

# 第四章 EMQ的高级功能使用(二)

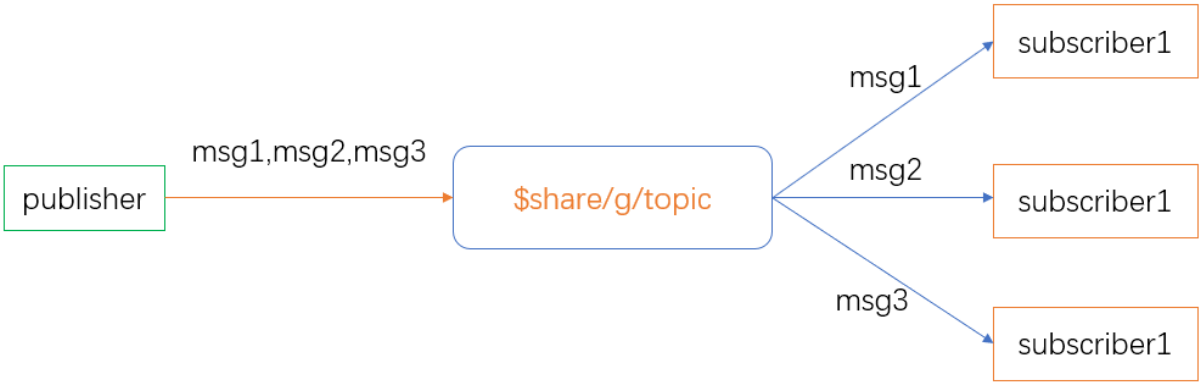
## 学习目标:

- 目标1：能够说出共享订阅的概念和两种共享订阅的方式，完成共享订阅的案例
- 目标2：能够说出延迟发布的概念以及完成延迟发布的案例
- 目标3：能够说出代理订阅的概念及如何配置，完成基于webhook和监控管理api的方式实现动态代理订阅
- 目标4：能够说出主题重写的概念以及如何配置主题重写的规则，完成主题重写的案例
- 目标5：能够说出EMQ X黑名单的作用，以及通过监控管理api去管理黑名单
- 目标6：能够说出EMQ X速率限制的原理以及如何配置

## 1.共享订阅

### 1.1 简介

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，3 个 subscriber 用共享订阅的方式订阅了同一个主题 `$share/g/topic`，其中 `topic` 才是它们订阅的真实主题名，而 `$share/g/` 只是共享订阅前缀。

EMQ X 支持两种格式的共享订阅前缀：

示例	前缀	真实主题名
<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>

注意：共享订阅的主题格式是针对订阅端来指定的，例如：`$share/g/t/a`；而消息的发布方是向主题：`t/a` 发布消息。这样在订阅方才能达到负载均衡的效果。

### 应用场景举例说明：

某智能售货机平台下在全国有50万台售货机设备，在实际运营过程中平均每秒中会收到5万台设备上报过来的出货结果数据，假如用普通主题订阅来处理的话，消息的消费节点会有5万的并发，极有可能会导致该节点宕机，造成出货数据的丢失，对后续结算等业务操作造成极大困扰；如果只是简单增加消费节点的话也无法解决该问题，因为每个节点都会收到所有同样的数据，在这种业务场景下我们，我们希望通过增加消费节点并且节点之间是分摊消息的消费，以此来增强整个系统的负载能力和可用性，那我们就可以通过共享订阅来满足这种业务场景。

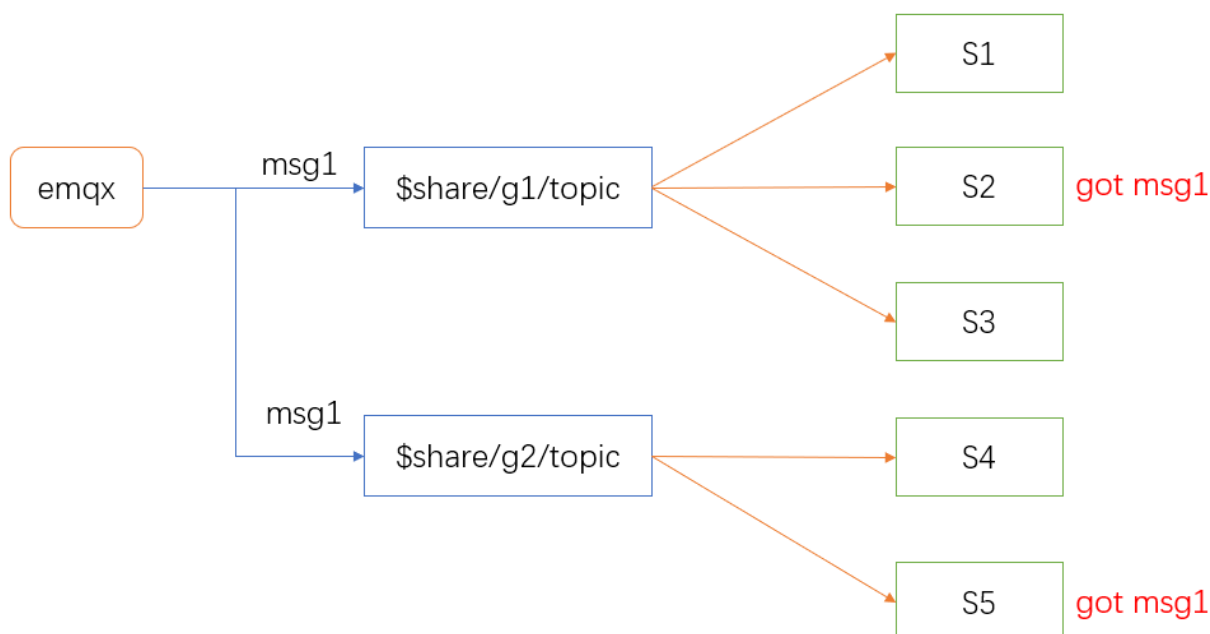
## 1.2 带群组的共享订阅

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

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

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

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



### 案例演示：


启动虚拟主机129上基于docker的EMQ X 服务，启动之前的项目 `emq-demo` 因为有一些webhook接口，启动之前放开所有用户发布订阅的ACL权限，在 `AuthController` 中的acl方法返回200状态码

1：打开两个MQTT X客户端工具来分别订阅 `$share/g1/t1/a` 主题下的消息：

创建第一个客户端：使用 `emq-client2/123456`

## 基础

\* 名称

\* Client ID  

\* 服务器地址

\* 端口

用户名

密码

SSL/TLS ☐ true ☒ false

## 添加订阅

## 添加订阅



\* Topic

\* QoS

标记



取消

确定

创建第二个的客户端：使用 emq-client3/123456


[返回](#)

新建

[连接](#)

#### 基础

\* 名称

\* Client ID  

\* 服务器地址

\* 端口

用户名

密码

SSL/TLS ☐ true ☒ false

#### 添加订阅

### 添加订阅



\* Topic

\* QoS

标记



取消

确定

2: 打开Dashboard, 开启认证组件 emqx\_auth\_username, 利用user/123456连接

Host

192.168.200.129

Port

8083

Path

/mqtt

Client ID

mqttjs\_d0a28afe26

Username

user

Password

123456

Keep Alive

60

☒ Clean Session

☐ SSL

ws://192.168.200.129:8083/mqtt

Connect

Disconnect

Current State: DISCONNECTED

利用里面的websocket来向一个主题 `t1/a` 发布3个消息：

Topic

t1/a

Messages

{ "msg": "共享消息" }

QoS

2

☐ Retained

send

Messages already sent

Messages received

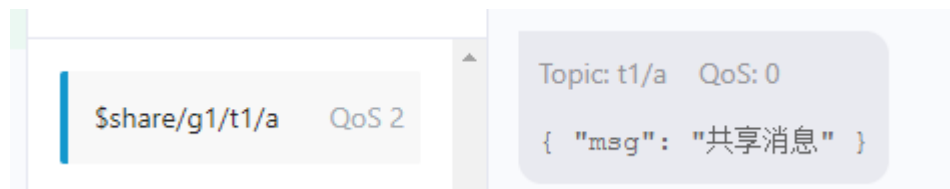
Messages	Topic	QoS	Time	Messages	Topic	QoS	Time
{ "msg": "共享消息" }	t1/a	2	2020-06-10 17:01:00	No Data			
{ "msg": "共享消息" }	t1/a	2	2020-06-10 17:00:58				
{ "msg": "共享消息" }	t1/a	2	2020-06-10 17:00:57				

3：查看客户端收到的消息

第一个客户端收到消息的结果如下



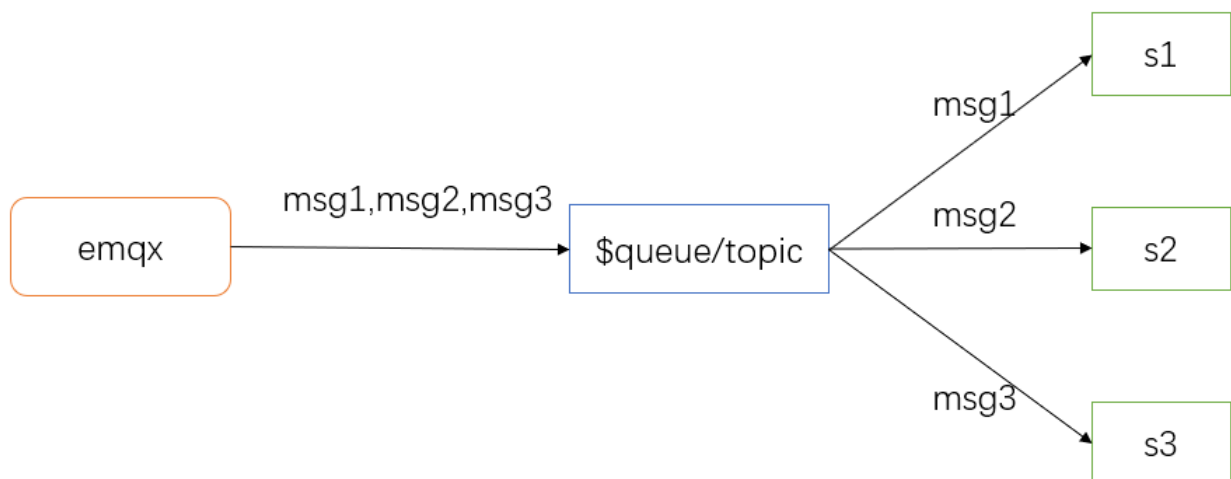
第二个客户端收到的消息结果如下：



我们可以发现发布方发布的消息被订阅者随机的收到了，已经满足了一种负载均衡的策略。

### 1.3 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：



演示：

两个客户端分别添加对 `$queue/t1/a` 的订阅

The screenshot shows two MQTT Explorer client windows. Both clients are subscribed to the topic `$queue/t1/a` with a QoS of 2. The top client window shows three received messages, each with a timestamp of 2020-06-10 17:33:27. The bottom client window shows one received message with a timestamp of 2020-06-10 17:21:22. All messages have the same payload: `{ "msg": "不带群组共享消息" }`.

我们还是在Dashboard的Websocket工具里发布消息：

The screenshot shows the MQTT Dashboard Websocket tool interface. At the top, the 'Topic' is `t1/a`, the 'Messages' field contains `{ "msg": "共享消息" }`, and the 'QoS' is set to 2. There is a 'Retained' checkbox and a 'send' button. Below this, there are two sections: 'Messages already sent' and 'Messages received'. The 'Messages already sent' section shows a list of three messages sent to the topic `t1/a` with QoS 2, all with the same payload `{ "msg": "共享消息" }` and timestamps around 2020-06-10 17:00:00. The 'Messages received' section shows 'No Data'.

查看客户端消息的接收情况

## 1.4 均衡策略与派发 Ack 配置

EMQ X 的共享订阅支持均衡策略与派发 Ack 配置：

```
1  # etc/emqx.conf
2
3  # 均衡策略
4  ## Dispatch strategy for shared subscription
5  ##
6  ## Value: Enum
7  ## - random
8  ## - round_robin
9  ## - sticky
10 ## - hash
11 broker.shared_subscription_strategy = random
12
13 # 共享分发时是否需要 ACK，适用于 QoS1 QoS2 消息，启用时，当通过shared_subscription_strategy选中的
    一个订阅者离线时，应该允许将消息发送到组中的另一个订阅者
14 broker.shared_dispatch_ack_enabled = false
```

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

#### 特别提示：

无论是单客户端订阅还是共享订阅都需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。

## 2.延迟发布

### 2.1 简介

EMQ X 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/{DelayInteval}` 发布消息到 EMQ X 时，将触发延迟发布功能。延迟发布的功能是针对消息发布者而言的，订阅方只需要按照正常的主题订阅即可。

应用场景举例说明：

某智能售货机平台在双十一当天要对设备中所有商品做5折销售，双十一过去之后要立马恢复原价，为了满足这样的场景，我们可以在双十一0点给所有设备发送两条消息，一条消息是通过正常的主题发送，消息内容打5折；第二条消息延迟消息，延迟24小时，消息内容是恢复原价。这样在一个实现中可以完成两个业务场景。

延迟发布主题的具体格式如下：



1 `$delayed/{DelayInterval}/{TopicName}`

- `$delayed`: 使用 `$delayed` 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, EMQ X 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`。

此功能由 `emqx-delay-publish` 插件提供, 该插件默认关闭, 需要开启插件后才能使用此功能。

## 2.2 案例演示

1: 直接在Dashboard上订阅主题: `t2/a`

Subscribe

Topic

QoS

2

▼

Subscribe

Subscribe

Topic	QoS	Time	Op
t2/a	2	2020-06-10 17:46:31	×

然后在Dashboard上向主题: `$delayed/10/t2/a` 发送消息

Messages

Topic

Messages

QoS

2

▼

☐ Retained

send

Messages already sent

Messages received

Messages	Topic	QoS	Time	Messages	Topic	QoS	Time
{ "msg": "延迟消息!" }	\$delayed/10/t2/a	2	2020-06-10 17:58:00	{ "msg": "延迟消息!" }	t2/a	2	2020-06-10 17:58:11

## 3.代理订阅

EMQ X 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。

应用场景举例说明：

某冷链平台下有大量的冷链运输车，前期车上只配备了温度传感器，将车内的温度数据通过物联网终端经过EMQ定期上报给指标采集系统，由该系统将相关数据透传到一个大数据分析平台，用来分析这些温度数据，后来车上又配置了湿度传感器，这些湿度信息也会上报到EMQ，大数据平台也需要分析这些湿度信息，此时我们的指标采集系统就需要再单独开发一套订阅湿度信息的代码，如果我们采用代理订阅的话就无需再开发这块代码，只需经过简单的配置就可以增加这个功能了。

### 3.1 内置代理订阅

#### 3.1.1 开启代理订阅

EMQ X 通过内置代理订阅模块就可以通过配置文件来指定代理订阅规则从而实现代理订阅，适用于有规律可循的静态的代理订阅需求。

代理订阅功能默认关闭，开启此功能需要修改 `etc/emqx.conf` 文件中的 `module.subscription` 配置项。默认 `off` 表示关闭，如需开启请修改为 `on`。

```
1 module.subscription = off
```

#### 3.1.2 配置代理订阅规则

仅仅开启并不意味代理订阅已经工作，你还需要配置相应的规则，EMQ X 的代理订阅规则支持用户自行配置，用户可以自行添加多条代理订阅规则，每条代理订阅规则都需要指定 Topic 和 QoS，规则的数量没有限制，代理订阅规则的格式如下：

```
1 module.subscription.<number>.topic = <topic>
2 module.subscription.<number>.qos = <qos>
```

在配置代理订阅的主题时，EMQ X 提供了 `%c` 和 `%u` 两个占位符供用户使用，EMQ X 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，在 `etc/emqx.conf` 文件中添加以下代理订阅规则：

```
1 module.subscription.1.topic = client/%c
2 module.subscription.1.qos = 1
3
4 module.subscription.2.topic = user/%u
5 module.subscription.2.qos = 2
6
7 module.subscription.3.topic = testtopic/#
8 module.subscription.3.qos = 2
```

当一个客户端连接 EMQ X 的时候, 假设客户端的 Client ID 为 testclient, Username 为 tester, 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/testclient 和 QoS 为 2 的 user/tester 这两个主题。

这些配置都是基于配置文件实现的**静态代理订阅**, 开源版本的EMQ X目前只支持这种静态代理订阅, 收费的EMQ X Enterprise 版本中支持**动态代理订阅**, 通过外部数据库设置主题列表在设备连接时读取列表实现代理订阅。

但是我们也可以结合使用Webhook和管理监控的API对处于连接状态的客户端实现动态的订阅和取消订阅的操作。比如我们在客户端连接上来之后通过程序调用API来实现为客户端自动订阅主题。

## 3.2 基于Webhook和API实现动态代理订阅

### 3.2.1 动态代理订阅需求

客户端连接上来之后动态订阅主题, 客户端下线后自动取消订阅

### 3.2.2 代码实现

**实现思路分析:**

开启了 emqx\_web\_hook 组件后, EMQ X的事件都会勾起对我们配置的webhook接口进行回调, 在该webhook接口中我们能够获取客户端的相关信息比如 clientId,username 等, 然后我们可以在该接口方法中针对该客户端自动订阅某一主题, 订阅的实现我们基于EMQ X给我们提供的监控管理的相关HTTP API, 意味着我们调用相关的HTTP API可完成客户端订阅的功能, 相关的HTTP API可在Dashboard中查看, 也可以在官方的产品文档中查找: <https://docs.emqx.io/broker/latest/cn/advanced/http-api.html>

反之客户端下线时我们可以自动取消订阅。

**实现过程:**

(1) 在 emq-demo 项目 WebHookController 中添加一个自动订阅的方法

```
1  /**
2      * 自动订阅或取消订阅
3      * @param clientId
4      * @param topicfilter
5      * @param qos
6      * @param sub
7      */
8  private void autoSub(String clientId,String topicfilter, QosEnum qos,boolean sub){
9      RestTemplate restTemplate = new RestTemplateBuilder()
10         .basicAuthentication("admin", "public")
11         .defaultHeader(HttpHeaders.CONTENT_TYPE,
org.springframework.http.MediaType.APPLICATION_JSON_VALUE)
12         .build();
13      //装配参数
14      Map param = new HashMap();
15      param.put("clientId",clientId);
16      param.put("qos",qos.value());
17      param.put("topic",topicfilter);
18      log.info("请求emq的相关参数:{}",param);
19      HttpHeaders headers = new HttpHeaders();
20      headers.setContentType(org.springframework.http.MediaType.APPLICATION_JSON);
```

```

21     HttpEntity<Object> entity = new HttpEntity<Object>(param,headers);
22     //自动订阅
23     if(sub){
24         new Thread()->{
25             ResponseEntity<String> responseEntity =
restTemplate.postForEntity("http://192.168.200.129:8081/api/v4/mqtt/subscribe", entity,
String.class);
26             log.info("自动订阅的结果:{}",responseEntity.getBody());
27         }.start();
28         return;
29     }
30     //自动取消订阅
31     ResponseEntity<String> responseEntity =
restTemplate.postForEntity("http://192.168.200.129:8081/api/v4/mqtt/unsubscribe", entity,
String.class);
32     log.info("自动取消订阅的结果:{}",responseEntity.getBody());
33 }

```

(2) 在 `hook` 方法中客户端连接上来之后调用该方法:

```

1  @PostMapping("/webhook")
2  public void hook(@RequestBody Map<String,Object> params){
3      log.info("emqx 触发 webhook,请求体数据={}",params);
4      String action = (String) params.get("action");
5      String clientId = (String) params.get("clientId");
6      if(action.equals("client_connected")){
7          //客户端成功接入
8          clientStatusMap.put(clientId,true);
9          //自动订阅autosub主题
10         autoSub(clientId,"autosub/#",QosEnum.QoS2,true);
11     }
12     if(action.equals("client_disconnected")){
13         //客户端断开连接
14         clientStatusMap.put(clientId,false);
15         //自动取消订阅autosub主题
16         autoSub(clientId,"autosub/#",QosEnum.QoS2,false);
17     }
18 }

```

(3) 启动 `emq-demo` 项目

(4) 用MQTTX客户端工具创建一个客户端, 使用: `emq-client2/123456`

[< 返回](#)[编辑](#)[连接](#)

### 基础

\* 名称

\* Client ID

[C](#)

\* 服务器地址

\* 端口

用户名

密码

SSL/TLS ☐ true ☒ false

查看控制台输出：

```
1 2020-06-11 16:30:29.312 INFO 21384 --- [ Thread-11]
  c.i.controller.mqtt.WebHookController : 请求emq的相关参数:{clientid=mqttx_7a3a7733, qos=2,
  topic=autosub/#}
2
3 2020-06-11 16:30:29.323 INFO 21384 --- [ Thread-11]
  c.i.controller.mqtt.WebHookController : 自动订阅的结果:{"code":0}
```

(5) 直接在该客户端上发送消息，如下：

Payload:

QoS:

Retain: ☐

autosub/123

```
{
  "msg": "fadsfadsfa"
}
```



(6) 查看MQTTX客户端上是否接收到数据

## 4. 主题重写

## 4.1 简介

EMQ X 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。

应用场景举例说明：

某共享单车平台A运营着大量的共享单车，每个单车上都装有一个物联网终端芯片，芯片上的程序是将一些数据通过mqtt协议上报到EMQ服务器；该公司某一天收购了另一家共享单车平台B，B平台下原有的单车也是通过mqtt上报消息数据，但是消息主题跟A平台的不一样，如果A平台想接入B平台的车上报的数据，我们就需要把B平台下所有车上芯片程序更改一下，这样虽然可行但是会耗费大量的人力物力成本，这时我们通过主题重写就可以实现B平台下所有单车数据的接收，几乎不需要编码，成本非常低。

EMQ X 的**保留消息**和**延迟发布**可以与主题重写配合使用，例如，当用户想使用延迟发布功能，但不方便修改客户端发布的主题时，可以使用主题重写将相关主题重写为延迟发布的主题格式。

主题重写功能默认关闭，开启此功能需要修改 `etc/emqx.conf` 文件中的 `module.rewrite` 配置项。默认 `off` 表示关闭，如需开启请修改为 `on`。

```
1 module.rewrite = off
```

## 4.2 配置主题重写规则

EMQ X 的主题重写规则需要用户自行配置，用户可以自行添加多条主题重写规则，规则的数量没有限制，但由于任何携带主题的 MQTT 报文都需要匹配一遍重写规则，因此此功能在高吞吐场景下带来的性能损耗与规则数量是成正比的，用户需要谨慎地使用此功能。

每条主题重写规则的格式如下：

```
1 module.rewrite.rule.<number> = 主题过滤器 正则表达式 目标表达式
```

每条重写规则都由以空格分隔的主题过滤器、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，EMQ X 在收到诸如 PUBLISH 报文等带有主题的消息时，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 `$N` 这种格式的变量匹配正则表达式中提取出来的元素，`$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，EMQ X 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，EMQ X 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。

## 4.3 主题重写配置

假设 `etc/emqx.conf` 文件中已经添加了以下主题重写规则：

```

1 module.rewrite.rule.1 = y/+/z/# ^y/(.+)/z/(.+) $ y/z/$2
2 module.rewrite.rule.2 = x/# ^x/y/(.+) $ z/y/x/$1
3 module.rewrite.rule.3 = x/y/+ ^x/y/(\d+) $ z/y/$1

```

正则表达式解析：

`^` 匹配输入字符串的开始位置，除非在方括号表达式中使用，当该符号在方括号表达式中使用时，表示不接受该方括号表达式中的字符集合

`$` 匹配输入字符串的结尾位置

`( )` 表示一个标记一个子表达式的开始和结束位置，

`[ ]` 标记一个中括号表达式的开始

`.` 匹配除换行符 `\n` 之外的任何单字符，

`+` 匹配前面的子表达式一次或多次

`*` 匹配前面的子表达式零次或多次

`?` 匹配前面的子表达式零次或一次

`|` 指明两项之间的一个选择

`{n}` `n` 是一个非负整数。匹配确定的 `n` 次

`{n,}` `n` 是一个非负整数。至少匹配 `n` 次

`{n,m}` `m` 和 `n` 均为非负整数，其中 `n <= m`。最少匹配 `n` 次且最多匹配 `m` 次

`\d` 匹配一个数字字符。等价于 `[0-9]`

此时我们分别订阅 `y/a/z/b`、`y/def`、`x/1/2`、`x/y/2`、`x/y/z` 五个主题：

- `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。
- `y/a/z/b` 匹配 `y/+/z/#` 主题过滤器，EMQ X 执行 `module.rewrite.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。
- `x/1/2` 匹配 `x/#` 主题过滤器，EMQ X 执行 `module.rewrite.rule.2` 规则，通过正则表达式未匹配到元素，不执行主题重写，实际订阅 `x/1/2` 主题。
- `x/y/2` 同时匹配 `x/#` 和 `x/y/+` 两个主题过滤器，EMQ X 通过倒序读取配置，所以优先匹配 `module.rewrite.rule.3`，通过正则替换，实际订阅了 `z/y/2` 主题。
- `x/y/z` 同时匹配 `x/#` 和 `x/y/+` 两个主题过滤器，EMQ X 通过倒序读取配置，所以优先匹配 `module.rewrite.rule.3`，通过正则表达式未匹配到元素，不执行主题重写，实际订阅 `x/y/z` 主题。需要注意的是，即使 `module.rewrite.rule.3` 的正则表达式匹配失败，也不会再次去匹配 `module.rewrite.rule.2` 的规则。

验证：

我们在 `emqx.conf` 配置文件中添加如下配置：

```
1  ## Rewrite Module
2
3  ## Enable Rewrite Module.
4  ##
5  ## Value: on | off
6  module.rewrite = on
7
8  ## {rewrite, Topic, Re, Dest}
9  module.rewrite.rule.1 = x/# ^x/y/(.+)$ z/y/$1
10 ## x/y/1 z/y/1
11 ## module.rewrite.rule.2 = y/+/z/# ^y/(.+)/z/(.+)$ y/z/$2
```

(1)：打开一个MQTTX客户端，使用 `user/123456`

[返回](#)

新建

[连接](#)

### 基础

\* 名称

\* Client ID



\* 服务器地址

\* 端口

用户名

密码

SSL/TLS ☐ true ☒ false

我们向 `x/y/1` 主题发送消息，根据正则我们需要去订阅主题： `z/y/1`，根据正则



z/y/1

QoS 2

```
"msg": "helloworld"
}
```

2020-06-11 18:48:21

Topic: z/y/1 QoS: 2

```
{
  "msg": "helloworld"
}
```

2020-06-11 18:48:21

Payload:

JSON

QoS:

2

Retain: ☐

x/y/1

```
{
  "msg": "helloworld"
}
```

## 5. 黑名单

### 5.1 简介

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

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，我们需要使用认证功能来实现。

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

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

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

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

用户可以在 `emqx.conf` 配置文件中调整触发阈值和封禁时长等配置：

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

此配置项的值以 `,` 分隔，依次表示客户端离线次数，检测的时间范围以及封禁时长，因此上述默认配置即表示如果客户端在 1 分钟内离线次数达到 30 次，那么该客户端使用的客户端标识符将被封禁 5 分钟。

## 5.2 黑名单实现

我们通过HTTP API来实现将clientid为 `emqx-demo` 的客户端加入黑名单，该API具体的参数如下：

### 5.2.1 获取黑名单

**GET /api/v4/banned**

获取黑名单

**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

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，小于等于查找
_lte_created_at	Integer	False	客户端会话创建时间，大于等于查找
_gte_connected_at	Integer	False	客户端连接创建时间，小于等于查找
_lte_connected_at	Integer	False	客户端连接创建时间，大于等于查找

#### Success Response Body (JSON):

Name	Type	Description
code	Integer	0
data	Array	由对象构成的数组，对象中的字段与 <code>POST</code> 方法中的 Request Body 相同
meta	Object	同 <code>/api/v4/clients</code>
meta.page	Integer	页码
meta.limit	Integer	每页显示的数据条数
meta.count	Integer	数据总条数

在VSCode上进行查询：

```
1 #####获取黑名单列表#####
2 GET http://{{hostname}}:{{port}}/api/v4/banned HTTP/1.1
3 Content-Type: {{contentType}}
4 Authorization: Basic {{userName}}:{{password}}
```

## 5.2.2 添加黑名单

### POST /api/v4/banned

将对象添加至黑名单

Parameters (json):

Name	Type	Required	Default	Description
who	String	必填		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	必填		用于区分黑名单对象类型，可以是 <code>clientid</code> ， <code>username</code> ， <code>peerhost</code>
by	String	可选	user	指示该对象被谁添加至黑名单
at	Integer	可选	当前系统时间	添加至黑名单的时间，单位：秒
until	Integer	可选	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

案例：

```
1 #####添加黑名单#####
2 POST http://{{hostname}}:{{port}}/api/v4/banned HTTP/1.1
3 Content-Type: {{contentType}}
4 Authorization: Basic {{userName}}:{{password}}
5
6 {
7     "who": "emq-demo",
8     "as": "clientid"
9 }
```

之后用客户端工具测试下clientid为 `emo-demo` 连接，发现已经无法连接到EMQ X服务器了。

[< 返回](#)**新建**[连接](#)**基础**

\* 名称

\* Client ID



\* 服务器地址

\* 端口

用户名

密码

SSL/TLS ☐ true ☒ false

**黑名单测试**

\* 名称

\* Client ID



Error: Connection refused: Not authorized



密码

Keep Alive

☒ Clean Session

▶ 连接

+ 添加订阅



全部

已接收

已发送

**5.2.3 删除黑名单**

**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

案例：

```
1 #####删除黑名单#####
2 DELETE http://{hostname}://{port}/api/v4/banned/clientid/emq-demo HTTP/1.1
3 Content-Type: {{contentType}}
4 Authorization: Basic {{userName}}:{{password}}
```

删除完成后客户端再次连接，看是否能够连接上

## 6. 速率限制

### 6.1 速率限制简介和配置

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

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

*backpressure* ：背压，反向压力

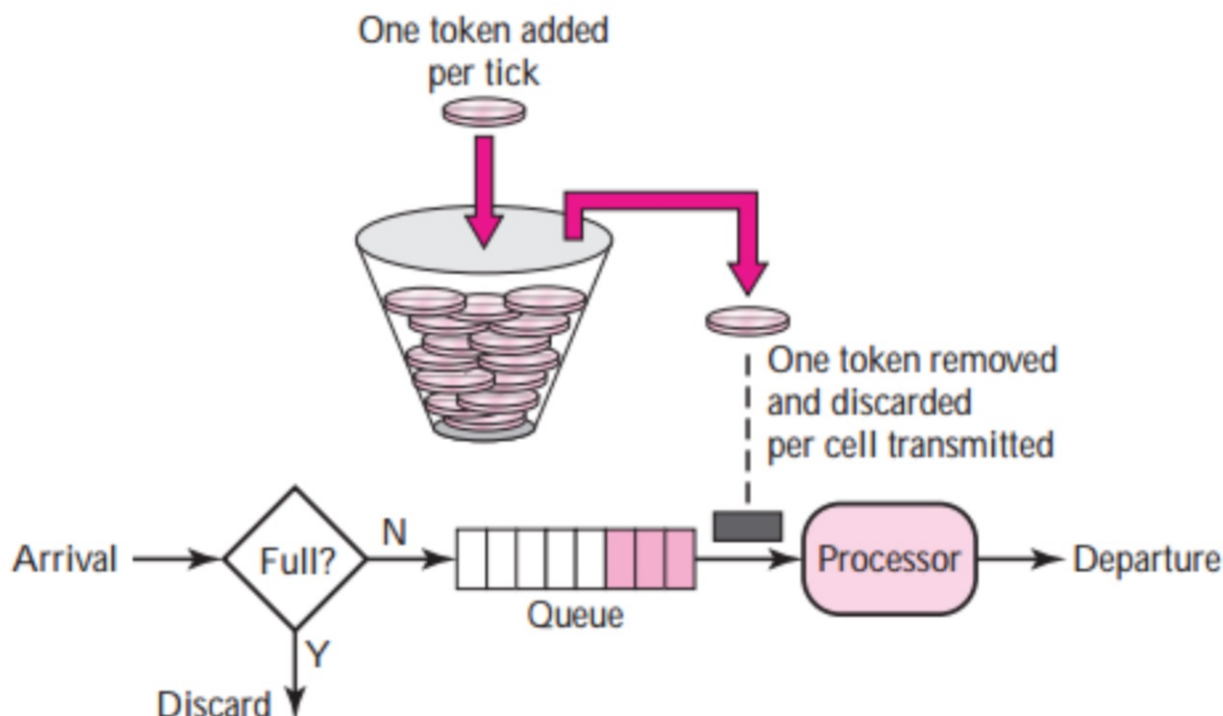
配置项	类型	默认值	描述
listener.tcp.external.max_conn_rate	Number	1000	本节点上允许的最大连接速率 (conn/s)
zone.external.publish_limit	Number,Duration	无限制	单连接上允许的最大发布速率 (msg/s)
listener.tcp.external.rate_limit	Size,Duration	无限制	单连接上允许的最大报文速率 (bytes/s)

- **max\_conn\_rate** 是单个 emqx 节点上连接建立的速度限制。 `1000` 代表每秒最多允许 1000 个客户端接入。
- **publish\_limit** 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。
- **rate\_limit** 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。

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

## 6.2 速率限制原理

EMQ X 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  $M = b + r$

## 6.3 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 listener.tcp.external.rate_limit = 100KB,10s
```

EMQ X 将使用两个值初始化每个连接的 rate-limit 处理器：

- $\text{rate} = 100 \text{ KB} / 10\text{s} = 10240 \text{ B/s}$
- $\text{burst} = 100 \text{ KB} = 102400 \text{ B}$

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  $10240 \text{ B/s}$

- 允许的峰值速率为  $102400 + 10240 = 112640$  B/s

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

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