实验八 MQTT 规则引擎与数据库操作实验

一、实验目的

1. 尝试使用 Docker 安装 EMQX Enterprise 和 MYSQL（可选）

2. 初步掌握 EMQX Enterprise 规则引擎相关内容，学习连接 MySQL 数据库及相应的基本技能。

二、背景知识

本实验主要参考内容:

https://docs.emqx.com/zh/enterprise/v4.4/

https://docs.emqx.com/zh/enterprise/v4.4/rule/rule-engine.html

https://docs.emqx.net/broker/latest/cn/backend/backend.html

1. 使用 Docker 方式安装 EMQX Enterprise 和 MYSQL（可选）

a) 安装 Docker（已 CentOS 为例，其他操作系统可参考线上教程）

curl -sSL https://get.daocloud.io/docker | sh 这里使用国内镜像网站进行安装

```
[root@IOTZUCC ~]# curl -sSL https://get.daocloud.io/docker | sh
# Executing docker install script, commit: 4f282167c425347a931ccfd95cc91fab041d414f
+ sh -c 'yum install -y -q yum-utils'
+ sh -c 'yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo'
添加仓库自： https://download.docker.com/linux/centos/docker-ce.repo
+ '[' stable '!=' stable ']'
+ sh -c 'yum makecache'
CentOS Linux 8 - AppStream                                      9.0 kB/s | 4.3 kB     00:00
CentOS Linux 8 - BaseOS                                          23 kB/s | 3.9 kB     00:00
CentOS Linux 8 - Extras                                          50 kB/s | 1.5 kB     00:00
Extra Packages for Enterprise Linux 8 - x86_64                  122 kB/s | 4.7 kB     00:00
Docker CE Stable - x86_64                                        15 kB/s |  31 kB     00:02
元数据缓存已建立。
+ sh -c 'yum install -y -q docker-ce docker-ce-cli containerd.io docker-scan-plugin docker-compose-plugin docker-ce-rootless-extras'
警告： /var/cache/dnf/docker-ce-stable-fa9dc42ab4cec2f4/packages/containerd.io-1.6.9-3.1.el8.x86_64.rpm: 头V4 RSA/SHA512 Signature, 密钥 ID 621e9f35: NOKEY
导入 GPG 公钥 0x621E9F35:
 Userid: "Docker Release (CE rpm) <docker@docker.com>"
 指纹： 060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35
 来自： https://download.docker.com/linux/centos/gpg

================================================================================

To run Docker as a non-privileged user, consider setting up the
Docker daemon in rootless mode for your user:
```

docker version 查看是否安装成功

```
[root@IOTZUCC ~]# docker version
Client: Docker Engine - Community
 Version:           20.10.21
 API version:       1.41
 Go version:        go1.18.7
 Git commit:        baeda1f
 Built:             Tue Oct 25 18:02:19 2022
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true
```

如果提示 **Cannot connect to the Docker daemon at**

unix:///var/run/docker.sock. Is the docker daemon running?

输入命令 **systemctl start docker** 启动 **docker**

b) 使用 **docker** 安装 **EMQX Enterprise**

docker pull emqx/emqx-ee:4.4.10 拉取 **EMQX Enterprise 4.4.10**

```
[root@IOTZUCC ~]# docker pull emqx/emqx-ee:4.4.10
4.4.10: Pulling from emqx/emqx-ee
3d2430473443: Pull complete
4ccb3f45a415: Pull complete
990cbc048e29: Pull complete
06b4564f2995: Pull complete
a7179ecfc4ba: Pull complete
9d15c5795445: Pull complete
4f4fb700ef54: Pull complete
c914b8f9006b: Pull complete
091f7f5a425a: Pull complete
Digest: sha256:081e4f56c57ffca5c47291a35db59e760f5def6c4b27475a48724ae93a0a6d8d
Status: Downloaded newer image for emqx/emqx-ee:4.4.10
docker.io/emqx/emqx-ee:4.4.10
```

启动容器

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:4.4.10

```
[root@IOTZUCC ~]# 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/em
qx-ee:4.4.10
6662e15d0a5d66f86134db58aaed590f33c425a5d4fd1302599de2a8f402bea8
```

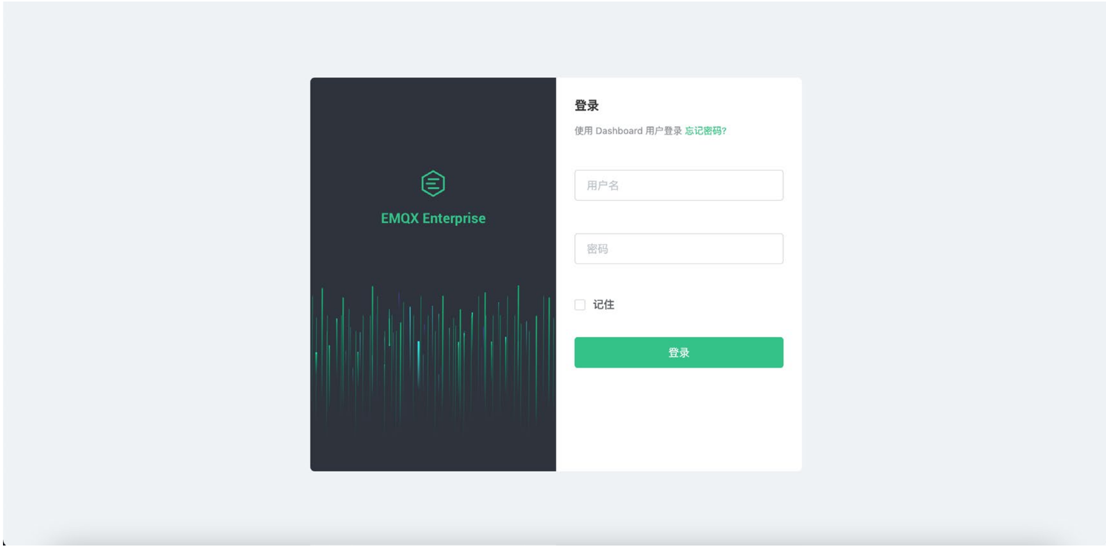
（-P 参数用于进行 docker 端口映射：**-p 1883:1883 -p 8081:8081 -p 8083:8083 -p 8084:8084 -p 8883:8883 -p 18083:18083**）

查看容器是否启动成功，状态是否为 **Up**

docker ps -a

```
[root@IOTZUCC ~]# docker ps -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED          STATUS          PORTS
                                                                                                                      NAMES
6662e15d0a5d   emqx/emqx-ee:4.4.10   "/usr/bin/docker-ent…"   18 seconds ago   Up 17 seconds   4369-4370/tcp, 5369/tcp, 0.0.0.0:1883->
1883/tcp, :::1883->1883/tcp, 0.0.0.0:8081->8081/tcp, :::8081->8081/tcp, 0.0.0.0:8083-8084->8083-8084/tcp, :::8083-8084->8083-8084/tcp,
 6369-6370/tcp, 0.0.0.0:8883->8883/tcp, :::8883->8883/tcp, 0.0.0.0:18083->18083/tcp, :::18083->18083/tcp, 11883/tcp   emqx-ee
```

登录到 机器 **IP:18083** 便可登陆到 **dashboard** 上，初始账号 **admin/public**



c) 使用 docker 安装 MYSQL

拉取最新 mysql 镜像，也可根据需要选择版本

docker pull mysql:latest

```
[root@IOTZUCC ~]# docker pull mysql:latest
latest: Pulling from library/mysql
feec22b5b798: Pull complete
3b33952322b1: Pull complete
8632ee03bb1c: Pull complete
636ccd115361: Pull complete
b07c8fac8eea: Pull complete
e44c54db9c14: Pull complete
cf9c45749101: Pull complete
9f2fa3febc47: Pull complete
44d5e1d3c311: Pull complete
bb3db2c5d8ec: Pull complete
e0ead729abd9: Pull complete
Digest: sha256:717e6f25ed8997b7ecb0408e063c4dcba202a68b341ebac4c4d97f51439b87ee
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest
```

启动 mysql 容器

docker run -itd --name mysql-test -p 3306:3306 -e MYSQL_ROOT_PASSWORD=123456 mysql

其中 MYSQL_ROOT_PASSWORD 为设置 mysql 密码

```
[root@IOTZUCC ~]# docker run -itd --name mysql-test -p 3306:3306 -e MYSQL_ROOT_PASSWORD=123456 mysql
c48cd77dadfc3a68af30df2303955870d640b93bb0fc1d634e3772b27ba7e02f
```

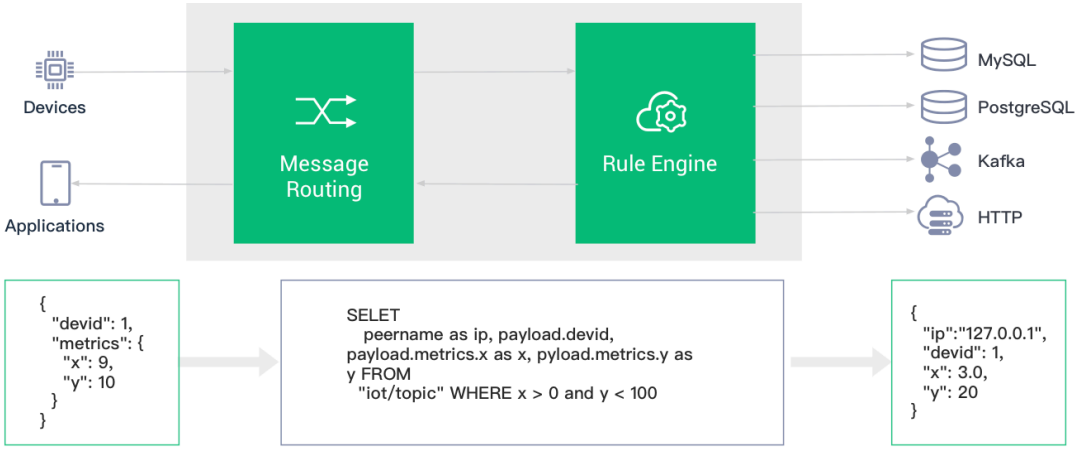
docker ps -a 查看

```
[root@IOTZUCC ~]# docker ps -a
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS
                                                                                                        NAMES
c48cd77dadfc   mysql                  "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp
, 33060/tcp
                                                                                                        mysql-test
6662e15d0a5d   emqx/emqx-ee:4.4.10    "/usr/bin/docker-ent…"   6 minutes ago   Up 6 minutes   4369-4370/tcp, 5369/tcp, 0.0.0.0:1883->18
83/tcp, :::1883->1883/tcp, 0.0.0.0:8081->8081/tcp, :::8081->8081/tcp, 0.0.0.0:8083-8084->8083-8084/tcp, :::8083-8084->8083-8084/tcp, 6
369-6370/tcp, 0.0.0.0:8883->8883/tcp, :::8883->8883/tcp, 0.0.0.0:18083->18083/tcp, :::18083->18083/tcp, 11883/tcp   emqx-ee
```

2. 规则引擎

EMQ X **Rule Engine (简称规则引擎)**

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



规划引擎工作流程图

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

EMQ X 规则引擎快速入门

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

消息发布

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

选择发布到 t/# 主题的消息，并筛选出全部字段：

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

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

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

事件触发

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

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

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

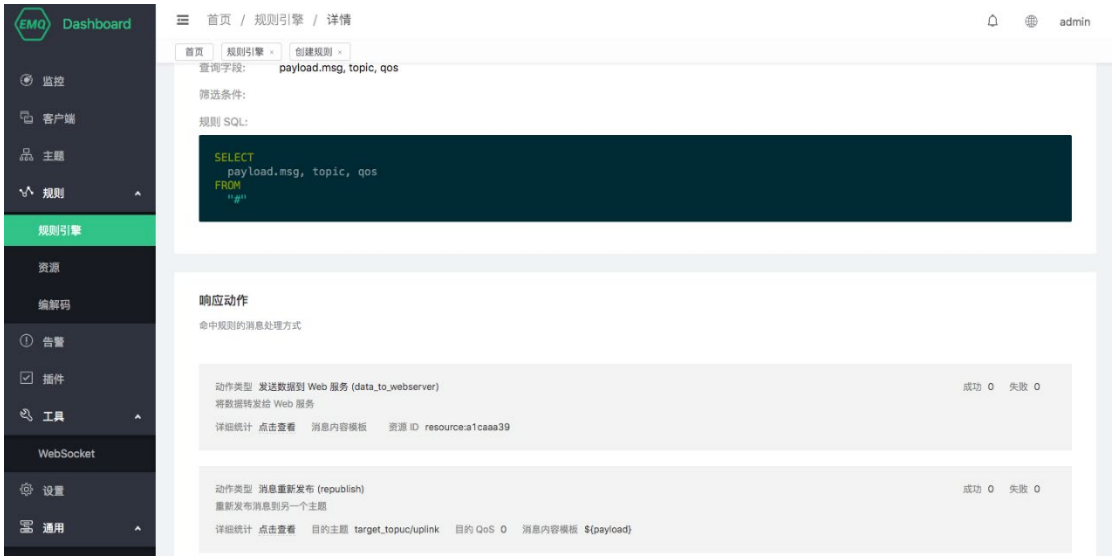
规则引擎数据和 SQL 语句格式，事件主题 列表详细教程参见 SQL 手册。

最小规则

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

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

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



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

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

规则引擎组成

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

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

规则、动作、资源的关系:

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

- 规则(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。

事件和事件主题

规则引擎的 SQL 语句既可以处理消息(消息发布)，也可以处理事件(客户端上下线、客户端订阅等)。对于消息，FROM 子句后面直接跟主题名；对于事件，FROM 子句后面跟事件主题。
事件消息的主题以 `"$events/"` 开头，比如 `"$events/client_connected"`, `"$events/session_subscribed"`。 如果想让 emqx 将事件消息发布出来，可以在 `emqx_rule_engine.conf` 文件中配置。
所有支持的事件及其可用字段详见: 规则事件。

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:

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

- 从 topic 为 "t/a" 或 "t/b" 的消息中提取所有字段:

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

- 从 topic 能够匹配到 't/#' 的消息中提取所有字段。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

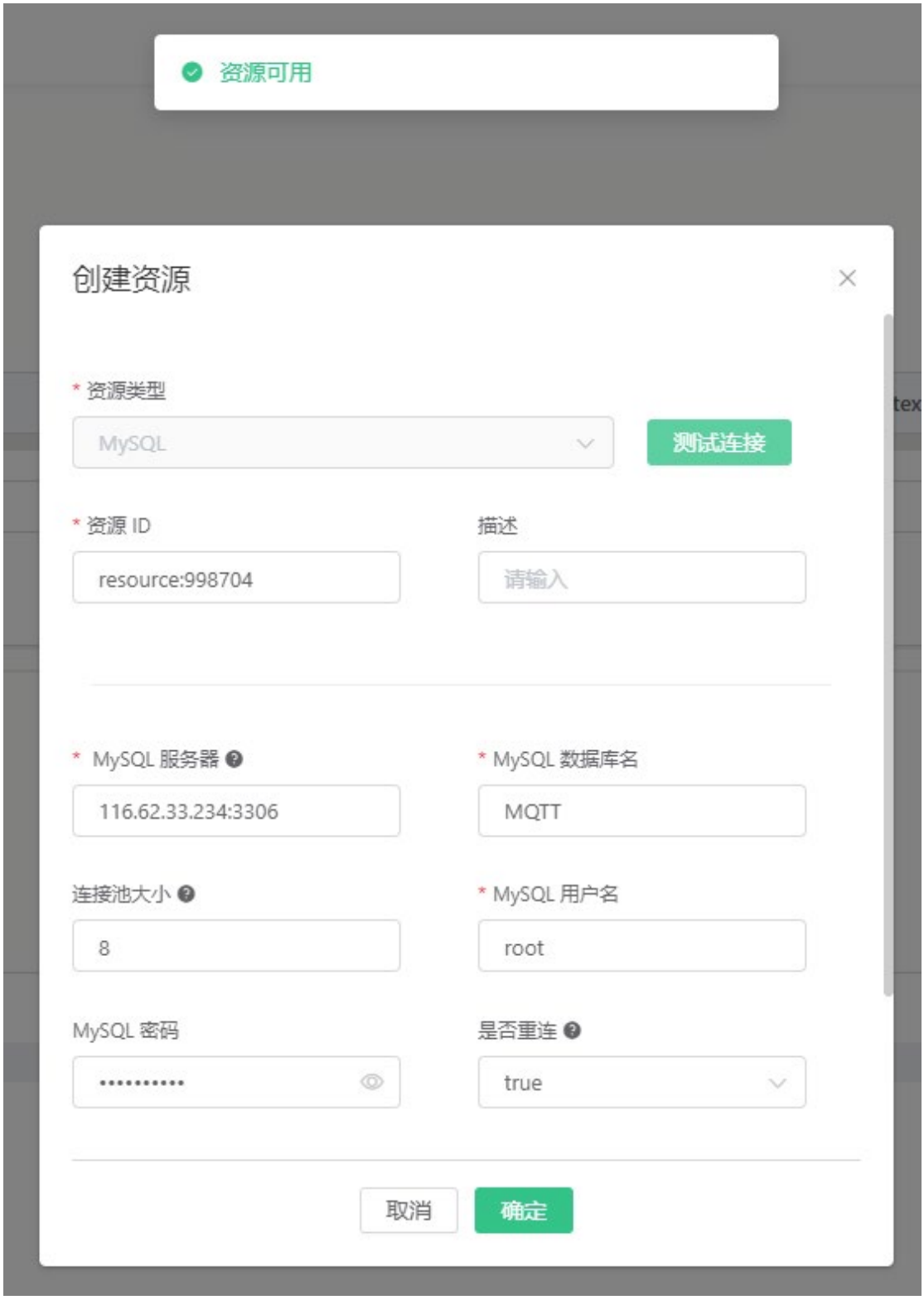
```
SELECT clientid FROM "$events/session_subscribed" WHERE topic =~ 't/#'
and qos = 1
```

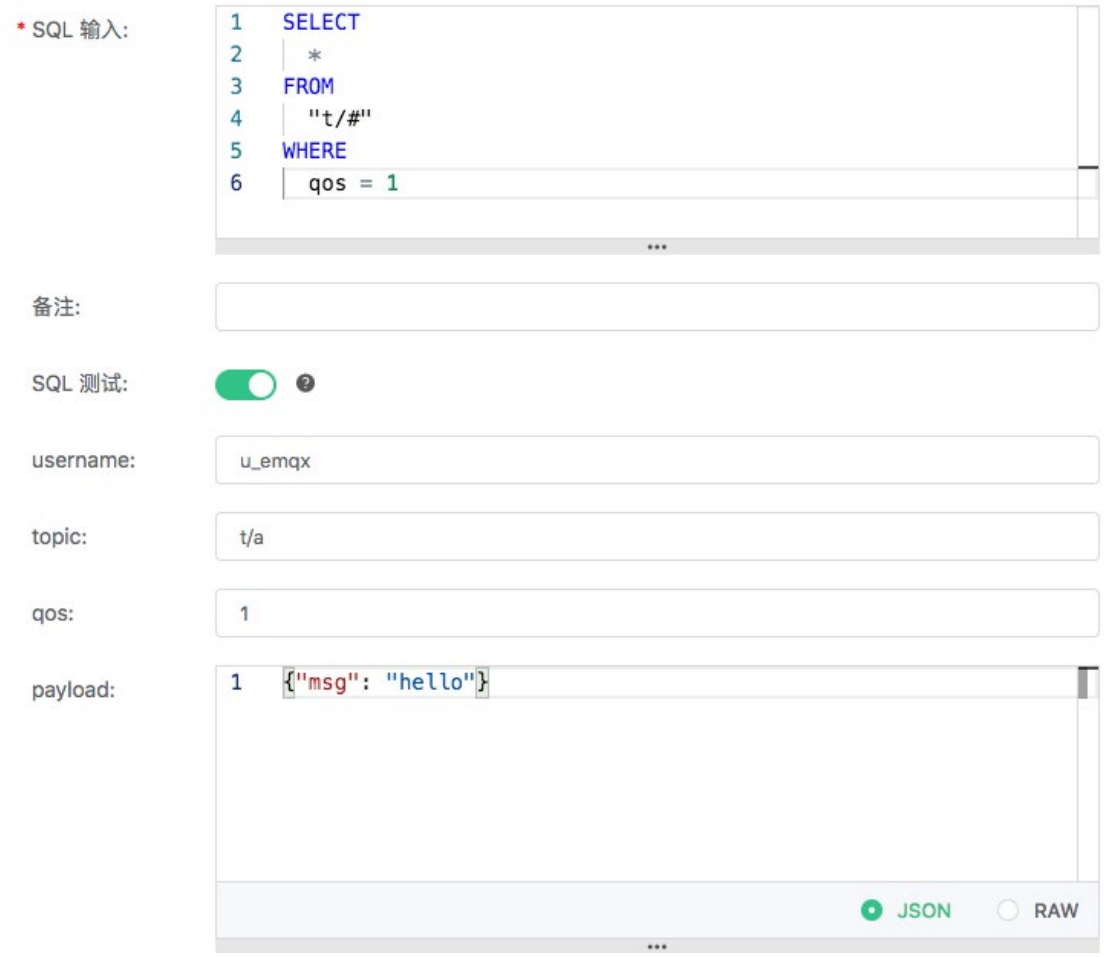
- FROM 子句后面的主题需要用双引号 "" 引起来。
- WHERE 子句后面接筛选条件，如果使用到字符串需要用单引号 '' 引起来。
- FROM 子句里如有多个主题，需要用逗号 "," 分隔。例如 SELECT * FROM "t/1", "t/2" 。
- 可以使用使用 "." 符号对 payload 进行嵌套选择。

在 Dashboard 中测试 SQL 语句

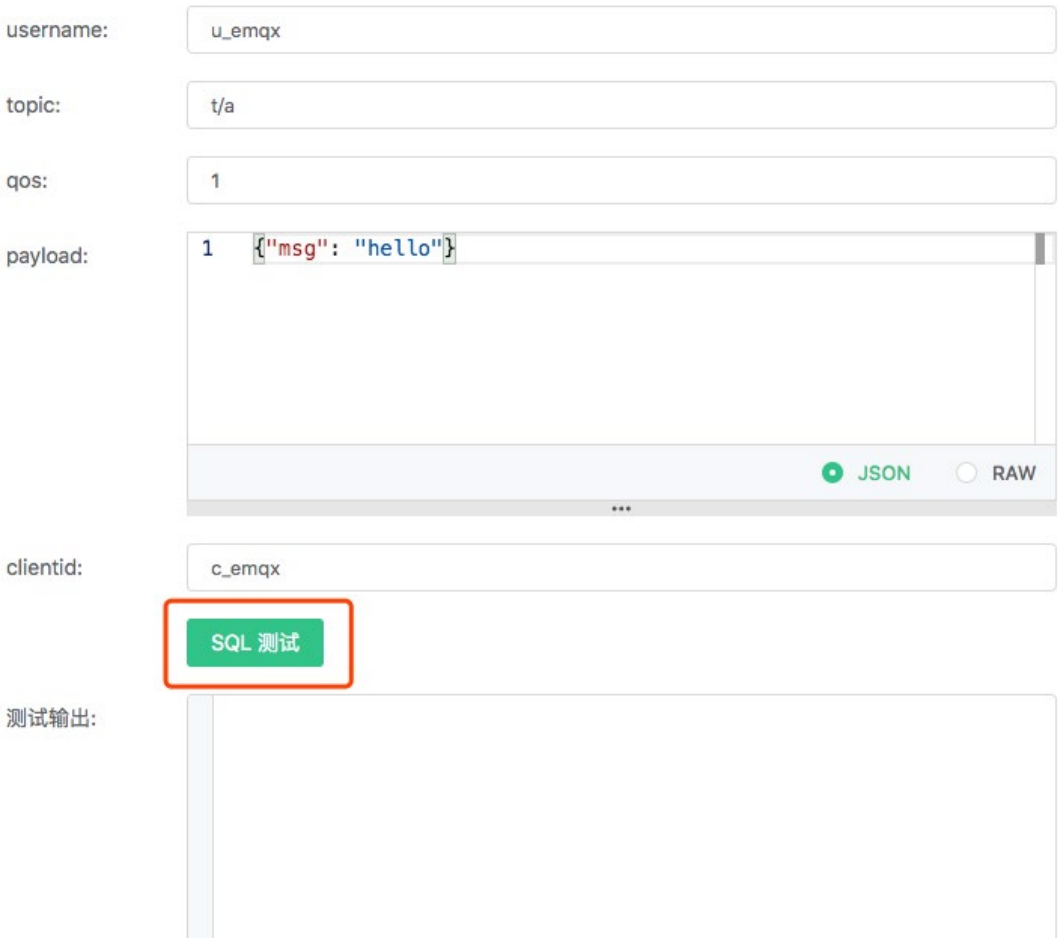
Dashboard 界面提供了 SQL 语句测试功能，通过给定的 SQL 语句和事件参数，展示 SQL 测试结果。

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

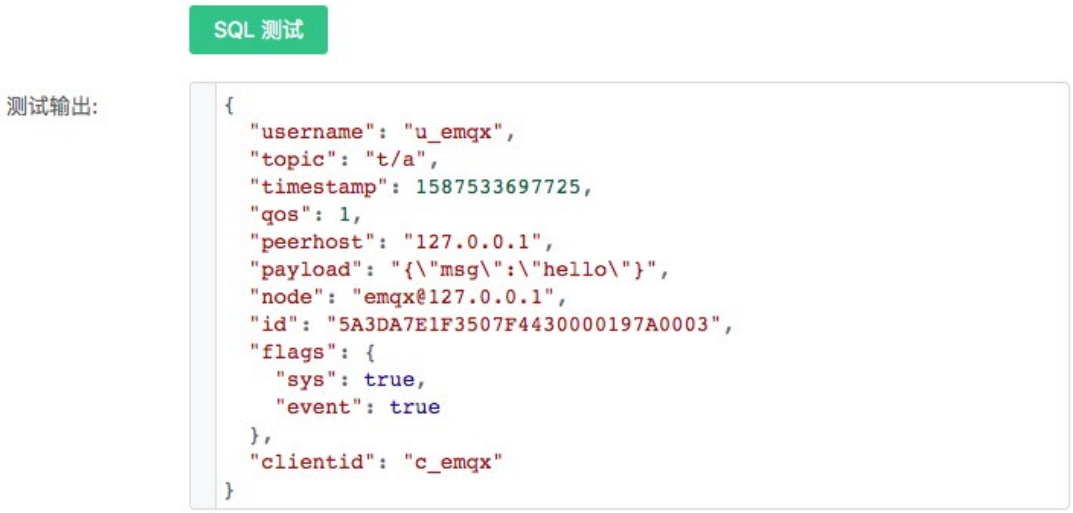




2.2 修改模拟事件的字段，或者使用默认的配置，点击 **测试** 按钮:



2.3 SQL 处理后的结果将在 **测试输出** 文本框里展示:



3. 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:

```
$ brew install mysql

$ brew services start mysql

$ mysql -u root -h localhost -p

ALTER USER 'root'@'localhost' IDENTIFIED BY 'public';
```

初始化 MySQL 表:

```
$ mysql -u root -h localhost -ppublic
```

创建 “test” 数据库:

```
CREATE DATABASE test;
```

创建 t_mqtt_msg 表:

```
USE test;
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;
```

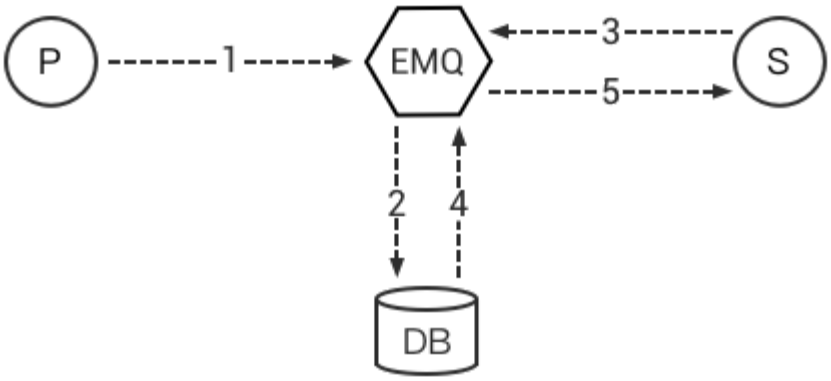
数据存储

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。

数据存储是 EMQ X Enterprise 专属功能。

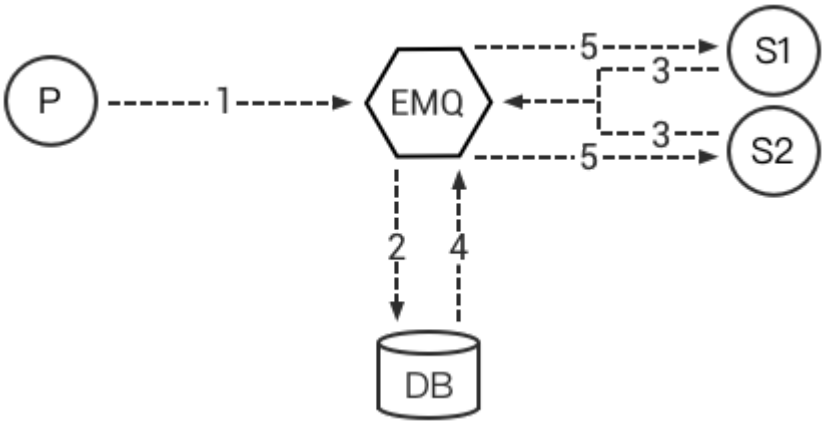
数据存储设计

一对一消息存储



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

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

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

存储插件	配置文件	说明
emqx_backend_redis	emqx_backend_redis.conf	Redis 消息存储
emqx_backend_mysql	emqx_backend_mysql.conf	MySQL 消息存储
emqx_backend_pgsql	emqx_backend_pgsql.conf	PostgreSQL 消息存储
emqx_backend_mongo	emqx_backend_mongo.conf	MongoDB 消息存储
emqx_backend_cassa	emqx_backend_cassa.conf	Cassandra 消息存储
emqx_backend_dynamo	emqx_backend_dynamo.conf	DynamoDB 消息存储
emqx_backend_influxdb	emqx_backend_influxdb.conf	InfluxDB 消息存储
emqx_backend_opentsdb	emqx_backend_opentsdb.conf	OpenTSDB 消息存储

配置步骤

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

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

```
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:

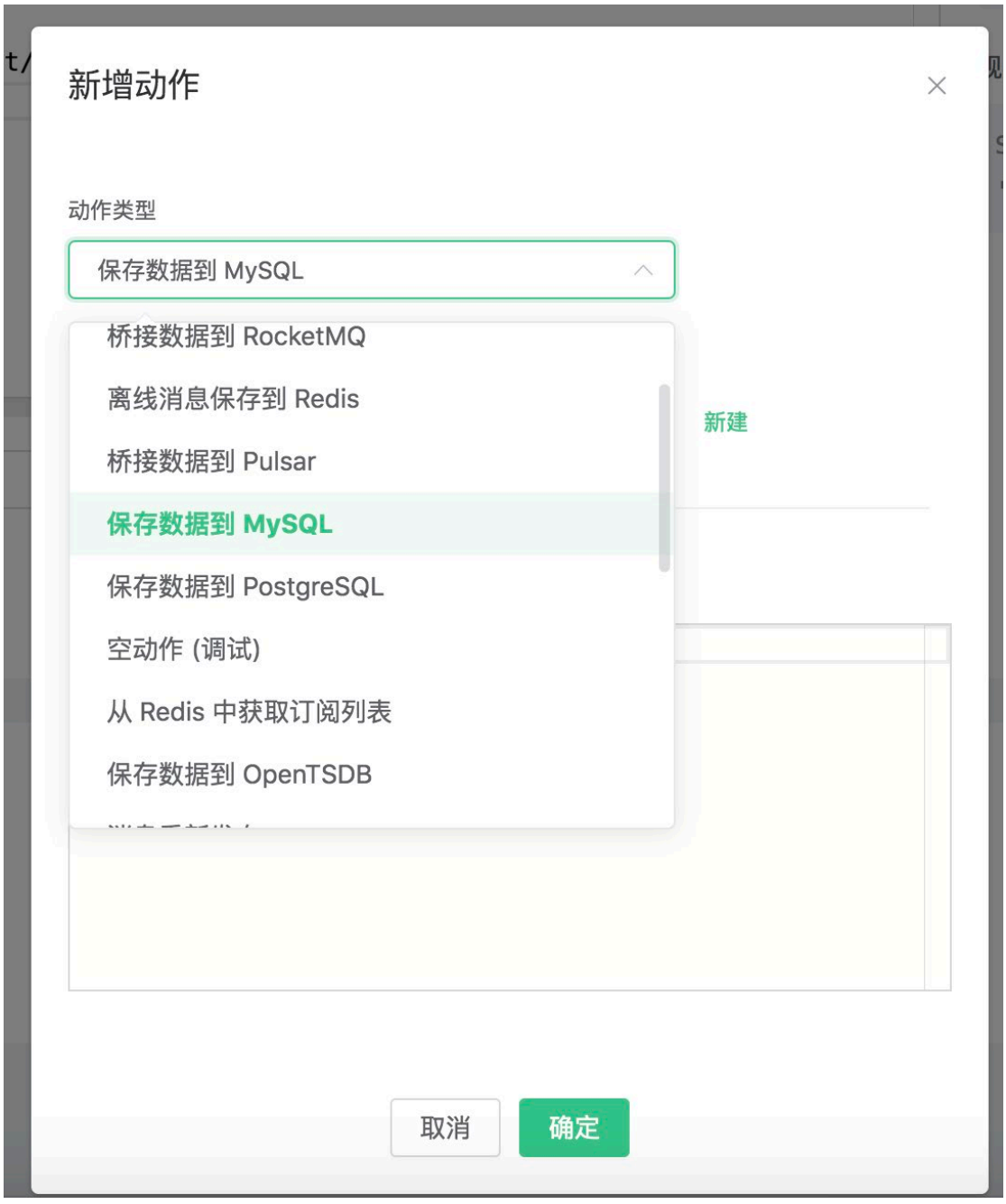
打开 EMQ X Dashboard，选择左侧的“规则”选项卡。

填写规则 SQL:

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

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”。

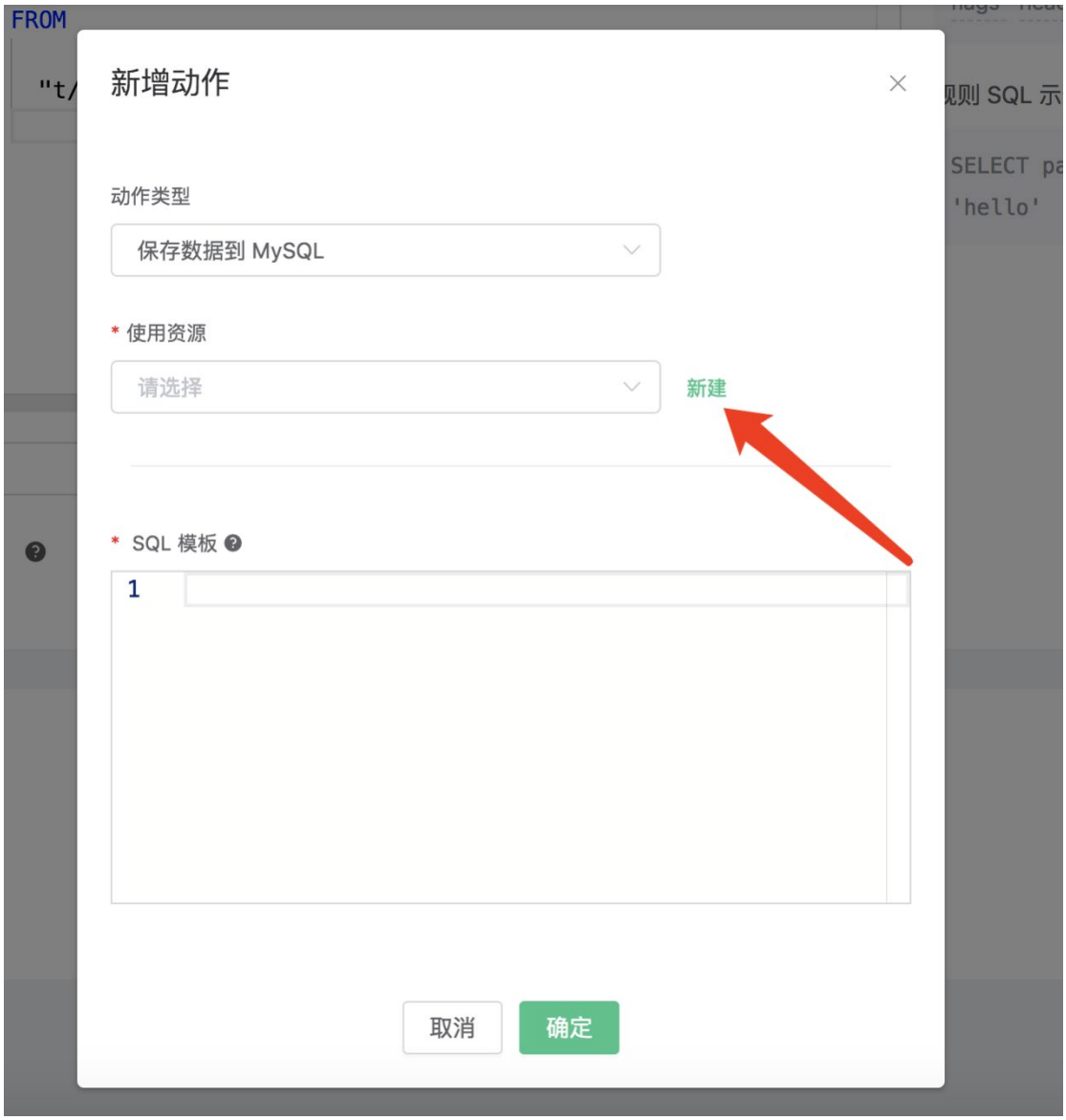


填写动作参数:

“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:

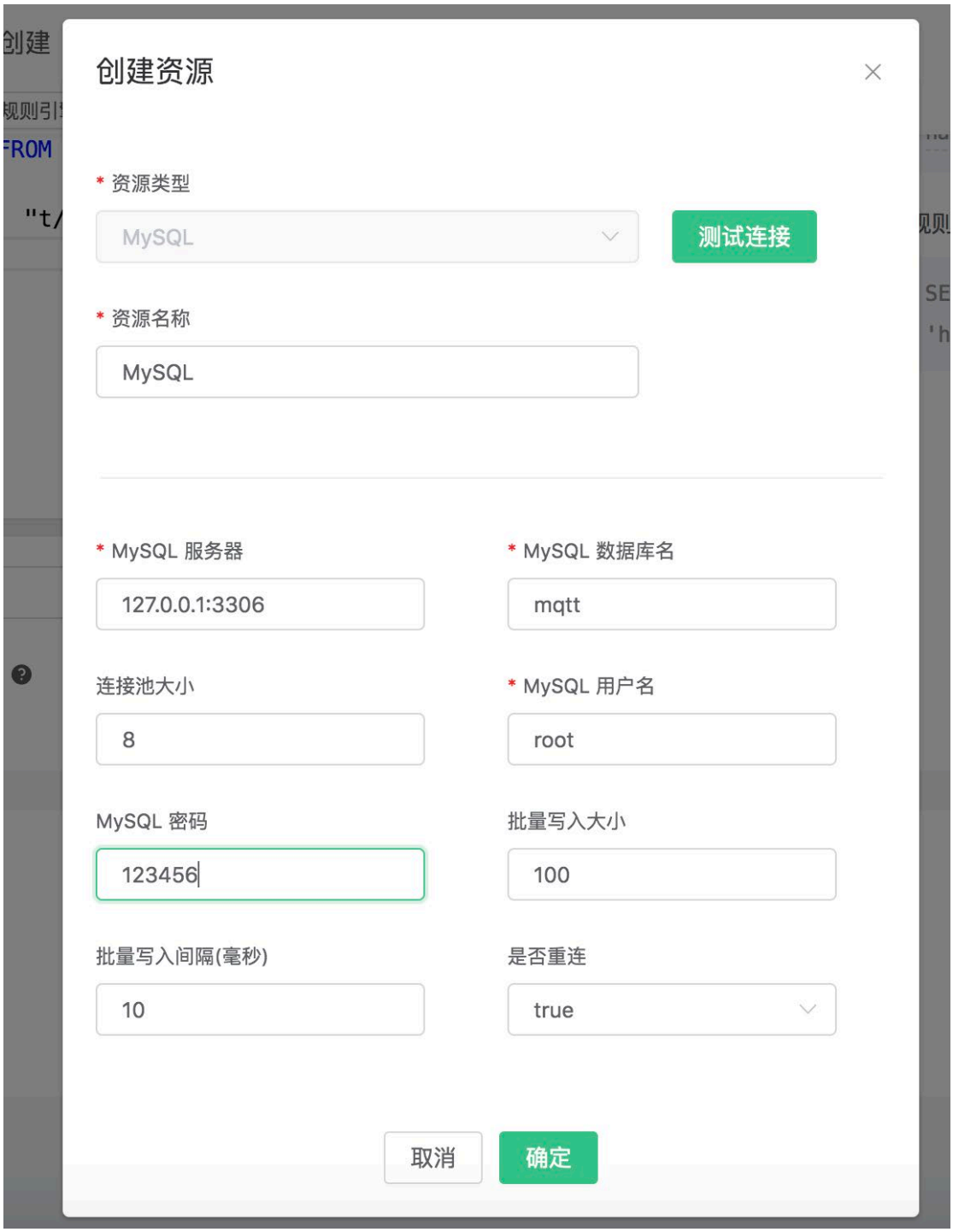
```
insert into t_mqtt_msg(msgid, topic, qos, payload, arrived) values
(${id}, ${topic}, ${qos}, ${payload},
FROM_UNIXTIME(${timestamp}/1000))
```



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:

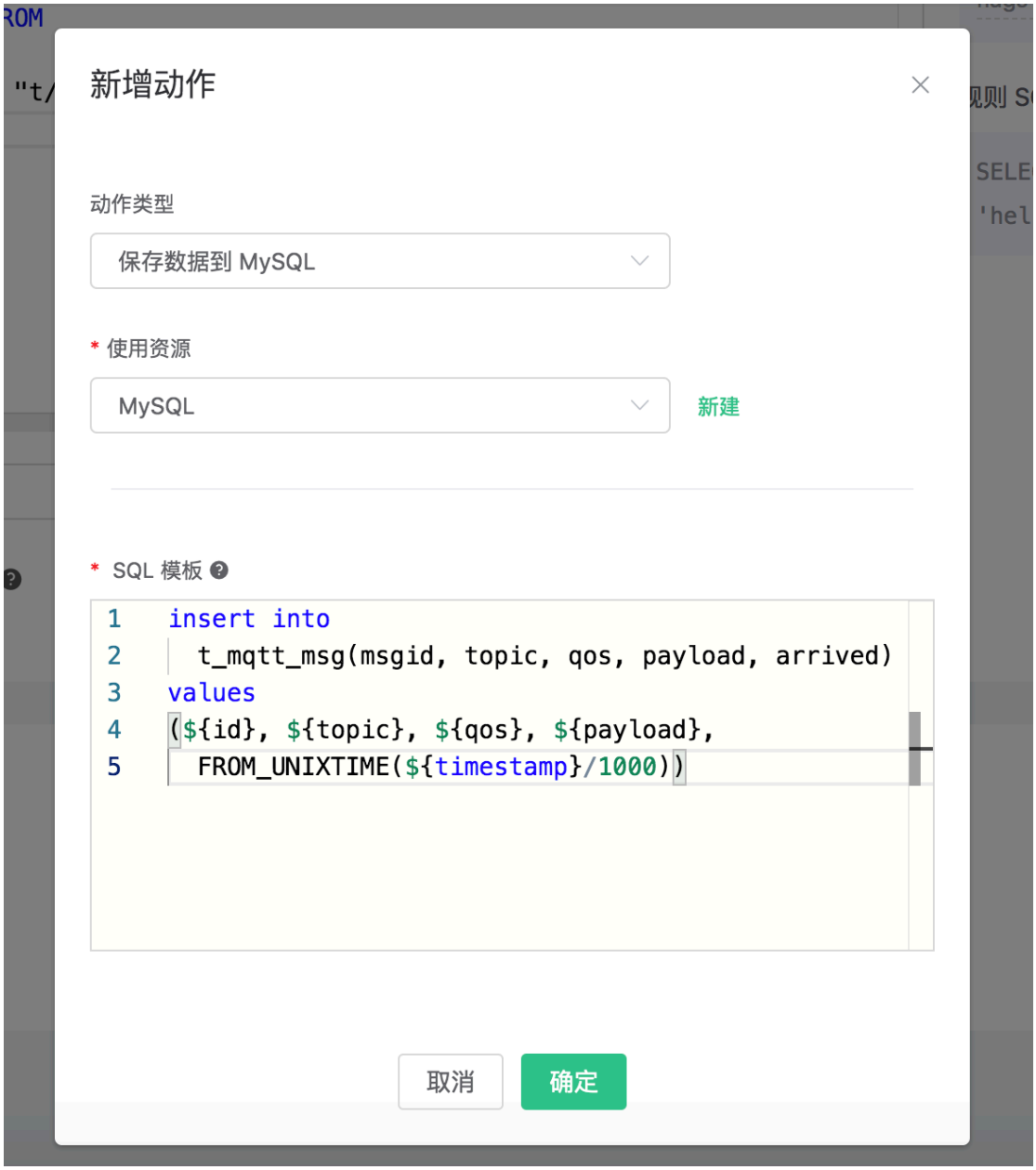
填写资源配置:

数据库名填写“mqtt”，用户名填写“root”，密码填写“123456”

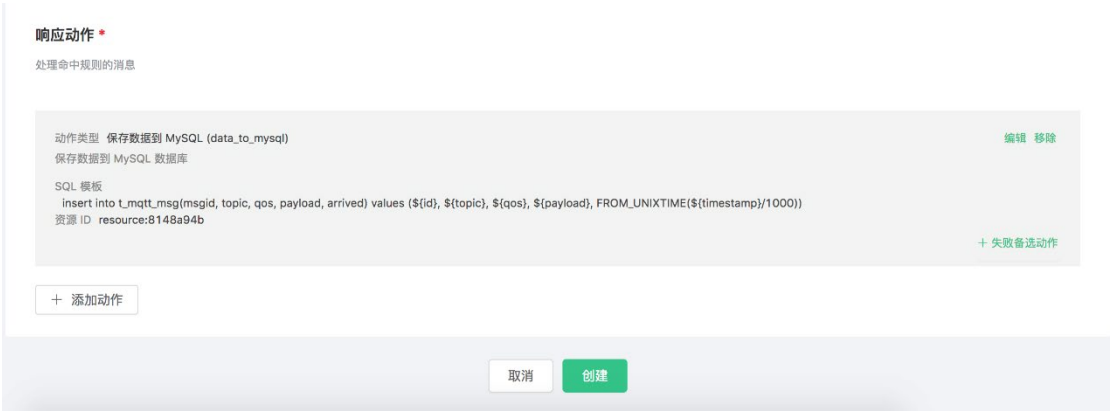


点击“新建”按钮。

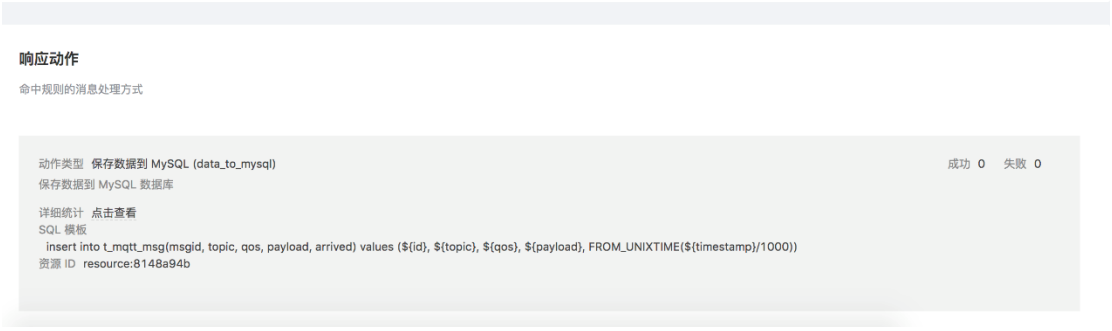
返回响应动作界面，点击“确认”。



返回规则创建界面，点击“创建”。



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:



规则已经创建完成，现在发一条数据:

```
Topic: "t/a"
QoS: 1
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>
```

MySQL 设备在线状态表

mqtt_client 存储设备在线状态:

```
DROP TABLE IF EXISTS `mqtt_client`;
CREATE TABLE `mqtt_client` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `clientid` varchar(64) DEFAULT NULL,
  `state` varchar(3) DEFAULT NULL,
  `node` varchar(64) DEFAULT NULL,
  `online_at` datetime DEFAULT NULL,
  `offline_at` datetime DEFAULT NULL,
  `created` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  KEY `mqtt_client_idx` (`clientid`),
  UNIQUE KEY `mqtt_client_key` (`clientid`),
  INDEX topic_index(`id`, `clientid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

查询设备在线状态:

```
select * from mqtt_client where clientid = ${clientid};
```

例如 ClientId 为 test 客户端上线:

```
 select * from mqtt_client where clientid = "test";


+----+----------+-------+---------------+---------------------+------
---------------+---------------------+
| id | clientid | state | node          | online_at           |
offline_at          | created             |
+----+----------+-------+---------------+---------------------+------
---------------+---------------------+
|  1 | test     | 1     | emqx@127.0.0.1 | 2016-11-15 09:40:40 |
NULL                | 2016-12-24 09:40:22 |
+----+----------+-------+---------------+---------------------+------
---------------+---------------------+
1 rows in set (0.00 sec)
```

例如 ClientId 为 test 客户端下线:

```
select * from mqtt_client where clientid = "test";


+----+----------+-------+---------------+---------------------+------
---------------+---------------------+
| id | clientid | state | node          | online_at           |
offline_at          | created             |
+----+----------+-------+---------------+---------------------+------
---------------+---------------------+
|  1 | test     | 0     | emqx@127.0.0.1 | 2016-11-15 09:40:40 | 2016-
11-15 09:46:10 | 2016-12-24 09:40:22 |
```

```
+----+----------+-------+---------------+--------------------+------
---------------+---------------------+
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:

```
DROP TABLE IF EXISTS `mqtt_sub`;
CREATE TABLE `mqtt_sub` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `clientid` varchar(64) DEFAULT NULL,
  `topic` varchar(180) DEFAULT NULL,
  `qos` tinyint(1) DEFAULT NULL,
  `created` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  KEY `mqtt_sub_idx` (`clientid`,`topic`,`qos`),
  UNIQUE KEY `mqtt_sub_key` (`clientid`,`topic`),
  INDEX topic_index(`id`, `topic`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

例如 ClientId 为 test 客户端订阅主题 test_topic1 test_topic2:

```
insert into mqtt_sub(clientid, topic, qos) values("test",
"test_topic1", 1);
insert into mqtt_sub(clientid, topic, qos) values("test",
"test_topic2", 2);
```

某个客户端订阅主题:

```
 select * from mqtt_sub where clientid = ${clientid};
```

查询 ClientId 为 test 的客户端已订阅主题:

```
select * from mqtt_sub where clientid = "test";

+----+--------------+-------------+------+---------------------+
| id | clientId     | topic       | qos  | created             |
+----+--------------+-------------+------+---------------------+
|  1 | test         | test_topic1 |    1 | 2016-12-24 17:09:05 |
|  2 | test         | test_topic2 |    2 | 2016-12-24 17:12:51 |
+----+--------------+-------------+------+---------------------+
2 rows in set (0.00 sec)
```

MySQL 消息存储表

mqtt_msg 存储 MQTT 消息:

```
DROP TABLE IF EXISTS `mqtt_msg`;
CREATE TABLE `mqtt_msg` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `msgid` varchar(64) DEFAULT NULL,
  `topic` varchar(180) NOT NULL,
  `sender` varchar(64) DEFAULT NULL,
```

```
  `node` varchar(64) DEFAULT NULL,
  `qos` tinyint(1) NOT NULL DEFAULT '0',
  `retain` tinyint(1) DEFAULT NULL,
  `payload` blob,
  `arrived` datetime NOT NULL,
  PRIMARY KEY (`id`),
  INDEX topic_index(`id`, `topic`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

查询某个客户端发布的消息:

```
select * from mqtt_msg where sender = ${clientid};
```

查询 ClientId 为 test 的客户端发布的消息:

```
select * from mqtt_msg where sender = "test";
```

```
+----+--------------------------------+----------+--------+------+-----
+--------+---------+---------------------+
| id | msgid                          | topic    | sender | node | qos |
retain | payload | arrived             |
+----+--------------------------------+----------+--------+------+-----
+--------+---------+---------------------+
| 1  | 53F98F80F66017005000004A60003 | hello    | test   | NULL |   1
|      0 | hello   | 2016-12-24 17:25:12 |
| 2  | 53F98F9FE42AD7005000004A60004 | world    | test   | NULL |   1
|      0 | world   | 2016-12-24 17:25:45 |
+----+--------------------------------+----------+--------+------+-----
+--------+---------+---------------------+
2 rows in set (0.00 sec)
```

MySQL 保留消息表

mqtt_retain 存储 retain 消息:

```
DROP TABLE IF EXISTS `mqtt_retain`;
CREATE TABLE `mqtt_retain` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `topic` varchar(180) DEFAULT NULL,
  `msgid` varchar(64) DEFAULT NULL,
  `sender` varchar(64) DEFAULT NULL,
  `node` varchar(64) DEFAULT NULL,
  `qos` tinyint(1) DEFAULT NULL,
  `payload` blob,
  `arrived` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `mqtt_retain_key` (`topic`),
  INDEX topic_index(`id`, `topic`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

查询 retain 消息:

```
select * from mqtt_retain where topic = ${topic};
```

查询 topic 为 retain 的 retain 消息:

```
select * from mqtt_retain where topic = "retain";

+----+----------+------------------------------+---------+------+----
--+---------+---------------------+
| id | topic    | msgid                        | sender  | node | qos
| payload | arrived             |
+----+----------+------------------------------+---------+------+----
--+---------+---------------------+
|  1 | retain   | 53F33F7E4741E7007000004B70001 | test    | NULL |
1 | www     | 2016-12-24 16:55:18 |
+----+----------+------------------------------+---------+------+----
--+---------+---------------------+


>    1 rows in set (0.00 sec)
```

MySQL 消息确认表

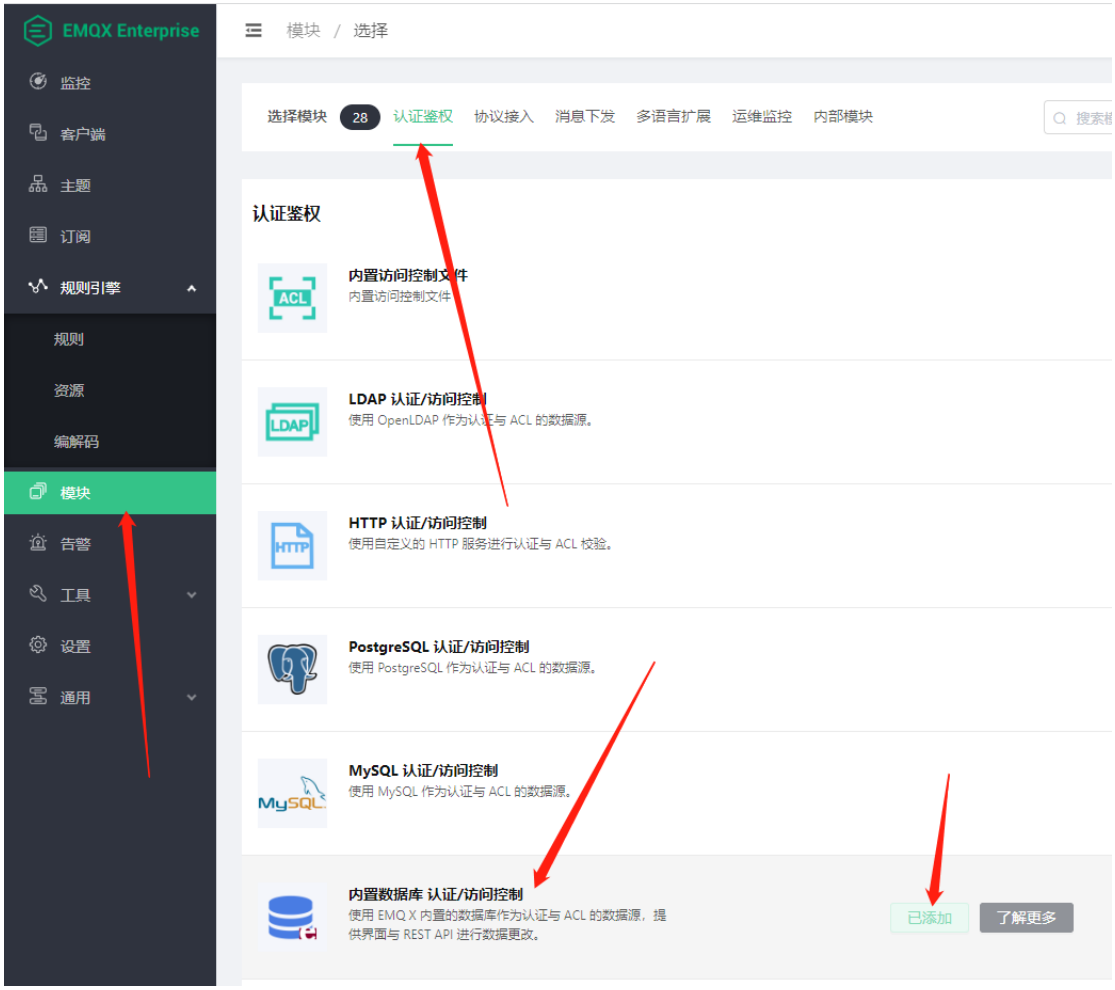
mqtt_acked 存储客户端消息确认:

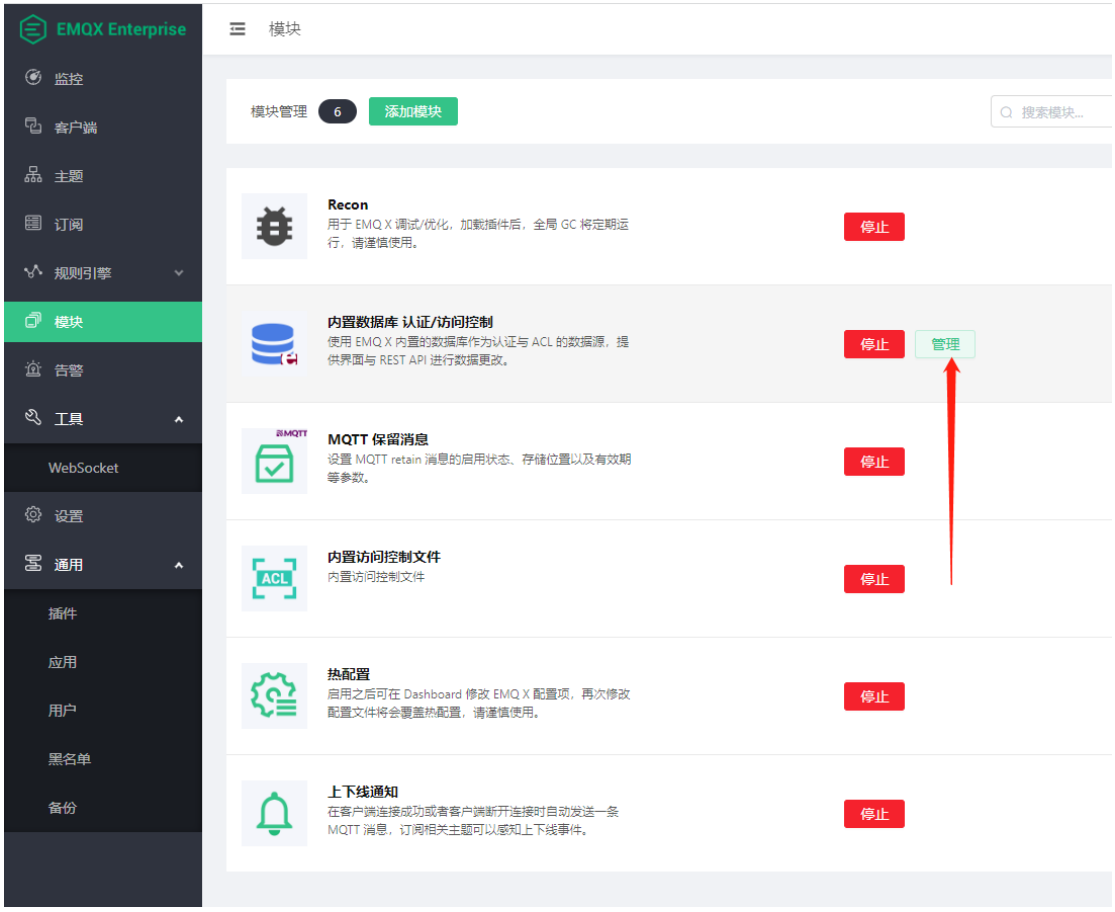
```
DROP TABLE IF EXISTS `mqtt_acked`;
CREATE TABLE `mqtt_acked` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `clientid` varchar(64) DEFAULT NULL,
  `topic` varchar(180) DEFAULT NULL,
  `mid` int(11) unsigned DEFAULT NULL,
  `created` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `mqtt_acked_key` (`clientid`,`topic`),
  INDEX topic_index(`id`, `topic`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8MB4;
```

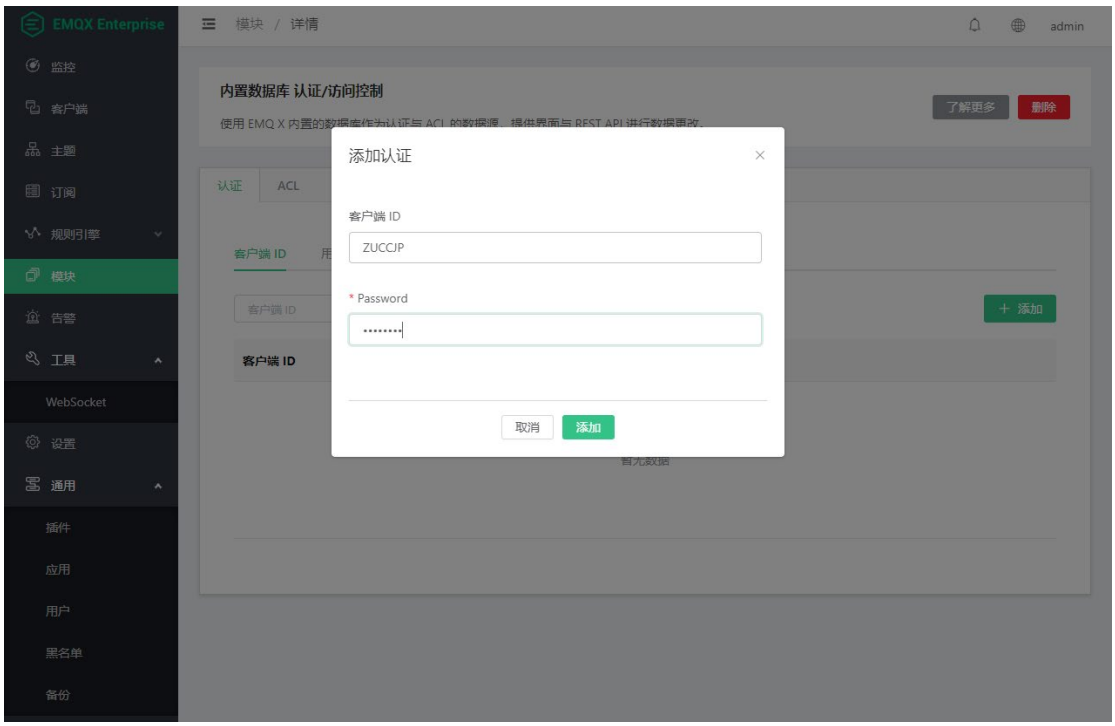
启用 **MySQL** 数据存储插件

```
./bin/emqx_ctl plugins load emqx_backend_mysql
```

补充：打开 MQTT 鉴权


EMQX Enterprise
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
通用
模块 / 选择
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
认证鉴权
内置访问控制文件
内置访问控制文件
LDAP 认证/访问控制
使用 OpenLDAP 作为认证与 ACL 的数据源。
HTTP 认证/访问控制
使用自定义的 HTTP 服务进行认证与 ACL 校验。
PostgreSQL 认证/访问控制
使用 PostgreSQL 作为认证与 ACL 的数据源。
MySQL 认证/访问控制
使用 MySQL 作为认证与 ACL 的数据源。
内置数据库 认证/访问控制
使用 EMQ X 内置的数据库作为认证与 ACL 的数据源，提供界面与 REST API 进行数据更改。
已添加
了解更多


EMQX Enterprise
模块
监控
客户端
主题
订阅
规则引擎
模块
告警
工具
WebSocket
设置
通用
插件
应用
用户
黑名单
备份
模块管理 6
添加模块
搜索模块...
Recon
用于 EMQ X 调试/优化，加载插件后，全局 GC 将定期运行，请谨慎使用。
停止
内置数据库 认证/访问控制
使用 EMQ X 内置的数据库作为认证与 ACL 的数据源，提供界面与 REST API 进行数据更改。
停止
管理
MQTT 保留消息
设置 MQTT retain 消息的启用状态、存储位置以及有效期等参数。
停止
内置访问控制文件
内置访问控制文件
停止
热配置
启用之后可在 Dashboard 修改 EMQ X 配置项，再次修改配置文件将会覆盖热配置，请谨慎使用。
停止
上下线通知
在客户端连接成功或者客户端断开连接时自动发送一条 MQTT 消息，订阅相关主题可以感知上下线事件。
停止


EMQX Enterprise
模块 / 详情
admin
内置数据库 认证/访问控制
使用 EMQ X 内置的数据库作为认证与 ACL 的数据源，提供界面与 REST API 进行数据更改。
了解更多
删除
添加认证
客户端 ID
ZUCCJP
* Password
取消
添加
认证
ACL
客户端 ID
+ 添加
暂无数据