Kafka - вебинар

Category	Developer
CreateAt	December 4, 2021 3:36 PM
Update	December 9, 2021 7:48 PM
Related By	Empty
Relation	
Tags	Golang Concurrency
URL	Empty
In Progress	☑

5 more properties

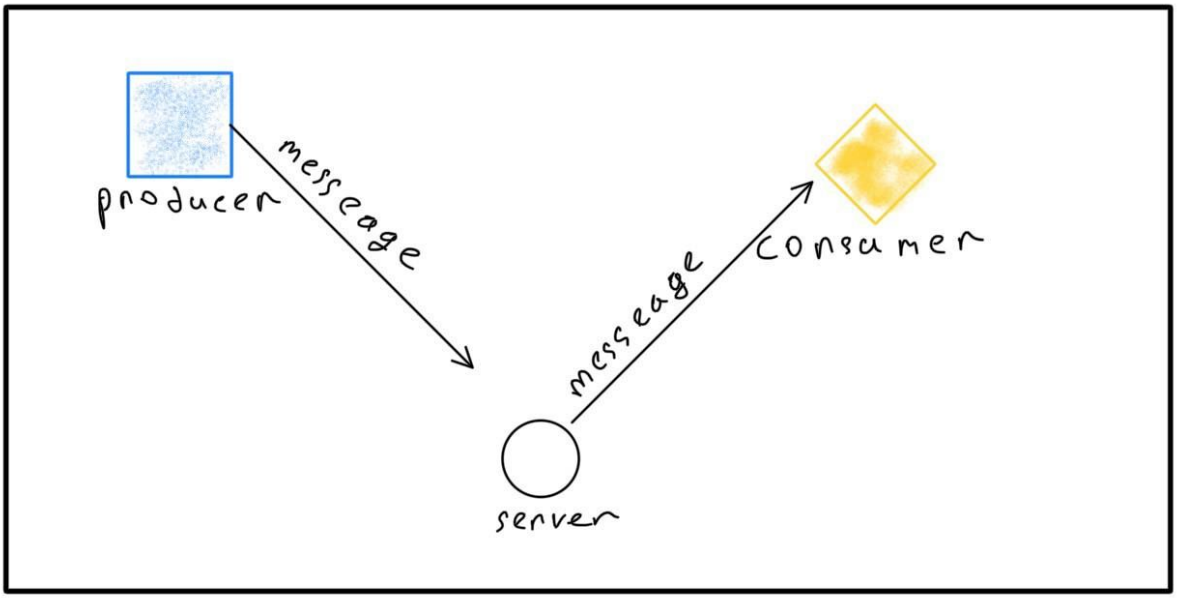
Kafka

Что такое kafka? Это брокер очередей. Брокеры очередей часто используют в микросервисной архитектуре. Системы очередей обычно состоят из трёх базовых компонентов:

1. Сервер

2. Продюссер(сервис отсылающий сообщения в очередь)

3. Консьюмер(сервис принимающий сообщения и исполняющий логику работы)



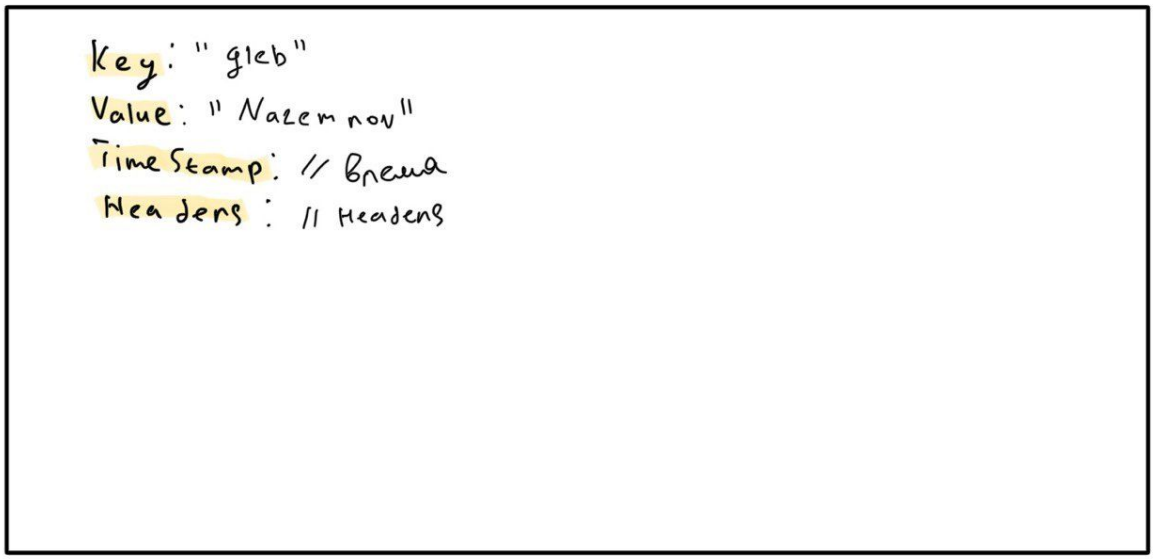
Особенности Kafka перед другими очередями:

- Сообщения в Kafka **не удаляются** брокерами по мере их обработки консьюмерами — данные в Kafka могут храниться днями, неделями, годами.
- Благодаря этому одно и то же сообщение может быть обработано **сколько угодно** разными консьюмерами и в разных контекстах.

К кафке в одну очередь, можно подключить сколько угодно много сервисов, и таким образом сообщение будет прочитано всего одним инстансом. Но так же в одну и туже очередь(топик) можно подключить и сервисы с другой логикой, отправляя сообщения всегда в один топик.
Kafka не удаляет данные после обработки консьюмерами, эти данные могут обрабатываться заново, как бы отматывая время назад сколько угодно раз. Это оказывается невероятно полезно для восстановления после сбоев и, например, верификации кода новых консьюмеров. В случае с RabbitMQ пришлось бы записывать все данные заново, при этом, скорее всего, в отдельную очередь, чтобы не сломать уже имеющихся клиентов.

Каждое сообщение (event или message) в Kafka состоит из ключа, значения, таймстампа и опционального набора метаданных (так называемых хедеров).

Ниже на картинке вы видите типичное кафка сообщение.



Еще из плюсов кафка, она может быть развернута в кластере и изи коробке поддерживает парцицирование. Эту тему в нашем вебинаре мы пропустим.

Переходим к работе с kafka:

В кафка существует понятие (**Consumer Groups**)

Читатели объединяются в группы, что так и называется — consumer group. При добавлении нового читателя или падении текущего, группа перебалансирутся. Это занимает какое-то время, поэтому лучший способ чтения — подключить читателя и не переподключать его без необходимости.

Переходим к разработке на go:

Мы будем использовать в коде sarama.

Для начала создадим продюссера:

```
type SyncProducer struct { sp sarama.SyncProducer } func
NewSyncProducer(addrs []string, config *sarama.Config)
(*SyncProducer, error) { sp, err := sarama.NewSyncProducer(addrs,
config) if err != nil { return nil, err } return &SyncProducer{sp:
sp}, nil } func (p *SyncProducer) Put(ctx context.Context, topic
string, val interface{}, in map[string]string) (partition int32,
offset int64, err error) { var b []byte b, err = json.Marshal(val)
if err != nil { return -1, -1, err } msg :=
&sarama.ProducerMessage{ Topic: topic, Value:
sarama.ByteEncoder(b), } partition, offset, err =
p.sp.SendMessage(msg) if err != nil { return -1, -1, err } return
partition, offset, nil } // Close close kafka connect func (p
*SyncProducer) Close() error { return p.sp.Close() }
```

Объявляем консьюмеров:

```
type ConsumerGroup struct { groupID string brokers []string } type
MessageHandler interface { Handle(ctx context.Context, msg
*sarama.ConsumerMessage) error } func NewConsumerGroup(brokers
[]string, groupID string) *ConsumerGroup { return &ConsumerGroup{
brokers: brokers, groupID: groupID, } } func (cg *ConsumerGroup)
ConsumeTopic(ctx context.Context, topics []string, handler
MessageHandler, opts ...ConsumeOption) (*TopicConsumer, error) {
consumeOpts := &ConsumeOptions{ initialOffset: sarama.OffsetNewest,
} for _, opt := range opts { opt(consumeOpts) } cfg :=
sarama.NewConfig() cfg.Consumer.Offsets.Initial =
consumeOpts.initialOffset cfg.Consumer.Return.Errors =
consumeOpts.returnErrors c, err :=
sarama.NewConsumerGroup(cg.brokers, cg.groupID, cfg) if err != nil
{ return nil, err } return &TopicConsumer{ msgHandler: handler,
topics: topics, client: c, groupID: cg.groupID, }, nil } type
TopicConsumer struct { msgHandler MessageHandler topics []string
client sarama.ConsumerGroup cancel context.CancelFunc groupID
string } func (tc *TopicConsumer) Run(opts ...RunOption) { runOpts
:= &RunOptions{ commitStrategy: func(cgs
sarama.ConsumerGroupSession, cm *sarama.ConsumerMessage, e error) {
cgs.MarkMessage(cm, "") }, } for _, opt := range opts {
opt(runOpts) } var ctx context.Context ctx, tc.cancel =
context.WithCancel(context.Background()) go func() { for { if
ctx.Err() == context.Canceled { break } if err :=
tc.client.Consume(ctx, tc.topics, &handler{ msgHandler:
tc.msgHandler, retries: runOpts.retries, timeout: runOpts.timeout,
setup: runOpts.setup, cleanup: runOpts.cleanup, commitStrategy:
runOpts.commitStrategy, groupID: tc.groupID, }); err != nil { //
залогать ошибку } } }() } func (tc *TopicConsumer) Stop() { if
tc.cancel != nil { tc.cancel() tc.cancel = nil } } func (tc
*TopicConsumer) Error() <-chan error { return tc.client.Errors() }
// Close closes topic's consumer group func (tc *TopicConsumer)
Close() error { tc.Stop() return tc.client.Close() } type handler
struct { msgHandler MessageHandler retries int timeout
time.Duration setup func(sarama.ConsumerGroupSession) error cleanup
func(sarama.ConsumerGroupSession) error commitStrategy
func(sarama.ConsumerGroupSession, *sarama.ConsumerMessage, error)
groupID string } func (h *handler) Setup(cgSessoin
sarama.ConsumerGroupSession) error { if h.setup != nil { return
h.setup(cgSessoin) } return nil } func (h *handler)
Cleanup(cgSession sarama.ConsumerGroupSession) error { if h.cleanup
!= nil { return h.cleanup(cgSession) } return nil } func (h
*handler) ConsumeClaim(cgSession sarama.ConsumerGroupSession,
cgClaim sarama.ConsumerGroupClaim) error { names :=
strings.Split(h.groupID, ":") service := "" branch := "" if
```

```
len(names) == 2 { service = names[0] branch =
strings.Replace(names[1], "/", "-", -1) } for { select { case msg,
ok := <-cgClaim.Messages(): if !ok { return nil } ctx :=
context.Background() var span opentracing.Span msgCtx, err :=
tracing.Extract(ctx, msg.Headers) if err != nil { span, msgCtx =
opentracing.StartSpanFromContext(ctx, "Kafka Read Sync") } else {
span = opentracing.SpanFromContext(msgCtx) } if span != nil {
span.SetTag("span.kind", "kafka_read_sync")
span.SetTag("kafka.group_id", h.groupID)
span.LogFields(log.String("topic", msg.Topic))
span.LogKV("partition", msg.Partition) span.LogKV("offset",
msg.Offset) } mCtx, mesh := tracing.ExtractMesh(msgCtx,
msg.Headers) if mesh != nil { if val, ok := mesh[service]; ok &&
!strings.EqualFold(val, branch) { cgSession.MarkMessage(msg, "")
continue } } err = retry(h.retries, h.timeout, func() error {
return h.msgHandler.Handle(mCtx, msg) })
h.commitStrategy(cgSession, msg, err) if err != nil { if span !=
nil { span.SetTag("error", true) span.LogFields(log.String("error",
err.Error())) span.Finish() } return err } if span != nil {
span.Finish() } case <-cgSession.Context().Done(): return nil } } }
func retry(retries int, timeout time.Duration, f func() error) (err
error) { defer func() { if rvr := recover(); rvr != nil { err =
errors.Errorf("panic recovered: %+v, stack: %s", rvr,
debug.Stack()) //залогать ошибку } }() var ( retried int ) for {
err = f() if err == nil { break } retried++ if retried > retries {
break } time.Sleep(timeout) } return err }
```

И на последок кастомизируем библиотеку:

Аналоги sarama:

- https://github.com/segmentio/kafka-go

Дополнительные материалы к прочтению:

- https://habr.com/ru/company/avito/blog/465315/
- https://medium.com/swlh/apache-kafka-with-golang-227f9f2eb818