1. Что такое программное обеспечение

Программное обеспечение (Software) - совокупность программ, выполняемых вычислительной системой.

Программное обеспечение — *неотъемлемая часть программно-аппаратного комплекса, поскольку сейчас даже в бытовой технике есть ПО системы*. Оно является управляющим элементом для технических средств, определяющим то, как они будут функционировать. Сфера применения конкретного компьютера определяется созданным для него ПО, однако, архитектура и аппаратная часть тоже важны, бортовой компьютер для автомобиля, самолета или космического корабля – это изделие, назначение которого уже заложено в аппаратной части, и сделать бытовой или бухгалтерский ПК из него либо не получится, либо очень сложно и дорого. Другое дело, когда речь идет о классе вычислительных систем, но и здесь скорее ПО устанавливается под задачи, а также важна аппаратура. Например, если компьютер используется для графических задач, то без специальных аппаратных ресурсов никакое ПО не поможет. Все должно быть в комплексе.

2. Что включает в себя разработка ПО

Разработка ПО включает в себя следующие направления и виды работ:

- технология проектирования программ (например, нисходящее проектирование, структурное и объектно-ориентированное проектирование и др.);
- методы тестирования программ [ссылка, ссылка];
- методы доказательства правильности программ;

- анализ качества работы программ;
- документирование программ;
- разработка и использование программных средств, облегчающих процесс проектирования программного обеспечения.

3. Что такое программа

Программа - упорядоченная последовательности команд. Конечная цель любой компьютерной программы - управление аппаратными средствами.

Состав программного обеспечения ПК называют программной конфигурацией. Между программами, как и между физическими узлами и блоками, существует взаимосвязь - многие программы работают, базируясь на программы более низкого уровня. Возможность существования такого интерфейса обеспечивается распределением программного обеспечения на несколько взаимодействующих между собой уровней.

4. Уровни программного обеспечения

Уровни программного обеспечения представляют собой иерархическую конструкцию. Каждый уровень опирается на программное обеспечение предшествующих уровней, показано на слайде.

Программное обеспечение можно классифицировать на системное и прикладное ПО. Показано на слайде

В свою очередь, системное программное обеспечение классифицируется на: операционные системы, системные интерфейсы, инструментальные средства, системы технического обслуживания

Итак перейдем к уровням ПО.

Базовый уровень - самый низкий уровень программного обеспечения. Он отвечает за взаимодействие с *базовыми аппаратными средствами.* Базовые программные средства входят в состав базового оборудования и хранятся в специальных микросхемах, называемых *постоянными запоминающими устройствами* (*ПЗУ).* Программы и данные записываются («прошиваются») в микросхемы ПЗУ на этапе производства и не могут быть изменены в процессе эксплуатации.

В тех случаях, когда изменение базовых программных средств во время эксплуатации является технически целесообразным (например, *BIOS*), вместо микросхем ПЗУ применяют *перепрограммируемые постоянные запоминающие устройства* (*ППЗУ).* В этом случае изменение содержания ПЗУ можно выполнять как непосредственно в составе ПК, так и вне её, на специальных устройствах, называемых *программаторами.*

- *ПЗУ отличаются наиболее высокой скоростью обмена;*
- *В ряде случаев, когда речь идет о компьютеризированных устройствах, например о бытовой технике, ПЗУ является единственной памятью в этих устройствах, и все необходимое ПО зашито в них;*

Инструментальный уровень (трансляторы и компиляторы языков программирования, системы программирования). Создание программ обеспечивают, текстовые редакторы, среды разработки. Трансляторы и компиляторы осуществляют преобразование созданных на языке высоко уровня программ в корректный набор машинных кодов.

Системный уровень. Переходный уровень. Программы, работающие на этом уровне, обеспечивают взаимодействие прочих программ компьютерной системы с программами базового уровня и непосредственно с аппаратным обеспечением.

От программного уровня этого уровня зависят эксплуатационные показатели всего компьютера в целом. Другой класс программ, системного уровня отвечает за взаимодействие с пользователем. Именно благодаря им мы получаем возможность вводить данные в компьютер, управлять его работой и получать результат в удобной для себя форме. Эти программные средства называют средствами обеспечения пользовательского интерфейса. От них напрямую зависит удобство работы с компьютером и производительность труда на рабочем месте.

Служебный уровень. ПО этого уровня взаимодействует как с программами базового уровня, так и с программами системного уровня. Основное назначение служебных программ (утилит) состоит в автоматизации работ по проверке, наладке и настройке имеющихся программно-аппаратных ресурсов. Во многих случаях они используются для расширения или улучшения функций системных программ. Некоторые служебные программы изначально включают в состав ОС.

Прикладной уровень. ПО прикладного уровня представляет собой комплекс прикладных программ, с помощью которых на данном рабочем месте выполняются конкретные задания. Спектр таких заданий очень широк – от производственных до творческих и развлекательно - обучающих. Огромный функциональный диапазон возможных приложений средств обусловлен наличием прикладных программ для разных видов деятельности. Именно прикладное ПО предназначено для решения задач, необходимых для конечного пользователя, или для решения задач на промежуточных этапах этого процесса».

Задача - проблема, подлежащая решению в интересах пользователя.

Термин "задача" в программировании означает единицу работы вычислительной системы, требующую выделения вычислительных ресурсов (процессорного времени, оперативной и внешней памяти, файлов и т.п.).

Приложение - программная реализация решения задачи на компьютере. Приложение может состоять из одной или нескольких взаимосвязанных и взаимодействующих программ. Условно принято делить программы на небольшие (простые), средней сложности и большие.

Программа считается небольшой как по размерам, так и по другим признакам, если она удовлетворяет следующим признакам:

- решает одну четко поставленную задачу в хорошо известных ограничениях, к тому же, не очень существенную для какой-либо практической или исследовательской деятельности;
- неважно, насколько быстро она работает;
- ущерб от неправильной работы программы - практически нулевой (за исключением возможности обрушения ею системы, в которой выполняются и другие, более важные задачи);
- не требуется дополнять программу новыми возможностями, практически никому не нужно разрабатывать ее новые версии или исправлять найденные ошибки;
- в связи со сказанным выше не очень нужно прилагать к программе подробную и понятную документацию - для человека, который ею заинтересуется, не составит большого труда понять, как ею пользоваться, просто по исходному коду.

5. Какими свойствами обладает сложная программа

Сложные программы, называемые также программными системами, программными комплексами, программными продуктами, отличаются от небольших не только по размерам, но и наличием дополнительных факторов. Эти факторы связаны с их востребованностью и готовностью пользователей платить деньги, как за приобретение самой программы, так и за ее сопровождение и даже за специальное обучение работе с ней.

Обычно сложная программа обладает следующими свойствами:

- программа решает одну или несколько связанных задач;
- существенно, чтобы программа была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, возможно, также специальную документацию для администраторов, а также набор документов для обучения работе с программой;
- низкая производительность программы на реальных данных приводит к значимым потерям для пользователей;
- неправильная работа программы наносит ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто;
- для выполнения своих задач программа должна взаимодействовать с другими программами и программно-аппаратными системами и обеспечивать работу на разных платформах;
- пользователи, работающие с программой, приобретают дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку (реинжиниринг);
- в разработку программы вовлечено значительное количество людей (более 5-ти человек). Большую программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку;
- большая программа имеет намного большее количество ее возможных пользователей по сравнению с небольшими программами, и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.

Примером большой программы может служить стандартная библиотека классов Java или C#, соответствующих систем программирования.
Строго говоря, ни одно из указанных свойств не является обязательным для того, чтобы программу можно было считать большой, но при наличии двух-трех из них достаточно уверенно можно утверждать, что она большая. На основании некоторых из перечисленных свойств можно сделать вывод, что большая программа или программная система чаще всего представляет собой не просто код или исполняемый файл, а включает еще и набор проектной и пользовательской документации.

6. Процесс создания программ

Процесс создания программ можно представить как последовательность следующих действий:

- 1. постановка задачи;
- 2. алгоритмизация решения задачи;
- 3. программирование.

Постановка задачи - это точная формулировка требований, предъявляемых к работе программы, с описанием входной и выходной информации, и, возможно, описание подходов к решению задачи.
Алгоритм - точный набор инструкций, описывающих однозначный порядок действий исполнителя (компьютера), от допустимых исходных данных для достижения результата решения задачи за конечное время.
Программирование (programming) - теоретическая и практическая деятельность, связанная с созданием программ.

7. Информационная система

Информационная система (ИС) — система, предназначена для хранения, поиска и обработки информации, и соответствующие организационные ресурсы, которые обеспечивают и распространяют информацию.

Предназначена для своевременного обеспечения потребителей надлежащей информацией, то есть для удовлетворения конкретных информационных потребностей в рамках определённой предметной области, при этом результатом функционирования информационных систем является информационная продукция — документы, информационные массивы, базы данных и информационные услуги.

Информационная система имеет цель – производство профессиональной информации, связанной с определенной профессиональной деятельностью. Их задача помочь в анализе проблем и создавать новые продукты.

Информационная система - взаимосвязанная совокупность средств, методов и персонала, используемых для хранения, обработки и выдачи информации в интересах достижения поставленной цели.

8. Свойства программного обеспечения:

Следует выделить следующие свойства программного обеспечения:

- Корректность
- Устойчивость
- Восстанавливаемость
- Надежность

Корректность программного обеспечения — свойство безошибочной реализации требуемого алгоритма при отсутствии таких мешающих

факторов, как ошибки входных данных, ошибки операторов, сбои и отказы ЭВМ.

Под корректностью понимаются свойства программы, свидетельствующие об отсутствии в ней ошибок, допущенных разработчиком на различных этапах проектирования (спецификации, проектирования алгоритма и структур данных, кодировании). Корректность самой программы понимают по отношению к целям, поставленным перед ее разработкой (т. е. это относительное свойство).

Устойчивость — свойство осуществлять требуемое преобразование информации при сохранении выходных решений программы в пределах допусков, установленных спецификацией. Устойчивость характеризует поведение программы при воздействии на нее таких факторов неустойчивости, как ошибки операторов ЭВМ, а также не выявленные ошибки программы.

Восстанавливаемость — свойство программного обеспечения, характеризующее возможность приспосабливаться к обнаружению ошибок и их устранению.

Надежность можно представить совокупностью следующих характеристик:

- целостностью программного средства (способностью его к защите от отказов);
- живучестью (способностью к входному контролю данных и их проверки в ходе работы);
- завершенностью (бездефектностью готового программного средства, характеристикой качества его тестирования);
- работоспособностью (способностью программного средства к восстановлению своих возможностей после сбоев).

Отличие понятия корректности и надежности программ состоит в следующем:

• надежность характеризует как программу, так и ее "окружение" (качество аппаратуры, квалификацию пользователя и т. п.);

• говоря о надежности программы, обычно допускают определенную, хотя и малую долю ошибок в ней и оценивают вероятность их появления.

Вернемся к понятию корректности. Очевидно, что не всякая синтаксически правильная программа является корректной, т. е. корректность характеризует семантические свойства программ.

С учетом специфики появления ошибок в программах можно выделить две стороны понятия корректности:

• корректность как точное соответствие целям разработки программы (которые отражены в спецификации) при условии ее завершения или частичная корректность;

• завершение программы, т. е. достижение программой в процессе ее выполнения своей конечной точки.

В зависимости от выполнения или невыполнения каждого из двух названных свойств программы различают шесть задач анализа корректности:

1) доказательство частичной корректности;

2) доказательство частичной некорректности;

3) доказательство завершения программы;

4) доказательство не завершения программы;

5) доказательство тотальной (полной) корректности (т. е. одновременное решение 1-й и 3- й задач);

6) доказательство некорректности (решение 2-й или 4-й задачи).

Характеристики программ показаны на слайде

Мобильность программных продуктов означает их независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и

т.п. Мобильный программный продукт может быть установлен на различных моделях компьютеров и операционных систем, без ограничений на его эксплуатацию в условиях вычислительной сети. Функции обработки такого программного продукта пригодны для массового использования без каких-либо изменений.

Надежность работы программного продукта определяется бессбойностью и устойчивостью в работе программ, точностью выполнения предписанных функций обработки, возможностью диагностики возникающих в процессе работы программ ошибок.

Эффективность программного продукта оценивается как с позиций прямого его назначения - требований пользователя, так и с точки зрения расхода вычислительных ресурсов, необходимых для его эксплуатации.

Учет человеческого фактора означает обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства, хорошей документации для освоения и использования заложенных в программном средстве функциональных возможностей, анализ и диагностику возникших ошибок и др.

Модифицируемость программных продуктов означает способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.

Коммуникативность программных продуктов основана на максимально возможной их интеграции с другими программами, обеспечении обмена данными в общих форматах представления (экспорт/импорт баз данных, внедрение или связывание объектов обработки и др.).

Программы любого вида характеризуются жизненным циклом, состоящим из нескольких этапов:

1. Проектирование структуры программы

2. Создание, тестирование и отладка программы

3. Распространение программы

4. Эксплуатация программы пользователями, послепродажная поддержка

5. Создание новой версии исходной программы или снятие программы с продажи или производства

9. Группы прогаммного обеспечения

В зависимости от условий распространения и использования можно выделить следующие группы ПО:

· Собственническое (или проприетарное)

· Свободное ПО

· ПО с открытым кодом

10. Направления в программировании(расписать)

В программировании выделяют направления:

- Разработка web-приложений
- Разработка desktop-приложений
- Разработка серверных приложений
- Разработка мобильных приложений
- Программирование встраиваемых систем
- Системное программирование
- Разработка игр
- Олимпиадное программирование и решение задач
- Программирование для бухгалтерских и финансовых продуктов
- Программирование баз данных
- Science

Разработка web-приложений-Это направление ориентировано на разработку веб-приложений. Web-программирование можно разделить на backend (написание серверных скриптов – PHP, Python, Ruby) и frontend (разработка юзерского интерфейса – Javascript, HTML, CSS).

Разработка desktop-приложений. Разработка программного обеспечения для различных операционных систем. Все разнообразие софта, что мы используем в повседневности.

Разработка серверных приложений. Это различные игровые сервера (ваша любимая Дотка, CS: GO), IM-сервисы (серверная часть Skype, ICQ, MSN), банковские базы данных.

Разработка мобильных приложений Множество Java-приложений. VK, Viber, Яндекс.Карты, переводчики, электронные читалки.

Программирование встраиваемых систем Отрасль программирования для различной домашней техники: пылесосы, холодильники, стиральные машины, плееры, навигаторы, электронные весы. Здесь задействованы научные разработки с использованием специализированных языков, типа MATLAB.

Системное программирование. Написание различных драйверов для оборудования, программирования «ядра» операционных систем. Кстати, создание компиляторов и интерпретаторов для языков программирования относятся сюда же.

Разработка игр Гигантская отрасль. Сюда включается разработка игр и для ПК, и для консолей, и для мобильный устройств.

Олимпиадное программирование и решение задач Программирование на различных «непрактичных» и не распространенных языках для решения каких-то оригинальных задач.

Программирование для бухгалтерских и финансовых продуктов «1С: Предприятие». Вся бухгалтерия в России завязана на этом продукте. Но недостаточно знать лишь сам язык, важно понимать основы бухгалтерского учета. Плюс в том, что работы очень много, и без хлеба вы не останетесь.

Программирование баз данных Серьезное направление. способность хранить миллиарды строк информации.

Science Наука и этим все сказано. Нейронные сети, моделирование структуры ДНК, запуск спутников, моделирование Большого Взрыва.

11. Основные классы инструментальных средств

В настоящее время выделяют три основных класса инструментальных сред разработки и сопровождения ПС(программных средств) (схема 1):

- инструментальные среды программирования,
- рабочие места компьютерной технологии,
- инструментальные системы технологии программирования.

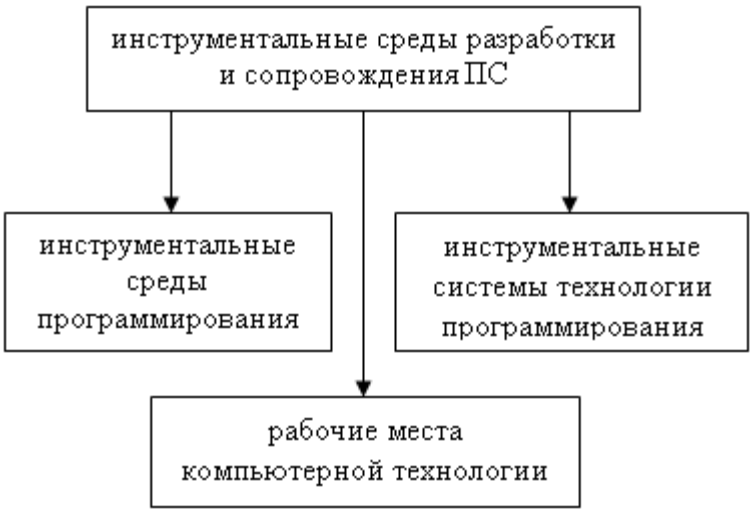


Схема 1.

Инструментальная среда программирования предназначена в основном для поддержки процессов программирования (кодирования), тестирования и отладки ПС. Она не обладает рассмотренными выше свойствами комплексности, ориентированности на конкретную технологию программирования, ориентированности на коллективную разработку и, как правило, свойством интегрированности, хотя имеется некоторая тенденция к созданию интегрированных сред программирования (в этом случае их следовало бы называть *системами программирования*). Иногда среда программирования может обладать свойством специализированности. Признак же ориентированности на конкретный язык программирования может иметь разные значения, что существенно используется для дальнейшей классификации сред программирования.

Рабочее место компьютерной технологии ориентировано на поддержку ранних этапов разработки ПС (системного анализа и спецификаций) и автоматической генерации программ по спецификациям. Оно существенно использует свойства специализированности, ориентированности на конкретную технологию программирования и, как правило, интегрированности. Более поздние рабочие места компьютерной технологии обладают также свойством комплексности. Что же касается

языковой ориентированности, то вместо языков программирования они ориентированы на те или иные формальные языки спецификаций. Свойством ориентированности на коллективную разработку указанные рабочие места в настоящее время, как правило, не обладают.

Инструментальная система технологии программирования предназначена для поддержки всех процессов разработки и сопровождения в течение всего жизненного цикла ПС и ориентирована на коллективную разработку больших программных систем с продолжительным жизненным циклом. Обязательными свойствами ее являются комплексность, ориентированность на коллективную разработку и интегрированность. Кроме того, она или обладает технологической определенностью или получает это свойство в процессе расширения (настройки). Значение признака языковой ориентированности может быть различным, что используется для дальнейшей классификации этих систем.

12. Что такое API, для чего нужно, напишите примеры

API (Application programming interface) — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это». Т.е. другими словами, API – это представление(view) предоставляемого функционала. Специально разработана для запросов в определенный удаленный сервис.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована. На рисунке 6. Показана схема обращения с API

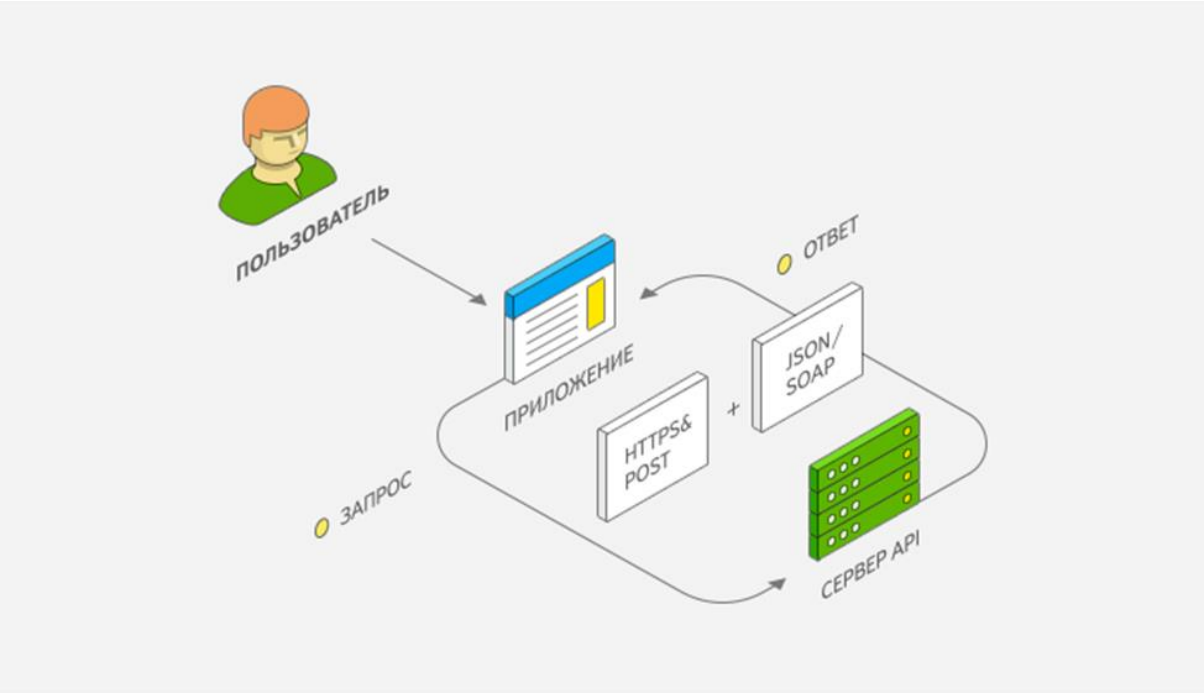


Рис.6.

Если программу (модуль, библиотеку) рассматривать как чёрный ящик, то API — это множество «ручек», которые доступны пользователю данного ящика и которые он может вертеть и дёргать.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

По такому принципу построены протоколы передачи данных по Интернету. Стандартный стек протоколов (сетевая модель OSI) содержит 7 уровней (от физического уровня передачи бит до уровня протоколов приложений, подобных протоколам HTTP и IMAP). Каждый уровень пользуется функциональностью предыдущего («нижележащего») уровня передачи данных и, в свою очередь, предоставляет нужную функциональность следующему («вышележащему») уровню.

Важно заметить, что понятие протокола близко по смыслу к понятию API. И то, и другое является абстракцией функциональности, только в первом случае речь идёт о передаче данных, а во втором — о взаимодействии приложений.

API библиотеки функций и классов включает в себя описание сигнатур и семантики функций.

Сигнатура функции — часть общего объявления функции, позволяющая средствам трансляции идентифицировать функцию среди других. В различных языках программирования существуют разные представления о сигнатуре функции, что также тесно связано с возможностями перегрузки функций в этих языках.

Иногда различают *сигнатуру вызова* и *сигнатуру реализации* функции. Сигнатура вызова обычно составляется по синтаксической конструкции вызова функции с учётом сигнатуры области видимости данной функции, имени функции, последовательности фактических типов аргументов в вызове и типе результата. В сигнатуре реализации обычно участвуют некоторые элементы из синтаксической конструкции объявления функции: спецификатор области видимости функции, её имя и последовательность формальных типов аргументов.

Например, в языке программирования C++ простая функция однозначно опознаётся компилятором по её имени и последовательности типов её аргументов, что составляет сигнатуру функции в этом языке. Если функция является методом некоторого класса, то в сигнатуре будет участвовать и имя класса.

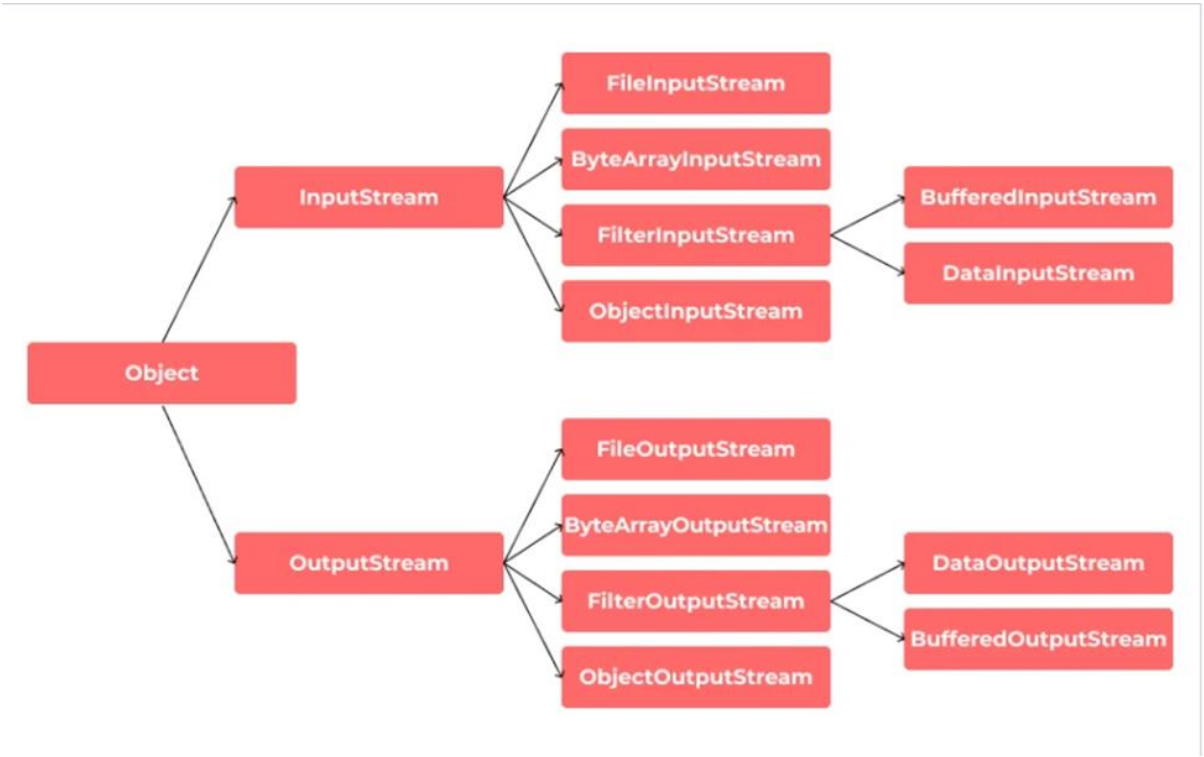
В языке программирования Java сигнатуру метода составляет его имя и последовательность типов параметров; тип возвращаемого значения в сигнатуре не участвует.

Семантика функции — это описание того, что данная функция делает. Семантика функции включает в себя описание того, что является результатом вычисления функции, как и от чего этот результат зависит. Обычно результат выполнения зависит только от значений аргументов функции, но в некоторых модулях есть понятие состояния. Тогда результат функции может зависеть от этого состояния, и, кроме того, результатом может стать изменение состояния. Логика этих зависимостей и изменений относится к семантике функции. Полным описанием семантики функций является исполняемый код функции или математическое определение функции.

API используется практически в любых современных сервисах. API для доступа почтовый программы к почтовому сервису, различным онлайн картам, сервисы по предоставлению погоды и многие другие. Рассмотрим пример использования API.

13. Что такое Java IO

IO (Input & Output) API — это Java API, которое облегчает разработчикам работу с потоками. Скажем, мы получили какие-то данные (например, фамилия, имя и отчество) и нам нужно записать их в файл — в этот момент и приходит время использовать **java.io**.



Но у **Java IO** есть свои недостатки:

1. Блокирующий доступ для ввода/вывода данных. Проблема состоит в том, что когда разработчик пытается прочитать файл или записать что-то в него, используя **Java IO**, он блокирует файл и доступ к нему до момента окончания выполнения своей задачи.
2. Отсутствует поддержка виртуальных файловых систем.
3. Нет поддержки ссылок.
4. Очень большое количество checked исключений.

Работа с файлами всегда несет за собой работу с исключениями: например, попытка создать новый файл, который уже существует, вызовет **IOException**. В данном случае работа приложения должна быть продолжена и пользователь должен получить уведомление о том, по какой причине файл не может быть создан.

```
try {
        File.createTempFile("prefix", "");
} catch (IOException e) {
        // Handle IOException
```

```
}
/**
* Creates an empty file in the default temporary-file directory
* any exceptions will be ignored. This is typically used in finally blocks.
* @param prefix
* @param suffix
* @throws IOException - If a file could not be created
*/
public static File createTempFile(String prefix, String suffix)
throws IOException {
...
}
```

Метод createTempFile выбрасывает **IOException**, когда файл не может быть создан. Это исключение должно быть обработано соответственно. Если попытаться вызвать этот метод вне блока try-catch, то компилятор выдаст ошибку и предложит нам два варианта исправления: окружить метод блоком try-catch или сделать так, чтобы метод, внутри которого вызывается File.createTempFile, выбрасывал исключение **IOException** (чтобы передать его на верхний уровень для обработки).

14. Что такое Java NIO

Java.**nio** (**NIO** расшифровывается как Non-blocking I/O) - это набор API-интерфейсов языка программирования **Java**, которые предлагают функции для интенсивных операций ввода-вывода. Он был представлен с выпуском **Java** J2SE 1.4 компанией Sun Microsystems в дополнение к существующему стандарту ввода-вывода.

15. Пример кода Java NIO, Java IO. Кто быстрее.

Сначала поговорим о работе с **Java IO**:

Класс InputStream

```
try(FileInputStream fin = new FileInputStream("C:/javarush/file.txt")){
        System.out.printf("File size: %d bytes \n", fin.available());
        int i=-1;
        while((i=fin.read())!=-1){
        System.out.print((char)i);
        }
} catch(IOException ex) {
        System.out.println(ex.getMessage());
}
```

Класс FileInputStream предназначен для считывания данных из файла. Он является наследником класса InputStream и поэтому реализует все его методы. Если файл не может быть открыт, то генерируется исключение **FileNotFoundException**.

Класс OutputStream

```
String text = "Hello world!"; // строка для записи
try(FileOutputStream fos = new FileOutputStream("C:/javarush/file.txt")){
        // переводим нашу строку в байты
        byte[] buffer = text.getBytes();
        fos.write(buffer, 0, buffer.length);
        System.out.println("The file has been written");
} catch(IOException ex){
        System.out.println(ex.getMessage());
```

```
}
```

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream.

Классы Reader и Writer

Класс FileReader позволяет нам читать символьные данные из потоков, а класс FileWriter используется для записи потоков символов. Реализация записи и чтения из файла приведена на слайде:

```
String fileName = "c:/javarush/Example.txt";

// Создание объекта FileWriter

try (FileWriter writer = new FileWriter(fileName)) {

    // Запись содержимого в файл

    writer.write("Это простой пример,\n в котором мы осуществляем\n с помощью языка Java\n запись в файл\n и чтение из файла\n");

    writer.flush();

} catch (IOException e) {

    e.printStackTrace();

}

// Создание объекта FileReader

try (FileReader fr = new FileReader(fileName)) {

    char[] a = new char[200];// Количество символов, которое будем считывать

    fr.read(a);  // Чтение содержимого в массив

    for (char c : a) {

    System.out.print(c); // Вывод символов один за другими

    }
```

```
} catch (IOException e) {
        e.printStackTrace();
}
```

Теперь немного поговорим о том, что нового появилось в **Java NIO2**.

Path

Path представляет из себя путь в файловой системе. Он содержит имя файла и список каталогов, определяющих путь к нему.

```
Path relative = Paths.get("Main.java");
System.out.println("Файл: " + relative);
//получение файловой системы
System.out.println(relative.getFileSystem());
```

Paths — это совсем простой класс с единственным статическим методом get(). Его создали исключительно для того, чтобы из переданной строки или URI получить объект типа Path.

```
Path path = Paths.get("c:\\data\\file.txt");
```

Files

Files — это утилитный класс, с помощью которого можно напрямую получать размер файла, копировать их, и не только.

```
Path path = Paths.get("files/file.txt");
boolean pathExists = Files.exists(path);
```

FileSystem

FileSystem предоставляет интерфейс к файловой системе. Файловая система работает как фабрика для создания различных объектов (Path, PathMatcher, Files). Этот объект помогает получить доступ к файлам и другим объектам в файловой системе.

```
try {
        FileSystem filesystem = FileSystems.getDefault();
        for (Path rootdir : filesystem.getRootDirectories()) {
        System.out.println(rootdir.toString());
        }
  } catch (Exception e) {
        e.printStackTrace();
  }
```

Тест производительности

Для этого теста возьмем два файла. Первый — маленький файл с текстом, а второй — большой видеоролик.
Создаем файл и добавляем в него немного слов и символов:

```
% touch text.txt
```

Наш файл по итогу занимает в памяти 42 байта:
Теперь напишем код, который будет копировать наш файл из одной папки в другую. Проверим его работу на маленьком и большом файлах, и тем самым сравним скорость работы **IO**, **NIO** и **NIO2**.
Код для копирования, написанный на **Java IO**:

```
public static void main(String[] args) {
        long currentMills = System.currentTimeMillis();
        long startMills = currentMills;
        File src = new File("/Users/IdeaProjects/testFolder/text.txt");
        File dst = new File("/Users/IdeaProjects/testFolder/text1.txt");
        copyFileByIO(src, dst);
        currentMills = System.currentTimeMillis();
```

```
        System.out.println("Время выполнения в миллисекундах: " +
(currentMills - startMills));
        }

        public static void copyFileByIO(File src, File dst){
        try(InputStream inputStream = new FileInputStream(src);
        OutputStream outputStream = new FileOutputStream(dst)){

        byte[] buffer = new byte[1024];
        int length;
        // Читаем данные в байтовый массив, а затем выводим в OutStream
        while((length = inputStream.read(buffer)) > 0){
                outputStream.write(buffer, 0, length);
        }
        } catch (IOException e) {
        e.printStackTrace();
        }
        }
```

И код для **Java NIO**:

```
public static void main(String[] args) {
        long currentMills = System.currentTimeMillis();
        long startMills = currentMills;
```

```
File src = new File("/Users/IdeaProjects/testFolder/text.txt");

File dst = new File("/Users/IdeaProjects/testFolder/text2.txt");

// копия nio

copyFileByChannel(src, dst);

currentMills = System.currentTimeMillis();

System.out.println("Время выполнения в миллисекундах: " +
(currentMills - startMills));

}


public static void copyFileByChannel(File src, File dst){

// 1. Получаем FileChannel исходного файла и целевого файла

try(FileChannel srcFileChannel  = new FileInputStream(src).getChannel();

FileChannel dstFileChannel = new FileOutputStream(dst).getChannel()){

// 2. Размер текущего FileChannel

long count = srcFileChannel.size();

while(count                                    >                                    0){
/**==========================================================
==========

* 3. Записать байты из FileChannel исходного файла в целевой
FileChannel

* 1. srcFileChannel.position (): начальная позиция в исходном
файле не может быть отрицательной

* 2. count: максимальное количество переданных байтов, не
может быть отрицательным
```

```
            * 3. dstFileChannel: целевой файл
        *==========================================================
==========*/

            long transferred =
srcFileChannel.transferTo(srcFileChannel.position(),

            count, dstFileChannel);

            // 4. После завершения переноса измените положение
исходного файла на новое место

            srcFileChannel.position(srcFileChannel.position() + transferred);

            // 5. Рассчитаем, сколько байтов осталось перенести

            count -= transferred;

        }


        } catch (FileNotFoundException e) {

        e.printStackTrace();

        } catch (IOException e) {

        e.printStackTrace();

        }

        }
```

Код для **Java NIO2**:

```
public static void main(String[] args) {
  long currentMills = System.currentTimeMillis();
  long startMills = currentMills;
```

```
Path sourceDirectory = Paths.get("/Users/IdeaProjects/testFolder/test.txt");

Path targetDirectory = Paths.get("/Users/IdeaProjects/testFolder/test3.txt");

Files.copy(sourceDirectory, targetDirectory);



currentMills = System.currentTimeMillis();

System.out.println("Время выполнения в миллисекундах: " + (currentMills - startMills));

}
```

Начнем с маленького файла.
Время выполнения с помощью **Java IO** в среднем было 1 миллисекунду. Запуская тест несколько раз, получаем результат от 0 до 2 миллисекунд.

Время выполнения в миллисекундах: 1
Время выполнения с помощью **Java NIO** гораздо больше. Среднее время — 11 миллисекунд. Результаты были от 9 до 16. Это связанно с тем, что **Java IO** работает не так, как наша операционная система. **IO** перемещает и обрабатывает файлы один за другим, в то время как операционная система отправляет данные в одном большом виде. А **NIO** показал плохие результаты из-за того, что он ориентирован на буфер, а не на поток, как **IO**.

Время выполнения в миллисекундах: 12
И так же тест для **Java NIO2**. **NIO2** имеет улучшенное управление с файлами по сравнению с **Java NIO**. Из-за этого результаты обновленной библиотеки так отличаются:

Время выполнения в миллисекундах: 3
Протестируем большой файл, видео на 521 МБ. Задача будет точно такой же: скопировать в другую папку.
Результаты с **Java IO**:

Время выполнения в миллисекундах: 1866
А вот результат **Java NIO**:

Время выполнения в миллисекундах: 205

Java NIO справился с файлом в 9 раз быстрее при первом тесте. Повторные тесты показывали примерно такие же результаты.
тест на **Java NIO2**:

Время выполнения в миллисекундах: 360
Почему же такой результат? Просто потому что нет особого смысла сравнивать производительность между ними, так как они служат разным целям. **NIO** представляет собой более абстрактный низкоуровневый ввод-вывод данных, а **NIO2** ориентирован на управление файлами.

Итоги

Java NIO существенно повышает эффективность работы с файлами за счет использования внутри блоков. Еще один плюс состоит в том, что библиотека **NIO** разбита на две части: одна для работы с файлами, вторая — для работы в сети.
Новый API, который используется в **Java NIO2** для работы с файлами, предлагает множество полезных функций:

- гораздо более полезную адресацию файловой системы с помощью Path,
- значительно улучшенную работу с ZIP-файлами с использованием пользовательского поставщика файловой системы,
- доступ к специальным атрибутам файла,
- множество удобных методов, например, чтение всего файла с помощью одной команды, копирование файла с помощью одной команда и т. д.

Все это связано с файлом и файловой системой, и все довольно высокого уровня.
В современных реалиях **Java NIO** занимает около 80-90% работы с файлами, хотя доля **Java IO** тоже еще существенна.

16. Отличия между Java IO и Java NIO

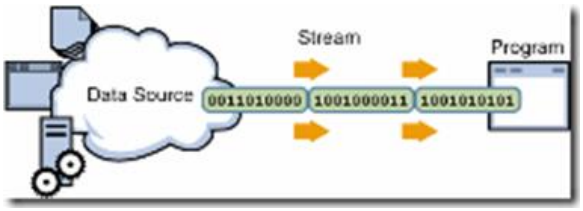
IO	NIO
Потокоориентированный	Буфер-ориентированный
Блокирующий (синхронный) ввод/вывод	Неблокирующий (асинхронный) ввод/вывод
	Селекторы

17. Потокоориентированный ввод/вывод 18. Буфер-ориентированный ввод/вывод

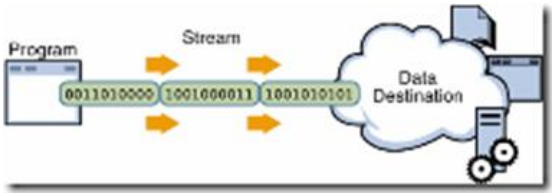
Основное отличие между двумя подходами к организации ввода/вывода в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным. Разберем подробней.

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если надо произвести подобные манипуляции, придется сначала кэшировать данные в буфере.

Потокоориентированный ввод:



Буферориентированный вывод:



19. Блокирующий и неблокирующий ввод/вывод

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.

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.

Каналы – это логические (не физические) порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе, данные из канала помещаются в предоставленный вами буфер.

Каналы напоминают трубопроводы, по которым эффективно транспортируются данные между буферами байтов и сущностями по ту сторону каналов. Каналы – это шлюзы, которые позволяют получить доступ к сервисам ввода/вывода операционной системы с минимальными накладными расходами, а буферы – внутренние конечные точки этих шлюзов, используемые для передачи и приема данных.

Тоже самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.

Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний. Наиболее

частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

20. API обращений к классам ввода/вывода

API обращений к классам ввода/вывода

Естественно, использование Java NIO серьезно отличается от использования Java IO. Так как, вместо чтения данных байт за байтом с использованием, например InputStream, данные для начала должны быть считаны в буфер и браться для обработки уже оттуда.

Обработка данных

Обработка данных при использовании Java NIO тоже отличается.

При использовании Java IO данные читаются байт за байтом с InputStream или Reader. Представьте, что идет считывание строк текстовой информации:

```
Name: Anna

Age: 25

Email: anna@mailserver.com

Phone: 1234567890
```

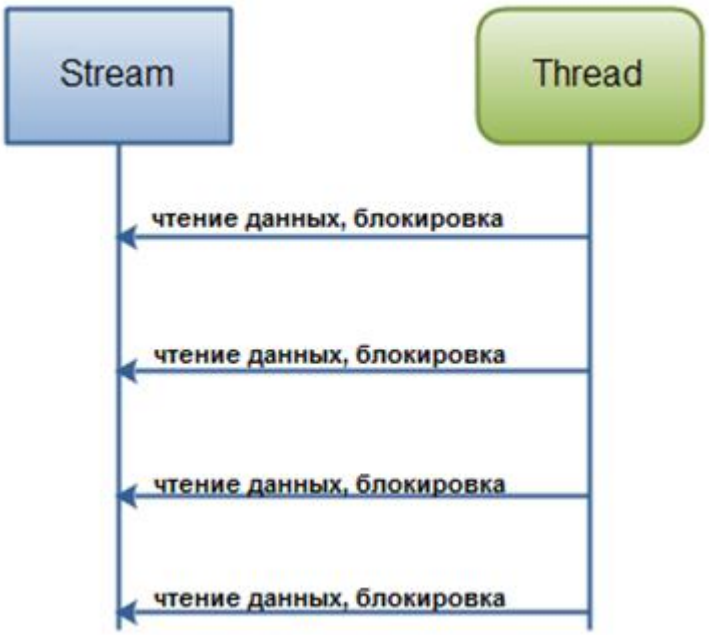
Этот поток строк текста может обрабатываться следующим образом:

```
InputStream input = ... ;

BufferedReader reader = new BufferedReader(new InputStreamReader(input));

String nameLine   = reader.readLine();

String ageLine    = reader.readLine();
```

```
String emailLine  = reader.readLine();

String phoneLine  = reader.readLine();
```

Состояние процесса обработки зависит от того, насколько далеко продвинулось выполнение программы. Когда первый метод readLine() возвращает результат выполнения– целая строка текста была считана. Метод является блокирующим и действие блокировки продолжается до тех пор, пока вся строка не будет считана. Данная строка содержит имя. Подобно этому, когда метод вызывается во второй раз, получим возраст.

Прогресс в выполнении программы достигается только тогда, когда доступны новые данные для чтения, и для каждого шага вы знаете что это за данные. Когда поток выполнения достигает прогресса в считывании определенной части данных, поток ввода (в большинстве случаев) уже не двигает данные назад. Данный принцип хорошо демонстрирует следующая схема:



Имплементация с использованием Java IO будет выглядеть несколько иначе:

```
ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);
```

Обратите внимание на вторую строчку кода, в которой происходит считывание байтов из канала в ByteBuffer. Когда возвращается результат выполнения данного метода, нельзя быть уверенным, что все необходимые вам данные находятся внутри буфера. Все, что известно, так это то, что буфер содержит некоторые байты. Это немного усложняет процесс обработки.

После первого вызова метода read(buffer), в буфер было считано только половину строки. Например, “Name: An”. Придется ждать пока, по крайней мере, одна полная строка текста не будет считана в буфер. Единственный вариант узнать достаточно ли данных для корректной обработки содержит буфер, это посмотреть на данные, содержащиеся внутри буфера. В результате придется по нескольку раз проверять данные в буфере, пока они не станут доступными для корректной обработки. Это неэффективно и может негативно сказаться на дизайне программы. Например:

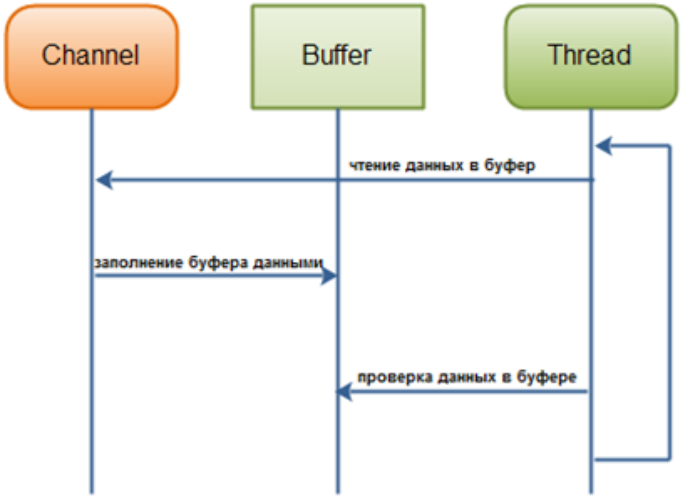
```
ByteBuffer buffer = ByteBuffer.allocate(48);

        int bytesRead = inChannel.read(buffer);

                while(! bufferFull(bytesRead) ) {

                        bytesRead = inChannel.read(buffer);

                }
```

Метод bufferFull() должен следить за тем, сколько данных считано в буфер и возвращать true или false, в зависимости от того, заполнен буфер или нет. Другими словами, если буфер готов к обработке, то он считается заполненным.

Также метод bufferFull() должен оставлять буфер в неизмененном состоянии, поскольку в противном случае следующая порция считанных данных может быть записана в неправильное место.

Если буфер заполнен, данные из него могут быть обработаны. Если он не заполнен вы все же будете иметь возможность обработать уже имеющиеся в нем данные, если это имеет смысл в вашем конкретном случае. В большинстве случаев – это бессмысленно.

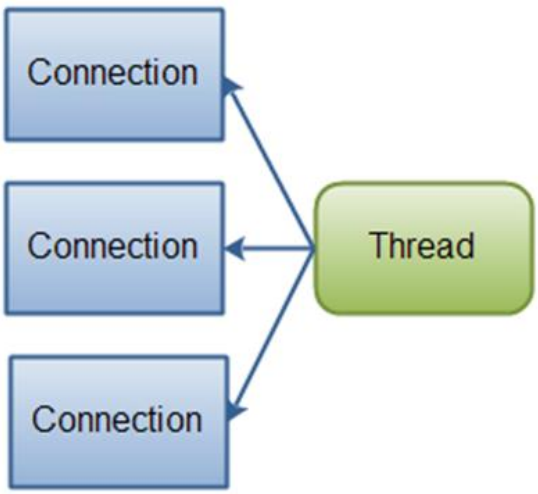
Схема демонстрирует процесс определения готовности данных в буфере для корректной обработки:



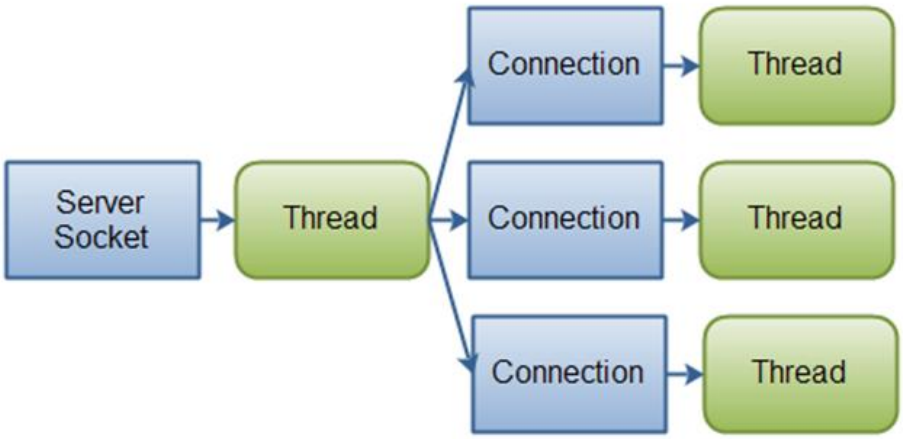
Итоги

Java NIO позволяет управлять несколькими каналами (сетевыми соединениями или файлами) используя минимальное число потоков выполнения. Однако ценой такого подхода является более сложный, чем при использовании блокирующих потоков, парсинг данных.

Если необходимо управлять тысячами открытых соединений одновременно, причем каждое из них передает лишь незначительный объем данных, выбор Java NIO для приложения может дать преимущество. Дизайн такого типа схематически изображен на рисунке сверху:



Если присутствует меньшее количество соединений, по которым передаются большие объемы данных, то лучшим выбором станет классический дизайн системы ввода/вывода (нижний рисунок):



21. Java NIO Buffer, что это, для чего нужно, пример

Буфер, по сути, является средой для хранения данных, пока поток не прочитает данные из него и не запросит новые данные. Основным шагом к чтению данных из источника ввода является чтение данных в буфер.

Буфер — это фиксированная часть памяти, используемая для хранения этих данных перед их чтением в канал. Буфер обеспечивает предварительную загрузку данных определенного размера для ускорения чтения файлов, входных данных и потоков данных. Размер буфера настраивается в блоках от 2 до степени n.
Данные считываются в буфер для последующей обработки. Разработчик может двигаться по буферу вперед и назад, что дает немного больше гибкости при обработке данных. В то же время нужно проверять, содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер не уничтожить еще не обработанные данные, находящиеся там.

```
ByteBuffer buf = ByteBuffer.allocate (2048);

int bytesRead = channel.read(buf);
```

```
buf.flip(); // меняем режим на чтение

while (buf.hasRemaining()) {

        byte data = buf.get(); // есть методы для примитивов

}

buf.clear(); // очистили и можно переиспользовать
```

В коде создается буфер из 2048 байтов. Размер буфера является обязательным для указания. 3-я строка выделяет область памяти 2048 байтов для буфера **buf.** Это гарантирует, что необходимая память предварительно выделена для буфера. Процесс чтения и записи в буфер необходимо понять, прежде чем перейдем к их использованию для чтения и записи канала. Затем буфер используется для чтения данных для канала с помощью оператора channel.read(buf) . При выполнении этого оператора буфер теперь будет содержать до 2048 байт данных как доступные. Чтобы начать читать данные, используется простое утверждение buf.get() .

Буфер читает и толкает первый байт влево. Буфер является распределением памяти типа «Последний пришел — первый вышел». Следовательно, когда надо прочитать файл с помощью буфера, необходимо перевернуть его, прежде чем прочитать файл. Без переворота данные вышли бы в обратном порядке. Чтобы перевернуть буфер, необходимо выполнить простую строку кода: buf.flip();
Как только данные были прочитаны в буфер, пришло время фактически получить данные, которые были прочитаны. Для чтения данных используется функция buf.get() . Этот вызов функции читает один байт / символ / пакет при каждом вызове в зависимости от типа буфера. После того, как прочитали доступные данные, также необходимо очистить буфер перед следующим чтением. Очистка необходима, чтобы освободить место для чтения дополнительных данных. Чтобы очистить буфер, есть два возможных способа — очистить буфер или сжать буфер.

Чтобы очистить буфер, выполните команду buf.clear() . Чтобы buf.compact() буфер, используйте команду buf.compact() . Обе эти команды в конечном итоге делают одно и то же. Однако compact() очищает только те данные, которые были прочитаны с использованием вызова функции

buf.get() . Таким образом, он используется, когда нам нужно продолжать оптимизировать объем памяти, используемой буфером.

Пометить и сбросить буфер

Во время процесса чтения часто возникают ситуации, когда необходимо повторно прочитать данные с определенной позиции. В обычном сценарии, когда данные получаются из буфера, они считаются пропавшими. Тем не менее, можно пометить буфер по определенному индексу, чтобы можно было снова прочитать его из определенной позиции. Код демонстрирует, как это делается.

```
buffer.mark();

char x = buffer.get();

char y = buffer.get();

char z = buffer.get();

char a = buffer.get();

//Do something with above data
buffer.reset();  //set position back to mark
```

Основными приложениями отметки и сброса являются многократный анализ данных, многократное повторение информации, отправка команды на определенное количество раз и многое другое.

22. Типы Java NIO Buffer которые могут использоваться в зависимости от типа ввода

Различные типы буферов, которые могут использоваться в зависимости от типа ввода:

- **ByteBuffer:** используется для чтения потоков символов или файлов в **байтовом выражении**
- **CharBuffer:** используется для чтения символов в полном наборе ASCII
- **DoubleBuffer:** используется специально для двойных значений данных, таких как показания датчиков

- **FloatBuffer:** используется для чтения постоянных потоков данных для таких целей, как аналитика
- **LongBuffer:** используется для чтения значений типа данных long
- **IntBuffer:** используется для чтения целочисленных значений для результатов или результатов.
- **ShortBuffer:** используется для чтения коротких целочисленных значений

Каждый буфер предназначен для его конкретного использования. Буферы, обычно используемые для файлов, являются **ByteBuffer** и **CharBuffer.**
Основные свойства буфера:

Основные атрибуты	
capacity	Размер буфера, который является длиной массива.
position	Начальная позиция для работы с данными.
limit	Операционный лимит. Для операций чтения предел — это объем данных, который можно поместить в оперативный режим, а для операций записи — предел емкости или доступная для записи квота, указанная ниже.
mark	Индекс значения, до которого будет сброшен параметр position при вызове метода reset().

Во время записи в буфер позиция буфера — это позиция, в которой записывается текущий байт. Во время процесса чтения из буфера позиция буфера — это позиция, из которой читается байт. Положение буфера продолжает динамически меняться, когда мы продолжаем чтение или запись.

Во время записи в буфер предел буфера — это максимальный размер данных, которые могут быть записаны в буфер. По существу, ограничение буфера и емкость буфера являются синонимами во время записи в буфер. Однако во время чтения из буфера предел буфера — это количество доступных байтов для чтения из буфера. Следовательно, ограничение буфера продолжает уменьшаться по мере выталкивания байтов.

Емкость буфера — это максимальные данные, которые можно записать в буфер или прочитать из буфера в любой момент времени. Таким образом, размер 48, выделенный выше, также называется буферной емкостью.

23. Канальные передачи

Передача канала, как следует из названия, представляет собой процесс передачи данных из одного канала в другой. Передача канала может осуществляться из определенной позиции буфера канала. Однако при значении позиции, равном нулю, можно копировать или реплицировать полный источник ввода в указанное место назначения вывода. Например, установление канала между ключевым словом и текстовым редактором позволит непрерывно переносить ввод с клавиатуры в текстовый редактор. Чтобы облегчить передачу канала, Java NIO оснащен двумя функциями, а именно — transferFrom() и transferTo() . Чтобы лучше понять эти функции, будем использовать файл **data.txt,** созданный ранее, в качестве источника ввода. Перенесем данные из этого файла в новый файл **output.txt** . Код на слайде делает то же самое, используя вызов метода TransferFrom transferFrom() .

ChannelTransfer.java

```
import java.io.RandomAccessFile;
import java.nio.channels.FileChannel;
public class ChannelTransfer {
    public static void main(String[] args) {
        try {
            RandomAccessFile copyFrom = new RandomAccessFile("src/data.txt",
"rw");
            FileChannel fromChannel = copyFrom.getChannel();
            RandomAccessFile copyTo = new RandomAccessFile("src/output.txt",
"rw");
            FileChannel toChannel = copyTo.getChannel();
            long count = fromChannel.size();
            toChannel.transferFrom(fromChannel, 0, count);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Как видно из приведенного кода, канал fromChannel используется для чтения данных из data.txt. toChannel используется для получения данных из fromChannel начиная с позиции 0. Важно отметить, что весь файл копируется с использованием FileChannel . Однако в некоторых реализациях SocketChannel это может быть не так. В таком случае копируются только те данные, которые доступны для чтения в буфере во время передачи. Такое поведение обусловлено динамической природой реализаций SocketChannel .

Реализация transferTo довольно похожа. Единственное изменение, которое потребуется, — это то, что вызов метода будет выполняться с использованием исходного объекта, а объект канала назначения будет аргументом в вызове метода.

24. Java NIO Channel

Каналы являются основной средой для неблокирующего ввода-вывода. Каналы аналогичны потокам, доступным для блокировки ввода-вывода. Эти каналы поддерживают данные по сетям, а также файлы ввода-вывода данных. Каналы читают данные из буферов по мере необходимости. Буферы хранят данные до тех пор, пока они не будут прочитаны.
Каналы имеют несколько реализаций в зависимости от данных, которые нужно прочитать.
В отличие от потоков, которые используются в **Java IO**, NIO Channel является двусторонним, то есть может и считывать, и записывать. Канал **Java NIO** поддерживает асинхронный поток данных как в режиме блокировки, так и в режиме без блокировки.

```
RandomAccessFile aFile = new RandomAccessFile("C:/javarush/file.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(100);
int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {
  System.out.println("Read " + bytesRead);
  buf.flip();
                while(buf.hasRemaining()){
                    System.out.print((char) buf.get());
                }
  buf.clear();
  bytesRead = inChannel.read(buf);
```

}

aFile.close();

Здесь реализован FileChannel. Для чтения данных из файла используется файловый канал. Объект файлового канала может быть создан только вызовом метода getChannel() для файлового объекта, поскольку нельзя напрямую создать объект файлового канала.

Кроме FileChannel есть и другие реализации каналов:

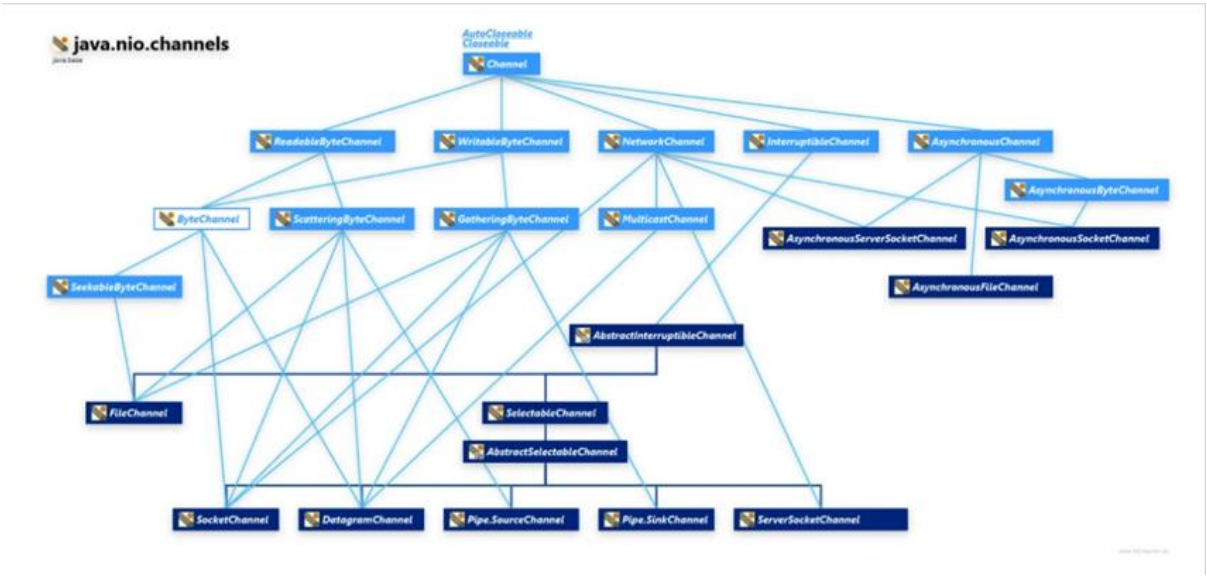
· FileChannel — используется для чтения и записи данных из файлов и в файлы

· DatagramChannel — используется для обмена данными по сети с использованием пакетов UDP.

· SocketChannel — канал TCP для обмена данными через сокеты TCP

● ServerSocketChannel реализация, аналогичная веб-серверу, который прослушивает запросы через определенный порт TCP. Создает новый экземпляр SocketChannel для каждого нового соединения.

Как можно понять из названий каналов, они также охватывают сетевой IO-трафик UDP + TCP в дополнение к IO-файлу. В отличие от потоков, которые могут либо читать, либо записывать в определенный момент, один и тот же канал может беспрепятственно читать и записывать ресурсы. Каналы поддерживают асинхронное чтение и запись, что обеспечивает чтение данных без ущерба для выполнения кода. Буферы, рассмотренные выше, поддерживают эту асинхронную работу каналов.



FileChannel нельзя переключить в неблокирующий режим. 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. В это же время SelectableChannel и его реализации могут устанавливаться в неблокирующем режиме с помощью метода connect().

Java NIO по своей природе поддерживает рассеяние и сбор данных для чтения и записи данных в несколько буферов. Java NIO достаточно умен, чтобы иметь возможность управлять чтением и записью в нескольких буферах.

Разброс Java NIO используется для разбивки чтения из канала на несколько буферов. Реализация кода довольно проста. Все, что нужно сделать, это добавить массив буферов в качестве аргумента для чтения. Фрагмент кода показан на слайде:

```
ByteBuffer buffer1 = ByteBuffer.allocate(128);

ByteBuffer buffer2  = ByteBuffer.allocate(128);

ByteBuffer[] buffers = {buffer1,buffer2};

channel.read(buffers);
```

В приведенном коде создается два буфера по 128 байт каждый. Обратите внимание, что массив создается из двух буферов. Этот массив затем передается в качестве аргумента в канал для чтения. Канал считывает данные в первый буфер, пока не будет достигнута емкость буфера. Как только емкость буфера достигнута, канал автоматически переключается на следующий буфер. Таким образом, чтение канала рассеивается без какого-либо влияния на поток.

Сбор Java NIO также работает аналогичным образом. Данные, считанные в несколько буферов, также могут быть собраны и записаны в один канал.

Информация записывается в канал, начиная с первого буфера. Канал автоматически переключается на следующий буфер при достижении предела для первого. Во время этой записи переворот не происходит. Если нужно перевернуть буфер перед записью, это нужно сделать перед назначением их в массив.

25. Что такое Direct буфер, для чего он нужен, пример

Java NIO поддерживает тип ByteBuffer, обычно известный как direct(прямой) буфер. Direct буфер по существу могут использоваться как любой другой ByteBuffer (и реализованы как подкласс ByteBuffer), но имеют свойство, заключающееся в том, что их базовая память **выделяется вне кучи Java** . В частности, Direct буферы обладают следующими свойствами:

- после выделения их адрес памяти фиксируется на время жизни буфера;
- поскольку их адрес фиксирован, ядро может безопасно обращаться к ним напрямую, и, следовательно, прямые буферы могут использоваться более эффективно в операциях ввода-вывода;
- в некоторых случаях доступ к ним из Java может быть более эффективным (потенциально меньше накладных расходов при поиске адреса памяти и/или других служебных операций, необходимых перед доступом к объекту Java) ;
- через Java Native Interface вы можете фактически установить адрес произвольно, если это необходимо (например, для доступа к оборудованию по определенному адресу или для самостоятельного выполнения выделения).

На практике в текущих версиях Hotspot память выделяется с помощью malloc(), хотя это может отличаться в других виртуальных машинах или в будущей версии Hotspot.

Чтобы создать прямой буфер из Java, надо вызывать:

ByteBuffer directBuf = **ByteBuffer.allocateDirect(noBytes);**

Затем возвращенный ByteBuffer можно использовать практически как любой другой байтовый буфер. Например, будут работать все различные методы get() и put , а также методы для создания представлений буфера. Одна вещь, которую *нельзя* сделать, по крайней мере, в Hotspot, — это вызвать array() — в основе прямого буфера лежит не массив Java, а просто «сырой» участок памяти. (Хотя строго, согласно Javadoc, реализации

фактически могут свободно реализовывать прямой буфер с резервным массивом, если они могут найти способ сделать это...)

Нет явного метода, который можно вызвать из Java для уничтожения или освобождения Direct буфера. Когда выделяется прямой буфер, виртуальная машина эффективно регистрирует метод «очистки» в сборщике мусора, который должен быть вызван в какой-то момент, когда сам объект ByteBuffer больше не доступен, чтобы освободить базовую память, зарезервированную для этого буфера.

Обычно это «автоматическое» освобождение работает достаточно хорошо. Но важно иметь в виду, что нет *непосредственной* связи между последним обращением к прямому буферу и моментом фактического освобождения памяти.

Когда выделяется прямой буфер, как правило, это влияет на другие части процесса, которые могли бы использовать malloc() для выделения памяти, особенно на память, выделенную из любых собственных библиотек, используемых программой.

В некоторых случаях может потребоваться ограничить объем памяти, которую Java-приложение может использовать для выделения прямых буферов. Для этого надо установить для свойства sun.nio.MaxDirectMemorySize требуемый предел в байтах при запуске виртуальной машины. Попытка выделить больше этого лимита безопасно выдаст OutOfMemoryError . Таким образом, можно гарантировать, что Java-приложение "изящно" откажет, если оно попытается выделить слишком много памяти для прямых буферов, вместо того, чтобы иметь эффект домино для другого машинного кода, который, возможно, не сможет так изящно завершиться ошибкой, если malloc() терпит неудачу.

В целом, прямые буферы лучше всего подходят для случаев, когда:

- создается относительно ограниченное количество буферов, которые будут жить относительно долго;
- производительность имеет решающее значение, использование прямого буфера даст прирост производительности (типичными случаями являются случаи, когда используется буфер для ввода-вывода).

26. Что такое MappedFileBuffer, для чего он нужен, пример

Когда надо загрузить область файла, можно загрузить ее в определенную область памяти, к которой можно получить доступ позже.

Когда заранее известно, что нужно будет прочитать содержимое файла несколько раз, рекомендуется оптимизировать дорогостоящий процесс, например, сохранив это содержимое в памяти. Благодаря этому последующие поиски этой части файла будут идти только в основную память без необходимости загружать данные с диска, что существенно сократит задержку. Одна нужно быть осторожным при использовании MappedByteBuffer , – это когда идет работа с очень большими файлами с диска – нужно убедиться, что файл поместится в память . В противном случае можно заполнить всю память и, как следствие, столкнуться с общим исключением OutOfMemoryException. Этого можно избежать, загрузив только часть файла, например, на основе шаблонов использования.

Чтение

Чтобы прочитать файл, например: есть файл под названием fileToRead.txt со следующим содержанием:

Тут написано содержание файла

Файл находится в каталоге /resource , поэтому можно загрузить его с помощью функции ***getFileURIFromResources***:

```
Path getFileURIFromResources(String fileName) throws Exception {
    ClassLoader classLoader = getClass().getClassLoader();
        return Paths.get(classLoader.getResource(fileName).getPath());
}
```

Чтобы создать MappedByteBuffer из файла, сначала нужно создать FileChannel из него. После того, как канал создан, можно вызвать метод map() на нем, передавая в режиме Map, позицию, с которой хотим читать, и параметр size , который указывает, сколько байтов необходимо:

```
CharBuffer charBuffer = null;
Path pathToRead = getFileURIFromResources("fileToRead.txt");
try (FileChannel fileChannel (FileChannel) Files.newByteChannel(
  pathToRead, EnumSet.of(StandardOpenOption.READ))) {
      MappedByteBuffer mappedByteBuffer = fileChannel
      .map(FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());
      if (mappedByteBuffer != null) {
      charBuffer = Charset.forName("UTF-8").decode(mappedByteBuffer);
      }}
```

Как только сопоставили файл с буфером памяти, можно прочитать данные из него в буфер Char. **Содержимое файла, которое читаем при вызове метода decode(),передающего MappedByteBuffer, чтение происходит из памяти, а не с диска.** Поэтому это чтение будет очень быстрым.

Содержимое, которое читаем из файла, является фактическим содержимым fileToRead.txt файл:

```
assertNotNull(charBuffer);
assertEquals(
  charBuffer.toString(), " Тут написано содержание файла");
```

Каждое последующее чтение из MappedByteBuffer будет очень быстрым, поскольку содержимое файла отображается в памяти, и чтение выполняется без необходимости поиска данных с диска.

Запись

Допустим, надо записать какой-то контент в файл fileToWriteTo.txt с помощью MappedByteBuffer API. Для этого нужно открыть файловый канал и вызвать на нем метод map () , передав его в файловый канал.MapMode.READ_WRITE.

Затем можно сохранить содержимое буфера Char в файл с помощью метода put() из MappedByteBuffer:

```
CharBuffer charBuffer = CharBuffer.wrap("Запись в файл");
```

```
Path pathToWrite = getFileURIFromResources("fileToWriteTo.txt");
try (FileChannel fileChannel = (FileChannel) Files
  .newByteChannel(pathToWrite, EnumSet.of(
        StandardOpenOption.READ,
    StandardOpenOption.WRITE,
    StandardOpenOption.TRUNCATE_EXISTING))) {
    MappedByteBuffer mappedByteBuffer = fileChannel
        .map(FileChannel.MapMode.READ_WRITE, 0, charBuffer.length());
        if (mappedByteBuffer != null) {
       mappedByteBuffer.put(
        Charset.forName("utf-8").encode(charBuffer));
        }
}
```

Фактическое содержимое CharBuffer было записано в файл, прочитав его содержимое:

```
List fileContent = Files.readAllLines(pathToWrite);
assertEquals(fileContent.get(0), " Запись в файл ");
```

Итог:

Это очень эффективный способ чтения содержимого файла несколько раз, так как файл отображается в память, и последующие чтения не нужно каждый раз переносить на диск.

27. Что такое Java NIO Selector, для чего он нужен, пример

Селектор, как следует из названия, используется для выбора канала из нескольких каналов. Селектор в Java NIO особенно полезен, когда вы планируете использовать один поток для параллельного управления несколькими ресурсами. Селектор действует как мост между потоком и открытыми каналами. Селекторы обычно используются, когда ожидается,

что поток имеет низкий трафик, но требует использования нескольких ресурсов. Схематическое изображение того, какую роль могут играть селекторы, изображено ниже.

Селекторы в Java NIO позволяют одному потоку выполнения мониторить несколько каналов ввода. Можно зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.
В Java NIO появилась возможность создать поток, который будет знать, какой канал готов для записи и чтения данных и может обрабатывать этот конкретный канал. Возможность эта реализуется с помощью класса Selector.
Реализация связи каналов и селектора

Создание селектора довольно просто. Приведенный фрагмент кода объясняет, как создать селектор и как зарегистрировать канал в селекторе.

```
Selector selector = Selector.open();

SocketChannel channel = SocketChannel.open();

channel.configureBlocking(false); // неблокирующий режим

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

Приведенный код создает SocketChannel , настраивает его для неблокирования и регистрирует его с помощью селектора. Обратите внимание, что используется SocketChannel . Для селектора требуется канал, который можно настроить как неблокирующий. Следовательно, селектор не может использоваться с FileChannel .

Еще один момент, на который следует обратить внимание — это второй аргумент при регистрации SocketChannel . Аргумент указывает события канала. Создается Selector и связывается с SelectableChannel.

Для использования с селектором канал должен находиться в неблокирующем режиме. Это значит, что нельзя использовать FileChannel с селектором, поскольку FileChannel нельзя переключить в неблокирующий режим. Сокетные каналы будут работать нормально.

В примере SelectionKey — это набор операций, которые можно выполнить с каналом. С помощью клавиши выбора можно узнать состояние канала.

Селектор ожидает события и меняет свой статус по мере запуска событий. События перечислены
Типы SelectionKey:

- SelectionKey.OP_CONNECT — канал, который готов к подключению к серверу.
- SelectionKey.OP_ACCEPT — канал, который готов принимать входящие соединения.
- SelectionKey.OP_READ — канал, который готов к чтению данных.
- SelectionKey.OP_WRITE — канал, который готов к записи данных.

Селектор предоставляет предопределенные функции для проверки возникновения этих событий. Приведенные вызовы методов говорят сами за себя и могут использоваться для отслеживания возникновения этих событий.

Таким образом, селектор чрезвычайно полезен, когда планируется управлять несколькими ресурсами с меньшими потоками.

Чтобы лучше понять концепцию и выгоду от применения селекторов, давайте абстрагируемся от программирования и представим себе железнодорожный вокзал. Вариант без селектора: есть три железнодорожных пути (каналы), на каждый из них в любой момент времени может прибыть поезд (данные из буфера), на каждом пути постоянно ожидает сотрудник вокзала (поток выполнения), задача которого – обслуживание прибывшего поезда. В результате трое сотрудников постоянно находятся на вокзале даже если там вообще нет поездов. Вариант с селектором: ситуация та же, но для каждой платформы есть индикатор, сигнализирующий сотруднику вокзала (поток выполнения) о прибытии поезда. Таким образом на вокзале достаточно присутствия одного сотрудника.

28. Java NIO – FileLock, для чего нужно, пример

Java NIO поддерживает параллелизм и многопоточность, что позволяет ему работать с несколькими потоками, работающими с несколькими файлами одновременно. Но в некоторых случаях требуется, чтобы файл не получал общий доступ ни от одного потока и не был доступен.

Для такого требования NIO снова предоставляет API, известный как FileLock, который используется для обеспечения блокировки всего файла или его части, чтобы файл или его часть не были доступны для общего доступа.

Чтобы применить такую блокировку, надо использовать FileChannel или AsynchronousFileChannel, который предоставляет для этой цели два метода **lock()** и **tryLock().** Предоставляемая блокировка может быть двух типов:

- **Эксклюзивная блокировка** – эксклюзивная блокировка не позволяет другим программам получать перекрывающуюся блокировку любого типа.
- **Общая блокировка** . Общая блокировка не позволяет другим одновременно работающим программам получать перекрывающуюся эксклюзивную блокировку, но позволяет им получать перекрывающиеся общие блокировки.

Методы, используемые для получения блокировки над файлом:

- **lock()** – этот метод FileChannel или AsynchronousFileChannel получает эксклюзивную блокировку для файла, связанного с данным каналом. Типом возврата этого метода является FileLock, который далее используется для мониторинга полученной блокировки.
- **lock(long position, long size, boolean shared)** – этот метод снова является перегруженным методом блокировки и используется для блокировки определенной части файла.
- **tryLock()** – этот метод возвращает FileLock или null, если блокировка не может быть получена, и он пытается получить явно исключительную блокировку для файла этого канала.
- **tryLock(long position, long size, boolean shared)** – этот метод пытается получить блокировку в заданной области файла этого канала, которая может быть эксклюзивной или общего типа.

Методы класса FileLock

- **acquBy ()** – этот метод возвращает канал, для которого была получена блокировка файла.
- **position ()** – этот метод возвращает позицию в файле первого байта заблокированной области. Блокированная область не должна содержаться внутри или даже перекрывать фактический базовый

файл, поэтому значение, возвращаемое этим методом, может превышать текущий размер.

- **size ()** – этот метод возвращает размер заблокированной области в байтах. Блокированная область не должна содержаться внутри или даже перекрывать фактический базовый файл, поэтому значение, возвращаемое этим методом, может превышать текущий размер файла.
- **isShared ()** – Этот метод используется для определения того, является ли блокировка общей или нет.
- **overlaps (длинная позиция, длинный размер)** – этот метод сообщает, перекрывает ли эта блокировка заданный диапазон блокировки.
- **isValid ()** – этот метод сообщает, действительна ли полученная блокировка. Объект блокировки остается действительным до тех пор, пока он не будет освобожден или соответствующий канал файла не будет закрыт, в зависимости от того, что произойдет раньше.
- **release ()** – Освобождает полученную блокировку. Если объект блокировки действителен, то вызов этого метода освобождает блокировку и делает объект недействительным. Если этот объект блокировки недействителен, то вызов этого метода не имеет никакого эффекта.

close () – этот метод вызывает метод release (). Он был добавлен в класс, чтобы его можно было использовать вместе с блочной конструкцией автоматического управления ресурсами

29. Архитектура сервера Netty

Создание сервера

```
ExecutorService bossExec = new OrderedMemoryAwareThreadPoolExecutor(1, 400000000, 2000000000, 60, TimeUnit.SECONDS);

ExecutorService ioExec = new OrderedMemoryAwareThreadPoolExecutor(4 /* число рабочих потоков */, 400000000, 2000000000, 60, TimeUnit.SECONDS);
```

```
ServerBootstrap networkServer = new ServerBootstrap(new NioServerSocketChannelFactory(bossExec, ioExec, 4 /* то же самое число рабочих потоков */));

networkServer.setOption("backlog", 500);

networkServer.setOption("connectTimeoutMillis", 10000);

networkServer.setPipelineFactory(new ServerPipelineFactory());

Channel channel = networkServer.bind(new InetSocketAddress(address, port));
```

Используется *OrderedMemoryAwareThreadPoolExecutor* для выполнения задач Netty. Можно использовать другие Executor'ы, например *Executors.newFixedThreadPool(n)*. Ни в коем случае не используйте *Executors.newCachedThreadPool()*, он создаёт неоправданно много потоков и ни какого выигрыша от Netty почти нет. Использовать более 4 рабочих потоков нет смысла, т.к. они более чем справляются с огромной нагрузкой. Босс-потоки должны быть по одному на каждый слушаемый порт. *Channel,* который возвращает функция bind, а так же *ServerBootsrap* необходимо сохранить, чтобы потом можно было остановить сервер.

PipelineFactory

То, как будут обрабатываться подключения и пакеты клиента, определяет *PipelineFactory*, которая при открытии канала с клиентом создаёт для него pipeline, в котором определены обработчики событий, которые происходят на канале. В нашем случае, это *ServerPipelineFactory*:

```
public class ServerPipelineFactory implements ChannelPipelineFactory {

        @Override

        public ChannelPipeline getPipeline() throws Exception {

       PacketFrameDecoder decoder = new PacketFrameDecoder();

       PacketFrameEncoder encoder = new PacketFrameEncoder();

        return Channels.pipeline(decoder, encoder, new PlayerHandler(decoder, encoder));

        }

}
```

В данном коде *PacketFrameDecoder*, *PacketFrameEncoder* и *PlayerHandler* — обработчки событий, которые мы определяем. Функция *Channels.pipeline()* создаёт новый pipeline с переданными ей обработчиками. События проходят обработчики в том порядке, в котором Вы передали из функции pipeline!

Протокол

Обмен данными происходит с помощью объектов классов, расширяющих класс Packet, в которых определены две функции, *get(ChannelBuffer input)* и *send(ChannelBuffer output)*. Соответственно, первая функция читает необходимые данные из канала, вторая — пишет данные пакета в канал.

```
public abstract class Packet {

        public static Packet read(ChannelBuffer buffer) throws IOException {

        int id = buffer.readUnsignedShort(); // Получаем ID пришедшего пакета, чтобы определить, каким классом его читать

        Packet packet = getPacket(id); // Получаем инстанс пакета с этим ID

                if(packet == null)

                        throw new IOException("Bad packet ID: " + id); // Если произошла ошибка и такого пакета не может быть, генерируем исключение

                packet.get(buffer); // Читаем в пакет данные из буфера

                return packet;

        }

        public statuc Packet write(Packet packet, ChannelBuffer buffer) {

                buffer.writeChar(packet.getId()); // Отправляем ID пакета

                packet.send(buffer); // Отправляем данные пакета

        }

        // Функции, которые должен реализовать каждый класс пакета
```

```
        public abstract void get(ChannelBuffer buffer);

        public abstract void send(ChannelBuffer buffer);

}
```

ChannelBuffer очень похож на DataInputStream и DataOutputStream. Большинство очень похожи.

Работа с клиентом

Работа с клиентом в основном определяется классом *PlayerHandler*:

```
public class PlayerHandler extends SimpleChannelUpstreamHandler {


        private PlayerWorkerThread worker;

        @Override

        public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {

        // Событие вызывается при подключении клиента. Я создаю здесь Worker игрока — объект, который занимается обработкой данных игрока непостредственно.

        // Я передаю ему канал игрока (функция e.getChannel()), чтобы он мог в него посылать пакеты

                worker = new PlayerWorkerThread(this, e.getChannel());

        }

        @Override

        public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {

        // Событие закрытия канала. Используется в основном, чтобы освободить ресурсы, или выполнить другие действия, которые происходят при отключении пользователя. Если его не обработать, Вы можете и не заметить, что пользователь отключился, если он напрямую не сказал этого серверу, а просто оборвался канал.

                worker.disconnectedFromChannel();
```

```
        }

        @Override

        public void messageReceived(ChannelHandlerContext ctx,
MessageEvent e) {

        // Функция принимает уже готовые Packet'ы от игрока, поэтому
их можно сразу посылать в worker. За их формирование отвечает другой
обработчик.

                if(e.getChannel().isOpen())

                        worker.acceptPacket((Packet) e.getMessage());

        }

        @Override

        public void exceptionCaught(ChannelHandlerContext ctx,
ExceptionEvent e) {

        // На канале произошло исключение. Выводим ошибку,
закрываем канал.

                Server.logger.log(Level.WARNING, "Exception from
downstream", e.getCause());

                ctx.getChannel().close();

        }

}
```

Worker может посылать игроку данные просто функцией channel.write(packet), где channel — канал игрока, который передаётся ему при подключении, а packet — объект класса Packet. За кодирование пакетов будет отвечать уже Encoder.

Decoder и Encoder

Собственно, сама важная часть системы — они отвечают за формирование пакетов Packet из потока пользователя и за отправку таких же пакетов в поток.

Encoder очень прост, он отправляет пакеты игроку:

```
public class PacketFrameEncoder extends OneToOneEncoder {

        @Override

        protected Object encode(ChannelHandlerContext channelhandlercontext, Channel channel, Object obj) throws Exception {

                if(!(obj instanceof Packet))

                        return obj; // Если это не пакет, то просто пропускаем его дальше

                Packet p = (Packet) obj;

                ChannelBuffer buffer = ChannelBuffers.dynamicBuffer(); // Создаём динамический буфер для записи в него данных из пакета. Если Вы точно знаете длину пакета, Вам не обязательно использовать динамический буфер — ChannelBuffers предоставляет и буферы фиксированной длинны, они могут быть эффективнее.

                Packet.write(p, buffer); // Пишем пакет в буфер

                return buffer; // Возвращаем буфер, который и будет записан в канал

        }

}
```

Decoder уже гораздо сложнее. Дело в том, что в буфере, пришедшем от клиента, может просто не оказаться достаточного количества байт для чтения всего пакета. В этом случае, поможет класс ReplayingDecoder. Нужно реализовать его функцию decode и читать в ней данные из потока, не заботясь не о чём:

```
public class PacketFrameDecoder extends ReplayingDecoder<VoidEnum> {

        @Override

        public void channelClosed(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
```

```
                ctx.sendUpstream(e);
        }
        @Override
        public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
                ctx.sendUpstream(e);
        }
        @Override
        protected Object decode(ChannelHandlerContext arg0, Channel arg1, ChannelBuffer buffer, VoidEnum e) throws Exception {
                return Packet.read(buffer);
        }
}
```

Перед вызовом функции decode декодер помечает текущий индекс чтения, если при чтении из буфера в нём не хватит данных, будет сгенерировано исключение. При этом буфер вернётся в начальное положение и decode будет повторён, когда больше данных будет получено от пользователя. В случае успешного чтения (возвращён не null), декодер попытается вызвать функции decode ещё раз, уже на оставшихся в буфере данных, если в нём есть ещё хотя бы один байт.

Результаты получились отличными. Во-первых, сервер больше не сыпет тысячей потоков — 4 потока Netty + 4 потока обработки данных прекрасно справляются с 250+ клиентами (тестирование продолжается). Во-вторых, нагрузка на процессор стала значительно меньшей и перестала линейно расти от числа подключений. В-третьих, время отклика в некоторых случаях стало меньше.

Ещё несколько полезных вещей

У Netty есть ещё несколько интересных особенностей, которые заслуживают отдельного упоминания:

Во-первых, остановка сервера:

```
ChannelFuture future = channel.close();
```

```
future.awaitUninterruptibly();
```

Где channel — канал, который возвратила функция bind в начале. future.awaitUninterruptibly() дождётся, пока канал закроется и выполнение кода продолжится.

Самое интересное: ChannelFuture. Когда отправляем на канал пакет, функцией channel.write(packet), она возвращает ChannelFuture — это особый объект, который отслеживает состояние выполняемого действия. Через него можно проверить, выполнилось ли действие.

Например, надо послать клиенту пакет отключения и закрыть за ним канал. Если сделаем

```
channel.write(new Packet255KickDisconnect("Пока!"));

channel.close();
```

то с вероятностью 99%, получим ChannelClosedException и пакет до клиента не дойдёт. Но можно сделать так:

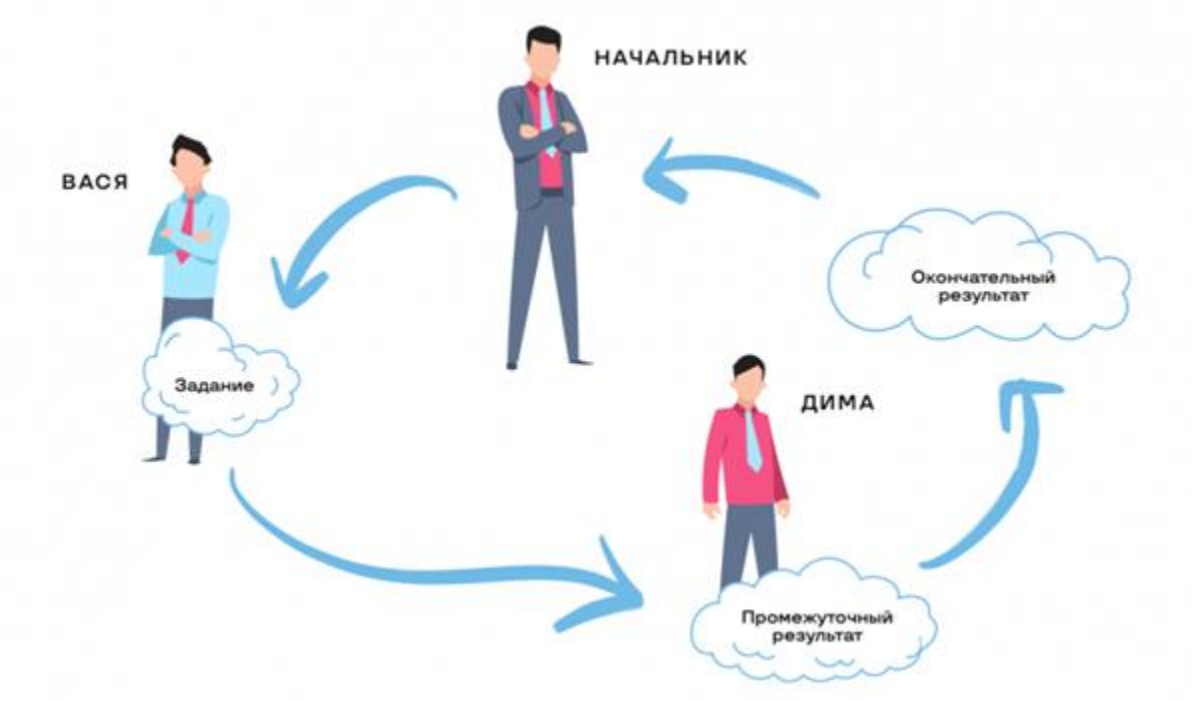
```
ChannelFuture future = channel.write(new Packet255KickDisconnect("Пока!"));

try {

    future.await(10000); // Ждём не более 10 секунд, пока действие закончится

} catch(InterruptedException e) {}

channel.close();
```

То всё будет хорошо, кроме того, что это может заблокировать поток выполнения, пока пакет не отправится пользователю. Поэтому на ChannelFuture можно повесит listener — объект, который будет уведомлён о том, что событие совершилось и выполнит какие-либо действия. Для закрытия соединения есть уже готовый listener *ChannelFutureListener.CLOSE*. Пример использования:

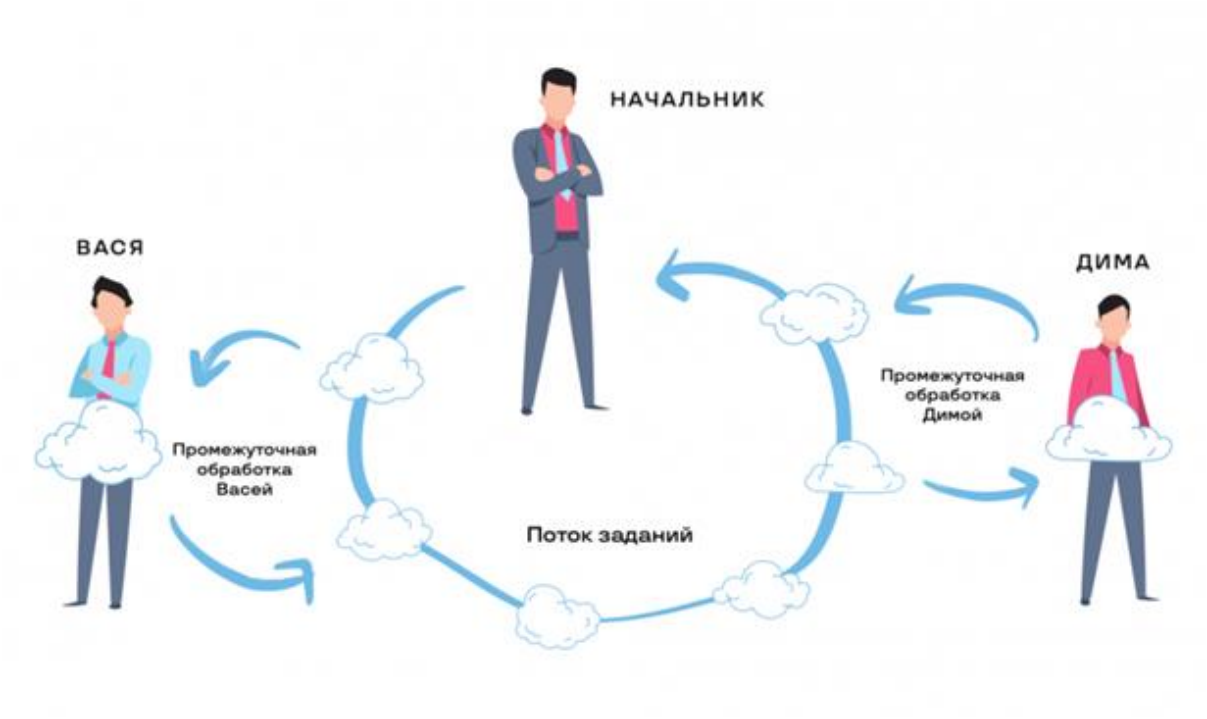
```
ChannelFuture future = channel.write(new Packet255KickDisconnect("Пока!"));

furute.addListener(ChannelFutureListener.CLOSE);
```

30. Что такое реактивность? пример

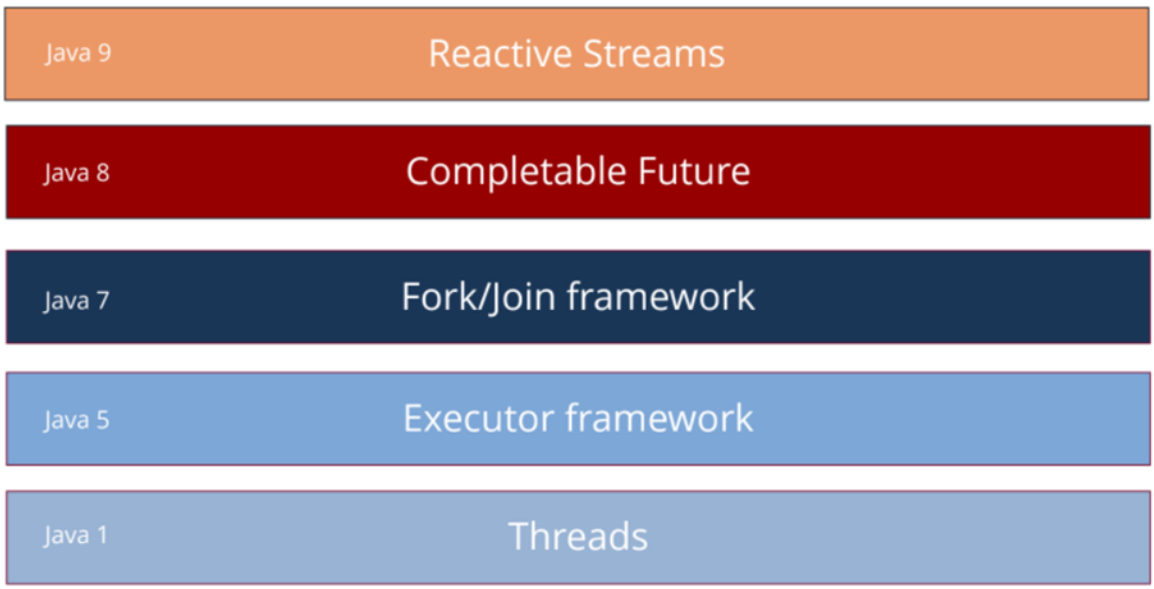
Реактивное программирование — это асинхронность, соединенная с потоковой обработкой данных. То есть если в асинхронной обработке нет блокировок потоков, но данные обрабатываются все равно порциями, то реактивность добавляет возможность обрабатывать данные потоком. Пример, когда начальник поручает задачу Васе, тот должен передать результат Диме, а Дима вернуть начальнику? Но задача — это некая порция, и пока она не будет сделана, дальше передать ее нельзя. Такой подход действительно разгружает начальника, но Дима и Вася периодически простаивают, ведь Диме надо дождаться результатов работы Васи, а Васе — дождаться нового задания.



А теперь представим, что задачу разбили на множество подзадач. И теперь они плывут непрерывным потоком:

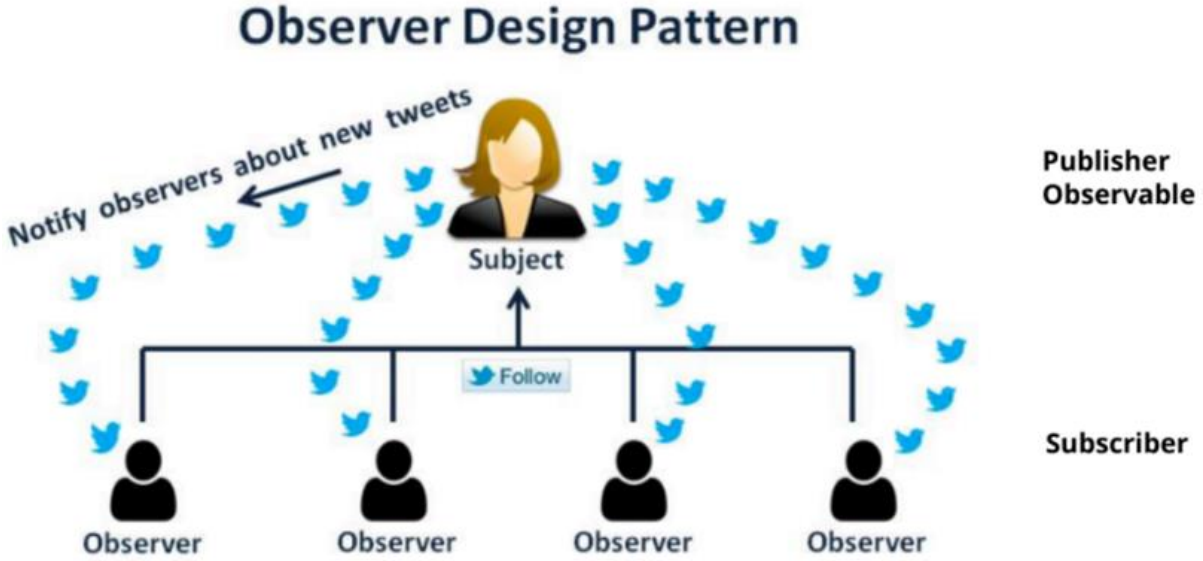


небольшие порции данных, конвейер с потоком данных, и каждый обработчик пропускает через себя эти данные, каким-то образом их преобразовывая. В качестве Васи и Димы выступают потоки выполнения (threads), обеспечивая, таким образом, многопоточную обработку данных.



На этой схеме показаны разные технологии распараллеливания, добавлявшиеся в Java в разных версиях. Как видим, спецификация Reactive Streams на вершине — она не заменяет всего, что было до нее, но добавляет

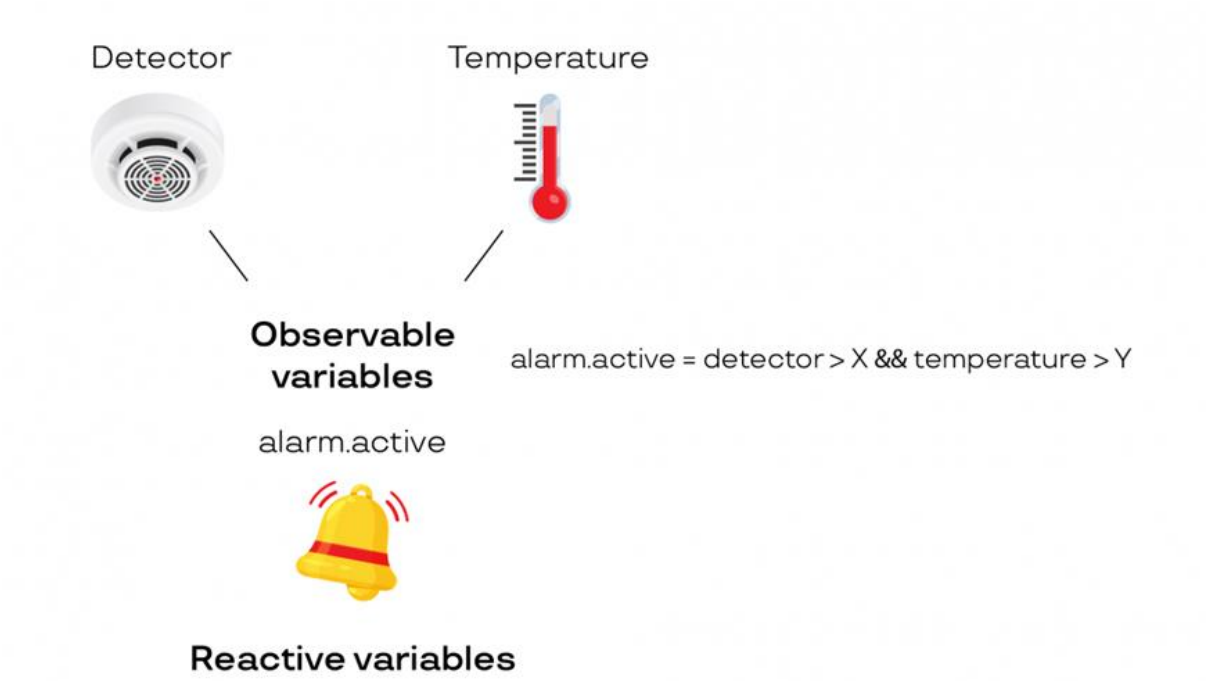
самый высокий уровень абстракции, а значит ее использование просто и эффективно. Попробуем в этом разобраться.
Идея реактивности построена на паттерне проектирования Observer.



Есть подписчики и то, на что подписываемся. В качестве примера рассмотрен Твиттер, подписаться на какое-то сообщество или человека, а потом получать обновления можно в любой соцсети. После подписки, как только появляется новое сообщение, всем подписчикам приходит notify, то есть уведомление. Это базовый паттерн.
В данной схеме есть:

- Publisher — тот, кто публикует новые сообщения;
- Observer — тот, кто на них подписан. В реактивных потоках подписчик обычно называется Subscriber. Термины разные, но по сути это одно и то же. В большинстве сообществ более привычны термины Publisher/Subscriber.

Это базовая идея, на которой все строится. Один из жизненных примеров реактивности — система оповещения при пожаре. Допустим, надо сделать систему, включающую тревогу в случае превышения задымленности и температуры.



Есть датчик дыма и градусник. Когда дыма становится много и/или температура растет, на соответствующих датчиках увеличивается значение. Когда значение и температура на датчике дыма оказываются выше пороговых, включается колокольчик и оповещает о тревоге.

Если бы был традиционный, а не реактивный подход, мы бы писали код, который каждые пять минут опрашивает детектор дыма и датчик температуры, и включает или выключает колокольчик. Однако в реактивном подходе это делает реактивный фреймворк, а мы только прописываем условия: колокольчик активен, когда детектор больше X, а температура больше Y. Это происходит каждый раз, когда приходит новое событие.

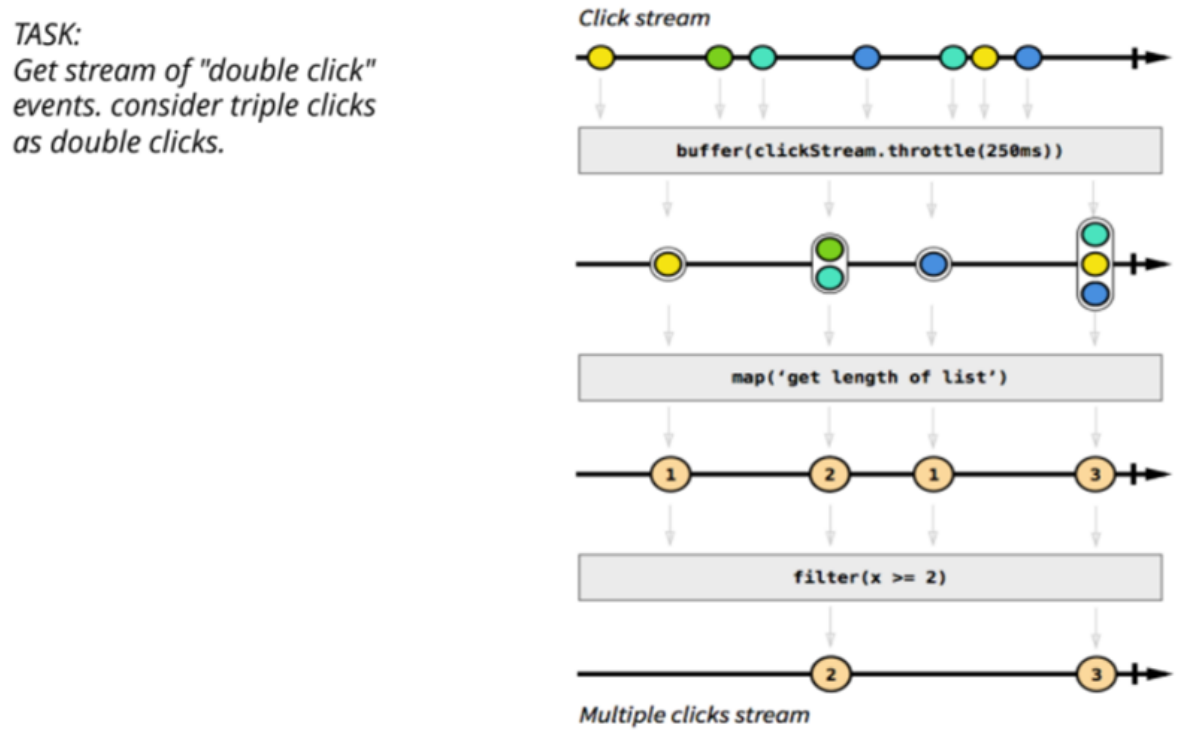
От детектора дыма идет поток данных: например, значение 10, потом 12, и т.д. Температура тоже меняется, это другой поток данных — 20, 25, 15. Каждый раз, когда появляется новое значение, результат пересчитывается, что приводит к включению или выключению системы оповещения. Нам достаточно сформулировать условие, при котором колокольчик должен включиться.

Если вернуться к паттерну Observer, детектор дыма и термометр — это публикаторы сообщений, то есть источники данных (Publisher), а колокольчик на них подписан, то есть он Subscriber, или наблюдатель (Observer).

Операторы позволяют каким-либо образом трансформировать потоки данных, меняя данные и создавая новые потоки.

Для примера рассмотрим оператор **distinctUntilChanged**. Он убирает одинаковые значения, идущие друг за другом. Действительно, если значение на детекторе дыма не изменилось — зачем на него реагировать и что-то там пересчитывать:

Рассмотрим еще один пример: допустим, разрабатываем UI, и нужно отслеживать двойные нажатия мышкой. Тройной клик будем считать как двойной.



Клики здесь — это поток щелчков мышкой (на схеме 1, 2, 1, 3). Нужно их сгруппировать. Для этого используется оператор throttle. Если два события (два клика) произошли в течение 250 мс, их нужно сгруппировать. На второй схеме представлены сгруппированные значения (1, 2, 1, 3). Это поток данных, но уже обработанных — в данном случае сгруппированных.

Таким образом начальный поток преобразовался в другой. Дальше нужно получить длину списка (1, 2, 1, 3). Фильтруем, оставляя только те значения, которые больше или равны 2. На нижней схеме осталось только два элемента (2, 3) — это и были двойные клики. Таким образом, преобразовали начальный поток в поток двойных кликов. Это и есть реактивное программирование: есть потоки на входе, каким-то образом пропускаем их через обработчики, и получаем поток на выходе. При этом вся обработка происходит асинхронно, то есть никто никого не ждет.

31. RxJava, кто такой Observable и Observer

Implementing and subscribing to an observer

В Java 9 нет реализации реактивных потоков — только спецификация. Но есть несколько библиотек — реализаций реактивного подхода.
RxJava - одна из самых обсуждаемых библиотек для обеспечения реактивного программирования в разработке Android. RxJava - это широко используемая библиотека для выполнения асинхронных задач. Он очень популярен среди разработчиков, потому что устраняет некоторый шаблонный код и повышает уровень абстракции.
Реактивное расширение (ReactiveX) - это библиотека для составления асинхронных и основанных на событиях программ с использованием наблюдаемых последовательностей"

Теперь, согласно определению, определим термины - асинхронный, основанный на событиях и т.д.

- *Асинхронный*

  Это означает, что разные части программы выполняются одновременно.

- *Основанный на событиях*

  Программа выполняет код на основе событий, сгенерированных во время выполнения программы. Например, нажатие кнопки запускает событие, а затем обработчик событий программы получает это событие и выполняет соответствующую работу.

- *Наблюдаемые последовательности*

  Observable и Flowable берут некоторые элементы и передают их подписчикам. Итак, эти элементы называются наблюдаемыми последовательностями или потоком данных.

- RxJava освобождает нас от ада обратного вызова, предоставляя стиль программирования. Мы можем подключать различные преобразования, которые напоминают функциональное программирование.

RxJava использует Observer и observable pattern, где субъект все время поддерживает своих наблюдателей, и если происходят какие-либо изменения, он уведомляет их, вызывая один из их методов.

Rx базируется на двух фундаментальных типах, в то время, как некоторые другие расширяют их функциональность. Этими базовыми типами являются **Observable** и **Observer**,

Rx построена на паттерне Observer. В этом нет ничего нового, обработчики событий уже существуют в Java (например, JavaFX EventHandler), однако они проигрывают в сравнении с Rx по следующим причинам:

- Обработку событий в них сложно компоновать
- Их вызов нельзя отложить
- Могут привести к утечке памяти
- Не существует простого способа сообщить об окончании потока событий
- Требуют ручного управления многопоточностью.

Observable

Observable – первый базовый тип. Этот класс содержит в себе основную часть реализации Rx, включая все базовые операторы. Рассмотрим их позже, а пока следует понять принцип работы метода **subscribe**. Вот ключевая перегрузка:

```
public final Subscription subscribe(Observer<? super T> observer)
```

Метод **subscribe** используется для получения данных выдаваемых observable. Эти данные передаются наблюдателю, который предполагает их обработку в зависимости от требований потребителя. Наблюдатель в этом случае является реализацией интерфейса **Observer**.

Observable сообщает три вида событий:

- Данные
- Сигнал о завершении последовательности (что означает, что новых данных больше не будет)
- Ошибку, если последовательность завершилась по причине исключительной ситуации (это событие так же предполагает завершение последовательности)

Observer

В Rx предусмотрена абстрактная реализация Observer, **Subscriber**. **Subscriber** реализует дополнительную функциональность и, как правило, именно его следует использовать для реализации **Observer**. Однако, для начала, рассмотрим только интерфейс:

```
interface Observer<T> {

    void onCompleted();

    void onError(java.lang.Throwable e);

    void onNext(T t);

}
```

Эти три метода являются поведением, которое описывает реакцию наблюдателя на сообщение от observable. **onNext** у наблюдателя будет вызван 0 или более раз, опционально сопровождаясь **onCompleted** или **onError**. После них вызовов больше не будет.

Реализация Observable и Observer

Можно в вручную реализовать **Observer** и **Observable**. В реальности в этом, как правило, нет необходимости: Rx предоставляет готовые решения, чтобы упростить разработку. Это также может быть не совсем безопасно, поскольку взаимодействие между частями библиотеки Rx включает в себя принципы и внутреннюю инфраструктуру, которые могут быть не очевидны новичку. В любом случае, будет проще для начала использовать множество инструментов уже предоставленных библиотекой для создания необходимого нам функционала. Чтобы подписаться на observable, совсем нет необходимости в реализации **Observer**. Существуют другие перегрузки метода **subscribe**, которые принимают в качестве аргументов соответствующие функции для **onNext**, **onError** и **onSubscribe**, инкапсулирующие создание экземпляра **Observer**. ~~Предоставлять их всех тоже не обязательно, вы можете описать только часть из них, например, только **onNext** или только **onNext** и **onError**. Лямбда-выражения в Java 1.8 делают эти перегрузки очень подходящими для использования в коротких примерах этой серии статей.~~

Subject

Subject'ы являются расширением **Observable**, одновременно реализуя интерфейс Observer. Они могут принимать сообщения о событиях (как observer) и сообщать о них своим подписчикам (как observable). Это делает их идеальной отправной точкой для знакомства с Rx кодом: когда есть данные, поступающие извне, можно передать их в Subject, превращая их таким образом в observable. Существует несколько реализаций Subject. Рассмотрим самые важные из них.

PublishSubject

PublishSubject – самая простая реализация **Subject**. Когда данные передаются в **PublishSubject**, он выдает их всем подписчикам, которые подписаны на него в данный момент.

```
public static void main(String[] args) {

    PublishSubject<Integer> subject = PublishSubject.create();
```

```
        subject.onNext(1);
        subject.subscribe(System.out::println);
        subject.onNext(2);
        subject.onNext(3);
        subject.onNext(4);
}
```

Вывод:

```
2
3
4
```

Как видим, **1** не была напечатана из-за того, что не были подписаны в момент когда она была передана. После того как подписались, начали получать все значения поступающие в subject.

Здесь впервые используем метод **subscribe**, так что стоит уделить этому внимание. В данном случае используем перегруженную версию, которая принимает один объект класса Function, отвечающий за **onNext**. Эта функция принимает значение типа Integer и ничего не возвращает. Функции, которые ничего не возвращают также называются actions.

Можно передать эту функцию следующими способами:

- Предоставить объект класса **Action1<Integer>**
- Неявно создать таковой используя лямбда-выражение
- Передать ссылку на существующий метод с соответствующей сигнатурой. В данном случае, **System.out::println** имеет перегруженную версию, которая принимает **Object**, поэтому мы передаем ссылку на него. Таким образом, подписка позволяет нам печатать в основной поток вывода все поступающие в **Subject** числа.

32. Для чего нужен ReplaySubject, пример

ReplaySubject имеет специальную возможность кэшировать все поступившие в него данные. Когда у него появляется новый подписчик, последовательность выдана ему начиная с начала. Все последующие поступившие данные будут выдаваться подписчикам как обычно.

```
ReplaySubject<Integer> s = ReplaySubject.create();

s.subscribe(v -> System.out.println("Early:" + v));

s.onNext(0);

s.onNext(1);

s.subscribe(v -> System.out.println("Late: " + v));

s.onNext(2);
```

Вывод

```
Early:0

Early:1

Late: 0

Late: 1

Early:2

Late: 2
```

Все значения были получены, не смотря на то, что один из подписчиков подписался позже другого. Следует обратить внимание, что до того как получить новое значение, подписчик получает все пропущенные. Таким образом, порядок последовательности для подписчика не нарушен.

Кэшировать всё подряд не всегда лучшая идея, так как последовательности могут быть длинными или даже бесконечными. Фабричный метод ReplaySubject.createWithSize ограничивает размер

буфера, а ReplaySubject.createWithTime время, которое объекты будут оставаться в кеше.

```
ReplaySubject<Integer> s = ReplaySubject.createWithSize(2);

s.onNext(0);

s.onNext(1);

s.onNext(2);

s.subscribe(v -> System.out.println("Late: " + v));

s.onNext(3);
```

Вывод

```
Late: 1

Late: 2

Late: 3
```

Подписчик на этот раз пропустил первое значение, которое выпало из буфера размером 2.

Таким же образом со временем из буфера выпадают объекты у Subject созданного при помощи createWithTime.

```
ReplaySubject<Integer> s =ReplaySubject.createWithTime(150, TimeUnit.MILLISECONDS, Schedulers.immediate());

s.onNext(0);

Thread.sleep(100);

s.onNext(1);

Thread.sleep(100);

s.onNext(2);

s.subscribe(v -> System.out.println("Late: " + v));
```

```
s.onNext(3);
```

Вывод

```
Late: 1

Late: 2

Late: 3
```

Создание ReplaySubject с ограничением по времени требует объект планировщика (Scheduler), который является представлением времени в Rx.

ReplaySubject.createWithTimeAndSize ограничивает буфер по обоим параметрам

33. Для чего нужен BehaviorSubject, пример

BehaviorSubject хранит только последнее значение. Это то же самое, что и ReplaySubject, но с буфером размером 1. Во время создания ему может быть присвоено начальное значение, таким образом гарантируя, что данные всегда будут доступны новым подписчикам.

```
BehaviorSubject<Integer> s = BehaviorSubject.create();

s.onNext(0);

s.onNext(1);

s.onNext(2);

s.subscribe(v -> System.out.println("Late: " + v));

s.onNext(3);
```

Вывод

```
Late: 2

Late: 3
```

Начальное значение предоставляется для того, чтобы быть доступным еще до поступления данных.

Так как роль BehaviorSubject – всегда иметь доступные данные, считается неправильным создавать его без начального значения, также как и завершать его.

34. Для чего нужен AsyncSubject, пример

AsyncSubject также хранит последнее значение. Разница в том, что он не выдает данных до тех пока не завершится последовательность. Его используют, когда нужно выдать единое значение и тут же завершиться.

```
AsyncSubject<Integer> s = AsyncSubject.create();

s.subscribe(v -> System.out.println(v));

s.onNext(0);

s.onNext(1);

s.onNext(2);

s.onCompleted();
```

Вывод

```
2
```

Обратите внимание, что если бы мы не вызвали s.onCompleted(), этот код ничего бы не напечатал.

Неявная инфраструктура

Существуют принципы, которые могут быть не очевидны в коде. Один из важнейших заключается в том, что ни одно событие не будет выдано после того, как последовательность завершена (onError или onCompleted). Реализация subject' уважает эти принципы:

```
Subject<Integer, Integer> s = ReplaySubject.create();
```

```
s.subscribe(v -> System.out.println(v));

s.onNext(0);

s.onCompleted();

s.onNext(1);

s.onNext(2);
```

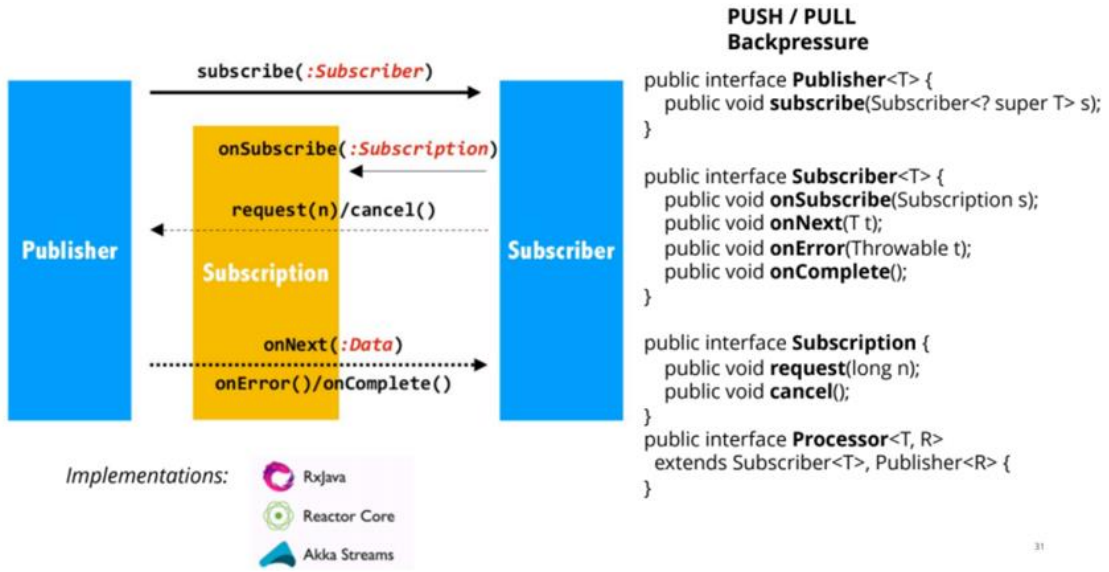
Вывод

```
0
```

Безопасность не может быть гарантирована везде, где используется Rx, поэтому вам лучше быть осведомленным и не нарушать этот принцип, так как это может привести к неопределенным последствиям.

35. Реактивные потоки в Java 9, для чего нужны, пример

Реактивные потоки вошли в Java 9 как спецификация.
Если предыдущие технологии (Completable Future, Fork/Join framework) получили свою имплементацию в JDK, то реактивные потоки имплементации не имеют. Есть только очень короткая спецификация. Там всего 4 интерфейса:



Если рассматривать пример из картинки про Твиттер, можно сказать, что:

Publisher — тот, кто постит твиты;
Subscriber — подписчик. Определяет , что делать, если:

- Начали слушать поток (onSubscribe). Когда успешно подписались, вызовется эта функция;
- Появилось очередное значение в потоке (onNext);
- Появилось ошибочное значение (onError);
- Поток завершился (onComplete).

Subscription — есть подписка, которую можно отменить (cancel) или запросить определенное количество значений (request(long n)). Можно определить поведение при каждом следующем значении, а можем забирать значения вручную.
Processor — обработчик — это два в одном: он одновременно и Subscriber, и Publisher. Он принимает какие-то значения и куда-то их кладет.
Если надо на что-то подписаться, вызываем Subscribe, подписываемся, и потом каждый раз будем получать обновления. Можно запросить их вручную с помощью request. А можно определить поведение при приходе нового сообщения (onNext): что делать, если появилось новое сообщение, что делать, если пришла ошибка и что делать, если Publisher завершил поток. Мы можем определить эти callbacks, или отписаться (cancel).

36. Модели реактивных потоков в Java 9

PUSH / PULL модели
Существует две модели потоков:

- Push-модель — когда идет «проталкивание» значений.

Например, подписались на кого-то в Telegram или Instagram и получаете оповещения (они так и называются — push-сообщения, их не надо запрашиваеть, они приходят сами). Это может быть, например, всплывающее сообщение. Можно определить, как реагировать на каждое новое сообщение.

- Pull-модель — когда мы сами делаем запрос.

Например, мы не хотим подписываться, т.к. информации и так слишком много, а хотим сами заходить на сайт и узнавать новости.
Для Push-модели определяем callbacks, то есть функции, которые будут вызваны, когда придет очередное сообщение, а для Pull-модели можно воспользоваться методом request, когда мы захотим узнать, что новенького.

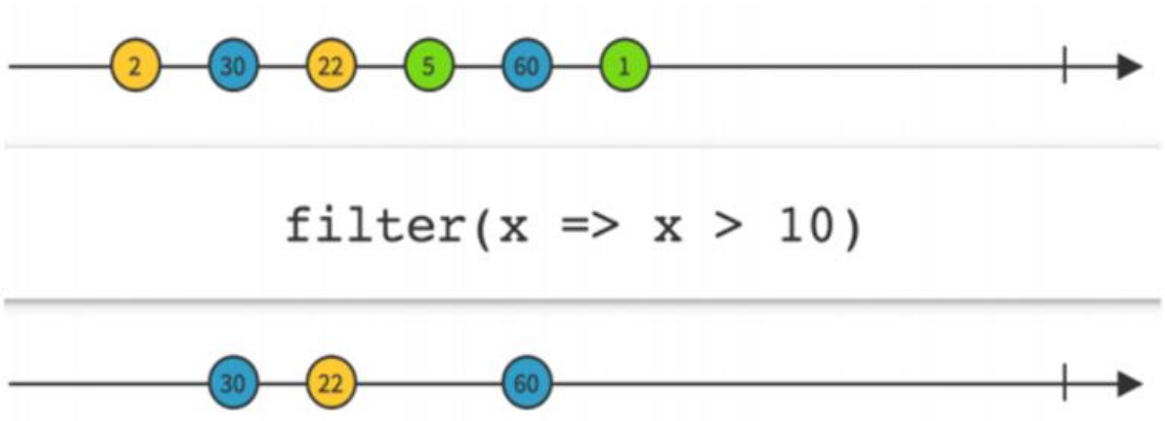
Pull-модель очень важна для Backpressure — «напирания» сзади. Что же это такое?

Вы можете быть просто заспамленными своими подписками. В этом случае прочитать их все нереально, и есть шанс потерять действительно важные данные — они просто утонут в этом потоке сообщений. Когда подписчик из-за большого потока информации не справляется со всем, что публикует Publisher, получается Backpressure. В этом случае можно использовать Pull-модель и делать request по одному сообщению, прежде всего из тех потоков данных, которые наиболее важны для вас.

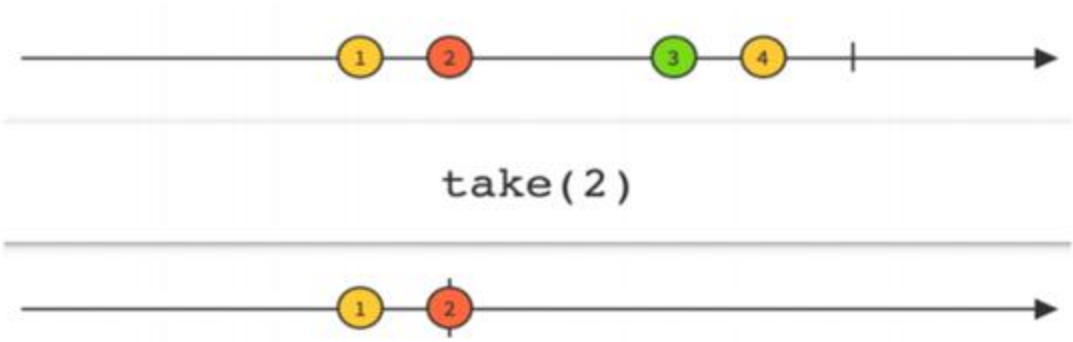
37. Что такое Filter operator,Map operator, Delay operator для чего нужны, примеры

В реактивных потоках огромное количество операторов. Многие из них похожи на те, которые есть в обычных стримах Java. Мы рассмотрим только несколько самых распространенных операторов, которые понадобятся нам для практического примера применения реактивности.

Filter operator



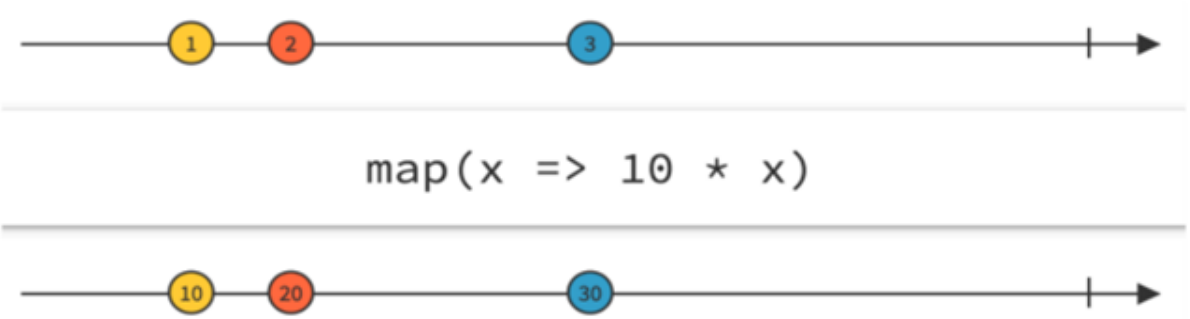
По синтаксису этот фильтр точно такой же, как обычный. Но если в стриме Java 8 все данные есть сразу, здесь они могут появляться постепенно. Стрелки вправо — это временная шкала, а в кружочках находятся появляющиеся данные. Мы видим, что фильтр оставляет в итоговом потоке только значения, превышающие 10.



Take 2 означает, что нужно взять только первые два значения.

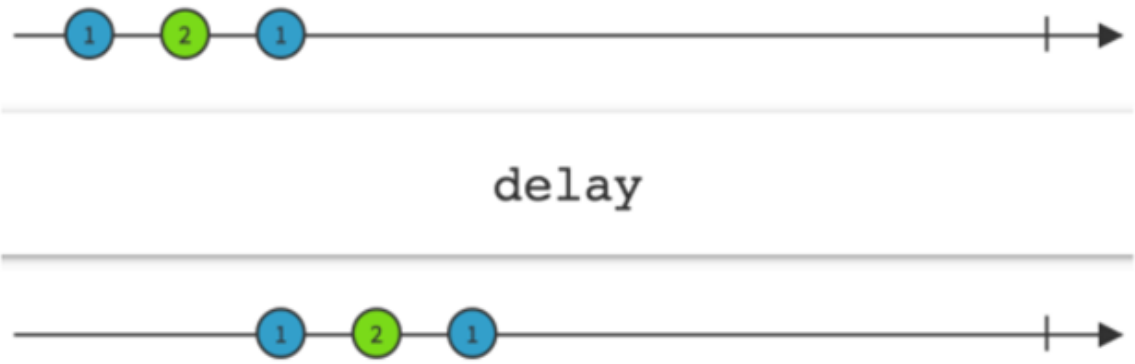
Map operator

Оператор Map тоже хорошо знаком:



Это действие, происходящее с каждым значением. Здесь — умножить на десять: было 3, стало 30; было 2, стало 20 и т.д.

Delay operator

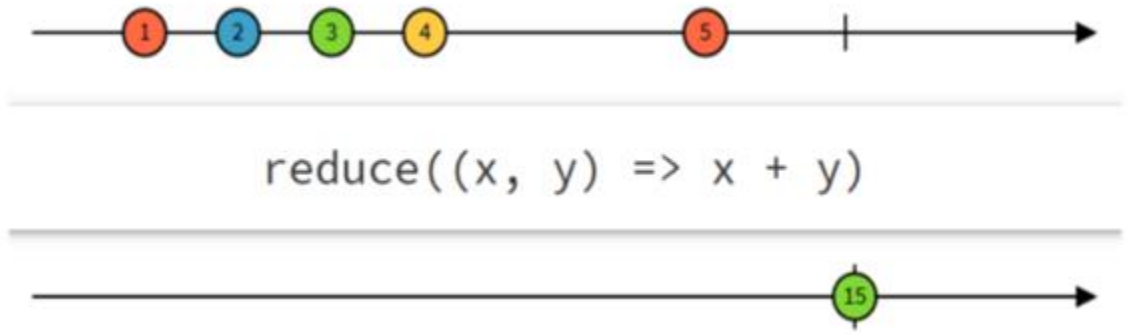


Задержка: все операции сдвигаются. Этот оператор может понадобиться, когда значения уже генерируются, но подготовительные процессы еще происходят, поэтому приходится отложить обработку данных из потока.

38. Что такое Reduce operator,Scan operator, Merge operator для чего нужны, примеры

Reduce operator

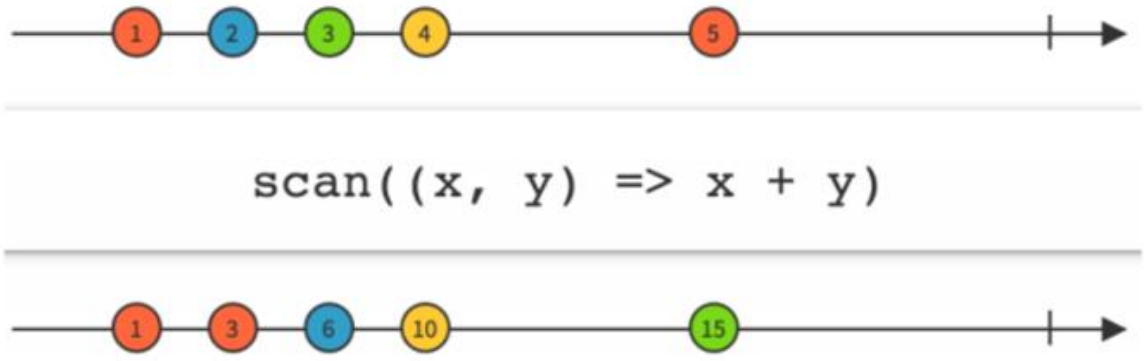
Еще один известный оператор:



Он дожидается конца работы потока (onComplete) — на схеме она представлена вертикальной чертой. После чего мы получаем результат — здесь это число 15. Оператор reduce сложил все значения, которые были в потоке.

Scan operator

Этот оператор отличается от предыдущего тем, что не дожидается конца работы потока.

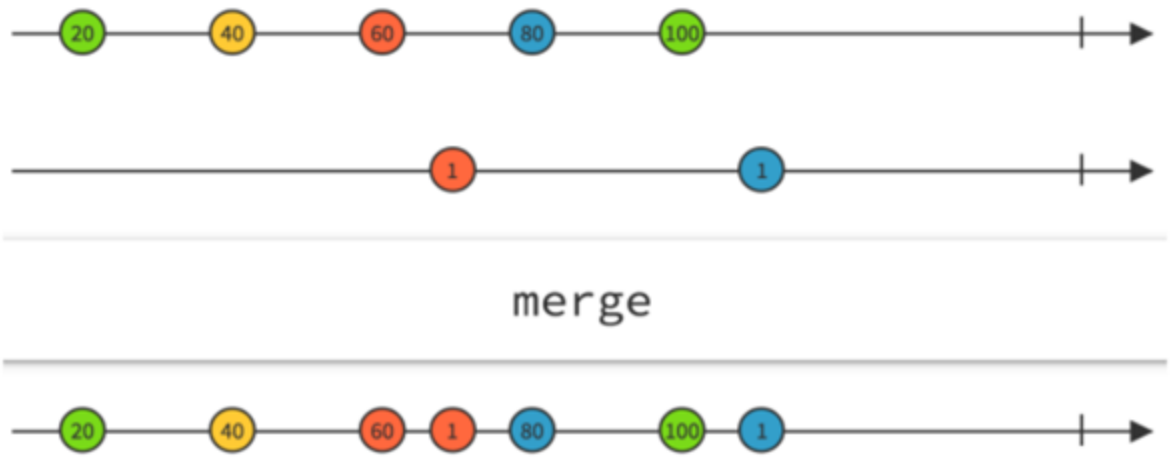


Оператор scan рассчитывает текущее значение нарастающим итогом: сначала был 1, потом прибавил к предыдущему значению 2, стало 3, потом прибавил 3, стало 6, еще 4, стало 10 и т.д. На выходе получили 15. Дальше мы видим вертикальную черту — onComplete. Но, может быть, его никогда

не произойдет: некоторые потоки не завершаются. Например, у термометра или датчика дыма нет завершения, но scan поможет рассчитать текущее суммарное значение, а при некоторой комбинации операторов — текущее среднее значение всех данных в потоке.

Merge operator

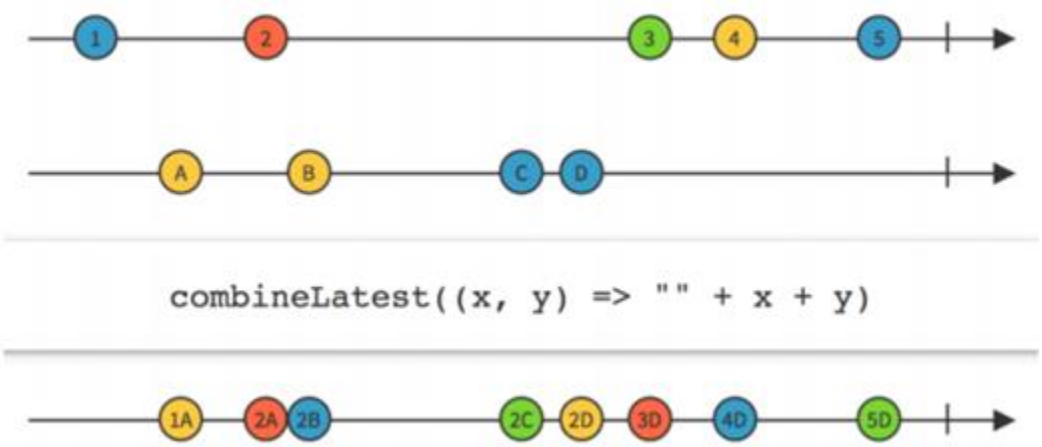
Объединяет значения двух потоков.



Например, есть два температурных датчика в разных местах, а нам нужно обрабатывать их единообразно, в общем потоке .

Combine latest

Получив новое значение, комбинирует его с последним значением из предыдущего потока.



Если в потоке возникает новое событие, мы его комбинируем с последним полученным значением из другого потока. Скажем, таким образом мы можем комбинировать значения от датчика дыма и термометра: при появлении нового значения температуры в потоке temperatureStream оно будет комбинироваться с последним полученным значением задымленности из smokeStream. И мы будем получать пару значений. А уже по этой паре можно выполнить итоговый расчет:

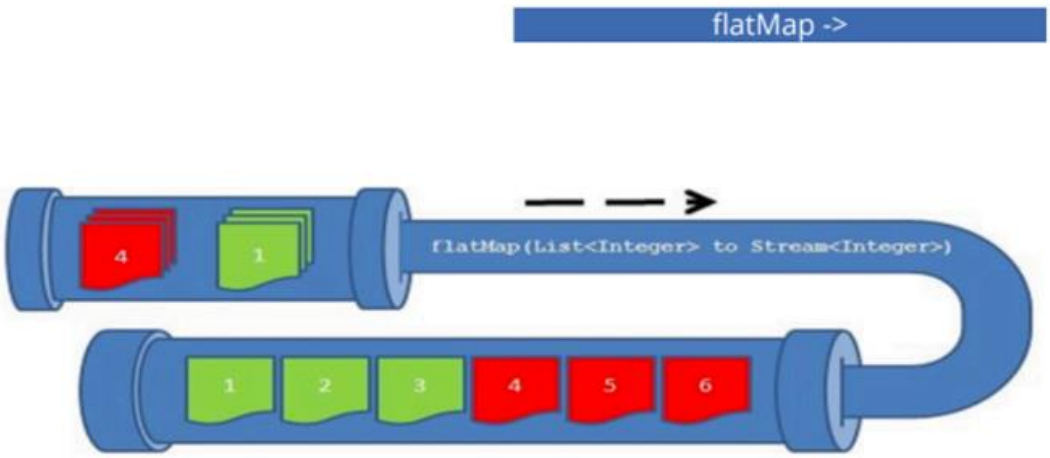
temperatureStream.combineLatest(smokeStream).map((x, y) -> x > X && y > Y)

В итоге на выходе у нас получается поток значений true или false — включить или выключить колокольчик. Он будет пересчитываться каждый раз, когда будет появляться новое значение в temperatureStream или в smokeStream.

39. Что такое FlatMap operator,Buffer operator, для чего нужны, примеры

FlatMap operator

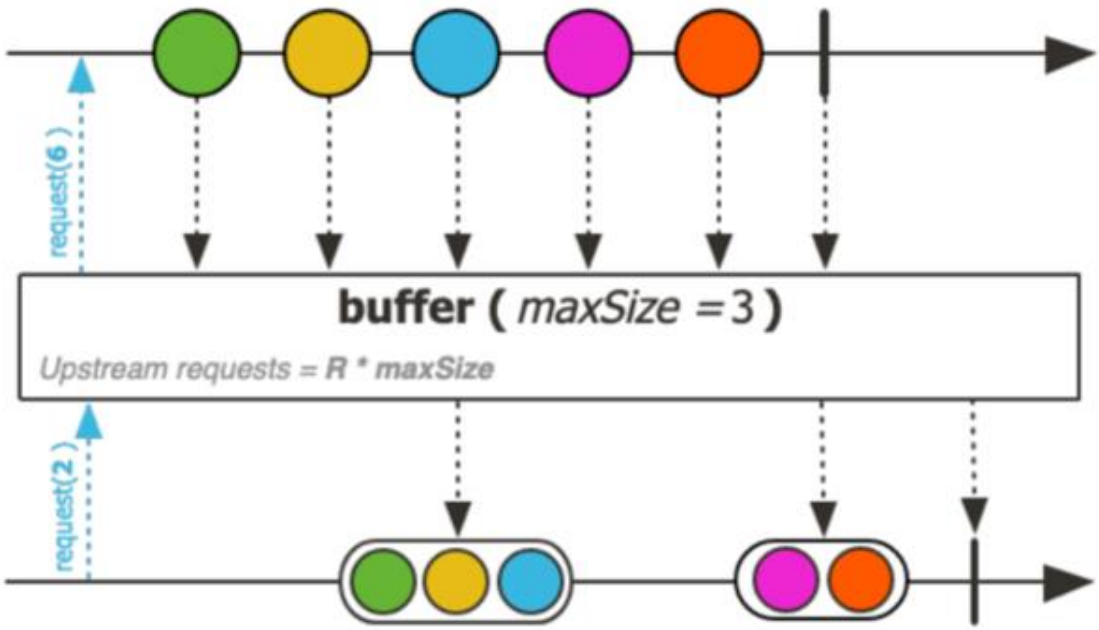
Этот оператор вам, скорее всего, знаком по стримам Java 8. Элементами потока в данном случае являются другие потоки. Получается поток потоков. Работать с ними неудобно, и в этих случаях нам может понадобиться «уплостить» поток.



Можно представить такой поток как конвейер, на который ставят коробки с запчастями. До того, как мы начнем их применять, запчасти нужно достать из коробок. Именно это делает оператор flatMap.

Flatmap часто используется при обработке потока данных, полученных с сервера. Т.к. сервер возвращает поток, чтобы мы смогли обрабатывать отдельные данные, этот поток сначала надо «развернуть». Это и делает flatMap.

Buffer operator



Это оператор, который помогает группировать данные. На выходе Buffer получается поток, элементами которого являются списки (List в Java). Он может пригодиться, когда мы хотим отправлять данные не по одному, а порциями.

Реактивные потоки позволяют разбить задачу на подзадачи, и обрабатывать их маленькими порциями. Но иногда лучше наоборот, собрать много маленьких частей в блоки. Скажем, продолжая пример с конвейером и запчастями, нам может понадобиться отправлять запчасти на другой завод (другой сервер). Но каждую отдельную запчасть отправлять неэффективно. Лучше их собрать в коробки, скажем по 100 штук, и отправлять более крупными партиями.

На схеме выше мы группируем отдельные значения по три элемента (так как всего их было пять, получилась «коробка» из трех, а потом из двух значений). То есть если flatMap распаковывает данные из коробок, buffer, наоборот, упаковывает их.

Всего существует более сотни операторов реактивного программирования. Здесь разобрана только небольшая часть.

Итого

Есть два подхода:

- Spring MVC — традиционная модель, в которой используется JDBC, императивная логика и т.д.
- Spring WebFlux, в котором используется реактивный подход и сервер Netty.

40. Что такое side effects, методы. Примеры

Методы побочных эффектов не влияют на поток сам по себе. Вместо этого они вызываются, когда происходят определенные события, чтобы позволить реагировать на эти события. Например: если заинтересованы в том, чтобы делать что-то вне ваших `Subscriber` обратных вызовов при возникновении какой-либо ошибки, вы должны использовать `doOnError()` метод и передавать ему функциональный интерфейс, который будет использоваться при возникновении ошибки:

```
someObservable
        .doOnError(new Action1() {
        @Override
        public void call(Throwable t) {
        // use this callback to clean up resources,
        // log the event or or report the
        // problem to the user
```

```
    }
    })
    //…
```

Наиболее важной частью является `call()` метод. Код этого метода будет выполнен до `Subscriber`вызова `onError()` метода.

В дополнение к исключениям RxJava предлагает еще много событий, на которые вы можете реагировать:

События и соответствующие им побочные эффекты		
етод	**Функциональный интерфейс**	**Мероприятие**
OnSubscribe ()	Action0	Подписчик подписывается на Observable
OnUnsubscribe ()	Action0	Подписчик отписывается от подписки
OnNext ()	Action1 <T>	Следующий элемент выбрасывается
OnCompleted ()	Action0	The Observable больше не будет излучать предметы
OnError ()	Action1 <T>	Произошла ошибка
OnTerminate ()	Action0	Либо произошла ошибка, либо Observable больше не будет излучать элементы

loOnEach ()	Action1 <Уведомление <T >>	Либо элемент был выпущен, наблюдаемое завершено, либо произошла ошибка. Объект Notification содержит информацию о типе события
loOnRequest ()	Action1 <Long>	Нижестоящий оператор запрашивает выделение большего количества элементов

<T> относится либо к типу испускаемого предмета, либо, в случае onError() метода, к типу броска Throwable. Все функциональные интерфейсы имеют тип Action0 или Action1 . Это означает, что отдельные методы этих интерфейсов ничего не возвращают и принимают либо нулевые аргументы, либо один аргумент, в зависимости от конкретного события. Поскольку эти методы ничего не возвращают, они не могут использоваться для изменения испускаемых элементов и, таким образом, никоим образом не изменяют поток элементов. Вместо этого эти методы предназначены для создания побочных эффектов, таких как запись чего-либо на диск, очистка состояния или чего-либо еще, что манипулирует состоянием самой системы вместо потока событий.

Для чего они полезны?

Теперь, поскольку они не меняют поток предметов, для них должно быть другое использование. Три примера того, чего можно достичь с помощью этих методов:

- Использовать doOnNext() для отладки
- Использовать doOnError() внутри flatMap() для обработки ошибок
- Используйте doOnNext() для сохранения / кэширования результатов сети

Итак, давайте посмотрим на эти примеры в деталях.

Использование doOnNext () для отладки

С RxJava Observable иногда работает не так, как ожидалось. Особенно, когда только начинаете его использовать. Поскольку используется свободный API для преобразования какого-либо источника во что-то, на что вы хотите подписаться, вы увидите только то, что получите в конце этого конвейера преобразования. Есть «плавный» API для перемещения из одного потока одного типа в другой поток другого типа. Но что, если это не сработает, как ожидалось? С Java 8 Streams есть `peek()` метод. Можно использовать `doOnNext()` метод в любом месте конвейера обработки, чтобы увидеть, что происходит и каков промежуточный результат.

Вот пример этого:

```
Observable someObservable = Observable

        .from(Arrays.asList(new Integer[]{2, 3, 5, 7, 11}))

        .doOnNext(System.out::println)

        .filter(prime -> prime % 2 == 0)

        .doOnNext(System.out::println)

        .count()

        .doOnNext(System.out::println)

        .map(number -> String.format("Contains %d elements", number));

Subscription subscription = o.subscribe(

        System.out::println,

        System.out::println,

        () -> System.out.println("Completed!"));
```

И вот вывод этого кода:

```
2

3

3
```

```
5

5

7

7

11

11

4

Contains 4 elements

Completed!
```

Таким образом, можно получить ценную информацию о том, что происходит, когда ваш Observable ведет себя не так, как вы ожидали.

Эти `doOnError()` и `doOnCompleted()` методы могут быть также полезны для отладки состояния

Использование doOnError() внутри flatMap()

Допустим, вы используете Retrofit(*Retrofit — это REST клиент для Java и Android. Он позволяет легко получить и загрузить JSON (или другие структурированные данные) через веб-сервис на основе REST.*) для доступа к какому-то ресурсу по сети. Поскольку Retrofit поддерживает наблюдаемые объекты, можно легко использовать эти вызовы в своей цепочке обработки с помощью `flatMap()`. Сетевые вызовы могут пойти не так во многих отношениях, особенно на мобильных устройствах. В `flatMap()` методе есть Observable. Таким образом, можно использовать `doOnError()` метод, чтобы каким-то образом изменить пользовательский интерфейс, но при этом иметь рабочий поток Observable для будущих событий.

Итак, как это выглядит:

```
flatMap(id -> service.getPost()

        .doOnError(t -> {
```

```
        // report problem to UI
        })
        .onErrorResumeNext(Observable.empty())
)
```

Этот метод особенно полезен, если вы запрашиваете удаленный ресурс в результате потенциально повторяющихся событий пользовательского интерфейса.

Использование doOnNext() для сохранения/кеширования сетевых результатов

Если в какой-то момент вашей цепочки вы совершаете сетевые вызовы, вы можете использовать `doOnNext()` для сохранения входящих результатов в вашей локальной базе данных или поместить их в какой-то кеш. Пример на слайде:

```
// getOrderById is getting a fresh order
// from the net and returns an observable of orders
// Observable<Order> getOrderById(long id) {…}

Observable.from(aListWithIds)
        .flatMap(id -> getOrderById(id)
                        .doOnNext(order -> cacheOrder(order))
        // carry on with more processing
```

41. Обаботчики ошибок в RxJava

В роли базового обработчика ошибок в RxJava используется RxJavaPlugins.onError. Он обрабатывает все ошибки, которые не удается

доставить до подписчика. По умолчанию, все ошибки отправляются именно в него, поэтому могут возникать критические сбои приложения.

Если у RxJava нет базового обработчика ошибок — подобные ошибки будут скрыты от нас и разработчики будут находится в неведении относительно потенциальных проблем в коде.

Начиная с версии 2.0.6, RxJavaPlugins.onError разделяет ошибки библиотеки/реализации и ситуации когда ошибку доставить невозможно. Ошибки, отнесенные к категории «багов» вызываются как есть, остальные же оборачиваются в UndeliverableException и после вызываются.

Одна из основных ошибок, с которыми сталкиваются в RxJava — OnErrorNotImplementedException. Эта ошибка возникает, если observable вызывает ошибку, а в подписчике не реализован метод onError. Данная ошибка — пример ошибки, которая для базового обработчика ошибок RxJava является «багом» и не оборачивается в UndeliverableException.

UndeliverableException

Поскольку ошибки относящиеся к «багам» легко исправить — не будем на них останавливаться.Ошибки,которые RxJava оборачивает в UndeliverableException, интереснее, так как не всегда может быть очевидно почему же ошибка не может быть доставлена до onError. Случаи, в которых это может произойти, зависят от того, что конкретно делают источники и подписчики. Примеры рассмотрим ниже, но в общем можно сказать, что такая ошибка возникает, если нет активного подписчика, которому может быть доставлена ошибка.

Пример с zipWith()

Первый вариант, в котором можно вызвать `UndeliverableException` — оператор `zipWith`.

```
val observable1 = Observable.error<Int>(Exception())

val observable2 = Observable.error<Int>(Exception())

val zipper = BiFunction<Int, Int, String> { one, two -> "$one - $two" }

observable1.zipWith(observable2, zipper)
```

```
    .subscribe(
        { System.out.println(it) },
        { it.printStackTrace() }
    )
```

Можно предположить, что onError будет вызван дважды, но это противоречит спецификации Reactive streams. После единственного вызова терминального события (onError, onCompelete) требуется, чтобы никаких вызовов больше не осуществлялось Получается, что при единственном вызове onError повторный вызов уже невозможен. Что произойдёт при возникновении в источнике второй ошибки? Она будет доставлена в RxJavaPlugins.onError. Простой способ попасть в подобную ситуацию — использовать zip для объединения сетевых вызовов (например, два вызова Retrofit, возвращающие Observable). Если в обоих вызовах возникает ошибка (например, нет интернет соединения) — оба источника вызовут ошибки, первая из которых попадёт в реализацию onError, а вторая будет доставлена базовому обработчику ошибок (RxJavaPlugins.onError).

Пример с ConnectableObservable без подписчиков

ConnectableObservable также может вызвать UndeliverableException. Стоит напомнить, что ConnectableObservable вызывает события независимо от наличия активных подписчиков, достаточно вызвать метод connect(). Если при отсутствии подписчиков в ConnectableObservable возникнет ошибка — она будет доставлена базовому обработчику ошибок.

пример, который может вызвать такую ошибку:

```
someApi.retrofitCall() // Сетевой вызов с использованием Retrofit
    .publish()
    .connect()
```

Если someApi.retrofitCall() вызовет ошибку (например, нет подключения к интернету) — приложение упадет, так как сетевая ошибка будет доставлена базовому обработчику ошибок RxJava. Этот пример кажется выдуманным, но очень легко попасть в ситуацию, когда ConnectableObservable все еще соединен(connected), но подписчиков у него нет.

Обрабатываем ошибки

Первый шаг — посмотреть на возникающие ошибки и попытаться определить что их вызывает. Идеально, если вам удастся исправить проблему у её источника, чтобы предотвратить передачу ошибки в RxJavaPlugins.onError.

Решение для примера с zipWith — взять один или оба источника и реализовать в них один из методов для перехвата ошибок. Например, можно использовать onErrorReturn для передачи вместо ошибки значения по умолчанию. Пример с ConnectableObservable исправить проще — просто надо убедиться в отсоединении Observable в момент, когда последний подписчик отписывается. autoConnect(), к примеру, имеет перегруженную реализацию, которая принимает функцию, отлавливающую момент соединения.

Другой путь решения проблемы — подменить базовый обработчик ошибок своим собственным. Метод RxJavaPlugins.setErrorHandler(Consumer<Throwable>) поможет в этом. Если это подходящее для решение — можно перехватывать все ошибки отправленные в RxJavaPlugins.onError и обрабатывать их по своему усмотрению. Это решение может оказаться довольно сложным — помните, что RxJavaPlugins.onError получает ошибки от всех потоков (streams) RxJava в приложении.

Если вручную создаются Observable, то можно вместо emitter.onError() вызывать emitter.tryOnError(). Этот метод передает ошибку только если поток (stream) не уничтожен (terminated) и имеет подписчиков. Надо помнить, что данный метод экспериментальный.

42. Горячие и холодные потоки в RxJava, что они делают

В RxJava есть два вида Observable: Hot и Cold.

Cold Observable:

• Не рассылает объекты, пока на него не подписался хотя бы один подписчик;

• Если observable имеет несколько подписчиков, то он будет рассылать всю последовательность объектов каждому подписчику.

Пример cold observable – методы ретрофит-интерфейса. Каждый раз когда вызывается метод subscribe(), выполняется соответствующий запрос на бэкенд и подписчик получает объект-респонс.

Hot Observable:

• Рассылает объекты, когда они появляются, независимо от того есть ли подписчики;

• Каждый новый подписчик получает только новые объекты, а не всю последовательность.

Рассмотрим cold Observables.

Cold Observables

Cold Observables очень похожа на музыкальный компакт-диск, который может быть воспроизведен каждым слушателем, и каждый может слушать эту музыку в любое время.

Точно так же холодные Observables могут воспроизводить свои Наблюдения для каждого Наблюдателя, гарантируя, что все Наблюдатели получают все данные. Большинство управляемых данными Observables холодные, включая фабрики Observable.just () и Observable.fromIterable ().

Пример: у нас есть два наблюдателя, подписанных на Observable. Observable сначала испускает все выбросы первому Observer, а затем вызывает onComplete (). Затем он снова отправляет все выбросы второму Обозревателю и затем вызывает onComplete (). Через два отдельных потока они оба получают одинаковые данные. Это типичное поведение cold Observables:

```
import io.reactivex.Observable;

public class Launcher {

        public static void main(String[] args) {

        Observable<String> source =

                Observable.just("Alpha", "Beta", "Gamma", "Delta",
"Epsilon");

        //first observer
```

```
        source.subscribe(s -> System.out.println("Observer 1 Received: " +
s));

        //second observer

        source.subscribe(s -> System.out.println("Observer 2 Received: " +
s));

    }

}
```

Даже после того, как второй наблюдатель преобразовал выбросы, он все равно получает поток собственных выбросов. Использование map() и filter() может по-прежнему создавать cold Observables:

```
import io.reactivex.Observable;

public class Launcher {

        public static void main(String[] args) {

        Observable<String> source =

                Observable.just("Alpha", "Beta", "Gamma", "Delta",
"Epsilon");

        //first observer

        source.subscribe(s -> System.out.println("Observer 1 Received: " +
s));

        //second observer

        source.map(String::length).filter(i -> i >= 5)

                .subscribe(s -> System.out.println("Observer 2 Received: " +
s));

    }

}
```

Его вывод

```
Observer 1 Received: Alpha

Observer 1 Received: Beta
```

```
Observer 1 Received: Gamma

Observer 1 Received: Delta

Observer 1 Received: Epsilon

Observer 2 Received: 5

Observer 2 Received: 5

Observer 2 Received: 5

Observer 2 Received: 7
```

Если надо сделать запрос к базе данных SQLite, можно включить драйвер JDBC SQLite и библиотеку RxJava-JDBC =.Затем можно запросить таблицу базы данных реактивно, например так:

```
import com.github.davidmoten.rx.jdbc.ConnectionProviderFromUrl;

import com.github.davidmoten.rx.jdbc.Database;

import rx.Observable;

import java.sql.Connection;

public class Launcher {

        public static void main(String[] args) {

        Connection conn = new
ConnectionProviderFromUrl("jdbc:sqlite:/home/thomas/rexon_metals.db"
).get();

        Database db = Database.from(conn);

        Observable<String> customerNames =

        db.select("SELECT NAME FROM
CUSTOMER").getAs(String.class);

        customerNames.subscribe(s -> System.out.println(s));

        }

}
```

Observable на основе SQL - это cold. Многие Observables передают данные из ограниченных источников данных (таких как базы данных, текстовые файлы или JSON), и все они холодные. RxJava-JDBC выполняет запросы для каждого наблюдателя. Это означает, что если данные изменяются между двумя подписками, второй наблюдатель может получать выбросы, отличные от первой. Cold Наблюдаемые будут восстанавливать эмиссию для каждого Наблюдателя.

Hot Observables

Hot Observable больше похожа на радиостанцию. В то же время он передает одинаковые выбросы всем наблюдателям. Если наблюдатель подписывается на hot **Observable**, получает те же выбросы, а затем приходит к другому наблюдателю, второй наблюдатель пропустит эти выбросы. Также как радиостанция, если вы включите ее поздно, вы не услышите эту песню. hot Observables обычно представляют события, а не ограниченные наборы данных. События могут нести данные, но они являются чувствительными ко времени компонентами, и более поздние наблюдатели будут пропускать предыдущие данные. Например, событие пользовательского интерфейса JavaFX или Android может быть представлено как Hot Observable. В JavaFX можно использовать метод selectedProperty() объекта ToggleButton Добавить наблюдателя. Затем надо преобразовать логическое излучение в строку, указывающую состояние кнопки (ВВЕРХ или ВНИЗ). Observer используется для отображения в метке:

```
import io.reactivex.Observable;

import javafx.application.Application;

import javafx.beans.value.ChangeListener;

import javafx.beans.value.ObservableValue;

import javafx.scene.Scene;

import javafx.scene.control.Label;

import javafx.scene.control.ToggleButton;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

public class MyJavaFxApp extends Application {

    @Override
```

```
        public void start(Stage stage) throws Exception {
        ToggleButton toggleButton = new ToggleButton("TOGGLE ME");
        Label label = new Label();
        Observable<Boolean> selectedStates = valuesOf(toggleButton.selectedProperty());
        selectedStates.map(selected -> selected ? "DOWN" : "UP")
        .subscribe(label::setText);
        VBox vBox = new VBox(toggleButton, label);
        stage.setScene(new Scene(vBox));
        stage.show();
        }
    private static <T> Observable<T> valuesOf(final ObservableValue<T> fxObservable) {
        return Observable.create(observableEmitter -> {
        //emit initial state
        observableEmitter.onNext(fxObservable.getValue());
        //emit value changes uses a listener
        final ChangeListener<T> listener = (observableValue, prev, current) -> observableEmitter.onNext(current);
        fxObservable.addListener(listener);
        });
        }
}
```

JavaFX ObservableValue не имеет ничего общего с RxJava Observable.

При каждом нажатии кнопки ToggleButton Observable будет выдавать соответствующее значение true или false в зависимости от состояния. События пользовательского интерфейса JavaFX и Android в основном

являются Hot Observable. Можно использовать Hot Observable для отражения запросов сервера. Если добавить Observable Твиты для темы в Твиттере, которая также популярна в Observable. Hot Observable не должна быть неограниченной, пока она делится выбросами со всеми наблюдателями, Выбросы, не пропущенные при воспроизведении, горячие.

ConnectableObservable

ConnectableObservable является полезным для hot Observable. Это может быть любая Наблюдаемая (в том числе cold), пусть она станет hot, так что все выбросы выдаются Наблюдателям только один раз. Чтобы сделать этот переход, надо вызвать publish() для любого Observable, и генерируется ConnectableObservable. Тем не менее, подписка не может начать выбросы. Нужно вызвать его метод connect(), чтобы начать эмиссию выбросов. Таким образом, можно предварительно настроить своих наблюдателей:

```
import io.reactivex.Observable;

import io.reactivex.observables.ConnectableObservable;

public class Launcher {

        public static void main(String[] args) {

        ConnectableObservable<String> source =

                Observable.just("Alpha", "Beta", "Gamma", "Delta",
"Epsilon").publish();

        //Set up observer 1

        source.subscribe(s -> System.out.println("Observer 1: " + s));

        //Set up observer 2

        source.map(String::length)

                .subscribe(i -> System.out.println("Observer 2: " + i));

        //Fire!

        source.connect();

    }

}
```

Его вывод

```
Observer 1: Alpha

Observer 2: 5

Observer 1: Beta

Observer 2: 4

Observer 1: Gamma

Observer 2: 5

Observer 1: Delta

Observer 2: 5

Observer 1: Epsilon

Observer 2: 7
```

Обратите внимание, что один наблюдатель получает строку, а другой - длину. Оба чередуются. Подписки предварительно настроены, а затем вызовите connect(), чтобы начать передачу. Каждое излучение отправляется каждому наблюдателю одновременно. Используйте ConnectableObservable, чтобы заставить каждую эмиссию отправляться всем наблюдателям одновременно, это называется многоадресной рассылкой, которая будет подробно обсуждаться позже. ConnectableObservable полезен для предотвращения воспроизведения данных для каждого наблюдателя.

43. Освобождение потоков

Disposable Является связующим звеном между Observable и active Observer, вызов его dispose()метода останавливает выбросы и удаляет все ресурсы, используемые для этого Observer:

```
fun main() {

    val seconds = Observable.interval(1, TimeUnit.SECONDS)

    val disposable = seconds.subscribe { println("Received: $it") }

    TimeUnit.SECONDS.sleep(5)
```

```
    // Dispose and stop emissions.
    disposable.dispose()
    // Sleep 5 secs: no new emissions.
    TimeUnit.SECONDS.sleep(5)
}
```

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
```

Disposable в наблюдателе

Observer получает свое собственное при подписке в качестве Disposable аргумента onSubscribe(d: Disposable). приведен пример Observer самоутилизации после потребления выбросов:

```
fun main() {
    val seconds = Observable.interval(1, TimeUnit.SECONDS)
    seconds.subscribe(MyObserver())
    TimeUnit.SECONDS.sleep(5)
}
class MyObserver<T> : Observer<T> {
    private lateinit var disposable: Disposable
    override fun onSubscribe(disposable: Disposable) {
    this.disposable = disposable
```

```
        }
        override fun onNext(t: T) {
        println("Received: $t")
        disposable.dispose() // Self dispose
        }
        override fun onError(e: Throwable) {
        e.printStackTrace()
        }
        override fun onComplete() {
        println("Done !")
        }
}
```

```
Received: 0
```

По умолчанию вызов subscribe() с Observer экземпляром не возвращает a Disposable, но его можно получить, расширив Resource Observer и подписавшись с помощью subscribeWith():

```
fun main() {
        val seconds = Observable.interval(1, TimeUnit.SECONDS)
        val disposable: Disposable = seconds.subscribeWith(MyResourceObserver())
        TimeUnit.SECONDS.sleep(5)
        // Dispose and stop emissions.
        disposable.dispose()
        // Sleep 5 secs: no new emissions.
        TimeUnit.SECONDS.sleep(5)
}
```

```
class MyResourceObserver<T> : ResourceObserver<T>() {
        override fun onNext(t: T) {
        println("Received: $t")
        }
        override fun onError(e: Throwable) {
        e.printStackTrace()
        }
        override fun onComplete() {
        println("Done !")
        }
}
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
```

CompositeDisposable

Для управления и удаления нескольких подписок CompositeDisposable полезно:

```
fun main() {
        val disposables = CompositeDisposable()
        val seconds = Observable.interval(1, TimeUnit.SECONDS)
        // Subscribe and capture disposables
        val disposable1 = seconds.subscribe { println("Observer 1: $it") }
        val disposable2 = seconds.subscribe { println("Observer 2: $it") }
        // Put disposables into CompositeDisposable
```

```
        disposables.addAll(disposable1, disposable2)
        // Sleep 5 secs
        TimeUnit.SECONDS.sleep(3)
        //dispose all disposables
        disposables.dispose()
        //sleep 5 seconds: no emissions.
        TimeUnit.SECONDS.sleep(3)
}
```

```
Observer 1: 0
Observer 2: 0
Observer 1: 1
Observer 2: 1
Observer 1: 2
Observer 2: 2
```

Удаление с помощью Observable.create()

В случае вызова Observable.create() и результат возврата *long-running или infinite* Observable , функция isDisposed() должна регулярно проверять, продолжать ли отправлять запросы:

```
fun main() {
        val source: Observable<Int> = Observable.create { observableEmitter ->
        try {
        for (i in 0..1_000_000) {
                if (observableEmitter.isDisposed)
                return@create
                observableEmitter.onNext(i)
```

```
        }
        observableEmitter.onComplete()
        } catch (e: Exception) {
        observableEmitter.onError(e)
        }
        }
}
```

В случае , если Observable.create() обернут вокруг какого-либо ресурса, удаление этого ресурса *должно быть обработано* для предотвращения утечек. ObservableEmitter имеет setCancellable() и setDisposable()методы для этого:

```
fun <T> valuesOf(fxObservable: ObservableValue<T>): Observable<T> {
        return Observable.create { observableEmitter ->
        // Emit initial state
        observableEmitter.onNext(fxObservable.value)
        // Emit value changes uses a listener
        val listener = ChangeListener<T> { _, _, current -> observableEmitter.onNext(current) }
        // Add listener to ObservableValue
        fxObservable.addListener(listener)
        // Handle disposing by specifying cancellable
        observableEmitter.setCancellable { fxObservable.removeListener(listener) }
        }
}
```

44. Реактивные операторы, какие бывают, примеры

Каждая языковая реализация ReactiveX реализует набор операторов. Несмотря на то, что между реализациями существует много общего, есть также некоторые операторы, которые реализованы только в определенных реализациях. Кроме того, каждая реализация имеет тенденцию называть свои операторы так, чтобы они напоминали аналогичные методы, которые уже знакомы из других контекстов этого языка.

Цепочки операторов

Большинство операторов работают с Observable и возвращают Observable. Это позволяет применять эти операторы один за другим, в цепочке. Каждый оператор в цепочке изменяет Observable, являющийся результатом операции предыдущего оператора.

Существуют и другие шаблоны, такие как шаблон Builder, в котором множество методов определенного класса воздействуют на элемент того же класса, изменяя этот объект посредством операции метода. Эти шаблоны также позволяют вам связывать методы аналогичным образом. Но в то время как в Builder Pattern порядок появления методов в цепочке обычно не имеет значения, с Observable операторами *порядок имеет значение* .

Цепочка операторов Observable не работает независимо от исходного Observable, который создает цепочку, но они работают *по очереди* , каждый из которых работает с Observable, сгенерированным оператором, непосредственно предшествующим в цепочке.

Операторы ReactiveX

Сначала перечислим операторы, которые можно считать «основными» в ReactiveX. Далее «дерево решений», которое может помочь выбрать оператора, наиболее подходящего для вашего варианта использования. Есть алфавитный список большинства операторов, доступных во многих языковых реализациях ReactiveX.

Операторы по категориям

Создание наблюдаемых

Операторы, которые создают новые Observables.

- **Create**— создать Observable с нуля, программно вызывая методы наблюдателя
- **Defer**— не создавать Observable, пока наблюдатель не подпишется, и создавать новый Observable для каждого наблюдателя

- **Empty// Never—Throw** создаем Observable с очень точным и ограниченным поведением
- **From**— преобразовать какой-либо другой объект или структуру данных в Observable
- **Interval**— создать Observable, который выдает последовательность целых чисел, разделенных определенным интервалом времени
- **Just**— конвертировать объект или набор объектов в Observable, который излучает те или иные объекты
- **Range**— создать Observable, который выдает диапазон последовательных целых чисел
- **Repeat**- создать Observable, который многократно испускает определенный элемент или последовательность элементов
- **Start**— создать Observable, который выдает возвращаемое значение функции
- **Timer**— создать Observable, который испускает один элемент после заданной задержки

Преобразование наблюдаемых

Операторы, преобразующие элементы, испускаемые Observable.

- **Buffer**— периодически собирать элементы из Observable в пакеты и выпускать эти пакеты, а не выпускать элементы по одному
- **FlatMap**— преобразовать элементы, испускаемые Observable, в Observable, а затем свести выбросы из них в один Observable.
- **GroupBy**- разделить Observable на набор Observable, каждый из которых испускает другую группу элементов из исходного Observable, организованную по ключу
- **Map**- преобразовать элементы, испускаемые Observable, применяя функцию к каждому элементу
- **Scan**— применить функцию к каждому элементу, испускаемому Observable, последовательно, и испускать каждое последующее значение
- **Window**— периодически подразделять элементы из Observable на окна Observable и создавать эти окна, а не создавать элементы по одному

Фильтрация наблюдаемых

Операторы, которые выборочно испускают элементы из исходного Observable.

- **Debounce**— испускать элемент из Observable только в том случае, если в течение определенного промежутка времени он не излучал другой элемент
- **Distinct**— подавлять повторяющиеся элементы, испускаемые Observable
- **ElementAt**— испускать только элемент *n* , испускаемый Observable
- **Filter**— испускать только те элементы из Observable, которые проходят предикатную проверку
- **First**— испускать только первый элемент или первый элемент, отвечающий условию, из Observable
- **IgnoreElements**— не испускать никаких элементов из Observable, но отражать его уведомление о прекращении
- **Last**— испускать только последний элемент, испускаемый Observable
- **Sample**— испускать самый последний элемент, испускаемый Observable в течение периодических интервалов времени
- **Skip**— подавить первые *n* элементов, испускаемых Observable
- **SkipLast**— подавить последние *n* элементов, испускаемых Observable
- **Take**— испускать только первые *n* элементов, испускаемых Observable
- **TakeLast**— испускать только последние *n* элементов, испускаемых Observable

Объединение наблюдаемых

Операторы, которые работают с несколькими исходными Observable для создания одного Observable

- **And// Then—When** объединить наборы элементов, испускаемых двумя или более Observable с помощьюPatterniPlanпосредников
- **CombineLatest**— когда элемент испускается одним из двух Observable, объединяйте последний элемент, испускаемый каждым Observable с помощью указанной функции, и испускайте элементы на основе результатов этой функции.
- **Join-** объединять элементы, испускаемые двумя наблюдаемыми, всякий раз, когда элемент из одного наблюдаемого испускается в

течение временного окна, определенного в соответствии с элементом, испускаемым другим наблюдаемым
- **Merge**— объединить несколько Observables в один, объединив их выбросы
- **StartWith**— испускать указанную последовательность элементов, прежде чем начать испускать элементы из исходного Observable
- **Switch**- преобразовать Observable, который испускает Observables, в один Observable, который испускает элементы, испускаемые самым последним из этих Observables
- **Zip**- объединять выбросы нескольких Observable вместе с помощью указанной функции и испускать отдельные элементы для каждой комбинации на основе результатов этой функции.

Операторы обработки ошибок
Операторы, которые помогают восстанавливаться после уведомлений об ошибках от Observable
- **Catch**— восстановиться после onErrorуведомления, продолжив последовательность без ошибок
- **Retry**— если исходный Observable отправляет onErrorуведомление, повторно подпишитесь на него в надежде, что оно завершится без ошибок

Наблюдаемые служебные операторы
Набор полезных операторов для работы с Observables

- **Delay**— сдвинуть выбросы от Observable вперед во времени на определенную величину
- **Do**— зарегистрировать действие для различных событий жизненного цикла Observable
- **Materialize/Dematerialize** — представлять как отправленные элементы, так и уведомления, отправленные как отправленные элементы, или отменить этот процесс
- **ObserveOn**— указать планировщик, на котором наблюдатель будет наблюдать за этим Observable
- **Serialize**- заставить Observable делать сериализованные вызовы и вести себя хорошо
- **Subscribe**- работать с выбросами и уведомлениями от Observable

- **SubscribeOn**— указать планировщик, который должен использовать Observable, когда он подписан
- **TimeInterval**- преобразовать Observable, который испускает элементы, в тот, который испускает указания количества времени, прошедшего между этими выбросами
- **Timeout**— зеркально отразить исходный Observable, но выдавать уведомление об ошибке, если в течение определенного периода времени отсутствуют какие-либо испускаемые элементы.
- **Timestamp**— прикрепить временную метку к каждому элементу, испускаемому Observable
- **Using**— создать одноразовый ресурс, который имеет тот же срок службы, что и Observable

Условные и логические операторы

Операторы, которые оценивают один или несколько Observables или элементов, испускаемых Observables

- **All**— определить, соответствуют ли все элементы, испускаемые Observable, некоторым критериям
- **Amb**- при наличии двух или более исходных Observables испускать все элементы только из первого из этих Observables, чтобы испускать элемент
- **Contains**— определить, испускает ли Observable конкретный элемент или нет
- **DefaultIfEmpty**— испускать элементы из исходного Observable или элемент по умолчанию, если исходный Observable ничего не испускает
- **SequenceEqual**— определить, испускают ли два Observable одну и ту же последовательность элементов
- **SkipUntil**- отбрасывать элементы, испускаемые Observable, до тех пор, пока второй Observable не испустит предмет
- **SkipWhile**— отбрасывать элементы, испускаемые Observable, пока указанное условие не станет ложным
- **TakeUntil**- отбрасывать элементы, испускаемые Observable после того, как второй Observable испускает элемент или завершается
- **TakeWhile**— отбрасывать элементы, испускаемые Observable после того, как указанное условие становится ложным

Математические и агрегатные операторы

Операторы, которые работают со всей последовательностью элементов, испускаемых Observable

- **Average**— вычисляет среднее число чисел, испускаемых Observable, и выдает это среднее
- **Concat**— испускать выбросы от двух или более Observable, не чередуя их
- **Count**— подсчитать количество элементов, испускаемых исходным Observable, и испускать только это значение
- **Max**- определить и испустить элемент с максимальным значением, испускаемый Observable
- **Min**- определить и выдать элемент с минимальным значением, испускаемый Observable
- **Reduce**— применить функцию к каждому элементу, испускаемому Observable, последовательно, и выдать окончательное значение
- **Sum**— вычислить сумму чисел, испускаемых Observable, и испустить эту сумму

Операторы противодавления

- **операторы обратного давления** — стратегии для работы с наблюдаемыми, которые производят элементы быстрее, чем их наблюдатели потребляют их.

Подключаемые наблюдаемые операторы

Специализированные Observables с более точным контролем динамики подписки.

- **Connect**- указать подключаемому Observable начать отправку элементов своим подписчикам
- **Publish**— преобразовать обычный Observable в подключаемый Observable
- **RefCount**— заставить Connectable Observable вести себя как обычный Observable
- **Replay**— убедитесь, что все наблюдатели видят одну и ту же последовательность испускаемых элементов, даже если они подписываются после того, как Observable начал излучать элементы

Операторы для преобразования наблюдаемых

- **To**— преобразовать Observable в другой объект или структуру данных

45. Дерево решений наблюдаемых операторов

Это дерево может помочь вам найти нужный оператор ReactiveX Observable.
Это дерево может помочь вам найти нужный оператор ReactiveX Observable.

Я хочу создать новый Наблюдательный
который испускает конкретную вещь
Just
который был возвращен из функции,вызванной в абонентское время.
Start
который был возвращен из Action , Callable , Runnable или чего-то в этом роде, вызванного во время подписки
From
с указанной задержкой
Timer
который извлекает свои выбросы из определенного Array , Iterable или чего-то в этом роде
From
извлекая его из будущего
Start
который получает свою последовательность из будущего
From
который многократно испускает последовательность элементов
Repeat
с нуля,с индивидуальной логикой
Create
для каждого наблюдателя,который подписался
Defer
который испускает последовательность целых чисел
Range
в определённые промежутки времени
Interval
с указанной задержкой
Timer

который завершается без выброса предметов

Empty

который ничего не делает

Never

Я хочу создать Наблюдателя,комбинируя другие Наблюдатели.

и выбрасывать все предметы из всех Наблюдателей в любом порядке их поступления.

Merge

и выбрасывая все предметы из всех Наблюдателей,по одному Наблюдателю за раз.

Concat

объединяя элементы из двух или более "Наблюдателей" последовательно,чтобы придумать новые элементы для излучения.

всякий раз, когда *каждый* из Observables испускает новый элемент

Zip

всякий раз, когда *какой-либо* из Observables испускает новый элемент

CombineLatest

всякий раз,когда объект испускается одним наблюдателем в окне,определенном объектом,испускаемым другим наблюдателем.

Join

через посредников Pattern и Plan

And/Then/When

и выбрасывая предметы только из самых последних из этих Наблюдателей.

Switch

Я хочу излучать предметы от Наблюдателя после их трансформации.

по одному с функцией

Map

испуская все элементы,испускаемые соответствующими наблюдателями.

FlatMap

по одному Наблюдаемому за раз,в том порядке,в котором они испускаются.

ConcatMap

на основе всех пунктов,которые им предшествовали.

Scan

прикрепив к ним временную метку

Timestamp

в показатель времени,которое прошло до выброса данного элемента

TimeInterval

Я хочу сдвинуть предметы,излучаемые Наблюдателем,вперед по времени,прежде чем возвращать их.

Delay

Я хочу преобразовать элементы *и* уведомления из Observable в элементы и повторно отправить их
заключая их в объекты Notification

Materialize

которую я могу снова развернуть

Dematerialize

Я хочу проигнорировать все элементы,излучаемые наблюдателем,и пройти только по его завершенному/уведомлению об ошибке.

IgnoreElements

Я хочу зеркально отобразить Наблюдаемые,но префиксные элементы в его последовательности.

StartWith

только если его последовательность пуста

DefaultIfEmpty

Я хочу собирать предметы из Наблюдаемого и возвращать их в качестве буферов предметов.

Buffer

содержащий только последние испущенные элементы

TakeLastBuffer

Я хочу разделить одну Наблюдаемую на несколько Наблюдаемых.

Window

чтобы похожие предметы оказались на одном и том же наблюдательном месте.

GroupBy

Я хочу получить конкретный предмет,излучаемый Наблюдателем:
последний пункт,высвобожденный до его завершения

Last

единственное изделие,которое оно выпустило

Single

первый элемент,который он выбросил

First

Я хочу вернуть только некоторые вещи от Наблюдателя.
отфильтровывая те,которые не совпадают с какими-либо предикатами.

Filter

то есть,только первый пункт

First
то есть только первый элемент *s*
Take
то есть,только последний пункт
Last
то есть,только пункт*n*
ElementAt
то есть,только те предметы после первых.
то есть,после первого*n*items
Skip
то есть до тех пор,пока один из этих предметов не совпадет с предикатом.
SkipWhile
то есть,по прошествии первоначального периода времени
Skip
то есть,после того,как "Наблюдатель" во второй раз испускает предмет.
SkipUntil
то есть,те предметы,кроме последних.
то есть,кроме последнего*n*items
SkipLast
то есть до тех пор,пока один из этих предметов не совпадет с предикатом.
TakeWhile
то есть,за исключением элементов,излучаемых в течение периода времени,предшествующего завершению работы источника.
SkipLast
то есть,за исключением предметов,испускаемых после второго Наблюдательный испускает предмет.
TakeUntil
путем периодической выборки Наблюдаемого
Sample
только излучая предметы,за которыми в течение некоторого времени не следуют другие предметы.
Debounce
путем подавления элементов,которые являются дубликатами уже выпущенных элементов.
Distinct
если они немедленно следуют за пунктом,то они являются дубликатами
DistinctUntilChanged

откладывая мою подписку на него на некоторое время после того,как он начнет испускать элементы.

DelaySubscription

Я хочу получить предметы из "Наблюдателя" только при условии,что он был первым из коллекции "Наблюдателей",который выпустил предмет.

Amb

Я хочу оценить всю последовательность элементов,излучаемых Наблюдателем.

и испустить одно логическое значение, указывающее, *все* ли элементы проходят некоторый тест

All

и испустить одно логическое значение, указывающее, испускал ли Observable *какой-либо* элемент (который проходит некоторый тест)

Contains

и испустить одно логическое значение, указывающее, не испускал ли Observable *никаких* элементов

IsEmpty

и испускает один логический сигнал,указывающий,идентична ли последовательность,испускаемая вторым Наблюдателем.

SequenceEqual

и излучают среднее значение всех их значений

Average

и выдать сумму всех их ценностей

Sum

и выдать число,указывающее,сколько единиц в последовательности.

Count

и выдать изделие с максимальным значением

Max

и выпустить изделие с минимальным значением

Min

путем применения функции агрегирования к каждому элементу по очереди и выдачи результата

Scan

Я хочу преобразовать всю последовательность элементов,испускаемых наблюдателем,в какую-то другую структуру данных.

To

Я хочу, чтобы оператор работал с конкретным планировщиком

SubscribeOn

когда он уведомляет наблюдателей

ObserveOn

Я хочу,чтобы наблюдатель вызывал определенное действие,когда происходят определенные события.

Do

Я хочу наблюдателя,который уведомит наблюдателей об ошибке.

Throw

если определенный период времени проходит без его испускания.

Timeout

Я хочу,чтобы наблюдатель изящно восстановился
от таймаута,переключившись на резервный Наблюдаемый

Timeout

предварительное уведомление об ошибке

Catch

при попытке переподписаться на "Наблюдаемый".

Retry

Я хочу создать ресурс,который будет иметь такую же продолжительность жизни,как и "Наблюдаемый".

Using

Я хочу подписаться на Observable и получить Future , который блокируется, пока Observable не завершится

Start

Я хочу "Наблюдателя",который не начинает излучать элементы подписчикам,пока не спросит.

Publish

а затем испускает только последний элемент в своей последовательности.

PublishLast

а затем излучает полную последовательность,даже тем,кто подписывается после того,как последовательность началась.

Replay

но я хочу,чтобы она исчезла,как только все ее подписчики откажутся от подписки.

RefCount

а потом я хочу попросить его начать

Connect

46. Subjects ReactiveX, виды и примеры

Subject — это своего рода мост или прокси, доступный в некоторых реализациях ReactiveX, который действует как наблюдатель(Observer) и наблюдаемый(Observable).Так как он является наблюдателем(observer), он может подписаться на один и более наблюдаемых(Observables), и потому как он наблюдатель(Observer), он может пройти через все элементы, за которыми он наблюдает, повторно передав их, а также может излучать(emit) новые элементы.

Subject которые есть в RxJava.

- Publish Subject
- Replay Subject
- Behavior Subject
- Async Subject

Observable: Предположим, что профессор является наблюдаемым(observable). Профессор учит какой-то теме.

Observer: Предположим, что студент наблюдатель(observer). Студент слушает(или наблюдает) тему, которую преподает профессор.

Publish Subject

Излучает(emit) все последующие элементы наблюдаемого источника в момент подписки.

Здесь, если студент вошел в аудиторию, он просто хочет слушать с того момента, когда он вошел в аудиторию. И так, Publish будет лучшим выбором для использования.

пример:

```
PublishSubject<Integer> source = PublishSubject.create();

// Получит 1, 2, 3, 4 и onComplete
```

source.subscribe(getFirstObserver());

source.onNext(1);

source.onNext(2);

source.onNext(3);

// Получит 4 и onComplete для следующего наблюдателя тоже.source.subscribe(getSecondObserver());

source.onNext(4);

source.onComplete();

Replay Subject

Излучает(emit все элементы источника наблюдаемого(Observable), независимо от того, когда подписчик(subscriber) подписывается. Здесь, если студент с опозданием вошел в аудиторию, он хочет послушать лекцию с самого начала. И так, для этого мы должны использовать Replay.

пример:

ReplaySubject<Integer> source = ReplaySubject.create();

// Он получит 1, 2, 3, 4

source.subscribe(getFirstObserver());

source.onNext(1);

source.onNext(2);

source.onNext(3);

source.onNext(4);

source.onComplete();

// Он также получит 1, 2, 3, 4 так как он использует Replay Subject

source.subscribe(getSecondObserver());

Behavior Subject

Он излучает(emit) совсем недавно созданый элемент и все последующие элементы наблюдаемого источника, когда наблюдатель(observer) присоединяется к нему. Здесь, если студент вошел в аудиторию, он хочет слушать самые последние вещи(не с начала) преподаваемые профессором таким образом, что он получает идею контекста. Итак, здесь мы будем использовать Behavior.

Пример:

```
BehaviorSubject<Integer> source = BehaviorSubject.create();

// Получит 1, 2, 3, 4 and onComplete

source.subscribe(getFirstObserver());

source.onNext(1);

source.onNext(2);

source.onNext(3);

// Получит 3(последний элемент) и 4(последующие элементы) и onComplete

source.subscribe(getSecondObserver());

source.onNext(4);

source.onComplete();
```

Async Subject

Он выдает только последнее значение наблюдаемого источника(и только последнее).

Здесь, если студент пришел в любой момент времени в аудиторию, и он хочет слушать только о последней вещи(и только последней) которую учат. Итак, здесь мы будем использовать Async.

пример:

```
AsyncSubject<Integer> source = AsyncSubject.create();
```

```
// Получит только 4 и onComplete
source.subscribe(getFirstObserver());
source.onNext(1);
source.onNext(2);
source.onNext(3);
// Тоже получит только 4 и onComplete
source.subscribe(getSecondObserver());
source.onNext(4);
source.onComplete();
```

47. Распараллеливание потоков RxJava

Многопоточное приложение состоит из двух или более частей, которые могут работать параллельно. Это позволяет приложению лучше использовать ядра внутри процессора устройства. Это позволяет выполнять задачи быстрее и обеспечивает более плавный и отзывчивый опыт для пользователя.

Кодирование для параллелизма в Java может быть болезненным, но благодаря RxJava теперь это сделать намного проще. С RxJava просто нужно объявить поток, в котором нужно, чтобы задача была выполнена (декларативно) вместо создания и управления потоками (обязательно).

RxJava использует Schedulers вместе с операторами параллельной subscribeOn() на subscribeOn() и observeOn() для достижения этой цели.

Планировщики в RxJava 2

Schedulers в RxJava используются для выполнения единицы работы над потоком. Scheduler предоставляет абстракцию для механизма потоков Android и Java. Если хотите запустить задачу и используете Scheduler для ее выполнения, Scheduler переходит в свой пул потоков (набор потоков, готовых к использованию), а затем запускает задачу в доступном потоке.

Можно указать, что задача должна выполняться в одном конкретном потоке. (Существует два оператора, subscribeOn() и observeOn() , которые можно использовать для указания, в каком потоке из пула потоков Scheduler должна быть выполнена задача.)

В Android длительные процессы или задачи с интенсивным использованием процессора не должны выполняться в главном потоке. Если подписка Observer на Observable проводится в основном потоке, любой связанный оператор также будет работать в основном потоке. В случае длительной задачи (например, выполнение сетевого запроса) или задачи с интенсивным использованием ЦП (например, преобразование изображения) это будет блокировать пользовательский интерфейс до завершения задачи, что приведет к ужасному диалоговому окну ANR (приложение не отвечает) и приложение вылетает. Вместо этого эти операторы могут быть переключены на другой поток с observeOn() оператора observeOn() .

Типы планировщиков

Вот некоторые из типов Schedulers доступных в RxJava и RxAndroid чтобы указать тип потока для выполнения задач.

- Schedulers.immediate() : возвращает Scheduler который мгновенно выполняет работу в текущем потоке. Имейте в виду, что это заблокирует текущий поток, поэтому его следует использовать с осторожностью.
- Schedulers.trampoline() : планирование задач в текущем потоке. Эти задачи не выполняются сразу, а выполняются после того, как поток завершил свои текущие задачи. Это отличается от Schedulers.immediate() потому что вместо немедленного выполнения задачи он ожидает завершения текущих задач.
- Schedulers.newThread() : запускает новый поток и возвращает Scheduler для выполнения задачи в новом потоке для каждого Observer . Вы должны быть осторожны при использовании этого, потому что новый поток не используется повторно, а вместо этого уничтожается.
- Schedulers.computation() : это дает нам Scheduler который предназначен для вычислительно-интенсивной работы, такой как преобразование изображений, сложные вычисления и т. Д. Эта операция полностью использует ядра ЦП. Этот Scheduler использует фиксированный размер пула потоков, который зависит от ядер ЦП для оптимального использования. Вы должны быть осторожны, чтобы не

создавать больше потоков, чем доступные ядра ЦП, поскольку это может снизить производительность.

- Schedulers.io() : создает и возвращает Scheduler предназначенный для работы, связанной с вводом-выводом, такой как выполнение асинхронных сетевых вызовов или чтение и запись в базу данных. Эти задачи не загружают процессор или используют Schedulers.computation() .
- Schedulers.single() : создает и возвращает Scheduler и выполняет несколько задач последовательно в одном потоке.
- Schedulers.from(Executor executor) : это создаст Scheduler который будет выполнять задачу или единицу работы с данным Executor .
- AndroidSchedulers.mainThread() : это создаст Scheduler который выполняет задачу в главном потоке приложения Android. Этот тип планировщика предоставляется библиотекой RxAndroid .

48. Для чего нужен оператор subscribeOn()

Используя оператор одновременной subscribeOn() , если указать, что Scheduler должен выполнять операцию в Observable восходящем потоке. Затем он отправит значения Observers используя тот же поток. Теперь давайте посмотрим на практический пример:

```
import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

import android.util.Log;

import io.reactivex.Observable;

import io.reactivex.ObservableOnSubscribe;

import io.reactivex.disposables.Disposable;

import io.reactivex.schedulers.Schedulers;

public class MainActivity extends AppCompatActivity{

private static final String[] STATES = { «Lagos», «Abuja», «Abia»,
```

```
        «Edo», «Enugu», «Niger», «Anambra»};
    private Disposable mDisposable = null;
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Observable<String> observable =
Observable.create(dataSource())
            .subscribeOn(Schedulers.newThread())
                .doOnComplete(() -> Log.d(«MainActivity», «Complete»));
        mDisposable = observable.subscribe(s -> {
        Log.d(«MainActivity», «received » + s + » on thread » +
Thread.currentThread().getName());
        });
    }
    private ObservableOnSubscribe<String> dataSource(){
        return(emitter -> {
        for(String state : STATES){
            emitter.onNext(state);
            Log.d(«MainActivity», «emitting » + state + » on thread » +
Thread.currentThread().getName());
            Thread.sleep(600);
        }
        emitter.onComplete();
```

```
        });
    }
    @Override
    protected void onDestroy() {
        if (mDisposable != null && !mDisposable.isDisposed()) {
        mDisposable.dispose();
        }
       super.onDestroy();
    }
    }
```

В приведенном выше коде есть статический ArrayList который содержит некоторые города. Также есть поле типа Disposable . Мы получаем экземпляр Disposable , вызывая Observable.subscribe() , и будем использовать его позже, когда вызовем метод dispose() для освобождения любых использованных ресурсов. Это помогает предотвратить утечки памяти. dataSource() (который может возвращать данные из удаленного или локального источника базы данных) вернет ObservableOnSubscribe<T> : это необходимо для того, чтобы мы позже создали собственный Observable используя метод Observable.create() .

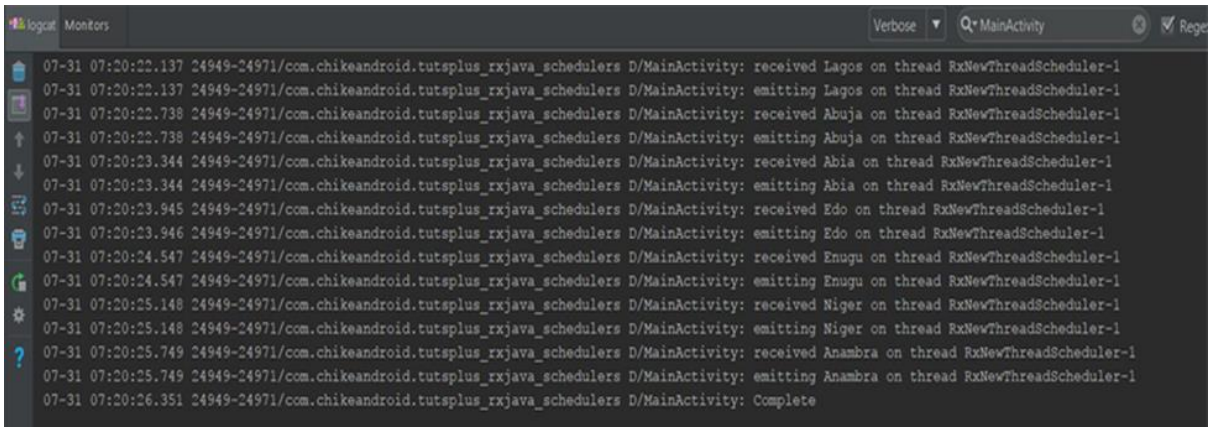
Внутри dataSource()перебираем массив, отправляя каждый элемент Observers , вызывая emitter.onNext() . После выдачи каждого значения спим потоком, чтобы смоделировать выполняемую интенсивную работу. Наконец, вызываем метод onComplete() чтобы сообщить Observers что мы завершили передачу значений, и они больше не должны ожидать.

Теперь dataSource() не должен выполняться в основном потоке пользовательского интерфейса. Но как это указано? В приведенном примере предоставили Schedulers.newThread() в качестве аргумента для subscribeOn() . Это означает, что dataSource() будет запущена в новом потоке. Также обратите внимание, что в приведенном выше примере у нас

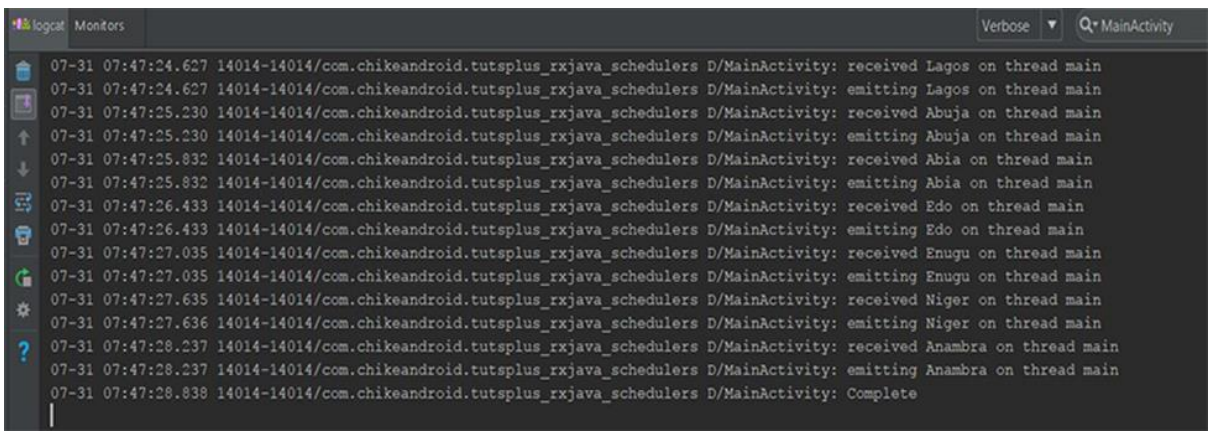
есть только один Observer . Если бы у нас было несколько Observers , каждый из них получал бы свой собственный поток.

Чтобы могли видеть, как это работает, наш Observer распечатывает значения, которые он получает в своем onNext() из Observable .

Когда запустим это и просмотрим наш logcat в Android Studio, видно, что dataSource() метода dataSource() в Observer произошла в том же потоке — RxNewThreadScheduler-1 — в котором Observer их получил.



Если не указать метод .subscribeOn() после метода Observable.create() , он будет выполнен в текущем потоке, который в нашем случае является основным, блокируя таким образом пользовательский интерфейс приложения.



Есть несколько важных деталей, о которых нужно знать, касающихся оператора subscribeOn() . Должен быть только один subscribeOn() в цепочке Observable ; добавление еще одного в любом месте цепочки не будет иметь никакого эффекта вообще. Рекомендуемое место для размещения этого

оператора максимально близко к источнику для ясности. Другими словами, поместите его первым в цепочке операторов.

```
Observable.create(dataSource())
        .subscribeOn(Schedulers.computation()) // this has effect
    .subscribeOn(Schedulers.io()) // has no effect
    .doOnNext(s -> {
        saveToCache(s);
    })
```

Оператор observeOn()

Оператор параллелизма subscribeOn() будет Observable какой Scheduler использовать, чтобы продвигать выбросы по цепочке Observers к Observers .

С другой стороны, observeOn() оператора одновременного observeOn() состоит в том, чтобы переключать последующие выбросы в другой поток или Scheduler . Этот оператор используется, чтобы контролировать, какой поток нижестоящие потребители будут получать выбросы. Давайте посмотрим на практический пример.

```
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.TextView;
import io.reactivex.Observable;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.android.schedulers.AndroidSchedulers;
```

```
import io.reactivex.disposables.Disposable;
import io.reactivex.schedulers.Schedulers;
public class ObserveOnActivity extends AppCompatActivity {
    private Disposable mDisposable = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView textView = (TextView) findViewById(R.id.tv_main);
        Observable<String> observable = Observable.create(dataSource())
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .doOnComplete(() -> Log.d(«ObserveOnActivity», «Complete»));
        mDisposable = observable.subscribe(s -> {
          Log.d(«ObserveOnActivity», «received » + s + » on thread » + Thread.currentThread().getName());
            textView.setText(s);
        });
    }
    private ObservableOnSubscribe<String> dataSource() {
        return(emitter -> {
            Thread.sleep(800);
            emitter.onNext(«Value»);
```

```
            Log.d(«ObserveOnActivity», «dataSource() on thread » + Thread.currentThread().getName());

            emitter.onComplete();

        });

    }

    // ...

    }
```

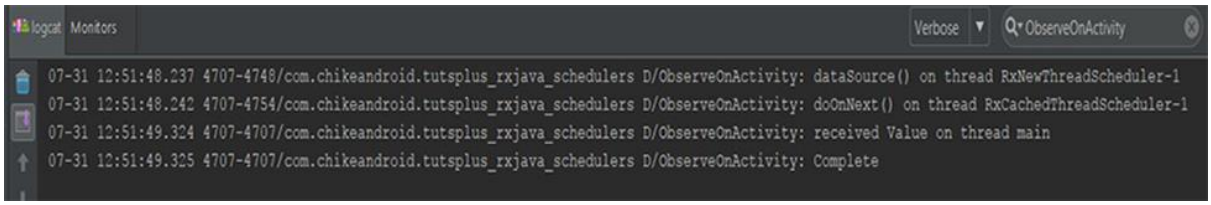
В приведенном коде использовали оператор observeOn() а затем передали ему AndroidSchedulers.mainThread() . Переключили поток с Schedulers.newThread() на основной поток Android. Это необходимо, потому что хотим обновить виджет TextView и можем это делать только из основного потока пользовательского интерфейса. Обратите внимание, что если не переключится на основной поток при попытке обновить виджет TextView , приложение завершится CalledFromWrongThreadException и CalledFromWrongThreadException .

В отличие от оператора subscribeOn() оператор observeOn() может применяться несколько раз в цепочке операторов, тем самым изменяя Scheduler более одного раза.

```
    Observable<String> observable = Observable.create(dataSource())

            .subscribeOn(Schedulers.newThread())

            .observeOn(Schedulers.io())

            .doOnNext(s -> {

                saveToCache(s);

                Log.d(«ObserveOnActivity», «doOnNext() on thread » + Thread.currentThread().getName());

            })

            .observeOn(AndroidSchedulers.mainThread())
```

.doOnComplete(() -> Log.d(«ObserveOnActivity», «Complete»));

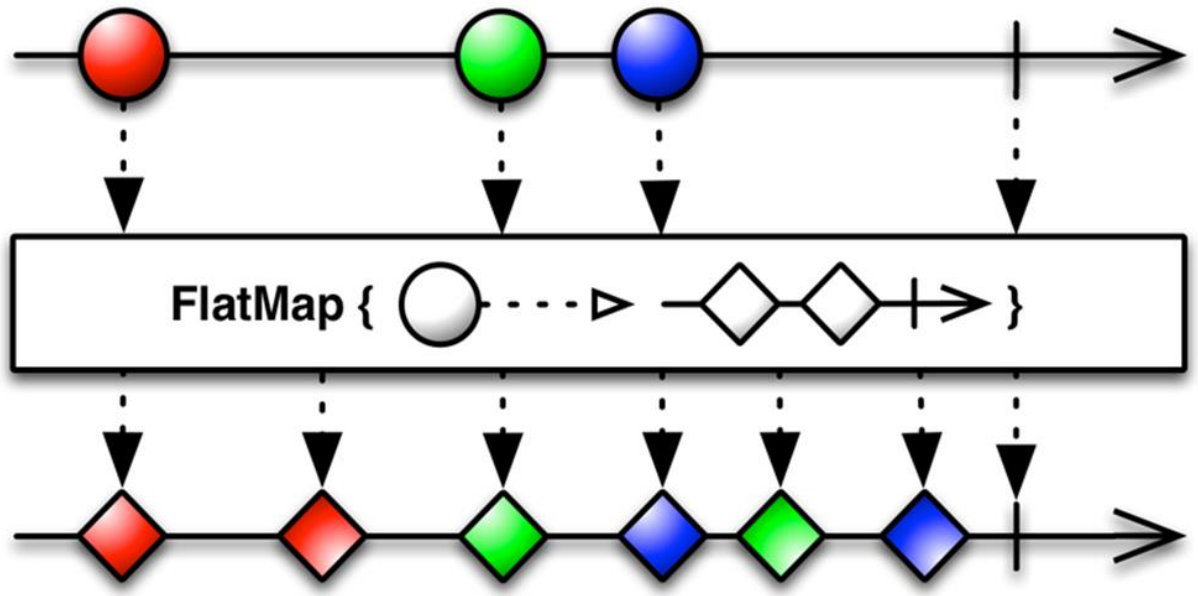
Этот код имеет два оператора observeOn() . Первый использует Schedulers.io() , что означает, что метод saveToCache() будет выполняться в потоке Schedulers.io() . После этого он переключается на AndroidSchedulers.mainThread() где Observers будут получать выбросы от восходящего потока.



49. Параллелизм с оператором flatMap()

Оператор flatMap() — это еще один очень мощный и важный оператор, который можно использовать для достижения параллелизма. Определение в соответствии с официальной документацией выглядит следующим образом:

Преобразуйте предметы, испускаемые Обсерваторией, в Наблюдаемые, затем сведите выбросы из них в единую Наблюдаемую.



\Посмотрим на практический пример, который использует этот оператор:

```
//...

@Override

protected void onCreate(Bundle savedInstanceState){

    // ...

final String[] states = {«Москва», «Ярославль», «Пушкино», «Н.Новгород»};

Observable<String> statesObservable = Observable.fromArray(states);

statesObservable.flatMap(

        s -> Observable.create(getPopulation(s))

 ).subscribe(pair -> Log.d(«MainActivity», pair.first + » population is » + pair.second));

}

  private ObservableOnSubscribe<Pair> getPopulation(String state){

    return(emitter -> {

      Random r = new Random();

      Log.d(«MainActivity», «getPopulation() for » + state + » called on » + Thread.currentThread().getName());

      emitter.onNext(new Pair(state, r.nextInt(300000 — 10000) + 10000));

      emitter.onComplete();

    });

}

}
```

В Android Studio logcat будет напечатано следующее:

getPopulation() for Москва called on main

Москва population is 80362

getPopulation() for Ярославль called on main

Ярославль population is 132559

getPopulation() for Пушкино called on main

Пушкино population is 34106

getPopulation() for Н.Новгород called on main

Н.Новгород population is 220301

Из приведенного выше результата видно, что полученные результаты были в том же порядке, что и в массиве. Кроме того, метод getPopulation() для каждого состояния обрабатывался в одном и том же потоке — основном потоке. Это замедляет вывод, потому что они обрабатывались последовательно в главном потоке.

Теперь, чтобы достичь параллелизма с этим оператором, мы хотим, чтобы метод getPopulation() для каждого состояния (эмиссии из statesObservable) обрабатывался в разных потоках. Это приведет к более быстрой обработке. Для этого мы будем использовать оператор flatMap() , поскольку он создает новую Observable для каждого излучения. Затем мы применяем к нему один оператор параллелизма subscribeOn() , передавая ему Scheduler .

```
statesObservable.flatMap(
        s -> Observable.create(getPopulation(s))
        .subscribeOn(Schedulers.io())
    ).subscribe(pair -> Log.d(«MainActivity», pair.first + » population is » + pair.second));
```

Поскольку каждое излучение создает Observable , flatMap() оператора flatMap() состоит в том, чтобы объединить их вместе и затем отправить их в виде единого потока.

getPopulation() for Lagos called on RxCachedThreadScheduler-1

Lagos population is 143965

getPopulation() for Abuja called on RxCachedThreadScheduler-2

getPopulation() for Enugu called on RxCachedThreadScheduler-4

Abuja population is 158363

Enugu population is 271420

getPopulation() for Imo called on RxCachedThreadScheduler-3

Imo population is 81564

В приведенном выше результате мы можем наблюдать, что метод getPopulation() каждого состояния обрабатывался в разных потоках. Это значительно ускоряет обработку, но также flatMap() что выбросы от оператора flatMap() , полученные Observer , не соответствуют порядку исходных выбросов в восходящем направлении.

50. Интерфейс Flowable 51. Backpressure RxJava

Противодавление — это процесс обработки быстрого производителя предметов, например. когда конвейер обработки Observable не может обрабатывать значения достаточно быстро и может столкнуться с исключением Out of Memory , тогда нам нужен способ сообщить вышестоящему производителю о замедлении, этот механизм называется Backpressuring или Backpressure . Проще говоря, мы можем сказать, что когда у нас слишком много данных, передаваемых наблюдателям, и наблюдатели не могут их обработать и сталкиваются с исключением Out of Memory , чтобы предотвратить это, мы должны (обратное давление) сказать вышестоящему производителю замедлить работу. .

Flowable похож на Observable. Есть два типа классов объектов: Observable и Flowable . Observable

Разница между Observable и Flowable: Противодавление было причиной того, что Flowable был введен в RxJava 2.x, поскольку основное различие между Observable и Flowable заключается в том, что Flowables осведомлены о противодавлении, а Observable — нет . Класс Observableимеет неограниченный размер буфера , т.е. Наблюдаемый объект будет буферизовать все и отдавать все подписчику в один момент времени, поэтому в то время, когда Observableon был введен в RxJava 0.x, обратное давление не было включено в него, однако с RxJava 2.x у нас есть новый Observableобъектный класс, называемый Текучесть с включенным противодавлением.

Observable используются, когда у нас есть относительно небольшое количество объектов, которые нужно генерировать с течением времени (<1000), и нет риска , что производитель переполнит потребителей, что может привести кOutOfMemoryException.

Flowable используются, когда у нас относительно большое количество элементов, и нам нужно тщательно контролировать их Producerповедение, чтобы избежать исчерпания ресурсов или перегрузки.

Терминология обратного давления:

- Горячие наблюдаемые/источники
- Холодные наблюдаемые/источники
- Многоадресные наблюдаемые/источники

Горячие наблюдаемые или горячий источник:

Когда Observables непрерывно испускает объекты в направлении Observers, независимо от того, могут ли Observers обработать это или нет, такие Observables называются Hot Observables или hot sources . Hot Observables обычно плохо справляются с моделью реактивного извлечения и являются лучшими кандидатами для некоторых других стратегий управления потоком, обсуждаемых ниже.

Холодные наблюдаемые или холодные источники:

Когда Observables испускают объекты только тогда, когда этого хотят наблюдатели, или мы можем сказать, что объекты, испускаемые Observables, не нуждаются в буферизации , поскольку Observables испускают объект только тогда, когда он нужен наблюдателям, мы также можем назвать это ленивой загрузкой, и этот вид наблюдаемых называется холодными наблюдаемыми или холодными источниками . Холодные наблюдаемые противоположны горячим наблюдаемым.

Многоадресные наблюдаемые:

Когда холодный Observable является многоадресным (когда он преобразуется в a ConnectableObservableи connect()вызывается его метод), он фактически становится горячим , и для целей противодавления и управления потоком его следует рассматривать как горячий Observable.

Предотвращение противодавления:

Чтобы предотвратить обратное давление, мы преобразуем Observable в Flowable с помощью observable.toFloawable()метода и указываем с ним стратегию буферизации.

Стратегии обратного давления

Противодавление встречается на горячих Observables с MissingBackPressureException ie. использование без противодействия или пользовательских операторов PublishProcessor, созданных с помощью , которые происходят в операторе, мы уже знаем, что для предотвращения этого мы используем Flowable и указываем с ним стратегию буферизации.timer()interval()create()observeOn

- Увеличение размеров буфера. Переполнение объектов происходит в наблюдателях из-за скачкообразных источников, пользователь приложения может очень быстро нажимать на представления, поэтому

наблюдатели не получают достаточно времени для использования испускаемых объектов из-за этого из RxJava 2.x, чтобы иметь достаточно место для всех значений, программисты могут указать размер своих внутренних буферов, используя чувствительные к обратному давлению операторы с соответствующими параметрами: bufferSize, prefetchили capacityHint. ниже пример. взято из документации RxJava 2.x.

```
PublishProcessor<Integer> source = PublishProcessor.create();

   source.observeOn(Schedulers.computation(), 1024 * 1024)

        .subscribe(e -> {}, Throwable::printStackTrace);

        for (int i = 0; i < 1_000_000; i++){

        source.onNext(i);

        }
```

Это нельзя рассматривать как постоянное исправление, так как в случае, если источник превышает прогнозируемый размер буфера, может произойти переполнение, для такого сценария мы можем использовать другие операторы, описанные ниже.

· Пакетирование/пропуск значений со стандартными операторами

Чтобы уменьшить вероятность того , что MissingBackpressureExceptionдаже после определения bufferSize, у нас все еще есть возможность использовать стандартные операторы пакетирования (по размеру и/или по времени).

```
PublishProcessor<Integer> source = PublishProcessor.create();

        source

        .buffer(1024)

        .observeOn(Schedulers.computation(), 1024)

        .subscribe(list -> {

           list.parallelStream().map(e -> e * e).first();

        }, Throwable::printStackTrace );
```

```
for (int i = 0; i < 1_000_000; i++){
source.onNext(i);
}
```

В тех случаях, когда объекты/значения можно безопасно игнорировать, у нас также есть дросселирующие операторы (throttleFirst, throttleLast, throttleWithTimeout), однако эти операторы только снижают скорость получения значения нисходящим потоком и, таким образом, они все еще могут привести к MissingBackpressureException.

- onBackpressureBuffer()

Этот оператор без параметров повторно вводит неограниченный буфер между восходящим источником и нижестоящим оператором. Неограниченность означает, что пока у JVM не заканчивается память, она может обрабатывать почти любой объем, поступающий из неустойчивого источника. например.

```
Flowable.range(1, 1_000_000)
    .onBackpressureBuffer()
    .observeOn(Schedulers.computation(), 8)
        .subscribe(e -> {}, Throwable::printStackTrace);
```

Один и тот же оператор можно использовать по-разному, добавив к нему определенные параметры:

- onBackpressureBuffer (целая емкость)
- onBackpressureBuffer (целая емкость, действие при переполнении)
- onBackpressureBuffer (целая емкость, действие при переполнении, стратегия BackpressureOverflowStrategy)

52. Разница между Observable и Flowable

К сожалению, до Rx 2.0 внедрение Backpressure было произведено непосредственно в Observable, вместо того, чтобы выделить отдельный класс с его поддержкой. Основная проблема с Backpressure заключается в том, что многие горячие Observable, не могут быть достаточно надежными, и при определенных обстоятельствах могут вызвать неожиданный MissingBackpressureException. А без определенного наработанного опыта очень сложно предсказать такое исключение.
В Rx 2.0 исправлена эта ситуация. Observable — класс без покрытия backpressure, а новый Flowable был наделен backpressure из коробки. Далее мы рассмотрим вариации, где и в каком кейсе использовать Flowable, а где Observable.

Так какой когда юзать?

Этот вопрос, пожалуй, самый логичный. При реализации ваших классов-репозиториев, классов бизнес-логики, при принятии решения о том, какой тип, Observable или Flowable, следует принимать и возвращать, вам стоит рассмотреть несколько факторов, которые помогут вам избежать проблем в получении исключений таких, как MissingBackpressureException или OutOfMemoryError, т.к. неаккуратное использование неправильного типа ведет к падению fps на перфомансе.

Когда юзать Observable

В тех кейсах, когда у вас в итерации, условно говоря, не более 1000 элементов, при этом это самый худший случай и не предполагается масштабирования, вам не требуется backpressure. В целом, можно сказать, что, если вы чувствуете, что в каком-то определенном кейсе нет шансов на OutOfMemoryException, то это ровно тот случай, когда можно и нужно использовать Observable. В основном, это случаи использования UI, самые разнообразные события onClick, Touch, Pointer-movement, и.т.п. По сути, это любые события, частота которых не превышает 1000 Hz. Согласитесь, врядли вы сможете нажимать на тачскрин чаще 1000 раз в секунду. Но все равно не забывайте по оператор debounce.

```
class MainActivity : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


        RxView.clicks(backpressure)
                .debounce(200, TimeUnit.MILLISECONDS)
                .subscribe({ openNewScreen(BackpressureExampleActivity::class.java) }, Throwable::printStackTrace)


        RxView.clicks(nullpointer)
                .debounce(200, TimeUnit.MILLISECONDS)
                .subscribe({ openNewScreen(NullPointerActivity::class.java) }, Throwable::printStackTrace)
    }
}
```

RxBinding я думаю всем известно, что библиотека Jake Wharton'a возвращает Observable при биндинге View, и это полностью оправдывает концепцию Rx2.x

Когда юзать Flowable?

Когда вы имеете дело уже с большими или непредсказуемыми объемами данных, скажем свыше 10000 элементов, или в ситуациях с непрерывной генерацией данных. Также поводом для использования Flowable является парсинг, чтение данных с разнообразных носителей информации(Internal/External storage).

Чтение из базы данных при помощи SQLiteOpenHelper тоже является предлогом для использования Flowable. Поэтому, если вы используете библиотеки https://github.com/square/sqlbrite и https://github.com/pushtorefresh/storio, не будет лишним приводить Observable к Flowable. Выполнение запросов к бэкенду — еще одна причина для использования Flowable.

Если присмотреться, можно заметить общую деталь у всех перечисленных кейсов — они блокируют UI-поток. Соответственно, если вы выполняете какие-то операции блокирующие MainThread — это повод использовать Flowable.

Ну и напоследок. Многие блокирующие и/или pull-based источники данных, от которых в конечном итоге можно получить неблокирующее реактивное API(Retrofit) также являются основанием для использования Flowable.

53. Протокол Rsocket, принцип работы

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.

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 односторонее сообщение и не ждет ответа

Fire-and-Forget:

Fire-and-forget предназначен для передачи данных от отправителя к получателю, в котором отправитель не заботится о результате операции — он отправил запрос и забыл о нем. Такое упрощенное сообщение может быть полезно при отправке уведомлений на мобильную связь например.

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

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.

Основные преимущества

Можем сделать простой CRUD — нет проблем, т.к. есть простой реквест-респонс.

Можем сделать более advanced CRUD — нет проблем. Например мы пушим какие-то метрики на другой сервер и нам все равно, удлятся они или нет, главное нам отправить их в сеть и забыть о них. Там используется Fire and Forget — отправляем сообщение и скорей забываем про него без ожидания окончания процессинга.

Есть стриминг: можем запросить стрим сервера и сервер отправит данные или мы можем сами про помощи Stream-Stream или Request-Channel Communication пушить данные или пушить стрим данных и сервер нам тоже может отвечать (пушить нам какой-то стрим даных).

Поддержка RSocket практически на всех языках.

Можете использовать или RPC архитектуру или Messaging, не важно.

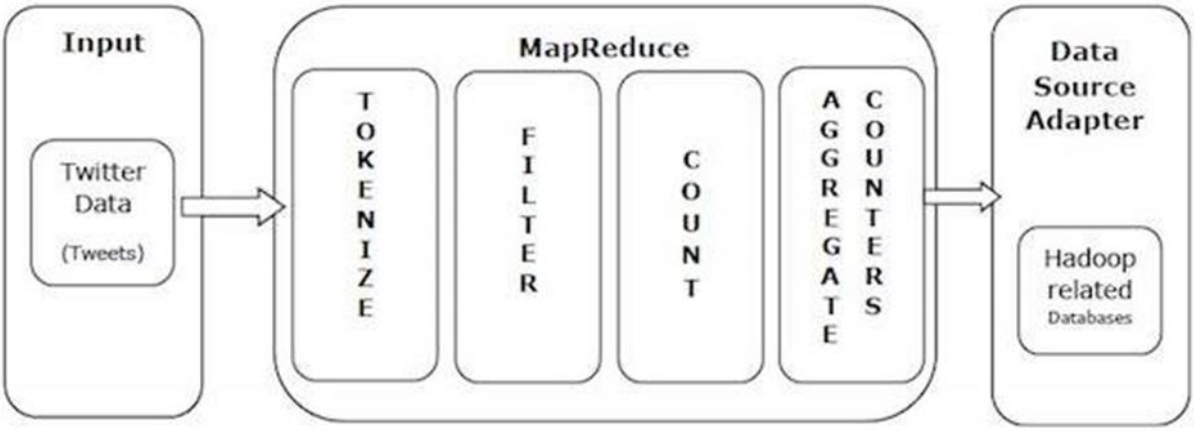
Вы можете использовать RSocket поверх любого транспортного протокола (WebSocket, HTTP.2, TCP и другие).

54. MapReduce это? Пример

MapReduce – это модель программирования для написания приложений, которые могут обрабатывать большие данные параллельно на нескольких узлах. MapReduce предоставляет аналитические возможности для анализа огромных объемов сложных данных.

MapReduce-пример

Пример из реальной жизни, чтобы понять мощь MapReduce. Twitter получает около 500 миллионов твитов в день, то есть почти 3000 твитов в секунду. На рисунке показано, как Tweeter управляет своими твитами с помощью MapReduce.



Как показано на рисунке, алгоритм MapReduce выполняет следующие действия:

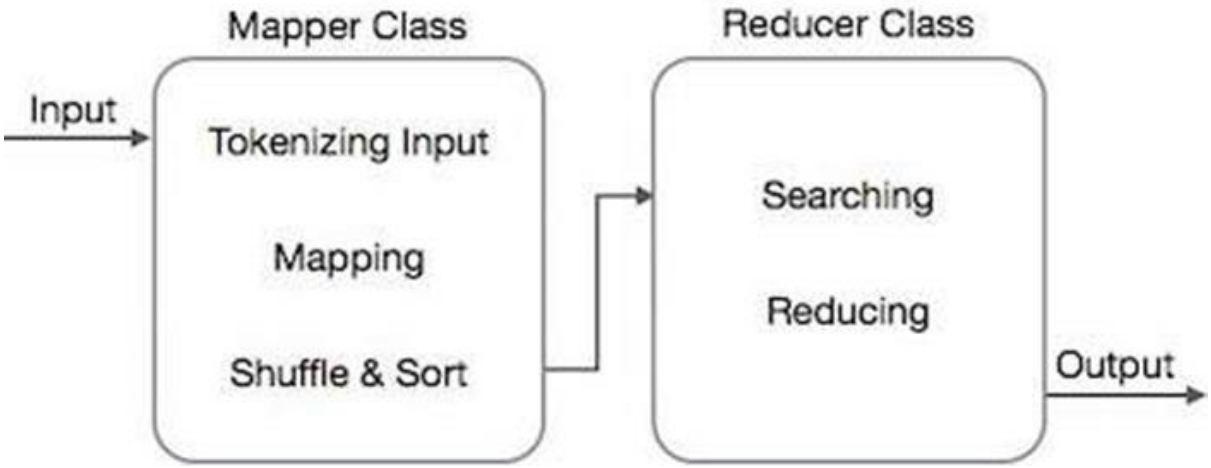
- **Tokenize** – токенизирует твиты в карты токенов и записывает их в виде пар ключ-значение.
- **Фильтр** – Фильтрует нежелательные слова из карт токенов и записывает отфильтрованные карты в виде пар ключ-значение.
- **Count** – генерирует счетчик токенов на слово.
- **Сводные счетчики** – готовит совокупность аналогичных значений счетчиков в небольшие управляемые единицы.

MapReduce – Алгоритм

Алгоритм MapReduce содержит две важные задачи, а именно Map и Reduce.

- Задача карты выполняется с помощью Mapper Class
- Задача уменьшения выполняется с помощью класса редуктора.

Класс Mapper принимает входные данные, маркирует их, отображает и сортирует их. Выходные данные класса Mapper используются в качестве входных данных классом Reducer, который, в свою очередь, ищет совпадающие пары и сокращает их.



MapReduce реализует различные математические алгоритмы, чтобы разделить задачу на маленькие части и назначить их нескольким системам. С технической точки зрения алгоритм MapReduce помогает отправлять задачи Map & Reduce на соответствующие серверы в кластере.

Эти математические алгоритмы могут включать в себя следующее:

- Сортировка
- поиск
- индексирование
- TF-IDF

<u>Сортировка</u>

Сортировка является одним из основных алгоритмов MapReduce для обработки и анализа данных. MapReduce реализует алгоритм сортировки для автоматической сортировки выходных пар ключ-значение из преобразователя по их ключам.

- Методы сортировки реализованы в самом классе mapper.

- На этапе перемешивания и сортировки после токенизации значений в классе сопоставления класс **Context** (определенный пользователем класс) собирает совпадающие значения ключей в виде коллекции.
- Чтобы собрать похожие пары ключ-значение (промежуточные ключи), класс Mapper использует класс **RawComparator** для сортировки пар ключ-значение.
- Набор промежуточных пар ключ-значение для данного редуктора автоматически сортируется Hadoop для формирования значений ключа (K2, {V2, V2,…}) до их представления редуктору.

Поиск

Поиск играет важную роль в алгоритме MapReduce. Это помогает в фазе объединителя (опция) и в фазе редуктора.

55. Что такое большие данные?

Большие данные – это набор больших наборов данных, которые не могут быть обработаны с использованием традиционных вычислительных технологий. Например, объем данных, которые Facebook или Youtube должны требовать для ежедневного сбора и обработки, может подпадать под категорию больших данных. Однако большие данные касаются не только масштаба и объема, но и одного или нескольких из следующих аспектов – скорость, разнообразие, объем и сложность.

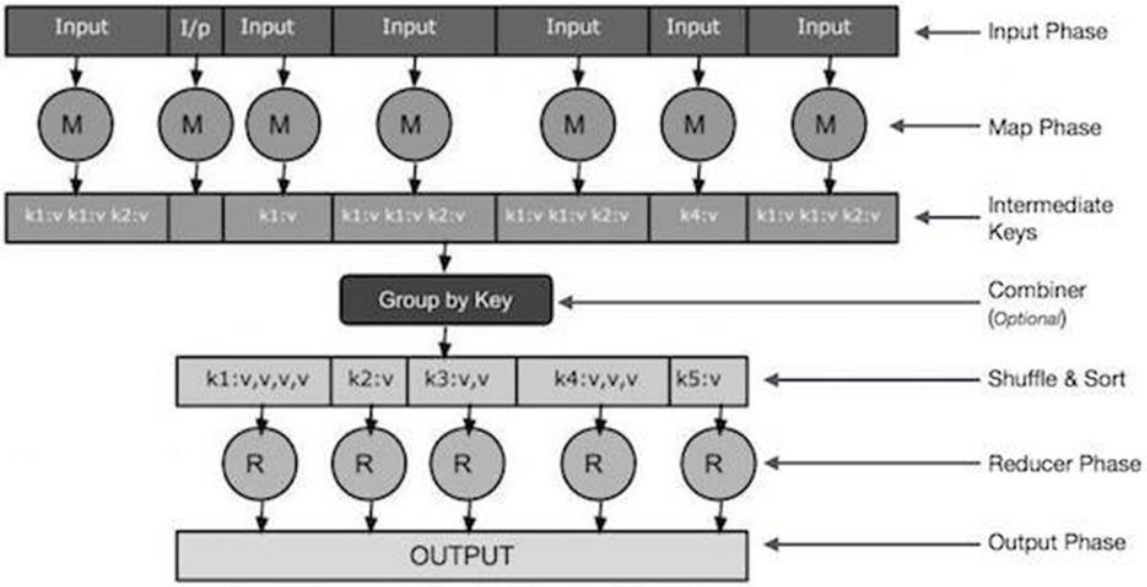
56. Как работает MapReduce?

Алгоритм MapReduce содержит две важные задачи, а именно Map и Reduce.

- Задача «Карта» берет набор данных и преобразует его в другой набор данных, где отдельные элементы разбиваются на кортежи (пары ключ-значение).
- Задача Reduce принимает выходные данные из карты в качестве входных данных и объединяет эти кортежи данных (пары ключ-значение) в меньший набор кортежей.

Задача уменьшения всегда выполняется после задания карты.

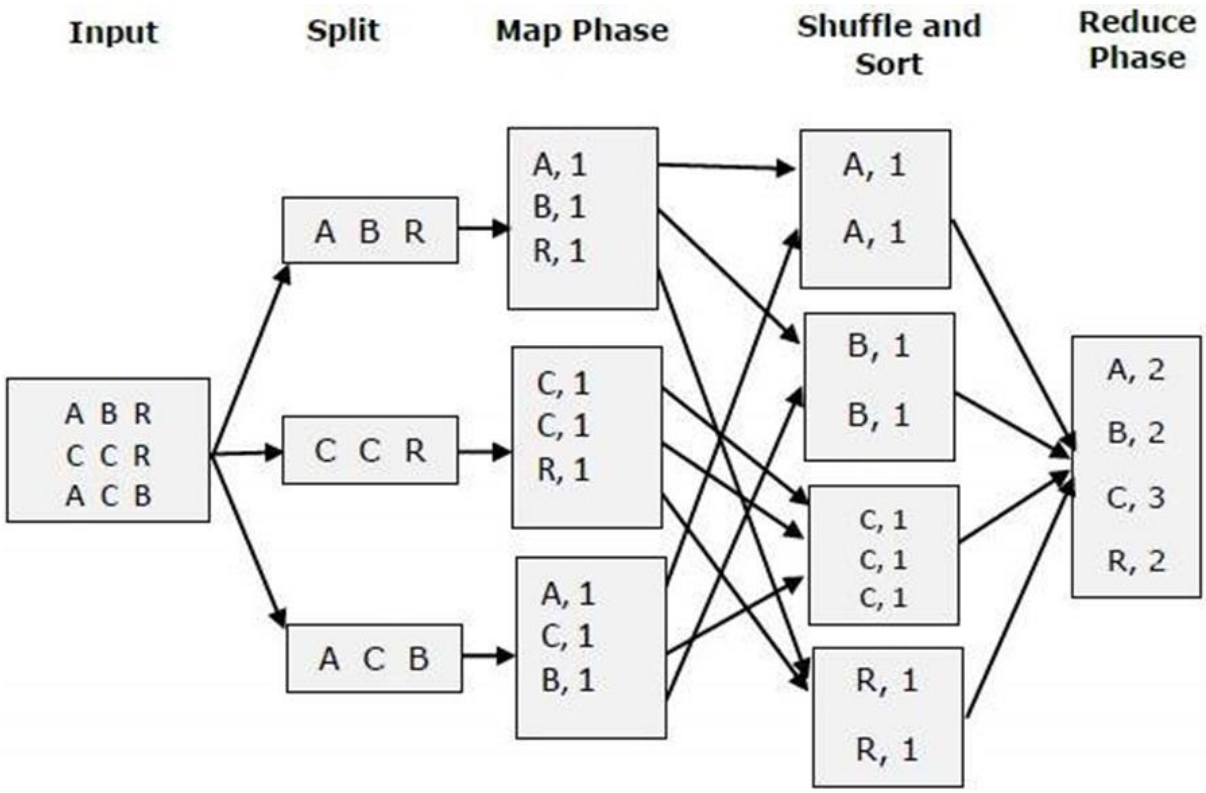
57. Этапы MapReduce



- **Фаза ввода** – Здесь есть Record Reader, который переводит каждую запись во входной файл и отправляет проанализированные данные в маппер в виде пар ключ-значение.
- **Карта** –это пользовательская функция, которая принимает серию пар ключ-значение и обрабатывает каждую из них, чтобы сгенерировать ноль или более пар ключ-значение.
- **Промежуточные ключи** – пары «ключ-значение», генерируемые картографом, называются промежуточными ключами.
- **Объединитель** – это тип локального редуктора, который группирует аналогичные данные из фазы карты в идентифицируемые наборы. Он принимает промежуточные ключи от преобразователя в качестве входных данных и применяет пользовательский код для агрегирования значений в небольшой области одного преобразователя. Он не является частью основного алгоритма MapReduce; это необязательно.
- **Перемешать и отсортировать** – задача «Восстановитель» начинается с шага «Перемешать и сортировать». Он загружает сгруппированные пары ключ-значение на локальный компьютер, на котором работает редуктор. Отдельные пары ключ-значение сортируются по ключу в больший список данных. Список данных группирует эквивалентные ключи вместе, так что их значения могут быть легко повторены в задаче Reducer.

- **Редуктор** –принимает сгруппированные парные данные ключ-значение в качестве входных данных и запускает функцию Редуктор для каждого из них. Здесь данные могут быть агрегированы, отфильтрованы и объединены различными способами, что требует широкого спектра обработки. Как только выполнение закончено, он дает ноль или более пар ключ-значение для последнего шага.
- **Фаза вывода.** На этапе вывода у нас есть выходной форматер, который переводит конечные пары ключ-значение из функции Reducer и записывает их в файл с помощью средства записи.

Теперь попробуем понять две задачи Map & fReduce с помощью небольшой диаграммы



58. Интерфейс JobContext

Интерфейс JobContext – это суперинтерфейс для всех классов, который определяет различные задания в MapReduce. Это дает доступ только для чтения к заданию, которое предоставляется задачам во время их выполнения.

Приведены подинтерфейсы интерфейса JobContext.

S.No.	Подинтерфейс Описание
1.	**MapContext <KEYIN, VALUEIN, KEYOUT, VALUEOUT>** Определяет контекст, который предоставляется Mapper.
2.	**ReduceContext <KEYIN, VALUEIN, KEYOUT, VALUEOUT>** Определяет контекст, который передается редуктору.

Класс Job – это основной класс, который реализует интерфейс JobContext.

Класс работы

Класс Job является наиболее важным классом в API MapReduce. Это позволяет пользователю настраивать задание, отправлять его, контролировать его выполнение и запрашивать состояние. Методы set работают только до отправки задания, после чего они генерируют исключение IllegalStateException. Обычно пользователь создает приложение, описывает различные аспекты задания, а затем отправляет задание и отслеживает его выполнение.Вот пример того, как подать заявку –

```
// Create a new Job

Job job = new Job(new Configuration());

job.setJarByClass(MyJob.class);



// Specify various job-specific parameters

job.setJobName("myjob");

job.setInputPath(new Path("in"));
```

```
job.setOutputPath(new Path("out"));


job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);


// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

Конструкторы

Ниже приводится краткое описание конструктора класса Job.

S.No	Сводка конструктора
1	**Работа** ()
2	**Job** (Конфиг конфигурации)
3	**Job** (Конфигурация conf, String jobName)

методы

Вот некоторые из важных методов класса Job:

S.No	Описание метода
1	**getJobName ()** Заданное пользователем имя задания.
2	**getJobState ()** Возвращает текущее состояние задания.

3	**завершено()** Проверяет, закончена ли работа или нет.
4	**setInputFormatClass ()** Устанавливает InputFormat для работы.
5	**setJobName (имя строки)** Устанавливает заданное пользователем имя задания.
6	**setOutputFormatClass ()** Устанавливает формат вывода для работы.
7	**setMapperClass (класс)** Устанавливает Mapper для работы.
8	**setReducerClass (класс)** Устанавливает Редуктор для работы.
9	**setPartitionerClass (класс)** Устанавливает Partitioner для работы.
10	**setCombinerClass (класс)** Устанавливает Combiner для работы.

59. Mapper Class

Класс Mapper определяет задание Map. Сопоставляет входные пары ключ-значение с набором промежуточных пар ключ-значение. Карты – это отдельные задачи, которые преобразуют входные записи в промежуточные записи. Преобразованные промежуточные записи не обязательно должны быть того же типа, что и входные записи. Заданная входная пара может отображаться на ноль или на множество выходных пар.

метод

карта является наиболее известным методом класса Mapper. Синтаксис представлен

```
map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)
```

Этот метод вызывается один раз для каждой пары ключ-значение во входном разбиении.

60. Класс Reducer

Класс Reducer определяет задание Reduce в MapReduce. Это уменьшает набор промежуточных значений, которые разделяют ключ, до меньшего набора значений. Реализации редуктора могут получить доступ к Конфигурации для задания через метод JobContext.getConfiguration (). Редуктор имеет три основных этапа – перемешивание, сортировка и уменьшение.

- Перемешать – Редуктор копирует отсортированный вывод из каждого Mapper, используя HTTP по всей сети.
- Сортировать – платформа объединяет сортировку входов Редуктора по ключам (поскольку разные Mappers могут выводить один и тот же ключ). Фазы тасования и сортировки происходят одновременно, т. Е. Во время выборки выходов они объединяются.
- Сокращение – На этом этапе метод Redu (Object, Iterable, Context) вызывается для каждого <ключа, (набора значений)> в отсортированных входных данных.

метод

Редукция – самый известный метод класса Редукторов. Синтаксис представлен–

```
reduce (KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)
```

Этот метод вызывается один раз для каждого ключа в коллекции пар ключ-значение.

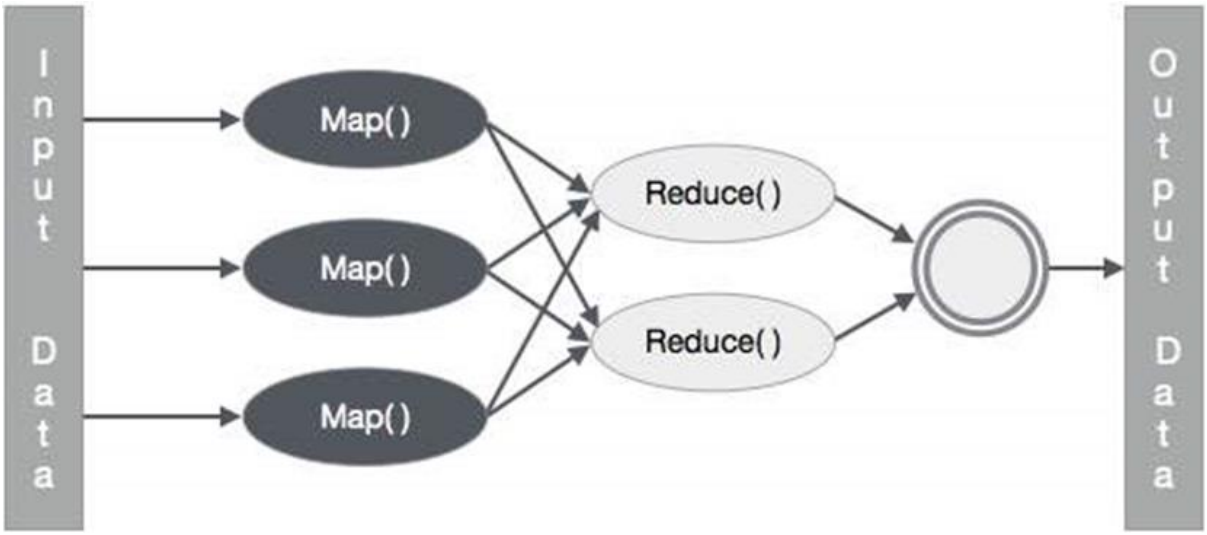
61. MapReduce – реализация Hadoop

MapReduce – это инфраструктура, которая используется для написания приложений для надежной обработки огромных объемов данных на больших кластерах аппаратного оборудования.

62. Алгоритм MapReduce

Обычно парадигма MapReduce основана на отправке программ сокращения карт на компьютеры, где хранятся фактические данные.

- Во время задания MapReduce Hadoop отправляет задачи Map и Reduce на соответствующие серверы в кластере.
- Каркас управляет всеми деталями передачи данных, такими как выдача задач, проверка выполнения задач и копирование данных вокруг кластера между узлами.
- Большая часть вычислений происходит на узлах с данными на локальных дисках, что снижает сетевой трафик.
- После выполнения заданной задачи кластер собирает и сокращает данные, чтобы сформировать соответствующий результат, и отправляет их обратно на сервер Hadoop.



Входы и выходы (перспектива Java)

Каркас MapReduce работает с парами ключ-значение, то есть каркас рассматривает входные данные для задания в виде набора пар ключ-значение и создает набор пар ключ-значение в качестве выходных данных задания, предположительно различных типов.

Классы ключа и значения должны быть сериализуемы платформой, и, следовательно, необходимо реализовать интерфейс Writable. Кроме того, ключевые классы должны реализовывать интерфейс WritableComparable для облегчения сортировки в рамках.

Оба формата ввода и вывода задания MapReduce представлены в виде пар ключ-значение

	вход	**Выход**
карта	<k1, v1>	список (<k2, v2>)
уменьшить	<k2, список (v2)>	список (<k3, v3>)

Реализация MapReduce

В таблице приведены данные, касающиеся потребления электроэнергии в организации. Таблица включает ежемесячное потребление электроэнергии и среднегодовое значение за пять лет подряд.

	январь	февраль	март	апрель	май	июнь	июль	август	сентябрь	октябрь	ноябрь	декабрь	в среднем
2017	23	23	2	43	24	25	26	26	26	26	25	26	25
2018	26	27	28	28	28	30	31	31	31	30	30	30	29
2019	31	32	32	32	33	34	35	36	36	34	34	34	34
2020	39	38	39	39	39	41	42	43	40	39	38	38	40
2021	38	39	39	39	39	41	41	41	00	40	39	39	45

Нужно написать приложение для обработки входных данных в данной таблице, чтобы найти год максимального использования, год минимального использования и так далее. Эта задача проста для программистов с конечным количеством записей, поскольку они просто напишут логику для получения требуемого вывода и передадут данные в написанное приложение. Теперь поднимем масштаб входных данных. Предположим, мы должны проанализировать потребление электроэнергии всеми крупными отраслями конкретного государства. Когда мы пишем приложение для обработки таких массовых данных,

- Они займут много времени, чтобы выполнить.
- При переносе данных из источника на сетевой сервер будет большой сетевой трафик.

Входные данные

Приведенные выше данные сохраняются как **sample.txt** и передаются в качестве входных данных. Входной файл выглядит так, как показано ниже.

2017	23	23	2	43	24	25	26	26	26	26	25	26	**25**
2018	26	27	28	28	28	30	31	31	31	30	30	30	**29**
2019	31	32	32	32	33	34	35	36	36	34	34	34	**34**
2022	39	38	39	39	39	41	42	43	40	39	38	38	**40**
2021	38	39	39	39	39	41	41	41	00	40	39	39	**45**

63. Как работает Combiner? Что такое Combiner? Реализация

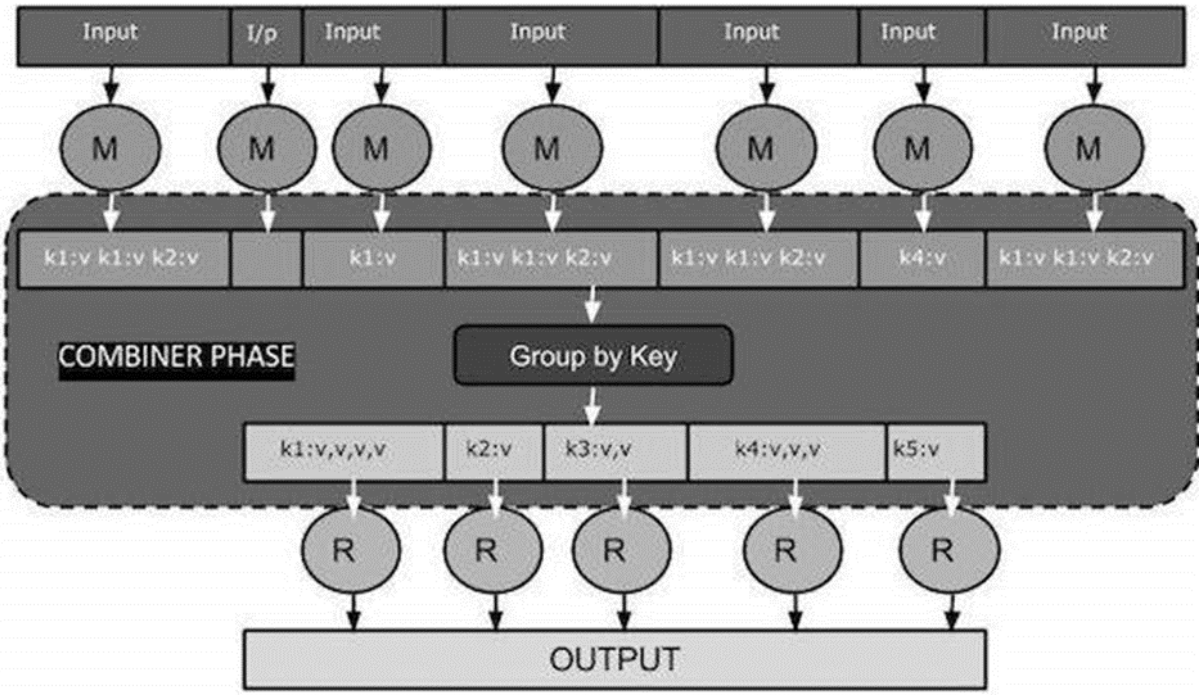
Combiner, также известный как **полуредуктор,** является необязательным классом, который работает, принимая входные данные из класса Map и затем передавая выходные пары ключ-значение в класс Reducer.

Основная функция Combiner состоит в том, чтобы суммировать выходные записи карты с одним и тем же ключом. Выход (сбор значения ключа) объединителя будет отправлен по сети фактической задаче «Редуктор» в качестве входных данных.

Сумматор

Класс Combiner используется между классом Map и классом Reduce для уменьшения объема передачи данных между Map и Reduce. Обычно выходные данные задачи карты большие, а данные, передаваемые в задачу сокращения, большие.

На диаграмме задачи MapReduce показана ФАЗА КОМБИНИРОВАНИЯ.



Как работает Combiner?

Вот краткое описание того, как работает MapReduce Combiner:

- У объединителя нет предопределенного интерфейса, и он должен реализовывать метод Redu () интерфейса Reducer.
- Комбайнер работает с каждым ключом вывода карты. Он должен иметь те же типы значений выходного ключа, что и класс Reducer.
- Объединитель может создавать сводную информацию из большого набора данных, поскольку он заменяет исходный вывод карты.

Хотя Combiner не является обязательным, он помогает разделить данные на несколько групп для фазы сокращения, что упрощает обработку.

Реализация MapReduce Combiner

Следующий пример дает теоретическое представление о комбайнерах. Предположим, у нас есть следующий входной текстовый файл с именем **input.txt** для MapReduce.

```
What do you mean by Object

What do you know about Java

What is Java Virtual Machine

How Java enabled High Performance
```

Record Reader

Это первая фаза MapReduce, где Record Reader считывает каждую строку из входного текстового файла в виде текста и выдает выходные данные в виде пар ключ-значение.

Ввод – построчно текст из входного файла.

Выход – формирует пары ключ-значение. Ниже приведен набор ожидаемых пар ключ-значение.

```
<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>
```

Фаза карты

Фаза Map принимает входные данные из Record Reader, обрабатывает их и создает выходные данные в виде другого набора пар ключ-значение.

Вход – следующая пара ключ-значение – это вход, полученный от устройства чтения записей.

```
<1, What do you mean by Object>

<2, What do you know about Java>
```

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

Фаза Map считывает каждую пару ключ-значение, делит каждое слово на значение, используя StringTokenizer, обрабатывает каждое слово как ключ, а счетчик этого слова – как значение. В следующем фрагменте кода показаны класс Mapper и функция map.

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>

{

  private final static IntWritable one = new IntWritable(1);

  private Text word = new Text();



  public void map(Object key, Text value, Context context) throws IOException, InterruptedException

  {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens())

    {

        word.set(itr.nextToken());

        context.write(word, one);

        }

  }

}
```

Вывод – ожидаемый вывод выглядит следующим образом –

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

Фаза Combiner

Фаза Combiner берет каждую пару ключ-значение из фазы Map, обрабатывает ее и создает выходные данные в виде пар **набора ключ-значение**.

Ввод – следующая пара ключ-значение является вводом, взятым из фазы карты.

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

Фаза Combiner считывает каждую пару ключ-значение, объединяет общие слова как ключ и значения как коллекцию. Как правило, код и работа для Combiner аналогичны таковым для Reducer. Ниже приведен фрагмент кода для объявления классов Mapper, Combiner и Reducer.

```
job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);
```

Вывод – ожидаемый вывод выглядит следующим образом –

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>
```

Фаза редуктора

Фаза редуктора берет каждую пару сбора ключ-значение из фазы Combiner, обрабатывает ее и передает выходные данные в виде пар ключ-значение. Обратите внимание, что функциональность Combiner такая же, как и у Reducer.

Вход – следующая пара ключ-значение является входом, взятым из фазы Combiner.

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>
```

Фаза редуктора считывает каждую пару ключ-значение. Ниже приведен фрагмент кода для Combiner.

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>

{

  private IntWritable result = new IntWritable();



  public void reduce(Text key, Iterable<IntWritable> values,Context context) throws IOException, InterruptedException

  {

       int sum = 0;

       for (IntWritable val : values)

    {

       sum += val.get();

       }

       result.set(sum);

       context.write(key, result);

  }

}
```

Выходные данные – ожидаемый выходной сигнал от фазы редуктора следующий –

<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,3>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

Record Writer

Это последняя фаза MapReduce, где Record Writer записывает каждую пару ключ-значение из фазы Reducer и отправляет вывод в виде текста.

Ввод – каждая пара ключ-значение из фазы редуктора вместе с форматом вывода.

Вывод – он дает вам пары ключ-значение в текстовом формате. Ниже приводится ожидаемый результат.

What 3

do 2

you 2

mean 1

by 1

Object 1

know 1

about 1

Java 3

is 1

Virtual 1

Machine 1

How 1

enabled 1

High 1

64. Основы блокчейн технологий

Блокчейн – выстроенная по определенным правилам непрерывная последовательная цепочка блоков (связанный список), содержащих информацию. Связь между блоками обеспечивается не только нумерацией, но и тем, что каждый блок содержит свою собственную хеш-сумму и хеш-сумму предыдущего блока. Для изменения информации в блоке придется редактировать и все последующие блоки. Чаще всего копии цепочек блоков хранятся на множестве разных компьютеров, независимо друг от друга. Это делает крайне затруднительным внесение изменений в информацию, уже включенную в блоки.

Блокчейн (*Block — блок, chain — цепь*) — это децентрализованная база данных, которая предназначена для хранения последовательных блоков с набором характеристик (версия, дата создания, информация о предыдущих действиях в сети).

Основное движение в такой системе происходит с помощью транзакций. Во время транзакции может выполняться какой-то скрипт, либо прописываться некая заметка с данными. То есть слово «транзакция» не равно денежному переводу и скорее обозначает способ обработки информации внутри сети.

В пределах одного блока может храниться несколько тысяч записей (транзакций). Когда память в блоке заканчивается — он закрывается, подписывается и переходит на новый блок в виде хеша или «отпечатка».

Хеш — это некий набор символов, несущий в себе уникальный отпечаток. Он формируется исходя из того, какие транзакции и в каком количестве хранит в себе каждый блок.

В процессе обработки транзакций постоянно проверяются хеши, после чего, словно по пирамиде, система поднимается к последнему хешу, где подтверждается целостность и верность всех предыдущих кодов, чтобы блок закрылся.

**Сеть состоит из блоков, которые можно менять здесь и сейчас, пока они не закрылись. Все записывается в виде транзакций с информацией, которая шифруется как хеши и постоянно хранится в сети в каждом последующем блоке. Если изменить что-то и не найти этому подтверждение у большинства участников, то такие изменения просто не применятся, а блок будет считаться невалидными.*
Простыми словами — в системе больше не получится подделать документы задним числом, как бы этого не хотели даже сотни человек, если общая сеть контролируется миллионами участников.
Отсюда и название Blockchain — все работает по цепочке, последовательно и непрерывно.

65. Принципы блокчейн

Основные принципы блокчейн:

- Децентрализация;
- Безопасность;
- Открытость;
- Неизменность;
- Распределенность;
- Защита;
- Прозрачность;
- Доверие;
- Взаимодействие без посредников.

66. Децентрализация и распределенность

Для поддержки сети требуется постоянная и непрерывная работа нескольких мощных компьютеров. На сегодняшний день блокчейн обладает самой большой вовлеченностью вычислительных систем на планете.

Чем больше людей используют блокчейн, тем мощнее и безопаснее он становится.

Участником сети может стать каждый: достаточно установить официальный кошелек и загрузить полную ноду к себе на диск. С этого момента компьютер станет полноправным узлом в сети.

Например, у одного человека на компьютере стоит копия блокчейна. Есть еще один компьютер, на котором есть еще одна копия блокчейна, и таких десятки тысяч по всему миру. Если какой-либо злоумышленник захочет взломать систему и «нарисовать» себе миллион, мало того, что ему придется пересчитывать все эти блоки самостоятельно, ему еще придется сделать это в каждом компьютере, на каждом узле. А это, конечно же, невозможно — система полностью децентрализована и не имеет управляющих узлов. И с каждым днем таких узлов становится все больше, а шансов на взлом все меньше.

В централизованной системе вся информация хранится на сервере, и, если с ним что-то произойдет — можно попрощаться с ценными данными. В случае с централизованной системой злоумышленникам также проще найти уязвимость, чтобы атаковать главные компьютеры. Это может быть простая брешь в программной безопасности или безответственная работа сотрудника безопасности банка.

Такой большой вопрос безопасности можно решить только с помощью такой же большой сети. А под эти критерии как раз подходит система блокчейн, где каждый блок с информацией копируется одновременно на тысячи устройств без главенствующих в структуре систем.

67. Транзакции в блокчейн

Блокчейн не требует от пользователей ничего: для работы понадобятся лишь два ключа, которые система выдаст при регистрации.

У каждого человека, желающего принять участие в блокчейне, есть свой публичный ключ, которым он подписывает транзакцию (как бы замыкает на ключ и пишет «отправить Васе»), а также приватный ключ, которым он может открыть посылку, которую ему прислал Вася обратно.

Публичный ключ — это некая фраза из цифр и символов, доступная к просмотру всем желающим. Если провести аналогию с биткоином, то публичный ключ — это номер кошелька, который можно отправить кому угодно для перевода средств.

Приватный ключ — это самое ценное. С помощью него подписываются все транзакции в пределах личного кошелька, а поэтому его нужно хранить в конфиденциальном месте. Например, как пароли от онлайн-банков.

68. Криптография в блокчейн

Криптографическая истина — это форма вычислений и ведения учета, которая является более точной, доступной, проверяемой и защищенной от взлома, чем существующие альтернативы.

Криптографическая истина происходит от инфраструктуры с минимизацией доверия — кода, который работает именно так, как задумано, потому что его выполнение и проверка не зависят от доверия к незнакомцам или неконтролируемым переменным. Блокчейн генерирует минимизацию доверия, используя криптографию для аутентификации данных и обеспечения хронологического порядка записей, а также децентрализованный консенсус для подтверждения новых записей и сохранения их неизменности. Благодаря этому механизму многосторонние процессы могут отслеживаться и выполняться на общем, достоверно нейтральном бэкенде, который стимулируется к честному управлению.

Криптографическая истина сочетает в себе криптографию и децентрализованный консенсус для создания золотой записи среди распределенного набора сущностей и детерминированного вычисления приложений.

**Криптография — это наука о безопасном общении в присутствии враждебного поведения. Роль криптографии в сохранении конфиденциальности коммуникации, используется для аутентификации происхождения, проверки целостности и установления безответности коммуникации; т.е. для определения того, что сообщение пришло от конкретного человека и не было подделано, и что эти свойства не могут быть опровергнуты отправителем. Криптография используется для преобразования оригинального сообщения в неразборчивое сообщение, которое может интерпретировать только тот, кто в курсе.*

Существует три общих класса современной криптографии на основе программного обеспечения: хэш-функции, симметричное шифрование и асимметричное шифрование.

Блокчейн использует две основные формы криптографии: криптографию с открытым ключом и хэш-функции. Однако все чаще используются и другие криптографические методы для обеспечения масштабирования, конфиденциальности и внешних подключений к блокчейну. Ниже перечислены некоторые способы использования криптографических функций в блокчейн. Эти свойства делают так, что изменение чего-либо во входных данных, например, написание буквы

заглавными буквами или добавление знака препинания, приведет к совершенно другому и, казалось бы, случайному хэшу.

Самым популярным безопасным алгоритмом хэширования на сегодняшний день является SHA-256, который создает строки длиной 256 бит. 256-битные хэши часто представляются в шестнадцатеричном формате строк, поэтому их длина может составлять 32 или 64 символа. SHA-256 существует в рамках набора хэш-функций SHA-2, пришедшего на смену ныне криптографически неполноценному SHA-1. SHA-3 также был представлен в 2015 году на основе криптографического примитива Keccak.

Изменение одного символа в хэше, например, добавление запятой во втором входе, полностью изменит результат и покажется наблюдателям совершенно случайным.

70. Симметричное шифрование в блокчейн

Симметричное шифрование (оно же шифрование с использованием секретных ключей) предполагает **использование общего секретного ключа**, который отправитель и получатель используют **для шифрования и расшифровки сообщений**.

Симметричное шифрование является быстрым и эффективным, но возникает проблема безопасного обмена секретным ключом через Интернет. Другая проблема заключается в том, что если секретный ключ скомпрометирован, то любой, кто его использует, рискует расшифровать все предыдущие и будущие сообщения. Именно поэтому многие интернет-протоколы, использующие симметричное шифрование, такие как TLS, также используют протоколы обмена ключами для безопасного создания общего секретного ключа без его передачи через интернет.

Наиболее популярным симметричным шифром является Advanced Encryption System (AES). AES предлагает ключи длиной 128, 192 и 256 бит — существенное улучшение по сравнению с 56-битными ключами, используемыми в предыдущем и ныне небезопасном стандарте шифрования данных (DES). Для сравнения, 56-битный ключ имеет 256, или 72 квадриллиона, возможных ключей, которые можно взломать перебором менее чем за 24 часа. В противном случае, для того чтобы перебрать 2 в степени 128 возможных ключей, даже при объединении всех компьютеров мира, потребуется триллионы и триллионы лет.

Симметричное шифрование использует один и тот же секретный ключ для шифрования и расшифровки сообщений.

71. Асимметричное шифрование в блокчейн

Асимметричное шифрование (оно же криптография с открытым ключом) **дает каждому пользователю пару открытого и закрытого ключей**. Открытый ключ виден всем, в то время как закрытый ключ известен только его владельцу.

Пользователи могут шифровать сообщения с помощью своего закрытого ключа, который может расшифровать любой человек с открытым ключом. Это известно как цифровая подпись, поскольку она доказывает знание секрета без раскрытия самого секрета (т.е. у пользователя есть закрытый ключ к адресу открытого ключа).

Пользователи также могут шифровать сообщения с помощью чужого открытого ключа, который может расшифровать только человек с закрытым ключом (т.е. отправка конфиденциальной информации).

Наиболее популярными алгоритмами асимметричного шифрования являются RSA (названный в честь его изобретателей Рамиса, Шамира и Адлемана), ECC (Elliptic Curve Cryptography), Diffie Hellman (популярный протокол обмена ключами) и DSS (Digital Signature Standard). Некоторые алгоритмы асимметричного шифрования устойчивы к квантованию, а другие — нет, и в будущем их, возможно, придется модернизировать или отказаться от них.

Асимметричное шифрование требует, чтобы у каждого пользователя была пара открытый/закрытый ключ.

72. Обработка транзакции в блокчейн

1. **Аутентификация и верификация транзакций**

   Каждый пользователь блокчейна должен иметь пару открытый/закрытый ключ и адрес блокчейна, чтобы отправлять транзакции в сети. Закрытый ключ используется для генерации открытого ключа, а открытый ключ используется для генерации адреса блокчейна — как правило, это хэш открытого ключа с последними 20 байтами, добавленными к префиксу, например 0x. *Обратите внимание, что многие блокчейн-кошельки и традиционные биржи абстрагируются от создания пары открытый/закрытый ключ и взаимодействия пользователя с ней.Адрес блокчейна похож на реальное имя, связанное с банковским*

счетом пользователя (например, Pay to the Order…), только в блокчейне это псевдоанонимная строка цифр и букв.

Адрес блокчейна — это место, где пользователь хранит средства и развертывает смарт-контракты. Закрытый ключ сродни пин-коду, который пользователь должен ввести, чтобы совершить действия со своим счетом, например, перевести средства или внести изменения в смарт-контракты. Открытый ключ похож на номер банковского счета пользователя и используется для проверки подписей закрытого ключа.

При отправке транзакций пользователь посылает в сеть сообщение со своего адреса в блокчейне, содержащее данные о транзакции и цифровую подпись. Данные транзакции описывают действие, которое пользователь хочет совершить в сети, а цифровая подпись подтверждает подлинность этого действия. Цифровая подпись генерируется из двух входных данных: хэша данных транзакции пользователя и его закрытого ключа. Затем цифровая подпись присоединяется к данным транзакции, образуя транзакцию с цифровой подписью.

Майнеры/валидаторы и полные узлы, которые управляют блокчейном, запускают протокол проверки цифровой подписи, чтобы проверить действительность транзакции. Протокол проверки берет исходные данные транзакции и хэширует их. Он также расшифровывает цифровую подпись с помощью открытого ключа пользователя, чтобы получить хэш. Если эти два хэша совпадают, то транзакция считается действительной. Благодаря сочетанию хэширования и криптографии с открытым ключом для поддержки цифровых подписей блокчейн гарантирует, что только владелец закрытого ключа может получить доступ к средствам, хранящимся на соответствующем адресе блокчейна.

Блокчейн использует цифровые подписи для аутентификации и проверки транзакций пользователей

73. Атака Сивиллы

Производство блоков — это процесс, в ходе которого майнеры/валидаторы объединяют ожидающие транзакции в структуры данных, называемые блоками, и предлагают их в сети. Блок обычно состоит из списка всех транзакций, включенных в блок, и заголовка блока,

содержащего метаданные блока. Чтобы создать блок, майнер/валидатор должен сгенерировать правильный хэш блока, иначе блок будет отклонен.

В блокчейнах Proof-of-Work (PoW) (например, в Bitcoin) проводится открытое соревнование между майнерами, где первый, кто сгенерирует достоверный хэш (т.е. хэш, начинающийся хотя бы с определенного количества нулей) методом перебора, будет выбран для предложения своего блока в реестр. Блокчейн PoW использует хэш-функции в производстве блоков для обеспечения устойчивости к атакам Сивиллы — защиты от того, что один субъект контролирует процесс производства блоков, подделывая альтернативные личности. Поскольку мощность хэширования — единственный способ для майнера PoW увеличить свои шансы стать автором блока, получение контроля над большей мощностью хэширования в сети сопровождается пропорциональными финансовыми затратами в виде выполнения большего количества вычислений. Это не только предотвращает атаки типа “отказ в обслуживании”, но и требует от майнеров “работы”, чтобы даже получить шанс на вознаграждение.

Блокчейн Proof-of-Stake (PoS), такой как Ethereum 2.0, также генерирует хэши при производстве блоков, но этот процесс специально разработан таким образом, чтобы быть легким, поскольку нет конкуренции между валидаторами. Вместо этого валидаторы PoS обычно выбираются в качестве авторов блоков случайным образом, часто на основе веса их доли. Блокчейн PoS создает устойчивость к атаке Сивиллы, требуя от валидаторов внесения криптовалюты (т.е. ставки) для участия в производстве блоков. Таким образом, валидаторы PoS должны вкладывать финансовые ресурсы, чтобы увеличить свои шансы быть выбранными в качестве автора блока. Зачастую их доля подлежит сокращению (т.е. конфискации), если они нарушают определенные правила протокола, например, включают в свой блок недействительные транзакции или подписывают два блока на одной высоте.

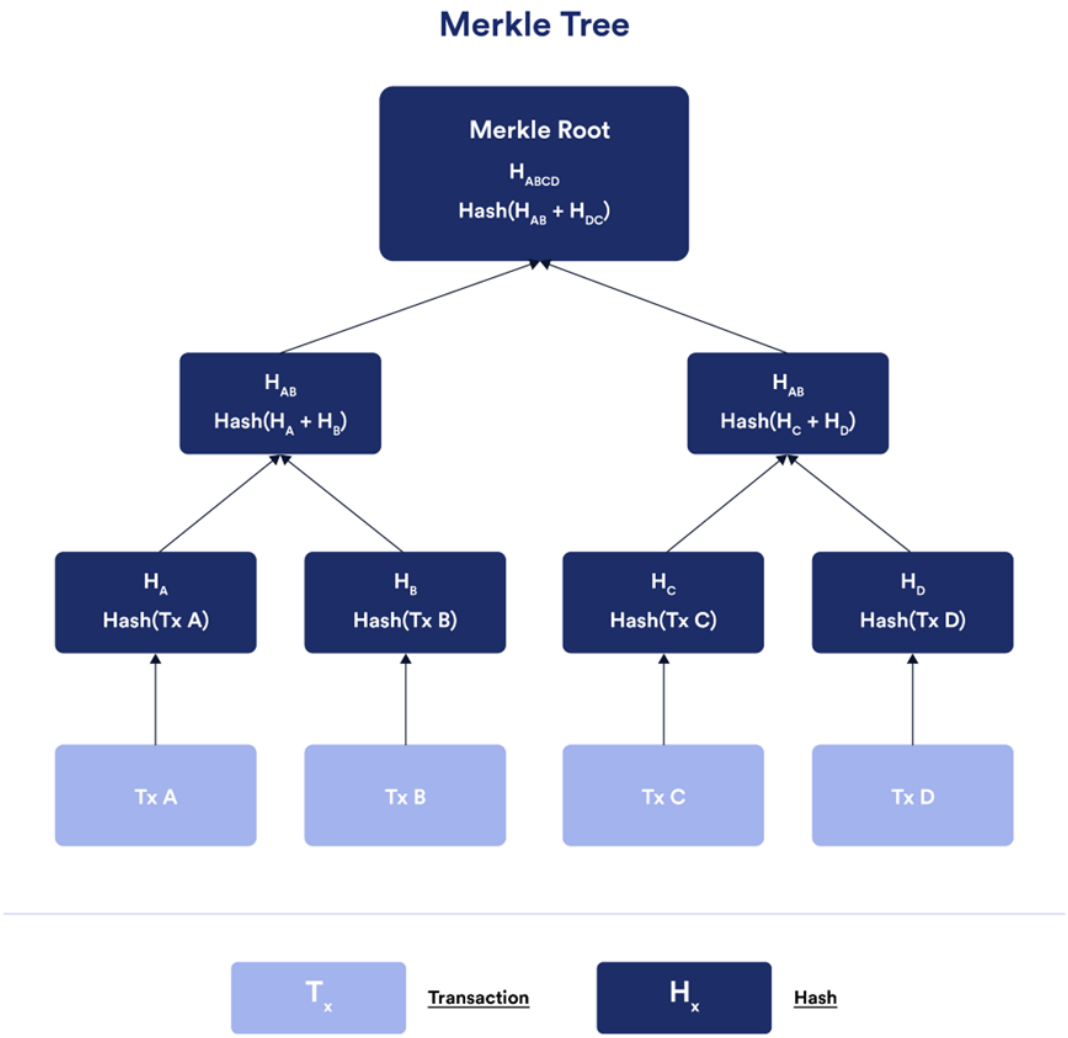
74. Хранение данных в блокчейн

Блокчейн — это реестр, содержащий все транзакции, которые когда-либо происходили в сети блокчейн в хронологическом порядке (хотя некоторые блокчейны изучают возможность обрезки исторических данных после определенной контрольной точки). Чем дольше работает блокчейн, тем больше становится реестр и тем дороже узлам обходится ее хранение и синхронизация. Чрезмерные требования к хранению и пропускной

способности ставят под угрозу децентрализацию сети блокчейн, поскольку увеличивают аппаратные требования к работе всего узла, что потенциально позволяет небольшой группе субъектов нарушить работу сети.

Для эффективного и безопасного кодирования данных реестра блокчейн использует структуры данных, называемые деревьями Меркла. В дереве Меркла каждая транзакция пользователя хэшируется, затем сопоставляется с другой хэшированной транзакцией и снова хэшируется. Хеши постоянно сопоставляются и хешируются вверх по дереву, пока не образуется единый хеш всех хешей, называемый корнем Меркла.

Биткойн использует дерево Меркла для транзакций на основе модели “Неизрасходованный выход транзакции” (UTXO), а Ethereum использует деревья Меркла для транзакций, состояния и квитанций (например, журналов и событий) в так называемой модели учётной записи с обобщенной поддержкой смарт-контрактов.



Деревья Меркла — это эффективный способ хранения данных о транзакциях в блокчейн.

Деревья Меркла полезны, поскольку занимают меньше дискового пространства по сравнению с другими структурами данных и способствуют эффективной проверке целостности данных и включению данных в распределённый реестр. Кроме того, поскольку корни Меркла включаются в заголовки блоков, они позволяют легким клиентам, подключенным к доверенному полному узлу, быстро и безопасно проверить, что конкретная транзакция была включена в блок, не загружая весь блокчейн.

Другие криптографические техники, используемые в блокчейн

Некоторые из других криптографических примитивов, используемых в блокчейн, включают:

- Доказательства нулевого знания (ZKPs) — метод, позволяющий пользователю (проверяющему) доказать другой стороне (проверяемому), что он обладает знаниями о конкретной части информации, не раскрывая фактической информации, лежащей в основе. Например, в блокчейне Zcash полные узлы могут доказать, что транзакции действительны, не зная ни отправителя, ни получателя, ни суммы транзакции.
- Доверенные среды выполнения (TEEs) — использование доверенного оборудования, такого как Intel SGX, для выполнения конфиденциальных вычислений, где целостность вычислений может быть доказана с помощью криптографических сертификатов. В качестве примера можно привести блокчейн Oasis, где все узлы-валидаторы используют доверенные среды выполнения для обеспечения конфиденциального и проверяемого выполнения смарт-контрактов.
- Пороговые схемы подписи (TSS) — форма распределенной генерации и подписания ключей, используемая для аутентификации вычислений, выполняемых децентрализованной сетью, с помощью одной криптографической подписи, для подписания которой требуется только пороговый набор участников. Например, транзакция, которую должны подписать только 10 из 15 узлов, чтобы она считалась действительной.

75. Блокчейн консенсус

Алгоритм консенсуса — это правила, по которым происходит генерация блоков в блокчейне.

В блокчейне проверкой операций и подтверждением того, что они записаны верно, занимается специальный встроенный механизм под названием алгоритм консенсуса.

Алгоритм консенсуса блокчейна — это способ, благодаря которому децентрализованные ноды сети достигают согласия (или консенсуса) о текущем состоянии данных во всех блоках.

Нода — это любой компьютер, подключенный к блокчейну, который проверяет и подтверждает транзакции, и хранит копию блокчейна.

Алгоритм консенсуса гарантирует соблюдение правил протокола и достоверность всех транзакций. Другими словами, он отвечает за то, чтобы все ноды сети были согласны с добавлением в нее нового блока. Таким образом консенсусный алгоритм поддерживает целостность и безопасность сети.

По лекции

Истина генерируется на основе децентрализованного консенсуса

Теория игр — это математический подход к стратегии, основанный на предсказании действий рациональных участников в определенном соревновании. В рамках теории игр разработка механизмов — это искусство использования стимулов для достижения желаемых результатов в рамках стратегических взаимодействий.

Блокчейн использует механизм проектирования для стимулирования децентрализованной сети рациональных, незаинтересованных узлов к поддержанию точному, неизменному, всегда доступному, устойчивому к цензуре реестру. Этот желаемый результат иногда называют достижением честного большинства — большинство узлов в сети блокчейн участвуют честно, что позволяет пользователям последовательно доверять консенсусу сети.

Структура игры

Блокчейн — это прозрачный реестр с открытым исходным кодом, то есть все участники могут просматривать всю историю реестра и проверять, как функционирует код. Участники сети блокчейн делятся на пять основных групп:

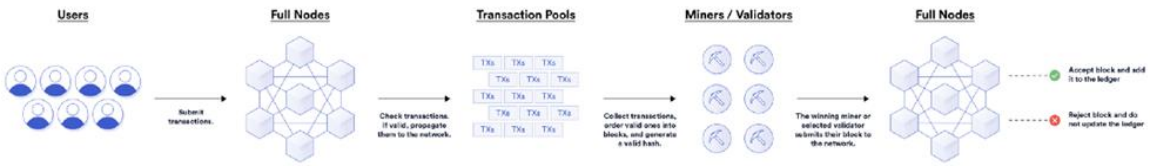
Пользователи совершают транзакции в сети либо по личным причинам, либо для размещения приложений. В блокчейн-цепочках без разрешений любому человеку разрешается отправлять транзакции в сеть в любое время.

Майнеры/валидаторы расширяют реестр, создавая новые блоки из транзакций пользователей. В некоторых блокчейнах майнером/валидатором может стать любой желающий, в то время как в

других блокчейнах количество участников ограничено. Некоторые компенсируют это ограничение, позволяя пользователям голосовать или делегировать долю валидаторам.

Полные узлы выполняют несколько функций в сетях блокчейн. Во-первых, полные узлы предоставляют конечные точки RPC, которые позволяют пользователям читать из блокчейна и писать в него. Конечные точки RPC могут быть созданы на самостоятельно размещенных узлах или доступны через сторонних поставщиков (например, Infura). При записи данных в блокчейн (т.е. отправке транзакций) полные узлы либо передают их в публичный мемпул, либо хранят в частных пулах транзакций. Это связано с тем, что в одноранговой децентрализованной сети не существует единого пула транзакций, а каждый полный узел имеет свой собственный пул транзакций.

Полные узлы также проверяют действительность транзакций пользователей перед передачей их в сеть или перед распространением транзакций других узлов в сети. Кроме того, полные узлы проверяют новые блоки, предложенные майнерами/валидаторами, что служит способом самопроверки блокчейна. Если они считают блок действительным, они добавляют его в свою копию реестра. Если они не считают его действительным, они отклоняют блок и не обновляют свою книгу. Консенсус среди децентрализованной сети полных узлов представляет собой текущее состояние реестра — совокупность всех адресов блокчейна, их балансов, развернутых смарт-контрактов и связанных с ними хранилищ. Любой человек может управлять полным узлом, хотя требования к аппаратному обеспечению в разных сетях различны. Майнеры/валидаторы часто участвуют в работе полных узлов.



Блокчейн разделяет роли майнеров/валидаторов и полноценных узлов для повышения безопасности сети.

Поставщики услуг — это внешние организации, которые предоставляют услуги блокчейн или создают услуги на основе содержимого блокчейн. К

ним относятся оракулы, протоколы индексирования, централизованные биржи, легкие клиенты, архивные узлы, исследователи блоков и многие другие. Эти организации обычно имеют бизнес-модели, которые требуют взаимодействия с блокчейном. Например, централизованные биржи обеспечивают переход между криптовалютами и традиционной финансовой системой, легкие клиенты позволяют пользователям проверять информацию, отправленную с полных узлов, без необходимости иметь полную копию реестра, а оракулы предоставляют ключевые внешние ресурсы для приложений смарт-контрактов.

MEV-боты — это субъекты, которые стремятся повлиять на то, как упорядочиваются транзакции во время добычи блоков, в частности майнеры и разработчики MEV. Извлекаемая майнерами стоимость (MEV), также называемая максимальной извлекаемой стоимостью, основана на способности майнеров/валидаторов выбирать, какие транзакции будут включены в блок и как эти транзакции будут упорядочены. Затем они могут использовать эту привилегию в своих интересах для извлечения стоимости, например, опережая или атакуя пользователей и используя возможности ликвидации и арбитража. MEV-боты обычно работают через частные пулы транзакций и офф-чейн аукционы в блокчейне, хотя могут действовать и самостоятельно.

Игровые стимулы

Стимулы делятся на две большие категории: неявные и явные. Явные стимулы — это прямые вознаграждения или наказания за действия, совершенные в игре, например, обязательный платеж за доступ к услуге или жестко закодированный штраф за нарушение правил протокола. Неявные стимулы — это косвенные вознаграждения и наказания, вытекающие из игры, например, упущенные участниками возможности получения будущих доходов или затраты труда без гарантированной компенсации.

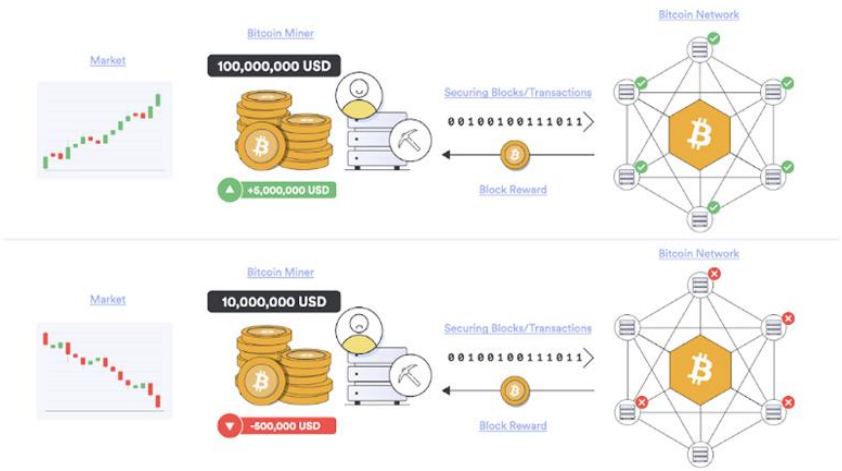
Наиболее очевидными стимулами в блокчейн являются вознаграждение за блок и плата за транзакции, выплачиваемая майнерам/валидаторам за создание одобренных блоков. Вознаграждение за блок обычно стандартизировано по количеству, в то время как плата за транзакции может колебаться в большую или меньшую сторону в зависимости от текущего спроса на ограниченное пространство блока. В процессе производства блоков также существуют стимулы для заказа транзакций определенным

образом, которыми MEV-боты пытаются воспользоваться для получения финансовой выгоды.

Блокчейн также использует финансовые санкции через механизмы, устойчивые к атаке Сивиллы, такие как PoW и PoS; т.е. майнеры/валидаторы должны сделать вычислительную или финансовую ставку, чтобы получить шанс заработать вознаграждение, которое может быть потрачено впустую или сокращено, если они идут против консенсуса сети. Еще одним преимуществом требования к производителям блоков делать вычислительную или финансовую ставку является то, что это стимулирует децентрализацию. В частности, потому, что майнеру/валидатору становится все дороже получать больший контроль над хэш-мощностью сети или долей.

Блокчейн также платит майнерам/валидаторам в их собственной криптовалюте. Это создает неявный стимул для майнеров/валидаторов честно участвовать в работе сети, поскольку безопасная и надежная сеть, скорее всего, привлечет больше пользователей. Большее количество пользователей ведет к увеличению комиссий за транзакции, а также к потенциально более высокой цене на зарабатываемую ими криптовалюту. Более высокая цена криптовалюты может означать больший доход от каждого блока и увеличение личных средств.

Implicit Staking: Holding an asset whose value can decrease based on deviations from a protocol run by you and others.



Биткойн имеет форму неявного стейкинга, когда майнеры получают вознаграждение в виде актива (BTC), который сохраняет свою ценность и покрывает их расходы только в том случае, если они поддерживают безопасность сети.

Хотя явного стимула для запуска полного узла в протоколе обычно нет, существует множество неявных стимулов для этого. Для начала, полные узлы необходимы для работы сети, поэтому, как минимум, майнеры должны их запускать. Пользователи, приложения и поставщики услуг также должны запускать полные узлы или иметь к ним доступ, чтобы отправлять и проверять статус ожидающих транзакций. Таким образом, для поддержки экосистемы существует несколько моделей бесплатных узлов и узлов как услуги.

Помимо финансовых стимулов, полные узлы критически важны для целостности реестра, поскольку отсутствие децентрализации дает меньшей группе участников контроль над сетью. Скомпрометированный реестр повлияет на поставщиков услуг и децентрализованные приложения, бизнес-модели которых зависят исключительно от предположения о честном большинстве в блокчейн.

Децентрализованная сеть с полным узлом также помогает отделить генерацию блока от его верификации, обеспечивая подотчетность майнеров/валидаторов за счет снижения их способности произвольно изменять правила протокола. Кроме того, организации, управляющие собственным полным узлом, имеют наибольшую степень устойчивости к цензуре и гарантии безопасности, поскольку им не нужно доверять третьим лицам для чтения или записи в блокчейн. Эти неявные стимулы побуждают пользователей и ключевых экономических субъектов запускать полные узлы.

Результат игры

Внедрение этих стимулов в рамках заявленной структуры игры приводит к желаемым результатам блокчейн:

- Точность — децентрализация блокчейн, неявные/явные затраты, стоящие на кону, и простота обнаружения недействительных транзакций стимулируют майнеров/валидаторов и полные узлы к

достижению честного консенсуса относительно действительности транзакций.

- Неизменность — финансовые затраты и риск при попытке изменить ранее одобренный блок делают это нежелательным, особенно когда поверх целевого блока создаются новые блоки.
- Доступность — достаточное вознаграждение помогает майнерам/валидаторам постоянно генерировать блоки, а наличие значительной суммы, поставленной на карту, стимулирует создание полноценных сетей узлов, которые расширяются за пределы майнеров/валидаторов.
- Устойчивость к цензуре — механизмы защиты от сибилов, финансовые вознаграждения и возможность беспрепятственного участия делают невероятно дорогостоящими попытки любой централизованной структуры цензурировать транзакции, особенно в течение длительного периода времени.

Благодаря этим желаемым свойствам у пользователей появляется больше стимулов для взаимодействия с сетью блокчейн. Не только использование сети может возрасти, но и пользователям станет удобнее хранить на блокчейне большие суммы ценностей и использовать свои активы на сети в приложениях смарт-контрактов.

Однако следует отметить, что, хотя блокчейн в целом устойчив к цензуре и точен, упорядочение транзакций по времени отправки или уплаченной комиссии не гарантируется из-за MEV. Именно поэтому Chainlink разрабатывает Fair Sequencing Services (FSS) — сервис оракула для децентрализованного упорядочивания транзакций для блокчейн, сетей второго уровня и приложений на основе времени их получения.

Следует также отметить, что блокчейн в конечном итоге является системой, управляемой людьми для обслуживания людей, поэтому если все придут к социальному консенсусу (т.е. офф-чейн соглашению между участниками), что что-то в блокчейне необходимо изменить, то это может быть сделано. Самым ярким примером социального консенсуса, способствующего изменениям, стал хард-форк Ethereum — обновление программного обеспечения, несовместимое с предыдущими версиями, — который восстановил украденные средства после взлома DAO в 2016 году. Общественный консенсус также может иметь обратную сторону, когда

участники сообщества препятствуют внедрению предложенных изменений, как, например, отказ от Segwit2x для сети Bitcoin и ProgPoW для сети Ethereum.

С учетом вышесказанного, блокчейн, вероятно, является наиболее эффективной сетью для защиты от нежелательных атак и изменений, которые идут вразрез с большинством общественного консенсуса, особенно с помощью криптографических и экономических механизмов.

Мир, основанный на криптографической истине

В конечном счете, криптографическая истина привносит детерминизм, точность и прозрачность в вычисления и ведение записей. Зная, что код приложения будет выполнен так, как задумано, что исторические записи были избыточно проверены и останутся защищенными от взлома, социальные и экономические отношения могут быть основаны на более глубоком уровне истины, чем когда-либо прежде. Наличие прочного фундамента правды ведет к росту экономической активности и укреплению социальной гармонии — мира, который все хотят видеть воплощенным в жизнь. Именно поэтому все мы должны создать и внедрить технологии блокчейна и блокчейн оракула, которые устанавливают криптографическую истину в основных сферах нашего коллективного общества.

76. Одноранговые сети

Одноранговые или пиринговые (от peer-to-peer) сети – это такие сети, в которых отельные элементы (узлы) обмениваются между собой файлами и хранят один и тот же набор данных. Узлы равны по мощности и выполняют одинаковые по смыслу задачи.

С финансовой точки зрения, одноранговые сети – финансовые платформы, которые позволяют осуществлять финансовые операции купли-продажи без посредников. В некоторых ситуациях, аналогичным образом организуются и процедуры получения и выдачи кредитов.

В любом случае, о первых эффективных одноранговых сетях начали говорить примерно в девяностых годах прошлого века, когда появились программы для эффективного обмена файлами. Однако основной интерес

к ним вспыхнул после появления криптовалютных сетей, основанных на классической P2P-архитектуре. Впрочем, теперь такие сети используются не только в криптоэкономике, но и при создании поисковых систем, стриминг-платформ, разнообразных онлайн-рынков и в рамках межпланетной файловой системы IPFS.

Как это работает?

Итак. Есть большое количество связанных друг с другом компьютеров. На каждом из них хранится копия всех файлов, содержащихся в сети. Это позволяет одновременно выполнять и роль клиента, и роль сервера. Кроме того, узлы равны в своих правах, так что ни у кого нет преимущества перед остальными.

Это позволяет каждому узлу одновременно закачивать новую информацию и раздавать уже хранящуюся. Процесс этого постоянен и непрерывен, поскольку все узлы обязаны реагировать на любое изменение содержимого сети.

Независимое хранение информации на разных источниках делает всю систему более устойчивой к внешним воздействиям и взломам. Кроме того, в сетях P2P нет единой точки отказа, в отличие от централизованных систем.

Выделяют три основных типа одноранговых сетей.

- Неструктурированные. Узлы случайным образом контактируют друг с другом, что обеспечивает высокую активность системы даже при значительной текучке узлов. Относительно просты в построении, однако требуют больших мощностей при работе. Дело в том, что поисковые или любые другие запросы отправляются на максимально возможное число машин одновременно, так что сеть перегружается «пустыми» информационными запросами, которые могут помешать прохождению отзывов.
- Структурированные. Узлы объединены в строгую систему со своей архитектурой. В такой системе активно используется поиск по хэш-функциям, что значительно снижает общую интенсивность информационного потока. Однако такие системы в некоторой степени уже являются централизованными, поэтому куда более требовательны к обслуживанию и установке. Да и при временном выпадении большой группы узлов, вся работа системы нарушается.
- Гибридные. Комбинация предыдущих вариантов. Например, наличие центрального сервера, упрощающего процесс обмена между

отдельными узлами. Как правило, подобные сети отличаются высокой производительностью и устойчивостью.

P2P и блокчейн

Когда Сатоши Накамото создавал сеть Биткоин, он сформулировал её как «одноранговая платёжная система электронных денежных средств». Это обеспечивалось благодаря работе новой технологии – блокчейн.

Технология позволяла обмениваться цифровой валютой с участниками по всему миру без использования услуг посредников и центральных серверов. Кроме того, любой желающий мог присоединиться к этому процессу.

Роль банков, фиксирующих и гарантирующих подлинность финансовых операций, взял на себя цифровой регистр, в котором фиксируется вся активность и к которому может обратиться любой желающий. Проще говоря, каждый узел, подключенный к сети, хранит всю информацию, содержащуюся в ней, и регулярно связывается с другими узлами, чтобы гарантировать достоверность и актуальность хранимой информации.

Но несмотря на то, что сеть является одноранговой, разные узлы могут брать на себя разные функции. Так, например, существуют так называемые «полные ноды», основная задача которых - проверка транзакций в системе на предмет соответствия их действующим алгоритмам консенсуса. А есть и другие узлы – майнеры, основная задача которых – генерация новых блоков хранения информации.

Преимущества

- Высокая устойчивость к DoS-атакам, поскольку информация не хранится в одном месте, а пропускная способность сети позволяет одновременно обрабатывать множество запросов.
- Высокая устойчивость к изменению уже внесённой информации. Чтобы переписать данные на всех машинах сети, нужно мощность, превышающая половину общей мощности системы. Это называется «Атака 51 процента». Довольно редкое явление, особенно в больших сетях.
- Высокая устойчивость к внутренним ошибкам и противоречивым сигналам. Сети P2P демонстрируют высокую «византийскую отказоустойчивость».
- Устойчивость к внешней цензуре. Криптовалютные кошельки, например, не могут быть заморожены решением судов и правительств, в отличие от счетов в банках.

Недостатки

- Чем больше сеть, тем больше вычислительной работы нужно проделать, чтобы обеспечить актуальность хранимой информации. За счёт этого, замедляется общая скорость работы. Эта проблема известна как «проблема масштабируемости». И очень немногие одноранговые сети нашли более-менее приемлемые способы её решения. Примерами этого могут служить Lightning Network, Ethereum Plasma и протокол Mimblewimble.
- Возможна ситуация «разделения» цепочки информации на две – ситуация «хардфорка». В таком случае, оба новых разветвления оказываются уязвимыми перед атакой повторного воспроизведения (достоверная транзакция с одной цепочки дублируется на второй, за счёт чего – снимается в 2 раза больше денег). А если учесть, что подобные хардфорки случаются регулярно, то приходится дополнительно обеспечивать защиту от них.
- Невозможность исправления информации может быть и минусом. Изъять, например, компрометирующие данные из такой сети – крайне сложно. Как и цифровые копии, связанные с нарушением авторских прав.

Одноранговая архитектура заметно улучшила эффективность хранения информации, сделав обмен более прозрачным, устойчивым и быстрым. Причём не только в рамках криптовалютных сетей, но и при обмене информацией иного рода – от файлов пиринговых сетей, типа torrent, до информации, связанной с крупными торговыми операциями или обработкой больших массивов цифровых данных.

77. Proof of Work

Proof-of-work (PoW) — децентрализованный алгоритм консенсуса, впервые представленный Bitcoin (BTC).

Сеть Bitcoin требует, чтобы ее участники вкладывали в нее вычислительные мощности (например, GPU) для решения математических задач, сгенерированных произвольным образом. Это требуется для защиты сети от единоличного контроля и манипуляций.

Каждая транзакция в сети валидируется перед добавлением в блокчейн. Затем каждый блок валидируется майнерами, получающими вознаграждение в токенах BTC за вложенные вычислительные мощности.

Поэтому консенсус и называется proof-of-work — «доказательство выполнения работы».

Этот алгоритм уже проверен временем, с его помощью в сети Bitcoin успешно про валидированы миллиарды транзакций. Поддерживая достоверность и надежность, proof-of-work доказал свою ценность как самый защищенный и децентрализованный алгоритм консенсуса.

78. Смарт-контракты Blockchain

Интеллектуальные контракты — это цифровые протоколы для передачи информации, которые используют математические алгоритмы для автоматического выполнения транзакции после выполнения установленных условий и полного контроля процесса. Это определение, которое опережало свое время более чем на десять лет, остается точным и по сей день. Однако в 1996 году эта концепция не могла быть реализована: в то время необходимые технологии не существовали, в частности, распределенная книга.

В 2008 году появился bitcoin, появилась первая криптовалюта, созданная на основе революционной технологии blockchain, которой ранее не хватало децентрализованной книги. Blockchain биткоина не позволяет устанавливать условия для совершения транзакции в новом блоке, поскольку он содержит только информацию о самой транзакции. Тем не менее появление технологии послужило толчком для разработки смарт-контрактов. Спустя пять лет блочная платформа Ethereum позволила использовать смарт-контракты на практике. Сегодня рынок предлагает множество платформ, которые позволяют использовать смарт-контракты, но Ethereum остается одним из самых распространенных.

Интеллектуальные контракты — это компьютерные протоколы или, проще говоря, компьютерный код. Код используется для ввода всех условий договора, заключенного между сторонами сделки, в blockchain. Обязательства участников предоставляются в интеллектуальном контракте в форме «если- то» (например: «если Сторона А переводит деньги, тогда Сторона В, передает права на квартиру»). Могут быть два или более участников, и они могут быть отдельными лицами или организациями. Как только данные условия будут выполнены, смарт-контракт самостоятельно выполняет транзакцию и гарантирует, что соглашение будет соблюдаться.

Смарт-контракты позволяют обменять деньги, товары, недвижимость, ценные бумаги и другие активы. Контракт хранится и повторяются в децентрализованной книге, в которой информация не может быть сфальсифицирована или удалена. В то же время шифрование данных обеспечивает анонимность сторон соглашения. **Важной особенностью** интеллектуальных контрактов является то, что **они могут работать только с активами, находящимися в их цифровой экосистеме**. *Как подключить виртуальный и реальный мир в настоящее время является одной из основных трудностей работы со смарт-контрактами. Это является причиной существования «оракулов», специальных программ, которые помогают компьютерным протоколам получать необходимую информацию из реального мира.*

Преимущества

- Скорость. Обработка документов вручную занимает много времени и задерживает выполнение задач. Смарт-контракты предполагают автоматизированный процесс и в большинстве случаев не требуют личного участия, что экономит драгоценное время.
- Независимость. Смарт-контракты исключают возможность вмешательства третьих сторон. Гарантия на транзакцию — сама программа, которая, в отличие от посредников, не даст основания сомневаться в ее целостности.
- Надежность. Данные, записанные в blockchain, не могут быть изменены или уничтожены. Если одна сторона сделки не выполняет свои обязательства, другая сторона будет защищена условиями интеллектуального договора.
- Нет ошибок — Автоматическая система для выполнения транзакций и удаления человеческого фактора обеспечивает высокую точность при выполнении контрактов.
- Сбережения. Смарт-контракты могут обеспечить значительную экономию за счет устранения расходов для посредников и сокращения операционных расходов, а также возможность для сторон работать вместе на более выгодных условиях.

Недостатки

- Отсутствие регулирования. В международно-правовой области отсутствуют концепции «blockchain», «умный контракт» и «криптовалюты».
- Сложность реализации. Интеграция интеллектуальных контрактов с элементами реального мира часто занимает много времени, денег, и усилия.
- Невозможность изменения интеллектуального контракта. Парадоксально, что один из главных плюсов интеллектуальных контрактов также можно рассматривать как конфликт. Если стороны достигают более выгодного соглашения или возникают новые факторы, они не смогут изменить контракт. По этой причине варианты дополнительных соглашений должны быть реализованы по мере разработки новых blockchain платформ.

Где можно использовать смарт-контракты?
Смарт-контракты могут изменять разные области. Мы можем выделить несколько отраслей, в которых интеллектуальные контракты будут наиболее эффективными:

- Финансы
- Страхование
- Электронная коммерция
- Аудит и налогообложение
- Выборы

79. Операторы Flux / Mono в Reactor

1. Введение в Project Reactor

Реактивное программирование поддерживается Spring Framework, начиная с версии 5. Эта поддержка построена на основе Project Reactor.

Project Reactor (или просто Reactor) - это библиотека Reactive для создания неблокирующих приложений на JVM, основанная на спецификации Reactive Streams. Reactor - это основа реактивного стека в экосистеме Spring, и он разрабатывается в тесном сотрудничестве со Spring. WebFlux, веб-фреймворк с реактивным стеком Spring, использует Reactor в качестве базовой зависимости.

1.1 Модули Reactor

Проект Reactor состоит из набора модулей, перечисленных в документации Reactor. Модули встраиваемы и совместимы. Основным артефактом является Reactor Core, который содержит реактивные типы Flux и Mono, которые реализуют интерфейс Publisher Reactive Stream (подробности см. в **первом сообщении этой серии**) и набор операторов, которые могут применяться к ним.

Некоторые другие модули:

- Reactor Test - предоставляет некоторые утилиты для тестирования реактивных потоков
- Reactor Extra - предоставляет некоторые дополнительные операторы Flux
- Reactor Netty - неблокирующие клиенты и серверы TCP, HTTP и UDP с поддержкой обратного давления - на основе инфраструктуры Netty
- Reactor Adapter - адаптер для других реактивных библиотек, таких как RxJava2 и Akka Streams
- Reactor Kafka - реактивный API для Kafka, который позволяет публиковать и получать сообщения в Kafka.

1.2 Настройка проекта

Прежде чем мы продолжим, если вы хотите настроить проект и запустить некоторые из приведенных ниже примеров кода, сгенерируйте новое приложение Spring Boot с помощью **Spring Initializr**. В качестве зависимости выберите Spring Reactive Web. После импорта проекта в вашу среду IDE взгляните на файл POM, и вы увидите, что добавлена зависимость spring-boot-starter-webflux, которая также внесет зависимость ядра-реактора. Также в качестве зависимости добавлен тест-реактор. Теперь вы готовы к запуску следующих примеров кода.

...

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-test</artifactId>
        <scope>test</scope>
    </dependency>
    </dependencies>
...
```

2. Возможности Reactor Core

Reactor Core определяет реактивные типы Flux и Mono.

2.1 FLUX и MONO

Flux - это Publisher, который может испускать от 0 до N элементов, а Mono может испускать от 0 до 1 элемента. Оба они завершаются либо сигналом завершения, либо ошибкой, и они вызывают методы onNext, onComplete и

onError нижестоящего подписчика. Помимо реализации функций, описанных в спецификации Reactive Streams, Flux и Mono предоставляют набор операторов для поддержки преобразований, фильтрации и обработки ошибок.

В качестве первого упражнения перейдите к классу тестирования, созданному в вашем новом проекте, добавьте следующий пример и запустите его:

```
@Test
void simpleFluxExample() {
    Flux<String> fluxColors = Flux.just("red", "green", "blue");
    fluxColors.subscribe(System.out::println);
}
```

Метод just создает поток, который испускает предоставленные элементы, а затем завершается. Ничего не передается, пока кто-нибудь на это не подпишется. Чтобы подписаться на него, мы вызываем метод subscribe и в этом случае просто распечатываем отправленные элементы. Создание Mono также может быть выполнено с помощью метода just, с той лишь разницей, что разрешен только один параметр.

2.2 Объединение операторов

Взгляните на Flux API, и вы увидите, что почти все методы возвращают Flux или Mono, что означает, что операторы могут быть связаны. Каждый оператор добавляет поведение к Publisher (Flux или Mono) и переносит Publisher предыдущего шага в новый экземпляр. Данные поступают от первого издателя и перемещаются по цепочке, трансформируясь каждым оператором. В конце концов, подписчик завершает процесс. Обратите внимание, что ничего не происходит, пока подписчик не подпишется на издателя.

Существует оператор log(), который обеспечивает регистрацию всех сигналов Reactive Streams, происходящих за кулисами. Просто измените последнюю строку приведенного выше примера на

fluxColors.log().subscribe(System.out::println);

и перезапустите тест. Теперь вы увидите, что к выходным данным добавляется следующее:

```
2020-09-12 16:16:39.779 INFO 6252 --- [ main] reactor.Flux.Array.1
: | onSubscribe([Synchronous Fuseable]
FluxArray.ArraySubscription)

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1
: | request(unbounded)

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1
: | onNext(red)

red

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1
: | onNext(green)

green

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1
: | onNext(blue)

blue

2020-09-12 16:16:39.782 INFO 6252 --- [ main] reactor.Flux.Array.1
: | onComplete()
```

Теперь, чтобы увидеть, что произойдет, если вы исключите вызов subscribe(), снова измените последнюю строку кода на следующую и повторно запустите тест:

```
fluxColors.log();
```

Как вы увидите из выходных данных журнала, сейчас никакие элементы не отправляются - поскольку нет подписчика, инициирующего процесс.

80. Категории Reactor

2.3 Поиск подходящего оператора

Reactor предоставляет длинный список операторов, и в качестве помощи в поиске подходящего оператора для конкретного варианта использования есть специальное приложение в справочной документации Reactor. Он разделен на различные категории, как показано в таблице ниже.

КАТЕГОРИЯ ОПЕРАТОРА	ПРИМЕРЫ

Создание новой последовательности	just, fromArray, fromIterable, fromStream
Преобразование существующей последовательности	map, flatMap, startWith, concatWith
Заглядывать в последовательность	doOnNext, doOnComplete, doOnError, doOnCancel
Фильтрация последовательности	filter, ignoreElements, distinct, elementAt, takeLast
Обработка ошибок	onErrorReturn, onErrorResume, retry
Работаем со временем	elapsed, interval, timestamp, timeout
Расщепление потока	buffer, groupBy, window
Возвращаясь к синхронному миру	block, blockFirst, blockLast, toIterable, toStream
Многоадресная рассылка потока нескольким подписчикам	publish, cache, replay

81. Работа с backpressure в Reactor

Как вы могли вспомнить из первой части этой серии блогов, противодавление - это способность потребителя сигнализировать производителю, с какой скоростью выброса он может справиться, чтобы он не перегружался.

В приведенном ниже примере показано, как подписчик может контролировать скорость передачи, вызывая request(n) метод в Subscription.

```
@Test
public void backpressureExample() {
        Flux.range(1,5)
        .subscribe(new Subscriber<Integer>() {
        private Subscription s;
        int counter;

        @Override
        public void onSubscribe(Subscription s) {
                System.out.println("onSubscribe");
                this.s = s;
                System.out.println("Requesting 2 emissions");
                s.request(2);
        }

        @Override
        public void onNext(Integer i) {
                System.out.println("onNext " + i);
                counter++;
                if (counter % 2 == 0) {
                System.out.println("Requesting 2 emissions");
                s.request(2);
                }
        }
```

```
        @Override
        public void onError(Throwable t) {
            System.err.println("onError");
        }

        @Override
        public void onComplete() {
            System.out.println("onComplete");
         }
        });
}
```

Запустите его, и вы увидите, что по запросу одновременно генерируются два значения:

```
onSubscribe
Requesting 2 emissions
onNext 1
onNext 2
Requesting 2 emissions
onNext 3
onNext 4
Requesting 2 emissions
onNext 5
onComplete
```

В Subscription также есть `cancel`метод, позволяющий запросить Издателя остановить эмиссию и очистить ресурсы.

82. Spring WebFlux

1. Введение в Spring WebFlux

Исходный веб-фреймворк для Spring - Spring Web MVC - был построен для Servlet API и контейнеров Servlet.

WebFlux был представлен как часть Spring Framework 5.0. В отличие от Spring MVC, он не требует Servlet API. Он полностью асинхронный и неблокирующий, реализует спецификацию Reactive Streams через проект Reactor (см. предыдущий пост в блоге).

WebFlux требует Reactor в качестве основной зависимости, но он также может взаимодействовать с другими реактивными библиотеками через Reactive Streams.

1.1 Модели программирования

Spring WebFlux поддерживает две разные модели программирования: на основе аннотаций и функциональную.

1.1.1 Аннотированные контроллеры

Если вы работали со Spring MVC, модель на основе аннотаций будет выглядеть довольно знакомой, поскольку в ней используются те же аннотации из веб-модуля Spring, что и в Spring MVC. Основное отличие состоит в том, что теперь методы возвращают реактивные типы Mono и Flux. См. Следующий пример RestController с использованием модели на основе аннотаций:

```
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentService studentService;
```

```
    public StudentController() {
    }

    @GetMapping("/{id}")
    public Mono<ResponseEntity<Student>> getStudent(@PathVariable long id)
{
        return studentService.findStudentById(id)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @GetMapping
    public Flux<Student> listStudents(@RequestParam(name = "name",
required = false) String name) {
        return studentService.findStudentsByName(name);
    }

    @PostMapping
    public Mono<Student> addNewStudent(@RequestBody Student student) {
        return studentService.addNewStudent(student);
    }

    @PutMapping("/{id}")
    public Mono<ResponseEntity<Student>> updateStudent(@PathVariable long
id, @RequestBody Student student) {
```

```
        return studentService.updateStudent(id, student)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public Mono<ResponseEntity<Void>> deleteStudent(@PathVariable long id)
{
        return studentService.findStudentById(id)
                .flatMap(s ->
                        studentService.deleteStudent(s)
                                .then(Mono.just(new
ResponseEntity<Void>(HttpStatus.OK)))
                )
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

}
```

Некоторые пояснения к функциям, использованным в примере:

- map функция используется для преобразования элемента, испускаемого Mono, применяя функцию синхронной к нему.
- flatMap функция используется для преобразования элемент, испускаемый Mono асинхронно, возвращая значение, излучаемого другим Mono.
- defaultIfEmpty функция обеспечивает значение по умолчанию, если Mono завершается без каких - либо данных.

1.1.2 Функциональные конечные точки

Модель функционального программирования основана на лямбда-выражении и оставляет за приложением полную обработку запроса. Он основан на концепциях HandlerFunctions и RouterFunctions.

HandlerFunctions используются для генерации ответа на данный запрос:

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
    Mono<T> handle(ServerRequest request);
}
```

RouterFunction используется для маршрутизации запросов к HandlerFunctions:

```
@FunctionalInterface
public interface RouterFunction<T extends ServerResponse> {
    Mono<HandlerFunction<T>> route(ServerRequest request);
    ...
}
```

Продолжая с тем же примером ученика, мы получим что-то вроде следующего, используя функциональный стиль.

A StudentRouter:

```
@Configuration
public class StudentRouter {

    @Bean
```

```
    public RouterFunction<ServerResponse> route(StudentHandler studentHandler){
        return RouterFunctions
            .route(
                GET("/students/{id:[0-9]+}")
                    .and(accept(APPLICATION_JSON)), studentHandler::getStudent)
            .andRoute(
                GET("/students")
                    .and(accept(APPLICATION_JSON)), studentHandler::listStudents)
            .andRoute(
                POST("/students")

.and(accept(APPLICATION_JSON)),studentHandler::addNewStudent)
            .andRoute(
                PUT("students/{id:[0-9]+}")
                    .and(accept(APPLICATION_JSON)),
studentHandler::updateStudent)
            .andRoute(
                DELETE("/students/{id:[0-9]+}")
                    .and(accept(APPLICATION_JSON)),
studentHandler::deleteStudent);
    }

}
```

И StudentHandler:

```
@Component
public class StudentHandler {
```

```
    private StudentService studentService;

    public StudentHandler(StudentService studentService) {
        this.studentService = studentService;
    }

    public Mono<ServerResponse> getStudent(ServerRequest serverRequest) {
        Mono<Student> studentMono = studentService.findStudentById(
                Long.parseLong(serverRequest.pathVariable("id")));
        return studentMono.flatMap(student -> ServerResponse.ok()
                .body(fromValue(student)))
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> listStudents(ServerRequest serverRequest) {
        String name = serverRequest.queryParam("name").orElse(null);
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(studentService.findStudentsByName(name), Student.class);
    }

    public Mono<ServerResponse> addNewStudent(ServerRequest serverRequest) {
        Mono<Student> studentMono = serverRequest.bodyToMono(Student.class);
```

```
        return studentMono.flatMap(student ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(studentService.addNewStudent(student), Student.class));


    }

    public Mono<ServerResponse> updateStudent(ServerRequest serverRequest) {
        final long studentId = Long.parseLong(serverRequest.pathVariable("id"));
        Mono<Student> studentMono = serverRequest.bodyToMono(Student.class);


        return studentMono.flatMap(student ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(studentService.updateStudent(studentId, student), Student.class));
    }


    public Mono<ServerResponse> deleteStudent(ServerRequest serverRequest) {
        final long studentId = Long.parseLong(serverRequest.pathVariable("id"));
        return studentService
                .findStudentById(studentId)
                .flatMap(s -> ServerResponse.noContent().build(studentService.deleteStudent(s)))
```

```
                .switchIfEmpty(ServerResponse.notFound().build());
    }

}
```

Некоторые пояснения к функциям, использованным в примере:

- switchIfEmpty функция имеет ту же цель, defaultIfEmpty, но вместо того, чтобы обеспечить значение по умолчанию, она используется для обеспечения альтернативного Mono.

Сравнивая две модели, мы видим, что:

- Для использования функционального варианта требуется еще немного кода для таких вещей, как получение входных параметров и синтаксический анализ до ожидаемого типа.
- Не полагаясь на аннотации, но написание явного кода предлагает некоторую большую гибкость и может быть лучшим выбором, если нам, например, нужно реализовать более сложную маршрутизацию.

1.2 Поддержка сервера

WebFlux работает в средах выполнения, отличных от сервлетов, таких как Netty и Undertow (неблокирующий режим), а также в средах выполнения сервлетов 3.1+, таких как Tomcat и Jetty.

По умолчанию стартер Spring Boot WebFlux использует Netty, но его легко переключить, изменив зависимости Maven или Gradle.

Например, чтобы переключиться на Tomcat, просто исключите spring-boot-starter-netty из зависимости spring-boot-starter-webflux и добавьте spring-boot-starter-tomcat:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <exclusions>
```

```
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-netty</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>

</dependency>
```

1.3 Конфигурация

Spring Boot обеспечивает автоматическую настройку Spring WebFlux, которая хорошо работает в общих случаях. Если вам нужен полный контроль над конфигурацией WebFlux, можно использовать аннотацию @EnableWebFlux (эта аннотация также потребуется в простом приложении Spring для импорта конфигурации Spring WebFlux).

Если вы хотите сохранить конфигурацию Spring Boot WebFlux и просто добавить дополнительную конфигурацию WebFlux, вы можете добавить свой собственный класс @Configuration типа WebFluxConfigurer (но без @EnableWebFlux).

Подробные сведения и примеры см. в документации по конфигурации WebFlux.

2. Защита ваших конечных точек

Чтобы получить поддержку Spring Security WebFlux, сначала добавьте в свой проект зависимость spring-boot-starter-security. Теперь вы можете включить его, добавив @EnableWebFluxSecurity аннотацию в свой класс Configuration (доступно с Spring Security 5.0).

В следующем упрощенном примере будет добавлена поддержка двух пользователей, один с ролью USER, а другой с ролью ADMIN, принудительно применить базовую аутентификацию HTTP и потребовать роль ADMIN для любого доступа к пути /student/admin:

```
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {

        UserDetails user = User
                .withUsername("user")
                .password(passwordEncoder().encode("userpwd"))
                .roles("USER")
                .build();

        UserDetails admin = User
                .withUsername("admin")
                .password(passwordEncoder().encode("adminpwd"))
                .roles("ADMIN")
                .build();

        return new MapReactiveUserDetailsService(user, admin);
    }

    @Bean
```

```
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
        return http.authorizeExchange()
                .pathMatchers("/students/admin")
                .hasAuthority("ROLE_ADMIN")
                .anyExchange()
                .authenticated()
                .and().httpBasic()
                .and().build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }


}
```

Также можно защитить метод, а не путь, сначала добавив аннотацию @EnableReactiveMethodSecurity к вашей конфигурации:

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {
    ...

}
```

А затем добавляем @PreAuthorize аннотацию к защищаемым методам. Например, мы можем захотеть, чтобы наши методы POST, PUT и DELETE были доступны только для роли ADMIN. Затем к этим методам можно применить аннотацию PreAuthorize, например:

```
@DeleteMapping("/{id}")
@PreAuthorize("hasRole('ADMIN')")
public Mono<ResponseEntity<Void>> deleteStudent(@PathVariable long id) {
    ...
}
```

Spring Security предлагает дополнительную поддержку, связанную с приложениями WebFlux, например защиту CSRF, интеграцию OAuth2 и реактивную аутентификацию X.509. Для получения дополнительной информации прочтите следующий раздел в документации Spring Security: Реактивные приложения

3. Веб-клиент

Spring WebFlux также включает реактивный, полностью неблокирующий веб-клиент. У него есть функциональный, свободный API, основанный на Reactor.

Давайте рассмотрим (еще раз) упрощенный пример того, как WebClient можно использовать для запроса нашего StudentController:

```
public class StudentWebClient {

    WebClient client = WebClient.create("http://localhost:8080");

    public Mono<Student> get(long id) {
        return client
                .get()
```

```
                .uri("/students/" + id)
                .headers(headers -> headers.setBasicAuth("user", "userpwd"))
                .retrieve()
                .bodyToMono(Student.class);
    }

    public Flux<Student> getAll() {
        return client.get()
                .uri("/students")
                .headers(headers -> headers.setBasicAuth("user", "userpwd"))
                .retrieve()
                .bodyToFlux(Student.class);
    }

    public Flux<Student> findByName(String name) {
        return client.get()
                .uri(uriBuilder -> uriBuilder.path("/students")
                .queryParam("name", name)
                .build())
                .headers(headers -> headers.setBasicAuth("user", "userpwd"))
                .retrieve()
                .bodyToFlux(Student.class);
    }

    public Mono<Student> create(Student s)  {
```

```
        return client.post()
                .uri("/students")
                .headers(headers -> headers.setBasicAuth("admin", "adminpwd"))
                .body(Mono.just(s), Student.class)
                .retrieve()
                .bodyToMono(Student.class);
    }

    public Mono<Student> update(Student student)  {
        return client
                .put()
                .uri("/students/" + student.getId())
                .headers(headers -> headers.setBasicAuth("admin", "adminpwd"))
                .body(Mono.just(student), Student.class)
                .retrieve()
                .bodyToMono(Student.class);
    }

    public Mono<Void> delete(long id) {
        return client
                .delete()
                .uri("/students/" + id)
                .headers(headers -> headers.setBasicAuth("admin", "adminpwd"))
                .retrieve()
                .bodyToMono(Void.class);
```

```
    }

}
```

4. Тестирование

Для тестирования вашего реактивного веб-приложения WebFlux предлагает WebTestClient, который поставляется с API, аналогичным WebClient.

Давайте посмотрим, как мы можем протестировать наш StudentController с помощью WebTestClient:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class StudentControllerTest {
    @Autowired
    WebTestClient webClient;

    @Test
    @WithMockUser(roles = "USER")
    void test_getStudents() {
        webClient.get().uri("/students")
                .header(HttpHeaders.ACCEPT, "application/json")
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBodyList(Student.class);

    }
```

```
    @Test
    @WithMockUser(roles = "ADMIN")
    void testAddNewStudent() {
        Student newStudent = new Student();
        newStudent.setName("some name");
        newStudent.setAddress("an address");

        webClient.post().uri("/students")
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON)
                .body(Mono.just(newStudent), Student.class)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBody()
                .jsonPath("$.id").isNotEmpty()
                .jsonPath("$.name").isEqualTo(newStudent.getName())
                .jsonPath("$.address").isEqualTo(newStudent.getAddress());
    }

    ...
}
```

5. WEBSOCKETS и RSOCKET

5.1 Веб-сокеты

В Spring 5 WebSockets также получает дополнительные реактивные возможности. Чтобы создать сервер WebSocket, вы можете создать реализацию `WebSocketHandler` интерфейса, которая содержит следующий метод:

```
Mono<Void> handle(WebSocketSession session)
```

Этот метод вызывается при установке нового соединения WebSocket и позволяет обрабатывать сеанс. Он принимает в `WebSocketSession` качестве входных данных и возвращает Mono <Void>, чтобы сигнализировать о завершении обработки сеанса приложением.

`WebSocketSession` имеет методы, определенные для обработки входящих и исходящих потоков:

```
Flux<WebSocketMessage> receive()
```

```
Mono<Void> send(Publisher<WebSocketMessage> messages)
```

Spring WebFlux также предоставляет `WebSocketClient` реализации для Reactor Netty, Tomcat, Jetty, Undertow и стандартной Java.

Для получения дополнительной информации прочтите следующую главу в документации Spring's Web on Reactive Stack: WebSockets

5.2 RSOCKET

RSocket - это протокол, моделирующий семантику реактивных потоков по сети. Это двоичный протокол для использования в транспортных потоках байтовых потоков, таких как TCP, WebSockets и Aeron. В качестве введения в эту тему я рекомендую следующий пост в блоге, который написал мой коллега Pär: An introduction to RSocket

А для получения дополнительной информации о поддержке Spring Framework протокола RSocket

6. Подводя итог…

Это сообщение в блоге продемонстрировало, как WebFlux можно использовать для создания реактивного веб-приложения. В следующем и последнем посте этой серии будет показано, как мы можем сделать весь наш стек приложений полностью неблокирующим, также реализовав неблокирующую связь с базой данных - с помощью R2DBC (Reactive Relational Database Connectivity)!

83. Стратегии backpressure

Backpressure (back pressure) – это то, с чем рано или поздно сталкивается почти любой разработчик, а для некоторых это становится частой и серьезной проблемой.

В мире программирования термин “backpressure” является аналогией, позаимствованной из физики. В двух словах, *backpressure* – это сопротивление или некоторая сила, действующая в направлении, противоположном желаемому направлению движению частиц в трубе. В контексте программирования эту фразу можно переиначить: …*это сопротивление желаемому потоку данных приложения*.

Цель любого приложения – это получение входных данных и приведение их к некоторому желаемому виду. Когда возникает *backpressure*, между входом и выходом возникает некое сопротивление. В большинстве случаев, узкое место возникает из-за ограничений в вычислительных мощностях – данные не успевают обрабатываться с всё нарастающей скоростью их поступления. Но есть и другие формы *backpressure*, например, когда приложение ожидает реакции пользователя на некоторое событие, которая, по понятным причинам, может запаздывать.

Иногда под *backpressure* понимается не само явление как таковое, а конкретный механизм его обработки.

Пример чтения и записи файлов

Запись файла процесс более медленный, чем чтение. Если комбинация ОС, жесткого диска и библиотек обеспечивает эффективную скорость чтения в 150 мб/с, а записи – в 100 мб/с, то при чтении файла с последующей его

записью обратно на диск, вы должны записывать в буфер “лишние” 50 мб каждую секунду.

Решение очевидно: читать ровно столько, сколько сможете записать. Практически все библиотеки имеют соответствующие абстракции для работы с этими случаями.

Взаимодействие между серверами

Микросервисная архитектура, где каждый микросервис может разворачиваться на отдельном сервере, пользуется все большей популярностью. *Backpressure* в этом случае возникает, когда один сервер отправляет запросы другому слишком быстро и второй сервер не успевает их обрабатывать.

Предположим, что **Сервер А** отправляет **Серверу В** 100 запросов в секунду, но **Сервер В** может обрабатывать только 75 из них. Имеем дефицит в 25 запросов в секунду. В любом случае, **Сервер В** должен каким-то образом работать с *backpressure.* Один из вариантов – это буферизация избыточных запросов, но если они будут сыпаться с прежней скоростью, однажды свободная память сервера закончится. Можно игнорировать избыточные запросы, но во многих случаях это запрещается требованиями к системе.

Идеальный случай, когда **Сервер В** может сам контролировать поток запросов от **Сервера А**, но и это не всегда возможно. Например, если **Сервер А** генерирует запросы от имени пользователя. Мы же не можем сказать пользователям “притормозите!” (хотя иногда стОит!).

Стратегии backpressure

Так как же работать с *backpressure*? Если опустить вариант увеличения производительности серверного железа, остаётся всего три опции:

Контролировать источник (увеличивать частоту или снижать – решает приёмник)

Буферизация (временно хранить запросы, которые не были обработаны “на лету”)

Игнорировать (пропускать обработку части запросов)

Контроль источника – безусловно лучший вариант. Если говорить о случаях реального применения, то единственный возможный оверхэд – реализация самого механизма контроля.

К сожалению этот вариант не всегда реализуем. Например случай с валом пользовательских запросов самый очевидный, ведь не так-то просто договориться с пользователями!

Буферизация обычно выбирается в качестве следующего пути решения проблемы. Но надо держать в уме, что неограниченная буферизация опасна, так как приводит к утечкам памяти с последующим падением сервера. Зачастую лучше начать игнорировать запросы, чем продолжать складывать их в буфер, отнимая остатки памяти у сервера.

Игнорирование входящих запросов это последняя стратегия, которая часто сочетается с буферизацией. Обычно “сбрасывают” какую-то часть запросов ежесекундно, чтобы увеличить время существования буфера.

84. TCP Flow control

Flow control (далее – контроль потока) означает, что TCP гарантирует, что отправитель не подавляет получателя , отправляя пакеты быстрее, чем тот может их обработать. Это и есть *backpressure* в контексте транспортного уровня модели OSI. Идея в том, что получатель даёт отправителю обратную связь с данными о своём текущем статусе.

Что происходит, когда мы отправляем данные по сети? Отправитель записывает данные в сокет, транспортный уровень (в нашем случае TCP) упаковывает их в сегмент и передаёт их в сетевой уровень (IP), который каким-то путём доставляет пакет получателю.

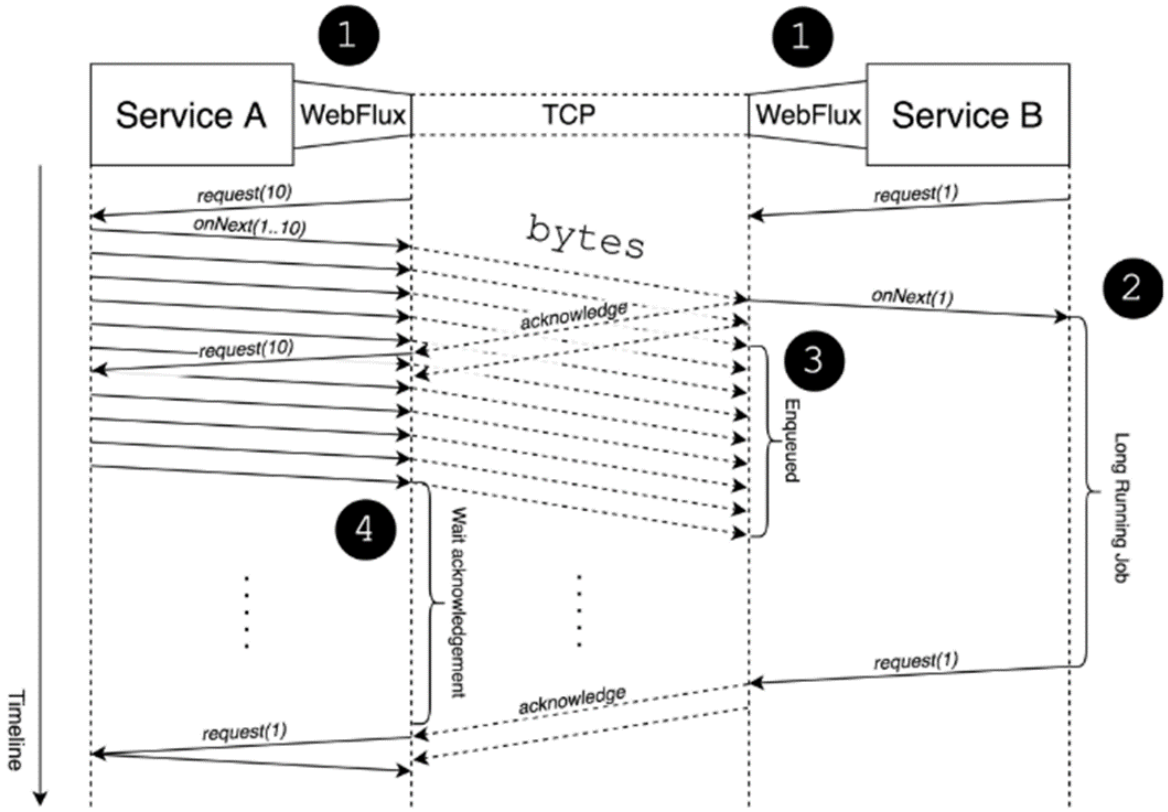
На принимающей стороне сетевой уровень доставляет эти данные до уровня TCP, что делает их доступными для получателя. TCP хранит данные для отправки в общем буфере отправки, а данные для получения – в общем буфере получения. Когда приложение готово, оно читает данные из буфера получения.

Контроль потока гарантирует, что мы не отправляем новых пакетов, если буфер получения уже полон, так как получатель все равно не сможет обработать эти запросы и будет вынужден сбросить (проигнорировать) их. Для управления объемом данных для отправки через TCP, получатель объявляет своё окно (receive window), то есть объем свободного места в буфере получения.

При получении нового пакета TCP отправляет ack сообщение отправителю, подтверждающее, что пакет получен корректно. Вместе с ack сообщением передаётся и размер текущего окна получения, поэтому отправитель знает, что он может или не может отправлять новые данные.

Вернёмся к WebFlux

В текущей реализации WebFlux *backpressure* регулируется транспортным протоколом, но не отражает реального состояния получателя.



https://stackoverflow.com/questions/52244808/backpressure-mechanism-in-spring-web-flux

Диаграмма выше отражает взаимодействие между двумя микросервисами, где левый отправляет потоки данных, а правый – их принимает.

WebFlux берёт на себя работу по преобразованию объектов в байты и обратно для дальнейшей передачи через TCP.

Начало длительной обработки элемента, которая по окончании запрашивает следующий элемент.

В этой точке WebFlux удерживает байты, приходящие по сети, без отправки ack, так как бизнес-логика принимающего микросервиса занята обработкой предыдущего запроса.

Из-за природы контроля потока TCP, левый микросервис может по-прежнему отправлять данные в сеть.

Как видно из диаграммы, demand получателя отличается от demand отправителя (demand измеряем в логических объектах). Это значит, что их demand изолирован и работает только в случае взаимодействия WebFlux с бизнес-уровнем приложения. То есть контроль backpressure в WebFlux не так справедлив, как бы мы хотели.

А если backpressure, но для HTTP-запросов?

Информация о *backpressure* не отправляется по протоколу HTTP (он просто-напросто не поддерживает такой функционал). Поэтому всё снова упирается в TCP flow control и буферы отправки и получения. Demand'ы отправки/получения в **Reactive Streams** также опираются на TCP flow control.

В Spring MVC подразумевается, что приложения могут блокировать текущий поток, например для выполнения удалённых вызовов в другие системы. Поэтому контейнеры сервлетов используют относительно большой пул потоков для сглаживания последствий блокировок.

WebFlux и другие неблокирующие серверы предполагают запуск неблокирующих приложений, поэтому используют небольшой пул потоков фиксированного размера (так называемые event loop workers).

85. Модель потоков Spring WebFlux

На WebFlux сервере можно увидеть следующие потоки:

На оригинальном сервере Spring WebFlux существует один поток для работы сервера и несколько потоков для обработки запросов (обычно их количество равно числу ядер в процессоре).

Реактивный WebClient работает в режиме event loop.

Reactor обеспечивает абстракции пула потоков, которые называются scheduler'ами. Для переключения обработки на другой пул используется метод publishOn(). Планировщики (scheduler'ы) названы в соответствии со стратегией работы, например parallel для CPU-ориентированной параллельной обработки с ограниченным количеством потоков, elastic для I/O – с большим количеством потоков.

Под капотом WebFlux трудится **Reactor-Netty** – обёртка над **Netty** с поддержкой *backpressure*. **Reactor-Netty** создаёт Runtime.getRuntime().availableProcessors() * 2 потоков для использования во внутреннем EventLoopThreadPool. Если входящий запрос завершается блокирующим удалённым вызовом, то его нужно соответствующим образом обернуть, чтобы текущий поток мог вернуться в **event-queue** пул для обработки новых запросов.

Для тяжёлых вычислительных операций имеет смысл использовать выделенный пул потоков. Это гарантирует, что **EventLoop** Netty будет только принимать и отправлять данные по сети, не тратя ресурсы на другие операции:

```
@PostMapping
public Mono<CpuIntensiveResult> cpuIntensiveProcessingHandler(
Mono<CpuIntensiveInput> monoInput
) {
return monoInput
.publishOn(Schedulers.fromExecutorService(myOwnDedicatedExecutor))
.map(i -> doCpuIntensiveInImperativeStyle(i));
}
```

Если интересен ручной контроль количества запросов, можно посмотреть в сторону метода limiRate(int n)

86. Что такое микросервисная архитектура

MSA — принципиальная организация распределенной системы на основе микросервисов и их взаимодействия друг с другом и со средой по сети, а также принципов, направляющих проектирование архитектуры, её создание и эволюцию.

87. Что такое микросервис (MS)

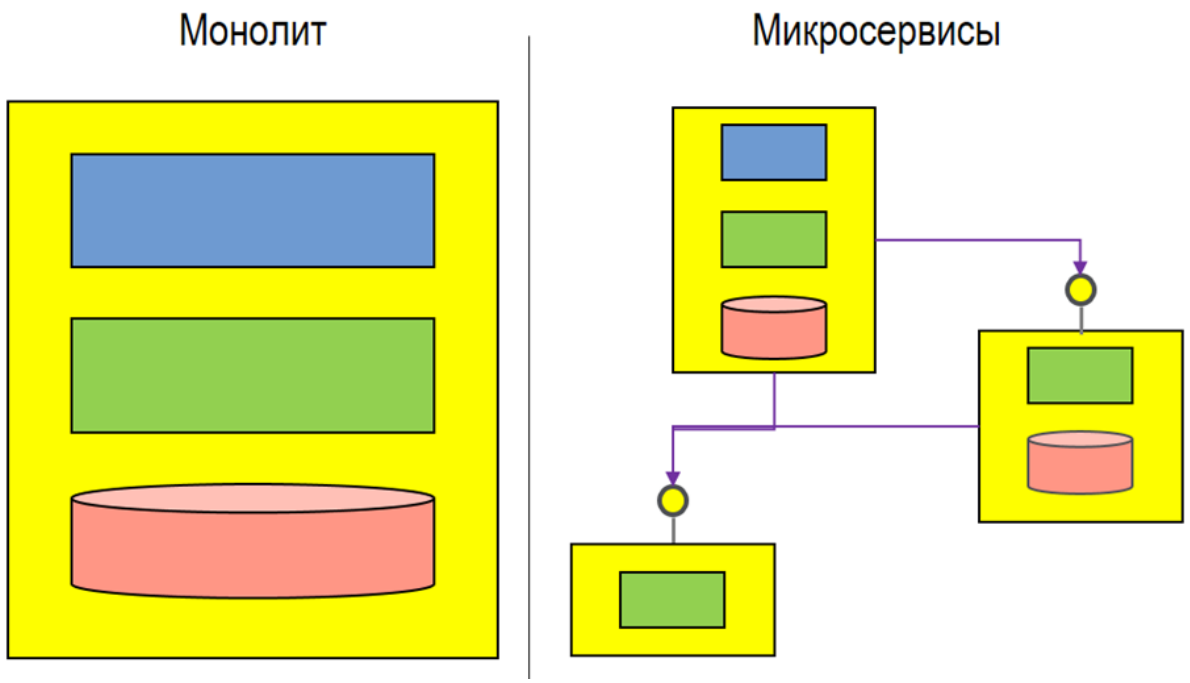
Микросервис — это **веб-сервис, отвечающий за один элемент логики в некой предметной области**. Микросервисы взаимодействуют друг с другом через простые сетевые протоколы, например REST, и совместно выполняют некоторые действия, но при этом ни один из них не имеет представления о внутреннем устройстве других сервисов. Такое согласованное взаимодействие между микросервисами называется микросервисной архитектурой.

88. Что такое Монолит

Монолитная архитектура — это **традиционная модель программного обеспечения, которая представляет собой единый модуль, работающий автономно и независимо от других приложений**. Монолитом часто называют нечто большое и неповоротливое, и эти два слова хорошо описывают монолитную архитектуру для проектирования ПО. Монолитная архитектура — это отдельная большая вычислительная сеть с единой базой кода, в которой объединены все бизнес-задачи

89. Свойства микросервиса

Понять суть микросервиса проще всего на сравнении, или даже противопоставлении его крупному приложению — монолиту.


Монолит
Микросервисы

Восемь свойств микросервиса:

1. Он небольшой.
2. Он независимый.
3. Он строится вокруг бизнес-потребности и использует ограниченный контекст *(Bounded Context)*.
4. Он взаимодействует с другими микросервисами по сети на основе паттерна *Smart endpoints and dumb pipes*.
5. Его распределенная суть обязывает использовать подход *Design for failure*.
6. Централизация ограничена сверху на минимуме.
7. Процессы его разработки и поддержки требуют автоматизации.
8. Его развитие итерационное.

Уже на этом месте те, которые якобы создавали микросервисы в доисторические времена, должны задуматься, действительно ли всё было так продвинуто.

Небольшой

Что такое «небольшой»? Размер микровервиса должен быть таким, чтобы выполнялось одно из условий:

1. Один сервис может развивать одна команда не более чем из дюжины человек.
2. Команда из полудюжины человек может развивать полдюжины сервисов.
3. Контекст (не только бизнеса, но и разработки) одного сервиса помещается в голове одного человека.
4. Один сервис может быть полностью переписан одной командой за одну Agile-итерацию.

Независимый

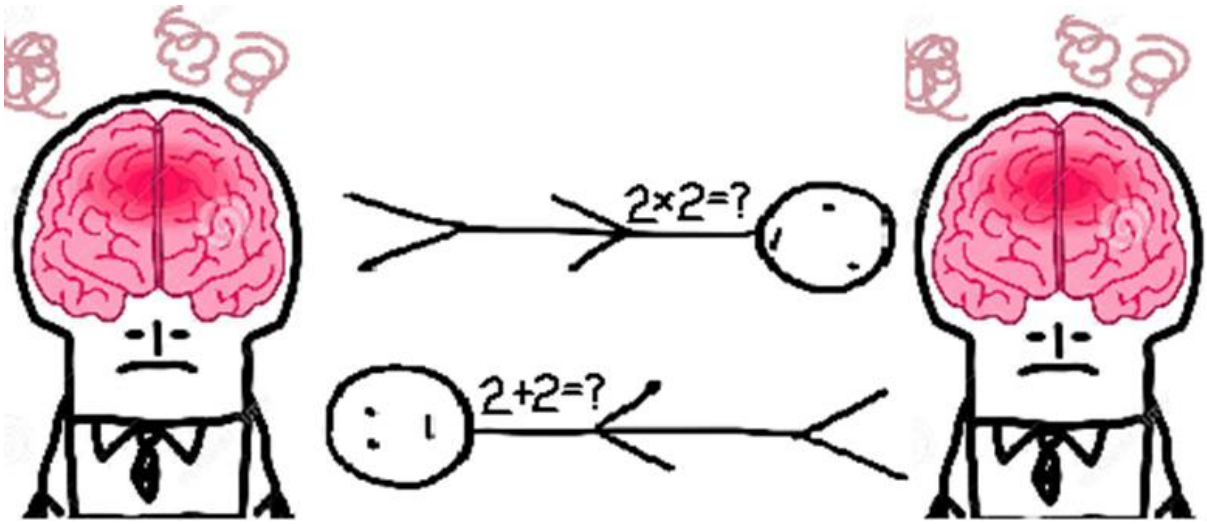
Микросервисная архитектура — воплощение паттернов High Cohesion и Low Coupling. Всё, что противоречит этому, отвергается беспощадно. В противном случае команду ждут большие проблемы. Микросервис обязан быть независимым компонентом. **Компонент** — это единица ПО, код которой может быть независимо заменен или обновлен.

Конечно любая мало-мальски серьезная программа пишется с разбиением на компоненты, которые, безусловно, основываются на тех же принципах. Но в монолите общая кодовая база открывает возможности для нарушения низкой связанности. Под такую формулировку компонента подходят и сторонние библиотеки. В то же время методология разбиения на отдельные микросервисы вынуждает придерживаться жесткого их разделения, ведь они должны отвечать более жестким критериям независимости.

Так, каждый микросервис работает в своем процессе и поэтому должен явно обозначить свой API. Учитывая, что другие компоненты могут использовать только этот API, и к тому же он удаленный, минимизация связей становится жизненно важной.

90. Интеграция микросервисов

Интеграция микросервисов обходится без ESB, как центрального промежуточного звена. Наверное, комьюнити уже натерпелось от неудачных вариантов реализации этого подхода. То, что были и удачные — не принимается в расчет. Впрочем, ESB ещё и противоречит таким критериям как децентрализация и независимость. Таким образом, сложность интеграции распределяется с центрального звена в виде ESB непосредственно на интегрируемые компоненты: «умные конечные точки».



Для интеграции, как правило, используются простые текстовые протоколы, основанные на HTTP, чтобы нивелировать возможную технологическую разность микросервисов. REST-подобные протоколы являются практически стандартом. Как исключение, могут использоваться бинарные протоколы типа Java RMI или .NET Remoting.

Бинарные протоколы гораздо эффективнее. Но, во-первых, появляются технологические ограничения. Во-вторых, на бинарных протоколах сложнее реализовывать шаблон Tolerant Reader, сохраняя эффективность. В-третьих, опять появляется зависимость провайдера и потребителей, поскольку они оперируют одними и теми же объектами и методами, то есть связаны по кодовой базе. Другая отличительная черта взаимодействия микросервисов — синхронные вызовы не приветствуются. Рекомендуется использовать один синхронный вызов на один запрос пользователя, или вообще отказаться от синхронных вызовов. И еще пара замечаний.

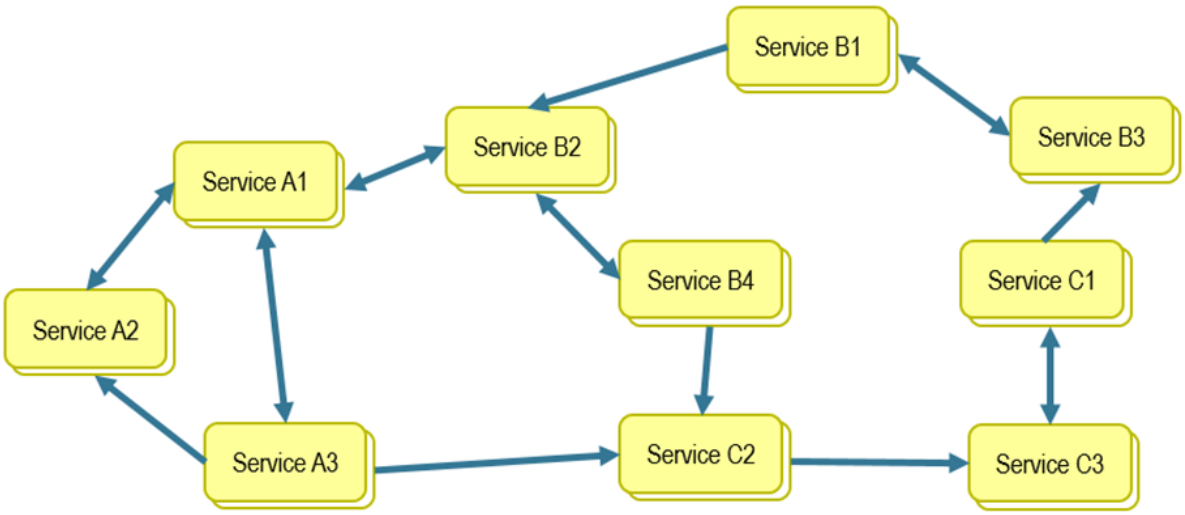
1. Основной сложностью разбиения монолита на микросервисы является не определение их границ. Они уже должны сформироваться и устояться. Сложность заключается в том, что локальные вызовы становятся удаленными. А это влияет не только на организацию вызовов, но и на стиль взаимодействия, так как частые вызовы уже не подходят. Скорее всего, надо пересматривать сам API, делать его более крупным, а, как следствие, пересматривать логику работы компонентов.

2. Поскольку асинхронное событийное взаимодействие — практически стандарт в микросервисной архитектуре, то надо разбираться в создании событийной архитектуры (Event Driven Architecture), а сами микросервисы должны соответствовать требованиям Reactive.

91. Design for failure для распределенной системы

Одно из наиболее критичных мест в микросервисной архитектуре — необходимость разрабатывать код для распределенной системы, составные элементы которой взаимодействуют через сеть. А сеть ненадежна по своей природе. Сеть может просто отказать, может работать плохо, может вдруг перестать пропускать какой-то тип сообщений, потому что изменились настройки файрвола. Десятки причин и видов недоступности.

Поэтому микросервисы могут вдруг перестать отвечать, могут начать отвечать медленнее, чем обычно. И каждый удаленный вызов должен это учитывать. Должен правильно обрабатывать разные варианты отказа, уметь ждать, уметь возвращаться к нормальной работе при восстановлении контрагента. Дополнительный уровень сложности привносит событийная архитектура. А отладку такой системы — не одного микросервиса, а системы, где много потоков разнонаправленных неупорядоченных событий — даже трудно представить. И даже если каждый из микросервисов будет безупречен с точки зрения бизнес-логики, этого мало.



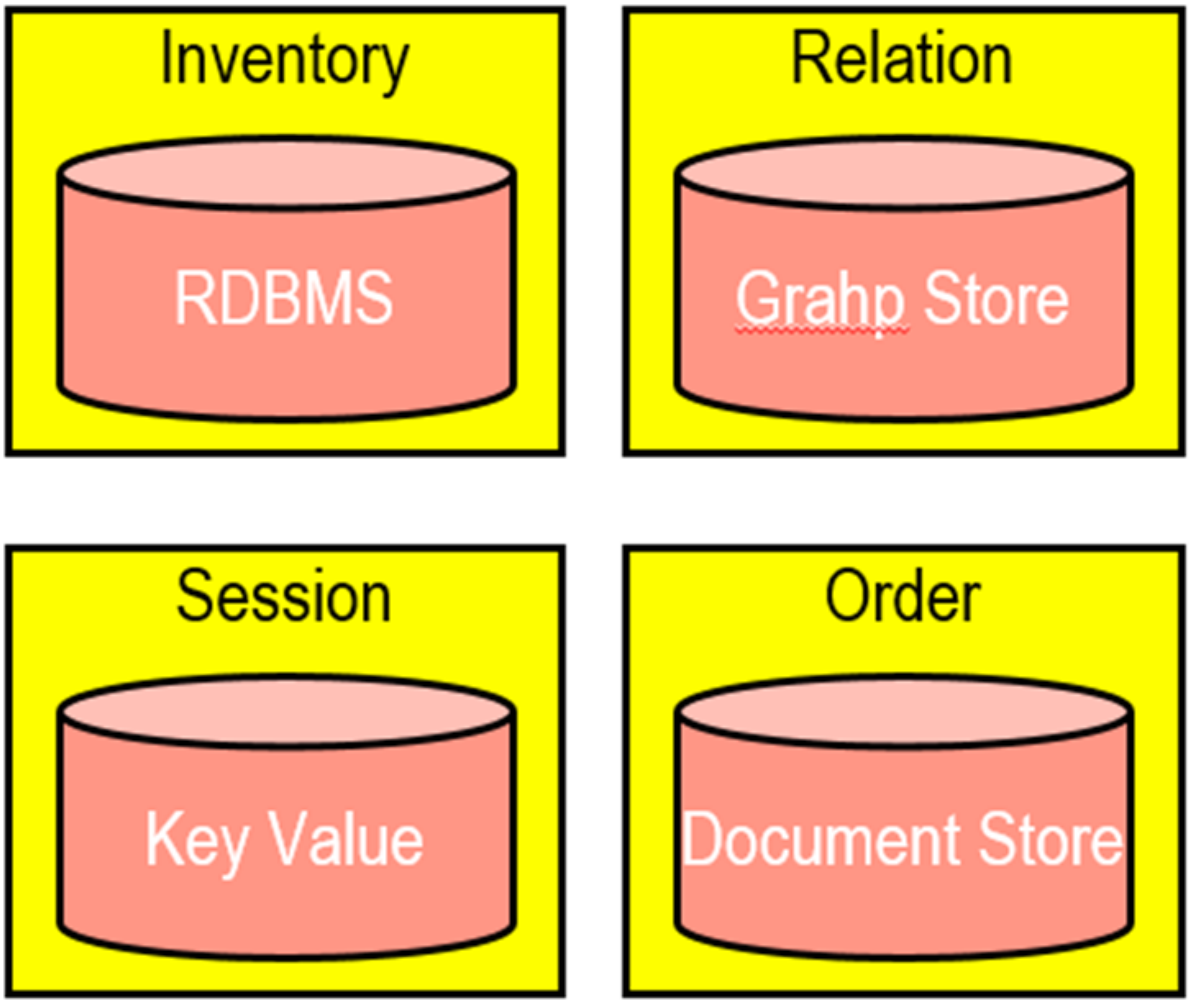
И поскольку сложность таких систем очень высока, то проблему решают так:

- Не доводят систему до состояния «без сучка без задоринки». Это очень дорого. Конечно, это не значит, что система валится от первого дуновения. Она просто отвечает необходимым нефункциональным требованиям. Но в ней могут присутствовать ошибки, незначительно влияющие на ее устойчивость и производительность.
- С другой стороны вкладываются в инфраструктуру, которая помогает быстрее устранять нештатные ситуации. Должно быть полное покрытие кода unit тестами, интеграционными и тестами производительности. Должен быть интеллектуальный мониторинг, который не только моментально показывает неработающие места, но и сигнализирует об ухудшении состояния системы с прогнозированием возможных сбоев. Должно быть продвинутое распределенное логирование, позволяющее оперативно проводить расследования. И часто по результатам исправляются скрытые ошибки.

92. Децентрализация данных

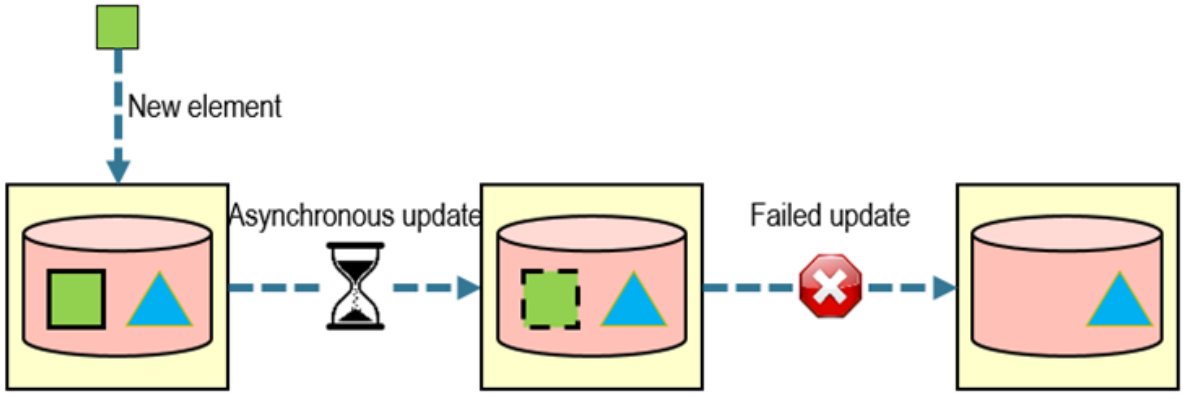
Еще один из важнейших элементов в парадигме микросервисов.

Каждому микросервису по своей базе данных! На самом деле и в монолите можно побороться за изолированность компонентов, например, на уровне серверного кода. Если время от времени изоляция даёт течь, современные инструменты предлагают продвинутые инструменты рефакторинга. Помимо изолированности есть и побочные плюсы. Например, легче реализовать *Polyglot Persistence*, когда база подбирается под конкретные цели. Ничто не мешает делать это и без микросервисов, и так часто делают. Но всё же в одном случае это закон, в другом — исключение.



Много баз, много контекстов, как их все согласовать? Старая техника распределенных транзакций сложна и обладает низкой скоростью. Возможно это иногда можно пережить. А вот необходимость синхронного взаимодействия нескольких микросервисов не может устраивать, и это не побороть. Проблема решается нетрадиционно для монолита: отказом от постоянной согласованности данных. Добро пожаловать в мир *Eventual consistency*. На первых порах это вызывает волну «справедливого» гнева. Но если разобраться, то нужна ли повсеместно немедленная

согласованность данных по окончании транзакции? При детальном рассмотрении значительную часть случаев можно отбросить. Где возможно, заменяют одну распределённую транзакцию серией локальных с компенсационными механизмами.



93. Монолит против микросервисов, плюсы и минусы

Микросервисный подход несет довольно много проблем. Их найти не трудно и каждый может поупражняться. Например, организационные вопросы. Как удержать в согласованном по версиям состоянии сотню микросервисов, которые еще и постоянно и непредсказуемо редеплоятся. А доступ к средам у каждого инженера каждой команды? Какая команда напишет интеграционные тесты? И если кто-то согласится, то попробуй еще их напиши для такой запутанной конфигурации. А если возникает ошибка, то чья она? Только той команды, у которой сломалось? Как не узнать вечером в пятницу, что версия API N-го сервиса, которой вы пользуетесь, вдруг стала deprecated? Да, это действительно проблемы. Но команды, которые практикуют Agile и DevOps, уже знают решение. Поэтому начинать путь к микросервисной архитектуре стоит с внедрения этих практик. Кроме организационных есть и чисто архитектурные. Как перейти от монолита, где всё синхронно, согласованно и едино, к распределенной событийной архитектуры, основанной на множестве мелких элементов, в которой надо учитывать возможную неконсистентность данных? Одного этого достаточно, чтобы задуматься: а стоит ли игра свеч? На этом фоне, например, падение скорости обработки одного запроса кажется мелочью. Хотя бы работает!

Тогда зачем? Если у вас нет проблем с вашим «монолитом», то не надо их искать. Но если проблемы есть, то посмотрите на плюсы MSA, и возможно она спасет вас.

Разбиение на независимые компоненты даёт безусловные и неоспоримые преимущества: легкое понимание контекста, гибкость развития, управления и масштабирования. Независимость и небольшой размер дают и неожиданные плюсы с точки зрения инфраструктуры. Вам теперь не нужна монструозная машина за 100500 долларов. Микросервисы можно устанавливать на обычные дешевые машинки. И окажется, что даже все вместе они будут стоить на порядок меньше, но работать эффективнее той самой супермашины, на которую у вас в организации, наверняка, молятся и сдувают с неё пылинки. Здесь уместен другой лозунг от популиста. Хотя, как и предыдущий, он вполне серьезен.

Каждому микросервису по своему серверу!

Посмотрим на лидеров IT-индустрии: Amazon, Netflix, Google и другие показывают впечатляющие результаты. Их гибкость и скорость вывода новых продуктов поражают. Поэтому игра точно стоит свеч! Здесь уместно вспомнить, что в упомянутых организациях команд «уровня бог» не одна и не две. Им сложности микросервисной архитектуры вполне по зубам. И если предложить создать монолит, то они и его сделают так, что он будет сверкать путеводной звездой. А, например, Amazon вполне себе работал на

монолите, уже будучи гигантом и имея миллиардные обороты. Сайт газеты Guardian до сих пор, а возможно и навсегда, базируется на микросервисах вокруг монолита. Это говорит о том, что значительная часть задач успешно, а зачастую и легче, решается без привлечения микросервисов.

94. Шаблоны проектирования CQRS и Event Sourcing

Event sourcing смотри вопрос 96.

CQRS - Command and Query Responsibility Segregation паттерн

CQRS расшифровывается как Command Query Responsibility Segregation (разделение ответственности на команды и запросы). В 1980 Бертран Мейер сформулировал термин CQS. В начале двухтысячных Грег Янг расширил и популяризовал эту концепцию к CQRS. CQRS предлагает разделять операции чтения и записи на отдельные типы операций Query и Commands.

- Command ориентированы на задачи, а не на данные. ("Забронировать номер в отеле", а не установить для ReservationStatus значение "зарезервировано").
- Command может помещаться в очередь для асинхронной обработки, а не обрабатываться синхронно.
- Query никогда не должен изменять базу данных. Query возвращает DTO, который не инкапсулирует знания предметной области.

Схематически работу клиента по CQRS модели можно отобразить следующим образом:

Наличие отдельных моделей запросов и команд упрощает проектирование таких систем как независимых друг от друга. Однако один из недостатков заключается в том, что код CQRS не может автоматически формироваться из схемы базы данных с помощью ORM или подобных механизмов. Для дополнительной изоляции часто физически разделяют данные для чтения и данные для записи. Как это описано на диаграмме выше. В этом случае в БД для чтения можно оптимизировать ее работу так,

чтобы максимально эффективно выполнять запросы. К примеру заюзать *materialized view*, чтобы не использовать сложные операции join'ов или сложные связи. Вы можете использовать в том числе другой тип хранилища данных. Например, база данных для записи останется реляционной, а для чтения вы можете применять NoSQL или наоборот, в зависимости от бизнес задач. Если вы пошли по пути отдельных БД для чтения и записи, они должны поддерживать синхронизацию. Обычно это реализуют с помощью событий при каждом обновлении базы данных. Обновление базы данных и публикации события должны выполняться в рамках одной транзакции.

95. Преимущества и недостатки CQRS

Преимущества CQRS

- **Независимое масштабирование**. CQRS позволяет раздельно масштабировать рабочие нагрузки чтения и записи, снижая риск конфликтов блокировки.
- **Оптимизированные схемы данных**. Для query применить схему, оптимизированную для запросов, а commands — другую схему, оптимизированную для обновлений.
- **Безопасность**. Разделение операций позволит настроить более гибкую систему доступа.
- **Разделение проблем**. Разделение операций позволяет получить более гибкие и простые в обслуживании классы.
- **Более простые запросы**. Сохраняя в базе данных для чтения materialized view, вы предотвратите использование сложных запросов и join'ов.
- **Не требует 2 хранилища данных**. Отдельные хранилища для query и command это одна из реализаций, а не обязательное требование

Недостатки CQRS

- **Сложность.** Основная идея CQRS звучит просто. Но ее реализация может привести к усложнению проекта приложения, особенно если реализовывать его в связке с Event Sourcing.

- **Обмен сообщениями**. Сама по себе модель CQRS не требует месседжинга, но месседж брокеры часто применяются для обработки команд и публикации событий. Это означает, нужно будет реализовывать обработку сбоев и дубликатов при передаче сообщений.
- **Eventual consistency**. Если вы разделите базы данных для чтения и записи, в базе данных для чтения могут оставаться устаревшие данные. БД для чтения должна быть up to date, чтобы отражать изменения из БД для записи, и может быть трудно трекать, когда пользователь сделал запрос на основе устаревших данных с БД для чтения.

96. Event Sourcing

Event sourcing (источники событий, регистрация событий, генерация событий) — это архитектурный паттерн, в котором все изменения, вносимые в состояние приложения, сохраняются в той последовательности, в которой они были выполнены. Эти записи служат как источником для получения текущего состояния, так и audit-log'OM того, что происходило в системе. Event sourcing способствует децентрализованному изменению и чтению данных. Такая архитектура хорошо масштабируется и подходит для систем, которые уже работают с событиями или подходят для миграции на такую архитектуру.

Преимущества Event Sourcing

- События immutable, и их можно сохранить с помощью append-only операции.
- События могут раниться на фоне.
- Event sourcing может помочь в предотвращении конфликтов, вызванных параллельными апдейтами, тк исключает необходимость непосредственного обновления объектов в Data store. Однако доменная модель должна уметь себя защищать от запросов, которые могут вызвать несогласованное состояние.
- Append-only storage предоставляет audit log, который можно использовать для мониторинга событий, произошедших в Data store, повторного создания текущего состояния в виде materialized view или

проекций путем воспроизведения событий в любое время, а также упрощения тестирования и отладки системы.

- Каждое событие могут обрабатывать несколько задач. Это обеспечивает простую интеграцию с другими службами и системами, которые только слушают новые события, вызванные data stor'ом. Однако event sourcing events зачастую являются низкоуровневыми, из -за чего может потребоваться создание определенных событий интеграции.

Недостатки Event Sourcing

- Самые большие сложности обычно связаны с перестроением мышления разработчиков. Разработчики должны забыть про обычные CRUD-приложения и хранилища сущностей. Теперь основной концепцией становятся события.
- При Event Sourcing много сил тратится на моделирование событий. После сохранения событий в сторедж они должны быть immutable, иначе история и состояние могут быть повреждены или искажены. Event Log — это исходные данные, а это значит, что необходимо очень внимательно следить за тем, чтобы они содержали всю информацию, необходимую для получения полного состояния системы на определенный момент времени. Также необходимо учитывать, что события могут интерпретироваться повторно, поскольку система (и бизнес, который она представляет) со временем изменяются.
- Для простой бизнес логики переход на Event Sourcing может быть довольно легким, но для более сложных может стать проблемой (особенно с большим количеством зависимостей и отношений между сущностями). Так же могут возникнуть сложности интеграции с внешними системами, которые не предоставляют данные на определенный момент времени.
- Event Sourcing может работать хорошо в больших системах, так как паттерн «Event Log» естественным образом масштабируется горизонтально. Например, event log одной сущности необязательно должен физически находиться вместе с журналом событий другой сущности. Однако, такая легкость масштабирования приводит к дополнительным проблемам в виде решения проблем и реализацией eventual consistency.

- Важно учитывать структуру событий. Структура событий может измениться в какой-то момент, например набор полей. Могут возникнуть ситуации, когда исторические события должны быть обработаны текущей бизнес-логикой. И наличие расширяемой схемы событий поможет в будущем при необходимости отличать новые события от старых. Периодические снапшоты также помогают отделить серьезные изменения структуры событий.

97. Когда следует и не следует использовать Event Sourcing

Когда следует использовать Event Sourcing

- Когда в данные необходимо записать намерение, цель или причину. Например, изменения в сущности клиента можно записать как ряд определенных типов событий, таких как *Возвращение к исходному, Закрытая учетная запись* или *Недействительные.*
- Когда очень важно свести к минимуму или полностью избежать конфликта операций обновления данных.
- Если требуется записывать происходящие события и иметь возможность воспроизвести их для восстановления определенного состояния системы, отката изменений или сохранения истории и audit - log. Например, если задача включает несколько шагов, необходимых для восстановления обновлений и последующего воспроизведения некоторых действий для восстановления согласованного состояния данных.
- Когда использование событий представляет собой стандартную возможность операции приложения и требует некоторой дополнительной разработки или усилий в отношении реализации.
- Если нужно разбить процесс ввода или обновления данных из задач, необходимых для применения этих действий. Это может быть в целях улучшения производительности пользовательского интерфейса или распределения событий в другие прослушиватели, выполняющие определенные действия при возникновении событий. Например, интеграция платежной системы с веб-сайтом о расходах требуется для того, чтобы события, вызванные с помощью хранилища событий в ответ на обновления данных, реализованные для веб-сайта, использовались как веб-сайтом, так и платежной системой.
- Если необходима гибкость для изменения формата материализованных моделей и данных сущности при изменении

требований или — использовании в сочетании с CQRS, необходимо адаптировать модель чтения или представления с данными.
- Если используется в сочетании с CQRS, eventual consistency допустима при обновлении модели чтения или допустимо влияние на производительность при восстановленных сущностях и данных из потока события.

Когда не следует использовать Event Sourcing

- Для небольших или простых доменов, систем, которые обычно хорошо взаимодействуют со стандартными механизмами управления данных CRUD.
- Систем, где для представления данных требуются согласованность и обновления в режиме реального времени.
- Систем, где для действий отката и воспроизведения не требуются определенные функции, история и audit-log.
- Систем, где имеется незначительный конфликт обновлений в базовых данных. Например, это системы, которые преимущественно добавляют данные, а не обновляют их.

98. Шаблон Saga

Ранее мы упоминали, что распределение транзакций между микросервисами может быть проблематичным. Проще говоря, транзакция будет успешной, только если все связанные службы успешно выполнят свою часть. В случае сбоя в одной службе вся транзакция должна завершиться неудачей. Более того, в этом случае службы, которые уже внесли свой вклад, должны откатить изменения.

В общем, за это отвечает шаблон saga. **Шаблон Saga** - это последовательность локальных транзакций, которые представляют собой единую распределенную транзакцию. Каждая служба выполняет локальную транзакцию. Если локальная транзакция завершается успешно, публикуется событие или сообщение, которое запускает следующую локальную транзакцию в последовательности. В случае сбоя saga предоставляет компенсирующие транзакции, которые откатывают изменения.

Существует два типа реализации шаблона saga :

- Управление – центральный контроллер (orchestrator) управляет всеми взаимодействиями между микросервисами
- Хореография – децентрализованный метод трансляции событий

99. SQL против NoSQL

Когда необходимо выбрать СУБД, главный вопрос обычно заключается в выборе реляционной (SQL) или нереляционной (NoSQL) структуры. У обоих вариантов есть свои преимущества, а также несколько ключевых особенностей, которые стоит иметь в виду при выборе.

Основные различия

Язык

Реляционные базы данных используют структурированный язык запросов (Structured Query Language, SQL) для определения и обработки данных. С одной стороны, это открывает большие возможности для разработки: SQL один из наиболее гибких и распространённых языков запросов, так что его выбор позволяет минимизировать ряд рисков, и будет особенно кстати, если предстоит работа с комплексными запросами. С другой стороны, в SQL есть ряд ограничений. Построение запросов на этом языке обязывает предопределять структуру данных, то есть последующее изменение структуры данных может быть губительным для всей системы.

Нереляционные базы данных, в свою очередь, предлагают динамическую структуру данных, которые могут храниться несколькими способами: ориентированно по колонкам, документо-ориентированно, в виде графов или на основе пар «ключ-значение». Такая гибкость означает следующее:

- Вы можете создавать документы, не задавая их структуру заранее;
- Каждый документ может обладать собственной структурой;
- У каждой базы данных может быть собственный синтаксис;
- Вы можете добавлять поля прямо во время работы с данными.

Масштабируемость

В большинстве случаев SQL базы данных вертикально масштабируемые, то есть вы можете увеличивать нагрузку на отдельно взятый сервер, наращивая мощность центральных процессоров, объёмы ОЗУ или системы хранения данных. А NoSQL базы данных горизонтально масштабируемы. Это означает, что вы можете увеличивать трафик, распределяя его или добавляя больше серверов к вашей СУБД. Всё равно, что добавлять больше этажей к вашему зданию, либо добавлять больше зданий на улицу. Во втором случае, система может стать куда больше и мощнее, делая выбор NoSQL базы данных предпочитаемым для больших или постоянно меняющихся структур данных.

Структура

В реляционных СУБД данные представлены в виде таблиц, в то время как в нереляционных — в виде документов, пар «ключ-значение», графов или wide-column хранилищ. Это делает SQL базы данных лучшим выбором для приложений, которые предполагают транзакции с несколькими записями — как, например, система учётных записей — или для устаревших систем, которые были построены для реляционных структур.

В число СУБД для SQL баз данных входят MySQL, Oracle, PostgreSQL и Microsoft SQL Server. Для работы с NoSQL подойдут MongoDB, BigTable, Redis, RavenDB Cassandra, HBase, Neo4j и CouchDB.

SQL против NoSQL

Разобравшись с ключевыми структурными различиями SQL и NoSQL баз данных, стоит внимательно рассмотреть их функциональные особенности на примере MySQL и MongoDB.

MySQL: реляционная СУБД

Преимущества MySQL:

- Проверено временем: MySQL — крайне развитая СУБД, что означает наличие большого сообщества вокруг неё, множество примеров и высокую надёжность;
- Совместимость: MySQL доступна на всех основных платформах, включая Linux, Windows, Mac, BSD и Solaris. Также у неё есть

библиотеки для языков вроде Node.js, Ruby, C#, C++, Java, Perl, Python и PHP;
- Окупаемость: Это СУБД с открытым исходным кодом, находящаяся в свободном доступе;
- Реплицируемость: Базу данных MySQL можно распределять между несколькими узлами, таким образом уменьшая нагрузку и улучшая масштабируемость и доступность приложения;
- Шардинг: В то время как шардинг невозможен на большинстве SQL баз данных, MySQL является исключением.

MongoDB: нереляционная СУБД

Преимущества MongoDB:

- Динамическая схема: Как упоминалось выше, эта СУБД позволяет гибко работать со схемой данных без необходимости изменять сами данные;
- Масштабируемость: MongoDB горизонтально масштабируема, что позволяет легко уменьшить нагрузку на сервера при больших объёмах данных;
- Удобство в управлении: СУБД не нуждается в отдельном администраторе базы данных. Благодаря достаточному удобству в использовании, ей легко могут пользоваться как разработчики, так и системные администраторы;
- Скорость: Высокая производительность при выполнении простых запросов;
- Гибкость: В MongoDB можно без вреда для существующих данных, их структуры и производительности СУБД добавлять поля или колонки.

Какую СУБД выбрать?

MySQL — верный выбор для любого проекта, который может положиться на предопределённую структуру и заданные схемы. С другой стороны, MongoDB — отличный вариант для быстрорастущих проектов без определённой схемы данных. В особенности если вы не можете определить схему для своей базы данных, вам не подходит ни одна из предлагаемых другими СУБД или в вашем проекте она постоянно

меняется, как, например, в случае с мобильными приложениями, системами аналитики в реальном времени или контент-менеджмента.

100. CAP теорема

Эта теорема была представлена на симпозиуме по принципам распределенных вычислений в 2000 году Эриком Брюером. В 2002 году Сет Гилберт и Нэнси Линч из MIT опубликовали формальное доказательство гипотезы Брюера, сделав ее теоремой. По словам Брюера, он хотел, чтобы сообщество начало дискуссию о компромиссах в распределённых системах и уже спустя некоторое количество лет начал вносить в неё поправки и оговорки.

Что стоит за CAP

В CAP говорится,что в распределенной системе возможно выбрать только 2 из 3-х свойств:

- C (consistency) — согласованность. Каждое чтение даст вам самую последнюю запись.
- A (availability) — доступность. Каждый узел (не упавший) всегда успешно выполняет запросы (на чтение и запись).
- P (partition tolerance) — устойчивость к распределению. Даже если между узлами нет связи, они продолжают работать независимо друг от друга.

Пример

Посмотрим на Postgresql

Следующие пункты относятся к абстрактной распределенной БД Postgresql:

- Репликация Master-Slave—одно из распространенных решений
- Синхронизация с Master в асинхронном/ синхронном режиме
- Система транзакций использует двухфазный коммит для обеспечения consistency
- Если возникает partition, вы не можете взаимодействовать с системой (в основном случае)

Таким образом, система не может продолжать работу в случае partition, но обеспечивает strong consistency и availability. Это система CA!