1. 每个nsqd进程的每个channel都有一个队列，可以为队列设置*--mem-queue-size，*含义：每个channel的内存队列长度如果超过该设置，那就将接下来的消息存储在磁盘上，具体存储格式未知，但无非也就是将消息打包成大文件存储，具体的文件格式我们不作深入研究，但注意nsqd启动时需要将磁盘上的消息加载到内存中。
2. 简单阅读了下nsq的代码，主要集中在nsqd.go http.go tcp.go

tcp.go: 用于接收客户端的连接，这里的客户端主要是指订阅者，主要处理函数是Handle；

http.go: 用于接收生产者推送的发布消息，put方法主要处理函数在pubHandler，其实这里可接收很多方法，有一个deprecatedRouter来根据方法名决定相应处理函数，发布消息的过程还是蛮有趣的，需要根据消息的主题查找相应的topic结构，甚至可能还得创建多个channel。然后将消息放在topic的消息队列incomingMsgChan中。

nsqd.go:主要提供了GetTopic方法，这个方法比较有意思，因为在pub之前可能已经有多个订阅者已经等在了该topic上，所以嘛，需要有多少个订阅者就得创建多少个channel。

1. 还不太理解消息的消费过程。
2. 异常处理
3. 对于每个topic，都有一个内存消息队列和磁盘消息队列。每当nsqd接收到一个消息时，放到一个channel称为incomingMsgChan。而且在创建topic的时候会启动两个协程A和B，分别作如下两件事：

A: router()，将incomingMsgChan中的消息存放在内存消息队列或者磁盘消息队列中，优先写入内存消息队列中，如果内存消息队列存不下的话，那只能放在磁盘消息队列中。实现的时候采用了一个select + 管道 + 超时的机制，比较规矩的做法。

B: messagePump()，主要作用是将上面的内存消息队列或者磁盘消息队列中的消息放到每个订阅该topic的消费者的channel中。注意：这里还涉及到许多的通信channel，很有趣，值得学习。

1. Nsq如何解决SPOF问题

* Nsq的架构中有一lookup节点，专门负责提供查询服务。所以这个节点应该也会有个从节点来负责主节点挂掉时随时接管服务。

1. Nsq消息生产流程

在http.go中，消息的生产过程主体函数代码如下：

**// 从请求中提取topic**

\_, topic, err := s.getTopicFromQuery(req)

if err != nil {

return err

}

**// 构造消息并调用topic的PUT方法**

msg := NewMessage(<-s.context.nsqd.idChan, body)

err = topic.PutMessage(msg)

if err != nil {

return httpError{503, "EXITING"}

}

Topic的PutMessage也比较简单，只是将消息体写入一个channel即可，会有专门的协程负责后续事宜。

func (t \*Topic) PutMessage(msg \*Message) error {

t.RLock()

defer t.RUnlock()

if atomic.LoadInt32(&t.exitFlag) == 1 {

return errors.New("exiting")

}

**t.incomingMsgChan <- msg**

atomic.AddUint64(&t.messageCount, 1)

return nil

}

主要有两个协程执行后续事宜，第一是messagePump，第二是router，接下来一一简单分析。

* + Router协程的主要作用是将API写入到topic的消息放到内存消息队列或者磁盘队列，当然优先是写入到内存消息队列中，如果暂时写不进去，再将其写入磁盘消息队列中将其持久化。

func (t \*Topic) router() {

var msgBuf bytes.Buffer

for msg := range t.incomingMsgChan {

select {

**case t.memoryMsgChan <- msg:**

default:

**err := writeMessageToBackend(&msgBuf, msg, t.backend)**

if err != nil {

}

}

}

log.Printf("TOPIC(%s): closing ... router", t.name)

}

* + messagePump()负责将内存或者磁盘消息队列中的消息发送至topic下的所有的channel中

func (t \*Topic) messagePump() {

var msg \*Message

var buf []byte

var err error

var chans []\*Channel

var memoryMsgChan chan \*Message

var backendChan chan []byte

for {

select {

**case msg = <-memoryMsgChan:**

**case buf = <-backendChan:**

msg, err = decodeMessage(buf)

if err != nil {

continue

}

……

for i, channel := range chans {

chanMsg := msg

if i > 0 {

**chanMsg = NewMessage(msg.ID, msg.Body)**

chanMsg.Timestamp = msg.Timestamp

}

**err := channel.PutMessage(chanMsg)**

if err != nil

}

}

}

1. 客户端的消费过程

客户端消费过程需要指定消费哪个topic和哪个channel。如果只指定topic，而不指定channel会怎么样？有点不太理解

在IOLoop()函数中，会启动一个协程，协程的执行函数为protocolV2.messagePump()，但是这里的处理非常复杂，尚不太理解到底干了什么？

ProtocolV2.SUB()接口中会调用channel.AddClient()为channel增加一个client，即当前的channel又多了一个消费者，

Client的请求在nsqd端的入口函数为.Handle()，这里判断协议版本后就进入了主要的处理流程：protocolV2.IOtcpServerLoop()

总结：综合上面的过程来看，client的订阅过程如下：

1. 每个client可订阅在某个topic的某个channel上面，而每个channel实际上可接收多个消费者，消费者之间实际上形成了一种竞争关系，这样就方便实现负载均衡，所有的消息的订阅过程可较均匀地分配到多个client上；
2. 每个topic下面又可以创建多个channel，这样，没个消息都会复制到所有的channel中，这样便可形成一种任务的并发处理关系，比如有的时候我们希望一边实时的处理消息，同时还能将消息进行归档。使用这种机制就非常简单，我们只需为该topic创建两个channel，一个用于实时处理客户端订阅，而另外一个则由归档客户端订阅，便可双管齐下，极大简化了实现上的复杂度；
3. 每个消息被订阅出去还需要被确认，只有被确认的消息才算是真正完成。对于超时的消息或者确认失败的消息，需要放入队列并在后续重新被订阅。
4. 每个channel的队列其实有内存消息队列(memoryMsgChan chan \*Message)和磁盘消息队列(backend BackendQueue)组成。在chanel的router协程中会将topic推送过来的消息放到内存消息队列和磁盘消息队列中,，存放的原则是优先放在内存消息队列，如果内存消息队列放不下，就放在backend中。每次消费的时候也是从这两个channel中取出来放在一个统一的地方。如下

func (c \*Channel) messagePump() {

**select {**

**case msg = <-c.memoryMsgChan:**

**case buf = <-c.backend.ReadChan():**

**msg, err = decodeMessage(buf)**

if err != nil {

log.Printf("ERROR: failed to decode message - %s", err.Error())

continue

}

case <-c.exitChan:

goto exit

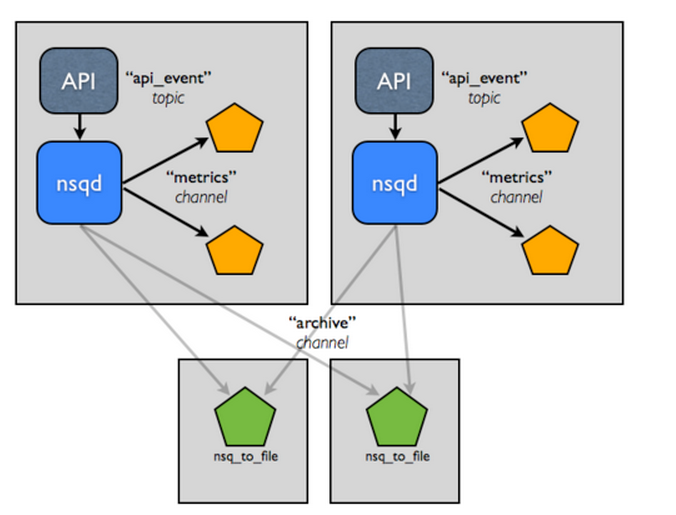
}

**c.clientMsgChan <- msg**

}

可能对于消费者来说，其并不关心消息的顺序，所以对于消息来说，其被放入内存中还是磁盘上都无关紧要。

1. 对于topic，实现上也一样，每个消息放在内存上或者磁盘消息队列上，推送的时候将消息从内存或者磁盘消息队列中读出。那为什么要搞出这两个队列呢？我直接搞个内存消息队列不就可以了么？
2. 消费者启动的时候连接的是nsqlookupd，通过这样一个内容查询服务器去定位自己感兴趣的topic所在的nsqd，然后再连接这些topic所在的所有的nsqd，为什么要连接所有的nsqd呢？这所有的 nsqd的内容是一样的还是会有所不同？难道生产者在生产的时候会将topic的数据推送到这所有的nsqd上去么？如下图所示：



之所以需要这样，是因为，如果前端部署了多个nsqd，那么必然需要采用某种机制作负载均衡，即请求会随机地分发至多个nsqd上，这样就导致了某个topic的消息会分散在多个nsqd上，而消费者为了能获取topic的所有消息，就必须要连接这所有的nsqd。这就是为什么消费者需要连接所有的nsqd。

1. Nsq的编程技巧学习
   * 利用channel控制协程退出的技巧

以nsqd为例，主程序在启动时会创建一个信号channel，一旦产生信号，会将信号写入到该信号channel，在一个单独的协程中，会检测到信号管道一旦可以读出，就向另外一个退出管道写入”1”，主协程中就一直阻塞在该退出管道的读中，如果读出内容了，那说明需要退出了，调用nsqd的退出函数，代码如下：

exitChan := make(chan int)

signalChan := make(chan os.Signal, 1)

go func() {

<-signalChan

exitChan <- 1

}()

**signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)**

……

nsqd.Main()

**<-exitChan**

nsqd.Exit()

对于nsqd这个对象来说，在创建这样一个对象的时候同样会创建一个exitChan用于它和内部一些协程之间的通信。如lookup协程和statusd协程

**n.waitGroup.Wrap(func() { n.lookupLoop() })**

**if n.options.StatsdAddress != "" {**

**n.waitGroup.Wrap(func() { n.statsdLoop() })**

**}**

在这两个协程的处理函数中，都会侦听nsqd创建的exitChan，一旦能读出，就说明需要退出了。而在nsqd的Exit函数中会调用关闭chan的方法，这样那些子协程就能够优雅地退出了。

**close(n.exitChan)**

* + **利用channel进行协程之间的同步**

还是以nsqd为例，它又一个notifyChan专门用来在nsqd和lookup协程之间进行同步，nsqd协程中的Notify方法

func (n \*NSQD) Notify(v interface{}) {

select {

case <-n.exitChan:

**case n.notifyChan <- v:**

n.Lock()

err := n.PersistMetadata()

if err != nil {

log.Printf("ERROR: failed to persist metadata - %s", err.Error())

}

n.Unlock()

}

}

如果有事件发生，那么就向notifyChan中写入v，其他地方会阻塞在该notifyChan的读事件上。比如lookup协程：

**case val := <-n.notifyChan:**

var cmd \*nsq.Command

var branch string

switch val.(type) {

这里就实现了主协程中向子协程发送一个通知，并将通知的信息通过interface来传递的功能，子协程收到通知并解析出参数，进行接下来的处理（向nsqlookupd发送更新channel或者topic的命令）。

* + time.Ticker的大量运用

关于ticker的使用可以参考<http://mmcgrana.github.io/2012/09/go-by-example-timers-and-tickers.html>

time.Ticker代表了定时事件。Nsq的实现里面大量运用了ticker，简单举例如下：

1. 在statsd协程中，会注册一个这样的定时器，每隔一定时间(n.options.StatsdInterval)其channel就会变得readable。这样就可以更新当前的统计状态了。代码实例如下：

**ticker := time.NewTicker(n.options.StatsdInterval)**

for {

select {

case <-n.exitChan:

goto exit

**case <-ticker.C:**

1. 使用ticker来实现heartbeat

在lookup.go中nsqd需要定期与lookup服务交互心跳，这又是ticker应用的一个极佳例子：

**ticker := time.Tick(15 \* time.Second)**

**for {**

**select {**

**case <-ticker:**

**// send a heartbeat and read a response (read detects closed conns)**

**for \_, lookupPeer := range n.lookupPeers {**

**log.Printf("LOOKUPD(%s): sending heartbeat", lookupPeer)**

**cmd := nsq.Ping()**

**\_, err := lookupPeer.Command(cmd)**

**if err != nil {**

**}**

**}**

* + godep的使用

解决package依赖问题：

1. <https://github.com/tools/godep>
2. <http://www.goinggo.net/2013_10_01_archive.html>