Паттерны взаимодействия между микросервисами и идемпотентность

Паттерн синхронного взаимодействия

Это паттерн, в котором один микросервис вызывает метод или процедуру другого микросервиса как если бы это был локальный метод. Это обычный синхронный способ взаимодействия между микросервисами через API.

Как это реализуется в монолите

В монолитных приложениях все компоненты обычно работают в одном адресном пространстве, поэтому вызовы методов происходят локально. API в чистом виде тут не применяется.

Зачем этот паттерн в микросервисах

API позволяет микросервисам общаться друг с другом как будто они являются частью одного большого приложения, обеспечивая тем самым высокую степень связанности и сильное сцепление.

Как он реализуется

1. **Микросервис А** нуждается в выполнении какой-то операции, которая реализована в **Микросервисе В**.
2. **Микросервис А** формирует запрос, который представляет собой вызов API в **Микросервисе В**.
3. Запрос отправляется через HTTP или другой протокол.
4. **Микросервис В** получает запрос, выполняет нужную процедуру и возвращает результат.
5. **Микросервис А** получает результат и продолжает свою работу.

Нюансы при использовании паттерна

1. **Связанность**: Этот паттерн создает сильную связанность между микросервисами. (помните? обсуждали что в идеале она должна быть слабой)
2. **Синхронность**: Так как API — синхронный паттерн, его использование может привести к проблемам с производительностью.
3. **Ошибка или задержка**: Если микросервис В недоступен или медленно работает, это негативно скажется на микросервисе А.

4. **Комплексность**: Реализация и поддержка могут быть сложными из-за необходимости контроля версий, таймаутов, обработки ошибок и т.д.

Конечно можно рассматривать не только полностью синхронное API, но и Webhook, WebSockets, Async HTTP и т.д. Но это еще будет усложнять разработку системы.

Паттерн асинхронного взаимодействия

Обмен через брокеры сообщений — это паттерн, в котором микросервисы общаются друг с другом через асинхронные сообщения. Эти сообщения обычно кладутся в очередь, и получатели извлекают их, когда им это удобно.

Как это реализуется в монолите

В монолитах асинхронная обработка может быть реализована через событийные системы, но чаще всего всё происходит в одном процессе, и не требуется дополнительной инфраструктуры для этого.

Зачем этот паттерн в микросервисах

Messaging обеспечивает разделение микросервисов (слабую связанность), позволяя им работать независимо и устойчиво. Это улучшает масштабируемость и распределенность системы.

Как он реализуется

1. **Микросервис A** нуждается в выполнении операции, которая реализована в **Микросервисе B**.
2. **Микросервис A** отправляет сообщение в очередь (например, RabbitMQ или Kafka).
3. **Микросервис B** подписан на эту очередь и извлекает сообщение, когда ему это удобно.
4. **Микросервис B** обрабатывает сообщение и может отправить ответное сообщение, если это необходимо.

Есть различные стили асинхронной коммуникации:

1. **Запрос/Ответ**: Микросервис отправляет запрос и ожидает быстрого ответа. Здесь ключевое слово — "ожидает", но это ожидание не блокирует основную работу сервиса.

2. **Уведомления**: В этом случае микросервис отправляет сообщение, но не ожидает никакого ответа. Это как отправить письмо по почте, не ожидая обратной связи.

3. **Запрос/Асинхронный ответ**: Микросервис отправляет запрос и ожидает ответ в какой-то момент в будущем, но не сразу. Это как заказать что-то в интернете и ожидать уведомление о доставке.

4. **Публикация/Подписка**: Здесь микросервис публикует сообщение, которое может быть прочитано нулем или более другими сервисами. Это как опубликовать новость, которую кто угодно может прочесть.

5. **Публикация/Асинхронный ответ**: Микросервис публикует запрос, и некоторые из подписанных на этот канал сервисов могут отправить обратно ответ. Это как задать вопрос в общем чате и получить ответы от тех, кто знает ответ.

Нюансы при использовании паттерна

1. **Сложность обработки ошибок**: В асинхронных системах обработка ошибок может быть сложнее.
2. **Порядок сообщений**: Если порядок обработки сообщений важен, его нужно явно учитывать.
3. **Доставка**: Не гарантируется, что сообщение будет доставлено или обработано.
4. **Ресурсы**: Требуется дополнительная инфраструктура для управления очередями.
5. **Мониторинг и логирование**: Сложнее отслеживать, что происходит в системе, когда у вас есть асинхронные процессы.

Оба паттерна - синхронный и асинхронный - имеют свои преимущества и недостатки, и выбор между ними зависит от конкретных требований к системе.

Но чаще всего в полноценной микросервисной архитектуре используют брокеры сообщений, так как это снижает связанность и повышает отказоустойчивость системы.

Идемпотентность

А если вы используете очереди сообщений, то у вас теперь может возникать проблема, когда сообщения приходят два раза (потому что вы настроили гарантированную доставку, но её побочный эффект - это доставка более одного раза). Как решать такую проблему? Вы же не будете создавать два одинаковых заказа на одного клиента из-за этой проблемы? Как обработать дубликаты сообщений корректным образом?

Решение

Используйте идемпотентность.

Простой пример

Представьте, что у вас есть сервис, который обрабатывает заказы. Он получает сообщения о новых заказах и обновляет статус заказа в базе данных.

Если сервис получит одно и то же сообщение дважды (что может случиться из-за сетевых ошибок, например), он не должен обрабатывать его как два разных заказа или делать двойные изменения в базе данных.

Как это работает

Чтобы сделать потребителя идемпотентным, вы можете хранить идентификаторы обработанных сообщений в базе данных. Когда потребитель получает сообщение, он сначала проверяет базу данных: если идентификатор сообщения уже есть, сообщение отбрасывается как дубликат. А если идентификатора нет - то создаёт заказ и идёт далее по процессу.

Эти идентификаторы можно хранить в разных местах. Один из вариантов — это создать отдельную таблицу в базе данных, названную, например, `PROCESSED`, и хранить там идентификаторы. Другой вариант — хранить идентификаторы прямо в бизнес-сущностях (например, в записях о заказах), которые потребитель создаёт или обновляет.