Transmission Control Protocol (TCP, протокол управления передачей) — один из основных протоколов передачи данных интернета. Предназначен для управления передачей данных интернета. Пакеты в TCP называются сегментами.

В стеке протоколов TCP/IP выполняет функции **транспортного уровня** модели OSI.

Механизм TCP предоставляет поток данных с **предварительной установкой соединения**, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым (в отличие от UDP) целостность передаваемых данных и уведомление отправителя о результатах передачи.

UDP (англ. User Datagram Protocol — протокол пользовательских датаграмм) — один из ключевых элементов набора сетевых протоколов для Интернета. С UDP компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по IP-**сети без необходимости предварительного сообщения для установки специальных каналов передачи** или путей данных.

UDP использует простую модель передачи, без явных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Датаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа, но гарантируется, что если они придут, то в целостном состоянии. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют UDP, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени

Разница

TCP — ориентированный на соединение протокол, что означает необходимость «рукопожатия» для установки соединения между двумя хостами. Как только соединение установлено, пользователи могут отправлять данные в обоих направлениях.

- Надёжность — TCP управляет подтверждением, повторной передачей и тайм-аутом сообщений. Производятся многочисленные попытки доставить сообщение. Если оно потеряется на пути, сервер

вновь запросит потерянную часть. В TCP нет ни пропавших данных, ни (в случае многочисленных тайм-аутов) разорванных соединений.

- Упорядоченность — если два сообщения последовательно отправлены, первое сообщение достигнет приложения-получателя первым. Если участки данных прибывают в неверном порядке, TCP отправляет неупорядоченные данные в буфер до тех пор, пока все данные не могут быть упорядочены и переданы приложению.
- Тяжеловесность — TCP необходимо три пакета для установки сокет-соединения перед тем, как отправить данные. TCP следит за надёжностью и перегрузками.
- Потоковость — данные читаются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

UDP — более простой, основанный на сообщениях протокол без установления соединения. Протоколы такого типа не устанавливают выделенного соединения между двумя хостами.

- Ненадёжный — когда сообщение посылается, неизвестно, достигнет ли оно своего назначения — оно может потеряться по пути. Нет таких понятий, как подтверждение, повторная передача, тайм-аут.
- Неупорядоченность — если два сообщения отправлены одному получателю, то порядок их достижения цели не может быть предугадан.
- Легковесность — никакого упорядочивания сообщений, никакого отслеживания соединений и т. д. Это небольшой транспортный уровень, разработанный на IP.
- Датаграммы — пакеты посылаются по отдельности и проверяются на целостность только если они прибыли. Пакеты имеют определенные границы, которые соблюдаются после получения, то есть операция чтения на сокете-получателе выдаст сообщение целиком, каким оно было изначально послано.
- Нет контроля перегрузок — UDP сам по себе не избегает перегрузок. Для приложений с большой пропускной способностью возможно вызвать коллапс перегрузок, если только они не реализуют меры контроля на прикладном уровне.

Internet Protocol (IP, досл. «межсетевой протокол») — маршрутизируемый протокол сетевого уровня стека TCP/IP. Именно IP стал тем протоколом, который объединил отдельные компьютерные сети во всемирную сеть Интернет. Неотъемлемой частью протокола является адресация сети

TCP/IP — это модель сетевой передачи данных, которая описывает, как цифровые данные передаются от отправителя к получателю через четыре уровня, каждый из которых использует протокол передачи данных. Эта модель состоит из стека протоколов передачи данных, на которых основывается Интернет. Название происходит от двух основных протоколов семейства - Transmission Control Protocol (TCP) и Internet Protocol (IP), которые были первоначально разработаны и описаны в этом стандарте.

Набор интернет-протоколов описывает, как данные должны быть упакованы, обработаны, переданы и приняты. Эти протоколы организованы в четыре уровня абстракции, которые классифицируют связанные протоколы по объёму задействованных сетей. Уровень связи используется для данных в пределах одной сети, интернет-уровень для взаимодействия между независимыми сетями, транспортный уровень для связи между хостами, а прикладной уровень для обмена данными между приложениями.

Стек протоколов TCP/IP включает в себя **четыре уровня**:

- Прикладной уровень (Application Layer)
    - На прикладном уровне (Application layer) работает большинство сетевых приложений.
    - Эти программы имеют свои собственные протоколы обмена информацией, например, интернет браузер для протокола HTTP, ftp-клиент для протокола FTP (передача файлов), почтовая программа для протокола SMTP (электронная почта), SSH (безопасное соединение с удалённой машиной), DNS (преобразование символьных имён в IP-адреса) и многие другие.
- Транспортный уровень (Transport Layer)
    - Протоколы транспортного уровня (Transport layer) могут решать проблему негарантированной доставки сообщений

(«дошло ли сообщение до адресата?»), а также гарантировать правильную последовательность прихода данных. В стеке TCP/IP транспортные протоколы определяют, для какого именно приложения предназначены эти данные.

- Межсетевой уровень (Сетевой уровень) (Internet Layer)
  - Межсетевой уровень (Network layer) изначально разработан для передачи данных из одной сети в другую. На этом уровне работают маршрутизаторы, которые перенаправляют пакеты в нужную сеть путём расчёта адреса сети по маске сети.
- Канальный уровень (Network Access Layer)
  - Канальный уровень (англ. Link layer) описывает способ кодирования данных для передачи пакета данных на физическом уровне (то есть специальные последовательности бит, определяющих начало и конец пакета данных, а также обеспечивающие помехоустойчивость). Ethernet, например, в полях заголовка пакета содержит указание того, какой машине или машинам в сети предназначен этот пакет.
  - Примеры протоколов канального уровня — Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS.

Stream API https://highload.today/java-stream-api/#1

Инструмент языка Java, который позволяет использовать функциональный стиль при работе с разными структурами данных.

Для начала стриму нужен **источник**, из которого он будет получать объекты. Чаще всего это коллекции, но не всегда. Например, можно взять в качестве источника генератор, у которого заданы правила создания объектов.

Данные в стриме обрабатываются на промежуточных операциях. Например: мы можем отфильтровать данные, пропустить несколько элементов, ограничить выборку, выполнить сортировку. Затем выполняется терминальная операция. Она поглощает данные и выдает результат.

Методы Stream API не изменяют исходные коллекции, уменьшая количество побочных эффектов.

Конвейерных методов в стриме может быть много. Терминальный метод — только один. После его выполнения стрим завершается.

Конвейерные

Метод	Что сделает	Использование
`filter`	отработает как фильтр, вернет значения, которые подходят под заданное условие	`collection.stream().filter(«e22»::equals).count()`
`sorted`	отсортирует элементы в естественном порядке; можно использовать `Comparator`	`collection.stream().sorted().collect(Collectors.toList())`
`limit`	лимитирует вывод по тому, количеству, которое вы укажете	`collection.stream().limit(10).collect(Collectors.toList())`
`skip`	пропустит указанное вами количество элементов	`collection.stream().skip(3).findFirst().orElse("4")`
`distinct`	найдет и уберет элементы, которые повторяются; вернет элементы без повторов	`collection.stream().distinct().collect(Collectors.toList())`
`peek`	выполнить действие над каждым элементом элементов, вернет стрим с исходными элементами	`collection.stream().map(String::toLowerCase).peek((e) -> System.out.print("," + e)). collect(Collectors.toList())`
`map`	выполнит действия над каждым элементом; вернет элементы с результатами функций	`Stream.of("3", "4", "5") .map(Integer::parseInt) .map(x -> x + 10) .forEach(System.out::println);`
`mapToInt`, `mapToDouble`, `mapToLong`	Сработает как `map`, только вернет числовой `stream`	`collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()`
`flatMap`, `flatMapToInt`, `flatMapToDouble`, `flatMapToLong`	сработает как `map`, но преобразует один элемент в ноль, один или множество других	`collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::`

Терминальные

Метод	Что сделает	Использование
`findFirst`	вернет элемент, соответствующий условию, который стоит первым	`collection.stream().findFirst().orElse("10")`
`findAny`	вернет любой элемент, соответствующий условию	`collection.stream().findAny().orElse("10")`
`collect`	соберет результаты обработки в коллекции и не только	`collection.stream().filter((s) -> s.contains("10")).collect(Collectors.toList())`
`count`	посчитает и выведет, сколько элементов, соответствующих условию	`collection.stream().filter("f5"::equals).count()`
`anyMatch`	`True`, когда хоть один элемент соответствует условиям	`collection.stream().anyMatch("f5"::equals)`
`noneMatch`	`True`, когда ни один элемент не соответствует условиям	`collection.stream().noneMatch("b6"::equals)`
`allMatch`	`True`, когда все элементы соответствуют условиям	`collection.stream().allMatch((s) -> s.contains("8"))`
`min`	найдет самый маленький элемент, используя переданный сравнитель	`collection.stream().min(String::compareTo).get()`
`max`	найдет самый большой элемент, используя переданный сравнитель	`collection.stream().max(String::compareTo).get()`

`forEach`	применит функцию ко всем элементам, но порядок выполнения гарантировать не может	`set.stream().forEach((p) -> p.append("_2"));`
`forEachOrdered`	применит функцию ко всем элементам по очереди, порядок выполнения гарантировать может	`list.stream().forEachOrdered((p) -> p.append("_nv"));`
`toArray`	приведет значения стрима к массиву	`collection.stream().map(String::toLowerCase).toArray(String[]::new);`
`reduce`	преобразует все элементы в один объект	`collection.stream().reduce((c1, c2) -> c1 + c2).orElse(0)`

Java Streams vs. Kotlin Sequences

Streams come from Java, where there are no inline functions, so Streams are the only way to use these functional chains on a collection in Java. Kotlin can do them directly on Iterables, which is better for performance in many cases because intermediate Stream or Sequence objects don't need to be created.

Kotlin has Sequences as an alternative to Streams with these advantages:

- They use null to represent missing items instead of Optional. Nullable values are easier to work with in Kotlin because of its null safety features. It's also better for performance to avoid wrapping all the items in the collection.
- Some of the operators and aggregation functions are much more concise and avoid having to juggle generic types (compare Sequence.groupBy to Stream.collect).
- There are more operators provided for Sequences, which result in performance advantages and simpler code by cutting out intermediate steps.
- Many terminal operators are inline functions, so they omit the last wrapper that a Stream would need.
- The sequence builder lets you create a complicated lazy sequence of items with simple sequential syntax in a coroutine. Very powerful.
- They work back to Java 1.6. Streams require Java 8 or higher. This one is irrelevant for Kotlin 1.5 and higher since Kotlin now requires JDK 8 or higher.

The reason is that designing Sequences for Kotlin first allows them to take advantage of all the features available in Kotlin that are not available in Java. An obvious one is null safety — when using Stream, all types in all lambdas will be platform types. Another is extension functions, which allow Sequences to have a (much) richer, and arguably simpler, API. You can read about further advantages here.

It should be mentioned that one thing Streams have that Sequences don’t is the parallel() method. However, this is something you’re actually discouraged from using, and the same thing can be achieved in a safer and much more powerful manner using Kotlin Flows.

Java NIO https://jenkov.com/tutorials/java-nio/index.html

In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.

Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.

Java NIO **Channels** are similar to streams with a few differences:

- You can both read and write to a Channels. Streams are typically one-way (read or write).
- Channels can be read and written asynchronously.
- Channels always read to, or write from, a Buffer.

A **buffer** is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

The Java NIO **Selector** is a component which can examine one or more Java NIO Channel instances, and determine which channels are ready for e.g. reading or writing. This way a single thread can manage multiple channels, and thus multiple network connections.

Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.

Kotlin DSL

By using well-named functions as builders in combination with function literals with receiver it is possible to create type-safe, statically-typed builders in Kotlin.

Type-safe builders allow creating Kotlin-based domain-specific languages (DSLs) suitable for building complex hierarchical data structures in a semi-declarative way. Sample use cases for the builders are:

- Generating markup with Kotlin code, such as HTML or XML
- Configuring routes for a web server: Ktor

Пример

```
import com.example.html.* // see declarations below

fun result() =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup to XML"}

            // an element with attributes and text content
            a(href = "https://kotlinlang.org") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // content generated by
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

Шаблоны проектирования

повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста.