EMQ X Enterprise

产品手册

目录

——— 杭州映云科技有限公司 ———

www.emqx.cn

2021-05

1. 产品概览:

1.1 EMQ X 消息服务器简介

EMQ X (Erlang/Enterprise/Elastic MQTT Broker) 是基于 Erlang/OTP 平台开发的开源物

联网 MQTT 消息服务器。

Erlang/OTP 是出色的软实时 (Soft-Realtime)、低延时 (Low-Latency)、分布式 (Distributed)的语言平台。

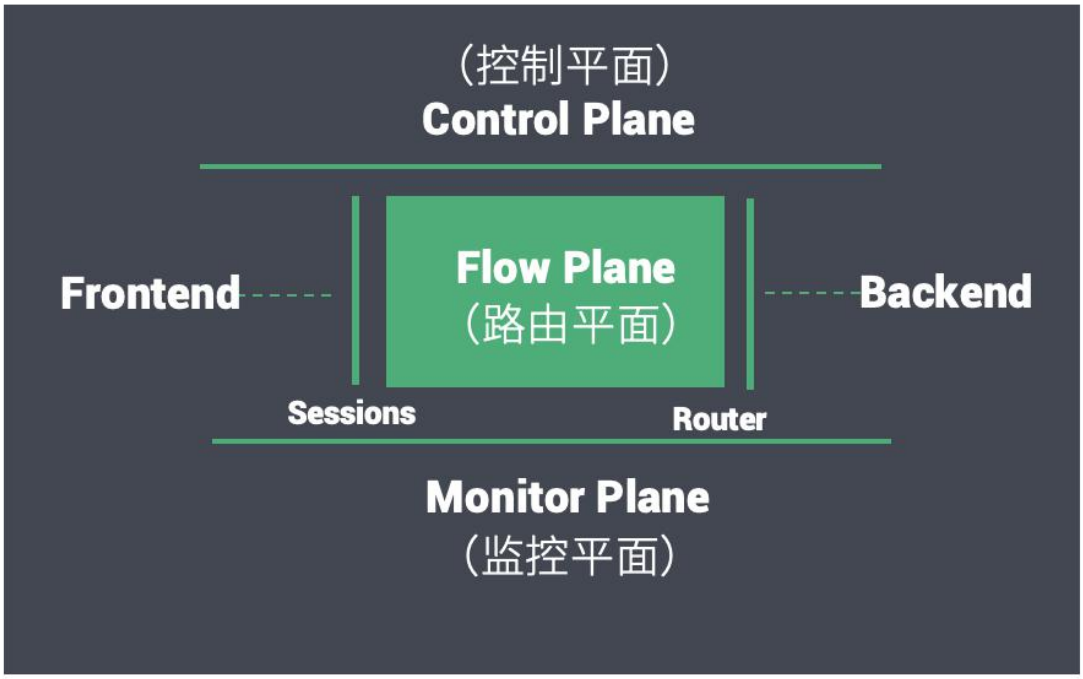
MQTT 是轻量的 (Lightweight)、发布订阅模式 (PubSub) 的物联网消息协议。

EMQ X 设计目标是实现高可靠，并支持承载海量物联网终端的 MQTT 连接，支持在海量物联网设备间低延时消息路由:

1) 稳定承载大规模的 MQTT 客户端连接，单服务器节点生产环境下支持 50 万到 100 万连接。
2) 分布式节点集群，快速低延时的消息路由，单集群支持 1000 万规模的路由。
3) 消息服务器内扩展，支持定制多种认证方式、高效存储消息到后端数据库。
4) 完整物联网协议支持，MQTT、MQTT-SN、CoAP、LwM2M、WebSocket 或私有协议支持。

1.2 系统架构设计：

EMQ X 在设计上，首先分离了前端协议 (FrontEnd) 与后端集成 (Backend)，其次分离了消息路由平面 (Flow Plane) 与监控管理平面 (Monitor/Control Plane):



1.2.1 全异步架构

EMQ X 是基于 Erlang/OTP 平台的全异步的架构：异步 TCP 连接处理、异步主题 (Topic) 订阅、异步消息发布。只有在资源负载限制部分采用同步设计，比如 TCP 连接创建和 Mnesia 数据库事务执行。

EMQ X 3.0 版本中，一条 MQTT 消息从发布者 (Publisher) 到订阅者 (Subscriber)，在 EMQ X Broker 内部异步流过一系列 Erlang 进程 Mailbox:

1.2.2 消息持久化

EMQ X 开源产品不支持服务器内部消息持久化，这是一个架构设计选择。首先，EMQ X 解决的核心问题是连接与路由；其次，我们认为内置持久化是个错误设计。

传统内置消息持久化的 MQ 服务器，比如广泛使用的 JMS 服务器 ActiveMQ，几乎每个大版本都在重新设计持久化部分。内置消息持久化在设计上有两个问题:

1) 如何权衡内存与磁盘的使用？消息路由是基于内存的，而消息存储是基于磁盘的。
2) 多服务器分布集群架构下，如何放置 Queue 如何复制 Queue 的消息？

Kafka 在上述问题上，做出了正确的设计：一个完全基于磁盘分布式 Commit Log 的消息服务器。

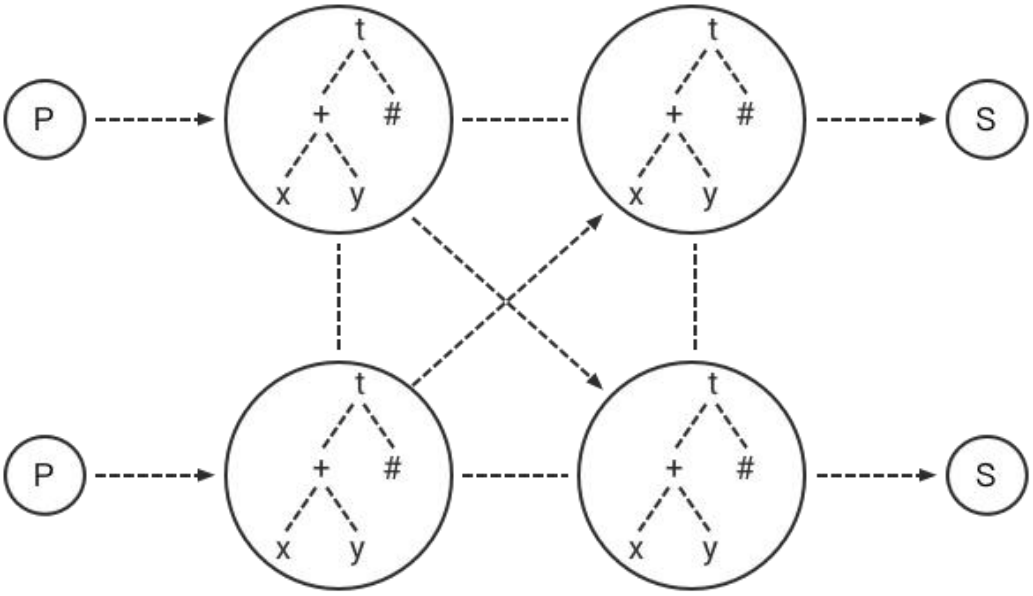
EMQ X 在设计上分离消息路由与消息存储职责后，数据复制容灾备份甚至应用集成，可以在数据层面灵活实现。

EMQ X 企业版产品中，可以通过规则引擎或插件的方式，持久化消息到 Redis、MongoDB、Cassandra、MySQL、PostgreSQL 等数据库，以及 RabbitMQ、Kafka 等消息队列。

1.2.3 系统架构模型

(1) 概念模型

EMQ X 概念上更像一台网络路由器 (Router) 或交换机 (Switch), 而不是传统的企业级消息队列 (MQ)。相比网络路由器按 IP 地址或 MPLS 标签路由报文，EMQ X 按主题树 (Topic Trie) 发布订阅模式在集群节点间路由 MQTT 消息:



（2）设计原则

1. EMQ X 核心解决的问题：处理海量的并发 MQTT 连接与路由消息。
2. 充分利用 Erlang/OTP 平台软实时、低延时、高并发、分布容错的优势。
3. 连接 (Connection)、会话 (Session)、路由 (Router)、集群 (Cluster) 分层。
4. 消息路由平面 (Flow Plane) 与控制管理平面 (Control Plane) 分离。
5. 支持后端数据库或 NoSQL 实现数据持久化、容灾备份与应用集成。

（3）系统分层

1. 连接层 (Connection Layer)：负责 TCP 连接处理、 MQTT 协议编解码。
2. 会话层 (Session Layer)：处理 MQTT 协议发布订阅消息交互流程。
3. 路由层 (Route Layer)：节点内路由派发 MQTT 消息。
4. 分布层 (Distributed Layer)：分布节点间路由 MQTT 消息。
5. 认证与访问控制 (ACL)：连接层支持可扩展的认证与访问控制模块。
6. 钩子 (Hooks) 与插件 (Plugins)：系统每层提供可扩展的钩子，支持插件方式扩展服务器。

1.2.4 连接层设计

连接层处理服务端 Socket 连接与 MQTT 协议编解码:

1. 基于 eSockd (opens new window)框架的异步 TCP 服务端
2. TCP Acceptor 池与异步 TCP Accept
3. TCP/SSL, WebSocket/SSL 连接支持
4. 最大并发连接数限制
5. 基于 IP 地址 (CIDR) 访问控制
6. 基于 Leaky Bucket 的流控
7. MQTT 协议编解码
8. MQTT 协议心跳检测
9. MQTT 协议报文处理

1.2.5 会话层设计

会话层处理 MQTT 协议发布订阅 (Publish/Subscribe) 业务交互流程:

1. 缓存 MQTT 客户端的全部订阅 (Subscription)，并终结订阅 QoS
2. 处理 QoS 0/1/2 消息接收与下发，消息超时重传与离线消息保存
3. 飞行窗口 (Inflight Window)，下发消息吞吐控制与顺序保证
4. 保存服务器发送到客户端的，已发送未确认的 QoS 1/2 消息
5. 缓存客户端发送到服务端，未接收到 PUBREL 的 QoS 2 消息
6. 客户端离线时，保存持久会话的离线 QoS 1/2 消息

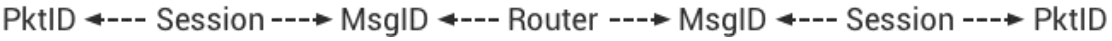
1.2.6 报文 ID 与消息 ID

MQTT 协议定义了一个 16bits 的报文 ID (PacketId)，用于客户端到服务器的报文收发与确认。MQTT 发布报文 (PUBLISH) 进入 Broker 后，转换为一个消息对象并分配 128bits 消息 ID (MessageId)。

全局唯一时间序列消息 ID 结构:

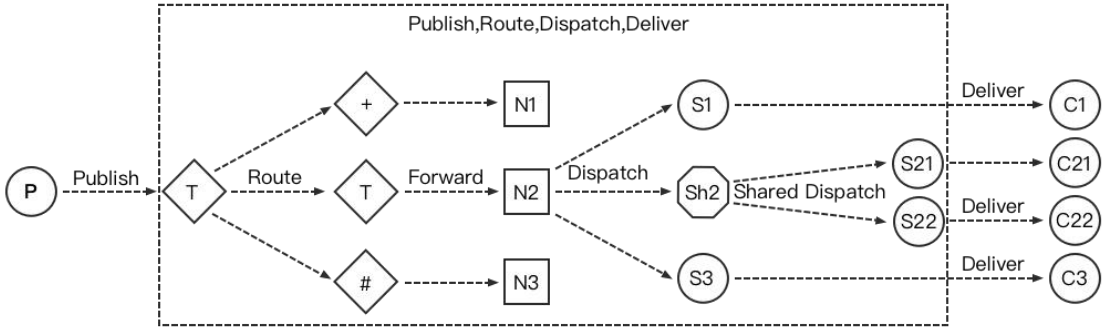
1. 64bits 时间戳: erlang:system_time if Erlang >= R18, otherwise os:timestamp
2. Erlang 节点 ID: 编码为 2 字节
3. Erlang 进程 PID: 编码为 4 字节
4. 进程内部序列号: 2 字节的进程内部序列号

端到端消息发布订阅 (Pub/Sub) 过程中，发布报文 ID 与报文 QoS 终结在会话层，由唯一 ID 标识的 MQTT 消息对象在节点间路由:



1.2.7 路由层设计

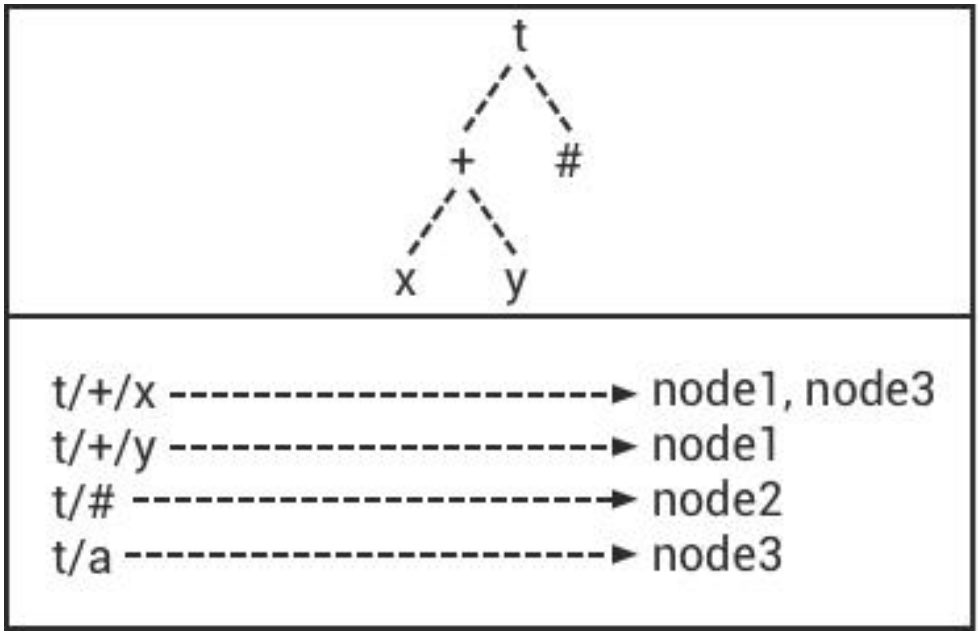
路由层维护订阅者 (Subscriber) 与订阅关系表 (Subscription)，并在本节点发布订阅模式派发 (Dispatch) 消息:



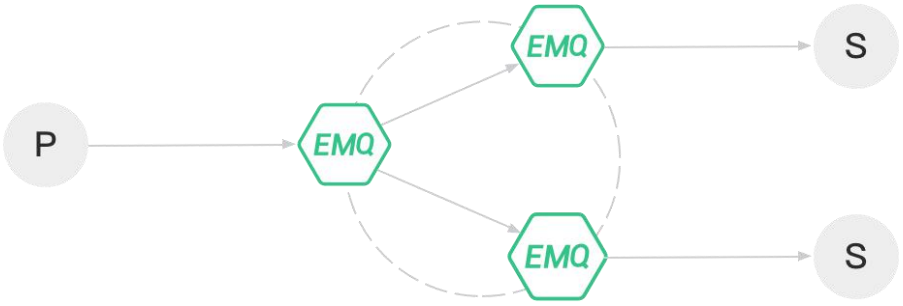
消息派发到会话 (Session) 后，由会话负责按不同 QoS 送达消息。

1.2.8 分布层设计

分布层维护全局主题树 (Topic Trie) 与路由表 (Route Table)。主题树由通配主题构成，路由表映射主题到节点:



分布层通过匹配主题树（Topic Trie）和查找路由表（Route Table），在集群的节点间转发路由 MQTT 消息:



1.2.9 Mnesia/ETS 表设计

Table	Type	Description
emqx_conn	ets	连接表

Table	Type	Description
emqx_metrics	ets	统计表
emqx_session	ets	会话表
emqx_hooks	ets	钩子表
emqx_subscriber	ets	订阅者表
emqx_subscription	ets	订阅表
emqx_admin	mnesia	Dashboard 用户表
emqx_retainer	mnesia	Retained 消息表
emqx_shared_subscription	mnesia	共享订阅表
emqx_session_registry	mnesia	全局会话注册表
emqx_alarm_history	mnesia	告警历史表
emqx_alarm	mnesia	告警表
emqx_banned	mnesia	禁止登陆表
emqx_route	mnesia	路由表
emqx_trie	mnesia	Trie 表

Table	Type	Description
emqx_trie_node	mnesia	Trie Node 表
mqtt_app	mnesia	App 表

1.2.10 Erlang 设计相关

1. 使用 Pool, Pool, Pool... 推荐 GProc 库: https://github.com/uwiger/gproc(opens new window)
2. 异步，异步，异步消息...连接层到路由层异步消息，同步请求用于负载保护
3. 避免进程 Mailbox 累积消息
4. 消息流经的 Socket 连接、会话进程必须 Hibernate，主动回收 binary 句柄
5. 多使用 Binary 数据，避免进程间内存复制
6. 使用 ETS, ETS, ETS... Message Passing vs. ETS
7. 避免 ETS 表非键值字段 select, match
8. 避免大量数据 ETS 读写，每次 ETS 读写会复制内存，可使用 lookup_element, update_counter
9. 适当开启 ETS 表 {write_concurrency, true}
10. 保护 Mnesia 数据库事务，尽量减少事务数量，避免事务过载(overload)
11. 避免对 Mnesia 数据表非索引、或非键值字段 match, select

2. 使用文档说明

建议您在使用前仔细阅读一遍下面列出的文档，未列出的其他文档可以按需选择查看：

（1）开始使用

- 安装：不同操作系统与安装包类型的下载、安装步骤。
- 启动 EMQ X：启动 EMQ X 并查看启动状态。

- Dashboard：通过 Dashboard 管理 EMQ X 及在线设备。

（2）认证鉴权

- 认证简介：选择内置插件、外部数据库、JWT 或者 HTTP 服务作为认证数据源，验证客户端连接合法性。
- 发布订阅 ACL：选择内置插件、外部数据库、或者 HTTP 服务作为 ACL 数据源，验证客户端发布订阅权限。
- 内置 ACL：内置 ACL 可能会影响到重要功能，使用前请详细了解。

（3）FAQ 常见问题解答

FAQ 常见问题解答定期收集整理 EMQ X 用户常见问题和经常遇到的错误，如 Topic 数量限制、开源版/企业版区别，企业服务收费等；开源版如何存储数据等。

（4）社区交流

- 资源：社区交流，包含社区热门教程、项目展示等资源。

（5）HTTP API

HTTP API 是物联网平台开发与 EMQ X 运维中频繁使用的功能，HTTP API 可以实现与外部系统的集成，例如查询并管理客户端信息、代理订阅、发布消息和创建规则等。

- HTTP API：包含 HTTP API 接入点、接入认证方式。
- 基本信息：获取 EMQ X 版本、运行状态等基本信息。
- 节点：获取 EMQ X 节点信息。
- 客户端：查看在线客户端信息，支持踢出客户端。
- 订阅信息：查看订阅主题列表与订阅关系。
- 路由：查看已订阅的主题。

- 消息发布：通过 HTTP 调用 EMQ X 发布 MQTT 消息，应用程序与客户端通信可靠的方式。
- 主题订阅：动态管理客户端订阅列表，无需客户端主动发起订阅/取消订阅。
- 插件：插件的状态管理，启动、停止操作。

其他更多 API 请通过左侧目录查看。

（6）规则引擎

规则引擎实现了消息数据与通过规则引擎能够筛选、处理、转发/存储消息到外部数据源，包括关系数据库、消息队列、Web 服务等等。

- 规则引擎：规则引擎的概念、基础使用方式。
- 创建规则：如何创建一条规则。
- 使用示例：规则引擎使用各类数据源的教程。

（7）数据存储

EMQ X 企业版特有功能，数据存储将客户端上下线状态，订阅关系，离线消息、消息内容，消息抵达后发送的消息回执等操作记录到各种数据库中。数据存储包含运行时数据与消息数据，能够在服务崩溃、客户端异常离线后仍然保留数据。

- 数据存储：基本概念与使用场景。
- 数据存储配置：使用不同的数据源进行数据存储。

（8）消息桥接

EMQ X 企业版桥接转发 MQTT 消息到 Kafka、RabbitMQ、Pulsar、RocketMQ、MQTT Broker 或其他 EMQ X 节点。

- MQTT 桥接：实现跨地域、跨集群部署。
- RPC 桥接
- Kafka 桥接

- RabbitMQ 桥接
- Pulsar 桥接
- RocketMQ 桥接

（9）运维部署

包含官方使用指南、最佳实践等信息。

- 设备管理
- 系统调优
- 生产部署
- Prometheus 监控告警
- 性能测试

（10）协议介绍

- MQTT 协议
- MQTT-SN 协议
- LwM2M 协议
- 私有 TCP 协议

2.2 EMQ X 消息服务器功能列表

（1）基础功能

- 完整的 MQTT V3.1/V3.1.1 及 V5.0 协议规范支持
    - QoS0, QoS1, QoS2 消息支持
    - 持久会话与离线消息支持
    - Retained 消息支持

- Last Will 消息支持

- TCP/SSL 连接支持
- MQTT/WebSocket/SSL 支持
- HTTP 消息发布接口支持
- $SYS/ 系统主题支持
- 客户端在线状态查询与订阅支持
- 客户端 ID 或 IP 地址认证支持
- 用户名密码认证支持
- LDAP、Redis、MySQL、PostgreSQL、MongoDB、HTTP 认证集成
- 浏览器 Cookie 认证
- 基于客户端 ID、IP 地址、用户名的访问控制（ACL）
- 多服务器节点集群（Cluster）
- 支持 manual、mcast、dns、etcd、k8s 等多种集群发现方式
- 网络分区自动愈合
- 消息速率限制
- 连接速率限制
- 按分区配置节点
- 多服务器节点桥接（Bridge）
- MQTT Broker 桥接支持
- Stomp 协议支持
- MQTT-SN 协议支持
- CoAP 协议支持
- LwM2M 协议支持
- Stomp/SockJS 支持
- 延时 Publish ($delay/topic)

- Flapping 检测
- 黑名单支持
- 共享订阅（$share/:group/topic）
- TLS/PSK 支持
- 规则引擎
    - 空动作（调试）
    - 消息重新发布
    - 桥接数据到 MQTT Broker
    - 检查（调试）
    - 发送数据到 Web 服务

(2) EMQ X Enterprise 特有功能

- Scalable RPC 架构: 分离 Erlang 自身的集群通道与 EMQ X 节点间的数据通道
- 数据持久化
    - Redis 存储订阅关系、设备在线状态、MQTT 消息、保留消息，发布 SUB/UNSUB 事件
    - MySQL 存储订阅关系、设备在线状态、MQTT 消息、保留消息
    - PostgreSQL 存储订阅关系、设备在线状态、MQTT 消息、保留消息
    - MongoDB 存储订阅关系、设备在线状态、MQTT 消息、保留消息
    - Cassandra 存储订阅关系、设备在线状态、MQTT 消息、保留消息
    - DynamoDB 存储订阅关系、设备在线状态、MQTT 消息、保留消息
    - InfluxDB 存储 MQTT 时序消息
    - OpenTDSB 存储 MQTT 时序消息
    - TimescaleDB 存储 MQTT 时序消息
- 消息桥接

    - Kafka 桥接：EMQ X 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 Kafka
    - RabbitMQ 桥接：EMQ X 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 RabbitMQ
    - Pulsar 桥接：EMQ X 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 Pulsar
    - RocketMQ 桥接：EMQ X 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 RocketMQ
- 规则引擎
    - 消息编解码
    - 桥接数据到 Kafka
    - 桥接数据到 RabbitMQ
    - 桥接数据到 RocketMQ
    - 桥接数据到 Pulsar
    - 保存数据到 PostgreSQL
    - 保存数据到 MySQL
    - 保存数据到 OpenTSDB
    - 保存数据到 Redis
    - 保存数据到 DynamoDB
    - 保存数据到 MongoDB
    - 保存数据到 InfluxDB
    - 保存数据到 Timescale
    - 保存数据到 Cassandra

    - 保存数据到 ClickHouse
    - 保存数据到 TDengine
    - 离线消息保存到 Redis
    - 离线消息保存到 MySQL
    - 离线消息保存到 PostgreSQL
    - 离线消息保存到 Cassandra
    - 离线消息保存到 MongoDB
    - 从 Redis 中获取订阅关系
    - 从 MySQL 中获取订阅关系
    - 从 PostgreSQL 中获取订阅关系
    - 从 Cassandra 中获取订阅关系
    - 从 MongoDB 中获取订阅关系
- Schema Registry：将 EMQ X 的事件、消息 提供了数据编解码能力

2.3 功能说明

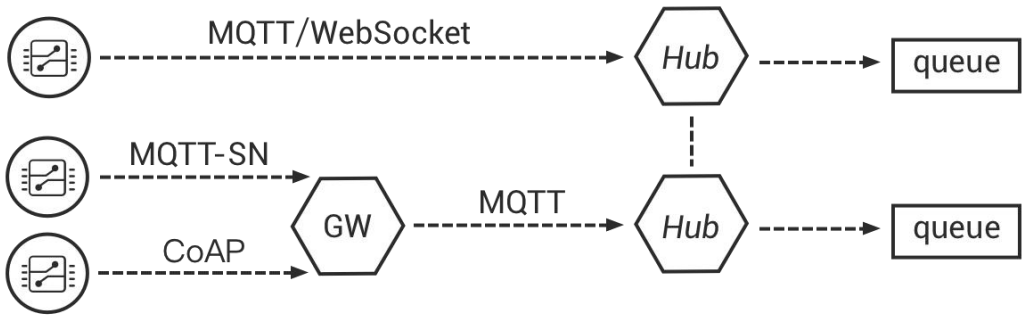
2.3.1 消息数据存储

EMQ X 企业版支持存储订阅关系、MQTT 消息、设备状态到 Redis、MySQL、PostgreSQL、MongoDB、Cassandra、TimescaleDB、InfluxDB、DynamoDB、OpenTDSB 数据库:

数据存储相关配置，详见"数据存储"章节。

2.3.2 消息桥接转发

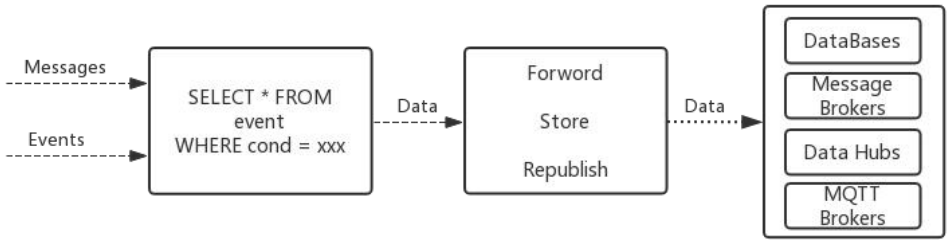
EMQ X 企业版支持直接转发 MQTT 消息到 RabbitMQ、Kafka、Pulsar、RocketMQ、MQTT Broker，可作为百万级的物联网接入服务器(IoT Hub):



2.3.3 规则引擎

EMQ X 规则引擎可以灵活地处理消息和事件。

EMQ X 企业版规则引擎支持消息重新发布；桥接数据到 Kafka、Pulsar、RocketMQ、RabbitMQ、MQTT Broker；保存数据到 MySQL、PostgreSQL、Redis、MongoDB、DynamoDB、Cassandra、InfluxDB、OpenTSDB、TimescaleDB；发送数据到 WebServer:

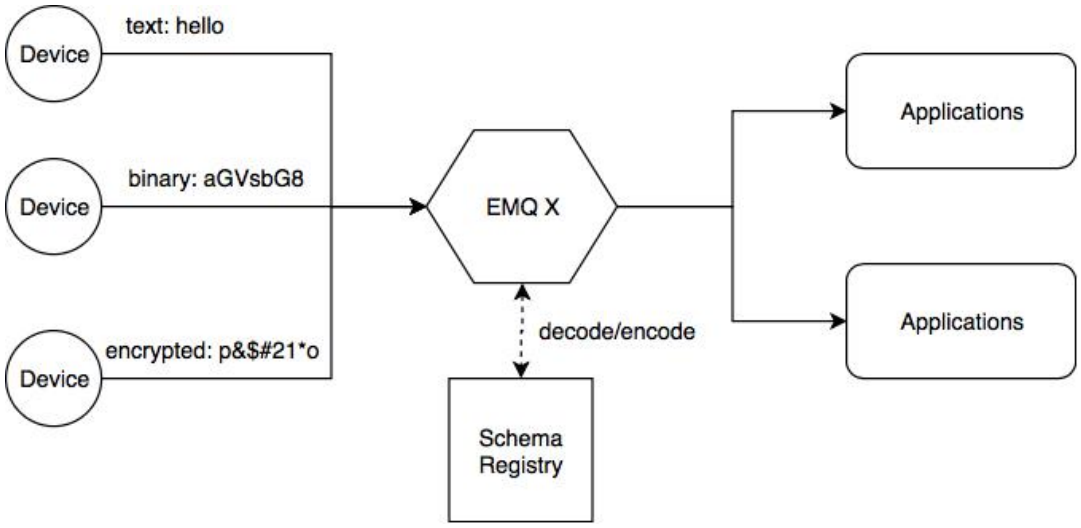


规则引擎相关配置，详见"规则引擎"章节。

2.3.4 编解码

Schema Registry 目前可支持三种格式的编解码：Avro (opens new window)，Protobuf (opens

new window)，以及自定义编码。其中 Avro 和 Protobuf 是依赖 Schema 的数据格式，编码后的数据为二进制，解码后为 Map 格式 。解码后的数据可直接被规则引擎和其他插件使用。用户自定义的 (3rd-party) 编解码服务通过 HTTP 或 TCP 回调的方式，进行更加贴近业务需求的编解码。



编解码相关配置，详见"编解码"章节。

3. 开始使用 EMQX

3.1 安装

1）EMQ X 企业版安装

EMQ X 消息服务器可跨平台运行在 Linux、FreeBSD、macOS、Windows 或 openSUSE 服务器上。

WARNING:产品部署建议 Linux 服务器，不推荐 Windows 服务器。

2）EMQ X License 文件获取

当您从云市场下单购买后，License 文件会以邮件的方式发送到邮箱。

3）EMQ X 程序包下载

EMQ X 消息服务器每个版本会发布 CentOS、Ubuntu、Debian、FreeBSD、Windows 、openSUSE 平台程序包与 Docker 镜像。

下载地址: https://www.emqx.cn/downloadsenterprise(opens new window)

（1）CentOS 下安装 EMQX

- CentOS6.X
- CentOS7.X

1）使用储存库(YUM)安装 EMQ X

1. **删除旧的 EMQ X**

   $ sudo yum remove emqx emqx-edge emqx-ee

2. **安装所需要的依赖包**

   $ sudo yum install -y yum-utils device-mapper-persistent-data lvm2

3. **使用以下命令设置 yum 库，以 CentOS7 为例**

   $sudo yum-config-manager --add-repo

   https://repos.emqx.io/emqx-ee/redhat/centos/7/emqx-ee.repo

4. **安装最新版本的 EMQ X**

   $ sudo yum install emqx-ee

5. **安装特定版本的 EMQ X**

   查询可用版本

```sh
$ yum list emqx-ee --showduplicates | sort -r

emqx-ee.x86_64                    4.0.0-1.el7
```

根据第二列中的版本字符串安装特定版本，例如 4.0.0

```sh
$ sudo yum install emqx-ee-4.0.0
```

导入 License 文件:

```
$ cp /path/to/emqx.lic /etc/emqx/emqx.lic
```

6. 启动 EMQ X

- 直接启动

```
$ emqx start
emqx v4.0.0 is started successfully!

$ emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

- systemctl 启动

```
$ sudo systemctl start emqx
```

- service 启动

```
$ sudo service emqx start
```

2）使用 rpm 包安装 EMQ X

通过 emqx.io (opens new window)选择 CentOS 版本，然后下载要安装的 EMQ X 版本的 rpm 包。

- 安装 EMQ X

$ sudo rpm -ivh emqx-ee-centos7-v4.0.0.x86_64.rpm

- 导入 License 文件:

$ cp /path/to/emqx.lic /etc/emqx/emqx.lic

启动 EMQ X

- 直接启动

```
$ emqx start
emqx v4.0.0 is started successfully!

$ emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

- systemctl 启动

```
$ sudo systemctl start emqx
```

- service 启动

```
$ sudo service emqx start
```

3）使用 zip 包安装 EMQ X

通过 emqx.io (opens new window)选择 Centos 版本，然后下载要安装的 EMQ X 版本的 zip 包。

- **解压程序包**

$ unzip emqx-ee-centos7-v4.0.0.zip

- **导入 License 文件:**

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

- **启动 EMQ X**

$ ./bin/emqx start

emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（2）Ubuntu 下安装 EMQ X

- Bionic 18.04 (LTS)
- Xenial 16.04 (LTS)
- Trusty 14.04 (LTS)
- Precise 12.04 (LTS)

1）使用储存库 apt-get 安装 EMQ X

- 删除旧的 EMQ X

$ sudo apt remove emqx emqx-edge emqx-ee

- 安装所需要的依赖包

$ sudo apt update && sudo apt install -y \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common

- 添加 EMQ X 的官方 GPG 密钥

$ curl -fsSL https://repos.emqx.io/gpg.pub | sudo apt-key add -

- 验证密钥

$ sudo apt-key fingerprint 3E640D53

```
$ sudo apt-key fingerprint 3E640D53

pub   rsa2048 2019-04-10 [SC]
    FC84 1BA6 3775 5CA8 487B  1E3C C0B4 0946 3E64 0D53
uid           [ unknown] emqx team <support@emqx.io>
```

- 添加 EMQ X 存储库。

$ sudo add-apt-repository \

"deb [arch=amd64] https://repos.emqx.io/emqx-ee/deb/ubuntu/ \

$(lsb_release -cs) \

stable"

- 更新 apt 包索引

$ sudo apt update

- 安装最新版本的 EMQ X

$ sudo apt install emqx-ee

- 安装特定版本的 EMQ X
- 查询可用版本

$ sudo apt-cache madison emqx-ee

emqx-ee | 4.0.0 | https://repos.emqx.io/emqx-ee/deb/ubuntu bionic/stable amd64 Packages

- 使用第二列中的版本字符串安装特定版本，例如

$ sudo apt install emqx-ee=4.0.0

- 导入 License 文件:

```
$ cp /path/to/emqx.lic /etc/emqx/emqx.lic1
```

- 启动 EMQ X
- 直接启动

```
$ emqx start
  emqx v4.0.0 is started successfully!
$ emqx_ctl status
  Node 'emqx@127.0.0.1' is started
  emqx 4.0.0 is running
```

- systemctl 启动

```
$ sudo systemctl start emqx
```

- service 启动

```
$ sudo service emqx start
```

2）使用 deb 包安装 EMQ X

通过 emqx.io (opens new window)选择 Ubuntu 版本，然后下载要安装的 EMQ X 版本的 deb 包。

- 安装 EMQ X

```
$ sudo dpkg -i emqx-ee-ubuntu18.04-v3.1.0_amd64.deb
```

- 导入 License 文件:

```
$ cp /path/to/emqx.lic /etc/emqx/emqx.lic
```

- 启动 EMQ X
- 直接启动

```
$ emqx start
emqx    is started successfully!
$ emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

- systemctl 启动

```
$ sudo systemctl start emqx
```

- service 启动

$ sudo service emqx start

3）使用 zip 包安装 EMQ X

通过 emqx.io (opens new window)选择 Ubuntu 版本，然后下载要安装的 EMQ X 版本的 zip 包。

- 解压程序包

$ unzip emqx-ee-ubuntu18.04-v4.0.0.zip

- 导入 License 文件:

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

- 启动 EMQ X

$ ./bin/emqx start

emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（3） Debian 下安装 EMQX

- Stretch (Debian 9)
- Jessie (Debian 8)

1）使用储存库 apt-get 安装 EMQ X

- 删除旧的 EMQ X

$ sudo apt remove emqx emqx-edge emqx-ee

- 安装所需要的依赖包

$ sudo apt update && sudo apt install -y \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common

- 添加 EMQ X 的官方 GPG 密钥

```
$ curl -fsSL https://repos.emqx.io/gpg.pub | sudo apt-key add -
```

- 验证密钥

```
$ sudo apt-key fingerprint 3E640D53
```

- 设置 EMQ X 存储库。

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://repos.emqx.io/emqx-ee-ce/deb/debian/ \
    $(lsb_release -cs) \
    stable"
```

- 更新 apt 包索引

```
$ sudo apt update
```

- 安装最新版本的 EMQ X

```
$ sudo apt install emqx-ee
```

- 安装特定版本的 EMQ X
  - 查询可用版本

  ```
  $ sudo apt-cache madison emqx-ee
  ```

  - 使用第二列中的版本字符串安装特定版本，例如

  ```
  $ sudo apt install emqx-ee=4.0.0
  ```

  - 导入 License 文件：

  ```
  $ cp /path/to/emqx.lic /etc/emqx/emqx.lic
  ```

- 启动 EMQ X
  - 直接启动

  ```
  $ emqx start
  emqx v4.0.0 is started successfully!
  $ emqx_ctl status
  Node 'emqx@127.0.0.1' is started
  emqx 4.0.0 is running
  ```

  - systemctl 启动

  ```
  $ sudo systemctl start emqx
  ```

- service 启动

$ sudo service emqx start

2）使用 deb 包安装 EMQ X

通过 emqx.io (opens new window)选择 Debian 版本，然后下载要安装的 EMQ X 版本的 deb 包。

- 安装 EMQ X

$ sudo dpkg -i emqx-ee-debian9-v3.1.0_amd64.deb

- 导入 License 文件：

$ cp /path/to/emqx.lic /etc/emqx/emqx.lic

- 启动 EMQ X
- 直接启动

$ emqx start

emqx v4.0.0 is started successfully!

$ emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

- systemctl 启动

$ sudo systemctl start emqx

- service 启动

$ sudo service emqx start

3）使用 zip 包安装 EMQ X

通过 emqx.io (opens new window)选择 Debian 版本，然后下载要安装的 EMQ X 版本的 zip 包。

- 解压程序包

$ unzip emqx-ee-debian9-v4.0.0.zip

- 导入 License 文件：

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

- 启动 EMQ X

$ ./bin/emqx start

emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（4）macOS 下安装 EMQ X

1）使用 ZIP 包安装 EMQ X

通过 emqx.io (opens new window)，选择 EMQ X 版本，然后下载要安装的 zip 包。

- 解压压缩包

$ unzip emqx-ee-macos-v4.0.0.zip

- 导入 License 文件:

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.li

- 启动 EMQ X

$ ./bin/emqx start

emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（5）Windows 下安装 EMQ X （能力满足不推荐）

通过 emqx.io (opens new window)下载 .zip 包。解压压缩包

- 导入 License 文件:

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

- 打开 Windows 命令行窗口，cd 到程序目录， 启动 EMQ X。

cd /path/to/emqx/bin

emqx start

（6）openSUSE 下安装 EMQ X

- openSUSE leap

1）使用储存库 zypper 安装 EMQ X

- 删除旧的 EMQ X

```
$ sudo zypper remove emqx emqx-edge emqx-ee
```

- 下载 GPG 公钥并导入。

```
$ curl -L -o /tmp/gpg.pub https://repos.emqx.io/gpg.pub
$ sudo rpmkeys --import /tmp/gpg.pub
```

- 添加储存库地址

```
$ sudo zypper ar -f -c https://repos.emqx.io/emqx-ee/redhat/opensuse/leap/stable emqx-ee
```

- 安装最新版本的 EMQ X

```
$ sudo zypper in emqx-ee
```

- 安装特定版本的 EMQ X

  查询可用版本

  ```
  $ sudo zypper pa emqx-ee
  Loading repository data...
  Reading installed packages...
  ```

  使用 Version 安装特定版本，例如

  ```
  $ sudo zypper in emqx-ee-4.0.0
  ```

- 导入 License 文件:

```
$ cp /path/to/emqx.lic /etc/emqx/emqx.lic
```

- 启动 EMQ X
  - 直接启动

  ```
  $ emqx start
  emqx v4.0.0 is started successfully!
  $ emqx_ctl status
  Node 'emqx@127.0.0.1' is started
  emqx 4.0.0 is running
  ```

  - systemctl 启动

  ```
  $ sudo systemctl start emqx
  ```

  - service 启动

  ```
  $ sudo service emqx start
  ```

2）使用 rpm 包安装 EMQ X

通过 emqx.io (opens new window)选择 openSUSE，然后下载要安装的 EMQ X 版本的 rpm 包。

- 安装 EMQ X，将下面的路径更改为您下载 EMQ X 软件包的路径。

```
$ sudo rpm -ivh emqx-ee-opensuse-v4.0.0.x86_64.rpm
```

- 导入 License 文件：

```
$ cp /path/to/emqx.lic /etc/emqx/emqx.lic
```

- 启动 EMQ X

- 直接启动

```
$ emqx start
  emqx v4.0.0 is started successfully!
$ emqx_ctl status
  Node 'emqx@127.0.0.1' is started
  emqx 4.0.0 is running
```

- systemctl 启动

```
$ sudo systemctl start emqx
```

- service 启动

```
$ sudo service emqx start
```

3）使用 zip 包安装 EMQ X

通过 emqx.io (opens new window)选择 openSUSE，然后下载要安装的 EMQ X 版本的 zip 包。

- 解压压缩包

```
$ unzip emqx-ee-opensuse-v4.0.0.zip
```

- 导入 License 文件：

```
$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic
```

- 启动 EMQ X

```
$ ./bin/emqx start
emqx v4.0.0 is started successfully!
```

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（7）FreeBSD

- FreeBSD 12

1）使用 zip 包安装 EMQ X

通过 emqx.io (opens new window)选择 FreeBSD，然后下载要安装的 EMQ X 版本的 zip 包。

- 解压压缩包

$ unzip emqx-ee-freebsd12-v4.0.0.zip

- 导入 License 文件:

$ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

- 启动 EMQ X

$ ./bin/emqx start

emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status

Node 'emqx@127.0.0.1' is started

emqx 4.0.0 is running

（8）Docker

- 获取 docker 镜像

通过 Docker Hub (opens new window)获取

$ docker pull emqx/emqx-ee:v4.0.0

通过 emqx.io (opens new window)手动下载 docker 镜像，并手动加载

$ wget -O emqx-ee-docker-v4.0.0.zip

https://www.emqx.io/downloads/enterprise/v4.0.0/emqx-ee-docker-v4.0.0-amd64.zip

$ unzip emqx-ee-docker.zip

$ docker load < emqx-ee-docker-v4.0.0

- 启动 docker 容器

$ docker run -d -\

-name emqx-ee \

-p 1883:1883 \

-p 8083:8083 \

-p 8883:8883 \

-p 8084:8084 \

-p 18083:18083 \

-v /path/to/emqx.lic:/opt/emqx/etc/emqx.lic

emqx/emqx-ee:v4.0.0

更多关于 EMQ X Docker 的信息请查看 Docker Hub

3.2 基本命令

EMQ X 提供了 emqx 命令行工具，方便用户对 EMQ X 进行启动、关闭、进入控制台等操作。

1）后台启动 EMQ X Broker

emqx start

2）关闭 EMQ X Broker；

emqx stop

3）重启 EMQ X Broker；

emqx restart

4）使用控制台启动 EMQ X Broker；

emqx console

5）使用控制台启动 EMQ X Broker

与 emqx console 不同，emqx foreground 不支持输入 Erlang 命令；

emqx foreground

6）Ping EMQ X Broker

emqx ping

以上命令为用户常用命令，此外 emqx 命令还有一些其他选项为方便开发者使用，普通用户

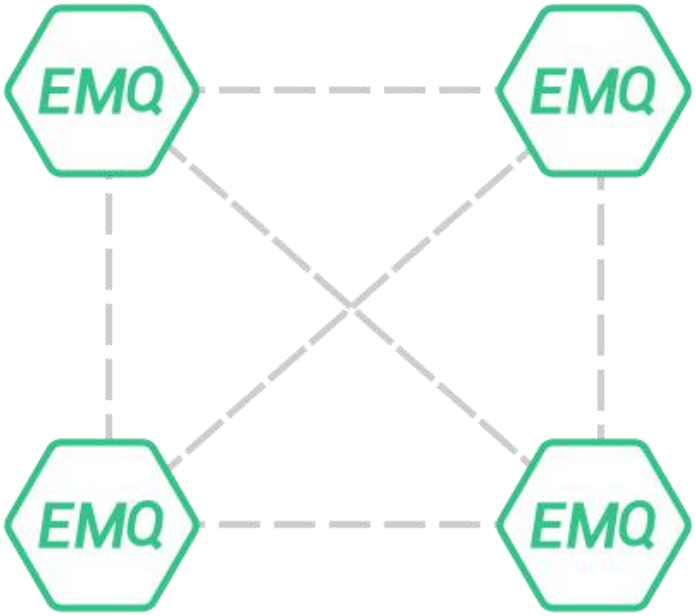
无需关心。

3.3 分布式集群

Erlang/OTP 分布式编程

Erlang/OTP 最初是爱立信为开发电信设备系统设计的编程语言平台，电信设备(路由器、接入网关等)典型设计是通过背板连接主控板卡与多块业务板卡的分布式系统。

Erlang/OTP 语言平台的分布式程序，由分布互联的 Erlang 运行系统组成，每个 Erlang 运行系统被称为节点(Node)，节点(Node) 间通过 TCP 互联，消息传递的方式通信:



1) 节点(Node)

Erlang 节点由唯一的节点名称标识，节点间通过名称进行通信寻址。 例如在本机启动四个 Erlang 节点，节点名称分别为:

```
erl -name node1@127.0.0.1
erl -name node2@127.0.0.1
erl -name node3@127.0.0.1
erl -name node4@127.0.0.1
```

node1@127.0.0.1 控制台下建立与其他节点的连接:

```
(node1@127.0.0.1)1> net_kernel:connect_node('node2@127.0.0.1').
true
(node1@127.0.0.1)2> net_kernel:connect_node('node3@127.0.0.1').
true
(node1@127.0.0.1)3> net_kernel:connect_node('node4@127.0.0.1').
true
(node1@127.0.0.1)4> nodes().
['node2@127.0.0.1','node3@127.0.0.1','node4@127.0.0.1']
```

(node1@127.0.0.1)4> nodes().

['node2@127.0.0.1','node3@127.0.0.1','node4@127.0.0.1']

2）EMQ X 分布集群设计

EMQ X 消息服务器集群基于 Erlang/OTP 分布式设计，集群原理可简述为下述两条规则:

- MQTT 客户端订阅主题时，所在节点订阅成功后广播通知其他节点：某个主题(Topic)被本节点订阅。
- MQTT 客户端发布消息时，所在节点会根据消息主题(Topic)，检索订阅并路由消息到相关节点。

EMQ X 消息服务器同一集群的所有节点，都会复制一份主题(Topic) -> 节点(Node)映射的路由表，例如:

```
topic1 -> node1, node2
topic2 -> node3
topic3 -> node2, node4
```

3）主题树(Topic Trie)与路由表(Route Table)

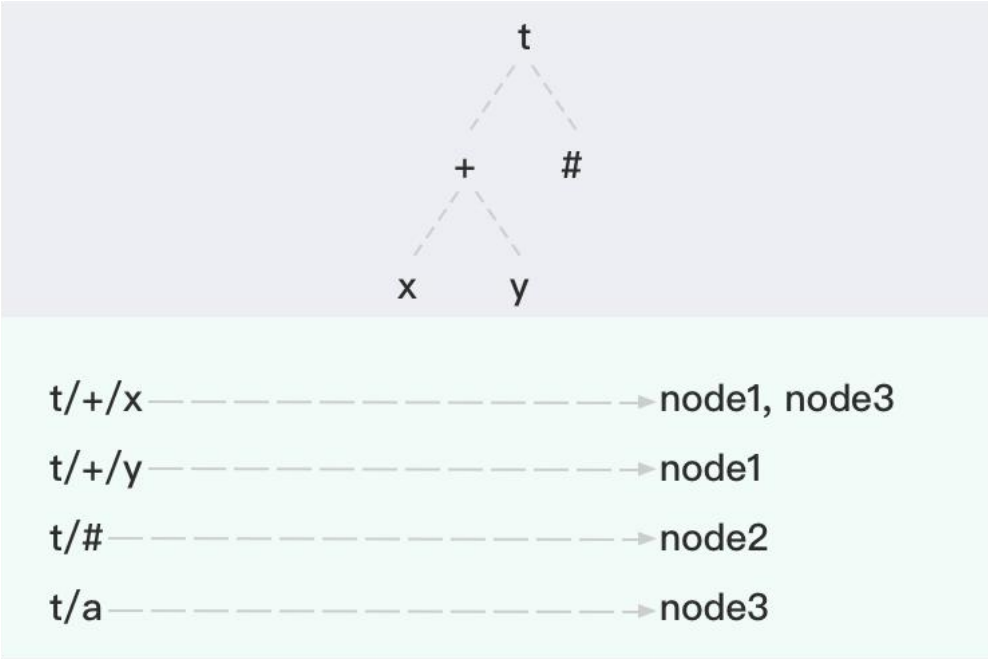
EMQ X 消息服务器每个集群节点，都保存一份主题树(Topic Trie)和路由表。

例如下述主题订阅关系:

客户端　节点　　订阅主题

客户端	节点	订阅主题
client1	node1	t/+/x, t/+/y
client2	node2	t/#
client3	node3	t/+/x, t/a

最终会生成如下主题树(Topic Trie)和路由表(Route Table):



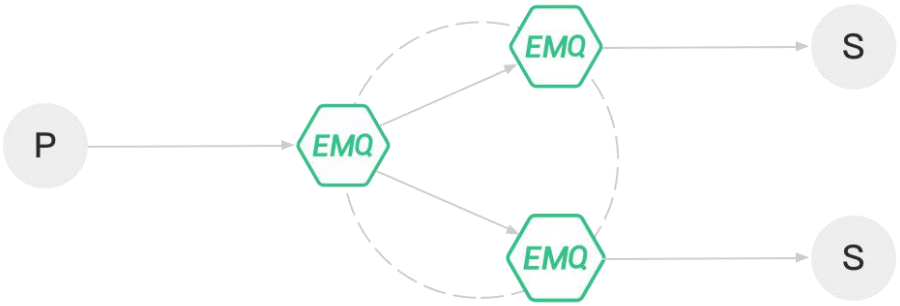
4）订阅(Subscription)与消息派发

客户端的主题订阅(Subscription)关系，只保存在客户端所在节点，用于本节点内派发消息到客户端。

例如 client1 向主题’t/a’发布消息，消息在节点间的路由与派发流程:

```
title: Message Route and Deliver

client1 -> node1: Publish[t/a]
    node1 --> node2: Route[t/#]
        node2 --> client2: Deliver[t/#]
    node1 --> node3: Route[t/a]
        node3 --> client3: Deliver[t/a]
```



5）节点发现与自动集群

EMQ X 支持基于 Ekka 库的集群自动发现 (Autocluster)。Ekka 是为 Erlang/OTP 应用开发的集群管理库，支持 Erlang 节点自动发现 (Service Discovery)、自动集群 (Autocluster)、脑裂自动愈合 (Network Partition Autoheal)、自动删除宕机节点 (Autoclean)。

EMQ X 支持多种节点发现策略:

策略	说明
manual	手动命令创建集群
static	静态节点列表自动集群
mcast	UDP 组播方式自动集群
dns	DNS A 记录自动集群
etcd	通过 etcd 自动集群
k8s	Kubernetes 服务自动集群

6）手动(manual) 方式管理集群介绍

假设要在两台服务器 s1.emqx.io, s2.emqx.io 上部署 EMQ X 集群:

节点名 主机名 (FQDN) IP 地址

节点名	主机名 (FQDN)	IP 地址
emqx@s1.emqx.io 或 emqx@192.168.0.10	s1.emqx.io	192.168.0.10
emqx@s2.emqx.io 或 emqx@192.168.0.20	s2.emqx.io	192.168.0.20

注意： 节点名格式为 Name@Host, Host 必须是 IP 地址或 FQDN (主机名。域名)

- 配置 emqx@s1.emqx.io 节点
  - 配置文件

emqx/etc/emqx.conf:

node.name = emqx@s1.emqx.io

或

node.name = emqx@192.168.0.10

  - 也可通过环境变量:

export EMQX_NODE_NAME=emqx@s1.emqx.io && ./bin/emqx start

注意：节点启动加入集群后，节点名称不能变更。

- 配置 emqx@s2.emqx.io 节点

emqx/etc/emqx.conf:

node.name = emqx@s2.emqx.io

或

node.name = emqx@192.168.0.20

7）节点加入集群

启动两台节点后，在 s2.emqx.io 上执行:

$ ./bin/emqx_ctl cluster join emqx@s1.emqx.io

Join the cluster successfully.

Cluster status: [{running_nodes,['emqx@s1.emqx.io','emqx@s2.emqx.io']}]

注意: s2.emqx.io 加入集群后会清除本身全部的数据，同步 s1.emqx.io 节点的数据。如果还有 s3.emqx.io 节点，那么需要在 s3.emqx.io 节点去执行命令加入 emqx@s1.emqx.io 或者 emqx@s2.emqx.io， 已经在集群的节点不能在 join 到其他节点，否则会退出当前集群和 join 的节点组成一个新的集群

8）在任意节点上查询集群状态:

$ ./bin/emqx_ctl cluster status

Cluster status: [{running_nodes,['emqx@s1.emqx.io','emqx@s2.emqx.io']}]

9）退出集群

节点退出集群，两种方式:

- leave：让本节点退出集群
- force-leave：从集群删除其他节点

让 emqx@s2.emqx.io 主动退出集群:

$ ./bin/emqx_ctl cluster leave

或在 s1.emqx.io 上，从集群删除 emqx@s2.emqx.io 节点:

$ ./bin/emqx_ctl cluster force-leave emqx@s2.emqx.io

10）防火墙设置

若预先设置了环境变量 WITH_EPMD=1，启动 emqx 时会使用启动 epmd (监听端口 4369) 做节点发现。称为 epmd 模式。 若环境变量 WITH_EPMD 没有设置，则启动 emqx 时不启用 epmd，而使用 emqx ekka 的节点发现，这也是 4.0 之后的默认节点发现方式。称为 ekka 模式。

- epmd 模式： 如果集群节点间存在防火墙，防火墙需要开启 TCP 4369 端口和一个 TCP 端口段。4369 由 epmd 端口映射服务使用，TCP 端口段用于节点间建立连接与通信。

防火墙设置后，需要在 emqx/etc/emqx.conf 中配置相同的端口段:

Distributed node port range

node.dist_listen_min = 6369

node.dist_listen_max = 7369

- ekka 模式（4.0 版本之后的默认模式）：

如果集群节点间存在防火墙，默认情况下，只需要开启 TCP 4370 端口。

但如果 node.name 配置制定的节点名字里，带有数字后缀(Offset)，则需要开启 4370 + Offset 端口。

比如:

node.name = emqx-1@192.168.0.12

则需要开启 4371 端口。

3.4 目录结构

不同安装方式得到的 EMQ X 其目录结构会有所不同，具体如下:

描述	使用 ZIP 压缩包安装	使用二进制包安装
可执行文件目录	./bin	/usr/lib/emqx/bin
数据文件	./data	/var/lib/emqx/data

描述	使用 ZIP 压缩包安装	使用二进制包安装
Erlang 虚拟机文件	./erts-*	/usr/lib/emqx/erts-*
配置文件目录	./etc	/etc/emqx/etc
依赖项目录	./lib	/usr/lib/emqx/lib
日志文件	./log	/var/log/emqx
启动相关的脚本、schema 文件	./releases	/usr/lib/emqx/releases

以上目录中，用户经常接触与使用的是 bin、etc、data、log 目录。

（1）bin 目录

emqx、emqx.cmd

EMQ X 的可执行文件，具体使用可以查看 基本命令。

emqx_ctl、emqx_ctl.cmd

EMQ X 管理命令的可执行文件，具体使用可以查看 管理命令 CLI。

（2）etc 目录

EMQ X 通过 etc 目录下配置文件进行设置，主要配置文件包括:

配置文件	说明
emqx.conf	EMQ X 配置文件
acl.conf	EMQ X 默认 ACL 规则配置文件
plugins/*.conf	EMQ X 各类插件配置文件
certs	EMQ X SSL 证书文件

配置文件	说明
emqx.lic	License 文件

EMQ X 具体的配置内容可以查看 配置项。

（3）data 目录

EMQ X 将运行数据存储在 data 目录下，主要的文件包括:

1）configs/app.*.config

EMQ X 读取 etc/emqx.conf 和 etc/plugins/*.conf 中的配置后，转换为 Erlang 原生配置文件格式，并在运行时读取其中的配置。

2）loaded_plugins

loaded_plugins 文件记录了 EMQ X 默认启动的插件列表，可以修改此文件以增删默认启动的插件。loaded_plugins 中启动项格式为 {<Plugin Name>, <Enabled>}.，<Enabled> 字段为布尔类型，EMQ X 会在启动时根据 <Enabled> 的值判断是否需要启动该插件。关于插件的更多内容，请查看 插件。

```
$ cat loaded_plugins
{emqx_management, true}.
{emqx_dashboard, true}.
{emqx_schema_registry, true}.
{emqx_rule_engine, true}.
```

（4）Mnesia 数据库

Mnesia 数据库是 Erlang 内置的一个分布式 DBMS，可以直接存储 Erlang 的各种数据结构。

EMQ X 使用 Mnesia 数据库存储自身运行数据，例如告警记录、规则引擎已创建的资源和规则、Dashbaord 用户信息等数据，这些数据都将被存储在 mnesia 目录下，因此一旦删除该目录，将导致 EMQ X 丢失所有业务数据。

可以通过 emqx_ctl mnesia 命令查询 EMQ X 中 Mnesia 数据库的系统信息，具体请查看 管理命令 CLI。

（5）log 目录

- emqx.log.*

EMQ X 运行时产生的日志文件，具体请查看 日志与追踪。

- crash.dump

EMQ X 的崩溃转储文件，可以通过 etc/emqx.conf 修改配置，具体内容可以查看 配置项。

- erlang.log.*

以 emqx start 方式后台启动 EMQ X 时，控制台日志的副本文件。

3.5 配置说明

（1）简介

EMQ X 的配置文件通常以 .conf 作为后缀名，你可以在 etc 目录找到这些配置文件，主要配置文件包括：

配置文件	说明
etc/emqx.conf	EMQ X 基础配置文件
etc/cluster.conf	EMQ X 集群相关配置文件
etc/rpc.conf	EMQ X 远程调用配置文件
etc/logger.conf	EMQ X 日志配置文件
etc/zones.conf	EMQ X Zone 配置文件
etc/listeners.conf	EMQ X 监听端口配置文件
etc/sys_mon.conf	EMQ X 告警监控配置文件

配置文件	说明
etc/acl.conf	EMQ X 默认 ACL 规则配置文件
etc/plugins/*.conf	EMQ X 扩展插件配置文件

需要注意的是，安装方式不同 etc 目录所处的路径可能不同，具体请参见 目录结构。

（2）语法规则

- 采用类似 sysctl 的 k = v 通用格式
- 单个配置项的所有信息都在同一行内，换行意味着创建一个新的配置项
- 键可以通过 . 进行分层，支持按树形结构管理配置项
- 值的类型可以

  是 integer, fload, percent, enum, ip, string, atom, flag, duration and bytesize
- 任何以 # 开头的行均被视为注释

示例：

```
mqtt.max_packet_size = 1MB
```

（3）数据类型

1）**integer**：整型数据。

2）**float**：浮点型数据。

3）**percent**：以 % 结尾的百分比数据，最终会被转换为 **float** 类型。

4）**enum**：通常我们会在类型为 **enum** 的配置项附近列出它的所有可选值。当然，你也可以查找 **配置项**。

5）**ip**：当你看到某个配置项的数据类型为 **ip** 时，意味着你可以使用 **<IP>:<Port>** 的形式来设置该配置项，例如 **0.0.0.0:1883**。

6）**string**：*.conf 文件中除注释以外的所有内容都会先被解析成字符串再转换为其他类型，因此没有必要对 **string** 类型的值额外使用双引号对值进行修饰，并且这种方式也不被支持。

7）**atom**：**atom** 类型的值最终会转换成 Erlang 的 **atom**，但它在 ***.conf** 文件中的使用方式与 **string** 完全一致。

8）**flag**：**flag** 用于那些具有两个可能值的变量，**flag** 默认可用值为 **on** 和 **off**，它们将映射为 **true** 和 **false** 以供应用程序使用。如果我们为某个配置项建立了其他的映射关系，我们会在配置文件中注明，你也可以在 **配置项** 中查找这些信息。

9）**duration**：用于指定那些固定的时间间隔，你可以使用以下时间单位：

- f – fortnight
- w – week
- d – day
- h – hour
- m – minute
- s – second
- ms – millisecond

你可以任意组合这些时间单位，例如 1w13ms，也可以使用浮点数，例如 0.5d，这些时间间隔最终将会被转换成我们指定的基准单位。这里有一点需要注意，如果你以毫秒为单位设置了某个配置项，而它的基准单位为秒，那么它将向上舍入至最接近的描述，例如 1s50ms = 2s。

因此，我们会列出这一类配置项的基准单位。

10）bytesize：

bytesize 支持以更易读的方式来设置报文大小、缓冲区大小等配置，单位可以是 KB，MB 和 GB，你也可以使用小写，例如 kb，但不支持大小写混合，例如 Kb，它们最终都将被转换为字节数。如果你未指定任何单位，那么它被直接作为字节数使用。

11）默认配置

在 EMQ X 的配置文件中，你会看到很多被注释掉的配置项，这意味着这些配置项会使用他们的默认值，通常我们会列出这些配置的默认值。

（4）Zone & Listener

EMQ X 提供了非常多的配置项，并支持全局配置和局部配置。例如，EMQ X 提供了匿名访问的功能，即允许客户端不需要用户名与密码就能连接 Broker，通常在用户的生产环境中，此功能被默认关闭，但用户可能又希望在他的内网环境中启用此功能。从 3.0 版本开始，EMQ X 就通过 Zone 与 Listener 为用户提供了这种可能。

（5）Listener

Listener 主要用于配置不同协议的监听端口和相关参数，EMQ X 支持配置多个 Listener 以同时监听多个协议或端口，以下是支持的 Listener：

监听器	说明
TCP Listener	A listener for MQTT which uses TCP
SSL Listener	A secure listener for MQTT which uses TLS
Websocket Listener	A listener for MQTT over WebSockets
Secure Websocket Listener	A secure listener for MQTT over secure WebSockets (TLS)

EMQ X 默认提供 5 个 Listener，它们将占用以下端口：

端口	说明
1883	MQTT/TCP 协议端口
11883	MQTT/TCP 协议内部端口，仅用于本机客户端连接
8883	MQTT/SSL 协议端口
8083	MQTT/WS 协议端口
8084	MQTT/WSS 协议端口

Listener 配置项的命名规则为 listener.<Protocol>.<Listener Name>.xxx，<Protocol> 即 Listener 使用的协议，目前支持 tcp, ssl, ws, wss。<Listener Name> 可以随意命名，但建议是全小写的英文单词，xxx 则是具体的配置项。不同协议的 Listener 的 <Listener Name> 可以重复，listener.tcp.external 与 listener.ssl.external 是两个不同的 Listener。

由于默认配置的存在，我们能够非常快速地展示如何添加新的 Listener，以 TCP Listener 为例，我们只需要在 emqx.conf 中添加以下一条配置即可：

当然这种情况我们更建议您复制一份默认 Listener 的配置进行修改。

（6）Zone

一个 Zone 定义了一组配置项（比如最大连接数等），Listener 可以通过配置项 listener.<Protocol>.<Listener Name>.zone 指定使用某个 Zone，以使用该 Zone 下的所有配置。多个 Listener 可以共享同一个 Zone。Zone 的命名规则为 zone.<Zone Name>.xxx，Zone Name 可以随意命名，但同样建议是全小写的英文单词，xxx 是具体的配置项，你可以在 配置项 中查看 Zone 支持的所有配置项。

此时，我们的每个配置项都存在三个可用值，分别是全局的值，Zone 里设置的值以及默认值，它们的优先级顺序为：Zone > Global > Default。

（7）配置更新

配置项会在 EMQ X Broker 与扩展插件被启动时读取并载入，EMQ X Broker 目前尚不支持运行时更新配置，但由于扩展插件支持动态加载和卸载，因此可以在修改插件配置后重

新加载插件来应用最新的配置项。

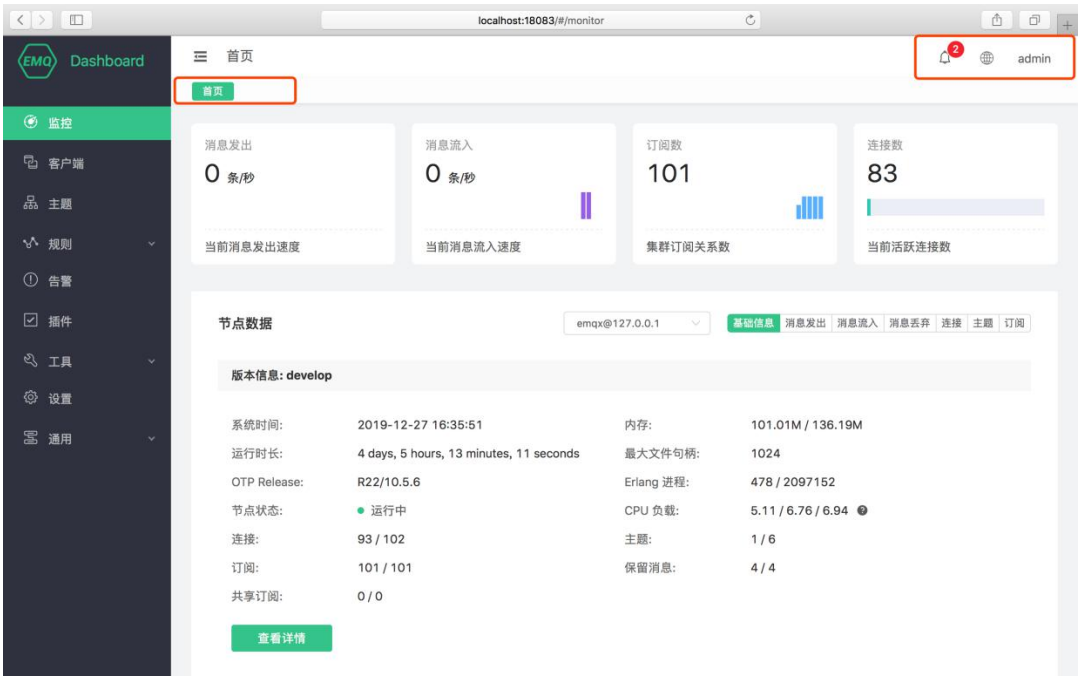
3.6 Dashboard

EMQ X Dashboard（EMQ X 管理控制台，以下简称 Dashboard）是 EMQ X 提供的一个后端 Web 控制台，用户可通过 Web 控制台查看服务器节点和集群的运行状态、统计指标，客户端的在线情况和订阅关系等信息，并进行插件配置与停启，HTTP API 密钥管理，EMQ X 集群的热配置管理和 MQTT 连接测试等操作。

（1）基本使用

如果 EMQ X 安装在本机，则使用浏览器打开地址 http://127.0.0.1:18083 (opens new window)，如需登录输入默认用户名 admin 与默认密码 public，登录进入 Dashboard。如果忘记管理用户账号信息，点击登录页面忘记密码按钮按指引操作或使用管理命令重置或新建管理账号。Dashboard 界面如下图所示，包含左侧导航栏、顶部控制栏和中间内容区，顶部控制栏（红框区域）四个功能分别是：

- 告警信息：EMQ X 告警信息，由资源使用过高、EMQ X 内部错误触发显示告警数量，点击可查看告警列表；
- 用户信息：当前登录 Dashboard 用户名，可进行登出、修改密码等操作；
- 多语言切换：Dashboard 根据用户浏览器语言默认显示中/英文，点击可进行语言切换；
- 最近页面导航：最近打开的页面将以 Tab 形式显示，点击可以快速进行页面切换。



（2）监控

监控页面可查看 EMQ X 当前集群的运行指标，界面从上到下功能区如下：

（3）集群运行指标

页面顶部四个指标卡片，包含集群的消息发出、消息流入速率，订阅数和当前连接数。

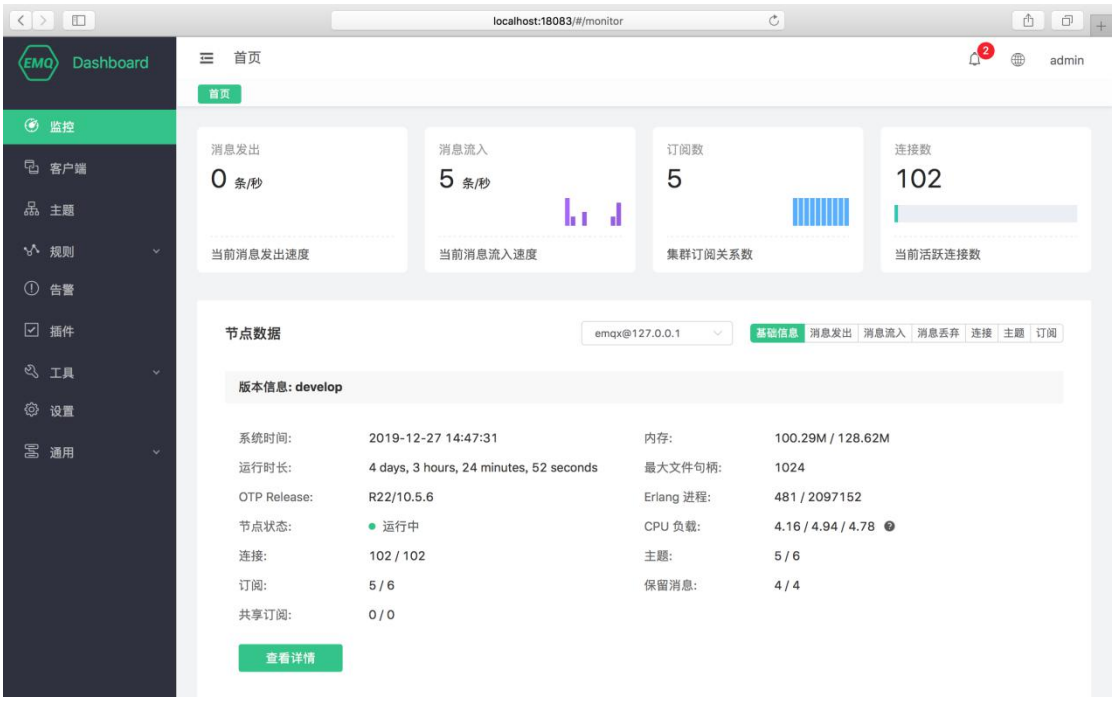
（4）节点数据

点击节点下拉列表可以切换查看节点的基本信息包括 EMQ X 版本信息、运行时间、资源占用、连接和订阅等数据。 部分信息释义如下：

内存： Erlang 虚拟机使用的当前内存/最大内存，其中最大内存由 EMQ X 视资源使用情况自动向系统申请，并非 EMQ X 所在节点服务器物理内存用户无需干预；

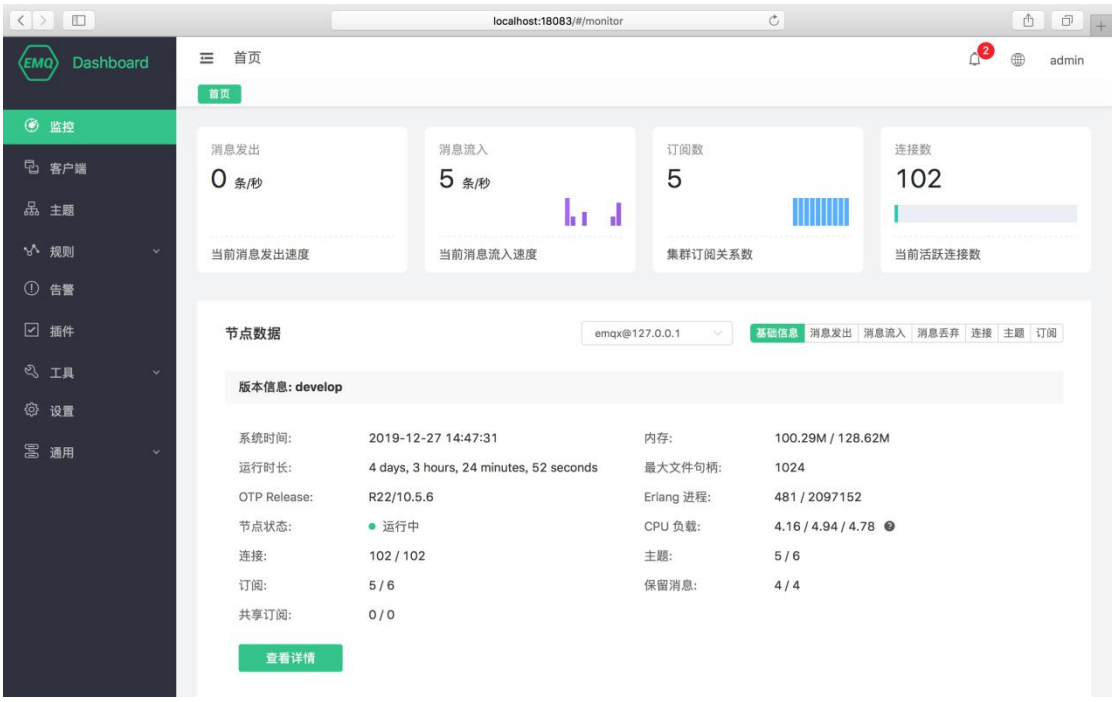
最大文件句柄：允许当前会话/进程打开文件句柄数，该值过小会限制 EMQ X 并发性能，在远小于 License 授权最大连接数时，请参照测试调优或联系 EMQ 技术人员进行修改；

Erlang 进程、连接、主题、订阅、保留消息、共享订阅：该四个值通过 / 分割为两组，分别是当前值与最大值。



（5）历史数据

点击节点数据区域右侧按钮组可以切换查看近期集群运行数据图表，图表数值均为抽样周期内实际数值：



（6）节点详细信息

点击节点数据下方 查看详情 按钮可以跳转至节点详情，查看当前节点的基础信息、监听器及连接情况，度量指标等信息。

(7) 监听器

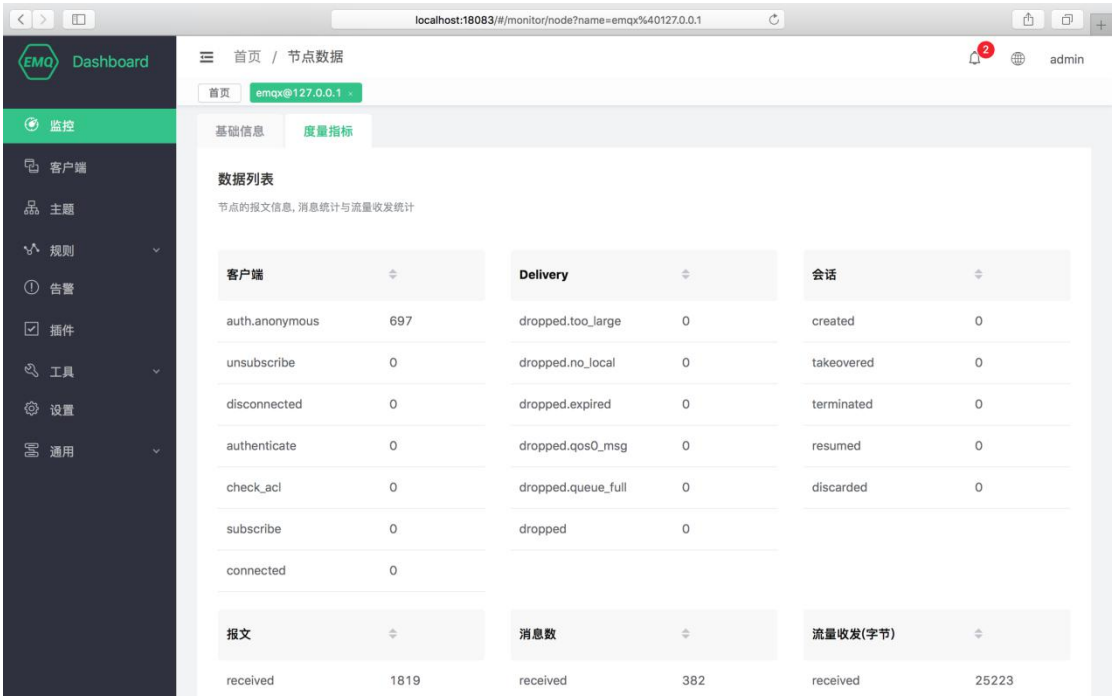
- 监听器为当前 EMQ X 监听网络端口列表，字段信息如下：
- 监听协议：监听的网络/应用协议，包含协议与功能信息：
  - mqtt:ssl： MQTT TCP TLS 加密协议，默认最大连接数 102400
  - mqtt:tcp： MQTT TCP 协议，默认最大连接数 1024000
  - http:dashboard：Dashboard (opens new window)使用的 HTTP 协议，默认最大连接数 512
  - http:management： EMQ (opens new window)X REST API 使用的 HTTP 协议，默认最大连接数 512
  - mqtt:ws：MQTT WebSocket 协议，默认最大连接数 102400
  - mqtt:wss：MQTT WebSocket TLS 加密协议，默认最大连接数 102400
- 监听地址：监听绑定的网络地址与端口，默认监听全部 IP 地址；
- Acceptors：监听处理器线程池，可通过各自协议 *.acceptors 字段配置；
- 连接：包含一组当前值/最大值，当前值为实际建立连接数量，最大值为配置文件配置的最大连接数量，每个监听器超出最大值后将无法建立新连接。

(8) 最大连接数说明

实际可用连接最大由 License 规格与配置文件共同决定：

- 节点内每个监听协议连接数不能超过配置文件最大连接数；
- 集群内 mqtt 协议的连接总数不能超过 License 规格上限。

系统调优与当前资源使用情况也可能会影响最大连接数，此处请参照 测试调优 (opens new window)或联系 EMQ 技术人员进行确认。



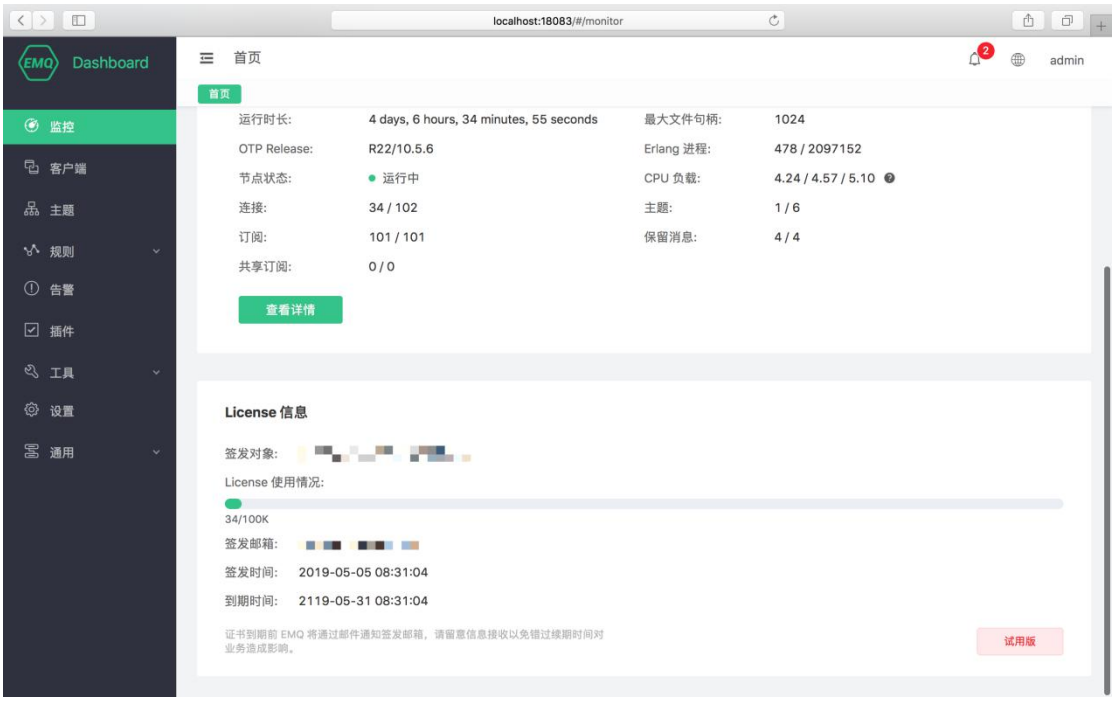
（9）License 信息

监控页面底部为 License 信息卡片，可以查看当前集群内 License 信息：

- 签发对象：同商务合同客户公司或部门名称；
- License 使用情况：License 规格与当前使用量；
- 签发邮箱：同商务合同客户联系邮箱；

- License 类型：右下角标识 License 信息，为试用版或正式版。

证书到期后您需要续签商务合同获取新的 License 证书并按照 License 更新方式或联系 EMQ 技术人员更新 License 证书，证书到期前 EMQ 将通过邮件通知签发邮箱，请留意信息接收以免错过续期时间对业务造成影响。

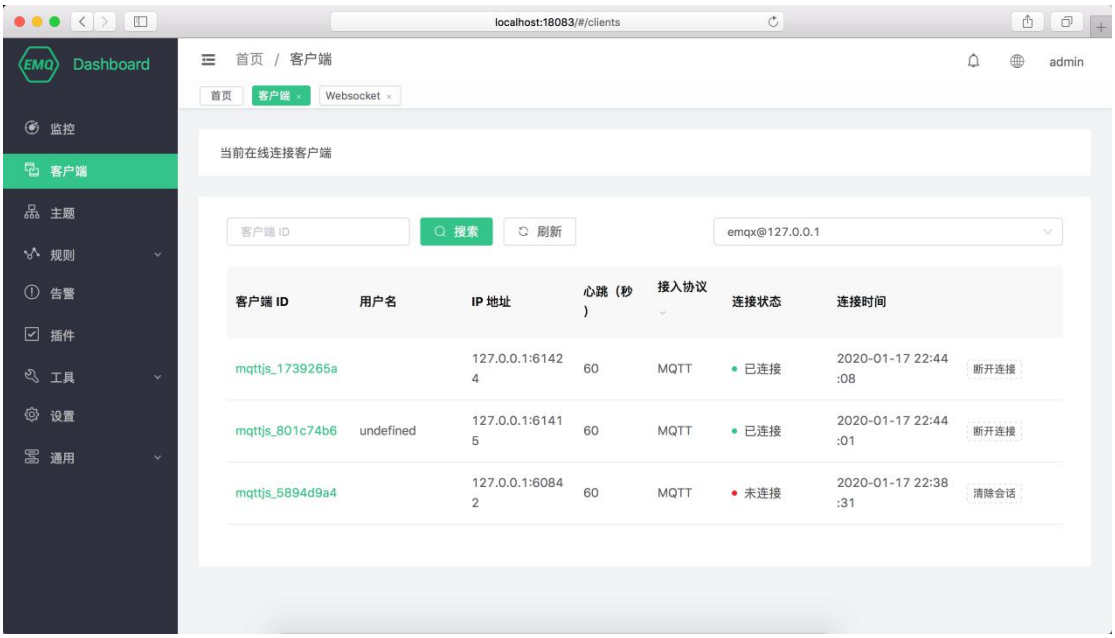


（10）客户端

1）客户端列表

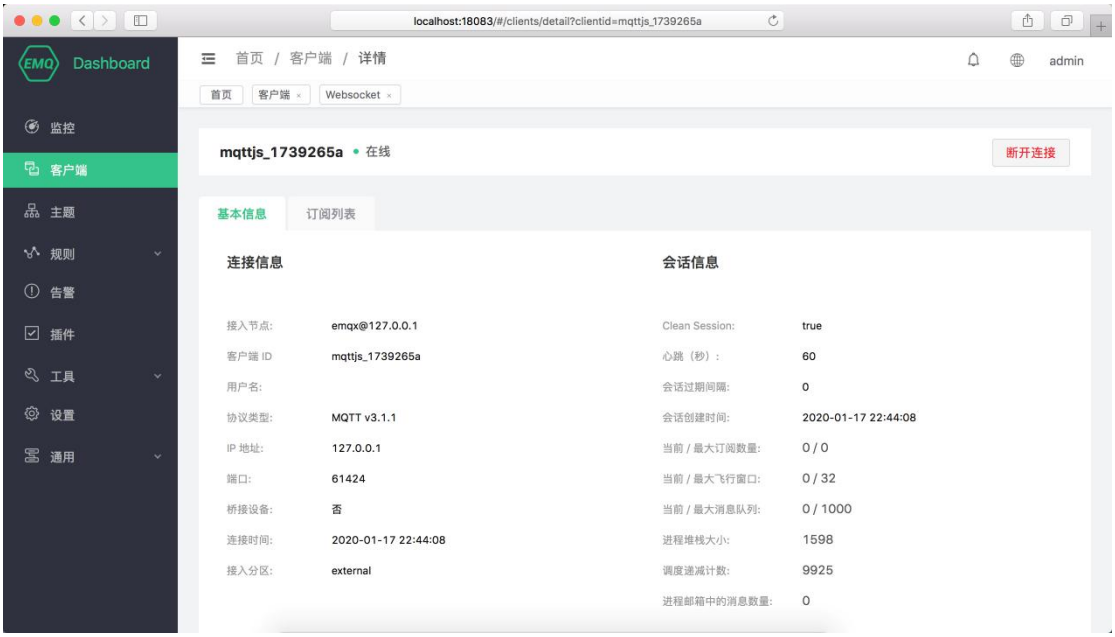
客户端列表页面显示当前连接客户端列表，列表中几个重要信息如下：

- 客户端 ID、用户名：分别是 MQTT Client ID 与 MQTT Username，点击绿色**客户端 ID** 可以查看客户端详情与订阅列表信息；
- IP 地址：客户端 IP + 端口信息；
- 连接状态：客户端在线状态，如果客户端已断开连接，但启用了**保留会话（Clean Session）**且会话未过期，此处将显示显示为**未连接**；
- 断开连接/清除会话：对于在线的客户端，将断开其连接并清除会话，如果客户端不在线，点击清除会话将清除客户端订阅关系等会话信息。



2）基本信息

点击绿色客户端 ID 可以查看客户端详情与订阅列表信息，基本信息包含所选客户端连接信息与会话信息，包含消息流量、消息统计等关键业务信息。

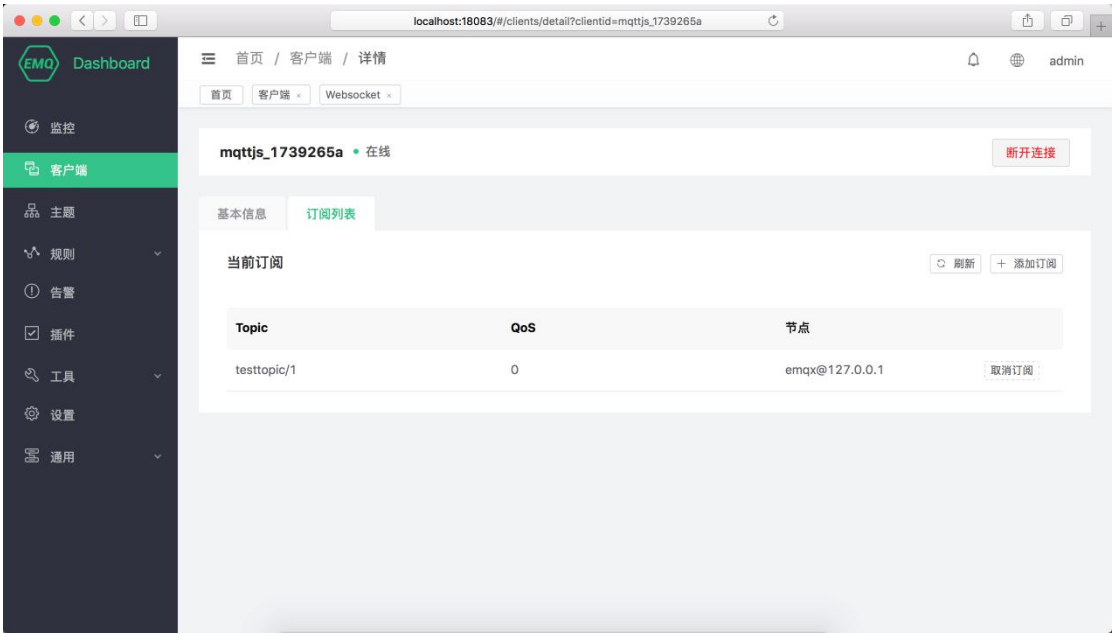


（11）订阅列表

订阅列表包含所选客户端订阅信息。

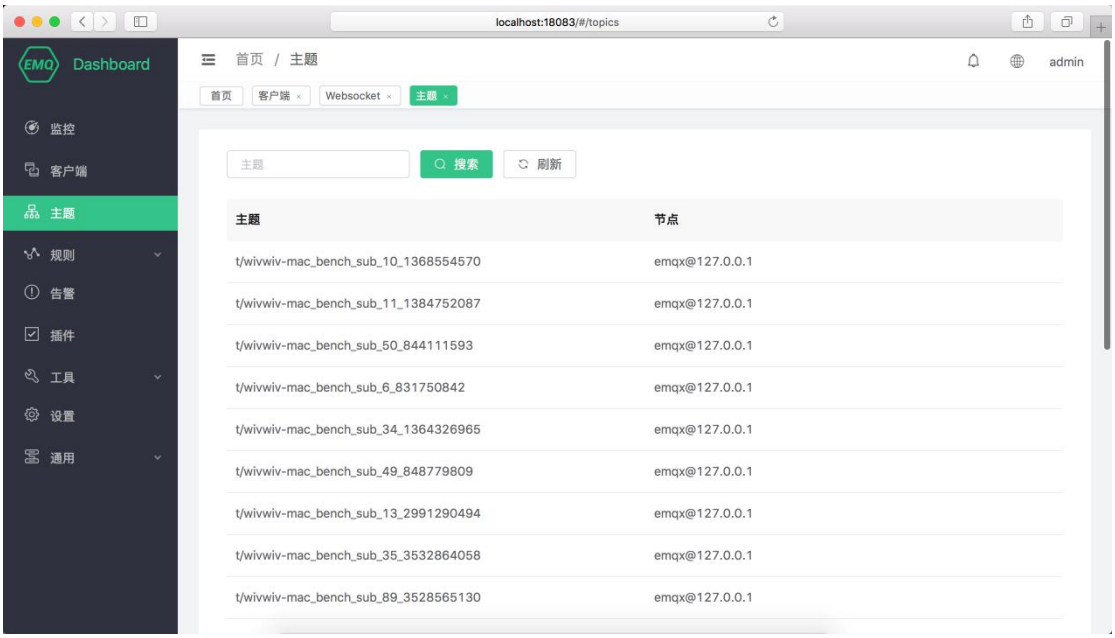
- 取消订阅：点击取消按钮将删除设备与主题的订阅关系，对于设备该操作是无感的；

- 添加订阅：为所选客户端代理订阅指定主题。



（12）主题

主题页面包含集群内全部主题（Topic）信息。



（13）规则引擎

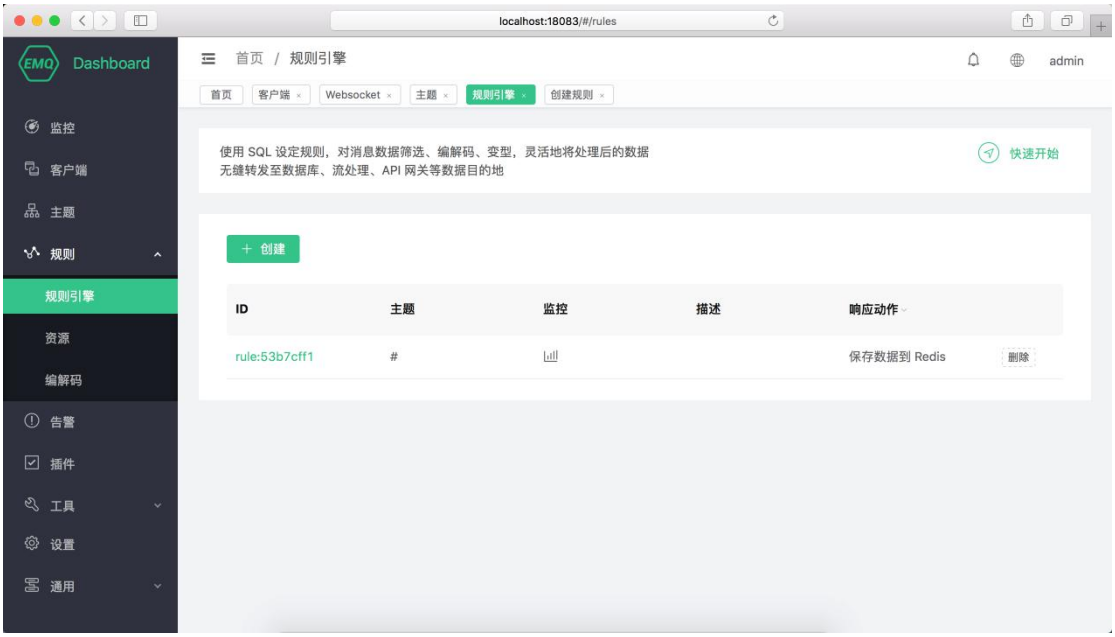
1）规则列表

规则引擎使用 SQL 设定规则，对消息数据进行筛选、编解码、变型，灵活地将处理后的数据无缝转发至数据库、流处理、API 网关等数据目的地。

规则引擎不仅提供了清晰、灵活的"配置式"的业务集成方案，简化了业务开发流程，提升用户易用性，降低业务系统与 EMQ X 的耦合度；也为用户私有功能定制提供了一个更优秀的基础架构。

规则引擎列表数据如下：

- ID：规则集群内唯一 ID，可用在管理命令和 REST API 操作中；
- 主题：规则对应的消息主题或事件主题；
- 监控：点击将弹出所选规则执行情况统计，包括规则命中次数与执行次数，动作触发成功/失败次数统计；
- 响应动作：一个或多个规则的响应动作。



2）创建规则

EMQ X 在消息发布、事件触发时将触发规则引擎，满足触发条件的规则将执行各自的 SQL 语句筛选并处理消息和事件的上下文信息。

规则引擎借助响应动作可将特定主题的消息处理结果存储到数据库，发送到 HTTP Server，转发到消息队列 Kafka 或 RabbitMQ，重新发布到新的主题甚至是另一个 Broker 集群中，每个规则可以配置多个响应动作。

1. 选择发布到 t/ 主题的消息，并筛选出全部字段：

```
SELECT * FROM "t/#"
```

S

ELECT * FROM "t/1

2. 选择发布到 t/a 主题的消息，并从 JSON 格式的消息内容中筛选出 "x" 字段：

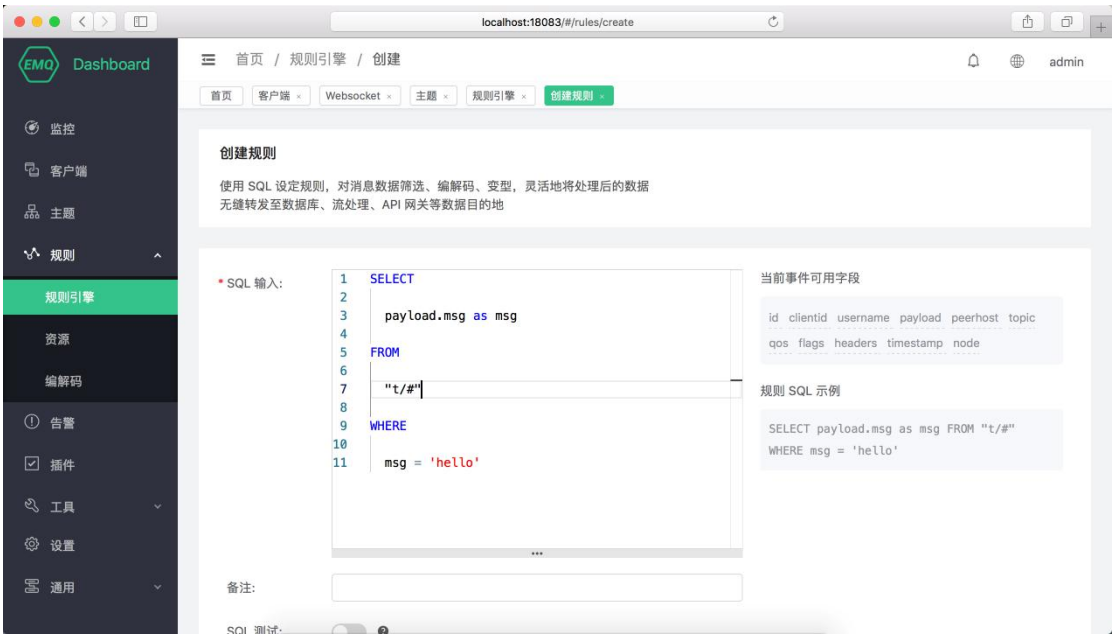
```
SELECT payload.x as x FROM "t/a"
```

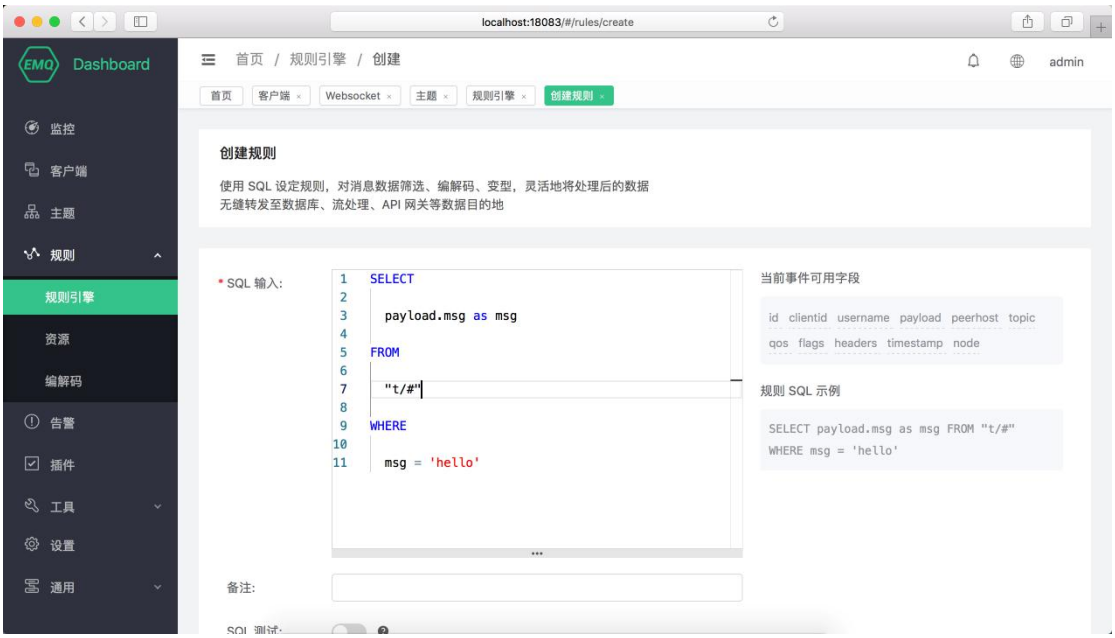
规则引擎使用 $events/ 开头的虚拟主题（事件主题）处理 EMQ X 内置事件，内置事件提供更精细的消息控制和客户端动作处理能力，可用在 QoS 1 QoS 2 的消息抵达记录、设备上下线记录等业务中。

1. 选择客户端连接事件，筛选 Username 为 'emqx' 的设备并获取连接信息：

SELECT clientid, connected_at FROM "$events/client_connected" WHERE username = 'emqx'

规则引擎数据和 SQL 语句格式，事件主题列表详细教程参见该文档规则引擎部分。





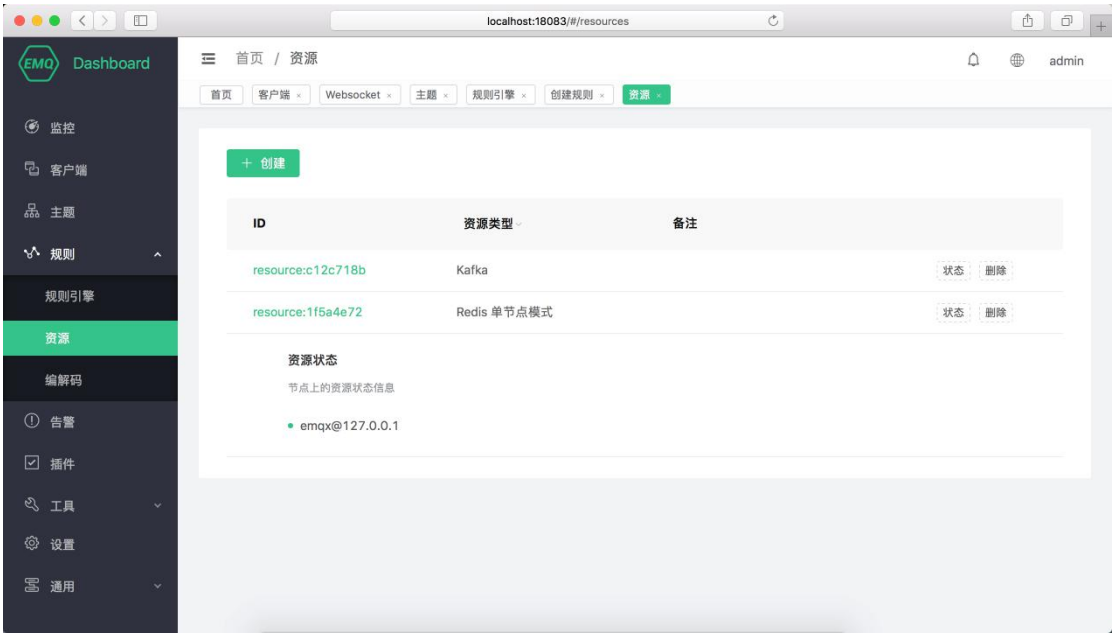
（14）资源

资源保存了规则引擎动作所需的资源实例(如数据库连接实例，Web Server 的连接信息)，创建规则前需要创建相关动作所需资源并保证资源可用。

1）资源列表

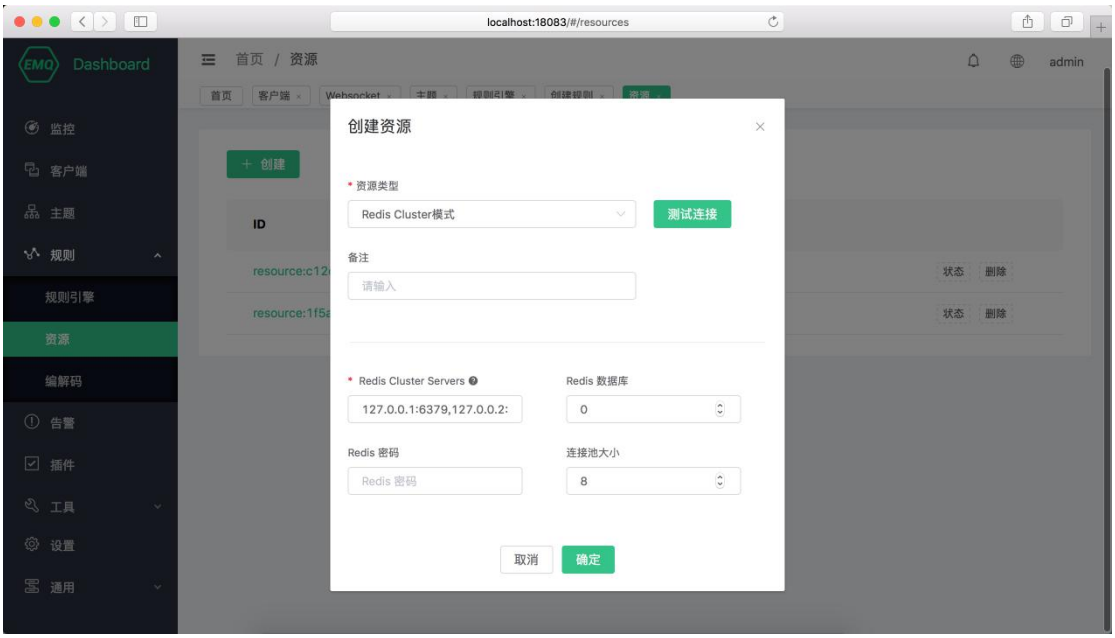
资源列表数据如下：

- ID：资源集群内唯一 ID，可用在管理命令和 REST API 操作中；
- 状态：资源创建后，集群中的每个节点都将与资源建立连接，点击展开节点上资源状态；
- 删除：规则引擎正在使用中的资源无法删除，请先删除依赖所选资源的规则再进行删除操作。



2）创建资源

点击**新建**按钮打开资源创建弹出框，选择资源类型、输入对应的连接信息即可创建资源，点击**测试连接**可在创建前进行资源连通性检查。



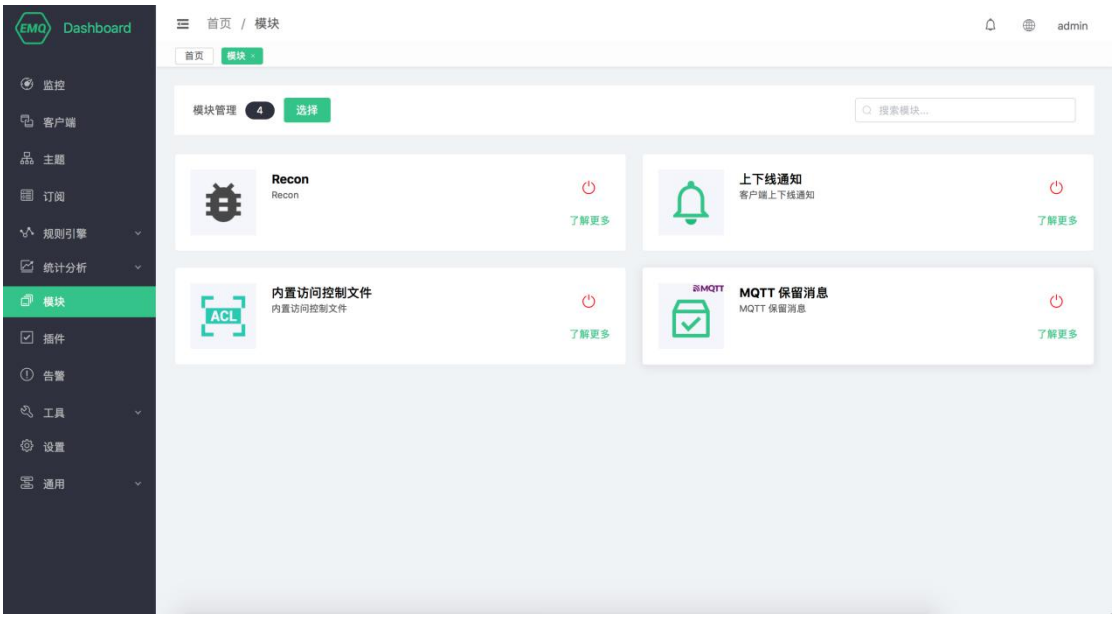
3）编解码

编解码（Schema Registry） 用于管理编解码使用的 Schema、处理编码或解码请求并返回结果。编解码配合规则引擎，可进行 Protobuf、Avro 以及私有协议上/下行消息解析处理，实现如消息加密、消息压缩、任意二进制-JSON 消息互转等复杂操作。

（15）模块

模块页面用于查看 EMQ X 创建管理模块功能操作。

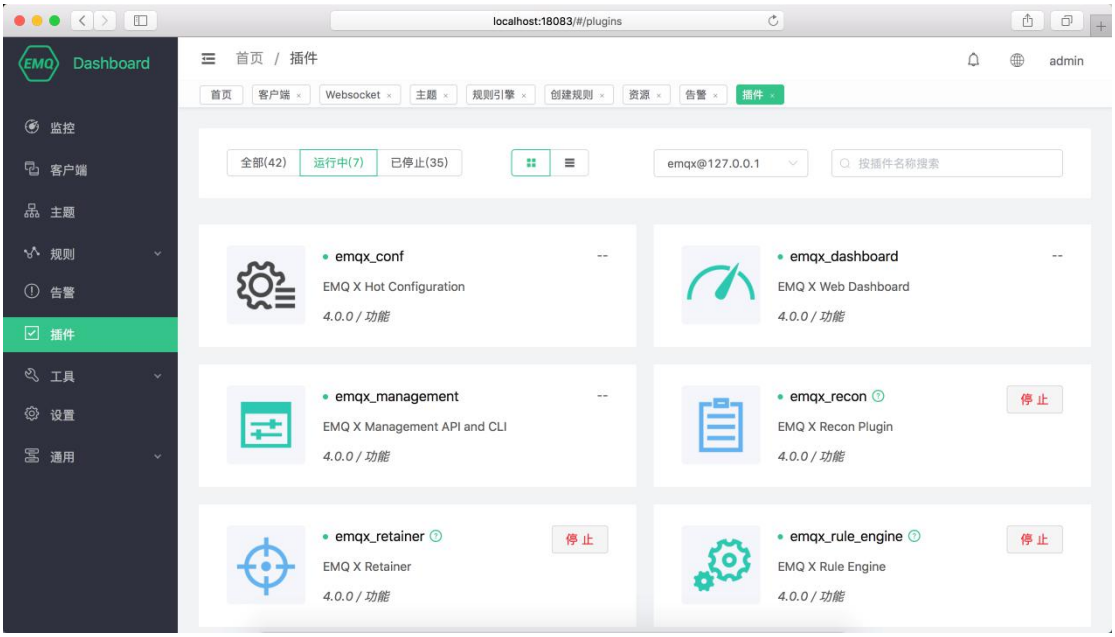
Dashboard 上模块的创建、启动、停止操作是集群同步的，如果模块启动失败，请检查集群内每个节点的配置是否正确，任意集群启动失败都无法成功启动模块。



（16）插件

插件页面用于查看 EMQ X 内置插件列表、进行插件的启动、停止操作。

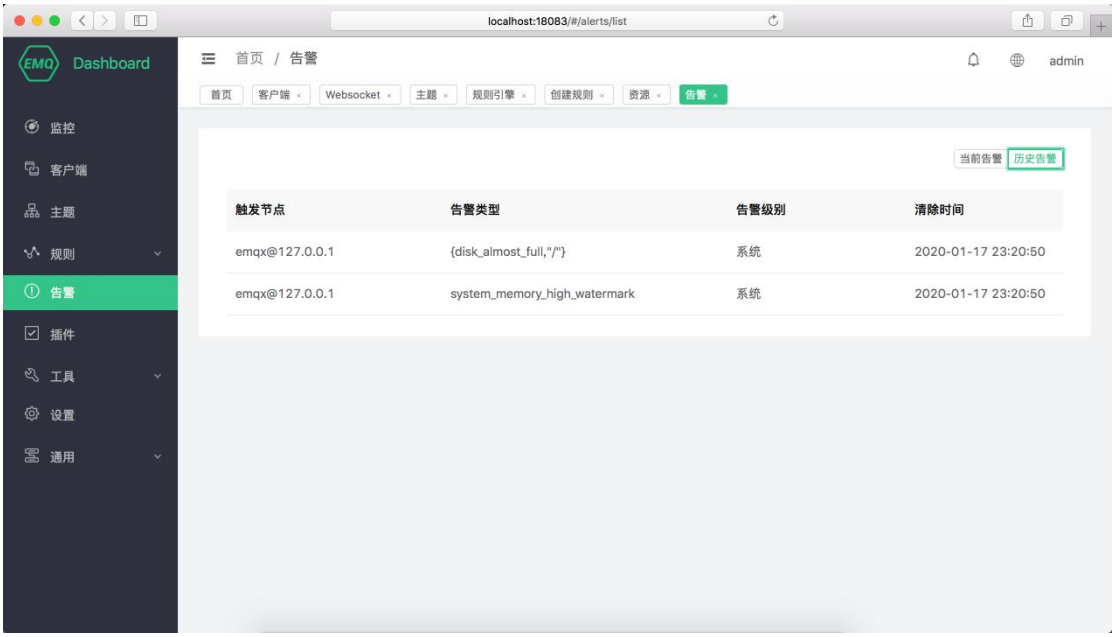
不同于命令行插件管理，Dashboard 上插件的启动、停止操作是集群同步的，如果插件启动失败，请检查集群内每个节点的配置是否正确，任意集群启动失败都无法成功启动插件。



（17）告警

用于展示 EMQ X 基础的告警信息，包含当前告警与历史告警信息。更高级的告警与日志

与监控管理由 EMQ X Control Center 提供，如有需要请联系 EMQ 技术人员获取。



（18）工具

提供 WebSocket MQTT 客户端测试工具，可同时实现多个 MQTT 连接的发布、订阅测试。

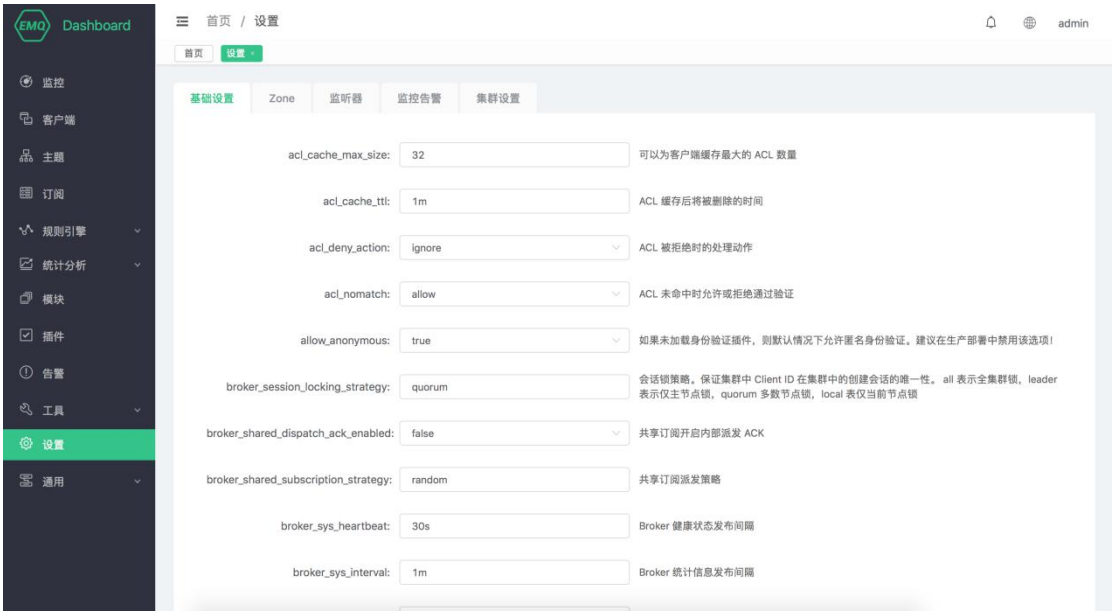
（19）设置

设置页面提供 EMQ X 集群的基础参数配置（热配置）与集群配置。

1）基础设置

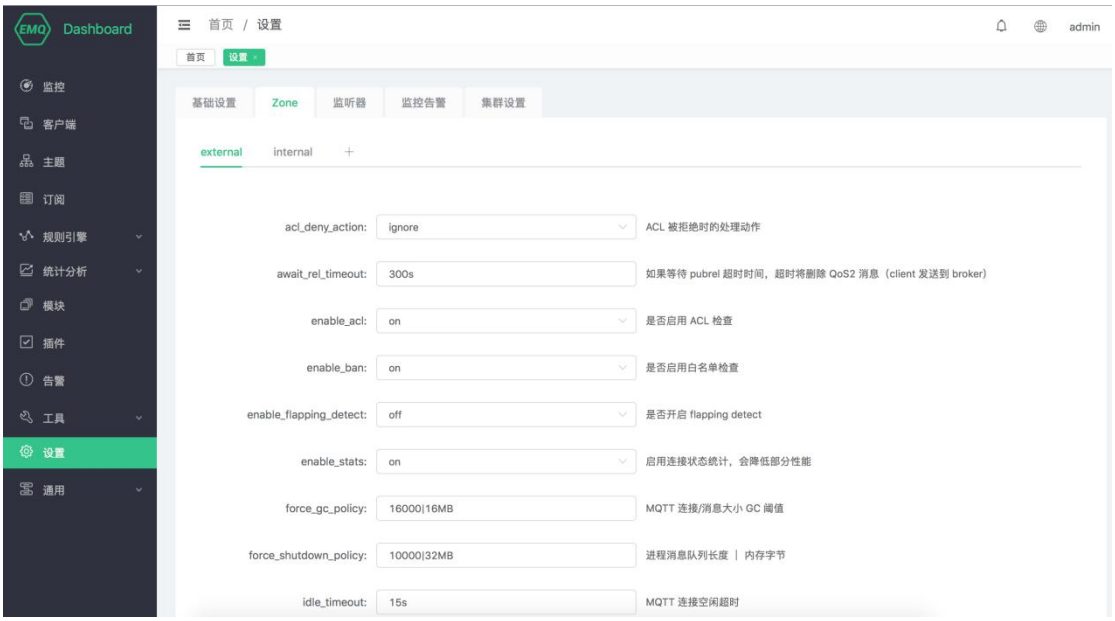
基础在设置开放了 emqx.conf 中可以进行热更新的部分配置项，您无需重启 EMQ X 即可完成大部分关键信息如是否开启匿名认证、ACL 缓存事件、ACL 缓存开关等配置。

基础设置是以 Zone 来组织的，默认情况下 external Zone 关联了 1883 端口所在监听器。



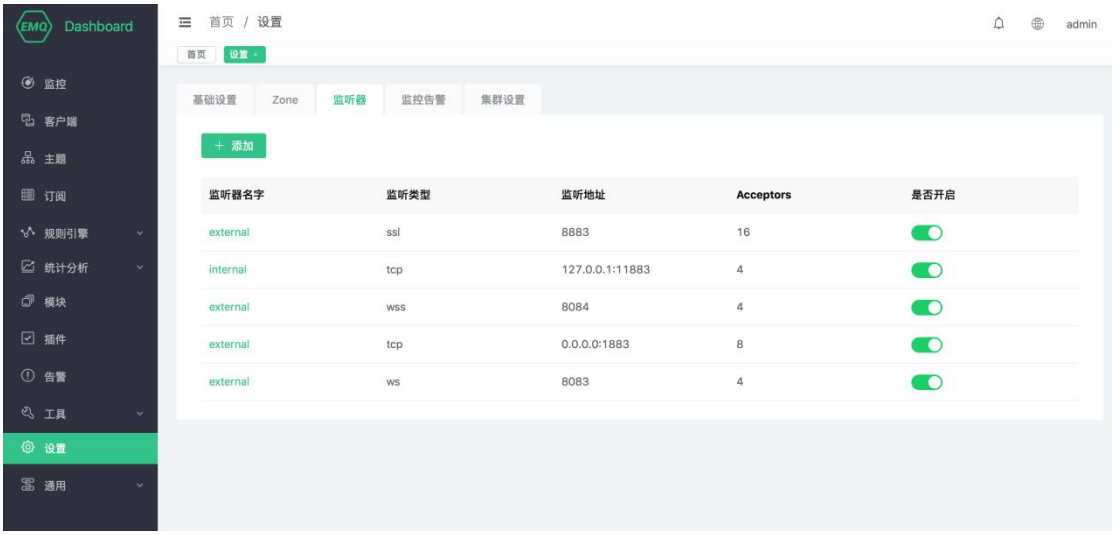
2）zones 设置

动态设置 zones 相关配置，修改后在整个集群生效，并且会持久化在 emqx 内部。(不会同步到 etc/zones.conf)



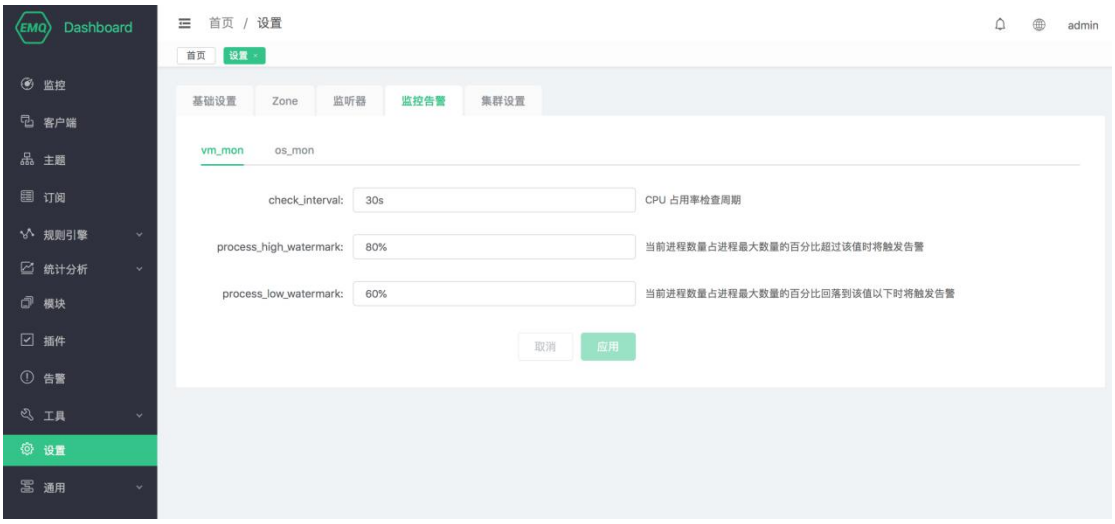
3）监听器设置

动态设置监听器相关配置，修改后在整个集群生效，并且会持久化在 emqx 内部。(不会同步到 etc/listeners.conf)



（20）监控告警设置

动态设置监控告警配置，修改后在整个集群生效，并且会持久化在 emqx 内部。(不会同步到 etc/sys_momn.conf)



1）集群设置

集群设置无法更改集群方式，但可用于手工集群邀请节点加入集群。

（21）通用

1）应用

应用为调用 REST API 认证凭证，通过 REST API 查询、调整 EMQ X 集群信息，对

客户端连接、插件、EMQ X 集群进行管理操作。

应用创建成功后，点击应用列表 App ID 列中的应用 ID 可以查看应用 ID 与应用密钥，您可以编辑应用状态与到期时间，新建或删除应用。

2）用户

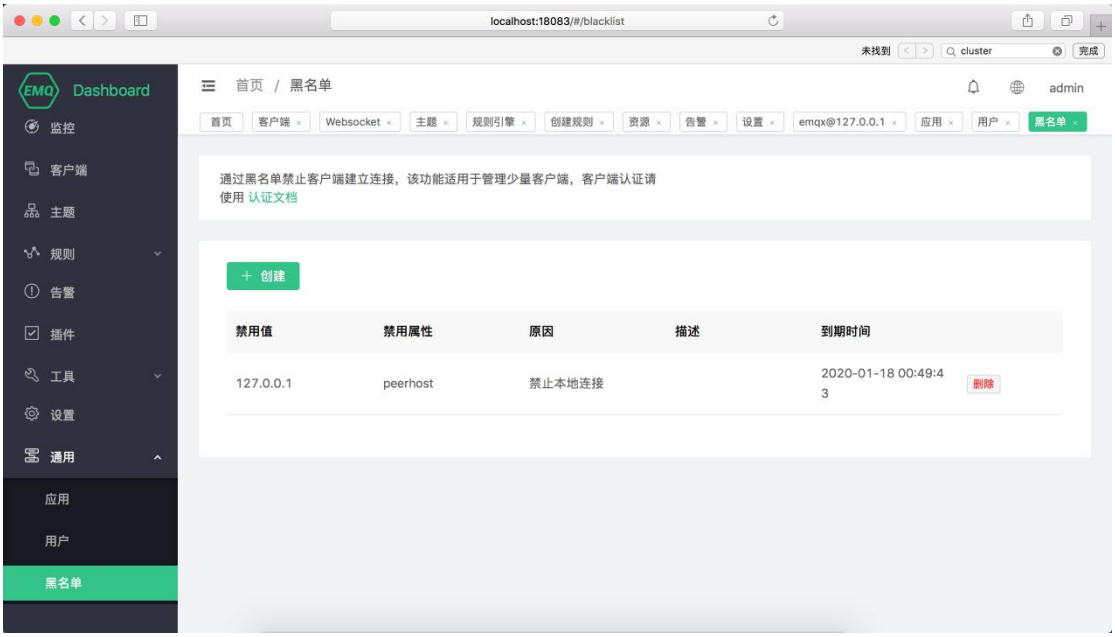
Dashboard 登录用户管理，您可以创建、编辑、删除用户，如果忘记用户密码，可通过管理命令进行密码重置。

（22）黑名单

黑名单用于禁止客户端建立连接，该功能适用于管理少量客户端，黑名单在有效时间到期后将失效。

黑名单支持以下三种方式禁止客户端连接：

- clientid：通过客户端 ID（Client ID） 进行封禁；
- username：通过用户名（Username）进行封禁；
- peerhost：通过对等主机（如 IP 地址）进行封禁。



4. 用户指南

4.1 认证鉴权

身份认证是大多数应用的重要组成部分，MQTT 协议支持用户名密码认证，启用身份认证能有效阻止非法客户端的连接。

EMQ X 中的认证指的是当一个客户端连接到 EMQ X 的时候，通过服务器端的配置来控制客户端连接服务器的权限。

EMQ X 的认证支持包括两个层面：

- MQTT 协议本身在 CONNECT 报文中指定用户名和密码，EMQ X 以插件形式支持基于 Username、ClientID、HTTP、JWT、LDAP 及各类数据库如 MongoDB、MySQL、PostgreSQL、Redis 等多种形式的认证。
- 在传输层上，TLS 可以保证使用客户端证书的客户端到服务器的身份验证，并确保服务器向客户端验证服务器证书。也支持基于 PSK 的 TLS/DTLS 认证。

本章节介绍了 EMQ X 支持的认证方式以及对应插件的配置方法。

（1）认证方式

EMQ X 支持使用内置数据源（文件、内置数据库）、JWT、外部主流数据库和自定义 HTTP API 作为身份认证数据源。

连接数据源、进行认证逻辑通过插件实现的，每个插件对应一种认证方式，使用前需要启用相应的插件。

客户端连接时插件通过检查其 username/clientid 和 password 是否与指定数据源的信息一致来实现对客户端的身份认证。

EMQ X 支持的认证方式：

1）内置数据源

- 内置数据库 认证/访问控制

使用配置文件与 EMQ X 内置数据库提供认证数据源，通过 HTTP API 进行管理，足够简单轻量。

2）外部数据库

- MySQL 认证/访问控制
- PostgreSQL 认证/访问控制
- Redis 认证/访问控制
- MongoDB 认证/访问控制
- LDAP 认证/访问控制

外部数据库可以存储大量数据，同时方便与外部设备管理系统集成。

3）其他

- HTTP 认证/访问控制
- JWT 认证

JWT 认证可以批量签发认证信息，HTTP 认证能够实现复杂的认证鉴权逻辑。

4）TIP

> 更改插件配置后需要重启插件才能生效，部分认证鉴权插件包含 ACL 功能。

（2）认证结果

任何一种认证方式最终都会返回一个结果：

- 认证成功：经过比对客户端认证成功
- 认证失败：经过比对客户端认证失败，数据源中密码与当前密码不一致
- 忽略认证（ignore）：当前认证方式中未查找到认证数据，无法显式判断结果是成功还是失败，交由认证链下一认证方式或匿名认证来判断

（3）匿名认证

EMQ X 默认配置中启用了匿名认证，任何客户端都能接入 EMQ X。没有启用认证插件或认证插件没有显式允许/拒绝（ignore）连接请求时，EMQ X 将根据匿名认证启用情况决定是否允许客户端连接。

配置匿名认证开关：

```
# etc/emqx.conf

## Value: true | false
allow_anonymous = true
```

WARNING

生产环境中请禁用匿名认证。

（4）密码加盐规则与哈希方法

EMQ X 多数认证插件中可以启用哈希方法，数据源中仅保存密码密文，保证数据安全。启用哈希方法时，用户可以为每个客户端都指定一个 salt（盐）并配置加盐规则，数据库中存储的密码是按照加盐规则与哈希方法处理后的密文。

以 MySQL 认证为例：

加盐规则与哈希方法配置：

```
# etc/plugins/emqx_auth_mysql.conf

## 不加盐，仅做哈希处理
auth.mysql.password_hash = sha256

## salt 前缀：使用 sha256 加密 salt + 密码 拼接的字符串
auth.mysql.password_hash = salt,sha256

## salt 后缀：使用 sha256 加密 密码 + salt 拼接的字符串
auth.mysql.password_hash = sha256,salt

## pbkdf2 with macfun iterations dklen
## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha5
## auth.mysql.password_hash = pbkdf2,sha256,1000,20
```

（5）如何生成认证信息

1) 为每个客户端分用户名、Client ID、密码以及 salt（盐）等信息

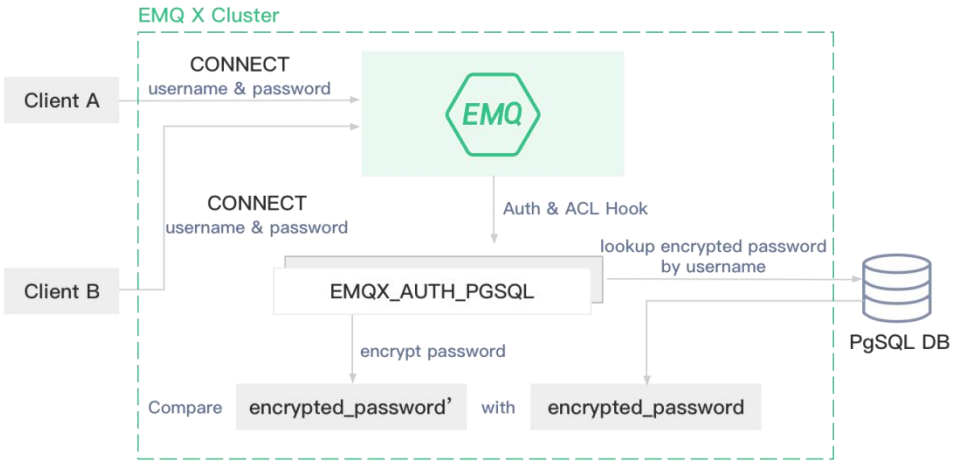
2) 使用与 MySQL 认证相同加盐规则与哈希方法处理客户端信息得到密文

3) 将客户端信息写入数据库，客户端的密码应当为密文信息

（6）EMQ X 身份认证流程

1. 根据配置的认证 SQL 结合客户端传入的信息，查询出密码（密文）和 salt（盐）等认证数据，没有查询结果时，认证将终止并返回 ignore 结果
2. 根据配置的加盐规则与哈希方法计算得到密文，没有启用哈希方法则跳过此步
3. 将数据库中存储的密文与当前客户端计算的到的密文进行比对，比对成功则认证通过，否则认证失败

PostgreSQL 认证功能逻辑图：

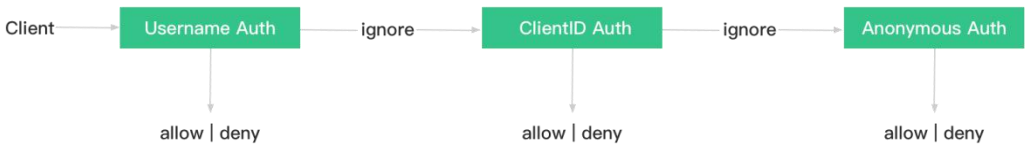


> **TIP**
>
> 写入数据的加盐规则、哈希方法与对应插件的配置一致时认证才能正常进行。更改哈希方法会造成现有认证数据失效。

（7）认证链

当同时启用多个认证方式时，EMQ X 将按照插件开启先后顺序进行链式认证：

- 一旦认证成功，终止认证链并允许客户端接入
- 一旦认证失败，终止认证链并禁止客户端接入
- 直到最后一个认证方式仍未通过，根据匿名认证配置判定
  - 匿名认证开启时，允许客户端接入
  - 匿名认证关闭时，禁止客户端接入



> **TIP**
>
> 同时只启用一个认证插件可以提高客户端身份认证效率。

（8）TLS 认证

MQTT TLS 的默认端口是 8883：

```
listener.ssl.external = 8883
```

配置证书和 CA：

```
listener.ssl.external.keyfile = etc/certs/key.pem
listener.ssl.external.certfile = etc/certs/cert.pem
listener.ssl.external.cacertfile = etc/certs/cacert.pem
```

注意，默认的 etc/certs 目录下面的 key.pem、cert.pem 和 cacert.pem 是 EMQ X 生成的自签名证书，所以在使用支持 TLS 的客户端测试的时候，需要将上面的 CA 证书 etc/certs/cacert.pem 配置到客户端。

服务端支持的 cipher 列表需要显式指定，默认的列表与 Mozilla 的服务端 cipher 列表一致：

```
listener.ssl.external.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384,ECD
```

（9）PSK 认证

如果希望使用 PSK 认证，需要将 TLS 认证 中的 listener.ssl.external.ciphers 注释掉，然后配置 listener.ssl.external.psk_ciphers：

```sh
#listener.ssl.external.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384,...
listener.ssl.external.psk_ciphers = PSK-AES128-CBC-SHA,PSK-AES256
```

然后启用 emqx_psk_file 插件：

```
$ emqx_ctl plugins load emqx_psk_file
```

PSK 的配置文件为 etc/psk.txt，使用冒号: 分隔 PSK ID 和 PSK：

```
client1:1234
client2:abcd
```

4.2 发布订阅 ACL

发布订阅 ACL 指对 发布 (PUBLISH)/订阅 (SUBSCRIBE) 操作的 权限控制。例如拒绝用户名为 Anna 向 open/elsa/door 发布消息。

EMQ X 支持通过客户端发布订阅 ACL 进行客户端权限的管理，本章节介绍了 EMQ X 支持的发布订阅 ACL 以及对应插件的配置方法。

（1）ACL 插件

EMQ X 支持使用配置文件、外部主流数据库和自定义 HTTP API 作为 ACL 数据源。

连接数据源、进行访问控制功能是通过插件实现的，使用前需要启用相应的插件。

客户端订阅主题、发布消息时插件通过检查目标主题（Topic）是否在指定数据源允许/禁止列表内来实现对客户端的发布、订阅权限管理。

1）配置文件/内置数据源

- 内置数据库 认证/访问控制

使用配置文件提供认证数据源，适用于变动较小的 ACL 管理。

2）外部数据库

- MySQL 认证/访问控制
- PostgreSQL 认证/访问控制
- Redis 认证/访问控制
- MongoDB 认证/访问控制
- LDAP 认证/访问控制

外部数据库可以存储大量数据、动态管理 ACL，方便与外部设备管理系统集成。

3）其他

- HTTP 认证/访问控制

HTTP ACL 能够实现复杂的 ACL 管理。

> TIP
>
> ACL 功能包含在认证鉴权插件中，更改插件配置后需要**重启插件**才能生效，

（2）规则详解

ACL 是允许与拒绝条件的集合，EMQ X 中使用以下元素描述 ACL 规则：

Allow-Deny Who Pub-Sub Topic

"允许(Allow) / 拒绝(Deny)" "谁(Who)" "订阅(Subscribe) / 发布(Publish)" "主题列表(Topics)"

```
## Allow-Deny Who Pub-Sub Topic

"允许(Allow) / 拒绝(Deny)"  "谁(Who)"  "订阅(Subscribe) / 发布(Publi
```

同时具有多条 ACL 规则时，EMQ X 将按照规则排序进行合并，以 ACL 文件 中的默认 ACL 为例，ACL 文件中配置了默认的 ACL 规则，规则从下至上加载：

- 第一条规则允许客户端发布订阅所有主题
- 第二条规则禁止全部客户端订阅 $SYS/ 与 主题
- 第三条规则允许 ip 地址为 127.0.0.1 的客户端发布/订阅 $SYS/ 与 主题，为第二条开了特例
- 第四条规则允许用户名为 dashboard 的客户端订阅 $SYS/ 主题，为第二条开了特例

```
{allow, {user, "dashboard"}, subscribe, ["$SYS/#"]}.

{allow, {ipaddr, "127.0.0.1"}, pubsub, ["$SYS/#", "#"]}.

{deny, all, subscribe, ["$SYS/#", {eq, "#"}]}.

{allow, all}.
```

（3）授权结果

任何一次 ACL 授权最终都会返回一个结果：

- 允许：经过检查允许客户端进行操作

- ✧ 禁止：经过检查禁止客户端操作
- ✧ 忽略（ignore）：未查找到 ACL 权限信息，无法显式判断结果是允许还是禁止，交由下一 ACL 插件或默认 ACL 规则来判断

（4）全局配置

默认配置中 ACL 是开放授权的，即授权结果为忽略（ignore）时允许客户端通过授权。

通过 etc/emqx.conf 中的 ACL 配置可以更改该属性

```
# etc/emqx.conf

## ACL 未匹配时默认授权
## Value: allow | deny
acl_nomatch = allow
```

配置默认 ACL 文件，使用文件定义默认 ACL 规则： {% endemqxee %}

```
# etc/emqx.conf

acl_file = etc/acl.conf
```

配置 ACL 授权结果为禁止的响应动作，为 disconnect 时将断开设备：

```
# etc/emqx.conf

## Value: ignore | disconnect
acl_deny_action = ignore
```

在 MQTT v3.1 和 v3.1.1 协议中，发布操作被拒绝后服务器无任何报文错误返回，这是协议设计的一个缺陷。但在 MQTT v5.0 协议上已经支持应答一个相应的错误报文。

（5）超级用户（superuser）

客户端可拥有“超级用户”身份，超级用户拥有最高权限不受 ACL 限制。

1. 认证鉴权插件启用超级用户功能后，发布订阅时 EMQ X 将优先检查客户端超级用户身份
2. 客户端为超级用户时，通过授权并跳过后续 ACL 检查

（6）ACL 缓存

ACL 缓存允许客户端在命中某条 ACL 规则后，便将其缓存至内存中，以便下次直接使用，客户端发布、订阅频率较高的情况下开启 ACL 缓存可以提高 ACL 检查性能。

在 etc/emqx.conf 可以配置 ACL 缓存大小与缓存时间：

```
# etc/emqx.conf

## 是否启用
enable_acl_cache = on

## 单个客户端最大缓存规则数量
acl_cache_max_size = 32

## 缓存失效时间，超时后缓存将被清除
acl_cache_ttl = 1m
```

（7）清除缓存

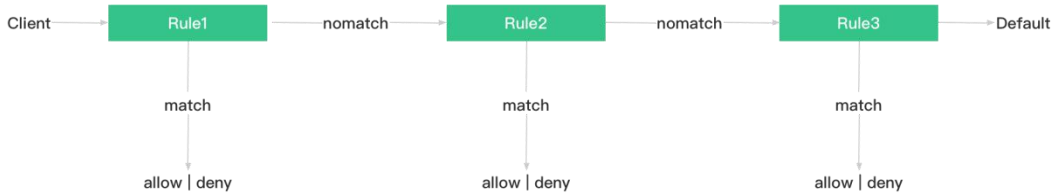
在更新 ACL 规则后，某些客户端由于已经存在缓存，则无法立即生效。若要立即生效，则需手动清除所有的 ACL 缓存：

参见 HTTP API - 清除 ACL 缓存

（8）ACL 鉴权链

当同时启用多个 ACL 插件时，EMQ X 将按照插件开启先后顺序进行链式鉴权：

- 一通过授权，终止链并允许客户端通过验证
- 一旦授权失败，终止链并禁止客户端通过验证
- 直到最后一个 ACL 插件仍未通过，根据默认授权配置判定
  - 默认授权为允许时，允许客户端通过验证
  - 默认授权为禁止时，禁止客户端通过验证

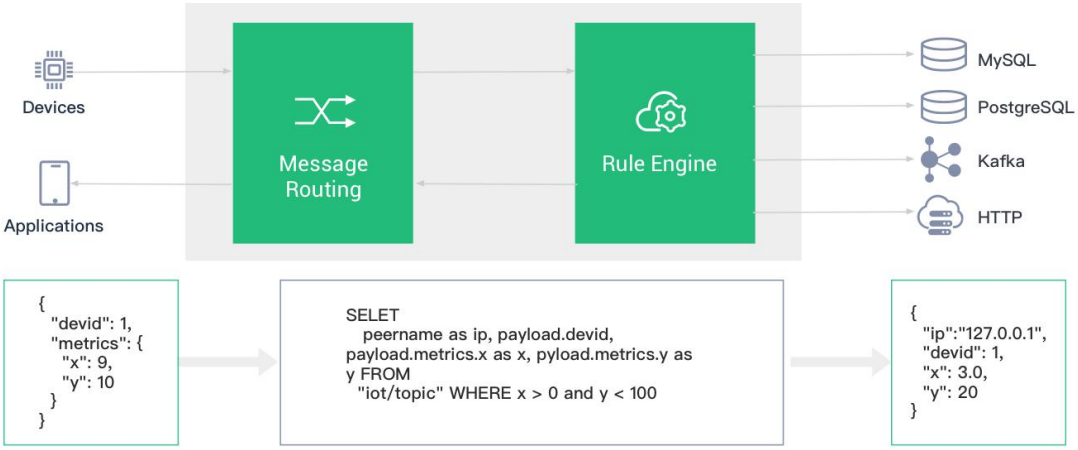


> **TIP**
>
> 同时只启用一个 ACL 插件可以提高客户端 ACL 检查性能。

4.3 规则引擎

4.3.1 规则引擎

EMQ X Rule Engine (以下简称规则引擎) 用于配置 EMQ X 消息流与设备事件的处理、响应规则。规则引擎不仅提供了清晰、灵活的 "配置式" 的业务集成方案，简化了业务开发流程，提升用户易用性，降低业务系统与 EMQ X 的耦合度；也为 EMQ X 的私有功能定制提供了一个更优秀的基础架构。



EMQ X 在 消息发布或事件触发 时将触发规则引擎，满足触发条件的规则将执行各自的 SQL 语句筛选并处理消息和事件的上下文信息。

> TIP
>
> 适用版本: EMQ X v3.1.0+
>
> 兼容提示: EMQ X v4.0 对规则引擎 SQL 语法做出较大调整，v3.x 升级用户请参照 迁移指南 进行适配。

（1）EMQ X 规则引擎快速入门

此处包含规则引擎的简介与实战，演示使用规则引擎结合华为云 RDS 上的 MySQL 服务，进行物联网 MQTT 设备在线状态记录、消息存储入库。

从本视频中可以快速了解规则引擎解决的问题和基础使用方法。

（2）消息发布

规则引擎借助响应动作可将特定主题的消息处理结果存储到数据库，发送到 HTTP Server，转发到消息队列 Kafka 或 RabbitMQ，重新发布到新的主题甚至是另一个 Broker 集群中，每个规则可以配置多个响应动作。

选择发布到 t/ 主题的消息，并筛选出全部字段：

```
SELECT * FROM "t/#"
```

选择发布到 t/a 主题的消息，并从 JSON 格式的消息内容中筛选出 "x" 字段：

```
SELECT payload.x as x FROM "t/a"
```

（3）事件触发

规则引擎使用 $events/ 开头的虚拟主题（事件主题）处理 EMQ X 内置事件，内置事件提供更精细的消息控制和客户端动作处理能力，可用在 QoS 1 QoS 2 的消息抵达记录、设备上下线记录等业务中。

选择客户端连接事件，筛选 Username 为 'emqx' 的设备并获取连接信息：

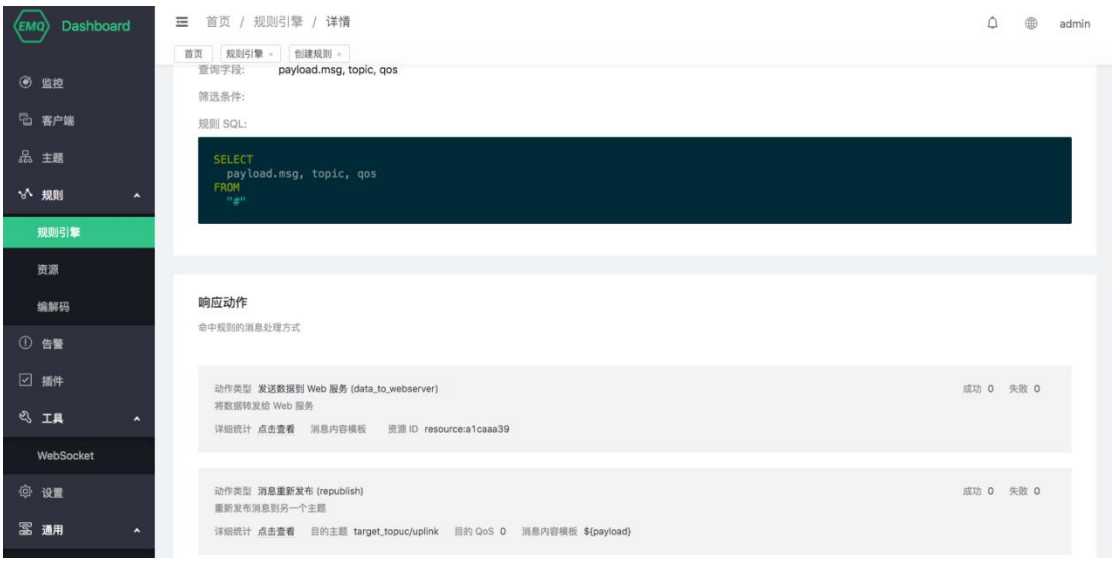
规则引擎数据和 SQL 语句格式，事件主题 列表详细教程参见 SQL 手册。

（4）最小规则

规则描述了 数据从哪里来、如何筛选并处理数据、处理结果到哪里去 三个配置，即一条可用的规则包含三个要素：

- 触发事件: 规则通过事件触发, 触发时事件给规则注入事件的上下文信息 (数据源），通过 SQL 的 FROM 子句指定事件类型;
- 处理规则 (SQL)：使用 SELECT 子句 和 WHERE 子句以及内置处理函数, 从上下文信息中过滤和处理数据;
- 响应动作：如果有处理结果输出，规则将执行相应的动作，如持久化到数据库、重新发布处理后的消息、转发消息到消息队列等。一条规则可以配置多个响应动作。

如图所示是一条简单的规则，该条规则用于处理 消息发布 时的数据，将全部主题消息的 msg 字段，消息 topic 、qos 筛选出来，发送到 Web Server 与 /uplink 主题：



（5）规则引擎典型应用场景举例

- 动作监听：智慧家庭智能门锁开发中，门锁会因为网络、电源故障、人为破坏等原因离线导致功能异常，使用规则引擎配置监听离线事件向应用服务推送该故障信息，可以在接入层实现第一时间的故障检测的能力;
- 数据筛选：车辆网的卡车车队管理，车辆传感器采集并上报了大量运行数据，应用平台仅关注车速大于 40 km/h 时的数据，此场景下可以使用规则引擎对消息进行条件过滤，向业务消息队列写入满足条件的数据;
- 消息路由：智能计费应用中，终端设备通过不同主题区分业务类型，可通过配置规则引擎将计费业务的消息接入计费消息队列并在消息抵达设备端后发送确认通知到业务系统，非计费信息接入其他消息队列，实现业务消息路由配置;

- 消息编解码：其他公共协议 / 私有 TCP 协议接入、工控行业等应用场景下，可以通过规则引擎的本地处理函数（可在 EMQ X 上定制开发）做二进制 / 特殊格式消息体的编解码工作；亦可通过规则引擎的消息路由将相关消息流向外部计算资源如函数计算进行处理（可由用户自行开发处理逻辑），将消息转为业务易于处理的 JSON 格式，简化项目集成难度、提升应用快速开发交付能力。

（6）迁移指南

4.0 版本中规则引擎 SQL 语法更加易用，3.x 版本中所有事件 FROM 子句后面均需要指定事件名称，4.0 以后我们引入 事件主题 概念，默认情况下 消息发布 事件不再需要指定事件名称：

```
3.x 版本
需要指定事件名称进行处理
SELECT * FROM "t/" WHERE          =~ 't/'


4.0 及以后版本
默认处理 message.publish 事件，FROM 后面直接填写 MQTT 主题
上述 SQL 语句等价于：
SELECT * FROM 't/'


其他事件，FROM 后面填写事件主题
SELECT * FROM "$events/message_acked" where          =~ 't/'
SELECT * FROM "$events/client_connected"
```

```
## 3.x 版本
## 需要指定事件名称进行处理
SELECT * FROM "t/#" WHERE topic =~ 't/#'

## 4.0 及以后版本
## 默认处理 message.publish 事件，FROM 后面直接填写 MQTT 主题
## 上述 SQL 语句等价于:
SELECT * FROM 't/#'

## 其他事件，FROM 后面填写事件主题
SELECT * FROM "$events/message_acked" where topic =~ 't/#'
SELECT * FROM "$events/client_connected"
```

TIP:Dashboard 中提供了旧版 SQL 语法转换功能可以完成 SQL 升级迁移。

（7）规则引擎组成

使用 EMQ X 的规则引擎可以灵活地处理消息和事件。使用规则引擎可以方便地实现诸如将消息转换成指定格式，然后存入数据库表，或者发送到消息队列等。

与 EMQ X 规则引擎相关的概念包括: 规则(rule)、动作(action)、资源(resource) 和 资源类型(resource-type)。

规则、动作、资源的关系:

```
规则: {
    SQL 语句,
    动作列表: [
        {
            动作1,
            动作参数,
            绑定资源: {
                资源配置
            }
        },
        {
            动作2,
            动作参数,
            绑定资源: {
                资源配置
            }
         }
    ]
}
```

规则: {规则(Rule): 规则由 SQL 语句和动作列表组成。动作列表包含一个或多个动作及其参数。

- SQL 语句用于筛选或转换消息中的数据。
- 动作(Action) 是 SQL 语句匹配通过之后，所执行的任务。动作定义了一个针对数据的操作。 动作可以绑定资源，也可以不绑定。例如，“inspect” 动作不需要绑定资源，它只是简单打印数据内容和动作参数。而 “data_to_webserver” 动作需要绑定一个 web_hook 类型的资源，此资源中配置了 URL。
- 资源(Resource): 资源是通过资源类型为模板实例化出来的对象，保存了与资源相关的配置(比如数据库连接地址和端口、用户名和密码等) 和系统资源(如文件句柄，连接套接字等)。
- 资源类型 (Resource Type): 资源类型是资源的静态定义，描述了此类型资源需要的配置项。

> TIP
>
> 动作和资源类型是由 emqx 或插件的代码提供的，不能通过 API 和 CLI 动态创建。

4.3.2 SQL 语句

(1) SQL 语法

1) FROM、SELECT 和 WHERE 子句:

规则引擎的 SQL 语句基本格式为:

```
SELECT <字段名> FROM <主题> [WHERE <条件>]
```

- FROM 子句将规则挂载到某个主题上
- SELECT 子句用于对数据进行变换，并选择出感兴趣的字段
- WHERE 子句用于对 SELECT 选择出来的某个字段施加条件过滤

2) FOREACH、DO 和 INCASE 子句:

如果对于一个数组数据，想针对数组中的每个元素分别执行一些操作并执行 Actions，需要使用 FOREACH-DO-INCASE 语法。其基本格式为:

```
FOREACH <字段名> [DO <条件>] [INCASE <条件>] FROM <主题> [WHERE <条件>]
```

- FOREACH 子句用于选择需要做 foreach 操作的字段, 注意选择出的字段必须为数组类型
- DO 子句用于对 FOREACH 选择出来的数组中的每个元素进行变换，并选择出感兴趣的字段
- INCASE 子句用于对 DO 选择出来的某个字段施加条件过滤

其中 DO 和 INCASE 子句都是可选的。DO 相当于针对当前循环中对象的 SELECT 子句，而 INCASE 相当于针对当前循环中对象的 WHERE 语句。

（2）事件和事件主题

规则引擎的 SQL 语句既可以处理消息(消息发布)，也可以处理事件(客户端上下线、客户端订阅等)。对于消息，FROM 子句后面直接跟主题名；对于事件，FROM 子句后面跟事件主题。

事件消息的主题以 "$events/" 开头，比

如 "$events/client_connected", "$events/session_subscribed"。如果想让 emqx 将事件消息发布出来，可以在 emqx_rule_engine.conf 文件中配置。

所有支持的事件及其可用字段详见: 规则事件。

（3）SQL 语句示例:

1）基本语法举例

- 从 topic 为 "t/a" 的消息中提取所有字段:

```
SELECT * FROM "t/a"
```

- 从 topic 为 "t/a" 或 "t/b" 的消息中提取所有字段:

```
SELECT * FROM "t/a","t/b"
```

- 从 topic 能够匹配到 't/' 的消息中提取所有字段。

```
SELECT * FROM "t/"1
```

- 从 topic 能够匹配到 't/' 的消息中提取 qos, username 和 clientid 字段:

```
SELECT qos, username, clientid FROM "t/"
```

- 从任意 topic 的消息中提取 username 字段，并且筛选条件为 username = 'Steven':

```
SELECT username FROM "" WHERE username='Steven'
```

- 从任意 topic 的 JSON 消息体(payload) 中提取 x 字段，并创建别名 x 以便在 WHERE 子句中使用。WHERE 子句限定条件为 x = 1。下面这个 SQL 语句可以匹配到消息体 {"x": 1}，但不能匹配到消息体 {"x": 2}:

```
SELECT payload as p FROM "" WHERE p.x = 1
```

- 类似于上面的 SQL 语句，但嵌套地提取消息体中的数据，下面的 SQL 语句可以匹配到 JSON 消息体 {"x": {"y": 1}}:

```
SELECT payload as a FROM "" WHERE a.x.y = 1
```

- 在 clientid = 'c1' 尝试连接时，提取其来源 IP 地址和端口号:

```
SELECT peername as ip_port FROM "$events/client_connected" WHERE clientid = 'c1'
```

- 筛选所有订阅 't/' 主题且订阅级别为 QoS1 的 clientid:

```
SELECT clientid FROM "$events/session_subscribed" WHERE topic = 't/' and qos = 1
```

- 筛选所有订阅主题能匹配到 't/' 且订阅级别为 QoS1 的 clientid。注意与上例不同的是，这里用的是主题匹配操作符 **'=~'**，所以会匹配订阅 't' 或 't/+/a' 的订阅事件:

```
SELECT clientid FROM "$events/session_subscribed" WHERE topic =~ 't/' and qos = 1
```

> TIP
>
> - FROM 子句后面的主题需要用双引号 `""` 引起来。
> - WHERE 子句后面接筛选条件，如果使用到字符串需要用单引号 `''` 引起来。
> - FROM 子句里如有多个主题，需要用逗号 `","` 分隔。例如 SELECT * FROM "t/1", "t/2" 。
> - 可以使用使用 `"."` 符号对 payload 进行嵌套选择。

2）遍历语法(FOREACH-DO-INCASE) 举例

假设有 ClientID 为 c_steve、主题为 t/1 的消息，消息体为 JSON 格式，其中 sensors 字段为包含多个 Object 的数组:

```
{
    "date": "2020-04-24",
    "sensors": [
        {"name": "a", "idx":0},
        {"name": "b", "idx":1},
        {"name": "c", "idx":2}
    ]
}
```

示例 1: 要求将 sensors 里的各个对象，分别作为数据输入重新发布消息到 sensors/${idx} 主题，内容为 ${name}。即最终规则引擎将会发出 3 条消息:

1. 主题：sensors/0 内容：a
2. 主题：sensors/1 内容：b
3. 主题：sensors/2 内容：c

要完成这个规则，我们需要配置如下动作:

- 动作类型：消息重新发布（republish）
- 目的主题：sensors/${idx}
- 目的 QoS：0

- 消息内容模板：${name}

以及如下 SQL 语句：

```
FOREACH
    payload.sensors
FROM "t/#"
```

示例解析:

这个 SQL 中，FOREACH 子句指定需要进行遍历的数组 sensors，则选取结果为:

```
[
  {
    "name": "a",
    "idx": 0
  },
  {
    "name": "b",
    "idx": 1
  },
  {
    "name": "c",
    "idx": 2
  }
]
```

FOREACH 语句将会对于结果数组里的每个对象分别执行 "消息重新发布" 动作，所以将会执行重新发布动作 3 次。

示例 2: 要求将 sensors 里的 idx 值大于或等于 1 的对象，分别作为数据输入重新发布消息到 sensors/${idx} 主题，内容为 clientid=${clientid},name=${name},date=${date}。即最终规则引擎将会发出 2 条消息:

1. 主题：sensors/1 内容：clientid=c_steve,name=b,date=2020-04-24

2. 主题：sensors/2 内容：clientid=c_steve,name=c,date=2020-04-24

要完成这个规则，我们需要配置如下动作：

- 动作类型：消息重新发布（republish）
- 目的主题：sensors/${idx}
- 目的 QoS：0
- 消息内容模板：clientid=${clientid},name=${name},date=${date}

以及如下 SQL 语句：

```
FOREACH
    payload.sensors
DO
    clientid,
    item.name as name,
    item.idx as idx
INCASE
    item.idx >= 1
FROM "t/#"
```

示例解析:

这个 SQL 中，FOREACH 子句指定需要进行遍历的数组 sensors; DO 子句选取每次操作需要的字段，这里我们选了外层的 clientid 字段，以及当前 sensor 对象的 name 和 idx 两个字段，注意 item 代表 sensors 数组中本次循环的对象。INCASE 子句是针对 DO 语句中字段的筛选条件，仅仅当 idx >= 1 满足条件。所以 SQL 的选取结果为:

```
[
  {
    "name": "b",
    "idx": 1,
    "clientid": "c_emqx"
  },
  {
    "name": "c",
    "idx": 2,
    "clientid": "c_emqx"
  }
]
```

FOREACH 语句将会对于结果数组里的每个对象分别执行 "消息重新发布" 动作, 所以将会执行重新发布动作 2 次。

在 DO 和 INCASE 语句里, 可以使用 item 访问当前循环的对象, 也可以通过在 FOREACH 使用 as 语法自定义一个变量名。所以本例中的 SQL 语句又可以写为:

```
FOREACH
    payload.sensors as s
DO
    clientid,
    s.name as name,
    s.idx as idx
INCASE
    s.idx >= 1
FROM "t/#"
```

示例 3: 在示例 2 的基础上, 去掉 clientid 字段 c_steve 中的 c_ 前缀

在 FOREACH 和 DO 语句中可以调用各类 SQL 函数, 若要将 c_steve 变为 steve, 则可以把例 2 中的 SQL 改为:

```
FOREACH
    payload.sensors as s
DO
    nth(2, tokens(clientid,'_')) as clientid,
    s.name as name,
    s.idx as idx
INCASE
    s.idx >= 1
FROM "t/#"
```

另外，FOREACH 子句中也可以放多个表达式，只要最后一个表达式是指定要遍历的数组即可。比如我们将消息体改一下，sensors 外面多套一层 Object：

```
{
    "date": "2020-04-24",
    "data": {
        "sensors": [
            {"name": "a", "idx":0},
            {"name": "b", "idx":1},
            {"name": "c", "idx":2}
        ]
    }
}
```

则 FOREACH 中可以在决定要遍历的数组之前把 data 选取出来：

```
FOREACH
    payload.data as data
    data.sensors as s
...
```

3）CASE-WHEN 语法示例

示例 1：将消息中 x 字段的值范围限定在 0~7 之间。

```
SELECT
  CASE WHEN payload.x < 0 THEN 0
       WHEN payload.x > 7 THEN 7
       ELSE payload.x
  END as x
FROM "t/#"
```

假设消息为:

```
{"x": 8}
```

则上面的 SQL 输出为:

```
{"x": 7}
```

4）数组操作语法举例

示例 1: 创建一个数组，赋值给变量 a:

```
SELECT
  [1,2,3] as a
FROM
  "t/#"
```

下标从 1 开始，上面的 SQL 输出为:

```
{
  "a": [1, 2, 3]
}
```

示例 2: 从数组中取出第 N 个元素。下标为负数时，表示从数组的右边取:

```
SELECT
  [1,2,3] as a,
  a[2] as b,
  a[-2] as c
FROM
  "t/#"
```

上面的 SQL 输出为:

```
{
  "b": 2,
  "c": 2,
  "a": [1, 2, 3]
}
```

示例 3: 从 JSON 格式的 payload 中嵌套的获取值:

```
SELECT
  payload.data[1].id as id
FROM
  "t/#"
```

假设消息为:

```
{"data": [
    {"id": 1, "name": "steve"},
    {"id": 2, "name": "bill"}
]}
```

则上面的 SQL 输出为:

```
{"id": 1}
```

示例 4: 数组范围(range)操作:

```
SELECT
  [1..5] as a,
  a[2..4] as b
FROM
  "t/#"
```

上面的 SQL 输出为:

```
{
  "b": [2, 3, 4],
  "a": [1, 2, 3, 4, 5]
}
```

示例 5: 使用下标语法修改数组中的某个元素:

```
SELECT
  payload,
  'STEVE' as payload.data[1].name
FROM
  "t/#"
```

假设消息为:

```
{"data": [
    {"id": 1, "name": "steve"},
    {"id": 2, "name": "bill"}
]}
```

则上面的 SQL 输出为:

```
{
  "payload": {
    "data": [
      {"name": "STEVE", "id": 1},
      {"name": "bill", "id": 2}
    ]
  }
}
```

（4）FROM 子句可用的事件主题

事件主题名	释义
$events/message_delivered	消息投递
$events/message_acked	消息确认
$events/message_dropped	消息丢弃
$events/client_connected	连接完成
$events/client_disconnected	连接断开
$events/session_subscribed	订阅
$events/session_unsubscribed	取消订阅

（5）SELECT 和 WHERE 子句可用的字段

SELECT 和 WHERE 子句可用的字段与事件的类型相关。其中 clientid, username 和 event 是通用字段，每种事件类型都有。

1）普通主题（消息发布）

event	**事件类型，固定为 "message.publish"**
id	MQTT 消息 ID
clientid	Client ID

event	事件类型，固定为 "message.publish"
username	用户名
payload	MQTT 消息体
peerhost	客户端的 IPAddress
topic	MQTT 主题
qos	MQTT 消息的 QoS
flags	MQTT 消息的 Flags
headers	MQTT 消息内部与流程处理相关的额外数据
timestamp	事件触发时间（ms）
publish_received_at	PUBLISH 消息到达 Broker 的时间（ms）
node	事件触发所在节点

2）events/message_delivered（消息投递）

event	事件类型，固定为 "message.delivered"
id	MQTT 消息 ID
from_clientid	消息来源 Client ID

event	**事件类型，固定为 "message.delivered"**
from_username	消息来源用户名
clientid	消息目的 Client ID
username	消息目的用户名
payload	MQTT 消息体
peerhost	客户端的 IPAddress
topic	MQTT 主题
qos	MQTT 消息的 QoS
flags	MQTT 消息的 Flags
timestamp	事件触发时间（ms）
publish_received_at	PUBLISH 消息到达 Broker 的时间（ms）
node	事件触发所在节点

3）$events/message_acked（消息确认）

event	**事件类型，固定为 "message.acked"**
id	MQTT 消息 ID

event	**事件类型，固定为 "message.acked"**
from_clientid	消息来源 Client ID
from_username	消息来源用户名
clientid	消息目的 Client ID
username	消息目的用户名
payload	MQTT 消息体
peerhost	客户端的 IPAddress
topic	MQTT 主题
qos	MQTT 消息的 QoS
flags	MQTT 消息的 Flags
timestamp	事件触发时间（ms）
publish_received_at	PUBLISH 消息到达 Broker 的时间（ms）
node	事件触发所在节点

4）$events/message_dropped (消息丢弃)

event	**事件类型，固定为 "message.dropped"**

event	**事件类型，固定为 "message.dropped"**
id	MQTT 消息 ID
reason	消息丢弃原因
clientid	消息目的 Client ID
username	消息目的用户名
payload	MQTT 消息体
peerhost	客户端的 IPAddress
topic	MQTT 主题
qos	MQTT 消息的 QoS
flags	MQTT 消息的 Flags
timestamp	事件触发时间（ms）
publish_received_at	PUBLISH 消息到达 Broker 的时间（ms）
node	事件触发所在节点

5）$events/client_connected (终端连接成功)

event	**事件类型，固定为 "client.connected"**

event	**事件类型，固定为 "client.connected"**
clientid	消息目的 Client ID
username	消息目的用户名
mountpoint	主题挂载点(主题前缀)
peername	终端的 IPAddress 和 Port
sockname	emqx 监听的 IPAddress 和 Port
proto_name	协议名字
proto_ver	协议版本
keepalive	MQTT 保活间隔
clean_start	MQTT clean_start
expiry_interval	MQTT Session 过期时间
is_bridge	是否为 MQTT bridge 连接
connected_at	终端连接完成时间（s）
timestamp	事件触发时间（ms）
node	事件触发所在节点

6）$events/client_disconnected (终端连接断开)

event	事件类型，固定为 "client.disconnected"
reason	终端连接断开原因
clientid	消息目的 Client ID
username	消息目的用户名
peername	终端的 IPAddress 和 Port
sockname	emqx 监听的 IPAddress 和 Port
disconnected_at	终端连接断开时间（s）
timestamp	事件触发时间（ms）
node	事件触发所在节点

7）$events/session_subscribed (终端订阅成功)

event	事件类型，固定为 "session.subscribed"
clientid	消息目的 Client ID
username	消息目的用户名
peerhost	客户端的 IPAddress

event	事件类型，固定为 "session.subscribed"
topic	MQTT 主题
qos	MQTT 消息的 QoS
timestamp	事件触发时间（ms）
node	事件触发所在节点

8）$events/session_unsubscribed (取消终端订阅成功)

event	事件类型，固定为 "session.unsubscribed"
clientid	消息目的 Client ID
username	消息目的用户名
peerhost	客户端的 IPAddress
topic	MQTT 主题
qos	MQTT 消息的 QoS
timestamp	事件触发时间（ms）
node	事件触发所在节点

（6）SQL 关键字和符号

1）SELECT – FROM – WHERE 语句

SELECT 语句用于决定最终的输出结果里的字段。比如:

下面 SQL 的输出结果中将只有两个字段 "a" 和 "b":

```
SELECT a, b FROM "t/#"
```

WHERE 语句用于对本事件中可用字段，或 SELECT 语句中定义的字段进行条件过滤。比如:

选取 username 为 'abc' 的终端发来的消息，输出结果为所有可用字段:

SELECT * FROM "" WHERE username = 'abc'

选取 clientid 为 'abc' 的终端发来的消息，输出结果将只有 cid 一个字段。

注意 cid 变量是在 SELECT 语句中定义的，故可在 WHERE 语句中使用:

SELECT clientid as cid FROM "" WHERE cid = 'abc'

选取 username 为 'abc' 的终端发来的消息，输出结果将只有 cid 一个字段。

注意虽然 SELECT 语句中只选取了 cid 一个字段，所有消息发布事件中的可用字段 (比如 clientid, username 等) 仍然可以在 WHERE 语句中使用:

SELECT clientid as cid FROM "" WHERE username = 'abc'

但下面这个 SQL 语句就不能工作了，因为变量 xyz 既不是消息发布事件中的可用字段，又没有在 SELECT 语句中定义:

SELECT clientid as cid FROM "" WHERE xyz = 'abc'

FROM 语句用于选择事件来源。如果是消息发布则填写消息的主题，如果是事件则填写对应的事件主题。

2）运算符号

函数名	函数作用	返回值
+	加法，或字符串拼接	加和，或拼接之后的字符串
–	减法	差值
*	乘法	乘积

/	除法	商值
div	整数除法	整数商值
mod	取模	模
=	比较两者是否完全相等。可用于比较变量和主题	true/false
=~	比较主题(topic)是否能够匹配到主题过滤器(topic filter)。只能用于主题匹配	true/false

（7）SQL 语句中可用的函数

1）数学函数

函数名	函数作用	参数	返回值
abs	绝对值	被操作数	绝对值
cos	余弦	被操作数	余弦值
cosh	双曲余弦	被操作数	双曲余弦值
acos	反余弦	被操作数	反余弦值
acosh	反双曲余弦	被操作数	反双曲余弦值
sin	正弦	被操作数	正弦值
sinh	双曲正弦	被操作数	双曲正弦值
asin	反正弦	被操作数	值
asinh	反双曲正弦	被操作数	反双曲正弦值
tan	正切	被操作数	正切值
tanh	双曲正切	被操作数	双曲正切值
atan	反正切	被操作数	反正切值
atanh	反双曲正切	被操作数	反双曲正切值
ceil	上取整	被操作数	整数值
floor	下取整	被操作数	整数值
round	四舍五入	被操作数	整数值
exp	幂运算	被操作数	e 的 x 次幂
power	指数运算	左操作数 x 2. 右操作数 y	x 的 y 次方
sqrt	平方根运算	被操作数	平方根
fmod	负点数取模函数	左操作数 2. 右操作数	模
log	以 e 为底对数	被操作数	值
log10	以 10 为底对数	被操作数	值
log2	以 2 为底对数	被操作数	值

2）数据类型判断函数

函数名	函数作用	参数	返回值
is_null	判断变量是否为空值	Data	Boolean 类型的数据。如果为空值

			(undefined) 则返回 true，否则返回 false
is_not_null	判断变量是否为非空值	Data	Boolean 类型的数据。如果为空值 (undefined) 则返回 false，否则返回 true
is_str	判断变量是否为 String 类型	Data	Boolean 类型的数据。
is_bool	判断变量是否为 Boolean 类型	Data	Boolean 类型的数据。
is_int	判断变量是否为 Integer 类型	Data	Boolean 类型的数据。
is_float	判断变量是否为 Float 类型	Data	Boolean 类型的数据。
is_num	判断变量是否为数字类型，包括 Integer 和 Float 类型	Data	Boolean 类型的数据。
is_map	判断变量是否为 Map 类型	Data	Boolean 类型的数据。
is_array	判断变量是否为 Array 类型	Data	Boolean 类型的数据。

3）数据类型转换函数

函数名	函数作用	参数	返回值
str	将数据转换为 String 类型	Data	String 类型的数据。无法转换将会导致 SQL 匹配失败
str_utf8	将数据转换为 UTF-8 String 类型	Data	UTF-8 String 类型的数据。无法转换将会导致 SQL 匹配失败
bool	将数据转换为 Boolean 类型	Data	Boolean 类型的数据。无法转换将会导致 SQL 匹配失败
int	将数据转换为整数类型	Data	整数类型的数据。无法转换将会导致 SQL 匹配失败
float	将数据转换为浮点型类型	Data	浮点型类型的数据。无法转换将会导致 SQL 匹配失败
map	将数据转换为 Map 类型	Data	Map 类型的数据。无法转换将会导致 SQL 匹配失败

4）字符串函数

函数名	函数作用	参数	返回值	举例
lower	转为小写	1. 原字符串	小写字符串	1. lower('AbC') = 'abc' 2. lower('abc') = 'abc'
upper	转为大写	1. 原字符串	大写字符串	1. upper('AbC') = 'ABC' 2. lower('ABC') = 'ABC'

trim	去掉左右空格	1. 原字符串	去掉空格后的字符串	1. trim(' hello ') = 'hello'
ltrim	去掉左空格	1. 原字符串	去掉空格后的字符串	1. ltrim(' hello ') = 'hello '
rtrim	去掉右空格	1. 原字符串	去掉空格后的字符串	1. rtrim(' hello ') = ' hello'
reverse	字符串反转	1. 原字符串	翻转后的字符串	1. reverse('hello') = 'olleh'
strlen	取字符串长度	1. 原字符串	整数值, 字符长度	1. strlen('hello') = 5
substr	取字符的子串	1. 原字符串 2. 起始位置. 注意: 下标从 0 开始	子串	1. substr('abcdef', 2) = 'cdef'
substr	取字符的子串	1. 原字符串 2. 起始位置 3. 要取出的子串长度. 注意: 下标从 0 开始	子串	1. substr('abcdef', 2, 3) = 'cde'
split	字符串分割	1. 原字符串 2. 分割符子串	分割后的字符串数组	1. split('a/b/ c', '/') = ['a', 'b', ' c']
split	字符串分割, 只查找左边第一个分隔符	1. 原字符串 2. 分割符子串 3. 'leading'	分割后的字符串数组	1. split('a/b/ c', '/', 'leading') = ['a', 'b/ c']
split	字符串分割, 只查找右边第一个分隔符	1. 原字符串 2. 分割符子串 3. 'trailing'	分割后的字符串数组	1. split('a/b/ c', '/', 'trailing') = ['a/b', ' c']
concat	字符串拼接	1. 左字符串 2. 右符子串	拼接后的字符串	1. concat('a', '/bc') = 'a/bc' 2. 'a' + '/bc' = 'a/bc'
tokens	字符串分解(按照指定字符串符分解)	1. 输入字符串 2. 分割符或字符串	分解后的字符串数组	1. tokens(' a/b/ c', '/') = [' a', 'b', ' c']

				2. tokens(' a/b/ c', '/ ') = ['a', 'b', 'c'] 3. tokens(' a/b/ c\n', '/ ') = ['a', 'b', 'c\n']
tokens	字符串分解(按照指定字符串和换行符分解)	1. 输入字符串 2. 分割符或字符串 3. 'nocrlf'	分解后的字符串数组	1. tokens(' a/b/ c\n', '/ ', 'nocrlf') = ['a', 'b', 'c'] 2. tokens(' a/b/ c\r\n', '/ ', 'nocrlf') = ['a', 'b', 'c']
sprintf	字符串格式化，格式字符串的用法详见 https://erlang.org/doc/man/io.htmlf write-1 里的 Format 部分	1. 格式字符串 2,3,4... 参数列表。参数个数不定	分解后的字符串数组	1. sprintf('hello, ~s!', 'steve') = 'hello, steve!' 2. sprintf('count: ~p~n', 100) = 'count: 100\n'
pad	字符串补足长度，补空格，从尾部补足	1. 原字符串 2. 字符总长度	补足后的字符串	1. pad('abc', 5) = 'abc '
pad	字符串补足长度，补空格，从尾部补足	1. 原字符串 2. 字符总长度 3. 'trailing'	补足后的字符串	1. pad('abc', 5, 'trailing') = 'abc '
pad	字符串补足长度，补空格，从两边补足	1. 原字符串 2. 字符总长度 3. 'both'	补足后的字符串	1. pad('abc', 5, 'both') = ' abc '
pad	字符串补足长度，补空格，从头部补足	1. 原字符串 2. 字符总长度 3. 'leading'	补足后的字符串	1. pad('abc', 5, 'leading') = ' abc'
pad	字符串补足长度，补指定字符，	1. 原字符串 2. 字符总长度 3.	补足后的字符串	1. pad('abc', 5, 'trailing',

	从尾部补足	'trailing' 4. 指定用于补足的字符		'*') = 'abc**' 2. pad('abc', 5, 'trailing', '*') = 'abc**'
pad	字符串补足长度，补指定字符，从两边补足	1. 原字符串 2. 字符总长度 3. 'both' 4. 指定用于补足的字符	补足后的字符串	1. pad('abc', 5, 'both', '*') = '*abc*' 2. pad('abc', 5, 'both', '*') = '*abc*'
pad	字符串补足长度，补指定字符，从头部补足	1. 原字符串 2. 字符总长度 3. 'leading' 4. 指定用于补足的字符	补足后的字符串	1. pad('abc', 5, 'leading', '*') = '**abc' 2. pad('abc', 5, 'leading', '*') = '**abc'
replace	替换字符串中的某子串，查找所有匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串	替换后的字符串	1. replace('ababef', 'ab', 'cd') = 'cdcdef'
replace	替换字符串中的某子串，查找所有匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'all'	替换后的字符串	1. replace('ababef', 'ab', 'cd', 'all') = 'cdcdef'
replace	替换字符串中的某子串，从尾部查找第一个匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'trailing'	替换后的字符串	1. replace('ababef', 'ab', 'cd', 'trailing') = 'abcdef'
replace	替换字符串中的某子串，从头部查找第一个匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'leading'	替换后的字符串	1. replace('ababef', 'ab', 'cd', 'leading') = 'cdabef'

regex_match	判断字符串是否与某正则表达式匹配	1. 原字符串 2. 正则表达式	true 或 false	1. regex_match('abc123', '[a-zA-Z1-9]*') = true
regex_replace	替换字符串中匹配到某正则表达式的子串	1. 原字符串 2. 正则表达式 3. 指定用于替换的字符串	替换后的字符串	1. regex_replace('ab1cd3ef', '[1-9]', '[&]') = 'ab[1]cd[3]ef' 2. regex_replace('ccefacef', 'c+', ':') = ':efa:ef'
ascii	返回字符对应的 ASCII 码	1. 字符	整数值，字符对应的 ASCII 码	1. ascii('a') = 97
find	查找并返回字符串中的某个子串，从头部查找	1. 原字符串 2. 要查找的子串	查抄到的子串，如找不到则返回空字符串	1. find('eeabcabcee', 'abc') = 'abcabcee'
find	查找并返回字符串中的某个子串，从头部查找	1. 原字符串 2. 要查找的子串 3. 'leading'	查抄到的子串，如找不到则返回空字符串	1. find('eeabcabcee', 'abc', 'leading') = 'abcabcee'
find	查找并返回字符串中的某个子串，从尾部查找	1. 原字符串 2. 要查找的子串 3. 'trailing'	查抄到的子串，如找不到则返回空字符串	1. find('eeabcabcee', 'abc', 'trailing') = 'abcee'

5）Map 函数

函数名	函数作用	参数	返回值
map_get	取 Map 中某个 Key 的值，如果没有则返回空值	1. Key 2. Map	Map 中某个 Key 的值。支持嵌套的 Key，比如 "a.b.c"

map_get	取 Map 中某个 Key 的值, 如果没有则返回指定默认值	1. Key 2. Map 3. Default Value	Map 中某个 Key 的值。支持嵌套的 Key, 比如 "a.b.c"
map_put	向 Map 中插入值	1. Key 2. Value 3. Map	插入后的 Map。支持嵌套的 Key, 比如 "a.b.c"

6）数组函数

函数名	函数作用	参数	返回值
nth	取第 n 个元素，下标从 1 开始	1. 原数组	第 n 个元素
length	获取数组的长度	1. 原数组	数组长度
sublist	取从第一个元素开始、长度为 len 的子数组。下标从 1 开始	1. 长度 len 2. 原数组	子数组
sublist	取从第 n 个元素开始、长度为 len 的子数组。下标从 1 开始	1. 起始位置 n 2. 长度 len 3. 原数组	子数组
first	取第 1 个元素。下标从 1 开始	1. 原数组	第 1 个元素
last	取最后一个元素。	1. 原数组	最后一个元素
contains	判断数据是否在数组里面	1. 数据 2. 原数组	Boolean 值

7）哈希函数

函数名	函数作用	参数	返回值
md5	求 MD5 值	数据	MD5 值
sha	求 SHA 值	数据	SHA 值
sha256	求 SHA256 值	数据	SHA256 值

8）编解码函数

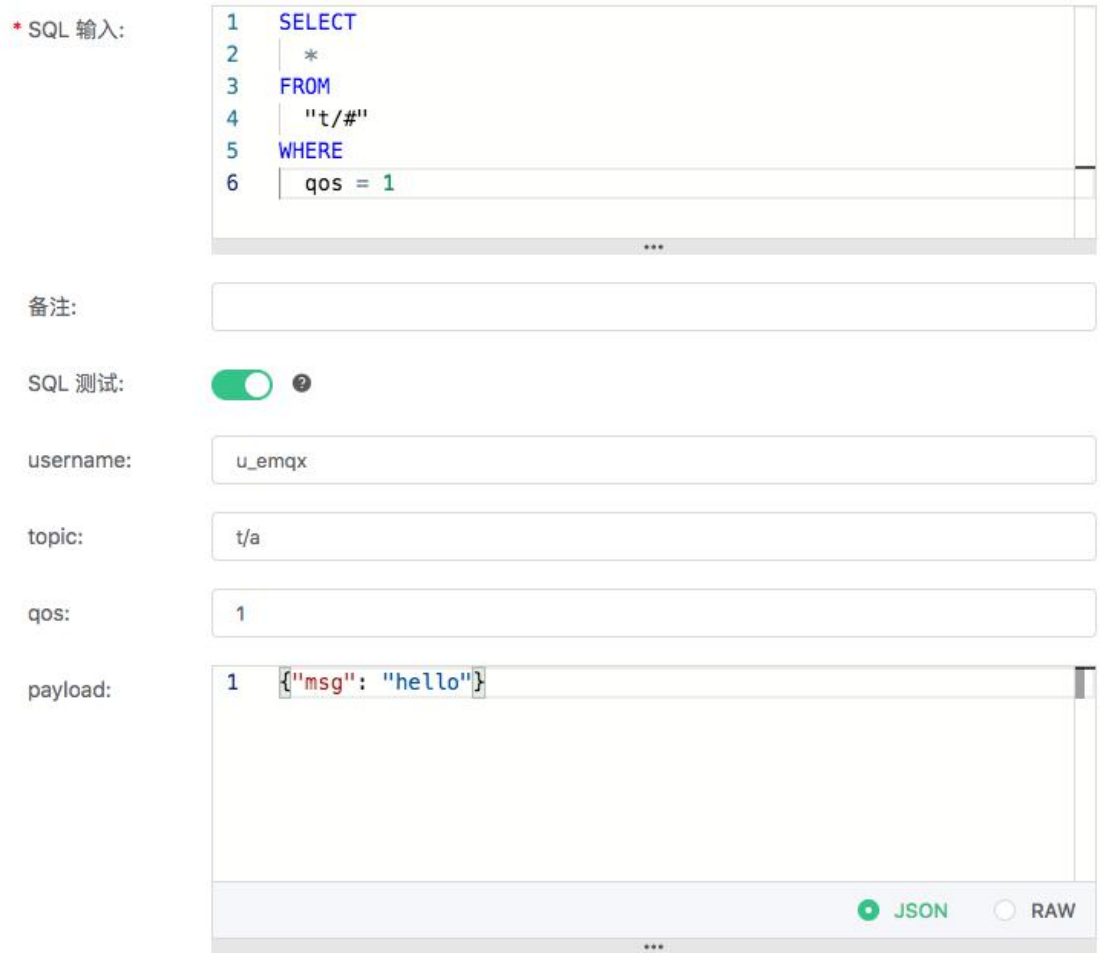
函数名	函数作用	参数	返回值
base64_encode	BASE64 编码	1. 数据	BASE64 字符串
base64_decode	BASE64 解码	1. BASE64 字符串	数据
json_encode	JSON 编码	1. JSON 字符串	内部 Map
json_decode	JSON 解码	1. 内部 Map	JSON 字符串
schema_encode	Schema 编码	1. Schema ID 2. 内部 Map	数据

schema_encode	Schema 编码	1. Schema ID 2. 内部 Map 3. Protobuf Message 名	数据
schema_decode	Schema 解码	1. Schema ID 2. 数据	内部 Map
schema_decode	Schema 解码	1. Schema ID 2. 数据 3. Protobuf Message 名	内部 Map

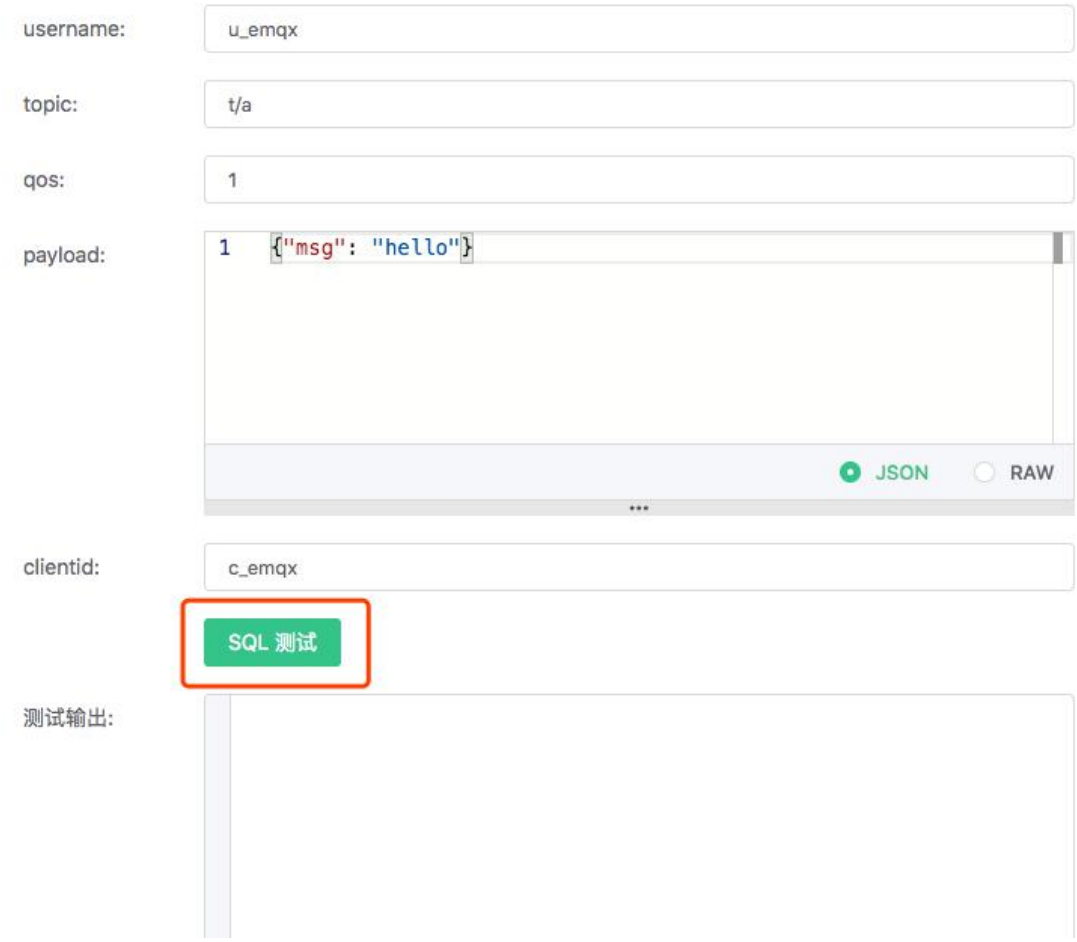
（8）在 Dashboard 中测试 SQL 语句

Dashboard 界面提供了 SQL 语句测试功能，通过给定的 SQL 语句和事件参数，展示 SQL 测试结果。

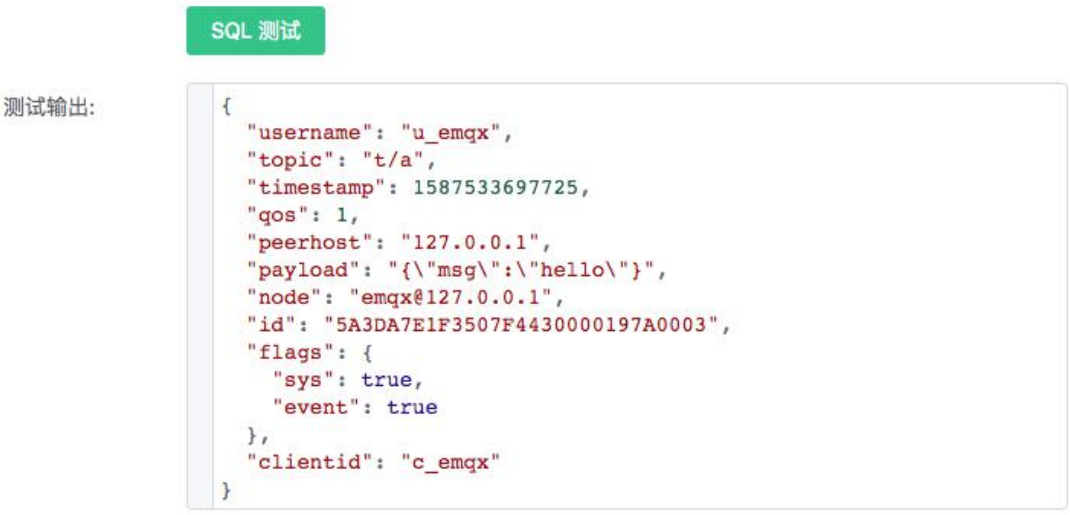
在创建规则界面，输入 规则 SQL，并启用 SQL 测试 开关:



1. 修改模拟事件的字段，或者使用默认的配置，点击 **测试** 按钮:



2. SQL 处理后的结果将在 **测试输出** 文本框里展示:



4.4 数据存储

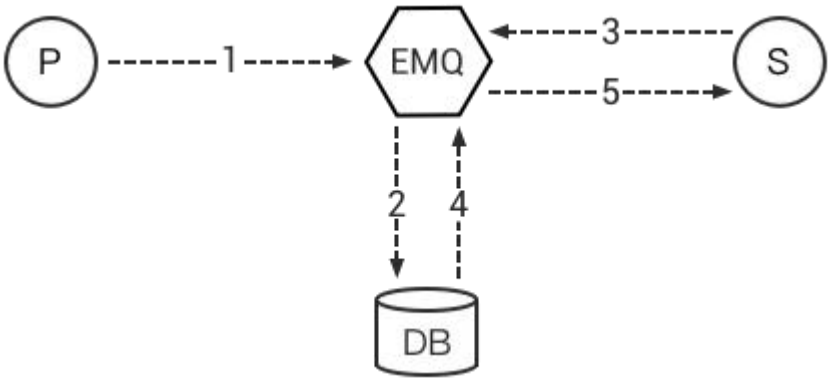
数据存储的主要使用场景包括将客户端上下线状态，订阅主题信息，消息内容，消息抵达后发送消息回执等操作记录到 Redis、MySQL、PostgreSQL、MongoDB、Cassandra 等各种数据库中。用户也可以通过订阅相关主题的方式来实现类似的功能，但是在企业版中内置了对这些持久化的支持；相比于前者，后者的执行效率更高，也能大大降低开发者的工作量。

WARNING

数据存储是 EMQ X Enterprise 专属功能。

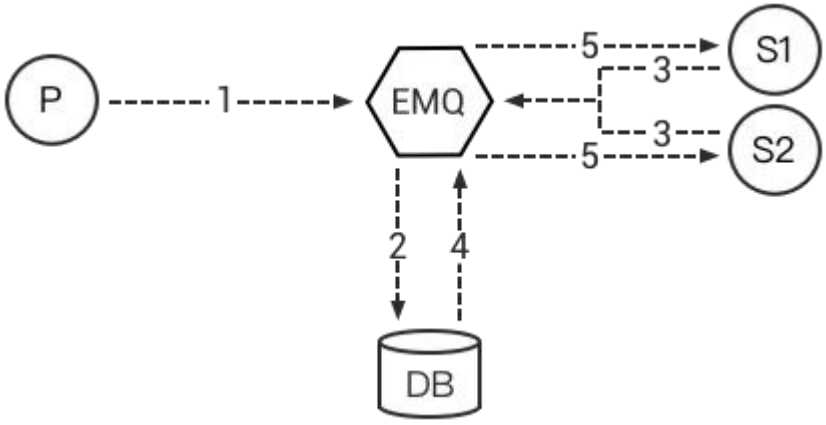
（1）数据存储设计

1）一对一消息存储



1. Publish 端发布一条消息；
2. Backend 将消息记录数据库中；
3. Subscribe 端订阅主题；
4. Backend 从数据库中获取该主题的消息；
5. 发送消息给 Subscribe 端；
6. Subscribe 端确认后 Backend 从数据库中移除该消息；

2）一对多消息存储



1. Publish 端发布一条消息；
2. Backend 将消息记录在数据库中；
3. Subscribe1 和 Subscribe2 订阅主题；
4. Backend 从数据库中获取该主题的消息；
5. 发送消息给 Subscribe1 和 Subscribe2；
6. Backend 记录 Subscribe1 和 Subscribe2 已读消息位置，下次获取消息从该位置开始。

（2）客户端在线状态存储

支持将设备上下线状态，直接存储到 Redis 或数据库。

（3）客户端代理订阅

支持代理订阅功能，设备客户端上线时，由存储模块直接从数据库，代理加载订阅主题。

（4）存储插件列表

EMQ X 支持 MQTT 消息直接存储 Redis、MySQL、PostgreSQL、MongoDB、Cassandra、DynamoDB、InfluxDB、OpenTSDB 数据库:

存储插件	配置文件	说明
emqx_backend_redis	emqx_backend_redis.conf	Redis 消息存储

emqx_backend_mysql	emqx_backend_mysql.conf	MySQL 消息存储
emqx_backend_pgsql	emqx_backend_pgsql.conf	PostgreSQL 消息存储
emqx_backend_mongo	emqx_backend_mongo.conf	MongoDB 消息存储
emqx_backend_cassa	emqx_backend_cassa.conf	Cassandra 消息存储
emqx_backend_dynamo	emqx_backend_dynamo.conf	DynamoDB 消息存储
emqx_backend_influxdb	emqx_backend_influxdb.conf	InfluxDB 消息存储
emqx_backend_opentsdb	emqx_backend_opentsdb.conf	OpenTSDB 消息存储

（5）配置步骤

EMQ X 中支持不同类型的数据库的持久化，虽然在一些细节的配置上有所不同，但是任何一种类型的持久化配置主要做两步操作：

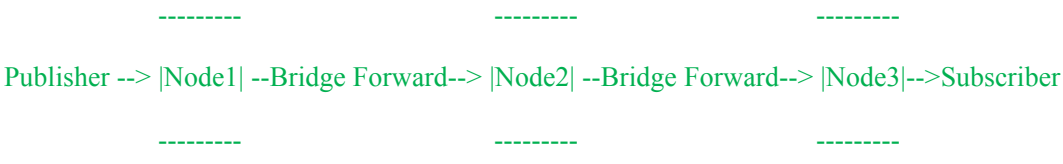
- 数据源连接配置：这部分主要用于配置数据库的连接信息，包括服务器地址，数据库名称，以及用户名和密码等信息，针对每种不同的数据库，这部分配置可能会有所不同；
- 事件注册与行为：根据不同的事件，你可以在配置文件中配置相关的行为（action），相关的行为可以是函数，也可以是 SQL 语句。

4.5 消息桥接

EMQ X 企业版桥接转发 MQTT 消息到 Kafka、RabbitMQ、Pulsar、RocketMQ、MQTT Broker 或其他 EMQ X 节点。

桥接是一种连接多个 EMQ X 或者其他 MQTT 消息中间件的方式。不同于集群，工作在桥接模式下的节点之间不会复制主题树和路由表。桥接模式所做的是：

按照规则把消息转发至桥接节点； 从桥接节点订阅主题，并在收到消息后在本节点/集群中转发该消息。



工作在桥接模式下和工作在集群模式下有不同的应用场景，桥接可以完成一些单纯使用集群

无法实现的功能：

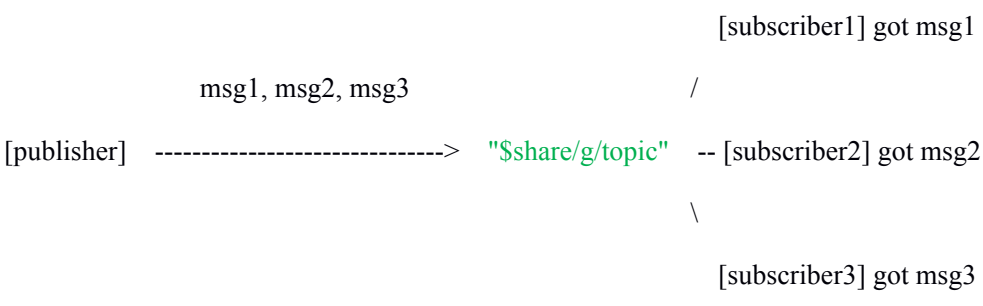
- 跨 VPC 部署。由于桥接不需要复制主题树和路由表，对于网络稳定性和延迟的要求相对于集群更低，桥接模式下不同的节点可以部署在不同的 VPC 上，客户端可以选择物理上比较近的节点连接，提高整个应用的覆盖能力。
- 支持异构节点。由于桥接的本质是对消息的转发和订阅，所以理论上凡是支持 MQTT 协议的消息中间件都可以被桥接到 EMQ X，甚至一些使用其他协议的消息服务，如果有协议适配器，也可以通过桥接转发消息过去。
- 提高单个应用的服务上限。由于内部的系统开销，单个的 EMQ X 有节点数上限。如果将多个集群桥接起来，按照业务需求设计桥接规则，可以将应用的服务上限再提高一个等级。 在具体应用中，一个桥接的发起节点可以被近似的看作一个远程节点的客户端。

（1）桥接插件列表

存储插件	配置文件	说明
emqx_bridge_mqtt	emqx_bridge_mqtt.conf	MQTT Broker 消息转发
emqx_bridge_kafka	emqx_bridge_kafka.conf	Kafka 消息队列
emqx_bridge_rabbit	emqx_bridge_rabbit.conf	RabbitMQ 消息队列
emqx_bridge_pulsar	emqx_bridge_pulsar.conf	Pulsar 消息队列
emqx_bridge_rocket	emqx_bridge_rocket.conf	RocketMQ 消息队列

4.6 共享订阅：

共享订阅是在多个订阅者之间实现负载均衡的订阅方式：



上图中，共享 3 个 subscriber 用共享订阅的方式订阅了同一个主题 $share/g/topic，其中 topic 是它们订阅的真实主题名，而 $share/g/ 是共享订阅前缀。EMQ X 支持两种格式的共

享订阅前缀：

示例	前缀	真实主题名
$queue/t/1	$queue/	t/1
$share/abc/t/1	$share/abc	t/1

（1）带群组的共享订阅

以 $share/<group-name> 为前缀的共享订阅是带群组的共享订阅：

group-name 可以为任意字符串，属于同一个群组内部的订阅者将以负载均衡接收消息，但 EMQ X 会向不同群组广播消息。

例如，假设订阅者 s1，s2，s3 属于群组 g1，订阅者 s4，s5 属于群组 g2。那么当 EMQ X 向这个主题发布消息 msg1 的时候：

- EMQ X 会向两个群组 g1 和 g2 同时发送 msg1
- s1，s2，s3 中只有一个会收到 msg1
- s4，s5 中只有一个会收到 msg1

```
                                      [s1]
          msg1                       /
[emqx]  ------>  "$share/g1/topic"    - [s2] got msg1
         |                           \
         |                             [s3]
         | msg1
          ---->  "$share/g2/topic"    --  [s4]
                                      \
                                       [s5] got msg1
```

（2）不带群组的共享订阅

以 $queue/ 为前缀的共享订阅是不带群组的共享订阅。它是 $share 订阅的一种特例，相当与所有订阅者都在一个订阅组里面：

```
                                        [s1] got msg1
        msg1,msg2,msg3                 /
[emqx]  --------------->  "$queue/topic" - [s2] got msg2
                                       \
                                        [s3] got msg3
```

（3）均衡策略与派发 Ack 配置

EMQ X 的共享订阅支持均衡策略与派发 Ack 配置：

```
# etc/emqx.conf

# 均衡策略
broker.shared_subscription_strategy = random

# 适用于 QoS1 QoS2 消息，启用时在其中一个组离线时，将派发给另一个组
broker.shared_dispatch_ack_enabled = false
```

均衡策略	描述
random	在所有订阅者中随机选择
round_robin	按照订阅顺序
sticky	一直发往上次选取的订阅者
hash	按照发布者 ClientID 的哈希值

> TIP
>
> 无论是单客户端订阅还是共享订阅都要注意客户端性能与消息接收速率，否则会引发消息堆积、客户端崩溃等错误。

4.7 $SYS 系统主题

EMQ X 周期性发布自身运行状态、消息统计、客户端上下线事件到以 $SYS/ 开头系统主题。

$SYS 主题路径以 $SYS/brokers/{node}/ 开头。{node} 是指产生该事件 / 消息所在的节点名称，例如：

```
$SYS/brokers/emqx@127.0.0.1/version
$SYS/brokers/emqx@127.0.0.1/uptime
```

$SYS 系统消息发布周期配置项：

```
broker.sys_interval = 1m
```

> WARNING
>
> EMQ X 默认**只允许**本机的 MQTT 客户端订阅 $SYS 主题，请参照 **内置 ACL** 修改发布订阅 ACL 规则。
>
> EMQ X 中 $SYS 主题中绝大部分数据都可以通过其他更耦合性更低的方式获取，设备上下线状态可通过 Webhook 获取，节点与集群状态可通过 **HTTP API - 统计指标** 获取。

（1）集群状态信息

主题	说明
$SYS/brokers	集群节点列表
$SYS/brokers/${node}/version	EMQ X 版本
$SYS/brokers/${node}/uptime	EMQ X 运行时间
$SYS/brokers/${node}/datetime	EMQ X 系统时间
$SYS/brokers/${node}/sysdescr	EMQ X 描述

（2）客户端上下线事件

$SYS 主题前缀: $SYS/brokers/${node}/clients/

主题（Topic）	说明
${clientid}/connected	上线事件。当任意客户端上线时，EMQ X 就会发布该主题的消息
${clientid}/disconnected	下线事件。当任意客户端下线时，EMQ X 就会发布该主题的消息

connected 事件消息的 Payload 解析成 JSON 格式如下:

```
{
    "username":"undefined",
    "ts":1582687922392,
    "sockport":1883,
```

```
    "proto_ver":5,
    "proto_name":"MQTT",
    "keepalive":300,
    "ipaddress":"127.0.0.1",
    "expiry_interval":0,
    "connected_at":1582687922,
    "connack":0,
    "clientid":"emqtt-8348fe27a87976ad4db3",
    "clean_start":true
}
```


disconnected 事件消息的 Payload 解析成 JSON 格式如下:


```
{
    "username":"undefined",
    "ts":1582688032203,
    "reason":"tcp_closed",
    "disconnected_at":1582688032,
    "clientid":"emqtt-8348fe27a87976ad4db3"
}
```


（3）系统统计（Statistics)

系统主题前缀: $SYS/brokers/${node}/stats/

1）客户端统计

主题（Topic）	说明
connections/count	当前客户端总数
connections/max	客户端数量历史最大值

2）订阅统计

主题（Topic）	说明
suboptions/count	当前订阅选项个数
suboptions/max	订阅选项总数历史最大值
subscribers/count	当前订阅者数量
subscribers/max	订阅者总数历史最大值
subscriptions/count	当前订阅总数
subscriptions/max	订阅数量历史最大值
subscriptions/shared/count	当前共享订阅个数
subscriptions/shared/max	当前共享订阅总数

3）主题统计

主题（Topic）	说明
topics/count	当前 Topic 总数
topics/max	Topic 数量历史最大值

4）路由统计

主题（Topic）	说明
routes/count	当前 Routes 总数

routes/max	Routes 数量历史最大值

topics/count 和 topics/max 与 routes/count 和 routes/max 数值上是相等的。

（4）收发流量 / 报文 / 消息统计

系统主题（Topic）前缀: $SYS/brokers/${node}/metrics/

1）收发流量统计

主题（Topic）	说明
bytes/received	累计接收流量
bytes/sent	累计发送流量

2）MQTT 报文收发统计

主题（Topic）	说明
packets/received	累计接收 MQTT 报文
packets/sent	累计发送 MQTT 报文
packets/connect	累计接收 MQTT CONNECT 报文
packets/connack	累计发送 MQTT CONNACK 报文
packets/publish/received	累计接收 MQTT PUBLISH 报文
packets/publish/sent	累计发送 MQTT PUBLISH 报文
packets/puback/received	累计接收 MQTT PUBACK 报文
packets/puback/sent	累计发送 MQTT PUBACK 报文
packets/puback/missed	累计丢失 MQTT PUBACK 报文
packets/pubrec/received	累计接收 MQTT PUBREC 报文
packets/pubrec/sent	累计发送 MQTT PUBREC 报文

packets/pubrec/missed	累计丢失 MQTT PUBREC 报文
packets/pubrel/received	累计接收 MQTT PUBREL 报文
packets/pubrel/sent	累计发送 MQTT PUBREL 报文
packets/pubrel/missed	累计丢失 MQTT PUBREL 报文
packets/pubcomp/received	累计接收 MQTT PUBCOMP 报文
packets/pubcomp/sent	累计发送 MQTT PUBCOMP 报文
packets/pubcomp/missed	累计丢失 MQTT PUBCOMP 报文
packets/subscribe	累计接收 MQTT SUBSCRIBE 报文
packets/suback	累计发送 MQTT SUBACK 报文
packets/unsubscribe	累计接收 MQTT UNSUBSCRIBE 报文
packets/unsuback	累计发送 MQTT UNSUBACK 报文
packets/pingreq	累计接收 MQTT PINGREQ 报文
packets/pingresp	累计发送 MQTT PINGRESP 报文
packets/disconnect/received	累计接收 MQTT DISCONNECT 报文
packets/disconnect/sent	累计接收 MQTT DISCONNECT 报文
packets/auth	累计接收 MQTT AUTH 报文

3）MQTT 消息收发统计

主题（Topic）	说明
messages/received	累计接收消息
messages/sent	累计发送消息
messages/expired	累计过期消息

messages/retained	Retained 消息总数
messages/dropped	丢弃消息总数
messages/forward	节点转发消息总数
messages/qos0/received	累计接收 QoS 0 消息
messages/qos0/sent	累计发送 QoS 0 消息
messages/qos1/received	累计接收 QoS 1 消息
messages/qos1/sent	累计发送 QoS 1 消息
messages/qos2/received	累计接收 QoS 2 消息
messages/qos2/sent	累计发送 QoS 2 消息
messages/qos2/expired	QoS 2 过期消息总数
messages/qos2/dropped	QoS 2 丢弃消息总数

（5）Alarms - 系统告警

系统主题（Topic）前缀: $SYS/brokers/${node}/alarms/

主题（Topic）	说明
alert	新产生的告警
clear	被清除的告警

（6）Sysmon - 系统监控

系统主题（Topic）前缀: $SYS/brokers/${node}/sysmon/

主题（Topic）	说明
long_gc	GC 时间过长警告
long_schedule	调度时间过长警告
large_heap	Heap 内存占用警告

busy_port	Port 忙警告
busy_dist_port	Dist Port 忙警告

4.8 黑名单

EMQ X 为用户提供了黑名单功能，用户可以通过相关的 HTTP API 将指定客户端加入黑名单以拒绝该客户端访问，除了客户端标识符以外，还支持直接封禁用户名甚至 IP 地址。

相关 HTTP API 的具体使用方法，请参见 HTTP API - 黑名单。

TIP: 黑名单只适用于少量客户端封禁需求, 如果有大量客户端需要认证管理, 请使用 认证 功能。

在黑名单功能的基础上，EMQ X 支持自动封禁那些被检测到短时间内频繁登录的客户端，并且在一段时间内拒绝这些客户端的登录，以避免此类客户端过多占用服务器资源而影响其他客户端的正常使用。

需要注意的是，自动封禁功能只封禁客户端标识符，并不封禁用户名和 IP 地址，即该机器只要更换客户端标识符就能够继续登录。

此功能默认关闭，用户可以在 emqx.conf 配置文件中将 enable_flapping_detect 配置项设为 on 以启用此功能。

```
zone.external.enable_flapping_detect = off
```

用户可以为此功能调整触发阈值和封禁时长，对应配置项如下：

```
flapping_detect_policy = 30, 1m, 5m
```

此配置项的值以 , 分隔，依次表示客户端离线次数，检测的时间范围以及封禁时长，因此上述默认配置即表示如果客户端在 1 分钟内离线次数达到 30 次，那么该客户端使用的客户端标识符将被封禁 5 分钟。当然你也可以使用其他诸如秒、小时在内的时间单位，关于这部分内容，请参见 配置说明。

4.9 Webhook

WebHook 是由 emqx_web_hook (opens new window)插件提供的 **将 EMQ X 中的钩子事件通知到某个 Web 服务** 的功能。

WebHook 的内部实现是基于 钩子，但它更靠近顶层一些。它通过在钩子上的挂载回调

函数，获取到 EMQ X 中的各种事件，并转发至 emqx_web_hook 中配置的 Web 服务器。

以 客户端成功接入(client.connected) 事件为例，其事件的传递流程如下：

TIP：WebHook 对于事件的处理是单向的，**它仅支持将 EMQ X 中的事件推送给 Web 服务，并不关心 Web 服务的返回**。 借助 Webhook 可以完成设备在线、上下线记录，订阅与消息存储、消息送达确认等诸多业务。

（1）配置项

Webhook 的配置文件位于 etc/plugins/emqx_web_hook.conf，配置项的详细说明可以查看 配置项。

（2）触发规则

在 etc/plugins/emqx_web_hooks.conf 可配置触发规则，其配置的格式如下：

（3）Event 触发事件

目前支持以下事件：

名称	说明	执行时机
client.connect	处理连接报文	服务端收到客户端的连接报文时
client.connack	下发连接应答	服务端准备下发连接应答报文时
client.connected	成功接入	客户端认证完成并成功接入系统后
client.disconnected	连接断开	客户端连接层在准备关闭时
client.subscribe	订阅主题	收到订阅报文后，执行 `client.check_acl` 鉴权前
client.unsubscribe	取消订阅	收到取消订阅报文后
session.subscribed	会话订阅主题	完成订阅操作后

名称	说明	执行时机
session.unsubscribed	会话取消订阅	完成取消订阅操作后
message.publish	消息发布	服务端在发布（路由）消息前
message.delivered	消息投递	消息准备投递到客户端前
message.acked	消息回执	服务端在收到客户端发回的消息 ACK 后
message.dropped	消息丢弃	发布出的消息被丢弃后

（4）Number

同一个事件可以配置多个触发规则，配置相同的事件应当依次递增。

（5）Rule

触发规则，其值为一个 JSON 字符串，其中可用的 Key 有：

- action：字符串，取固定值
- topic：字符串，表示一个主题过滤器，操作的主题只有与该主题匹配才能触发事件的转发

例如，我们只将与 a/b/c 和 foo/ 主题匹配的消息转发到 Web 服务器上，其配置应该为：

这样 Webhook 仅会转发与 a/b/c 和 foo/ 主题匹配的消息，例如 foo/bar 等，而不是转发 a/b/d 或 fo/bar。

（6）Webhook 事件参数

事件触发时 Webhook 会按照配置将每个事件组成一个 HTTP 请求发送到 url 所配置的 Web 服务器上。其请求格式为：

对于不同的事件，请求 Body 体内容有所不同，下表列举了各个事件中 Body 的参数列表：

1）client.connect

Key	类型	说明
action	string	事件名称 固定为："client_connect"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
ipaddress	string	客户端源 IP 地址
keepalive	integer	客户端申请的心跳保活时间
proto_ver	integer	协议版本号

2）client.connack

Key	类型	说明
action	string	事件名称 固定为："client_connack"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
ipaddress	string	客户端源 IP 地址
keepalive	integer	客户端申请的心跳保活时间

Key	类型	说明
proto_ver	integer	协议版本号
conn_ack	string	"success" 表示成功，其它表示失败的原因

3）client.connected

Key	类型	说明
action	string	事件名称 固定为："client_connected"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
ipaddress	string	客户端源 IP 地址
keepalive	integer	客户端申请的心跳保活时间
proto_ver	integer	协议版本号
connected_at	integer	时间戳(秒)

4）client.disconnected

Key	类型	说明

Key	类型	说明
action	string	事件名称 固定为："client_disconnected"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
reason	string	错误原因

5）client.subscribe

Key	类型	说明
action	string	事件名称 固定为："client_subscribe"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
topic	string	将订阅的主题
opts	json	订阅参数

6）opts 包含

Key	类型	说明

Key	类型	说明
qos	enum	QoS 等级，可取 `0` `1` `2`

7）client.unsubscribe

Key	类型	说明
action	string	事件名称 固定为："client_unsubscribe"
clientid	string	客户端 ClientId
username	string	客户端 Username，不存在时该值为 "undefined"
topic	string	取消订阅的主题

session.subscribed：同 `client.subscribe`，action 为 `session_subscribed`

session.unsubscribed：同 `client.unsubscribe`，action 为 `session_unsubscribe`

session.terminated：同 `client.disconnected`，action 为 `session_terminated`

8）message.publish

Key	类型	说明

Key	类型	说明
action	string	事件名称 固定为："message_publish"
from_client_id	string	发布端 ClientId
from_username	string	发布端 Username，不存在时该值为 "undefined"
topic	string	取消订阅的主题
qos	enum	QoS 等级，可取 `0` `1` `2`
retain	bool	是否为 Retain 消息
payload	string	消息 Payload
ts	integer	消息的时间戳(毫秒)

9）message.delivered

Key	类型	说明
action	string	事件名称 固定为："message_delivered"
clientid	string	接收端 ClientId
username	string	接收端 Username，不存在时该值为 "undefined"

Key	类型	说明
from_client_id	string	发布端 ClientId
from_username	string	发布端 Username，不存在时该值为 "undefined"
topic	string	取消订阅的主题
qos	enum	QoS 等级，可取 `0` `1` `2`
retain	bool	是否为 Retain 消息
payload	string	消息 Payload
ts	integer	消息时间戳(毫秒)

10）message.acked

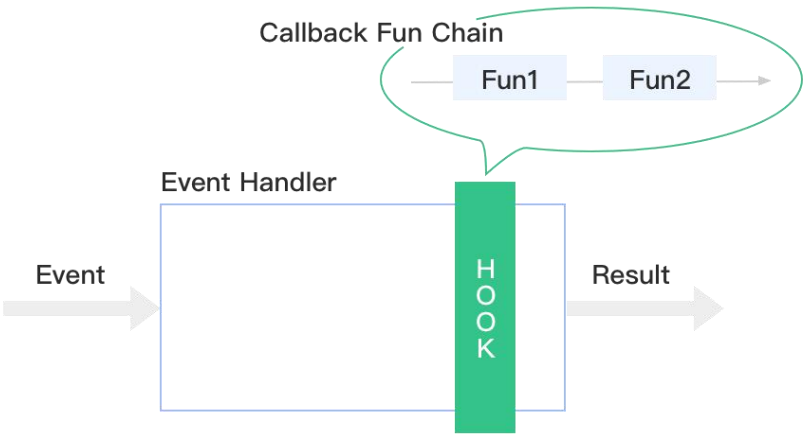
Key	类型	说明
action	string	事件名称 固定为："message_acked"
clientid	string	接收端 ClientId
from_client_id	string	发布端 ClientId
from_username	string	发布端 Username，不存在时该值为 "undefined"

Key	类型	说明
topic	string	取消订阅的主题
qos	enum	QoS 等级，可取 `0` `1` `2`
retain	bool	是否为 Retain 消息
payload	string	消息 Payload
ts	integer	消息时间戳(毫秒)

4.10 钩子

钩子(Hooks) 是 EMQ X 提供的一种机制，它通过拦截模块间的函数调用、消息传递、事件传递来修改或扩展系统功能。

简单来讲，该机制目的在于增强软件系统的扩展性、方便与其他三方系统的集成、或者改变其系统原有的默认行为。如：



当系统中不存在 钩子 (Hooks) 机制时，整个事件处理流程 从 事件 (Event) 的输入，到处理 (Handler)，再到完成后的返回 结果 (Result) 对于系统外部而讲，都是不可见、且无法

修改的。

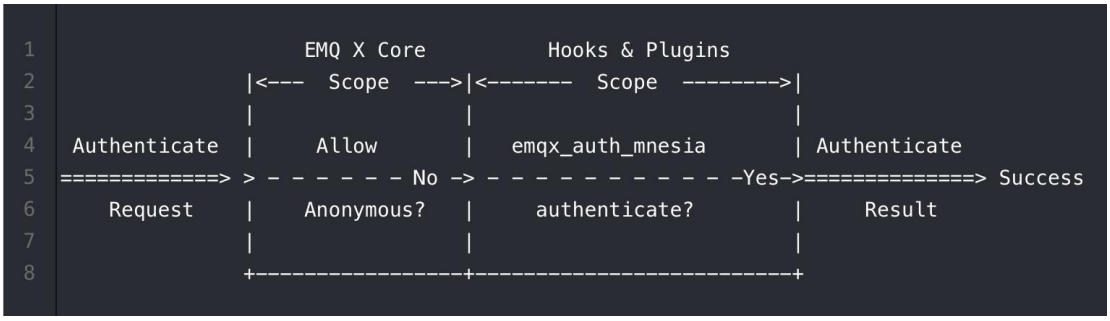
而在这个过程中加入一个可挂载函数的点 (HookPoint)，允许外部插件挂载多个回调函数，形成一个调用链。达到对内部事件处理过程的扩展和修改。

系统中常用到的认证插件则是按照该逻辑进行实现的。以最简单的 emqx_auth_mnesia (opens new window)为例：

在只开启 emqx_auth_mnesia 认证插件，且关闭匿名用户登录时。按照上图对事件的处理逻辑可知，此时认证模块的逻辑为：

1. 收到用户认证请求 (Authenticate)
2. 读取 是否允许匿名登录 参数，得到 拒绝登录
3. 执行 认证事件的钩子，即回调到 emqx_auth_mnesia 插件中，假设其认为此次登录合法，得到 允许登录
4. 返回 认证成功，成功接入系统

即，如下图所示：



因此，在 EMQ X 中，钩子 (Hooks) 这种机制极大地方便了系统的扩展。我们不需要修改 emqx (opens new window)核心代码，仅需要在特定的位置埋下 挂载点 (HookPoint)，便能允许外部插件扩展 EMQ X 的各种行为。

对于实现者来说仅需要关注：

1. 挂载点 (HookPoint) 的位置：包括其作用、执行的时机、和如何挂载和取消挂载。
2. 回调函数 的实现：包括回调函数的入参个数、作用、数据结构等，及返回值代表的含义。
3. 了解回调函数在 链 上执行的机制：包括回调函数执行的顺序，及如何提前终止链的执行。

如果你是在开发扩展插件中使用钩子，你应该能 完全地明白这三点，且尽量不要在钩

子内部使用阻塞函数，这会影响系统的吞吐。

（1）回调链

单个 挂载点 上可能会存在多个插件都需要关心该事件并执行相应操作，所以每个 挂载点 上都可能会存在多个回调函数。

我们称这种由多个回调函数顺序执行所构成的链为 回调链 (Callback Functions Chain)。

回调链 目前按照 职责链(Chain-of-Responsibility) (opens new window)的理念进行实现。为了满足钩子的功能和使用的灵活性，它必须具有以下属性：

- 回调链 上的回调函数必须按某种先后顺序执行。
- 回调链 一定会存在一个输入、和输出 (在通知类事件输出则是非必须的，例如 “某客户端已成功登陆”)。
- 回调链 具有传递性，意思是指，链会将输入给链的入参输入给第一个回调函数，第一个回调函数的返回值会传递给第二个回调函数，直到最后一个函数，最后一个函数的返回值则为整个链的返回值。
- 回调链 需要允许其上面的函数 提前终止链 和 忽略本次操作。
- 提前终止：本函数执行完成后，直接终止链的执行。忽略链上后续所有的回调函数。例如：某认证插件认为，此类客户端允许登录后便不需要再检查其他认证插件，所以需要提前终止。
- 忽略本次操作：不修改链上的处理结果，直接透传给下一个回调函数。例如：存在多个认证插件的情况下，某认证插件认为，此类客户端不属于其认证范围，所以我不需要修改认证结果，应当忽略本次操作，直接将前一个函数的返回值传递给链上的下一个函数。

由此，我们可以得到一个链的设计简图：

该图的含义是指：

1. 链的入参为只读的 Args 与用于链上的函数修改的参数 Acc

2. 链无论以何种方式终止执行，其返回值均为新的 Acc
3. 图中链上一共注册了三个回调函数；分别为 Fun1 Fun2 Fun3 并按所表示的顺序执行
4. 回调函数执行顺序，由一个优先级确定，同一优先级的按挂载的先后顺序执行
5. 回调函数通过返回：
   - ok：忽略本次操作，以只读的 Args 和上个函数返回的 Acc 继续链的执行
   - {ok, NewAcc}：执行了某些操作，修改了 Acc 内容，以只读的 Args 和新的 NewAcc 继续链的执行
6. 回调函数也可通过返回：
   - stop：表示终止链的传递，立即返回上个函数的结果 Acc
   - {stop, NewAcc}：表示终止链的传递，立即返回本次修改的结果 NewAcc

以上为回调链的主要设计理念，它规范了钩子上的回调函数的执行逻辑。

接下来 挂载点，回调函数 两节中，对于钩子的所有操作都是依赖于 emqx (opens new window)提供的 Erlang 代码级的 API。他们是整个钩子逻辑实现的基础。如需寻求：钩子和 HTTP 服务器的应用，参见： WebHook

（2）挂载点

EMQ X 以一个客户端在其生命周期内的关键活动为基础，预置了大量的 挂载点 (HookPoint)。目前系统中预置的挂载点有：

名称	说明	执行时机
client.connect	处理连接报文	服务端收到客户端的连接报文时
client.connack	下发连接应答	服务端准备下发连接应答报文时
client.connected	成功接入	客户端认证完成并成功接入系统后
client.disconnected	连接断开	客户端连接层在准备关闭时
client.authenticate	连接认证	执行完 `client.connect` 后
client.check_acl	ACL 鉴权	执行 发布/订阅 操作前
client.subscribe	订阅主题	收到订阅报文后，执行 `client.check_acl` 鉴权前
client.unsubscribe	取消订阅	收到取消订阅报文后

session.created	会话创建	`client.connected` 执行完成，且创建新的会话后
session.subscribed	会话订阅主题	完成订阅操作后
session.unsubscribed	会话取消订阅	完成取消订阅操作后
session.resumed	会话恢复	`client.connected` 执行完成，且成功恢复旧的会话信息后
session.discarded	会话被移除	会话由于被**移除**而终止后
session.takeovered	会话被接管	会话由于被**接管**而终止后
session.terminated	会话终止	会话由于其他原因被终止后
message.publish	消息发布	服务端在发布（路由）消息前
message.delivered	消息投递	消息准备投递到客户端前
message.acked	消息回执	服务端在收到客户端发回的消息 ACK 后
message.dropped	消息丢弃	发布出的消息被丢弃后

TIP

- **会话被移除** 是指：当客户端以 清除会话 的方式登入时，如果服务端中已存在该客户端的会话，那么旧的会话就会被丢弃。
- **会话被接管** 是指：当客户端以 保留会话 的方式登入时，如果服务端中已存在该客户端的会话，那么旧的会话就会被新的连接所接管。

4.10.2 挂载与取消挂载

EMQ X 提供了 API 进行钩子的挂载与取消挂载的操作。

1) 挂载：

```
%% Name: 钩子的名称（挂载点）如：'client.authenticate'
%% {Module, Function, Args}: 回调函数的模块、方法、和附加参数
%% Priority: 优先级，整数；不提供则默认为 0
emqx:hook(Name, {Module, Function, Args}, Priority).
```

挂载完成后，回调函数会按优先级从大到小执行，同一优先级按挂载的先后顺序执行。所有官方插件挂载的钩子优先级都为 0。

2）取消挂载：

```
%% Name: 钩子的名称（挂载点）如：'client.authenticate'
%% {Module, Function}: 回调函数的模块、方法
emqx:unhook(Name, {Module, Function}).
```

（2）回调函数

回调函数的入参及返回值要求，见下表：

(参数数据结构参见：emqx_types.erl (opens new window))

名称	入参	返回
client.connect	ConnInfo：客户端连接层参数 Props：MQTT v5.0 连接报文的 Properties 属性	新的 Props
client.connack	ConnInfo：客户端连接层参数 Rc：返回码 Props: MQTT v5.0 连接应答报文的 Properties 属性	新的 Props
client.connected	ClientInfo: 客户端信息参数 ConnInfo：客户端连接层参数	–
client.disconnected	ClientInfo：客户端信息参数 ConnInfo：客户端连接层参数 ReasonCode：错误码	–
client.authenticate	ClientInfo：客户端信息参数 AuthResult：认证结果	新的 AuthResult
client.check_acl	ClientInfo：客户端信息参数 Topic：发布/订阅的主题 PubSub: 发布或订阅 ACLResult：鉴权结果	新的 ACLResult
client.subscribe	ClientInfo：客户端信息参数	新

	Props： MQTT v5.0 订阅报文的 Properties 参数 TopicFilters：需订阅的主题列表	的 TopicFilters
client.unsubscribe	ClientInfo：客户端信息参数 Props： MQTT v5.0 取消订阅报文的 Properties 参数 TopicFilters：需取消订阅的主题列表	新的 TopicFilters
session.created	ClientInfo：客户端信息参数 SessInfo：会话信息	–
session.subscribed	ClientInfo：客户端信息参数 Topic：订阅的主题 SubOpts：订阅操作的配置选项	–
session.unsubscribed	ClientInfo：客户端信息参数 Topic：取消订阅的主题 SubOpts：取消订阅操作的配置选项	–
session.resumed	ClientInfo：客户端信息参数 SessInfo：会话信息	–
session.discarded	ClientInfo：客户端信息参数 SessInfo：会话信息	–
session.takeovered	ClientInfo：客户端信息参数 SessInfo：会话信息	
session.terminated	ClientInfo：客户端信息参数 Reason：终止原因 SessInfo：会话信息	–
message.publish	Message：消息对象	新的 Message
message.delivered	ClientInfo：客户端信息参数 Message：消息对象	新的 Message
message.acked	ClientInfo：客户端信息参数 Message：消息对象	–
message.dropped	Message：消息对象 By：被谁丢弃 Reason：丢弃原因	–

具体对于这些钩子的应用，参见：emqx_plugin_template

4.11 指标监控

EMQ X 为用户提供了指标监控功能，允许用户以及运维人员根据这些指标来了解当前服务状态。指标监控功能强制启用，但此功能拥有很高的性能，用户不必担心影响高吞吐场景下的系统性能。

EMQ X 为用户提供了多种查看指标与状态的手段。最直接的，用户可以在 EMQ X Dashboard 的 Overview 页面看到这些数据。

如果不方便访问 Dashboard，你还可以通过 HTTP API 和系统主题消息来获取这些数据，具体操作方法分别参见 HTTP API 与 $SYS 系统主题。

TIP:EMQ X 提供 emqx_statsd (opens new window)插件，用于将系统的监控数据输出到第三方的监控系统中，使用示例参考 Prometheus 监控告警。

（1）Metrics & Stats

EMQ X 将指标分为了 Metrics 与 Stats 两种。Metrics 通常指那些只会单调递增的数据，例如发送字节数量、发送报文数量。EMQ X 目前提供的 Metrics 覆盖了字节、报文、消息和事件四个维度。Stats 则通常指那些成对出现的数据，包括当前值和历史最大值，例如当前订阅数量和订阅历史最大数量。

从 v4.1.0 版本开始，EMQ X 增加了针对指定主题的 Metrics 统计，包括消息收发数量和**收发速率**。我们提供了新建主题统计、取消主题统计和返回指定主题统计信息的 HTTP API，参见 HTTP API，你也可以直接在 Dashboard -> Analysis -> Topic Metrics 页面进行相关操作。

（2）Metrics

1）字节

Name	Data Type	Description
bytes.received	Integer	接收字节数量
bytes.sent	Integer	发送字节数量

2）报文

Name	Data Type	Description
packets.received	Integer	接收的报文数量
packets.sent	Integer	发送的报文数量
packets.connect.received	Integer	接收的 CONNECT 报文数量
packets.connack.auth_error	Integer	发送的原因码为 0x86 和 0x87 的 CONNACK 报文数量
packets.connack.error	Integer	发送的原因码不为 0x00 的 CONNACK 报文数量，此指标的值大于等于 `packets.connack.auth_error` 的值
packets.connack.sent	Integer	发送的 CONNACK 报文数量
packets.publish.received	Integer	接收的 PUBLISH 报文数量
packets.publish.sent	Integer	发送的 PUBLISH 报文数量
packets.publish.inuse	Integer	接收的报文标识符已被占用的 PUBLISH 报文数量
packets.publish.auth_error	Integer	接收的未通过 ACL 检查的 PUBLISH 报文数量
packets.publish.error	Integer	接收的无法被发布的 PUBLISH 报文数量
packets.publish.dropped	Integer	超出接收限制而被丢弃的 PUBLISH 报文数量
packets.puback.received	Integer	接收的 PUBACK 报文数量
packets.puback.sent	Integer	发送的 PUBACK 报文数量
packets.puback.inuse	Integer	接收的报文标识符已被占用的 PUBACK 报文数量
packets.puback.missed	Integer	接收的未知报文标识符 PUBACK 报文数量
packets.pubrec.received	Integer	接收的 PUBREC 报文数量
packets.pubrec.sent	Integer	发送的 PUBREC 报文数量
packets.pubrec.inuse	Integer	接收的报文标识符已被占用的 PUBREC 报文数量
packets.pubrec.missed	Integer	接收的未知报文标识符 PUBREC 报文数量
packets.pubrel.received	Integer	接收的 PUBREL 报文数量
packets.pubrel.sent	Integer	发送的 PUBREL 报文数量
packets.pubrel.missed	Integer	接收的未知报文标识符 PUBREL 报文数量
packets.pubcomp.received	Integer	接收的 PUBCOMP 报文数量
packets.pubcomp.sent	Integer	发送的 PUBCOMP 报文数量
packets.pubcomp.inuse	Integer	接收的报文标识符已被占用的 PUBCOMP 报文数量
packets.pubcomp.missed	Integer	发送的 PUBCOMP 报文数量
packets.subscribe.received	Integer	接收的 SUBSCRIBE 报文数量
packets.subscribe.error	Integer	接收的订阅失败的 SUBSCRIBE 报文数量
packets.subscribe.auth_error	Integer	接收的未通过 ACL 检查的 SUBACK 报文数量
packets.suback.sent	Integer	发送的 SUBACK 报文数量
packets.unsubscribe.received	Integer	接收的 UNSUBSCRIBE 报文数量
packets.unsubscribe.error	Integer	接收的取消订阅失败的 UNSUBSCRIBE 报文数量

packets.unsuback.sent	Integer	发送的 UNSUBACK 报文数量
packets.pingreq.received	Integer	接收的 PINGREQ 报文数量
packets.pingresp.sent	Integer	发送的 PUBRESP 报文数量
packets.disconnect.received	Integer	接收的 DISCONNECT 报文数量
packets.disconnect.sent	Integer	发送的 DISCONNECT 报文数量
packets.auth.received	Integer	接收的 AUTH 报文数量
packets.auth.sent	Integer	发送的 AUTH 报文数量

3）消息（PUBLISH 报文）

Name	Data Type	Description
delivery.dropped.too_large	Integer	发送时由于长度超过限制而被丢弃的消息数量
delivery.dropped.queue_full	Integer	发送时由于消息队列满而被丢弃的 QoS 不为 0 的消息数量
delivery.dropped.qos0_msg	Integer	发送时由于消息队列满而被丢弃的 QoS 为 0 的消息数量
delivery.dropped.expired	Integer	发送时由于消息过期而被丢弃的消息数量
delivery.dropped.no_local	Integer	发送时由于 `No Local` 订阅选项而被丢弃的消息数量
delivery.dropped	Integer	发送时丢弃的消息总数
messages.delayed	Integer	EMQ X 存储的延迟发布的消息数量
messages.delivered	Integer	EMQ X 内部转发到订阅进程的消息数量
messages.dropped	Integer	EMQ X 内部转发到订阅进程前丢弃的消息总数
messages.dropped.expired	Integer	接收时由于消息过期而被丢弃的消息数量
messages.dropped.no_subscribers	Integer	由于没有订阅者而被丢弃的消息数量
messages.forward	Integer	向其他节点转发的消息数量
messages.publish	Integer	除系统消息外发布的消息数量
messages.qos0.received	Integer	接收来自客户端的 QoS 0 消息数量
messages.qos2.received	Integer	接收来自客户端的 QoS 1 消息数量
messages.qos1.received	Integer	接收来自客户端的 QoS 2 消息数量

messages.qos0.sent	Integer	发送给客户端的 QoS 0 消息数量
messages.qos1.sent	Integer	发送给客户端的 QoS 1 消息数量
messages.qos2.sent	Integer	发送给客户端的 QoS 2 消息数量
messages.received	Integer	接收来自客户端的消息数量，等于 messages.qos0.received，messages.qos1.received 与 messages.qos2.received 之和
messages.sent	Integer	发送给客户端的消息数量，等于 messages.qos0.sent，messages.qos1.sent 与 messages.qos2.sent 之和
messages.retained	Integer	EMQ X 存储的保留消息数量
messages.acked	Integer	已经应答的消息数量

4）事件

Name	Data Type	Description
actions.failure	Integer	规则引擎 action 执行成功次数
actions.success	Integer	规则引擎 action 执行失败次数
rules.matched	Integer	规则的匹配次数
client.auth.anonymous	Integer	客户端最终匿名形式登录的次数
client.connect	Integer	client.connect 钩子触发次数
client.authenticate	Integer	client.authenticate 钩子触发次数
client.connack	Integer	client.connack 钩子触发次数
client.connected	Integer	client.connected 钩子触发次数
client.disconnected	Integer	client.disconnected 钩子触发次数
client.check_acl	Integer	client.check_acl 钩子触发次数
client.subscribe	Integer	client.subscribe 钩子触发次数
client.unsubscribe	Integer	client.unsubscribe 钩子触发次数
session.created	Integer	session.created 钩子触发次数
session.discarded	Integer	session.discarded 钩子触发次数
session.resumed	Integer	session.resumed 钩子触发次数
session.takeovered	Integer	session.takeovered 钩子触发次数
session.terminated	Integer	session.terminated 钩子触发次数

5）Stats

Name	Data Type	Description
connections.count	Integer	当前连接数量
connections.max	Integer	连接数量的历史最大值

channels.count	Integer	即 `sessions.count`
channels.max	Integer	即 `session.max`
sessions.count	Integer	当前会话数量
sessions.max	Integer	会话数量的历史最大值
topics.count	Integer	当前主题数量
topics.max	Integer	主题数量的历史最大值
suboptions.count	Integer	即 `subscriptions.count`
suboptions.max	Integer	即 `subscriptions.max`
subscribers.count	Integer	当前订阅者数量
subscribers.max	Integer	订阅者数量的历史最大值
subscriptions.count	Integer	当前订阅数量，包含共享订阅
subscriptions.max	Integer	订阅数量的历史最大值
subscriptions.shared.count	Integer	当前共享订阅数量
subscriptions.shared.max	Integer	共享订阅数量的历史最大值
routes.count	Integer	当前路由数量
routes.max	Integer	路由数量的历史最大值
retained.count	Integer	当前保留消息数量
retained.max	Integer	保留消息的历史最大值

4.12 速率限制

EMQ X 提供对接入速度、消息速度的限制：当客户端连接请求速度超过指定限制的时候，暂停新连接的建立；当消息接收速度超过指定限制的时候，暂停接收消息。

速率限制是一种 backpressure 方案，从入口处避免了系统过载，保证了系统的稳定和可预测的吞吐。速率限制可在 etc/emqx.conf 中配置:

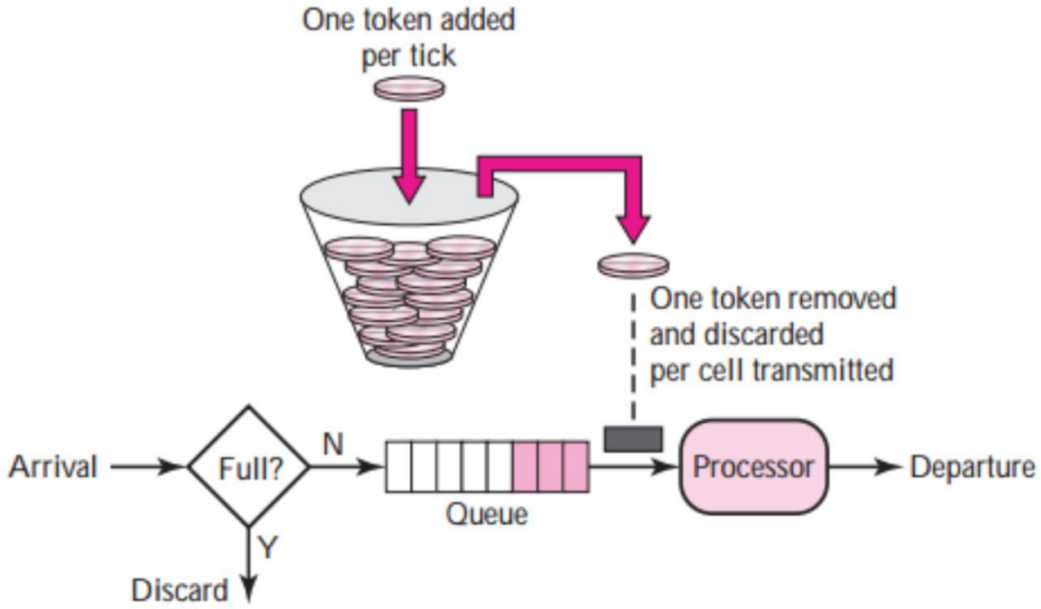
配置项	类型	默认值	描述
listener.tcp.external.max_conn_rate	Number	1000	本节点上允许的最大连接速率（conn/s）
zone.external.rate_limit.conn_messagess_in	Number,Duration	无限制	单连接上允许的最大发布速率（msg/s）
zone.external.rate_limit.conn_bytes_in	Size,Duration	无限制	单连接上允许的最大报文速率（bytes/s）

- **max_conn_rate** 是单个 emqx 节点上连接建立的速度限制。1000 代表秒最多允许 1000 个客户端接入。
- **conn_messages_in** 是单个连接上接收 PUBLISH 报文的速率限制。100,10s 代表每个连接上允许收到的最大 PUBLISH 消息速率是每 10 秒 100 个。
- **conn_bytes_in** 是单个连接上接收 TCP 数据包的速率限制。100KB,10s 代表每个连接上允许收到的最大 TCP 报文速率是每 10 秒 100KB。

conn_messages_in 和 conn_bytes_in 提供的都是针对单个连接的限制，EMQ X 目前没有提供全局的消息速率限制。

4.13 速率限制原理

EMQ X 使用令牌桶 (Token Bucket) (opens new window)算法来对所有的 Rate Limit 来做控制。 令牌桶算法 的逻辑如下图:



- 存在一个可容纳令牌(Token) 的最大值 burst 的桶(Bucket)，最大值 burst 简记为 b 。
- 存在一个 rate 为每秒向桶添加令牌的速率，简记为 r 。当桶满时则不不再向桶中加入入令牌。

- 每当有 1 个(或 N 个)请求抵达时，则从桶中拿出 1 个 (或 N 个) 令牌。如果令牌不不够则阻塞，等待令牌的生生成。

由此可知该算法中:

- 长期来看，所限制的请求速率的平均值等于 rate 的值。
- 记实际请求达到速度为 M，且 M > r，那么，实际运行中能达到的最大(峰值)速率为 M = b + r，证明:

容易想到，最大速率 M 为：能在 1 个单位时间内消耗完满状态令牌桶的速度。而桶中令牌的消耗速度为 M - r，故可知：b / (M - r) = 1，得 M = b + r

4.13.1 令牌桶算法在 EMQ X 中的应用

当使用如下配置做报文速率限制的时候:

EMQ X 将使用两个值初始化每个连接的 rate-limit 处理器:

- rate = 100 KB / 10s = 10240 B/s
- burst = 100 KB = 102400 B

根据 消息速率限制原理 中的算法，可知:

- 长期来看允许的平均速率限制为 10240 B/s
- 允许的峰值速率为 102400 + 10240 = 112640 B/s

为提高系统吞吐，EMQ X 的接入模块不会一条一条的从 socket 读取报文，而是每次从 socket 读取 N 条报文。rate-limit 检查的时机就是在收到这 N 条报文之后，准备继续收取下个 N 条报文之前。故实际的限制速率不会如算法一样精准。EMQ X 只提供了一个大概的速率限制。N 的值可以在 etc/emqx.conf 中配置:

配置项	类型	默认值	描述
listener.tcp.external.active_n	Number	100	emqx 每次从 TCP 栈读取多少条消

			息

4.14 飞行窗口与消息队列

为了提高消息吞吐效率和减少网络波动带来的影响，EMQ X 允许多个未确认的 QoS 1 和 QoS 2 报文同时存在于网路链路上。这些已发送但未确认的报文将被存放在 Inflight Window 中直至完成确认。

当网络链路中同时存在的报文超出限制，即 Inflight Window 到达长度限制（见 max_inflight）时，EMQ X 将不再发送后续的报文，而是将这些报文存储在 Message Queue 中。一旦 Inflight Window 中有报文完成确认，Message Queue 中的报文就会以先入先出的顺序被发送，同时存储到 Inflight Window 中。

当客户端离线时，Message Queue 还会被用来存储 QoS 0 消息，这些消息将在客户端下次上线时被发送。这功能默认开启，当然你也可以手动关闭，见 mqueue_store_qos0。

需要注意的是，如果 Message Queue 也到达了长度限制，后续的报文将依然缓存到 Message Queue，但相应的 Message Queue 中最先缓存的消息将被丢弃。如果队列中存在 QoS 0 消息，那么将优先丢弃 QoS 0 消息。因此，根据你的实际情况配置一个合适的 Message Queue 长度限制（见 max_mqueue_len）是非常重要的。

1）飞行队列与 Receive Maximum

MQTT v5.0 协议为 CONNECT 报文新增了一个 Receive Maximum 的属性，官方对它的解释是：

客户端使用此值限制客户端愿意同时处理的 QoS 为 1 和 QoS 为 2 的发布消息最大数量。没有机制可以限制服务端试图发送的 QoS 为 0 的发布消息 。

也就是说，服务端可以在等待确认时使用不同的报文标识符向客户端发送后续的 PUBLISH 报文，直到未被确认的报文数量到达 Receive Maximum 限制。

不难看出，Receive Maximum 其实与 EMQ X 中的 Inflight Window 机制如出一辙，只是在 MQTT v5.0 协议发布前，EMQ X 就已经对接入的 MQTT 客户端提供了这一功能。现在，使用 MQTT v5.0 协议的客户端将按照 Receive Maximum 的规范来设置 Inflight Window 的最大长度，而更低版本 MQTT 协议的客户端则依然按照配置来设置。

2）配置项

配置项	类型	可取值	默认值	说明
max_inflight	integer	>= 0	32 (external), 128 (internal)	Inflight Window 长度限制，0 即无限制
max_mqueue_len	integer	>= 0	1000 (external), 10000 (internal)	Message Queue 长度限制，0 即无限制
mqueue_store_qos0	enum	`true`, `false`	true	客户端离线时 EMQ X 是否存储 QoS 0 消息至 Message Queue

4.15 消息重传

消息重传 (Message Retransmission) 是属于 MQTT 协议标准规范的一部分。协议中规定了作为通信的双方 服务端 和 客户端 对于自己发送到对端的 PUBLISH 消息都应满足其 服务质量 (Quality of Service levels) 的要求。如：

- QoS 1：表示 消息至少送达一次 (At least once delivery)；即发送端会一直重发该消息，除非收到了对端对该消息的确认。意思是在 MQTT 协议的上层（即业务的应用层）相同的 QoS 1 消息可能会收到多次。
- QoS 2：表示 消息只送达一次 (Exactly once delivery)；即该消息在上层仅会接收到一次。

虽然，QoS 1 和 QoS 2 的 PUBLISH 报文在 MQTT 协议栈这一层都会发生重传，但请你谨记的是：

- QoS 1 消息发生重传后，在 MQTT 协议栈上层，也会收到这些重发的 PUBLISH 消息。
- QoS 2 消息无论如何重传，最终在 MQTT 协议栈上层，都只会收到一条 PUBLISH 消息

（1）基础配置

有两种场景会导致消息重发：

1. PUBLISH 报文发送给对端后，规定时间内未收到应答。则重发这个报文。
2. 在保持会话的情况下，客户端重连后；EMQ X 会自动重发 未应答的消息，以确保 QoS 流程的正确。

在 etc/emqx.conf 中可配置：

配置项	类型	可取值	默认值	说明
retry_interval	duration	–	30s	等待一个超时间隔，如果没收到应答则重传消息

一般来说，你只需要关心以上内容就足够了。

如需了解更多 EMQ X 在处理 MQTT 协议的重传的细节见以下内容。

（2）协议规范与设计

1）重传的对象

首先, 在了解 EMQ X 对于重传机制的设计前, 我们需要先确保你已经了解协议中 QoS 1 和 QoS 2 的传输过程，否则请参见 MQTTv3.1.1 - QoS 1: At least once delivery (opens new window)和 MQTTv3.1.1 - QoS 2: Exactly once delivery (opens new window)

此处，仅作一个简单的回顾，用来说明不同 QoS 下重传的对象有哪些。

2）QoS 1

QoS 1 要求消息至少送达一次；所以消息在 MQTT 协议层中，可能会不断的重传，直到发送端收到了该消息的确认报文。

其流程示意图如下：

```
                PUBLISH
#1 Sender  ---------------->  Receiver       (*)
                PUBACK
#2 Sender  <----------------  Receiver
```

涉及到 2 个报文；共 2 次发送动作，发送端和接收端各 1 次；这 2 个报文都持有相同的 PacketId。

- 行尾标记为 * 号的，表示发送方在等待确认报文超时后，可能会主动发起重传。

可见 QoS 1 消息只需要对 PUBLISH 报文进行重发

3）QoS 2

QoS 2 要求消息只送达一次；所以在实现它时，需要更复杂的流程。其流程示意图如下：

```
                PUBLISH
#1 Sender  ---------------->  Receiver       (*)
                PUBREC
#2 Sender  <----------------  Receiver
                PUBREL
#3 Sender  ---------------->  Receiver       (*)
                PUBCOM
#4 Sender  <----------------  Receiver
```

- 涉及到 4 个报文；共 4 次发送动作，发送端和接收端各 2 次；这 4 个报文都持有相同的 PacketId。
- 行尾标记为 * 号的，表示发送方在等待确认报文超时后，可能会主动发起重传。

可见 QoS 2 消息需要对 PUBLISH 和 PUBREL 报文进行重发

综上：

重传动作 都是由于 发送端 报文发送后，在 规定时间 内未收到其期待的返回而触发的。

重传对象 仅包含以下三种：

- QoS 1 的 PUBLISH 报文
- QoS 2 的 PUBLISH 报文
- QoS 2 的 PUBREL 报文

当 EMQ X 作为 PUBLISH 消息的接收端时，它不需要重发操作

（3）飞行窗口与最大接收值

其概念的定义和解释参见 飞行窗口与消息队列

引入这两个概念的作用是为了理解：

1. EMQ X 作为发送端时，再次重发的消息，必然是已存储在飞行窗口中的消息
2. EMQ X 作为接收端时，发送端重发的消息时：
    - 如 QoS 1，EMQ X 则直接回复 PUBACK 进行应答；

- 如 QoS 2, EMQ X 则会释放, 存储在 最大接收消息 队列中的 PUBLISH 或者 PUBREL 报文。

1) 消息顺序

当然，以上的概念仅需要了解即可，你最需要关心的是，消息在被重复发送后，消息顺序出现的变化，尤其是 QoS 1 类的消息。例如:

假设，当前飞行窗口设置为 2 时，EMQ X 计划向客户端的某主题投递 4 条 QoS 1 的消息。并假设客户端程序、或网络在中间出现过问题，那么整个发送流程会变成:

```
1  [4,3,2,1 || ]    ----->    []
2  [4,3 || 2, 1]    ----->    [1, 2]
3  [4 || 3, 2]      ----->    [1, 2, 3]
4  [4 || 3, 2]      ----->    [1, 2, 3, 2, 3]
5  [ || 4]          ----->    [1, 2, 3, 2, 3, 4]
6  [ || ]           ----->    [1, 2, 3, 2, 3, 4]
```

流程共 6 个步骤；左边表示 EMQ X 的 消息队列 和 飞行窗口，以 || 分割；右侧表示客户端收到的消息顺序，其中每步表示:

1. Broker 将 4 条消息放入消息队列中。
2. Broker 依次发送 1 2，并将其放入 **飞行窗口** 中；客户端仅应答消息 1；且此时由于客户端发送流出现了问题，无法发送后续应答报文。
3. Broker 收到消息 1 的应答；从飞行窗口移除消息 1；并将 3 发送出去；继续等待 2 3 的应答;
4. Broker 等待应答超时，重发了报文 2 3；客户端收到重发的报文 2 3 并正常应答。
5. Broker 从飞行窗口移除了消息 2 3，并发送报文 4；客户端收到了报文 4 并回复应答。
6. 至此，所有报文处理完成。客户端收到的报文顺序为 [1, 2, 3, 2, 3, 4]，并也依次上报给 MQTT 协议栈的上层。

虽然，存在重复的报文消息。但这是完全符合协议的规范的，每个报文第一次出现的位置都是有序的，并且重复收到的报文 2 3 的报文中，会携带一个标识位，表明其为重发报文。MQTT 协议和 EMQ X 将这个主题认为是 有序的主题 (Ordered Topic)

它确保 相同的主题和 QoS 下，消息是按顺序投递和应答的。

此外，如果用户期望所有主题下的 QoS 1 与 QoS 2 消息都严格有序，那么需要设置飞行窗口的最大长度为 1，但代价是会降低该客户端的吞吐。

2) 相关配置

此节列举了上述机制中，用到的所有配置。它们都包含在 etc/emqx.conf 中：

配置项	类型	可取值	默认值	说明
mqueue_store_qos0	bool	`true`, `false`	true	是否将 QoS 0 消息存入消息队列中
max_mqueue_len	integer	>= 0	1000	消息队列长度
max_inflight	integer	>= 0	0	飞行窗口大小；默认 `0` 即无限制
max_awaiting_rel	integer	>= 0	0	最大接收；默认 `0` 即无限制
await_rel_timeout	durtaion	> 0	300s	最大接收 中消息等待释放的最大超时时间；超过则直接丢弃

4.16 告警

EMQ X Broker 内置监控与告警功能，目前支持监控 CPU 占用率、（系统、进程）内存占用率、进程数量、规则引擎资源状态、集群脑裂与愈合并进行告警。激活和取消告警都将产生一条警告日志并由 Broker 发布一个主题为 $SYS/brokers/<Node>/alarms/activate 或 $SYS/brokers/<Node>/alarms/deactivate 的 MQTT 消息，用户可以通过订阅 $SYS/brokers/+/alarms/avtivate 和 $SYS/brokers/+/alarms/deactivate 主题来获取告警通知。

告警通知消息的 Payload 为 Json 格式，包含以下字段：

字段	类型	说明
name	string	告警名称
details	object	告警详情
message	string	人类易读的告警说明
activate_at	integer	激活告警的时间，以微秒为单位的 UNIX 时间戳
deactivate_at	integer / string	取消激活告警的时间，以微秒为单位的 UNIX 时间戳，激活中的告警此字段值为 infinity
activated	boolean	告警是否处于激活状态

以系统内存占用率过高告警为例，您将收到以下格式的消息：

PUBLISH PACKET

Topic Name `$SYS/brokers/emqx@127.0.0.1/alarms/activate`

QoS `0`

Retain `1`

Payload

```
{
    "name":"high_system_memory_usage",
    "message":"System memory usage is higher than 60%",
    "details":{
        "high_watermark":60
    },
    "deactivate_at":"infinity",
    "activated":true,
    "activate_at":1597993108657522
}
```

告警不会重复产生，即如果 CPU 占用率过高告警已经激活，则在其激活期间，不会出现第二个 CPU 占用率过高告警。告警会在被监控项恢复正常后自动取消激活，但同样支持用户手动取消激活（如果用户明确自己不关心该告警）。用户除了可以在 Dashboard 查看当前告警（激活中的告警）与历史告警（已取消激活的告警）以外，还可以通过 EMQ X Broker 提供的 HTTP API 查询和管理告警。

EMQ X Broker 允许用户对告警功能进行一定程度的调整以适应实际需要，目前开放了以下配置项：

配置项	类型	默认值	说明
os_mon.cpu_check_interval	duration	60s	CPU 占用率的检查间隔
os_mon.cpu_high_watermark	percent	80%	CPU 占用率高水位，即 CPU 占用率达到多少时激活告警
os_mon.cpu_low_watermark	percent	60%	CPU 占用率低水位，即 CPU 占用率降低到多少时取消告警
os_mon.mem_check_interval	duration	60%	内存占用率的检查间隔
os_mon.sysmem_high_watermark	percent	70%	系统内存占用率高水位，即申请的总内存占比达到多少时激活告警
os_mon.procmem_high_watermark	percent	5%	进程内存占用率高水位，即单个进程申请的内存占比达到多少时激活告警
vm_mon.check_interval	duration	30s	进程数量的检查间隔
vm_mon.process_high_watermark	percent	80%	进程占用率高水位，即创建的

			进程数量与最大数量限制的占比达到多少时激活告警
vm_mon.process_low_watermark	percent	60%	进程占用率低水位，即创建的进程数量与最大数量限制的占比降低到多少时取消告警
alarm.actions	string	log,publish	告警激活时触发的动作，目前仅支持 log 与 publish，即输出日志与发布系统消息
alarm.size_limit	integer	1000	已取消激活告警的最大保存数量，达到限制后将以 FIFO 原则清理这些告警
alarm.validity_period	duration	24h	已取消激活告警的最大保存时间，过期的告警将被清理

4.17 数据导入导出

EMQ X Broker 为用户提供了数据导入导出功能，以满足服务器升级、迁移以及数据备份等需要。数据导入导出功能支持将当前运行的 EMQ X Broker 中的黑名单、规则引擎配置等存储在 EMQ X Broker 默认数据库 Mnesia 中的数据以 Json 格式导出至本地文件。当然用户无需关心导出文件中的数据内容。导出文件可以导入至其他 EMQ X Broker 的运行实例，EMQ X Broker 可以是相同版本，也可以是不同版本，但目前仅支持 4.1.0 及之后的版本。

EMQ X Broker 为数据导入导出功能提供了命令行接口、HTTP API 以及 Dashboard 的可视化界面（企业版）。目前支持导入导出的数据如下:

- 规则引擎配置数据（资源、规则）
- 规则引擎编解码配置数据（企业版）
- 黑名单数据
- Application 数据
- Dashboard 用户数据
- 通过 emqx-auth-mnesia 插件添加的 MQTT 用户数据和 ACL 数据
- 通过 emqx-auth-clientid 插件添加的 MQTT 用户数据和 ACL 数据
- 通过 emqx-auth-username 插件添加的 MQTT 用户数据和 ACL 数据
- 编解码 Schema

（1）示例

1）命令行接口

1. 导出数据, 导出文件的文件名格式为 emqx-export-YYYY-MM-DD-HH-mm-SS.json, 默认导出路径为 data 目录（请参见 目录结构）

```
$ ./emqx_ctl data export
The emqx data has been successfully exported to
/var/lib/emqx/data/emqx-export-2020-5-15-17-39-0.json.
```

2. 保存导出文件，这里将导出文件保存到 tmp 目录

```
$ cp /var/lib/emqx/data/emqx-export-2020-5-15-17-39-0.json /tmp
```

3. 重新安装 EMQ X Broker 并启动

```
$ ./emqx start
EMQ X Broker v4.1-rc.1 is started successfully!
```

4. 导入数据，导入的文件名必须以绝对路径形式指定

```
$ ./emqx_ctl data import /tmp/emqx-export-2020-5-15-17-39-0.json
The emqx data has been imported successfully.
```

5. 数据重载。有时由于一些版本间的兼容性问题和自定义数据处理的需求，我们允许重载被导入的数据内容。通过指定 --env 参数来指定一个 JSON 格式重载数据或兼容性指令：

```
$ ./emqx_ctl data import /tmp/emqx-export-2020-5-15-17-39-0.json --env
'{"auth.mnesia.as":"username"}'
The emqx data has been imported successfully.
```

2）HTTP API

1. 导出数据

```
$ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/export"
{"data":{"size":388,"filename":"emqx-export-2020-9-4-10-24-16.json","created_at":"2020-9
-4 10:24:16"},"code":0}
```

导出的数据文件位于 .../emqx/data 或 /var/lib/emqx/data 目录

2. 下载数据文件

$ curl --basic -u admin:public -X GET

http://localhost:8081/api/v4/data/file/emqx-export-2020-9-4-10-24-16.json -o

/tmp/emqx-export-2020-9-4-10-24-16.json

3. 上传数据文件

$ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/import" -d @/tmp/emqx-export-2020-9-4-10-24-16.json

{"code":0}

$第 2、3 步适用于在不同机器上迁移 emqx

4. 导入数据

$ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/import" -d '{"filename":"emqx-export-2020-9-4-10-24-16.json"}'

{"code",0}

5. 集群部署

5.1 分布式 Erlang

Erlang/OTP 最初是爱立信为开发电信设备系统设计的编程语言平台，电信设备 (路由器、接入网关...) 典型设计是通过背板连接主控板卡与多块业务板卡的分布式系统。

5.1.1 节点与分布式 Erlang

Erlang/OTP 语言平台的分布式程序，由分布互联的 Erlang 运行时系统组成，每个 Erlang 运行时系统被称为节点(Node)，节点间通过 TCP 两两互联，组成一个网状结构。

Erlang 节点由唯一的节点名称标识，节点名称由 @ 分隔的两部分组成:

<name>@<ip-address>

节点间通过节点名称进行通信寻址。例如在本机启动四个 shell 终端，然后使用 -name 参数分别启动四个 Erlang 节点:

erl -name node1@127.0.0.1 -setcookie my_nodes

erl -name node2@127.0.0.1 -setcookie my_nodes

erl -name node3@127.0.0.1 -setcookie my_nodes

erl -name node4@127.0.0.1 -setcookie my_nodes

使用 node(). 可查看本节点名，使用 nodes(). 可查看已与当前节点建立连接的其他节点。我们现在到 'node1@127.0.0.1' 的控制台下，查看当前节点名和已连接的节点:

```
(node1@127.0.0.1) 4> node().
'node1@127.0.0.1'


(node1@127.0.0.1) 4> nodes().
[]
```

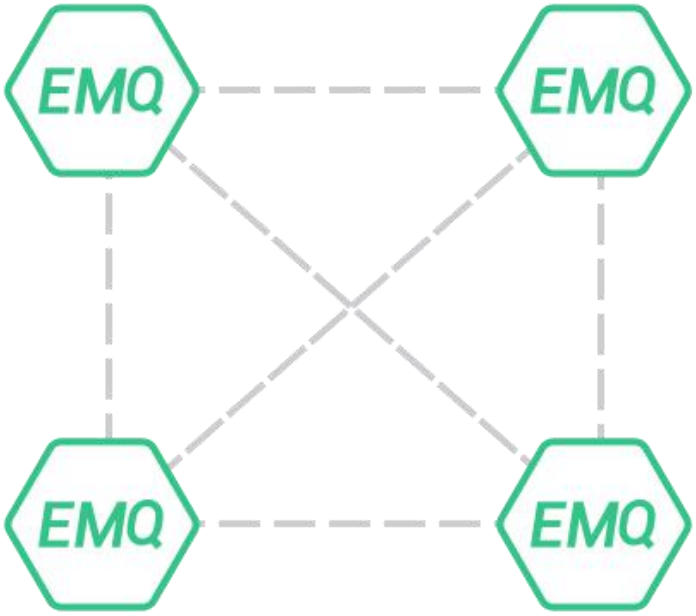
然后我们让 node1 发起与其他节点的连接:

```
(node1@127.0.0.1) 1> net_kernel:connect_node('node2@127.0.0.1').
true
(node1@127.0.0.1) 2> net_kernel:connect_node('node3@127.0.0.1').
true
(node1@127.0.0.1) 3> net_kernel:connect_node('node4@127.0.0.1').
true
```

现在再次可查看已与 node1 建立连接的其他节点:

```
(node1@127.0.0.1) 4> nodes().
['node2@127.0.0.1','node3@127.0.0.1','node4@127.0.0.1']
```

可以看到 node2、node3、node4 都已与 node1 建立了分布式连接，四个节点组成了一个集群。注意每当一个新的节点加入集群时，它会与集群中所有的节点都建立一个 TCP 连接。至此，四个节点完成了如下图所示的网状结构:



5.1.2 安全

Erlang 节点间通过 cookie 进行互连认证。cookie 是一个字符串，只有 cookie 相同的两个节点才能建立连接。上节 中我们曾经使用 -setcookie my_nodes 参数给四个节点设置了相同的 cookie: my_nodes。

详见: http://erlang.org/doc/reference_manual/distributed.html(opens new window)

5.1.3 EMQ X 集群协议设置

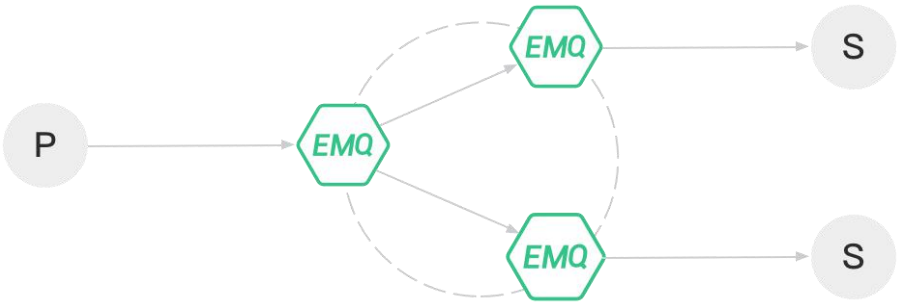
Erlang 集群中各节点可通过 TCPv4、TCPv6 或 TLS 方式连接，可在 etc/emqx.conf 中配置连接方式:

配置名	类型	默认值	描述
cluster.proto_dist	enum	`inet_tcp`	分布式协议，可选值: – inet_tcp: 使用 TCP IPv4 – inet6_tcp: 使用 TCP IPv6 – inet_tls: 使用 TLS
node.ssl_dist_optfile	文件路径	`etc/ssl_dist.conf`	当 `cluster.proto_dist` 选定为 inet_tls 时，需要配

			置 etc/ssl_dist.conf 文件， 指定 TLS 证书等

5.2 EMQ X 分布式集群设计

EMQ X 分布式的基本功能是将消息转发和投递给各节点上的订阅者，如下图所示：



为实现此过程，EMQ X 维护了几个与之相关的数据结构：订阅表，路由表，主题树。

（1）订阅表：主题 - 订阅者

MQTT 客户端订阅主题时，EMQ X 会维护主题(Topic) -> 订阅者(Subscriber) 映射的**订阅表**。订阅表只存在于订阅者所在的 EMQ X 节点上，例如:

```
node1:

    topic1 -> client1, client2
    topic2 -> client3

node2:

    topic1 -> client4
```

（2）路由表：主题 - 节点

而同一集群的所有节点，都会**复制**一份主题(Topic) -> 节点(Node) 映射的**路由表**，例如:

topic1 -> node1, node2

topic2 -> node3

topic3 -> node2, node4

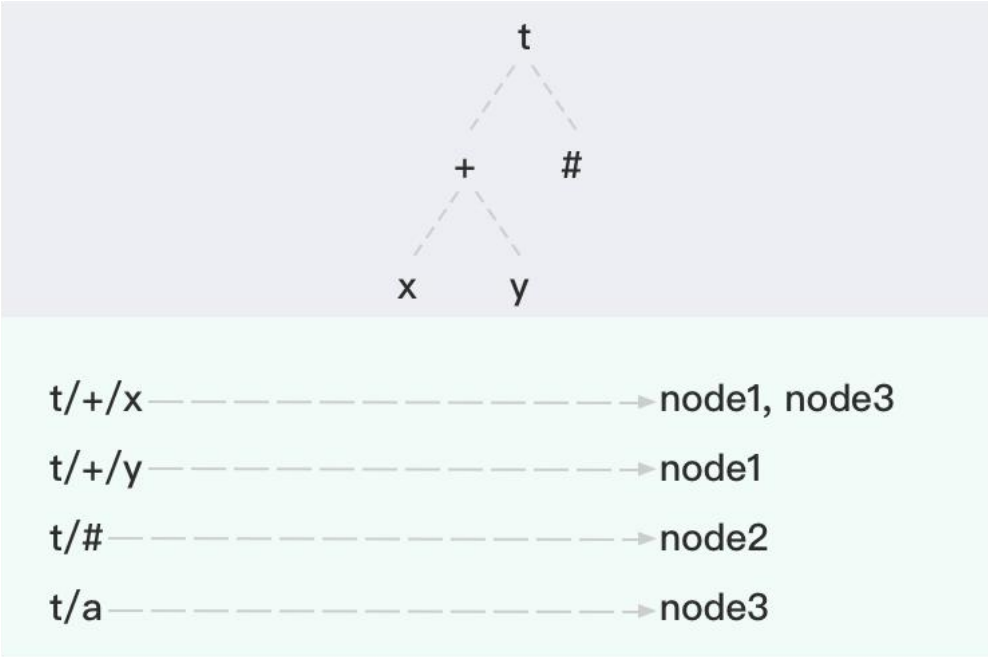
（3）主题树：带统配符的主题匹配

除路由表之外，EMQ X 集群中的每个节点也会维护一份**主题树**(Topic Trie) 的备份。

例如下述主题订阅关系:

客户端	节点	订阅主题
client1	node1	t/+/x, t/+/y
client2	node2	t/
client3	node3	t/+/x, t/a

在所有订阅完成时，EMQ X 中会维护如下主题树（Topic Trie）和路由表(Route Table):



5.2.2 消息派发过程

当 MQTT 客户端发布消息时，所在节点会根据消息主题，检索路由表并转发消息到相关节点，再由相关节点检索本地的订阅表并将消息发送给相关订阅者。

例如 client1 向主题 t/a 发布消息，消息在节点间的路由与派发流程:

1. client1 发布主题为 t/a 的消息到节点 node1
2. node1 通过查询主题树，得知 t/a 可匹配到现有的 t/a、t/ 这两个主题。
3. node1 通过查询路由表，得知主题 t/a 只在 node3 上有订阅者，而主题 t/ 只在 node2 上有订阅者。故 node1 将消息转发给 node2 和 node3。
4. node2 收到转发来的 t/a 消息后，查询本地订阅表，获取本节点上订阅了 t/ 的订阅者，并把消息投递给他们。
5. node3 收到转发来的 t/a 消息后，查询本地订阅表，获取本节点上订阅了 t/a 的订阅者，并把消息投递给他们。
6. 消息转发和投递结束。

5.2.3 数据分片与共享方式

EMQ X 的订阅表在集群中是分片(partitioned)的，而主题树和路由表是共享(replicated)的。

5.3 节点发现与自动集群

EMQ X 支持基于 Ekka 库的集群自动发现 (Autocluster)。Ekka 是为 Erlang/OTP 应用开发的集群管理库，支持 Erlang 节点自动发现 (Service Discovery)、自动集群 (Autocluster)、脑裂自动愈合 (Network Partition Autoheal)、自动删除宕机节点 (Autoclean)。

EMQ X 支持多种节点发现策略:

策略	说明
manual	手动命令创建集群
static	静态节点列表自动集群
mcast	UDP 组播方式自动集群
dns	DNS A 记录自动集群
etcd	通过 etcd 自动集群
k8s	Kubernetes 服务自动集群

（1）manual 手动创建集群

默认配置为手动创建集群，节点须通过 ./bin/emqx_ctl join <Node> 命令加入:

cluster.discovery = manual

（2）基于 static 节点列表自动集群

配置固定的节点列表，自动发现并创建集群:

cluster.discovery = static

cluster.static.seeds = emqx1@127.0.0.1,emqx2@127.0.0.1

（3）基于 mcast 组播自动集群

基于 UDP 组播自动发现并创建集群:

cluster.discovery = mcast

cluster.mcast.addr = 239.192.0.1

cluster.mcast.ports = 4369,4370

cluster.mcast.iface = 0.0.0.0

cluster.mcast.ttl = 255

cluster.mcast.loop = on

（4）基于 DNS A 记录自动集群

基于 DNS A 记录自动发现并创建集群:

cluster.discovery = dns

cluster.dns.name = localhost

cluster.dns.app = ekka

（5）基于 etcd 自动集群

基于 etcd (opens new window)自动发现并创建集群:

cluster.discovery = etcd

cluster.etcd.server = http://127.0.0.1:2379

cluster.etcd.prefix = emqcl

cluster.etcd.node_ttl = 1m

（6）基于 kubernetes 自动集群

Kubernetes (opens new window)下自动发现并创建集群:

cluster.discovery = k8s

cluster.k8s.apiserver = http://10.110.111.204:8080

cluster.k8s.service_name = ekka

cluster.k8s.address_type = ip

cluster.k8s.app_name = ekka

Kubernetes 不建议使用 Fannel 网络插件，推荐使用 Calico 网络插件。

（7）手动(manual) 方式管理集群介绍

假设要在两台服务器 s1.emqx.io, s2.emqx.io 上部署 EMQ X 集群:

节点名	主机名（FQDN）	IP 地址
emqx@s1.emqx.io 或 emqx@192.168.0.10	s1.emqx.io	192.168.0.10
emqx@s2.emqx.io 或 emqx@192.168.0.20	s2.emqx.io	192.168.0.20

注意： 节点名格式为 Name@Host, Host 必须是 IP 地址或 FQDN（主机名。域名）

1）配置 emqx@s1.emqx.io 节点

emqx/etc/emqx.conf:

node.name = emqx@s1.emqx.io

或

node.name = emqx@192.168.0.10

也可通过环境变量添加

export EMQX_NODE_NAME=emqx@s1.emqx.io && ./bin/emqx start

注意: 节点启动加入集群后，节点名称不能变更。

2）配置 emqx@s2.emqx.io 节点

emqx/etc/emqx.conf:

```
node.name = emqx@s2.emqx.io
```

或

```
node.name = emqx@192.168.0.20
```

3）节点加入集群

启动两台节点后，在 s2.emqx.io 上执行:

```
$ ./bin/emqx_ctl cluster join emqx@s1.emqx.io

Join the cluster successfully.

Cluster status: [{running_nodes,['emqx@s1.emqx.io','emqx@s2.emqx.io']}]
```

或者在 s1.emqx.io 上执行:

```
$ ./bin/emqx_ctl cluster join emqx@s2.emqx.io


Join the cluster successfully.

Cluster status: [{running_nodes,['emqx@s1.emqx.io','emqx@s2.emqx.io']}]
```

在任意节点上查询集群状态:

```
$ ./bin/emqx_ctl cluster status

Cluster status: [{running_nodes,['emqx@s1.emqx.io','emqx@s2.emqx.io']}]
```

4）退出集群

节点退出集群，两种方式:

1. leave: 让本节点退出集群
2. force-leave: 从集群删除其他节点

让 emqx@s2.emqx.io 主动退出集群:

```
$ ./bin/emqx_ctl cluster leave
```

或在 s1.emqx.io 上，从集群删除 emqx@s2.emqx.io 节点:

```
$ ./bin/emqx_ctl cluster force-leave emqx@s2.emqx.io
```

5.4 集群脑裂与自动愈合

EMQ X 支持集群脑裂自动恢复(Network Partition Autoheal)，可在 etc/emqx.conf 中配置:

cluster.autoheal = on

集群脑裂自动恢复流程:

1. 节点收到 Mnesia 的 inconsistent_database 事件 3 秒后进行集群脑裂确认;
2. 节点确认集群脑裂发生后，向 Leader 节点 (集群中最早启动节点) 上报脑裂消息;
3. Leader 节点延迟一段时间后，在全部节点在线状态下创建脑裂视图 (SplitView);
4. Leader 节点在多数派 (majority) 分区选择集群自愈的 Coordinator 节点;
5. Coordinator 节点重启少数派 (minority) 分区节点恢复集群。

5.5 集群节点自动清除

EMQ X 支持从集群自动删除宕机节点 (Autoclean)，可在 etc/emqx.conf 中配置:

cluster.autoclean = 5m

5.6 防火墙设置

5.6.1 集群节点发现端口

若预先设置了环境变量 WITH_EPMD=1，启动 emqx 时会使用启动 epmd (监听端口 4369) 做节点发现。称为 epmd 模式。

若环境变量 WITH_EPMD 没有设置，则启动 emqx 时不启用 epmd，而使用 emqx ekka 的节点发现，这也是 4.0 之后的默认节点发现方式。称为 ekka 模式。

(1) epmd 模式:

如果集群节点间存在防火墙，防火墙需要为每个节点开通 TCP 4369 端口，用来让各节点能互相访问。

防火墙还需要开通一个 TCP 从 node.dist_listen_min(包含) 到 node.dist_listen_max(包含) 的端口段， 这两个配置的默认值都是 6369。

(2) ekka 模式 (4.0 版本之后的默认模式):

跟 empd 模式不同，在 ekka 模式下，集群发现端口的映射关系是约定好的，而不是动态的。node.dist_listen_min and node.dist_listen_max 两个配置在 ekka 模式下不起作用。

如果集群节点间存在防火墙，防火墙需要放开这个约定的端口。约定端口的规则如下:

ListeningPort = BasePort + Offset

其中 BasePort 为 4370 (不可配置), Offset 为节点名的数字后缀. 如果节点名没有数字后

缀的话，Offsset 为 0。

举例来说，如果 emqx.conf 里配置了节点名：node.name = emqx@192.168.0.12，那么监听端口为 4370， 但对于 emqx1 (或者 emqx-1) 端口就是 4371，以此类推。

5.7 The Cluster PRC Port

每个节点还需要监听一个 RPC 端口，也需要被防火墙也放开。跟上面说的 ekka 模式下的集群发现端口一样，这个 RPC 端口也是约定式的。

RPC 端口的规则跟 ekka 模式下的集群发现端口类似，只不过 BasePort = 5370。就是说，如果 emqx.conf 里配置了节点名：node.name = emqx@192.168.0.12，那么监听端口为 5370， 但对于 emqx1 (或者 emqx-1) 端口就是 5371，以此类推。

6. HTTP API 接口

EMQ X 提供了 HTTP API 以实现与外部系统的集成，例如查询客户端信息、发布消息和创建规则等。

EMQ X 的 HTTP API 服务默认监听 8081 端口，可通过 etc/plugins/emqx_management.conf 配置文件修改监听端口，或启用 HTTPS 监听。EMQ X 4.0.0 (opens new window)以后的所有 API 调用均以 api/v4 开头。

6.1 接口安全

EMQ X 的 HTTP API 使用 Basic 认证 (opens new window)方式，id 和 password 须分别填写 AppID 和 AppSecret。 默认的 AppID 和 AppSecret 是：admin/public。你可以在 Dashboard 的左侧菜单栏里，选择 "管理" -> "应用" 来修改和添加 AppID/AppSecret。

6.2 响应码

6.2.1 HTTP 状态码 (status codes)

EMQ X 接口在调用成功时总是返回 200 OK，响应内容则以 JSON 格式返回。

可能的状态码如下：

Status Code	Description
200	成功，返回的 JSON 数据将提供更多信息
400	客户端请求无效，例如请求体或参数错误
401	客户端未通过服务端认证，使用无效的身份验证凭据可能会发生

404	找不到请求的路径或者请求的对象不存在
500	服务端处理请求时发生内部错误

6.2.2 返回码 (result codes)

EMQ X 接口的响应消息体为 JSON 格式，其中总是包含返回码 code。

可能的返回码如下：

Return Code	Description
0	成功
101	RPC 错误
102	未知错误
103	用户名或密码错误
104	空用户名或密码
105	用户不存在
106	管理员账户不可删除
107	关键请求参数缺失
108	请求参数错误
109	请求参数不是合法 JSON 格式
110	插件已开启
111	插件已关闭
112	客户端不在线
113	用户已存在
114	旧密码错误
115	不合法的主题

6.3 API Endpoints

6.3.1 GET /api/v4

返回 EMQ X 支持的所有 Endpoints。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array	Endpoints 列表
- data[0].path	String	Endpoint
- data[0].name	String	Endpoint 名
- data[0].method	String	HTTP Method
- data[0].descr	String	描述

6.4 Broker 基本信息

6.4.1 GET /api/v4/brokers/{node}

返回集群下所有节点的基本信息。

Path Parameters:

Name	Type	Required	Description
node	String	False	节点名字，如 "emqx@127.0.0.1。 不指定时返回所有节点的信息

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object/Array of Objects	node 参数存在时返回指定节点信息， 不存在时返回所有节点的信息
data.datetime	String	当前时间，格式为 "YYYY-MM-DD HH:mm:ss"
data.node	String	节点名称
data.node_status	String	节点状态
data.otp_release	String	EMQ X 使用的 Erlang/OTP 版本
data.sysdescr	String	软件描述
data.uptime	String	EMQ X 运行时间，格式为 "H hours, m minutes, s seconds"
data.version	String	EMQ X 版本

6.5 节点

6.5.1 GET /api/v4/nodes/{node}

返回节点的状态。

Path Parameters:

Name	Type	Required	Description
node	String	False	节点名字，如 "emqx@127.0.0.1。 不指定时返回所有节点的信息

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

data	Object/Array of Objects	node 参数存在时返回指定节点信息，不存在时以 Array 形式返回所有节点的信息
data.connections	Integer	当前接入此节点的客户端数量
data.load1	String	1 分钟内的 CPU 平均负载
data.load5	String	5 分钟内的 CPU 平均负载
data.load15	String	15 分钟内的 CPU 平均负载
data.max_fds	Integer	操作系统的最大文件描述符限制
data.memory_total	String	VM 已分配的系统内存
data.memory_used	String	VM 已占用的内存大小
data.node	String	节点名称
data.node_status	String	节点状态
data.otp_release	String	EMQ X 使用的 Erlang/OTP 版本
data.process_available	Integer	可用的进程数量
data.process_used	Integer	已占用的进程数量
data.uptime	String	EMQ X 运行时间
data.version	String	EMQ X 版本

6.6 客户端

6.6.1 GET /api/v4/clients

返回集群下所有客户端的信息，支持分页。

Query String Parameters:

Name	Type	Required	Default	Description
_page	Integer	False	1	页码
_limit	Integer	False	10000	每页显示的数据条数，未指定时由 `emqx-management` 插件的配置项 `max_row_limit` 决定

在 4.1 后，支持多条件和模糊查询，其包含的查询参数有：

Name	Type	Required	Description
clientid	String	False	客户端标识符
username	String	False	客户端用户名
zone	String	False	客户端配置组名称
ip_address	String	False	客户端 IP 地址
conn_state	Enum	False	客户端当前连接状态， 可取值有：`connected`,`idle`,`disconnected`
clean_start	Bool	False	客户端是否使用了全新的会话
proto_name	Enum	False	客户端协议名称， 可取值有：`MQTT`,`CoAP`,`LwM2M`,`MQTT-SN`

proto_ver	Integer	False	客户端协议版本
_like_clientid	String	False	客户端标识符，子串方式模糊查找
_like_username	String	False	客户端用户名，子串方式模糊查找
_gte_created_at	Integer	False	客户端会话创建时间，小于等于查找
_lte_created_at	Integer	False	客户端会话创建时间，大于等于查找
_gte_connected_at	Integer	False	客户端连接创建时间，小于等于查找
_lte_connected_at	Integer	False	客户端连接创建时间，大于等于查找

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有客户端的信息
data[0].node	String	客户端所连接的节点名称
data[0].clientid	String	客户端标识符
data[0].username	String	客户端连接时使用的用户名
data[0].proto_name	String	客户端协议名称
data[0].proto_ver	Integer	客户端使用的协议版本
data[0].ip_address	String	客户端的 IP 地址
data[0].port	Integer	客户端的端口
data[0].is_bridge	Boolean	指示客户端是否通过桥接方式连接
data[0].connected_at	String	客户端连接时间，格式为 "YYYY-MM-DD HH:mm:ss"
data[0].disconnected_at	String	客户端离线时间，格式为 "YYYY-MM-DD HH:mm:ss"， 此字段仅在 `connected` 为 `false` 时有效并被返回
data[0].connected	Boolean	客户端是否处于连接状态
data[0].zone	String	指示客户端使用的配置组
data[0].keepalive	Integer	保持连接时间，单位：秒
data[0].clean_start	Boolean	指示客户端是否使用了全新的会话
data[0].expiry_interval	Integer	会话过期间隔，单位：秒
data[0].created_at	String	会话创建时间，格式为 "YYYY-MM-DD HH:mm:ss"
data[0].subscriptions_cnt	Integer	此客户端已建立的订阅数量
data[0].max_subscriptions	Integer	此客户端允许建立的最大订阅数量
data[0].inflight	Integer	飞行队列当前长度
data[0].max_inflight	Integer	飞行队列最大长度
data[0].mqueue_len	Integer	消息队列当前长度
data[0].max_mqueue	Integer	消息队列最大长度
data[0].mqueue_dropped	Integer	消息队列因超出长度而丢弃的消息数量
data[0].awaiting_rel	Integer	未确认的 PUBREC 报文数量

data[0].max_awaiting_rel	Integer	允许存在未确认的 PUBREC 报文的最大数量
data[0].recv_oct	Integer	EMQ X Broker（下同）接收的字节数量
data[0].recv_cnt	Integer	接收的 TCP 报文数量
data[0].recv_pkt	Integer	接收的 MQTT 报文数量
data[0].recv_msg	Integer	接收的 PUBLISH 报文数量
data[0].send_oct	Integer	发送的字节数量
data[0].send_cnt	Integer	发送的 TCP 报文数量
data[0].send_pkt	Integer	发送的 MQTT 报文数量
data[0].send_msg	Integer	发送的 PUBLISH 报文数量
data[0].mailbox_len	Integer	进程邮箱大小
data[0].heap_size	Integer	进程堆栈大小，单位：字节
data[0].reductions	Integer	Erlang reduction
meta	Object	分页信息
meta.page	Integer	页码
meta.limit	Integer	每页显示的数据条数
meta.count	Integer	数据总条数

6.6.2 GET /api/v4/clients/{clientid}

返回指定客户端的信息

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	客户端的信息，详细请参见 GET /api/v4/clients

6.6.3 DELETE /api/v4/clients/{clientid}

踢除指定客户端。注意踢除客户端操作会将连接与会话一并终结。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.6.4 GET /api/v4/nodes/{node}/clients

类似 GET /api/v4/clients，返回指定节点下所有客户端的信息，支持分页。

Query String Parameters:

Name	Type	Required	Default	Description
_page	Integer	False	1	页码
_limit	Integer	False	10000	每页显示的数据条数，未指定时由 `emqx-management` 插件的配置项 `max_row_limit` 决定

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有客户端的信息，详情请参看 GET /api/v4/clients

6.6.5 GET /api/v4/nodes/{node}/clients/{clientid}

类似 GET /api/v4/clients/{clientid}，返回指定节点下指定客户端的信息。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	客户端的信息，详细请参见 GET /api/v4/clients

6.6.6 DELETE /api/v4/nodes/{node}/clients/{clientid}

类似 DELETE /api/v4/clients/{clientid}，踢除指定节点下的指定客户端。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.6.7 GET /api/v4/clients/username/{username}

通过 Username 查询客户端的信息。由于可能存在多个客户端使用相同的用户名的情况，所以可能同时返回多个客户端信息。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	客户端的信息，详细请参见 GET /api/v4/clients

6.6.8 GET /api/v4/nodes/{node}/clients/username/{username}

类似 GET /api/v4/clients/username/{username}，在指定节点下，通过 Username 查询指定客户端的信息。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	客户端的信息，详细请参见 GET /api/v4/clients

6.6.9 GET /api/v4/clients/{clientid}/acl_cache

查询指定客户端的 ACL 缓存。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	ACL 详情
data[0].access	String	发布/订阅
data[0].topic	String	MQTT 主题
data[0].result	String	允许/拒绝
data[0].updated_time	Integer	ACL 缓存建立时间

6.6.10 DELETE /api/v4/clients/{clientid}/acl_cache

清除指定客户端的 ACL 缓存。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.7 订阅信息

6.7.1 GET /api/v4/subscriptions

返回集群下所有订阅信息，支持分页机制。

Query String Parameters:

Name	Type	Required	Default	Description
_page	Integer	False	1	页码
_limit	Integer	False	10000	每页显示的数据条数，未指定时由 `emqx-management` 插件的配置项 `max_row_limit` 决定

在 4.1 版本后，支持多条件和模糊查询：

Name	Type	Description
clientid	String	客户端标识符
topic	String	主题，全等查询
qos	Enum	可取值为：`0`,`1`,`2`
share	String	共享订阅的组名称
_match_topic	String	主题，匹配查询

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有订阅信息
data[0].node	String	节点名称
data[0].clientid	String	客户端标识符
data[0].topic	String	订阅主题
data[0].qos	Integer	QoS 等级
meta	Object	同 `/api/v4/clients`

6.7.2 GET /api/v4/subscriptions/{clientid}

返回集群下指定客户端的订阅信息。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	所有订阅信息
data.node	String	节点名称
data.clientid	String	客户端标识符
data.topic	String	订阅主题
data.qos	Integer	QoS 等级

6.7.3 GET /api/v4/nodes/{node}/subscriptions

类似 GET /api/v4/subscriptions，返回指定节点下的所有订阅信息，支持分页机制。

Query String Parameters:

Name	Type	Required	Default	Description

_page	Integer	False	1	页码
_limit	Integer	False	10000	每页显示的数据条数，未指定时由 `emqx-management` 插件的配置项 `max_row_limit` 决定

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有订阅信息
data[0].node	String	节点名称
data[0].clientid	String	客户端标识符
data[0].topic	String	订阅主题
data[0].qos	Integer	QoS 等级
meta	Object	同 `/api/v4/clients`

6.7.4 GET /api/v4/nodes/{node}/subscriptions/{clientid}

类似 GET /api/v4/subscriptions/{clientid}，在指定节点下，查询某 clientid 的所有订阅信息，支持分页机制。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	所有订阅信息
data.node	String	节点名称
data.clientid	String	客户端标识符
data.topic	String	订阅主题
data.qos	Integer	QoS 等级

6.8 路由

6.8.1 GET /api/v4/routes

返回集群下的所有路由信息，支持分页机制。

Query String Parameters:

Name	Type	Required	Default	Description
_page	Integer	False	1	页码
_limit	Integer	False	10000	每页显示的数据条数，未指定时由 `emqx-management` 插件的配置项 `max_row_limit` 决定

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有路由信息
data[0].topic	String	MQTT 主题
data[0].node	String	节点名称
meta	Object	同 `/api/v4/clients`

6.8.2 GET /api/v4/routes/{topic}

返回集群下指定主题的路由信息。

Path Parameters:

Name	Type	Required	Description
topic	Integer	True	主题

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	所有路由信息
data.topic	String	MQTT 主题
data.node	String	节点名称

6.9 消息发布

6.9.1 POST /api/v4/mqtt/publish

发布 MQTT 消息。

Parameters (json):

Name	Type	Required	Default	Description
topic	String	Optional		主题，与 `topics` 至少指定其中之一
topics	String	Optional		以 `,` 分割的多个主题，使用此字段能够同时发布消息到多个主题

clientid	String	Required		客户端标识符
payload	String	Required		消息正文
encoding	String	Optional	plain	消息正文使用的编码方式，目前仅支持 `plain` 与 `base64` 两种
qos	Integer	Optional	0	QoS 等级
retain	Boolean	Optional	false	是否为保留消息

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.10 主题订阅

6.10.1 POST /api/v4/mqtt/subscribe

订阅 MQTT 主题。

Parameters (json):

Name	Type	Required	Default	Description
topic	String	Optional		主题，与 `topics` 至少指定其中之一
topics	String	Optional		以 `,` 分割的多个主题，使用此字段能够同时订阅多个主题
clientid	String	Required		客户端标识符
qos	Integer	Optional	0	QoS 等级

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.10.2 POST /api/v4/mqtt/unsubscribe

取消订阅。

Parameters (json):

Name	Type	Required	Default	Description
topic	String	Required		主题
clientid	String	Required		客户端标识符

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.11 消息批量发布

6.11.1 POST /api/v4/mqtt/publish_batch

批量发布 MQTT 消息。

Parameters (json):

Name	Type	Required	Default	Description
[0].topic	String	Optional		主题, 与 `topics` 至少指定其中之一
[0].topics	String	Optional		以 `,` 分割的多个主题, 使用此字段能够同时发布消息到多个主题
[0].clientid	String	Required		客户端标识符
[0].payload	String	Required		消息正文
[0].encoding	String	Optional	plain	消息正文使用的编码方式, 目前仅支持 `plain` 与 `base64` 两种
[0].qos	Integer	Optional	0	QoS 等级
[0].retain	Boolean	Optional	false	是否为保留消息

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.12 主题批量订阅

6.12.1 POST /api/v4/mqtt/subscribe_batch

批量订阅 MQTT 主题。

Parameters (json):

Name	Type	Required	Default	Description
[0].topic	String	Optional		主题, 与 `topics` 至少指定其中之一
[0].topics	String	Optional		以 `,` 分割的多个主题, 使用此字段能够同时订阅多个主题
[0].clientid	String	Required		客户端标识符
[0].qos	Integer	Optional	0	QoS 等级

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.12.2 POST /api/v4/mqtt/unsubscribe_batch

批量取消订阅。

Parameters (json):

Name	Type	Required	Default	Description
[0].topic	String	Required		主题
[0].clientid	String	Required		客户端标识符

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.13 插件

6.13.1 GET /api/v4/plugins

返回集群下的所有插件信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有路由信息
data[0].node	String	节点名称
data[0].plugins	Array	插件信息，由对象组成的数组，见下文
data[0].plugins.name	String	插件名称
data[0].plugins.version	String	插件版本
data[0].plugins.description	String	插件描述
data[0].plugins.active	Boolean	插件是否启动
data[0].plugins.type	String	插件类型，目前有 `auth`、`bridge`、`feature`、`protocol` 四种类型

6.13.2 GET /api/v4/nodes/{node}/plugins

类似 GET /api/v4/plugins，返回指定节点下的插件信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有路由信息
data[0].name	String	插件名称
data[0].version	String	插件版本
data[0].description	String	插件描述
data[0].active	Boolean	插件是否启动
data[0].type	String	插件类型，目前有 `auth`、`bridge`、`feature`、`protocol` 四种类型

6.13.3 PUT /api/v4/nodes/{node}/plugins/{plugin}/load

加载指定节点下的指定插件。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.13.4 PUT /api/v4/nodes/{node}/plugins/{plugin}/unload

卸载指定节点下的指定插件。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.13.5 PUT /api/v4/nodes/{node}/plugins/{plugin}/reload

重新加载指定节点下的指定插件。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description

code	Integer	0

6.14 监听器

6.14.1 GET /api/v4/listeners

返回集群下的所有监听器信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点的监听器列表
data[0].node	String	节点名称
data[0].listeners	Array of Objects	监听器列表
data[0].listeners[0].acceptors	Integer	Acceptor 进程数量
data[0].listeners[0].listen_on	String	监听端口
data[0].listeners[0].protocol	String	插件描述
data[0].listeners[0].current_conns	Integer	插件是否启动
data[0].listeners[0].max_conns	Integer	允许建立的最大连接数量
data[0].listeners[0].shutdown_count	Array of Objects	连接关闭原因及计数

常见 shutdown_count

Name	Type	Description
normal	Integer	正常关闭的连接数量，仅在计数大于 0 时返回
kicked	Integer	被手动踢除的连接数量，仅在计数大于 0 时返回
discarded	Integer	由于 `Clean Session` 或 `Clean Start` 为 `true` 而被丢弃的连接数量
takeovered	Integer	由于 `Clean Session` 或 `Clean Start` 为 `false` 而被接管的连接数量

6.14.2 GET /api/v4/nodes/{node}/listeners

类似 GET /api/v4/listeners，返回指定节点的监听器信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

data	Array of Objects	各节点的监听器列表
data[0].acceptors	Integer	Acceptor 进程数量
data[0].listen_on	String	监听端口
data[0].protocol	String	插件描述
data[0].current_conns	Integer	插件是否启动
data[0].max_conns	Integer	允许建立的最大连接数量
data[0].shutdown_count	Array of Objects	连接关闭原因及计数

6.15 内置模块

6.15.1 GET /api/v4/modules

返回集群下所有内置模块信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点上的内置模块列表
data[0].node	String	节点名称
data[0].modules	Object	内置模块信息列表，详见下面的 modules:

modules:

Name	Type	Description
name	String	模块名
description	String	模块功能描述
active	Boolean	是否处于活跃状态（是否正在运行）

6.15.2 GET /api/v4/nodes/{node}/modules

类似 GET /api/v4/modules，返回指定节点下所有内置模块信息。

6.15.3 PUT /api/v4/modules/{module}/load

加载集群下所有节点的指定内置模块。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description

code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.15.4 PUT /api/v4/nodes/{node}/modules/{module}/load

类似 PUT /api/v4/modules/{module}/load，加载指定节点下的指定内置模块。

6.15.5 PUT /api/v4/modules/{module}/unload

卸载集群下所有节点的指定内置模块。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.15.6 PUT /api/v4/nodes/{node}/modules/{module}/unload

类似 PUT /api/v4/modules/{module}/unload，卸载指定节点下的指定内置模块。

6.15.7 PUT /api/v4/modules/{module}/reload

重新加载集群下所有节点的指定内置模块，仅为 `emqx_mod_acl_internal` 提供此功能。

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

PUT /api/v4/nodes/{node}/modules/{module}/reload

类似 PUT /api/v4/modules/{module}/reload，重新加载指定节点下的指定内置模块，仅为 `emqx_mod_acl_internal` 提供此功能。

6.16 统计指标

6.16.1 GET /api/v4/metrics

返回集群下所有统计指标数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点上的统计指标列表
data[0].node	String	节点名称
data[0].metrics	Object	监控指标数据，详见下面的 metrics:

metrics:

Name	Type	Description
actions.failure	Integer	规则引擎 action 成功失败次数
actions.success	Integer	规则引擎 action 执行失败次数
bytes.received	Integer	EMQ X 接收的字节数
bytes.sent	Integer	EMQ X 在此连接上发送的字节数
client.authenticate	Integer	客户端认证次数
client.auth.anonymous	Integer	匿名登录的客户端数量
client.connect	Integer	客户端连接次数
client.connack	Integer	发送 CONNACK 报文的次数
client.connected	Integer	客户端成功连接次数
client.disconnected	Integer	客户端断开连接次数
client.check_acl	Integer	ACL 规则检查次数
client.subscribe	Integer	客户端订阅次数

client.unsubscribe	Integer	客户端取消订阅次数
delivery.dropped.too_large	Integer	发送时由于长度超过限制而被丢弃的消息数量
delivery.dropped.queue_full	Integer	发送时由于消息队列满而被丢弃的 QoS 不为 0 的消息数量
delivery.dropped.qos0_msg	Integer	发送时由于消息队列满而被丢弃的 QoS 为 0 的消息数量
delivery.dropped.expired	Integer	发送时由于消息过期而被丢弃的消息数量
delivery.dropped.no_local	Integer	发送时由于 `No Local` 订阅选项而被丢弃的消息数量
delivery.dropped	Integer	发送时丢弃的消息总数
messages.delayed	Integer	EMQ X 存储的延迟发布的消息数量
messages.delivered	Integer	EMQ X 内部转发到订阅进程的消息数量
messages.dropped	Integer	EMQ X 内部转发到订阅进程前丢弃的消息总数
messages.dropped.expired	Integer	接收时由于消息过期而被丢弃的消息数量
messages.dropped.no_subscribers	Integer	由于没有订阅者而被丢弃的消息数量
messages.forward	Integer	向其他节点转发的消息数量
messages.publish	Integer	除系统消息外发布的消息数量
messages.qos0.received	Integer	接收来自客户端的 QoS 0 消息数量
messages.qos1.received	Integer	接收来自客户端的 QoS 1 消息数量
messages.qos2.received	Integer	接收来自客户端的 QoS 2 消息数量
messages.qos0.sent	Integer	发送给客户端的 QoS 0 消息数量
messages.qos1.sent	Integer	发送给客户端的 QoS 1 消息数量
messages.qos2.sent	Integer	发送给客户端的 QoS 2 消息数量
messages.received	Integer	接收来自客户端的消息数量，等于 `messages.qos0.received`，`messages.qos1.received` 与 `messages.qos2.received` 之和

messages.sent	Integer	发送给客户端的消息数量，等于 `messages.qos0.sent`，`messages.qos1.sent` 与 `messages.qos2.sent` 之和
messages.retained	Integer	EMQ X 存储的保留消息数量
messages.acked	Integer	接收的 PUBACK 和 PUBREC 报文数量
packets.received	Integer	接收的报文数量
packets.sent	Integer	发送的报文数量
packets.connect.received	Integer	接收的 CONNECT 报文数量
packets.connack.auth_error	Integer	接收的认证失败的 CONNECT 报文数量
packets.connack.error	Integer	接收的未成功连接的 CONNECT 报文数量
packets.connack.sent	Integer	发送的 CONNACK 报文数量
packets.publish.received	Integer	接收的 PUBLISH 报文数量
packets.publish.sent	Integer	发送的 PUBLISH 报文数量
packets.publish.inuse	Integer	接收的报文标识符已被占用的 PUBLISH 报文数量
packets.publish.auth_error	Integer	接收的未通过 ACL 检查的 PUBLISH 报文数量
packets.publish.error	Integer	接收的无法被发布的 PUBLISH 报文数量
packets.publish.dropped	Integer	超出接收限制而被丢弃的消息数量
packets.puback.received	Integer	接收的 PUBACK 报文数量
packets.puback.sent	Integer	发送的 PUBACK 报文数量
packets.puback.inuse	Integer	接收的报文标识符已被占用的 PUBACK 报文数量
packets.puback.missed	Integer	接收的未知报文标识符 PUBACK 报文数量
packets.pubrec.received	Integer	接收的 PUBREC 报文数量
packets.pubrec.sent	Integer	发送的 PUBREC 报文数量
packets.pubrec.inuse	Integ	接收的报文标识符已被占用的 PUBREC 报文数量

	er	
packets.pubrec.missed	Integer	接收的未知报文标识符 PUBREC 报文数量
packets.pubrel.received	Integer	接收的 PUBREL 报文数量
packets.pubrel.sent	Integer	发送的 PUBREL 报文数量
packets.pubrel.missed	Integer	接收的未知报文标识符 PUBREL 报文数量
packets.pubcomp.received	Integer	接收的 PUBCOMP 报文数量
packets.pubcomp.sent	Integer	发送的 PUBCOMP 报文数量
packets.pubcomp.inuse	Integer	接收的报文标识符已被占用的 PUBCOMP 报文数量
packets.pubcomp.missed	Integer	发送的 PUBCOMP 报文数量
packets.subscribe.received	Integer	接收的 SUBSCRIBE 报文数量
packets.subscribe.error	Integer	接收的订阅失败的 SUBSCRIBE 报文数量
packets.subscribe.auth_error	Integer	接收的未通过 ACL 检查的 SUBACK 报文数量
packets.suback.sent	Integer	发送的 SUBACK 报文数量
packets.unsubscribe.received	Integer	接收的 UNSUBSCRIBE 报文数量
packets.unsubscribe.error	Integer	接收的取消订阅失败的 UNSUBSCRIBE 报文数量
packets.unsuback.sent	Integer	发送的 UNSUBACK 报文数量
packets.pingreq.received	Integer	接收的 PINGREQ 报文数量
packets.pingresp.sent	Integer	发送的 PUBRESP 报文数量
packets.disconnect.received	Integer	接收的 DISCONNECT 报文数量
packets.disconnect.sent	Integer	发送的 DISCONNECT 报文数量
packets.auth.received	Integer	接收的 AUTH 报文数量
packets.auth.sent	Integer	发送的 AUTH 报文数量

rules.matched	Integer	规则的匹配次数
session.created	Integer	创建的会话数量
session.discarded	Integer	由于 `Clean Session` 或 `Clean Start` 为 `true` 而被丢弃的会话数量
session.resumed	Integer	由于 `Clean Session` 或 `Clean Start` 为 `false` 而恢复的会话数量
session.takeovered	Integer	由于 `Clean Session` 或 `Clean Start` 为 `false` 而被接管的会话数量
session.terminated	Integer	终结的会话数量

6.16.2 GET /api/v4/nodes/{node}/metrics

类似 GET /api/v4/metrics，返回指定节点下所有监控指标数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	各节点上的统计指标列表，详见 GET /api/v4/metrics

6.17 主题统计指标

6.17.1 GET /api/v4/topic-metrics

返回所有主题统计指标数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点上的统计指标列表
data[0].topic	String	主题名
data[0].metrics	Object	主题统计指标数据，详见下面的 metrics:

metrics:

Name	Type	Description

messages.qos2.out.rate	Integer	QoS 2 消息 5 秒内平均发送速率
messages.qos2.out.count	Integer	QoS 2 消息发送数量统计
messages.qos2.in.rate	Integer	QoS 2 消息 5 秒内平均接收速率
messages.qos2.in.count	Integer	QoS 2 消息接收数量统计
messages.qos1.out.rate	Integer	QoS 1 消息 5 秒内平均发送速率
messages.qos1.out.count	Integer	QoS 1 消息发送数量统计
messages.qos1.in.rate	Integer	QoS 1 消息 5 秒内平均接收速率
messages.qos1.in.count	Integer	QoS 1 消息接收数量统计
messages.qos0.out.rate	Integer	QoS 0 消息 5 秒内平均发送速率
messages.qos0.out.count	Integer	QoS 0 消息发送数量统计
messages.qos0.in.rate	Integer	QoS 0 消息 5 秒内平均接收速率
messages.qos0.in.count	Integer	QoS 0 消息接收数量统计
messages.out.rate	Integer	MQTT 消息 5 秒内平均发送速率
messages.out.count	Integer	MQTT 消息发送数量统计
messages.in.rate	Integer	MQTT 消息 5 秒内平均接收速率
messages.in.count	Integer	MQTT 消息接收数量统计
messages.dropped.rate	Integer	MQTT 消息 5 秒内平均丢弃速率
messages.dropped.count	Integer	MQTT 消息丢弃数量统计

6.17.2 GET /api/v4/topic-metrics/{topic}

返回指定主题的统计指标数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	主题统计指标数据，详见下面的 data:

data:

Name	Type	Description
messages.qos2.out.rate	Integer	QoS 2 消息 5 秒内平均发送速率
messages.qos2.out.count	Integer	QoS 2 消息发送数量统计
messages.qos2.in.rate	Integer	QoS 2 消息 5 秒内平均接收速率
messages.qos2.in.count	Integer	QoS 2 消息接收数量统计
messages.qos1.out.rate	Integer	QoS 1 消息 5 秒内平均发送速率
messages.qos1.out.count	Integer	QoS 1 消息发送数量统计
messages.qos1.in.rate	Integer	QoS 1 消息 5 秒内平均接收速率
messages.qos1.in.count	Integer	QoS 1 消息接收数量统计
messages.qos0.out.rate	Integer	QoS 0 消息 5 秒内平均发送速率
messages.qos0.out.count	Integer	QoS 0 消息发送数量统计

messages.qos0.in.rate	Integer	QoS 0 消息 5 秒内平均接收速率
messages.qos0.in.count	Integer	QoS 0 消息接收数量统计
messages.out.rate	Integer	MQTT 消息 5 秒内平均发送速率
messages.out.count	Integer	MQTT 消息发送数量统计
messages.in.rate	Integer	MQTT 消息 5 秒内平均接收速率
messages.in.count	Integer	MQTT 消息接收数量统计
messages.dropped.rate	Integer	MQTT 消息 5 秒内平均丢弃速率
messages.dropped.count	Integer	MQTT 消息丢弃数量统计

6.17.3 POST /api/v4/topic-metrics

开启对指定主题的指标统计。

Parameters (json):

Name	Type	Required	Default	Description
topic	String	Required		MQTT 主题名

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.17.4 DELETE /api/v4/topic-metrics/{topic}

关闭对指定主题的指标统计。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.17.5 DELETE /api/v4/topic-metrics

关闭所有主题的指标统计。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.18 状态

6.18.1 GET /api/v4/stats

返回集群下所有状态数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点上的状态数据列表
data[0].node	String	节点名称
data[0].stats	Array	状态数据，详见下面的 stats

stats:

Name	Type	Description
connections.count	Integer	当前连接数量
connections.max	Integer	连接数量的历史最大值
channels.count	Integer	即 `sessions.count`
channels.max	Integer	即 `session.max`
sessions.count	Integer	当前会话数量
sessions.max	Integer	会话数量的历史最大值
topics.count	Integer	当前主题数量
topics.max	Integer	主题数量的历史最大值
suboptions.count	Integer	即 `subscriptions.count`
suboptions.max	Integer	即 `subscriptions.max`
subscribers.count	Integer	当前订阅者数量
subscribers.max	Integer	订阅者数量的历史最大值
subscriptions.count	Integer	当前订阅数量，包含共享订阅
subscriptions.max	Integer	订阅数量的历史最大值
subscriptions.shared.count	Integer	当前共享订阅数量
subscriptions.shared.max	Integer	共享订阅数量的历史最大值
routes.count	Integer	当前路由数量
routes.max	Integer	路由数量的历史最大值
retained.count	Integer	当前保留消息数量
retained.max	Integer	保留消息的历史最大值

6.18.2 GET /api/v4/nodes/{node}/stats

类似 GET /api/v4/stats，返回指定节点上的状态数据。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.19 告警

6.19.1 GET /api/v4/alarms

返回集群下当前告警信息。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	各节点上的告警列表
data[0].node	String	节点名称
data[0].alarms	Array of Objects	当前告警列表
data[0].alarms[0].name	String	告警名称
data[0].alarms[0].message	String	人类易读的告警信息
data[0].alarms[0].details	Object	告警详情
data[0].alarms[0].activate_at	Integer	告警激活时间，以微秒为单位的 UNIX 时间戳
data[0].alarms[0].deactivate_at	Integer	告警取消激活时间，以微秒为单位的 UNIX 时间戳
data[0].alarms[0].activated	Boolean	是否激活

6.19.2 GET /api/v4/nodes/{node}/alarms

返回指定节点下的告警信息。接口参数和返回请参看 GET /api/v4/alarms。

6.19.3 GET /api/v4/alarms/activated

返回集群下激活中的告警。接口参数和返回请参看 GET /api/v4/alarms。

6.19.4 GET /api/v4/nodes/{node}/alarms/activated

返回指定节点下激活中的告警。接口参数和返回请参看 GET /api/v4/alarms。

6.19.5 GET /api/v4/alarms/deactivated

返回集群下已经取消的告警。接口参数和返回请参看 GET /api/v4/alarms/activated。

6.19.6 GET /api/v4/nodes/{node}/alarms/deactivated

返回指定节点下已经取消的告警。接口参数和返回请参看 GET /api/v4/alarms/activated。

6.19.7 POST /api/v4/alarms/deactivated

取消指定告警。

Parameters (json):

Name	Type	Required	Default	Description
node	String	Required		告警所在节点
name	String	Required		告警名称

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.19.8 DELETE /api/v4/alarms/deactivated

清除所有已经取消的告警。

Parameters (json): 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.19.9 DELETE /api/v4/nodes/{node}/alarms/deactivated

清除指定节点下所有已经取消的告警。

Parameters (json): 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.20 ACL 缓存

6.20.1 DELETE /api/v4/acl-cache

清除集群中所有的 ACL 缓存

Query String Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

(1) DELETE /api/v4/node/{node}/acl-cache

清除指定节点的 ACL 缓存

Query String Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.21 黑名单

6.21.1 GET /api/v4/banned

获取黑名单

Query String Parameters:

同 `/api/v4/clients`。

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array	由对象构成的数组，对象中的字段与 POST 方法中的 Request Body 相同
meta	Object	同 `/api/v4/clients`

6.21.2 POST /api/v4/banned

将对象添加至黑名单

Parameters (json):

Name	Type	Required	Default	Description
who	String	Required		添加至黑名单的对象，可以是客户端标识符、用户名和 IP 地址
as	String	Required		用于区分黑名单对象类型，可以是 `clientid`，`username`，`peerhost`
by	String	Optional	user	指示该对象被谁添加至黑名单
at	Integer	Optional	当前系统时间	添加至黑名单的时间，单位：秒
until	Integer	Optional	当前系统时间 + 5 分钟	何时从黑名单中解除，单位：秒

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	与传入的 Request Body 相同

6.21.3 DELETE /api/v4/banned/{as}/{who}

将对象从黑名单中删除

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.22 数据导入导出

数据导入导出。

6.22.1 GET /api/v4/data/export

获取当前的导出文件信息列表，包括文件名、大小和创建时间。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array of Objects	所有路由信息
data[0].filename	String	文件名
data[0].created_at	String	"YYYY-MM-DD HH-mm-SS" 格式的文件创建时间
data[0].size	String	文件大小，单位：字节

6.22.2 POST /api/v4/data/export

导出当前数据到文件。

Path Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	文件信息
data.filename	String	文件名
data.created_at	String	"YYYY-MM-DD HH-mm-SS" 格式的文件创建时间
data.size	String	文件大小，单位：字节

6.22.3 POST /api/v4/data/import

从指定文件导入数据。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Default	Description
filename	String	Required		导入的文件名

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.22.4 GET /api/v4/data/file/{filename}

下载数据文件。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Default	Description
filename	String	Required		导入的文件名

Success Response Body (JSON):

Name	Type	Description
filename	String	文件名
file	String	文件内容

6.22.5 POST /api/v4/data/file

上传数据文件。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Default	Description
filename	String	Required		文件名
file	String	Required		文件内容

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.22.6 DELETE /api/v4/data/file/{filename}

远程删除数据文件。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Default	Description
filename	String	Required		文件名

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
message	String	仅在发生错误时返回，用于提供更详细的错误信息

6.23 规则

查询规则引擎的动作

6.23.1 GET /api/v4/rules/{rule_id}

获取某个规则的详情，包括规则的 SQL、Topics 列表、动作列表等。还会返回当前规则和动作的统计指标的值。

Path Parameters:

Name	Type	Required	Description
rule_id	String	False	可选，Rule ID。如不指定 rule_id 则以数组形式返回所有已创建的规则

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.id	String	Rule ID
- data.rawsql	String	SQL 语句，与请求中的 rawsql 一致
- data.for	String	Topic 列表，表示哪些 topic 可以匹配到此规则
- data.metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics
- data.description	String	规则的描述信息，与请求中的 description 一致
- data.actions	Array	动作列表

- data.actions[0].id	String	Action ID
- data.actions[0].params	Object	动作参数，与请求中的 actions.params 一致
- data.actions[0].name	String	动作名字，与请求中的 actions.name 一致
- data.actions[0].metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics

6.23.2 POST /api/v4/rules

创建规则，返回规则 ID。

Parameters (json):

Name	Type	Required	Description
rawsql	String	True	规则的 SQL 语句
actions	Array	True	动作列表
- actions[0].name	String	True	动作名称
- actions[0].params	Object	True	动作参数。参数以 key-value 形式表示。详情可参看添加规则的示例
description	String	False	可选，规则描述

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	创建成功的规则对象，包含 Rule ID
- data.id	String	Rule ID
- data.rawsql	String	SQL 语句，与请求中的 rawsql 一致
- data.for	String	Topic 列表，表示哪些 topic 可以匹配到此规则
- data.metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics
- data.description	String	规则的描述信息，与请求中的 description 一致
- data.actions	Array	动作列表，每个动作是一个 Object
- data.actions[0].id	String	Action ID
- data.actions[0].params	Object	动作参数，与请求中的 actions.params 一致
- data.actions[0].name	String	动作名字，与请求中的 actions.name 一致
- data.actions[0].metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics

6.23.3 PUT /api/v4/rules/{rule_id}

更新规则，返回规则 ID。

Parameters (json):

Name	Type	Required	Description
rawsql	String	True	可选，规则的 SQL 语句
actions	Array	True	可选，动作列表

- actions[0].name	String	True	可选，动作名称
- actions[0].params	Object	True	可选，动作参数。参数以 key-value 形式表示。详情可参看添加规则的示例
description	String	False	可选，规则描述

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	创建成功的规则对象，包含 Rule ID
- data.id	String	Rule ID
- data.rawsql	String	SQL 语句，与请求中的 rawsql 一致
- data.for	String	Topic 列表，表示哪些 topic 可以匹配到此规则
- data.metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics
- data.description	String	规则的描述信息，与请求中的 description 一致
- data.actions	Array	动作列表，每个动作是一个 Object
- data.actions[0].id	String	Action ID
- data.actions[0].params	Object	动作参数，与请求中的 actions.params 一致
- data.actions[0].name	String	动作名字，与请求中的 actions.name 一致
- data.actions[0].metrics	Array	统计指标，具体可参看 Dashboard 上的 Rule Metrics

6.23.4 DELETE /api/v4/rules/{rule_id}

删除规则。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

6.24 动作

查询规则引擎的动作。注意动作只能由 emqx 提供，不能添加。

6.24.1 GET api/v4/actions/{action_name}

获取某个动作的详情，包括动作名字、参数列表等。

Path Parameters:

Name	Type	Required	Description

action_name	String	False	可选，动作名。如不指定 action_name 则以数组形式返回当前支持的所有动作。

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.types	String	指示当前动作从属于那些资源类型
- data.title	Object	动作的简述，中英文。
- data.params	Object	动作的参数列表。参数以 key-value 形式表示。详情可参看后面的示例
- data.description	Object	动作的描述信息，中英文。
- data.app	String	动作的提供者

6.25 资源类型

查询规则引擎的资源类型。注意资源类型只能由 emqx 提供，不能添加。

6.25.1 GET api/v4/resource_types/{resource_type_name}

获取某个资源的详情，包括资源描述、参数列表等。

Path Parameters:

Name	Type	Required	Description
resource_type_name	String	False	可选，资源类型名。如不指定 resource_type_name 则以数组形式返回当前支持的所有资源类型。

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.title	Object	资源类型的简述，中英文。
- data.params	Object	资源类型的参数列表。参数以 key-value 形式表示。详情可参看后面的示例
- data.description	Object	资源类型的描述信息，中英文。
- data.provider	String	资源类型的提供者

6.26 资源

管理规则引擎的资源。资源是资源类型的实例，用于维护数据库连接等相关资源。

6.26.1 GET api/v4/resources/{resource_id}

获取指定的资源的详细信息。

Path Parameters:

Name	Type	Required	Description
resource_id	String	False	可选，资源类型 ID。如不指定 resource_id 则以数组形式返回当前所有的资源。

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.id	String	资源 ID
- data.type	String	资源所从属的资源类型的名字。
- data.config	Object	资源的配置。参数以 key-value 形式表示。 详情可参看后面的示例
- data.status	Array	资源的状态信息。详情请参看 Dashboard 上资源的状态。
- data.description	Object	资源的描述信息，中英文。

6.26.2 POST /api/v4/resources

创建规则，返回资源 ID。

Parameters (json):

Name	Type	Required	Description
type	String	True	资源类型名。指定要使用哪个资源类型创建资源。
config	Object	True	资源参数。要跟对应的资源类型的 params 里指定的格式相一致。
description	String	False	可选，资源描述

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

data	Object	规则对象
- data.id	String	资源 ID
- data.type	String	资源所从属的资源类型的名字。
- data.config	Object	资源的配置。参数以 key-value 形式表示。 详情可参看后面的示例
- data.description	Object	资源的描述信息，中英文。

6.26.3 DELETE /api/v4/resources/{resource_id}

删除资源。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

7. 版本管理与升级

7.1 版本管理

EMQ 通过版本迭代管理来提升产品性能和修复产品存在的一些问题。自版本发布之日起，EMQ X 企业版将为产品主要版本提供 18 个月的维护周期，同时我们会对最近两个主要版本分支的最后次要版本持续维护。

7.1.1 版本类型说明

- 主要版本，如 3.0.0, 4.0.0, 5.0.0, and 6.0.0 提供给我们一个引入非向后兼容功能的机会。
- 次要版本，比如 4.1.0 和 4.2.0，提供给我们加入新功能的机会。
- 维护版本，比如 4.1.1 和 4.1.2，只用于修复问题。维护活动在所有版本都会发生，但是主要通过次要版本的支流版本（例如 4.1.x）来确定为特定代码分支提供多长时间的维护。对次要版本的主动维护是指我们会修复一些错误，并将一些修复向后迁移至此代码分支。

7.1.2 版本维护政策

我们的目标是维护当前主要版本的最新次要版本，以及上一个主要版本的最新次要版本。我们观察到有些用户经常升级，随时与我们的版本支流保持同步。这些用户可以一直使用最

新的次要版本支流版本，并通过他们选择部署的维护版本获取修复。例如，这些用户可以紧跟我们的 EMQ X 版本：4.0.0、4.0.1、4.2.0 等等。

我们知道，并非所有用户都会在我们发布新版本后就立即升级。为方便这一类用户，我们会继续维护上一个主要版本的最新次要版本。以 EMQ X 3.x 为例，我们会继续为 3.x.x 系列提供维护。这样一来，这一类用户只需对目前运行的软件做一些小更改，即可完成修复。我们对最近发布的次要版本的维护将持续到下一个主要版本发布。例如，我们对 EMQ X 3.x.x 的维护将持续到 EMQ X 5.0.0 正式版本发布。EMQ X 5.0.0 发布后，我们将继续维护最近发布的 4.x 系列，并开始维护 5.0.x 次要版本系列，接着是 5.1.x 次要版本系列、5.2.x 次要版本系列。

我们有时会将修复向后迁移至次要版本的其他支流版本。比如，当多个分支可能都出现一个非常严重的安全漏洞时，我们会慎重地做出向后迁移的决定，不过我们希望尽少发生这样的情况。

7.2 版本升级

自 4.2.0 版本之后，EMQ X Broker 支持版本热升级。

使用版本热升级功能，用户可以快速、安全地升级生产环境的 EMQ X Broker，并避免了因重启服务导致的系统可用性降低。

目前 EMQ X Broker 仅支持 Patch 版本 (Patch 版本是版本号的第三位) 的热升级。即，目前支持 4.2.0 -> 4.2.1，4.2.0 -> 4.2.2，... 等的热升级，但 4.2.x 无法热升级到 4.3.0 或者 5.0。

目前 EMQ X for Windows 暂不支持热升级功能。

7.3 热升级步骤

查看当前已安装的 EMQ X Broker 的版本列表。

从 EMQ X 官网下载要升级的软件包。

访问 https://www.emqx.io/downloads#broker (opens new window)选择对应的版本和操作系统类型，然后选择 **"zip"** 包类型。

找到 EMQ X 的安装目录:

将下载的 zip 包放到 EMQ X 安装目录下的 releases 目录下:

升级到指定版本:

再次查看版本列表，之前的版本的状态将会变成 old:

7.4 升级后手动持久化

上面的 emqx upgrade 4.2.1 命令其实执行了三个动作:

- 解压 zip 包 (unpack)
- 安装 (install)
- 持久化 (permanent)

持久化之后，这次版本升级将会被固定下来，这意味着热升级后，如果 emqx 发生重启，使用的将是升级之后的新版本。 如果不想在升级的同时持久化，可以使用 --no-permanent 参数:

这时版本已经成功升级到了 4.2.1，但如果重启 emqx，将会还原到旧版本 4.2.0。 现在如果查看版本列表，会发现 4.2.1 的状态为 "当前版本"(current)，而不是持久化版本:

在系统稳定运行一段时间后，若决定持久化新版本，可以再次执行 install 命令:

7.5 版本降级

如果升级后发现问题想要回退，可以执行版本降级命令。 比如下面的例子会将 emqx 回退到 4.2.0 版本:

7.6 删除版本

在系统稳定运行一段时间后，若决定删除一个旧版本，可以执行版本卸载命令。 比如下面的例子将会卸载旧版本的 4.2.0: