



NSQ指南

极客学院出版

前言

NSQ 是实时的分布式消息处理平台，其设计的目的是用来大规模地处理每天数以十亿计级别的消息。它具有分布式和去中心化拓扑结构，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。

本指南是 [NSQ 官网](#) 的中文翻译版本。

适用人群

本教程是给那些想详细了解如何使用 NSQ 分布式实时系统的开发人员编写的。

学习前提

在学习本教程之前，你需要对 Go 语言和计算机网络相关的知识有一定了解。

你将学会

- NSQ 性能和设计模式
- NSQ 组件和客户端库
- NSQ 部署

版本信息

书中演示代码基于以下版本：

平台	版本信息
NSQ	0.3.5

目录

前言	1
第 1 章 概述	3
介绍	4
快速开始	6
特性和担保	7
常见问题	9
性能	13
设计	16
内幕	23
第 2 章 组件	33
nsqd	34
nsqlookupd	44
nsqadmin	46
工具	48
第 3 章 客户端	52
TCP 协议规范	53
客户端库	61
编译客户端库	63
第 4 章 部署	73
安装	74
产品配置	76
拓扑模式	78
Docker	85



T



1

概述



















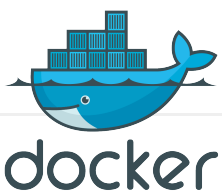







介绍

NSQ 是实时的分布式消息处理平台，其设计的目的是用来大规模地处理每天数以十亿计级别的消息。

NSQ 具有分布式和去中心化拓扑结构，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。参见 [features & guarantees](#)。

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。另外，官方还提供了开箱即用 Go 和 Python 库。如果读者有兴趣构建自己的客户端的话，还可以参考官方提供的[协议规范](#)。

产品

Twitter

[Favorite Tweets by @nsqio](#)

帮助

- 源码: <https://github.com/bitly/nsq>
- 问题: <https://github.com/bitly/nsq/issues>
- 邮件组: nsq-users@googlegroups.com
- IRC: #nsq on freenode
- Twitter: [@nsqio](#)

问题

所有的问题必须通过 [github issues](#) 提交。提交前请搜索一下之前的问题，避免重复。

贡献

NSQ 拥有一个成长型的社区，欢迎贡献者（尤其是文档方面）。大家可以 fork 工程 [github](#) 并提交 pull 请求。

致谢

NSQ 是由 Matt Reiferson ([@imsnakes](#)) 和 Jehiah Czebotar ([@jehiah](#)) 设计开发的，同时也离不开 [bitly](#) 和所有 [contributors](#) 贡献者们。

快速开始

下面的步骤将通过推送(publishing)、消费(consuming)和归档(archiving)消息到本地磁盘，在本地环境演示一个小型的 NSQ 集群

1. 根据[文档安装](#)安装 NSQ。
2. 在另外一个 shell 中，运行 `nsqlookupd`：

```
$ nsqlookupd
```

3. 再开启一个 shell，运行 `nsqd`：

```
$ nsqd --lookupd-tcp-address=127.0.0.1:4160
```

4. 再开启第三个 shell，运行 `nsqadmin`：

```
$ nsqadmin --lookupd-http-address=127.0.0.1:4161
```

5. 开启第四个 shell，推送一条初始化数据(并且在集群中创建一个 topic):

```
$ curl -d 'hello world 1' 'http://127.0.0.1:4151/put?topic=test'
```

6. 最后，开启第五个 shell，运行 `nsq_to_file`：

```
$ nsq_to_file --topic=test --output-dir=/tmp --lookupd-http-address=127.0.0.1:4161
```

7. 推送更多地数据到 `nsqd`：

```
$ curl -d 'hello world 2' 'http://127.0.0.1:4151/put?topic=test'
$ curl -d 'hello world 3' 'http://127.0.0.1:4151/put?topic=test'
```

8. 按照预先设想的，在浏览器中打开 `http://127.0.0.1:4171/` 就能查看 `nsqadmin` 的 UI 界面和队列统计数据。同时，还可以在 `/tmp` 目录下检查 (`test*.log`) 文件。

这个教程中最重要的是：`nsq_to_file` (客户端)没有明确地指出 `test` 主题从哪里产生，它从 `nsqlookupd` 获取信息，即使在消息推送之后才开始连接 `nsqd`，消息也并没有消失。

特性和担保

NSQ 是分布式实时消息系统。

特性

- 支持无 SPOF 的分布式拓扑
- 水平扩展(没有中间件, 无缝地添加更多的节点到集群)
- 低延迟消息传递 ([性能](#))
- 结合负载均衡和多播消息路由风格
- 擅长面向流媒体(高通量)和工作(低吞吐量)工作负载
- 主要是内存中(除了高水位线消息透明地保存在磁盘上)
- 运行时发现消费者找到生产者服务([nsqlookupd](#))
- 传输层安全性 (TLS)
- 数据格式不可知
- 一些依赖项(容易部署)和健全的, 有界, 默认配置
- 任何语言都有简单 TCP 协议支持客户端库
- HTTP 接口统计、管理行为和生产者(不需要客户端库发布)
- 为实时检测集成了 [statsd](#)
- 健壮的集群管理界面 ([nsqadmin](#))

担保

对于任何分布式系统来说, 都是通过智能权衡来实现目标。通过这些透明的可靠性指标, 我们希望能使得 NSQ 在部署到产品上的行为是可达预期的。

消息不可持久化 (默认)

虽然系统支持消息持久化存储在磁盘中 (通过 `--mem-queue-size`), 不过默认情况下消息都在内存中。

如果将 `--mem-queue-size` 设置为 0，所有的消息将会存储到磁盘。我们不用担心消息会丢失，nsq 内部机制保证在程序关闭时将队列中的数据持久化到硬盘，重启后就会恢复。

NSQ 没有内置的复制机制，却有各种各样的方法管理这种权衡，比如部署拓扑结构和技术，在容错的时候从属并持久化内容到磁盘。

消息最少会被投递一次

如上所述，这个假设成立于 `nsqd` 节点没有错误。

因为各种原因，消息可以被投递多次（客户端超时，连接失效，重新排队，等等）。由客户端负责操作。

接收到的消息是无序的

不要依赖于投递给消费者的消息的顺序。

和投递消息机制类似，它是由重新队列(requeues)，内存和磁盘存储的混合导致的，实际上，节点间不会共享任何信息。

它是相对的简单完成疏松队列，（例如，对于某个消费者来说，消息是有次序的，但是不能给你作为一个整体跨集群），通过使用时间窗来接收消息，并在处理前排序（虽然为了维持这个变量，必须抛弃时间窗外的消息）。

消费者最终找出所有话题的生产者

这个服务(`nsqlookupd`) 被设计成最终一致性。`nsqlookupd` 节点不会维持状态，也不会回答查询。

网络分区并不会影响可用性，分区的双方仍然能回答查询。部署性拓扑可以显著的减轻这类问题。

常见问题

部署

- 有什么为 `nsqd` 推荐的拓扑结构？

强烈推荐 `nsqd` 和生产消息的服务一起运行。

`nsqd` 是一个相对轻量的进程，它能很好和其他进程协同运行。

这个模式有利于结构化消息流为一个消费问题，而不是一个生产问题。

另一个好处是它能将来自服务端的内容形成有效的独立，分享，筒仓（silo）的数据。

注意: 这并不是必须得要求，它只是能让事情简单些（参见下面的问题）。

- 为什么不能用 `nsqlookupd` 来查询生产的内容给谁？

NSQ 提升了消费端的发现模型，减轻了前期的配置负载（需要告诉所有消费者去那里找他们要的内容）。

然而，它并没有提供任何方法来解决发布端将内容发布给谁。这是鸡和蛋的问题，在发布前并不存在内容。

通过使用 `nsqd`，你可以避开这个问题（你的服务只是简单的将内容发布给本地的 `nsqd`），并且允许 NSQ 实时发现系统正常运行。

- 我只是想在某个节点上将 `nsqd` 作为一个工作队列来使用，有没有合适的例子？

是的，`nsqd` 可以很好的单独运行。

`nsqlookupd` 非常有利于大型分布式环境。

- 我需要运行多少个 `nsqlookupd` ？

依赖于集群的大小，`nsqd` 的节点数量，消费者，和你希望的容错能力。

3 个或 5 个就可以非常好的服务于百级别的主机和千级的消费者。

`nsqlookupd` 节点不需要回答查询。集群里的元数据是最终一致的。

发布

- 是否需要客户端库来发布消息？

不需要！使用 HTTP 节点来发布消息就好（`/pub` 和 `/mpub`）。它简单，容易，在任意一个开发环境都可用。

绝大多数人使用 HTTP 来发布 NSQ 部署。

- 为什么强制客户端响应 TCP 协议 `PUB` 和 `MPUB` 命令？

我们相信 NSQ 操作的默认模式必须安全优先，并且我们希望协议简单并完整。

- 什么时候 `PUB` 或 `MPUB` 会失败？

1. 话题（topic）的名字没有正确格式化（长度限制）。参见[topic and channel name spec](#)。
2. 消息过大(具体限制参见 `nsqd` 的参数)。
3. 中间的话题（topic）被删除。
4. `nsqd` 被清除。
5. 发布的时候客户端产生连接失败

(1) 和 (2) 是开发错误。(3) 和 (4) 很少见，(5) 是基于 TCP 协议都会遇到的问题。

- 如何避免之前 (3) 出现的问题？

删除话题（topic）是少见的操作。如果你想删除一个话题（topic），需要精确计算时间，确保删除后有充足的时间，发布的话题（topic）不会被执行。

设计和理论

- 如何命名话题（topic）和通道（channel）？

话题（topic）名需要描述在流中的数据。

通道（channel）名需要描述消费者的工作类型。

例如，好的话题（topic）名 `编码（encode）`，`解码（decode）`，`api_请求（api_request）`，`页面_视图`。
好的通道（channel）名 `归档（archive）`，`分析_增长（analytics_increment）`，`垃圾_分析（spam_analysis）`。

- 一个 `nsqd` 最多能支持多少个话题 (topic) 和通道 (channel) ?

没有内置的限制。它仅和 `nsqd` 所在的服务端的内存, CPU 限制有关 (每个客户端 CPU 使用率已经大为改进了[issue #236](#))。

- 如何为集群声明一个新的话题 (topic) ?

话题 (topic) 的第一个 `PUB` 或 `SUB`, 将会在 `nsqd` 上创建一个话题 (topic)。话题 (topic) 的元数据将会传播给 `nsqlookupd` 的配置。其他的读者将会通过周期性的查询 `nsqlookupd` 发现这个话题 (topic)。

- ** NSQ 能操作 RPC 吗? **

是的, 有这个可能性, 但是 NSQ 并不是为它设计的。

我们想发布一些文档说明它是如何结构化的, 如果你感兴趣, 可以来帮我们。

特定的 `pynsq`

- 为什么强制我使用 Tornado?

`pynsq` 初始设计的时候, 就聚焦于消费端的库, 并且 NSQ 协议和 Python 的异步架构非常类似 (尤其和 NSQ 的面向推送协议)。

Tornado 的 API 非常简单并且执行合理。

- Tornado `IOLoop` 是否必须发布?

不, `nsqd` 为了发布简单, 暴露了 HTTP 端 (`/pub` 和 `/mpub`)。

不必担心 HTTP 的过载。同时, `/mpub` 通过批量发布, 减少了 HTTP 的过载。

- 那么什么时候使用 `Writer` ?

当高性能, 低负载优先级比较高的时候。

`Writer` 使用 TCP 协议里的 `PUB` 和 `MPUB` 命令, 它们比 HTTP 负载更低。

- 如果我就想”启动并忘记“将会发生什么(我能容忍消息丢失)?

使用 `Writer` 并且不给发布的方法指定回调。

注意: 仅在简单的客户端代码有效, `pynsq` 场景必须处理 `nsqd` 的消息 (比如, 做这些事情不会导致性能提高)。

特别感谢 Dustin Oprea ([@DustinOprea](#)) 开始了这篇常见问题。

性能

分布式性能

主仓库包含一段代码（ `bench/bench.py` ），它能在 EC2 上自动完成分布式基准。

它引导 `N` 个节点，一些运行 `nsqd`，一些运行加载生成工具（ `PUB` 和 `SUB` ），并分析它们的输出来提供聚合。

初始化

下面的代码反应了默认参数 `6 c3.2xlarge`，这个实例支持 1g 比特的连接。3 个节点运行 `nsqd` 实例，剩下的运行 `bench_reader`（ `SUB` ）和 `bench_writer`（ `PUB` ）实例，来生成依赖于基准模式的负载。

```
$ ./bench/bench.py --access-key=... --secret-key=... --ssh-key-name=...
[I 140917 10:58:10 bench:102] launching 6 instances
[I 140917 10:58:12 bench:111] waiting for instances to launch...
...
[I 140917 10:58:37 bench:130] (1) bootstrapping ec2-54-160-145-64.compute-1.amazonaws.com (i-0a018ce1)
[I 140917 10:59:37 bench:130] (2) bootstrapping ec2-54-90-195-149.compute-1.amazonaws.com (i-0f018ce4)
[I 140917 11:00:00 bench:130] (3) bootstrapping ec2-23-22-236-55.compute-1.amazonaws.com (i-0e018ce5)
[I 140917 11:00:41 bench:130] (4) bootstrapping ec2-23-23-40-113.compute-1.amazonaws.com (i-0d018ce6)
[I 140917 11:01:10 bench:130] (5) bootstrapping ec2-54-226-180-44.compute-1.amazonaws.com (i-0c018ce7)
[I 140917 11:01:43 bench:130] (6) bootstrapping ec2-54-90-83-223.compute-1.amazonaws.com (i-10018cfc)
```

生产者吞吐量

这个基准仅反应了生产者吞吐量。消息体有 100 个字节，并且消息通过 3 个话题（ `topic` ）分布。

```
$ ./bench/bench.py --access-key=... --secret-key=... --ssh-key-name=... --mode=pub --msg-size=100 run
[I 140917 12:39:37 bench:140] launching nsqd on 3 host(s)
[I 140917 12:39:41 bench:163] launching 9 producer(s) on 3 host(s)
...
[I 140917 12:40:20 bench:248] [bench_writer] 10.002s - 197.463mb/s - 2070549.631ops/s - 4.830us/op
```

入口处 `~2.07mm` msgs/sec,使用了 `197mb/s` 的带宽。

生产和消费吞吐量

通过服务生产者和消费者，这个基准更加准确的反应了实际情况。这个消息也是 100 个字节，并且通过 3 个话题（topic）分布，每个都包含一个通道（channel）（每个通道（channel）24 个客户端）。

```
$ ./bench/bench.py --access-key=... --secret-key=... --ssh-key-name=... --msg-size=100 run
[I 140917 12:41:11 bench:140] launching nsqd on 3 host(s)
[I 140917 12:41:15 bench:163] launching 9 producer(s) on 3 host(s)
[I 140917 12:41:22 bench:186] launching 9 consumer(s) on 3 host(s)
...
[I 140917 12:41:55 bench:248] [bench_reader] 10.252s - 76.946mb/s - 806838.610ops/s - 12.706us/op
[I 140917 12:41:55 bench:248] [bench_writer] 10.030s - 80.315mb/s - 842149.615ops/s - 11.910us/op
```

入口处的 **~842k** **~806k** msgs/s, 合计消费带宽 **156mb/s**，我们已经尽力提升了 nsqd 节点的 CPU 处理能力。通过引入消费者，nsqd 需要维持每个通道（channel），因此负载自然会高一点。

消费者的数量略微少于生产者，因为消费者发送2次命令（每个消息都要发送 FIN 命令）。

增加两个节点（一个是 nsqd 另一个是产生负载），达到了 **1mm** msgs/s:

```
$ ./bench/bench.py --access-key=... --secret-key=... --ssh-key-name=... --msg-size=100 run
[I 140917 13:38:28 bench:140] launching nsqd on 4 host(s)
[I 140917 13:38:32 bench:163] launching 16 producer(s) on 4 host(s)
[I 140917 13:38:43 bench:186] launching 16 consumer(s) on 4 host(s)
...
[I 140917 13:39:12 bench:248] [bench_reader] 10.561s - 100.956mb/s - 1058624.012ops/s - 9.976us/op
[I 140917 13:39:12 bench:248] [bench_writer] 10.023s - 105.898mb/s - 1110408.953ops/s - 9.026us/op
```

单个节点性能

声明：请牢记 NSQ 设计的初衷是分布式。单个节点的性能非常重要，但这并不是我们所追求的。

- 2012 MacBook Air i7 2ghz
- go1.2
- NSQ v0.2.24
- 200 byte messages

GOMAXPROCS=1 (单个生产者, 单个消费者)

```
$ ./bench.sh
results...
PUB: 2014/01/12 22:09:08 duration: 2.311925588s - 82.500mb/s - 432539.873ops/s - 2.312us/op
SUB: 2014/01/12 22:09:19 duration: 6.009749983s - 31.738mb/s - 166396.273ops/s - 6.010us/op
```

GOMAXPROCS=4 (4 个生产者, 4 个消费者)

```
$ ./bench.sh
results...
PUB: 2014/01/13 16:58:05 duration: 1.411492441s - 135.130mb/s - 708469.965ops/s - 1.411us/op
SUB: 2014/01/13 16:58:16 duration: 5.251380583s - 36.321mb/s - 190426.114ops/s - 5.251us/op
```


设计

注意：可视化的演示参见 [slide deck](#)。

NSQ 是继承于 [simplequeue](#)(部分的 [simplequeue](#))，因此被设计为（排名不分先后）

- 提供更简单的拓扑方案，达到高可用性和消除单点故障
- 满足更强的消息可靠传递的保证
- 限制单个进程的内存占用（通过持久化一些消息到硬盘上）
- 极大简化了生产者和消费者的配置要求
- 提供了一个简单的升级路径
- 提升效率

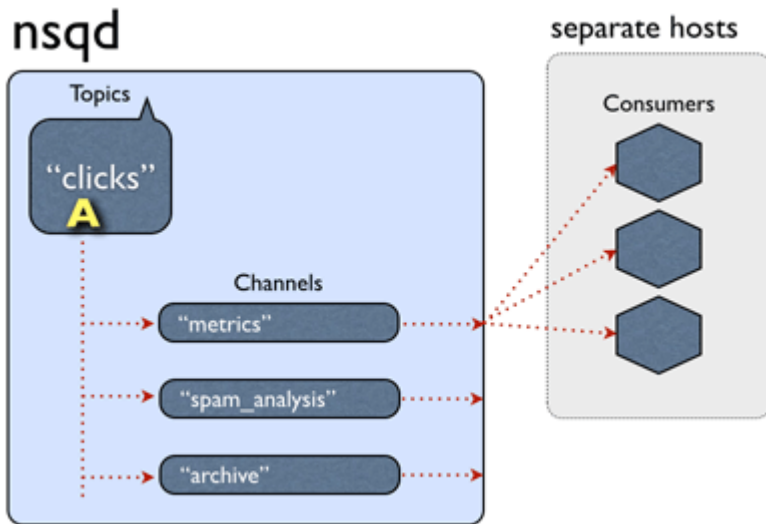
简化配置和管理

单个 `nsqd` 实例被设计成可以同时处理多个数据流。流被称为“话题”和话题有 1 个或多个“通道”。每个通道都接收到一个话题中所有消息的拷贝。在实践中，一个通道映射到下行服务消费一个话题。

话题和通道都没有预先配置。话题由第一次发布消息到命名的话题或第一次通过订阅一个命名话题来创建。通道被第一次订阅到指定的通道创建。

话题 和通道的所有缓冲的数据相互独立，防止缓慢消费者造成对其他通道的积压（同样适用于话题级别）。

一个通道一般会有多个客户端连接。假设所有已连接的客户端处于准备接收消息的状态，每个消息将被传递到一个随机的客户端。例如：



图片 1.1 nsqd clients

总之，消息从话题→通道是多路传送的（每个通道接收的所有该话题消息的副本），即使均匀分布在通道→消费者之间（每个消费者收到该通道的消息的一部分）。

NSQ 还包括一个辅助应用程序，`nsqllookupd`，它提供了一个目录服务，消费者可以查找到提供他们感兴趣订阅话题的 `nsqd` 地址。在配置方面，把消费者与生产者解耦开（它们都分别只需要知道哪里去连接 `nsqllookupd` 的共同实例，而不是对方），降低复杂性和维护。

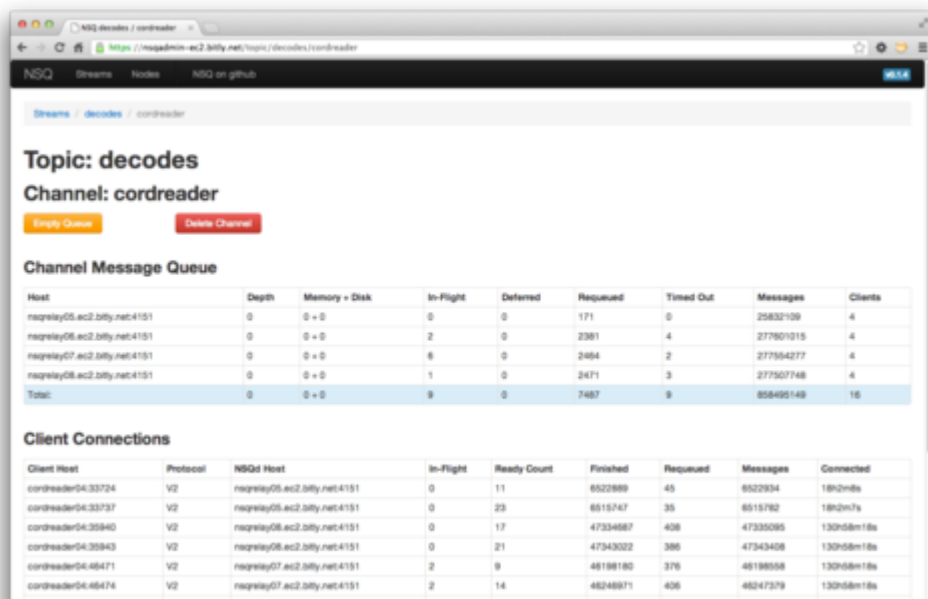
在更底的层面，每个 `nsqd` 有一个与 `nsqllookupd` 的长期 TCP 连接，定期推动其状态。这个数据被 `nsqllookupd` 用于给消费者通知 `nsqd` 地址。对于消费者来说，一个暴露的 HTTP `/lookup` 接口用于轮询。

为话题引入一个新的消费者，只需启动一个配置了 `nsqllookupd` 实例地址的 NSQ 客户端。无需为添加任何新的消费者或生产者更改配置，大大降低了开销和复杂性。

注：在将来的版本中，启发式 `nsqllookupd` 可以基于深度，已连接的客户端数量，或其他“智能”策略来返回地址。当前的实现是简单的返回所有地址。最终的目标是要确保所有深度接近零的生产者被读取。

值得注意的是，重要的是 `nsqd` 和 `nsqllookupd` 守护进程被设计成独立运行，没有相互之间的沟通或协调。

我们还认为重要的是有一个方式来聚合查看，监测，并管理集群。我们建立 `nsqadmin` 做到这一点。它提供了一个 Web UI 来浏览 topics/channels/consumers 和深度检查每一层的关键统计数据。此外，它还支持几个管理命令例如，移除通道和清空通道（这是一个有用的工具，当在一个通道中的信息可以被安全地扔掉，以使深度返回到 0）。



Topic: decodes
Channel: cordreader

Empty Queue Delete Channel

Channel Message Queue

Host	Depth	Memory + Disk	In-Flight	Deferred	Requested	Timed Out	Messages	Clients
nsqreplay05.ec2.billy.net:4151	0	0 + 0	0	0	171	0	25832109	4
nsqreplay06.ec2.billy.net:4151	0	0 + 0	2	0	2381	4	277801015	4
nsqreplay07.ec2.billy.net:4151	0	0 + 0	6	0	2484	2	277564277	4
nsqreplay08.ec2.billy.net:4151	0	0 + 0	1	0	2471	3	277507746	4
Total:	0	0 + 0	9	0	7487	9	858485149	16

Client Connections

Client Host	Protocol	NSQd Host	In-Flight	Ready Count	Finished	Requested	Messages	Connected
cordreader04.33724	V2	nsqreplay05.ec2.billy.net:4151	0	11	6522889	45	6522934	18h2m8s
cordreader04.33737	V2	nsqreplay05.ec2.billy.net:4151	0	23	6515747	35	6515782	18h2m7s
cordreader04.35940	V2	nsqreplay08.ec2.billy.net:4151	0	17	47334887	408	47335085	130h58m18s
cordreader04.35943	V2	nsqreplay08.ec2.billy.net:4151	0	21	47343022	396	47343408	130h58m18s
cordreader04.46471	V2	nsqreplay07.ec2.billy.net:4151	2	9	46198180	376	46198558	130h58m18s
cordreader04.46474	V2	nsqreplay07.ec2.billy.net:4151	2	14	46248971	406	46247379	130h58m18s

图片 1.2 nsqadmin

简单的升级路径

这是我们的高优先级之一。我们的生产系统处理大量的流量，都建立在我们现有的消息工具上，所以我们需要一种方法来慢慢地、有条不紊地升级我们特定部分的基础设施，而不产生任何影响。

首先，在消息生产者方面，我们建立 `nsqd` 匹配 `simplequeue`。具体来说，`nsqd` 暴露了一个 HTTP /PUT 端点，就像 `simplequeue`，上传二进制数据（需要注意的一点是 endpoint 需要一个额外的查询参数来指定“话题”）。想切换到发布消息到 `nsqd` 的服务只需要很少的代码变更。

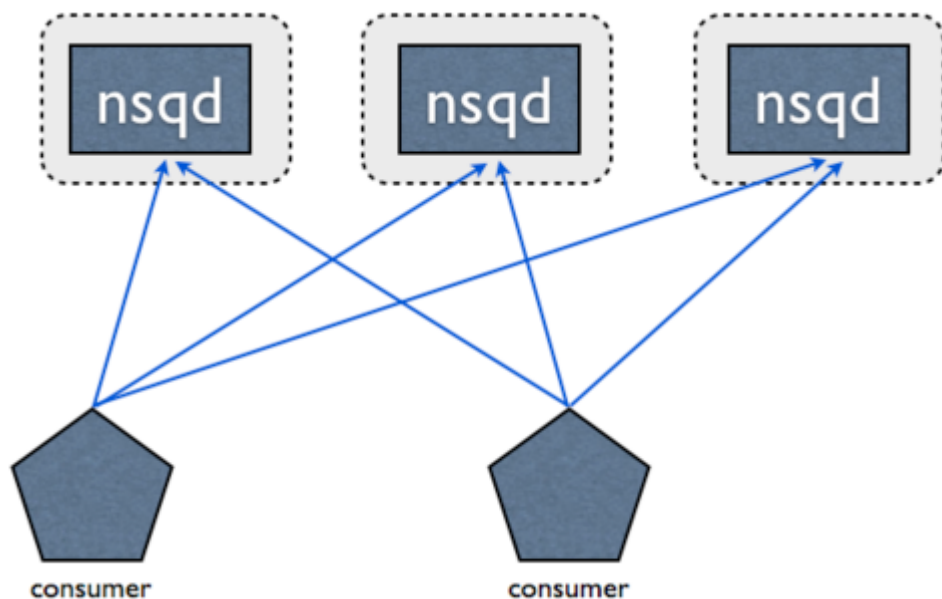
第二，我们建立了兼容已有库功能和语义的 Python 和 Go 库。这使得消息的消费者通过很少的代码改变就可使用。所有的业务逻辑保持不变。

最后，我们建立工具连接起新旧组件。这些都在仓库的示例（`examples`）目录中：

- `nsq_pubsub` – 在 NSQ 集群中以 HTTP 接口的形式暴露的一个 `pubsub`
- `nsq_to_file` – 将一个给定话题的所有消息持久化到文件
- `nsq_to_http` – 对一个话题的所有消息的执行 HTTP 请求到（多个）endpoints。

消除单点故障

NSQ 被设计以分布的方式被使用。`nsqd` 客户端（通过 TCP）连接到指定话题的所有生产者实例。没有中间人，没有消息代理，也没有单点故障：



图片 1.3 nsq clients

这种拓扑结构消除单链，聚合，反馈。相反，你的消费者直接访问所有生产者。从技术上讲，哪个客户端连接到哪个 NSQ 不重要，只要有足够的消费者连接到所有生产者，以满足大量的消息，保证所有东西最终将被处理。

对于 `nsqlookupd`，高可用性是通过运行多个实例来实现。他们不直接相互通信和数据被认为是最终一致。消费者轮询所有的配置的 `nsqlookupd` 实例和合并 response。失败的，无法访问的，或以其他方式故障的节点不会让系统陷于停顿。

消息传递担保

NSQ 保证消息将交付至少一次，虽然消息可能是重复的。消费者应该关注到这一点，删除重复数据或执行 [idempotent](#) 等操作

这个担保是作为协议和工作流的一部分，工作原理如下（假设客户端成功连接并订阅一个话题）：

1. 客户表示他们已经准备好接收消息
2. NSQ 发送一条消息，并暂时将数据存储在本地（在 re-queue 或 timeout）

3. 客户端回复 FIN（结束）或 REQ（重新排队）分别指示成功或失败。如果客户端没有回复, NSQ 会在设定的时间超时, 自动重新排队消息

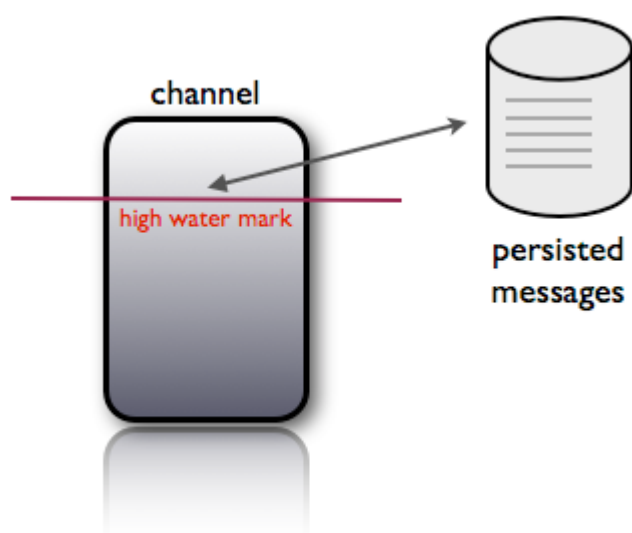
这确保了消息丢失唯一可能的情况是不正常结束 `nsqd` 进程。在这种情况下, 这是在内存中的任何信息（或任何缓冲未刷新到磁盘）都将丢失。

如何防止消息丢失是最重要的, 即使是这个意外情况可以得到缓解。一种解决方案是构成冗余 `nsqd` 对（在不同的主机上）接收消息的相同部分的副本。因为你实现的消费者是幂等的, 以两倍时间处理这些消息不会对下游造成影响, 并使得系统能够承受任何单一节点故障而不会丢失信息。

附加的是 NSQ 提供构建基础以支持多种生产用例和持久化的可配置性。

限定内存占用

`nsqd` 提供一个 `--mem-queue-size` 配置选项, 这将决定一个队列保存在内存中的消息数量。如果队列深度超过此阈值, 消息将透明地写入磁盘。`nsqd` 进程的内存占用被限定于 `--mem-queue-size * #of_channels_and_topics` :



图片 1.4 message overflow

此外, 一个精明的观察者可能会发现, 这是一个方便的方式来获得更高的传递保证: 把这个值设置的比较低（如 1 或甚至是 0）。磁盘支持的队列被设计为在不重启的情况下存在（虽然消息可能被传递两次）。

此外, 涉及到信息传递保证, 干净关机（通过给 `nsqd` 进程发送 TERM 信号）坚持安全地把消息保存在内存中, 传输中, 延迟, 以及内部的各种缓冲区。

请注意，一个以 `#ephemeral` 结束的通道名称不会在超过 `mem-queue-size` 之后刷新到硬盘。这使得消费者并不需要订阅频道的消息担保。这些临时通道将在最后一个客户端断开连接后消失。

效率

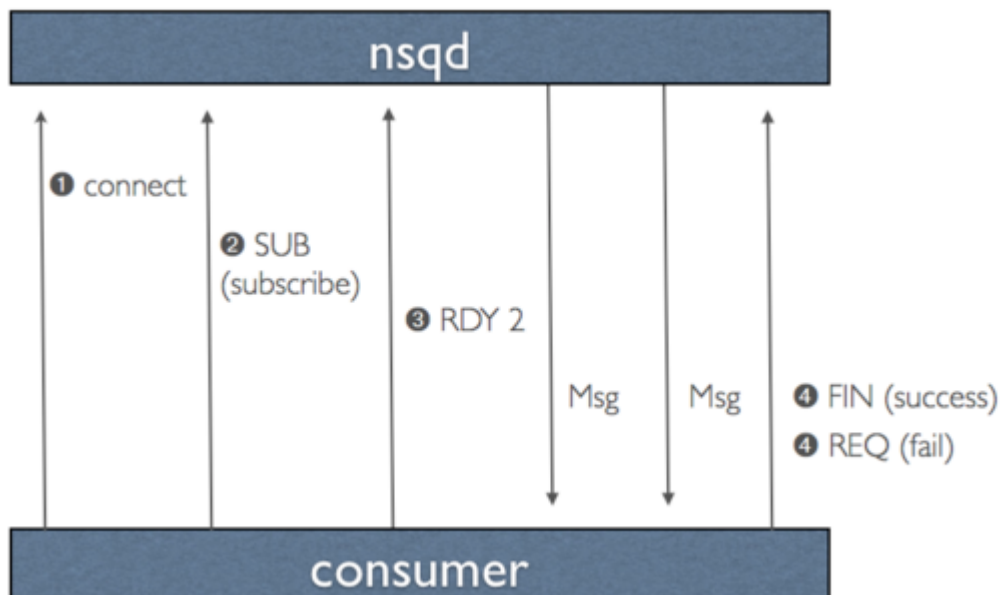
NSQ 被设计成一个使用简单 `size-prefixed` 为前缀的，与“`memcached-like`”类似的命令协议。所有的消息数据被保持在核心中，包括像尝试次数、时间戳等元数据类。这消除了数据从服务器到客户端来回拷贝，当重新排队消息时先前工具链的固有属性。这也简化了客户端，因为他们不再需要负责维护消息的状态。

此外，通过降低配置的复杂性，安装和开发的时间大大缩短（尤其是在有超过 > 1 消费者的话题）。

对于数据的协议，我们做了一个重要的设计决策，通过推送数据到客户端最大限度地提高性能和吞吐量的，而不是等待客户端拉数据。这个概念，我们称之为 `RDY` 状态，基本上是客户端流量控制的一种形式。

当客户端连接到 `nsqd` 和并订阅到一个通道时，它被放置在一个 `RDY` 为 0 状态。这意味着，还没有信息被发送到客户端。当客户端已准备好接收消息发送，更新它的命令 `RDY` 状态到它准备处理的数量，比如 100。无需任何额外的指令，当 100 条消息可用时，将被传递到客户端（服务器端为那个客户端每次递减 `RDY` 计数）。

客户端库的被设计成在 `RDY` 数达到配置 `max-in-flight` 的 25% 发送一个命令来更新 `RDY` 计数（并适当考虑连接到多个 `nsqd` 情况下，适当地分配）。



图片 1.5 nsq protocol

这是一个重要的性能控制，使一些下游系统能够更轻松地批量处理信息，并从更高的 `max-in-flight` 中受益。

值得注意的是，因为它既是基于缓冲和推送来满足需要(通道)流的独立副本的能力，我们已经提供了行为像 `simplequeue` 和 `pubsub` 相结合的守护进程。这是简化我们的系统拓扑结构的强大工具，如上述讨论那样我们会维护传统的 `toolchain`。

Go

我们很早做了一个战略决策，利用 `Go` 来建立 `NSQ` 的核心。我们最近的博客上讲述我们在 `bitly` 如何使用 `Go`，并提到这个适合的项目—通过浏览那篇文章可能对理解我们如何重视这么语言有所帮助。

关于 `NSQ`，`Go channels`（不要与 `NSQ` 通道混淆），并且内置并发性功能的语言的非常适合于的 `nsqd` 的内部工作。我们充分利用缓冲的通道来管理我们在内存中的消息队列和无缝把溢出消息放到硬盘。

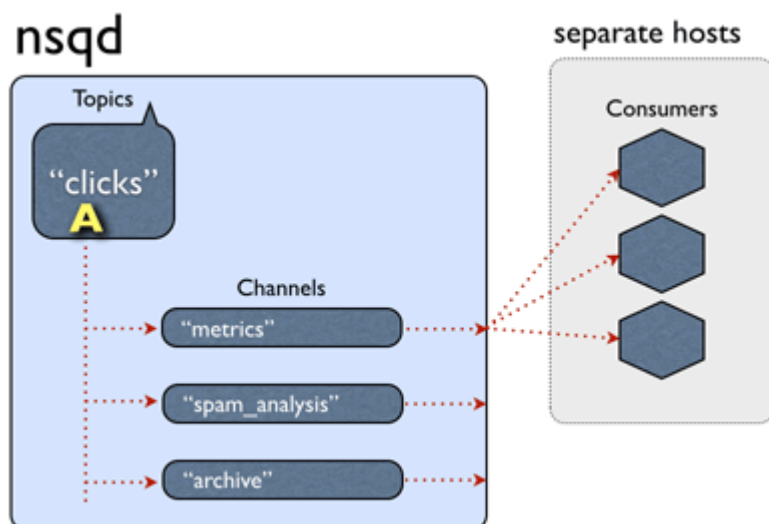
标准库让我们很容易地编写网络层和客户端代码。只需要付出很少的努力，来整合内置的内存和 `CPU` 剖析进行优化。我们还发现它易于单独测试组件，模拟类型接口，以迭代方式构建功能。

内幕

NSQ 由 3 个守护进程组成：

- [nsqd](#) 是接收、队列和传送消息到客户端的守护进程。
- [nsqllookupd](#) 是管理的拓扑信息，并提供了最终一致发现服务的守护进程。
- [nsqadmin](#) 是一个 Web UI 来实时监控集群（和执行各种管理任务）。

在 NSQ 数据流建模为一个消息流和消费者的树。一个话题（topic）是一个独特的数据流。一个通道（channel）是消费者订阅了某个话题的逻辑分组。



图片 1.6 topics/channels

单个 nsqd 可以有很多的话题，每个话题可以有多通道。一个通道接收到一个话题中所有消息的副本，启用组播方式的传输，使消息同时在每个通道的所有订阅用户间分发，从而实现负载均衡。

这些原语组成一个强大的框架，用于表示各种[简单和复杂的拓扑结构](#)。

有关 NSQ 的设计的更多信息请参见[设计文档](#)。

话题和通道

话题（topic）和通道（channel），NSQ 的核心基础，最能说明如何把 Go 语言的特点无缝地转化为系统设计。

Go 语言中的通道（channel）（为消除歧义以下简称为“go-chan”）是实现队列一种自然的方式，因此一个 NSQ 话题（topic）/通道（channel），其核心，只是一个缓冲的 go-chan `Message` 指针。缓冲区的大小等于 `--mem-queue-size` 的配置参数。

在懂了读数据后，发布消息到一个话题（topic）的行为涉及到：

1. 消息结构的初始化（和消息体的内存分配）
2. 获取 话题（topic） 时的读-锁；
3. 是否能发布的读-锁；
4. 发布缓存的 go-chan

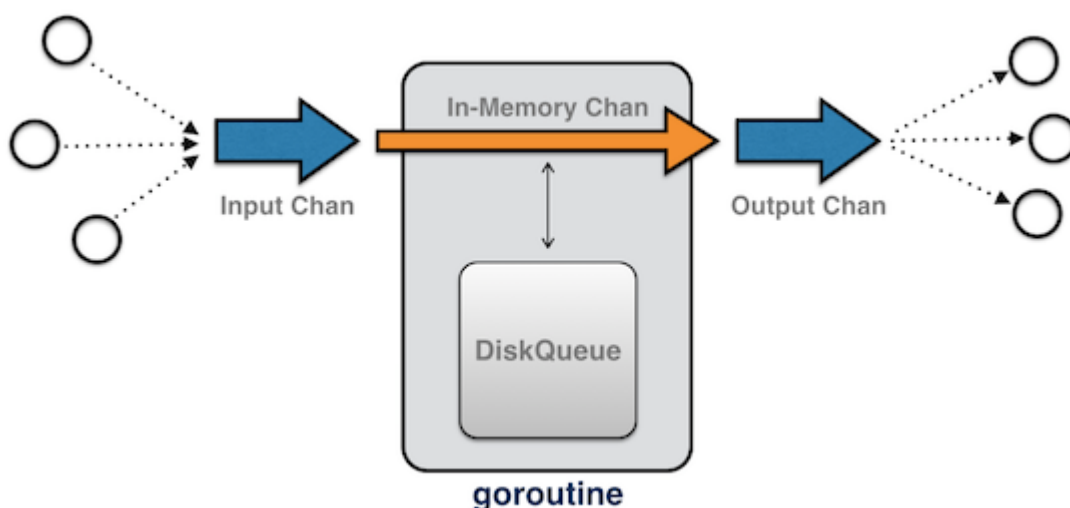
从一个话题中的通道获取消息不能依赖于经典的 go-chan 语义，因为多个 goroutines 在一个 go-chan 上接收消息将会分发消息，而最终要的结果是复制每个消息到每一个通道（goroutine）。

替代的是，每个话题维护着 3 个主要的 goroutines。第一个被称为 `router`，它负责用来从 incoming go-chan 读取最近发布的消息，并把消息保存到队列中（内存或硬盘）。

第二个，称为 `messagePump`，是负责复制和推送消息到如上所述的通道。

第三个是负责 `DiskQueue` IO 和将在后面讨论。

通道是一个有点复杂，但共享着 go-chan 单一输入和输出（抽象出来的事实是，在内部，消息可能会在内存或磁盘上）：



图片 1.7 queue goroutine

此外，每个通道的维护负责 2 个时间排序优先级队列，用来实现传输中（in-flight）消息超时（第 2 个随行 goroutines 用于监视它们）。

并行化的提高是通过每个数据结构管理一个通道，而不是依靠 Go 运行时的全局定时器调度。

注意：在内部，Go 运行时使用一个单一优先级队列和的 goroutine 来管理定时器。这支持（但不局限于）的整个 `time` package。它通常避免了需要一个用户空间的时间顺序的优先级队列，但要意识到这是一个很重要的一个有着单一锁的数据结构，有可能影响 `GOMAXPROCS > 1` 的表现。

Backend / DiskQueue

NSQ 的设计目标之一就是要限定保持在内存中的消息数。它通过 `DiskQueue` 透明地将溢出的消息写入到磁盘上（对于一个话题或通道而言，`DiskQueue` 拥有的第三个主要的 goroutine）。

由于内存队列只是一个 go-chan，把消息放到内存中显得不重要，如果可能的话，则退回到磁盘：

```
for msg := range c.incomingMsgChan {
    select {
    case c.memoryMsgChan <- msg:
    default:
        err := WriteMessageToBackend(&msgBuf, msg, c.backend)
        if err != nil {
            // ... handle errors ...
        }
    }
}
```

说到 Go `select` 语句的优势在于用在短短的几行代码实现这个功能：`default` 语句只在 `memoryMsgChan` 已满的情况下执行。

NSQ 还具有的临时通道的概念。临时的通道将丢弃溢出的消息（而不是写入到磁盘），在没有客户端订阅时消失。这是一个完美的 Go's Interface 案例。话题和通道有一个结构成员声明为一个 `Backend` interface，而不是一个具体的类型。正常的话题和通道使用 `DiskQueue`，而临时通道连接在 `DummyBackendQueue` 中，它实现了一个 no-op 的 `Backend`。

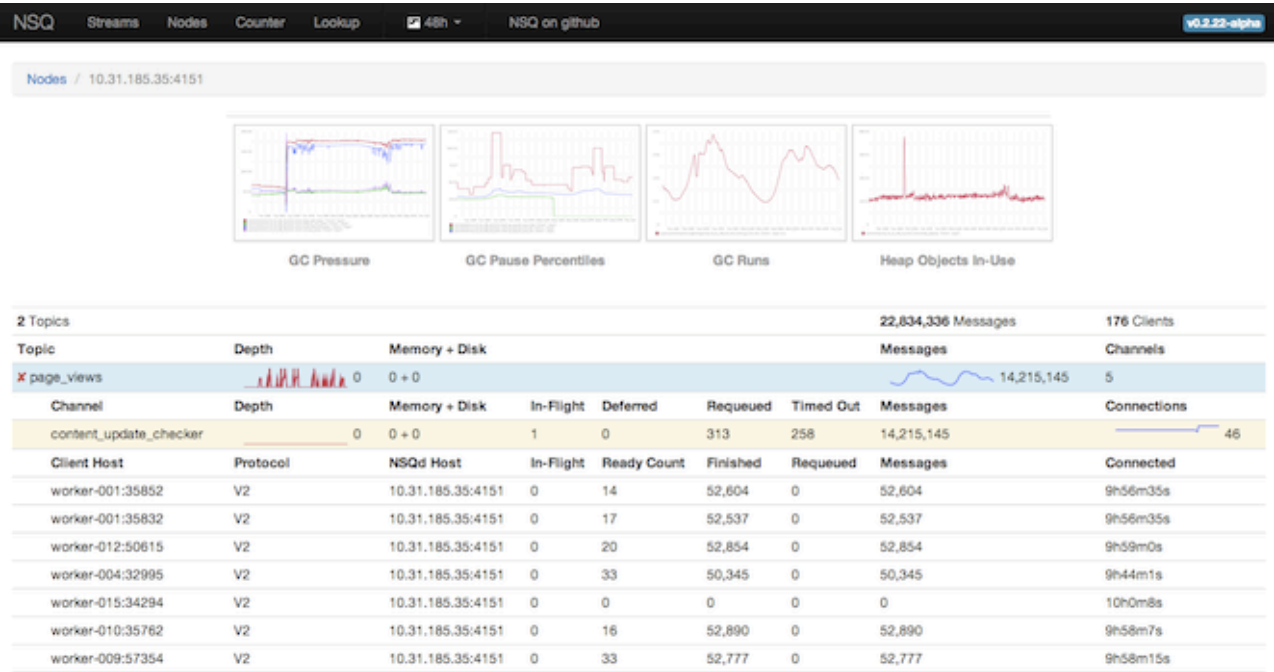
降低 GC 的压力

在任何垃圾回收环境中，你可能会关注到吞吐量（做无用功），延迟（响应），并驻留集大小（footprint）。

Go 的 1.2 版本，GC 采用，mark-and-sweep (parallel), non-generational, non-compacting, stop-the-world 和 mostly precise。这主要是因为剩余的工作未完成（它预定于 Go 1.3 实现）。

Go 的 GC 一定会不断改进，但普遍的真理是：*你创建的垃圾越少，收集的时间越少*。

首先，重要的是要了解 GC 在真实的工作负载下是如何表现。为此，nsqd 以 [statsd](#) 格式发布的 GC 统计（伴随着其他的内部指标）。nsqadmin 显示这些度量的图表，让您洞察 GC 的影响，频率和持续时间：



图片 1.8 single node view

为了切实减少垃圾，你需要知道它是如何生成的。再次 Go toolchain 提供了答案：

1. 使用 `testing` package 和 `go test -benchmem` 来 benchmark 热点代码路径。它分析每个迭代分配的内存数量（和 benchmark 运行可以用 `benchcmp` 进行比较）。
2. 编译时使用 `go build -gcflags -m`，会输出[逃逸分析](#)的结果。

考虑到这一点，下面的优化证明对 nsqd 是有用的：

1. 避免 `[]byte` 到 `string` 的转换
2. buffers 或 object 的重新利用（并且某一天可能面临 [sync.Pool](#) 又名 [issue 4720](#)）
3. 预先分配 slices(在 `make` 时指定容量)并且总是知道其中承载元素的数量和大小
4. 对各种配置项目使用一些明智的限制（例如消息大小）
5. 避免装箱（使用 `interface{}`）或一些不必要的包装类型（例如一个多值的”`go-chan`”结构体）
6. 避免在热点代码路径使用 `defer`（它也消耗内存）

TCP 协议

NSQ 的 TCP 协议 [protocol_spec](#) 是一个这些 GC 优化概念发挥了很大作用的例子。

该协议用含有长度前缀的帧构造，使其可以直接高效的编码和解码：



由于提前知道了帧部件的确切类型与大小，我们避免了 `encoding/binary` 便利 `Read()` 和 `Write()` 包装（以及它们外部 `interface` 的查询与转换），而是直接调用相应的 `binary.BigEndian` 方法。

为了减少 `socket` 的 `IO` 系统调用，客户端 `net.Conn` 都用 `bufio.Reader` 和 `bufio.Writer` 包装。`Reader` 暴露了 `ReadSlice()`，它会重复使用其内部缓冲区。这几乎消除了从 `socket` 读出数据的内存分配，大大降低 `GC` 的压力。这可能是因为与大多数命令关联的数据不会被忽视（在边缘情况下，这是不正确的，数据是显示复制的）。

在一个更低的水平，提供一个 `MessageID` 被声明为 `[16]byte`，以便能够把它作为一个 `map` `key`（`slice` 不能被用作 `map key`）。然而，由于从 `socket` 读取数据存储为 `[]byte`，而不是通过分配字符串键产生垃圾，并避免从 `slice` 的副本拷贝的数组形式的 `MessageID`，`unsafe` package 是用来直接把 `slice` 转换成一个 `MessageID`：

```
id := (*nsq.MessageID)(unsafe.Pointer(&msgID))
```

注：这是一个 *hack*。它将不是必要的，如果编译器优和 [Issue 3512](#) 解决这个问题。另外值得一读通过[issue 5376](#)，其中谈到的“const like” `byte` 类型与 `string` 类型可以互换使用，而不需要分配和复制。

同样，`Go` 标准库只提供了一个数字转换成 `string` 的方法。为了避免 `string` 分配，`nsqd` 使用一个自定义的10进制转换方法在 `[]byte` 直接操作。

这些看似微观优化，但却包含了 `TCP` 协议中一些最热门的代码路径。总体而言，每秒上万消息的速度，对分配和开销的数目显著影响：

benchmark	old ns/op	new ns/op	delta
BenchmarkProtocolV2Data	3575	1963	-45.09%

benchmark	old ns/op	new ns/op	delta
BenchmarkProtocolV2Sub256	57964	14568	-74.87%
BenchmarkProtocolV2Sub512	58212	16193	-72.18%
BenchmarkProtocolV2Sub1k	58549	19490	-66.71%
BenchmarkProtocolV2Sub2k	63430	27840	-56.11%

benchmark	old allocs	new allocs	delta
BenchmarkProtocolV2Sub256	56	39	-30.36%

BenchmarkProtocolV2Sub512	56	39	-30.36%
BenchmarkProtocolV2Sub1k	56	39	-30.36%
BenchmarkProtocolV2Sub2k	58	42	-27.59%

HTTP

NSQ 的 HTTP API 是建立在 Go 的 `net/http` 包之上。因为它只是 `net/http`，它可以利用没有特殊的客户端库的几乎所有现代编程环境。

它的简单性掩盖了它的能力，作为 Go 的 HTTP tool-chest 最有趣的方面之一是广泛的调试功能支持。该 `net/http/pprof` 包直接集成了原生的 HTTP 服务器，暴露获取 CPU，堆，goroutine 和操作系统线程性能的 endpoints。这些可以直接从 `go tool` 找到：

```
$ go tool pprof http://127.0.0.1:4151/debug/pprof/profile
```

这对调试和分析一个运行的进程非常有价值！

此外，`/stats` endpoint 返回的指标以任何 JSON 或良好格式的文本来呈现，很容易使管理员能够实时从命令行监控：

```
$ watch -n 0.5 'curl -s http://127.0.0.1:4151/stats | grep -v connected'
```

这产生的连续输出如下：

```
[page_views ] depth: 0  be-depth: 0  msgs: 105525994 e2e%: 6.6s, 6.2s, 6.2s
 [page_view_counter ] depth: 0  be-depth: 0  inflt: 432 def: 0  re-q: 34684 timeout: 34038 msgs: 105525994 e2e%: 6.6s, 6.2s, 6.2s
 [realtime_score ] depth: 1828 be-depth: 0  inflt: 1368 def: 0  re-q: 25188 timeout: 11336 msgs: 105525994 e2e%: 6.6s, 6.2s, 6.2s
 [variants_writer ] depth: 0  be-depth: 0  inflt: 592 def: 0  re-q: 37068 timeout: 37068 msgs: 105525994 e2e%: 6.6s, 6.2s, 6.2s

[poll_requests ] depth: 0  be-depth: 0  msgs: 11485060 e2e%: 167.5ms, 167.5ms, 138.1ms
 [social_data_collector ] depth: 0  be-depth: 0  inflt: 2  def: 3  re-q: 7568 timeout: 402  msgs: 11485060 e2e%: 167.5ms, 167.5ms, 138.1ms

[social_data ] depth: 0  be-depth: 0  msgs: 60145188 e2e%: 199.0s, 199.0s, 199.0s
 [events_writer ] depth: 0  be-depth: 0  inflt: 226 def: 0  re-q: 32584 timeout: 30542 msgs: 60145188 e2e%: 199.0s, 199.0s, 199.0s
 [social_delta_counter ] depth: 17328 be-depth: 7327 inflt: 179 def: 1  re-q: 155843 timeout: 11514 msgs: 60145188 e2e%: 199.0s, 199.0s, 199.0s

[time_on_site_ticks] depth: 0  be-depth: 0  msgs: 35717814 e2e%: 0.0ns, 0.0ns, 0.0ns
 [tail821042#ephemeral ] depth: 0  be-depth: 0  inflt: 0  def: 0  re-q: 0  timeout: 0  msgs: 33909699 e2e%: 0.0ns, 0.0ns, 0.0ns
```

最后，每个 Go release 版本带来可观的 HTTP 性能提升[autobench](#)。与 Go 的最新版本重新编译时，它总是很高兴为您提供免费的性能提升！

依赖

对于其它生态系统，Go 依赖关系管理（或缺乏）的哲学需要一点时间去适应。

NSQ 从一个单一的巨大仓库衍化而来的，包含相关的 imports 和小到未分离的内部 packages，完全遵守构建和依赖管理的最佳实践。

有两大流派的思想：

1. **Vendoring**: 拷贝正确版本的依赖到你的应用程序的仓库，并修改您的 import 路径来引用本地副本。
2. **Virtual Env**: 列出你在构建时所需要的依赖版本，产生一种原生的 `GOPATH` 环境变量包含这些固定依赖。

注：这确实只适用于二进制包，因为它没有任何意义的一个导入的包，使中间的决定，如一种依赖使用的版本。

NSQ 使用 `gpm` 提供如上述2种的支持。

它的工作原理是在 `Godeps` 文件记录你的依赖，方便日后构建 `GOPATH` 环境。为了编译，它在环境里包装并执行的标准 Go toolchain。该 `Godeps` 文件仅仅是 JSON 格式，可以进行手工编辑。

测试

Go 提供了编写测试和基准测试的内建支持，这使用 Go 很容易并发操作进行建模，这是微不足道的建立起来的一个完整的实例 `nsqd` 到您的测试环境中。

然而，最初实现有可能变成测试问题的一个方面：全局状态。最明显的 offender 是运行时使用该持有 `nsqd` 的引用实例的全局变量，例如包含配置元数据和到 `parent nsqd` 的引用。

某些测试会使用短形式的变量赋值，无意中在局部范围掩盖这个全局变量，即 `nsqd := NewNSQd(...)`。这意味着，全局引用没有指向了当前正在运行的实例，破坏了测试实例。

要解决这个问题，一个包含配置元数据和到 `parent nsqd` 的引用上下文结构被传来传去。到全局状态的所有引用都替换为本地的语境，允许 children（话题（topic），通道（channel），协议处理程序等）来安全地访问这些数据，使之更可靠的测试。

健壮性

一个面对不断变化的网络条件或突发事件不健壮的系统，不会是一个在分布式生产环境中表现良好的系统。

NSQ 设计的方式是使系统能够容忍故障而表现出一致的，可预测的和令人吃惊的方式来实现。

总体理念是快速失败，把错误当作是致命的，并提供了一种方式来调试发生的任何问题。

但是，为了应对，你需要能够检测异常情况。

心跳和超时

NSQ 的 TCP 协议是面向 push 的。在建立连接，握手，和订阅后，消费者被放置在一个为 0 的 `RDY` 状态。当消费者准备好接收消息，它更新的 `RDY` 状态到准备接收消息的数量。NSQ 客户端库不断在幕后管理，消息控制流的结果。

每隔一段时间，`nsqd` 将发送一个心跳线连接。客户端可以配置心跳之间的间隔，但 `nsqd` 会期待一个回应在它发送下一个心跳之前。

组合应用级别的心跳和 `RDY` 状态，避免头阻塞现象，也可能使心跳无用（即，如果消费者是在后面的处理消息流的接收缓冲区中，操作系统将被填满，堵心跳）

为了保证进度，所有的网络 IO 时间上限势必与配置的心跳间隔相关联。这意味着，你可以从字面上拔掉之间的网络连接 `nsqd` 和消费者，它会检测并正确处理错误。

当检测到一个致命错误，客户端连接被强制关闭。在传输中的消息会超时而重新排队等待传递到另一个消费者。最后，错误会被记录并累计到各种内部指标。

管理 Goroutines

非常容易启动 goroutine。不幸的是，不是很容易以协调他们的清理工作。避免死锁也极具挑战性。大多数情况下这可以归结为一个顺序的问题，在上游 goroutine 发送消息到 `go-chan` 之前，另一个 goroutine 从 `go-chan` 上接收消息。

为什么要关心这些？这很显然，孤立的 goroutine 是内存泄漏。内存泄露在长期运行的守护进程中是相当糟糕的，尤其当期望的是你的进程能够稳定运行，但其它都失败了。

更复杂的是，一个典型的 `nsqd` 进程中有许多参与消息传递 goroutines。在内部，消息的“所有权”频繁变化。为了能够完全关闭，统计全部进程内的消息是非常重要的。

虽然目前还没有任何灵丹妙药，下列技术使它变得更轻松管理。

WaitGroups

`sync` 包提供了 `sync.WaitGroup`, 可以被用来累计多少个 goroutine 是活跃的（并且意味着一直等待直到它们退出）。

为了减少典型样板，`nsqd` 使用以下装饰器：

```
type WaitGroupWrapper struct {
    sync.WaitGroup
}

func (w *WaitGroupWrapper) Wrap(cb func()) {
    w.Add(1)
    go func() {
        cb()
        w.Done()
    }()
}

// can be used as follows:
wg := WaitGroupWrapper{}
wg.Wrap(func() { n.idPump() })
...
wg.Wait()
```

退出信号

有一个简单的方式在多个 child goroutine 中触发一个事件是提供一个 `go-chane`，当你准备好时关闭它。所有在那个 `go-chan` 上挂起的 `go-chan` 都将会被激活，而不是向每个 goroutine 中发送一个单独的信号。

```
func work() {
    exitChan := make(chan int)
    go task1(exitChan)
    go task2(exitChan)
    time.Sleep(5 * time.Second)
    close(exitChan)
}

func task1(exitChan chan int) {
    <-exitChan
    log.Printf("task1 exiting")
}

func task2(exitChan chan int) {
    <-exitChan
    log.Printf("task2 exiting")
}
```


退出时的同步

实现一个可靠的，无死锁的，所有传递中的消息的退出路径是相当困难的。一些提示：

1. 理想的情况是负责发送到 go-chan 的 goroutine 中也应负责关闭它。
2. 如果 message 不能丢失，确保相关的 go-chan 被清空（尤其是无缓冲的！），以保证发送者可以取得进展。
3. 另外，如果消息是不重要的，发送给一个单一的 go-chan 应转换为一个 `select` 附加一个退出信号（如上所述），以保证取得进展。
4. 一般的顺序应该是
 1. 停止接受新的连接（close listeners）
 2. 发送退出信号给 child goroutines（如上文）
 3. 在 `WaitGroup` 等待 goroutine 退出（如上文）
 4. 恢复缓冲数据
 5. 刷新所有东西到硬盘

日志

最后，日志是您所获得的记录 goroutine 进入和退出的重要工具！。这使得它相当容易识别造成死锁或泄漏的情况的罪魁祸首。

nsqd 日志行包括 goroutine 与他们的 siblings (and parent) 的信息，如客户端的远程地址或话题（topic）/通道（channel）名。

该日志是详细的，但不是详细的日志是压倒性的。有一条细线，但 nsqd 倾向于发生故障时在日志中提供更多的信息，而不是试图减少繁琐的有效性为代价。



组件



nsqd

`nsqd` 是一个守护进程，负责接收，排队，投递消息给客户端。

它可以独立运行，不过通常它是由 `nsqlookupd` 实例所在集群配置的（它在这能声明 topics 和 channels，以便大家能找到）。

它在 2 个 TCP 端口监听，一个给客户端，另一个是 HTTP API。同时，它也能在第三个端口监听 HTTPS。

命令行选项

```
-auth-http-address=: <addr>:<port> 查询授权服务器 (可能会给多次)
-broadcast-address="": 通过 lookupd 注册的地址 (默认名是 OS)
-config="": 配置文件路径
-data-path="": 缓存消息的磁盘路径
-deflate=true: 运行协商压缩特性 (客户端压缩)
-e2e-processing-latency-percentile=: 消息处理时间的百分比 (通过逗号可以多次指定, 默认为 none)
-e2e-processing-latency-window-time=10m0s: 计算这段时间里, 点对点时间延迟 (例如, 60s 仅计算过去 60 秒)
-http-address="0.0.0.0:4151": 为 HTTP 客户端监听 <addr>:<port>
-https-address="": 为 HTTPS 客户端 监听 <addr>:<port>
-lookupd-tcp-address=: 解析 TCP 地址名字 (可能会给多次)
-max-body-size=5123840: 单个命令体的最大尺寸
-max-bytes-per-file=104857600: 每个磁盘队列文件的字节数
-max-deflate-level=6: 最大的压缩比率等级 (> values == > nsqd CPU usage)
-max-heartbeat-interval=1m0s: 在客户端心跳间, 最大的客户端配置时间间隔
-max-message-size=1024768: (弃用 --max-msg-size) 单个消息体的最大字节数
-max-msg-size=1024768: 单个消息体的最大字节数
-max-msg-timeout=15m0s: 消息超时的最大时间间隔
-max-output-buffer-size=65536: 最大客户端输出缓存可配置大小(字节)
-max-output-buffer-timeout=1s: 在 flushing 到客户端前, 最长的配置时间间隔。
-max-rdy-count=2500: 客户端最大的 RDY 数量
-max-req-timeout=1h0m0s: 消息重新排队的超时时间
-mem-queue-size=10000: 内存里的消息数(per topic/channel)
-msg-timeout="60s": 自动重新队列消息前需要等待的时间
-snappy=true: 打开快速选项 (客户端压缩)
-statsd-address="": 统计进程的 UDP <addr>:<port>
-statsd-interval="60s": 从推送到统计的时间间隔
-statsd-mem-stats=true: 切换发送内存和 GC 统计数据
-statsd-prefix="nsq.%s": 发送给统计keys 的前缀(%s for host replacement)
-sync-every=2500: 磁盘队列 fsync 的消息数
-sync-timeout=2s: 每个磁盘队列 fsync 平均耗时
```

```

-tcp-address="0.0.0.0:4150": TCP 客户端 监听的 <addr>:<port>
-tls-cert="": 证书文件路径
-tls-client-auth-policy="": 客户端证书授权策略 ('require' or 'require-verify')
-tls-key="": 私钥路径文件
-tls-required=false: 客户端连接需求 TLS
-tls-root-ca-file="": 私钥证书授权 PEM 路径
-verbose=false: 打开日志
-version=false: 打印版本
-worker-id=0: 进程的唯一码(默认是主机名的哈希值)

```

HTTP API

- [/ping \(页 0\)](#) – 活跃度
- [/info \(页 0\)](#) – 版本
- [/stats \(页 0\)](#) – 检查综合运行
- [/pub \(页 0\)](#) – 发布消息到话题 (topic)
- [/mpub \(页 0\)](#) – 发布多个消息到话题 (topic)
- [/debug/pprof \(页 0\)](#) – pprof 调试入口
- [/debug/pprof/profile \(页 0\)](#) – 生成 pprof CPU 配置文件
- [/debug/pprof/goroutine \(页 0\)](#) – 生成 pprof 计算配置文件
- [/debug/pprof/heap \(页 0\)](#) – 生成 pprof 堆配置文件
- [/debug/pprof/block \(页 0\)](#) – 生成 pprof 块配置文件
- [/debug/pprof/threadcreate \(页 0\)](#) – 生成 pprof OS 线程配置文件

v1 命名空间 (as of nsqd v0.2.29+):

- [/topic/create \(页 0\)](#) – 创建一个新的话题 (topic)
- [/topic/delete \(页 0\)](#) – 删除一个话题 (topic)
- [/topic/empty \(页 0\)](#) – 清空话题 (topic)
- [/topic/pause \(页 0\)](#) – 暂停话题 (topic)的消息流
- [/topic/unpause \(页 0\)](#) – 恢复话题 (topic)的消息流
- [/channel/create \(页 0\)](#) – 创建一个新的通道 (channel)
- [/channel/delete \(页 0\)](#) – 删除一个通道 (channel)
- [/channel/empty \(页 0\)](#) – 清空一个通道 (channel)

- `/channel/pause` (页 0) – 暂停通道 (channel) 的消息流
- `/channel/unpause` (页 0) – 恢复通道 (channel) 的消息流

以抛弃的命名空间:

- `/create_topic` (页 0) – 创建一个新的话题 (topic)
- `/delete_topic` (页 0) – 删除一个话题 (topic)
- `/empty_topic` (页 0) – 清空话题 (topic)
- `/pause_topic` (页 0) – 暂停话题 (topic) 的消息流
- `/unpause_topic` (页 0) – 恢复话题 (topic) 的消息流
- `/create_channel` (页 0) – 创建一个新的通道 (channel)
- `/delete_channel` (页 0) – 删除一个通道 (channel)
- `/empty_channel` (页 0) – 清空一个通道 (channel)
- `/pause_channel` (页 0) – 暂停通道 (channel) 的消息流
- `/unpause_channel` (页 0) – 恢复通道 (channel) 的消息流

NOTE: 这些结束点返回 "wrapped" JSON:

```
{"status_code":200, "status_text":"OK", "data":{...}}
```

发送 `Accept: application/vnd.nsq; version=1.0` 头将会协商使用未封装的 JSON 响应格式 (as of `nsqd v0.2.29+`)。

`/pub`

发布一个消息

参数:

topic – the topic to publish to

POST body – the raw message bytes

```
$ curl -d "<message>" http://127.0.0.1:4151/pub?topic=message_topic`
```

`/mpub`

一个往返发布多个消息

参数:

topic – 发布到的话题 (topic)
 binary – bool ('true' or 'false') 允许二进制模式

POST body – '\n' 分离原始消息字节

注意：默认的 /mpub 希望消息使用 \n 切割，使用 ?binary=true 查询参数来允许二进制模式，希望发送的消息体能成为以下的格式（HTTP 'Content-Length' 头必须是要发送的消息体的总大小）：

```
[ 4-byte num messages ]
[ 4-byte message #1 size ][ N-byte binary data ]
... (repeated <num_messages> times)
```

```
$ curl -d "<message>\n<message>\n<message>" http://127.0.0.1:4151/mpub?topic=message_topic`
```

/topic/create

已经抛弃的别名 /create_topic

创建一个话题 (topic)

参数:

话题 (topic) – 将要创建的话题 (topic)

/topic/delete

已经抛弃的别名: /delete_topic

删除一个已经存在的话题 (topic) (和所有的通道 (channel))

参数:

topic – 现有的话题 (topic) to delete

/channel/create

已抛弃的别名: /create_channel

为现有的话题 (topic) 创建一个通道 (channel)

参数:

topic – 现有的话题 (topic)
 channel – the channel to create

/channel/delete

已抛弃的别名: /delete_channel

删除现有的话题 (topic) 一个的通道 (channel)

参数:

topic – 现有的话题 (topic)
channel – 待删除的通道 (channel)

/topic/empty

已抛弃的别名: /empty_topic

清空现有话题 (topic) 队列中所有的消息 (内存和磁盘中)

参数:

topic – 待清空的话题 (topic)

/channel/empty

已抛弃的别名: /empty_channel

清空现有通道 (channel) 队列中所有的消息 (内存和磁盘中)

参数:

topic – 现有的话题 (topic)
channel – 待清空的通道 (channel)

/topic/pause

已抛弃的别名: /pause_topic

暂停已有话题 (topic) 的所有通道 (channel) 的消息 (消息将会在话题 (topic) 里排队)

参数:

topic – 现有的话题 (topic)

/topic/unpause

已抛弃的别名: /unpause_topic

为现有的话题 (topic) 的通道 (channel) 重启消息流

参数:

topic – 现有的话题 (topic)

/channel/pause

已抛弃的别名: /channel_pause

暂停发送已有的通道 (channel) 给消费者 (消息将会队列)

参数:

topic – 现有的话题 (topic)
channel – 已有的通道 (channel) 将会被暂停

/channel/unpause

已抛弃的别名: /unpause_channel

重新发送通道 (channel) 里的消息给消费者

参数:

topic – 现有的话题 (topic)
channel – 将要暂停的通道 (channel)

/stats

返回内部统计数据

参数

format – (可选) `text` or `json` (默认 = `text`)

/ping

监控结束点, 必须返回 `OK`。如果有问题返回 500。同时, 如果写消息到磁盘失败将会返回错误状态。

/info

返回版本信息

/debug/pprof

可用的调节点码的页码

/debug/pprof/profile

开始 30 秒的 `pprof` CPU 配置, 并通过请求返回。

注意, 因为它在运行时的性能和时间, 这个结束点并没有在 `/debug/pprof` 页面列表中。

/debug/pprof/goroutine

为所有运行的 goroutines 返回栈记录。

/debug/pprof/heap

返回堆和内存配置信息 (前面的内容可作为 `pprof` 配置信息)

/debug/pprof/block

返回 goroutine 块配置信息

/debug/pprof/threadcreate

返回 goroutine 栈记录

Debugging and Profiling

`nsqd` 提供一套节点的配置信息，直接通过 Go 的 `pprof` 工具。如果你有 go 工具套装，只要运行：

```
# memory profiling

$ go 工具 pprof http://localhost:4151/debug/pprof/heap

# cpu profiling

$ go 工具 pprof http://localhost:4151/debug/pprof/profile
```

TLS

为了加强安全性，可以通过 `--tls-cert` 和 `--tls-key` 客户端配置 `nsqd`，升级他们的链接为 TLS。

另外，你可以要求客户端使用 `--tls-required` (`nsqd` v0.2.28+) 协商 TLS。

你可以通过 `--tls-client-auth-policy` (`require` 或 `require-verify`) 配置一个 `nsqd` 客户端证书：

- `require` - 客户端必须提供一个证书，否则将会被拒绝
- `require-verify` - 客户端必须提供一个有效的证书，根据 `--tls-root-ca-file` 指定的链接或者默认的 CA，否则将会被拒绝。

可以当做客户端授权的表单 (`nsqd` v0.2.28+)。

如果你想生成一个 password-less，自签名证书，用：

```
$ openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

AUTH

注意：在 `nsqd` v0.2.29+ 可用

通过使用一个遵从 Auth HTTP 协议的授权服务器，指定 `-auth-http-address=host:port` 标志，你可以配置 `nsqd`。

注意：希望当仅有 `nsqd` TCP 协议暴露给外部客户端时使用授权，而不是 HTTP(S) 节点。参见底下说明：

Auth 服务器必须接受 HTTP 请求：

```
/auth?remote_ip=...&tls=...&auth_secret=...
```

返回结果格式如下：

```
{
  "ttl": 3600,
  "identity": "username",
  "identity_url": "http://....",
  "authorizations": [
    {
      "permissions": [
        "subscribe",
        "publish"
      ],
      "topic": ".*",
      "channels": [
        ".*"
      ]
    }
  ]
}
```

注意话题（topic）和通道（channel）字符串必须用 `nsqd` 的正则表达式来申请授权。`nsqd` 将会为 TTL 间隔，并会在这个间隔时间里重新请求授权。

通常情况，将会使用 TLS 来加强安全性。`nsqd` 和 授权服务器间通过信任的网络通信（并没被加密）。如果一个授权服务器通过远程 IP 信息来授权，客户端可以使用占位符（比如 `.`），作为 `AUTH` 命令（Auth 服务器忽略）。

授权服务器例子 [pynsqauthd](#)。

帮助服务器暴露 `nsqlookupd` 和 `nsqd` `/stats` 数据给客户端，从授权服务器通过权限过滤，在以下可以找到 [nsqauthfilter](#)。

当使用命令行工具，可以通过使用 `--reader-opt` 标志来授权。

```
$ nsq_tail ... -reader-opt="tls_v1,true" -reader-opt="auth_secret,$SECRET"
```

点对点处理延迟

你可以选择设置 `nsqd` 来收集和发射点对点信息处理延迟，通过 `--e2e-processing-latency-percentile` 标志位来配置百分比。

使用概率百分比技术（参见 [Effective Computation of Biased Quantiles over Data Streams](#)）来计算值。我们通过 [bmizerany](#) 来使用 [perks](#) 包，它能实现这个算法。

我们内部维持 2 个通道（channel），每个通道（channel）存储 $N/2$ 分钟的延迟数据。每个 $N/2$ 分钟我们重置了每个通道（channel）（并开始插入新的数据）。

因为我们仅在通道级别收集数据，对于话题我们聚合并合并所有的通道数量的 quantiles。如果数据在同一个 `nsqd` 实例上时，可以使用这个技术。然而当数据已经精确的通过 `nsqd`（通过 `nsqllookupd`），我们为每个 `nsqd` 取平均值。为了维持统计的精确性，除了平均值，我们也提供最大最小值。

注意: 如果没有消费者连接，不能更新值，尽管消息队列的点对点时间会缓慢增长。这是因为仅在 `nsqd` 收到从客户端发来 `FIN` 消息时才会重新计算。当消费者重新连接，这些值将会重新调整。

Statsd / Graphite Integration

当使用 `--statsd-address` 来为 `statsd`（或类似 `statsdaemon`）指定 UDP `<addr>:<port>` 时，`nsqd` 将会在 `--statsd-interval` 定期推送数据给 `statsd`（注意：这个间隔必须始终小于等于 graphite 的刷入间隔）。设置 `nsqadmin` 可以显示图标。

推荐以下配置（但是这些选择必须建立在你的可用资源和要求上）。同时，`statsd` 的刷入间隔必须小于或者等于 `storage-schemas.conf` 的最小值，并且 `nsqd` 必须通过 `--statsd-interval` 来确认刷入时间小于等于时间间隔。

```
# storage-schemas.conf

[nsq]
pattern = ^nsq\..*
retentions = 1m:1d,5m:30d,15m:1y

# storage-aggregation.conf

[default_nsq]
pattern = ^nsq\..*
xFilesFactor = 0.2
aggregationMethod = average
```

`nsqd` 实例将会推送给以下 `statsd` 路径：

```
nsq.<nsqd_host>.<nsqd_port>.topic.<topic_name>.backend_depth [gauge]
nsq.<nsqd_host>.<nsqd_port>.topic.<topic_name>.depth [gauge]
nsq.<nsqd_host>.<nsqd_port>.topic.<topic_name>.message_count
nsq.<nsqd_host>.<nsqd_port>.topic.<topic_name>.channel.<channel_name>.backend_depth [gauge]
```

```

nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.clients [gauge]
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.deferred_count [gauge]
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.depth [gauge]
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.in_flight_count [gauge]
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.message_count
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.requeue_count
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.timeout_count

```

```

# if --statsd-mem-stats is enabled

```

```

nsq.<nsqd_host>_<nsqd_port>.mem.heap_objects [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.heap_idle_bytes [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.heap_in_use_bytes [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.heap_released_bytes [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.gc_pause_usec_100 [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.gc_pause_usec_99 [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.gc_pause_usec_95 [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.mem.next_gc_bytes [gauge]
nsq.<nsqd_host>_<nsqd_port>.mem.gc_runs

```

```

# if --e2e-processing-latency-percentile is specified, for each percentile

```

```

nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.e2e_processing_latency_<percent> [gauge]
nsq.<nsqd_host>_<nsqd_port>.topic.<topic_name>.channel.<channel_name>.e2e_processing_latency_<percent> [ga

```

nsqlookupd

`nsqlookupd` 是守护进程负责管理拓扑信息。客户端通过查询 `nsqlookupd` 来发现指定话题（topic）的生产者，并且 `nsqd` 节点广播话题（topic）和通道（channel）信息。

有两个接口：TCP 接口，`nsqd` 用它来广播。HTTP 接口，客户端用它来发现和管理。

命令行选项

```
-http-address="0.0.0.0:4161": <addr>:<port> 监听 HTTP 客户端
-inactive-producer-timeout=5m0s: 从上次 ping 之后，生产者驻留在活跃列表中的时长
-tcp-address="0.0.0.0:4160": TCP 客户端监听的 <addr>:<port>
-broadcast-address: 这个 lookupd 节点的外部地址, (默认是 OS 主机名)
-tombstone-lifetime=45s: 生产者保持 tombstoned 的时长
-verbose=false: 允许输出日志
-version=false: 打印版本信息
```

HTTP 接口

`/lookup`

返回某个话题（topic）的生产者列表。

参数:

```
topic – the 话题（topic） to list producers for
```

`/topics`

返回所有已知的话题（topic）

`/channels`

返回已知话题（topic）里的通道（channel）

参数:

```
topic – the topic to list 通道（channel）s for
```

`/nodes`

返回所有已知的 `nsqd` 列表

`/delete_topic`

删除一个已存在的话题（topic）

参数:

topic – 需要删除的话题 (topic)

`/delete_channel`

删除一个已存在话题 (topic) 的通道 (channel)

参数:

topic – 已经存在的话题 (topic)

channel – 将要删除的通道 (channel)

`/tombstone_topic_producer`

逻辑删除 (Tombstones) 某个话题 (topic) 的生产者。参见 [deletion and tombstones \(页 0\)](#)。

参数:

topic – 已经存在的话题 (topic)

node – 将要逻辑删除 (tombstones) 的生产者(nsqd) (通过 <broadcast_address>:<http_port> 识别)

`/ping`

监控端点, 必须返回 `OK`

`/info`

返回版本信息

删除和逻辑删除 (Tombstones)

当一个话题 (topic) 不再全局生产, 相对简单的操作是从集群里清理这个消息。假设所有的应用生产的消息下降, 使用 `/delete_topic` 结束 `nsqlookupd` 实例的, 是必须要完成的操作。(内部来说, 它将会识别 `nsqd` 生产者, 并对这些节点执行合适的操作)。

全局来看, 通道 (channel) 删除进程都很类似, 不同点是你需用 `/delete_channel` 结束 `nsqlookupd` 实例, 并且你必须保证所有的订阅了通道 (channel) 的消费者已经下降 (downed)。

然而, 当话题 (topic) 不再在节点的子集上生产的时候情况比较复杂。因为消费者查询 `nsqlookupd` 的方法并且连接到所有生产者, 你加入的竞争环境尝试移除集群的信息, 消费者发现这些节点并重新连接。(因此推送更新, 话题 (topic) 仍然在节点上生产)。解决办法就是逻辑删除 (tombstones)。逻辑删除 (tombstones) 在 `nsqlookupd` 上下文是特定的生产者和最后的配置 `--tombstone-lifetime` 时间。在这个窗口中, 生产者不会在 `/lookup` 查询中列出, 允许节点删除话题 (topic), 扩散这些信息到 `nsqlookupd` (接着逻辑删除 (tombstones) 生产者), 并阻止生产者重新发现这个节点。

nsqadmin

`nsqadmin` 是一套 WEB UI，用来汇集集群的实时统计，并执行不同的管理任务。

命令行选项

```
-graphite-url="": URL to graphite HTTP 地址
-http-address="0.0.0.0:4171": <addr>:<port> HTTP clients 监听的地址和端口
-lookupd-http-address=[]: lookupd HTTP 地址 (可能会提供多次)
-notification-http-endpoint="": HTTP 端点 (完全限定)，管理动作将会发送到
-nsqd-http-address=[]: nsqd HTTP 地址 (可能会提供多次)
-proxy-graphite=false: Proxy HTTP requests to graphite
-template-dir="": 临时目录路径
-use-statsd-prefixes=true: expect statsd prefixed keys in graphite (ie: 'stats_counts.')
-version=false: 打印版本信息
```

statsd / Graphite Integration

使用 `nsqd --statsd-address=...` 的时候，你可以指定一个 `nsqadmin --graphite-url=http://graphite.yourdomain.com` 允许 `nsqadmin` 上的 graphite 图表。如果使用一个统计克隆 (例如 [statsdaemon](#))，它没有前缀的键值，也可以指定 `--use-statsd-prefix=false`。

Admin 通知

如果设置了 `--notification-http-endpoint` 标志，每次 admin 动作执行的时候（例如暂停一个通道（channel）），`nsqadmin` 将会发送一个 POST 请求到指定(完全限定)端点。

请求的内容包含的动作信息，例如：

```
{
  "action": "unpause_channel",
  "channel": "mouth",
  "topic": "beer",
  "timestamp": 1357683731,
  "user": "df",
  "user_agent": "Mozilla/5.0 (Macintosh; Iphone 8)"
  "remote_ip": "1.2.3.4:5678"
}
```

如果在请求时用户名可用，`user` 字段将会填充，如果之前使用 `htpasswd` 授权，或者 [google-auth-proxy](#) 之后，否则为空字符串。当不可用时 `channel` 字段也会为空。

提示: 通过设置 `--notification-http-endpoint` 为 `http://addr.of.nsqd/put?topic=admin_actions`，你可以创建一个 admin 的动作通知 NSQ 流，话题（topic）名为 `admin_actions`。

工具

这些工具辅助到数据流的通用功能和内部检查。

nsq_stat

为所有的话题（topic）和通道（channel）的生产者轮询 `/stats`，并显示统计数据：

```
-----depth-----+-----metadata-----
total mem disk inflt def | req t-o      msgs clients
24660 24660   0  0  20 | 102688   0 132492418   1
25001 25001   0  0  20 | 102688   0 132493086   1
21132 21132   0  0  21 | 102688   0 132493729   1
```

命令行参数

```
-channel="": NSQ 通道 (channel)
-lookupd-http-address=: lookupd HTTP 地址 (可能会给多次)
-nsqd-http-address=: nsqd HTTP 地址 (可能会给多次)
-status-every=2s: 轮询/打印输出见的时间间隔
-topic="": NSQ 话题 (topic)
-version=false: 打印版本
```

nsq_tail

消费指定的话题（topic）/通道（channel），并写到 stdout (和 tail(1) 类似)。

命令行参数

```
-channel="": NSQ 通道 (channel)
-consumer-opt=: 传递给 nsq.Consumer (可能会给多次, http://godoc.org/github.com/bitly/go-nsq#Config)
-lookupd-http-address=: lookupd HTTP 地址 (可能会给多次)
-max-in-flight=200: 最大的消息数 to allow in flight
-n=0: total messages to show (will wait if starved)
-nsqd-tcp-address=: nsqd TCP 地址 (可能会给多次)
-reader-opt=: (已经抛弃) 使用 --consumer-opt
```

```
-topic="": NSQ 话题 (topic)
-version=false: 打印版本信息
```

nsq_to_file

消费指定的话题 (topic) / 通道 (channel)，并写到文件中，有选择的滚动和/或压缩文件。

命令行参数

```
-channel="nsq_to_file": nsq 通道 (channel)
-consumer-opt=: 传递给 nsq.Consumer 的参数 (可能会给多次, http://godoc.org/github.com/bitly/go-nsq#Config)
-datetime-format="%Y-%m-%d_%H": strftime, 和 filename 里 <DATETIME> 格式兼容
-filename-format="<TOPIC>.<HOST><GZIPREV>.<DATETIME>.log": output 文件名格式 (<TOPIC>, <HOST>, <DATETIME>)
-gzip=false: gzip 输出文件
-gzip-compression=3: (已经抛弃) 使用 --gzip-level, gzip 压缩级别(1 = 速度最佳, 2 = 最近压缩, 3 = 默认压缩)
-gzip-level=6: gzip 压缩级别 (1-9, 1=BestSpeed, 9=BestCompression)
-host-identifier="": 输出到 log 文件, 提供主机名。<SHORT_HOST> 和 <HOSTNAME> 是有效的替换者
-lookupd-http-address=: lookupd HTTP 地址 (可能会给多次)
-max-in-flight=200: 最大的消息数 to allow in flight
-nsqd-tcp-address=: nsqd TCP 地址 (可能会给多次)
-output-dir="/tmp": 输出文件所在的文件夹
-reader-opt=: (已经抛弃) 使用 --consumer-opt
-skip-empty-files=false: 忽略写空文件
-topic=: nsq 话题 (topic) (可能会给多次)
-topic-refresh=1m0s: 话题 (topic) 列表刷新的频率是多少?
-version=false: 打印版本信息
```

nsq_to_http

消费指定的话题 (topic) / 通道 (channel) 和执行 HTTP requests (GET/POST) 到指定的端点。

命令行参数

```
-channel="nsq_to_http": nsq 通道 (channel)
-consumer-opt=: 参数, 通过 nsq.Consumer (可能会给多次, http://godoc.org/github.com/bitly/go-nsq#Config)
-content-type="application/octet-stream": the Content-Type 使用d for POST requests
-get=: HTTP 地址 to make a GET request to. '%s' will be printf replaced with data (可能会给多次)
-http-timeout=20s: timeout for HTTP connect/read/write (each)
-http-timeout-ms=20000: (已经抛弃) 使用 --http-timeout=X, timeout for HTTP connect/read/write (each)
```

```

-lookupd-http-address=: lookupd HTTP 地址 (可能会给多次)
-max-backoff-duration=2m0s: (已经抛弃) 使用 --consumer-opt=max_backoff_duration,X
-max-in-flight=200: 最大的消息数 to allow in flight
-mode="round-robin": the upstream request mode options: multicast, round-robin, hostpool
-n=100: number of concurrent publishers
-nsqd-tcp-address=: nsqd TCP 地址 (可能会给多次)
-post=: HTTP 地址 to make a POST request to. data will be in the body (可能会给多次)
-reader-opt=: (已经抛弃) 使用 --consumer-opt
-round-robin=false: (已经抛弃) 使用 --mode=round-robin, enable round robin mode
-sample=1: % of messages to publish (float b/w 0 -> 1)
-status-every=250: the # of requests between logging status (per handler), 0 disables
-throttle-fraction=1: (已经抛弃) 使用 --sample=X, publish only a fraction of messages
-topic="": nsq 话题 ( topic )
-version=false: 打印版本信息

```

nsq_to_nsq

消费者指定的话题/通道和重发消息到目的地 `nsqd` 通过 TCP。

命令行参数

```

-channel="nsq_to_nsq": nsq 通道 ( channel )
-consumer-opt=: 参数, 通过 nsq.Consumer (可能会给多次, see http://godoc.org/github.com/bitly/go-nsq#Config)
-destination-nsqd-tcp-address=: destination nsqd TCP 地址 (可能会给多次)
-destination-topic="": destination nsq 话题 ( topic )
-lookupd-http-address=: lookupd HTTP 地址 (可能会给多次)
-max-backoff-duration=2m0s: (已经抛弃) 使用 --consumer-opt=max_backoff_duration,X
-max-in-flight=200: 允许 flight 最大的消息数
-mode="round-robin": 上行请求的参数: round-robin (默认), hostpool
-nsqd-tcp-address=: nsqd TCP 地址 (可能会给多次)
-producer-opt=: 传递到 nsq.Producer (可能会给多次, 参见 http://godoc.org/github.com/bitly/go-nsq#Config)
-reader-opt=: (已经抛弃) 使用 --consumer-opt
-require-json-field="": JSON 消息: 仅传递消息, 包含这个参数
-require-json-value="": JSON 消息: 仅传递消息要求参数有这个值
-status-every=250: # 请求日志的状态(每个目的地), 0 不可用
-topic="": nsq 话题 ( topic )
-version=false: 打印版本信息
-whitelist-json-field=: JSON 消息: 传递这个字段 (可能会给多次)

```

to_nsq

采用 stdin 流，并分解到新行（默认），通过 TCP 重新发布到目的地 nsqd。

命令行参数

```
-delimiter="\n": 分割字符串(默认'\n')  
-nsqd-tcp-address=: 目的地 nsqd TCP 地址 (可能会给多次)  
-producer-opt=: 参数，通过 nsq.Producer (可能会给多次, http://godoc.org/github.com/bitly/go-nsq#Config)  
-topic="": 发布到的 NSQ 话题 (topic)
```



客户端



TCP 协议规范

NSQ 协议足够简单，用任何语言编译客户端都很容易。我们提供官方的 Go 和 Python 客户端库。

`nsqd` 进程通过监听配置的 TCP 端口来接受客户端连接。

连接后，客户端必须发送一个 4 字节的 "magic" 标识码，表示通讯协议的版本。

- V2 (4 个字节的 ASCII `[space][space][V][2]`) 消费用到的推送流协议 (和发布用到的请求/响应协议)

认证后，客户端可以发送 `IDENTIFY` 命令来提供常用的元数据 (比如，更多的描述标识码) 和协商特性。为了消费消息，客户端必须 `SUB` 到一个通道 (channel)。

订阅的时候，客户端的 `RDY` 状态为 0。意味着没有消息会被发送到客户端。当客户端已经准备好接受消息时，需要把 `RDY` 设置为 #。比如设置为 100，不需要任何附加命令，将会有 100 条消息推送到客户端 (每次服务端都会相应的减少 `RDY` 的值)。

V2 版本的协议让客户端拥有心跳功能。每隔 30 秒 (默认设置)，`nsqd` 将会发送一个 `_heartbeat_` 响应，并期待返回。如果客户端空闲，发送 `NOP` 命令。如果 2 个 `_heartbeat_` 响应没有被应答，`nsqd` 将会超时，并且强制关闭客户端连接。`IDENTIFY` 命令可以用来改变/禁用这个行为。

注意

- 除非 stated，所有的传输的二级制大小/整数都是网络字节顺序。(列如. *big endian*)
- 有效的话题 (*topic*) 和通道 (*channel*) 名必须是字符 `[a-zA-Z0-9_-]` 和数字 `1 < length <= 64` (在 `nsqd` 0.2.28 版本前最长 32 位)

命令

IDENTIFY

更新服务器上的客户端元数据和协商功能。

```
IDENTIFY\n
[ 4-byte size in bytes ][ N-byte JSON data ]
```

注意: 这个命令包含 JSON 的相关内容，包括:

- `short_id` (`nsqd` v0.2.28+ 版本之后已经抛弃, 使用 `client_id` 替换)这个标示符是描述的简易格式 (比如, 主机名)
- `long_id` (v0.2.28+ 版之后已经抛弃, 使用 `hostname` 替换)这个标示符是描述的长格式。(比如. 主机名全名)
- `client_id` 这个标示符用来消除客户端的歧义 (比如. 一些指定给消费者)
- `hostname` 部署了客户端的主机名
- `feature_negotiation` (`nsqd` v0.2.19+) bool, 用来标示客户端支持的协商特性。如果服务器接受, 将会以 JSON 的形式发送支持的特性和元数据。
- `heartbeat_interval` (`nsqd` v0.2.19+) 心跳的毫秒数。

有效范围: `1000 <= heartbeat_interval <= configured_max` (`-1` 禁用心跳)

`--max-heartbeat-interval` (nsqd 标志位) 控制最大值

默认值 `--client-timeout / 2`

- `output_buffer_size` (`nsqd` v0.2.21+) 当 `nsqd` 写到这个客户端时将会用到的缓存的大小 (字节数)。

有效范围: `64 <= output_buffer_size <= configured_max` (`-1` 禁用输出缓存)

`--max-output-buffer-size` (nsqd 标志位) 控制最大值

默认值 `16kb`

- `output_buffer_timeout` (`nsqd` v0.2.21+) 超时后, `nsqd` 缓冲的数据都会刷新到此客户端。

有效范围: `1ms <= output_buffer_timeout <= configured_max` (`-1` 禁用 timeouts)

`--max-output-buffer-timeout` (nsqd 标志位) 控制最大值

默认值 `250ms`

警告: 使用极小值 `output_buffer_timeout` (`< 25ms`) 配置客户端, 将会显著提高 `nsqd` CPU 的使用率 (通常客户端连接时 `> 50`)。

这依赖于 Go 的 timers 的实现, 它通过 Go 的优先队列运行时间维护。

- `tls_v1` (`nsqd` v0.2.22+) 允许 TLS 来连接

`--tls-cert` and `--tls-key` (nsqd 标志位s) 允许 TLS 并配置服务器证书

如果服务器支持 TLS，将会回复 `"tls_v1": true`。

客户端读取 `IDENTIFY` 响应后，必须立即开始 TLS 握手。

完成 TLS 握手后服务器将会响应 `OK`。

- `snappy` (`nsqd` `v0.2.23+`) 允许 `snappy` 压缩这次连接

`--snappy` (`nsqd` 标志位) 允许服务端支持

客户端不允许同时 `snappy` 和 `deflate`。

- `deflate` (`nsqd` `v0.2.23+`) 允许 `deflate` 压缩这次连接

`--deflate` (`nsqd` 标志位) 允许服务端支持

客户端不允许同时 `snappy` 和 `deflate`。

- `deflate_level` (`nsqd` `v0.2.23+`) 配置 `deflate` 压缩这次连接的级别

`--max-deflate-level` (`nsqd` 标志位) 配置允许的最大值

有效范围: `1 <= deflate_level <= configured_max`

值越高压缩率越好，但是 CPU 负载也高。

- `sample_rate` (`nsqd` `v0.2.25+`) 投递此次连接的消息接收率。

有效范围: `0 <= sample_rate <= 99` (`0` 禁用)

默认值 `0`

- `user_agent` (`nsqd` `v0.2.25+`) 这个客户端的代理字符串

默认值: `<client_library_name>/<version>`

- `msg_timeout` (`nsqd` `v0.2.28+`) 配置服务端发送消息给客户端的超时时间

成功后响应：

OK

注意: 如果客户端发送了 `feature_negotiation` (并且服务端支持)，响应体将会是 JSON。

错误后的响应内容：


```
E_INVALID
E_BAD_BODY
```

SUB

订阅话题 (topic) /通道 (channel)

```
SUB <topic_name> <channel_name>\n

<topic_name> - 字符串 (建议包含 #ephemeral 后缀)
<channel_name> - 字符串 (建议包含 #ephemeral 后缀)
```

成功后响应:

```
OK
```

错误后响应:

```
E_INVALID
E_BAD_TOPIC
E_BAD_CHANNEL
```

PUB

发布一个消息到 话题 (topic):

```
PUB <topic_name>\n
[ 4-byte size in bytes ][ N-byte binary data ]

<topic_name> - 字符串 (建议 having #ephemeral suffix)
```

成功后响应:

```
OK
```

错误后响应:

```
E_INVALID
E_BAD_TOPIC
E_BAD_MESSAGE
E_PUB_FAILED
```

MPUB

发布多个消息到 话题 (topic) (自动):

注意: nsqd v0.2.16+ 有效

```
MPUB <topic_name>\n
[ 4-byte body size ]
[ 4-byte num messages ]
[ 4-byte message #1 size ][ N-byte binary data ]
... (repeated <num_messages> times)

<topic_name> - 字符串 (建议 having #ephemeral suffix)
```

成功后响应:

```
OK
```

错误后响应:

```
E_INVALID
E_BAD_TOPIC
E_BAD_BODY
E_BAD_MESSAGE
E_MPUB_FAILED
```

RDY

更新 RDY 状态 (表示你已经准备好接收 N 消息)

注意: nsqd v0.2.20+ 使用 --max-rdy-count 表示这个值

```
RDY <count>\n

<count> - a string representation of integer N where 0 < N <= configured_max
```

注意: 这个没有成功后响应

错误后响应:

```
E_INVALID
```

FIN

完成一个消息 (表示成功处理)

```
FIN <message_id>\n
```

<message_id> – message id as 16-byte hex string

注意: 这里没有成功后响应

错误后响应:

```
E_INVALID
E_FIN_FAILED
```

REQ

重新将消息队列 (表示处理失败)

这个消息放在队尾, 表示已经发布过, 但是因为很多实现细节问题, 不要严格信赖这个, 将来会改进。

简单来说, 消息在传播途中, 并且超时就表示 `REQ`。

```
REQ <message_id> <timeout>\n
```

<message_id> – message id as 16-byte hex string

<timeout> – a string representation of integer N where $N \leq \text{configured max timeout}$

0 is a special case that will not defer re-queueing

注意: 这里没有成功后响应

错误后响应:

```
E_INVALID
E_REQ_FAILED
```

TOUCH

重置传播途中的消息超时时间

注意: 在 `nsqd v0.2.17+` 可用

```
TOUCH <message_id>\n
```

<message_id> – the hex id of the message

注意: 这里没有成功后响应

错误后响应:

```
E_INVALID
E_TOUCH_FAILED
```

CLS

清除连接（不再发送消息）

```
CLS\n
```

成功后响应s:

```
CLOSE_WAIT
```

错误后响应:

```
E_INVALID
```

NOP

No-op

```
NOP\n
```

注意: 这里没有 response

AUTH

注意: 在 `nsqd v0.2.29+` 可用

如果 `IDENTIFY` 响应中有 `auth_required=true`，客户端必须在 `SUB`，`PUB` 或 `MPUB` 命令前发送 `AUTH`。否则，客户端不需要认证。

当 `nsqd` 接收到 `AUTH` 命令，它通过执行 HTTP 配置 `--auth-http-address`，这个请求包括以下查询参数：连接的远程地址，TLS 状态，支持的认证密码。更多细节参见：[AUTH](#)

```
AUTH\n
[ 4-byte size in bytes ][ N-byte Auth Secret ]
```

成功后响应:

JSON 包含授权给客户端的身份, 可选的 URL, 和授权过的权限列表。

```
{"identity":"...", "identity_url":"...", "permission_count":1}
```

错误后响应:

```
E_AUTH_FAILED – An error occurred contacting an auth server
E_UNAUTHORIZED – No permissions found
```

数据格式

数据异步传输给客户端, 并且支持各种回复体, 比如

```
[x][x][x][x][x][x][x][x][x][x][x]...
| (int32) || (int32) || (binary)
| 4-byte || 4-byte || N-byte
-----...
size  frame type  data
```

客户端必须是以下类型之一:

```
FrameTypeResponse int32 = 0
FrameTypeError   int32 = 1
FrameTypeMessage int32 = 2
```

以及消息格式:

```
[x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x][x]...
| (int64)   || || (hex string encoded in ASCII) || (binary)
| 8-byte   || || 16-byte || N-byte
-----
nanosecond timestamp ^^      message ID      message body
      (uint16)
      2-byte
      attempts
```

客户端库

下面表里的信息是否过时了？你是否在使用这些客户端库来开发？用 [mailing list](#) 或 Twitter [@imsnakes](#) / [@je hiah](#) 告诉我们。

Name	Language	SU B	PU B	Discov ery	Back off	TL S	Snap py	Sampli ng	AUT H	Note s
nsqd	HTTP		✓							built-in
go-nsq	Go	✓	✓	✓	✓	✓	✓	✓	✓	official
pynsq	Python	✓	✓	✓	✓	✓	✓	✓	✓	official
nsqjs	JavaScript (CoffeeScript)	✓	✓	✓	✓	✓	✓	✓	✓	official
nsq-py	Python	✓	✓	✓	✓	✓	✓	✓	✓	
gnsq	Python	✓	✓	✓	✓	✓	✓	✓	✓	
krakow	Ruby	✓	✓	✓	✓	✓	✓	✓		
JavaNSQClient	Java	✓	✓	✓	✓	✓	✓	✓		
ensq	Erlang	✓	✓	✓	✓					
nsq.js	JavaScript	✓	✓	✓						
TrendrrNSQClient	Java	✓	✓	✓						
nsqjava	Java	✓	✓							
nsqphp	PHP	✓	✓	✓						
node-nsqueue	JavaScript	✓	✓							
ruby_nsq	Ruby	✓	✓		✓					
libnsq	C	✓								official
nsq-ruby	Ruby	✓	✓	✓						
NsqSpinner	Python	✓	✓	✓	✓	✓	✓	✓		
nsq-java	Java	✓	✓	✓						
NSQnet	.NET	✓	✓	✓						
nsq-client	JavaScript	✓	✓							
hsnsq	Haskell	✓	✓							
perl-anyevent-nsq	Perl	✓	✓	✓						
nsq-clojure	Clojure									
nsqie	Scala	✓		✓						

nodensq	JavaScript	✓	✓							
---------	------------	---	---	--	--	--	--	--	--	--

编译客户端库

NSQ 将一些功能集成到客户端库中，以便维持集群的健壮性和性能。

这篇文章试图列出客户端库通常需要完成的功能。因为发布到 `nsqd` 非常的琐碎（仅用 HTTP POST `/put` 节点就可），这个文档主要关注消费者。

通过规范，我们希望各种语言实现的时候都能保持一致性。

配置

从高层看，配置相关的设计理念是希望系统能支持不同的工作负载，使用相同的默认值能立即可用，并且能将拨号数最小化。

消费者通过 TCP 连接到 `nsqd` 实例，订阅 通道 (channel) 上的 话题 (topic)。每个连接只能订阅一个话题 (topic)，因此消费多个话题 (topic)，必须响应的结构化。

使用 `nsqlookupd` 来发现是方案之一，所以客户端库必须支持消费者直接连接一个或多个 `nsqd` 实例，或者它可用轮询一个或多个 `nsqlookupd` 实例。当消费者轮询 `nsqlookupd` 的时候，时间间隔必须是可配置的。另外，因为 NSQ 的标准部署是分布式环境，包含很多消费者和生产者，客户端库必须根据配置值得随机性自动添加抖动。更多细节参考[发现 \(页 0\)](#)。

对于消费者来说，在 `nsqd` 响应前能接收到多少消息是非常重要的指标。这个管道促进缓存，批处理，异步消息处理。这个值称为 `max_in_flight`，并且它影响了 `RDY` 状态。更多细节参见[RDY 状态 \(页 0\)](#)。

设计系统时通常会考虑优雅处理失败，客户端库希望能实现失败消息的重试，并提供边界参数来处理每个消息尝试次数。更多细节参见[消息处理 \(页 0\)](#)。

当消息处理失败的时候，客户端库能自动将消息重新队列。NSQ 支持使用 `REQ` 命令发送延迟。客户端库需要能提供延迟的初始化值（第一次失败时），以及重新队列失败该如何改变。更多细节参见[Backoff \(页 0\)](#)。

最重要的时，客户端库必须支持消息处理的回调函数配置。这些回调函数必须简单，通常都支持一个参数（消息对象的实例）。

发现

`nsqlookupd` 是 NSQ 的重要组成部分，它为消费者发现服务提供来定位 `nsqd`，它在运行时提供一个指定话题（topic）。

虽然使用 `nsqlookupd` 能大幅减少配置数目，但是需要维持并放大一个巨大的分布式 NSQ 集群。

当消费者使用 `nsqlookupd` 来发现时，客户端库必须管理轮询所有 `nsqlookupd` 实例的进程，最新的 `nsqd` 组合以问题形式提供了话题（topic），并且管理到这些 `nsqd` 的连接。

查询一个 `nsqlookupd` 实例非常的简单。执行一个 HTTP 请求，使用消费者试图发现的话题（topic）作为查询参数来查找节点（例如 `/lookup?topic=clicks`）。响应体是 JSON：

```
{
  "status_code": 200,
  "status_txt": "OK",
  "data": {
    "channels": ["archive", "science", "metrics"],
    "producers": [
      {
        "broadcast_address": "clicksapi01.routable.domain.net",
        "hostname": "clicksapi01.domain.net",
        "tcp_port": 4150,
        "http_port": 4151,
        "version": "0.2.18"
      },
      {
        "broadcast_address": "clicksapi02.routable.domain.net",
        "hostname": "clicksapi02.domain.net",
        "tcp_port": 4150,
        "http_port": 4151,
        "version": "0.2.18"
      }
    ]
  }
}
```

`broadcast_address` 和 `tcp_port` 必须用来连接 `nsqd`。因为从设计上来说 `nsqlookupd` 实例不会分享或协调他们的数据，客户端库必须联合它接收到得所有 `nsqlookupd` 查询列表来建立 `nsqd` 最终列表。使用 `broadcast_address:tcp_port` 作为这个联合的唯一 KEY。

必须用周期性的计时器来重复的轮询 `nsqlookupd` 的配置，这样消费者能自动的发现新的 `nsqd`。客户端库必须自动的初始化到所有新发现的实例的连接。

当客户端库开始执行的时候，它必须通过踢开配置 `nsqlookupd` 实例的一组请求，来引导轮询。

连接处理

一旦消费者有一个 `nsqd` 可以连接(通过发现或手工配置), 它就必须打开一个 TCP 连接到 `broadcast_address:port`。一个单独的 TCP 连接必须能让消费者可以订阅到每个 `nsqd` 的话题 (topic)。

当连接到一个 `nsqd` 实例时，客户端库必须发送以下数据，顺序是：

1. 魔术标识符
2. 一个 `IDENTIFY` 命令 (和负载) 和读/验证响应
3. 一个 `SUB` 命令 (指定需要的话题 (topic)) 和读/验证响应
4. 一个初始化 `RDY` 值 1

(低级别的细节参见 [spec](#))

重新连接

客户端库必须通过以下方法自动重新连接：

- 如果消费者通过特定的 `nsqd` 列表指定，重新连接必须通过延迟重试来处理。（例如，8s, 16s, 32s, 等等，到最大值后重试）。
- 如果消费者通过 `nsqlookupd` 来发现实例，必须通过轮询间隔来自动处理重新连接（例如，如果消费者断开和 `nsqd` 的连接，客户端库仅在随后的 `nsqlookupd` 轮询发现的实例后重新连接）。这能保证消费者了解 `nsqd`。

特性协商

`IDENTIFY` 命令可以用来设置 `nsqd` 端的元数据，修改客户端设置，并特性协商，它满足亮点：

1. 某些情况下，客户端可能会修改 `nsqd` 的交互方式（比如，修改客户端的心跳间隔，并允许压缩，TLS，输出缓存，等等-完整列表参见 [spec](#)）

2. `nsqd` 使用 JSON payload 来响应 `IDENTIFY` 命令，它包含了重要的服务端配置值，客户端和之交互时必须遵守。

连接后，根据用户的配置，客户端库必须发送一个 `IDENTIFY` 命令，它的内容是 JSON payload:

```
{
  "client_id": "metrics_increment",
  "hostname": "app01.bitly.net",
  "heartbeat_interval": 30000,
  "feature_negotiation": true
}
```

`feature_negotiation` 位表示客户端可以接受返回值是 JSON payload。`client_id` 和 `hostname` 是随意的文本字段，`nsqd` (和 `nsqadmin`) 会用来区别客户端。`heartbeat_interval` 配置每个客户端的心跳间隔。

`nsqd` 必须响应 `OK`，如果它不支持特性协商 (`nsqd`v0.2.20+`` 引入), 否则:

```
{
  "max_rdy_count": 2500,
  "version": "0.2.20-alpha"
}
```

数据流和心跳

一旦消费者处于订阅状态，NSQ 协议里的数据流时异步的。对于消费者来说，这就是说如果想建立一个健壮并高效的客户端库，就必须使用异步的网络 IO 循环和/或“线程”（线程表示 OS 级别的线程和用户空间（`userland`）的进程，比如协同程序（`coroutines`））。

另外，期望客户端能响应它们连接到的 `nsqd` 实例的周期性心跳。通常这个周期是 30 秒。客户端可以使用任何命令响应，不过通常方便起见，使用 `NOP` 响应心跳。更多细节参见 [protocol spec](#)。

“进程”必须专注于读取 TCP socket 的数据，解包帧数据，并执行多路逻辑来传输。这也是处理心跳最佳点。从最低级别看，读取协议包括以下步骤：

1. 读取 4 字节 big endian uint32 大小
2. 读取字节大小数据
3. 解包数据
4. ...
5. profit

6. goto 1

一个和错误相关小插曲

根据系统的异步特性，会采用更多的状态来追踪相关协议的由命令产生的错误。我们会采用“快速错误”（"fail fast"）方法，所以大量协议级别错误处理都是致命的。这意味着如果客户端发送一个无效命令（或者自己是无效状态），通过强制关闭连接（如果可能，发送一个错误给客户端），它连接到的 `nsqd` 实例将会保护自己（和系统）。和之前提到的连接处理相配合，使得系统更加健壮和稳定。

仅有的几个非致命错误是：

- `E_FIN_FAILED` - `FIN` 命令, 无效的消息 ID
- `E_REQ_FAILED` - `REQ` 命令 无效的消息 ID
- `E_TOUCH_FAILED` - `TOUCH` 命令 无效的消息 ID

因为这些错误通常和时间有关，所以不当做致命错误。这些错误通常发生在 `nsqd` 端消息超时，重新队列时，和投递到其他消费者时。原先的接受者不再允许响应这个消息。

消息处理

当 IO 循环解包包含消息的帧数据时，它必须路由这个消息给配置处理函数来处理。

发送 `nsqd`，在配置消息超时时希望收到回复（默认：60秒）。可能有以下场景：

1. 处理函数表示消息已经成功处理
2. 处理函数表示消息正处理成功
3. 处理函数表示需要更多的时间来处理消息
4. in-flight 超时，并且 `nsqd` 自动重新队列消息

前 3 个情况，客户端库必须发送合适消费者方面的命令（`FIN`，`REQ`，和 `TOUCH`）。

`FIN` 命令最简单。它告诉 `nsqd` 它能安全的抛弃消息。`FIN` 也能抛弃那些你不想处理或重试的消息。

`REQ` 命令告诉 `nsqd`，消息必须重新队列（可选参数指定了重试的次数）。如果消费者没有指定可选参数，客户端库必须自动算出相关联的消息处理的时长（通常设置为多倍，这样效率更高）。客户端库必须抛弃超过最多重试次数的消息。当它发生的时候，必须执行用户提供的回调来通知，并运行特定的回调。

如果消息处理函数需要的时间超过配置的超时时间，可以用 `TOUCH` 命令来重置 `nsqd` 端的计时器。可以重复这个动作，直到消息 `FIN` 或 `REQ`，或发送 `nsqd` 的配置属性 `max_msg_timeout`。客户端库不能自动 `TOUCH` 代表消费者。

如果发送 `nsqd` 实例没有接收到响应，消息将会超时，并会自动重新队列来投递到可用的消费者。

最后，每个消息的属性是尝试次数。客户端库必须比较这个值和配置的最大值，并且抛弃已经超过这个值得消息。当消息已经抛弃的时候，需要触发回调。通常这个回调的实现必须包括写入磁盘，日志等等。用户必须能重写默认的处理函数。

RDY 状态

因为消息是从 `nsqd` 推送到消费者那，我们必须拥有一个方法来管理数据流，而不仅依赖于低级别的 TCP 语法。消费者的 `RDY` 状态是 NSQ 的流控制机制。

如配置中列出的内容，通过 `max_in_flight` 配置消费者。这是并行的并且性能 knob。比如一些 downstream 系统可以更加容易进行消息批处理，并对更高级的 `max-in-flight` 有利。

当消费者连接到 `nsqd`（并且订阅），`RDY` 初始化状态为 `0`。不会投递任何消息。

客户端库拥有很少的责任：

1. 引导并最终分布配置 `max_in_flight` 到所有的连接。
2. 永远不允许汇集所有连接 `RDY` 的和(`total_rdy_count`)，为超过 `max_in_flight` 的配置。
3. 永远不要超过每个连接 `nsqd` 配置的 `max_rdy_count`。
4. 暴露一个 API 方法给值得信赖的消息流。

1. 引导和分布

为连接选择 `RDY` 值，需要考虑的因素很少（最终分布为 `max_in_flight`）：

- 连接 # 是动态的，通常并不知道次数（例如，当通过 `nsqlookupd` 发现 `nsqd`）。
- `max_in_flight` 可能会小于你的连接数

为了开始消息流，客户端库必须发送一个初始的 `RDY` 值。因为最终的连接数并不知道（通常从 '1' 开始），所以客户端库必能公平对待每个连接。

另外，每个消息处理后，客户端库必须评估什么时候更新 RDY 状态。如果当前值是 '0'，或者低于最后发送的值的 25% 必须触发更新。

客户端库必须一直尝试最终分布 RDY 值到所有的连接。

通常来说，它可以通过 $\text{max_in_flight} / \text{num_conns}$ 实现。

然而，当 $\text{max_in_flight} < \text{num_conns}$ 这个简单的公式无效的时候。客户端库必须执行一个动态的运行评估，自从通过之前的连接接收到消息后，连接的 nsqd '活跃度'的时间。当配置到期后，他必须重新分布，不论 RDY 值是否对于新的 nsqd 有效。这么做，你能保证你可以通过消息找到 nsqd。这些会有延迟的影响。

2. 维护 max_in_flight

客户端库必须维护指定消费者的消息 in flight 的最大值。尤其，汇集每个连接的 RDY 值永远不能超过配置的 max_in_flight 值。

底下的 Python 代码，它指出 RDY 值是否对于指定的连接有效。

```
def send_ready(reader, conn, count):
    if (reader.total_ready_count + count) > reader.max_in_flight:
        return

    conn.send_ready(count)
    conn.rdy_count = count
    reader.total_ready_count += count
```

3. nsqd 最大 RDY 值

每个 nsqd 通过 `--max-rdy-count` 配置，如果消费者发送的 RDY 值超过了可接受的范围，它的连接将强制关闭。为了向后兼容，这个值必须假设为 2500，如果 nsqd 实例不能支持特性协商。

4. 消息流 Starvation

最终，客户端库必须提供一个 API 方法，来表示消息流 starvation。对于消费者（消费者处理函数）来说，简单比较 in-flight 的消息数和 max_in_flight 值，来决定是否”批处理“不太合适。有两种情况有问题：

1. 当消费者配置 $\text{max_in_flight} > 1$ ，根据变量 num_conns ， max_in_flight 除 num_conns 除不尽。因为你永远不能超过 max_in_flight ，你必须降低，并且在 RDY 值少于 max_in_flight 时结束。

2. 如果仅仅 `nsqd` 的子集有消息，因为even distribution 的 `RDY` 预期值, 这些活跃 `nsqd` 仅有 `max_in_flight` 的片段。

以上两种情况，消费者实际上永远不会接受消息的 `max_in_flight` 。因此，客户端库必须暴露一个方法 `is_starved` ，表示任何连接是否 `starved`，如下：

```
def is_starved(conns):
    for c in conns:
        # the constant 0.85 is designed to *anticipate* starvation rather than wait for it
        if c.in_flight > 0 and c.in_flight >= (c.last_ready * 0.85):
            return True
    return False
```

`is_starved` 方法必须由消息处理函数使用，来发现什么时候处理批量消息。

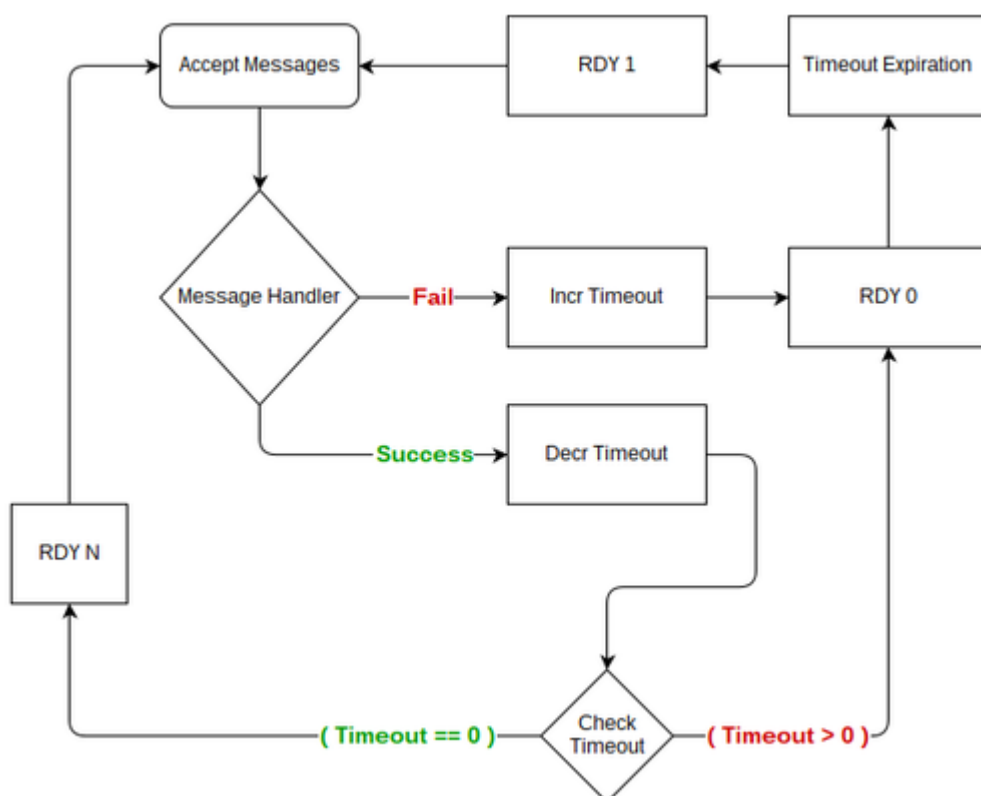
Backoff

消息处理失败的时候如何处理是一个非常复杂的问题。消息处理章节介绍了客户端库动作，它会处理和时间相关的失败的消息。其他的问题是是否减少吞吐量。这两个功能对于整个系统的稳定性至关重要。

通过减慢处理的速率，或者 "backing off"，消费者允许 downstream 系统回收传输失败。然而这个行为必须是可配置的，因为不是什么时候都能称心如意，这种情况下延迟必须优先处理。

Backoff 必须通过发送 `RDY 0` 到合适的 `nsqd` 来实现，停止消息流。这个状态的时长通过重试的失败来计算。处理成功会减少这个时长，直到 reader 不再是 backoff 状态。

当 reader 是 backoff 状态时，超时后，客户端库必须仅发送过 `RDY 1`，而不是 `max_in_flight`。在返回完整的 throttle 前，这是有效的 "tests the waters"。另外，backoff 超时，客户端库必须忽略任何和计算 backoff 时间成功或者失败结果。（比如，每次超时时它仅信任一个结果）



图片 3.1 nsq_客户端_flow

加密/压缩

NSQ 支持加密和/或压缩特性协商，通过 `IDENTIFY` 命令。TLS 用来加密。[Snappy](#) 和 DEFLATE 都支持压缩。Snappy 可作为第三方库使用，但是基本所有的语言都支持 DEFLATE。

收到 `IDENTIFY` 响应时，并且你通过 `tls_v1` 标志位请求 TLS，你得到的东西和以下内容类似：

```
{
  "deflate": false,
  "deflate_level": 0,
  "max_deflate_level": 6,
  "max_msg_timeout": 900000,
  "max_rdy_count": 2500,
  "msg_timeout": 60000,
  "sample_rate": 0,
  "snappy": true,
  "tls_v1": true,
  "version": "0.2.28"
}
```


确认 `tls_v1` 为 `true` 后（意味着服务器支持 TLS），在接受和发送任何消息前，你需要初始化 TLS 握手（例如，Python 使用 `ssl.wrap_socket` 表示完成）。TLS 握手成功后，你必须立即读取一个 NSQ 加密的 `OK` 响应。

如果你想压缩，可以设置 `snappy` 或 `deflate` 为 `true`，并且使用合适压缩（解压缩）调用读写。同样的你必须立即读取一个 NSQ 压缩的 `OK` 响应。

这些压缩特性是互斥的。

你不能阻止缓存直到加密/压缩协商完成，或者确保小心的读取到内存。

汇总

分布式系统非常有意思。

不同的 NSQ 集群部门间交互在一个平台上，它健壮，高性能，并且稳定。希望您能这篇文章里了解到客户端是多么重要。

这些细节的实现，我们将 `pynsq` 和 `go-nsq` 作为代码基础。`pynsq` 可以切割为 2 个部分：

- `Message` – 高级别的消息对象，它暴露了状态方法，来响应 `nsqd`（`FIN`，`REQ`，`TOUCH` 等等），同时元数据包含目的和时间戳。
- `Connection` – 高级别的封装，包含 TCP 连接到一个指定的 `nsqd`，它包含 flight 消息，`RDY` 状态，协商特性，和不同时间。
- `消费者` – 和用户打交道的 API，它处理发现，创建连接（和订阅），引导和管理 `RDY` 状态，解析收到的数据，创建消息对象，和分发消息给处理函数。
- `Producer` – 和用户打交道的 API，处理发布。

我们很高兴能帮助任何对编写客户端库有兴趣的人。我们希望大家能加入到社区，扩展目前已经存在的库。社区已经开源[很多客户端库](#)。



部署



安装

Binary Releases

为 linux 和 darwin 预编译二进制文件 (`nsqd` , `nslookupd` , `nsqadmin` , 以及所有的例子应用), 可用来下载。

当前稳定 Release 版本: v0.3.5

- [nsq-0.3.5.darwin-amd64.go1.4.2.tar.gz](#)
- [nsq-0.3.5.linux-amd64.go1.4.2.tar.gz](#)

老的稳定 Release 版本

- [nsq-0.3.2.darwin-amd64.go1.4.1.tar.gz](#)
- [nsq-0.3.2.linux-amd64.go1.4.1.tar.gz](#)
- [nsq-0.3.1.darwin-amd64.go1.4.1.tar.gz](#)
- [nsq-0.3.1.linux-amd64.go1.4.1.tar.gz](#)
- [nsq-0.3.0.darwin-amd64.go1.3.3.tar.gz](#)
- [nsq-0.3.0.linux-amd64.go1.3.3.tar.gz](#)
- [nsq-0.2.31.darwin-amd64.go1.3.1.tar.gz](#)
- [nsq-0.2.31.linux-amd64.go1.3.1.tar.gz](#)

Docker

参见 [the docs](#) 使用 [Docker](#) 部署 NSQ。

OSX

```
$ brew install nsq
```

从源文件编译

Pre-requisites

- [golang](#) (version `1.2+` is required)
- [gpm](#) (dependency manager)

编译

NSQ 使用 [gpm](#) 来管理依赖文件。编译源文件，`gpm` 是首选方案。

```
$ gpm install  
$ go get github.com/bitly/nsq/...
```

NSQ 保持了 `go get` 兼容，但是不推荐，因为之后不能保证仍然能稳定编译。

Testing

```
$ ./test.sh
```

产品配置

虽然 `nsqd` 可以单独运行成节点，我们还是假设你希望使用分布式系统的优势。

以下是独立的二进制文件，需要安装并运行：

nsqd

`nsqd` 是守护进程，接收，缓存，并投递消息给客户端

所有的配置都通过命令行参数来管理。我们希望默认配置能满足多数应用场景，有一部分参数值得注意：

`--mem-queue-size` 调整每个话题（topic）/通道（channel）消息队列数。超过上限的消息，将会写到持平，通过 `--data-path` 定义。

同时，`nsqd` 将会需要通过 `nsqlookupd` 配置（参见以下详情），为每个实例指定参数。

拓扑结构，我们推荐运行 `nsqd`，和生产消息服务共同写作。

`nsqd` 可以配置来推送数据到 [statsd](#)，通过指定 `--statsd-address`。在 `nsq.*` 命令空间里，`nsqd` 发送统计数据，参见 [nsqd statsd](#)。

nsqlookupd

`nsqlookupd` 是一个守护进程，为消费者提供运行时发现服务，来查找指定话题（topic）的生产者 `nsqd`。

它维护非持久化状态，并且不需要和其他 `nsqlookupd` 实例来满足产线。

运行数据取决于你的冗余需求。使用很少的支援，并且可以和其他服务协同操作。我们推荐每个数据中心最少运行 3 个集群。

nsqadmin

`nsqadmin` 是 Web 服务，用来实时的管理你的 NSQ 集群。它通过和 `nsqlookupd` 实例交流，来确定生产者和 [graphite](#) 图表（要求打开 `nsqd` 端 `statsd`）。

我们仅需运行一个，并使它可以公开访问（安全）。

仅有一些 HTML 模板需要部署。默认 `nsqadmin`，位于 `/usr/local/share/nsqadmin/templates`，可以通过 `-template-dir` 重写。

要显示 `graphite` 图表，指定 `--graphite-url`。如果你已经使用 `statsd`，给所有的 keys 添加前缀，就需要指定 `--use-statsd-prefixes`。最后，如果 `graphite` 不能公开访问，通过指定 `--proxy-graphite`，你可以使用 `nsqadmin` 代理这些请求。

Monitoring

每个守护进程，都拥有 `/ping` HTTP 端点，它可以用来创建监控检测。

即使实时调试，它也能运行的非常好：

```
$ watch -n 0.5 "curl -s http://127.0.0.1:4151/stats"
```

一般通过 `nsqadmin` 进行调试，分析，管理。

拓扑模式

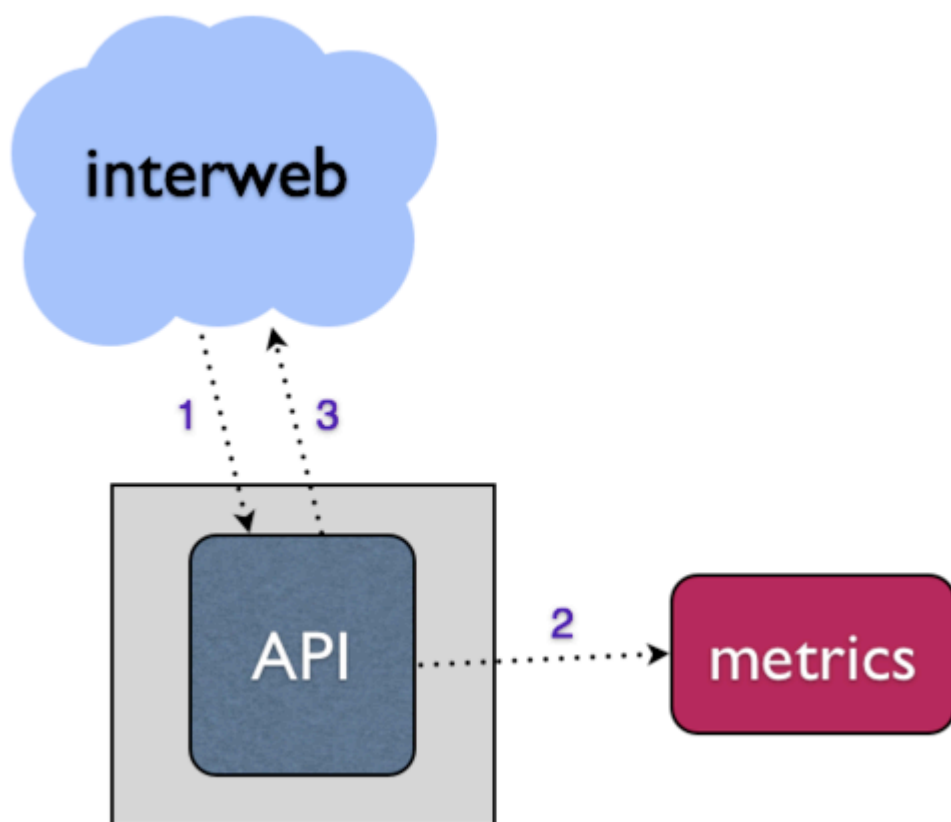
这个文档描述了一些 NSQ 模式，解决不同的问题。

免责声明: 已经有一些明显的技术建议，但是这个文档通常会忽略，深层的个人选择合适工具的细节，获取软件安装到生产环境，管理服务在哪里运行，服务配置，并管理运行进程 (`daemontools` , `supervisord` , `init.d` 等等)。

指标收集

无论你编译的是哪个类型的 Web 服务，多数情况下你会想收集各种指标，来了解你的基础架构，你的用户，你的业务。

对于 Web 服务，多数指标是由 HTTP 请求事件产生的，比如 API。本地的方法可能会结构化这个异步操作，通过 API 请求直接写到你的指标系统。



图片 4.1 naive approach

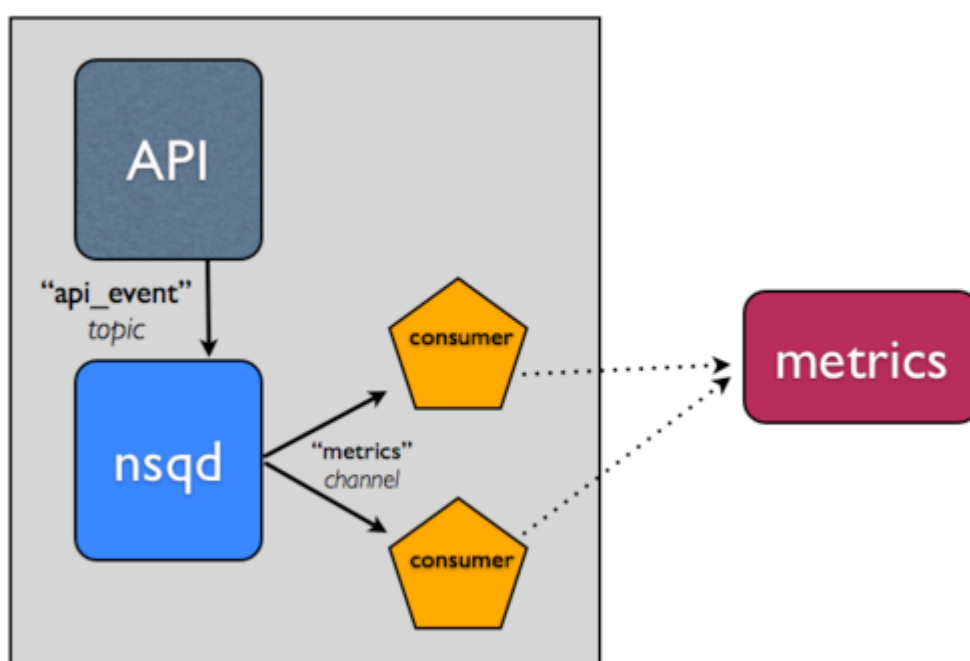
- 当你的指标系统下降的时候会发生什么？

- 你的 API 请求是否挂起和/或失败？
- 你如何处理增长 API 请求挑战？

解决这些问题的一个方法是设法异步写入到你的指标系统，就是说，将数据放到本地队列中，并通过其他进程写到你的下行系统（消费这个队列）。这个独立操作让系统更加健壮，容错性更强。在 bitly，我们使用 NSQ 完成这个目标。

NSQ 有话题（topic）和通道（channel）两个概念。假设将话题当做消息的唯一流（如同我们的 API 事件流）。假设，将通道当做这个消息流的指定消费者集合的拷贝。话题和通道都是独立队列。这些特性允许 NSQ 支持多播（话题拷贝每个消息到 N 通道）并且分发消息投递（通道平分它的消息到 N 个消费）。

更多细节参考[design doc](#) 和 [slides from our Golang NYC talk](#), 尤其 [slides 19 through 33](#) 描述了话题（topic）和通道（channel）的细节。



图片 4.2 architecture with NSQ

完整的 NSQ 比较简单，注意两点：

1. 在 API 应用所运行的主机上，运行 `nsqd` 实例。
2. 更新你的 API 应用，写到本地 `nsqd` 实例队列事件，而不是写到指标系统。能够容易的内审和操作流，我们通常将数据格式化为 line-oriented JSON。写到 `nsqd` 可以简单的执行一个 HTTP post 请求到 `/put` 端点。
3. 用 [client libraries](#) 在你的语言创建一个消费者。这个“工作者”将会订阅数据流，并会处理事件，写到你的指标系统。它也能运行在主机上，运行你的 API 应用和 `nsqd`。

这有一个使用官方[Python client library](#) 写的例子：

除了解耦之外，通过使用我们官方的客户端库，当消息处理错误的时候，消费者可以优雅的降级。我们的库有 2 个关键特性：

1. **重试** – 当你的消息处理函数返回失败，这个消息将会以 `REQ` (re-queue) 命令形式发送给 `nsqd` 。同时，如果在时间窗里消息还没被响应，`nsqd` 将会自动将消息超时（并重新队列）。这两个特性对于消息投递保障非常关键。
2. **Backoff 指数** – 当消息处理失败，读取库将会延迟附加信息的收据，放大建立在 # 连续的失败指数。当读取者处于 backoff 状态，并且开始处理成功，直到为 0 时，会发生相反序列。

从概念上来说，这两个特性允许系统优雅的自动响应下行失败。

持久化

好，现在你可以应付你的指标系统因为没有数据并且没有 degraded 的 API 服务到其他端点，导致的不可用。你也可以通过从同一个通道（channel）添加更多的工作实例给消费者，放大这个流系统的水平线。

但是，提前想清楚所有的 API 事件的指标，也不太可能。

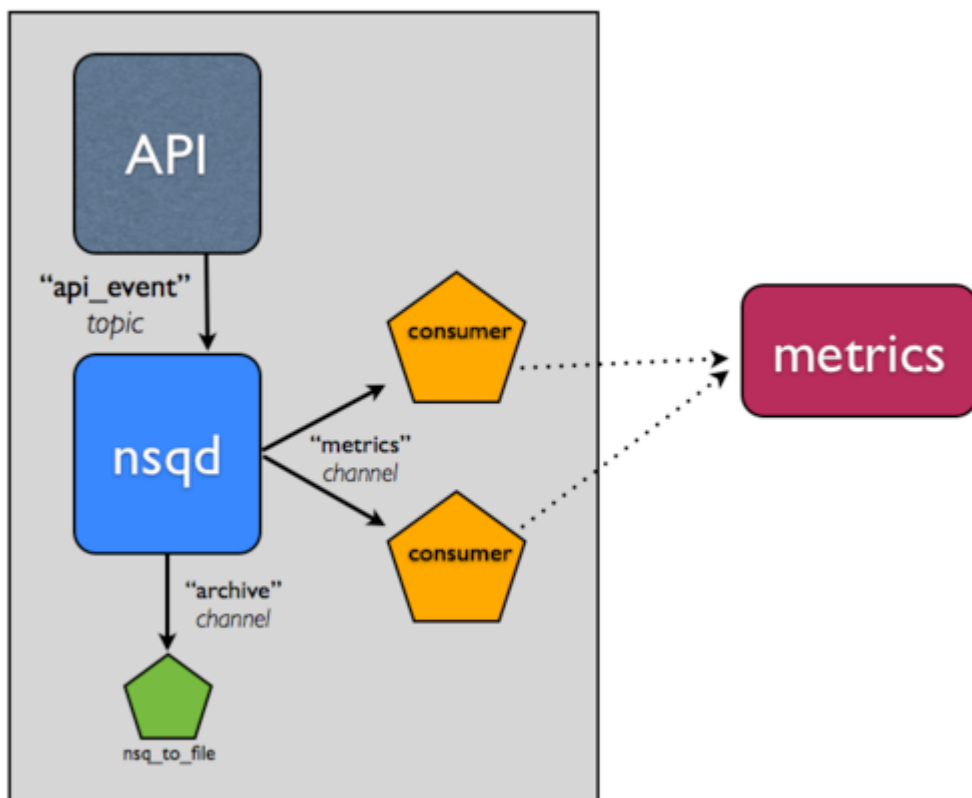
是否有数据流系统的 log，能满足未来任何操作？日志会很容易导致冗余备份，可以把它作为 downstream 系统发生灾难时的 "plan z"。但是，你会希望消费者来完成消息数据的备份？也许不是，因为这是整个 "separation of concerns" 的事情。

归档 NSQ 话题（topic）非常常见，所以我们建了一个工具，[nsq_to_file](#)（使用 NSQ 打包），你可用它来完成。

记住，在 NSQ 中，每个话题（topic）的通道（channel）是独立的，并且接收所有消息的拷贝。你可以利用这个特性，来完成流的归档。实际上，这意味着如果你的指标系统已经有这些问题，并且 `metrics` 通道得到支持，它就不会影响到独立的 `archive` 通道，你将会持久化消息到磁盘。

所以，添加一个 `nsq_to_file` 实例到同一个主机，并且使用命令行如下：

```
/usr/local/bin/nsq_to_file --nsqd-tcp-address=127.0.0.1:4150 --topic=api_requests --channel=archive
```



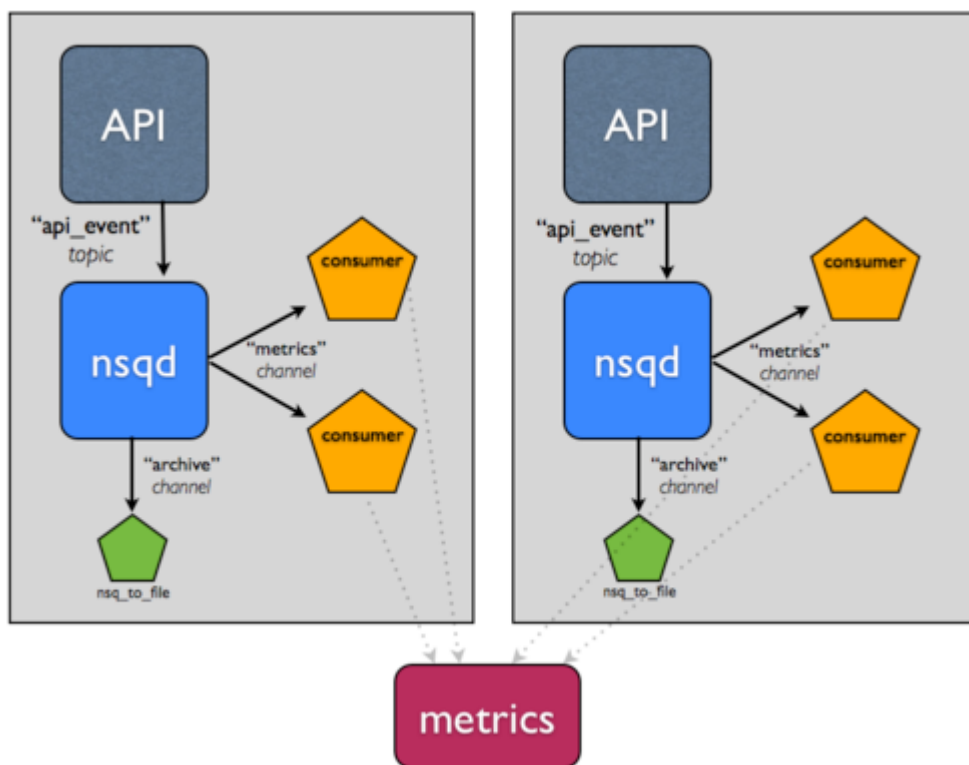
图片 4.3 archiving the stream

分布式系统

可能你已经注意到了，目前系统还没有超出单机，这是明显的错误。

不幸的是，要建立一个分布式系统很难。幸运的是，NSQ 可以帮助你。底下的改变演示了 NSQ 如何减轻建立分布式系统的痛苦，以及如何完成高可用性和容错。

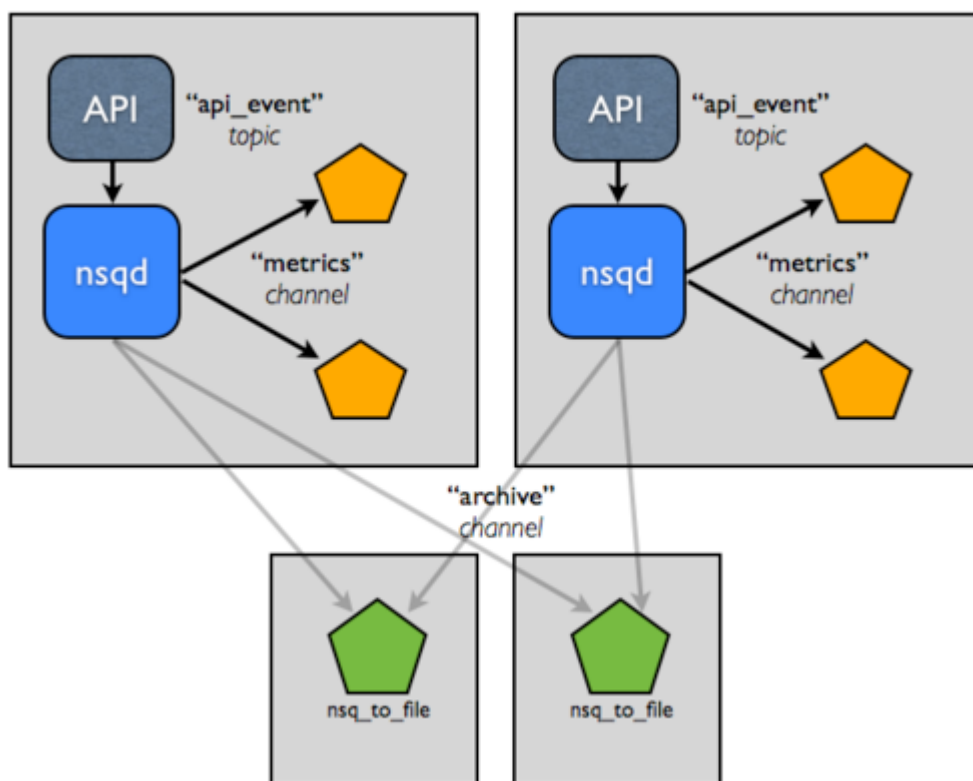
假设这个事件流非常重要。你希望能容忍主机错误，并保证消息最终能到归档，所以你增加了另一个主机。



图片 4.4 adding a second host

假设你已经在这两个主机间负载均衡，这样你就可以容忍单个主机错误。

现在，我们觉的持久化处理，压缩，传输这些日志会影响性能。如何切割这些责任到这些主机上，让它们拥有更高的 IO 能力？

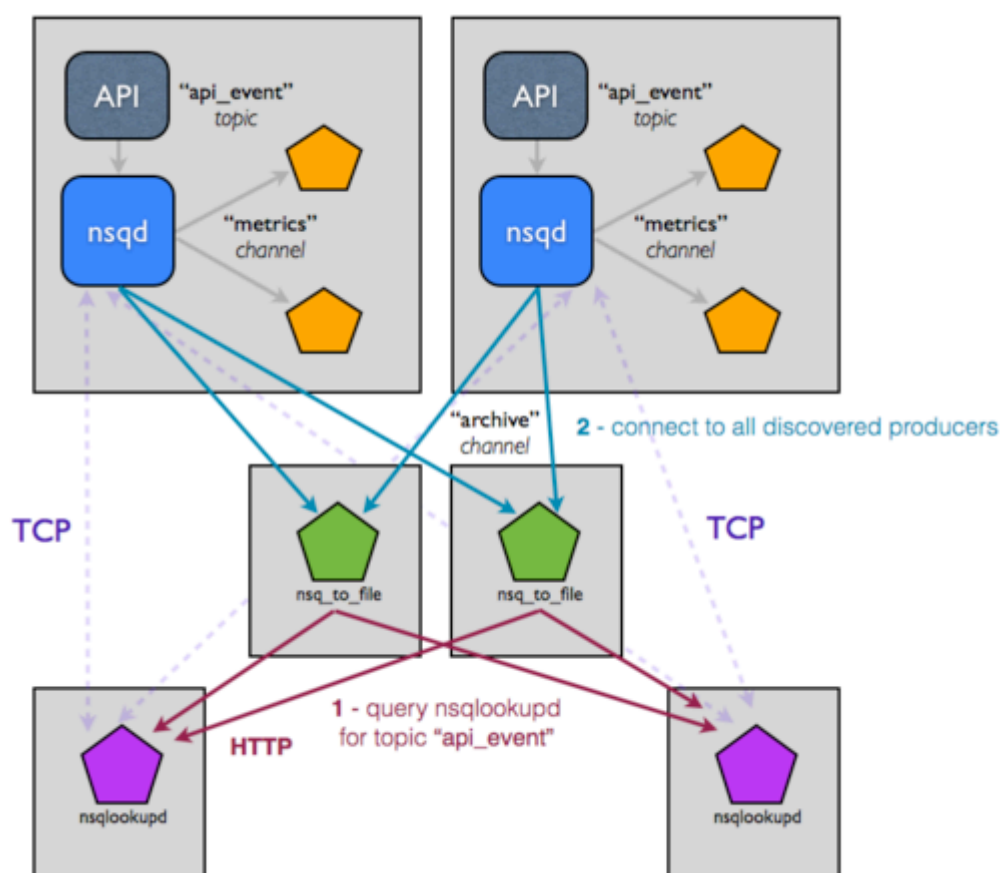


图片 4.5 separate archive hosts

这个拓扑结构和配置可以容易放大到双位主机，但是你仍然手工管理这些服务配置。尤其对于每个消费者，这个创建过程很难从代码上确定 `nsqd` 实例在哪里。你真正希望的是配置能进化，并且在 NSQ 集群的状态基础上运行时可访问。这是我们建立 `nsqlookupd` 的目的。

`nsqlookupd` 是一个守护进程记录并传播 NSQ 集群运行时候的状态。`nsqd` 实例维护 TPC 连接的持久化来 `nsqlookupd`，并且推送状态变化。具体来说，`nsqd` 将自己注册为某个话题（topic）的生产者，以及所有它知道的通道（channel）。它允许消费者查询 `nsqlookupd`，来确定谁是感兴趣的话题（topic）的生产者。久而久之，他们将会知道新的生产者的存在，并能路由失败。

你需要做的改变仅仅是，`nsqlookupd` 时指出存在的 `nsqd` 和消费者实例（每个人都知道 `nsqlookupd` 实例在哪里，但是消费者不会明确的知道生产者在哪里，反之亦然）。现在拓扑结构如下：



图片 4.6 adding nsqlookupd

乍一看，这个图变复杂了。这图具有误导性，整个图节点变多了，导致很难直接通讯。因为 `nsqlookupd` 能作为文件夹服务，你能高效的把生产者和消费者解耦。依赖于已有的流添加一个下行服务非常简单，只需指定你感兴趣的话题（topic）（通过 `nsqlookupd` 可以找到生产者）。

如何保证查找数据的可用性和一致性？`nsqlookupd` 并不会占用大量资源，并且能很容易和其他服务器在一起工作。同时 `nsqlookupd` 实例不需要调整，或者和其他组合。消费者通常只需要一个 `nsqlookupd`（它们将会联合它们所知的 `nsqlookupd` 实例响应）。这样就很容易迁移到新的 `nsqlookupd` 组合中。

Docker

这篇文章详细介绍了如何部署并在 [Docker](#) 容器里运行 NSQ 二进制文件。

这有一个最小化的 NSQ 镜像文件，它包含了所有的 NSQ 二进制文件。每个二进制文件可以通过命令指定运行。基本格式如下：

```
docker run nsqio/nsq /<command>
```

注意命令前的 `/`，例如：

```
docker run nsqio/nsq /nsq_to_file
```

链接

- [docker](#)
- [nsq image](#)

运行 nsqlookupd

```
docker pull nsqio/nsq
docker run --name lookupd -p 4160:4160 -p 4161:4161 nsqio/nsq /nsqlookupd
```

运行 nsqd

首先，得到 docker 主机 ip：

```
ifconfig | grep addr
```

接着，运行 `nsqd` 容器：

```
docker pull nsqio/nsq
docker run --name nsqd -p 4150:4150 -p 4151:4151 \
  nsqio/nsq /nsqd \
  --broadcast-address=<host> \
  --lookupd-tcp-address=<host>:<port>
```

设置 `--lookupd-tcp-address` 标志位到主机 IP，以及之前运行的 TCP 端口：

nsqlookupd，i.e. dockerIP:4160：

例如，指定主机IP 172.17.42.1：

```
docker run --name nsqd -p 4150:4150 -p 4151:4151 \
  nsqio/nsq /nsqd \
  --broadcast-address=172.17.42.1 \
  --lookupd-tcp-address=172.17.42.1:4160
```

注意：这里使用端口 4160，端口暴露了什么我们什么开始运行 nsqlookupd 容器（它也是 nsqlookupd 的端口）。

如果你不想使用默认端口，改变 -p 参数：

```
docker run --name nsqlookupd -p 5160:4160 -p 5161:4161 nsqio/nsq /nsqlookupd
```

它将会使得 nsqlookupd 在主机 IP 上的端口 5160 和 5161 可用。

使用 TLS

如果在 NSQ 容器上使用 TLS，你必须包含证书文件，私钥文件，和根 CA 文件。Docker 镜像里 /etc/ssl/certs/ 包含这些内容。挂载一个主机文件夹包含这些文件，并在命令行里指定，比如：

```
docker run -p 4150:4150 -p 4151:4151 -p 4152:4152 -v /home/docker/certs:/etc/ssl/certs \
  nsqio/nsq /nsqd \
  --tls-root-ca-file=/etc/ssl/certs/certs.crt \
  --tls-cert=/etc/ssl/certs/cert.pem \
  --tls-key=/etc/ssl/certs/key.pem \
  --tls-required=true \
  --tls-client-auth-policy=require-verify
```

上面的代码，运行的时候将会从 /home/docker/certs 里加载文件到 Docker 容器里。

持久化 NSQ 数据

使用 /data 目录来存储 nsqd 到主机磁盘上，它能让你加载到 [data-only Docker container](#)，或者加载主机文件夹里：

```
docker run nsqio/nsq /nsqd \
  --data-path=/data
```

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/nsq-guide/>