Golang语言常用方法和接口

陈念

# GO语言基础知识

## 关于package

一个文件夹下的文件只能有同一个package名称

一个package里的函数只有函数名**首字母为大写**的时候才能被其他package里的文件调用

可以将一个package理解为一个c++的类，只有首字母大写的才是public属性，对外可见，所以在单元测试的时候，如果是对package内部的函数进行单元测试的代码最好和代码在同一个目录下，否则如果仅仅是为了单元测试而将内部函数声明为public（首字母大写）是不合适的。

golang的package的特点：

1、go的package不局限于一个文件，可以由多个文件组成。

　　组成一个package的多个文件，编译后实际上和一个文件类似，组成包的不同文件相互之间可以直接引用变量和函数，不论是否导出；

　　因此，组成包的多个文件中不能有相同的全局变量和函数

2、go不要求package的名称和所在目录名相同，但是你最好保持相同，否则容易引起歧义。因为引入包的时候，go会使用子目录名作为包的路径，而你在代码中真正使用时，却要使用你package的名称。

3、每个子目录中只能存在一个package，否则编译时会报错。

## 关于指针

Go 的原生数据类型可以分为基本类型和高级类型，基本类型主要包含 string， bool， int 及 float 系列，高级类型包含 struct，array/slice，map，chan, func 。

相比 Java，Python，Javascript 等引用类型的语言，Golang 拥有类似C语言的指针这个相对古老的特性。但不同于 C 语言，Golang 的指针是单独的类型，而不是 C 语言中的 int 类型，而且也不能对指针做整数运算。从这一点看，Golang 的指针基本就是一种引用。

那么 Golang 为什么需要指针？这种指针又能有什么独特的用途呢？

在学习引用类型语言的时候，总是要先搞清楚，当给一个函数/方法传参的时候，传进去的是值还是引用。实际上，在大部分引用型语言里，参数为基本类型时，传进去的大都是值，也就是另外复制了一份参数到当前的函数调用栈。参数为高级类型时，传进去的基本都是引用。这个主要是因为虚拟机的内存管理导致的。

内存管理中的内存区域一般包括 heap 和 stack， stack 主要用来存储当前调用栈用到的简单类型数据：string，boolean，int，float 等。这些类型的内存占用小，容易回收，基本上它们的值和指针占用的空间差不多，因此可以直接复制，GC也比较容易做针对性的优化。 复杂的高级类型占用的内存往往相对较大，存储在 heap 中，GC 回收频率相对较低，代价也较大，因此传引用/指针可以避免进行成本较高的复制操作，并且节省内存，提高程序运行效率。

因此，在下列情况可以考虑使用指针：1，需要改变参数的值；2，避免复制操作；3，节省内存；

而在 Golang 中，具体到高级类型 struct，slice，map，也各有不同。实际上，只有 struct 的使用有点复杂，slice，map，chan 都可以直接使用，不用考虑是值还是指针。

# Json包

包含json包

|  |
| --- |
| **import "encoding/json"** |

## 支持的数据类型

Json支持的数据类型与Golang语言的关系如下：

|  |
| --- |
| **bool, for JSON booleans**  **float64, for JSON numbers**  **string, for JSON strings**  **[]interface{}, for JSON arrays**  **map[string]interface{}, for JSON objects**  **nil for JSON null** |

**注意** ： 只会将可外部访问的字段(即结构体的**变量名称使用大写字母开头**)序列化到json里面去

## 编码

Json编码有两个接口：encode和Marshal

### Encode用法

**函数原型：**

**func NewEncoder(w io.Writer) \*Encoder**

**用法：**

|  |
| --- |
| file**, \_ := os.Create("json.txt")**  enc **:= json.NewEncoder(**file**)**  err **:=** enc**.**Encode**(&v)** |

数据结构v会以json格式写入json.txt文件。

### Marshal用法

**函数原型**：

**func Marshal(v interface{}) ([]byte, error)**

**用法：**

|  |
| --- |
| Msg := json.**Marshal**(msg\_batch)  log.Println("msg:", **string(msg)**) |

**注意：**Marshal返回的并不是可见的string，如果打印的话需要string强转一下，如上所示

## 解码

golang的解码有两种方式：decode和Unmarshal

decode用法

当JSON返回的格式未知，或者不想定义过多的struct时，可以使用这种方法。

函数原型：

func NewDecoder(r io.Reader) \*Decoder

用法1：

|  |
| --- |
| d := json.**NewDecoder**(bytes.NewReader(data))  d.**UseNumber**()  var i interface{}  if err := d.**Decode**(&i); err != nil {  return err  }  ji := i.(map[string]interface{})  if ji["dataSet"] == "ptd\_log" {  for \_, data := range ji["data"].([]interface{}) {  xdata := data.(map[string]interface{})  if xdata["type"] == "heartbeat" {  continue  //ignore heartbeat  }  return nil  } |

Json解码的时候会将数字解码成float64类型，使用d.UseNumber()表示对于数字不要解析成float64的类型，而是暂时将其作为json.Number类型，这种类型实质是一个字符串，如果需要将一个number解析为int64，需要使用如下的方法：

|  |
| --- |
| data,\_ := (val.(json.Number)).Int64() |

用法2：

|  |
| --- |
| fp, \_ os.Open("json.txt")  dec := json.**NewDecoder**(fp)  for {  var V v  err := dec.**Decode**(&v)  if err != nil {  break  }  //use v  } |

### Unmarshal用法

**明确知道返回JSON的数据格式，用这种方法**

函数原型：func Unmarshal(data []byte, v interface{})

用法：

|  |
| --- |
| **// content saves json repsonse in []byte formated**  **err = json.Unmarshal(content, &json\_rslt)**  **if err != nil {**  **panic(err)**  **}**  **fmt.Println(json\_rslt)** |

# NSQ

包含package

|  |
| --- |
| import "github.com/nsqio/go-nsq" |

NSQ是由四个重要组件构成：

nsqd：一个负责接收、排队、转发消息到客户端的守护进程

nsqlookupd：管理拓扑信息并提供最终一致性的发现服务的守护进程

nsqadmin：一套Web用户界面，可实时查看集群的统计数据和执行各种各样的管理任务

utilities：常见基础功能、数据流处理工具，如nsq\_stat、nsq\_tail、nsq\_to\_file、nsq\_to\_http、nsq\_to\_nsq、to\_nsq

创建生产者代码示例

|  |
| --- |
| producer, err := nsq.**NewProducer**(nsqConf.Remote.Addr, nsq.NewConfig())  if err != nil {  print("new producer error: ", err.Error(), "\n")  return  }  producer.**Publish**("test\_topic", msg) |

创建消费者代码示例

|  |
| --- |
| //对消息的处理函数  func handlerMsg(message \*nsq.Message) error {  defer message.Finish()  return msg\_to\_stor.Process\_json(message.Body)  }  //创建消费者，绑定处理函数  for {  consumer, err := nsq.**NewConsumer**(“test\_topic”, “t\_channel”, nsq.NewConfig())  if err != nil {  fmt.Println("new consumer error: ", err, "\n")  time.Sleep(1 \* time.Second) //wait 1s  continue  }  consumer.**AddHandler**(hdlr)  err = consumer.**ConnectToNSQD**(addr)  if err != nil {  fmt.Println("connect nsqd error: ", err, "\n")  time.Sleep(1 \* time.Second) //wait 1s  continue  }  \_ = <-consumer.**StopChan**  } |

更多NSQ的介绍见[参考-NSQ](#_NSQ)

# time包

**包含package**

|  |
| --- |
| Import “time” |

## 数据类型

**Time类型**

时间类型,包含了秒和纳秒以及Location

**Duration类型**

**type Duration int64** 持续时间。定义了以下持续时间类型，多用于时间的加减需要传入Duration做为参数的时候.

const (

Nanosecond Duration = 1

Microsecond = 1000 \* Nanosecond

Millisecond = 1000 \* Microsecond

Second = 1000 \* Millisecond

Minute = 60 \* Second

Hour = 60 \* Minute

)

## 常用方法

### 获取当前时间绝对秒数

**原型：**

**Now**() Time

**Unix**() int64

**用法：**

time\_sec = time.Now().Unix()

### 将绝对秒数转换为Time类型

**原型：**

**Unix**(sec int64, nsec int64) Time

**用法：**

ts := time.Unix(time\_sec, 0)

### 比较两个Time类型的时间差

**原型：**

**Sub**(u Time) Duration

**用法：**

d := time1.Sub(time2)

### 获取一个Time类型的时间点

**函数原型：**

**ParseDuration**(s string) (Duration, error)

将duration字符串("ns", "us" (or "碌s"), "ms", "s", "m", "h".)转换为Duration类型.就是纳秒

**Add**(d Duration) Time

**用法：**

**1、设置一个1小时后的time类型**

ts := time.Now()

t, \_ := time.ParseDuration("1h")

ts2 := ts.Add(t)

**2、设置一个1分钟前的time类型**

ts := time.Now()

t, \_ := time.ParseDuration("-1m")

ts2 := ts.Add(t)

**3、设置一个10秒钟前的time类型**

ts := time.Now()

t, \_ := time.ParseDuration("-10s")

ts2 := ts.Add(t)

### 比较两个time类型的先后

**原型：**

**After**(u Time) bool

**Before**(u Time) bool

**用法：**

time1.Before(time2)

time1.After(time2)

### Sleep 5秒

**原型：**

func **Sleep**(d Duration)

**用法**

d := time.Duration(5) \* time.Second

time.Sleep(d)

### 当前格式化时间

fmt.Println(time.Now().Format("2006-01-02 15:04:05"))  // 这是个奇葩,必须是这个时间点, 据说是go诞生之日

### 时间戳转时间字符串

str\_time := time.Unix(1389058332, 0).Format("2006-01-02 15:04:05")

### 时间字符串转Time格式

the\_time, err := time.Parse("2006-01-02 15:04:05", "2014-01-08 09:04:41")

time包的更多接口和用法见[参考time](#_time)

# strings包

## 包含strings包

import "strings"

## 常用接口

### 字符串str是否以某个字符串prestr开头

strings.**HasPrefix**(str, prestr)

### 一个字符串str中是否包含某个子串substr

strings.**Contains**(str, substr)

### 两个字符串是否相等

strings.**EqualFold**(str1, str2)

### 替换字符串里内容

func Replace(s, old, new string, n int) string

# encoding/hex包

import " encoding/hex "

## 用途

用于二进制和二进制字符串的转换，比如MD5

## 常用接口

### 二进制转字符串

func EncodeToString(src []byte) string

### 字符串转二进制

f\_md5,\_ = hex.DecodeString(md5.(string))

# encoding/binary

import " encoding/ binary "

encoding/binary用了reflect会比较慢

## 用途

Byte数组和数字转换

## 常用接口

### Uint32转byte数组

binary.BigEndian.PutUint32(ip, nn)

### byte数组转uint32

nn = binary.BigEndian.Uint32(ip)

# net包

import "net"

## 用途

IP地址从二进制到字符串的相互转换可以使用net包的函数接口

## IP类型定义：

type IP []byte

//IP表示一个简单的IP地址，它是一个byte类型的slice，能够接受4字节（IPV4）或者16字节（IPV6）输入。注意，IP地址是IPv4地址还是IPv6地址是语义上的特性，而不取决于切片的长度：16字节的切片也可以是IPv4地址。

## 常用接口：

### 字符串转换为IP类型

func ParseIP(s string) IP  //ParseIP将s解析为IP地址，并返回该地址。如果s不是合法的IP地址表示，则ParseIP会返回nil。字符串可以是小数点分隔的IPv4格式（如"74.125.19.99"）或IPv6格式（如"2001:4860:0:2001::68"）格式。

### 判断两个IP类型是否相等

func (ip IP) Equal(x IP) bool   //判断两个ip地址是否相等，其中一个ipv4地址以及相同具有ipv6格式的相同的地址认为是相等的，返回true

### IP类型转换为字符串

func (ip IP) String() string      //获取ip地址的字符串表示，如果ip是IPv4地址，返回值的格式为点分隔的，如"74.125.19.99"；否则表示为IPv6格式，如"2001:4860:0:2001::68"。

### IP类型转换为16字节

func (ip IP) To16() IP     //将一个IP地址转换为16字节表示。如果ip不是一个IP地址（长度错误），To16会返回nil。To16可对ip地址进行转换，包括IPV4和IPV6，而To4只能对IPV4地址进行转换，这就是To16和To4的区别。

### IP类型转换为4字节

func (ip IP) To4() IP       //将一个IPV4地址转换为4字节表示，如果ip不是一个ipv4地址，则返回nil

更多net包的详细用法参见[参考net包](#_Net包)

# 单元测试

## testing包

### 包含package

|  |
| --- |
| import “testing” |

### 用途：

**用于单元测试**

### 用法：

1. 文件名必须是\_test.go结尾的，这样在执行go test的时候才会执行到相应的代码
2. 所有的测试用例函数必须是Test开头 测试用例会按照源代码中写的顺序依次执行
3. 单元测试函数原型

|  |
| --- |
| func Test\_check\_time\_out(t \*testing.T){  } |

在做单元测试时，我们经常需要判断执行结果是否符合预期，可以使用assert包

Testing包的更多接口和用法见[参考testing](#_Testing)

### 注意事项：

1. **单个单元测试函数执行时间不能太长，超过规定时间单元测试会强制退出，根据之前的经验是不能超过9分钟**
2. **如果执行make test出现以下错误：**

make: `test' is up to date.

**需要在Makefile文件添加一行：**

**.PHONY: all test clean**

1. **如果需要统计单元测试代码覆盖率，在Makefile里添加如下语句（如果test代码和功能代码在一个文件夹可以去掉红色部分，不在一个文件夹根据实际情况指定路径和包名）**

|  |
| --- |
| test:  mkdir -p testdata/ && \  go test -coverpkg="../Stor" -cover -coverprofile=testdata/cover.data -v . && \  go tool cover -html=testdata/cover.data -o testdata/cover.html |

## assert包

**包含package**

|  |
| --- |
| import "github.com/stvp/assert" |

**函数原型:**

func **True**(t \*testing.T, got interface{}, messages ...interface{})

func **Nil**(t \*testing.T, got interface{}, messages ...interface{})

func **NotNil**(t \*testing.T, got interface{}, messages ...interface{})

**用法举例**

|  |  |
| --- | --- |
| func Test\_check\_time\_out(t \*testing.T){  msg = create\_msg\_body()  err = Process\_json(msg)  assert.**Nil**(t,err)  assert.**True**(t,31 == totalCapture, "totalCapture should 31, but", totalCapture)  assert.**True**(t,29 == totalDataQueueItem,"totalDataQueueItem should 29, but", totalDataQueueItem)  } |  |

其中第三个参数字符串表示assert失败时输出的语句

更多接口和用法见[参考-assert](#_assert)

# 参考

## NSQ

NSQ是一个基于Go语言的分布式实时消息平台，它基于MIT开源协议发布，代码托管在GitHub。NSQ可用于大规模系统中的实时消息服务，并且每天能够处理数亿级别的消息，其设计目标是为在分布式环境下运行的去中心化服务提供一个强大的基础架构。NSQ具有分布式、去中心化的拓扑结构，该结构具有无单点故障、故障容错、高可用性以及能够保证消息的可靠传递的特征。NSQ非常容易配置和部署，且具有最大的灵活性，支持众多消息协议。

NSQ是由四个重要组件构成：

nsqd：一个负责接收、排队、转发消息到客户端的守护进程

nsqlookupd：管理拓扑信息并提供最终一致性的发现服务的守护进程

nsqadmin：一套Web用户界面，可实时查看集群的统计数据和执行各种各样的管理任务

utilities：常见基础功能、数据流处理工具，如nsq\_stat、nsq\_tail、nsq\_to\_file、nsq\_to\_http、nsq\_to\_nsq、to\_nsq

topic:一个可供订阅的话题。  
channel:属于topic的下一级，一个topic可以有多个channel。

举个例子：  
topic：比做一个广播，如交通广播。打开收音机，你可以换很多频率，如果换到91.6MHZ，你就会听到交通广播，（我们这里交通广播是91.6）。相当于你订阅了“交通广播”这个topic。  
一个topic下有多个channel，可以看作是广播里会有很多节目，比如路况信息、美食、用车知识等等。每一个节目可以比作一个channel。

如果你一直订阅了交通广播，那你就会收到这个下面所有channel的信息，订阅了topic，就会收到topic下所有消息。  
当然，你可能只关注用车知识这个消息，那可以在这个节目的播出时间听这个广播就可以了，等节目播完，就可以不收听这个广播。相当于：nsq里可以只订阅某一个channel的信息。这样的话，一个topic下无关的channel就不会发过来。  
如果一个channel有多个订阅者，NSQ会使用负载均衡的策略，给其它一个订阅者发消息。

核心概念

在讨论NSQ如何在实践中使用前，先理解NSQ队列的架构原理是非常值得的。它的设计很简单，可以通过几个核心概念来理解。

**Topic** ——一个topic就是程序发布消息的一个逻辑键，当程序第一次发布消息时就会创建topic。

**Channels** ——channel组与消费者相关，是消费者之间的负载均衡，channel在某种意义上来说是一个“队列”。每当一个发布者发送一条消息到一个topic，消息会被复制到所有消费者连接的channel上，消费者通过这个特殊的channel读取消息，实际上，在消费者第一次订阅时就会创建channel。

Channel会将消息进行排列，如果没有消费者读取消息，消息首先会在内存中排队，当量太大时就会被保存到磁盘中。

**Message** s——消息构成了我们数据流的中坚力量，消费者可以选择结束消息，表明它们正在被正常处理，或者重新将他们排队待到后面再进行处理。每个消息包含传递尝试的次数，当消息传递超过一定的阀值次数时，我们应该放弃这些消息，或者作为额外消息进行处理。

NSQ在操作期间同样运行着两个程序：

**Nsqd** ——nsqd守护进程是NSQ的核心部分，它是一个单独的监听某个端口进来的消息的二进制程序。每个nsqd节点都独立运行，不共享任何状态。当一个节点启动时，它向一组nsqlookupd节点进行注册操作，并将保存在此节点上的topic和channel进行广播。

客户端可以发布消息到nsqd守护进程上，或者从nsqd守护进程上读取消息。通常，消息发布者会向一个单一的local nsqd发布消息，消费者从连接了的一组nsqd节点的topic上远程读取消息。如果你不关心动态添加节点功能，你可以直接运行standalone模式。

**Nsqlookupd** ——nsqlookupd服务器像consul或etcd那样工作，只是它被设计得没有协调和强一致性能力。每个nsqlookupd都作为nsqd节点注册信息的短暂数据存储区。消费者连接这些节点去检测需要从哪个nsqd节点上读取消息。

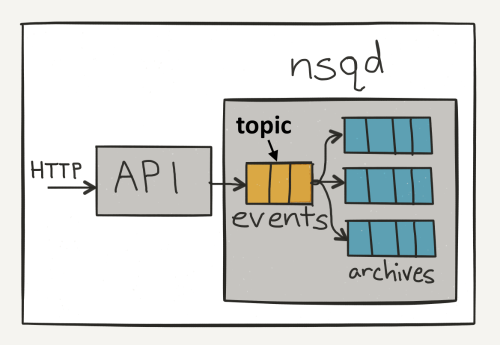
消息的生命周期

让我们观察一个关于nsq如何在实际中工作的更为详细的例子。

NSQ推荐通过他们相应的nsqd实例使用协同定位发布者，这意味着即使面对网络分区，消息也会被保存在本地，直到它们被一个消费者读取。更重要的是，发布者不必去发现其他的nsqd节点，他们总是可以向本地实例发布消息。

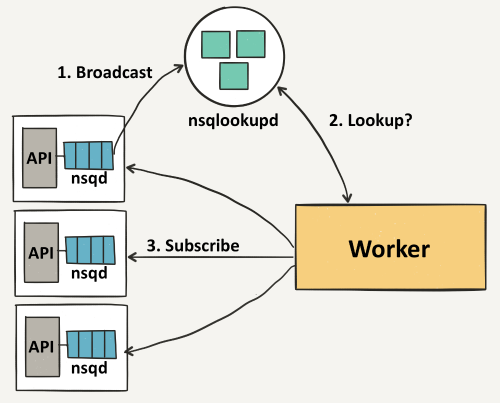
首先，一个发布者向它的本地nsqd发送消息，要做到这点，首先要先打开一个连接，然后发送一个包含topic和消息主体的发布命令，在这种情况下，我们将消息发布到事件topic上以分散到我们不同的worker中。

事件topic会复制这些消息并且在每一个连接topic的channel上进行排队，在我们的案例中，有三个channel，它们其中之一作为档案channel。消费者会获取这些消息并且上传到S3。



每个channel的消息都会进行排队，直到一个worker把他们消费，如果此队列超出了内存限制，消息将会被写入到磁盘中。

Nsqd节点首先会向nsqlookup广播他们的位置信息，一旦它们注册成功，worker将会从nsqlookup服务器节点上发现所有包含事件topic的nsqd节点。



然后每个worker向每个nsqd主机进行订阅操作，用于表明worker已经准备好接受消息了。这里我们不需要一个完整的连通图，但我们必须要保证每个单独的nsqd实例拥有足够的消费者去消费它们的消息，否则channel会被队列堆着。

从客户端库代码中抽取一部分，这里是一个关于如何处理我们的消息的一段代码：

msg := <- ch

**if** msg.Attempts > MAX\_DELIVERY\_ATTEMPTS {

// we discard the message if it's more than the max delivery attempts

// normally this is handled by the library

msg.Finish()

**return**

}

err, \_ := request(msg)

**if** err != nil {

log.Errorf("error making request %v\n", err)

msg.Requeue(BACKOFF\_TIME) // an error occurred, requeue... (╯°□°)╯︵ ┻━┻

**return**

}

msg.Finish() // everything worked... (？■\_■)

如果因为某些原因第三方发生故障了，我们可以处理这些故障，在这个代码片中，我们有三种处理逻辑：

1、如果超过了某个尝试次数阀值，我们就将消息丢弃。

2、如果消息已经被处理成功了，我们就结束消息。

3、如果发生了错误，我们将需要传递的消息重新进行排队。

正如你所看到的，NSQ队列的行为既简单又明确。

在我们的案例中，我们在丢弃消息之前将容忍MAX\_DELIVERY\_ATTEMPTS \* BACKOFF\_TIME分钟的故障。

在Segment系统中，我们统计消息尝试的次数、消息丢弃数、消息重新排队数等等，然后结束某些消息以保证我们有一个好的服务质量。如果消息丢弃数超过了我们设置的阀值，我们将在任何时候对服务发出警报。

在实践中

在生产环境中，我们几乎在我们所有的实例中运行nsqd守护程序，发布者之间协同定位。NSQ在实际生产中运行良好有几个原因：

**简单的协议** ——如果你的队列已经有了一个很好的客户端库，这个不是一个很大的问题，但如果你现在的客户端库存在bug或者过时了，一个简单的协议就能体现出优势了。

NSQ有一个快速的二进制协议，通过短短的几天工作量就可以很简单地实现这些协议，我们还自己创建了我们的纯JS驱动（当时只存在coffeescript驱动），这个纯JS驱动运行的很稳定可靠。

**运行简单** ——NSQ没有复杂的水印设置或JVM级别的配置，相反，你可以配置保存到内存中的消息的数量和消息最大值，如果队列被消息填满了，消息会被保存到磁盘上。

**分布式** ——因为NSQ没有在守护程序之间共享信息，所以它从一开始就是为了分布式操作而生。个别的机器可以随便宕机随便启动而不会影响到系统的其余部分，消息发布者可以在本地发布，即使面对网络分区。

这种“分布式优先”的设计理念意味着NSQ基本上可以永远不断地扩展，需要更高的吞吐量？那就添加更多的nsqd吧。

唯一的共享状态就是保存在lookup节点上，甚至它们不需要全局视图，配置某些nsqd注册到某些lookup节点上这是很简单的配置，唯一关键的地方就是消费者可以通过lookup节点获取所有完整的节点集。

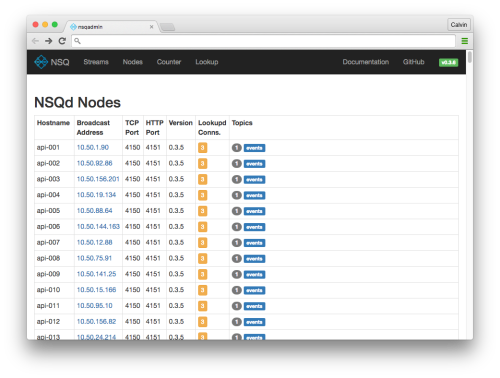
清晰的故障事件——NSQ在组件内建立了一套明确关于可能导致故障的的故障权衡机制，这对消息传递和恢复都有意义。

我是最少意外原则的坚定信仰者，尤其是当它涉及到分布式系统时。系统发生故障，我们接收它，但我们不可能会指望系统以意外的形式发生故障，你最终会忽略这些故障案例，因为你甚至都不打算考虑它们为什么会发生。

虽然它们可能不像Kafka系统那样提供严格的保证级别，但NSQ简单的操作使故障情况非常明显。

**UNIX-y工具** ——NSQ是一个很好的通用型工具，所以NSQ附带了很多实用的程序，这些程序是多用途和可组合的。

除了TCP协议，NSQ提供一个简单的CURL的HTTP接口用于维护操作，它从CLI附带了二进制文件管道，用tail跟踪队列的尾部，从一个队列使用管道到另外一个队列，还有HTTP发布订阅。



甚至还有一个用于监控和暂停队列的管理面板，包括一个动态的计数器在上面。

丢失了什么？

正如我所提到的，简单并不是没有折衷：

**没有复制** ——不像其他的队列组件，NSQ并没有提供任何形式的复制和集群，也正是这点让它能够如此简单地运行，但它确实对于一些高保证性高可靠性的消息发布没有足够的保证。

我们可以通过降低文件同步的时间来部分避免，只需通过一个标志配置，通过EBS支持我们的队列。但是这样仍然存在一个消息被发布后马上死亡，丢失了有效的写入的情况。

**基本消息路由** ——在NSQ中，topic和channel几乎是你所有能获得到的东西，没有关于路由和基于key的亲和力的观念。我们很乐意为各种用例提供支持，无论是根据条件去筛选消息，还是根据条件路由到某些节点上。取而代之的是，我们最终建立了路由worker，它们处于队列之间，扮演一个聪明的直通滤波器。

**没有严格的顺序** ——虽然Kafka由一个有序的日志构成，但NSQ不是。消息可以在任何时间以任何顺序进入队列。在我们使用的案例中，这通常没有关系，因为所有的数据都被加上了时间戳，但它并不适合需要严格顺序的情况。

**无数据重复删除功能** ——Aphyr已经在他的文章中广泛探讨了基于超时系统的危险性。NSQ同样也调入了这个陷阱，它使用了心跳检测机制去测试消费者是否存活还是死亡。我们之前已经写过关于很多原因会导致我们的worker无法完成心跳检测，所以在worker中必须有一个单独的步骤确保幂等性。

简单的工作原理

正如你所看到的，后面看到的所有好处的基本核心就是简单性，NSQ是一个简单的队列，这意味着它很容易进行故障推理和很容易发现bug。消费者可以自行处理故障事件而不会影响系统剩下的其余部分。

事实上，简单性是我们决定使用NSQ的首要因素，这方便与我们的许多其他软件一起维护，通过引入队列使我们得到了堪称完美的表现，通过队列甚至让我们增加了几个数量级的吞吐量。

今天，我们面临一个更加复杂的未来，我们越来越多的worker需要一套严格可靠性和顺序性保障，这已经超过了NSQ提供的简单功能。

我们计划在其他基础设施中用Kafka替换NSQ，在生产上从JVM中运行可以获取更多的好处。关于Kafka我们有一个明确的权衡，我们自己必须肩负起更多负责的运营。另一方面，它拥有一个可复制的、有序的日志可以提供给我们更好的服务。

但对于其他适合NSQ的worker，它为我们服务的相当好，我们期待着继续巩固它的坚实的基础。

## time包

|  |
| --- |
| Time 时间类型,包含了秒和纳秒以及Location Month type Month int 月份.定义了十二个月的常量 Weekday type Weekday int 周,定义了一周的七天 Duration type Duration int64 持续时间.定义了以下持续时间类型.多用于时间的加减 需要传入Duration做为参数的时候.可以直接传入time.Second  const (  Nanosecond Duration = 1  Microsecond = 1000 \* Nanosecond  Millisecond = 1000 \* Microsecond  Second = 1000 \* Millisecond  Minute = 60 \* Second  Hour = 60 \* Minute  ) Location 在time包里有两个时区变量:   * time.UTC utc时间 * time.Local 本地时间   FixedZone(name string, offset int) \*Location  设置时区名,以及与UTC0的时间偏差.返回Location 时间格式化 Format(layout string) string  传入目标模板(Mon Jan 02 15:04:05 -0700 2006).时间以这个为准  p(t.Format("3:04PM"))  p(t.Format("Mon Jan \_2 15:04:05 2006"))  p(t.Format("2006-01-02T15:04:05.999999-07:00"))  p(t.Format("2006-01-02T15:04:05Z07:00"))  fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",  t.Year(), t.Month(), t.Day(),  t.Hour(), t.Minute(), t.Second())    Parse(layout, value string) (Time, error)  将字符窜转换为Time类型.  p := fmt.Println  withNanos := "2006-01-02 15:04:05"  t, \_ := time.Parse(withNanos, "2013-10-05 18:30:50")  p(t.Year())    ParseDuration(s string) (Duration, error)  将字duration符窜("ns", "us" (or "碌s"), "ms", "s", "m", "h".)转换为Duration类型.就是纳秒  p := fmt.Println  t, \_ := time.ParseDuration("1h")  p(t.Seconds())   Time相关time常用函数 Now() Time  获取当前时间,返回Time类型  Unix(sec int64, nsec int64) Time  根据秒数和纳秒,返回Time类型  Date(year int, month Month, day, hour, min, sec, nsec int, loc \*Location) Time  设置年月日返回,Time类型  Since(t Time) Duration  返回与当前时间的时间差 time常用方法 After(u Time) bool  时间类型比较,是否在Time之后  Before(u Time) bool  时间类型比较,是否在Time之前  Equal(u Time) bool  比较两个时间是否相等  IsZero() bool  判断时间是否为零值,如果sec和nsec两个属性都是0的话,则该时间类型为0  Date() (year int, month Month, day int)  返回年月日,三个参数  Year() int  返回年份  Month() Month  返回月份.是Month类型  Day() int  返回多少号  Weekday() Weekday  返回星期几,是Weekday类型  ISOWeek() (year, week int)  返回年份,和该填是在这年的第几周.  Clock() (hour, min, sec int)  返回小时,分钟,秒  Hour() int  返回小时  Minute() int  返回分钟  Second() int  返回秒数  Nanosecond() int  返回纳秒  Add(d Duration) Time  为一个时间,添加的时间类型为Duration.更精确到纳秒.比起AddDate  Sub(u Time) Duration  计算两个时间的差.返回类型Duration  AddDate(years int, months int, days int) Time  添加时间.以年月日为参数  UTC() Time  设置location为UTC,然后返回时间.就是utc为0.比中国晚了八个小时.  Local() Time  设置location为本地时间.就是电脑时间.  In(loc \*Location) Time  设置location为指定location  Location() \*Location  获取时间的Location,如果是nic,返回UTC,如果为空,则代表本地  Zone() (name string, offset int)  返回时区,以及与utc的时间偏差  Unix() int64  返回时间戳,自从1970年1月1号到现在  UnixNano() int64  返回时间戳.包含纳秒  func main() {  now := time.Now()  secs := now.Unix()  nanos := now.UnixNano()  fmt.Println(now)  millis := nanos / 1000000  fmt.Println(secs)  fmt.Println(millis)  fmt.Println(nanos)  fmt.Println(time.Unix(secs, 0))  fmt.Println(time.Unix(0, nanos))  }    GobEncode() ([]byte, error)  编码为gob  GobDecode(buf []byte) error  从gob解码  MarshalJSON() ([]byte, error)  编列为json  UnmarshalJSON(data []byte) (err error)  解码为json  func main() {  p := fmt.Println  now := time.Now()  p(now)  d := time.Duration(7200 \* 1000 \* 1000 \* 1000)  p(d)  then := time.Date(  2013, 1, 7, 20, 34, 58, 651387237, time.UTC)  p(then)  p(then.Year())  p(then.Month())  p(then.Day())  p(then.Hour())  p(then.Minute())  p(then.Second())  p(then.Nanosecond())  p(then.Location())  p(then.Weekday())  p(then.Before(now))  p(then.After(now))  p(then.Equal(now))  p(then.Date())  p(then.ISOWeek())  p("----------")  p(now.UTC())  p(now.Local())  p(now.Location())  p(now.Zone())  p(now.Unix())  p(time.Unix(now.Unix(), 0))  p(now.UnixNano())  p(time.Unix(0, now.UnixNano()))  p(now.GobEncode())  p(now.MarshalJSON())  p(time.Since(now))  p("----------")  diff := now.Sub(then)  p(diff)  p(diff.Hours())  p(diff.Minutes())  p(diff.Seconds())  p(diff.Nanoseconds())  p(then.Add(diff))  p(then.Add(-diff))  p(d)  p(d.Hours())  p(d.Minutes())  p(d.Seconds())  p(d.Nanoseconds())  p(then.Add(d))  } |

## Net包

type IP  //IP表示一个简单的IP地址，它是一个byte类型的slice，能够接受4字节（IPV4）或者16字节（IPV6）输入。注意，IP地址是IPv4地址还是IPv6地址是语义上的特性，而不取决于切片的长度：16字节的切片也可以是IPv4地址。

1. type IP []byte

func IPv4(a, b, c, d byte) IP //ipv4返回一个ipv4地址格式（a.b.c.d)的地址，这是16-byte的

func ParseCIDR(s string) (IP, \*IPNet, error)   //ParseCIDR将字符串s解析成一个ip地址和子网掩码的结构体中，其中字符串格式必须是IP地址和子网掩码的字符串，如："192.168.100.1/24"或"2001:DB8::/48“等。

func ParseIP(s string) IP  //ParseIP将s解析为IP地址，并返回该地址。如果s不是合法的IP地址表示，则ParseIP会返回nil。字符串可以是小数点分隔的IPv4格式（如"74.125.19.99"）或IPv6格式（如"2001:4860:0:2001::68"）格式。

func (ip IP) DefaultMask() IPMask  //返回IP的默认子网掩码，只有ipv4具有默认子网掩码，如果ip不是一个有效的ipv4地址，则默认子网掩码返回nil  
func (ip IP) Equal(x IP) bool          //判断两个ip地址是否相等，其中一个ipv4地址以及相同具有ipv6格式的相同的地址认为是相等的，返回true  
func (ip IP) IsGlobalUnicast() bool  //判断是否是全局单播地址  
func (ip IP) IsInterfaceLocalMulticast() bool  //判断是不是本地组播地址  
func (ip IP) IsLinkLocalMulticast() bool  //判断是否是链路本地组播地址  
func (ip IP) IsLinkLocalUnicast() bool    //判断是否时链路本地单播地址  
func (ip IP) IsLoopback() bool             //判断是否是回环地址  
func (ip IP) IsMulticast() bool       //判断是否是组播地址  
func (ip IP) IsUnspecified() bool    //判断是否是未指定地址  
func (ip IP) MarshalText() ([]byte, error)//实现了encoding.TextMarshaler的接口，其编码方式同String()函数的返回值一样。  
func (ip IP) Mask(mask IPMask) IP  //将mask作为ip的子网掩码获取其IP地址  
func (ip IP) String() string      //获取ip地址的字符串表示，如果ip是IPv4地址，返回值的格式为点分隔的，如"74.125.19.99"；否则表示为IPv6格式，如"2001:4860:0:2001::68"。  
func (ip IP) To16() IP     //将一个IP地址转换为16字节表示。如果ip不是一个IP地址（长度错误），To16会返回nil。To16可对ip地址进行转换，包括IPV4和IPV6，而To4只能对IPV4地址进行转换，这就是To16和To4的区别。  
func (ip IP) To4() IP       //将一个IPV4地址转换为4字节表示，如果ip不是一个ipv4地址，则返回nil  
func (ip \*IP) UnmarshalText(text []byte) error  //将ip进行反序列化，其实现了encoding.TextUnmarshaler的接口，IP地址字符串应该是ParseIP函数可以接受的格式。

type IPAddr   //表示一个IP终端的地址

**[html]** [view plain](http://blog.csdn.net/chenbaoke/article/details/42782521) [copy](http://blog.csdn.net/chenbaoke/article/details/42782521)

1. type IPAddr struct {
2. IP   IP
3. Zone string // IPv6 寻址范围
4. }

func ResolveIPAddr(net, addr string) (\*IPAddr, error)  //将ip地址解析成形如"host"或者"ipv6-host%zone"的地址形式,解析域名必须在指定的网络中,指定网络包括ip,ip4或者ip6  
func (a \*IPAddr) Network() string //返回地址的网络类型"ip"  
func (a \*IPAddr) String() string

type IPConn  //IPConn类型代表IP网络连接，实现了Conn和PacketConn接口  
func DialIP(netProto string, laddr, raddr \*IPAddr) (\*IPConn, error)//DialIP在网络协议netProto上连接本地地址laddr和远端地址raddr，netProto必须是"ip"、"ip4"或"ip6"后跟冒号和协议名或协议号。  
func ListenIP(netProto string, laddr \*IPAddr) (\*IPConn, error)//监听传输到本地ip地址的数据包,返回的ReadFrom和WriteTo方法能够用来发送和接受IP数据包  
func (c \*IPConn) Close() error//关闭连接  
func (c \*IPConn) File() (f \*os.File, err error)//File设定底层的os.File为阻塞模式并返回一个copy副本,当结束时,调用者需要关闭文件,其中原文件和副本之间相互不影响,返回的os.file的文件描述符与网络连接中的文件是不同的,使用该副本修改本体的属性可能会也可能不会得到期望的效果.  
func (c \*IPConn) LocalAddr() Addr  //返回本地的网络地址  
func (c \*IPConn) Read(b []byte) (int, error)  //实现conn接口的读方法,将数据读入b中  
func (c \*IPConn) ReadFrom(b []byte) (int, Addr, error)//实现了conn的readfrom方法  
func (c \*IPConn) ReadFromIP(b []byte) (int, \*IPAddr, error)//从c中读取一个ip包,将有效信息拷贝到b,返回拷贝的字节数和数据包的来源地址.可以通过timeout()使得该函数超时并且返回一个错误.  
func (c \*IPConn) ReadMsgIP(b, oob []byte) (n, oobn, flags int, addr \*IPAddr, err error)//从c中读取一个ip包,将有效数据拷贝b,相关的额外信息拷贝进oob,返回拷贝进b和oob的字节数,数据包的flag,数据包来源地址以及可能出现的错误.  
func (c \*IPConn) RemoteAddr() Addr //返回远端网络地址  
func (c \*IPConn) SetDeadline(t time.Time) error//设定读写操作的绝对过期时间,是一个时间点  
func (c \*IPConn) SetReadBuffer(bytes int) error//设定该连接的接受缓存的大小  
func (c \*IPConn) SetReadDeadline(t time.Time) error//设定读操作的绝对过期时间  
func (c \*IPConn) SetWriteBuffer(bytes int) error//设定该连接的发送缓存大小  
func (c \*IPConn) SetWriteDeadline(t time.Time) error//设定写操作的绝对过期时间  
func (c \*IPConn) Write(b []byte) (int, error)//将b中数据写入c中,并返回写入的字节数  
func (c \*IPConn) WriteMsgIP(b, oob []byte, addr \*IPAddr) (n, oobn int, err error)//将b和oob中的有效信息写入c中的地址,返回写入的字节数目  
func (c \*IPConn) WriteTo(b []byte, addr Addr) (int, error)//WriteTo实现PacketConn接口WriteTo方法  
func (c \*IPConn) WriteToIP(b []byte, addr \*IPAddr) (int, error)//通过c向add写一个ip包,并从b中复制有效信息.可以通过设定timeout值使其过期.

type IPMask  //IpMask代表一个ip地址

**[html]** [view plain](http://blog.csdn.net/chenbaoke/article/details/42782521) [copy](http://blog.csdn.net/chenbaoke/article/details/42782521)

1. type IPMask []byte

func CIDRMask(ones, bits int) IPMask //返回一个CIDRMask,其中CIDRMask总bit数目是bits,钱ones位是1,其余位是0.  
func IPv4Mask(a, b, c, d byte) IPMask//返回ip掩码,其中ip掩码形式是ipv4掩码(4 byte模式)a.b.c.d  
func (m IPMask) Size() (ones, bits int)//返回掩码的前面1的数目以及总数目,如果m不是规范的子网掩码(前面为1后面为0),则返回0,0  
func (m IPMask) String() string//返回掩码m的16机制表示,没有标点符号

type IPNet  //表示一个ip网络

**[html]** [view plain](http://blog.csdn.net/chenbaoke/article/details/42782521) [copy](http://blog.csdn.net/chenbaoke/article/details/42782521)

1. type IPNet struct {
2. IP   IP     //网络地址
3. Mask IPMask // 子网掩码
4. }

func (n \*IPNet) Contains(ip IP) bool//判定是否n中包含ip  
func (n \*IPNet) Network() string//返回地址的网络名,形式是ip+net

func (n \*IPNet) String() string//返回ipnet n的cidr模式,形如RFC 4632 和RFC 4291中定义的 "192.168.100.1/24"or "2001:DB8::/48",如果掩码不是规范模式,将会返回一个如下形式的字符串:ip地址/一个由16进制字符组成不含标点的一个字符串,例如"192.168.100.1/c000ff00".

参考：

<http://docscn.studygolang.com/pkg/net/#pkg-constants>

## Testing包

|  |
| --- |
| 单个测试  1. 文件名必须是\_test.go结尾的，这样在执行go test的时候才会执行到相应的代码 2. 你必须import testing这个包 3. 所有的测试用例函数必须是Test开头 测试用例会按照源代码中写的顺序依次执行 4. 测试格式：func TestXxx(t \*testing.T),Xxx部分可以为任意的字母数字的组合，但是首字母不能是小写字母[a-z]，例如Testintdiv是错误的函数名。 5. 函数中通过调用testing.T的Error, Errorf, FailNow, Fatal,FatalIf方法，说明测试不通过，调用Log方法用来记录测试的信息。   举个例子  package gotest  import (  "testing"  )  func Division(a, b **float64**) (**float64**, error) {  if b == 0 {  return 0, errors.New("除数不能为0")  }  return a / b, nil  }  func Test\_Division\_1(t \*testing.T) {  if i, e := Division(6, 2); i != 3 || e != nil { //测试函数  t.Error("除法函数测试没通过") // 如果不是如预期的那么就报错  } else {  t.Log("第一个测试通过了") //记录一些你期望记录的信息  }  }  func Test\_Division\_2(t \*testing.T) {  t.Error("就是不通过")  }  我们在项目目录下面执行go test,就会显示如下信息  — FAIL: Test\_Division\_2 (0.00 seconds)  gotest\_test.go:16: 就是不通过  FAIL  exit status 1  FAIL gotest 0.013s 压力测试 压力测试用来检测函数(方法）的性能，和编写单元功能测试的方法类似   1. 压力测试用例必须遵循如下格式，其中XXX可以是任意字母数字的组合，但是首字母不能是小写字母   func BenchmarkXXX(b \*testing.B) { … } go   1. test不会默认执行压力测试的函数，如果要执行压力测试需要带上参数-test.bench，语法:-test.bench=”test\_name\_regex”,例如go test -test.bench=”.\*”表示测试全部的压力测试函数 2. 在压力测试用例中,请记得在循环体内使用testing.B.N,以使测试可以正常的运行 3. 文件名也必须以\_test.go结尾   例子  package gotest  import (  "testing"  )  func Benchmark\_Division(b \*testing.B) {  for i := 0; i < b.N; i++ { //use b.N for looping  Division(4, 5)  }  }  func Benchmark\_TimeConsumingFunction(b \*testing.B) {  b.StopTimer() //调用该函数停止压力测试的时间计数  //做一些初始化的工作,例如读取文件数据,数据库连接之类的,  //这样这些时间不影响我们测试函数本身的性能  b.StartTimer() //重新开始时间  for i := 0; i < b.N; i++ {  Division(4, 5)  }  }  我们执行命令go test -file webbench\_test.go -test.bench=”.\*”，可以看到如下结果：  PASS  Benchmark\_Division 500000000 7.76 ns/op  Benchmark\_TimeConsumingFunction 500000000 7.80 ns/op  ok gotest 9.364s  上面的结果显示我们没有执行任何TestXXX的单元测试函数，显示的结果只执行了压力测试函数，第一条显示了Benchmark\_Division执行了500000000次，每次的执行平均时间是7.76纳秒，第二条显示了Benchmark\_TimeConsumingFunction执行了500000000，每次的平均执行时间是7.80纳秒。最后一条显示总共的执行时间。  GItHub 原文地址: [**https://github.com/astaxie/build-web-application-with-golang/blob/master/zh/11.3.md**](https://github.com/astaxie/build-web-application-with-golang/blob/master/zh/11.3.md) |

## assert包

assert提供的接口

|  |
| --- |
| func Equal(t \*testing.T, expected, got interface{}, messages ...interface{})  func NotEqual(t \*testing.T, expected, got interface{}, messages ...interface{})  func True(t \*testing.T, got interface{}, messages ...interface{})  func False(t \*testing.T, got interface{}, messages ...interface{})  func Nil(t \*testing.T, got interface{}, messages ...interface{})  func NotNil(t \*testing.T, got interface{}, messages ...interface{})  func Contains(t \*testing.T, expected string, got string, messages ...interface{})  func NotContains(t \*testing.T, unexpected string, got string, messages ...interface{})  func WithinDuration(t \*testing.T, duration time.Duration, goalTime, gotTime time.Time, messages ...interface{})  func Panics( t \*testing.T, expected interface{}, messages ...interface{} ) |

assert使用方法

|  |
| --- |
| 1. func TestAssert(t \*testing.T) { 2. var v interface{} = 5 3. // 直接调用包函数 4. assert.True(t, v == 5, "v的值[%v]不等于5", v) 5. assert.Equal(t, 5, v, "v的值[%v]不等于5", v) 6. assert.Nil(t, v) 7. // 以Assertion对象方式使用 8. a := assert.New(t) 9. a.True(v==5, "v的值[%v]不等于5", v) 10. a.Equal(5, v, "v的值[%v]不等于5", v) 11. a.Nil(v) 12. a.T().Log("success") 13. // 以函数链的形式调用Assertion对象的方法 14. a.True(false).Equal(5,6) 15. } |