Pulsar Go-SDK 使用说明

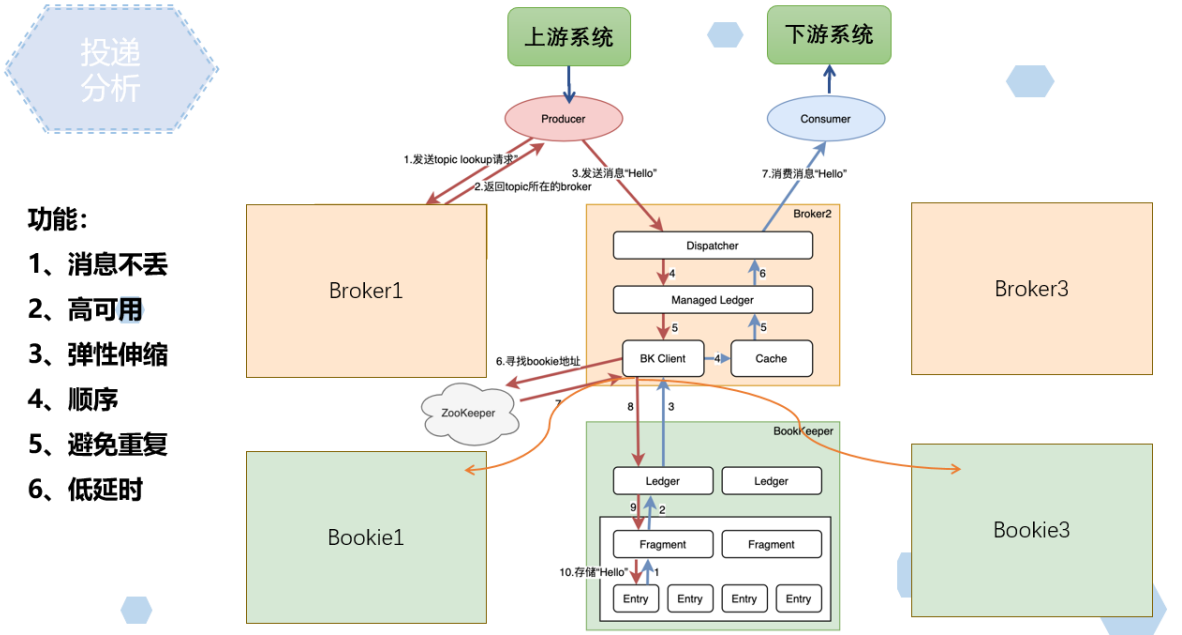
官方文档

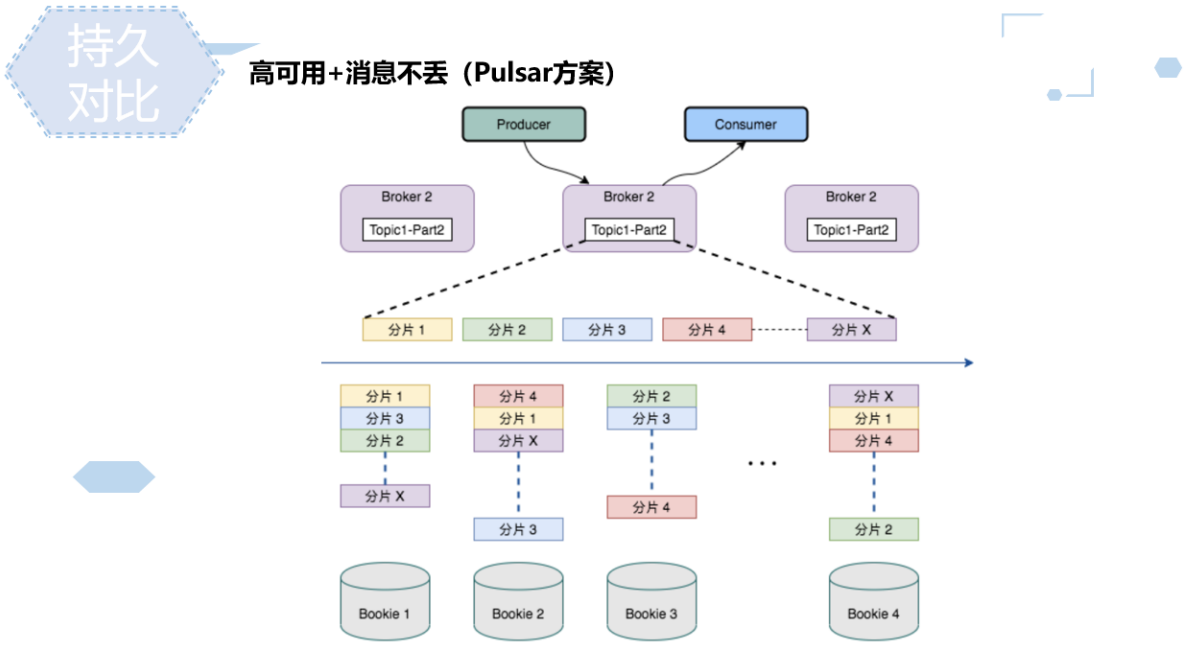
- Pulsar Go client
- Pulsar Admin Interface

Pulsar 简介

系统架构

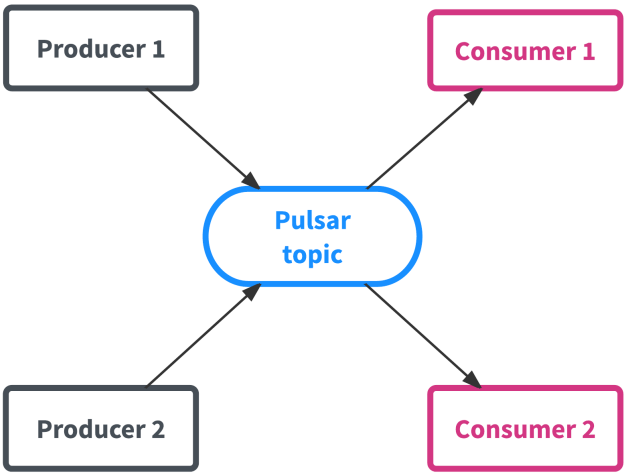
Pulsar 集群由 zookeeper, bookie(bookeeper), broker 三部分组成。三者都支持集群化部署。在满足最小集群配置的情况下集群中任一组件崩溃都不影响整个系统的运行。实际测试 kill 掉任一个组件不会造成消息丢失，生产消费即使受影响也能很快恢复。





基本概念

Topic

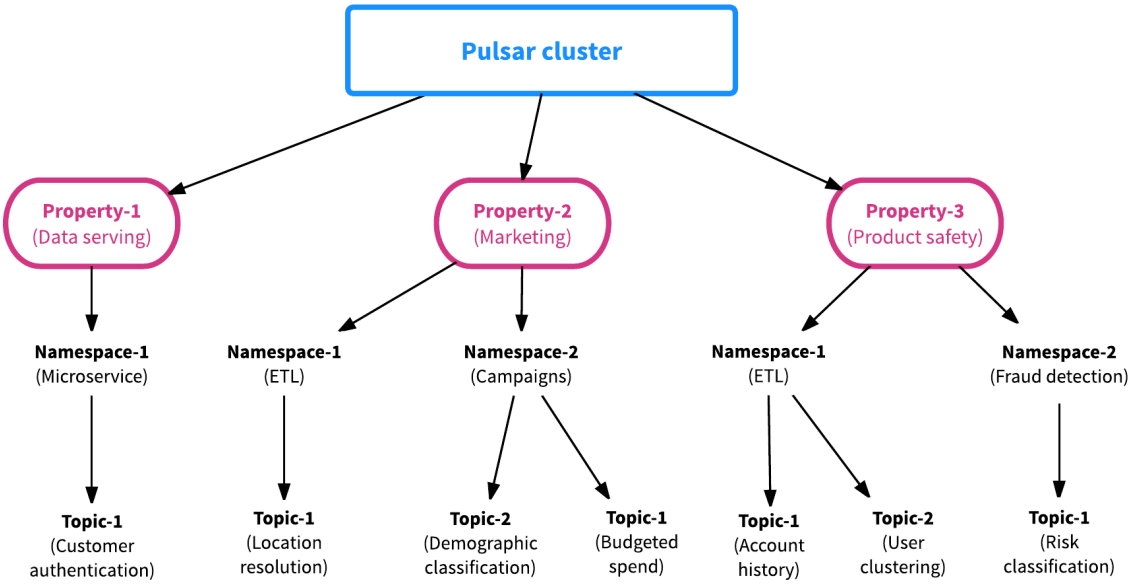


Topic是Pulsar的核心概念，表示一个“channel”，Producer可以写入数据，Consumer从中消费数据（Kafka、RocketMQ都是这样）。

Topic名称的URL类似如下的结构：

```
{persistent|non-persistent}://tenant/namespace/topic
// 默认情况只填写 topic，pulsar 默认会填充全名
topic => persistent://public/default/topic
```

- persistent|non-persistent表示数据是否持久化（Pulsar支持消息持久化和非持久化两种模式）
- Tenant为租户
- Namespace一般聚合一系列相关的Topic，一个租户下可以有多个Namespace



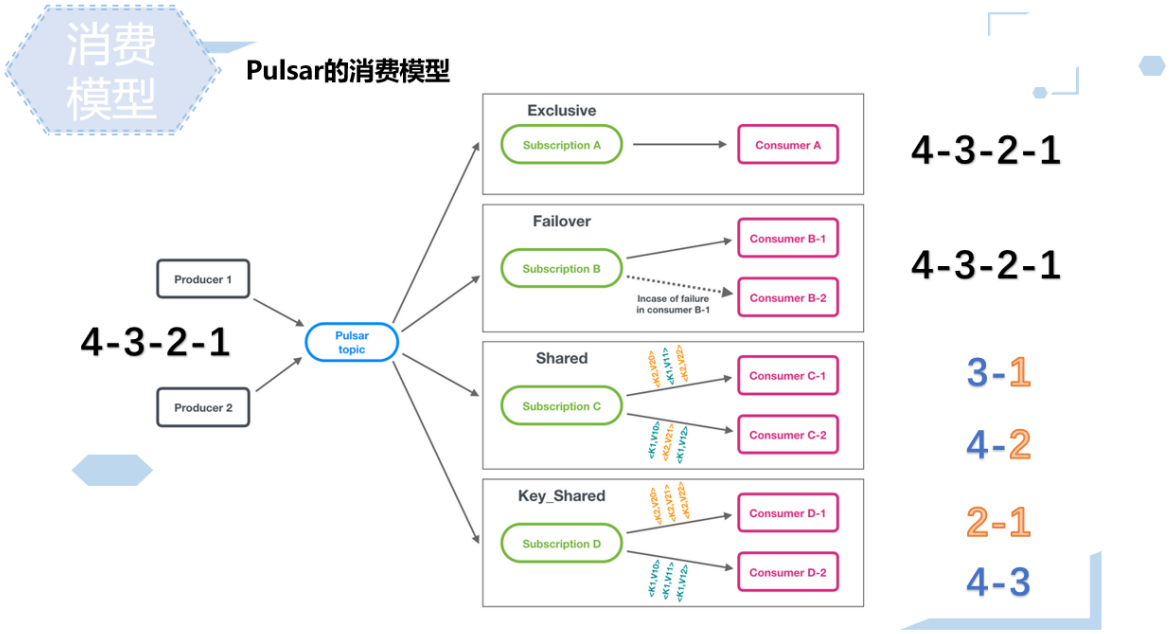
上图中Property即为租户，每个租户下可以有多个Namespace，每个Namespace下有多个Topic。

Namespace是Pulsar中的操作单元，包括Topic是配置在Namespace级别的，包括多地域复制，消息过期策略等都是配置在Namespace上的。

订阅模型

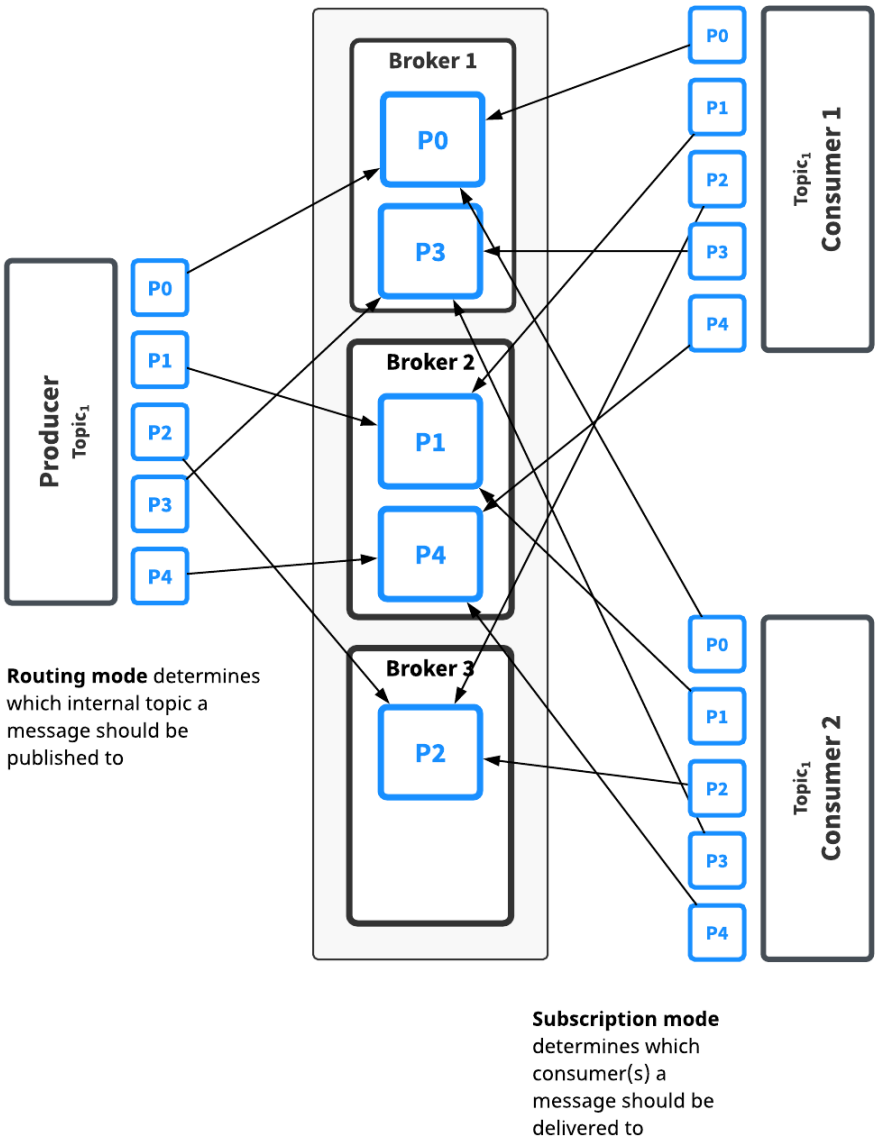
Pulsar提供了灵活的消息模型，支持四种订阅类型：

- Exclusive subscription：排他的，只能有一个Consumer，接收一个Topic所有的消息。
- Failover subscription：Failover模式，同一时刻只有一个有效的Consumer，其余的Consumer作为备用节点，在Master Consumer不可用后进行替代（看起来适用于数据量小，且解决单点故障的场景）
- Shared subscription：共享的，可以同时存在多个Consumer，每个Consumer处理Topic中一部消息（Shared模型是不保证消息顺序的，Consumer数量可以超过分区的数量）
- Key Shared：类似与共享模式，多个 Consumer 会轮询方式处理同一个Topic 的消息。不同的是 Key Shared 模式会保证持有相同 Key 的消息将被分发到同一个 Consumer 上。



分区

为了解决吞吐等问题，Pulsar和Kafka一样，采用了分区（Partition）的机制。Consumer 消息的分发也是与 Partition 一一对应的，SDK 内部维护了 consumer 与 partition 的对应关系。默认情况下 topic 只拥有一个 partition。



Pulsar提供了一些策略来处理消息到Partition的路由（MessageRouter）：

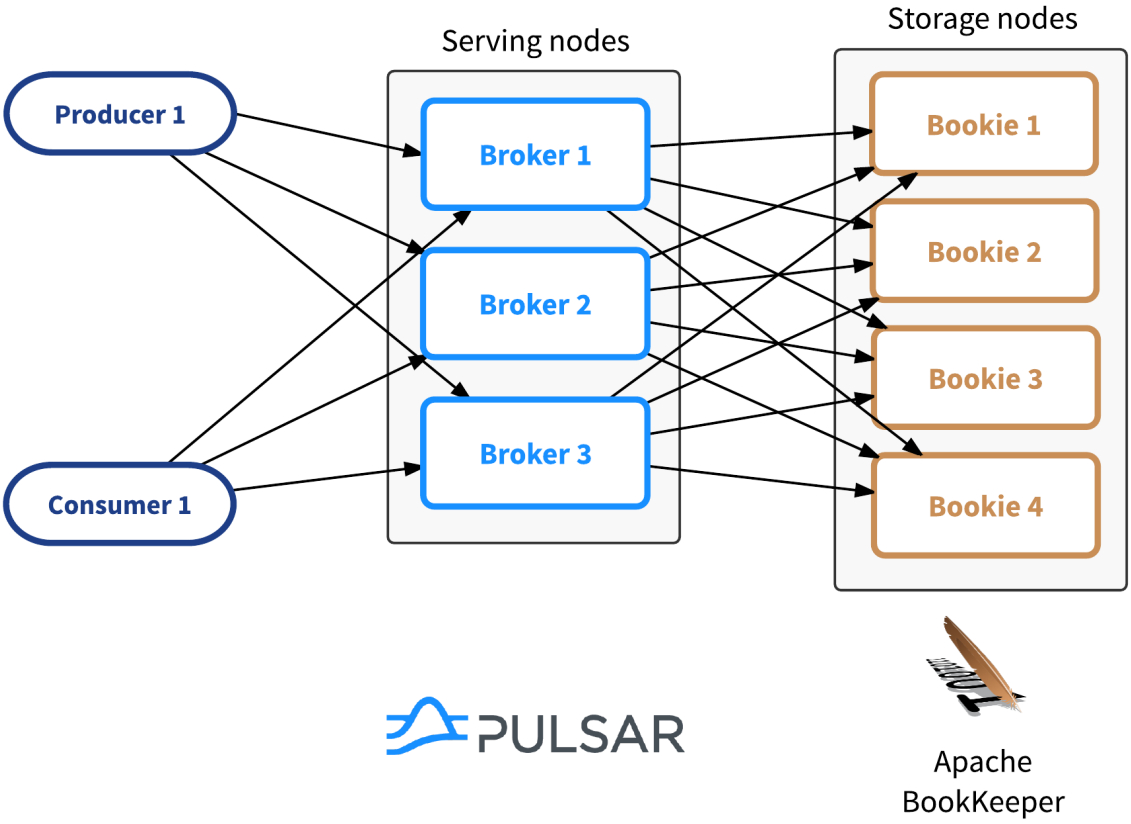
- Single partitioning：Producer随机选择一个Partition并将所有消息写入到这个分区
- Round robin partitioning：采用Round robin的方式，轮训所有分区进行消息写入
- Hash partitioning：这种模式每条消息有一个Key，Producer根据消息的Key的哈希值进行分区的选择（Key相同的消息可以保证顺序）。
- Custom partitioning：用户自定义路由策略

不同于别的MQ系统，Pulsar允许Consumer的数量超过分区的数量（对于RocketMQ，超过分区数的Consumer会分配不到分区而“空跑”）。

在Shared subscription的订阅模式下，Consumer数量可以大于分区的数量，每个Consumer处理每个Partition中的一部分消息，不保证消息的顺序。

持久化

Pulsar通过BookKeeper来存储消息，保证消息不会丢失（BookKeeper：A scalable，fault-tolerant，and low-latency storage service optimized for real-time workloads）。



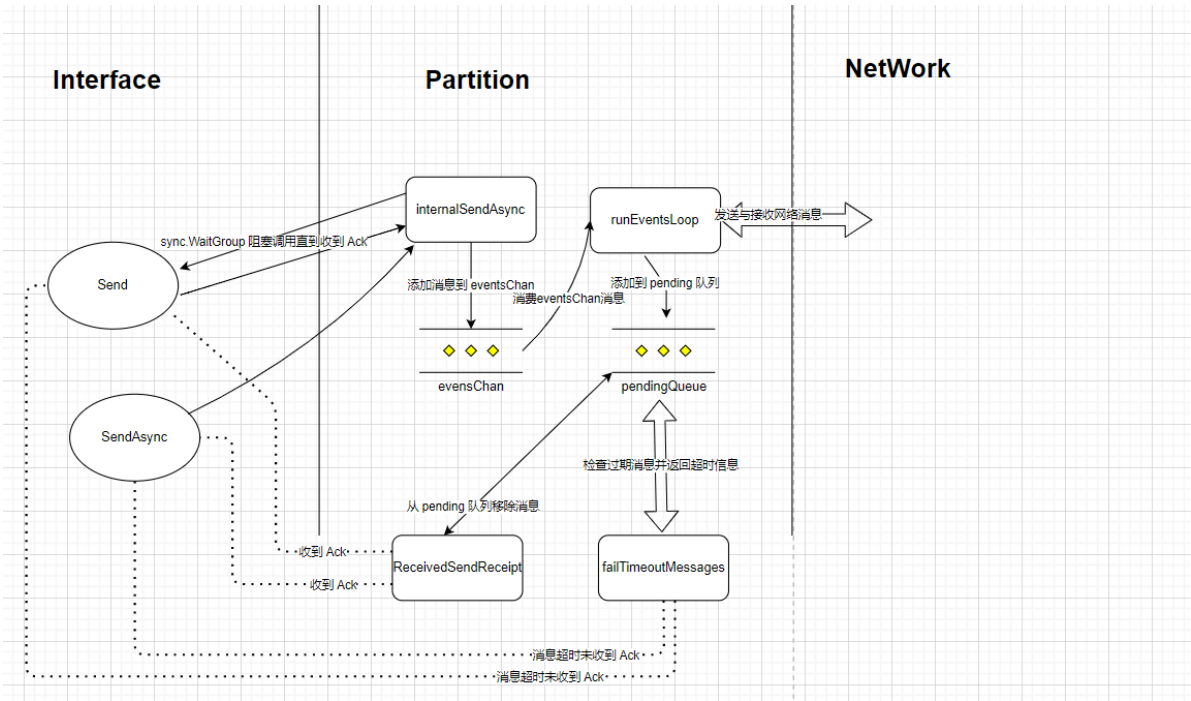
Pulsar采用"存储和服务分离"的两层架构（这是Pulsar区别于其他MQ系统最重要的一点，也是所谓的"下一代消息系统"的核心）：

- Broker：提供发布和订阅的服务（Pulsar的组件）
- Bookie：提供存储能力（BookKeeper的存储组件）

优势是Broker成为了stateless的组件，可以水平扩容（RocketMQ的Broker是包含存储的，是有状态的，Broker的扩容更像是"拆分"）。高可靠，一致性等通过Bookie去保证。

SDK 原理

SDK 发送消息



```
// 同步发送消息，内部调用 sync.WaitGroup 实现阻塞的异步发送
func (p *partitionProducer) Send(ctx context.Context, msg *ProducerMessage)
(MessageID, error) {
    wg := sync.WaitGroup{}
    wg.Add(1)
    ...
    p.internalSendAsync(ctx, msg, func(ID MessageID, message *ProducerMessage, e
error) {
        err = e
        msgID = ID
        wg.Done()
    }, true)

    wg.Wait()
    return msgID, err
}

// 异步发送消息，内部调用 internalSendAsync 实现异步发送
func (p *partitionProducer) SendAsync(ctx context.Context, msg *ProducerMessage,
    callback func(MessageID, *ProducerMessage, error)) {
    p.internalSendAsync(ctx, msg, callback, false)
}

// 具体的发送函数，internalSendAsync 负责把消息写到 eventsChan 此后由 runEventsLoop 消
费此 channel 发送消息并把消息加入到 pendingQueue, failTimeoutMessages 会周期性检测
pendingQueue 头部的消息是否超时，并处理超时消息
func (p *partitionProducer) internalSendAsync(ctx context.Context, msg
*ProducerMessage,
    callback func(MessageID, *ProducerMessage, error), flushImmediately bool) {
    ...
    sr := &sendRequest{
        ...
    }
    ...
    p.eventsChan <- sr
}

// 接收消息会调用到这个函数
func (p *partitionProducer) ReceivedSendReceipt(response *pb.CommandSendReceipt)
{
    pi, ok := p.pendingQueue.Peek().(*pendingItem)
    ...
    for idx, i := range pi.sendRequests {
        sr := i.(*sendRequest)
        if sr.callback != nil || len(p.options.Interceptors) > 0 {
            ...
        }
    }
    ...
}

func (p *partitionProducer) failTimeoutMessages() {
    diff := func(sentAt time.Time) time.Duration {
        return p.options.SendTimeout - time.Since(sentAt)
    }

    t := time.NewTimer(p.options.SendTimeout)
```

```
    defer t.Stop()

    for range t.C {
        ...
        item := p.pendingQueue.Peek()
        ...
    }
}
```

SDK 接收消息

```
// SDK consumer 接收到消息会调用这个函数将消息发送到 queueCh 上
func (pc *partitionConsumer) MessageReceived(response *pb.CommandMessage,
headersAndPayload internal.Buffer) error {
    pbMsgID := response.GetMessageId()
    ...
    // send messages to the dispatcher
    pc.queueCh <- messages
    return nil
}

// 如下: dispatcher 会接收 queueCh 上的消息与其他事件, 并最终把消息发送到 `messageCh` 上,
这里 `messageCh` 就是 consumer 消费的 channel
func (pc *partitionConsumer) dispatcher() {
    ...
    for {
        ...
        } else {
            // we are ready for more messages
            queueCh = pc.queueCh
        }

        select {
        ...
        case msgs, ok := <-queueCh:
            if !ok {
                return
            }
            // we only read messages here after the consumer has processed all
messages
            // in the previous batch
            messages = msgs

        // if the messageCh is nil or the messageCh is full this will not be
selected
        case messageCh <- nextMessage:
            // allow this message to be garbage collected
            messages[0] = nil
            messages = messages[1:]
            ...
        }
    }
}
```

Go SDK BUG

handleSendError nil 解引用崩溃

go mod 自动下载的 v0.6.0 版本 SDK，internal -> connection.go，handleSendError(cmdError *pb.CommandError) 入参有可能为 nil，会导致如下问题的崩溃错误。
建议使用 master 分支版本(使用此仓库的vendor 版本)
本地包地址 `https://git.iflytek.com/xbli10/pulsar-vendor.git`

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x99f942]

goroutine 101 [running]:
github.com/apache/pulsar-client-go/pulsar/internal.
(*connection).handleSendError(0xc0000bd760, 0x0)
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:755
+0x182
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x99f942]

goroutine 101 [running]:
github.com/apache/pulsar-client-go/pulsar/internal.
(*connection).handleSendError(0xc0000bd760, 0x0)
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:755
+0x182
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:539
+0x1f2
github.com/apache/pulsar-client-go/pulsar/internal.
(*connection).run(0xc0000bd760)
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:408
+0x3c5
github.com/apache/pulsar-client-go/pulsar/internal.
(*connection).start.func1(0xc0000bd760)
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:227
+0x85
created by github.com/apache/pulsar-client-go/pulsar/internal.
(*connection).start
        /mnt/d/Users/xbli10/wdir/pulsar-benchmark-
go/vendor/github.com/apache/pulsar-client-go/pulsar/internal/connection.go:223
+0x56
```

Go SDK 示例与参数说明

创建客户端

使用示例

```
import (
    "log"
    "runtime"

    "github.com/apache/pulsar/pulsar-client-go/pulsar"
```

```
)

func main() {
    logger := logrus.New()
    logger.Out, _ = os.OpenFile("/dev/null", os.O_WRONLY, 0666)

    client, err := pulsar.NewClient(pulsar.ClientOptions{
        URL: "pulsar://10.1.87.69:6650,10.1.87.52:6650,10.1.87.54:6650",
        MaxConnectionsPerBroker: 100,
        Logger:                  log.NewLoggerWithLogrus(logger),
    })

    if err != nil {
        log.Fatal(err)
    }

    defer client.Close()
}
```

pulsar.ClientOptions 常用参数说明

参数名	说明	必须	默认值
`URL`	Pulsar URI	是	
`ConnectionTimeout`	建立 TCP 连接的超时时间	否	5s
`OperationTimeout`	设置操作超时时间(Producer-create, subscribe, unsubscribe)	否	30s
`MaxConnectionsPerBroker`	连接池中与某个特定的 broker 的 TCP 连接数	否	1
`Logger`	配置客户端的 logger. 默认使用 log.NewLoggerWithLogrus(logrus.StandardLogger())	否	

生产者

同步方式发送消息，单个消息的耗时达到 100ms，所以吞吐较低

使用示例

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    client, err := pulsar.NewClient(pulsar.ClientOptions{
        URL: "pulsar://localhost:6650",
    })
```

```
    if err != nil {
        log.Fatal(err)
    }

    defer client.Close()

    producer, err := client.CreateProducer(pulsar.ProducerOptions{
        Topic: "topic-1",
    })
    if err != nil {
        log.Fatal(err)
    }

    defer producer.Close()

    ctx := context.Background()

    for i := 0; i < 10; i++ {
        if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
            Payload: []byte(fmt.Sprintf("hello-%d", i)),
        }); err != nil {
            log.Fatal(err)
        } else {
            log.Println("Published message: ", msgId)
        }
    }
}
```

pulsar.ProducerOptions 常用参数说明

建议设置如下参数：

- pulsar.producer.batchingEnabled=false
- pulsar.producer.blockIfQueueFull=true
- 发送超时设置

参数名	说明	必须	默认值
Topic	Topic	是	
Name	生产者名称，必须在同一 Topic 下集群唯一。如果不指定，系统会自动生成一个名称	否	
Properties	Properties attach a set of application defined properties to the producer. This properties will be visible in the topic stats	否	
SendTimeout	消息发送超时时间。在超时时间内没收到 ACK 发送认为失败。-1 标识没有超时时间	否	30s
DisableBlockIfQueueFull	发送队列满时的阻塞还是返回错误	否	false
MaxPendingMessages	等待 ACK 的等待队列大小	否	1000
CompressionType	压缩算法: LZ4, ZLIB, ZSTD，默认不压缩	否	
CompressionLevel	压缩等级: Default, Faster, Better	否	
DisableBatching	是否禁用批量发送。类似于 TCP Nagel 算法批处理启用期间消息将被堆积直到发送条件满足，然后一次发送出去。	否	true
BatchingMaxPublishDelay	启用批处理时，消息最大等待时长	否	10ms
BatchingMaxMessages	启用批处理时，等待队列最大长度	否	1000
BatchingMaxSize	启用批处理时，消息最大堆积大小	否	128KB

消费者(consume)

消费消息必须先进行订阅。produce 生产消息到没有订阅者的 topic，生产的消息将被丢失。
类似于 Kafka 消费者按组管理，同一个订阅里的消费者属于一个消费者组。每个组内消费者支持：共享，独占，failover三种消费方式

使用示例

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    client, err := pulsar.NewClient(pulsar.ClientOptions{URL:
"pulsar://localhost:6650"})
    if err != nil {
        log.Fatal(err)
    }
```

```
    defer client.Close()

    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
        Topic:            "topic-1",
        SubscriptionName: "my-sub",
        Type:             pulsar.Shared,
    })
    if err != nil {
        log.Fatal(err)
    }
    defer consumer.Close()

    for i := 0; i < 10; i++ {
        msg, err := consumer.Receive(context.Background())
        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("Received message msgId: %#v -- content: '%s'\n",
            msg.ID(), string(msg.Payload()))

        consumer.Ack(msg)
    }

    if err := consumer.Unsubscribe(); err != nil {
        log.Fatal(err)
    }
}
```

pulsar.ConsumerOptions 常用参数说明

参数名	说明	必须	默认值
Topic	订阅的 Topic	Topic,Topics,TopicsPattern 选一	
Topics	订阅的一组 Topic	Topic,Topics,TopicsPattern 选一	
TopicsPattern	订阅的 Topic 正则表达式	Topic,Topics,TopicsPattern 选一	
AutoDiscoveryPeriod	TopicsPattern 模式下刷新 Topic 列表的时间	否	60s
SubscriptionName	订阅组名称	是	
Properties	Attach a set of application defined properties to the consumer. This properties will be visible in the topic stats	否	
SubscriptionType	订阅类型	否	Exclusive
SubscriptionInitialPosition	订阅组消息游标	否	SubscriptionPositionLatest
MessageChannel	接受消息的 chan	否	
ReceiverQueueSize	接收队列大小	否	1000
NackRedeliveryDelay	Nack 消息重新分发延时	否	60s
Name	消费者名	否	
Interceptors	消息拦截器。将在特定场合(消费前，ACK，NACK)被调用	否	

消费者(Reader)

使用示例

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    client, err := pulsar.NewClient(pulsar.ClientOptions{URL:
"pulsar://localhost:6650"})
    if err != nil {
        log.Fatal(err)
```

```
    }

    defer client.Close()

    reader, err := client.CreateReader(pulsar.ReaderOptions{
        Topic:          "topic-1",
        StartMessageID: pulsar.EarliestMessageID(),
    })
    if err != nil {
        log.Fatal(err)
    }
    defer reader.Close()

    for reader.HasNext() {
        msg, err := reader.Next(context.Background())
        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("Received message msgId: %#v -- content: '%s'\n",
            msg.ID(), string(msg.Payload()))
    }
}
```

Schema

对于围绕消息队列如 pulsar 搭建的应用来说，类型安全非常的重要。
Producer 和 consumer 需要某种机制，以在主题级别上协调类型，从而避免出现各种潜在问题。 例如，序列化和反序列化问题。
应用程序通常采用以下方法来保证消息传递中的类型安全。 这两种方法都被Pulsar支持，你可以在topic的基础上，自由选择采用哪一种，或者混用。

使用示例(参考 schema_test.go)

```
type testJSON struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
}

var (
    exampleSchemaDef = "
{\"type\":\"record\",\"name\":\"Example\",\"namespace\":\"test\"," +
        "\"fields\":[{\"name\":\"ID\",\"type\":\"int\"},
{\"name\":\"Name\",\"type\":\"string\"}]}"
    protoSchemaDef = "
{\"type\":\"record\",\"name\":\"Example\",\"namespace\":\"test\"," +
        "\"fields\":[{\"name\":\"num\",\"type\":\"int\"},
{\"name\":\"msf\",\"type\":\"string\"}]}"
)

// json schema
func jsonSchema() {
    client := connect()
    defer client.Close()

    producer1, err := client.CreateProducer(pulsar.ProducerOptions{
```

```
        Topic: "json",
        Schema: pulsar.NewJSONSchema(exampleSchemaDef,
            map[string]string{"pulsar": "hello"},
        ),
    })
    panicOnError(err)
    defer producer1.Close()

    _, err = producer1.Send(context.Background(), &pulsar.ProducerMessage{
        Value: &testJSON{
            ID:   100,
            Name: "pulsar",
        },
    })
    panicOnError(err)

    //create consumer
    var s testJSON

    consumerJS := pulsar.NewJSONSchema(exampleSchemaDef, nil)
    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
        Topic:            "json",
        SubscriptionName: "mysub",
        Schema:           consumerJS,
    })
    defer consumer.Close()
    msg, err := consumer.Receive(context.Background())
    panicOnError(err)
    err = msg.GetSchemaValue(&s)
    panicOnError(err)
    fmt.Println(s.ID, 100)
    fmt.Println(s.Name, "pulsar")
}

// proto schema
func protoSchema() {
    client := connect()
    defer client.Close()

    // create producer
    psProducer := pulsar.NewProtoSchema(protoSchemaDef, nil)
    producer, err := client.CreateProducer(pulsar.ProducerOptions{
        Topic:  "proto",
        Schema: psProducer,
    })
    panicOnError(err)

    if _, err := producer.Send(context.Background(), &pulsar.ProducerMessage{
        Value: &pb.Test{
            Num: 100,
            Msf: "pulsar",
        },
    }); err != nil {
        log.Fatal(err)
    }

    //create consumer
    unobj := pb.Test{}
```

```
    psConsumer := pulsar.NewProtoSchema(protoSchemaDef, nil)
    consumer, err := client.Subscribe(pulsar.ConsumerOptions{
        Topic:            "proto",
        SubscriptionName: "mysub",
        Schema:           psConsumer,
    })
    panicOnError(err)

    msg, err := consumer.Receive(context.Background())
    panicOnError(err)
    err = msg.GetSchemaValue(&unobj)
    panicOnError(err)
    fmt.Println(unobj.Num, int32(100))
    fmt.Println(unobj.Msf, "pulsar")
    defer consumer.Close()
}
```