## NSQ内部模块设计

本文档试图通过对NSQ内部模块设计来学习先进的编程设计和思想，以提升自己在大型系统软件编程方面的经验和能力。

NSQ意在对外部提供一个统一、可靠的异步消息队列。应用程序可以将消息放心地存入NSQ且无需担心消息丢失。同时，还可以通过nsq提供的工具或者API来访问存储的消息。

实现中，NSQ对整个系统作了如下抽象：

1. NSQD：生产者，每个NSQD在设计者看来就是一个生产者。生产者之间相互独立，可减少系统之间的耦合程度。应用程序可以向每个生产者写入多个topic的消息。关于topic我们会在下面描述。
2. Client：消费者。与生产者扮演相反的角色。消费者可以连接多个NSQD，实际上消费者主要是针对感兴趣的topic。因此，生产者NSQD和消费者Client之间其实是一种N:M的关系。
3. Topic:消息的主题。每个主题的消息可能会存储在多个生产者NSQD上。Topic其实是一组相关消息的集合，这没什么可深究的地方。
4. Channel:管道。顾名思义，channel是消息经由生产者流入消费者的通道。NSQ实现了一个比较有趣的模型：一个消息可以经由多个管道流向多个消费者。其实这是一个非常强大的模型，我们就不在此深究。每个channel我们可以理解成消息的一个副本流向一个客户端的管道。因此，每个topic会对应多个channel，而且可以动态的添加、删除channel。
5. Message:消息的载体。无需太多深究。
6. NSQLOOKUP:是生产者和消费者的中间人。主要负责给消费者提供其感兴趣的topic所在的生产者的位置信息。除此之外，它还提供了系统整体的统计信息。如当前的topic、channel等信息。因此，每个生产者必须定时和它保持联系，并在生产者状态改变时，以某种方式告知NSQD。

总的说来，整个NSQ系统由上面的几个大的模块组成，而且，这几个模块之间并非相互独立，还存在着千丝万缕的联系。我们接下来分析的重点就是在弄清楚每个模块功能的同时重点关注它和其它模块之间交互的实现方式。

### NSQD

#### 内部模块

NSQD是消息的源头。应用程序将消息写入NSQD，NSQD负责存储并转发这些消息。作者在系统设计时尽量减少各个模块之间的耦合程度。NSQD相关联的模块只有topic，NSQD有专门的成员保存了它上面存储的所有topic的信息。除此之外，NSQD还需要通信的模块就只有NSQLOOKUP了，因为它需要向NSQLOOKUP汇报自己的信息。

NSQD模块内部主要有以下几个独立协程完成相关功能：

1. idPump()：没研究其具体作用是干啥
2. lookupLoop()：负责与NSQLOOKUP模块通信协程；
3. tcpServer()：负责接收消费者的tcp请求；
4. httpsServer()：负责接收并处理应用程序的消息推送请求；
5. statsdLoop()：应该是负责处理统计状态信息的请求；

可以看到，这些子协程之间相互独立，彼此之间无需任何交互。他们与NSQD主协程之间通过管道进行协程之间的通信，所谓的通信也只是在主协程退出的时候如何以一种优雅的方式通知它的所有子协程，在所有子协程完成自己的退出逻辑后主协程再退出。

这就是为什么NSQD结构定义中存在一个exitChan的原因。在上面所有的子协程中都存在如下的语句：

select {

case <-n.exitChan:

goto exit

}

这样，主协程中的退出函数中调用close(exitChan)的话，所有子协程中都会进入case，这样子协程就进入退出逻辑处理了，非常优雅。

#### 外部交互

Nsq中记录了该生产者所有的topic列表，同时其暴露了一些方法用来查询NSQD上的topic信息。除此之外，NSQD基本上不会存在与topic过多的交互。设计上做到尽量松耦合。不过我们会在后面说到topic与NSQD之间还有一点千丝万缕的联系，这是后话。

* **NSQLOOKUP交互**

因为NSQD需要向LOOKUP协程汇报自己所有的状态信息。以topic的变化为例，我们看看NSQD如何优雅地设计。

如果系统中动态增加、删除了一个topic。此时，TOPIC模块会通过特定的机制通知到NSQD(我们会在后面仔细描述)，NSQD会将这个事件通知到自己的lookupLoop子协程，由子协程负责全权处理与NSQLOOKUP模块的交互。NSQD与lookupLoop子协程之间共享一个称为notifyChan的管道，NSQD将事件写入该channel，子协程就可以在自己的处理循环中侦听该管道，有事件便可第一时间感知并处理。

### TOPIC

前面我们说到，topic是一组消息的载体。一组消息可以通过多个channel重复发送至多个客户端。因此，topic可以说是生产者NSQD和消息传输通道channel之间的管理者。其上接生产者，下接管道，它负责将将消息从生产者重复推送至多个消息通道。因此，其设计上相较于NSQD必然会复杂很多。同样，我们也会从”内部模块”和”外部交互”两个方面来窥探topic的设计。

首先我们看看topic内部数据结构的设计，

1. **channelMap**：因为每个topic可能会有多个消息通道，因此，我们需要一个结构来保存所有管道的信息
2. **incomingMsgChan：**外部向topic写入消息的管道，唯一管道
3. **backend：**每个消息不仅仅要存储在内存中，为了可靠性，我们还必须将其刷入磁盘，因此，需要一个队列来保存刷入磁盘的消息
4. **memoryMsgChan：**保存储存在内存中的消息队列
5. **exitChan：**topic主协程和子协程同步channel，机制与NSQD中描述的一致
6. **channelUpdateChan：**管理员可能随时会为每个topic增加一个channel，需要一种机制来通知topic
7. **pauseChan：**管理员可暂停一个消息主题，这种事件也需要一个通知媒介。

#### 2.1 内部模块

* **router协程**

主要作用是将incomingMsgChan中写入的外部消息存入内存消息队列或者磁盘消息队列中。优先存入内存消息队列，如果内存消息队列满，则存入磁盘消息队列。

* **messagePump协程**

负责将内存或者磁盘消息队列中的消息推送至该topic下面所有的channel中。注意的是：这里需要处理如下情况：1) 增加、删除channel；2) 暂停、继续topic。

* **Notify协程**

每当创建一个新的topic或者删除一个已有的topic的时候，topic内部会创建一个子协程来通知NSQD这一事件。而通知的方法也很简单，向topic与NSQD共享的一个称为notifyChan管道中写入具体的topic信息即可。

结合我们在NSQD中的描述可以发现，topic其实并不感知notifyChan，它只是创建一个协程，而该协程执行了NSQD中的函数，具体的操纵notifyChan也只是NSQD内部的事情。Topic只是提供了”消息是什么”这个信息而已。

#### 外部交互

* **接口**

**PutMessage/ PutMessages**

Topic对外提供的接口主要是PutMessage/PutMessages。其他模块通过该接口向topic中写入一个或者一批消息。内部实现时，将消息写入incomingMsgChan即可。

**Pause**

用于暂停一个topic，实现中，向每个topic内部的pauseChan写入一个bool消息，true代表暂停，false代表从暂停状态继续。在messagePump中会尝试读这个pauseChan以便作后续处理。

**GetChannel/** **DeleteExistingChannel**

用于获取或者删除一个channel，实现中向channelUpdateChan写入”1”，后面的messagePump会尝试读channelUpdateChan，如果读出并作后续处理即可。

### CHANNEL

在NSQ的设计中，channel是消息从生产者流入消费者的管道。每个topic可绑定多个channel，同时，channel可以动态改变。而且，每个channel上可以绑定多个消费者。消费者采用PULL的方式从channel中主动去获取消息。

首先我们可以看看channel数据结构的定义。

**IncomingMsgChan：**topic模块会将消息写入这里

**memoryMsgChan：**channel的内存消息队列

**backend：**channel的磁盘消息队列

**clientMsgChan：**是channel与client模块间消息交互的管道，下面会仔细阐述

**exitChan：**channel的主协程和子协程的同步管道，机理类似其他模块

**clients：**每个channel上绑定的所有client

#### 3.1 内部模块

与topic类似，channel也有如下自己的内部模块：

* messagePump

该子协程专门负责将内存&磁盘消息队列中的消息写入至**clientMsgChan**

* router

负责将**IncomingMsgChan**中的消息写入内存或者磁盘消息队列中

* deferredWorker(暂时不明)
* inFlightWorker(暂时不明)
* nsqd.Notify

这个和topic中介绍的机制一模一样。

注意：这些子协程和channel主协程之间的通信基本都采用了exitChan，这和上面描述的机制基本一致。

#### 外部交互

* 与Client模块交互

作为消息传递的通道，channel需要与上游的topic以及下游的消费者交互。它与topic的交互很简单，是一个单向的过程：topic将消息写入channel的接收队列即可。而它与下游消费者的交互就略显复杂：因为可能会有多个client绑定至同一个channel。

简单来说，channel和client之间的交互通过管道clientMsgChan进行。在channel模块中，messagePump协程负责将内存或者磁盘消息队列中的消息写入clientMsgChan中，而client端负责从该clientMsgChan中读取消息体（当然client端会做比较复杂的控制，我们会在后面描述）。

当channel退出的时候，它会显示地调用close(clientMsgChan)，这样，从该channel上读取消息的client也会检测到该事件并根据自己的实际情况作相应的处理。

### Client

在NSQ中，client就是消费者了。每个消费者会绑定至某个topic的某个channel上，然后从该channel上消费信息。

严格意义上来说，client只会与channel模块打交道，当然了，为了找到这些channel所在的NSQD信息，它需要首先与LOOKUP模块通信，进行一次查找。

首先我们简单描述下client的连接以及请求处理过程，这对我们后面的理解可能会有些帮助。

我们在NSQD模块中说到过，该模块会开启一个tcp监听器，这就是专门用来侦听消费者的连接请求的：

**func (p \*tcpServer) Handle(clientConn net.Conn) {**

log.Printf("TCP: new client(%s)", clientConn.RemoteAddr())

…

buf := make([]byte, 4)

\_, err := io.ReadFull(clientConn, buf)

if err != nil {

log.Printf("ERROR: failed to read protocol version - %s", err.Error())

return

}

protocolMagic := string(buf)

var prot util.Protocol

switch protocolMagic {

case " V2":

prot = &protocolV2{context: p.context}

default:

util.SendFramedResponse(clientConn, frameTypeError, []byte("E\_BAD\_PROTOCOL"))

clientConn.Close()

return

}

**err = prot.IOLoop(clientConn)**

**……**

**}**

上面的部分，NSQ接受了消费者的连接请求，后面会在prot.IOLoop()中处理该连接上客户端发送的各种命令。

**func (p \*protocolV2) IOLoop(conn net.Conn) error {**

var err error

var line []byte

var zeroTime time.Time

clientID := atomic.AddInt64(&p.context.nsqd.clientIDSequence, 1)

**client := newClientV2(clientID, conn, p.context)**

messagePumpStartedChan := make(chan bool)

**go p.messagePump(client, messagePumpStartedChan)**

<-messagePumpStartedChan

……

}

可以看到，在这里为客户端创建了一个client结构，并启动了额外的协程：**messagePump**。后面我们会看到，client对于消息的消费就是由这个协程来控制的。

而IOLoop接下来的处理就是从该连接上读取命令并执行命令：

**response, err := p.Exec(client, params)**

例如，我们下面就以SUB命令为例来看看它是如何处理客户端的消费请求的。这个命令的具体处理函数是

**func (p \*protocolV2) SUB(client \*clientV2, params [][]byte) ([]byte, error) {**

接下来，我们简单描述下client数据结构的定义：

**ExitChan：**控制主协程与子协程同步的管道，与前面的描述一致

**IdentifyEventChan：**目前尚不确定其用途

**SubEventChan：**用来控制消费的管道，见后面详述

#### 4.1 消费过程

对于client模块，其最大的亮点可能就是消息消费过程的设计了。我们从从说起这个故事。

故事的开始是客户端通过与NSQD的连接发来了SUB命令。我们看看SUB命令的处理：

topic := p.context.nsqd.GetTopic(topicName)

channel := topic.GetChannel(channelName)

channel.AddClient(client.ID, client)

atomic.StoreInt32(&client.State, stateSubscribed)

client.Channel = channel

**client.SubEventChan <- channel**

return okBytes, nil

上面处理过程中最主要的就是这句了：**client.SubEventChan <- channel**

它将client需要消费的channel写入了管道**SubEventChan**中。接下来我们看看谁从这个管道中读取信息：

在**func (p \*protocolV2) messagePump(client \*clientV2, startedChan chan bool)** 中，

case subChannel = <-subEventChan:

// you can't SUB anymore

subEventChan = nil

对上面的理解：一旦一个client调用了SUB请求，向subEventChan写入了内容，而messagePump携程从该subEventChan读出了内容，接下来子协程将subEventChan置为nil，以后在select的时候就会自动忽略掉该subEventChan，因此，一个客户端只可能调用SUB一次。Client调用了SUB以后，接下来的消息推送是由messagePump来控制了，它会考虑每个client的处理能力来决定是否再从channel中读取消息。如果client处理的快，那么它消费的消息就会多点。

另外需要注意一点就是：如果多个client同时来消费一个channel，那么其实这所有的client形成的就是一种竞争关系。