Иначе пришлось бы скачивать много разных версий браузера для Ethernet, Wi-Fi, итд, потому что он должен был бы формировать разные пакеты нижнего уровня (Link). При этом нам нужна адресация, не зависящая от вида соединения. Для этого служит уровень Network. Теперь поговорим подробнее об уровне Transport.

4.6. Уровень Transport

На транспортном уровне есть два протокола – TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). При UDP, если отправленный пакет не дошел до получателя, отправитель об этом не узнает. При TCP отправителю постараются об этом сообщить. Но гарантии, что он узнает, нет.

4.6.1. Лирическое отступление. Задача о двух генералах



Условие. Есть замок. Он стоит в расщелине между холмами. Расщелина покрыта туманом. К замку с разных сторон подходят две армии. Одну возглавляет Алиса, а другую Боб. Они хотят одновременно напасть на замок, но из-за тумана они не видят друг друга. Генералы могут посылать друг к другу лазутчиков, но те по дороге могут погибнуть, или их могут перевербовать или подменить. Однако гарантируется, что часть лазутчиков добираются успешно. Каким образом Алисе и Бобу договориться о времени начала штурма?

Решения этой задачи нет. Гарантированно правильно договориться не получится. Но если Алиса и Боб знают некий общий секрет (например, число 30), то договориться получится. Посмотрим, как это работает на примере TCP.

Во время TCP-соединения по сети передается куча служебных пакетов, которые это соединение поддерживают. Алиса шлет Бобу пронумерованные пакеты пачками (окнами), Боб шлет Алисе подтверждения, о том, какой последний пакет он получил. Если Алиса не получила подтверждение от Боба, она отправляет окно еще раз. И так несколько раз, пока ей не надоест. После этого она считает, что соединение умерло и перестает отправлять.

Таким образом, при общении по TCP, есть три варианта:

- Пакет успешно дошел, и Алиса получила подтверждение;
- Пакет успешно дошел, но Алиса **не** получила подтверждение;
- Пакет **не** дошел, и Алиса **не** получила подтверждение.

Преимущество UDP в том, что он быстрее, так как у него нет накладных расходов, связанных с проверками доставки и повторными отправками. При плохом соединении, когда пакеты часто теряются, TCP работает крайне медленно.

Но зачем использовать соединение, при котором пакеты теряются? Например, есть вебкамера. Она снимает и отсылает в сеть кадры несколько раз в секунду. Здесь нам не жалко потерять несколько кадров, но важно, чтобы задержка была как можно меньше. Второй пример – торренты. Они создают кучу соединений с разными людьми и скачивают у них кусочки файлов. Если кусочек не скачался, попросим у другого человека.

Есть такая проблема. Приложение хочет отправить в сеть данные. Они могут быть как 2 байта, так и 10 гигабайтов. Передавать пакеты размером в несколько гигабайтов – плохая идея. Поэтому TCP И UDP, когда получают с верхнего уровня пакет, разбивают его на кусочки и передают их по отдельности. При этом часть кусочков теряется, а некоторые кусочки меняются местами (в UDP кусочки не пронумерованы). Чтобы избежать перемешивания, современные маршрутизаторы стараются внутри себя нумеровать все пакеты и собирать их в нужном порядке, но никто ничего не гарантирует.

4.7. Таблица маршрутизации

Маршрутизатор отличается от обычного компьютера, в первую очередь, тем, что у него много сетевых карт – своя карта для каждого сетевого разъема. У каждого разъема есть своя маска. Вопрос – что делать со шлюзом? У каждого маршрутизатора есть **таблица маршрутизации**. Она состоит из следующих столбиков.

IP	Mask	Gateway	Interface	Metric
192.168.5.0	255.255.255.0	192.168.5.1	eth0	12
194.68.0.0	255.255.0.0	194.168.2.2	eth1	10
0.0.0.0	255.255.255.255	1.1.1.1	eth2	8

Про первые три столбика мы уже всё знаем. Interface – это идентификатор сетевой карты, через которую будут отправляться пакеты (в нашем случае везде Ethernet). Наш маршрутизатор получил пакет, адресованный не ему. Он перебирает строчки в таблице и проверяет, подходит ли адрес получателя под адрес и маску в этой строчке. Если подходящих строчек несколько, выбирается та, у которой максимальная метрика. Таким образом, маршрутизатор может находиться сразу во многих сетях и быть связующим звеном между ними.

У обычного компьютера тоже есть таблица маршрутизации, но она состоит из одной строчки.

В таблице маршрутизации есть еще одна строчка:

127.0.0.1	255.0.0.0	–	loo01	7

Это **LoopBack Interface** (обратная петля). Дело в том, что у компьютера есть симулированный сетевой интерфейс, к которому подключен только этот компьютер. Причем он имеет все адреса 127.*.*.*. Это сделано, чтобы можно было удобно обмениваться данными между приложениями на одном компьютере.

4.8. Виды сетевых устройств

Мы уже знакомы с Wi-Fi роутерами. Они являются маршрутизаторами. Но маршрутизаторов бывает много разных. Маршрутизатор отличается от других устройств тем, что у него есть собственный IP-адрес, через который можно передавать данные.

В компьютерном классе все компьютеры соединены проводами через одну коробочку. Чаще всего она **не является** маршрутизатором. Она называется Switch (сетевой коммутатор). В базовом понимании, когда на свитч приходит пакет через один из разъемов, он его копирует и отправляет на все остальные разъемы. Такая разновидность свитча называется Hub (сетевой концентратор). Современные свитчи делают умнее. Они смотрят, на какой MAC-адрес надо отправить пакет, и отправляют на соответствующий порт. Если они не знают, на каком порте находится этот адрес, они отправляют пакет всем.

Свитч, который анализирует MAC-адреса, называется Layer 4 Switch, потому что анализирует информацию на четвертом уровне (Link). Бывают свитчи более высоких уровней, вплоть до первого (Application). Они уже являются маршрутизаторами. Чтобы делать NAT, необходимо работать на уровене 2. Wi-Fi-роутеры это умеют, но медленно. Поэтому их чаще всего называют 3+ или 3+2. Чем выше уровень, тем дороже стоит устройство, потому что надо быстро обрабатывать большое количество информации.

Свитчи первого уровня используют, например, чтобы анализировать трафик, вести статистику или запрещать доступ к определенным сайтам по доменному имени или даже к конкретным страницам. Почему плохо блокировать доступ по IP-адресу? Потому что на одном адресе может находиться несколько сайтов, и все они заблокируются.

Еще один пример использования свитчей первого уровня – Proxy. Допустим, пользователи в одной сети заходят на один и тот же сайт и скачивают какие-то картинки и элементы оформления страницы. Чтобы снизить трафик, нужен сервер, который бы хешировал все эти картинки и выдавал пользователям, а не скачивал каждый раз с сайта. Чтобы пользоваться прокси-сервером, надо отдельно настроить браузер. Чтобы каждому пользователю не приходилось настраивать свой браузер, устанавливают свитч первого уровня, который анализирует запросы и перенаправляет их на прокси-сервер.

4.9. Немного о мировом интернете

Алиса сидит в Санкт-Петербурге и отправляет запрос в гугл. Этот запрос пойдет к провайдеру Алисы, потом к вышестоящему провайдеру, итд. В какой-то момент запрос должен покинуть город. В Петербурге есть четыре точки, через которые запросы уходят в другие города.

- Спутниковая тарелка на здании ИТМО;
- Оптоволоконный кабель, проложенный под землей, вдоль железной дороги между Питером и Москвой;
- Оптоволоконный кабель, натянутый по столбам, идущий в Финляндию;
- Оптоволоконный кабель, идущий в Прибалтику.

Покинуть континент можно через спутниковые тарелки, через кабель, проложенный по дну Атлантического океана и другими путями. Заранее маршрут запроса не известен. В реальности таблицы маршрутизации постоянно перестраиваются и перераспределяют потоки запросов. Поэтому, например, если приложение отправило большой запрос, и он разделился на мелкие части, они могут пойти разными путями и прийти в случайном порядке.

Точки на границах государств контролируются с каждой стороны правительством данного государства.

Если в каких-то узких местах случаются аварии, вся сеть начинает перестраиваться. Это может занимать вплоть до часа. В это время у большого количества пользователей могут возникнуть проблемы и перебои с интернетом.

4.10. Динамическая маршрутизация

Мы поняли, что "облачко", через которое передаются все пакеты, состоит из кучи маршрутизаторов, сложным образом соединенных между собой. Хочется, чтобы в случае отказа некоторых узлов, не приходилось вручную перенастраивать все маршрутизаторы в сети. Для этого придумали динамические таблицы маршрутизации.

Формально, зачада состоит с следующем. Есть связный граф из маршрутизаторов, к некоторым из которых подключены компьютеры. Надо, чтобы при отправке пакета с одного компьютера, система сама строила путь, по которому этот пакет дойдет до получателя.

Рассмотрим одно из решений – протокол OSPF (Open Shortest Path First). Первая идея состоит в том, что каждый маршрутизатор должен знать весь граф. Предположим, что все устройства только что включились. Компьютеры начинают отправлять пакеты на свои шлюзы, благодаря этому шлюзы получают информацию (IP-адреса) о подключенных к ним компьютерах. Теперь маршрутизатор должен что-то понять про соединения с другими маршрутизаторами. Поэтому, при включении, маршрутизаторы шлют всем своим соседям информацию о себе, о своих компах и информацию, полученную от соседей. Когда процесс завершается, все маршрутизаторы знают всё про граф сети.

Дальше нужно выбрать одинаковую для всех политику передачи пакетов. Для это выбирается главный маршрутизатор. Выбор происходит по нескольким параметрам – количеству подключенных компов (чем больше тем лучше), скорости обработки запросов, уровню (см. уровни свитчей) и производителю. Главный маршрутизатор строит минимальное остовное дерево сети и рассылает его своим соседям, а те – всем остальным. Это не самая оптимальная политика, потому что иногда выгодно отправить пакет непосредственно по ребру графа, но в остовном дереве этого ребра может не быть. Однако в маленьких сетях на скорости это почти не сказывается.

Если какое-то ребро дерева выходит из строя, маршрутизаторы на его концах начинают сообщать об этом всем соседям. Когда эта информация доходит до главного маршрутизатора, он перестраивает дерево и передает всем новую версию. Если выходит из строя главный маршрутизатор, то выбирается новый. Процесс распространения всей этой информации по сети длится 2-3 секунды на сетях из ≈ 2000 устройств.

Некоторые маршрутизаторы также могут быть подключены ко внешнему миру. Если выходов во внешнюю сеть несколько, OSPF выбирает один, и все пакеты передаются через него.

Существуют и другие протоколы, решающие задачу динамической маршрутизации, например протокол RIP. OSPF хорошо подходит для небольших сетей. Для мирового же интернета он слишком неоптимальный.

4.11. Еще немного о мировом интернете

Большой интернет разбит на **автономные системы**. У каждой автономной системы есть id и диапазон IP-адресов. Задача АС состоит в том чтобы доставлять пакеты внутри своего диапазона любым доступным ей способом. Всего в мире 1024 автономные системы. За каждую систему отвечает какая-то компания или человек.

Автономные системы соединены между собой. На концах таких соединений находятся граничные серверы, которые общаются друг с другом по протоколу BGP. (Border Gateway Protocol). Идея этого протокола похожа на OSPF, но реализация отличается (например, в BGP нет главного сервера). Граничных серверов не очень много (≈ 20000), поэтому такая система работает стабильно. Если конфигурация сети поменялась, процесс распространения служебной информации длится 20-60 минут.

Каждая автономная система разбита на более мелкие подсистемы, которые устроены примерно так же. Подсистемы общаются по протоколу IGP (Interior gateway protocol).

То есть когда в большую автономную систему приходит пакет, есть два варианта. Если пакет надо передать другой АС, то он отправляется на соответствующий граничный сервер. Если же пакет надо передать на какой-то IP-адрес из диапазона этой АС, то пакет отправляется в подсистему, ответственную за этот адрес.

4.12. Протокол ICMP

ICMP (Internet Control Message Protocol) – это протокол, позволяющий сообщить отправителю, что его пакет не дошел до получателя. Из того узла, где пакет умер, посылается ICMP-пакет и, если этот пакет сам не умрет, отправитель узнает, что его пакет не доставлен.

Пусть мы отправили сообщение об ошибке, оно не дошло. Из-за этого нам отправили новое сообщение об ошибке, оно тоже не дошло, итд. Чтобы этого избежать, сообщения об ошибке отправляются только для обычных (не ICMP) пакетов.

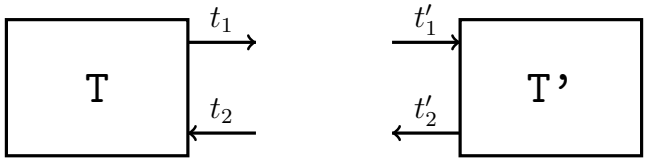
Когда пакет умирает? У каждого пакета есть TTL (Time To Live). Это число, из которого, при проходе через каждый маршрутизатор, вычитается единица. Когда TTL становится равен нулю, пакет признается мертвым. Обычно при отправке пакета, его TTL около 128. Это сделано для защиты от зацикливания, если в сети окажется петля.

Также с помощью TTL реализована утилита `traceroute`. Она выдает список всех маршрутизаторов от данного компьютера до заданного узла. Утилита отправляет обычный пакет с $\texttt{TTL} = 1$. Пакет доходит до первого маршрутизатора и умирает. Маршрутизатор отправляет ICMP-пакет, а в нем содержится IP-адрес этого маршрутизатора. Далее отправляется пакет с $\texttt{TTL} = 2$, итд. Есть маленькая вероятность, что очередной пакет пойдет по другому пути, чем предыдущие. Тогда результат может быть некорректным.

ICMP-пакеты бывают разных типов. Некоторые из них:

3. Адресат не доступен (пакет не доставлен). Далее идет описание причины.
8. `Echo request` (Эхо-запрос) – пакет, отправляемый командой `ping`.
0. `Echo response` (Эхо-ответ) – пакет, отправляемый в ответ на Эхо-запрос.

Еще один пример использования ICMP – синхронизация времени.



Хотим выставить на нашем компьютере такое же время, как на другом. Мы отправляем на второй компьютер пакет в момент t_1 и в момент t_2 получаем ответ. Второй компьютер получил наш пакет в момент t'_1 и отправил ответ в момент t'_2. Тогда можно считать, что в одну сторону пакет шел $\frac{t_2-t_1-(t'_2-t'_1)}{2}$ времени. В реальности это будет неточно, но мы этим пренебрегаем. В итоге получаем разницу времени на компьютерах: $t_1 + \frac{t_2-t_1-(t'_2-t'_1)}{2} - t'_1$.

4.13. Внутреннее устройство пакетов

Как мы знаем, когда устройство получает Ethernet-пакет, оно проверяет MAC-адрес получателя, разворачивает пакет и передает на уровень выше. На следующем уровне есть разные протоколы (а еще есть ARP, который на том же уровне). Надо понять, какому обработчику передать развернутый пакет. Для этого у Ethernet-пакетов есть еще одно поле – EtherType, занимающее 2 байта, которое говорит, какой протокол использовать для дальнейшей обработки. Дальше идут сами данные, и в конце – контрольная сумма. Таким образом, непосредственный размер передаваемых данных не превосходит 1500 байтов. Максимальная величина данных имеет название MTU (Maximum Transmission Unit). В разных сетях MTU бывает разным и зависит от качества сигнала, скорости передачи, итп.

MAC1	MAC2	EtherType	Data ($\leq$ MTU)	Check Sum

Допустим, на следующем уровне оказался протокол IP. Заголовок IP-пакета занимает 24 байта. Он содержит, помимо прочего, IP-адреса отправителя и получателя, идентификатор протокола (ICMP, TCP, UDP и проч.), TTL (1 байт) и длина содержимого.

...	Length	...	Fragment Offset	TTL	Protocol	...	IP1	IP2	...	Data

Поле, описывающее длину, занимает 2 байта. Это значит, что максимальная длина данных равна 65536 (2^{16}) байтов. Получается, что IP-пакет может полностью не поместиться в Ethernet-пакет. Поэтому его иногда приходится дробить. Этот процесс называется **фрагментация**. Поле Fragment Offset указывает, как текущий фрагмент смещен относительно всего пакета.

Допустим, приложение хочет отправить по сети данные размером 2 гигабайта. Оно создает Application-пакет и отдает уровню ниже, допустим, протоколу UDP. Тот понимает, что в нижестоящем протоколе IP максимальный размер пакета гораздо меньше. Поэтому он разбивает пакет на кусочки и отправляет. При передаче пакетов по UDP, они теряются и меняются местами. В результате получатель не сможет правильно восстановить пакет. Отсюда вывод – не надо посылать большие пакеты через UDP.

TCP, с точки зрения написания кода, сам занимается формированием пакетов и сам определяет их размер. Чтобы избежать фрагментации на уровне Ethernet, он формирует пакеты не больше MTU. Таким образом, MTU – самое узкое место. И оно задает размеры пакетов всех высших уровней.

Фрагментация всё же может произойти, если пакет по пути к получателю проходит через сеть с маленьким MTU. Минимально возможное значение MTU равно 576 байтов. Поэтому, чтобы гарантированно избежать фрагментацию, надо передавать пакеты, не превышающие 508 байтов чистых данных (68 байтов тратятся на заголовки).

Ограничение MTU в 1500 байтов действует в рамках Ethernet-протокола. Для других видов соединений эта величина может быть больше. Например, для LoopBack она может быть порядка нескольких гигабайтов.

4.14. Лирическое отступление-2. Как работает телеграм

Что делает сами знаете кто, когда хочет забанить какой-то IP-адрес? Они говорят провайдерам настроить их маршрутизаторы так, чтобы пакеты, идущие на этот адрес, выбрасывались. Так делать плохо, потому что на одном IP-адресе могут находиться несколько сайтов, и доступ прекратится ко всем из них. Нормальные провайдеры, использующие Switch Level 1, разворачивают пакет до верхнего уровня и смотрят, на какой сайт послан пакет.

Допустим, мы хотим написать мессенджер, поддерживающий большой поток клиентов. Для любого соединения нужен свой порт. Портов на одном компьютере максимум 2^{16}. Поэтому для обслуживания бóльшего числа клиентов, надо иметь много компьютеров. Чтобы не покупать собственные компьютеры, серверы можно запускать в облаке. Мы арендуем у какой-нибудь компании серверы, и наш шлюз, на который приходят все запросы, сообщает каждому пользователю, к какому серверу он должен обратиться. Поскольку компания обычно выделяет целую подсеть адресов, её всю блокируют, и другие серверы в этой подсети оказываются недоступны.

Здесь есть проблема, что достаточно забанить только распределяющий компьютер, тогда пользователи не смогут узнавать, к каким серверам им подключаться. Можно иметь несколько распределяющих серверов и тоже хранить их в облаке. Чтобы пользователи узнали об адресах распределяющих компов, телеграм посылает на пользовательские устройства push-уведомления. Чтобы забанить их, придется забанить все push-уведомления, поступающие на устройство.

4.14.1. Proxy

Proxy – это приложение уровня Application, которое получает пакеты и куда-то передает, предварительно выполнив какие-то действия. Например, закешировать картинку, чтобы не скачивать каждый раз с сайта заново.

Чтобы обойти блокировку, можно расположить proxy за границей и настроить его так, чтобы он получал пакет от пользователя, отправлял в телеграм, получал ответ и отправлял ответ пользователю.

Чтобы этим пользоваться, приложение должно иметь поддержку прокси.

4.14.2. VPN

VPN (Virtual Private Network) делает так, что компьютер начинает видеть новый сетевой интерфейс. Когда компьютер отправляет пакет на этот интерфейс, специальная программа его получает, передает по специальному протоколу на какой-то сервер, который как-то этот пакет обработает, преобразует в нормальный сетевой пакет и передаст.

Эта штука похожа на прокси, но она не требует поддержки приложением. Также VNP работает медленнее и расходует на устройстве много энергии, потому что пакеты разворачиваются и обрабатываются на более высоком уровне.

5. Сети в Java

В джаве можно работать только с пакетами уровня Application и Transport.

Для работы с сетью используется **сокет** – это низкоуровневый API, для пересылки байтов по сети. Сокеты в джаве поддерживают протоколы TCP и UDP.

5.1. UDP-Socket

При создании сокета, ОС выделяет порт и закрепляет за этим сокетом.

Рассмотрим пример отправки UDP-запросов (клиент).

```
try (DatagramSocket s = new DatagramSocket()) {
  DatagramPacket p = new DatagramPacket(buf, buf.length, remoteAddress);
  s.send(p);
}
```

Создаем сокет и пакет, который хотим передать. В качестве параметров конструктор DatagramPacket принимает массив байтов, длину массива и адрес получателя. Адрес включает в себя IP-адрес и порт получателя.

В нашем примере порт на нашем компьютере выбирается случайно. Если важно использовать конкретный порт, его номер можно передать в конструктор сокета.

Теперь рассмотрим пример сервера.

```
try (DatagramSocket s = new DatagramSocket(port)) {
  byte[] buf = new byte [1024];
  DatagramPacket p = new DatagramPacket(buf, buf.length);
  s.receive(p);
}
```

Что будет, если придет пакет размера больше buf.length байтов? Тогда первые buf.length байта считаются, а остальные останутся в сокете, и их можно будет прочитать при следующем вызове receive().

Далее у пакета можно спросить много разной информации – кто отправил, какая длина, итд. В конце надо из полученных байтов как-то восстановить данные, которые мы ожидали получить.

5.2. TCP-Socket

В отличие от UDP, в TCP мы не создаем пакет, а отправляем данные в сокет, и он всё формирует и отправляет за нас.

Рассмотрим процесс общения по TCP. Клиент хочет создать соединение с vk.com. Он посылает запрос на сервер, на 80 порт. Потом приходит второй клиент. TCP-соединение необходимо постоянно поддерживать, а значит нельзя со всеми клиентами общаться через один порт (иначе буфер, в который приходят запросы, будет переполняться). Поэтому, когда сервер получает запрос на 80 порт, он в ответ высылает номер другого порта, через который данный клиент должен продолжить общение с сервером.

В UDP такой проблемы нет, поэтому там все клиенты общаются через один порт.

Рассмотрим пример использования TCP сокета (клиент).

```
Socket socket = new Socket (''localhost'', 11111);

OutputStream os = socket.getOutputStream();
os.write(requestBytes);
os.flush();

InputStream is = socket.getInputStream();
is.read(responseBytes);
```

Создаем сокет, передаем ему адрес и порт, с которыми соединяемся, запрашиваем у сокета `OutputStream` и `InputStream`. Метод `flush()` сбрасывает всё, что накопилось в буфере и пытается отправить это по сети.

Варианты конструктора для сокета:

- `Socket(String host, int port) throws UnknownHostException, IOException` – получает имя хоста и номер порта.
- `Socket(InetAddress host, int port) throws IOException` – то же самое, только вместо имени хоста – адрес.
- `Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException` – дополнительно получает локальный адрес и порт. Локальный адрес бывает надо передать, если у компьютера несколько IP-адресов, так как у него есть разные сетевые интерфейсы. Такой сокет может слушать, например, только `Wi-fi`.
- `Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException`.
- `Socket()` – неподключенный сокет. Чтобы его подключить, надо вызвать метод `connect()`.

Поскольку мы не знаем, сколько байтов нам хотят прислать, есть соглашение первым делом отправлять количество данных, а затем сами данные.

Рассмотрим пример серверного TCP-сокета.

```
ServerSocket server = new ServerSocket(11111);
Socket socket = server.accept();

InputStream is = socket.getInputStream();
is.read(requestBytes);

OutputStream os = socket.getOutputStream();
os.write(responseBytes);
os.flush();
```

`ServerSocket` – это как раз тот сокет, который не общается с клиентом, а отправляет его на другой сокет. Ему в качестве параметра передается порт, на который приходят запросы соединения. Далее блокирующий метод `accept()` ждет, пока не придет клиент, а потом создает и возвращает сокет для общения.

Варианты конструктора серверного сокета:

- `ServerSocket(int port) throws IOException` – слушает заданный порт.
- `ServerSocket(int port, int backlog) throws IOException` – получает максимальное количество клиентов в очереди на соединение.
- `ServerSocket(int port, int backlog, InetAddress address) throws IOException` – слушает только запросы, поступившие на заданный адрес.
- `ServerSocket() throws IOException` – не привязанный сокет, привязывается методом `bind()`.

5.3. Сервер с блокирующей архитектурой

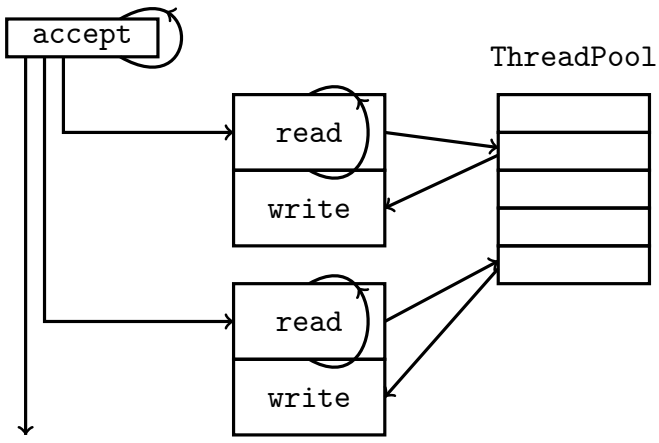
Код из предыдущего примера сможет обработать только одного клиента, потому что `accept()` вызывается только один раз. Чтобы обрабатывать много клиентов, можно, например, выполнять этот код в цикле, но есть проблема. Мы работаем в одном потоке, поэтому сможем общаться только с одним клиентом одновременно. Еще одна проблема такого подхода в том, что создается соединение ради одноразового обмена пакетами.

Другой вариант – принимать соединение и создавать отдельный поток для общения с клиентом. Проблема в том, что клиентов может быть очень много, а много потоков это плохо. Хочется их переиспользовать.

Для этого заведем тредпул. Теперь надо понять, какие задачи ему давать. Задача "пообщаться с клиентом" плохая, потому что обычно общение подразумевается длительное, поэтому одновременно будут обрабатываться столько клиентов, сколько потоков в пуле, а остальные будут ждать в очереди. Хорошая задача для тредпула – обработка одного запроса (например, клиент просит отсортировать массив).

Помимо выполнения запроса, надо отправить клиенту ответ. Тут есть два варианта. Мы можем отправить ответ сами, а можем куда-то сложить результат, и кто-то другой его отправит.

Итого, есть основной поток, который принимает соединения и создает для каждого новый поток. Внутри этих потоков происходит чтение запросов клиентов и отправка задач в тредпул. Тредпул выполняет задачи и либо отправляет результат клиенту, либо возвращает его в поток, ответственный за этого клиента, и тот уже отправляет. На самом деле отправлять ответ в тредпуле плохо, потому что метод `write()` – блокирующий. Он ждет, пока сеть не освободится для записи.



При таком подходе всё еще создается очень много потоков, но из них реально работают только потоки тредпула и потоки, которые в данный момент читают или пишут данные, общаясь с клиентом. Остальные потоки находятся в заблокированном состоянии и на работу других не влияют. Такая архитектура называется **блокирующая**. Блокирующая – потому что методы `accept()`, `read()` и `write()` блокируют поток, пока не дождутся. Она хорошо работает, если клиентов много, но одновременно общаются мало. В других ситуациях могут пригодиться другие архитектуры.

Как мы помним, в многопоточных программах надо аккуратно работать с разделяемыми ресурсами. В нашем случае разделяемый ресурс – это сеть. Посмотрим подробнее, что надо делать, а что не надо.

- Сокеты, привязанные к разным портам, блокировать не надо, потому что они все работают независимо.
- У сокета есть `InputStream` и `OutputStream`. Если один поток читает, а другой пишет, блокировку брать тоже не надо.
- Брать блокировку надо, например, если разные потоки пытаются одновременно из одного сокета читать или одновременно в него писать. Тут как раз пригождается `SingleThreadExecutor`. Заводим такой для каждого клиента, и тредпул просит его выводить результаты выполненных задач.

5.4. Пакет `java.nio`

`NIO` (New Input/Output) – альтернативная библиотека ввода/вывода.

5.4.1. Каналы

Каналы пришли на замену `InputStream` и `OutputStream` и имеют ряд существенных отличий от предшественников:

- Канал двунапрваленный – из него можно читать и одновременно в него писать.
- Каналы позволяют асинхронные чтение и запись.
- Канал пишет данные в буфер и читает из буфера.

Основные реализации каналов в `java.nio`:

- `FileChannel` – работает с файлами.
- `DatagramChannel` – передает данные по `UDP`.
- `SocketChannel` – передает данные по `TCP`.
- `ServerSocketChannel` – принимает входящие `TCP` соединения. На каждое новое соединение создается `SocketChannel` для дальнейшего общения с клиентом.

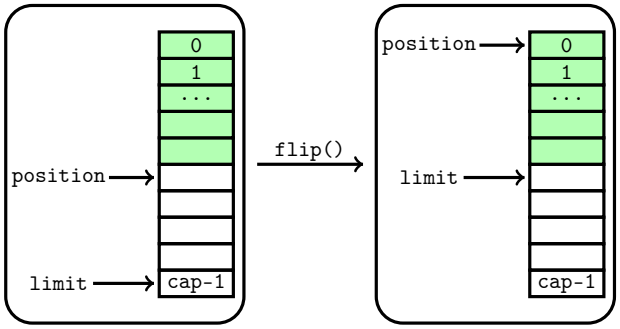
5.4.2. Буферы

Буфер – это блок данных, в который можно писать данные и потом считывать. Буферы бывают байтовые (`ByteBuffer`), символьные (`CharBuffer`), вещественные (`DoubleBuffer`), итд.

По смыслу, буфер – это массив с

- заданной вместимостью (`capacity`)
- отмеченной границей, сколько максимально данных можно читать/писать (`limit`)
- отмеченной текущей позицией чтения/записи (`position`)

Мы записали какие-то данные и находимся в состоянии записи (левая картинка). Хотим перейти в состояние чтения (правая картинка). Для этого вызываем метод `flip()`, который переводит `position` в нулевую позицию, а `limit` переводит в максимальное значение, которого достигал `position`.



Buffer – режим записи Buffer – режим чтения

Чтобы перевести буфер обратно из состояния чтения в состояние записи, используется `clear()` или `compact()`. Метод `clear()` переводит `limit` в `capacity`, а `position` в ноль, то есть непрочитанные данные забываются. Метод `compact()` копирует непрочитанные данные в начало буфера и устанавливает `position` в первую свободную ячейку.

В чем преимущество буфера над массивом? Можно передать буфер каналу, чтобы тот что-то в него записал, а потом снова передать каналу, и новые данные допишутся в этот же буфер.

Буфер можно получить методом `allocate()`.

```
ByteBuffer buf1 = ByteBuffer.allocate(48); // буфер на 48 байтов
IntBuffer buf2 = IntBuffer.allocate(1024); // буфер на 1024 инта
```

Обратите внимание, что здесь, как и у всех классов `nio`, объекты создаются не конструкторами, а статическими методами.

Записать данные в буфер можно вручную методом `put()` или через канал.

```
buf.put(127); // запись в буфер вручную
int bytesRead = channel.read(buffer); // запись в буфер из канала
```

Метод `read()` возвращает количество записанных в буфер байтов.

Прочитать данные из буфера можно вручную методом `get()` или через канал.

```
byte value = buf.get(); // чтение из буфера вручную
int bytesWritten = channel.write(buffer); // чтение из буфера в канал
```

Рассмотрим пример.

```
RandomAccessFile file = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel channel = file.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = channel.read(buf);

while (bytesRead != -1) {
  System.out.println("Read " + bytesRead);
  buf.flip();
  while(buf.hasRemaining()) {
    System.out.print((char) buf.get());
  }
  buf.clear();
  bytesRead = channel.read(buf);
}
file.close();
```

Открываем файл на чтение и запись, создаем канал и буфер на 48 байтов. Читаем из файла байты и пишем их в буфер. Если что-то еще прочиталось, выводим количество, переводим буфер в режим чтения, читаем по одному байту из буфера всё содержимое, выводим на экран, переводим буфер в режим записи и снова читаем из файла.

Если вместо `clear()` вызвать `flip`, может произойти что-то плохое. Если при записи `position` не дошел до конца, то `limit` установится не в конец, и при дальнейшем чтении доступный размер буфера будет меньше.

Можно пометить какую-то позицию в буфере с помощью метода `mark()`, а потом вернуться в неё, вызвав метод `reset()`. Например, если надо через буфер передавать пакеты с одинаковым заголовком, можно его один раз записать, а потом перед каждой записью переходить в позицию после заголовка и дописывать новый пакет.

Методу `read()` можно передать массив буферов. Тогда данные будут записываться в эти буферы последовательно – сначала в первый, потом во второй, итд. Это называется `Scattering`. Методу `write()` тоже можно передать массив буферов. Тогда данные будут читаться сначала из первого, потом из второго, итд. Это называется `Gathering`.

Это полезно при работе с пакетами. Если пакеты имеют заголовок известной фиксированной длины, имеет смысл при получении и отправке передавать каналу два буфера – с заголовком и с данными.

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);

// заполняем заголовок и тело пакета

ByteBuffer[] bufferArray = { header, body };
channel.write(bufferArray);

// пример с чтением выглядит аналогично
```

5.4.3. Селекторы

Пусть есть несколько каналов. Селектор может отвечать на вопрос, какие каналы в данный момент готовы к каким-либо действиям (читать, писать, ...).

Селекторы работают только с каналами в неблокирующем режиме. В этом режиме операции `accept()`, `read()` и `write()` не ждут, а завершаются очень быстро. Что это значит? Когда пакеты приходят на компьютер, они разворачиваются и складываются в специальный буфер. Грубо говоря, неблокирующая операция `read()` заглядывает в буфер. Если там что-то есть, она это прочитывает (возможно даже не полностью), а иначе возвращает ноль. Блокирующий `read()` будет сидеть и ждать, пока придет столько байтов, сколько его попросили прочитать.

Почему селекторы работают с каналами только в неблокирующем режиме? Во-первых, потому что селектор постоянно взаимодействует с каналами. Если бы они блокировались, всё бы зависало. Во-вторых, идея селектора в том, чтобы в одном потоке взаимодействовать с кучей клиентов сразу. Если этот поток блокируется первой же операцией, толку в нем мало.

Создание и регистрация селектора:

```
Selector selector = Selector.open(); // создаем селектор
channel.configureBlocking(false); // переводим канал в неблокирующий режим
SelectionKey key = channel.register(selector, SelectionKey.OP_READ); // регистрируем канал
                                                                     // в селекторе
```

Вторым аргументом метод `register()` принимает набор ключей, задающих, какие изменения в канале должен слушать селектор:

- `SelectionKey.OP_CONNECT`
- `SelectionKey.OP_ACCEPT`
- `SelectionKey.OP_READ`
- `SelectionKey.OP_WRITE`

Ключи можно комбинировать с помощью оператора побитового "или".

Метод `register()` возвращает объект класса `SelectionKey`. Он содержит следующую информацию:

- набор ключей, на которые подписан селектор (interest set):

```
// получаем interest set
int interestSet = selectionKey.interestedOps();
// подписаны ли мы на чтение
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
// подписаны ли мы на запись
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

- набор действий, к которым готов канал (ready set):

```
// первый способ
int readySet = selectionKey.readyOps();
boolean isReadyToRead = readySet & SelectionKey.OP_READ;

// второй, аналогичный способ
selectionKey.isReadable();
```

- ссылки на канал и селектор:

```
Channel channel = selectionKey.channel();
Selector selector = selectionKey.selector();
```

- прикрепленный объект:

```
// прикрепляем так
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
key.attach(myObject);

// или так
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, myObject);

// получаем так
Object attachedObject = key.attachment();
```

После того, как мы зарегистрировали каналы в селекторе, можно использовать один из методов `select()`. Эти методы возвращают количество каналов, которые стали готовы к интересующим нас действиям с момента предыдущего вызова. То есть если мы однажды вызвали `select()` и получили 1, и после этого еще один канал стал готов, то следующий вызов снова вернет нам 1, не зависимо от того, обработали мы первый канал или нет.

Метод `select()` бывает:

- `int select()` – блокирующий метод. Ждет, когда хотя бы один канал станет готов.
- `int select(long timeout)` – ждет не дольше заданного времени.
- `int selectNow()` – выдает результат сразу, возможно пустой.

Также у селектора можно вызвать метод `selectedKeys()` и получить `Set<SelectionKey>`, содержащий информацию о каналах, готовых к интересующим нас действиям. Из полученных `SelectionKey` можно получать сами каналы и прикрепленные объекты. Например, можно было прикрепить к каналу абстракцию, связанную с клиентом и теперь как-то с ней работать.

Пример. Получаем готовые к чему-либо потоки, перебираем их итератором, проверяем, к чему готов каждый из них, и удаляем `SelectionKey` из множества. Удалять ключи надо, потому что каждый следующий вызов `selectedKeys()` добавляет новые ключи к уже имеющимся, поэтому информация о старых ключах может быть уже неактуальной.

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while (keyIterator.hasNext()) {
  SelectionKey key = keyIterator.next();
  if (key.isAcceptable()) {
    // ServerSocketChannel принял входящее соединение
  } else if (key.isConnectable()) {
    // установлено соединение с удаленным сервером
  } else if (key.isReadable()) {
    // канал готов к чтению
  }  else if (key.isWritable()) {
    // канал готов к записи
  }
  keyIterator.remove();
}
```

5.5. SocketChannel

`SocketChannel` – это канал для работы с `TCP` соединениями. Как и обычный `Socket`, его можно создать вручную и привязать к удаленному серверу либо получить на сервере из метода `accept()`.

Поскольку в `nio` конструкторов нет, вручную канал создается статическим методом `open()`.

```
SocketChannel channel = SocketChannel.open();
channel.connect(new InetSocketAddress("http://aptu.ru", 80));
```

Теперь мы можем читать и писать через буферы. Пример записи.

```
String newData = "New String to write to file..." + System.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();

while (buf.hasRemaining()) {
  channel.write(buf);
}
```

После использования канал, как и сокет и файл, надо закрыть методом `close()`.

В неблокирующем режиме метод `connect()` завершается быстро, даже если еще не успел подключиться. Чтобы узнать, когда канал уже подключился, есть метод `finishConnect()`. Выглядит это так.

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.connect(new InetSocketAddress("vk.com", 80));

while (!socket.finishConnect()) {
  // ждем или делаем еще что-то
}
```

Методы `read()` и `write()` в неблокирующем режиме могут прочитать/записать данные не до конца и завершиться. Поэтому их надо вызывать в цикле, пока они не прочитают/запишут столько, сколько нужно.

5.5.1. ServerSocketChannel

`ServerSocketChannel` принимает входящие соединения и создает им каналы для общения.

Пример с блокирующим режимом.

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.socket().bind(new InetSocketAddress(9999));

while (true) {
  SocketChannel socketChannel = serverSocketChannel.accept();
  // общаемся с клиентом
}

serverSocketChannel.close();
```

Пример с неблокирующим режимом. Здесь `accept()` надо вызывать в цикле.

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);

while (true) {
  SocketChannel socketChannel = serverSocketChannel.accept();
  if (socketChannel != null) {
    //обрабатываем клиента
  }
}
```

Так писать смысла нет, потому что мы, вместо блокировки, активно ожидаем. Неблокирующий режим полезно использовать вместе с селекторами.

5.5.2. DatagramChannel

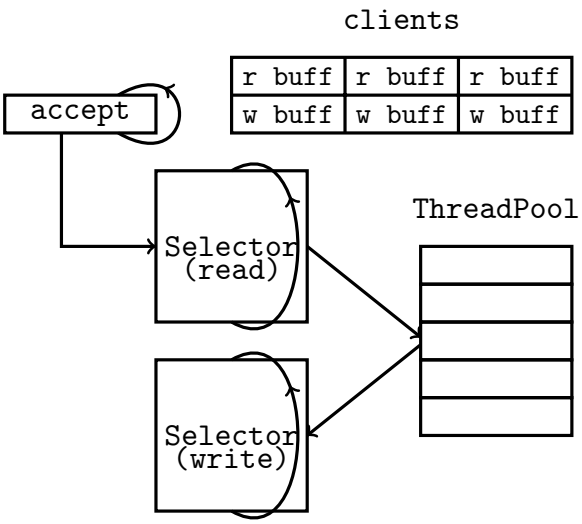
DatagramChannel – это канал для работы с UDP соединениями.

Пример. Обратите внимание, что чтобы привязать канал к порту, надо вызвать метод socket(), хотя формально у UDP никаких сокетов нет.

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();

channel.receive(buf);
....
String newData = ''New String to write to file...'' + System.currentTimeMillis();
buf.clear();
buf.put(newData.getBytes());
buf.flip();

int bytesSent = channel.send(buf, new InetSocketAddress(''aptu.ru'', 80));
```

Неблокирующий режим для DatagramChannel есть, но он бесполезен, так как, во-первых, пакеты могут теряться, а во-вторых, все клиенты шлют пакеты на один порт, поэтому их все можно обрабатывать в одном потоке.

5.6. Сервер с неблокирующей архитектурой



Первое что нужно – accept(). Для этого обычно в отдельном потоке запускают блокирущий ServerSocketChannel. Для каждого нового подключения создается SocketChannel и отправляется в селектор, крутящийся в другом потоке и следящий за чтением. Поскольку в неблокирующем режиме мы можем не прочитать всё сразу, для каждого канала хранятся еще буферы чтения и записи. Каждый раз, когда какой-то поток готов к чтению, данные из него читаются и дописываются в соответствующий буфер чтения. Когда в каком-то буфере накапливается полностью сообщение, оно отправляется на обработку в тредпул. После этого результат записывается в буфер записи, канал соответствующего клиента регистрируется в селектор для записи, и ответ постепенно отправляется. Когда ответ полностью отправлен, канал удаляется из селектора. Получилось $|\texttt{ThreadPool}| + 3$ потока.

Здесь можно сэкономить поток или даже два. Во-первых, можно селекторы объединить в один поток и вызывать их в цикле друг за другом. Во-вторых, можно ServerSocketChannel сделать неблокирующим и поместить в селектор для чтения. В этом случае тредпулу можно будет выделить больше потоков, чтобы они эффективно распараллеливались на всех имеющихся ядрах.

По этой ссылке можно почитать чуть больше про nio и неблокирующую архитектуру.

5.7. Сравнение архитектур

Обработка данных в обеих архитектурах происходит в тредпуле. Разница в том, что в неблокирующей для общения с клиентами используются 2 потока с селекторами, а не 2n, как в блокирующей.

Узкое место неблокирующей архитектуры – селекторы. Когда есть много клиентов, которые часто шлют запросы, селектор, работающий в одном потоке, работает крайне медленно. Для блокирующей архитектуры эта ситуация тоже плохая, потому что создается куча потоков и происходит частое переключение контекста. Это решается увеличением ядер и процессоров.

Итого, неблокирующая архитектура выигрывает, когда клиентов много, но они не очень активно пишут. Если клиентов мало, но пишут они часто, блокирующая архитектура тормозит из-за частого переключения контекста, а неблокирующая – когда как, зависит от пропускной способности селектора.

Пример, когда блокирующая лучше – если запросы от клиентов большие. Блокирующий `read()` их будет быстро считывать, сразу отправлять на обработку и засыпать. А неблокирующий будет долго крутиться в цикле, пока не считает всё.

На практике чаще всего выбирают блокирующую архитектуру, потому что её проще реализовывать, и она лучше масштабируется, то есть её можно увеличить, купив еще несколько процессоров.

5.8. Асинхронные каналы

Асинхронные каналы позволяют запустить чтение/запись в отдельном потоке и вернуть `Future`, в котором потом появится результат, или по завершении вызвать некоторую функцию (`callback`), в которую результат передастся и сразу обработается. При этом сам метод `read()`/`write()` выполнится мгновенно. Если делать через `Future`, операцию можно будет отменить, вызвав у `Future` метод `cancel()`.

`Callback` передается методу `read()`/`write()` в виде объекта, реализующего интерфейс `CompletionHandler<V, A>`. Если делать таким образом, то операцию отменить нельзя. У этого интерфейса есть два метода – `completed(v, a)` и `failed(exception, a)`.

`CompletionHandler<V, A>` параметризуется двумя типами. Тип `V` – это результат. Он попадает только в метод `completed()`. В метод `failed()` вместо этого передается исключение. Тип `A` – это тип прикрепляемого объекта, который чаще всего называют **контекстом**.

Пример.

```
socketChannel.read(
  buffer, // буфер, куда читать
  context, // контекст
  new CompletionHandler<Integer, Context>() {
    public void completed(Integer result, Context context) { // если успех
      if (context.process(result)) { // обрабатываем то что считали
        socketChannel.read(buffer, context, this); // читаем еще раз с тем же обработчиком
      }
    }
    public void failed(Throwable e, Context context) { // если неуспех
      context.error(); // обрабатываем ошибку контекстом
      e.printStackTrace();
    }
  }
);
```

По сути, такая асинхронная конструкция заменяет бесконечный цикл, который был в примере блокирующей архитектуры. Отличие в том, что все колбэки выполняются в отдельной группе потоков. Таким образом мы экономим потоки, потому что не создаем их для каждого клиента. При этом в колбэках не стоит делать долгие вычисления, потому что вся эта группа потоков будет тормозить.

Интерфейс `AsynchronousByteChannel` – наследник интерфейса `AsynchronousChannel` – читает/пишет в буфер. В качестве результата выдает `Integer` – количество прочитанных/записанных байтов.

Класс `AsynchronousSocketChannel` реализует этот интерфейс.

```
// создаем асинхронный сетевой канал
AsynchronousSocketChannel socketChannel = AsynchronousSocketChannel.open();
// привязываем к локальному адресу
socketChannel.bind(localAddress);
// подключаемся к удаленному адресу
socketChannel.connect(remoteAddress);
// или
socketChannel.connect(remoteAddress, context, handler);
// читаем/пишем
. . .
// прерываем все запущенные операции чтения
socketChannel.shutdownInput()
// прерываем все запущенные операции записи
socketChannel.shutdownOutput()
```

Класс `AsynchronousServerSocketChannel` принимает входящие подключения

```
// создаем
AsynchronousServerSocketChannel ssc = AsynchronousServerSocketChannel.open();
// привязываем к локальному адресу
ssc.bind(localAddress);
// устанавливаем соединение и получаем AsynchronousSocketChannel
ssc.accept();
// или
ssc.accept(context, handler);
```

Класс `AsynchronousChannelGroup` – это как раз группа потоков, в которой могут выполняться колбэки асинхронных операций. По умолчанию они все исполняются в одной группе, размер которой недокументирован. Обычно размер либо 1, либо количество ядер - 1.

Такую группу можно создать вручную, но так делают редко.

```
// создаем
channelGroup = AsynchronousChannelGroup.withThreadPool(executor);
// или
channelGroup = AsynchronousChannelGroup.withCachedThreadPool(executor, minThreads);
// или
channelGroup = AsynchronousChannelGroup.withFixedThreadPool(threadsNumber, threadFactory);
// создаем канал с привязкой к нашей группе
AsynchronousSocketChannel socketChannel = AsynchronousSocketChannel.open(channelGroup);
// завершаем работу
channelGroup.shutdown();
// завершаем работу и разрываем соединения
channelGroup.shutdownNow();
// дожидаемся завершения
channelGroup.awaitTermination(timeout, units);
```