ДИСЦИПЛИНА **Алгоритмы и структуры данных с использованием компилируемых языков**

(полное наименование дисциплины без сокращений)

ИНСТИТУТ **Искусственного интеллекта**

КАФЕДРА **Технологий Искусственного Интеллекта**

(полное наименование кафедры)

ВИД УЧЕБНОГО МАТЕРИАЛА **Материалы для практических/семинарских занятий**

(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ **Куликов Александр Анатольевич**

(фамилия, имя, отчество)

СЕМЕСТР **1, 2023-2024**

(указать семестр обучения, учебный год)

Практическая работа №4

по дисциплине «Алгоритмы и структуры данных с использованием компилируемых языков»

Тема: Реактивные протоколы. Протокол Rsocket

Теоретическое введение

Rsocket это протокол, обеспечивающий семантику Reactive Streams.

Это двоичный протокол связи точка-точка, разработанный для использования в распределенных приложениях. В этом смысле он предоставляет альтернативу другим протоколам, таким как HTTP.

С RSocket можно узнать, когда лучше отправлять запрос, а когда нет. Сделать это с HTTP нельзя.

RSocket имеет реализации на нескольких языках. Библиотека Java построена на Project Reactor и Reactor Netty для транспорта. Это означает, что сигналы от Reactive Streams Publishers в вашем приложении распространяются через RSocket по сети.

Протокол RSocket использует транспортный протокол более низкого уровня для передачи кадров RSocket .

RSocket позволяет общаться с использованием следующих транспортных протоколов:

- TCP
- WebSocket
- Aeron
- HTTP / 2 Stream

RSocket пользуется доверием и поддержкой некоторых крупнейших компаний в Интернете, таких как Netifi, Pivotal, Facebook, Netflix, Alibaba и других.

Почему RSocket ?

Использование микросервисов очень популярно сейчас.

Но ситуация усложняется, если решите запустить свои приложения в облаке, где проблемы с сетью, периоды повышенной задержки и большой поток запросов — это то, чего вы не можете полностью избежать. Вместо того, чтобы пытаться исправить проблемы с сетью, лучше сделать вашу архитектуру устойчивой и полностью работоспособной даже при таких условиях.

Давайте посмотрим на пример:

У нас есть много микросервисов, которые общаются друг с другом через HTTP. Мы также используем облачные серверы (AWS, GCP, Azure). Каждый компонент предоставляет простые REST API.

Первая проблема — модель взаимодействия запрос/ответ HTTP . Некоторые шаблоны коммуникации трудно реализовать эффективным способом, используя модель взаимодействия запрос / ответ. Даже выполнение простой операции «fire and forget» имеет побочные эффекты — сервер должен отправить ответ обратно клиенту, даже если клиенту это не нужно.

Вторая проблема — производительность.

Обмен сообщениями на основе RabbitMQ, gRPC или даже HTTP 2 с его поддержкой мультиплексирования будет намного лучше с точки зрения производительности и эффективности, чем простой HTTP 1.x.

Дополнительные ресурсы влекут за собой дополнительные расходы, хотя все, что нам нужно, — это простое сообщение “fire and forget”.

Кроме того, множество различных протоколов может создавать серьезные проблемы, связанные с управлением приложениями, особенно если наша система состоит из сотен микросервисов.

Упомянутые выше проблемы являются основными причинами, по которым RSocket был изобретен. Благодаря своей реактивности и встроенной надежной модели взаимодействия RSocket может применяться в различных бизнес-сценариях.

Разница между gRPC, WebSocket и HTTP

HTTP.1 — это хорошее решение, проверенное временем, но оно не будет работать на производительность и реактивность.

WebSocket слишком сложен для разработки и в тяжелых стресс-тестах не работает так, как ожидалось.

gRPC — несомненно лучше, чем HTTP и WebSocket, он прост в разработке. Он работает поверх HTTP.1 (то есть, по HTTP.2) — на самом деле, это не совсем так, то есть это просто оболочка над HTTP. 2. Для каждого действия отправляется обычный запрос POST (HTTP). Поэтому это неэффективное использование

ресурсов. Он использует прокси (envoy) между браузером и сервером и отправляет регулярные HTTP-запросы POST к прокси, и прокси уже отправляет HTTP.2 на сервер. С помощью больших стресс-тестов он показал себя не очень хорошо, если, например, у нас нет очень мощного сервера.

gRPC и RSocket пытаются решить разные проблемы. gRPC — это среда RPC, использующая HTTP/2.

gRPC и RSocket находятся на разных уровнях в стеке.

gRPC находится на уровне 7 OSI — уровне RPC, построенном поверх HTTP/2.

RSocket — это уровень OSI 5/6, который моделирует семантику Reactive Streams по сети. Реактивные потоки позволяют моделировать асинхронные потоки с backpressure.

gRPC предназначен для работы с семантикой HTTP/2. Если вы хотите отправить его через другой транспорт, вы должны имитировать семантику HTTP / 2. По всей сети он фактически связан с HTTP/2 и TCP.

RSocket требуется только дуплексное соединение — т.е. то, что может отправлять и получать байты. Это может быть TCP, WebSockets и т. д.

У gRPC есть традиционная модель клиент-сервер, потому что она основана на семантике HTTP/2 и RPC. В gRPC клиент подключается к серверу, но сервер не может совершать вызовы клиенту.

RSocket не имеет взаимодействия клиент-сервер в традиционном смысле HTTP. В RSocket, когда клиент подключается к серверу, они оба могут быть запросчиком и ответчиком. Это означает, что после того, как они подключены, RSocket является полностью дуплексным, и запрашивающая сторона сервера может инициировать вызовы запрашивающей стороне клиента — то есть сервер может совершать вызовы в веб-браузер после подключения.

Разница между WebSocket и RSocket

Веб-сокеты не обеспечивают backpressure на уровне приложения, только backpressure на уровне байтов на основе TCP. Веб-сокеты также обеспечивают только создание, они не обеспечивают семантику приложения. Разработчик должен разработать протокол приложения для взаимодействия с веб-сокетом.

RSocket обеспечивает кадрирование, семантику приложения, backpressure на уровне приложения, и оно не привязано к конкретному транспорту.

Разница между HTTP и RSocket

RSocket — это протокол приложений для реактивных потоков, который обеспечивает управление потоком приложений по сети, чтобы предотвратить

сбои и повысить отказоустойчивость. В отличие от HTTP, RSocket не ожидает ответа или запроса от клиента.

В отличие от этого, RSocket использует идею асинхронной обработки потока с неблокирующим противодавлением , при котором отказавший компонент будет вместо того, чтобы просто отбрасывать трафик, сообщать о своих нагрузках вышестоящим компонентам, заставляя их снижать нагрузку и позволяя системе «Изящно реагировать на нагрузку, а не разрушаться под ней», согласно глоссарию Reactive Manifesto .

Основные понятия

RSocket использует кадрирование.

Кадр RSocket — это отдельное сообщение, которое содержит запрос, ответ или обработку протокола. К кадру RSocket может добавляться 24-битное поле длины кадра (Frame Length), представляющее длину кадра в байтах. Зависит от базового транспортного протокола, используемого RSocket, поле длины кадра может не требоваться.

Кадры RSocket начинаются с заголовка RSocket Frame.

RSocket поддерживает два типа полезных данных: данные и метаданные. Данные и метаданные могут быть закодированы в разных форматах.

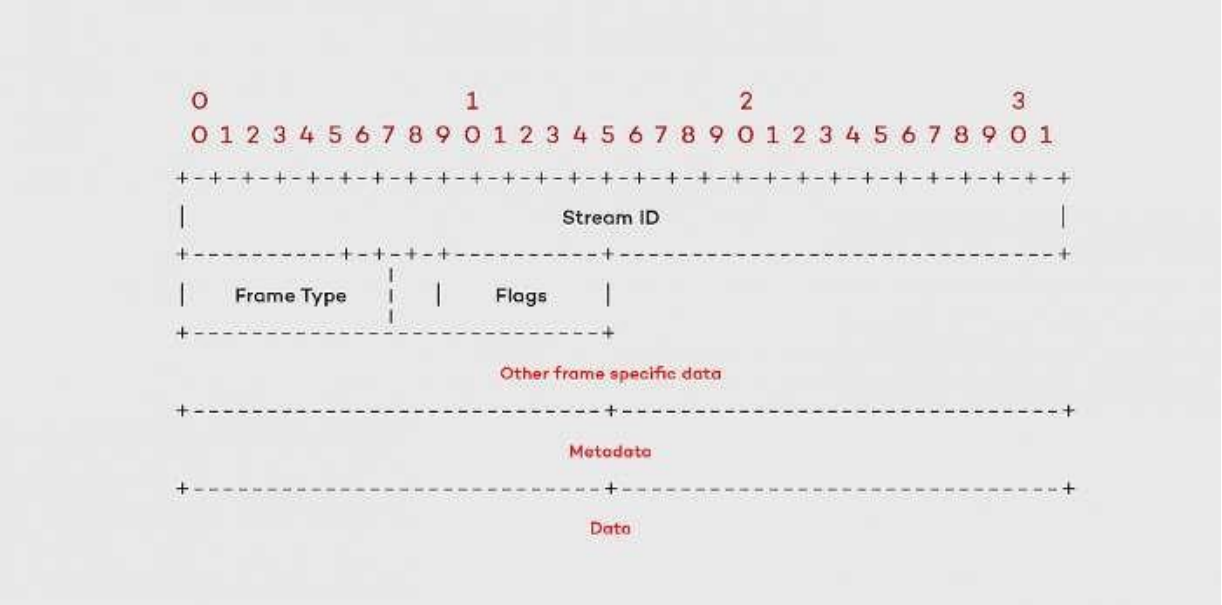
В настоящее время в протоколе RSocket имеется 16 типов кадров:

- SETUP - Настройка подключения. Всегда использует идентификатор потока 0.
- REQUEST_RESPONSE - Используется в модели запроса-ответа. Запрос одного сообщения.
- REQUEST_STREAM - Запрашивается поток сообщений.
- REQUEST_FNF - Используется в модели «fire-and-forget». Отправляет сообщение и не ждет ответа.
- REQUEST_CHANNEL - Используется в модели канала. Запрашивает поток сообщений в обоих направлениях.
- REQUEST_N - Запрашивает больше данных и используется для контроля потока.
- PAYLOAD - Полезная нагрузка сообщения.
- ERROR - Ошибка на уровне соединения или приложения.
- CANCEL - Отмена невыполненного запроса.

Принцип работы

Взаимодействие в RSocket разбито на фреймы. Каждый кадр состоит из заголовка кадра, который содержит идентификатор потока, определение типа кадра и другие данные, относящиеся к типу кадра. За заголовком кадра следуют

метаданные и полезная нагрузка — эти части несут данные, указанные пользователем.



Клиент отправляет установочный фрейм на сервер в самом начале связи. Этот кадр можно настроить так, чтобы вы могли добавлять свои собственные правила безопасности или другую информацию, требуемую при инициализации соединения. Следует отметить, что RSocket не различает клиента и сервер после фазы настройки соединения. Каждая сторона может начать отправку данных другой — это делает протокол почти полностью симметричным.

Кадры отправляются в виде потока байтов. Это делает RSocket более эффективным, чем обычные текстовые протоколы. С точки зрения разработчика, легче отлаживать систему, когда JSON летают туда-сюда по сети, но влияние на производительность делает такое удобство очень и очень сомнительным. Протокол не навязывает какой-либо конкретный механизм сериализации/десериализации, он рассматривает кадр как пакет битов, которые могут быть преобразованы во что угодно.

Следующим фактором, который оказывает огромное влияние на производительность RSocket, является мультиплексирование. Протокол создает логические потоки (каналы) поверх единственного физического соединения. Каждый поток имеет свой уникальный идентификатор, который в некоторой степени можно интерпретировать как очередь. Такой дизайн имеет дело с основными проблемами, известными из HTTP 1.x — модель соединения на запрос и слабая производительность «конвейерной обработки».

Более того, RSocket изначально поддерживает передачу больших полезных нагрузок. В таком случае кадр полезной нагрузки разделяется на несколько кадров с дополнительным флагом — порядковым номером данного фрагмента.

RSocket использует Reactor, поэтому на уровне API мы в основном работаем с объектами Mono и Flux.

Интеграция с Spring

Maven:

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-rsocket</artifactId>
  <version>5.2.3.RELEASE</version>
</dependency>
```

Gradle:

```
compile "org.springframework.integration:spring-integration-rsocket:5.2.3.RELEASE"
```

Spring Boot 2.2 поддерживает установку сервера RSocket через TCP или WebSocket.

Существует также поддержка клиентов и автоматическая настройка для RSocketRequester.Builder и RSocketStrategies.

Характеристики

- Он бинарный.

Вся отправка данных уже оптимизирована по максимуму. Все что надо, это сконвертировать сообщение в байты и потом деконвертировать.

RSocket все сделает за вас.

Если данные большие, RSocket сделает фрейминг сообщения и удостовериться в том, что оно целое и невредимое придет к получателю.

- Он является Multiplexed

Он позволяет только одно соединение для всех логических стримов.

- Bi-directional

Как только мы получили соединение, обе стороны могут как запрашивать данные так и отдавать их.

- Backpressure

  RSocket — это имплементация реактивных стримов поверх Network.

  Он предоставляет настоящий backpressure.

Например когда вы говорите ему: дай мне 11 элементов, он конвертирует их в фрейм, отправляет на другую сторону (получателю), декодирует его и нотифицирует продьюсера, сколько элементов ему надо отправить (т.е. больше 11 он не сможет отправить никак!).

- Возобновление сессии

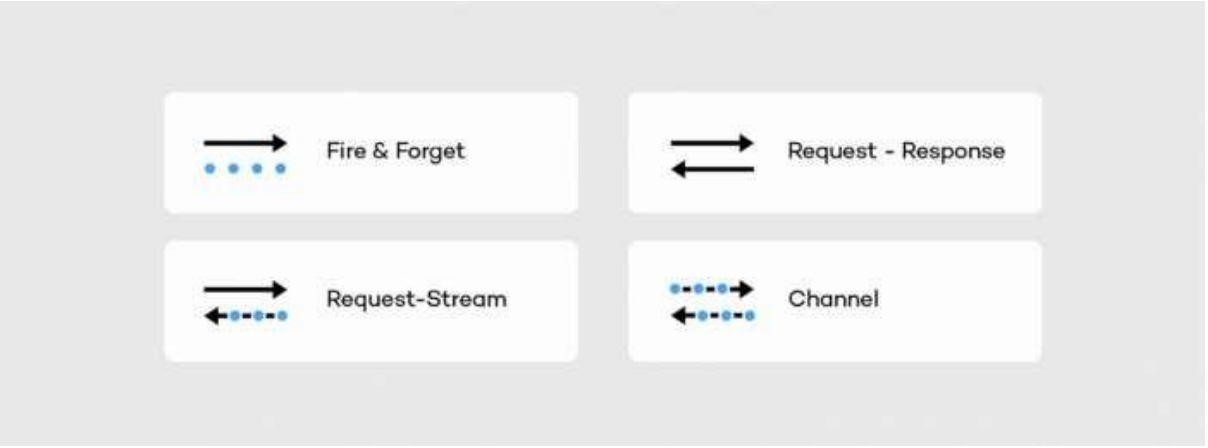
Управление состоянием прозрачно для приложений и хорошо работает в сочетании с backpressure, которое может по возможности остановить производителя и уменьшить количество требуемого состояния. Возобновление — это возможность возобновить работу в случае сбоя (например, восстановление внезапно закрытого соединения).

Это особенно полезно, так как при отправке кадра RESUME, содержащего информацию о последнем принятом кадре, клиент может возобновить соединение и запрашивать только те данные, которые он еще не получил, избегая ненужной нагрузки на сервер и тратя время на попытки восстановить данные, которые уже были получены.

Его следует использовать везде, где это имеет смысл.

Стратегии использования

- Request-Response — отправляет сообщение и получает результат
- Request-Stream — отправляет сообщение и получает обратно поток данных
- Channel — отправляет потоки сообщений в обоих направлениях
- Fire-and-Forget — отправляет одностороннее сообщение и не ждет ответа



Fire-and-Forget

Здесь клиент не получит ответа от сервера. Метод fireAndForget() в основном используется для односторонних push-уведомлений. Тип возвращаемого значения определяется как Mono<Void>.

Давайте создадим 2 простых сервиса: сервер и клиент.

Сервер:

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.2.4.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    developmentOnly
    runtimeClasspath {
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/milestone' }
}
```

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
}

test {
    useJUnitPlatform()
}
```

файл настроек выглядит так

```
spring.rsocket.server.port=7000
spring.main.lazy-initialization=true
```

И основной класс выглядит очень просто.

Чтобы избежать необходимости реализации каждого метода интерфейса RSocket, API предоставляет абстрактные возможности, при помощиAbstractRSocket который мы можем расширить. Соединяя SocketAcceptor и AbstractRSocket , мы получаем реализацию на стороне сервера, которая в базовом сценарии может выглядеть следующим образом:

```
@SpringBootApplication
public class ServerRsocketApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServerRsocketApplication.class, args);

    RSocket rSocketImpl = new AbstractRSocket() {
      @Override
      public Mono<Void> fireAndForget(Payload payload) {
```

```
            System.out.println(payload.getDataUtf8());
            return Mono.empty();
          }
        };

        Disposable server = RSocketFactory.receive()
            .acceptor((setupPayload, reactiveSocket) -> Mono.just(rSocketImpl))
            .transport(TcpServerTransport.create("localhost", 7000))
            .start()
            .subscribe();

        server.dispose();

      }

    }
```

Для простоты напишем все в одном классе. Здесь RSocket использует TCP на порту 7000.

Код сервера реализует необходимую функциональность, переопределяя метод findAndForget() , который наследуется от AbstractRSocket .

Модель findAndForget не возвращает данные с сервера. Таким образом, он возвращает Mono<Void>, который объявляет о возврате Mono.empty().

И мы использовали TcpClientTransport для транспорта. RSocket дает вам возможность выбрать либо TCP, либо WebSocket (если вы хотите работать с WebSocket, вы можете использовать WebsocketClientTransport — вместо TcpServerTransport).

receive() — означает, что он получит запрос от клиента.

acceptor - определяет, как обрабатывать запрос, он принимает класс SocketAcceptor. Это место, где мы можем встроить наш обработчик для обработки входящего запроса. AbstractRSocket — это удобный класс, предоставляющий методы для всех интерактивных моделей, поддерживаемых в

RSocket. Например, здесь нам нужно обрабатывать модель fire-and-forget только путем переопределения метода requestResponse.

transport - определяет информацию о сервере, включая хост, порт, протокол и т. Д. Здесь мы используем TCP в качестве протокола и запускаем его на локальном хосте на порту 7000.

Клиент:

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.2.4.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    developmentOnly
    runtimeClasspath {
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
```

```
    maven { url 'https://repo.spring.io/milestone' }
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
    testImplementation 'io.projectreactor:reactor-test'
}

test {
    useJUnitPlatform()
}
```

На стороне отправителя использование модели взаимодействия довольно просто, ведь все, что нам нужно сделать, это вызвать конкретный метод для экземпляра RSocket, который мы создали с помощью RSocketFactory.

Основной класс:

```
@SpringBootApplication
public class ClientRsocketApplication {

   public static void main(String[] args) throws InterruptedException {
      SpringApplication.run(ClientRsocketApplication.class, args);

      RSocket socket =
          RSocketFactory.connect()
              .transport(TcpClientTransport.create("localhost", 7000))
              .start()
              .block();
```

```
        socket
            .fireAndForget(DefaultPayload.create("Client service is available now"))
            .subscribe();

        Thread.sleep(3000);
        socket.dispose();

    }

}
```

На клиентской стороне мы также используем RSocketFactory, но на этот раз с методом connect() вместо receive().

connect()- указывает на то, что текущий клиент будет подключаться к серверу.

transport - инфа о сервере, на который мы будем подключаться.

Метод start() возвращает RSocket, его можно использовать для взаимодействия со стороной сервера. Здесь мы отправляем запрос типа requestResponse, он требует ответа от сервера.

Наконец, мы можем использовать subcribe() для отображения полученного ответа от сервера.

После запуска ожидаем следующее:

Клиент отправил сообщение «Client-service is available now» модели findAndForget и сервер вывел полученное сообщение в консоль. Это очень простой пример.

Например, это может быть полезно, когда один из наших сервисов вышел из строя (сбой сети или большая нагрузка), а затем поднялся и отправил так называемое push-уведомление другому сервису о том, что он жив.

Request-Channel

Благодаря мультиплексированию и поддержке двунаправленной передачи данных, мы можем сделать очень интересную на мой взгляд вещь, используя метод request channel. RSocket может передавать данные от запрашивающей стороны к ответчику и наоборот, используя одно физическое соединение. Такое взаимодействие может быть полезно, когда запрашивающая сторона например обновляет подписку. Без двунаправленного канала клиент должен был бы отменить поток и повторно запросить его с новыми параметрами.

Давайте создадим 2 простых сервиса: сервер и клиент.

Сервер:

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.2.4.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    developmentOnly
    runtimeClasspath {
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}
```

```
repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/milestone' }
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
}

test {
    useJUnitPlatform()
}
```

файл настроек выглядит так

```
spring.rsocket.server.port=7000
spring.main.lazy-initialization=true
```

Основной класс

```
@SpringBootApplication
public class ServerRsocketApplication {

   public static void main(String[] args) {
      SpringApplication.run(ServerRsocketApplication.class, args);

      RSocket rSocketImpl = new AbstractRSocket() {
         @Override
         public Flux<Payload> requestChannel(Publisher<Payload> payloads) {
```

```
            return Flux.from(payloads).flatMap(payload ->
                Flux.fromStream(
                    payload.getDataUtf8().codePoints()
                        .mapToObj(c -> String.valueOf((char) c))
                        .map(i -> DefaultPayload.create("channel: " + i))))
                .doOnNext(System.out::println);
          }
        };

        Disposable server = RSocketFactory.receive()
            .acceptor((setupPayload, reactiveSocket) -> Mono.just(rSocketImpl))
            .transport(TcpServerTransport.create("localhost", 7000))
            .start()
            .subscribe();

        server.dispose();

      }

    }
```

receive() — означает, что он получит запрос от клиента.

acceptor — определяет, как обрабатывать запрос, он принимает класс SocketAcceptor. Это место, где мы можем встроить наш обработчик для обработки входящего запроса. AbstractRSocket — это удобный класс, предоставляющий методы для всех интерактивных моделей, поддерживаемых в RSocket. Например, здесь нам нужно обрабатывать модель request-channel только путем переопределения метода requestResponse.

transport — определяет информацию о сервере, включая хост, порт, протокол и т. Д. Здесь мы используем TCP в качестве протокола и запускаем его на локальном хосте на порту 7000.

Клиент:

build.gradle

```
plugins {
   id 'org.springframework.boot' version '2.2.4.RELEASE'
   id 'io.spring.dependency-management' version '1.0.9.RELEASE'
   id 'java'
}apply plugin: 'io.spring.dependency-management'group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'configurations {
   developmentOnly
   runtimeClasspath {
      extendsFrom developmentOnly
   }
   compileOnly {
      extendsFrom annotationProcessor
   }
}repositories {
   mavenCentral()
   maven { url 'https://repo.spring.io/snapshot' }
   maven { url 'https://repo.spring.io/milestone' }
}dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-rsocket'
   implementation 'org.springframework.boot:spring-boot-starter-webflux'
   compileOnly 'org.projectlombok:lombok'
   developmentOnly 'org.springframework.boot:spring-boot-devtools'
   annotationProcessor 'org.projectlombok:lombok'
   testImplementation('org.springframework.boot:spring-boot-starter-test') {
      exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
      exclude group: 'junit', module: 'junit'
   }
   testImplementation 'io.projectreactor:reactor-test'
}test {
   useJUnitPlatform() }
```

На стороне отправителя использование модели взаимодействия довольно просто, ведь все, что нам нужно сделать, это вызвать конкретный метод для экземпляра RSocket, который мы создали с помощью RSocketFactory.

Основной класс:

```
@SpringBootApplication
public class ClientRsocketApplication {

  public static void main(String[] args) {
    SpringApplication.run(ClientRsocketApplication.class, args);

    RSocket socket =
        RSocketFactory.connect()
            .transport(TcpClientTransport.create("localhost", 7000))
            .start()
            .block();

    socket.requestChannel(Flux.just("one", "two")
        .map(DefaultPayload::create))
        .delayElements(Duration.ofMillis(1000))
        .map(Payload::getDataUtf8)
        .doOnNext(System.out::println)
        .blockLast();

  }

}
```

На клиентской стороне мы также используем RSocketFactory, но на этот раз с методом connect() вместо receive().

connect()- указывает на то, что текущий клиент будет подключаться к серверу.

transport — инфа о сервере, на который мы будем подключаться.

Метод start() возвращает RSocket, его можно использовать для взаимодействия со стороной сервера. Здесь мы отправляем запрос типа requestResponse, он требует ответа от сервера.

Можем использовать subcribe() для отображения полученного ответа от сервера.

Request-Response

RSocket также может имитировать поведение HTTP. Он поддерживает семантику запрос-ответ, и, вероятно, это будет основной тип взаимодействия, которое вы собираетесь использовать с RSocket.

В контексте потоков такая операция может быть представлена в виде потока, который состоит из одного объекта. В этом сценарии клиент ожидает ответный кадр, но делает это полностью неблокирующим образом.

Fire-and-Forgot вернул Mono<Void>, но метод RequestResponse возвращает Mono<Payload>, так как клиент должен доставить один объект и получить ответ от сервера. Однако мы не используем Flux, потому что это только один из ответов на наш запрос от клиента.

Давайте создадим 2 простых сервиса: сервер и клиент.

Сервер:

build.gradle

```
plugins {
   id 'org.springframework.boot' version '2.2.4.RELEASE'
   id 'io.spring.dependency-management' version '1.0.9.RELEASE'
   id 'java'
}apply plugin: 'io.spring.dependency-management'group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'configurations {
   developmentOnly
```

```
    runtimeClasspath {
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/milestone' }
}dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
}test {
    useJUnitPlatform()
}
```

файл настроек выглядит так

```
spring.rsocket.server.port=7000
spring.main.lazy-initialization=true
```

Основной класс

```
@SpringBootApplication
public class ServerRsocketApplication {

   public static void main(String[] args) {
      SpringApplication.run(ServerRsocketApplication.class, args);

      RSocket rSocketImpl = new AbstractRSocket() {
         @Override
```

```
            public Mono<Payload> requestResponse(Payload payload) {
              System.out.println(payload.getDataUtf8());
              return Mono.just(DefaultPayload.create("Nice to meet you"));
            }
          };

          Disposable server = RSocketFactory.receive()
              .acceptor((setupPayload, reactiveSocket) -> Mono.just(rSocketImpl))
              .transport(TcpServerTransport.create("localhost", 7000))
              .start()
              .subscribe();

          server.dispose();

        }

      }
```

В приведенном выше коде RSocketFactory.receive() создает объект ServerRSocketFactory для построения сервера RSocket.

ServerRSocketFactory.acceptor() устанавливает объект SocketAcceptor для приема соединений.

Объект AbstractRSocket реализует только метод requestResponse() для обработки модели запрос-ответ. Тип возврата Mono<Payload> означает, что в качестве ответа ожидается не более одного объекта Payload.

Для каждого полученного запроса ответом будет полезная нагрузка запроса с ECHO >> в качестве префикса.

Метод DefaultPayload.create() — это простой способ создания объектов Payload.

ServerRSocketFactory.transport() устанавливает транспортный уровень, используемый RSocket. Здесь TcpServerTransport используется для TCP на порту 7000.

Клиент:

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.2.4.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}apply plugin: 'io.spring.dependency-management'group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'configurations {
    developmentOnly
    runtimeClasspath {
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/milestone' }
}dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
    testImplementation 'io.projectreactor:reactor-test'
}test {
```

```
    useJUnitPlatform()
}
```

На стороне отправителя использование модели взаимодействия довольно просто, ведь все, что нам нужно сделать, это вызвать конкретный метод для экземпляра RSocket, который мы создали с помощью RSocketFactory.

Основной класс:

```
@SpringBootApplication
public class ClientRsocketApplication {

  public static void main(String[] args) throws InterruptedException {
    SpringApplication.run(ClientRsocketApplication.class, args);

    RSocket socket =
        RSocketFactory.connect()
            .transport(TcpClientTransport.create("localhost", 7000))
            .start()
            .block();

    assert socket != null;

    socket.requestResponse(DefaultPayload.create("Hello!"))
        .map(Payload::getDataUtf8)
        .doOnNext(System.out::println)
        .block();

    socket.dispose();

  }

}
```

На клиентской стороне мы также используем RSocketFactory, но на этот раз с методом connect() вместо receive().

connect()- указывает на то, что текущий клиент будет подключаться к серверу.

transport — инфа о сервере, на который мы будем подключаться.

Метод start() возвращает RSocket, его можно использовать для взаимодействия со стороной сервера. Здесь мы отправляем запрос типа requestResponse, он требует ответа от сервера.

Наконец, мы можем использовать subcribe() для отображения полученного ответа от сервера.

Request-Stream

В случае операции request-stream запрашивающая сторона отправляет ответчику один кадр и возвращает поток данных. Вместо отправки периодических запросов сереру (клиенту) можно подписаться на поток и реагировать на поступающие данные — они будут поступать автоматически, когда они станут доступны.

Давайте создадим 2 простых сервиса: сервер и клиент.

Сервер:

build.gradle

```
plugins {
   id 'org.springframework.boot' version '2.2.4.RELEASE'
   id 'io.spring.dependency-management' version '1.0.9.RELEASE'
   id 'java'
}apply plugin: 'io.spring.dependency-management'group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'configurations {
   developmentOnly
   runtimeClasspath {
```

```
        extendsFrom developmentOnly
    }
    compileOnly {
        extendsFrom annotationProcessor
    }
}repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/milestone' }
}dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-rsocket'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
}test {
    useJUnitPlatform()
}
```

файл настроек выглядит так

```
spring.rsocket.server.port=7000
spring.main.lazy-initialization=true
```

Основной класс

```
@SpringBootApplication
public class ServerRsocketApplication {

   public static void main(String[] args) {
      SpringApplication.run(ServerRsocketApplication.class, args);

      RSocket rSocketImpl = new AbstractRSocket() {
         @Override
         public Flux<Payload> requestStream(Payload payload) {
```

```
            System.out.println(payload.getDataUtf8());
            return Flux.range(1, 5)
                .map(i -> DefaultPayload.create("onNext-" + i));
          }
        };

        Disposable server = RSocketFactory.receive()
            .acceptor((setupPayload, reactiveSocket) -> Mono.just(rSocketImpl))
            .transport(TcpServerTransport.create("localhost", 7001))
            .start()
            .subscribe();

        server.dispose();

      }

    }
```

Для простоты напишем все в одном классе. Здесь RSocket использует TCP на порту 7000.

Код сервера реализует необходимую функциональность, переопределяя метод request-stream() , который наследуется от AbstractRSocket .

Модель findAndForget не возвращает данные с сервера. Таким образом, он возвращает Mono<Void>, который объявляет о возврате Mono.empty().

И мы использовали TcpClientTransport для транспорта. RSocket дает вам возможность выбрать либо TCP, либо WebSocket (если вы хотите работать с WebSocket, вы можете использовать WebsocketClientTransport — вместо TcpServerTransport).

receive() — означает, что он получит запрос от клиента.

acceptor — определяет, как обрабатывать запрос, он принимает класс SocketAcceptor. Это место, где мы можем встроить наш обработчик для обработки входящего запроса. AbstractRSocket — это удобный класс,

предоставляющий методы для всех интерактивных моделей, поддерживаемых в RSocket. Например, здесь нам нужно обрабатывать модель request-stream только путем переопределения метода requestResponse.

transport — определяет информацию о сервере, включая хост, порт, протокол и т. Д. Здесь мы используем TCP в качестве протокола и запускаем его на локальном хосте на порту 7000.

Клиент:

build.gradle

```
plugins {
   id 'org.springframework.boot' version '2.2.4.RELEASE'
   id 'io.spring.dependency-management' version '1.0.9.RELEASE'
   id 'java'
}apply plugin: 'io.spring.dependency-management'group = 'com.stergioulas.tutorials.springbootrsocket'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'configurations {
   developmentOnly
   runtimeClasspath {
      extendsFrom developmentOnly
   }
   compileOnly {
      extendsFrom annotationProcessor
   }
}repositories {
   mavenCentral()
   maven { url 'https://repo.spring.io/snapshot' }
   maven { url 'https://repo.spring.io/milestone' }
}dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-rsocket'
   implementation 'org.springframework.boot:spring-boot-starter-webflux'
   compileOnly 'org.projectlombok:lombok'
   developmentOnly 'org.springframework.boot:spring-boot-devtools'
```

```
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        exclude group: 'junit', module: 'junit'
    }
    testImplementation 'io.projectreactor:reactor-test'
}test {
    useJUnitPlatform()
}
```

На стороне отправителя использование модели взаимодействия довольно просто, ведь все, что нам нужно сделать, это вызвать конкретный метод для экземпляра RSocket, который мы создали с помощью RSocketFactory.

Основной класс:

```
@SpringBootApplication
public class ClientRsocketApplication {

  public static void main(String[] args) throws InterruptedException {
    SpringApplication.run(ClientRsocketApplication.class, args);

    RSocket socket =
        RSocketFactory.connect()
            .transport(TcpClientTransport.create("localhost", 7001))
            .start()
            .block();

    socket.requestStream(DefaultPayload.create("request-stream example!"))
        .delayElements(Duration.ofMillis(1000))
        .subscribe(
            payload -> System.out.println(payload.getDataUtf8()),
            e -> System.out.println("error" + e.toString()),
            () -> System.out.println("completed")
        );

    Thread.sleep(3000);
```

```
        socket.dispose();

    }

}
```

На клиентской стороне мы также используем RSocketFactory, но на этот раз с методом connect() вместо receive().

connect()- указывает на то, что текущий клиент будет подключаться к серверу.

transport — инфа о сервере, на который мы будем подключаться.

Метод start() возвращает RSocket, его можно использовать для взаимодействия со стороной сервера. Здесь мы отправляем запрос типа requestResponse, он требует ответа от сервера.

И в конце мы можем использовать subcribe() для отображения полученного ответа от сервера.

Задание на практическую работу №4

Задание

Разработать клиент-серверную систему с использованием протокола RSocket.

Предметная область разрабатываемой системы выбирается студентом самостоятельно, но должна быть уникальна в рамках учебной группы.

Система должна удовлетворять следующим требованиям:

- Должны быть продемонстрированы все 4 типа взаимодействия в рамках протокола RSocket:
  - Request-Response;
  - Request-Stream;
  - Channel;
  - Fire-and-Forget).
- Должна быть продемонстрирована работа с операторами преобразования потоков.
- Должно быть продемонстрировано взаимодействие с базой данных.
- Потоки должны закрываться после отработки бизнес-логики.
- Серверная часть приложения должна быть покрыта Unit-тестами.