第三章 高级功能使用(一)

学习目标：

目标1：能够说出发布订阅ACL的作用，具体规则是什么，授权结果有哪几种，如何进行全局配置

目标2：能够说出超级用户有什么特权，ACL缓存和ACL链的具体含义是什么

目标3：能够说出HTTP ACL的授权原理，完成HTTP ACL的案例

目标4：能够简要说出EMQ X Webhook的作用，完成利用Webhook实现客户端断连监控的案例

目标5：能够说出EMQ X集群的概念和作用，路由表/订阅表/主题树的概念，完成使用manual搭建集群的案例

目标6：能够说出EMQ X管理监控相关API的大致作用

目标7：能够说出保留消息的作用，以及如何配置

1.发布订阅ACL

1.1 发布订阅ACL简介

发布订阅ACL是指对发布(publish)/订阅(subscribe)操作的权限控制。例如拒绝用户`emq-demo`向`testTopic/a`主题发布消息。 EMQ X 支持通过客户端发布订阅 ACL 进行客户端权限的管理。

1.1.1 ACL 插件

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

连接数据源、进行访问控制功能是通过插件实现的，使用前需要启用相应的插件。

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

配置文件

- 内置 ACL

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 ACL
- PostgreSQL ACL
- Redis ACL
- MongoDB ACL

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 ACL

HTTP ACL 能够实现复杂的 ACL 管理。

> 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，

1.1.2 ACL规则详解

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

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

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

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

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

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

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

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

{allow, all}.
```

1.1.3 授权结果

任何一次 ACL 授权最终都会返回一个结果：

- 允许：经过检查允许客户端进行操作
- 禁止：经过检查禁止客户端操作
- 忽略（ignore）：未查找到 ACL 权限信息（no match），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

1.1.4 全局配置

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

通过 `etc/emqx.conf` 中的 ACL 配置可以更改该属性：

```
# etc/emqx.conf

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

此处我们需要修改全局配置文件中关于acl的配置，将`acl_nomatch`配置项的值改为：`deny`

完成配置后使用`emqx restart`重启emqx broker服务

配置默认 ACL 文件，使用文件定义默认 ACL 规则：

```
# etc/emqx.conf

acl_file = etc/acl.conf
```

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

```
# etc/emqx.conf

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

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

1.1.5 超级用户

客户端在进行认证的时候客户端可拥有“超级用户”身份，超级用户拥有最高权限不受 ACL 限制。

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

1.1.6 ACL缓存

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

在 `etc/emqx.conf` 可以配置 ACL 缓存大小与缓存时间：

```
# etc/emqx.conf

## 是否启用
enable_acl_cache = on

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

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

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，清除缓存需要需要使用EMQ X Broker提供的监控管理的HTTP API

查询指定客户端的 ACL 缓存：`GET /api/v4/clients/{clientid}/acl_cache`

```
###############查询指定客户端的 ACL 缓存#####################
GET  http://{{hostname}}:{{port}}/api/v4/clients/emq-client1/acl_cache HTTP/1.1
Content-Type: {{contentType}}
Authorization: Basic {{userName}}:{{password}}
```

清除指定客户端的ACL缓存：`DELETE /api/v4/clients/{clientid}/acl_cache`

```
###############清除指定客户端的 ACL 缓存#####################
DELETE  http://{{hostname}}:{{port}}/api/v4/clients/emq-client1/acl_cache HTTP/1.1
Content-Type: {{contentType}}
Authorization: Basic {{userName}}:{{password}}
```

测试时注意开启对应的插件

1.1.7 ACL 鉴权链

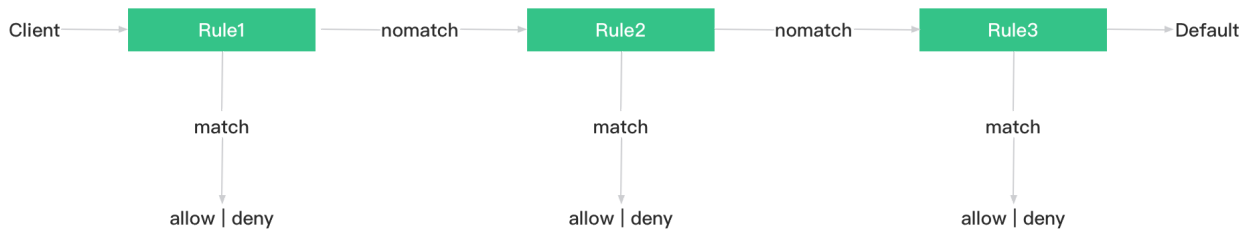
当同时启用多个 ACL 插件时，EMQ X 将按照插件开启先后顺序进行链式鉴权：

- 一通过授权，终止链并允许客户端通过验证
- 一旦授权失败，终止链并禁止客户端通过验证
- 直到最后一个 ACL 插件仍未通过，根据

  默认授权

  配置判定

  - 默认授权为允许时，允许客户端通过验证
  - 默认授权为禁止时，禁止客户端通过验证



同时只启用一个 ACL 插件可以提高客户端 ACL 检查性能。

1.2 内置ACL

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。

ACL 规则文件：

```
etc/acl.conf
```

内置 ACL 优先级最低，可以被 ACL 插件覆盖，如需禁用全部注释即可。规则文件更改后需重启 EMQ X 以应用生效。

1.2.1 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：

```
%% 允许 "dashboard" 用户 订阅 "$SYS/#" 主题
{allow, {user, "dashboard"}, subscribe, ["$SYS/#"]}.

%% 允许 IP 地址为 "127.0.0.1" 的用户 发布/订阅 "#SYS/#", "#" 主题
{allow, {ipaddr, "127.0.0.1"}, pubsub, ["$SYS/#", "#"]}.

%% 拒绝 "所有用户" 订阅 "$SYS/#" "#" 主题
{deny, all, subscribe, ["$SYS/#", {eq, "#"}]}.

%% 允许其它任意的发布订阅操作
{allow, all}.
```

1.2.2 acl.conf 编写规则

文件中的**规则按书写顺序从上往下匹配。注意，匹配是从上往下，规则加载是从下往上加载**

`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
  - `{clientid, "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

在 `acl.conf` 修改完成后，并不会自动加载至 EMQ X 系统。需要手动执行：

```
./bin/emqx_ctl acl reload
```

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，需要使用认证插件。

1.3 HTTP ACL

HTTP 认证使用外部自建 HTTP 应用认证授权数据源，根据 HTTP API 返回的数据判定授权结果，能够实现复杂的 ACL 校验逻辑。

插件：

```
emqx_auth_http
```

> 注意：emqx_auth_http 插件同时包含认证功能，可通过注释禁用。

1.3.1 ACL授权原理

EMQ X 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 4xx 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 ignore

1.3.2 HTTP 请求信息

要启用 HTTP ACL，需要在 `etc/plugins/emqx_auth_http.conf` 中配置

与认证HTTP请求相同的HTTP API 基础请求信息，配置证书、请求头与重试规则。

```
# etc/plugins/emqx_auth_http.conf

## 启用 HTTPS 所需证书信息
## auth.http.ssl.cacertfile = etc/certs/ca.pem

## auth.http.ssl.certfile = etc/certs/client-cert.pem

## auth.http.ssl.keyfile = etc/certs/client-key.pem

## 请求头设置
## auth.http.header.Accept = */*

## 重试设置
auth.http.request.retry_times = 3

auth.http.request.retry_interval = 1s


auth.http.request.retry_backoff = 2.0
```

进行发布、订阅认证时，EMQ X 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。

1.3.3 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。

```
# etc/plugins/emqx_auth_http.conf
####使用vi编辑该配置，修改URL请求地址

##--------------------------------------------------------------------
## Superuser request.
##
## Variables:
##  - %u: username
##  - %c: clientid
##  - %a: ipaddress
##  - %r: protocol
##  - %P: password
##  - %p: sockport of server accepted
##  - %C: common name of client TLS cert
##  - %d: subject of client TLS cert
##
## Value: URL 请求地址
auth.http.super_req = http://192.168.200.10:8991/mqtt/superuser
## Value: post | get | put 请求方法
auth.http.super_req.method = post
## Value: Params 请求参数
auth.http.super_req.params = clientid=%c,username=%u
```

1.3.4 ACL 授权查询请求

```
# etc/plugins/emqx_auth_http.conf
####使用vi编辑该配置，修改URL请求地址以及请求方式

##--------------------------------------------------------------------
## ACL request.
##
## Variables:
##  - %A: 1 | 2, 1 = sub, 2 = pub
##  - %u: username
##  - %c: clientid
##  - %a: ipaddress
##  - %r: protocol
##  - %m: mountpoint
##  - %t: topic
##
## Value: URL
auth.http.acl_req = http://192.168.200.10:8991/mqtt/acl

## Value: post | get | put
```

```
auth.http.acl_req.method = post
## Value: Params
auth.http.acl_req.params =
access=%A,username=%u,clientid=%c,ipaddr=%a,topic=%t,mountpoint=%m
```

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 EMQ X 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
- %m：topic的安装点，是桥接的连接属性

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。

1.3.5 HTTP ACL接口开发

在原有的项目`emq-demo`中我们已经开发了基于HTTP API的认证Controller：`AuthController`，按照我们的请求URL配置，我们需要在该Controller中添加两个接口方法，一个是用于查询superuser的，一个是用于进行ACL授权查询的，这两个方法分别如下：

1：查询客户端是否为超级用户，

```
@PostMapping("/superuser")
public ResponseEntity<?> superUser(@RequestParam("clientid") String clientid,
                                   @RequestParam("username") String username){
    log.info("emqx 查询是否是超级用户,clientid={},username={}",clientid,username);
    if(clientid.contains("admin") || username.contains("admin")){
        log.info("用户{}是超级用户",username);
        //是超级用户
        return new ResponseEntity<Object>(HttpStatus.OK);
    }else {
         log.info("用户{}不是超级用户",username);
        //不是超级用户
        return new ResponseEntity<Object>(HttpStatus.UNAUTHORIZED);
    }
}
```

注意，我们在初始化方法init中添加一个超级用户：admin/admin

```
@PostConstruct
public void init(){
    users = new HashMap<>();
    users.put("user","123456");
    users.put("emq-client2","123456");
    users.put("emq-client3","123456");
    users.put("admin","admin");
}
```

同理：API返回200状态码代表是超级用户，API返回4XX状态码则不是超级用户

2：ACL 授权查询请求

```
@PostMapping("/acl")
public ResponseEntity acl(@RequestParam("access")int access,
                          @RequestParam("username")String username,
                          @RequestParam("clientid")String clientid,
                          @RequestParam("ipaddr")String ipaddr,
                          @RequestParam("topic")String topic,
                          @RequestParam("mountpoint")String mountpoint){

    log.info("EMQX发起客户端操作授权查询请求,access={},username={},clientid={},ipaddr={},topic={},mountpoint={}",
             access,username,clientid,ipaddr,topic,mountpoint);

    if(username.equals("emq-client2") && topic.equals("testtopic/#") && access == 1){
        log.info("客户端{}有权限订阅{}",username,topic);
        return new ResponseEntity<>(HttpStatus.OK);
    }

    if(username.equals("emq-client3") && topic.equals("testtopic/123") && access == 2){
        log.info("客户端{}有权限向{}发布消息",username,topic);
        return new ResponseEntity<>(null, HttpStatus.OK);
    }

    log.info("客户端{},username={},没有权限对主题{}进行{}操作",clientid,username,topic,access==1?"订阅":"发布");
    return new ResponseEntity(HttpStatus.UNAUTHORIZED);//无权限
}
```

在这个方法中我们设置了：

只有用户名为`emq-client2`的客户端能够去订阅`testtopic/#`的权限其他客户端都不可以

只有用户名为`emq-client3`的客户端能够向`testtopic/123`发布消息的权限其他都没有

1.3.6 HTTP ACL接口测试

1：首先测试超级用户，在Dashboard中打开插件`emqx_auth_http`，其他的认证插件`emqx_auth_clientid`,`emqx_auth_username`可以暂时停掉

使用MQTTX客户端工具：创建一个新的连接，username/password使用超级用户：admin/admin

查看控制台输出：

```
INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: emqx认证组件调用自定义的认证服务开始认证,clientid=admin-client,username=admin,password=admin

INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: emqx 查询是否是超级用户,clientid=admin-client,username=admin

INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: 用户admin是超级用户
```

然后我们用此客户端订阅主题：`testtopic/#`，预期能够订阅成功并且不会发起ACL查询请求

接着我们用此客户端向主题：`testtopic/123`发布消息，预期能够发布成功并且不会发起ACL查询请求

2：创建一个客户端，用户名密码使用：emq-client2/123456，如下

查看服务控制台输出：

```
INFO 3904 --- [nio-8991-exec-3] c.i.controller.mqtt.AuthController: emqx认证组件调用自定义的认证服务开始认证,clientid=mqttx_998150bd,username=emq-client2,password=123456

INFO 3904 --- [nio-8991-exec-2] c.i.controller.mqtt.AuthController: emqx 查询是否是超级用户,clientid=mqttx_998150bd,username=emq-client2

INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: 用户emq-client2不是超级用户
```

然后我们用该客户端订阅主题`testtopic/#`，然后查看服务控制台输出

```
INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: EMQX发起客户端操作授权查询请求,access=1,username=emq-client2,clientid=mqttx_998150bd,ipaddr=192.168.200.10,topic=testtopic/#,mountpoint=undefined

INFO 3904 --- [io-8991-exec-10] c.i.controller.mqtt.AuthController: 客户端emq-client2有权限订阅testtopic/#
```

接着我们用该客户端向主题`testtopic/123`发送消息，查看服务控制台输出

```
INFO 3904 --- [nio-8991-exec-9] c.i.controller.mqtt.AuthController: EMQX发起客户端操作授权查询请求,access=2,username=emq-client2,clientid=mqttx_998150bd,ipaddr=192.168.200.10,topic=testtopic/123,mountpoint=undefined

INFO 3904 --- [nio-8991-exec-9] c.i.controller.mqtt.AuthController: 客户端mqttx_998150bd,username=emq-client2,没有权限对主题testtopic/123进行发布操作
```

此时注意该客户端，因为成功订阅了`testtopic/#`，如果我们的消息发布成功了我们应该能收到消息，事实是没有收到消息，证明消息发布没有成功！

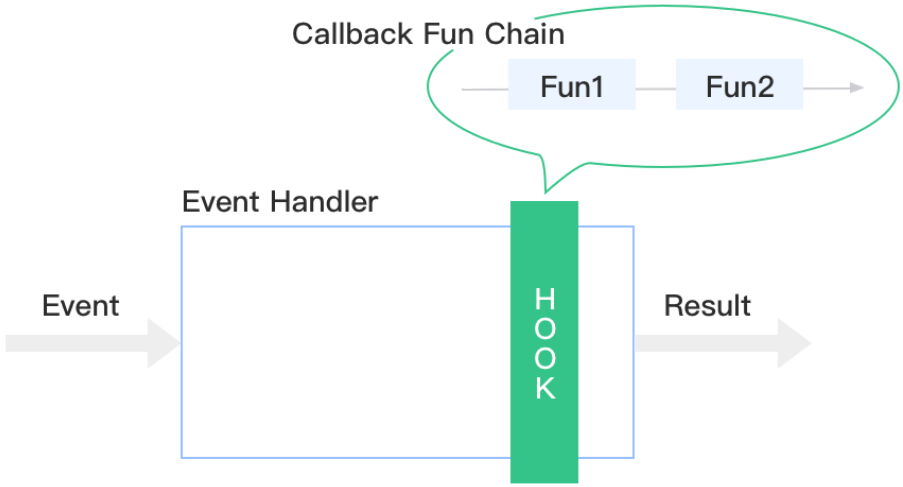
同理：可以在创建一个客户端使用：emq-client3/123456，该客户端只能向`testtopic/123`中发布消息但是不能订阅`testtopic/#`

2. WebHook

2.1 WebHook简介

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

> **钩子(Hooks)** 是 EMQ X 提供的一种机制，它通过拦截模块间的函数调用、消息传递、事件传递来修改或扩展系统功能。
>
> 简单来讲，该机制目的在于增强软件系统的扩展性、方便与其他三方系统的集成、或者改变其系统原有的默认行为。如下图：



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。

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

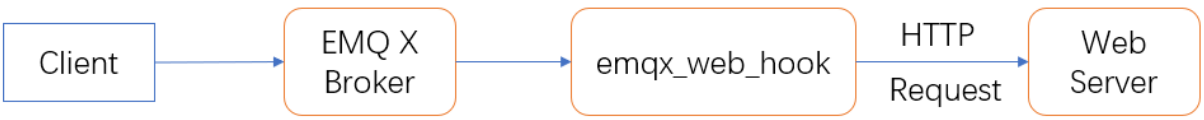
对于实现者来说仅需要关注：

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

如果你是在开发扩展插件中使用钩子，你应该能 **完全地明白这三点，且尽量不要在钩子内部使用阻塞函数，这会影响系统的吞吐。**

WebHook 的内部实现是基于钩子，但它更靠近顶层一些。它通过在钩子上的挂载回调函数，获取到 EMQ X 中的各种事件，并转发至 emqx_web_hook 中配置的 Web 服务器。

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



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

2.2 配置项说明

Webhook 的配置文件位于 `etc/plugins/emqx_web_hook.conf`：

配置项	类型	可取值	默认值	说明
api.url	string	-	http://127.0.0.1:8080	事件需要转发的目的服务器地址
encode_payload	enum	`base64`, `base62`	undefined	对**消息类事件中的 Payload 字段**进行编码，注释或其他则表示不编码

说明：当消息内容是不可见字符（如二进制数据）时，为了能够在 HTTP 协议中传输，使用 encode_payload 是十分有用的。

配置触发规则：

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

```
## 格式示例
web.hook.rule.<Event>.<Number> = <Rule>

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

Event 触发事件:

目前支持以下事件：

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

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，每种事件下规则中的action在`etc/plugins/emqx_web_hooks.conf`文件中有定义

```
web.hook.rule.client.connect.1       = {"action": "on_client_connect"}
web.hook.rule.client.connack.1       = {"action": "on_client_connack"}
web.hook.rule.client.connected.1     = {"action": "on_client_connected"}
web.hook.rule.client.disconnected.1  = {"action": "on_client_disconnected"}
web.hook.rule.client.subscribe.1     = {"action": "on_client_subscribe"}
web.hook.rule.client.unsubscribe.1   = {"action": "on_client_unsubscribe"}
web.hook.rule.session.subscribed.1   = {"action": "on_session_subscribed"}
web.hook.rule.session.unsubscribed.1 = {"action": "on_session_unsubscribed"}
web.hook.rule.session.terminated.1   = {"action": "on_session_terminated"}
web.hook.rule.message.publish.1      = {"action": "on_message_publish"}
web.hook.rule.message.delivered.1    = {"action": "on_message_delivered"}
web.hook.rule.message.acked.1        = {"action": "on_message_acked"}
```

- 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：

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

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`。

2.3 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 `api.url` 所配置的 Web 服务器上。其请求格式为：

```
URL: <api.url>      # 来自于配置中的 `api.url` 字段
Method: POST        # 固定为 POST 方法

Body: <JSON>        # Body 为 JSON 格式字符串
```

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

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	boolean	是否为 Retain 消息
payload	string	消息 Payload
ts	integer	消息时间戳(毫秒)

2.3 Webhook实现客户端断连监控

2.3.1 断连监控需求

系统需要知道所有客户端当前的连接状态，方便在后台管理系统中进行直观展示

2.3.2 代码实现

通过EMQX 的webhook将客户端的连接断开等事件通知到我们自建的服务上，通过事件类型获取客户端的连接状态，然后将客户端的连接状态进行存储，并且提供HTTP API供后台系统查询所有客户端的状态

1：在原有项目`emq-demo`中创建：com.itheima.controller.mqtt.WebHookController

```
/**
 * Created by 传智播客*黑马程序员.
 */
@RestController
@RequestMapping("/mqtt")
public class WebHookController {

    private static final Logger log = LoggerFactory.getLogger(WebHookController.class);

    private Map<String,Boolean> clientStatusMap = new HashMap<>();

    @PostMapping("/webhook")
    public void hook(@RequestBody Map<String,Object> params){
        log.info("emqx 触发 webhook,请求体数据={}",params);

        String action = (String) params.get("action");
```

```
        String clientId = (String) params.get("clientid");
        if(action.equals("client_connected")){
            //客户端成功接入
            clientStatusMap.put(clientId,true);
        }
        if(action.equals("client_disconnected")){
            //客户端断开连接
            clientStatusMap.put(clientId,false);
        }
    }

    @GetMapping("/getall")
    public Map getAllStatus(){
        return clientStatusMap;
    }
}
```

hook方法用来接收EMQ X传入过来的请求，将客户端Id的连接状态记录到map中，getAllStatus方法用来返回所有客户端状态。

然后通过客户端连接/断开EMQ X之后，通过访问`all`接口就能得到这些客户端得状态了。当然了，在实际得项目中肯定就不会这么简单，我们会将这些客户端的状态存入类似redis这样的分布式缓存中，方便整个系统进行存取随时获取客户端状态。

2：修改配置文件`/etc/plugins/emqx_web_hook.conf`，配置webhook访问地址

```
web.hook.api.url = http://192.168.200.10:8991/mqtt/webhook
```

关于事件规则：该配置文件中已默认配置了客户端成功连接和断开的事件及规则，我们就不需要在配置了，如下

```
web.hook.rule.client.connected.1     = {"action": "on_client_connected"}
web.hook.rule.client.disconnected.1  = {"action": "on_client_disconnected"}
```

其他事件的默认配置如果不想要可以注释掉

3：在EMQX Dashboard中开启`emqx_web_hook`插件

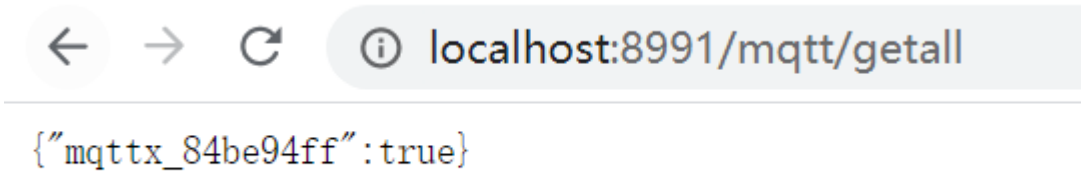
2.3.3 测试

1：打开客户端MQTTX工具，创建新的连接，如下

使用用户：user/123456登陆，点击连接查看服务控制台输出

```
INFO 6028 --- [nio-8991-exec-8] c.i.controller.mqtt.WebHookController: emqx 触发 webhook,请求体数据={username=user, proto_ver=4, keepalive=60, ipaddress=192.168.200.10, connected_at=1591690815, clientid=mqttx_84be94ff, action=client_connected}
```

2：然后在浏览器中通过访问：`<http://localhost:8991/mqtt/getall>`获取所有客户端的状态



3：从MQTTX客户端工具上断开连接，查看服务控制台输出，再次访问`http://localhost:8991/mqtt/getall`查看客户端状态数据

3.EMQ X集群

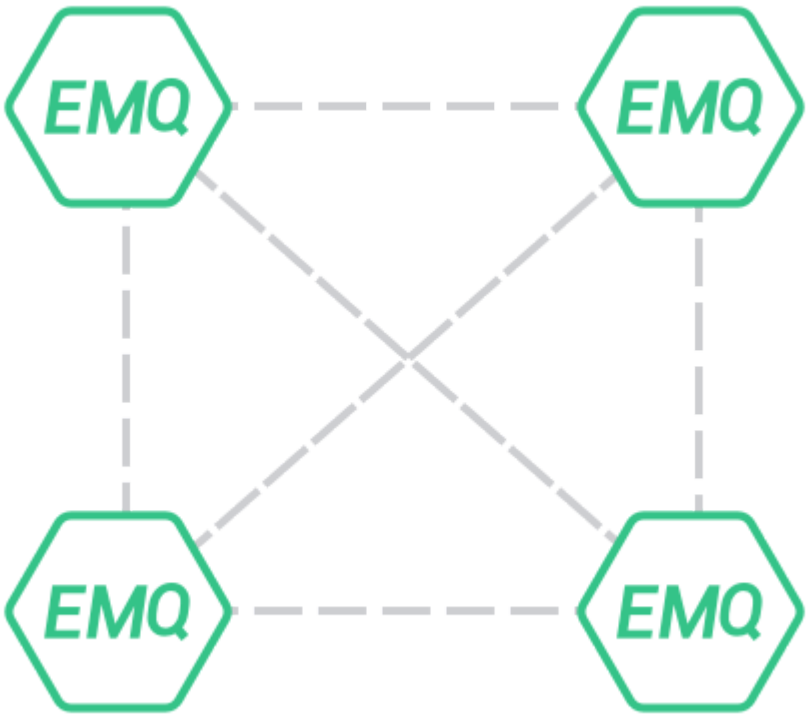
3.1 EMQ X 集群概述

EMQ X是由Erlang语言编写的，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。Erlang 分布式的定义为：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 节点有着唯一的节点名称标识，节点名称由 `@` 分隔的两部分组成

```
<name>@<ip-address>
```

节点间通过节点名称进行通信寻址，所有节点组成一个集群后，每个节点都会与其他节点建立一个TCP连接，每当一个新的节点加入集群时，它也会与集群中所有的节点都建立一个 TCP 连接，最终构成一个网状结构如下：



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

cookie 的配置在`etc/emqx.conf`配置文件中，默认配置如下

```
## Cookie for distributed node communication.
##  采用默认配置即可
## Value: String
node.cookie = emqxsecretcookie
```

另外可查看该配置文件中默认的节点名称如下：

```
## Node name.
##
## See: http://erlang.org/doc/reference_manual/distributed.html
##
## Value: <name>@<host>
##
## Default: emqx@127.0.0.1
node.name = 14332431afdad9@172.17.0.2
```

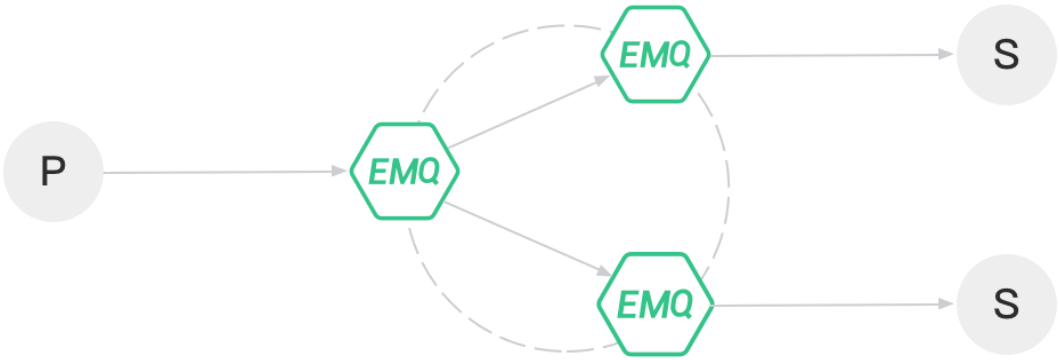
EMQ X 集群协议设置

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

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

3.2 EMQ X 分布式集群设计

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



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

3.2.1 订阅表: 主题 - 订阅者

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

```
node1:

    topic1 -> client1, client2
    topic2 -> client3`

node2:

    topic1 -> client4
```

3.2.2 路由表: 主题 - 节点

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

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

这样的话：一旦某一个主题有消息发布过来，我们可以通过路由表将消息路由到该主题对应的节点上，然后根据订阅表将该主题的消息推送给具体的订阅者

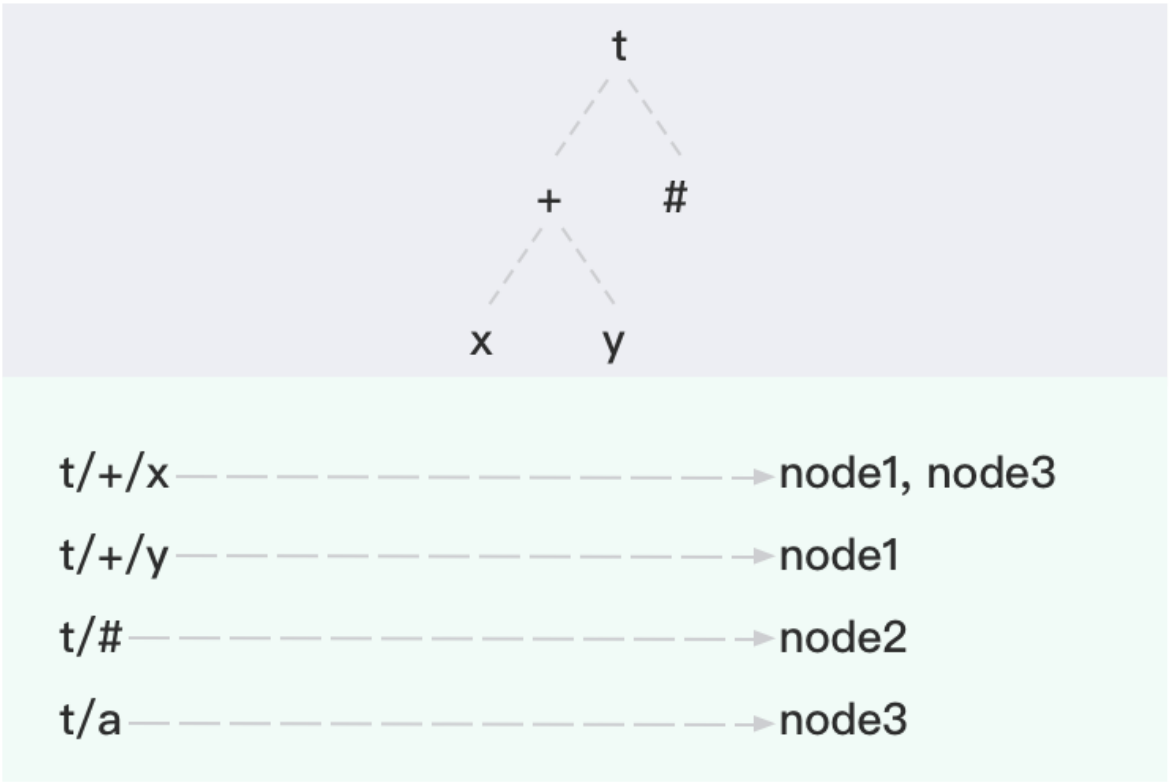
3.2.3 主题树: 带通配符的主题匹配

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

例如下述主题订阅关系:

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

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



3.2.4 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

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

3.3 节点发现与自动集群

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

EMQ X 支持多种节点发现策略:

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

节点发现策略在配置文件：`etc/emqx.conf`中的配置项`cluster.discovery`

可通过如下命令查看默认的配置：

```
/opt/emqx/etc $ cat emqx.conf | grep cluster.discovery
cluster.discovery = manual
```

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 \`命令加入：

```
cluster.discovery = manual
```

基于 static 节点列表自动集群

配置固定的节点列表，自动发现并创建集群：

```
cluster.discovery = static
##节点名称列表
cluster.static.seeds = emqx1@127.0.0.1,emqx2@127.0.0.1
```

基于 mcast 组播自动集群

基于 UDP 组播自动发现并创建集群：

```
cluster.discovery = mcast
cluster.mcast.addr = 239.192.0.1
cluster.mcast.ports = 4369,4370
cluster.mcast.iface = 0.0.0.0
cluster.mcast.ttl = 255
cluster.mcast.loop = on
```

基于 DNS A 记录自动集群

基于 DNS A 记录自动发现并创建集群:

```
cluster.discovery = dns
cluster.dns.name = localhost
cluster.dns.app  = ekka
```

基于 etcd 自动集群

基于 etcd 自动发现并创建集群:

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

基于 kubernetes 自动集群

Kubernetes 下自动发现并创建集群:

```
cluster.discovery = k8s
cluster.k8s.apiserver = http://10.110.111.204:8080
cluster.k8s.service_name = ekka
cluster.k8s.address_type = ip
cluster.k8s.app_name = ekka
```

3.4 manual方式管理集群实践

本环节中将通过manual方式搭建集群

节点名	主机名 (FQDN)	IP 地址
emqx130@centos7-docker1 或 emqx130@192.168.200.130	centos7-docker1	192.168.200.130
emqx129@centos7-docker2 或 emqx129@192.168.200.129	centos7-docker2	192.168.200.129

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

> 主机名可以通过`hostname`命令查看，因为没有配置主机名许可我们采用IP地址

在这里为了操作方便，我们在本地部署两个EMQ X节点`emqx1@127.0.0.1`和`emqx2@127.0.0.1`

3.4.1 配置节点1

1：从课程资料中找到EMQ X的windows版本压缩包`emqx-windows-v4.0.5.zip`解压至D盘

2：修改配置文件`D:/emqx/etc/emqx.conf`，修改集群发现方式`cluster.discovery = manual`

3：修改配置文件`D:/emqx/etc/emqx.conf`，修改节点名称如下：

```
node.name = emqx1@127.0.0.1
```

也可通过环境变量:

```
export EMQX_NODE_NAME=emqx1@127.0.0.1 && ./bin/emqx start
```

注意: 节点启动加入集群后，节点名称不能变更。

4：同时确保`node.cookie = emqxsecretcookie`，同一集群中的节点需要通过cookie进行互联认证，要保证和另一节点上的一致，因为采用的是默认值故是一致的

防火墙设置

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

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

```
## Distributed node port range
node.dist_listen_min = 6369
node.dist_listen_max = 7369
```

5：启动本地节点1，进入`D:/emqx/bin`在DOS命令行窗口下执行命令：`emqx start`

6：启动成功后访问本地节点的Dashboard：`http://localhost:18083/`

3.4.2 配置节点2

1：直接从本地`D:/emqx`复制出一份`D:/emqx1`作为第二个节点

2：由于我们是基于第一个节点复制的，因此集群发现方式，cookie，集群节点连接与通信端口段等相关设置都是一样的，现在需要修改节点2的节点名称，在配置文件`D:/emqx1/etc/emqx.conf`

```
node.name = emqx2@127.0.0.1
```

3：在本地同时启动两个节点会有端口占用冲突的问题，因为我们需要修改节点二相关的端口占用情况

全局配置文件`D:/emqx1/etc/emqx.conf`，需要修改的相关端口占用情况

```
## TCP server port for RPC.
## Value: Port [1024-65535]
rpc.tcp_server_port = 15369
## TCP port for outgoing RPC connections.
## Value: Port [1024-65535]
rpc.tcp_client_port = 15369
## MQTT/TCP - External TCP Listener for MQTT Protocol
## listener.tcp.$name is the IP address and port that the MQTT/TCP
## listener will bind.
## Value: IP:Port | Port
## Examples: 1883, 127.0.0.1:1883, ::1:1883
listener.tcp.external = 0.0.0.0:2883
```

```
## Internal TCP Listener for MQTT Protocol
## The IP address and port that the internal MQTT/TCP protocol listener
## will bind.
## Value: IP:Port, Port
## Examples: 11883, 127.0.0.1:11883, ::1:11883
listener.tcp.internal = 127.0.0.1:21883
## MQTT/SSL - External SSL Listener for MQTT Protocol
## listener.ssl.$name is the IP address and port that the MQTT/SSL
## listener will bind.
## Value: IP:Port | Port
##
## Examples: 8883, 127.0.0.1:8883, ::1:8883
listener.ssl.external = 18883
## External WebSocket listener for MQTT protocol
## listener.ws.$name is the IP address and port that the MQTT/WebSocket
## listener will bind.
## Value: IP:Port | Port
##
## Examples: 8083, 127.0.0.1:8083, ::1:8083
listener.ws.external = 8183
## External WebSocket/SSL listener for MQTT Protocol
## listener.wss.$name is the IP address and port that the MQTT/WebSocket/SSL
## listener will bind.
## Value: IP:Port | Port
## Examples: 8084, 127.0.0.1:8084, ::1:8084
listener.wss.external = 8184

```

EMQX相关插件的端口占用情况也需要修改

Dashboard插件的端口，在`D:\emqx1\etc\plugins\emqx_dashboard.conf`

```
## HTTP Listener
## The port that the Dashboard HTTP listener will bind.
## Value: Port
## Examples: 18083
dashboard.listener.http = 28083
```

management查看的端口在`D:\emqx1\etc\plugins\emqx_management.conf`

```
## HTTP Listener
management.listener.http = 8181
```

4：启动节点2，进入`D:/emqx1/bin`执行命令`emqx start`

5：访问节点2Dashboard，`<http://localhost:28083/>`

3.4.3 节点加入集群

启动两台节点后，在节点1上执行如下操作

```
D:\emqx\bin>emqx_ctl cluster join emqx2@127.0.0.1

=CRITICAL REPORT==== 10-Jun-2020::15:47:02.697000 ===
[EMQ X] emqx shutdown for join
Join the cluster successfully.
Cluster status: #{running_nodes => ['emqx1@127.0.0.1','emqx2@127.0.0.1'],
                  stopped_nodes => []}
```

或者在节点2上执行

```
D:\emqx1\bin>emqx_ctl cluster join emqx1@127.0.0.1

=CRITICAL REPORT==== 10-Jun-2020::15:47:02.697000 ===
[EMQ X] emqx shutdown for join
Join the cluster successfully.
Cluster status: #{running_nodes => ['emqx1@127.0.0.1','emqx2@127.0.0.1'],
                  stopped_nodes => []}
```

在任意节点上查询集群状态:

```
D:\emqx\bin>emqx_ctl cluster status

Cluster status: #{running_nodes => ['emqx1@127.0.0.1','emqx2@127.0.0.1'],
                  stopped_nodes => []}
```

在任意节点的Dahboard上查看集群节点的情况

Nodes(2)

Name	Erlang/OTP Release	Erlang Processes (used/avaliable)	CPU Info (1load/5load/15load)	Memory Info (used/total)	MaxFds
emqx1@127.0.0.1	R22/10.7	308 / 262144	0.00 / 0.00 / 0.00	55.33M / 75.69M	16384
emqx2@127.0.0.1	R22/10.7	307 / 262144	0.00 / 0.00 / 0.00	56.61M / 73.69M	16384

3.4.4 退出集群

节点退出集群，两种方式:

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

让节点2主动退出集群:

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

或节点1上，从集群删除节点2:

```
$ ./bin/emqx_ctl cluster force-leave emqx2@127.0.0.1
```

3.4.5 集群脑裂与自动愈合

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

```
cluster.autoheal = on
```

集群脑裂自动恢复流程:

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

3.4.6 集群节点自动清除

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

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

4.管理监控API的使用

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

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

4.1 接口安全及响应码

EMQ X 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：`amdin/public`。你可以在 Dashboard 的左侧菜单栏里，选择 "MANAGEMENT" -> "Applications" 来修改和添加 AppID/AppSecret。

响应码

EMQ X 接口在调用成功时总是通过HTTP status code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

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

可能的返回码如下：

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

4.2 核心关键API

- 获取所有支持的API接口

  GET /api/v4
- 获取Broker基本信息

  GET /api/v4/brokers/{node}
- 获取集群中的某个节点信息

  GET /api/v4/nodes/{node}
- 获取客户端列表信息(支持分页)

  GET /api/v4/clients

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

- 获取集群下所有订阅信息

  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

- 获取集群下所有状态数据

  GET /api/v4/stats

- 获取集群下当前告警信息

  GET /api/v4/alarms/present

- 获取黑名单

  GET /api/v4/banned

5. 保留消息

5.1 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。

EMQ X 默认开启保留消息的功能，可以在 `etc/emqx.conf` 中修改 `mqtt.retain_available` 为 `false` 以禁用保留消息功能。如果 EMQ X 在保留消息功能被禁用的情况下依然收到了保留消息，那么将返回原因码为 0x9A（不支持保留消息）的 DISCONNECT 报文。

应用场景举例：

某车联网项目，车辆出租公司会实时监控所有车辆的GPS地理位置信息，这些信息是通过每个车辆每10分钟定时上报的GPS信息，这些信息需要展示在某调度系统的大屏上，该调度系统因为其他模块升级需要重新部署，升级后也需要去订阅获取所有车辆的GPS信息，上线完成后刚好错过了车辆最近一次上报的GPS信息，如果这些消息不是保留消息，该调度系统大屏上是空白的，必须等10分钟后才能调度这些车辆，10分钟内无法做出任何操作，用户体验非常差，但是如果这些信息是保留消息，该系统上线后立即就会收到最近所有车辆的位置信息，立即就可以展示然后进行调度。

5.2 保留消息配置

EMQ X 的保留消息功能是由 `emqx_retainer` 插件实现，该插件默认开启，通过修改 `emqx_retainer` 插件的配置，可以调整 EMQ X 储存保留消息的位置，限制接收保留消息数量和 Payload 最大长度，以及调整保留消息的过期时间。

`emqx_retainer` 插件默认开启，插件的配置路径为 `etc/plugins/emqx_retainer.conf`。

配置项	类型	可取值	默认值	说明
retainer.storage_type	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。
retainer.max_retained_messages	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。
retainer.max_payload_size	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。
retainer.expiry_interval	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。

EMQ X Enterprise 中可将保留消息存储到多种外部数据库。