



There's nothing here.

[Take me home](#)

目录

产品概览	22
EMQX 消息服务器功能列表	22
EMQX 消息服务器功能列表	24
功能说明	25
EMQX 企业版安装	28
EMQX License 文件获取	28
EMQX 程序包下载	28
CentOS	28
Ubuntu	29
Debian	31
macOS	32
Docker	33
利用 EMQX Operator 部署 EMQX 集群	34
启动 EMQX	36
查看 EMQX 的状态	36
License	36
基本命令	39
分布式集群	40
Erlang/OTP 分布式编程	40
EMQX 分布集群设计	41
节点发现与自动集群	42
防火墙设置	44
目录结构	46
bin 目录	46
etc 目录	46
log 目录	47
配置说明	48
简介	48
语法规则	48
数据类型	48
默认配置	50
Zone & Listener	50
配置更新	51
日志与追踪	52
控制日志输出	52
日志级别	52
日志文件和日志滚动	52
针对日志级别输出日志文件	53
日志格式	53

日志级别和 log handlers	54
运行时修改日志级别	55
日志追踪	56
Dashboard	58
基本使用	58
监控	58
客户端	62
主题	64
规则引擎	64
资源	66
编解码	67
模块	67
插件	68
告警	69
工具	69
设置	69
通用	71
认证	73
认证方式	73
认证结果	74
匿名认证	74
密码加盐规则与哈希方法	74
认证链	76
TLS 认证	76
PSK 认证	77
发布订阅 ACL	78
ACL 插件	78
规则详解	78
授权结果	79
全局配置	79
超级用户 (superuser)	80
ACL 缓存	80
ACL 鉴权链	80
规则引擎	82
EMQX 规则引擎快速入门	82
消息发布	83
事件触发	83
规则引擎组成	83
规则引擎典型应用场景举例	85
在 Dashboard 中测试 SQL 语句	85
迁移指南	87

数据存储	89
数据存储设计	89
消息桥接	91
桥接插件列表	91
共享订阅	92
带群组的共享订阅	92
不带群组的共享订阅	92
均衡策略与派发 Ack 配置	93
\$SYS 系统主题	94
集群状态信息	94
客户端上下线事件	94
系统统计 (Statistics)	95
收发流量 / 报文 / 消息统计	96
Alarms - 系统告警	98
Sysmon - 系统监控	98
黑名单	99
WebHook	100
配置项	100
触发规则	100
Webhook 事件参数	101
分布式集群	106
分布式 Erlang	106
EMQX 分布式集群设计	108
节点发现与自动集群	111
集群脑裂与自动愈合	114
集群节点自动清除	114
防火墙设置	114
钩子	116
定义	116
回调链	117
挂载点	118
回调函数	120
指标监控	123
Metrics & Stats	123
Stats	126
速率限制	128
速率限制原理	128
飞行窗口和消息队列	131
简介	131
飞行队列与 Receive Maximum	131

配置项	131
消息重传	132
基础配置	132
协议规范与设计	132
Alarm	136
数据导入导出	138
示例	138
模块管理	140
模块列表	140
启停模块	140
ACL File 访问控制	142
创建模块	142
定义 ACL	142
acl.conf 编写规则	143
MySQL 认证/访问控制	145
安装MySQL	145
创建模块	145
超级用户	150
加密规则	150
特殊说明	151
PostgreSQL 认证/访问控制	152
安装PostgreSQL	152
创建模块	152
超级用户	157
加密规则	157
Redis 认证/访问控制	159
创建模块	159
认证默认数据结构	160
加盐规则与哈希方法	161
访问控制默认数据结构	161
ACL 查询命令 (acl cmd)	162
超级用户查询命令 (super cmd)	162
HTTP 认证/访问控制	164
创建模块	164
HTTP 认证原理	165
认证请求	166
HTTP 访问控制原理	166
HTTP 请求信息	167
superuser 请求	167
ACL 访问控制请求	167

请求说明	167
内置数据库 认证/访问控制	169
Dashboard 管理	169
HTTP API	172
CLI	194
MongoDB 认证/访问控制	199
安装MongoDB	199
创建模块	199
认证集合	201
访问控制集合	202
超级用户查询	202
加密规则	203
PSKFile 认证	205
创建模块	205
LDAP 认证/访问控制	208
创建模块	208
LDAP Schema	210
LDAP 访问控制配置方式	211
JWT 认证	213
创建模块	213
认证原理	215
LwM2M 协议网关	217
协议介绍	217
创建模块	217
MQTT-SN 协议网关	223
协议介绍	223
创建模块	223
TCP 协议网关	227
协议介绍	227
创建模块	227
私有 TCP 协议 - v1	231
JT/T808 协议网关	235
协议介绍	235
创建模块	235
GB/T 32960 协议网关	242
协议介绍	242
创建模块	242
数据上报流程	245
数据下发流程	254
CoAP 协议网关	258

创建模块	258
使用示例	260
Stomp 协议网关	264
创建模块	264
Kafka 消费组	269
创建模块	269
Pulsar 消费组	273
创建模块	273
MQTT 订阅者	277
创建模块	277
多语言扩展 - 协议接入	279
特性	279
架构	279
接口设计	280
开发指南	281
## 创建模块	282
多语言扩展 - 钩子	285
设计	285
接口设计	285
开发指南	287
创建模块	287
recon	290
创建模块	290
EMQX Prometheus Agent	291
创建模块	291
热配置	294
创建模块	294
主题监控	296
创建模块	296
如何使用主题监控	296
MQTT 增强认证	298
上下线通知	299
创建模块	299
上下线消息通知格式	299
MQTT 代理订阅	301
创建模块	301
订阅选项	303
代理订阅规则	303
主题重写	305
创建模块	305

主题重写规则	307
主题重写示例	307
MQTT 保留消息	309
简介	309
创建模块	309
保留消息配置简介	310
Trace 日志追踪	311
简介	311
创建模块	311
追踪指定 ClientID	311
追踪指定 Topic	311
追踪指定 IP	312
慢订阅统计	314
开启模块	314
实现说明	314
配置说明	314
慢订阅记录	315
延迟发布	317
创建模块	317
延迟发布简介	317
规则引擎	319
EMQX 规则引擎快速入门	319
消息发布	320
事件触发	320
规则引擎组成	320
规则引擎典型应用场景举例	322
在 Dashboard 中测试 SQL 语句	322
迁移指南	324
规则引擎语法与示例	326
SQL 语法	326
SQL 语句示例	327
规则引擎 SQL 语句中可用的字段	335
使用规则引擎 SQL 语句处理消息发布	335
使用规则引擎 SQL 语句处理事件	336
SQL 语句中可用的函数	348
数学函数	348
数据类型判断函数	349
数据类型转换函数	350
字符串函数	350
Map 函数	353

数组函数	354
哈希函数	354
压缩解压缩函数	355
比特操作函数	355
编解码函数	356
时间与日期函数	357
创建规则	361
使用 Dashboard 创建规则	361
通过 CLI 创建简单规则	366
如何更新资源	370
通过管理界面	370
通过命令行	371
检查 (调试)	374
发送数据到 Web 服务	375
保存数据到 MySQL	378
保存数据到 PostgreSQL	386
保存数据到 MongoDB	393
创建资源	393
创建规则	397
测试验证	399
保存数据到 Redis	401
保存数据到 Cassandra	407
保存数据到 DynamoDB	414
保存数据到 ClickHouse	420
保存数据到 OpenTSDB	426
保存数据到 InfluxDB	433
数据保存到 InfluxDB V2 & InfluxDB Cloud	439
保存数据到 TimescaleDB	444
保存数据到 Oracle DB	451
保存数据到 SQLServer	456
保存数据到 DolphinDB	463
搭建 DolphinDB	463
配置规则引擎	464
保存数据到 MatrixDB	471
Save data to TDengine	477
原生方式 (企业版)	478
通过发送数据到 Web 服务写入	480
测试	484

保存数据到 Lindorm 数据库 (自 e3.3.6 和 e4.4.1 起)	485
保存数据到 Alibaba Tablestore 数据库	489
创建时间线	489
创建规则引擎资源	489
创建规则	490
桥接数据到 Kafka	495
桥接数据到 Pulsar	501
使用 Pulsar basic/token 认证	506
桥接数据到 RabbitMQ	508
桥接数据到 RocketMQ	515
桥接数据到 SAP Event Mesh	521
准备 SAP Event Mesh 环境	521
创建规则:	522
发送消息测试	527
桥接数据到 MQTT Broker	528
桥接数据到 RPC 服务	534
离线消息保存到 Redis	540
离线消息保存到 MySQL	548
离线消息保存到 PostgreSQL	557
离线消息保存到 Cassandra	565
离线消息保存到 MongoDB	574
离线消息保存到 ClickHouse	583
从 Redis 中获取订阅关系	591
从 MySQL 中获取订阅关系	597
从 PostgreSQL 中获取订阅关系	604
从 Cassandra 中获取订阅关系	609
从 MongoDB 中获取订阅关系	616
从 Redis 中获取订阅关系	621
编解码 (Schema Registry) 介绍	627
数据格式	627
架构设计	628
编解码 + 规则引擎	628
编解码举例 - Avro	631
规则需求	631
创建 Schema	631
创建规则	631
设备端代码	632
检查规则执行结果	632

编解码举例 - Protobuf	633
规则需求	633
创建 Schema	633
创建规则	633
设备端代码	634
检查规则执行结果	634
编解码举例 - 自定义 HTTP 编解码	635
规则需求	635
创建 Parser HTTP 资源	635
创建 Schema	635
创建规则	635
编解码服务端代码	636
检查规则执行结果	637
编解码举例 - 自定义 gRPC 编解码	638
规则需求	638
创建 Parser gRPC 资源	638
创建 Schema	638
创建规则	638
编解码服务端代码	639
检查规则执行结果	640
插件	641
插件列表	641
启停插件	642
插件开发	643
数据存储	647
数据存储设计	647
Redis 数据存储	649
配置 Redis 服务器	649
配置 Redis 存储规则	649
Redis 存储规则说明	650
Redis 命令行参数说明	651
Redis 命令行配置 Action	651
Redis 设备在线状态 Hash	652
Redis 保留消息 Hash	652
Redis 消息存储 Hash	653
Redis 消息确认 SET	653
Redis 订阅存储 Hash	653
Redis SUB/UNSUB 事件发布	654
启用 Redis 数据存储插件	654
MySQL 数据存储	656
配置 MySQL 服务器	656

配置 MySQL 存储规则	656
MySQL 存储规则说明	657
SQL 语句参数说明	657
SQL 语句配置 Action	658
创建 MySQL 数据库表	658
导入 MySQL 库表结构	658
MySQL 设备在线状态表	658
MySQL 主题订阅表	659
MySQL 消息存储表	660
MySQL 保留消息表	661
MySQL 消息确认表	662
启用 MySQL 数据存储插件	663
PostgreSQL 数据存储	664
配置 PostgreSQL 服务器	664
配置 PostgreSQL 存储规则	664
PostgreSQL 存储规则说明	665
SQL 语句参数说明	665
SQL 语句配置 Action	666
创建 PostgreSQL 数据库	666
导入 PostgreSQL 库表结构	666
PostgreSQL 设备在线状态表	666
PostgreSQL 代理订阅表	667
PostgreSQL 消息存储表	668
PostgreSQL 保留消息表	669
PostgreSQL 消息确认表	669
启用 PostgreSQL 数据存储插件	670
MongoDB 消息存储	671
配置 MongoDB 消息存储	671
配置 MongoDB 服务器	671
MongoDB 数据库初始化	673
MongoDB 用户状态集合(Client Collection)	673
MongoDB 用户订阅主题集合(Subscription Collection)	674
MongoDB 发布消息集合(Message Collection)	675
MongoDB 保留消息集合(Retain Message Collection)	676
MongoDB 接收消息 ack 集合(Message Acked Collection)	676
启用 MongoDB 数据存储插件	676
Cassandra 消息存储	677
配置 Cassandra 服务器	677
Cassandra 创建一个 Keyspace	679
导入 Cassandra 表结构	679
Cassandra 用户状态表(Client Table)	679
Cassandra 用户订阅主题表(Sub Table)	680

Cassandra 发布消息表(Msg Table)	681
Cassandra 保留消息表(Retain Message Table)	681
Cassandra 接收消息 ack 表(Message Acked Table)	682
启用 Cassandra 存储插件	682
DynamoDB 消息存储	683
配置 DyanmoDB 消息存储	683
DynamoDB 数据库创建表	684
DynamoDB 用户状态表(Client Table)	684
DynamoDB 用户订阅主题(Subscription Table)	685
DynamoDB 发布消息(Message Table)	686
DynamoDB 保留消息(Retain Message Table)	687
DynamoDB 接收消息 ack (Message Acked Table)	688
InfluxDB 消息存储	690
InfluxDB 配置	690
配置 InfluxDB 消息存储	690
OpenTSDB 消息存储	697
配置 OpenTSDB 消息存储	697
Timescale 消息存储	701
配置 Timescale 消息存储	701
消息桥接	706
桥接插件列表	706
MQTT 桥接	707
配置 MQTT 桥接的 Broker 地址	707
配置 MQTT 桥接转发和订阅主题	708
MQTT 桥接转发和订阅主题说明	709
启用 bridge_mqtt 桥接插件	709
桥接 CLI 命令	709
列出全部 bridge 状态	709
启动指定 bridge	710
停止指定 bridge	710
列出指定 bridge 的转发主题	710
添加指定 bridge 的转发主题	710
删除指定 bridge 的转发主题	710
列出指定 bridge 的订阅	710
添加指定 bridge 的订阅主题	710
删除指定 bridge 的订阅主题	711
RPC 桥接	712
配置 RPC 桥接的 Broker 地址	712
配置 MQTT 桥接转发和订阅主题	712
MQTT 桥接转发和订阅主题说明	712
桥接 CLI 命令	712

Kafka 桥接	713
配置 Kafka 集群地址	713
配置 Kafka 桥接规则	714
Kafka 桥接规则说明	715
客户端上下线事件转发 Kafka	716
客户端订阅主题事件转发 Kafka	716
MQTT 消息转发到 Kafka	717
MQTT 消息派发 (Deliver) 事件转发 Kafka	717
MQTT 消息确认 (Ack) 事件转发 Kafka	717
Kafka 消费示例	718
启用 Kafka 桥接插件	719
RabbitMQ 桥接	720
配置 RabbitMQ 桥接地址	720
配置 RabbitMQ 桥接规则	721
客户端订阅主题事件转发 RabbitMQ	721
客户端取消订阅事件转发 RabbitMQ	721
MQTT 消息转发 RabbitMQ	721
MQTT 消息确认 (Ack) 事件转发 RabbitMQ	722
RabbitMQ 订阅消费 MQTT 消息示例	722
启用 RabbitMQ 桥接插件	722
Pulsar 桥接	723
配置 Pulsar 集群地址	723
配置 Pulsar 桥接规则	723
Pulsar 桥接规则说明	724
客户端上下线事件转发 Pulsar	725
客户端订阅主题事件转发 Pulsar	725
客户端取消订阅主题事件转发 Pulsar	725
MQTT 消息转发到 Pulsar	726
MQTT 消息派发 (Deliver) 事件转发 Pulsar	726
MQTT 消息确认 (Ack) 事件转发 Pulsar	726
Pulsar 消费示例	727
启用 Pulsar 桥接插件	727
RocketMQ 桥接	729
配置 RocketMQ 集群地址	729
配置 RocketMQ 桥接规则	729
RocketMQ 桥接规则说明	730
客户端上下线事件转发 RocketMQ	730
客户端订阅主题事件转发 RocketMQ	731
客户端取消订阅主题事件转发 RocketMQ	731
MQTT 消息转发到 RocketMQ	731

MQTT 消息派发 (Deliver) 事件转发 RocketMQ	731
MQTT 消息确认 (Ack) 事件转发 RocketMQ	732
RocketMQ 消费示例	732
启用 RocketMQ 桥接插件	733
设备管理	734
设备认证	734
在线状态与连接历史管理	734
发布订阅/ACL	734
代理订阅	734
HTTP 消息发布	734
系统调优	735
Linux 操作系统参数	735
TCP 协议栈网络参数	735
Erlang 虚拟机参数	736
docker 参数调优	736
EMQX 消息服务器参数	737
测试客户端设置	737
生产部署	738
部署架构	738
青云 (QingCloud) 部署	739
亚马逊 (AWS) 部署	740
私有网络部署	742
Prometheus 监控告警	744
配置	744
性能测试	745
编译安装	745
使用	745
典型压测场景	747
HTTP API	750
接口安全	750
响应码	750
API Endpoints	751
/api/v4	751
Broker 基本信息	752
节点	753
客户端	754
订阅信息	763
路由	766
消息发布	767
主题订阅	768
消息批量发布	769

主题批量订阅	770
插件	771
监听器	773
内置模块	775
统计指标	777
主题统计指标	782
状态	786
告警	788
ACL 缓存	791
黑名单	792
数据导入导出	794
规则	797
动作	801
资源类型	802
资源	803
配置项	807
Cluster	807
Node	813
RPC	817
Log	821
authacl	825
mqtt	827
zoneexternal	829
zoneinternal	837
tcpexternal	842
tcpinternal	847
tlsexternal	851
wsexternal	860
wssexternal	867
plugins	879
broker	880
broker.perf.route_lock_type = key	881
broker.perf.trie_compaction = true	882
monitor	882
插件 emqx-auth-http	885
插件 emqx_auth_jwt	890
插件 emqx_auth_ldap	892
插件 emqx_auth_mongo	895
插件 emqx_auth_mysql	904
插件 emqx_auth_pgsql	907
插件 emqx_auth_redis	910
插件 emqx_bridge_mqtt	913

插件 emqx_coap	919
插件 emqx_dashboard	921
插件 emqx_lwm2m	927
插件 emqx_management	931
插件 emqx_retainer	937
插件 emqx_rule_engine	938
插件 emqx_sn	939
插件 emqx_prometheus	941
插件 emqx_stomp	942
插件 emqx_web_hook	947
使用环境变量修改配置	953
示例	953
命令行接口	954
status 命令	954
mgmt 命令	954
broker 命令	955
cluster 命令	958
acl 命令	959
clients 命令	960
routes 命令	961
subscriptions 命令	962
plugins 命令	963
modules 命令	964
vm 命令	965
mnesia 命令	967
log 命令	967
trace 命令	969
listeners	970
recon 命令	972
retainer 命令	972
admins 命令	973
规则引擎(rule engine) 命令	974
与规则引擎相关状态、统计指标和告警	978
EMQX 内置数据库 Auth 与 ACL 规则	979
pem_cache 命令	983
版本热升级	984
热升级步骤	984
升级后手动持久化	985
版本降级	986
删除版本	986
在运行时安装 EMQX 补丁包	987

安装 EMQX 补丁包的步骤	987
回滚补丁包	988
入门概念	989
EMQX 是什么?	989
为什么选择EMQX?	989
EMQX 的主题数量有限制吗?	989
EMQX 开源版怎么存储数据?	989
EMQX 与物联网平台的关系是什么?	989
EMQX 有哪些产品?	989
EMQX 与 NB-IoT、LoRAWAN 的关系是什么?	990
MQTT 协议与 HTTP 协议相比, 有何优点和弱点?	990
什么是认证鉴权? 使用场景是什么?	990
什么是 Hook? 使用场景是什么?	991
什么是 mqueue? 如何配置 mqueue?	991
什么是 WebSocket? 什么情况下需要通过 WebSocket 去连接 EMQX 服务器?	991
什么是共享订阅? 有何使用场景?	992
什么是离线消息?	992
什么是代理订阅? 使用场景是什么?	992
系统主题有何用处? 都有哪些系统主题?	992
为什么开启认证后, 客户端还是可以不需要用户名密码就能连接?	993
使用教程	994
怎么样才能使用 EMQX?	994
怎样更新 EMQX license?	994
EMQX 支持私有协议进行扩展吗? 如支持应该如何实现?	994
我可以捕获设备上下线的事件吗? 该如何使用?	995
我想限定某些主题只为特定的客户端所使用, EMQX 该如何进行配置?	995
EMQX 能做流量控制吗?	995
EMQX 是如何实现支持大规模并发和高可用的?	996
EMQX 能把接入的 MQTT 消息保存到数据库吗?	996
在服务器端能够直接断开一个 MQTT 连接吗?	996
EMQX 能把接入的消息转发到 Kafka 吗?	997
EMQX 企业版中桥接 Kafka, 一条 MQTT 消息到达 EMQX 集群之后就回 MQTT Ack 报文还是写入 Kafka 之后才回	997
EMQX 支持集群自动发现吗? 有哪些实现方式?	997
我可以把 MQTT 消息从 EMQX 转发其他消息中间件吗? 例如 RabbitMQ?	998
我可以把消息从 EMQX 转到公有云 MQTT 服务上吗? 比如 AWS 或者 Azure 的 IoT Hub?	998
MQTT Broker (比如 Mosquitto) 可以转发消息到 EMQX 吗?	998
我想跟踪特定消息的发布和订阅过程, 应该如何做?	998
为什么我做压力测试的时候, 连接数目和吞吐量老是上不去, 有系统调优指南吗?	998
EMQX 支持加密连接吗? 推荐的部署方案是什么?	999
EMQX 安装之后无法启动怎么排查?	999
EMQX中ssl resumption session的使用	1001

MQTT 客户端断开连接统计	1001
安装部署	1002
EMQX 推荐部署的操作系统是什么?	1002
EMQX 支持 Windows 操作系统吗?	1002
EMQX 如何预估资源的使用?	1002
EMQX 的百万连接压力测试的场景是什么?	1002
我的连接数目并不大, EMQX 生产环境部署需要多节点吗?	1002
常见错误	1004
EMQX 无法连接 MySQL 8.0	1004
OPENSSL 版本不正确	1004
Windows 缺失 MSVCR120.dll	1006
SSL 连接失败	1007
商业服务	1009
EMQX 企业版 (Enterprise) 和开源版 (Broker) 的主要区别是什么?	1009
EMQX 提供方案咨询服务吗?	1009
标签	1010
启动失败	1010
企业版	1010
NB-IoT	1010
LoRAWAN	1010
多协议	1010
License	1010
扩展	1010
资源估算	1010
认证鉴权	1010
WebHook	1010
系统主题	1011
消息队列	1011
WebSocket	1011
ACL	1011
发布订阅	1011
共享订阅	1011
流量控制	1011
离线消息	1011
代理订阅	1011
性能	1011
高并发	1011
持久化	1012
HTTP API	1012
Dashboard	1012
Kafka	1012

桥接	1012
配置	1012
集群	1012
RabbitMQ	1012
Mosquitto	1012
Trace	1012
调试	1013
性能测试	1013
TLS	1013
加密连接	1013
MySQL	1013
认证	1013
指标	1013
MQTT 客户端库	1014
MQTT 客户端生命周期	1014
MQTT C 客户端库	1016
Paho C 使用示例	1016
Paho C MQTT 5.0 支持	1017
MQTT Java 客户端库	1019
通过 Maven 安装 Paho Java	1019
Paho Java 使用示例	1019
Paho Java MQTT 5.0 支持	1021
MQTT Go 客户端库	1022
MQTT Go 使用示例	1022
Paho Golang MQTT 5.0 支持	1023
MQTT Erlang 客户端库	1025
emqtt 使用示例	1025
emqtt MQTT 5.0 支持	1025
MQTT JavaScript 客户端库	1026
MQTT.js 使用示例	1026
MQTT.js MQTT 5.0 支持	1027
MQTT Python 客户端库	1028
Paho Python 使用示例	1028
Paho Python MQTT 5.0 支持	1028
EMQX MQTT 微信小程序接入	1029
参考资料	1029
详细步骤	1029
其他资源	1031
协议介绍	1036
MQTT协议	1036

MQTT-SN 协议	1042
LWM2M 协议	1043
私有 TCP 协议	1045
LwM2M 协议网关	1048
协议介绍	1048
创建模块	1048
MQTT-SN 协议网关	1054
协议介绍	1054
创建模块	1054
TCP 协议网关	1058
协议介绍	1058
创建模块	1058
私有 TCP 协议 - v1	1062
JT/T808 协议网关	1066
协议介绍	1066
创建模块	1066
CoAP 协议网关	1073
创建模块	1073
使用示例	1075
Stomp 协议网关	1079
创建模块	1079
版本发布	1084
4.4.4 版本	1084
4.4.3 版本	1085
4.4.2 版本	1085
4.4.1 版本	1087
4.4.0 版本	1088
生命周期 EOL	1090
概要	1090
版本类型	1090
维护政策	1090
维护表	1090
从 4.3 升级到 4.4 版本	1092
数据及配置备份	1092
卸载 4.3 版本	1092
安装 4.4 版本， 并导入4.3版本数据。	1093
启动 4.4 版本	1093
架构设计	1094
前言	1094
系统架构	1095

连接层设计	1095
会话层设计	1096
路由层设计	1096
分布层设计	1097
Mnesia/ETS 表设计	1097
Erlang 设计相关	1098
资源	1099
官方资源	1099
社区、讨论、贡献和支持	1099
使用 EMQX 的项目	1099
中文教程	1099
MQTT 规范	1099

产品概览

EMQX (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。

EMQX 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，单服务器节点支持 **200** 万连接。
2. 分布式节点集群，快速低延时的消息路由。
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。

EMQX 消息服务器功能列表

- 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
- **MQTT/WebSocket TCP/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

EMQX 消息服务器功能列表

- 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

以下是 **EMQX Enterprise** 特有功能

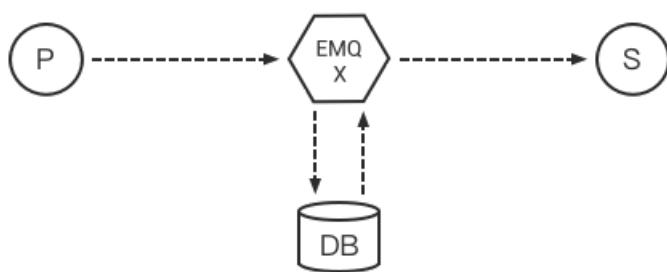
- **Scalable RPC** 架构: 分离 **Erlang** 自身的集群通道与 **EMQX** 节点间的数据通道
- 数据持久化
 - **Redis** 存储订阅关系、设备在线状态、**MQTT** 消息、保留消息，发布 **SUB/UNSUB** 事件
 - **MySQL** 存储订阅关系、设备在线状态、**MQTT** 消息、保留消息

- 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：EMQX 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 Kafka
 - RabbitMQ 桥接：EMQX 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 RabbitMQ
 - Pulsar 桥接：EMQX 内置 Bridge 直接转发 MQTT 消息、设备上下线事件到 Pulsar
 - RocketMQ 桥接：EMQX 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 EMQX 的事件、消息 提供了数据编解码能力

功能说明

消息数据存储

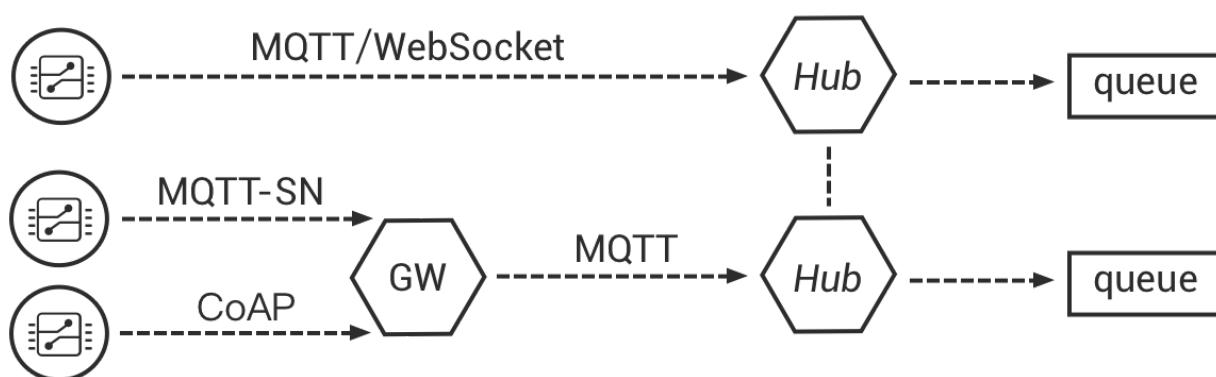
EMQX 企业版支持存储订阅关系、MQTT 消息、设备状态到 Redis、MySQL、PostgreSQL、MongoDB、Cassandra、TimescaleDB、InfluxDB、DynamoDB、OpenTDSB 数据库：



数据存储相关配置，详见“数据存储”章节。

消息桥接转发

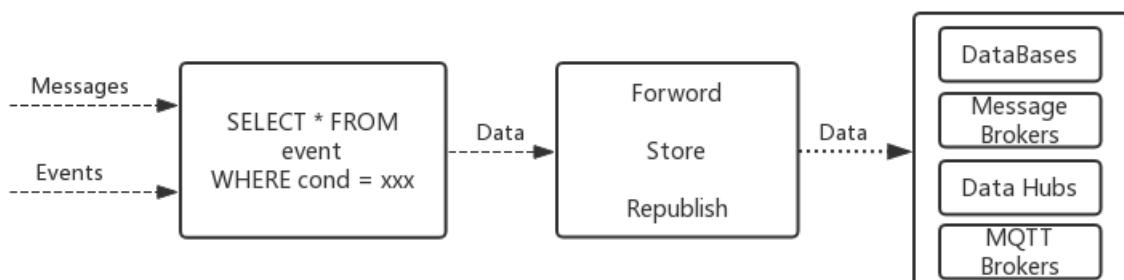
EMQX 企业版支持直接转发 MQTT 消息到 **RabbitMQ**、**Kafka**、**Pulsar**、**RocketMQ**、**MQTT Broker**，可作为百万级的物联网接入服务器(**IoT Hub**)：



规则引擎

EMQX 规则引擎可以灵活地处理消息和事件。

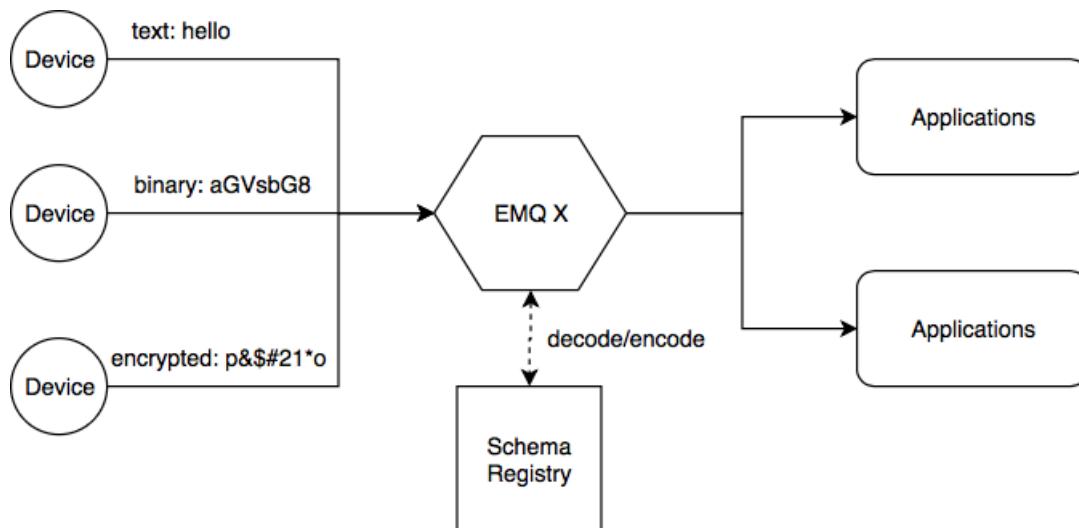
EMQX 企业版规则引擎支持消息重新发布；桥接数据到 **Kafka**、**Pulsar**、**RocketMQ**、**RabbitMQ**、**MQTT Broker**；保存数据到 **MySQL**、**PostgreSQL**、**Redis**、**MongoDB**、**DynamoDB**、**Cassandra**、**InfluxDB**、**OpenTSDB**、**TimescaleDB**；发送数据到 **WebServer**。



规则引擎相关配置，详见“规则引擎”章节。

编解码

Schema Registry 目前可支持三种格式的编解码：[Avro](#)，[Protobuf](#)，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。

EMQX 企业版安装

EMQX 消息服务器可跨平台运行在 **Linux** 服务器上。

EMQX License 文件获取

联系商务或登陆 <https://www.emqx.com> 注册账号获取免费的试用 License 文件

EMQX 程序包下载

EMQX 消息服务器每个版本会发布 **CentOS**、**Ubuntu**、**Debian** 平台程序包与 **Docker** 镜像。

下载地址: <https://www.emqx.com/zh/downloads>

CentOS

- **CentOS 7 (EL7)**
- **CentOS 8 (EL8)**

使用 rpm 包安装 EMQX

1. 访问[emqx.com](https://www.emqx.com) 选择 **CentOS** 版本，然后下载要安装的 EMQX 版本的 **rpm** 包。

2. 安装 EMQX

```
1 $ sudo rpm -ivh emqx-ee-centos7-v4.0.0.x86_64.rpm           sh
```

3. 导入License文件:

```
1 $ cp /path/to/emqx.lic /etc/emqx/emqx.lic                  sh
```

4. 启动 EMQX

- 直接启动

```
1 $ emqx start                                              sh
2 emqx  is started successfully!
3
4 $ emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running
```

- **systemctl** 启动

```
1 $ sudo systemctl start emqx
```

sh

- **service** 启动

```
1 $ sudo service emqx start
```

sh

使用 **zip** 包安装 EMQX

注意

ZIP包适用于测试和热更，如果不知道如何手动安装所有可能的运行时依赖，请勿在生产环境中使用

1. 通过 [emqx.com](#) 选择 **Centos** 版本，然后下载要安装的 **EMQX** 版本的 **zip** 包。

2. 解压程序包

```
1 $ unzip emqx-ee-centos7-v4.0.0.zip
```

sh

3. 导入**License**文件:

```
1 $ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic
```

sh

4. 启动 **EMQX**

```
1 $ ./bin/emqx start
2 emqx v4.0.0 is started successfully!
3
4 $ ./bin/emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running
```

sh

Ubuntu

- **Bionic 18.04 (LTS)**
- **Xenial 16.04 (LTS)**

使用 **deb** 包安装 EMQX

1. 通过 [emqx.com](#) 选择 **Ubuntu** 版本，然后下载要安装的 **EMQX** 版本的 **deb** 包。

2. 安装 **EMQX**

```

1 # for ubuntu
2 $ sudo apt install ./emqx-ee-ubuntu18.04-v3.1.0_amd64.deb
3 # for debian
4 $ sudo dpkg -i emqx-ee-ubuntu18.04-v3.1.0_amd64.deb

```

sh

3. 导入 License 文件:

```

1 $ cp /path/to/emqx.lic /etc/emqx/emqx.lic

```

sh

4. 启动 EMQX

- 直接启动

```

1 $ emqx start
2 emqx is started successfully!
3
4 $ emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running

```

sh

- systemctl** 启动

```

1 $ sudo systemctl start emqx

```

sh

- service** 启动

```

1 $ sudo service emqx start

```

sh

使用 zip 包安装 EMQX

注意

ZIP包适用于测试和热更，如果不知道如何手动安装所有可能的运行时依赖，请勿在生产环境中使用

- 通过 [emqx.com](#) 选择 Ubuntu 版本，然后下载要安装的 EMQX 版本的 zip 包。

- 解压程序包

```

1 $ unzip emqx-ee-ubuntu18.04-v4.0.0.zip

```

sh

3. 导入 License 文件:

```

1 $ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic

```

sh

4. 启动 EMQX

```

1 $ ./bin/emqx start
2 emqx v4.0.0 is started successfully!
3
4 $ ./bin/emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running

```

sh

Debian

- [Debian 9](#)
- [Debian 10](#)

使用 **deb** 包安装 EMQX

1. 通过 [emqx.com](#) 选择 **Debian** 版本，然后下载要安装的 **EMQX** 版本的 **deb** 包。
2. 安装 **EMQX**

```

1 # for ubuntu
2 $ sudo apt install ./emqx-ee-debian9-v3.1.0_amd64.deb
3
4 # for debian
5 # 首先确保已安装 libodbc
6 $ sudo dpkg -i emqx-ee-debian9-v3.1.0_amd64.deb

```

sh

3. 导入**License**文件:

```

1 $ cp /path/to/emqx.lic /etc/emqx/emqx.lic

```

sh

4. 启动 **EMQX**

- 直接启动

```

1 $ emqx start
2 emqx v4.0.0 is started successfully!
3
4 $ emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running

```

sh

- **systemctl** 启动

```

1 $ sudo systemctl start emqx

```

sh

- **service** 启动

```

1 $ sudo service emqx start

```

sh

使用 zip 包安装 EMQX

注意

ZIP包适用于测试和热更，如果不知道如何手动安装所有可能的运行时依赖，请勿在生产环境中使用

1. 通过 [emqx.com](#) 选择 **Debian** 版本，然后下载要安装的 **EMQX** 版本的 **zip** 包。

2. 解压程序包

```
1 $ unzip emqx-ee-debian9-v4.0.0.zip
```

sh

3. 导入**License**文件:

```
1 $ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.lic
```

sh

4. 启动 **EMQX**

```
1 $ ./bin/emqx start
2 emqx v4.0.0 is started successfully!
3
4 $ ./bin/emqx_ctl status
5 Node 'emqx@127.0.0.1' is started
6 emqx 4.0.0 is running
```

sh

macOS

使用 ZIP 包安装 EMQX

1. 通过 [emqx.com](#)，选择 **EMQX** 版本，然后下载要安装的 **zip** 包。

2. 解压缩包

```
1 $ unzip emqx-ee-macos-v4.0.0.zip
```

sh

3. 导入**License**文件:

```
1 $ cp /path/to/emqx.lic /path/to/emqx/etc/emqx.li
```

sh

4. 启动 **EMQX**

```
1 $ ./bin/emqx start  
2 emqx v4.0.0 is started successfully!  
3  
4 $ ./bin/emqx_ctl status  
5 Node 'emqx@127.0.0.1' is started  
6 emqx 4.0.0 is running
```

sh

Docker

1. 获取 docker 镜像

- 通过 [Docker Hub](#) 获取

```
1 $ docker pull emqx/emqx-ee:v4.0.0
```

sh

2. 启动 docker 容器

```
1 $ docker run -d -\  
2   --name emqx-ee \  
3   -p 1883:1883 \  
4   -p 8083:8083 \  
5   -p 8883:8883 \  
6   -p 8084:8084 \  
7   -p 18083:18083 \  
8   -v /path/to/emqx.lic:/opt/emqx/etc/emqx.lic  
9   emqx/emqx-ee:v4.0.0
```

sh

更多关于 **EMQX Docker** 的信息请查看 [Docker Hub](#)

利用 EMQX Operator 部署 EMQX 集群

1. 使用 [cert-manager](#) 给 **webhook** 服务提供证书，可以参考 [cert-manager 文档](#) 安装

2. 可以通过如下两种方式之一来安装 **Operator Controller**:

- 使用静态文件安装

```
1 curl -f -L "https://github.com/emqx/emqx-operator/releases/download/1.1.7/emqx-operator-controller.yaml" | kubectl apply -f -
```

- 使用 **Helm** 安装

- 添加 **EMQX Helm** 仓库

```
1 helm repo add emqx https://repos.emqx.io/charts
2 helm repo update
```

- 用 **Helm** 安装 **EMQX Operator** 控制器

```
1 $ helm install emqx-operator emqx/emqx-operator \
2   --set installCRDs=true \
3   --namespace emqx-operator-system \
4   --create-namespace
```

3. 检查 **EMQX Operator** 控制器状态

```
1 $ kubectl get pods -l "control-plane=controller-manager" -n emqx-operator-system
2 NAME READY STATUS RESTARTS AGE
3 emqx-operator-controller-manager-68b866c8bf-kd4g6 1/1 Running 0 15s
```

4. 部署 **EMQX Enterprise**

- 部署 **EMQX Custom Resource**

```
1 cat << "EOF" | kubectl apply -f -
2 apiVersion: apps.emqx.io/v1beta2
3 kind: EmqxEnterprise
4 metadata:
5   name: emqx-ee
6 spec:
7   image: emqx/emqx-ee:4.4.3
8 EOF
```

- 检查 **EMQX** 状态

```
1 $ kubectl get pods
2 NAME READY STATUS RESTARTS AGE
3 emqx-ee-0 1/1 Running 0 22s
4 emqx-ee-1 1/1 Running 0 22s
5 emqx-ee-2 1/1 Running 0 22s
6 $ kubectl exec -it emqx-ee-0 -- emqx_ctl status
7 Node 'emqx-ee@emqx-ee-0.emqx-ee-headless.default.svc.cluster.local' 4.4.3 is started
8 $ kubectl exec -it emqx-ee-0 -- emqx_ctl cluster status
9 Cluster status: #{running_nodes =>
10           ['emqx-ee@emqx-ee-0.emqx-ee-headless.default.svc.cluster.local',
11            'emqx-ee@emqx-ee-1.emqx-ee-headless.default.svc.cluster.local',
12            'emqx-ee@emqx-ee-2.emqx-ee-headless.default.svc.cluster.local'],
13 stopped_nodes => []}
```

启动 EMQX

后台启动 **EMQX**

```
1 $ emqx start
2 EMQX v4.0.0 is started successfully!
```

sh

systemctl 启动

```
1 $ sudo systemctl start emqx
2 EMQX v4.0.0 is started successfully!
```

sh

service 启动

```
1 $ sudo service emqx start
2 EMQX v4.0.0 is started successfully!
```

sh

查看 EMQX 的状态

EMQX 正常启动:

```
1 $ emqx_ctl status
2 Node 'emqx@127.0.0.1' is started
3 emqx 4.0.0 is running
```

sh

EMQX 未能正常启动:

```
1 $ emqx_ctl status
2 Node 'emqx@127.0.0.1' not responding to pings.
```

sh

你可以查看 `logs` 下的日志文件并确认是否属于 [常见错误](#)。

License

EMQX Enterprise 需要 **License** 文件才能正常启动, 请联系销售人员或在线自助购买/申请试用以获取 **License**。

- 试用版 **License**: 到期后将停止正在运行的 **EMQX**;
- 正式版 **License**: 到期后不会停止正在运行的 **EMQX**, 但是新节点或手动停止之后的节点将无法启动。

申请试用 **License**

- 访问 [EMQX Enterprise 下载页面](#), 点击 [免费获取 License](#)。

类型



开源版



企业版



Cloud

版本

v4.3.3

更新日志 | 历史版本

NEW
配置估算

Docker



CentOS



Debian



macOS



Ubuntu

1. 获取 Docker 镜像

```
docker pull emqx/emqx-ee:4.3.3
```



2. 启动 Docker 容器

```
docker run -d --name emqx-ee -p 1883:1883 -p 8081:8081 -p 8083:8083 -p 8084:8084 -p 8883:8883 -p 18083:18083 emqx/emqx-ee:4.3.3
```



免费获取 License

使用指南

- 申请 **License** 文件试用, 下载 **License** 文件。

姓名 *

公司 *

Email (该邮箱将用来接收 License) *

应用场景 *

电话 *

短信验证码 *

获取验证码

留言

免费申请

放置 License

- 替换默认证书目录下的 **License** 文件（`etc/emqx.lic`），当然你也可以选择变更证书文件的读取路径，修改 `etc/emqx.conf` 文件中的 `license.file`，并确保 **License** 文件位于更新后的读取路径且 **EMQX Enterprise** 拥有读取权限，然后启动 **EMQX Enterprise**。**EMQX Enterprise** 的启动方式与 **EMQX** 相同，见下文。
- 如果是正在运行的 **EMQX Enterprise** 需要更新 **License** 文件，那么可以使用 `emqx_ctl license reload [license 文件所在路径]` 命令直接更新 **License** 文件，无需重启 **EMQX Enterprise**。

提示

`emqx_ctl license reload` 命令加载的证书仅在 **EMQX Enterprise** 本次运行期间生效，如果需要永久更新 **License** 证书的路径，依然需要替换旧证书或修改配置文件。

基本命令

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

- `emqx start`

后台启动 **EMQX Broker**；

- `emqx stop`

关闭 **EMQX Broker**；

- `emqx restart`

重启 **EMQX Broker**；

- `emqx console`

使用控制台启动 **EMQX Broker**；

- `emqx foreground`

使用控制台启动 **EMQX Broker**，与 `emqx console` 不同，`emqx foreground` 不支持输入 **Erlang** 命令；

- `emqx ping`

Ping EMQX Broker。

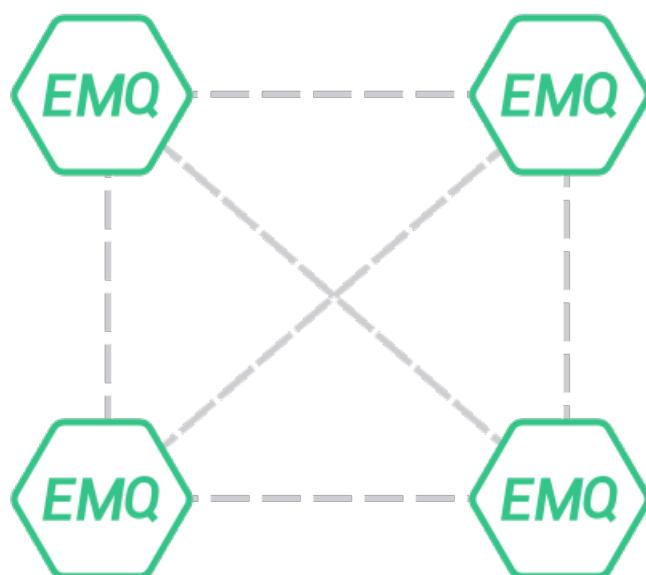
以上命令为用户常用命令，此外 `emqx` 命令还有一些[其他选项](#)为方便开发者使用，普通用户无需关心。

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：



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：

```

1 erl -name node1@127.0.0.1
2 erl -name node2@127.0.0.1
3 erl -name node3@127.0.0.1
4 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：

```

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

EMQX 分布集群设计

EMQX 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。

EMQX 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：

```
1 topic1 -> node1, node2
2 topic2 -> node3
3 topic3 -> node2, node4
```

sh

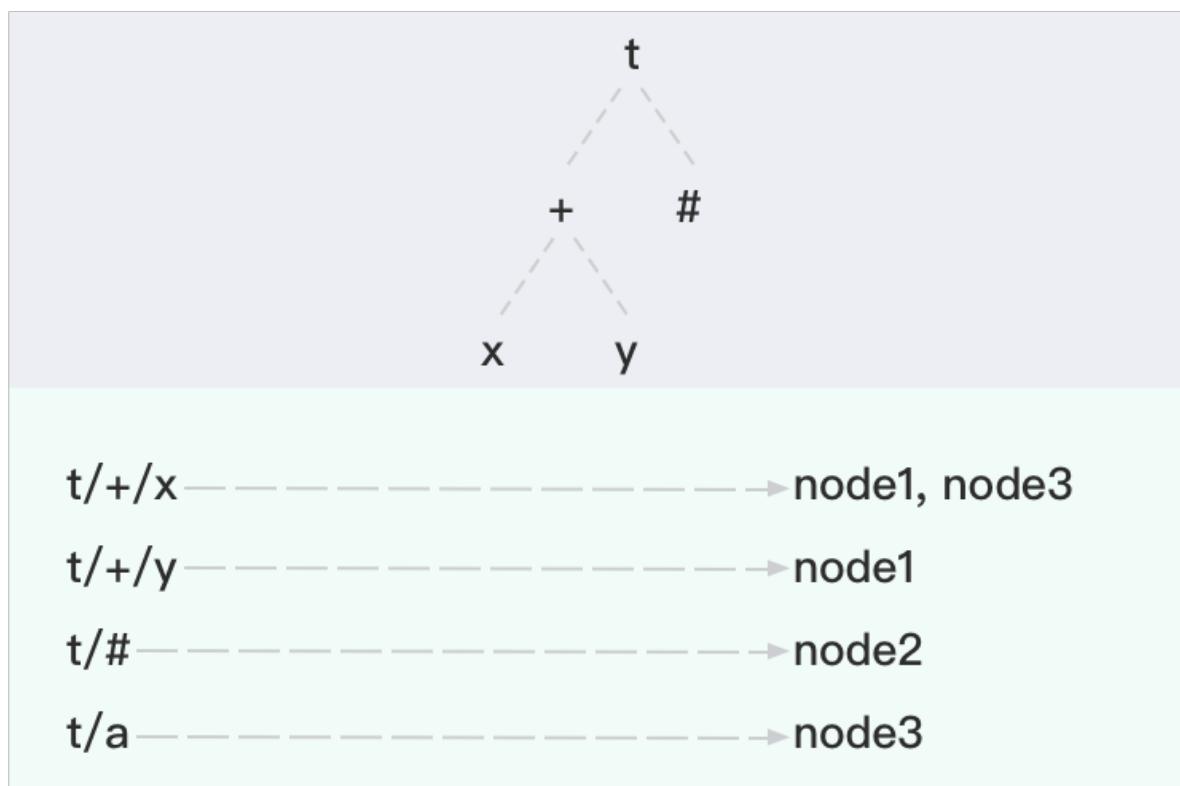
主题树(**Topic Trie**)与路由表(**Route Table**)

EMQX 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。

例如上述主题订阅关系：

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

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



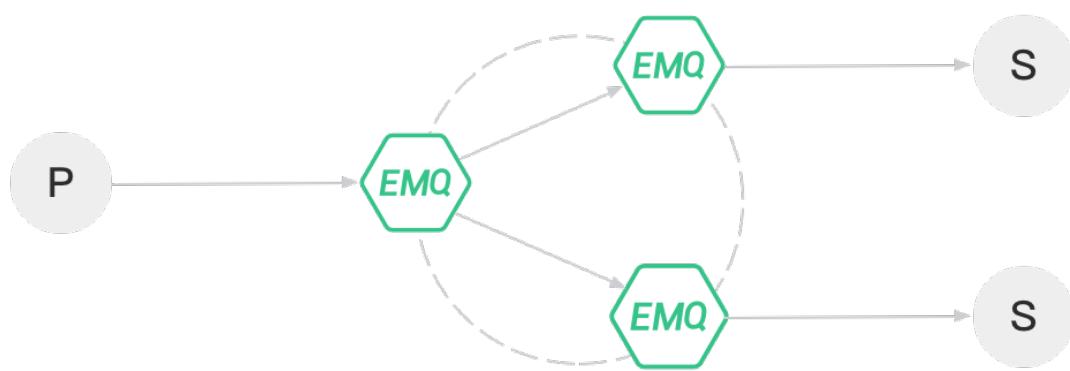
订阅(Subscription)与消息派发

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：

```

1 title: Message Route and Deliver
2
3 client1 -> node1: Publish[t/a]
4     node1 --> node2: Route[t/#]
5         node2 --> client2: Deliver[t/#]
6     node1 --> node3: Route[t/a]
7         node3 --> client3: Deliver[t/a]
```



节点发现与自动集群

EMQX 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**)。

EMQX 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

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 EMQX 集群:

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

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

`emqx/etc/emqx.conf`:

```
1 node.name = emqx@s1.emqx.io
# 或
3 node.name = emqx@192.168.0.10
```

也可通过环境变量:

```
1 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`:

```
1 node.name = emqx@s2.emqx.io
# 或
3 node.name = emqx@192.168.0.20
```

节点加入集群

启动两台节点后, 在 `s2.emqx.io` 上执行:

```
1 $ ./bin/emqx_ctl cluster join emqx@s1.emqx.io
2
3 Join the cluster successfully.
4 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

在任意节点上查询集群状态:

```
1 $ ./bin/emqx_ctl cluster status
2
3 Cluster status: [{running_nodes, ['emqx@s1.emqx.io', 'emqx@s2.emqx.io']}]
```

sh

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：

```
1 $ ./bin/emqx_ctl cluster leave
```

sh

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

```
1 $ ./bin/emqx_ctl cluster force-leave emqx@s2.emqx.io
```

sh

单机伪分布式

对于只有个人电脑或者一台服务器的用户来说，可以使用伪分布式集群。请注意，我们若要在单机上启动两个或多个 **emqx** 实例，为避免端口冲突，我们需要对其它节点的监听端口做出调整。

基本思路是复制一份 **emqx** 文件夹然后命名为 **emqx2**，将原先所有 **emqx** 节点监听的端口 **port** 加上一个偏移 **offset** 作为新的 **emqx2** 节点的监听端口。例如，将原先 **emqx** 的**MQTT/TCP** 监听端口由默认的 **1883** 改为了 **2883** 作为 **emqx2** 的 **MQTT/TCP** 监听端口。完成以上操作的自动化脚本可以参照 [集群脚本](#)，具体配置请参见 [配置说明](#) 与 [配置项](#)。

防火墙设置

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**。

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**。

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：

```
1 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, `Offset` 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。

The Cluster RPC 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。

目录结构

不同安装方式得到的 **EMQX** 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	Homebrew(MacOS)安装
可执行文件目录	./bin	/usr/lib/emqx/bin	/usr/local/bin
数据文件	./data	/var/lib/emqx/data	/usr/local/Cellar/emqx/*/data
Erlang 虚拟机文件	./erts-*	/usr/lib/emqx/erts-*	/usr/local/Cellar/emqx/*/erts-
配置文件目录	./etc	/etc/emqx/etc	/usr/local/Cellar/emqx/*/etc
依赖项目目录	./lib	/usr/lib/emqx/lib	/usr/local/Cellar/emqx/*/lib
日志文件	./log	/var/log/emqx	/usr/local/Cellar/emqx/*/log
启动相关的脚本、 schema 文件	./releases	/usr/lib/emqx/releases	/usr/local/Cellar/emqx/*/releases

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

bin 目录

emqx、**emqx.cmd**

EMQX 的可执行文件，具体使用可以查看 [基本命令](#)。

emqx_ctl、**emqx_ctl.cmd**

EMQX 管理命令的可执行文件，具体使用可以查看 [管理命令 CLI](#)。

etc 目录

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

```
1 $ cat loaded_plugins  
2 {emqx_management, true}.  
3 {emqx_dashboard, true}.  
4 {emqx_schema_registry, true}.  
5 {emqx_rule_engine, true}.
```

sh

mnesia

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。

EMQX 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、**Dashboard** 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 **EMQX** 丢失所有业务数据。

可以通过 `emqx_ctl mnesia` 命令查询 **EMQX** 中 **Mnesia** 数据库的系统信息, 具体请查看 [管理命令 CLI](#)。

log 目录

`emqx.log.*`

EMQX 运行时产生的日志文件, 具体请查看 [日志与追踪](#)。

`crash.dump`

EMQX 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 **EMQX** 时, 控制台日志的副本文件。

配置说明

简介

EMQX 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
<code>etc/emqx.conf</code>	EMQX 基础配置文件
<code>etc/cluster.conf</code>	EMQX 集群相关配置文件
<code>etc/rpc.conf</code>	EMQX 远程调用配置文件
<code>etc/logger.conf</code>	EMQX 日志配置文件
<code>etc/zones.conf</code>	EMQX Zone 配置文件
<code>etc/listeners.conf</code>	EMQX 监听端口配置文件
<code>etc/sys_mon.conf</code>	EMQX 告警监控配置文件
<code>etc/acl.conf</code>	EMQX 默认 ACL 规则配置文件
<code>etc/plugins/*.conf</code>	EMQX 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](#)。

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` , `float` , `percent` , `enum` , `ip` , `string` , `atom` , `flag` , `duration` **and** `bytesize`
- 任何以 `#` 开头的行均被视为注释

示例：

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

数据类型

`integer`

整型数据。

`float`

浮点型数据。

`percent`

以 `%` 结尾的百分比数据，最终会被转换为 `float` 类型。

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](#)。

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`。

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。

Yes!

```
1 dir = tmp
```

sh

No!!!

```
1 dir = "tmp"
```

sh

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。

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。

duration

`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。

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。

默认配置

在 **EMQX** 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。

Zone & Listener

EMQX 提供了非常多的配置项，并支持全局配置和局部配置。例如，**EMQX** 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，**EMQX** 就通过 **Zone** 与 **Listener** 为用户提供了这种可能。

Listener

Listener 主要用于配置不同协议的监听端口和相关参数，**EMQX** 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)

EMQX 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：

```
1    listener.tcp.example = 12345
```

sh

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。

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**。

配置更新

配置项会在 **EMQX Broker** 与扩展插件被启动时读取并载入, **EMQX 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。

日志与追踪

控制日志输出

EMQX 支持将日志输出到控制台或者日志文件，或者同时使用两者。可在 `emqx.conf` 中配置：

```
1 log.to = file
```

`log.to` 的默认值为 `file`，它有以下可选值：

- **off**: 完全关闭日志功能
- **file**: 仅将日志输出到文件
- **console**: 仅将日志输出到标准输出(**emqx** 控制台)
- **both**: 同时将日志输出到文件和标准输出(**emqx** 控制台)

从 **4.3.0** 版本开始，如果使用 **Docker** 部署 **EMQX**，默认只能通过 `docker logs` 命令查看 **EMQX** 日志。如需继续按日志文件的方式查看，可以在启动容器时将环境变量 `EMQX_LOG_TO` 设置为 `file` 或者 `both`。

日志级别

EMQX 的日志分 **8** 个等级 ([RFC 5424](#))，由低到高分别为：

```
1 debug < info < notice < warning < error < critical < alert < emergency
```

EMQX 的默认日志级别为 **warning**，可在 `emqx.conf` 中修改：

```
1 log.level = warning
```

此配置将所有 **log handler** 的配置设置为 **warning**。

日志文件和日志滚动

EMQX 的默认日志文件目录在 `./log` (**zip**包解压安装) 或者 `/var/log/emqx` (**二进制**包安装)。可在 `emqx.conf` 中配置：

```
1 log.dir = log
```

在文件日志启用的情况下 (**log.to = file** 或 **both**)，日志目录下会有如下几种文件：

- **emqx.log.N**: 以 `emqx.log` 为前缀的文件为日志文件，包含了 **EMQX** 的所有日志消息。比如 `emqx.log.1`, `emqx.log.2` ...
- **emqx.log.siz** 和 **emqx.log.idx**: 用于记录日志滚动信息的系统文件。

- **run_erl.log:** 以 `emqx start` 方式后台启动 **EMQX** 时，用于记录启动信息的系统文件。
- **erlang.log.N:** 以 `erlang.log` 为前缀的文件为日志文件，是以 `emqx start` 方式后台启动 **EMQX** 时，控制台日志的副本文件。比如 `erlang.log.1`, `erlang.log.2` ...

可在 `emqx.conf` 中修改日志文件的前缀，默认为 `emqx.log`：

```
1 log.file = emqx.log
```

sh

EMQX 默认在单日志文件超过 **10MB** 的情况下，滚动日志文件，最多可有 **5** 个日志文件：第 **1** 个日志文件为 `emqx.log.1`，第 **2** 个为 `emqx.log.2`，并以此类推。当最后一个日志文件也写满 **10MB** 的时候，将从序号最小的日志的文件开始覆盖。文件大小限制和最大日志文件个数可在 `emqx.conf` 中修改：

```
1 log.rotation.size = 10MB
2 log.rotation.count = 5
```

sh

针对日志级别输出日志文件

如果想把大于或等于某个级别的日志写入到单独的文件，可以在 `emqx.conf` 中配置 `log.<level>.file`：

将 `info` 及 `info` 以上的日志单独输出到 `info.log.N` 文件中：

```
1 log.info.file = info.log
```

sh

将 `error` 及 `error` 以上的日志单独输出到 `error.log.N` 文件中

```
1 log.error.file = error.log
```

sh

日志格式

可在 `emqx.conf` 中修改单个日志消息的最大字符长度，如长度超过限制则截断日志消息并用 `...` 填充。默认不限制长度：

将单个日志消息的最大字符长度设置为 **8192**：

```
1 log.chars_limit = 8192
```

sh

日志消息的格式为(各个字段之间用空格分隔)：

date time level client_info module_info msg

- **date:** 当地时间的日期。格式为：**YYYY-MM-DD**
- **time:** 当地时间，精确到毫秒。格式为：**hh:mm:ss.ms**
- **level:** 日志级别，使用中括号包裹。格式为：**[Level]**
- **client_info:** 可选字段，仅当此日志消息与某个客户端相关时存在。其格式为：**ClientId@Peername** 或 **ClientId** 或 **Peername**
- **module_info:** 可选字段，仅当此日志消息与某个模块相关时存在。其格式为：**[Module Info]**

- **msg:** 日志消息内容。格式任意，可包含空格。

日志消息举例 1:

```
1  2020-02-18 16:10:03.872 [debug] <<"mqttjs_9e49354bb3">>@127.0.0.1:57105 [MQTT/WS] SEND CONNA
CK(Q0, R0, D0, AckFlags=0, ReasonCode=0) sh
```

此日志消息里各个字段分别为：

- **date:** 2020-02-18
- **time:** 16:10:03.872
- **level:** [debug]
- **client_info:** <<"mqttjs_9e49354bb3">>@127.0.0.1:57105
- **module_info:** [MQTT/WS]
- **msg:** SEND CONNACK(Q0, R0, D0, AckFlags=0, ReasonCode=0)

日志消息举例 2:

```
1  2020-02-18 16:10:08.474 [warning] [Alarm Handler] New Alarm: system_memory_high_watermark, Al
arm Info: [] sh
```

此日志消息里各个字段分别为：

- **date:** 2020-02-18
- **time:** 16:10:08.474
- **level:** [warning]
- **module_info:** [Alarm Handler]
- **msg:** New Alarm: system_memory_high_watermark, Alarm Info: []

注意此日志消息中，**client_info** 字段不存在。

日志级别和 log handlers

EMQX 使用了分层的日志系统，在日志级别上，包括全局日志级别 (**primary log level**)、以及各 **log handler** 的日志级别。

```
1      [Primary Level]      -- global log level and filters
2          / \
3      [Handler 1]  [Handler 2] -- log levels and filters at each handler sh
```

log handler 是负责日志处理和输出的工作进程，它由 **log handler id** 唯一标识，并负有如下任务：

- 接收什么级别的日志
- 如何过滤日志消息
- 将日志输出到什么地方

我们来看一下 **emqx** 默认安装的 **log handlers**:

```

1 $ emqx_ctl log handlers list
2
3 LogHandler(id=ssl_handler, level=debug, destination=console, status=started)
4 LogHandler(id=file, level=warning, destination=log/emqx.log, status=started)
5 LogHandler(id=default, level=warning, destination=console, status=started)

```

- **file**: 负责输出到日志文件的 **log handler**。它没有设置特殊过滤条件，即所有日志消息只要级别满足要求就输出。输出目的地为日志文件。
- **default**: 负责输出到控制台的 **log handler**。它没有设置特殊过滤条件，即所有日志消息只要级别满足要求就输出。输出目的地为控制台。
- **ssl_handler**: **ssl** 的 **log handler**。它的过滤条件设置为当日志是来自 **ssl** 模块时输出。输出目的地为控制台。

日志消息输出前，首先检查消息是否高于 **primary log level**，日志消息通过检查后流入各 **log handler**，再检查各 **handler** 的日志级别，如果日志消息也高于 **handler level**，则由对应的 **handler** 执行相应的过滤条件，过滤条件通过则输出。

设想一个场景，假设 **primary log level** 设置为 **info**，**log handler default** (负责输出到控制台) 的级别设置为 **debug**，**log handler file** (负责输出到文件) 的级别设置为 **warning**：

- 虽然 **console** 日志是 **debug** 级别，但此时 **console** 日志只能输出 **info** 以及 **info** 以上的消息，因为经过 **primary level** 过滤之后，流到 **default** 和 **file** 的日志只剩下 **info** 及以上的级别；
- **emqx.log.N** 文件里面，包含了 **warning** 以及 **warning** 以上的日志消息。

在 [日志级别](#) 章节中提到的 **log.level** 是修改了全局的日志级别。这包括 **primary log level** 和各个 **handlers** 的日志级别，都设置为了同一个值。

Primary Log Level 相当于一个自来水管道系统的总开关，一旦关闭则各个分支管道都不再有水流通过。这个机制保证了日志系统的高性能运作。

运行时修改日志级别

你可以使用 **EMQX** 的命令行工具 **emqx_ctl** 在运行时修改 **emqx** 的日志级别：

修改全局日志级别：

例如，将 **primary log level** 以及所有 **log handlers** 的级别设置为 **debug**：

```

1 $ emqx_ctl log set-level debug

```

修改主日志级别：

例如，将 **primary log level** 设置为 **debug**：

```

1 $ emqx_ctl log primary-level debug

```

修改某个 **log handler** 的日志级别:

例如, 将 **log handler** `file` 设置为 **debug**:

```
1 $ emqx_ctl log handlers set-level file debug
```

sh

停止某个 **log handler**:

例如, 为了让日志不再输出到 **console**, 可以停止 **log handler** `default`:

```
1 $ emqx_ctl log handlers stop default
```

sh

启动某个已经停止的 **log handler**:

例如, 启动上面已停止的 **log handler** `default`:

```
1 $ emqx_ctl log handlers start default
```

sh

日志追踪

EMQX 支持针对 **ClientID** 或 **Topic** 过滤日志并输出到文件。在使用日志追踪功能之前, 必须将 **primary log level** 设置为 **debug**:

```
1 $ emqx_ctl log primary-level debug
```

sh

开启 **ClientID** 日志追踪, 将所有 **ClientID** 为 '**my_client**' 的日志都输出到 **log/my_client.log**:

```
1 $ emqx_ctl log primary-level debug
2   debug
3
4 $ emqx_ctl trace start client my_client log/my_client.log
5   trace clientid my_client successfully
```

sh

开启 **Topic** 日志追踪, 将主题能匹配到 '**t/#**' 的消息发布日志输出到 **log/topic_t.log**:

```
1 $ emqx_ctl log primary-level debug
2   debug
3
4 $ emqx_ctl trace start topic 't/#' log/topic_t.log
5   trace topic t/# successfully
```

sh

提示

即使 `emqx.conf` 中, `log.level` 设置为 **error**, 使用消息追踪功能仍然能够打印出某 **client** 或 **topic** 的 **debug** 级别的信息。这在生产环境中非常有用。

日志追踪的原理

日志追踪的原理是给 **emqx** 安装一个新的 **log handler**, 并设置 **handler** 的过滤条件。在 [日志级别和 log handlers](#) 小节, 我们讨论过 **log handler** 的细节。

比如使用如下命令启用 **client** 日志追踪:

```
1 $ emqx_ctl log primary-level debug && emqx_ctl trace start client my_client log/my_client.log
```

然后查询已经开启的追踪:

```
1 $ emqx_ctl trace list
2 Trace(clientid=my_client, level=debug, destination="log/my_client.log", status=started)
```

在后台, **emqx** 会安装一个新的 **log handler**, 并给其指定过滤条件为: 仅当 **ClientID** 为 "my_client" 的时候, 输出日志:

```
1 $ emqx_ctl log handlers list
2 LogHandler(id=trace_clientid_my_client, level=debug, destination=log/my_client.log, status=started)
3 ...
```

这里看到新添加的 **log handler** 的 **id** 为 **trace_clientid_my_client**, 并且 **handler level** 为 **debug**。这就是为什么在 **trace** 之前, 我们必须将 **primary log level** 设置为 **debug**。

如果使用默认的 **primary log level (warning)**, 这个**log handler** 永远不会输出 **warning** 以下的日志消息。

另外, 由于我们是启用了一个新的 **log handler**, 所以我们的日志追踪不受控制台日志和 **emqx.log.N** 文件日志的级别的约束。即使 **log.level = warning**, 我们任然可以追踪到 **my_client** 的 **debug** 级别的日志。

Dashboard

EMQX Dashboard (**EMQX** 管理控制台，以下简称 **Dashboard**) 是 **EMQX** 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，**EMQX** 集群的热配置管理和**MQTT** 连接测试等操作。

基本使用

如果 **EMQX** 安装在本机，则使用浏览器打开地址 <http://127.0.0.1:18083>，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：**EMQX** 告警信息，由资源使用过高、**EMQX** 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。

The screenshot shows the EMQX Dashboard homepage at `localhost:18083/#/monitor`. The top navigation bar includes a back button, forward button, refresh button, and a search bar. On the right side of the top bar, there is a red-bordered box containing a bell icon with a '2' notification, a globe icon, and the user name `admin`.

The left sidebar has a dark theme with a green header bar labeled "EMQ Dashboard". It contains several menu items: 监控 (Monitoring) which is selected and highlighted in green, followed by 客户端 (Client), 主题 (Topic), 规则 (Rule), 告警 (Alert), 插件 (Plugin), 工具 (Tools), and 设置 (Settings). Below these are通用 (General) and 通用 (General) sections.

The main content area is titled "首页" (Home) and displays four key metrics in cards:

- 消息发出 (Message Sent): 0 条/秒 (Current sending rate)
- 消息流入 (Message Received): 0 条/秒 (Current receiving rate)
- 订阅数 (Subscription Count): 101 (Number of subscriptions)
- 连接数 (Connection Count): 83 (Number of connections)

Below these cards is a section titled "节点数据" (Node Data) for the node `emqx@127.0.0.1`. It shows detailed system information:

版本信息: develop			
系统时间:	2019-12-27 16:35:51	内存:	101.01M / 136.19M
运行时长:	4 days, 5 hours, 13 minutes, 11 seconds	最大文件句柄:	1024
OTP Release:	R22/10.5.6	Erlang 进程:	478 / 2097152
节点状态:	● 运行中	CPU 负载:	5.11 / 6.76 / 6.94
连接:	93 / 102	主题:	1 / 6
订阅:	101 / 101	保留消息:	4 / 4
共享订阅:	0 / 0		

A green "查看详情" (View Details) button is located at the bottom of this section.

监控

监控页面可查看 **EMQX** 当前集群的运行指标，界面从上到下功能区如下：

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。

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 **EMQX** 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 **EMQX** 视资源使用情况自动向系统申请，并非 **EMQX** 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 **EMQX** 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。

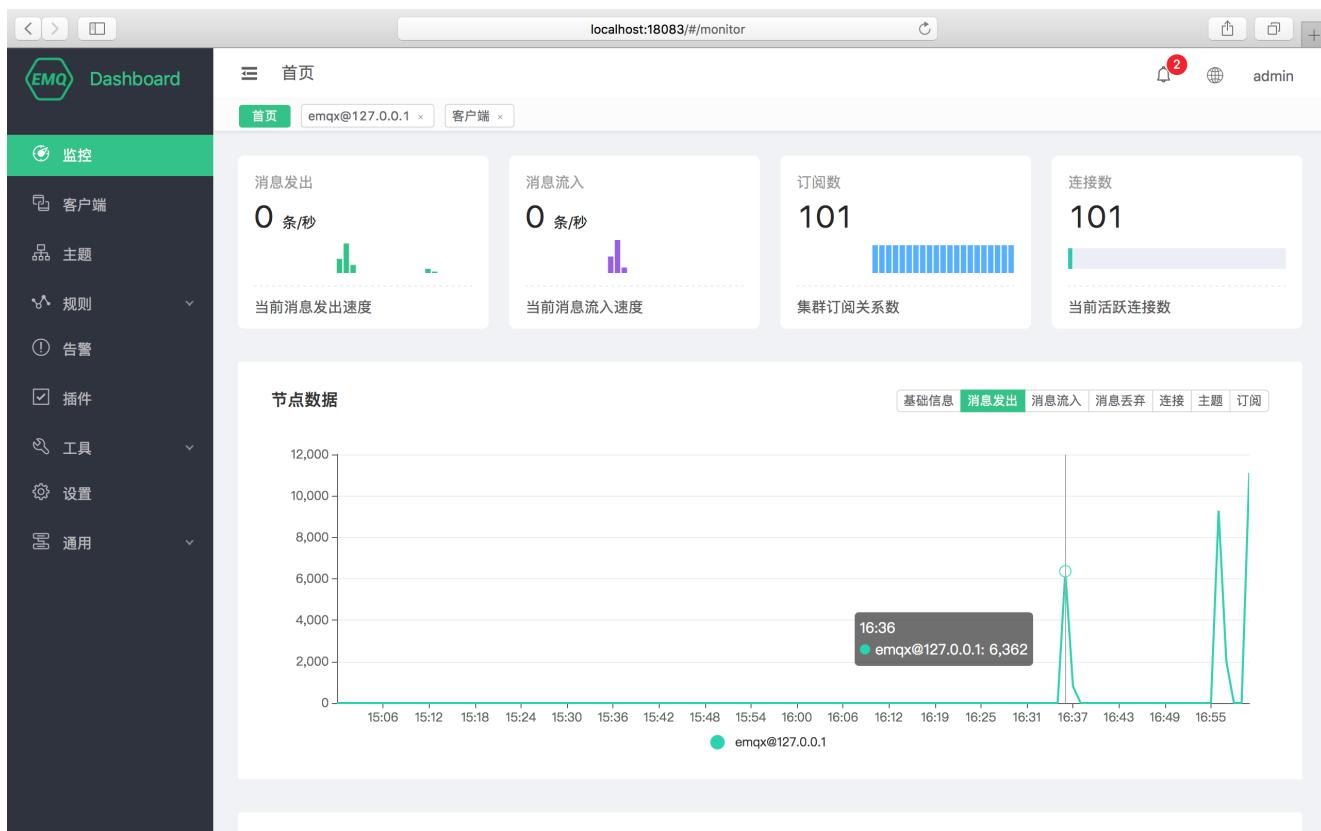
The screenshot shows the EMQX Dashboard interface at the URL `localhost:18083/#/monitor`. The left sidebar is titled "Dashboard" and includes sections for "监控" (Monitoring), "客户端" (Clients), "主题" (Topics), "规则" (Rules), "告警" (Alerts), "插件" (Plugins), "工具" (Tools), and "设置" (Settings). The main content area has a header "首页" (Home) and a sub-header "节点数据" (Node Data). It displays four key metrics in cards: "消息发出" (0条/秒), "消息流入" (5条/秒), "订阅数" (5), and "连接数" (102). Below these cards is a table titled "版本信息: develop" with the following data:

系统时间:	2019-12-27 14:47:31	内存:	100.29M / 128.62M
运行时长:	4 days, 3 hours, 24 minutes, 52 seconds	最大文件句柄:	1024
OTP Release:	R22/10.5.6	Erlang 进程:	481 / 2097152
节点状态:	● 运行中	CPU 负载:	4.16 / 4.94 / 4.78
连接:	102 / 102	主题:	5 / 6
订阅:	5 / 6	保留消息:	4 / 4
共享订阅:	0 / 0		

A green "查看详情" (View Details) button is located at the bottom of the table.

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：



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。

监听器

监听器为当前 **EMQX** 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 使用的 HTTP 协议，默认最大连接数 **512**
 - **http:management**: EMQ 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。

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：

1. 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；
2. 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](#) 或联系 **EMQ** 技术人员进行确认。

The screenshot shows the EMQX monitoring interface for the node `emqx@127.0.0.1`. The left sidebar is titled "Dashboard" and includes sections for Monitoring, Client, Topic, Rule, Alert, Plugin, Tool, and General. The "Monitoring" section is selected. The main content area displays the node's status and configuration.

基础信息

版本信息: develop			
系统时间:	2019-12-27 14:48:48	内存:	102.8MB / 132.7MB
运行时长:	4 days, 3 hours, 26 minutes, 9 seconds	最大文件句柄:	1024
OTP Release:	R22/10.5.6	Erlang 进程:	479 / 2097152
节点状态:	● 运行中	CPU 负载:	2.93 / 4.38 / 4.58 ⓘ
连接:	102 / 102	主题:	5 / 6
订阅:	5 / 6	保留消息:	4 / 4
共享订阅:	0 / 0		

The screenshot shows the EMQX monitoring interface for the node `emqx@127.0.0.1`. The left sidebar is titled "Dashboard" and includes sections for Monitoring, Client, Topic, Rule, Alert, Plugin, Tool, and General. The "Monitoring" section is selected. The main content area displays message statistics.

数据列表

节点的报文信息, 消息统计与流量收发统计

客户端	Delivery	会话
auth.anonymous	dropped.too_large	created
unsubscribe	dropped.no_local	takeovered
disconnected	dropped.expired	terminated
authenticate	dropped.qos0_msg	resumed
check_acl	dropped.queue_full	discarded
subscribe	dropped	
connected		

报文	消息数	流量收发(字节)
received	1819	received 382 25223

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。

The screenshot shows the EMQX Enterprise V4.4 Docs monitor page at localhost:18083/#/monitor. The left sidebar is titled "Dashboard" and includes sections for Monitoring, Clients, Topics, Rules, Alarms, Plugins, Tools, and General settings. The main content area is titled "首页" (Home) and displays various system metrics. Below these metrics is a green button labeled "查看详情" (View Details). Further down is a section titled "License 信息" (License Information) which includes fields for "签发对象" (Issuing Object), "License 使用情况" (License Usage Status), "签发邮箱" (Issuing Email), "签发时间" (Issuing Time), and "到期时间" (Expiration Time). A note at the bottom states: "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。" (Before the certificate expires, EMQ will notify the issuing email, please pay attention to receive information to avoid missing renewal time and affecting business.) A red button labeled "试用版" (Trial Version) is located in the bottom right corner of this section.

客户端

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。

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。

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。

The screenshot shows the client details page for a client named `mqttjs_1739265a`. The client is listed as online. The interface includes tabs for basic information and subscription lists, and a table showing the current subscription `testtopic/1` with QoS 0 and node `emqx@127.0.0.1`.

主题

主题页面包含集群内全部主题（Topic）信息。

The screenshot shows the topics page, displaying a list of topics and their corresponding nodes. The topics listed are: `t/wivviv-mac_bench_sub_10_1368554570`, `t/wivviv-mac_bench_sub_11_1384752087`, `t/wivviv-mac_bench_sub_50_844111593`, `t/wivviv-mac_bench_sub_6_831750842`, `t/wivviv-mac_bench_sub_34_1364326965`, `t/wivviv-mac_bench_sub_49_848779809`, `t/wivviv-mac_bench_sub_13_2991290494`, `t/wivviv-mac_bench_sub_35_3532864058`, and `t/wivviv-mac_bench_sub_89_3528565130`. All topics are associated with the node `emqx@127.0.0.1`.

规则引擎

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 **EMQX** 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。

ID	主题	监控	描述	响应动作
rule:53b7cff1	#	null		保存数据到 Redis

创建规则

EMQX 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：

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

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：

```
1 | 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 **EMQX** 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：

```
1      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。

The screenshot shows the EMQX Dashboard interface for creating a rule. The left sidebar has a 'Rules' section selected under 'Rule Engine'. The main area is titled 'Create Rule' and contains a code editor with the following SQL query:

```

1  SELECT
2    payload.msg as msg
3  FROM
4    "t/#"
5  WHERE
6    msg = 'hello'

```

To the right of the code editor, there are two panes: 'Current event available fields' and 'Rule SQL example'. The 'Current event available fields' pane lists fields like id, clientid, username, payload, peerhost, topic, qos, flags, headers, timestamp, and node. The 'Rule SQL example' pane shows a simplified version of the query:

```

SELECT payload.msg as msg FROM "t/#"
WHERE msg = 'hello'

```

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。

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。

The screenshot shows the EMQX Enterprise V4.4 'Resources' management interface. On the left is a dark sidebar with various navigation options like Dashboard, Monitoring, Clients, Topics, Rules, Rule Engine, Resources (which is highlighted), Decoding, Alarms, Plugins, Tools, Settings, and General. The main content area has a header '首页 / 资源' and a breadcrumb trail: 首页 > 资源. Below the header is a toolbar with links: 首页, 客户端, Websocket, 主题, 规则引擎, 创建规则, and 资源 (highlighted). A green '+ 创建' (Create) button is prominently displayed. The main table lists two resources:

ID	资源类型	备注
resource:c12c718b	Kafka	<button>状态</button> <button>删除</button>
resource:1f5a4e72	Redis 单节点模式	<button>状态</button> <button>删除</button>

Below the table is a section titled '资源状态' (Resource Status) which displays '节点上的资源状态信息' (Resource status information on nodes) with a single entry: 'emqx@127.0.0.1'.

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。

The screenshot shows the 'Create Resource' dialog box overlying the main 'Resources' page. The dialog has a title '创建资源' (Create Resource) and contains the following fields:

- 资源类型:** A dropdown menu currently set to 'Redis Cluster模式' (Redis Cluster Mode).
- 备注:** An input field containing the placeholder '请输入' (Please input).
- Redis Cluster Servers:** A required field containing the value '127.0.0.1:6379,127.0.0.2:'.
- Redis 数据库:** A dropdown menu set to '0'.
- Redis 密码:** An input field containing the placeholder 'Redis 密码' (Redis Password).
- 连接池大小:** A dropdown menu set to '8'.

At the bottom of the dialog are '取消' (Cancel) and '确定' (Confirm) buttons.

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。

模块

模块页面用于查看 EMQX 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。

插件

插件页面用于查看 EMQX 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。

告警

用于展示 **EMQX** 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 **EMQX Control Center** 提供，如有需要请联系 **EMQ** 技术人员获取。

The screenshot shows the EMQX Control Center interface with the URL `localhost:18083/#/alerts/list`. The left sidebar has a green header 'Dashboard' and a 'Alerts' item selected. The main content area shows a table of alerts:

触发节点	告警类型	告警级别	清除时间
emqx@127.0.0.1	{disk_almost_full,"/"}	系统	2020-01-17 23:20:50
emqx@127.0.0.1	system_memory_high_watermark	系统	2020-01-17 23:20:50

At the top right, there are two buttons: '当前告警' (Current Alerts) and '历史告警' (Historical Alerts), with '历史告警' being highlighted.

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。

设置

设置页面提供 **EMQX** 集群的基础参数配置（热配置）与集群配置。

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 **EMQX** 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。

基础设置

- acl_cache_max_size: 32 可以为客户端缓存最大的 ACL 数量
- acl_cache_ttl: 1m ACL 缓存后将被删除的时间
- acl_deny_action: ignore ACL 被拒绝时的处理动作
- acl_nomatch: allow ACL 未命中时允许或拒绝通过验证
- allow_anonymous: true 如果未加载身份验证插件，则默认情况下允许匿名身份验证。建议在生产部署中禁用该选项！
- broker_session_locking_strategy: quorum 会话锁策略。保证集群中 Client ID 在集群中的唯一性。all 表示全集群锁，leader 表示仅主节点锁，quorum 多数节点锁，local 表仅当前节点锁
- broker_shared_dispatch_ack_enabled: false 共享订阅开启内部派发 ACK
- broker_shared_subscription_strategy: random 共享订阅派发策略
- broker_sys_heartbeat: 30s Broker 健康状态发布间隔
- broker_sys_interval: 1m Broker 统计信息发布间隔

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**)

Zone

	external	internal	+
acl_deny_action:	ignore		ACL 被拒绝时的处理动作
await_rel_timeout:	300s		如果等待 pubrel 超时时间，超时将删除 QoS2 消息 (client 发送到 broker)
enable_acl:	on		是否启用 ACL 检查
enable_ban:	on		是否启用白名单检查
enable_flapping_detect:	off		是否开启 flapping detect
enable_stats:	on		启用连接状态统计，会降低部分性能
force_gc_policy:	16000 16MB		MQTT 连接/消息大小 GC 阈值
force_shutdown_policy:	10000 32MB		进程消息队列长度 内存字节
idle_timeout:	15s		MQTT 连接空闲超时

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**)

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)

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。

通用

应用

应用为调用 **REST API** 认证凭证，通过 **REST API** 查询、调整 **EMQX** 集群信息，对客户端连接、插件、**EMQX** 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。

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。

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。

The screenshot shows the EMQX Enterprise Dashboard interface. The left sidebar has a dark theme with green highlights for the 'Dashboard' icon and the 'Blacklist' item under the 'User' section. The main content area is titled '黑名单' (Blacklist). It includes a note about using certificates for client authentication and a table listing a single entry:

禁用值	禁用属性	原因	描述	到期时间
127.0.0.1	peerhost	禁止本地连接		2020-01-18 00:49:43

A red 'Delete' button is visible next to the last column of the table.

认证

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。

EMQX 中的认证指的是当一个客户端连接到 **EMQX** 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。

EMQX 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，**EMQX** 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 **EMQX** 支持的认证方式以及对应插件的配置方法。

认证方式

EMQX 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。

EMQX 支持的认证方式：

内置数据源

- [内置数据库 认证/访问控制](#)

使用配置文件与 **EMQX** 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。

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。

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。

提示

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](#)。

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

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 **EMQX**。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，**EMQX** 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：

```
1 # etc/emqx.conf                                         sh
2
3 ## Value: true | false
4 allow_anonymous = true
```

警告

生产环境中请禁用匿名认证。

密码加盐规则与哈希方法

EMQX 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：

```

1 # etc/plugins/emqx_auth_mysql.conf
2
3 ## 不加盐, 仅做哈希处理
4 auth.mysql.password_hash = sha256
5
6 ## salt 前缀: 使用 sha256 加密 salt + 密码 拼接的字符串
7 auth.mysql.password_hash = salt,sha256
8
9 ## salt 后缀: 使用 sha256 加密 密码 + salt 拼接的字符串
10 auth.mysql.password_hash = sha256,salt
11
12 ## pbkdf2 with macfun iterations dklen
13 ## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha512
14 ## auth.mysql.password_hash = pbkdf2,sha256,1000,20

```

sh

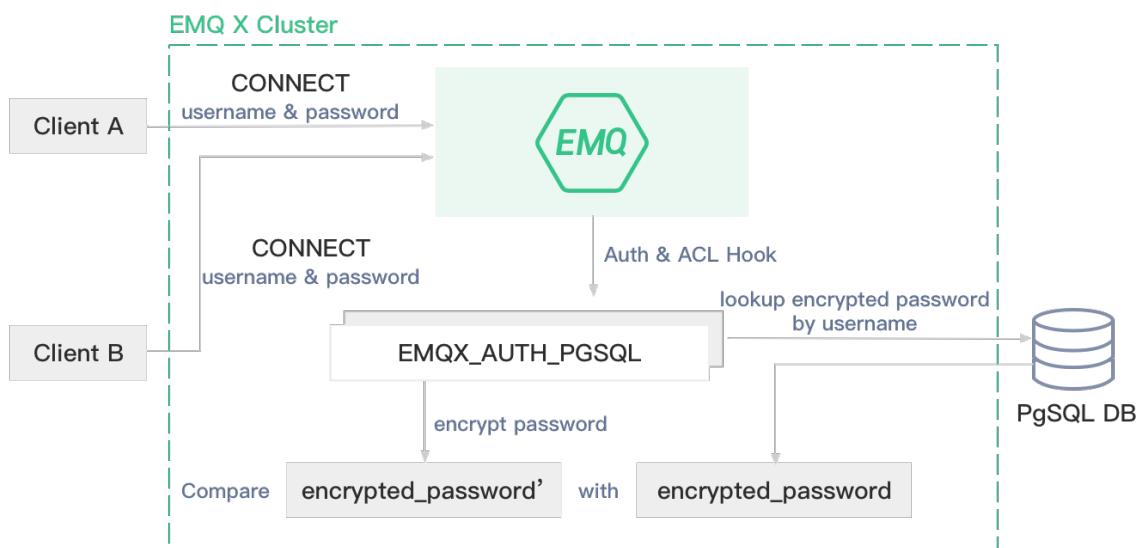
如何生成认证信息

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

EMQX 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：



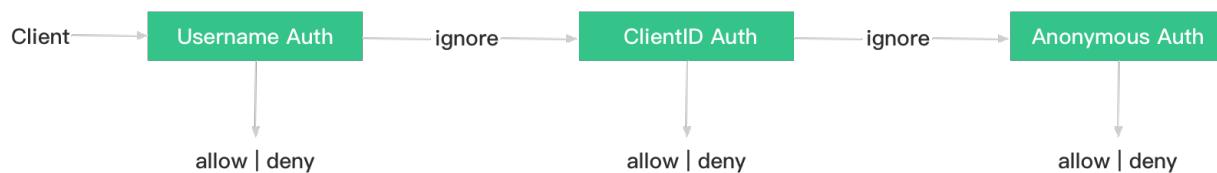
提示

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。

认证链

当同时启用多个认证方式时，**EMQX** 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



提示

同时只启用一个认证插件可以提高客户端身份认证效率。

TLS 认证

MQTT TLS 的默认端口是 **8883**：

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

配置证书和 **CA**：

```
1 listener.ssl.external.keyfile = etc/certs/key.pem
2 listener.ssl.external.certfile = etc/certs/cert.pem
3 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 **EMQX** 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：

```

1    listener.ssl.external.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384,ECDHE-RSA-AES256-GCM-SHA384,ECDH
     E-ECDSA-AES256-SHA384,ECDHE-RSA-AES256-SHA384,ECDHE-ECDSA-DES-CBC3-SHA,ECDH-ECDSA-AES256-GCM-SH
     384,ECDH-RSA-AES256-GCM-SHA384,ECDH-ECDSA-AES256-SHA384,ECDH-RSA-AES256-SHA384,DHE-DSS-AES256-G
     M-SHA384,DHE-DSS-AES256-SHA256,AES256-GCM-SHA384,AES256-SHA256,ECDHE-ECDSA-AES128-GCM-SHA256,EC
     HE-RSA-AES128-GCM-SHA256,ECDHE-ECDSA-AES128-SHA256,ECDHE-RSA-AES128-SHA256,ECDH-ECDSA-AES128-GC
     -SHA256,ECDH-RSA-AES128-GCM-SHA256,ECDH-ECDSA-AES128-SHA256,ECDH-RSA-AES128-SHA256,DHE-DSS-AES1
     8-GCM-SHA256,DHE-DSS-AES128-SHA256,AES128-GCM-SHA256,AES128-SHA256,ECDHE-ECDSA-AES256-SHA,ECDHE
     RSA-AES256-SHA,DHE-DSS-AES256-SHA,ECDH-ECDSA-AES256-SHA,ECDH-RSA-AES256-SHA,AES256-SHA,ECDHE-EC
     SA-AES128-SHA,ECDHE-RSA-AES128-SHA,DHE-DSS-AES128-SHA,ECDH-ECDSA-AES128-SHA,ECDH-RSA-AES128-SHA,AES128-SHA

```

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 :
```

```

1 #listener.ssl.external.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384,...
2 listener.ssl.external.psk_ciphers = PSK-AES128-CBC-SHA,PSK-AES256-CBC-SHA,PSK-3DES-EDE-CBC-SHA
3 ,PSK-RC4-SHA

```

然后启用 `emqx_psk_file` 插件：

```
1 $ 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**：

```

1 client1:1234
2 client2:abcd

```

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。

EMQX 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 **EMQX** 支持的发布订阅 **ACL** 以及对应插件的配置方法。

ACL 插件

EMQX 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。

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。

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。

其他

- [HTTP 认证/访问控制](#)

HTTP ACL 能够实现复杂的 **ACL** 管理。

提示

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

规则详解

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

```

1 ## Allow-Deny Who Pub-Sub Topic
2
3 "允许(Allow) / 拒绝(Deny)" "谁(Who)" "订阅(Subscribe) / 发布(Publish)" "主题列表(Topics)"

```

同时具有多条 **ACL** 规则时，**EMQX** 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：

1. 第一条规则允许客户端发布订阅所有主题
2. 第二条规则禁止全部客户端订阅 `$SYS/#` 与 `#` 主题
3. 第三条规则允许 ip 地址为 `127.0.0.1` 的客户端发布/订阅 `$SYS/#` 与 `#` 主题，为第二条开了特例
4. 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

```

1 {allow, {user, "dashboard"}, subscribe, ["$SYS/#"]}.
2
3 {allow, {ipaddr, "127.0.0.1"}, pubsub, ["$SYS/#", "#"]}.
4
5 {deny, all, subscribe, ["$SYS/#", {eq, "#"}]}.
6
7 {allow, all}.

```

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

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：

```

1 # etc/emqx.conf
2
3 ## ACL 未匹配时默认授权
4 ## Value: allow | deny
5 acl_nomatch = allow

```

配置默认 **ACL** 文件，使用文件定义默认 **ACL** 规则：

```

1 # etc/emqx.conf
2
3 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：

```

1 # etc/emqx.conf
2
3 ## Value: ignore | disconnect
4 acl_deny_action = ignore

```

sh

提示

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。

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 **EMQX** 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

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：

```

1 # etc/emqx.conf
2
3 ## 是否启用
4 enable_acl_cache = on
5
6 ## 单个客户端最大缓存规则数量
7 acl_cache_max_size = 32
8
9 ## 缓存失效时间，超时后缓存将被清除
10 acl_cache_ttl = 1m

```

sh

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：

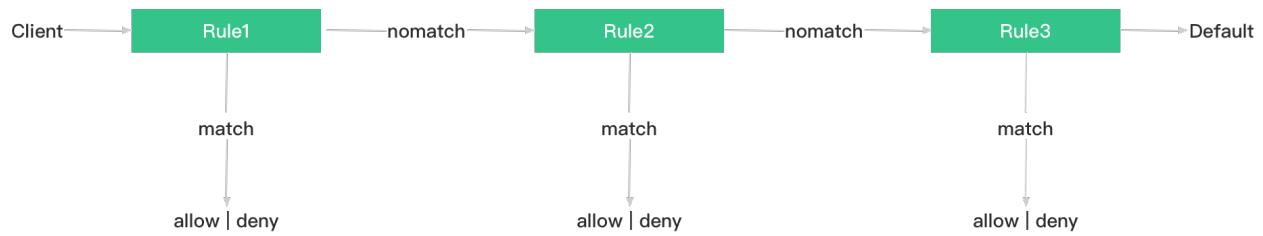
参见 [HTTP API - 清除 ACL 缓存](#)

ACL 鉴权链

当同时启用多个 **ACL** 插件时，**EMQX** 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

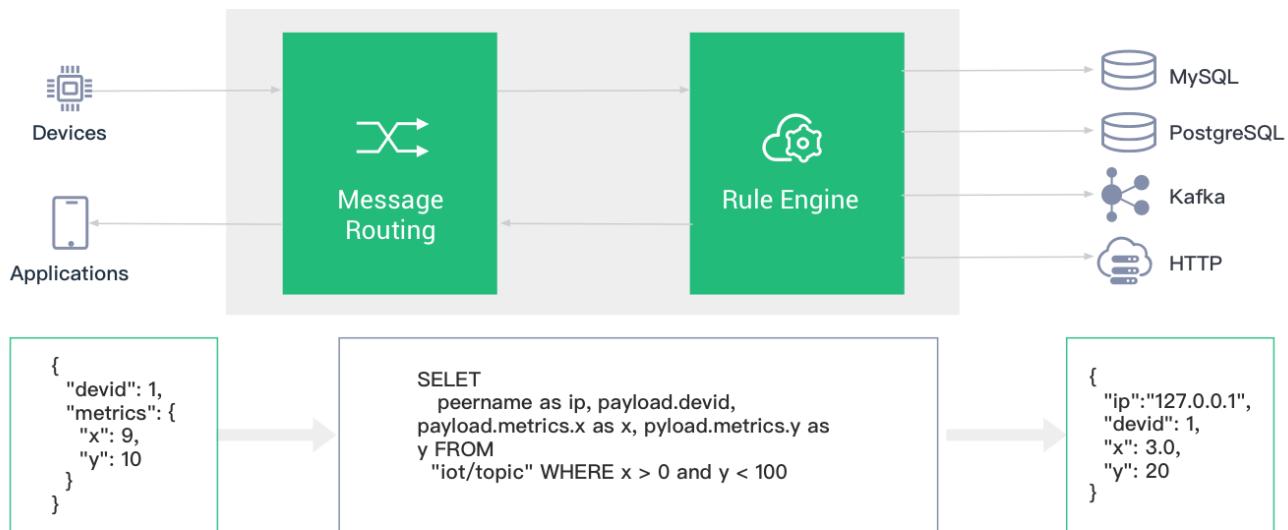


提示

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

规则引擎

EMQX Rule Engine (以下简称规则引擎) 用于配置 EMQX 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 EMQX 的耦合度；也为 EMQX 的私有功能定制提供了一个更优秀的基础架构。



EMQX 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。

提示

适用版本: **EMQX v3.1.0+**

兼容提示: **EMQX 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。

EMQX 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。

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：

```
1 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：

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

事件触发

规则引擎使用 **\$events/** 开头的虚拟主题（事件主题）处理 **EMQX** 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：

```
1 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 [SQL 手册](#)。

规则引擎组成

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：

The screenshot shows the EMQX Dashboard with the 'Rule Engine' selected in the sidebar. The main area displays a rule named 'payload.msg, topic, qos'. The 'SQL' section contains the following query:

```
SELECT payload.msg, topic, qos
FROM "#"
```

The 'Response Actions' section shows two actions:

- Action Type: data_to_webserver**: Sends data to a web server. Success: 0, Failure: 0.
- Action Type: republish**: Republishes messages to another topic. Success: 0, Failure: 0.

使用 **EMQX** 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 **EMQX** 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：

```

1  规则: {
2      SQL 语句,
3      动作列表: [
4          {
5              动作1,
6              动作参数,
7              绑定资源: {
8                  资源配置
9              }
10         },
11         {
12             动作2,
13             动作参数,
14             绑定资源: {
15                 资源配置
16             }
17         }
18     ]
19 }
```

- **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。

提示

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。

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 **EMQX** 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。

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。

1. 在创建规则界面，输入 规则**SQL**，并启用 **SQL** 测试 开关：

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4   "t/#"
5   WHERE
6     qos = 1
```

备注:

SQL 测试: [?](#)

username: u_emqx

topic: t/a

qos: 1

payload:

```
1 {"msg": "hello"}
```

JSON RAW

2. 修改模拟事件的字段，或者使用默认的配置，点击 测试 按钮：

username:	<input type="text" value="u_emqx"/>
topic:	<input type="text" value="t/a"/>
qos:	<input type="text" value="1"/>
payload:	<input "hello"}"="" msg":="" type="text" value="1 {"/> ... <input checked="" type="radio"/> JSON <input type="radio"/> RAW
clientid:	<input type="text" value="c_emqx"/>
<input style="background-color: #28a745; color: white; border-radius: 5px; padding: 5px; margin-bottom: 10px;" type="button" value="SQL 测试"/>	
测试输出:	<div style="border: 1px solid #ccc; padding: 10px; height: 150px;"></div>

3. SQL 处理后的结果将在 测试输出 文本框里展示:

<input style="background-color: #28a745; color: white; border-radius: 5px; padding: 5px; margin-bottom: 10px;" type="button" value="SQL 测试"/>	
测试输出:	<pre>{ "username": "u_emqx", "topic": "t/a", "timestamp": 1587533697725, "qos": 1, "peerhost": "127.0.0.1", "payload": "{\"msg\": \"hello\"}", "node": "emqx@127.0.0.1", "id": "5A3DA7E1F3507F4430000197A0003", "flags": { "sys": true, "event": true }, "clientid": "c_emqx" }</pre>

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：

```
1 ## 3.x 版本
2 ## 需要指定事件名称进行处理
3 SELECT * FROM "t/#" WHERE topic =~ 't/#'
4
5 ## 4.0 及以后版本
6 ## 默认处理 message.publish 事件, FROM 后面直接填写 MQTT 主题
7 ## 上述 SQL 语句等价于:
8 SELECT * FROM 't/#'
9
10 ## 其他事件, FROM 后面填写事件主题
11 SELECT * FROM "$events/message_acked" where topic =~ 't/#'
12 SELECT * FROM "$events/client_connected"
```

提示

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

[下一部分, 规则引擎语法和示例](#)

数据存储

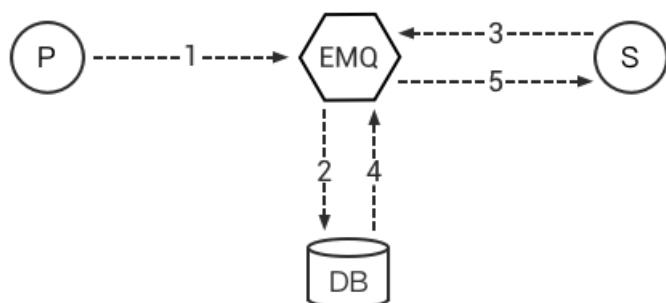
数据存储的主要使用场景包括将客户端上下线状态，订阅主题信息，消息内容，消息抵达后发送消息回执等操作记录到 **Redis**、**MySQL**、**PostgreSQL**、**MongoDB**、**Cassandra** 等各种数据库中。用户也可以通过订阅相关主题的方式来实现类似的功能，但是在企业版中内置了对这些持久化的支持；相比于前者，后者的执行效率更高，也能大大降低开发者的工作量。

提示

数据存储是 **EMQX Enterprise** 专属功能。

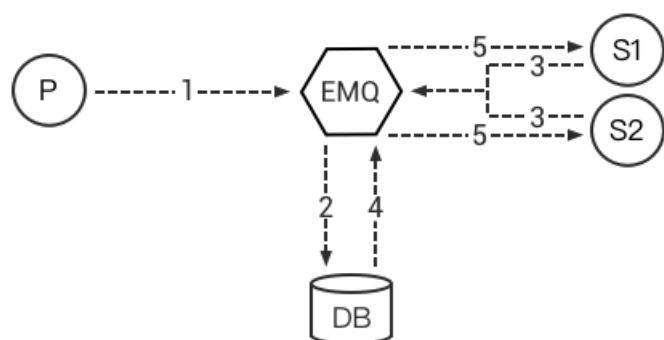
数据存储设计

一对消息存储



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；

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。

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。

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。

存储插件列表

EMQX 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

配置步骤

EMQX 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。

消息桥接

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

桥接是一种连接多个 **EMQX** 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。

```
1 -----  
2 Publisher --> | Node1 | --Bridge Forward--> | Node2 | --Bridge Forward--> | Node3 | --> Subs  
3 criber  
-----  
-----  
-----
```

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 **EMQX**，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 **EMQX** 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。

桥接插件列表

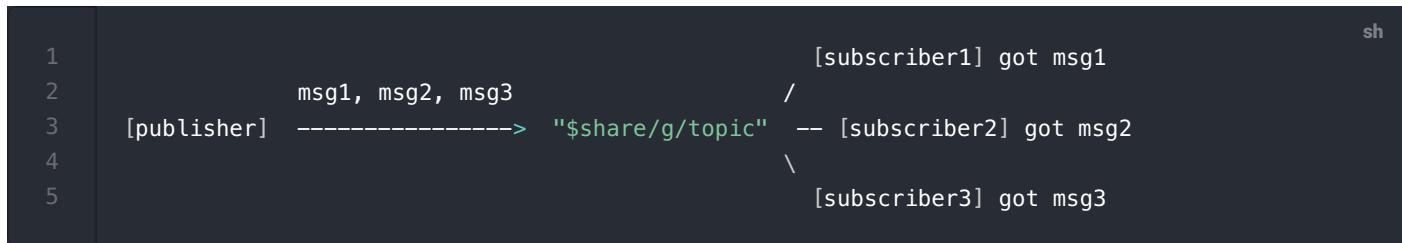
存储插件	配置文件	说明
emqx_bridge_mqtt	emqx_bridge_mqtt.conf	MQTT Broker 消息转发
emqx_bridge_kafka	emqx_bridge_kafka.conf	Kafka 消息队列
emqx_bridge_rabbit	emqx_bridge_rabbit.conf	RabbitMQ 消息队列
emqx_bridge_pulsar	emqx_bridge_pulsar.conf	Pulsar 消息队列
emqx_bridge_rocket	emqx_bridge_rocket.conf	RocketMQ 消息队列

提示

推荐使用 [规则引擎](#) 以实现更灵活的桥接功能。

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。**EMQX** 支持两种格式的共享订阅前缀：

示例	前缀	真实主题名
<code>\$queue/t/1</code>	<code>\$queue/</code>	<code>t/1</code>
<code>\$share/abc/t/1</code>	<code>\$share/abc</code>	<code>t/1</code>

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 **EMQX** 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 **EMQX** 向这个主题发布消息 `msg1` 的时候：

- **EMQX** 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`



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：

```
1                               [s1] got msg1
2           msg1,msg2,msg3      /
3 [emqx] -----> "$queue/topic" - [s2] got msg2
4                               \
5                               [s3] got msg3
```

sh

均衡策略与派发 Ack 配置

EMQX 的共享订阅支持均衡策略与派发 Ack 配置：

```
1 # etc/emqx.conf
2
3 # 均衡策略
4 broker.shared_subscription_strategy = random
5
6 # 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
7 broker.shared_dispatch_ack_enabled = false
```

sh

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

提示

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。

\$SYS 系统主题

EMQX 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：

```
1 $SYS/brokers/emqx@127.0.0.1/version
2 $SYS/brokers/emqx@127.0.0.1/uptime
```

sh

\$SYS 系统消息发布周期配置项：

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

sh

EMQX 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。:::

集群状态信息

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

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， EMQX 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， EMQX 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：

```

1  {
2      "username": "foo",
3      "ts": 1625572213873,
4      "sockport": 1883,
5      "proto_ver": 4,
6      "proto_name": "MQTT",
7      "keepalive": 60,
8      "ipaddress": "127.0.0.1",
9      "expiry_interval": 0,
10     "connected_at": 1625572213873,
11     "connack": 0,
12     "clientid": "emqtt-8348fe27a87976ad4db3",
13     "clean_start": true
14 }
```

sh

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

```

1  {
2      "username": "foo",
3      "ts": 1625572213873,
4      "sockport": 1883,
5      "reason": "tcp_closed",
6      "proto_ver": 4,
7      "proto_name": "MQTT",
8      "ipaddress": "127.0.0.1",
9      "disconnected_at": 1625572213873,
10     "clientid": "emqtt-8348fe27a87976ad4db3"
11 }
```

sh

系统统计 (Statistics)

系统主题前缀: `$/SYS/brokers/${node}/stats/`

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

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

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

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。

收发流量 / 报文 / 消息统计

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

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

MQTT 报文收发统计

主题 (Topic)	说明
<code>packets/received</code>	累计接收 MQTT 报文
<code>packets/sent</code>	累计发送 MQTT 报文
<code>packets/connect</code>	累计接收 MQTT CONNECT 报文
<code>packets/connack</code>	累计发送 MQTT CONNACK 报文
<code>packets/publish/received</code>	累计接收 MQTT PUBLISH 报文
<code>packets/publish/sent</code>	累计发送 MQTT PUBLISH 报文
<code>packets/puback/received</code>	累计接收 MQTT PUBACK 报文
<code>packets/puback/sent</code>	累计发送 MQTT PUBACK 报文
<code>packets/puback/missed</code>	累计丢失 MQTT PUBACK 报文
<code>packets/pubrec/received</code>	累计接收 MQTT PUBREC 报文
<code>packets/pubrec/sent</code>	累计发送 MQTT PUBREC 报文
<code>packets/pubrec/missed</code>	累计丢失 MQTT PUBREC 报文
<code>packets/pubrel/received</code>	累计接收 MQTT PUBREL 报文
<code>packets/pubrel/sent</code>	累计发送 MQTT PUBREL 报文
<code>packets/pubrel/missed</code>	累计丢失 MQTT PUBREL 报文
<code>packets/pubcomp/received</code>	累计接收 MQTT PUBCOMP 报文
<code>packets/pubcomp/sent</code>	累计发送 MQTT PUBCOMP 报文
<code>packets/pubcomp/missed</code>	累计丢失 MQTT PUBCOMP 报文
<code>packets/subscribe</code>	累计接收 MQTT SUBSCRIBE 报文
<code>packets/suback</code>	累计发送 MQTT SUBACK 报文
<code>packets/unsubscribe</code>	累计接收 MQTT UNSUBSCRIBE 报文
<code>packets/unsuback</code>	累计发送 MQTT UNSUBACK 报文
<code>packets/pingreq</code>	累计接收 MQTT PINGREQ 报文
<code>packets/pingresp</code>	累计发送 MQTT PINGRESP 报文
<code>packets/disconnect/received</code>	累计接收 MQTT DISCONNECT 报文
<code>packets/disconnect/sent</code>	累计发送 MQTT DISCONNECT 报文
<code>packets/auth</code>	累计接收 MQTT AUTH 报文

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

Alarms - 系统告警

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

主题 (Topic)	说明
activate	新产生的告警
deactivate	被清除的告警

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

黑名单

EMQX 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](#)。

提示

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，**EMQX** 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。

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

sh

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：

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

sh

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](#)。

WebHook

WebHook 是由 [emqx_web_hook](#) 插件提供的 将 **EMQX** 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 **EMQX** 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：



提示

WebHook 对于事件的处理是单向的，它仅支持将 **EMQX** 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。

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](#)。

触发规则

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



Event 触发事件

目前支持以下事件：

名称	说明	执行时机
<code>client.connect</code>	处理连接报文	服务端收到客户端的连接报文时
<code>client.connack</code>	下发连接应答	服务端准备下发连接应答报文时
<code>client.connected</code>	成功接入	客户端认证完成并成功接入系统后
<code>client.disconnected</code>	连接断开	客户端连接层在准备关闭时
<code>client.subscribe</code>	订阅主题	收到订阅报文后, 执行 <code>client.check_acl</code> 鉴权前
<code>client.unsubscribe</code>	取消订阅	收到取消订阅报文后
<code>session.subscribed</code>	会话订阅主题	完成订阅操作后
<code>session.unsubscribed</code>	会话取消订阅	完成取消订阅操作后
<code>message.publish</code>	消息发布	服务端在发布 (路由) 消息前
<code>message.delivered</code>	消息投递	消息准备投递到客户端前
<code>message.acked</code>	消息回执	服务端在收到客户端发回的消息 ACK 后
<code>message.dropped</code>	消息丢弃	发布出的消息被丢弃后

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。

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:

```
1   web.hook.rule.message.publish.1 = {"action": "on_message_publish", "topic": "a/b/c"}  
2   web.hook.rule.message.publish.2 = {"action": "on_message_publish", "topic": "foo/#"}  
sh
```

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`。

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:

```
1   URL: <url>      # 来自于配置中的 `url` 字段  
2   Method: POST    # 固定为 POST 方法  
3  
4   Body: <JSON>     # Body 为 JSON 格式字符串  
sh
```

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：

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

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
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

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)

client.disconnected

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

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

opts 包含

Key	类型	说明
qos	enum	QoS 等级, 可取 0 1 2

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`**message.publish**

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)

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 "
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)

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 "
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)

分布式集群

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。

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：

```
1 <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：

```
1 erl -name node1@127.0.0.1 -setcookie my_nodes
2 erl -name node2@127.0.0.1 -setcookie my_nodes
3 erl -name node3@127.0.0.1 -setcookie my_nodes
4 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：

```
1 (node1@127.0.0.1) 4> node().
2 'node1@127.0.0.1'
3
4 (node1@127.0.0.1) 4> nodes().
5 []
```

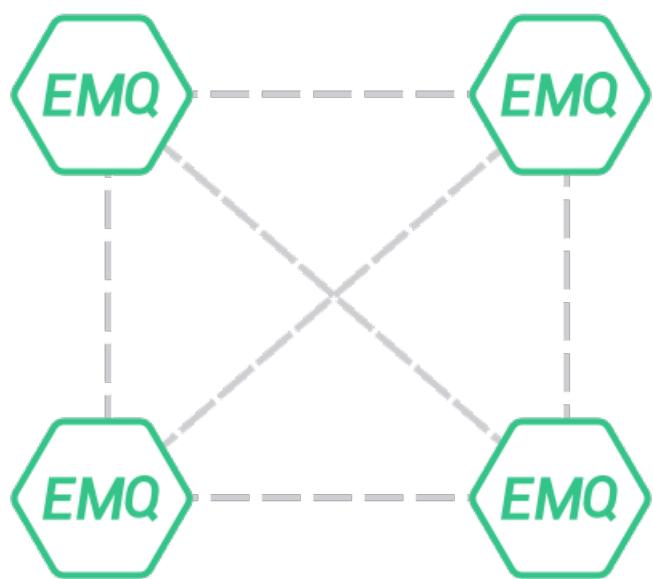
然后我们让 **node1** 发起与其他节点的连接：

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

现在再次可查看已与 **node1** 建立连接的其他节点：

```
1 (node1@127.0.0.1) 4> nodes().
2 ['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：



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

节点间RPC使用TLS

为保障节点间通信的安全性，可以为节点间的**RPC**连接开启**TLS**。**TLS** 可能会影响到节点的**CPU**使用率上升。

1. 创建一个自签名的根证书

```

1 # Create self-signed root CA:
2 openssl req -nodes -x509 -sha256 -days 1825 -newkey rsa:2048 -keyout rootCA.key -out rootCA.pem
   -subj "/O=LocalOrg/CN=LocalOrg-Root-CA"

```

2. 使用第一步的根证书签发节点证书

```

1 # Create a private key:
2 openssl genrsa -out domain.key 2048
3 # Create openssl extfile:
4 cat <<EOF > domain.ext
5 authorityKeyIdentifier=keyid,issuer
6 basicConstraints=CA:FALSE
7 subjectAltName = @alt_names
8 [alt_names]
9 DNS.1 = backplane
10 EOF
11 # Create a CSR:
12 openssl req -key domain.key -new -out domain.csr -subj "/O=LocalOrg"
13 # Sign the CSR with the Root CA:
14 openssl x509 -req -CA rootCA.pem -CAkey rootCA.key -in domain.csr -out domain.pem -days 365 -CAcreateserial -extfile domain.ext

```

请注意，集群中的所有节点必须使用同一个跟证书。

3. 为每个节点，将生成的私钥以及证书文件 `domain.pem`，`domain.key` 和 `rootCA.pem` 放置在 `/var/lib/emqx/ssl`。请保证 `emqx` 用户是这些文件的所有者，并设置权限为 `600`。

4. 如果版本是企业版 **4.4.0**，需要在 `releases/4.4.0/emqx.schema` 末尾增加如下配置

```

1 {mapping, "rpc.default_client_driver", "gen_rpc.default_client_driver",
2 [{default, tcp}, {datatype, {enum, [tcp, ssl]}}]}].

```

5. 在企业版的 `etc/rpc.conf`，或开源版的 `etc/emqx.conf` 中加入如下配置：

```

1 rpc.driver=ssl
2 rpc.default_client_driver=ssl
3 rpc.certfile=/var/lib/emqx/ssl/domain.pem
4 rpccacertfile=/var/lib/emqx/ssl/rootCA.pem
5 rpc.keyfile=/var/lib/emqx/ssl/domain.key
6 rpc.enable_ssl=5369

```

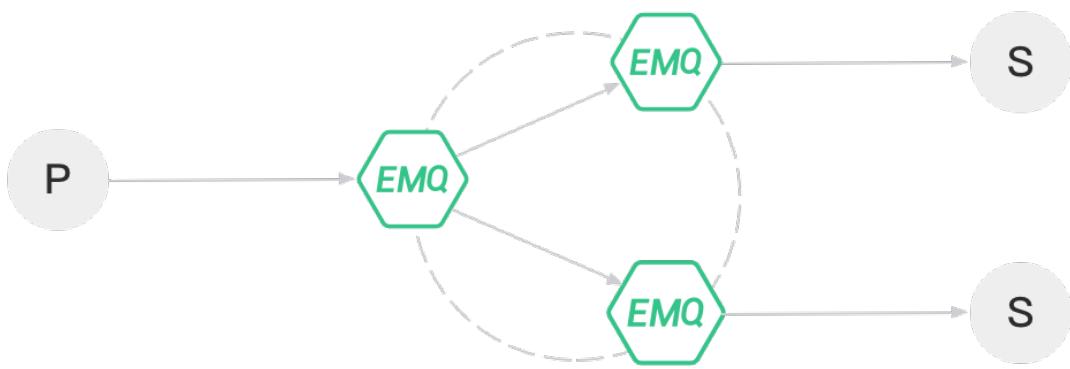
EMQX 集群协议设置

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
<code>cluster.proto_dist</code>	<code>enum</code>	<code>inet_tcp</code>	分布式协议，可选值： - <code>inet_tcp</code> : 使用 TCP IPv4 - <code>inet6_tcp</code> : 使用 TCP IPv6 - <code>inet_tls</code> : 使用 TLS
<code>node.ssl_dist_optfile</code>	文件路径	<code>etc/ssl_dist.conf</code>	当 <code>cluster.proto_dist</code> 选定为 <code>inet_tls</code> 时，需要配置 <code>etc/ssl_dist.conf</code> 文件，指定 TLS 证书等

EMQX 分布式集群设计

EMQX 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，**EMQX** 维护了几个与之相关的数据结构：订阅表，路由表，主题树。

订阅表: 主题 - 订阅者

MQTT 客户端订阅主题时，**EMQX** 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 **EMQX** 节点上，例如：

```

1  node1:
2
3      topic1 -> client1, client2
4      topic2 -> client3
5
6  node2:
7
8      topic1 -> client4

```

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：

```

1  topic1 -> node1, node2
2  topic2 -> node3
3  topic3 -> node2, node4

```

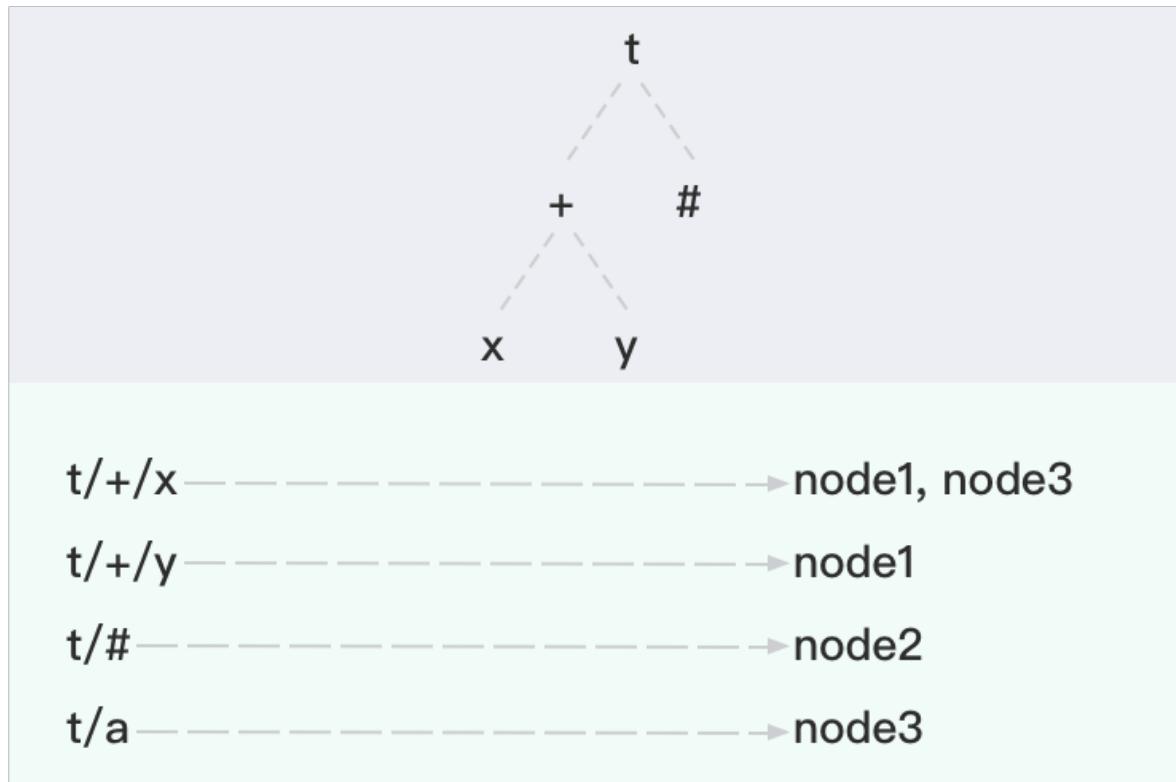
主题树: 带统配符的主题匹配

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

例如上述主题订阅关系：

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，EMQX 中会维护如下主题树 (Topic Trie) 和路由表 (Route Table):



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。

数据分片与共享方式

EMQX 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。

节点发现与自动集群

EMQX 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**)。

EMQX 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

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：

```
1 cluster.discovery = manual
```

sh

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：

```
1 cluster.discovery = static
2 cluster.static.seeds = emqx1@127.0.0.1,emqx2@127.0.0.1
```

sh

基于 mcast 组播自动集群

基于 **UDP** 组播自动发现并创建集群：

```
1 cluster.discovery = mcast
2 cluster.mcast.addr = 239.192.0.1
3 cluster.mcast.ports = 4369,4370
4 cluster.mcast.iface = 0.0.0.0
5 cluster.mcast.ttl = 255
6 cluster.mcast.loop = on
```

sh

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：

```

1  cluster.discovery = dns
2  cluster.dns.name = localhost
3  cluster.dns.app = ekka

```

sh

基于 etcd 自动集群

基于 [etcd](#) 自动发现并创建集群:

```

1  cluster.discovery = etcd
2  cluster.etcd.server = http://127.0.0.1:2379
3  cluster.etcd.prefix = emqcl
4  cluster.etcd.node_ttl = 1m

```

sh

基于 kubernetes 自动集群

[Kubernetes](#) 下自动发现并创建集群:

```

1  cluster.discovery = k8s
2  cluster.k8s.apiserver = http://10.110.111.204:8080
3  cluster.k8s.service_name = ekka
4  cluster.k8s.address_type = ip
5  cluster.k8s.app_name = ekka

```

sh

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

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 EMQX 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)

配置 **emqx@s1.emqx.io** 节点

emqx/etc/emqx.conf:

```

1  node.name = emqx@s1.emqx.io
2  # 或
3  node.name = emqx@192.168.0.10

```

sh

也可通过环境变量:

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

sh

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:

```
1 node.name = emqx@s2.emqx.io
2 # 或
3 node.name = emqx@192.168.0.20
```

sh

节点加入集群

启动两台节点后，在 **s2.emqx.io** 上执行：

```
1 $ ./bin/emqx_ctl cluster join emqx@s1.emqx.io
2
3 Join the cluster successfully.
4 Cluster status: [{running_nodes, ['emqx@s1.emqx.io', 'emqx@s2.emqx.io']}]
```

sh

或者在 **s1.emqx.io** 上执行：

```
1 $ ./bin/emqx_ctl cluster join emqx@s2.emqx.io
2
3 Join the cluster successfully.
4 Cluster status: [{running_nodes, ['emqx@s1.emqx.io', 'emqx@s2.emqx.io']}]
```

sh

在任意节点上查询集群状态：

```
1 $ ./bin/emqx_ctl cluster status
2
3 Cluster status: [{running_nodes, ['emqx@s1.emqx.io', 'emqx@s2.emqx.io']}]
```

sh

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：

```
1 $ ./bin/emqx_ctl cluster leave
```

sh

或在 **s1.emqx.io** 上，从集群删除 **emqx@[s2.emqx.io](#)** 节点：

```
1 $ ./bin/emqx_ctl cluster force-leave emqx@s2.emqx.io
```

sh

单机伪分布式

对于只有个人电脑或者一台服务器的用户来说，可以使用伪分布式集群。请注意，我们若要在单机上启动两个或多个 **emqx** 实例，为避免端口冲突，我们需要对其它节点的监听端口做出调整。

基本思路是复制一份 **emqx** 文件夹然后命名为 **emqx2**，将原先所有 **emqx** 节点监听的端口 **port** 加上一个偏移 **offset** 作为新的 **emqx2** 节点的监听端口。例如，将原先 **emqx** 的**MQTT/TCP** 监听端口由默认的 **1883** 改为了 **2883** 作为 **emqx2** 的 **MQTT/TCP** 监听端口。完成以上操作的自动化脚本可以参照 [集群脚本](#)，具体配置请参见 [配置说明与配置项](#)。

集群脑裂与自动愈合

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

```
1 cluster.autoheal = on
```

sh

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。

集群节点自动清除

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

```
1 cluster.autoclean = 5m
```

sh

防火墙设置

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。

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。

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：

```
1 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，`Offset` 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。

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`。

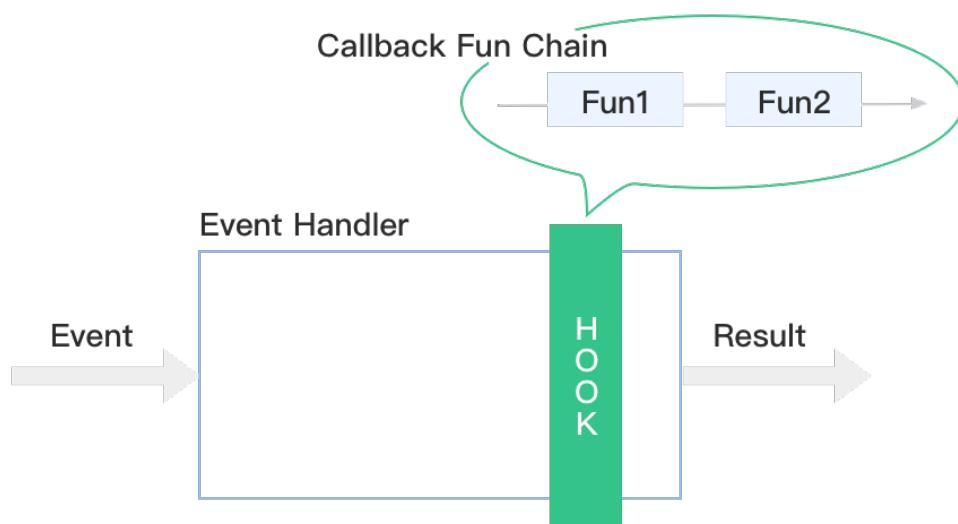
就是说，如果 `emqx.conf` 里配置了节点名：`node.name = emqx@192.168.0.12`，那么监听端口为 `5370`，但对于 `emqx1` (或者 `emqx-1`) 端口就是 `5371`，以此类推。

钩子

定义

钩子(Hooks) 是 EMQX 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。

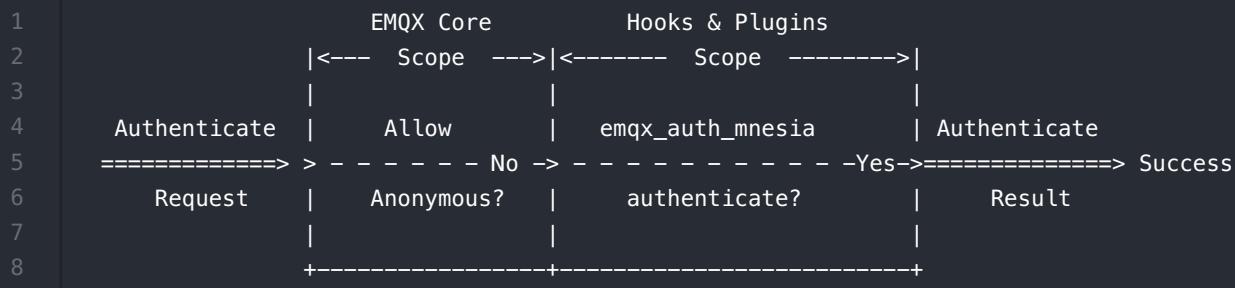
而在这个过程中加入一个可挂载函数的点 (HookPoint)，允许外部插件挂载多个回调函数，形成一个调用链。达到对内部事件处理过程的扩展和修改。

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` 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 **EMQX** 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](#) 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 **EMQX** 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。

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\(Chain-of-Responsibility\)](#) 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](#) 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](#)
- 钩子和 **HTTP** 服务器的应用，参见：[WebHook](#)

挂载点

EMQX 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
<code>client.connect</code>	处理连接报文	服务端收到客户端的连接报文时
<code>client.connack</code>	下发连接应答	服务端准备下发连接应答报文时
<code>client.connected</code>	成功接入	客户端认证完成并成功接入系统后
<code>client.disconnected</code>	连接断开	客户端连接层在准备关闭时
<code>client.authenticate</code>	连接认证	执行完 <code>client.connect</code> 后
<code>client.check_acl</code>	ACL 鉴权	执行 发布/订阅 操作前
<code>client.subscribe</code>	订阅主题	收到订阅报文后, 执行 <code>client.check_acl</code> 鉴权前
<code>client.unsubscribe</code>	取消订阅	收到取消订阅报文后
<code>session.created</code>	会话创建	<code>client.connected</code> 执行完成, 且创建新的会话后
<code>session.subscribed</code>	会话订阅主题	完成订阅操作后
<code>session.unsubscribed</code>	会话取消订阅	完成取消订阅操作后
<code>session.resumed</code>	会话恢复	<code>client.connected</code> 执行完成, 且成功恢复旧的会话信息后
<code>session.discarded</code>	会话被移除	会话由于被移除而终止后
<code>session.takeovered</code>	会话被接管	会话由于被接管而终止后
<code>session.terminated</code>	会话终止	会话由于其他原因被终止后
<code>message.publish</code>	消息发布	服务端在发布(路由)消息前
<code>message.delivered</code>	消息投递	消息准备投递到客户端前
<code>message.acked</code>	消息回执	服务端在收到客户端发回的消息 ACK 后
<code>message.dropped</code>	消息丢弃	发布出的消息被丢弃后
<code>message.slow_subs_stats</code>	订阅者平均消息传输时延过高	QoS1或QoS2消息传输完成时

提示

- 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。

挂载与取消挂载

EMQX 提供了 API 进行钩子的挂载与取消挂载的操作。

挂载：

```
1 %% Name: 钩子的名称（挂载点）如: 'client.authenticate'  
2 %% {Module, Function, Args}: 回调函数的模块、方法、和附加参数  
3 %% Priority: 优先级, 整数; 不提供则默认为 0  
4 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`。

取消挂载：

```
1 %% Name: 钩子的名称（挂载点）如: 'client.authenticate'  
2 %% {Module, Function}: 回调函数的模块、方法  
3 emqx:unhook(Name, {Module, Function}).
```

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](#))

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 Props : MQTT v5.0 订阅报文的 Properties 参数 TopicFilters : 需订阅的主题列表	新的 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](#)

指标监控

EMQX 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。

EMQX 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 **EMQX 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](#)。

提示

EMQX 提供 [emqx_statsd](#) 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](#)。

Metrics & Stats

EMQX 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。**EMQX** 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，**EMQX** 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。

Metrics

字节

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

报文

Name	Data Type	Description
<code>packets.received</code>	<code>Integer</code>	接收的报文数量
<code>packets.sent</code>	<code>Integer</code>	发送的报文数量
<code>packets.connect.received</code>	<code>Integer</code>	接收的 CONNECT 报文数量
<code>packets.connack.auth_error</code>	<code>Integer</code>	发送的原因码为 0x86 和 0x87 的 CONNACK 报文数量
<code>packets.connack.error</code>	<code>Integer</code>	发送的原因码不为 0x00 的 CONNACK 报文数量，此指标的值大于等于 <code>packets.connack.auth_error</code> 的值

Name	Type	Description
packets.connpack.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

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	EMQX 存储的延迟发布的消息数量
messages.delivered	Integer	EMQX 内部转发到订阅进程的消息数量
messages.dropped	Integer	EMQX 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	EMQX 存储的保留消息数量
messages.acked	Integer	已经应答的消息数量

事件

Name	Data Type	Description
<code>client.auth.anonymous</code>	<code>Integer</code>	客户端最终匿名形式登录的次数
<code>client.connect</code>	<code>Integer</code>	<code>client.connect</code> 钩子触发次数
<code>client.authenticate</code>	<code>Integer</code>	<code>client.authenticate</code> 钩子触发次数
<code>client.connack</code>	<code>Integer</code>	<code>client.connack</code> 钩子触发次数
<code>client.connected</code>	<code>Integer</code>	<code>client.connected</code> 钩子触发次数
<code>client.disconnected</code>	<code>Integer</code>	<code>client.disconnected</code> 钩子触发次数
<code>client.check_acl</code>	<code>Integer</code>	<code>client.check_acl</code> 钩子触发次数
<code>client.subscribe</code>	<code>Integer</code>	<code>client.subscribe</code> 钩子触发次数
<code>client.unsubscribe</code>	<code>Integer</code>	<code>client.unsubscribe</code> 钩子触发次数
<code>client.auth.success</code>	<code>Integer</code>	客户端认证成功次数
<code>client.auth.success.anonymous</code>	<code>Integer</code>	匿名认证成功次数
<code>client.auth.failure</code>	<code>Integer</code>	客户端认证失败次数
<code>client.acl.allow</code>	<code>Integer</code>	客户端 ACL 校验通过次数
<code>client.acl.deny</code>	<code>Integer</code>	客户端 ACL 校验失败次数
<code>client.acl.cache_hit</code>	<code>Integer</code>	ACL 校验缓存命中次数
<code>session.created</code>	<code>Integer</code>	<code>session.created</code> 钩子触发次数
<code>session.discarded</code>	<code>Integer</code>	<code>session.discarded</code> 钩子触发次数
<code>session.resumed</code>	<code>Integer</code>	<code>session.resumed</code> 钩子触发次数
<code>session.takeovered</code>	<code>Integer</code>	<code>session.takeovered</code> 钩子触发次数
<code>session.terminated</code>	<code>Integer</code>	<code>session.terminated</code> 钩子触发次数

Stats

Name	Data Type	Description
<code>connections.count</code>	<code>Integer</code>	当前连接数量
<code>connections.max</code>	<code>Integer</code>	连接数量的历史最大值
<code>channels.count</code>	<code>Integer</code>	即 <code>sessions.count</code>
<code>channels.max</code>	<code>Integer</code>	即 <code>session.max</code>
<code>sessions.count</code>	<code>Integer</code>	当前会话数量
<code>sessions.max</code>	<code>Integer</code>	会话数量的历史最大值
<code>topics.count</code>	<code>Integer</code>	当前主题数量
<code>topics.max</code>	<code>Integer</code>	主题数量的历史最大值
<code>suboptions.count</code>	<code>Integer</code>	即 <code>subscriptions.count</code>
<code>suboptions.max</code>	<code>Integer</code>	即 <code>subscriptions.max</code>
<code>subscribers.count</code>	<code>Integer</code>	当前订阅者数量
<code>subscribers.max</code>	<code>Integer</code>	订阅者数量的历史最大值
<code>subscriptions.count</code>	<code>Integer</code>	当前订阅数量，包含共享订阅
<code>subscriptions.max</code>	<code>Integer</code>	订阅数量的历史最大值
<code>subscriptions.shared.count</code>	<code>Integer</code>	当前共享订阅数量
<code>subscriptions.shared.max</code>	<code>Integer</code>	共享订阅数量的历史最大值
<code>routes.count</code>	<code>Integer</code>	当前路由数量
<code>routes.max</code>	<code>Integer</code>	路由数量的历史最大值
<code>retained.count</code>	<code>Integer</code>	当前保留消息数量
<code>retained.max</code>	<code>Integer</code>	保留消息的历史最大值

速率限制

EMQX 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：

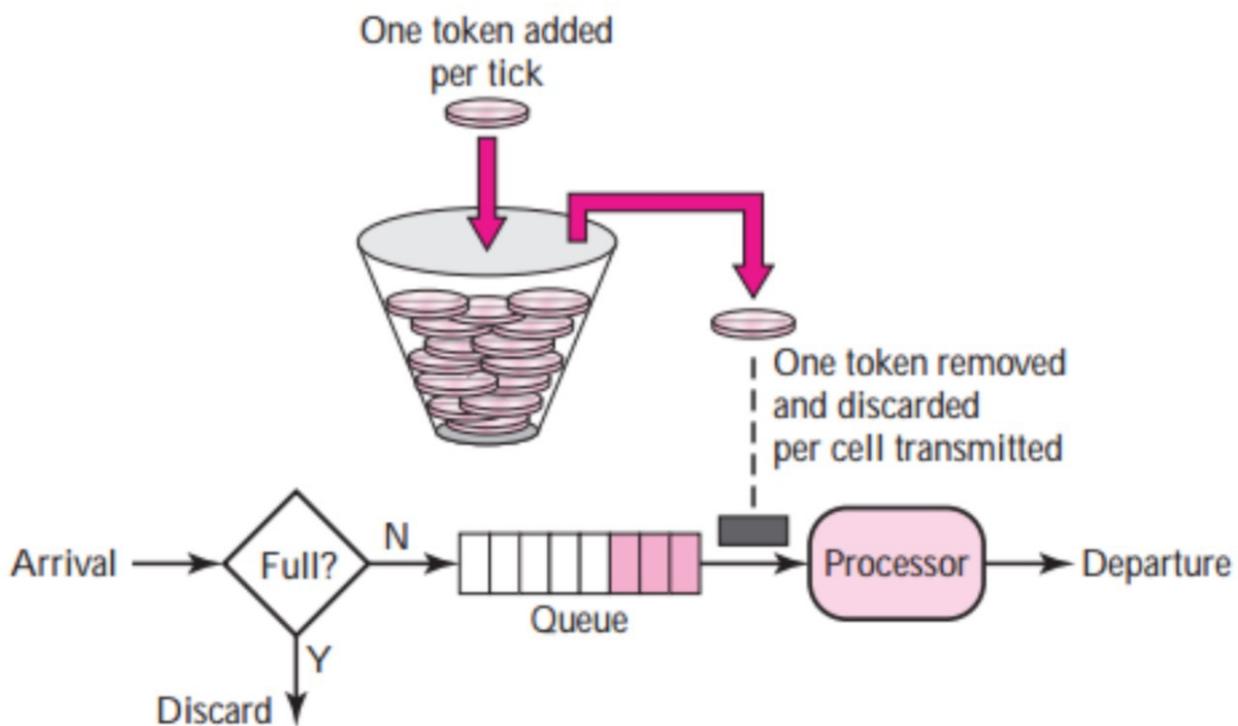
配置项	类型	默认值	描述
<code>listener.tcp.external.max_conn_rate</code>	<code>Number</code>	<code>1000</code>	本节点上允许的最大连接速率(<code>conn/s</code>)
<code>zone.external.rate_limit.conn_messages_in</code>	<code>Number,Duration</code>	无限制	单连接上允许的最大发布速率(<code>msg/s</code>)
<code>zone.external.rate_limit.conn_bytes_in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

- `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，**EMQX** 目前没有提供全局的消息速率限制。

速率限制原理

EMQX 使用[令牌桶 \(Token Bucket\)](#) 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。如果令牌不够则阻塞，等待令牌的生成。

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 $M = b + r$

令牌桶算法在 EMQX 中的应用

当使用如下配置做报文速率限制的时候：

```
1 zone.external.rate_limit.conn_bytes_in = 100KB,10s
```

EMQX 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，EMQX 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。**EMQX** 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
<code>listener.tcp.external.active_n</code>	<code>Number</code>	<code>100</code>	<code>emqx</code> 每次从 <code>TCP</code> 栈读取多少条消息

飞行窗口和消息队列

简介

为了提高消息吞吐效率和减少网络波动带来的影响，**EMQX** 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，**EMQX** 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。

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 **EMQX** 中的 **Inflight Window** 机制如出一辙，只是在 **MQTT v5.0** 协议发布前，**EMQX** 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。

配置项

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

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

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；**EMQX** 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
<code>retry_interval</code>	<code>duration</code>	-	<code>30s</code>	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 **EMQX** 在处理 **MQTT** 协议的重传的细节见以下内容。

协议规范与设计

重传的对象

首先，在了解 **EMQX** 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](#) 和 [MQTTv3.1.1 - QoS 2: Exactly once delivery](#)。

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。

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：

```

1          PUBLISH
2 #1 Sender -----> Receiver      (*)
3          PUBACK
4 #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

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：

```

1          PUBLISH
2 #1 Sender -----> Receiver      (*)
3          PUBREC
4 #2 Sender <----- Receiver
5          PUBREL
6 #3 Sender -----> Receiver      (*)
7          PUBCOM
8 #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 **EMQX** 作为 **PUBLISH** 消息的接收端时，它不需要重发操作

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. **EMQX** 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. **EMQX** 作为接收端时，发送端重发的消息时：
 - 如 **QoS 1**，**EMQX** 则直接回复 **PUBACK** 进行应答；
 - 如 **QoS 2**，**EMQX** 则会释放，存储在 **最大接收消息队列** 中的 **PUBLISH** 或者 **PUBREL** 报文。

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，**EMQX** 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 #1 [4,3,2,1 || ] -----> []
2 #2 [4,3 || 2, 1] -----> [1, 2]
3 #3 [4 || 3, 2] -----> [1, 2, 3]
4 #4 [4 || 3, 2] -----> [1, 2, 3, 2, 3]
5 #5 [ || 4] -----> [1, 2, 3, 2, 3, 4]
6 #6 [ || ] -----> [1, 2, 3, 2, 3, 4]

```

流程共 **6** 个步骤；左边表示 **EMQX** 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 **EMQX** 将这个主题认为是 `有序的主题 (Ordered Topic)` 参见：[MQTTv3.1.1 - Message ordering](#)。

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。

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
<code>mqueue_store_qos0</code>	<code>bool</code>	<code>true</code> , <code>false</code>	<code>true</code>	是否将 QoS 0 消息存入消息队列中
<code>max_mqueue_len</code>	<code>integer</code>	<code>>= 0</code>	<code>1000</code>	消息队列长度
<code>max_inflight</code>	<code>integer</code>	<code>>= 0</code>	<code>0</code>	飞行窗口大小；默认 <code>0</code> 即无限制
<code>max_awaiting_rel</code>	<code>integer</code>	<code>>= 0</code>	<code>0</code>	最大接收；默认 <code>0</code> 即无限制
<code>await_rel_timeout</code>	<code>durtation</code>	<code>> 0</code>	<code>300s</code>	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

Alarm

EMQX 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	<code>\$SYS/brokers/emqx@127.0.0.1/alarms/activate</code>
QoS	0
Retain	1
Payload	<pre> { "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 } </pre>

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 **EMQX Broker** 提供的 **HTTP API** 查询和管理告警。

EMQX 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
<code>os_mon.cpu_check_interval</code>	<code>duration</code>	<code>60s</code>	CPU 占用率的检查间隔
<code>os_mon.cpu_high_watermark</code>	<code>percent</code>	<code>80%</code>	CPU 占用率高水位, 即 CPU 占用率达到多少时激活告警
<code>os_mon.cpu_low_watermark</code>	<code>percent</code>	<code>60%</code>	CPU 占用率低水位, 即 CPU 占用率降低到多少时取消告警
<code>os_mon.mem_check_interval</code>	<code>duration</code>	<code>60%</code>	内存占用率的检查间隔
<code>os_mon.sysmem_high_watermark</code>	<code>percent</code>	<code>70%</code>	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
<code>os_mon.procmem_high_watermark</code>	<code>percent</code>	<code>5%</code>	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
<code>vm_mon.check_interval</code>	<code>duration</code>	<code>30s</code>	进程数量的检查间隔
<code>vm_mon.process_high_watermark</code>	<code>percent</code>	<code>80%</code>	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
<code>vm_mon.process_low_watermark</code>	<code>percent</code>	<code>60%</code>	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
<code>alarm.actions</code>	<code>string</code>	<code>log,publish</code>	告警激活时触发的动作, 目前仅支持 <code>log</code> 与 <code>publish</code> , 即输出日志与发布系统消息
<code>alarm.size_limit</code>	<code>integer</code>	<code>1000</code>	已取消激活告警的最大保存数量, 达到限制后将以 <code>FIFO</code> 原则清理这些告警
<code>alarm.validity_period</code>	<code>duration</code>	<code>24h</code>	已取消激活告警的最大保存时间, 过期的告警将被清理

EMQX 企业版在证书到期日小于**30天**或连接数超过高水位线时会发出告警。用户可以根据实际情况对连接数的高/低水位线进行调整：

配置项	类型	默认值	说明
<code>license.connection_high_watermark_alarm</code>	<code>percent</code>	<code>80%</code>	证书允许的最大连接数高水位, 即实时在线数/最大允许连接数达到多少百分比激活告警
<code>license.connection_low_watermark_alarm</code>	<code>percent</code>	<code>75%</code>	证书允许的最大连接数低水位, 即实时在线数/最大允许连接数达到多少百分比取消告警

数据导入导出

EMQX 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 **EMQX Broker** 中的黑名单、规则引擎配置等存储在 **EMQX 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 **EMQX Broker** 的运行实例，**EMQX 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。

EMQX 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**

示例

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](#))

```
1 $ ./emqx_ctl data export
2 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

```
1 $ cp /var/lib/emqx/data/emqx-export-2020-5-15-17-39-0.json /tmp
```

3. 重新安装 **EMQX Broker** 并启动

```
1 $ ./emqx start
2 EMQX 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

```
1 $ ./emqx_ctl data import /tmp/emqx-export-2020-5-15-17-39-0.json
2 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：

```
1 $ ./emqx_ctl data import /tmp/emqx-export-2020-5-15-17-39-0.json --env '{"auth.mnesia.as":"us'
2 ername"}'
The emqx data has been imported successfully.
```

HTTP API

1. 导出数据

```
1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/export" sh
2
3 {"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

```
1 $ curl --basic -u admin:public -X GET http://localhost:8081/api/v4/data/file/emqx-export-2020
sh
-9-4-10-24-16.json -o /tmp/emqx-export-2020-9-4-10-24-16.json
```

3. 导入数据

```
1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/import" -d @ sh
2 /tmp/emqx-export-2020-9-4-10-24-16.json
3 {"code": 0}
```

第 2、3 步适用于在不同机器上迁移 emqx

模块管理

EMQX 发行包中提供了丰富了功能模块，包括 认证鉴权、协议接入、消息下发、多语言扩展、运维监控、内部模块等。模块管理页面可以启动、关闭模块，还可以进行模块的配置和数据管理。

模块列表

目前 **EMQX** 发行包提供的模块包括：

- 认证鉴权
 - 内置访问控制文件
 - **MySQL** 认证/访问控制
 - **PostgreSQL** 认证/访问控制
 - **Redis** 认证/访问控制
 - **HTTP** 认证/访问控制
 - 内置数据库 认证/访问控制
 - **MongoDB** 认证/访问控制
 - **LDAP** 认证/访问控制
 - **JWT** 认证
- 协议接入
 - **LwM2M** 协议网关
 - **MQTT-SN** 协议网关
 - **TCP** 协议网关
 - **JT/T808** 协议网关
 - **CoAP** 协议网关
 - **Stomp** 协议网关
- 消息下发
 - **Kafka** 消费组
 - **Pulsar** 消费组
 - **MQTT** 订阅者
- 多语言扩展
 - 协议接入
 - 钩子
- 运维监控
 - **Recon**
 - **Prometheus Agent**
 - 日志追踪
- 内部模块
 - 主题监控
 - **MQTT** 增强认证
 - **MQTT** 上下线通知
 - **MQTT** 代理订阅
 - **MQTT** 主题重写
 - **MQTT** 保留消息
 - **MQTT** 延迟发布

启停模块

目前启动模块有以下两种方式：

1. 默认加载
2. 使用 **Dashboard** 启停模块

开启默认加载

如需在 **EMQX** 启动时就默认启动某模块，则直接在 `data/loaded_modules` 添加需要启动的模块名称。

例如，目前 **EMQX** 自动加载的模块有：

```
1  [
2      {
3          "name": "internal_acl",
4          "enable": true,
5          "configs": {"acl_rule_file": "{{ acl_file }}"}
6      },
7      {
8          "name": "presence",
9          "enable": true,
10         "configs": {"qos": 0}
11     },
12     {
13         "name": "recon",
14         "enable": true,
15         "configs": {}
16     },
17     {
18         "name": "retainer",
19         "enable": true,
20         "configs": {
21             "expiry_interval": 0,
22             "max_payload_size": "1MB",
23             "max_retained_messages": 0,
24             "storage_type": "ram"
25         }
26     }
27 ]
```

使用 **Dashboard** 启停模块

若开启了 **Dashboard** 的模块，可以直接通过访问 `http://localhost:18083/modules` 中的模块管理页面启停模块。

ACL File 访问控制

内置 **ACL** 通过文件设置规则，使用上足够简单轻量，适用于规则数量可预测、无变动需求或变动较小的项目。

创建模块

内置访问控制模块默认启动，可以通过**dashboard**页面进行停止和更新，但不可以删除。

The screenshot shows the EMQ Dashboard interface. On the left is a dark sidebar with various menu items: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 规则引擎 (Rule Engine), 统计分析 (Statistics), 模块 (Modules) (which is selected and highlighted in green), 插件 (Plugins), 告警 (Alerts), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area is titled '模块管理' (Module Management) and shows four modules: 'Recon' (status off), '上下线通知' (status off), '内置访问控制文件' (status off), and 'MQTT 保留消息' (status off). Each module has a '了解更多' (More Information) link and a red power-off button.

直接编辑**ACL**文件，或选择具体配置文件进行替换

The screenshot shows the EMQ Monitoring page. The sidebar on the left is identical to the one in the previous screenshot. The main content area is titled '内置访问控制文件' (Built-in Access Control File) and shows the file content: '%{--%} %% [ACL](https://docs.emqx.io/broker/v3/en/config.html) %% %% -type(who()) :: all | binary() | %% {ipaddr, esockd_access:cldr()} | %% {ipaddrs, esockd_access:cldr()} | ...'. There are '了解更多' (More Information) and '删除' (Delete) buttons at the top right of the file preview area. At the bottom are '取消' (Cancel) and '确定' (Confirm) buttons.

定义 ACL

内置 **ACL** 是优先级最低规则表，在所有的 **ACL** 检查完成后，如果仍然未命中则检查默认的 **ACL** 规则。

该规则文件以 **Erlang** 语法的格式进行描述：

```

1  %% 允许 "dashboard" 用户 订阅 "$SYS/#" 主题
2
3  {allow, {user, "dashboard"}, subscribe, ["$SYS/#"]}.
4
5  %% 允许 IP 地址为 "127.0.0.1" 的用户 发布/订阅 "#SYS/#", "#" 主题
6
7  {allow, {ipaddr, "127.0.0.1"}, pubsub, ["$SYS/#", "#"]}.
8
9  %% 拒绝 "所有用户" 订阅 "$SYS/#" "#" 主题
10
11 {deny, all, subscribe, ["$SYS/#", {eq, "#"}]}.
12
13 %% 允许其它任意的发布订阅操作
14
15 {allow, all}.

```

1. 第一条规则允许客户端发布订阅所有主题
2. 第二条规则禁止全部客户端订阅 \$SYS/# 与 # 主题
3. 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
4. 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

可知, 默认的 **ACL** 主要是为了限制客户端对系统主题 \$SYS/# 和全通配主题 # 的权限。

acl.conf 编写规则

`acl.conf` 文件中的规则按书写顺序从上往下匹配。

`acl.conf` 的语法规则包含在顶部的注释中, 熟悉 **Erlang** 语法的可直接阅读文件顶部的注释。或参考以下的释义:

- 以 %% 表示行注释。
- 每条规则由四元组组成, 以 . 结束。
- 元组第一位: 表示规则命中成功后, 执行权限控制操作, 可取值为:
 - * `allow` : 表示 允许
 - * `deny` : 表示 拒绝
- 元组第二位: 表示规则所生效的用户, 可使用的格式为:
 - * `{user, "dashboard"}` : 表明规则仅对 **用户名 (Username)** 为 "dashboard" 的用户生效
 - * `{client, "dashboard"}` : 表明规则仅对 **客户端标识 (ClientId)** 为 "dashboard" 的用户生效
 - * `{ipaddr, "127.0.0.1"}` : 表明规则仅对 **源地址** 为 "127.0.0.1" 的用户生效
 - * `all` : 表明规则对所有的用户都生效
- 元组第三位: 表示规则所控制的操作, 可取值为:
 - * `publish` : 表明规则应用在 **PUBLISH** 操作上
 - * `subscribe` : 表明规则应用在 **SUBSCRIBE** 操作上

- * `pubsub` : 表明规则对 **PUBLISH** 和 **SUBSCRIBE** 操作都有效
- 元组第四位：表示规则所限制的主题列表，内容以数组的格式给出，例如：
 - * `"$SYS/#"` : 为一个 主题过滤器 (**Topic Filter**)；表示规则可命中与 `$SYS/#` 匹配的主题；如：可命中 `"$SYS/#"`，也可命中 `"$SYS/a/b/c"`
 - * `{eq, "#!"}` : 表示字符的全等，规则仅可命中主题为 `#` 的字串，不能命中 `/a/b/c` 等
- 除此之外还存在两条特殊的规则：
- `{allow, all}` : 允许所有操作
- `{deny, all}` : 拒绝所有操作

提示

acl.conf 中应只包含一些简单而通用的规则，使其成为系统基础的 **ACL** 原则。如果需要支持复杂、大量的 **ACL** 内容可以选择外部资源去实现它。

MySQL 认证/访问控制

MySQL 认证/访问控制使用外部 MySQL 数据库作为数据源，可以存储大量数据，同时方便与外部设备管理系统集成。

安装MySQL

打开MySQL官网:<https://dev.mysql.com/downloads/MySQL/5.7.html#downloads>，选择自己需要的版本，这里我们选择MySQL版本为macos-5.7.31

MySQL Community Downloads

MySQL Community Server

The screenshot shows the MySQL Community Downloads page. At the top, there are tabs for "General Availability (GA) Releases" (which is selected), "Archives", and a help icon. Below the tabs, it says "MySQL Community Server 5.7.31". There are dropdown menus for "Select Version" (set to 5.7.31) and "Select Operating System" (set to macOS). A note says "Packages for Catalina (10.15) are compatible with Mojave (10.14)". The main area lists download options:

Download Type	Version	File Size	Action
macOS 10.14 (x86, 64-bit), DMG Archive	5.7.31	372.8M	Download
macOS 10.14 (x86, 64-bit), Compressed TAR Archive	5.7.31	214.3M	Download
macOS 10.14 (x86, 64-bit), Compressed TAR Archive Test Suite	5.7.31	158.9M	Download
macOS 10.14 (x86, 64-bit), TAR	5.7.31	373.1M	Download

At the bottom, a note says "We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download."

安装完毕以后启动MySQL。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

选择“添加”然后点击“模块”菜单，选择“添加”，然后选择MySQL 认证/权限控制模块

需要配置MySQL的地址，用户名，密码（可选）等基本连接参数表

MySQL 认证/访问控制

MySQL 认证/访问控制

配置信息

* MySQL 服务器	* MySQL 数据库名
127.0.0.1:3306	MySQL 数据库名
连接池大小	* MySQL 用户名
8	MySQL 用户名
MySQL 密码	是否重连
MySQL 密码	true
加密方式	用户名/密码 SQL 认证语句
sha256	select password from mqtt_user where username = '%u' limit 1
访问控制 SQL 查询语句	超级用户 SQL 查询语句
select allow, ipaddr, username, clientid, access, topic from mqtt_a	select is_superuser from mqtt_user where username = "%u" limit 1
查询超期时间	开启 SSL
5s	false

取消 **添加**

最后点击“添加”按钮，模块即可添加成功

The screenshot shows the EMQX Dashboard interface. On the left, there's a sidebar with various modules: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 模块 (Modules), 规则引擎 (Rule Engine), 统计分析 (Statistics), 告警 (Alerts), 工具 (Tools), 设置 (Settings), and 通用 (General). The "模块" (Modules) item is highlighted with a green background. In the main content area, there's a "模块管理" (Module Management) section with a count of 1. A green button labeled "选择" (Select) is visible. Below it, there's a card for the "MySQL 认证/访问控制" (MySQL Authentication/Access Control) module. The card features the MySQL logo, the module name, a red power icon, and a "了解更多" (More Information) link. At the top of the main content area, there are tabs: 首页 (Home), 模块 (Modules), 选择模块 (Select Module), and 订阅 (Subscriptions).

认证表：

```

1 CREATE TABLE `mqtt_user` (
2   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
3   `username` varchar(100) DEFAULT NULL,
4   `password` varchar(100) DEFAULT NULL,
5   `salt` varchar(35) DEFAULT NULL,
6   `is_superuser` tinyint(1) DEFAULT 0,
7   `created` datetime DEFAULT NULL,
8   PRIMARY KEY (`id`),
9   UNIQUE KEY `mqtt_username` (`username`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

字段说明:

- **username**: 连接客户端的用户名, 此处的值如果设置为 `$all` 表示该规则适用于所有的用户
- **password**: 连接客户端的密码参数
- **salt**: 密码加盐字符串
- **is_superuser**: 是否是超级用户

进行身份认证时, **EMQX** 将使用当前客户端信息填充并执行用户配置的认证 **SQL**, 查询出该客户端在数据库中的认证数据。

```
1 select password from mqtt_user where username = '%u' limit 1
```

字段说明

- **%u**: 用户名
- **%c**: **clientid**
- **%P**: 明文密码
- **%C**: **TLS** 证书公用名 (证书的域名或子域名), 仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**, 仅当 **TLS** 连接时有效

可以根据业务需要调整认证 **SQL**, 如添加多个查询条件、使用数据库预处理函数, 以实现更多业务相关的功能。但是在任何情况下认证 **SQL** 需要满足以下条件:

1. 查询结果中必须包含 **password** 字段, **EMQX** 使用该字段与客户端密码比对
2. 如果启用了加盐配置, 查询结果中必须包含 **salt** 字段, **EMQX** 使用该字段作为 **salt** (盐) 值
3. 查询结果只能有一条, 多条结果时只取第一条作为有效数据

默认配置下示例数据如下:

```

1 INSERT INTO `mqtt_user` ( `username`, `password`, `salt` )
2 VALUES
3   ('emqx', 'efa1f375d76194fa51a3556a97e641e61685f914d446979da50a551a4333ffd7', NULL);

```

启用 **MySQL** 认证后, 你可以通过用户名: **emqx**, 密码: **public** 连接。

提示

可以在 **SQL** 中使用 **AS** 语法为字段重命名指定 **password**, 或者将 **salt** 值设为固定值。

默认表结构中，我们将 **username** 字段设为了唯一索引 (**UNIQUE**)，与默认的查询语句 (`select password from mqtt_user where username = '%u' limit 1`) 配合使用可以获得非常不错的查询性能。

如果默认查询条件不能满足您的需要，例如你需要根据 **Client ID** 查询相应的 **Password Hash** 和 **Salt**，请确保将 **Client ID** 设置为索引；又或者您想要对 **Username**、**Client ID** 或者其他更多字段进行多条件查询，建议设置正确的单索引或是联合索引。总之，设置正确的表结构和查询语句，尽可能不要让索引失效而影响查询性能。

访问控制表

```

1 CREATE TABLE `mqtt_acl` (
2   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
3   `allow` int(1) DEFAULT 1 COMMENT '0: deny, 1: allow',
4   `ipaddr` varchar(60) DEFAULT NULL COMMENT 'IpAddress',
5   `username` varchar(100) DEFAULT NULL COMMENT 'Username',
6   `clientid` varchar(100) DEFAULT NULL COMMENT 'ClientId',
7   `access` int(2) NOT NULL COMMENT '1: subscribe, 2: publish, 3: pubsub',
8   `topic` varchar(100) NOT NULL DEFAULT '' COMMENT 'Topic Filter',
9   PRIMARY KEY (`id`),
10  INDEX (ipaddr),
11  INDEX (username),
12  INDEX (clientid)
13 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

字段说明：

- **allow**: 禁止 (0)，允许 (1)
- **ipaddr**: 设置 IP 地址
- **username**: 连接客户端的用户名，此处的值如果设置为 `$all` 表示该规则适用于所有的用户
- **clientid**: 连接客户端的**clientid**
- **access**: 允许的操作：订阅 (1)，发布 (2)，订阅发布都可以 (3)
- **topic**: 控制的主题，可以使用通配符，并且可以在主题中加入占位符来匹配客户端信息，例如 `t/%c` 在匹配时主题将会替换为当前客户端的 **clientid**

访问控制的原理是从MySQL中查找跟客户端相关的条目，然后进行鉴权，默认的查询SQL如下：

```

1 select allow, ipaddr, username, clientid, access, topic from mqtt_acl where ipaddr = '%a' or
username = '%u' or username = '$all' or clientid = '%c'

```

可以在认证 SQL 中使用以下占位符，执行时 EMQX 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: **clientid**
- **%a**: 客户端 IP 地址
- **%P**: 明文密码
- **%C**: **TLS** 证书公用名（证书的域名或子域名），仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**，仅当 **TLS** 连接时有效

默认配置下示例数据：

```

1 -- 所有用户不可以订阅系统主题
2 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (0, NULL, '$a
3 ll', NULL, 1, '$SYS/#');
4
5 -- 允许 10.59.1.100 上的客户端订阅系统主题
6 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (1, '10.59.1.
7 100', NULL, NULL, 1, '$SYS/#');
8
9 -- 禁止客户端订阅 /smarthome/+/temperature 主题
10 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (0, NULL, '$a
11 ll', NULL, 1, '/smarthome/+/temperature');
-- 允许客户端订阅包含自身 Client ID 的 /smarthome/${clientid}/temperature 主题
INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (1, NULL, '$a
ll', NULL, 1, '/smarthome/%c/temperature');

```

超级用户

超级用户可以订阅和发布任何**Topic**, 默认**SQL**如下:

```
1 select is_superuser from mqtt_user where username = '%u' limit 1
```

你可以在 **SQL** 中使用以下占位符, 执行时 **EMQX** 将自动填充为客户端信息:

- **%u**: 用户名
- **%c**: **clientid**
- **%C**: **TLS** 证书公用名 (证书的域名或子域名), 仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**, 仅当 **TLS** 连接时有效

你可以根据业务需要调整超级用户 **SQL**, 如添加多个查询条件、使用数据库预处理函数, 以实现更多业务相关的功能。但是任何情况下超级用户 **SQL** 需要满足以下条件:

1. 查询结果中必须包含 **is_superuser** 字段, **is_superuser** 应该显式的为 **true**
2. 查询结果只能有一条, 多条结果时只取第一条作为有效数据

加密规则

```
1 ## 不加盐, 明文
2 plain
3
4 ## 不加盐, 仅做哈希处理
5 sha256
6
7 ## salt 前缀: 使用 sha256 加密 salt + 密码 拼接的字符串
8 salt,sha256
9
10 ## salt 后缀: 使用 sha256 加密 密码 + salt 拼接的字符串
11 sha256,salt
12
13 ## pbkdf2 with macfun iterations dklen
14 ## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha512
15 pbkdf2,sha256,1000,20
```

提示

可参考[:加盐规则与哈希方法](#)。

特殊说明

MySQL 8.0 及以后版本使用了 `caching_sha2_password` 作为默认身份验证模块, 受限于客户端驱动你必须将其更改为 `MySQL_native_password` 模块:

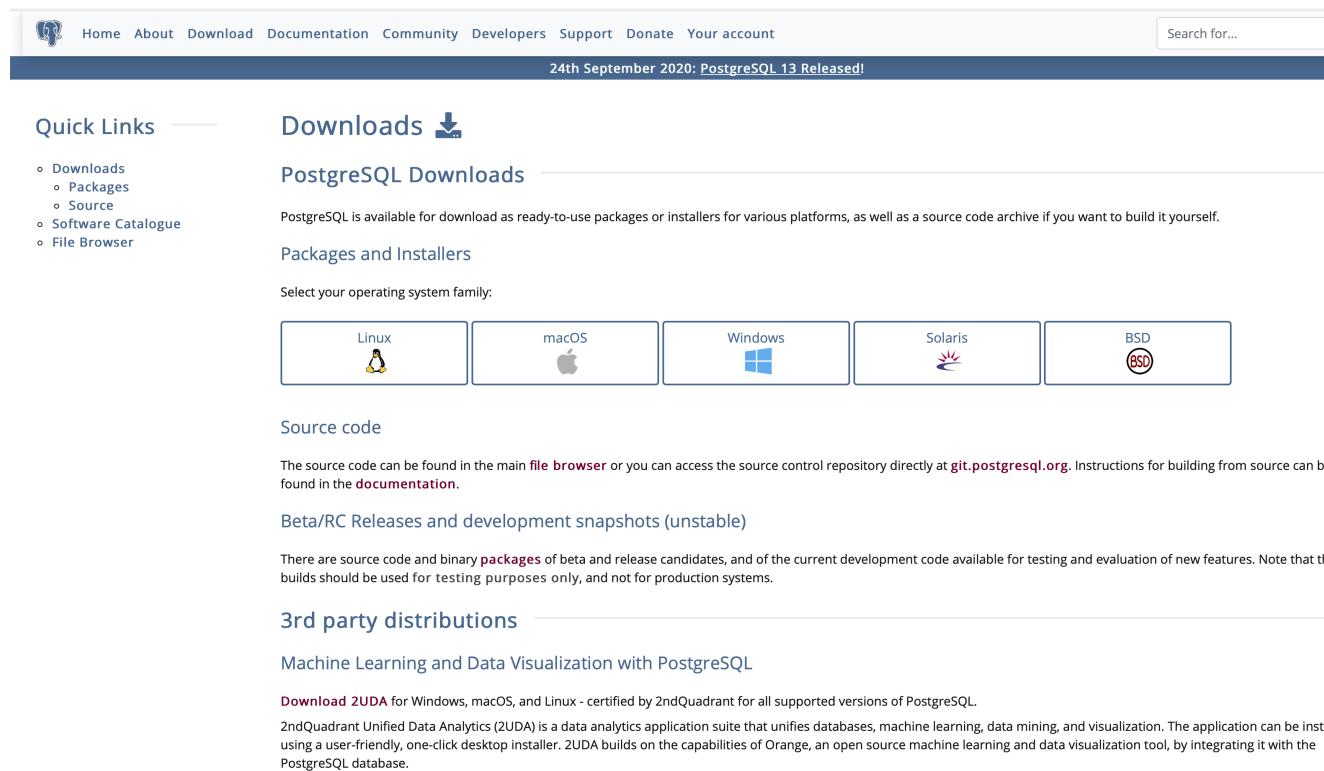
```
1 ALTER USER 'your_username'@'your_host' IDENTIFIED WITH MySQL_native_password BY 'your_password';
```

PostgreSQL 认证/访问控制

PostgreSQL 认证/访问控制使用外部 **PostgreSQL** 数据库作为数据源，可以存储大量数据，同时方便与外部设备管理系统集成。

安装PostgreSQL

打开**PostgreSQL**官网:<https://www.postgresql.org/download/>，选择自己需要的版本，这里我们选择**PostgreSQL**版本为**macos-10.13**



The screenshot shows the PostgreSQL Downloads page. At the top, there's a navigation bar with links for Home, About, Download, Documentation, Community, Developers, Support, Donate, and Your account. A search bar is also present. Below the navigation bar, a banner indicates "24th September 2020: PostgreSQL 13 Released!". The main content area is titled "Downloads" with a downward arrow icon. It has two main sections: "PostgreSQL Downloads" and "Source code". Under "PostgreSQL Downloads", there's a note about ready-to-use packages or installers for various platforms, including source code archives. Below this, there's a section for "Packages and Installers" with a note about selecting an operating system family. Five boxes represent different platforms: Linux (Ubuntu logo), macOS (Apple logo), Windows (Windows logo), Solaris (Solaris logo), and BSD (BSD logo). Under "Source code", there's a note about finding source code in the file browser or directly from git.postgresql.org. Below this, there's a section for "Beta/RC Releases and development snapshots (unstable)" with a note about using them for testing purposes only. Finally, there's a section for "3rd party distributions" with a link to "Machine Learning and Data Visualization with PostgreSQL".

安装完毕以后启动**PostgreSQL**。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

Dashboard

首页 / 模块

模块管理 4 选择

搜索模块...

- Recon
- 内置访问控制文件
- 上线下线通知
- MQTT 保留消息

选择PostgreSQL 认证/权限控制模块

选择模块 26 认证/访问控制

搜索模块...

模块	描述	操作
内置访问控制	内置访问控制	已添加
LDAP 认证/访问控制	LDAP 认证/访问控制	选择
HTTP 认证/访问控制	HTTP 认证/访问控制	选择
PostgreSQL 认证/访问控制	PostgreSQL 认证/访问控制	选择
MySQL 认证/访问控制	MySQL 认证/访问控制	已添加
内置数据库 认证/访问控制	Erlang 内置数据库 认证/访问控制	选择
MongoDB 认证/访问控制	MongoDB 认证/访问控制	已添加
PSKFile 认证	PKSFile 认证	选择
Redis 认证/访问控制	Redis 认证/访问控制	选择

需要配置PostgreSQL的地址，用户名，密码（可选）等基本连接参数表

PostgreSQL 认证/访问控制

PostgreSQL 认证/访问控制

配置信息

* PostgreSQL 服务器	连接池大小
<input type="text" value="127.0.0.1:5432"/>	<input type="text" value="8"/>
* PostgreSQL 数据库名	* PostgreSQL 用户名
<input type="text" value="PostgreSQL 数据库名"/>	<input type="text" value="PostgreSQL 用户名"/>
PostgreSQL 密码	是否重连
<input type="text" value="PostgreSQL 密码"/>	<input type="text" value="true"/>
加密方式	认证 SQL 语句
<input type="text" value="sha256"/>	<input type="text" value="select password from mqtt_user where username = '%u' limit 1"/>
访问控制 SQL 语句	超级用户 SQL 查询语句
<input type="text" value="select allow, ipaddr, username, clientid, access, topic from mqtt_a"/>	<input type="text" value="select is_superuser from mqtt_user where username = '%u' limit 1"/>
查询超期时间	开启 SSL
<input type="text" value="5s"/>	<input type="text" value="false"/>

[取消](#) [添加](#)

最后点击“添加”按钮，模块即可添加成功

The screenshot shows the EMQX Enterprise Dashboard interface. On the left, there's a sidebar with various navigation items: 监控 (Monitoring), 客户端 (Client), 主题 (Topic), 订阅 (Subscription), 模块 (Module), 规则引擎 (Rule Engine), 统计分析 (Statistics), 告警 (Alert), 工具 (Tools), 设置 (Settings), and 通用 (General). The '模块' item is currently selected and highlighted in green. The main content area has a breadcrumb navigation: 首页 / 模块. Below this, there's a search bar with filters: 首页, 模块 (selected), 选择模块, and 订阅. A '模块管理' section shows 1 module selected, with a green '选择' (Select) button. A detailed view of the 'PostgreSQL 认证/访问控制' module is shown, featuring its icon (a blue PostgreSQL logo), name, description, and a red 'OFF' status indicator. A green '了解更多' (More information) button is located to the right.

认证表：

```

1 CREATE TABLE mqtt_user (
2     id SERIAL PRIMARY KEY,
3     username CHARACTER VARYING(100),
4     password CHARACTER VARYING(100),
5     salt CHARACTER VARYING(40),
6     is_superuser BOOLEAN,
7     UNIQUE (username)
8 )

```

字段说明：

- **username**: 连接客户端的用户名，此处的值如果设置为 `$all` 表示该规则适用于所有的用户
- **password**: 连接客户端的密码参数
- **salt**: 密码加盐字符串
- **is_superuser**: 是否是超级用户

进行身份认证时，**EMQX** 将使用当前客户端信息填充并执行用户配置的认证 **SQL**，查询出该客户端在数据库中的认证数据。

```

1 select password from mqtt_user where username = '%u' limit 1

```

字段说明

- `%u`: 用户名
- `%c`: **clientid**
- `%P`: 明文密码
- `%C`: **TLS** 证书公用名（证书的域名或子域名），仅当 **TLS** 连接时有效
- `%d`: **TLS** 证书 **subject**，仅当 **TLS** 连接时有效

可以根据业务需要调整认证 **SQL**，如添加多个查询条件、使用数据库预处理函数，以实现更多业务相关的功能。但是任何情况下认证 **SQL** 需要满足以下条件：

1. 查询结果中必须包含 **password** 字段，**EMQX** 使用该字段与客户端密码比对
2. 如果启用了加盐配置，查询结果中必须包含 **salt** 字段，**EMQX** 使用该字段作为 **salt**（盐）值
3. 查询结果只能有一条，多条结果时只取第一条作为有效数据

默认配置下示例数据如下：

```

1 INSERT INTO `mqtt_user` ( `username` , `password` , `salt` )
2 VALUES
3     ('emqx', 'efa1f375d76194fa51a3556a97e641e61685f914d446979da50a551a4333ffd7', NULL);

```

启用 **PostgreSQL** 认证后，你可以通过用户名：**emqx**，密码：**public** 连接。

提示

可以在 **SQL** 中使用 **AS** 语法为字段重命名指定 **password**，或者将 **salt** 值设为固定值。

进阶

默认表结构中，我们将 **username** 字段设为了唯一索引 (**UNIQUE**)，与默认的查询语句 (`select password from mqtt_user where username = '%u' limit 1`) 配合使用可以获得非常不错的查询性能。

如果默认查询条件不能满足您的需要，例如你需要根据 **Client ID** 查询相应的 **Password Hash** 和 **Salt**，请确保将 **Client ID** 设置为索引；又或者您想要对 **Username**、**Client ID** 或者其他更多字段进行多条件查询，建议设置正确的单索引或是联合索引。总之，设置正确的表结构和查询语句，尽可能不要让索引失效而影响查询性能。

访问控制表

```

1  CREATE TABLE mqtt_acl (
2      id SERIAL PRIMARY KEY,
3      allow INTEGER,
4      ipaddr CHARACTER VARYING(60),
5      username CHARACTER VARYING(100),
6      clientid CHARACTER VARYING(100),
7      access INTEGER,
8      topic CHARACTER VARYING(100)
9  );
10 CREATE INDEX ipaddr ON mqtt_acl (ipaddr);
11 CREATE INDEX username ON mqtt_acl (username);
12 CREATE INDEX clientid ON mqtt_acl (clientid);

```

字段说明：

- **allow**: 禁止 (0)，允许 (1)
- **ipaddr**: 设置 IP 地址
- **username**: 连接客户端的用户名，此处的值如果设置为 `$all` 表示该规则适用于所有的用户
- **clientid**: 连接客户端的 **clientid**
- **access**: 允许的操作：订阅 (1)，发布 (2)，订阅发布都可以 (3)
- **topic**: 控制的主题，可以使用通配符，并且可以在主题中加入占位符来匹配客户端信息，例如 `t/%c` 在匹配时主题将会替换为当前客户端的 **clientid**

访问控制的原理是从PostgreSQL中查找跟客户端相关的条目，然后进行鉴权，默认的查询SQL如下：

```

1  select allow, ipaddr, username, clientid, access, topic from mqtt_acl where ipaddr = '%a' or
   username = '%u' or username = '$all' or clientid = '%c'

```

可以在认证 SQL 中使用以下占位符，执行时 EMQX 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: **clientid**
- **%a**: 客户端 IP 地址
- **%P**: 明文密码
- **%C**: **TLS** 证书公用名（证书的域名或子域名），仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**，仅当 **TLS** 连接时有效

默认配置下示例数据：

```

1 -- 所有用户不可以订阅系统主题
2 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (0, NULL, '$a
3 ll', NULL, 1, '$SYS/#');
4
5 -- 允许 10.59.1.100 上的客户端订阅系统主题
6 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (1, '10.59.1.
7 100', NULL, NULL, 1, '$SYS/#');
8
9 -- 禁止客户端订阅 /smarthome/+/temperature 主题
10 INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (0, NULL, '$a
11 ll', NULL, 1, '/smarthome/+/temperature');
-- 允许客户端订阅包含自身 Client ID 的 /smarthome/${clientid}/temperature 主题
INSERT INTO mqtt_acl (allow, ipaddr, username, clientid, access, topic) VALUES (1, NULL, '$a
ll', NULL, 1, '/smarthome/%c/temperature');

```

超级用户

超级用户可以订阅和发布任何**Topic**, 默认**SQL**如下:

```
1 select is_superuser from mqtt_user where username = '%u' limit 1
```

你可以在 **SQL** 中使用以下占位符, 执行时 **EMQX** 将自动填充为客户端信息:

- **%u**: 用户名
- **%c**: **clientid**
- **%C**: **TLS** 证书公用名 (证书的域名或子域名), 仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**, 仅当 **TLS** 连接时有效

你可以根据业务需要调整超级用户 **SQL**, 如添加多个查询条件、使用数据库预处理函数, 以实现更多业务相关的功能。但是任何情况下超级用户 **SQL** 需要满足以下条件:

1. 查询结果中必须包含 **is_superuser** 字段, **is_superuser** 应该显式的为 **true**
2. 查询结果只能有一条, 多条结果时只取第一条作为有效数据

提示

如果不使用超级用户功能, 注释并禁用该选项能有效提高效率

加密规则

```
1 ## 不加盐, 明文
2 plain
3
4 ## 不加盐, 仅做哈希处理
5 sha256
6
7 ## salt 前缀: 使用 sha256 加密 salt + 密码 拼接的字符串
8 salt,sha256
9
10 ## salt 后缀: 使用 sha256 加密 密码 + salt 拼接的字符串
11 sha256,salt
12
13 ## pbkdf2 with macfun iterations dklen
14 ## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha512
15 pbkdf2,sha256,1000,20
```

提示

可参考[:加盐规则与哈希方法](#)。

Redis 认证/访问控制

Redis 认证/访问控制使用外部 Redis 数据库作为数据源，可以存储大量数据，同时方便与外部设备管理系统集成。

搭建 Redis 环境，以 MacOS X 为例：

```

1   $ wget http://download.redis.io/releases/redis-4.0.14.tar.gz
2   $ tar xzf redis-4.0.14.tar.gz
3   $ cd redis-4.0.14
4   $ make && make install
5
6   # 启动 redis
7   $ redis-server

```

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there is a sidebar with various options: 监控 (Monitoring), 客户端 (Client), 主题 (Topic), 订阅 (Subscription), 模块 (Module) [highlighted in green], 规则引擎 (Rule Engine), 统计分析 (Statistics Analysis), 告警 (Alert), 工具 (Tools), and WebSocket. The main area is titled '模块管理' (Module Management) with a count of 2 modules. It lists two modules: '上下线通知' (Online/Offline Notification) and '内置访问控制' (Built-in Access Control). A red arrow points to the '选择' (Select) button in the top right corner of the module management dialog.

选择 Redis 认证/访问控制模块

The screenshot shows the 'Select Module' page with a search bar at the top. Below it, there are several module cards arranged in a grid:

- MySQL 认证/访问控制**: MySQL 认证/访问控制. Buttons: 选择 (Select), 了解更多 (More Information).
- MongoDB 认证/访问控制**: MongoDB 认证/访问控制. Buttons: 选择 (Select), 了解更多 (More Information).
- 内置数据库 认证/访问控制**: Erlang 内置数据库 认证/访问控制. Buttons: 选择 (Select), 了解更多 (More Information).
- PKSFile 认证**: PKSFile 认证. Buttons: 选择 (Select), 了解更多 (More Information).
- JWT 认证**: JWT 认证. Buttons: 选择 (Select), 了解更多 (More Information).
- Redis 认证/访问控制**: Redis 认证/访问控制. This card is highlighted with a red border. Buttons: 选择 (Select), 了解更多 (More Information).

At the bottom left, there is a 'Protocol Access' section with a 'Protocol Access' button.

配置相关参数

点击添加后，模块添加完成

认证默认数据结构

Redis 认证默认配置下使用哈希表存储认证数据，使用 `mqtt_user:` 作为 **Redis** 键前缀，数据结构如下：

```

1  redis> hgetall mqtt_user:emqx
2      password public

```

默认配置下示例数据如下：

```

1  HMSET mqtt_user:emqx password public

```

启用 **Redis** 认证后，你可以通过用户名：**emqx**，密码：**public** 连接。

提示

这是默认配置使用的数据结构，熟悉该模块的使用后，你可以使用任何满足条件的数据结构进行认证。

加盐规则与哈希方法

```

1  ## 不加盐，明文
2  plain
3
4  ## 不加盐，仅做哈希处理
5  sha256
6
7  ## salt 前缀：使用 sha256 加密 salt + 密码 拼接的字符串
8  salt,sha256
9
10 ## salt 后缀：使用 sha256 加密 密码 + salt 拼接的字符串
11 sha256,salt
12
13 ## pbkdf2 with macfun iterations dklen
14 ## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha512
15 pbkdf2,sha256,1000,20

```

认证查询命令 (**auth query cmd**)

进行身份认证时，**EMQX** 将使用当前客户端信息填充并执行用户配置的认证查询命令，查询出该客户端在 **Redis** 中的认证数据。

```
1 HMGET mqtt_user:%u password
```

你可以在命令中使用以下占位符，执行时 **EMQX** 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: Client ID
- **%C**: TLS 证书公用名（证书的域名或子域名），仅当 TLS 连接时有效
- **%d**: TLS 证书 subject，仅当 TLS 连接时有效

你可以根据业务需要调整认证查询命令，使用任意 [Redis 支持的命令](#)，但是任何情况下认证查询命令需要满足以下条件：

1. 查询结果中第一个数据必须为 **password**，**EMQX** 使用该字段与客户端密码比对
2. 如果启用了加盐配置，查询结果中第二个数据必须是 **salt** 字段，**EMQX** 使用该字段作为 **salt**（盐）值

访问控制默认数据结构

ACL 规则数据

```

1  ## 格式
2  HSET mqtt_acl:[username clientid][topic] [access]
3
4  ## 结构
5  redis> hgetall mqtt_acl:emqx
6  testtopic/1 1

```

默认配置下示例数据：

```

1  HSET mqtt_acl:emqx # 1
2
3  HSET mqtt_acl:testtopic/2 2

```

ACL 查询命令 (acl cmd)

进行 **ACL** 鉴权时，**EMQX** 将使用当前客户端信息填充并执行用户配置的超级用户命令，如果没有启用超级用户命令或客户端不是超级用户，则使用 **ACL** 查询命令查询出该客户端在数据库中的 **ACL** 规则。

```
1  HGETALL mqtt_acl:%u
```

sh

你可以在 **ACL** 查询命令中使用以下占位符，执行时 **EMQX** 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: Client ID

你可以根据业务需要调整 **ACL** 查询命令，但是任何情况下 **ACL** 查询命令需要满足以下条件：

1. 哈希中使用 **topic** 作为键，**access** 作为值

超级用户查询命令 (super cmd)

进行 **ACL** 鉴权时，**EMQX** 将使用当前客户端信息填充并执行用户配置的超级用户查询命令，查询客户端是否为超级用户。客户端为超级用户时将跳过 **ACL** 查询命令。

```
1  HGET mqtt_user:%u is_superuser
```

sh

你可以在命令中使用以下占位符，执行时 **EMQX** 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: Client ID
- **%C**: TLS 证书公用名（证书的域名或子域名），仅当 TLS 连接时有效
- **%d**: TLS 证书 subject，仅当 TLS 连接时有效

你可以根据业务需要调整超级用户查询命令，如添加多个查询条件、使用数据库预处理函数，以实现更多业务相关的功能。但是任何情况下超级用户查询命令需要满足以下条件：

1. 查询结果中第一个数据必须为 **is_superuser** 数据

提示

如果不需要超级用户功能，注释并禁用该选项能有效提高效率

HTTP 认证/访问控制

HTTP 认证/访问控制使用外部自建 HTTP 应用认证数据源，根据 **HTTP API** 返回的数据判定认证结果，能够实现复杂的认证鉴权逻辑和实现复杂的 **ACL** 校验逻辑。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard's 'Modules' management screen. On the left is a dark sidebar with various navigation options like Monitoring, Clients, Themes, Subscriptions, Rule Engines, Statistics, and Modules. The 'Modules' option is selected and highlighted in green. The main area is titled 'Module Management' and shows four available modules: 'Recon' (with a Recon icon), '上下线通知' (with a bell icon), '内置访问控制文件' (with an ACL icon), and 'MQTT 保留消息' (with a checkmark icon). Each module has a 'More' link and a 'Select' button. A search bar at the top right says 'Search module...'. The top navigation bar includes '首页 / 模块' and tabs for '选择模块', 'WebSocket', and '插件'.

选择 HTTP 认证/访问控制模块

This screenshot shows the 'Select Module' screen under the 'Authentication/Access Control' tab. The sidebar on the left remains the same. The main area lists several authentication modules: '内嵌访问控制' (Built-in Access Control), 'LDAP 认证/访问控制' (LDAP Authentication/Access Control), 'PostgreSQL 认证/访问控制' (PostgreSQL Authentication/Access Control), 'HTTP 认证/访问控制' (HTTP Authentication/Access Control), 'MySQL 认证/访问控制' (MySQL Authentication/Access Control), 'MongoDB 认证/访问控制' (MongoDB Authentication/Access Control), '内置数据库 认证/访问控制' (Built-in Database Authentication/Access Control), and 'JWT 认证' (JWT Authentication). The 'HTTP 认证/访问控制' module is highlighted with a red border. The top navigation bar includes '首页 / 模块 / 选择' and tabs for '选择模块', 'Rule Engine', 'emqx@127.0.0.1', 'Settings', and 'Subscriptions'.

配置相关参数

The screenshot shows the EMQX Enterprise V4.4 configuration interface. On the left is a dark sidebar with a green header bar labeled '模块' (Module). The main area has a title '配置信息' (Configuration Information) and several input fields for configuring the 'HTTP Authentication/Acl' module. The fields include:

- 认证请求地址: http://127.0.0.1:8991/mqtt/auth
- 认证请求参数: clientid=%c,username=%u,password=%P
- 访问控制请求地址: http://127.0.0.1:8991/mqtt/acl
- 访问控制请求参数: access=%A,username=%u,clientid=%c,ipaddr=%a,tc
- 超级用户请求地址: 超级用户请求地址
- 超级用户请求参数: 超级用户请求参数
- HTTP 请求方法: POST
- HTTP 请求类型: x-www-form-urlencoded
- 请求头: A table with columns '键' (Key) and '值' (Value), currently empty.
- 重试次数: 3
- 重试间隔: 1s
- Retry Backoff: 2
- HTTP Timeout: 2000

点击添加后，模块添加完成

The screenshot shows the EMQX Enterprise V4.4 dashboard. The left sidebar has a green header bar labeled '模块' (Module). The main area has a title '首页 / 模块' (Home / Modules) and a sub-section titled '模块管理'. It shows a card for the 'HTTP 认证/访问控制' (HTTP Authentication/Access Control) module, which is currently selected (indicated by a green border around its button). The card includes:

- 模块管理: 1
- 选择 (Select)
- HTTP 认证/访问控制 (HTTP Authentication/Access Control)
- HTTP 认证 (HTTP Authentication)
- 了解更多 (More information)

HTTP 认证原理

EMQX 在设备连接事件中使用当前客户端相关信息作为参数，向用户自定义的认证服务发起请求查询权限，通过返回的 **HTTP 响应状态码 (HTTP statusCode)** 来处理认证请求。

- 认证失败：API 返回非 200 的状态码

- 认证成功: **API** 返回 **200** 状态码
- 忽略认证: **API** 返回 **200** 状态码且消息体 **ignore**

认证请求

进行身份认证时, **EMQX** 将使用当前客户端信息填充并发起用户配置的认证查询请求, 查询出该客户端在 **HTTP** 服务器端的认证数据。

```

1 ## 认证请求地址
2 http://127.0.0.1:8991/mqtt/auth
3
4 ## HTTP 请求方法
5 ## Value: POST | GET
6 POST
7
8 ## 请求参数
9 clientid=%c,username=%u,password=%p
10

```

HTTP 请求方法为 **GET** 时, 请求参数将以 **URL** 查询字符串的形式传递; **POST** 请求则将请求参数以普通表单形式提交 (**content-type** 为 **x-www-form-urlencoded**) 。

你可以在认证请求中使用以下占位符, 请求时 **EMQX** 将自动填充为客户端信息:

- **%u**: 用户名
- **%c**: Client ID
- **%a**: 客户端 IP 地址
- **%r**: 客户端接入协议
- **%P**: 明文密码
- **%p**: 客户端端口
- **%C**: TLS 证书公用名 (证书的域名或子域名), 仅当 TLS 连接时有效
- **%d**: TLS 证书 subject, 仅当 TLS 连接时有效

警告

推荐使用 **POST** 与 **PUT** 方法, 使用 **GET** 方法时明文密码可能会随 **URL** 被记录到传输过程中的服务器日志中。

HTTP 访问控制原理

EMQX 在设备发布、订阅事件中使用当前客户端相关信息作为参数, 向用户自定义的认证服务发起请求权限, 通过返回的 **HTTP** 响应状态码 (**HTTP statusCode**) 来处理 **ACL** 授权请求。

- 无权限: **API** 返回非 **200** 状态码
- 授权成功: **API** 返回 **200** 状态码
- 忽略授权: **API** 返回 **200** 状态码且消息体为固定内容: "**ignore**"

HTTP 请求信息

HTTP API 基础请求信息，配置证书、请求头与重试规则。

进行发布、订阅认证时，**EMQX** 将使用当前客户端信息填充并发起用户配置的 **ACL** 授权查询请求，查询出该客户端在 **HTTP** 服务器端的授权数据。

superuser 请求

首先查询客户端是否为超级用户，客户端为超级用户时将跳过 **ACL** 查询。

```

1 # etc/plugins/emqx_auth_http.conf
2
3 ## 超级用户请求地址
4 http://127.0.0.1:8991/mqtt/superuser
5
6 ## HTTP 请求方法
7 ## Value: POST | GET
8 POST
9
10 ## 超级用户请求参数
11 clientid=%c,username=%u

```

ACL 访问控制请求

```

1
2 ## 访问控制请求地址
3 http://127.0.0.1:8991/mqtt/acl
4
5 ## HTTP 请求方法
6 ## Value: POST | GET
7 POST
8
9 ## 访问控制请求参数
10 access=%A,username=%u,clientid=%c,ipaddr=%a,topic=%t,mountpoint=%m
11

```

请求说明

HTTP 请求方法为 **GET** 时，请求参数将以 **URL** 查询字符串的形式传递；**POST**、**PUT** 请求则将请求参数以普通表单形式提交（**content-type** 为 **x-www-form-urlencoded**）。

你可以在认证请求中使用以下占位符，请求时 **EMQX** 将自动填充为客户端信息：

- **%A**: 操作类型，'1' 订阅；'2' 发布
- **%u**: 客户端用户名
- **%c**: Client ID
- **%a**: 客户端 IP 地址
- **%r**: 客户端接入协议

- %m: Mountpoint
- %t: 主题

警告

推荐使用 **POST** 与 **PUT** 方法，使用 **GET** 方法时明文密码可能会随 **URL** 被记录到传输过程中的服务器日志中。

内置数据库 认证/访问控制

内置数据库认证使用 **EMQX** 内置 **Mnesia** 数据库存储客户端 **Clientid/Username** 与密码，支持通过 **HTTP API** 管理认证数据。

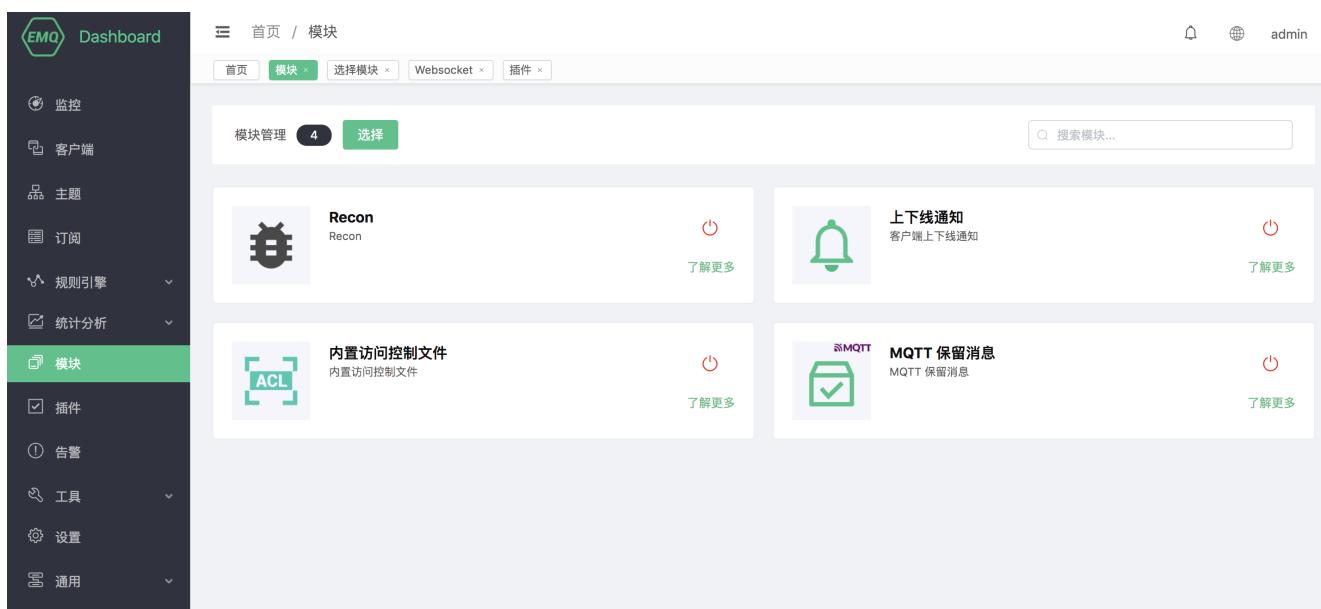
内置数据库认证不依赖外部数据源，使用上足够简单轻量。

Dashboard 管理

内置数据库认证可以通过 **EMQX Dashboard** 的“模块”进行开关以及管理

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



点击“选择”，然后选择内置数据库认证模块

选择模块 23 认证授权 协议接入 消息下发 多语言扩展 运维监控 内部模块 搜索模块...

认证授权

- ACL 内置访问控制文件 已添加 了解更多
- LDAP 认证/访问控制 LDAP 认证/访问控制 选择 了解更多
- HTTP 认证/访问控制 HTTP 认证/访问控制 选择 了解更多
- PostgreSQL 认证/访问控制 PostgreSQL 认证/访问控制 选择 了解更多
- MySQL 认证/访问控制 MySQL 认证/访问控制 选择 了解更多
- MongoDB 认证/访问控制 MongoDB 认证/访问控制 选择 了解更多
- PKSFile 认证 PKSFile 认证 选择 了解更多
- JWT 认证 JWT 认证 选择 了解更多
- Redis 认证/访问控制 Redis 认证/访问控制 选择 了解更多

协议接入

- COAP 接入网关 COAP 接入网关 选择 了解更多
- STOMP 接入网关 STOMP 接入网关 选择 了解更多
- LwM2M 接入网关 LwM2M 接入网关 选择 了解更多
- MQTT-SN 接入网关 MQTT-SN 接入网关 选择 了解更多
- TCP 接入网关 TCP 接入网关 选择 了解更多
- JT/T808 接入网关 JT/T808 接入网关 选择 了解更多

配置相关参数

内置数据库 认证/访问控制 Erlang 内置数据库 认证/访问控制 删除

配置信息

加密类型 sha256

取消 确定

最后点击“添加”按钮模块即可添加成功。

管理数据

内置数据库可以通过 **dashboard** 管理认证与访问控制数据

认证数据

可以通过 **dashboard** 对认证数据进行管理

当客户端连接 **EMQX** 时，内置数据库认证会获取 **CONNENT** 报文中的 **Clientid** 与 **Username**，然后数据库中记录的密码进行匹配，一旦匹配成功则认证成功。

访问控制数据

可以通过 **dashboard** 对访问控制数据进行管理

HTTP API

内置数据库 认证/访问控制 还提供了 **HTTP API**

Mnesia 认证

Mnesia 认证使用 **EMQX** 内置 **Mnesia** 数据库存储客户端 **Client ID/Username** 与密码，支持通过 **HTTP API** 管理认证数据。

Mnesia 认证不依赖外部数据源，使用上足够简单轻量，**Mnesia** 支持使用 **Client ID** 或 **Username** 进行认证。

POST api/v4/auth_clientid

创建基于 **Client ID** 的认证规则。

Parameters (json):

Name	Type	Required	Description
clientid	String	True	Client ID
password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X POST \
6   -d '{"clientid": "emqx_c", "password": "emqx_p"}' \
7   http://localhost:8081/api/v4/auth_clientid
8
9 ## Return
10 {"code":0}

```

POST api/v4/auth_username

创建基于 **Username** 的认证规则。

Parameters (json):

Name	Type	Required	Description
username	String	True	Username
password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X POST \
6   -d '{"username": "emqx_u", "password": "emqx_p"}' \
7   http://localhost:8081/api/v4/auth_username
8
9 ## Return
10 {"code":0}

```

POST api/v4/auth_clientid

批量创建基于 **Client ID** 的认证规则。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Description
].clientId	String	True	Client ID
].password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	

Examples

```

1  ## Request
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X POST \
6      -d '[{"clientId": "emqx_c_1", "password": "emqx_p"}, {"clientId": "emqx_c_2", "password": "emqx_p"}]' \
7      http://localhost:8081/api/v4/auth_clientid
8
9
10 ## Return
11 {
12     "data":{
13         "emqx_c_2":"ok",
14         "emqx_c_1":"ok"
15     },
16     "code":0
17 }
```

POST api/v4/auth_username

批量创建基于 **Username** 的认证规则。

Path Parameters: 无

Parameters (json):

Name	Type	Required	Description
].username	String	True	Username
].password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	

Examples

```

1  ## Request
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X POST \
6      -d '[{"username": "emqx_u_1", "password": "emqx_p"}, {"username": "emqx_u_2", "password": "emqx_p"}]' \
7      http://localhost:8081/api/v4/auth_username
8
9
10 ## Return
11 {
12     "data": {
13         "emqx_u_2": "ok",
14         "emqx_u_1": "ok"
15     },
16     "code": 0
17 }
```

GET api/v4/auth_clientid

查看已经添加的认证数据。

Query String Parameters:

支持模糊查询，其包含的查询参数有：

Name	Type	Required	Description
_like_clientid	String	False	客户端标识符，子串方式模糊查找

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
meta	Object	规则对象
data	Object	规则对象
- data.[] .clientid	String	Client ID

Example

```

1  ## Return
2  $ curl -i \
3  --basic \
4  -u admin:public \
5  -X GET \
6  http://localhost:8081/api/v4/auth_clientid?_like_clientid=emqx
7
8  ## Request
9  {
10    "meta": {
11      "page": 1,
12      "limit": 10,
13      "count": 3
14    },
15    "data": [
16      {
17        "clientid": "emqx_c_1"
18      },
19      {
20        "clientid": "emqx_c_2"
21      },
22      {
23        "clientid": "emqx_c"
24      }
25    ],
26    "code": 0
27  }

```

GET api/v4/auth_username

查看已经添加的认证数据。

Query String Parameters: 支持模糊查询，其包含的查询参数有：

Name	Type	Required	Description
_like_username	String	False	客户端用户名，子串方式模糊查找

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
meta	Object	规则对象
data	Object	规则对象
- data.[].username	String	Client ID

Example

```

1  ## Return
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X GET \
6      http://localhost:8081/api/v4/auth_username?_like_username=emqx
7
8  ## Request
9  {
10     "meta": {
11         "page": 1,
12         "limit": 10,
13         "count": 3
14     },
15     "data": [
16         {
17             "username": "emqx_u"
18         },
19         {
20             "username": "emqx_u_2"
21         },
22         {
23             "username": "emqx_u_1"
24         }
25     ],
26     "code": 0
27 }

```

GET api/v4/auth_clientid/{clientid}

获取指定的资源的详细信息。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	Client ID

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.clientid	String	Client ID
- data.password	String	注意此处返回的密码是使用配置文件指定哈希方式加密后的密码

Example

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/auth_clientid/emqx_c
7
8 ## Return
9 {
10   "data": {
11     "password": "bb7bb456355aaeb55a4eb26ea286314fc360138720cfca2c852d4dfb8cd834",
12     "clientid": "emqx_c"
13   },
14   "code": 0
15 }

```

GET api/v4/auth_username/{username}

获取指定的资源的详细信息。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.username	String	Username
- data.password	String	注意此处返回的密码是使用配置文件指定哈希方式加密后的密码

Example

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/auth_username/emqx_u
7
8 ## Return
9 {
10   "data": {
11     "password": "bb7bb456355aaeb55a4eb26ea286314fc360138720cfca2c852d4dfb8cd834",
12     "clientid": "emqx_u"
13   },
14   "code": 0
15 }

```

PUT api/v4/auth_clientid/{clientid}

更新已添加的认证数据。

Parameters (json):

Name	Type	Required	Description
clientid	String	True	Client ID

Parameters (json):

Name	Type	Required	Description
password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples

```

1  ## Request
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X PUT \
6      -d '{"password": "emqx_new_p"}' \
7      http://localhost:8081/api/v4/auth_clientid/emqx_c
8
9  ## Return
10 {"code":0}

```

PUT api/v4/auth_username/{username}

更新已添加的认证数据。

Parameters (json):

Name	Type	Required	Description
username	String	True	Username

Parameters (json):

Name	Type	Required	Description
password	String	True	密码

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples

```

1  ## Request
2  curl -i \
3    --basic \
4    -u admin:public \
5    -X PUT \
6    -d '{"password": "emqx_new_p"}' \
7    http://localhost:8081/api/v4/auth_username/emqx_u
8
9  ## Return
10 {"code":0}

```

DELETE /api/v4/auth_clientid/{clientid}

删除认证规则。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	Client ID

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

```

1  ## Request
2  curl -i \
3    --basic \
4    -u admin:public \
5    -X Delete\
6    http://localhost:8081/api/v4/auth_clientid/emqx_c
7
8  ## Return
9  {"code":0}

```

DELETE /api/v4/auth_username/{username}

删除认证规则。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X Delete\
6   http://localhost:8081/api/v4/auth_username/emqx_u
7
8 ## Return
9 {"code":0}

```

Mnesia 访问控制

Mnesia ACL 使用 **EMQX** 内置的 **Mnesia** 数据库存储 **ACL** 规则，可以存储数据、动态管理 **ACL**，方便与外部设备管理系统集成

POST api/v4/acl

添加 **ACL** 规则。

- **Clientid ACL**

Parameters (json):

Name	Type	Required	Description
clientid	String	True	Client ID
topic	String	True	主题
action	sub/pub/pubsub	True	动作
access	allow/deny	True	是否允许

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.clientid	String	Clientid
- data.topic	String	主题
- data.action	String	动作
- data.access	String	是否允许

Examples

```

1  ## Request
2  curl -i \
3  --basic \
4  -u admin:public \
5  -X POST \
6  -d '{"clientid":"emqx_c", "topic":"Topic/A", "action":"pub", "access": "allow"}' \
7  http://localhost:8081/api/v4/acl
8
9  ## Return
10 {
11   "data":{
12     "topic":"Topic/A",
13     "result":"ok",
14     "clientid":"emqx_c",
15     "action":"pub",
16     "access":"allow"
17   },
18   "code":0
19 }
```

- **Username ACL**

Parameters (json):

Name	Type	Required	Description
username	String	True	Username
topic	String	True	主题
action	sub/pub/pubsub	True	动作
access	allow/deny	True	是否允许

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.username	String	Username
- data.topic	String	主题
- data.action	String	动作
- data.access	String	是否允许

Examples

```

1  ## Request
2  curl -i \
3  --basic \
4  -u admin:public \
5  -X POST \
6  -d '{"username":"emqx_u", "topic":"Topic/A", "action":"pub", "access": "allow"}' \
7  http://localhost:8081/api/v4/acl
8
9  ## Return
10 {
11   "data":{
12     "topic":"Topic/A",
13     "result":"ok",
14     "username":"emqx_u",
15     "action":"pub",
16     "access":"allow"
17   },
18   "code":0
19 }
```

- **\$all ACL**

Parameters (json):

Name	Type	Required	Description
topic	String	True	主题
action	sub/pub/pubsub	True	动作
access	allow/deny	True	是否允许

Success Response Body (JSON):

name	type	description
code	integer	0
data	object	规则对象
- data.all	string	\$all
- data.topic	string	主题
- data.action	string	动作
- data.access	string	是否允许

Examples

```

1  ## Request
2  curl -i \
3  --basic \
4  -u admin:public \
5  -X POST \
6  -d '{"topic":"Topic/A", "action":"pub", "access": "allow"}' \
7  http://localhost:8081/api/v4/acl
8
9  ## Return
10 {
11   "data":{
12     "topic":"Topic/A",
13     "result":"ok",
14     "all":"$all",
15     "action":"pub",
16     "access":"allow"
17   },
18   "code":0
19 }
```

POST api/v4/acl

批量添加 **ACL** 规则。

Parameters (json):

Name	Type	Required	Description
[0].clientid	String	True	Clientid
[0].topic	String	True	主题
[0].action	sub/pub/pubsub	True	动作
[0].access	allow/deny	True	是否允许
[1].username	String	True	Username
[1].topic	String	True	主题
[1].action	sub/pub/pubsub	True	动作
[1].access	allow/deny	True	是否允许
[2].topic	String	True	主题
[2].action	sub/pub/pubsub	True	动作
[2].access	allow/deny	True	是否允许

Success Response Body (JSON):

name	type	description
code	integer	0
data	object	规则对象
- data.[0].clientid	string	Client ID
- data.[0].topic	string	主题
- data.[0].action	string	动作
- data.[0].access	string	是否允许
- data.[1].username	string	Username
- data.[1].topic	string	主题
- data.[1].action	string	动作
- data.[1].access	string	是否允许
- data.[2].all	string	\$all
- data.[2].topic	string	主题
- data.[2].action	string	动作
- data.[2].access	string	是否允许

Examples

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X POST \
6   -d '[
7     {
8       "clientid": "emqx_c_1",
9       "topic": "Topic/A",
10      "action": "pub",
11      "access": "allow"
12    },
13    {
14      "username": "emqx_u_1",
15      "topic": "Topic/A",
16      "action": "sub",
17      "access": "allow"
18    },
19    {
20      "topic": "Topic/+",
21      "action": "pubsub",
22      "access": "deny"
23    }
24  ]' \
25 http://localhost:8081/api/v4/auth_clientid
26
27 ## Return
28 {
29   "data": [
30     {
31       "clientid": "emqx_c_1",
32       "topic": "Topic/A",
33       "action": "pub",
34       "access": "allow",
35       "result": "ok"
36     },
37     {
38       "username": "emqx_u_1",
39       "topic": "Topic/A",
40       "action": "pub",
41       "access": "allow",
42       "result": "ok"
43     },
44     {
45       "all": "$all",
46       "topic": "Topic/+",
47       "action": "pubsub",
48       "access": "deny"
49     },
50   ],
51   "code": 0
52 }
```

GET api/v4/acl/clientid

查看已经添加的 **ACL** 规则

Query String Parameters:

支持多条件和模糊查询，其包含的查询参数有：

Name	Type	Required	Description
access	Enum	False	是否允许 <code>deny</code> , <code>allow</code>
action	Enum	False	动作 可取值有: <code>pub</code> , <code>sub</code> , <code>pubsub</code>
topic	String	False	MQTT 主题
_like_clientid	String	False	客户端标识符, 子串方式模糊查找

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.[] .clientid	String	Clientid
- data.[] .topic	String	主题
- data.[] .action	Enum	动作 <code>pub</code> , <code>sub</code> , <code>pubsub</code>
- data.[] .access	Enum	是否允许 <code>deny</code> , <code>allow</code>

Examples

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/acl/clientid
7
8 ## Return
9 {
10   "meta": {
11     "page": 1,
12     "limit": 10,
13     "count": 1
14   },
15   "data": [
16     {
17       "clientid": "emqx_c",
18       "topic": "Topic/A",
19       "action": "pub",
20       "access": "allow"
21     },
22     {
23       "clientid": "emqx_c_1",
24       "topic": "Topic/A",
25       "action": "pub",
26       "access": "allow"
27     },
28     {
29       "clientid": "emqx_c_2",
30       "topic": "Topic/A",
31       "action": "pub",
32       "access": "allow"
33     }
34   ],
35   "code": 0
36 }
```

GET api/v4/acl/username

查看已经添加的 **ACL** 规则 **Query String Parameters:**

支持多条件和模糊查询，其包含的查询参数有：

Name	Type	Required	Description
access	Enum	False	权限 deny , allow
action	Enum	False	动作 可取值有： pub , sub , pubsub
topic	String	False	MQTT 主题
_like_username	String	False	客户端标识符，子串方式模糊查找

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.[].username	String	Username
- data.[]topic	String	主题
- data.[]action	Enum	动作 <code>pub</code> , <code>sub</code> , <code>pubsub</code>
- data.[]access	Enum	是否允许 <code>deny</code> , <code>allow</code>

Examples

```

1  ## Request
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X GET \
6      http://localhost:8081/api/v4/acl/username
7
8  ## Return
9  {
10     "meta": {
11         "page": 1,
12         "limit": 10,
13         "count": 1
14     },
15     "data": [
16         {
17             "clientid": "emqx_u",
18             "topic": "Topic/A",
19             "action": "pub",
20             "access": "allow"
21         },
22         {
23             "clientid": "emqx_u_1",
24             "topic": "Topic/A",
25             "action": "pub",
26             "access": "allow"
27         },
28         {
29             "clientid": "emqx_u_2",
30             "topic": "Topic/A",
31             "action": "pub",
32             "access": "allow"
33         }
34     ],
35     "code": 0
36 }
```

GET api/v4/acl/\$all

查看已经添加的 **ACL** 规则

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Object	规则对象
- data.\$all	String	\$all
- data.\$topic	String	主题
- data.\$action	String	动作
- data.\$access	String	是否允许

Examples

```

1  ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/acl/$all
7
8 ## Return
9 {
10   "meta": {
11     "page": 1,
12     "limit": 10,
13     "count": 1
14   },
15   "data": [
16     {
17       "all": "$all",
18       "topic": "Topic/A",
19       "action": "pub",
20       "access": "allow"
21     },
22     {
23       "all": "$all",
24       "topic": "Topic/+",
25       "action": "pubsub",
26       "access": "deny"
27     }
28   ],
29   "code": 0
30 }
```

GET /api/v4/acl/clientid/{clientid}

查看指定的 **ACL** 规则。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	object	规则对象
- data.clientid	string	ClientID
- data.topic	string	主题
- data.action	string	动作
- data.access	string	是否允许

Examples:

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/acl/clientid/emqx_c
7
8 ## Return
9 {
10   "data": {
11     "topic": "Topic/A",
12     "clientid": "emqx_c",
13     "allow": true,
14     "action": "pub"
15   },
16   "code": 0
17 }
```

GET /api/v4/acl/username/{username}

查看指定的 **ACL** 规则。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username

Parameters: 无

Success response body (json):

name	type	description
code	integer	0
data	object	规则对象
- data.username	string	Username
- data.topic	string	主题
- data.action	string	动作
- data.access	string	是否允许

Examples:

```

1  ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X GET \
6   http://localhost:8081/api/v4/acl/usernmae/emqx_u
7
8 ## Return
9 {
10   "data": {
11     "topic": "Topic/A",
12     "username": "emqx_u",
13     "allow": true,
14     "action": "pub"
15   },
16   "code": 0
17 }
```

DELETE /api/v4/acl/clientid/{clientid}/topic/{topic}

删除指定的 **ACL** 规则。

Path Parameters:

Name	Type	Required	Description
clientid	String	True	ClientID
topic	String	True	主题, 可能需要使用 UrlEncode 编码

Parameters: 无

Success response body (json):

name	type	description
code	integer	0

Examples:

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X DELETE \
6   http://localhost:8081/api/v4/acl/clientid/emqx_c/topic/Topic%2fA
7
8 ## Return
9 {"code": 0}

```

DELETE /api/v4/acl/username/{username}/topic/{topic}

删除指定的 **ACL** 规则。

Path Parameters:

Name	Type	Required	Description
username	String	True	Username
topic	String	True	主题, 可能需要使用 UrlEncode 编码

Parameters: 无

Success response body (json):

name	type	description
code	integer	0

Examples:

```

1 ## Request
2 curl -i \
3   --basic \
4   -u admin:public \
5   -X DELETE \
6   http://localhost:8081/api/v4/acl/username/emqx_u/topic/Topic%2fA
7
8 ## Return
9 {"code": 0}

```

DELETE /api/v4/acl/all/\$all/topic/{topic}

删除指定的 **ACL** 规则。

Path Parameters:

Name	Type	Required	Description
topic	String	True	主题, 可能需要使用 UrlEncode 编码

Parameters: 无

Success response body (json):

name	type	description
code	integer	0

Examples:

```

1  ## Request
2  curl -i \
3      --basic \
4      -u admin:public \
5      -X DELETE \
6      http://localhost:8081/api/v4/acl/all/\$all/topic/Topic%2fA
7
8  ## Return
9  {"code": 0}

```

CLI

内置数据库 认证/访问控制 还提供了 `./bin/emqx_ctl` 的管理命令行。

clientid 命令

clientid 命令查询管理内置数据库的 **clientid** 认证。

命令	描述
<code>clientid list</code>	列出 clientid 身份验证规则
<code>clientid add <ClientID> <Password></code>	添加 clientid 身份验证规则
<code>clientid update <ClientID> <NewPassword></code>	更新 clientid 身份验证规则
<code>clientid del <ClientID></code>	删除 clientid 身份验证规则

```
clientid list
```

列出 **clientid** 身份验证规则

```

1  $ ./bin/emqx_ctl clientid list
2  emqx_clientid

```

sh

```
clientid add <ClientID> <Password>
```

添加 **clientid** 身份验证规则

```

1  ./bin/emqx_ctl clientid add emqx_clientid password
2  ok

```

sh

```
clientid update <clientid> <newpassword>
```

更新 **clientid** 身份验证规则

```
1 $ ./bin/emqx_ctl clientid update emqx_clientid new_password
2 ok
```

sh

```
clientid del <ClientID>
```

删除 **clientid** 身份验证规则

```
1 $ ./bin/emqx_ctl clientid del emqx_clientid
2 ok
```

sh

user 命令

user 命令查询管理内置数据库的 **username** 认证。

命令	描述
<code>user list</code>	列出 user 身份验证规则
<code>user add <Username> <Password></code>	添加 user 身份验证规则
<code>user update <Username> <NewPassword></code>	更新 user 身份验证规则
<code>user del <Username></code>	删除 user 身份验证规则

```
user list
```

列出 **username** 身份验证规则

```
1 $ ./bin/emqx_ctl user list
2 emqx_username
```

sh

```
user add <Username> <Password>
```

添加 **username** 身份验证规则

```
1 ./bin/emqx_ctl user add emqx_username password
2 ok
```

sh

```
user update <Username> <NewPassword>
```

更新 **username** 身份验证规则

```
1 $ ./bin/emqx_ctl user update emqx_username new_password
2 ok
```

sh

`user del <Username>`

删除 **username** 身份验证规则

```
1 $ ./bin/emqx_ctl user del emqx_username
2 ok
```

sh

acl 命令

user 命令查询管理内置数据库的访问控制。

命令	描述
<code>acl list clientid</code>	列出 clientid 访问控制规则
<code>acl list username</code>	列出 username 访问控制规则
<code>acl list _all</code>	列出 \$all 访问控制规则
<code>acl show clientid <Clientid></code>	查看 clientid 访问控制详情
<code>acl show username <Username></code>	查看 username 访问控制详情
<code>acl add clientid <Clientid> <Topic> <Action> <Access></code>	增加 clientid 访问控制规则
<code>acl add Username <Username> <Topic> <Action> <Access></code>	增加 username 访问控制规则
<code>acl add _all <Topic> <Action> <Access></code>	增加 \$all 访问控制规则
<code>acl del clientid <Clientid> <Topic></code>	删除 clientid 访问控制规则
<code>acl del username <Username> <Topic></code>	删除 username 访问控制规则
<code>acl del _all <Topic></code>	删除 \$all 访问控制规则

`acl list clientid`

列出 **clientid** 访问控制规则

```
1 $ ./bin/emqx_ctl acl list clientid
2 Acl(clientid = <<"emqx_clientid">> topic = <<"Topic/A">> action = pub access = allow)
```

sh

`acl list username`

列出 **username** 访问控制规则

```
1 $ ./bin/emqx_ctl acl list username
2 Acl(username = <<"emqx_username">> topic = <<"Topic/A">> action = pub access = allow)
```

sh

`acl list _all`列出 **\$all** 访问控制规则

```
1 $ ./bin/emqx_ctl acl list _all
2 Acl($all topic = <<"Topic/A">> action = pub access = allow)
```

sh

`acl show clientid <Clientid>`查看 **clientid** 访问控制详情

```
1 $ ./bin/emqx_ctl acl show clientid emqx_clientid
2 Acl(clientid = <<"emqx_clientid">> topic = <<"Topic/A">> action = pub access = allow)
```

sh

`acl show username <Username>`查看 **username** 访问控制详情

```
1 $ ./bin/emqx_ctl acl show username emqx_username
2 Acl(username = <<"emqx_username">> topic = <<"Topic/A">> action = pub access = allow)
```

sh

`acl aad clientid <Clientid> <Topic> <Action> <Access>`增加 **clientid** 访问控制规则

```
1 $ ./bin/emqx_ctl acl add clientid emqx_clientid Topic/A pub allow
2 ok
```

sh

`acl aad username <Username> <Topic> <Action> <Access>`增加 **username** 访问控制规则

```
1 $ ./bin/emqx_ctl acl add username emqx_username Topic/A pub allow
2 ok
```

sh

`acl aad _all <Topic> <Action> <Access>`增加 **\$all** 访问控制规则

```
1 $ ./bin/emqx_ctl acl add _all Topic/A pub allow
2 ok
```

sh

```
acl del clientid <Clientid> <Topic>
```

删除 **clientid** 访问控制规则

```
1 $ ./bin/emqx_ctl acl del clientid emqx_clientid Topic/A  
2 ok
```

sh

```
acl del username <Username> <Topic>
```

删除 **username** 访问控制规则

```
1 $ ./bin/emqx_ctl acl del username emqx_username Topic/A  
2 ok
```

sh

```
acl del _all <Topic>
```

删除 **\$all** 访问控制规则

```
1 $ ./bin/emqx_ctl acl del _all Topic/A  
2 ok
```

sh

MongoDB 认证/访问控制

MongoDB 认证/访问控制使用外部 **MongoDB** 数据库作为数据源，可以存储大量数据，同时方便与外部设备管理系统集成。

安装MongoDB

打开 **MongoDB** 官网地址: <https://www.mongodb.com/try/download/community>, 选择你需要的版本, 这里我们用 **macOS 4.4.1** 版本:

The screenshot shows the MongoDB download page. At the top, it asks "Choose which type of deployment is best for you". Three options are shown: "Cloud" (the easiest way to run MongoDB), "On-Premises" (download on your own infrastructure, highlighted with a red box and labeled '1'), and "Tools" (do more with your data(base)). Below this, there are two sections: "MongoDB Enterprise Server" and "MongoDB Community Server". The "MongoDB Community Server" section is expanded. It describes MongoDB's offerings and features like In-memory Storage Engine, Advanced Security, and Encrypted Storage Engine. To the right, under "Available Downloads", a dropdown menu is set to "Version 4.4.1 (current)", "Platform macOS", and "Package tgz". A green "Download" button is highlighted with a red box and labeled '2'. Below the download area, links for Current releases & packages, Development releases, Archived releases, Changelog, and Release Notes are visible.

安装后启动**MongoDB**

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

模块管理 4 选择

Recon
Recon

内置访问控制文件
内嵌访问控制文件

上下线通知
客户端上下线通知

MQTT 保留消息
MQTT 保留消息

选择 MongoDB 认证/访问控制模块

选择模块 28 认证/访问控制 协议接入 消息下发 多语言扩展 运维监控 内部模块

MySQL 认证/访问控制
MySQL 认证/访问控制

MongoDB 认证/访问控制
MongoDB 认证/访问控制

JWT 认证
JWT 认证

PKSFile 认证
PKSFile 认证

Redis 认证/访问控制
Redis 认证/访问控制

配置 MongoDB 相关参数

The screenshot shows the 'Parameter Settings' section of the MongoDB module configuration. It includes fields for connection mode ('single'), SRV record ('false'), MongoDB server ('192.168.1.172:27017'), connection pool size ('8'), username, password, database name ('mqtt'), authentication query collection ('mqtt_user'), authentication query field ('username=%u'), access control query collection ('mqtt_acl'), and access control query parameters ('username=%u'). There are also sections for superuser query collections and fields.

SRV 记录的相关说明和使用方式请参考 [规则引擎 - 保存数据到 MongoDB](#)。

点击添加后，模块添加完成：

The screenshot shows the 'Module Management' section of the dashboard. It lists several modules: '上下线通知' (Online/Offline Notification), '内置访问控制文件' (Built-in Access Control File), and 'MongoDB 认证/访问控制' (MongoDB Authentication/Access Control). The 'MongoDB 认证/访问控制' module is highlighted with a red border.

认证集合

```

1  {
2    "username": "user",
3    "password": "password hash",
4    "salt": "password salt",
5    "is_superuser": false,
6    "created": "2020-02-20 12:12:14"
7  }

```

进行身份认证时，**EMQX** 将使用当前客户端信息填充并执行用户配置的认证 **Query**，查询出该客户端在数据库中的认证数据。

MongoDB 支持配置集合名称、认证字段、认证占位符等等参数。

配置项	说明
认证查询集合	认证查询的 MongoDB 集合
认证查询字段名	需要从集合里面查询出来的字段，如果需要查询多个，使用逗号分隔。例如 password,salt
认证条件字段	认证查询的条件，如果需要查询多个，使用逗号分隔。例如 username=%u,clientid=%c

你可以在认证查询占位符中使用以下占位符，执行时 **EMQX** 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: **clientid**
- **%C**: **TLS** 证书公用名（证书的域名或子域名），仅当 **TLS** 连接时有效
- **%d**: **TLS** 证书 **subject**，仅当 **TLS** 连接时有效

你可以根据业务需要调整认证查询，如添加多个查询条件、使用数据库预处理函数，以实现更多业务相关的功能。但是任何情况下认证查询需要满足以下条件：

1. 查询结果中必须包含 **password** 字段，**EMQX** 使用该字段与客户端密码比对
2. 如果启用了加盐配置，查询结果中必须包含 **salt** 字段，**EMQX** 使用该字段作为 **salt**（盐）值
3. **MongoDB** 使用 **findOne** 查询命令，确保你期望的查询结果能够出现在第一条数据中

提示

这是默认配置使用的集合结构，熟悉该插件的使用后你可以使用任何满足条件的集合进行认证。

访问控制集合

```

1  {
2      username: "username",
3      clientid: "clientid",
4      publish: ["topic1", "topic2", ...],
5      subscribe: ["subtop1", "subtop2", ...],
6      pubsub: ["topic/#", "topic1", ...]
7  }

```

MongoDB ACL 一条规则中定义了发布、订阅和发布/订阅的信息，在规则中的都是允许列表。

规则字段说明：

配置项	说明
访问控制查询集合	访问控制查询的 MongoDB 集合
访问控制查询字段名	需要从集合里面查询出来的字段
访问控制条件字段	访问控制查询的条件，支持 and 和 or 操作， and 操作通过逗号分隔，例如： username=%u,clientid=%c, or 操作需要添加多条数据

超级用户查询

进行 **ACL** 鉴权时，**EMQX** 将使用当前客户端信息填充并执行用户配置的超级用户查询，查询客户端是否为超级用户。客户端为超级用户时将跳过 **ACL** 查询。同一个选择器的多个条件时实际查询中使用 **MongoDB and** 查询：

```

1  db.mqtt_user.find({
2      "username": "wivviv"
3      "clientid": "$all"
4  })

```

你可以在查询条件中使用以下占位符，执行时 **EMQX** 将自动填充为客户端信息：

- **%u**: 用户名
- **%c**: **clientid**

你可以根据业务需要调整超级用户查询，如添加多个查询条件、使用数据库预处理函数，以实现更多业务相关的功能。但是任何情况下超级用户查询需要满足以下条件：查询结果中必须包含 **is_superuser** 字段，**is_superuser** 应该显式的为 **true**。**MongoDB** 支持配置集合名称、认证字段、认证占位符等等参数。

配置项	说明
超级用户查询集合	超级用户查询的 MongoDB 集合
超级用户查询字段名	需要从集合里面查询出来的字段
超级用户条件字段	超级用户查询的条件，如果需要查询多个，使用逗号分隔。例如 username=%u,clientid=%c

提示

MongoDB ACL 规则需严格使用上述数据结构。**MongoDB ACL** 中添加的所有规则都是 允许 规则，可以搭配 `etc/emqx.conf` 中 `acl_nomatch = deny` 使用。

加密规则

```

1  ## 不加盐，明文
2  plain
3
4  ## 不加盐，仅做哈希处理
5  sha256
6
7  ## salt 前缀：使用 sha256 加密 salt + 密码 拼接的字符串
8  salt,sha256
9
10 ## salt 后缀：使用 sha256 加密 密码 + salt 拼接的字符串
11 sha256,salt
12
13 ## pbkdf2 with macfun iterations dklen
14 ## macfun: md4, md5, ripemd160, sha, sha224, sha256, sha384, sha512
15 pbkdf2,sha256,1000,20

```

提示

可参考:[加盐规则与哈希方法](#)。

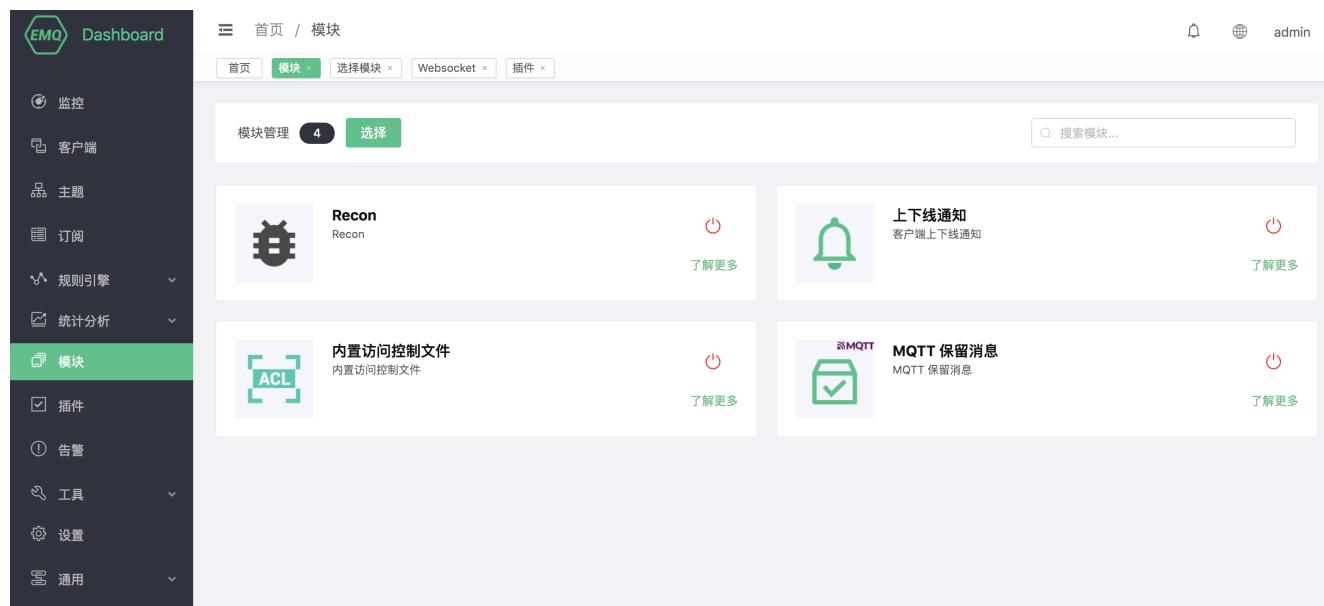
PSKFile 认证

如果希望使用 **PSKFile** 认证，需要将 `emqx.conf` 的 `listener.ssl.external.ciphers` 注释掉，然后配置 `listener.ssl.external.psk_ciphers`：

```
1 #listener.ssl.external.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384, ...
2 listener.ssl.external.psk_ciphers = PSK-AES128-CBC-SHA,PSK-AES256-CBC-SHA,PSK-3DES-EDE-CBC-SHA
3 ,PSK-RC4-SHA
```

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



The screenshot shows the EMQX Dashboard interface. On the left, a dark sidebar menu lists various options like Monitoring, Clients, Topics, Subscriptions, Rules Engine, Statistics, Modules (which is highlighted in green), Plugins, Alarms, Tools, Settings, and General. The main content area has a header "Dashboard" with tabs for Home, Modules (selected), Select Module, WebSocket, and Plugins. Below the header, there's a search bar labeled "Search module...". The main content is titled "Module Management" and shows four modules: "Recon" (with a gear icon), "Online/Offline Notifications" (with a bell icon), "Internal ACL" (with a lock icon), and "MQTT Retain Message" (with a checkmark icon). Each module card includes a status indicator (red circle with a white dot) and a "More" link.

选择 **PSK** 认证

配置相关参数

点击添加后，模块添加完成

The screenshot shows the EMQX Enterprise V4.4 Dashboard. On the left, there's a sidebar with various navigation options like Monitoring, Clients, Themes, Subscriptions, Rule Engines, Statistics, Modules, Plugins, Alarms, Tools, Settings, and General. The 'Modules' option is selected and highlighted in green. The main content area is titled '模块管理' (Module Management) and shows five modules: Recon (Recon), 上下线通知 (Client Online/Offline Notification), 内置访问控制文件 (Built-in Access Control File), MQTT 保留消息 (MQTT Retained Message), and PKSFile 认证 (PKSFile Authentication). The 'PKSFile 认证' module is highlighted with a red border.

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

1	client1:1234	sh
2	client2:abcd	

LDAP 认证/访问控制

LDAP 认证/访问控制使用外部 **OpenLDAP** 服务器作为认证数据源，可以存储大量数据，同时方便与外部设备管理系统集成。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

选择 LDAP 认证/访问控制模块

配置OpenLDAP相关参数

LDAP 认证/访问控制

LDAP 认证/访问控制

配置信息

* LDAP 服务器	* LDAP 服务器端口
<input type="text" value="127.0.0.1"/>	<input type="text" value="389"/>
超期时间(s)	连接池大小
<input type="text" value="30s"/>	<input type="text" value="8"/>
* LDAP 识别名	* LDAP 密码
<input type="text" value="cn=root,dc=emqx,dc=io"/>	<input type="text" value="public"/>
设备识别名	* 对象类
<input type="text" value="cn=root,dc=emqx,dc=io"/>	<input type="text" value="mqttUser"/>
* 用户参数	* 密码
<input type="text" value="uid"/>	<input type="text" value="userPassword"/>
开启 SSL	
<input type="text" value="false"/>	

[取消](#) [添加](#)

最后点击“添加”按钮，模块即可添加成功：

The screenshot shows the EMQX Dashboard interface. On the left, there is a sidebar with various navigation options: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 模块 (Modules), 规则引擎 (Rule Engine), 统计分析 (Statistics), 告警 (Alerts), 工具 (Tools), 设置 (Settings), and 通用 (General). The 'Modules' option is currently selected and highlighted in green.

The main content area is titled '首页 / 模块' (Home / Modules) and shows a 'Module Management' section. It displays a list of modules, with 'LDAP 认证/访问控制' (LDAP Authentication/Access Control) being the most recent addition, indicated by a small '1' in a circle next to its entry. There is also a '选择' (Select) button next to the module entry.

The 'LDAP 认证/访问控制' module entry itself has a small icon, the text 'LDAP 认证/访问控制', and a red power button icon. Below the module entry, there is a link labeled '了解更多' (Learn More).

LDAP Schema

需要在 **LDAP schema** 目录配置数据模型，默认配置下数据模型如下：

/etc/openldap/schema/emqx.schema

```

1 attributetype ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.1.3 NAME 'isEnabled'
2   EQUALITY booleanMatch
3   SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
4   SINGLE-VALUE
5   USAGE userApplications )

6
7 attributetype ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.4.1 NAME ( 'mqttPublishTopic' 'mpt' )
8   EQUALITY caseIgnoreMatch
9   SUBSTR caseIgnoreSubstringsMatch
10  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
11  USAGE userApplications )
12 attributetype ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.4.2 NAME ( 'mqttSubscriptionTopic' 'mst' )
13   EQUALITY caseIgnoreMatch
14   SUBSTR caseIgnoreSubstringsMatch
15   SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
16   USAGE userApplications )
17 attributetype ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.4.3 NAME ( 'mqttPubSubTopic' 'mpst' )
18   EQUALITY caseIgnoreMatch
19   SUBSTR caseIgnoreSubstringsMatch
20   SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
21   USAGE userApplications )

22 objectclass ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.4 NAME 'mqttUser'
23   AUXILIARY
24   MAY ( mqttPublishTopic $ mqttSubscriptionTopic $ mqttPubSubTopic ) )

25 objectclass ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.2 NAME 'mqttDevice'
26   SUP top
27   STRUCTURAL
28   MUST ( uid )
29   MAY ( isEnabled ) )

30 objectclass ( 1.3.6.1.4.1.11.2.53.2.2.3.1.2.3.3 NAME 'mqttSecurity'
31   SUP top
32   AUXILIARY
33   MAY ( userPassword $ userPKCS12 $ pwdAttribute $ pwdLockout ) )

```

编辑 **Idap** 的配置文件 **slapd.conf** 引用 **Schema**：

/etc/openldap/slapd.conf

```

1 include /etc/openldap/schema/core.schema
2 include /etc/openldap/schema/cosine.schema
3 include /etc/openldap/schema/inetorgperson.schema
4 include /etc/openldap/schema/ppolicy.schema
5 include /etc/openldap/schema/emqx.schema
6
7 database bdb
8 suffix "dc=emqx,dc=io"
9 rootdn "cn=root,dc=emqx,dc=io"
10 rootpw {SSHA}eoF7NhNrejVYYyGHqnt+MdKNBh4r1w3W
11
12 directory /etc/openldap/data

```

默认配置下示例数据如下：

```

1 ## create emqx.io
2
3 dn:dc=emqx,dc=io
4 objectclass: top
5 objectclass: dcobject
6 objectclass: organization
7 dc:emqx
8 o:emqx,Inc.
9
10 # create testdevice.emqx.io
11 dn:ou=testdevice,dc=emqx,dc=io
12 objectClass: top
13 objectclass:organizationalUnit
14 ou:testdevice
15
16 dn:uid=mqttuser0001,ou=testdevice,dc=emqx,dc=io
17 objectClass: top
18 objectClass: mqttUser
19 objectClass: mqttDevice
20 objectClass: mqttSecurity
21 uid: mqttuser0001
22 isEnabled: TRUE
23 mqttAccountName: user1
24 mqttPublishTopic: mqttuser0001/pub/1
25 mqttSubscriptionTopic: mqttuser0001/sub/1
26 mqttPubSubTopic: mqttuser0001/publish/1
27 userPassword:: e1NIQX1tbGIzZmF0NDBNS0JUWFVWWndDS21MNzNSLzA9

```

启用 **LDAP** 认证后，你可以通过用户名：**mqttuser0001**，密码：**public** 连接。

LDAP 访问控制配置方式

`mqttPublishTopic` 允许发布的主题(可以配置多个)

`mqttSubscriptionTopic` 允许订阅的主题(可以配置多个)

`mqttPubSubTopic` 允许订阅/发布的主题(可以配置多个)

提示

目前版本仅支持 **OpenLDAP**, 不支持 **Microsoft Active Directory**。

JWT 认证

[JWT](#) 认证是基于 **Token** 的鉴权机制，不依赖服务端保留客户端的认证信息或者会话信息，在持有密钥的情况下可以批量签发认证信息，是最简便的认证方式。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择“添加模块”：

The screenshot shows the EMQX Enterprise Dashboard's 'Monitoring / Modules' page. On the left, a dark sidebar has a green 'Modules' tab selected. The main area is titled 'Module Management' with a count of 4 modules. A search bar at the top right says 'Search module...'. Below are four listed modules, each with a stop button:

- 上下线通知**: MQTT topic for online/offline notifications.
- MQTT 保留消息**: MQTT retain message configuration.
- 内置访问控制文件**: Built-in access control file.
- Recon**: A plugin for GC optimization.

然后选择“认证鉴权”下的“**JWT** 认证”：

JWT 认证
通过 Username 或 Password 携带 JWT 作为认证信息。

JWT 认证提供了以下配置项：

1. 认证来源：客户端连接时存放 **JWT** 的字段，目前支持选择 **username** 或 **password**。
2. 密钥：签发 **JWT** 时使用的密钥。这里将用于验证 **EMQX** 收到的 **JWT** 是否合法，适用于 **HMAC** 算法签发的 **JWT**。
3. 公钥文件：将用于验证 **EMQX** 收到的 **JWT** 是否合法，适用于 **RSA** 或 **ECDSA** 算法签发的 **JWT**。
4. **JWKS** 服务器地址：**EMQX** 将从 **JWKS** 服务器定期查询最新的公钥列表，并用于验证收到的 **JWT** 是否合法，适用于 **RSA** 或 **ECDSA** 算法签发的 **JWT**。
5. 验证声明字段：是否需要验证 **JWT Payload** 中的声明与“声明字段列表”一致。
6. 声明字段列表：用于验证 **JWT Payload** 中的声明是否合法。最常见的用法是，添加一个键为 `username` 值为 `%u` 的键值对，`%u` 作为占位符将在运行时被替换为客户端实际连接时使用的 **Username**，替换后的值将被用于与 **JWT Payload** 的同键声明的值比较，以起到 **JWT** 与 **Username** 一一对应的效果。声明字段列表中目前支持以下两种占位符：
 1. `%u`：将在运行时被替换为客户端连接时使用的 **Username**。
 2. `%c`：将在运行时被替换为客户端连接时使用的 **Client ID**。

注意：**EMQX** 会按照 `Secret`、`Pubkey` 和 `JWKS Addr` 的固定顺序验证 **JWT**。没有配置的字段将被忽略。

The screenshot shows the 'JWT 认证' (JWT Authentication) configuration page. On the left, there's a sidebar with navigation links: 监控 (Monitoring), 客户端 (Client), 主题 (Topic), 订阅 (Subscription), 规则引擎 (Rule Engine), 模块 (Module), 告警 (Alert), 工具 (Tools), 设置 (Settings), and 通用 (General). The '模块' (Module) link is highlighted in green. The main content area has a header 'JWT 认证' with a note: '通过 Username 或 Password 携带 JWT 作为认证信息。' (Authenticates via Username or Password carrying JWT). A '了解更多' (More Information) button is in the top right. Below is a '参数设置' (Parameter Settings) section with fields for '认证来源' (Authentication Source) set to 'password', '密钥' (Key) as '.....', '公钥文件' (Public Key File) with a '选择文件' (Select File) button, 'JWKS 服务器地址' (JWKS Server Address) as an empty input, '验证声明字段' (Verify Statement Field) as 'false', and a '声明字段列表' (Statement Field List) table with one row: '键' (Key) and '值' (Value) both empty. A '添加' (Add) button is at the bottom right.

配置完成后点击“添加”按钮即可成功添加 **JWT** 认证模块。

The screenshot shows the '模块管理' (Module Management) page. The sidebar is identical to the previous screenshot. The main area shows a list of modules: 上下线通知 (Online/Offline Notification), MQTT 保留消息 (MQTT Retain Message), 内置访问控制文件 (Built-in Access Control File), Recon, and JWT 认证 (JWT Authentication). The 'JWT 认证' module is highlighted with a red border. Each module has a status indicator ('停止' - Stop) and a '搜索模块...' (Search Module...) search bar at the top.

认证原理

客户端使用用户名或密码字段携带 **JWT**（取决于模块配置），发起连接时 **EMQX** 使用配置中的密钥、证书进行解密，如果能成功解密则认证成功，否则认证失败。

默认配置下启用 **JWT** 认证后，你可以通过任意用户名+以下密码进行连接：

1

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7ImF1dGhvcii6IndpdndpdiIsInNpdGUiOiJodHRwczovL3pdndpdi5jb20ifSwiZXhwIjoxNTgyMjU1MzYwNjQyMDAwMCwiawF0IjoxNTgyMjU1MzYwfQ.FdyAx2fYahm6h3g47m88ttyNzptzKy_speimyUcma4

提示

上述**JWT Token**仅做测试使用，可根据自己的业务需求用相关工具生成。此处提供一个在线生成工具：

<https://www.jsonwebtoken.io/>。

LwM2M 协议网关

协议介绍

LwM2M 全称是 **Lightweight Machine-To-Machine**，是由 **Open Mobile Alliance(OMA)** 定义的一套适用于物联网的轻量级协议，它提供了设备管理和通讯的功能，尤其适用于资源有限的终端设备。协议可以在 [这里](#) 下载。

LwM2M 基于 **REST** 架构，使用 **CoAP** 作为底层的传输协议，承载在 **UDP** 或者 **SMS** 上，因而报文结构简单小巧，并且在网络资源有限及无法确保设备始终在线的环境里同样适用。

LwM2M 最主要的实体包括 **LwM2M Server** 和 **LwM2M Client**。

LwM2M Server 作为服务器，部署在 **M2M** 服务供应商处或网络服务供应商处。**LwM2M** 定义了两种服务器

- 一种是 **LwM2M BOOTSTRAP SERVER**，**emqx-lwm2m** 插件并未实现该服务器的功能。
- 一种是 **LwM2M SERVER**，**emqx-lwm2m** 实现该服务器在 **UDP** 上的功能，**SMS** 并没有实现。

LwM2M Client 作为客户端，部署在各个 **LwM2M** 设备上。

在 **LwM2M Server** 和 **LwM2M Client** 之间，**LwM2M** 协议定义了4个接口。

1. 引导接口 **Bootstrap**：向 **LwM2M** 客户端提供注册到 **LwM2M** 服务器的必要信息，例如服务器访问信息、客户端支持的资源信息等。
2. 客户端注册接口 **Client Registration**：使 **LwM2M** 客户端与 **LwM2M** 服务器互联，将 **LwM2M** 客户端的相关信息存储在 **LwM2M** 服务器上。只有完成注册后，**LwM2M** 客户端与服务器端之间的通信与管理才成为可能。
3. 设备管理与服务实现接口 **Device Management and Service Enablement**：该接口的主控方为 **LwM2M** 服务器，服务器向客户端发送指令，客户端对指令做出回应并将回应消息发送给服务器。
4. 信息上报接口 **Information Reporting**：允许 **LwM2M** 服务器端向客户端订阅资源信息，客户端接收订阅后按照约定的模式向服务器端报告自己的资源变化情况。

LwM2M 把设备上的服务抽象为 **Object** 和 **Resource**，在 **XML** 文件中定义各种 **Object** 的属性和功能。可以在 [这里](#) 找到 **XML** 的各种定义。

LwM2M 协议预定义了8种 **Object** 来满足基本的需求，分别是：

- **Security** 安全对象
- **Server** 服务器对象
- **Access Control** 访问控制对象
- **Device** 设备对象
- **Connectivity Monitoring** 连通性监控对象
- **Firmware** 固件对象
- **Location** 位置对象
- **Connectivity Statistics** 连通性统计对象

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

模块管理 4 选择

Recon
Recon

内置访问控制文件
内置访问控制文件

上下线通知
客户端上下线通知

MQTT 保留消息
MQTT 保留消息

选择 LwM2M 协议接入网关:

选择模块 28 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

协议接入

LwM2M 接入网关
EMQ X LwM2M 接入网关

COAP 接入网关
EMQ X COAP 接入网关

Stomp 接入网关
EMQ X Stomp 接入网关

TCP 接入网关
EMQ X TCP 接入网关

MQTT-SN 接入网关
EMQ X MQTT-SN 接入网关

JT/T808 接入网关
EMQ X JT/T808 接入网关

配置相关基础参数:

配置信息

- * 最小心跳时间: 1
- * 最大心跳时间: 864000
- * QMode 窗口: 22
- 自动 Observe: false
- * 挂载点: lwm2m/%e/
- * 下行命令主题: dn/#
- * 上行应答主题: up/resp
- * 注册消息主题: up/resp
- * 上行通知主题: up/notify
- * 更新消息主题: up/resp
- * XML 文件路径: etc/lwm2m_xml

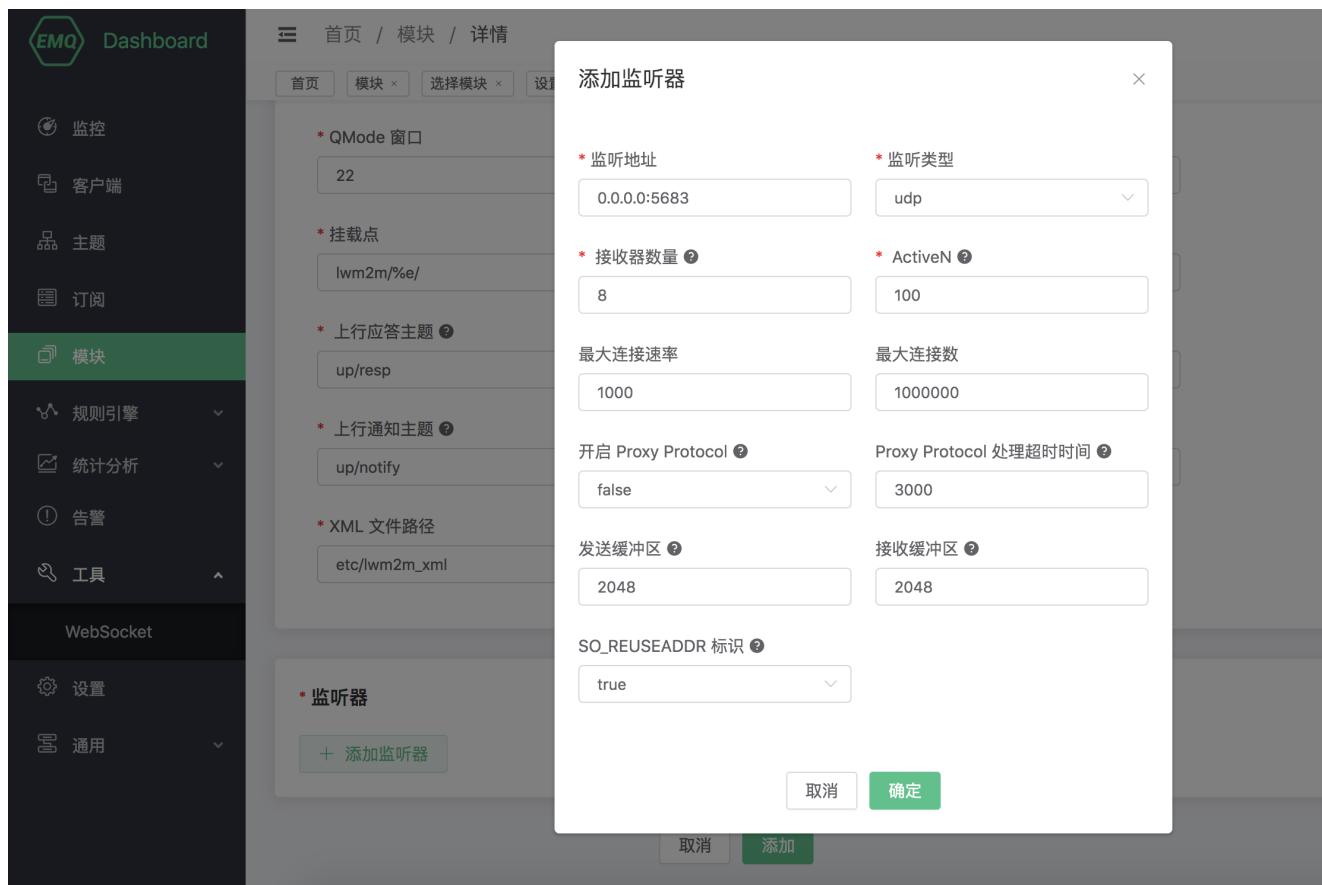
监听器

添加监听端口:

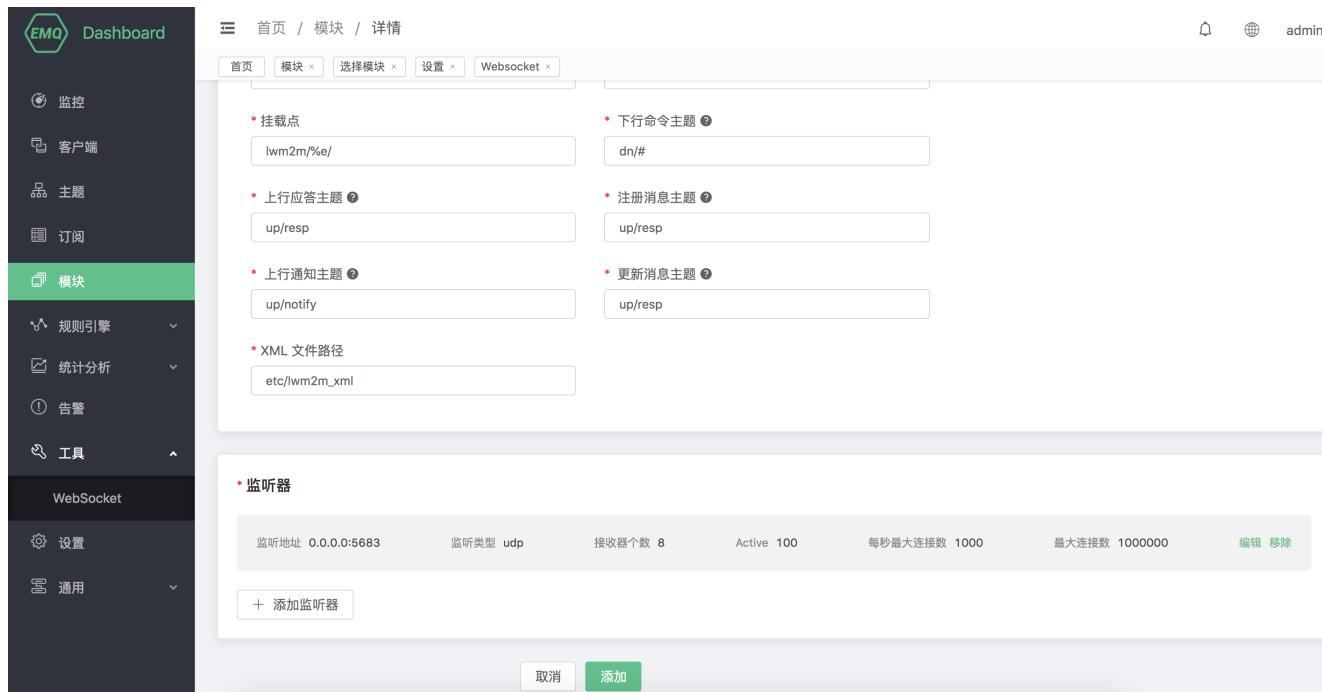
监听器

+ 添加监听器

配置监听参数:



点击确认到配置参数页面:



点击添加后，模块添加完成:

The screenshot shows the EMQX Enterprise V4.4 Dashboard. On the left is a sidebar with various options like Dashboard, Monitoring, Client, Themes, Subscriptions, Modules (which is selected), Rules Engine, Statistics Analysis, Alarms, Tools, WebSocket, and Settings. The main area is titled 'Module Management' and shows three modules: 'Online/Offline Notifications' (客户端上下线通知), 'ACL' (内置访问控制文件), and 'LwM2M Gateway' (LwM2M 接入网关). The 'LwM2M Gateway' module is highlighted with a red box.

EMQX-LWM2M 是 **EMQX** 服务器的一个网关模块，实现了 **LwM2M** 的大部分功能。**MQTT** 客户端可以通过 **EMQX-LWM2M** 访问支持 **LwM2M** 的设备。设备也可以往 **EMQX-LWM2M** 上报 **notification**，为 **EMQX** 后端的服务采集数据。

配置参数

配置项	说明
最小心跳时间	注册/更新允许设置的最小 lifetime , 以秒为单位
最大心跳时间	注册/更新允许设置的最大 lifetime , 以秒为单位
QMode 窗口	QMode 时间窗口, 指示发送到客户机的下行命令经过多长时间后将被缓存, 以秒为单位
自动 Observe	注册成功后, 是否自动 Observe 上报的 objectlist
挂载点	主题前缀
下行命令主题	下行命令主题 %e 表示取值 endpoint name
上行应答主题	上行应答主题 %e 表示取值 endpoint name
注册消息主题	注册消息主题 %e 表示取值 endpoint name
上行通知主题	上行通知主题 %e 表示取值 endpoint name
更新消息主题	更新消息主题 %e 表示取值 endpoint name
XML 文件路径	存放 XML 文件的目录, 这些 XML 用来定义 LwM2M Object

MQTT 和 LwM2M 的转换

从 **MQTT** 客户端可以发送 **Command** 给 **LwM2M** 设备。**MQTT** 到 **LwM2M** 的命令使用如下的 **topic**

```
1 "lwm2m/{?device_end_point_name}/command".
```

其中 **MQTT Payload** 是一个 **json** 格式的字符串，指定要发送的命令，更多的细节请参见 **emqx-lwm2m** 的文档。

LwM2M 设备的回复用如下 **topic** 传送

```
1 "lwm2m/{?device_end_point_name}/response".
```

sh

MQTT Payload 也是一个 **json** 格式的字符串，更多的细节请参见 **emqx-lwm2m** 的文档。

MQTT-SN 协议网关

协议介绍

MQTT-SN 的信令和 **MQTT** 大部分都相同，比如都有 **Will**，都有 **Connect/Subscribe/Publish** 命令。

MQTT-SN 最大的不同是，**Topic** 使用 **TopicId** 来代替，而 **TopicId** 是一个16比特的数字。每一个数字对应一个 **Topic**，设备和云端需要使用 **REGISTER** 命令映射 **TopicId** 和 **Topic** 的对应关系。

MQTT-SN 可以随时更改 **Will** 的内容，甚至可以取消。而 **MQTT** 只允许在 **CONNECT** 时设定 **Will** 的内容，而且不允许更改。

MQTT-SN 的网络中有网关这种设备，它负责把 **MQTT-SN** 转换成 **MQTT**，和云端的 **MQTT Broker** 通信。**MQTT-SN** 的协议支持自动发现网关的功能。

MQTT-SN 还支持设备的睡眠功能，如果设备进入睡眠状态，无法接收 **UDP** 数据，网关将把下行的 **PUBLISH** 消息缓存起来，直到设备苏醒后再传送。

EMQX-SN 是 **EMQX** 的一个网关接入模块，实现了 **MQTT-SN** 的大部分功能，它相当于一个在云端的 **MQTT-SN** 网关，直接和 **EMQ X Broker** 相连。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there is a dark sidebar with various options: 监控, 客户端, 主题, 订阅, 规则引擎, 统计分析, 模块 (which is highlighted in green), 插件, 告警, 工具, 设置, and 通用. The main content area has a header with 首页 / 模块, and tabs for 首页, 模块 (selected), 选择模块, Websocket, and 插件. Below this is a search bar with 搜索模块... and a '模块管理' section showing 4 modules: Recon (status off), 上下线通知 (status off), 内置访问控制文件 (status off), and MQTT 保留消息 (status off). Each module card has a '了解更多' button.

选择 **MQTT-SN** 接入网关模块：

选择模块 28 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

搜索模块...

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

配置相关基础参数：

选择模块 28 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

搜索模块...

配置信息

用户名

密码

* 启动 QoS -1

* 启用数据统计

* 空闲超时时间

预设主题列表

MQTT 主题 **添加** **删除**

* 监听器 **添加监听器**

取消 **添加**

添加监听端口：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left is a dark sidebar with various navigation options. The main area is titled '首页 / 模块 / 详情'. It contains sections for '配置信息' (Configuration), 'MQTT 主题' (MQTT Topics), and '监听器' (Listeners). In the 'Listeners' section, there is a button labeled '+ 添加监听器' (Add Listener) which is highlighted with a red arrow.

配置监听参数:

The screenshot shows the 'Add Listener' configuration dialog. It includes fields for '监听地址' (Listen Address) set to '0.0.0.0:1184', '监听类型' (Listen Type) set to 'udp', '网关ID' (Gateway ID) set to '1', '接收器数量' (Receiver Count) set to '8', 'ActiveN' (ActiveN) set to '100', '最大连接速率' (Max Connection Rate) set to '1000', '最大连接数' (Max Connections) set to '1000000', '开启 Proxy Protocol' (Enable Proxy Protocol) set to 'false', 'Proxy Protocol 处理超时时间' (Proxy Protocol Processing Timeout) set to '3000', '发送缓冲区' (Send Buffer Size) set to '2048', '接收缓冲区' (Receive Buffer Size) set to '2048', and 'SO_REUSEADDR 标识' (SO_REUSEADDR Flag) set to 'true'.

点击确认到配置参数页面:

点击添加后，模块添加完成：

配置参数

配置项	说明
用户名	可选的参数，指定所有 MQTT-SN 连接的用户名，用于 EMQX 鉴权模块
密码	可选的参数，和 username 一起使用于 EMQX 鉴权模块

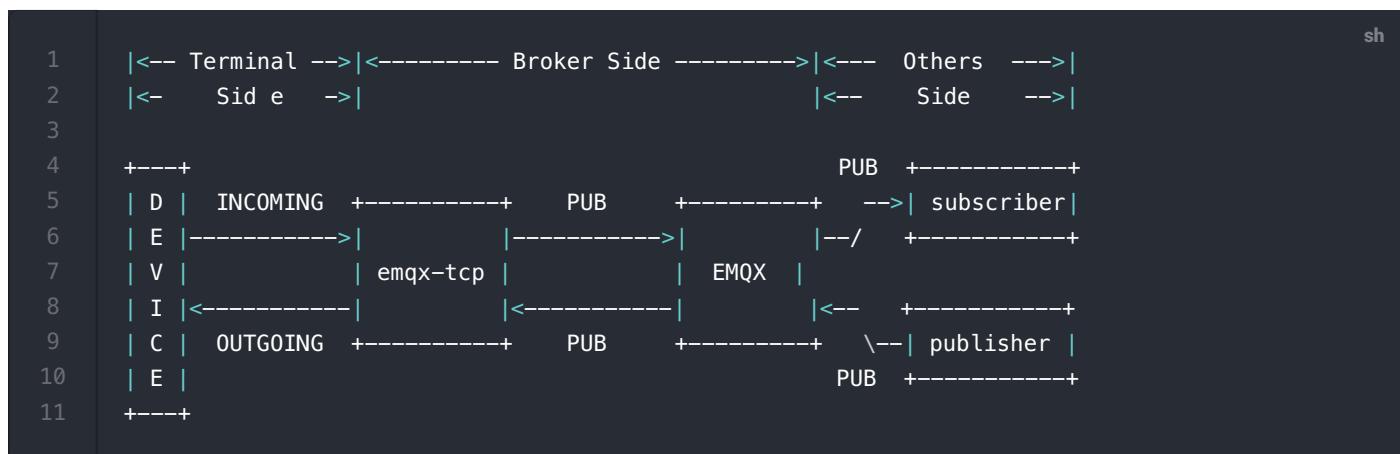
MQTT-SN 客户端库

1. <https://github.com/eclipse/paho.mqtt-sn.embedded-c/>
2. <https://github.com/ty4tw/MQTT-SN>
3. <https://github.com/njh/mqtt-sn-tools>
4. <https://github.com/arobenko/mqtt-sn>

TCP 协议网关

协议介绍

EMQX 提供 **emqx-tcp** 模块作为一个靠近端侧的一个接入模块，按照其功能逻辑和整个系统的关系，将整个消息交换的过程可以分成三个部分：终端侧，平台侧和其它侧：



1. 终端侧，通过本模块定义的 **TCP** 私有协议进行接入，然后实现数据的上报，或者接收下行的消息。
2. 平台侧，主体是 **emqx-tcp** 模块和 **EMQX** 系统。**emqx-tcp** 负责报文的编解码，代理订阅下行主题。实现将上行消息转为 **EMQX** 系统中的 **MQTT** 消息 **PUBLISH** 到整个系统中；将下行的 **MQTT** 消息转化为 **TCP** 私有协议的报文结构，下发到终端。
3. 其它侧，可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅，以接收上行消息。或对发布消息到具体的下行的主题，以发送数据到终端侧。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

模块	状态	描述
Recon	●	Recon
上下线通知	●	客户端上下线通知
内置访问控制文件	●	内置访问控制文件
MQTT 保留消息	●	MQTT 保留消息

选择 **TCP** 私有协议接入网关：

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

配置相关基础参数：

配置信息

* 空闲超时时间 15000	* 启用数据统计 false
* 上行主题 tcp/%c/up	* 下行主题 tcp/%c/dn
* 报文最大长度 65535	* 强制 GC 策略 1000 1MB
强制关闭策略 8000 800MB	

* 监听器

+ 添加监听器

取消 添加

添加监听端口：

配置信息

* 空闲超时时间: 15000 * 启用数据统计: false

* 上行主题: tcp/%c/up * 下行主题: tcp/%c/dn

* 报文最大长度: 65535 * 强制 GC 策略: 1000|1MB

强制关闭策略: 8000|800MB

* 监听器

+ 添加监听器

取消 添加

配置监听参数:

添加监听器

* 监听地址: 0.0.0.0:8090 * 监听类型: tcp

* 接收器数量: 8 * ActiveN: 100

最大连接速率: 1000 最大连接数: 1000000

开启 Proxy Protocol: false Proxy Protocol 处理超时时间: 3000

发送缓冲区: 2048 接收缓冲区: 2048

SO_REUSEADDR 标识: true TCP 连接队列长度: 1000

发送超时时间: 150000 关闭发送超时连接: true

TCP_NODELAY 标识: true

点击确认到配置参数页面:

点击添加后，模块添加完成：

配置参数

配置项	说明
空闲超时时间	闲置时间。超过该时间未收到 CONNECT 帧, 将直接关闭该 TCP 连接
上行主题	上行消息到 EMQ 系统中的消息主题 %c: 接入客户端的 ClientId , %u: 接入客户端的 Username
下行主题	下行主题。上行消息到 EMQ 系统中的消息主题 %c: 接入客户端的 ClientId , %u: 接入客户端的 Username
报文最大长度	最大处理的单个 TCP 私有协议报文大小
强制 GC 策略	强制 GC , 当进程已处理 1000 消息或发送数据超过 1M
强制关闭策略	强制关闭连接, 当进程堆积 8000 消息或堆栈内存超过 800M

私有 TCP 协议 - v1

设计准则

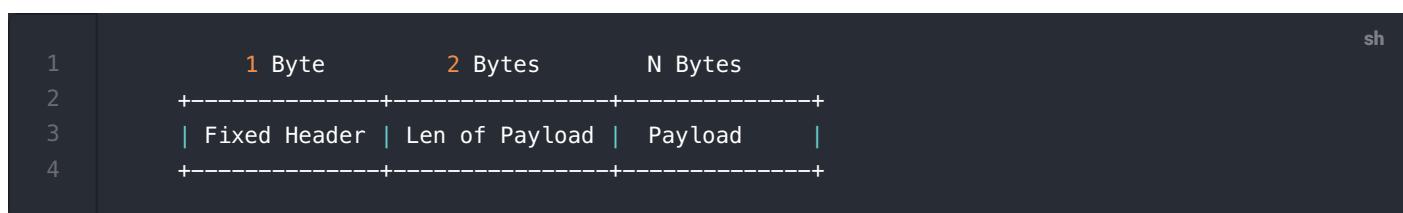
私有 TCP 协议的设计原则有三:

1. 轻量: 尽量减少头部、控制字段的字节大小
2. 简洁: 私有 TCP 协议, 主要功能定位在透传上层应用/协议的数据报文。所以功能应当简洁, 专注透明传输即可
3. 可靠: 保证消息有序可达

报文结构

报文主要有俩部分构成: 固定头部(**FixedHeader**) + 有效载荷(**Payload**)

其中固定头部固定 **1** 字节; 有效载荷为变长, 且前面有 **2** 个字节标识整个 **Payload** 的长度:



部分类型报文中不含 **Payload**; 则整个报文仅只有 **1** 个字节的固定头部

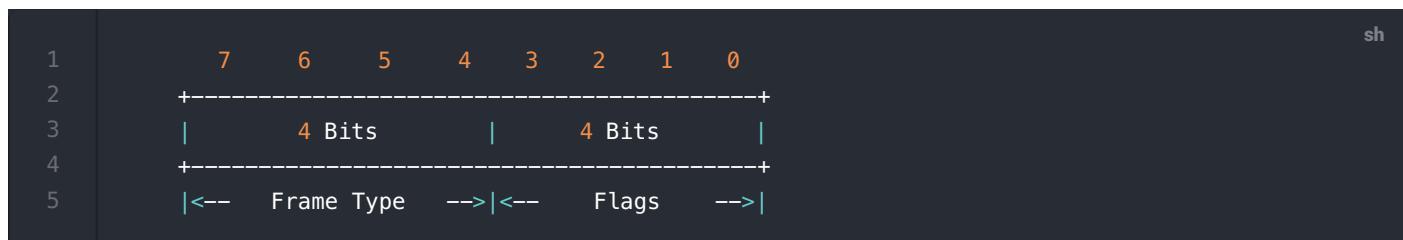
数据类型

本协议设计到的数据类型

Name	Bytes	Description
UINT(x)	x	固定 x 字节的 无符号整型
BIN(n)	变长	带 2 字节标示长度的变长二进制。n 取值为 0 到 65535 内容为空时, 需使用 2 个字节, 来标识长度值0
STR(n)	变长	带 2 字节标示长度的变长字符串。n 取值为 0 到 65535; 空串表达方式同上。
BIN	-	不带长度标识的二进制串

固定头部

固定头部有俩部分组成: 帧类型、标志位



帧类型(**Frame Type**) 有以下几种可选值

Name	Value	Direction of Flow	Description
CONNECT	1	Client --> Server	连接报文
CONNACK	2	Server --> Client	连接应答
DATATRANS	3	Client <==> Server	透明传输
PING	4	Client --> Server	心跳报文
PONG	5	Server --> Client	心跳应答
DISCONNECT	6	Client --> Server	主动断开连接
Reserved	7-15	保留	保留字段

标志位(**Flags**) 针对每个类型的报文，标志位代表的含义都不相同。

报文詳解

CONNECT 帧

连接报文. 帧类型为 **2#0001**. 标志位 **4 Bits** 代表协议 版本号(**Version**) 目前为 **1** 即 **2#0001**。因此 **CONNECT** 帧固定头部为 **0x11**

而，**Payload** 中包含连接用的所有字段。则必须按照以下顺序给出，否则为非法报文，立即断开 **TCP** 链接：

1	Len	Keepalive[x]	ClientId[x]	Username	Password	sh
2	UINT(2)	UINT(1)	STR(n)	STR(n)	STR(n)	

其中 **Keepalive** 和 **ClientId** 为必填字段；**Username** 与 **Password** 可不带。

因此，一个 **Keepalive** 为 **60**; **ClientId** 为 '**abcd**'; **Username** 和 **Password** 均为空时，报文的内容为：

1	0x11 00 07 3c 00 04 61 62 63 64	sh
---	---------------------------------	----

若是 **Username** 和 **Password** 不为空且假设都为 '**abcd**' 的情况下，报文内容为：

1	0x11 00 13 3c 00 04 61 62 63 64 00 04 61 62 63 64 00 04 61 62 63 64	sh
---	---	----

CONNACK 帧

连接应答报文. 帧类型为 **2#0010**. 标志位 **4 Bits** 为应答连接结果(**ACK Code**)。可以为以下枚举值：

Name	Value	Description
SUCCESSFUL	0	连接成功
AUTHFAILED	1	认证失败
ILLEGALVER	2	不支持的协议版本
Reserved	3-15	保留字段

而，**Payload** 字段，为连接应答后传递的 **Message**；该串可为空串。

1	Message
2	STR(n)

sh

所以，当连接成功时，并返回 `Connect Sucessfully` 时，报文内容为：

1	0x20 00 14 43 6f 6e 6e 65 63 74 20 53 75 63 63 65 73 73 66 75 6c 6c 79
---	--

sh

若是，返回 `认证失败` 且 **Message** 为空时::

1	0x21 00 00
---	------------

sh

DATATRANS 帧

数据传输帧。帧类型为 **2#0011**。标志位 前 **2 Bits** 表达 消息质量等级(**Qos**) 目前恒为**0**；后两位为保留位。所以 **DATATRANS** 帧固定头部恒为 **0x30**

Payload 内容为为透传的 数据字段

1	Len	Payload
2	UINT(2)	BIN
3		

sh

注：由于 **Len** 固定为 **2** 字节，所以最大仅支持 **65535** 字节的负载。

因此，如果透传 `abcd` 这个字符串时，该报文的内容为：

1	0x30 00 04 61 62 63 63
2	

sh

PING 帧

心跳帧。帧类型为 **2#0100**。标志位 **Flags** 固定为 **0**。即固定头部 固定为：**0x40**

Payload 为空

因此，一个 **PING** 帧仅有一个字节：

1	0x40
2	

sh

PONG 帧

心跳应答帧。帧类型为 **2#0101**。标志位 **Flags** 固定为 **0**。即固定头部 固定为：**0x50**

Payload 为空

因此，一个 **PONG** 帧仅有一个字节：

1	0x50	sh
---	------	----

DISCONNECT 帧

断开连接帧. 帧类型为 **2#0111**. Flags 为空。即固定头部 固定为: **0x60**

Payload 为空

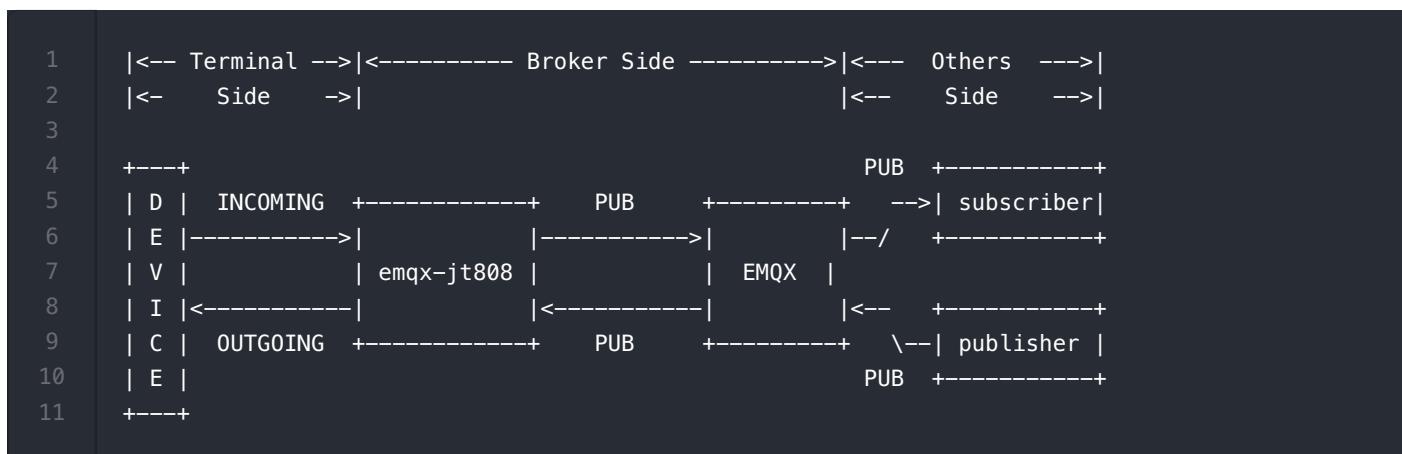
因此, **DISCONNECT** 帧仅有 **1** 个字节:

1	0x60	sh
---	------	----

JT/T808 协议网关

协议介绍

emqx-jt808 做为 **emqx** 的接入网关，按照其功能逻辑和整个系统的关系，将整个消息交换的过程可以分成三个部分：终端侧，平台侧和其它侧：



1. 终端侧：通过 **JT/T 808** 协议进行交换数据，实现不同类型的数据的上报，或者发送下行的消息到终端。
2. 平台侧：**emqx-jt808** 将报文解码后执行 注册/鉴权、或将数据报文 **PUBLISH** 到特定的主题上；代理订阅下行主题，将下行的 **PUBLISH** 消息转化为 **JT/T 808** 协议的报文结构，下发到终端。
3. 其它侧，可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅，以接收上行消息。或对发布消息到具体的下行的主题，以发送数据到终端侧。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. The left sidebar has a green highlighted '模块' (Module) tab. The main area displays a '模块管理' (Module Management) section with four cards: 'Recon' (status off), '上下线通知' (status off), '内置访问控制文件' (status off), and 'MQTT 保留消息' (status off). A search bar at the top right says '搜索模块...'.

选择 **JT/T808** 协议接入网关：

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- STOMP 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

消息下发

配置相关基础参数：

配置信息

* 空闲超时时间	* 允许匿名登录
15000	true
注册服务地址	认证服务地址
下行消息主题格式	上行消息主题格式
jt808/%c/dn	jt808/%c/up
* 是否开启统计	允许最大报文长度
false	8192

* 监听器

+ 添加监听器

取消 添加

添加监听端口：

配置信息

* 空闲超时时间: 15000 * 允许匿名登录: true

注册服务地址: 认证服务地址:

下行消息主题格式: jt808/%c/dn 上行消息主题格式: jt808/%c/up

* 是否开启统计: false 允许最大报文长度: 8192

* 监听器

+ 添加监听器

取消 添加

配置监听参数:

添加监听器

* 监听地址: 0.0.0.0:8090 * 监听类型: tcp

* 接收器数量: 8 * ActiveN: 100

最大连接速率: 1000 最大连接数: 1000000

开启 Proxy Protocol: false Proxy Protocol 处理超时时间: 3000

发送缓冲区: 2048 接收缓冲区: 2048

SO_REUSEADDR 标识: true TCP 连接队列长度: 1000

发送超时时间: 150000 关闭发送超时连接: true

TCP_NODELAY 标识: true

点击确认到配置参数页面:

点击添加后，模块添加完成：

emqx-jt808 实现规定：

- 系统内以手机号作为一个连接的唯一标识，即 **ClientId**

配置参数

emqx-jt808 的实现支持匿名的方式接入认证：

配置项	说明
允许匿名登录	是否允许匿名用户登录
注册服务地址	JT/T808 终端注册的 HTTP 接口地址
认证服务地址	JT/T808 终端接入鉴权的 HTTP 接口地址
下行消息主题格式	上行主题。上行消息到 EMQX 系统中的消息主题 %c : 接入客户端的 ClientId , %p : 接入客户端的 Phone
上行消息主题格式	下行主题。上行消息到 EMQX 系统中的消息主题 %c : 接入客户端的 ClientId , %p : 接入客户端的 Phone
允许最大报文长度	最大处理的单个 JT/T808 协议报文大小

注册及鉴权

注册请求详细格式如下：

注册请求：

```

1   URL: http://127.0.0.1:8991/jt808/registry
2   Method: POST
3   Body:
4     { "province": 58,
5       "city": 59,
6       "manufacturer": "Infinity",
7       "model": "Q2",
8       "license_number": "ZA334455",
9       "dev_id": "xx11344",
10      "color": 3,
11      "phone": "00123456789"
12    }

```

注册应答：

```

1   {
2     "code": 0,
3     "authcode": "132456789"
4   }
5   或:
6   {
7     "code": 1
8   }
12   其中返回码可以为:
13
14   0: 成功
15   1: 车辆已被注册
16   2: 数据库中无该车辆
17   3: 终端已被注册
18   4: 数据库中无该终端

```

鉴权请求：

```

1     URL: http://127.0.0.1:8991/jt808/auth
2     Method: POST
3     Body:
4         { "code": "authcode",
5             "phone", "00123456789"
6         }

```

鉴权应答：

```

1     HTTP 状态码 200: 鉴权成功
2     其他: 鉴权失败

```

注：鉴权请求只会在系统未保存鉴权码时调用（即终端直接发送鉴权报文进行登录系统）

数据上下行

emqx-jt808 中通过配置上下行主题来收发终端消息：

上行

例如：制造商**Id** 为 `abcde` 和 终端**Id** 为 `1234567` 的设备。

首先先使用 **MQTT** 客户端订阅主题 `jt808/abcde1234567/up`：

```

1     $ mosquitto_sub -t jt808/abcde1234567/up

```

例如终端在上报 `数据上行透传(0x0900)` 类型的消息后，订阅端会收到：

```

1     { "body":
2         { "data":"MTIzNDU2",
3             "type":240
4         },
5         "header":
6             { "encrypt":0,
7                 "len":7,
8                 "msg_id":2304,
9                 "msg_sn":4,
10                "phone":"011111111111"
11            }
12     }

```

js

注：透明传输类，**data** 域的内容会 **base64** 编码一次在上报出来

数据下行

同样，以上行的**ID**为例；在终端鉴权成功后，使用 **MQTT** 客户端向该终端下发一个 '数据下行透传(**0x8900**)' 类型的消息：

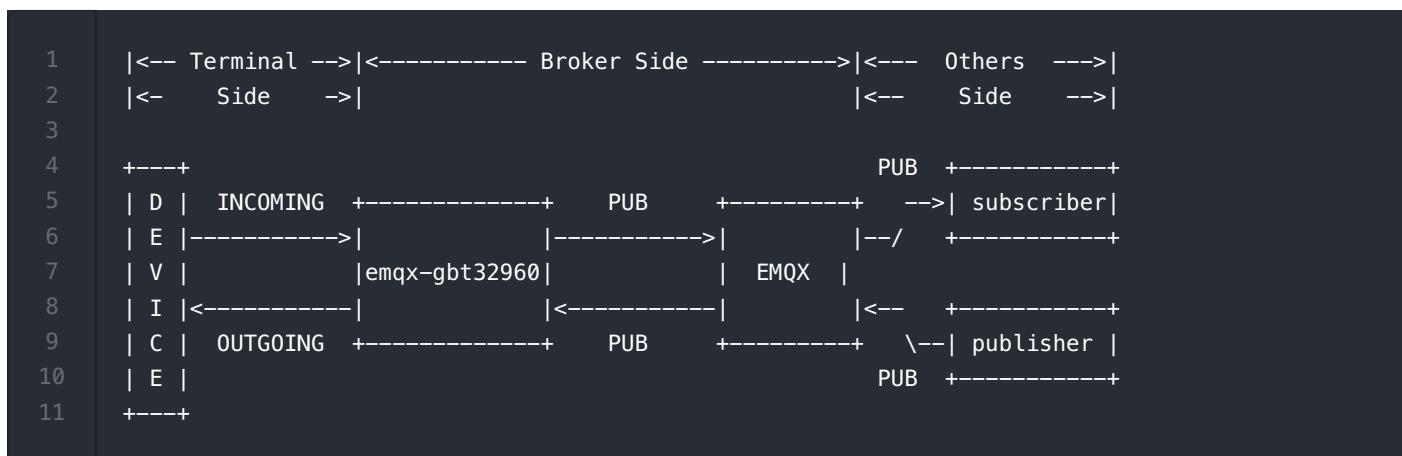
```
1 $ mosquitto_pub -t jt808/abcde1234567/dn -m '{"body":{"data":"MTIzNDU2","type":240},"header":{"sg_id":35072}}'
```

注：下行 **JSON** 中，**header** 中的内容只需要带 **msg_id** 即可；**body** 中的内容根据不同的 **msg_id** 有不同的结构

GB/T 32960 协议网关

协议介绍

emqx-gbt32960 做为 **emqx** 的接入网关，按照其功能逻辑和整个系统的关系，将整个消息交换的过程可以分成三个部分：终端侧，平台侧和其它侧：



1. 终端侧：通过 **GB/T 32960** 协议进行交换数据，实现不同类型的数据的上报，或者发送下行的消息到终端。
2. 平台侧：**emqx-gbt32960** 将报文解码后执行 注册/鉴权、或将数据报文 **PUBLISH** 到特定的主题上；代理订阅下行主题，将下行的 **PUBLISH** 消息转化为 **GB/T 32960** 协议的报文结构，下发到终端。
3. 其它侧，可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅，以接收上行消息。或对发布消息到具体的下行的主题，以发送数据到终端侧。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. The left sidebar is dark-themed with white text and icons. The '模块' (Module) option is highlighted with a green background. The main content area has a light gray background. At the top, there's a navigation bar with tabs: 首页 / 模块 (Home / Modules), 选择模块 (Select Module), Websocket, and 插件 (Plugin). Below the navigation is a search bar labeled '搜索模块...' (Search module...). The main area is titled '模块管理' (Module Management) and shows four available modules:

- Recon**: Represented by a gear icon.
- 上下线通知**: Represented by a bell icon.
- 内置访问控制文件**: Represented by a lock icon.
- MQTT 保留消息**: Represented by a checkmark icon.

Each module card has a red power button icon and a green '了解更多' (More information) link.

选择 **GB/T 32960** 协议接入网关：

监控 / 模块 / 选择

选择模块 27 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

搜索模块...

协议接入

- GB/T 32960 接入网关** EMQ X GB/T32960 接入网关
- CoAP 接入网关** EMQ X CoAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关

配置相关基础参数：

监控 / 模块 / Base.detail

GB/T 32960 接入网关

EMQ X GB/T32960 接入网关

配置信息

* 允许最大报文长度	* 重传间隔时间
8KB	8s
* 最大重传次数	* 消息队列长度
3	10

*** 监听器**

+ 添加监听器

取消

添加监听端口：

MQ 监控 客户端 主题 订阅 规则引擎 模块 插件 告警 工具 设置 通用

监控 / 模块 / Base.detail

GB/T 32960 接入网关

EMQ X GB/T32960 接入网关

配置信息

- * 允许最大报文长度: 8KB
- * 重传间隔时间: 8s
- * 最大重传次数: 3
- * 消息队列长度: 10

监听器

+ 添加监听器

取消 添加

配置监听参数:

MQ 监控 客户端 主题 订阅 规则引擎 模块 插件 告警 工具 设置 通用

监控 / 模块 / Base.detail

GB/T 32960 接入网关

EMQ X GB/T32960 接入网关

配置信息

- * 允许最大报文长度: 8KB
- * 最大重传次数: 3

监听器

+ 添加监听器

添加监听器

* 监听地址: 0.0.0.0:7325

* 监听类型: tcp

* 接收器数量: 8

* ActiveN: 100

最大连接速率: 1000

最大连接数: 1000000

开启 Proxy Protocol: false

Proxy Protocol 处理超时时间: 3s

发送缓冲区: 2KB

接收缓冲区: 2KB

SO_REUSEADDR 标识: true

TCP 连接队列长度: 1000

发送超时时间: 15s

关闭发送超时连接: true

TCP_NODELAY 标识:

点击确认到配置参数页面:

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left is a dark sidebar with various navigation items: 监控, 客户端, 主题, 订阅, 规则引擎, 模块 (highlighted in green), 插件, 告警, 工具, 设置, 通用. The main content area has a title "监控 / 模块 / Base.detail" and a sub-title "EMQ X GB/T32960 接入网关". It contains two sections: "配置信息" and "监听器". In "配置信息", there are four input fields with validation stars: 允许最大报文长度 (8KB), 重传间隔时间 (8s), 最大重传次数 (3), 消息队列长度 (10). In "监听器", it shows a single entry: 监听地址 0.0.0.0:7325, 监听类型 tcp, 接收器个数 8, Active 100, 每秒最大连接数 1000, 最大连接数 1000000. Buttons at the bottom right are "取消" and "添加".

点击添加后，模块添加完成：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. The sidebar is identical to the previous screenshot. The main content area has a title "监控 / 模块" and a sub-title "模块管理". It lists several modules: GB/T 32960 接入网关 (selected, highlighted with a red border), MQTT 保留消息, Recon, 上下线通知, and 内置访问控制文件. Each module entry includes a status icon (red circle with a white symbol), a power button, and a "了解更多" link.

配置参数

配置项	说明
允许最大报文长度	最大处理的单个 GB/T32960 协议报文大小
重传间隔时间	消息重传间隔时间
最大重传次数	最大的消息重传次数
消息队列长度	最大的消息缓存队列长度

约定：

- **Payload** 采用 **Json** 格式进行组装
- **Json Key** 采用大驼峰格式命名

数据上报流程

数据流向: Terminal -> emqx_gbt32960 -> EMQX

车辆登入

Topic: gbt32960/\${vin}/upstream/vlogin

```

1  {
2      "Cmd": 1,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "ICCID": "12345678901234567890",
7          "Id": "C",
8          "Length": 1,
9          "Num": 1,
10         "Seq": 1,
11         "Time": {
12             "Day": 29,
13             "Hour": 12,
14             "Minute": 19,
15             "Month": 12,
16             "Second": 20,
17             "Year": 12
18         }
19     }
20 }
```

车辆登出

Topic: gbt32960/\${vin}/upstream/vlogout

```

1  {
2      "Cmd": 4,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "Seq": 1,
7          "Time": {
8              "Day": 1,
9              "Hour": 2,
10             "Minute": 59,
11             "Month": 1,
12             "Second": 0,
13             "Year": 16
14         }
15     }
16 }
```

信息上报

Topic: gbt32960/\${vin}/upstream/info

不同信息类型上报，格式上只有 **Infos** 里面的对象属性不同，通过 **Type** 进行区分 **Infos** 为数组，代表车载终端

每次报文可以上报多个信息

整车数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "Infos": [
7              {
8                  "AcceleratorPedal": 90,
9                  "BrakePedal": 0,
10                 "Charging": 1,
11                 "Current": 15000,
12                 "DC": 1,
13                 "Gear": 5,
14                 "Mileage": 999999,
15                 "Mode": 1,
16                 "Resistance": 6000,
17                 "SOC": 50,
18                 "Speed": 2000,
19                 "Status": 1,
20                 "Type": "Vehicle",
21                 "Voltage": 5000
22             }
23         ],
24         "Time": {
25             "Day": 1,
26             "Hour": 2,
27             "Minute": 59,
28             "Month": 1,
29             "Second": 0,
30             "Year": 16
31         }
32     }
33 }
```

驱动电机数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          "Infos": [
7              {
8                  "Motors": [
9                      {
10                         "CtrlTemp": 125,
11                         "DCBusCurrent": 31203,
12                         "InputVoltage": 30012,
13                         "MotorTemp": 125,
14                         "No": 1,
15                         "Rotating": 30000,
16                         "Status": 1,
17                         "Torque": 25000
18                     },
19                     {
20                         "CtrlTemp": 125,
21                         "DCBusCurrent": 30200,
22                         "InputVoltage": 32000,
23                         "MotorTemp": 145,
24                         "No": 2,
25                         "Rotating": 30200,
26                         "Status": 1,
27                         "Torque": 25300
28                     }
29                 ],
30                 "Number": 2,
31                 "Type": "DriveMotor"
32             }
33         ],
34         "Time": [
35             "Day": 1,
36             "Hour": 2,
37             "Minute": 59,
38             "Month": 1,
39             "Second": 0,
40             "Year": 16
41         }
42     }
43 }
```

燃料电池数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          {
7              "Infos": [
8                  {
9                      "CellCurrent": 12000,
10                     "CellVoltage": 10000,
11                     "DCStatus": 1,
12                     "FuelConsumption": 45000,
13                     "H_ConcSensorCode": 11,
14                     "H_MaxConc": 35000,
15                     "H_MaxPress": 500,
16                     "H_MaxTemp": 12500,
17                     "H_PressSensorCode": 12,
18                     "H_TempProbeCode": 10,
19                     "ProbeNum": 2,
20                     "ProbeTemps": [120, 121],
21                     "Type": "FuelCell"
22                 }
23             ],
24             "Time": {
25                 "Day": 1,
26                 "Hour": 2,
27                 "Minute": 59,
28                 "Month": 1,
29                 "Second": 0,
30                 "Year": 16
31             }
32         }
33     }
34 }
```

发动机数据

json

```

1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "Infos": [
7              {
8                  "CrankshaftSpeed": 2000,
9                  "FuelConsumption": 200,
10                 "Status": 1,
11                 "Type": "Engine"
12             }
13         ],
14         "Time": {
15             "Day": 1,
16             "Hour": 22,
17             "Minute": 59,
18             "Month": 10,
19             "Second": 0,
20             "Year": 16
21         }
22     }
23 }
```

车辆位置数据

json

```

1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "Infos": [
7              {
8                  "Latitude": 100,
9                  "Longitude": 10,
10                 "Status": 0,
11                 "Type": "Location"
12             }
13         ],
14         "Time": {
15             "Day": 1,
16             "Hour": 22,
17             "Minute": 59,
18             "Month": 10,
19             "Second": 0,
20             "Year": 16
21         }
22     }
23 }
```

极值数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          {
7              "Infos": [
8                  {
9                      "MaxBatteryVoltage": 7500,
10                     "MaxTemp": 120,
11                     "MaxTempProbeNo": 12,
12                     "MaxTempSubsysNo": 14,
13                     "MaxVoltageBatteryCode": 10,
14                     "MaxVoltageBatterySubsysNo": 12,
15                     "MinBatteryVoltage": 2000,
16                     "MinTemp": 40,
17                     "MinTempProbeNo": 13,
18                     "MinTempSubsysNo": 15,
19                     "MinVoltageBatteryCode": 11,
20                     "MinVoltageBatterySubsysNo": 13,
21                     "Type": "Extreme"
22                 }
23             ],
24             "Time": {
25                 "Day": 30,
26                 "Hour": 12,
27                 "Minute": 22,
28                 "Month": 5,
29                 "Second": 59,
30                 "Year": 17
31             }
32         }
33     }
34 }
```

报警数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          {
7              "Infos": [
8                  {
9                      "FaultChargeableDeviceNum": 1,
10                     "FaultChargeableDeviceList": ["00C8"],
11                     "FaultDriveMotorNum": 0,
12                     "FaultDriveMotorList": [],
13                     "FaultEngineNum": 1,
14                     "FaultEngineList": ["006F"],
15                     "FaultOthersNum": 0,
16                     "FaultOthersList": [],
17                     "GeneralAlarmFlag": 3,
18                     "MaxAlarmLevel": 1,
19                     "Type": "Alarm"
20                 }
21             ],
22             "Time": {
23                 "Day": 20,
24                 "Hour": 22,
25                 "Minute": 23,
26                 "Month": 12,
27                 "Second": 59,
28                 "Year": 17
29             }
30         }
31     }
32 }
```

可充电储能装置电压数据

```
1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          "Infos": [
7              {
8                  "Number": 2,
9                  "SubSystems": [
10                     {
11                         "CellsTotal": 2,
12                         "CellsVoltage": [5000],
13                         "ChargeableCurrent": 10000,
14                         "ChargeableSubsysNo": 1,
15                         "ChargeableVoltage": 5000,
16                         "FrameCellsCount": 1,
17                         "FrameCellsIndex": 0
18                     },
19                     {
20                         "CellsTotal": 2,
21                         "CellsVoltage": [5001],
22                         "ChargeableCurrent": 10001,
23                         "ChargeableSubsysNo": 2,
24                         "ChargeableVoltage": 5001,
25                         "FrameCellsCount": 1,
26                         "FrameCellsIndex": 1
27                     }
28                 ],
29                 "Type": "ChargeableVoltage"
30             }
31         ],
32         "Time": {
33             "Day": 1,
34             "Hour": 22,
35             "Minute": 59,
36             "Month": 10,
37             "Second": 0,
38             "Year": 16
39         }
40     }
41 }
```

可充电储能装置温度数据

```

1  {
2      "Cmd": 2,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": [
6          "Infos": [
7              {
8                  "Number": 2,
9                  "SubSystems": [
10                     {
11                         "ChargeableSubsysNo": 1,
12                         "ProbeNum": 10,
13                         "ProbesTemp": [0, 0, 0, 0, 0, 0, 0, 0, 19, 136]
14                     },
15                     {
16                         "ChargeableSubsysNo": 2,
17                         "ProbeNum": 1,
18                         "ProbesTemp": [100]
19                     }
20                 ],
21                 "Type": "ChargeableTemp"
22             }
23         ],
24         "Time": {
25             "Day": 1,
26             "Hour": 22,
27             "Minute": 59,
28             "Month": 10,
29             "Second": 0,
30             "Year": 16
31         }
32     }
33 }

```

数据补发

Topic: `gbt32960/${vin}/upstream/reinfo`

数据格式: 略 (与实时数据上报相同)

数据下发流程

请求数据流向: `EMQX -> emqx_gbt32960 -> Terminal`

应答数据流向: `Terminal -> emqx_gbt32960 -> EMQX`

下行主题: `gbt32960/${vin}/dnstream` 上行应答主题: `gbt32960/${vin}/upstream/response`

参数查询

Req:

json

```

1  {
2      "Action": "Query",
3      "Total": 2,
4      "Ids": ["0x01", "0x02"]
5  }

```

Response:

json

```

1  {
2      "Cmd": 128,
3      "Encrypt": 1,
4      "Vin": "1G1BL52P7TR115520",
5      "Data": {
6          "Total": 2,
7          "Params": [
8              {"0x01": 6000},
9              {"0x02": 10}
10         ],
11         "Time": {
12             "Day": 2,
13             "Hour": 11,
14             "Minute": 12,
15             "Month": 2,
16             "Second": 12,
17             "Year": 17
18         }
19     }
20 }

```

参数设置

Req:

json

```

1  {
2      "Action": "Setting",
3      "Total": 2,
4      "Params": [{"0x01": 5000},
5                  {"0x02": 200}]
6  }

```

Response:

json

```

1 // fixme? 终端是按照这种方式返回?
2 {
3     "Cmd": 129,
4     "Encrypt": 1,
5     "Vin": "1G1BL52P7TR115520",
6     "Data": {
7         "Total": 2,
8         "Params": [
9             {"0x01": 5000},
10            {"0x02": 200}
11        ],
12        "Time": {
13            "Day": 2,
14            "Hour": 11,
15            "Minute": 12,
16            "Month": 2,
17            "Second": 12,
18            "Year": 17
19        }
20    }
21 }

```

终端控制

命令的不同, 参数不同; 无参数时为空

远程升级: **Req:**

json

```

1 {
2     "Action": "Control",
3     "Command": "0x01",
4     "Param": {
5         "DialingName": "hz203",
6         "Username": "user001",
7         "Password": "password01",
8         "Ip": "192.168.199.1",
9         "Port": 8080,
10        "ManufacturerId": "BMWA",
11        "HardwareVer": "1.0.0",
12        "SoftwareVer": "1.0.0",
13        "UpgradeUrl": "ftp://emqtt.io/ftp/server",
14        "Timeout": 10
15    }
16 }

```

车载终端关机:

json

```

1 {
2     "Action": "Control",
3     "Command": "0x02"
4 }

```

...

车载终端报警:

```
1  {
2      "Action": "Control",
3      "Command": "0x06",
4      "Param": {"Level": 0, "Message": "alarm message"}
5 }
```

json

CoAP 协议网关

CoAP 协议网关为 **EMQX** 提供了 **CoAP** 协议的接入能力。它允许符合某种定义的 **CoAP** 消息格式向 **EMQX** 执行发布，订阅，和接收消息等操作。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there is a dark sidebar with various menu items: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 规则引擎 (Rule Engine), 统计分析 (Statistics), 模块 (Modules) [highlighted in green], 插件 (Plugins), 告警 (Alarms), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area has a header '首页 / 模块' and a sub-header '模块管理'. It displays several modules: Recon (Recon), 上下线通知 (Online/Offline Notification), 内置访问控制文件 (Built-in Access Control File), and MQTT 保留消息 (MQTT Retained Message). Each module card includes a '选择' (Select) button and a '了解更多' (More Information) link.

点击“选择”，然后选择“**CoAP** 接入网关”：

This screenshot shows the '选择模块' (Select Module) page within the EMQX Dashboard. The left sidebar remains the same as the previous screenshot. The main area has a header '首页 / 模块 / 选择' and a sub-header '选择模块'. It lists several protocol gateway modules under the '协议接入' (Protocol Gateway) category: COAP 接入网关 (COAP Gateway), Stomp 接入网关 (Stomp Gateway), LwM2M 接入网关 (LwM2M Gateway), MQTT-SN 接入网关 (MQTT-SN Gateway), TCP 接入网关 (TCP Gateway), JT/T808 接入网关 (JT/T808 Gateway), Kafka 消费组 (Kafka Consumer Group), and MQTT Subscriber (MQTT Subscriber). The 'COAP 接入网关' module is highlighted with a red box.

配置相关基础参数：

COAP 接入网关
EMQ X COAP 接入网关

配置信息

- * 启用数据统计:

监听器

+ 添加监听器

取消 添加

添加监听端口：

COAP 接入网关
EMQ X COAP 接入网关

配置信息

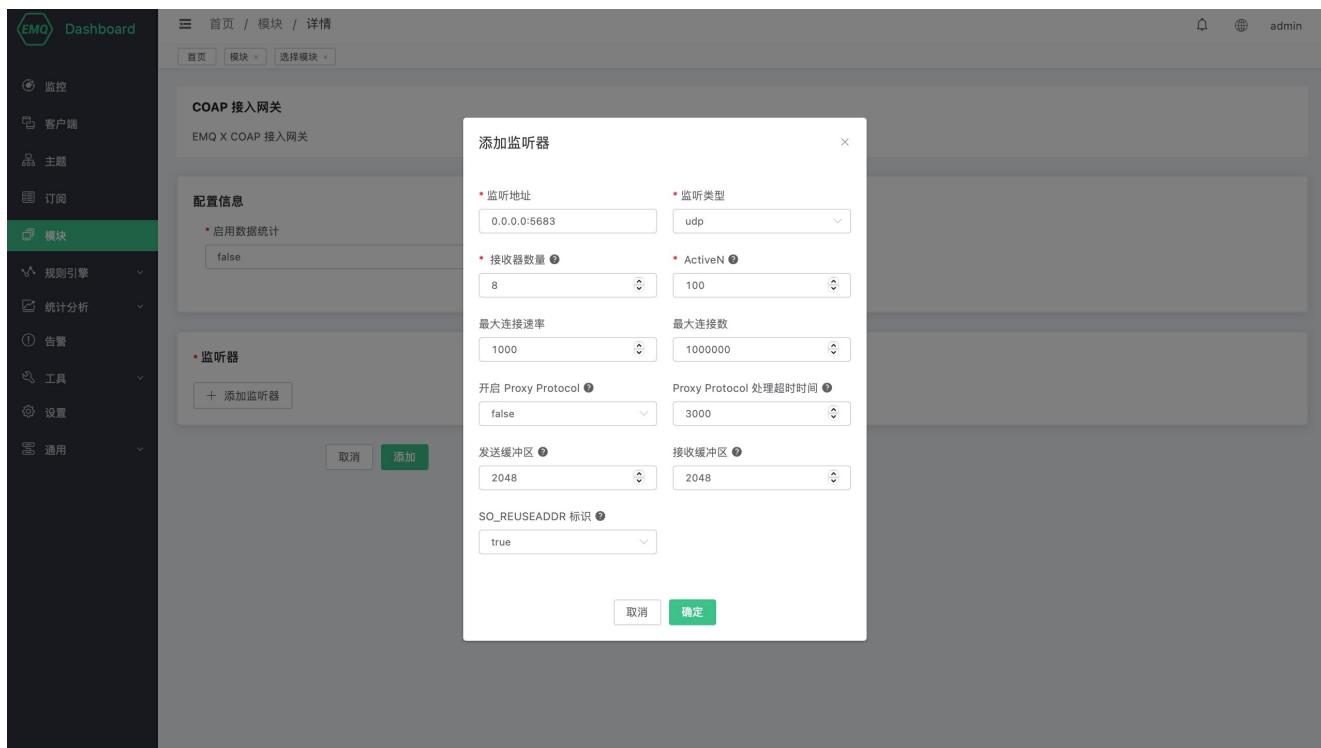
- * 启用数据统计:

监听器

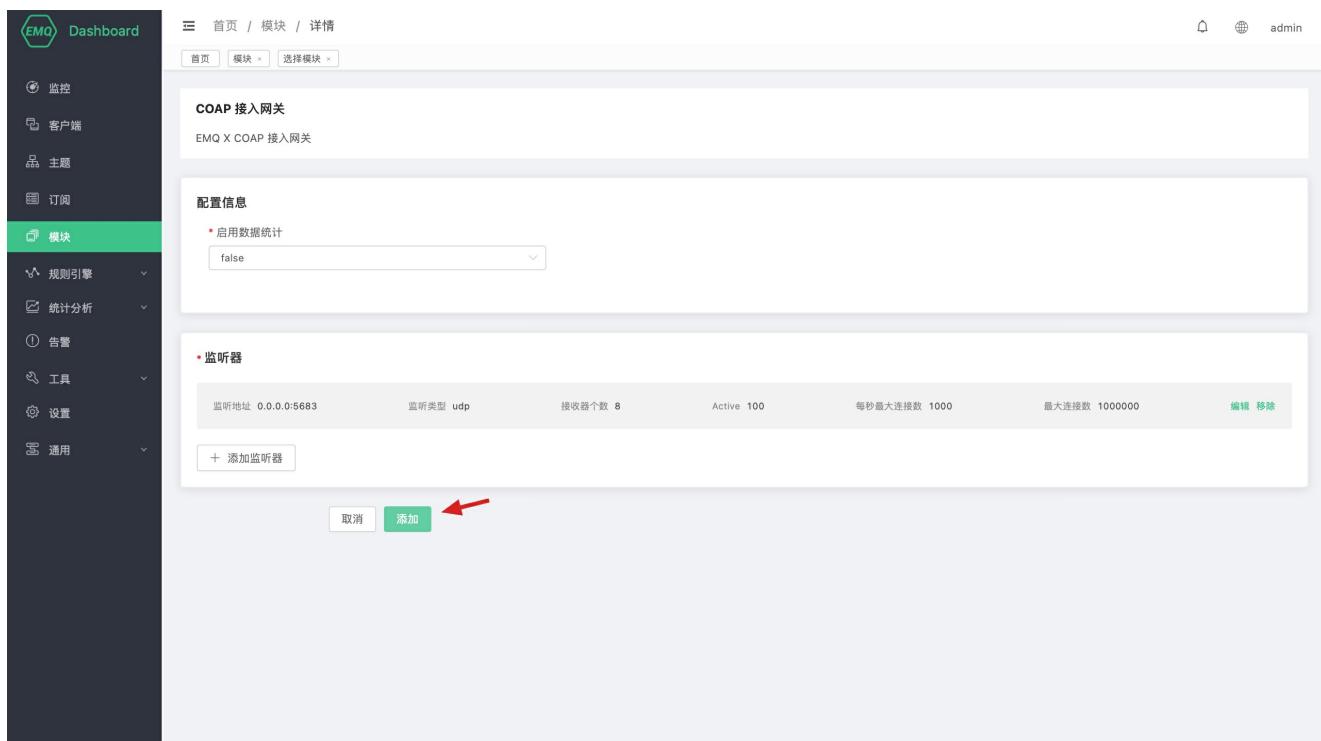
+ 添加监听器

取消 添加

配置监听参数：



点击“确定”完成监听器的配置，然后点击“添加”完成模块的创建：



使用示例

客户端

[libcoap](#) 是一个非常易用的 CoAP 客户端库，此处我们使用它作为 CoAP 客户端来测试 EMQX CoAP 接入网关的功能。

```

1 git clone http://github.com/obgm/libcoap
2 cd libcoap
3 ./autogen.sh
4 ./configure --enable-documentation=no --enable-tests=no
5 make

```

PUBLISH 示例

使用 `libcoap` 发布一条消息：

```

1 libcoap/examples/coap-client -m put -e 1234 "coap://127.0.0.1/mqtt/topic1?c=client1&u=tom&p=secret"

```

- 主题名称为：**"topic1"** (不是 **"/topic1"**)
- Client ID** 为：**"client1"**
- 用户名为：**"tom"**
- 密码为：**"secret"**
- Payload** 为：**"1234"**

SUBSCRIBE 示例

使用 `libcoap` 订阅一个主题：

```

1 libcoap/examples/coap-client -m get -s 10 "coap://127.0.0.1/mqtt/topic1?c=client1&u=tom&p=secret"

```

- 主题名称为：**"topic1"** (不是 **"/topic1"**)
- Client ID** 为：**"client1"**
- 用户名为：**"tom"**
- 密码为：**"secret"**
- 订阅的持续时间为：**10** 秒

在这个期间，如果 `topic1` 主题上有消息产生，`libcoap` 便会收到该条消息。

通信接口说明

CoAP Client Observe Operation

在 **EMQX CoAP** 接入网关中，可以使用 **CoAP** 的 **Observe** 操作实现一个订阅主题的操作：

```

1 GET coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password} with OBSERVE=0

```

- 路径中的 **"mqtt"** 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项

- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误
- 订阅的 **QoS** 等级恒定为 1

CoAP Client Unobserve Operation

使用 **Unobserve** 操作，取消订阅主题：

```
1 GET coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password} with OBSERVE=1
```

- 路径中的 "mqtt" 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项
- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误

CoAP Client Notification Operation

接入网关会将订阅主题上收到的消息，以 **observe-notification** 的方式投递到 **CoAP** 客户端：

- 它的 **payload** 正是 **MQTT** 消息中的的 **payload**
- payload** 数据类型为 "**application/octet-stream**"

CoAP Client Publish Operation

使用 **CoAP** 的 **PUT** 命令执行一次 **PUBLISH** 操作：

```
1 PUT coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password}
```

- 路径中的 "mqtt" 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项
- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误
- payload** 可以是任何二进制数据
- payload** 数据类型为 "**application/octet-stream**"
- 发布信息将以 **qos0** 发送

CoAP Client 保活

设备应定期发出 **GET** 命令，作为 **ping** 操作保持会话在线

```
1   GET  coap://localhost/mqtt/{any_topicname}?c={clientid}&u={username}&p={password}
```

- 路径中的 "mqtt" 为必填项
- 将 {topicname}、{clientid}、{username} 和 {password} 替换为你的真实值
- {topicname} 和 {clientid} 为必填项
- 如果 clientid 不存在, 将返回 "bad_request"
- URI 中的 {topicname} 应该用 percent-encoded, 以防止特殊字符, 如 + 和 #
- {username} 和 {password} 是可选的
- 如果 {username} 和 {password} 不正确, 将返回一个 unauthorized 错误
- 客户端应该定期做 keepalive 工作, 以保持会话在线, 尤其是在 NAT 网络中的设备

备注

CoAP 接入网关不支持 POST 和 DELETE 方法。

在 URI 中的主题名称必须先经过 URI 编码处理(参考: [RFC 7252 - section 6.4](#))

CoAP URI 中的 **ClientId**, **Username**, **Password**, **Topic**是 MQTT 中的概念。也就是说, CoAP 接入网关是通过借用 MQTT 中的名词概念, 试图将 CoAP 信息融入到 MQTT 系统中。

EMQX 的 认证, 访问控制, 钩子等功能也适用于 CoAP 接入网关。比如:

- 如果 用户名/密码 没有被授权, CoAP 客户端就会得到一个 unauthorized 的错误
- 如果 用户名/客户端ID 不允许发布特定的主题, CoAP 消息实际上会被丢弃, 尽管 CoAP 客户端会从接入网关上得到一个 Acknowledgement
- 如果一个 CoAP 消息被发布, 'message.publish' 钩子也能够捕获这个消息

Well-known locations

CoAP 接入网关的 well-known 发现恒定的返回 ","

例如:

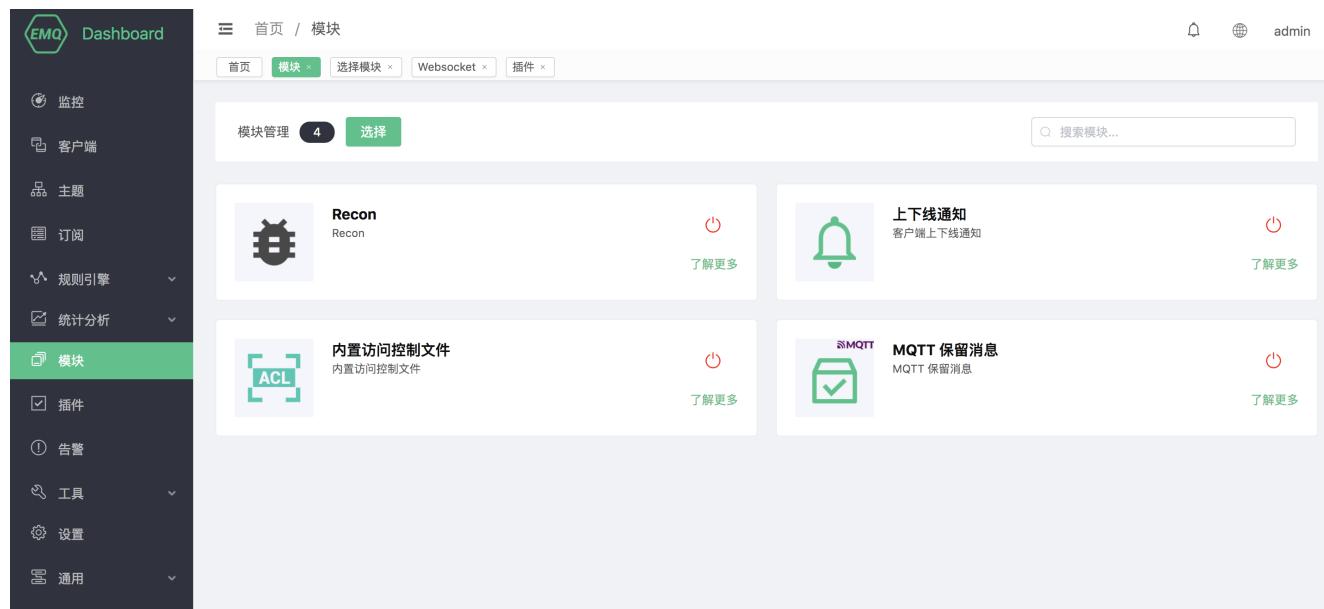
```
1   libcoap/examples/coap-client -m get "coap://127.0.0.1/.well-known/core"
```

Stomp 协议网关

Stomp 协议网关为 EMQX 提供了 Stomp 协议的接入能力。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



The screenshot shows the EMQX Dashboard interface. On the left, a dark sidebar menu lists various options: 监控 (Monitoring), 客户端 (Clients), 品牌 (Brands), 订阅 (Subscriptions), 规则引擎 (Rule Engine), 统计分析 (Statistics), 模块 (Modules) [highlighted in green], 插件 (Plugins), 告警 (Alerts), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area is titled '模块管理' (Module Management) and shows four modules listed: 'Recon' (Recon), '上下线通知' (Online/Offline Notification), '内置访问控制文件' (Built-in Access Control File), and 'MQTT 保留消息' (MQTT Retained Message). A green '选择' (Select) button is located at the top right of the module list. The top navigation bar includes '首页' (Home), '模块' (Modules), '选择模块' (Select Module), 'WebSocket' (WebSocket), and '插件' (Plugins). The top right corner shows user information: admin.

点击“选择”，然后选择“**Stomp** 接入网关”：

协议接入

- CoAP 接入网关 (选择了解更多)
- Stomp 接入网关** (选择了解更多) (highlighted)
- LwM2M 接入网关 (选择了解更多)
- MQTT-SN 接入网关 (选择了解更多)
- TCP 接入网关 (选择了解更多)
- JT/T808 接入网关 (选择了解更多)

消息下发

- Kafka 消费组 (选择了解更多)
- MQTT Subscriber (选择了解更多)
- Pulsar 消费组 (选择了解更多)

多语言扩展

- 多语言扩展协议接入 (选择了解更多)
- 多语言扩展钩子 (选择了解更多)

配置相关基础参数：

Stomp 接入网关

配置信息

- 最大报文头部数量: 10
- 最大报文长度: 1024
- 报文体最大长度: 8192
- 默认密码: guest
- 允许匿名登录: true
- 默认用户名: guest

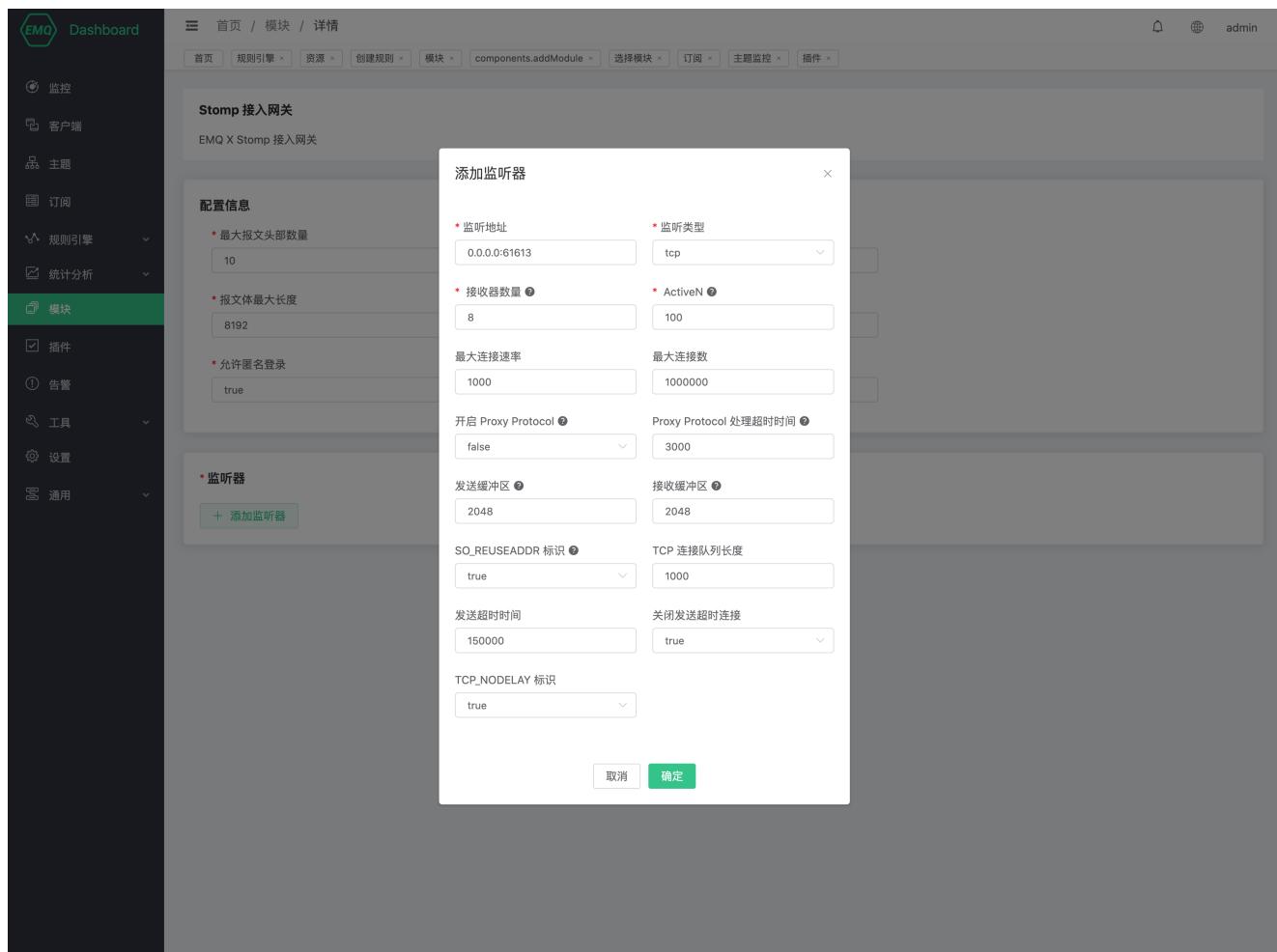
监听器

+ 添加监听器

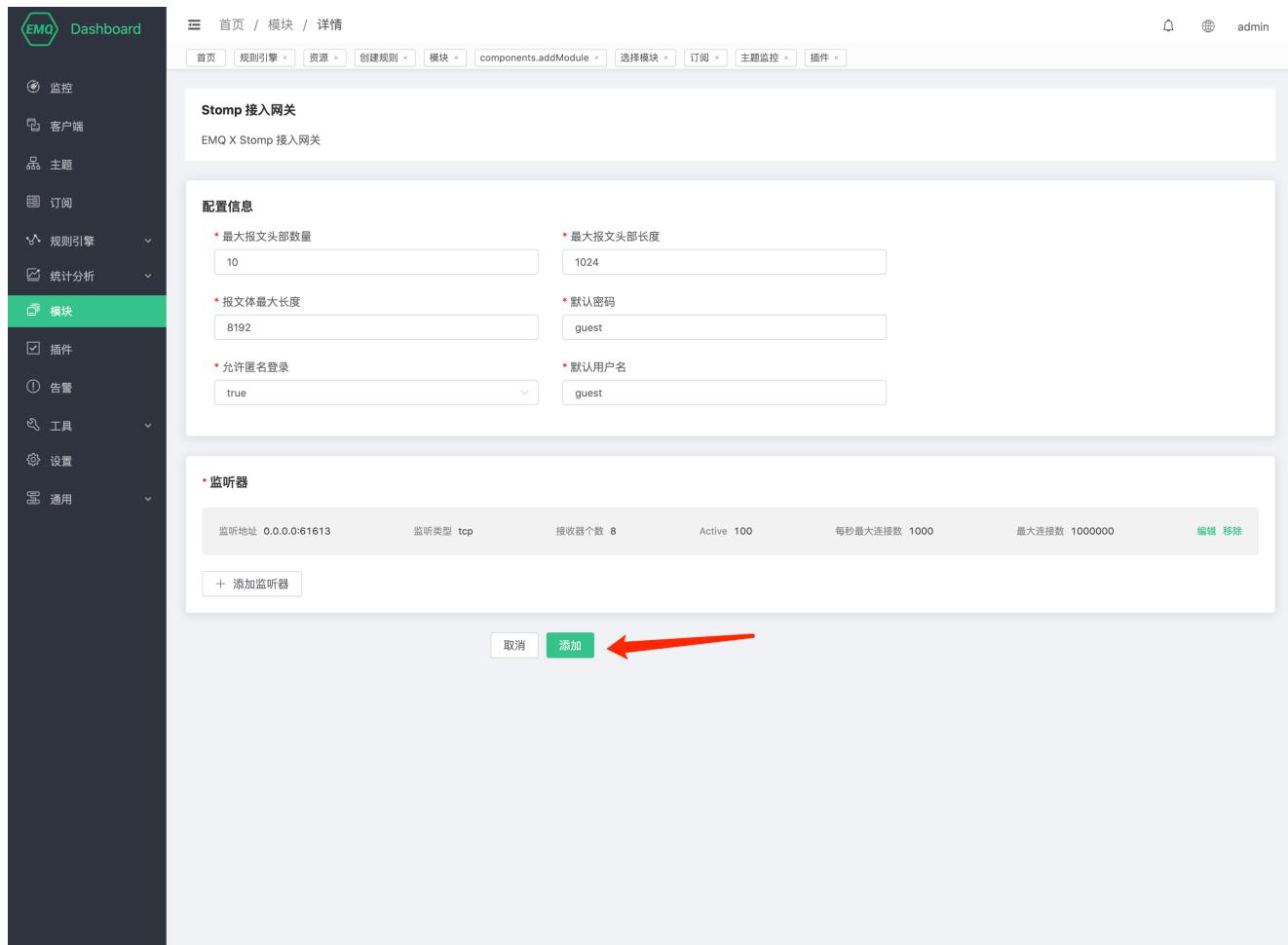
添加监听端口：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left is a dark sidebar with a green header 'Dashboard' and various menu items: 监控, 客户端, 主题, 订阅, 规则引擎, 统计分析, 模块 (highlighted in green), 插件, 告警, 工具, 设置, 通用. The main area has a title 'Stomp 接入网关' and a subtitle 'EMQ X Stomp 接入网关'. Below this is a '配置信息' section with several input fields: 最大报文头部数量 (10), 最大报文头部长度 (1024), 报文体最大长度 (8192), 默认密码 (guest), 允许匿名登录 (true), 和 默认用户名 (guest). At the bottom of this section is a 'Listeners' section with a button '+ 添加监听器' (highlighted with a red arrow) and a '添加' (Add) button. At the very bottom are '取消' (Cancel) and '添加' (Add) buttons.

配置监听参数：



点击“确定”完成监听器的配置，然后点击“添加”完成模块的创建：



配置参数

配置项	说明
最大报文头部数量	Stomp frame headers 的最大数量
最大报文头部长度	Stomp frame headers 的最大长度
报文体最大长度	Stomp frame body 的最大长度
允许匿名登录	是否允许匿名登录
默认用户名	指定 Stomp 模块登录使用的 Username
默认密码	指定 Stomp 模块登录使用的 Password

Kafka 消费组

Kafka 消费组使用外部 **Kafka** 作为消息队列，可以从 **Kafka** 中消费消息转换成为 **MQTT** 消息发布在 **emqx** 中。

搭建 **Kafka** 环境，以 **MacOS X** 为例：

```

1 wget https://archive.apache.org/dist/kafka/2.8.0/kafka_2.13-2.8.0.tgz
2
3 tar -xzf kafka_2.13-2.8.0.tgz
4
5 cd kafka_2.13-2.8.0
6
7 # 启动 Zookeeper
8 ./bin/zookeeper-server-start.sh config/zookeeper.properties
9 # 启动 Kafka
10 ./bin/kafka-server-start.sh config/server.properties

```

提示

Kafka 消费组不支持 **Kafka0.9** 以下版本

创建资源之前，需要提前创建 **Kafka** 主题，不然会提示错误

创建 **Kafka** 的主题：

```

1 $ ./bin/kafka-topics.sh --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic testTopic --create

```

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there is a sidebar with various navigation options: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 规则引擎 (Rules Engine), 统计分析 (Statistics), 模块 (Modules) [highlighted in green], 插件 (Plugins), 告警 (Alarms), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area is titled '模块管理' (Module Management) and shows four available modules: 'Recon' (status red), '上下线通知' (status green), '内置访问控制文件' (status red), and 'MQTT 保留消息' (status red). Each module has a '了解更多' (More Information) link. At the top of the main area, there are tabs for 首页 (Home), 模块 (Modules) [selected], 选择模块 (Select Module), Websocket, and 插件 (Plugins).

选择 **Kafka** 消费组模块：

127.0.0.1:18083/#/modules

The screenshot shows the EMQX Enterprise dashboard with the 'Modules' section selected. The left sidebar has a green header '模块' (Module) and includes links for Rules Engine, Statistics, Alarms, Tools, Settings, and General. The main area shows a 'Module Management' section with a button '选择' (Select) and a message '暂无数据' (No data available).

填写相关参数：

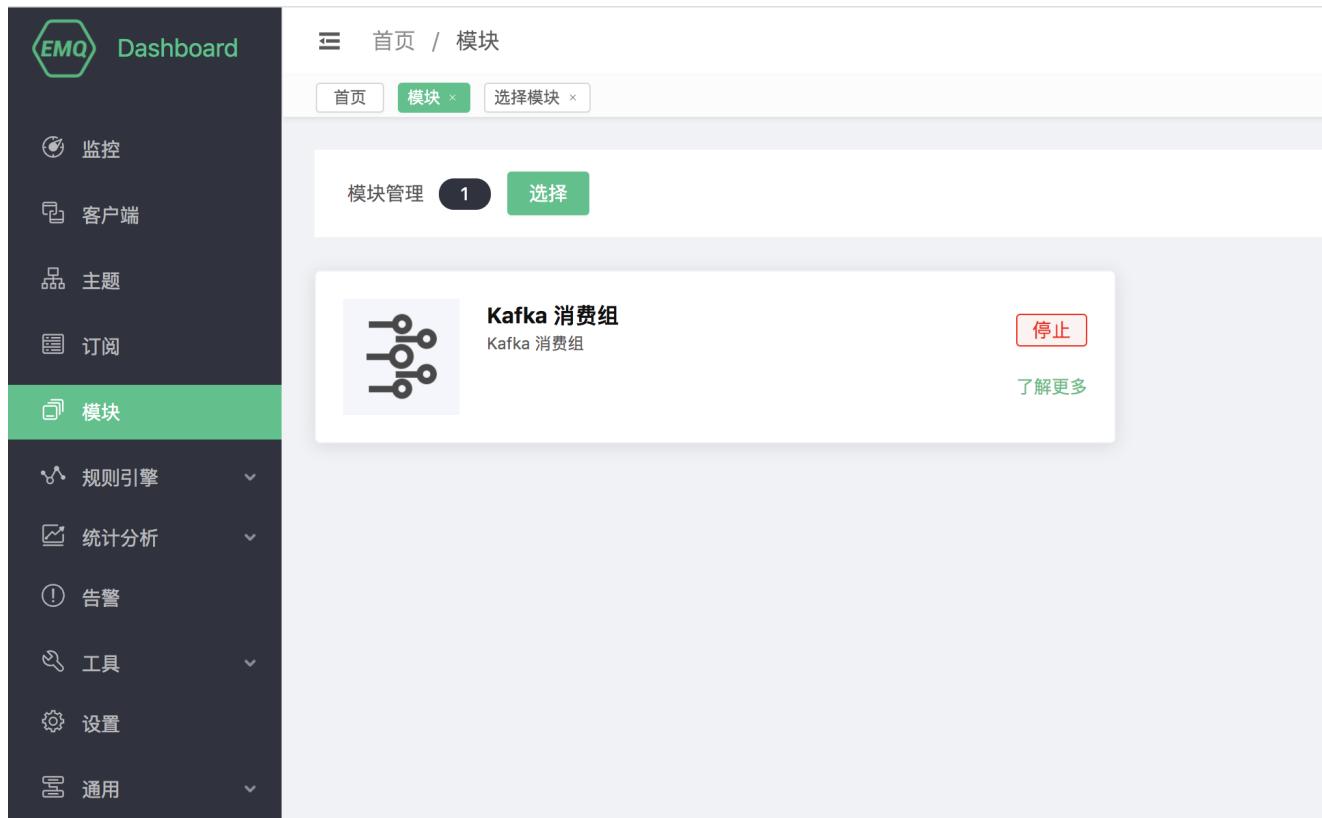
The screenshot shows the 'Parameter Settings' for the Kafka module. It includes fields for Kafka Server (127.0.0.1:9092), Pool Size (8), Kafka Username, Kafka Password, and a mapping table for Kafka topics to MQTT topics. The table has columns for Kafka Topic, MQTT Topic, MQTT QoS, MQTT Payload, and an 'Add' button. It also includes settings for binary key encoding, offset reset strategy, and SSL configuration.

Kafka Topic	MQTT Topic	MQTT QoS	MQTT Payload	Add
TestTopic	TestTopic	0	message.value	删除

- **Kafka 服务器地址**
- **Kafka consumer** 连接池大小
- **Kafka** 的订阅主题
- **MQTT** 的消息主题
- **MQTT** 的主题服务质量
- **MQTT PayLoad**, 可选使用 **Kafka message.value** 或者 **message** 全部信息
- 二进制 **key** 编码模式, **UTF-8** 或 **base64**, 消息中 **key** 的编码方式, 如果 **key** 值为非字符串或可能产生字符集编码异常的值, 推荐使用 **base64** 模式

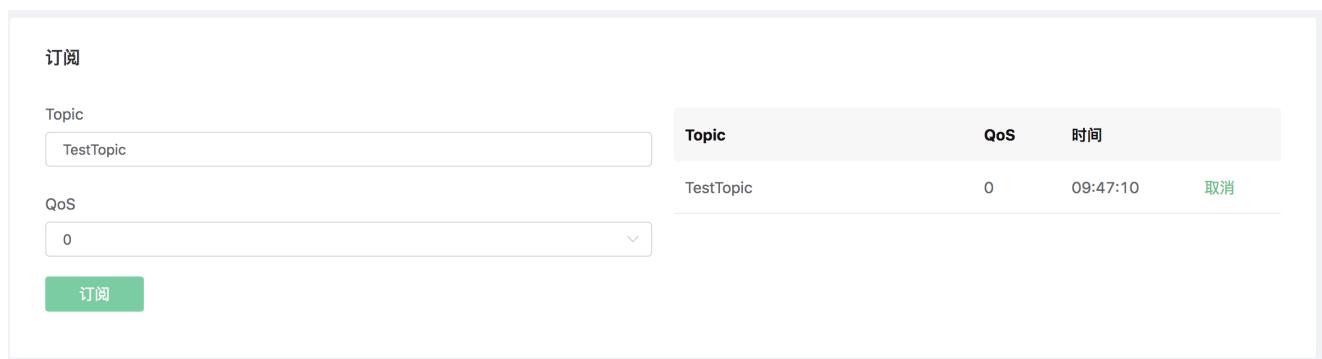
- 二进制 **value** 编码模式, **UTF-8** 或 **base64**, 消息中 **value** 的编码方式, 如果 **value** 值为非字符串或可能产生字符集编码异常的值, 推荐使用 **base64** 模式
- **Kafka Max Bytes** (每次从 Kafka 里消费消息的最大字节数)
- **Kafka Offset Reset Policy** (重置Offset策略,**reset_to_latest | reset_by_subdcriber**)
- **Kafka consumer** 是否重连
- **SSL** 连接参数

点击添加后, 模块添加完成:



The screenshot shows the EMQX Dashboard interface. On the left is a dark sidebar with a green header bar containing the word "模块". Below it are several menu items: 监控 (Monitoring), 客户端 (Client), 主题 (Topic), 订阅 (Subscription), 规则引擎 (Rule Engine), 统计分析 (Statistics), 告警 (Alert), 工具 (Tools), 设置 (Settings), and 通用 (General). The "模块" item is highlighted with a green background. To the right of the sidebar is the main content area. At the top of the content area is a navigation bar with tabs: 首页 (Home), 模块 (Module), and 选择模块 (Select Module). Below the navigation bar is a sub-navigation bar with tabs: 模块管理 (Module Management) and 选择 (Select). A button labeled "1" is positioned between the two tabs. In the main content area, there is a card titled "Kafka 消费组" (Kafka Consumer Group) with a subtitle "Kafka 消费组". To the left of the title is a icon depicting three interconnected circles. To the right is a red "停止" (Stop) button and a green "了解更多" (Learn More) button.

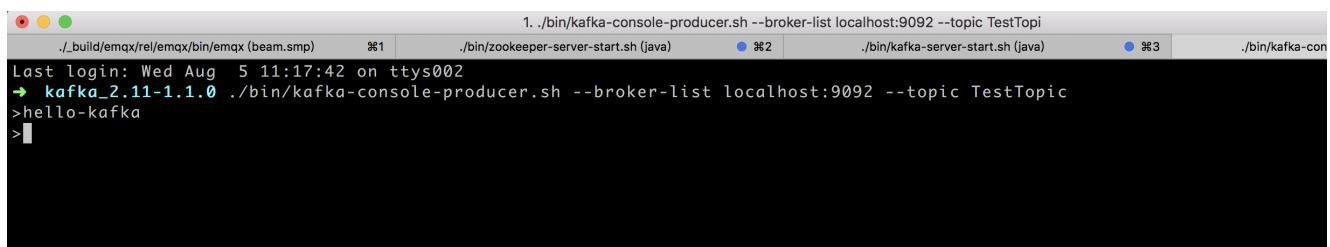
资源已经创建完成, 现在用**Dashboard**的**websocket**工具订阅**MQTT**的主题 "**TestTopic**":



The screenshot shows the "Subscription" section of the EMQX Dashboard. It has a header "订阅" (Subscription). Below the header are two input fields: "Topic" containing "TestTopic" and "QoS" containing "0". To the right of these fields is a table with three columns: Topic, QoS, and Time. The table contains one row with the values "TestTopic", "0", and "09:47:10". At the bottom of the section is a green "订阅" (Subscribe) button.

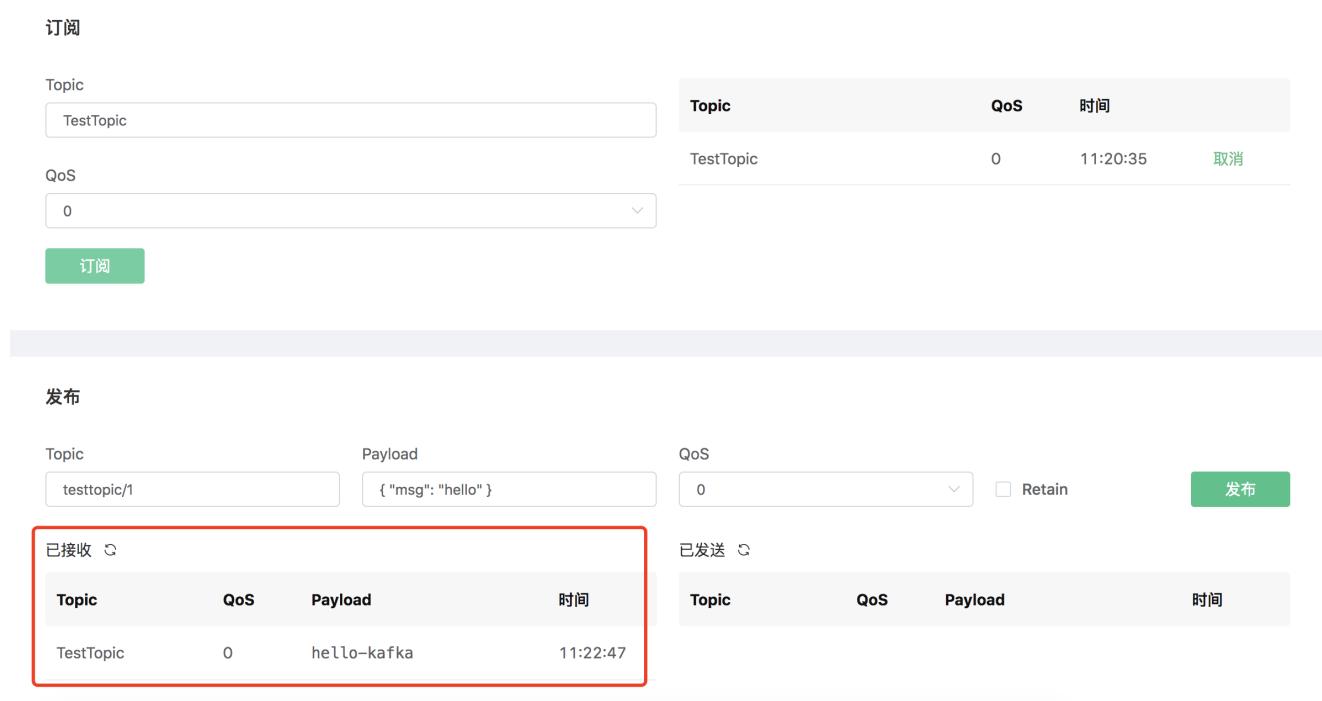
使用**kafka** 命令行 生产一条消息:

```
1 ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic TestTopic      sh
```



```
1. ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic TestTopic
Last login: Wed Aug  5 11:17:42 on ttys002
→ kafka_2.11-1.1.0 ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic TestTopic
>hello-kafka
>
```

Dashboard的websocket工具接收到了Kafka 生产的消息"hello-kafka":



订阅

Topic: TestTopic

QoS: 0

订阅

Topic	QoS	时间
TestTopic	0	11:20:35

发布

Topic: testtopic/1

Payload: { "msg": "hello" }

QoS: 0

发布

已接收

Topic	QoS	Payload	时间
TestTopic	0	hello-kafka	11:22:47

已发送

Topic	QoS	Payload	时间

Pulsar 消费组

Pulsar 消费组使用外部 Pulsar 作为消息队列，可以从 Pulsar 中消费消息转换成为 MQTT 消息发布在 emqx 中。

搭建 Pulsar 环境，以 MacOS X 为例：

```

1 $ wget https://archive.apache.org/dist/pulsar/pulsar-2.3.2/apache-pulsar-2.3.2-bin.tar.gz
2
3 $ tar xvfz apache-pulsar-2.3.2-bin.tar.gz
4
5 $ cd apache-pulsar-2.3.2
6
7 # 启动 Pulsar
8 $ ./bin/pulsar standalone

```

创建 Pulsar 的主题：

```

1 $ ./bin/pulsar-admin topics create-partitioned-topic -p 5 testTopic

```

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there is a sidebar with various navigation options: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 规则引擎 (Rules Engine), 统计分析 (Statistics), 模块 (Modules) [highlighted in green], 插件 (Plugins), 告警 (Alarms), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area is titled '模块管理' (Module Management) and shows four modules listed: Recon (Recon), 上下线通知 (Online/Offline Notifications), 内置访问控制文件 (Built-in Access Control File) [highlighted in blue], and MQTT 保留消息 (MQTT Retained Message). Each module has a status indicator (red circle with a white dot for Recon, green circle for others), a '了解更多' (More Information) link, and a power button icon.

选择 Pulsar 消费组模块：

← → ⌂ 127.0.0.1:18083/#/modules

Dashboard

首页 / 模块

模块管理 0 选择

暂无数据

- 监控
- 客户端
- 主题
- 订阅
- 模块**
- 规则引擎
- 统计分析
- 警告
- 工具
- 设置
- 通用

填写相关参数：

Dashboard

首页 / 模块 / 详情

Pulsar 消费组

Pulsar 消费组

配置信息

* Pulsar 服务器	Consumer Num
127.0.0.1:6650	1
流控阈值 ?	重置流控阈值的百分比 ?
1000	80%

* Pulsar 主题转 MQTT 主题映射关系

* Pulsar 主题	* MQTT 主题	* MQTT 主题服务质量	添加
TestTopic	TestTopic	0	删除

取消 添加

1). Pulsar 服务器地址

2). Pulsar consumer 进程数量

3). Pulsar 的订阅主题

4). MQTT 的消息主题

5). MQTT 的主题服务质量

6). Pulsar 流控阈值 (Pulsar 流控阈值, 配置 Pulsar 向消费者发送多少条消息后阻塞 Pulsar Consumer)

7). EMQX 重置流控阈值百分比 (Pulsar 流控阈值重置百分比。此配置让消费者处理完成一定数量的消息之后, 提前重置 Pulsar 流控阈值。比如, Pulsar 流控阈值 为 1000, 阈值重置百分比 为 80%, 则重置)

点击添加后, 模块添加完成:

The screenshot shows the EMQX Dashboard interface. On the left, there's a sidebar with various modules: 监控 (Monitoring), 客户端 (Client), 主题 (Topic), 订阅 (Subscription), and 模块 (Module). The '模块' (Module) item is highlighted with a green background. The main content area has a title '首页 / 模块'. Below it, there's a '模块管理' (Module Management) section with a counter '1' and a green '选择' (Select) button. A card for 'Pulsar 消费组' (Pulsar Consumer Group) is shown, featuring the Pulsar logo, the text 'Pulsar 消费组', and a green '启动' (Start) button.

资源已经创建完成, 现在用Dashboard的websocket工具订阅MQTT的主题 "TestTopic":

Topic	QoS	时间
TestTopic	0	09:47:10

使用pulsar-cli 生产一条消息:

```
1 ./bin/pulsar-client produce TestTopic --messages "hello-pulsar" sh
```

```

}
09:47:29.924 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ProducerStatsRecorderImpl - Pulsar client config: {
    "serviceUrl" : "pulsar://localhost:6650/",
    "operationTimeoutMs" : 30000,
    "statsIntervalSeconds" : 60,
    "numIoThreads" : 1,
    "numListenerThreads" : 1,
    "connectionsPerBroker" : 1,
    "useTcpNoDelay" : true,
    "useTls" : false,
    "tlsTrustCertsFilePath" : "",
    "tlsAllowInsecureConnection" : false,
    "tlsHostnameVerificationEnable" : false,
    "concurrentLookupRequest" : 5000,
    "maxLookupRequest" : 50000,
    "maxNumberOfRejectedRequestPerConnection" : 50,
    "keepAliveIntervalSeconds" : 30,
    "connectionTimeoutMs" : 10000
}
09:47:29.945 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ConnectionPool - [[id: 0xd5dc2440, L:/127.0.0.1:53355 - R:localhost/127.0.0.1:6650]] Connected to server
09:47:29.946 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ClientCnx - [id: 0xd5dc2440, L:/127.0.0.1:53355 - R:localhost/127.0.0.1:6650] Connected through proxy to target broker at 127.0.0.1:6650
09:47:29.948 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ProducerImpl - [TestTopic] [null] Creating producer on cnx [id: 0xd5dc2440, L:/127.0.0.1:53355 - R:localhost/127.0.0.1:6650]
09:47:29.958 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ProducerImpl - [TestTopic] [standalone-2-3] Created producer on cnx [id: 0xd5dc2440, L:/127.0.0.1:53355 - R:localhost/127.0.0.1:6650]
09:47:29.990 [pulsar-timer-4-1] WARN com.scurrilous.circe.checksum.Crc32cIntChecksum - Failed to load Circe JNI library. Falling back to Java based CRC32c provider
09:47:30.008 [main] INFO org.apache.pulsar.client.impl.PulsarClientImpl - Client closing. URL: pulsar://localhost:6650/
09:47:30.019 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ProducerImpl - [TestTopic] [standalone-2-3] Closed Producer
09:47:30.033 [main] INFO org.apache.pulsar.client.cli.PulsarClientTool - 1 messages successfully produced
→ apache-pulsar-2.3.2
→ apache-pulsar-2.3.2

```

Dashboard的websocket工具接收到了pulsar 生产的消息"hello-pulsar":

The screenshot shows the EMQX Dashboard interface with two main sections: '订阅' (Subscription) and '发布' (Publication).

订阅 (Subscription):

- Topic: TestTopic
- QoS: 0
- 订阅 (Subscribe) button

发布 (Publication):

- Topic: testtopic/1
- Payload: {"msg": "hello"}
- QoS: 0
- Retain: unchecked
- 发布 (Publish) button

已接收 (Received): A red box highlights this section, which contains a table showing a received message.

Topic	QoS	Payload	时间
TestTopic	0	hello-pulsar	09:47:30

已发送 (Published): This section is currently empty.

MQTT 订阅者

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left sidebar, the 'Modules' option is selected. In the main content area, there is a 'Module Management' section with four modules listed:

- Recon**: Recon module, status off, '了解更多' button.
- 上下线通知**: Client online/offline notification module, status off, '了解更多' button.
- 内置访问控制文件**: Internal access control file module, status off, '了解更多' button.
- MQTT 保留消息**: MQTT retained message module, status off, '了解更多' button.

A red box highlights the 'MQTT 保留消息' (MQTT Retained Message) module.

选择 **MQTT 订阅者** 模块：

The screenshot shows the 'Select Module' page in the EMQX Dashboard. The left sidebar shows the 'Modules' section is selected. The main area has tabs for '选择模块' (27), '认证鉴权', '协议接入', '消息下发', '多语言扩展', '运维监控', and '内部模块'. The '消息下发' tab is active. It lists several modules:

- Kafka 消费组**: Kafka Consumer group, '选择' and '了解更多' buttons.
- MQTT Subscriber**: MQTT Subscriber module, highlighted with a red box, '选择' and '了解更多' buttons.
- Pulsar 消费组**: Pulsar Consumer group, '选择' and '了解更多' buttons.

Below this, there are sections for '多语言扩展' (Multi-language Extension) with two items: '多语言扩展协议接入' and '多语言扩展钩子'.

填写相关参数：

点击添加后，模块添加完成：

多语言扩展 - 协议接入

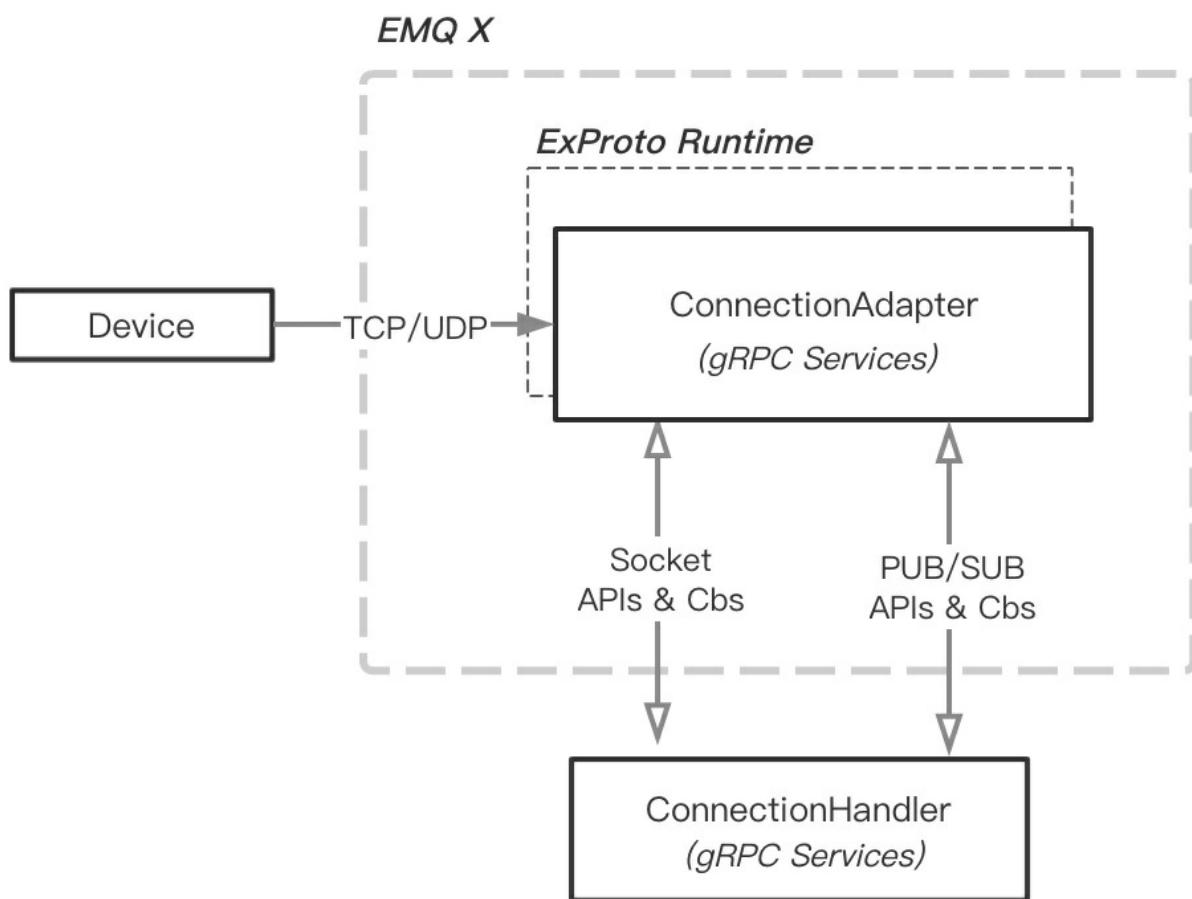
在 **EMQX e4.2.0** 中，我们提供了多语言扩展的支持。其中 多语言扩展协议接入 模块允许其他编程语言（例如：**Python, Java** 等）直接处理字节数据报文实现自定义协议的解析，并提供 **Pub/Sub** 接口实现与系统的消息交换。

该功能给 **EMQX** 带来的扩展性十分的强大，它能以用户熟悉的编程语言处理任何的私有协议，并享受由 **EMQX** 系统带来的极高并发连接的优点。

特性

- 极强的扩展能力。使用 **gRPC** 作为 **RPC** 通信框架，支持各个主流编程语言
- 完全的异步 **I/O**。连接层以完全的异步非阻塞式 **I/O** 的方式实现
- 连接层透明。完全的支持 **TCP\TLS UDP\DTLS** 类型的连接管理，并对上层提供统一一个 **API**
- 连接管理能力。例如，最大连接数，连接和吞吐的速率限制，**IP** 黑名单等

架构



该模块主要需要处理的内容包括：

1. 连接层： 该部分主要维持 **Socket** 的生命周期，和数据的收发。它的功能要求包括：

- 监听某个端口。当有新的 **TCP/UDP** 连接到达后，启动一个连接进程，来维持连接的状态。
- 调用 `OnSocketCreated` 回调。用于通知外部模块已新建立了一个连接。
- 调用 `OnScoektClosed` 回调。用于通知外部模块连接已关闭。

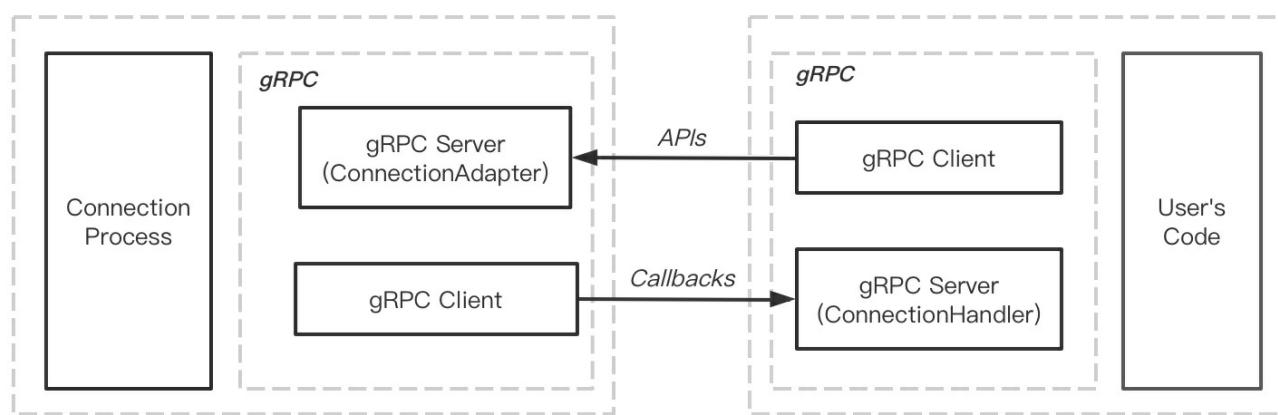
- 调用 `OnReceivedBytes` 回调。用于通知外部模块该连接新收到的数据包。
- 提供 `Send` 接口。供外部模块调用，用于发送数据包。
- 提供 `Close` 接口。供外部模块调用，用于主动关闭连接。

2. 协议/会话层： 该部分主要提供 **PUB/SUB** 接口，以实现与 **EMQX Broker** 系统的消息互通。包括：

- 提供 `Authenticate` 接口。供外部模块调用，用于向集群注册客户端。
- 提供 `StartTimer` 接口。供外部模块调用，用于为该连接进程启动心跳等定时器。
- 提供 `Publish` 接口。供外部模块调用，用于发布消息 **EMQX Broker** 中。
- 提供 `Subscribe` 接口。供外部模块调用，用于订阅某主题，以实现从 **EMQX Broker** 中接收某些下行消息。
- 提供 `Unsubscribe` 接口。供外部模块调用，用于取消订阅某主题。
- 调用 `OnTimerTimeout` 回调。用于处理定时器超时的事件。
- 调用 `OnReceivedMessages` 回调。用于接收下行消息（在订阅主题成功后，如果主题上有消息，便会回调该方法）

接口设计

从 **gRPC** 的角度上看，**ExProto** 会作为客户端向 `ConnectionHandler` 服务发送回调请求。同时，它也会作为服务端向外部模块提供 `ConnectionAdapter` 服务，以提供各类接口的调用。如图：



详情参见：[exproto.proto](#)，例如接口的定义有：

```

1  syntax = "proto3";
2
3  package emqx.exproto.v1;
4
5  // The Broker side service. It provides a set of APIs to
6  // handle a protocol access
7  service ConnectionAdapter {
8
9    // -- socket layer
10
11   rpc Send(SendBytesRequest) returns (CodeResponse) {};
12
13   rpc Close(CloseSocketRequest) returns (CodeResponse) {};
14
15   // -- protocol layer
16
17   rpc Authenticate(AuthenticateRequest) returns (CodeResponse) {};
18
19   rpc StartTimer(TimerRequest) returns (CodeResponse) {};
20
21   // -- pub/sub layer
22
23   rpc Publish(PublishRequest) returns (CodeResponse) {};
24
25   rpc Subscribe(SubscribeRequest) returns (CodeResponse) {};
26
27   rpc Unsubscribe(UnsubscribeRequest) returns (CodeResponse) {};
28 }
29
30 service ConnectionHandler {
31
32   // -- socket layer
33
34   rpc OnSocketCreated(SocketCreatedRequest) returns (EmptySuccess) {};
35
36   rpc OnSocketClosed(SocketClosedRequest) returns (EmptySuccess) {};
37
38   rpc OnReceivedBytes(ReceivedBytesRequest) returns (EmptySuccess) {};
39
40   // -- pub/sub layer
41
42   rpc OnTimerTimeout(TimerTimeoutRequest) returns (EmptySuccess) {};
43
44   rpc OnReceivedMessages(ReceivedMessagesRequest) returns (EmptySuccess) {};
45 }
```

开发指南

在使用该模块之前，用户需要开发和部署一个 **gRPC** 的服务，并实现 `exproto.proto` 定义的接口。

其步骤如下：

1. 拷贝出当前版本的 `lib/emqx_exproto-<x.y.z>/priv/protos/exproto.proto` 文件。
2. 使用对应编程语言的 **gRPC** 框架，生成 `exproto.proto` 的 **gRPC** 服务端的代码。
3. 实现 `exproto.proto` 当中 `ConnectionHandler` 服务的接口。

开发完成后，需将该服务部署到与 **EMQX** 能够通信的服务器上，并保证端口的开放。

其中各个语言的 **gRPC** 框架可参考：[grpc-ecosystem/awesome-grpc](#)

创建模块

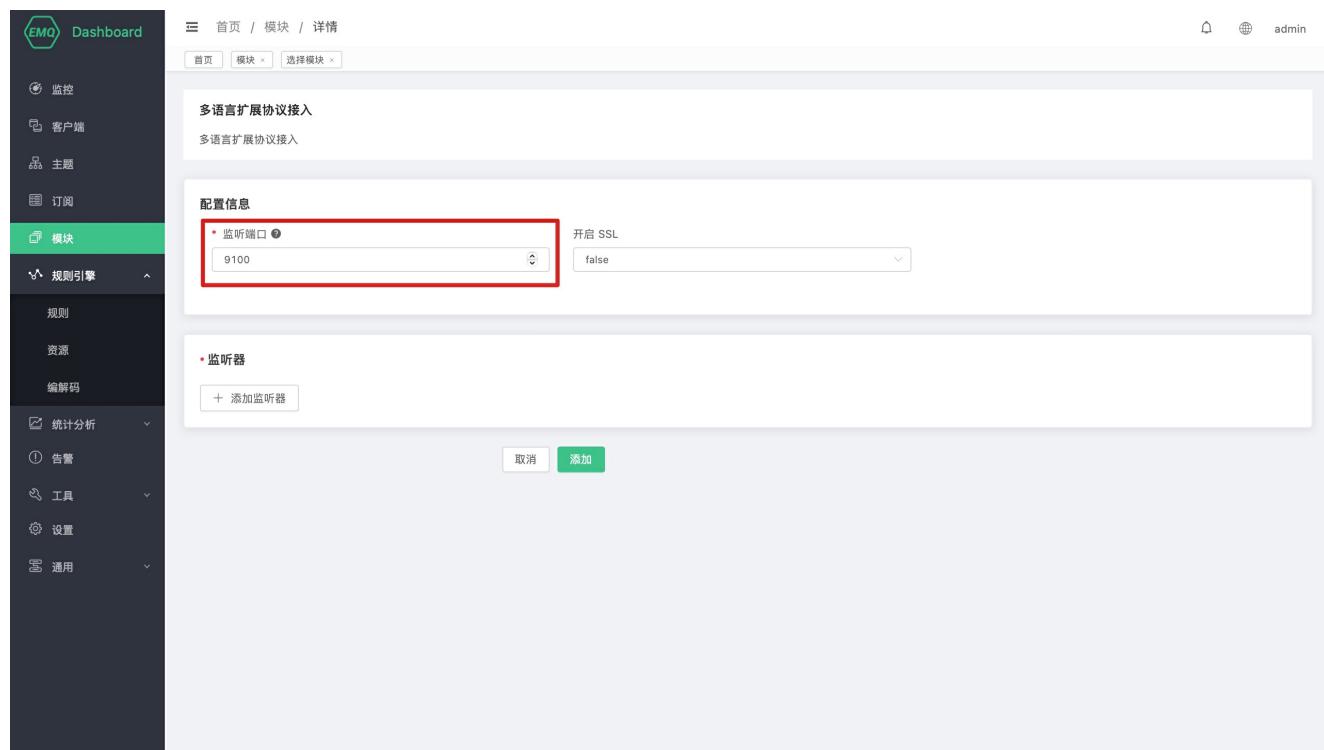
成功部署 **gRPC** 服务后，通过 **dashboard** 页面可以开启多语言扩展协议接入模块，并配置以下三部分的内容，即可成功使用：

- ExProto** 的 `ConnectionAdapter` 服务的监听地址。用于接收 **gRPC** 请求。
- 配置 **监听器(Listener)**，提供 **TCP/UDP/SSL/DTLS** 的地址监听。用于监听、接收设备的连接。
- 为每个监听器指定一个 `ConnectionHandler` 的服务地址。用于发送各种事件回调到用户的服务。

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

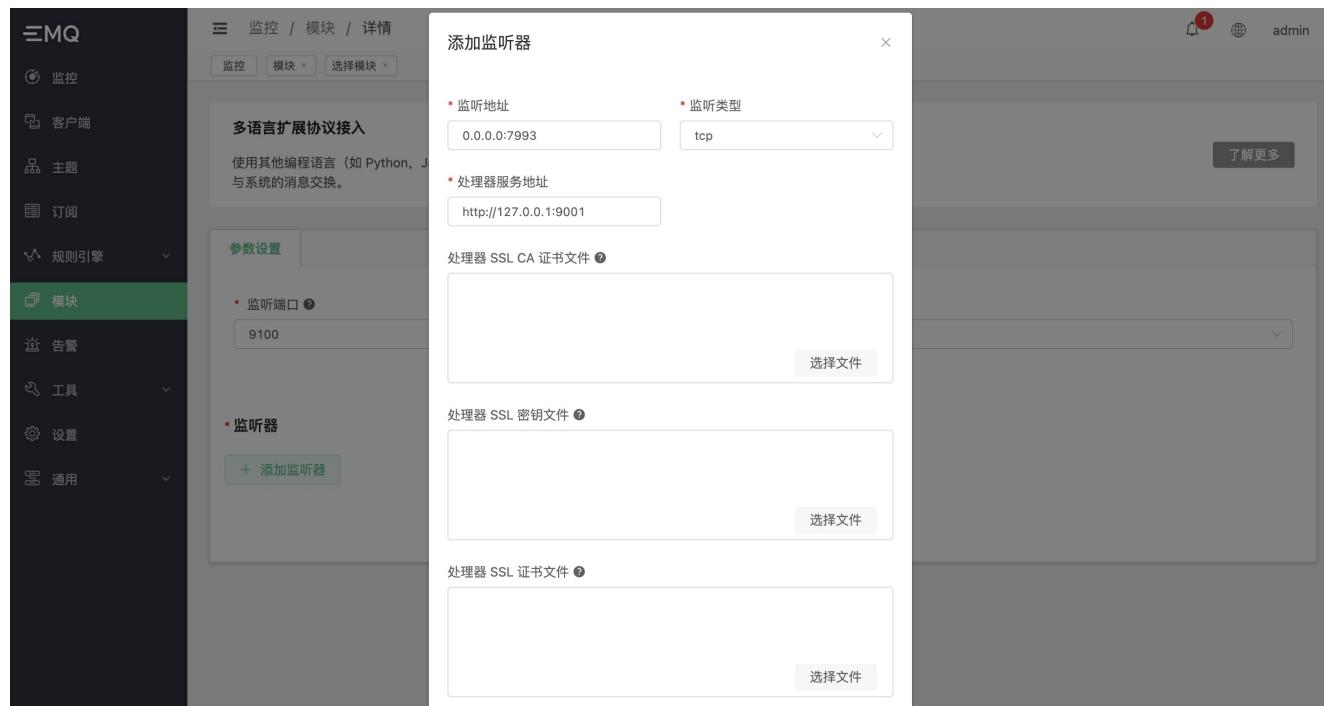
选择“多语言扩展协议接入”：

配置 ConnectionAdapter 服务的监听地址，和是否为其开启 SSL 监听：



点击“添加监听器”为 ExProto 模块配置监听器，其中包括：

1. 监听器的 监听地址 和 监听类型，它表明以何种方式接收自定义协议的 Socket 连接。
2. 连接处理服务(ConnectionHandler) 的 处理器服务地址 和可能会有的 SSL 证书配置，它表明 ExProto 如何访问 ConnectionHandler 服务。



点击确定，完成监听器添加；在点击添加完成模块的创建：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left, a dark sidebar menu includes 'Dashboard', '监控' (Monitoring), '客户端' (Client), '主题' (Topic), '订阅' (Subscription), '模块' (Module) which is highlighted in green, '规则引擎' (Rule Engine), '统计分析' (Statistics Analysis), '告警' (Alerts), '工具' (Tools), '设置' (Settings), and '通用' (General). The main content area has a header '首页 / 模块' and a search bar '搜索模块...'. A success message '模块添加成功!' is shown in a box. Below it, 'Module Management' shows one module named '多语言扩展协议接入' with a status icon and a '了解更多' link. A search bar '搜索模块...' is also present.

至此，多语言扩展协议接入的配置已经完成。

多语言扩展 - 钩子

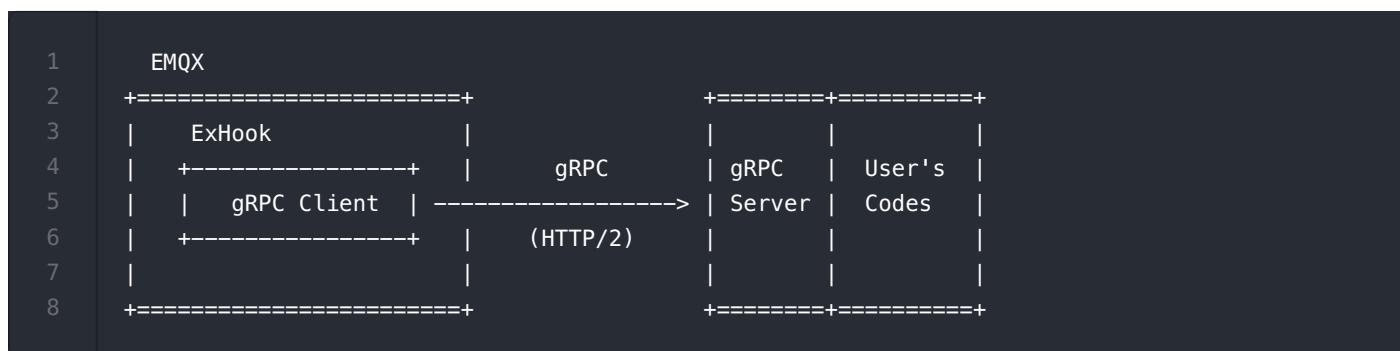
在 **EMQX e4.2.0** 中，我们提供了多语言扩展的支持。其中 多语言扩展钩子 模块允许使用其他编程语言（例如：**Python, Java** 等）直接向 **EMQX** 系统挂载钩子，以接收并处理 **EMQX** 系统的钩子事件，达到扩展和定制 **EMQX** 的目的。例如，用户可以使用其他编程语言来自定义：

- 客户端接入的认证授权
- 发布/订阅的 **ACL** 鉴权
- 消息的持久化，桥接
- 发布/订阅，或者客户端上下线事件的通知处理

设计

多语言扩展钩子 功能由 `emqx-exhook` 插件提供。它使用 **gRPC** 作为 **RPC** 的通信框架。

架构如下图：



它表明：**EMQX** 中的 **ExHook** 模块作为一个 **gRPC** 客户端，将系统中所有的钩子事件发送到用户的 **gRPC** 服务端。

和 **EMQX** 原生的钩子一致，它也支持链式的方式计算和返回：



接口设计

作为事件的处理端，即 **gRPC** 的服务端。它需要用户自定义实现需要挂载的钩子列表，和每个钩子事件到达后如何去处理的回调函数。这些接口在 多语言扩展钩子 中被定义为一个名为 `HookProvider` 的 **gRPC** 服务，其需要实现的接口的列表包含：

```

1  syntax = "proto3";
2
3  package emqx.exhook.v1;
4
5  service HookProvider {
6
7      rpc OnProviderLoaded(ProviderLoadedRequest) returns (LoadedResponse) {};
8
9      rpc OnProviderUnloaded(ProviderUnloadedRequest) returns (EmptySuccess) {};
10
11     rpc OnClientConnect(ClientConnectRequest) returns (EmptySuccess) {};
12
13     rpc OnClientConnack(ClientConnackRequest) returns (EmptySuccess) {};
14
15     rpc OnClientConnected(ClientConnectedRequest) returns (EmptySuccess) {};
16
17     rpc OnClientDisconnected(ClientDisconnectedRequest) returns (EmptySuccess) {};
18
19     rpc OnClientAuthenticate(ClientAuthenticateRequest) returns (ValuedResponse) {};
20
21     rpc OnClientCheckAcl(ClientCheckAclRequest) returns (ValuedResponse) {};
22
23     rpc OnClientSubscribe(ClientSubscribeRequest) returns (EmptySuccess) {};
24
25     rpc OnClientUnsubscribe(ClientUnsubscribeRequest) returns (EmptySuccess) {};
26
27     rpc OnSessionCreated(SessionCreatedRequest) returns (EmptySuccess) {};
28
29     rpc OnSessionSubscribed(SessionSubscribedRequest) returns (EmptySuccess) {};
30
31     rpc OnSessionUnsubscribed(SessionUnsubscribedRequest) returns (EmptySuccess) {};
32
33     rpc OnSessionResumed(SessionResumedRequest) returns (EmptySuccess) {};
34
35     rpc OnSessionDiscarded(SessionDiscardedRequest) returns (EmptySuccess) {};
36
37     rpc OnSessionTakeovered(SessionTakeoveredRequest) returns (EmptySuccess) {};
38
39     rpc OnSessionTerminated(SessionTerminatedRequest) returns (EmptySuccess) {};
40
41     rpc OnMessagePublish(MessagePublishRequest) returns (ValuedResponse) {};
42
43     rpc OnMessageDelivered(MessageDeliveredRequest) returns (EmptySuccess) {};
44
45     rpc OnMessageDropped(MessageDroppedRequest) returns (EmptySuccess) {};
46
47     rpc OnMessageAcked(MessageAckedRequest) returns (EmptySuccess) {};
48 }

```

其中 **HookProvider** 部分：

- `OnProviderLoaded`：定义 **HookProvider** 如何被加载，返回需要挂载的钩子列表。仅在该列表中的钩子会被回调到 **HookProvider** 服务。
- `OnProviderUnloaded`：定义 **HookProvider** 如何被卸载，仅用作通知。

钩子事件部分：

- `OnClient*`, `OnSession*`, `OnMessage*` 为前缀的方法与 [钩子](#) 的当中的方法一一对应。它们有着相同的调用时机和相似的参数列表。
- 仅 `OnClientAuthenticate`, `OnClientCheckAcl`, `OnMessagePublish` 允许携带返回值到 **EMQX** 系统, 其它回调则不支持。

其中接口和参数数据结构的详情参考: [exhook.proto](#)

开发指南

用户在使用多语言扩展钩子的功能时, 需要先实现 `HookProvider` 的 **gRPC** 服务来接收 **ExHook** 发送出来的回调事件。

其步骤如下:

1. 拷贝出当前版本的 `lib/emqx_exhook-<x.y.z>/priv/protos/exhook.proto` 文件。
2. 使用对应编程语言的 **gRPC** 框架, 生成 `exhook.proto` 的 **gRPC** 服务端的代码。
3. 按需实现 `exhook.proto` 当中定义的接口。

开发完成后, 需将该服务部署到与 **EMQX** 能够通信的服务器上, 并保证端口的开放。

其中各个语言的 **gRPC** 框架可参考: [grpc-ecosystem/awesome-grpc](#)

创建模块

在成功部署 `HookProvider` 服务后, 通过 **dashboard** 页面可以开启多语言钩子扩展模块, 并配置其服务地址即可正常使用。

打开 [EMQX Dashboard](#), 点击左侧的“模块”选项卡, 选择添加:

选择“多语言扩展钩子”:

The screenshot shows the '选择模块' (Select Module) page under the '多语言扩展' (Multi-language Extension) category. The '多语言扩展钩子' (Multi-language Extension Hook) module is highlighted with a red box. Other visible modules include '多语言扩展协议接入' (Multi-language Extension Protocol Access), 'Recon', '热配置' (Hot Configuration), '主题监控' (Topic Monitoring), 'MQTT 增强认证' (MQTT Enhanced Authentication), and '上下线通知' (Online/Offline Notification).

配置 `HookProvider` 服务相关参数:

The screenshot shows the '参数设置' (Parameter Settings) page for the '多语言扩展钩子' (Multi-language Extension Hook) module. It includes fields for '服务器 URL' (Server URL) set to 'http://127.0.0.1:8991', '自动重连间隔' (Automatic Reconnection Interval) set to '60s', '备选动作' (Fallback Action) set to 'deny', '请求超时时间' (Request Timeout) set to '5s', and an '开启 HTTPS' (Enable HTTPS) dropdown set to 'false'. Buttons for '取消' (Cancel) and '添加' (Add) are at the bottom.

点击添加后，模块添加完成

模块添加成功!

模块管理 1 选择

搜索模块...

多语言扩展钩子
多语言扩展钩子

了解更多

至此，多语言扩展钩子的配置已经完成。

recon

EMQX Recon 调试/调优

创建模块

recon模块默认启动，可以通过**dashboard**页面进行启动和停止。

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡：

The screenshot shows the EMQX Dashboard interface. On the left, a dark sidebar menu is open, with the 'Modules' option highlighted in green. The main content area is titled '模块管理' (Module Management) and shows four modules listed:

- Recon**: Status: On, Description: Recon, More info link.
- 上下线通知**: Status: Off, Description: 客户端上下线通知, More info link.
- 内置访问控制文件**: Status: Off, Description: 内置访问控制文件, More info link.
- MQTT 保留消息**: Status: Off, Description: MQTT 保留消息, More info link.

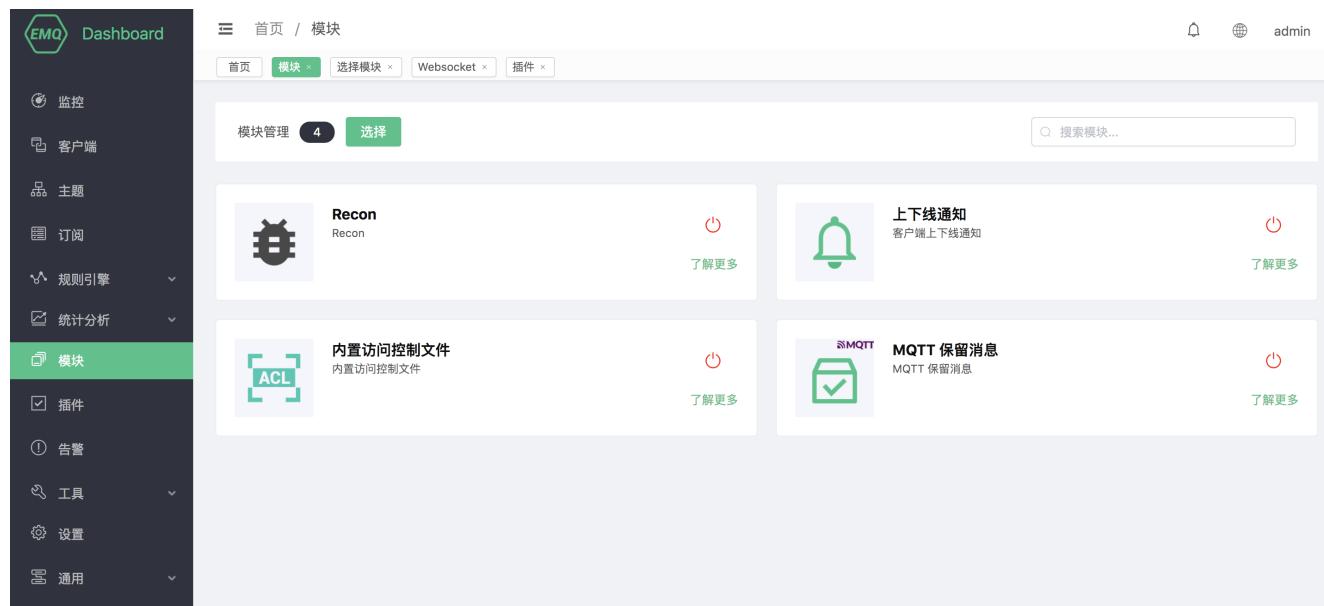
At the top of the main content area, there are navigation tabs: 首页 / 模块 / 选择模块 / Websocket / 插件. On the right side of the header, there are user account and system settings icons.

EMQX Prometheus Agent

EMQX Prometheus Agent 支持将数据推送至 **Pushgateway** 中，然后再由 **Prometheus Server** 拉取进行存储。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



Dashboard 首页 / 模块

模块管理 4 选择

搜索模块...

Recon Recon

了解更多

上下线通知 客户端上下线通知

了解更多

内置访问控制文件 内置访问控制文件

了解更多

MQTT 保留消息 MQTT 保留消息

了解更多

选择 **EMQX Prometheus Agent**

The screenshot shows the EMQX Enterprise V4.4 Dashboard. In the top navigation bar, there are links for '首页' (Home), '规则引擎' (Rule Engine), '资源' (Resources), '创建规则' (Create Rule), '模块' (Module), 'components.addModule' (Add Module), '选择模块' (Select Module), '订阅' (Subscription), and '主题监控' (Topic Monitoring). The '模块' (Module) link is highlighted.

In the main content area, under the '多语言扩展' (Multi-language Extension) heading, the 'EMQ X Prometheus Agent' module is selected and highlighted with a red border. Other modules shown include 'exhook' and 'Recon'. Below this, under the '运维监控' (Operations Monitoring) heading, the 'EMQ X Reloader' and 'Recon' modules are listed. The '内部模块' (Internal Modules) section contains several MQTT-related modules: '主题监控' (Topic Monitoring), 'MQTT 增强认证' (MQTT Enhanced Authentication), '上下线通知' (Online/Offline Notifications), 'MQTT 代理订阅' (MQTT Broker Subscription), '主题重写' (Topic Rewrite), 'MQTT 保留消息' (MQTT Retained Message), and '延迟发布' (Delayed Publish).

配置相关参数

The screenshot shows the configuration page for the 'EMQ X Prometheus Agent' module. The left sidebar includes links for '监控' (Monitoring), '客户端' (Client), '主题' (Topic), '订阅' (Subscription), '模块' (Module), '规则引擎' (Rule Engine), '统计分析' (Statistics Analysis), '主题监控' (Topic Monitoring), '告警' (Alerts), '工具' (Tools), '设置' (Settings), and '通用' (General). The '模块' (Module) link is highlighted.

The main content area displays the configuration information for the 'EMQ X Prometheus Agent'. It includes fields for 'PushGateway URL' (with value 'http://127.0.0.1:9091') and '推送间隔' (Push Interval) (with value '15s'). At the bottom of the configuration panel are '取消' (Cancel) and '添加' (Add) buttons.

点击添加后，模块添加完成

Grafana 数据模板

`emqx_prometheus` 插件提供了 **Grafana** 的 **Dashboard** 的模板文件。这些模板包含了所有 **EMQX** 监控数据的展示。用户可直接导入到 **Grafana** 中，进行显示 **EMQX** 的监控状态的图标。

模板文件位于：[emqx_prometheus/grafana_template](#)。

热配置

开启热配置后，**EMQX** 将从配置文件中拷贝一份配置副本，所有可在 **Dashboard** 修改的配置都会持久化到磁盘中。如果修改配置文件将会覆盖热配置，请谨慎使用。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡：

选择 热配置模块：

点击选择后，模块添加完成：

The screenshot shows the EMQX Dashboard interface. On the left is a dark sidebar with various navigation options like Monitoring, Clients, Themes, Subscriptions, Rule Engines, Statistics, Modules, Plugins, Alarms, Tools, and General settings. The 'Modules' option is currently selected and highlighted in green. The main content area is titled 'Module Management' and shows five modules: Recon, Internal Access Control File (ACL), MQTT Persistence, Online/Offline Notifications, and Hot Configuration. The 'Hot Configuration' module is highlighted with a red border.

emqx提供了较多的配置在dashboard修改，包括基础配置、zones、监听器、监控告警等

The screenshot shows the 'Settings' page within the 'Hot Configuration' module. The sidebar remains the same. The main area has tabs for 'Basic Settings', 'Zone', 'Listener', 'Monitoring Alarm', and 'Cluster Settings'. The 'Basic Settings' tab is active. It contains several configuration parameters with descriptions:

- acl_cache_max_size: 32 (can be cached maximum ACL number)
- acl_cache_ttl: 1m (ACL cache time to live)
- acl_deny_action: ignore (action when denied)
- acl_nomatch: allow (allow or deny when not matched)
- allow_anonymous: true (allow anonymous users, note: disabled by default)
- broker_session_locking_strategy: quorum (session locking strategy: all, leader, quorum, local)
- broker_shared_dispatch_ack_enabled: false (enable internal ACK for shared dispatch)
- broker_shared_subscription_strategy: random (shared subscription strategy)
- broker_sys_heartbeat: 30s (Broker health status publish interval)
- broker_sys_interval: 1m (Broker statistics publish interval)

主题监控

EMQX 提供了主题指标统计功能，可以统计指定主题下的消息收发数量、速率等指标。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left sidebar, the '模块' (Module) option is selected. In the main content area, there is a '模块管理' (Module Management) section with four modules listed: 'Recon' (status off), '内置访问控制文件' (status off), '上下线通知' (status off), and 'MQTT 保留消息' (status off). A green '选择' (Select) button is located at the top right of this section.

选择主题监控模块，无需配置参数，直接开启

The screenshot shows the 'Select Module' screen at the URL <http://127.0.0.1:18083/#/modules/select>. The '模块' (Module) option is selected in the sidebar. In the main content area, there are two tabs: '选择模块' (Select Module) and '已安装' (Installed). Under '选择模块', there are several modules listed: 'Recon' (status off), '热配置' (status off), 'EMQ X Prometheus Agent' (status off), and 'MQTT 保留消息' (status off). Under '已安装' (Installed), there are several modules listed: '主题监控' (Topic Monitoring) (status on, highlighted with a red border), 'MQTT 增强认证' (status off), 'MQTT 代理订阅' (status off), and '延迟发布' (status off).

如何使用主题监控

主题监控页面位于 **Dashboard** 的统计分析标签下，主题指标统计功能开启后，你可以点击页面右上角的 **Create** 按钮来创建新的主题指标统计。以下是在已经创建了 `a/c` 与 `a/b` 主题指标统计之后的页面，你将可以看到这两个主题下消息流入流出、丢弃的总数和当前速率。

主题	消息流入	消息流出	消息丢弃
<code>a/c</code>	0	0	0
<code>a/b</code>	0	0	0

出于整体性能考虑，目前主题指标统计功能仅支持主题名，即不支持带有 `+` 或 `#` 通配符的主题过滤器，例如 `a/+` 等。也许将来有一天我们会实现它，如果我们解决了性能问题。

HTTP API

我们为您提供了与 **Dashboard** 操作一致的 **HTTP API**，以便您与自己的应用进行集成。相关 **HTTP API** 的具体使用方法，请参见 [HTTP API - 主题指标统计](#)。

MQTT 增强认证

上下线通知

EMQX 的上下线系统消息通知功能在客户端连接成功或者客户端断开连接，自动发送一条系统主题的消息，**EMQX** 默认开启上下线通知模块。

创建模块

上下线通知模块默认启动，可以通过**dashboard**页面进行启动和停止。

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡：

The screenshot shows the EMQX Dashboard interface. On the left, there's a sidebar with various navigation options: Monitoring, Client, Topic, Subscription, Rule Engine, Statistics, Modules (which is selected and highlighted in green), Plugins, Alarms, Tools, Settings, and General. The main content area is titled 'Module Management' and shows four modules: 'Recon' (status off), 'Online/Offline' (status off), 'ACL' (status off), and 'MQTT Persistence' (status off). There are also tabs for 'Home', 'Module Selection', 'WebSocket', and 'Plugin'.

上下线消息通知格式

\$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， EMQX 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， EMQX 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：

```
1  {
2      "username": "foo",
3      "ts": 1625572213873,
4      "sockport": 1883,
5      "proto_ver": 4,
6      "proto_name": "MQTT",
7      "keepalive": 60,
8      "ipaddress": "127.0.0.1",
9      "expiry_interval": 0,
10     "connected_at": 1625572213873,
11     "connack": 0,
12     "clientid": "emqtt-8348fe27a87976ad4db3",
13     "clean_start": true
14 }
```

sh

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

```
1  {
2      "username": "foo",
3      "ts": 1625572213873,
4      "sockport": 1883,
5      "reason": "tcp_closed",
6      "proto_ver": 4,
7      "proto_name": "MQTT",
8      "ipaddress": "127.0.0.1",
9      "disconnected_at": 1625572213873,
10     "clientid": "emqtt-8348fe27a87976ad4db3"
11 }
```

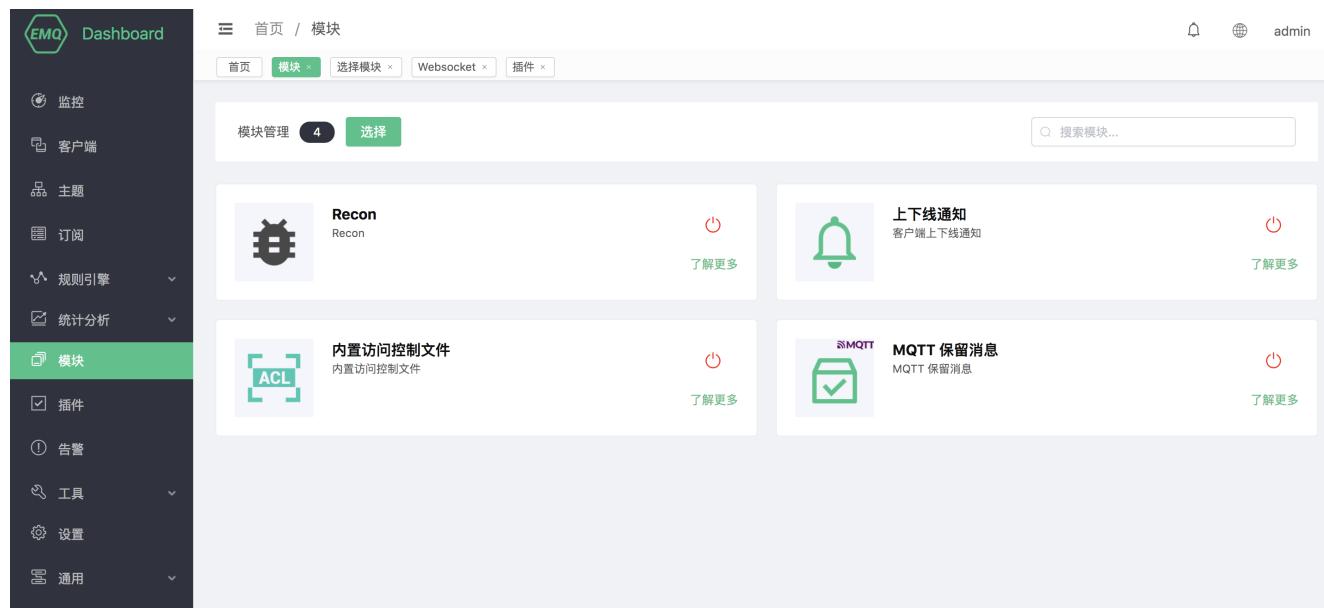
sh

MQTT 代理订阅

EMQX 的代理订阅功能使得客户端在连接建立时，不需要发送额外的 **SUBSCRIBE** 报文，便能自动建立用户预设的订阅关系。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

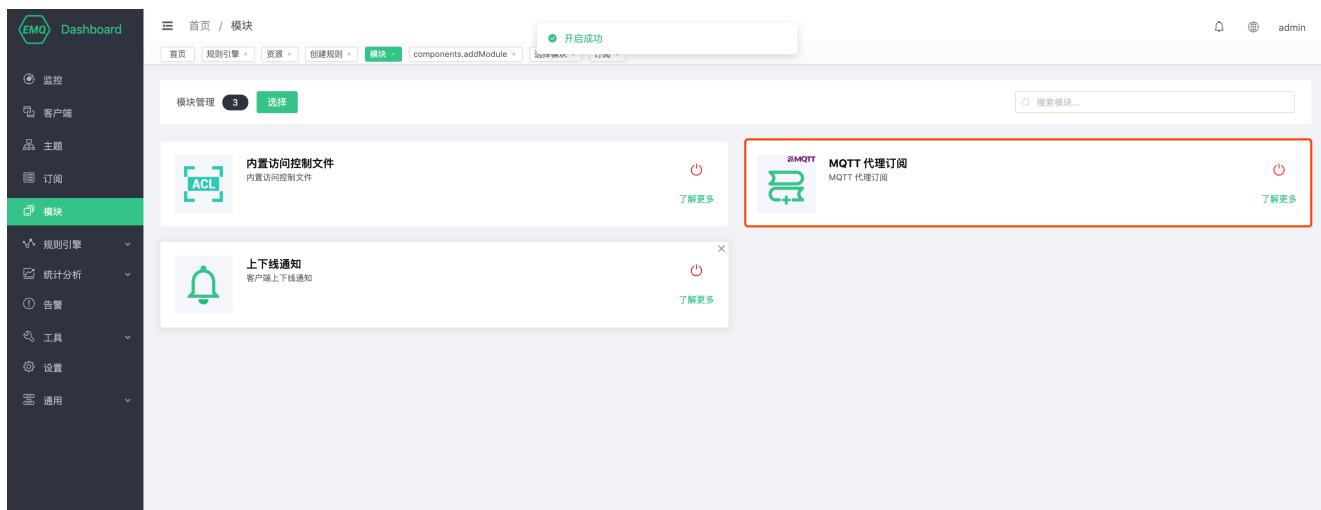


The screenshot shows the EMQX Dashboard interface. On the left, there's a dark sidebar with various navigation options like Monitoring, Clients, Themes, Subscriptions, Rules Engine, Statistics, Plugins, Alarms, Tools, Settings, and General. The 'Modules' option is highlighted with a green background. The main content area is titled 'Module Management' and shows four available modules: 'Recon' (Recon), '上下线通知' (Client Online/Offline Notification), '内置访问控制文件' (Built-in Access Control File), and 'MQTT 保留消息' (MQTT Retained Message). Each module card includes a status icon (red circle with a white dot), the module name, a brief description, and a 'More' link.

选择 **MQTT** 代理订阅模块

配置相关参数

点击添加后，模块添加完成



订阅选项

- 服务质量 (**QoS**)

服务端可以向客户端发送的应用消息的最大 **QoS** 等级。

- NL (No Local)**

应用消息是否能够被转发到发布此消息的客户端。

- **NL** 值为 **0** 时，表示应用消息可以被转发给发布此消息的客户端。

- **NL** 值为 **1** 时，表示应用消息不能被转发给发布此消息的客户端。

- RAP (Retain As Published)**

向此订阅转发应用消息时，是否保持消息被发布时设置的保留(**RETAIN**)标志。

- **RAP** 值为 **0** 时，表示向此订阅转发应用消息时把保留标志设置为 **0**。

- **RAP** 值为 **1** 时，表示向此订阅转发应用消息时保持消息被发布时设置的保留标志。

- RH (Retain Handling)**

当订阅建立时，是否发送保留消息

- **0**: 订阅建立时发送保留消息

- **1**: 订阅建立时，若该订阅当前不存在则发送保留消息

- **2**: 订阅建立时不要发送保留消息

代理订阅规则

在配置代理订阅的主题时，**EMQX** 提供了 `%c` 和 `%u` 两个占位符供用户使用，**EMQX** 会在执行代理订阅时将配置中的 `%c` 和 `%u` 分别替换为客户端的 `Client ID` 和 `Username`，需要注意的是，`%c` 和 `%u` 必须占用一整个主题层级。

例如，添加上文图中的规则后：配置 **A**、**B** 两个客户端，客户端 **A** 的 `Client ID` 为 `testclientA`，`Username` 为 `testerA`，客户端 **B** 的 `Client ID` 为 `testclientB`，`Username` 为 `testerB`。

A 客户端使用 **MQTT V3.1.1** 协议连接 **EMQX**, 根据上文的配置规则, 代理订阅功能会主动帮客户端订阅 **QoS 为 1** 的 `client/testclientA` 和 **QoS 为 2** 的 `user/testerA` 这两个主题, 因为连接协议为 **MQTT V3.1.1**, 所以配置中的 **No Local**、**Retain As Published**、**Retain Handling** 不生效。

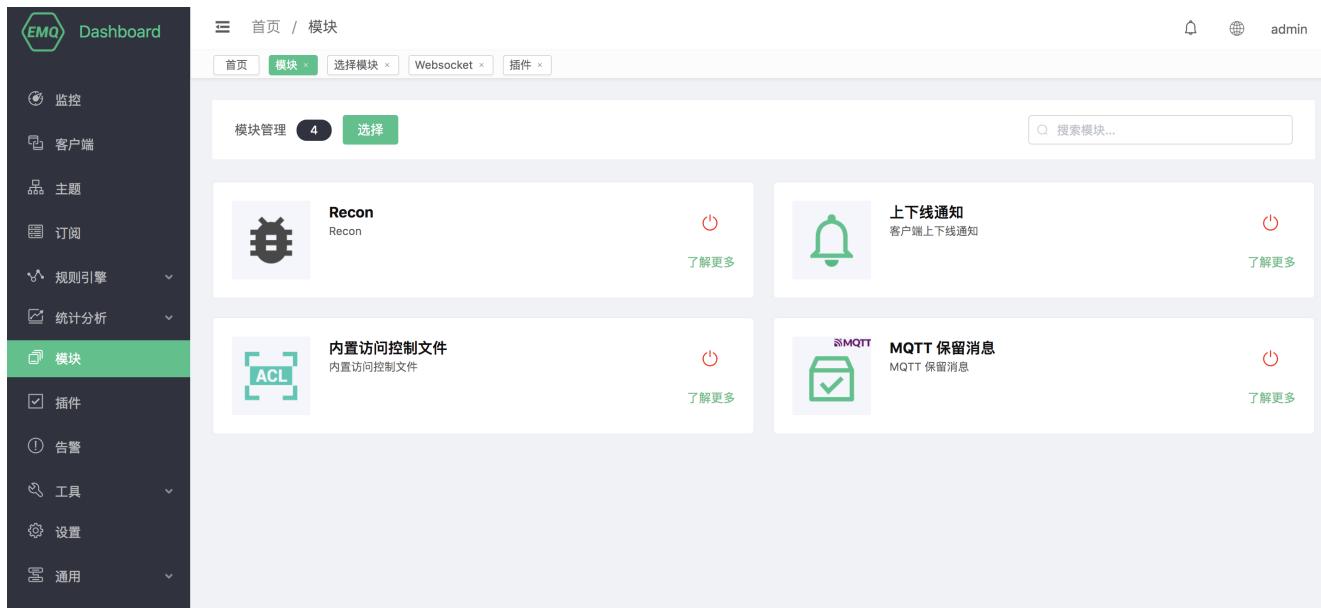
B 客户端使用 **MQTT V5** 协议连接 **EMQX**, 根据上文的配置规则, 代理订阅功能会主动帮客户端订阅 `client/testclientB` 和 `user/testerB` 这两个主题, 其中 `client/testclientB` 的订阅选项为 **Qos = 1**, **No Local**、**Retain As Published**、**Retain Handling** 均为 **0**; `user/testerB` 的订阅选项为 **Qos = 2**、**No Local = 1**、**Retain As Published = 1**、**Retain Handling = 1**。

主题重写

EMQX 的主题重写功能支持根据用户配置的规则在客户端订阅主题、发布消息、取消订阅的时候将 **A** 主题重写为 **B** 主题。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



The screenshot shows the EMQX Dashboard interface. On the left, there's a dark sidebar with various options like Monitoring, Clients, Topics, Subscriptions, Rules Engine, Statistics, Plugins, Alarms, Tools, Settings, and General. The 'Modules' option is highlighted with a green background. The main content area has a light gray background. At the top, there's a navigation bar with '首页 / 模块' (Home / Modules), a user icon, and the word 'admin'. Below the navigation is a search bar labeled '搜索模块...' (Search module...). A button bar at the top right includes '模块管理' (Module Management) with a count of 4, a '选择' (Select) button, and a '搜索模块...' input field. The main area displays four module cards: 'Recon' (with a robot icon), '上下线通知' (with a bell icon), '内置访问控制文件' (with an ACL icon), and 'MQTT 保留消息' (with a checkmark icon). Each card has a small red circular icon with a white symbol (possibly a gear or power icon) and a '了解更多' (More information) link.

选择 **MQTT** 主题重写模块

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left is a dark sidebar with various navigation items like Dashboard, Monitoring, Client, Topic, Subscription, Rules Engine, Statistics Analysis, Topic Monitoring, Alerts, Tools, Settings, and General. The '模块' (Module) item is currently selected. The main content area has a header with '首页 / 模块 / 选择' and a search bar. Below the header, there are tabs for '选择模块' (27), '认证授权' (selected), '协议接入', '消息下发', '多语言扩展' (highlighted with a red border), '运维监控', and '内部模块'. A search input field '搜索模块...' is also present. The main content area is divided into sections: '多语言扩展' (Multi-language Extension), '运维监控' (Operations Monitoring), and '内部模块' (Internal Modules). The '多语言扩展' section contains '多语言扩展协议接入' (Multi-language Extension Protocol Access) and 'exhook'. The '运维监控' section contains 'EMQ X Reloader' (EMQ X Beam Reloader) and 'Recon'. The '内部模块' section contains '主题监控' (Topic Monitoring), 'MQTT 增强认证' (MQTT Enhanced Authentication), '上下线通知' (Online/Offline Notification), 'MQTT 代理订阅' (MQTT Broker Subscription), '主题重写' (Topic Rewrite) (highlighted with a red border), 'MQTT 保留消息' (MQTT Retained Message), and '延迟发布' (Delayed Publish).

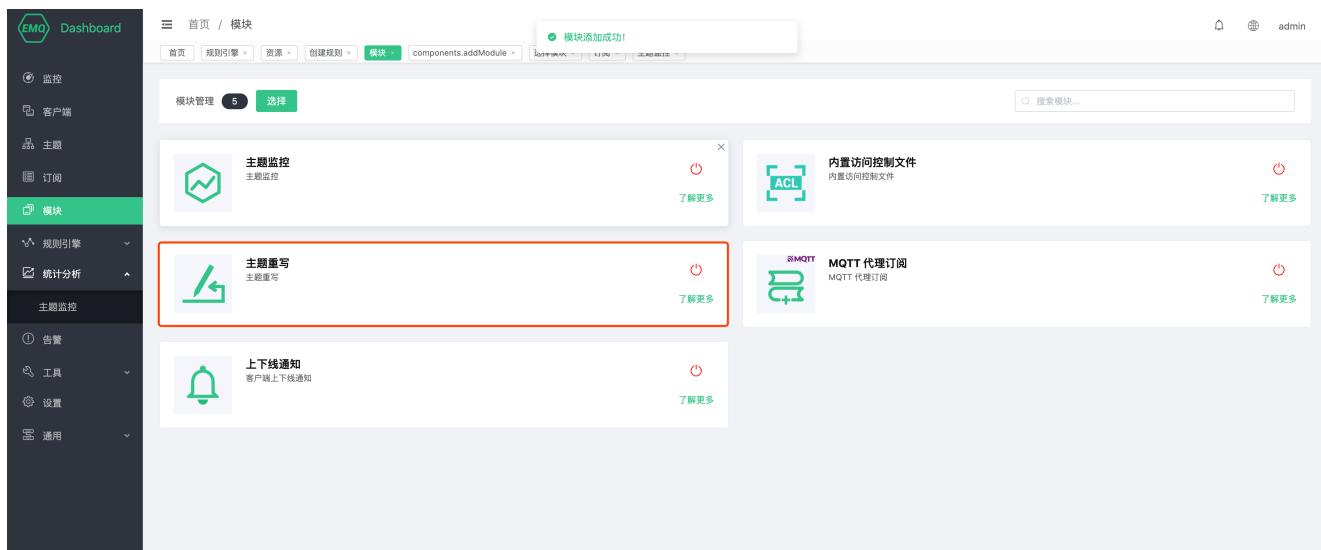
配置相关参数

The screenshot shows the 'Topic Rewrite' configuration page. The sidebar on the left is identical to the previous one. The main content area has a header with '首页 / 模块 / 详情' and a search bar. Below the header, there is a '主题重写' (Topic Rewrite) section with a sub-section '配置信息' (Configuration Information). This section contains a table for '主题重写规则' (Topic Rewrite Rules). The table has columns for '动作类型' (Action Type), '原始主题' (Original Topic), '正则表达式' (Regular Expression), '目标主题' (Target Topic), and '添加' (Add). There are three rows in the table:

动作类型	原始主题	正则表达式	目标主题	添加
subscribe	y/+/{z}/#	^y/(.+)/z/(.+)\${	y/z/\${2}	添加
publish	x/#	^x/(.+)\${	z/y/x/\${1}	删除
publish	x/y/+	^x/y/(d+){	z/y/\${1}	删除

At the bottom of the configuration section are '取消' (Cancel) and '添加' (Add) buttons.

点击添加后，模块添加完成



主题重写规则

重写规则分为 **Pub** 规则和 **Sub** 规则，**Pub** 规则匹配 **PUBLISH** 报文携带的主题，**Sub** 规则匹配 **SUBSCRIBE**、**UNSUBSCRIBE** 报文携带的主题。

每条重写规则都由主题过滤器、正则表达式、目标表达式三部分组成。在主题重写功能开启的前提下，**EMQX** 在收到诸如 **PUBLISH** 报文等带有主题的 **MQTT** 报文时，将使用报文中的主题去依次匹配配置文件中规则的主题过滤器部分，一旦成功匹配，则使用正则表达式提取主题中的信息，然后替换至目标表达式以构成新的主题。

目标表达式中可以使用 `$N` 这种格式的变量匹配正则表达中提取出来的元素，`$N` 的值为正则表达式中提取出来的第 **N** 个元素，比如 `$1` 即为正则表达式提取的第一个元素。

需要注意的是，**EMQX** 使用倒序读取配置文件中的重写规则，当一条主题可以同时匹配多条主题重写规则的主题过滤器时，**EMQX** 仅会使用它匹配到的第一条规则进行重写，如果该条规则中的正则表达式与 **MQTT** 报文主题不匹配，则重写失败，不会再尝试使用其他的规则进行重写。因此用户在使用时需要谨慎的设计 **MQTT** 报文主题以及主题重写规则。

主题重写示例

添加上图中的主题重写规则并分别订阅 `y/a/z/b`、`y/def`、`x/1/2`、`x/y/2`、`x/y/z` 五个主题：

- 当客户端订阅 `y/def` 主题时，`y/def` 不匹配任何一个主题过滤器，因此不执行主题重写，直接订阅 `y/def` 主题。
- 当客户端订阅 `y/a/z/b` 主题时，`y/a/z/b` 匹配 `y/+/#` 主题过滤器，**EMQX** 执行 `module.rewrite.sub.rule.1` 规则，通过正则表达式匹配出元素 `[a, b]`，将匹配出来的第二个元素带入 `y/z/$2`，实际订阅了 `y/z/b` 主题。
- 当客户端向 `x/1/2` 主题发送消息时，`x/1/2` 匹配 `x/#` 主题过滤器，**EMQX** 执行 `module.rewrite.pub.rule.1` 规则，通过正则表达式未匹配到元素，不执行主题重写，因此直接向 `x/1/2` 主题发送消息。
- 当客户端向 `x/y/2` 主题发送消息时，`x/y/2` 同时匹配 `x/#` 和 `x/y/+` 两个主题过滤器，**EMQX** 通过倒序读取配置，所以优先匹配 `module.rewrite.pub.rule.2`，通过正则替换，实际向 `z/y/2` 主题发送消息。
- 当客户端向 `x/y/z` 主题发送消息时，`x/y/z` 同时匹配 `x/#` 和 `x/y/+` 两个主题过滤器，**EMQX** 通过倒序读

取配置，所以优先匹配 `module.rewrite.pub.rule.2`，通过正则表达式未匹配到元素，不执行主题重写，直接向 `x/y/z` 主题发送消息。需要注意的是，即使 `module.rewrite.pub.rule.2` 的正则表达式匹配失败，也不会再次去匹配 `module.rewrite.pub.rule.1`` 的规则。

MQTT 保留消息

简介

服务端收到 **Retain** 标志为 **1** 的 **PUBLISH** 报文时，会将该报文视为保留消息，除了被正常转发以外，保留消息会被存储在服务端，每个主题下只能存在一份保留消息，因此如果已经存在相同主题的保留消息，则该保留消息被替换。

当客户端建立订阅时，如果服务端存在主题匹配的保留消息，则这些保留消息将被立即发送给该客户端。借助保留消息，新的订阅者能够立即获取最近的状态，而不需要等待无法预期的时间，这在很多场景下非常重要的。

创建模块

MQTT 保留消息 模块默认启动，可以通过**dashboard** 页面进行停止和更新。

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡：

The screenshot shows the EMQX Dashboard interface. On the left, a dark sidebar menu is open with various options: 监控 (Monitoring), 客户端 (Clients), 主题 (Topics), 订阅 (Subscriptions), 规则引擎 (Rule Engine), 统计分析 (Statistics), 模块 (Modules) [highlighted in green], 插件 (Plugins), 告警 (Alerts), 工具 (Tools), 设置 (Settings), and 通用 (General). The main content area is titled '首页 / 模块' and shows a '模块管理' section with four items: 'Recon' (status off), '内置访问控制文件' (ACL, status off), '上下线通知' (Online/Offline Notifications, status off), and 'MQTT 保留消息' (MQTT Retention, status off). A search bar '搜索模块...' is at the top right.

配置相关参数

The screenshot shows the 'MQTT 保留消息' configuration page. The left sidebar is identical to the previous one. The main page title is '首页 / 模块 / 详情'. The configuration section is titled 'MQTT 保留消息' and contains a sub-section '配置信息' (Configuration Information). This section includes fields for '存储类型' (Storage Type) set to 'ram', '最大保留消息数' (Max Retention Count) set to '0', '最大保留消息大小' (Max Message Size) set to '1MB', and '有效期限' (Expiration Time) set to '0'. At the bottom are '取消' (Cancel) and '添加' (Add) buttons.

保留消息配置简介

配置项	类型	可取值	默认值	说明
存储类型	enum	ram , disc , disc_only	ram	ram : 仅储存在内存中; disc : 储存在内存和硬盘中; disc_only : 仅储存在硬盘中。
最大保留消息数	integer	>= 0	0	保留消息的最大数量, 0 表示没有限制。保留消息数量超出最大值限制后, 可以替换已存在的保留消息, 但不能为新的主题储存保留消息。
最大保留消息大小	bytesize		1MB	保留消息的最大 Payload 值。 Payload 大小超出最大值后 EMQ X 消息服务器会把收到的保留消息作为普通消息处理。
有效期限	duration		0	保留消息的过期时间, 0 表示永不过期。如果 PUBLISH 报文中设置了消息过期间隔, 那么以 PUBLISH 报文中的消息过期间隔为准。
拦截清理消息	boolean	true , false	false	是否拦截清理保留消息的发布

Trace 日志追踪

简介

针对指定 **ClientID** 或 **Topic** 或 **IP** 实时过滤日志，用于调试和排查错误。

创建模块

Trace 日志追踪模块默认关闭，可以通过**dashboard**页面进行开启和停止。

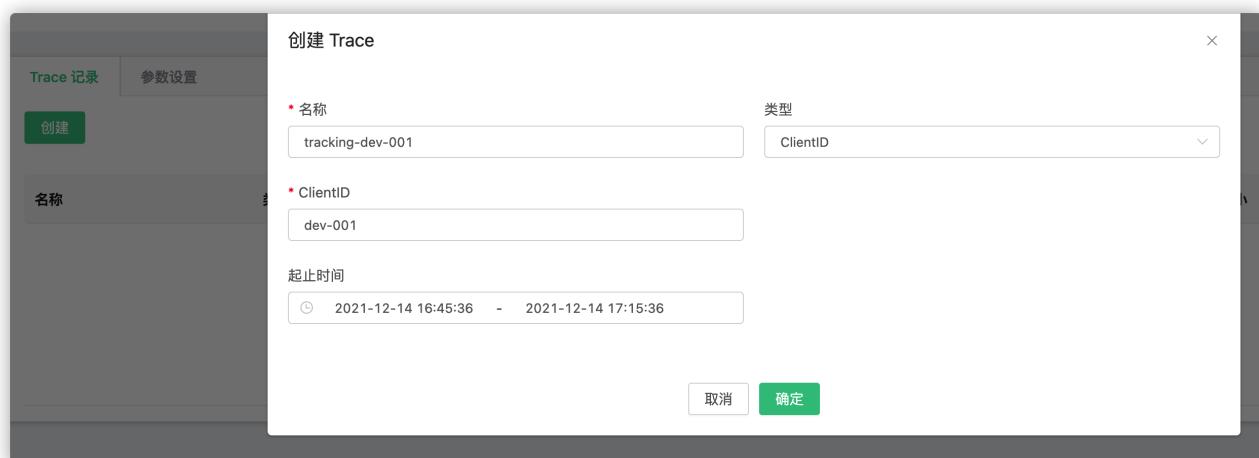
打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡：



可以创建追踪指定 **ClientID** 或 **Topic** 或 **IP** 客户端的日志。

追踪指定 ClientID

1. 点击创建后，选择类型为 **ClientID**；
2. 填写需要追踪的 **ClientID** 信息（必须是精确的**ClientID**）；
3. 选择起止时间，如果开始时间小于等于当前时间，会默认从当前时间开始。

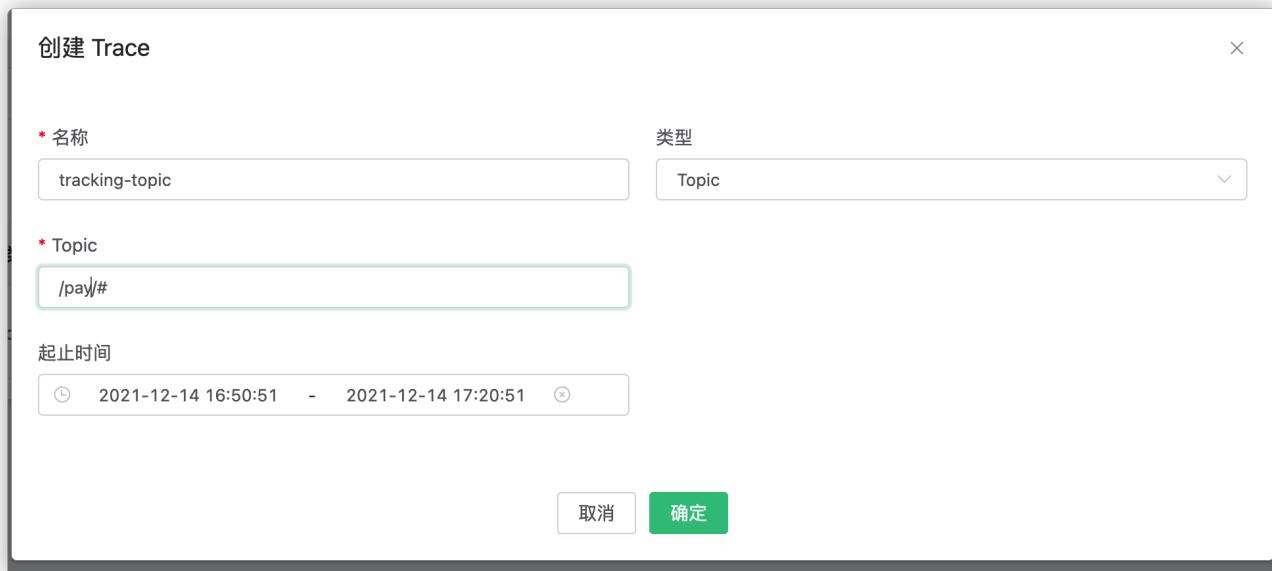


名称	类型	条件	起止时间	状态	日志大小	操作
tracking-dev-001	clientid	dev-001	2021-12-14 16:48:34 2021-12-14 17:15:36	运行中	0.00KB	查看 下载 停止

创建成功后，可以在列表中看到当前的**Trace**记录，可以查看，或下载日志。日志内容包含了当前 **ClientID** 与 **EMQX** 连接所有交互信息。

追踪指定 Topic

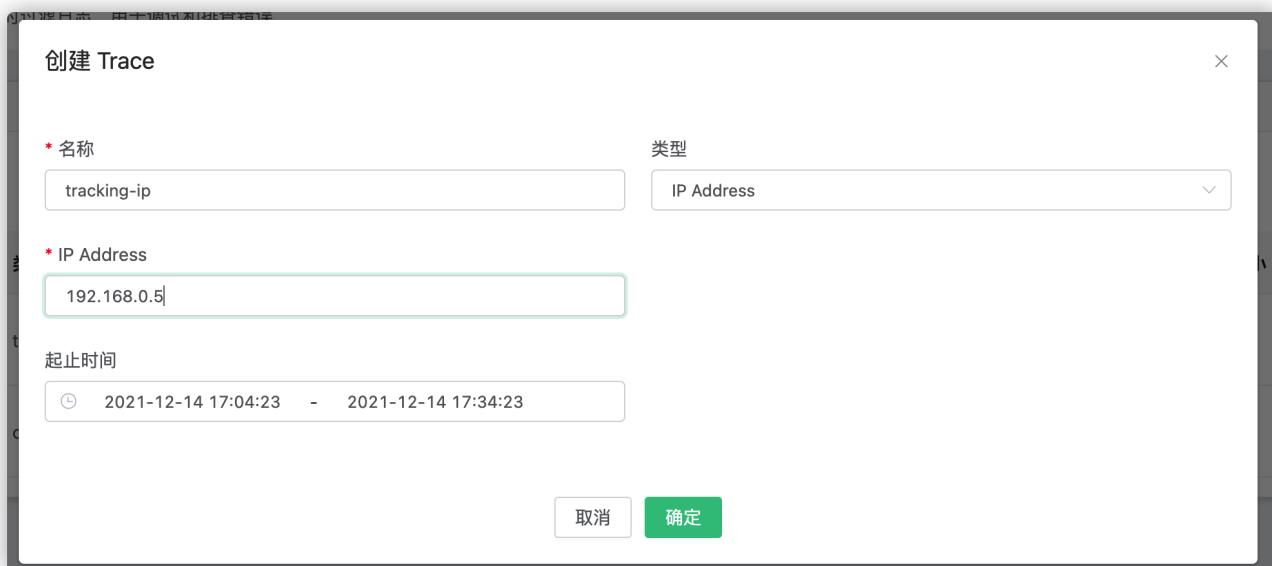
1. 点击创建后，选择类型为 **Topic**；
2. 填写需要追踪的 **Topic** 信息（支持通配符）；
3. 选择起止时间，如果开始时间小于等于当前时间，会默认从当前时间开始。



创建成功后，可以在列表中看到当前的**Trace**记录，可以查看，或下载日志。 日志内容包含了当前主题在 **EMQX** 上的 **Publish/Subscribe/UnSubscribe** 信息。

追踪指定 IP

1. 点击创建后，选择类型为 **IP Address**；
2. 填写需要追踪的 IP 地址信息（必须是精确的IP）；
3. 选择起止时间，如果开始时间小于等于当前时间，会默认从当前时间开始。



创建成功后，可以在列表中看到当前的**Trace**记录，可以查看，或下载日志。 日志内容包含了当前 IP 与 **EMQX** 连接所有交互信息。

Trace 记录		参数设置			
创建					
名称	类型	条件	起止时间	状态	日志大小
tracking-ip	ip_address	192.168.0.5	2021-12-14 17:05:07 2021-12-14 17:34:23	运行中	0.00KB
tracking-topic	topic	/pay/#	2021-12-14 17:03:50 2021-12-14 17:20:51	运行中	0.00KB
tracking-dev-001	clientid	dev-001	2021-12-14 16:48:34 2021-12-14 17:15:36	运行中	0.00KB

注意事项

- 最多可保存**30**个追踪日志。
- 追踪日志在每个节点产生的日志最大为 **512M**。如果产生的日志文件达到最大值，则会停止追加日志，并在主日志文件中给出提示。
- 可以选择手动停止记录，或等到结束时间时自动停止。
- 列表中查看到的日志文件大小为未压缩过的文件大小总和。
- EMQX** 集群重启后，会继续未完成的追踪。

慢订阅统计

该功能按照消息传输的耗时，从高到低对订阅者进行排名

开启模块

打开 **EMQX Dashboard**，点击左侧的“模块”选项卡，选择添加：

模块名称	描述	状态	操作
emqx_mod_delayed	EMQ X Delayed Publish Module	● 已停用	<button>启用</button>
emqx_mod_topic_metrics	EMQ X Topic Metrics Module	● 已停用	<button>启用</button>
emqx_mod_subscription	EMQ X Subscription Module	● 已停用	<button>启用</button>
emqx_mod_acl_internal	EMQ X Internal ACL Module	● 已启用	<button>停用</button>
emqx_mod_rewrite	EMQ X Topic Rewrite Module	● 已停用	<button>启用</button>
emqx_mod_presence	EMQ X Presence Module	● 已启用	<button>停用</button>
emqx_mod_trace	EMQ X Trace Module	● 已停用	<button>启用</button>
emqx_mod_slow_subs	EMQ X Slow Subscribers Statistics Module	● 已停用	<button>启用</button>

选择 **慢订阅统计** 模块，然后点击 **启动** 即可

实现说明

该功能会追踪 **QoS1** 和 **QoS2** 消息到达 **EMQX** 后，完成消息传输全流程的时间消耗，然后根据配置中的选项，计算消息的传输时延，之后按照时延高低对订阅者、主题进行统计排名

配置说明

- 时延阈值/**threshold**

时延阈值 用来判断订阅者是否可以参与统计，如果订阅者的时延低于这个值，将不会进行统计

- 最大统计条数/**top_k_num**

这个字段决定统计记录表中数量上限

- 有效时长/**expire_interval**

有效时长 控制统计记录中每一条数据的有效时间，如果该数据在这个时间范围内，一直没有被更新过，将会被移除（比如发送一条消息后因为时延很长，被加入到统计记录中，之后长时间没有再次发送消息，在超过这个字段后，将会被清除掉）

- 统计类型/**stats_type** 计算时延的方式，分别为：

- whole**

从消息到达 **EMQX** 时起，直到消息完成传输时

- internal**

从消息到达 **EMQX** 时起，直到 **EMQX** 开始投递消息时

3. response

从 **EMQX** 开始投递消息时起，直到消息完成传输时

消息完成传输的定义：

1. QoS0

EMQX 开始投递时

2. QoS1

EMQX 收到客户端的 **puback** 时

3. QoS2

EMQX 收到客户端的 **pubcomp** 时

注意：开源版配置在 **emqx.conf** 中

慢订阅记录

Client ID	主题	时长	节点	更新时间
s2	/test/5	53 ms	emqx@127.0.0.1	2022-02-09 10:22:36
s2	/test/2	38 ms	emqx@127.0.0.1	2022-02-09 10:22:21
s2	/test/4	37 ms	emqx@127.0.0.1	2022-02-09 10:22:32
s2	/test/3	36 ms	emqx@127.0.0.1	2022-02-09 10:22:27
s2	/test/1	33 ms	emqx@127.0.0.1	2022-02-09 10:22:17
s1	/test/4	29 ms	emqx@127.0.0.1	2022-02-09 10:22:31
s1	/test/2	28 ms	emqx@127.0.0.1	2022-02-09 10:22:21
s1	/test/5	27 ms	emqx@127.0.0.1	2022-02-09 10:22:36
s1	/test/3	26 ms	emqx@127.0.0.1	2022-02-09 10:22:26

这个标签页下会按照时延，从高到底依次显示订阅者和主题信息，点击 **Client ID** 将会显示订阅者详情，可以通过订阅者详情来进行问题分析 和查找。

延迟发布

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

选择延迟发布模块，无需配置参数，直接开启

延迟发布简介

EMQX 的延迟发布功能可以实现按照用户配置的时间间隔延迟发布 **PUBLISH** 报文的功能。当客户端使用特殊主题前缀 `$delayed/{DelayInterval}` 发布消息到 **EMQX** 时，将触发延迟发布功能。

延迟发布主题的具体格式如下：

```
1 $delayed/{DelayInterval}/{TopicName}
```

sh

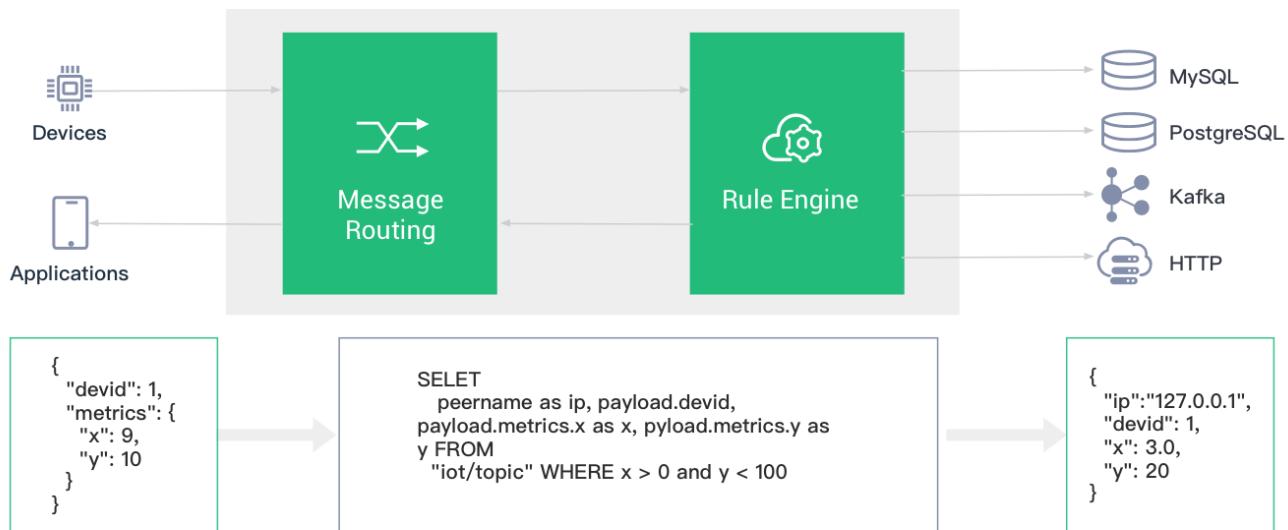
- `$delayed`：使用 `$delay` 作为主题前缀的消息都将被视为需要延迟发布的消息。延迟间隔由下一主题层级中的内容决定。
- `{DelayInterval}`：指定该 **MQTT** 消息延迟发布的时间间隔，单位是秒，允许的最大间隔是 **4294967** 秒。如果 `{DelayInterval}` 无法被解析为一个整型数字，**EMQX** 将丢弃该消息，客户端不会收到任何信息。
- `{TopicName}`：**MQTT** 消息的主题名称。

例如：

- `$delayed/15/x/y`：15 秒后将 **MQTT** 消息发布到主题 `x/y`。
- `$delayed/60/a/b`：1 分钟后将 **MQTT** 消息发布到 `a/b`。
- `$delayed/3600/$SYS/topic`：1 小时后将 **MQTT** 消息发布到 `$SYS/topic`。

规则引擎

EMQX Rule Engine (以下简称规则引擎) 用于配置 EMQX 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 EMQX 的耦合度；也为 EMQX 的私有功能定制提供了一个更优秀的基础架构。



EMQX 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。

提示

适用版本: **EMQX v3.1.0+**

兼容提示: **EMQX 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。

EMQX 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。

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：

```
1 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：

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

事件触发

规则引擎使用 **\$events/** 开头的虚拟主题（事件主题）处理 **EMQX** 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：

```
1 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 [SQL 手册](#)。

规则引擎组成

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：

The screenshot shows the EMQX Dashboard with the 'Rule Engine' selected in the sidebar. The main area displays a rule named 'rule1' with the following configuration:

- 规则 SQL:**

```
SELECT payload.msg, topic, qos
FROM "#"
```
- 响应动作:**
 - 动作类型: 发送数据到 Web 服务 (data_to_webserver)
将数据转发给 Web 服务
详细统计 点击查看 消息内容模板 资源 ID resource:a1caa39
 - 动作类型: 消息重新发布 (republish)
重新发布消息到另一个主题
详细统计 点击查看 目的主題 target_topic/uplink 目的 QoS 0 消息内容模板 \${payload}

使用 **EMQX** 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 **EMQX** 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：

```

1  规则: {
2      SQL 语句,
3      动作列表: [
4          {
5              动作1,
6              动作参数,
7              绑定资源: {
8                  资源配置
9              }
10         },
11         {
12             动作2,
13             动作参数,
14             绑定资源: {
15                 资源配置
16             }
17         }
18     ]
19 }
```

- **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。

提示

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。

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 **EMQX** 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。

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。

1. 在创建规则界面，输入 规则**SQL**，并启用 **SQL** 测试 开关：

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4   "t/#"
5   WHERE
6     qos = 1
```

备注:

SQL 测试: [?](#)

username: u_emqx

topic: t/a

qos: 1

payload:

```
1 {"msg": "hello"}
```

JSON RAW

2. 修改模拟事件的字段，或者使用默认的配置，点击 测试 按钮：

username:	<input type="text" value="u_emqx"/>
topic:	<input type="text" value="t/a"/>
qos:	<input type="text" value="1"/>
payload:	<input "hello"}"="" msg":="" type="text" value="1 {"/> ... <input checked="" type="radio"/> JSON <input type="radio"/> RAW
clientid:	<input type="text" value="c_emqx"/>
<input style="background-color: #28a745; color: white; border-radius: 5px; padding: 5px; margin-bottom: 10px;" type="button" value="SQL 测试"/>	
测试输出:	<div style="border: 1px solid #ccc; padding: 10px; height: 150px;"></div>

3. SQL 处理后的结果将在 测试输出 文本框里展示:

<input style="background-color: #28a745; color: white; border-radius: 5px; padding: 5px; margin-bottom: 10px;" type="button" value="SQL 测试"/>	
测试输出:	<div style="border: 1px solid #ccc; padding: 10px; height: 150px;"><pre>{ "username": "u_emqx", "topic": "t/a", "timestamp": 1587533697725, "qos": 1, "peerhost": "127.0.0.1", "payload": "{\"msg\": \"hello\"}", "node": "emqx@127.0.0.1", "id": "5A3DA7E1F3507F4430000197A0003", "flags": { "sys": true, "event": true }, "clientid": "c_emqx" }</pre></div>

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：

```
1 ## 3.x 版本
2 ## 需要指定事件名称进行处理
3 SELECT * FROM "t/#" WHERE topic =~ 't/#'
4
5 ## 4.0 及以后版本
6 ## 默认处理 message.publish 事件, FROM 后面直接填写 MQTT 主题
7 ## 上述 SQL 语句等价于:
8 SELECT * FROM 't/#'
9
10 ## 其他事件, FROM 后面填写事件主题
11 SELECT * FROM "$events/message_acked" where topic =~ 't/#'
12 SELECT * FROM "$events/client_connected"
```

提示

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

[下一部分, 规则引擎语法和示例](#)

规则引擎语法与示例

SQL 语法

FROM、SELECT 和 WHERE 子句

规则引擎的 SQL 语句基本格式为：

```
1   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

```
1 ## SELECT 语句用于决定最终的输出结果里的字段。比如：
2 ## 下面 SQL 的输出结果中将只有两个字段 "a" 和 "b"：
3
4 SELECT a, b FROM "#"
5
6 # 选取 username 为 'abc' 的终端发来的消息，输出结果为所有可用字段：
7
8 SELECT * FROM "#" WHERE username = 'abc'
9
10 ## 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。
11 ## 注意 cid 变量是在 SELECT 语句中定义的，故可在 WHERE 语句中使用：
12
13 SELECT clientid as cid FROM "#" WHERE cid = 'abc'
14
15 ## 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。
16 ## 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：
17
18 SELECT clientid as cid FROM "#" WHERE username = 'abc'
19
20 ## 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：
21
22 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。

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：

```
1   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

```

1
2     FOREACH
3         payload.sensors as e ## 选择出的字段必须为数组类型
4             DO ## DO相当于针对当前循环中对象的 SELECT 子句，决定最终的输出结果里的字段
5                 clientid,
6                 e.name as name,
7                 e.idx as idx
8             INCASE ## INCASE 相当于针对当前循环中对象的 WHERE 语句
9                 e.idx >= 1 ## 对DO选择出来的某个字段施加条件过滤
10            FROM "t/#" ## 子句将规则挂载到某个主题上

```

其中 **DO** 和 **INCASE** 子句都是可选的。

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

比较符号

函数名	函数作用	返回值
>	大于	true/false
<	小于	true/false
<=	小于等于	true/false
>=	大于等于	true/false
<>	不等于	true/false
!=	不等于	true/false

SQL 语句示例

基本语法举例

- 从 **topic** 为 "t/a" 的消息中提取所有字段:

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

- 从 **topic** 为 "t/a" 或 "t/b" 的消息中提取所有字段:

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

- 从 **topic** 能够匹配到 't/#' 的消息中提取所有字段。

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

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

```
1 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'**:

```
1 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}:

```
1 SELECT payload FROM "#" WHERE payload.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}}:

```
1 SELECT payload FROM "#" WHERE payload.x.y = 1
```

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

```
1 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**:

```
1 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:

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

- 对于一个 **MQTT 5.0 PUBLISH** 消息，筛选出 **Key** 为 "foo" 的 **User Property**:

```
1   SELECT pub_props.'User-Property'.foo as foo FROM "t/#"
```

提示

- FROM** 子句后面的主题需要用双引号 " "，或者单引号 ' ' 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。
- 尽量不要给 **payload** 创建别名，否则会影响运行性能。即尽量不要这么写： **SELECT payload as p**

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：

```
1   {
2     "date": "2020-04-24",
3     "sensors": [
4       {"name": "a", "idx": 0},
5       {"name": "b", "idx": 1},
6       {"name": "c", "idx": 2}
7     ]
8 }
```

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：

```
1   FOREACH
2     payload.sensors
3     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:

```

1  [
2    {
3      "name": "a",
4      "idx": 0
5    },
6    {
7      "name": "b",
8      "idx": 1
9    },
10   {
11     "name": "c",
12     "idx": 2
13   }
14 ]

```

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:

```

1 FOREACH
2   payload.sensors
3 DO
4   clientid,
5   item.name as name,
6   item.idx as idx
7 INCASE
8   item.idx >= 1
9 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:

```

1  [
2    {
3      "name": "b",
4      "idx": 1,
5      "clientid": "c_emqx"
6    },
7    {
8      "name": "c",
9      "idx": 2,
10     "clientid": "c_emqx"
11   }
12 ]

```

json

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：

```

1 FOREACH
2   payload.sensors as s
3 DO
4   clientid,
5   s.name as name,
6   s.idx as idx
7 INCASE
8   s.idx >= 1
9 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：

```

1 FOREACH
2   payload.sensors as s
3 DO
4   nth(2, tokens(clientid, '_')) as clientid,
5   s.name as name,
6   s.idx as idx
7 INCASE
8   s.idx >= 1
9 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**：

```

1  {
2      "date": "2020-04-24",
3      "data": {
4          "sensors": [
5              {"name": "a", "idx": 0},
6              {"name": "b", "idx": 1},
7              {"name": "c", "idx": 2}
8          ]
9      }
10 }

```

json

则 **FOREACH** 中可以在决定要遍历的数组之前把 **data** 选取出来：

```

1 FOREACH
2     payload.data as data
3     data.sensors as s
4 ...

```

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。

```

1 SELECT
2     CASE WHEN payload.x < 0 THEN 0
3         WHEN payload.x > 7 THEN 7
4         ELSE payload.x
5     END as x
6     FROM "t/#"

```

假设消息为：

```

1 {"x": 8}

```

json

则上面的 **SQL** 输出为：

```

1 {"x": 7}

```

json

数组操作语法举例

示例1: 创建一个数组，赋值给变量 **a**:

```

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

```

下标从 **1** 开始，上面的 **SQL** 输出为：

json

```

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

```

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

上面的 **SQL** 输出为:

```

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

示例3: 从 **JSON** 格式的 **payload** 中嵌套的获取值:

```

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

假设消息为:

```

1  {"data": [
2      {"id": 1, "name": "steve"},
3      {"id": 2, "name": "bill"}
4  ]}
```

json

则上面的 **SQL** 输出为:

```

1  {"id": 1}
```

json

示例4: 数组范围(**range**)操作:

```

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

上面的 **SQL** 输出为:

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

json

示例5：使用下标语法修改数组中的某个元素：

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

假设消息为：

```
1 {"data": [  
2     {"id": 1, "name": "steve"},  
3     {"id": 2, "name": "bill"}  
4 ]}
```

json

则上面的 **SQL** 输出为：

```
1 {  
2     "payload": {  
3         "data": [  
4             {"name": "STEVE", "id": 1},  
5             {"name": "bill", "id": 2}  
6         ]  
7     }  
8 }
```

json

[下一部分，规则 SQL 语句中可用的字段](#)

规则引擎 SQL 语句中可用的字段

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。

使用规则引擎 SQL 语句处理消息发布

规则引擎的 **SQL** 语句可以处理消息发布。在一个规则语句中，用户可以用 **FROM** 子句指定一个或者多个主题，当任何消息发布到指定的主题时都会触发该规则。

字段	解释
id	MQTT 消息 ID
clientid	消息来源 Client ID
username	消息来源用户名
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
pub_props	PUBLISH Properties (仅适用于 MQTT 5.0)
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

示例

```

1  SELECT
2      payload.msg as msg,
3      clientid,
4      username,
5      payload,
6      topic,
7      qos
8  FROM
9      "t/#"

```

输出

```

1  {
2    "username": "u_emqx",
3    "topic": "t/a",
4    "qos": 1,
5    "payload": "{\"msg\":\"hello\"}",
6    "msg": "hello",
7    "clientid": "c_emqx"
8  }

```

使用规则引擎 SQL 语句处理事件

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。

FROM 子句可用的事件主题

事件主题名	释义
<code>\$events/message_delivered</code>	消息投递
<code>\$events/message_acked</code>	消息确认
<code>\$events/message_dropped</code>	消息丢弃
<code>\$events/client_connected</code>	连接完成
<code>\$events/client_disconnected</code>	连接断开
<code>\$events/client_connack</code>	连接确认
<code>\$events/client_check_acl_complete</code>	鉴权结果
<code>\$events/session_subscribed</code>	订阅
<code>\$events/session_unsubscribed</code>	取消订阅

\$events/message_delivered (消息投递)

当消息被放入底层**socket**时触发规则

字段	解释
id	MQTT 消息 ID
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
pub_props	PUBLISH Properties (仅适用于 MQTT 5.0)
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

示例

```

1  SELECT
2      from_clientid,
3      from_username,
4      topic,
5      qos,
6      node,
7      timestamp
8  FROM
9      "$events/message_delivered"

```

输出

```

1  {
2      "topic": "t/a",
3      "timestamp": 1645002753259,
4      "qos": 1,
5      "node": "emqx@127.0.0.1",
6      "from_username": "u_emqx_1",
7      "from_clientid": "c_emqx_1"
8  }

```

json

\$events/message_acked (消息确认)

当消息发送到客户端，并收到客户端回复的**ack**时触发规则，仅**QOS1**, **QOS2**会触发

字段	解释
id	MQTT 消息 ID
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
pub_props	PUBLISH Properties (仅适用于 MQTT 5.0)
puback_props	PUBACK Properties (仅适用于 MQTT 5.0)
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

示例

```

1  SELECT
2      from_clientid,
3      from_username,
4      topic,
5      qos,
6      node,
7      timestamp
8  FROM
9      "$events/message_acked"

```

输出

```

1  {
2      "topic": "t/a",
3      "timestamp": 1645002965664,
4      "qos": 1,
5      "node": "emqx@127.0.0.1",
6      "from_username": "u_emqx_1",
7      "from_clientid": "c_emqx_1"
8  }

```

json

\$events/message_dropped (消息在转发的过程中被丢弃)

当一条消息无任何订阅者时触发规则

字段	解释
id	MQTT 消息 ID
reason	消息丢弃原因, 可能的原因: no_subscribers : 没有订阅者
clientid	消息来源 Client ID
username	消息来源用户名
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
pub_props	PUBLISH Properties (仅适用于 MQTT 5.0)
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

示例

```

1  SELECT
2    reason,
3    topic,
4    qos,
5    node,
6    timestamp
7  FROM
8    "$events/message_dropped"

```

输出

```

1  {
2    "topic": "t/a",
3    "timestamp": 1645003103004,
4    "reason": "no_subscribers",
5    "qos": 1,
6    "node": "emqx@127.0.0.1"
7  }

```

json

\$events/delivery_dropped (消息在投递的过程中被丢弃)

当订阅者的消息队列已满时触发规则

字段	解释
id	MQTT 消息 ID
reason	消息丢弃原因，可能的原因： queue_full : 消息队列已满(QoS>0) no_local : 不允许客户端接收自己发布的消息 expired : 消息或者会话过期 qos0_msg : QoS0 的消息因为消息队列已满被丢弃
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
pub_props	PUBLISH Properties (仅适用于 MQTT 5.0)
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

示例

```

1  SELECT
2    from_clientid,
3    from_username,
4    reason,
5    topic,
6    qos
7   FROM "$events/delivery_dropped"

```

输出

```

1  {
2    "topic": "t/a",
3    "reason": "queue_full",
4    "qos": 1,
5    "from_username": "u_emqx_1",
6    "from_clientid": "c_emqx_1"
7  }

```

json

\$events/client_connected (终端连接成功)

当终端连接成功时触发规则

字段	解释
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)
conn_props	CONNECT Properties (仅适用于 MQTT 5.0)
timestamp	事件触发时间 (ms)
node	事件触发所在节点

示例

```

1  SELECT
2    clientid,
3    username,
4    keepalive,
5    is_bridge
6  FROM
7    "$events/client_connected"
```

输出

```

1  {
2    "username": "u_emqx",
3    "keepalive": 60,
4    "is_bridge": false,
5    "clientid": "c_emqx"
6  }
```

json

\$events/client_disconnected (终端连接断开)

当终端连接断开时触发规则

字段	解释
reason	终端连接断开原因： normal : 客户端主动断开 kicked : 服务端踢出, 通过 REST API keepalive_timeout : keepalive 超时 not_authorized : 认证失败, 或者 acl_nomatch = disconnect 时没有权限的 Pub/Sub 会主动断开客户端 tcp_closed : 对端关闭了网络连接 internal_error : 畸形报文或其他未知错误
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)
disconn_props	DISCONNECT Properties (仅适用于 MQTT 5.0)
timestamp	事件触发时间 (ms)
node	事件触发所在节点

示例

```

1  SELECT
2    clientid,
3    username,
4    reason,
5    disconnected_at,
6    node
7  FROM
8    "$events/client_disconnected"

```

输出

```

1  {
2    "username": "u_emqx",
3    "reason": "normal",
4    "node": "emqx@127.0.0.1",
5    "disconnected_at": 1645003578536,
6    "clientid": "c_emqx"
7  }

```

json

\$events/client_connack (连接确认)

当服务端向客户端发送**CONNACK**报文时触发规则, **reason_code** 包含各种错误原因代码

字段	解释
reason_code	各种原因代码
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
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
conn_props	CONNECT Properties (仅适用于 MQTT 5.0)
timestamp	事件触发时间 (ms)
node	事件触发所在节点

MQTT v5.0 协议将返回码重命名为原因码，增加了一个原因码来指示更多类型的错误([Reason code and ACK - MQTT 5.0 new features](#))。因此**reason_code** 在**MQTT v3.1.1**与**MQTT v5.0**中有很大的不同。

MQTT v3.1.1

reason_code	描述
connection_accepted	已接受连接
unacceptable_protocol_version	服务器不支持客户端请求的 MQTT 协议
client_identifier_not_valid	客户端 ID 是正确的 UTF-8 字符串，但服务器不允许
server_unavailable	网络连接已建立，但 MQTT 服务不可用
malformed_username_or_password	用户名或密码中的数据格式错误
unauthorized_client	客户端连接未授权

MQTT v5.0

reason_code	描述
success	连接成功
unspecified_error	未指定的错误
malformed_packet	畸形数据包
protocol_error	协议错误
implementation_specific_error	实现特定错误
unsupported_protocol_version	不支持的协议版本
client_identifier_not_valid	客户端标识符无效
bad_username_or_password	错误的用户名或密码
not_authorized	未经授权
server_unavailable	服务器无法使用
server_busy	服务器繁忙
banned	禁止访问
bad_authentication_method	错误的身份验证方法
topic_name_invalid	主题名称无效
packet_too_large	数据包太大
quota_exceeded	超出配额
retain_not_supported	不支持的retain
qos_not_supported	不支持的qos
use_another_server	使用另一台服务器
server_moved	服务器迁移了
connection_rate_exceeded	超出连接速率

示例

```

1  SELECT
2    clientid,
3    username,
4    reason,
5    node
6  FROM
7    "$events/client_connack"

```

输出

```

1  {
2      "username": "u_emqx",
3      "reason": "success",
4      "node": "emqx@127.0.0.1",
5      "connected_at": 1645003578536,
6      "clientid": "c_emqx"
7  }

```

json

\$events/client_check_acl_complete (鉴权结果)

当客户端鉴权结束时触发规则

字段	解释
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
action	publish or subscribe , 发布或者订阅事件
result	allow or deny , 鉴权结果
is_cache	true or false , 鉴权时数据的来源 is_cache 为 true 时, 鉴权数据来源于 cache is_cache 为 false 时, 鉴权数据来源于插件
timestamp	事件触发时间 (ms)
node	事件触发所在节点

示例

```

1  SELECT
2      clientid,
3      username,
4      topic,
5      action,
6      result,
7      is_cache,
8      node
9  FROM
10     "$events/client_check_acl_complete"

```

输出

json

```

1  {
2    "username": "u_emqx",
3    "topic": "t/a",
4    "action": "publish",
5    "result": "allow",
6    "is_cache": "false",
7    "node": "emqx@127.0.0.1",
8    "clientid": "c_emqx"
9  }

```

\$events/session_subscribed (终端订阅成功)

当终端订阅成功时触发规则

字段	解释
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
sub_props	SUBSCRIBE Properties (仅适用于 5.0)
timestamp	事件触发时间 (ms)
node	事件触发所在节点

示例

```

1  SELECT
2    clientid,
3    username,
4    topic,
5    qos
6  FROM
7    "$events/session_subscribed"

```

输出

json

```

1  {
2    "username": "u_emqx",
3    "topic": "t/a",
4    "qos": 1,
5    "clientid": "c_emqx"
6  }

```

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

当取消终端订阅成功时触发规则

字段	解释
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
unsub_props	UNSUBSCRIBE Properties (仅适用于 5.0)
timestamp	事件触发时间 (ms)
node	事件触发所在节点

示例

```

1  SELECT
2    clientid,
3    username,
4    topic,
5    qos
6  FROM
7    "$events/session_unsubscribed"

```

输出

```

1  {
2    "username": "u_emqx",
3    "topic": "t/a",
4    "qos": 1,
5    "clientid": "c_emqx"
6  }

```

json

[下一部分，规则引擎内置函数](#)

SQL 语句中可用的函数

数学函数

函数名	函数作用	参数	返回值
abs	绝对值	1. 被操作数	绝对值
cos	余弦	1. 被操作数	余弦值
cosh	双曲余弦	1. 被操作数	双曲余弦值
acos	反余弦	1. 被操作数	反余弦值
acosh	反双曲余弦	1. 被操作数	反双曲余弦值
sin	正弦	1. 被操作数	正弦值
sinh	双曲正弦	1. 被操作数	双曲正弦值
asin	反正弦	1. 被操作数	值
asinh	反双曲正弦	1. 被操作数	反双曲正弦值
tan	正切	1. 被操作数	正切值
tanh	双曲正切	1. 被操作数	双曲正切值
atan	反正切	1. 被操作数	反正切值
atanh	反双曲正切	1. 被操作数	反双曲正切值
ceil	上取整	1. 被操作数	整数值
floor	下取整	1. 被操作数	整数值
round	四舍五入	1. 被操作数	整数值
exp	幂运算	1. 被操作数	e 的 x 次幂
power	指数运算	1. 左操作数 x 2. 右操作数 y	x 的 y 次方
sqrt	平方根运算	1. 被操作数	平方根
fmod	负点数取模函数	1. 左操作数 2. 右操作数	模
log	以 e 为底对数	1. 被操作数	值
log10	以 10 为底对数	1. 被操作数	值
log2	以 2 为底对数	1. 被操作数	值

```

1  abs(-12) = 12
2  cos(1.5) = 0.0707372016677029
3  cosh(1.5) = 2.352409615243247
4  acos(0.0707372016677029) = 1.5
5  acosh(2.352409615243247) = 1.5
6  sin(0.5) = 0.479425538604203
7  sinh(0.5) = 0.5210953054937474
8  asin(0.479425538604203) = 0.5
9  asinh(0.5210953054937474) = 0.5
10 tan(1.4) = 5.797883715482887
11 tanh(1.4) = 0.8853516482022625
12 atan(5.797883715482887) = 1.4
13 atanh(0.8853516482022625) = 1.4000000000000001
14 ceil(1.34) = 2
15 floor(1.34) = 1
16 round(1.34) = 1
17 round(1.54) = 2
18 exp(10) = 22026.465794806718
19 power(2, 10) = 1024
20 sqrt(2) = 1.4142135623730951
21 fmod(-32, 5) = -2
22 log10(1000) = 3
23 log2(1024) = 10

```

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。

```

1  is_null(undefined) = true
2  is_not_null(1) = true
3  is_str(1) = false
4  is_str('val') = true
5  is_bool(true) = true
6  is_int(1) = true
7  is_float(1) = false
8  is_float(1.234) = true
9  is_num(2.3) = true
10 is_num('val') = false

```

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
float2str	将浮点型数字以指定精度转换为字符串	1. 浮点型数字 2. 精度	字符串
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

```

1  str(1234) = '1234'
2  str_utf8(1234) = '1234'
3  bool('true') = true
4  int('1234') = 1234
5  float('3.14') = 3.14
6  float2str(20.2, 10) = '20.2'
7  float2str(20.2, 17) = '20.1999999999999928'

```

注意浮点型转换为字符串的时候，输出结果会受到精度的影响，详情见：<https://floating-point-gui.de/>

字符串函数

函数名	函数作用	参数	返回值
lower	转为小写	1. 原字符串	小写字符串
upper	转为大写	1. 原字符串	大写字符串
trim	去掉左右空格	1. 原字符串	去掉空格后的字符串

函数名	函数作用	参数	返回值
ltrim	去掉左空格	1. 原字符串	去掉空格后的字符串
rtrim	去掉右空格	1. 原字符串	去掉空格后的字符串
reverse	字符串反转	1. 原字符串	翻转后的字符串
strlen	取字符串长度	1. 原字符串	整数值，字符长度
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
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
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
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
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
concat	字符串拼接	1. 左字符串 2. 右字符串	拼接后的字符串
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
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
sprintf	字符串格式化, 格式字符串的用法详见 https://erlang.org/doc/man/io.html#fwrite-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
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
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
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
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
pad	字符串补足长度, 补指定字符, 从尾部补足	1. 原字符串 2. 字符总长度 3. 'trailing' 4. 指定用于补足的字符	补足后的字符串

函数名	函数作用		返回值
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
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
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
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
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
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
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
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
ascii	返回字符对应的 ASCII 码	1. 字符	整数值，字符对应的 ASCII 码
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
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
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 lower('AbC') = 'abc'
2 lower('abc') = 'abc'
3
4 upper('AbC') = 'ABC' `` lower('ABC') = 'ABC'

```

```

5   trim(' hello  ') = 'hello'
6
7   ltrim(' hello  ') = 'hello '
8
9   rtrim(' hello  ') = ' hello'
10
11  reverse('hello') = 'olleh'
12
13  strlen('hello') = 5
14
15
16  substr('abcdef', 2) = 'cdef'
17  substr('abcdef', 2, 3) = 'cde'
18
19  split('a/b/ c', '/') = ['a', 'b', ' c']
20  split('a/b/ c', '/', 'leading') = ['a', 'b/ c']
21  split('a/b/ c', '/', 'trailing') = ['a/b', ' c']
22
23  concat('a', '/bc') = 'a/bc'
24  'a' + '/bc' = 'a/bc'
25
26  tokens(' a/b/ c', '/') = [' a', 'b', ' c']
27  tokens(' a/b/ c', '/ ') = ['a', 'b', 'c']
28  tokens(' a/b/ c\n', '/ ') = ['a', 'b', 'c\n']
29  tokens(' a/b/ c\n', '/ ', 'nocrlf') = ['a', 'b', 'c']
30  tokens(' a/b/ c\r\n', '/ ', 'nocrlf') = ['a', 'b', 'c']
31
32  sprintf('hello, ~s!', 'steve') = 'hello, steve!'
33  sprintf('count: ~p~n', 100) = 'count: 100\n'
34
35  pad('abc', 5) = 'abc  '
36  pad('abc', 5, 'trailing') = 'abc  '
37  pad('abc', 5, 'both') = ' abc '
38  pad('abc', 5, 'leading') = '  abc'
39  pad('abc', 5, 'trailing', '*') = 'abc**'
40  pad('abc', 5, 'trailing', '*#') = 'abc*##'
41  pad('abc', 5, 'both', '*') = '*abc*'
42  pad('abc', 5, 'both', '*#') = '*#abc*#'
43  pad('abc', 5, 'leading', '*') = '**abc'
44  pad('abc', 5, 'leading', '*#') = '**#abc'
45
46  replace('ababef', 'ab', 'cd') = 'cdcdef'
47  replace('ababef', 'ab', 'cd', 'all') = 'cdcdef'
48  replace('ababef', 'ab', 'cd', 'trailing') = 'abcdef'
49  replace('ababef', 'ab', 'cd', 'leading') = 'cdabef'
50
51  regex_match('abc123', '[a-zA-Z1-9]*') = true
52
53  regex_replace('ab1cd3ef', '[1-9]', '[&]') = 'ab[1]cd[3]ef'
54  regex_replace('ccefacef', 'c+', ':') = ':efa:ef'
55
56  ascii('a') = 97
57
58  find('eeabcabcee', 'abc') = 'abcabcee'
59  find('eeabcabcee', 'abc', 'leading') = 'abcabcee'
60  find('eeabcabcee', 'abc', 'trailing') = 'abcee'

```

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"

```

1 map_get('a', json_decode('{"a": 1}') ) = 1
2 map_get('b', json_decode('{"a": 1}') , 2) = 2
3 map_get('a', map_put('a', 2, json_decode('{"a": 1}')) ) = 2

```

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. 起始位置 2. 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

```

1 nth(2, [1,2,3,4]) = 2
2 length([1,2,3,4]) = 4
3 sublist(3, [1,2,3,4]) = [1,2,3,4]
4 sublist(1,2,[1,2,3,4]) = [1, 2]
5 first([1,2,3,4]) = 1
6 last([1,2,3,4]) = 4
7 contains(2, [1,2,3,4]) = true

```

哈希函数

函数名	函数功能	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

```

1 md5('some val') = '1b68352b3e9c2de52ffd322e30bfffcc4'
2 sha('some val') = 'f85ba28ff5ea84a0cbfa118319acb0c5e58ee2b9'
3 sha256('some val') = '67f97635d8a0e064f60ba6e8846a0ac0be664f18f0c1dc6445cd3542d2b71993'

```

压缩解压缩函数

函数名	函数功能	参数	返回值
gzip	压缩数据, 结果包含 gz 数据头和校验和	原始的二进制数据	压缩后的二进制数据
gunzip	解压缩数据, 原始数据中包含 gz 数据头和校验和	压缩后的二进制数据	原始的二进制数据
zip	压缩数据, 结果不包含 zlib 数据头和校验和	原始的二进制数据	压缩后的二进制数据
unzip	解压缩数据, 原始数据中不包含 zlib 数据头和校验和	压缩后的二进制数据	原始的二进制数据
zip_compress	压缩数据, 结果包含 zlib 数据头和校验和	原始的二进制数据	压缩后的二进制数据
zip_uncompress	解压缩数据, 原始数据中包含 zlib 数据头和校验和	压缩后的二进制数据	原始的二进制数据

```

1 bin2hexstr(gzip('hello world')) = '1F8B08000000000000003CB48CDC9C95728CF2FCA49010085114A0D0B000
2 000'
3 gunzip(hexstr2bin('1F8B08000000000000003CB48CDC9C95728CF2FCA49010085114A0D0B000000')) = 'hell
4 o world'
5
6 bin2hexstr(zip('hello world')) = 'CB48CDC9C95728CF2FCA490100'
7 unzip(hexstr2bin('CB48CDC9C95728CF2FCA490100')) = 'hello world'
8
9 bin2hexstr(zip_compress('hello world')) = '789CCB48CDC9C95728CF2FCA4901001A0B045D'
10 zip_uncompress(hexstr2bin('789CCB48CDC9C95728CF2FCA4901001A0B045D')) = 'hello world'

```

比特操作函数

函数名	函数功能	参数	返回值
subbits	从二进制数据的起始位置获取指定长度的比特位, 然后转换为无符号整型 (大端).	1. 二进制数据 2. 要获取的长度(bits)	无
subbits	从二进制数据的指定下标位置获取指定长度的比特位, 然后转换为无符号整型 (大端). 下标是从 1 开始的	1. 二进制数据 2. 起始位置的下标 3. 要获取的长度(bits)	subbits
subbits	从二进制数据的指定下标位置获取指定长度的比特位, 然后按照给定的参数转换为想要的数据类型. 下标是从 1 开始的.	1. 二进制数据 2. 起始位置的下标 3. 要获取的长度(bits) 4. 数据类型, 可选值: 'integer', 'float', 'bits' 5. 符号类型, 只对整型数据有效, 可选值: 'unsigned', 'signed', 6. 大端还是小端, 只对整型数据有效, 可选值: 'big', 'little'	获取到的数据

```

1 subbits('abc', 8) = 97
2 subbits('abc', 9, 8) = 98
3 subbits('abc', 17, 8) = 99
4 subbits('abc', 9, 16, 'integer', 'signed', 'big') = 25187
5 subbits('abc', 9, 16, 'integer', 'signed', 'little') = 25442

```

编解码函数

函数名	函数功能	参数	返回值
base64_encode	BASE64 编码	要编码的二进制数据	Base64 编码的字符串
base64_decode	BASE64 解码	Base64 编码的字符串	解码后的二进制数据
json_encode	JSON 编码	要转成 JSON 的数据结构	JSON 字符串
json_decode	JSON 解码	要解码的 JSON 字符串	解码后的数据结构
bin2hexstr	二进制数据转为 Hex 字符串	二进制数据	Hex 字符串
hexstr2bin	Hex 字符串转为二进制数据	Hex 字符串	二进制数据

```

1 base64_encode('some val') = 'c29tZSB2YWw='
2 base64_decode('c29tZSB2YWw=') = 'some val'
3 json_encode(json_decode( '{ "a" : 1 }' )) = '{"a":1}'
4 bin2hexstr(hexstr2bin('ABEF123')) = 'ABEF123'

```

Function	Purpose	Parameters	Returned value
<code>schema_encode</code>	通过 Schema 做编码。使用前需要先创建 Schema	1. Schema registry 里定义的 Schema ID 2. 要编码的数据 3..N. 其他的参数，有哪些参数取决于 Schema 的类型	编码后的数据
<code>schema_decode</code>	通过 Schema 做解码。使用前需要先创建 Schema	1. Schema registry 里定义的 Schema ID 2. 要解码的数据 3..N. 其他的参数，有哪些参数取决于 Schema 的类型	解码后的数据

函数 `schema_encode()` 和 `schema_decode()` 的示例请参见 [schema registry](#)

时间与日期函数

Function	Purpose	Parameters	Returned value
<code>now_timestamp</code>	返回当前时间的 Unix 秒级时间戳	-	Unix 时间戳
<code>now_timestamp</code>	指定时间单位，返回当前时间的 Unix 时间戳	1. 时间单位	Unix 时间戳
<code>now_rfc3339</code>	生成当前时间的 RFC3339 字符串，秒级	-	RFC3339 时间字符串
<code>now_rfc3339</code>	指定时间单位，生成当前时间的 RFC3339 字符串	1. 时间单位	RFC3339 时间字符串
<code>unix_ts_to_rfc3339</code>	将秒级 Unix 时间戳转换为 RFC3339 时间字符串	1. Unix 时间戳 (秒)	RFC3339 时间字符串
<code>unix_ts_to_rfc3339</code>	指定时间单位，将 Unix 时间戳转换为 RFC3339 时间字符串	1. Unix 时间戳 2. 时间单位	RFC3339 时间字符串
<code>rfc3339_to_unix_ts</code>	将秒级 RFC3339 时间字符串转换为 Unix 时间戳	1. RFC3339 时间字符串	Unix 时间戳
<code>rfc3339_to_unix_ts</code>	指定时间单位，将 RFC3339 时间字符串转换为 Unix 时间戳	1. RFC3339 时间字符串 2. 时间单位	Unix 时间戳
<code>format_date</code>	时间戳转格式化时间	1. 时间戳精度 (参考时间戳精度定义) 2. 时间偏移量 (参考时间偏移量定义) 3. 日期格式 (参考时间字符串编解码格式) 4. 时间戳 (可选参数，默認為当前时间)	格式化时间字符串
<code>date_to_unix_ts</code>	格式化时间转时间戳	1. 时间戳精度 (参考时间戳精度定义) 2. 时间偏移量 (可选，未填写时，使用格式化时间字符串中的时间偏移量，参考时间偏移量定义) 3. 日期格式 (参考时间字符串编解码格式) 4. 格式化时间字符串	Unix 时间戳

时间戳精度

时间戳精度名称	精度	示例
second	秒	1653557821
millisecond	毫秒	1653557852982
microsecond	微秒	1653557892926417
nanosecond	纳秒	1653557916474793000

时间字符串编解码格式

占位符	含义	取值范围
%Y	年	0000 - 9999
%m	月	01 - 12
%d	日	01 - 31
%H	时	00 - 12
%M	分	00 - 59
%S	秒	01 - 59
%N	纳秒	000000000 - 999999999
%3N	毫秒	000000 - 999999
%6N	微秒	000 - 000
%z	时间偏移量 [+ -]HHMM	-1159 至 +1159
%:z	时间偏移量 [+ -]HH:MM	-11:59 至 +11:59
%::z	时间偏移量 [+ -]HH:MM:SS	-11:59:59 至 +11:59:59

时间偏移量定义

格式	含义	示例
<code>z</code>	UTC Zulu 时间	固定值 <code>+00:00</code>
<code>Z</code>	UTC Zulu 时间, 与 <code>z</code> 相同	固定值 <code>+00:00</code>
<code>local</code>	系统时间	自动获取, 例如 北京时间 <code>+08:00</code> Zulu 时间 <code>+00:00</code> 瑞典斯德哥尔摩时间 <code>+02:00</code> 洛杉矶时间 <code>-08:00</code>
<code>[+\ -]HHMM</code>	<code>%z</code> 格式	北京时间 <code>+0800</code> Zulu 时间 <code>+0000</code> 瑞典斯德哥尔摩时间 <code>+0200</code> 洛杉矶时间 <code>-0800</code>
<code>[+\ -]HH:MM</code>	<code>%:z</code> 格式	北京时间 <code>+08:00</code> Zulu 时间 <code>+00:00</code> 瑞典斯德哥尔摩时间 <code>+02:00</code> 洛杉矶时间 <code>-08:00</code>
<code>[+\ -]HH:MM:SS</code>	<code>%::z</code> 格式	北京时间 <code>+08:00:00</code> Zulu 时间 <code>+00:00:00</code> 瑞典斯德哥尔摩时间 <code>+02:00:00</code> 洛杉矶时间 <code>-08:00:00</code>
<code>integer()</code>	时间偏移量秒数	北京时间 28800 Zulu 时间 0 瑞典斯德哥尔摩时间 7200 洛杉矶时间 -28800

```

1 now_timestamp() = 1650874276
2 now_timestamp('millisecond') = 1650874318331
3 now_rfc3339() = '2022-04-25T16:08:41+08:00'
4 now_rfc3339('millisecond') = '2022-04-25T16:10:10.652+08:00'
5 unix_ts_to_rfc3339(1650874276) = '2022-04-25T16:11:16+08:00'
6 unix_ts_to_rfc3339(1650874318331, 'millisecond') = '2022-04-25T16:11:58.331+08:00'
7 rfc3339_to_unix_ts('2022-04-25T16:11:16+08:00') = 1650874276
8 rfc3339_to_unix_ts('2022-04-25T16:11:58.331+08:00', 'millisecond') = 1650874318331
9 format_date('second', '+0800', '%Y-%m-%d %H:%M:%S%:z', 1653561612) = '2022-05-26 18:40:12+08:00'
10 format_date('second', 'local', '%Y-%m-%d %H:%M:%S%:z') = "2022-05-26 18:48:01+08:00"
11 format_date('second', 0, '%Y-%m-%d %H:%M:%S%:z') = '2022-05-26 10:42:41+00:00'
12 date_to_unix_ts('second', '%Y-%m-%d %H:%M:%S%:z', '2022-05-26 18:40:12+08:00') = 1653561612
13 date_to_unix_ts('second', 'local', '%Y-%m-%d %H-%M-%S', '2022-05-26 18:40:12') = 1653561612
14 date_to_unix_ts('second', '%Y-%m-%d %H-%M-%S', '2022-05-26 10:40:12') = 1653561612

```

Function	Purpose	Parameters	Returned value
<code>mongo_date</code>	生成当前时间的 mongodb ISODate 类型	-	ISODate 类型的时间
<code>mongo_date</code>	生成指定 Unix 时间戳的 mongodb ISODate 类型, 毫秒级	1. 毫秒级 Unix 时间戳	ISODate 类型的时间
<code>mongo_date</code>	指定时间单位, 生成指定 Unix 时间戳的 mongodb ISODate 类型	1. Unix 时间戳 2. 时间单位	ISODate 类型的时间

时间单位可以是以下其中之一:

'second', 'millisecond', 'microsecond' or 'nanosecond'.

```
1 mongo_date() = 'ISODate("2012-12-19T06:01:17.171Z")'
2 mongo_date(timestamp) = 'ISODate("2012-12-19T06:01:17.171Z")'
3 mongo_date(timestamp, 'millisecond') = 'ISODate("2012-12-19T06:01:17.171Z")'
```

创建规则

使用 Dashboard 创建规则

创建 WebHook 规则

- 搭建 Web 服务，这里使用 `nc` 命令做一个简单的Web 服务：

提示

`nc` 命令在部分 Linux 操作系统上有问题，无法与 EMQX 发起的 HTTP 请求连接握手成功，第 7 步可能无法正常进行。

```
1 $ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 8081; done;
```

- 创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写用以处理 `t/#` 主题的规则 SQL：

```
1   SELECT
2     *
3   FROM
4     "t/#"
5   WHERE
6     qos = 1
```

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
5 WHERE
6 qos = 1
```

当前事件可用字段

id	clientid	username	payload	peerhost	topic	qos	flags
headers	timestamp	node					

规则 SQL 示例

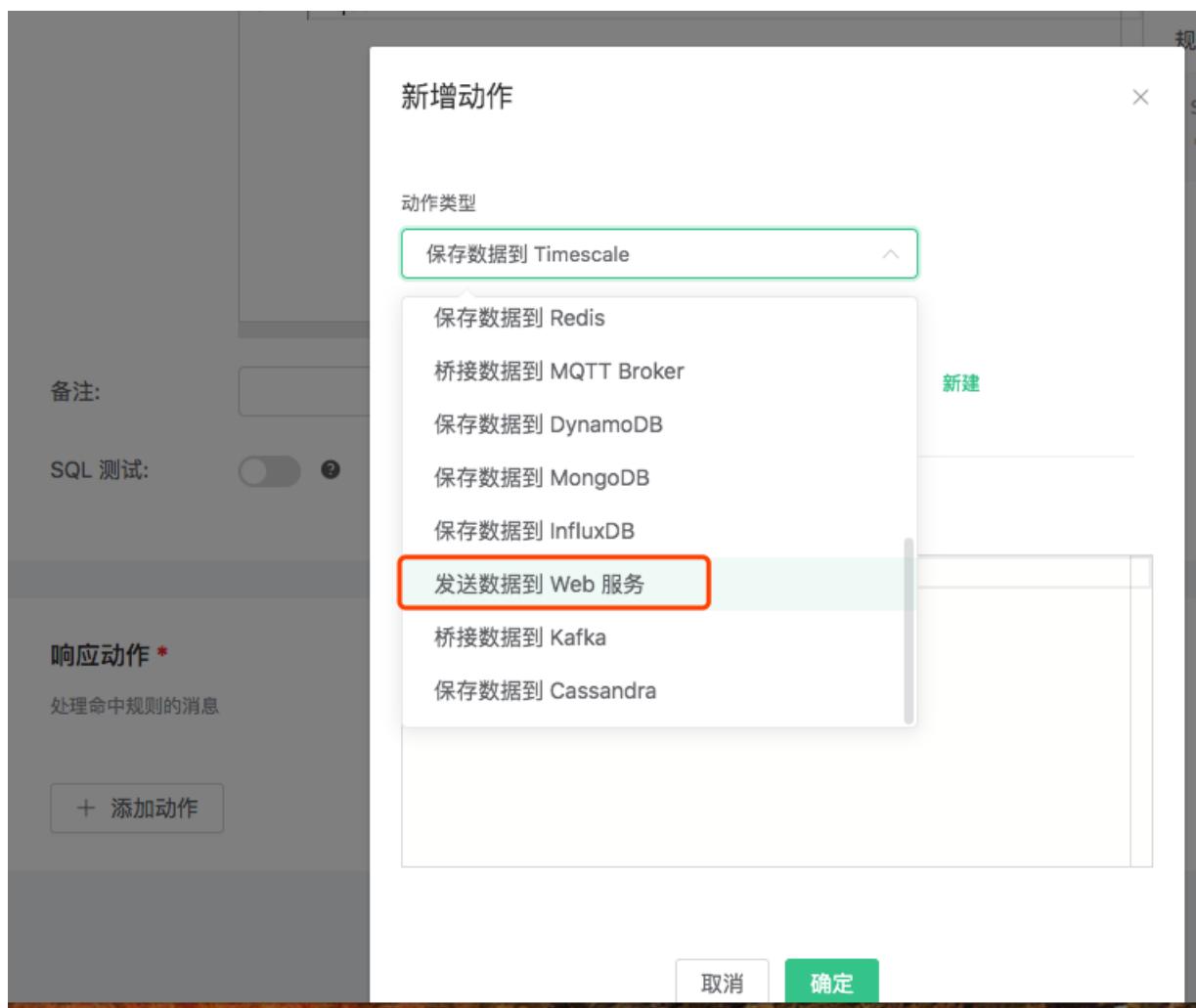
```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

备注:

SQL 测试:

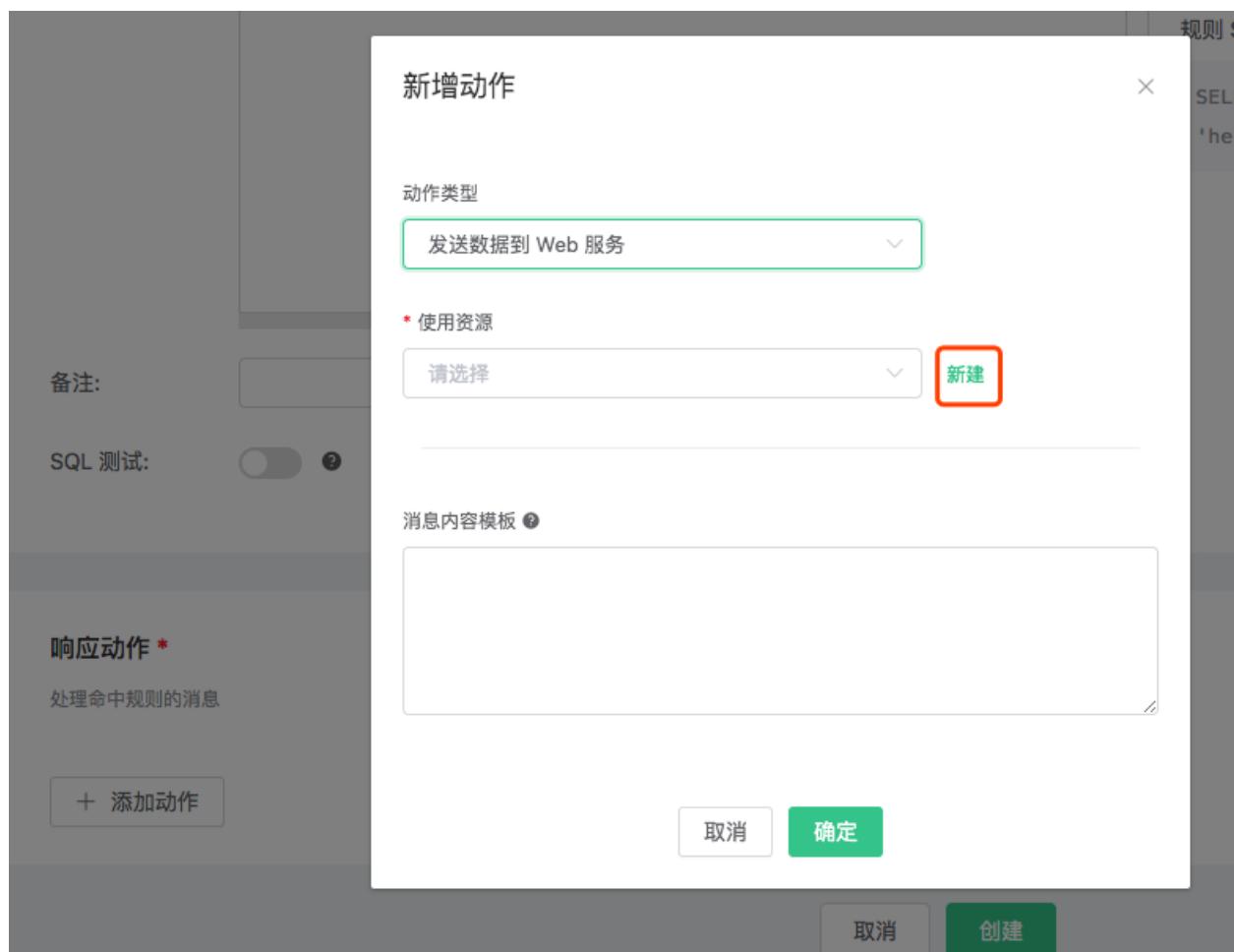
- 关联动作：

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“发送数据到 Web 服务”。

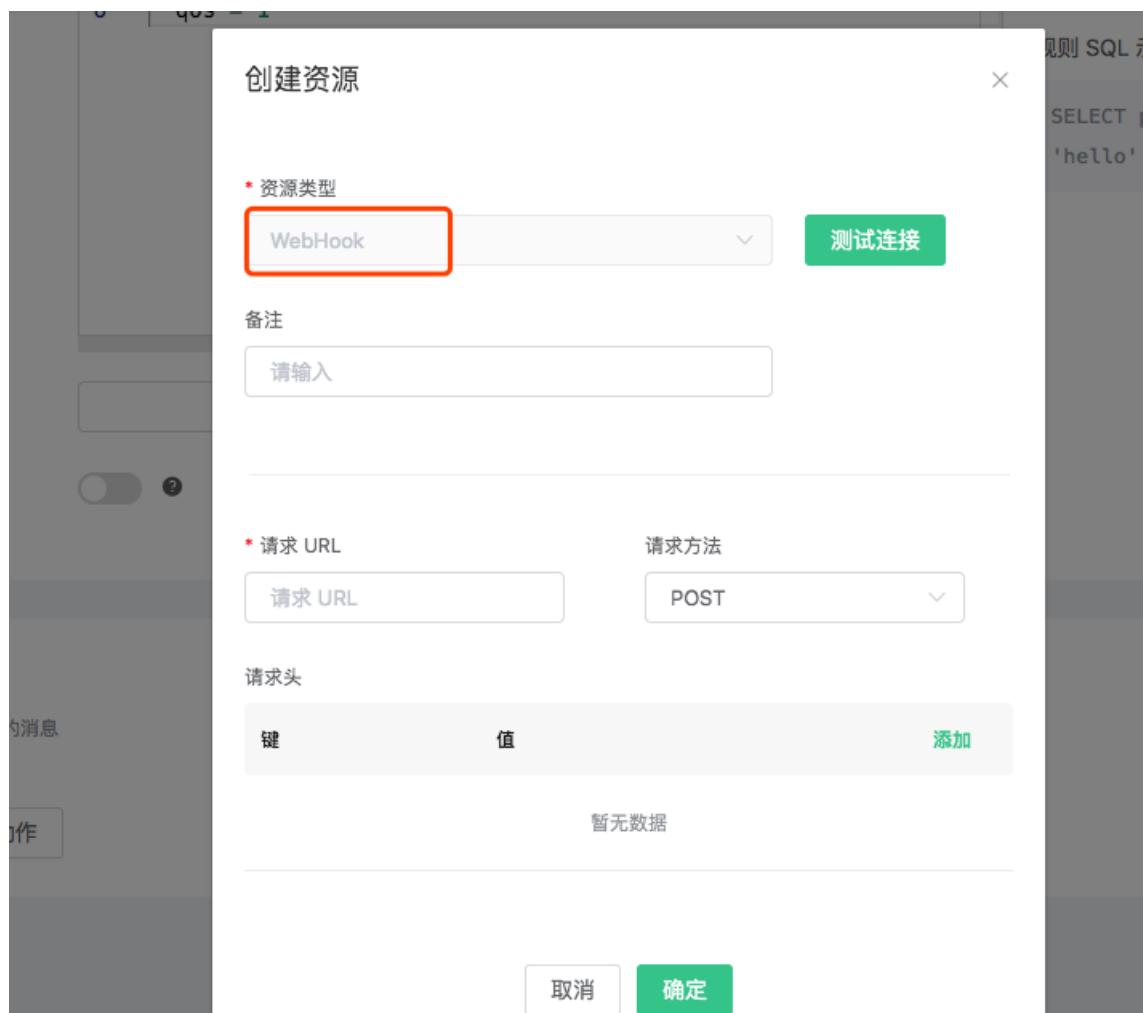


3. 给动作关联资源:

现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **WebHook** 资源：



选择 “WebHook 资源”:

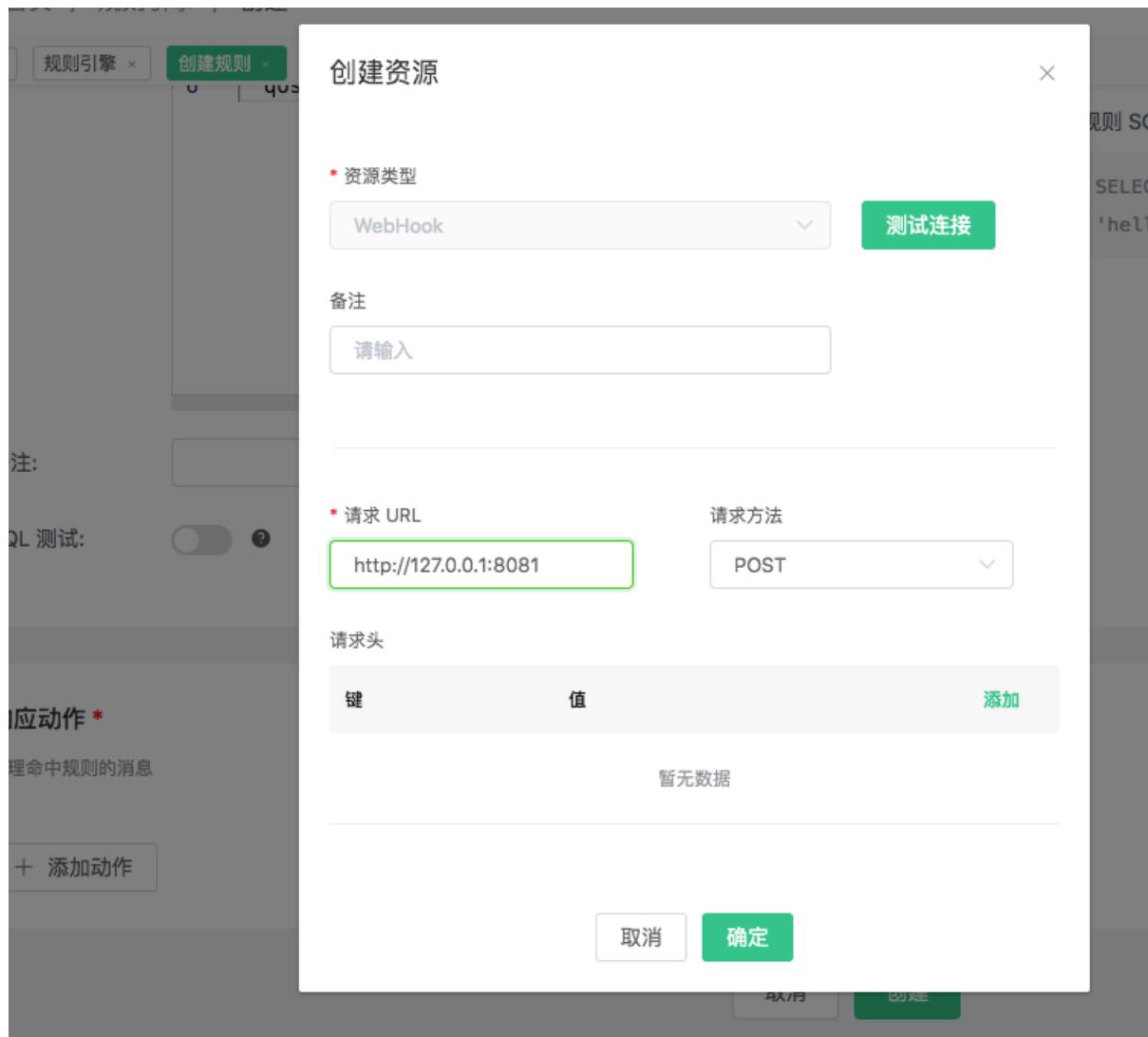


4. 填写资源配置:

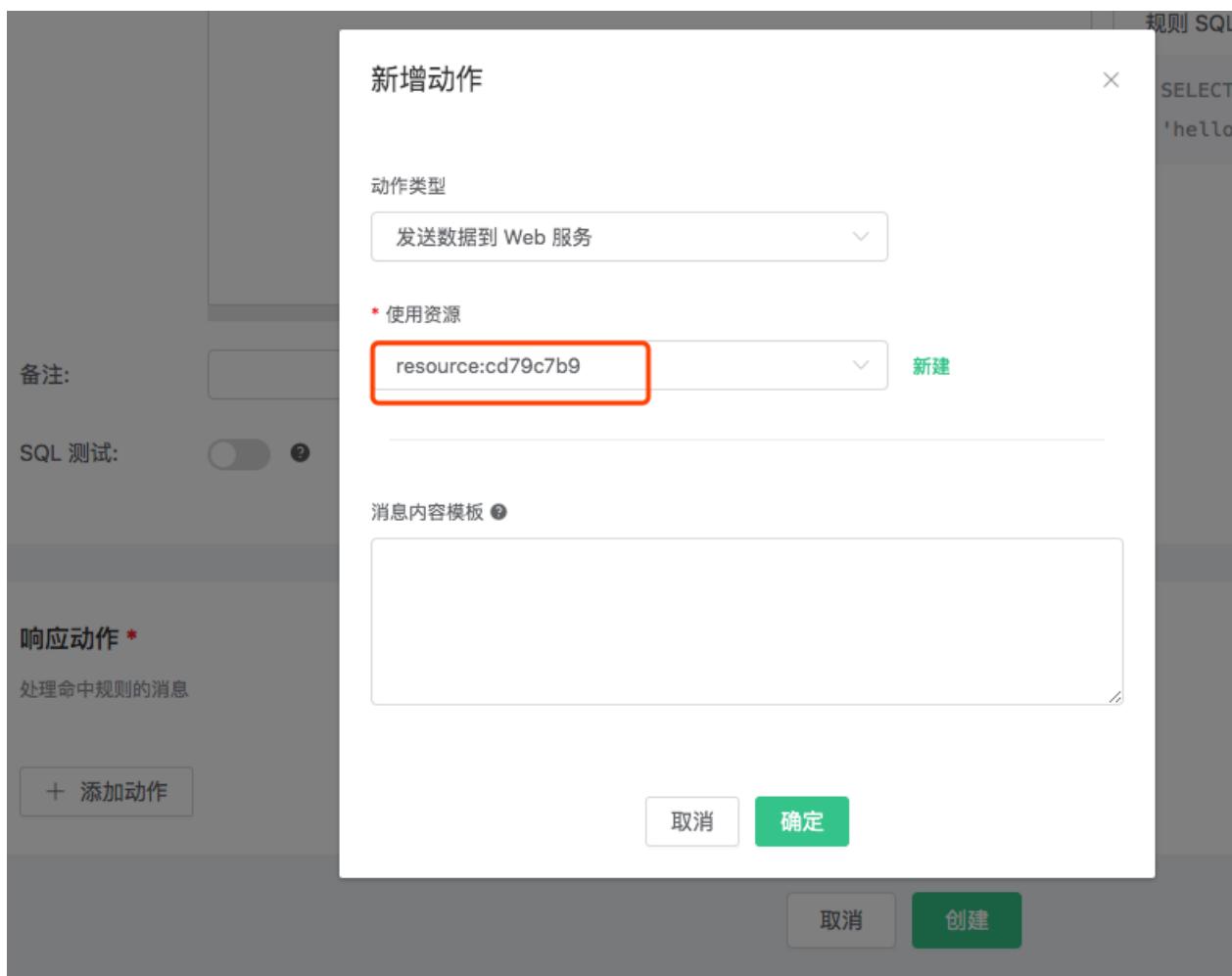
填写“请求 URL”和请求头(可选):

http://127.0.0.1:8081

点击“测试连接”按钮，确保连接测试成功，最后点击“新建”按钮:



5. 返回响应动作界面，点击“确认”。



6. 返回规则创建界面，点击“新建”。

动作类型	发送数据到 Web 服务 (data_to_webserver)	编辑 移除
将数据转发给 Web 服务		
消息内容模板	资源 ID	resource:cd79c7b9

+ 添加动作

取消 创建

规则已经创建完成，规则列表里展示出了新创建的规则：

ID	主题	监控	描述	响应动作
rule:695c1bd8	t/#	null		发送数据到 Web 服务

+ 创建

7. 发一条消息：

Topic: "t/1"

QoS: 1

Payload: "Hello web server"

然后检查 **Web** 服务是否收到消息:

```
{19-06-28 20:53}EMQ:~ emqer%
→ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 9901; done;
POST / HTTP/1.1
content-type: application/json
content-length: 238
te:
host: 127.0.0.1:9901
connection: keep-alive

{"client_id": "clientId-2J41Kt0Exq", "event": "message.publish", "id": "58C61D4F5BDECF443000009B00007", "payload": "Hello web server", "peername": "127.0.0.1:54925", "qos": 1, "retain": 0, "timestamp": 1561726696144, "topic": "t/1", "username": "undefined"}]
```

通过 **CLI** 创建简单规则

创建 **Inspect** 规则

创建一个测试规则，当有消息发送到 '**t/a**' 主题时，打印消息内容以及动作参数细节。

- 规则的筛选 **SQL** 语句为: **SELECT * FROM "t/a";**
- 动作是: "打印动作参数细节"，需要使用内置动作 '**inspect**'。

```
1 $ ./bin/emqx_ctl rules create \
2   "SELECT * FROM \"t/a\""
3   '[{"name": "inspect", "params": {"a": 1}}]' \
4   -d 'Rule for debug'
5
6 Rule rule:803de6db created
```

上面的 **CLI** 命令创建了一个 ID 为 '**Rule rule:803de6db**' 的规则。

参数中前两个为必选参数:

- SQL** 语句: **SELECT * FROM "t/a"**
- 动作列表: **[{"name": "inspect", "params": {"a": 1}}]**。动作列表是用 **JSON Array** 格式表示的。**name** 字段是动作的名字，**params** 字段是动作的参数。注意 **inspect** 动作是不需要绑定资源的。

最后一个可选参数，是规则的描述: '**Rule for debug**'。

接下来当发送 "**hello**" 消息到主题 '**t/a**' 时，上面创建的 "**Rule rule:803de6db**" 规则匹配成功，然后 "**inspect**" 动作被触发，将消息和参数内容打印到 **emqx** 控制台:

```

1 $ tail -f log/erlang.log.1
2
3 (emqx@127.0.0.1)1> [inspect]
4     Selected Data: #{client_id => <<"shawn">>,event => 'message.publish',
5                     flags => #{dup => false,retain => false},
6                     id => <<"5898704A55D6AF443000083D0002">>,
7                     payload => <<"hello">>,
8                     peername => <<"127.0.0.1:61770">>,qos => 1,
9                     timestamp => 1558587875090,topic => <<"t/a">>,
10                    username => undefined}
11     Envs: #{event => 'message.publish',
12                 flags => #{dup => false,retain => false},
13                 from => <<"shawn">>,
14                 headers =>
15                   #{allow_publish => true,
16                     peername => {{127,0,0,1},61770},
17                     username => undefined},
18                   id => <<0,5,137,135,4,165,93,106,244,67,0,0,8,61,0,2>>,
19                   payload => <<"hello">>,qos => 1,
20                   timestamp => {1558,587875,89754},
21                   topic => <<"t/a">>}
22     Action Init Params: #{<<"a">> => 1}

```

- Selected Data 列出的是消息经过 **SQL** 筛选、提取后的字段，由于我们用的是 `select *`，所以这里会列出所有可用的字段。
- Envs 是动作内部可以使用的环境变量。
- Action Init Params 是初始化动作的时候，我们传递给动作的参数。

创建 WebHook 规则

创建一个规则，将所有发送自 `client_id='Steven'` 的消息，转发到地址为 '<http://127.0.0.1:9910>' 的 **Web** 服务器：

- 规则的筛选条件为: **SELECT username as u, payload FROM "t/a" where u='Steven';**
- 动作是: "转发到地址为 '<http://127.0.0.1:9910>' 的 **Web** 服务";
- 资源类型是: **web_hook**;
- 资源是: "到 `url='http://127.0.0.1:9910'` 的 **WebHook** 资源"。

0. 首先我们创建一个简易 **Web** 服务，这可以使用 `nc` 命令实现：

```

1 $ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 9910; done;

```

1. 使用 **WebHook** 类型创建一个资源，并配置资源参数 `url`:

1). 列出当前所有可用的资源类型，确保 '**web_hook**' 资源类型已存在：

```

1 $ ./bin/emqx_ctl resource-types list
2
3 resource_type(name='web_hook', provider='emqx_web_hook', params=#{...}, on_create={emqx_w
4 eb_hook_actions, on_resource_create}, description='WebHook Resource')
...

```

2). 使用类型 '**web_hook**' 创建一个新的资源，并配置 "`url`"="<http://127.0.0.1:9910>"：

```

1 $ ./bin/emqx_ctl resources create \
2   'web_hook' \
3   -c '{"url": "http://127.0.0.1:9910", "headers": {"token": "axfw34y235wrq234t4ersgw4t"}, "met-
4   hod": "POST"}'
5
Resource resource:691c29ba created

```

上面的 **CLI** 命令创建了一个 **ID** 为 '[resource:691c29ba](#)' 的资源，第一个参数是必选参数 - 资源类型(**web_hook**)。参数表明此资源指向 **URL** = "<http://127.0.0.1:9910>" 的 **Web** 服务，方法为 **POST**，并且设置了一个 **HTTP Header** **"token"**。

2. 然后创建规则，并选择规则的动作作为 '**data_to_webserver**'：

1). 列出当前所有可用的动作，确保 '**data_to_webserver**' 动作已存在：

```

1 $ ./bin/emqx_ctl rule-actions list
2
3 action(name='data_to_webserver', app='emqx_web_hook', for='$any', types=[web_hook], params
4 =#{'$resource' => ...}, title ='Data to Web Server', description='Forward Messages to Web Ser-
ver')
5 ...

```

2). 创建规则，选择 **data_to_webserver** 动作，并通过 "**\$resource**" 参数将 [resource:691c29ba](#) 资源绑定到动作上：

```

1 $ ./bin/emqx_ctl rules create \
2   "SELECT username as u, payload FROM \"message.publish\" where u='Steven'" \
3   '[{"name":"data_to_webserver", "params": {"$resource": "resource:691c29ba"}}]' \
4   -d "Forward publish msgs from steven to webserver"
5
6 rule:26d84768

```

上面的 **CLI** 命令与第一个例子里创建 **Inspect** 规则时类似，区别在于这里需要把刚才创建的资源 '[resource:691c29ba](#)' 绑定到 '**data_to_webserver**' 动作上。这个绑定通过给动作设置一个特殊的参数 '**\$resource**' 完成。**data_to_webserver** 动作的作用是将数据发送到指定的 **Web** 服务器。

3. 现在我们使用 **username "Steven"** 发送 **"hello"** 到任意主题，上面创建的规则就会被触发，**Web Server** 收到消息并回复 **200 OK**：

```

1 $ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 9910; done;
2
3 POST / HTTP/1.1
4 content-type: application/json
5 content-length: 32
6 te:
7 host: 127.0.0.1:9910
8 connection: keep-alive
9 token: axfw34y235wrq234t4ersgw4t
10
11 {"payload":"hello","u":"Steven"}

```


如何更新资源

通过管理界面

首先打开规则引擎的【资源】选项卡，选择你要更新的资源，然后点击【编辑】按钮：

此时会弹出编辑窗口：

输入你要修改的参数以后，点击【确认】按钮：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left, there's a sidebar with various navigation options like Monitoring, Client, Topics, Subscriptions, Rule Engine, Rules, and Resources. The 'Resources' option is currently selected. In the main area, there's a table titled '资源' (Resources) with columns for ID, Description, Resource Type, and Operations. One row is selected, showing ID: resource:863369, Description: testq, Resource Type: WebHook. A modal window titled 'Notice' is open over the table, containing a message '确认修改配置信息?' (Confirm modification configuration information?) and two buttons: '取消' (Cancel) and '确认' (Confirm). The '确认' button is highlighted with a red box.

此时会提示是否确认修改，点击【确认】按钮即可，下图是修改后的效果：

This screenshot shows the same interface after the update. The 'Resource Management' table now displays the updated 'Description' value 'testq' for the selected row (highlighted with a red box). A green success message '编辑成功' (Edit successful) is shown above the table. The rest of the interface remains the same, with the sidebar and other components visible.

此时“描述”信息已经被更新。

通过命令行

通过命令行更新，需要事先知道资源的ID：



然后使用下列命令更新：

```
1 emqx_ctl resources update $ID -d $Desc -c $Config
```

其中 `update` 后面的第一个参数为资源ID，`-d` 参数为“描述” `-c` 参数为具体的资源参数的 **JSON字符串** 格式：

例如下面的这个**JSON**：

```
1 {
2   "verify":false,
3   "url":"http://www.demo.com",
4   "request_timeout":5,
5   "pool_size":32,
6   "keyfile":"",
7   "connect_timeout":5,
8   "certfile":"",
9   "cacertfile":""
10 }
```

json

字符串格式是：

```
1 "{\"verify\":false,\"url\":\"http://www.demo.com\",\"request_timeout\":5,\"pool_size\":32,\"keyfile\":\"\",\"connect_timeout\":5,\"certfile\":\"\",\"cacertfile\":\"\"}"
```

完整命令：

```
1 emqx_ctl resources update resource:001 -d "hello" -c "{\"verify\":false,\"url\":\"http://www.demo.com\",\"request_timeout\":5,\"pool_size\":32,\"keyfile\":\"\",\"connect_timeout\":5,\"certfile\":\"\",\"cacertfile\":\"\"}"
```

至此为止，我们完成了通过管理界面和命令行两种更新资源的方式。

提示

EMQX Broker 中仅适用以下操作：

- 检查 (调试)
- 发送数据到 **Web** 服务
- 桥接数据到 **MQTT Broker**
- 保存数据到 **TDengine**(使用 发送数据到 **Web** 服务 实现) 其余均是 **EMQX Enterprise** 专属功能。

检查 (调试)

创建一个测试规则，当有消息发送到 '**t/a**' 主题时，打印消息内容以及动作参数细节。

- 规则的筛选 **SQL** 语句为: **SELECT * FROM "t/a";**
- 动作是: "打印动作参数细节"，需要使用内置动作 '**inspect**'。

```

1 $ ./bin/emqx_ctl rules create \
2   "SELECT * FROM \"t/a\" WHERE \" \
3     '[{"name":"inspect", "params": {"a": 1}}]' \
4     -d 'Rule for debug'
5
6 Rule rule:803de6db created

```

sh

上面的 **CLI** 命令创建了一个 **ID** 为 '**Rule rule:803de6db**' 的规则。

参数中前两个为必参数：

- **SQL** 语句: **SELECT * FROM "t/a"**
- 动作列表: **[{"name":"inspect", "params": {"a": 1}}]**。动作列表是用 **JSON Array** 格式表示的。**name** 字段是动作的名字，**params** 字段是动作的参数。注意 **inspect** 动作是不需要绑定资源的。

最后一个可选参数，是规则的描述: '**Rule for debug**'。

接下来当发送 "**hello**" 消息到主题 '**t/a**' 时，上面创建的 "**Rule rule:803de6db**" 规则匹配成功，然后 "**inspect**" 动作被触发，将消息和参数内容打印到 **emqx** 控制台：

```

1 $ tail -f log/erlang.log.1
2
3 (emqx@127.0.0.1)1> [inspect]
4     Selected Data: #{client_id => <<"shawn">>,event => 'message.publish',
5                         flags => #{dup => false},
6                         id => <<"5898704A55D6AF443000083D0002">>,
7                         payload => <<"hello">>,
8                         peername => <<"127.0.0.1:61770">>,qos => 1,
9                         timestamp => 1558587875090,topic => <<"t/a">>,
10                        username => undefined}
11     Env: #{event => 'message.publish',
12                  flags => #{dup => false},
13                  from => <<"shawn">>,
14                  headers =>
15                      #{allow_publish => true,
16                         peername => {{127,0,0,1},61770},
17                         username => undefined},
18                         id => <<0,5,137,135,4,165,93,106,244,67,0,0,8,61,0,2>>,
19                         payload => <<"hello">>,qos => 1,
20                         timestamp => {1558,587875,89754},
21                         topic => <<"t/a">>}
22     Action Init Params: #{<<"a">> => 1}

```

- `Selected Data` 列出的是消息经过 **SQL** 筛选、提取后的字段，由于我们用的是 `select *`，所以这里会列出所有可用的字段。
- `Env` 是动作内部可以使用的环境变量。
- `Action Init Params` 是初始化动作的时候，我们传递给动作的参数。

发送数据到 Web 服务

创建一个规则，将所有发送自 `client_id='Steven'` 的消息，转发到地址为 '<http://127.0.0.1:9910>' 的 **Web** 服务器：

- 规则的筛选条件为: **SELECT username as u, payload FROM "#" where u='Steven';**
- 动作是: "转发到地址为 '<http://127.0.0.1:9910>' 的 **Web** 服务";
- 资源类型是: **web_hook**;
- 资源是: "到 `url='http://127.0.0.1:9910'` 的 **WebHook** 资源"。

首先我们创建一个简易 **Web** 服务，这可以使用 `nc` 命令实现：

```

1 $ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 9910; done;

```

使用 **WebHook** 类型创建一个资源，并配置资源参数 `url`:

1). 列出当前所有可用的资源类型，确保 '**web_hook**' 资源类型已存在：

```

1 $ ./bin/emqx_ctl resource-types list
2     resource_type{name='web_hook', provider='emqx_web_hook', params=#{...}}, on_create={emqx_web
3     _hook_actions,on_resource_create}, description='WebHook Resource'
4     ...

```

2). 使用类型 '**web_hook**' 创建一个新的资源，并配置 "`url="http://127.0.0.1:9910"`"：

```

1 $ ./bin/emqx_ctl resources create \
2   'web_hook' \
3   -c '{"url": "http://127.0.0.1:9910", "headers": {"token": "axfw34y235wrq234t4ersgw4t"}, "method": "POST"}'
4
5 Resource resource:691c29ba create

```

上面的 **CLI** 命令创建了一个 ID 为 '[resource:691c29ba](#)' 的资源，第一个参数是必选参数 - 资源类型(**web_hook**)。参数表明此资源指向 **URL** = "<http://127.0.0.1:9910>" 的 **Web** 服务，方法为 **POST**，并且设置了一个 **HTTP Header** **"token"**。

然后创建规则，并选择规则的动作作为 '**data_to_webserver**'：

- 1). 列出当前所有可用的动作，确保 '**data_to_webserver**' 动作已存在：

```

1 $ ./bin/emqx_ctl rule-actions list
2
3 action(name='data_to_webserver', app='emqx_web_hook', for='any', types=[web_hook], params=#{}
4   '$resource' => ...}, title ='Data to Web Server', description='Forward Messages to Web Server'
5
6 ...

```

- 2). 创建规则，选择 **data_to_webserver** 动作，并通过 "**\$resource**" 参数将 [resource:691c29ba](#) 资源绑定到动作上：

```

1 $ ./bin/emqx_ctl rules create \
2   "SELECT username as u, payload FROM \"#\\" where u='Steven'" \
3   '[{"name": "data_to_webserver", "params": {"$resource": "resource:691c29ba"}}]' \
4   -d "Forward publish msgs from steven to webserver"
5
6 rule:26d84768

```

上面的 **CLI** 命令与第一个例子里创建 **Inspect** 规则时类似，区别在于这里需要把刚才创建的资源 '[resource:691c29ba](#)' 绑定到 '**data_to_webserver**' 动作上。这个绑定通过给动作设置一个特殊的参数 '**\$resource**' 完成。
'data_to_webserver' 动作的作用是将数据发送到指定的 **Web** 服务器。

现在我们使用 **username "Steven"** 发送 **"hello"** 到任意主题，上面创建的规则就会被触发，**Web Server** 收到消息并回复 **200 OK**：

```

1 $ while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l 127.0.0.1 9910; done;
2
3 POST / HTTP/1.1
4 content-type: application/json
5 content-length: 32
6 te:
7 host: 127.0.0.1:9910
8 connection: keep-alive
9 token: axfw34y235wrq234t4ersgw4t
10
11 {"payload": "hello", "u": "Steven"}

```


保存数据到 MySQL

搭建 MySQL 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```
1 $ brew install mysql  
2  
3 $ brew services start mysql  
4  
5 $ mysql -u root -h localhost -p  
6  
7 ALTER USER 'root'@'localhost' IDENTIFIED BY 'public';
```

初始化 MySQL 表：

```
1 $ mysql -u root -h localhost -ppublic
```

创建 “**test**” 数据库：

```
1 CREATE DATABASE test;
```

创建 **t_mqtt_msg** 表：

```
1 USE test;  
2 CREATE TABLE `t_mqtt_msg` (  
3   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
4   `msgid` varchar(64) DEFAULT NULL,  
5   `topic` varchar(255) NOT NULL,  
6   `qos` tinyint(1) NOT NULL DEFAULT '0',  
7   `payload` blob,  
8   `arrived` datetime NOT NULL,  
9   PRIMARY KEY (`id`),  
10  INDEX topic_index(`id`, `topic`)  
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

```

mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.00 sec)

mysql>
mysql> USE test;
Database changed
mysql> DROP TABLE IF EXISTS `t_mqtt_msg`;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE TABLE `t_mqtt_msg` (
    ->   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    ->   `msgid` varchar(64) DEFAULT NULL,
    ->   `topic` varchar(255) NOT NULL,
    ->   `qos` tinyint(1) NOT NULL DEFAULT '0',
    ->   `payload` blob,
    ->   `arrived` datetime NOT NULL,
    ->   PRIMARY KEY (`id`),
    ->   INDEX topic_index(`id`, `topic`)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
Query OK, 0 rows affected (0.07 sec)

mysql> describe t_mqtt_msg;
+-----+-----+-----+-----+-----+-----+
| Field | Type           | Null | Key | Default | Extra        |
+-----+-----+-----+-----+-----+-----+
| id   | int(11) unsigned | NO  | PRI | NULL    | auto_increment |
| msgid | varchar(64)      | YES |     | NULL    |               |
| topic | varchar(255)     | NO  |     | NULL    |               |
| qos  | tinyint(1)       | NO  |     | 0       |               |
| payload | blob            | YES |     | NULL    |               |
| arrived | datetime        | NO  |     | NULL    |               |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.02 sec)

```

创建规则:

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL:

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

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at	timestamp	node			

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

备注:

SQL 测试:

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 MySQL”。

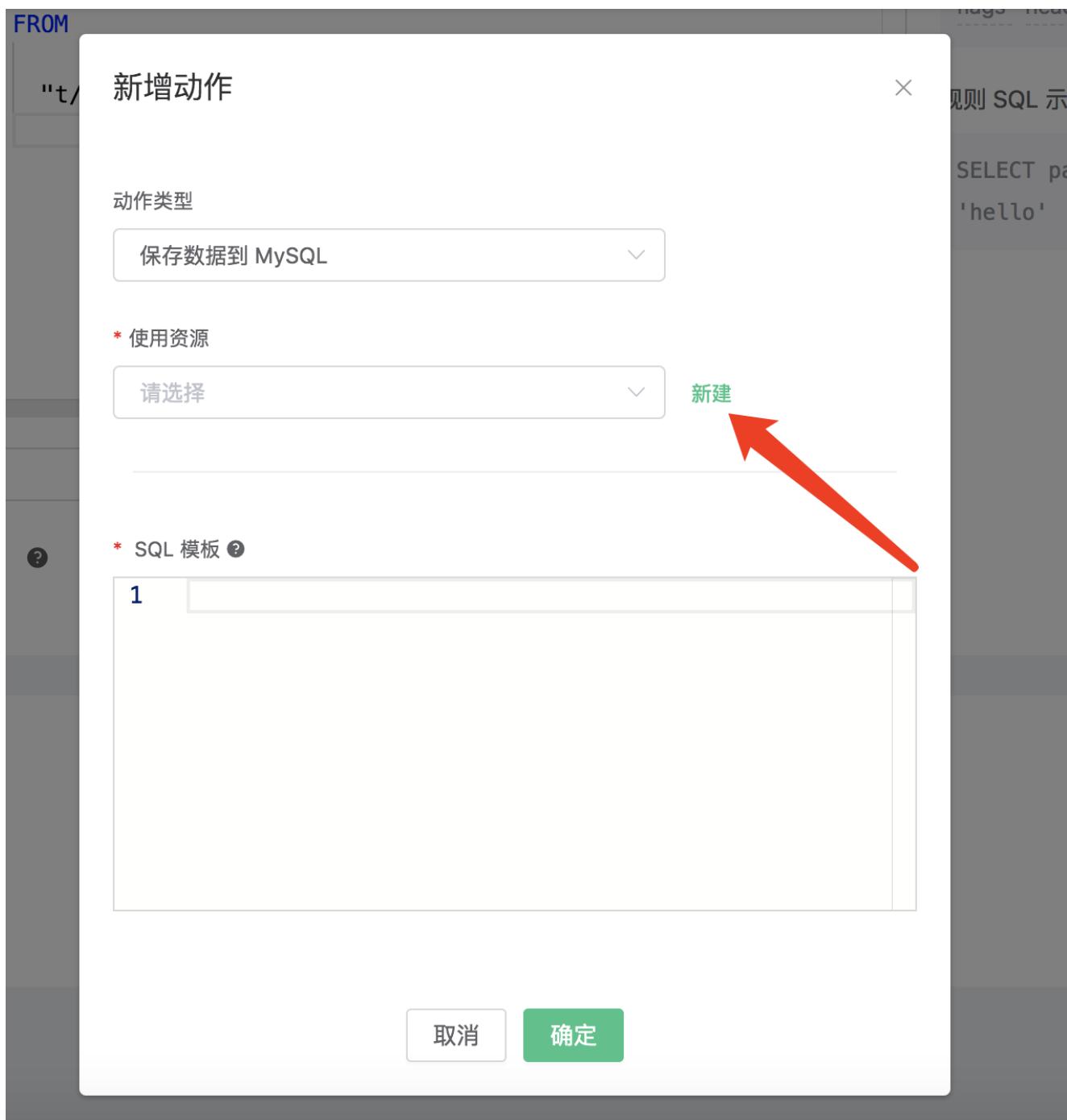


填写动作参数:

“保存数据到 MySQL” 动作需要两个参数:

1). SQL 模板。这个例子里我们向 MySQL 插入一条数据, SQL 模板为:

```
1 insert into t_mqtt_msgmsgid, topic, qos, payload, arrived) values (${id}, ${topic}, ${qos},  
${payload}, FROM_UNIXTIME(${timestamp}/1000))
```



2). 关联资源的 ID。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 MySQL 资源：

填写资源配置：

数据库名填写“mqtt”，用户名填写“root”，密码填写“123456”

创建

创建资源

* 资源类型

MySQL

测试连接

* 资源名称

MySQL

* MySQL 服务器

127.0.0.1:3306

* MySQL 数据库名

mqtt

连接池大小

8

* MySQL 用户名

root

MySQL 密码

123456

批量写入大小

100

批量写入间隔(毫秒)

10

是否重连

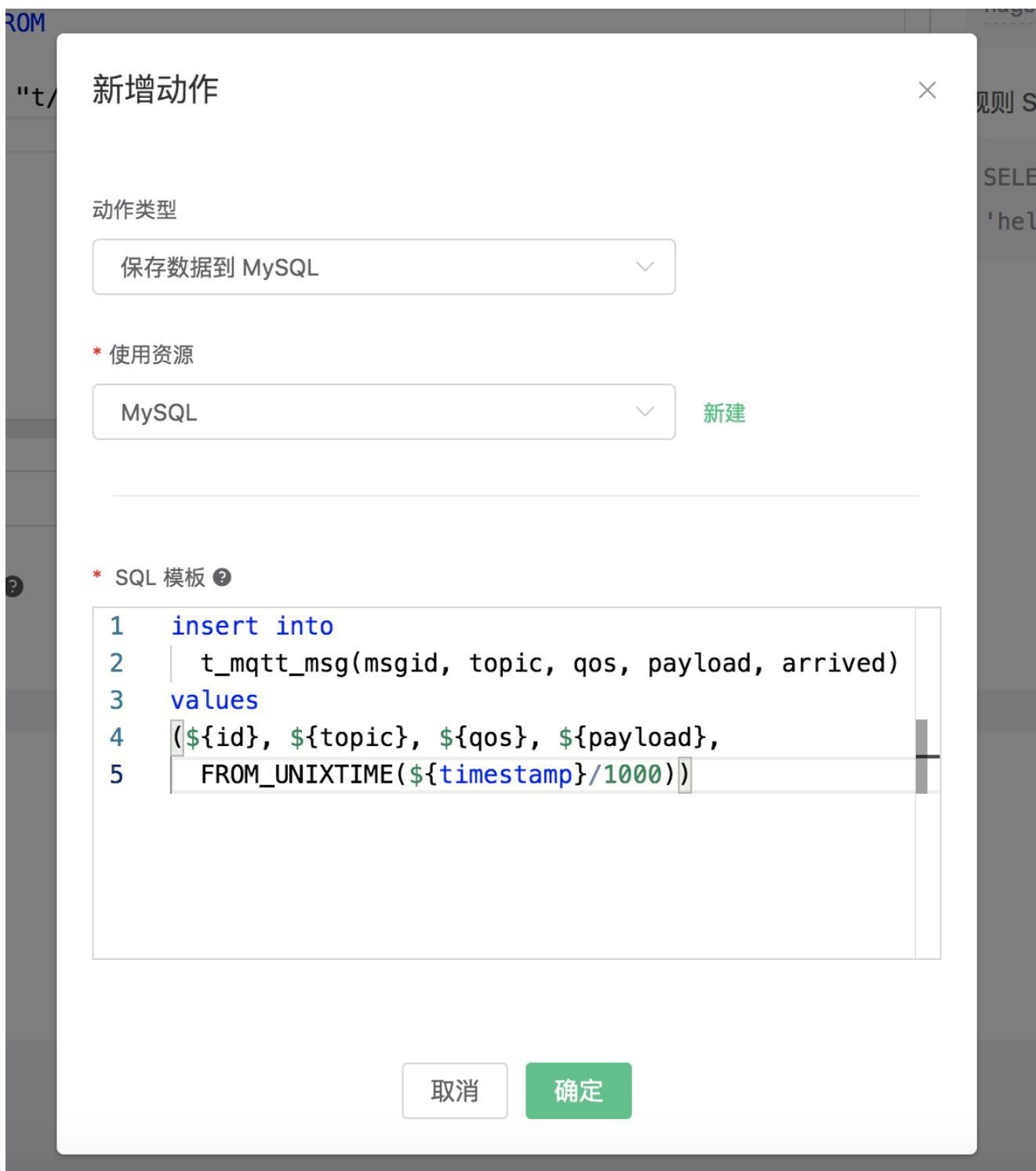
true

取消 确定

This screenshot shows the 'Create Resource' dialog for a MySQL connection. It includes fields for resource type (MySQL), name (MySQL), server (127.0.0.1:3306), database (mqtt), user (root), password (123456), batch size (100), batch interval (10ms), and whether to reconnect (true). A green 'Test Connection' button is visible at the top right.

点击“新建”按钮。

返回响应动作界面，点击“确认”。



返回规则创建界面，点击“创建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 MySQL (data_to_mysql)	编辑 移除
保存数据到 MySQL 数据库	
SQL 模板	
<pre>insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values (\${id}, \${topic}, \${qos}, \${payload}, FROM_UNIXTIME(\${timestamp}/1000))</pre>	
资源 ID resource:8148a94b	+ 失败备选动作
+ 添加动作	
取消 创建	

在规则列表里，点击“查看”按钮或规则 ID 连接，可以预览刚才创建的规则：

查询字段： *

筛选条件：

规则 SQL：

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

响应动作

命中规则的消息处理方式

动作类型 保存数据到 MySQL (data_to_mysql) 保存数据到 MySQL 数据库 详细统计 点击查看 SQL 模板 <code>insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values (\${id}, \${topic}, \${qos}, \${payload}, FROM_UNIXTIME(\${timestamp}/1000))</code> 资源 ID resource:8148a94b	成功 0 失败 0
---	-----------

规则已经创建完成，现在发一条数据：

```
1 Topic: "t/a"
2 QoS: 1
3 Payload: "hello"
```

然后检查 MySQL 表，新的 record 是否添加成功：

```
mysql> select * from t_mqtt_msg;
Empty set (0.00 sec)

mysql>
mysql> select * from t_mqtt_msg;
+----+-----+-----+-----+-----+
| id | msgid           | topic | qos | payload | arrived          |
+----+-----+-----+-----+-----+
| 1  | 589886299C168F442000008E50002 | t/a   | 1   | hello    | 2019-05-23 14:42:26 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> []
```

保存数据到 PostgreSQL

提示

支持 **PostgreSQL 13** 及以下版本

搭建 PostgreSQL 数据库，以 **MacOS X** 为例：

```
1 $ brew install postgresql
2 $ brew services start postgresql
```

```
1 ## 使用用户名 root 创建名为 'mqtt' 的数据库
2 $ createdb -U root mqtt
3
4 $ psql -U root mqtt
5
6 mqtt=> \dn;
7   List of schemas
8   Name    | Owner
9   ----+-----
10  public | shawn
11  (1 row)
```

初始化 PgSQL 表：

```
1 $ psql -U root mqtt
```

创建 `t_mqtt_msg` 表：

```
1 CREATE TABLE t_mqtt_msg (
2     id SERIAL primary key,
3     msgid character varying(64),
4     sender character varying(64),
5     topic character varying(255),
6     qos integer,
7     payload text,
8     arrived timestamp without time zone
9 );
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

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

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at	timestamp	node			

规则 SQL 示例

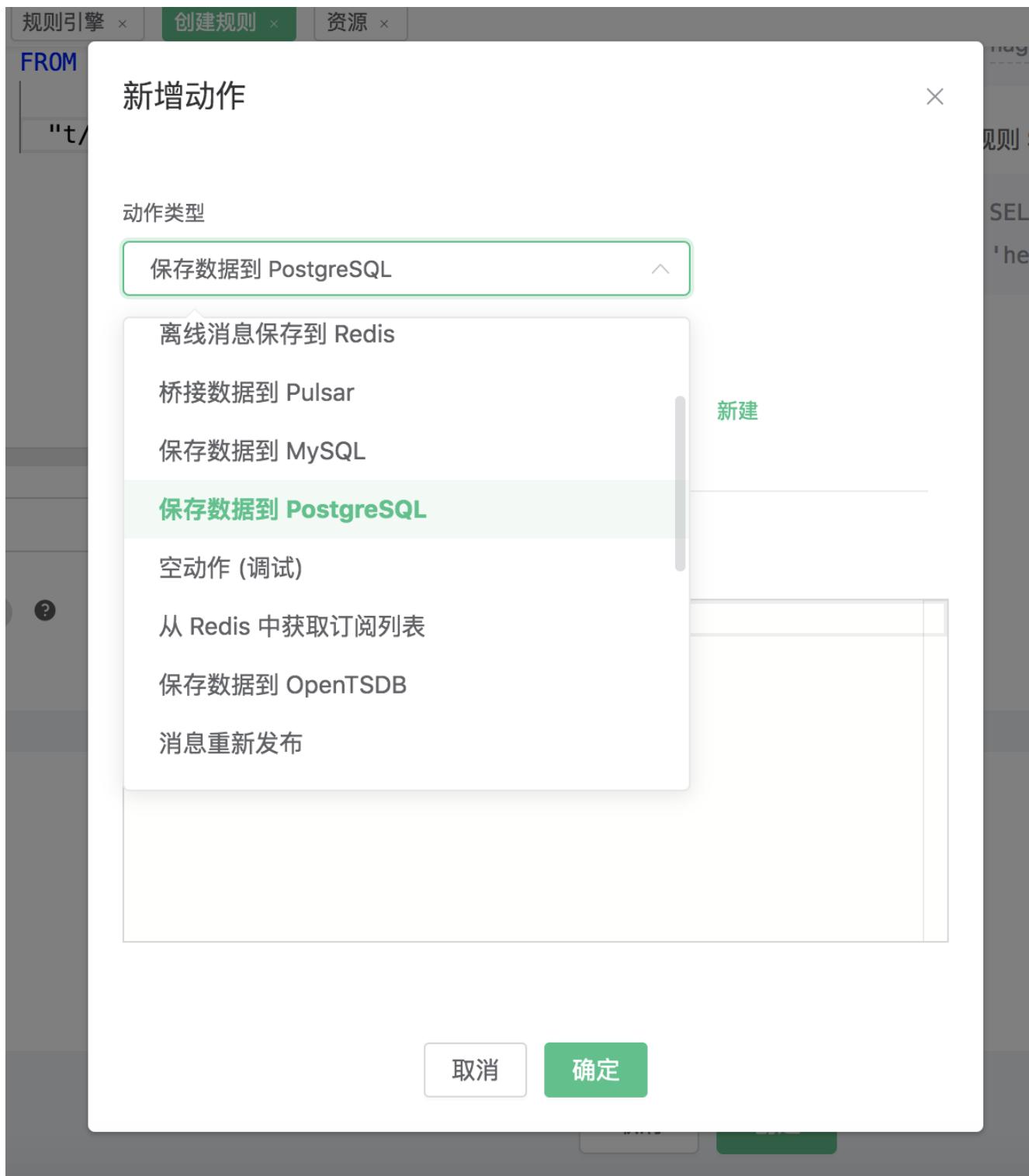
```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

备注:

SQL 测试:

关联动作:

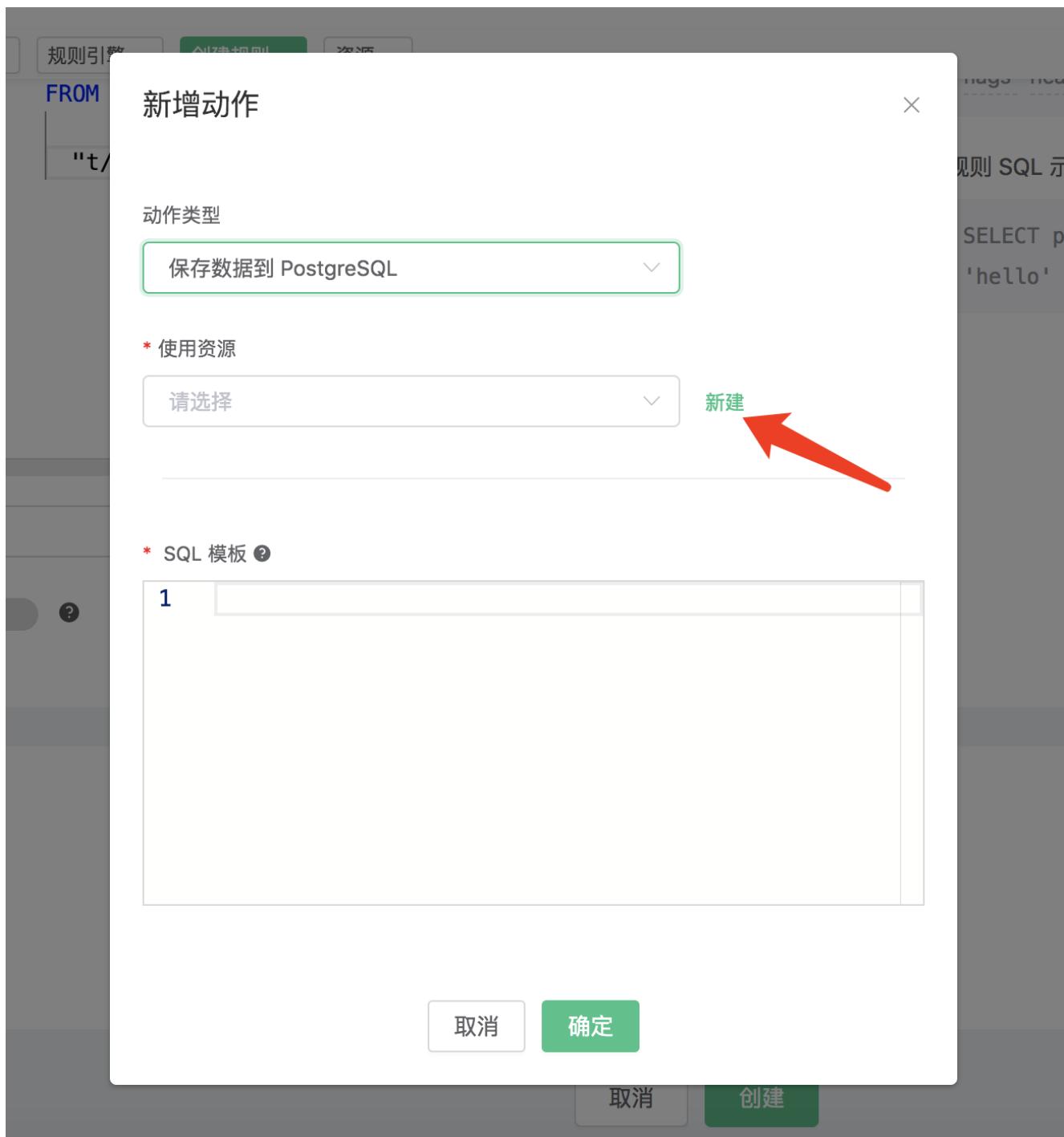
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **PostgreSQL**”。

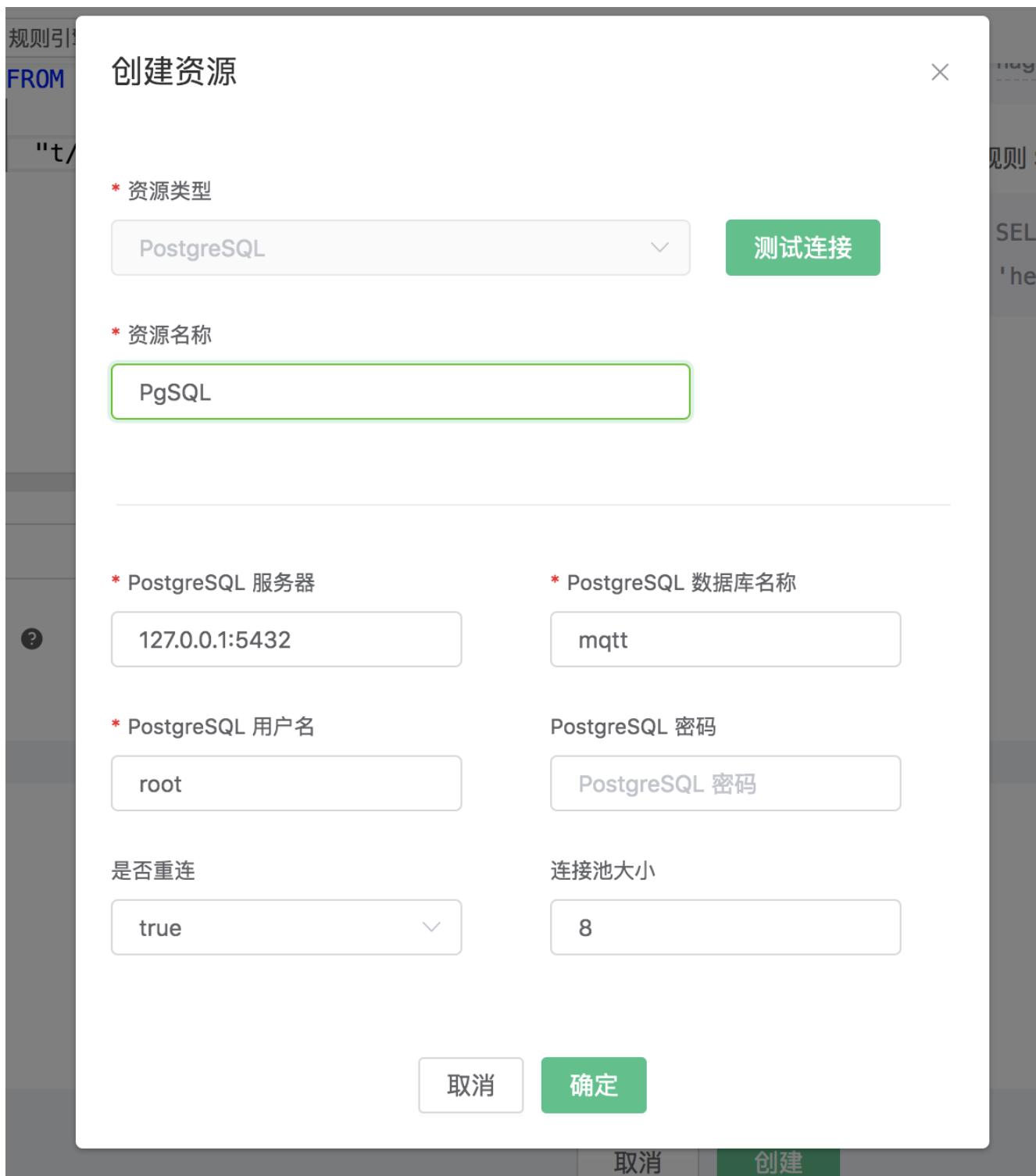


填写动作参数:

“保存数据到 PostgreSQL” 动作需要两个参数:

- 1). 关联资源的 ID。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 PostgreSQL 资源:





返回响应动作界面，点击“确认”。

2).SQL 模板。这个例子里我们向 **PostgreSQL** 插入一条数据，**SQL** 模板为：

```
1 insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values (${id}, ${topic}, ${qos}, ${payload}, to_timestamp(${timestamp}::double precision /1000))
```

插入数据之前，**SQL** 模板里的 **`\${key}`** 占位符会被替换为相应的值。



返回规则创建界面，点击“创建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 PostgreSQL (data_to_pgsql)
保存数据到 PostgreSQL

SQL 模板

```
insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values (${id}, ${topic}, ${qos}, ${payload}, to_timestamp(${timestamp}::double precision /1000)) returning id
```

资源 ID resource:c6401486

编辑 移除
+ 失败备选动作

+ 添加动作

取消 创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2 QoS: 0
3 Payload: "hello1"

```

然后检查 PostgreSQL 表，新的 record 是否添加成功：

id	msgid	sender	topic	qos	retain	payload	arrived
3	58C2292328ECBF442000008BC0001		t/1	0	f	hello1	2019-06-25 17:29:53.117

(1 row)

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

+ 创建

ID	主题	监控	描述	状态	响应动作
rule:82398433	t/#	full		<input checked="" type="checkbox"/>	保存数据到 PostgreSQL 编辑 移除

保存数据到 MongoDB

为方便演示部分功能，后续内容我们将基于 **MongoDB Cloud** 展开，不过以其他方式部署 **MongoDB** 的用户依然通过本文档了解学习规则引擎 **MongoDB** 资源和动作的使用。

我们首先在 **MongoDB Cloud** 上以副本集方式部署一个名为 **Cluster0** 的集群实例：

The screenshot shows the MongoDB Atlas interface for a database deployment named 'Cluster0'. The left sidebar includes sections for Deployment, Databases (selected), Data Services, and Security. The main area displays metrics for the cluster, such as R: 0, W: 0 operations per second, 6.0 connections, and 57.4 B/s throughput. It also shows the logical size of 36.1 KB and a 'FREE' tier status. A green 'Upgrade' button is visible on the right.

在本示例中，该集群实例包含了一个主节点和两个从节点：

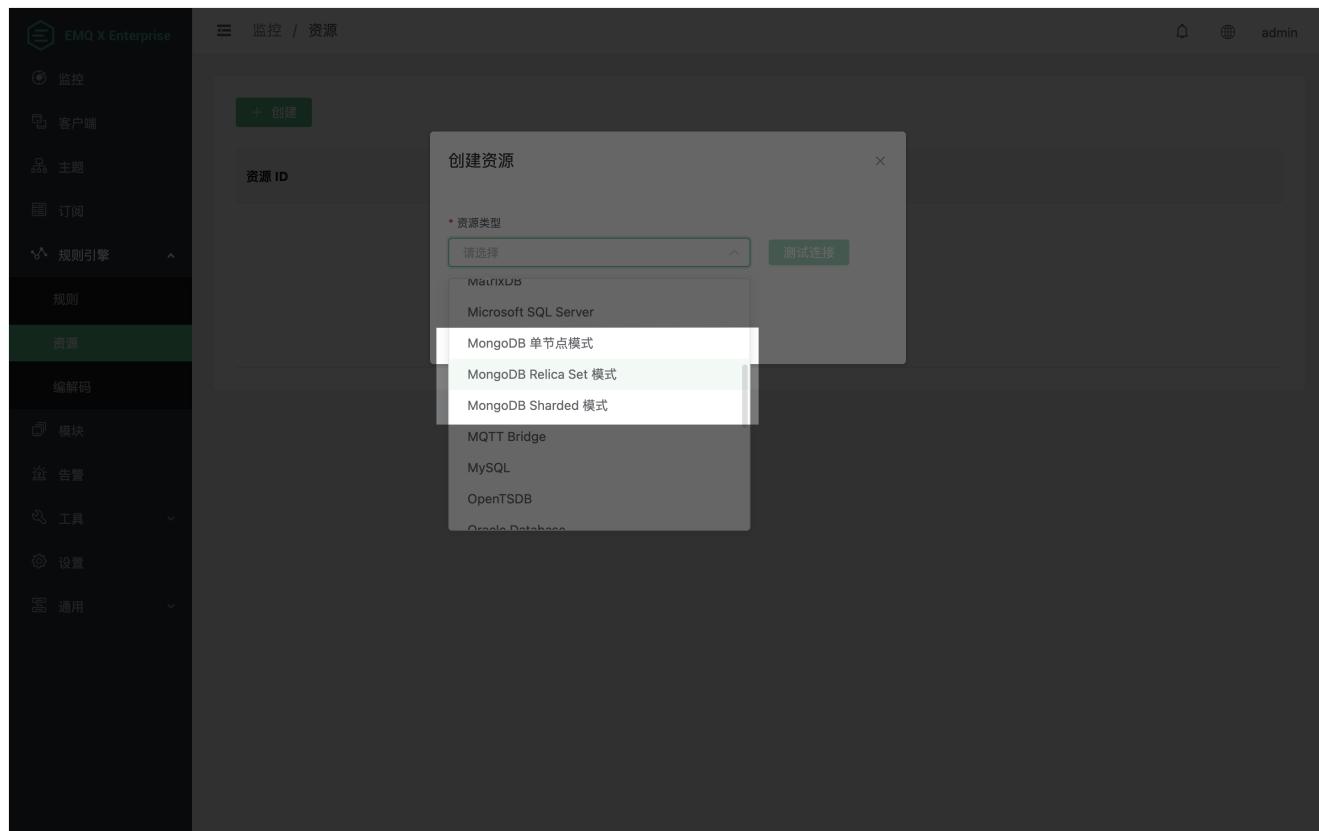
The screenshot shows the MongoDB Atlas interface for the 'Cluster0' database. The left sidebar includes sections for Deployment, Databases (selected), Data Services, and Security. The main area displays the overview of the database, including its version (4.4.10), region (AWS N. Virginia), and cluster tier (M0 Sandbox (General)). It shows three shards: 'cluster0-shard-00-00.6ur...' (SECONDARY), 'cluster0-shard-00-01.6urw...' (SECONDARY), and 'cluster0-shard-00-02.6ur...' (PRIMARY). A note indicates this is a Shared Tier Cluster. Metrics on the right show operations at 100.0/s, connections at 3, and logical size at 36.1 KB.

在进入下一步之前，我们还需要在 **Database Access** 和 **Network Access** 页面配置用户密码和 IP 访问白名单以确保能成功访问。

创建资源

完成以上工作后，接下来我们将在 **EMQX Dashboard** 中完成 **MongoDB** 资源和规则的创建。

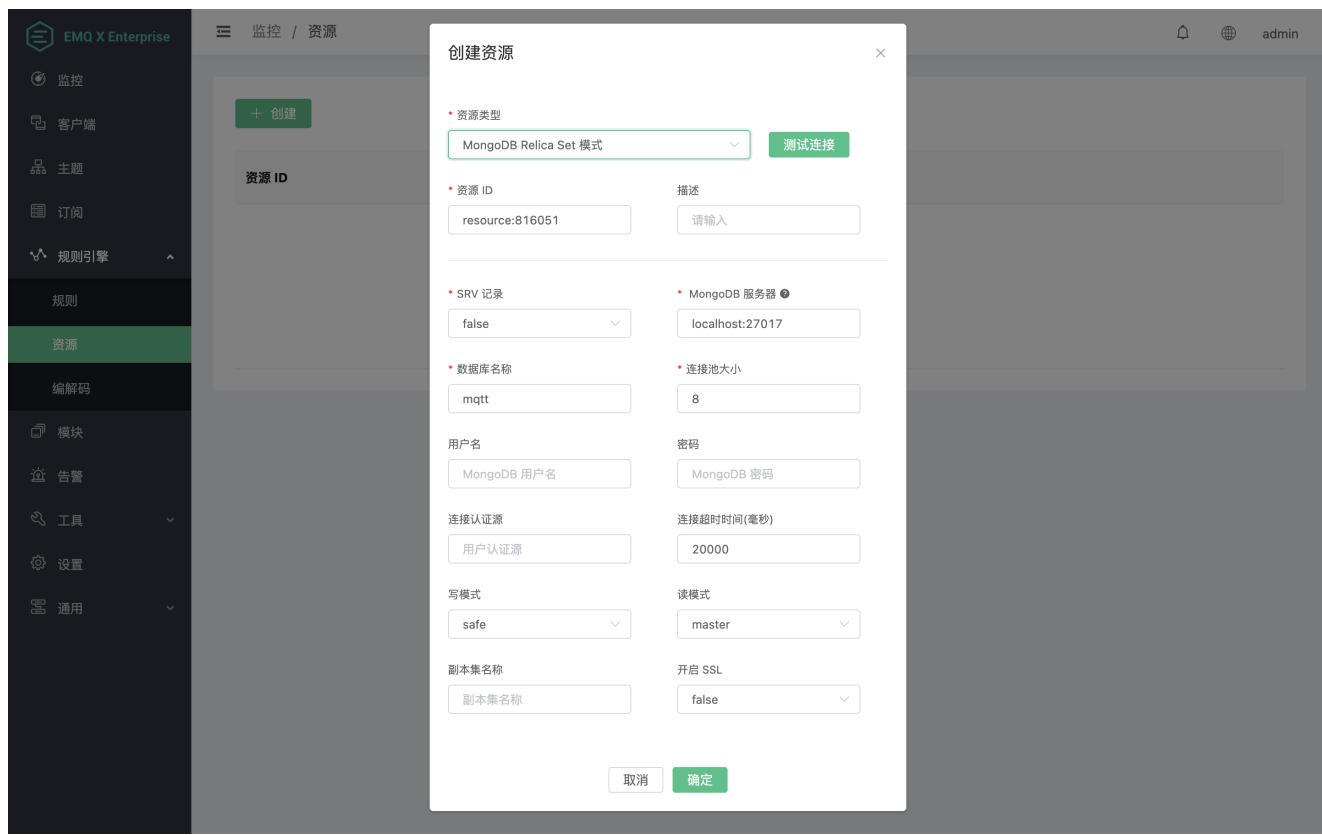
首先打开 **EMQX Dashboard**，进入规则引擎的资源页面，点击左上角的创建按钮，将弹出 创建资源 表单，在表单中的 资源类型 下拉框中我们可以看到 **MongoDB** 单节点模式、**MongoDB Replica Set** 模式和 **MongoDB Sharded** 模式三个资源类型，分别对应 **MongoDB** 的三种部署方式。



这里我们选择 **MongoDB Replica Set** 模式，然后根据 **MongoDB Server** 的实际情况完成相关参数的配置即可。

以下为 **MongoDB** 资源的部分参数说明：

- **SRV** 记录，是否查询 **SRV** 和 **TXT** 记录以获取服务器列表和 **authSource**、**replicaSet** 选项。
- **MongoDB** 服务器，指定服务器列表或添加了 **DNS SRV** 和 **TXT** 记录的域名。
- 数据库名称，**MongoDB** 数据库名称。
- 连接池大小，配置连接进程池大小，合理配置连接池大小以获取最佳性能。
- 用户名、密码，身份验证凭据。
- 连接认证源，指定用于授权的数据库，默认为 **admin**。如果启用了 **SRV** 记录，且您的 **MongoDB** 服务器域名添加了包含 **authSource** 选项的 **DNS TXT** 记录，将优先使用该记录中的 **authSource** 选项。
- 写模式，可设置为 **unsafe** 或 **safe**，设置为 **safe** 时会等待 **MongoDB Server** 的响应并返回给调用者。未指定时将使用默认值 **safe**。
- 读模式，可设置为 **master** 或 **slave_ok**，设置为 **master** 时表示每次查询都将从主节点读取最新数据。未指定时将使用默认值 **master**。
- 副本集名称，如果您的 **MongoDB** 以副本集方式部署，则需要指定相应的副本集名称。但如果 **SRV** 记录设置为 **true**，且您的 **MongoDB** 服务器域名添加了包含 **replicaSet** 选项的 **DNS TXT** 记录，那么可以忽略此配置项。
- 开启 **SSL**，是否启用 **TLS** 连接，设置为 **true** 时会出现更多 **TLS** 相关配置，请按需配置。注意：连接至 **MongoDB Cloud** 时必须开启 **SSL**。

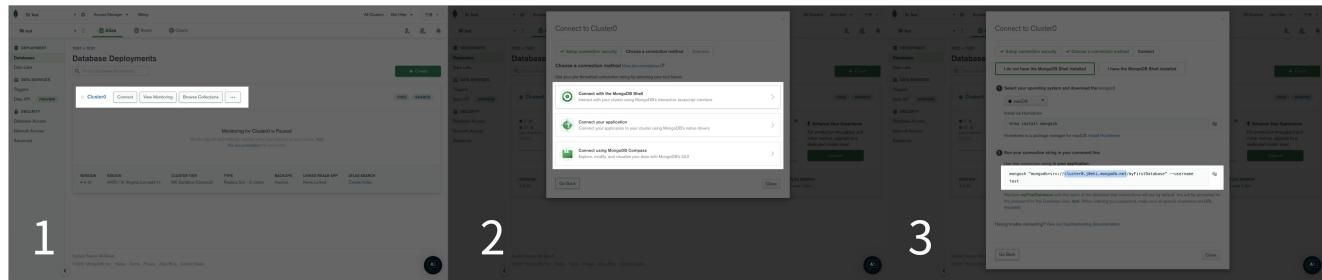


根据是否启用 **SRV Record**, 我们可以使用以下两种方式配置 MongoDB 资源:

启用 SRV Record

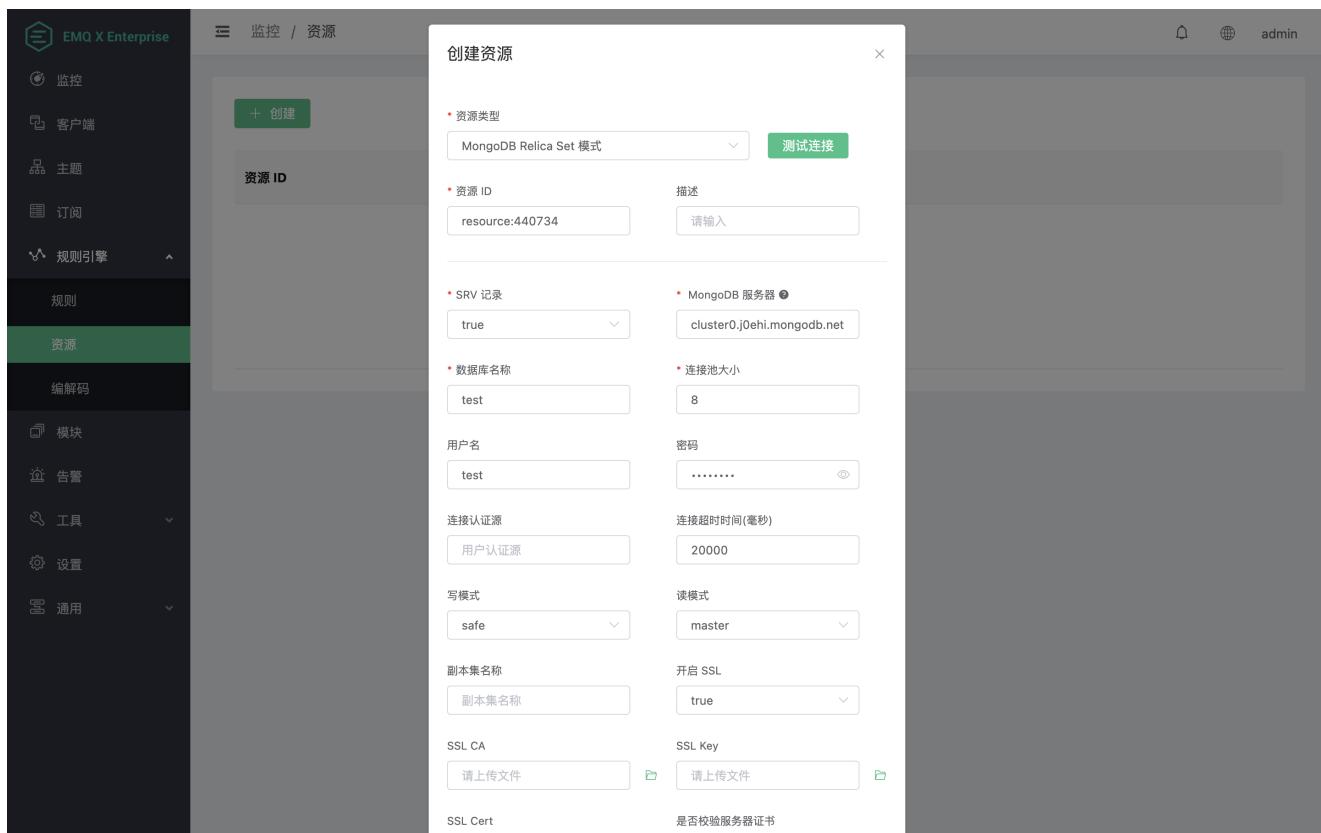
MongoDB Cloud 默认提供了一个已经添加了 **SRV** 和 **TXT** 记录的域名以供连接。

我们在 **MongoDB Cloud** 的 **Databases** 页面点击 **Cluster0** 实例的 **Connect** 按钮, 三个连接方式任选其一, 然后就可以看到当前实例需要使用的连接字符串, 其中光标选中的部分就是我们稍后需要配置到 **EMQX** 规则引擎 **MongoDB** 资源的 **MongoDB** 服务器 字段的内容。

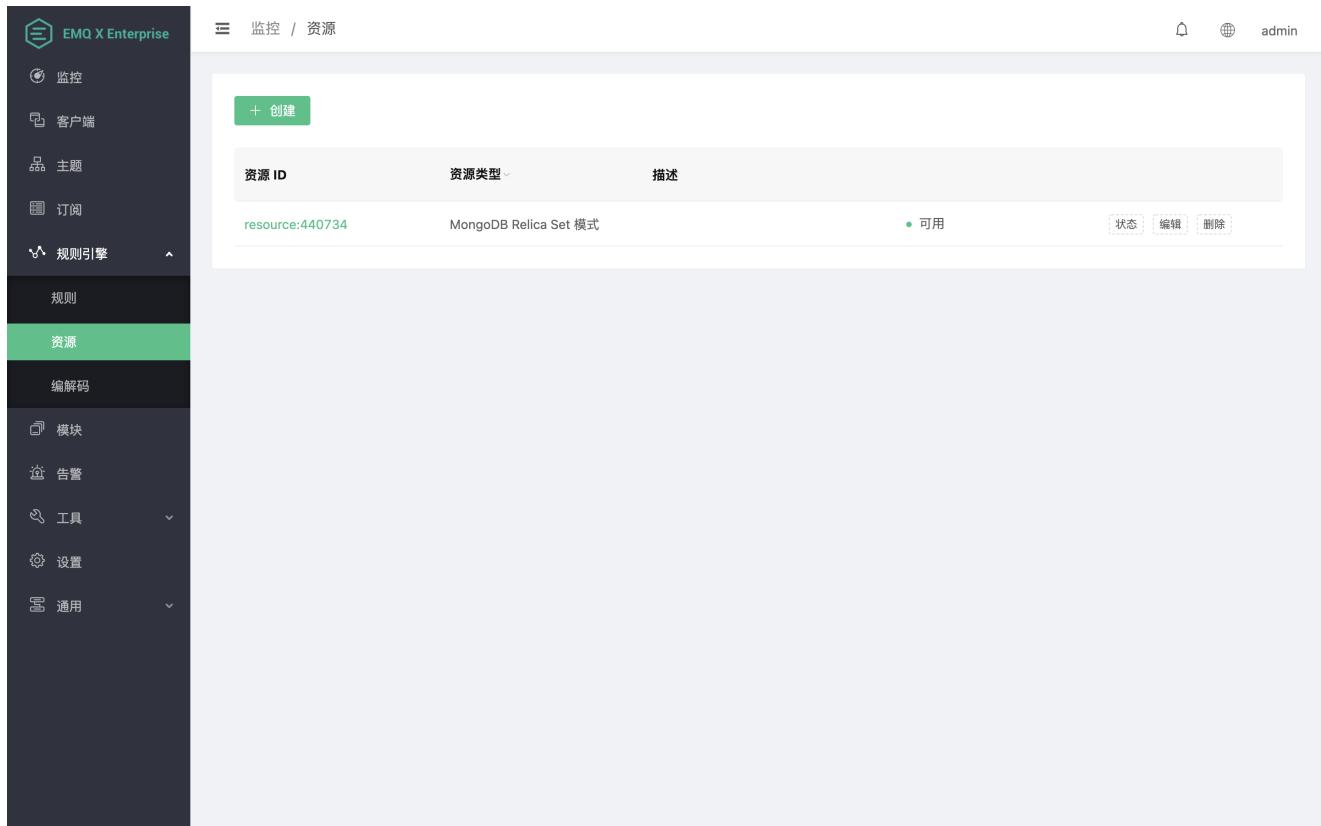


现在, 我们继续完成 **MongoDB** 资源的配置, 这里我主要进行了以下修改:

1. 将 **SRV** 记录 设置为 **true**, 然后将 **MongoDB** 服务器 设置为我们刚刚获取的域名。
2. 将 数据库名称 设置为 **test**, 这是 **MongoDB Cloud** 的默认数据库, 你可以按需配置。
3. 配置 用户名 和 密码, 你需要按实际情况配置。
4. 连接认证源 和 副本集名称 保持为空, **EMQX** 会自动查询 **DNS TXT** 记录。
5. 将 开启 SSL 设置为 **true**, 这是 **MongoDB Cloud** 的连接要求, 其他方式部署时请按需配置。



最后，我们点击 创建资源 表单最下方的 确定 按钮以完成创建，此时一个新的 **MongoDB** 资源实例就在 **EMQX** 中创建成功了：



不启用 SRV Record

如果我们选择不启用 **SRV Record**，那么在副本集和分片模式下我们就需要将 **MongoDB** 集群的所有节点地址都填写到 **MongoDB 服务器** 选项中，并且在副本集模式下还必须指定副本集名称。

为了快速获取这些配置信息，我们可以使用 `nslookup` 命令来查询 **DNS** 记录：

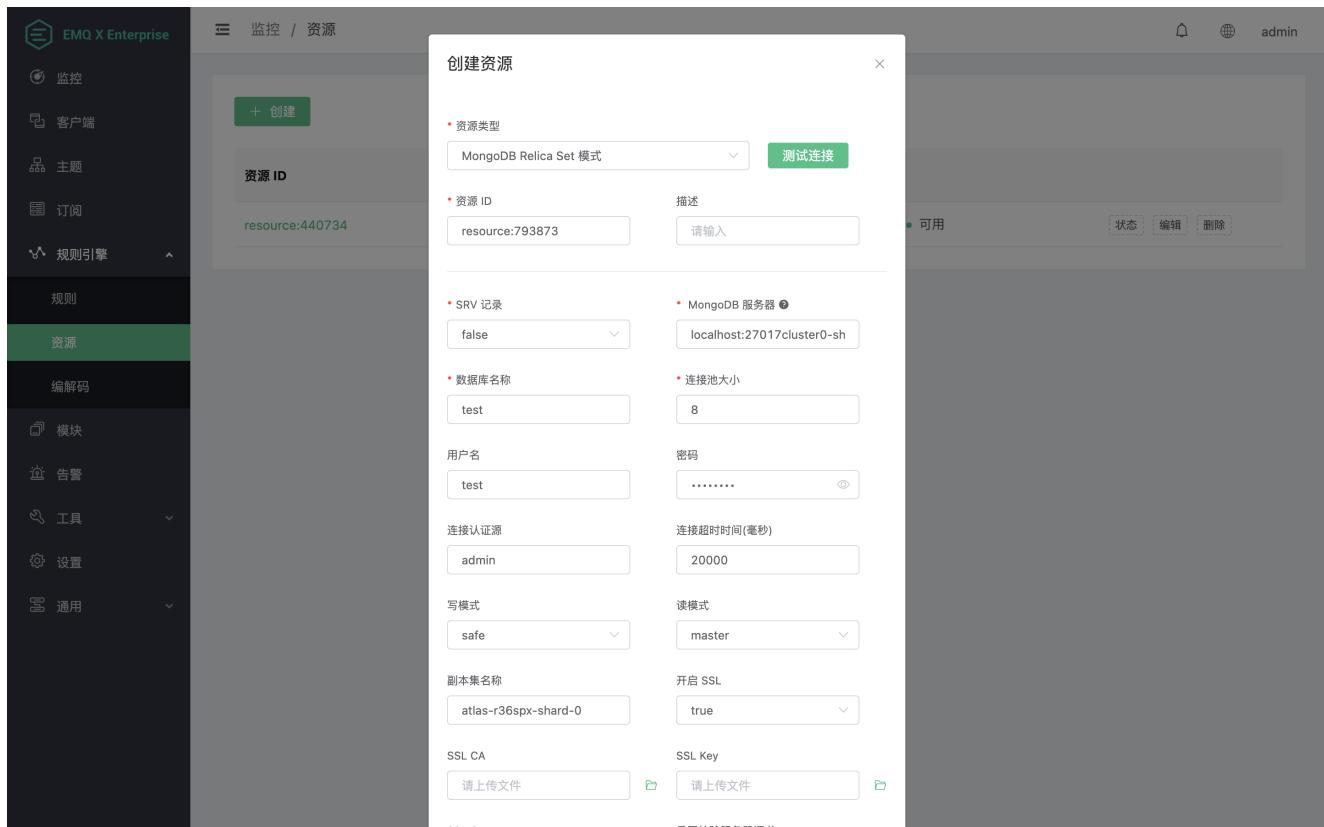
```

1 $ nslookup
2 > set type=SRV
3 > _mongodb._tcp.cluster0.j0ehi.mongodb.net
4 Server:      26.26.26.53
5 Address:     26.26.26.53#53
6
7 Non-authoritative answer:
8 _mongodb._tcp.cluster0.j0ehi.mongodb.net      service = 0 0 27017 cluster0-shard-00-01.j0ehi.
9 ongodb.net.
10 _mongodb._tcp.cluster0.j0ehi.mongodb.net      service = 0 0 27017 cluster0-shard-00-02.j0ehi.
11 ongodb.net.
12 _mongodb._tcp.cluster0.j0ehi.mongodb.net      service = 0 0 27017 cluster0-shard-00-00.j0ehi.
13 ongodb.net.
14
15 Authoritative answers can be found from:
16 > set type=TXT
17 > cluster0.j0ehi.mongodb.net
18 Server:      26.26.26.53
19 Address:     26.26.26.53#53

Non-authoritative answer:
cluster0.j0ehi.mongodb.net      text = "authSource=admin&replicaSet=atlas-r36spx-shard-0"

```

然后将查询到的服务器列表按 `host[:port][,...hostN[:portN]]` 格式填写到 **MongoDB** 服务器 选项中，并且按照查询到的 **TXT** 记录内容来配置 认证数据源 和 副本集名称：



最后，我们同样点击 **创建资源** 表单最下方的 **确定** 按钮以完成创建。

创建规则

1. 配置 SQL

资源创建完成后，我们还需创建相应的规则。点击规则页面左上角的 **创建** 按钮进入 **创建规则** 页面，输入以下 **SQL**：

```

1  SELECT
2      id as msgid,
3      topic,
4      qos,
5      payload,
6      publish_received_at as arrived
7  FROM
8      "t/#"

```

这个 **SQL** 表示所有主题与主题过滤器 `t/#` 匹配的消息都将触发这条规则，例如 `t/1`，`t/1/2` 等，并且使用筛选出来的 **msgid**, **topic** 等数据执行后续的动作。

EMQ X Enterprise

监控 / 规则引擎 / 创建

admin

编辑规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入:

```

1  SELECT
2      id as msgid,
3      topic,
4      qos,
5      payload,
6      publish_received_at as arrived
7  FROM
8      "t/#"

```

规则引擎是标准 MQTT 之上基于 SQL 的核心数据处理与分发组件，可以方便的筛选并处理 MQTT 消息与设备生命周期事件，并将数据分发移动到 HTTP Server、数据库、消息队列甚至是另一个 MQTT Broker 中。

1. 选择 't/#' 主题的消息，提取全部字段:
SELECT * FROM "t/#"

2. 通过事件主题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'

规则引擎和 SQL 语句的详细教程参见 [EMQ X 文档](#)。

规则 ID: rule:608990

描述:

SQL 测试:

* 响应动作

取消 确定

2. 添加响应动作

点击 **添加动作** 按钮，动作类型 选择数据持久话和保存数据到 **MongoDB**，然后在 使用资源 下拉列表选择一个我们刚刚创建的资源。**Collection** 按需配置，这里我配置为 **demo**。**消息内容模板** ** 保持为空，表示将 **SQL** 筛选出来的数据，以 **Key-Value** 列表的形式转换为 **Json** 数据写入 **MongoDB**。每条规则里面都可以添加多个响应动作，这里我们只需要用到一个响应动作，所以添加完以下动作时，就可以点击页面最下方的 **创建** 按钮完成规则的创建。

规则引擎是标准 MQTT 之上基于 SQL 的核心数据处理与分发组件，可以方便的筛选并处理 MQTT 消息与设备生命周期事件，并将数据分发移动到 HTTP Server、数据库、消息队列甚至是另一个 MQTT Broker 中。

- 选择 't/#' 主题的消息，提取全部字段：

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

- 通过事件主题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'
```

规则引擎和 SQL 语句的详细教程参见 [EMQ X 文档](#)。

测试验证

我们直接使用 **Dashboard** 中的 **MQTT** 客户端工具来发布一条消息。本示例中我们将消息主题改为 `t/1` 以命中我们设置的规则，**Payload** 和 **QoS** 保持不变，然后点击 **发布**。

Topic	QoS	已发送
t/1	0	暂无数据

Topic	QoS	Payload	时间
t/1	0	{ "msg": "hello" }	14:48:07

消息发布成功后，我们就可以在 **MongoDB Cloud** 的 **Cluster0** 集群实例的 **Collections** 页面看到刚刚写入的数据：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. At the top, there are navigation links for 'Test', 'Access Manager', 'Billing', 'All Clusters', 'Get Help', and a user profile for '子博'. Below the header, the main navigation bar includes 'test', 'Atlas' (which is highlighted in green), 'Realm', and 'Charts'. On the left, a sidebar menu is open under the 'DEPLOYMENT' section, showing 'Databases' (selected), 'Data Lake', 'DATA SERVICES', 'Triggers', 'Data API' (with a 'PREVIEW' button), and 'SECURITY' sections like 'Database Access', 'Network Access', and 'Advanced'. The main content area is titled 'TEST > TEST > DATABASES' and 'Cluster0'. It shows 'Collections' (selected) with tabs for 'Overview', 'Real Time', 'Metrics', 'Collections', 'Search', 'Profiler', 'Performance Advisor', 'Online Archive', and 'Command Line Tools'. A 'VISUALIZE YOUR DATA' button and a 'REFRESH' button are also present. Under 'Collections', it lists 'test.demo' with collection details: 'COLLECTION SIZE: 139B', 'TOTAL DOCUMENTS: 1', and 'INDEXES TOTAL SIZE: 20KB'. It has tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'FILTER' button with a query '{ field: 'value' }' and an 'OPTIONS' button are available. On the right, there is an 'INSERT DOCUMENT' button, a 'Reset' button, and a modal window for document editing with fields for '_id', 'arrived', 'msgid', 'payload', 'qos', and 'topic'. The '_id' field is set to ObjectId("61a477a76969e471cf000001"). The 'payload' field contains a JSON object: { "msg": "hello" }. The 'topic' field is set to "t/1". Buttons for 'CANCEL' and 'UPDATE' are at the bottom of the modal. A small message icon is located in the bottom right corner of the main content area.

保存数据到 Redis

搭建 Redis 环境，以 MacOS X 为例：

```

1 $ wget http://download.redis.io/releases/redis-4.0.14.tar.gz
2 $ tar xzf redis-4.0.14.tar.gz
3 $ cd redis-4.0.14
4 $ make && make install
5
6 # 启动 redis
7 $ redis-server

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

1 SELECT * FROM "t/#"

* SQL 输入:

```

1 SELECT
2 *
3 FROM
4 "t/#"

```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers			publish_received_at	timestamp	node	

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

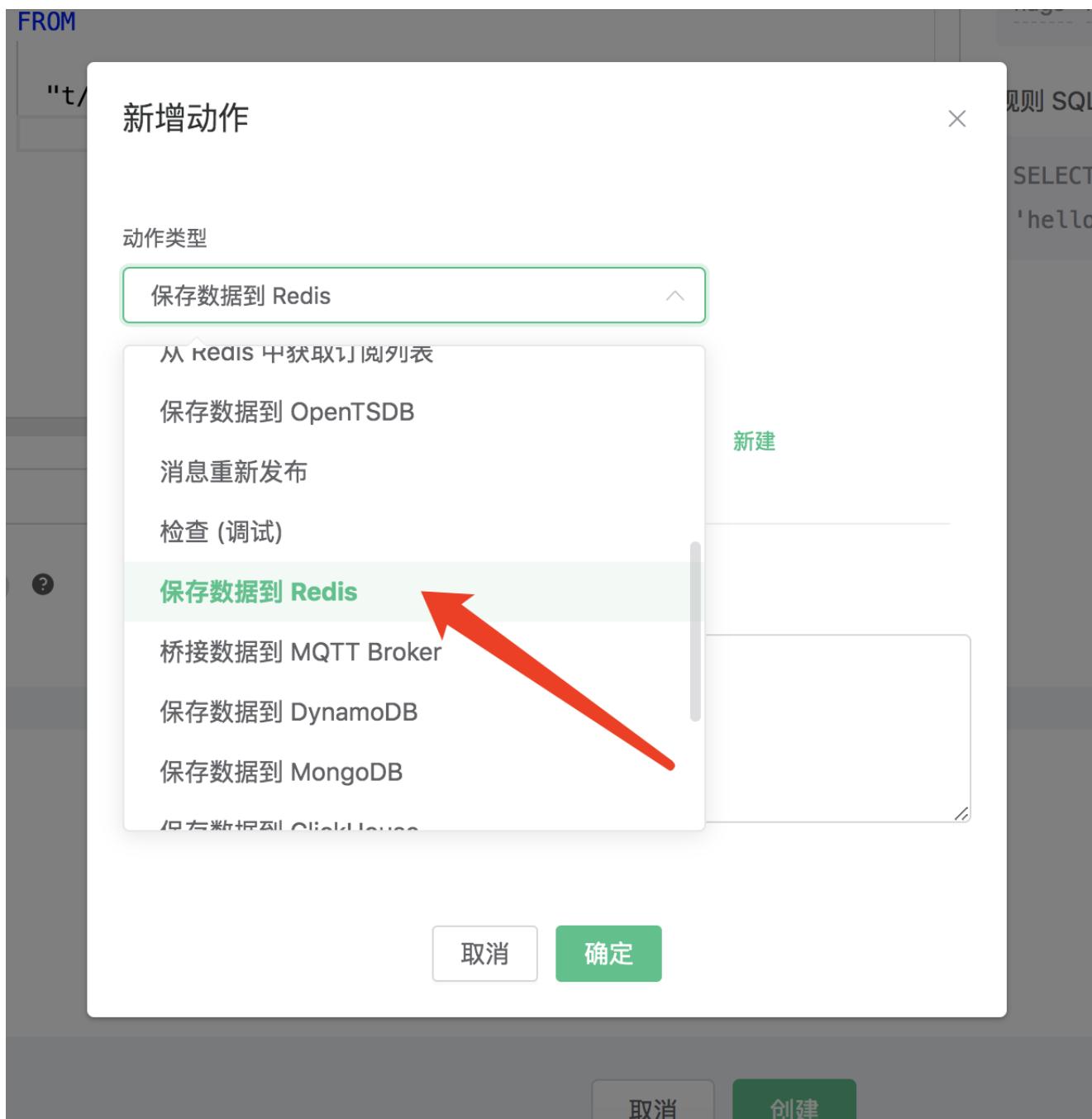
备注:

SQL 测试:

?

关联动作：

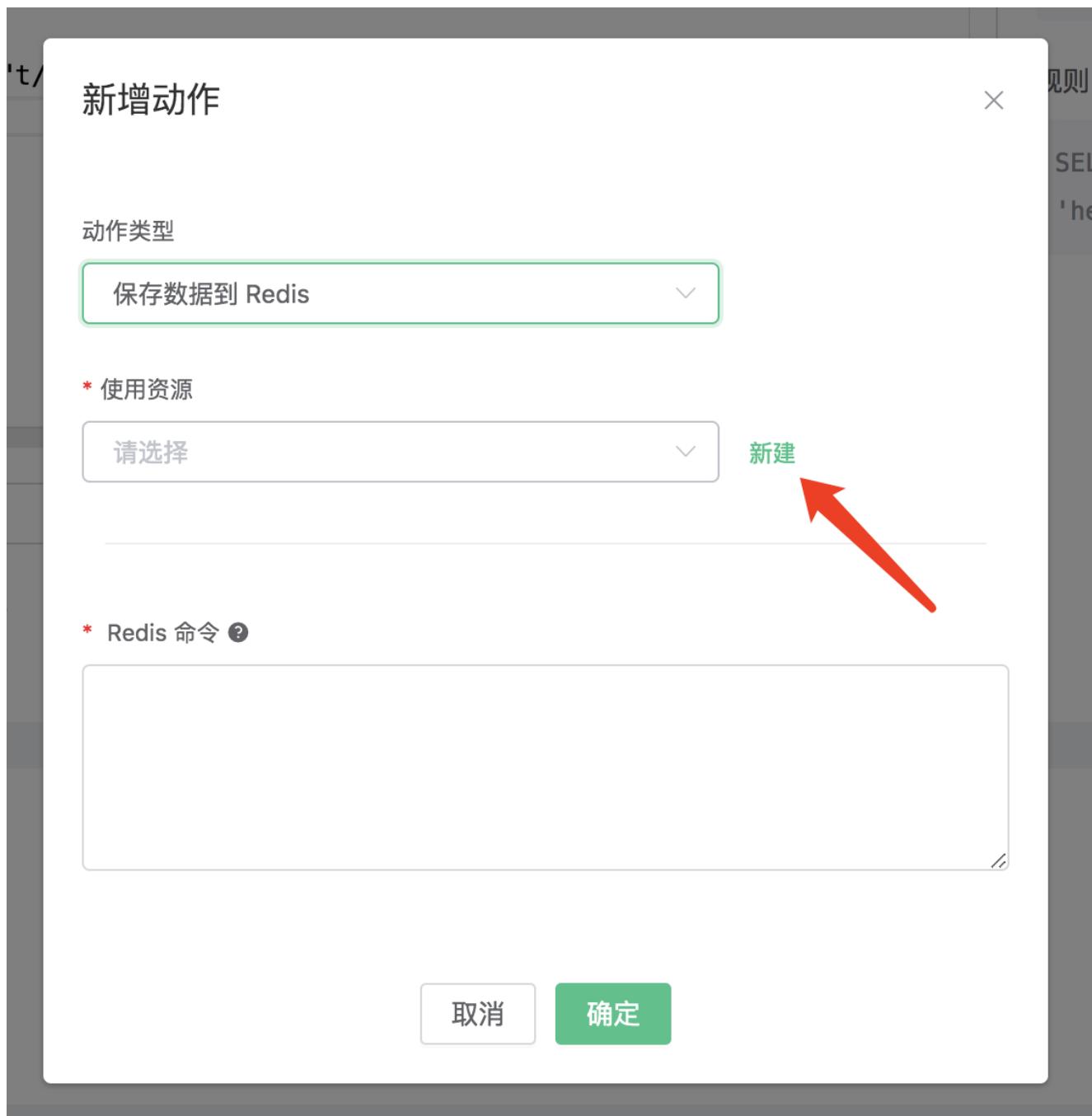
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 Redis”。



填写动作参数：

“保存数据到 Redis”动作需要两个参数：

1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 Redis 资源：

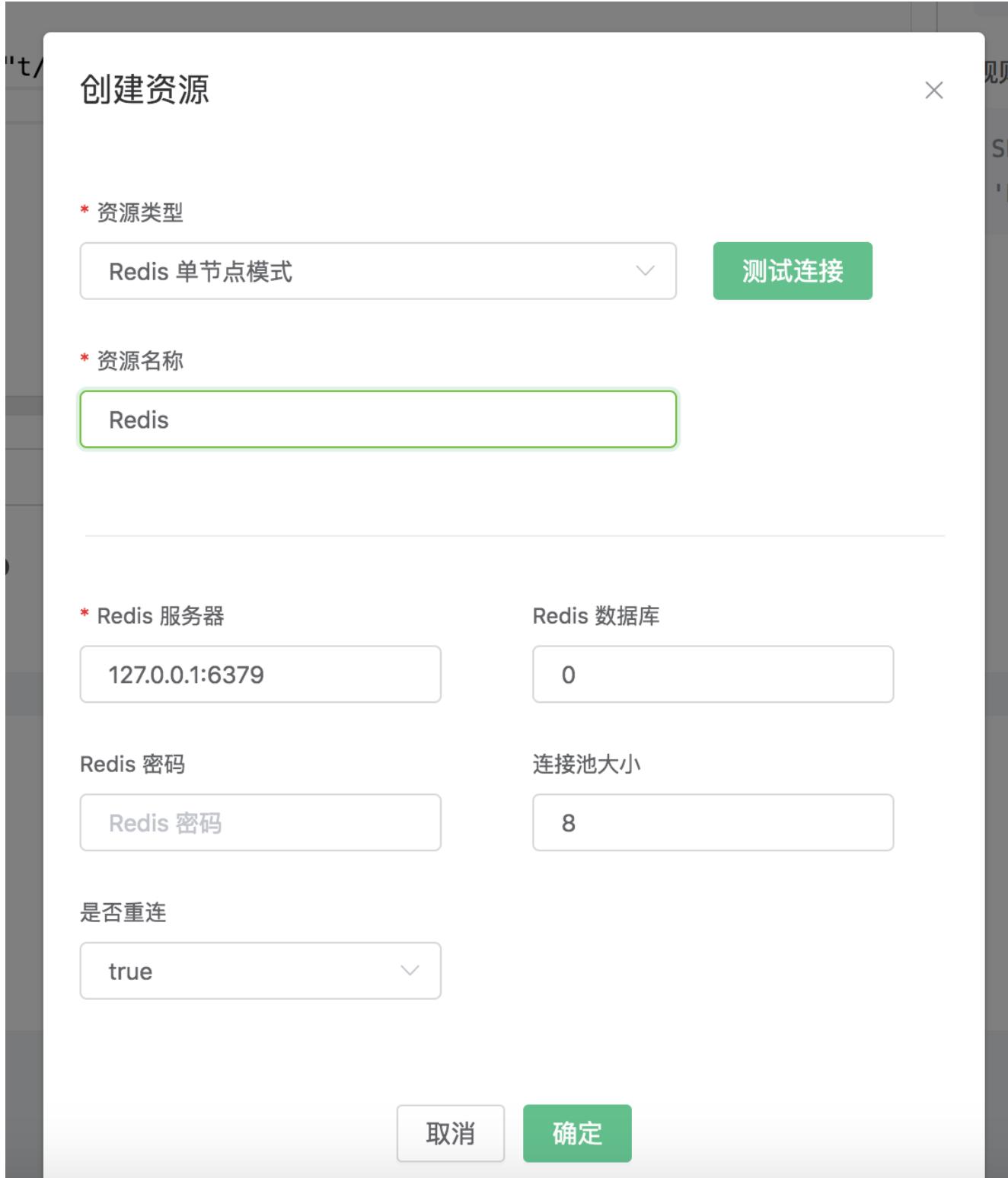


选择 **Redis** 单节点模式资源。

填写资源配置：

填写真实的 **Redis** 服务器地址，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

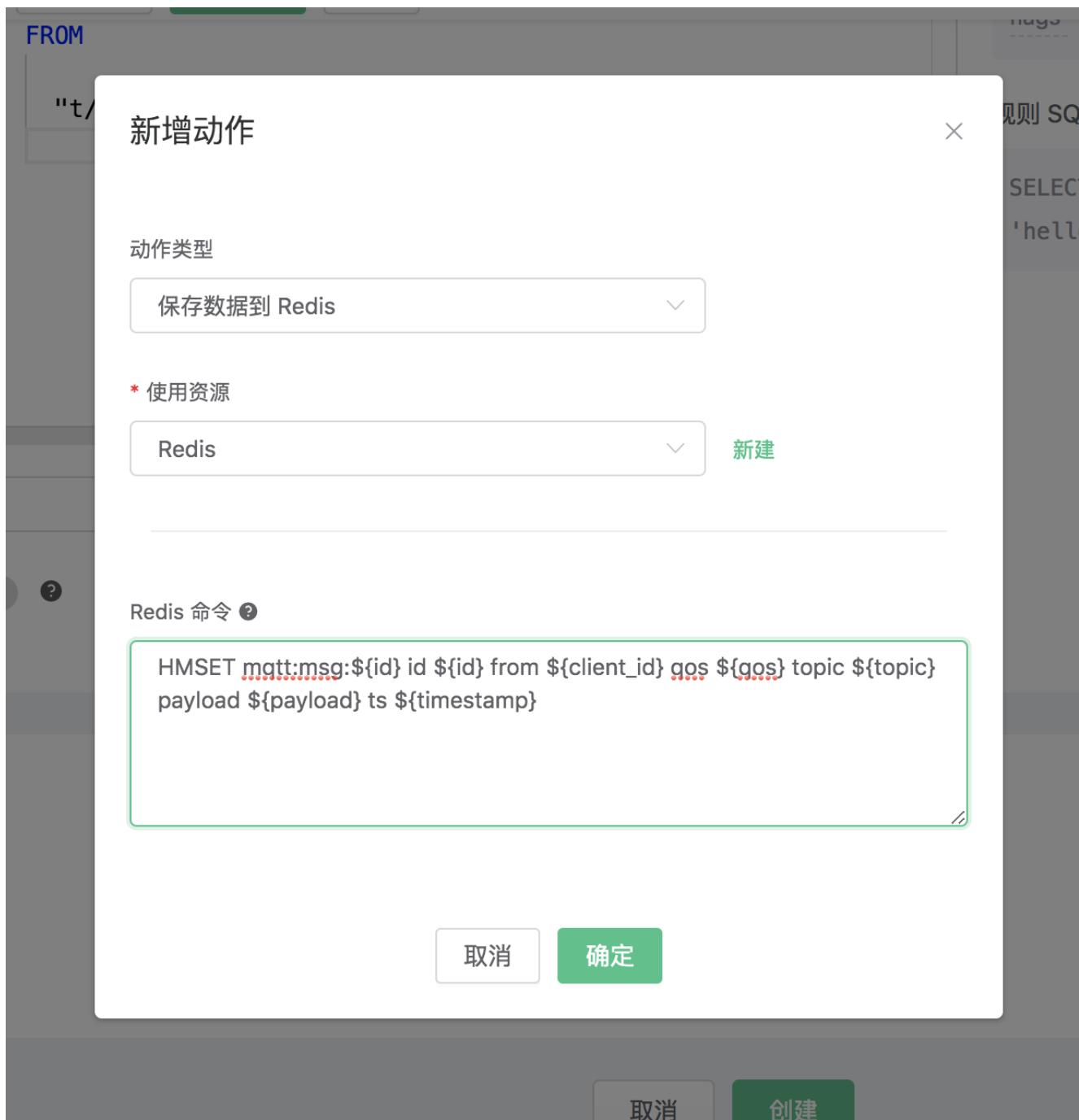
最后点击“新建”按钮。



返回响应动作界面，点击“确认”。

2). Redis 的命令:

```
1 HMSET mqtt:msg:${id} id ${id} from ${client_id} qos ${qos} topic ${topic} payload ${payload} ts ${timestamp}
```



返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 Redis (data_to_redis)
保存数据到 Redis

编辑 移除

资源 ID resource:60380ce0
Redis 命令 HMSET mqtt:msg:\${id} id \${id} from \${client_id} qos \${qos} topic \${topic} payload \${payload} ts \${timestamp}

+ 失败备选动作

+ 添加动作

取消 创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "hello"

```

然后通过 **Redis** 命令去查看消息是否生产成功:

```

1 $ redis-cli
2
3 KEYS mqtt:msg*
4
5 hgetall Key

```

```

[127.0.0.1:6379>
[127.0.0.1:6379> KEYS mqtt:msg*
1) "mqtt:msg:58CBE39D13232F4420000198F0001"
[127.0.0.1:6379> hgetall mqtt:msg:58CBE39D13232F4420000198F0001
1) "id"
2) "58CBE39D13232F4420000198F0001"
3) "from"
4) "mqttjs_b3719cdb09"
5) "qos"
6) "0"
7) "topic"
8) "t/1"
9) "payload"
10) "hello"
11) "retain"
12) "0"
13) "ts"
14) "1562123525239"
[127.0.0.1:6379>

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1:



ID	主题	监控	描述	状态	响应动作
rule:40cfa21d	t/#	1/1		<input checked="" type="checkbox"/>	保存数据到 Redis 编辑 删除

保存数据到 Cassandra

搭建 **Cassandra** 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```

1 $ brew install cassandra
2 ## 修改配置, 关闭匿名认证
3 $ vim /usr/local/etc/cassandra/cassandra.yaml
4
5     authenticator: PasswordAuthenticator
6     authorizer: CassandraAuthorizer
7
8 $ brew services start cassandra
9
10 ## 创建 root 用户
11 $ cqlsh -ucassandra -pcassandra
12
13 create user root with password 'public' superuser;

```

初始化 **Cassandra** 表：

```

1 $ cqlsh -uroot -ppublic

```

创建 "**test**" 表空间：

```

1 CREATE KEYSPACE test WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'
'} AND durable_writes = true;

```

创建 "**t_mqtt_msg**" 表：

```

1 USE test;
2 CREATE TABLE t_mqtt_msg (
3     msgid text,
4     topic text,
5     qos int,
6     payload text,
7     arrived timestamp,
8     PRIMARY KEY (msgid, topic)
9 );

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

```

1 SELECT * FROM "t/#"

```

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
```

当前事件可用字段

event id clientid username payload peerhost topic qos
flags headers publish_received_at timestamp node

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg =  
'hello'
```

备注:

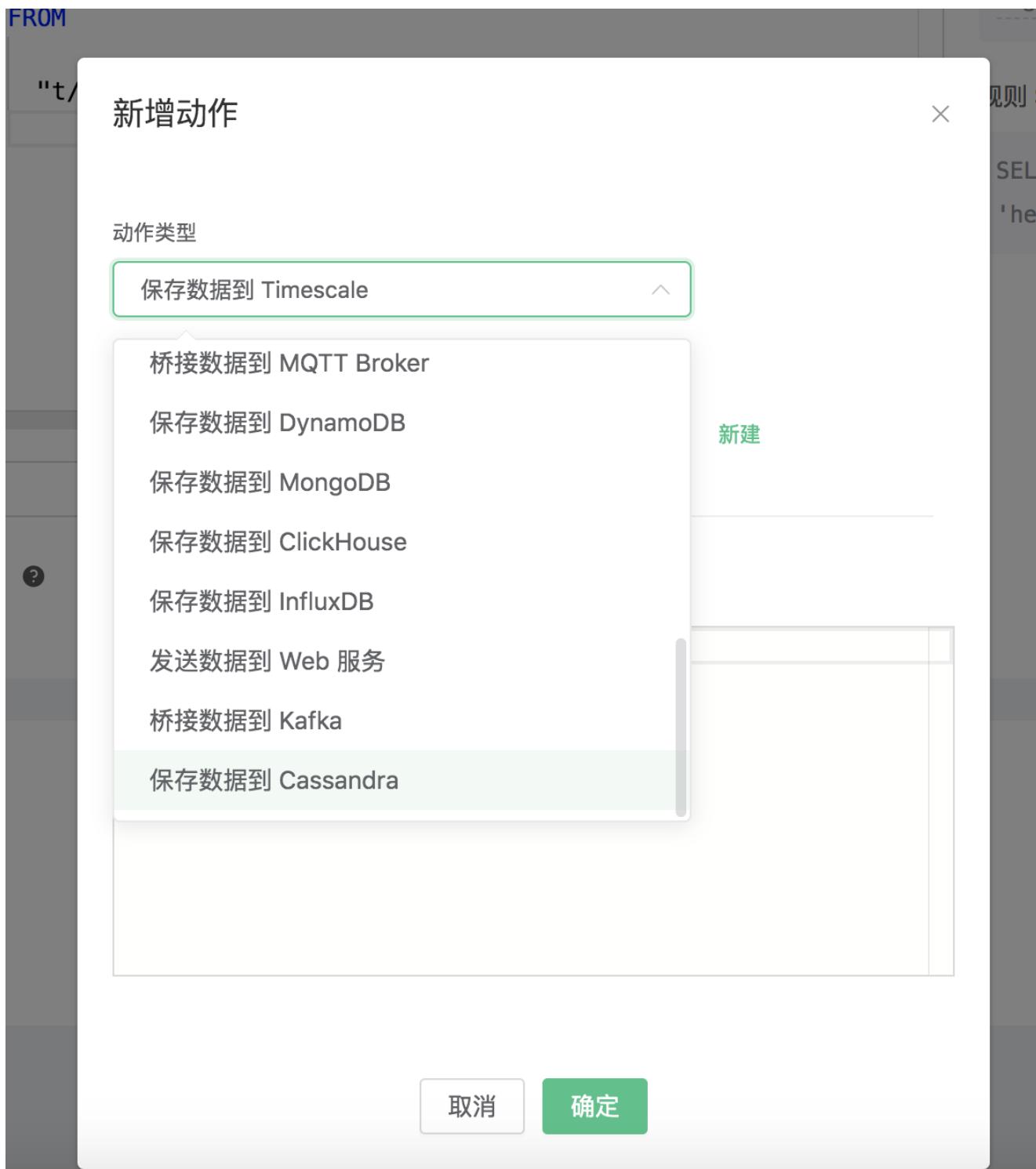
SQL 测试:



?

关联动作:

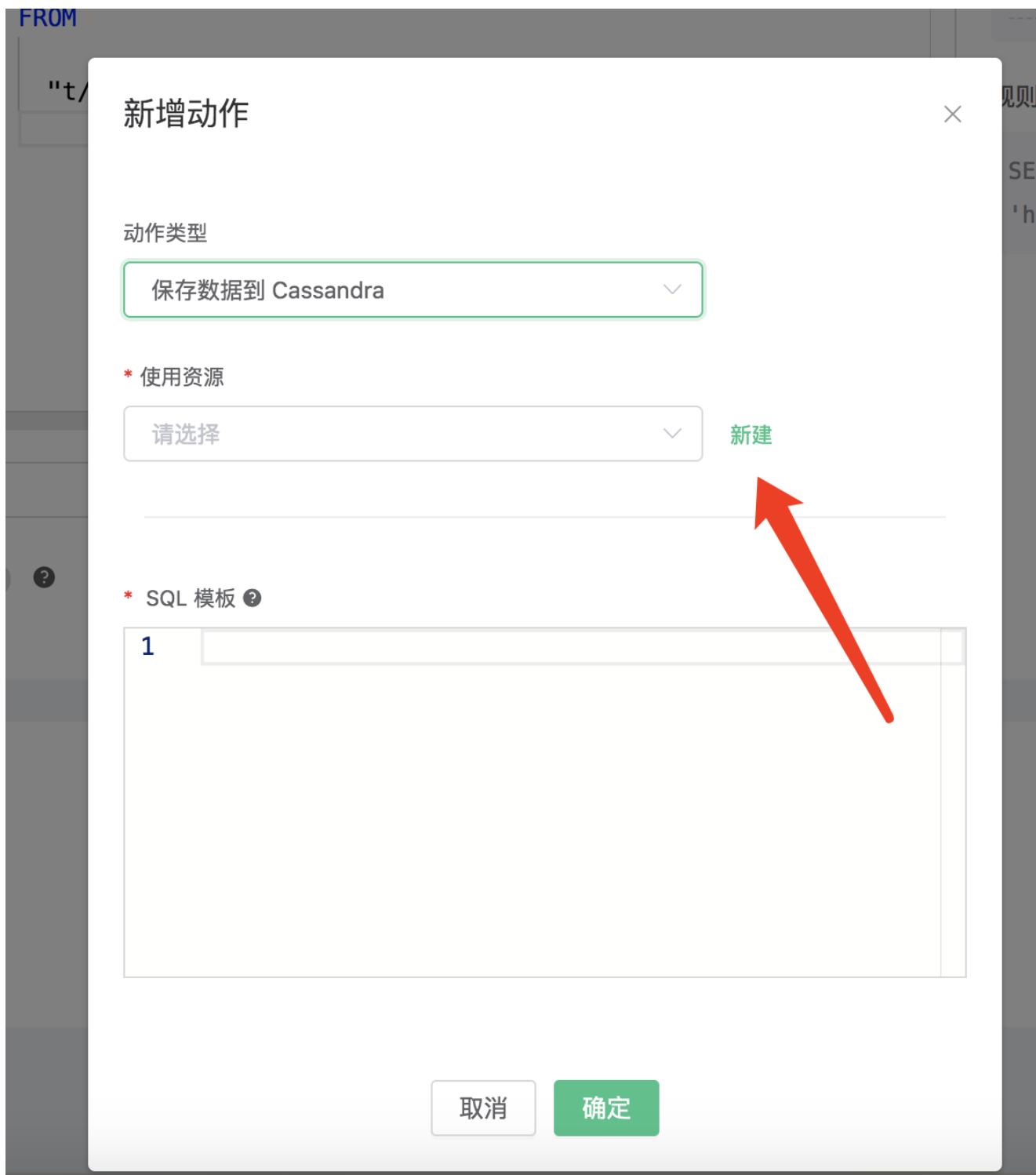
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **Cassandra**”。



填写动作参数:

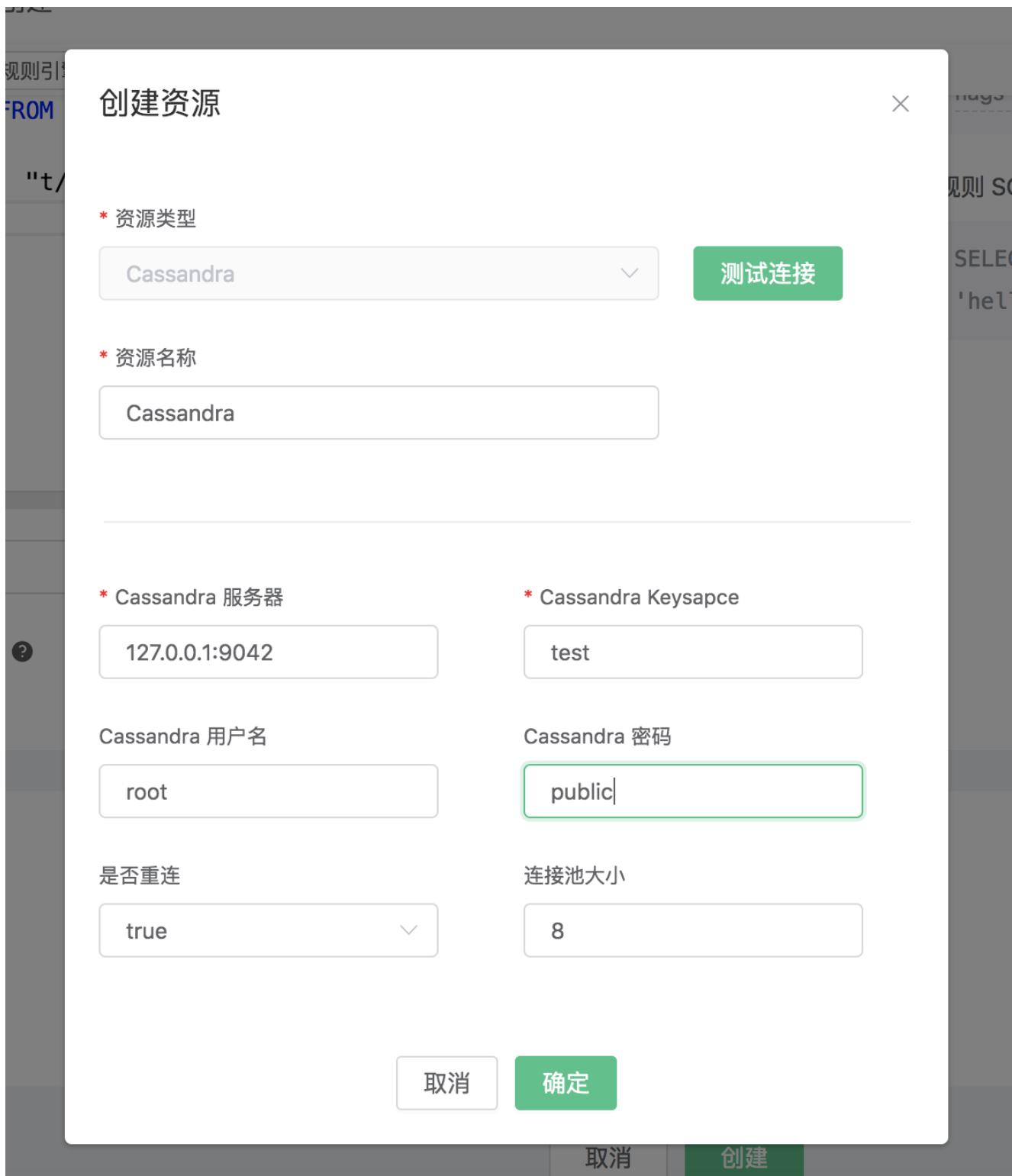
"保存数据到 Cassandra" 动作需要两个参数:

1). 关联资源的 ID。初始状况下，资源下拉框为空，现点击右上角的“新建资源”来创建一个 Cassandra 资源。



填写资源配置：

Keyspace 填写 “test”，用户名填写 “root”，密码填写 “public” 其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

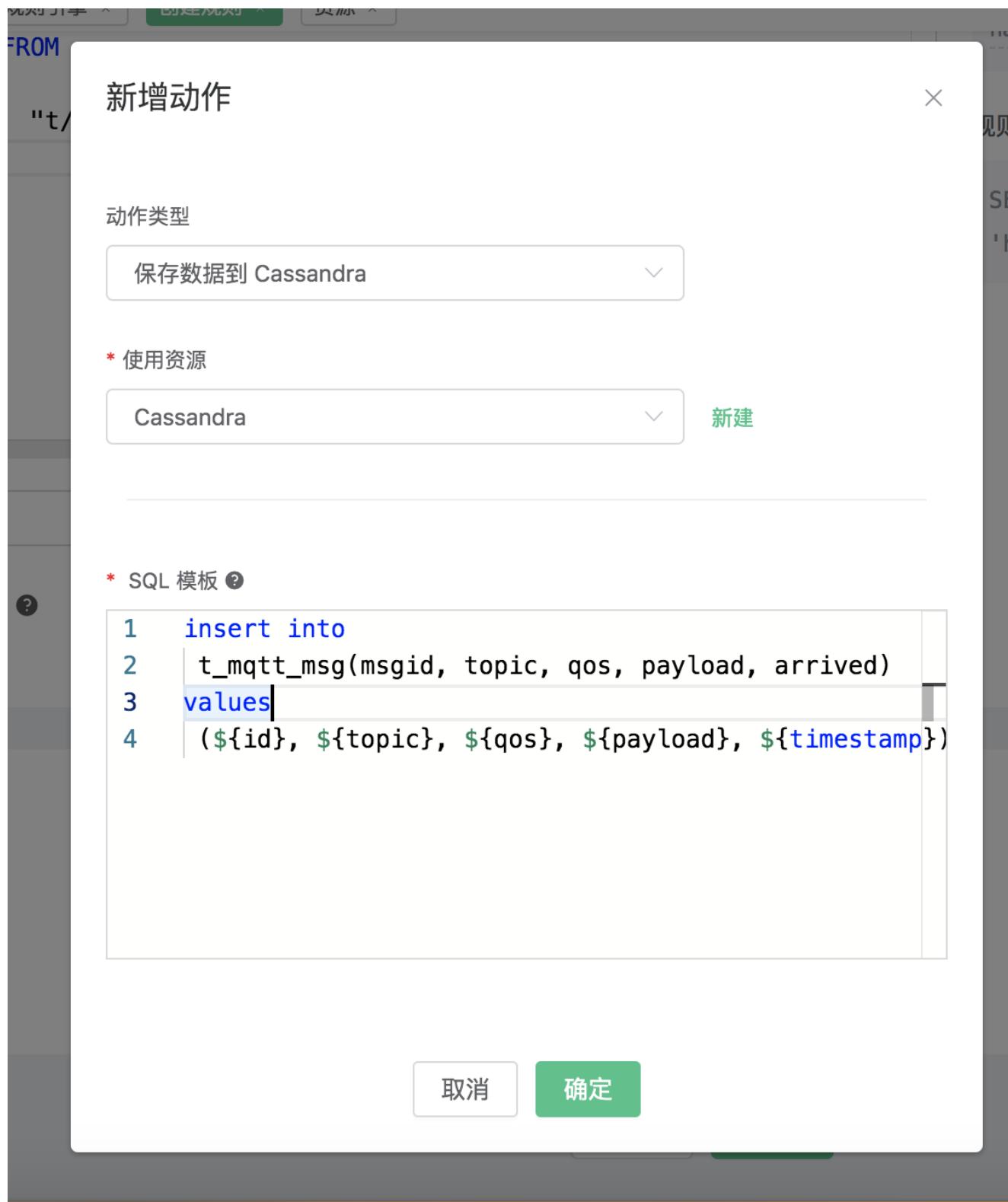


点击“新建”按钮，完成资源的创建。

2). SQL 模板。这个例子里我们向 **Cassandra** 插入一条数据，**SQL** 模板为：

```
1 insert into t_mqtt_msgmsgid, topic, qos, payload, arrived) values (${id}, ${topic}, ${qos}, ${payload}, ${timestamp})
```

插入数据之前，**SQL** 模板里的 **\${key}** 占位符会被替换为相应的值。



在点击“新建”完成规则创建

响应动作 *

处理命中规则的消息

动作类型 保存数据到 Cassandra (data_to_cassa)
保存数据到 Cassandra 数据库
SQL 模板 insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values (\${id}, \${topic}, \${qos}, \${payload}, \${timestamp})
资源 ID resource:3ea6302

编辑 移除
+ 失败备选动作

+ 添加动作

取消 创建

现在发送一条数据，测试该规则：

```

1 Topic: "t/cass"
2 QoS: 1
3 Payload: "hello"

```

然后检查 **Cassandra** 表，可以看到该消息已成功保存：

```

root@cqlsh:test> select * from t_mqtt_msg;

  msgid           | topic   | arrived
-----+-----+-----+-----+-----+-----+
58C4A92E6A7ACF44000000A9F0001 | t/cass | 2019-06-27 09:13:23.612000+0000 | hello | 1 | 1

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

ID	主题	监控	描述	状态	响应动作
rule:775a468c	t/#	1		<input checked="" type="checkbox"/>	保存数据到 Cassandra 编辑 删除

保存数据到 DynamoDB

搭建 **DynamoDB** 数据库，以 **MacOS X** 为例：

```
1 $ brew install dynamodb-local  
2 $ dynamodb-local
```

sh

创建 **DynamoDB** 表定义文件 **mqtt_msg.json**：

```
1 {  
2     "TableName": "mqtt_msg",  
3     "KeySchema": [  
4         { "AttributeName": "msgid", "KeyType": "HASH" }  
5     ],  
6     "AttributeDefinitions": [  
7         { "AttributeName": "msgid", "AttributeType": "S" }  
8     ],  
9     "ProvisionedThroughput": {  
10         "ReadCapacityUnits": 5,  
11         "WriteCapacityUnits": 5  
12     }  
13 }
```

json

初始化 **DynamoDB** 表：

```
1 $ aws dynamodb create-table --cli-input-json file://mqtt_msg.json --endpoint-url http://localhost:8000
```

sh

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

```
1 SELECT id, topic, payload FROM "#"
```

* SQL 输入:

```

1 SELECT
2   id,
3   topic,
4   payload
5
6 FROM
7   "#"

```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at			timestamp	node	

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

备注:

SQL 测试:

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **DynamoDB**”。

FROM
"#"

新增动作

动作类型

保存数据到 DynamoDB

检查 (调试)

保存数据到 Redis

桥接数据到 MQTT Broker

保存数据到 DynamoDB

保存数据到 MongoDB

保存数据到 ClickHouse

保存数据到 InfluxDB

发送数据到 Web 服务

新建

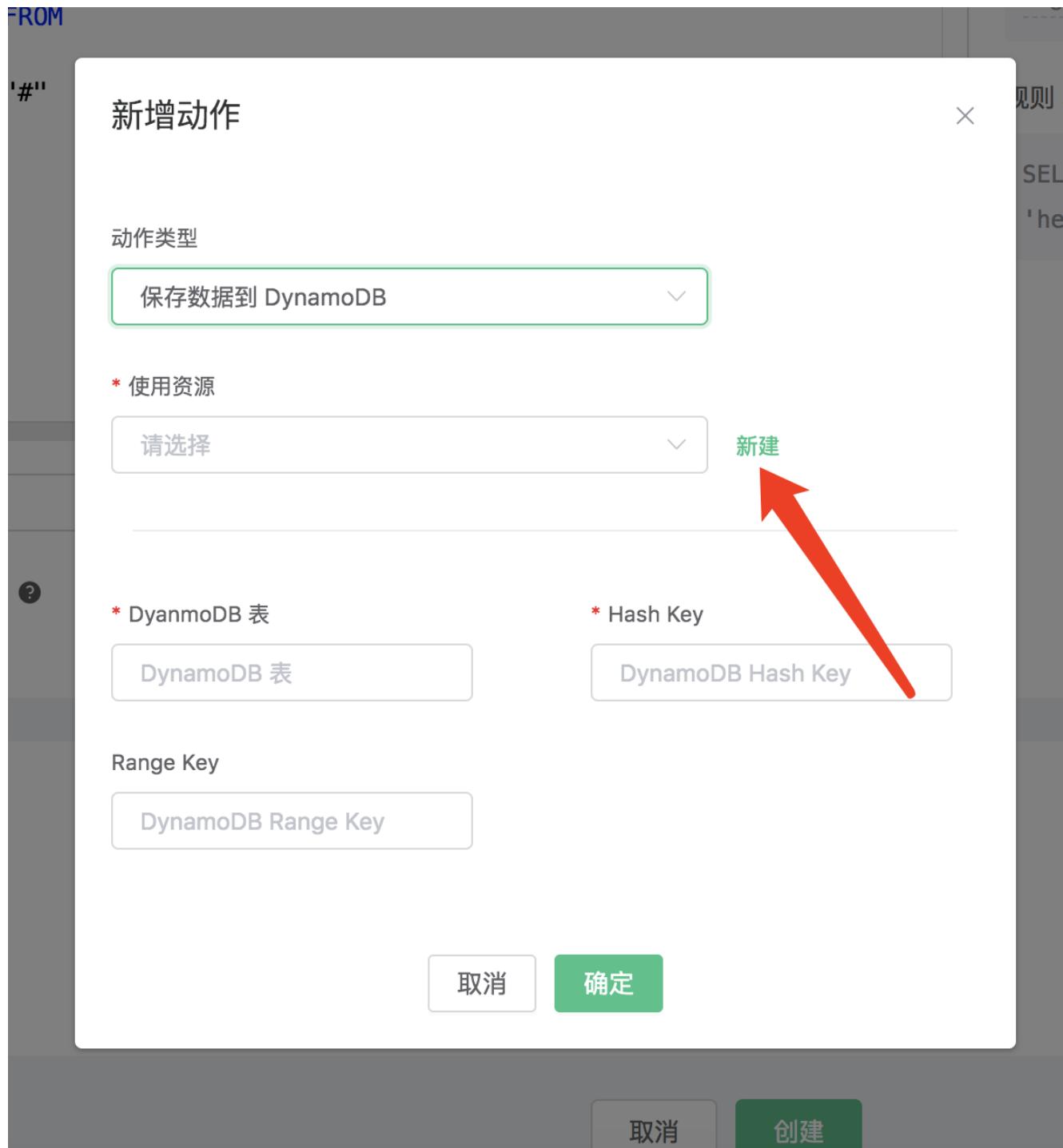
DB Hash Key

取消 确定

填写动作参数：

“保存数据到 **DynamoDB**” 动作需要两个参数：

- 1). **DynamoDB 表名**。这个例子里我们设置的表名为 "**mqtt_msg**"
- 2). **DynamoDB Hash Key**。这个例子里我们设置的 **Hash Key** 要与表定义的一致
- 3). **DynamoDB Range Key**。由于我们表定义里没有设置 **Range Key**。这个例子里我们把 **Range Key** 设置为空。



- 4). 关联资源的 **ID**。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **DynamoDB** 资源：

填写资源配置：

规则引擎 × 创建规则 × 资源 ×

ROM

创建资源

X

#"

规则 SC
SELECT
'hell

* 资源类型

DynamoDB

测试连接

* 资源名称

请输入

* DynamoDB 区域

us-west-2

DynamoDB 服务器

http://127.0.0.1:8000

* 连接池大小

8

* 连接访问 ID

aws_access_key_id

* 连接访问密钥

aws_secret_access_key

自动重连间隔

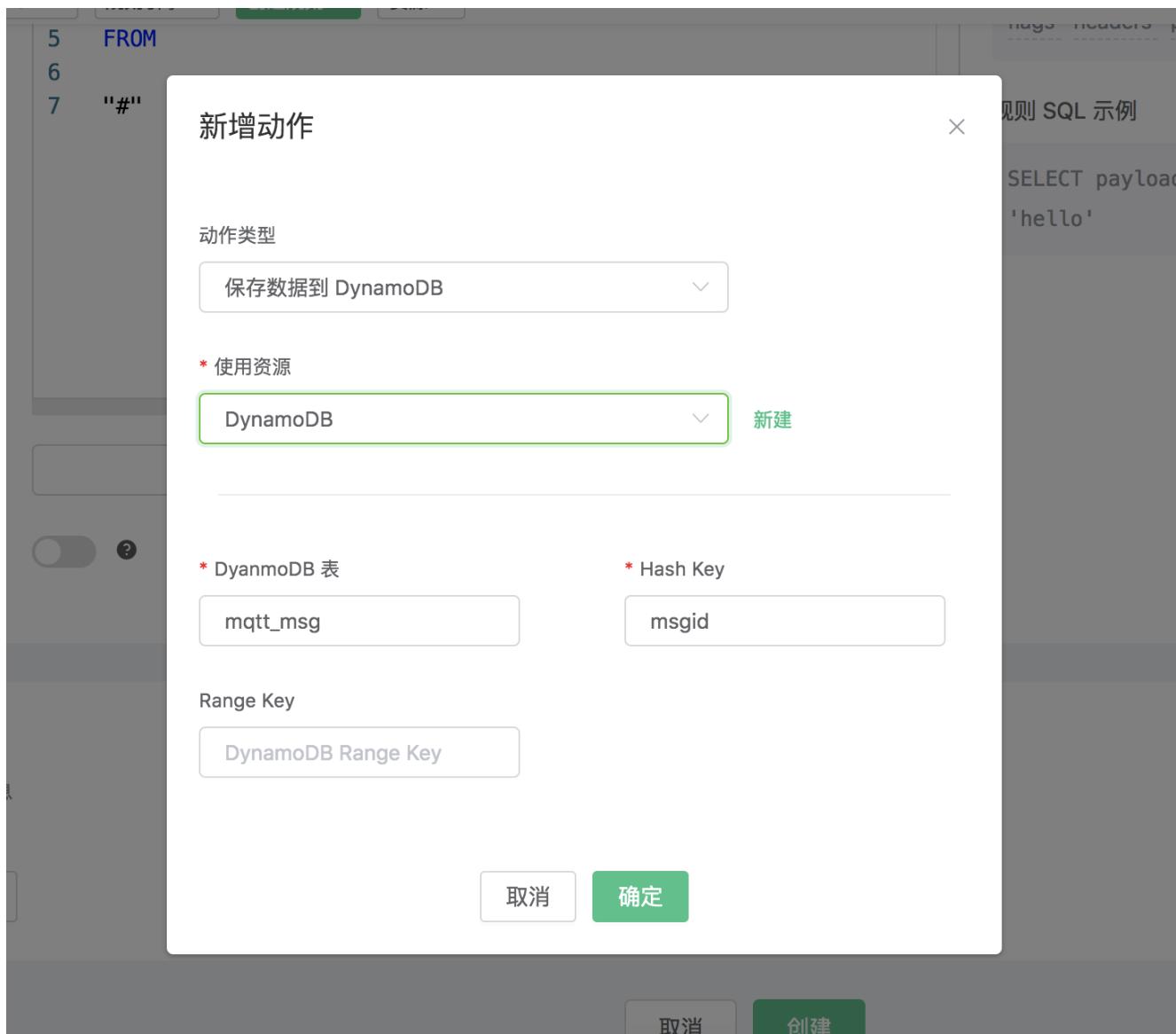
2

取消

确定

点击“新建”按钮。

返回响应动作界面，点击“确认”。



返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 DynamoDB (data_to_dynamo)		编辑 移除
保存数据到 DynamoDB		
DynamoDB 表	mqtt_msg	Hash Key
Range Key	msgid	
资源 ID resource:a43b8ba9		
		+ 失败备选动作

+ 添加动作

取消 创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/a"
2 QoS: 1
3 Payload: "hello"

```

然后检查 **DynamoDB** 的 **mqtt_msg** 表，新的 **record** 是否添加成功：

```
└▶ aws dynamodb scan --table-name mqtt_msg --region us-west-2 --endpoint-url http://localhost:8000
{
  "Items": [
    {
      "topic": {
        "S": "t/1"
      },
      "payload": {
        "S": "{ \"msg\": \"Hello, World!\" }"
      },
      "msgid": {
        "S": "58D64DDA6C3F5F4430000ED60002"
      }
    }
  ],
  "Count": 1,
  "ScannedCount": 1,
  "ConsumedCapacity": null
}
```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

ID	主题	监控	描述	状态	响应动作
rule:ab6b6f19	#	1		<input checked="" type="checkbox"/>	保存数据到 Dynamo DB 编辑 删除

保存数据到 ClickHouse

搭建 ClickHouse 数据库，并设置用户名密码为 **default/public**，以 CentOS 为例：

```

1 ## 安装依赖
2 sudo yum install -y epel-release
3
4 ## 下载并运行packagecloud.io提供的安装shell脚本
5 curl -s https://packagecloud.io/install/repositories/altinity/clickhouse/script.rpm.sh | sudo bash
6
7
8 ## 安装ClickHouse服务器和客户端
9 sudo yum install -y clickhouse-server clickhouse-client
10
11 ## 启动ClickHouse服务器
12 clickhouse-server
13
14 ## 启动ClickHouse客户端程序
clickhouse-client

```

创建 “**test**” 数据库：

```

1 create database test;

```

创建 **t_mqtt_msg** 表：

```

1 use test;
2 create table t_mqtt_msg (msgid Nullable(String), topic Nullable(String), clientid Nullable(String), payload Nullable(String)) engine = Log;

```

```

[i-pvbq9baz :] use test;
] USE test
Ok.

0 rows in set. Elapsed: 0.001 sec.

[i-pvbq9baz :] create table t_mqtt_msg (msgid Nullable(String), topic Nullable(String), clientid Nullable(String), payload Nullable(String)) engine = Log;
CREATE TABLE t_mqtt_msg
(
    `msgid` Nullable(String),
    `topic` Nullable(String),
    `clientid` Nullable(String),
    `payload` Nullable(String)
)
ENGINE = Log
Ok.

0 rows in set. Elapsed: 0.003 sec.

[i-pvbq9baz :]
]
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

1 SELECT * FROM "#"

* SQL 输入:

```

1 SELECT
2 *
3 FROM
4
5 "t/#"
6
7

```

当前事件可用字段

```

id clientid username payload peerhost topic qos flags
headers publish_received_at timestamp node

```

规则 SQL 示例

```

SELECT payload.msg as msg FROM "t/#" WHERE msg =
'hello'

```

备注:

SQL 测试:

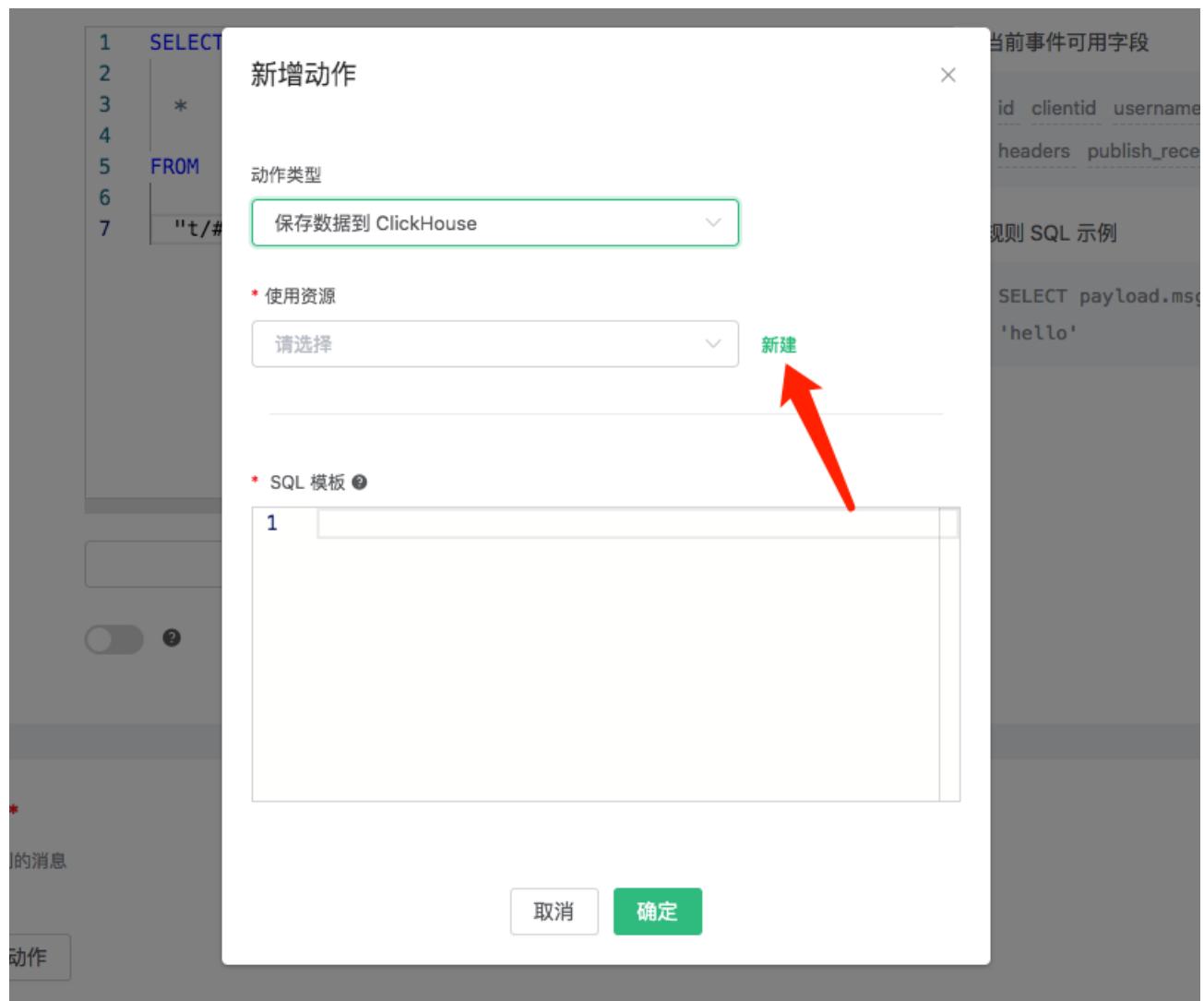
关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 ClickHouse”。

填写动作参数:

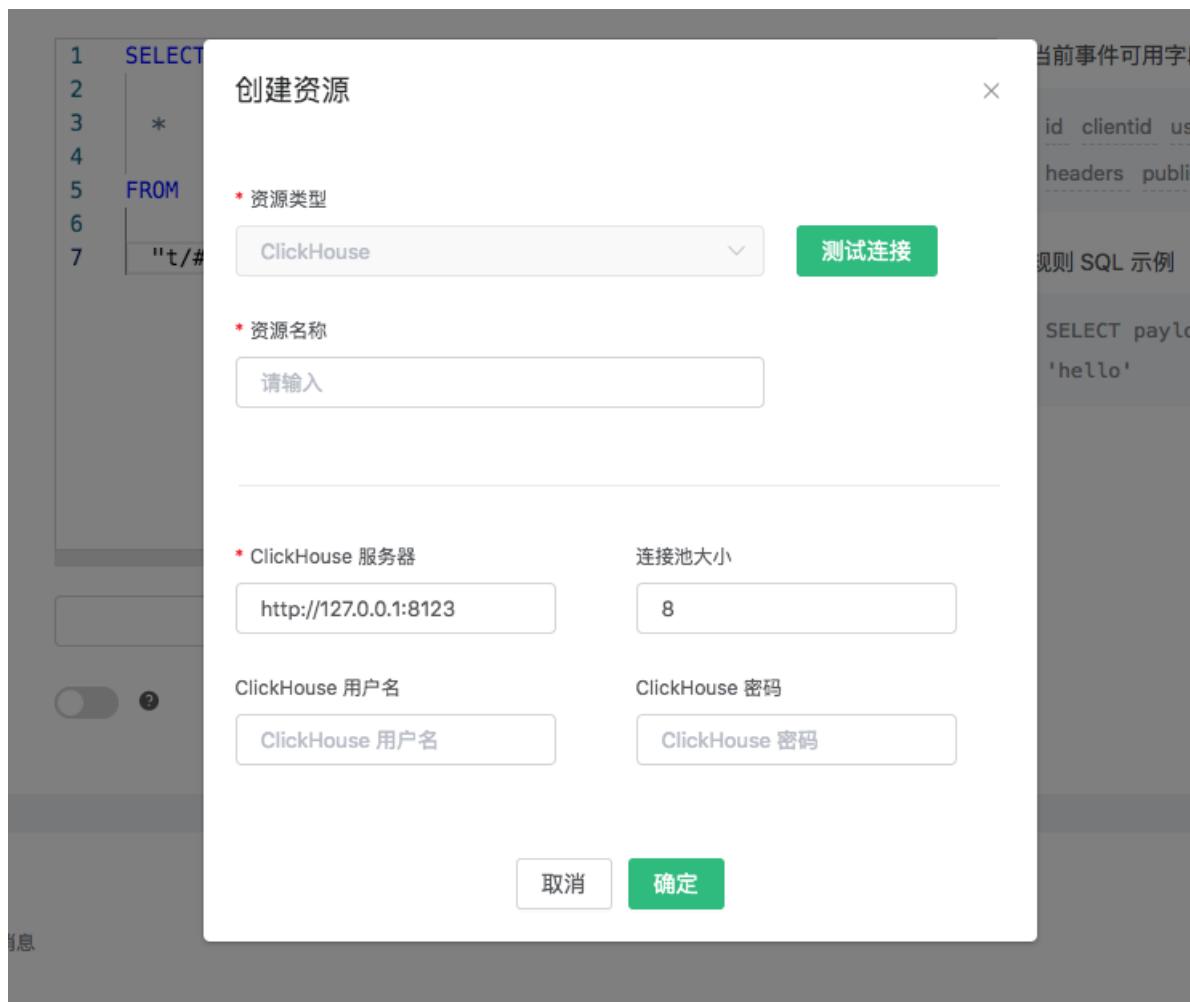
“保存数据到 ClickHouse” 动作需要两个参数：

1). 关联资源的 ID。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 ClickHouse 资源：



选择 “ClickHouse 资源”。

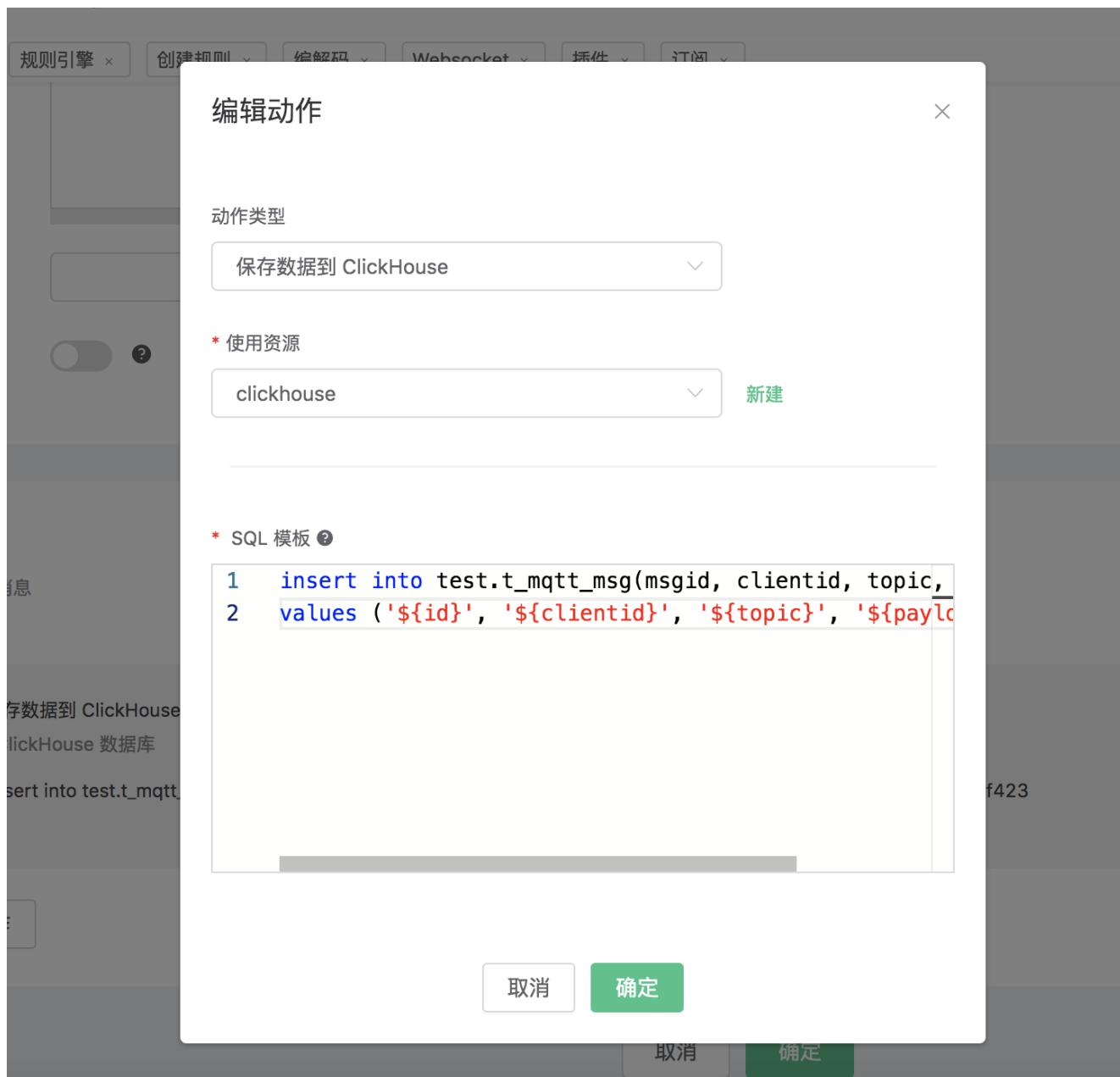
填写资源配置：



点击“新建”按钮。

2). SQL 模板。这个例子里我们向 **ClickHouse** 插入一条数据，SQL 模板为：

```
1 insert into test.t_mqtt_msg(msgid, clientid, topic, payload) values ('${id}', '${clientid}', '${topic}', '${payload}')
```



返回响应动作界面，点击“确认”。

规则列表					
ID	主题	监控	描述	状态	响应动作
rule:b817f256	t/#	开		开启	保存数据到 ClickHouse 编辑 删除

在规则列表里，点击“查看”按钮或规则 ID 连接，可以预览刚才创建的规则：

备注:

查询字段: *

筛选条件:

规则 SQL:

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

响应动作

命中规则的消息处理方式

动作类型 保存数据到 ClickHouse (data_to_clickhouse)
 成功 0 失败 0
 保存数据到 ClickHouse 数据库
 详细统计 点击查看 SQL 模板 insert into test.t_mqtt_msg(msgid, clientid, topic, payload) values ('\${id}', '\${clientid}', '\${topic}', '\${payload}')
 资源 ID resource:2e5ff423

规则已经创建完成，现在发一条数据:

1	Topic: "t/a"	sh
2	QoS: 1	
3	Payload: "hello"	

然后检查 **ClickHouse** 表，新的 **record** 是否添加成功:

```
[i-pvbq9baz :) select * from test.t_mqtt_msg;
```

```
SELECT *
FROM test.t_mqtt_msg
```

msgid	topic	clientid	payload
5AA4CF9A7611AC84C040045820001	t/a	test1	{ "hello" }

```
1 rows in set. Elapsed: 0.009 sec.
```

```
i-pvbq9baz :)
```

保存数据到 OpenTSDB

搭建 OpenTSDB 数据库环境，以 MacOS X 为例：

```
1 $ docker pull petergrace/opentsdb-docker  
2  
3 $ docker run -d --name opentsdb -p 4242:4242 petergrace/opentsdb-docker
```

创建规则：

打开 EMQX Dashboard，选择左侧的“规则”选项卡。

填写规则 SQL:

```
1 SELECT
2     payload.metric as metric,
3     payload.tags as tags,
4     payload.value as value
5 FROM
6     "#"
```

* SQL 输入:

```

1 SELECT
2
3     payload as p,
4     p.metric as metric, p.tags as tags, p.value as value
5
6 FROM
7
8     "#"

```

当前事件可用字段

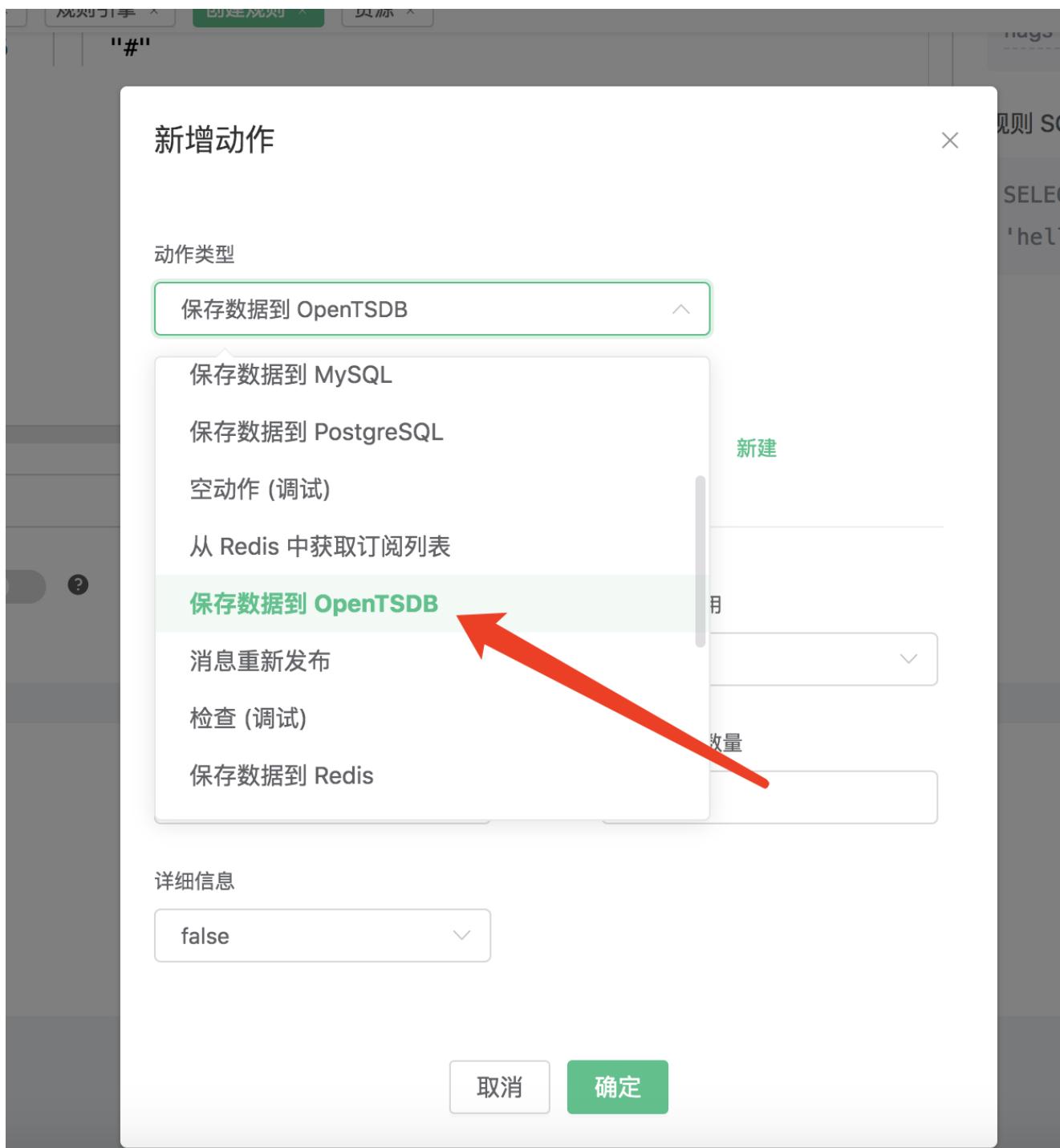
event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at	timestamp	node			

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

美联储动作·

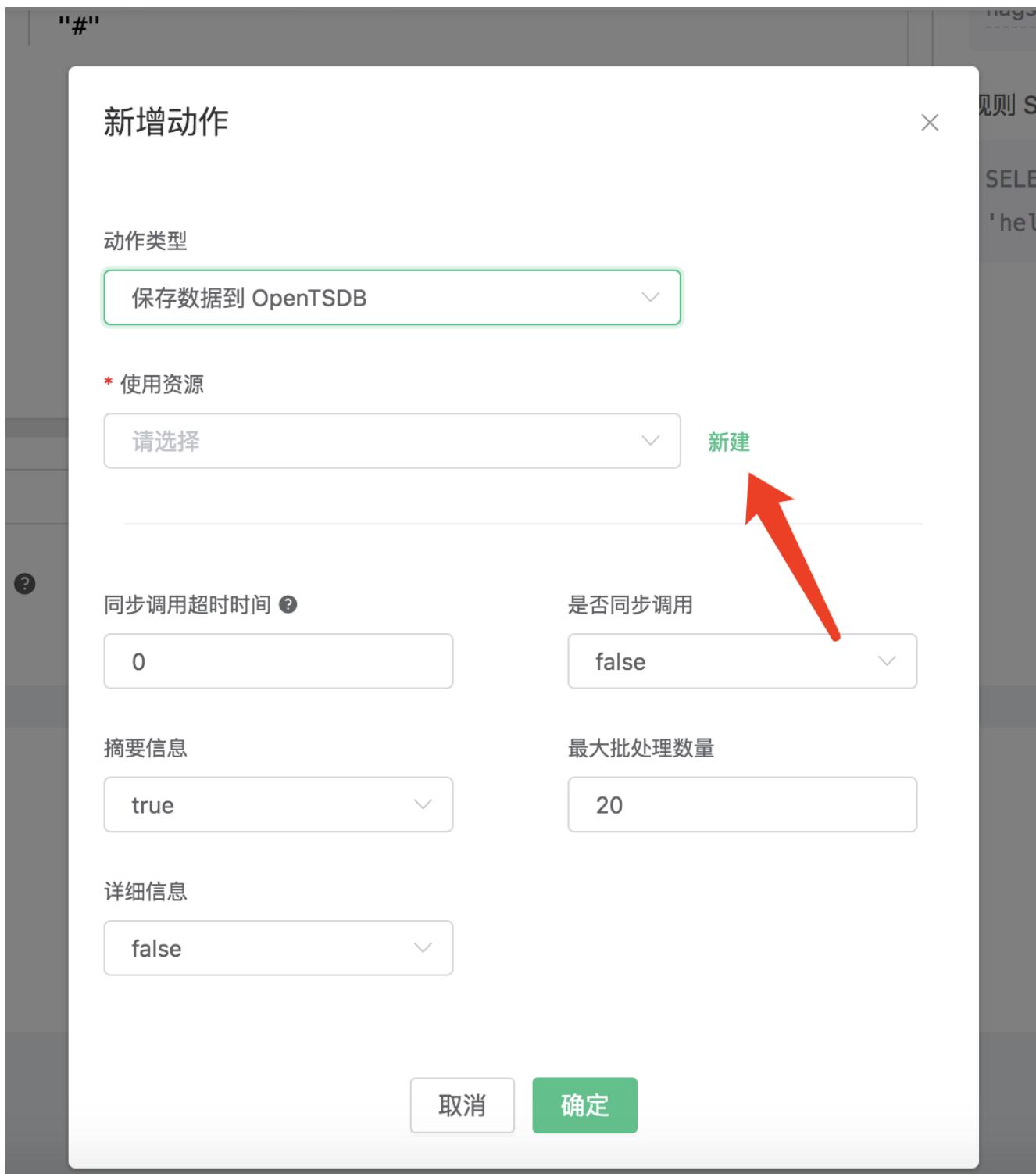
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 OpenTSDB”。



填写动作参数:

“保存数据到 OpenTSDB” 动作需要六个参数:

- 1). 详细信息。是否需要 OpenTSDB Server 返回存储失败的 **data point** 及其原因的列表，默認為 **false**。
- 2). 摘要信息。是否需要 OpenTSDB Server 返回 **data point** 存储成功与失败的数量，默認為 **true**。
- 3). 最大批处理数量。消息请求频繁时允许 OpenTSDB 驱动将多少个 **Data Points** 合并为一次请求，默認為 **20**。
- 4). 是否同步调用。指定 OpenTSDB Server 是否等待所有数据都被写入后才返回结果，默認為 **false**。
- 5). 同步调用超时时间。同步调用最大等待时间，默認為 **0**。
- 6). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 OpenTSDB 资源:



选择 “OpenTSDB 资源”：

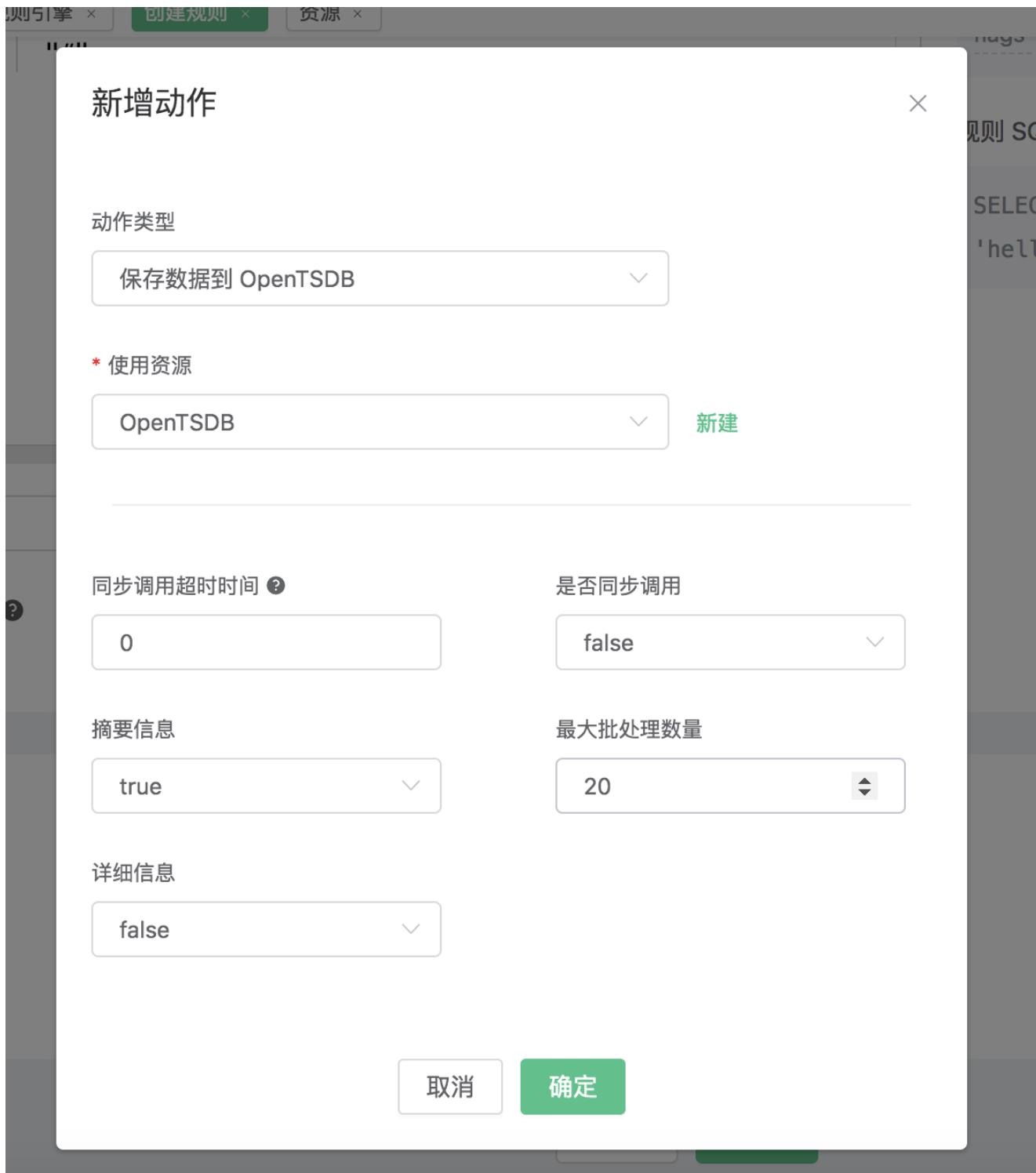
填写资源配置：

本示例中所有配置保持默认值即可，点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。



返回响应动作界面，点击“确认”。



返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 OpenTSDB (data_to_opentsdb) 保存数据到 OpenTSDB	编辑 移除
同步调用超时时间 0 是否同步调用 false 摘要信息 true 最大批处理数量 20 详细信息 false 资源 ID resource:b87ad29f	+ 失败备选动作

+ 添加动作 取消 创建

规则已经创建完成，现在发一条消息：

```
1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "{\"metric\":\"cpu\", \"tags\": {\"host\":\"serverA\"}, \"value\":12}"
```

我们通过 **Postman** 或者 **curl** 命令，向 **OpenTSDB Server** 发送以下请求：

```
1 POST /api/query HTTP/1.1
2 Host: 127.0.0.1:4242
3 Content-Type: application/json
4 cache-control: no-cache
5 Postman-Token: 69af0565-27f8-41e5-b0cd-d7c7f5b7a037
6 {
7     "start": 1560409825000,
8     "queries": [
9         {
10             "aggregator": "last",
11             "metric": "cpu",
12             "tags": {
13                 "host": "*"
14             }
15         }
16     ],
17     "showTSUIDs": "true",
18     "showQuery": "true",
19     "delete": "false"
20 }
21 -----WebKitFormBoundary7MA4YWxkTrZu0gW--
```

如果 **data point** 存储成功，将会得到以下应答：

```

1   [
2     {
3       "metric": "cpu",
4       "tags": {
5         "host": "serverA"
6       },
7       "aggregateTags": [],
8       "query": {
9         "aggregator": "last",
10        "metric": "cpu",
11        "tsuids": null,
12        "downsample": null,
13        "rate": false,
14        "filters": [
15          {
16            "tag": "host",
17            "filter": "*",
18            "group_by": true,
19            "type": "wildcard"
20          }
21        ],
22        "index": 0,
23        "tags": {
24          "host": "wildcard(*)"
25        },
26        "rateOptions": null,
27        "filterTagKs": [
28          "AAAC"
29        ],
30        "explicitTags": false
31      },
32      "tsuids": [
33        "000002000002000007"
34      ],
35      "dps": {
36        "1561532453": 12
37      }
38    }
39  ]

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：



ID	主题	监控	描述	状态	响应动作
rule:366faafc	#	1		开启	保存数据到 OpenTSDB [编辑] [删除]

保存数据到 InfluxDB

启动 **InfluxDB**，请确保启用了相应的 **Listener**（我们假设您已经成功安装了 **InfluxDB** 环境）

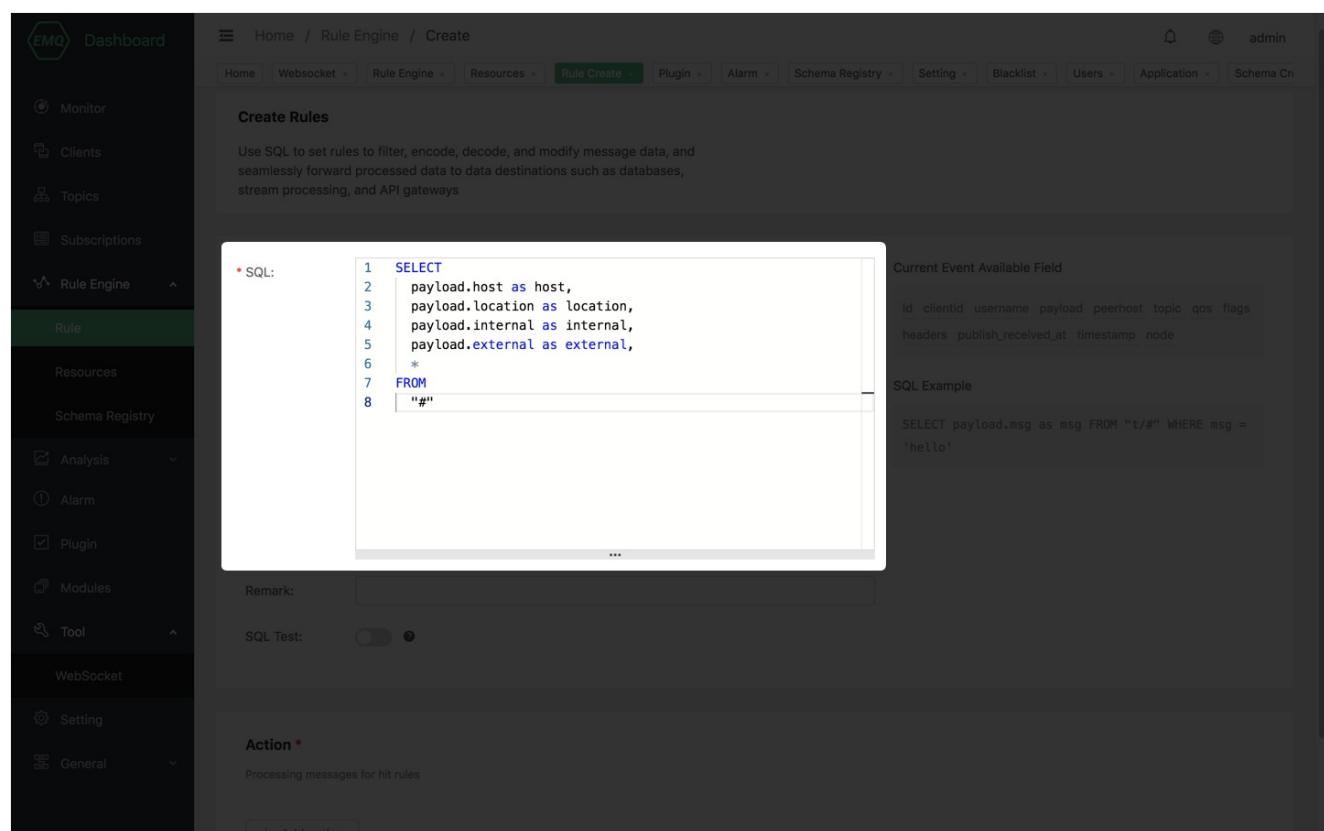
```
1 $ influxd -config /usr/local/etc/influxdb.conf
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

```
1 SELECT
2     payload.host as host,
3     payload.location as location,
4     payload.internal as internal,
5     payload.external as external,
6     *
7 FROM
8     "#"
```



关联动作：

在“Action”界面选择“Add action”，然后在“Action Type”下拉框里选择“Data to InfluxDB”。

1

2

填写动作参数：

“Data to InfluxDB” 动作有以下参数：

Action Type: Data to InfluxDB

Timestamp: Timestamp

Tags:

Key	Value	Add
		No Data

Fields:

Key	Value	Add
		No Data

Measurement:

Cancel Confirm

- 1). **Measurement**。指定写入到 InfluxDB 的 data point 的 Measurement，支持固定字符串和占位符两种设置方式。
- 2). **Fields**。指定写入到 InfluxDB 的 data point 的 Fields 的 Key 和 Value，支持固定字符串和占位符两种设置方式。
- 3). **Tags**。指定写入到 InfluxDB 的 data point 的 Tags 的 Key 和 Value，支持固定字符串和占位符两种设置方式。
- 4). **Timestamp Key**。指定写入到 InfluxDB 的 data point 的 Timestamp 的值从哪里获取。
- 5). **Use of resources**。指定动作关联的资源，现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 InfluxDB 资源，目前支持 HTTP/HTTPS 和 UDP 两种资源。

The screenshot shows the 'Create' dialog for a new rule. The 'SQL' section is expanded, showing a partial SQL query:

```

1 * SQL:
2   SELECT ...
3   FROM ...
4   WHERE ...
5   ...
6   ...
7   ...
8
  
```

The 'Types' dropdown is set to 'InfluxDB HTTP Service'. Below it, two options are listed: 'InfluxDB HTTP Service' (selected) and 'InfluxDB UDP Service'. A 'Test' button is visible.

The 'Action' section contains fields for 'Batch Size' (1000) and 'Pool Size' (8). There is also a 'Remark' field and a 'SQL Test' toggle switch.

On the right side of the dialog, there is a 'Current Event Available Field' table and a 'SQL Example' section with the query:

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

At the bottom of the dialog are 'Cancel' and 'Confirm' buttons.

填写资源配置：

InfluxDB HTTP 资源包括以下配置：

The screenshot shows the 'Create' dialog for a new rule. The 'SQL' section is expanded, showing a partial SQL query:

```

1 * SQL:
2   SELECT ...
3   FROM ...
4   WHERE ...
5   ...
6   ...
7   ...
8
  
```

The 'Types' dropdown is set to 'InfluxDB HTTP Service'. The 'Resource Name' input field is highlighted with a red border and contains the placeholder 'Please enter'.

The 'Action' section contains fields for 'Batch Size' (1000) and 'Pool Size' (8). There is also a 'Remark' field and a 'SQL Test' toggle switch.

On the right side of the dialog, there is a 'Current Event Available Field' table and a 'SQL Example' section with the query:

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

At the bottom of the dialog are 'Cancel' and 'Confirm' buttons.

1). Resource Name。 资源名称，支持以易读的形式唯一标识资源。

2). InfluxDB Host。 InfluxDB HTTP 主机地址。

3). InfluxDB Port。 InfluxDB HTTP 主机端口。

4). **InfluxDB Database。** InfluxDB 数据库名。

5). **InfluxDB Username。** InfluxDB 用户名。

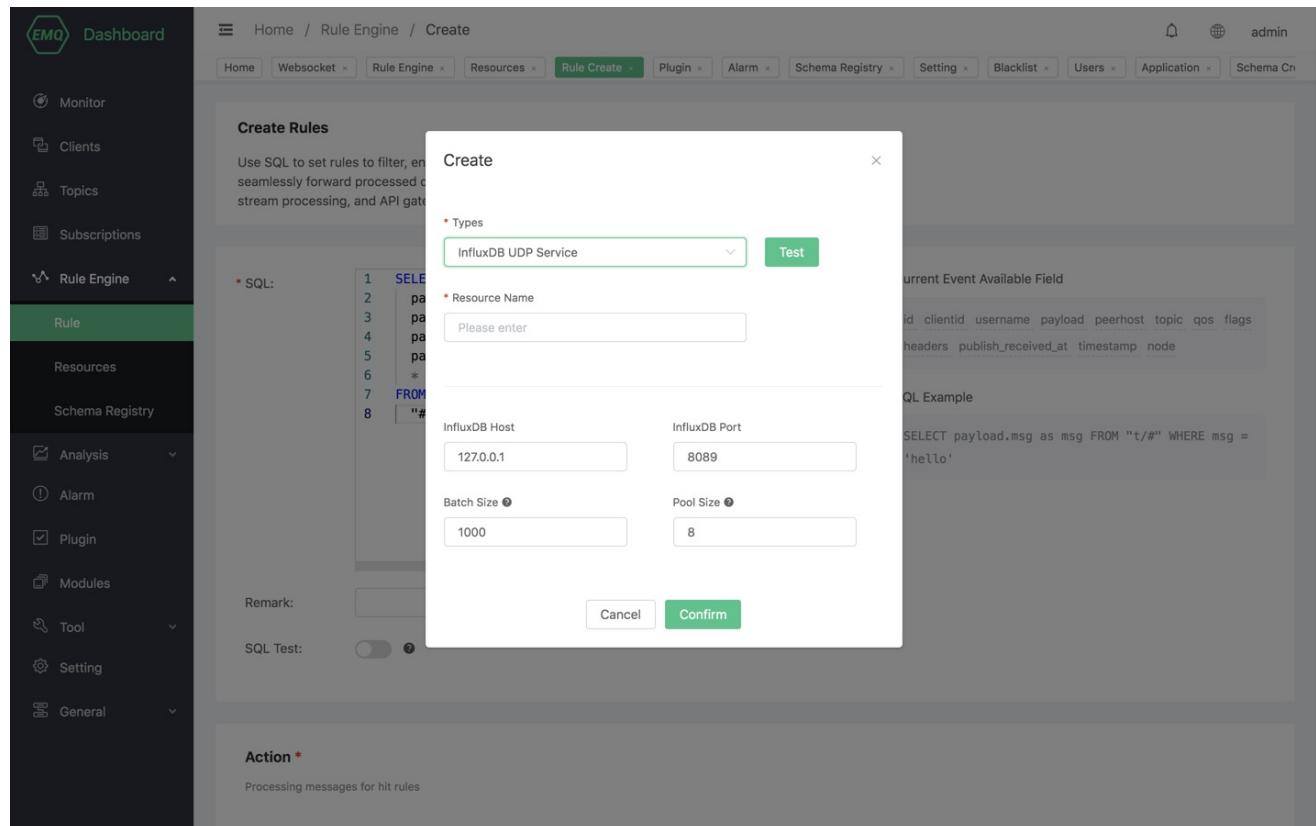
6). **InfluxDB Password。** InfluxDB 密码。

7). **Precision of timestamp。** 时间戳精度。

8). **Batch Size。** 单次写入能够收集的最大数据点数量，用于提升高并发时的性能。

9). **Pool Size。** InfluxDB 写进程池大小，合适的进程池大小可以在一定程度上提升写入性能。

InfluxDB UDP 资源包括以下配置：



1). **Resource Name。** 资源名称，支持以易读的形式唯一标识资源。

2). **InfluxDB Host。** InfluxDB UDP 主机地址。

3). **InfluxDB Port。** InfluxDB UDP 主机端口。

8). **Batch Size。** 单次写入能够收集的最大数据点数量，用于提升高并发时的性能。

9). **Pool Size。** InfluxDB 写进程池大小，合适的进程池大小可以在一定程度上提升写入性能。

本示例中所有配置保持默认值即可，点击“测试连接”按钮，确保连接测试成功。

最后点击“Confirm”按钮。

返回响应动作界面，选择刚刚创建的 InfluxDB 资源，填写其余配置后点击“Confirm”按钮。

最后返回规则创建界面，点击页面底部的“Create”按钮，完成规则创建。

The screenshot shows the EMQX Enterprise Rule Engine interface. On the left is a sidebar with various navigation options like Dashboard, Monitor, Clients, Topics, Subscriptions, Rule Engine (which is selected), Resources, Schema Registry, Analysis, Alarm, Plugin, Modules, Tool, WebSocket, Setting, and General. The main content area has a header "Home / Rule Engine" with tabs for Home, Websocket, and Rule Engine. Below the header is a brief description: "Use SQL to set rules to filter, encode, decode, and modify message data, and seamlessly forward processed data to data destinations such as databases, stream processing, and API gateways". A "Quick Start" button is also present. The central part of the screen displays a table with one row of data:

ID	Topic	Monitor	Description	Status	Action
rule:6d1e258f	#	full		<input checked="" type="checkbox"/>	Data to InfluxDB <button>Edit</button> <button>Delete</button>

规则已经创建完成，现在发一条消息：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload:
6  >{"host":"serverA","location":"roomA","internal":25,"external":37}

```

然后检查 InfluxDB，新的 **data point** 是否添加成功：

```

1 $ influx -precision rfc3339
2
3 > use db
4 Using database db
5 > select * from "temperature"
6 name: temperature
7 time           external  from          host      internal location
8 ----           -----  -----  -----  -----
9 1561535778444457348 37      mqttjs_46355e19 serverA 25      roomA

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

The screenshot shows the EMQX Enterprise V4.4 Rule Engine interface. On the left, a sidebar navigation includes: Dashboard, Monitor, Clients, Topics, Subscriptions, Rule Engine (selected), Resources, Schema Registry, Analysis, Alarm, Plugin, Modules, Tool, WebSocket, Setting, and General. The main content area has a breadcrumb navigation: Home / Rule Engine. Below this is a descriptive text: "Use SQL to set rules to filter, encode, decode, and modify message data, and seamlessly forward processed data to data destinations such as databases, stream processing, and API gateways". A "Create" button is present. A table lists a single rule:

ID	Topic	Monitor	Description	Status
rule:6d1e258f	#	[edit]		<input checked="" type="checkbox"/>

On the right, a panel titled "rule:6d1e258f" displays metrics and action metrics:

Metrics

- matched: 1 matched
- Current Speed: 0 matched/second
- Maximum speed: 0 matched/second
- Last 5 minutes speed: 0 matched/second

Action metrics

Action	Success	Fail
data_to_influxdb	1	0

数据保存到 InfluxDB V2 & InfluxDB Cloud

启动 **InfluxDB**，或者注册 **InfluxDB Cloud** 账号申请开通服务。本文中使用 **docker** 部署作为演示，并使用默认端口 **8086**，如有需要请自行替换成其他端口。

```
1 # docker
2 docker run -d -p 8086:8086 influxdb
```

创建资源

打开 [EMQX Dashboard](#)，选择左侧的“资源”选项卡，点击创建，资源类型选择“**InfluxDB HTTP V2 服务**”。资源包括以下配置：

- **InfluxDB 主机**：填写主机地址，或是 **InfluxDB Cloud** 中创建的服务地址；
- **InfluxDB 端口**：本地安装或者**docker**安装默认端口是**8086**，**InfluxDB Cloud** 使用**https**默认端口**443**；
- **InfluxDB Bucket**：数据库/数据集名称；
- **InfluxDB 组织名称**：创建服务时填写的组织名称；
- **InfluxDB Token**：对应数据库权限的 **API Token**，可以在 **InfluxDB** 控制台中找到；
- **时间戳精度**：默认使用毫秒；
- **进程池大小**：连接进程池大小，可根据业务并发量调节（在网速没有瓶颈的情况下，推荐每**1万**并发增加一个进程数量）；
- **启用 HTTPS**：根据安装与服务配置启用证书（**InfluxDB Cloud** 请打开此选项，但不需要配置额外证书）；

InfluxDB Cloud 获取组织名称示例（本地或 **docker** 部署，访问部署地址的**8086**端口即可进入控制台）：

The screenshot shows the 'Organization' profile page in the InfluxDB Cloud interface. On the left, there is a sidebar with icons for users, organizations, and other settings. The main area has tabs for 'USERS' and 'ABOUT'. Under 'Organization Profile', it shows the 'Name' field with 'er' and a 'RENAME' button. Below that is a section for 'Common IDs' with 'User ID' showing '08' and 'Organization ID' showing '35a'. Each ID has a 'COPY TO CLIPBOARD' button and a tooltip indicating the copied value ('huangdi@emqx.io | User ID' for User ID, 'emqx_test | Organization ID' for Organization ID). At the bottom, there is a 'Delete Organization' section with a 'DELETE' button.

InfluxDB Cloud 获取 **API Token** 示例（本地或 **docker** 部署，访问部署地址的**8086**端口即可进入控制台）：

The screenshot shows the 'API TOKENS' section of the EMQX Enterprise interface. It displays a single API token entry:

- Created at:** 2021-10-25 17:52:22
- Status:** Active (indicated by a blue switch)

创建资源：

The screenshot shows the 'Create Resource' dialog in the EMQX Dashboard. The configuration fields are as follows:

- 资源类型:** InfluxDB HTTP V2 服务
- 资源 ID:** resource:945258
- 描述:** 请输入
- InfluxDB 主机:** 127.0.0.1
- InfluxDB 端口:** 8086
- InfluxDB Bucket:** bucket
- InfluxDB 组织名称:** emqx
- InfluxDB Token:** uHi7tcN-wPyP39Fmw1Hl_C
- 时间戳精度:** ms
- 进程池大小:** 4
- 启用 HTTPS:** false

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

```

1  SELECT
2    payload.msg as msg,
3    clientid
4  FROM
5    "#"

```

EMQ

监控 / 规则引擎 / 创建

admin

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入:

```

1  SELECT
2    payload.msg as msg,
3    clientid
4  FROM
5    "#"

```

规则引擎是标准 MQTT 之上基于 SQL 的核心数据处理与分发组件，可以方便的筛选并处理 MQTT 消息与设备生命周期事件，并将数据分发移动到 HTTP Server、数据库、消息队列甚至是另一个 MQTT Broker 中。

1. 选择 't/#' 主题的消息，提取全部字段:

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

2. 通过事件主题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'
```

规则引擎和 SQL 语句的详细教程参见 [EMQ X 文档](#)。

* 规则 ID: rule:054380

描述:

SQL 测试:

取消

关联动作:

在 “Action” 界面选择 “Add action”，然后在 “Action Type” 下拉框里选择 “Data to InfluxDB”。

需要的字段:

- 启用批量插入：是否开启批量功能；
- 最大批量数：单个请求包含的数据最大条数；
- 最大批量间隔：批量消息最大间隔时间；
- Measurement: InfluxDB Measurement 单元；**
- Fields: 数据键值对 Fields；**
- Tags: 数据标签 Tags；**

最后点击“**Confirm**”按钮。

返回响应动作界面，选择刚刚创建的 **InfluxDB** 资源，填写其余配置后点击“**Confirm**”按钮。

规则已经创建完成，现在发一条消息：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload:
6 "hello"

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

The screenshot shows the EMQX Enterprise V4.4 Rule Engine interface. On the left sidebar, under the '规则引擎' (Rule Engine) section, there are tabs for '规则' (Rules), '资源' (Resources), and '编解码' (Coding/Decoding). The main content area displays a table with one row for the rule 'rule:737401'. The table columns are: ID, 主题 (Topic), 监控 (Monitoring), 描述 (Description), and 状态 (Status). The status column shows a green switch icon indicating the rule is active. To the right of the table, there are sections for '规则统计' (Rule Statistics) and '动作统计' (Action Statistics), both of which show minimal activity.

在InfluxDB控制台中查询结果：

The screenshot shows the InfluxDB Data Explorer interface. On the left, there is a sidebar with various icons. The main area has a title 'Data Explorer' and a message 'Now you can use Notebooks to explore and take action on your data Create a Notebook'. Below this is a table view showing a single row of data from a measurement named 'demo_m'. The table columns are: _start, _stop, _time, _value, _field, _measurement, and clientid. The data row is: 2021-12-07 01:41:17 GMT+8, 2021-12-07 02:41:17 GMT+8, 2021-12-07 02:40:47 GMT+8, hello, msg, demo_m, emqx_demo_1204. Below the table, the 'Query 1' editor shows the following InfluxQL query:

```

1 | from(bucket: "emqx_demo")
2 | |> range(start: v.timeRangeStart, stop: v.timeRangeStop)

```

The 'QUERY BUILDER' panel on the right contains sections for 'Transformations' (with 'aggregate.rate' and 'changeMomentumOscill...' listed) and 'Functions' (with 'columns' and 'cov' listed).

保存数据到 TimescaleDB

搭建 TimescaleDB 数据库环境，以 **MacOS X** 为例：

```

1 $ docker pull timescale/timescaledb
2
3 $ docker run -d --name timescaledb -p 5432:5432 -e POSTGRES_PASSWORD=password timescale/timescaledb:latest-pg11
4
5
6 $ docker exec -it timescaledb psql -U postgres
7
8 ## 创建并连接 tutorial 数据库
9 CREATE database tutorial;
10
11 \c tutorial
12
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
```

初始化 TimescaleDB 表：

```
1 $ docker exec -it timescaledb psql -U postgres -d tutorial
```

创建 `conditions` 表：

```

1 CREATE TABLE conditions (
2     time      TIMESTAMPTZ      NOT NULL,
3     location  TEXT            NOT NULL,
4     temperature DOUBLE PRECISION  NULL,
5     humidity   DOUBLE PRECISION  NULL
6 );
7
8 SELECT create_hypertable('conditions', 'time');
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

```

1 SELECT
2     payload.temp as temp,
3     payload.humidity as humidity,
4     payload.location as location
5 FROM
6     "#"
```

* SQL 输入:

```
1 SELECT
2
3   payload as p,
4   p.temp as temp,
5   p.humidity as humidity,
6   p.location as location
7
8 FROM
9   "#"
```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at timestamp node					

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

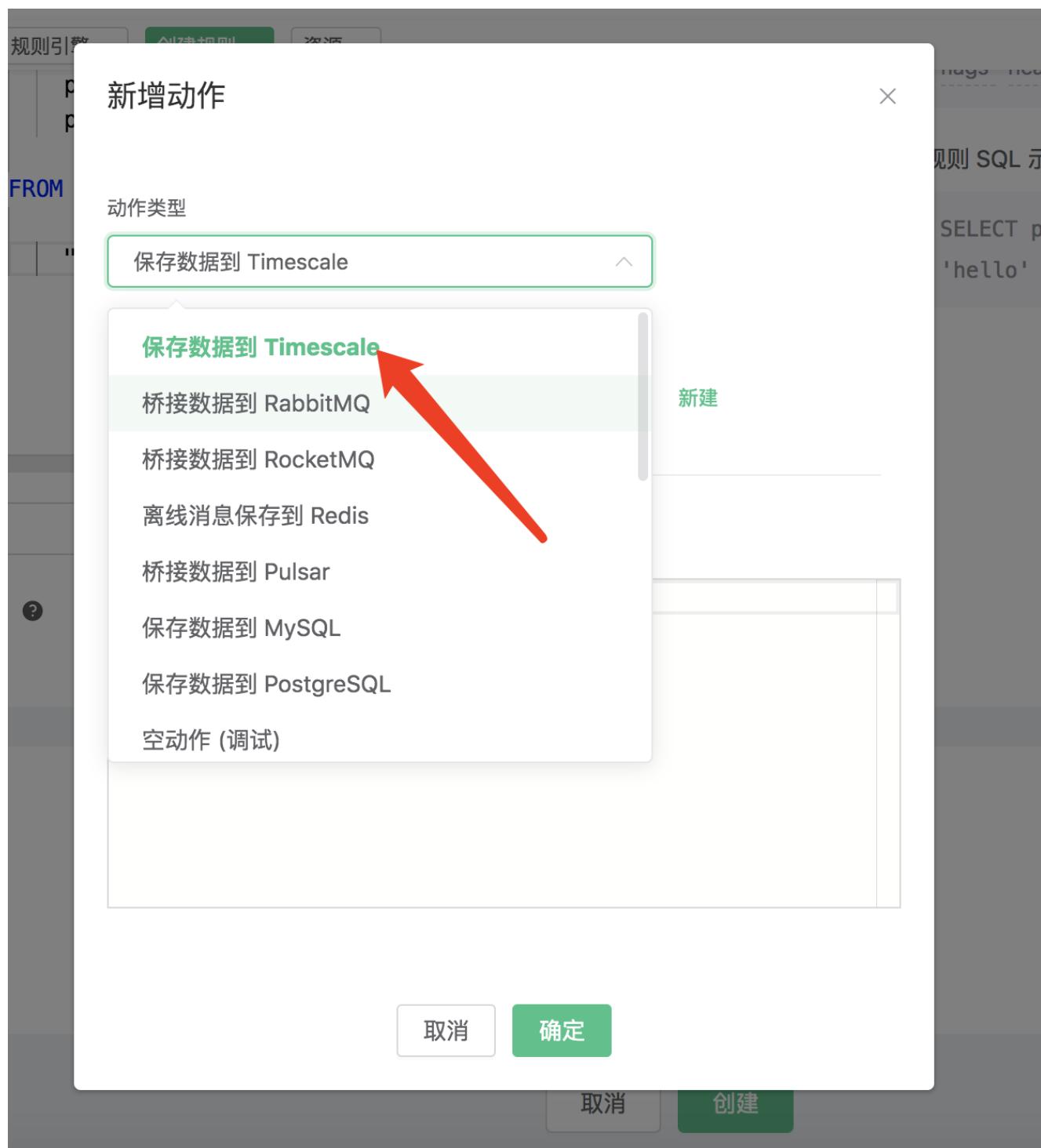
备注:

SQL 测试:



关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **TimescaleDB**”。



填写动作参数:

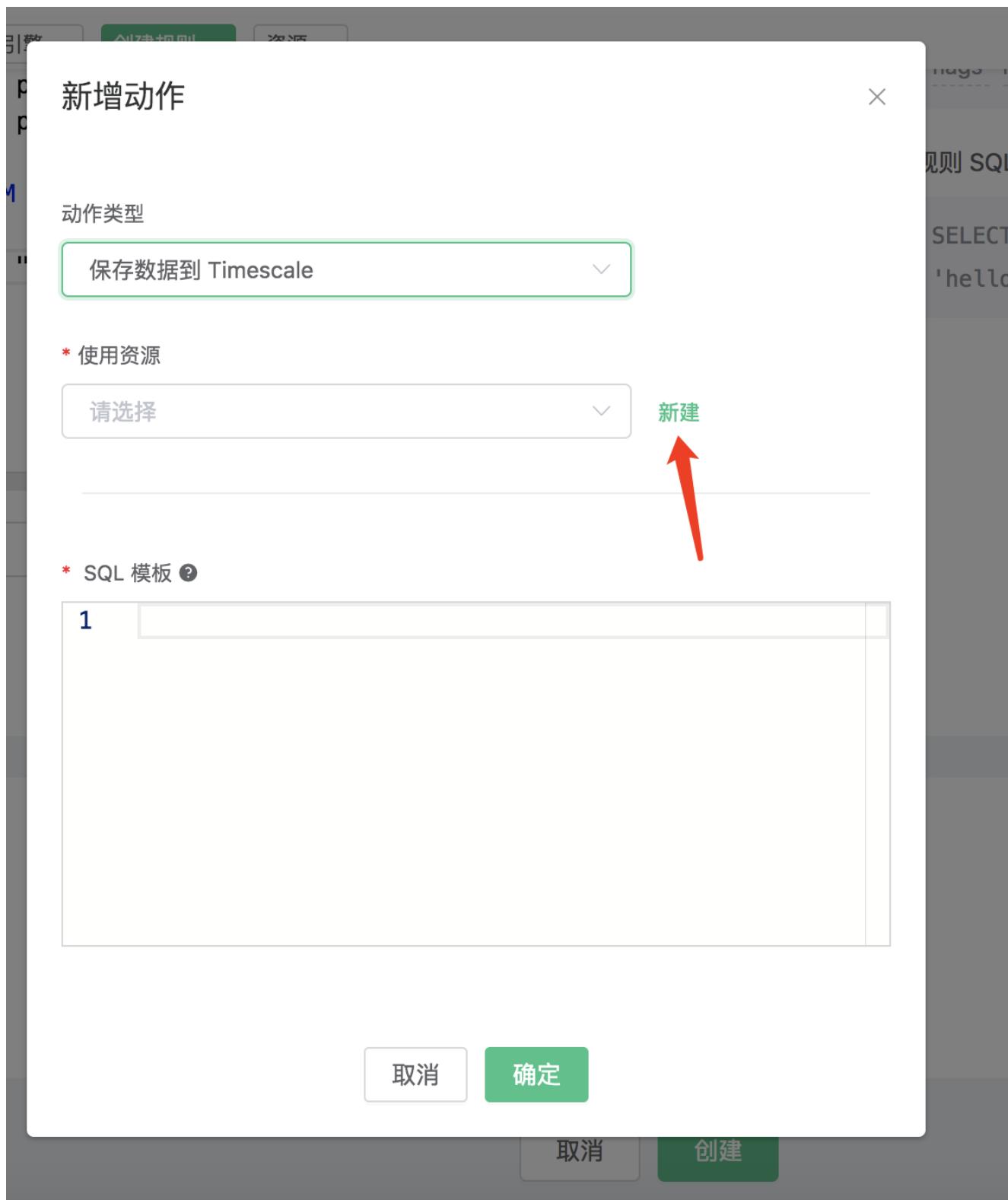
"保存数据到 **TimescaleDB**" 动作需要两个参数:

1). **SQL** 模板。这个例子里我们向 **TimescaleDB** 插入一条数据, **SQL** 模板为:

```
1 insert into conditions(time, location, temperature, humidity) values (NOW(), ${location}, ${temp}, ${humidity})
```

插入数据之前, **SQL** 模板里的 **\${key}** 占位符会被替换为相应的值。

2). 关联资源。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 **TimescaleDB** 资源:



选择 “PostgreSQL 资源”。

填写资源配置：

数据库名填写 “mqtt”，用户名填写 “root”，其他配置保持默认值，然后点击 “测试连接” 按钮，确保连接测试成功。

最后点击 “新建” 按钮。

规则引擎

创建资源

* 资源类型

PostgreSQL

测试连接

* 资源名称

PgSQL

* PostgreSQL 服务器

127.0.0.1:5432

* PostgreSQL 数据库名称

mqtt

* PostgreSQL 用户名

root

PostgreSQL 密码

PostgreSQL 密码

是否重连

true

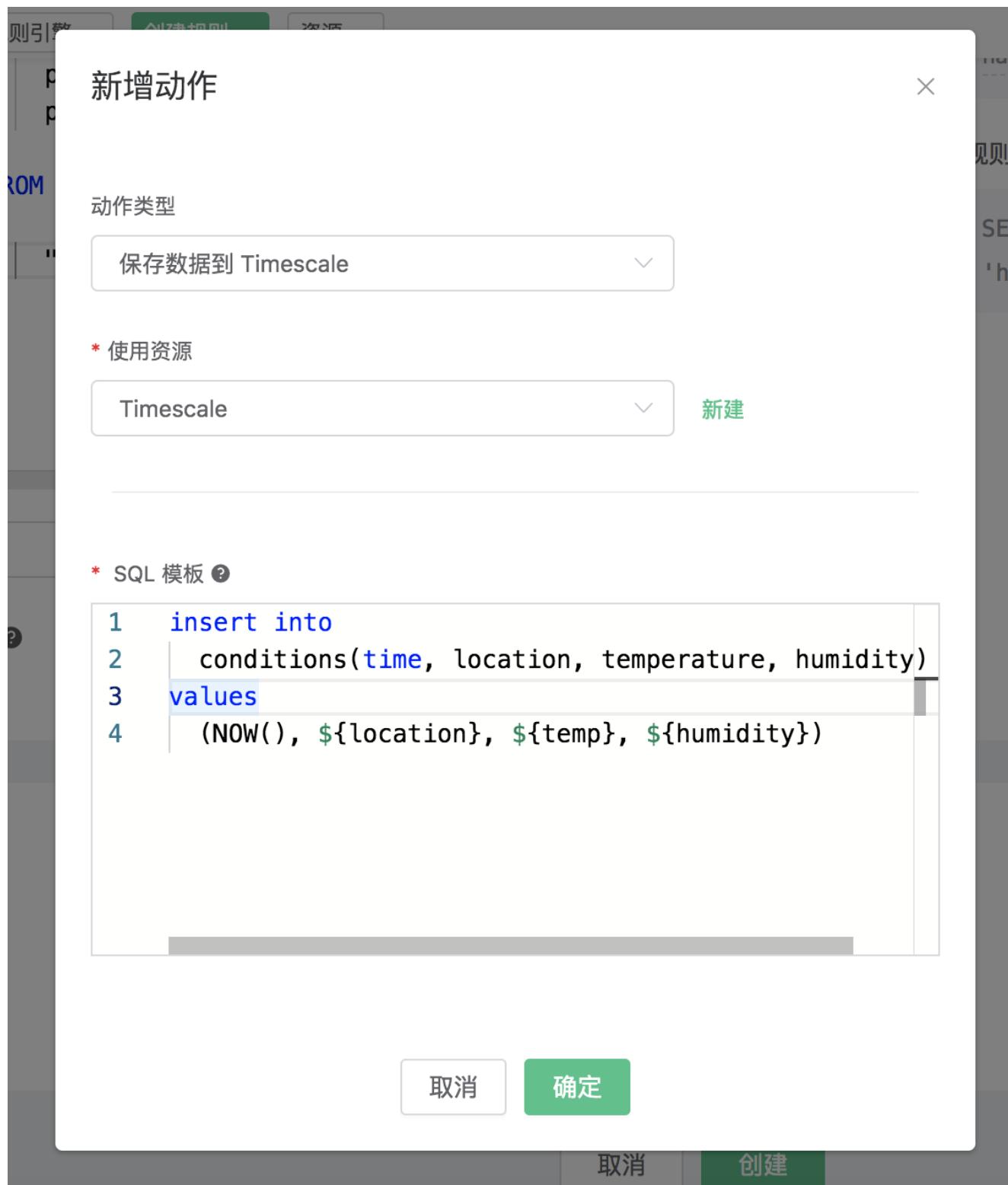
连接池大小

8

取消 确定

取消 创建

返回响应动作界面，点击“确认”。



返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 保存数据到 Timescale (data_to_timescale)
保存数据到 Timescale

SQL 模板 insert into conditions(time, location, temperature, humidity) values (NOW(), \${location}, \${temp}, \${humidity})
资源 ID: resource:958426ed

[编辑](#) [移除](#)

[+ 失败备选动作](#)

[+ 添加动作](#)

[取消](#) [创建](#)

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "{\"temp\":24,\"humidity\":30,\"location\":\"hangzhou\"}"

```

然后检查 TimescaleDB 表，新的 record 是否添加成功：

```

1 tutorial=# SELECT * FROM conditions LIMIT 100;
2           time | location | temperature | humidity
3 -----
4 2019-06-27 01:41:08.752103+00 | hangzhou |          24 |         30

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

[+ 创建](#)

ID	主题	监控	描述	状态	响应动作
rule:546273fb	#			<input checked="" type="checkbox"/>	保存数据到 Timescale

保存数据到 Oracle DB

创建 `t_mqtt_msg` 表:

```
1 CREATE TABLE t_mqtt_msg (msgid VARCHAR2(64),topic VARCHAR2(255), qos NUMBER(1), payload NCLOB )
```

```
SQL> CREATE TABLE t_mqtt_msg (msgid VARCHAR2(64),topic VARCHAR2(255), qos NUMBER(1), payload NCLOB);
```

Table created.

```
SQL> DESCRIBE t_mqtt_msg;
```

Name	Null?	Type
MSGID		VARCHAR2(64)
TOPIC		VARCHAR2(255)
QOS		NUMBER(1)
PAYLOAD		NCLOB

```
SQL> █
```

创建规则:

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL:

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

* SQL 输入:

```
1 SELECT
2 *
3
4
5 FROM
6
7 "t/#"
```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at	timestamp	node			

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

备注:

SQL 测试:



?

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 Oracle Database”。

新增动作

X

* 动作类型

数据持久化

保存数据到 Oracle Database

* 使用资源

请选择

* SQL 模板 ?

1

保存数据到 ClickHouse

保存数据到 DolphinDB

保存数据到 DynamoDB

保存数据到 InfluxDB

保存数据到 MongoDB

保存数据到 MySQL

保存数据到 OpenTSDB

新建

保存数据到 Oracle Database

取消

确定

填写动作参数:

“保存数据到 Oracle Database” 动作需要两个参数:

1). SQL 模板。这个例子里我们向 Oracle Database 插入一条数据, SQL 模板为:

```
1      INSERT INTO T_MQTT_MSG (MSGID, TOPIC, QOS, PAYLOAD) values ('${id}', '${topic}', '${qos}', '${payload}');
```

2). 关联资源的 ID。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 Oracle Database 资源:

填写资源配置:

创建资源

X

* 资源类型

Oracle Database

测试连接

* 资源 ID

resource:569883

描述

请输入

* Oracle Database 服务器

127.0.0.1:1521

连接池大小

8

* Oracle Database Sid

ORCL

* Oracle Database 用户

admin

* Oracle Database 密码

system

批量写入大小

100

批量写入间隔(毫秒) ②

10

同步或者异步插入

sync

是否重连 ②

true

折叠 ^

取消

确定

453 / 1099

点击“新建”按钮。

返回响应动作界面，点击“确认”。

编辑动作

×

* 动作类型

数据持久化

保存数据到 Oracle Database

* 使用资源

resource:963628

新建

* SQL 模板 ②

```
1  INSERT INTO
2    T_MQTT_MSG (MSGID, TOPIC, QOS, PAYLOAD)
3  values
4    ('${id}', '${topic}', '${qos}', '${payload}');
```

取消

确定

返回规则创建界面，点击“创建”。

响应动作

处理命中规则的消息

The screenshot shows the 'Response Actions' section of a rule configuration. It includes a table with one row, a 'Add Action' button, and a confirmation dialog box.

动作类型	保存数据到 Oracle Database (data_to_oracle)
SQL 模板	INSERT INTO T_MQTT_MSG (MSGID, TOPIC, QOS, PAYLOAD) values ('\${id}', '\${topic}', '\${qos}', '\${payload}'); 资源 ID resource:963628

+ 失败备选动作

+ 添加动作

取消 确定

在规则列表里，点击“查看”按钮或规则 ID 连接，可以预览刚才创建的规则：

基本信息

主题: 消息发布 (\$events/message_publish)

备注:

查询字段: *

筛选条件:

规则 SQL:

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

响应动作

命中规则的消息处理方式

动作类型	保存数据到 Oracle Database (data_to_oracle)
SQL 模板	INSERT INTO T_MQTT_MSG (MSGID, TOPIC, QOS, PAYLOAD) values ('\${id}', '\${topic}', '\${qos}', '\${payload}'); 资源 ID resource:963628

成功 0 失败 0

规则已经创建完成，现在发一条数据：

1	Topic: "t/a"	sh
2	QoS: 1	
3	Payload: "hello"	

查看规则命中次数

运行统计

命中次数

1 次

规则启用后的执行次数

当前速度

0.1 次/秒

最近5分钟执行速度

0.01 次/秒

最近5分钟平均执行速度

折叠详情 ^

节点	命中次数	当前速度	最大执行速度	最近5分钟执行速度
emqx@127.0.0.1	1	0.1	0.1	0.01

保存数据到 SQLServer

搭建 **SQLServer** 数据库，并设置用户名密码为 **sa/mqtt_public**，以 **MacOS X** 为例：

```
1 docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD= mqtt_public' -p 1433:1433 -d mcr.microsoft.com  
/mssql/server:2017-latest
```

进入**SQLServer**容器， 初始化 **SQLServer** 表：

```
1 $ /opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P mqtt_public -d master  
2 $ mysql -u root -h localhost -ppublic
```

创建“**test**”数据库：

```
1 CREATE DATABASE test;  
2 go;
```

创建 **t_mqtt_msg** 表：

```
1 USE test;  
2 go;  
3 CREATE TABLE t_mqtt_msg (id int PRIMARY KEY IDENTITY(1000000001,1) NOT NULL,  
4                             msgid  VARCHAR(64) NULL,  
5                             topic   VARCHAR(100) NULL,  
6                             qos    tinyint NOT NULL DEFAULT 0,  
7                             payload NVARCHAR(100) NULL,  
8                             arrived DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP);  
9 go;
```

Mac上配置 **odbc** 驱动：

```
1 $ brew install unixodbc freetds  
2 $ vim /usr/local/etc/odbcinst.ini  
3  
4 [ms-sql]  
5 Description = ODBC for FreeTDS  
6 Driver      = /usr/local/lib/libtdsodbc.so  
7 Setup       = /usr/local/lib/libtdsodbc.so  
8 FileUsage   = 1
```

Centos上配置 **odbc** 驱动：

```

1 $ yum install unixODBC unixODBC-devel freetds freetds-devel perl-DBD-ODBC perl-local-lib
2 $ vim /etc/odbcinst.ini
3 # 加入以下内容
4 [ms-sql]
5 Description = ODBC for FreeTDS
6 Driver      = /usr/lib64/libtdsodbc.so
7 Setup       = /usr/lib64/libtdsS.so.2
8 Driver64    = /usr/lib64/libtdsodbc.so
9 Setup64     = /usr/lib64/libtdsS.so.2
10 FileUsage   = 1

```

Ubuntu上配置 odbc 驱动（以 Ubuntu20.04 为例，其他版本请参考 [odbc 官方文档](#)）：

```

1 $ apt-get install unixodbc unixodbc-dev tdsodbc freetds-bin freetds-common freetds-dev libdbd-odbc-perl liblocal-lib-perl
2 $ vim /etc/odbcinst.ini
3 # 加入以下内容
4 [ms-sql]
5 Description = ODBC for FreeTDS
6 Driver      = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
7 Setup       = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so
8 FileUsage   = 1

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

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

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入：

```

1 SELECT
2 *
3
4
5 FROM
6
7 "t/#"
8

```

当前事件可用字段

id clientid username payload peerhost topic qos flags
headers publish_received_at timestamp node

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

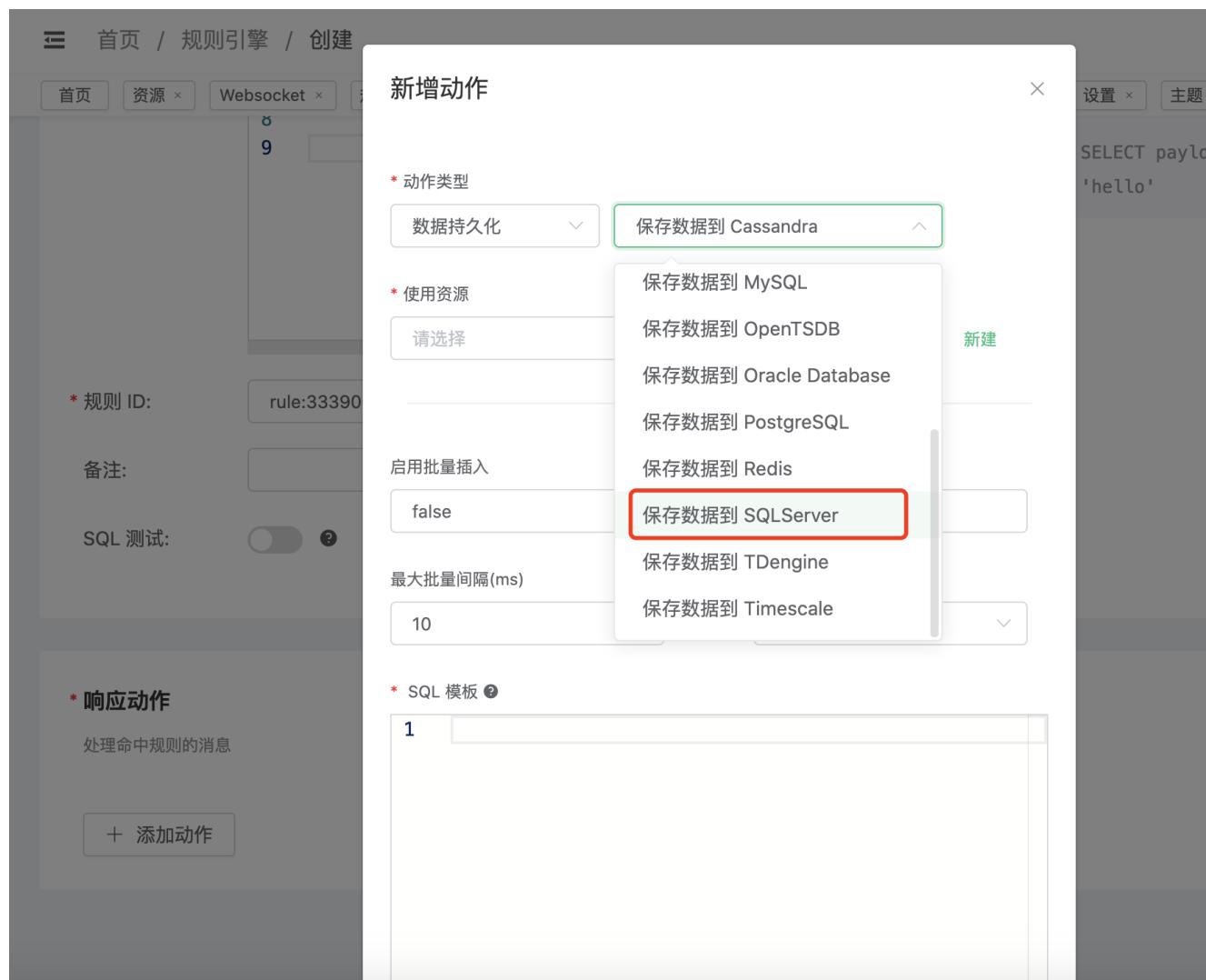
* 规则 ID：

rule:333906

备注：

关联动作：

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **SQLServer**”。



填写动作参数：

“保存数据到 **SQLServer**” 动作需要两个参数： **4 2). SQL 模板**。这个例子里我们向 **SQLServer** 插入一条数据，**SQL** 模板为：

```
1 insert into t_mqtt_msgmsgid, topic, qos, payload) values ('${id}', '${topic}', ${qos}, '${payload}')
```

新增动作

* 动作类型

数据持久化

保存数据到 SQLServer

* 使用资源

resource:297250

[新建](#)

启用批量插入

false

最大批量数

100

最大批量间隔(ms)

10

同步或者异步插入

sync

* SQL 模板

```
1 insert into t_mqtt_msgmsgid, topic, qos, payload)
2 values ('${id}', '${topic}', ${qos}, '${payload}')
```

1). 关联资源的 ID。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **SQLServer** 资源：

填写资源配置：数据库名填写 “mqtt”，用户名填写 “sa”，密码填写 “mqtt_public”

创建资源

* 资源类型

Microsoft SQL Server 测试连接

* 资源 ID

resource:297250

描述

请输入

* Microsoft SQL 服务器 ?

192.168.1.172:1433

* SQL Server 数据库名

mqtt

连接池大小

8

* SQL Server 用户名

sa

* SQL Server 密码

..... 眼睛图标

驱动名称

ms-sql

取消 确定

取消 创建

点击“新建”按钮。

返回响应动作界面，点击“确认”。

响应动作

处理命中规则的消息

```
动作类型 保存数据到 SQLServer (data_to_sq|server)
保存数据到 SQLServer 数据库
启用批量插入 false 最大批量数 100 最大批量间隔(ms) 10 同步或者异步插入 sync
SQL 模板 insert into t_mqtt_msg(msgid, topic, qos, payload) values ('${id}', '${topic}', ${qos}, '${payload}') 资源 ID resource:297250
```

[编辑](#) [移除](#)

+ 失败备选动作

+ 添加动作

[取消](#) [创建](#)

返回规则创建界面，点击“创建”。

基本信息

主题: 消息发布 (\$events/message_publish)

备注:

查询字段: *

筛选条件:

规则 SQL:

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

响应动作

命中规则的消息处理方式

```
动作类型 保存数据到 SQLServer (data_to_sq|server)
保存数据到 SQLServer 数据库
详细统计 点击查看 SQL 模板 insert into t_mqtt_msg(msgid, topic, qos, payload) values ('${id}', '${topic}', ${qos}, '${payload}')
同步或者异步插入 sync 启用批量插入、false 最大批量数 100 最大批量间隔(ms) 10 资源 ID resource:297250
```

成功 0 失败 0

在规则列表里，点击“查看”按钮或规则 ID 连接，可以预览刚才创建的规则：

rule:009b7125

[删除](#)

运行统计

命中次数

1 次

规则启用后的执行次数

当前速度

0.1 次/秒

最大执行速度: 0.1 次/秒

最近5分钟执行速度

0.01 次/秒

最近5分钟平均执行速度

详细信息

规则已经创建完成，现在发一条数据：

1	Topic: "t/a"	sh
2	QoS: 1	
3	Payload: "hello"	

然后检查 **SQLServer** 表，新的 **record** 是否添加成功：

```
1> select * from t_mqtt_msg;
2> go
id      msgid          topic           qos payload
-----+-----+-----+-----+
1000000001 5B4FAEE0C6692F44000000BA80001   t/1            1 hello
                                                arrived        2020-11-26 04:25:12.227
(1 rows affected)
1>
```

保存数据到 **DolphinDB**

DolphinDB 是由浙江智臾科技有限公司研发的一款高性能分布式时序数据库，集成了功能强大的编程语言和高容量高速度的流数据分析系统，为海量结构化数据的快速存储、检索、分析及计算提供一站式解决方案，适用于量化金融及工业物联网等领域。

EMQX 用 **Erlang** 实现了 **DolphinDB** 的客户端 **API**，它通过 **TCP** 的方式将数据传输到 **DolphinDB** 进行存储。

搭建 **DolphinDB**

目前，**EMQX** 仅适配 **DolphinDB 1.20.7** 的版本。

以 **Linux** 版本为例，前往官网下载社区最新版本的 **Linux64** 安装包：<https://www.dolphindb.cn/downloads.html>

将安装包的 **server** 目录上传至服务器目录 `/opts/app/dolphindb`，并测试启动是否正常：

```

1 chmod +x ./dolphindb
2 ./dolphindb
3
4 ## 启动成功后，会进入到 dolphindb 命令行，执行 1+1
5 >1+1
6 2

```

启动成功，并得到正确输出，表示成功安装 **DolphinDB**。然后使用 `<CRTL+D>` 关闭 **DolphinDB**。

现在，我们需要打开 **DolphinDB** 的 **StreamTable** 的发布/订阅的功能，并创建相关数据表，以实现 **EMQX** 消息存储并持久化的功能：

1. 修改 **DolphinDB** 的配置文件 `vim dolphindb.cfg` 加入以下配置项，以打开 发布/订阅 的功能：

```

1 ## Publisher for streaming
2 maxPubConnections=10
3 persistenceDir=/ddb/pubdata/
4 #persistenceWorkerNum=
5 #maxPersistenceQueueDepth=
6 #maxMsgNumPerBlock=
7 #maxPubQueueDepthPerSite=
8
9 ## Subscriber for streaming
10 subPort=8000
11 #subExecutors=
12 #maxSubConnections=
13 #subExecutorPooling=
14 #maxSubQueueDepth=

```

2. 后台启动 **DolphinDB** 服务：

```

1 ## 启动完成后，DolphinDB 会监听 8848 端口供客户端使用。
2 nohup ./dolphindb -console 0 &

```

3. 前往 **DolphinDB** 官网，下载合适的 **GUI** 客户端连接 **DolphinDB** 服务：

- 前往 [下载页](#) 下载 **DolphinDB GUI**
- **DolphinDB GUI** 客户端依赖 **Java** 环境，先确保已经安装 **Java**
- 前往 **DolphinDB GUI** 目录中执行 `sh gui.sh` 启动客户端
- 在客户端中添加 **Server** 并创建一个 **Project**，和脚本文件。

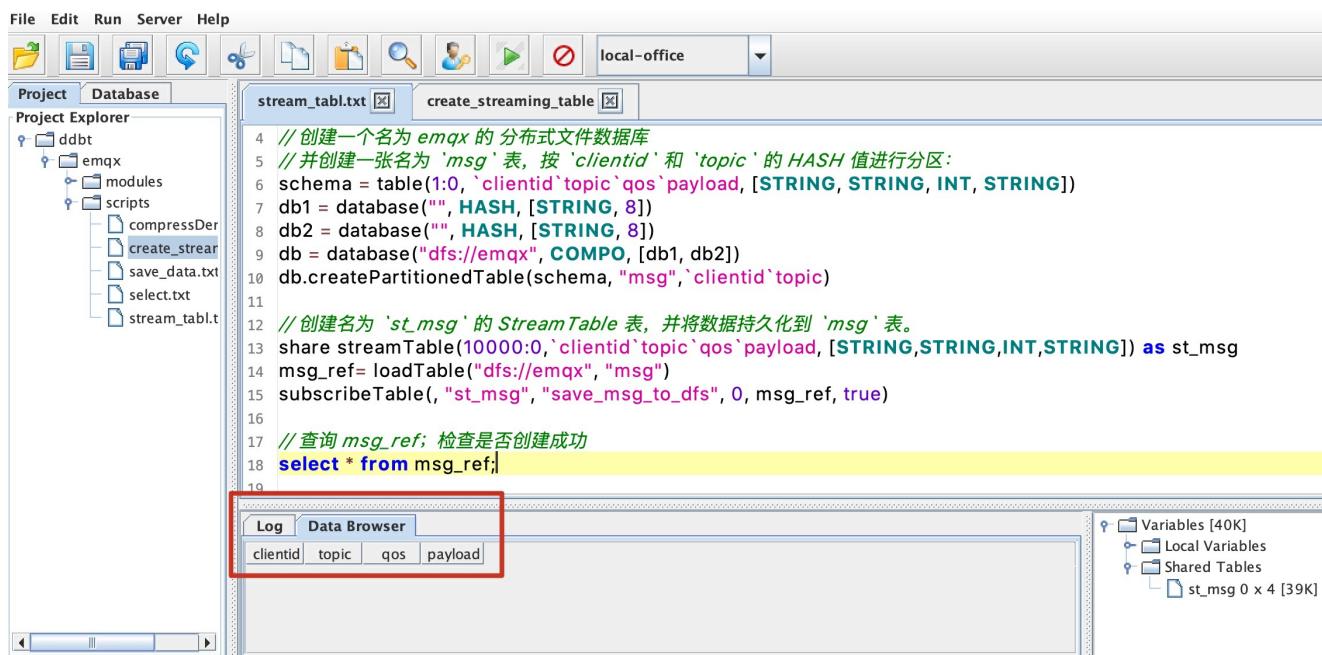
4. 创建分布式数据库，和 **StreamTable** 表；并将 **StreamTable** 的数据持久化到分布式表中：

```

1 // 创建一个名为 emqx 的 分布式文件数据库
2 // 并创建一张名为 `msg` 表，按 `clientid` 和 `topic` 的 HASH 值进行分区：
3 schema = table(1:0, `clientid` `topic` `qos` `payload, [STRING, STRING, INT, STRING])
4 db1 = database("", HASH, [STRING, 8])
5 db2 = database("", HASH, [STRING, 8])
6 db = database("dfs://emqx", COMPO, [db1, db2])
7 db.createPartitionedTable(schema, "msg", `clientid` `topic)
8
9 // 创建名为 `st_msg` 的 StreamTable 表，并将数据持久化到 `msg` 表。
10 share streamTable(10000:0, `clientid` `topic` `qos` `payload, [STRING, STRING, INT, STRING]) as st_msg
11 msg_ref= loadTable("dfs://emqx", "msg")
12 subscribeTable(, "st_msg", "save_msg_to_dfs", 0, msg_ref, true)
13
14 // 查询 msg_ref; 检查是否创建成功
15 select * from msg_ref;

```

完成后，可以看到一张空的 `msg_ref` 已创建成功：



至此，**DolphinDB** 的配置已经完成了。

详细的 **DolphinDB** 使用文档请参考：

- 用户指南：https://github.com/dolphindb/Tutorials_CN/blob/master/dolphindb_user_guide.md
- IoT 场景示例：https://gitee.com/dolphindb/Tutorials_CN/blob/master/iot_examples.md
- 流处理指南：https://github.com/dolphindb/Tutorials_CN/blob/master/streamingTutorial.md
- 编程手册：<https://www.dolphindb.cn/cn/help/index.html>

配置规则引擎

创建规则:

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**:

The screenshot shows the EMQX Rule configuration interface. On the left, there is a code editor window with the following SQL query:

```
1 * SQL 输入:  
2 SELECT * FROM "t/#"  
3  
4  
5  
6  
7  
8
```

To the right of the code editor, there are two panels: "当前事件可用字段" (Available fields for current event) and "规则 SQL 示例" (Rule SQL example). The "当前事件可用字段" panel lists various event fields. The "规则 SQL 示例" panel shows a sample rule SQL query.

Below the code editor, there is a "备注:" (Remarks) input field and a "SQL 测试:" (SQL Test) toggle switch.

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **DolphinDB**”。

新增动作

* 动作类型

数据持久化

保存数据到 **DolphinDB**

* 使用资源

请选择

启用批量插入

false

* SQL 模板

1

保存数据到 Cassandra

保存数据到 ClickHouse

新建

保存数据到 DolphinDB

保存数据到 DynamoDB

保存数据到 InfluxDB

保存数据到 MongoDB

保存数据到 MySQL

保存数据到 OpenTSDB

填写动作参数:

“保存数据到 **DolphinDB**” 动作需要两个参数:

1). **SQL** 模板。这个例子里我们向流表 `st_msg` 中插入一条数据, **SQL** 模板为:

```
1   insert into st_msg values('${clientid}', '${topic}', ${qos}, '${payload}')
```

2). 关联资源的 **ID**。现在资源下拉框为空, 可以点击右上角的 “新建资源” 来创建一个**DolphinDB** 资源:

填写资源配置:

服务器地址填写对应上文部署的 **DolphinDB** 的服务器, 用户名为 `admin` 密码为 `123456`

创建资源

X

*** 资源类型**

DolphinDB

测试连接

*** 资源 ID**

resource:419372

描述

请输入

*** DolphinDB 服务器**

192.168.1.172:8848

连接池大小

8

^
v*** DolphinDB 用户名**

admin

DolphinDB 密码

123456

是否重连

true

取消

确定

点击“确定”按钮。

返回响应动作界面，点击“确定”。

新增动作

X

*** 动作类型**

数据持久化

保存数据到 DolphinDB

*** 使用资源** ?

resource:419372

新建

启用批量插入

true

最大批量数 ?

100

最大批量间隔(ms)

10

同步或者异步插入

async

调用超时时间(ms)

5000

*** SQL 模板** ?

```
1   insert into st_msg values('${clientid}', '${topic}',  
2   |
```

取消**确定**

返回规则创建界面，点击“创建”。

• 响应动作

处理命中规则的消息

动作类型 保存数据到 DolphinDB (data_to_dolphindb)
保存数据到 DolphinDB 数据库
SQL 模板 insert into st_msg values(\${clientid}, \${topic}, \${qos}, \${payload}) 资源 ID resource:454997

+ 添加动作 取消 创建

在规则列表里，点击“查看”按钮或规则 ID 连接，可以预览刚才创建的规则：

rule:a0fb8eb

运行统计

命中次数 0 次

当前速度 0 次/秒

最近5分钟执行速度 0 次/秒

详细信息

基本信息

主题: 消息发布 (\$events/message_publish)

备注:

查询字段:

筛选条件:

规则 SQL:

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

响应动作

命中规则的消息处理方式

动作类型 保存数据到 DolphinDB (data_to_dolphindb)
保存数据到 DolphinDB 数据库
详细统计 点击查看 SQL 模板 insert into st_msg values(\${clientid}, \${topic}, \${qos}, \${payload}) 资源 ID resource:941580 成功 0 失败 0

规则已经创建完成，现在发一条数据：

```
1 Topic: "t/a"
2 QoS: 1
3 Payload: "hello"
```

然后检查持久化的 `msg_dfs` 表，新的数据是否添加成功：

File Edit Run Server Help

Project Database

Project Explorer

- ddbt
 - emqx
 - modules
 - scripts
 - compressDer
 - create_strear
 - save_data.txt
 - select.txt
 - stream_table.t

stream_table.txt create_streaming_table select.txt

```
1 msg_ref=loadTable("dfs://emqx", "msg")
2
3 select * from msg_ref
```

Log Data Browser

clientid	topic	qos	payload
rule_tester	t/a	1	hello

Variables [40K]
 Local Variables
 Shared Tables
 st_msg 1 x 4 [39K]

保存数据到 MatrixDB

搭建 **MatrixDB** 数据库，设置用户名密码为 **root/public**，并创建一个名为 **mqtt** 的数据库。

通过命令行工具 **psql** 访问 **MatrixDB** 并创建 `t_mqtt_msg` 表：

```

1 $ psql -h localhost -U root mqtt
sh

1 CREATE TABLE t_mqtt_msg (
2     id SERIAL primary key,
3     msgid character varying(64),
4     sender character varying(64),
5     topic character varying(255),
6     qos integer,
7     payload text,
8     arrived timestamp without time zone
9 );

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

```

1 SELECT * FROM "t/#"
sh

```

EMQ X Enterprise

监控 / 规则引擎 / 创建

admin

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入:

```

1 SELECT
2 *
3
4 FROM
5
6
7 "t/#"

```

规则引擎是标准 MQTT 之上基于 SQL 的核心数据处理与分发组件，可以方便的筛选并处理 MQTT 消息与设备生命周期事件，并将数据分发移动到 HTTP Server、数据库、消息队列甚至是另一个 MQTT Broker 中。

- 选择 't/#' 主题的消息，提取全部字段:

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

- 通过事件主题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'
```

规则引擎和 SQL 语句的详细教程参见 [EMQ X 文档](#)。

* 规则 ID: rule:819548

描述:

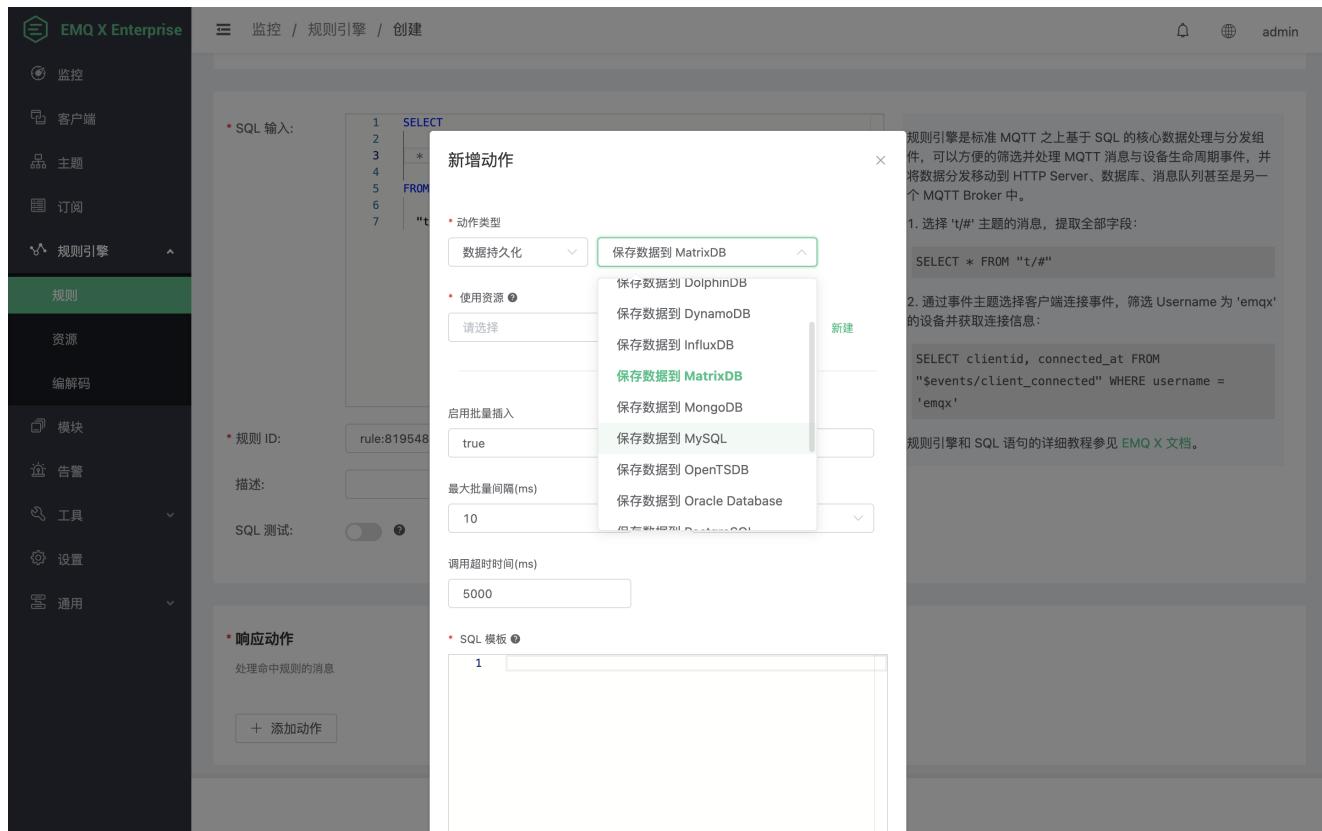
SQL 测试:

* 响应动作

取消 创建

关联动作：

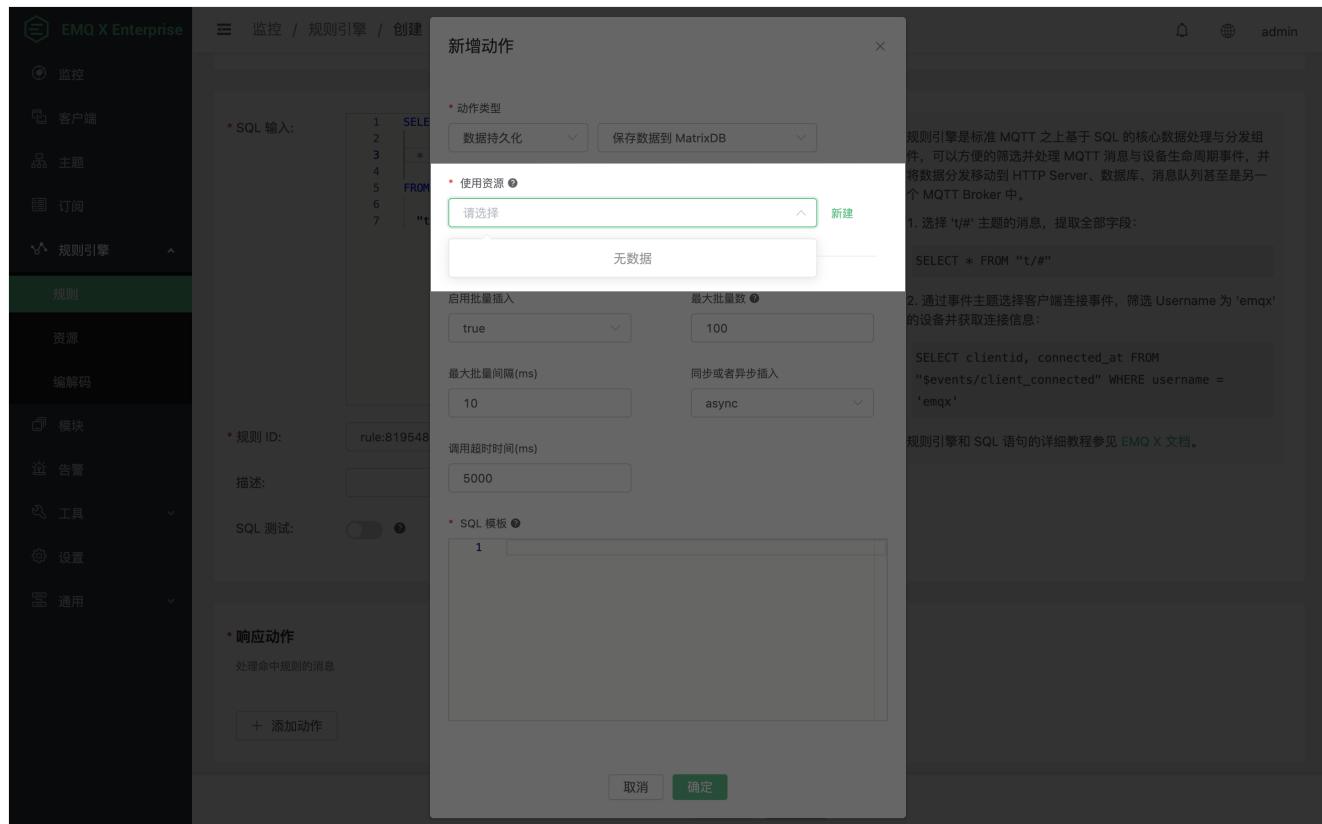
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 MatrixDB”。



填写动作参数:

“保存数据到 MatrixDB” 动作需要以下参数:

1). 使用资源，即资源 ID，现在资源下拉框为空，需要先创建一个可用的 MatrixDB 资源实例。



点击 使用资源 右侧的新建按钮，进入 创建资源 页面，MatrixDB 资源提供了以下可配置项：

服务器, **MatrixDB** 的服务器地址。

数据库名称, **MatrixDB** 数据库名称。

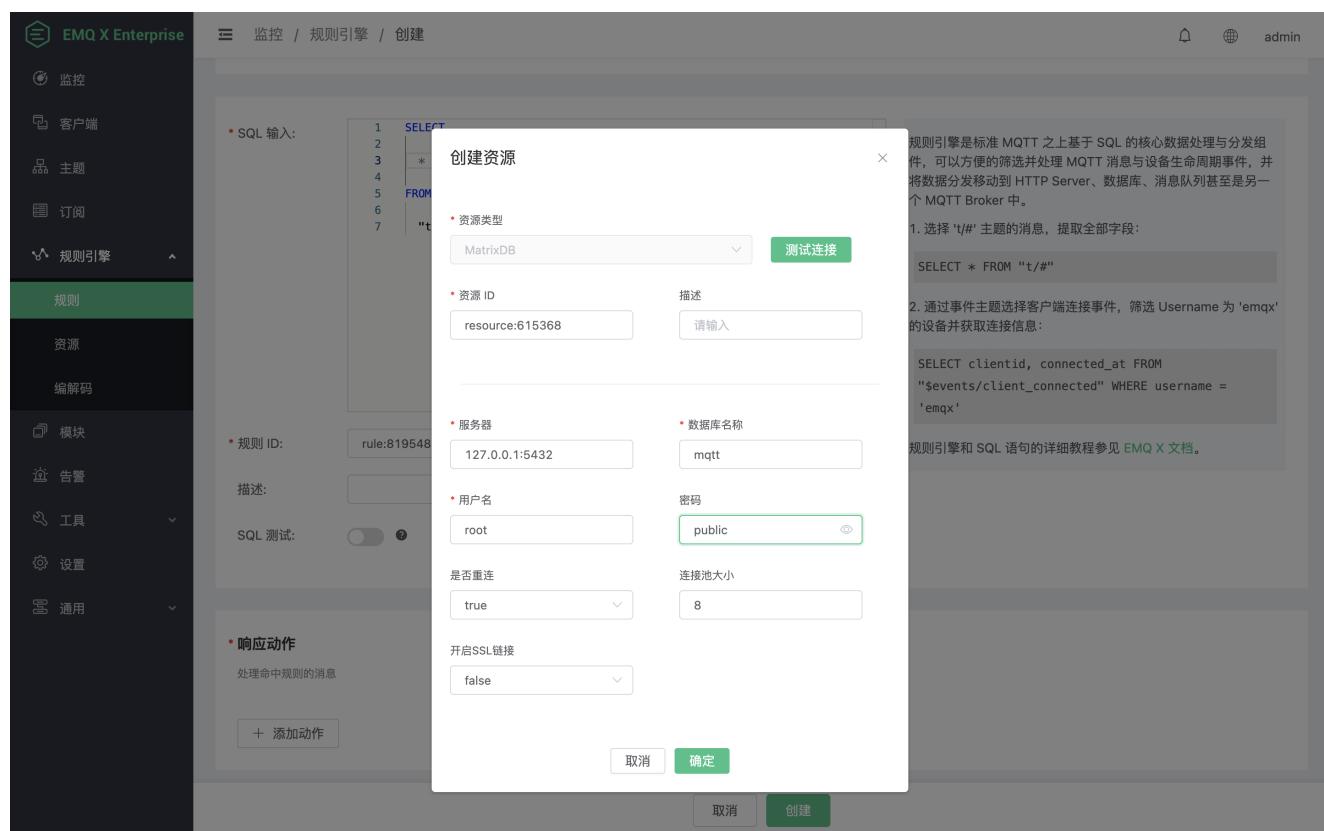
用户名、密码, 身份验证凭据。

是否重连, 是否启用自动重连。

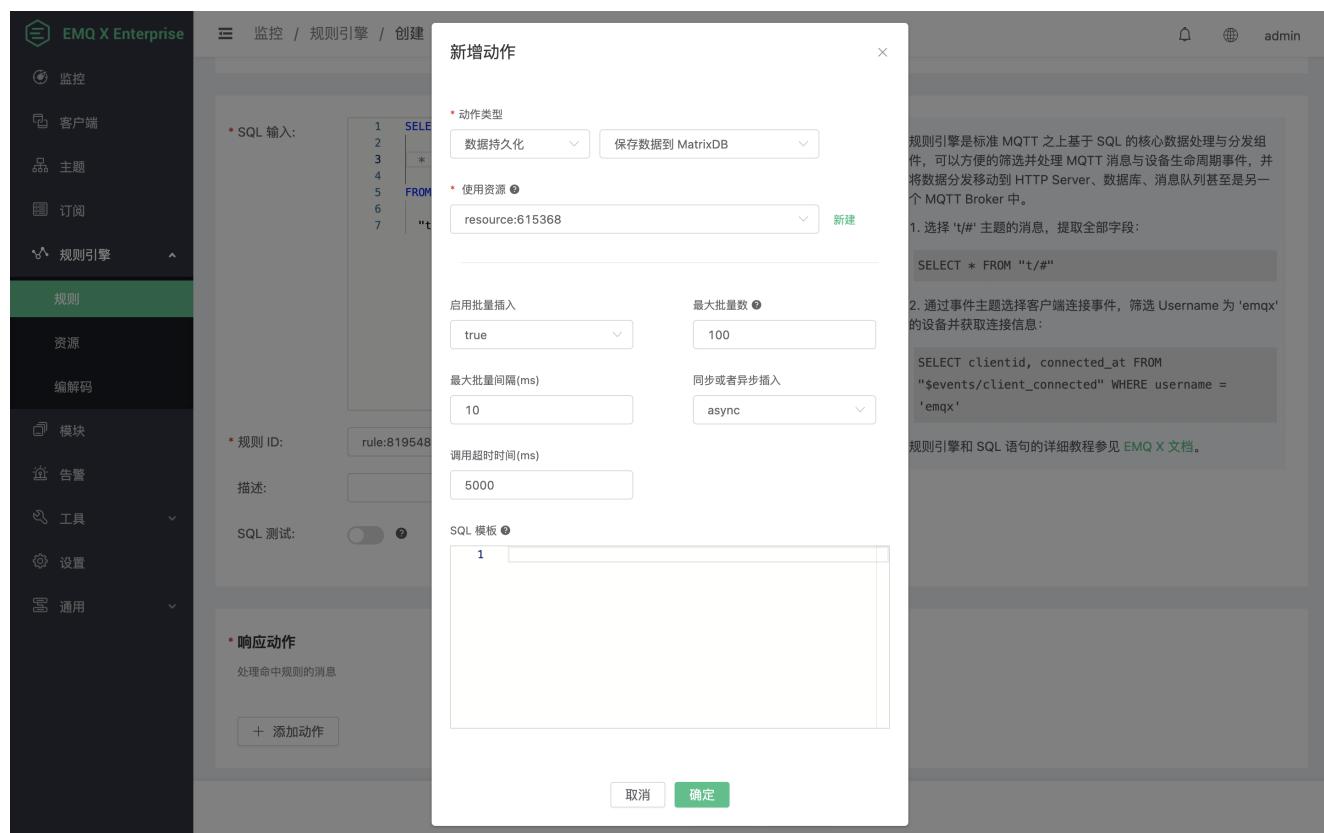
连接池大小, 连接进程池大小, 合理配置连接池大小以获取最佳性能。

开启 **SSL** 连接, 是否启用 **TLS** 连接。

配置完成后, 点击 **确定** 以完成创建。



资源创建成功后我们将回到 新增动作 页面, 使用资源 也自动填充为我们刚刚创建的 **Matrix** 资源的资源 ID。



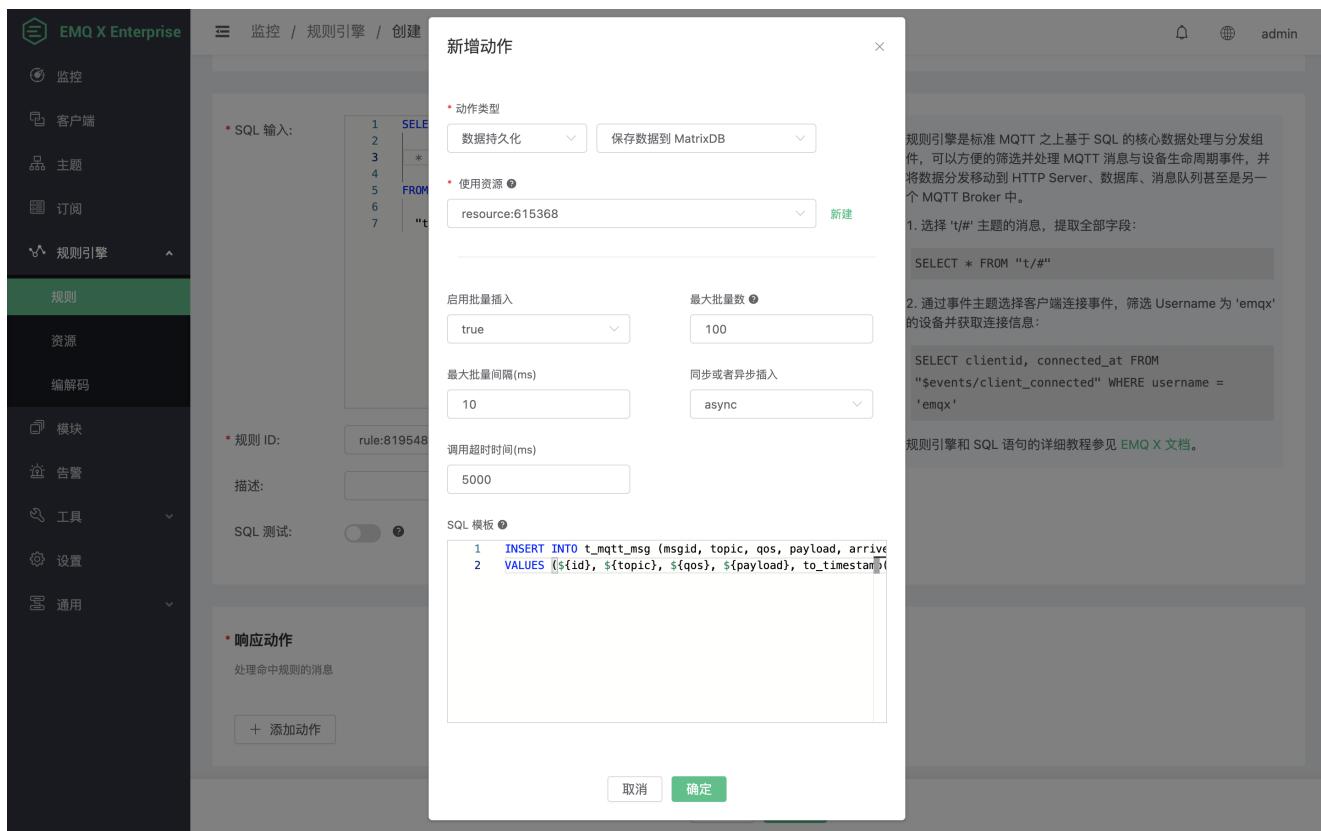
- 2). 启用批量插入，是否启用批量插入，高并发场景下开启批量插入可显著改善写入性能。
- 3). 最大批量数，单次批量请求可以发送的最大 **INSERT SQL** 条目。
- 4). 最大批量间隔，两次批量请求之间最大的等待间隔。
- 5). 同步或异步插入，指定进行同步或异步调用。
- 6). 调用超时时间，以同步方式执行动作的超时时间，此选项仅在同步插入时有效。
- 7). **SQL 模板**，包含了占位符的 **SQL** 模板，用以插入或更新数据到数据库。本示例中我们使用以下 **SQL**：

```

1   INSERT INTO t_mqtt_msg (msgid, topic, qos, payload, arrived)
2     VALUES (${id}, ${topic}, ${qos}, ${payload}, to_timestamp(${timestamp}::double precision / 1000
3   )

```

这里我们使用了 **\${id}** 等占位符，它们会在动作执行时被替换为运行时数据。



配置完成后，点击 确定 以完成动作的添加。然后在规则页面点击最下方的 创建 按钮完成规则创建。

完成 **MatrixDB** 资源和规则的创建后，我们来测试验证一下。我们直接使用 **Dashboard** 中的 **MQTT** 客户端工具来发布一条消息。本示例中我们将消息主题改为 `t/1` 以命中我们设置的规则，**Payload** 和 **QoS** 保持不变，然后点击发布。

Topic	QoS	Payload	时间
t/1	0	{ "msg": "hello" }	16:48:22

消息发布成功后，我们将可以在 `t_mqtt_msg` 表中看到新写入的数据：

```
postgres=# select * from t_mqtt_msg;
 id |          msgid          | sender | topic | qos |      payload      |      arrived
----+-----+-----+-----+-----+-----+-----+
 93 | 0005D211BC820098F440000011E30001 |     | t/1 | 0 | { "msg": "hello" } | 2021-12-01 16:48:22.467
(1 row)
```

Save data to TDengine

[TDengine](#) 是[涛思数据](#) 推出的一款开源的专为物联网、车联网、工业互联网、**IT** 运维等设计和优化的大数据平台。除核心的快 **10** 倍以上的时序数据库功能外，还提供缓存、数据订阅、流式计算等功能，最大程度减少研发和运维的复杂度。

EMQX 支持通过发送到 **Web** 服务的方式保存数据到 **TDengine**，也在企业版上提供原生的 **TDengine** 驱动实现直接保存。

使用 **Docker** 安装 **TDengine** 或在 [Cloud](#) 上部署：

```
1 docker run --name TDengine -d -p 6030:6030 -p 6035:6035 -p 6041:6041 -p 6030-6040:6030-6040
  /udp tdengine/tdengine
```

进入 **Docker** 容器：

```
1 docker exec -it TDengine bash
2 taos
```

创建 “**test**” 数据库：

```
1 create database test;
```

创建 **t_mqtt_msg** 表，关于 **TDengine** 数据结构以及 **SQL** 命令参见 [TAOS SQL](#)：

```
1 USE test;
2 CREATE TABLE t_mqtt_msg (
3     ts timestamp,
4     msgid NCHAR(64),
5     mqtt_topic NCHAR(255),
6     qos TINYINT,
7     payload BINARY(1024),
8     arrived timestamp
9 );
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：

```

1  SELECT
2
3      *,
4      now_timestamp('millisecond')  as ts
5
6  FROM
7
8  "#"

```

* SQL 输入:

```

1  SELECT
2
3      *
4
5  FROM
6
7  "#"
8

```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at			timestamp	node	

规则 SQL 示例

```
SELECT payload.msg as msg FROM "#" WHERE msg = 'hello'
```

备注:

SQL 测试: [?](#)

后续动作创建操作可以根据你的 **EMQX** 版本灵活选择。

原生方式（企业版）

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“保存数据到 **TDengine**”。

仅限企业版 **4.1.1** 及以后版本。

填写动作参数:

“保存数据到 **TDengine**” 动作需要两个参数:

1. **SQL 模板**。这个例子里我们向 **TDengine** 插入一条数据，注意我们应当在 **SQL** 中指定数据库名，字符类型也要用单引号括起来，**SQL** 模板为：

```

1  insert into test.t_mqtt_msg(ts, msgid, mqtt_topic, qos, payload, arrived) values (${ts}, '${id}', '${topic}', ${qos}, '${payload}', ${timestamp})

```

2. 关联资源的 **ID**。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **TDengine** 资源：

填写资源配置:

用户名填写“**root**”，密码填写缺省密码“**taosdata**”，**TDengine** 不在资源中配置数据库名，请在 **SQL** 中自行配置。

* 资源类型

 ▼ 测试连接

* 资源 ID

 描述

TDengine 主机 ?

 TDengine 端口

连接池大小 ?

 * TDengine 用户名

* TDengine 密码

 取消 确定

点击“新建”按钮。

返回响应动作界面，点击“确认”。

* 动作类型

数据持久化	▼	保存数据到 TDengine	▼
-------	---	----------------	---

* 使用资源 ?

resource:997514	▼	新建
-----------------	---	----

启用批量插入

true	▼
------	---

最大批量数 ?

100	▼
-----	---

最大批量间隔(ms) ?

10

同步或者异步插入 ?

async	▼
-------	---

调用超时时间(ms) ?

5000

* SQL 模板 ?

1 insert into test.t_mqtt_msg(ts, msgid, mqtt_topic, qos, pa 2
--

返回规则创建界面，点击“创建”。

通过发送数据到 Web 服务写入

为支持各种不同类型平台的开发, **TDengine** 提供符合 **REST** 设计标准的 **API**。通过 [RESTful Connector](#) 提供了最简单的连接方式, 即使用 **HTTP** 请求携带认证信息与要执行的 **SQL** 操作 **TDengine**。

EMQX 规则引擎中有功能强大的发送数据到 **Web** 服务功能, 可以实现无缝实现上述操作。

关联动作:

在“响应动作”界面选择“添加”, 然后在“动作”下拉框里选择“保存数据到 **Web** 服务”。

EMQX 规则引擎中有功能强大的****发送数据到 **Web** 服务功能****, 可以实现无缝实现上述操作。

填写动作参数:

“保存数据到 **Web** 服务”动作需要 **5** 个参数:

1. 使用资源: 关联 **Web Server** 资源, 可点击创建按钮新建资源
2. **Method**: **HTTP** 请求方式, 使用 **POST**
3. **Path**: 写入数据路径 **/rest/sql**
4. **Headers**: 使用 **Basic** 认证, 默认用户名密码 "**root:taosdata**" **base64** 编码 **cm9vdDp0YW9zZGF0YQ==**
5. **Body**: 请求体 这个例子里我们向 **TDengine** 插入一条数据, 应当在请求体内拼接携带 **INSERT SQL**。注意我们应该在 **SQL** 中指定数据库名, 字符类型也要用单引号括起来, 消息内容模板为:

```
1 -- 注意: topic 处添加了作为标识, 因为此示例中我们会有两个资源写入 TDengine, 标识区分了原生方式与 Web Server 写入的数据
2 insert into test.t_mqtt_msg(ts, msgid, mqtt_topic, qos, payload, arrived) values (${ts}, '${id}', 'http server ${topic}', ${qos}, '${payload}', ${timestamp})
```

* 动作类型

数据转发	发送数据到 Web 服务
------	--------------

* 使用资源 [?](#)

resource:718618	新建
-----------------	----

Method [?](#)

POST	▼
------	---

Path [?](#)

/rest/sql

Headers [?](#)

键	值	添加
Authorization	Basic cm9vdDp0YW9zZGF0Y	删除

Body [?](#)

insert into test.t_mqtt_msg(ts, msgid, mqtt_topic, qos, payload, arrived) values (\${ts}, '\${id}', 'web server \${topic}', \${qos}, '\${payload}', \${timestamp})
--

取消

确定

创建关联的 **Web Server** 资源：

填写资源配置：

请求 **URL** 填写 <http://127.0.0.1:6041>，其他字段按照服务器设定填写即可。配置了加密传输请填写证书信息。示例中仅使用默认的参数配置。

返回规则创建界面，点击“创建”。

创建资源

X

*** 资源类型**

WebHook

测试连接

*** 资源 ID**

resource:912780

描述

请输入

*** 请求 URL** ⓘ

http://127.0.0.1:6041

连接超时时间

5s

请求超时时间

5s

连接池大小

8

Enable Pipelining ⓘ

true

CA 证书文件

选择文件

SSL Key

测试

在规则列表里，点击规则 **ID** 连接，可以预览刚才创建的规则：

The screenshot shows the EMQX Enterprise interface with a sidebar on the left containing navigation links like 监控, 客户端, 主题, 订阅, 规则引擎, 规则, 资源, 编解码, 模块, 告警, 工具, 设置, 和 通用。右侧是规则详情页，标题为 rule:889979。该规则的基本信息包括主题 #, 描述, 查询字段 *, 筛选条件, 和规则 SQL:

```

SELECT
  *
FROM
  "#"
  
```

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/a"
2 QoS: 1
3 Payload: {"msg": "hello"}      sh
  
```

然后检查 TDengine 表，新的 record 是否添加成功：

```

1 select * from t_mqtt_msg;      sh

taos> select * from test.t_mqtt_msg;
          ts      |      msgid      |      mqtt_topic      |      qos      |      payload      |      arrived      |
=====
2022-06-06 09:40:25.163 | 0005E0C442F3A6DEF44300000BE... | web server t/a      |      1      | {"msg": "hello"}      | 2022-06-06 09:40:25.029 |
2022-06-06 09:40:25.172 | 0005E0C442F3A6DEF44300000BE... | t/a      |      1      | {"msg": "hello"}      | 2022-06-06 09:40:25.029 |
Query OK, 2 row(s) in set (0.005928s)
taos> 
  
```

保存数据到 **Lindorm** 数据库 (自 e3.3.6 和 e4.4.1 起)

首先确保 **Lindorm** 数据库开通服务。

确保部署 **EMQX** 的主机 IP 在访问白名单中。阿里云部署的 **EMQX** 可以通过云主机内网访问 **Lindorm**，其他类型的部署方式需要开启 **Lindorm** 外网访问功能。此类操作步骤请参考阿里 **Lindorm** 操作文档。

创建数据库

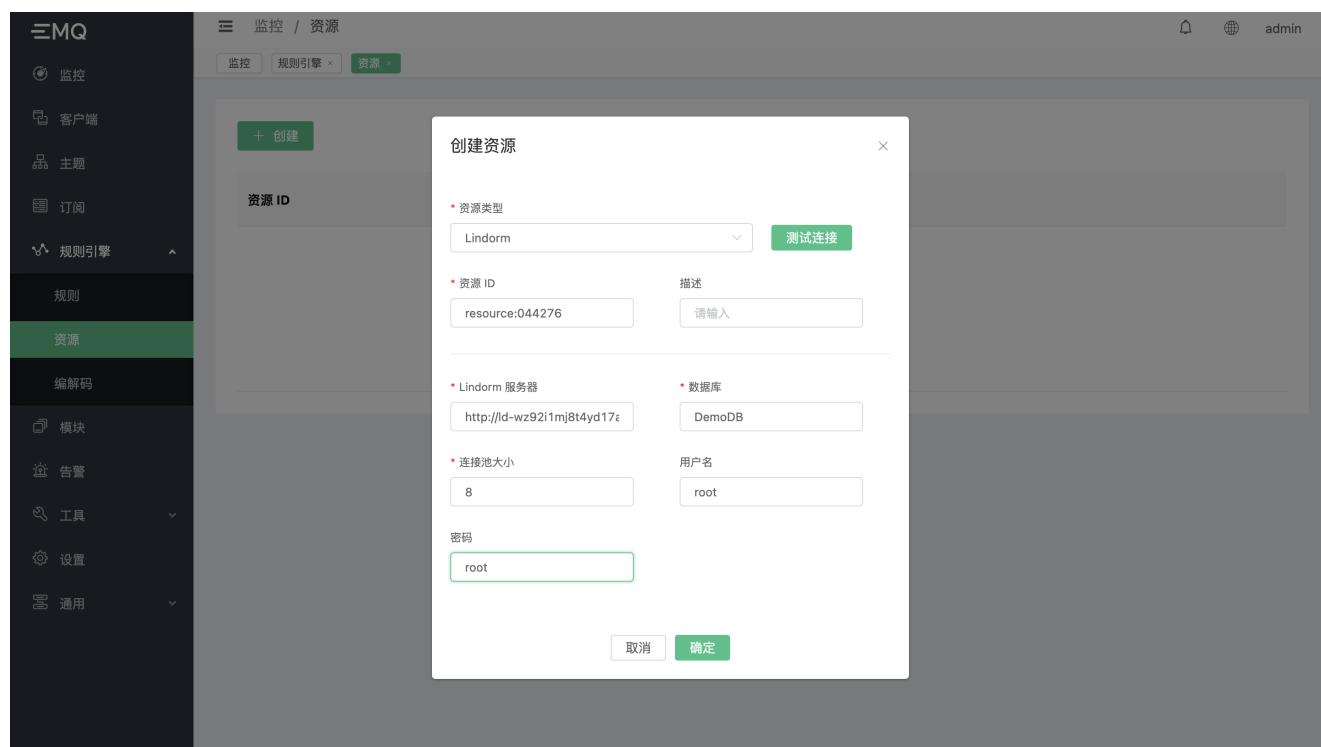
```
1 CREATE DATABASE DemoDB1
```

创建表

```
1 CREATE TABLE demo_sensor(
2     device_id VARCHAR TAG,
3     time BIGINT,
4     msg VARCHAR)
```

资源需要以下启动参数：

- **Lindorm** 服务器： **Lindorm** 访问地址， **Lindorm** 提供了域名访问，阿里云主机请填写内网访问地址， 默认端口 8242，根据实际情况填写，需要添加 http:// 前缀；
- 数据库：数据存储的数据库名称，根据创建的服务填写；
- 连接池大小：数据库写入数据的进程池，根据业务量填写，写入是阻塞型请求，高并发业务推荐 CPU 核心数 * 4 或者 CPU 核心数 * 8 以上；
- 用户名：未开启用户认证不填写，已经开启请按照实际情况填写；
- 密码：未开启用户认证不填写，已经开启请按照实际情况填写；



确保资源状态可用（非阿里云部署，可能会出现创建后首次链接比较慢导致不可用状态，点击状态按钮刷新状态）。

The screenshot shows the EMQX interface with the sidebar open. The 'Resources' tab is selected. A table lists a single resource entry:

资源 ID	资源类型	描述
resource:044276	Lindorm	可用

创建规则

```

1   SELECT
2     payload as msg,
3     clientid,
4     timestamp
5   FROM
6     "#"

```

The screenshot shows the 'Create Rule' dialog. In the 'SQL Input' field, the following SQL query is entered:

```

* SQL 输入:
1 ┌─ SELECT
2   payload as msg,
3   clientid,
4   timestamp
5
6 ┌─ FROM
7   "#"

```

The 'Rule ID' field contains 'rule:595998'. Below the dialog, a note explains that rule engines process MQTT messages based on SQL queries and can be used for various operations like saving to databases or triggering external events.

添加动作：

- 动作类型：数据持久化；保存数据到 **Lindorm**；
- 使用资源：选择创建的资源 **ID**；
- 表名：同步模式可以使用占位符动态规划，异步批量写入请严格使用表名，不能动态改变；

- 同步写入：同步即每条数据立即入库，异步模式开启批量功能，数据将按照批量大小和间隔时间规则写入；
- 异步模式批量大小：默认 **100**，推荐 **100 - 400** 取值，可根据业务规划改变；同步模式下忽略此参数；
- 异步模式批量间隔：默认 **100**，单位毫秒，推荐 **10 - 200**；
- 时间戳：单位毫秒，推荐使用消息时间，空数据会按照规则命中的时间戳计算；
- **Tags**：数据标签键值对，根据创建的表结构填写；
- **Fields**：数据键值对，根据创建的表结构填写；

The screenshot shows the EMQX Rule Engine creation interface. On the left is a sidebar with navigation options like Monitoring, Client, Topic, Subscription, Rule Engine, Rules, Resources, Decoding, Modules, Alerts, Tools, Settings, and General. The Rule Engine section is expanded. The main area is titled 'Create Rule' and contains fields for 'Action Type' (选择持久化), 'Resource' (resource:044276), 'Table Name' (demo_table), 'Batch Size' (100), 'Sync Mode' (false), 'Interval' (100ms), 'Timestamp' (\${timestamp}), and 'Tags'. Below these are sections for 'Response Actions' and 'Data Path'. A preview window on the right shows the generated SQL code:

```

1. 选择 't/#' 主题的消息，提取全部字段：
SELECT * FROM "t/#"

2. 通过事件主题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'

```

规则引擎和 SQL 语句的详细教程参见 [EMQ X 文档](#)。

点击确定，点击创建，查看规则：

The screenshot shows the EMQX Rule Engine list interface. The sidebar is identical to the previous screenshot. The main area displays a table of rules:

ID	主题	监控	描述	状态	响应动作
1 rule:595998	#	null		<input checked="" type="checkbox"/>	保存数据到 Lindorm [编辑] [删除]

使用 MQTT 客户端发布消息，查看规则命中与成功失败计数；

The screenshot shows the EMQX Enterprise V4.4 Rules interface. On the left, a sidebar menu includes '规则引擎' (Rule Engine) under '规则' (Rules). The main content area is titled '规则引擎 / 规则引擎' and displays a single rule entry:

ID	主题	监控	描述	状态
1 rule:721955	#	lindorm		开启 (green switch)

On the right, there are two sections: '规则统计' (Rule Statistics) and '动作统计' (Action Statistics). The '规则统计' section shows metrics like命中次数 (Matched Count), 当前速度 (Current Rate), 最大执行速度 (Max Execution Rate), and 最近5分钟执行速度 (Last 5 Minutes Execution Rate). The '动作统计' section shows the action 'data_to_lindorm' with its success and failure counts.

使用 **API** 查询数据库写入结果：

```
1 # 替换
2 # ${LINDORM_SERVER}: 服务器地址
3 # ${DB_NAME}: 数据库名
4 # ${LINDORM_TABLE}: 表名
5 curl -X POST http://${LINDORM_SERVER}:8242/api/v2/sql?database=${DB_NAME} -H "Content-Type: text/plain" -d 'SELECT count(*) FROM ${LINDORM_TABLE}'
```

保存数据到 Alibaba Tablestore 数据库

创建数据库实例（创建实例操作步骤请参考 **Tablestore** 官方文档）。

点击实例管理，点击时序表列表。创建时序表。

The screenshot shows the Alibaba Tablestore management console. The left sidebar has sections like Overview, All Instances, Audit Log, and Best Practices. The main area is titled 'Instance Management' with tabs for 'Basic Details', 'Instance Monitoring', 'Network Management', 'Data Migration', and 'SQL Query'. Below these tabs, there's a section for 'Instance Access Address' with fields for Endpoint, Public IP, VPC, and Dual Network. Under 'Basic Information', it shows the instance name 'emqx-test', storage mode 'Row Mode', table size, and last update time. A red box highlights the 'Time Series Table List' tab in the navigation bar at the bottom of the main content area. Another red arrow points to the 'Create Time Series Table' button.

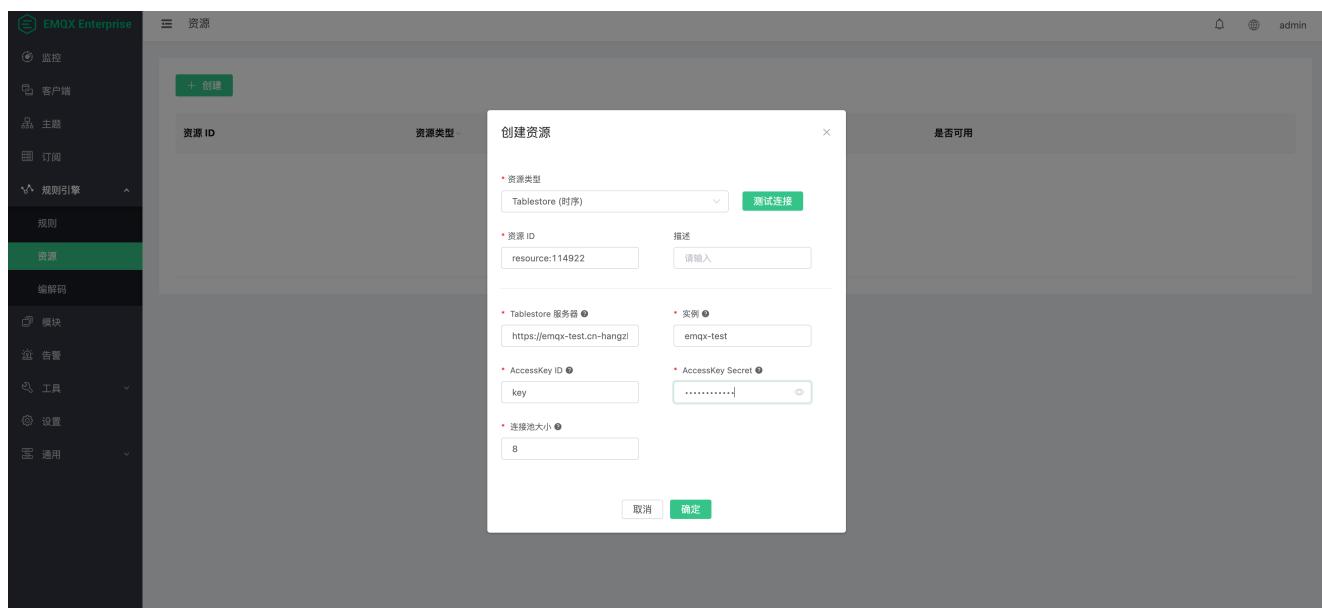
创建时间线

点击上一步中创建的时序表，进入时序表管理。点击数据管理，创建时间线，按照业务设计创建出时间线。（此步骤可以跳过，当写入不存在的时间线时，**Tablestore** 会自动创建出新的时间线，示例中不对时间线操作）

This screenshot shows the 'Time Series Table Management' page for the 'ots_demo' table. It includes tabs for 'Basic Details', 'Data Management' (which is selected), and 'SQL Query'. On the left, there are sections for 'Metric Name' (demo_line) and 'Data Source' (demo_resource). Below these are filters for 'Measurement Name' and 'Time Range'. A modal window titled 'Add Timeline' is open in the center. It has fields for 'Measurement Name' (必填), 'Data Source', and 'Tags'. There are also sections for 'Attributes' and 'Operations'. At the bottom of the modal are 'Confirm' and 'Cancel' buttons. A red box highlights the 'Add Timeline' button in the modal.

创建规则引擎资源

点击规则引擎，资源，创建资源，下拉框中找到 **Tablestore**（时序）



使用创建出的实例属性，填写对应 Tablestore 服务器 和 实例 名称。

AccessKey & Secret 需要使用阿里账号申请，详细申请步骤，请参考 **Tablestore** 官方文档。

服务器地址请按照部署方式的不同，填写不同的域名。

This screenshot shows the Alibaba Cloud Tablestore instance management interface. The instance 'emqx-test' is selected. The 'Instance Address' section is highlighted with a red box, showing four network addresses: '内网: https://emqx-test.cn-hangzhou.ots-internal.aliyuncs.com', '公网: https://emqx-test.cn-hangzhou.ots.aliyuncs.com', 'VPC: https://emqx-test.cn-hangzhou.vpc.tablestore.aliyuncs.com', and '公网(双栈): https://emqx-test.cn-hangzhou.tablestore.aliyuncs.com'. The 'Instance Basic Information' section shows the instance name 'emqx-test' and other details like '表模式: 按量模式', '表数据大小: ..', '表大小更新时间: ..', and '存储规格: 存储弹性, 容量无限制'. On the right, there's a summary of table statistics: '表总数 / 上限: 0 / 64', '创建时间: 2022年4月19日 16:47:58', and '实例描述: emqx-ots联调测试'.

创建规则

规则中 **SQL** 语句的编写，请参考官网文档中规则引擎章节。

编辑规则

使用 SQL 设定规则，对消息数据筛选、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。

* SQL 输入:

```

1  SELECT
2
3  payload.k_int as k_int,
4  payload.k_bool as k_bool,
5  payload.k_str as k_str,
6  payload.k_bin as k_bin,
7  payload.k_float as k_float,
8  clientid
9
10 FROM
11 "#"
12
13

```

* 规则 ID: rule:229536

描述:

SQL 测试:

* 响应动作

处理命中规则的消息

规则引擎是标准 MQTT 之上基于 SQL 的核心数据处理与分发组件，可 MQTT 消息与设备生命周期事件，并将数据分发移动到 HTTP Server，是另一个 MQTT Broker 中。

1. 选择 `t/#` 主题的消息，提取全部字段:
SELECT * FROM "t/#"
2. 通过事件主题选择客户端连接事件，筛选 Username 为 'emqx' 的连接:
SELECT clientid, connected_at FROM "\$events/client_connected" WHERE username = 'emqx'

规则引擎和 SQL 语句的详细教程参见 [EMQX 文档](#)。

示例中使用的规则 SQL

```

1  SELECT
2
3  payload.k_int as k_int,
4  payload.k_bool as k_bool,
5  payload.k_str as k_str,
6  payload.k_bin as k_bin,
7  payload.k_float as k_float,
8  clientid
9
10 FROM
11 "#"

```

创建动作

表名与度量名称，请按照创建的表名与度量名称填写（度量名称可以为空字符串）。

参数列表

参数	定义
度量名称	Tablestore 度量名称
数据源	Tablestore 数据源, 可为空字符串
时间戳 (微秒)	单位微秒, EMQX 中 MQTT 消息中默认带有毫秒精度的时间戳, 不可直接使用。缺省值为消息到达 EMQX 的微秒时间戳
时间线缓存	Tablestore 识别当前数据是否需要创建或更新时间线元数据, 默认开启
同步写入	开启批量写入, 或单条同步写入。批量写入时, 备选动作不会触发
异步模式批量大小	批量写入最大数据条数, 仅在关闭同步写入时生效
异步模式批量间隔 (毫秒)	批量写入最长时间间隔, 仅在关闭同步写入时生效
Tags	数据标签, 所有数据都以字符串形式处理
数据列名	数据键值对, 数据类型自动识别。字符串数据默认按照二进制数据处理以保证字符集最佳兼容性
字符串型数据列名	字符串键值对, 数据会按照字符串类型处理

Tablestore 支持的数据格式:

- **int**
- **float**
- **boolean**
- **string**
- **binary**

这些格式可以被规则引擎自动的识别并分类。但是字符串型数据, 默认处理方式为二进制数据, 以保证最佳的兼容性。如果需要指定字段值为字符串类型, 可以在创建动作时, 将字段填写到 **字符串型数据列名** 中。之后会则引擎会按照字符串的格式方式处理改字段值。

生产测试数据

使用先进的桌面 MQTT 客户端 **MQTT X**，登录设备，并发送一条数据。

输入度量名称（演示使用的是 `m_re2`），**client ID** 为 `123456`，我们使用 `client=123456` 作为查询条件，点击查询。

The screenshot shows the EMQX Enterprise V4.4 Tablestore Data Management interface. On the left, there's a sidebar with navigation links like 'Tablestore', '概览', '全部实例', '审计日志', '权威指南', and '最佳实践'. The main area has a breadcrumb path: '首页 / 实例列表 / emqx-test / rule_engine_test'. A top bar includes a search input, user information, and navigation links for '费用', '工单', 'ICP 备案', '企业', '支持', 'App', and '简体'.

In the center, there's a table titled '时序表管理' (Time Series Table Management) with tabs for '基本详情', '数据管理' (selected), and 'SQL查询'. The table lists data series with columns: '度量名称' (Metric Name), '数据源' (Data Source), '标签' (Tags), '属性' (Properties), '更新时间' (Last Update), and '操作' (Operations). One row is highlighted with a red border, showing a timestamp of '2022-04-24 16:45:57' and a tag 'client=123456'.

Below the table is a query form with fields for '度量名称' (Metric Name), '数据源' (Data Source), '标签' (Tags), and '查询方式' (Query Type). It also includes a '时间范围' (Time Range) selector and a '插入数据' (Insert Data) button.

At the bottom, there are two tabs: '列表展示' (List Display) and '图展示' (Diagram Display). The 'List Display' tab is selected, showing a table with columns: 'Time' (Time), 'k_bin', 'k_bool', 'k_float', 'k_str2', and 'key_int'. A specific row from this table is also highlighted with a red border.

可以看到数据已经写入成功。

桥接数据到 Kafka

搭建 Kafka 环境，以 MacOS X 为例：

```

1 wget https://archive.apache.org/dist/kafka/2.8.0/kafka_2.13-2.8.0.tgz
2
3 tar -xzf kafka_2.13-2.8.0.tgz
4
5 cd kafka_2.13-2.8.0
6
7 # 启动 Zookeeper
8 ./bin/zookeeper-server-start.sh config/zookeeper.properties
9 # 启动 Kafka
10 ./bin/kafka-server-start.sh config/server.properties

```

创建 Kafka 的主题：

```

1 $ ./bin/kafka-topics.sh --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --t
opic testTopic --create

```

提示

创建 **Kafka Rule** 之前必须先在 **Kafka** 中创建好主题，否则创建 **Kafka Rule** 失败。

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

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

* SQL 输入：

```

1 SELECT
2 *
3
4 FROM
5
6 "t/#"
7
8

```

当前事件可用字段

event	id	clientid	username	payload	peerhost	topic	qos
flags	headers	publish_received_at	timestamp	node			

规则 SQL 示例

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

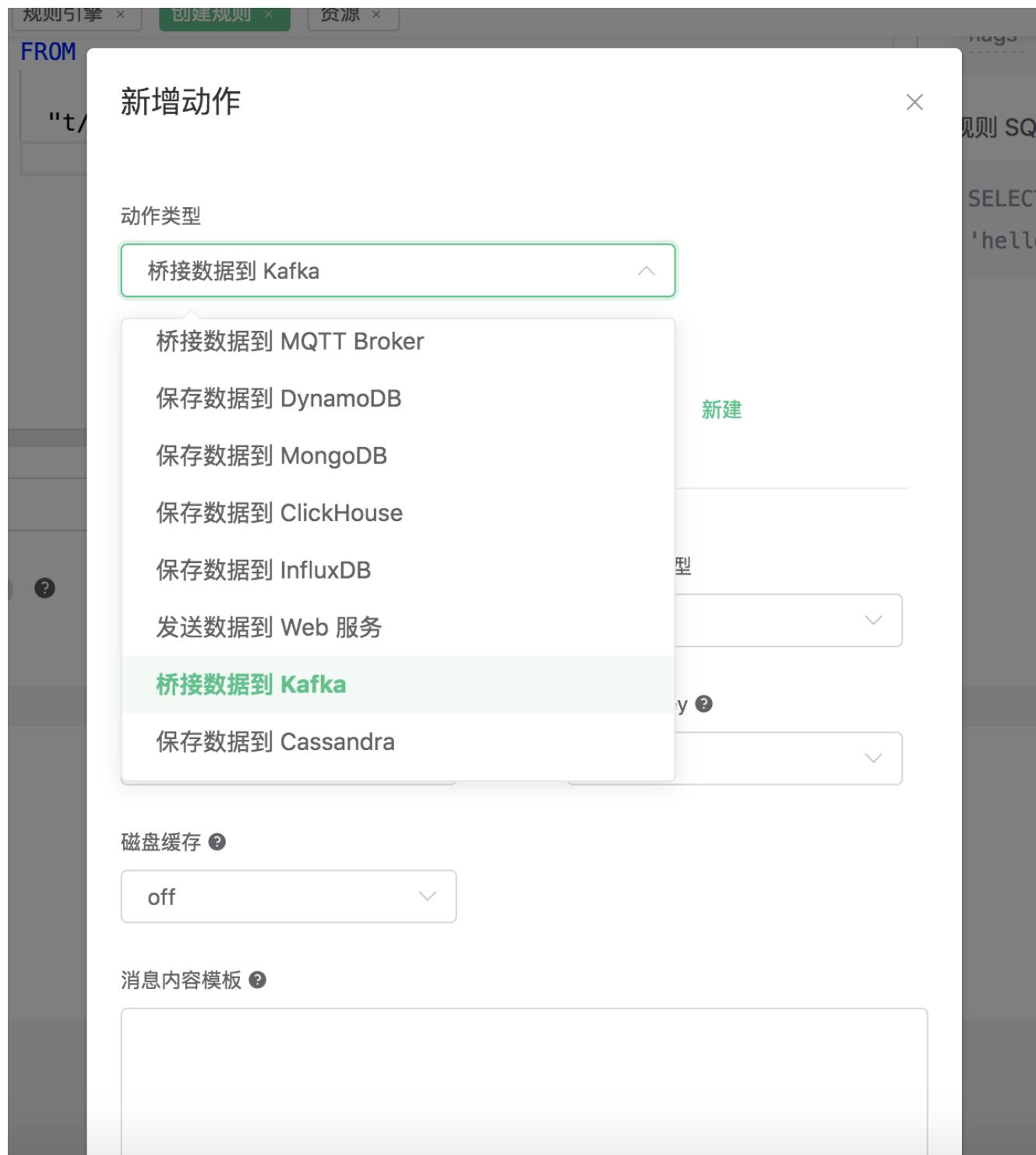
备注：

SQL 测试：



关联动作：

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 **Kafka**”。

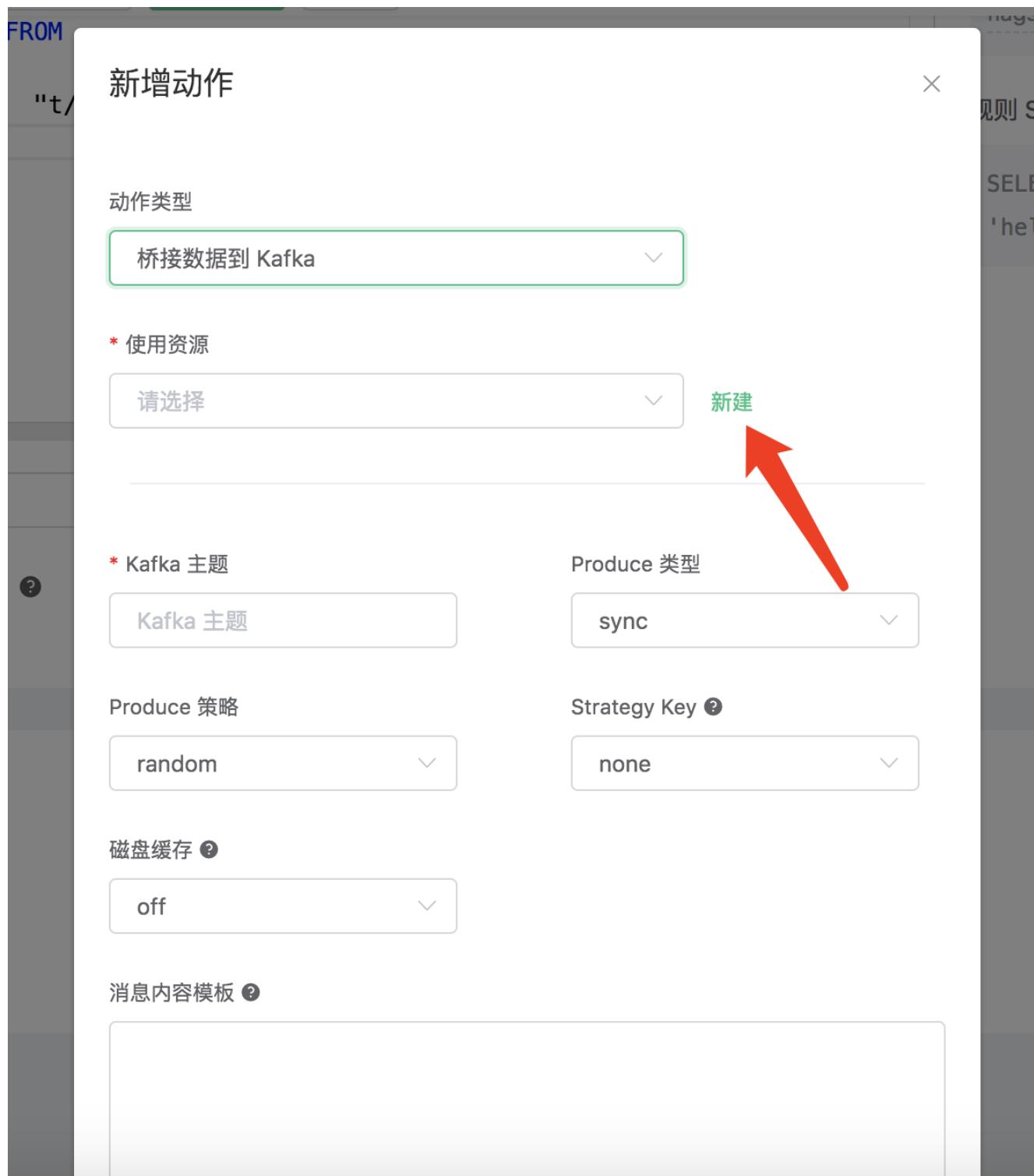


填写动作参数：

“保存数据到 **Kafka**” 动作需要两个参数：

1). **Kafka** 的消息主题

2). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **Kafka** 资源：



选择 **Kafka** 资源”。

填写资源配置：

填写真实的 **Kafka** 服务器地址，多个地址用逗号分隔，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。

创建资源

* 资源类型

Kafka

测试连接

* 资源名称

Kafka

* Kafka 服务器

127.0.0.1:9092

Metadata 更新间隔

3s

同步调用超时时间

3s

最大批处理字节数

1024KB

压缩

no_compression

发送消息的缓冲区大小

1024KB

查询 API 版本号

true

取消 确定

返回响应动作界面，点击“确认”。

新增动作

动作类型

桥接数据到 Kafka

* 使用资源

Kafka 新建

Kafka 主题

testTopic

Produce 类型

sync

?

Produce 策略

random

Strategy Key ?

none

磁盘缓存 ?

off

消息内容模板 ?

返回规则创建界面，点击“新建”。

按照**Kafka**的业务数据，填写 **Kafka 主题** **Produce 类型** **Produce 策略** **Strategy Key** **磁盘缓存**，其中，**消息内容模板** 字段，支持变量。若使用空模板（默认），消息内容为 **JSON** 格式的所有字段。

响应动作 *

处理命中规则的消息

动作类型 桥接数据到 Kafka (data_to_kafka)	编辑 移除
桥接数据到 Kafka	
Produce 类型 sync Produce 策略 random Strategy Key none 磁盘缓存 off 消息内容模板 资源 ID resource:c44d9c4e	
Kafka 主题 testTopic	+ 失败备选动作

+ 添加动作

取消 创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "hello"

```

然后通过 **Kafka** 命令去查看消息是否生产成功：

```

1 $ ./bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic testTopic --from-beginning

```

```

JianBoPro:kafka_2.12-2.3.0 JianBo$ ./bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic testTopic --from-beginning
{"client_id":"mosapub114395-JianBoPro","event":"message.publish","id":"58CED5F91588DF44200000C3F0001","node":"emqx@127.0.0.1","payload":"hello","peername":"127.0.0.1:50840","qos":0,"retain":0,"timestamp":1562326022052,"topic":"t/1","ts":1562326022052,"username":"undefined"}

```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

+ 创建

ID	主题	监控	描述	状态	响应动作
rule:1ccad994	t/#	1		<input checked="" type="checkbox"/>	桥接数据到 Kafka 编辑 删除

桥接数据到 Pulsar

搭建 Pulsar 环境, 以 **MacOS X** 为例:

```

1 $ wget https://archive.apache.org/dist/pulsar/pulsar-2.3.2/apache-pulsar-2.3.2-bin.tar.gz
2
3 $ tar xvfz apache-pulsar-2.3.2-bin.tar.gz
4
5 $ cd apache-pulsar-2.3.2
6
7 # 启动 Pulsar
8 $ ./bin/pulsar standalone

```

创建 Pulsar 的主题:

```

1 $ ./bin/pulsar-admin topics create-partitioned-topic -p 5 testTopic

```

创建规则:

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL:

```

1 SELECT * FROM "t/#"

```

* SQL 输入:

```

1 SELECT
2   *
3   FROM
4     "t/#"

```

当前事件可用字段

```

event id clientid username payload peerhost topic qos
flags headers publish_received_at timestamp node

```

规则 SQL 示例

```

SELECT payload.msg as msg FROM "t/#" WHERE msg =
'hello'

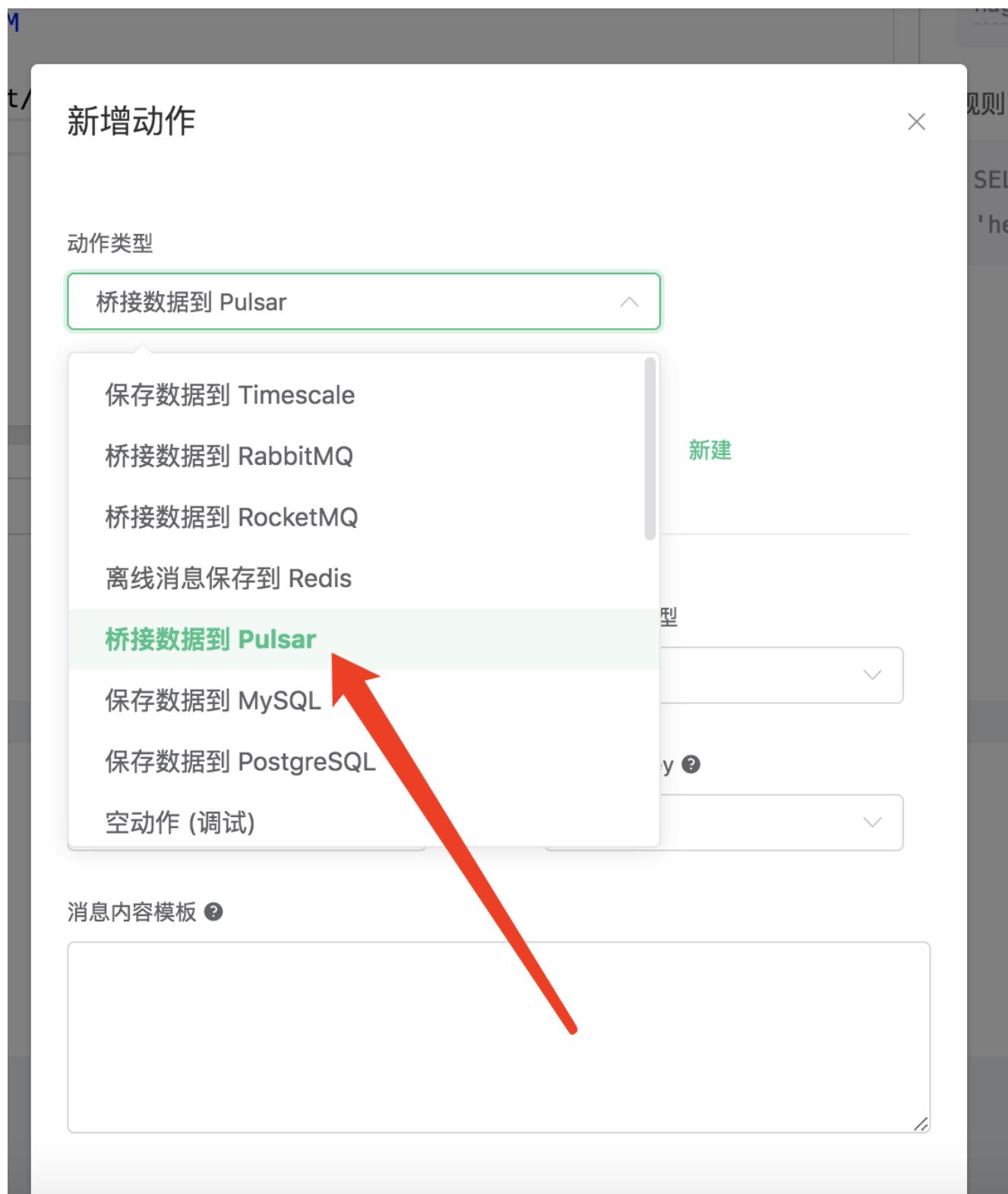
```

备注:

SQL 测试:

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 Pulsar”。

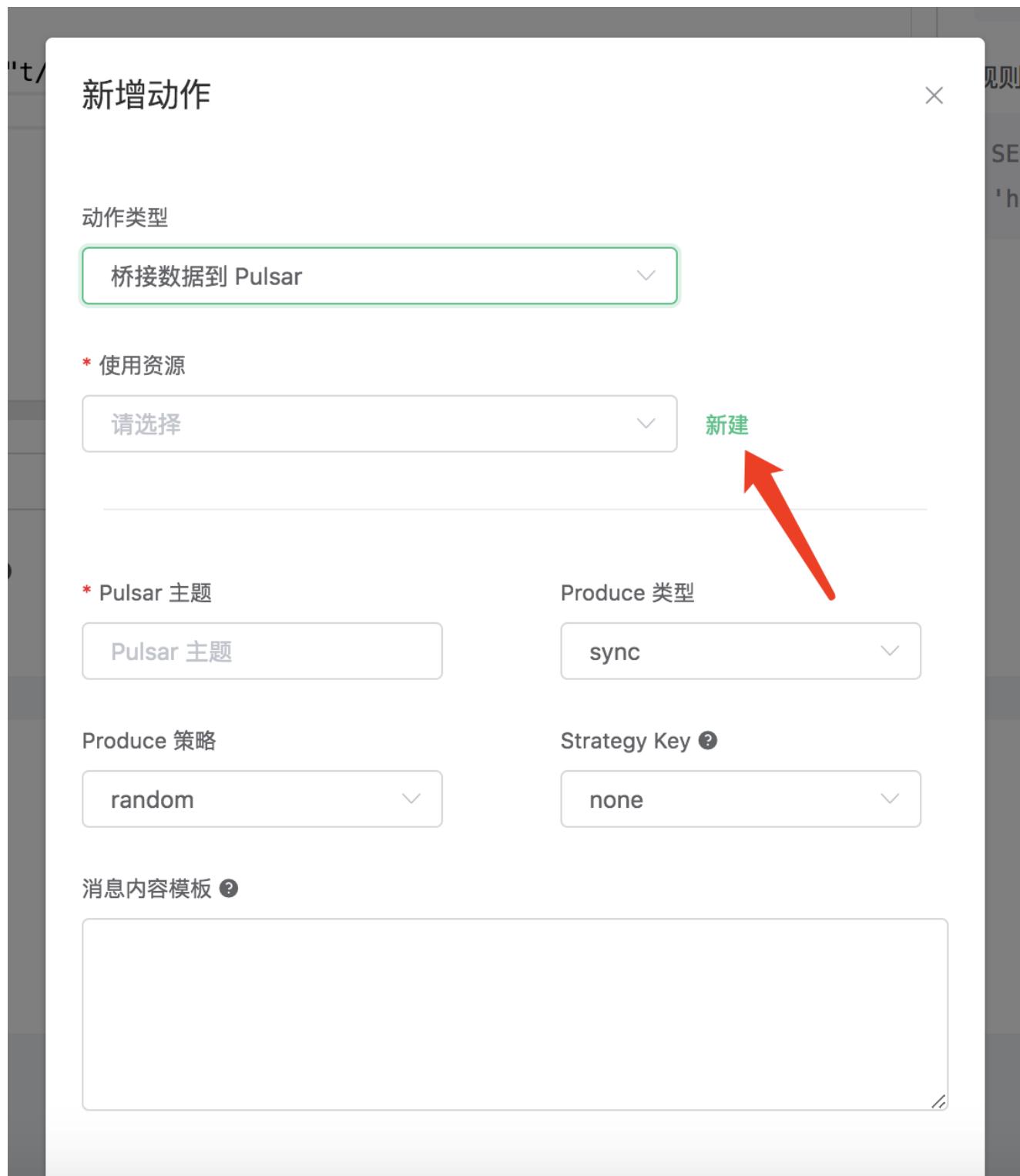


填写动作参数:

"保存数据到 **Pulsar** 动作需要两个参数:

1). **Pulsar** 的消息主题

2). 关联资源。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 **Pulsar** 资源:



选择 **Pulsar 资源**。

填写资源配置：

填写真实的 **Pulsar** 服务器地址，多个地址用逗号分隔，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

服务器地址是逗号分隔的多个 **Pulsar URL**, URL 格式为 `pulsar://<hostname>:<port>`，如果 **Pulsar** 端开启了 **TLS** 的话，则格式为 `pulsar+ssl://<hostname>:<port>`。

最后点击“新建”按钮。

编辑资源

X

* 资源类型

Pulsar

测试连接

* 资源 ID

resource:438118

描述

请输入

* Pulsar 服务器 ?

pulsar://pulsar-broker1:6651

认证类型

none

同步调用超时时间

3s

最大批处理数

100

压缩

no_compression

发送消息的缓冲区大小

1024KB

返回响应动作界面，点击“确认”。

新增动作

动作类型

桥接数据到 Pulsar

* 使用资源

Pulsar

新建

Pulsar 主题

testTopic

Produce 类型

sync

Produce 策略

random

Strategy Key

none

消息内容模板

取消 确定

返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 桥接数据到 Pulsar (data_to_pulsar)
 桥接数据到 Pulsar

编辑 移除

Produce 类型 sync Produce 策略 random Strategy Key none 消息内容模板 资源 ID resource:773fb512 Pulsar 主题 testTopic

+ 失败备选动作

+ 添加动作

取消 创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "hello"
```

然后通过 **Pulsar** 命令去查看消息是否生产成功：

```

1 $ ./bin/pulsar-client consume testTopic -s "sub-name" -n 1000
sh

15:10:21.998 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ConsumerImpl - [persistent://public/default/testTopic-partition-0][sub-name] Subscribed to topic on localhost:127.0.0.1:6650 -- consumer: 3
15:10:21.998 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ConsumerImpl - [persistent://public/default/testTopic-partition-2][sub-name] Subscribed to topic on localhost:127.0.0.1:6650 -- consumer: 2
15:10:21.998 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ConsumerImpl - [persistent://public/default/testTopic-partition-1][sub-name] Subscribed to topic on localhost:127.0.0.1:6650 -- consumer: 1
15:10:21.998 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ConsumerImpl - [persistent://public/default/testTopic-partition-0][sub-name] Subscribed to topic on localhost:127.0.0.1:6650 -- consumer: 0
15:10:22.006 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.MultiTopicsConsumerImpl - [TopicsConsumerFakeTopicNamefa448] [sub-name] Success subscribe new topic testTopic in topics consumer, partitions: 5, allTopicPartitionsNumber: 5
15:10:48.417 [pulsar-client-io-1-1] WARN com.scurrilous.circe.checksum.Crc32cIntChecksum - Failed to load Circe JNI library. Falling back to Java based CRC32c provider
    got message
{"client_id": "mattjs_07f1978c36", "event": "message.publish", "flags": {"dup": false, "retain": false}, "id": "58C9953DA9948F440000009330001", "node": "emqx@127.0.0.1", "payload": "hello", "username": "", "ts": 1561965048273, "topic": "t/a", "ts": 1561965048273, "username": ""}
```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

ID	主题	监控	描述	状态	响应动作
rule:1b9951af	t/#	on		开启	桥接数据到 Pulsar

使用 Pulsar basic/token 认证

EMQX 企业版 4.3.10 以及 4.4.4 以后，我们支持了 Pulsar 的 basic 和 token 认证。

比如，要启用 token 认证，从 Authentication Type 下拉框选择 token：

编辑资源

X

* 资源类型

Pulsar

测试连接

* 资源 ID

resource:438118

描述

请输入

* Pulsar 服务器 ?

pulsar://pulsar-broker1:6651

认证类型

token

basic

token

none

1024KB

同步调用超时时间

3s

压缩

no_compression

然后在对话框的下方提供 JWT:

是否校验服务器证书

false

* JWT

eyJhbGciOiJIUzI1NiJ9.eyJz

取消

确定

桥接数据到 RabbitMQ

搭建 RabbitMQ 环境，以 MacOS X 为例：

```
1 $ brew install rabbitmq  
2  
3 # 启动 rabbitmq  
4 $ rabbitmq-server
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
```

当前事件可用字段

```
event_id clientid username payload peerhost topic qos
flags headers publish_received_at timestamp node
```

规则 SQL 示例

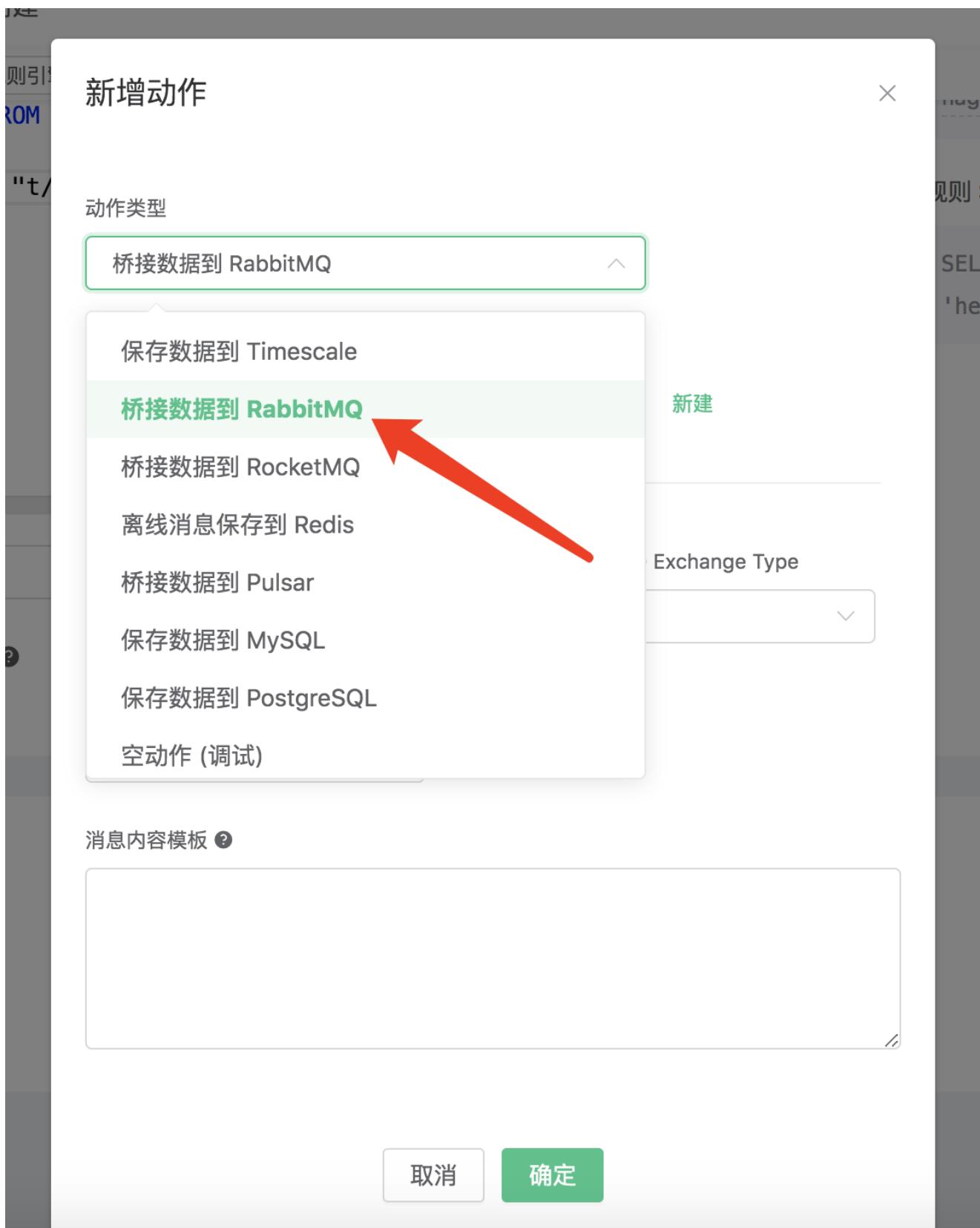
```
SELECT payload.msg as msg FROM "t/#" WHERE msg =
'hello'
```

备注:

SQL 测试:

关联动作：

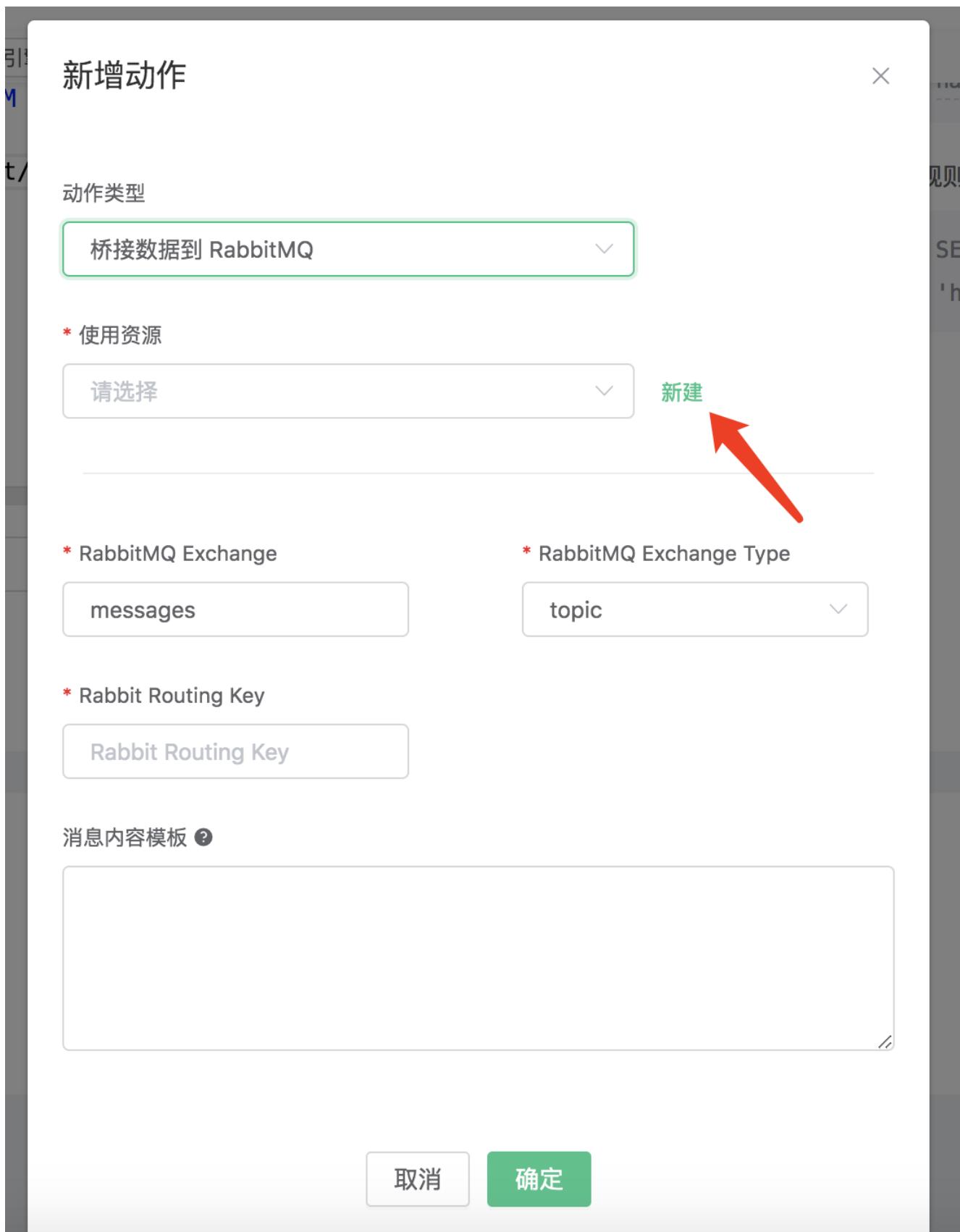
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 RabbitMQ”。



填写动作参数:

"桥接数据到 RabbitMQ" 动作需要四个参数:

- 1). **RabbitMQ Exchange**。这个例子里我们设置 Exchange 为 "messages"，
- 2). **RabbitMQ Exchange Type**。这个例子我们设置 Exchange Type 为 "topic"
- 3). **RabbitMQ Routing Key**。这个例子我们设置 Routing Key 为 "test"
- 4). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 RabbitMQ 资源：



选择 **RabbitMQ** 资源。

填写资源配置：

填写真实的 **RabbitMQ** 服务器地址，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。

创建资源

* 资源类型

RabbitMQ

* 测试连接

* 资源名称

RabbitMQ

* RabbitMQ 服务器

127.0.0.1:5672

* 连接池大小

8

用户名

guest

密码

guest

连接超时时间

5s

虚拟主机

/

心跳间隔

30s

自动重连间隔

2s

取消 确定

返回响应动作界面，点击“确认”。

新增动作

动作类型

桥接数据到 RabbitMQ

* 使用资源

RabbitMQ

RabbitMQ Exchange

messages

RabbitMQ Exchange Type

topic

Rabbit Routing Key

test

消息内容模板 ?

取消 确定

返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "Hello, World\!"
```

编写 **amqp** 协议的客户端，以下是用 **python** 写的 **amqp** 客户端的示例代码：

```

1 #!/usr/bin/env python
2 import pika
3
4 connection = pika.BlockingConnection(
5     pika.ConnectionParameters(host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='messages', exchange_type='topic')
9
10 result = channel.queue_declare(queue='', exclusive=True)
11 queue_name = result.method.queue
12
13 channel.queue_bind(exchange='messages', queue=queue_name, routing_key='test')
14
15 print('[*] Waiting for messages. To exit press CTRL+C')
16
17 def callback(ch, method, properties, body):
18     print(" [x] %r" % body)
19
20 channel.basic_consume(
21     queue=queue_name, on_message_callback=callback, auto_ack=True)
22
23 channel.start_consuming()
```

然后通过 **amqp** 协议的客户端查看消息是否发布成功，以下是

```

↳ python amqp_client.py
[*] Waiting for messages. To exit press CTRL+C
[x] '{"client_id": "mqttjs_8d8b9b6c7b", "event": "message.publish", "id": "58D7899C4EBEBF4430000A7B0004", "node": "emqx@127.0.0.1", "payload": "{ \\"msg\\": \"Hello, World!\" }", "peername": "127.0.0.1:63998", "qos": 1, "retain": 0, "timestamp": 1562923998965, "topic": "t/1", "username": ""}'
```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

ID	主题	监控	描述	状态	响应动作
rule:8edd0fba	t/#	all	命中 1	<input checked="" type="checkbox"/>	桥接数据到 RabbitMQ [编辑] [删除]

桥接数据到 RocketMQ

搭建 RocketMQ 环境，以 MacOS X 为例：

```

1 $ wget https://mirrors.bfsu.edu.cn/apache/rocketmq/4.8.0/rocketmq-all-4.8.0-bin-release.zip
2 $ unzip rocketmq-all-4.8.0-bin-release.zip
3 $ cd rocketmq-all-4.8.0-bin-release/
4
5 # 在conf/broker.conf添加了2个配置
6 brokerIP1 = 127.0.0.1
7 autoCreateTopicEnable = true
8
9 # 启动 RocketMQ NameServer
10 $ ./bin/mqnamesrv
11
12 # 启动 RocketMQ Broker
13 $ ./bin/mqbroker -n localhost:9876 -c conf/broker.conf

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

The screenshot shows the EMQX Rule configuration interface. On the left, there is a code editor window with the following SQL query:

```

1 SELECT * FROM "t/#"

```

To the right of the editor, there are two panels: "当前事件可用字段" (Available fields for current event) and "规则 SQL 示例" (Rule SQL example). The "当前事件可用字段" panel lists several fields: event, id, clientid, username, payload, peerhost, topic, qos, flags, headers, publish_received_at, timestamp, and node. The "规则 SQL 示例" panel contains an example rule SQL:

```

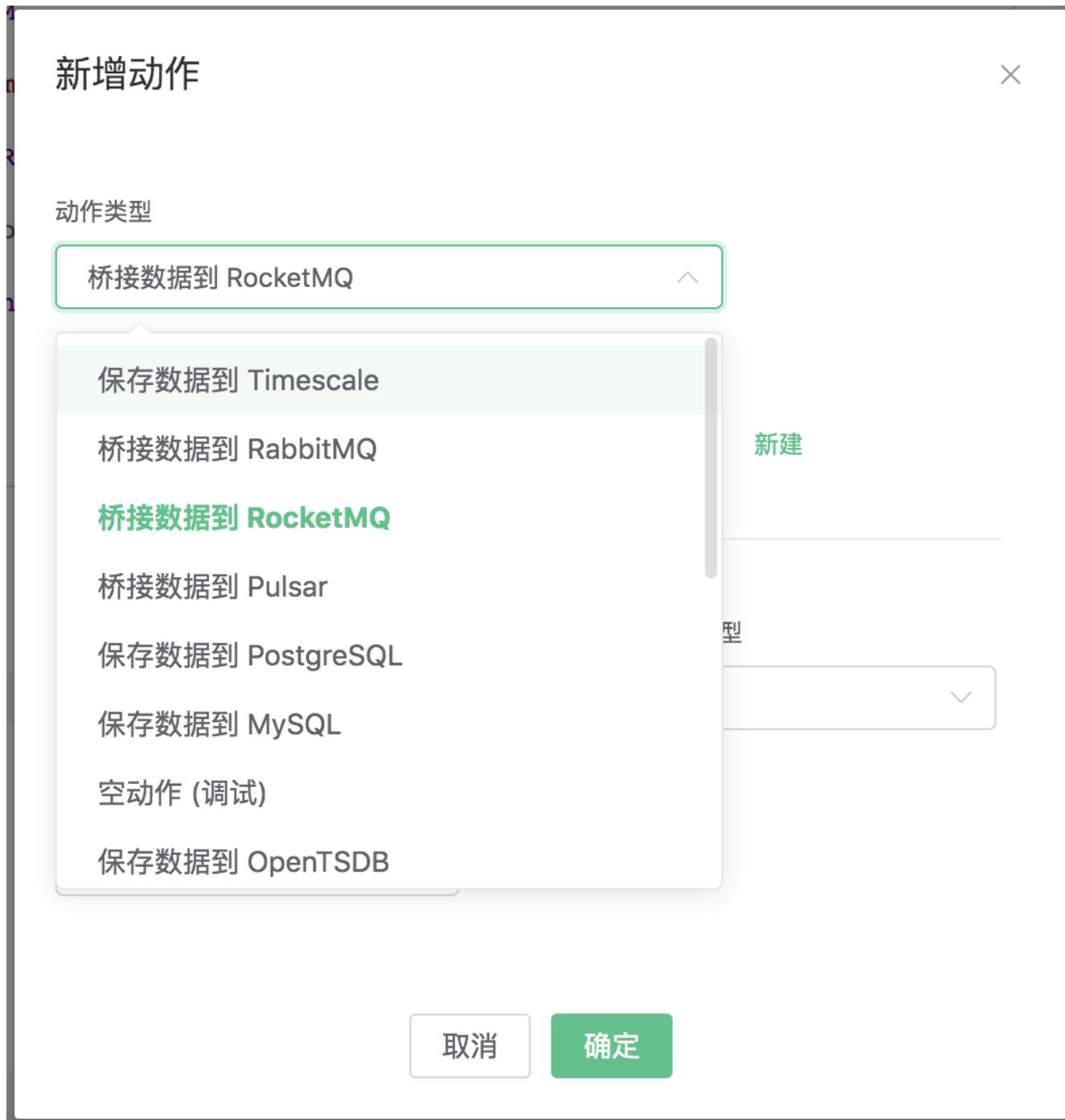
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'

```

Below the editor, there are input fields for "备注" (Remarks) and "SQL 测试" (SQL Test), along with a test button.

关联动作：

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 RocketMQ”。



“保存数据到 **RocketMQ** 动作需要两个参数:

1). **RocketMQ** 的消息主题

2). 关联资源。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 **RocketMQ** 资源:

新增动作

×

动作类型

桥接数据到 RocketMQ

* 使用资源

请选择

新建

* RocketMQ 主题

RocketMQ 主题

Produce 类型

sync

最大批处理数

100

取消

确定

填写资源配置：

填写真实的 **RocketMQ** 服务器地址，多个地址用逗号分隔，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。

创建资源

* 资源类型

RocketMQ

测试连接

备注

请输入

* RocketMQ 服务器

127.0.0.1:9876

同步调用超时时间

3s

Topic Route 更新间隔

3s

发送消息的缓冲区大小

1024KB

取消

确定

返回响应动作界面，点击“确认”。

新增动作

动作类型

桥接数据到 RocketMQ



* 使用资源

resource:55bda9e9



新建

* RocketMQ 主题

Produce 类型

TopicTest

sync



最大批处理数

100

取消

确定

返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 桥接数据到 RocketMQ (data_to_rocket)
桥接数据到 RocketMQ

编辑 移除

RocketMQ 主题 TopicTest Produce 类型 sync 最大批处理数 100 资源 ID resource:55bda9e9

+ 添加动作

取消

创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "hello"

```

然后通过 **RocketMQ** 命令去查看消息是否生产成功:

```

1 $ ./bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer TopicTest

```

```

denghaiguide@BP:rocketmq-all-4.5.2-bin-release denghaiguide$ ./bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
15:04:02.291 [main] DEBUG i.n.u.i..InternalLoggerFactory - Using SLF4J as the default logging framework
Consumer Started.
ConsumerMessageThread_1 Receive New Messages: [MessageExt [queueId=0, storeSize=355, queueOffset=0, sysFlag=0, bornTimestamp=1572505261255, bornHost=/127.0.0.1:53982, storeTim
estamp=1572505261283, storeHost=/127.0.0.1:10911, msgId=F0000010002AF000000000000000000, commitLogOffset=0, bodyCRC=617651257, reconsumeTimes=0, preparedTransactionOffset=0,
toString()=Message[topic='TopicTest', flag=0, properties={MIN_OFFSET=0, MAX_OFFSET=1, CONSUME_START_TIME=1572505442800}, body=[123, 34, 99, 108, 105, 101, 110, 116, 95, 105,
100, 34, 58, 34, 109, 113, 116, 116, 106, 115, 95, 50, 99, 49, 101, 51, 99, 100, 97, 34, 44, 34, 104, 34, 44, 34, 105, 100, 34, 58, 34, 53, 57, 54, 50, 70, 54, 57, 56, 48, 67, 52, 51, 55, 70, 52, 52, 51, 48, 48, 48, 48, 66, 54, 68, 48, 48, 49, 34, 44, 34, 118, 111, 100, 101, 34, 58, 34, 101, 109, 113, 120, 64, 49, 58, 55, 46, 48, 46, 49, 34, 44, 34, 112, 97, 121, 108, 111, 97, 100, 105, 101, 114, 110, 97, 109, 101, 34, 58, 34, 49, 50, 55, 46, 48, 46, 49, 34, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125], transactionId='null']]]
ConsumerMessageThread_2 Receive New Messages: [MessageExt [queueId=1, storeSize=338, queueOffset=0, sysFlag=0, bornTimestamp=1572505371432, bornHost=/127.0.0.1:53981, storeTim
estamp=1572505371434, storeHost=/127.0.0.1:10911, msgId=F0000010002AF00000000000000163, commitLogOffset=355, bodyCRC=2060229905, reconsumeTimes=0, preparedTransactionOffset
=0, toString()=Message[topic='TopicTest', flag=0, properties={MIN_OFFSET=0, MAX_OFFSET=1, CONSUME_START_TIME=1572505442800}, body=[123, 34, 99, 108, 105, 101, 110, 116, 95, 105,
100, 34, 58, 34, 109, 113, 116, 116, 106, 115, 95, 50, 99, 49, 101, 51, 99, 100, 97, 34, 44, 34, 104, 34, 44, 34, 105, 100, 34, 58, 34, 53, 57, 54, 50, 70, 55, 48, 49, 50, 53, 51, 50, 70, 70, 52, 52, 51, 48, 48, 48, 48, 66, 54, 68, 48, 48, 50, 34, 44, 34, 110, 111, 100, 101, 34, 58, 34, 101, 109, 113, 120, 64, 49, 50, 55, 46, 48, 46, 49, 34, 44, 34, 112, 97, 121, 108, 111, 97, 100, 105, 101, 34, 58, 34, 49, 50, 55, 46, 48, 46, 49, 34, 114, 115, 116, 97, 109, 112, 34, 58, 49, 53, 55, 50, 53, 48, 53, 51, 55, 49, 52, 51, 50, 4
0, 41, 101, 108, 108, 111, 34, 44, 34, 112, 101, 101, 114, 110, 97, 109, 101, 34, 58, 34, 49, 50, 55, 46, 48, 46, 49, 34, 116, 105, 109, 101, 115, 116, 97, 109, 112, 34, 58, 49, 53, 55, 50, 53, 48, 53, 51, 55, 49, 52, 51, 50, 4
4, 34, 116, 111, 112, 105, 99, 34, 58, 34, 116, 47, 49, 34, 44, 34, 117, 115, 101, 114, 110, 97, 109, 101, 34, 58, 34, 34, 125], transactionId='null']]
```

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1:

The screenshot shows the EMQX Enterprise Rule Engine interface. On the left, there's a sidebar with navigation links: 首页 / 规则引擎, 快速开始, 产品文档. Below it is a search bar and a '创建' (Create) button. The main area displays a table of rules. One rule is highlighted: **rule:a964f4c1**. Its details are shown on the right:

- 规则统计 (Rule Statistics):**
 - 命中次数 (Matched Count): 2 次
 - 当前速度 (Current Rate): 0 次/秒
 - 最大执行速度 (Max Execution Rate): 0.1 次/秒
 - 最近5分钟执行速度 (Last 5 Minutes Rate): 0.01 次/秒
- 动作统计 (Action Statistics):**
 - data_to_rocket (data_to_rocket):
 - 成功 (Success): 2
 - 失败 (Failure): 0

ID	监控	描述	触发事件
rule:a964f4c1	all		message.publish

桥接数据到 SAP Event Mesh

EMQX 规则引擎支持通过 **HTTP** 请求方式 (`httprest`) 将消息发送到 [SAP Event Mesh](#)。

Event Mesh 是 [SAP BTP](#) 重要的消息交换组件。 **SAP BTP** 囊括了 **SAP** 的所有技术组合，例如 **SAP HANA**（内存计算平台）、**SAP Analytics Cloud**（分析云）、**SAP Integration** 套件（集成套件）和 **SAP Extension** 套件（扩展套件）。

EMQX 的物联网数据可以通过此通道进入到 **SAP BTP** 平台的诸多产品中。

准备 SAP Event Mesh 环境

准备 **SAP Event Mesh** 环境，并获取 **Service Keys**。

相关操作步骤见 [Create Instance of SAP Event Mesh](#)。

以下面的 **Service Keys** 为例：

```

1  {
2      "xsappname": "some-app-name",
3      "management": [
4          {
5              "oa2": {
6                  ...
7              },
8              "uri": "..."
9          }
10     ],
11     "messaging": [
12         {
13             "oa2": {
14                 "clientid": "my_clientid",
15                 "clientsecret": "my_clientsecret",
16                 "tokenendpoint": "https://123trial.authentication.demo.com/oauth/token",
17                 "granttype": "client_credentials"
18             },
19             "protocol": [
20                 "amqp10ws"
21             ]
22         },
23         {
24             "oa2": {
25                 "clientid": "my_clientid",
26                 "clientsecret": "my_clientidsecret",
27                 "tokenendpoint": "https://123trial.authentication.demo.com/oauth/token",
28                 "granttype": "client_credentials"
29             },
30             "protocol": [
31                 "httprest"
32             ],
33             "broker": {
34                 "type": "saprestmgw"
35             },
36             "uri": "https://sap-messaging-pubsub.fooapps.demo.com"
37         }
38     ],
39     "serviceinstanceid": "188783-7893-8765-8872-77866"
40 }

```

我们只关心 **Service Keys** 里面的 `"messaging"` 字段，我们可以获取到以下与 `protocol: ["httprest"]` 相关的信息：

Key	Value
Token Endpoint URI	https://123trial.authentication.demo.com/oauth/token
Send Message URI	https://sap-messaging-pubsub.fooapps.demo.com
ClientId	my_clientid
ClientSecret	my_clientsecret

在 **SAP Event Mesh** 平台创建一个消息队列，名字为：`"my_queue_name"`。

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL:

```
1 SELECT  
2   *  
3 FROM  
4   "#"
```

* SQL 输入:

```
1 SELECT  
2   *  
3 FROM  
4   "#"  
5  
6  
7  
8
```

* 规则 ID:

rule:619224

关联动作:

在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“发送数据到 **SAP Event Mesh**”。

新增动作

* 动作类型

数据转发
发送数据到 SAP Event Mesh

* 使用资源 ?

请选择
新建

* 队列名 ?

my_queue_name
QoS ?

消息内容 ?

\${payload}

取消
确定

“发送数据到 SAP Event Mesh” 动作需要填写以下几个参数：

1). 消息内容模板。这个例子里我们向 **SAP Event Mesh** 发送一条数据，消息模板为：

1	\${payload}
---	-------------

2). 队列名。这里填写我们刚才在 **SAP Event Mesh** 平台创建的消息队列名字：“**my_queue_name**”。

3). QoS。选择 **SAP Event Mesh** 的 QoS 级别。

4). 关联资源。现在资源下拉框为空，可以点击旁边的“新建”来创建一个 **SAP Event Mesh** 资源：

填写资源配置：

这里需要填写我们通过 **Service Keys** 获取到的信息，包括 `Token Endpoint URI`，`ClientId`，`ClientSecret`，`Send Message URI` 等。

其他参数保持默认，然后点击“测试连接”按钮，确保连接测试成功。最后点击“确定”按钮。

创建资源

*** 资源类型**

SAP Event Mesh ▼ 测试连接

*** 资源 ID**

resource:046912 描述 请输入

协议 ?

httprest ▼

*** Token Endpoint URI** ?

https://123trial.authenticati...

*** ClientId** ?

my_clientid

*** ClientSecret** ?

my_clientsecret 🔑 👁️

*** Send Message URI** ?

https://sap-messaging-pub...

HTTP 连接超时时间

15s

HTTP 请求超时时间 ?

15s

HTTP 连接池大小 ?

8 ↑ ↓

Enable HTTP Pipelining ?

true ▼

返回响应动作界面，点击“确定”。

新增动作

* 动作类型

数据转发 发送数据到 SAP Event Mesh

* 使用资源 ?

resource:046912 新建

队列名 ?

my_queue_name

QoS ?

1

消息内容 ?

```
`${payload}`
```

取消 确定

返回规则创建界面，点击“创建”。

* 响应动作

处理命中规则的消息



+ 添加动作

取消

创建

发送消息测试

规则已经创建完成，现在使用 **MQTT** 客户端向 **emqx** 发送一条数据：

```
1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "abc"
```

现在您可以到 **SAP Event Mesh** 平台，检查是否可以从 "**my_queue_name**" 这个队列消费到我们刚才发送的数据：**"abc"**。

桥接数据到 MQTT Broker

搭建 MQTT Broker 环境，以 MacOS X 为例：

```
1 $ brew install mosquitto  
2  
3 启动 mosquitto  
4 $ mosquitto
```

sh

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 SQL：

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

* SQL 输入:

```
1 SELECT
2 *
3 FROM
4 "t/#"
```

当前事件可用字段

```
event_id clientid username payload peerhost topic qos
flags headers publish_received_at timestamp node
```

规则 SQL 示例

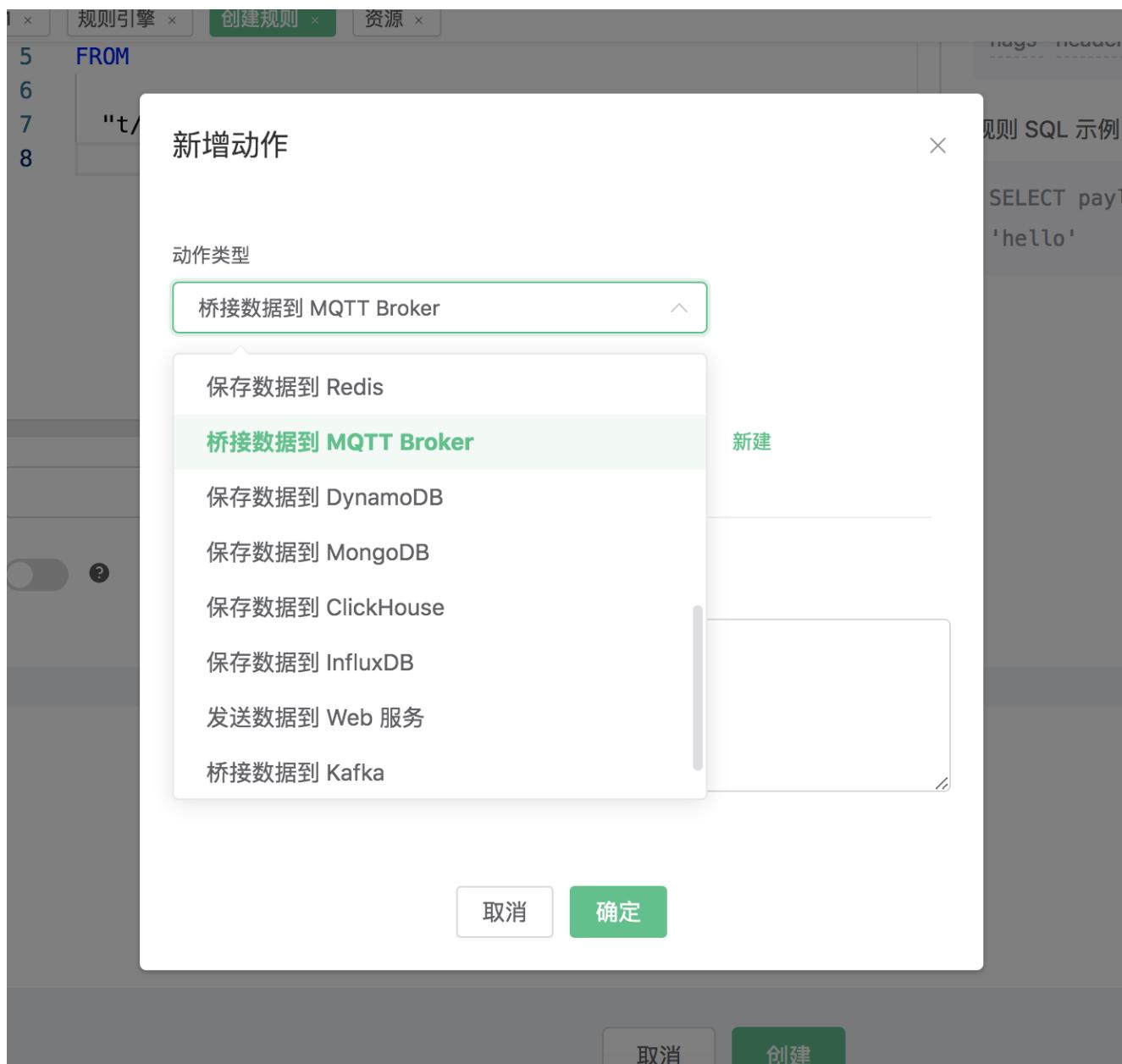
```
SELECT payload.msg as msg FROM "t/#" WHERE msg =
'hello'
```

备注:

SQL 测试:

关联动作：

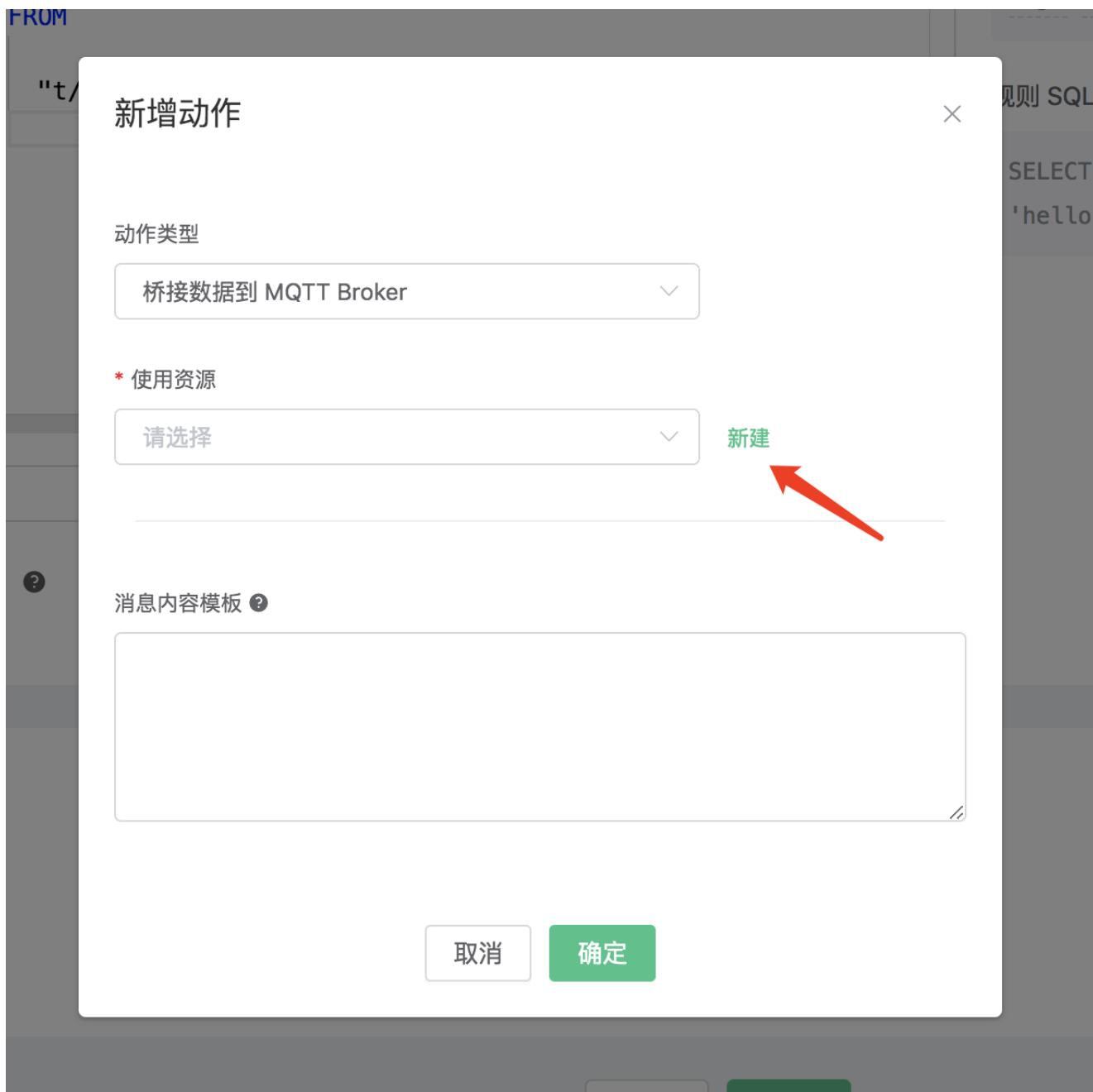
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 MQTT Broker”。



Fill in the action parameters:

"Bridge data to MQTT Broker" action only needs one parameter:

Associate resources. Now the resource dropdown is empty, you can click the "Create Resource" button in the top right corner to create a **MQTT Bridge** resource:



选择 **MQTT Bridge** 资源, 填写资源配置:

填写真实的 **mosquitto** 服务器地址, 其他配置保持默认值, 然后点击“测试连接”按钮, 确保连接测试成功。

注意:

附加 **GUID** 选项, 设置为 `true` 时, **MQTT** 连接使用的 **clientid** 增加随机后缀以保证全局唯一性。设置为 `false` 时, 会导致 **clientid** 使用同一个, 连接池中线程互踢, **EMQX** 多个节点之间的桥接也会互踢, 推荐仅在单节点 **EMQX** 且连接池大小为 1 时开启此选项。

最后点击“新建”按钮。

创建资源

* 资源类型

MQTT Bridge

测试连接

* 资源名称

MQTT Bridge

* 远程 broker 地址

127.0.0.1:1883

* 连接池大小

8

* 客户端 Id

client

附加 GUID ?

true

用户名

连接远程 Broker 的用户名

密码

连接远程 Broker 的密码

桥接挂载点 ?

bridge/aws/\${node}/

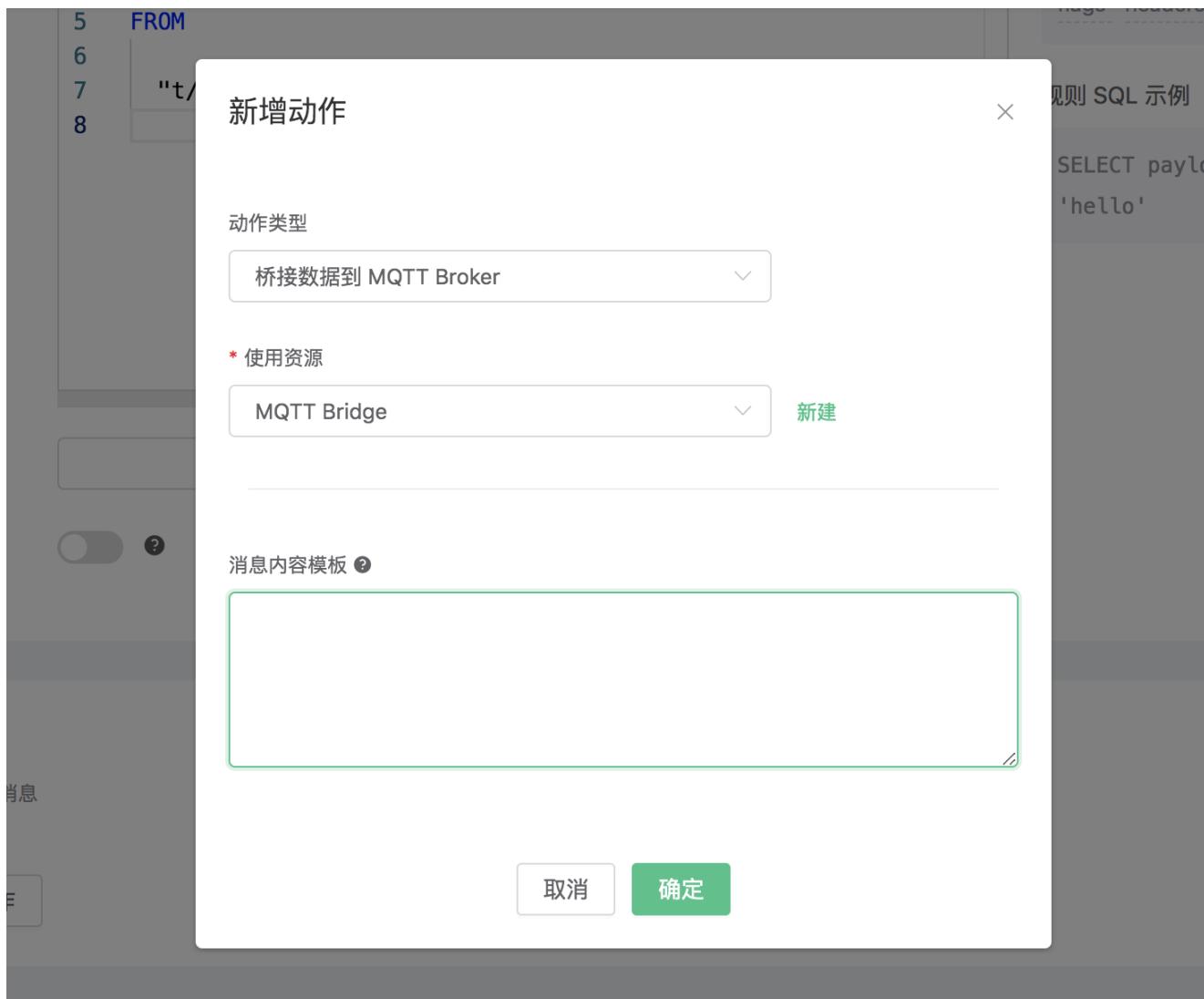
磁盘缓存 ?

off

展开 ▼

取消 确定

返回响应动作界面，点击“确认”。



返回规则创建界面，点击“新建”。

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "Hello, World\!"
```

然后通过 **mqtt** 客户端查看消息是否发布成功

The screenshot shows the EMQX Enterprise V4.4 Docs interface. At the top left, there is a search bar with the placeholder "bridge/#". Below it, a section titled "服务质量" (Service Quality) shows a value of "2". A green "订阅" (Subscribe) button is visible. In the center, there is a "消息" (Message) section with fields for "主题" (Topic) set to "testtopic", "消息" (Message) set to "{ "msg": "Hello, World!" }", and "服务质量" (Service Quality) set to "0". A "发送" (Send) button is located to the right. Below this, there are two tables: "发布消息列表" (Publish Message List) and "订阅消息列表" (Subscribe Message List). The "发布消息列表" table has one row with the message content: {"client_id": "mqqtjs_de0b9e615e", "even_t": "message.publish", "id": "58E7DFOF977B6F440000009660001", "node": "emqx@127.0.0.1", "payload": "Hello, Worl", "peername": "127.0.0.1:56430", "qos": 1, "retain": 0, "timestamp": 156404644849, "topic": "t/1", "username": ""}. The "订阅消息列表" table also has one row with the message content: {"client_id": "mqx@127.0.0.1", "even_t": "message.subscribe", "id": "1", "node": "emqx@127.0.0.1", "payload": "Hello, Worl", "peername": "127.0.0.1:56430", "qos": 1, "retain": 0, "timestamp": 156404644849, "topic": "t/1", "username": ""}.

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

The screenshot shows the rule list interface. It features a table with columns: ID, 主题 (Topic), 监控 (Monitoring), 描述 (Description), 状态 (Status), and 响应动作 (Response Action). A red arrow points to the "命中" (Matched) value in the "监控" column for the first rule, which is "rule:9a4a5db2" and has a topic of "t/#". The "命中" value is currently "0". To the right of the table, there is a button labeled "桥接数据到 MQTT Broker" (Bridge data to MQTT Broker).

桥接数据到 RPC 服务

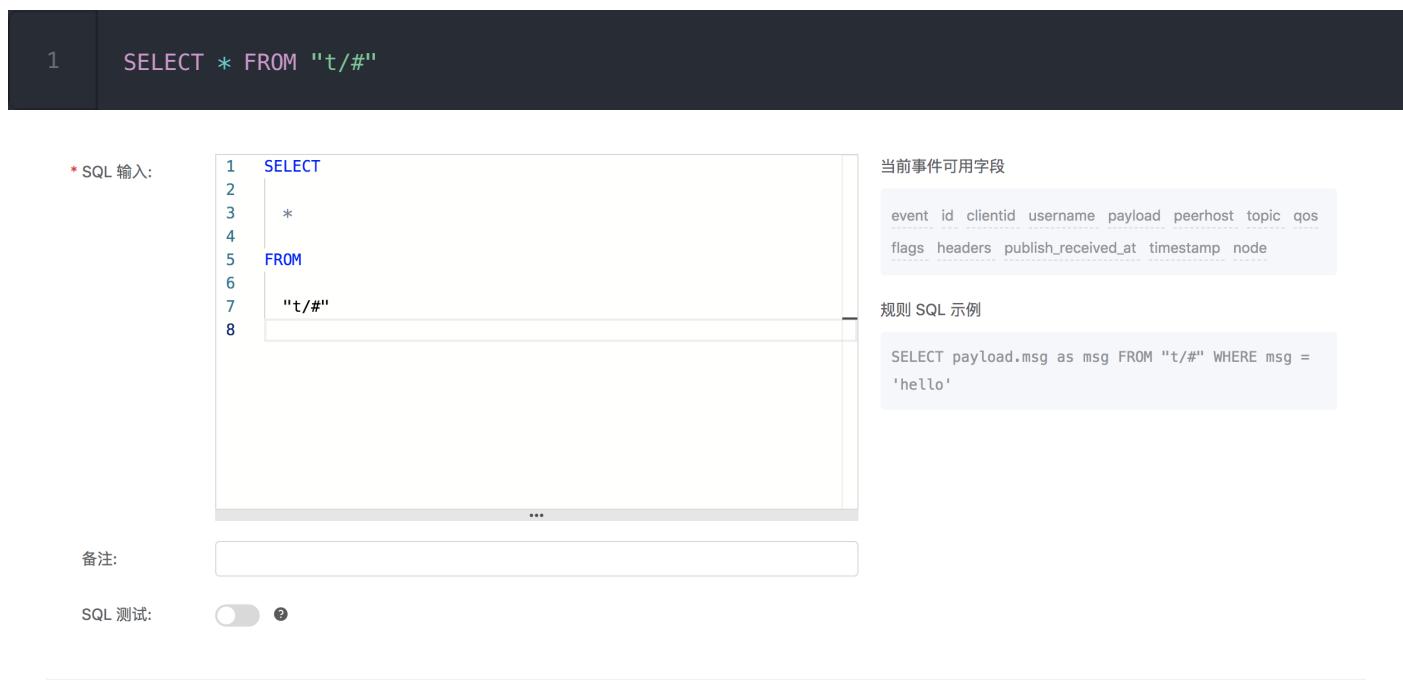
搭建 **EMQX Broker** 环境，以 **MacOS X** 为例：

```
1 $ brew tap emqx/emqx/emqx  
2  
3 $ brew install emqx  
4  
5 # 启动 emqx  
6 $ emqx console
```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

填写规则 **SQL**：



The screenshot shows the 'Rules' section of the EMQX Dashboard. On the left, there's a code editor for writing SQL queries. The current query is:

```
1 * SQL 输入:  
2 SELECT  
3 *  
4 FROM  
5 "t/#"  
6
```

To the right of the editor, there are two panels: '当前事件可用字段' (Available fields for current event) and '规则 SQL 示例' (Rule SQL example). The available fields panel lists:

```
event id clientid username payload peerhost topic qos  
flags headers publish_received_at timestamp node
```

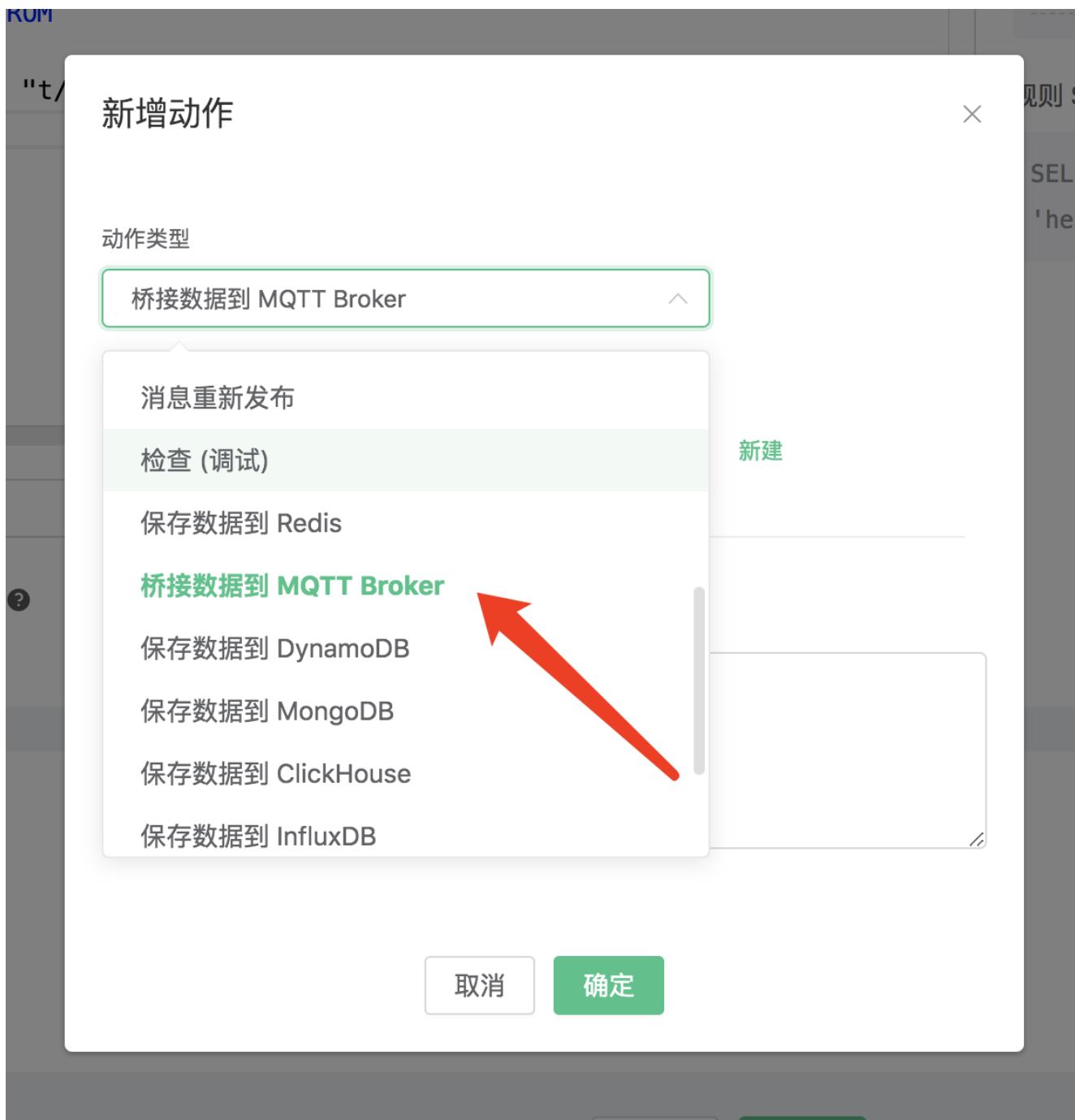
The rule SQL example panel shows:

```
SELECT payload.msg as msg FROM "t/#" WHERE msg = 'hello'
```

Below the editor, there are '备注:' (Remarks) and 'SQL 测试:' (SQL Test) buttons.

关联动作：

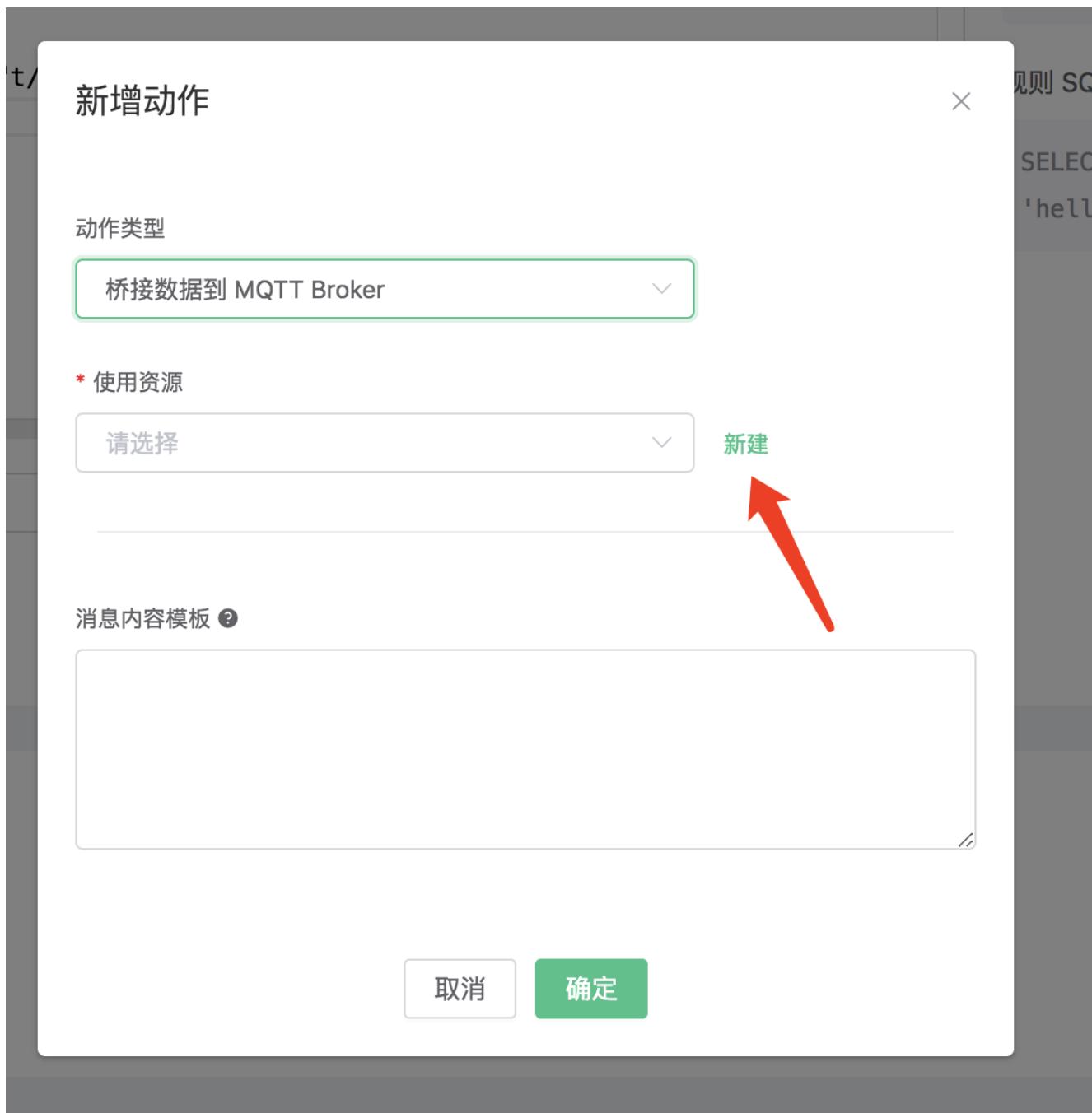
在“响应动作”界面选择“添加”，然后在“动作”下拉框里选择“桥接数据到 MQTT Broker”。



填写动作参数:

桥接数据到 **MQTT Broker** 动作只需要一个参数:

关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **RPC Bridge** 资源:

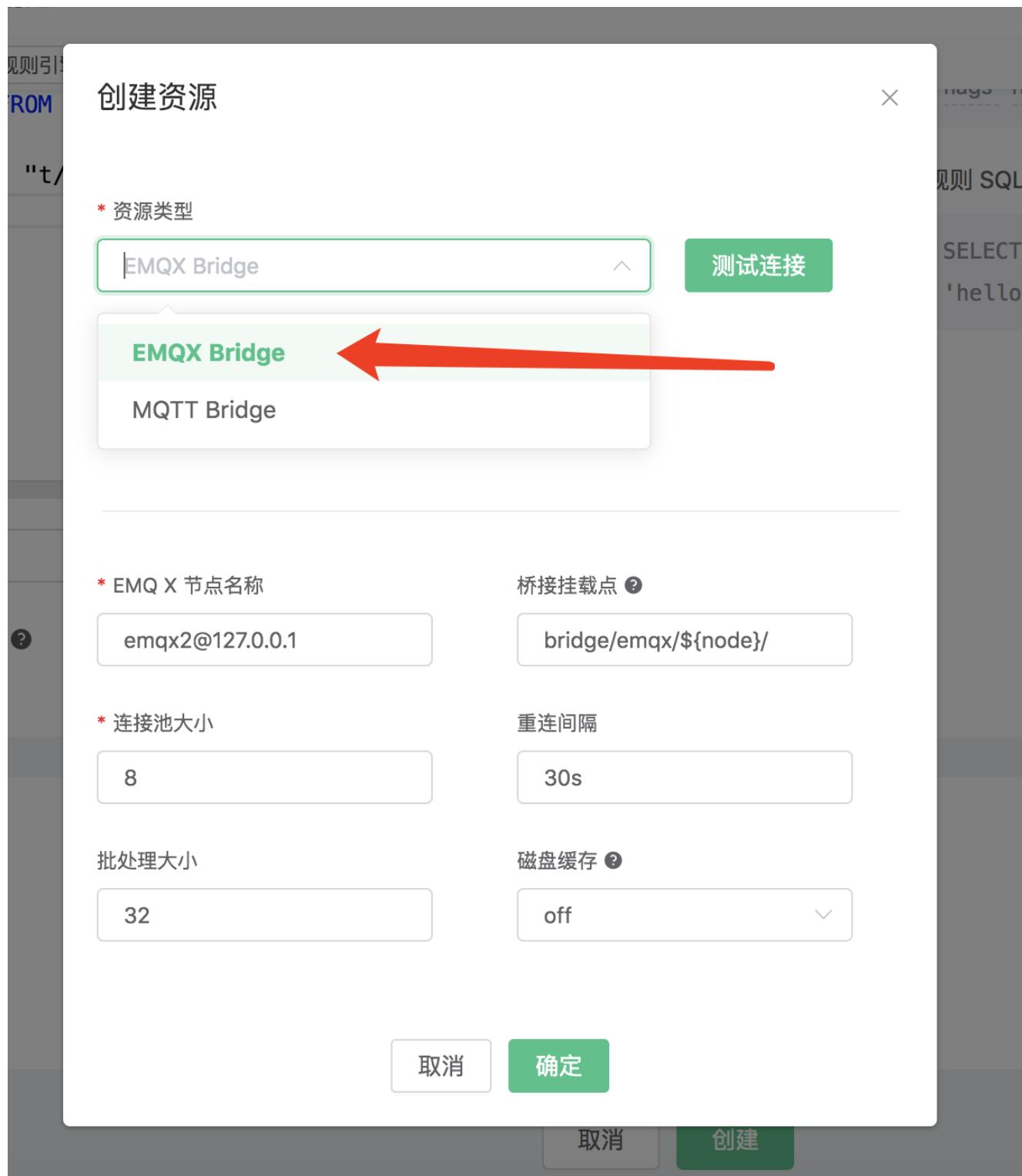


选择 **RPC Bridge** 资源。

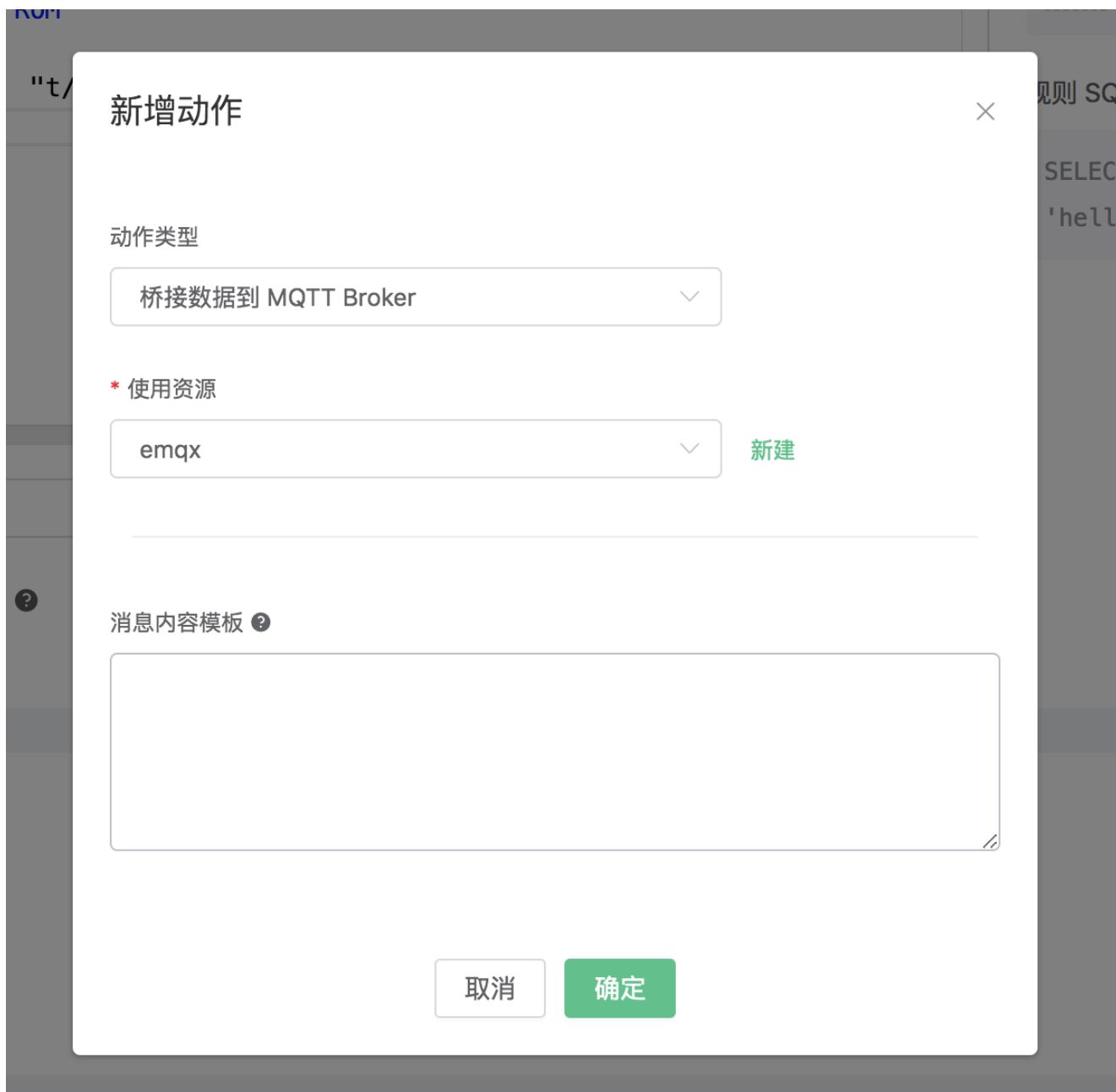
填写资源配置：

填写真实的 **emqx** 节点名，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。



返回响应动作界面，点击“确认”。



返回规则创建界面，点击“新建”。

响应动作 *

处理命中规则的消息

动作类型 桥接数据到 MQTT Broker (data_to_mqtt_broker)

编辑 移除

桥接数据到 MQTT Broker

消息内容模板 资源 ID resource:ed4baace

+ 失败备选动作

+ 添加动作

取消

创建

规则已经创建完成，现在发一条数据：

```

1 Topic: "t/1"
2
3 QoS: 0
4
5 Payload: "Hello, World\!"
```

然后通过 **mqtt** 客户端查看消息是否发布成功

The screenshot shows the EMQX Enterprise interface with the following sections:

- 桥接列表** (bridge/#) at the top left.
- 服务质量** (Quality of Service) dropdown set to 1.
- 订阅** (Subscribe) button.
- 消息** (Messages) section with a table:

主题	消息	服务质量	操作
testtopic	{ "msg": "Hello, World!" }	0	<input checked="" type="checkbox"/> 保留 <button>发送</button>
- 发布消息列表** (Published Messages) and **订阅消息列表** (Subscribed Messages) sections.
- 发布消息列表** table:

消息	主题	服务质量	时间
暂无数据			
- 订阅消息列表** table:

消息	主题	服务质量	时间
{"client_id": "mqttjs_3fc2638ae7", "even": true, "id": "58E7E0855D", "topic": "bridge/aws/e", "timestamp": 156404684042, "topic_id": 1, "username": "127.0.0.1", "qos": 1, "retain": 0, "payload": "Hello, World!", "peername": "127.0.0.1:5670", "seq": 1, "time": "2019-07-25 17:27:20", "type": "puback"}, {"client_id": "mqttjs_3fc2638ae7", "even": true, "id": "58E7E0855D", "topic": "bridge/aws/e", "timestamp": 156404684042, "topic_id": 1, "username": "127.0.0.1", "qos": 1, "retain": 0, "payload": "Hello, World!", "peername": "127.0.0.1:5670", "seq": 1, "time": "2019-07-25 17:27:20", "type": "puback"}	bridge/aws/e	1	2019-07-25 17:27:20
			1/t/1

在规则列表里，可以看到刚才创建的规则的命中次数已经增加了 1：

The screenshot shows the Rules list with the following details:

ID	主题	监控	描述	状态	响应动作
rule:9b024f56	t/#	1/1		<input checked="" type="checkbox"/>	桥接数据到 MQTT Broker <button>编辑</button> <button>删除</button>

A red arrow points to the '1/1' value under the '监控' (Monitoring) column, indicating one successful match for the rule.

离线消息保存到 Redis

搭建 Redis 环境，以 MacOS X 为例：

```

1   $ wget http://download.redis.io/releases/redis-4.0.14.tar.gz
2   $ tar xzf redis-4.0.14.tar.gz
3   $ cd redis-4.0.14
4   $ make && make install
5
6   # 启动 redis
7   $ redis-server

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 SQL：

FROM说明

t/#: 发布者发布消息触发保存离线消息到Redis

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息ACK后触发删除已经被接收的离线消息

```

1   SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't
/#'

```

* SQL 输入:

```

1  SELECT
2    *
3  FROM
4    "t/#",
5    "$events/session_subscribed",
6    "$events/message_acked"
7  WHERE
8    topic =~ 't/#'

```

当前事件可用字段

event	clientid	username	peerhost	topic	qos	timestamp
node						

规则 SQL 示例

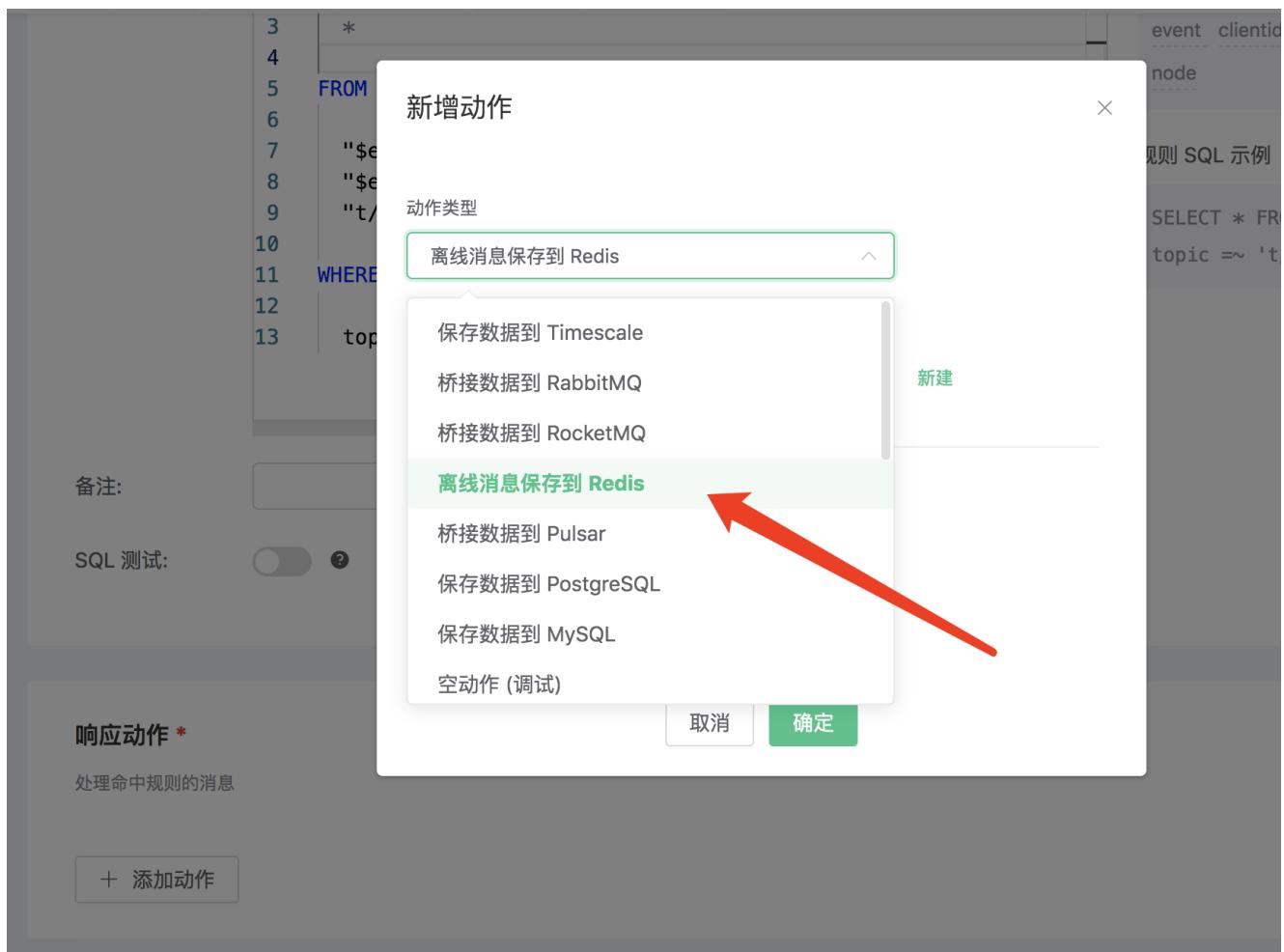
```
SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'
```

备注:

SQL 测试:

关联动作：

在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 Redis”。

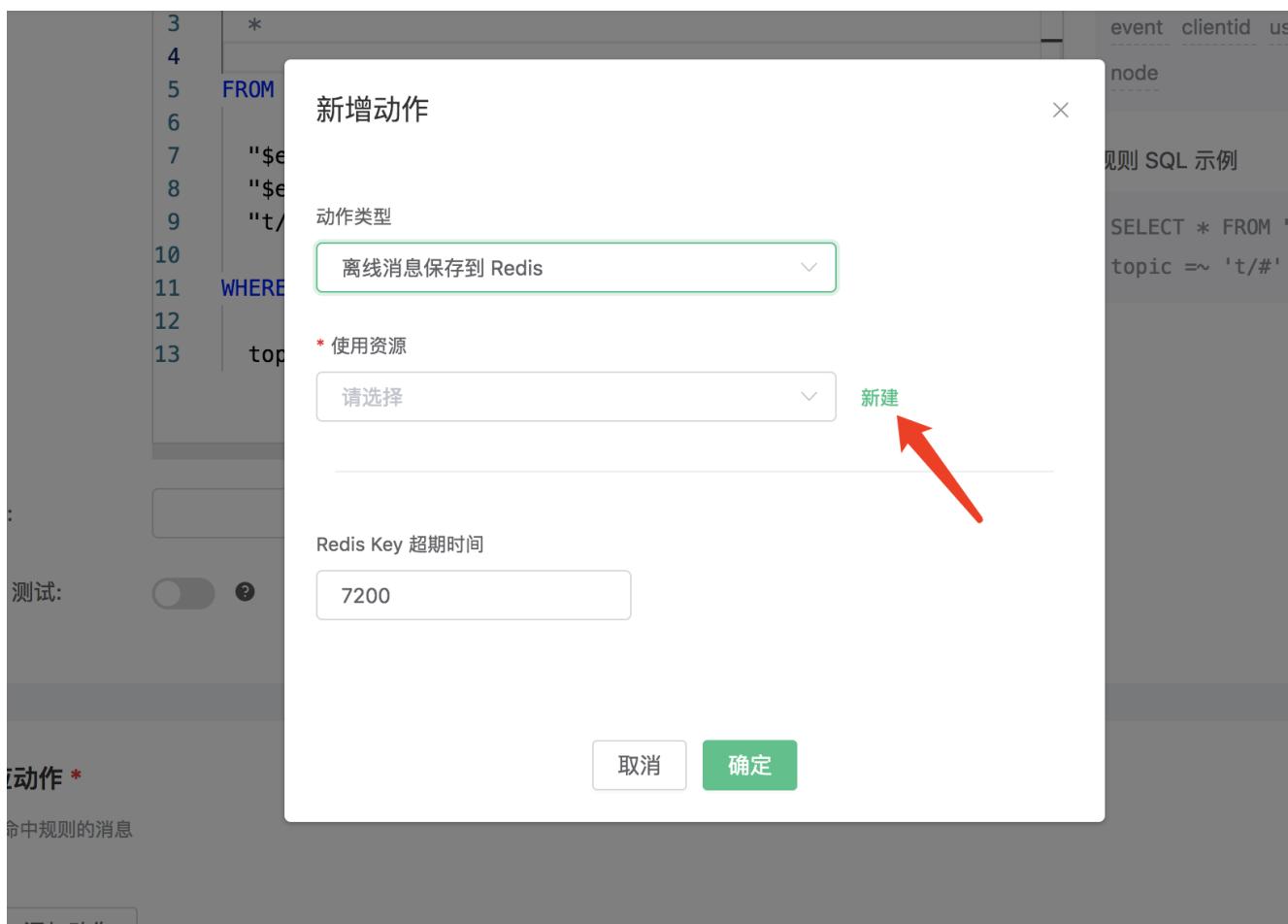


填写动作参数：

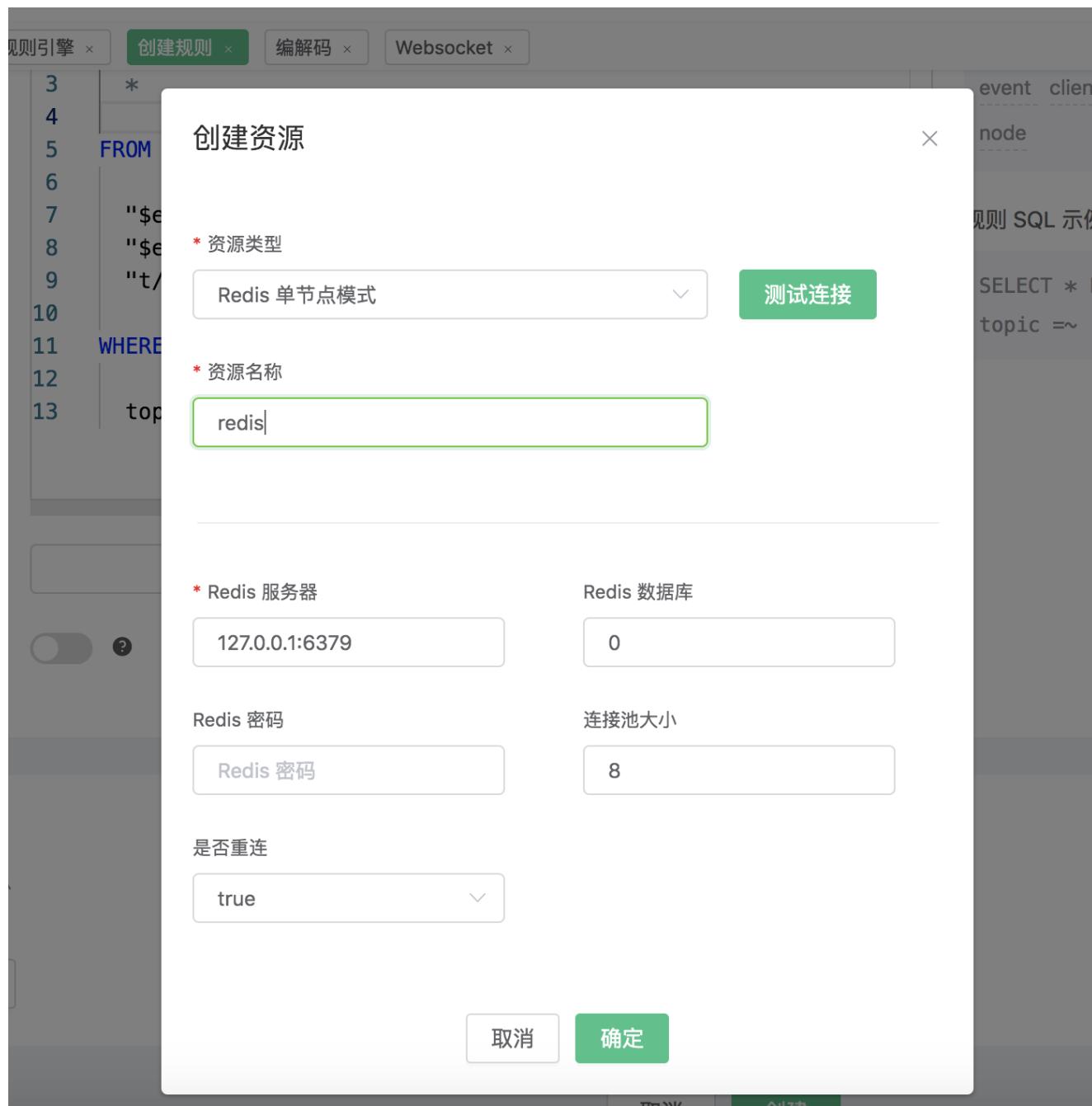
“离线消息保存到 Redis” 动作需要两个参数：

1). **Redis Key** 超期的 TTL

2). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **Redis** 资源：



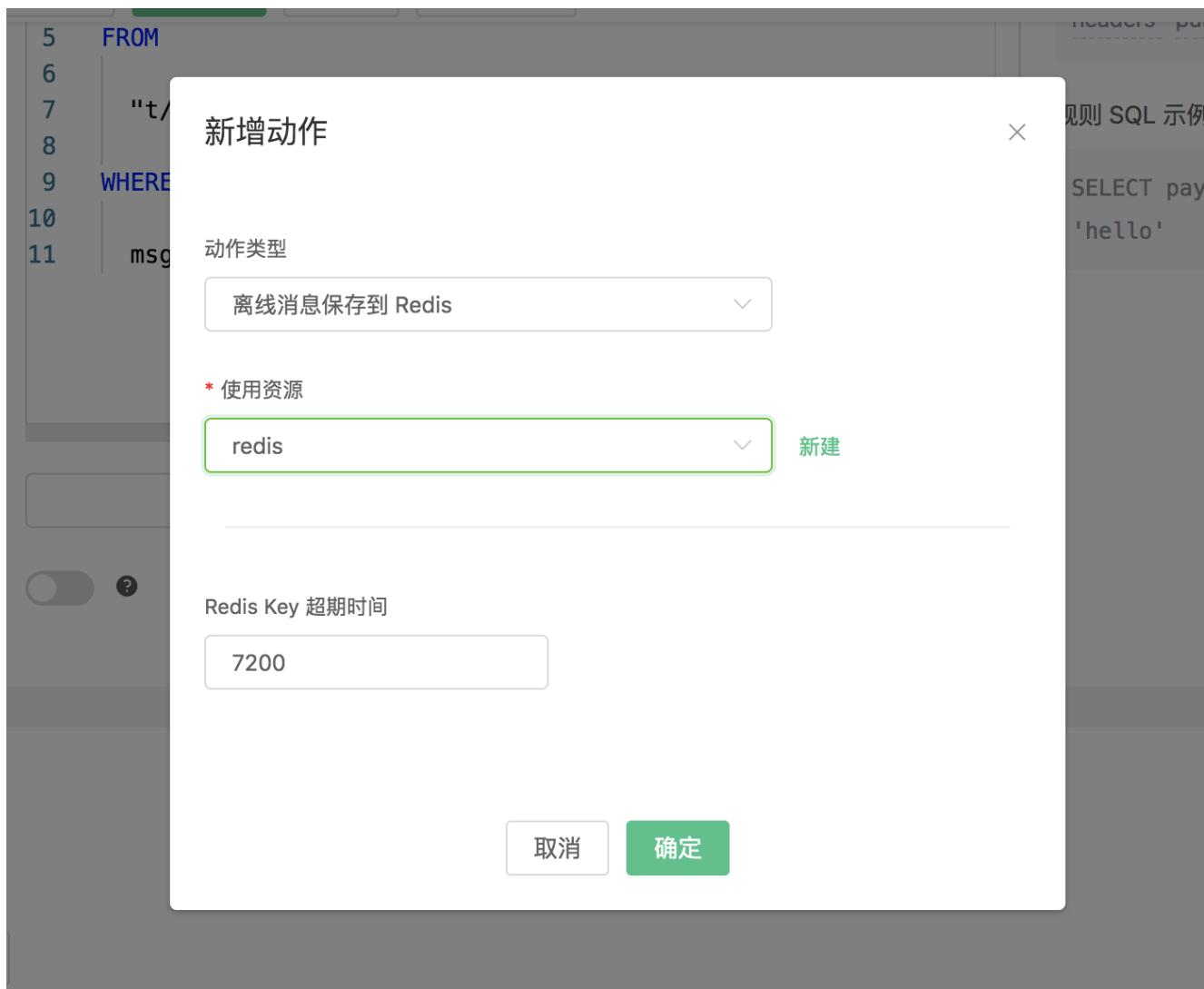
选择 **Redis** 单节点模式资源”。



填写资源配置：

填写真实的 **Redis** 服务器地址，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 离线消息保存到 Redis (offline_msg_to_redis)

Redis Key 超期时间 7200

编辑 移除

+ 失败备选动作

+ 添加动作

取消 确定

返回规则创建界面，点击“新建”。

ID	主题	监控	描述	状态	响应动作
rule:63f71807	\$events/message_ack ed\$events/session_su bscribedt/#	ull		<input checked="" type="checkbox"/>	离线消息保存到 Redis 编辑 删除

规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据**(发布消息的**QoS**必须大于**0**)**:

Topic	QoS	Payload	时间
t/1	1	hello	10:09:10

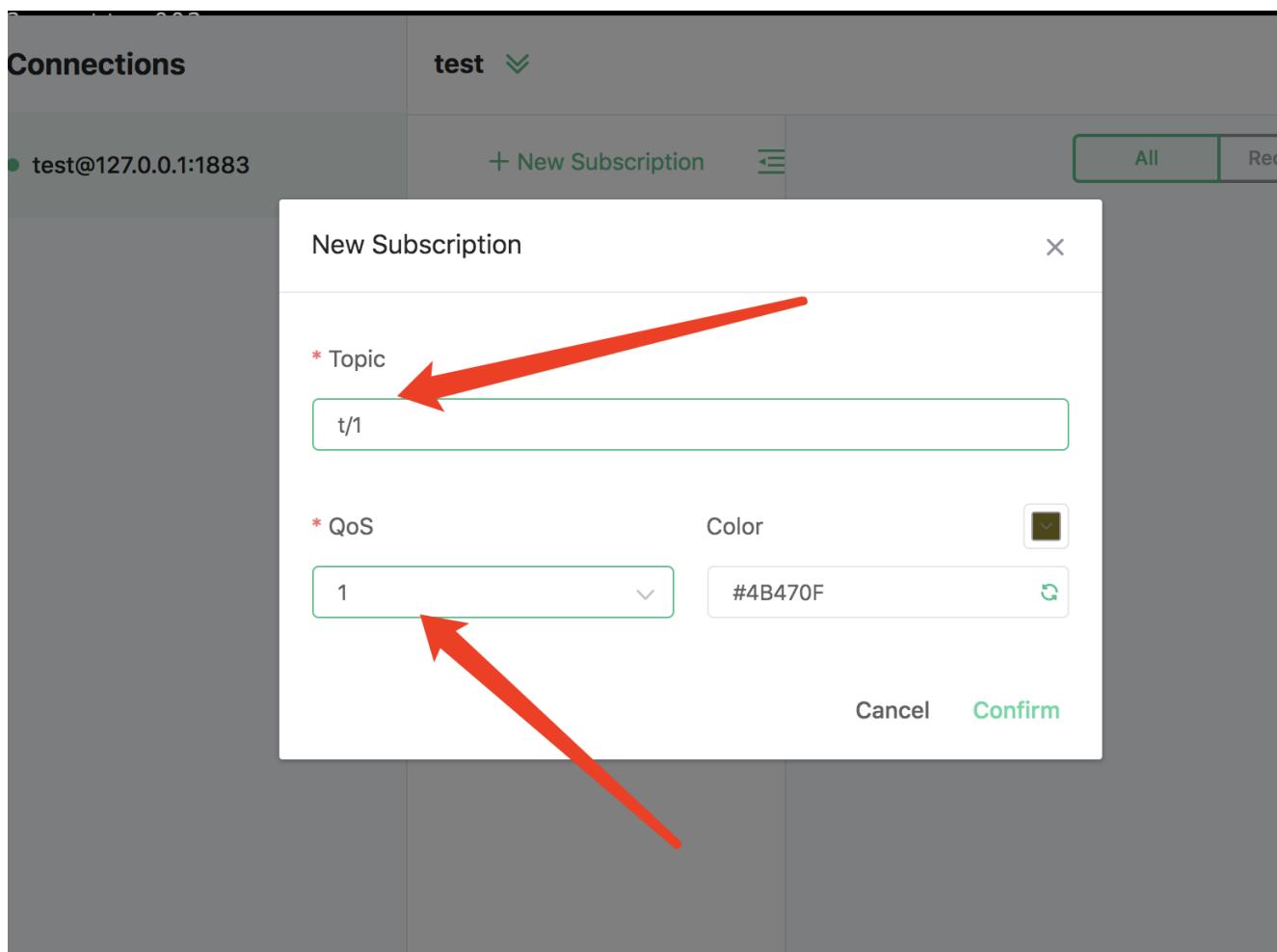
消息发送后，通过 **Redis CLI** 查看到消息被保存到 **Redis** 里面:

```

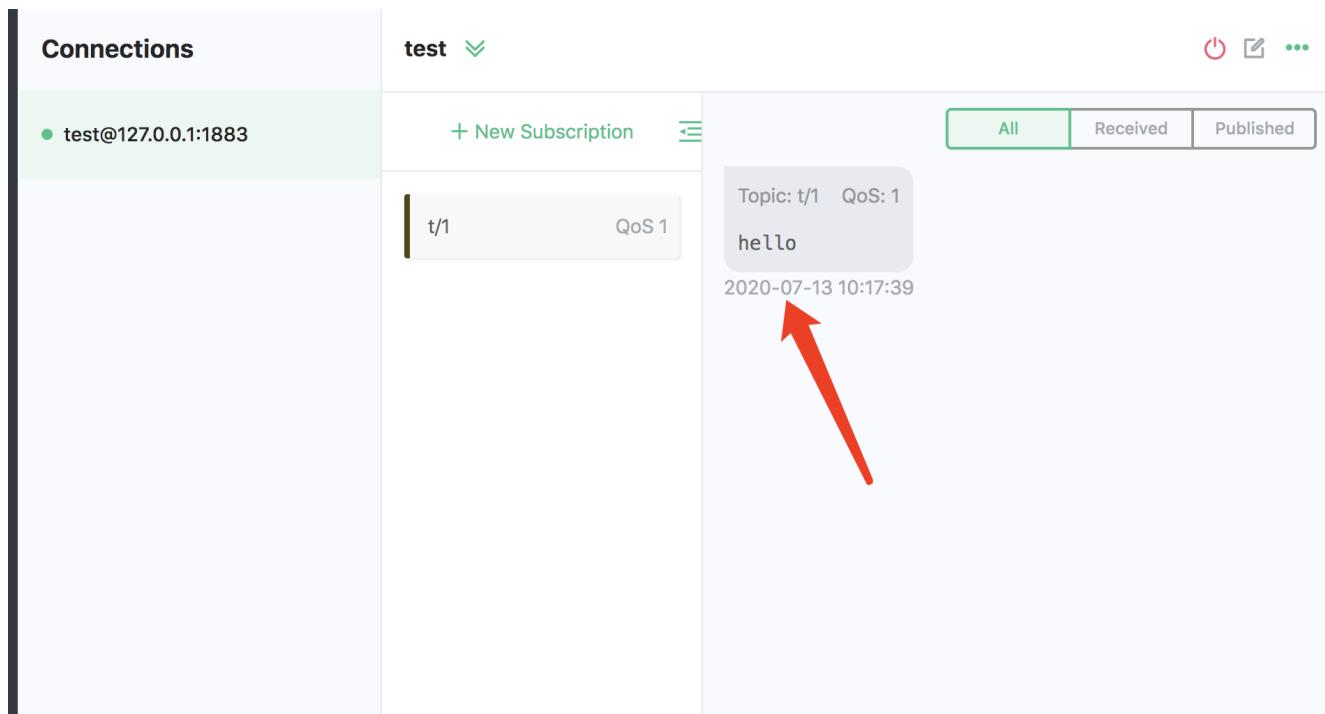
1 $ redis-cli
2
3 KEYS mqtt:msg\*
4
5 hgetall Key
sh
127.0.0.1:6379>
127.0.0.1:6379> KEYS *
1) "mqtt:msg:Mjk0MTUyOTE1NjA10DYxNjAzMTEzMDC4NzE4NDk5Mzg5NDF"
2) "mqtt:msg:t/1"
127.0.0.1:6379> hgetall "mqtt:msg:Mjk0MTUyOTE1NjA10DYxNjAzMTEzMDC4NzE4NDk5Mzg5NDF"
1) "id"
2) "Mjk0MTUyOTE1NjA10DYxNjAzMTEzMDC4NzE4NDk5Mzg5NDF"
3) "from"
4) "mqttjs_362f6942"
5) "qos"
6) "1"
7) "topic"
8) "t/1"
9) "payload"
10) "hello"
11) "ts"
12) "1594606150714"
13) "retain"
14) "false"
127.0.0.1:6379> ttl "mqtt:msg:Mjk0MTUyOTE1NjA10DYxNjAzMTEzMDC4NzE4NDk5Mzg5NDF"
(integer) 6806
127.0.0.1:6379> 

```

使用另外一个客户端，订阅主题 "**t/1**" (订阅主题的**QoS**必须大于**0**，否则消息会被重复接收，不支持主题通配符方式订阅获取离线消息):



订阅后马上接收到了保存到 **Redis** 里面的离线消息:



离线消息被接收后会在 **Redis** 中删除:

```
Last login: Mon Jul 13 09:55:00 on ttys003
➔ redis redis-cli
127.0.0.1:6379> KEYS *
(empty list or set)
127.0.0.1:6379> █
```

离线消息保存到 MySQL

搭建 MySQL 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```

1 $ brew install mysql
2
3 $ brew services start mysql
4
5 $ mysql -u root -h localhost -p
6
7 ALTER USER 'root'@'localhost' IDENTIFIED BY 'public';

```

初始化 MySQL 数据库：

```

1 $ mysql -u root -h localhost -ppublic
2
3 create database mqtt;

```

创建 **mqtt_msg** 表：

```

1 DROP TABLE IF EXISTS `mqtt_msg`;
2 CREATE TABLE `mqtt_msg` (
3     `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4     `msgid` varchar(64) DEFAULT NULL,
5     `topic` varchar(180) NOT NULL,
6     `sender` varchar(64) DEFAULT NULL,
7     `qos` tinyint(1) NOT NULL DEFAULT '0',
8     `retain` tinyint(1) DEFAULT NULL,
9     `payload` blob,
10    `arrived` datetime NOT NULL,
11    PRIMARY KEY (`id`),
12    INDEX topic_index(`id`, `topic`)
13 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;

```

提示

消息表结构不能修改，请使用上面SQL语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 SQL：

FROM说明

t/#: 发布者发布消息触发保存离线消息到MySQL

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息ACK后触发删除已经被接收的离线消息

```
1 SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't/#'
```

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4     "t/#",
5     "$events/session_subscribed",
6     "$events/message_acked"
7   WHERE
8     topic =~ 't/#'
9
```

当前事件可用字段

event clientid username peerhost topic qos timestamp node

规则 SQL 示例

```
SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'
```

备注:

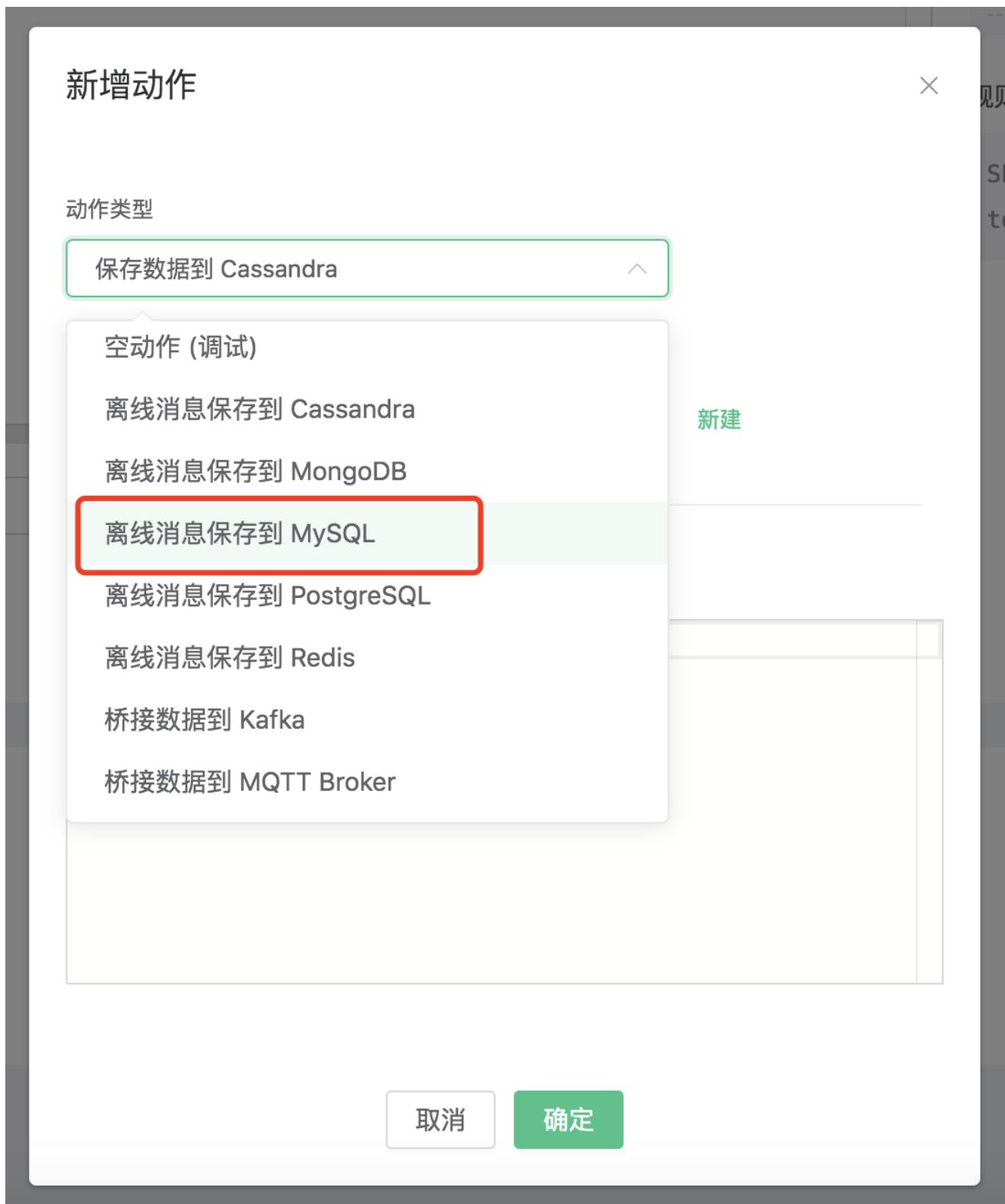
...

SQL 测试:

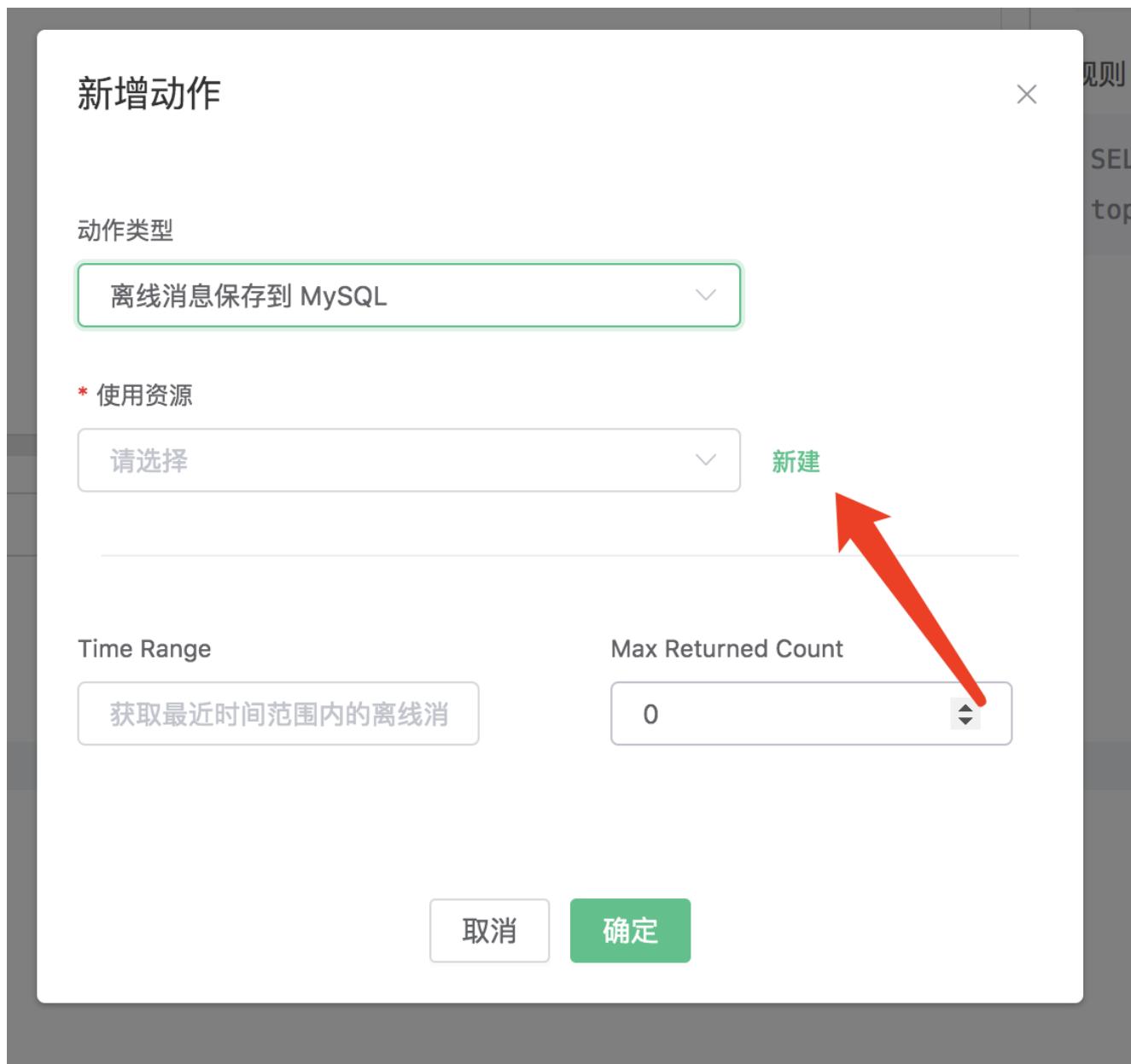


关联动作:

在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 MySQL”。



现在资源下拉框为空，可以点击右上角的“新建”来创建一个 **MySQL** 资源：



弹出一个“创建资源”对话框

创建资源

X

* 资源类型

MySQL

测试连接

* 资源名称

请输入

* MySQL 服务器

192.168.1.173:3306

* MySQL 数据库名

mqtt

连接池大小

8

* MySQL 用户名

root

MySQL 密码

public

批量写入大小

100

批量写入间隔(毫秒)

10

是否重连

true

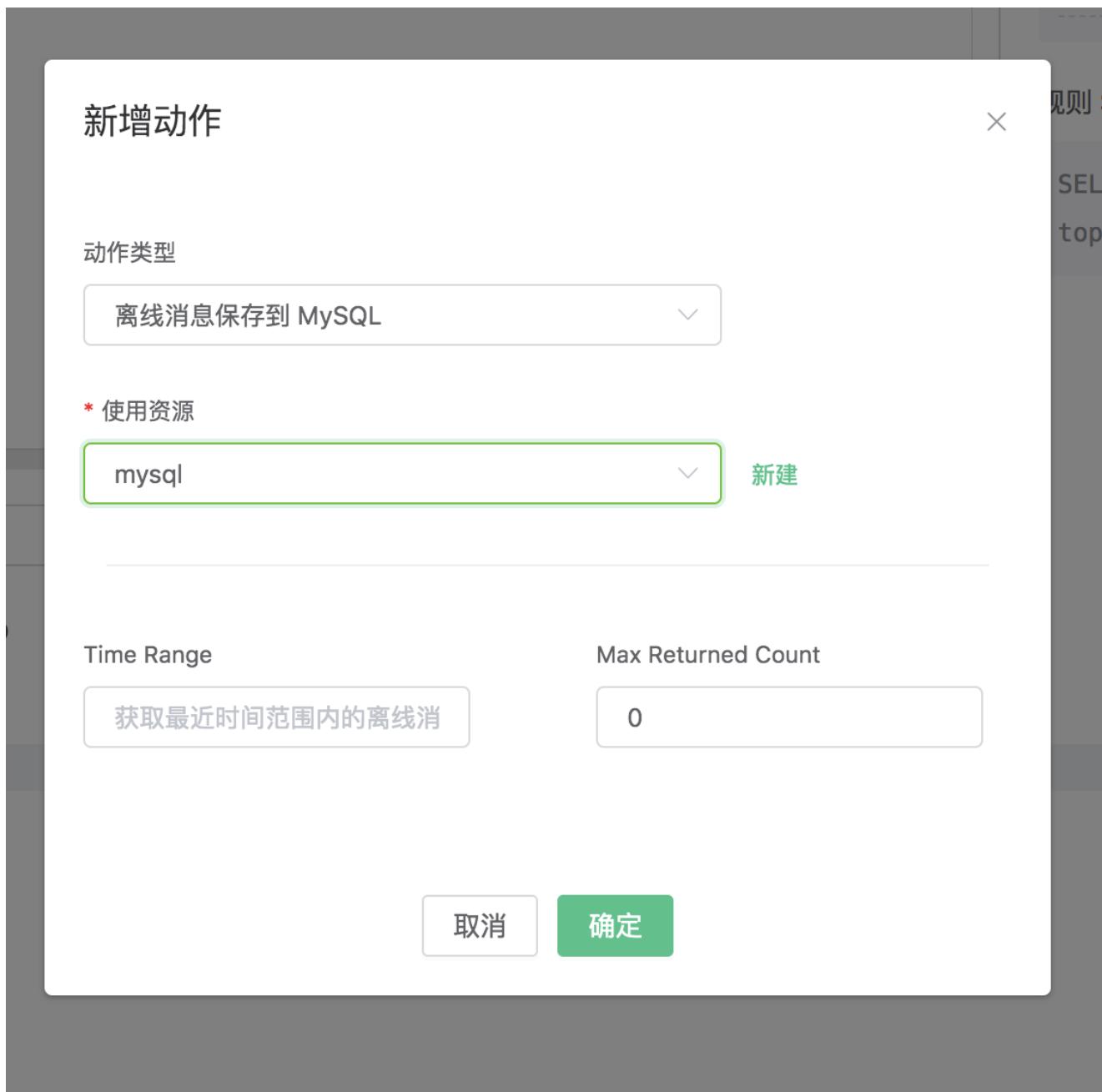
取消

确定

填写资源配置：

填写真实的 **MySQL** 服务器地址，其他配置填写相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 离线消息保存到 MySQL (offline_msg_to_mysql)	编辑 移除
离线消息保存到 MySQL	
Time Range Max Returned Count 0 资源 ID resource:88bbe17	+ 失败备选动作

+ 添加动作

取消 创建

返回规则创建界面，点击“创建”。

The screenshot shows a table with columns: ID, 主题 (Topic), 监控 (Monitoring), 描述 (Description), 状态 (Status), and 响应动作 (Response Actions). The rule details are:

ID	主题	监控	描述	状态	响应动作
rule:560dcb8e	t/# \$events/session_subscribed \$events/message_acked	full		<input checked="" type="checkbox"/>	离线消息保存到 MySQL QL 编辑 删除

规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据**(发布消息的**QoS**必须大于0)**：

The Publish section of the dashboard shows the following configuration:

- Topic: t/1
- Payload: hello
- QoS: 1
- Retain:

Below the configuration, there are two tables:

已接收	已发送		
Topic	QoS	Payload	时间
t/1	1	hello	10:09:10

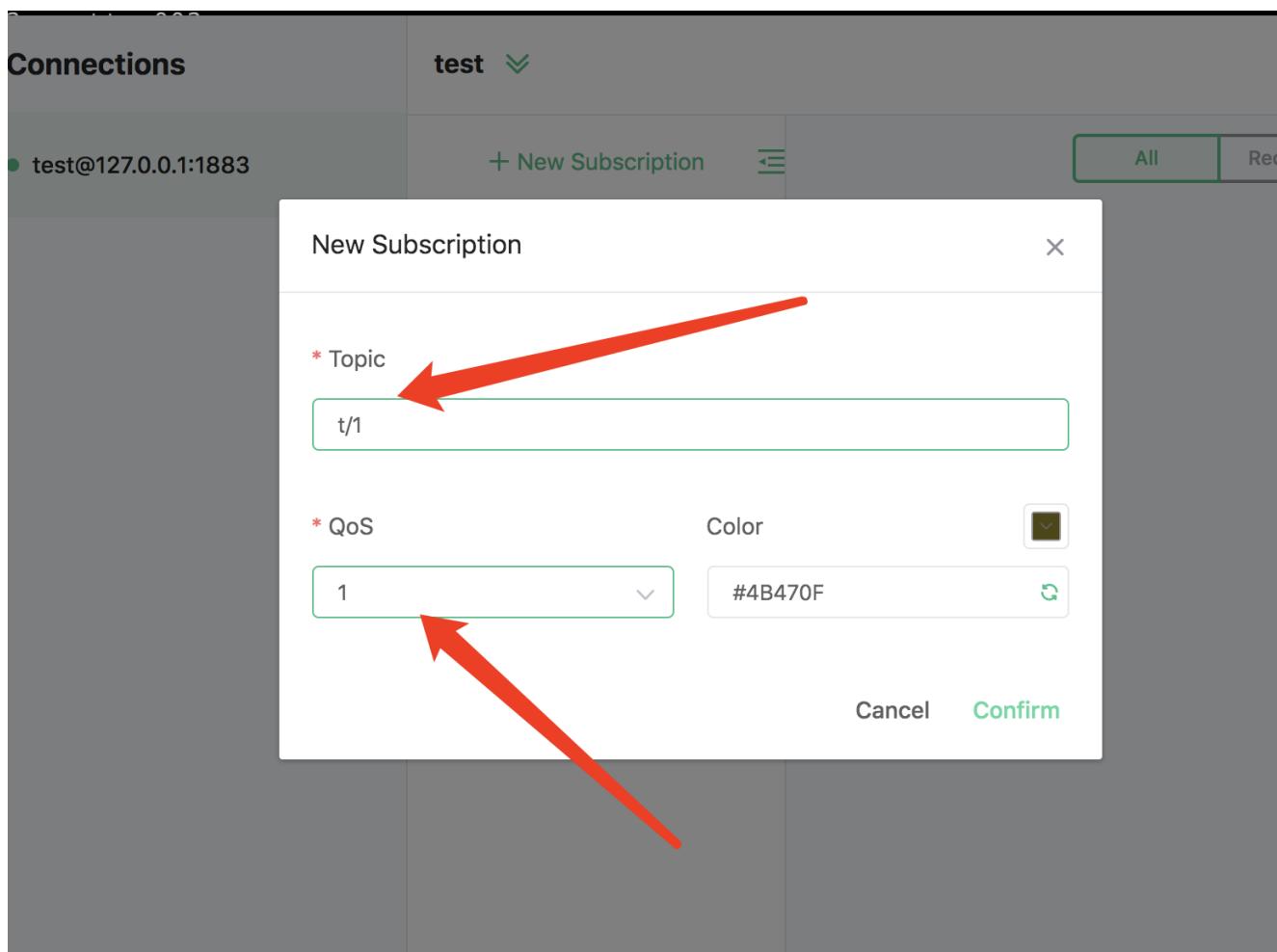
暂无数据

消息发送后，通过 **mysql** 查看到消息被保存到 **MySQL** 里面：

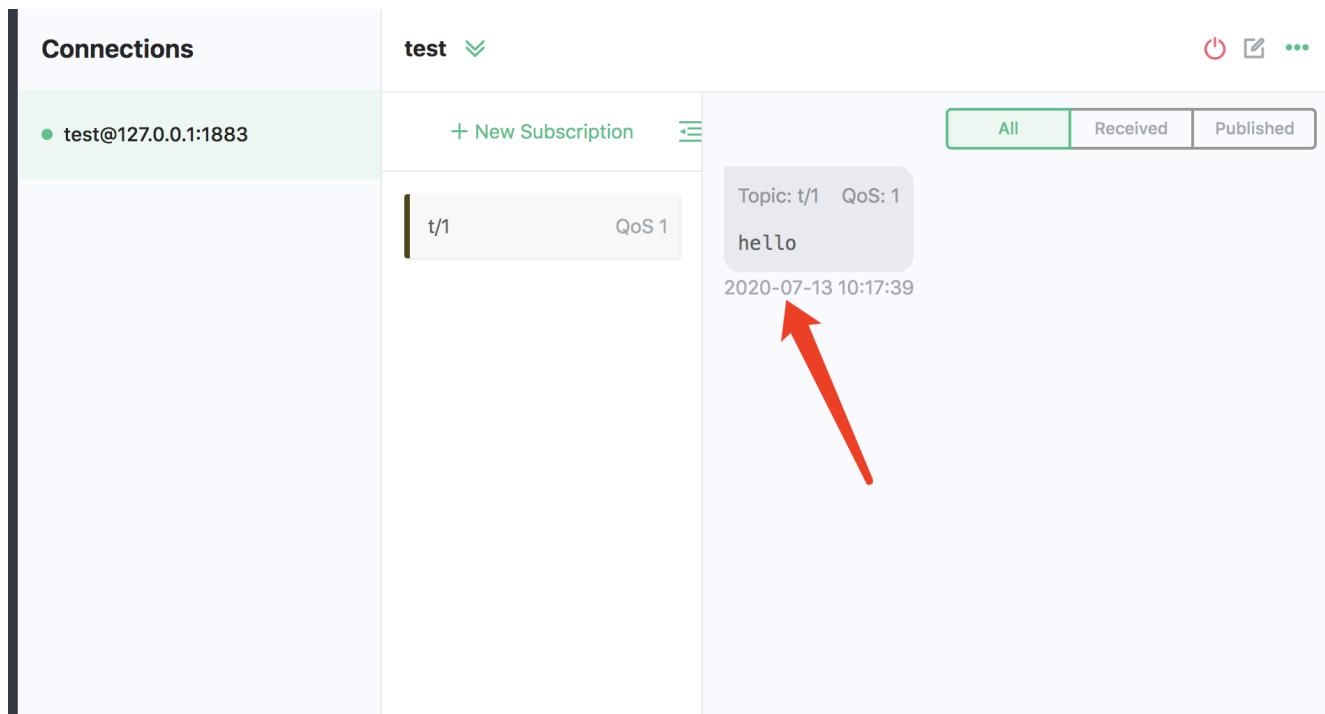
```
mysql> select * from mqtt_msg;
+----+-----+-----+-----+-----+-----+-----+
| id | msgid           | topic | sender | qos | retain | payload | arrived          |
+----+-----+-----+-----+-----+-----+-----+
| 1  | 5AC1A78431C3EF44200000C740001 | t/1   | test   | 1   | 0      | hello   | 2020-08-05 05:16:10 |
+----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

使用另外一个客户端，订阅主题 "t/1" (订阅主题的**QoS**必须大于0，否则消息会被重复接收)：



订阅后马上接收到了保存到 MySQL 里面的离线消息:



离线消息被接收后会在 MySQL 中删除:

```
mysql> select * from mqtt_msg;
+----+-----+-----+-----+-----+-----+
| id | msgid          | topic | sender | qos  | retain | payload | arrived      |
+----+-----+-----+-----+-----+-----+
| 1  | SAC1A78431C3EF44200000C740001 | t/1   | test   | 1   | 0    | hello   | 2020-08-05 05:16:10 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql> select * from mqtt_msg;
Empty set (0.00 sec)

mysql>
```

离线消息保存到 PostgreSQL

提示

支持 **PostgreSQL 13** 及以下版本

搭建 **PostgreSQL** 数据库，以 **MacOS X** 为例：

```
1 $ brew install postgresql  
2 $ brew services start postgresql
```

sh

创建 **mqtt** 数据库：

```
1 # 使用用户名 postgres 创建名为 'mqtt' 的数据库  
2 $ createdb -U postgres mqtt  
3  
4 $ psql -U postgres mqtt  
5  
6 mqtt=> \dn;  
7 List of schemas  
8 Name   | Owner  
9 -----+-----  
10 public | postgres  
11 (1 row)
```

创建 **mqtt_msg** 表：

```
1 $ psql -U postgres mqtt  
2  
3 CREATE TABLE mqtt_msg (  
4     id SERIAL8 primary key,  
5     msgid character varying(64),  
6     sender character varying(64),  
7     topic character varying(255),  
8     qos integer,  
9     retain integer,  
10    payload text,  
11    arrived timestamp without time zone  
12 );
```

提示

消息表结构不能修改，请使用上面**SQL**语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 **SQL**：

FROM说明

t/#: 发布者发布消息触发保存离线消息到**PostgreSQL**

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息**ACK**后触发删除已经被接收的离线消息

```
1   SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't
  /#'
```

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4     "t/#",
5     "$events/session_subscribed",
6     "$events/message_acked"
7   WHERE
8     topic =~ 't/#'
9
```

sh

当前事件可用字段

```
event clientid username peerhost topic qos timestamp node
```

规则 SQL 示例

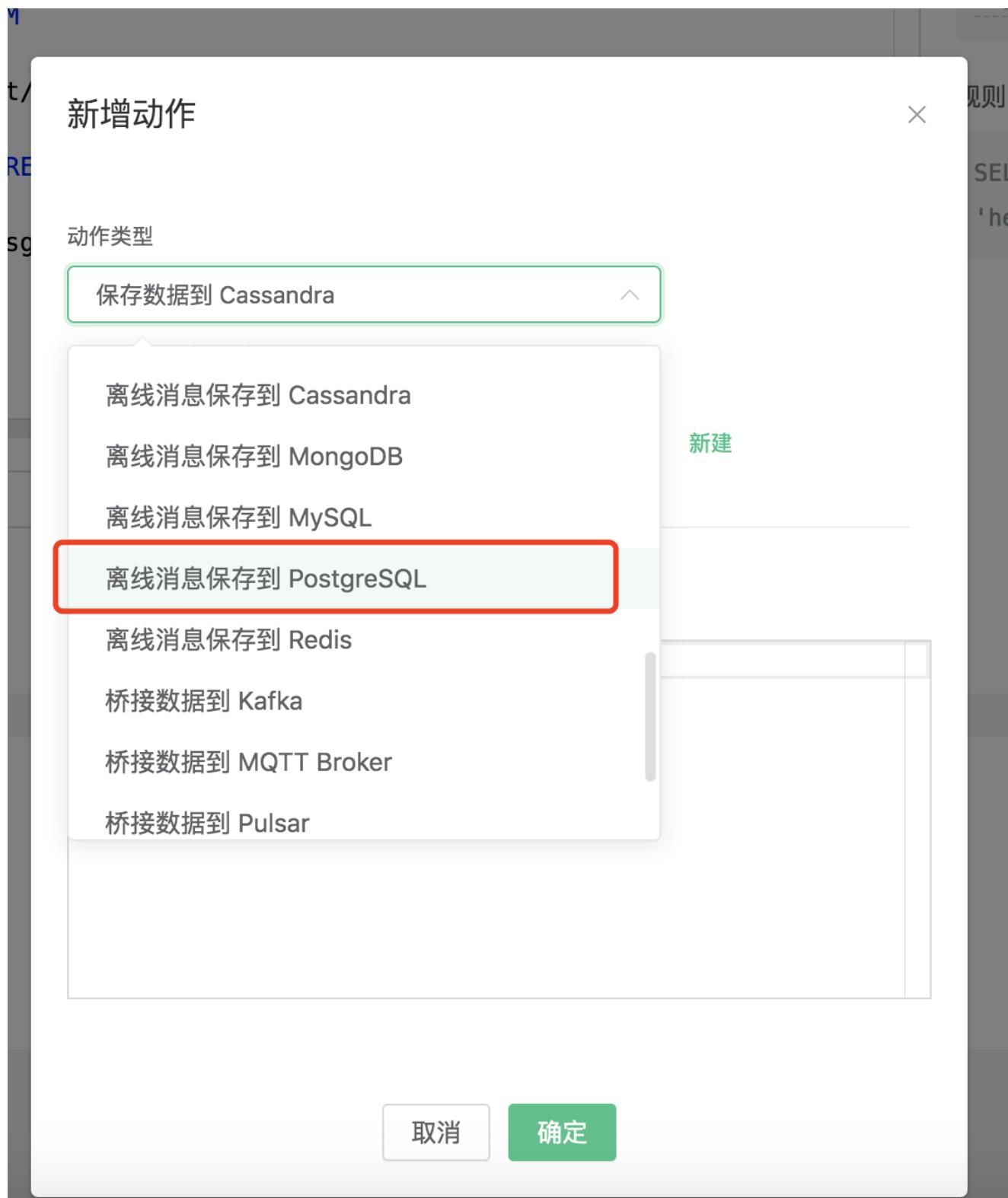
```
SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'
```

备注:

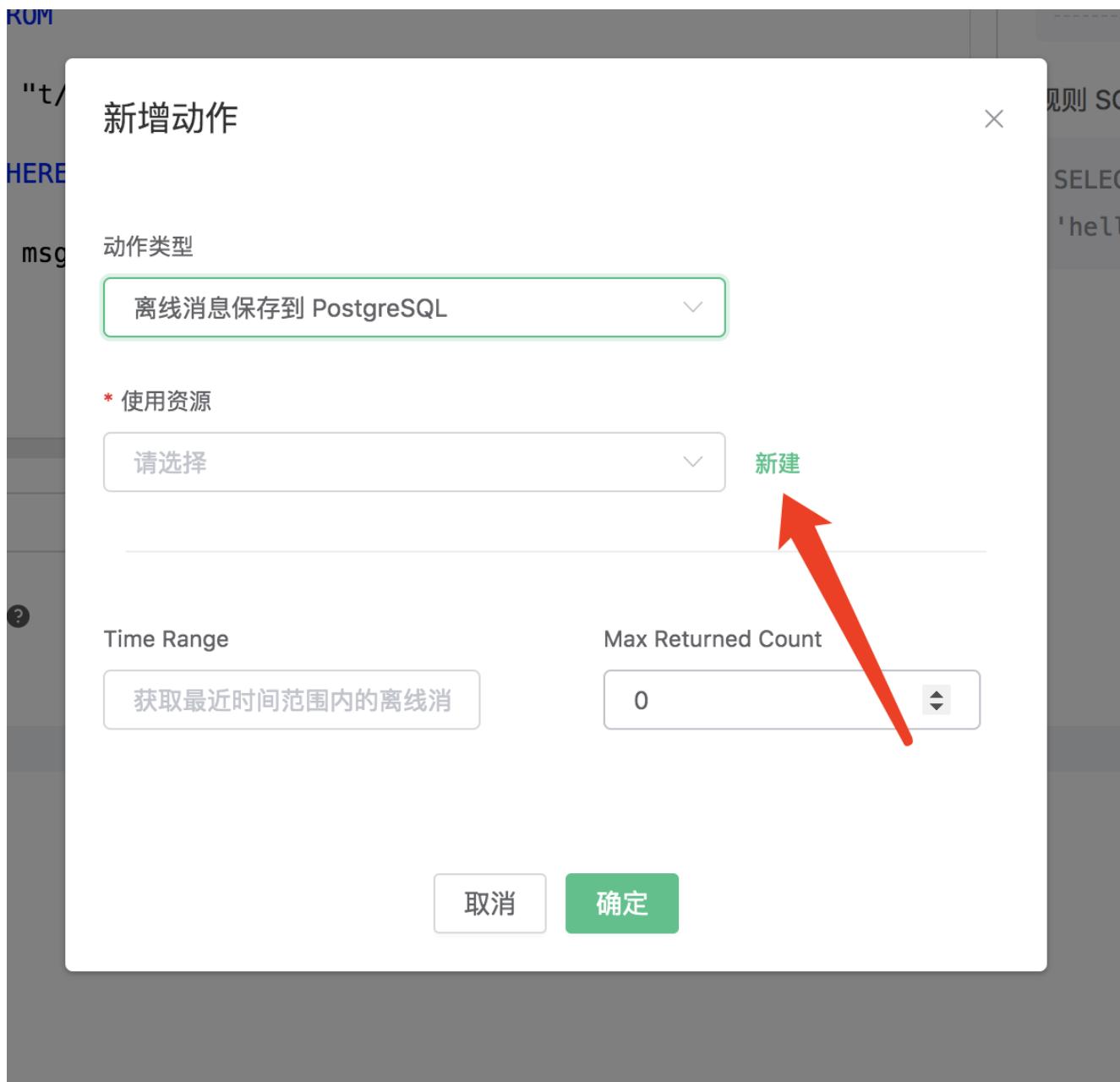
SQL 测试:

关联动作:

在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 **PostgreSQL**”。

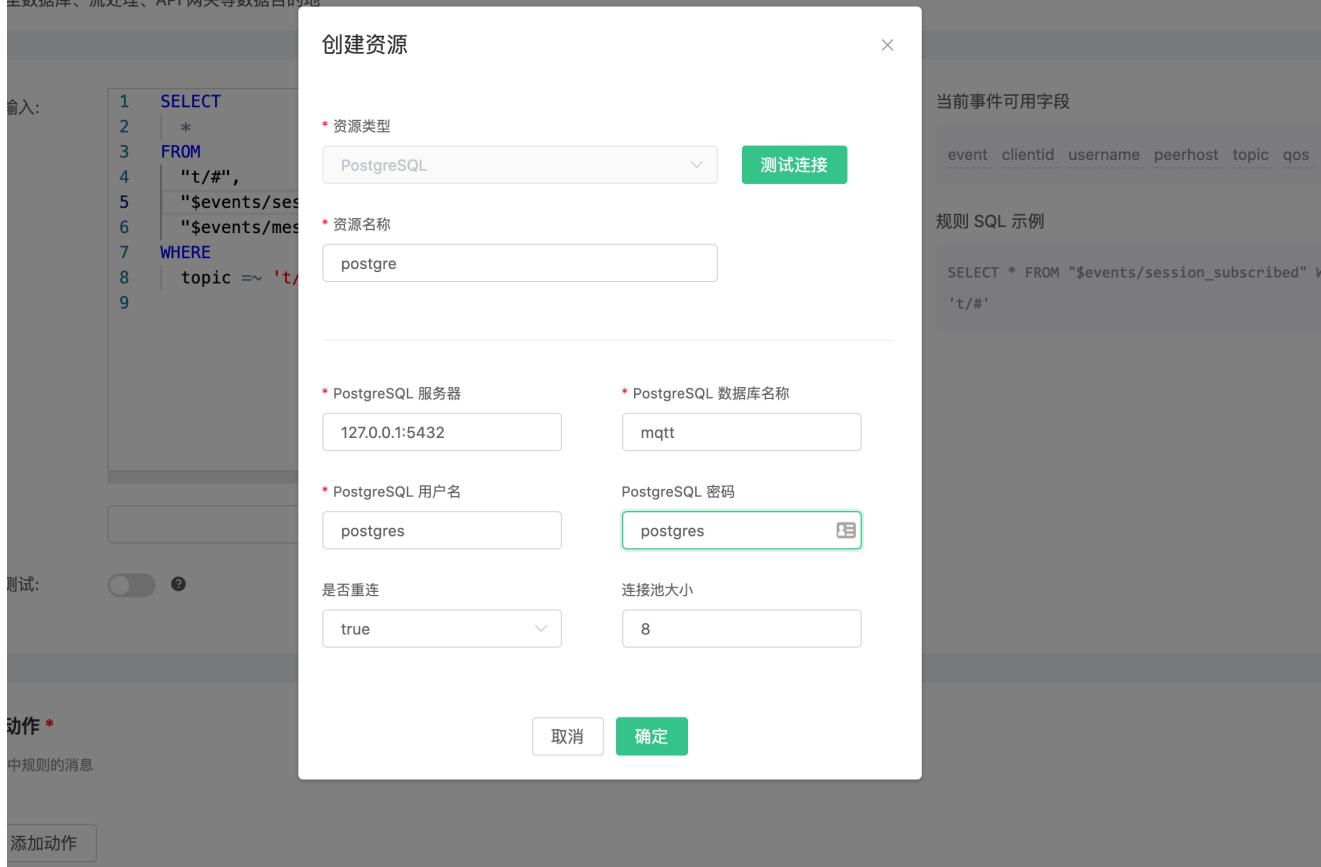


现在资源下拉框为空，可以点击右上角的“新建”来创建一个 PostgreSQL 资源：



弹出一个“创建资源”对话框

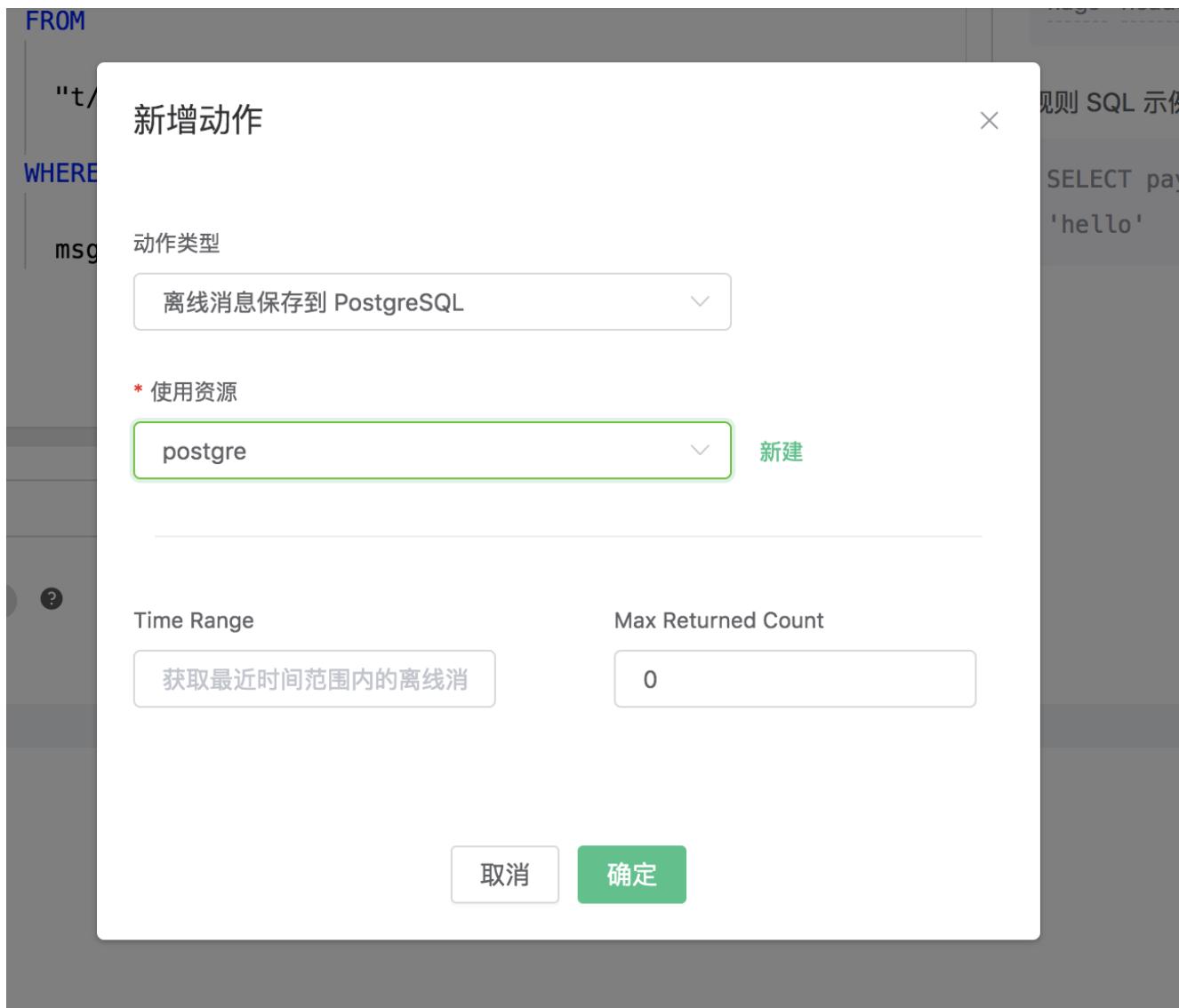
SQL 文本规则，对消息队列插件、插件脚本、安全，灵活地持久化存储的数据发送至数据库、流处理、API 网关等数据目的地。



填写资源配置：

填写真实的 **PostgreSQL** 服务器地址，其他配置填写相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 离线消息保存到 PostgreSQL (offline_msg_to_pgsql)	编辑 移除
离线消息保存到 PostgreSQL	
资源 ID resource:d55aea9c	+ 失败备选动作
+ 添加动作	
取消 确定	

返回规则创建界面，点击“创建”。

t/# rule:e967cb91 \$events/session_subscribed \$events/message_acked	离线消息保存到 PostgreSQL	编辑 删除
<input checked="" type="checkbox"/> 离线消息保存到 PostgreSQL		

规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据****(发布消息的QoS必须大于0)****：

发布

Topic: t/1 Payload: hello QoS: 1 Retain:

已接收: 0 已发送: 0

Topic	QoS	Payload	时间
t/1	1	hello	10:09:10

暂无数据

消息发送后，通过 **psql** 查看到消息被保存到 **PostgreSQL** 里面：

```
mqtt=# select * from mqtt_msg;
 id | msgid | sender | topic | qos | retain | payload | arrived
---+-----+-----+-----+-----+-----+-----+-----+
 61 | Mjk0NDExODY40DMxNTIwMjE4NjA5MzY1NDU0MTIzODI3MjB | test | t/1 | 1 |      | hello | 2020-07-29 08:05:38.963
(1 row)
```

使用另外一个客户端，订阅主题 "**t/1**" (订阅主题的**QoS**必须大于**0**，否则消息会被重复接收)：

The screenshot shows the EMQX Broker's subscription management interface. A modal window titled "New Subscription" is open. It contains fields for "Topic" (set to "t/1") and "QoS" (set to "1"). Red arrows point to both of these fields. In the background, the main interface shows a connection named "test@127.0.0.1:1883" and a list of subscriptions.

订阅后马上接收到了保存到 **PostgreSQL** 里面的离线消息：

The screenshot shows the EMQX Enterprise V4.4 Connections interface. On the left, there's a sidebar with the title 'Connections' and a list item 'test@127.0.0.1:1883'. The main area is titled 'test' with a dropdown arrow. It contains a button '+ New Subscription' and a list of subscriptions: 't/1' (QoS 1). Below the subscriptions, a message is displayed in a box: 'Topic: t/1 QoS: 1' followed by the payload 'hello' and the timestamp '2020-07-13 10:17:39'. At the bottom right of the interface are buttons for 'All', 'Received', and 'Published'. A red arrow points from the text '(0 rows)' in the PostgreSQL query result below to the timestamp '2020-07-13 10:17:39' in the message details.

离线消息被接收后会在 **PostgreSQL** 中删除:

```
mqtt=# select * from mqtt_msg;
 id | msgid | sender | topic | qos | retain | payload | arrived
----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

离线消息保存到 Cassandra

搭建 **Cassandra** 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```
1 $ brew install cassandra
2 ## 修改配置, 关闭匿名认证
3 $ vim /usr/local/etc/cassandra/cassandra.yaml
4
5     authenticator: PasswordAuthenticator
6     authorizer: CassandraAuthorizer
7
8 $ brew services start cassandra
9
10 ## 创建 root 用户
11 $ cqlsh -ucassandra -pcassandra
12
13 create user root with password 'public' superuser;
```

初始化 **Cassandra** 表空间：

```
1 $ cqlsh -uroot -ppublic
2
3 CREATE KEYSPACE mqtt WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'
4 } AND durable_writes = true;
```

创建 **mqtt_msg** 表：

```

1 CREATE TABLE mqtt.mqtt_msg (
2     topic text,
3     msgid text,
4     arrived timestamp,
5     payload text,
6     qos int,
7     retain int,
8     sender text,
9     PRIMARY KEY (topic, msgid)
10 ) WITH CLUSTERING ORDER BY (msgid DESC)
11     AND bloom_filter_fp_chance = 0.01
12     AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
13     AND comment = ''
14     AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
15     AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
16     AND crc_check_chance = 1.0
17     AND dclocal_read_repair_chance = 0.1
18     AND default_time_to_live = 0
19     AND gc_grace_seconds = 864000
20     AND max_index_interval = 2048
21     AND memtable_flush_period_in_ms = 0
22     AND min_index_interval = 128
23     AND read_repair_chance = 0.0
24     AND speculative_retry = '99PERCENTILE';
25
26

```

提示

消息表结构不能修改, 请使用上面SQL语句创建

创建规则:

打开 [EMQX Dashboard](#), 选择左侧的“规则”选项卡。

然后填写规则 SQL:

FROM说明

t/#: 发布者发布消息触发保存离线消息到**Cassandra**

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息**ACK**后触发删除已经被接收的离线消息

```

1 SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't
/#'

```

* SQL 输入:

```

1 SELECT
2   *
3   FROM
4     "t/#",
5     "$events/session_subscribed",
6     "$events/message_acked"
7   WHERE
8     topic =~ 't/#'
9

```

当前事件可用字段

```

event clientid username peerhost topic qos timestamp node

```

规则 SQL 示例

```

SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'

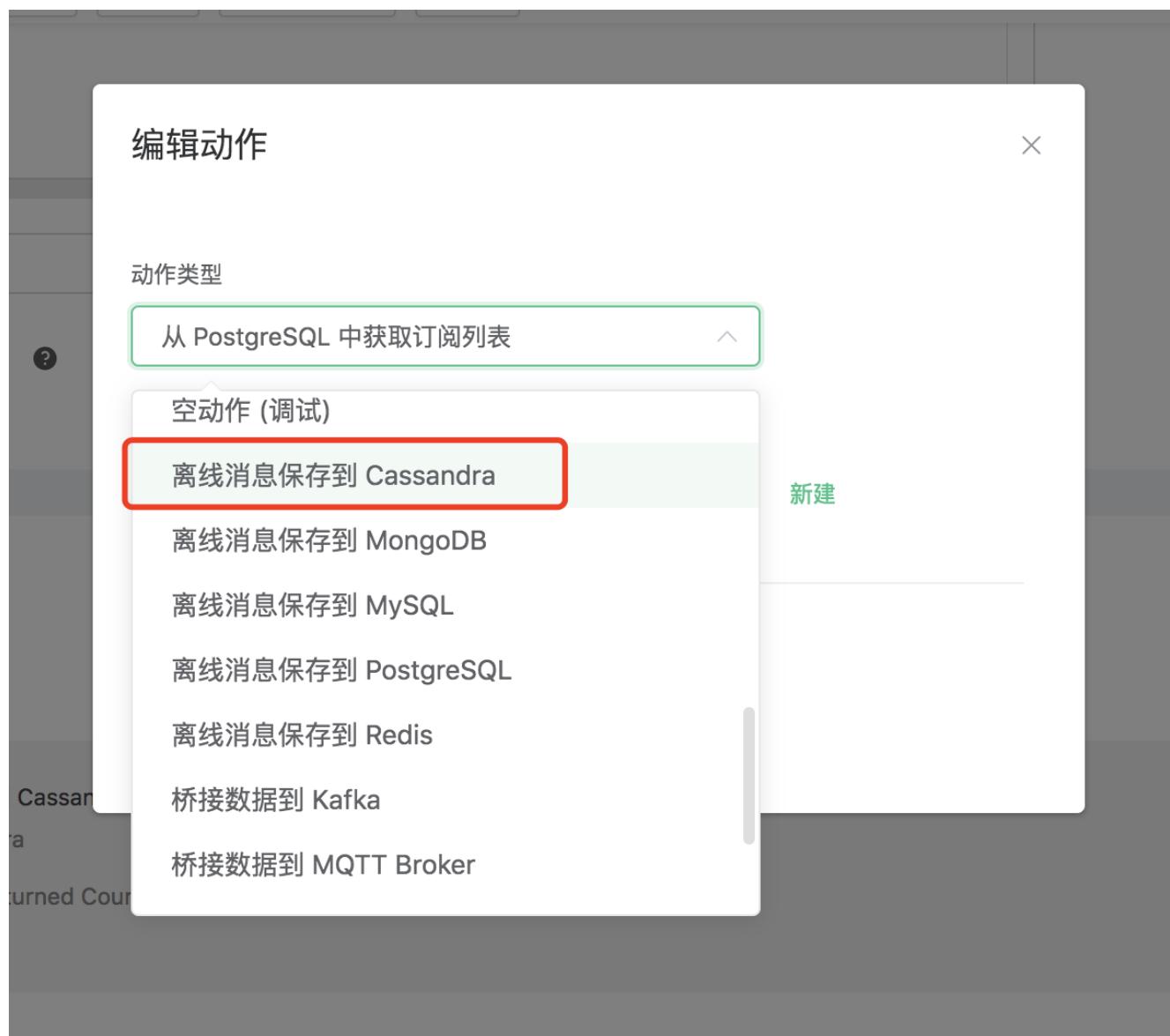
```

备注:

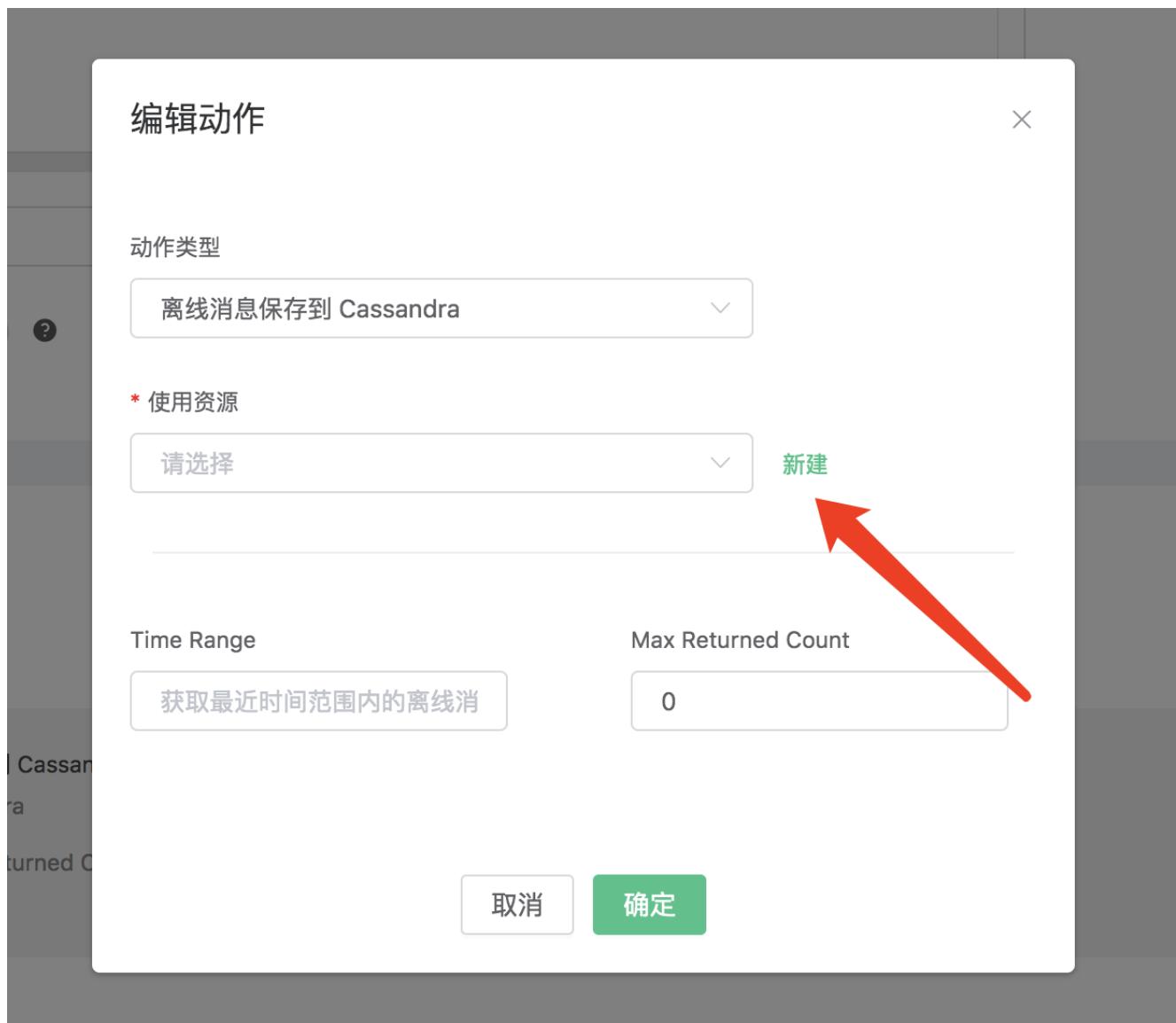
SQL 测试:

关联动作:

在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 Cassandra ”。



现在资源下拉框为空，可以点击右上角的“新建”来创建一个 Cassandra 资源:



弹出一个“创建资源”对话框

创建资源

* 资源类型

Cassandra

测试连接

* 资源名称

cassandra

* Cassandra 服务器

192.168.1.173:9042

* Cassandra Keyspace

mqtt

Cassandra 用户名

root

Cassandra 密码

public

是否重连

true

连接池大小

8

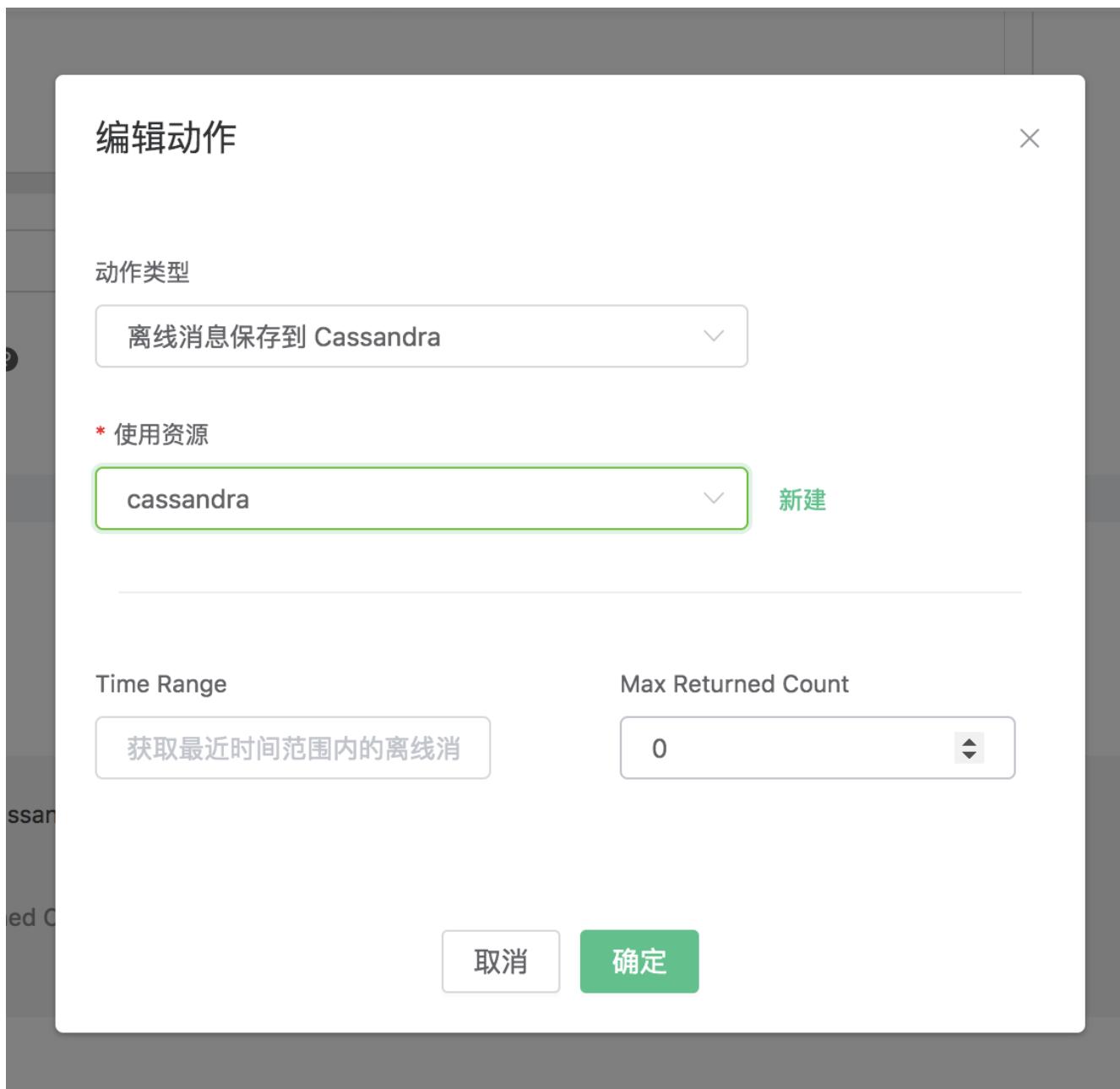
取消 确定

取消 创建

填写资源配置：

填写真实的 **Cassandra** 服务器地址，其他配置填写相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。



返回规则创建界面，点击“创建”。

ID	主题	监控	描述	状态	响应动作
rule:7d41df08	t/# \$events/session_subscribed \$events/message_acked	full		<input checked="" type="checkbox"/>	离线消息保存到 Cassandra 编辑 删除

规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据**(发布消息的**QoS**必须大于**0**)**:

发布

Topic: t/1 Payload: hello QoS: 1 Retain: 发布

已接收: 0 已发送: 1

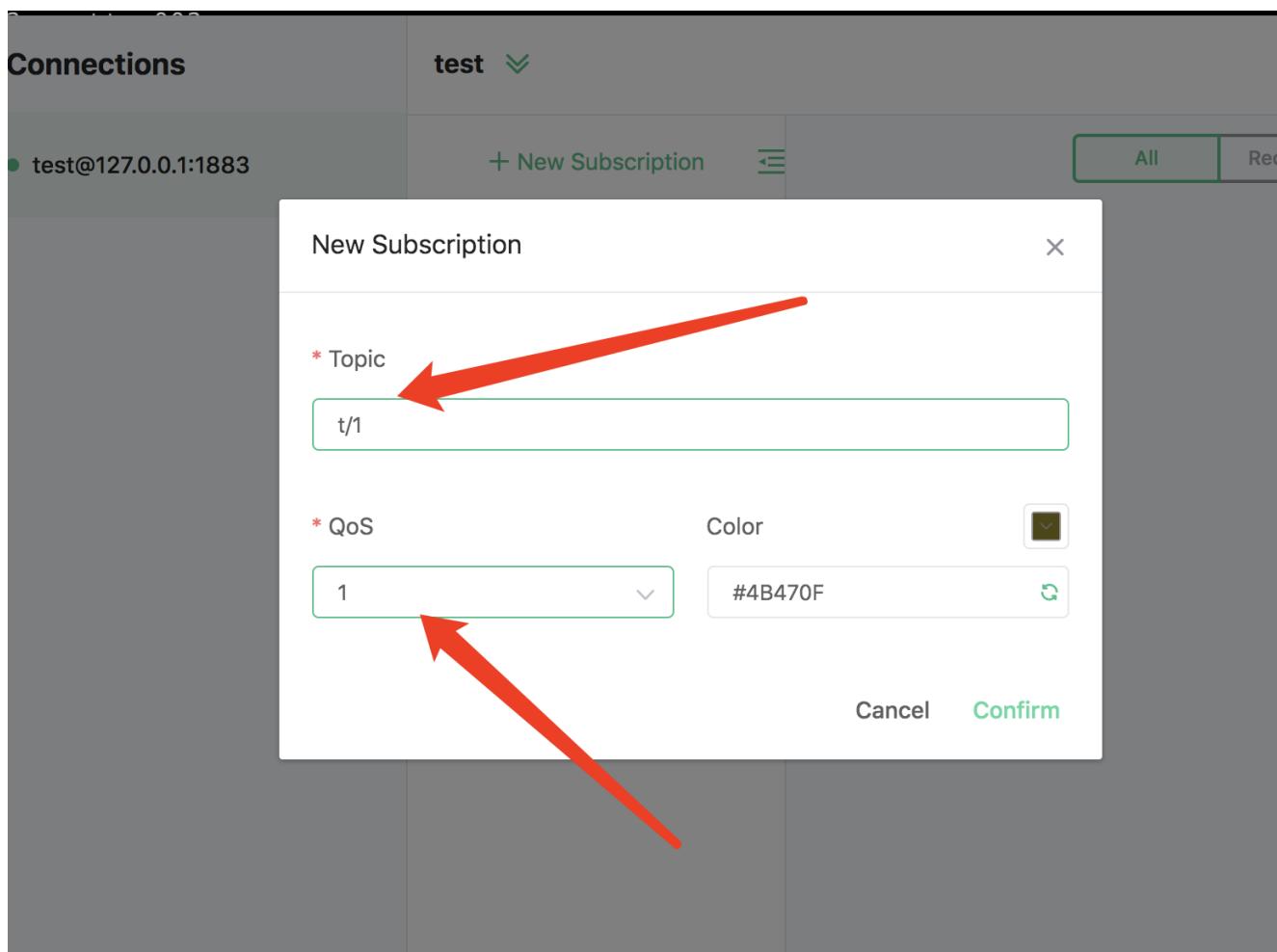
Topic	QoS	Payload	时间
t/1	1	hello	10:09:10

暂无数据

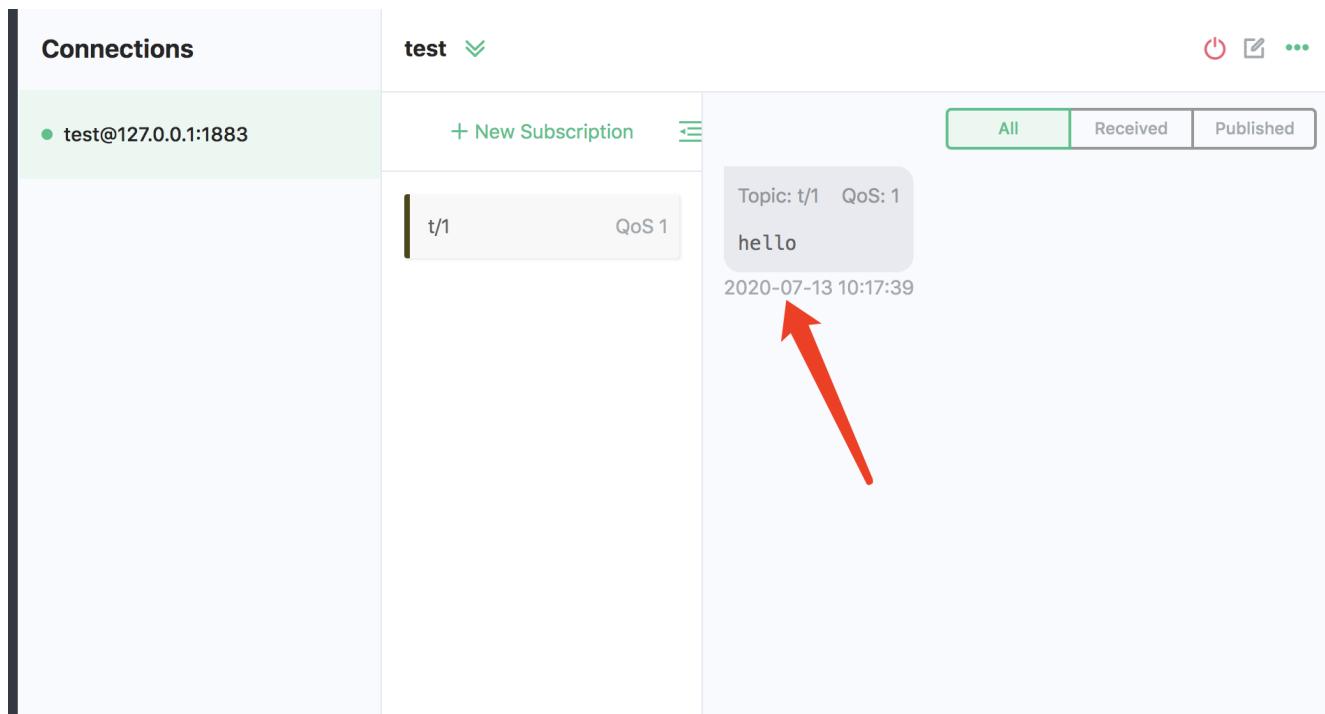
消息发送后，通过 **cqlsh** 查看到消息被保存到 **Cassandra** 里面:

```
root@cqlsh:mqtt> select * from mqtt_msg;
topic | msgid | arrived | payload | qos | retain | sender
-----+-----+-----+-----+-----+-----+-----+
t/1 | 5AC1C6D8D81FAF442000009880001 | 2020-08-05 07:36:20.605000+0000 | hello | 1 | 0 | test
(1 rows)
root@cqlsh:mqtt>
```

使用另外一个客户端，订阅主题 "**t/1**" (订阅主题的**QoS**必须大于**0**，否则消息会被重复接收):



订阅后马上接收到了保存到 **Cassandra** 里面的离线消息:



离线消息被接收后会在 **Cassandra** 中删除:

```
Use HELP for help.
root@cqlsh> use mqtt;
root@cqlsh:mqtt> select * from mqtt_msg;

topic | msgid | arrived | payload | qos | retain | sender
-----+-----+-----+-----+-----+-----+-----+
t/1 | 5AC1C6D8D81FAF442000009880001 | 2020-08-05 07:36:20.605000+0000 | hello | 1 | 0 | test

(1 rows)
root@cqlsh:mqtt>
root@cqlsh:mqtt>
root@cqlsh:mqtt>
root@cqlsh:mqtt> select * from mqtt_msg;

topic | msgid | arrived | payload | qos | retain | sender
-----+-----+-----+-----+-----+-----+-----+
(0 rows)
root@cqlsh:mqtt>
```

离线消息保存到 MongoDB

搭建 **MongoDB** 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```

1 $ brew install mongodb
2 $ brew services start mongodb
3
4 ## 新增 root/public 用户
5 $ use mqtt;
6 $ db.createUser({user: "root", pwd: "public", roles: [{role: "readWrite", db: "mqtt"}]});
7
8 ## 修改配置，关闭匿名认证
9 $ vi /usr/local/etc/mongod.conf
10
11     security:
12         authorization: enabled
13
14 $ brew services restart mongodb

```

创建 **mqtt_msg** 表：

```

1 $ mongo 127.0.0.1/mqtt -uroot -ppublic
2 db.createCollection("mqtt_msg");

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 **SQL**：

FROM说明

t/#: 发布者发布消息触发保存离线消息到**MongoDB**

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息**ACK**后触发删除已经被接收的离线消息

```

1 SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't
/#'

```

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4     "t/#",
5     "$events/session_subscribed",
6     "$events/message_acked"
7   WHERE
8     topic =~ 't/#'
9
```

当前事件可用字段

event	clientid	username	peerhost	topic	qos	timestamp	node
-------	----------	----------	----------	-------	-----	-----------	------

规则 SQL 示例

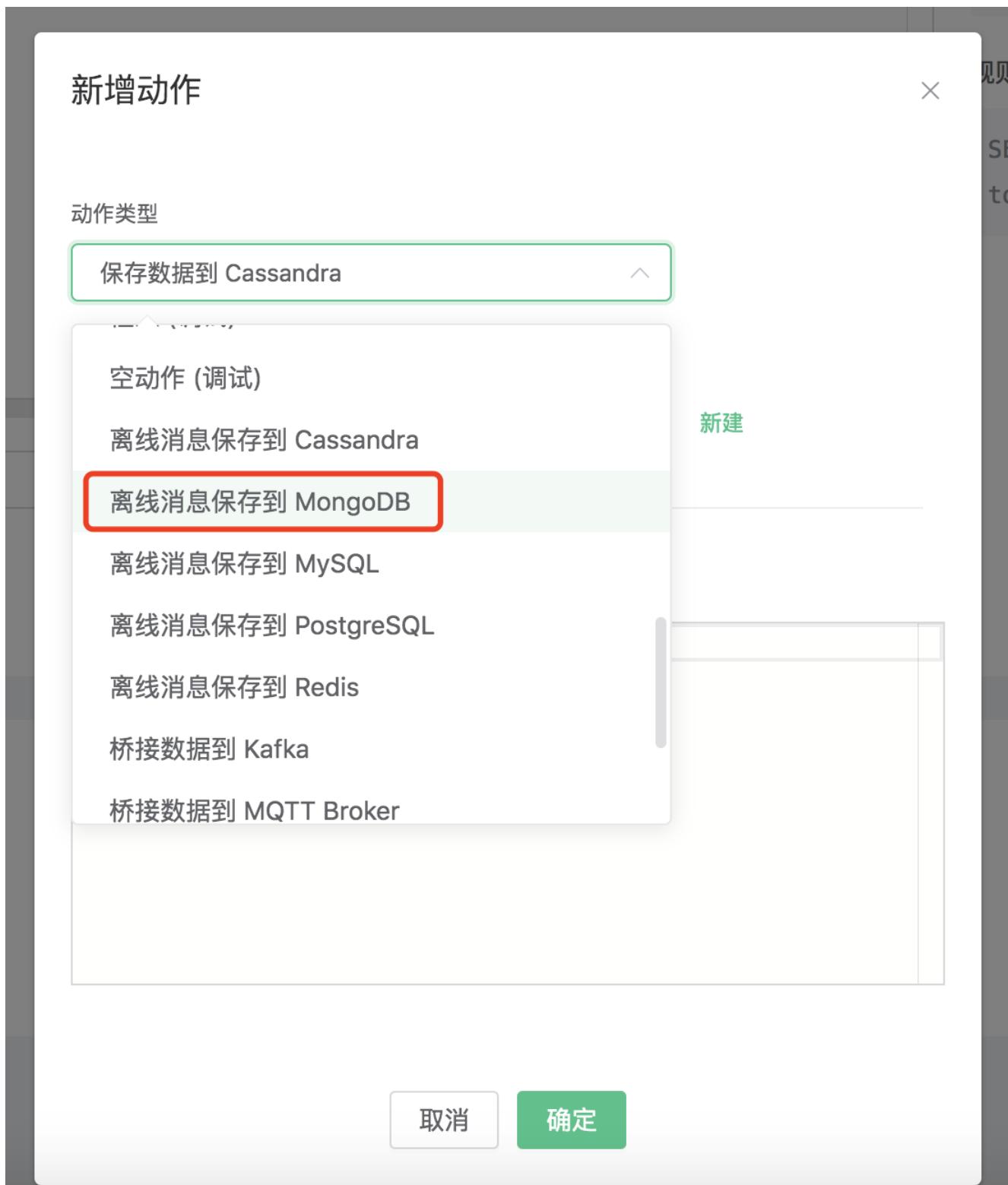
```
SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'
```

备注:

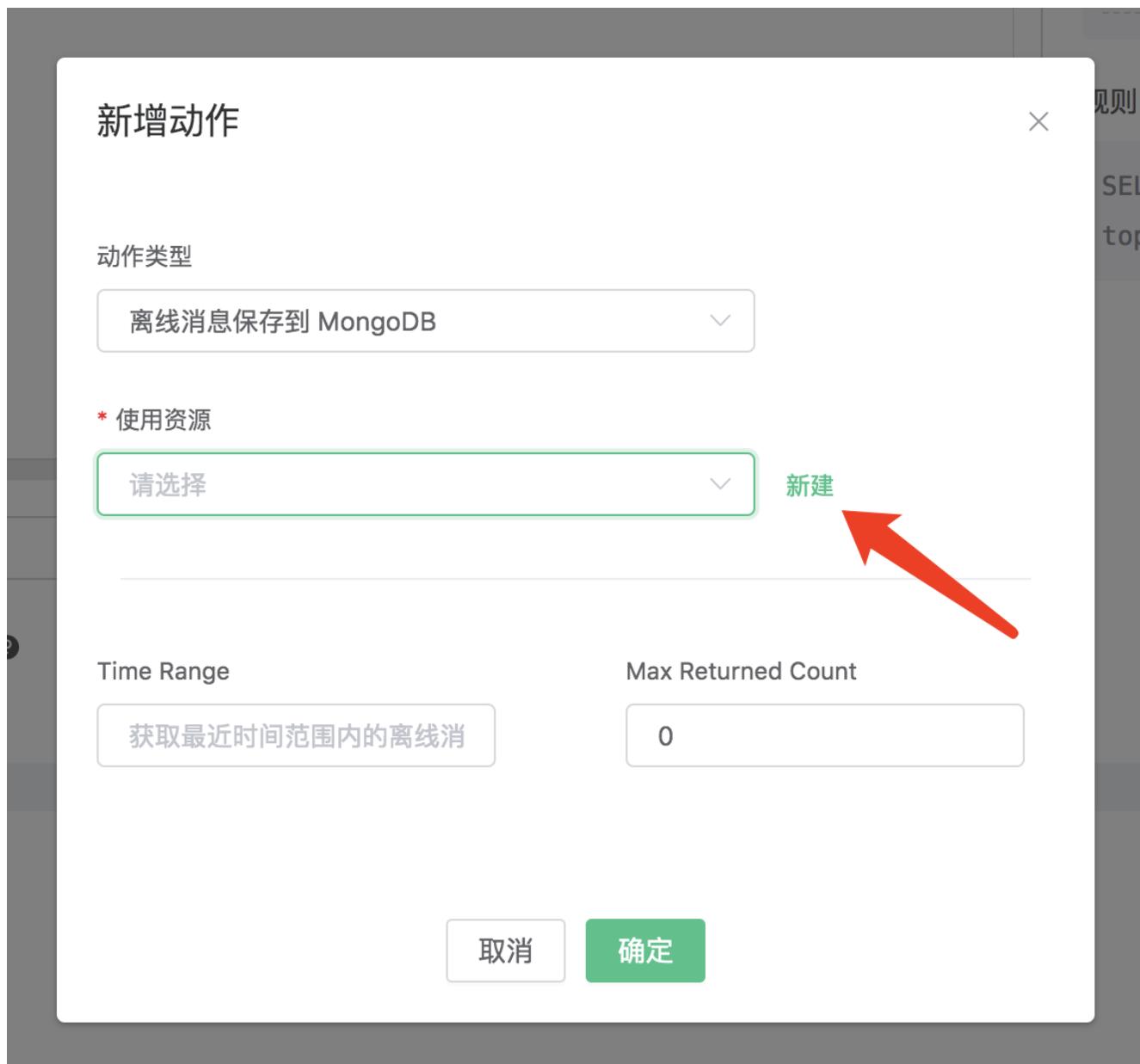
SQL 测试:

关联动作:

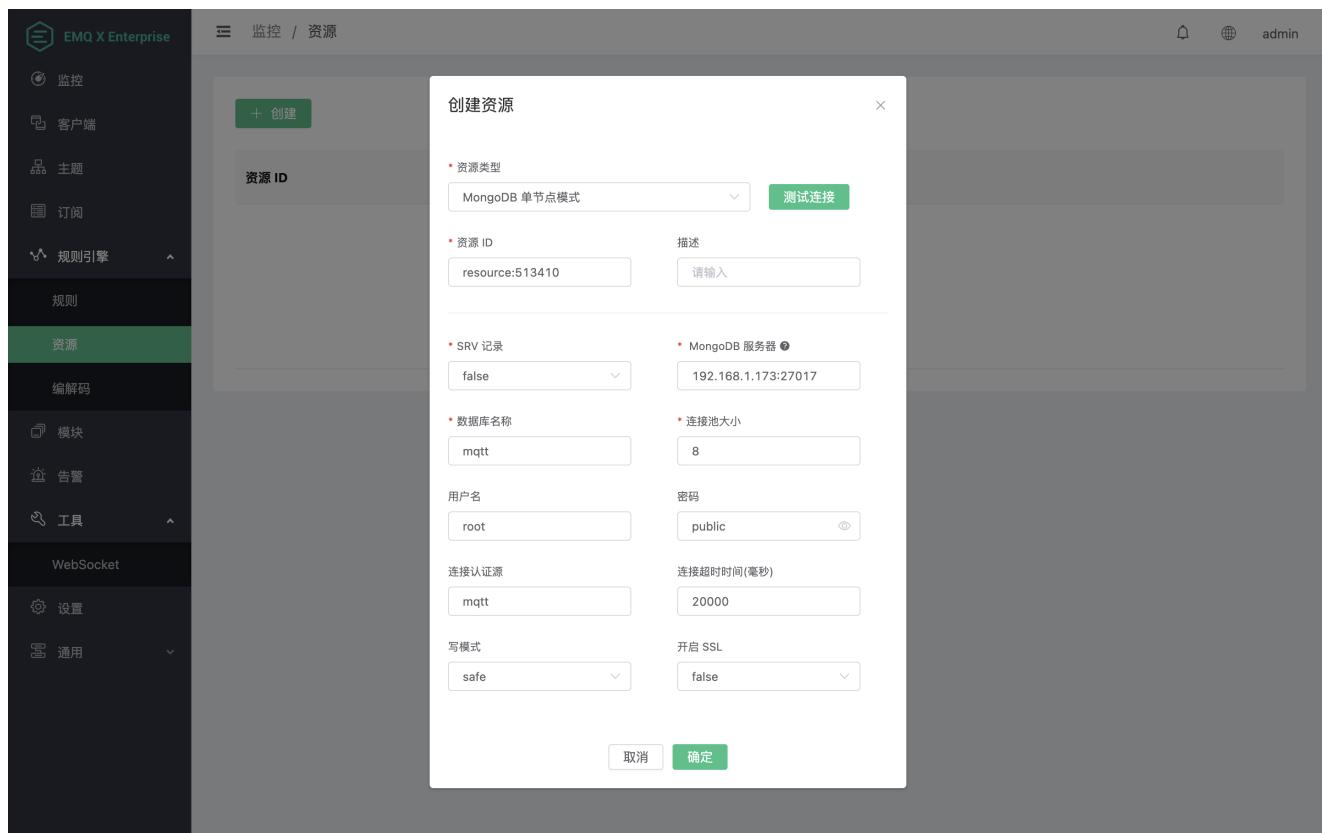
在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 **MongoDB**”。



现在资源下拉框为空，可以点击右上角的“新建”来创建一个 **MongoDB** 资源：



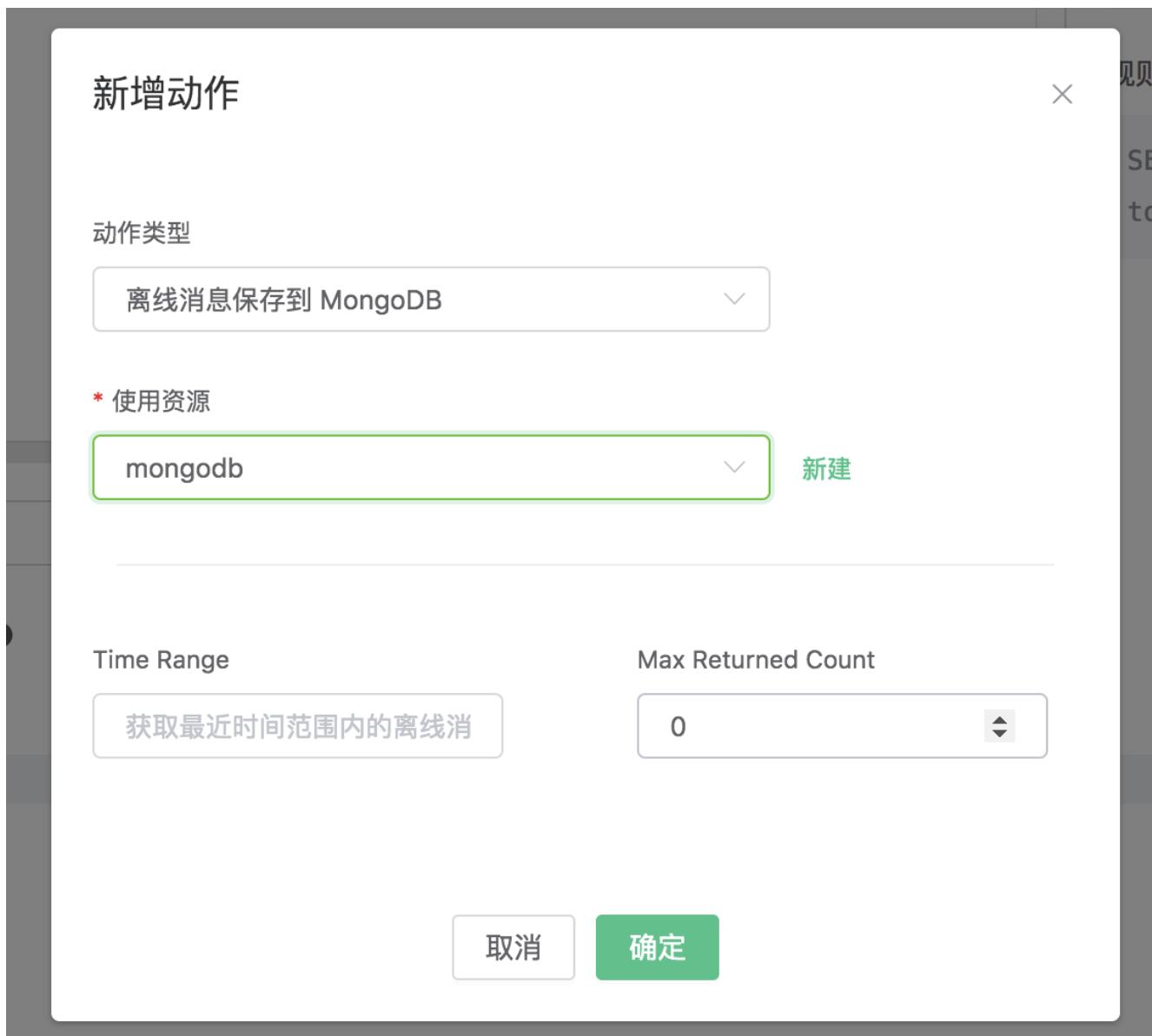
弹出一个“创建资源”对话框



填写资源配置：

填写真实的 **MongoDB** 服务器地址，其他配置填写相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

The screenshot shows the "Response Actions" interface. It lists an action: "动作类型 离线消息保存到 MongoDB (offline_msg_to_mongo)" with resource ID "resource:5ad5b4f0". There are buttons for "编辑" (Edit) and "移除" (Delete). Below the list is a button "+ 添加动作" (Add Action). At the bottom are "取消" (Cancel) and "创建" (Create) buttons.

返回规则创建界面，点击“创建”。

ID	主题	监控	描述	状态	响应动作
rule:7dd747f4	t/# \$events/session_subscribed \$events/message_acked	full		<input checked="" type="checkbox"/>	离线消息保存到 MongoDB 编辑 删除

规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据****(发布消息的QoS必须大于0)****：

The screenshot shows the '发布' (Publish) section of the EMQX Dashboard. It includes fields for Topic (t/1), Payload (hello), and QoS (1). A 'Retain' checkbox is unchecked. A green '发布' (Publish) button is visible. Below the form, there are two tables: '已接收' (Received) and '已发送' (Sent). The '已发送' table contains one entry: Topic t/1, QoS 1, Payload hello, and Time 10:09:10.

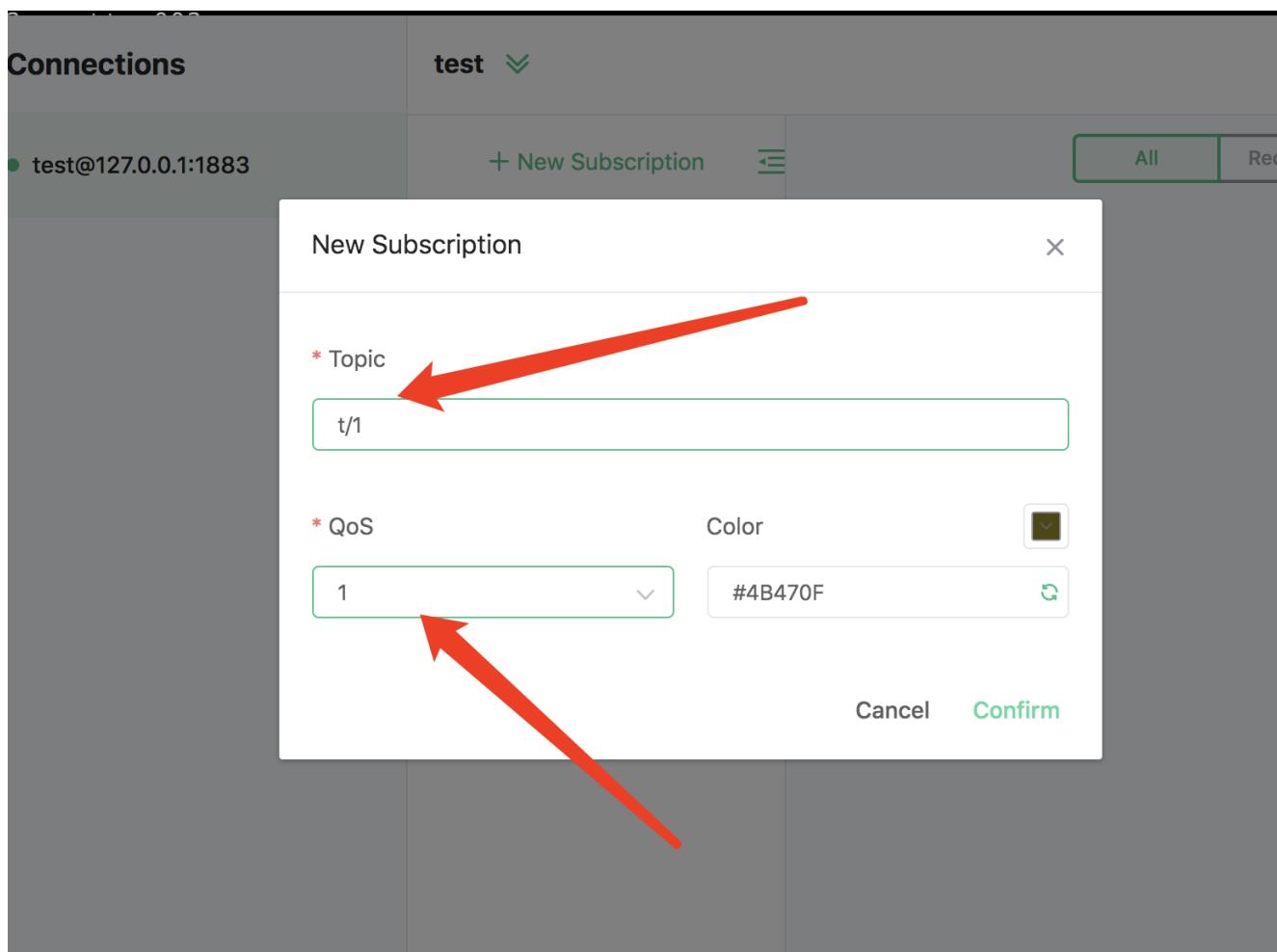
Topic	QoS	Payload	时间
t/1	1	hello	10:09:10

消息发送后，通过 **mongo** 查看到消息被保存到 **MongoDB** 里面：

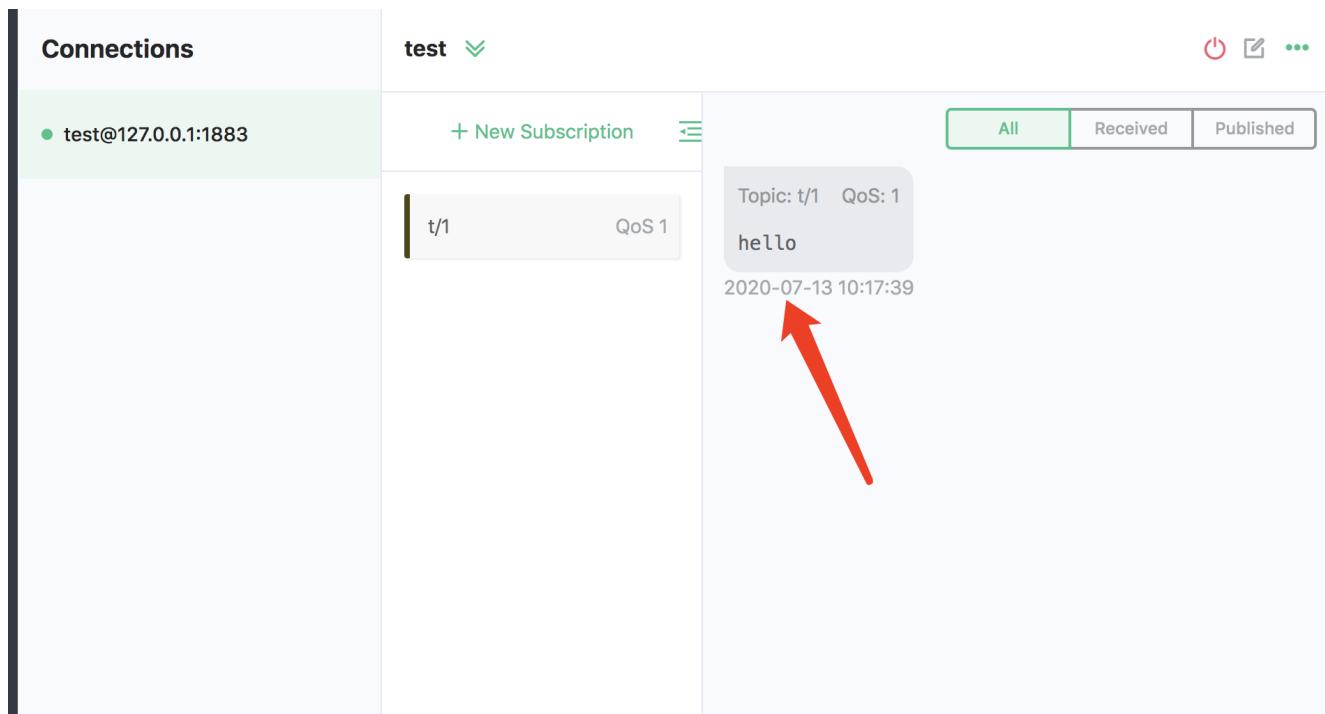
```
1 db.mqtt_msg.find()

> db.mqtt_msg.find()
{ "_id" : ObjectId("5f2a718452336c529d000001"), "topic" : "t/1", "msgid" : "5AC1D62A5CEEF440000019490002", "sender" : "test", "qos" : 1, "retain" : 0, "payload" : "hello", "timestamp" : NumberLong("1596617092615") }
> |
```

使用另外一个客户端，订阅主题 "**t/1**" (订阅主题的**QoS**必须大于**0**，否则消息会被重复接收)：



订阅后马上接收到了保存到 **MongoDB** 里面的离线消息:



离线消息被接收后会在 **MongoDB** 中删除:

```
> db.mqtt_msg.find()
{ "_id" : ObjectId("5f2a718452336c529d000001"), "topic" : "t/1", "msgid" : "5AC1D62A5CEEEF440000019490002", "sender" : "test", "qos" : 1, "retained" : 0, "payload" : "hello", "timestamp" : NumberLong("1596617092615") }
>
>
>
db.mqtt_msg.find()
```

离线消息保存到 ClickHouse

搭建 **ClickHouse** 数据库，并设置用户名密码为 **default/public**，以 **CentOS** 为例：

```

1 ## 安装依赖
2 sudo yum install -y epel-release
3
4 ## 下载并运行packagecloud.io提供的安装shell脚本
5 curl -s https://packagecloud.io/install/repositories/altinity/clickhouse/script.rpm.sh | sudo bash
6
7
8 ## 安装ClickHouse服务器和客户端
9 sudo yum install -y clickhouse-server clickhouse-client
10
11 ## 启动ClickHouse服务器
12 clickhouse-server
13
14 ## 启动ClickHouse客户端程序
clickhouse-client

```

创建 “**mqtt**” 数据库：

```
1 create database mqtt;
```

创建 **mqtt_msg** 表：

```

1 use mqtt;
2 create table mqtt_msg (
3     msgid String,
4     sender String,
5     topic String,
6     qos Nullable(Int8) DEFAULT 0,
7     retain Nullable(Int8) DEFAULT 0,
8     payload String,
9     arrived Int64) engine = MergeTree() ORDER BY msgid;

```

提示

消息表结构不能修改，请使用上面**SQL**语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 **SQL**：

FROM说明

t/#: 发布者发布消息触发保存离线消息到 **ClickHouse**

\$events/session_subscribed: 订阅者订阅主题触发获取离线消息

\$events/message_acked: 订阅者回复消息ACK后触发删除已经被接收的离线消息

```
1   SELECT * FROM "t/#", "$events/session_subscribed", "$events/message_acked" WHERE topic =~ 't
 /#'
```

* SQL 输入:

```
1 SELECT
2   *
3   FROM
4   "t/#",
5   "$events/session_subscribed",
6   "$events/message_acked"
7   WHERE
8   | topic =~ 't/#'
9
```

当前事件可用字段

```
event clientid username peerhost topic qos timestamp node
```

规则 SQL 示例

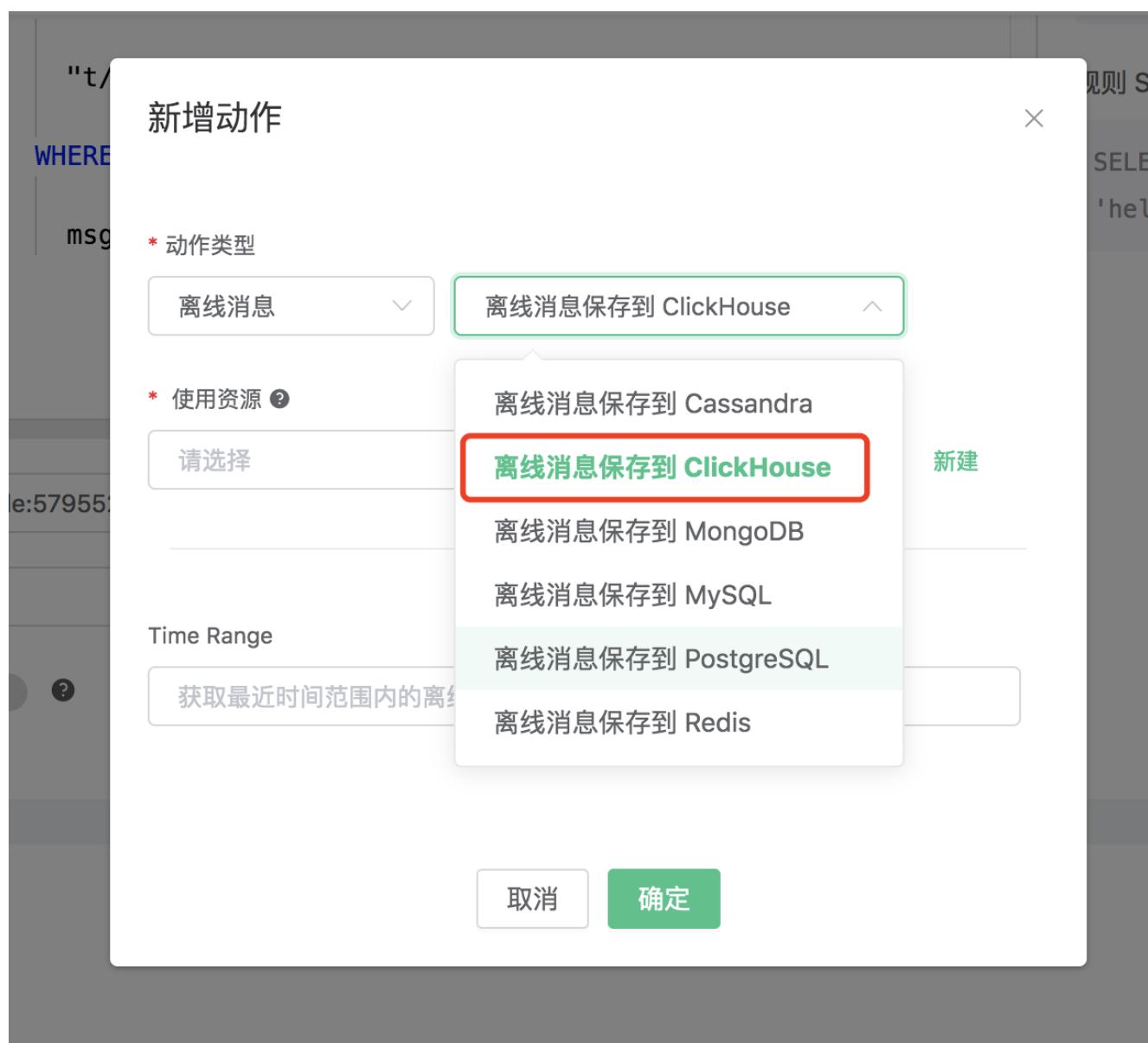
```
SELECT * FROM "$events/session_subscribed" WHERE topic =~ 't/#'
```

备注:

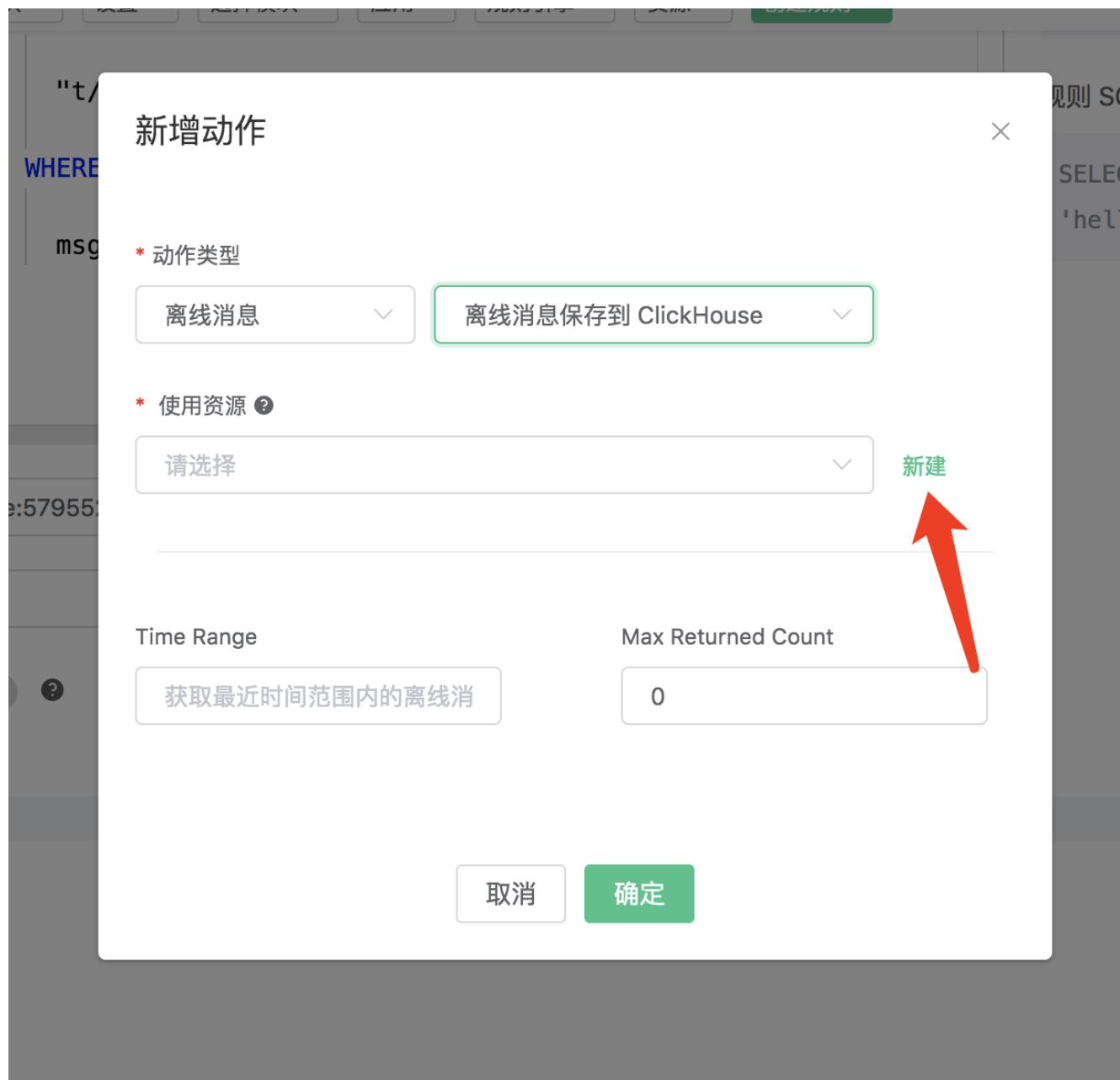
SQL 测试:

关联动作:

在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“离线消息保存到 ClickHouse ”。



现在资源下拉框为空，可以点击右上角的“新建”来创建一个 **ClickHouse** 资源：



弹出一个“创建资源”对话框

创建资源

* 资源类型

ClickHouse

测试连接

* 资源 ID

resource:037903

描述

请输入

* ClickHouse 服务器

http://192.168.0.172:8123

连接池大小

8

ClickHouse 数据库名

mqtt

ClickHouse 用户名

ClickHouse 用户名

ClickHouse 密码

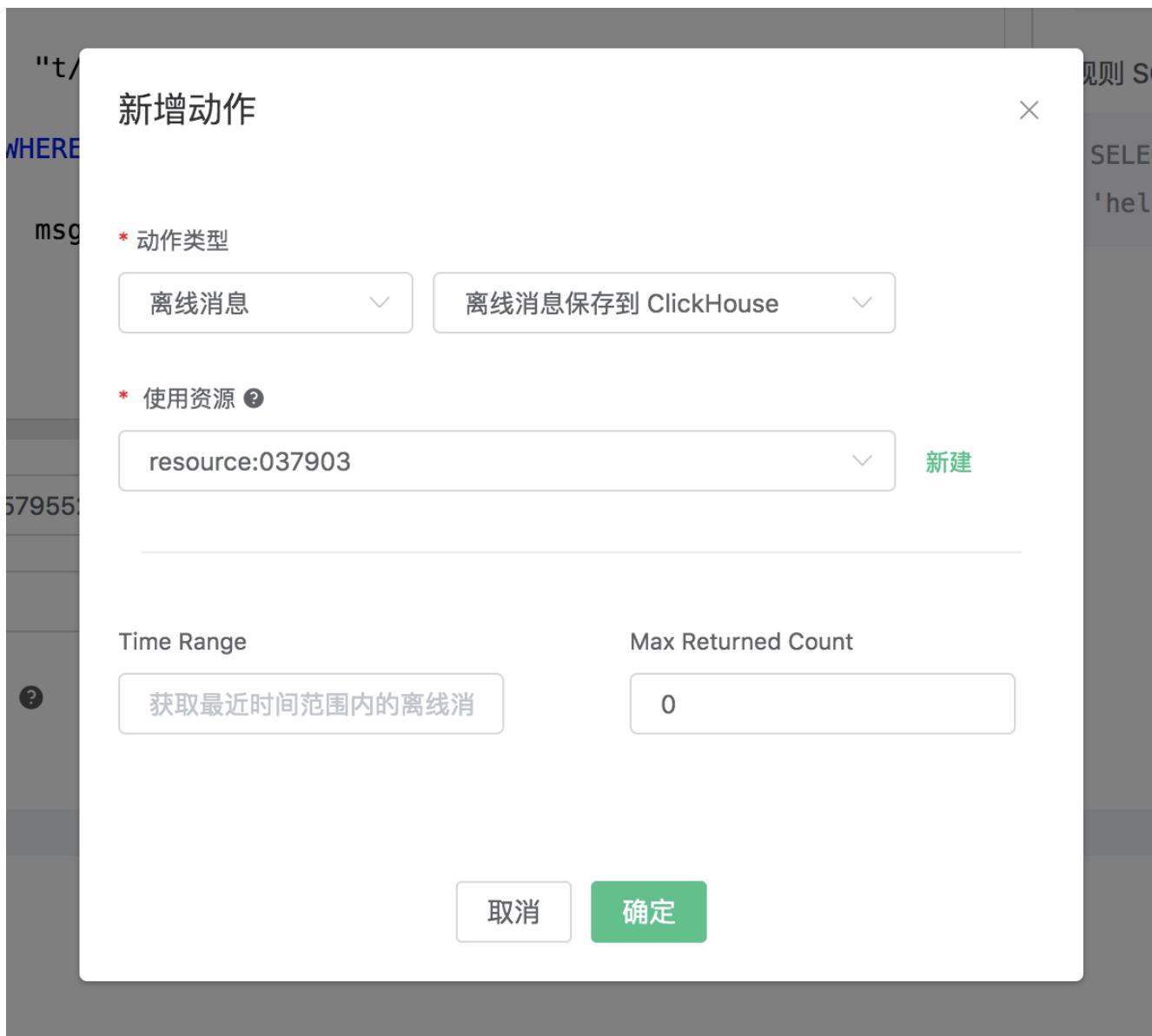
ClickHouse 密码

取消 确定

填写资源配置：

填写真实的 **ClickHouse** 服务器地址，其他配置填写相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

The screenshot shows the "Response Actions" section of the rule configuration. It includes the following elements:

- * 响应动作 (Response Action):** A dropdown menu showing "离线消息保存到 ClickHouse (offline_msg_to_clickhouse)".
- 处理命中规则的消息 (Handle messages matched by the rule):** A note indicating this action handles the matched messages.
- Actions Summary:** A bar showing the action type, count, resource ID, and time range.
- Buttons:** At the bottom are two buttons: "取消" (Cancel) on the left and a green "创建" (Create) button on the right.

返回规则创建界面，点击“创建”。

ID	主题	监控	描述	状态	响应动作
1 rule:393327	t/# \$events/session_subscribed \$events/message_acked			<input checked="" type="checkbox"/>	离线消息保存到 ClickHouse 编辑 删除

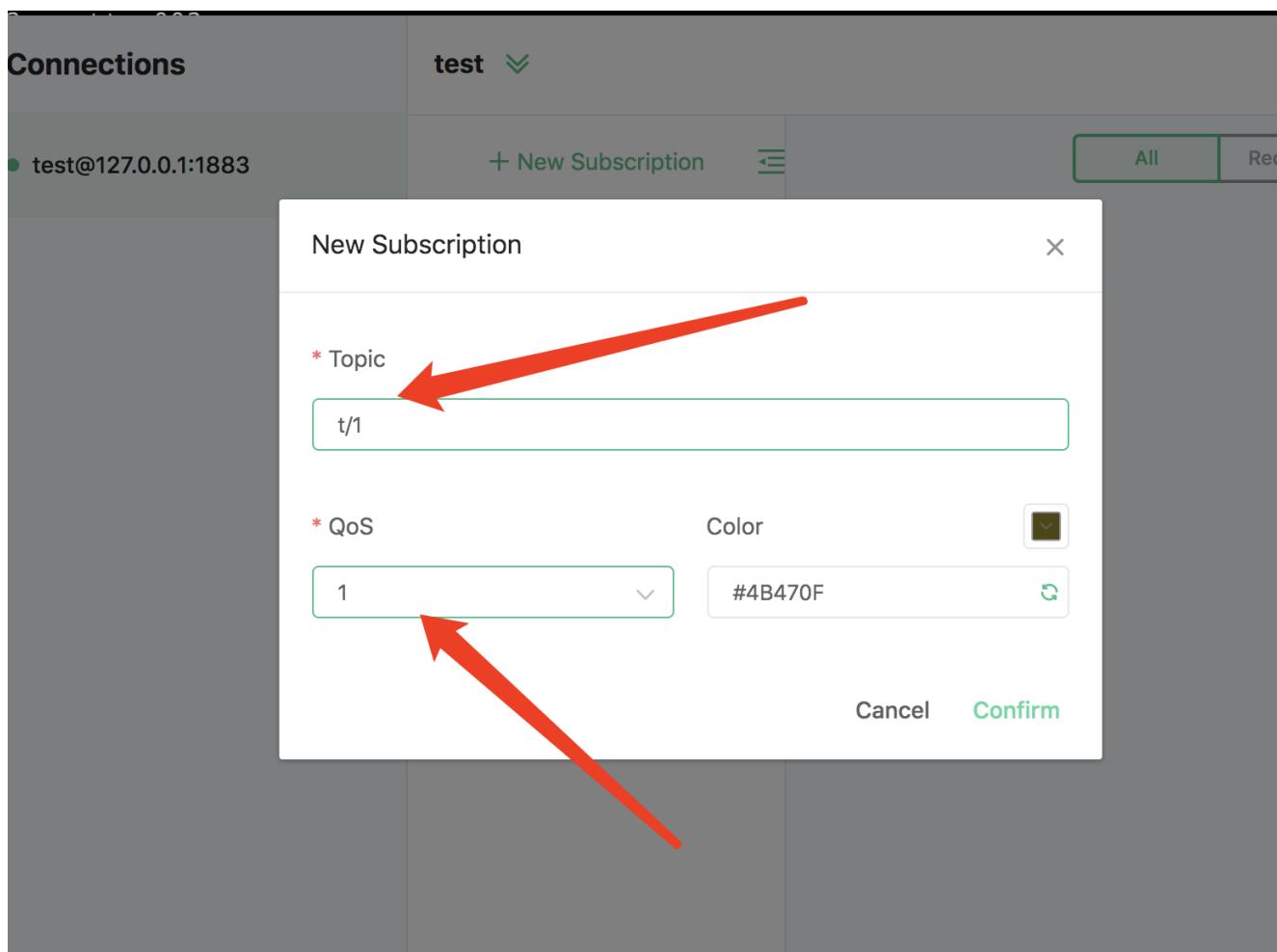
规则已经创建完成，通过 **Dashboard** 的 **WebSocket** 客户端发一条数据**(发布消息的**QoS**必须大于0)**：

The screenshot shows the '发布' (Publish) section of the EMQX Dashboard. It includes fields for Topic (t/1), Payload (hello), and QoS (1). A 'Retain' checkbox is unchecked. A green '发布' (Publish) button is visible. Below the form, two sections show message history: '已接收' (Received) and '已发送' (Sent). The '已发送' section lists a single message: Topic t/1, QoS 1, Payload hello, and Time 10:09:10. A note '暂无数据' (No data available) is present.

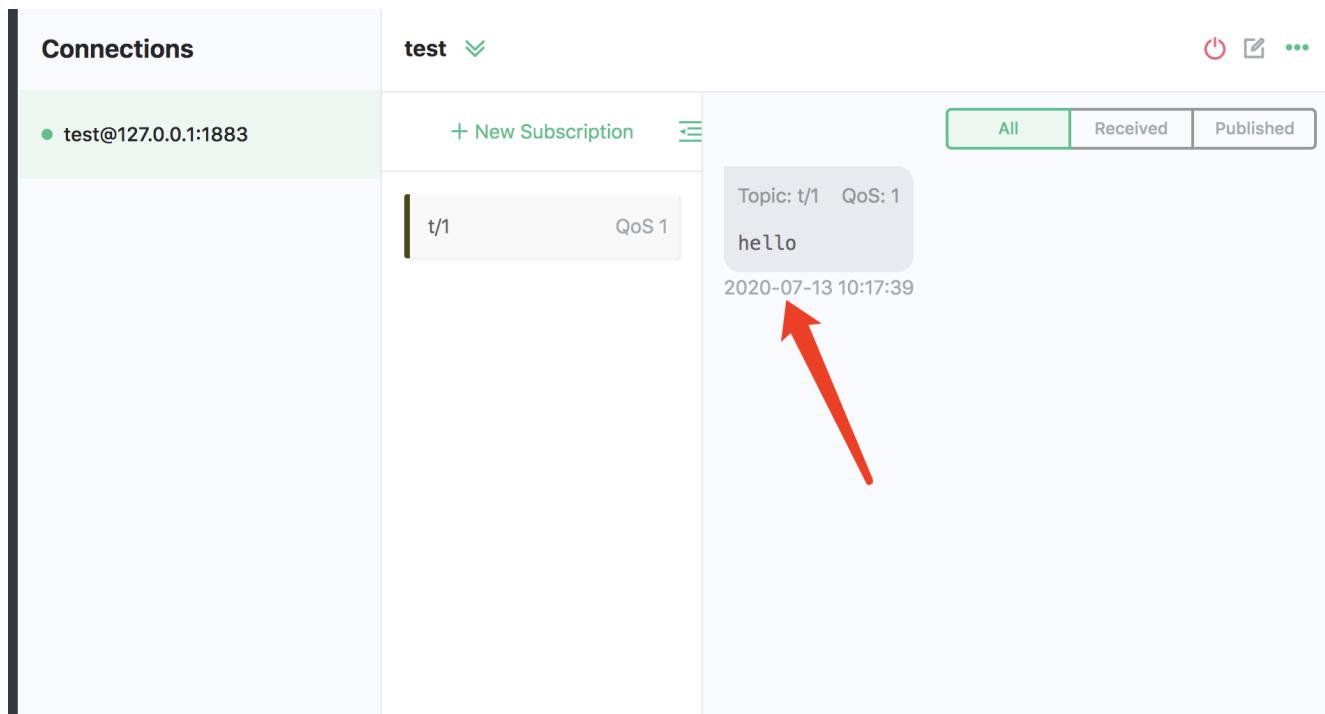
消息发送后，通过 **cqlsh** 查看到消息被保存到 **ClickHouse** 里面：

```
3f1a96942178 :){ select * from mqtt_msg;
SELECT *
FROM mqtt_msg
[msgid] 5BD8A81D6F2B0F44200000A780002 [sender] mqttjs_e67abf4 [topic] t/1 [qos] 1 [retain] 0 [payload] hello [arrived] 1615777465037
1 rows in set. Elapsed: 0.002 sec.
3f1a96942178 :){
```

使用另外一个客户端，订阅主题 "t/1" (订阅主题的**QoS**必须大于0，否则消息会被重复接收)：



订阅后马上接收到了保存到 **ClickHouse** 里面的离线消息:



离线消息被接收后会在 **ClickHouse** 中删除:

```
3f1a96942178 :)
3f1a96942178 :) select * from mqtt_msg;

SELECT *
FROM mqtt_msg



| msgid                         | sender          | topic | qos | retain | payload | arrived       |
|-------------------------------|-----------------|-------|-----|--------|---------|---------------|
| 5BD8A81D6F2B0F44200000A780002 | mqttjs_ee67abf4 | t/1   | 1   | 0      | hello   | 1615777465037 |



1 rows in set. Elapsed: 0.002 sec.

3f1a96942178 :)
3f1a96942178 :)
3f1a96942178 :)
3f1a96942178 :) select * from mqtt_msg;

SELECT *
FROM mqtt_msg

Ok.

0 rows in set. Elapsed: 0.002 sec.

3f1a96942178 :)
```

从 Redis 中获取订阅关系

搭建 Redis 环境，以 MacOS X 为例：

```

1 $ wget http://download.redis.io/releases/redis-4.0.14.tar.gz
2 $ tar xzf redis-4.0.14.tar.gz
3 $ cd redis-4.0.14
4 $ make && make install
5
6 # 启动 redis
7 $ redis-server

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 SQL：

```
1 SELECT * FROM "$events/client_connected"
```

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入:

```

1 SELECT
2 *
3
4
5 FROM
6
7 "$events/client_connected"

```

当前事件可用字段

event	clientid	username	mountpoint	peername	sockname
proto_name	proto_ver	keepalive	clean_start	expiry_interval	
is_bridge	connected_at	timestamp	node		

规则 SQL 示例

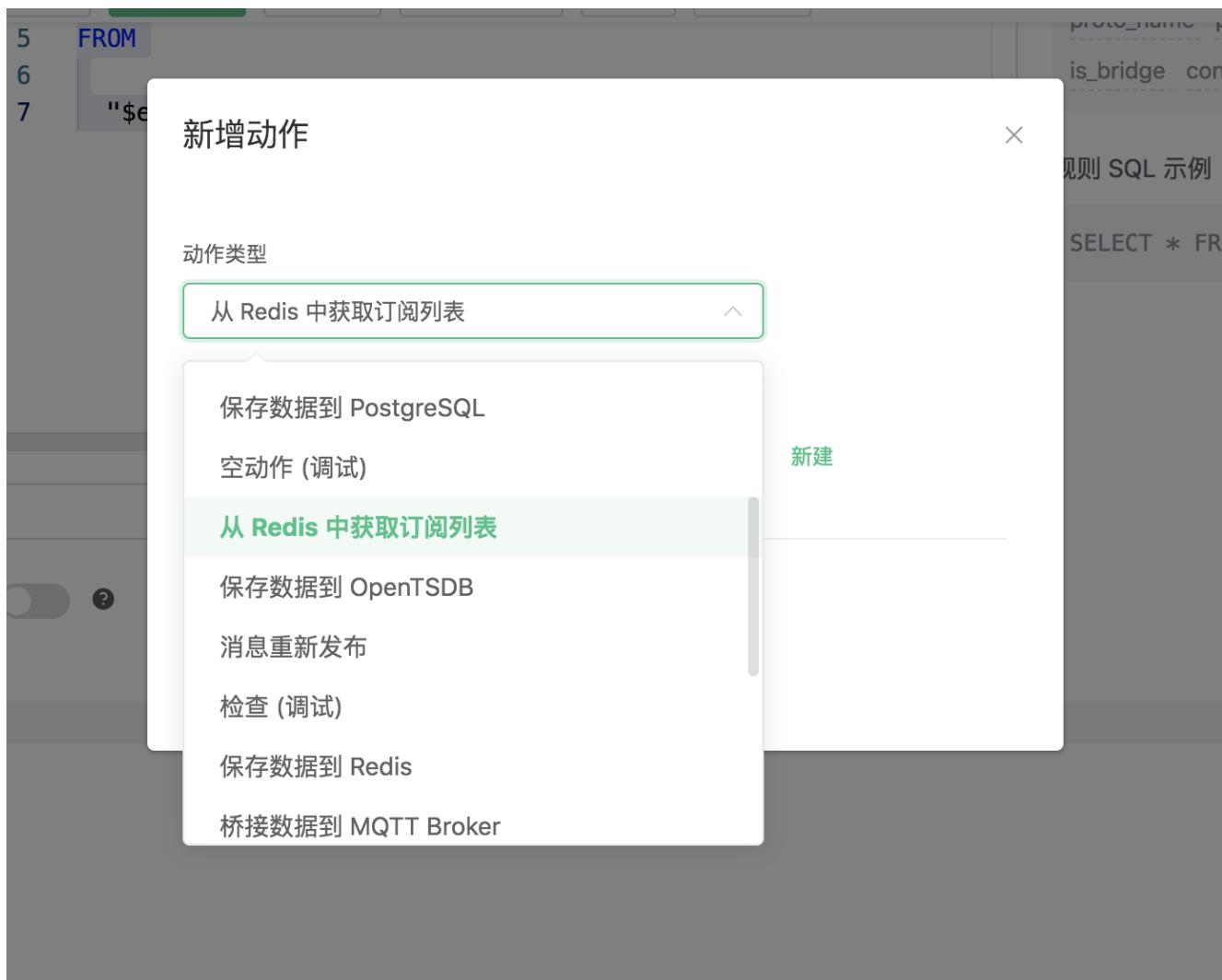
```
SELECT * FROM "$events/client_connected"
```

备注:

SQL 测试:

关联动作：

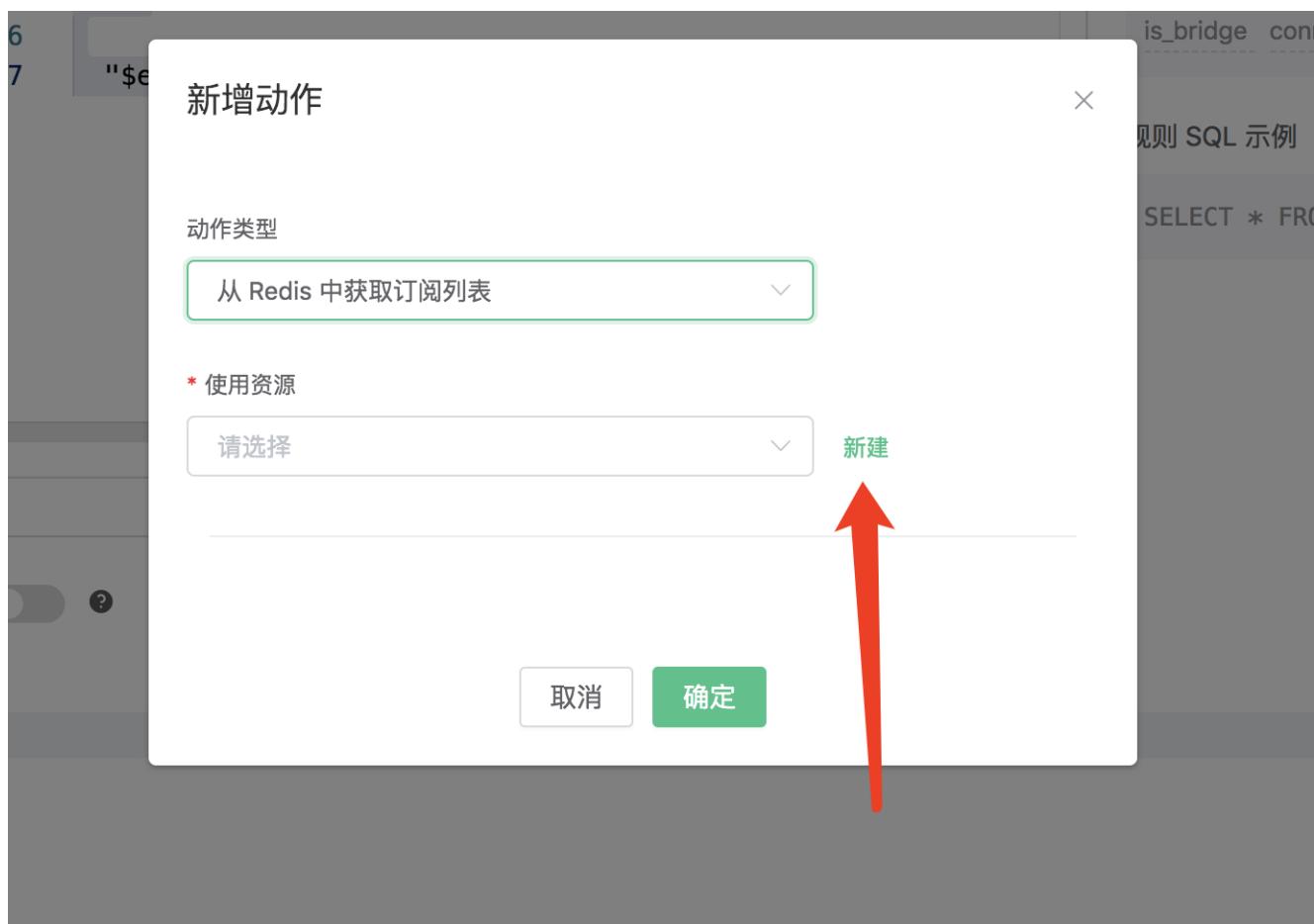
在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“从 Redis 中获取订阅关系”。



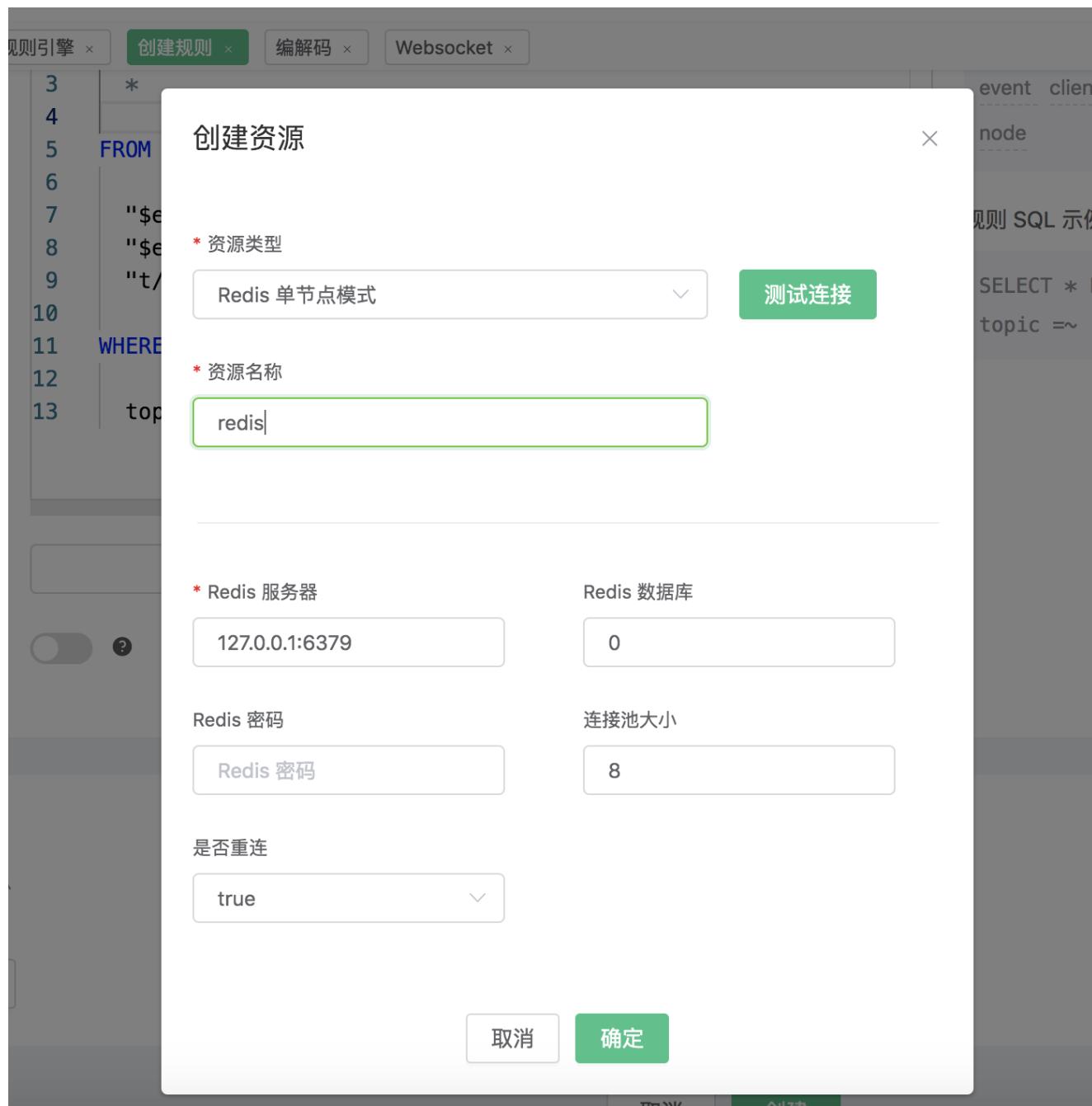
填写动作参数:

“从 Redis 中获取订阅列表”动作需要一个参数:

- 1). 关联资源。现在资源下拉框为空, 可以点击右上角的“新建资源”来创建一个 Redis 资源:



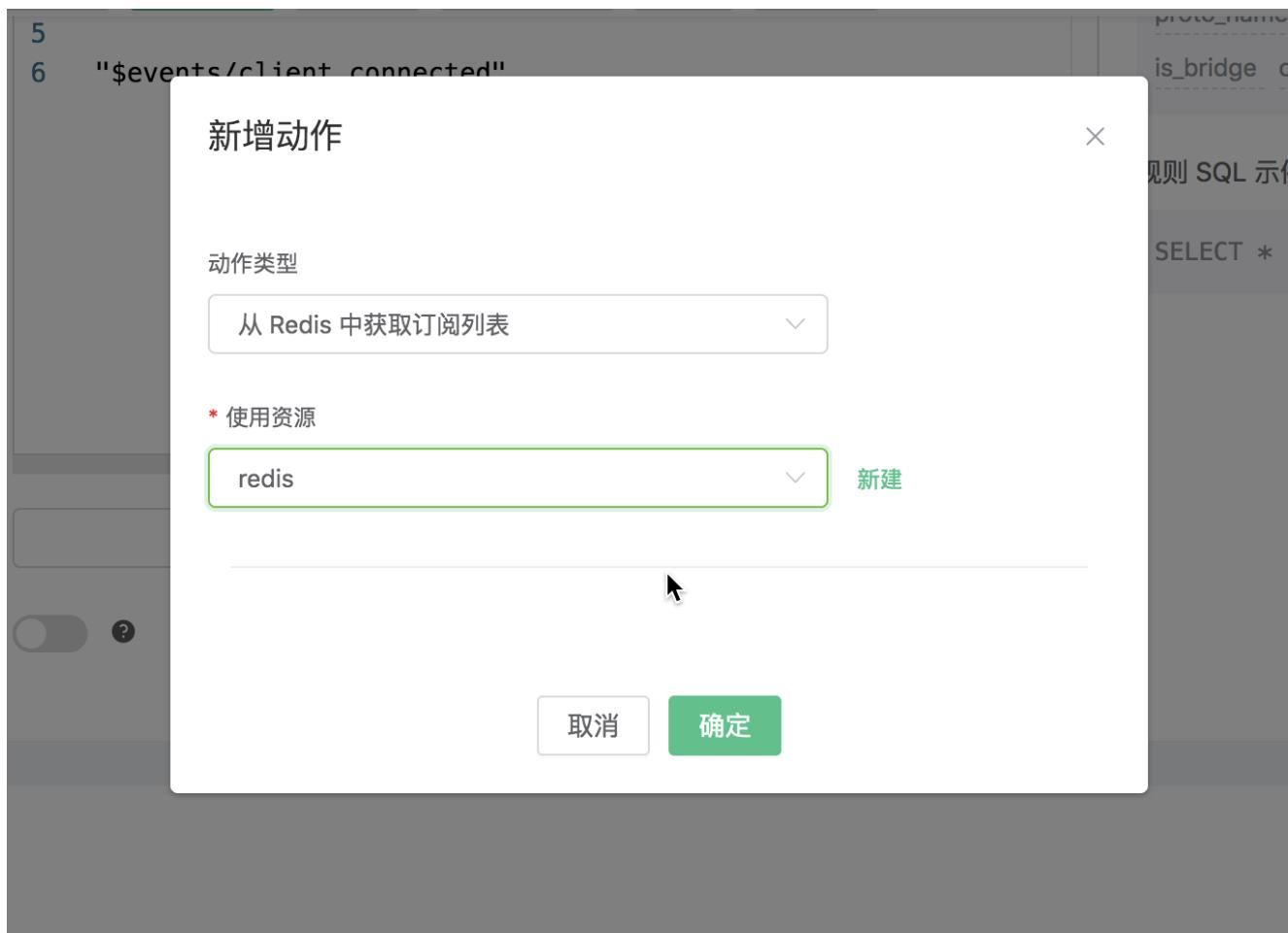
选择 **Redis** 单节点模式资源”。



填写资源配置：

填写真实的 **Redis** 服务器地址，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 从 Redis 中获取订阅列表 (lookup_sub_to_redis)	编辑 移除
从 Redis 中获取订阅列表	
资源 ID resource:7d6ff195	+ 失败备选动作
+ 添加动作	

取消 确定

返回规则创建界面，点击“新建”。

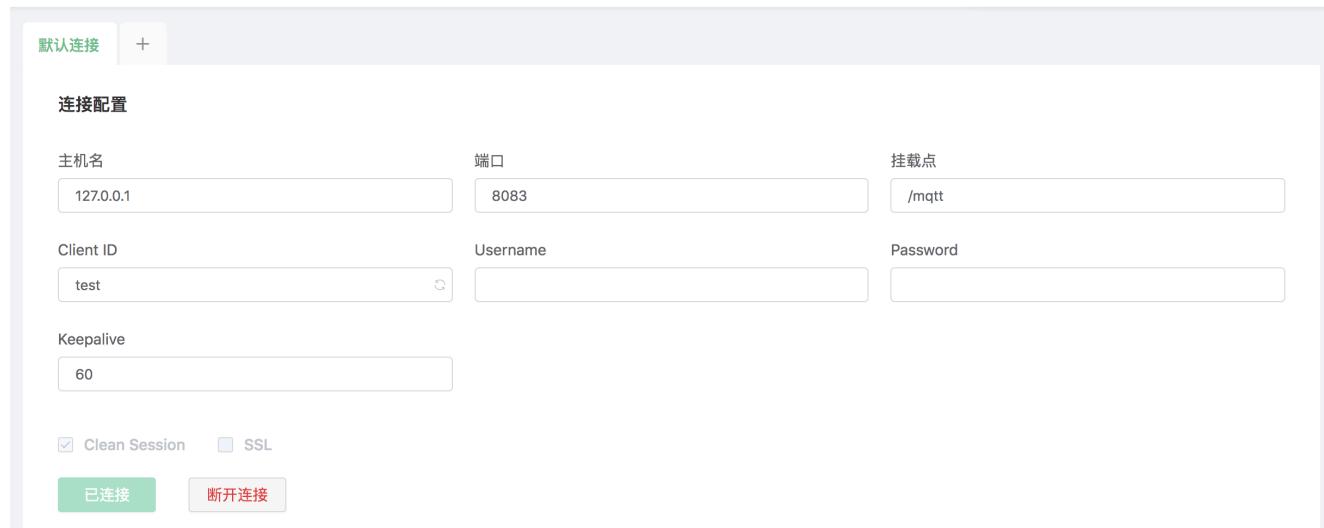
规则列表					
创建	ID	主题	监控	描述	响应动作
+ 创建	rule:da6d9ed8	\$events/client_connected	on	<input checked="" type="checkbox"/>	从 Redis 中获取订阅列表 编辑 删除

规则已经创建完成，通过 **Redis CLI** 往Redis插入一条订阅关系：

```
1 HSET mqtt:sub:test t1 1
sh
```

```
127.0.0.1:6379> hset mqtt:sub:test t1 1
(integer) 1
127.0.0.1:6379> HGETALL mqtt:sub:test
1) "t1"
2) "1"
127.0.0.1:6379> 
```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备:



查看订阅列表，可以看到 **test** 设备已经订阅了 **t1** 主题:

The screenshot shows the '当前订阅主题列表' (Current Subscribed Topics) section of the EMQX Dashboard. The sidebar on the left has the '订阅' (Subscriptions) link selected. The main area displays a table with columns: '客户端 ID' (Client ID), '主题' (Topic), and 'QoS'. There is one entry: 'test' under '客户端 ID', 't1' under '主题', and '1' under 'QoS'. At the top of this section, there are search and filter inputs, and buttons for '搜索' (Search), '重置' (Reset), and '展开' (Expand).

从 MySQL 中获取订阅关系

搭建 MySQL 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```
1 $ brew install mysql
2
3 $ brew services start mysql
4
5 $ mysql -u root -h localhost -p
6
7 ALTER USER 'root'@'localhost' IDENTIFIED BY 'public';
```

初始化 MySQL 数据库：

```
1 $ mysql -u root -h localhost -ppublic
2
3 create database mqtt;
```

创建 **mqtt_sub** 表：

```
1 DROP TABLE IF EXISTS `mqtt_sub`;
2
3 CREATE TABLE `mqtt_sub` (
4   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
5   `clientid` varchar(64) DEFAULT NULL,
6   `topic` varchar(180) DEFAULT NULL,
7   `qos` tinyint(1) DEFAULT NULL,
8   PRIMARY KEY (`id`),
9   KEY `mqtt_sub_idx` (`clientid`, `topic`, `qos`),
10  UNIQUE KEY `mqtt_sub_key` (`clientid`, `topic`),
11  INDEX topic_index(`id`, `topic`)
12 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

提示

订阅关系表结构不能修改，请使用上面SQL语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 SQL：

```
1 SELECT * FROM "$events/client_connected"
```

创建规则

使用 SQL 设定规则，对消息数据筛选、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

The screenshot shows the 'Create Rule' page in the EMQX Enterprise interface. On the left, there is a code editor for writing SQL queries. The current query is:

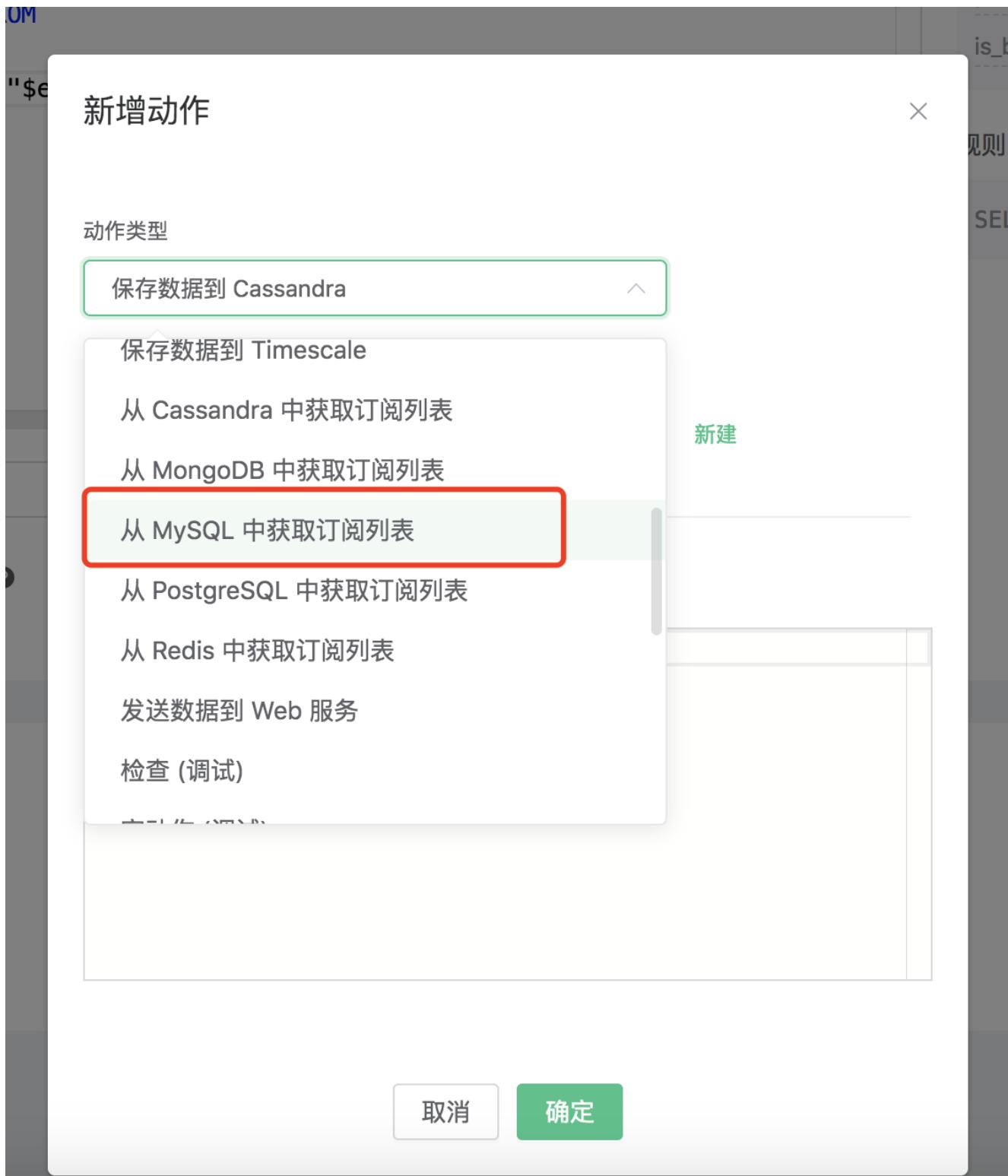
```
1 SELECT
2 *
3 FROM
4 "$events/client_connected"
```

To the right of the editor, there are several sections:

- 当前事件可用字段**: A list of available fields for the current event, including: event, clientid, username, mountpoint, peername, sockname, proto_name, proto_ver, keepalive, clean_start, expiry_interval, is_bridge, connected_at, timestamp, node.
- 规则 SQL 示例**: A preview of the generated SQL query: `SELECT * FROM "$events/client_connected"`.
- 备注:** A text input field for notes.
- SQL 测试:** A toggle switch and a help icon (?) for testing the SQL query.

关联动作:

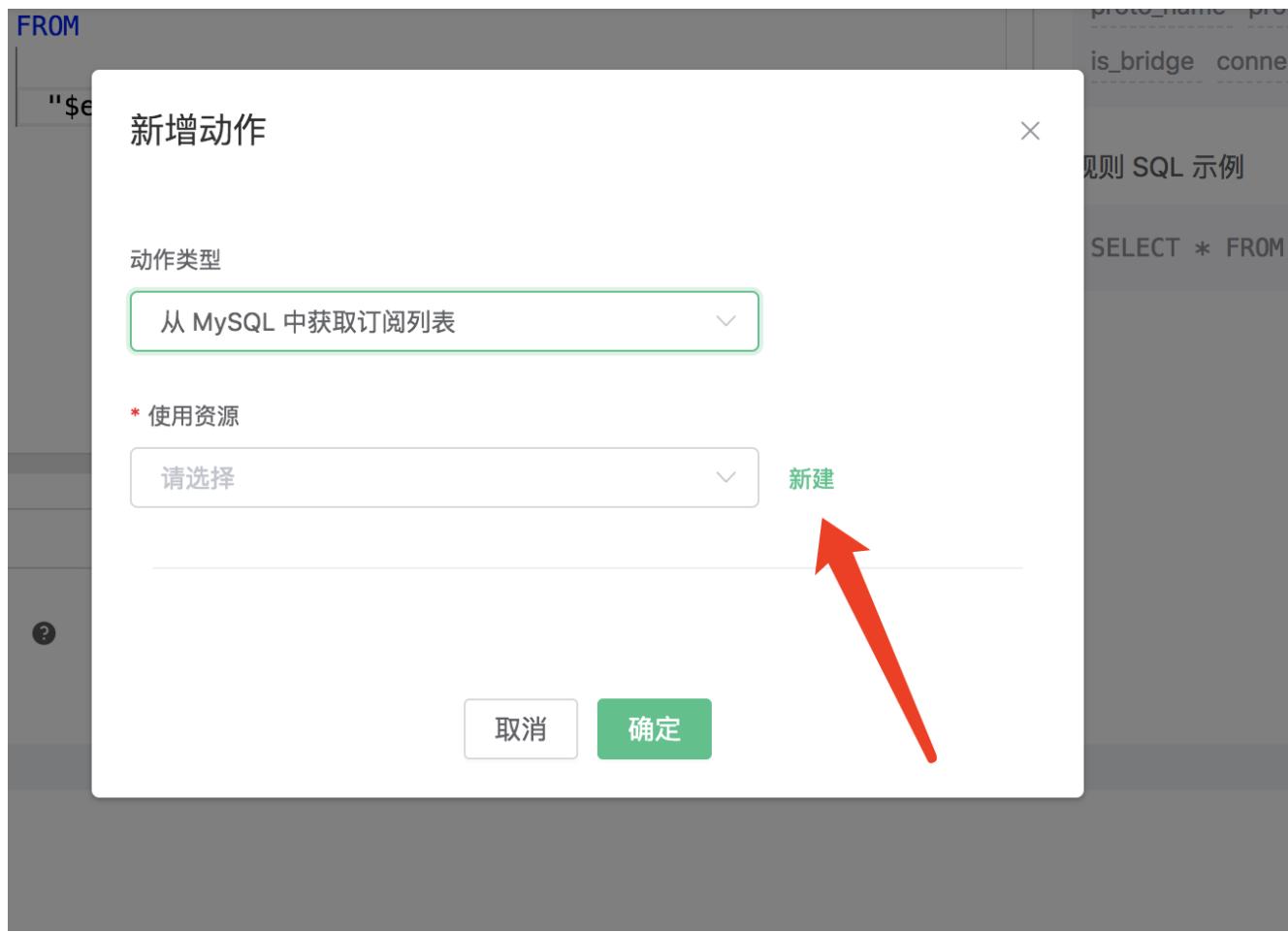
在“响应动作”界面选择“添加动作”，然后在“新增动作”下拉框里选择“从MySQL中获取订阅列表”



填写动作参数：

“从 MySQL 中获取订阅列表”动作需要一个参数：

- 1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建”来创建一个 MySQL 资源：



创建资源

X

* 资源类型

MySQL

测试连接

* 资源名称

请输入

* MySQL 服务器

192.168.1.173:3306

* MySQL 数据库名

mqtt

连接池大小

8

* MySQL 用户名

root

MySQL 密码

public

批量写入大小

100

批量写入间隔(毫秒)

10

是否重连

true

取消

确定

填写资源配置：

填写真实的 **MySQL** 服务器地址，其他配置相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



Return to the response action interface, click "Confirm".

Return to the rule creation interface, click "Create".

The rule has been created successfully, through "mysql" to MySQL insert a subscription relationship:

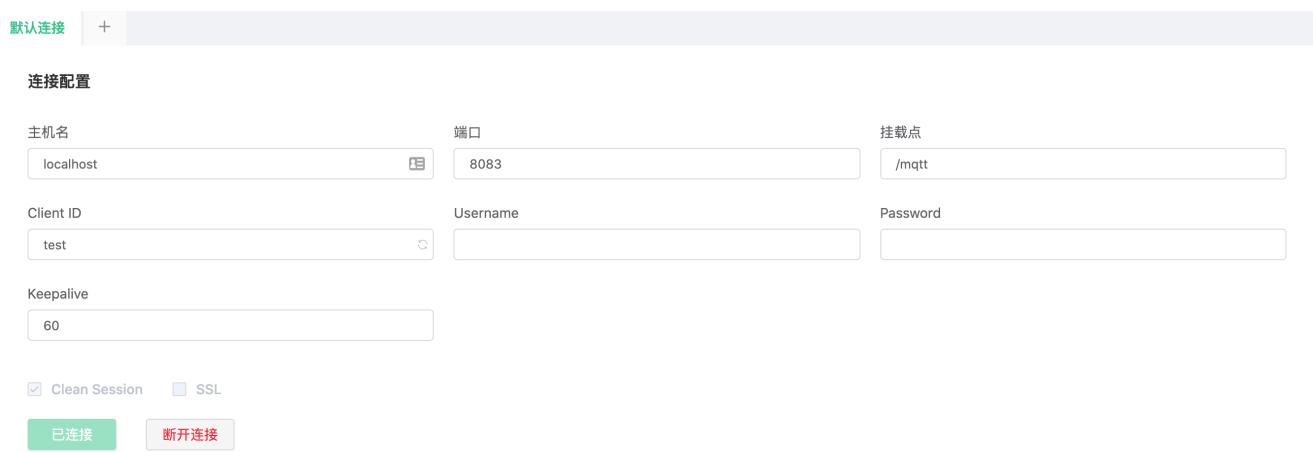
```
1     insert into mqtt_sub(clientid, topic, qos) values("test", "t1", 1);
```

```
mysql>
mysql> insert into mqtt_sub(clientid, topic, qos) values("test", "t1", 1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from mqtt_sub;
+----+-----+-----+
| id | clientid | topic | qos |
+----+-----+-----+
| 1  | test     | t1    | 1   |
+----+-----+-----+
1 row in set (0.00 sec)

mysql> ■
```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备:



查看“订阅”列表，可以看到 **Broker** 从 **MySQL** 里面获取到订阅关系，并代理设备订阅:



从 PostgreSQL 中获取订阅关系

提示

支持 **PostgreSQL 13** 及以下版本

搭建 **PostgreSQL** 数据库，以 **MacOS X** 为例：

```
1 $ brew install postgresql
2 $ brew services start postgresql
```

sh

创建 **mqtt** 数据库：

```
1 # 使用用户名 postgres 创建名为 'mqtt' 的数据库
2 $ createdb -U postgres mqtt
3
4 $ psql -U postgres mqtt
5
6 mqtt=> \dn;
7   List of schemas
8   Name    | Owner
9   -----+-----
10  public | postgres
11  (1 row)
```

创建 **mqtt_sub** 表：

```
1 $ psql -U postgres mqtt
2 CREATE TABLE mqtt_sub(
3     id SERIAL8 primary key,
4     clientid character varying(64),
5     topic character varying(255),
6     qos integer,
7     UNIQUE (clientid, topic)
8 );
```

提示

订阅关系表结构不能修改，请使用上面**SQL**语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 **SQL**：

```
1 SELECT * FROM "$events/client_connected"
```

sh

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入:

```

1 SELECT
2 *
3 FROM
4 "$events/client_connected"

```

当前事件可用字段

event	clientid	username	mountpoint	peername	sockname
proto_name	proto_ver	keepalive	clean_start	expiry_interval	
is_bridge	connected_at	timestamp	node		

规则 SQL 示例

```
SELECT * FROM "$events/client_connected"
```

备注:

SQL 测试:

关联动作:

在“响应动作”界面选择“添加动作”，然后在“新增动作”下拉框里选择“从PostgreSQL中获取订阅列表”

新增动作

动作类型

- 从 PostgreSQL 中获取订阅列表
- 保存数据到 OpenTSDB
- 消息重新发布
- 检查 (调试)
- 保存数据到 Redis
- 桥接数据到 MQTT Broker
- 保存数据到 DynamoDB
- 保存数据到 MongoDB

当前事件可用字段

event	clientid	username	mountpoint	peername	sockname
proto_name	proto_ver	keepalive	clean_start	expiry_interval	
is_bridge	connected_at	timestamp	node		

规则 SQL 示例

```
SELECT * FROM "$events/client_connected"
```

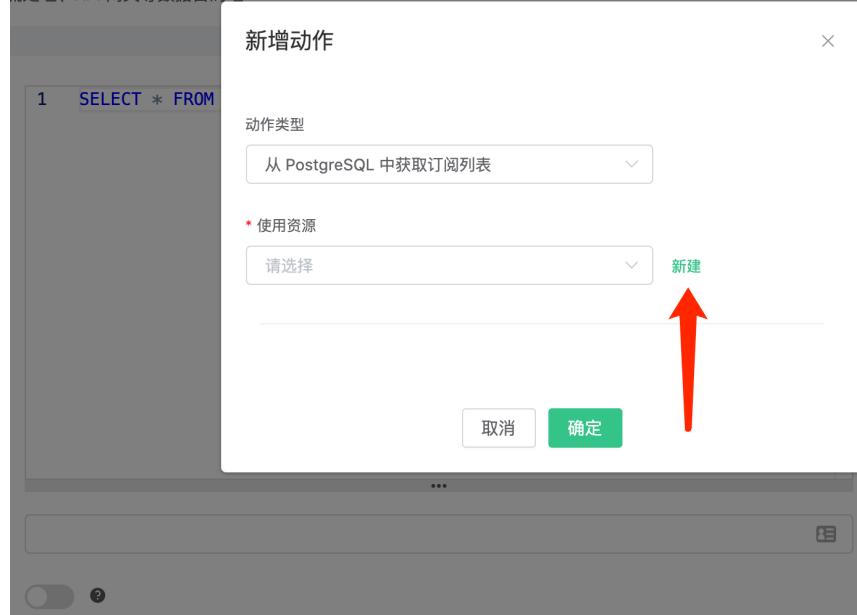
新建

填写动作参数:

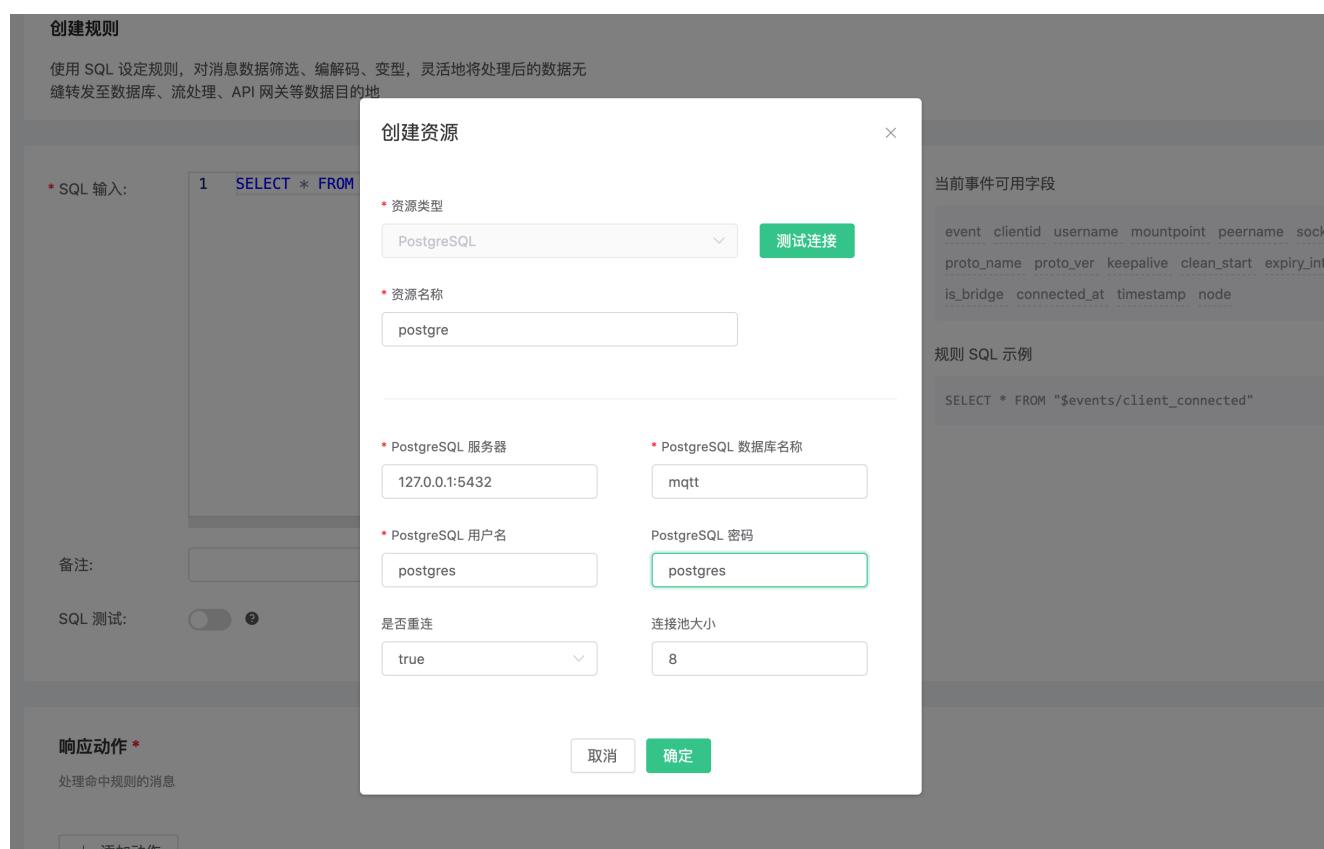
“从PostgreSQL中获取订阅列表” 动作需要一个参数:

- 1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建”来创建一个 PostgreSQL 资源:

对消息数据筛选、编解码、变型，灵活地将处理后的数据无缝处理、API 网关等数据目的地



弹出“创建资源”对话框



填写资源配置：

填写真实的 **PostgreSQL** 服务器地址，其他配置相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。

建规则

用 SQL 设定规则，对消息数据筛选、编解码、变型，灵活地将处理后的数据无转发至数据库、流处理、API 网关等数据目的地

返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

返回规则创建界面，点击“创建”。

规则已经创建完成，通过“**psql**”往**PostgreSQL**插入一条订阅关系

```

1 | insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1)

[ mqtt=> insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1);
INSERT 0 1
[ mqtt=> select * from mqtt_sub;
 id | clientid | topic | qos
----+-----+-----+
 1 | test     | t1    | 1
(1 行记录 )

```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备：

默认连接 +

连接配置

主机名 localhost	端口 8083	挂载点 /mqtt
Client ID test	Username	Password
Keepalive 60		

Clean Session SSL

已连接 断开连接

查看“订阅”列表，可以看到 Broker 从 PostgreSQL 里面获取到订阅关系，并代理设备订阅：

The screenshot shows the EMQX Enterprise web interface. On the left, there is a sidebar with the following navigation items:

- 监控 (Monitoring)
- 客户端 (Clients)
- 主题 (Topics)
- 订阅 (Subscriptions)** (highlighted in green)
- 规则引擎 (Rule Engine) (with a dropdown arrow)
- 规则 (Rules)
- 资源 (Resources)
- 编解码 (Coding/Decoding)

The main content area is titled "当前订阅主题列表" (Current Subscription Topic List). It includes a search bar with fields for "客户端 ID" (Client ID), "filter" (filter), and a search button. There is also a "重置" (Reset) button and a "展开" (Expand) button.

客户端 ID	主题	QoS
test	t1	1

从 Cassandra 中获取订阅关系

搭建 Cassandra 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```

1 $ brew install cassandra
2 ## 修改配置, 关闭匿名认证
3 $ vim /usr/local/etc/cassandra/cassandra.yaml
4
5     authenticator: PasswordAuthenticator
6     authorizer: CassandraAuthorizer
7
8 $ brew services start cassandra
9
10 ## 创建 root 用户
11 $ cqlsh -ucassandra -pcassandra
12
13 create user root with password 'public' superuser;

```

创建 "mqtt" 表空间：

```

1 $ cqlsh -uroot -ppublic
2
3 CREATE KEYSPACE mqtt WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}
4   AND durable_writes = true;

```

创建 **mqtt_sub** 表：

```

1
2 CREATE TABLE mqtt_sub (
3     clientid text,
4     topic text,
5     qos int,
6     PRIMARY KEY (clientid, topic)
7 ) WITH CLUSTERING ORDER BY (topic ASC)
8     AND bloom_filter_fp_chance = 0.01
9     AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
10    AND comment = ''
11    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
12        'max_threshold': '32', 'min_threshold': '4'}
13    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
14    AND crc_check_chance = 1.0
15    AND dclocal_read_repair_chance = 0.1
16    AND default_time_to_live = 0
17    AND gc_grace_seconds = 864000
18    AND max_index_interval = 2048
19    AND memtable_flush_period_in_ms = 0
20    AND min_index_interval = 128
21    AND read_repair_chance = 0.0
22    AND speculative_retry = '99PERCENTILE';

```

提示

订阅关系表结构不能修改, 请使用上面SQL语句创建

创建规则:

打开 [EMQX Dashboard](#), 选择左侧的“规则”选项卡。

然后填写规则 SQL:

```
1 SELECT * FROM "$events/client_connected" sh
```

创建规则

使用 SQL 设定规则, 对消息数据筛选、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

The screenshot shows the EMQX Dashboard's rule configuration page. On the left, there is a code editor-like area for writing SQL. The current query is:

```
1 SELECT
2 *
3
4 FROM
5
6 "$events/client_connected"
```

To the right of the editor, there is a list of "当前事件可用字段" (Available fields for current events) which includes:

- event
- clientid
- username
- mountpoint
- peername
- sockname
- proto_name
- proto_ver
- keepalive
- clean_start
- expiry_interval
- is_bridge
- connected_at
- timestamp
- node

Below the editor, there is a "规则 SQL 示例" (Rule SQL example) section containing the same SQL query.

At the bottom of the editor, there is a "备注:" (Notes:) input field and a "SQL 测试:" (Test SQL) button with a help icon.

关联动作:

在“响应动作”界面选择“添加动作”, 然后在“新增动作”下拉框里选择“从 Cassandra 中获取订阅列表”



填写动作参数：

“从 Cassandra 中获取订阅列表”动作需要一个参数：

- 1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建”来创建一个 Cassandra 资源：



弹出“创建资源”对话框

创建资源

* 资源类型

Cassandra

测试连接

* 资源名称

cassandra

* Cassandra 服务器

192.168.1.173:9042

* Cassandra Keyspace

mqtt

Cassandra 用户名

root

Cassandra 密码

public

是否重连

true

连接池大小

8

取消 确定

取消 创建

填写资源配置：

填写真实的 **Cassandra** 服务器地址，其他配置相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 从 Cassandra 中获取订阅列表 (lookup_sub_to_cassa)
从 Cassandra 中获取订阅列表

资源 ID resource:b7b31156

编辑 移除
+ 失败备选动作

+ 添加动作

取消 创建

返回规则创建界面，点击“创建”。

ID	主题	监控	描述	状态	响应动作
rule:74881dde	\$events/client_connected	on		<input checked="" type="checkbox"/>	从 Cassandra 中获取 订阅列表 编辑 删除

+ 创建

规则已经创建完成，通过 “**cqlsh**” 往 **Cassandra** 插入一条订阅关系：

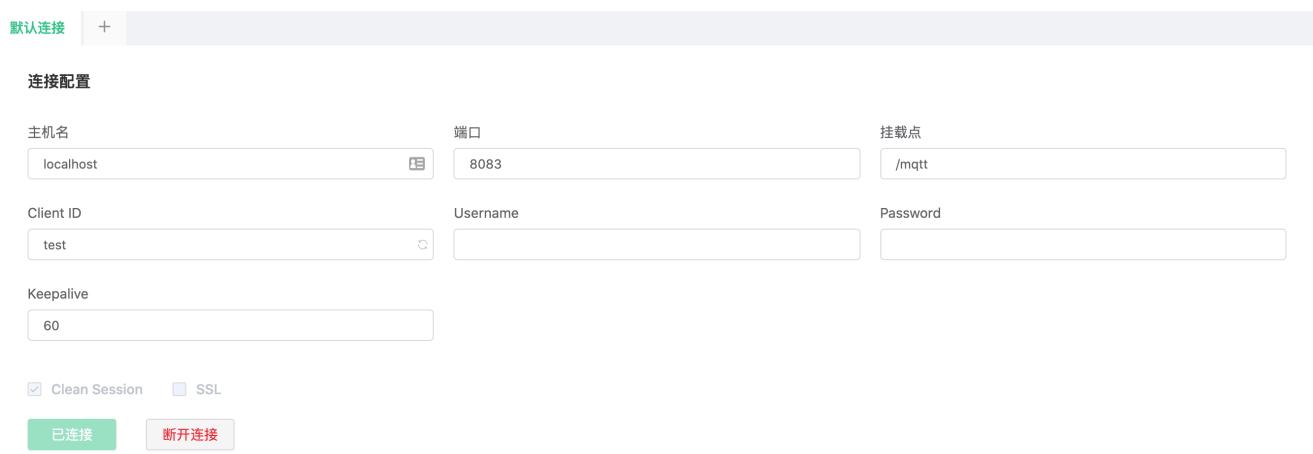
```
1     insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1);
```

```
root@cqlsh:mqtt> insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1);
root@cqlsh:mqtt> select * from mqtt_sub;

clientid | topic | qos
-----+-----+-----
test   |    t1 |    1

(1 rows)
root@cqlsh:mqtt>
```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备：



查看“订阅”列表，可以看到 **Broker** 从 **Cassandra** 里面获取到订阅关系，并代理设备订阅：

The screenshot shows the "Subscriptions" list in the EMQX Dashboard. The left sidebar has the "订阅" (Subscriptions) link selected. The main area is titled "当前订阅主题列表" (Current Subscribed Topics). It shows a table with columns: "客户端 ID" (Client ID), "主题" (Topic), and "QoS". There is one entry: "test" is subscribed to "t1" with QoS 1. At the top right of the table, there are buttons for "搜索" (Search), "重置" (Reset), and "展开" (Expand).

从 MongoDB 中获取订阅关系

搭建 **MongoDB** 数据库，并设置用户名密码为 **root/public**，以 **MacOS X** 为例：

```

1 $ brew install mongodb
2 $ brew services start mongodb
3
4 ## 新增 root/public 用户
5 $ use mqtt;
6 $ db.createUser({user: "root", pwd: "public", roles: [{role: "readWrite", db: "mqtt"}]});
7
8 ## 修改配置，关闭匿名认证
9 $ vi /usr/local/etc/mongod.conf
10
11     security:
12         authorization: enabled
13
14 $ brew services restart mongodb

```

创建 **mqtt_sub** 表：

```

1 $ mongo 127.0.0.1/mqtt -uroot -ppublic
2 db.createCollection("mqtt_sub");

```

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 **SQL**：

```

1 SELECT * FROM "$events/client_connected"

```

创建规则

使用 SQL 设定规则，对消息数据筛选、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

* SQL 输入：

```

1 SELECT
2 *
3
4
5 FROM
6
7 "$events/client_connected"

```

当前事件可用字段

event	clientid	username	mountpoint	peername	sockname
proto_name	proto_ver	keepalive	clean_start	expiry_interval	
is_bridge	connected_at	timestamp	node		

规则 SQL 示例

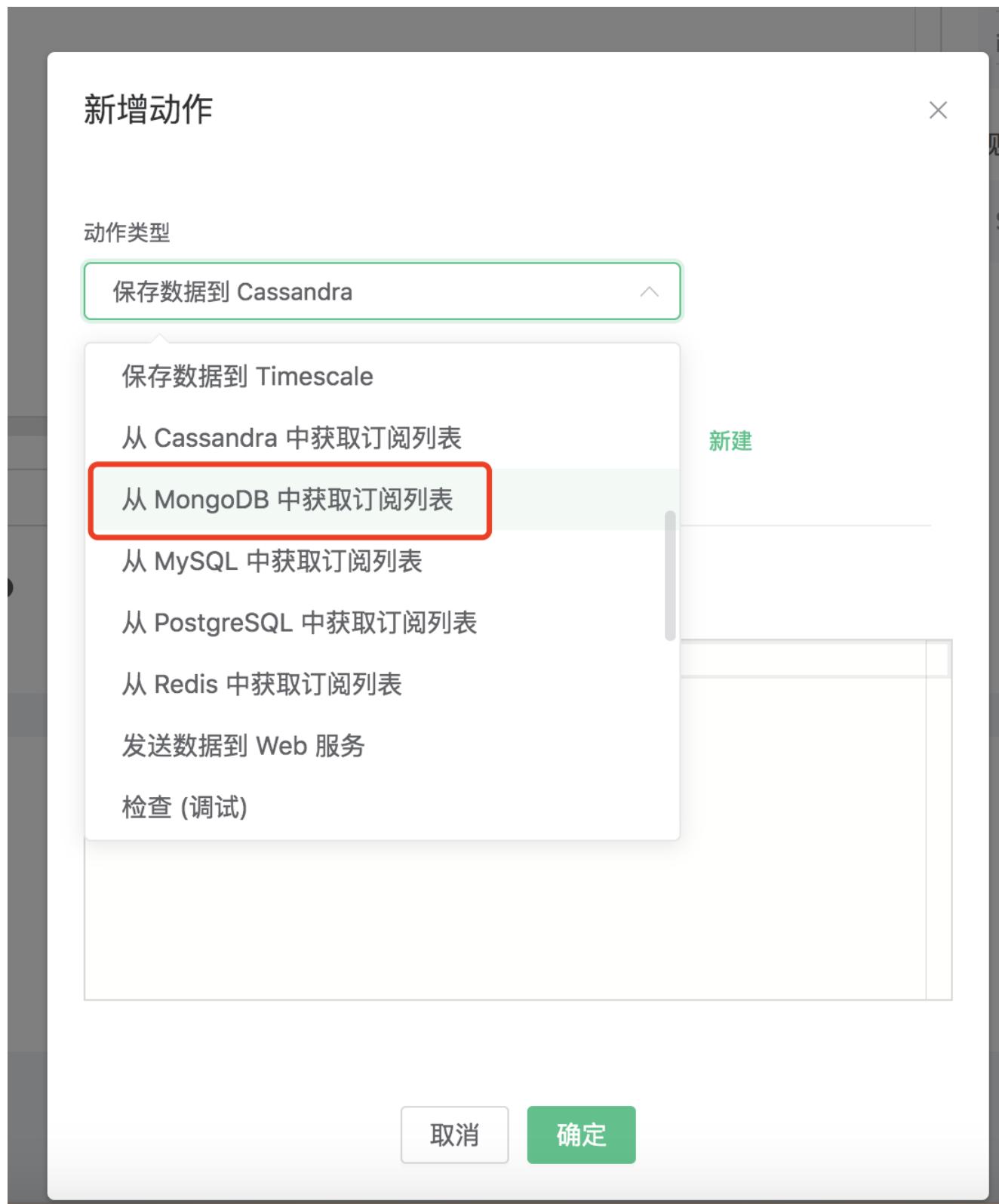
```
SELECT * FROM "$events/client_connected"
```

备注：

SQL 测试：

关联动作:

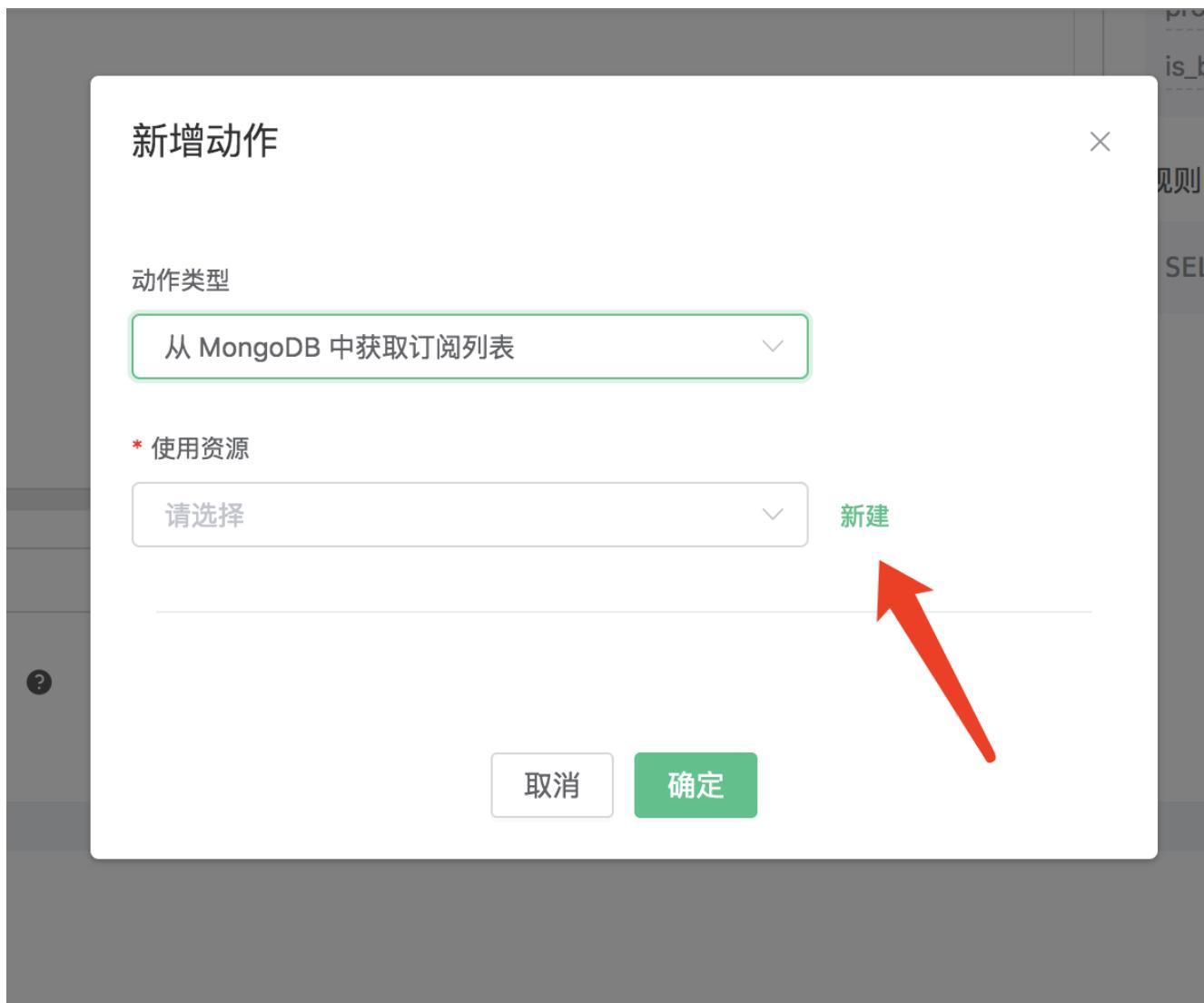
在“响应动作”界面选择“添加动作”，然后在“新增动作”下拉框里选择“从MongoDB中获取订阅列表”



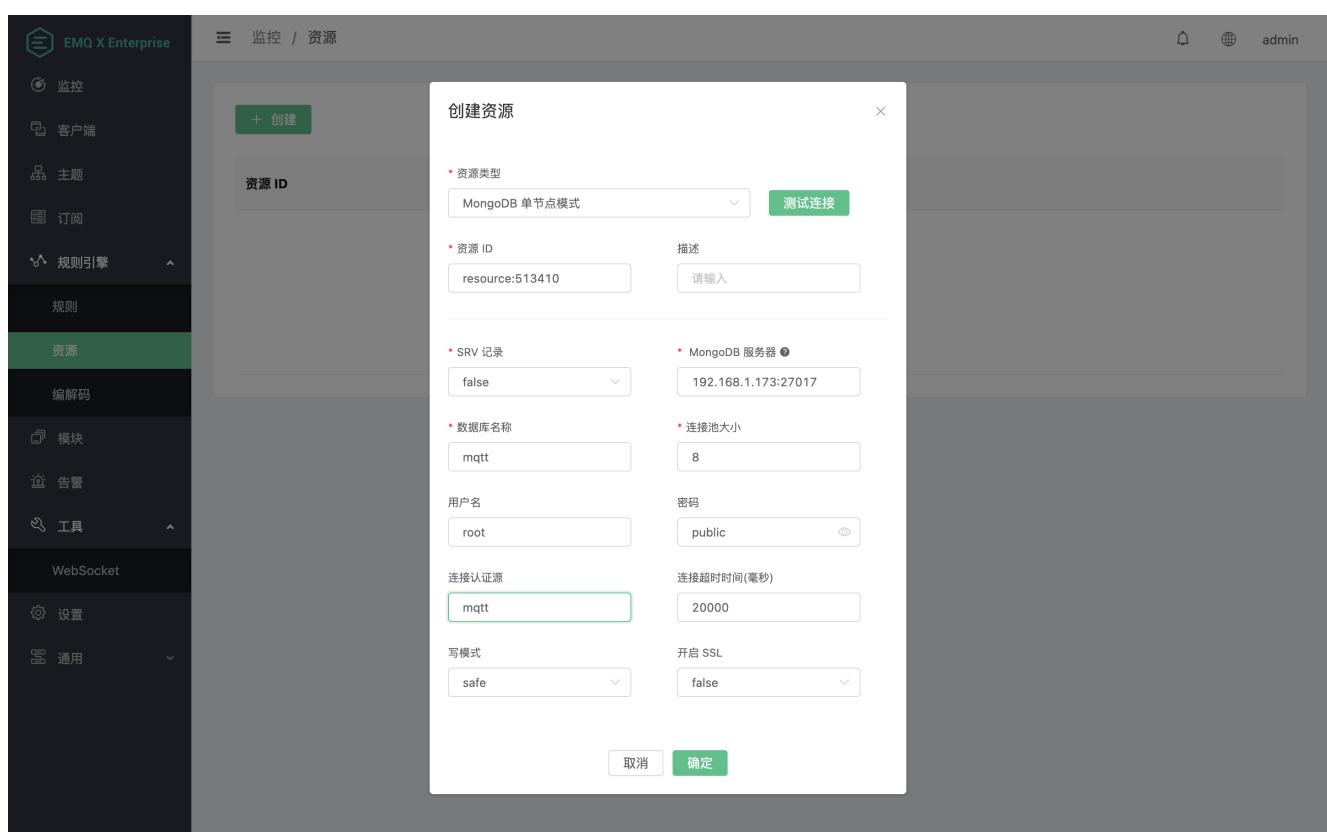
填写动作参数:

“从 MongoDB 中获取订阅列表”动作需要一个参数:

- 1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建”来创建一个 MongoDB 资源：



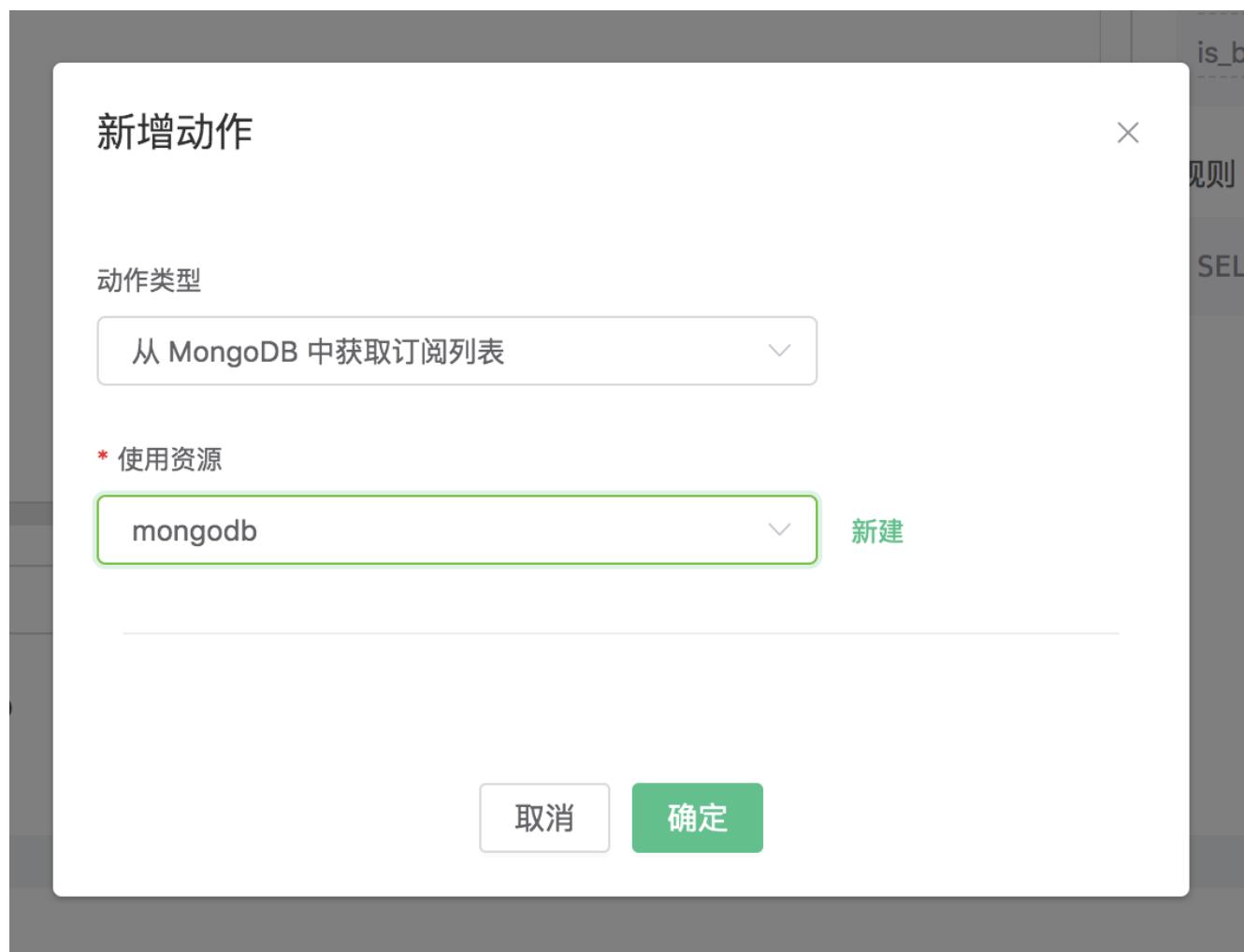
弹出“创建资源”对话框



填写资源配置：

填写真实的 **MongoDB** 服务器地址，其他配置相应的值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“确定”按钮。



返回响应动作界面，点击“确认”。

响应动作 *

处理命中规则的消息

动作类型 从 MongoDB 中获取订阅列表 (lookup_sub_to_mongo)
从 MongoDB 中获取订阅列表
资源 ID resource:5ad5b4f0

编辑 移除

+ 失败备选动作

+ 添加动作

取消 创建

返回规则创建界面，点击“创建”。

+ 创建

ID	主题	监控	描述	状态	响应动作
rule:6e43e913	\$events/client_connected	full		<input checked="" type="checkbox"/>	从 MongoDB 中获取 订阅列表

规则已经创建完成，通过 “**mongo**” 往**MongoDB**插入一条订阅关系

```
1 db.mqtt_sub.insert({clientid: "test", topic: "t1", qos: 1})
```

```
> db.createCollection("mqtt_sub");
{ "ok" : 1 }
> db.mqtt_sub.insert({clientid: "test", topic: "t1", qos: 1})
WriteResult({ "nInserted" : 1 })
> db.mqtt_sub.find({clientid: "test"})
{ "_id" : ObjectId("5f2a6b62ed7fc5638f902f75"), "clientid" : "test", "topic" : "t1", "qos" : 1 }
> 
```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备：

默认连接 +

连接配置

主机名	端口	挂载点
localhost	8083	/mqtt
Client ID	Username	Password
test		
Keepalive		
60		
<input checked="" type="checkbox"/> Clean Session <input type="checkbox"/> SSL		
<input type="button" value="已连接"/> <input type="button" value="断开连接"/>		

查看“订阅”列表，可以看到 **Broker** 从 **MongoDB** 里面获取到订阅关系，并代理设备订阅：

监控

客户端

主题

订阅

规则引擎

规则

资源

编解码

当前订阅主题列表

emqx@127.0.0.1

客户端 ID	主题	QoS
test	t1	1

从 Redis 中获取订阅关系

搭建 ClickHouse 数据库，并设置用户名密码为 **default/public**，以 CentOS 为例：

```

1 ## 安装依赖
2 sudo yum install -y epel-release
3
4 ## 下载并运行packagecloud.io提供的安装shell脚本
5 curl -s https://packagecloud.io/install/repositories/altinity/clickhouse/script.rpm.sh | sudo bash
6
7
8 ## 安装ClickHouse服务器和客户端
9 sudo yum install -y clickhouse-server clickhouse-client
10
11 ## 启动ClickHouse服务器
12 clickhouse-server
13
14 ## 启动ClickHouse客户端程序
clickhouse-client

```

创建 “**mqtt**” 数据库：

```

1 create database mqtt;

```

创建 **mqtt_sub** 表：

```

1 use mqtt;
2 create table mqtt_sub (
3     clientid String,
4     topic String,
5     qos Nullable(Int8) DEFAULT 0
6     ) engine = MergeTree() ORDER BY clientid;

```

提示

消息表结构不能修改，请使用上面SQL语句创建

创建规则：

打开 [EMQX Dashboard](#)，选择左侧的“规则”选项卡。

然后填写规则 SQL：

```

1 SELECT * FROM "$events/client_connected"

```

创建规则

使用 SQL 设定规则，对消息数据筛选、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

The screenshot shows the 'SQL 输入' (SQL Input) field containing the following SQL query:

```

1 SELECT
2 *
3 FROM
4 "$events/client_connected"

```

To the right of the input field is a panel titled '当前事件可用字段' (Available Fields for Current Event) listing various message fields:

- event
- clientid
- username
- mountpoint
- peername
- sockname
- proto_name
- proto_ver
- keepalive
- clean_start
- expiry_interval
- is_bridge
- connected_at
- timestamp
- node

Below the input field is a '备注:' (Remarks) text input and an 'SQL 测试:' (SQL Test) button.

关联动作：

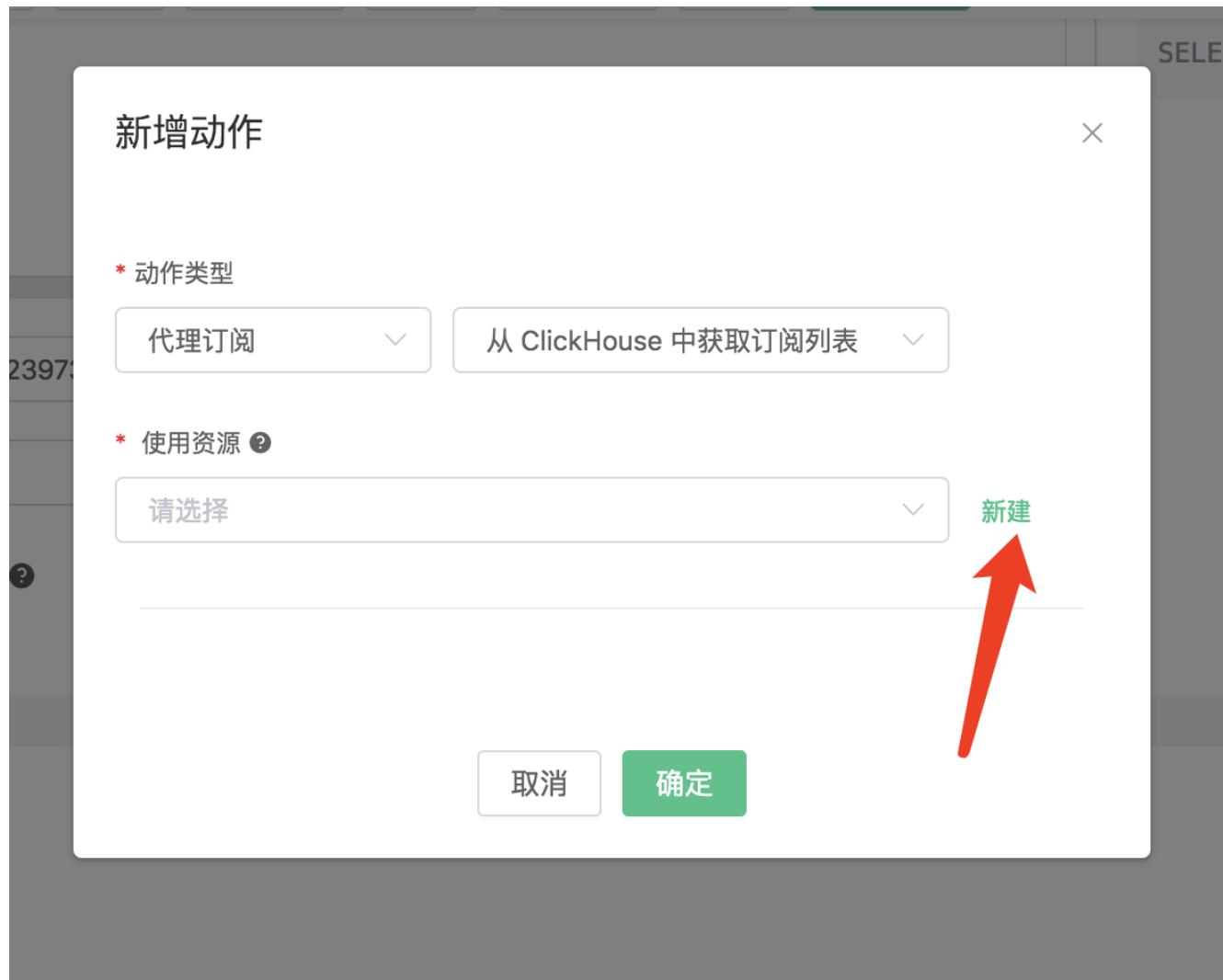
在“响应动作”界面选择“添加动作”，然后在“动作”下拉框里选择“从 ClickHouse 中获取订阅关系”。



填写动作参数：

“从 ClickHouse 中获取订阅列表”动作需要一个参数：

1). 关联资源。现在资源下拉框为空，可以点击右上角的“新建资源”来创建一个 **ClickHouse** 资源：



选择 **ClickHouse** 单节点模式资源"。

创建资源

* 资源类型

ClickHouse

测试连接

* 资源 ID

resource:006900

描述

请输入

* ClickHouse 服务器

http://192.168.0.172:8123

连接池大小

8

ClickHouse 数据库名

ClickHouse 数据库名

ClickHouse 用户名

ClickHouse 用户名

ClickHouse 密码

ClickHouse 密码

取消 确定

The screenshot shows a modal dialog titled "Create Resource" for creating a ClickHouse resource. The dialog has the following fields:

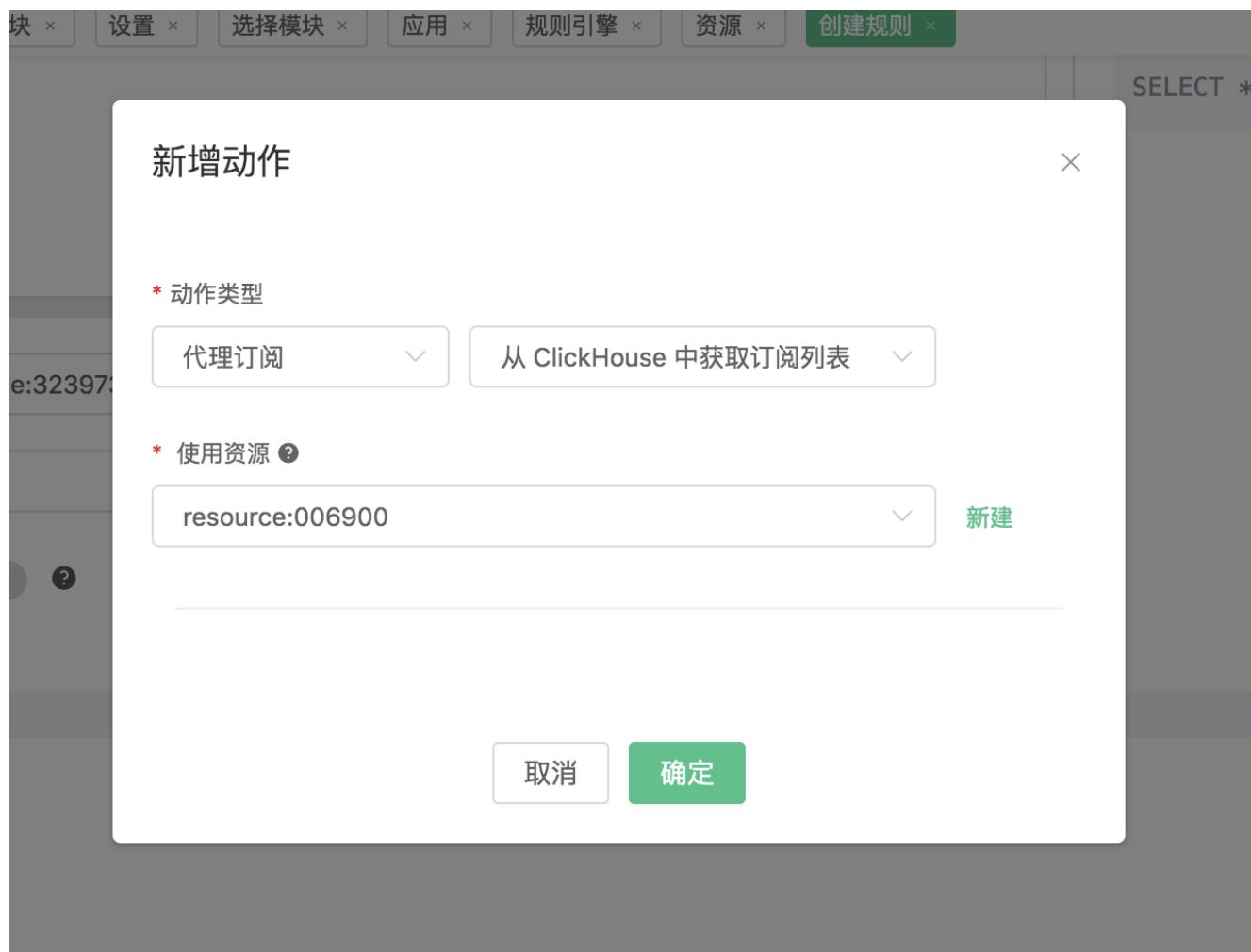
- Resource Type:** ClickHouse (selected)
- Resource ID:** resource:006900
- Description:** Please input
- ClickHouse Server:** http://192.168.0.172:8123 (highlighted with a green border)
- Connection Pool Size:** 8
- ClickHouse Database Name:** ClickHouse Database Name
- ClickHouse Username:** ClickHouse Username
- ClickHouse Password:** ClickHouse Password

At the bottom are "Cancel" and "OK" buttons.

填写资源配置：

填写真实的 **ClickHouse** 服务器地址，其他配置保持默认值，然后点击“测试连接”按钮，确保连接测试成功。

最后点击“新建”按钮。



返回响应动作界面，点击“确认”。

* 响应动作
处理命中规则的消息

动作类型 从 ClickHouse 中获取订阅列表 (lookup_sub_to_clickhouse)
从 ClickHouse 中获取订阅列表
资源 ID resource:006900

编辑 移除
+ 失败备选动作

+ 添加动作

取消 创建

返回规则创建界面，点击“新建”。

ID	主题	监控	描述	状态	响应动作
1 rule:323973	\$events/client_connected	正常		<input checked="" type="checkbox"/>	从 ClickHouse 中获取订阅列表 编辑 删除

规则已经创建完成，通过 **ClickHouse** 命令行往 **ClickHouse** 插入一条订阅关系：

1

```
insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1);
```

sh

```
3f1a96942178 :) insert into mqtt_sub(clientid, topic, qos) values('test', 't1', 1);
```

```
INSERT INTO mqtt_sub (clientid, topic, qos) VALUES
```

```
Ok.
```

```
1 rows in set. Elapsed: 0.002 sec.
```

```
3f1a96942178 :)
```

```
3f1a96942178 :)
```

```
3f1a96942178 :)
```

```
3f1a96942178 :) select * from mqtt_sub;
```

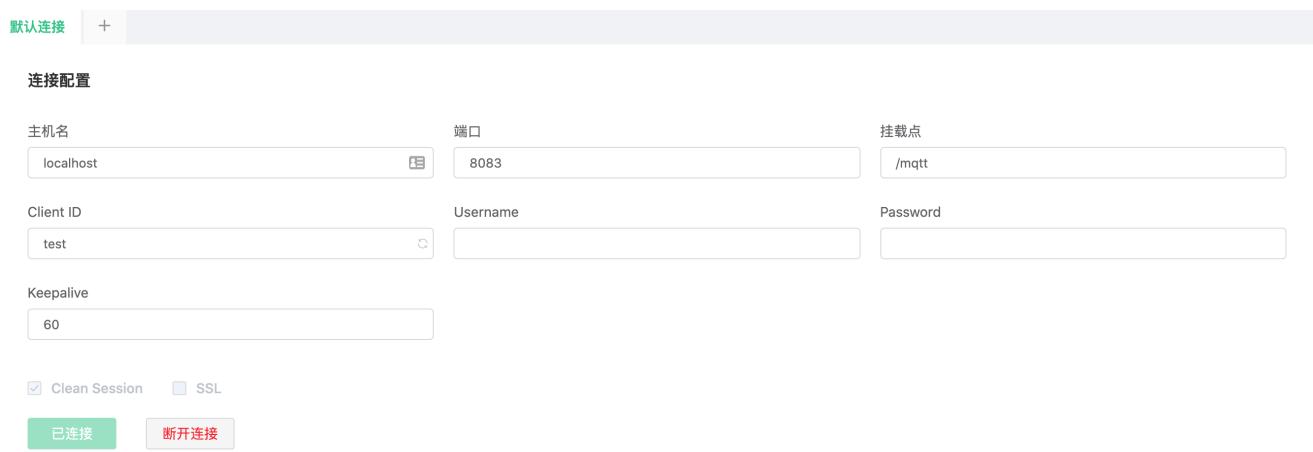
```
SELECT *
```

```
FROM mqtt_sub
```

clientid	topic	qos
test	t1	1

```
1 rows in set. Elapsed: 0.002 sec.
```

通过 **Dashboard** 登录 **clientid** 为 **test** 的设备:



查看订阅列表，可以看到 **test** 设备已经订阅了 **t1** 主题:

The screenshot shows the '订阅' (Subscriptions) section of the EMQX Dashboard. On the left is a sidebar with options like 监控, 客户端, 主题, 订阅 (highlighted in green), 规则引擎, 规则, 资源, and 编解码. The main area displays a table titled '当前订阅主题列表' (Current Subscribed Topics) with columns '客户端 ID', '主题', and 'QoS'. A single row is shown: 'test' | 't1' | 1. There are also search and filter buttons at the top of the table.

编解码（Schema Registry）介绍

物联网设备终端种类繁杂，各厂商使用的编码格式各异，所以在接入物联网平台的时候就产生了统一数据格式的需求，以便平台之上的应用进行设备管理。

Schema Registry 管理编解码使用的 **Schema**、处理编码或解码请求并返回结果。**Schema Registry** 配合规则引擎，可适配各种场景的设备接入和规则设计。

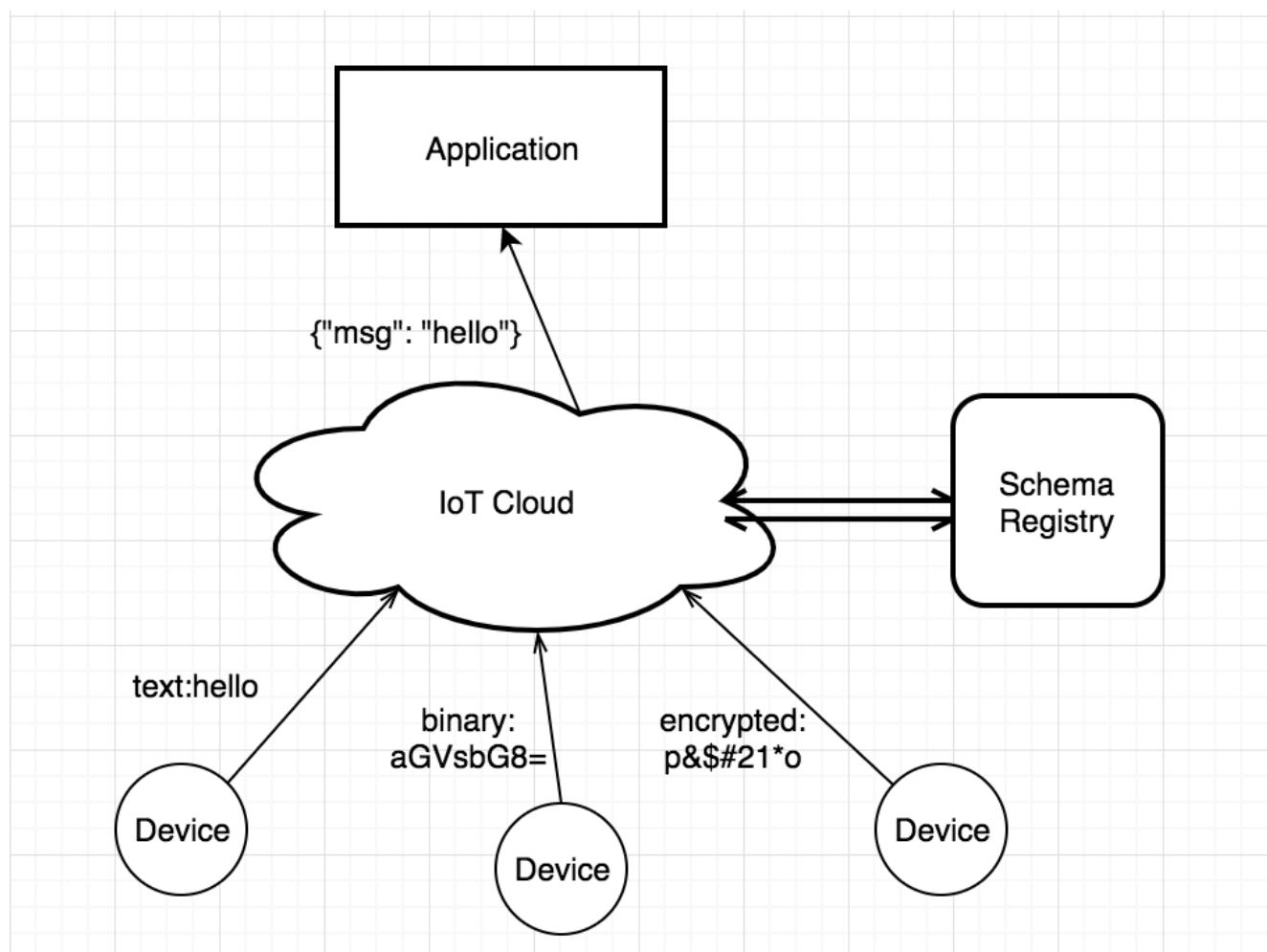
EMQX Schema Registry 目前可支持三种格式的编解码：[Avro](#)，[Protobuf](#)，以及自定义编码。其中 **Avro** 和 **Protobuf** 是依赖 **Schema** 的数据格式，编码后的数据为二进制，解码后为 **Map** 格式。解码后的数据可直接被规则引擎和其他插件使用。用户自定义的 **(3rd-party)** 编解码服务通过 **HTTP** 或 **GRPC** 回调的方式，进行更加贴近业务需求的编解码。

提示

Schema Registry 为 **Avro** 和 **Protobuf** 等内置编码格式维护 **Schema** 文本，但对于自定义编解码 **(3rd-party)** 格式，如需要，**Schema** 文本需由编解码服务自己维护

数据格式

下图展示了 **Schema Registry** 的一个应用案例。多个设备上报不同格式的数据，经过 **Schema Registry** 解码之后，变为统一的内部格式，然后转发给后台应用。



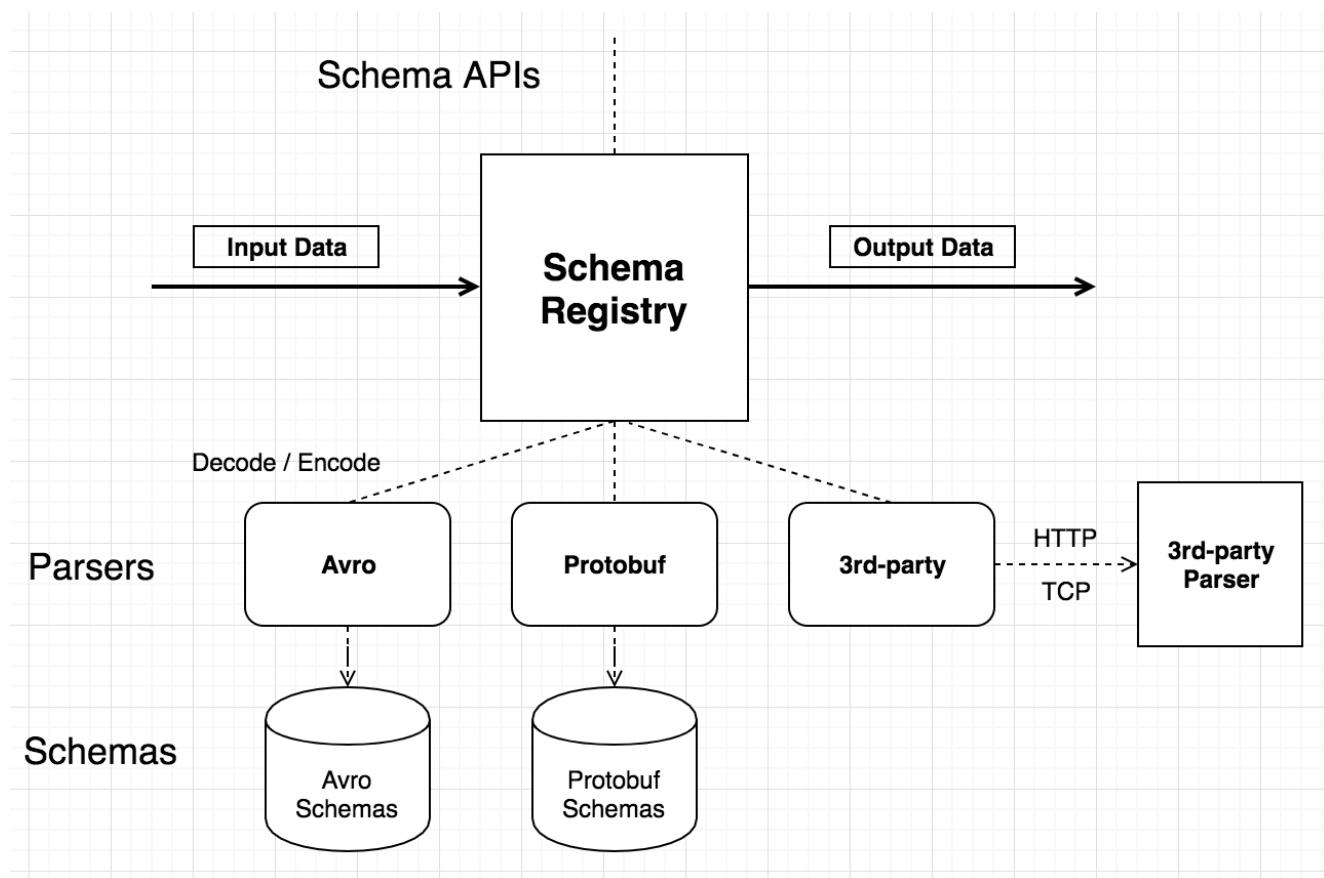
二进制格式支持

Schema Registry 数据格式包括 [Avro](#) 和 [Protobuf](#)。Avro 和 Protobuf 是依赖 **Schema** 的数据格式，编码后的数据为二进制，使用 **Schema Registry** 解码后的内部数据格式(**Map**，稍后讲解)可直接被规则引擎和其他插件使用。此外 **Schema Registry** 支持用户自定义的 (**3rd-party**) 编解码服务，通过 **HTTP** 或 **GRPC** 回调的方式，进行更加贴近业务需求的编解码。

架构设计

Schema Registry 为 Avro 和 Protobuf 等内置编码格式维护 **Schema** 文本，但对于自定义编解码 (**3rd-party**) 格式，如需要 **Schema**，**Schema** 文本需由编解码服务自己维护。**Schema API** 提供了通过 **Schema Name** 的添加、查询和删除操作。

Schema Registry 既可以解码，也可以编码。编码和解码时需要指定 **Schema Name**。



编码调用示例：参数为 **Schema**

```
1 schema_encode(SchemaName, Data) -> RawData
```

解码调用示例：

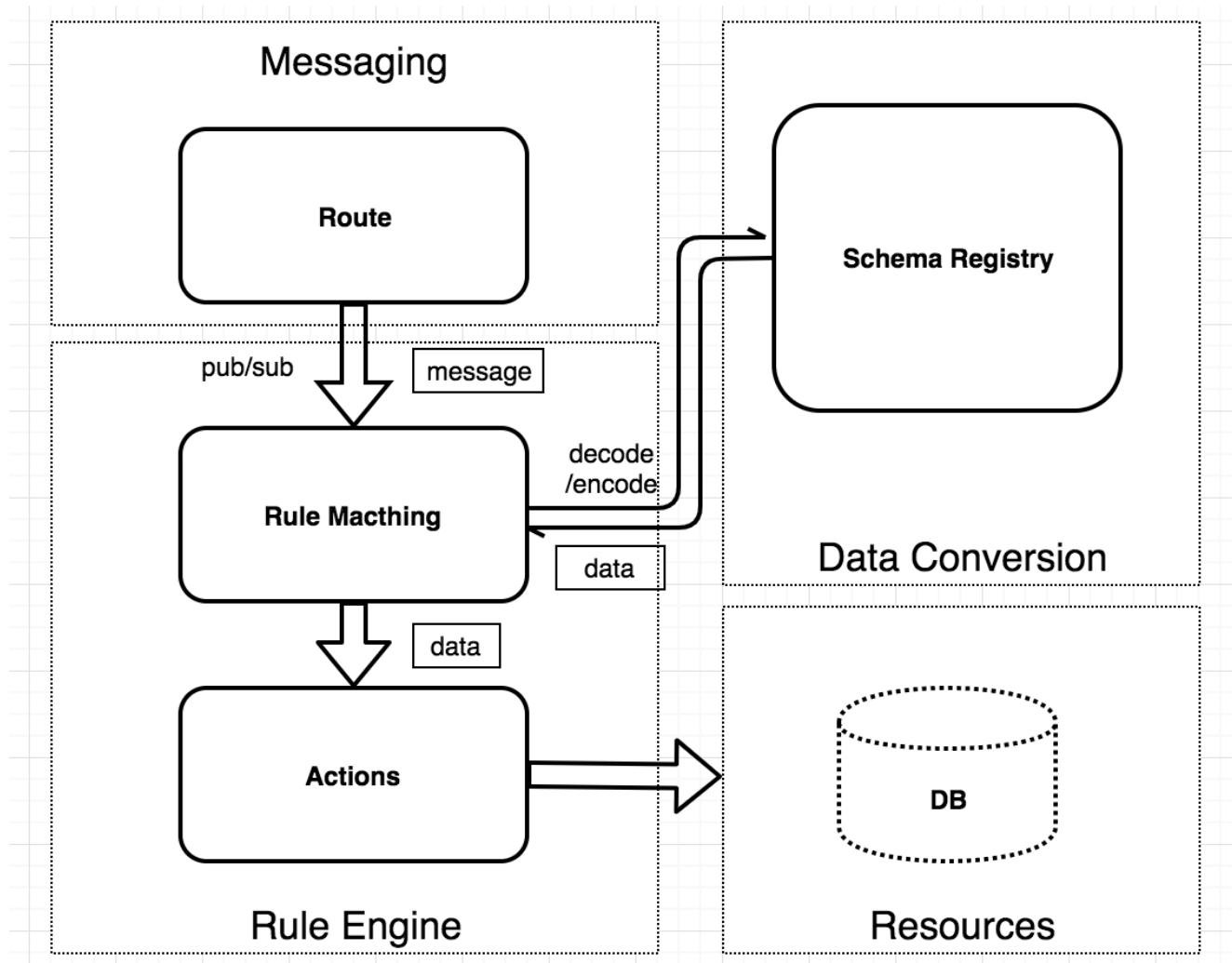
```
1 schema_decode(SchemaName, RawData) -> Data
```

常见的使用案例是，使用规则引擎来调用 **Schema Registry** 提供的编码和解码接口，然后将编码或解码后的数据作为后续动作的输入。

编解码 + 规则引擎

EMQX 的消息处理层面可分为消息路由(**Messaging**)、规则引擎(**Rule Engine**)、数据格式转换(**Data Conversion**) 三个部分。

EMQX 的 **PUB/SUB** 系统将消息路由到指定的主题。规则引擎可以灵活地配置数据的业务规则，按规则匹配消息，然后指定相应动作。数据格式转换发生在规则匹配的过程之前，先将数据转换为可参与规则匹配的 **Map** 格式，然后进行匹配。



规则引擎内部数据格式(Map)

规则引擎内部使用的数据格式为 **Erlang Map**，所以如果原数据内容为二进制或者其他格式，必须使用编解码函数(比如上面提到的 `schema_decode` 和 `json_decode` 函数) 将其转换为 **Map**。

Map 是一个 **Key-Value** 形式的数据结构，形如 `#{key => value}`。例如，`user = #{id => 1, name => "Steve"}` 定义了一个 `id` 为 `1`，`name` 为 `"Steve"` 的 `user` **Map**。

SQL 语句提供了 `".` 操作符嵌套地提取和添加 **Map** 字段。下面是使用 **SQL** 语句对这个 **Map** 操作的示例：

1	<code>SELECT user.id AS my_id</code>
---	--------------------------------------

SQL 语句的筛选结果为 `#{my_id => 1}`。

规则引擎的 **SQL** 语句提供了对 **JSON** 格式字符串的编解码支持，将 **JSON** 字符串和 **Map** 格式相互转换的 **SQL** 函数为 **json_decode()** 和 **json_encode()**:

```
1   SELECT json_decode(payload) AS p FROM "t/#" WHERE p.x = p.y
```

上面这个 **SQL** 语句将会匹配到 **payload** 内容为 **JSON** 字符串: `{"x" = 1, "y" = 1}`，并且 **topic** 为 `t/a` 的 **MQTT** 消息。

`json_decode(payload) as p` 将 **JSON** 字符串解码为下面的 **Map** 数据结构，从而可以在 `WHERE` 子句中使用 `p.x` 和 `p.y` 使用 **Map** 中的字段:

```
1  #{  
2      p => #{  
3          x => 1,  
4          y => 1  
5      }  
6  }
```

注意: `AS` 子句是必须的，将解码之后的数据赋值给某个**Key**，后面才能对其进行后续操作。

编解码举例 - Avro

规则需求

设备发布一个使用 **Avro** 编码的二进制消息，需要通过规则引擎匹配过后，将消息重新发布到与 "name" 字段相关的主题上。主题的格式为 "**avro_user/\${name}**"。

比如，将 "name" 字段为 "Shawn" 的消息重新发布到主题 "**avro_user/Shawn**"。

创建 Schema

在 EMQX 的 [Dashboard](#) 界面，使用下面的参数创建一个 **Avro Schema**:

1. 名称: **avro_user**
2. 编解码类型: **avro**
3. Schema:

```

1  {
2    "type": "record",
3    "fields": [
4      {"name": "name", "type": "string"},
5      {"name": "favorite_number", "type": ["int", "null"]},
6      {"name": "favorite_color", "type": ["string", "null"]}
7    ]
8  }

```

创建规则

使用刚才创建好的 **Schema** 来编写规则 **SQL** 语句:

```

1  SELECT
2    schema_decode('avro_user', payload) as avro_user, payload
3  FROM
4    "t/#"
5  WHERE
6    avro_user.name = 'Shawn'

```

这里的关键点在于 `schema_decode('avro_user', payload)` :

- `schema_decode` 函数将 **payload** 字段的内容按照 '**avro_user**' 这个 **Schema** 来做解码;
- `as avro_user` 将解码后的值保存到变量 "**avro_user**" 里。

然后使用以下参数添加动作:

- 动作类型: 消息重新发布
- 目的主题: **avro_user/\${avro_user.name}**
- 消息内容模板: **\${avro_user}**

这个动作将解码之后的 "user" 以 JSON 的格式发送到 `avro_user/${avro_user.name}` 这个主题。其中 `${avro_user.name}` 是个变量占位符，将在运行时被替换为消息内容中 "name" 字段的值。

设备端代码

规则创建好之后，就可以模拟数据进行测试了。

下面的代码使用 **Python** 语言填充了一个 **User** 消息并编码为二进制数据，然后将其发送到 "**t/1**" 主题。详见 [完整代码](#)。

```

1 def publish_msg(client):
2     datum_w = avro.io.DatumWriter(SCHEMA)
3     buf = io.BytesIO()
4     encoder = avro.io.BinaryEncoder(buf)
5     datum_w.write({"name": "Shawn", "favorite_number": 666, "favorite_color": "red"}, encoder)
6 )
7     message = buf.getvalue()
8     topic = "t/1"
9     print("publish to topic: t/1, payload:", message)
    client.publish(topic, payload=message, qos=0, retain=False)

```

检查规则执行结果

- 在 **Dashboard** 的 [Websocket](#) 工具里，登录一个 **MQTT Client** 并订阅 "`avro_user/#`"。
- 安装 **python** 依赖，并执行设备端代码：

```

1 $ pip3 install protobuf
2 $ pip3 install paho-mqtt
3
4 $ python3 avro_mqtt.py
5 Connected with result code 0
6 publish to topic: t/1, payload: b'\nShawn\x00\xb4\n\x00\x06red'

```

- 检查 **Websocket** 端收到主题为 `avro_user/Shawn` 的消息：

```
1 {"favorite_color": "red", "favorite_number": 666, "name": "Shawn"}
```

编解码举例 - Protobuf

规则需求

设备发布一个使用 **Protobuf** 编码的二进制消息，需要通过规则引擎匹配过后，将消息重新发布到与 “**name**” 字段相关的主题上。主题的格式为 “**person/\${name}**”。

比如，将 “**name**” 字段为 “**Shawn**” 的消息重新发布到主题 “**person/Shawn**”。

创建 Schema

在 EMQX 的 [Dashboard](#) 界面，使用下面的参数创建一个 **Protobuf Schema**:

1. 名称: **protobuf_person**
2. 编解码类型: **protobuf**
3. **Schema**: 下面的 **protobuf schema** 定义了一个 **Person** 消息。

```

1 message Person {
2     required string name = 1;
3     required int32 id = 2;
4     optional string email = 3;
5 }
```

创建规则

使用刚才创建好的 **Schema** 来编写规则 **SQL** 语句:

```

1 SELECT
2     schema_decode('protobuf_person', payload, 'Person') as person, payload
3 FROM
4     "t/#"
5 WHERE
6     person.name = 'Shawn'
```

这里的关键点在于 `schema_decode('protobuf_person', payload, 'Person')` :

- `schema_decode` 函数将 **payload** 字段的内容按照 '**protobuf_person**' 这个 **Schema** 来做解码;
- `as person` 将解码后的值保存到变量 "**person**" 里;
- 最后一个参数 `Person` 指明了 **payload** 中的消息的类型是 **protobuf schema** 里定义的 '**Person**' 类型。

然后使用以下参数添加动作:

- 动作类型: 消息重新发布
- 目的主题: **person/\${person.name}**
- 消息内容模板: `${person}`

这个动作将解码之后的 "**person**" 以 **JSON** 的格式发送到 `person/${person.name}` 这个主题。其

中 `${person.name}` 是个变量占位符，将在运行时被替换为消息内容中 "name" 字段的值。

设备端代码

规则创建好之后，就可以模拟数据进行测试了。

下面的代码使用 **Python** 语言填充了一个 **Person** 消息并编码为二进制数据，然后将其发送到 "t/1" 主题。详见 [完整代码](#)。

```

1 def publish_msg(client):
2     p = person_pb2.Person()
3     p.id = 1
4     p.name = "Shawn"
5     p.email = "liuxy@emqx.io"
6     message = p.SerializeToString()
7     topic = "t/1"
8     print("publish to topic: t/1, payload:", message)
9     client.publish(topic, payload=message, qos=0, retain=False)

```

检查规则执行结果

1. 在 **Dashboard** 的 [Websocket](#) 工具里，登录一个 **MQTT Client** 并订阅 "person/#"。
2. 安装 **python** 依赖，并执行设备端代码：

```

1 $ pip3 install protobuf
2 $ pip3 install paho-mqtt
3
4 $ python3 ./pb2_mqtt.py
5 Connected with result code 0
6 publish to topic: t/1, payload: b'\n\x05Shawn\x10\x01\x1a\rliuxy@emqx.io'
7 t/1 b'\n\x05Shawn\x10\x01\x1a\rliuxy@emqx.io'

```

3. 检查 **Websocket** 端收到主题为 `person/Shawn` 的消息：

```

1 {"email":"liuxy@emqx.io","id":1,"name":"Shawn"}

```

编解码举例 - 自定义 HTTP 编解码

规则需求

设备发布一个任意的消息，验证自部署的编解码服务能正常工作。

创建 Parser HTTP 资源

在 EMQX Dashboard 的 [Resource 创建](#) 界面，使用下面的参数创建一个 **Parser HTTP** 资源：

- **URL:** `http://127.0.0.1:9003/parser`
- **Request Method:** POST

其他保持默认，点击创建之后，得到 **Resource ID**，比如：`resource:606631`

创建 Schema

在 EMQX Dashboard 的 [Schema 创建](#) 界面，使用下面的参数创建一个 **3rd-Party Schema**：

1. 名称: `my_http_parser`
2. 编解码类型: `3rd-party`
3. 第三方类型: `Resources`
4. **Resource:** `resource:606631` (这里选择我们刚才创建的 **Parser HTTP** 资源)
5. 编解码配置: xor

其他配置保持默认。

上面第 5 项编解码配置是个可选项，是个字符串，内容跟编解码服务的业务相关。

创建规则

使用刚才创建好的 **Schema** 来编写规则 **SQL** 语句：

```

1  SELECT
2      schema_encode('my_http_parser', payload) as encoded_data,
3      schema_decode('my_http_parser', encoded_data) as decoded_data
4  FROM
5      "t/#"

```

这个 **SQL** 语句首先对数据做了 **Encode**，然后又做了 **Decode**，目的在于验证编解码过程是否正确：

- `schema_encode` 函数将 `payload` 字段的内容按照 '`my_http_parser`' 这个 **Schema** 来做编码，结果存储到 `encoded_data` 这个变量里；
- `schema_decode` 函数将 `payload` 字段的内容按照 '`my_http_parser`' 这个 **Schema** 来做解码，结果存储到 `decoded_data` 这个变量里；

最终这个 **SQL** 语句的筛选结果是 `encoded_data` 和 `decoded_data` 这两个变量。

然后使用以下参数添加动作：

- 动作类型：检查(调试)

这个检查动作会把 **SQL** 语句筛选的结果打印到 **emqx** 控制台 (**erlang shell**) 里。

如果是使用 **emqx console** 启动的服务，打印会直接显示在控制台里；如果是使用 **emqx start** 启动的服务，打印会输出到日志目录下的 **erlang.log.N** 文件里，这里 "N" 为整数，比如 "**erlang.log.1**", "**erlang.log.2**"。

编解码服务端代码

规则创建好之后，就可以模拟数据进行测试了。所以首先需要编写一个自己的编解码服务。

下面的代码使用 **Python** 语言实现了一个 **HTTP** 编解码服务，为简单起见，这个服务提供两种简单的方式来进行编解码(加解密)，详见 [完整代码](#)：

- 按位异或
- 字符替换

```

1  def xor(data):
2      """
3          >>> xor(xor(b'abc'))
4          b'abc'
5          >>> xor(xor(b'!}~*'))
6          b'!}~*'
7          """
8
9      length = len(data)
10     bdata = bytearray(data)
11     bsecret = bytearray(secret * length)
12     result = bytearray(length)
13     for i in range(length):
14         result[i] = bdata[i] ^ bsecret[i]
15     return bytes(result)
16
17
18  def subst(dtype, data, n):
19      """
20          >>> subst('decode', b'abc', 3)
21          b'def'
22          >>> subst('decode', b'ab~', 1)
23          b'bc!'
24          >>> subst('encode', b'def', 3)
25          b'abc'
26          >>> subst('encode', b'bc!', 1)
27          b'ab~'
28          """
29
30      adata = array.array('B', data)
31      for i in range(len(adata)):
32          if dtype == 'decode':
33              adata[i] = shift(adata[i], n)
34          elif dtype == 'encode':
35              adata[i] = shift(adata[i], -n)
36
37      return bytes(adata)

```

将这个服务运行起来：

```

1 $ pip3 install flask
2 $ python3 http_parser_server.py
3   * Serving Flask app "http_parser_server" (lazy loading)
4   * Environment: production
5     WARNING: This is a development server. Do not use it in a production deployment.
6     Use a production WSGI server instead.
7   * Debug mode: off
8   * Running on http://127.0.0.1:9003/ (Press CTRL+C to quit)

```

检查规则执行结果

由于本示例比较简单，我们直接使用 **MQTT Websocket** 客户端来模拟设备端发一条消息。

1. 在 **Dashboard** 的 **Websocket** 工具里，登录一个 **MQTT Client** 并发布一条消息到 "t/1"，内容为 "hello"。
2. 检查 **emqx** 控制台 (**erlang shell**) 里的打印：

```

1 (emqx@127.0.0.1)1> [inspect]
2   Selected Data: #{decoded_data => <<"hello">>,
3                     encoded_data => <<9,4,13,13,14>>}
4   Envs: #{event => 'message.publish',
5            flags => #{dup => false, retain => false},
6            from => <<"mqttjs_76e5a35b">>,
7            headers =>
8              #{allow_publish => true,
9               peername => {{127,0,0,1},54753},
10              username => <<>>},
11              id => <<0,5,146,30,146,38,123,81,244,66,0,0,62,117,0,1>>,
12              node => 'emqx@127.0.0.1', payload => <<"hello">>, qos => 0,
13              timestamp => {1568,34882,222929},
14              topic => <<"t/1">>}
15   Action Init Params: #{}

```

Select Data 是经过 **SQL** 语句筛选之后的数据，**Envs** 是规则引擎内部可用的环境变量，**Action Init Params** 是动作的初始化参数。这三个数据均为 **Map** 格式。

Selected Data 里面的两个字段 `decoded_data` 和 `encoded_data` 对应 **SELECT** 语句里面的两个 **AS**。因为 `decoded_data` 是编码然后再解码之后的结果，所以它又被还原为了我们发送的内容 "hello"，表明编解码插件工作正常。

编解码举例 - 自定义 gRPC 编解码

规则需求

设备发布一个任意的消息，验证自部署的编解码服务能正常工作。

创建 Parser gRPC 资源

在 EMQX Dashboard 的 [Resource 创建](#) 界面，使用下面的参数创建一个 **Parser gRPC** 资源：

- URL: `http://127.0.0.1:50051`
- Resource ID: `my_grpc_parser_resource`

创建 Schema

在 EMQX Dashboard 的 [Schema 创建](#) 界面，使用下面的参数创建一个 **3rd-Party Schema**：

1. 名称: `my_grpc_parser`
2. 编解码类型: `3rd-party`
3. 第三方类型: `Resources`
4. Resource: `my_grpc_parser_resource` (这里选择我们刚才创建的 **Parser gRPC** 资源)

其他配置保持默认。

创建规则

使用刚才创建好的 **Schema** 来编写规则 **SQL** 语句：

```

1  SELECT
2
3      schema_encode('my_grpc_parser', payload) as encode_resp,
4      schema_decode('my_grpc_parser', encode_resp.result) as decode_resp
5
6  FROM
7
8  "t/#"

```

这个 **SQL** 语句首先对数据做了 **Encode**，然后又做了 **Decode**，目的在于验证编解码过程是否正确：

- `schema_encode` 函数将 `payload` 字段的内容按照 '`my_grpc_parser`' 这个 **Schema** 来做编码，结果存储到 `encode_resp` 这个 **Map** 里；
- `schema_decode` 函数将编码结果内容按照 '`my_grpc_parser`' 这个 **Schema** 来做解码，结果存储到 `decode_resp` 这个变量里；

最终这个 **SQL** 语句的筛选结果是 `encode_resp` 和 `decode_resp` 这两个变量。

然后使用以下参数添加动作：

- 动作类型：检查(调试)

这个检查动作会把 **SQL** 语句筛选的结果打印到 **emqx** 控制台 (**erlang shell**) 里。

如果是使用 **emqx console** 启动的服务，打印会直接显示在控制台里；如果是使用 **emqx start** 启动的服务，打印会输出到日志目录下的 **erlang.log.N** 文件里，这里 "N" 为整数，比如 "**erlang.log.1**"，"**erlang.log.2**"。

编解码服务端代码

规则创建好之后，就可以模拟数据进行测试了。所以首先需要编写一个自己的编解码服务。

下面的代码使用 **Python** 语言实现了一个 **gRPC** 编解码服务。为简单起见，这个服务在编码时，只是对收到的字符串做 **base64_encode**，解码时进行 **base64_decode**。详见 [完整代码](#)：

```

1  class Parser(emqx_schema_registry_pb2_grpc.ParserServicer):
2      def HealthCheck(self, request, context):
3          return request
4      def Parse(self, request, context):
5          if request.type == 1:
6              print("parser got encode request: ", request)
7              encoded_d = base64.b64encode(request.data)
8              return emqx_schema_registry_pb2.ParseResponse(code='SUCCESS', message="ok",
9                  result=encoded_d)
10         elif request.type == 0:
11             print("parser got decode request: ", request)
12             decoded_d = base64.b64decode(request.data)
13             return emqx_schema_registry_pb2.ParseResponse(code='SUCCESS', message="ok",
14                 result=decoded_d)
15
16     def serve():
17         server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
18         emqx_schema_registry_pb2_grpc.add_ParserServicer_to_server(
19             Parser(), server)
20         server.add_insecure_port('[::]:50051')
21         server.start()
22         server.wait_for_termination()
23
24     if __name__ == '__main__':
25         logging.basicConfig()
26         serve()
```

将这个服务运行起来：

```

1  pip3 install grpcio
2  pip3 install grpcio-tools
3  python3 -m grpc_tools.protoc -I./protos --python_out=. --grpc_python_out=. ./protos/emqx_schema
4  registry.proto
5
6  python3 emqx_schema_registry_server.py
```

检查规则执行结果

由于本示例比较简单，我们直接使用 **MQTT Websocket** 客户端来模拟设备端发一条消息。

- 在 **Dashboard** 的 **Websocket** 工具里，登录一个 **MQTT Client** 并发布一条消息到 "t/1"，内容为 "hello"。

- 检查 **emqx** 控制台 (**erlang shell**) 里的打印：

```

1  (emqx@127.0.0.1)1> [inspect]
2      Selected Data: #{{<<"decode_resp">>} =>
3          #{code => 'SUCCESS', message => <<"ok">>,
4              result => <<"hello">>},
5          <<"encode_resp">>} =>
6          #{code => 'SUCCESS', message => <<"ok">>,
7              result => <<"aGVsbG8=">>}}
8      Envs: #{'__bindings__' =>
9          #{'Id' => <<"inspect_1649928007719256000">>,
10             'Params' => #{}, 
11             clientid => <<"mqttjs_4c8818ae">>, event => 'message.publish',
12             flags => #{dup => false, retain => false},
13             headers =>
14                 #{peerhost => <<"127.0.0.1">>, properties => #{},
15                     proto_ver => 4, protocol => mqtt, username => <<>>},
16                 id => <<"0005DC99CDA113B6F4420000CEB0001">>,
17                 metadata => #{rule_id => <<"rule:440083">>},
18                 node => 'emqx@127.0.0.1', payload => <<"hello">>,
19                 peerhost => <<"127.0.0.1">>, pub_props => #{},
20                 publish_received_at => 1649928021545, qos => 0,
21                 timestamp => 1649928021545, topic => <<"t/1">>,
22                 username => <<>>}
23     Action Init Params: #{}

```

Select Data 是经过 **SQL** 语句筛选之后的数据，**Envs** 是规则引擎内部可用的环境变量，**Action Init Params** 是动作的初始化参数。这三个数据均为 **Map** 格式。

Selected Data 里面的两个字段 `decode_resp` 和 `encode_resp` 对应 **SELECT** 语句里面的两个 **AS**。

因为 `decode_resp` 是编码然后再解码之后的结果，所以它又被还原为了我们发送的内容 "hello"，表明编解码插件工作正常。

插件

EMQX 发行包中，包含了大量的官方插件，提供了一些基础的、或各类扩展的功能。

它们依赖于 [emqx](#) 的代码 **API** 或者 [钩子](#) 进行实现其特殊的功能。

然后通过打包编译工具 [emqx-rel](#) 将其与 [emqx](#) 核心项目一起编译并打包至一个可运行的软件包中。

提示

EMQX Enterprise 商业销售时不提供源代码，不支持客户自行开发、编译插件。

插件列表

目前 **EMQX** 发行包提供的插件包括：

插件	配置文件	说明
emqx_dashboard	<code>etc/plugins/emqx_dashboard.conf</code>	Web 控制台插件 (默认加载)
emqx_management	<code>etc/plugins/emqx_management.conf</code>	HTTP API and CLI 管理插件
emqx_auth_mnesia	<code>etc/plugins/emqx_auth_mnesia.conf</code>	Mnesia 认证 / 访问控制
emqx_auth_jwt	<code>etc/plugins/emqx_auth_jwt.conf</code>	JWT 认证 / 访问控制
emqx_auth_ldap	<code>etc/plugins/emqx_auth_ldap.conf</code>	LDAP 认证 / 访问控制
emqx_auth_http	<code>etc/plugins/emqx_auth_http.conf</code>	HTTP API 与 CLI 管理插件
emqx_auth_mongo	<code>etc/plugins/emqx_auth_mongo.conf</code>	MongoDB 认证 / 访问控制
emqx_auth_mysql	<code>etc/plugins/emqx_auth_mysql.conf</code>	MySQL 认证 / 访问控制
emqx_auth_pgsql	<code>etc/plugins/emqx_auth_pgsql.conf</code>	PostgreSQL 认证 / 访问控制
emqx_auth_redis	<code>etc/plugins/emqx_auth_redis.conf</code>	Redis 认证 / 访问控制
emqx_psk_file	<code>etc/plugins/emqx_psk_file.conf</code>	PSK 支持
emqx_web_hook	<code>etc/plugins/emqx_web_hook.conf</code>	Web Hook 插件
emqx_lua_hook	<code>etc/plugins/emqx_lua_hook.conf</code>	Lua Hook 插件
emqx_retainer	<code>etc/plugins/emqx_retainer.conf</code>	Retain 消息存储模块
emqx_rule_engine	<code>etc/plugins/emqx_rule_engine.conf</code>	规则引擎
emqx_bridge_mqtt	<code>etc/plugins/emqx_bridge_mqtt.conf</code>	MQTT 消息桥接插件
emqx_coap	<code>etc/plugins/emqx_coap.conf</code>	CoAP 协议支持
emqx_lwm2m	<code>etc/plugins/emqx_lwm2m.conf</code>	LwM2M 协议支持
emqx_sn	<code>etc/plugins/emqx_sn.conf</code>	MQTT-SN 协议支持
emqx_stomp	<code>etc/plugins/emqx_stomp.conf</code>	Stomp 协议支持
emqx_recon	<code>etc/plugins/emqx_recon.conf</code>	Recon 性能调试
emqx_plugin_template	<code>etc/plugins/emqx_plugin_template.conf</code>	代码热加载插件

启停插件

目前启动插件有以下四种方式：

1. 默认加载
2. 命令行启停插件
3. 使用 **Dashboard** 启停插件
4. 调用管理 API 启停插件

开启默认加载

如需在 **EMQX** 启动时就默认启动某插件，则直接在 `data/loaded_plugins` 添加需要启动的插件名称。

例如，目前 **EMQX** 自动加载的插件有：

```

1 {emqx_management, true}.
2 {emqx_recon, true}.
3 {emqx_retainer, true}.
4 {emqx_dashboard, true}.
5 {emqx_rule_engine, true}.
6 {emqx_bridge_mqtt, false}.

```

命令行启停插件

在 **EMQX** 运行过程中，可通过 [CLI - Load/Unload Plugin](#) 的方式查看、和启停某插件。

使用 **Dashboard** 启停插件

若开启了 **Dashboard** 的插件，可以直接通过访问 <http://localhost:18083/plugins> 中的插件管理页面启停插件。

使用管理 **API** 启停插件

在 **EMQX** 运行过程中，可通过 [管理监控 API - Load Plugin](#) 的方式查看、和启停某插件。

插件开发

创建插件项目

参考 [emqx_plugin_template](#) 插件模版创建新的插件项目。

备注：在 `<plugin name>_app.erl` 文件中必须加上标签 `-emqx_plugin(?MODULE).` 以表明这是一个 **EMQX** 的插件。

创建 认证/访问控制 模块

接入认证示例代码 - `emqx_auth_demo.erl` :

```

1 -module(emqx_auth_demo).
2
3 -export([ init/1
4     , check/2
5     , description/0
6     ]).
7
8 init(Opts) -> {ok, Opts}.
9
10 check(_ClientInfo = #{clientid := ClientId, username := Username, password := Password}, _State) ->
11     io:format("Auth Demo: clientId=~p, username=~p, password=~p~n", [ClientId, Username, Password]),
12     ok.
13
14
15 description() -> "Auth Demo Module".

```

访问控制示例代码 - `emqx_acl_demo.erl` :

```
1 -module(emqx_acl_demo).
2
3 -include_lib("emqx/include/emqx.hrl").
4
5 %% ACL callbacks
6 -export([ init/1
7     , check_acl/5
8     , reload_acl/1
9     , description/0
10    ]).
11
12 init({Opts}) ->
13     {ok, Opts}.
14
15 check_acl({ClientInfo, PubSub, _NoMatchAction, Topic}, _State) ->
16     io:format("ACL Demo: ~p ~p ~p~n", [ClientInfo, PubSub, Topic]),
17     allow.
18
19 reload_acl(_State) ->
20     ok.
21
22 description() -> "ACL Demo Module".
```

挂载认证、访问控制钩子示例代码 - `emqx_plugin_template_app.erl` :

```
1 ok = emqx:hook('client.authenticate', fun emqx_auth_demo:check/2, []),
2 ok = emqx:hook('client.check_acl', fun emqx_acl_demo:check_acl/5, []).
```

挂载钩子

在扩展插件中，可通过挂载 [钩子](#) 来处理客户端上下线、主题订阅、消息收发等事件。

钩子挂载示例代码 - `emqx_plugin_template.erl` :

```

1  load(Env) ->
2      emqx:hook('client.connect',      {?MODULE, on_client_connect, [Env]}),
3      emqx:hook('client.connack',     {?MODULE, on_client_connack, [Env]}),
4      emqx:hook('client.connected',   {?MODULE, on_client_connected, [Env]}),
5      emqx:hook('client.disconnected', {?MODULE, on_client_disconnected, [Env]}),
6      emqx:hook('client.authenticate', {?MODULE, on_client_authenticate, [Env]}),
7      emqx:hook('client.check_acl',    {?MODULE, on_client_check_acl, [Env]}),
8      emqx:hook('client.subscribe',    {?MODULE, on_client_subscribe, [Env]}),
9      emqx:hook('client.unsubscribe',  {?MODULE, on_client_unsubscribe, [Env]}),
10     emqx:hook('session.created',    {?MODULE, on_session_created, [Env]}),
11     emqx:hook('session.subscribed',  {?MODULE, on_session_subscribed, [Env]}),
12     emqx:hook('session.unsubscribed',  {?MODULE, on_session_unsubscribed, [Env]}),
13     emqx:hook('session.resumed',    {?MODULE, on_session_resumed, [Env]}),
14     emqx:hook('session.discarded',  {?MODULE, on_session_discarded, [Env]}),
15     emqx:hook('session.takeovered',  {?MODULE, on_session_takeovered, [Env]}),
16     emqx:hook('session.terminated',  {?MODULE, on_session_terminated, [Env]}),
17     emqx:hook('message.publish',    {?MODULE, on_message_publish, [Env]}),
18     emqx:hook('message.delivered',  {?MODULE, on_message_delivered, [Env]}),
19     emqx:hook('message.acked',      {?MODULE, on_message_acked, [Env]}),
20     emqx:hook('message.dropped',    {?MODULE, on_message_dropped, [Env]}).

```

注册 CLI 命令

处理命令行命令示例代码 - `emqx_cli_demo.erl` :

```

1  -module(emqx_cli_demo).
2
3  -export([cmd/1]).
4
5  cmd(["arg1", "arg2"]) ->
6      emqx_cli:print("ok");
7
8  cmd(_) ->
9      emqx_cli:usage ({{"cmd arg1 arg2", "cmd demo"} }).

```

注册命令行示例代码 - `emqx_plugin_template_app.erl` :

```
1  ok = emqx_ctl:register_command(cmd, {emqx_cli_demo, cmd}, []),
```

插件加载后，使用 `./bin/emqx_ctl` 验证新增的命令行：

```
1  ./bin/emqx_ctl cmd arg1 arg2
```

sh

插件配置文件

插件自带配置文件放置在 `etc/${plugin_name}.conf|config` 。 **EMQX** 支持两种插件配置格式：

1. **Erlang** 原生配置文件格式 - `${plugin_name}.config` :

```

1  [
2      {plugin_name, [
3          {key, value}
4      ]}
5  ].
```

2. **sysctl** 的 `k = v` 通用格式 - `${plugin_name}.conf` :

```

1  plugin_name.key = value
```

注: `k = v` 格式配置需要插件开发者创建 `priv/plugin_name.schema` 映射文件。

编译和发布插件

clone emqx-rel 项目:

```

1  git clone https://github.com/emqx/emqx-rel.git
```

sh

rebar.config 添加依赖:

```

1  {deps,
2      [{plugin_name, {git, "url_of_plugin", {tag, "tag_of_plugin"}}}]
3      , ...
4      ...
5  ]
6 }
```

rebar.config 中 **relx** 段落添加:

```

1  {relx,
2      [...]
3      , ...
4      , {release, {emqx, git_describe}},
5          [
6              {plugin_name, load},
7              ]
8          }
9      ]
10 }
```

数据存储

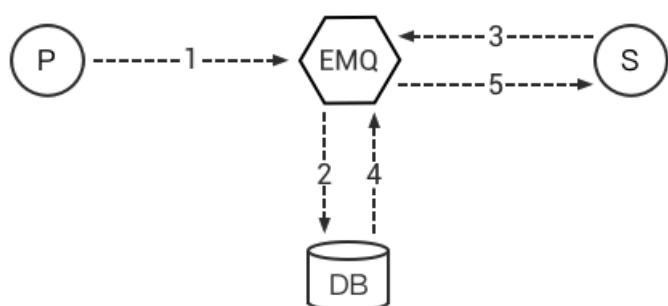
数据存储的主要使用场景包括将客户端上下线状态，订阅主题信息，消息内容，消息抵达后发送消息回执等操作记录到 **Redis**、**MySQL**、**PostgreSQL**、**MongoDB**、**Cassandra** 等各种数据库中。用户也可以通过订阅相关主题的方式来实现类似的功能，但是在企业版中内置了对这些持久化的支持；相比于前者，后者的执行效率更高，也能大大降低开发者的工作量。

提示

数据存储是 **EMQX Enterprise** 专属功能。

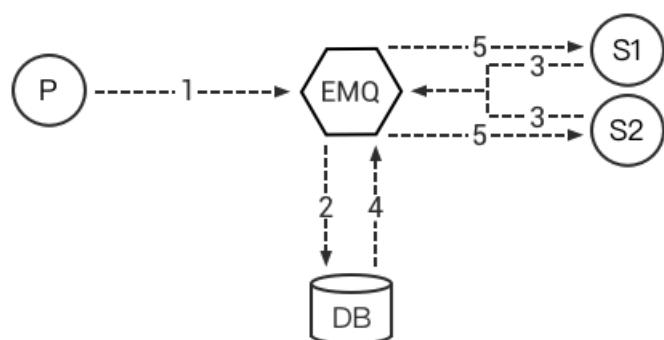
数据存储设计

一对消息存储



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；

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。

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。

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。

存储插件列表

EMQX 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

配置步骤

EMQX 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。

Redis 数据存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 Redis](#)规则引擎中创建 保存数据到 Redis

配置文件: `emqx_backend_redis.conf`

配置 Redis 服务器

支持配置多台 Redis 服务器连接池:

```
1 ## Redis 服务集群类型: single | sentinel | cluster
2 backend.redis.pool1.type = single
3
4 ## Redis 服务器地址列表
5 backend.redis.pool1.server = 127.0.0.1:6379
6
7 ## Redis sentinel 模式下的 sentinel 名称
8 ## backend.redis.pool1.sentinel = mymaster
9
10 ## Redis 连接池大小
11 backend.redis.pool1.pool_size = 8
12
13 ## Redis 数据库名称
14 backend.redis.pool1.database = 0
15
16 ## Redis 密码
17 ## backend.redis.pool1.password =
18
19 ## 订阅的 Redis channel 名称
20 backend.redis.pool1.channel = mqtt_channel
```

配置 Redis 存储规则

```
1 backend.redis.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "sh
2 pool": "pool1"}
3 backend.redis.hook.session.created.1 = {"action": {"function": "on_subscribe_lookup"}, "p
4 pool": "pool1"}
5 backend.redis.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}}
6 , "pool": "pool1"}
7 backend.redis.hook.session.subscribed.1 = {"topic": "queue/#", "action": {"function": "on_m
8 essage_fetch_for_queue"}, "pool": "pool1"}
9 backend.redis.hook.session.subscribed.2 = {"topic": "pubsub/#", "action": {"function": "on_
10 message_fetch_for_pubsub"}, "pool": "pool1"}
11 backend.redis.hook.session.subscribed.3 = {"action": {"function": "on_retain_lookup"}, "poo
12 l": "pool1"}
13 backend.redis.hook.session.unsubscribed.1= {"topic": "#", "action": {"commands": ["DEL mqtt:
14 acked:${clientid}:${topic}"]}, "pool": "pool1"}
15 backend.redis.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_
16 _publish"}, "expired_time" : 3600, "pool": "pool1"}
backend.redis.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message_
_retain"}, "expired_time" : 3600, "pool": "pool1"}
backend.redis.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_
delete"}, "pool": "pool1"}
backend.redis.hook.message.acked.1 = {"topic": "queue/#", "action": {"function": "on_m
essage_acked_for_queue"}, "pool": "pool1"}
backend.redis.hook.message.acked.2 = {"topic": "pubsub/#", "action": {"function": "on_
message_acked_for_pubsub"}, "pool": "pool1"}

## backend.redis.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_
fetch_for_keep_latest"}, "pool": "pool1"}
## backend.redis.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_
store_keep_latest"}, "expired_time" : 3600, "pool": "pool1"}
## backend.redis.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message_
acked_for_keep_latest"}, "pool": "pool1"}
```

Redis 存储规则说明

hook	topic	action/function	说明
client.connected		on_client_connected	存储客户端在线状态
session.created		on_subscribe_lookup	订阅主题
client.disconnected		on_client_disconnected	存储客户端离线状态
session.subscribed	queue/#	on_message_fetch_for_queue	获取一对一离线消息
session.subscribed	pubsub/#	on_message_fetch_for_pubsub	获取一对多离线消息
session.subscribed	#	on_retain_lookup	获取 retain 消息
session.unsubscribed	#		删除 acked 消息
message.publish	#	on_message_publish	存储发布消息
message.publish	#	on_message_retain	存储 retain 消息
message.publish	#	on_retain_delete	删除 retain 消息
message.acked	queue/#	on_message_acked_for_queue	一对一消息 ACK 处理
message.acked	pubsub/#	on_message_acked_for_pubsub	一对多消息 ACK 处理

Redis 命令行参数说明

hook	可用参数	示例(每个字段分隔, 必须是一个空格)
client.connected	clientid	SET conn:\${clientid} \${clientid}
client.disconnected	clientid	SET disconn:\${clientid} \${clientid}
session.subscribed	clientid, topic, qos	HSET sub:\${clientid} \${topic} \${qos}
session.unsubscribed	clientid, topic	SET unsub:\${clientid} \${topic}
message.publish	message, msgid, topic, payload, qos, clientid	RPUSH pub:\${topic} \${msgid}
message.acked	msgid, topic, clientid	HSET ack:\${clientid} \${topic} \${msgid}
message.deliver	msgid, topic, clientid	HSET deliver:\${clientid} \${topic} \${msgid}

Redis 命令行配置 Action

Redis 存储支持用户采用 **Redis Commands** 语句配置 Action, 例如:

```
1 ## 在客户端连接到 EMQX 服务器后, 执行一条 redis
2 backend.redis.hook.client.connected.3 = {"action": {"commands": ["SET conn:${clientid} ${clientid}"]}, "pool": "pool1"}
```

Redis 设备在线状态 Hash

mqtt:client Hash 存储设备在线状态:

```

1 hmset
2   key = mqtt:client:${clientId}
3   value = {state:int, online_at:timestamp, offline_at:timestamp}
4
5 hset
6   key = mqtt:node:${node}
7   field = ${clientId}
8   value = ${ts}
```

查询设备在线状态:

```

1 HGETALL "mqtt:client:${clientId}"
```

例如 **ClientId** 为 **test** 客户端上线:

```

1 HGETALL mqtt:client:test
2 1) "state"
3 2) "1"
4 3) "online_at"
5 4) "1481685802"
6 5) "offline_at"
7 6) "undefined"
```

例如 **ClientId** 为 **test** 客户端下线:

```

1 HGETALL mqtt:client:test
2 1) "state"
3 2) "0"
4 3) "online_at"
5 4) "1481685802"
6 5) "offline_at"
7 6) "1481685924"
```

Redis 保留消息 Hash

mqtt:retain Hash 存储 **Retain** 消息:

```

1 hmset
2   key = mqtt:retain:${topic}
3   value = {id: string, from: string, qos: int, topic: string, retain: int, payload: string, ts: timestamp}
```

查询 **retain** 消息:

```
1 HGETALL "mqtt:retain:${topic}"
```

sh

例如查看 **topic** 为 **topic** 的 **retain** 消息:

```
1      HGETALL mqtt:retain:topic
2          1) "id"
3
4      >      -      2) "6P9NLcJ65VXBbC22sYb4"
5      >      3) "from"
6      >      -      4) "test"
7      >      5) "qos"
8      >      6) "1"
9      >      7) "topic"
10     >      8) "topic"
11     >      9) "retain"
12     >      - 10\)"true"
13     >      11) "payload"
14     >      12) "Hello world\!"
15     >      13) "ts"
16     >      14) "1481690659"
```

sh

Redis 消息存储 Hash

mqtt:msg Hash 存储 MQTT 消息:

```
1 hmset
2 key = mqtt:msg:${msgid}
3 value = {id: string, from: string, qos: int, topic: string, retain: int, payload: string, ts: timestamp}
4
5 zadd
6 key = mqtt:msg:${topic}
7 field = 1
8 value = ${msgid}
```

sh

Redis 消息确认 SET

mqtt:acked SET 存储客户端消息确认:

```
1 set
2 key = mqtt:acked:${clientid}:${topic}
3 value = ${msgid}
```

sh

Redis 订阅存储 Hash

mqtt:sub Hash 存储订阅关系:

```

1 hset
2 key = mqtt:sub:${clientId}
3 field = ${topic}
4 value = ${qos}

```

sh

某个客户端订阅主题:

```
1 HSET mqtt:sub:${clientId} ${topic} ${qos}
```

sh

例如为 **ClientId** 为 **test** 的客户端订阅主题 **topic1, topic2** :

```

1 HSET "mqtt:sub:test" "topic1" 1
2 HSET "mqtt:sub:test" "topic2" 2

```

sh

查询 **ClientId** 为 **test** 的客户端已订阅主题:

```

1 HGETALL mqtt:sub:test
2 1) "topic1"
3 2) "1"
4 3) "topic2"
5 4) "2"

```

sh

Redis SUB/UNSUB 事件发布

设备需要订阅/取消订阅主题时，业务服务器向 **Redis** 发布事件消息:

```

1 PUBLISH
2 channel = "mqtt_channel"
3 message = {type: string , topic: string, clientId: string, qos: int}
4 \*type: [subscribe/unsubscribe]

```

sh

例如 **ClientId** 为 **test** 客户端订阅主题 **topic0**:

```
1 PUBLISH "mqtt_channel" "{\"type\": \"subscribe\", \"topic\": \"topic0\", \"clientId\": \"test\", \"qos\": \"0\"}"
```

sh

例如 **ClientId** 为 **test** 客户端取消订阅主题:

```
1 PUBLISH "mqtt_channel" "{\"type\": \"unsubscribe\", \"topic\": \"test_topic0\", \"clientId\": \"test\"}"
```

sh

提示

Redis Cluster 无法使用 **Redis PUB/SUB** 功能。

启用 Redis 数据存储插件

```
1 ./bin/emqx_ctl plugins load emqx_backend_redis      sh
```

MySQL 数据存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 MySQL](#)规则引擎中创建 保存数据到 MySQL

配置文件: **emqx_backend_mysql.conf**

配置 MySQL 服务器

支持配置多台 MySQL 服务器连接池:

```
1 ## Mysql 服务器地址
2 backend.mysql.pool1.server = 127.0.0.1:3306
3
4 ## Mysql 连接池大小
5 backend.mysql.pool1.pool_size = 8
6
7 ## Mysql 用户名
8 backend.mysql.pool1.user = root
9
10 ## Mysql 密码
11 backend.mysql.pool1.password = public
12
13 ## Mysql 数据库名称
14 backend.mysql.pool1.database = mqtt
```

配置 MySQL 存储规则

```

1 backend.mysql.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "sh
2 pool": "pool1"}
3 backend.mysql.hook.session.created.1 = {"action": {"function": "on_subscribe_lookup"}, "p
4 pool": "pool1"}
5 backend.mysql.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}}
6 , "pool": "pool1"}
7 backend.mysql.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message
8 _fetch"}, "pool": "pool1"}
9 backend.mysql.hook.session.subscribed.2 = {"topic": "#", "action": {"function": "on_retain_
10 lookup"}, "pool": "pool1"}
11 backend.mysql.hook.session.unsubscribed.1= {"topic": "#", "action": {"sql": ["delete from mq
12 tt_acked where clientid = ${clientid} and topic = ${topic}"]}, "pool": "pool1"}
13 backend.mysql.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message
14 _publish"}, "pool": "pool1"}
15 backend.mysql.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message
16 _retain"}, "pool": "pool1"}
17 backend.mysql.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_
18 delete"}, "pool": "pool1"}
19 backend.mysql.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message
20 _acked"}, "pool": "pool1"}

## 获取离线消息
### "offline_opts": 获取离线消息的配置
#### - max_returned_count: 单次拉去的最大离线消息数目
#### - time_range: 仅拉去在当前时间范围的消息
## backend.mysql.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message
fetch"}, "offline_opts": {"max_returned_count": 500, "time_range": "2h"}, "pool": "pool1"}

## 如果需要存储 Qos0 消息, 可开启以下配置
## 警告: 当开启以下配置时, 需关闭 'on_message_fetch', 否则 qos1, qos2 消息会被存储俩次
## backend.mysql.hook.message.publish.4 = {"topic": "#", "action": {"function": "on_message
store"}, "pool": "pool1"}

```

MySQL 存储规则说明

hook	topic	action	说明
client.connected		on_client_connected	存储客户端在线状态
session.created		on_subscribe_lookup	订阅主题
client.disconnected		on_client_disconnected	存储客户端离线状态
session.subscribed	#	on_message_fetch	获取离线消息
session.subscribed	#	on_retain_lookup	获取 retain 消息
message.publish	#	on_message_publish	存储发布消息
message.publish	#	on_message_retain	存储 retain 消息
message.publish	#	on_retain_delete	删除 retain 消息
message.acked	#	on_message_acked	消息 ACK 处理

SQL 语句参数说明

hook	可用参数	示例(sql语句中\${name} 表示可获取的参数)
client.connected	clientid	insert into conn(clientid) values(\${clientid})
client.disconnected	clientid	insert into disconn(clientid) values(\${clientid})
session.subscribed	clientid, topic, qos	insert into sub(topic, qos) values(\${topic}, \${qos})
session.unsubscribed	clientid, topic	delete from sub where topic = \${topic}
message.publish	msgid, topic, payload, qos, clientid	insert into msg(msgid, topic) values(\${msgid}, \${topic})
message.acked	msgid, topic, clientid	insert into ack(msgid, topic) values(\${msgid}, \${topic})
message.deliver	msgid, topic, clientid	insert into deliver(msgid, topic) values(\${msgid}, \${topic})

SQL 语句配置 Action

MySQL 存储支持用户采用 SQL 语句配置 Action:

```
1 ## 在客户端连接到 EMQX 服务器后, 执行一条 sql 语句(支持多条 sql 语句)
2 backend.mysql.hook.client.connected.3 = {"action": {"sql": ["insert into conn(clientid) value s(${clientid})"]}, "pool": "pool1"}
```

创建 MySQL 数据库表

```
1 create database mqtt;
```

导入 MySQL 库表结构

```
1 mysql -u root -p mqtt < etc/sql/emqx_backend_mysql.sql
```

提示

数据库名称可自定义

MySQL 设备在线状态表

mqtt_client 存储设备在线状态:

```

1  DROP TABLE IF EXISTS `mqtt_client`;
2  CREATE TABLE `mqtt_client` (
3      `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4      `clientid` varchar(64) DEFAULT NULL,
5      `state` varchar(3) DEFAULT NULL,
6      `node` varchar(64) DEFAULT NULL,
7      `online_at` datetime DEFAULT NULL,
8      `offline_at` datetime DEFAULT NULL,
9      `created` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
10     PRIMARY KEY (`id`),
11     KEY `mqtt_client_idx` (`clientid`),
12     UNIQUE KEY `mqtt_client_key` (`clientid`),
13     INDEX topic_index(`id`, `clientid`)
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;

```

查询设备在线状态:

```
1  select * from mqtt_client where clientid = ${clientid};
```

例如 ClientId 为 test 客户端上线:

```

1  select * from mqtt_client where clientid = "test";
2
3  +----+-----+-----+-----+-----+-----+
4  | id | clientid | state | node           | online_at        | offline_at       | creat
5  ed   |          | 1     | emqx@127.0.0.1 | 2016-11-15 09:40:40 | NULL             | 20
6
7  +----+-----+-----+-----+-----+-----+
8  | 1 | test    | 1     | emqx@127.0.0.1 | 2016-11-15 09:40:40 | NULL             | 20
9  16-12-24 09:40:22 |
10 +----+-----+-----+-----+-----+-----+
11 -----+
12 1 rows in set (0.00 sec)

```

例如 ClientId 为 test 客户端下线:

```

1  select * from mqtt_client where clientid = "test";
2
3  +----+-----+-----+-----+-----+-----+
4  | id | clientid | state | node           | online_at        | offline_at       | creat
5  ed   |          | 0     | emqx@127.0.0.1 | 2016-11-15 09:40:40 | 2016-11-15 09:46:10 | 2016-
6
7  +----+-----+-----+-----+-----+-----+
8  | 1 | test    | 0     | emqx@127.0.0.1 | 2016-11-15 09:40:40 | 2016-11-15 09:46:10 | 2016-
9  12-24 09:40:22 |
10 +----+-----+-----+-----+-----+-----+
11 -----+
12 1 rows in set (0.00 sec)

```

MySQL 主题订阅表

mqtt_sub 存储设备的主题订阅关系:

```

1  DROP TABLE IF EXISTS `mqtt_sub`;
2  CREATE TABLE `mqtt_sub` (
3      `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4      `clientid` varchar(64) DEFAULT NULL,
5      `topic` varchar(180) DEFAULT NULL,
6      `qos` tinyint(1) DEFAULT NULL,
7      `created` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
8      PRIMARY KEY (`id`),
9      KEY `mqtt_sub_idx` (`clientid`,`topic`,`qos`),
10     UNIQUE KEY `mqtt_sub_key` (`clientid`,`topic`),
11     INDEX topic_index(`id`, `topic`)
12 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;

```

例如 ClientId 为 test 客户端订阅主题 test_topic1 test_topic2:

```

1  insert into mqtt_sub(clientid, topic, qos) values("test", "test_topic1", 1);
2  insert into mqtt_sub(clientid, topic, qos) values("test", "test_topic2", 2);

```

某个客户端订阅主题:

```
1  select * from mqtt_sub where clientid = ${clientid};
```

sh

查询 ClientId 为 test 的客户端已订阅主题:

```

1  select * from mqtt_sub where clientid = "test";
2
3  +----+-----+-----+-----+
4  | id | clientId | topic | qos | created |
5  +----+-----+-----+-----+
6  | 1 | test | test_topic1 | 1 | 2016-12-24 17:09:05 |
7  | 2 | test | test_topic2 | 2 | 2016-12-24 17:12:51 |
8  +----+-----+-----+-----+
9  2 rows in set (0.00 sec)

```

sh

MySQL 消息存储表

mqtt_msg 存储 MQTT 消息:

```

1  DROP TABLE IF EXISTS `mqtt_msg`;
2  CREATE TABLE `mqtt_msg` (
3      `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4      `msgid` varchar(64) DEFAULT NULL,
5      `topic` varchar(180) NOT NULL,
6      `sender` varchar(64) DEFAULT NULL,
7      `node` varchar(64) DEFAULT NULL,
8      `qos` tinyint(1) NOT NULL DEFAULT '0',
9      `retain` tinyint(1) DEFAULT NULL,
10     `payload` blob,
11     `arrived` datetime NOT NULL,
12     PRIMARY KEY (`id`),
13     INDEX topic_index(`id`, `topic`)
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;

```

查询某个客户端发布的消息:

```

1  select * from mqtt_msg where sender = ${clientid};

```

查询 **ClientId** 为 **test** 的客户端发布的消息:

```

1  select * from mqtt_msg where sender = "test";
2
3  +-----+-----+-----+-----+-----+-----+-----+
4  | id | msgid           | topic    | sender | node   | qos   | retain |
5  | 1  | 53F98F80F66017005000004A60003 | hello   | test   | NULL  | 1    | 0      |
6  | arrived |          |          |          |        |       |         |
7  +-----+-----+-----+-----+-----+-----+-----+
8  | 2  | 53F98F9FE42AD7005000004A60004 | world  | test   | NULL  | 1    | 0      |
9  | 016-12-24 17:25:45 |          |          |          |        |       |         |
+-----+-----+-----+-----+-----+-----+-----+
-----+
2 rows in set (0.00 sec)

```

MySQL 保留消息表

mqtt_retain 存储 **retain** 消息:

```

1  DROP TABLE IF EXISTS `mqtt_retain`;
2  CREATE TABLE `mqtt_retain` (
3      `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4      `topic` varchar(180) DEFAULT NULL,
5      `msgid` varchar(64) DEFAULT NULL,
6      `sender` varchar(64) DEFAULT NULL,
7      `node` varchar(64) DEFAULT NULL,
8      `qos` tinyint(1) DEFAULT NULL,
9      `payload` blob,
10     `arrived` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
11     PRIMARY KEY (`id`),
12     UNIQUE KEY `mqtt_retain_key` (`topic`),
13     INDEX topic_index(`id`, `topic`)
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;

```

查询 **retain** 消息:

```
1  select * from mqtt_retain where topic = ${topic};
```

查询 **topic** 为 **retain** 的 **retain** 消息:

```

1  select * from mqtt_retain where topic = "retain";
2
3  +----+-----+-----+-----+-----+-----+-----+
4  | id | topic | msgid | sender | node | qos | payload | arrived
5  |    |       |        |        |      |   |        |        |
6  +----+-----+-----+-----+-----+-----+-----+
7  | 1 | retain | 53F33F7E4741E7007000004B70001 | test | NULL | 1 | www | 2016-12-
8  | 24 | 16:55:18 |           |
9  +----+-----+-----+-----+-----+-----+-----+
10
11
> 1 rows in set (0.00 sec)

```

MySQL 消息确认表

mqtt_acked 存储客户端消息确认:

```
1  DROP TABLE IF EXISTS `mqtt_acked`;
2  CREATE TABLE `mqtt_acked` (
3      `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
4      `clientid` varchar(64) DEFAULT NULL,
5      `topic` varchar(180) DEFAULT NULL,
6      `mid` int(11) unsigned DEFAULT NULL,
7      `created` timestamp NULL DEFAULT NULL,
8      PRIMARY KEY (`id`),
9      UNIQUE KEY `mqtt_acked_key` (`clientid`,`topic`),
10     INDEX topic_index(`id`, `topic`)
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

启用 MySQL 数据存储插件

```
1 ./bin/emqx_ctl plugins load emqx_backend_mysql
```

sh

PostgreSQL 数据存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 PostgreSQL](#)规则引擎中创建 保存数据到 PostgreSQL

配置文件: `emqx_backend_pgsql.conf`

配置 PostgreSQL 服务器

提示

支持 **PostgreSQL 13** 及以下版本

支持配置多台PostgreSQL服务器连接池:

```
1 ## Pgsql 服务器地址
2 backend.pgsql.pool1.server = 127.0.0.1:5432
3
4 ## Pgsql 连接池大小
5 backend.pgsql.pool1.pool_size = 8
6
7 ## Pgsql 用户名
8 backend.pgsql.pool1.username = root
9
10 ## Pgsql 密码
11 backend.pgsql.pool1.password = public
12
13 ## Pgsql 数据库名称
14 backend.pgsql.pool1.database = mqtt
15
16 ## Pgsql Ssl
17 backend.pgsql.pool1.ssl = false
```

sh

配置 PostgreSQL 存储规则

```

1 backendpgsql.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "sh
2 pool": "pool1"}
3 backendpgsql.hook.session.created.1 = {"action": {"function": "on_subscribe_lookup"}, "p
4 pool": "pool1"}
5 backendpgsql.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}}
6 , "pool": "pool1"}
7 backendpgsql.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message
8 _fetch"}, "pool": "pool1"}
9 backendpgsql.hook.session.subscribed.2 = {"topic": "#", "action": {"function": "on_retain_
10 lookup"}, "pool": "pool1"}
11 backendpgsql.hook.session.unsubscribed.1= {"topic": "#", "action": {"sql": ["delete from mq
12 tt_acked where clientid = ${clientid} and topic = ${topic}"]}, "pool": "pool1"}
13 backendpgsql.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message
14 _publish"}, "pool": "pool1"}
15 backendpgsql.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message
16 _retain"}, "pool": "pool1"}
17 backendpgsql.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_
18 delete"}, "pool": "pool1"}
19 backendpgsql.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message
20 _acked"}, "pool": "pool1"}

## 获取离线消息
### "offline_opts": 获取离线消息的配置
#### - max_returned_count: 单次拉去的最大离线消息数目
#### - time_range: 仅拉去在当前时间范围的消息
## backendpgsql.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message
fetch"}, "offline_opts": {"max_returned_count": 500, "time_range": "2h"}, "pool": "pool1"}

## 如果需要存储 Qos0 消息, 可开启以下配置
## 警告: 当开启以下配置时, 需关闭 'on_message_fetch', 否则 qos1, qos2 消息会被存储俩次
## backendpgsql.hook.message.publish.4 = {"topic": "#", "action": {"function": "on_message
store"}, "pool": "pool1"}

```

PostgreSQL 存储规则说明

hook	topic	action	说明
client.connected		on_client_connected	存储客户端在线状态
session.created		on_subscribe_lookup	订阅主题
client.disconnected		on_client_disconnected	存储客户端离线状态
session.subscribed	#	on_message_fetch	获取离线消息
session.subscribed	#	on_retain_lookup	获取 retain 消息
message.publish	#	on_message_publish	存储发布消息
message.publish	#	on_message_retain	存储 retain 消息
message.publish	#	on_retain_delete	删除 retain 消息
message.acked	#	on_message_acked	消息 ACK 处理

SQL 语句参数说明

hook	可用参数	示例(sql语句中\${name} 表示可获取的参数)
client.connected	clientid	insert into conn(clientid) values(\${clientid})
client.disconnected	clientid	insert into disconn(clientid) values(\${clientid})
session.subscribed	clientid, topic, qos	insert into sub(topic, qos) values(\${topic}, \${qos})
session.unsubscribed	clientid, topic	delete from sub where topic = \${topic}
message.publish	msgid, topic, payload, qos, clientid	insert into msg(msgid, topic) values(\${msgid}, \${topic})
message.acked	msgid, topic, clientid	insert into ack(msgid, topic) values(\${msgid}, \${topic})
message.deliver	msgid, topic, clientid	insert into deliver(msgid, topic) values(\${msgid}, \${topic})

SQL 语句配置 Action

PostgreSQL 存储支持用户采用SQL语句配置 Action, 例如:

```
1 ## 在客户端连接到 EMQX 服务器后, 执行一条 sql 语句(支持多条sql语句)
2 backendpgsql.hook.client.connected.3 = {"action": {"sql": ["insert into conn(clientid) value
s(${clientid})"]}, "pool": "pool1"}
```

创建 PostgreSQL 数据库

```
1 createdb mqtt -E UTF8 -e
```

导入 PostgreSQL 库表结构

```
1 \i etc/sql/emqx_backend_pgsql.sql
```

PostgreSQL 设备在线状态表

mqtt_client 存储设备在线状态:

```

1 CREATE TABLE mqtt_client(
2     id SERIAL8 primary key,
3     clientid character varying(64),
4     state integer,
5     node character varying(64),
6     online_at timestamp ,
7     offline_at timestamp,
8     created timestamp without time zone,
9     UNIQUE (clientid)
10 );

```

sh

查询设备在线状态:

```

1 select * from mqtt_client where clientid = ${clientid};

```

例如 ClientId 为 test 客户端上线:

```

1 select * from mqtt_client where clientid = 'test';
2
3     id | clientid | state | node           | online_at          | offline_at         | creat
4 ed
5
6
-----+-----+-----+-----+-----+-----+-----+
7
8     1 | test      | 1      | emqx@127.0.0.1 | 2016-11-15 09:40:40 | NULL              | 2016-
9 12-24 09:40:22
10 (1 rows)

```

sh

例如 ClientId 为 test 客户端下线:

```

1 select * from mqtt_client where clientid = 'test';
2
3     id | clientid | state | nod           | online_at          | offline_at         | creat
4 ed
5
6
-----+-----+-----+-----+-----+-----+-----+
7
8     1 | test      | 0      | emqx@127.0.0.1 | 2016-11-15 09:40:40 | 2016-11-15 09:46:10 | 2016-1
9 2-24 09:40:22
10 (1 rows)

```

sh

PostgreSQL 代理订阅表

mqtt_sub 存储订阅关系:

```

1 CREATE TABLE mqtt_sub(
2     id SERIAL8 primary key,
3     clientid character varying(64),
4     topic character varying(255),
5     qos integer,
6     created timestamp without time zone,
7     UNIQUE (clientid, topic)
8 );

```

例如 ClientId 为 **test** 客户端订阅主题 **test_topic1 test_topic2**:

```

1 insert into mqtt_sub(clientid, topic, qos) values('test', 'test_topic1', 1);
2 insert into mqtt_sub(clientid, topic, qos) values('test', 'test_topic2', 2);

```

某个客户端订阅主题:

```

1 select * from mqtt_sub where clientid = ${clientid};

```

sh

查询 ClientId 为 **test** 的客户端已订阅主题:

```

1 select * from mqtt_sub where clientid = 'test';
2
3   id | clientId      | topic        | qos  | created
4   ---+---+---+---+---+
5   1 | test          | test_topic1 | 1   | 2016-12-24 17:09:05
6   2 | test          | test_topic2 | 2   | 2016-12-24 17:12:51
7 (2 rows)

```

sh

PostgreSQL 消息存储表

mqtt_msg 存储MQTT消息:

```

1 CREATE TABLE mqtt_msg (
2     id SERIAL8 primary key,
3     msgid character varying(64),
4     sender character varying(64),
5     topic character varying(255),
6     qos integer,
7     retain integer,
8     payload text,
9     arrived timestamp without time zone
10 );

```

查询某个客户端发布的消息:

```

1 select * from mqtt_msg where sender = ${clientid};

```

查询 ClientId 为 **test** 的客户端发布的消息:

```

1 select * from mqtt_msg where sender = 'test';
2
3     id | msgid                  | topic      | sender | node | qos | retain | payload | a
4 arrived
5
6
7     1 | 53F98F80F66017005000004A60003 | hello     | test   | NULL | 1   | 0       | hello   | 2
016-12-24 17:25:12
8     2 | 53F98F9FE42AD7005000004A60004 | world     | test   | NULL | 1   | 0       | world   | 2
016-12-24 17:25:45
(2 rows)

```

PostgreSQL 保留消息表

mqtt_retain 存储 Retain 消息:

```

1 CREATE TABLE mqtt_retain(
2     id SERIAL8 primary key,
3     topic character varying(255),
4     msgid character varying(64),
5     sender character varying(64),
6     qos integer,
7     payload text,
8     arrived timestamp without time zone,
9     UNIQUE (topic)
10 );

```

查询 **retain** 消息:

```

1 select * from mqtt_retain where topic = ${topic};

```

查询 **topic** 为 **retain** 的 **retain** 消息:

```

1 select * from mqtt_retain where topic = 'retain';
2
3     id | topic      | msgid                  | sender | node | qos | payload | arrived
4
5
6     1 | retain    | 53F33F7E4741E7007000004B70001 | test   | NULL | 1   | www     | 2016-12-2
4 16:55:18
(1 rows)

```

PostgreSQL 消息确认表

mqtt_acked 存储客户端消息确认:

```
1 CREATE TABLE mqtt_acked (
2     id SERIAL8 primary key,
3     clientid character varying(64),
4     topic character varying(64),
5     mid integer,
6     created timestamp without time zone,
7     UNIQUE (clientid, topic)
8 );
```

启用 PostgreSQL 数据存储插件

```
1 ./bin/emqx_ctl plugins load emqx_backend_pgsql
```

sh

MongoDB 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 MongoDB](#)规则引擎中创建保存数据到 **MongoDB**

配置 MongoDB 消息存储

配置文件: **emqx_backend_mongo.conf**

配置 MongoDB 服务器

支持配置多台 **MongoDB** 服务器连接池:

```

1 ## MongoDB 部署类型: single | unknown | sharded | rs
2 backend.mongo.pool1.type = single
3
4 ## 是否启用 SRV 和 TXT 记录解析
5 backend.mongo.pool1.srv_record = false
6
7 ## 如果您的 MongoDB 以副本集方式部署, 则需要指定相应的副本集名称
8 ##
9 ## 如果启用了 srv_record, 即 backend.mongo.<Pool>.srv_record 设置为 true,
10 ## 且您的 MongoDB 服务器域名添加了包含 replicaSet 选项的 DNS TXT 记录,
11 ## 那么可以忽略此配置项
12 ## backend.mongo.pool1.rs_set_name = testrs
13
14 ## MongoDB 服务器地址列表
15 ##
16 ## 如果你的 URI 具有以下格式:
17 ## mongodb://[username:password@]host1[:port1][,...hostN[:portN]][/[defaultauthdb][?options]]
18
19 ## 请将 backend.mongo.<Pool>.server 配置为 host1[:port1][,...hostN[:portN]]
20 ##
21 ## 如果你的 URI 具有以下格式:
22 ## mongodb+srv://server.example.com
23 ## 请将 backend.mongo.<Pool>.server 配置为 server.example.com, 并将 srv_record
24 ## 设置为 true, EMQX 将自动查询 SRV 和 TXT 记录以获取服务列表和 replicaSet 等选项
25 ##
26 ## 现已支持 IPv6 和域名
27 backend.mongo.pool1.server = 127.0.0.1:27017
28
29 ## MongoDB 连接池大小
30 backend.mongo.pool1.c_pool_size = 8
31
32 ## 连接的数据库名称
33 backend.mongo.pool1.database = mqtt
34
35 ## MongoDB 认证用户名密码
36 ## backend.mongo.pool1.login = emqtt
## backend.mongo.pool1.password = emqtt

```

```

37 ## backend.mongo.pool1.password = emqx
38 ## 指定用于授权的数据库, 没有指定时默认为 admin
39 ##
40 ## 如果启用了 srv_record, 即 backend.mongo.<Pool>.srv_record 设置为 true,
41 ## 且您的 MongoDB 服务器域名添加了包含 authSource 选项的 DNS TXT 记录,
42 ## 那么可以忽略此配置项
43 ## backend.mongo.pool1.auth_source = admin
44
45 ## 是否开启 SSL
46 ## backend.mongo.pool1.ssl = false
47
48 ## SSL 密钥文件路径
49 ## backend.mongo.pool1.keyfile =
50
51 ## SSL 证书文件路径
52 ## backend.mongo.pool1.certfile =
53
54 ## SSL CA 证书文件路径
55 ## backend.mongo.pool1.cacertfile =
56
57 ## MongoDB 数据写入模式: unsafe | safe
58 ## backend.mongo.pool1.w_mode = safe
59
60 ## MongoDB 数据读取模式: master | slaver_ok
61 ## backend.mongo.pool1.r_mode = slave_ok
62
63 ## MongoDB 底层 driver 配置, 保持默认即可
64 ## backend.mongo.topology.pool_size = 1
65 ## backend.mongo.topology.max_overflow = 0
66 ## backend.mongo.topology.overflow_ttl = 1000
67 ## backend.mongo.topology.overflow_check_period = 1000
68 ## backend.mongo.topology.local_threshold_ms = 1000
69 ## backend.mongo.topology.connect_timeout_ms = 20000
70 ## backend.mongo.topology.socket_timeout_ms = 100
71 ## backend.mongo.topology.server_selection_timeout_ms = 30000
72 ## backend.mongo.topology.wait_queue_timeout_ms = 1000
73 ## backend.mongo.topology.heartbeat_frequency_ms = 10000
74 ## backend.mongo.topology.min_heartbeat_frequency_ms = 1000
75
76 ## MongoDB Backend Hooks
77 backend.mongo.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "pool": "pool1"}
78 backend.mongo.hook.session.created.1 = {"action": {"function": "on_subscribe_lookup"}, "pool": "pool1"}
79 backend.mongo.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}, "pool": "pool1"}
80 backend.mongo.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_fetch"}, "pool": "pool1", "offline_opts": {"time_range": "2h", "max_returned_count": 500}}
81 backend.mongo.hook.session.subscribed.2 = {"topic": "#", "action": {"function": "on_retain_lookup"}, "pool": "pool1"}
82 backend.mongo.hook.session.unsubscribed.1 = {"topic": "#", "action": {"function": "on_acked_delete"}, "pool": "pool1"}
83 backend.mongo.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}
84 backend.mongo.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message_retain"}, "pool": "pool1"}
85 backend.mongo.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_delete"}, "pool": "pool1"}
86 backend.mongo.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message_acked"}, "pool": "pool1"}
87
88
89
90
91
92
93
94
95

```

96

```
_acked"}, "pool": "pool1"

## 获取离线消息
### "offline_opts": 获取离线消息的配置
#### - max_returned_count: 单次拉去的最大离线消息数目
#### - time_range: 仅拉去在当前时间范围的消息
## backend.mongo.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_fetch"}, "pool": "pool1", "offline_opts": {"time_range": "2h", "max_returned_count": 500}}

## 如果需要存储 Qos0 消息, 可开启以下配置
## 警告: 当开启以下配置时, 需关闭 'on_message_fetch', 否则 qos1, qos2 消息会被存储两次
## backend.mongo.hook.message.publish.4 = {"topic": "#", "action": {"function": "on_message_store"}, "pool": "pool1", "payload_format": "mongo_json"}
```

backend 消息存储规则包括:

hook	topic	action	说明
client.connected		on_client_connected	存储客户端在线状态
session.created		on_subscribe_lookup	订阅主题
client.disconnected		on_client_disconnected	存储客户端离线状态
session.subscribed	#	on_message_fetch	获取离线消息
session.subscribed	#	on_retain_lookup	获取retain消息
session.unsubscribed	#	on_acked_delete	删除 acked 消息
message.publish	#	on_message_publish	存储发布消息
message.publish	#	on_message_retain	存储retain消息
message.publish	#	on_retain_delete	删除retain消息
message.acked	#	on_message_acked	消息ACK处理

MongoDB 数据库初始化

```
1 use mqtt
2 db.createCollection("mqtt_client")
3 db.createCollection("mqtt_sub")
4 db.createCollection("mqtt_msg")
5 db.createCollection("mqtt_retain")
6 db.createCollection("mqtt_acked")
7
8 db.mqtt_client.ensureIndex({clientid:1, node:2})
9 db.mqtt_sub.ensureIndex({clientid:1})
10 db.mqtt_msg.ensureIndex({sender:1, topic:2})
11 db.mqtt_retain.ensureIndex({topic:1})
```

MongoDB 用户状态集合(Client Collection)

mqtt_client 存储设备在线状态:

```

1  {
2      clientid: string,
3      state: 0,1, //0离线 1在线
4      node: string,
5      online_at: timestamp,
6      offline_at: timestamp
7  }

```

js

查询设备在线状态:

```
1 db.mqtt_client.findOne({clientid: ${clientid}})
```

js

例如 **ClientId** 为 **test** 客户端上线:

```

1 db.mqtt_client.findOne({clientid: "test"})
2
3 {
4     "_id" : ObjectId("58646c9bdde89a9fb9f7fb73"),
5     "clientid" : "test",
6     "state" : 1,
7     "node" : "emqx@127.0.0.1",
8     "online_at" : 1482976411,
9     "offline_at" : null
10 }

```

js

例如 **ClientId** 为 **test** 客户端下线:

```

1 db.mqtt_client.findOne({clientid: "test"})
2
3 {
4     "_id" : ObjectId("58646c9bdde89a9fb9f7fb73"),
5     "clientid" : "test",
6     "state" : 0,
7     "node" : "emqx@127.0.0.1",
8     "online_at" : 1482976411,
9     "offline_at" : 1482976501
10 }

```

js

MongoDB 用户订阅主题集合(Subscription Collection)

mqtt_sub 存储订阅关系:

```

1  {
2      clientid: string,
3      topic: string,
4      qos: 0,1,2
5  }

```

js

用户 **test** 分别订阅主题 **test_topic0 test_topic1 test_topic2**:

```

1 db.mqtt_sub.insert({clientid: "test", topic: "test_topic1", qos: 1})
2 db.mqtt_sub.insert({clientid: "test", topic: "test_topic2", qos: 2})

```

js

某个客户端订阅主题:

```

1 db.mqtt_sub.find({clientid: ${clientid}})

```

js

查询 **ClientId** 为 "test" 的客户端已订阅主题:

```

1 db.mqtt_sub.find({clientid: "test"})
2
3 { "_id" : ObjectId("58646d90c65dff6ac9668ca1"), "clientid" : "test", "topic" : "test_topic1",
4   "qos" : 1 }
5 { "_id" : ObjectId("58646d96c65dff6ac9668ca2"), "clientid" : "test", "topic" : "test_topic2",
6   "qos" : 2 }

```

js

MongoDB 发布消息集合(Message Collection)

mqtt_msg 存储 MQTT 消息:

```

1 {
2   _id: int,
3   topic: string,
4   msgid: string,
5   sender: string,
6   qos: 0,1,2,
7   retain: boolean (true, false),
8   payload: string,
9   arrived: timestamp
10 }

```

js

查询某个客户端发布的消息:

```

1 db.mqtt_msg.find({sender: ${clientid}})

```

js

查询 **ClientId** 为 "test" 的客户端发布的消息:

```

1 db.mqtt_msg.find({sender: "test"})
2 {
3   "_id" : 1,
4   "topic" : "/World",
5   "msgid" : "AAVEwm0la4RufgAABeIAAQ==",
6   "sender" : "test",
7   "qos" : 1,
8   "retain" : 1,
9   "payload" : "Hello world!",
10  "arrived" : 1482976729
11 }

```

js

MongoDB 保留消息集合(Retain Message Collection)

mqtt_retain 存储 Retain 消息:

```

1  {
2    topic: string,
3    msgid: string,
4    sender: string,
5    qos: 0,1,2,
6    payload: string,
7    arrived: timestamp
8 }
```

查询 retain 消息:

```
1 db.mqtt_retain.findOne({topic: ${topic}})
```

查询topic为 "t/retain" 的 retain 消息:

```

1 db.mqtt_retain.findOne({topic: "t/retain"})
{
3   "_id" : ObjectId("58646dd9dde89a9fb9f7fb75"),
4   "topic" : "t/retain",
5   "msgid" : "AAVEwm0la4RufgAABeIAAQ==",
6   "sender" : "c1",
7   "qos" : 1,
8   "payload" : "Hello world!",
9   "arrived" : 1482976729
10 }
```

MongoDB 接收消息 ack 集合(Message Acked Collection)

mqtt_acked 存储客户端消息确认:

```

1  {
2    clientid: string,
3    topic: string,
4    mongo_id: int
5 }
```

启用 MongoDB 数据存储插件

```
1 ./bin/emqx_ctl plugins load emqx_backend_mongo
```

Cassandra 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 Cassandra](#)规则引擎中创建保存数据到 Cassandra

配置 Cassandra 服务器

配置文件: **emqx_backend_cassa.conf**

支持配置多台Cassandra服务器连接池:

```

1 ## Cassandra 节点地址
2 backend.ecql.pool1.nodes = 127.0.0.1:9042
3
4 ## Cassandra 连接池大小
5 backend.ecql.pool1.size = 8
6
7 ## Cassandra 自动重连间隔(s)
8 backend.ecql.pool1.auto_reconnect = 1
9
10 ## Cassandra 认证用户名/密码
11 backend.ecql.pool1.username = cassandra
12 backend.ecql.pool1.password = cassandra
13
14 ## Cassandra Keyspace
15 backend.ecql.pool1.keyspace = mqtt
16
17 ## Cassandra Logger type
18 backend.ecql.pool1.logger = info
19
20 #-----
21 ## Cassandra Backend Hooks
22 #-----
23
24 ## Client Connected Record
25 backend.cassa.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "pool": "pool1"}
26
27 ## Subscribe Lookup Record
28 backend.cassa.hook.session.created.1 = {"action": {"function": "on_subscription_lookup"}, "pool": "pool1"}
29
30 ## Client DisConnected Record
31 backend.cassa.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}, "pool": "pool1"}
32
33 ## Lookup Unread Message QOS > 0
34 backend.cassa.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_fetch"}, "pool": "pool1"}
35
36 ## Lookup Retain Message
37
38
39
40

```

```

41 backend.cassa.hook.session.subscribed.2 = {"action": {"function": "on_retain_lookup"}, "pool": "pool1"}
42
43
44 ## Store Publish Message QOS > 0
45 backend.cassa.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}
46
47
48 ## Delete Acked Record
49 backend.cassa.hook.session.unsubscribed.1= {"topic": "#", "action": {"cql": ["delete from acked where clientid = ${clientid} and topic = ${topic}"]}, "pool": "pool1"}
50
51
52 ## Store Retain Message
53 backend.cassa.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message_retain"}, "pool": "pool1"}
54
55
56 ## Delete Retain Message
57 backend.cassa.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_delete"}, "pool": "pool1"}
58
59
60 ## Store Ack
61 backend.cassa.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message_acked"}, "pool": "pool1"}
62

## 获取离线消息
#### "offline_opts": 获取离线消息的配置
##### - max_returned_count: 单次拉去的最大离线消息数目
##### - time_range: 仅拉去在当前时间范围的消息
## backend.cassa.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_fetch"}, "offline_opts": {"max_returned_count": 500, "time_range": "2h"}, "pool": "pool1"}

## 如果需要存储 Qos0 消息, 可开启以下配置
## 警告: 当开启以下配置时, 需关闭 'on_message_fetch', 否则 qos1, qos2 消息会被存储俩次
## backend.cassa.hook.message.publish.4 = {"topic": "#", "action": {"function": "on_message_store"}, "pool": "pool1"}

```

backend 消息存储规则包括:

hook	topic	action	说明
client.connected		on_client_connected	存储客户端在线状态
session.created		on_subscribe_lookup	订阅主题
client.disconnected		on_client_disconnected	存储客户端离线状态
session.subscribed	#	on_message_fetch	获取离线消息
session.subscribed	#	on_retain_lookup	获取 retain 消息
session.unsubscribed	#		删除 akced 消息
message.publish	#	on_message_publish	存储发布消息
message.publish	#	on_message_retain	存储 retain 消息
message.publish	#	on_retain_delete	删除 retain 消息
message.acked	#	on_message_acked	消息 ACK 处理

自定义 **CQL** 语句 可用参数包括:

hook	可用参数	示例(cql语句中\${name} 表示可获取的参数)
client.connected	clientid	insert into conn(clientid) values(\${clientid})
client.disconnected	clientid	insert into disconn(clientid) values(\${clientid})
session.subscribed	clientid, topic, qos	insert into sub(topic, qos) values(\${topic}, \${qos})
session.unsubscribed	clientid, topic	delete from sub where topic = \${topic}
message.publish	msgid, topic, payload, qos, clientid	insert into msg(msgid, topic) values(\${msgid}, \${topic})
message.acked	msgid, topic, clientid	insert into ack(msgid, topic) values(\${msgid}, \${topic})
message.deliver	msgid, topic, clientid	insert into deliver(msgid, topic) values(\${msgid}, \${topic})

支持 **CQL** 语句配置：

考虑到用户的需求不同, **backend cassandra** 自带的函数无法满足用户需求, 用户可根据自己的需求配置 **cql** 语句

在 **etc/plugins/emqx_backend_cassa.conf** 中添加如下配置:

```
1 ## 在客户端连接到 EMQX 服务器后, 执行一条 cql 语句(支持多条 cql 语句)
2 backend.cassa.hook.client.connected.3 = {"action": {"cql": ["insert into conn(clientid) value
s(${clientid})"]}, "pool": "pool1"}
```

Cassandra 创建一个 Keyspace

```
1 CREATE KEYSPACE mqtt WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'
2 '} AND durable_writes = true;
USE mqtt;
```

导入 Cassandra 表结构

```
1 cqlsh -e "SOURCE 'emqx_backend_cassa.cql'"
```

Cassandra 用户状态表(Client Table)

mqtt.client 存储设备在线状态:

```

1 CREATE TABLE mqtt.client (
2     clientid text PRIMARY KEY,
3     connected timestamp,
4     disconnected timestamp,
5     node text,
6     state int
7 );

```

查询设备在线状态:

```
1 select * from mqtt.client where clientid = ${clientid};
```

sh

例如 **ClientId** 为 **test** 的客户端上线:

```

1 select * from mqtt.client where clientid = 'test';
2
3   clientid | connected           | disconnected | node      | state
4   +-----+-----+-----+-----+
5   test    | 2017-02-14 08:27:29.872000+0000 | null       | emqx@127.0.0.1 | 1

```

sh

例如**ClientId**为**test**客户端下线:

```

1 select * from mqtt.client where clientid = 'test';
2
3   clientid | connected           | disconnected | node
4   | state
5   +-----+-----+-----+-----+
6
7   test    | 2017-02-14 08:27:29.872000+0000 | 2017-02-14 08:27:35.872000+0000 | emqx@127.0.
8   0.1 | 0

```

sh

Cassandra 用户订阅主题表(Sub Table)

mqtt.sub 存储订阅关系:

```

1 CREATE TABLE mqtt.sub (
2     clientid text,
3     topic text,
4     qos int,
5     PRIMARY KEY (clientid, topic)
6 );

```

用户**test**分别订阅主题**test_topic1** **test_topic2**:

```

1 insert into mqtt.sub(clientid, topic, qos) values('test', 'test_topic1', 1);
2 insert into mqtt.sub(clientid, topic, qos) values('test', 'test_topic2', 2);

```

某个客户端订阅主题:

```
1 select * from mqtt_sub where clientid = ${clientid};
```

查询**ClientId**为'test'的客户端已订阅主题:

```
1 select * from mqtt_sub where clientid = 'test';
2
3   clientid | topic      | qos
4   -----+-----+-----
5     test  | test_topic1 | 1
6     test  | test_topic2 | 2
```

Cassandra 发布消息表(Msg Table)

mqtt.msg 存储MQTT消息:

```
1 CREATE TABLE mqtt.msg (
2   topic text,
3   msgid text,
4   arrived timestamp,
5   payload text,
6   qos int,
7   retain int,
8   sender text,
9   PRIMARY KEY (topic, msgid)
10 ) WITH CLUSTERING ORDER BY (msgid DESC);
```

查询某个客户端发布的消息:

```
1 select * from mqtt_msg where sender = ${clientid};
```

查询**ClientId**为'test'的客户端发布的消息:

```
1 select * from mqtt_msg where sender = 'test';
2
3   topic | msgid           | arrived          | payload | qos | retain
4   n    | sender
5   -----+-----+-----+-----+-----+-----+
6
7   hello | 2PguFrHsrzEvIIBdctmb | 2017-02-14 09:07:13.785000+0000 | Hello world! | 1 | 0
8   | test
9   world | 2PguFrHsrzEvIIBdctmb | 2017-02-14 09:07:13.785000+0000 | Hello world! | 1 | 0
0   | test
```

Cassandra 保留消息表(Retain Message Table)

mqtt.retain 存储 Retain 消息:

```
1 CREATE TABLE mqtt.retain (
2     topic text PRIMARY KEY,
3     msgid text
4 );
```

查询 **retain** 消息:

```
1 select * from mqtt_retain where topic = ${topic};
```

查询 **topic** 为 '**t/retain**' 的 **retain** 消息:

```
1 select * from mqtt_retain where topic = 't/retain';
2
3     topic | msgid
4   -----+-----
5     retain | 2PguFrHsrzEvIIBdctmb
```

Cassandra 接收消息 **ack** 表(Message Acked Table)

mqtt.acked 存储客户端消息确认:

```
1 CREATE TABLE mqtt.acked (
2     clientid text,
3     topic text,
4     msgid text,
5     PRIMARY KEY (clientid, topic)
6 );
```

启用 Cassandra 存储插件

```
1 ./bin/emqx_ctl plugins load emqx_backend_cassa
```

DynamoDB 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 DynamoDB](#)规则引擎中创建 保存数据到 **DynamoDB**

配置 DyanmoDB 消息存储

配置文件: **etc/plugins/emqx_backend_dynamo.conf**

```

1 ## DynamoDB Region
2 backend.dynamo.region = us-west-2
3
4 ## DynamoDB Server
5 backend.dynamo.pool1.server = http://localhost:8000
6
7 ## DynamoDB Pool Size
8 backend.dynamo.pool1.pool_size = 8
9
10 ## AWS ACCESS KEY ID
11 backend.dynamo.pool1.aws_access_key_id = AKIAU5IM2XOC7AQWG7HK
12
13 ## AWS SECRET ACCESS KEY
14 backend.dynamo.pool1.aws_secret_access_key = TZt7XoRi+vtCJYQ9YsAinh19jR1rngm/hxZMWR2P
15
16 ## DynamoDB Backend Hooks
17 backend.dynamo.hook.client.connected.1 = {"action": {"function": "on_client_connected"}, "pool": "pool1"}
18 backend.dynamo.hook.session.created.1 = {"action": {"function": "on_subscribe_lookup"}, "pool": "pool1"}
19 backend.dynamo.hook.client.disconnected.1 = {"action": {"function": "on_client_disconnected"}, "pool": "pool1"}
20 backend.dynamo.hook.session.subscribed.1 = {"topic": "#", "action": {"function": "on_message_fetch_for_queue"}, "pool": "pool1"}
21 backend.dynamo.hook.session.subscribed.2 = {"topic": "#", "action": {"function": "on_retain_lookup"}, "pool": "pool1"}
22 backend.dynamo.hook.session.unsubscribed.1 = {"topic": "#", "action": {"function": "on_acked_delete"}, "pool": "pool1"}
23 backend.dynamo.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}
24 backend.dynamo.hook.message.publish.2 = {"topic": "#", "action": {"function": "on_message_retain"}, "pool": "pool1"}
25 backend.dynamo.hook.message.publish.3 = {"topic": "#", "action": {"function": "on_retain_delete"}, "pool": "pool1"}
26 backend.dynamo.hook.message.acked.1 = {"topic": "#", "action": {"function": "on_message_acked_for_queue"}, "pool": "pool1"}
27
28 # backend.dynamo.hook.message.publish.4 = {"topic": "#", "action": {"function": "on_message_sore"}, "pool": "pool1"}  


```

backend 消息存储规则包括:

hook	topic	action	说明
<code>client.connected</code>		<code>on_client_connected</code>	存储客户端在线状态
<code>client.connected</code>		<code>on_subscribe_lookup</code>	订阅主题
<code>client.disconnected</code>		<code>on_client_disconnected</code>	存储客户端离线状态
<code>session.subscribed</code>	#	<code>on_message_fetch_for_queue</code>	获取一对一离线消息
<code>session.subscribed</code>	#	<code>on_retain_lookup</code>	获取retain消息
<code>message.publish</code>	#	<code>on_message_publish</code>	存储发布消息
<code>message.publish</code>	#	<code>on_message_retain</code>	存储retain消息
<code>message.publish</code>	#	<code>on_retain_delete</code>	删除retain消息
<code>message.acked</code>	#	<code>on_message_acked_for_queue</code>	一对一消息ACK处理

DynamoDB 数据库创建表

```
1 ./test/dynamo_test.sh
```

sh

DynamoDB 用户状态表(Client Table)

mqtt_client 表定义(存储设备在线状态):

```
1 {
2     "TableName": "mqtt_client",
3     "KeySchema": [
4         { "AttributeName": "clientid", "KeyType": "HASH" }
5     ],
6     "AttributeDefinitions": [
7         { "AttributeName": "clientid", "AttributeType": "S" }
8     ],
9     "ProvisionedThroughput": {
10         "ReadCapacityUnits": 5,
11         "WriteCapacityUnits": 5
12     }
13 }
```

json

查询设备在线状态:

```

1 aws dynamodb scan --table-name mqtt_client --region us-west-2 --endpoint-url http://localhost:4000
2
3
4 {
5     "Items": [
6         {
7             "offline_at": { "N": "0" },
8             "node": { "S": "emqx@127.0.0.1" },
9             "clientid": { "S": "mqttjs_384b9c73a9" },
10            "connect_state": { "N": "1" },
11            "online_at": { "N": "1562224940" }
12        }
13    ],
14    "Count": 1,
15    "ScannedCount": 1,
16    "ConsumedCapacity": null
17 }

```

DynamoDB 用户订阅主题(Subscription Table)

mqtt_sub 表定义(存储订阅关系):

```

1 {
2     "TableName": "mqtt_sub",
3     "KeySchema": [
4         { "AttributeName": "clientid", "KeyType": "HASH" },
5         { "AttributeName": "topic", "KeyType": "RANGE" }
6     ],
7     "AttributeDefinitions": [
8         { "AttributeName": "clientid", "AttributeType": "S" },
9         { "AttributeName": "topic", "AttributeType": "S" }
10    ],
11    "ProvisionedThroughput": {
12        "ReadCapacityUnits": 5,
13        "WriteCapacityUnits": 5
14    }
15 }

```

查询 ClientId 为 "test-dynamo" 的客户端已订阅主题:

```

1 aws dynamodb scan --table-name mqtt_sub --region us-west-2 --endpoint-url http://localhost:8000
2
3
4 {
5     "Items": [{"qos": { "N": "2" }, "topic": { "S": "test-dynamo-sub" }, "clientid": { "S": "test-dynamo" }},
6                 {"qos": { "N": "2" }, "topic": { "S": "test-dynamo-sub-1"}, "clientid": { "S": "test-dynamo" }},
7                 {"qos": { "N": "2" }, "topic": { "S": "test-dynamo-sub-2"}, "clientid": { "S": "test-dynamo" }}],
8             "Count": 3,
9             "ScannedCount": 3,
10            "ConsumedCapacity": null
11        }
12    }

```

DynamoDB 发布消息(Message Table)

mqtt_msg 表定义(存储 MQTT 消息):

```

1 {
2     "TableName": "mqtt_msg",
3     "KeySchema": [
4         { "AttributeName": "msgid", "KeyType": "HASH" }
5     ],
6     "AttributeDefinitions": [
7         { "AttributeName": "msgid", "AttributeType": "S" }
8     ],
9     "ProvisionedThroughput": {
10         "ReadCapacityUnits": 5,
11         "WriteCapacityUnits": 5
12     }
13 }

```

mqtt_topic_msg_map 表定义(存储主题和消息的映射关系):

```

1 {
2     "TableName": "mqtt_topic_msg_map",
3     "KeySchema": [
4         { "AttributeName": "topic", "KeyType": "HASH" }
5     ],
6     "AttributeDefinitions": [
7         { "AttributeName": "topic", "AttributeType": "S" }
8     ],
9     "ProvisionedThroughput": {
10         "ReadCapacityUnits": 5,
11         "WriteCapacityUnits": 5
12     }
13 }

```

某个客户端向主题 **test** 发布消息后, 查询 **mqtt_msg** 表和 **mqtt_topic_msg_map** 表:

查询 **mqtt_msg** 表:

```

1 aws dynamodb scan --table-name mqtt_msg --region us-west-2 --endpoint-url http://localhost:8000
2
3
4 > - {
5 >   - "Items": [
6 >     - {
7 >       "arrived": { "N": "1562308553" }, "qos": { "N": "1" },
8 >       "sender": { "S": "mqttjs_231b962d5c" }, "payload": { "S": "
9 >       "{ \"msg\": \"Hello, World\\!\" }" }, "retain": { "N": "0" },
10 >      "msgid": { "S": "
11 >        "Mjg4MTk1MDYwNTk0NjYwNzYzMTg4MDk30TQ2MDU2Nzg10TD"
12 >      "topic": { "S": "test" }
13 >    }
14 >  ], "Count": 1, "ScannedCount": 1, "ConsumedCapacity": null
15 > }
```

查询 **mqtt_topic_msg_map** 表:

```

1 aws dynamodb scan --table-name mqtt_topic_msg_map --region us-west-2 --endpoint-url http://localhost:8000
2
3
4 > - {
5 >   - "Items": [
6 >     - {
7 >       "topic": { "S": "test" }, "MsgId": { "SS": [
8 >         "Mjg4MTk1MDYwNTk0NjYwNzYzMTg4MDk30TQ2MDU2Nzg10TD"
9 >       ] }
10 >     }
11 >   ], "Count": 1, "ScannedCount": 1, "ConsumedCapacity": null
12 > }
```

DynamoDB 保留消息(Retain Message Table)

mqtt_retain 表定义(存储 retain 消息):

```

1 {
2   "TableName": "mqtt_retain",
3   "KeySchema": [
4     { "AttributeName": "topic", "KeyType": "HASH" }
5   ],
6   "AttributeDefinitions": [
7     { "AttributeName": "topic", "AttributeType": "S" }
8   ],
9   "ProvisionedThroughput": {
10     "ReadCapacityUnits": 5,
11     "WriteCapacityUnits": 5
12   }
13 }
```

某个客户端向主题 **test** 发布消息后, 查询 **mqtt_retain** 表:

```

1  {
2      "Items": [
3          {
4              "arrived": { "N": "1562312113" },
5              "qos": { "N": "1" },
6              "sender": { "S": "mqttjs_d0513acfce" },
7              "payload": { "S": "test" },
8              "retain": { "N": "1" },
9              "msgid": { "S": "Mjg4MTk1NzE3MTY4MjYxMjA5MDExMDg0NTk50DgzMjAyNTH" },
10             "topic": { "S": "testtopic" }
11         }
12     ],
13     "Count": 1,
14     "ScannedCount": 1,
15     "ConsumedCapacity": null
16 }

```

json

DynamoDB 接收消息 ack (Message Acked Table)

mqtt_acked 表定义(存储确认的消息):

```

1  {
2      "TableName": "mqtt_acked",
3      "KeySchema": [
4          { "AttributeName": "topic", "KeyType": "HASH" },
5          { "AttributeName": "clientid", "KeyType": "RANGE" }
6      ],
7      "AttributeDefinitions": [
8          { "AttributeName": "topic", "AttributeType": "S" },
9          { "AttributeName": "clientid", "AttributeType": "S" }
10     ],
11     "ProvisionedThroughput": {
12         "ReadCapacityUnits": 5,
13         "WriteCapacityUnits": 5
14     }
15 }

```

json

某个客户端向主题 **test** 发布消息后, 查询 **mqtt_acked** 表:

```

1  {
2      "Items": [
3          {
4              "topic": { "S": "test" },
5              "msgid": { "S": "Mjg4MTk1MDYwNTk0NjYwNzYzMjg4MDk30TQ2MDU2Nzg10TD" },
6              "clientid": { "S": "mqttjs_861e582a70" }
7          }
8      ],
9      "Count": 1,
10     "ScannedCount": 1,
11     "ConsumedCapacity": null
12 }

```

json

启用 **DynamoDB** 消息存储:

```
1 ./bin/emqx_ctl plugins load emqx_backend_dynamo      sh
```

InfluxDB 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 InfluxDB](#)规则引擎中创建保存数据到 InfluxDB

InfluxDB 配置

EMQX 仅支持通过 UDP 协议连接 InfluxDB，需要修改 InfluxDB 配置文件：

```
1  [[udp]]  
2    enabled = true  
3    bind-address = ":8089"  
4    # 消息保存的数据库  
5    database = "emqx"  
6  
7    # InfluxDB precision for timestamps on received points ("n" or "n", "u", "ms", "s", "m", "h")  
8  
9    # EMQX 默认时间戳是毫秒  
10   precision = "ms"  
11  
12   # 其他配置根据需要自行修改  
13   #   batch-size = 1000  
14   #   batch-pending = 5  
15   #   batch-timeout = "5s"  
#   read-buffer = 1024
```

配置 InfluxDB 消息存储

配置文件 `etc/plugins/emqx_backend_influxdb.conf`:

sh

```
1 ## 写数据到 InfluxDB 时使用的协议
2 backend.influxdb.pool1.common.write_protocol = udp
3
4 ## 批量写入大小
5 backend.influxdb.pool1.common.batch_size = 1000
6
7 ## InfluxDB 写进程池大小
8 backend.influxdb.pool1.pool_size = 8
9
10 ## InfluxDB UDP 主机地址
11 backend.influxdb.pool1.udp.host = 127.0.0.1
12
13 ## InfluxDB UDP 主机端口
14 backend.influxdb.pool1.udp.port = 8089
15
16 ## InfluxDB HTTP/HTTPS 主机地址
17 backend.influxdb.pool1.http.host = 127.0.0.1
18
19 ## InfluxDB HTTP/HTTPS 主机端口
20 backend.influxdb.pool1.http.port = 8086
21
22 ## InfluxDB 数据库名
23 backend.influxdb.pool1.http.database = mydb
24
25 ## 连接到 InfluxDB 的用户名
26 ## backend.influxdb.pool1.http.username = admin
27
28 ## 连接到 InfluxDB 的密码
29 ## backend.influxdb.pool1.http.password = public
30
31 ## 时间戳精度
32 backend.influxdb.pool1.http.precision = ms
33
34 ## 是否启用 HTTPS
35 backend.influxdb.pool1.http.https_enabled = false
36
37 ## 连接 InfluxDB 时使用的 TLS 协议版本
38 ## backend.influxdb.pool1.http.ssl.version = tlsv1.2
39
40 ## 密钥文件
41 ## backend.influxdb.pool1.http.ssl.keyfile =
42
43 ## 证书文件
44 ## backend.influxdb.pool1.http.ssl.certfile =
45
46 ## CA 证书文件
47 ## backend.influxdb.pool1.http.ssl.cacertfile =
48
49 ## 存储 PUBLISH 消息
50 backend.influxdb.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}
```

InfluxDB Backend 消息存储规则参数:

Option	Description
topic	配置哪些主题下的消息需要执行 hook
action	配置 hook 具体动作, function 为 Backend 提供的内置函数, 实现通用功能
pool	Pool Name , 实现连接多个 InfluxDB Server 功能

Example:

```

1 ## 存储主题为 "sensor/#" 的 PUBLISH 消息
2 backend.influxdb.hook.message.publish.1 = {"topic": "sensor/#", "action": {"function": "on_m
3   essage_publish"}, "pool": "pool1"}
4
5 ## 存储主题为 "stat/#" 的 PUBLISH 消息
6 backend.influxdb.hook.message.publish.2 = {"topic": "stat/#", "action": {"function": "on_mes
7   sage_publish"}, "pool": "pool1"}sh
```

InfluxDB Backend 支持 **Hook** 与 相应内置函数列表:

Hook	Function list
message.publish	on_message_publish

由于 **MQTT Message** 无法直接写入 **InfluxDB**, **InfluxDB Backend** 提供了 **emqx_backend_influxdb.tpl** 模板文件将 **MQTT Message** 转换为可写入 **InfluxDB** 的 **DataPoint**。

模板文件采用 **JSON** 格式, 组成部分:

- **key** - **MQTT Topic**, 字符串, 支持通配符
- **value** - **Template**, **Json** 对象, 用于将 **MQTT Message** 转换成 `measurement,tag_key=tag_value,... field_key=field_value,... timestamp` 的形式以写入 **InfluxDB**。

你可以为不同 **Topic** 定义不同的 **Template**, 也可以为同一个 **Topic** 定义多个 **Template**, 类似:

```

1 {
2     <Topic 1>: <Template 1>,
3     <Topic 2>: <Template 2>
4 }sh
```

Template 格式如下:

```

1 {
2     "measurement": <Measurement>,
3     "tags": {
4         <Tag Key>: <Tag Value>
5     },
6     "fields": {
7         <Field Key>: <Field Value>
8     },
9     "timestamp": <Timestamp>
10 }sh
```

`measurement` 与 `fields` 为必选项, `tags` 与 `timestamp` 为可选项。

所有的值 (例如 `<Measurement>`) 你都可以直接在 **Template** 中配置为一个固定值, 它支持的数据类型依赖于你定义的数据表。当然更符合实际情况的是, 你可以通过我们提供的占位符来获取 **MQTT** 消息中的数据。

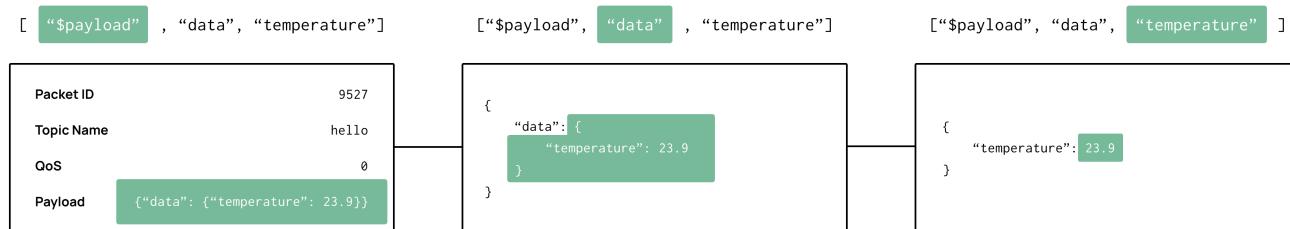
目前我们支持的占位符如下:

Placeholder	Description
<code>\$id</code>	MQTT 消息 UUID , 由 EMQX 分配
<code>\$clientid</code>	客户端使用的 Client ID
<code>\$username</code>	客户端使用的 Username
<code>\$peerhost</code>	客户端 IP
<code>\$qos</code>	MQTT 消息的 QoS
<code>\$topic</code>	MQTT 消息主题
<code>\$payload</code>	MQTT 消息载荷, 必须为合法的 Json
<code>\$<Number></code>	必须配合 <code>\$payload</code> 使用, 用于从 Json Array 中获取数据
<code>\$timestamp</code>	EMQX 准备转发消息时设置的时间戳, 精度: 纳秒

`$payload` 与 `$<Number>`:

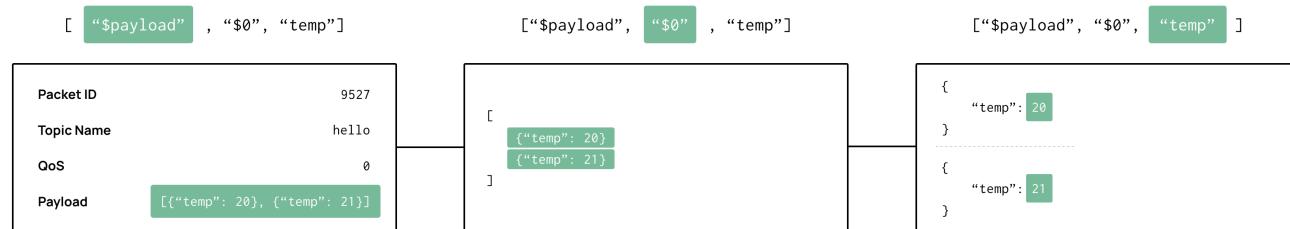
你可以直接使用 `$payload` 取得完整的消息载荷, 也可以通过 `["$payload", <Key>, ...]` 取得消息载荷内部的数据。

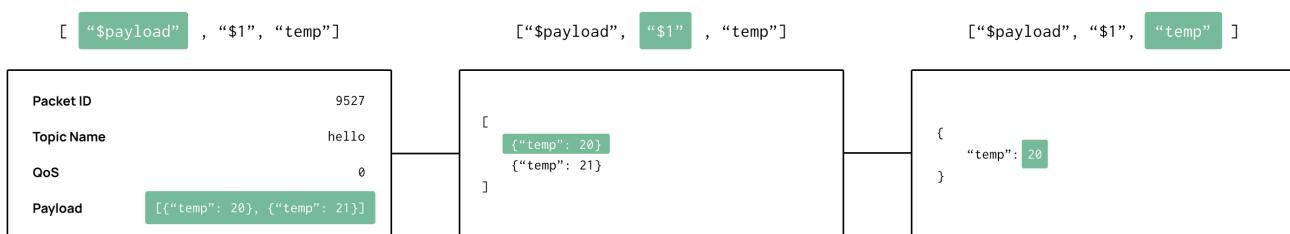
例如 `payload` 为 `{"data": {"temperature": 23.9}}`, 你可以通过占位符 `["$payload", "data", "temperature"]` 来获取其中的 `23.9`。



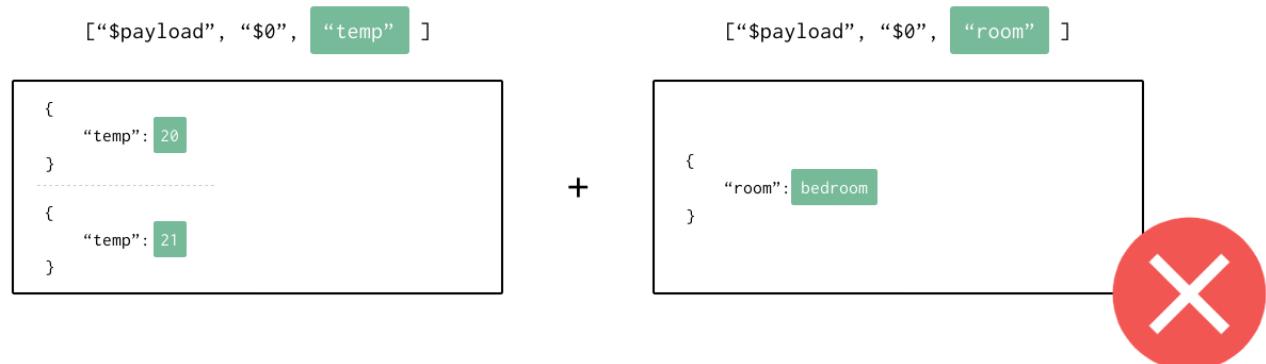
考虑到 **Json** 还有数组这一数据类型的情况, 我们引入了 `$0` 与 `$<pos_integer>`, `$0` 表示获取数组内所有元素, `$<pos_integer>` 表示获取数组内第 `<pos_integer>` 个元素。

一个简单例子, `["$payload", "$0", "temp"]` 将从 `[{"temp": 20}, {"temp": 21}]` 中取得 `[20, 21]`, 而 `["$payload", "$1", "temp"]` 将只取得 `20`。





值得注意的是，当你使用 `$0` 时，我们希望你取得的数据个数都是相等的。因为我们需要将这些数组转换为多条记录写入 **InfluxDB**，而当你一个字段取得了 3 份数据，另一个字段却取得了 2 份数据，我们将无从判断应当怎样为你组合这些数据。



Example

data/templates 目录下提供了一个示例模板 (`emqx_backend_influxdb_example.tpl`，正式使用时请去掉文件名中的 `_example` 后缀) 供用户参考：

```

1  {
2      "sample": {
3          "measurement": "$topic",
4          "tags": {
5              "host": ["$payload", "data", "$0", "host"],
6              "region": ["$payload", "data", "$0", "region"],
7              "qos": "$qos",
8              "clientid": "$clientid"
9          },
10         "fields": {
11             "temperature": ["$payload", "data", "$0", "temp"]
12         },
13         "timestamp": "$timestamp"
14     }
15 }
```

提示

当 **Template** 中设置 **timestamp** 或插件配置 `backend.influxdb.pool1.set_timestamp = true` 时，请将 **InfluxDB UDP** 配置中的 **precision** 设为 "ms"。

当 Topic 为 "sample" 的 MQTT Message 拥有以下 Payload 时：

json

```

1  {
2      "data": [
3          {
4              "temp": 1,
5              "host": "serverA",
6              "region": "hangzhou"
7          },
8          {
9              "temp": 2,
10             "host": "serverB",
11             "region": "ningbo"
12         }
13     ]
14 }

```

Backend 会将 **MQTT Message** 转换为:

json

```

1  [
2      {
3          "measurement": "sample",
4          "tags": {
5              "clientid": "mqttjs_ebcc36079a",
6              "host": "serverA",
7              "qos": "0",
8              "region": "hangzhou",
9          },
10         "fields": {
11             "temperature": "1"
12         },
13         "timestamp": "1560743513626681000"
14     },
15     {
16         "measurement": "sample",
17         "tags": {
18             "clientid": "mqttjs_ebcc36079a",
19             "host": "serverB",
20             "qos": "0",
21             "region": "ningbo",
22         },
23         "fields": {
24             "temperature": "2"
25         },
26         "timestamp": "1560743513626681000"
27     }
28 ]

```

最终编码为以下数据写入 **InfluxDB**:

sh

```

1 "sample,clientid= mqttjs_6990f0e886,host=serverA,qos=0,region=hangzhou temperature=\"1\" 1560
745505429670000\nsample,clientid= mqttjs_6990f0e886,host=serverB,qos=0,region=ningbo temperature
\"2\" 1560745505429670000\n"

```

启用 **InfluxDB** 消息存储:

```
1 ./bin/emqx_ctl plugins load emqx_backend_influxdb          sh
```

OpenTSDB 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 OpenTSDB](#)规则引擎中创建保存数据到 **OpenTSDB**

配置 OpenTSDB 消息存储

配置文件: **etc/plugins/emqx_backend_opentsdb.conf**:

```
1 ## OpenTSDB 服务地址
2 backend.opentsdb.pool1.server = 127.0.0.1:4242
3
4 ## OpenTSDB 连接池大小
5 backend.opentsdb.pool1.pool_size = 8
6
7 ## 是否返回 summary info
8 ##
9 ## Value: true | false
10 backend.opentsdb.pool1.summary = true
11
12 ## 是否返回 detailed info
13 ##
14 ## Value: true | false
15 backend.opentsdb.pool1.details = false
16
17 ## 是否同步写入
18 ##
19 ## Value: true | false
20 backend.opentsdb.pool1.sync = false
21
22 ## 同步写入超时时间, 单位毫秒
23 ##
24 ## Value: Duration
25 ##
26 ## Default: 0
27 backend.opentsdb.pool1.sync_timeout = 0
28
29 ## 最大批量写条数
30 ##
31 ## Value: Number >= 0
32 ## Default: 20
33 backend.opentsdb.pool1.max_batch_size = 20
34
35 ## 存储 PUBLISH 消息
36 backend.opentsdb.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}
```

OpenTSDB Backend 消息存储规则参数:

Option	Description
topic	配置哪些主题下的消息需要执行 hook
action	配置 hook 具体动作, function 为 Backend 提供的内置函数, 实现通用功能
pool	Pool Name , 实现连接多个 OpenTSDB Server 功能

示例:

```
1 ## 存储主题为 "sensor/#" 的 PUBLISH 消息
2 backend.influxdb.hook.message.publish.1 = {"topic": "sensor/#", "action": {"function": "on_m
3   essage_publish"}, "pool": "pool1"}
4
5 ## 存储主题为 "stat/#" 的 PUBLISH 消息
6 backend.influxdb.hook.message.publish.2 = {"topic": "stat/#", "action": {"function": "on_mes
7   sage_publish"}, "pool": "pool1"}
```

OpenTSDB Backend 支持 **Hook** 与 相应内置函数列表:

Hook	Function list
message.publish	on_message_publish

由于 **MQTT Message** 无法直接写入 **OpenTSDB**, **OpenTSDB Backend** 提供了 **emqx_backend_opentsdb.tpl** 模板文件将 **MQTT Message** 转换为可写入 **OpenTSDB** 的 **DataPoint**。

模板文件采用 **Json** 格式, 组成部分:

- **key** - **MQTT Topic**, 字符串, 支持通配符主题
- **value** - **Template**, **Json** 对象, 用于将 **MQTT Message** 转换成 **OpenTSDB** 的 **DataPoint**。

你可以为不同 **Topic** 定义不同的 **Template**, 也可以为同一个 **Topic** 定义多个 **Template**, 类似:

```
1 {
2     <Topic 1>: <Template 1>,
3     <Topic 2>: <Template 2>
4 }
```

Template 格式如下:

```
1 {
2     "measurement": <Measurement>,
3     "tags": {
4         <Tag Key>: <Tag Value>
5     },
6     "value": <Value>,
7     "timestamp": <Timestamp>
8 }
```

measurement 与 **value** 为必选项, **tags** 与 **timestamp** 为可选项。

所有的值 (例如 **<Measurement>**) 你都可以直接在 **Template** 中配置为一个固定值, 它支持的数据类型依赖于你定义的数据表。当然更符合实际情况的是, 你可以通过我们提供的占位符来获取 **MQTT** 消息中的数据。

目前我们支持的占位符如下:

Placeholder	Description
\$id	MQTT 消息 UUID, 由 EMQX 分配
\$clientid	客户端使用的 Client ID
\$username	客户端使用的 Username
\$peerhost	客户端 IP
\$qos	MQTT 消息的 QoS
\$topic	MQTT 消息主题
\$payload	MQTT 消息载荷, 必须为合法的 Json
\$<Number>	必须配合 \$payload 使用, 用于从 Json Array 中获取数据
\$timestamp	EMQX 准备转发消息时设置的时间戳, 精度: 毫秒

\$payload 与 \$<Number>:

你可以直接使用 `$payload` 取得完整的消息载荷, 也可以通过 `["$payload", <Key>, ...]` 取得消息载荷内部的数据。

例如 `payload` 为 `{"data": {"temperature": 23.9}}`, 你可以通过占位符 `["$payload", "data", "temperature"]` 来获取其中的 `23.9`。

考虑到 **Json** 还有数组这一数据类型的情况, 我们引入了 `$0` 与 `$<pos_integer>`, `$0` 表示获取数组内所有元素, `$<pos_integer>` 表示获取数组内第 `<pos_integer>` 个元素。

一个简单例子, `[$payload, "$0", "temp"]` 将从 `[{"temp": 20}, {"temp": 21}]` 中取得 `[20, 21]`, 而 `[$payload, "$1", "temp"]` 将只取得 `20`。

值得注意的是, 当你使用 `$0` 时, 我们希望你取得的数据个数都是相等的。因为我们需要将这些数组转换为多条记录写入 **OpenTSDB**, 而当你一个字段取得了 **3** 份数据, 另一个字段却取得了 **2** 份数据, 我们将无从判断应当怎样为你组合这些数据。

Example

data/templates 目录下提供了一个示例模板 (**emqx_backend_opentsdb_example.tpl**, 正式使用时请去掉文件名中的 `_example` 后缀) 供用户参考:

```

1  {
2      "sample": {
3          "measurement": "$topic",
4          "tags": {
5              "host": ["$payload", "data", "$0", "host"],
6              "region": ["$payload", "data", "$0", "region"],
7              "qos": "$qos",
8              "clientid": "$clientid"
9          },
10         "value": ["$payload", "data", "$0", "temp"],
11         "timestamp": "$timestamp"
12     }
13 }
```

当 Topic 为 "sample" 的 MQTT Message 拥有以下 Payload 时:

```

1  {
2      "data": [
3          {
4              "temp": 1,
5              "host": "serverA",
6              "region": "hangzhou"
7          },
8          {
9              "temp": 2,
10             "host": "serverB",
11             "region": "ningbo"
12         }
13     ]
14 }
```

Backend 将 MQTT Message 转换为以下数据写入 OpenTSDB:

```

1  [
2      {
3          "measurement": "sample",
4          "tags": {
5              "clientid": "mqttjs_ebcc36079a",
6              "host": "serverA",
7              "qos": "0",
8              "region": "hangzhou",
9          },
10         "value": "1",
11         "timestamp": "1560743513626681000"
12     },
13     {
14         "measurement": "sample",
15         "tags": {
16             "clientid": "mqttjs_ebcc36079a",
17             "host": "serverB",
18             "qos": "0",
19             "region": "ningbo",
20         },
21         "value": "2",
22         "timestamp": "1560743513626681000"
23     }
24 ]
```

启用 OpenTSDB 消息存储:

```

1  ./bin/emqx_ctl plugins load emqx_backend_opentsdb
```

Timescale 消息存储

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[保存数据到 Timescale](#)规则引擎中创建保存数据到 **Timescale**

配置 Timescale 消息存储

etc/plugins/emqx_backend_timescale.conf:

```

1 ## Timescale Server
2 backend.timescale.pool1.server = 127.0.0.1:5432
3 ## Timescale Pool Size
4 backend.timescale.pool1.pool_size = 8
5 ## Timescale Username
6 backend.timescale.pool1.username = postgres
7 ## Timescale Password
8 backend.timescale.pool1.password = password
9 ## Timescale Database
10 backend.timescale.pool1.database = tutorial
11 ## Timescale SSL
12 backend.timescale.pool1.ssl = false
13
14 ## SSL keyfile.
15 ##
16 ## Value: File
17 ## backend.timescale.pool1.keyfile =
18
19 ## SSL certfile.
20 ##
21 ## Value: File
22 ## backend.timescale.pool1.certfile =
23
24 ## SSL cacertfile.
25 ##
26 ## Value: File
27 ## backend.timescale.pool1.cacertfile =
28
29 ## Store Publish Message
30 backend.timescale.hook.message.publish.1 = {"topic": "#", "action": {"function": "on_message_publish"}, "pool": "pool1"}sh
```

Timescale Backend 消息存储规则参数:

Option	Description
topic	配置哪些主题下的消息需要执行 hook
action	配置 hook 具体动作, function 为 Backend 提供的内置函数, 实现通用功能
pool	Pool Name , 实现连接多个 Timescale Server 功能

Example:

```

1 ## Store Publish message with "sensor/#" topic
2 backend.timescale.hook.message.publish.1 = {"topic": "sensor/#", "action": {"function": "on_
3 message_publish"}, "pool": "pool1"}
4
5 ## Store Publish message with "stat/#" topic
6 backend.timescale.hook.message.publish.2 = {"topic": "stat/#", "action": {"function": "on_me
7 ssage_publish"}, "pool": "pool1"}sh

```

Timescale Backend 支持 **Hook** 与 相应内置函数列表:

Hook	Function list
message.publish	on_message_publish

Timescale Backend 提供 **emqx_backend_timescale.tpl** 模板文件，用于从不同主题的 **MQTT Message** 中提取数据以写入 **Timescale**。

模板文件采用 **Json** 格式，组成部分:

- `key` - **MQTT Topic**, 字符串, 支持通配符主题
- `value` - **Template, Json** 对象, 用于将 **MQTT Message** 转换成 `measurement,tag_key=tag_value,...` `field_key=field_value,... timestamp` 的形式以写入 **InfluxDB**。

你可以为不同 **Topic** 定义不同的 **Template**, 也可以为同一个 **Topic** 定义多个 **Template**, 类似:

```

1 {
2     <Topic 1>: <Template 1>,
3     <Topic 2>: <Template 2>
4 }sh

```

Template 格式如下:

```

1 {
2     "name": <Name of template>,
3     "sql": <SQL INSERT INTO>,
4     "param_keys": <Param Keys>
5 }sh

```

`name` , `sql` 和 `param_keys` 都是必选项。

`name` 可以是任意的字符串，确保没有重复即可。

`sql` 为 **Timescale** 可用的 **SQL INSERT INTO** 语句，例如: `insert into sensor_data(time, location, temperature, humidity) values (NOW(), $1, $2, $3)`。

`param_keys` 是一个数组，它的第一个元素对应 `sql` 中出现的 `$1`，并以此类推。

数组中任意元素都可以是一个固定值，它支持的数据类型依赖于你定义的数据表。当然更符合实际情况的是，你可以通过我们提供的占位符来获取 **MQTT** 消息中的数据。

目前我们支持的占位符如下:

Placeholder	Description
\$id	MQTT 消息 UUID, 由 EMQX 分配
\$clientid	客户端使用的 Client ID
\$username	客户端使用的 Username
\$peerhost	客户端 IP
\$qos	MQTT 消息的 QoS
\$topic	MQTT 消息主题
\$payload	MQTT 消息载荷, 必须为合法的 Json
\$<Number>	必须配合 \$payload 使用, 用于从 Json Array 中获取数据
\$timestamp	EMQX 准备转发消息时设置的时间戳, 精度: 毫秒

\$payload 与 \$<Number>:

你可以直接使用 `$payload` 取得完整的消息载荷, 也可以通过 `["$payload", <Key>, ...]` 取得消息载荷内部的数据。

例如 `payload` 为 `{"data": {"temperature": 23.9}}`, 你可以通过占位符 `["$payload", "data", "temperature"]` 来获取其中的 `23.9`。

考虑到 Json 还有数组这一数据类型的情况, 我们引入了 `$0` 与 `$<pos_integer>`, `$0` 表示获取数组内所有元素, `$<pos_integer>` 表示获取数组内第 `<pos_integer>` 个元素。

一个简单例子, `["$payload", "$0", "temp"]` 将从 `[{"temp": 20}, {"temp": 21}]` 中取得 `[20, 21]`, 而 `["$payload", "$1", "temp"]` 将只取得 `20`。

值得注意的是, 当你使用 `$0` 时, 我们希望你取得的数据个数都是相等的。因为我们需要将这些数组转换为多条记录写入 **Timescale**, 而当你一个字段取得了 3 份数据, 另一个字段却取得了 2 份数据, 我们将无从判断应当怎样为你组合这些数据。

Example

`data/templates` 目录下提供了一个示例模板 (`emqx_backend_timescale_example.tpl`, 正式使用时请去掉文件名中的 `_example` 后缀) 供用户参考:

```

1  {
2      "sensor_data": {
3          "name": "insert_sensor_data",
4          "sql": "insert into sensor_data(time, location, temperature, humidity) values (NOW(), $1,
5 $2, $3)",
6          "param_keys": [
7              ["$payload", "data", "$0", "location"],
8              ["$payload", "data", "$0", "temperature"],
9              ["$payload", "data", "$0", "humidity"]
10         ]
11     },
12     "sensor_data2/#": {
13         "name": "insert_sensor_data2",
14         "sql": "insert into sensor_data(time, location, temperature, humidity) values (NOW(), $1,
15 $2, $3)",
16         "param_keys": [
17             ["$payload", "location"],
18             ["$payload", "temperature"],
19             ["$payload", "humidity"]
20         ]
21     },
22     "easy_data": {
23         "name": "insert_easy_data",
24         "sql": "insert into easy_data(time, data) values (NOW(), $1)",
25         "param_keys": [
26             "$payload"
27         ]
28     }
29 }

```

当 Topic 为 "sensor_data" 的 MQTT Message 拥有以下 Payload 时:

```

1  {
2      "data": [
3          {
4              "location": "bedroom",
5              "temperature": 21.3,
6              "humidity": 40.3
7          },
8          {
9              "location": "bathroom",
10             "temperature": 22.3,
11             "humidity": 61.8
12         },
13         {
14             "location": "kitchen",
15             "temperature": 29.5,
16             "humidity": 58.7
17         }
18     ]
19 }

```

["\$payload", "data", "\$0", "location"] 会先获取 MQTT Message 的 Payload, 如果 Payload 为 json 格式, 则继续尝试读取 data。data 的值是数组, 这里我们用到了 "\$0" 表示获取数组中所有的元素, 因此 **["\$payload", "data", "\$0", "location"]** 将帮我们获得 **["bedroom", "bathroom", "kitchen"]**。相应的, 如果将 "\$0" 替换为 "\$1", 将只获得

["bedroom"]。相应的，如果将

那么在这个场景中，我们将得到以下 **SQL** 语句：

```
1 insert into sensor_data(time, location, temperature, humidity) values (NOW(), 'bedroom', 21.  
2   .3, 40.3)  
3 insert into sensor_data(time, location, temperature, humidity) values (NOW(), 'bathroom', 22  
.3, 61.8)  
insert into sensor_data(time, location, temperature, humidity) values (NOW(), 'kitchen', 29.  
5, 58.7)
```

最终 **Timescale Backend** 执行这些 **SQL** 语句将数据写入 **Timescale**。

消息桥接

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

桥接是一种连接多个 **EMQX** 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 **EMQX**，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 **EMQX** 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。

桥接插件列表

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

提示

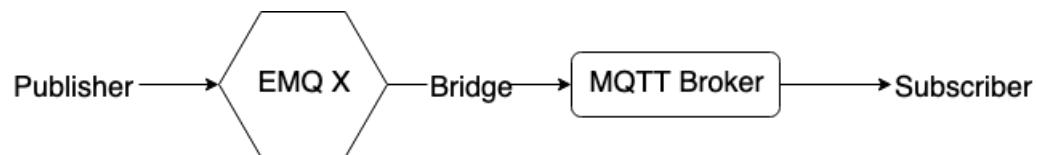
推荐使用 [规则引擎](#) 以实现更灵活的桥接功能。

MQTT 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[MQTT 桥接](#)规则引擎中创建 **MQTT 桥接**

EMQX 桥接转发 **MQTT** 消息到 **MQTT Broker**，支持桥接至常见 **MQTT** 云服务：



mqtt bridge 桥接插件配置文件: `etc/plugins/emqx_bridge_mqtt.conf`。

配置 MQTT 桥接的 Broker 地址

sh

```

1 ## 桥接地址： 使用节点名则用于 rpc 桥接，使用 host:port 用于 mqtt 连接
2 bridge.mqtt.aws.address = 127.0.0.1:1883
3
4 ## 桥接的协议版本
5 ## 枚举值: mqttv3 | mqttv4 | mqttv5
6 bridge.mqtt.aws.proto_ver = mqttv4
7
8 ## mqtt 连接是否启用桥接模式
9 bridge.mqtt.aws.bridge_mode = true
10
11 ## mqtt 客户端的 client_id
12 bridge.mqtt.aws.client_id = bridge_aws
13
14 ## mqtt 客户端的 clean_start 字段
15 ## 注：有些 MQTT Broker 需要将 clean_start 值设成 `true`
16 bridge.mqtt.aws.clean_start = true
17
18 ## mqtt 客户端的 username 字段
19 bridge.mqtt.aws.username = user
20
21 ## mqtt 客户端的 password 字段
22 bridge.mqtt.aws.password = passwd
23
24 ## mqtt 客户端是否使用 ssl 来连接远程服务器
25 bridge.mqtt.aws.ssl = off
26
27 ## 客户端 SSL 连接的 CA 证书 (PEM格式)
28 bridge.mqtt.aws.cacertfile = etc/certs/cacert.pem
29
30 ## 客户端 SSL 连接的 SSL 证书
31 bridge.mqtt.aws.certfile = etc/certs/client-cert.pem
32
33 ## 客户端 SSL 连接的密钥文件
34 bridge.mqtt.aws.keyfile = etc/certs/client-key.pem
35
36 ## SSL 加密算法
37 bridge.mqtt.aws.ciphers = ECDHE-ECDSA-AES256-GCM-SHA384,ECDHE-RSA-AES256-GCM-SHA384
38
39 ## TLS PSK 的加密算法
40 ## 注意 'listener.ssl.external.ciphers' 和 'listener.ssl.external.psk_ciphers' 不能同时配置
41 ##
42 ## See 'https://tools.ietf.org/html/rfc4279#section-2'.
43 bridge.mqtt.aws.psk_ciphers = PSK-AES128-CBC-SHA,PSK-AES256-CBC-SHA,PSK-3DES-EDE-CBC-SHA,PSK-R
44 C4-SHA
45
46 ## 客户端的心跳间隔
47 bridge.mqtt.aws.keepalive = 60s
48
49 ## 支持的 TLS 版本
50 bridge.mqtt.aws.tls_versions = tlsv1.2,tlsv1.1,tlsv1

```

配置 MQTT 桥接转发和订阅主题

```

1 ## 桥接的 mountpoint(挂载点)
2 bridge.mqtt.aws.mountpoint = bridge/aws/${node}/
3
4 ## 转发消息的主题
5 bridge.mqtt.aws.forwards = topic1/#,topic2/# 
6
7 ## 用于桥接的订阅主题
8 bridge.mqtt.aws.subscription.1.topic = cmd/topic1
9
10 ## 用于桥接的订阅 qos
11 bridge.mqtt.aws.subscription.1.qos = 1
12
13 ## 用于桥接的订阅主题
14 bridge.mqtt.aws.subscription.2.topic = cmd/topic2
15
16 ## 用于桥接的订阅 qos
17 bridge.mqtt.aws.subscription.2.qos = 1

```

sh

MQTT 桥接转发和订阅主题说明

挂载点 **Mountpoint: mountpoint** 用于在转发消息时加上主题前缀，该配置选项须配合 **forwards** 使用，转发主题为 **sensor1/hello** 的消息，到达远程节点时主题为 **bridge/aws/emqx1@192.168.1.1/sensor1/hello**。

转发主题 **Forwards:** 转发到本地 EMQX 指定 **forwards** 主题上的消息都会被转发到远程 MQTT Broker 上。

订阅主题 **Subscription:** 本地 EMQX 通过订阅远程 MQTT Broker 的主题来将远程 MQTT Broker 上的消息同步到本地。

启用 bridge_mqtt 桥接插件

```
1 ./bin/emqx_ctl plugins load emqx_bridge_mqtt
```

sh

桥接 CLI 命令

```

1 $ cd emqx1/ && ./bin/emqx_ctl bridges
2 bridges list                                # List bridges
3 bridges start <Name>                         # Start a bridge
4 bridges stop <Name>                           # Stop a bridge
5 bridges forwards <Name>                      # Show a bridge forward topic
6 bridges add-forward <Name> <Topic>          # Add bridge forward topic
7 bridges del-forward <Name> <Topic>          # Delete bridge forward topic
8 bridges subscriptions <Name>                 # Show a bridge subscriptions topic
9 bridges add-subscription <Name> <Topic> <Qos> # Add bridge subscriptions topic

```

sh

列出全部 bridge 状态

```
1 $ ./bin/emqx_ctl bridges list  
2 name: emqx      status: Stopped
```

sh

启动指定 **bridge**

```
1 $ ./bin/emqx_ctl bridges start emqx  
2 Start bridge successfully.
```

sh

停止指定 **bridge**

```
1 $ ./bin/emqx_ctl bridges stop emqx  
2 Stop bridge successfully.
```

sh

列出指定 **bridge** 的转发主题

```
1 $ ./bin/emqx_ctl bridges forwards emqx  
2 topic: topic1/#  
3 topic: topic2/#
```

sh

添加指定 **bridge** 的转发主题

```
1 $ ./bin/emqx_ctl bridges add-forward emqx topic3/#  
2 Add-forward topic successfully.
```

sh

删除指定 **bridge** 的转发主题

```
1 $ ./bin/emqx_ctl bridges del-forward emqx topic3/#  
2 Del-forward topic successfully.
```

sh

列出指定 **bridge** 的订阅

```
1 $ ./bin/emqx_ctl bridges subscriptions emqx  
2 topic: cmd/topic1, qos: 1  
3 topic: cmd/topic2, qos: 1
```

sh

添加指定 **bridge** 的订阅主题

```
1 $ ./bin/emqx_ctl bridges add-subscription emqx cmd/topic3 1  
2 Add-subscription topic successfully.
```

sh

删除指定 **bridge** 的订阅主题

```
1 $ ./bin/emqx_ctl bridges del-subscription emqx cmd/topic3  
2 Del-subscription topic successfully.
```

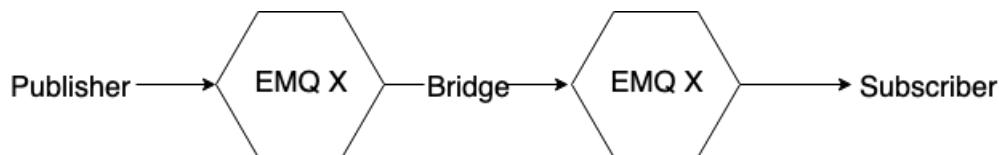
sh

RPC 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[EMQX 桥接](#)规则引擎中创建 **EMQX 桥接**

EMQX 桥接转发 MQTT 消息到远程 EMQX:



rpc bridge 桥接插件配置文件: `etc/plugins/emqx_bridge_mqtt.conf`

配置 RPC 桥接的 Broker 地址

```
1 bridge.mqtt.emqx.address = emqx2@192.168.1.2
```

sh

配置 MQTT 桥接转发和订阅主题

```
1 ## 桥接的 mountpoint(挂载点)
2 bridge.mqtt.emqx.mountpoint = bridge/emqx1/${node}/
3
4 ## 转发消息的主题
5 bridge.mqtt.emqx.forwards = topic1/#,topic2/#
```

sh

MQTT 桥接转发和订阅主题说明

挂载点 **Mountpoint**: **mountpoint** 用于在转发消息时加上主题前缀，该配置选项须配合 **forwards** 使用，转发主题为 `sensor1/hello` 的消息，到达远程节点时主题为 `bridge/aws/emqx1@192.168.1.1/sensor1/hello`。

转发主题 **Forwards**: 转发到本地 EMQX 指定 **forwards** 主题上的消息都会被转发到远程 MQTT Broker 上。

桥接 CLI 命令

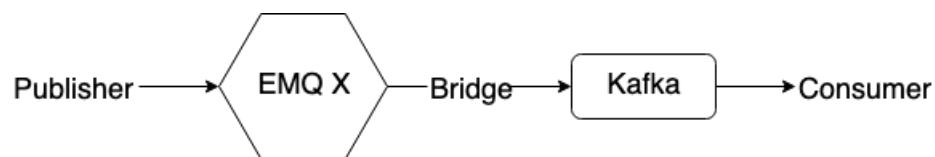
桥接 CLI 的使用方式与 `mqtt bridge` 相同。

Kafka 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[Kafka 桥接](#)

EMQX 桥接转发 **MQTT** 消息到 **Kafka** 集群，**Apache Kafka**是一个快速、高可扩展、高吞吐的分布式日志系统，配合**kafka Stream**，在流式数据处理中非常常用。



Kafka 桥接插件配置文件: `etc/plugins/emqx_bridge_kafka.conf`。

配置 Kafka 集群地址

sh

```

1 ## Kafka 服务器地址
2 ## bridge.kafka.servers = 127.0.0.1:9092,127.0.0.2:9092,127.0.0.3:9092
3 bridge.kafka.servers = 127.0.0.1:9092
4
5 ## Kafka 分区策略。可选值: per_partition | per_broker
6 bridge.kafka.connection_strategy = per_partition
7
8 bridge.kafka.min_metadata_refresh_interval = 5S
9
10 ## Produce 写类型。可选值: sync | async
11 bridge.kafka.produce = sync
12
13 bridge.kafka.produce.sync_timeout = 3S
14
15 ## 指定 replayq 在磁盘上存储消息的基本目录。
16 ## 如果该配置项缺失活着设置为 undefined, replayq 将以使用内存的
17 ## 的方式工作。也就是说, 消息不在磁盘上排队 -- 在这种情况下, send
18 ## 和 send_async API 的调用者负责处理在应用程序、网络或 kafka
19 ## 干扰时可能丢失的消息。
20 ## bridge.kafka.replayq_dir = /tmp/emqx_bridge_kafka/
21
22 ## default=10MB, replayq 分段大小。
23 ## bridge.kafka.producer.replayq_seg_bytes = 10MB
24
25 ## producer required_acks. 可选值: all_isr | leader_only | none.
26 bridge.kafka.producer.required_acks = none
27
28 ## default=10000. leader 在回复 producer 前等待副本的超时时间。
29 bridge.kafka.producer.ack_timeout = 10S
30
31 ## 收集到一次 produce 请求中的最大字节数
32 bridge.kafka.producer.max_batch_bytes = 1024KB
33
34 ## 收集到一次 produce 请求中的最少字节数
35 bridge.kafka.producer.min_batch_bytes = 0
36
37 ## 在没有接收到上次请求的 ack 的情况下, 可以提前发送的 batch 数。
38 ## 如果消息必须严格按照顺序传递, 则必须为0。
39 bridge.kafka.producer.max_send_ahead = 0
40
41 ## 默认为无压缩
42 ## bridge.kafka.producer.compression = no_compression
43
44 ## 默认值为 base64, 可选值: base64 | plain
45 ## bridge.kafka.encode_payload_type = base64
46
47 ## bridge.kafka.sock.buffer = 32KB
48 ## bridge.kafka.sock.redbuf = 32KB
49 bridge.kafka.sock.sndbuf = 1MB
50 ## bridge.kafka.sock.read_packets = 20

```

配置 Kafka 桥接规则

sh

```

1 ## Bridge Kafka Hooks
2 ## ${topic}: the kafka topics to which the messages will be published.
3 ## ${filter}: the mqtt topic (may contain wildcard) on which the action will be performed.
4
5 ## Client Connected Record Hook
6 bridge.kafka.hook.client.connected.1      = {"topic": "client_connected"}
7
8 ## Client Disconnected Record Hook
9 bridge.kafka.hook.client.disconnected.1  = {"topic": "client_disconnected"}
10
11 ## Session Subscribed Record Hook
12 bridge.kafka.hook.session.subscribed.1   = {"filter": "#", "topic": "session_subscribed"}
13
14 ## Session Unsubscribed Record Hook
15 bridge.kafka.hook.session.unsubscribed.1 = {"filter": "#", "topic": "session_unsubscribed"}
16
17 ## Message Publish Record Hook
18 bridge.kafka.hook.message.publish.1       = {"filter": "#", "topic": "message_publish"}
19
20 ## Message Delivered Record Hook
21 bridge.kafka.hook.message.delivered.1     = {"filter": "#", "topic": "message_delivered"}
22
23 ## Message Acked Record Hook
24 bridge.kafka.hook.message.acked.1        = {"filter": "#", "topic": "message_acked"}
25
26 ## More Configures
27 ## partitioner strategy:
28 ## Option: random | roundrobin | first_key_dispatch
29 ## Example: bridge.kafka.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "strategy": "random"}
30
31
32 ## key:
33 ## Option: ${clientid} | ${username}
34 ## Example: bridge.kafka.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "key": "${clientid}"}
35
36
37 ## format:
38 ## Option: json | json
39 ## Example: bridge.kafka.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "format": "json"}

```

Kafka 桥接规则说明

事件	说明
bridge.kafka.hook.client.connected.1	客户端登录
bridge.kafka.hook.client.disconnected.1	客户端退出
bridge.kafka.hook.session.subscribed.1	订阅主题
bridge.kafka.hook.session.unsubscribed.1	取消订阅主题
bridge.kafka.hook.message.publish.1	发布消息
bridge.kafka.hook.message.delivered.1	delivered 消息
bridge.kafka.hook.message.acked.1	ACK 消息

客户端上下线事件转发 Kafka

设备上线 EMQX 转发上线事件消息到 Kafka:

```

1  topic = "client_connected",
2  value = {
3      "client_id": ${clientid},
4      "username": ${username},
5      "node": ${node},
6      "ts": ${ts}
7 }
```

sh

设备下线 EMQX 转发下线事件消息到 Kafka:

```

1  topic = "client_disconnected",
2  value = {
3      "client_id": ${clientid},
4      "username": ${username},
5      "reason": ${reason},
6      "node": ${node},
7      "ts": ${ts}
8 }
```

sh

客户端订阅主题事件转发 Kafka

```

1  topic = session_subscribed
2
3  value = {
4      "client_id": ${clientid},
5      "topic": ${topic},
6      "qos": ${qos},
7      "node": ${node},
8      "ts": ${timestamp}
9 }
```

sh

客户端取消订阅主题事件转发 Kafka

```
topic = session_unsubscribed  
value = {  
    "client_id": ${clientId},  
    "topic": ${topic},  
    "qos": ${qos},  
    "node": ${node},  
    "ts": ${timestamp}  
}
```

sh

MQTT 消息转发到 Kafka

```
topic = message_publish  
  
value = {  
    "client_id": ${clientId},  
    "username": ${username},  
    "topic": ${topic},  
    "payload": ${payload},  
    "qos": ${qos},  
    "node": ${node},  
    "ts": ${timestamp}  
}
```

sh

MQTT 消息派发 (Deliver) 事件转发 Kafka

```
topic = message_delivered  
  
value = {  
    "client_id": ${clientId},  
    "username": ${username},  
    "from": ${fromClientId},  
    "topic": ${topic},  
    "payload": ${payload},  
    "qos": ${qos},  
    "node": ${node},  
    "ts": ${timestamp}  
}
```

sh

MQTT 消息确认 (Ack) 事件转发 Kafka

```

1   topic = message_acked
2
3   value = {
4     "client_id": ${clientId},
5     "username": ${username},
6     "from": ${fromClientId},
7     "topic": ${topic},
8     "payload": ${payload},
9     "qos": ${qos},
10    "node": ${node},
11    "ts": ${timestamp}
12  }

```

sh

Kafka 消费示例

Kafka 读取 MQTT 客户端上下线事件消息:

```

1 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic client_connected --from-b
2 eginning
3
4 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic client_disconnected --fr
5 m-beginning

```

sh

Kafka 读取 MQTT 主题订阅事件消息:

```

1 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic session_subscribed --from-
2 -beginning
3
4 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic session_unsubscribed --fr
5 om-beginning

```

sh

Kafka 读取 MQTT 发布消息:

```

1 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic message_publish --from-be
2 ginning

```

sh

Kafka 读取 MQTT 消息发布 (Deliver)、确认 (Ack) 事件:

```

1 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic message_delivered --from-
2 beginning
3
4 kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic message_acked --from-begi
5 nning

```

sh

提示

默认 **payload** 被 **base64** 编码，可通过修改配置 **bridge.kafka.encode_payload_type** 指定 **payload** 数据格

式。

启用 Kafka 桥接插件

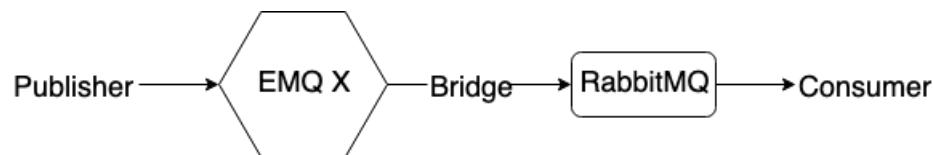
```
1 ./bin/emqx_ctl plugins load emqx_bridge_kafka      sh
```

RabbitMQ 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[RabbitMQ 桥接](#)规则引擎中创建 RabbitMQ 桥接

EMQX 桥接转发 **MQTT** 消息到 **RabbitMQ** 集群：



RabbitMQ 桥接插件配置文件: `etc/plugins/emqx_bridge_rabbit.conf`。

配置 RabbitMQ 桥接地址

```

1 ## RabbitMQ 的服务器地址
2 bridge.rabbit.1.server = 127.0.0.1:5672
3
4 ## RabbitMQ 的连接池大小
5 bridge.rabbit.1.pool_size = 4
6
7 ## RabbitMQ 的用户名
8 bridge.rabbit.1.username = guest
9
10 ## RabbitMQ 的密码
11 bridge.rabbit.1.password = guest
12
13 ## RabbitMQ 的虚拟 Host
14 bridge.rabbit.1.virtual_host =
15
16 ## RabbitMQ 的心跳间隔
17 bridge.rabbit.1.heartbeat = 0
18
19 # bridge.rabbit.2.server = 127.0.0.1:5672
20
21 # bridge.rabbit.2.pool_size = 8
22
23 # bridge.rabbit.2.username = guest
24
25 # bridge.rabbit.2.password = guest
26
27 # bridge.rabbit.2.virtual_host =
28
29 # bridge.rabbit.2.heartbeat = 0
  
```

配置 RabbitMQ 桥接规则

```

1 ## Bridge Hooks
2 bridge.rabbit.hook.client.subscribe.1 = {"action": "on_client_subscribe", "rabbit": 1, "exchange": "direct:emq.subscription"}
3
4
5 bridge.rabbit.hook.client.unsubscribe.1 = {"action": "on_client_unsubscribe", "rabbit": 1, "exchange": "direct:emq.unsubscription"}
6
7
8 bridge.rabbit.hook.message.publish.1 = {"topic": "$SYS/#", "action": "on_message_publish", "rabbit": 1, "exchange": "topic:emq.$sys"}
9
10 bridge.rabbit.hook.message.publish.2 = {"topic": "#", "action": "on_message_publish", "rabbit": 1, "exchange": "topic:emq.pub"}
11
12 bridge.rabbit.hook.message.acked.1 = {"topic": "#", "action": "on_message_acked", "rabbit": 1, "exchange": "topic:emq.acked"}
13

```

客户端订阅主题事件转发 RabbitMQ

```

1 routing_key = subscribe
2 exchange = emq.subscription
3 headers = [{<<"x-emq-client-id">>, binary, ClientId}]
4 payload = jsx:encode([{Topic, proplists:get_value(qos, Opts)} || {Topic, Opts} <- TopicTable])

```

客户端取消订阅事件转发 RabbitMQ

```

1 routing_key = unsubscribe
2 exchange = emq.unsubscription
3 headers = [{<<"x-emq-client-id">>, binary, ClientId}]
4 payload = jsx:encode([Topic || {Topic, _Opts} <- TopicTable]),

```

MQTT 消息转发 RabbitMQ

```

1 routing_key = binary:replace(binary:replace(Topic, <<"/">>, <<".">>, [global]), << "+">>, << "*">>, [global])
2
3 exchange = emq.$sys | emq.pub
4 headers = [{<<"x-emq-publish-qos">>, byte, Qos},
5             {<<"x-emq-client-id">>, binary, pub_from(From)},
6             {<<"x-emq-publish-msgid">>, binary, emqx_base62:encode(Id)},
7             {<<"x-emqx-topic">>, binary, Topic}]
8
9 payload = Payload

```

MQTT 消息确认 (Ack) 事件转发 RabbitMQ

```

1  routing_key = puback
2  exchange = emq.acked
3  headers = [{<<"x-emq-msg-acked">>}, binary, ClientId}],
4  payload = emqx_base62:encode(Id)

```

sh

RabbitMQ 订阅消费 MQTT 消息示例

Python RabbitMQ消费者代码示例:

```

1  #!/usr/bin/env python
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
6  channel = connection.channel()
7
8  channel.exchange_declare(exchange='direct:emq.subscription', exchange_type='direct')
9
10 result = channel.queue_declare(exclusive=True)
11 queue_name = result.method.queue
12
13 channel.queue_bind(exchange='direct:emq.subscription', queue=queue_name, routing_key='subscribe')
14
15 def callback(ch, method, properties, body):
16     print(" [x] %r:%r" % (method.routing_key, body))
17
18 channel.basic_consume(callback, queue=queue_name, no_ack=True)
19
20 channel.start_consuming()

```

py

其他语言 RabbitMQ 客户端代码示例:

<https://github.com/rabbitmq/rabbitmq-tutorials>

启用 RabbitMQ 桥接插件

```
1  ./bin/emqx_ctl plugins load emqx_bridge_rabbit
```

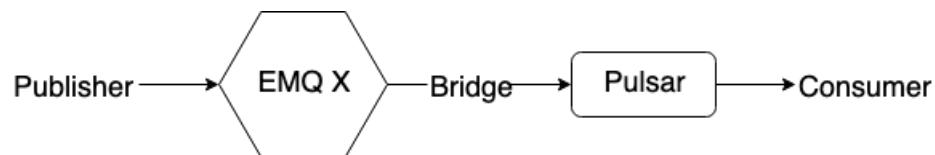
sh

Pulsar 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[Pulsar 桥接](#)规则引擎中创建 Pulsar 桥接

EMQX 桥接转发 MQTT 消息到 Pulsar 集群：



Pulsar 桥接插件配置文件: `etc/plugins/emqx_bridge_pulsar.conf`。

配置 Pulsar 集群地址

```

1 ## Pulsar 服务器集群配置
2 ## bridge.pulsar.servers = 127.0.0.1:6650,127.0.0.2:6650,127.0.0.3:6650
3 bridge.pulsar.servers = 127.0.0.1:6650
4
5 ## 分区生产者是同步/异步模式选择
6 bridge.pulsar.produce = sync
7
8 ## 生产者同步模式下的超时时间
9 ## bridge.pulsar.produce.sync_timeout = 3s
10
11 ## 生产者 batch 的消息数量
12 ## bridge.pulsar.producer.batch_size = 1000
13
14 ## 默认情况下不为生产者启用压缩选项
15 ## bridge.pulsar.producer.compression = no_compression
16
17 ## 采用 base64 编码或不编码
18 ## bridge.pulsar.encode_payload_type = base64
19
20 ## bridge.pulsar.sock.buffer = 32KB
21 ## bridge.pulsar.sock.redbuf = 32KB
22 bridge.pulsar.sock.sndbuf = 1MB
23 ## bridge.pulsar.sock.read_packets = 20
  
```

配置 Pulsar 桥接规则

sh

```

1 ## Bridge Pulsar Hooks
2 ## ${topic}: the pulsar topics to which the messages will be published.
3 ## ${filter}: the mqtt topic (may contain wildcard) on which the action will be performed .
4
5 ## Client Connected Record Hook
6 bridge.pulsar.hook.client.connected.1      = {"topic": "client_connected"}
7
8 ## Client Disconnected Record Hook
9 bridge.pulsar.hook.client.disconnected.1  = {"topic": "client_disconnected"}
10
11 ## Session Subscribed Record Hook
12 bridge.pulsar.hook.session.subscribed.1   = {"filter": "#", "topic": "session_subscribed"}
13
14 ## Session Unsubscribed Record Hook
15 bridge.pulsar.hook.session.unsubscribed.1 = {"filter": "#", "topic": "session_unsubscribed"}
16 }
17
18 ## Message Publish Record Hook
19 bridge.pulsar.hook.message.publish.1       = {"filter": "#", "topic": "message_publish"}
20
21 ## Message Delivered Record Hook
22 bridge.pulsar.hook.message.delivered.1    = {"filter": "#", "topic": "message_delivered"}
23
24 ## Message Acked Record Hook
25 bridge.pulsar.hook.message.acked.1        = {"filter": "#", "topic": "message_acked"}
26
27 ## More Configures
28 ## partitioner strategy:
29 ## Option: random | roundrobin | first_key_dispatch
30 ## Example: bridge.pulsar.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "strategy": "random"}
31
32
33 ## key:
34 ## Option: ${clientid} | ${username}
35 ## Example: bridge.pulsar.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "key": "${clientid}"}
36
37
## format:
## Option: json | json
## Example: bridge.pulsar.hook.message.publish.1 = {"filter": "#", "topic": "message_publish", "format": "json"}

```

Pulsar 桥接规则说明

事件	说明
bridge.pulsar.hook.client.connected.1	客户端登录
bridge.pulsar.hook.client.disconnected.1	客户端退出
bridge.pulsar.hook.session.subscribed.1	订阅主题
bridge.pulsar.hook.session.unsubscribed.1	取消订阅主题
bridge.pulsar.hook.message.publish.1	发布消息
bridge.pulsar.hook.message.delivered.1	delivered 消息
bridge.pulsar.hook.message.acked.1	ACK 消息

客户端上下线事件转发 Pulsar

设备上线 EMQX 转发上线事件消息到 Pulsar:

```

1 topic = "client_connected",
2 value = {
3     "client_id": ${clientid},
4     "username": ${username},
5     "node": ${node},
6     "ts": ${ts}
7 }
```

sh

设备下线 EMQX 转发下线事件消息到 Pulsar:

```

1 topic = "client_disconnected",
2 value = {
3     "client_id": ${clientid},
4     "username": ${username},
5     "reason": ${reason},
6     "node": ${node},
7     "ts": ${ts}
8 }
```

sh

客户端订阅主题事件转发 Pulsar

```

1 topic = session_subscribed
2
3 value = {
4     "client_id": ${clientid},
5     "topic": ${topic},
6     "qos": ${qos},
7     "node": ${node},
8     "ts": ${timestamp}
9 }
```

sh

客户端取消订阅主题事件转发 Pulsar

```
topic = session_unsubscribed
value = {
    "client_id": ${clientId},
    "topic": ${topic},
    "qos": ${qos},
    "node": ${node},
    "ts": ${timestamp}
}
```

sh

MQTT 消息转发到 Pulsar

```
topic = message_publish
value = {
    "client_id": ${clientId},
    "username": ${username},
    "topic": ${topic},
    "payload": ${payload},
    "qos": ${qos},
    "node": ${node},
    "ts": ${timestamp}
}
```

sh

MQTT 消息派发 (Deliver) 事件转发 Pulsar

```
topic = message_delivered
value = {
    "client_id": ${clientId},
    "username": ${username},
    "from": ${fromClientId},
    "topic": ${topic},
    "payload": ${payload},
    "qos": ${qos},
    "node": ${node},
    "ts": ${timestamp}
}
```

sh

MQTT 消息确认 (Ack) 事件转发 Pulsar

```

1   topic = message_acked
2
3   value = {
4       "client_id": ${clientid},
5       "username": ${username},
6       "from": ${fromClientId},
7       "topic": ${topic},
8       "payload": ${payload},
9       "qos": ${qos},
10      "node": ${node},
11      "ts": ${timestamp}
12  }

```

sh

Pulsar 消费示例

Pulsar 读取 **MQTT** 客户端上下线事件消息:

```

1   pulsar-client consume client_connected -s "client_connected" -n 1000
2
3   pulsar-client consume client_disconnected -s "client_disconnected" -n 1000

```

sh

Pulsar 读取 **MQTT** 主题订阅事件消息:

```

1   pulsar-client consume session_subscribed -s "session_subscribed" -n 1000
2
3   pulsar-client consume session_unsubscribed -s "session_unsubscribed" -n 1000

```

sh

Pulsar 读取 **MQTT** 发布消息:

```

1   pulsar-client consume message_publish -s "message_publish" -n 1000

```

sh

Pulsar 读取 **MQTT** 消息发布 (**Deliver**)、确认 (**Ack**)事件:

```

1   pulsar-client consume message_delivered -s "message_delivered" -n 1000
2
3   pulsar-client consume message_acked -s "message_acked" -n 1000

```

sh

提示

默认 **payload** 被 **base64** 编码，可通过修改配置 **bridge.pulsar.encode_payload_type** 指定 **payload** 数据格式。

启用 Pulsar 桥接插件

```

1   ./bin/emqx_ctl plugins load emqx_bridge_pulsar

```

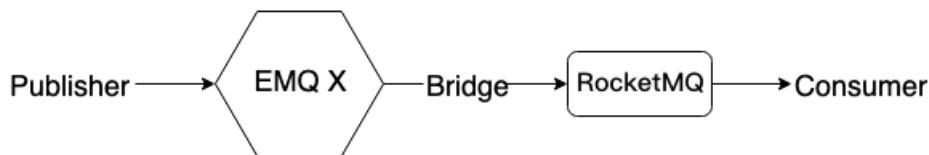
sh

RocketMQ 桥接

提示

EMQX 3.1 版本后推出强大的规则引擎用于替换插件，建议您前往使用[RocketMQ 桥接](#)规则引擎中创建 RocketMQ 桥接

EMQX 桥接转发 **MQTT** 消息到 **RocketMQ** 集群:



RocketMQ 桥接插件配置文件: `etc/plugins/emqx_bridge_rocket.conf`。

配置 RocketMQ 集群地址

```

1 ## RocketMQ 服务器集群配置
2 ## bridge.rocket.servers = 127.0.0.1:9876,127.0.0.2:9876,127.0.0.3:9876
3 bridge.rocket.servers = 127.0.0.1:9876
4
5 bridge.rocket.refresh_topic_route_interval = 5S
6
7 ## 分区生产者是同步/异步模式选择
8 bridge.rocket.produce = sync
9
10 ## 生产者同步模式下的超时时间
11 ## bridge.rocket.produce.sync_timeout = 3s
12
13 ## 生产者 batch 的消息数量
14 ## bridge.rocket.producer.batch_size = 100
15
16 ## 采用 base64 编码或不编码
17 ## bridge.rocket.encode_payload_type = base64
18
19 ## bridge.rocket.sock.buffer = 32KB
20 ## bridge.rocket.sock.recbuf = 32KB
21 bridge.rocket.sock.sndbuf = 1MB
22 ## bridge.rocket.sock.read_packets = 20
  
```

配置 RocketMQ 桥接规则

```

1 ## Bridge RocketMQ Hooks
2 ## ${topic}: the RocketMQ topics to which the messages will be published.
3 ## ${filter}: the mqtt topic (may contain wildcard) on which the action will be performed .
4
5 ## Client Connected Record Hook
6 bridge.rocket.hook.client.connected.1      = {"topic": "ClientConnected"}
7
8 ## Client Disconnected Record Hook
9 bridge.rocket.hook.client.disconnected.1  = {"topic": "ClientDisconnected"}
10
11 ## Session Subscribed Record Hook
12 bridge.rocket.hook.session.subscribed.1   = {"filter": "#", "topic": "SessionSubscribed"}
13
14 ## Session Unsubscribed Record Hook
15 bridge.rocket.hook.session.unsubscribed.1 = {"filter": "#", "topic": "SessionUnsubscribed"}
16
17 ## Message Publish Record Hook
18 bridge.rocket.hook.message.publish.1       = {"filter": "#", "topic": "MessagePublish"}
19
20 ## Message Delivered Record Hook
21 bridge.rocket.hook.message.delivered.1     = {"filter": "#", "topic": "MessageDeliver"}
22
23 ## Message Acked Record Hook
24 bridge.rocket.hook.message.acked.1        = {"filter": "#", "topic": "MessageAcked"}sh
```

RocketMQ 桥接规则说明

事件	说明
bridge.rocket.hook.client.connected.1	客户端登录
bridge.rocket.hook.client.disconnected.1	客户端退出
bridge.rocket.hook.session.subscribed.1	订阅主题
bridge.rocket.hook.session.unsubscribed.1	取消订阅主题
bridge.rocket.hook.message.publish.1	发布消息
bridge.rocket.hook.message.delivered.1	delivered 消息
bridge.rocket.hook.message.acked.1	ACK 消息

客户端上下线事件转发 RocketMQ

设备上线 EMQX 转发上线事件消息到 RocketMQ:

```

1 topic = "ClientConnected",
2 value = {
3     "client_id": ${clientid},
4     "username": ${username},
5     "node": ${node},
6     "ts": ${ts}
7 }sh
```

设备下线 **EMQX** 转发下线事件消息到 **RocketMQ**:

```

1  topic = "ClientDisconnected",
2  value = {
3      "client_id": ${clientid},
4      "username": ${username},
5      "reason": ${reason},
6      "node": ${node},
7      "ts": ${ts}
8  }

```

客户端订阅主题事件转发 **RocketMQ**

```

1  topic = "SessionSubscribed"
2
3  value = {
4      "client_id": ${clientid},
5      "topic": ${topic},
6      "qos": ${qos},
7      "node": ${node},
8      "ts": ${timestamp}
9  }

```

客户端取消订阅主题事件转发 **RocketMQ**

```

1  topic = "SessionUnsubscribed"
2
3  value = {
4      "client_id": ${clientid},
5      "topic": ${topic},
6      "qos": ${qos},
7      "node": ${node},
8      "ts": ${timestamp}
9  }

```

MQTT 消息转发到 **RocketMQ**

```

1  topic = "MessagePublish"
2
3  value = {
4      "client_id": ${clientid},
5      "username": ${username},
6      "topic": ${topic},
7      "payload": ${payload},
8      "qos": ${qos},
9      "node": ${node},
10     "ts": ${timestamp}
11 }

```

MQTT 消息派发 (Deliver) 事件转发 RocketMQ

```

1  topic = "MessageDeliver"
2
3  value = {
4      "client_id": ${clientId},
5      "username": ${username},
6      "from": ${fromClientId},
7      "topic": ${topic},
8      "payload": ${payload},
9      "qos": ${qos},
10     "node": ${node},
11     "ts": ${timestamp}
12 }
```

MQTT 消息确认 (Ack) 事件转发 RocketMQ

```

1  topic = "MessageAcked"
2
3  value = {
4      "client_id": ${clientId},
5      "username": ${username},
6      "from": ${fromClientId},
7      "topic": ${topic},
8      "payload": ${payload},
9      "qos": ${qos},
10     "node": ${node},
11     "ts": ${timestamp}
12 }
```

RocketMQ 消费示例

RocketMQ 读取 MQTT 客户端上下线事件消息:

```

1  bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer ClientConnected
2
3  bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer ClientDisconnected
```

RocketMQ 读取 MQTT 主题订阅事件消息:

```

1  bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer SessionSubscribed
2
3  bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer SessionUnsubscribed
```

RocketMQ 读取 MQTT 发布消息:

```

1  bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer MessagePublish
```

RocketMQ 读取 MQTT 消息发布 (**Deliver**)、确认 (**Ack**) 事件:

```
1 bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer MessageDeliver  
2  
3 bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer MessageAcked
```

sh

提示

默认 **payload** 被 **base64** 编码，可通过修改配置 **bridge.rocket.encode_payload_type** 指定 **payload** 数据格式。

启用 RocketMQ 桥接插件

```
1 ./bin/emqx_ctl plugins load emqx_bridge_rocket
```

sh

设备管理

借助 **EMQX 管理监控 API** 提供的客户端管理及主题订阅、消息发布管理接口以及认证、**ACL**、**WebHook** 等插件，能够快速搭建设备与消息的管理系统，构建联网接入平台。

设备认证

使用 [认证](#) 功能，实现设备连接认证。

管理系统可以直接读写外部认证数据库或通过 **API** 更改 **EMQX** 内置认证数据，实现设备连接认证动态管理。

在线状态与连接历史管理

设备连接、断开时 **EMQX** 可以通过 **Webhook** 插件、[规则引擎](#) 向管理系统 **HTTP API** 发送上下线信息，实现设备在线状态改写、设备连接 / 断开历史记录等功能。

使用断开设备 **API** 可以实现在线设备踢出，会话清除操作。

发布订阅/**ACL**

使用 [发布订阅 ACL](#) 功能，实现设备发布订阅权限管理。

管理系统可以直接读写外部认证数据库，实现设备发布订阅权限动态管理。

代理订阅

管理系统可通过代理订阅功能为在线设备订阅/取消订阅指定主题，在业务 **Topic** 更改后无需重新重新设定设备程序，有较高的灵活性。

HTTP 消息发布

消息发布 **API** 管理系统可以向任意 **Topic** 发布消息而无需使用额外的客户端，实现了 **HTTP-MQTT** 的消息转换。

HTTP 消息发布解耦了用户与设备、管理系统与设备之间的通信，通过间接通信能够降低系统复杂度并进一步提升安全性。

系统调优

EMQX 自 4.2 版本以来，在一台 8 核心、32G 内存的 CentOS 服务器上，MQTT 连接压力测试可达到 130 万。

100 万连接测试所需的 Linux 内核参数，网络协议栈参数，Erlang 虚拟机参数，EMQX 消息服务器参数设置如下：

Linux 操作系统参数

系统全局允许分配的最大文件句柄数：

```
1 # 2 millions system-wide
2 sysctl -w fs.file-max=2097152
3 sysctl -w fs.nr_open=2097152
4 echo 2097152 > /proc/sys/fs/nr_open
```

允许当前会话 / 进程打开文件句柄数：

```
1 ulimit -n 1048576
```

/etc/sysctl.conf

持久化 'fs.file-max' 设置到 /etc/sysctl.conf 文件：

```
1 fs.file-max = 1048576
```

/etc/systemd/system.conf 设置服务最大文件句柄数：

```
1 DefaultLimitNOFILE=1048576
```

/etc/security/limits.conf

/etc/security/limits.conf 持久化设置允许用户 / 进程打开文件句柄数：

```
1 * soft  nofile 1048576
2 * hard  nofile 1048576
```

TCP 协议栈网络参数

并发连接 backlog 设置：

```

1 sysctl -w net.core.somaxconn=32768
2 sysctl -w net.ipv4.tcp_max_syn_backlog=16384
3 sysctl -w net.core.netdev_max_backlog=16384

```

sh

可用知名端口范围:

```

1 sysctl -w net.ipv4.ip_local_port_range='1000 65535'

```

sh

TCP Socket 读写 Buffer 设置:

```

1 sysctl -w net.core.rmem_default=262144
2 sysctl -w net.core.wmem_default=262144
3 sysctl -w net.core.rmem_max=16777216
4 sysctl -w net.core.wmem_max=16777216
5 sysctl -w net.core.optmem_max=16777216
6
7 #sysctl -w net.ipv4.tcp_mem='16777216 16777216 16777216'
8 sysctl -w net.ipv4.tcp_rmem='1024 4096 16777216'
9 sysctl -w net.ipv4.tcp_wmem='1024 4096 16777216'

```

sh

TCP 连接追踪设置:

```

1 sysctl -w net.netfilter.nf_conntrack_max=1000000
2 sysctl -w net.netfilter.nf_conntrack_tcp_timeout_time_wait=30

```

sh

TIME-WAIT Socket 最大数量、回收与重用设置:

```

1 sysctl -w net.ipv4.tcp_max_tw_buckets=1048576
2
3 # 注意: 不建议开启该设置, NAT 模式下可能引起连接 RST
4 # sysctl -w net.ipv4.tcp_tw_recycle=1
5 # sysctl -w net.ipv4.tcp_tw_reuse=1

```

sh

FIN-WAIT-2 Socket 超时设置:

```

1 sysctl -w net.ipv4.tcp_fin_timeout=15

```

sh

Erlang 虚拟机参数

优化设置 Erlang 虚拟机启动参数, 配置文件 `emqx/etc/emqx.conf`:

```

1 ## Erlang Process Limit
2 node.process_limit = 2097152
3
4 ## Sets the maximum number of simultaneously existing ports for this system
5 node.max_ports = 1048576

```

sh

docker 参数调优

通常调优应该在**docker**的主机上做，但是如果一定要从**docker**内部做，可以参考如下例子：

```

1 docker run -d --name emqx -p 18083:18083 -p 1883:1883 -p 4369:4369 \
2   --sysctl fs.file-max=2097152 \
3   --sysctl fs.nr_open=2097152 \
4   --sysctl net.core.somaxconn=32768 \
5   --sysctl net.ipv4.tcp_max_syn_backlog=16384 \
6   --sysctl net.core.netdev_max_backlog=16384 \
7   --sysctl net.ipv4.ip_local_port_range=1000 65535 \
8   --sysctl net.core.rmem_default=262144 \
9   --sysctl net.core.wmem_default=262144 \
10  --sysctl net.core.rmem_max=16777216 \
11  --sysctl net.core.wmem_max=16777216 \
12  --sysctl net.core.optmem_max=16777216 \
13  --sysctl net.ipv4.tcp_rmem=1024 4096 16777216 \
14  --sysctl net.ipv4.tcp_wmem=1024 4096 16777216 \
15  --sysctl net.ipv4.tcp_max_tw_buckets=1048576 \
16  --sysctl net.ipv4.tcp_fin_timeout=15 \
17  emqx/emqx:latest

```

::: 友情提示 不要使用 `--privileged` 或者将系统内核目录挂载到**docker**中进行调优。 :::

EMQX 消息服务器参数

ceptor 池大小，最大允许连接数。`emqx/etc/emqx.conf` `emqx/etc/listeners.conf` }

```

1 ## TCP Listener
2 listener.tcp.external = 0.0.0.0:1883
3 listener.tcp.external.acceptors = 64
4 listener.tcp.external.max_connections = 1024000

```

测试客户端设置

测试客户端服务器在一个接口上，最多只能创建 **65000** 连接：

```

1 sysctl -w net.ipv4.ip_local_port_range="500 65535"
2 echo 1000000 > /proc/sys/fs/nr_open
3 ulimit -n 100000

```

emqtt_bench

并发连接测试工具：http://github.com/emqx/emqtt_bench

生产部署

在开发时我们通常使用压缩包方式以单节点的形式启动服务，生产运行需要一个更加简单稳定的方式。本页主要从部署架构最佳实践讲解如何部署你的 **EMQX** 服务。

提示

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，且需要拿到客户端真实的源 IP 地址与端口，则需打开 **Proxy Protocol** 配置，配置项：[EMQX 监听器 proxy_protocol](#)

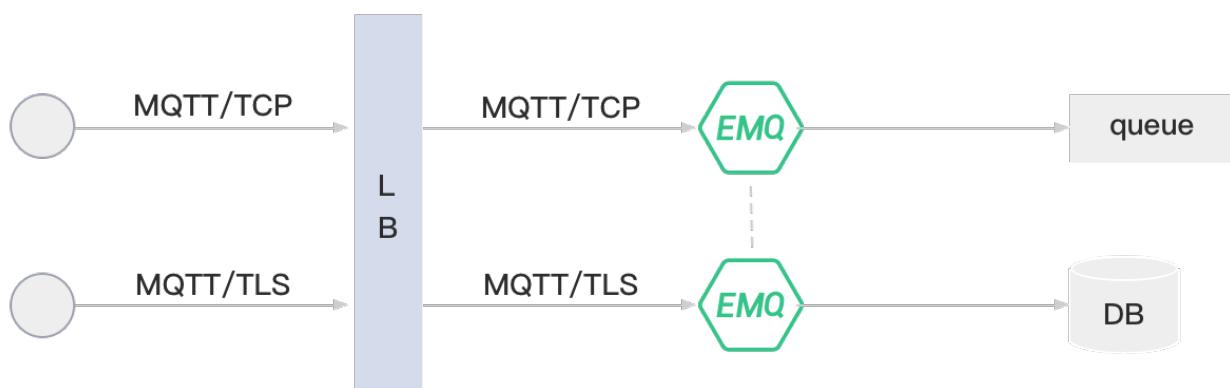
Proxy Protocol 参考：<https://www.haproxy.com/blog/haproxy/proxy-protocol>。

Nginx 使用 Proxy Protocol 参考：<https://docs.nginx.com/nginx/admin-guide/load-balancer/using-proxy-protocol/>

部署架构

EMQX 集群可作为物联网接入服务（**IoT Hub**）部署，目前 **EMQ** 在青云、阿里云、**AWS** 等云服务提供商上均提供开箱即用的免费软件镜像，对于特殊硬件平台和系统版本如树莓派、**Linux ARM**，可使用源码编译安装。

典型部署架构：



LB (负载均衡)

LB (负载均衡器) 负责分发设备的 **MQTT** 连接与消息到 **EMQX** 集群，**LB** 提高 **EMQX** 集群可用性、实现负载平衡以及动态扩容。

部署架构推荐在 **LB** 终结 **SSL** 连接。设备与 **LB** 之间 **TLS** 安全连接，**LB** 与 **EMQX** 之间普通 **TCP** 连接。这种部署模式下 **EMQX** 单集群可轻松支持 **100** 万设备。

公有云厂商 **LB** 产品：

云计算厂商	是否支持 TLS 终结	LB 产品介绍
青云	是	https://docs.qingcloud.com/product/network/loadbalancer/
AWS	是	https://aws.amazon.com/cn/elasticloadbalancing/
阿里云	否	https://www.aliyun.com/product/slb
UCloud	未知	https://ucloud.cn/site/product/ulb.html
QCloud	未知	https://www.qcloud.com/product/clb

私有部署 LB 服务器:

开源 LB	是否支持 TLS 终结	方案介绍
HAProxy	是	https://www.haproxy.com/solutions/load-balancing.html
NGINX	是	https://www.nginx.com/solutions/load-balancing/

提示

国内公有云部署推荐青云 (EMQX 合作伙伴), 国外部署推荐 AWS , 私有部署推荐使用 HAProxy 作为 LB。

EMQX 集群

EMQX 节点集群部署在 LB 之后, 建议部署在 VPC 或私有网络内。公有云厂商青云、AWS、UCloud、QCloud 均支持 VPC 网络。

EMQX 默认开启的 MQTT 服务 TCP 端口:

端口	说明
1883	MQTT 协议端口
8883	MQTT/SSL 端口
8083	MQTT/WebSocket 端口
8084	MQTT/WebSocket/SSL 端口
8081	管理 API 端口
18083	Dashboard 端口

防火墙根据使用的 MQTT 接入方式, 开启上述端口的访问权限。

EMQX 节点集群使用的 TCP 端口:

端口	说明
4369	集群节点发现端口 (EPMD 模式)
4370	集群节点发现端口
5370	集群节点 PRC 通道

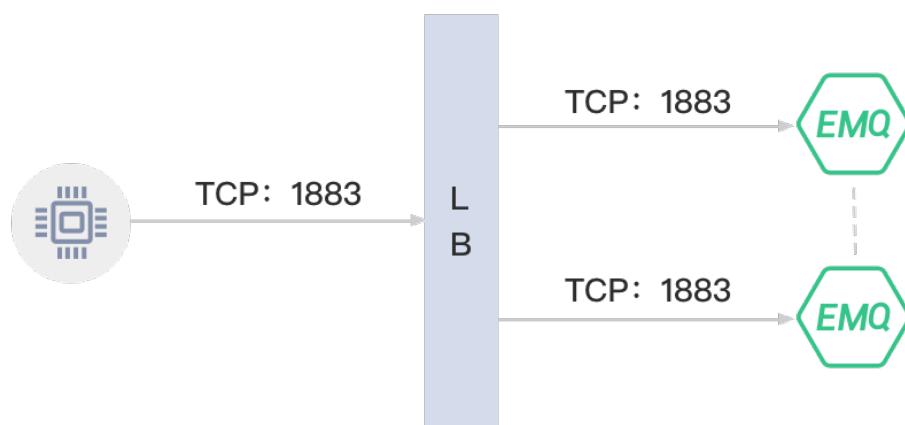
集群节点间如有防护墙, 需开启上述 TCP 端口互访权限。

青云 (QingCloud) 部署

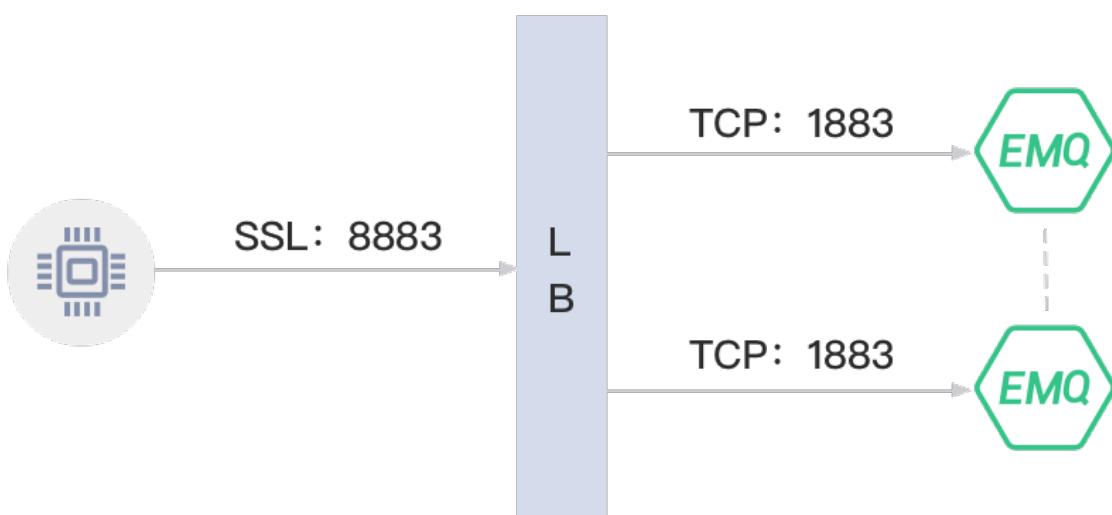
1. 创建 **VPC** 网络。
2. **VPC** 网络内创建 **EMQX** 集群 '私有网络'，例如: **192.168.0.0/24**
3. 私有网络内创建两台 **EMQX** 主机，例如:

节点	IP 地址
emqx1	192.168.0.2
emqx2	192.168.0.3

4. 安装并集群 **EMQX** 主机，具体配置请参考安装集群章节。
5. 创建 **LB** (负载均衡器) 并指定公网 IP 地址。
6. 在 **LB** 上创建 **MQTT TCP** 监听器:



或创建 **SSL** 监听器，并终结 **SSL** 在 **LB** :



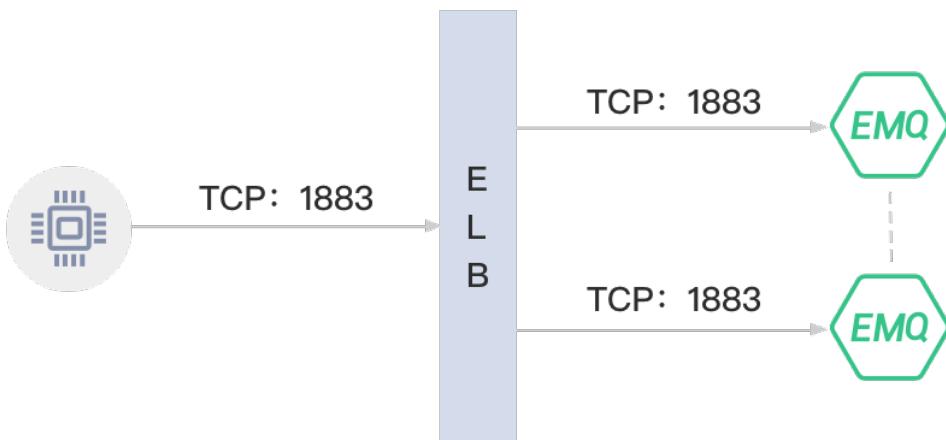
7. **MQTT** 客户端连接 **LB** 公网地址测试。

亚马逊 (AWS) 部署

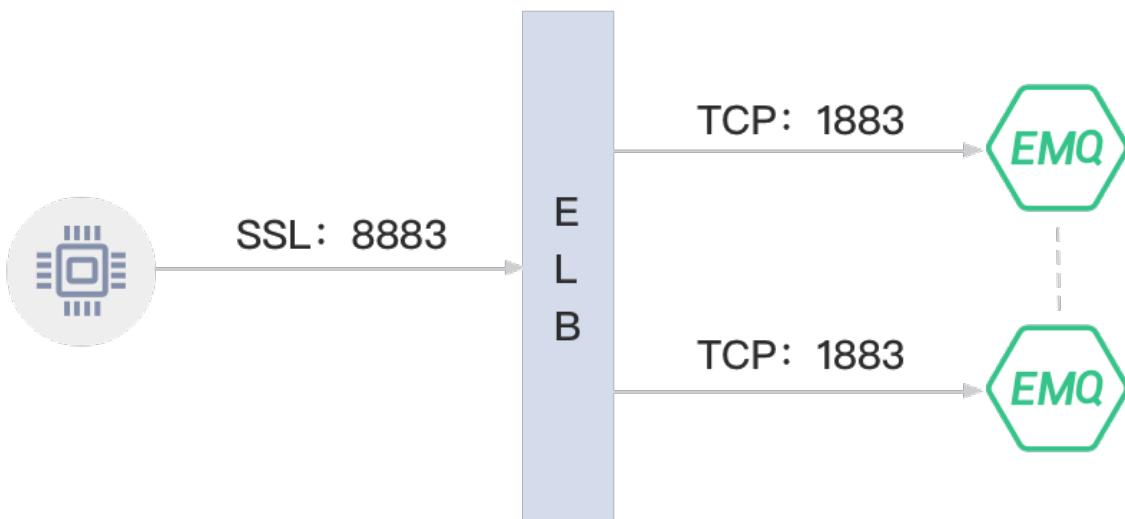
1. 创建 **VPC** 网络。
2. **VPC** 网络内创建 **EMQX** 集群 '私有网络'，例如: **192.168.0.0/24**
3. 私有网络内创建两台 **EMQX** 主机，指定上面创建的 **VPC** 网络，例如:

节点	IP 地址
emqx1	192.168.0.2
emqx2	192.168.0.3

4. 在安全组中，开放 **MQTT** 服务的 **TCP** 端口，比如 **1883, 8883**。
5. 安装并集群 **EMQX** 主机，具体配置请参考安装集群章节。
6. 创建 **ELB (Classic 负载均衡器)**，指定 **VPC** 网络，并指定公网 **IP 地址**。
7. 在 **ELB** 上创建 **MQTT TCP 监听器**:



或创建 **SSL** 监听器，并终结 **SSL** 在 **LB** :



8. **MQTT** 客户端连接 **LB** 公网地址测试。

私有网络部署

EMQX 集群直连

EMQX 集群直接挂 **DNS** 轮询，设备通过域名或者 **IP** 地址列表访问：

1. 部署 **EMQX** 集群
2. **EMQX** 节点防火墙开启外部 **MQTT** 访问端口，例如 **1883, 8883**
3. 设备通过 **IP** 地址列表或域名访问 **EMQX** 集群

提示

产品部署不推荐这种部署方式。

HAProxy 负载均衡

HAProxy 作为 **LB** 部署 **EMQX** 集群，并终结 **SSL** 连接：

1. 创建 **EMQX** 集群节点，例如：

节点	IP 地址
emqx1	192.168.0.2
emqx2	192.168.0.3

2. 配置 **/etc/haproxy/haproxy.cfg**，示例：

```

1  listen mqtt-ssl
2    bind *:8883 ssl crt /etc/ssl/emqx/emq.pem no-sslv3
3    mode tcp
4    maxconn 50000
5    timeout client 600s
6    default_backend emqx_cluster
7
8  backend emqx_cluster
9    mode tcp
10   balance source
11   timeout server 50s
12   timeout check 5000
13   server emqx1 192.168.0.2:1883 check inter 10000 fall 2 rise 5 weight 1
14   server emqx2 192.168.0.3:1883 check inter 10000 fall 2 rise 5 weight 1

```

yaml

Nginx 负载均衡

Nginx 产品作为 **EMQX** 集群 **LB**，并终结 **SSL** 连接：

1. 创建 **EMQX** 节点集群，例如：

节点	IP 地址
emqx1	192.168.0.2
emqx2	192.168.0.3

3. 配置 /etc/nginx/nginx.conf, 示例:

```
sh
1 stream {
2     upstream stream_backend {
3         zone tcp_servers 64k;
4         hash $remote_addr;
5         server 192.168.0.2:1883 max_fails=2 fail_timeout=30s;
6         server 192.168.0.3:1883 max_fails=2 fail_timeout=30s;
7     }
8
9     server {
10        listen 8883 ssl;
11        status_zone tcp_server;
12        proxy_pass stream_backend;
13        proxy_buffer_size 4k;
14        ssl_handshake_timeout 15s;
15        ssl_certificate      /etc/emqx/certs/cert.pem;
16        ssl_certificate_key  /etc/emqx/certs/key.pem;
17    }
18 }
```

Prometheus 监控告警

从 EMQX Enterprise v4.1.0 开始，`emqx_statsd` 更名为 `emqx_prometheus`，相关插件名称、目录均有变更。

EMQX 提供 [emqx_prometheus](#) 插件，用于将系统的监控数据输出到第三方的监控系统中。

以 [Prometheus](#) 为例：

`emqx_prometheus` 支持将数据推送至 **Pushgateway** 中，然后再由 **Promethues Server** 拉取进行存储。

注意：`emqx_prometheus` 不支持 **Prometheus** 的 **Pull** 操作。

配置

`emqx_prometheus` 插件内部会启动一个定时器，使其每间隔一段时间便采集 EMQX 中的监控数据。

`emqx_prometheus` 推送的监控数据包含的具体字段和含义，参见：[Metrics & Stats](#)

配置文件位于 `etc/plugins/emqx_prometheus.conf`，其中：

配置项	类型	可取值	默认值	说明
<code>push.gateway.server</code>	<code>string</code>	-	<code>http://127.0.0.1:9091</code>	Prometheus 的 PushGateway 地址
<code>interval</code>	<code>integer</code>	<code>> 0</code>	<code>5000</code>	推送间隔，单位：毫秒

Grafana 数据模板

`emqx_prometheus` 插件提供了 **Grafana** 的 **Dashboard** 的模板文件。这些模板包含了所有 EMQX 监控数据的展示。用户可直接导入到 **Grafana** 中，进行显示 EMQX 的监控状态的图标。

模板文件位于：[emqx_prometheus/grafana_template](#)。

性能测试

`emqtt_bench` 是基于 **Erlang** 编写的，一个简洁强大的 **MQTT** 协议性能测试工具，如需大规模场景、深度定制化的测试服务推荐使用 **EMQ** 合作伙伴 [XMeter](#) 测试服务。

编译安装

`emqtt_bench` 的运行依赖于 **Erlang/OTP 21.2** 以上版本运行环境，安装过程略过，详情请参考网上各个安装教程。

Erlang 环境安装完成后，下载 `emqtt-bench` 最新代码，并编译：

```
1 git clone https://github.com/emqx/emqtt-bench
2 cd emqtt-bench
3
4 make
```

编译完成后，当前目录下会生成一个名为 `emqtt_bench` 的可执行脚本。执行以下命令，确认其能正常使用：

```
1 ./emqtt_bench
2 Usage: emqtt_bench pub | sub | conn [--help]
```

输出以上内容，则证明 `emqtt_bench` 已正确安装到主机。

使用

`emqtt_bench` 共三个子命令：

1. `pub`：用于创建大量客户端执行发布消息的操作。
2. `sub`：用于创建大量客户端执行订阅主题，并接受消息的操作。
3. `conn`：用于创建大量的连接。

发布

执行 `./emqtt_bench pub --help` 会得到可用的参数输出，此处整理：

参数	简写	可选值	默认值	说明
--host	-h	-	localhost	要连接的 MQTT 服务器地址
--port	-p	-	1883	MQTT 服务端口
--version	-V	3 4 5	5	使用的 MQTT 协议版本
--count	-c	-	200	客户端总数
--startnumber	-n	-	0	客户端数量起始值
--interval	-i	-	10	每间隔多少时间创建一个客户端；单位：毫秒
--interval_of_msg	-I	-	1000	每间隔多少时间发送一个消息
--username	-u	-	无；非必选	客户端用户名
--password	-P	-	无；非必选	客户端密码
--topic	-t	-	无；必选	发布的主题；支持占位符： %c : 表示 ClientId %u : 表示 Username %i : 表示客户端的序列数
--size	-s	-	256	消息 Payload 的大小；单位：字节
--qos	-q	-	0	Qos 等级
--retain	-r	true false	false	消息是否设置 Retain 标志
--keepalive	-k	-	300	客户端心跳时间
--clean	-C	true false	true	是否以清除会话的方式建立连接
--ssl	-S	true false	false	是否启用 SSL
--certfile	-	-	无	客户端 SSL 证书
--keyfile	-	-	无	客户端 SSL 秘钥文件
--ws	-	true false	false	是否以 Websocket 的方式建立连接
--ifaddr	-	-	无	指定客户端连接使用的本地网卡

例如，我们启动 10 个连接，分别每秒向主题 t 发送 100 条 Qos0 消息，其中每个消息体的大小为 16 字节大小：

```
1 ./emqtt_bench pub -t t -h emqx-server -s 16 -q 0 -c 10 -I 10
```

sh

订阅

执行 ./emqtt_bench sub --help 可得到该子命令的所有的可用参数。它们的解释已包含在上表中，此处略过。

例如，我们启动 500 个连接，每个都以 Qos0 订阅 t 主题：

```
1 ./emqtt_bench sub -t t -h emqx-server -c 500
```

sh

连接

执行 `./emqtt_bench conn --help` 可得到该子命令所有可用的参数。它们的解释已包含在上表中，此处略过。

例如，我们启动 **1000** 个连接：

```
1 ./emqtt_bench conn -h emqx-server -c 1000
```

sh

SSL 连接

`emqtt_bench` 支持建立 **SSL** 的安全连接，并执行测试。

单向证书，例如：

```
1 ./emqtt_bench sub -c 100 -i 10 -t bench/%i -p 8883 -S
2 ./emqtt_bench pub -c 100 -I 10 -t bench/%i -p 8883 -s 256 -S
```

sh

双向证书，例如：

```
1 ./emqtt_bench sub -c 100 -i 10 -t bench/%i -p 8883 --certfile path/to/client-cert.pem --keyfile path/to/client-key.pem
2 ./emqtt_bench pub -c 100 -i 10 -t bench/%i -s 256 -p 8883 --certfile path/to/client-cert.pem --keyfile path/to/client-key.pem
```

sh

典型压测场景

场景说明

此处我们以 **2** 类最典型的场景来验证工具的使用：

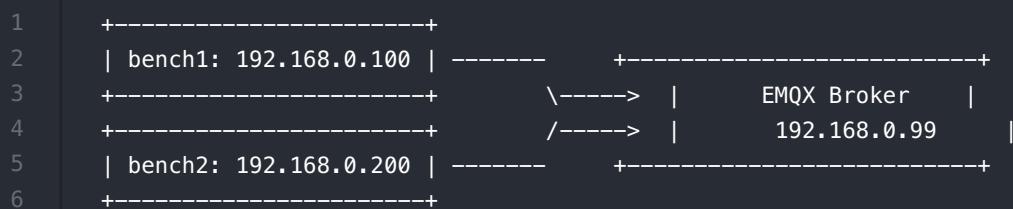
1. 连接量：使用 `emqtt_bench` 创建百万连接到 **EMQX Broker**。
2. 吞吐量：使用 `emqtt_bench` 在 **EMQX** 中创建出 **10W/s** 的 **QoS0** 消息吞吐量。

机器及部署拓扑图

共需准备三台 **8C16G** 服务器，一台为 **EMQX Broker**，两台为 客户端压力机。其中：

- 系统：`CentOS Linux release 7.7.1908 (Core)`
- CPU：`Intel Xeon Processor (Skylake)` 主频：`2693.670 MHZ`
- 服务端：`emqx-centos7-v4.0.2.zip`
- 压力机：`emqtt-bench v0.3.1`
 - 每台压力机分别配置 **10** 张网卡，用于连接测试中建立大量的 **MQTT** 客户端连接

拓扑结构如下：



调优

客户端的压力机和服务端的机器都需要执行系统参数的调优，参见：[系统调优](#)

连接量测试

在执行完系统调优后，首先启动服务端：

```
1 ./bin/emqx start
```

然后在 `bench1` 上的每张网卡上启动 **5w** 的连接数，共计 **50w** 的连接：

```

1 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.100
2 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.101
3 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.102
4 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.103
5 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.104
6 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.105
7 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.106
8 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.107
9 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.108
10 ./emqtt_bench -h 192.168.0.99 -c 50000 --ifaddr 192.168.0.109
11

```

在 `bench2` 上也执行同样的操作。

在所有连接建立完成后，执行 `./bin/emqx_ctl listeners`，并找到以下的内容，查看 **EMQX** 中连接数的信息：

```

1 listener on mqtt:tcp:0.0.0.0:1883
2 acceptors      : 8
3 max_conns      : 1024000
4 current_conn   : 1000000
5 shutdown_count : []

```

吞吐测试

同样的，首先启动服务端：

```
1 ./bin/emqx start
```

在 `bench1` 启动 **500** 个订阅客户端:

```
1 ./emqtt_bench sub -t t -h 192.168.0.99 -c 500
```

sh

然后再 `bench2` 上启动 **20** 个发布端，并且每秒发布 **10** 条消息:

```
1 ./emqtt_bench pub -t t -h 192.168.0.99 -c 20 -I 100
```

sh

然后，回到 `bench1` 上的订阅客户端，可看到当前接收消息的速率，类似于:

```
1 recv(28006): total=2102563, rate=99725(msg/sec)
```

sh

HTTP API

EMQX 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。

EMQX 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。[EMQX 4.0.0](#) 以后的所有 **API** 调用均以 `api/v4` 开头。

接口安全

EMQX 的 **HTTP API** 使用 [Basic 认证](#) 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**。

响应码

HTTP 状态码 (status codes)

EMQX 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

返回码 (result codes)

EMQX 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

API Endpoints

/api/v4

GET /api/v4

返回 EMQX 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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4"
2
3 {"data": [{"path": "/auth_clientid", "name": "list_clientid", "method": "GET", "descr": "List available clientid in the cluster"}, ...], "code": 0}

```

sh

Broker 基本信息

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	EMQX 使用的 Erlang/OTP 版本
data.sysdescr	String	软件描述
data.uptime	String	EMQX 运行时间, 格式为 "H hours, m minutes, s seconds"
data.version	String	EMQX 版本

Examples:

获取所有节点的基本信息:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/brokers"
2
3 {"data": [{"version": "develop", "uptime": "4 hours, 21 minutes, 19 seconds", "sysdescr": "EMQX Broker", "otp_release": "R21/10.3.5", "node_status": "Running", "node": "emqx@127.0.0.1", "datetime": "2020-02-19 15:27:24"}], "code": 0}

```

sh

获取节点 emqx@127.0.0.1 的基本信息:

```

1   $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/brokers/emqx@127.0.0.1"
2
3
{"data": {"version": "develop", "uptime": "1 minutes, 51 seconds", "sysdescr": "EMQX Broker", "otp_release": "R21/10.3.5", "node_status": "Running", "node": "emqx@127.0.0.1", "datetime": "2020-02-20 14:11:31"}, "code": 0}

```

sh

节点

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	EMQX 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	EMQX 运行时间
data.version	String	EMQX 版本

Examples:

获取所有节点的状态：

```
1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes" sh
2
3 {"data":[{"version":"develop","uptime":"7 seconds","process_used":315,"process_available":2097152,"otp_release":"R21/10.3.5","node_status":"Running","node":"emqx@127.0.0.1","memory_use_d":"96.75M","memory_total":"118.27M","max_fds":10240,"load5":"2.60","load15":"2.65","load1":"2.31","connections":0}],"code":0}
```

获取指定节点的状态：

```
1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1" sh
2
3 {"data":{"version":"develop","uptime":"2 minutes, 21 seconds","process_used":310,"process_available":2097152,"otp_release":"R21/10.3.5","node_status":"Running","node":"emqx@127.0.0.1","memory_used":101379168,"memory_total":123342848,"max_fds":10240,"load5":"2.50","load15":"2.61","load1":"1.99","connections":0},"code":0}
```

客户端

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: <code>connected</code> , <code>idle</code> , <code>disconnected</code>
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: <code>MQTT</code> , <code>CoAP</code> , <code>LwM2M</code> , <code>MQTT-SN</code>
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, 大于等于查找
_lte_created_at	Integer	False	客户端会话创建时间, 小于等于查找
_gte_connected_at	Integer	False	客户端连接创建时间, 大于等于查找
_lte_connected_at	Integer	False	客户端连接创建时间, 小于等于查找
_gte_mqueue_len	Integer	False	客户端消息队列当前长度, 大于等于查找
_lte_mqueue_len	Integer	False	客户端消息队列当前长度, 小于等于查找
_gte_mqueue_dropped	Integer	False	消息队列因超出长度而丢弃的消息数量丢弃个数, 大于等于查找
_lte_mqueue_dropped	Integer	False	消息队列因超出长度而丢弃的消息数量丢弃个数, 小于等于查找

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"

<code>data[0].disconnected_at</code>	<code>Type String</code>	客户端离线时间，格式为 "YYYY-MM-DD HH:mm:ss"，此字段仅在 <code>connected</code> 为 <code>false</code> 时有效并被返回
<code>data[0].connected</code>	<code>Boolean</code>	客户端是否处于连接状态
<code>data[0].zone</code>	<code>String</code>	指示客户端使用的配置组
<code>data[0].keepalive</code>	<code>Integer</code>	保持连接时间，单位：秒
<code>data[0].clean_start</code>	<code>Boolean</code>	指示客户端是否使用了全新的会话
<code>data[0].expiry_interval</code>	<code>Integer</code>	会话过期间隔，单位：秒
<code>data[0].created_at</code>	<code>String</code>	会话创建时间，格式为 "YYYY-MM-DD HH:mm:ss"
<code>data[0].subscriptions_cnt</code>	<code>Integer</code>	此客户端已建立的订阅数量
<code>data[0].max_subscriptions</code>	<code>Integer</code>	此客户端允许建立的最大订阅数量
<code>data[0].inflight</code>	<code>Integer</code>	飞行队列当前长度
<code>data[0].max_inflight</code>	<code>Integer</code>	飞行队列最大长度
<code>data[0].mqueue_len</code>	<code>Integer</code>	消息队列当前长度
<code>data[0].max_mqueue</code>	<code>Integer</code>	消息队列最大长度
<code>data[0].mqueue_dropped</code>	<code>Integer</code>	消息队列因超出长度而丢弃的消息数量
<code>data[0].awaiting_rel</code>	<code>Integer</code>	未确认的 PUBREC 报文数量
<code>data[0].max_awaiting_rel</code>	<code>Integer</code>	允许存在未确认的 PUBREC 报文的最大数量
<code>data[0].recv_oct</code>	<code>Integer</code>	EMQX Broker (下同) 接收的字节数量
<code>data[0].recv_cnt</code>	<code>Integer</code>	接收的 TCP 报文数量
<code>data[0].recv_pkt</code>	<code>Integer</code>	接收的 MQTT 报文数量
<code>data[0].recv_msg</code>	<code>Integer</code>	接收的 PUBLISH 报文数量
<code>data[0].send_oct</code>	<code>Integer</code>	发送的字节数量
<code>data[0].send_cnt</code>	<code>Integer</code>	发送的 TCP 报文数量
<code>data[0].send_pkt</code>	<code>Integer</code>	发送的 MQTT 报文数量
<code>data[0].send_msg</code>	<code>Integer</code>	发送的 PUBLISH 报文数量
<code>data[0].mailbox_len</code>	<code>Integer</code>	进程邮箱大小
<code>data[0].heap_size</code>	<code>Integer</code>	进程堆栈大小，单位：字节
<code>data[0].reductions</code>	<code>Integer</code>	Erlang reduction
<code>meta</code>	<code>Object</code>	分页信息
<code>meta.page</code>	<code>Integer</code>	页码
<code>meta.limit</code>	<code>Integer</code>	每页显示的数据条数
<code>meta.count</code>	<code>Integer</code>	数据总条数

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/clients?_page=1&_limit
2 =10"
3
4 {"meta": {"page": 1, "limit": 10, "count": 1}, "data": [{"zone": "external", "recv_cnt": 2, "max_mqueue": 1000, "node": "emqx@127.0.0.1", "username": "test", "mqueue_len": 0, "max_inflight": 32, "is_bridge": false, "mqueue_dropped": 0, "inflight": 0, "heap_size": 2586, "max_subscriptions": 0, "proto_name": "MQTT", "created_at": "2020-02-19 17:01:26", "proto_ver": 4, "reductions": 3997, "send_msg": 0, "ip_address": "127.0.0.1", "send_cnt": 0, "mailbox_len": 1, "awaiting_rel": 0, "keepalive": 60, "recv_msg": 0, "send_pkt": 0, "recv_oct": 29, "clientid": "example", "clean_start": true, "expiry_interval": 0, "connected": true, "port": 64491, "send_oct": 0, "recv_pkt": 1, "connected_at": "2020-02-19 17:01:26", "max_awaiting_rel": 100, "subscriptions_cnt": 0}], "code": 0}

```

注：在 **4.1** 后，返回的 `meta` 内容做了修改：

- `count`：仍表示总数，但在 多条件/模糊查询时，固定为 **-1**。
- `hasnext`：为新增字段，表示是否存在下一页。

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

Examples:

查询指定客户端

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/clients/example"
2
3 {"data": [{"recv_cnt": 2, "max_subscriptions": 0, "node": "emqx@127.0.0.1", "proto_ver": 4, "recv_pkt": 0, "inflight": 0, "max_mqueue": 1000, "heap_size": 2586, "username": "test", "proto_name": "MQTT", "subscriptions_cnt": 0, "send_pkt": 0, "created_at": "2020-02-20 13:38:51", "reductions": 3978, "ip_address": "127.0.0.1", "send_msg": 0, "send_cnt": 0, "expiry_interval": 0, "keepalive": 60, "mqueue_dropped": 0, "is_bridge": false, "max_inflight": 32, "recv_msg": 0, "max_awaiting_rel": 100, "awaiting_rel": 0, "mailbox_len": 1, "mqueue_len": 0, "recv_oct": 29, "connected_at": "2020-02-20 13:38:51", "clean_start": true, "clientid": "example", "connected": true, "port": 54889, "send_oct": 0, "zone": "external"}], "code": 0}

```

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

Examples:

踢除指定客户端

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/clients/example"      sh
2
3 {"code":0}

```

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 <code>emqx-management</code> 插件的配置项 <code>max_row_limit</code> 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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/c
2 lients?_page=1&_limit=10"
3
4 {"meta": {"page": 1, "limit": 10, "count": 1}, "data": [{"recv_cnt": 2, "max_subscriptions": 0, "node": "emqx@127.0.0.1", "proto_ver": 4, "recv_pkt": 1, "inflight": 0, "max_mqueue": 1000, "heap_size": 2586, "username": "test", "proto_name": "MQTT", "subscriptions_cnt": 0, "send_pkt": 0, "created_at": "2020-02-19 18:25:18", "reductions": 4137, "ip_address": "127.0.0.1", "send_msg": 0, "send_cnt": 0, "expiry_interval": 0, "keepalive": 60, "mqueue_dropped": 0, "is_bridge": false, "max_inflight": 32, "recv_msg": 0, "max_awaiting_rel": 100, "awaiting_rel": 0, "mailbox_len": 1, "mqueue_len": 0, "recv_oct": 29, "connected_at": "2020-02-19 18:25:18", "clean_start": true, "clientid": "example", "connected": true, "port": 49509, "send_oct": 0, "zone": "external"}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/c
2 lients/example"
3
4 {"data": [{"recv_cnt": 4, "max_subscriptions": 0, "node": "emqx@127.0.0.1", "proto_ver": 4, "recv_pkt": 1, "inflight": 0, "max_mqueue": 1000, "heap_size": 2586, "username": "test", "proto_name": "MQTT", "subscriptions_cnt": 0, "send_pkt": 3, "created_at": "2020-02-20 13:38:51", "reductions": 5994, "ip_address": "127.0.0.1", "send_msg": 0, "send_cnt": 3, "expiry_interval": 0, "keepalive": 60, "mqueue_dropped": 0, "is_bridge": false, "max_inflight": 32, "recv_msg": 0, "max_awaiting_rel": 100, "awaiting_rel": 0, "mailbox_len": 0, "mqueue_len": 0, "recv_oct": 33, "connected_at": "2020-02-20 13:38:51", "clean_start": true, "clientid": "example", "connected": true, "port": 54889, "send_oct": 8, "zone": "externa
l"}, "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/clients/username/steve" sh
2
3
4 {"data": [{"clean_start": true, "awaiting_rel": 0, "recv_msg": 0, "proto_name": "MQTT", "recv_cnt": 2, "mailbox_len": 0, "node": "emqx@127.0.0.1", "mqueue_len": 0, "max_subscriptions": 0, "created_at": "2020-02-20 13:50:11", "is_bridge": false, "heap_size": 2586, "proto_ver": 4, "subscriptions_cnt": 0, "clientid": "example", "expiry_interval": 0, "send_msg": 0, "inflight": 0, "reductions": 4673, "send_pkts": 1, "zone": "external", "send_cnt": 1, "ip_address": "127.0.0.1", "keepalive": 60, "max_inflight": 32, "recv_oct": 29, "recv_pkt": 1, "max_awaiting_rel": 100, "username": "steve", "connected_at": "2020-02-20 13:50:11", "connected": true, "port": 56429, "send_oct": 4, "mqueue_dropped": 0, "max_mqueue": 1000}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/c
2 lients/username/test"
3
4 {"data": [{"clean_start": true, "awaiting_rel": 0, "recv_msg": 0, "proto_name": "MQTT", "recv_cnt": 6,
5 "mailbox_len": 0, "node": "emqx@127.0.0.1", "mqueue_len": 0, "max_subscriptions": 0, "created_at": "2
6 020-02-20 13:50:11", "is_bridge": false, "heap_size": 1598, "proto_ver": 4, "subscriptions_cnt": 0,
7 "clientid": "example", "expiry_interval": 0, "send_msg": 0, "inflight": 0, "reductions": 7615, "send_pk
8 t": 5, "zone": "external", "send_cnt": 5, "ip_address": "127.0.0.1", "keepalive": 60, "max_inflight": 3
9 2, "recv_oct": 37, "recv_pkt": 1, "max_awaiting_rel": 100, "username": "test", "connected_at": "2020-0
10 2-20 13:50:11", "connected": true, "port": 56429, "send_oct": 12, "mqueue_dropped": 0, "max_mqueue": 1
11 000}], "code": 0}

```

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

Examples:

查询 **ACL** 缓存

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/clients/example/acl_ca
2 che"
3
4 {"data": [{"updated_time": 1582180824571, "topic": "test", "result": "allow", "access": "publish"}],
5 "code": 0}

```

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

Examples:

清除 ACL 缓存

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/clients/example/acl
2   _cache"
3
4 {"code":0}

```

PUT /api/v4/clients/{clientid}/keepalive设置指定客户端的**keepalive**时间（秒）。**Path Parameters:**

Name	Type	Required	Description
clientid	String	True	ClientID

Query String Parameters:

Name	Type	Required	Description
interval	Integer	True	秒：0~65535，0表示不启动 keepalive 检查

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:更新指定客户端（example）**keepalive**为10秒

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/clients/example/keepal
2   ive?interval=10"
3
4 {"code":0}

```

除了通过 **Query String** 传参外，还可以使用 **Body**。

```

1 curl -u admin:public -X 'PUT' http://127.0.0.1:18083/api/v4/clients/test/keepalive -d '{"i'
2 nterval": 10}'
3 {"code":0}

```

sh

订阅信息

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/subscriptions?_page=1&
2 _limit=10"
3
{"meta": {"page": 1, "limit": 10000, "count": 2}, "data": [{"topic": "a/+/c", "qos": 0, "node": "emqx@127.0.0.1", "clientid": "78082755-e8eb-4a87-bab7-8277541513f0"}, {"topic": "a/b/c", "qos": 1, "node": "emqx@127.0.0.1", "clientid": "7a1dfceb-89c0-4f7e-992b-dfeb09329f01"}], "code": 0}

```

注：在 **4.1** 后，返回的 `meta` 内容做了修改：

- `count`：仍表示总数，但在 多条件/模糊查询 时，固定为 **-1**。
- `hasnext`：为新增字段，表示是否存在下一页。

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
<code>clientid</code>	String	True	ClientID

Success Response Body (JSON):

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/subscriptions/123"
2
3 {"data": [{"topic": "a/b/c", "qos": 1, "node": "emqx@127.0.0.1", "clientid": "123"}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/subscriptions?_page=1&limit=10"
2
3
4 {"meta": {"page": 1, "limit": 10000, "count": 2}, "data": [{"topic": "a/+c", "qos": 0, "node": "emqx@127.0.0.1", "clientid": "78082755-e8eb-4a87-bab7-8277541513f0"}, {"topic": "a/b/c", "qos": 1, "node": "emqx@127.0.0.1", "clientid": "7a1dfceb-89c0-4f7e-992b-dfeb09329f01"}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/subscriptions/sample"
2
3 {"data":[{"topic":"a/+/c","qos":0,"node":"emqx@127.0.0.1","clientid":"sample"}],"code":0}

```

路由

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/routes"
2
3 {"meta":{"page":1,"limit":10000,"count":2},"data":[{"topic":"a/+/c","node":"emqx@127.0.0.1"}, {"topic":"a/b/c","node":"emqx@127.0.0.1"}],"code":0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/routes/a%2fb%2fc"
2
3 {"data":[{"topic":"a/b/c","node":"emqx@127.0.0.1"}],"code":0}

```

消息发布

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
user_properties	Object	Optional	{}	PUBLISH 消息里的 User Property 字段 (MQTT 5.0)

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/publish" -d \
2   '{"topic":"a/b/c", "payload":"Hello World", "qos":1, "retain":false, "clientid":"example", "use"
3   _properties": { "id": 10010, "name": "emqx", "foo": "bar"}}'
4
{"code":0}

```

主题订阅

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

Examples:

同时订阅 a, b, c 三个主题

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/subscribe" -d \
2   {"topics":"a,b,c","qos":1,"clientid":"example"}
3
{"code":0}

```

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

Examples:取消订阅 `a` 主题

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/unsubscribe" -d
2   '{"topic":"a","qos":1,"clientid":"example"}'
3
4   {"code":0}

```

消息批量发布

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 <code>topics</code> 至少指定其中之一
[0].topics	String	Optional		以 <code>,</code> 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 <code>plain</code> 与 <code>base64</code> 两种
[0].qos	Integer	Optional	0	QoS 等级
[0].retain	Boolean	Optional	false	是否为保留消息
[0].user_properties	Object	Optional	{}	PUBLISH 消息里的 User Property 字段 (MQTT 5.0)

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/publish_batch"
2 -d '[{"topic":"a/b/c","payload":"Hello World","qos":1,"retain":false,"clientid":"example","user_"
3 _properties":{"id": 10010, "name": "emqx", "foo": "bar"}}, {"topic":"a/b/c","payload":"Hello Wor
d Again","qos":0,"retain":false,"clientid":"example","user_properties": { "id": 10010, "name": "emqx", "foo": "bar"}}]'
4
5 {"data":[{"topic":"a/b/c","code":0}, {"topic":"a/b/c","code":0}], "code":0}

```

主题批量订阅

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

Examples:

一次性订阅 a, b, c 三个主题

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/subscribe_batch"
2 -d '[{"topic":"a","qos":1,"clientid":"example"}, {"topic":"b","qos":1,"clientid":"example"}, {"topic":"c","qos":1,"clientid":"example"}]'
3
4 {"code":0}

```

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

Examples:

一次性取消订阅 `a` , `b` 主题

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/mqtt/unsubscribe_batch"
2   -d '[{"topic":"a","qos":1,"clientid":"example"}, {"topic":"b","qos":1,"clientid":"example"}]'
3
4 {"code":0}

```

插件

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 <code>auth</code> 、 <code>bridge</code> 、 <code>feature</code> 、 <code>protocol</code> 四种类型

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/plugins"
2
3 {"data": [{"plugins": [{"version": "develop", "type": "auth", "name": "emqx_auth_clientid", "description": "EMQX Authentication with ClientId/Password", "active": false}, ...], "node": "emqx@127.0.1"}}, "code": 0}

```

sh

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/plugins"
2
3 {"data": [{"version": "develop", "type": "auth", "name": "emqx_auth_clientid", "description": "EMQX Authentication with ClientId/Password", "active": false}, ...], "code": 0}

```

sh

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

Examples:

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/p
2 lugins/emqx_delayed_publish/load"
3 {"code":0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/p
2 lugins/emqx_delayed_publish/unload"
3 {"code":0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/p
2 lugins/emqx_delayed_publish/reload"
3 {"code":0}

```

监听器

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/listeners"
2
3 {"data": [{"node": "emqx@127.0.0.1", "listeners": [{"shutdown_count": [], "protocol": "mqtt:ssl", "max_conns": 102400, "listen_on": "8883", "current_conns": 0, "acceptors": 16}, {"shutdown_count": [], "protocol": "mqtt:tcp", "max_conns": 1024000, "listen_on": "0.0.0.0:1883", "current_conns": 13, "acceptors": 8}, {"shutdown_count": [], "protocol": "mqtt:tcp", "max_conns": 1024000, "listen_on": "127.0.0.1:11883", "current_conns": 0, "acceptors": 4}, {"shutdown_count": [], "protocol": "http:dashboard", "max_conns": 512, "listen_on": "18083", "current_conns": 0, "acceptors": 4}, {"shutdown_count": [], "protocol": "http:management", "max_conns": 512, "listen_on": "8081", "current_conns": 1, "acceptors": 2}, {"shutdown_count": [], "protocol": "https:dashboard", "max_conns": 512, "listen_on": "18084", "current_conns": 0, "acceptors": 2}, {"shutdown_count": [], "protocol": "mqtt:ws:8083", "max_conns": 102400, "listen_on": "8083", "current_conns": 1, "acceptors": 4}, {"shutdown_count": [], "protocol": "mqtt:wss:8084", "max_conns": 16, "listen_on": "8084", "current_conns": 0, "acceptors": 4}]}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/listeners"
2
3
4 {"data": [{"shutdown_count": [], "protocol": "mqtt:ssl", "max_conns": 102400, "listen_on": "8883", "current_conns": 0, "acceptors": 16}, {"shutdown_count": [], "protocol": "mqtt:tcp", "max_conns": 102400, "listen_on": "0.0.0.0:1883", "current_conns": 13, "acceptors": 8}, {"shutdown_count": [], "protocol": "mqtt:tcp", "max_conns": 1024000, "listen_on": "127.0.0.1:11883", "current_conns": 0, "acceptors": 4}, {"shutdown_count": [], "protocol": "http:dashboard", "max_conns": 512, "listen_on": "18083", "current_conns": 0, "acceptors": 4}, {"shutdown_count": [], "protocol": "http:management", "max_conns": 512, "listen_on": "8081", "current_conns": 1, "acceptors": 2}, {"shutdown_count": [], "protocol": "http:dashboard", "max_conns": 512, "listen_on": "18084", "current_conns": 0, "acceptors": 2}, {"shutdown_count": [], "protocol": "mqtt:ws:8083", "max_conns": 102400, "listen_on": "8083", "current_conns": 1, "acceptors": 4}, {"shutdown_count": [], "protocol": "mqtt:wss:8084", "max_conns": 16, "listen_on": "8084", "current_conns": 0, "acceptors": 4}], "code": 0}

```

内置模块

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)

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/modules"
2
3 {"data": [{"node": "emqx@127.0.0.1", "modules": [{"name": "emqx_mod_delayed", "description": "EMQX Delayed Publish Module", "active": true}, {"name": "emqx_mod_topic_metrics", "description": "EMQX Topic Metrics Module", "active": false}, {"name": "emqx_mod_subscription", "description": "EMQX Subscription Module", "active": false}, {"name": "emqx_mod_acl_internal", "description": "EMQX Internal ACL Module", "active": true}, {"name": "emqx_mod_rewrite", "description": "EMQX Topic Rewrite Module", "active": false}, {"name": "emqx_mod_presence", "description": "EMQX Presence Module", "active": true}], "code": 0}

```

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。

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

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/modules/emqx_mod_topic
2 _metrics/load"
3
4 {"code": 0}

```

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。

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

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/modules/emqx_mod_topic
2 _metrics/unload"
3 {"code":0}

```

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。

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

```

1 $ curl -i --basic -u admin:public -X PUT "http://localhost:8081/api/v4/modules/emqx_mod_acl_i
2 nternal/reload"
3 {"code":0}

```

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。

统计指标

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
bytes.received	Integer	EMQX 接收的字节数
bytes.sent	Integer	EMQX 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	EMQX 存储的延迟发布的消息数量
messages.delivered	Integer	EMQX 内部转发到订阅进程的消息数量
messages.dropped	Integer	EMQX 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

<code>messages.qos0.sent</code>	<code>Integer</code>	发送给客户端的 QoS 0 消息数量
<code>messages.qos1.sent</code>	<code>Integer</code>	发送给客户端的 QoS 1 消息数量
<code>messages.qos2.sent</code>	<code>Integer</code>	发送给客户端的 QoS 2 消息数量
<code>messages.received</code>	<code>Integer</code>	接收来自客户端的消息数量，等于 <code>messages.qos0.received</code> , <code>messages.qos1.received</code> 与 <code>messages.qos2.received</code> 之和
<code>messages.sent</code>	<code>Integer</code>	发送给客户端的消息数量，等于 <code>messages.qos0.sent</code> , <code>messages.qos1.sent</code> 与 <code>messages.qos2.sent</code> 之和
<code>messages.retained</code>	<code>Integer</code>	EMQX 存储的保留消息数量
<code>messages.acked</code>	<code>Integer</code>	接收的 PUBACK 和 PUBREC 报文数量
<code>packets.received</code>	<code>Integer</code>	接收的报文数量
<code>packets.sent</code>	<code>Integer</code>	发送的报文数量
<code>packets.connect.received</code>	<code>Integer</code>	接收的 CONNECT 报文数量
<code>packets.connack.auth_error</code>	<code>Integer</code>	接收的认证失败的 CONNECT 报文数量
<code>packets.connack.error</code>	<code>Integer</code>	接收的未成功连接的 CONNECT 报文数量
<code>packets.connack.sent</code>	<code>Integer</code>	发送的 CONNACK 报文数量
<code>packets.publish.received</code>	<code>Integer</code>	接收的 PUBLISH 报文数量
<code>packets.publish.sent</code>	<code>Integer</code>	发送的 PUBLISH 报文数量
<code>packets.publish.inuse</code>	<code>Integer</code>	接收的报文标识符已被占用的 PUBLISH 报文数量
<code>packets.publish.auth_error</code>	<code>Integer</code>	接收的未通过 ACL 检查的 PUBLISH 报文数量
<code>packets.publish.error</code>	<code>Integer</code>	接收的无法被发布的 PUBLISH 报文数量
<code>packets.publish.dropped</code>	<code>Integer</code>	超出接收限制而被丢弃的消息数量
<code>packets.puback.received</code>	<code>Integer</code>	接收的 PUBACK 报文数量
<code>packets.puback.sent</code>	<code>Integer</code>	发送的 PUBACK 报文数量
<code>packets.puback.inuse</code>	<code>Integer</code>	接收的报文标识符已被占用的 PUBACK 报文数量
<code>packets.puback.missed</code>	<code>Integer</code>	接收的未知报文标识符 PUBACK 报文数量
<code>packets.pubrec.received</code>	<code>Integer</code>	接收的 PUBREC 报文数量
<code>packets.pubrec.sent</code>	<code>Integer</code>	发送的 PUBREC 报文数量
<code>packets.pubrec.inuse</code>	<code>Integer</code>	接收的报文标识符已被占用的 PUBREC 报文数量
<code>packets.pubrec.missed</code>	<code>Integer</code>	接收的未知报文标识符 PUBREC 报文数量
<code>packets.pubrel.received</code>	<code>Integer</code>	接收的 PUBREL 报文数量
<code>packets.pubrel.sent</code>	<code>Integer</code>	发送的 PUBREL 报文数量
<code>packets.pubrel.missed</code>	<code>Integer</code>	接收的未知报文标识符 PUBREL 报文数量
<code>packets.pubcomp.received</code>	<code>Integer</code>	接收的 PUBCOMP 报文数量
<code>packets.pubcomp.sent</code>	<code>Integer</code>	发送的 PUBCOMP 报文数量

<code>packets.pubcomp.inuse</code>	<code>Integer</code>	接收的报文标识符已被占用的 PUBCOMP 报文数量
<code>packets.pubcomp.missed</code>	<code>Integer</code>	发送的 PUBCOMP 报文数量
<code>packets.subscribe.received</code>	<code>Integer</code>	接收的 SUBSCRIBE 报文数量
<code>packets.subscribe.error</code>	<code>Integer</code>	接收的订阅失败的 SUBSCRIBE 报文数量
<code>packets.subscribe.auth_error</code>	<code>Integer</code>	接收的未通过 ACL 检查的 SUBACK 报文数量
<code>packets.suback.sent</code>	<code>Integer</code>	发送的 SUBACK 报文数量
<code>packets.unsubscribe.received</code>	<code>Integer</code>	接收的 UNSUBSCRIBE 报文数量
<code>packets.unsubscribe.error</code>	<code>Integer</code>	接收的取消订阅失败的 UNSUBSCRIBE 报文数量
<code>packets.unsuback.sent</code>	<code>Integer</code>	发送的 UNSUBACK 报文数量
<code>packets.pingreq.received</code>	<code>Integer</code>	接收的 PINGREQ 报文数量
<code>packets.pingresp.sent</code>	<code>Integer</code>	发送的 PUBRESP 报文数量
<code>packets.disconnect.received</code>	<code>Integer</code>	接收的 DISCONNECT 报文数量
<code>packets.disconnect.sent</code>	<code>Integer</code>	发送的 DISCONNECT 报文数量
<code>packets.auth.received</code>	<code>Integer</code>	接收的 AUTH 报文数量
<code>packets.auth.sent</code>	<code>Integer</code>	发送的 AUTH 报文数量
<code>session.created</code>	<code>Integer</code>	创建的会话数量
<code>session.discarded</code>	<code>Integer</code>	由于 <code>Clean Session</code> 或 <code>Clean Start</code> 为 <code>true</code> 而被丢弃的会话数量
<code>session.resumed</code>	<code>Integer</code>	由于 <code>Clean Session</code> 或 <code>Clean Start</code> 为 <code>false</code> 而恢复的会话数量
<code>session.takeovered</code>	<code>Integer</code>	由于 <code>Clean Session</code> 或 <code>Clean Start</code> 为 <code>false</code> 而被接管的会话数量
<code>session.terminated</code>	<code>Integer</code>	终结的会话数量

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/metrics"
2
3 {"data": [{"node": "emqx@127.0.0.1", "metrics": {"messages.dropped.no_subscribers": 0, "packets.co
4 nnack.sent": 13, "bytes.received": 805, "messages.received": 0, "packets.unsuback.sent": 0, "message
5 s.delivered": 0, "client.disconnected": 0, "packets.puback.sent": 0, "packets.subscribe.auth_error": 0,
6 "delivery.dropped.queue_full": 0, "messages.forward": 0, "delivery.dropped.qos0_msg": 0, "deliv
7 ery.dropped.expired": 0, "bytes.sent": 52, "messages.sent": 0, "delivery.dropped.no_local": 0, "pack
8 etcs.pubrec.received": 0, "packets.pubcomp.received": 0, "client.check_acl": 0, "packets.puback.rec
9 eived": 0, "session.takeovered": 0, "messages.dropped.expired": 0, "messages.qos1.sent": 0, "messag
10 es.retained": 0, "packets.pubcomp.inuse": 0, "packets.pubrec.sent": 0, "packets.received": 13, "mess
11 ages.acked": 0, "session.terminated": 0, "packets.sent": 13, "packets.unsubscribe.error": 0, "client.
12 connect": 13, "packets.pubrec.missed": 0, "packets.auth.sent": 0, "packets.disconnect.received": 0,
13 "messages.qos2.sent": 0, "client.auth.anonymous": 13, "packets.auth.received": 0, "packets.unsubscribe.
14 received": 0, "packets.publish.auth_error": 0, "client.connected": 13, "packets.disconnect.se
15 nt": 0, "session.created": 13, "packets.pingreq.received": 0, "messages.dropped": 0, "packets.publis
16 h.sent": 0, "session.resumed": 0, "packets.connack.auth_error": 0, "packets.pubrel.sent": 0, "delive
17 ry.dropped": 0, "packets.pubcomp.sent": 0, "messages.qos2.received": 0, "messages.qos0.received": 0,
18 "packets.publish.inuse": 0, "client.unsubscribe": 0, "packets.pubrel.received": 0, "client.connc
19 ak": 13, "packets.connack.error": 0, "packets.publish.dropped": 0, "packets.publish.received": 0, "cl
20 ient.subscribe": 0, "packets.subscribe.error": 0, "packets.suback.sent": 0, "packets.pubcomp.mis
21 sed": 0, "messages.qos1.received": 0, "delivery.dropped.too_large": 0, "packets.pingresp.sent": 0, "pa
22 ckets.pubrel.missed": 0, "messages.qos0.sent": 0, "packets.connect.received": 13, "packets.puback.
23 missed": 0, "packets.subscribe.received": 0, "packets.puback.inuse": 0, "client.authenticate": 13, "m
24 essages.publish": 0, "packets.pubrec.inuse": 0, "packets.publish.error": 0, "messages.delayed": 0,
25 "session.discarded": 0}], "code": 0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/metrics"
2
3
4 {"data": {"bytes.received": 0, "client.connected": 0, "packets.pingreq.received": 0, "messages.delayed": 0, "packets.puback.sent": 0, "packets.pingresp.sent": 0, "packets.publish.auth_error": 0, "client.check_acl": 0, "delivery.dropped.queue_full": 0, "packets.publish.error": 0, "packets.pubcomp.received": 0, "bytes.sent": 0, "packets.pubrec.inuse": 0, "packets.pubrec.missed": 0, "packets.pubrel.sent": 0, "delivery.dropped.too_large": 0, "packets.pubcomp.missed": 0, "packets.subscribe.error": 0, "packets.suback.sent": 0, "messages.qos2.sent": 0, "messages.qos1.sent": 0, "packets.pubrel.missed": 0, "messages.publish": 0, "messages.forward": 0, "packets.auth收到了": 0, "delivery.dropped": 0, "packets.sent": 0, "packets.puback.inuse": 0, "delivery.dropped.qos0_msg": 0, "packets.publish.dropped": 0, "packets.disconnect.sent": 0, "packets.auth.sent": 0, "packets.unsubscribe.received": 0, "session.takeovered": 0, "messages.delivered": 0, "client.auth.anonymous": 0, "packets.connack.error": 0, "packets.connack.sent": 0, "packets.subscribe.auth_error": 0, "packets.unsuback.sent": 0, "packets.pubcomp.sent": 0, "packets.publish.sent": 0, "client.connack": 0, "packets.publish.received": 0, "client.subscribe": 0, "session.created": 0, "delivery.dropped.expired": 0, "client.unsubscribe": 0, "packets.received": 0, "packets.pubrel.received": 0, "packets.unsubscribe.error": 0, "messages.qos0.sent": 0, "packets.connack.auth_error": 0, "session.resumed": 0, "delivery.dropped.no_local": 0, "packets.puback.missed": 0, "packets.pubcomp.inuse": 0, "packets.pubrec.sent": 0, "messages.dropped.expired": 0, "messages.dropped.no_subscribers": 0, "session.discard": 0, "messages.sent": 0, "messages.received": 0, "packets.puback.received": 0, "messages.qos0.received": 0, "messages.acked": 0, "client.connect": 0, "packets.disconnect.received": 0, "client.disconnected": 0, "messages.retained": 3, "session.terminated": 0, "packets.publish.inuse": 0, "packets.pubrec.received": 0, "messages.qos2.received": 0, "messages.dropped": 0, "packets.connect.received": 0, "client.authenticate": 0, "packets.subscribe.received": 0, "messages.qos1.received": 0}, "code": 0}

```

主题统计指标

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/topic-metrics"
2
3 {"data":[],"code":0}
4 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/topic-metrics" -d '{
5   "topic":"a/b/c"}'
6
7 {"code":0}
8 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/topic-metrics"
9
{"data": [{"topic": "a/b/c", "metrics": {"messages.qos2.out.rate": 0.0, "messages.qos2.out.count": 0, "messages.qos2.in.rate": 0.0, "messages.qos2.in.count": 0, "messages.qos1.out.rate": 0.0, "messages.qos1.out.count": 0, "messages.qos1.in.rate": 0.0, "messages.qos1.in.count": 0, "messages.qos0.out.rate": 0.0, "messages.qos0.out.count": 0, "messages.qos0.in.rate": 0.0, "messages.qos0.in.count": 0, "messages.out.rate": 0.0, "messages.out.count": 0, "messages.in.rate": 0.0, "messages.in.count": 0, "messages.dropped.rate": 0.0, "messages.dropped.count": 0}}}, "code": 0}
```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/topic-metrics/a%2Fb%2F
2 c"
3
{"data": {"messages.qos2.out.rate": 0.0, "messages.qos2.out.count": 0, "messages.qos2.in.rate": 0.0,
4 "messages.qos2.in.count": 0, "messages.qos1.out.rate": 0.0, "messages.qos1.out.count": 0, "mess
5 ges.qos1.in.rate": 0.0, "messages.qos1.in.count": 0, "messages.qos0.out.rate": 0.0, "mess
6 ages.qos0.out.count": 0, "messages.qos0.in.rate": 0.0, "messages.qos0.in.count": 0, "mess
7 ages.out.rate": 0.0, "messages.out.count": 0, "messages.in.rate": 0.0, "messages.in.count": 0, "mess
8 ages.dropped.rate": 0.0, "messages.dropped.count": 0}, "code": 0}

```

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

Examples:

开启对 `a/b/c` 主题的指标统计

```
1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/topic-metrics" -d '{sh
2   "topic":"a/b/c"}'
3
4 {"code":0}
```

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

Examples:

关闭对 `a/b/c` 主题的指标统计

```
1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/topic-metrics/a%2Fb%sh
2 %2Fc"
3
4 {"code":0}
```

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

Examples:

关闭所有主题的指标统计

```
1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/topic-metrics"      sh
2
3 {"code":0}
```

状态

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/stats"                                     sh
2
3 {"data": [{"stats": {"topics.max": 0, "topics.count": 0, "subscriptions.shared.max": 0, "subscriptions.shared.count": 0, "subscriptions.max": 0, "subscriptions.count": 0, "subscribers.max": 0, "subscribers.count": 0, "suboptions.max": 0, "suboptions.count": 0, "sessions.max": 0, "sessions.count": 0, "routes.max": 0, "routes.count": 0, "retained.max": 3, "retained.count": 3, "resources.max": 0, "resources.count": 0, "connections.max": 0, "connections.count": 0, "channels.max": 0, "channels.count": 0}, "node": "emqx@127.0.0.1"}], "code": 0}

```

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
data	Array of Objects	各节点上的状态数据列表, 详见 GET /api/v4/stats

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/stats"
2
3
4 {"data": {"topics.max": 0, "topics.count": 0, "subscriptions.shared.max": 0, "subscriptions.shared.count": 0, "subscriptions.max": 0, "subscriptions.count": 0, "subscribers.max": 0, "subscribers.count": 0, "suboptions.max": 0, "suboptions.count": 0, "sessions.max": 0, "sessions.count": 0, "routes.max": 0, "routes.count": 0, "retained.max": 3, "retained.count": 3, "resources.max": 0, "resources.count": 0, "connections.max": 0, "connections.count": 0, "channels.max": 0, "channels.count": 0}, "code": 0}

```

告警

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

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/alarms"
2
3 {"data": [{"node": "emqx@127.0.0.1", "alarms": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": "infinity", "activated": true, "activate_at": 1597996203658236}, {"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": 1597994359335482, "activated": false, "activate_at": 1597993108657522}]}], "code": 0}

```

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](#)。

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/alarms"
2
3 {"data": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": "infinity", "activated": true, "activate_at": 1597996203658236}, {"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": 1597994359335482, "activated": false, "activate_at": 1597993108657522}], "code": 0}

```

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](#)。

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/alarms/activated"
2
3 {"data": [{"node": "emqx@127.0.0.1", "alarms": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": "infinity", "activated": true, "activate_at": 1597996203658236}]}], "code": 0}

```

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](#)。

Examples:

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/alarms/activated"
2
3 {"data": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": "infinity", "activated": true, "activate_at": 1597996203658236}], "code": 0}

```

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](#)。

Examples:

```
1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/alarms/deactivated" sh
2
3 {"data": [{"node": "emqx@127.0.0.1", "alarms": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": 1597994359335482, "activated": false, "activate_at": 1597993108657522}], "code": 0}
```

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](#)。

Examples:

```
1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/nodes/emqx@127.0.0.1/alarms/deactivated" sh
2
3 {"data": [{"name": "high_system_memory_usage", "message": "System memory usage is higher than 60%", "details": {"high_watermark": 60}, "deactivate_at": 1597994359335482, "activated": false, "activate_at": 1597993108657522}], "code": 0}
```

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

Examples:

```
1 $ curl -i --basic -u admin:public -vX POST "http://localhost:8081/api/v4/alarms/deactivated" sh
2 -d '{"node": "emqx@127.0.0.1", "name": "high_system_memory_usage"}'
3
4 {"code": 0}
```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/alarms/deactivated"
2
3 {"code":0}

```

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

Examples:

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/nodes/emqx@127.0.0.
2 1/alarms/deactivated"
3 {"code":0}

```

ACL 缓存

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

Examples:

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/acl-cache"
2
3 {"code":0}

```

sh

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

Examples:

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/node/emqx@127.0.0.1
2 /acl-cache"
3 {"code":0}

```

sh

黑名单

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 <code>/api/v4/clients</code>

Examples:

获取黑名单列表：

```

1 $ curl -i --basic -u admin:public -vX GET "http://localhost:8081/api/v4/banned"
2
3 {"meta": {"page": 1, "limit": 10000, "count": 1}, "data": [{"who": "example", "until": 1582265833, "reas
on": "undefined", "by": "user", "at": 1582265533, "as": "clientid"}], "code": 0}

```

sh

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
reason	String	Required		详细信息
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

Examples:

将 client 添加到黑名单:

```

1 $ curl -i --basic -u admin:public -vX POST "http://localhost:8081/api/v4/banned" -d '{"who":'
2   "example", "as": "clientid", "reason": "example"}'
3
4 {"data": {"who": "example", "as": "clientid"}, "code": 0}

```

sh

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

Examples:

将 `client` 从黑名单中移除：

```
1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/banned/clientid/exa
2 mple"
3 {"code":0}
```

数据导入导出

数据导入导出。

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

Examples:

列出当前的导出文件信息列表：

```
1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/data/export"
2
3 {"data":[{"size":350,"filename":"emqx-export-2020-5-15-18-6-29.json","created_at":"2020-5-15
18:6:29"}, {"size":388,"filename":"emqx-export-2020-5-15-17-39-0.json","created_at":"2020-5-
15 17:39:0"}], "code":0}
```

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

Examples:

导出文件:

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/export"           sh
2
3 {"data": {"size": 350, "filename": "emqx-export-2020-5-18-17-17-44.json", "created_at": "2020-5-18"
17:17:44"}, "code": 0}

```

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

Examples:

从指定文件导入数据:

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/import" -d '{"f'
2 ilename": "emqx-export-2020-5-18-17-17-44.json"}'
3 {"code": 0}

```

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

Examples:

下载指定的数据文件：

```

1 $ curl -i --basic -u admin:public -X GET "http://localhost:8081/api/v4/data/file/emqx-export-
2 2020-5-18-17-17-44.json"
3
{"filename":"/Users/zhouzibo/emqx-rel/_build/emqx/rel/emqx/data/emqx-export-2020-5-18-17-17-44.
json","file":{"version":"dev","users":[{"username":"admin","tags":["administrator","password":"oKQPB1hbivg6+2ntALELNOb1fF0="]},{"schemas":[],"rules":[],"resources":[],"date":"2020-05-18 17:17:44","blacklist":[]}, {"auth_mnesia":[],"apps": [{"status":true,"secret":"public","name":"Default","id":"admin","expired":"undefined","desc":"Application user"}], "acl_mnesia":[]}}}
```

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

Examples:

上传指定的数据文件：

```

1 $ curl -i --basic -u admin:public -X POST "http://localhost:8081/api/v4/data/file" -d '{"fil
2 ename":"emqx-export-2020-5-18-17-17-44.json","file":"{\"version\":\"dev\",\"users\":[{\"username\":
3 \":\"admin\"},\"tags\":{\"administrator\"},\"password\":\"oKQPB1hbigv6+2ntALELNOb1ff0=\",\"schemas\":[],\"rules\":[],\"resources\":[],\"date\":\"2020-05-18 17:17:44\",\"blacklist\":[],\"auth_m
esia\":[],\"apps\":[{\"status\":true,\"secret\":\"public\",\"name\":\"Default\",\"id\":\"admin\",
,\"expired\":\"undefined\",\"desc\":\"Application user\"}],\"acl_mnesia\":[]}"}'
4 {"code":0}

```

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

Examples:

删除指定的数据文件：

```

1 $ curl -i --basic -u admin:public -X DELETE "http://localhost:8081/api/v4/data/file/emqx-expo
2 rt-2020-5-18-17-17-44.json"
3 {"code":0}

```

规则

查询规则引擎的动作

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

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

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

DELETE /api/v4/rules/{rule_id}

删除规则。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

添加一个规则，对于所有匹配到主题 "t/a" 的消息，打印其规则运行参数。

```

1 $ curl -XPOST -d '{
2   "rawsql": "select * from \"t/a\"",
3   "actions": [
4     {
5       "name": "inspect",
6       "params": {
7         "a": 1
8       }
9     ],
10   "description": "test-rule"
11 }' --basic -u admin:public 'http://localhost:8081/api/v4/rules'
12 {"data": {"rawsql": "select * from \"t/a\"", "metrics": [{"speed_max": 0, "speed_last5m": 0.0, "speed": 0.0, "node": "emqx@127.0.0.1", "matched": 0}], "id": "rule:7fdb2c9e", "for": ["t/a"], "enabled": true, "description": "test-rule", "actions": [{"params": {"a": 1}, "name": "inspect", "metrics": [{"success": 0, "node": "emqx@127.0.0.1", "failed": 0}], "id": "inspect_1582434715354188116"}]}, "code": 0}

```

使用规则 ID 获取刚才创建的规则详情：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/rules/rule:7fdb2c9e'
2
3 {"data": {"rawsql": "select * from \"t/a\"", "metrics": [{"speed_max": 0, "speed_last5m": 0.0, "speed": 0.0, "node": "emqx@127.0.0.1", "matched": 0}], "id": "rule:7fdb2c9e", "for": ["t/a"], "enabled": true, "description": "test-rule", "actions": [{"params": {"a": 1}, "name": "inspect", "metrics": [{"success": 0, "node": "emqx@127.0.0.1", "failed": 0}], "id": "inspect_1582434715354188116"}]}, "code": 0}

```

获取所有的规则，注意返回值里的 **data** 是个规则对象的数组：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/rules'
2
3 {"data": [{"rawsql": "select * from \"t/a\"", "metrics": [{"speed_max": 0, "speed_last5m": 0.0, "speed": 0.0, "node": "emqx@127.0.0.1", "matched": 0}], "id": "rule:7fdb2c9e", "for": ["t/a"], "enabled": true, "description": "test-rule", "actions": [{"params": {"a": 1}, "name": "inspect", "metrics": [{"success": 0, "node": "emqx@127.0.0.1", "failed": 0}], "id": "inspect_1582434715354188116"}]}, "code": 0}

```

更新一下规则的 **SQL** 语句，改为 `select * from "t/b"`：

```

1 $ curl -XPUT --basic -u admin:public 'http://localhost:8081/api/v4/rules/rule:7fdb2c9e' -d '
2   {"rawsql": "select * from \"t/b\""}
3
4 {"data": {"rawsql": "select * from \"t/b\"", "metrics": [{"speed_max": 0, "speed_last5m": 0.0, "speed": 0.0, "node": "emqx@127.0.0.1", "matched": 0}], "id": "rule:7fdb2c9e", "for": ["t/a"], "enabled": true, "description": "test-rule", "actions": [{"params": {"a": 1}, "name": "inspect", "metrics": [{"success": 0, "node": "emqx@127.0.0.1", "failed": 0}], "id": "inspect_1582434715354188116"}]}, "code": 0}

```

停用规则 (**disable**)：

```

1 $ curl -XPUT --basic -u admin:public 'http://localhost:8081/api/v4/rules/rule:7fdb2c9e' -d '
2   {"enabled": false}
3
4 {"data": {"rawsql": "select * from \"t/b\"", "metrics": [{"speed_max": 0, "speed_last5m": 0.0, "speed": 0.0, "node": "emqx@127.0.0.1", "matched": 0}], "id": "rule:7fdb2c9e", "for": ["t/a"], "enabled": false, "description": "test-rule", "actions": [{"params": {"a": 1}, "name": "inspect", "metrics": [{"success": 0, "node": "emqx@127.0.0.1", "failed": 0}], "id": "inspect_1582434715354188116"}]}, "code": 0}

```

删除规则：

```

1 $ curl -XDELETE --basic -u admin:public 'http://localhost:8081/api/v4/rules/rule:7fdb2c9e'
2
3 {"code": 0}

```

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。

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

Examples:

查询 **inspect** 动作的详情：

```
1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/actions/inspect' sh
2
3 {"data": {"types": [], "title": {"zh": "检查 (调试)", "en": "Inspect (debug)"}, "params": {}, "name": "in-spect", "for": "$any", "description": {"zh": "检查动作参数 (用以调试)", "en": "Inspect the details of a-ction params for debug purpose"}, "app": "emqx_rule_engine"}, "code": 0}
```

查询当前所有的动作：

```
1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/actions' sh
2
3 {"data": [{"types": [], "title": {"zh": "空动作 (调试)", "en": "Do Nothing (debug)"}, "params": {}, "name": "do_nothing", "for": "$any", "description": {"zh": "此动作什么都不做，并且不会失败 (用以调试)", "en": "This action does nothing and never fails. It's for debug purpose"}, "app": "emqx_rule_engine"}, ...], "code": 0}
```

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。

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
<code>resource_type_name</code>	String	False	可选，资源类型名。如不指定 <code>resource_type_name</code> 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>code</code>	Integer	0
<code>data</code>	Object	规则对象
- <code>data.title</code>	Object	资源类型的简述，中英文。
- <code>data.params</code>	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
- <code>data.description</code>	Object	资源类型的描述信息，中英文。
- <code>data.provider</code>	String	资源类型的提供者

Examples:

查询 `web_hook` 资源类型的详细信息：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/resource_types/web_hook' sh
2
3 {"data": {"title": {"zh": "WebHook", "en": "WebHook"}, "provider": "emqx_web_hook", "params": {"url": {"type": "string", "title": {"zh": "请求 URL", "en": "Request URL"}, "required": true, "format": "url", "description": {"zh": "请求 URL", "en": "Request URL"}}, "method": {"type": "string", "title": {"zh": "请求方法", "en": "Request Method"}, "enum": ["PUT", "POST"], "description": {"zh": "请求方法", "en": "Request Method"}, "default": "POST"}, "headers": {"type": "object", "title": {"zh": "请求头", "en": "Request Header"}, "schema": {}, "description": {"zh": "请求头", "en": "Request Header"}, "default": {}}, "name": "web_hook", "description": {"zh": "WebHook", "en": "WebHook"}}, "code": 0}

```

查询当前所有的资源类型：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/resource_types' sh
2
3 {"data": [{"title": {"zh": "WebHook", "en": "WebHook"}, "provider": "emqx_web_hook", "params": {"url": {"type": "string", "title": {"zh": "请求 URL", "en": "Request URL"}, "required": true, "format": "url", "description": {"zh": "请求 URL", "en": "Request URL"}}, "method": {"type": "string", "title": {"zh": "请求方法", "en": "Request Method"}, "enum": ["PUT", "POST"], "description": {"zh": "请求方法", "en": "Request Method"}, "default": "POST"}, "headers": {"type": "object", "title": {"zh": "请求头", "en": "Request Header"}, "schema": {}, "description": {"zh": "请求头", "en": "Request Header"}, "default": {}}, "name": "web_hook", "description": {"zh": "WebHook", "en": "WebHook"}}, ...], "code": 0}

```

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。

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。

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。

DELETE /api/v4/resources/{resource_id}

删除资源。

Parameters: 无

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

创建一个 **webhook** 资源，**webserver** 的 **URL** 为 **http://127.0.0.1:9910**：

```

1 $ curl -XPOST -d '{
2   "type": "web_hook",
3   "config": {
4     "url": "http://127.0.0.1:9910",
5     "headers": {"token": "axfw34y235wrq234t4ersgw4t"},
6     "method": "POST"
7   },
8   "description": "web hook resource-1"
9 }' --basic -u admin:public 'http://localhost:8081/api/v4/resources'
10
11 {"data": {"type": "web_hook", "id": "resource:b12d3e44", "description": "web hook resource-1", "config": {"url": "http://127.0.0.1:9910", "method": "POST", "headers": {"token": "axfw34y235wrq234t4ersgw4t"}}, "code": 0}

```

使用资源 **ID** 查询刚创建的资源：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/resources/resource:b12d3e44'
2
3 {"data": {"type": "web_hook", "status": [{"node": "emqx@127.0.0.1", "is_alive": false}], "id": "resource:b12d3e44", "description": "web hook resource-1", "config": {"url": "http://127.0.0.1:9910", "method": "POST", "headers": {"token": "axfw34y235wrq234t4ersgw4t"}}, "code": 0}

```

查询当前已创建的所有资源：

```

1 $ curl --basic -u admin:public 'http://localhost:8081/api/v4/resources'
2
3 {"data": [{"type": "web_hook", "id": "resource:b12d3e44", "description": "web hook resource-1", "co
4 nfig": {"url": "http://127.0.0.1:9910", "method": "POST", "headers": {"token": "axfw34y235wrq234t4e
5 rsgw4t"}}, "code": 0}

```

删除资源:

```

1 $ curl -XDELETE --basic -u admin:public 'http://localhost:8081/api/v4/resources/resource:b12d
2 3e44'
3 {"code": 0}

```

License

EMQX 软件许可管理。

POST /api/v4/license/upload

将一个新的许可证文件上传到集群。许可证被验证，然后被复制到集群中的所有节点并重新加载。新的内容被写入节点中配置的相同的文件路径，旧的许可证内容被备份到一个文件，该文件后缀为发生变化时的时间戳。

Body (bytes)

要上传的许可证内容。

Success Response Body (JSON):

Name	Type	Description
code	Integer	0

Examples:

上传一个许可证文件。

```

1 $ curl -XPOST --basic -u admin:public -d @<(jq -sR '{license: .}' < path/to/new.license) 'ht
2 tp://localhost:8081/api/v4/license/upload'
3 {"code": 0}

```

配置项

Cluster

cluster.name

Type	Default
string	emqxcl

说明

集群名称。

cluster.proto_dist

Type	Optional Value	Default
enum	inet_tcp , inet6_tcp , inet_tls	inet_tcp

说明

分布式 Erlang 集群协议类型。可选值为：

- inet_tcp : 使用 **IPv4**
- inet6_tcp 使用 **IPv6**
- inet_tls : 使用 **TLS**, 需要与 node.ssl_dist_optfile 配置一起使用。

cluster.discovery

Type	Optional Value	Default
enum	manual , static , mcast , dns , etcd , k8s	manual

说明

集群节点发现方式。可选值为：

- manual : 手动加入集群
- static : 配置静态节点。配置几个固定的节点，新节点通过连接固定节点中的某一个来加入集群。
- mcast : 使用 **UDP** 多播的方式发现节点。
- dns : 使用 **DNS A** 记录的方式发现节点。
- etcd : 使用 **etcd** 发现节点。
- k8s : 使用 **Kubernetes** 发现节点。

cluster.autorebal

Type	Optional Value	Default
enum	on , off	on

说明

启用或关闭集群脑裂自动恢复机制。

cluster.autoclean

Type	Default
duration	5m

说明

指定多久之后从集群中删除离线节点。

cluster.static.seeds

Type	Default	Example
string	-	emqx1@192.168.0.100, emqx2@192.168.0.101

说明

当使用 **static** 方式集群时，指定固定的节点列表，多个节点间使用逗号 `,` 分隔。

cluster.mcast.addr

Type	Default
ipaddr	239.192.0.1

说明

当使用 **mcast** 方式集群时，指定多播地址。

cluster.mcast.ports

Type	Default
string	4369

说明

当使用 **mcast** 方式集群时，指定多播端口。如有多个端口使用逗号 `,` 分隔。

cluster.mcast iface

Type	Default
<code>ipaddr</code>	<code>0.0.0.0</code>

说明

当使用 **mcast** 方式集群时，指定节点发现服务需要绑定到本地哪个 IP 地址。

cluster.mcast ttl

Type	Default
<code>integer</code>	<code>255</code>

说明

当使用 **mcast** 方式集群时，指定多播的 Time-To-Live 值。

cluster.mcast loop

Type	Optional Value	Default
<code>enum</code>	<code>on</code> , <code>off</code>	<code>on</code>

说明

当使用 **mcast** 方式集群时，设置多播的报文是否投递到本地回环地址。

cluster.dns.name

Type	Default	Example
<code>string</code>	-	<code>mycluster.com</code>

说明

当使用 **dns** 方式集群时，指定 DNS A 记录的名字。**emqx** 会通过访问这个 **DNS A** 记录来获取 IP 地址列表，然后拼接 `cluster.dns.app` 里指定的 **APP** 名得到集群中所有节点的列表。

示例

设置 `cluster.dns.app = emqx`，并且配置了一个 **DNS**: `mycluster.com`，其指向 **3** 个 **IP** 地址：

1	192.168.0.100
2	192.168.0.101
3	192.168.0.102

则得到集群节点列表如下：

1	emqx@192.168.0.100
2	emqx@192.168.0.101
3	emqx@192.168.0.102

cluster.dns.app

Type	Default	Example
string	-	emqx

说明

当使用 **dns** 方式集群时，用来与从 `cluster.dns.name` 获取的 **IP** 列表拼接得到节点名列表。

cluster.etcd.server

Type	Default	Example
string	-	http://127.0.0.1:2379

说明

当使用 **etcd** 方式集群时，指定 **etcd** 服务的地址。如有多个服务使用逗号 `,` 分隔。

cluster.etcd.prefix

Type	Default	Example
string	-	emqxcl

说明

当使用 **etcd** 方式集群时，指定 **etcd** 路径的前缀。每个节点在 **etcd** 中都会创建一个路径：

1	v2/keys/<prefix>/<cluster.name>/<node.name>
---	---

cluster.etcd.node_ttl

Type	Default	Example
duration	-	1m

说明

当使用 **etcd** 方式集群时，指定 **etcd** 中节点路径的过期时间。

cluster.etcd.ssl.keyfile

Type	Default	Example
string	-	etc/certs/client-key.pem

说明

当使用 **SSL** 连接 **etcd** 时，指定客户端的私有 **Key** 文件。

cluster.etcd.ssl.certfile

Type	Default	Example
string	-	etc/certs/client.pem

说明

当使用 **SSL** 连接 **etcd** 时，指定 **SSL** 客户端的证书文件。

cluster.etcd.ssl.cacertfile

Type	Default	Example
string	-	etc/certs/ca.pem

说明

当使用 **SSL** 连接 **etcd** 时，指定 **SSL** 的 **CA** 证书文件。

cluster.k8s.apiserver

Type	Default	Example
string	-	http://10.110.111.204:8080

说明

当使用 **k8s** 方式集群时，指定 **Kubernetes API Server**。如有多个 **Server** 使用逗号 `,` 分隔。

cluster.k8s.service_name

Type	Default	Example
string	-	<code>emqx</code>

说明

当使用 **k8s** 方式集群时，指定 **Kubernetes** 中 **EMQX** 的服务名。

cluster.k8s.address_type

Type	Optional Value	Default
enum	<code>ip</code> , <code>dns</code> , <code>hostname</code>	<code>ip</code>

说明

当使用 **k8s** 方式集群时，**address_type** 用来从 **Kubernetes** 接口的应答里获取什么形式的 **Host** 列表。

示例

指定 `cluster.k8s.address_type` 为 `ip`，则将从 **Kubernetes** 接口中获取 **emqx** 服务的 **IP** 地址列表：

```

1 172.16.122.31
2 172.16.122.32
3 172.16.122.33

```

然后与 `cluster.k8s.app_name` 配置指定的 **app name** 拼接，得到 **emqx** 节点列表：

```

1 emqx@172.16.122.31
2 emqx@172.16.122.32
3 emqx@172.16.122.33

```

cluster.k8s.app_name

Type	Default	Example
string	-	<code>emqx</code>

说明

当使用 **k8s** 方式集群时，**app_name** 用来跟获取的 **Host** 列表拼接，得到节点列表。

cluster.k8s.suffix

Type	Default	Example
string	-	pod.cluster.local

说明

当使用 **k8s** 方式并且 `cluster.k8s.address_type` 指定为 **dns** 类型时，可设置 **emqx** 节点名的后缀。与 `cluster.k8s.namespace` 一起使用用以拼接得到节点名列表。

cluster.k8s.namespace

Type	Default	Example
string	-	default

说明

当使用 **k8s** 方式并且 `cluster.k8s.address_type` 指定为 **dns** 类型时，可设置 **emqx** 节点名的命名空间。与 `cluster.k8s.suffix` 一起使用用以拼接得到节点名列表。

示例

设置 `cluster.k8s.address_type` 为 **dns**，则将从 **Kubernetes** 接口中获取 **emqx** 服务的 **dns** 列表：

```

1 172-16-122-31
2 172-16-122-32
3 172-16-122-33

```

然后拼接上 `cluster.k8s.app_name = emqx`，`cluster.k8s.suffix = pod.cluster.local`，`cluster.k8s.namespace = default` 得到 **dns** 形式的 **emqx** 节点名列表：

```

1 emqx@172-16-122-31.default.pod.cluster.local
2 emqx@172-16-122-32.default.pod.cluster.local
3 emqx@172-16-122-33.default.pod.cluster.local

```

Node

node.name

Type	Default
string	emqx@127.0.0.1

说明

节点名。格式为 `<name>@<host>`。其中 `<host>` 可以是 IP 地址，也可以是 FQDN。详见 http://erlang.org/doc/reference_manual/distributed.html。

node.cookie

Type	Default
<code>string</code>	<code>emqxsecretcookie</code>

说明

分布式 Erlang 集群使用的 cookie 值。

node.data_dir

Type	Default
<code>folder</code>	<code>./data</code>

说明

节点的 data 目录，用于存放 Mnesia 数据文件等。

node.heartbeat

Type	Optional Value	Default
<code>enum</code>	<code>on , off</code>	<code>off</code>

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `-heart` 参数。

启用或关闭 Erlang 运行时检测机制，并在运行时终止时自动重启。需小心使用，以免手动关闭 emqx 时被监控进程重新启动。

node.async_threads

Type	Optional Value	Default
<code>integer</code>	<code>0 - 1024</code>	<code>4</code>

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `+A` 参数。

设置 **Erlang** 运行时异步线程池中的线程数量。详情请参见 <http://erlang.org/doc/man/erl.html>。

node.process_limit

Type	Optional Value	Default
integer	1024 - 134217727	2097152

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `+P` 参数。

设置 **Erlang** 允许的最大进程数，这将影响 **emqx** 节点能处理的连接数。详情请参见 <http://erlang.org/doc/man/erl.html>。

node.max_ports

Type	Optional Value	Default
integer	1024 - 134217727	1048576

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `+Q` 参数。

设置 **Erlang** 允许的最大 **Ports** 数量。详情请参见 <http://erlang.org/doc/man/erl.html>。

node.dist_buffer_size

Type	Optional Value	Default
bytesize	1KB - 2GB	8MB

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `+zdbbl` 参数。

设置 **Erlang** 分布式通信使用的最大缓存大小。详情请参见 <http://erlang.org/doc/man/erl.html>。

node.max_ets_tables

Type	Default
integer	262144

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `+e` 参数。

设置 **Erlang** 运行时允许的最大 **ETS** 表数量。详情请参见 <http://erlang.org/doc/man/erl.html>。

node.global_gc_interval

Type	Default
duration	<code>15m</code>

说明

系统调优参数，设置 **Erlang** 运行多久强制进行一次全局垃圾回收。

node.fullsweep_after

Type	Optional Value	Default
integer	<code>0 - 65535</code>	<code>1000</code>

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `-env ERL_FULLSWEEP_AFTER` 参数。

设置 **Erlang** 运行时多少次 **generational GC** 之后才进行一次 **fullsweep GC**。详情请参见 http://erlang.org/doc/man/erlang.html#spawn_opt-4。

node.crash_dump

Type	Default
string	<code>log/crash.dump</code>

说明

设置 **Erlang crash_dump** 文件的存储路径和文件名。

node.ssl_dist_optfile

Type	Default
string	<code>etc/ssl_dist.conf</code>

说明

此配置将覆盖 `vm.args` 文件里的 `-ssl_dist_optfile` 参数。

如使用 **SSL** 方式建立 **emqx** 集群，需指定 **SSL** 分布式协议的配置文件。需要与 `cluster.proto_dist = inet_tls` 一起使用。

node.dist_net_ticktime

Type	Default
<code>integer</code>	<code>120</code>

说明

系统调优参数，此配置将覆盖 `vm.args` 文件里的 `-kernel net_ticktime` 参数。

当一个节点持续无响应多久之后，认为其已经宕机并断开连接。详情请参见 http://www.erlang.org/doc/man/kernel_app.html#net_ticktime。

node.dist_listen_min

Type	Optional Value	Default
<code>integer</code>	<code>1024 - 65535</code>	<code>6369</code>

说明

与 `node.dist_listen_max` 一起设定一个 **TCP** 端口段，此端口段用于分配给分布式 **Erlang**，作为分布式通道的监听端口。注意如果在节点之间设置了防火墙，需要将此端口段放进防火墙的端口白名单里。

node.dist_listen_max

Type	Optional Value	Default
<code>integer</code>	<code>1024 - 65535</code>	<code>6369</code>

说明

与 `node.dist_listen_min` 一起设定一个 **TCP** 端口段，此端口段用于分配给分布式 **Erlang**，作为分布式通道的监听端口。注意如果在节点之间设置了防火墙，需要将此端口段放进防火墙的端口白名单里。

RPC

rpc.mode

Type	Optional Value	Default
<code>enum</code>	<code>sync , async</code>	<code>async</code>

说明

RPC 模式。可选同步或异步模式。

rpc.async_batch_size

Type	Default
integer	256

说明

异步模式下最大的批量发送消息数。注意此配置在同步模式下不起作用。

rpc.port_discovery

Type	Optional Value	Default
enum	manual , stateless	

说明

`manual` : 手动指定服务器客户端的端口号 `tcp_server_port` and `tcp_client_port`. `stateless` : discover ports in a stateless manner. If node name is `emqx<N>@127.0.0.1`, where the `<N>` is an integer, then the listening port will be `5370 + <N>`

Default is `manual` when started from docker (environment variable override from docker-entrypoint) otherwise `stateless`.

node.tcp_server_port

Type	Optional Value	Default
integer	1024 - 65535	5369

说明

设置 **RPC** 本地服务使用的监听 **port**。注意，该配置仅在 `rpc.port_discovery` 设置成 `manual` 时有效

rpc.tcp_client_num

Type	Optional Value	Default
integer	1 - 256	CPU 核心数 / 2

说明

设置由本节点发起，通往每个远程节点的 **RPC** 通信通道数量。设置为 **1** 可保证消息顺序。保持默认值（**CPU** 核心数的一半）可提高 **RPC** 的吞吐能力。

rpc.connect_timeout

Type	Default
duration	5s

说明

建立 **RPC** 连接超时时间。建立连接时若远程节点无响应，多久之后放弃尝试。

rpc.send_timeout

Type	Default
duration	5s

说明

发送超时时间。发送消息多久之后放弃。

rpc.authentication_timeout

Type	Default
duration	5s

说明

RPC 认证超时时间。尝试认证若远程节点无响应，多久之后放弃。

rpc.call_receive_timeout

Type	Default
duration	15s

说明

RPC 同步模式的超时时间。**RPC** 同步调用若收不到回复，用多久之后放弃。

rpc.socket_keepalive_idle

Type	Default
duration	900s

说明

在最近一次数据包发送多久之后，发送 **keepalive** 探测报文。

rpc.socket_keepalive_interval

Type	Default
duration	75s

说明

发送 **keepalive** 探测报文的间隔。

rpc.socket_keepalive_count

Type	Default
integer	9

说明

连续多少次 **keepalive** 探测报文都收不到回复的情况下，认为 **RPC** 连接已丢失。

rpc.socket_sndbuf

Type	Default
bytesize	1MB

说明

TCP 调优参数。**TCP** 发送缓冲区大小。

rpc.socket_recbuf

Type	Default
bytesize	1MB

说明

TCP 调优参数。**TCP** 接收缓冲区大小。

rpc.socket_buffer

Type	Default
bytesize	1MB

说明

TCP 调优参数。用户态的 **Socket** 缓冲区大小。

Log

log.to

Type	Optional Value	Default
enum	off, file, console, both	file

说明

将日志输出到什么地方。可选值为：

- **off**: 完全关闭日志功能
- **file**: 仅将日志输出到文件
- **console**: 仅将日志输出到标准输出(emqx 控制台)
- **both**: 同时将日志输出到文件和标准输出(emqx 控制台)

log.level

Type	Optional Value	Default
enum	debug, info, notice, warning error, critical, alert, emergency	warning

说明

全局的日志级别。这包括 **primary log level** 以及所有的 **log handlers**。详情请参见 [日志级别和 log handlers](#)。

log.dir

Type	Default
dir	./log

说明

日志文件目录。

log.file

Type	Default
string	emqx.log

说明

日志文件的前缀。例如，若使用默认值 (`log.file = emqx.log`)，日志文件名将为
`emqx.log.1` , `emqx.log.2` , ...。

log.chars_limit

Type	Default
integer	-1

说明

设置单个日志消息的最大长度。如超过此长度，日志消息将被截断。`-1` 表示无限制。

log.max_depth

Type	Default
union(integer, 'unlimited')	20

Description

控制 Erlang 数据结构的打印深度，和 Erlang 进程消息队列查看的深度。或配置成 '**'unlimited'** (不带引号) 不限深度打印。

log.rotation.size

Type	Default
bytesize	10MB

说明

设置单个日志文件大小。如超过此大小，则进行日志文件滚动，创建新的日志文件。

log.rotation.count

Type	Default
integer	5

说明

设置日志文件总个数。如超过此文件个数，则下一次日志文件滚动将会覆盖第一个文件。

log.<level>.file

Type	Default
string	-

说明

针对某日志级别设置单独的日志文件。

示例

将 **info** 及 **info** 以上的日志单独输出到 `info.log.N` 文件中：

```
1 log.info.file = info.log
```

将 **error** 及 **error** 以上的日志单独输出到 `error.log.N` 文件中

```
1 log.error.file = error.log
```

log.max_depth

Type	Default
integer	20

说明

控制对大的数据结构打印日志时的最大深度。超过深度的部分将被 '...' 代替。

log.single_line

Type	Default
boolean	true

说明

设置成 `true` 时，单行打印日志。如果设置成 `false`，如 `crash` 日志中的堆栈信息等将打印多行

log.formatter

Type	Optional Value	Default
enum	text , json	text

说明

选择打印日志的格式

log.formatter.text.date.format

Type	Optional Value	Default
enum	rfc3339	FORMAT

注意: 这个配置在 EMQX 开源版 **4.3.15, 4.4.4** 和 EMQX 企业版 **4.3.10, 4.4.4** 及之后可以使用。

说明

指定 `text` `logger` 的时间戳格式。可以是 `rfc3339` 或者 `FORMAT` 字符串。

其中 `FORMAT` 里支持的控制符如下:

控制符	说明	格式示例
%Y	年	2022
%m	月 (01..12)	11
%d	日	01
%H	时 (00..23)	06
%M	分 (00..59)	43
%S	秒 (00..60)	31
%N	纳秒 (000000000..999999999)	019085000
%6N	微秒 (00000..999999)	019085
%3N	毫秒 (000..999)	019
%z	+HHMM 数字时区	-0400
%:z	+HH:MM 数字时区	-04:00
%::z	+HH:MM:SS 数字时区	-04:00:00

举例:

```

1 ## 2022-06-02T14:23:09.230000 +08:00
2 log.formatter.text.date.format = %Y-%m-%dT%H:%M:%S.%6N %:z
3
4 ## 如下配置可以让时间戳跟 4.2.x 版本的一样 (2022-06-02 14:24:36.124):
5 log.formatter.text.date.format = %Y-%m-%d %H:%M:%S.%3N

```

authacl

allow_anonymous

Type	Optional Value	Default
enum	true , false	true

说明

是否允许匿名用户登录系统。

注：生产环境建议关闭此选项。

acl_nomatch

Type	Optional Value	Default
enum	allow , deny	allow

说明

ACL 未命中时，允许或者拒绝 发布/订阅 操作。

acl_file

Type	Default
string	etc/acl.conf

说明

默认 **ACL** 文件的路径。

enable_acl_cache

Type	Optional Value	Default
enum	on , off	on

说明

是否启用 **ACL** 缓存。

acl_cache_max_size

Type	Default
integer	32

说明

ACL 规则最大缓存条数。

acl_cache_ttl

Type	Default
duration	1m

说明

ACL 规则最大缓存时间。

acl_deny_action

Type	Optional Value	Default
enum	ignore , disconnect	ignore

说明

ACL 检查失败后，执行的操作。

- ignore : 不做任何操作。
- disconnect : 断开连接。

mqtt

flapping_detect_policy

Type	Default
string	30, 1m, 5m

说明

指定 Flapping 检查策略。

格式: <threshold>,<duration>,<banned>。

例如, 30, 1m, 5m , 它表示如果客户端在 1 分钟内断开连接 30 次, 那么在后续 5 分钟内禁止登录。

mqtt.max_packet_size

Type	Default
bytesize	1MB

说明

允许的 MQTT 报文最大长度。

mqtt.max_clientid_len

Type	Default
integer	65535

说明

允许的 **Client ID** 串的最大长度。

mqtt.max_topic_levels

Type	Default
integer	128

说明

允许客户端订阅主题的最大层级。**0** 表示不限制。

Warning

Topic层级过多可能导致订阅时的性能问题。

mqtt.max_qos_allowed

Type	Optional Value	Default
enum	0 , 1 , 2	2

说明

允许客户端发布的最大 **QoS** 等级。

mqtt.max_topic_alias

Type	Default
integer	65535

说明

允许最大的主题别名数。**0** 表示不支持主题别名。

mqtt.retain_available

Type	Optional Value	Default
enum	true , false	true

说明

是否支持 **Retain** 消息。

mqtt.wildcard_subscription

Type	Optional Value	Default
enum	true , false	true

说明

是否支持订阅通配主题。

mqtt.shared_subscription

Type	Optional Value	Default
enum	true , false	true

说明

是否支持共享订阅。

mqtt.ignore_loop_deliver

Type	Optional Value	Default
enum	true , false	false

说明

是否忽略自己发送的消息。如果忽略，则表明 **EMQX** 不会向消息的发送端投递此消息。

mqtt.strict_mode

Type	Optional Value	Default
enum	true , false	false

说明

是否开启严格检查模式。严格检查模式会更细致的检查 **MQTT** 报文的正确性。

zoneexternal

zone.external.idle_timeout

Type	Default
duration	15s

说明

TCP 连接建立后的发呆时间，如果这段时间内未收到任何报文，则会关闭该连接。

zone.external.enable_acl

Type	Optional Value	Default
enum	on , off	on

说明

是否开启 **ACL** 检查。

zone.external.enable_ban

Type	Optional Value	Default
enum	on , off	on

说明

是否开启黑名单。

zone.external.enable_stats

Type	Optional Value	Default
enum	on , off	on

说明

是否开启客户端状态统计。

zone.external.acl_deny_action

Type Optional Value Default	---- ----- - -----	enum	ignore , disconnect
ignore			

说明

ACL 检查失败后，执行的操作。

- `ignore` : 不做任何操作。
- `disconnect` : 断开连接。

zone.external.force_gc_policy

Type	Default
<code>string</code>	<code>16000 16MB</code>

说明

当收到一定数量的消息，或字节，就强制执行一次垃圾回收。

格式：`<Number> | <Bytes>`。

例如，`16000 | 16MB` 表示当收到 `16000` 条消息，或 `16MB` 的字节流入就强制执行一次垃圾回收。

zone.external.force_shutdown_policy

Type	Default
<code>string</code>	-

说明

当进程消息队列长度，或占用的内存字节到达某值，就强制关闭该进程。

这里的 `消息队列` 指的是 **Erlang** 进程的 `消息邮箱`，并非 **QoS 1** 和 **QoS 2** 的 `mqueue`。

格式：`<Number> | <Bytes>`。

例如，`32000 | 32MB` 表示当进程堆积了 `32000` 条消息，或进程占用内存达到 `32MB` 则关闭该进程。

zone.external.max_packet_size

Type	Default
<code>bytesize</code>	-

说明

允许的 **MQTT** 报文最大长度。

zone.external.max_clientid_len

Type	Default
<code>integer</code>	-

说明

允许的 **Client ID** 串的最大长度。

zone.external.max_topic_levels

Type	Default
integer	-

说明

允许客户端订阅主题的最大层级。**0** 表示不限制。

Warning

Topic层级过多可能导致订阅时的性能问题。

zone.external.max_qos_allowed

Type	Optional Value	Default
enum	0 , 1 , 2	-

说明

允许客户端发布的最大 **QoS** 等级。

zone.external.max_topic_alias

Type	Default
integer	-

说明

允许最大的主题别名数。**0** 表示不支持主题别名。

zone.external.retain_available

Type	Optional Value	Default
enum	true , false	-

说明

是否支持 **Retain** 消息。

zone.external.wildcard_subscription

Type	Optional Value	Default
enum	true , false	-

说明

是否支持订阅通配主题。

zone.external.shared_subscription

Type	Optional Value	Default
enum	true , false	-

说明

是否支持共享订阅。

zone.external.server_keepalive

Type	Default
integer	-

说明

服务端指定的 **Keepalive** 时间。用于 **MQTT v5.0** 协议的 **CONNACK** 报文。

zone.external.keepalive_backoff

Type	Optional Value	Default
float	> 0.5	0.75

说明

Keepalive 退避指数。**EMQX** 如果在 `Keepalive * backoff * 2` 的时间内未收到客户端的任何数据报文，则认为客户端已心跳超时。

zone.external.max_subscriptions

Type	Default
integer	0

说明

单个客户端允许订阅的最大主题数。`0` 表示不限制。

zone.external.upgrade_qos

Type	Optional Value	Default
enum	on , off	off

说明

允许 **EMQX** 在投递消息时，强制升级消息的 **QoS** 等级为订阅的 **QoS** 等级。

zone.external.max_inflight

Type	Default
integer	32

说明

飞行窗口大小。飞行窗口用于存储未被应答的 **QoS 1** 和 **QoS 2** 消息。

zone.external.retry_interval

Type	Default
duration	30s

说明

消息重发间隔。**EMQX** 在每个间隔检查是否需要进行消息重发。

zone.external.max_awaiting_rel

Type	Default
integer	100

说明

QoS 2 消息的最大接收窗口，配置 **EMQX** 能够同时处理多少从客户端发来的 **QoS 2** 消息。**0** 表示不限制。

zone.external.await_rel_timeout

Type	Default
duration	300s

说明

QoS 2 消息处理超时时间，在超时后若还未收到 **QoS** 的 **PUBREL** 报文，则将消息从接收窗口中丢弃。

zone.external.session_expiry_interval

Type	Default
duration	2h

说明

会话默认超时时间，主要用于 **MQTT v3.1** 和 **v3.1.1** 协议。在 **MQTT v5.0** 中，该值通常会携带在客户端的连接报文中。

zone.external.max_mqueue_len

Type	Default
integer	1000

说明

消息队列最大长度。当飞行窗口满，或客户端离线后，消息会被存储至该队列中。**0** 表示不限制。

zone.external.mqueue_priorities

Type	Optional Value	Default
string	none , <Spec>	none

说明

队列消息优先级配置：

- **none**：表示无优先级区分。
- **<Spec>**：表示为一个消息优先表，它配置了某主题下消息的优先级。例如：

- `topic/1=10` : 表示主题 `topic/1` 的消息优先级为 `10`。
- `topic/1=10,topic/2=8` : 表示配置了两个主题的优先级，其分别为 `10` 和 `8`。
- 其中，优先级数值越高，优先等级越高。

当消息队列长度有限时，会优先丢弃低优先级的消息。

zone.external.mqueue_default_priority

Type	Optional Value	Default
<code>enum</code>	<code>highest</code> , <code>lowest</code>	<code>highest</code>

说明

消息默认的优先等级。

zone.external.mqueue_store_qos0

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	<code>true</code>

说明

消息队列是否存储 **QoS 0** 消息。

zone.external.enable_flapping_detect

Type	Optional Value	Default
<code>enum</code>	<code>on</code> , <code>off</code>	<code>off</code>

说明

是否开启 `Flapping` 检查。

zone.external.mountpoint

Type	Default
<code>string</code>	-

说明

主题挂载点。配置后，所有订阅和发布的主题在 **EMQX** 都会为其增加一个前缀。

其中可用的占位符有：

- `%c` : 表示客户端的 **Client ID**。
- `%u` : 表示客户端的 **Username**。

例如，配置挂载点为 `user/%c/`。那么 **Client ID** 为 `tom` 的客户端在发布主题 `open` 消息时，实际在 **EMQX** 中路由的主题是 `user/tom/open`。

zone.external.use_username_as_clientid

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>false</code>

说明

是否用客户端的 **Username** 作为其 **Client ID**。

zone.external.ignore_loop_deliver

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>false</code>

说明

是否忽略自己发送的消息。如果忽略，则表明 **EMQX** 不会向消息的发送端投递此消息。

zone.external.strict_mode

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>false</code>

说明

是否开启严格检查模式。严格检查模式会更细致的检查 **MQTT** 报文的正确性。

zoneinternal

zone.internal.allow_anonymous

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>true</code>

说明

是否允许匿名用户登录系统。

zone.internal.enable_stats

Type	Optional Value	Default
enum	on , off	on

说明

是否开启客户端状态统计。

zone.internal.enable_acl

Type	Optional Value	Default
enum	on , off	off

说明

是否开启 **ACL** 检查。

zone.internal.acl_deny_action

Type	Optional Value	Default
enum	ignore , disconnect	ignore

说明

ACL 检查失败后，执行的操作。

- `ignore` : 不做任何操作。
- `disconnect` : 断开连接。

zone.internal.force_gc_policy

Type	Default
string	-

说明

当收到一定数量的消息，或字节，就强制执行一次垃圾回收。

格式：`<Number>|<Bytes>`。

例如，`16000|16MB` 表示当收到 `16000` 条消息，或 `16MB` 的字节流入就强制执行一次垃圾回收。

zone.internal.wildcard_subscription

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	-

说明

是否支持订阅通配主题。

zone.internal.shared_subscription

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	-

说明

是否支持共享订阅。

zone.internal.max_subscriptions

Type	Default
<code>integer</code>	<code>0</code>

说明

单个客户端允许订阅的最大主题数。`0` 表示不限制。

zone.internal.max_inflight

Type	Default
<code>integer</code>	<code>128</code>

说明

飞行窗口大小。飞行窗口用于存储未被应答的 **QoS 1** 和 **QoS 2** 消息。

zone.internal.max_awaiting_rel

Type	Default
integer	1000

说明

QoS 2 消息的最大接收窗口，配置 EMQX 能够同时处理多少从客户端发来的 **QoS 2** 消息。`0` 表示不限制。

zone.internal.max_mqueue_len

Type	Default
integer	10000

说明

消息队列最大长度。当飞行窗口满，或客户端离线后，消息会被存储至该队列中。`0` 表示不限制。

zone.internal.mqueue_store_qos0

Type	Optional Value	Default
enum	true , false	true

说明

消息队列是否存储 **QoS 0** 消息。

zone.internal.enable_flapping_detect

Type	Optional Value	Default
enum	on , off	off

说明

是否开启 Flapping 检查。

zone.internal.force_shutdown_policy

Type	Default
string	-

说明

当进程消息队列长度，或占用的内存字节到达某值，就强制关闭该进程。

这里的 消息队列 指的是 Erlang 进程的 消息邮箱，并非 QoS 1 和 QoS 2 的 mqueue。

格式：`<Number>|<Bytes>`。

例如，`32000|32MB` 表示当进程堆积了 `32000` 条消息，或进程占用内存达到 `32MB` 则关闭该进程。

zone.internal.mountpoint

Type	Default
<code>string</code>	-

说明

主题挂载点。配置后，所有订阅和发布的主题在 EMQX 都会为其增加一个前缀。

其中可用的占位符有：

- `%c`：表示客户端的 Client ID。
- `%u`：表示客户端的 Username。

例如，配置挂载点为 `user/%c/`。那么 Client ID 为 `tom` 的客户端在发布主题 `open` 消息时，实际在 EMQX 中路由的主题是 `user/tom/open`。

zone.internal.ignore_loop_deliver

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	<code>false</code>

说明

是否忽略自己发送的消息。如果忽略，则表明 EMQX 不会向消息的发送端投递此消息。

zone.internal.strict_mode

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	<code>false</code>

说明

是否开启严格检查模式。严格检查模式会更细致的检查 MQTT 报文的正确性。

zone.internal.bypass_auth_plugins

Type	Optional Value	Default
enum	true , false	true

说明

是否允许该 **Zone** 下的客户端绕过认证插件的认证步骤。

tcpexternal

listener.tcp.external

Type	Default
string	0.0.0.0:1883

说明

配置名称为 `external` 的 **MQTT/TCP** 监听器的监听地址。

示例

`1883` : 表监听 **IPv4** 的 `0.0.0.0:1883` 。 `127.0.0.1:1883` : 表监听地址为 `127.0.0.1` 网卡上的 `1883` 端口。 `::1:1883` : 表监听 **IPv6** 地址为 `::1` 网卡上的 `1883` 端口。

listener.tcp.external.acceptors

Type	Default
integer	8

说明

监听器的接收池大小。

listener.tcp.external.max_connections

Type	Default
integer	1024000

说明

监听器允许的最大并发连接数量。

listener.tcp.external.max_conn_rate

Type	Default
integer	1000

说明

监听器允许的最大接入速率。单位：个/秒

listener.tcp.external.active_n

Type	Default
integer	100

说明

监听器持续接收 **TCP** 报文的次数。

listener.tcp.external.zone

Type	Default
string	external

说明

监听器所属的配置域 (**Zone**)。

listener.tcp.external.rate_limit

Type	Default
string	-

说明

监听器的速率限制。格式为 `<limit>,<duration>`。

示例

`100KB,10s` : 表 限制 10 秒内的流入字节数不超过 100 KB。

listener.tcp.external.access.1

Type	Default
string	allow all

说明

监听器的 **ACL** 规则列表。它用于设置连接层的白/黑名单。

示例

`allow all` : 表允许所有的 **TCP** 连接接入。 `allow 192.168.0.0/24` : 表允许网络地址为 `192.168.0.0/24` 的 **TCP** 连接接入。

同时，该配置可配置多条规则：

```
1   listener.tcp.external.access.1 = deny 192.168.0.1
2   listener.tcp.external.access.2 = allow all
```

它表示，除 `192.168.0.1` 外的 **TCP** 连接都允许接入。

listener.tcp.external.proxy_protocol

Type	Optional Value	Default
enum	on , off	-

说明

监听器是否开启 **Proxy Protocol** 的支持。

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，且需要拿到客户端真实的源 IP 地址与端口，则需打开此配置。

Proxy Protocol 参考: <https://www.haproxy.com/blog/haproxy/proxy-protocol>。

listener.tcp.external.proxy_protocol_timeout

Type	Default
duration	-

说明

设置 **Proxy Protocol** 解析的超时时间。如果该时间内没收到 **Proxy Protocol** 的报文，**EMQX** 会关闭其连接。

listener.tcp.external.peer_cert_as_username

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **Username** 字段的值。其可选值为：

- **cn**: 客户端证书的 **Common Name** 字段值
- **dn**: 客户端证书的 **Subject Name** 字段值
- **crt**: **DER** 格式编码的客户端证书二进制
- **pem**: 基于 **DER** 格式上的 **base64** 编码后的字符串
- **md5**: **DER** 格式证书的 **MD5** 哈希值

注意：在 **TCP** 的监听器下，该配置仅在负载均衡服务器终结 **SSL** 的部署情况下可以用；且负载均衡服务器需要配置 **Proxy Protocol** 将证书域的内容给发送至 **EMQX**。例如 **HAProxy** 的配置可参考 [send-proxy-v2-ssl](#)

listener.tcp.external.peer_cert_as_clientid

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **ClientID** 字段的值。其可选值的含义同上。

listener.tcp.external.backlog

Type	Default
integer	1024

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

listener.tcp.external.send_timeout

Type	Default
duration	15s

说明

TCP 报文发送超时时间。

listener.tcp.external.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

TCP 报文发送超时后，是否关闭该连接。

listener.tcp.external.redbuf

Type	Default
bytesize	-

说明

TCP 接收缓存区大小（操作系统内核级参数）

参见：<http://erlang.org/doc/man/inet.html>

listener.tcp.external.sndbuf

Type	Default
bytesize	-

说明

TCP 发送缓存区大小（操作系统内核级参数）。

参见：<http://erlang.org/doc/man/inet.html>。

listener.tcp.external.buffer

Type	Default
bytesize	-

说明

TCP 缓冲区大小（用户级）。

该值建议大于等于 `sndbuff` 和 `recbuff` 的最大值，以避免一些性能问题。在不配置的情况下，它默认等于 `sndbuff` 和 `recbuff` 的最大值。

参见：<http://erlang.org/doc/man/inet.html>。

listener.tcp.external.tune_buffer

Type	Optional Value	Default
enum	on , off	-

说明

如果打开此配置, 请设置该值等于 `sndbuff` 与 `recbuff` 的最大值。

listener.tcp.external.nodelay

Type	Optional Value	Default
enum	true , false	true

说明

即 `TCP_NODELAY` 参数。开启该选项即允许小的 **TCP** 数据报文将会立即发送。

listener.tcp.external.reuseaddr

Type	Optional Value	Default
enum	true , false	true

说明

即 `SO_REUSEADDR` 参数。开启该选项即允许本地重用端口, 无需等待 `TIME_WAIT` 状态结束。

tcpinternal

listener.tcp.internal

Type	Default
string	127.0.0.1:11883

说明

配置名称为 `internal` 的 **MQTT/TCP** 监听器的监听地址。

示例

`11883` : 表监听 **IPv4** 的 `0.0.0.0:11883` 。 `127.0.0.1:11883` : 表监听地址为 `127.0.0.1` 网卡上的 `11883` 端口。 `::1:11883` : 表监听 **IPv6** 地址为 `::1` 网卡上的 `11883` 端口。

listener.tcp.internal.acceptors

Type	Default
integer	4

说明

监听器的接收池大小。

listener.tcp.internal.max_connections

Type	Default
integer	1024000

说明

监听器允许的最大并发连接数量。

listener.tcp.internal.max_conn_rate

Type	Default
integer	1000

说明

监听器允许的最大接入速率。单位：个/秒

listener.tcp.internal.active_n

Type	Default
integer	1000

说明

监听器持续接收 **TCP** 报文的次数。

listener.tcp.internal.zone

Type	Default
string	internal

说明

监听器所属的配置域 (**Zone**)。

listener.tcp.internal.rate_limit

Type	Default
string	-

说明

监听器的速率限制。格式为 `<limit>, <duration>`。

示例

`100KB, 10s` : 表 **限制 10 秒内的流入字节数不超过 100 KB**。

listener.tcp.internal.backlog

Type	Default
integer	512

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

listener.tcp.internal.send_timeout

Type	Default
duration	5s

说明

TCP 报文发送超时时间。

listener.tcp.internal.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

TCP 报文发送超时后，是否关闭该连接。

listener.tcp.internal.redbuf

Type	Default
bytesize	64KB

说明

TCP 接收缓存区大小 (操作系统内核级参数)

listener.tcp.internal.sndbuf

Type	Default
bytesize	64KB

说明

TCP 发送缓存区大小 (操作系统内核级参数)

listener.tcp.internal.buffer

Type	Default
bytesize	-

说明

TCP 缓冲区大小 (用户级)。

listener.tcp.internal.tune_buffer

Type	Optional Value	Default
enum	on , off	-

说明

如果打开此配置, 请设置该值等于 `sndbuf` 与 `recbuff` 的最大值。

listener.tcp.internal.nodelay

Type	Optional Value	Default
enum	true , false	false

说明

即 `TCP_NODELAY` 参数。开启该选项即允许小的 **TCP** 数据报文将会立即发送。

listener.tcp.internal.reuseaddr

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	<code>true</code>

说明

即 `SO_REUSEADDR` 参数。开启该选项即允许本地重用端口，无需等待 `TIME_WAIT` 状态结束。

tlsexternal

listener.ssl.external

Type	Default
<code>string</code>	<code>0.0.0.0:8883</code>

说明

配置名称为 `external` 的 **SSL** 监听器。

listener.ssl.external.acceptors

Type	Default
<code>integer</code>	<code>16</code>

说明

监听器的接收池大小。

listener.ssl.external.max_connections

Type	Default
<code>integer</code>	<code>102400</code>

说明

监听器允许的最大并发连接数量。

listener.ssl.external.max_conn_rate

Type	Default
integer	500

说明

监听器允许的最大接入速率。单位：个/秒。

listener.ssl.external.active_n

Type	Default
integer	100

说明

监听器持续接收 **TCP** 报文的次数。

listener.ssl.external.zone

Type	Default
string	external

说明

监听器所属的配置组 (**Zone**)。

listener.ssl.external.access.1

Type	Default
string	allow all

说明

监听器的 **ACL** 规则列表。它用于设置连接层的白/黑名单。

例如：

`allow all` : 表允许所有的 **TCP** 连接接入。 `allow 192.168.0.0/24` : 表允许网络地址为 `192.168.0.0/24` 的 **TCP** 连接接入。

同时，该配置可配置多条规则：

```

1   listener.ssl.external.access.1 = deny 192.168.0.1
2   listener.ssl.external.access.2 = allow all

```

listener.ssl.external.rate_limit

Type	Default
string	-

说明

监听器的速率限制。格式为 `<limit>,<duration>`。

listener.ssl.external.proxy_protocol

Type	Optional Value	Default
enum	on , off	-

说明

监听器是否开启 `Proxy Protocol` 的支持。

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，且需要拿到客户端真实的源 IP 地址与端口，则需打开此配置。

`Proxy Protocol` 参考: <https://www.haproxy.com/blog/haproxy/proxy-protocol>。

listener.ssl.external.proxy_protocol_timeout

Type	Default
duration	-

说明

设置 `Proxy Protocol` 解析的超时时间。如果该时间内没收到 `Proxy Protocol` 的报文，**EMQX** 会关闭其连接。

listener.ssl.external.tls_versions

Type	Default
string	tlsv1.3,tlsv1.2,tlsv1.1,tlsv1

说明

指定服务端支持的 `SSL` 的版本列表。详情请参见 <http://erlang.org/doc/man/ssl.html>。

listener.ssl.external.handshake_timeout

Type	Default
duration	15s

说明

指定 **SSL** 握手过程的超时时间。

listener.ssl.external.depth

Type	Default
number	10

说明

证书链中非自签发的中间证书的最大数量。如果该值为 **0** 则表示，对端证书必须是根 **CA** 直接授信的。

listener.ssl.external.key_password

Type	Default
string	-

说明

证书密钥文件的密码。如果你的证书密钥设置了密码，则需要配置该选项。

listener.ssl.external.keyfile

Type	Default
string	etc/certs/key.pem

说明

指定 **SSL** 的私钥文件 (**PEM**)。

listener.ssl.external.certfile

Type	Default
string	etc/certs/cert.pem

说明

指定 **SSL** 的证书文件 (**PEM**)。

listener.ssl.external.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

指定 **SSL** 的 **CA** 证书文件 (**PEM**)。该文件应包含发布服务器证书的所有中间**CA**证书以及根证书。该文件还应包含所有受信**CA**的证书用以用于验证客户端的证书。

listener.ssl.external.dhfile

Type	Default
string	etc/certs/dh-params.pem

说明

若使用 **Ephemeral Diffie-Helman** 算法，指定算法使用的 **key** 文件。

listener.ssl.external.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	verify_peer

说明

指定握手过程中是否校验客户端。

listener.ssl.external.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	false

说明

SSL 握手过程中若客户端没有证书，是否让握手失败。

listener.ssl.external.ciphers

Type	Default
string	ECDHE–ECDSA–AES256–GCM–SHA384, ECDHE–RSA–AES256–GCM–SHA384, ECDHE–ECDSA–AES256–SHA384, ECDHE–RSA–AES256–SHA384, ECDHE–ECDSA–DES–CBC3–SHA, ECDH–ECDSA–AES256–GCM–SHA384, ECDH–RSA–AES256–GCM–SHA384, ECDH–ECDSA–AES256–SHA384, ECDH–RSA–AES256–SHA384, DHE–DSS–AES256–GCM–SHA384, DHE–DSS–AES256–SHA256, AES256–GCM–SHA384, AES256–SHA256, ECDHE–ECDSA–AES128–GCM–SHA256, ECDHE–RSA–AES128–GCM–SHA256, ECDHE–ECDSA–AES128–SHA256, ECDHE–RSA–AES128–SHA256, ECDH–ECDSA–AES128–GCM–SHA256, ECDH–RSA–AES128–GCM–SHA256, ECDH–ECDSA–AES128–SHA256, ECDH–RSA–AES128–SHA256, DHE–DSS–AES128–GCM–SHA256, DHE–DSS–AES128–SHA256, AES128–GCM–SHA256, AES128–SHA256, ECDHE–ECDSA–AES256–SHA, ECDHE–RSA–AES256–SHA, ECDH–ECDSA–AES256–SHA, AES256–SHA, ECDHE–ECDSA–AES128–SHA, ECDHE–RSA–AES128–SHA, DHE–DSS–AES128–SHA, ECDH–ECDSA–AES128–SHA, ECDH–RSA–AES128–SHA, ECDH–ECDSA–AES128–SHA, ECDH–RSA–AES128–SHA, AES128–SHA

说明

指定服务端支持的密码套件。

listener.ssl.external.psk_ciphers

Type	Default
string	PSK–AES128–CBC–SHA, PSK–AES256–CBC–SHA, PSK–3DES–EDE–CBC–SHA, PSK–RC4–SHA

说明

若使用 **PSK** 算法，指定服务端支持的 **PSK Cipher** 列表。注意 '**listener.ssl.external.ciphers**' 和 '**listener.ssl.external.psk_ciphers**' 只能配置一个。

listener.ssl.external.secure_renegotiate

Type	Optional Value	Default
enum	on , off	off

说明

指定在客户端不遵循 **RFC 5746** 的情况下，是否拒绝 **renegotiation** 请求。

listener.ssl.external.reuse_sessions

Type	Optional Value	Default
enum	on , off	on

说明

指定是否支持 **SSL session** 重用。详情见 <http://erlang.org/doc/man/ssl.html>。

listener.ssl.external.honor_cipher_order

Type	Optional Value	Default
enum	on , off	on

说明

指定是否使用服务端的偏好设置选择 **Ciphers**。

listener.ssl.external.peer_cert_as_username

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **Username** 字段的值。其可选值为：

- **cn**: 客户端证书的 **Common Name** 字段值
- **dn**: 客户端证书的 **Subject Name** 字段值
- **crt**: **DER** 格式编码的客户端证书二进制
- **pem**: 基于 **DER** 格式上的 **base64** 编码后的字符串
- **md5**: **DER** 格式证书的 **MD5** 哈希值

注意 `listener.ssl.external.verify` 应当设置为 `verify_peer`。

listener.ssl.external.peer_cert_as_clientid

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **ClientID** 字段的值。其可选值的含义同上。

注意 `listener.ssl.external.verify` 应当设置为 `verify_peer`。

listener.ssl.external.backlog

Type	Default
integer	1024

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

listener.ssl.external.send_timeout

Type	Default
duration	15s

说明

TCP 报文发送超时时间。

listener.ssl.external.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

TCP 报文发送超时后，是否关闭该连接。

listener.ssl.external.redbuf

Type	Default
bytesize	-

说明

TCP 接收缓存区大小（操作系统内核级参数）。

参见：<http://erlang.org/doc/man/inet.html>。

listener.ssl.external.sndbuf

Type	Default
bytesize	-

说明

TCP 发送缓存区大小（操作系统内核级参数）。

参见：<http://erlang.org/doc/man/inet.html>。

listener.ssl.external.buffer

Type	Default
bytesize	-

说明

TCP 缓冲区大小 (用户级)。

该值建议大于等于 `sndbuff` 和 `recbuff` 的最大值，以避免一些性能问题。在不配置的情况下，它默认等于 `sndbuff` 和 `recbuff` 的最大值。

参见：<http://erlang.org/doc/man/inet.html>。

listener.ssl.external.tune_buffer

Type	Optional Value	Default
enum	on , off	-

说明

如果打开此配置，请设置该值等于 `sndbuff` 与 `recbuff` 的最大值。

listener.ssl.external.nodelay

Type	Optional Value	Default
enum	true , false	true

说明

即 `TCP_NODELAY` 参数。开启该选项即表示禁用 **Nagle** 算法，小包将被立即发送。

listener.ssl.external.reuseaddr

Type	Optional Value	Default
enum	true , false	true

说明

即 `SO_REUSEADDR` 参数。开启该选项即允许本地重用端口，无需等待 `TIME_WAIT` 状态结束。

wsexternal

listener.ws.external

Type	Default
string	8083

说明

配置名称为 `external` 的 **MQTT/WS** 监听器的监听地址。

示例

`8083` : 表监听 **IPv4** 的 `0.0.0.0:8083` 。 `127.0.0.1:8083` : 表监听地址为 `127.0.0.1` 网卡上的 `8083` 端口。 `::1:8083` : 表监听 **IPv6** 地址为 `::1` 网卡上的 `8083` 端口。

listener.ws.external.mqtt_path

Type	Default
string	/mqtt

说明

WebSocket 的 **MQTT** 协议路径。因此 **EMQX** 的 **WebSocket** 的地址是：`ws://{ip}:{port}/mqtt`。

listener.ws.external.acceptors

Type	Default
integer	4

说明

监听器的接收池大小。

listener.ws.external.max_connections

Type	Default
integer	102400

说明

监听器允许的最大并发连接数量。

listener.ws.external.max_conn_rate

Type	Default
integer	1000

说明

监听器允许的最大接入速率。单位：个/秒

listener.ws.external.active_n

Type	Default
integer	100

说明

监听器持续接收 **TCP** 报文的次数。

listener.ws.external.rate_limit

Type	Default
string	100KB,10s

说明

监听器的速率限制。格式为 `<limit>,<duration>`。

示例

`100KB,10s` : 表 限制 **10** 秒内的流入字节数不超过 **100 KB**。

listener.ws.external.zone

Type	Default
string	external

说明

监听器所属的配置域 (**Zone**)。

listener.ws.external.access.1

Type	Default
string	allow all

说明

监听器的 **ACL** 规则列表。它用于设置连接层的白/黑名单。

listener.ws.external.fail_if_no_subprotocol

Type	Optional Value	Default
enum	true , false	true

说明

如果设置为 **true**, 则服务器将在客户端没有携带 **Sec-WebSocket-Protocol** 字段时返回错误。微信小程序需关闭该验证。

listener.ws.external.supported_protocols

Type	Default
string	mqtt, mqtt-v3, mqtt-v3.1.1, mqtt-v5

说明

指定支持的子协议，子协议之间以逗号分隔。

listener.ws.external.proxy_address_header

Type	Optional Value	Default
string	X-Forwarded-For	-

说明

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，则可打开该配置获取客户端真实的 **IP** 地址。

listener.ws.external.proxy_port_header

Type	Optional Value	Default
string	X-Forwarded-Port	-

说明

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，则可打开该配置获取客户端真实的端口。

listener.ws.external.proxy_protocol

Type	Optional Value	Default
enum	<code>on</code> , <code>off</code>	-

说明

监听器是否开启 `Proxy Protocol` 的支持。

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后，且需要拿到客户端真实的源 IP 地址与端口，则需打开此配置。

`Proxy Protocol` 参考: <https://www.haproxy.com/blog/haproxy/proxy-protocol>。

listener.ws.external.proxy_protocol_timeout

Type	Default
duration	-

说明

设置 `Proxy Protocol` 解析的超时时间。如果该时间内没收到 `Proxy Protocol` 的报文，**EMQX** 会关闭其连接。

listener.ws.external.backlog

Type	Default
integer	1024

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

listener.ws.external.send_timeout

Type	Default
duration	<code>15s</code>

说明

TCP 报文发送超时时间。

listener.ws.external.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

TCP 报文发送超时后，是否关闭该连接。

listener.ws.external.redbuf

Type	Default
bytesize	-

说明

TCP 接收缓存区大小 (操作系统内核级参数)

listener.ws.external.sndbuf

Type	Default
bytesize	-

说明

TCP 发送缓存区大小 (操作系统内核级参数)

listener.ws.external.buffer

Type	Default
bytesize	-

说明

TCP 缓冲区大小 (用户级)。

listener.ws.external.tune_buffer

Type	Optional Value	Default
enum	on , off	-

说明

如果打开此配置, 请设置该值等于 `sndbuff` 与 `recbuff` 的最大值。

listener.ws.external.nodelay

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>true</code>

说明

即 `TCP_NODELAY` 参数。开启该选项即允许小的 **TCP** 数据报文将会立即发送。

listener.ws.external.compress

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	-

说明

是否压缩 **WebSocket** 消息。压缩的实现依赖 [zlib](#)。

`defalte_opts` 下的配置项, 都属于压缩相关的参数配置, 如无必要请不需要修改它。

listener.ws.external.deflate_opts.level

Type	Optional Value	Default
enum	<code>none</code> , <code>default</code> , <code>best_compression</code> , <code>best_speed</code>	-

说明

压缩等级。

listener.ws.external.deflate_opts.mem_level

Type	Optional Value	Default
integer	<code>1 - 9</code>	-

说明

压缩参数。内存使用限制等级, 配置可开辟多少内存来参与压缩过程。

`1` : 最少的内存, 但会降低压缩率。 `9` : 最多的内存, 会提高计算速度和压缩率。

不配置，则默认为 8。

listener.ws.external.deflate_opts.strategy

Type	Optional Value	Default
enum	default , filtered , huffman_only , rle	-

说明

压缩策略，用于调优压缩率：

- default：针对普通数据。
- filtered：由过滤器或预测器产生的数据，适用于分布随机性强的内容。
- huffman_only：强制使用 **Huffman** 算法。优于 filtered。
- rle：将匹配距离限制为 **1 (Run-Length Encoding)**，比 huffman_only 要快，但主要用于 **PNG** 图片。

这些策略仅影响压缩率，不会对正确性带来任何影响。

listener.ws.external.deflate_opts.server_context_takeover

Type	Optional Value	Default
enum	takeover , no_takeover	-

说明

是否允许服务端的压缩上下文在帧之间传递。

listener.ws.external.deflate_opts.client_context_takeover

Type	Optional Value	Default
enum	takeover , no_takeover	-

说明

是否允许客户端的压缩上下文在帧之间传递。

listener.ws.external.deflate_opts.server_max_window_bits

Type	Optional Value	Default
integer	8 - 15	-

说明

服务端最大窗口值。设置一个较大的值会有更好的压缩率，但会额外的消耗内存。

listener.ws.external.deflate_opts.client_max_window_bits

Type	Optional Value	Default
integer	8 - 15	-

说明

客户端最大窗口值。设置一个较大的值会有更好的压缩率，但会额外的消耗内存。

listener.ws.external.idle_timeout

Type	Default
duration	-

说明

TCP 连接建立后的发呆时间，如果这段时间内未收到任何报文，则会关闭该连接。

listener.ws.external.max_frame_size

Type	Default
integer	-

说明

允许的单个 **MQTT** 报文长度的最大值。

wsexternal

listener.wss.external

Type	Default
string	0.0.0.0:8084

说明

配置名称为 `external` 的 **WSS (MQTT/WebSocket/SSL)** 监听器。

listener.wss.external.mqtt_path

Type	Default
string	/mqtt

说明

WebSocket 的 URL Path。

listener.wss.external.acceptors

Type	Default
integer	4

说明

监听器的接收池大小。

listener.wss.external.max_connections

Type	Default
integer	16

说明

监听器允许的最大并发连接数量。

listener.wss.external.max_conn_rate

Type	Default
integer	1000

说明

监听器允许的最大接入速率。单位：个/秒。

listener.wss.external.active_n

Type	Default
integer	100

说明

监听器持续接收 **TCP** 报文的次数。

listener.wss.external.rate_limit

Type	Default
string	-

说明

监听器的速率限制。格式为 `<limit>,<duration>`。

listener.wss.external.zone

Type	Default
string	external

说明

监听器所属的配置组 (**Zone**)。

listener.wss.external.access.1

Type	Default
string	allow all

说明

监听器的 **ACL** 规则列表。它用于设置连接层的白/黑名单。

例如:

`allow all` : 表允许所有的 **TCP** 连接接入。 `allow 192.168.0.0/24` : 表允许网络地址为 `192.168.0.0/24` 的 **TCP** 连接接入。

同时，该配置可配置多条规则:

1	<code>listener.wss.external.access.1 = deny 192.168.0.1</code>
2	<code>listener.wss.external.access.2 = allow all</code>

listener.wss.external.fail_if_no_subprotocol

Type	Optional Value	Default
enum	true , false	true

说明

如果设置为 `true`, 则服务器将在客户端没有携带 `Sec-WebSocket-Protocol` 字段时返回错误。微信小程序需关闭该验证。

listener.wss.external.supported_protocols

Type	Default
string	mqtt, mqtt-v3, mqtt-v3.1.1, mqtt-v5

说明

指定支持的子协议，子协议之间以逗号分隔。

listener.wss.external.proxy_address_header

Type	Default
string	X-Forwarded-For

说明

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx**, 则可打开该配置获取客户端真实的 IP 地址。

listener.wss.external.proxy_protocol

Type	Optional Value	Default
enum	on , off	-

说明

监听器是否开启 `Proxy Protocol` 的支持。

如果 **EMQX** 集群部署在 **HAProxy** 或 **Nginx** 后, 且需要拿到客户端真实的源 IP 地址与端口, 则需打开此配置。

Proxy Protocol 参考: <https://www.haproxy.com/blog/haproxy/proxy-protocol>。

listener.wss.external.proxy_protocol_timeout

Type	Default
duration	-

说明

设置 **Proxy Protocol** 解析的超时时间。如果该时间内没收到 **Proxy Protocol** 的报文, **EMQX** 会关闭其连接。

listener.wss.external.tls_versions

Type	Default
string	tlsv1.3,tlsv1.2,tlsv1.1,tlsv1

说明

指定服务端支持的 **SSL** 的版本列表。详情请参见 <http://erlang.org/doc/man/ssl.html>。

listener.wss.external.keyfile

Type	Default
string	etc/certs/key.pem

说明

指定 **SSL** 的私钥文件 (**PEM**)。

listener.wss.external.certfile

Type	Default
string	etc/certs/cert.pem

说明

指定 **SSL** 的证书文件 (**PEM**)。

listener.wss.external.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

指定 **SSL** 的 **CA** 证书文件 (**PEM**)。该文件应包含发布服务器证书的所有中间**CA**证书以及根证书。该文件还应包含所有受信**CA**的证书用以用于验证客户端的证书。

listener.wss.external.depth

Type	Default
number	10

说明

证书链中非自签发的中间证书的最大数量。如果该值为 **0** 则表示，对端证书必须是根 **CA** 直接授信的。

listener.wss.external.key_password

Type	Default
string	-

说明

证书密钥文件的密码。如果你的证书密钥设置了密码，则需要配置该选项。

listener.wss.external.dhfile

Type	Default
string	etc/certs/dh-params.pem

说明

若使用 **Ephemeral Diffie-Helman** 算法，指定算法使用的 **key** 文件。

listener.wss.external.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	verify_peer

说明

指定握手过程中是否校验客户端。

listener.wss.external.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	false

说明

SSL 握手过程中若客户端没有证书，是否让握手失败。

listener.wss.external.ciphers

Type	Default
string	ECDHE–ECDSA–AES256–GCM–SHA384, ECDHE–RSA–AES256–GCM–SHA384, ECDHE–ECDSA–AES256–SHA384, ECDHE–RSA–AES256–SHA384, ECDHE–ECDSA–DES–CBC3–SHA, ECDH–ECDSA–AES256–GCM–SHA384, ECDH–RSA–AES256–GCM–SHA384, ECDH–ECDSA–AES256–SHA384, ECDH–RSA–AES256–SHA384, DHE–DSS–AES256–GCM–SHA384, DHE–DSS–AES256–SHA256, AES256–GCM–SHA384, AES256–SHA256, ECDHE–ECDSA–AES128–GCM–SHA256, ECDHE–RSA–AES128–GCM–SHA256, ECDHE–ECDSA–AES128–SHA256, ECDHE–RSA–AES128–SHA256, ECDH–ECDSA–AES128–GCM–SHA256, ECDH–RSA–AES128–GCM–SHA256, ECDH–ECDSA–AES128–SHA256, ECDH–RSA–AES128–SHA256, DHE–DSS–AES128–GCM–SHA256, DHE–DSS–AES128–SHA256, AES128–GCM–SHA256, AES128–SHA256, ECDHE–ECDSA–AES256–SHA, ECDHE–RSA–AES256–SHA, ECDH–ECDSA–AES256–SHA, ECDH–RSA–AES256–SHA, AES256–SHA, ECDHE–ECDSA–AES128–SHA, ECDHE–RSA–AES128–SHA, DHE–DSS–AES128–SHA, ECDH–ECDSA–AES128–SHA, AES128–SHA, ECDH–RSA–AES128–SHA, AES128–SHA

说明

指定服务器支持的密码套件。

listener.wss.external.psk_ciphers

Type	Default
string	PSK–AES128–CBC–SHA, PSK–AES256–CBC–SHA, PSK–3DES–EDE–CBC–SHA, PSK–RC4–SHA

说明

若使用 **PSK** 算法，指定服务端支持的 **PSK Cipher** 列表。注意 '**listener.wss.external.ciphers**' 和 '**listener.wss.external.psk_ciphers**' 只能配置一个。

listener.wss.external.secure_renegotiate

Type	Optional Value	Default
enum	on , off	off

说明

指定在客户端不遵循 **RFC 5746** 的情况下，是否拒绝 **renegotiation** 请求。

listener.wss.external.reuse_sessions

Type	Optional Value	Default
enum	on , off	on

说明

指定是否支持 **SSL session** 重用。详情见 <http://erlang.org/doc/man/ssl.html>。

listener.wss.external.honor_cipher_order

Type	Optional Value	Default
enum	on , off	on

说明

指定是否使用服务端的偏好设置选择 **Ciphers**。

listener.wss.external.peer_cert_as_username

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **Username** 字段的值。其可选值为：

- **cn**: 客户端证书的 **Common Name** 字段值
- **dn**: 客户端证书的 **Subject Name** 字段值
- **crt**: **DER** 格式编码的客户端证书二进制
- **pem**: 基于 **DER** 格式上的 **base64** 编码后的字符串
- **md5**: **DER** 格式证书的 **MD5** 哈希值

注意 `listener.wss.external.verify` 应当设置为 `verify_peer`。

listener.wss.external.peer_cert_as_clientid

Type	Optional Value	Default
enum	cn , dn , crt , pem , md5	cn

说明

使用客户端证书来覆盖 **ClientID** 字段的值。其可选值的含义同上。

注意 `listener.wss.external.verify` 应当设置为 `verify_peer`。

listener.wss.external.backlog

Type	Default
integer	1024

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

listener.wss.external.send_timeout

Type	Default
duration	15s

说明

TCP 报文发送超时时间。

listener.wss.external.send_timeout_close

Type	Optional Value	Default
enum	<code>on</code> , <code>off</code>	<code>on</code>

说明

TCP 报文发送超时后，是否关闭该连接。

listener.wss.external.redbuf

Type	Default
bytesize	-

说明

TCP 接收缓存区大小（操作系统内核级参数）

参见：<http://erlang.org/doc/man/inet.html>

listener.wss.external.sndbuf

Type	Default
bytesize	-

说明

TCP 发送缓存区大小 (操作系统内核级参数)

参见: <http://erlang.org/doc/man/inet.html>

listener.wss.external.buffer

Type	Default
bytesize	-

说明

TCP 缓冲区大小 (用户级)。

该值建议大于等于 `sndbuff` 和 `recbuff` 的最大值，以避免一些性能问题。在不配置的情况下，它默认等于 `sndbuff` 和 `recbuff` 的最大值

参见: <http://erlang.org/doc/man/inet.html>

listener.wss.external.tune_buffer

Type	Optional Value	Default
enum	on , off	-

说明

如果打开此配置，请设置该值等于 `sndbuff` 与 `recbuff` 的最大值。

listener.wss.external.nodelay

Type	Optional Value	Default
enum	true , false	true

说明

即 `TCP_NODELAY` 参数。开启该选项即允许小的 **TCP** 数据报文将会立即发送。

listener.wss.external.compress

Type	Optional Value	Default
enum	true , false	false

说明

该选项若设置为 true, **WebSocket** 消息将会被压缩。

listener.wss.external.deflate_opts.level

Type	Optional Value	Default
enum	none , default , best_compression , best_speed	default

说明

压缩等级。

listener.wss.external.deflate_opts.mem_level

Type	Optional Value	Default
integer	1 - 9	-

说明

压缩参数。内存使用限制等级，配置可开辟多少内存来参与压缩过程。

1 : 最少的内存，但会降低压缩率。 9 : 最多的内存，会提高计算速度和压缩率。

不配置，则默认为 8 。

listener.wss.external.deflate_opts.strategy

Type	Optional Value	Default
enum	default , filtered , huffman_only , rle	-

说明

压缩策略，用于调优压缩率：

- default : 针对普通数据。
- filtered : 由过滤器或预测器产生的数据，适用于分布随机性强的内容。
- huffman_only : 强制使用 **Huffman** 算法。优于 filtered 。
- rle : 将匹配距离限制为 **1 (Run-Length Encoding)**，比 huffman_only 要快，但主要用于 **PNG** 图片。

这些策略仅影响压缩率，不会对正确性带来任何影响。

listener.wss.external.deflate_opts.server_context_takeover

Type	Optional Value	Default
enum	<code>takeover</code> , <code>no_takeover</code>	-

说明

是否允许服务端的压缩上下文在帧之间传递。

listener.wss.external.deflate_opts.client_context_takeover

Type	Optional Value	Default
enum	<code>takeover</code> , <code>no_takeover</code>	-

说明

是否允许客户端的压缩上下文在帧之间传递。

listener.wss.external.deflate_opts.server_max_window_bits

Type	Optional Value	Default
integer	8 - 15	-

说明

服务端最大窗口值。设置一个较大的值会有更好的压缩率，但会额外的消耗内存。

listener.wss.external.deflate_opts.client_max_window_bits

Type	Optional Value	Default
integer	8 - 15	-

说明

客户端最大窗口值。设置一个较大的值会有更好的压缩率，但会额外的消耗内存。

listener.wss.external.idle_timeout

Type	Default
duration	-

说明

TCP 连接建立后的发呆时间，如果这段时间内未收到任何报文，则会关闭该连接。

listener.wss.external.max_frame_size

Type	Default
integer	-

说明

允许的单个 MQTT 报文长度的最大值。

plugins

plugins/etc_dir

Type	Default
string	etc/plugins

说明

插件的配置目录。

plugins.loaded_file

Type	Default
string	etc/loaded_plugins

说明

插件启动列表的配置文件路径。

plugins.expand_plugins_dir

Type	Default
string	plugins/

说明

外部插件存放目录。

broker

broker.sys_interval

Type	Default
duration	1m

说明

设置系统主题 (\$SYS) 消息的发布间隔。

broker.sys_heartbeat

Type	Default
duration	30s

说明

设置系统心跳消息的发布间隔。系统心跳消息包括下面两个主题：

- "\$SYS/brokers/<node>/uptime"
- "\$SYS/brokers/<node>/datetime"

broker.enable_session_registry

Type	Optional Value	Default
enum	on , off	on

说明

启用或关闭全局会话注册。

broker.session_locking_strategy

Type	Optional Value	Default
enum	local , leader , quorum , all	quorum

说明

设置会话集群锁的类型。会话的集群锁用来防止同一个客户端在多个不同节点上创建多个会话，常见于客户端频繁切换节点登录的情况。

broker.shared_subscription_strategy

Type	Optional Value	Default
enum	random , round_robin , sticky , hash_clientid , hash_topic	random

说明

设置共享订阅的分发策略。可选值为：

- **random**: 在所有订阅者中随机选择
- **round_robin**: 按照一个固定的顺序选择下一个订阅者
- **sticky**: 首次分发时随机选择一个订阅者，后续消息一直发往这一个订阅者
- **hash_clientid**: 按照发布者 ClientID 的哈希值
- **hash_topic**: 按照源消息主题的哈希值

broker.shared_dispatch_ack_enabled

Type	Optional Value	Default
enum	true , false	false

说明

开启或关闭共享订阅对于 **qos1/qos2** 消息的 **ACK** 检查功能。开启后，如果投递到某个订阅者但收不到**ACK**，将尝试投递给订阅组里的下一个订阅者。

broker.route_batch_clean

Type	Optional Value	Default
enum	on , off	off

说明

开启或关闭批量清理路由信息。批量清理路由可用在短时间内大量客户端掉线的情况下，以提高清理效率。

broker.perf.route_lock_type = key

Type	Optional Value	Default
enum	key , tab , global	key

Description

选择在数据库中为通配符订阅更新路由信息时锁的粒度。

- key (默认值) 为每个前缀拿一次数据库锁。
- tab 表锁
- global 全局锁

对于较大集群, (如7个node或以上), 尤其是node之间网络延迟大的, 推荐是用 tab 和 global。注意: 是需要重启整个集群来使得更新生效。

broker.perf.trie_compaction = true

Type	Optional Value	Default
enum	true , false	true

Description

设置为 true 时, 对通配符订阅表进行压缩。压缩可优化写操作, 降低高并发量的订阅请求响应时间, 内存使用量也只有非压缩时的一半。非压缩优化读操作, 适用于发布主题层数较多的场景。

注意: 将该配置从 false 改成 true 时, 集群中的节点可依次重启来使配置生效。从 true 改为 false 时, 需要将集群中所有的节点重启, 否则会发生有些消息无法被路由的情况。

monitor

sysmon.long_gc

Type	Default
duration	0ms

说明

启用垃圾回收时间监控并在回收时间超过设定值时触发警告, 0 表示禁用此监控。

sysmon.long_schedule

Type	Default
duration	240ms

说明

启用进程调度时间监控并在调度时间超过设定值时触发告警，**0** 表示禁用此监控。

sysmon.large_heap

Type	Default
bytesize	8MB

说明

启用堆栈大小监控并在进程执行垃圾回收后堆栈大小仍大于设定值时触发告警，**0** 表示禁用此监控。

sysmon.busy_port

Type	Optional Value	Default
enum	true , false	false

说明

指定是否启用进程间消息通道拥塞监控。

sysmon.busy_dist_port

Type	Optional Value	Default
enum	true , false	true

说明

指定是否启用集群 **RPC** 通道拥塞监控。

os_mon.cpu_check_interval

Type	Default
duration	60s

说明

CPU 占用率检查周期。

os_mon.cpu_high_watermark

Type	Default
percent	80%

说明

CPU 占用率超过 `os_mon.cpu_high_watermark` 时将触发告警。

os_mon.cpu_low_watermark

Type	Default
percent	60%

说明

CPU 占用率回落到 `os_mon.cpu_low_watermark` 以下时将清除告警。

os_mon.mem_check_interval

Type	Default
duration	60s

说明

内存占用率检查周期。

os_mon.sysmem_high_watermark

Type	Default
percent	70%

说明

EMQX 为所有进程分配的内存占系统内存的百分比超过 `os_mon.sysmem_high_watermark` 时将触发告警。

os_mon.procmem_high_watermark

Type	Default
percent	5%

说明

EMQX 为单个进程分配的内存占系统内存的百分比超过 `os_mon.procmem_high_watermark` 时将触发告警。

vm_mon.check_interval

Type	Default
<code>duration</code>	<code>30s</code>

说明

进程数量检查周期。

vm_mon.process_high_watermark

Type	Default
<code>percent</code>	<code>80%</code>

说明

当前进程数量占进程最大数量的百分比超过 `vm_mon.process_high_watermark` 时将触发告警。进程最大数量由 `node.process_limit` 配置项决定。

vm_mon.process_low_watermark

Type	Default
<code>percent</code>	<code>60%</code>

说明

当前进程数量占进程最大数量的百分比回落到 `vm_mon.process_low_watermark` 以下时将触发告警。进程最大数量由 `node.process_limit` 配置项决定。

插件 `emqx-auth-http`

auth.http.auth_req.url

Type	Default
<code>string</code>	<code>http://127.0.0.1:80/mqtt/auth</code>

说明

指定认证请求的目标 **URL**。

auth.http.auth_req.method

Type	Optional Value	Default
enum	get , post	post

说明

指定认证请求的请求方法。

auth.http.auth_req.headers.<Any>

示例

```
1 auth.http.auth_req.headers.content-type = application/x-www-form-urlencoded
2 auth.http.auth_req.headers.accept = */*
```

说明

指定 **HTTP** 请求头部中的数据。<Key> 指定 **HTTP** 请求头部中的字段名，此配置项的值为相应的字段值。<Key> 可以是标准的 **HTTP** 请求头部字段，也可以自定义的字段，可以配置多个不同的请求头部字段。

auth.http.auth_req.params

Type	Format	Default
string	以 <code>,</code> 分隔的 <code>k=v</code> 键值对， <code>v</code> 可以是固定内容，也可以是占位符	<code>clientid=%c,username=%u,password=%P</code>

说明

指定认证请求中携带的数据。使用 **GET** 方法时 `auth.http.auth_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以查询字符串参数的形式发送。使用 **POST** 方法时 `auth.http.auth_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以 **Request Body** 的形式发送。所有的占位符都会被运行时数据所替换，可用的占位符如下：

占位符	替换内容
%u	用户名
%c	MQTT Client ID
%a	客户端的网络 IP 地址
%r	客户端使用的协议，可以是: <code>mqtt</code> , <code>mqtt-sn</code> , <code>coap</code> , <code>lwm2m</code> 以及 <code>stomp</code>
%P	密码
%p	客户端连接的服务端端口
%C	客户端证书中的 Common Name
%d	客户端证书中的 Subject

auth.http.super_req.url

Type	Default
string	<code>http://127.0.0.1:80/mqtt/superuser</code>

说明

指定超级用户认证请求的目标 **URL**。

auth.http.super_req.method

Type	Optional Value	Default
enum	<code>get</code> , <code>post</code>	<code>post</code>

说明

指定超级用户认证请求的请求方法。

auth.http.super_req.headers.<Any>

示例

```
1 auth.http.super_req.headers.content-type = application/x-www-form-urlencoded
2 auth.http.super_req.headers.accept = */*
```

说明

指定 **HTTP** 请求头部中的数据。<Key> 指定 **HTTP** 请求头部中的字段名，此配置项的值为相应的字段值。<Key> 可以是标准的 **HTTP** 请求头部字段，也可以自定义的字段，可以配置多个不同的请求头部字段。

auth.http.super_req.params

Type	Format	Default
string	以 <code>,</code> 分隔的 <code>k=v</code> 键值对, <code>v</code> 可以是固定内容, 也可以是占位符	<code>clientid=%c,username=%u</code>

说明

指定超级用户认证请求中携带的数据。使用 **GET** 方法时 `auth.http.super_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以查询字符串参数的形式发送。使用 **POST** 方法时 `auth.http.super_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以 **Request Body** 的形式发送。所有的占位符都会被运行时数据所替换, 可用的占位符同 `auth.http.auth_params`。

auth.http.acl_req

Type	Default
string	<code>http://127.0.0.1:8991/mqtt/acl</code>

说明

指定 **ACL** 验证请求的目标 **URL**。

auth.http.acl_req.method

Type	Optional Value	Default
enum	<code>get, post</code>	<code>post</code>

说明

指定 **ACL** 验证请求的请求方法。

auth.http.acl_req.headers.<Any>

示例

```
1 auth.http.acl_req.headers.content-type = application/x-www-form-urlencoded
2 auth.http.acl_req.headers.accept = */*
```

说明

指定 **HTTP** 请求头部中的数据。`<Key>` 指定 **HTTP** 请求头部中的字段名, 此配置项的值为相应的字段值。`<Key>` 可以是标准的 **HTTP** 请求头部字段, 也可以自定义的字段, 可以配置多个不同的请求头部字段。

auth.http.acl_req.params

Type	Format	Default
string	以 <code>,</code> 分隔的 <code>k=v</code> 键值对, <code>v</code> 可以是固定内容, 也可以是占位符	<code>access=%A,username=%u,clientid=%c,ipaddr=%a,topic=%t,mountpoint=%m</code>

说明

指定 **ACL** 验证请求中携带的数据。使用 **GET** 方法时 `auth.http.acl_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以查询字符串参数的形式发送。使用 **POST** 方法时 `auth.http.acl_req.params` 的值将被转换为以 `&` 分隔的 `k=v` 键值对以 **Request Body** 的形式发送。所有的占位符都会被运行时数据所替换, 可用的占位符如下:

占位符	替换内容
<code>%A</code>	需要验证的权限, 1 表示订阅, 2 表示发布
<code>%u</code>	MQTT Client ID
<code>%c</code>	客户端标识符
<code>%a</code>	客户端的网络 IP 地址
<code>%r</code>	客户端使用的协议, 可以是: <code>mqtt</code> , <code>mqtt-sn</code> , <code>coap</code> , <code>lwm2m</code> 以及 <code>stomp</code>
<code>%m</code>	挂载点
<code>%t</code>	主题

auth.http.timeout

Type	Default
<code>duration</code>	<code>5s</code>

说明

HTTP 请求超时时间。任何等价于 `0s` 的设定值都表示永不超时。

auth.http.connect_timeout

Type	Default
<code>duration</code>	<code>5s</code>

说明

HTTP 请求的连接超时时间。任何等价于 `0s` 的设定值都表示永不超时。

auth.http.ssl.cacertfile

Type	Default
string	etc/certs/ca.pem

说明

CA 证书文件路径。

auth.http.ssl.certfile

Type	Default
string	etc/certs/client-cert.pem

说明

客户端证书文件路径。

auth.http.ssl.keyfile

Type	Default
string	etc/certs/client.key.pem

说明

客户端私钥文件路径。

插件 emqx_auth_jwt

auth.jwt.secret

Type	Default
string	emqxsecret

说明

设置 HMAC Secret。

auth.jwt.from

Type	Optional Value	Default
enum	username , password	password

说明

从什么地方获取 **JWT**。可选值为:

- **username**: MQTT CONNECT 报文的 **username** 字段作为 **JWT**。
- **password**: MQTT CONNECT 报文的 **password** 字段作为 **JWT**。

auth.jwt.pubkey

Type	Default
string	etc/certs/jwt_public_key.pem

说明

若使用 **RSA** 或者 **ECDSA** 加密算法，须指定私钥文件。

auth.jwt.verify_claims

Type	Optional Value	Default
enum	on , off	off

说明

启用或关闭 **Claims** 校验功能。

auth.jwt.verify_claims.<claims>

Type	Default
string	-

说明

启用 **Claims** 校验功能时，可设置 **JWT** 中字段的可选值。

例如，若期望 **JWT** 中的 **Claim** 字段 `sub` 的值为 "abc"，则可以配置如下规则:

```
1 auth.jwt.verify_claims.sub = abc
```

期望值支持两个通配符:

- `%u` : **username**
- `%c` : **clientid**

例如，若期望 **JWT** 中的 **Claim** 字段 `sub` 的值与 **MQTT CONNECT** 报文中 **username** 字段相同，则可以配置如下规则:

1	auth.jwt.verify_claims.sub = %u
---	---------------------------------

插件 emqx_auth_ldap

auth.ldap.servers

Type	Default
string	127.0.0.1

说明

LDAP 服务地址。

auth.ldap.port

Type	Default
integer	389

说明

LDAP 服务端口。

auth.ldap.pool

Type	Optional Value	Default
integer	> 0	8

说明

连接池大小。

auth.ldap.bind_dn

Type	Default
string	cn=root,dc=emqx,dc=io

说明

登入 LDAP 服务的 DN。

auth.ldap.bind_password

Type	Default
string	public

说明

登入 **LDAP** 服务的密码。

auth.ldap.timeout

Type	Default
duration	30s

说明

查询操作的超时时间。

auth.ldap.device_dn

Type	Default
string	ou=device,dc=emqx,dc=io

说明

客户端隶属的 **DN**。

auth.ldap.match_objectclass

Type	Default
string	mqttUser

说明

客户端对象的名称。

auth.ldap.username.attributetype

Type	Default
string	uid

说明

Username 属性的数据类型。

auth.ldap.password.attributetype

Type	Default
string	userPassword

说明

Password 属性的数据类型。

auth.ldap.ssl

Type	Optional Value	Default
enum	true , false	false

说明

是否开启 **SSL**。

auth.ldap.ssl.certfile

Type	Default
string	-

说明

SSL 服务端证书路径。

auth.ldap.ssl.keyfile

Type	Default
string	-

说明

SSL 服务端秘钥文件路径。

auth.ldap.ssl.cacertfile

Type	Default
string	-

说明

CA 证书文件路径。

auth.ldap.ssl.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	-

说明

SSL 认证方式:

- `verify_none` : 单向认证。
- `verify_peer` : 双向认证。

auth.ldap.ssl.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	false

说明

如果客户端未提供 **SSL** 证书，则断开连接。

插件 emqx_auth_mongo

auth.mongo.type

Type	Optional Value	Default
enum	single , unknown , sharded , rs	single

说明

设置 **MongoDB** 的拓扑类型:

- **single**: 单节点
- **unknown**: 未知
- **sharded**: 分片模式

- **rs:** 副本模式 (replicated set)

auth.mongo.rs_set_name

Type	Default
string	127.0.0.1:27017

说明

设置 **MongoDB** 服务的地址。如有多个使用逗号 `,` 分隔。

auth.mongo.pool

Type	Default
integer	8

说明

设置 **MongoDB** 连接池的进程数。

auth.mongo.login

Type	Default
string	-

说明

设置 **MongoDB** 的用户名。

auth.mongo.password

Type	Default
string	-

说明

设置 **MongoDB** 的密码。

auth.mongo.auth_source

Type	Default
string	mqtt

说明

设置 **MongoDB** 的认证源数据库名。

auth.mongo.database

Type	Default
string	mqtt

说明

设置 **MongoDB** 的数据库名。

auth.mongo.query_timeout

Type	Default
duration	5s

说明

设置访问 **MongoDB** 超时时间。

auth.mongo.ssl

Type	Optional Value	Default
enum	true , false	false

说明

设置是否使用 **SSL** 访问 **MongoDB**。

auth.mongo.ssl_opts.keyfile

Type	Default
string	-

说明

若使用 **SSL** 访问 **MongoDB**, 设置 **SSL** 客户端的私钥文件。

auth.mongo.ssl_opts.certfile

Type	Default
string	-

说明

若使用 **SSL** 访问 **MongoDB**, 设置 **SSL** 客户端的证书文件。

auth.mongo.ssl_opts.cacertfile

Type	Default
string	-

说明

若使用 **SSL** 访问 **MongoDB**, 设置 **SSL** 的 **CA** 证书文件。

auth.mongo.w_mode

Type	Optional Value	Default
enum	<code>unsafe</code> , <code>safe</code> , <code>undef</code>	<code>undef</code>

说明

设置 **MongoDB** 的写入模式。

auth.mongo.r_mode

Type	Optional Value	Default
enum	<code>master</code> , <code>slave_ok</code> , <code>undef</code>	<code>undef</code>

说明

设置 **MongoDB** 的读取模式。

auth.mongo.auth_query.collection

Type	Default
string	mqtt_user

说明

认证过程用的 **Collection** 名字。

auth.mongo.auth_query.password_field

Type	Default
string	password

说明

认证过程用的主要字段。如需在密码之后加 **salt**, 可以配置为:

```
1 auth.mongo.auth_query.password_field = password,salt
```

auth.mongo.auth_query.password_hash

Type	Optional Value	Default
enum	plain , md5 , sha , sha256 , bcrypt	sha256

说明

设置密码字段用的哈希算法。如需在 **sha256** 密码之后加 **salt**, 可以设置为:

```
1 auth.mongo.auth_query.password_hash = sha256,salt
```

如需在 **sha256** 密码之前加 **salt**, 可以设置为:

```
1 auth.mongo.auth_query.password_hash = salt,sha256
```

如需在 **bcrypt** 密码之前加 **salt**, 可以设置为:

```
1 auth.mongo.auth_query.password_hash = salt,bcrypt
```

auth.mongo.auth_query.selector

Type	Default
string	username=%u

说明

认证过程执行的 **MongoDB** 语句。命令可支持通配符:

- %u: username
- %c: clientid
- %C: 客户端 TLS 证书里的 Common Name
- %d: 客户端 TLS 证书里的 Subject

auth.mongo.auth_query.super_query

Type	Optional Value	Default
enum	on , off	on

说明

认证中是否使用 SuperUser。

auth.mongo.super_query.collection

Type	Default
string	mqtt_user

说明

若使用 SuperUser, 指定 SuperUser 的 MongoDB Collection。

auth.mongo.super_query.selector

Type	Default
string	username=%u, clientid=%c

说明

若使用 SuperUser, 指定查询 SuperUser 使用的 MongoDB 语句。

auth.mongo.acl_query

Type	Optional Value	Default
enum	on , off	on

说明

是否开启 **ACL** 功能。

auth.mongo.acl_query.collection

Type	Default
string	mqtt_acl

说明

若使用 **ACL** 功能，指定查询 **ACL** 规则的 **MongoDB Collection**。

auth.mongo.acl_query.selector

Type	Default
string	username=%u

说明

若使用 **ACL** 功能，指定查询 **ACL** 规则使用的 **MongoDB** 语句。可支持多个 **ACL** 语句，多个语句之间使用 **or** 连接。

例如，配置如下两条访问规则：

```
1 auth.mongo.acl_query.selector.1 = username=%u
2 auth.mongo.acl_query.selector.2 = username=$all
```

并且客户端的 **username='ilyas'**，则在查询 **acl** 规则的时候，会执行如下 **MongoDB** 语句：

```
1 db.mqtt_acl.find({$or: [{username: "ilyas"}, {username: "$all"}]});
```

auth.mongo.topology.pool_size

Type	Default
integer	1

说明

MongoDB 拓扑参数，设置线程池大小。

auth.mongo.topology.max_overflow

Type	Default
integer	0

说明

MongoDB 拓扑参数，当线程池中所有 **workers** 都处于忙碌状态时，允许创建多少额外的 **worker** 线程。

auth.mongo.topology.overflow_ttl

Type	Default
integer	1000

说明

MongoDB 拓扑参数，当有 **worker** 空闲时。多久之后释放额外的 **worker** 线程。单位：毫秒。

auth.mongo.topology.overflow_check_period

Type	Default
integer	1000

说明

MongoDB 拓扑参数，多长时间检查一次有无空闲线程，以释放额外的 **worker**。

auth.mongo.topology.local_threshold_ms

Type	Default
integer	1000

说明

MongoDB 拓扑参数，选择用来处理用户请求的 **Secondary** 节点的策略。记到所有节点的 **RTT** 中的最小值为 **LowestRTT**，那么只有那些 $RTT < \text{LowestRTT} + \text{local_threshold_ms}$ 的 **Secondary** 节点会被选择。

auth.mongo.topology.connect_timeout_ms

Type	Default
integer	20000

说明

MongoDB 拓扑参数，**MongoDB** 连接超时时间，单位：毫秒。

auth.mongo.topology.socket_timeout_ms

Type	Default
integer	100

说明

MongoDB 拓扑参数，**MongoDB** 消息发送超时时间，单位：毫秒。

auth.mongo.topology.server_selection_timeout_ms

Type	Default
integer	30000

说明

MongoDB 拓扑参数，选择 **MongoDB Server** 的超时时间，单位：毫秒。

auth.mongo.topology.wait_queue_timeout_ms

Type	Default
integer	1000

说明

MongoDB 拓扑参数，从线程池中选取 **worker** 的等待超时时间，单位：毫秒。

auth.mongo.topology.heartbeat_frequency_ms

Type	Default
integer	10000

说明

MongoDB 拓扑参数，拓扑扫描之间的间隔时间，单位：毫秒。

auth.mongo.topology.min_heartbeat_frequency_ms

Type	Default
integer	1000

说明

MongoDB 拓扑参数，`heartbeat_frequency_ms` 允许的最小值，单位：毫秒。

插件 emqx_auth_mysql

auth.mysql.server

Type	Default
ip	127.0.0.1:3306

说明

MySQL 服务器地址。

auth.mysql.pool

Type	Default
integer	8

说明

数据库连接线程池大小。

auth.mysql.username

Type	Default
string	-

说明

MySQL 用户名。

auth.mysql.password

Type	Default
string	-

说明

MySQL 密码。

auth.mysql.database

Type	Default
string	mqtt

说明

MySQL 数据库名称。

auth.mysql.query_timeout

Type	Default
duration	5s

说明

MySQL 数据查询超时时间。查询超时将等同于未找到用户数据处理。

auth.mysql.auth_query

Type	Default
string	select password from mqtt_user where username = '%u' limit 1

说明

认证时使用的 **MySQL** 选取语句，选取出来的数据将与经过由 `auth.mysql.password_hash` 指定的加密方式加密的密码进行比较，比较后内容一致的客户端将被允许登录。加盐后存储的密码需要同时选取盐对应的字段，例如 `select password, salt from mqtt_user where username = '%u' limit 1.` `password` 与 `salt` 字段名不可以修改，表名与 **WHERE** 子句中的字段名可以视情况变化。**WHERE** 子句支持以下占位符：

占位符	说明
%u	将被替换为 MQTT 客户端在 CONNECT 报文中指定的用户名
%c	将被替换为 MQTT 客户端在 CONNECT 报文中指定的客户端标识符
%C	将被替换为 TLS 连接时客户端证书中的 Common Name
%d	将被替换为 TLS 连接时客户端证书中的 Subject

auth.mysql.password_hash

Type	Default
string	sh256

说明

存储在数据库的密码所使用的加密方式。支持以下加密方式：

- plain，支持前后加盐，例如 salt,plain
- md5，支持前后加盐
- sha，支持前后加盐
- sha256，支持前后加盐
- sha512，支持前后加盐
- pbkdf2，格式为 pbkdf2,<Hashfun>,<Iterations>,<Dklen>。其中，<Hashfun> 为使用的哈希函数，支持 md4，md5，ripemd160，sha，sha224，sha256，sha384，sha512，<Iterations> 为迭代次数，<Dklen> 为导出密钥长度。示例：pbkdf2,sha256,1000,20
- bcrypt，仅支持前向加盐，例如 salt,bcrypt

auth.mysql.super_query

Type	Default
string	select is_superuser from mqtt_user where username = '%u' limit 1

说明

超级用户认证时使用的 **SQL** 选取语句，此语句中所有表名与字段名都可视情况修改，当且仅当选取得到字段的值为 1 时，该用户为超级用户。**WHERE** 子句中支持的占位符与 auth.mysql.auth_query 相同。

auth.mysql.acl_query

Type	Default
string	select allow, ipaddr, username, clientid, access, topic from mqtt_acl where ipaddr = '%a' or username = '%u' or username = '\$all' or clientid = '%c'

ACL 校验时使用的 **SQL** 选取语句，此语句中所有表名与字段名都可视情况修改。**WHERE** 子句中支持的占位符如下：

占位符	说明
%a	将被替换为客户端 IP 地址
%u	将被替换为 MQTT 客户端在 CONNECT 报文中指定的用户名
%c	将被替换为 MQTT 客户端在 CONNECT 报文中指定的客户端标识符

插件 emqx_auth_pgsql

auth.pgsql.server

Type	Default
ip	127.0.0.1:5432

说明

PostgreSQL 服务器地址。

auth.pgsql.pool

Type	Default
integer	8

说明

数据库连接线程池大小。

auth.pgsql.username

Type	Default
string	root

说明

PostgreSQL 用户名。

auth.pgsql.password

Type	Default
string	-

说明

PostgreSQL 密码。

auth.pgsql.database

Type	Default
string	mqtt

说明

PostgreSQL 数据库名称。

auth.pgsql.encoding

Type	Default
string	utf8

说明

PostgreSQL 数据库字符编码格式。

auth.pgsql.ssl

Type	Optional Value	Default
enum	true , false	false

说明

是否启用 **TLS** 连接。

auth.pgsql.ssl_opts.keyfile

Type	Default
string	-

说明

客户端私钥文件路径。

auth_pgsql_ssl_opts_certfile

Type	Default
string	-

说明

客户端证书文件路径。

authpgsql.ssl_opts.cacertfile

Type	Default
string	-

说明

客户端 **CA** 证书文件路径。

authpgsql.auth_query

Type	Default
string	select password from mqtt_user where username = '%u' limit 1

说明

认证时使用的 **SQL** 选取语句，同 `auth.mysql.auth_query`。

authpgsql.password_hash

Type	Default
string	sh256

说明

存储在数据库的密码所使用的加密方式，同 `auth.mysql.password_hash`。

authpgsql.super_query

Type	Default
string	select is_superuser from mqtt_user where username = '%u' limit 1

说明

超级用户认证时使用的 **SQL** 选取语句，同 `auth.mysql.super_query`。

authpgsql.acl_query

Type	Default
string	<code>select allow, ipaddr, username, clientid, access, topic from mqtt_acl where ipaddr = '%a' or username = '%u' or username = '\$all' or clientid = '%c'</code>

说明

ACL 校验时使用的 **SQL** 选取语句，同 `auth.mysql.acl_query`。

插件 `emqx_auth_redis`

auth.redis.type

Type	Optional Value	Default
enum	<code>single</code> , <code>sentinel</code> , <code>cluster</code>	<code>single</code>

说明

Redis 服务集群类型：

- `single`：单节点服务。
- `sentinel`：哨兵模式。
- `cluster`：集群模式。

auth.redis.server

Type	Default
string	<code>127.0.0.1:6379</code>

说明

Redis 服务地址，如果有多个则以逗号分隔。例如，`192.168.0.1:6379, 192.168.0.2:6379`。

auth.redis.sentinel

Type	Default
string	-

说明

Redis sentinel 模式下的集区名称。如果非 `sentinel` 模式，则不需要配置。

auth.redis.pool

Type	Optional Value	Default
integer	> 0	8

说明

连接池大小。

auth.redis.database

Type	Default
integer	0

说明

要连接的 **Redis** 数据库序号。

auth.redis.password

Type	Default
string	-

说明

Redis 用户密码。

auth.redis.query_timeout

Type	Default
duration	5s

说明

Redis 查询超时时间。

auth.redis.auth_cmd

Type	Default
string	HGET mqtt_user:%u password

说明

认证查询命令，可用站位符有：

- %u : 客户端用户名。
- %c : 客户端标识。
- %C : 客户端 **SSL** 证书的 cn 。
- %d : 客户端 **SSL** 证书的 dn 。

auth.redis.password_hash

Type	Optional Value	Default
enum	plain , md5 , sha , sha256 , bcrypt	plain

说明

Redis 存储的 password 字段的编码格式。

auth.redis.super_cmd

Type	Default
string	HGET mqtt_user:%u is_superuser

说明

超级用户查询命令，可用的占位符有：

- %u : 客户端用户名。
- %c : 客户端标识。
- %C : 客户端 **SSL** 证书的 cn 。
- %d : 客户端 **SSL** 证书的 dn 。

auth.redis.acl_cmd

Type	Default
string	HGETALL mqtt_acl:%u

ACL 查询命令。可用的占位符有：

- `%u` : 客户端用户名。
- `%c` : 客户端标识。

插件 emqx_bridge_mqtt

bridge.mqtt.aws.address

Type	Default
<code>string</code>	<code>127.0.0.1:1883</code>

说明

桥接地址，支持两种格式，例如：

- `emqx@192.168.0.100` : **EMQX** 节点名称，它表示将该节点的消息桥接到另外一个 **EMQX** 节点。
- `192.168.0.100:1883` : IP 地址和端口，它表示将该节点的消息通过一个 **MQTT** 连接桥接到另外一个 **MQTT** 服务器。

bridge.mqtt.aws.proto_ver

Type	Optional Value	Default
<code>enum</code>	<code>mqttv3</code> , <code>mqttv4</code> , <code>mqttv5</code>	<code>mqttv4</code>

说明

MQTT 桥接的客户端协议版本。

bridge.mqtt.aws.start_type

Type	Optional Value	Default
<code>eunm</code>	<code>manual</code> , <code>auto</code>	<code>manual</code>

说明

启动类型：

- `auto` : 跟随插件自动启动。
- `manual` : 手动启动桥接。

bridge.mqtt.aws.bridge_mode

Type	Optional Value	Default
boolean	true , false	true

说明

是否开启桥接模式，仅 **MQTT** 桥接支持。开启后 `emqx_bridge_mqtt` 启动的 **MQTT** 客户端在发送连接报文时会携带一个标志位，标识这是一个桥接客户端。

注：**RabbitMQ** 目前不支持该标志。

bridge.mqtt.aws.clientid

Type	Default
string	bridge_aws

说明

MQTT 桥接的客户端标识。

bridge.mqtt.aws.clean_start

Type	Optional Value	Default
boolean	true , false	true

说明

MQTT 桥接的 `clean_start` 标志。它表示客户端是否以 `清楚会话` 的方式连接到远程 **MQTT Broker**。

bridge.mqtt.aws.username

Type	Default
string	user

说明

MQTT 桥接客户端的用户名。

bridge.mqtt.aws.password

Type	Default
string	passwd

说明

MQTT 桥接客户端的密码。

bridge.mqtt.aws.forwards

Type	Default
string	topic1/#,topic2/#

说明

桥接转发规则。例如：

- topic1/#, topic2/# : emqx_bridge_mqtt 会将 **EMQX** 中所以与 topic1/#, topic2/# 匹配的主题消息进行转发。

bridge.mqtt.aws.forward_mountpoint

Type	Default
string	bridge/aws/\${node}/

说明

转发主题的前缀。将消息转发到目标系统时，支持给该主题添加一个统一的前缀。

bridge.mqtt.aws.subscription.1.topic

Type	Default
string	-

说明

订阅对端系统的主题。

bridge.mqtt.aws.subscription.1.qos

Type	Optional Value	Default
enum	0 , 1 , 2	1

说明

订阅对端系统主题的 **QoS**。

bridge.mqtt.aws.receive_mountpoint

Type	Default
string	receive/aws/

说明

接收消息的主题前缀。 `emqx_bridge_mqtt` 支持给来着对端的消息添加一个统一的主题前缀。

bridge.mqtt.aws.ssl

Type	Optional Value	Default
boolean	true , false	true

说明

MQTT 桥接客户端是否开启 **SSL**。

bridge.mqtt.aws.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

MQTT 桥接客户端的 **CA** 证书文件路径。

bridge.mqtt.aws.certfile

Type	Default
string	etc/certs/client-cert.pem

说明

MQTT 桥接客户端的 **SSL** 证书文件路径。

bridge.mqtt.aws.keyfile

Type	Default
string	etc/certs/client-key.pem

说明

MQTT 桥接客户端的 **SSL** 秘钥文件路径。

bridge.mqtt.aws.ciphers

Type	Default
string	ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-RSA-AES256-GCM-SHA384

说明

SSL 握手支持的加密套件。

bridge.mqtt.aws.psk_ciphers

Type	Default
string	PSK-AES128-CBC-SHA, PSK-AES256-CBC-SHA, PSK-3DES-EDE-CBC-SHA, PSK-RC4-SHA

说明

SSL PSK 握手支持的加密套件。

bridge.mqtt.aws.keepalive

Type	Default
duration	60s

说明

MQTT 桥接客户端的心跳间隔。

bridge.mqtt.aws.tls_versions

Type	Default
string	tlsv1.3, tlsv1.2, tlsv1.1, tlsv1

说明

MQTT 桥接客户端的 SSL 版本。

bridge.mqtt.aws.reconnect_interval

Type	Default
duration	30s

说明

重连间隔。

bridge.mqtt.aws.retry_interval

Type	Default
duration	20s

说明

QoS 1/2 消息重发间隔。

bridge.mqtt.aws.batch_size

Type	Default
integer	32

说明

EMQX 桥接的批处理大小。emqx_bridge_mqtt 的 EMQX 桥接模式支持批量发送消息以提高吞吐。

bridge.mqtt.aws.max_inflight_size

Type	Default
integer	32

说明

飞行窗口大小。

bridge.mqtt.aws.queue.replayq_dir

Type	Default
string	etc/emqx_aws_bridge/

说明

设置消息队列文件路径。不配置则仅使用内存存储。

bridge.mqtt.aws.queue.replayq_seg_bytes

Type	Default
bytesize	10MB

说明

消息队列存储在磁盘的单个文件大小。

bridge.mqtt.aws.queue.max_total_size

Type	Default
bytesize	5GB

说明

消息队列允许存储的最大值。

插件 emqx_coap

coap.port

Type	Default
integer	5683

说明

指定 CoAP 插件的 UDP 绑定端口。

coap.enable_stats

Type	Optional Value	Default
enum	on , off	off

说明

启用或关闭 **CoAP** 的统计功能。

coap.dtls.port

Type	Default
integer	5684

说明

指定 **CoAP** 插件的 **DTLS** 绑定端口。

coap.dtls.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	verify_peer

说明

使用 **DTLS** 时，指定 **DTLS** 握手过程中是否校验客户端。

coap.dtls.keyfile

Type	Default
string	etc/certs/key.pem

说明

使用 **DTLS** 时，指定 **DTLS** 的私钥文件。

coap.dtls.certfile

Type	Default
string	etc/certs/cert.pem

说明

使用 **DTLS** 时，指定 **DTLS** 的证书文件。

coap.dtls.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

使用 **DTLS** 时，指定 **DTLS** 的 **CA** 证书文件。

coap.dtls.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	false

说明

使用 **DTLS** 时，**DTLS** 握手过程中若客户端没有证书，是否让握手失败。

coap.dtls.ciphers

Type	Default
string	ECDHE–ECDSA–AES256–GCM–SHA384, ECDHE–RSA–AES256–GCM–SHA384, ECDHE–ECDSA–AES256–SHA384, ECDHE–RSA–AES256–SHA384, ECDHE–ECDSA–DES–CBC3–SHA, ECDH–ECDSA–AES256–GCM–SHA384, ECDH–RSA–AES256–GCM–SHA384, ECDH–ECDSA–AES256–SHA384, ECDH–RSA–AES256–SHA384, DHE–DSS–AES256–GCM–SHA384, DHE–DSS–AES256–SHA256, AES256–GCM–SHA384, AES256–SHA256, ECDHE–ECDSA–AES128–GCM–SHA256, ECDHE–RSA–AES128–GCM–SHA256, ECDHE–ECDSA–AES128–SHA256, ECDHE–RSA–AES128–SHA256, ECDH–ECDSA–AES128–GCM–SHA256, ECDH–RSA–AES128–GCM–SHA256, ECDH–ECDSA–AES128–SHA256, ECDH–RSA–AES128–SHA256, DHE–DSS–AES128–GCM–SHA256, DHE–DSS–AES128–SHA256, AES128–GCM–SHA256, AES128–SHA256, ECDHE–ECDSA–AES256–SHA, ECDHE–RSA–AES256–SHA, ECDH–ECDSA–AES256–SHA, ECDH–RSA–AES256–SHA, AES256–SHA, ECDHE–ECDSA–AES128–SHA, ECDHE–RSA–AES128–SHA, DHE–DSS–AES128–SHA, ECDH–ECDSA–AES128–SHA, ECDH–RSA–AES128–SHA, AES128–SHA

说明

使用 **DTLS** 时，指定 **DTLS** 服务端支持的 **Cipher** 列表。

插件 emqx_dashboard

dashboard.default_user.login & dashboard.default_user.password

Type	Default
string	-

说明

Dashboard 默认用户的认证数据。`dashboard.default_user.login` 与 `dashboard.default_user.password` 必须同时存在。

dashboard.listener.http

Type	Default
integer	18083
string	0.0.0.0:18083

说明

HTTP 监听器的监听端口。

使用 `ip:port` 监听指定网卡端口。默认 `0.0.0.0:18083` 会监听所有网卡的 **18083** 端口。

dashboard.listener.http.acceptors

Type	Default
integer	4

说明

此监听器将创建的监听进程数量。

dashboard.listener.http.max_clients

Type	Default
integer	512

说明

此监听器允许同时建立的最大连接数量限制。

dashboard.listener.http.inet6

Type	Optional Value	Default
enum	true , false	false

说明

是否设置套接字允许 **IPv6** 连接。

dashboard.listener.http.ipv6_v6only

Type	Optional Value	Default
enum	true , false	false

说明

是否限制套接字仅使用 **IPv6**，禁止任何 **IPv4** 连接。仅适用于 **IPv6** 套接字，即仅在 `dashboard.listener.http.inet6` 被设置为 `true` 时此配置项的值有实际意义。需要注意的是，在某些操作系统上，例如 **Windows**，此配置项唯一允许的值为 `true`。

dashboard.listener.https

Type	Default
integer	18084

说明

HTTPS 监听器的监听端口，默认此监听器被禁用。

dashboard.listener.https.acceptors

Type	Default
integer	2

说明

同 `dashboard.listener.http.acceptors`。

dashboard.listener.https.max_clients

Type	Default
integer	512

说明

同 `dashboard.listener.http.max_clients`。

dashboard.listener.https.inet6

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>false</code>

说明

同 `dashboard.listener.http.inet6`。

dashboard.listener.https.ipv6_v6only

Type	Optional Value	Default
enum	<code>true</code> , <code>false</code>	<code>false</code>

说明

同 `dashboard.listener.http.ipv6_v6only`。

dashboard.listener.https.keyfile

Type	Default
string	<code>etc/certs/key.pem</code>

说明

服务端私钥文件路径。

dashboard.listener.https.certfile

Type	Default
string	<code>etc/certs/cert.pem</code>

说明

服务端证书文件路径。

dashboard.listener.https.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

指定 **SSL** 的 **CA** 证书文件 (**PEM**)。该文件应包含发布服务器证书的所有中间**CA**证书以及根证书。该文件还应包含所有受信**CA**的证书用以用于验证客户端的证书。

dashboard.listener.https.dhfile

Type	Default
string	etc/certs/dh-params.pem

说明

如果协商使用 **Diffie Hellman** 密钥交换的密码套件，则可以通过此配置项指定包含 **PEM** 编码的 **Diffie Hellman** 参数的文件路径。如果未指定，则使用默认参数。

dashboard.listener.https.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	verify_peer

说明

`verify_none` 表示关闭对端证书验证，服务端不会向客户端发出证书请求。`verify_peer` 表示开启对端证书验证，服务端会向客户端发出证书请求。当此配置项被设置为 `verify_peer` 时，通常需要配合 `dashboard.listener.https.fail_if_no_peer_cert` 一起使用，以指定是否强制客户端提供证书。

dashboard.listener.https.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	true

说明

必须配合 `dashboard.listener.https.verify` 一起使用。如果设置为 `true`，则服务端向客户端请求证书时如果客户端不提供证书将导致握手失败。如果设置为 `false`，则客户端即使不提供证书也能握手成功。

dashboard.listener.https.tls_versions

Type	Default
<code>string</code>	<code>tlsv1.3,tlsv1.2,tlsv1.1,tlsv1</code>

说明

指定服务端支持的 **TLS** 协议版本，版本之间由 `,` 分隔，支持的 **TLS** 协议版本有：`tlsv1.3`，`tlsv1.2`，`tlsv1.1`，`tlsv1`，`sslv3`。

dashboard.listener.https.ciphers

Type	Default
<code>string</code>	<code>ECDHE-ECDSA-AES256-GCM-SHA384,ECDHE-RSA-AES256-GCM-SHA384,ECDHE-ECDSA-AES256-SHA384,ECDHE-RSA-AES256-SHA384,ECDHE-ECDSA-DES-CBC3-SHA,ECDH-ECDSA-AES256-GCM-SHA384,ECDH-RSA-AES256-GCM-SHA384,ECDH-ECDSA-AES256-SHA384,ECDH-RSA-AES256-SHA384,DHE-DSS-AES256-GCM-SHA384,DHE-DSS-AES256-SHA256,AES256-GCM-SHA384,AES256-SHA256,ECDHE-ECDSA-AES128-GCM-SHA256,ECDHE-RSA-AES128-GCM-SHA256,ECDHE-ECDSA-AES128-SHA256,ECDHE-RSA-AES128-SHA256,ECDH-ECDSA-AES128-GCM-SHA256,ECDH-RSA-AES128-GCM-SHA256,ECDH-ECDSA-AES128-SHA256,ECDH-RSA-AES128-SHA256,DHE-DSS-AES128-GCM-SHA256,DHE-DSS-AES128-SHA256,AES128-GCM-SHA256,AES128-SHA256,ECDHE-ECDSA-AES256-SHA,ECDHE-RSA-AES256-SHA,DHE-DSS-AES256-SHA,ECDH-ECDSA-AES256-SHA,ECDH-RSA-AES256-SHA,AES256-SHA,ECDHE-ECDSA-AES128-SHA,ECDHE-RSA-AES128-SHA,DHE-DSS-AES128-SHA,ECDH-ECDSA-AES128-SHA,ECDH-RSA-AES128-SHA,AES128-SHA</code>

说明

指定服务端支持的加密套件。

dashboard.listener.https.secure_renegotiate

Type	Optional Value	Default
<code>enum</code>	<code>on, off</code>	<code>off</code>

说明

指定是否启动安全重协商机制。

dashboard.listener.https.reuse_sessions

Type	Optional Value	Default
<code>enum</code>	<code>on, off</code>	<code>on</code>

说明

指定是否启用会话复用机制。

dashboard.listener.https.honor_cipher_order

Type	Optional Value	Default
enum	<code>on</code> , <code>off</code>	<code>on</code>

说明

如果设置为 `on`，则使用服务器的首选项进行密码选择。如果设置为 `off`，则使用客户端的首选项。

插件 emqx_lwm2m

lwm2m.port

Type	Default
integer	5683

说明

指定 **LwM2M** 使用的 **UDP** 端口。

lwm2m.lifetime_min

Type	Default
duration	<code>1s</code>

说明

指定允许的 **LwM2M lifetime** 最小值，单位：秒。

lwm2m.lifetime_max

Type	Default
duration	<code>86400s</code>

说明

指定允许的 **LwM2M lifetime** 最大值，单位：秒。

lwm2m.qmode_time_window

Type	Default
integer	22

说明

指定 **LwM2M Q** 模式使用的窗口大小，单位：秒。

这个窗口期之内可以下发执行给 **Q** 模式的设备，过了窗口期则缓存下行数据。

lwm2m.lb

Type	Optional Value	Default
enum	coaproxy , undefined	undefined

说明

设置是否使用 **coaproxy**。设置为 `undefined` 则不使用 **coaproxy**。

lwm2m.auto_observe

Type	Optional Value	Default
enum	on , off	off

说明

在设备注册后是否自动下发 **observe** 命令。

lwm2m.mountpoint

Type	Default
string	lwm2m/%e/

说明

设置 **LwM2M** 主题的挂载点。支持以下通配符：

- '%e': Endpoint Name
- '%a': IP Address

lwm2m.topics.command

Type	Default
string	dn/#

说明

设备注册完成后，需要订阅的下行命令主题。

lwm2m.topics.response

Type	Default
string	up/resp

说明

设备的上行回复需要发布到哪个主题。

lwm2m.topics.notify

Type	Default
string	up/notify

说明

设备的上行报告消息 (**notify**) 需要发布到哪个主题。

lwm2m.topics.register

Type	Default
string	up/resp

说明

设备的上行注册消息 (**register**) 需要发布到哪个主题。

lwm2m.topics.update

Type	Default
string	up/resp

lwm2m.update_msg_publshn_condition

Type	Optional Value	Default
enum	contains_object_list , always	contains_object_list

Description

发布 **UPDATE** 事件的条件。可以为下列两种之一：

- **contains_object_list:** 仅当 **UPDATE** 消息包含 `object list` 时发布
- **always:** 总是发布

说明

设备的上行更新消息 (**update**) 需要发布到哪个主题。

lwm2m.opts.buffer

Type	Default
bytesize	1024KB

说明

UDP 调优参数，指定 **UDP** 用户态缓存大小。

lwm2m.opts.redbuf

Type	Default
bytesize	1024KB

说明

UDP 调优参数，指定 **UDP** 接收缓存大小。

lwm2m.opts.sndbuf

Type	Default
bytesize	1024KB

说明

UDP 调优参数，指定 **UDP** 发送缓存大小。

lwm2m.opts.read_packets

Type	Default
integer	20

说明

UDP 调优参数，指定每次从 **UDP socket** 读取多少个报文。

lwm2m.certfile

Type	Default
string	etc/certs/cert.pem

说明

指定 **UDP DTLS** 使用的证书文件。

lwm2m.keyfile

Type	Default
string	etc/certs/key.pem

说明

指定 **UDP DTLS** 使用的私钥文件。

lwm2m.xml_dir

Type	Default
dir	etc/lwm2m_xml

说明

指定 **LwM2M Object** 定义文件存放的目录。

插件 `emqx_management`

management.max_row_limit

Type	Default
integer	10000

说明

分页查询时返回的最大记录数量。

management.default_application.id

Type	Default
string	admin

说明

默认应用的 **AppId**。

management.default_application.secret

Type	Default
string	public

说明

默认应用的 **AppSecret**。

management.listener.http

Type	Default
integer	8081
string	0.0.0.0:8081

说明

HTTP 监听器的监听端口。

使用 `ip:port` 监听指定网卡端口。默认 `0.0.0.0:8081` 会监听所有网卡的 **8081** 端口。

management.listener.http.acceptors

Type	Default
integer	2

说明

此监听器将创建的监听进程数量。

management.listener.http.max_clients

Type	Default
integer	512

说明

此监听器允许同时建立的最大连接数量限制。

management.listener.http.backlog

Type	Default
integer	512

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 **TCP** 连接队列最大个数。

management.listener.http.send_timeout

Type	Default
duration	15s

说明

HTTP 报文发送超时时间。

management.listener.http.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

HTTP 报文发送超时后，是否关闭该连接。

management.listener.http.inet6

Type	Optional Value	Default
enum	true , false	false

说明

是否设置套接字允许 **IPv6** 连接。

management.listener.http.ipv6_v6only

Type	Optional Value	Default
enum	true , false	false

说明

是否限制套接字仅使用 **IPv6**，禁止任何 **IPv4** 连接。仅适用于 **IPv6** 套接字，即仅在

`management.listener.http.inet6` 被设置为 `true` 时此配置项的值有实际意义。需要注意的是，在某些操作系统上，例如 **Windows**，此配置项唯一允许的值为 `true`。

management.listener.https

Type	Default	Example
integer	-	8081
string	-	<code>0.0.0.0:8081</code>

说明

HTTPS 监听器的监听端口。

使用 `ip:port` 监听指定网卡端口。默认 `0.0.0.0:8081` 会监听所有网卡的 **8081** 端口。

management.listener.https.acceptors

Type	Default
integer	2

说明

此监听器将创建的监听进程数量。

management.listener.https.max_clients

Type	Default
integer	512

说明

此监听器允许同时建立的最大连接数量限制。

management.listener.https.backlog

Type	Default
integer	512

说明

TCP 连接队列的最大长度。它表明了系统中允许的正在三次握手的 TCP 连接队列最大个数。

management.listener.https.send_timeout

Type	Default
duration	15s

说明

HTTPS 报文发送超时时间。

management.listener.https.send_timeout_close

Type	Optional Value	Default
enum	on , off	on

说明

HTTPS 报文发送超时后，是否关闭该连接。

management.listener.https.keyfile

Type	Default
string	etc/certs/key.pem

说明

服务端私钥文件路径。

management.listener.https.certfile

Type	Default
<code>string</code>	<code>etc/certs/cert.pem</code>

说明

服务端证书文件路径。

management.listener.https.cacertfile

Type	Default
<code>string</code>	<code>etc/certs/cacert.pem</code>

说明

指定 **SSL 的 CA 证书文件 (PEM)**。该文件应包含发布服务器证书的所有中间**CA**证书以及根证书。该文件还应包含所有受信**CA**的证书用以用于验证客户端的证书。

management.listener.https.verify

Type	Optional Value	Default
<code>enum</code>	<code>verify_peer</code> , <code>verify_none</code>	<code>verify_peer</code>

说明

`verify_none` 表示关闭对端证书验证，服务端不会向客户端发出证书请求。`verify_peer` 表示开启对端证书验证，服务端会向客户端发出证书请求。当此配置项被设置为 `verify_peer` 时，通常需要配合 `management.listener.https.fail_if_no_peer_cert` 一起使用，以指定是否强制客户端提供证书。

management.listener.https.fail_if_no_peer_cert

Type	Optional Value	Default
<code>enum</code>	<code>true</code> , <code>false</code>	<code>true</code>

说明

必须配合 `management.listener.https.verify` 一起使用。如果设置为 `true`，则服务端向客户端请求证书时如果客户端不提供证书将导致握手失败。如果设置为 `false`，则客户端即使不提供证书也能握手成功。

management.listener.https.inet6

Type	Optional Value	Default
enum	true , false	false

说明

是否设置套接字允许 **IPv6** 连接。

management.listener.https.ipv6_v6only

Type	Optional Value	Default
enum	true , false	false

说明

是否限制套接字仅使用 **IPv6**，禁止任何 **IPv4** 连接。仅适用于 **IPv6** 套接字，即仅在

`management.listener.https.inet6` 被设置为 `true` 时此配置项的值有实际意义。需要注意的是，在某些操作系统上，例如 **Windows**，此配置项唯一允许的值为 `true`。

插件 emqx_retainer

retainer.storage_type

Type	Optional Value	Default
enum	ram , disc , disc_only	ram

说明

保留消息的存储类型，以下选项可用：

`ram`

保留消息仅存储在内存中。

`disc`

保留消息同时存储在内存和磁盘中。

`disc_only`

保留消息仅存储在磁盘中。

retainer.max_retained_messages

Type	Default
integer	0

说明

保留消息的存储数量限制。一旦存储数量达到限制，可以替换已存在的保留消息，但不能为新的主题存储保留消息。**0** 表示没有限制。

retainer.max_payload_size

Type	Default
bytesize	1MB

说明

允许存储的保留消息的 **Payload** 最大长度限制。如果 **Payload** 超出最大限制，该保留消息可以被正常处理，但不会存储在服务端。

retainer.expiry_interval

Type	Default
duration	0

说明

保留消息的过期间隔，仅对协议版本低于 **MQTT v5.0** 的客户端生效，**MQTT v5.0** 客户端的保留消息过期间隔将以 `Message Expiry Interval` 的值为准。**0** 表示永不过期。

插件 emqx_rule_engine

rule-engine.ignore_sys_message

Type	Optional Value	Default
enum	on , off	on

说明

忽略系统消息 (**\$SYS**)。启用此选项规则引擎将不会处理系统消息。

rule-engine.events.<event-name>

Type	Optional Value	Default
enum	on , off	off

说明

设置是否发布事件消息。可指定事件消息的 **QoS**，例如：

```
1 rule-engine.events.client_connected = on, qos1
2
```

若启用此选项，规则引擎会将系统消息使用 `$events/<event-name>` 主题发布出来。可支持的 `<event-name>` 有：

- **client_connected**: 客户端登录完成
- **client_disconnected**: 客户端下线
- **session_subscribed**: 客户端订阅
- **session_unsubscribed**: 客户端取消订阅
- **message_delivered**: 消息已投递
- **message_acked**: 消息已确认
- **message_dropped**: 消息被丢弃

如果禁用此选项，事件消息将不会发布，但事件规则仍然可以使用。例如，即使

```
rule_engine.events.client_connected = off
```

```
1 SELECT * FROM "$events/client_connected"
```

插件 emqx_sn

mqtt.sn.port

Type	Default
string	1884

说明

`emqx_sn` 监听的 **UDP** 端口。

mqtt.sn.advertise_duration

Type	Default
duration	15s

说明

ADVERTISE 消息广播间隔，单位：秒。

mqtt.sn.gateway_id

Type	Default
integer	1

说明

ADVERTISE 中的 MQTT-SN 网关 ID。

mqtt.sn.enable_stats

Type	Optional Value	Default
enum	on , off	off

说明

是否开启客户端状态统计信息。

mqtt.sn.enable_qos3

Type	Optional Value	Default
enum	on , off	off

说明

是否处理 QoS 为 -1 的消息。

mqtt.sn.idle_timeout

Type	Default
duration	30s

说明

建立后的发呆时间，如果这段时间内未收到任何报文，则会关闭该连接。

mqtt.sn.predefined.topic.0

Type	Default
string	reserved

说明

预定义的 **Topic** 与 **TopicId** 映射。**Id** 为 **0** 的主题是保留项，固定为 `reserved`。例如，预定义主题 `foo/bar` 的 **Id** 为 `1`：

```
1 mqtt.sn.predefined.topic.1 = foo/bar
```

mqtt.sn.username

Type	Default
string	<code>mqtt_sn_user</code>

说明

`emqx_sn` 连接至 **EMQX** 的用户名。

mqtt.sn.password

Type	Default
string	<code>abc</code>

说明

`emqx_sn` 连接至 **EMQX** 的密码。

插件 `emqx_prometheus`

prometheus.push.gateway.server

Type	Default
string	<code>http://127.0.0.1:9091</code>

说明

指定 **Prometheus gateway** 的 **URI**。

prometheus.interval

Type	Default
integer	15000

说明

指定 **Stats** 数据的收集间隔，单位：毫秒。

prometheus.collector.<N>

Type	Default
string	emqx_prometheus

说明

指定 **Prometheus** 的 **Collector**。

插件 emqx_stomp

stomp.listener

Type	Default
integer	61613

说明

指定 **Stomp** 插件监听的本地端口。

stomp.listener.acceptors

Type	Default
integer	4

说明

指定 **Stomp** 服务 **Acceptor** 线程池的大小。

stomp.listener.max_connections

Type	Default
integer	512

说明

指定 **Stomp** 服务支持的最大连接数。

stomp.listener.ssl

Type	Optional Value	Default
enum	on , off	off

说明

指定是否使用 **SSL**。

stomp.listener.keyfile

Type	Default
string	etc/certs/key.pem

说明

若使用 **SSL**, 指定 **SSL** 的私钥文件。

stomp.listener.certfile

Type	Default
string	etc/certs/cert.pem

说明

若使用 **SSL**, 指定 **SSL** 的证书文件。

stomp.listener.cacertfile

Type	Default
string	etc/certs/cacert.pem

说明

若使用 **SSL**, 指定 **SSL** 的 **CA** 证书文件。

stomp.listener.dhfile

Type	Default
string	etc/certs/dh-params.pem

若使用 **SSL**, 指定 **Ephemeral Diffie-Helman** 算法使用的 **key** 文件。

stomp.listener.verify

Type	Optional Value	Default
enum	verify_peer , verify_none	verify_peer

说明

若使用 **SSL**, 指定握手过程中是否校验客户端。

stomp.listener.fail_if_no_peer_cert

Type	Optional Value	Default
enum	true , false	false

说明

若使用 **SSL**, **SSL** 握手过程中若客户端没有证书, 是否让握手失败。

stomp.listener.tls_versions

Type	Default
string	tlsv1.2,tlsv1.1,tlsv1

说明

若使用 **SSL**, 指定服务端支持的 **SSL** 的版本列表。

stomp.listener.handshake_timeout

Type	Default
duration	15s

说明

若使用 **SSL**, 指定 **SSL** 握手过程的超时时间。

stomp.listener.ciphers

Type	Default
string	ECDHE–ECDSA–AES256–GCM–SHA384, ECDHE–RSA–AES256–GCM–SHA384, ECDHE–ECDSA–AES256–SHA384, ECDHE–RSA–AES256–SHA384, ECDHE–ECDSA–DES–CBC3–SHA, ECDH–ECDSA–AES256–GCM–SHA384, ECDH–RSA–AES256–GCM–SHA384, ECDH–ECDSA–AES256–SHA384, ECDH–RSA–AES256–SHA384, DHE–DSS–AES256–GCM–SHA384, DHE–DSS–AES256–SHA256, AES256–GCM–SHA384, AES256–SHA256, ECDHE–ECDSA–AES128–GCM–SHA256, ECDHE–RSA–AES128–GCM–SHA256, ECDHE–ECDSA–AES128–SHA256, ECDHE–RSA–AES128–SHA256, ECDH–ECDSA–AES128–GCM–SHA256, ECDH–RSA–AES128–SHA256, ECDH–RSA–AES128–GCM–SHA256, DHE–DSS–AES128–GCM–SHA256, DHE–DSS–AES128–SHA256, AES128–GCM–SHA256, AES128–SHA256, ECDHE–ECDSA–AES256–SHA, ECDHE–RSA–AES256–SHA, DHE–DSS–AES256–SHA, ECDH–ECDSA–AES256–SHA, ECDH–RSA–AES256–SHA, AES256–SHA, ECDHE–ECDSA–AES128–SHA, ECDHE–RSA–AES128–SHA, DHE–DSS–AES128–SHA, ECDH–ECDSA–AES128–SHA, ECDH–RSA–AES128–SHA, AES128–SHA

说明

若使用 **SSL**, 指定服务端支持的 **Cipher** 列表。

stomp.listener.secure_renegotiate

Type	Optional Value	Default
enum	on , off	off

说明

若使用 **SSL**, 指定在客户端不遵循 **RFC 5746** 的情况下, 是否拒绝 **renegotiation** 请求。

stomp.listener.reuse_sessions

Type	Optional Value	Default
enum	on , off	on

说明

若使用 **SSL**, 指定是否支持 **SSL session** 重用。

stomp.listener.honor_cipher_order

Type	Optional Value	Default
enum	on , off	on

说明

若使用 **SSL**, 指定是否使用服务端的偏好设置选择 **Ciphers**。

stomp.default_user.login

Type	Default
string	guest

说明

指定 **Stomp** 插件登录使用的 **Username**。

stomp.default_user.passcode

Type	Default
string	guest

说明

指定 **Stomp** 插件登录使用的 **Password**。

stomp.allow_anonymous

Type	Optional Value	Default
enum	true , false	true

说明

是否允许匿名登录。

stomp.frame.max_headers

Type	Default
integer	10

说明

指定 **Stomp** 最大报文头数量。

stomp.frame.max_header_length

Type	Default
integer	1024

说明

指定 **Stomp** 最大报文头长度。

stomp.frame.max_body_length

Type	Default
integer	8192

说明

指定 **Stomp** 最大报文体长度。

插件 emqx_web_hook

web.hook.url

Type	Default
string	http://127.0.0.1:80

说明

Webhook 请求转发的目的 **Web** 服务器地址。

web.hook.headers.<Any>

示例

```
1 web.hook.headers.content-type = application/json
2 web.hook.headers.accept = */*
```

说明

指定 **HTTP** 请求头部中的数据。<Key> 指定 **HTTP** 请求头部中的字段名，此配置项的值为相应的字段值。<Key> 可以是标准的 **HTTP** 请求头部字段，也可以自定义的字段，可以配置多个不同的请求头部字段。

web.hook.encoding_of_payload_field

Type	Optional Value	Default
enum	plain , base62 , base64	plain

说明

PUBLISH 报文中 **Payload** 字段的编码格式。

web.hook.ssl.cacertfile

Type	Default
string	-

说明

CA 证书文件路径。

web.hook.ssl.certfile

Type	Default
string	-

说明

客户端证书文件路径。

web.hook.ssl.keyfile

Type	Default
string	-

说明

客户端私钥文件路径。

web.hook.ssl.verify

Type	Optional Value	Default
enum	true , false	false

说明

指定是否校验对端证书。

web.hook.ssl.pool_size

Type	Default
integer	32

说明

HTTP 连接进程池大小。

web.hook.rule.client.connect.1

Type	Default
string	{"action": "on_client_connect"}

说明

转发 收到连接报文 事件。

web.hook.rule.client.connack.1

Type	Default
string	{"action": "on_client_connack"}

说明

转发 下发连接应答 事件。

web.hook.rule.client.connected.1

Type	Default
string	{"action": "on_client_connected"}

说明

转发 客户端成功接入 事件。

web.hook.rule.client.disconnected.1

Type	Default
string	{"action": "on_client_disconnected"}

说明

转发 `客户端已断开` 事件。

web.hook.rule.client.subscribe.1

Type	Default
<code>string</code>	<code>{"action": "on_client_subscribe"}</code>

说明

转发 `将订阅` 事件。

web.hook.rule.client.unsubscribe.1

Type	Default
<code>string</code>	<code>{"action": "on_client_unsubscribe"}</code>

说明

转发 `将取消订阅` 事件。

web.hook.rule.session.subscribed.1

Type	Default
<code>string</code>	<code>{"action": "on_session_subscribed"}</code>

说明

转发 `已订阅` 事件。

web.hook.rule.session.unsubscribed.1

Type	Default
<code>string</code>	<code>{"action": "on_session_unsubscribed"}</code>

说明

转发 `已取消订阅` 事件。

web.hook.rule.session.terminated.1

Type	Default
string	{"action": "on_session_terminated"}

说明

转发 会话已终止 事件。

web.hook.rule.message.publish.1

Type	Default
string	{"action": "on_message_publish"}

说明

转发 消息发布 事件。

web.hook.rule.message.delivered.1

Type	Default
string	{"action": "on_message_delivered"}

说明

转发 消息已投递 事件。

web.hook.rule.message.acked.1

Type	Default
string	{"action": "on_message_acked"}

说明

转发 消息已应答 事件。

license.file

Type	Default
string	etc/emqx.lic

说明

企业版证书存放的路径。

license.connection_high_watermark_alarm

Type	Default
percent	80%

说明

连接数高水位线告警，达到企业版证书允许实时在线连接数的百分比。超出水位线时会产生告警，不影响实际使用，

- 发生告警后，可以参照[【如何更新证书？】](#)([..../faq/use-guide.md#怎样更新 EMQX license?](#))进行热更新。
- 当连接数超过最大允许值时，新客户端会被拒绝连接，已连接的客户端不受影响。

license.connection_low_watermark_alarm

Type	Default
percent	75%

说明

连接数低水位线告警，低于达到企业版证书允许实时在线连接数的百分比则解除告警。

使用环境变量修改配置

默认情况下 **EMQX** 使用带有 `EMQX_` 的前缀的环境变量来覆盖配置文件中的配置项

环境变量名称到配置文件键值名称映射规则如下：

- 将 `EMQX_` 前缀移除
- 大写字符替换成小写
- 双下划线 `_` 替换成本点 `.`

示例

```
1 # management.listener.http = 9000
2 $ export EMQX_MANAGEMENT_LISTENER_HTTP=9000
3 $ _build/emqx/rel/emqx/bin/emqx console
4 ...
5 ...
6 Starting emqx on node emqx@127.0.0.1
7 Start http:management listener on 9000 successfully.
```

Tip

环境变量只在运行 **EMQX** 的终端中有效，且没有持久化，在一个终端中设置的环境变量无法在另一个终端中使用。

命令行接口

EMQX 提供了 `./bin/emqx_ctl` 的管理命令行，用于用户对 **EMQX** 进行管理、配置、查询。

status 命令

查询 **EMQX** 运行状态：

```
1 $ ./bin/emqx_ctl status
2 Node 'emqx@127.0.0.1' is started
3 emqx v4.0.0 is running
```

sh

mgmt 命令

mgmt 命令查询应用程序。

命令	描述
<code>mgmt list</code>	列出应用程序列表
<code>mgmt insert <AppId> <Name></code>	添加允许访问 HTTP API 的应用程序
<code>mgmt update <AppId> <status></code>	更新允许访问 HTTP API 的应用程序
<code>mgmt lookup <AppId></code>	获取允许访问 HTTP API 的应用程序详情
<code>mgmt delete <AppId></code>	删除允许访问 HTTP API 的应用程序

mgmt list

列出应用程序列表：

```
1 $ ./bin/emqx_ctl mgmt list
2 app_id: 901abdba8eb8c, secret: MjgzMzQ5MjM1MzUzMTC4MjgyMjE3NzU40DcwMDg0NjQ40TG, name: hello, de
c: , status: true, expired: undefined
```

sh

mgmt insert <AppId> <Name>

添加 **HTTP API** 的应用程序：

```
1 $ ./bin/emqx_ctl mgmt insert dbcb6e023370b world
2 AppSecret: MjgzMzQ5MjYyMTY30Dk4MjA5NzMw0DEx0DMxMDM1NDk0NDA
```

sh

mgmt update <AppId> <status>

更新 HTTP API 的应用程序:

```
1 $ ./bin/emqx_ctl mgmt update dbcb6e023370b stop
2 update successfully.
```

sh

mgmt lookup <AppId>

获取 HTTP API 的应用程序详情:

```
1 $ ./bin/emqx_ctl mgmt lookup dbcb6e023370b
2 app_id: dbcb6e023370b
3 secret: MjgzMzQ5MjYyMTY30Dk4MjA5NzMw0DEx0DMxMDM1NDk0NDA
4 name: world
5 desc: Application user
6 status: stop
7 expired: undefined
```

sh

mgmt delete <AppId>

删除 HTTP API 的应用程序:

```
1 $ ./bin/emqx_ctl mgmt delete dbcb6e023370b
2 ok
```

sh

broker 命令

broker 命令查询服务器基本信息，启动时间，统计数据与性能数据。

命令	描述
broker	查询 EMQX 描述、版本、启动时间
broker stats	查询连接 (Connection)、会话 (Session)、主题 (Topic)、订阅 (Subscription)、路由 (Route) 统计信息
broker metrics	查询 MQTT 报文 (Packet)、消息 (Message) 收发统计

查询 EMQX 基本信息，包括版本、启动时间等:

```
1 $ ./bin/emqx_ctl broker
2 sysdescr : EMQX Broker
3 version   : 4.0.0
4 uptime    : 4 minutes, 52 seconds
5 datetime  : 2020-02-21 09:39:58
```

sh

broker stats

查询服务器客户端连接 (**Connections**)、主题 (**Topics**)、订阅 (**Subscriptions**)、路由 (**Routes**) 统计:

```

1 $ ./bin/emqx_ctl broker stats
2 channels.count : 0
3 channels.max : 0
4 connections.count : 0
5 connections.max : 0
6 resources.count : 0
7 resources.max : 0
8 retained.count : 3
9 retained.max : 3
10 routes.count : 0
11 routes.max : 0
12 sessions.count : 0
13 sessions.max : 0
14 suboptions.count : 0
15 suboptions.max : 0
16 subscribers.count : 0
17 subscribers.max : 0
18 subscriptions.count : 0
19 subscriptions.max : 0
20 subscriptions.shared.count : 0
21 subscriptions.shared.max : 0
22 topics.count : 0
23 topics.max : 0

```

sh

broker metrics

查询服务器流量 (**Bytes**)、MQTT 报文 (**Packets**)、消息 (**Messages**) 收发统计:

```

1 $ ./bin/emqx_ctl broker metrics
2 bytes.received : 0
3 bytes.sent : 0
4 client.auth.anonymous : 0
5 client.authenticate : 0
6 client.check_acl : 0
7 client.connack : 0
8 client.connect : 0
9 client.connected : 0
10 client.disconnected : 0
11 client.subscribe : 0
12 client.unsubscribe : 0
13 delivery.dropped : 0
14 delivery.dropped.expired : 0
15 delivery.dropped.no_local : 0
16 delivery.dropped.qos0_msg : 0
17 delivery.dropped.queue_full : 0
18 delivery.dropped.too_large : 0
19 messages.acked : 0
20 messages.delayed : 0
21 messages.delivered : 0
22 messages.dropped : 0
23 messages.dropped.expired : 0
24 messages.dropped.no_subscriber: 0
25 messages.forward : 0

```

sh

```
26 messages.publish : 0
27 messages.qos0.received : 0
28 messages.qos0.sent : 0
29 messages.qos1.received : 0
30 messages.qos1.sent : 0
31 messages.qos2.received : 0
32 messages.qos2.sent : 0
33 messages.received : 0
34 messages.retained : 3
35 messages.sent : 0
36 packets.auth.received : 0
37 packets.auth.sent : 0
38 packets.connack.auth_error : 0
39 packets.connack.error : 0
40 packets.connack.sent : 0
41 packets.connect.received : 0
42 packets.disconnect.received : 0
43 packets.disconnect.sent : 0
44 packets.pingreq.received : 0
45 packets.pingresp.sent : 0
46 packets.puback.inuse : 0
47 packets.puback.missed : 0
48 packets.puback.received : 0
49 packets.puback.sent : 0
50 packets.pubcomp.inuse : 0
51 packets.pubcomp.missed : 0
52 packets.pubcomp.received : 0
53 packets.pubcomp.sent : 0
54 packets.publish.auth_error : 0
55 packets.publish.dropped : 0
56 packets.publish.error : 0
57 packets.publish.received : 0
58 packets.publish.sent : 0
59 packets.pubrec.inuse : 0
60 packets.pubrec.missed : 0
61 packets.pubrec.received : 0
62 packets.pubrec.sent : 0
63 packets.pubrel.missed : 0
64 packets.pubrel.received : 0
65 packets.pubrel.sent : 0
66 packets.received : 0
67 packets.sent : 0
68 packets.suback.sent : 0
69 packets.subscribe.auth_error : 0
70 packets.subscribe.error : 0
71 packets.subscribe.received : 0
72 packets.unsubscribe.sent : 0
73 packets.unsubscribe.error : 0
74 packets.unsubscribe.received : 0
75 session.created : 0
76 session.discarded : 0
77 session.resumed : 0
78 session.takeovered : 0
79 session.terminated : 0
```

cluster 命令

cluster 命令可以管理由多个 **EMQX** 节点（进程）组成的集群：

命令	描述
<code>cluster join <Node></code>	加入集群
<code>cluster leave</code>	离开集群
<code>cluster force-leave <Node></code>	从集群删除节点
<code>cluster status</code>	查询集群状态

示例：

为更好地展示 **cluster** 命令，我们可以先在单机上启动两个节点并组成集群，这称之为伪分布式启动模式。由于我们要在单机上启动两个 **emqx** 实例，为避免端口冲突，我们需要对其它节点的监听端口做出调整。

基本思路是复制一份 **emqx** 文件夹然后命名为 **emqx2**，将原先所有 **emqx** 节点监听的端口 **port** 加上一个偏移 **offset** 作为新的 **emqx2** 节点的监听端口。例如，将原先 **emqx** 的**MQTT/TCP** 监听端口由默认的 **1883** 改为了 **2883** 作为 **emqx2** 的 **MQTT/TCP** 监听端口。完成以上操作的自动化脚本可以参照 [集群脚本](#)，具体配置请参见 [配置说明](#) 与 [配置项](#)。

启动 **emqx1**：

```
1 $ cd emqx1 && ./bin/emqx start
```

启动 **emqx2**：

```
1 $ cd emqx2 && ./bin/emqx start
```

使用 `cluster join <Node>` 将两个节点组成集群：

```
1 $ cd emqx2 && ./bin/emqx_ctl cluster join emqx1@127.0.0.1
2 Join the cluster successfully.
3 Cluster status: [{running_nodes, ['emqx1@127.0.0.1', 'emqx2@127.0.0.1']}]
```

任意节点目录下查询集群状态：

```
1 $ ./bin/emqx_ctl cluster status
2 Cluster status: [{running_nodes, ['emqx2@127.0.0.1', 'emqx1@127.0.0.1']}]
```

集群消息路由测试：MQTT 命令行工具使用由 **EMQX** 团队开发的 [emqtt](#) 客户端。

```

1 # emqx1 节点 (1883 端口) 订阅主题 x
2 $ ./bin/emqtt sub -t x -q 1 -p 1883
3 Client emqtt-a7de8fffbe2fbef2fadb sent CONNECT
4 Client emqtt-a7de8fffbe2fbef2fadb subscribed to x
5
6 # 向 emqx2 节点 (2883 端口 ) 发布消息
7 $ ./bin/emqtt pub -t x -q 1 -p 2883 --payload hello
8 Client emqtt-0898fa447676e17479a5 sent CONNECT
9 Client emqtt-0898fa447676e17479a5 sent PUBLISH (Q1, R0, D0, Topic=x, Payload=... (5 bytes))
10 Client emqtt-0898fa447676e17479a5 sent DISCONNECT
11
12 # emqx1 节点 (1883 端口) 收到消息
13 $ ./bin/emqtt sub -t x -q 1 -p 1883
14 hello

```

emqx2 节点离开集群:

```

1 $ cd emqx2 && ./bin/emqx_ctl cluster leave

```

强制 **emqx2** 节点离开集群，需要在集群下的目标节点以外的节点上进行操作:

```

1 $ cd emqx1 && ./bin/emqx_ctl cluster force-leave emqx2@127.0.0.1

```

注意，**EMQX** 不支持一个已经在一个集群中的节点加入另外一个集群，因为这会导致两个集群数据不一致，但支持加入过集群的节点在离开该集群后加入另一个集群。

acl 命令

从 **v4.1** 之后引入了 `modules` 的命令，我们使用以下命令重新加载 **ACL**:

```

1 $ ./bin/emqx_ctl modules reload emqx_mod_acl_internal
2 Module emqx_mod_acl_internal reloaded successfully.

```

在 **v4.1** 之前，则仍然使用:

```

1 $ ./bin/emqx_ctl acl reload
2 ok

```

acl cache-clean

在 **v4.3** 之后，引入了的命令来清理 **ACL** 缓存:

命令	描述
acl cache-clean all	清除集群中所有的 ACL 缓存
acl cache-clean node <Node>	清除指定节点的 ACL 缓存
acl cache-clean <ClientId>	清除指定客户端的 ACL 缓存

clients 命令

clients 命令查询连接的 **MQTT** 客户端。

命令	描述
<code>clients list</code>	列出所有客户端连接
<code>clients show <ClientId></code>	查询指定 ClientId 的客户端
<code>clients kick <ClientId></code>	踢除指定 ClientId 的客户端，连接与会话将一并终结。

clients list

列出所有客户端连接:

```

1 $ ./bin/emqx_ctl clients list
2 Client (mosqsub/43832-airlee.lo, username=test1, peername=127.0.0.1:62135, clean_start=true,
3   keepalive=60, session_expiry_interval=0, subscriptions=0, inflight=0, awaiting_rel=0, deliv
4   ered_msgs=0, enqueued_msgs=0, dropped_msgs=0, connected=true, created_at=1582249657, connect
ed_at=1582249657)
Client (mosqsub/44011-airlee.lo, username=test2, peername=127.0.0.1:64961, clean_start=true,
keepalive=60, session_expiry_interval=0, subscriptions=0, inflight=0, awaiting_rel=0, deliv
ered_msgs=0, enqueued_msgs=0, dropped_msgs=0, connected=true, created_at=1582249657, connect
ed_at=1582249657, disconnected_at=1582249702)
...

```

返回 **Client** 对象的属性:

Name	描述
username	用户名
peername	客户端 IP 与端口
clean_start	MQTT Clean Start
keepalive	MQTT KeepAlive
session_expiry_interval	会话过期间隔
subscriptions	当前订阅数量
inflight	当前正在下发的 QoS 1 和 QoS 2 的消息总数
awaiting_rel	等待客户端发送 PUBREL 的 QoS2 消息数
delivered_msgs	EMQX 向此客户端转发的消息数量 (包含重传)
queued_msgs	消息队列当前长度
dropped_msgs	消息队列达到最大长度后丢弃的消息数量
connected	是否在线
created_at	会话创建时间戳
connected_at	客户端连接时间戳
disconnected_at	客户端断开连接时间戳 (仅当断开连接还保留会话时才会出现)

clients show <ClientId>

查询指定 **ClientId** 的客户端:

```
1 $ ./bin/emqx_ctl clients show "mosqsub/43832-airlee.lo"                                         sh
2 Client (mosqsub/43832-airlee.lo, username=test1, peername=127.0.0.1:62747, clean_start=false
, keepalive=60, session_expiry_interval=7200, subscriptions=0, inflight=0, awaiting_rel=0, d
elivered_msgs=0, queued_msgs=0, dropped_msgs=0, connected=true, created_at=1576479557, con
nected_at=1576479557)
```

clients kick <ClientId>

踢除指定 **ClientId** 的客户端:

```
1 $ ./bin/emqx_ctl clients kick "clientid"                                                 sh
2 ok
```

routes 命令

routes 命令用于查询路由信息。

EMQX 中路由是指主题与节点的映射关系，用于在多个节点之间路由消息。

命令	描述
routes list	列出所有路由
routes show <Topic>	查询指定 Topic 的路由

routes list

列出所有路由:

```
1 $ ./bin/emqx_ctl routes list
2 t2/# -> emqx2@127.0.0.1
3 t/+/* -> emqx2@127.0.0.1,emqx@127.0.0.1
```

sh

routes show <Topic>

查询指定 **Topic** 的路由:

```
1 $ ./bin/emqx_ctl routes show t/+/*
2 t/+/* -> emqx2@127.0.0.1,emqx@127.0.0.1
```

sh

subscriptions 命令

subscriptions 命令查询消息服务器的订阅 (**Subscription**) 表。

命令	描述
subscriptions list	列出所有订阅
subscriptions show <ClientId>	查询指定 ClientId 客户端的订阅
subscriptions add <ClientId> <Topic> <QoS>	手动添加静态订阅
subscriptions del <ClientId> <Topic>	手动删除静态订阅

subscriptions list

列出所有订阅:

```
1 $ ./bin/emqx_ctl subscriptions list
2 mosqsub/91042-airlee.lo -> t/y:1
3 mosqsub/90475-airlee.lo -> t/+/*:2
```

sh

subscriptions show <ClientId>

查询某个 **Client** 的订阅:

```
1 $ ./bin/emqx_ctl subscriptions show 'mosqsub/90475-airlee.lo'
2 mosqsub/90475-airlee.lo -> t/+/x:2
```

sh

subscriptions add <ClientId> <Topic> <QoS>

手动添加订阅关系:

```
1 $ ./bin/emqx_ctl subscriptions add 'mosqsub/90475-airlee.lo' '/world' 1
2 ok
```

sh

subscriptions del <ClientId> <Topic>

手动删除订阅关系:

```
1 $ ./bin/emqx_ctl subscriptions del 'mosqsub/90475-airlee.lo' '/world'
2 ok
```

sh

plugins 命令

plugins 命令用于加载、卸载、查询插件应用。EMQX 通过插件扩展认证、定制功能，插件配置位于 `etc/plugins/` 目录下。

命令	描述
<code>plugins list</code>	列出全部插件 (Plugin)
<code>plugins load <Plugin></code>	加载插件 (Plugin)
<code>plugins unload <Plugin></code>	卸载插件 (Plugin)
<code>plugins reload <Plugin></code>	重载插件 (Plugin)

当配置文件发生更改，如果需要配置立即生效，你可以执行 `emqx_ctl reload <Plugin>` 命令，即使插件在配置修改时并未处于运行状态，你也应当使用此命令而不是 `emqx_ctl load <Plugin>`，因为 `emqx_ctl load <Plugin>` 不会编译新的配置文件。

plugins list

列出全部插件:

```
1 $ ./bin/emqx_ctl plugins list
2 ...
3 Plugin(emqx_auth_http, description=EMQX Authentication/ACL with HTTP API, active=false)
4 Plugin(emqx_auth_jwt, description=EMQX Authentication with JWT, active=false)
5 Plugin(emqx_auth_ldap, description=EMQX Authentication/ACL with LDAP, active=false)
6 ...
```

sh

插件属性:

Name	描述
version	插件版本
description	插件描述
active	是否已加载

plugins load <Plugin>

加载插件:

```
1 $ ./bin/emqx_ctl plugins load emqx_lua_hook
2 Plugin emqx_lua_hook loaded successfully.
```

sh

plugins unload <Plugin>

卸载插件:

```
1 $ ./bin/emqx_ctl plugins unload emqx_lua_hook
2 Plugin emqx_lua_hook unloaded successfully.
```

sh

plugins reload <Plugin>

重载插件:

```
1 $ ./bin/emqx_ctl plugins reload emqx_lua_hook
2 Plugin emqx_lua_hook reloaded successfully.
```

sh

modules 命令

自 v4.1 之后，引入了 `modules` 命令用于在运行时管理 EMQX 内置的模块。

命令	描述
<code>modules list</code>	列出全部内置模块 (Module)
<code>modules load <Module></code>	加载内置模块 (Module)
<code>modules unload <Module></code>	卸载内置模块 (Module)
<code>modules reload <Module></code>	重载内置模块 (Module)

modules list

列出全部内置模块:

```

1 $ ./bin/emqx_ctl modules list
2 Module(emqx_mod_delayed, description=EMQX Delayed Publish Module, active=false)
3 Module(emqx_mod_topic_metrics, description=EMQX Topic Metrics Module, active=false)
4 Module(emqx_mod_subscription, description=EMQX Subscription Module, active=false)
5 Module(emqx_mod_acl_internal, description=EMQX Internal ACL Module, active=true)
6 Module(emqx_mod_rewrite, description=EMQX Topic Rewrite Module, active=false)
7 Module(emqx_mod_presence, description=EMQX Presence Module, active=true)

```

modules load

加载内置模块:

```

1 $ ./bin/emqx_ctl modules load emqx_mod_delayed
2 Module emqx_mod_delayed loaded successfully.

```

modules unload

卸载内置模块:

```

1 $ ./bin/emqx_ctl modules unload emqx_mod_delayed
2 Module emqx_mod_delayed unloaded successfully.

```

modules reload

重载内置模块:

```

1 $ ./bin/emqx_ctl modules reload emqx_mod_acl_internal
2 Module emqx_mod_acl_internal reloaded successfully.

```

vm 命令

vm 命令用于查询 **Erlang** 虚拟机负载、内存、进程、**IO** 信息。

命令	描述
vm	等同于 vm all
vm all	查询 VM 全部信息
vm load	查询 VM 负载
vm memory	查询 VM 内存
vm process	查询 VM Erlang 进程数量
vm io	查询 VM io 最大文件句柄
vm ports	查询 VM 的端口

viii.111

查询 **VM** 全部信息，包括负载、内存、**Erlang** 进程数量等：

```
1 $ ./bin/emqx_ctl vm all
2   cpu/load1          : 4.22
3   cpu/load5          : 3.29
4   cpu/load15         : 3.16
5   memory/total       : 99995208
6   memory/processes   : 38998248
7   memory/processes_used : 38938520
8   memory/system      : 60996960
9   memory/atom         : 1189073
10  memory/atom_used   : 1173808
11  memory/binary      : 100336
12  memory/code        : 25439961
13  memory/ets          : 7161128
14  process/limit       : 2097152
15  process/count      : 315
16  io/max_fds         : 10240
17  io/active_fds      : 0
18  ports/count        : 18
19  ports/limit         : 1048576
```

sh

vm load

查询 **VM** 负载：

```
1 $ ./bin/emqx_ctl vm load
2   cpu/load1          : 2.21
3   cpu/load5          : 2.60
4   cpu/load15         : 2.36
```

sh

vm memory

查询 **VM** 内存：

```
1 $ ./bin/emqx_ctl vm memory
2   memory/total       : 23967736
3   memory/processes   : 3594216
4   memory/processes_used : 3593112
5   memory/system      : 20373520
6   memory/atom         : 512601
7   memory/atom_used   : 491955
8   memory/binary      : 51432
9   memory/code        : 13401565
10  memory/ets          : 1082848
```

sh

vm process

查询 **Erlang** 进程数量及其限制：

```

1 $ ./bin/emqx_ctl vm process
2   process/limit      : 2097152
3   process/count      : 314

```

sh

vm io

查询文件描述符数量及其限制:

```

1 $ ./bin/emqx_ctl vm io
2   io/max_fds        : 10240
3   io/active_fds     : 0

```

sh

vm ports

查询端口占用数量及其限制:

```

1 $ ./bin/emqx_ctl vm ports
2   ports/count       : 18
3   ports/limit        : 1048576

```

sh

mnesia 命令

查询 **mnesia** 数据库系统状态。

log 命令

log 命令用于设置日志等级。访问 [Documentation of logger](#) 以获取详细信息

命令	描述
<code>log set-level <Level></code>	设置主日志等级和所有 Handlers 日志等级
<code>log primary-level</code>	查看主日志等级
<code>log primary-level <Level></code>	设置主日志等级
<code>log handlers list</code>	查看当前安装的所有 Handlers
<code>log handlers start <HandlerId></code>	启动某个已停止的 Handler
<code>log handlers stop <HandlerId></code>	停止某个 Handler
<code>log handlers set-level <HandlerId> <Level></code>	设置指定 Handler 的日志等级

日志的等级由低到高分别为: `debug | info | notice | warning | error | critical | alert | emergency`，日志等级越低，系统输出的日志数量越多，消耗的系统资源越大。为提高系统运行性能，默认的主日志等级是 **error**。

log set-level <Level>

设置主日志等级和所有 **Handlers** 日志等级:

```
1 $ ./bin/emqx_ctl log set-level debug  
2 debug
```

sh

log primary-level

查看主日志等级:

```
1 $ ./bin/emqx_ctl log primary-level  
2 debug
```

sh

log primary-level <Level>

设置主日志等级:

```
1 $ ./bin/emqx_ctl log primary-level info  
2 info
```

sh

log handlers list

查看当前安装的所有 **Handlers**:

```
1 $ ./bin/emqx_ctl log handlers list  
2 LogHandler(id=ssl_handler, level=debug, destination=console, status=started)  
3 LogHandler(id=file, level=warning, destination=log/emqx.log, status=started)  
4 LogHandler(id=default, level=warning, destination=console, status=started)
```

sh

log handlers start <HandlerId>

启动 **log handler** `'default'` :

```
1 $ ./bin/emqx_ctl log handlers start default  
2 log handler default started
```

sh

log handlers stop <HandlerId>

停止 **log handler** `'default'` :

```
1 $ ./bin/emqx_ctl log handlers stop default  
2 log handler default stopped
```

sh

log handlers set-level <HandlerId> <Level>

设置指定 **Handler** 的日志等级:

```
1 $ ./bin/emqx_ctl log handlers set-level emqx_logger_handler error
2   error
```

sh

trace 命令

trace 命令用于追踪某个 **Client** 或 **Topic**, 打印日志信息到文件, 详情请查看 [日志与追踪](#)。

命令	描述
trace list	列出所有开启的追踪
trace start client <ClientId> <File> [<Level>]	开启 Client 追踪, 存储指定等级的日志到文件
trace stop client <ClientId>	关闭 Client 追踪
trace start topic <Topic> <File> [<Level>]	开启 Topic 追踪, 存储指定等级的日志到文件
trace stop topic <Topic>	关闭 Topic 追踪

trace start client <ClientId> <File> [<Level>]

开启 **Client** 追踪:

```
1 $ ./bin/emqx_ctl log primary-level debug
2   debug
3
4 $ ./bin/emqx_ctl trace start client clientid log/clientid_trace.log
5   trace clientid clientid successfully
6
7 $ ./bin/emqx_ctl trace start client clientid2 log/clientid2_trace.log error
8   trace clientid clientid2 successfully
```

sh

trace stop client <ClientId>

关闭 **Client** 追踪:

```
1 $ ./bin/emqx_ctl trace stop client clientid
2   stop tracing clientid clientid successfully
```

sh

trace start topic <Topic> <File> [<Level>]

开启 **Topic** 追踪:

```

1 $ ./bin/emqx_ctl log primary-level debug
2 debug
3
4 $ ./bin/emqx_ctl trace start topic topic log/topic_trace.log
5 trace topic topic successfully
6
7 $ ./bin/emqx_ctl trace start topic topic2 log/topic2_trace.log error
8 trace topic topic2 successfully

```

sh

trace stop topic <Topic>

关闭 Topic 追踪:

```

1 $ ./bin/emqx_ctl trace topic topic off
2 stop tracing topic topic successfully

```

sh

trace list

列出所有开启的追踪:

```

1 $ ./bin/emqx_ctl trace list
2 Trace (clientid=clientid2, level=error, destination="log/clientid2_trace.log")
3 Trace (topic=topic2, level=error, destination="log/topic2_trace.log")

```

sh

listeners

listeners 命令用于查询开启的 **TCP** 服务监听器。

命令	描述
listeners	# List listeners
listeners stop <Identifier>	# Stop a listener
listeners stop <Proto> <Port>	# Stop a listener
listeners restart <Identifier>	# Restart a listener

listeners list

查询开启的 **TCP** 服务监听器:

```

1 $ ./bin/emqx_ctl listeners
2 mqtt:ssl:external
3   listen_on      : 0.0.0.0:8883
4   acceptors     : 16
5   max_conn      : 102400
6   current_conn   : 0
7   shutdown_count : []
8 mqtt:tcp:external
9   listen_on      : 0.0.0.0:1883
10  acceptors     : 8
11  max_conn      : 1024000
12  current_conn   : 0
13  shutdown_count : []
14 mqtt:tcp:internal
15  listen_on      : 127.0.0.1:11883
16  acceptors     : 4
17  max_conn      : 1024000
18  current_conn   : 0
19  shutdown_count : []
20 http:dashboard
21  listen_on      : 0.0.0.0:18083
22  acceptors     : 4
23  max_conn      : 512
24  current_conn   : 0
25  shutdown_count : []
26 http:management
27  listen_on      : 0.0.0.0:8081
28  acceptors     : 2
29  max_conn      : 512
30  current_conn   : 0
31  shutdown_count : []
32 mqtt:ws:external
33  listen_on      : 0.0.0.0:8083
34  acceptors     : 4
35  max_conn      : 102400
36  current_conn   : 0
37  shutdown_count : []
38 mqtt:wss:external
39  listen_on      : 0.0.0.0:8084
40  acceptors     : 4
41  max_conn      : 16
42  current_conn   : 0
43  shutdown_count : []

```

listener 参数说明:

Name	描述
acceptors	TCP Acceptor 池
max_conn	最大允许连接数
current_conn	当前连接数
shutdown_count	连接关闭原因统计

listeners stop

停止监听端口:

```
1 $ ./bin/emqx_ctl listeners stop mqtt:tcp:external
2 Stop mqtt:tcp:external listener on 0.0.0.0:1883 successfully.
```

sh

listeners restart

重启监听器:

```
1 $ ./bin/emqx_ctl listeners restart http:dashboard
2 Stop http:dashboard listener on 0.0.0.0:18083 successfully.
3 Start http:dashboard listener on 0.0.0.0:18083 successfully.
4
```

sh

recon 命令

EMQX 的 **recon** 命令基于 **Erlang Recon** 库实现，用于帮助 **DevOps** 人员诊断生产节点中的问题，普通用户无需关心。使用 **recon** 命令会耗费一定的性能，请谨慎使用。

命令	描述
recon memory	recon_alloc:memory/2
recon allocated	recon_alloc:memory (allocated_types, current/max)
recon bin_leak	recon:bin_leak (100)
recon node_stats	recon:node_stats_print(10, 1000)
recon remote_load Mod	recon:remote_load (Mod)
recon proc_count Attr N	recon:proc_count(Attr, N)

访问 [Documentation for recon](#) 以获取详细信息。

retainer 命令

命令	描述
retainer info	显示保留消息的数量
retainer topics	显示当前存储的保留消息的所有主题
retainer clean	清除所有保留的消息
retainer clean <Topic>	清除指定的主题下的保留的消息

retainer info

显示保留消息的数量:

```
1 $ ./bin/emqx_ctl retainer info
2 retained/total: 3
```

sh

retainer topics

显示当前存储的保留消息的所有主题:

```
1 $ ./bin/emqx_ctl retainer topics
2 $SYS/brokers/emqx@127.0.0.1/version
3 $SYS/brokers/emqx@127.0.0.1/sysdescr
4 $SYS/brokers
```

sh

retainer clean

清除所有保留的消息:

```
1 $ ./bin/emqx_ctl retainer clean
2 Cleaned 3 retained messages
```

sh

retainer clean <Topic>

清除指定的主题下的保留的消息:

```
1 $ ./bin/emqx_ctl retainer clean topic
2 Cleaned 1 retained messages
```

sh

admins 命令

用于创建、删除管理员账号，重置管理员密码。

命令	描述
admins add <Username> <Password> <Tags>	创建 admin 账号
admins passwd <Username> <Password>	重置 admin 密码
admins del <Username>	删除 admin 账号

admins add <Username> <Password> <Tags>

创建 **admin** 账户:

```
1 $ ./bin/emqx_ctl admins add root public test
2 ok
```

sh

admins passwd <Username> <Password>

重置 **admin** 账户密码:

```
1 $ ./bin/emqx_ctl admins passwd root private
2 ok
```

sh

admins del <Username>

删除 **admin** 账户:

```
1 $ ./bin/emqx_ctl admins del root
2 ok
```

sh

规则引擎(rule engine) 命令**rules 命令**

命令	描述
rules list	List all rules
rules show <RuleId>	Show a rule
rules create <sql> <actions> [-d <descr>]	Create a rule
rules delete <RuleId>	Delete a rule

rules create

创建一个新的规则。参数:

- `<sql>` : 规则 SQL
- `<actions>` : JSON 格式动作列表
- `-d <descr>` : 可选, 规则描述信息

使用举例:

```
1 ## 创建一个测试规则, 简单打印所有发送到 't/a' 主题的消息内容
2 $ ./bin/emqx_ctl rules create \
3   'select * from "t/a"' \
4   '[{"name": "inspect", "params": {"a": 1}}]' \
5   -d 'Rule for debug'
6
7 Rule rule:9a6a725d created
```

sh

上例创建了一个 ID 为 `rule:9a6a725d` 的规则, 动作列表里只有一个动作: 动作名为 **inspect**, 动作的参数是 `{"a": 1}`。

rules list

列出当前所有的规则:

```
1 $ ./bin/emqx_ctl rules list
2
3 rule(id='rule:9a6a725d', for='[{"t/a"}]', rawsql='select * from "t/a"', actions=[{"metrics":...,
4 ..,"name":"inspect","params":...}], metrics=..., enabled='true', description='Rule for debug')
```

rules show

查询规则:

```
1 ## 查询 RuleID 为 'rule:9a6a725d' 的规则
2 $ ./bin/emqx_ctl rules show 'rule:9a6a725d'
3
4 rule(id='rule:9a6a725d', for='[{"t/a"}]', rawsql='select * from "t/a"', actions=[{"metrics":...,"name":"inspect","params":...}], metrics=..., enabled='true', description='Rule for debug')
```

rules delete

删除规则:

```
1 ## 删除 RuleID 为 'rule:9a6a725d' 的规则
2 $ ./bin/emqx_ctl rules delete 'rule:9a6a725d'
3
4 ok
```

rule-actions 命令

命令	描述
rule-actions list	List actions
rule-actions show <ActionId>	Show a rule action

提示

动作可以由 **EMQX** 内置(称为系统内置动作), 或者由 **EMQX** 插件编写, 但不能通过 **CLI/API** 添加或删除。

rule-actions show

查询动作:

```

1     ## 查询名为 'inspect' 的动作
2     $ ./bin/emqx_ctl rule-actions show 'inspect'
3
4     action(name='inspect', app='emqx_rule_engine', types=[], title ='Inspect (debug)', description='Inspect the details of action params for debug purpose')

```

rule-actions list

列出符合条件的动作:

```

1     ## 列出当前所有的动作
2     $ ./bin/emqx_ctl rule-actions list
3
4     action(name='data_to_rabbit', app='emqx_bridge_rabbit', types=[bridge_rabbit], title ='Data
5     bridge to RabbitMQ', description='Store Data to Kafka')
6     action(name='data_to_timestabledb', app='emqx_backend_pgsql', types=[timestabledb], title ='Da
7     ta to TimescaleDB', description='Store data to TimescaleDB')
8     ...

```

resources 命令

命令	描述
<code>resources create <type> [-c [<config>]] [-d [<descr>]]</code>	Create a resource
<code>resources list [-t <ResourceType>]</code>	List resources
<code>resources show <ResourceId></code>	Show a resource
<code>resources delete <ResourceId></code>	Delete a resource

resources create

创建一个新的资源, 参数:

- **type:** 资源类型
- **-c config:** JSON 格式的配置
- **-d descr:** 可选, 资源的描述

```

1     $ ./bin/emqx_ctl resources create 'web_hook' -c '{"url": "http://host-name/chats"}' -d 'forw
2     ard msgs to host-name/chats'
3
4     Resource resource:a7a38187 created

```

resources list

列出当前所有的资源:

```

1 $ ./bin/emqx_ctl resources list
2
3 resource(id='resource:a7a38187', type='web_hook', config=#{<<"url">> => <<"http://host-name/chats">>}, status=#{is_alive => false}, description='forward msgs to host-name/chats')

```

列出当前某个类型的资源:

```

1 $ ./bin/emqx_ctl resources list --type='web_hook'
2
3 resource(id='resource:a7a38187', type='web_hook', config=#{<<"url">> => <<"http://host-name/chats">>}, status=#{is_alive => false}, description='forward msgs to host-name/chats')

```

resources show

查询资源:

```

1 $ ./bin/emqx_ctl resources show 'resource:a7a38187'
2
3 resource(id='resource:a7a38187', type='web_hook', config=#{<<"url">> => <<"http://host-name/chats">>}, status=#{is_alive => false}, description='forward msgs to host-name/chats')

```

resources delete

删除资源:

```

1 $ ./bin/emqx_ctl resources delete 'resource:a7a38187'
2
3 ok

```

resource-types 命令

命令	描述
resource-types list	List all resource-types
resource-types show <Type>	Show a resource-type

提示

资源类型可以由 **EMQX** 内置(称为系统内置资源类型), 或者由 **EMQX** 插件编写, 但不能通过 **CLI/API** 添加或删除。

resource-types list

列出当前所有的资源类型:

```

1 ./bin/emqx_ctl resource-types list
2
3 resource_type(name='backend_mongo_rs', provider='emqx_backend_mongo', title ='MongoDB Replic
4 a Set Mode', description='MongoDB Replica Set Mode')
5 resource_type(name='backend_cassa', provider='emqx_backend_cassa', title ='Cassandra', descr
6 iption='Cassandra Database')
...

```

resource-types show

查询资源类型:

```

1 $ ./bin/emqx_ctl resource-types show backend_mysql
2
3 resource_type(name='backend_mysql', provider='emqx_backend_mysql', title ='MySQL', descrip
4 tion='MySQL Database')

```

与规则引擎相关的状态、统计指标和告警

规则状态和统计指标

度量指标				
节点	已命中	命中速度	最大命中速度	5分钟平均速度
emqx@127.0.0.1	0	0	0	0

- **已命中:** 规则命中(规则 **SQL** 匹配成功)的次数,
- **命中速度:** 规则命中的速度(**次/秒**)
- **最大命中速度:** 规则命中速度的峰值(**次/秒**)
- **5分钟平均速度:** 5分钟内规则的平均命中速度(**次/秒**)

动作状态和统计指标

响应动作		
类型	参数	度量指标
inspect		emqx@127.0.0.1 成功: 0 失败: 0 合计 成功: 0 失败: 0

- **成功:** 动作执行成功次数
- **失败:** 动作执行失败次数

资源状态和告警

ID	资源类型	备注	操作
resource:813cb5d7	backend_mysql		<button>查看</button> <button>删除</button>
emqx@127.0.0.1 ● 可用			

- 可用: 资源可用
- 不可用: 资源不可用(比如数据库连接断开)

EMQX 内置数据库 Auth 与 ACL 规则

此命令只有在开启 `emqx_auth_mnesia` 插件后生效

clientid 命令

命令	描述
<code>clientid list</code>	List clientid auth rules
<code>clientid add <ClientID> <Password></code>	Add clientid auth rule
<code>clientid update <ClientID> <Password></code>	Update clientid auth rule
<code>clientid del <ClientID> <Password></code>	Delete clientid auth rule

clientid list

列出所有的 `clientid` 验证规则

```

1 ./bin/emqx_ctl clientid list
2 emqx

```

clientid add <ClientID> <Password>

增加 `clientid` 验证规则

```

1 ./bin/emqx_ctl clientid add emqx public
2 ok

```

clientid update <ClientID> <Password>

更新 ClientID 验证的密码

```
1 ./bin/emqx_ctl clientid update emqx new_password
2 ok
```

sh

clientid del <ClientID> <Password>删除 **clientid** 验证规则码

```
1 ./bin/emqx_ctl clientid del emqx new_password
2 ok
```

sh

user 命令

命令	描述
user list	List username auth rules
user add <Username> <Password>	Add username auth rule
user update <Username> <Password>	Update username auth rule
user del <Username> <Password>	Delete username auth rule

user list列出所有的 **username** 验证规则

```
1 ./bin/emqx_ctl user list
2 emqx
```

sh

user add <Username> <Password>增加 **username** 验证规则

```
1 ./bin/emqx_ctl user add emqx public
2 ok
```

sh

user update <Username> <Password>更新 **ClientID** 验证的密码

```
1 ./bin/emqx_ctl user update emqx new_password
2 ok
```

sh

user del <Username> <Password>删除 **username** 验证规则码

```
1 ./bin/emqx_ctl user del emqx new_password
2 ok
```

sh

acl 命令

命令	描述
acl list clientid	List clientid acls
acl list username	List username acls
acl list _all	List \$all acls
acl show clientid <Clientid>	Lookup clientid acl detail
acl show username <Username>	Lookup username acl detail
acl add clientid <Clientid> <Topic> <Action> <Access>	Add clientid acl
acl add Username <Username> <Topic> <Action> <Access>	Add username acl
acl add _all <Topic> <Action> <Access>	Add \$all acl
acl del clientid <Clientid> <Topic>	Delete clientid acl
acl del username <Username> <Topic>	Delete username acl
acl del _all \<Topic>	Delete \$all acl

acl list

- **acl list clientid**

列出 **clientid** 的 ACL 规则

```
1 ./bin/emqx_ctl acl list clientid
2 Acl(clientid = <<"emqx_clientid">> topic = <<"Topic/A">> action = pub access = allow)
```

sh

- **acl list username**

列出 **username** 的 ACL 规则

```
1 ./bin/emqx_ctl acl list username
2 Acl(username = <<"emqx_username">> topic = <<"Topic/B">> action = sub access = deny)
```

sh

- **acl list _all**

列出 **\$all** 的 ACL 规则

```
1 ./bin/emqx_ctl acl list _all
2 Acl($all topic = <<"Topic/C">> action = pubsub access = allow)
```

sh

acl show

- **acl show clientid <Clientid>**

展示某一的 **clientid ACL** 规则

```
1 ./bin/emqx_ctl acl show clientid emqx_clientid          sh
2 Acl(clientid = <<"emqx_clientid">> topic = <<"Topic/A">> action = pub access = allow)
```

- **acl show username <Username>**

展示某一的 **username ACL** 规则

```
1 ./bin/emqx_ctl acl show username emqx_username          sh
2 Acl(username = <<"emqx_username">> topic = <<"Topic/B">> action = sub access = deny)
```

acl add

- **acl add clientid <Clientid> <Topic> <Action> <Access>**

增加一条 **clientid** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl add clientid emqx_clientid Topic/A pub allow          sh
2 ok
```

- **acl add username <Username> <Topic> <Action> <Access>**

增加一条 **username** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl add username emqx_username Topic/B sub deny          sh
2 ok
```

- **acl add _all <Topic> <Action> <Access>**

增加一条 **\$all** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl add _all Topic/C pubsub allow          sh
2 ok
```

acl del

- **acl del clientid <Clientid> <Topic>**

删除一条 **clientid** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl del clientid emqx_clientid Topic/A          sh
2 ok
```

- **acl del username <Username> <Topic>**

删除一条 **username** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl del clientid emqx_username Topic/B  
2 ok
```

sh

- **acl del _all <Topic>**

删除一条 **\$all** 的 **ACL** 规则

```
1 ./bin/emqx_ctl acl del _all Topic/C  
2 ok
```

sh

pem_cache 命令

pem_cache 命令用于清理所有 **PEM** 证书的缓存。例如，在替换了 **SSL** 监听器的证书文件后，你可以使用该命令让新的证书文件生效，而不是重启 **EMQX** 的 **SSL** 监听器。

命令	描述
pem_cache clean	清理所有节点的 x509 的证书缓存
pem_cache clean node <Node>	清理指定节点的 x509 的证书缓存

注：更新证书并不会影响当前已建立的 **TLS** 连接。

提示

该命令在 **v4.3.13** 版本中引入。在这之前都版本都不支持该命令

版本热升级

自 **4.2.0** 版本之后，**EMQX 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 **EMQX Broker**，并避免了因重启服务导致的系统可用性降低。

目前 **EMQX 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 **Windows**、**MacOSX** 暂不支持热升级功能。

热升级步骤

1. 查看当前已安装的 **EMQX Broker** 的版本列表。

```
1 $ emqx versions  
2  
3     Installed versions:  
4 * 4.2.0 permanent
```

2. 从 **EMQX** 官网下载要升级的软件包。

访问 <https://www.emqx.com/en/downloads?product=broker> 选择对应的版本和操作系统类型，然后选择 "**zip**" 包类型。

3. 找到 **EMQX** 的安装目录：

```
1 $ EMQX_ROOT_DIR=$(emqx root_dir)  
2  
3 $ echo ${EMQX_ROOT_DIR}  
4 "/usr/lib/emqx"
```

4. 将下载的 **zip** 包放到 **EMQX** 安装目录下的 `releases` 目录下：

```
1 $ cp emqx-4.2.1.zip ${EMQX_ROOT_DIR}/releases/
```

5. 升级到指定版本：

```

1 $ emqx upgrade 4.2.1
2
3 Release 4.2.1 not found, attempting to unpack releases/emqx-4.2.1.tar.gz
4 Unpacked successfully: "4.2.1"
5 Installed Release: 4.2.1
6 Made release permanent: "4.2.1"
7

```

6. 再次查看版本列表，之前的版本的状态将会变成 `old`：

```

1
2 $ emqx versions
3
4 Installed versions:
5 * 4.2.1 permanent
6 * 4.2.0 old
7

```

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：

```

1
2 $ emqx upgrade --no-permanent 4.2.1
3
4 Release 4.2.1 not found, attempting to unpack releases/emqx-4.2.1.tar.gz
5 Unpacked successfully: "4.2.1"
6 Installed Release: 4.2.1
7

```

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：

```

1
2 $ emqx versions
3
4 Installed versions:
5 * 4.2.1 current
6 * 4.2.0 permanent
7

```

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：

```
1 $ emqx install 4.2.1  
2  
3 Release 4.2.1 is already installed and current, making permanent.  
4 Made release permanent: "4.2.1"  
5  
6
```

sh

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：

```
1  
2 $ emqx downgrade 4.2.0  
3  
4 Release 4.2.0 is marked old, switching to it.  
5 Installed Release: 4.2.0  
6 Made release permanent: "4.2.0"  
7
```

sh

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**：

```
1  
2 $ emqx uninstall 4.2.0  
3  
4 Release 4.2.0 is marked old, uninstalling it.  
5 Uninstalled Release: 4.2.0  
6
```

sh

在运行时安装 EMQX 补丁包

如果一个 **Bug** 修复只更新了少数几个 **module**, 在已经知道需要更新的 **module** 列表的情况下, 可以使用补丁包方式升级 **EMQX**。

注意: 如果可以使用版本热升级的方式, 则首选版本热升级。只有版本热升级不可用, 并且你了解在生产环境中 安装补丁包的后果的情况下, 才可以使用此方式解决问题。

安装 EMQX 补丁包的步骤

- 从 **EMQX** 开发者那里获取本次需要更新的 **modules** 列表。比如:

```
1 emqx.beam
2 emqx_rule_engine.beam
```

- 从 **EMQX** 官网, 或者 **EMQX** 开发者那里获取本次更新对应的软件包。

访问 [开源版下载地址](#) 或者 [企业版下载地址](#), 下载对应版本的 **zip** 软件包。

注意选择正确的软件版本号、**OTP** 版本号、以及操作系统类型, 并且选择 "**zip**" 包类型。

- 解压下载的 **zip** 包, 并找到要更新的 **modules**:

```
1 $ unzip -q emqx-ee-4.4.1-otp24.1.5-3-ubuntu20.04-amd64.zip
```

sh

假设我们要更新 **emqx.beam** 和 **emqx_rule_engine.beam**, 则在解压目录下寻找他们:

```
1 $ find ./emqx -name "emqx.beam"
2 ./emqx/lib/emqx-4.4.1/ebin/emqx.beam
3
4 $ find ./emqx -name "emqx_rule_engine.beam"
5 ./emqx/lib/emqx_rule_engine-4.4.1/ebin/emqx_rule_engine.beam
```

sh

- 确保 **EMQX** 是已经启动的状态:

```
1 $ emqx_ctl status
2 Node 'emqx@127.0.0.1' 4.4.1 is started
```

sh

- 找到 **beam** 文件的对应位置, 备份并替换原 **beam** 文件:

找到 **EMQX** 的安装目录:

```
1 $ emqx root_dir
2 "/usr/lib/emqx"
```

sh

在安装目录下的 **lib** 目录下查找原 **beam** 文件的路径:

```

1 $ find /usr/lib/emqx/lib -name "emqx.beam"
2 /usr/lib/emqx/lib/emqx-4.4.0/ebin/emqx.beam
3
4 $ find /usr/lib/emqx/lib -name "emqx_rule_engine.beam"
5 /usr/lib/emqx/lib/emqx_rule_engine-4.4.0/ebin/emqx_rule_engine.beam

```

sh

备份原 **beam** 文件到 `/tmp` 目录:

```

1 $ cp /usr/lib/emqx/lib/emqx-4.4.0/ebin/emqx.beam \
2      /usr/lib/emqx/lib/emqx_rule_engine-4.4.0/ebin/emqx_rule_engine.beam /tmp

```

sh

使用新 **beam** 文件覆盖到对应位置:

```

1 $ cp -f ./emqx/lib/emqx-4.4.1/ebin/emqx.beam /usr/lib/emqx/lib/emqx-4.4.0/ebin/
2 $ cp -f ./emqx/lib/emqx_rule_engine-4.4.1/ebin/emqx_rule_engine.beam /usr/lib/emqx/lib/emqx_r
ule_engine-4.4.0/ebin/

```

sh

6. 加载新的 **beam** 文件:

```

1 $ emqx eval 'c:lm().''
2 [{module, emqx},
3 {module, emqx_rule_engine}]

```

sh

回滚补丁包

1. 把安装补丁包时备份好的文件复制回原来的目录:

```

1 $ cp -f /tmp/emqx.beam /usr/lib/emqx/lib/emqx-4.4.0/ebin/
2 $ cp -f /tmp/emqx_rule_engine.beam /usr/lib/emqx/lib/emqx_rule_engine-4.4.0/ebin/

```

sh

2. 重新加载 **beam** 文件:

```

1 $ emqx eval 'c:lm().''
2 [{module, emqx},
3 {module, emqx_rule_engine}]

```

sh

入门概念

EMQX 是什么？

EMQX 是开源百万级分布式 **MQTT** 消息服务器（**MQTT Messaging Broker**），用于支持各种接入标准 **MQTT** 协议的设备，实现从设备端到服务器端的消息传递，以及从服务器端到设备端的设备控制消息转发。从而实现物联网设备的数据采集，和对设备的操作和控制。

为什么选择EMQX？

EMQX 与别的 **MQTT** 服务器相比，有以下的优点：

- 经过100+版本的迭代，**EMQX** 目前为开源社区中最流行的 **MQTT** 消息中间件，在各种客户严格的生产环境上经受了严苛的考验；
- **EMQX** 支持丰富的物联网协议，包括 **MQTT**、**MQTT-SN**、**CoAP**、**LwM2M**、**LoRaWAN** 和 **WebSocket** 等；
- 优化的架构设计，支持超大规模的设备连接。企业版单机能支持百万的 **MQTT** 连接；集群能支持千万级别的 **MQTT** 连接；
- 易于安装和使用；
- 灵活的扩展性，支持企业的一些定制场景；
- 中国本地的技术支持服务，通过微信、QQ等线上渠道快速响应客户需求；

EMQX 的主题数量有限制吗？

主题使用没有数量限制，主题数量增长对性能影响不大，可以放心使用。

EMQX 开源版怎么存储数据？

开源版不支持数据存储功能，可以使用企业版，或者使用外部程序订阅主题/**Webhook** 的方式获取数据，然后写入到数据库。

EMQX 与物联网平台的关系是什么？

典型的物联网平台包括设备硬件、数据采集、数据存储、分析、**Web / 移动应用**等。**EMQX** 位于数据采集这一层，分别与硬件和数据存储、分析进行交互，是物联网平台的核心：前端的硬件通过 **MQTT** 协议与位于数据采集层的 **EMQX** 交互，通过 **EMQX** 将数据采集后，通过 **EMQX** 提供的数据接口，将数据保存到后台的持久化平台中（各种关系型数据库和 **NOSQL** 数据库），或者流式数据处理框架等，上层应用通过这些数据分析后得到的结果呈现给最终用户。

EMQX 有哪些产品？

标签: [企业版](#)

EMQX 公司主要提供[三个产品](#)，主要体现在支持的连接数量、产品功能和商业服务等方面的区别：

- **EMQX Broker**: **EMQX** 开源版，提供 **MQTT** 协议、**CoAP** 和 **LwM2M** 等常见物联网协议的支持；

- **EMQX Enterprise**: **EMQX** 企业版，在开源版基础上，增加了数据持久化 **Redis**、**MySQL**、**MongoDB** 或 **PostgreSQL**，数据桥接转发 **Kafka**、**LoRaWAN** 支持，监控管理，**Kubernetes** 部署等方面的支持；支持百万级并发连接；
- **EMQX Cloud**: **EMQX Cloud**[↗](#) 是 **EMQ** 公司推出的一款面向物联网领域的 **MQTT** 消息中间件产品。作为全球首个全托管的 **MQTT 5.0** 公有云服务，**EMQX Cloud** 提供了一站式运维代管、独有隔离环境的 **MQTT** 消息服务。在万物互联的时代，**EMQX Cloud** 可以帮助您快速构建面向物联网领域的行业应用，轻松实现物联网数据的采集、传输、计算和持久化。

EMQX 与 NB-IoT、LoRAWAN 的关系是什么？

标签: [NB-IoT](#) [LoRAWAN](#) [多协议](#)

EMQX 是一个开源的 **MQTT** 消息服务器，并且 **MQTT** 是一个 **TCP** 协议栈上位于应用层的协议；而 **NB-IoT** 和 **LoRAWAN** 在 **TCP** 协议层处于物理层，负责物理信号的传输。因此两者在 **TCP** 协议栈的不同层次上，实现不同的功能。

MQTT 协议与 HTTP 协议相比，有何优点和弱点？

标签: [多协议](#)

HTTP 协议是一个无状态的协议，每个 **HTTP** 请求为 **TCP** 短连接，每次请求都需要重新创建一个 **TCP** 连接（可以通过 **keep-alive** 属性来优化 **TCP** 连接的使用，多个 **HTTP** 请求可以共享该 **TCP** 连接）；而 **MQTT** 协议为长连接协议，每个客户端都会保持一个长连接。与 **HTTP** 协议相比优势在于：

- **MQTT** 的长连接可以用于实现从设备端到服务器端的消息传送之外，还可以实现从服务器端到设备端的实时控制消息发送，而 **HTTP** 协议要实现此功能只能通过轮询的方式，效率相对来说比较低；
- **MQTT** 协议在维护连接的时候会发送心跳包，因此协议以最小代价内置支持设备“探活”的功能，而 **HTTP** 协议要实现此功能的话需要单独发出 **HTTP** 请求，实现的代价会更高；
- 低带宽、低功耗。**MQTT** 在传输报文的大小上与 **HTTP** 相比有巨大的优势，因为 **MQTT** 协议在连接建立之后，由于避免了建立连接所需要的额外的资源消耗，发送实际数据的时候报文传输所需带宽与 **HTTP** 相比有很大的优势，参考网上[有人做的测评](#)[↗](#)，发送一样大小的数据，**MQTT** 比 **HTTP** 少近 50 倍的网络传输数据，而且速度快了将近 20 倍。在网上有人做的[另外一个评测显示](#)[↗](#)，接收消息的场景，**MQTT** 协议的耗电量为 **HTTP** 协议的百分之一，而发送数据的时候 **MQTT** 协议的耗电量为 **HTTP** 协议的十分之一；
- **MQTT** 提供消息质量控制（**QoS**），消息质量等级越高，消息交付的质量就越有保障，在物联网的应用场景下，用户可以根据不同的应用场景来设定不同的消息质量等级；

什么是认证鉴权？使用场景是什么？

标签: [认证鉴权](#)

认证鉴权指的是当一个客户端连接到 **MQTT** 服务器的时候，通过服务器端的配置来控制客户端连接服务器的权限。**EMQ** 的认证机制包含了三种，

- 用户名密码：针对每个 **MQTT** 客户端的连接，可以在服务器端进行配置，用于设定用户名和密码，只有在用户名和密码匹配的情况下才可以让客户端进行连接
- **ClientID**：每个 **MQTT** 客户端在连接到服务器的时候都会有个唯一的 **ClientID**，可以在服务器中配置可以连接该服

务器的 **ClientID** 列表，这些 **ClientID** 的列表里的客户端可以连接该服务器

- 匿名：允许匿名访问

通过用户名密码、**ClientID** 认证的方式除了通过配置文件之外，还可以通过各类数据库和外部应用来配置，比如 **MySQL**、**PostgreSQL**、**Redis**、**MongoDB**、**HTTP** 和 **LDAP** 等。

什么是 Hook？使用场景是什么？

标签: [WebHook](#)

钩子（hook）指的是由 **EMQX** 在连接、对话和消息触发某些事件的时候提供给对外部的接口，主要提供了如下的钩子，**EMQX** 提供了将这些 hook 产生的事件持久化至数据库的功能，从而很方便地查询得知客户端的连接、断开等各种信息。

- **client.connected**: 客户端上线
- **client.disconnected**: 客户端连接断开
- **client.subscribe**: 客户端订阅主题
- **client.unsubscribe**: 客户端取消订阅主题
- **session.created**: 会话创建
- **session.resumed**: 会话恢复
- **session.subscribed**: 会话订阅主题后
- **session.unsubscribed**: 会话取消订阅主题后
- **session.terminated**: 会话终止
- **message.publish**: MQTT 消息发布
- **message.delivered**: MQTT 消息送达
- **message.acked**: MQTT 消息回执
- **message.dropped**: MQTT 消息丢弃

什么是 mqueue？如何配置 mqueue？

标签: [消息队列](#)

mqueue 是 **EMQX** 在消息发布流程中保存在会话中的一个消息队列，当 **MQTT** 连接报文中的 **clean session** 设置为 **false** 的时候，即使是客户端断开连接的情况下，**EMQX** 也会为断连客户端保存一个会话，这个会话会缓存订阅关系，并代替断开连接的客户端去接收订阅主题上的消息，而这些消息会存在 **EMQX** 的 **mqueue** 中，等到客户端重新上线再将消息重新发给客户端。由于 **qos0** 消息在 **MQTT** 协议中的优先级比较低，所以 **EMQX** 默认不缓存 **qos 0** 的消息，**mqueue** 在 **EMQX** 中是可以配置的，通过配置 `zone.$name.mqueue_store_qos0 = true` 可以将 **qos0** 消息也存在 **mqueue** 中，**mqueue** 的大小也是有限制的，通过配置项 `zone.external.max_mqueue_len`，可以确定每个会话缓存的消息数量。注意，这些消息是存储在内存中的，所以尽量不要将 **mqueue** 长度限制修改为 **0**（设置为 **0** 代表 **mqueue** 长度没有限制），否则在实际的业务场景中，有内存耗光的风险。

什么是 WebSocket？什么情况下需要通过 WebSocket 去连接 EMQX 服务器？

标签: [WebSocket 多协议](#)

WebSocket 是一种在基于 **HTTP** 协议上支持全双工通讯的协议，通过该协议，用户可以实现浏览器和服务器之间的双向通信，比如可以通过服务器往浏览器端推送消息。**EMQX** 提供了 **WebSocket** 连接支持，用户可以在浏览器端直接

实现对主题的订阅和消息发送等操作。

什么是共享订阅？有何使用场景？

标签: [共享订阅](#)

共享订阅是 **MQTT 5.0** 标准的新特性，在标准发布前，**EMQX** 就已经把共享订阅作为标准外特性进行了支持。在普通订阅中，所有订阅者都会收到订阅主题的所有消息，而在共享订阅中，订阅同一个主题的客户端会轮流的收到这个主题下的消息，也就是说同一个消息不会发送到多个订阅者，从而实现订阅端的多个节点之间的负载均衡。

共享订阅对于数据采集 / 集中处理类应用非常有用。在这样的场景下，数据的生产者远多余数据的消费者，且同一条数据只需要被任意消费者处理一次。

更多使用方式请参考 [共享订阅](#)。

什么是离线消息？

标签: [离线消息](#)

一般情况下 **MQTT** 客户端仅在连接到消息服务器的时候，如果客户端离线将收不到消息。但是在客户端有固定的 **ClientID**, **clean_session** 为 **false**, 且 **QoS** 设置满足服务器端的配置要求时，在客户端离线时，服务器可以为客户端保持一定量的离线消息，并在客户端再次连接时发送给客户端。

离线消息在网络连接不是很稳定时，或者对 **QoS** 有一定要求时非常有用。

什么是代理订阅？使用场景是什么？

标签: [代理订阅](#)

通常情况下客户端需要在连接到 **EMQX** 之后主动订阅主题。代理订阅是指服务器为客户端订阅主题，这一过程不需要客户端参与，客户端和需要代理订阅的主题的对应关系保存在服务器中。

使用代理订阅可以集中管理大量的客户端的订阅，同时为客户端省略掉订阅这个步骤，可以节省客户端侧的计算资源和网络带宽。

以 **Redis** 数据库为例，代理订阅在 **EMQX** 上使用方式请参考 [Redis 实现客户端代理订阅](#)

提示

注：目前 **EMQX** 企业版支持代理订阅。

系统主题有何用处？都有哪些系统主题？

标签: [系统主题](#)

系统主题以 `$SYS/` 开头。**EMQX** 会以系统主题的方式周期性的发布关于自身运行状态、**MQTT** 协议统计、客户端上下线状态到系统主题。订阅系统主题可以获得这些信息。

这里列举一些系统主题，完整的系统主题请参考 **EMQX** 文档的相关章节：

- **\$SYS/brokers:** 集群节点列表

- **\$SYS/brokers/\${node}/clients/\${clientId}/connected:** 当客户端连接时发送的客户端信息
- **\$SYS/broker/\${node}/stats/connections/count:** 当前客户端总数
- **\$SYS/broker/\${node}/stats/sessions/count:** 当前会话总数

为什么开启认证后，客户端还是可以不需要用户名密码就能连接？

标签: [认证鉴权](#)

EMQX 支持匿名认证，并默认启用。开启认证后，未指定用户名密码的客户端将以匿名认证的方式成功登录。想要更改这一行为，需要修改 `emqx.conf` 文件中的 `allow_anonymous` 配置项，将其设为 `false`，然后重启 **EMQX** 即可。

注：如果您的客户端连接的是 **11883** 端口，则需要修改 `zone.internal.allow_anonymous`，**Zone** 与 **Listener** 的相关知识可参阅 [配置说明](#)。

使用教程

怎么样才能使用 EMQX?

EMQX 开源版可免费下载使用，下载地址：<https://www.emqx.com/en/downloads?product=broker>

EMQX 企业版支持下载试用，用户可以在 <https://www.emqx.com/en/downloads?product=enterprise> 下载，[申请试用 license](#) 之后即可试用。

另外，还可以在公有云直接创建 **EMQX** 企业版：

- [阿里云](#)
- [青云](#)

怎样更新 EMQX license?

标签: [License](#)

点击 "Download License" 按钮下载 **license**，然后找到您下载的 "**license.zip**" 文件并解压。

复制压缩包里的两个文件 (**emqx.lic**, **emqx.key**) 到 **EMQX** 的 **license** 目录。

如果您的 **EMQX** 是使用 **zip** 包安装的，那么压缩包里的两个文件需要拷贝到 "**emqx/etc/**" 目录；如果是用 **DEB/RPM** 包安装的，两个文件需要拷贝到 "**/etc/emqx/**" 目录；如果是用 **Docker** 镜像安装的，两个文件需要拷贝到 "**/opt/emqx/etc/**" 目录。

拷贝完成后需要通过命令行重新加载 **license** 以完成更新：

基础命令：

```
1 emqx_ctl license reload [license 文件所在路径]
```

不同安装方式更新命令如下：

```
1 ## 适用于 zip 包
2 ./bin/emqx_ctl license reload etc/emqx.lic
3
4 ## DEB/RPM 包安装
5 emqx_ctl license reload /etc/emqx/emqx.lic
6
7 ## Docker 镜像安装
8 docker exec -it emqx-ee emqx_ctl license reload /opt/emqx/etc/emqx.lic
```

EMQX 支持私有协议进行扩展吗？如支持应该如何实现？

标签: [多协议 扩展](#)

对于新开发的私有协议，**EMQX** 提供一套 **TCP** 协议接入规范，私有协议可以按照该规范进行开发接入。如果您所使用

的协议已经定型或协议底层非 **TCP**，可以通过网关进行转换处理，之后通过 **MQTT** 协议接入 **EMQX**，或直接联系 **EMQ** 官方支持私有协议适配。

我可以捕获设备上下线的事件吗？该如何使用？

标签: [WebHook 系统主题](#)

EMQX 企业版可以通过以下的三种方式捕获设备的上下线的事件，

- **Web Hook**
- 订阅相关的 **\$SYS** 主题
 - **\$SYS/brokers/\${node}/clients/\${clientid}/connected**
 - **\$SYS/brokers/\${node}/clients/\${clientid}/disconnected**
- 直接保存到数据库

最后一种方法只有在企业版里才支持，支持的数据库包括 **Redis**、**MySQL**、**PostgreSQL**、**MongoDB** 和 **Cassandra**。用户可以通过配置文件指定所要保存的数据库，以及监听 **client.connected** 和 **client.disconnected** 事件，这样在设备上、下线的时候把数据保存到数据库中。

我想限定某些主题只为特定的客户端所使用，**EMQX** 该如何进行配置？

标签: [ACL 发布订阅](#)

EMQX 支持限定客户端可以使用的主题，从而实现设备权限的管理。如果要做这样的限定，需要在 **EMQX** 启用 **ACL (Access Control List)**，并禁用匿名访问和关闭无 **ACL** 命中的访问许可（为了测试调试方便，在默认配置中，后两项是开启的，请注意关闭）。

```
1 ## etc/emqx.conf
2
3 ## ACL nomatch
4 mqtt.acl_nomatch = allow
```

ACL 可以配置在文件 `etc/acl.conf` 中，或者配置在后台数据库中。下面例子是 **ACL** 控制文件的一个配置行，含义是用户 “**dashboard**” 可以订阅 “**\$SYS/#**” 主题。**ACL** 在后台数据库中的配置思想与此类似，详细配置方法请参阅 **EMQX** 文档的 [ACL 访问控制](#) 章节。

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

EMQX 能做流量控制吗？

标签: [流量控制](#)

能。目前 **EMQX** 支持连接速率和消息率控制。配置如下：

```

1 ## Value: Number
2 listener.tcp.external.max_conn_rate = 1000
3
4 ## Value: rate,burst
5 listener.tcp.external.rate_limit = 1024,4096

```

EMQX 是如何实现支持大规模并发和高可用的?

标签: [性能](#) [高并发](#)

高并发和高可用是 **EMQX** 的设计目标, 为了实现这些目标 **EMQX** 中应用了多种技术, 比如:

- 利用 **Erlang/OTP** 平台的软实时、高并发和容错;
- 全异步架构;
- 连接、会话、路由、集群的分层设计;
- 消息平面和控制平面的分离等。

在精心设计和实现之后, 单个 **EMQX Enterprise** 节点就可以处理百万级的连接。

EMQX 支持多节点集群, 集群下整个系统的性能会成倍高于单节点, 并能在单节点故障时保证系统服务不中断。

EMQX 能把接入的 MQTT 消息保存到数据库吗?

标签: [持久化](#)

EMQX 企业版支持消息持久化, 可以将消息保存到数据库, 开源版还暂时不支持。目前 **EMQX** 企业版消息持久化支持的数据库有:

- [Redis](#)
- [MongoDB](#)
- [MySQL](#)
- [PostgreSQL](#)
- [Cassandra](#)
- [AWS DynamoDB](#)
- [TimescaleDB](#)
- [OpenTSDB](#)
- [InfluxDB](#)

有关数据持久化的支持请参见 [EMQX 数据持久化概览](#)。

在服务器端能够直接断开一个 MQTT 连接吗?

标签: [HTTP API Dashboard](#)

可以的。**EMQX** 提供的 **HTTP API** 中包含断开 **MQTT** 连接, 该操作在 **EMQX 2.x** 和 **3.0** 的实现方式有所不同:

- 在 **2.x** 版本中是由 **EMQX** 自定义扩展协议实现的
- 在 **3.0** 版本之后按照 **MQTT 5.0** 协议对从服务器端断开连接的规范要求实现的

调用的 **API** 如下所示:

```

1   HTTP 方法: DELETE
2   URL: api/[v2|v3]/clients/{clientid}
3   <!--请注意区分 URL 中第二部分的版本号, 请根据使用的版本号来决定 -->
4
5   返回内容:
6   {
7       "code": 0,
8       "result": []
9   }

```

HTTP API 使用方式参考 [管理监控API \(HTTP API\)](#)

EMQX 能把接入的消息转发到 Kafka 吗?

标签: [Kafka 桥接 持久化](#)

能。目前 **EMQX** 企业版提供了内置的 **Kafka** 桥接方式, 支持把消息桥接至 **Kafka** 进行流式处理。

EMQX 使用 **Kafka** 参照 [EMQX 到 Kafka 的桥接](#)

EMQX 企业版中桥接 Kafka, 一条 MQTT 消息到达 EMQX 集群之后就回 MQTT Ack 报文还是写入 Kafka 之后才回 MQTT Ack 报文?

标签: [Kafka 配置](#)

取决于 **Kafka** 桥接的配置, 配置文件位于 `/etc/emqx/plugins/emqx_bridge_kafka.conf`

```

1   ## Pick a partition producer and sync/async.
2   bridge.kafka.produce = sync

```

sh

- 同步: **EMQX** 在收到 **Kafka** 返回的 **Ack** 之后才会给前端返回 **MQTT Ack** 报文
- 异步: **MQTT** 消息到达 **EMQX** 集群之后就回 **MQTT Ack** 报文, 而不会等待 **Kafka** 返回给 **EMQX** 的 **Ack**

如果运行期间, 后端的 **Kafka** 服务不可用, 则消息会被累积在 **EMQX** 服务器中,

- EMQX 2.4.3** 之前的版本会将未发送至 **Kafka** 的消息在内存中进行缓存, 直至内存使用完毕, 并且会导致 **EMQX** 服务不可用。
- EMQX 2.4.3** 版本开始会将未发送至 **Kafka** 的消息在磁盘中进行缓存, 如果磁盘用完可能会导致数据丢失。

因此建议做好 **Kafka** 服务的监控, 在发现 **Kafka** 服务有异常情况的时候尽快恢复 **Kafka** 服务。

EMQX 支持集群自动发现吗? 有哪些实现方式?

标签: [集群](#)

EMQX 支持集群自动发现。集群可以通过手动配置或自动配置的方式实现。

目前支持的自动发现方式有:

- 手动集群
- 静态集群

- **IP Multi-cast** 自动集群
- **DNS** 自动集群
- **ETCD** 自动集群
- **K8S** 自动集群

有关集群概念和组建集群方式请参照 [EMQX 的集群概念](#)

我可以把 MQTT 消息从 EMQX 转发其他消息中间件吗？例如 RabbitMQ？

标签: [RabbitMQ 桥接 持久化](#)

EMQX 支持转发消息到其他消息中间件，通过 **EMQX** 提供的桥接方式就可以做基于主题级别的配置，从而实现主题级别的消息转发。

EMQX 桥接相关的使用方式请参照 [EMQX 桥接](#)

我可以把消息从 EMQX 转到公有云 MQTT 服务上吗？比如 AWS 或者 Azure 的 IoT Hub？

标签: [桥接](#)

EMQX 可以转发消息到标准 **MQTT Broker**，包括其他 **MQTT** 实现、公有云的 **IoT Hub**，通过 **EMQX** 提供的桥接就可以实现。

MQTT Broker（比如 Mosquitto）可以转发消息到 EMQX 吗？

标签: [Mosquitto 桥接](#)

Mosquitto 可以配置转发消息到 **EMQX**，请参考[数据桥接](#)。

提示

EMQX 桥接相关的使用方式请参照 [EMQX 桥接](#)

我想跟踪特定消息的发布和订阅过程，应该如何做？

标签: [Trace 调试](#)

EMQX 支持追踪来自某个客户端的报文或者发布到某个主题的报文。追踪消息的发布和订阅需要使用命令行工具 (**emqx_ctl**) 的 **trace** 命令，下面给出一个追踪‘topic’主题的消息并保存在 `trace_topic.log` 中的例子。更详细的说明请参阅 **EMQX** 文档的相关章节。

```
1 ./bin/emqx_ctl trace topic "topic" "trace_topic.log"
```

为什么我做压力测试的时候，连接数目和吞吐量老是上不去，有系统调

优指南吗？

标签: [调试](#) [性能测试](#)

在做压力测试的时候，除了要选用有足够的计算能力的硬件，也需要对软件运行环境做一定的调优。比如修改修改操作系统的全局最大文件句柄数，允许用户打开的文件句柄数，**TCP** 的 **backlog** 和 **buffer**，**Erlang** 虚拟机的进程数限制等等。甚至包括需要在客户端上做一定的调优以保证客户端可以有足够的连接资源。

系统的调优在不同的需求下有不同的方式，在 **EMQX** 的[文档-测试调优](#) 中对用于普通场景的调优有较详细的说明

EMQX 支持加密连接吗？推荐的部署方案是什么？

标签: [TLS 加密连接](#)

EMQX 支持加密连接。在生产环境部署时，推荐的方案是使用负载均衡终结 **TLS**。通过该方式，设备端和服务器端（负载均衡）的采用加密的连接，而负载均衡和后端的 **EMQX** 节点采用一般的 **TCP** 连接。

EMQX 安装之后无法启动怎么排查？

标签: [调试](#)

执行 `$ emqx console`，查看输出内容

- `logger` 命令缺失

```

1 $ emqx console
2 Exec: /usr/lib/emqx/erts-10.3.5.1/bin/erlexec -boot /usr/lib/emqx/releases/v3.2.1/emqx -mode em
3 edded -boot_var ERTS_LIB_DIR /usr/lib/emqx/erts-10.3.5.1/../lib -mnesia dir "/var/lib/emqx/mnes
4 sia/emqx@127.0.0.1" -config /var/lib/emqx/configs/app.2019.07.23.03.07.32.config -args_file /var/
5 ib/emqx/configs/vm.2019.07.23.03.07.32.args -vm_args /var/lib/emqx/configs/vm.2019.07.23.03.07.
2.args -- console
Root: /usr/lib/emqx
/usr/lib/emqx
/usr/bin/emqx: line 510: logger: command not found

```

解决办法：

- `Centos/Redhat`

```

1 $ yum install rsyslog

```

- `Ubuntu/Debian`

```

1 $ apt-get install bsutils

```

- `openssl` 缺失

```

1      $ emqx console
2      Exec: /emqx/erts-10.3/bin/erlexec -boot /emqx/releases/v3.2.1/emqx -mode embedded -boot_var
3      ERTS_LIB_DIR /emqx/erts-10.3/..../lib -mnesia dir "/emqx/data/mnesia/emqx@127.0.0.1" -config /emq
4      /data/configs/app.2019.07.23.03.34.43.config -args_file /emqx/data/configs/vm.2019.07.23.03.34.
5      .args -vm_args /emqx/data/configs/vm.2019.07.23.03.34.43.args -- console
6      Root: /emqx
7      /emqx
8      Erlang/OTP 21 [erts-10.3] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:32] [hipe
9      ]
10

11      {"Kernel pid terminated",application_controller,"{application_start_failure,kernel,{{shutdown,{failed_to_start_child,kernel_safe_sup,{on_load_function_failed,crypto}}},{{kernel,start,[normal,[]]}}}}"}
12      Kernel pid terminated (application_controller) ({application_start_failure,kernel,{{shutdown,{failed_to_start_child,kernel_safe_sup,{on_load_function_failed,crypto}}},{{kernel,start,[normal,[]]}}}})

13      Crash dump is being written to: log/crash.dump...done

```

解决办法： 安装**1.1.1**以上版本的 `openssl`

- `License` 文件缺失

```

1      $ emqx console
2      Exec: /usr/lib/emqx/erts-10.3.5.1/bin/erlexec -boot /usr/lib/emqx/releases/v3.2.1/emqx -mode
3      embedded -boot_var ERTS_LIB_DIR /usr/lib/emqx/erts-10.3.5.1/..../lib -mnesia dir "/var/lib/emqx/mn
4      sia/emqx@127.0.0.1" -config /var/lib/emqx/configs/app.2019.07.23.05.52.46.config -args_file /va
5      /lib/emqx/configs/vm.2019.07.23.05.52.46.args -vm_args /var/lib/emqx/configs/vm.2019.07.23.05.5
6      .46.args -- console
7      Root: /usr/lib/emqx
8      /usr/lib/emqx
9      Erlang/OTP 21 [erts-10.3.5.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:32] [hip
10     ]

11      Starting emqx on node emqx@127.0.0.1
12      Start http:management listener on 8080 successfully.
13      Start http:dashboard listener on 18083 successfully.
14      Start mqtt:tcp listener on 127.0.0.1:11883 successfully.
15      Start mqtt:tcp listener on 0.0.0.0:1883 successfully.
16      Start mqtt:ws listener on 0.0.0.0:8083 successfully.
17      Start mqtt:ssl listener on 0.0.0.0:8883 successfully.
18      Start mqtt:wss listener on 0.0.0.0:8084 successfully.
19      EMQX 3.2.1 is running now!
20      "The license certificate is expired!"
21      2019-07-23 05:52:51.355 [critical] The license certificate is expired!
22      2019-07-23 05:52:51.355 [critical] The license certificate is expired! System shutdown!
23      Stop mqtt:tcp listener on 127.0.0.1:11883 successfully.
24      Stop mqtt:tcp listener on 0.0.0.0:1883 successfully.
25      Stop mqtt:ws listener on 0.0.0.0:8083 successfully.
      Stop mqtt:ssl listener on 0.0.0.0:8883 successfully.
      Stop mqtt:wss listener on 0.0.0.0:8084 successfully.
      [os_mon] memory supervisor port (memsup): Erlang has closed
      [os_mon] cpu supervisor port (cpu_sup): Erlang has closed

```

解决办法： 登陆[emqx.io](#) 申请**license**或安装开源版的 **EMQX Broker**

EMQX中ssl resumption session的使用

标签: [TLS](#)

修改`emqx.conf`配置中的 `reuse_sessions = on` 并生效后。如果客户端与服务端通过 **SSL** 已经连接成功，当第二次遇到客户端连接时，会跳过 **SSL** 握手阶段，直接建立连接，节省连接时间，增加客户端连接速度。

MQTT 客户端断开连接统计

标签: [指标](#)

执行 `emqx_ctl listeners`，查看对应端口下的 `shutdown_count` 统计。

客户端断开链接错误码列表：

- `keepalive_timeout` : **MQTT keepalive** 超时
- `closed` : **TCP**客户端断开连接（客户端发来的**FIN**, 但没收到**MQTT DISCONNECT**）
- `normal` : **MQTT**客户端正常断开
- `einval` : **EMQX** 想向客户端发送一条消息，但是**Socket** 已经断开
- `function_clause` : **MQTT** 报文格式错误
- `etimedout` : **TCP** 发送超时（没有收到**TCP ACK** 回应）
- `proto_unexpected_c` : 在已经有一条**MQTT**连接的情况下重复收到了**MQTT**连接请求
- `idle_timeout` : **TCP** 连接建立 **15s** 之后，还没收到 **connect** 报文

安装部署

EMQX 推荐部署的操作系统是什么？

EMQX 支持跨平台部署在 **Linux**、**Windows**、**MacOS**、**ARM** 嵌入系统，生产系推荐在 **CentOS**、**Ubuntu**、**Debian** 等 **Linux** 发行版上部署。

EMQX 支持 Windows 操作系统吗？

支持。部署参考[文章](#)。

EMQX 如何预估资源的使用？

标签: [资源估算](#)

EMQX 对资源的使用主要有以下的影响因素，每个因素都会对计算和存储资源的使用产生影响：

- 连接数：对于每一个 **MQTT** 长连接，EMQX 会创建两个 **Erlang** 进程，每个进程都会耗费一定的资源。连接数越高，所需的资源越多；
- 平均吞吐量：指的是每秒 **Pub** 和 **Sub** 的消息数量。吞吐量越高，EMQX 的路由处理和消息转发处理就需要更多的资源；
- 消息体大小：消息体越大，在 EMQX 中处理消息转发的时候在内存中进行数据存储和处理，所需的资源就越多；
- 主题数目：如果主题数越多，在 EMQX 中的路由表会相应增长，因此所需的资源就越多；
- **QoS**：消息的 **QoS** 越高，EMQX 服务器端所处理的逻辑会更多，因此会耗费更多的资源；

另外，如果设备通过 **TLS**（加密的连接）连接 EMQX，EMQX 会需要额外的资源（主要是 **CPU** 资源）。推荐方案是在 EMQX 前面部署负载均衡，由负载均衡节点卸载 **TLS**，实现职责分离。

可参考 <https://www.emqx.com/zh/server-estimate> 来预估计算资源的使用；公有云快速部署 EMQX 实例，请参考[TODO](#)。

EMQX 的百万连接压力测试的场景是什么？

标签: [性能测试](#)

在**EMQ 2.0**版本发布的时候，由第三方软件测试工具服务提供商 [XMeter](#) 执行了一次百万级别连接的性能测试。测试基于开源社区中最流行的性能测试工具 [Apache JMeter](#)，以及开源[性能测试插件](#)。该性能测试场景为测试客户端到服务器端的**MQTT**协议连接，该测试场景下除了发送**MQTT**协议的控制包和**PING**包（每5分钟发送一次）外，不发送用户数据，每秒新增连接数为**1000**，共计运行**30分钟**。

在该测试中，还执行了一些别的性能测试，主要为在为**10万MQTT**背景连接的情况下，执行了不同条件下的消息发送和接收的场景。具体请参见[性能测试报告](#)。

我的连接数目并不大，EMQX 生产环境部署需要多节点吗？

标签: [集群](#)

即使在连接数量，消息率不高的情况下（服务器低负载），在生产环境下部署多节点的集群依然是很有意义的。集群能提高系统的可用性，降低单点故障的可能性。当一个节点宕机时，其他在线节点可以保证整个系统的服务不中断。

常见错误

EMQX 无法连接 MySQL 8.0

标签: [MySQL 认证](#)

提示

4.3 已兼容 `caching_sha2_password`, 该问题仅在 **4.3** 以下的版本中出现。

不同于以往版本, **MySQL 8.0** 对账号密码配置默认使用 `caching_sha2_password` 插件, 需要将密码插件改成 `mysql_native_password`

- 修改 `mysql.user` 表

```

1 ## 切换到 mysql 数据库
2 mysql> use mysql;
3
4 ## 查看 user 表
5
6 mysql> select user, host, plugin from user;
7 +-----+-----+-----+
8 | user      | host     | plugin          |
9 +-----+-----+-----+
10 | root       | %        | caching_sha2_password |
11 | mysql.infoschema | localhost | caching_sha2_password |
12 | mysql.session    | localhost | caching_sha2_password |
13 | mysql.sys        | localhost | caching_sha2_password |
14 | root       | localhost | caching_sha2_password |
15 +-----+-----+-----+
16
17 ## 修改密码插件
18 mysql> ALTER USER 'your_username'@'your_host' IDENTIFIED WITH mysql_native_password BY 'your_
19 password';
20 Query OK, 0 rows affected (0.01 sec)
21
22 ## 刷新
23 mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

- 修改 `my.conf`

在 `my.cnf` 配置文件里面的 **[mysqld]** 下面加一行

```
1 default_authentication_plugin=mysql_native_password
```

- 重启 MySQL 即可

OPENSSL 版本不正确

标签: [启动失败](#)

现象

执行 `./bin/emqx console` 输出的错误内容包含:

```
1 {application_start_failure,kernel,{{shutdown,{failed_to_start_child, kernel_safe_sup,{on_load
 _function_failed,crypto}}}, ..}}
```

它表示, **EMQX** 依赖的 **Erlang/OTP** 中的 `crypto` 应用启动失败。

解决方法

Linux

进入到 **EMQX** 的安装目录 (如果使用包管理工具安装 **EMQX**, 则应该进入与 **EMQX** 的 `lib` 目录同级的位置)

```
1 ## 安装包安装
2 $ cd emqx
3
4 ## 包管理器安装, 例如 yum。则它的 lib 目录应该在 /lib/emqx
5 $ cd /lib/emqx
```

查询 `crypto` 依赖的 `.so` 动态库列表及其在内存中的地址:

```
1 $ ldd lib/crypto->/priv/lib/crypto.so
2
3 lib/crypto-4.6/priv/lib/crypto.so: /lib64/libcrypto.so.10: version `OPENSSL_1.1.1' not found
4 (required by lib/crypto-4.6/priv/lib/crypto.so)
5     linux-vdso.so.1 => (0x00007fff67bfc000)
6     libcrypto.so.10 => /lib64/libcrypto.so.10 (0x00007fee749ca000)
7     libc.so.6 => /lib64/libc.so.6 (0x00007fee74609000)
8     libdl.so.2 => /lib64/libdl.so.2 (0x00007fee74404000)
9     libz.so.1 => /lib64/libz.so.1 (0x00007fee741ee000)
10    /lib64/ld-linux-x86-64.so.2 (0x00007fee74fe5000)
```

其中 `OPENSSL_1.1.1' not found` 表明指定的 **OPENSSL** 版本的 `.so` 库未正确安装。

源码编译安装 **OPENSSL 1.1.1**, 并将其 `so` 文件放置到可以被系统识别的路径:

```

1 ## 下载最新版本 1.1.1
2 $ wget https://www.openssl.org/source/openssl-1.1.1c.tar.gz
3
4 ## 上传至 ct-test-ha
5 $ scp openssl-1.1.1c.tar.gz ct-test-ha:~
6
7 ## 解压并编译安装
8 $ tar zxf openssl-1.1.1c.tar.gz
9 $ cd openssl-1.1.1c
10 $ ./config
11 $ make test      # 执行测试; 如果输出 PASS 则继续
12 $ make install
13
14 ## 确保库的引用
15 $ ln -s /usr/local/lib64/libssl.so.1.1 /usr/lib64/libssl.so.1.1
16 $ ln -s /usr/local/lib64/libcrypto.so.1.1 /usr/lib64/libcrypto.so.1.1

```

完成后，执行在 **EMQX** 的 **lib** 同级目录下执行 `ldd lib/crypto-*/priv/lib/crypto.so`，检查是否已能正确识别。如果不在有 `not found` 的 `.so` 库，即可正常启动 **EMQX**。

macOS

进入到 **EMQX** 的安装目录：

```

1 ## 安装包安装
2 $ cd emqx
3
4 ## brew 安装
5 $ cd /usr/local/Cellar/emqx/<version>

```

查询 `crypto` 依赖的 `.so` 动态库列表：

```

1 $ otool -L lib/crypto-*/priv/lib/crypto.so
2
3 lib/crypto-4.4.2.1/priv/lib/crypto.so:
4   /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib (compatibility version 1.1.0, current ve
5 rsion 1.1.0)
6   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1252.200.5)

```

检查其显示 **OPENSSL** 已成功安装至指定的目录：

```

1 $ ls /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib
2 ls: /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib: No such file or directory

```

若不存在该文件，则需安装与 `otool` 打印出来的对应的 **OPENSSL** 版本，例如此处显示的为 `openssl@1.1`：

```

1 $ brew install openssl@1.1

```

安装完成后，即可正常启动 **EMQX**。

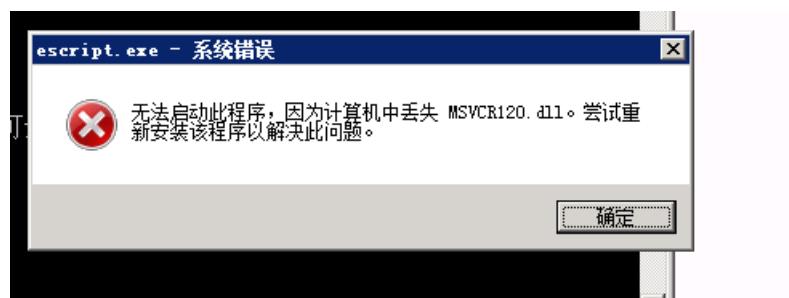
Windows 缺失 MSVCR120.dll

标签: [启动失败](#)

现象

Windows 执行 `./bin/emqx console` 弹出错误窗口:

1	无法启动此程序，因为计算机中丢失 <code>MSVCR120.dll</code> 。请尝试重新安装该程序以解决此问题。	sh
---	---	----



解决方法

安装 [Microsoft Visual C++ Redistributable Package](#)

SSL 连接失败

现象

客户端无法与 **EMQX** 建立 **SSL** 连接。

解决方法

可以借助 **EMQX** 日志中的关键字来进行简单的问题排查, **EMQX** 日志相关内容请参考: [日志与追踪](#)。

1. certificate_expired

日志中出现 `certificate_expired` 关键字, 说明证书已经过期, 请及时续签。

2. no_suitable_cipher

日志中出现 `no_suitable_cipher` 关键字, 说明握手过程中没有找到合适的密码套件, 可能原因有证书类型与密码套件不匹配、没有找到服务端和客户端同时支持的密码套件等等。

3. handshake_failure

日志中出现 `handshake_failure` 关键字, 原因有很多, 可能要结合客户端的报错来分析, 例如, 可能是客户端发现连接的服务器地址与服务器证书中的域名不匹配。

4. unknown_ca

日志中出现 `unknown_ca` 关键字, 意味着证书校验失败, 常见原因有遗漏了中间 **CA** 证书、未指定 **Root CA** 证书或者指定了错误的 **Root CA** 证书。在双向认证中我们可以根据日志中的其他信息来判断是服务端还是客户端的证书

配置出错。如果是服务端证书存在问题，那么报错日志通常为：

```
1 {ssl_error,{tls_alert,{unknown_ca,"TLS server: In state certify received CLIENT ALERT: Fatal - Unknown CA\n"}}}
```

看到 `CLIENT ALERT` 就可以得知，这是来自客户端的警告信息，服务端证书未能通过客户端的检查。

如果是客户端证书存在问题，那么报错日志通常为：

```
1 {ssl_error,{tls_alert,{unknown_ca,"TLS server: In state certify at ssl_handshake.erl:1887 generated SERVER ALERT: Fatal - Unknown CA\n"}}}
```

看到 `SERVER ALERT` 就能够得知，表示服务端在检查客户端证书时发现该证书无法通过认证，而客户端将收到来自服务端的警告信息。

5. `protocol_version`

日志中出现 `protocol_version` 关键字，说明客户端与服务器支持的 **TLS** 协议版本不匹配。

商业服务

EMQX 企业版（Enterprise）和开源版（Broker）的主要区别是什么？

标签: [企业版](#)

EMQX 企业版基于开源版，包含了开源版的所有功能。与开源版相比，主要有以下方面的区别：

- 接入设备量级：开源版的稳定接入为 **10** 万，而企业版为 **100** 万。
- 数据持久化：企业版支持将消息转储到各类持久化数据库中，包括流行的关系型数据库，比如 **MySQL**、**PostgreSQL**；内存数据库 **Redis**；非关系型数据库 **MongoDB** 等；
- **Kafka** 数据桥接：通过内置桥接插件高效转发 **MQTT** 消息到 **Kafka** 集群，用户可以通过消费 **Kafka** 消息来实现实时流式数据的处理；
- **RabbitMQ** 数据桥接：支持 **MQTT** 消息桥接转发 **RabbitMQ**，应用可以通过消费 **RabbitMQ** 消息来实现可能的异构系统的集成；
- 监控管理（**EMQX Control Center**）
 - **EMQX** 集群监控：包括连接、主题、消息和对话（**session**）统计等
 - **Erlang** 虚拟机监测：**Erlang** 虚拟机的进程、线程、内存、数据库和锁的使用等
 - 主机监控：**CPU**、内存、磁盘、网络和操作系统等各类指标
- 安全特性：通过配置基于 **TLS**、**DTLS** 的安全连接（证书）等来提供更高级别安全保证。

EMQX 提供方案咨询服务吗？

提供。**EMQX** 在为客户搭建物联网平台的咨询方面有丰富的经验，包括为互联网客户和电信运营商搭建千万级物联网平台的实践。包括如何搭建负载均衡、集群、安全策略、数据存储和分析方案等方面可以根据客户的需求制定方案，满足业务发展的需求。

标签

启动失败

- OPENSSL 版本不正确

企业版

- EMQX 有哪些产品？
- EMQX 企业版（Enterprise）和开源版（Broker）的主要区别是什么？

NB-IoT

- EMQX 与 NB-IoT、LoRAWAN 的关系是什么？

LoRAWAN

- EMQX 与 NB-IoT、LoRAWAN 的关系是什么？

多协议

- 什么是 WebSocket？什么情况下需要通过 WebSocket 去连接 EMQX 服务器？

License

- 怎样更新 EMQX license？

扩展

- EMQX 支持私有协议进行扩展吗？如支持应该如何实现？

资源估算

- EMQX 如何预估资源的使用？

认证鉴权

- 什么是认证鉴权？使用场景是什么？

WebHook

- 我可以捕获设备上下线的事件吗？该如何使用？

- 什么是 Hook? 使用场景是什么?

系统主题

- 我可以捕获设备上下线的事件吗? 该如何使用?
- 系统主题有何用处? 都有哪些系统主题?

消息队列

- 什么是 mqueue? 如何配置 mqueue?

WebSocket

- 什么是 WebSocket? 什么情况下需要通过 WebSocket 去连接 EMQX 服务器?

ACL

- 我想限定某些主题只为特定的客户端所使用, EMQX 该如何进行配置?

发布订阅

- 我想限定某些主题只为特定的客户端所使用, EMQX 该如何进行配置?

共享订阅

- 什么是共享订阅? 有何使用场景?

流量控制

- EMQX 能做流量控制吗?

离线消息

- 什么是离线消息?

代理订阅

- 什么是代理订阅? 使用场景是什么?

性能

- EMQX 是如何实现支持大规模并发和高可用的?

向外卖

- EMQX 是如何实现支持大规模并发和高可用的？

持久化

- 我可以把 MQTT 消息从 EMQX 转发其他消息中间件吗？例如 RabbitMQ？

HTTP API

- 在服务器端能够直接断开一个 MQTT 连接吗？

Dashboard

- 在服务器端能够直接断开一个 MQTT 连接吗？

Kafka

- EMQX 企业版中桥接 Kafka，一条 MQTT 消息到达 EMQX 集群之后就回 MQTT Ack 报文还是写入 Kafka 之后才回 MQTT Ack 报文？

桥接

- MQTT Broker（比如 Mosquitto）可以转发消息到 EMQX 吗？

配置

- EMQX 企业版中桥接 Kafka，一条 MQTT 消息到达 EMQX 集群之后就回 MQTT Ack 报文还是写入 Kafka 之后才回 MQTT Ack 报文？

集群

- EMQX 支持集群自动发现吗？有哪些实现方式？
- 我的连接数目并不大，EMQX 生产环境部署需要多节点吗？

RabbitMQ

- 我可以把 MQTT 消息从 EMQX 转发其他消息中间件吗？例如 RabbitMQ？

Mosquitto

- MQTT Broker（比如 Mosquitto）可以转发消息到 EMQX 吗？

Trace

- 我想跟踪特定消息的发布和订阅过程，应该如何做？

调试

- 为什么我做压力测试的时候，连接数目和吞吐量老是上不去，有系统调优指南吗？
- EMQX 安装之后无法启动怎么排查？

性能测试

- 为什么我做压力测试的时候，连接数目和吞吐量老是上不去，有系统调优指南吗？
- EMQX 的百万连接压力测试的场景是什么？

TLS

- EMQX 支持加密连接吗？推荐的部署方案是什么？
- EMQX中ssl resumption session的使用

加密连接

- EMQX 支持加密连接吗？推荐的部署方案是什么？

MySQL

- EMQX 无法连接 MySQL 8.0

认证

- EMQX 无法连接 MySQL 8.0

指标

- MQTT 客户端断开连接统计

MQTT 客户端库

本页选取各个编程语言中热门 **MQTT** 客户端库进行介绍说明，提供连接、发布、订阅、取消订阅基本功能代码示例。

- [MQTT C 客户端库](#)
- [MQTT Java 客户端库](#)
- [MQTT Go 客户端库](#)
- [MQTT Erlang 客户端库](#)
- [MQTT JavaScript 客户端库](#)
- [MQTT Python 客户端库](#)

MQTT 社区收录了完整的 **MQTT** 客户端库列表，本章节对热门的每个库都提供连接样例、支持度分析，你可以[点击此处](#)查看。

MQTT 客户端生命周期

MQTT 客户端整个生命周期的行为可以概括为：建立连接、订阅主题、接收消息并处理、向指定主题发布消息、取消订阅、断开连接。

标准的客户端库在每个环节都暴露出相应的方法，不同库在相同环节所需方法参数含义大致相同，具体选用哪些参数、启用哪些功能特性需要用户深入了解 **MQTT** 协议特性并结合实际应用场景而定。

以一个客户端连接并发布、处理消息为例，每个环节大致需要进行的步骤：

- 建立连接：
 - 指定 **MQTT Broker** 基本信息接入地址与端口
 - 指定传输类型是 **TCP** 还是 **MQTT over WebSocket**
 - 如果启用 **TLS** 需要选择协议版本并携带相应的证书
 - **Broker** 启用了认证鉴权则客户端需要携带相应的 **MQTT Username Password** 信息
 - 配置客户端参数如 **keepalive** 时长、**clean session** 回话保留标志位、**MQTT** 协议版本、遗嘱消息（**LWT**）等
- 订阅主题：连接建立成功后可以订阅主题，需要指定主题信息
 - 指定主题过滤器 **Topic**，订阅的时候支持主题通配符 `+` 与 `#` 的使用
 - 指定 **QoS**，根据客户端库和 **Broker** 的实现可选 **Qos 0 1 2**，注意部分 **Broker** 与云服务提供商不支持部分 **QoS** 级别，如 **AWS IoT**、阿里云 **IoT** 套件、**Azure IoT Hub** 均不支持 **QoS 2** 级别消息
 - 订阅主题可能因为网络问题、**Broker** 端 **ACL** 规则限制而失败
- 接收消息并处理：
 - 一般是在连接时指定处理函数，依据客户端库与平台的网络编程模型不同此部分处理方式略有不同
- 发布消息：向指定主题发布消息
 - 指定目标主题，注意该主题不能包含通配符 `+` 或 `#`，若主题中包含通配符可能会导致消息发布失败、客户端断开等情况（视 **Broker** 与客户端库实现方式）
 - 指定消息 **QoS** 级别，同样存在不同 **Broker** 与平台支持的 **QoS** 级别不同，如 **Azure IoT Hub** 发布 **QoS 2** 的消息将断开客户端连接
 - 指定消息体内容，消息体内容大小不能超出 **Broker** 设置最大消息大小
 - 指定消息 **Retain** 保留消息标志位

- 取消订阅：
 - 指定目标主题即可
- 断开连接：
 - 客户端主动断开连接，服务器端将发布遗嘱消息（**LWT**）

MQTT C 客户端库

[Eclipse Paho C](#) 与 [Eclipse Paho Embedded C](#) 均为 **Eclipse Paho** 项目下的 **C** 语言客户端库 (**MQTT C Client**)，均为使用 **ANSI C** 编写的功能齐全的 **MQTT** 客户端。

Eclipse Paho Embedded C 可以在桌面操作系统上使用，但主要针对 [mbed](#)，[Arduino](#) 和 [FreeRTOS](#) 等嵌入式环境。

该客户端有同步/异步两种 **API**，分别以 **MQTTClient** 和 **MQTTAsync** 开头：

- 同步 **API** 旨在更简单，更有用，某些调用将阻塞直到操作完成为止，使用编程上更加容易；
- 异步 **API** 中只有一个调用块 `API-waitForCompletion`，通过回调进行结果通知，更适用于非主线程的环境。

Paho C 使用示例

MQTT C 语言相关两个客户端库的比较、下载、使用方式等详细说明请移步至项目主页查看，本示例包含 **C** 语言的 **Paho C** 连接 **EMQX Broker**，并进行消息收发完整代码：

```

1 #include "stdio.h"
2 #include "stdlib.h"
3 #include "string.h"
4
5 #include "MQTTClient.h"
6
7 #define ADDRESS      "tcp://broker.emqx.io:1883"
8 #define CLIENTID    "emqx_test"
9 #define TOPIC        "testtopic/1"
10 #define PAYLOAD      "Hello World!"
11 #define QOS          1
12 #define TIMEOUT     10000L
13
14 int main(int argc, char* argv[])
15 {
16     MQTTClient client;
17     MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
18     MQTTClient_message pubmsg = MQTTClient_message_initializer;
19     MQTTClient_deliveryToken token;
20     int rc;
21
22     MQTTClient_create(&client, ADDRESS, CLIENTID,
23                       MQTTCLIENT_PERSISTENCE_NONE, NULL);
24
25     // MQTT 连接参数
26     conn_opts.keepAliveInterval = 20;
27     conn_opts.cleansession = 1;
28
29     if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
30     {
31         printf("Failed to connect, return code %d\n", rc);
32         exit(-1);
33     }
34
35     // 发布消息
36     pubmsg.payload = PAYLOAD;
37     pubmsg.payloadlen = strlen(PAYLOAD);
38     pubmsg.qos = QOS;
39     pubmsg.retained = 0;
40     MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token);
41     printf("Waiting for up to %d seconds for publication of %s\n"
42           "on topic %s for client with ClientID: %s\n",
43           (int)(TIMEOUT/1000), PAYLOAD, TOPIC, CLIENTID);
44     rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
45     printf("Message with delivery token %d delivered\n", token);
46
47     // 断开连接
48     MQTTClient_disconnect(client, 10000);
49     MQTTClient_destroy(&client);
50     return rc;
51 }
```

Paho C MQTT 5.0 支持

目前 Paho C 已经完整支持 MQTT 5.0。

MQTT Java 客户端库

[Eclipse Paho Java Client](#) 是用 **Java** 编写的 **MQTT** 客户端库 (**MQTT Java Client**) , 可用于 **JVM** 或其他 **Java** 兼容平台 (例如[Android](#)) 。

Eclipse Paho Java Client 提供了**MqttAsyncClient** 和 **MqttClient** 异步和同步 API。

通过 Maven 安装 Paho Java

通过包管理工具 **Maven** 可以方便地安装 **Paho Java** 客户端库, 截止目前最新版本安装如下:

```

1 <dependency>
2   <groupId>org.eclipse.paho</groupId>
3   <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
4   <version>1.2.2</version>
5 </dependency>
```

Paho Java 使用示例

Java 体系中 **Paho Java** 是比较稳定、广泛应用的 **MQTT** 客户端库, 本示例包含 **Java** 语言的 **Paho Java** 连接 **EMQX Broker**, 并进行消息收发完整代码:

App.java

```

1 package io.emqx;
2
3 import org.eclipse.paho.client.mqttv3.MqttClient;
4 import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
5 import org.eclipse.paho.client.mqttv3.MqttException;
6 import org.eclipse.paho.client.mqttv3.MqttMessage;
7 import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
8
9
10 public class App {
11     public static void main(String[] args) {
12         String subTopic = "testtopic/#";
13         String pubTopic = "testtopic/1";
14         String content = "Hello World";
15         int qos = 2;
16         String broker = "tcp://broker.emqx.io:1883";
17         String clientId = "emqx_test";
18         MemoryPersistence persistence = new MemoryPersistence();
19
20         try {
21             MqttClient client = new MqttClient(broker, clientId, persistence);
22
23             // MQTT 连接选项
24             MqttConnectOptions connOpts = new MqttConnectOptions();
25             connOpts.setUserName("emqx_test");
26             connOpts.setPassword("emqx_test_password".toCharArray());
27             // 保留会话
28         } catch (MqttException e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

```
28     connOpts.setCleanSession(true);
29
30     // 设置回调
31     client.setCallback(new PushCallback());
32
33     // 建立连接
34     System.out.println("Connecting to broker: " + broker);
35     client.connect(connOpts);
36
37     System.out.println("Connected");
38     System.out.println("Publishing message: " + content);
39
40     // 订阅
41     client.subscribe(subTopic);
42
43     // 消息发布所需参数
44     MqttMessage message = new MqttMessage(content.getBytes());
45     message.setQos(qos);
46     client.publish(pubTopic, message);
47     System.out.println("Message published");
48
49     client.disconnect();
50     System.out.println("Disconnected");
51     client.close();
52     System.exit(0);
53 } catch (MqttException me) {
54     System.out.println("reason " + me.getReasonCode());
55     System.out.println("msg " + me.getMessage());
56     System.out.println("loc " + me.getLocalizedMessage());
57     System.out.println("cause " + me.getCause());
58     System.out.println("excep " + me);
59     me.printStackTrace();
60 }
61 }
62 }
63 }
```

回调消息处理类 **OnMessageCallback.java**

```
1 package io.emqx;
2
3 import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
4 import org.eclipse.paho.client.mqttv3.MqttCallback;
5 import org.eclipse.paho.client.mqttv3.MqttMessage;
6
7 public class OnMessageCallback implements MqttCallback {
8     public void connectionLost(Throwable cause) {
9         // 连接丢失后，一般在这里面进行重连
10        System.out.println("连接断开，可以做重连");
11    }
12
13    public void messageArrived(String topic, MqttMessage message) throws Exception {
14        // subscribe后得到的消息会执行到这里面
15        System.out.println("接收消息主题:" + topic);
16        System.out.println("接收消息Qos:" + message.getQos());
17        System.out.println("接收消息内容:" + new String(message.getPayload()));
18    }
19
20    public void deliveryComplete(IMqttDeliveryToken token) {
21        System.out.println("deliveryComplete-----" + token.isComplete());
22    }
23 }
```

java

Paho Java MQTT 5.0 支持

目前 Paho Java 还在适配 MQTT 5.0，尚未全面支持。

MQTT Go 客户端库

Eclipse Paho MQTT Go Client[↗] 为 Eclipse Paho 项目下的 Go 语言版客户端库，该库能够连接到 MQTT Broker 以发布消息，订阅主题并接收已发布的消息，支持完全异步的操作模式。

客户端依赖于 Google 的 proxy[↗] 和 websockets[↗] 软件包，通过以下命令完成安装：

```
1 go get github.com/eclipse/paho.mqtt.golang
```

sh

MQTT Go 使用示例

本示例包含 Go 语言的 Paho MQTT 连接 EMQX Broker，并进行消息收发完整代码：

```

1 package main
2
3 import (
4     "fmt"
5     "log"
6     "os"
7     "time"
8
9     "github.com/eclipse/paho.mqtt.golang"
10 )
11
12 var f mqtt.MessageHandler = func(client mqtt.Client, msg mqtt.Message) {
13     fmt.Printf("TOPIC: %s\n", msg.Topic())
14     fmt.Printf("MSG: %s\n", msg.Payload())
15 }
16
17 func main() {
18     mqtt.DEBUG = log.New(os.Stdout, "", 0)
19     mqtt.ERROR = log.New(os.Stdout, "", 0)
20     opts := mqtt.NewClientOptions().AddBroker("tcp://broker.emqx.io:1883").SetClientID("emqx_te
21 st_client")
22
23     opts.SetKeepAlive(60 * time.Second)
24     // 设置消息回调处理函数
25     opts.SetDefaultPublishHandler(f)
26     opts.SetPingTimeout(1 * time.Second)
27
28     c := mqtt.NewClient(opts)
29     if token := c.Connect(); token.Wait() && token.Error() != nil {
30         panic(token.Error())
31     }
32
33     // 订阅主题
34     if token := c.Subscribe("testtopic/#", 0, nil); token.Wait() && token.Error() != nil {
35         fmt.Println(token.Error())
36         os.Exit(1)
37     }
38
39     // 发布消息
40     token := c.Publish("testtopic/1", 0, false, "Hello World")
41     token.Wait()
42
43     time.Sleep(6 * time.Second)
44
45     // 取消订阅
46     if token := c.Unsubscribe("testtopic/#"); token.Wait() && token.Error() != nil {
47         fmt.Println(token.Error())
48         os.Exit(1)
49     }
50
51     // 断开连接
52     c.Disconnect(250)
53     time.Sleep(1 * time.Second)
54 }
```

go

Paho Golang MQTT 5.0 支持

目前 **Paho Golang** 还在适配 **MQTT 5.0**, 尚未全面支持。

MQTT Erlang 客户端库

[emqtt](#) 是开源 **MQTT Broker EMQX** 官方提供的客户端库，适用于 **Erlang** 语言。

Erlang 生态有多个 **MQTT Broker** 实现，如通过插件支持 **MQTT** 的 **RabbitMQ**、**VerenMQ**、**EMQX** 等。但受限于语言小众性，可用的 **MQTT** 客户端库几乎没有选择的余地，**MQTT** 社区收录的 **Erlang** 客户端库中 [emqtt](#) 是最佳选择。

emqtt 完全由 **Erlang** 实现，完成整支持 **MQTT v3.1.1** 和 **MQTT v5.0** 协议版本，支持 **SSL** 单双向认证与 **WebSocket** 连接。另一款 **MQTT** 基准测试工具 [emqtt_bench](#) 就基于该客户端库构建。

emqtt 使用示例

本示例包含 **Erlang** 的 **emqtt** 客户端库连接 **EMQX Broker**，并进行消息收发完整代码：

```

1 ClientId = <<"test">>.
2 {ok, ConnPid} = emqtt:start_link([{clientid, ClientId}]). 
3 {ok, _Props} = emqtt:connect(ConnPid).
4 Topic = <<"guide/#">>.
5 QoS = 1.
6 {ok, _Props, _ReasonCodes} = emqtt:subscribe(ConnPid, {Topic, QoS}).
7 {ok, _PktId} = emqtt:publish(ConnPid, <<"guide/1">>, <<"Hello World!">>, QoS).
8 %% If the qos of publish packet is 0, `publish` function would not return packetid.
9 ok = emqtt:publish(ConnPid, <<"guide/2">>, <<"Hello World!">>, 0).
10
11 %% Recursively get messages from mail box.
12 Y = fun (Proc) -> ((fun (F) -> F(F) end)((fun(ProcGen) -> Proc(fun() -> (ProcGen(ProcGen))() end) end))) end.
13 Rec = fun(Receive) -> fun()-> receive {publish, Msg} -> io:format("Msg: ~p~n", [Msg]), Receive();
14   _Other -> Receive() after 5 -> ok end end.
15   (Y(Rec))().
16
17 %% If you don't like y combinator, you can also try named function to recursively get messages
18 n erlang shell.
19 Receive = fun Rec() -> receive {publish, Msg} -> io:format("Msg: ~p~n", [Msg]), Rec();
20   _Other -> Rec() after 5 -> ok end end.
21   Receive().
22
23 {ok, _Props, _ReasonCode} = emqtt:unsubscribe(ConnPid, <<"guide/#">>).
24
25 ok = emqtt:disconnect(ConnPid).
```

emqtt MQTT 5.0 支持

目前 **emqtt** 已经完整支持 **MQTT 5.0**。

MQTT JavaScript 客户端库

[MQTT.js](#) 是 **JavaScript** 编写的，实现了 **MQTT** 协议客户端功能的模块，可以在浏览器和 **Node.js** 环境中使用。

由于 **JavaScript** 单线程特性，**MQTT.js** 是全异步 **MQTT** 客户端，**MQTT.js** 支持 **MQTT** 与 **MQTT over WebSocket**，在不同运行环境支持的度如下：

- 浏览器环境：**MQTT over WebSocket**（包括微信小程序、支付宝小程序等定制浏览器环境）
- **Node.js** 环境：**MQTT**、**MQTT over WebSocket**

不同环境里除了少部分连接参数不同，其他 **API** 均是相同的。

使用 **npm** 安装：

```
1 npm i mqtt
```

sh

使用 **CDN** 安装（浏览器）：

```
1 <script src="https://unpkg.com/mqtt/dist/mqtt.min.js"></script>
2 <script>
3     // 将在全局初始化一个 mqtt 变量
4     console.log(mqtt)
5 </script>
```

html

在安装 **Node.js** 的环境里，可以通过 `npm i mqtt -g` 命令全局安装以命令行的形式使用 **MQTT.js**。

```
1 npm i mqtt -g
2
3 mqtt help
4
5 > MQTT.js command line interface, available commands are:
6
7 * publish    publish a message to the broker
8 * subscribe  subscribe for updates from the broker
9 * version    the current MQTT.js version
10 * help       help about commands
11
12 > Launch 'mqtt help [command]' to know more about the commands.
```

sh

MQTT.js 使用示例

本示例包含 **JavaScript** 语言的 **MQTT.js** 连接 **EMQX Broker**，并进行消息收发完整代码：

```
1 // const mqtt = require('mqtt')
2 import mqtt from 'mqtt'
3
4 // 连接选项
5 const options = {
6     clean: true, // true: 清除会话, false: 保留会话
7     connectTimeout: 4000, // 超时时间
8     // 认证信息
9     clientId: 'emqx_test',
10    username: 'emqx_test',
11    password: 'emqx_test',
12 }
13
14 // 连接字符串, 通过协议指定使用的连接方式
15 // ws 未加密 WebSocket 连接
16 // wss 加密 WebSocket 连接
17 // mqtt 未加密 TCP 连接
18 // mqtts 加密 TCP 连接
19 // wxs 微信小程序连接
20 // alis 支付宝小程序连接
21 const connectUrl = 'wss://broker.emqx.io:8084/mqtt'
22 const client = mqtt.connect(connectUrl, options)
23
24 client.on('reconnect', (error) => {
25     console.log('正在重连:', error)
26 })
27
28 client.on('error', (error) => {
29     console.log('连接失败:', error)
30 })
31
32 client.on('message', (topic, message) => {
33     console.log('收到消息: ', topic, message.toString())
34 })
```

js

MQTT.js MQTT 5.0 支持

目前 **MQTT.js** 已经完整支持 **MQTT 5.0**。

MQTT Python 客户端库

Eclipse Paho Python 为 Eclipse Paho 项目下的 Python 语言版客户端库，该库能够连接到 MQTT Broker 以发布消息，订阅主题并接收已发布的消息。

使用 PyPi 包管理工具安装：

```
1 pip install paho-mqtt
```

sh

Paho Python 使用示例

本示例包含 Python 语言的 Paho Python 连接 EMQX Broker，并进行消息收发完整代码：

```
1 import paho.mqtt.client as mqtt
2
3 # 连接成功回调
4 def on_connect(client, userdata, flags, rc):
5     print('Connected with result code '+str(rc))
6     client.subscribe('testtopic/#')
7
8 # 消息接收回调
9 def on_message(client, userdata, msg):
10    print(msg.topic+" "+str(msg.payload))
11
12 client = mqtt.Client()
13
14 # 指定回调函数
15 client.on_connect = on_connect
16 client.on_message = on_message
17
18 # 建立连接
19 client.connect('broker.emqx.io', 1883, 60)
20 # 发布消息
21 client.publish('emqtt', payload='Hello World', qos=0)
22
23 client.loop_forever()
```

py

Paho Python MQTT 5.0 支持

目前 Paho Python 还在适配 MQTT 5.0，尚未全面支持。

EMQX MQTT 微信小程序接入

微信小程序支持通过 **WebSocket** 进行即时通信，**EMQX** 的 **MQTT Over WebSocket** 能够完全兼容使用在微信小程序上。

提示

由于微信小程序的规范限制，**EMQX** 使用微信小程序接入时需要注意以下几点：

- 必须使用已经通过[域名备案](#)的域名接入
- 域名需要在[小程序管理后台](#) 域名/IP 白名单中(开发 -> 开发设置 -> 服务器域名 -> **socket** 合法域名)
- 仅支持 **WebSocket/TLS** 协议，需要为域名分配受信任 **CA** 颁发的证书
- 由于微信小程序 **BUG**，安卓真机必须使用 **TLS/443** 端口，否则会连接失败（即连接地址不能带端口号）

参考资料

- [EMQX 使用 WebSocket 连接 MQTT 服务器](#)
- [CSDN Nginx 反向代理 WebSocket](#)
- [微信小程序 MQTT 接入 Demo](#)

详细步骤

1、注册微信小程序帐号，并下载[微信开发者工具](#)。由于微信小程序安全要求比较高，在与后台服务器之间的通讯必须使用 **https** 或 **wss** 协议，因此要在微信小程序后台设置域名服务器。

登录小程序后台后，找到 **开发设置 -> 服务器域名** 进行服务器配置，在 **socket** 合法域名处输入格式如 **wss://xxx.emqx.io**，此处不带端口号。

2、服务器端要申请并安装证书，华为云等云提供商均可以[申请免费证书](#)。

这里必须注意的是，证书申请绑定时，必须与所使用的服务器域名一致，此处建议使用 **Nginx** 来做反向代理并终结证书，相关配置如下：

```
1 server {
2     listen 443 ssl;
3     server_name xxx.emqx.io;
4     ssl_certificate cert/***.pem;
5     ssl_certificate_key cert/***.key;
6     ssl_session_timeout 5m;
7     ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE:ECDH:AES:HIGH:!NULL:!aNULL:!MD5:!ADH:!RC4;
8     ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
9     ssl_prefer_server_ciphers on;
10
11    # 添加反向代理
12    location /mqtt {
13        proxy_pass http://127.0.0.1:8083/mqtt;
14        proxy_set_header Host $host;
15        proxy_set_header X-Real-IP $remote_addr;
16        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
17        # client_max_body_size 35m;
18        proxy_http_version 1.1;
19        proxy_set_header Upgrade $http_upgrade;
20        proxy_set_header Connection "upgrade";
21    }
22
23 }
```

3、开源社区提供了小程序 **MQTT** 接入的 **Demo**: <https://github.com/iAoe444/WeChatMiniEsp8266> 下载解压到一个文件夹后，用微信小程序开发者工具，打开 `index.js` 文件，将 **MQTT** 地址、用户名和密码改为实际参数即可。

按照以上 3 步的安装配置，你的微信小程序已经能够成功连接到 **EMQX** 服务器了。

其他资源

本页整理提供 **EMQX** 客户端库、客户端工具、性能测试工具以及硬件等资源，如果你有相关推荐，欢迎提交到此页。

[MQTT Clients](#) [MQTT Tools](#) [CoAP Clients](#) [Hardware](#) [Library](#) [Benchmark Tools](#)



C

[Eclipse Paho C](#)



C

[Embedded C](#)



C

[libmosquitto](#)



C

[libemqtt](#)



C

[wolfMQTT](#)



C++

[Eclipse Paho C++](#)



C++

[Embedded C++](#)



Swift

[CocoaMQTT](#)



Erlang

[emqttc](#)



Erlang

[erlmqtt](#)



Java

[Eclipse Paho Java](#)



Java

[Xenqtt](#)



Java

MeQanTT



Java

mqtt-client



Javascript

Eclipse Paho



Javascript

MQTT.js



Javascript

node_mqtt_client



Javascript

ascoltatori



Objective-C

mqttlIO-objc



[Objective-C](#)

[MQTTKit](#)



[Objective-C](#)

[MQTT-Client](#)



[PHP](#)

[phpMQTT](#)



[PHP](#)

[Mosquitto-PHP](#)



[PHP](#)

[sskaje's MQTT library](#)



[Python](#)

[Eclipse Paho Python](#)



[Python](#)

[nyamuk](#)



[Python](#)

[MQTT-For-Twisted](#)



[Python](#)

[HBMQTT](#)



[Ruby](#)

[ruby-mqtt](#)



[Ruby](#)

[mosquitto](#)



[Swift](#)

[Aphid](#)



[Swift](#)

[SwiftMQTT](#)

协议介绍

MQTT协议

概览

MQTT是一个轻量的发布订阅模式消息传输协议，专门针对低带宽和不稳定网络环境的物联网应用设计。

MQTT官网: <http://mqtt.org>

MQTT V3.1.1协议规范: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>

特点

1. 开放消息协议，简单易实现
2. 发布订阅模式，一对多消息发布
3. 基于**TCP/IP**网络连接
4. 1字节固定报头，2字节心跳报文，报文结构紧凑
5. 消息**QoS**支持，可靠传输保证

应用

MQTT协议广泛应用于物联网、移动互联网、智能硬件、车联网、电力能源等领域。

1. 物联网**M2M**通信，物联网大数据采集
2. **Android**消息推送，**WEB**消息推送
3. 移动即时消息，例如**Facebook Messenger**
4. 智能硬件、智能家具、智能电器
5. 车联网通信，电动汽车充电桩采集
6. 智慧城市、远程医疗、远程教育
7. 电力、石油与能源等行业市场

MQTT基于主题(**Topic**)消息路由

MQTT协议基于主题(**Topic**)进行消息路由，主题(**Topic**)类似**URL**路径，例如:

```

1  chat/room/1
2
3  sensor/10/temperature
4
5  sensor/+/temperature
6
7  $SYS/broker/metrics/packets/received
8
9  $SYS/broker/metrics/#
```

主题(**Topic**)通过'/'分割层级，支持'*'，'#'通配符：

```
1 '+'： 表示通配一个层级，例如a/+, 匹配a/x, a/y  
2 '#'： 表示通配多个层级，例如a/#, 匹配a/x, a/b/c/d
```

订阅者与发布者之间通过主题路由消息进行通信，例如采用**mosquitto**命令行发布订阅消息：

```
1 mosquitto_sub -t a/b/+ -q 1  
2  
3 mosquitto_pub -t a/b/c -m hello -q 1
```

sh

订阅者可以订阅含通配符主题，但发布者不允许向含通配符主题发布消息。

MQTT V3.1.1协议报文

报文结构

- 固定报头(**Fixed header**)
- 可变报头(**Variable header**)
- 报文有效载荷(**Payload**)

固定报头

```
1 +-----+-----+-----+-----+-----+-----+-----+  
2 | Bit    | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |  
3 +-----+-----+-----+-----+-----+-----+-----+  
4 | byte1  | MQTT Packet type |          Flags      |  
5 +-----+-----+-----+-----+-----+-----+-----+  
6 | byte2... | Remaining Length |  
7 +-----+-----+-----+-----+
```

报文类型

类型名称	类型值	报文说明
CONNECT	1	发起连接
CONNACK	2	连接回执
PUBLISH	3	发布消息
PUBACK	4	发布回执
PUBREC	5	QoS2 消息回执
PUBREL	6	QoS2 消息释放
PUBCOMP	7	QoS2 消息完成
SUBSCRIBE	8	订阅主题
SUBACK	9	订阅回执
UNSUBSCRIBE	10	取消订阅
UNSUBACK	11	取消订阅回执
PINGREQ	12	PING 请求
PINGRESP	13	PING 响应
DISCONNECT	14	断开连接

PUBLISH发布消息

PUBLISH报文承载客户端与服务器间双向的发布消息。 **PUBACK**报文用于接收端确认**QoS1**报文， **PUBREC/PUBREL/PUBCOMP**报文用于**QoS2**消息流程。

PINGREQ/PINGRESP心跳

客户端在无报文发送时，按保活周期(**KeepAlive**)定时向服务端发送**PINGREQ**心跳报文，服务端响应**PINGRESP**报文。**PINGREQ/PINGRESP**报文均2个字节。

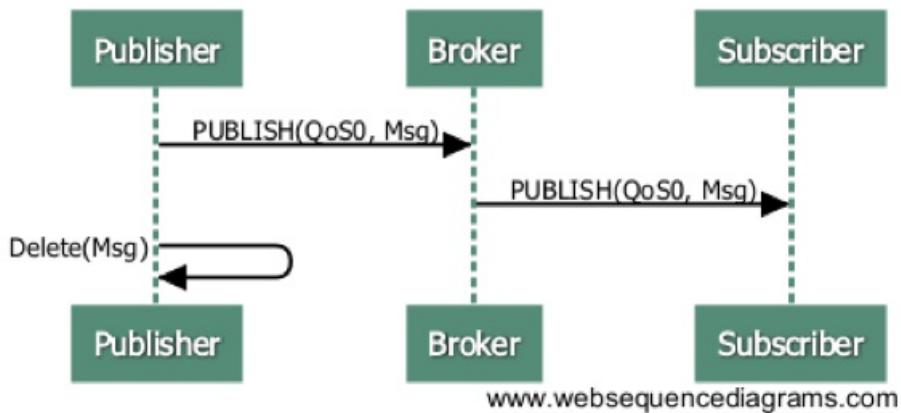
MQTT消息**QoS**

MQTT发布消息**QoS**保证不是端到端的，是客户端与服务器之间的。订阅者收到**MQTT**消息的**QoS**级别，最终取决于发布消息的**QoS**和主题订阅的**QoS**。

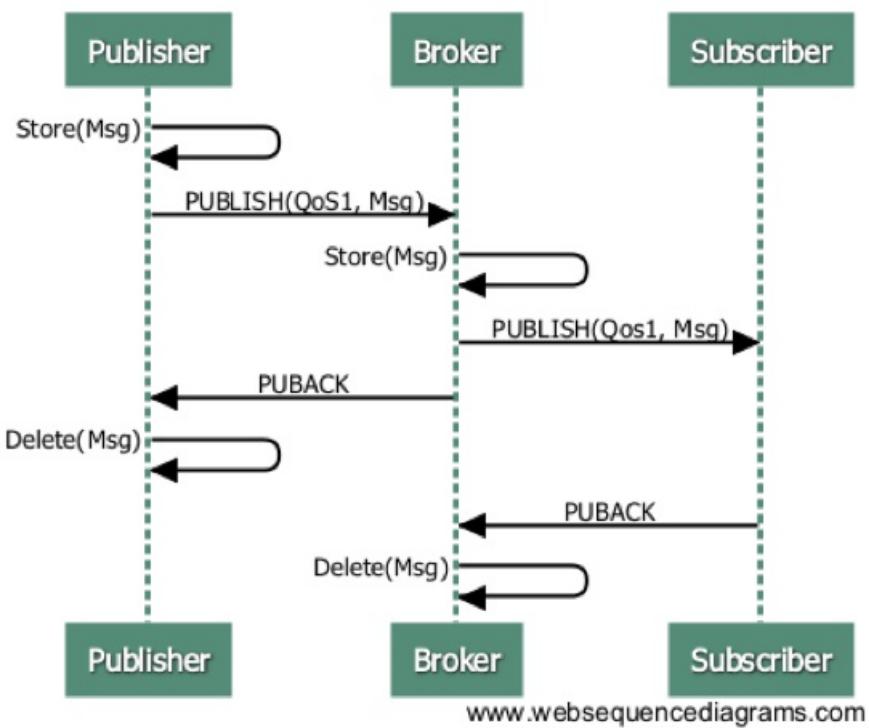
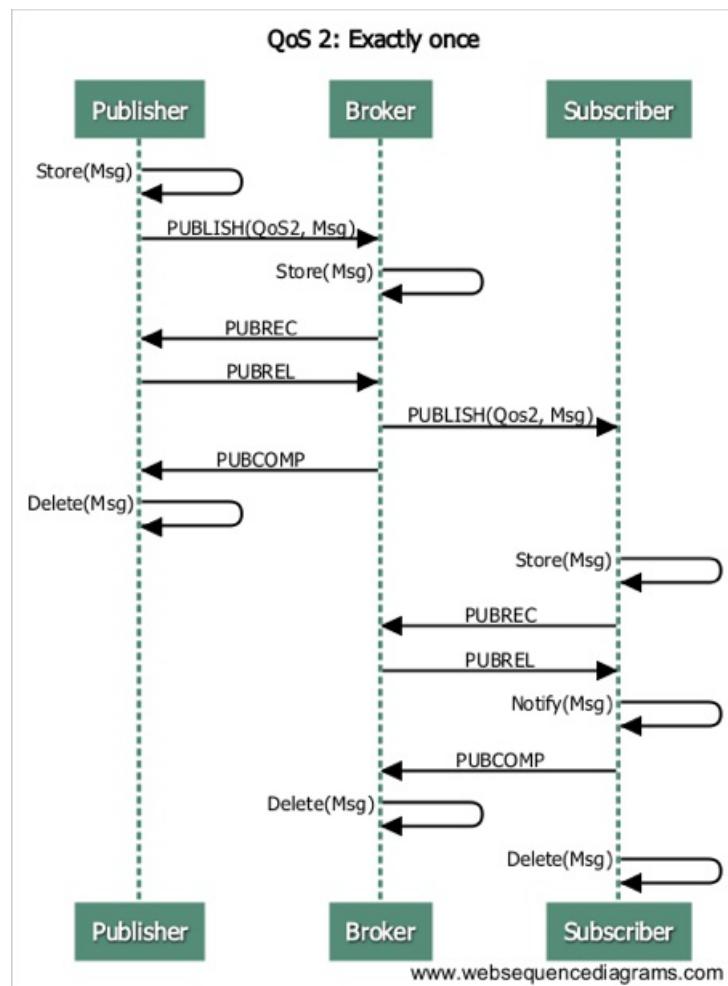
发布消息的 QoS	主题订阅的 QoS	接收消息的 QoS
0	0	0
0	1	0
0	2	0
1	0	0
1	1	1
1	2	1
2	0	0
2	1	1
2	2	2

Qos0消息发布订阅

QoS 0: At most once(deliver and forgot)



Qos1消息发布订阅

QoS 1: At least once**QoS2消息发布订阅****MQTT会话(Clean Session)**

MQTT客户端向服务器发起**CONNECT**请求时，可以通过‘**Clean Session**’标志设置会话。

‘**Clean Session**’设置为**0**，表示创建一个持久会话，在客户端断开连接时，会话仍然保持并保存离线消息，直到会话超时注销。

‘**Clean Session**’设置为**1**，表示创建一个新的临时会话，在客户端断开时，会话自动销毁。

MQTT连接保活心跳

MQTT客户端向服务器发起**CONNECT**请求时，通过**KeepAlive**参数设置保活周期。

客户端在无报文发送时，按**KeepAlive**周期定时发送**2**字节的**PINGREQ**心跳报文，服务端收到**PINGREQ**报文后，回复**2**字节的**PINGRESP**报文。

服务端在**1.5**个心跳周期内，既没有收到客户端发布订阅报文，也没有收到**PINGREQ**心跳报文时，主动心跳超时断开客户端**TCP**连接。

emqx 消息服务器默认按最长 **2.5** 心跳周期超时设计。

MQTT遗愿消息(**Last Will**)

MQTT客户端向服务器端**CONNECT**请求时，可以设置是否发送遗愿消息(**Will Message**)标志，和遗愿消息主题(**Topic**)与内容(**Payload**)。

MQTT客户端异常下线时(客户端断开前未向服务器发送**DISCONNECT**消息)，**MQTT**消息服务器会发布遗愿消息。

MQTT保留消息(**Retained Message**)

MQTT客户端向服务器发布(**PUBLISH**)消息时，可以设置保留消息(**Retained Message**)标志。保留消息(**Retained Message**)会驻留在消息服务器，后来的订阅者订阅主题时仍可以接收该消息。

例如**mosquitto**命令行发布一条保留消息到主题'**a/b/c**':

```
1 mosquitto_pub -r -q 1 -t a/b/c -m 'hello'
```

之后连接上来的**MQTT**客户端订阅主题'**a/b/c**'时候，仍可收到该消息：

```
1 $ mosquitto_sub -t a/b/c -q 1
2 hello
```

保留消息(**Retained Message**)有两种清除方式：

1. 客户端向有保留消息的主题发布一个空消息：

```
1 mosquitto_pub -r -q 1 -t a/b/c -m ''
```

2. 消息服务器设置保留消息的超期时间。

MQTT WebSocket连接

MQTT协议除支持**TCP**传输层外，还支持**WebSocket**作为传输层。通过**WebSocket**浏览器可以直连**MQTT**消息服务器，发布订阅模式与其他**MQTT**客户端通信。

MQTT协议的**WebSocket**连接，必须采用**binary**模式，并携带子协议**Header**:

1	Sec-WebSocket-Protocol: mqttv3.1 或 mqttv3.1.1
---	---

MQTT 与 XMPP 协议对比

MQTT协议设计简单轻量、路由灵活，将在移动互联网物联网消息领域，全面取代**PC**时代的**XMPP**协议：

1. **MQTT**协议一个字节固定报头，两个字节心跳报文，报文体积小编解码容易。**XMPP**协议基于繁重的**XML**，报文体积大且交互繁琐。
2. **MQTT**协议基于主题(**Topic**)发布订阅模式消息路由，相比**XMPP**基于**JID**的点对点消息路由更为灵活。
3. **MQTT**协议未定义报文内容格式，可以承载**JSON**、二进制等不同类型报文。**XMPP**协议采用**XML**承载报文，二进制必须**Base64**编码等处理。
4. **MQTT**协议支持消息收发确认和**QoS**保证，**XMPP**主协议并未定义类似机制。**MQTT**协议有更好的消息可靠性保证。

MQTT-SN 协议

MQTT-SN 协议是 **MQTT** 的直系亲属，它使用 **UDP** 进行通信，标准的端口是**1884**。**MQTT-SN** 的主要目的是为了适应受限的设备和网络，比如一些传感器，只有很小的内存和 **CPU**，**TCP** 对于这些设备来说非常奢侈。还有一些网络，比如 **ZIGBEE**，报文的长度在**300**字节以下，无法承载太大的数据包。所以 **MQTT-SN** 的数据包更小巧。

MQTT-SN 的官方标准下载地址: http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf

MQTT-SN 和 MQTT 的区别

MQTT-SN 的信令和 **MQTT** 大部分都相同，比如都有 **Will**，都有 **Connect/Subsribe/Publish** 命令。

MQTT-SN 最大的不同是，**Topic** 使用 **TopicId** 来代替，而 **TopicId** 是一个**16**比特的数字。每一个数字对应一个 **Topic**，设备和云端需要使用 **REGISTER** 命令映射 **TopicId** 和 **Topic** 的对应关系。

MQTT-SN 可以随时更改 **Will** 的内容，甚至可以取消。而 **MQTT** 只允许在 **CONNECT** 时设定 **Will** 的内容，而且不允许更改。

MQTT-SN 的网络中有网关这种设备，它负责把 **MQTT-SN** 转换成 **MQTT**，和云端的 **MQTT Broker** 通信。**MQTT-SN** 的协议支持自动发现网关的功能。

MQTT-SN 还支持设备的睡眠功能，如果设备进入睡眠状态，无法接收 **UDP** 数据，网关将把下行的 **PUBLISH** 消息缓存起来，直到设备苏醒后再传送。

EMQX-SN 网关插件

EMQX-SN 是 **EMQX** 的一个网关插件，实现了 **MQTT-SN** 的大部分功能，它相当于一个在云端的 **MQTT-SN** 网关，直接和 **EMQ X Broker** 相连。

配置参数

```

1 # File: etc/plugins/emqx_sn.conf:
2
3 mqtt.sn.port = 1884
4
5 mqtt.sn.advertise_duration = 900
6
7 mqtt.sn.gateway_id = 1
8
9 mqtt.sn.username = mqtt_sn_user
10
11 mqtt.sn.password = abc

```

sh

配置项	说明
mqtt.sn.port	指定 MQTT-SN 监听的端口号
mqtt.sn.advertise_duration	ADVERTISE 消息的发送间隔(秒)
mqtt.sn.gateway_id	网关 ID
mqtt.sn.username	这是可选的参数，指定所有 MQTT-SN 连接的用户名，用于鉴权模块
mqtt.sn.password	这也是可选的参数，和 username 一起使用

启动 emqx-sn

```
1 ./bin/emqx_ctl plugins load emqx_sn
```

sh

MQTT-SN 客户端库

1. <https://github.com/eclipse/paho.mqtt-sn.embedded-c>
2. <https://github.com/ty4tw/MQTT-SN>
3. <https://github.com/njh/mqtt-sn-tools>
4. <https://github.com/arobenko/mqtt-sn>

LWM2M 协议

LwM2M 全称是 **Lightweight Machine-To-Machine**，是由 **Open Mobile Alliance(OMA)** 定义的一套适用于物联网的轻量级协议，它提供了设备管理和通讯的功能，尤其适用于资源有限的终端设备。协议可以在 [这里](#) 下载。

LwM2M 基于 **REST** 架构，使用 **CoAP** 作为底层的传输协议，承载在 **UDP** 或者 **SMS** 上，因而报文结构简单小巧，并且在网络资源有限及无法确保设备始终在线的环境里同样适用。

LwM2M 最主要的实体包括 **LwM2M Server** 和 **LwM2M Client**。

LwM2M Server 作为服务器，部署在 **M2M** 服务供应商处或网络服务供应商处。**LwM2M** 定义了两种服务器

- 一种是 **LwM2M BOOTSTRAP SERVER**，**emqx-lwm2m** 插件并未实现该服务器的功能。
- 一种是 **LwM2M SERVER**，**emqx-lwm2m** 实现该服务器在 **UDP** 上的功能，**SMS** 并没有实现。

LwM2M Client 作为客户端，部署在各个 **LwM2M** 设备上。

在 **LwM2M Server** 和 **LwM2M Client** 之间，**LwM2M** 协议定义了4个接口。

1. 引导接口 **Bootstrap**: 向 **LwM2M** 客户端提供注册到 **LwM2M** 服务器的必要信息, 例如服务器访问信息、客户端支持的资源信息等。
2. 客户端注册接口 **Client Registration**: 使 **LwM2M** 客户端与 **LwM2M** 服务器互联, 将 **LwM2M** 客户端的相关信息存储在 **LwM2M** 服务器上。只有完成注册后, **LwM2M** 客户端与服务器端之间的通信与管理才成为可能。
3. 设备管理与服务实现接口 **Device Management and Service Enablement**: 该接口的主控方为 **LwM2M** 服务器, 服务器向客户端发送指令, 客户端对指令做出回应并将回应消息发送给服务器。
4. 信息上报接口 **Information Reporting**: 允许 **LwM2M** 服务器端向客户端订阅资源信息, 客户端接收订阅后按照约定的模式向服务器端报告自己的资源变化情况。

LwM2M 把设备上的服务抽象为 **Object** 和 **Resource**, 在 **XML** 文件中定义各种 **Object** 的属性和功能。可以在 [这里](#) 找到 **XML** 的各种定义。

LwM2M 协议预定义了**8种 Object** 来满足基本的需求, 分别是:

- **Security** 安全对象
- **Server** 服务器对象
- **Access Control** 访问控制对象
- **Device** 设备对象
- **Connectivity Monitoring** 连通性监控对象
- **Firmware** 固件对象
- **Location** 位置对象
- **Connectivity Statistics** 连通性统计对象

EMQX-LWM2M 插件

EMQX-LWM2M 是 **EMQX** 服务器的一个网关插件, 实现了 **LwM2M** 的大部分功能。**MQTT** 客户端可以通过 **EMQX-LWM2M** 访问支持 **LwM2M** 的设备。设备也可以往 **EMQX-LWM2M** 上报 **notification**, 为 **EMQX** 后端的服务采集数据。

MQTT 和 LwM2M的转换

从 **MQTT** 客户端可以发送 **Command** 给 **LwM2M** 设备。**MQTT** 到 **LwM2M** 的命令使用如下的 **topic**

```
1 "lwm2m/{?device_end_point_name}/command".
```

其中 **MQTT Payload** 是一个 **json** 格式的字符串, 指定要发送的命令, 更多的细节请参见 [emqx-lwm2m](#) 的文档。

LwM2M 设备的回复用如下 **topic** 传递

```
1 "lwm2m/{?device_end_point_name}/response".
```

MQTT Payload 也是一个 **json** 格式的字符串, 更多的细节请参见 [emqx-lwm2m](#) 的文档。

配置参数

```

1 ## File: etc/emqx_lwm2m.conf:
2
3 lwm2m.port = 5683
4
5 lwm2m.certfile = etc/certs/cert.pem
6
7 lwm2m.keyfile = etc/certs/key.pem
8
9 lwm2m.xml_dir = etc/lwm2m_xml

```

配置项	说明
lwm2m.port	指定 LwM2M 监听的端口号, 为了避免和 emqx-coap 冲突, 使用了非标准的 5783 端口
lwm2m.certfile	DTLS 使用的证书
lwm2m.keyfile	DTLS 使用的秘钥
lwm2m.xml_dir	存放 XML 文件的目录, 这些 XML 用来定义 LwM2M Object

启动 emqx-lwm2m

```
1 ./bin/emqx_ctl plugins load emqx_lwm2m
```

LwM2M 的客户端库

- <https://github.com/eclipse/wakaama>
- <https://github.com/OpenMobileAlliance/OMA-LWM2M-DevKit>
- <https://github.com/AVSystem/Anjay>
- <http://www.eclipse.org/leshan/>

私有 TCP 协议

EMQX 提供 **emqx-tcp** 插件, 插件作为一个靠近端侧的一个接入模块, 按照其功能逻辑和整个系统的关系, 将整个消息交换的过程可以分成三个部分: 终端侧, 平台侧和其它侧:

```

1 |<-- Terminal -->|<----- Broker Side ----->|<--- Others --->|
2 |<- Side ->|                                |<-- Side -->|
3
4 +---+                               PUB +-----+
5 | D | INCOMING +-----+   PUB   +-----+ -->| subscriber |
6 | E | ----->|           |----->|           |--/ +-----+
7 | V |           | emqx-tcp |           | EMQX |           |
8 | I |<-----|           |<-----|           |<-- +-----+
9 | C | OUTGOING +-----+   PUB   +-----+ \--| publisher |
10 | E |                               PUB +-----+
11 +---+

```

1. 终端侧, 通过本插件定义的 **TCP** 私有协议进行接入, 然后实现数据的上报, 或者接收下行的消息。
2. 平台侧, 主体是 **emqx-tcp** 插件和 **EMQX** 系统。 **emqx-tcp** 负责报文的编解码, 代理订阅下行主题。实现将上行消息转为 **EMQX** 系统中的 **MQTT** 消息 **PUBLISH** 到整个系统中; 将下行的 **MQTT** 消息转化为 **TCP** 私有协议的报文结

构，下发到终端。

3. 其它侧，可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅，以接收上行消息。或对发布消息到具体的下行的主题，以发送数据到终端侧。

配置说明

协议层

```

1  ## 闲置时间。超过该时间未收到 CONNECT 帧，将
2  ## 直接关闭该 TCP 连接
3  tcp.proto.idle_timeout = 1s
4
5  ## 上行主题。上行消息到 EMQ 系统中的消息主题
6  ##
7  ## 占位符：
8  ##### - %c: 接入客户端的 ClientId
9  ##### - %u: 接入客户端的 Username
10 tcp.proto.up_topic = tcp/%c/up
11
12 ## 下行主题。客户端接入成功后，emqx-tcp 会订阅
13 ## 该主题，以接收 EMQ 系统向该类型的客户端下
14 ## 发的消息。
15 ##
16 ## 占位符：(同上)
17 tcp.proto.dn_topic = tcp/%c/dn
18
19 ## 最大处理的单个 TCP 私有协议报文大小
20 tcp.proto.max_packet_size = 65535
21
22 ## 开启状态统计。开启后，emqx-tcp 会定期更新
23 ## 连接信息，并检测连接的健康状态
24 tcp.proto.enable_stats = on
25
26 ## 强制 GC，当进程已处理 1000 消息或发送数据超过 1M
27 tcp.proto.force_gc_policy = 1000|1MB
28
29 ## 强制关闭连接，当进程堆积 8000 消息或堆栈内存超过 800M
30 tcp.proto.force_shutdown_policy = 8000|800MB

```

sh

监听器

监听器配置比较广泛，在此仅列举部分常用部分：

```
1 ## 配置监听的端口
2 tcp.listener.external = 0.0.0.0:8090
3
4 ## 配置监听池大小。影响 TCP 建链的并发速率。
5 tcp.listener.external.acceptors = 8
6
7 ## 最大连接数
8 tcp.listener.external.max_connections = 1024000
9
10 ## 每秒支持的最大并发连接数
11 tcp.listener.external.max_conn_rate = 1000
12
```

LwM2M 协议网关

协议介绍

LwM2M 全称是 **Lightweight Machine-To-Machine**，是由 **Open Mobile Alliance(OMA)** 定义的一套适用于物联网的轻量级协议，它提供了设备管理和通讯的功能，尤其适用于资源有限的终端设备。协议可以在 [这里](#) 下载。

LwM2M 基于 **REST** 架构，使用 **CoAP** 作为底层的传输协议，承载在 **UDP** 或者 **SMS** 上，因而报文结构简单小巧，并且在网络资源有限及无法确保设备始终在线的环境里同样适用。

LwM2M 最主要的实体包括 **LwM2M Server** 和 **LwM2M Client**。

LwM2M Server 作为服务器，部署在 **M2M** 服务供应商处或网络服务供应商处。**LwM2M** 定义了两种服务器

- 一种是 **LwM2M BOOTSTRAP SERVER**，**emqx-lwm2m** 插件并未实现该服务器的功能。
- 一种是 **LwM2M SERVER**，**emqx-lwm2m** 实现该服务器在 **UDP** 上的功能，**SMS** 并没有实现。

LwM2M Client 作为客户端，部署在各个 **LwM2M** 设备上。

在 **LwM2M Server** 和 **LwM2M Client** 之间，**LwM2M** 协议定义了4个接口。

1. 引导接口 **Bootstrap**：向 **LwM2M** 客户端提供注册到 **LwM2M** 服务器的必要信息，例如服务器访问信息、客户端支持的资源信息等。
2. 客户端注册接口 **Client Registration**：使 **LwM2M** 客户端与 **LwM2M** 服务器互联，将 **LwM2M** 客户端的相关信息存储在 **LwM2M** 服务器上。只有完成注册后，**LwM2M** 客户端与服务器端之间的通信与管理才成为可能。
3. 设备管理与服务实现接口 **Device Management and Service Enablement**：该接口的主控方为 **LwM2M** 服务器，服务器向客户端发送指令，客户端对指令做出回应并将回应消息发送给服务器。
4. 信息上报接口 **Information Reporting**：允许 **LwM2M** 服务器端向客户端订阅资源信息，客户端接收订阅后按照约定的模式向服务器端报告自己的资源变化情况。

LwM2M 把设备上的服务抽象为 **Object** 和 **Resource**，在 **XML** 文件中定义各种 **Object** 的属性和功能。可以在 [这里](#) 找到 **XML** 的各种定义。

LwM2M 协议预定义了8种 **Object** 来满足基本的需求，分别是：

- **Security** 安全对象
- **Server** 服务器对象
- **Access Control** 访问控制对象
- **Device** 设备对象
- **Connectivity Monitoring** 连通性监控对象
- **Firmware** 固件对象
- **Location** 位置对象
- **Connectivity Statistics** 连通性统计对象

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

模块管理 4 选择

搜索模块...

- Recon** Recon
- 内置访问控制文件** 内置访问控制文件
- 上下线通知** 客户端上下线通知
- MQTT 保留消息** MQTT 保留消息

选择 LwM2M 协议接入网关:

选择模块 28 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

搜索模块...

协议接入

- LwM2M 接入网关** EMQ X LwM2M 接入网关
- COAP 接入网关** EMQ X COAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

消息下发

配置相关基础参数:

配置信息

- * 最小心跳时间: 1
- * 最大心跳时间: 864000
- * QMode 窗口: 22
- 自动 Observe: false
- * 挂载点: lwm2m/%e/
- * 下行命令主题: dn/#
- * 上行应答主题: up/resp
- * 注册消息主题: up/resp
- * 上行通知主题: up/notify
- * 更新消息主题: up/resp
- * XML 文件路径: etc/lwm2m_xml

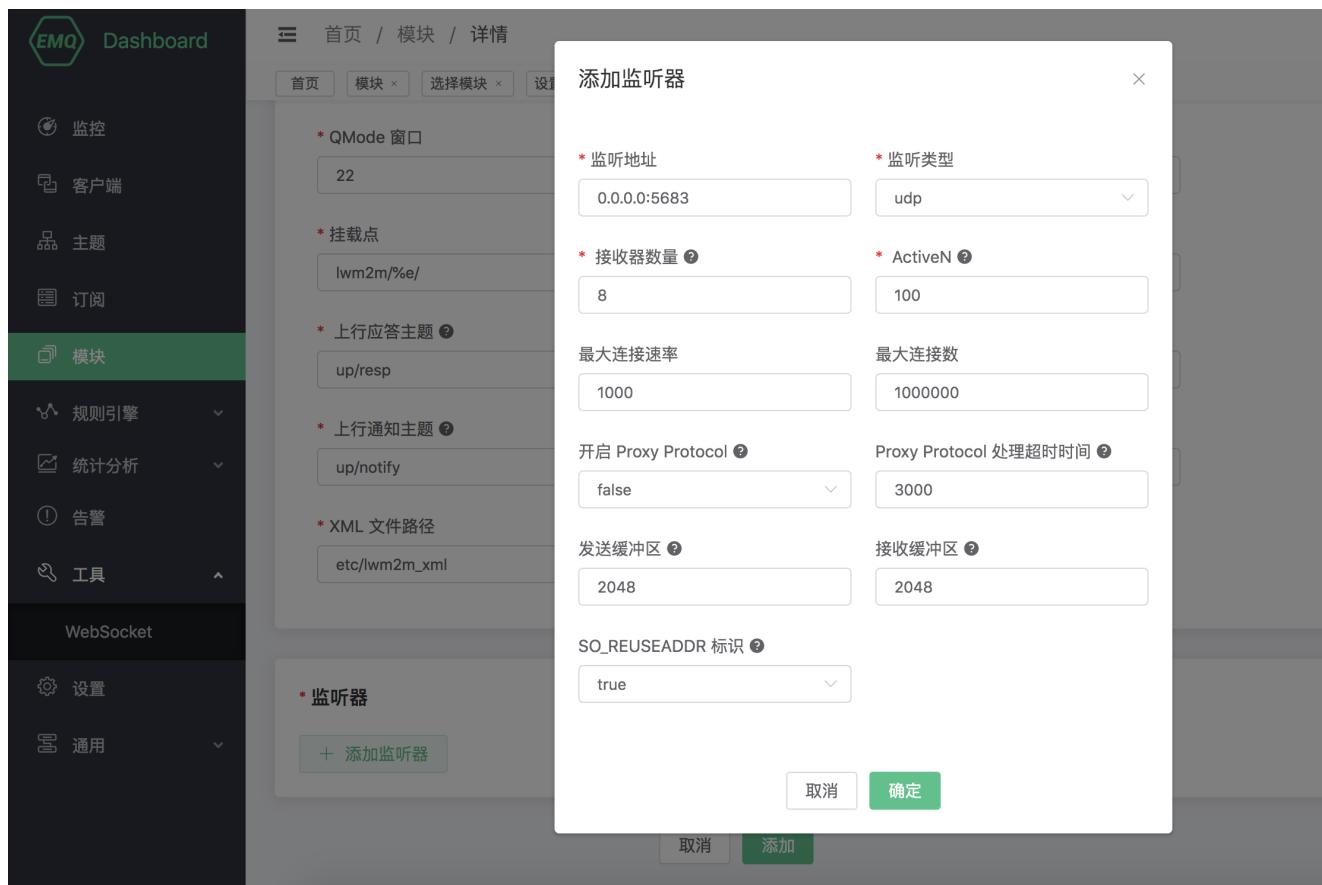
监听器

添加监听端口:

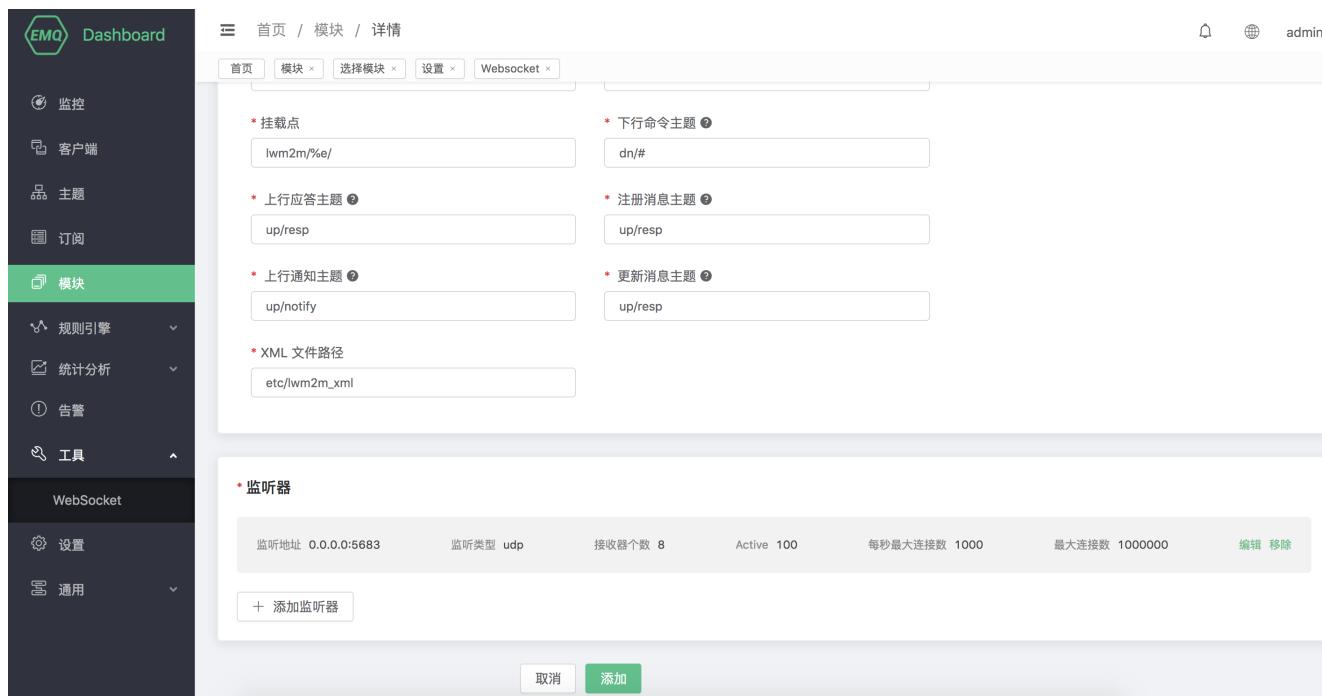
监听器

+ 添加监听器

配置监听参数:



点击确认到配置参数页面:



点击添加后，模块添加完成:

The screenshot shows the EMQX Enterprise V4.4 Dashboard. On the left is a sidebar with various menu items like Dashboard, Monitoring, Client, Theme, Subscriptions, Modules (which is selected), Rules Engine, Statistics Analysis, Alarms, Tools, WebSocket, and Settings. The main area is titled 'Module Management' and shows three modules: 'Online/Offline Notifications' (客户端上下线通知), 'ACL' (内置访问控制文件), and 'LwM2M Gateway' (LwM2M 接入网关). The 'LwM2M Gateway' module is highlighted with a red border.

EMQX-LWM2M 是 **EMQX** 服务器的一个网关模块，实现了 **LwM2M** 的大部分功能。**MQTT** 客户端可以通过 **EMQX-LWM2M** 访问支持 **LwM2M** 的设备。设备也可以往 **EMQX-LWM2M** 上报 **notification**，为 **EMQX** 后端的服务采集数据。

配置参数

配置项	说明
最小心跳时间	注册/更新允许设置的最小 lifetime , 以秒为单位
最大心跳时间	注册/更新允许设置的最大 lifetime , 以秒为单位
QMode 窗口	QMode 时间窗口, 指示发送到客户机的下行命令经过多长时间后将被缓存, 以秒为单位
自动 Observe	注册成功后, 是否自动 Observe 上报的 objectlist
挂载点	主题前缀
下行命令主题	下行命令主题 %e 表示取值 endpoint name
上行应答主题	上行应答主题 %e 表示取值 endpoint name
注册消息主题	注册消息主题 %e 表示取值 endpoint name
上行通知主题	上行通知主题 %e 表示取值 endpoint name
更新消息主题	更新消息主题 %e 表示取值 endpoint name
XML 文件路径	存放 XML 文件的目录, 这些 XML 用来定义 LwM2M Object

MQTT 和 LwM2M 的转换

从 **MQTT** 客户端可以发送 **Command** 给 **LwM2M** 设备。**MQTT** 到 **LwM2M** 的命令使用如下的 **topic**

```
1 "lwm2m/{?device_end_point_name}/command".
```

其中 **MQTT Payload** 是一个 **json** 格式的字符串，指定要发送的命令，更多的细节请参见 **emqx-lwm2m** 的文档。

LwM2M 设备的回复用如下 **topic** 传送

```
1 "lwm2m/{?device_end_point_name}/response".
```

sh

MQTT Payload 也是一个 **json** 格式的字符串，更多的细节请参见 **emqx-lwm2m** 的文档。

MQTT-SN 协议网关

协议介绍

MQTT-SN 的信令和 **MQTT** 大部分都相同，比如都有 **Will**，都有 **Connect/Subscribe/Publish** 命令。

MQTT-SN 最大的不同是，**Topic** 使用 **TopicId** 来代替，而 **TopicId** 是一个16比特的数字。每一个数字对应一个 **Topic**，设备和云端需要使用 **REGISTER** 命令映射 **TopicId** 和 **Topic** 的对应关系。

MQTT-SN 可以随时更改 **Will** 的内容，甚至可以取消。而 **MQTT** 只允许在 **CONNECT** 时设定 **Will** 的内容，而且不允许更改。

MQTT-SN 的网络中有网关这种设备，它负责把 **MQTT-SN** 转换成 **MQTT**，和云端的 **MQTT Broker** 通信。**MQTT-SN** 的协议支持自动发现网关的功能。

MQTT-SN 还支持设备的睡眠功能，如果设备进入睡眠状态，无法接收 **UDP** 数据，网关将把下行的 **PUBLISH** 消息缓存起来，直到设备苏醒后再传送。

EMQX-SN 是 **EMQX** 的一个网关接入模块，实现了 **MQTT-SN** 的大部分功能，它相当于一个在云端的 **MQTT-SN** 网关，直接和 **EMQ X Broker** 相连。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

选择 **MQTT-SN** 接入网关模块：

选择模块 28 认证鉴权 协议接入 消息下发 多语言扩展 运维监控 内部模块

搜索模块...

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

配置相关基础参数:

用户名 ?

密码 ?

* 启动 QoS -1

* 启用数据统计

* 空闲超时时间

预设主题列表

MQTT 主题

reserved

添加

删除

* 监听器

+ 添加监听器

取消

添加

添加监听端口:

EMQ Dashboard

首页 / 模块 / 详情

配置信息

用户名: mqtt-sn-1 密码:

* 启动 QoS -1: false * 启用数据统计: false

* 空闲超时时间: 30000

预设主题列表

MQTT 主题

reserved 添加 删除

监听器

+ 添加监听器

取消 添加

配置监听参数:

EMQ Dashboard

首页 / 模块 / 详情

配置信息

用户名: mqtt-sn-1

* 启动 QoS -1: false

* 空闲超时时间: 30000

预设主题列表

MQTT 主题

reserved

监听器

+ 添加监听器

添加监听器

* 监听地址: 0.0.0.0:1184 * 监听类型: udp

* 网关ID: 1 * 接收器数量: 8

* ActiveN: 100 最大连接速率: 1000

最大连接数: 1000000 开启 Proxy Protocol: false

Proxy Protocol 处理超时时间: 3000 发送缓冲区: 2048

接收缓冲区: 2048 SO_REUSEADDR 标识: true

取消 确定

点击确认到配置参数页面:

点击添加后，模块添加完成：

配置参数

配置项	说明
用户名	可选的参数，指定所有 MQTT-SN 连接的用户名，用于 EMQX 鉴权模块
密码	可选的参数，和 username 一起使用于 EMQX 鉴权模块

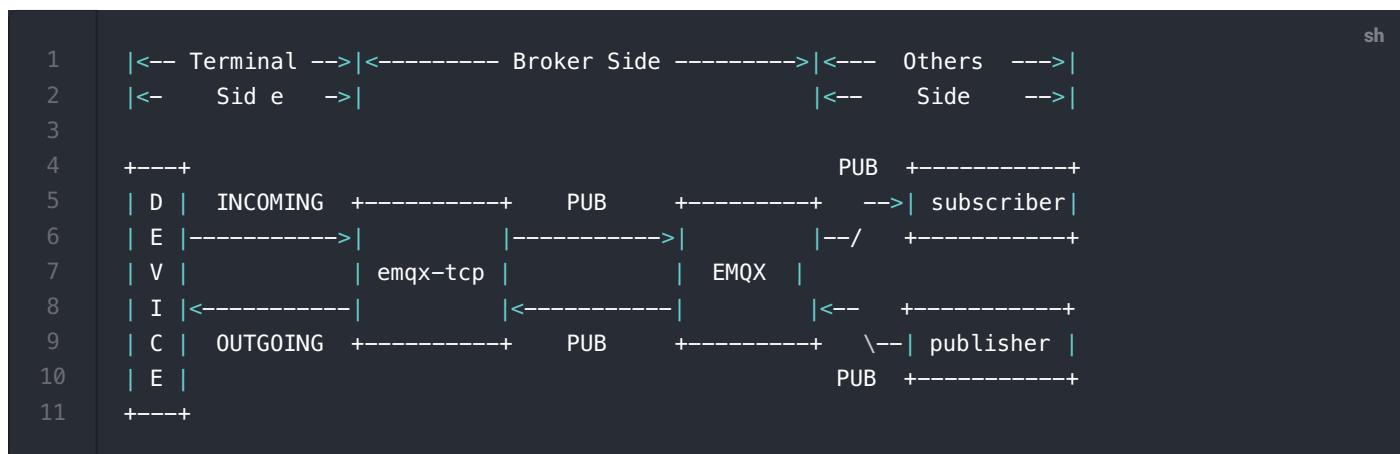
MQTT-SN 客户端库

1. <https://github.com/eclipse/paho.mqtt-sn.embedded-c/>
2. <https://github.com/ty4tw/MQTT-SN>
3. <https://github.com/njh/mqtt-sn-tools>
4. <https://github.com/arobenko/mqtt-sn>

TCP 协议网关

协议介绍

EMQX 提供 **emqx-tcp** 模块作为一个靠近端侧的一个接入模块，按照其功能逻辑和整个系统的关系，将整个消息交换的过程可以分成三个部分：终端侧，平台侧和其它侧：



1. 终端侧，通过本模块定义的 **TCP** 私有协议进行接入，然后实现数据的上报，或者接收下行的消息。
2. 平台侧，主体是 **emqx-tcp** 模块和 **EMQX** 系统。**emqx-tcp** 负责报文的编解码，代理订阅下行主题。实现将上行消息转为 **EMQX** 系统中的 **MQTT** 消息 **PUBLISH** 到整个系统中；将下行的 **MQTT** 消息转化为 **TCP** 私有协议的报文结构，下发到终端。
3. 其它侧，可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅，以接收上行消息。或对发布消息到具体的下行的主题，以发送数据到终端侧。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

模块	描述	状态	操作
Recon	Recon	未启用	了解更多
上下线通知	客户端上下线通知	未启用	了解更多
内置访问控制文件	内置访问控制文件	未启用	了解更多
MQTT 保留消息	MQTT 保留消息	未启用	了解更多

选择 **TCP** 私有协议接入网关：

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- Stomp 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

配置相关基础参数：

配置信息

* 空闲超时时间 15000	* 启用数据统计 false
* 上行主题 tcp/%c/up	* 下行主题 tcp/%c/dn
* 报文最大长度 65535	* 强制 GC 策略 1000 1MB
强制关闭策略 8000 800MB	

* 监听器

+ 添加监听器

添加监听端口：

配置信息

* 空闲超时时间: 15000 * 启用数据统计: false

* 上行主题: tcp/%c/up * 下行主题: tcp/%c/dn

* 报文最大长度: 65535 * 强制 GC 策略: 1000|1MB

强制关闭策略: 8000|800MB

* 监听器

+ 添加监听器

取消 添加

配置监听参数:

添加监听器

* 监听地址: 0.0.0.0:8090 * 监听类型: tcp

* 接收器数量: 8 * ActiveN: 100

最大连接速率: 1000 最大连接数: 1000000

开启 Proxy Protocol: false Proxy Protocol 处理超时时间: 3000

发送缓冲区: 2048 接收缓冲区: 2048

SO_REUSEADDR 标识: true TCP 连接队列长度: 1000

发送超时时间: 150000 关闭发送超时连接: true

TCP_NODELAY 标识: true

点击确认到配置参数页面:

配置信息

* 空闲超时时间 15000	* 启用数据统计 false
* 上行主题 tcp/%c/up	* 下行主题 tcp/%c/dn
* 报文最大长度 65535	* 强制 GC 策略 1000 1MB
强制关闭策略 8000 800MB	

* 监听器

监听地址 0.0.0.0:8090	监听类型 tcp	接收器个数 8	Active 100	每秒最大连接数 1000	最大连接数 1000000	编辑 移除
-------------------	----------	---------	------------	--------------	---------------	---------------------------------------

+ 添加监听器

取消 [添加](#)

点击添加后，模块添加完成：

模块管理 3 选择 搜索模块...

上下线通知 客户端上下线通知	内置访问控制文件 内置访问控制文件
TCP 接入网关 EMQ X TCP 接入网关	

配置参数

配置项	说明
空闲超时时间	闲置时间。超过该时间未收到 CONNECT 帧，将直接关闭该 TCP 连接
上行主题	上行消息到 EMQ 系统中的消息主题 %c: 接入客户端的 ClientId , %u: 接入客户端的 Username
下行主题	下行主题。上行消息到 EMQ 系统中的消息主题 %c: 接入客户端的 ClientId , %u: 接入客户端的 Username
报文最大长度	最大处理的单个 TCP 私有协议报文大小
强制 GC 策略	强制 GC ，当进程已处理 1000 消息或发送数据超过 1M
强制关闭策略	强制关闭连接，当进程堆积 8000 消息或堆栈内存超过 800M

私有 TCP 协议 - v1

设计准则

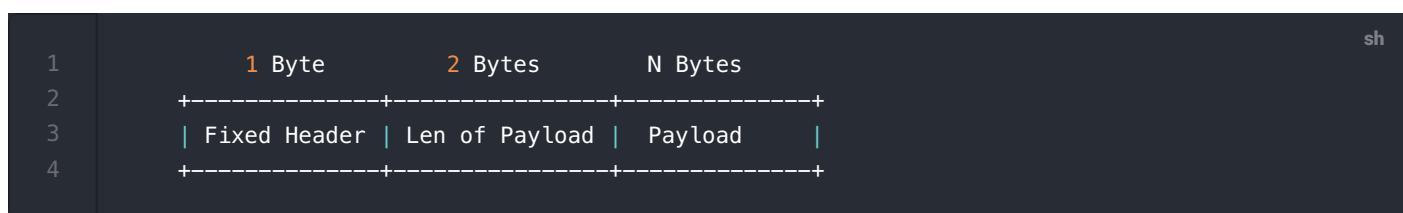
私有 TCP 协议的设计原则有三:

1. 轻量: 尽量减少头部、控制字段的字节大小
2. 简洁: 私有 TCP 协议, 主要功能定位在透传上层应用/协议的数据报文。所以功能应当简洁, 专注透明传输即可
3. 可靠: 保证消息有序可达

报文结构

报文主要有俩部分构成: 固定头部(**FixedHeader**) + 有效载荷(**Payload**)

其中固定头部固定 **1** 字节; 有效载荷为变长, 且前面有 **2** 个字节标识整个 **Payload** 的长度:



部分类型报文中不含 **Payload**; 则整个报文仅只有 **1** 个字节的固定头部

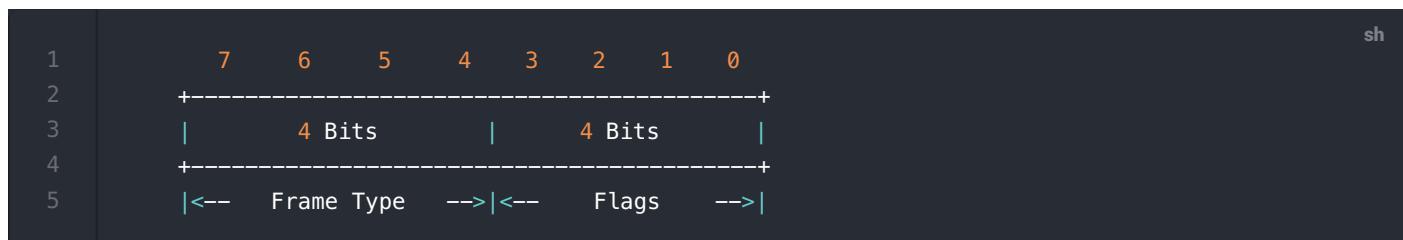
数据类型

本协议设计到的数据类型

Name	Bytes	Description
UINT(x)	x	固定 x 字节的 无符号整型
BIN(n)	变长	带 2 字节标示长度的变长二进制。n 取值为 0 到 65535 内容为空时, 需使用 2 个字节, 来标识长度值0
STR(n)	变长	带 2 字节标示长度的变长字符串。n 取值为 0 到 65535; 空串表达方式同上。
BIN	-	不带长度标识的二进制串

固定头部

固定头部有俩部分组成: 帧类型、标志位



帧类型(**Frame Type**) 有以下几种可选值

Name	Value	Direction of Flow	Description
CONNECT	1	Client --> Server	连接报文
CONNACK	2	Server --> Client	连接应答
DATATRANS	3	Client <==> Server	透明传输
PING	4	Client --> Server	心跳报文
PONG	5	Server --> Client	心跳应答
DISCONNECT	6	Client --> Server	主动断开连接
Reserved	7-15	保留	保留字段

标志位(**Flags**) 针对每个类型的报文，标志位代表的含义都不相同。

报文詳解

CONNECT 帧

连接报文. 帧类型为 **2#0001**. 标志位 **4 Bits** 代表协议 版本号(**Version**) 目前为 **1** 即 **2#0001**。因此 **CONNECT** 帧固定头部为 **0x11**

而，**Payload** 中包含连接用的所有字段。则必须按照以下顺序给出，否则为非法报文，立即断开 **TCP** 链接：

1	Len	Keepalive[x]	ClientId[x]	Username	Password	sh
2	UINT(2)	UINT(1)	STR(n)	STR(n)	STR(n)	

其中 **Keepalive** 和 **ClientId** 为必填字段；**Username** 与 **Password** 可不带。

因此，一个 **Keepalive** 为 **60**; **ClientId** 为 '**abcd**'; **Username** 和 **Password** 均为空时，报文的内容为：

1	0x11 00 07 3c 00 04 61 62 63 64	sh
---	---------------------------------	----

若是 **Username** 和 **Password** 不为空且假设都为 '**abcd**' 的情况下，报文内容为：

1	0x11 00 13 3c 00 04 61 62 63 64 00 04 61 62 63 64 00 04 61 62 63 64	sh
---	---	----

CONNACK 帧

连接应答报文. 帧类型为 **2#0010**. 标志位 **4 Bits** 为应答连接结果(**ACK Code**)。可以为以下枚举值：

Name	Value	Description
SUCCESSFUL	0	连接成功
AUTHFAILED	1	认证失败
ILLEGALVER	2	不支持的协议版本
Reserved	3-15	保留字段

而，**Payload** 字段，为连接应答后传递的 **Message**；该串可为空串。

1	Message
2	STR(n)

sh

所以，当连接成功时，并返回 `Connect Sucessfully` 时，报文内容为：

1	0x20 00 14 43 6f 6e 6e 65 63 74 20 53 75 63 63 65 73 73 66 75 6c 6c 79
---	--

sh

若是，返回 `认证失败` 且 **Message** 为空时::

1	0x21 00 00
---	------------

sh

DATATRANS 帧

数据传输帧。帧类型为 **2#0011**。标志位 前 **2 Bits** 表达 消息质量等级(**Qos**) 目前恒为**0**；后两位为保留位。所以 **DATATRANS** 帧固定头部恒为 **0x30**

Payload 内容为为透传的 数据字段

1	Len	Payload
2	UINT(2)	BIN
3		

sh

注：由于 **Len** 固定为 **2** 字节，所以最大仅支持 **65535** 字节的负载。

因此，如果透传 `abcd` 这个字符串时，该报文的内容为：

1	0x30 00 04 61 62 63 63
2	

sh

PING 帧

心跳帧。帧类型为 **2#0100**。标志位 **Flags** 固定为 **0**。即固定头部 固定为：**0x40**

Payload 为空

因此，一个 **PING** 帧仅有一个字节：

1	0x40
2	

sh

PONG 帧

心跳应答帧。帧类型为 **2#0101**。标志位 **Flags** 固定为 **0**。即固定头部 固定为：**0x50**

Payload 为空

因此，一个 **PONG** 帧仅有一个字节：

1	0x50	sh
---	------	----

DISCONNECT 帧

断开连接帧. 帧类型为 **2#0111**. Flags 为空。即固定头部 固定为: **0x60**

Payload 为空

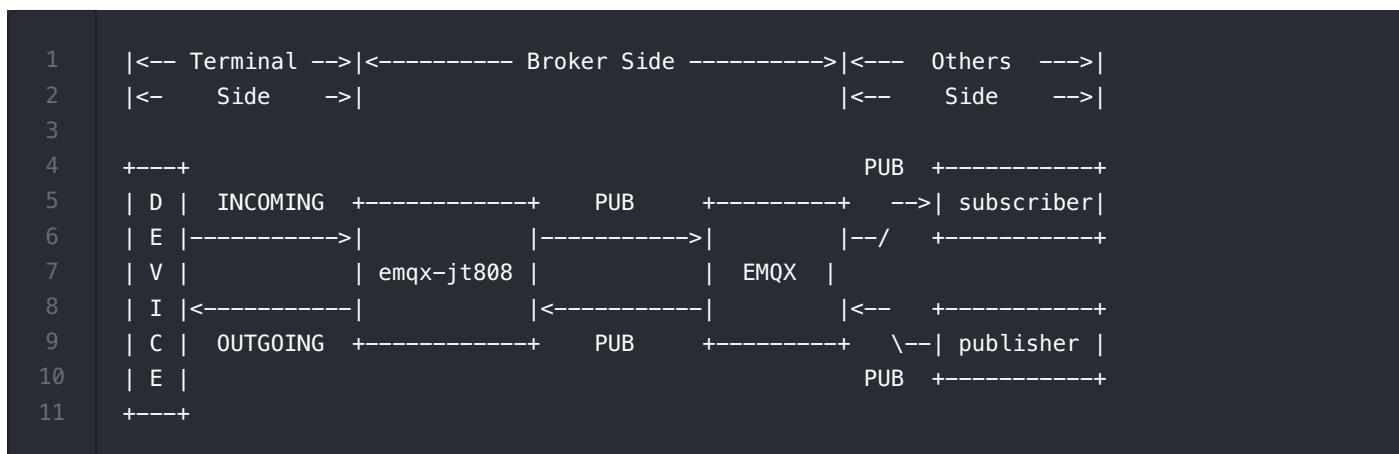
因此, **DISCONNECT** 帧仅有 **1** 个字节:

1	0x60	sh
---	------	----

JT/T808 协议网关

协议介绍

emqx-jt808 做为 **emqx** 的接入网关, 按照其功能逻辑和整个系统的关系, 将整个消息交换的过程可以分成三个部分: 终端侧, 平台侧和其它侧:



1. 终端侧: 通过 **JT/T 808** 协议进行交换数据, 实现不同类型的数据的上报, 或者发送下行的消息到终端。
2. 平台侧: **emqx-jt808** 将报文解码后执行 注册/鉴权、或将数据报文 **PUBLISH** 到特定的主题上; 代理订阅下行主题, 将下行的 **PUBLISH** 消息转化为 **JT/T 808** 协议的报文结构, 下发到终端。
3. 其它侧, 可以对 2 中出现的上行的 **PUBLISH** 消息的主题进行订阅, 以接收上行消息。或对发布消息到具体的下行的主题, 以发送数据到终端侧。

创建模块

打开 [EMQX Dashboard](#), 点击左侧的“模块”选项卡, 选择添加:

The screenshot shows the EMQX Dashboard interface. The left sidebar has a green highlighted '模块' (Module) tab. The main area displays a '模块管理' (Module Management) section with four cards: 'Recon' (selected), '上下线通知' (Online/Offline Notification), '内置访问控制文件' (Built-in Access Control File), and 'MQTT 保留消息' (MQTT Retained Message). Each card has a red power button icon and a '了解更多' (More Details) link.

选择 **JT/T808** 协议接入网关:

协议接入

- COAP 接入网关** EMQ X COAP 接入网关
- STOMP 接入网关** EMQ X Stomp 接入网关
- LwM2M 接入网关** EMQ X LwM2M 接入网关
- MQTT-SN 接入网关** EMQ X MQTT-SN 接入网关
- TCP 接入网关** EMQ X TCP 接入网关
- JT/T808 接入网关** EMQ X JT/T808 接入网关

消息下发

配置相关基础参数：

配置信息

* 空闲超时时间	* 允许匿名登录
15000	true
注册服务地址	认证服务地址
下行消息主题格式	上行消息主题格式
jt808/%c/dn	jt808/%c/up
* 是否开启统计	允许最大报文长度
false	8192

* 监听器

+ 添加监听器

取消 添加

添加监听端口：

配置信息

* 空闲超时时间: 15000 * 允许匿名登录: true

注册服务地址: 认证服务地址:

下行消息主题格式: jt808/%c/dn 上行消息主题格式: jt808/%c/up

* 是否开启统计: false 允许最大报文长度: 8192

* 监听器

+ 添加监听器

取消 添加

配置监听参数:

添加监听器

* 监听地址: 0.0.0.0:8090 * 监听类型: tcp

* 接收器数量: 8 * ActiveN: 100

最大连接速率: 1000 最大连接数: 1000000

开启 Proxy Protocol: false Proxy Protocol 处理超时时间: 3000

发送缓冲区: 2048 接收缓冲区: 2048

SO_REUSEADDR 标识: true TCP 连接队列长度: 1000

发送超时时间: 150000 关闭发送超时连接: true

TCP_NODELAY 标识: true

点击确认到配置参数页面:

点击添加后，模块添加完成：

emqx-jt808 实现规定：

- 系统内以手机号作为一个连接的唯一标识，即 **ClientId**

配置参数

emqx-jt808 的实现支持匿名的方式接入认证：

配置项	说明
允许匿名登录	是否允许匿名用户登录
注册服务地址	JT/T808 终端注册的 HTTP 接口地址
认证服务地址	JT/T808 终端接入鉴权的 HTTP 接口地址
下行消息主题格式	上行主题。上行消息到 EMQX 系统中的消息主题 %c : 接入客户端的 ClientId , %p : 接入客户端的 Phone
上行消息主题格式	下行主题。上行消息到 EMQX 系统中的消息主题 %c : 接入客户端的 ClientId , %p : 接入客户端的 Phone
允许最大报文长度	最大处理的单个 JT/T808 协议报文大小

注册及鉴权

注册请求详细格式如下：

注册请求：

```

1   URL: http://127.0.0.1:8991/jt808/registry
2   Method: POST
3   Body:
4     { "province": 58,
5       "city": 59,
6       "manufacturer": "Infinity",
7       "model": "Q2",
8       "license_number": "ZA334455",
9       "dev_id": "xx11344",
10      "color": 3,
11      "phone": "00123456789"
12    }

```

注册应答：

```

1   {
2     "code": 0,
3     "authcode": "132456789"
4   }
5   或:
6   {
7     "code": 1
8   }
12   其中返回码可以为:
13
14   0: 成功
15   1: 车辆已被注册
16   2: 数据库中无该车辆
17   3: 终端已被注册
18   4: 数据库中无该终端

```

鉴权请求：

```

1     URL: http://127.0.0.1:8991/jt808/auth
2     Method: POST
3     Body:
4         { "code": "authcode",
5             "phone", "00123456789"
6         }

```

鉴权应答：

```

1     HTTP 状态码 200: 鉴权成功
2     其他: 鉴权失败

```

注：鉴权请求只会在系统未保存鉴权码时调用（即终端直接发送鉴权报文进行登录系统）

数据上下行

emqx-jt808 中通过配置上下行主题来收发终端消息：

上行

例如：制造商**Id** 为 `abcde` 和 终端**Id** 为 `1234567` 的设备。

首先先使用 **MQTT** 客户端订阅主题 `jt808/abcde1234567/up`：

```

1     $ mosquitto_sub -t jt808/abcde1234567/up

```

例如终端在上报 `数据上行透传(0x0900)` 类型的消息后，订阅端会收到：

```

1     { "body":
2         { "data":"MTIzNDU2",
3             "type":240
4         },
5         "header":
6             { "encrypt":0,
7                 "len":7,
8                 "msg_id":2304,
9                 "msg_sn":4,
10                "phone":"011111111111"
11            }
12     }

```

js

注：透明传输类，**data** 域的内容会 **base64** 编码一次在上报出来

数据下行

同样，以上行的**ID**为例；在终端鉴权成功后，使用 **MQTT** 客户端向该终端下发一个 '数据下行透传(**0x8900**)' 类型的消息：

```
1 $ mosquitto_pub -t jt808/abcde1234567/dn -m '{"body":{"data":"MTIzNDU2","type":240},"header":{"sg_id":35072}}'
```

注：下行 **JSON** 中，**header** 中的内容只需要带 **msg_id** 即可；**body** 中的内容根据不同的 **msg_id** 有不同的结构

CoAP 协议网关

CoAP 协议网关为 **EMQX** 提供了 **CoAP** 协议的接入能力。它允许符合某种定义的 **CoAP** 消息格式向 **EMQX** 执行发布，订阅，和接收消息等操作。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：

The screenshot shows the EMQX Dashboard interface. On the left, there's a dark sidebar with various menu items like Monitoring, Client, Themes, Subscriptions, Rules Engine, Statistics, Modules (which is selected and highlighted in green), Alarms, Tools, Settings, and General. The main content area has a header '首页 / 模块'. Below it, there's a search bar and a button '选择' (Select). A list of modules is shown in cards:

- Recon**: Recon module.
- 上下线通知**: Client online/offline notification module.
- 内置访问控制文件**: Internal access control file module.
- MQTT 保留消息**: MQTT retained message module.

点击“选择”，然后选择“**CoAP** 接入网关”：

This screenshot shows the '选择模块' (Select Module) page within the EMQX Dashboard. The sidebar remains the same. The main area has a header '首页 / 模块 / 选择'. Below it, there's a search bar and a tab navigation: '选择模块' (31), 认证鉴权, 协议接入 (highlighted in green), 消息下发, 多语言扩展, 运维监控, 内部模块. Under the '协议接入' (Protocol Access) heading, there are several modules listed in a grid:

协议接入	模块名	描述	操作
CoAP	CoAP 接入网关	EMQ X CoAP 接入网关	选择
LwM2M	LwM2M 接入网关	EMQ X LwM2M 接入网关	选择
TCP	TCP 接入网关	EMQ X TCP 接入网关	选择
STOMP	Stomp 接入网关	EMQ X Stomp 接入网关	选择
MQTT	MQTT-SN 接入网关	EMQ X MQTT-SN 接入网关	选择
JT/T808	JT/T808 接入网关	EMQ X JT/T808 接入网关	选择

Below this, under '消息下发' (Message Delivery), there are two more modules:

消息下发	模块名	描述	操作
Kafka	Kafka 消费组	Kafka 消费组	选择
Pulsar	Pulsar 消费组	Pulsar 消费组	选择
MQTT	MQTT Subscriber	MQTT 订阅者	选择

配置相关基础参数：

COAP 接入网关
EMQ X COAP 接入网关

配置信息

- * 启用数据统计:

监听器

+ 添加监听器

取消 添加

添加监听端口：

COAP 接入网关
EMQ X COAP 接入网关

配置信息

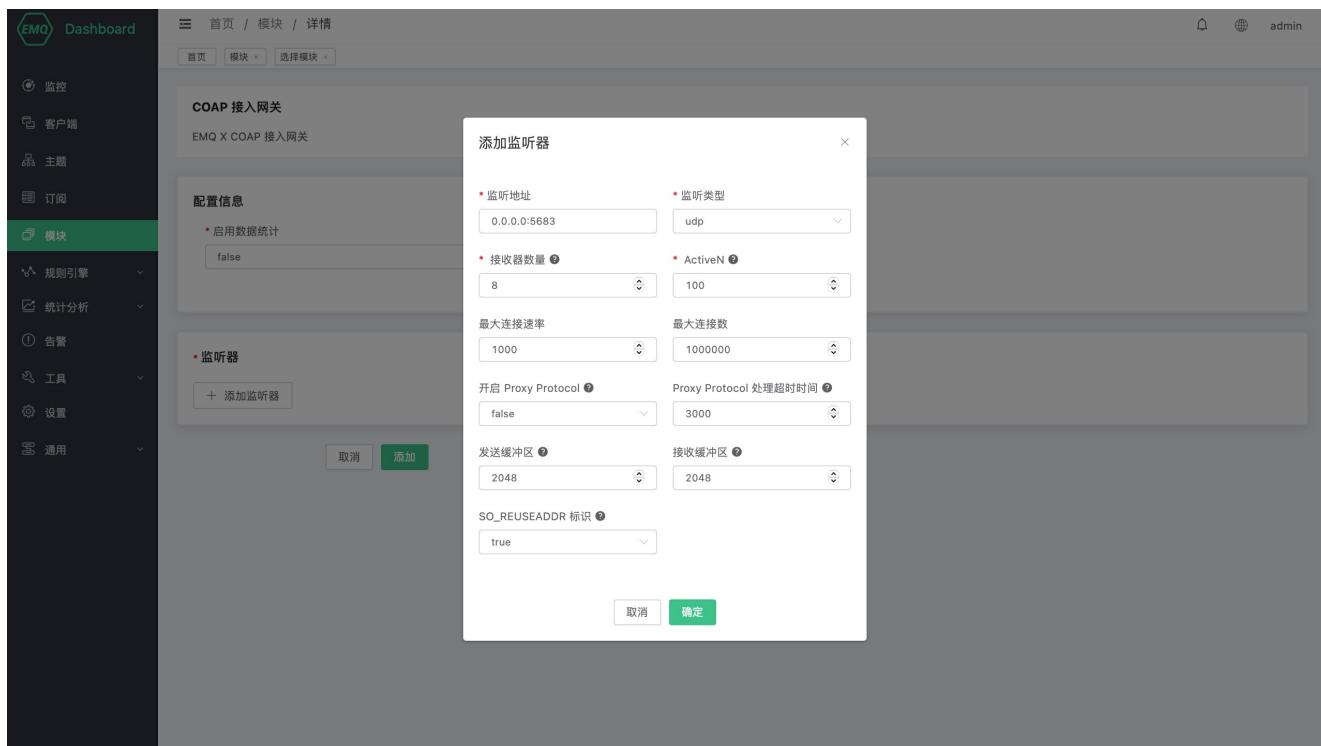
- * 启用数据统计:

监听器

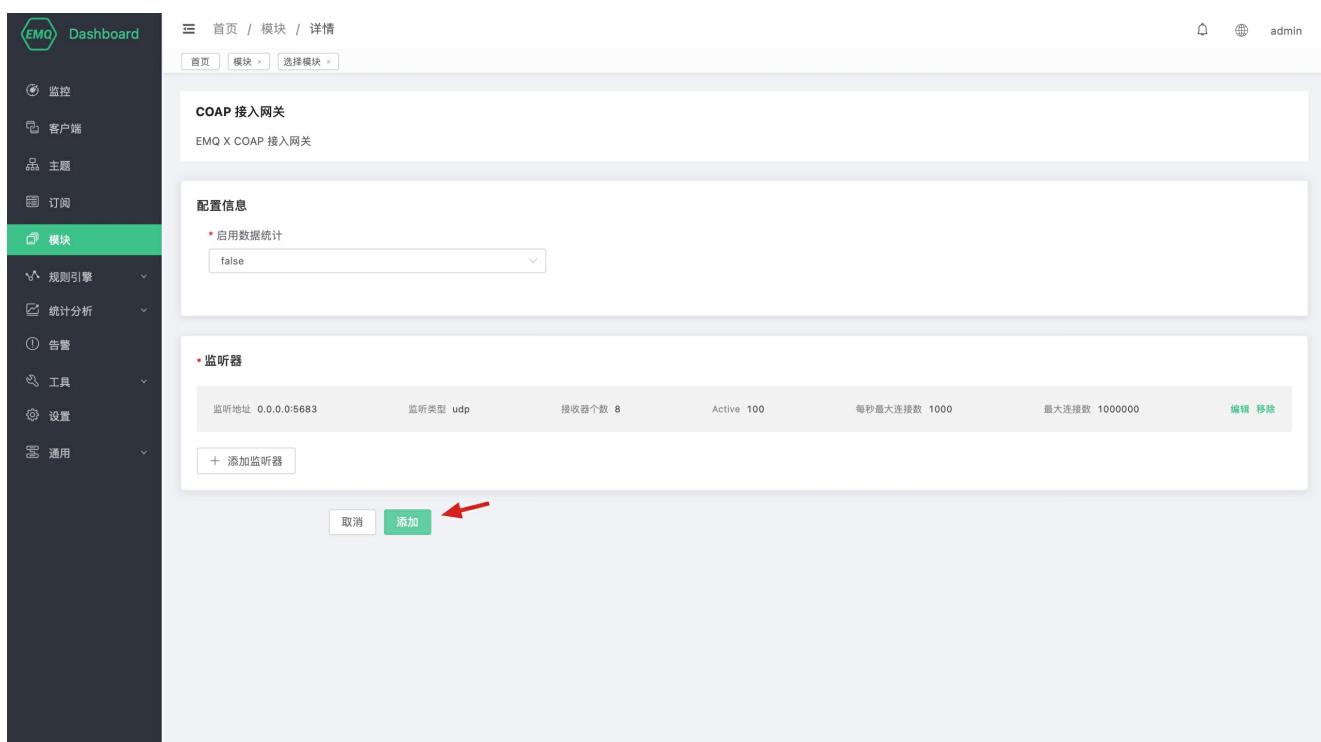
+ 添加监听器

取消 添加

配置监听参数：



点击“确定”完成监听器的配置，然后点击“添加”完成模块的创建：



使用示例

客户端

[libcoap](#) 是一个非常易用的 CoAP 客户端库，此处我们使用它作为 CoAP 客户端来测试 EMQX CoAP 接入网关的功能。

```

1 git clone http://github.com/obgm/libcoap
2 cd libcoap
3 ./autogen.sh
4 ./configure --enable-documentation=no --enable-tests=no
5 make

```

PUBLISH 示例

使用 `libcoap` 发布一条消息：

```

1 libcoap/examples/coap-client -m put -e 1234 "coap://127.0.0.1/mqtt/topic1?c=client1&u=tom&p=secret"

```

- 主题名称为：**"topic1"** (不是 **"/topic1"**)
- Client ID** 为：**"client1"**
- 用户名为：**"tom"**
- 密码为：**"secret"**
- Payload** 为：**"1234"**

SUBSCRIBE 示例

使用 `libcoap` 订阅一个主题：

```

1 libcoap/examples/coap-client -m get -s 10 "coap://127.0.0.1/mqtt/topic1?c=client1&u=tom&p=secret"

```

- 主题名称为：**"topic1"** (不是 **"/topic1"**)
- Client ID** 为：**"client1"**
- 用户名为：**"tom"**
- 密码为：**"secret"**
- 订阅的持续时间为：**10** 秒

在这个期间，如果 `topic1` 主题上有消息产生，`libcoap` 便会收到该条消息。

通信接口说明

CoAP Client Observe Operation

在 **EMQX CoAP** 接入网关中，可以使用 **CoAP** 的 **Observe** 操作实现一个订阅主题的操作：

```

1 GET coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password} with OBSERVE=0

```

- 路径中的 **"mqtt"** 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项

- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误
- 订阅的 **QoS** 等级恒定为 1

CoAP Client Unobserve Operation

使用 **Unobserve** 操作，取消订阅主题：

```
1 GET coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password} with OBSERVE=1
```

- 路径中的 "mqtt" 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项
- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误

CoAP Client Notification Operation

接入网关会将订阅主题上收到的消息，以 **observe-notification** 的方式投递到 **CoAP** 客户端：

- 它的 **payload** 正是 **MQTT** 消息中的的 **payload**
- payload** 数据类型为 "**application/octet-stream**"

CoAP Client Publish Operation

使用 **CoAP** 的 **PUT** 命令执行一次 **PUBLISH** 操作：

```
1 PUT coap://localhost/mqtt/{topicname}?c={clientid}&u={username}&p={password}
```

- 路径中的 "mqtt" 为必填项
- 将 **{topicname}**、**{clientid}**、**{username}** 和 **{password}** 替换为你的真实值
- {topicname}** 和 **{clientid}** 为必填项
- 如果 **clientid** 不存在，将返回 "bad_request"
- URI** 中的 **{topicname}** 应该用 **percent-encoded**，以防止特殊字符，如 + 和 #
- {username}** 和 **{password}** 是可选的
- 如果 **{username}** 和 **{password}** 不正确，将返回一个 **uauthorized** 错误
- payload** 可以是任何二进制数据
- payload** 数据类型为 "**application/octet-stream**"
- 发布信息将以 **qos0** 发送

CoAP Client 保活

设备应定期发出 **GET** 命令，作为 **ping** 操作保持会话在线

1	GET coap://localhost/mqtt/{any_topicname}?c={clientid}&u={username}&p={password}
---	---

- 路径中的 "mqtt" 为必填项
- 将 {topicname}、{clientid}、{username} 和 {password} 替换为你的真实值
- {topicname} 和 {clientid} 为必填项
- 如果 clientid 不存在, 将返回 "bad_request"
- URI 中的 {topicname} 应该用 percent-encoded, 以防止特殊字符, 如 + 和 #
- {username} 和 {password} 是可选的
- 如果 {username} 和 {password} 不正确, 将返回一个 unauthorized 错误
- 客户端应该定期做 keepalive 工作, 以保持会话在线, 尤其是在 NAT 网络中的设备

备注

CoAP 接入网关不支持 POST 和 DELETE 方法。

在 URI 中的主题名称必须先经过 URI 编码处理(参考: [RFC 7252 - section 6.4](#))

CoAP URI 中的 ClientId, Username, Password, Topic 是 MQTT 中的概念。也就是说, CoAP 接入网关是通过借用 MQTT 中的名词概念, 试图将 CoAP 信息融入到 MQTT 系统中。

EMQX 的 认证, 访问控制, 钩子等功能也适用于 CoAP 接入网关。比如:

- 如果 用户名/密码 没有被授权, CoAP 客户端就会得到一个 unauthorized 的错误
- 如果 用户名/客户端ID 不允许发布特定的主题, CoAP 消息实际上会被丢弃, 尽管 CoAP 客户端会从接入网关上得到一个 Acknowledgement
- 如果一个 CoAP 消息被发布, 'message.publish' 钩子也能够捕获这个消息

Well-known locations

CoAP 接入网关的 well-known 发现恒定的返回 ","

例如:

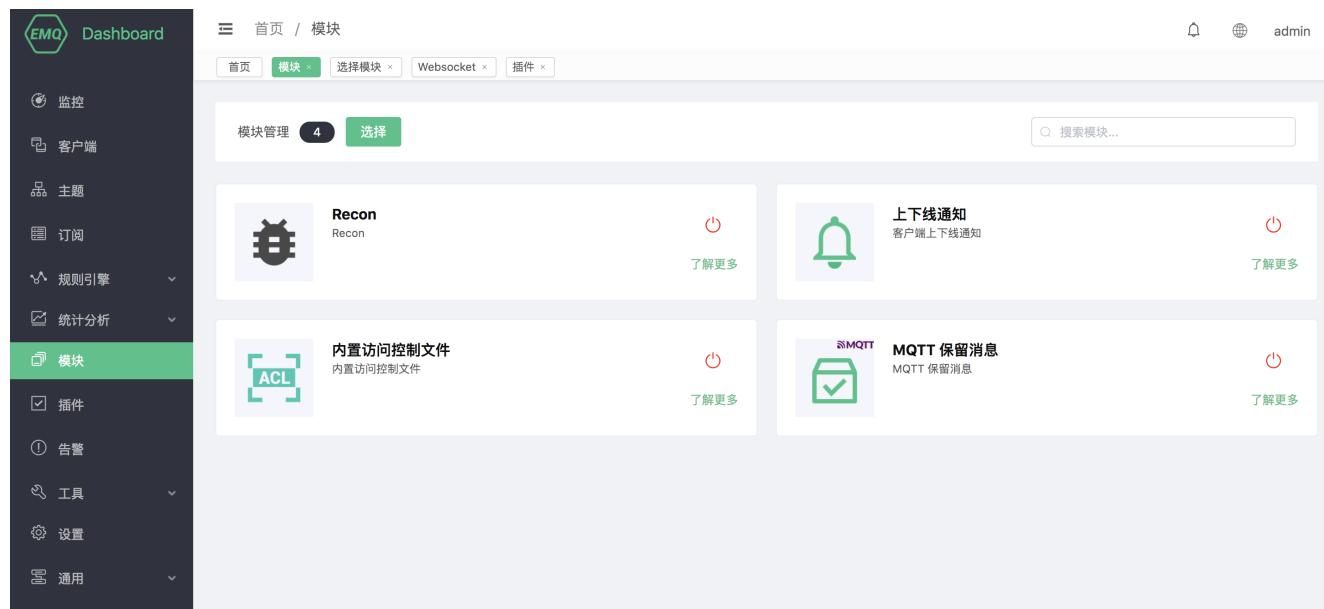
1	libcoap/examples/coap-client -m get "coap://127.0.0.1/.well-known/core"
---	---

Stomp 协议网关

Stomp 协议网关为 EMQX 提供了 Stomp 协议的接入能力。

创建模块

打开 [EMQX Dashboard](#)，点击左侧的“模块”选项卡，选择添加：



The screenshot shows the EMQX Dashboard interface. The left sidebar has a dark theme with various options like Monitoring, Clients, Topics, Subscriptions, Rules Engine, Statistics, Modules (which is selected and highlighted in green), Plugins, Alarms, Tools, Settings, and General. The main content area is titled '模块管理' (Module Management) and shows four modules listed: 'Recon' (Recon), '上下线通知' (Online/Offline Notification), '内置访问控制文件' (Built-in Access Control File), and 'MQTT 保留消息' (MQTT Retained Message). Each module card includes a status icon (red power symbol), a '了解更多' (More Information) button, and a search bar at the top right.

点击“选择”，然后选择“**Stomp** 接入网关”：

协议接入

- CoAP 接入网关 (选择了解更多)
- Stomp 接入网关** (选择了解更多) (highlighted)
- LwM2M 接入网关 (选择了解更多)
- MQTT-SN 接入网关 (选择了解更多)
- TCP 接入网关 (选择了解更多)
- JT/T808 接入网关 (选择了解更多)

消息下发

- Kafka 消费组 (选择了解更多)
- MQTT Subscriber (选择了解更多)
- Pulsar 消费组 (选择了解更多)

多语言扩展

- 多语言扩展协议接入 (选择了解更多)
- 多语言扩展钩子 (选择了解更多)

配置相关基础参数：

Stomp 接入网关

配置信息

- 最大报文头部数量: 10
- 最大报文长度: 1024
- 报文体最大长度: 8192
- 默认密码: guest
- 允许匿名登录: true
- 默认用户名: guest

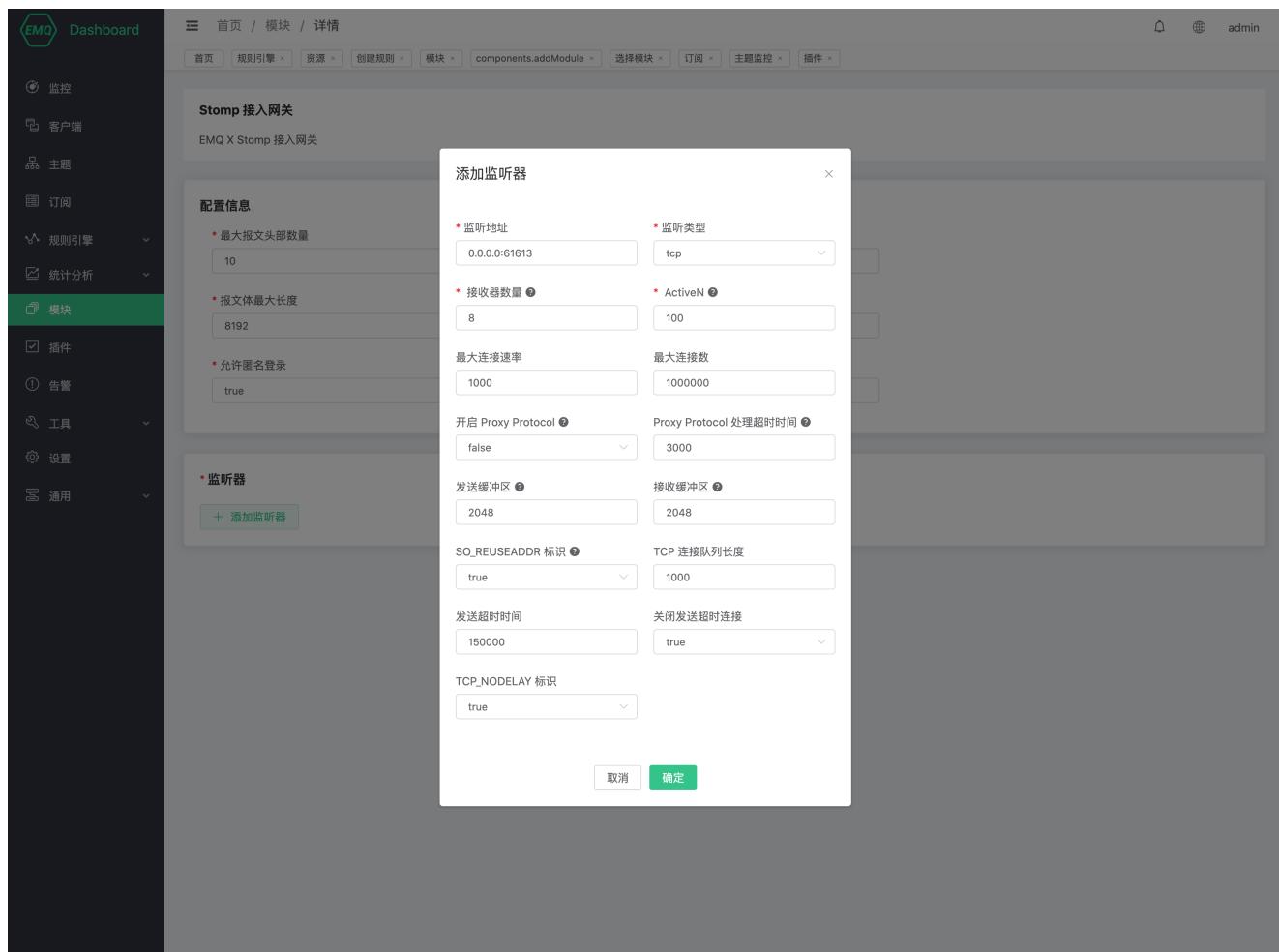
监听器

+ 添加监听器

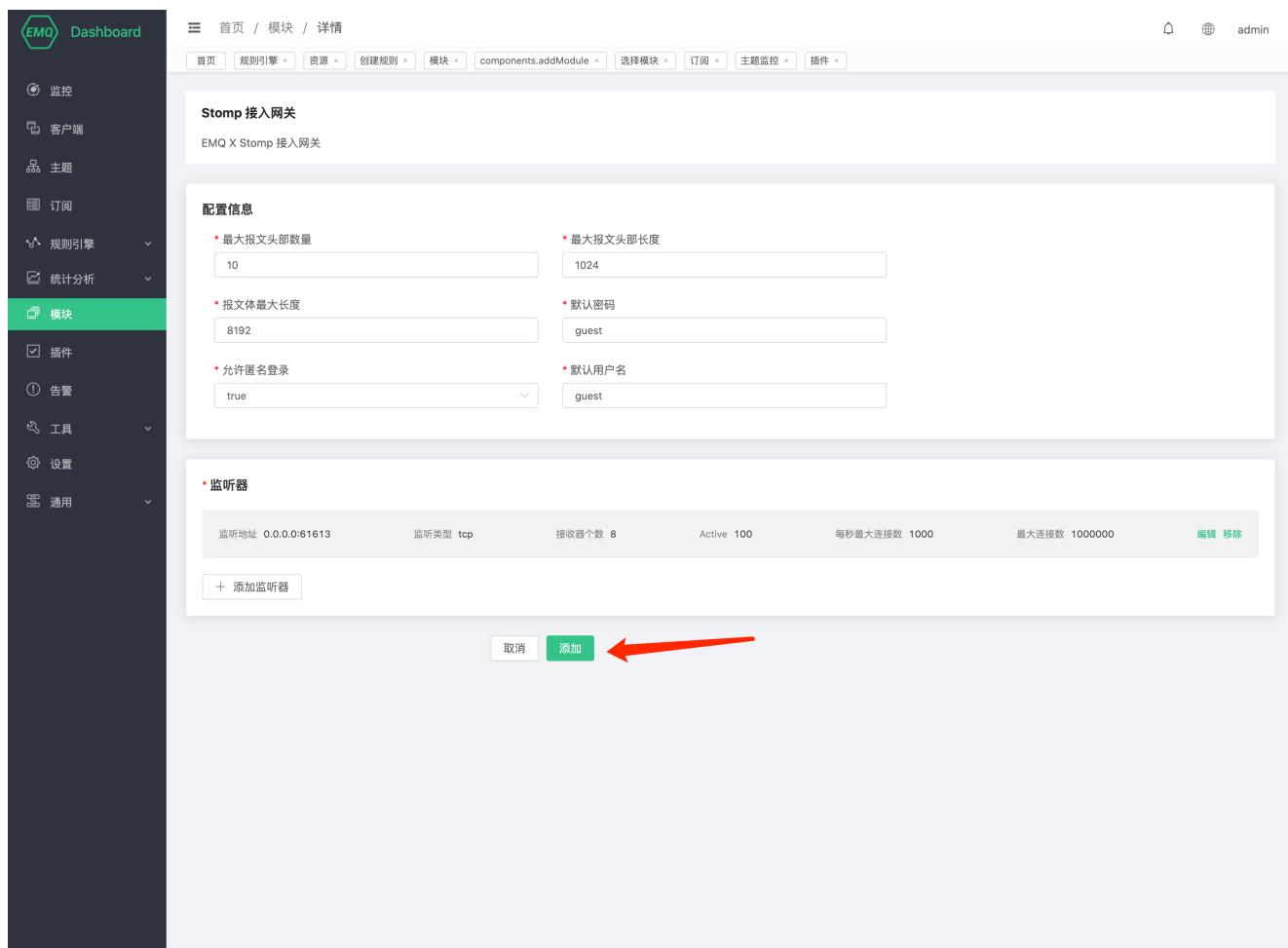
添加监听端口：

The screenshot shows the EMQX Enterprise V4.4 Docs interface. On the left is a dark sidebar with a green header 'Dashboard' and a navigation menu including '监控', '客户端', '主题', '订阅', '规则引擎', '统计分析', '模块' (which is highlighted in green), '插件', '告警', '工具', '设置', and '通用'. The main content area has a title 'Stomp 接入网关' and a sub-section 'EMQ X Stomp 接入网关'. Below this is a '配置信息' section with several input fields: '最大报文头部数量' (10), '最大报文头部长度' (1024), '报文体最大长度' (8192), '默认密码' (guest), '允许匿名登录' (true), and '默认用户名' (guest). At the bottom of this section is a '监听器' section with a button '+ 添加监听器' (highlighted with a red arrow) and a '添加' button. The top right corner shows a user 'admin'.

配置监听参数：



点击“确定”完成监听器的配置，然后点击“添加”完成模块的创建：



配置参数

配置项	说明
最大报文头部数量	Stomp frame headers 的最大数量
最大报文头部长度	Stomp frame headers 的最大长度
报文体最大长度	Stomp frame body 的最大长度
允许匿名登录	是否允许匿名登录
默认用户名	指定 Stomp 模块登录使用的 Username
默认密码	指定 Stomp 模块登录使用的 Password

版本发布

4.4.4 版本

发布日期: 2022-06-01

功能增强

- 为规则引擎 **SQL** 增加更多的时间转换函数
- 为规则引擎 **SQL** 增加 `float2str/2` 函数，支持指定浮点输出精度
- 规则引擎支持消息持久化到 **Alibaba TableStore**
- 规则引擎支持使用 **Basic** 和 **JWT** 认证连接 **Pulsar**
- 规则引擎 **Oracle** 资源新增 `service_name` 选项以支持 **Oracle Database RAC**
- 支持将 **JWT** 用于鉴权，现在 **MQTT** 客户端可以使用包含发布订阅白名单的特定声明进行授权
- 改进认证相关指标使更易理解，现在 `client.authenticate = client.auth.success + client.auth.failure`
- 支持将 **REST API** 的监听器绑定到指定的网络接口上
- 上传 **License** 将自动同步至整个集群，无需每个节点单独上传，提供 **HTTP API**
- 支持对使用内置数据库作为数据源的认证鉴权中的用户数据进行多条件查询和模糊查询
- 支持将消息队列长度以及丢弃消息数量作为条件查询客户端
- 支持配置日志时间格式以兼容旧版本中的时间格式
- 当 `use_username_as_clientid` 配置为 `true` 且客户端连接时未指定 `username`，现在将拒绝连接并返回 `0x85` 原因码
- App secret** 从部分随机改为完全随机
- 通过 **CLI** 进行备份恢复时，不再要求备份文件必须位于 **EMQX** 数据目录的 `backup` 文件夹下
- 现在不兼容版本之间的热升级将被拒绝
- 允许 **EMQX** 的安装路径中有空格
- 引导脚本将在遇到无效的节点名称时快速失败，并提高错误消息的可读性

错误修复

- 修复使用 **PostgreSQL** 离线消息插件时客户端上线后获取不到消息的问题
- 修复某些情况下规则引擎无法与 **Pulsar** 成功建立 **TLS** 连接的问题
- 修复规则引擎 **SQL** 函数 `hexstr_to_bin/1` 无法处理半字节的问题
- 修复规则引擎资源删除时告警未被清除的问题
- 修复 **Dashboard HTTPS** 监听器的 `verify` 选项未生效的问题
- 修复共享订阅投递 **QoS 1** 消息过程中对端会话关闭导致消息丢失的问题
- 修复日志跟踪功能跟踪大报文时堆大小增长过快而触发连接进程强制关闭策略的问题
- 修复模块禁用时未正确卸载相关钩子导致功能异常的问题
- 修复 **MQTT-SN** 客户端重传 **QoS 2** 消息时会被断开连接的问题
- 修复备份文件中关闭的模块会在恢复备份后自动启用的问题
- 修复消息发布 **API** `api/v4/mqtt/publish` 中用户属性类型错误导致订阅端连接断开的问题
- 修复 **DynamoDB** 驱动未适配 **OTP 24** 导致不可用的问题
- 修复 **PostgreSQL** 驱动未适配 **OTP 24** 导致某些认证算法不可用的问题
- 修复对订阅进行多条件查询时返回结果与查询条件不符的问题
- 修复规则引擎资源连接测试不工作的问题

- 修复多项 **Dashboard** 显示问题

4.4.3 版本

发布日期: 2022-04-18

功能增强

- 编解码现已支持使用 **gRPC** 服务将任意二进制有效负载解码为 **JSON** 数据
- 支持使用 **TLS** 连接到 **Pulsar**
- 规则引擎 **SQL** 新增 `mongo_date` 函数，支持将时间戳保存为 **MongoDB Date** 对象
- 规则引擎支持重置指定规则的统计指标
- 规则引擎新增连接确认和鉴权完成事件
- 规则引擎支持拷贝规则以快速复用
- 规则引擎 **SQL** 支持 **zip**、**gzip** 等压缩和解压缩函数
- 改进规则引擎在解析 **Payload** 失败时的错误提示
- 优化规则引擎部分资源的连接测试
- 支持为 **ExHook** 设置执行优先级
- ExHook** 回调接口新增 `RequestMeta meta` **Protobuf** 字段用于返回 **EMQX** 集群名称
- 为共享订阅添加 `local` 策略，这将优先向消息流入的节点下的共享订阅者发送消息。在某些场景下会提升共享消息调度的效率，尤其是在 **MQTT** 桥接配置为共享订阅时
- 为 **TLS** 新增对 `RSA-PSK-AES256-GCM-SHA384`、`RSA-PSK-AES256-CBC-SHA384`、`RSA-PSK-AES128-GCM-SHA256`、`RSA-PSK-AES128-CBC-SHA256` 四个 **PSK** 加密套件的支持，从默认配置中移除 `PSK-3DES-EDE-CBC-SHA` 和 `PSK-RC4-SHA` 这两个不安全的加密套件
- 打印 **Mnesia** `wait_for_table` 诊断日志
 - 打印 **Mnesia** 内部统计的检查点
 - 打印每个表加载统计的检查点，帮助定位表加载时间长的问题
- 严格模式下禁止订阅为空的主题
- 当 `loaded_modules` 和 `loaded_plugins` 文件不存在时生成默认文件

错误修复

- 修复 **TLS** 配置项 `server_name_indication` 设置为 **disable** 不生效的问题
- 修复 **MongoDB** 驱动潜在的进程泄漏问题
- 修复通过 **CLI** 命令修改的 **Dashboard** 默认用户的密码会在节点离开集群后重置的问题
- 静默 `docker-entrypoint.sh` 中的 **grep** 和 **sed** 命令的运行错误日志
- 修复 **API** 路径包含 **ISO8859-1** 转义字符时，备份文件无法被正确删除和下载
- 修复 **Redis** 驱动在 **DNS** 解析失败等情况下会引发崩溃的问题
- 修复 **MQTT Bridge** 插件仅配置订阅主题但未配置 **QoS** 时无法启动的问题
- 创建规则时如果已经有使用相同 **ID** 的规则存在，现在规则引擎将报错而不是替换已有规则
- 修复 **HTTP** 驱动进程池可能无法删除的问题
- 修复模块参数更改报错后无法再次更新的问题
- 修复 **Dashboard** 中 **GB/T 32960** 接入网关模块部分字段类型错误问题
- 修复 **Kafka**、**Pulsar** 等 **Bridge** 资源的配置无法更新的问题
- 修复启用匿名认证时 **JT/T 808** 客户端认证失败的问题

4.4.2 版本

发布日期: 2022-04-01

重要变更

- 对于 **Docker** 镜像, 配置目录 `/opt/emqx/etc` 已经从 **VOLUME** 列表中删除, 这使用户可以更容易地使用更改后的配置来重建镜像。
- CentOS 7 Erlang** 运行系统在 **OpenSSL-1.1.1n** (之前是 **1.0**) 上重建, 在 **v4.3.13** 之前, 客户端使用某些密码套件时, **EMQX** 将无法成功握手并触发 `malformed_handshake_data` 异常。
- CentOS 8 Erlang** 运行时系统在 **RockyLinux 8** 上重新构建。`centos8` 将继续保留在包名中以保持向后兼容。

功能增强

- 规则引擎桥接数据到 **Pulsar** 新增对 **Pulsar proxy** 的支持。
- 为 **Kafka** 生产者增加 **OOM** 保护。
- 新增命令行接口 `emqx_ctl pem_cache clean`, 允许强制清除 **x509** 证书缓存, 以便在证书文件更新后立即重新加载。
- 重构 **ExProto**, 以便匿名客户端也可以显示在 **Dashboard** 上。
- 桥接中的主题配置项现在可以使用 `${node}` 占位符。
- 严格模式下新增对 **MQTT** 报文中的 **UTF-8** 字符串有效性检查。设置为 `true` 时, 无效的 **UTF-8** 字符串将导致客户端连接断开。
- MQTT-SN** 网关支持会话恢复时主动同步注册主题。
- 将规则引擎浮点型数据的写入精度从为小数点后 **10** 位提升至 **17** 位。
- EMQX** 将在启动时提示如何修改 **Dashboard** 的初始密码。

错误修复

- 修复 **MQTT Subscriber** 模块无法使用双向 **SSL** 连接的问题。
- 修复 **PSKFile** 模块启动失败的问题。
- 修复 **Kafka** 消费组 模块无法处理二进制数据的问题。
- 修复日志追踪功能无法停止的问题。
- 修复规则引擎持久化数据到 **Oracle** 和 **Lindorm** 的动作 (仅限同步操作) 执行失败时无法触发备选动作的问题。
- 修复规则引擎数据持久化到 **Oracle** 失败但成功计数仍然增加的问题。
- 修复部分 **zone** 配置无法清除的问题。
- 修复部分监控告警配置的修改在重启后失效的问题。
- 修复编解码功能在集群环境下不可用的问题。
- 修复集群环境下 **LwM2M** 客户端列表查询 **API** 返回数据错误导致无法访问 **LwM2M** 网关模块管理页面的问题。
- 修复 **JT/T 808** 位置报告报文解析错误的问题。
- 修复 **el8** 安装包在 **Amazon Linux 2022** 上无法启动的问题, 错误内容为 `errno=13 Permission denied` 。
- 修复某些情况下如果连接进程阻塞, 客户端无法重连的问题, 现在等待超过 **15** 秒无响应将强制关闭旧的连接进程。
- 修复规则引擎资源不可用时查询资源请求超时的问题。
- 修复热升级运行失败后再次运行出现 `{error, eexist}` 错误的问题。
- 修复向不存在的主题别名发布消息会导致连接崩溃的问题。
- 修复通过 **HTTP API** 在另一个节点上查询 **lwm2m** 客户端列表时的 **500** 错误。
- 修复主题订阅的 **HTTP API** 在传入非法的 **QoS** 参数时崩溃的问题。
- 修复通过多语言协议扩展功能接入的连接进程异常退出时未释放相关资源导致连接计数不更新的问题。
- 修复 `server_keepalive` 配置项的值会被错误应用于 **MQTT v3.1.1** 客户端的问题。
- 修复 **Stomp** 客户端无法触发 `$event/client_connection` 事件消息的问题。

- 修复 **EMQX** 启动时系统内存告警误激活的问题。
- 修复向 **MQTT-SN** 客户端成功注册主题时没有重传此前因未注册主题而投递失败的消息的问题。
- 修复 `loaded_plugins` 文件中配置了重复的插件时 **EMQX** 启动输出错误日志的问题。
- 修复 **MongoDB** 相关功能在配置不正确时输出过量错误日志的问题。
- 增加对 **Dashboard User** 与 **AppID** 的格式检查，不允许出现 / 等特殊字符。
- 将踢除客户端时返回的 **DISCONNECT** 报文中的原因码更正为 0x98。
- 代理订阅将忽略为空的主题。

4.4.1 版本

发布日期: 2022-02-18

注意: 4.4.1 与 4.3.7 保持同步。

此更改集的比较基础是 4.4.0。

重要变更

- 我们在 4.4.1 中修复了 **License** 总连接数计算的 **Bug**, **License** 将正确地检查集群的总连接数，而非错误地仅检查每个节点上的连接数。请计划升级的用户注意此变化可能导致的客户端达到 **License** 限制而无法连接的可能性。
- 慢订阅功能改进，支持统计消息传输过程中花费的时间，并记录和展示耗时较高的客户端和主题。
- 规则引擎支持 **Lindorm** 数据库
- 支持客户端级别的消息丢弃统计指标
- 优化 **Dashboard** 上在线 **Trace** 日志显示，支持语法高亮

次要变更

- license** 连接数预警，默认连接数达到证书允许的 **80%** 则告警，小于 **75%** 时解除告警。用户也可在 `emqx.conf` 中进行自定义：`license.connection_high_watermark_alarm`，
`license.connection_low_watermark_alarm`
- license** 过期预警，当有效期小于 **30** 天时，会告警提示
- 规则引擎支持为客户端消息异常丢失事件配置规则与动作，以增强用户在这一场景的自定义处理能力
- 改进规则引擎 **SQL** 匹配执行过程中的相关统计指标
- 客户端模糊搜索支持 *，(,) 等特殊字符
- 改进 **ACL** 相关统计指标，解决命中 **ACL** 缓存导致计数不增加的问题
- Webhook** 事件通知中新增 `connected_at` 字段
- 在因持有锁太久而终止客户端状态之前记录客户端状态

问题修复

- 修复数据导入导出在某些情况下不可用的问题
- Module** 更新机制改进，解决更新失败后 **Module** 不可用的问题
- 修复规则引擎在执行比较大小的语句时候未进行类型检查的问题
- 修复更新规则引擎动作后相关计数清零的问题
- 修复 **Metrics** 接口默认情况下不返回 `client.acl.deny` 等认证鉴权指标的问题
- 修复订阅查询接口未返回分页数据的问题
- 修复 **STOMP** 处理 **TCP** 粘包时解析失败的问题
- 修复客户端过滤查询时会话创建时间选项不可用的问题

- 修复重启后内存告警可能不会触发的问题
- 修复 `emqx_auth_mnesia` 插件中存在用户数据时导入数据崩溃的问题

4.4.0 版本

发布日期: 2021-12-21

EMQX 4.4.0 现已正式发布，主要包含以下改动:

重要变更

- 从 **4.4** 开始，**EMQX** 的发行包命名将包含 **Erlang/OTP** 的版本号，例如 `emqx-ee-4.4.0-otp24.1.5-3-centos7-arm64.rpm`
- 对于 **Debian/Ubuntu** 用户，**Debian/Ubuntu** 包 (**deb**) 安装的 **EMQX** 现在可以在 **systemd** 上运行，这是为了利用 **systemd** 的监督功能来确保 **EMQX** 服务在崩溃后重新启动。包安装服务从 **init.d** 升级到 **systemd** 已经过验证，但仍建议您在部署到生产环境之前再次验证确认，至少确保 **systemd** 在您的系统中可用
- 规则引擎 **InfluxDB** 集成新增对 **InfluxDB v2 API** 的支持，规则引擎现已支持 **InfluxDB 2.0** 与 **InfluxDB Cloud**
- 规则引擎新增对 **SAP Event Mesh** 的支持
- 规则引擎新增对超融合时空数据库 **MatrixDB** 的支持
- MongoDB** 集成支持 **DNS SRV** 和 **TXT Records** 解析，可以与 **MongoDB Altas** 无缝对接
- 新增在线 **Trace** 功能，用户可以在 **Dashboard** 上完成对客户端和主题的追踪操作，以及查看或下载追踪日志
- 新增慢订阅统计功能，用以及时发现生产环境中消息堵塞等异常情况
- 支持动态修改 **MQTT Keep Alive** 以适应不同能耗策略
- 集群从 **4.3** 到 **4.4** 支持滚动升级。详情请见升级指南。
- 节点间的 **RPC** 链接支持配置 **TLS**。详情请见[集群文档](#)

次要变更

- Dashboard** 支持查看客户端活跃连接数
- Dashboard** 支持相对路径和自定义访问路径
- Dashboard** 移除选项卡导航
- 支持配置是否将整型数据以浮点类型写入 **InfluxDB**
- 支持配置是否转发为 **Payload** 为空的保留消息，以适应仍在使用 **MQTT v3.1** 的用户，相关配置项为 `retainer.stop_publish_clear_msg`
- 多语言钩子扩展（**exhook**）支持动态取消客户端消息的后续转发
- 规则引擎 **SQL** 支持在 **FROM** 子句中使用单引号，例如： `SELECT * FROM 't/#'`
- 优化内置访问控制文件模块的使用交互

- 将 `max_topic_levels` 配置项的默认值更改为 **128**，以前它没有限制（配置为 **0**），这可能是潜在的 **DoS** 威胁
- 改进了接收到 **Proxy Protocol** 报文但 `proxy_protocol` 配置项未开启时的错误日志内容
- 为网关上报消息添加额外的消息属性。来自 **CoAP**, **LwM2M**, **Stomp**, **ExProto** 等网关的消息，在转换为 **EMQX** 的消息时，添加例如协议名称，协议版本，用户名，客户端 IP 等字段，可用于多语言钩子扩展
- HTTP** 性能优化
- 将 **openssl-1.1** 添加到 **RPM** 依赖

问题修复

- 修复节点间 **RPC** 调用堵塞导致客户端进程失去响应的问题
- 修复锁管理进程 `ekka_locker` 在杀死挂起的锁持有者后 **Crash** 的问题
- 修复 **RocketMQ** 异步写入时数据乱码问题
- 修复 **RocketMQ** 统计指标不准的问题
- 修复集群节点数量超过七个时 **Dashboard** 监控页面的显示错误
- 修复规则引擎保存数据到 **MySQL** 时可能出现较高失败率的问题
- 修复规则引擎 **Clickhouse** 离线消息功能不可用的问题
- 修复规则引擎 **MongoDB** 离线消息功能中 **Max Returned Count** 选项无法使用的问题
- 修复规则引擎 **WebHook Action** 中的 **Path** 参数无法使用规则引擎变量的问题
- 修复 **MongoDB** 认证模块无法使用 **Replica Set** 模式等问题
- 修复集群间消息转发失序问题，相关配置项为 `rpc.tcp_client_num`
- 修复内存占用计算错误的问题
- 修复部分热配置失效的问题
- 修复远程主机无法访问时的 **MQTT Bridge** 故障问题（连接挂起）
- 修复 **HTTP Headers** 可能重复的问题

生命周期 EOL

概要

自版本发布之日起，**EMQX** 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。

版本类型

- 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。

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 **EMQX** 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 **EMQX 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 **EMQX 3.x.x** 的维护将持续到 **EMQX 5.0.0** 正式版本发布。**EMQX 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。

维护表

以下表格内容基于上述政策。不过偶尔会进行调整，如我们在一个新的主要版本发布之后发布了一个新的次要版本，那样的话，下面的表格将会更新，然后实施上述的政策。

版本	发布日期	EOL 日期	维护至
3.4.x	2019-12-02	2021-06-30	5.0.0
4.0.x	2020-01-18	2021-07-17	4.1.0
4.1.x	2020-07-18	2022-01-17	4.2.0
4.2.x	2020-10-13	2022-04-12	4.3.0
4.3.x	2021-05-19	2022-11-18	6.0.0
4.4.x	2021-12-21	2023-06-20	6.0.0

从 4.3 升级到 4.4 版本

4.3 和 **4.4** 的节点可以运行在同一集群中，可以采用滚动升级从 **4.3** 升级到 **4.4** 的最新版本。注意是**4.4**的最新版本比如：`connection_low_watermark_alarm` 配置为 **e4.3.7 e4.4.1** 新增加了字段。如果从 **e4.3.7** 升级到 **e4.4.0**时，就会无法识别，需要升级到 **e4.4.1**，所以推荐升级到 **4.4** 的最新版本。

推荐按以下步骤完成滚动升级：

1. 可选步骤：把准备升级的节点从负载均衡器中移除
2. 停止节点 (e.g. `emqx stop`，或者 `systemctl stop emqx`)
3. 在 **4.3** 节点上备份**data**及**etc**目录（见下文）。
4. 卸载 **4.3** 版本。
5. 安装 **4.4** 版本，并把备份数据**data**及配置**etc**覆盖到相应位置。
6. 启动 **4.4** 版本，检查系统是否正常
7. 可选步骤：把流量导入到此节点。

数据及配置备份

在升级前，应备在每个**4.3**版本的节点上对 `data` `etc` 目录中的数据和配置进行备份。根据不同的安装方式和配置，`data` `etc` 目录可能在一下这些位置

- 环境变量 `EMQX_NODE__DATA_DIR` 指向的位置
- 配置文件中 `node.data_dir` 指向的位置
- 在**docker**中运行的默认位置: `/opt/emqx/data` `opt/emqx/etc` (通常是一个挂在的外部**volume**)
- 直接使用 `zip` 安装包的默认位置: `<install-path>/data` `<install-path>/etc`
- 使用**RPM**或**DEB**安装包安装的默认位置: `/var/lib/emqx/` `/etc/emqx`

以 **RPM** 默认安装为例，你需要

```

1 ## 创建备份文件夹
2 mkdir -p ~/emqx-backup/etc/
3 mkdir -p ~/emqx-backup/data/
4
5 ## 确认本节点emqx已经停止运行
6 systemctl stop emqx
7 systemctl status emqx
8
9 ## 复制备份文件
10 cp -r /etc/emqx ~/emqx-backup/etc/
11 cp -r /var/lib/emqx/ ~/emqx-backup/data/

```

卸载 4.3 版本

以**RPM**默认安装为例：

```

1 ## 查看确认安装的版本是否为需要卸载的
2 rpm -qa | grep emqx
3
4 ## 确认无误后卸载旧版本
5 rpm -e emqx-4.3.x-x.x86_64

```

sh

安装 4.4 版本，并导入4.3版本数据。

- 以**RPM**默认安装为例：

```
1 rpm -ivh emqx-ee-4.4.0-otp24.1.5-3-centos7-amd64.rpm
```

sh

- 导入上步已备份的**4.3**数据及配置。

```

1 cp -r ~/emqx-backup/etc/ /etc/emqx/
2 cp -r ~/emqx-backup/data/ /var/lib/emqx/

```

sh

启动 4.4 版本

如果你使用的是**systemctl**启动：

```

1 systemctl start emqx
2 systemctl status emqx

```

sh

- 通过**emqx_ctl** 查看集群状态

```

1 /usr/bin/emqx_ctl cluster status
2

```

sh

- 查看日志是否有异常

```

1 ## 找到最新的写入日志文件
2 ls -ahl /var/log/emqx/emqx.log.*[0-9] |head -n 1
3 ## 查看此日志：
4 tail -f -n 100 /var/log/emqx/emqx.log.x

```

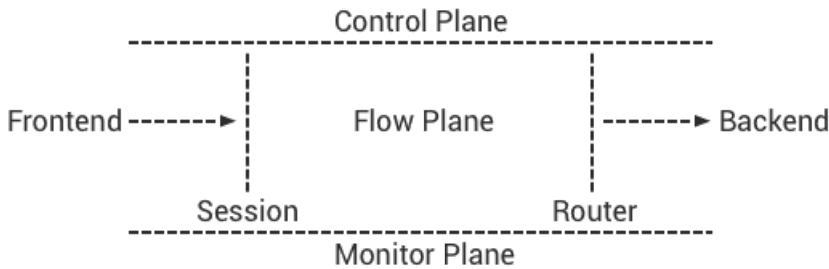
sh

- 通过**dashboard**查看是否有报警，本节点的状态是否正常

架构设计

前言

EMQX 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**)：



100 万连接

多核服务器和现代操作系统内核层面，可以很轻松支持 **100 万 TCP** 连接，核心问题是应用层面如何处理业务瓶颈。

EMQX 在业务和应用层面，解决了单节点承载**100万**连接的各类瓶颈问题。连接测试的操作系统内核、**TCP** 协议栈、**Erlang** 虚拟机参数参见：http://docs.emqtt.cn/zh_CN/latest/tune.html。

全异步架构

EMQX 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。

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



消息持久化

EMQX 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，**EMQX** 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。

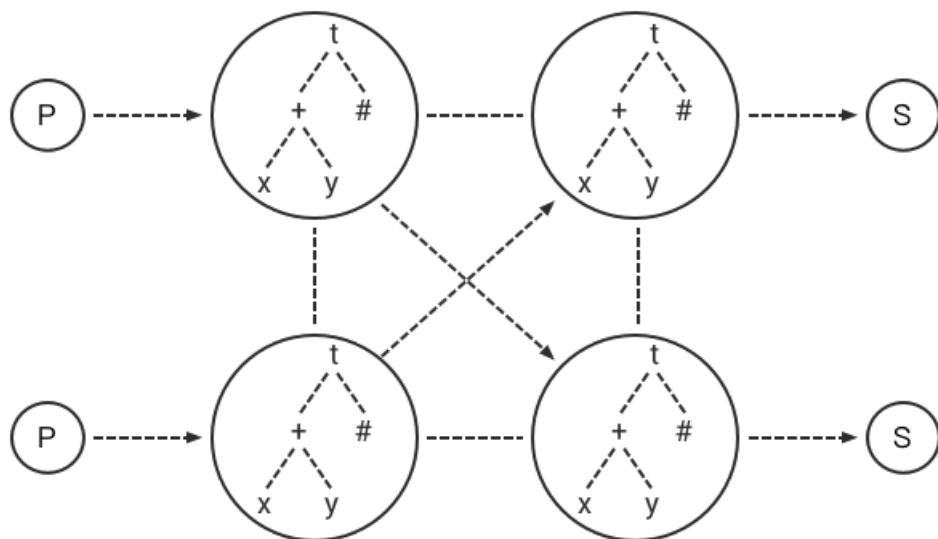
EMQX 在设计上分离消息路由与消息存储职责后，数据复制容灾备份甚至应用集成，可以在数据层面灵活实现。

EMQX 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。

系统架构

概念模型

EMQX 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，**EMQX** 按主题树 (**Topic Trie**) 发布订阅模式在集群节点间路由 **MQTT** 消息：



设计原则

1. **EMQX** 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。

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。

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** 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

会话层设计

会话层处理 MQTT 协议发布订阅 (Publish/Subsribe) 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

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：



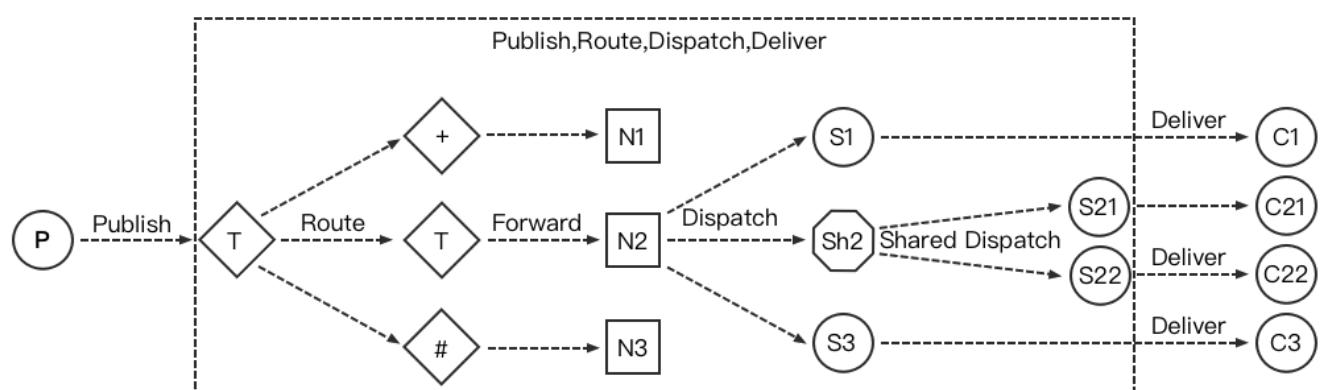
1. 64bits 时间戳: erlang:system_time if Erlang >= R18, otherwise os:timestamp
2. Erlang 节点 ID: 编码为2字节
3. Erlang 进程 PID: 编码为4字节
4. 进程内部序列号: 2字节的进程内部序列号

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：

PktID ←→ Session ←→ MsgID ←→ Router ←→ MsgID ←→ Session ←→ PktID

路由层设计

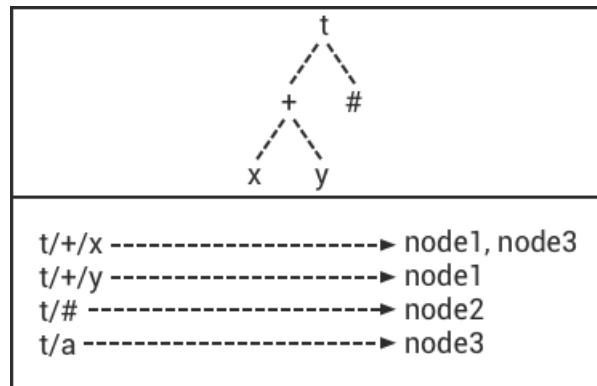
路由层维护订阅者 (Subscriber) 与订阅关系表 (Subscription)，并在本节点发布订阅模式派发 (Dispatch) 消息：



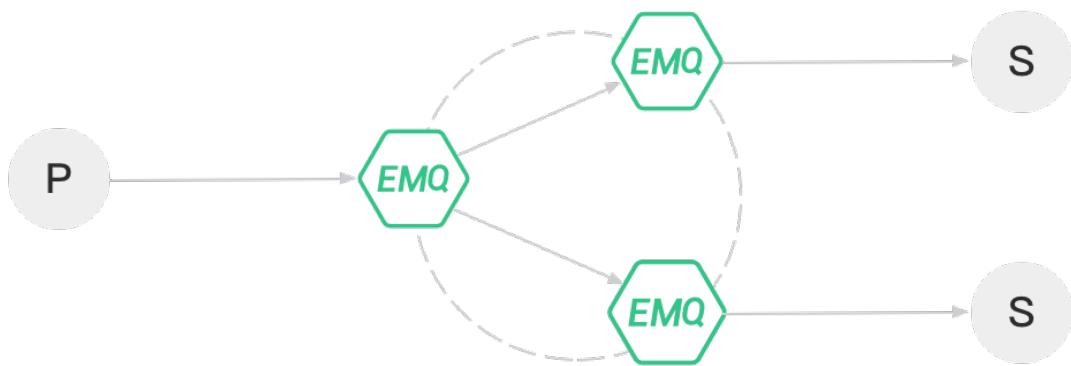
消息派发到会话 (**Session**) 后，由会话负责按不同 **QoS** 送达消息。

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：



Mnesia/ETS 表设计

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

Erlang 设计相关

1. 使用 **Pool, Pool, Pool...** 推荐 **GProc** 库: <https://github.com/uwiger/gproc>
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**

资源



官方资源

- [EMQ 官方网站](#)
- [EMQ 官方博客](#)
- [EMQX GitHub 仓库](#)

社区、讨论、贡献和支持

你可通过以下途径与 **EMQ** 社区及开发者联系：

- [EMQX Slack](#)
- [Twitter](#)
- [Forum](#)
- [Medium](#)
- [Reddit](#)

欢迎你将任何 **bug**、问题和功能请求提交到 [emqx/emqx](#)。

使用 **EMQX** 的项目

以下是提交投稿的使用 **EMQX** 的开源项目，未提交项目/商业项目不在此列：

- [ActorCloud](#) - 开源一站式物联网平台服务

中文教程

EMQX 系列教程及 **Erlang/IoT** 相关博客、视频站点，欢迎留言或 [Issues 投稿](#)。

- 暂无

MQTT 规范

你可以通过以下链接了解与查阅 **MQTT** 协议：

- [MQTT Version 3.1.1](#)
- [MQTT Version 5.0](#)
- [MQTT SN](#)