There's nothing here.

Take me home

目录

产品概览

EMQX (Erlang/Enterprise/Elastic MQTT Broker) 是基于 Erlang/OTP 平台开发的开源物联网 MQTT 消息服务器。

Erlang/OTP是出色的软实时 (Soft-Realtime)、低延时 (Low-Latency)、分布式 (Distributed)的语言平台。

MQTT 是轻量的 (Lightweight)、发布订阅模式 (PubSub) 的物联网消息协议。

EMQX 设计目标是实现高可靠，并支持承载海量物联网终端的 MQTT 连接，支持在海量物联网设备间低延时消息路由:

1. 稳定承载大规模的 MQTT 客户端连接，单服务器节点支持 200 万连接。
2. 分布式节点集群，快速低延时的消息路由。
3. 消息服务器内扩展，支持定制多种认证方式、高效存储消息到后端数据库。
4. 完整物联网协议支持，MQTT、MQTT-SN、CoAP、LwM2M、WebSocket 或私有协议支持。

EMQX 消息服务器功能列表

- 完整的 MQTT V3.1/V3.1.1 及 V5.0 协议规范支持
  - QoS0, QoS1, QoS2 消息支持
  - 持久会话与离线消息支持
  - Retained 消息支持
  - Last Will 消息支持
- MQTT/WebSocket TCP/SSL 支持
- HTTP 消息发布接口支持
- $SYS/# 系统主题支持
- 客户端在线状态查询与订阅支持
- 客户端 ID 或 IP 地址认证支持
- 用户名密码认证支持
- LDAP、Redis、MySQL、PostgreSQL、MongoDB、HTTP 认证集成
- 浏览器 Cookie 认证
- 基于客户端 ID、IP 地址、用户名的访问控制 (ACL)
- 多服务器节点集群 (Cluster)
- 支持 manual、mcast、dns、etcd、k8s 等多种集群发现方式
- 网络分区自动愈合
- 消息速率限制
- 连接速率限制
- 按分区配置节点
- 多服务器节点桥接 (Bridge)
- MQTT Broker 桥接支持
- Stomp 协议支持
- MQTT-SN 协议支持
- CoAP 协议支持
- LwM2M 协议支持
- Stomp/SockJS 支持
- 延时 Publish ($delay/topic)
- Flapping 检测

- 黑名单支持
- 共享订阅 ($share/:group/topic)
- TLS/PSK 支持
- 规则引擎
  - 空动作 (调试)
  - 消息重新发布
  - 桥接数据到 MQTT Broker
  - 检查 (调试)
  - 发送数据到 Web 服务

EMQX 企业版安装

EMQX 消息服务器可跨平台运行在 Linux 服务器上。

EMQX License 文件获取

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

EMQX 程序包下载

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

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

CentOS

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

使用 rpm 包安装 EMQX

1. 访问emqx.com 选择 CentOS 版本，然后下载要安装的 EMQX 版本的 rpm 包。

2. 安装 EMQX

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

3. 导入License文件:

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

4. 启动 EMQX

   - 直接启动

```sh
$ emqx start
emqx  is started successfully!

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

   - systemctl 启动

```sh
$ sudo systemctl start emqx
```

- service 启动

```sh
$ sudo service emqx start
```

使用 zip 包安装 EMQX

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

1. 通过 emqx.com 选择 Centos 版本，然后下载要安装的 EMQX 版本的 zip 包。
2. 解压程序包

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

3. 导入License文件:

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

4. 启动 EMQX

```sh
$ ./bin/emqx start
emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

Ubuntu

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

使用 deb 包安装 EMQX

1. 通过 emqx.com 选择 Ubuntu 版本，然后下载要安装的 EMQX 版本的 deb 包。
2. 安装 EMQX

```sh
# for ubuntu
$ sudo apt install ./emqx-ee-ubuntu18.04-v3.1.0_amd64.deb
# for debian
$ sudo dpkg -i emqx-ee-ubuntu18.04-v3.1.0_amd64.deb
```

3. 导入License文件:

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

4. 启动 EMQX

- 直接启动

```sh
$ emqx start
emqx  is started successfully!

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

- systemctl 启动

```sh
$ sudo systemctl start emqx
```

- service 启动

```sh
$ sudo service emqx start
```

使用 zip 包安装 EMQX

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

1. 通过 emqx.com 选择 Ubuntu 版本，然后下载要安装的 EMQX 版本的 zip 包。

2. 解压程序包

```sh
$ unzip emqx-ee-ubuntu18.04-v4.0.0.zip
```

3. 导入License文件:

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

4. 启动 EMQX

```sh
$ ./bin/emqx start
emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

Debian

- Debian 9
- Debian 10

使用 deb 包安装 EMQX

1. 通过 emqx.com 选择 Debian 版本，然后下载要安装的 EMQX 版本的 deb 包。

2. 安装 EMQX

```sh
# for ubuntu
$ sudo apt install ./emqx-ee-debian9-v3.1.0_amd64.deb

# for debian
# 首先确保已安装 libodbc
$ sudo dpkg -i emqx-ee-debian9-v3.1.0_amd64.deb
```

3. 导入License文件:

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

4. 启动 EMQX

   - 直接启动

```sh
$ emqx start
emqx v4.0.0 is started successfully!

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

   - systemctl 启动

```sh
$ sudo systemctl start emqx
```

   - service 启动

```sh
$ sudo service emqx start
```

使用 zip 包安装 EMQX

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

1. 通过 emqx.com 选择 Debian 版本，然后下载要安装的 EMQX 版本的 zip 包。

2. 解压程序包

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

3. 导入License文件:

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

4. 启动 EMQX

```sh
$ ./bin/emqx start
emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

macOS

使用 ZIP 包安装 EMQX

1. 通过 emqx.com ，选择 EMQX 版本，然后下载要安装的 zip 包。

2. 解压压缩包

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

3. 导入License文件:

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

4. 启动 EMQX

```sh
$ ./bin/emqx start
emqx v4.0.0 is started successfully!

$ ./bin/emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

Docker

1. 获取 docker 镜像

- 通过 Docker Hub 获取

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

2. 启动 docker 容器

```sh
$ docker run -d -\
    -name emqx-ee \
    -p 1883:1883 \
    -p 8083:8083 \
    -p 8883:8883 \
    -p 8084:8084 \
    -p 18083:18083 \
    -v /path/to/emqx.lic:/opt/emqx/etc/emqx.lic
    emqx/emqx-ee:v4.0.0
```

更多关于 EMQX Docker 的信息请查看 Docker Hub

启动 EMQX

后台启动 EMQX

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

systemctl 启动

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

service 启动

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

查看 EMQX 的状态

EMQX 正常启动:

```sh
$ emqx_ctl status
Node 'emqx@127.0.0.1' is started
emqx 4.0.0 is running
```

EMQX 未能正常启动:

```sh
$ emqx_ctl status
Node 'emqx@127.0.0.1' not responding to pings。
```

你可以查看 `logs` 下的日志文件并确认是否属于 常见错误。

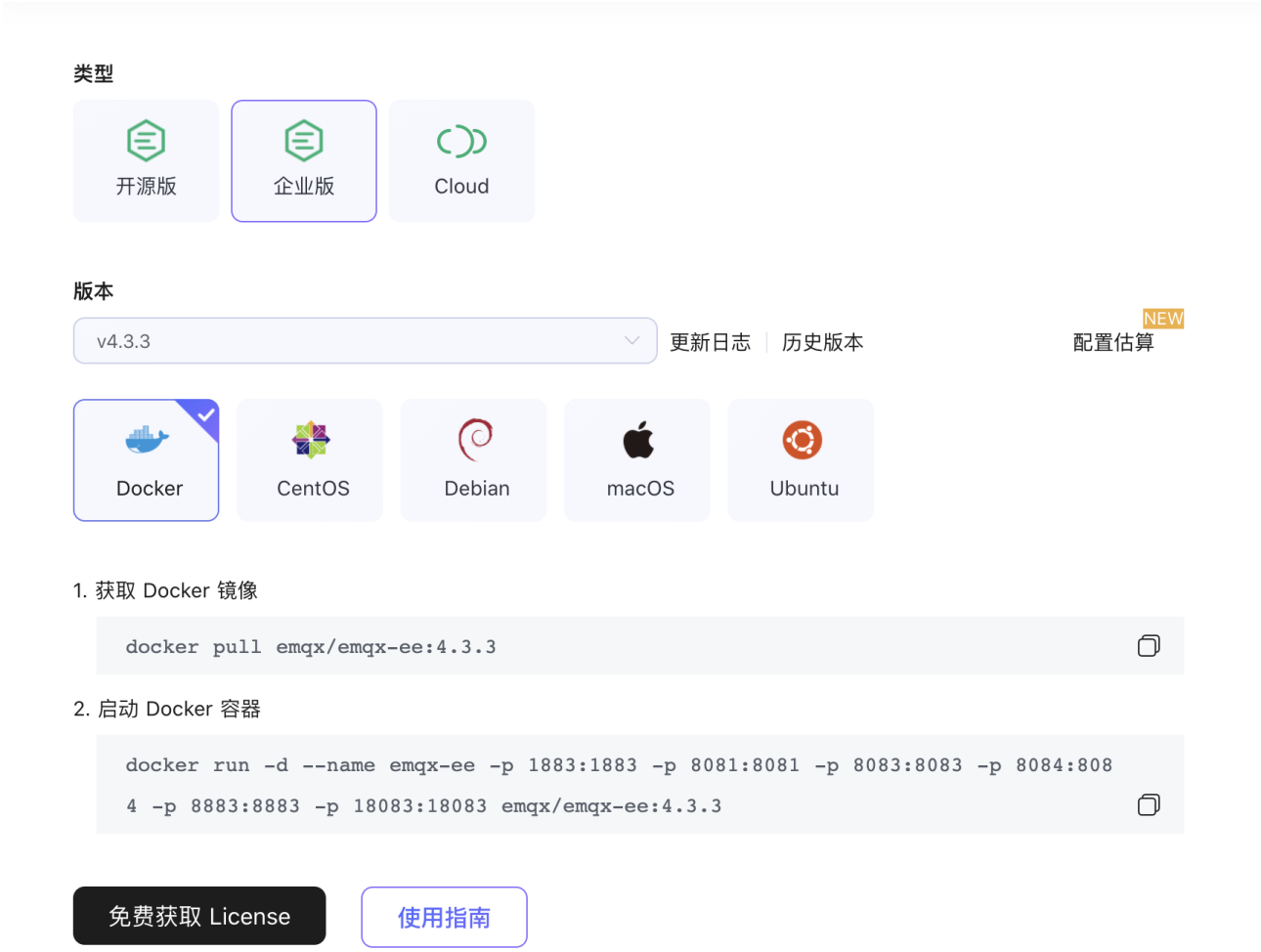
License

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

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

申请试用 License

- 访问 EMQX Enterprise 下载页面，点击 免费获取 License 。



- 申请 License 文件试用，下载 License 文件。

放置 License

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

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

常见错误

EMQX 无法连接 MySQL 8.0

标签: *MySQL 认证*

> 提示
>
> 4.3 已兼容 caching_sha2_password，该问题仅在 4.3 以下的版本中出现。

不同于以往版本，MySQL 8.0 对账号密码配置默认使用 `caching_sha2_password` 插件，需要将密码插件改成 `mysql_native_password`

- 修改 `mysql.user` 表

```
## 切换到 mysql 数据库
mysql> use mysql;

## 查看 user 表

mysql> select user, host, plugin from user;
+------------------+-----------+-----------------------+
| user             | host      | plugin                |
+------------------+-----------+-----------------------+
| root             | %         | caching_sha2_password |
| mysql.infoschema | localhost | caching_sha2_password |
| mysql.session    | localhost | caching_sha2_password |
| mysql.sys        | localhost | caching_sha2_password |
| root             | localhost | caching_sha2_password |
+------------------+-----------+-----------------------+

## 修改密码插件
mysql> ALTER USER 'your_username'@'your_host' IDENTIFIED WITH mysql_native_password BY 'your_
password';
Query OK, 0 rows affected (0.01 sec)

## 刷新
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

- 修改 `my.conf`

  在 `my.cnf` 配置文件里面的 [mysqld] 下面加一行

```
default_authentication_plugin=mysql_native_password
```

- 重启 MySQL 即可

OPENSSL 版本不正确

标签: *启动失败*

现象

执行 `./bin/emqx console` 输出的错误内容包含：

```sh
{application_start_failure,kernel,{{shutdown,{failed_to_start_child,kernel_safe_sup,{on_load_function_failed,crypto}}}, ..}
```

它表示，EMQX 依赖的 Erlang/OTP 中的 `crypto` 应用启动失败。

解决方法

Linux

进入到 EMQX 的安装目录（如果使用包管理工具安装 EMQX，则应该进入与 EMQX 的 `lib` 目录同级的位置）

```sh
## 安装包安装
$ cd emqx

## 包管理器安装，例如 yum。则它的 lib 目录应该在 /lib/emqx
$ cd /lib/emqx
```

查询 `crypto` 依赖的 `.so` 动态库列表及其在内存中的地址：

```sh
$ ldd lib/crypto-*/priv/lib/crypto.so

lib/crypto-4.6/priv/lib/crypto.so: /lib64/libcrypto.so.10: version `OPENSSL_1.1.1' not found (required by lib/crypto-4.6/priv/lib/crypto.so)
        linux-vdso.so.1 =>  (0x00007fff67bfc000)
        libcrypto.so.10 => /lib64/libcrypto.so.10 (0x00007fee749ca000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fee74609000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007fee74404000)
        libz.so.1 => /lib64/libz.so.1 (0x00007fee741ee000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fee74fe5000)
```

其中 `OPENSSL_1.1.1' not found` 表明指定的 OPENSSL 版本的 `.so` 库未正确安装。

源码编译安装 OPENSSL 1.1.1，并将其 so 文件放置到可以被系统识别的路径：

```sh
## 下在最新版本 1.1.1
$ wget https://www.openssl.org/source/openssl-1.1.1c.tar.gz

## 上传至 ct-test-ha
$ scp openssl-1.1.1c.tar.gz ct-test-ha:~/

## 解压并编译安装
$ tar zxf   openssl-1.1.1c.tar.gz
$ cd openssl-1.1.1c
$ ./config
$ make test     # 执行测试；如果输出 PASS 则继续
$ make install

## 确保库的引用
$ ln -s /usr/local/lib64/libssl.so.1.1 /usr/lib64/libssl.so.1.1
$ ln -s /usr/local/lib64/libcrypto.so.1.1 /usr/lib64/libcrypto.so.1.1
```

完成后，执行在 EMQX 的 lib 同级目录下执行 `ldd lib/crypto-*/priv/lib/crypto.so` ，检查是否已能正确识别。如果不在有 `not found` 的 `.so` 库，即可正常启动 EMQX。

macOS

进入到 EMQX 的安装目录：

```sh
## 安装包安装
$ cd emqx

## brew 安装
$ cd /usr/local/Cellar/emqx/<version>/
```

查询 `crypto` 依赖的 `.so` 动态库列表：

```sh
$ otool -L lib/crypto-*/priv/lib/crypto.so

lib/crypto-4.4.2.1/priv/lib/crypto.so:
 /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib (compatibility version 1.1.0, current ve
rsion 1.1.0)
 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1252.200.5)
```

检查其显示 OPENSSL 已成功安装至指定的目录：

```sh
$ ls /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib
ls: /usr/local/opt/openssl@1.1/lib/libcrypto.1.1.dylib: No such file or directory
```

若不存在该文件，则需安装与 `otool` 打印出来的对应的 OPENSSL 版本，例如此处显示的为 `openssl@1.1` ：

```sh
$ brew install openssl@1.1
```

安装完成后，即可正常启动 EMQX。

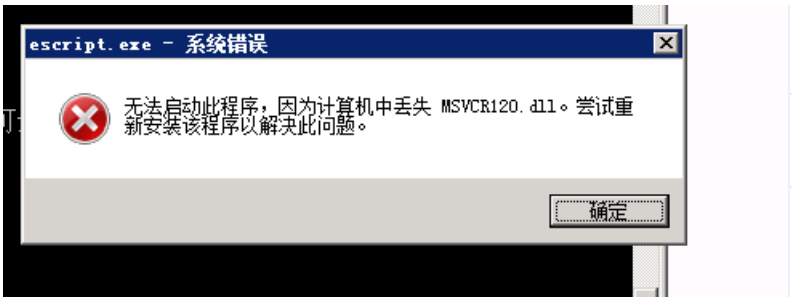
Windows 缺失 MSVCR120.dll

标签: *启动失败*

现象

Windows 执行 `./bin/emqx console` 弹出错误窗口：

```sh
无法启动次程序，因为计算机中丢失 MSVCR120.dll。请尝试重新安装该程序以解决此问题。
```



解决方法

安装 Microsoft Visual C++ RedistributablePackage

SSL 连接失败

现象

客户端无法与 EMQX 建立 SSL 连接。

解决方法

可以借助 EMQX 日志中的关键字来进行简单的问题排查，EMQX 日志相关内容请参考：日志与追踪。

1. certificate_expired

   日志中出现 `certificate_expired` 关键字，说明证书已经过期，请及时续签。

2. no_suitable_cipher

   日志中出现 `no_suitable_cipher` 关键字，说明握手过程中没有找到合适的密码套件，可能原因有证书类型与密码套件不匹配、没有找到服务端和客户端同时支持的密码套件等等。

3. handshake_failure

   日志中出现 `handshake_failure` 关键字，原因有很多，可能要结合客户端的报错来分析，例如，可能是客户端发现连接的服务器地址与服务器证书中的域名不匹配。

4. unknown_ca

   日志中出现 `unknown_ca` 关键字，意味着证书校验失败，常见原因有遗漏了中间 CA 证书、未指定 Root CA 证书或者指定了错误的 Root CA 证书。在双向认证中我们可以根据日志中的其他信息来判断是服务端还是客户端的证书

配置出错。如果是服务端证书存在问题，那么报错日志通常为：

```
{ssl_error,{tls_alert,{unknown_ca,"TLS server: In state certify received CLIENT ALERT: Fatal - Unknown CA\n"}}}
```

看到 `CLIENT ALERT` 就可以得知，这是来自客户端的警告信息，服务端证书未能通过客户端的检查。

如果是客户端证书存在问题，那么报错日志通常为：

```
{ssl_error,{tls_alert,{unknown_ca,"TLS server: In state certify at ssl_handshake.erl:1887 generated SERVER ALERT: Fatal - Unknown CA\n"}}}
```

看到 `SERVER ALERT` 就能够得知，表示服务端在检查客户端证书时发现该证书无法通过认证，而客户端将收到来自服务端的警告信息。

5. protocol_version

日志中出现 `protocol_version` 关键字，说明客户端与服务器支持的 TLS 协议版本不匹配。

消息桥接

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

桥接是一种连接多个 EMQX 或者其他 MQTT 消息中间件的方式。不同于集群，工作在桥接模式下的节点之间不会复制主题树和路由表。桥接模式所做的是：

按照规则把消息转发至桥接节点； 从桥接节点订阅主题，并在收到消息后在本节点/集群中转发该消息。

```sh
                 _________                    _________                    _________
Publisher --> | Node1 | --Bridge Forward--> | Node2 | --Bridge Forward--> | Node3 | --> Subs
criber
                 _________                    _________                    _________
```

工作在桥接模式下和工作在集群模式下有不同的应用场景，桥接可以完成一些单纯使用集群无法实现的功能：

- 跨 VPC 部署。由于桥接不需要复制主题树和路由表，对于网络稳定性和延迟的要求相对于集群更低，桥接模式下不同的节点可以部署在不同的 VPC 上，客户端可以选择物理上比较近的节点连接，提高整个应用的覆盖能力。
- 支持异构节点。由于桥接的本质是对消息的转发和订阅，所以理论上凡是支持 MQTT 协议的消息中间件都可以被桥接到 EMQX，甚至一些使用其他协议的消息服务，如果有协议适配器，也可以通过桥接转发消息过去。
- 提高单个应用的服务上限。由于内部的系统开销，单个的 EMQX 有节点数上限。如果将多个集群桥接起来，按照业务需求设计桥接规则，可以将应用的服务上限再提高一个等级。 在具体应用中，一个桥接的发起节点可以被近似的看作一个远程节点的客户端。

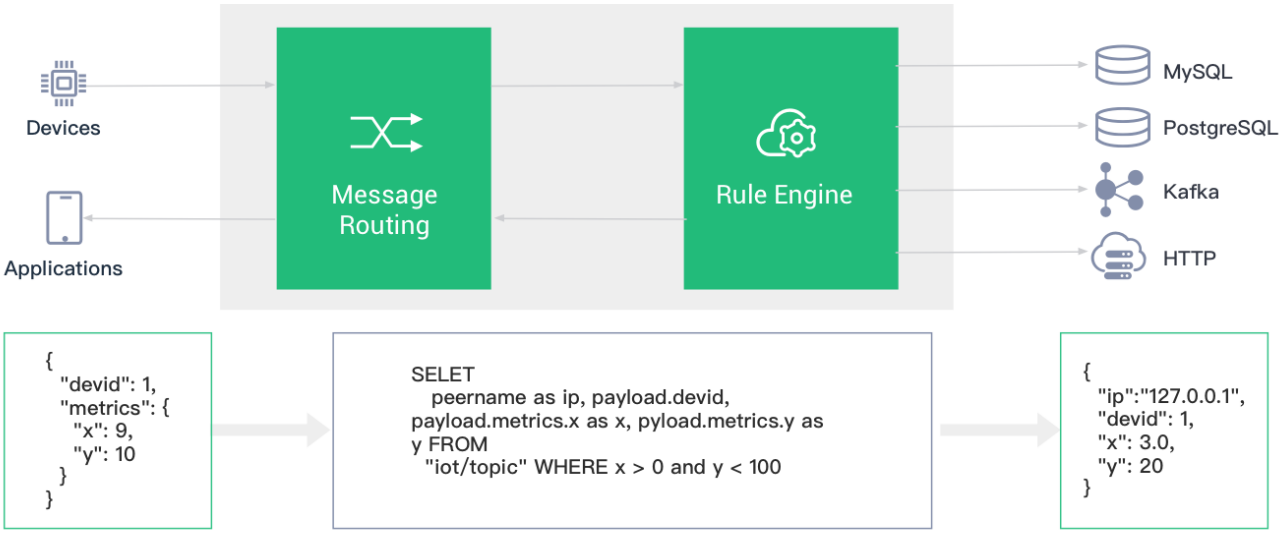
桥接插件列表

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

> 提示
>
> 推荐使用 规则引擎 以实现更灵活的桥接功能。

规则引擎

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



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

> 提示
>
> 适用版本: EMQX v3.1.0+
>
> 兼容提示: EMQX v4.0 对规则引擎 SQL 语法做出较大调整，v3.x 升级用户请参照 迁移指南 进行适配。

EMQX 规则引擎快速入门

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

从本视频中可以快速了解规则引擎解决的问题和基础使用方法。

消息发布

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

选择发布到 t/# 主题的消息，并筛选出全部字段：

```
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 EMQX 内置事件，内置事件提供更精细的消息控制和客户端动作处理能力，可用在 QoS 1 QoS 2 的消息抵达记录、设备上下线记录等业务中。

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

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

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

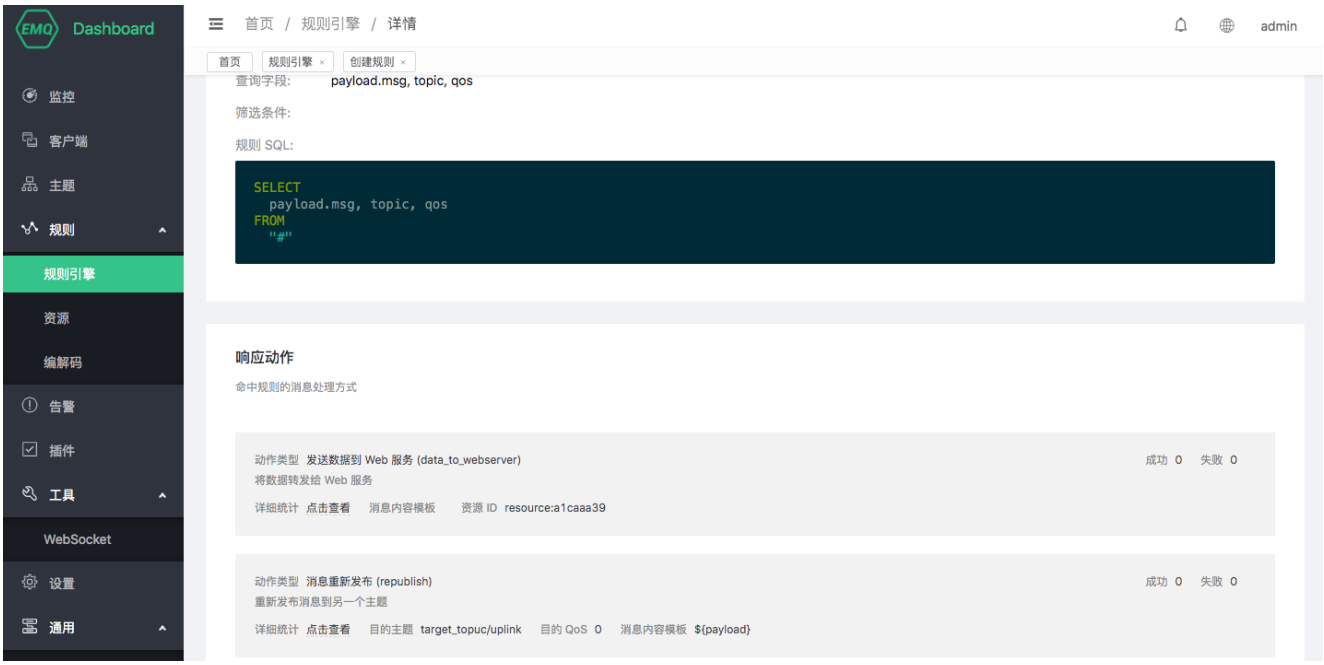
规则引擎组成

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

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

到消息队列等。一条规则可以配置多个响应动作。

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



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

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

规则、动作、资源的关系:

```
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。

> 提示
>
> 动作和资源类型是由 emqx 或插件的代码提供的，不能通过 API 和 CLI 动态创建。

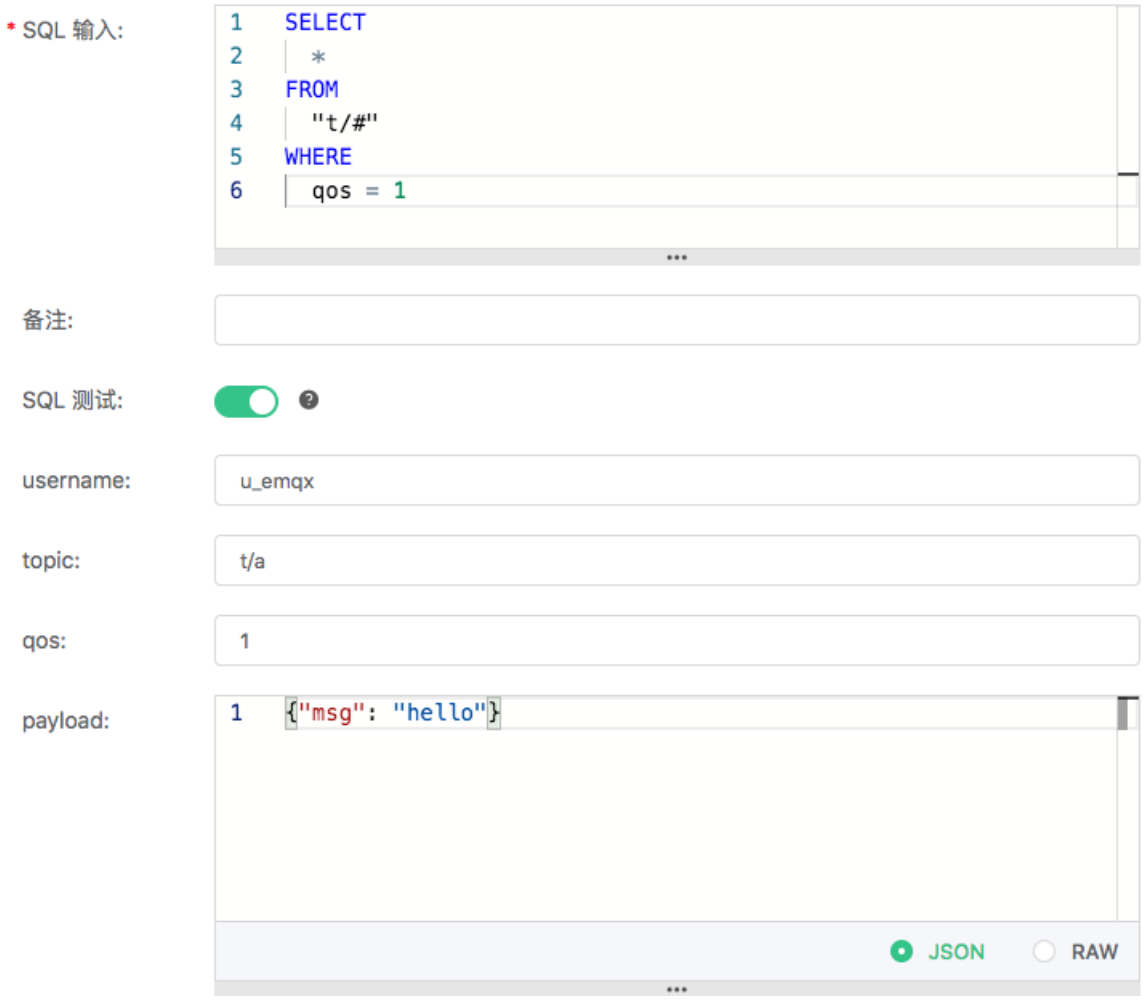
规则引擎典型应用场景举例

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

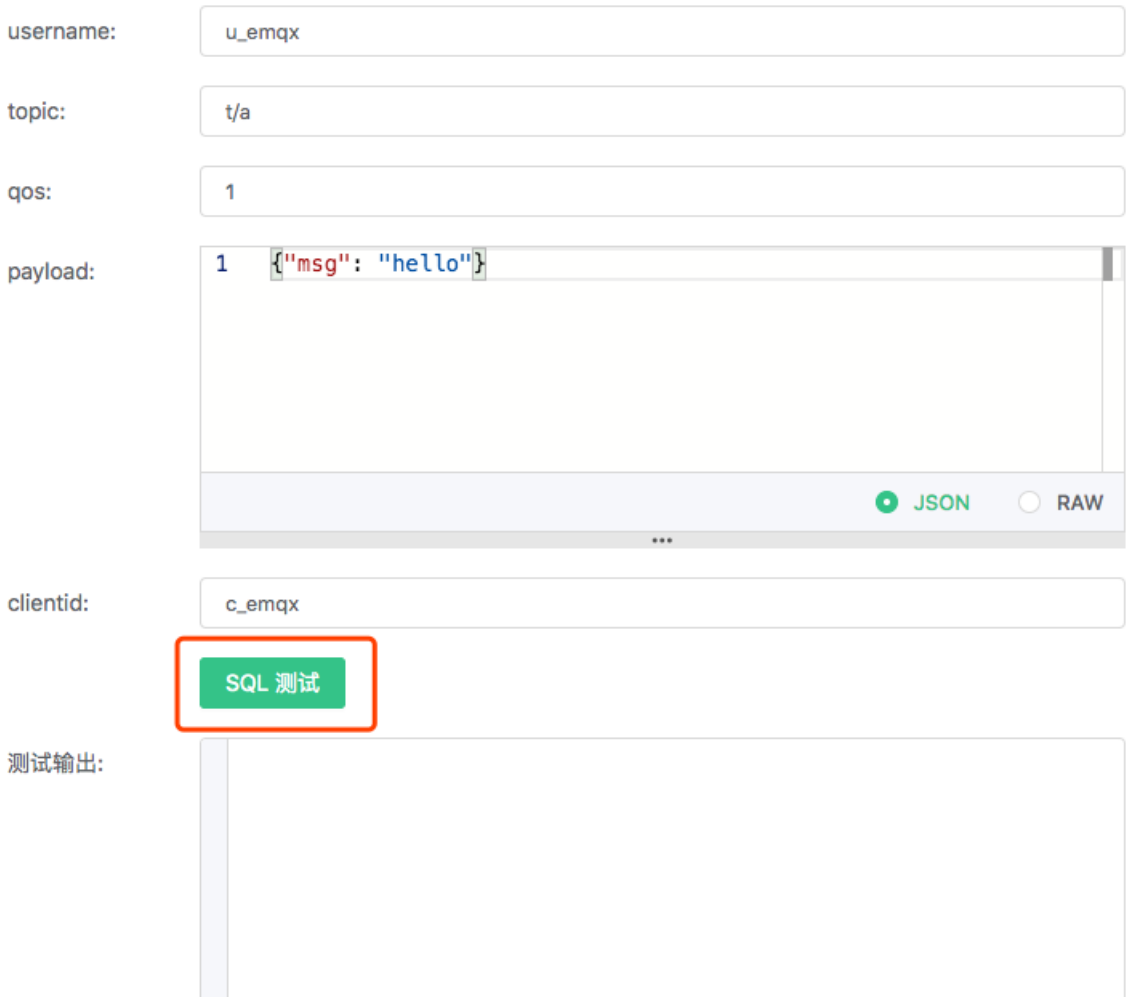
在 Dashboard 中测试 SQL 语句

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

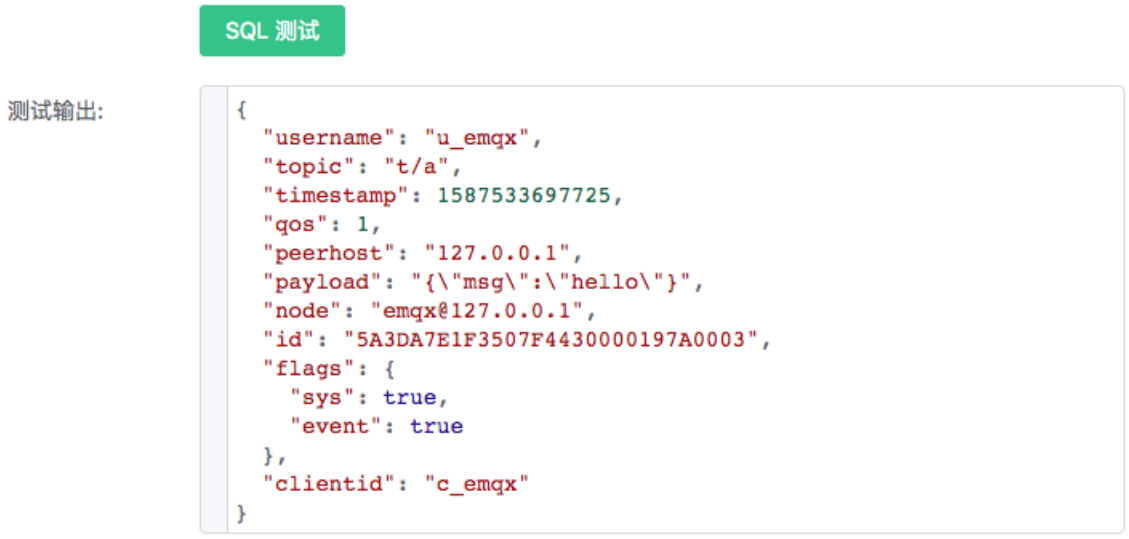
1. 在创建规则界面，输入 规则SQL，并启用 SQL 测试 开关:



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:



3. SQL 处理后的结果将在 测试输出 文本框里展示:



迁移指南

4.0 版本中规则引擎 SQL 语法更加易用，3.x 版本中所有事件 FROM 子句后面均需要指定事件名称，4.0 以后我们引入 事件主题 概念，默认情况下 消息发布 事件不再需要指定事件名称：

```
## 3.x 版本
## 需要指定事件名称进行处理
SELECT * FROM "t/#" WHERE topic =~ 't/#'

## 4.0 及以后版本
## 默认处理 message.publish 事件，FROM 后面直接填写 MQTT 主题
## 上述 SQL 语句等价于：
SELECT * FROM 't/#'

## 其他事件，FROM 后面填写事件主题
SELECT * FROM "$events/message_acked" where topic =~ 't/#'
SELECT * FROM "$events/client_connected"
```

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

下一部分，规则引擎语法和示列

日志与追踪

控制日志输出

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

```
log.to = file
```

`log.to` 的默认值为 `file`，它有以下可选值：

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

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

日志级别

EMQX 的日志分 8 个等级 (RFC 5424)，由低到高分别为：

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

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

```sh
log.level = warning
```

此配置将所有 log handler 的配置设置为 warning。

日志文件和日志滚动

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

```sh
log.dir = log
```

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

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

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

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

```sh
log.file = emqx.log
```

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

```sh
log.rotation.size = 10MB
log.rotation.count = 5
```

针对日志级别输出日志文件

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

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

```sh
log.info.file = info.log
```

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

```sh
log.error.file = error.log
```

日志格式

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

将单个日志消息的最大字符长度设置为 8192:

```sh
log.chars_limit = 8192
```

日志消息的格式为(各个字段之间用空格分隔)：

date time level client_info module_info msg

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

- msg: 日志消息内容。格式任意，可包含空格。

日志消息举例 1：

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

此日志消息里各个字段分别为:

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

日志消息举例 2：

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

此日志消息里各个字段分别为:

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

注意此日志消息中，client_info 字段不存在。

日志级别和 log handlers

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

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

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

- 接收什么级别的日志
- 如何过滤日志消息
- 将日志输出到什么地方

我们来看一下 emqx 默认安装的 log handlers:

```sh
$ emqx_ctl log handlers list

LogHandler(id=ssl_handler, level=debug, destination=console, status=started)
LogHandler(id=file, level=warning, destination=log/emqx.log, status=started)
LogHandler(id=default, level=warning, destination=console, status=started)
```

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

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

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

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

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

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

运行时修改日志级别

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

修改全局日志级别：

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

```sh
$ emqx_ctl log set-level debug
```

修改主日志级别：

例如，将 primary log level 设置为 debug:

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

修改某个 log handler 的日志级别：

例如，将 log handler `file` 设置为 debug:

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

停止某个 log handler：

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

```sh
$ emqx_ctl log handlers stop default
```

启动某个已经停止的 log handler：

例如，启动上面已停止的 log handler `default`：

```sh
$ emqx_ctl log handlers start default
```

日志追踪

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

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

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

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

$ emqx_ctl trace start client my_client log/my_client.log
trace clientid my_client successfully
```

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

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

$ emqx_ctl trace start topic 't/#' log/topic_t.log
trace topic t/# successfully
```

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

日志追踪的原理

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

比如使用如下命令启用 client 日志追踪：

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

然后查询已经开启的追踪:

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

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

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

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

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

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