HighLoad в микросервисах

Как спроектировать качественный микросервис и не обосраться

Проектирование микросервисов – Издание II

Автор: Владимир Желнов

Год издания: 2024

Принципы проектирования микросервисов

Проектирование микросервисов — это подход к архитектуре приложения, при котором приложение разбивается на маленькие, автономные, легко масштабируемые и управляемые сервисы. Вот несколько основных принципов проектирования микросервисов:

1. **Разделение по бизнес-контексту:**
   Каждый микросервис должен решать конкретную бизнес-задачу или отвечать за определенный функционал. Это позволяет достичь легкости понимания, разработки и сопровождения каждого сервиса.

2. **Автономность:**
   Микросервисы должны быть автономными, что означает, что они могут разрабатываться, тестироваться, разворачиваться и масштабироваться независимо друг от друга. Это уменьшает связанность и упрощает процесс разработки.

3. **Ответственность за данные:**
   Каждый микросервис должен быть ответственным за свои собственные данные. Избегайте общих баз данных и старайтесь минимизировать зависимость от данных, которые управляются другими сервисами.

4. **Композиция через API:**
   Микросервисы должны взаимодействовать между собой через четкие и стабильные API. Хорошо определенные API упрощают согласование и интеграцию между сервисами.

5. **Гибкая архитектура:**
   Архитектура микросервисов должна быть гибкой и легко изменяемой. Это позволяет адаптироваться к изменениям в бизнес-требованиях и технологическим инновациям.

6. **Распределенные данные и согласованность:**
   Работа с распределенными данными требует внимания к вопросам согласованности и согласованности данных между микросервисами. Выбор метода согласования (как, например, событийное согласование) зависит от конкретных требований приложения.

7. **Использование контейнеров и оркестраторов:**

Используйте контейнеры (например, Docker) и оркестраторы (например, Kubernetes) для упрощения развертывания и управления микросервисами. Это обеспечивает надежность и масштабируемость.

8. **Мониторинг и отказоустойчивость:**
   Обеспечьте мониторинг состояния микросервисов и предусмотрите механизмы отказоустойчивости. Это важно для выявления проблем и обеспечения бесперебойной работы системы.
9. **Независимое масштабирование:**
Каждый микросервис должен масштабироваться независимо от других. Это позволяет выделять ресурсы в соответствии с потребностями конкретных сервисов.

10. **Быстрое развертывание и CI/CD:**
Внедряйте методы непрерывной интеграции (CI) и непрерывной доставки (CD) для автоматизации процесса разработки, тестирования и развертывания микросервисов.

11. **Безопасность:**
Обеспечьте безопасность каждого микросервиса и общей системы. Используйте методы аутентификации и авторизации, а также обеспечьте защиту данных в пути и покое.

Проектирование микросервисов — это комплексная задача, требующая учета множества факторов. Однако с соблюдением данных принципов можно создать гибкую, масштабируемую и легко управляемую архитектуру микросервисов.

Проектирование HighLoad микросервиса

Для проектирования микросервиса пользователей, который способен выдерживать высокие запросы в секунду (RPS), следует учитывать несколько ключевых аспектов:

1. **Горизонтальное масштабирование:**
Разрабатывайте микросервис так, чтобы его было легко масштабировать горизонтально. Используйте контейнеры (например, Docker) и оркестраторы контейнеров (например, Kubernetes), чтобы легко добавлять новые экземпляры сервиса по мере необходимости.

2. **База данных:**

Рассмотрите использование распределенной базы данных или кэширование для обработки запросов. Можно разделить данные по пользователям так, чтобы каждый микросервис работал с подмножеством данных, и масштабировать их независимо.

3. **Кэширование:**

Используйте кэширование для хранения часто запрашиваемых данных в близком к микросервису хранилище (например, Redis). Это может существенно уменьшить нагрузку на базу данных.

4. **Асинхронность:**

Рассмотрите использование асинхронных обработчиков для некоторых операций, особенно если они могут быть обработаны позже. Например, асинхронная обработка отправки электронной почты или обновления статистики.

5. **Кэширование на уровне HTTP:**

Используйте механизмы кэширования HTTP (например, HTTP-заголовки Cache-Control) для кэширования ответов на запросы, которые не требуют актуальных данных.

6. **Отказоустойчивость:**

Внедрите механизмы отказоустойчивости, такие как обработка ошибок, резервирование ресурсов, и в случае возможности, отказоустойчивость базы данных.

7. **Мониторинг и логирование:**

Разверните системы мониторинга и логирования, чтобы отслеживать производительность микросервиса и идентифицировать узкие места и проблемы.

8. **Оптимизация кода и запросов:**

Постоянно оптимизируйте код и запросы к базе данных. Избегайте избыточных запросов и используйте эффективные индексы.

9. **Тестирование производительности:**

Проводите тестирование производительности, чтобы оценить, как микросервис обрабатывает нагрузку. Это может помочь выявить узкие места и оптимизировать систему.

10. **Внимание к сетевой архитектуре:**

Учитывайте сетевую архитектуру. Оптимизируйте сетевые запросы, уменьшайте задержки, и используйте CDN (Content Delivery Network), если это целесообразно.

Применение этих подходов в комбинации может помочь создать микросервис для пользователей, который способен выдерживать высокую нагрузку и обеспечивать высокую производительность. Однако конкретные решения могут зависеть от деталей вашего приложения и требований.

Примеры готовых стеков технологий для микросервиса

1. Java-ориентированный стек

Язык программирования: Java
Фреймворк: Spring Boot
Контейнеризация и оркестрация: Docker, Kubernetes
База данных: PostgreSQL, MongoDB
Протоколы: RESTful API
Мониторинг и трассировка: Prometheus, Jaeger
CI/CD: Jenkins, GitLab CI
JavaScript/Node.js стек:

2. Язык программирования: JavaScript/Node.js

Фреймворк: Express.js
Контейнеризация и оркестрация: Docker, Kubernetes
База данных: MongoDB, Redis
Протоколы: RESTful API, GraphQL
Мониторинг и трассировка: Prometheus, Zipkin
CI/CD: Jenkins, Travis CI
Go-ориентированный стек:

3. Язык программирования: Go (Golang)

Фреймворк: Gin
Контейнеризация и оркестрация: Docker, Kubernetes
База данных: CockroachDB, etcd
Протоколы: RESTful API
Мониторинг и трассировка: Prometheus, Jaeger

CI/CD: Jenkins, GitLab CI

4. Python-ориентированный стек:

Язык программирования: Python
Фреймворк: Flask, Django
Контейнеризация и оркестрация: Docker, Kubernetes
База данных: SQLAlchemy, Cassandra
Протоколы: RESTful API
Мониторинг и трассировка: Prometheus, Zipkin
CI/CD: Jenkins, Travis CI

5. .NET-ориентированный стек

Язык программирования: C#
Фреймворк: ASP.NET Core
Контейнеризация и оркестрация: Docker, Kubernetes
База данных: SQL Server, Redis
Протоколы: RESTful API
Мониторинг и трассировка: Prometheus, Jaeger
CI/CD: Azure DevOps, Jenkins

Способы взаимодействия между микросервисами

В микросервисной архитектуре существует несколько способов взаимодействия между микросервисами. Каждый из них имеет свои преимущества и подходит для различных сценариев. Вот несколько основных способов взаимодействия между микросервисами:

1. HTTP/REST API

Взаимодействие через HTTP/REST API является одним из наиболее распространенных способов связи между микросервисами. Каждый микросервис предоставляет API для выполнения операций с данными, и другие микросервисы могут отправлять HTTP-запросы для доступа к функционалу.

2. Message Queues (очереди сообщений)

Использование систем сообщений, таких как RabbitMQ, Apache Kafka или AWS SQS, позволяет микросервисам обмениваться сообщениями асинхронно. Это особенно полезно для обработки асинхронных событий, а также для управления асинхронными задачами.

3. gRPC

gRPC - это открытый и высокопроизводительный протокол взаимодействия, который использует протокол буферизации Google (protobuf). *Он обеспечивает эффективное и типизированное взаимодействие* между микросервисами и может использоваться для передачи данных и вызова удаленных процедур.

4. GraphQL

GraphQL - это язык запросов для вашего API, который позволяет клиентам запрашивать только те данные, которые им нужны. Это может быть полезным, когда клиенту требуется специфическая информация, и позволяет сократить количество необходимых запросов.

5. Service Mesh

Service mesh (например, Istio или Linkerd) представляет собой инфраструктуру, предоставляющую слой управления и мониторинга для микросервисов. Он может управлять сетевым взаимодействием, обеспечивать безопасность и обнаружение сервисов.

6. Synchronous RPC (Remote Procedure Call)

Микросервисы могут использовать удаленные вызовы процедур для синхронного взаимодействия. Однако это часто считается *менее желательным в микросервисной архитектуре* из-за возможных проблем с устойчивостью системы и сложностью обработки ошибок.

7. Database Communication

Микросервисы могут взаимодействовать напрямую с базой данных другого микросервиса. Однако этот подход может привести к *проблемам с прозрачностью и управлением данными*.

8. Event-Driven Communication

Взаимодействие на основе событий может использоваться для распределенной обработки событий и уведомлений. Микросервисы могут публиковать и подписываться на события, обеспечивая асинхронный обмен данными.

Выбор конкретного метода взаимодействия зависит от требований приложения, контекста использования и предпочтений разработчиков. В большинстве случаев комбинация нескольких методов обеспечивает гибкость и соответствие требованиям приложения.

Плюсы и минусы gRPC

gRPC (gRPC Remote Procedure Call) - это фреймворк для разработки удаленных API, основанный на протоколе Protocol Buffers и предназначенный для эффективного взаимодействия между клиентом и сервером. Вот некоторые плюсы и минусы использования gRPC:

Плюсы gRPC

1. **Высокая производительность:**

gRPC использует протокол бинарной сериализации Protocol Buffers, что обеспечивает более компактное и эффективное представление данных. Это сокращает объем сетевого трафика и улучшает производительность.

2. **Языковая независимость:**

gRPC поддерживает множество языков программирования, включая Java, C++, Python, Go, Node.js, и другие. Это обеспечивает гибкость в выборе языка для клиентской и серверной частей.

3. **Сильная типизация данных:**

Использование Protocol Buffers предоставляет сильную типизацию данных, что облегчает разработку и поддержание кода. Клиент и сервер могут явно определить формат и структуру обмена данными.

4. **HTTP/2 протокол:**

gRPC базируется на протоколе HTTP/2, который обеспечивает многопоточное и многозадачное взаимодействие, а также поддержку долгоживущих соединений. Это улучшает эффективность передачи данных.

5. **Code Generation:**

gRPC автоматически генерирует клиентский и серверный код на основе определений служб и сообщений в файле Protocol Buffers. Это упрощает процесс разработки и обеспечивает согласованность интерфейса.

6. **Поддержка различных платформ:**

gRPC предоставляет библиотеки и поддержку для различных платформ, включая мобильные устройства (Android, iOS), веб-приложения и серверы.
Механизмы аутентификации и безопасности:

gRPC включает в себя встроенную поддержку механизмов аутентификации и безопасности, таких как SSL/TLS. Это обеспечивает защищенное взаимодействие между клиентом и сервером.

Минусы gRPC

1. **Сложность отладки:**

Использование бинарного формата данных и автоматической генерации кода может усложнить отладку в случае проблем.

2. **Человекочитаемость:**

Формат данных Protocol Buffers не так легко читаем, как, например, JSON. Это может затруднить отладку и тестирование вручную.

3. **Сложность миграции:**

Переход с существующих систем, основанных на других технологиях, на gRPC может потребовать значительных усилий, особенно если используется другой формат обмена данными.

4. **Зависимость от HTTP/2:**

Хотя HTTP/2 обеспечивает высокую производительность, некоторые сети или прокси-серверы могут не полностью поддерживать этот протокол.

5. **Ограничения веб-браузеров:**

Прямая поддержка gRPC в веб-браузерах ограничена, что может создавать сложности для использования этой технологии в веб-приложениях.

6. **Неудобство при работе с большими данными:**

В случае передачи больших объемов данных, особенно неструктурированных, преимущество компактности Protocol Buffers может быть уменьшено.

7. **Затраты на обучение:**

Внедрение gRPC может потребовать времени и затрат на обучение команды, особенно если ранее использовались другие технологии взаимодействия.

В целом, gRPC является мощным и эффективным инструментом для построения распределенных систем, но его использование требует внимательного обдумывания, особенно с учетом конкретных потребностей и ограничений проекта.

Поддержка gRPC в Nginx

Да, начиная с версии 1.13.10, Nginx поддерживает проксирование gRPC-запросов. Это означает, что вы можете использовать Nginx в качестве прокси для ваших gRPC-сервисов.

Для использования gRPC с Nginx, вам нужно удостовериться, что ваша версия Nginx поддерживает gRPC и включает модуль ngx_http_grpc_module. После этого вы можете настроить ваше конфигурационное файла Nginx для обработки gRPC-запросов.

Пример конфигурации для проксирования gRPC-запросов с использованием Nginx может выглядеть примерно так:

```
server {
    listen 80;

    location /your_grpc_service {
        grpc_pass grpc://localhost:50051;

        # Опциональные заголовки, которые вы можете настроить
        # grpc_set_header ...;
        # grpc_read_timeout ...;
    }
}
```

В этом примере, /your_grpc_service - это путь, который будет проксироваться к вашему gRPC-сервису, и grpc://localhost:50051 - это адрес и порт вашего gRPC-сервиса.

Убедитесь, что у вас установлена поддержка gRPC и настройте конфигурацию Nginx в соответствии с вашими требованиями.

Передача параметров из URL в gRPC на основе Nginx

При работе с gRPC в Nginx, вы можете передавать параметры запроса с использованием стандартных возможностей Nginx для работы с параметрами URL. Однако, в gRPC запросы обычно передаются в теле HTTP-запроса, а не как параметры URL, как это делается в традиционных HTTP REST-запросах.

Пример настройки передачи параметров запроса через URL с использованием Nginx:

```
server {
    listen 80;

    location /your_grpc_service {
        grpc_pass grpc://localhost:50051;

        # Опциональные заголовки, которые вы можете настроить
        grpc_set_header ...;
        grpc_read_timeout ...;

        # Передача параметров запроса через gRPC-заголовки
        grpc_set_header param1 $arg_param1;
        grpc_set_header param2 $arg_param2;
    }
}
```

В этом примере параметры запроса (param1 и param2) передаются через заголовки gRPC, и их значения берутся из параметров URL запроса с помощью переменных **$arg_param1** и **$arg_param2**.

Если вам нужно передавать параметры в теле запроса, то в gRPC обычно используется protobuf для сериализации данных. В таком случае, параметры могут быть переданы в теле запроса в сериализованном формате (например, JSON или Protocol Buffers).

Убедитесь, что ваш gRPC-сервис поддерживает прием параметров в том формате, который вы выбрали, и что вы корректно обрабатываете их в вашем коде сервера.

Переменные $arg_ в Nginx используются для извлечения параметров запроса из URL. В случае HTTP-запроса, параметры могут быть переданы в URL после знака вопроса (?), и Nginx автоматически извлекает их в переменные.

Давайте рассмотрим пример:

```
server {
    listen 80;

    location /your_grpc_service {
        grpc_pass grpc://localhost:50051;

        # Опциональные заголовки, которые вы можете настроить
        grpc_set_header ...;
        grpc_read_timeout ...;

        # Извлечение параметра запроса param1
        grpc_set_header param1 $arg_param1;
    }
}
```

Если у вас есть URL вида http://example.com/your_grpc_service?param1=value1, то переменная **$arg_param1** будет содержать значение value1.

Примечание: Если параметры запроса передаются через URL, они автоматически доступны в переменных, начинающихся с $arg_. Если параметры передаются в теле запроса (например, в методе POST), то Nginx предоставляет другие способы работы с ними.

Обратите внимание, что для корректного извлечения параметров из URL, ваш gRPC-сервис должен быть настроен на прием и обработку этих параметров в соответствии с вашими требованиями.

MongoDB, FastAPI, Docker, Kubernetes, gRPC – как микросервис

Идеальность технологического стека зависит от конкретных требований вашего проекта и вашего опыта работы с этими технологиями. Однако, MongoDB, gRPC, FastAPI, Docker и Kubernetes - это комбинация технологий, которая может быть ***весьма эффективной*** для построения микросервисной

архитектуры и их развертывания в кластере с использованием оркестратора типа Kubernetes.

Давайте разберем основные компоненты и их преимущества:

1. MongoDB

Преимущества:
MongoDB является NoSQL базой данных, что может быть полезным в микросервисной архитектуре, где каждый сервис может использовать свой собственный тип хранилища данных. MongoDB хорошо масштабируется, обеспечивает гибкость схемы данных и поддерживает горизонтальное масштабирование.

Использование в микросервисе:
MongoDB может быть хорошим выбором, особенно если ваши микросервисы нуждаются в гибкости в работе с данными.

2. gRPC

Преимущества:
gRPC обеспечивает эффективное и типизированное взаимодействие между микросервисами. Использует Protocol Buffers для сериализации данных, что делает его компактным и быстрым. Поддерживает различные языки программирования и обеспечивает возможность работы с асинхронными вызовами.

Использование в микросервисе:
gRPC подходит для внутреннего взаимодействия между микросервисами, обеспечивая эффективную передачу данных.

3. FastAPI

Преимущества:
FastAPI предоставляет быстрый и современный фреймворк для разработки веб-приложений на Python. Он основан на стандарте OpenAPI и автоматически генерирует интерфейс API, что упрощает его документацию и использование.

Использование в микросервисе:
FastAPI может быть использован для разработки веб-сервисов, предоставляя удобный API для взаимодействия с фронтендом и другими микросервисами.

4. Docker и Kubernetes

Преимущества:

Docker обеспечивает контейнеризацию, что упрощает развертывание и масштабирование микросервисов. Kubernetes является оркестратором контейнеров, предоставляющим средства для автоматизации управления контейнерами, масштабирования и обеспечения надежности системы.

Использование в микросервисе:

Docker и Kubernetes предоставляют инфраструктурную базу для развертывания и управления вашими микросервисами.

Этот стек технологий хорошо сочетается, предоставляя инструменты для эффективной разработки, масштабирования и управления микросервисами. Однако, прежде чем принять окончательное решение, важно учитывать конкретные требования проекта, уровень опыта вашей команды и другие факторы.

Что можно добавить?

Для создания полноценного микросервиса, помимо базового стека технологий, вы можете добавить следующие компоненты и практики:

Централизованный журнал (Centralized Logging):

Интегрируйте централизованный журнал, такой как ELK Stack (Elasticsearch, Logstash, Kibana) или аналогичные, для сбора, анализа и мониторинга логов всех микросервисов. Это поможет в отслеживании проблем и производительности в распределенной системе.

Мониторинг и трассировка (Monitoring & Tracing):

Используйте инструменты мониторинга, такие как Prometheus, Grafana, и трассировки, такие как Jaeger, для отслеживания состояния, производительности и взаимодействий между микросервисами.

Управление конфигурациями (Configuration Management):

Рассмотрите использование инструментов управления конфигурациями, таких как Consul или etcd, для хранения и централизованного управления конфигурациями микросервисов.

API Gateway:

Введение API Gateway (например, Kong, Tyk, или Apigee) может облегчить управление точками входа, контроль доступа и маршрутизацию запросов.

Безопасность:
Обеспечьте безопасность микросервисов с использованием токенов, аутентификации и авторизации. Рассмотрите внедрение инструментов для обнаружения уязвимостей и сканирования безопасности.

Резервирование (Circuit Breaking):
Интегрируйте механизм резервирования (например, Hystrix), чтобы предотвращать распространение сбоев в системе и обеспечивать более устойчивую работу.

Тестирование:
Разработайте надежные юнит-тесты, интеграционные тесты и тесты сценариев использования для обеспечения надежности и качества микросервисов.

Масштабирование:
Рассмотрите возможности горизонтального масштабирования и поддерживайте их с использованием инструментов, таких как Kubernetes, для обеспечения высокой доступности и производительности.

Обновление и развертывание:
Внедрите стратегии развертывания, такие как канареечные релизы, синий/зеленый деплой, чтобы минимизировать воздействие обновлений на работу системы.

Документация:
Предоставьте четкую и подробную документацию для API и внутренней архитектуры микросервисов.

Техническая поддержка и мониторинг:
Обеспечьте механизмы технической поддержки и мониторинга, чтобы оперативно реагировать на проблемы и улучшать производительность.

Добавление этих компонентов и практик обеспечит большую функциональность, управляемость и надежность вашего микросервиса. Однако, учтите, что конечный набор инструментов и компонентов будет зависеть от конкретных требований вашего проекта и контекста использования.

RPS на одном инстансе PostgreSQL

Допустим у нас имеется виртуальный сервер:

- 2 vCPU 3.3Ghz
- 4 Gb RAM

Примерное количестве запросов в секунду (RPS) для данного сервера, это число может варьироваться в диапазоне от примерно 200 до 1000 RPS в зависимости от различных факторов, таких как тип запросов, структура данных, оптимизация кода и базы данных, а также нагрузка на сервер.

Это приблизительные значения, и реальные результаты могут быть выше или ниже в зависимости от конкретных характеристик вашего приложения и обстоятельств его использования. Для получения точных данных вам рекомендуется провести нагрузочное тестирование в реальных условиях вашего приложения.