GO-12 04: Асинхронное pub-sub взаимодействие (kafka)

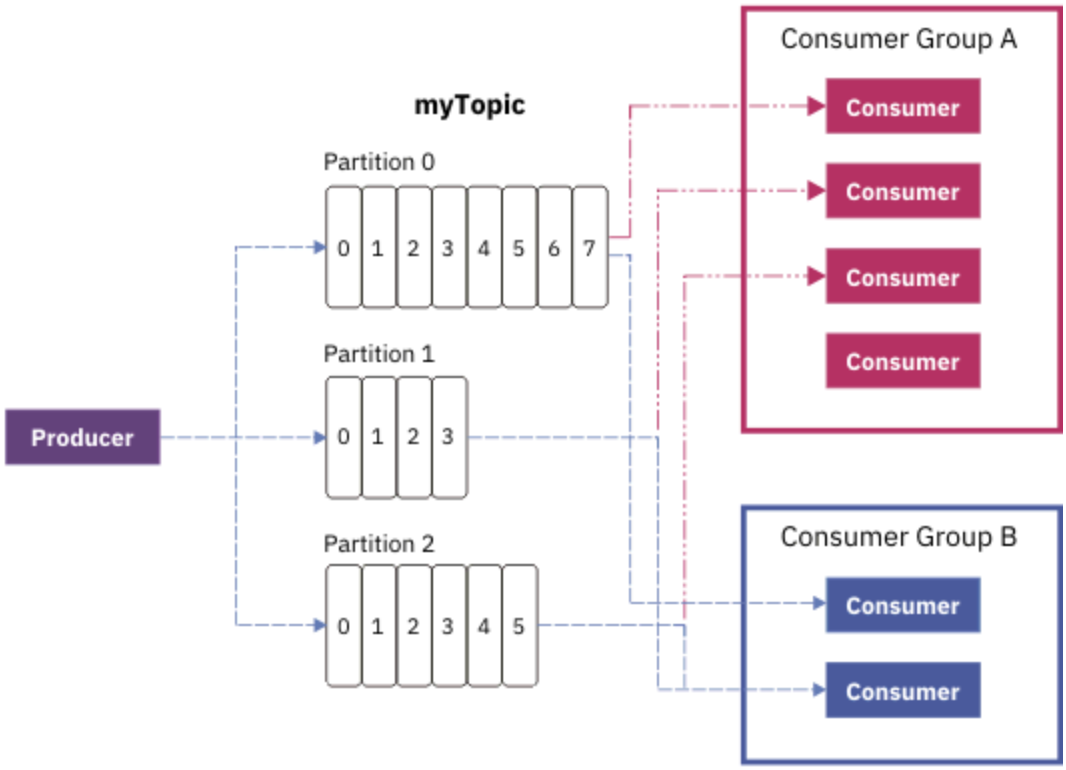
Описание:

В предшествующих заданиях мы изучали синхронное межсервисное взаимодействие - то есть вызывающая сторона ждала, пока ей ответит целевой сервис, и только после этого продолжал выполняться поток, сделавший вызов. Теперь мы рассмотрим концепцию pub-sub через брокер сообщений, в качестве брокера будем использовать Kafka.
Давайте для начала поговорим о том, что такое pub-sub и для чего это нужно.
Представим, что у нас есть сервис, который обслуживает совершение покупок в интернет-магазине, и есть другой сервис, который умеет отправлять уведомления (в нашем случае это будет совершение покупки), предположим, через email. Если покупатель совершает покупку - отрабатывает бизнес-логика первого сервиса, и в какой-то момент ему нужно вызвать сервис уведомлений, чтобы отправить сообщение пользователю.
Если мы это делаем синхронно (REST или gRPC), то сервису совершения покупок придется ждать ответа от сервиса уведомлений. Это создает лишний овертайм на время ожидания или, возможно, сервис окажется недоступен и пользователю вернется ошибка (хотя пользователю, в принципе, не очень важно, отправилось ему письмо или нет - ему важен только факт успешного совершения покупки).
Получается, что у нас появляется потребность в асинхронном вызове сервиса уведомлений, который никак не повлияет на ответ пользователю - например, gRPC имеет асинхронную модель или средствами Go мы можем запустить отдельную горутину, которая независимо от потока запроса пользователя отправит запрос в сервис уведомлений. И иногда это решение и правда имеет место, но какие у него минусы?

- В случае, если нам нельзя терять события, а вызываемый сервис недоступен - наш сервис становится узким местом, которое копит в себе сообщения.
- Не контролируются нагрузки на вызываемый сервис.
- Если есть необходимость в сохранении последовательности, масштабируемости, гарантии доставки и/или персистентном хранении событий на диске - это куча дополнительной логики, которую нужно как-то реализовывать.
- Сильная связность с вызываемой стороной - в случае, если мы, к примеру, хотим добавить к email сервису еще сервис, отправляющий SMS-сообщение - нам нужно будет вносить правки в вызывающий сервис.

От всех этих проблем нас может избавить брокер сообщений - сервис, который принимает сообщения от publisher и доставляет их subscriber. При этом нам нужно просто положить сообщение в нужную очередь и забыть про него, а подписанный на него читатель (если такой есть) его получит и как-то обработает. То есть, теперь оба сервиса зависят, по сути, от брокера - получается некая инверсия зависимости. Существует множество подобных решений со своими особенностями - RabbitMQ, Kafka, Nats, kubeMQ и т.д. Подобный

сервис полностью инкапсулирует в себя организацию работы с сообщениями: гарантия доставки, очередность, партицирование, отказоустойчивость, масштабирование, реплицирование, предоставляет инструменты для организации конкурентной/неконкурентной вычитки и т.д. Кстати, вы можете использовать в качестве брокера даже обычную СУБД со своей структурой и имплементацией логики работы с ней в publisher и subscriber. Но все-таки удобнее использовать специальные инструменты, разработанные для этих целей.
Реализация всех подобных инструментов также очень разнится: Kafka - это под капотом журнал транзакций, в котором подписчик перемещает указатель на следующее сообщение и вычитывает его, а RabbitMQ - это очередь, которая сама проталкивает сообщение подписчикам, а вот Nats не имеет гарантии доставки и сохранения очередности (не путать с Nats Streaming - он имеет), но зато очень быстрый. В общем, простор для выбора огромен, но мы будем рассматривать асинхронное взаимодействие на примере Kafka- поскольку это зарекомендовавшее себя отказоустойчивое и масштабируемое решение, используемое во многих крупных проектах и покрывающее большую часть потребностей.
Рассмотрим подробнее основные сущности Apache Kafka:
Основой Kafka является топик (topic) - это набор журналов сообщений определенного типа. Топики делятся на партиции (partition) - которые и представлют из себя журнал сообщений и между которыми распределяются сообщения топика (пишутся в конец партиции). Партиция - единица параллельности в Kafka. Т.е. в партиции не может быть конкурирующих читателей - чтобы конкурировать, читатели просто должны читать из разных партиций. Еще одной абстракцией для работы с конкурентной вычиткой являются группы - в которые входит один или более потребителей. Каждая группа читает из всех партиций топика, но внутри себя распределяет эти партиции между имеющимися подписчиками - между ними создается конкурентное чтение. Если же разбить двух подписчиков по двум группам - они будут читать неконкурентно - у каждого будет свой указатель в каждой из партиций. Кстати, поскольку Kafka - журнал транзакций - указатель можно по нему передвигать)
Вот пример организации топика и consumer:



Kafka использует бинарный протокол поверх TCP.
Apache Kafka обладает огромными возможностями: Compacted Topics, KTable, Stream API, exactly-once семантика и транзакции, SASL и TLS авторизация и т.д.; Kafka является основой платформы Confluent.
Мы не будем углубляться в Kafka, а рассмотрим взаимодействие с ней средствами Go.
Поднимем Apache Kafka и рассмотрим работу с ней средствами Go
Для поднятия Kafka в контейнере мы будем использовать https://github.com/wurstmeister/kafka-docker:
docker-compose.yml

```
 zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_CREATE_TOPICS: "test:1:1"
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    volumes:
```

```
      - /var/run/docker.sock:/var/run/docker.sock
```

Zookeeper нужен, так как Kafka хранит в нем свою конфигурацию для кластера.
Тут с помощью KAFKA_CREATE_TOPICS мы сразу создаем топик test с одной партицией и одной репликой.
В качестве библиотеки для Go мы будем использовать
https://github.com/segmentio/kafka-go
Проверим, что наш топик успешно создан:

```
conn1, err := kafka.Dial("tcp", "kafka:9092")
	if err != nil {
		panic(err.Error())
	}
	defer conn1.Close()

	partitions, err := conn1.ReadPartitions()
	if err != nil {
		panic(err.Error())
	}

	m := map[string]struct{}{}

	for _, p := range partitions {
		m[p.Topic] = struct{}{}
	}
	for k := range m {
		fmt.Println(k)
	}

Output:
test
```

И мы видим в выводе название нашего топика, отлично - мы можем в него писать:

```
	w := kafka.NewWriter(kafka.WriterConfig{
		Brokers: []string{"kafka:9092"},
		Topic:   "test",
	})

	err := w.WriteMessages(context.Background(),
		kafka.Message{
			Key:   []byte("push"),
```

```
                Value: []byte("Hello World!"),
        },
    )

    if err := w.Close(); err != nil {
        fmt.Println("failed to close writer:", err)
    }
```

И теперь давайте в другом потоке попробуем прочитать из этого топика:

```
    r := kafka.NewReader(kafka.ReaderConfig{
        Brokers: []string{"kafka:9092"},
        Topic:   "test",
    })

    for {
        m, err := r.ReadMessage(context.Background())
        if err != nil {
            break
        }
        fmt.Printf("message at offset %d: %s = %s\n", m.Offset,
string(m.Key), string(m.Value))
    }

    if err := r.Close(); err != nil {
        fmt.Println("failed to close reader:", err)
    }
```

```
Output:
message at offset 0: push = Hello World!
```

Но если мы запустим этот код еще раз, то получим то же самое - получается, что при перезапуске мы скидываем наш offset (указатель с msg логов). Чтобы этого не происходило, можно добавить в Reader значение для поля GroupID - что задаст для нашего консьюмера группу потребителей. Также offset можно задавать вручную методом SetOffset.
Коммит, то есть подтверждение получения происходит автоматически, но можно перевести его в ручной режим, используя в цикле для получения FetchMessage. Тогда придется делать r.CommitMessagesсамим.

Полезные ссылки:

- Apache Kafka
- kafka-go
- Kafka (хабр)

Задание:

1. В вашем проекте module12 создайте из ветки master ветку module12_04.
2. Дополните docker-compose.yml примером выше, чтобы поднять у себя Apache Kafka.
3. Создайте топик notifications (количество реплик и партиций не важно).
4. Дополните клиента из предыдущих заданий логикой, которая при успешном получении сконвертированного файла будет отправлять произвольную информацию об этом событии в топик notifications.
5. Создайте в проекте еще один сервис, который будет подписан на топик notifications, и, получив оттуда сообщение, будет отправлять его на заданный email (задать можно любой).

В качестве SMTP сервера также можете использовать произвольный.

6. В ответе пришлите MP ветки module12_04 в ветку master.