第五章 EMQ的高级功能使用(三)

学习目标

目标1：能够说出飞行窗口的含义

目标2：能够说出在何种场景下会产生消息重传

目标3：能够说出规则引擎的组成结构，SQL语句的语法结构以及能够完成规则引擎的案例

目标4：能够说出系统调优的几个重要参数

1.飞行窗口和消息队列

1.1 简介

为了提高消息吞吐效率和减少网络波动带来的影响，EMQ X 允许多个未确认的 QoS 1 和 QoS 2 报文同时存在于网路链路上。这些已发送但未确认的报文将被存放在飞行窗口(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，即飞行窗口到达长度限制（在emqx.conf配置文件中的`max_inflight`相关配置）时，EMQ X 将不再发送后续的报文，而是将这些报文存储在 Message Queue 中。一旦飞行窗口中有报文完成确认，Message Queue 中的报文就会以先入先出的顺序被发送，同时存储到飞行窗口中。

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。因此，根据你的实际情况配置一个合适的 Message Queue 长度限制（在emqx.conf配置文件中的`max_mqueue_len`相关配置）是非常重要的。

1.2 飞行队列与Receive Maximum

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

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

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

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

配置项说明：

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

2 消息重传

2.1 简介

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

2.2 基础配置

有两种场景会导致消息重发：

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

在 `etc/emqx.conf` 中可配置：

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

3. 规则引擎

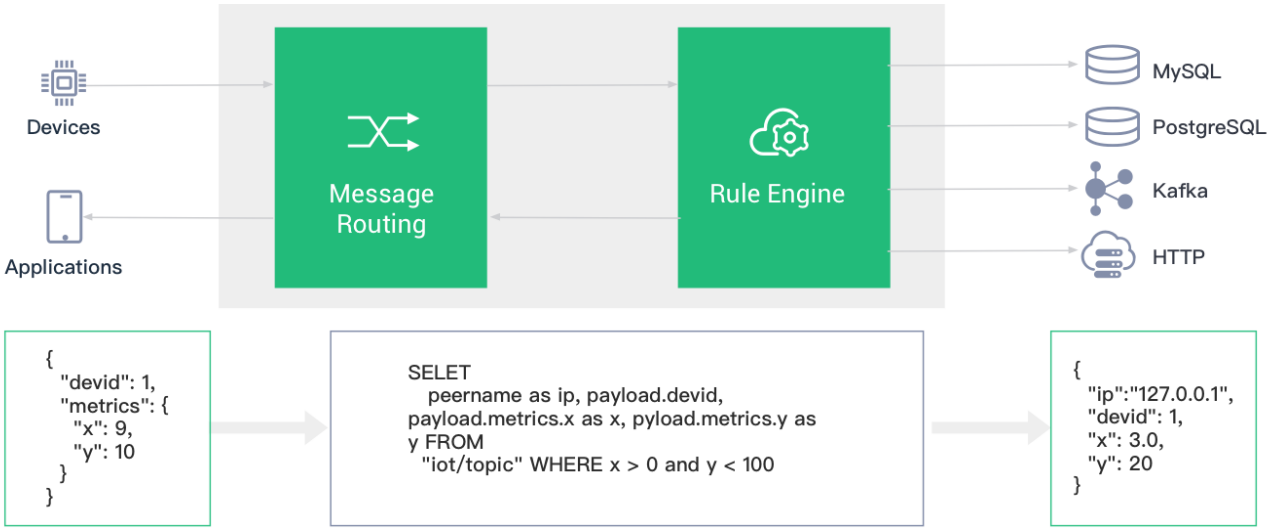
3.1规则引擎概述

3.1.1 简介

EMQ X Rule Engine (以下简称规则引擎) 用于配置 EMQ X 消息流与设备事件的处理、响应规则。

> 规则引擎用于配置一套规则，该规则是针对EMQ X的消息流和设备事件如何处理的一套细则。

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



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

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关系型数据库（mysql，PostgreSQL），NoSql（Redis，MongoDB），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。

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。

3.1.2 应用场景

- 动作监听：智慧家庭智能门锁开发中，门锁会因为网络、电源故障、人为破坏等原因离线导致功能异常，使用规则引擎配置监听离线事件向应用服务推送该故障信息，可以在接入层实现第一时间的故障检测的能力；
- 数据筛选：车辆网的卡车车队管理，车辆传感器采集并上报了大量运行数据，应用平台仅关注车速大于 40 km/h 时的数据，此场景下可以使用规则引擎对消息进行条件过滤，向业务消息队列写入满足条件的数据；
- 消息路由：智能计费应用中，终端设备通过不同主题区分业务类型，可通过配置规则引擎将计费业务的消息接入计费消息队列并在消息抵达设备端后发送确认通知到业务系统，非计费信息接入其他消息队列，实现业务消息路由配置；
- 消息编解码：其他公共协议 / 私有 TCP 协议接入、工控行业等应用场景下，可以通过规则引擎的本地处理函数（可在 EMQ X 上定制开发）做二进制 / 特殊格式消息体的编解码工作；亦可通过规则引擎的消息路由

将相关消息流向外部计算资源如函数计算进行处理（可由用户自行开发处理逻辑），将消息转为业务易于处理的 JSON 格式，简化项目集成难度、提升应用快速开发交付能力。

3.2 规则引擎的组成

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。

注意： 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。

总的来说：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。

3.3 SQL语句

3.3.1 SQL语法

FROM、SELECT 和 WHERE 子句:

SQL 语句用于从原始数据中，根据条件筛选出字段，并进行预处理和转换，基本格式为:

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

FROM、SELECT 和 WHERE 子句:

- `FROM` 子句将规则挂载到某个主题上（向该主题发布消息时触发，该主题是事件主题则事件发生时触发）
- `SELECT` 子句用于选择输出结果中的字段
- `WHERE` 子句用于根据条件筛选消息

FOREACH、DO 和 INCASE 子句:

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

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

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

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

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

3.3.2 SQL语句相关示例

3.3.2.1 基本语法举例

- 从 topic 为 "t/a" 的消息中提取所有字段:

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

- 从 topic 为 "t/a" **或** "t/b" 的消息中提取所有字段:

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

- 从 topic 能够匹配到 't/#' 的消息中提取所有字段。

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

- 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

3.3.2.2 遍历语法举例

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

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

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

1) 主题：sensors/0 内容：a

2) 主题：sensors/1 内容：b

3) 主题：sensors/2 内容：c

要完成这个规则，我们需要配置如下动作：

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

以及如下 SQL 语句：

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

示例解析:

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

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

```
]
```

> 因为选出来的结果中所有字段我们都需要因此不需要DO子句，没有条件过滤因为不需要INCASE子句

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:

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

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

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

要完成这个规则，我们需要配置如下动作：

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

以及如下 SQL 语句：

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

示例解析:

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

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

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

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

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

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

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

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

CASE-WHEN 语法示例

示例1: 将消息中 x 字段的值范围限定在 0~7 之间。

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

假设消息为:

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

则上面的 SQL 输出为:

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

3.3.3 SQL事件和字段

FROM 子句可用的事件主题

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

From子句除了写事件主题外就是普通的主题了，譬如`t/+,q/#`等。

SELECT 和 WHERE 子句可用的字段

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

（1）普通主题 (消息发布)，比如`t/+,q/#等`

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

（2）$events/message_delivered (消息投递)

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

（3）$events/message_acked (消息确认)

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

（4）$events/message_dropped (消息丢弃)

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

（5）$events/client_connected (终端连接成功)

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

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

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

（7）$events/session_subscribed (终端订阅成功)

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

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

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

3.3.4 SQL 运算符和函数

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	

SQL 语句中可用的函数

（1）数学函数

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

（2）数据类型判断函数

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

（3）数据类型转换函数

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

（4）字符串函数

函数名	函数作用	参数	返回值	举例
lower	转为小写	1. 原字符串	小写字符串	`1. lower('AbC') = 'abc'``2. lower('abc') = 'abc'`
upper	转为大写	1. 原字符串	大写字符串	`1. upper('AbC') = 'ABC'``2. lower('ABC') = 'ABC'`
trim	去掉左右空格	1. 原字符串	去掉空格后的字符串	`1. trim(' hello ') = 'hello'`
ltrim	去掉左空格	1. 原字符串	去掉空格后的字符串	`1. ltrim(' hello ') = 'hello '`
rtrim	去掉右空格	1. 原字符串	去掉空格后的字符串	`1. rtrim(' hello ') = ' hello'`
reverse	字符串反转	1. 原字符串	翻转后的字符串	`1. reverse('hello') = 'olleh'`
strlen	取字符串长度	1. 原字符串	整数值，字符长度	`1. strlen('hello') = 5`
substr	取字符的子串	1. 原字符串 2. 起始位置. 注意: 下标从 0 开始	子串	`1. substr('abcdef', 2) = 'cdef'`
substr	取字符的子串	1. 原字符串 2. 起始位置 3. 要取出的子串长度. 注意: 下标从 0 开始	子串	`1. substr('abcdef', 2, 3) = 'cde'`
split	字符串分割	1. 原字符串 2. 分割符子串	分割后的字符串数组	`1. split('a/b/ c', '/') = ['a', 'b', ' c']`
split	字符串分割, 只查找左边第一个分隔符	1. 原字符串 2. 分割符子串 3. 'leading'	分割后的字符串数组	`1. split('a/b/ c', '/', 'leading') = ['a', 'b/ c']`

函数名	函数作用	参数	返回值	举例
split	字符串分割, 只查找右边第一个分隔符	1. 原字符串 2. 分割符子串 3. 'trailing'	分割后的字符串数组	`1. split('a/b/ c', '/', 'trailing') = ['a/b', ' c']`
concat	字符串拼接	1. 左字符串 2. 右符子串	拼接后的字符串	`1. concat('a', '/bc') = 'a/bc'``2. 'a' + '/bc' = 'a/bc'`
tokens	字符串分解(按照指定字符串符分解)	1. 输入字符串 2. 分割符或字符串	分解后的字符串数组	`1. tokens(' a/b/ c', '/') = [' a', 'b', ' c']``2. tokens(' a/b/ c', '/') = ['a', 'b', 'c']``3. tokens(' a/b/ c\n', '/ ') = ['a', 'b', 'c\n']`
tokens	字符串分解(按照指定字符串和换行符分解)	1. 输入字符串 2. 分割符或字符串 3. 'nocrlf'	分解后的字符串数组	`1. tokens(' a/b/ c\n', '/ ', 'nocrlf') = ['a', 'b', 'c']``2. tokens(' a/b/ c\r\n', '/ ', 'nocrlf') = ['a', 'b', 'c']`
sprintf	字符串格式化,	1. 格式字符串 2,3,4... 参数列表。参数个数不定	分解后的字符串数组	`1. sprintf('hello, ~s!', 'steve') = 'hello, steve!'``2. sprintf('count: ~p~n', 100) = 'count: 100\n'`
pad	字符串补足长度，补空格，从尾部补足	1. 原字符串 2. 字符总长度	补足后的字符串	`1. pad('abc', 5) = 'abc '`
pad	字符串补足长度，补空格，从尾部补足	1. 原字符串 2. 字符总长度 3. 'trailing'	补足后的字符串	`1. pad('abc', 5, 'trailing') = 'abc '`
pad	字符串补足长度，补空格，从两边补足	1. 原字符串 2. 字符总长度 3. 'both'	补足后的字符串	`1. pad('abc', 5, 'both') = ' abc '`
pad	字符串补足长度，补空格，从头部补足	1. 原字符串 2. 字符总长度 3. 'leading'	补足后的字符串	`1. pad('abc', 5, 'leading') = ' abc'`

函数名	函数作用	参数	返回值	举例
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	`1. pad('abc', 5, 'trailing', '*') = 'abc**'``2. pad('abc', 5, 'trailing', '*#') = 'abc*#*#'`
pad	字符串补足长度，补指定字符，从两边补足	1. 原字符串 2. 字符总长度 3. 'both' 4. 指定用于补足的字符	补足后的字符串	`1. pad('abc', 5, 'both', '*') = '*abc*'``2. pad('abc', 5, 'both', '*#') = '*#abc*#'`
pad	字符串补足长度，补指定字符，从头部补足	1. 原字符串 2. 字符总长度 3. 'leading' 4. 指定用于补足的字符	补足后的字符串	`1. pad('abc', 5, 'leading', '*') = '**abc'``2. pad('abc', 5, 'leading', '*#') = '*#*#abc'`
replace	替换字符串中的某子串，查找所有匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串	替换后的字符串	`1. replace('ababef', 'ab', 'cd') = 'cdcdef'`
replace	替换字符串中的某子串，查找所有匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'all'	替换后的字符串	`1. replace('ababef', 'ab', 'cd', 'all') = 'cdcdef'`
replace	替换字符串中的某子串，从尾部查找第一个匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'trailing'	替换后的字符串	`1. replace('ababef', 'ab', 'cd', 'trailing') = 'abcdef'`
replace	替换字符串中的某子串，从头部查找第一个匹配子串替换	1. 原字符串 2. 要被替换的子串 3. 指定用于替换的字符串 4. 'leading'	替换后的字符串	`1. replace('ababef', 'ab', 'cd', 'leading') = 'cdabef'`
regex_match	判断字符串是否与某正则表达式匹配	1. 原字符串 2. 正则表达式	true 或 false	`1. regex_match('abc123', '[a-zA-Z1-9]*') = true`

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

（5）Map 函数

函数名	函数作用	参数	返回值
map_get	取 Map 中某个 Key 的值，如果没有则返回空值	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	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	Key 2. Value 3. Map	插入后的 Map。支持嵌套的 Key，比如 "a.b.c"

（6）数组函数

函数名	函数作用	参数	返回值
nth	取第 n 个元素，下标从 1 开始	原数组	第 n 个元素
length	获取数组的长度	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	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	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	原数组	第 1 个元素
last	取最后一个元素。	原数组	最后一个元素
contains	判断数据是否在数组里面	数据 2. 原数组	Boolean 值

（7）哈希函数

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

（8）编解码函数

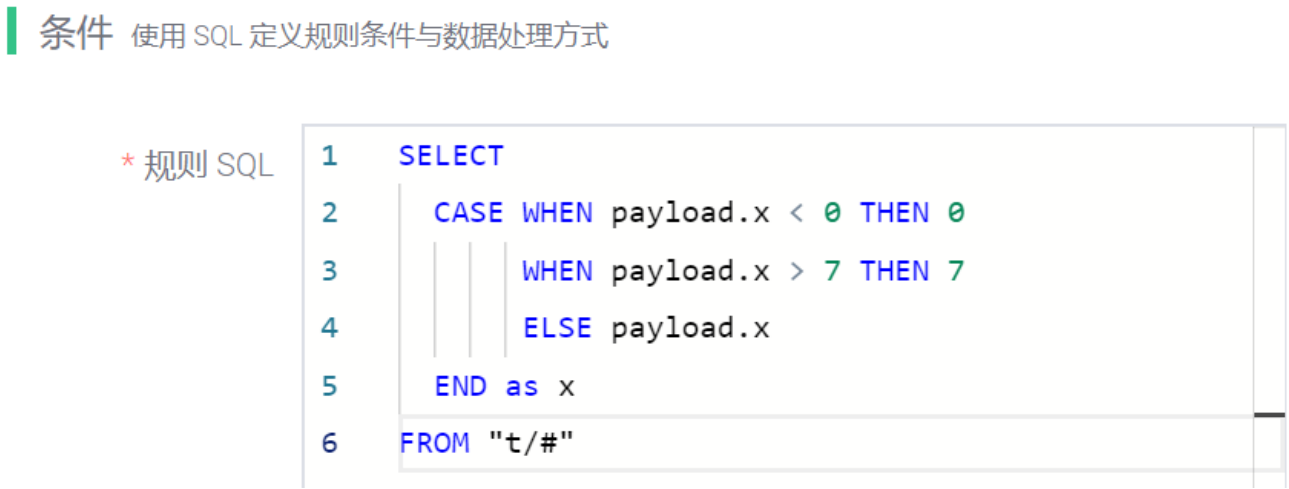
函数名	函数作用	参数	返回值
base64_encode	BASE64 编码	数据	BASE64 字符串
base64_decode	BASE64 解码	BASE64 字符串	数据
json_encode	JSON 编码	JSON 字符串	内部 Map
json_decode	JSON 解码	内部 Map	JSON 字符串
schema_encode	Schema 编码	Schema ID 2. 内部 Map	数据
schema_encode	Schema 编码	Schema ID 2. 内部 Map 3. Protobuf Message 名	数据
schema_decode	Schema 解码	Schema ID 2. 数据	内部 Map
schema_decode	Schema 解码	Schema ID 2. 数据 3. Protobuf Message 名	内部 Map

3.3.5 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。

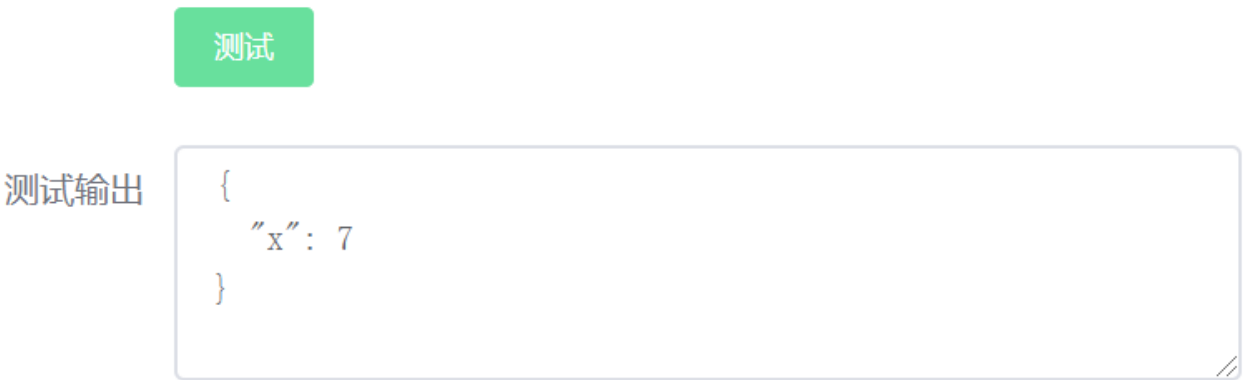
1：在创建规则界面，输入 **规则SQL**，

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



并启用 **SQL 测试** 开关，输入payload数据，

点击测试查看结果：



3.4 规则引擎案例

需求：现需要通过规则引擎提取出从`username=emq-client2`该客户端发送过来原始数据中的`msg,user,orderNo`等数据，需要过滤`password`字段，同时还需要提取消息发布的qos信息，然后将最终过滤出来的消息通知到我们的web服务上。

消息主体是：`rule/#`，消息数据模板为

```
{
  "msg": "hello",
  "user":"emq-client2",
  "password":"123456",
  "orderNo":"12345sfd"
}
```

3.4.1 创建资源

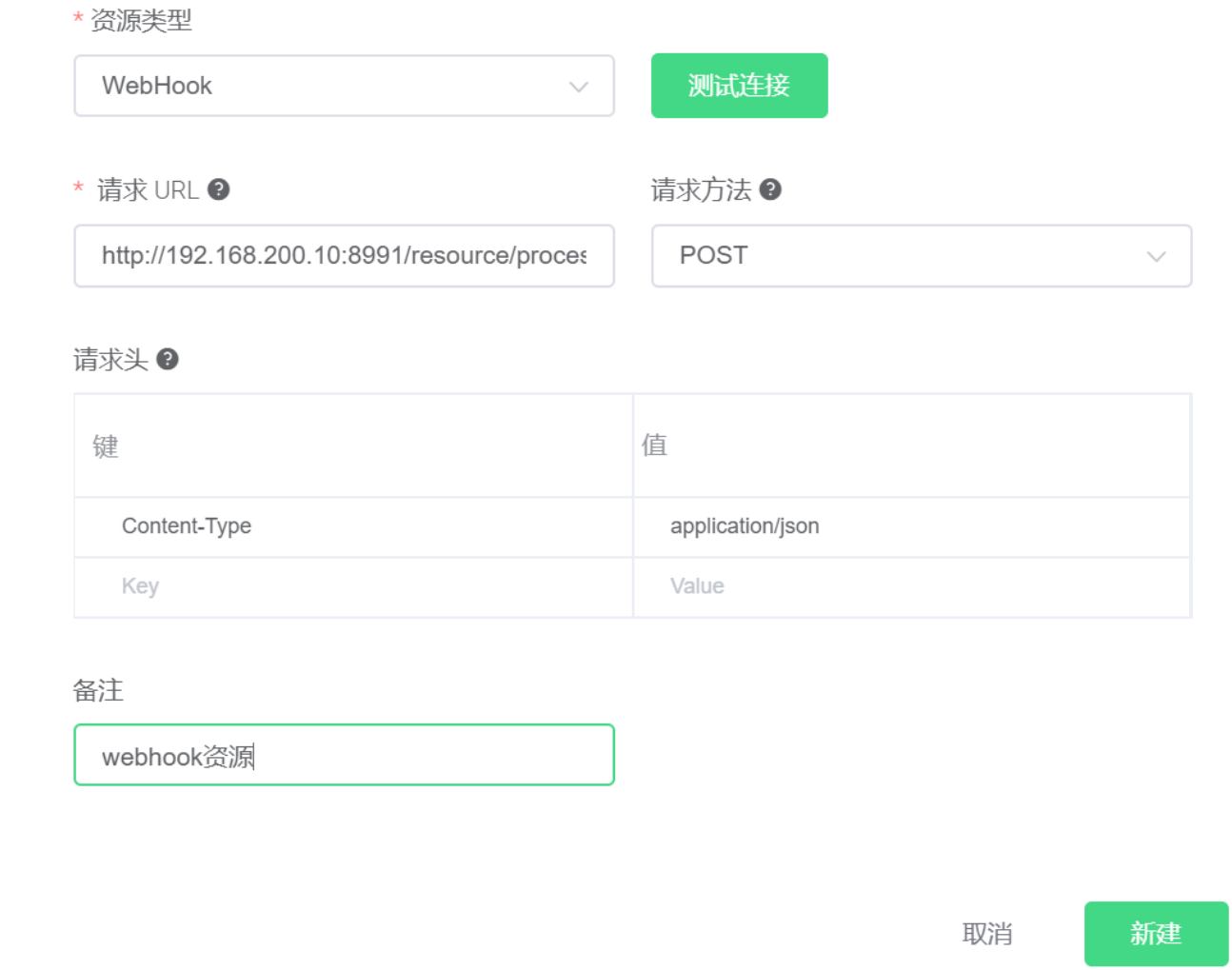
（1）打开`emqx dashboard`，选择左侧的“规则引擎”选项卡，打开资源页面，

（2）：点击新建，创建资源

开源的EMQ X Broker默认支持的资源类型只有这三种，EMQ X 企业版支持的更多，这些资源类型其实对应了规则匹配后的具体动作`Action`，

（3）填写资源信息

资源请求URL：`http://192.168.200.10:8991/resource/process`



创建完成后可以看到列表展示：

3.4.2 创建规则

（1）打开`emqx dashboard`，选择左侧的“规则引擎”选项卡，打开规则页面，

（2）右上角点击`新建`创建规则，填写规则 SQL，消息数据模板为

```
{
  "msg": "hello",
  "user":"emq-client2",
  "password":"123456",
  "orderNo":"12345sfd"
}
```

需要通过规则引擎提取出从`username=emq-client2`该客户端发送过来原始数据中的`msg,user,orderNo`等数据，需要过滤`password`字段，同时还需要提取消息发布的qos信息。

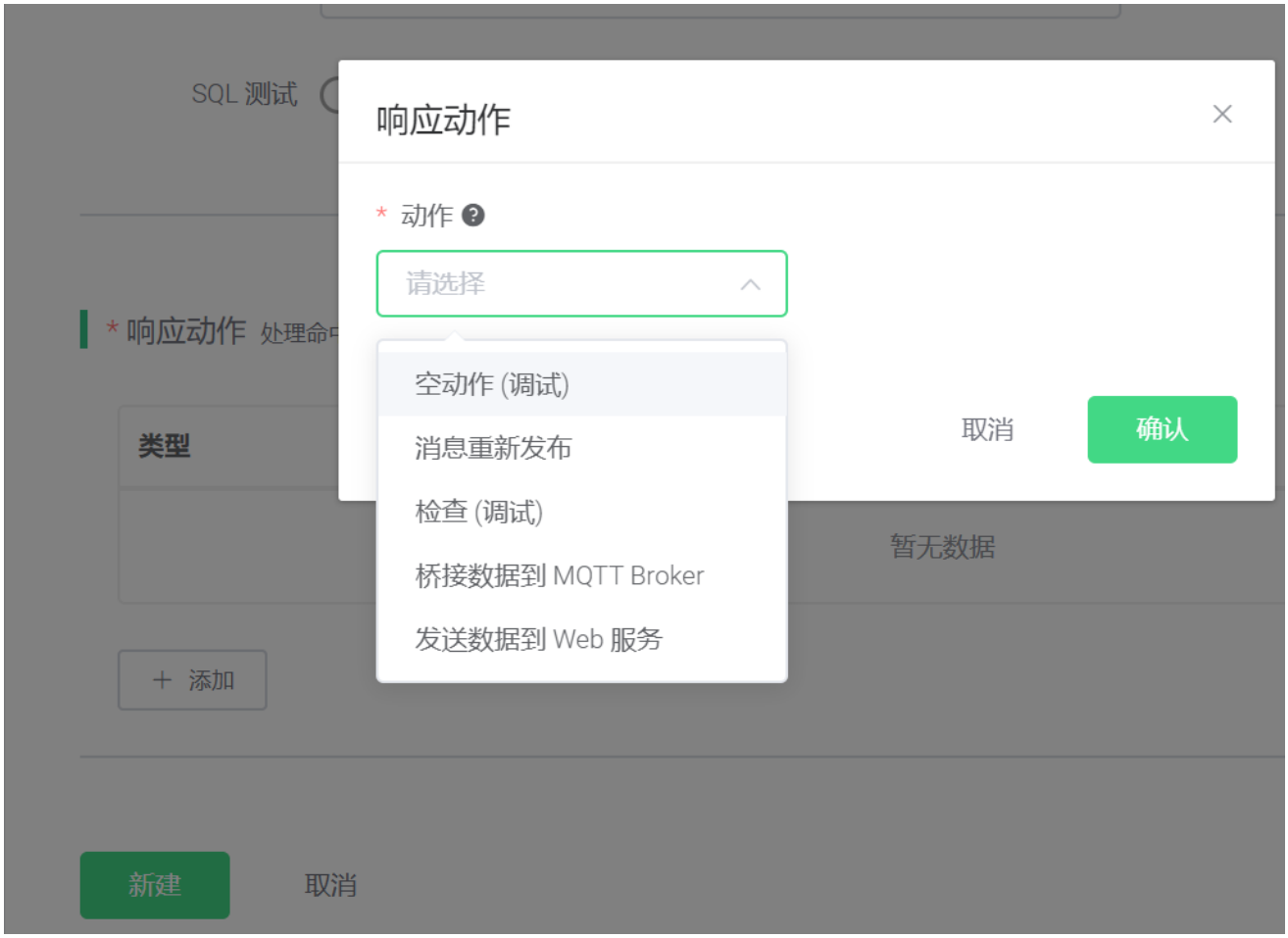
请思考该规则SQL如何编写?

```
SELECT
  payload.msg as msg,
  payload.user as user,
  payload.orderNo as orderNo,
  qos
FROM
  "rule/#"
WHERE
  username = 'emq-client2'
```

将该SQL存入

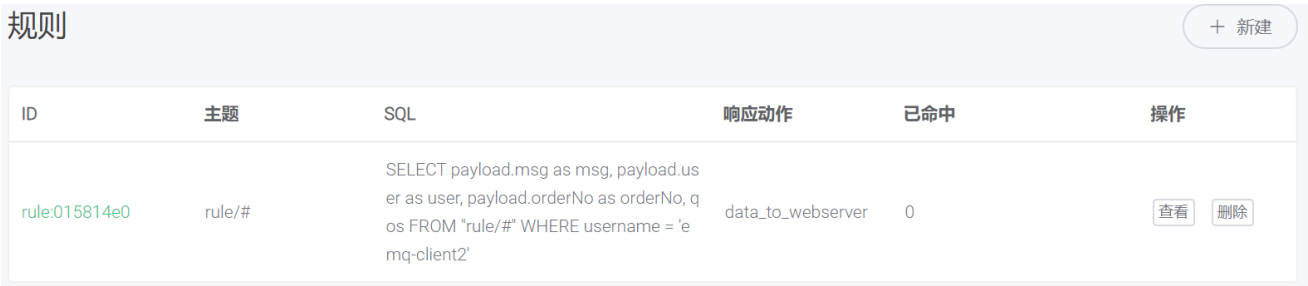
（3）：关联动作:

在 “响应动作” 界面选择 “添加”，然后在 “动作” 下拉框里选择 要操作的动作为`发送数据到web服务`，



关联如图所示：

创建完成的规则如下所示：



3.4.3 创建资源webhook接口

修改原有代码，添加一个http接口用来接收emq通过Post发送过来的数据，在这里我们只是简单的输出到控制台，证明我们的程序已经接收到了通过自己创建的规则引擎转发过来的数据。在实际业务中，我们会将接收到的数据进行后续复杂的业务处理，这里只是简单演示。

创建：com.itheima.controller.mqtt.RuleController

```
@RestController
@RequestMapping("/resource")
public class RuleController {


    @PostMapping("/process")
    public void process(@RequestBody Map<String,Object> params){
        params.entrySet().stream().forEach(x->{
            System.out.println(x.getKey() + ":" + x.getValue());
        });
    }
}
```

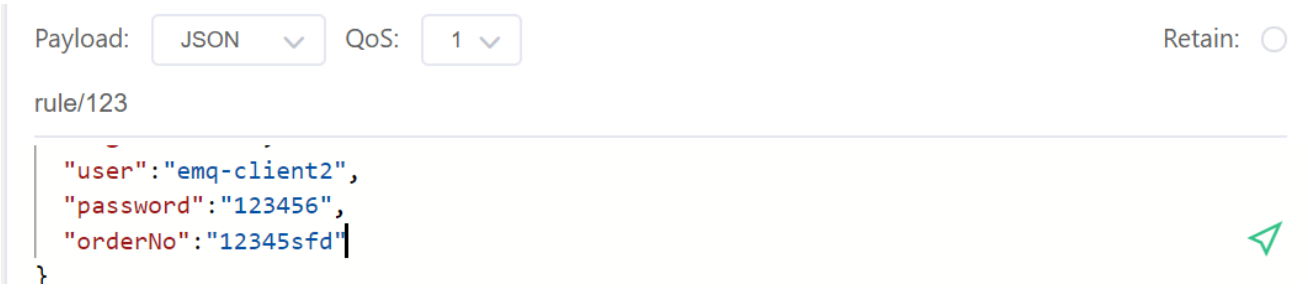
3.4.4 测试

（1）启动`emq-demo`项目

（2）打开MQTTX客户端，用`emq-client2/123456登陆`

（3）：向主题`rule/123`发送如下数据

```
{
  "msg": "hello",
  "user":"emq-client2",
  "password":"123456",
  "orderNo":"12345sfd"
}
```



查看服务控制台的输出！

4.系统调优

EMQ X 消息服务器 4.x 版本 MQTT 连接压力测试到 130 万，在一台 8 核心、32G 内存的 CentOS 服务器上。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**，*EMQ X* 消息服务器参数设置如下:

4.1 Linux 操作系统参数

（1）系统全局允许分配的最大文件句柄数:

> 在文件I/O中，要从一个文件读取数据，应用程序首先要调用操作**系统函数**并传送文件名，并选一个到该文件的路径来打开文件。该函数取回一个顺序号，即文件句柄（file handle），该文件句柄对于打开的文件是唯一的识别依据。要从文件中读取一块数据，应用程序需要调用函数ReadFile，并将文件句柄在内存中的地址和要拷贝的字节数传送给操作系统。当完成任务后，再通过调用系统函数来关闭该文件。

```
# 2 millions system-wide
sysctl -w fs.file-max=2097152
sysctl -w fs.nr_open=2097152
echo 2097152 > /proc/sys/fs/nr_open
```

> file-max是所有进程最大的文件数
>
> nr_open是单个进程可分配的最大文件数

(2) 允许当前会话 / 进程打开文件句柄数:

```
ulimit -n 1048576
```

> ulimit其实就是对单一程序的限制,进程级别的，ulimit 参数说明如下

选项 [options]	含义	例子
-H	设置硬资源限制，一旦设置不能增加。	ulimit – Hs 64；限制硬资源，线程栈大小为 64K。
-S	设置软资源限制，设置后可以增加，但是不能超过硬资源设置。	ulimit – Sn 32；限制软资源，32 个文件描述符。
-a	显示当前所有的 limit 信息。	ulimit – a；显示当前所有的 limit 信息。
-c	最大的 core 文件的大小， 以 blocks 为单位。	ulimit – c unlimited； 对生成的 core 文件的大小不进行限制。
-d	进程最大的数据段的大小，以 Kbytes 为单位。	ulimit -d unlimited；对进程的数据段大小不进行限制。
-f	进程可以创建文件的最大值，以 blocks 为单位。	ulimit – f 2048；限制进程可以创建的最大文件大小为 2048 blocks。
-l	最大可加锁内存大小，以 Kbytes 为单位。	ulimit – l 32；限制最大可加锁内存大小为 32 Kbytes。
-m	最大内存大小，以 Kbytes 为单位。	ulimit – m unlimited；对最大内存不进行限制。
-n	可以打开最大文件描述符的数量。	ulimit – n 128；限制最大可以使用 128 个文件描述符。
-p	管道缓冲区的大小，以 Kbytes 为单位。	ulimit – p 512；限制管道缓冲区的大小为 512 Kbytes。
-s	线程栈大小，以 Kbytes 为单位。	ulimit – s 512；限制线程栈的大小为 512 Kbytes。
-t	最大的 CPU 占用时间，以秒为单位。	ulimit – t unlimited；对最大的 CPU 占用时间不进行限制。
-u	用户最大可用的进程数。	ulimit – u 64；限制用户最多可以使用 64 个进程。
-v	进程最大可用的虚拟内存，以 Kbytes 为单位。	ulimit – v 200000；限制最大可用的虚拟内存为 200000 Kbytes。

/etc/sysctl.conf

持久化 'fs.file-max' 设置到 `/etc/sysctl.conf` 文件:

```
fs.file-max = 1048576
```

`/etc/systemd/system.conf` 设置服务最大文件句柄数:

```
DefaultLimitNOFILE=1048576
```

/etc/security/limits.conf

是linux资源限制配置文件，限制用户进程的数量，limits.conf文件限制着用户可以使用的最大文件数，最大线程，最大内存等资源使用量。

`/etc/security/limits.conf` 持久化设置允许用户 / 进程打开文件句柄数:

```
#<domain>   <type>  <item>   <value>
#
#*          soft    core     0
#*          hard    rss      10000
*       soft    nofile      1048576
*       hard    nofile      1048576
```

```
* soft nofile 655350  #任何用户可以打开的最大的文件描述符数量，默认1024，这里的数值会限制tcp连接
* hard nofile 655350
* soft nproc  655350  #任何用户可以打开的最大进程数
* hard nproc  650000

@student hard nofile 65535
@student soft nofile 4096
@student hard nproc 50  #学生组中的任何人不能拥有超过50个进程，并且会在拥有30个进程时发出警告
@student soft nproc 30
```

hard和soft两个值都代表什么意思呢?

soft是一个警告值，而hard则是一个真正意义的阀值，超过就会报错

4.2 TCP 协议栈网络参数

并发连接 backlog 设置:

```
sysctl -w net.core.somaxconn=32768 #backlog值
sysctl -w net.ipv4.tcp_max_syn_backlog=16384 #控制半连接的队列的长度
sysctl -w net.core.netdev_max_backlog=16384 #网卡设备的backlog
```

`TCP` 通过三次握手建立连接的过程应该都不陌生了。从服务器的角度看，它分为以下几步

1. 将 `TCP` 状态设置为 `LISTEN` 状态，开启监听客户端的连接请求
2. 收到客户端发送的 `SYN` 报文后，`TCP` 状态切换为 `SYN RECEIVED`，并发送 `SYN ACK` 报文
3. 收到客户端发送的 `ACK` 报文后，`TCP` 三次握手完成，状态切换为 `ESTABLISHED`

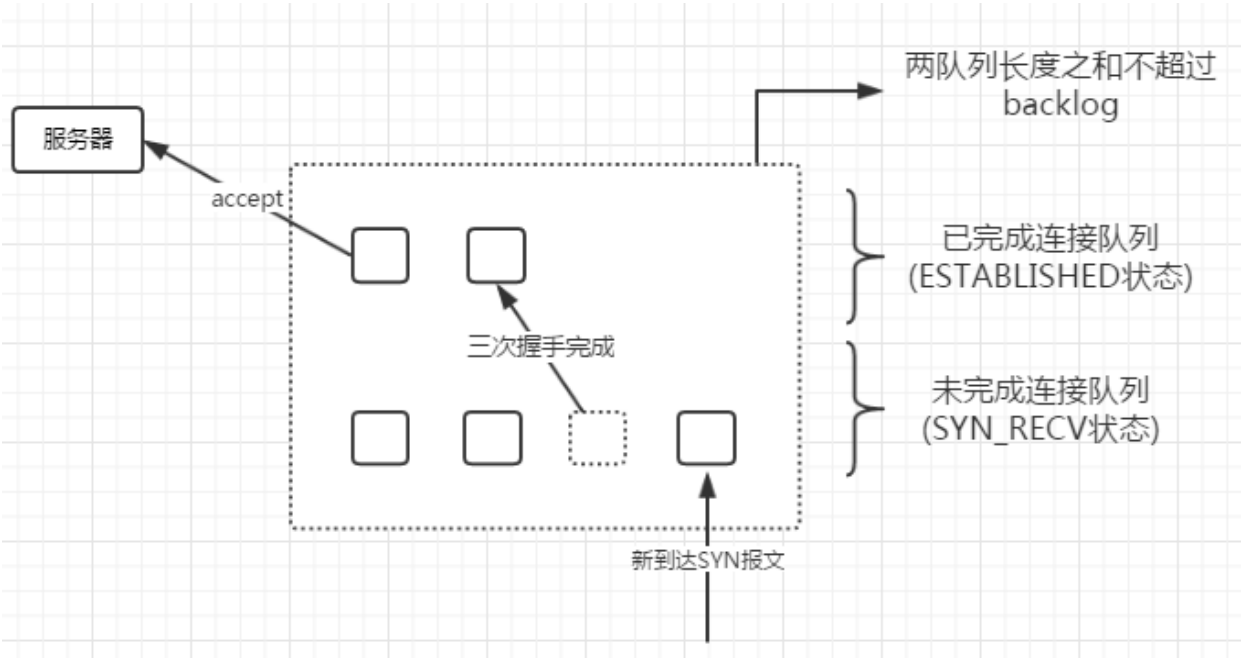
在 `Unix` 系统中，开启监听是通过 `listen` 完成。

```
int listen(int sockfd, int backlog)
```

`listen`有两个参数，第一个参数`sockfd`表示要设置的套接字，本文主要关注的是其第二个参数`backlog`；

backlog： 将其描述为**已完成的连接队列**(`ESTABLISHED`)与**未完成连接队列**(`SYN_RCVD`)之和的上限。

一般我们将`ESTABLISHED`状态的连接称为**全连接**，而将`SYN_RCVD`状态的连接称为**半连接**



当服务器收到一个`SYN`后，它创建一个**子连接**加入到`SYN_RCVD`队列。在收到`ACK`后，它将这个**子连接**移动到`ESTABLISHED`队列。最后当用户调用`accept()`时，会将连接从`ESTABLISHED`队列取出。

backlog对应参数是：net.core.somaxconn

tcp_max_syn_backlog：这个参数是控制半连接的队列的长度；这个表现出来的故障是： 无法建立连接。

netdev_max_backlog：表示网卡设备的backlog, 因为网卡接收数据包的速度远大于内核处理这些数据包的速度，所以，就出现了网卡设备的backlog.

可用端口范围:

```
sysctl -w net.ipv4.ip_local_port_range='1000 65535'
```

在对于繁忙的网络服务器，如代理服务器或负载均衡器，我们可能需要增加网络端口范围来增强它的处理能力

在Linux上，有一个sysctl参数`ip_local_port_range`，可用于定义网络连接可用作其源（本地）端口的最小和最大端口的限制，同时适用于TCP和UDP连接。

TCP Socket 读写 Buffer 设置:

```
sysctl -w net.core.rmem_default=262144  #默认的接收窗口大小
sysctl -w net.core.wmem_default=262144  #默认的发送窗口大小
sysctl -w net.core.rmem_max=16777216    #最大的TCP数据接收缓冲
sysctl -w net.core.wmem_max=16777216    #最大的TCP数据发送缓冲
sysctl -w net.core.optmem_max=16777216  #每个套接字所允许的最大缓冲区的大小。

#sysctl -w net.ipv4.tcp_mem='16777216 16777216 16777216'
sysctl -w net.ipv4.tcp_rmem='1024 4096 16777216'
#定义socket使用的内存。第一个值是为socket接收缓冲区分配的最少字节数；第二个值是默认值（该值会被rmem_default覆盖），缓冲区在系统负载不重的情况下可以增长到这个值；第三个值是接收缓冲区空间的最大字节数（该值会被rmem_max覆盖）
sysctl -w net.ipv4.tcp_wmem='1024 4096 16777216'
#定义socket使用的内存。第一个值是为socket发送缓冲区分配的最少字节数；第二个值是默认值（该值会被wmem_default覆盖），缓冲区在系统负载不重的情况下可以增长到这个值；第三个值是发送缓冲区空间的最大字节数（该值会被wmem_max覆盖）。
```

TCP 连接追踪设置:

```
sysctl -w net.nf_conntrack_max=1000000
sysctl -w net.netfilter.nf_conntrack_max=1000000
sysctl -w net.netfilter.nf_conntrack_tcp_timeout_time_wait=30
```

TIME-WAIT Socket 最大数量、回收与重用设置:

```
sysctl -w net.ipv4.tcp_max_tw_buckets=1048576
#在 TIME_WAIT 数量等于 tcp_max_tw_buckets 时，不会有新的 TIME_WAIT 产生
# 注意：不建议开启该设置，NAT 模式下可能引起连接 RST
# sysctl -w net.ipv4.tcp_tw_recycle=1
# sysctl -w net.ipv4.tcp_tw_reuse=1
```

FIN-WAIT-2 Socket 超时设置:

```
sysctl -w net.ipv4.tcp_fin_timeout=15
```

4.3 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；

```
## Erlang Process Limit
node.process_limit = 2097152

## Sets the maximum number of simultaneously existing ports for this system
node.max_ports = 1048576
```

4.4 EMQ X 消息服务器参数

设置 TCP 监听器的 Acceptor 池大小，最大允许连接数。配置文件 emqx/etc/emqx.conf:

```
## TCP Listener
listener.tcp.external = 0.0.0.0:1883
listener.tcp.external.acceptors = 64
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:

```
sysctl -w net.ipv4.ip_local_port_range="500 65535"
echo 1000000 > /proc/sys/fs/nr_open
ulimit -n 100000
```

emqtt_bench

并发连接测试工具: http://github.com/emqx/emqtt_bench