EMQ

Release: R24-07

2024

# MQTT 入门手册

## MQTT Tutorial for IoT Beginners

杭州映云科技有限公司

# 目录

序言 ..... 1
初识 MQTT ..... 2
MQTT 发布/订阅模式介绍 ..... 8
创建 MQTT 连接 ..... 11
MQTT 主题与通配符 ..... 16
持久会话 ..... 22
QoS 0, 1, 2 ..... 29
遗嘱消息 ..... 38
Keep Alive ..... 43
保留消息 ..... 47
请求/响应 ..... 56
用户属性 ..... 64
主题别名 ..... 69
载荷格式指示与内容类型 ..... 73
共享订阅 ..... 79
订阅选项 ..... 85
订阅标识符 ..... 95
消息过期间隔 ..... 101
最大报文大小 ..... 106
原因码（Reason Code） ..... 110
增强认证 ..... 113
控制报文 ..... 118
结语 ..... 123

# 序言

随着物联网技术的爆发式发展，越来越多的设备与网络连接在一起，构成了庞大的物联网生态系统。在这个系统中，通信协议扮演着至关重要的角色。海量的设备接入和设备管理对网络带宽、时延要求、通信协议以及平台服务架构都带来了巨大的挑战。对于物联网协议来说，必须针对性地解决物联网设备通信的几个关键问题。

MQTT 协议正是为了解决这些问题而被创建的。经过多年的发展，MQTT 协议凭借其轻量、高效、可靠的消息传递、海量连接支持、以及安全的双向通信等优点，已成为物联网协议的实施标准，在车联网、工业物联网、智能家居、智慧交通等领域发挥着重要作用。

在本电子书中，我们将从 MQTT 协议的相关基础概念入手，逐步深入对 MQTT 各项特性的使用进行详细讲解，为读者上手使用 MQTT 协议进行物联网平台构建与业务开发提供一个全面、专业、详实的参考手册。

# 初识 MQTT

## MQTT 协议简介

### 概览

MQTT 是一种基于发布/订阅模式的轻量级消息传输协议，专门针对低带宽和不稳定网络环境的物联网应用而设计，可以用极少的代码为联网设备提供实时可靠的消息服务。MQTT 协议广泛应用于物联网、移动互联网、智能硬件、车联网、智慧城市、远程医疗、电力、石油与能源等领域。

MQTT 协议由 Andy Stanford-Clark（IBM）和 Arlen Nipper（Arcom，现为 Cirrus Link）于 1999 年发布。 按照 Nipper 的介绍，MQTT 必须具备以下几点：

- 简单容易实现
- 支持 QoS（设备网络环境复杂）
- 轻量且省带宽（因为那时候带宽很贵）
- 数据无关（不关心 Payload 数据格式）
- 有持续地会话感知能力（时刻知道设备是否在线）

据 Arlen Nipper 在 IBM Podcast 上的自述，MQTT 原名是 **MQ TT**，注意 **MQ** 与 **TT** 之间的空格，其全称为: MQ Telemetry Transport，是九十年代早期他在参与 Conoco Phillips 公司的一个原油管道数据采集监控系统（pipeline SCADA system）时开发的一个实时数据传输协议。它的目的在于让传感器通过带宽有限的 VSAT ，与 IBM 的 MQ Integrator 通信。由于 Nipper 是遥感和数据采集监控专业出身，所以按业内惯例取了 **MQ TT** 这个名字。

### MQTT 与其他协议对比

#### MQTT vs HTTP

- MQTT 的最小报文仅为 2 个字节，比 HTTP 占用更少的网络开销。
- MQTT 与 HTTP 都能使用 TCP 连接，并实现稳定、可靠的网络连接。
- MQTT 基于发布订阅模型，HTTP 基于请求响应，因此 MQTT 支持双工通信。
- MQTT 可实时推送消息，但 HTTP 需要通过轮询获取数据更新。
- MQTT 是有状态的，但是 HTTP 是无状态的。

- MQTT 可从连接异常断开中恢复，HTTP 无法实现此目标。

**MQTT vs XMPP**

MQTT 协议设计简单轻量、路由灵活，将在移动互联网、物联网消息领域，全面取代 PC 时代的 XMPP 协议。

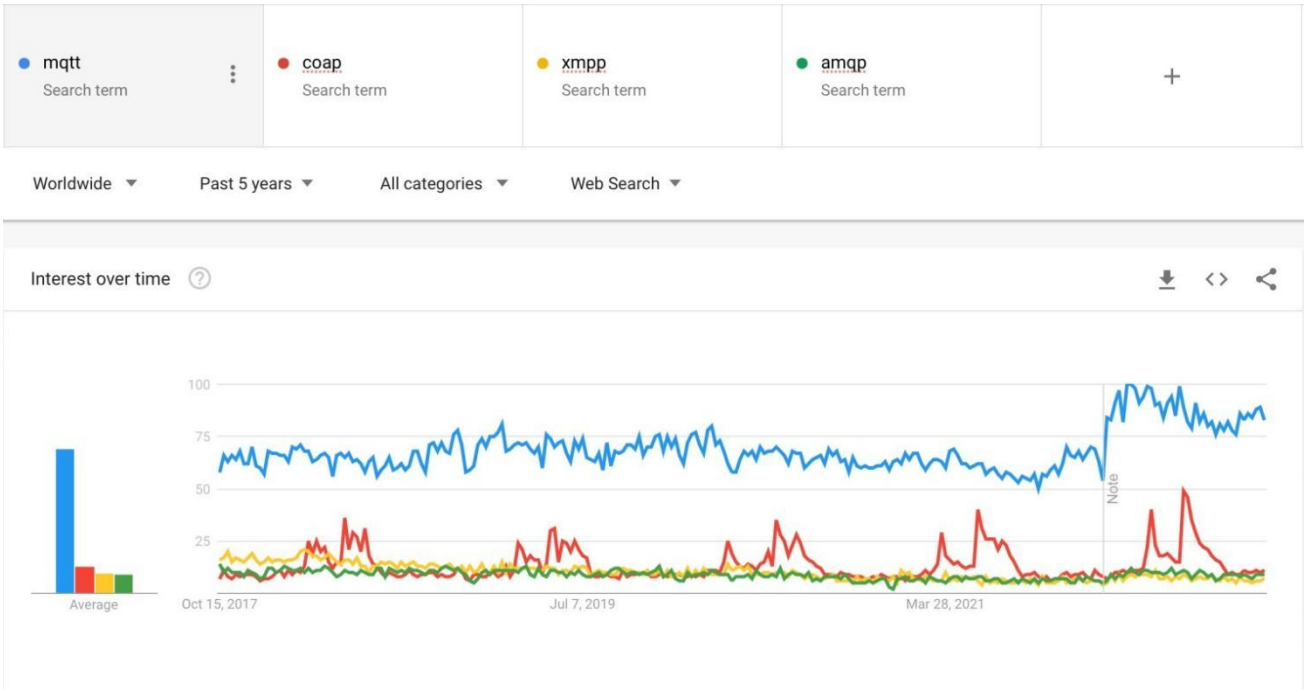
- MQTT 报文体积小且编解码容易，XMPP 基于繁重的 XML，报文体积大且交互繁琐。
- MQTT 基于发布订阅模式，相比 XMPP 基于 JID 的点对点消息路由更为灵活。
- MQTT 支持 JSON、二进制等不同类型报文。XMPP 采用 XML 承载报文，二进制必须 Base64 编码等处理。
- MQTT 通过 QoS 保证消息可靠传输，XMPP 主协议并未定义类似机制。

## 为什么 MQTT 是适用于物联网的最佳协议?

据 IoT Analytics 最新发布的《2022 年春季物联网状况》研究报告显示，到 2022 年，物联网市场预计将增长 18%，达到 144 亿活跃连接。

在如此大规模的物联网需求下，海量的设备接入和设备管理对网络带宽、通信协议以及平台服务架构都带来了巨大的挑战。对于**物联网协议**来说，必须针对性地解决物联网设备通信的几个关键问题：网络环境复杂而不可靠、内存和闪存容量小、处理器能力有限。

MQTT 协议正是为了应对以上问题而创建，经过多年的发展凭借其轻量高效、可靠的消息传递、海量连接支持、安全的双向通信等优点已成为物联网行业的首选协议。




## 轻量高效，节省带宽

MQTT 将协议本身占用的额外消耗最小化，消息头部最小只需要占用 2 个字节，可稳定运行在带宽受限的网络环境下。同时，MQTT 客户端只需占用非常小的硬件资源，能运行在各种资源受限的边缘端设备上。

## 可靠的消息传递

MQTT 协议提供了 3 种消息服务质量等级（Quality of Service），保证了在不同的网络环境下消息传递的可靠性。

- QoS 0：消息最多传递一次。

  如果当时客户端不可用，则会丢失该消息。发布者发送一条消息之后，就不再关心它有没有发送到对方，也不设置任何重发机制。

- QoS 1：消息传递至少 1 次。

  包含了简单的重发机制，发布者发送消息之后等待接收者的 ACK，如果没收到 ACK 则重新发送消息。这种模式能保证消息至少能到达一次，但无法保证消息重复。

- QoS 2：消息仅传送一次。

  设计了重发和重复消息发现机制，保证消息到达对方并且严格只到达一次。

除了 QoS 之外，MQTT 还提供了清除会话（Clean Session）机制。对于那些想要在重新连接后，收到离线期间错过的消息的客户端，可在连接时设置关闭清除会话，此时服务端将会为客户端存储订阅关系及离线消息，并在客户端再次上线后发送给客户端。

### 海量连接支持

MQTT 协议从诞生之时便考虑到了日益增长的海量物联网设备，得益于其优秀的设计，基于 MQTT 的物联网应用及服务可轻松具备高并发、高吞吐、高可扩展能力。

连接海量的物联网设备，离不开 MQTT 服务器的支持。目前，MQTT 服务器中支持并发连接数最多的是 EMQX。最近发布的 EMQX 5.0 通过一个 23 节点的集群达成了 1 亿 MQTT 连接+每秒 100 万消息吞吐，这使得 EMQX 5.0 成为目前为止全球最具扩展性的 MQTT 服务器。

### 安全的双向通信

依赖于发布订阅模式，MQTT 允许在设备和云之间进行双向消息通信。发布订阅模式的优点在于：发布者与订阅者不需要建立直接连接，也不需要同时在线，而是由消息服务器负责所有消息的路由和分发工作。

安全性是所有物联网应用的基石，MQTT 支持通过 TLS/SSL 确保安全的双向通信，同时 MQTT 协议中提供的客户端 ID、用户名和密码允许我们实现应用层的身份验证和授权。

### 在线状态感知

为了应对网络不稳定的情况，MQTT 提供了心跳保活（Keep Alive）机制。在客户端与服务端长时间无消息交互的情况下，Keep Alive 保持连接不被断开，若一旦断开，客户端可即时感知并立即重连。

同时，MQTT 设计了遗愿（Last Will）消息，让服务端在发现客户端异常下线的情况下，帮助客户端发布一条遗愿消息到指定的 MQTT 主题。

另外，部分 MQTT 服务器如 EMQX 也提供了上下线事件通知功能，当后端服务订阅了特定主题后，即可收到所有客户端的上下线事件，这样有助于后端服务统一处理客户端的上下线事件。

## MQTT 5.0 与 3.1.1

在 MQTT 3.1.1 发布并成为 OASIS 标准的四年后，MQTT 5.0 正式发布。这是一次重大的改进和升级，它的目的不仅仅是满足现阶段的行业需求，更是为行业未来的发展变化做了充足的准备。

MQTT 5.0 在 3.1.1 版本基础上增加了会话/消息延时、原因码、主题别名、用户属性、共享订阅等更加符合现代物联网应用需求的特性，提高了大型系统的性能、稳定性与可扩展性。目前，MQTT 5.0 已成为绝大多数物联网企业的首选协议，我们建议初次接触 MQTT 的开发者直接使用该版本。

如果您已经对 MQTT 5.0 产生了一些兴趣，想了解更多，您可以尝试阅读 MQTT 5.0 探索系列文章，该系列文章将以通俗易懂的方式为您介绍 MQTT 5.0 的重要特性。

## MQTT 服务器

MQTT 服务器负责接收客户端发起的连接，并将客户端发送的消息转发到另外一些符合条件的客户端。一个成熟的 MQTT 服务器可支持海量的客户端连接及百万级的消息吞吐，帮助物联网业务提供商专注于业务功能并快速创建一个可靠的 MQTT 应用。

EMQX 是一款应用广泛的大规模分布式物联网 MQTT 服务器。自 2013 年在 GitHub 发布开源版本以来，目前全球下载量已超千万，累计连接物联网关键设备超过 1 亿台。

感兴趣的读者可通过如下 Docker 命令安装 EMQX 5.0 开源版进行体验。

```
1. docker run -d --name emqx -p 1883:1883 -p 8083:8083 -p 8084:8084 -p 8883:8883 -p 18083:18083 emqx/emqx:latest
```

也可直接在 EMQX Cloud 上创建完全托管的 MQTT 服务，免费试用 EMQX Cloud，无需绑定信用卡。

## MQTT 客户端

MQTT 应用通常需要基于 MQTT 客户端库来实现 MQTT 通信。目前，基本所有的编程语言都有成熟的开源 MQTT 客户端库，读者可参考 EMQ 整理的 MQTT 客户端库大全选择一个合适的客户端库来构建满足自身业务需求的 MQTT 客户端。也可直接访问 EMQ 提供的 MQTT 客户端编程系列博客，学习如何在 Java、Python、PHP、Node.js 等编程语言中使用 MQTT。

MQTT 应用开发还离不开 MQTT 测试工具的支持，一款易用且功能强大的 MQTT 测试工具可帮助开发者缩短开发周期，创建一个稳定的物联网应用。

MQTTX 是一款开源的跨平台桌面客户端，它简单易用且提供全面的 MQTT 5.0 功能、特性测试，可运行在 macOS、Linux 和 Windows 上。同时，它还提供了命令行及浏览器版本，满足不同场景下的 MQTT 测试需求。感兴趣的读者可访问 MQTTX 官网进行下载试用：https://mqttx.app/zh。

# MQTT 发布/订阅模式介绍

## 什么是发布/订阅模式?

发布订阅模式（Publish-Subscribe Pattern）是一种消息传递模式，它将发送消息的客户端（发布者）与接收消息的客户端（订阅者）解耦，使得两者不需要建立直接的联系也不需要知道对方的存在。

MQTT 发布/订阅模式的精髓在于由一个被称为代理（Broker）的中间角色负责所有消息的路由和分发工作，发布者将带有主题的消息发送给代理，订阅者则向代理订阅主题来接收感兴趣的消息。

在 MQTT 中，主题和订阅无法被提前注册或创建，所以代理也无法预知某一个主题之后是否会有订阅者，以及会有多少订阅者，所以只能将消息转发给当前的订阅者，**如果当前不存在任何订阅，那么消息将被直接丢弃**。

MQTT 发布/订阅模式有 4 个主要组成部分：发布者、订阅者、代理和主题。

- **发布者（Publisher）**

  负责将消息发布到主题上，发布者一次只能向一个主题发送数据，发布者发布消息时也无需关心订阅者是否在线。

- **订阅者（Subscriber）**

  订阅者通过订阅主题接收消息，且可一次订阅多个主题。MQTT 还支持通过共享订阅的方式在多个订阅者之间实现订阅的负载均衡。

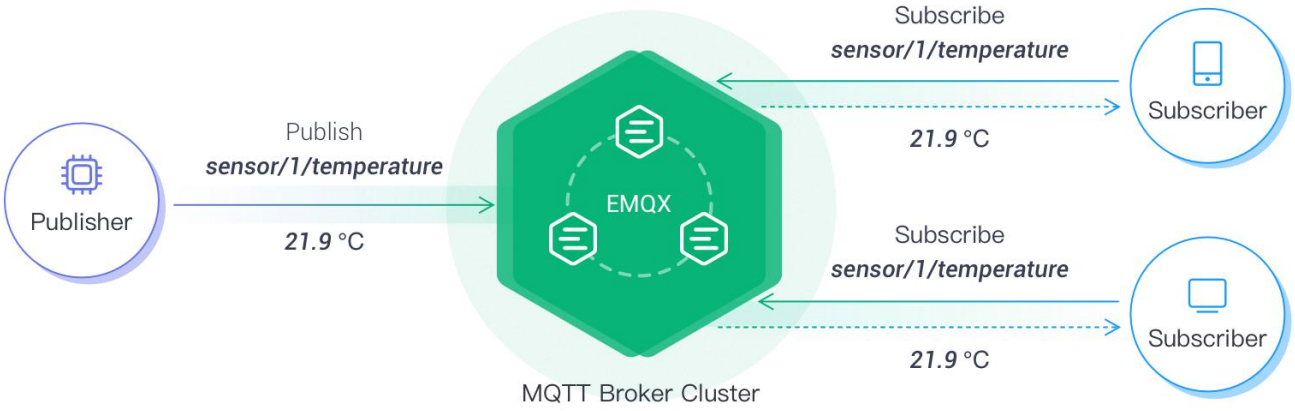
- **代理（Broker）**

  负责接收发布者的消息，并将消息转发至符合条件的订阅者。另外，代理也需要负责处理客户端发起的连接、断开连接、订阅、取消订阅等请求。

- **主题（Topic）**

  主题是 MQTT 进行消息路由的基础，它类似 URL 路径，使用斜杠 / 进行分层，比如 `sensor/1/temperature`。一个主题可以有多个订阅者，代理会将该主题下的消息转发给所有订阅者；一个主题也可以有多个发布者，代理将按照消息到达的顺序转发。

MQTT 还支持订阅者使用主题通配符一次订阅多个主题。




MQTT 发布/订阅架构

## MQTT 发布/订阅中的消息路由

在 MQTT 发布/订阅模式中，一个客户端既可以是发布者，也可以是订阅者，也可以同时具备这两个身份。当客户端发布一条消息时，它会被发送到代理，然后代理将消息路由到该主题的所有订阅者。 当客户端订阅一个主题时，它会收到代理转发到该主题的所有消息。

一般来说，大多数发布/订阅系统主要通过以下两种方式过滤并路由消息。

- 根据主题

  订阅者向代理订阅自己感兴趣的主题，发布者发布的所有消息中都会包含自己的主题，代理根据消息的主题判断需要将消息转发给哪些订阅者。

- 根据消息内容

  订阅者定义其感兴趣的消息的条件，只有当消息的属性或内容满足订阅者定义的条件时，消息才会被投递到该订阅者。

MQTT 协议是基于主题进行消息路由的，在这个基础上，EMQX 从 3.1 版本开始通过基于 SQL 的规则引擎提供了额外的按消息内容进行路由的能力。关于规则引擎的详细信息，请查看 EMQX 文档。

## MQTT 与 HTTP 请求响应

HTTP 是万维网数据通信的基础，其简单易用无客户端依赖，被广泛应用于各个行业。在物联网领域，HTTP 也可以用于连接物联网设备和 Web 服务器，实现设备的远程监控和控制。

虽然使用简单、开发周期短，但是基于请求响应的 HTTP 在物联网领域的应用却有一定的局限性。首先，协议层面 HTTP 报文相较与 MQTT 需要占用更多的网络开销；其次，HTTP 是一种无状态协议，这意味着服务器在处理请求时不会记录客户端的状态，也无法实现从连接异常断开中恢复；最后，请求响应模式需要通过轮询才能获取数据更新，而 MQTT 通过订阅即可获取实时数据更新。

发布订阅模式的松耦合特性，也给 MQTT 带来了一些副作用。由于发布者并不知晓订阅者的状态，因此发布者也无法得知订阅者是否收到了消息，或者是否正确处理了消息。为此，MQTT 5.0 增加了请求响应特性，以实现订阅者收到消息后向某个主题发送应答，发布者收到应答后再进行后续操作。

## MQTT 与消息队列

尽管 MQTT 与消息队列的很多行为和特性非常接近，比如都采用发布/订阅模式，但是他们面向的场景却有着显著的不同。消息队列主要用于服务端应用之间的消息存储与转发，这类场景往往数据量大但客户端数量少。MQTT 是一种消息传输协议，主要用于物联网设备之间的消息传递，这类场景的特点是海量的设备接入、管理与消息传输。

在一些实际的应用场景中，MQTT 与消息队列往往会被结合起来使用，以使 MQTT 服务器能专注于处理设备的连接与设备间的消息路由。比如先由 MQTT 服务器接收物联网设备上报的数据，然后再通过消息队列将这些数据转发到不同的业务系统进行处理。

不同于消息队列，MQTT 主题不需要提前创建。MQTT 客户端在订阅或发布时即自动的创建了主题，开发者无需再关心主题的创建，并且也不需要手动删除主题。

# 创建 MQTT 连接

## MQTT 连接的基本概念

建立一个 MQTT 连接是使用 MQTT 协议进行通信的第一步。为了保证高可扩展性，在建立连接时 MQTT 协议提供了丰富的连接参数，以方便开发者能创建满足不同业务需求的物联网应用。

MQTT 连接由客户端向服务器端发起。任何运行了 MQTT 客户端库的程序或设备都是一个 MQTT 客户端，而 MQTT 服务器则负责接收客户端发起的连接，并将客户端发送的消息转发到另外一些符合条件的客户端。

客户端与服务器建立网络连接后，需要先发送一个 `CONNECT` 数据包给服务器。服务器收到 `CONNECT` 包后会回复一个 `CONNACK` 给客户端，客户端收到 `CONNACK` 包后表示 MQTT 连接建立成功。如果客户端在超时时间内未收到服务器的 `CONNACK` 数据包，就会主动关闭连接。

大多数场景下，MQTT 通过 TCP/IP 协议进行网络传输，但是 MQTT 同时也支持通过 WebSocket 或者 UDP 进行网络传输。

### MQTT over TCP

TCP/IP 应用广泛，是一种面向连接的、可靠的、基于字节流的传输层通信协议。它通过 ACK 确认和重传机制，能够保证发送的所有字节在接收时是完全一样的，并且字节顺序也是正确的。

MQTT 通常基于 TCP 进行网络通信，它继承了 TCP 的很多优点，能稳定运行在低带宽、高延时、及资源受限的环境下。

### MQTT over WebSocket

近年来随着 Web 前端的快速发展，浏览器新特性层出不穷，越来越多的应用可以在浏览器端通过浏览器渲染引擎实现，Web 应用的即时通信方式 WebSocket 也因此得到了广泛的应用。

很多物联网应用需要以 Web 的方式被使用，比如很多设备监控系统需要使用浏览器实时显示设备数据。但是浏览器是基于 HTTP 协议传输数据的，也就无法使用 MQTT over TCP。

MQTT 协议在创建之初便考虑到了 Web 应用的重要性，它支持通过 MQTT over WebSocket 的方式进

行 MQTT 通信。关于如何使用 MQTT over WebSocket，读者可查看博客使用 WebSocket 连接 MQTT 服务器。

# MQTT 连接参数的使用

## 连接地址

MQTT 的连接地址通常包含 ： 服务器 IP 或者域名、服务器端口、连接协议。

### 基于 TCP 的 MQTT 连接

`mqtt` 是普通的 TCP 连接，端口一般为 1883。

`mqtts` 是基于 TLS/SSL 的安全连接，端口一般为 8883。

比如 `mqtt://broker.emqx.io:1883` 是一个基于普通 TCP 的 MQTT 连接地址。

### 基于 WebSocket 的连接

`ws` 是普通的 WebSocket 连接，端口一般为 8083。

`wss` 是基于 WebSocket 的安全连接，端口一般为 8084。

当使用 WebSocket 连接时，连接地址还需要包含 Path，EMQX 默认配置的 Path 是 `/mqtt`。比如 `ws://broker.emqx.io:8083/mqtt` 是一个基于 WebSocket 的 MQTT 连接地址。

## 客户端 ID（Client ID）

MQTT 服务器使用 Client ID 识别客户端，连接到服务器的每个客户端都必须要有唯一的 Client ID。Client ID 的长度通常为 1 至 23 个字节的 UTF-8 字符串。

**如果客户端使用一个重复的 Client ID 连接至服务器，将会把已使用该 Client ID 连接成功的客户端踢下线。**

## 用户名与密码（Username & Password）

MQTT 协议可以通过用户名和密码来进行相关的认证和授权，但是如果此信息未加密，则用户名和密码将以明文方式传输。如果设置了用户名与密码认证，那么最好要使用 mqtts 或 wss 协议。

大多数 MQTT 服务器默认为匿名认证，匿名认证时用户名与密码设置为空字符串即可。

### 连接超时（Connect Timeout）

连接超时时长，收到服务器连接确认前的等待时间，等待时间内未收到连接确认则为连接失败。

### 保活周期（Keep Alive）

保活周期，是一个以秒为单位的时间间隔。客户端在无报文发送时，将按 Keep Alive 设定的值定时向服务端发送心跳报文，确保连接不被服务端断开。

在连接建立成功后，如果服务器没有在 Keep Alive 的 1.5 倍时间内收到来自客户端的任何包，则会认为和客户端之间的连接出现了问题，此时服务器便会断开和客户端的连接。

### 清除会话（Clean Session）

为 `false` 时表示创建一个持久会话，在客户端断开连接时，会话仍然保持并保存离线消息，直到会话超时注销。为 `true` 时表示创建一个新的临时会话，在客户端断开时，会话自动销毁。

持久会话避免了客户端掉线重连后消息的丢失，并且免去了客户端连接后重复的订阅开销。这一功能在带宽小，网络不稳定的物联网场景中非常实用。

> **注意**： 持久会话恢复的前提是客户端使用固定的 Client ID 再次连接，如果 Client ID 是动态的，那么连接成功后将会创建一个新的持久会话。

服务器为持久会话保存的消息数量取决于服务器的配置，比如 EMQ 提供的免费的公共 MQTT 服务器设置的离线消息保存时间为 5 分钟，最大消息数为 1000 条，且不保存 QoS 0 消息。

### 遗嘱消息（Last Will）

遗嘱消息是 MQTT 为那些可能出现**意外断线**的设备提供的将**遗嘱**优雅地发送给其他客户端的能力。设置了遗嘱消息消息的 MQTT 客户端异常下线时，MQTT 服务器会发布该客户端设置的遗嘱消息。

> **意外断线包括**：因网络故障，连接被服务端关闭；设备意外掉电；设备尝试进行不被允许的操作而被服务端关闭连接等。

遗嘱消息可以看作是一个简化版的 MQTT 消息，它也包含 Topic、Payload、QoS、Retain 等信息。

- 当设备意外断线时，遗嘱消息将被发送至遗嘱 Topic；
- 遗嘱 Payload 是待发送的消息内容；
- 遗嘱 QoS 与普通 MQTT 消息的 QoS 一致
- 遗嘱 Retain 为 `true` 时表明遗嘱消息是保留消息。MQTT 服务器会为每个主题存储最新一条保留消息，以方便消息发布后才上线的客户端在订阅主题时仍可以接收到该消息。

### 协议版本

使用较多的 MQTT 协议版本有 MQTT v3.1、MQTT v3.1.1 及 MQTT v5.0。目前，MQTT 5.0 已成为绝大多数物联网企业的首选协议，我们建议初次接触 MQTT 的开发者直接使用该版本。

感兴趣的读者可查看 EMQ 提供的 MQTT 5.0 系列文章，了解 MQTT 5.0 相关特性的使用。

### MQTT 5.0 新增连接参数

#### Clean Start & Session Expiry Interval

MQTT 5.0 中将 Clean Session 拆分成了 Clean Start 与 Session Expiry Interval。

Clean Start 用于指定连接时是创建一个全新的会话还是尝试复用一个已存在的会话。为 `true` 时表示必须丢弃任何已存在的会话，并创建一个全新的会话；为 `false` 时表示必须使用与 Client ID 关联的会话来恢复与客户端的通信（除非会话不存在）。

Session Expiry Interval 用于指定网络连接断开后会话的过期时间。设置为 0 或未设置，表示断开连接时会话即到期；设置为大于 0 的数值，则表示会话在网络连接关闭后会保持多少秒；设置为 0xFFFFFFFF 表示会话永远不会过期。

更多细节可查看博客：Clean Start 与 Session Expiry Interval。

#### 连接属性（Connect Properties）

MQTT 5.0 还新引入了连接属性的概念，进一步增强了协议的可扩展性。更多细节可查看博客：MQTT 5.0 连接属性。

## 如何建立一个安全的 MQTT 连接?

虽然 MQTT 协议提供了用户名、密码、Client ID 等认证机制，但是这对于物联网安全来说还远远不够。基于传统的 TCP 通信使用明文传输，信息的安全性很难得到保证，数据也会存在被**窃听、篡改、伪造、冒充**的风险。

SSL/TLS 的出现很好的解决了通信中的风险问题，其以非对称加密技术为主干，混合了不同模式的加密方式，既保证了通信中消息都以密文传输，避免了被窃听的风险，同时也通过签名防止了消息被篡改。

不同 MQTT 服务器启用 SSL/TLS 的步骤都各有不同，EMQX 内置了对 TLS/SSL 的支持，包括支持单/双向认证、X.509 证书、负载均衡 SSL 等多种安全认证。

单向认证是一种仅通过验证服务器证书来建立安全通信的方式，它能保证通信是加密的，但是不能验证客户端的真伪，通常需要与用户名、密码、Client ID 等认证机制结合。读者可参考博客 EMQX MQTT 服务器启用 SSL/TLS 安全连接来建立一个安全的单向认证 MQTT 连接。

双向认证是指在进行通信认证时要求服务端和客户端都提供证书，双方都需要进行身份认证，以确保通信中涉及的双方都是受信任的。 双方彼此共享其公共证书，然后基于该证书执行验证、确认。一些对安全性要求较高的应用场景，就需要开启双向 SSL/TLS 认证。读者查看博客 EMQX 启用双向 SSL/TLS 安全连接了解如何建立一个安全的双向认证 MQTT 连接。

感兴趣的读者也可查看以下博客来学习物联网安全相关知识：

- 如何保障物联网平台的安全性与健壮性
- 灵活多样认证授权，零开发投入保障 IoT 安全
- 车联网通信安全之 SSL/TLS 协议

> **注意**： 如果在浏览器端使用 MQTT over WebSocket 进行安全连接的话，目前还暂不支持双向认证通信。

# MQTT 主题与通配符

## 什么是 MQTT 主题?

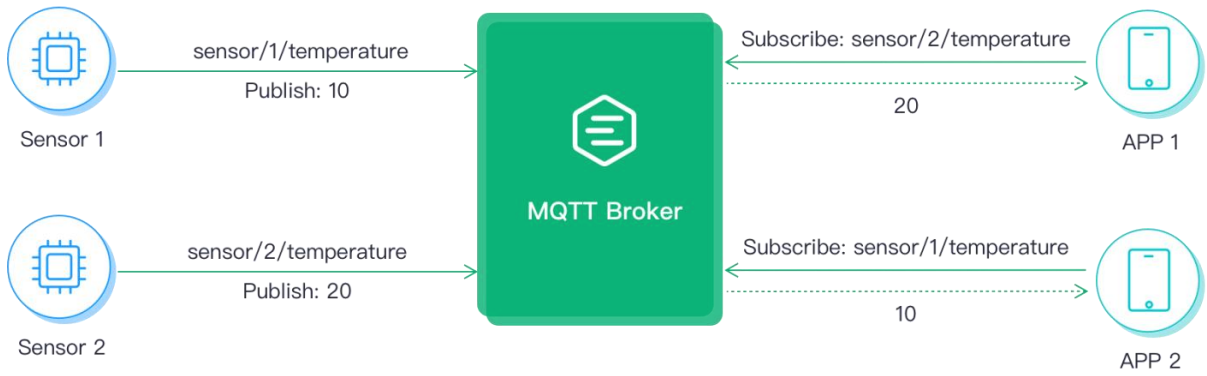
MQTT 主题本质上是一个 UTF-8 编码的字符串，是 MQTT 协议进行消息路由的基础。MQTT 主题类似 URL 路径，使用斜杠 `/` 进行分层：

```
1. chat/room/1
2. sensor/10/temperature
3. sensor/+/temperature
4. sensor/#
```

为了避免歧义且易于理解，通常不建议主题以 `/` 开头或结尾，例如 **`/chat`** 或 **`chat/`**。

不同于消息队列中的主题（比如 Kafka 和 Pulsar），MQTT 主题不需要提前创建。MQTT 客户端在订阅或发布时即自动的创建了主题，开发者无需再关心主题的创建，并且也不需要手动删除主题。

下图是一个简单的 MQTT 订阅与发布流程， **APP 1** 订阅了 **sensor/2/temperature** 主题后，将能接收到 **Sensor 2** 发布到该主题的消息。




## MQTT 主题通配符

MQTT 主题通配符包含单层通配符 `+` 及多层通配符 `#`，主要用于客户端一次订阅多个主题。

> 注意： 通配符只能用于订阅，不能用于发布。

### 单层通配符

加号（“+” U+002B）是用于单个主题层级匹配的通配符。在使用单层通配符时，单层通配符必须占据整个层级，例如：

1. + 有效
2. sensor/+ 有效
3. sensor/+/temperature 有效
4. sensor+ 无效（没有占据整个层级）

如果客户端订阅了主题 **sensor/+/temperature**，将会收到以下主题的消息：

1. sensor/1/temperature
2. sensor/2/temperature
3. ...
4. sensor/n/temperature

但是不会匹配以下主题：

1. sensor/temperature
2. sensor/bedroom/1/temperature

### 多层通配符

井字符号（“#” U+0023）是用于匹配主题中任意层级的通配符。多层通配符表示它的父级和任意数量的子层级，在使用多层通配符时，它必须占据整个层级并且必须是主题的最后一个字符，例如：

1. # 有效，匹配所有主题
2. sensor/# 有效
3. sensor/bedroom# 无效（没有占据整个层级）
4. sensor/#/temperature 无效（不是主题最后一个字符）

如果客户端订阅主题 senser/#，它将会收到以下主题的消息：

1. sensor
2. sensor/temperature
3. sensor/1/temperature

# 以 $ 开头的主题

## 系统主题

以 `$SYS/` 开头的主题为系统主题，系统主题主要用于获取 MQTT 服务器自身运行状态、消息统计、客户端上下线事件等数据。目前，MQTT 协议暂未明确规定 `$SYS/` 主题标准，但大多数 MQTT 服务器都遵循该标准建议。

例如，EMQX 服务器支持通过以下主题获取集群状态。

| 主题 | 说明 |
| --- | --- |
| $SYS/brokers | EMQX 集群节点列表 |
| $SYS/brokers/emqx@127.0.0.1/version | EMQX 版本 |
| $SYS/brokers/emqx@127.0.0.1/uptime | EMQX 运行时间 |
| $SYS/brokers/emqx@127.0.0.1/datetime | EMQX 系统时间 |
| $SYS/brokers/emqx@127.0.0.1/sysdescr | EMQX 系统信息 |

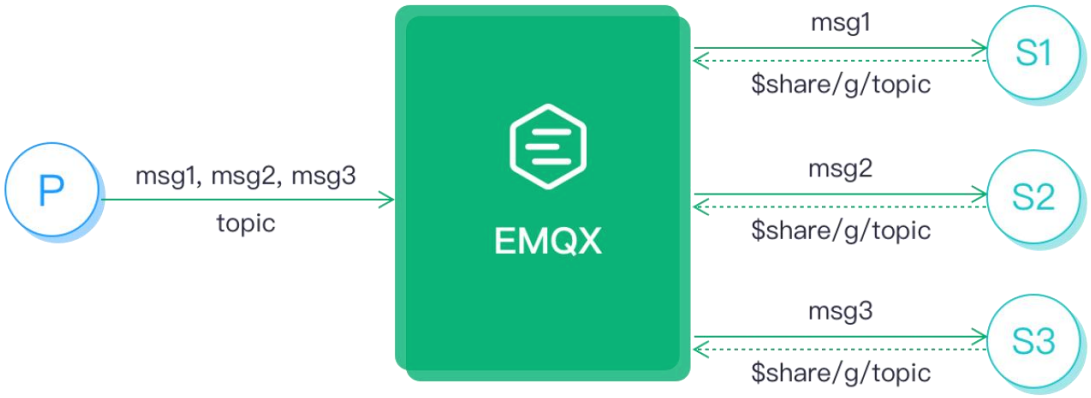
EMQX 还支持客户端上下线事件、收发流量、消息收发、系统监控等丰富的系统主题，用户可通过订阅 `$SYS/#` 主题获取所有系统主题消息。详细请见：EMQX 系统主题文档。

## 共享订阅

共享订阅是 MQTT 5.0 引入的新特性，用于在多个订阅者之间实现订阅的负载均衡，MQTT 5.0 规定的共享订阅主题以 `$share` 开头。

> 虽然 MQTT 协议在 5.0 版本才引入共享订阅，但是 EMQX 从 MQTT 3.1.1 版本开始就支持共享订阅。

下图中，3 个订阅者用共享订阅的方式订阅了同一个主题 `$share/g/topic`，其中 `topic` 是它们订阅的真实主题名，而 `$share/g/` 是共享订阅前缀（`g/` 是群组名，可为任意 UTF-8 编码字符串）。




另外，对于 MQTT 5.0 以下的版本，EMQX 还支持不带群组的共享订阅前缀 `$queue`，关于共享订阅的更多详情请查看 EMQX 共享订阅文档。

## 不同场景中的主题设计

### 智能家居

比如我们用传感器监测卧室、客厅以及厨房的温度、湿度和空气质量，可以设计以下几个主题：

- `myhome/bedroom/temperature`
- `myhome/bedroom/humidity`
- `myhome/bedroom/airquality`
- `myhome/livingroom/temperature`
- `myhome/livingroom/humidity`
- `myhome/livingroom/airquality`
- `myhome/kitchen/temperature`
- `myhome/kitchen/humidity`
- `myhome/kitchen/airquality`

接下来，可以通过订阅 `myhome/bedroom/+` 主题获取卧室的温度、湿度及空气质量数据，订阅 `myhome/+/temperature` 主题获取三个房间的温度数据，订阅 `myhome/#` 获取所有的数据。

### 充电桩

充电桩的上行主题格式为 `ocpp/cp/${cid}/notify/${action}`，下行主题格式为 `ocpp/cp/${cid}/reply/${action}`。

- `ocpp/cp/cp001/notify/bootNotification`

  充电桩上线时向该主题发布上线请求。

- `ocpp/cp/cp001/notify/startTransaction`

  向该主题发布充电请求。

- `ocpp/cp/cp001/reply/bootNotification`

  充电桩上线前需订阅该主题接收上线应答。

- `ocpp/cp/cp001/reply/startTransaction`

  充电桩发起充电请求前需订阅该主题接收充电请求应答。

## 即时消息

- `chat/user/${user_id}/inbox`

  **一对一聊天**：用户上线后订阅该收件箱主题 ，将能接收到好友发送给自己的消息。给好友回复消息时，只需要将该主题的 `user_id` 换为好友的的 id 即可。

- `chat/group/${group_id}/inbox`

  **群聊**：用户加群成功后，可订阅该主题获取对应群组的消息，回复群聊时直接给该主题发布消息即可。

- `req/user/${user_id}/add`

  **添加好友**：可向该主题发布添加好友的申请（`user_id` 为对方的 id）。

  **接收好友请求**：用户可订阅该主题（`user_id` 为自己的 id）接收其他用户发起的好友请求。

- `resp/user/${user_id}/add`

  **接收好友请求的回复**：用户添加好友前，需订阅该主题接收请求结果（`user_id` 为自己的 id）。

  **回复好友申请**：用户向该主题发送消息表明是否同意好友申请（`user_id` 为对方的 id）。

- `user/${user_id}/state`

  **用户在线状态**：用户可以订阅该主题获取好友的在线状态。

# MQTT 主题常见问题及解答

## 主题的层级及长度有什么限制吗?

MQTT 协议规定主题的长度为两个字节，因此主题最多可包含 65,535 个字符。

建议主题层级为 7 个以内。使用较短的主题名称和较少的主题层级意味着较少的资源消耗，例如 `my-home/room1/data` 比 `my/home/room1/data` 更好。

## 服务器对主题数量有限制吗?

不同消息服务器对最大主题数量的支持各不一致，目前 EMQX 的默认配置对主题数量没有限制，但是主题数量越多将会消耗越多的服务器内存。考虑到连接到 MQTT Broker 的设备数量一般较多，我们建议一个客户端订阅的主题数量最好控制在 10 个以内。

## 通配符主题订阅与普通主题订阅性能是否一致?

通配符主题订阅的性能弱于普通主题订阅，且会消耗更多的服务器资源，用户可根据实际业务情况选择订阅类型。

## 同一个主题能被共享订阅与普通订阅同时使用吗?

可以，但是不建议同时使用。

## 常见的 MQTT 主题使用建议有哪些?

- 不建议使用 `#` 订阅所有主题；
- 不建议主题以 `/` 开头或结尾，例如 `/chat` 或 `chat/`；
- 不建议在主题里添加空格及非 ASCII 特殊字符；
- 同一主题层级内建议使用下划线 `_` 或横杆 `-` 连接单词（或者使用驼峰命名）；
- 尽量使用较少的主题层级；
- 当使用通配符时，将唯一值的主题层（例如设备号）越靠近第一层越好。例如，`device/00000001/command/#` 比 `device/command/00000001/#` 更好。

# 持久会话

## 什么是 MQTT 持久会话?

不稳定的网络及有限的硬件资源是物联网应用需要面对的两大难题，MQTT 客户端与服务器的连接可能随时会因为网络波动及资源限制而异常断开。为了解决网络连接断开对通信造成的影响，MQTT 协议提供了持久会话功能。

MQTT 客户端在发起到服务器的连接时，可以设置是否创建一个持久会话。持久会话会保存一些重要的数据，以使会话能在多个网络连接中继续。持久会话主要有以下三个作用：

- 避免因网络中断导致需要反复订阅带来的额外开销。
- 避免错过离线期间的消息。
- 确保 QoS 1 和 QoS 2 的消息质量保证不被网络中断影响。

## 持久会话需要存储哪些数据?

通过上文我们知道持久会话需要存储一些重要的数据，以使会话能被恢复。这些数据有的存储在客户端，有的则存储在服务端。

客户端中存储的会话数据：

- 已发送给服务端，但是还没有完成确认的 QoS 1 与 QoS 2 消息。
- 从服务端收到的，但是还没有完成确认的 QoS 2 消息。

服务端中存储的会话数据：

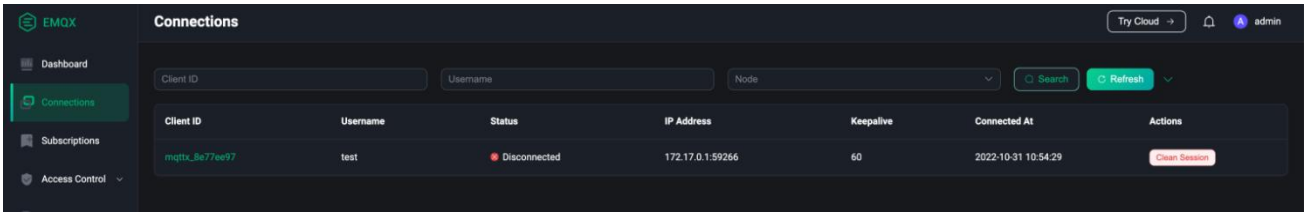
- 会话是否存在，即使会话状态其余部分为空。
- 已发送给客户端，但是还没有完成确认的 QoS 1 与 QoS 2 消息。
- 等待传输给客户端的 QoS 0 消息（可选），QoS 1 与 QoS 2 消息。
- 从客户端收到的，但是还没有完成确认的 QoS 2 消息，遗嘱消息和遗嘱延时间隔。

# MQTT Clean Session 的使用

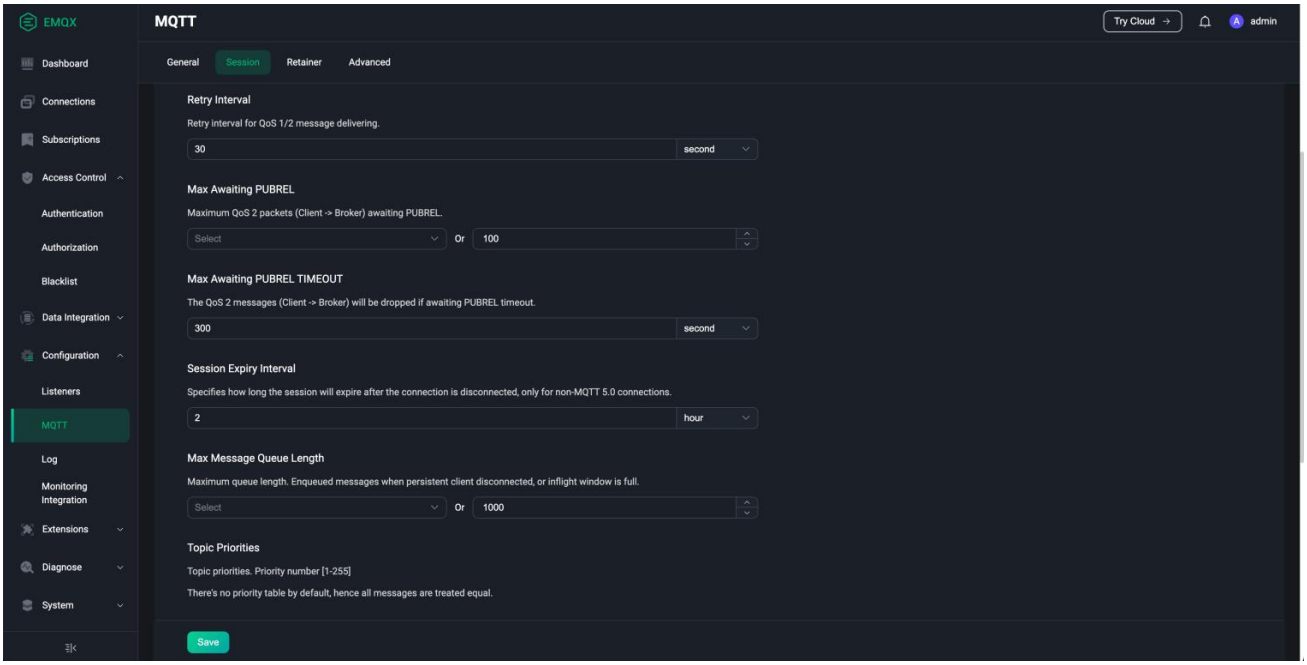
Clean Session 是用来控制会话状态生命周期的标志位，为 **true** 时表示创建一个新的会话，在客户端断开连接时，会话将自动销毁。为 **false** 时表示创建一个持久会话，在客户端断开连接后会话仍然保持，直到会话超时注销。

> 注意： 持久会话能被恢复的前提是客户端使用固定的 Client ID 再次连接，如果 Client ID 是动态的，那么连接成功后将会创建一个新的持久会话。

如下为开源 MQTT 服务器 EMQX 的 Dashboard，可以看到图中的连接虽然是断开状态，但是因为它是持久会话，所以仍然能被查看到，并且可以在 Dashboard 中手动清除该会话。




同时，EMQX 也支持在 Dashboard 中设置 Session 相关参数。

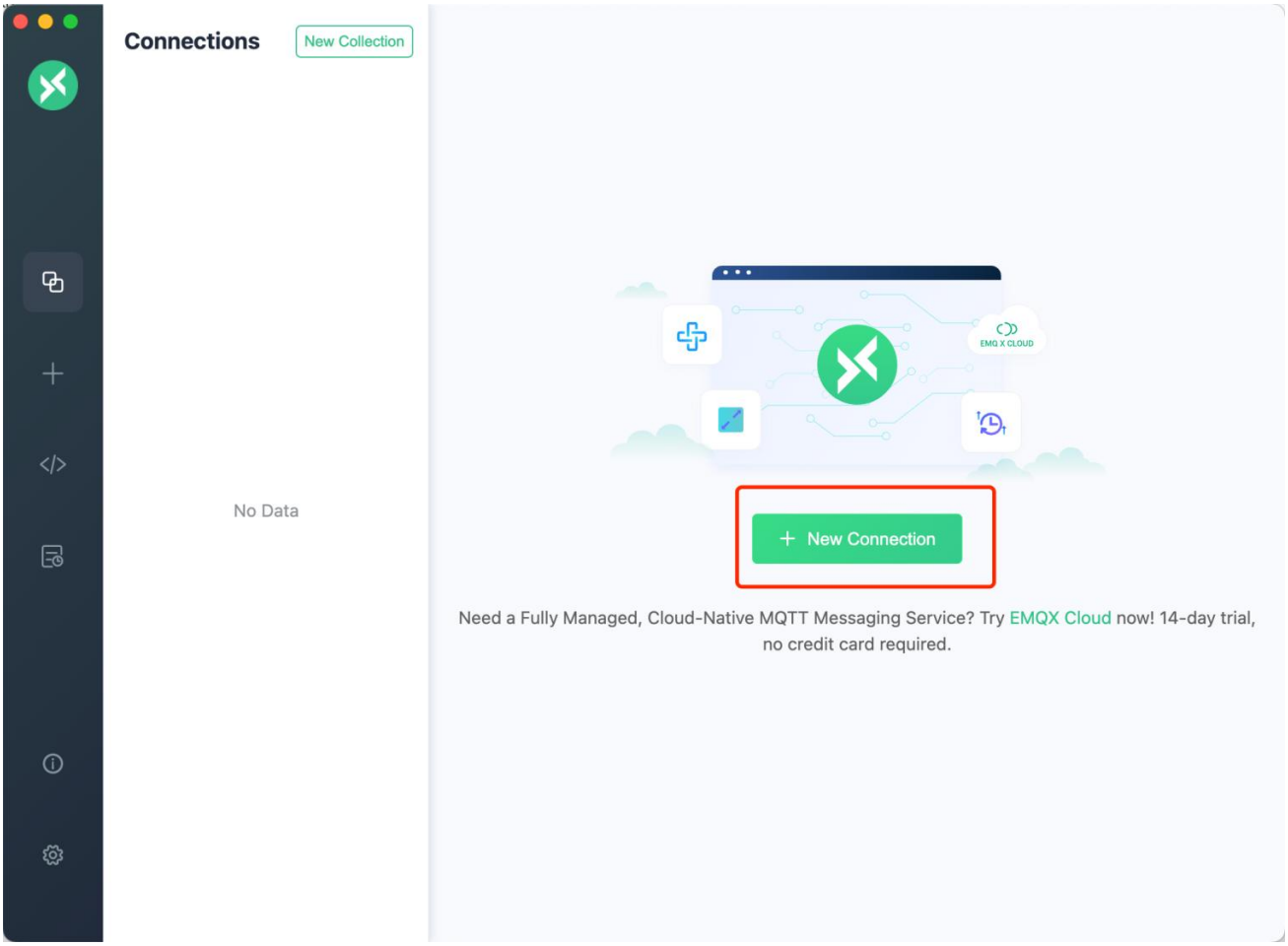



MQTT 3.1.1 没有规定持久会话应该在什么时候过期，如果仅从协议层面理解的话，这个持久会话应该永久存在。但在实际场景中这并不现实，因为它会非常占用服务端的资源，所以服务端通常不会完全遵循协议来实现，而是向用户提供一个全局配置来限制会话的过期时间。

比如 EMQ 提供的 免费的公共 MQTT 服务器 设置的会话过期时间为 5 分钟，最大消息数为 1000 条，且不保存 QoS 0 消息。

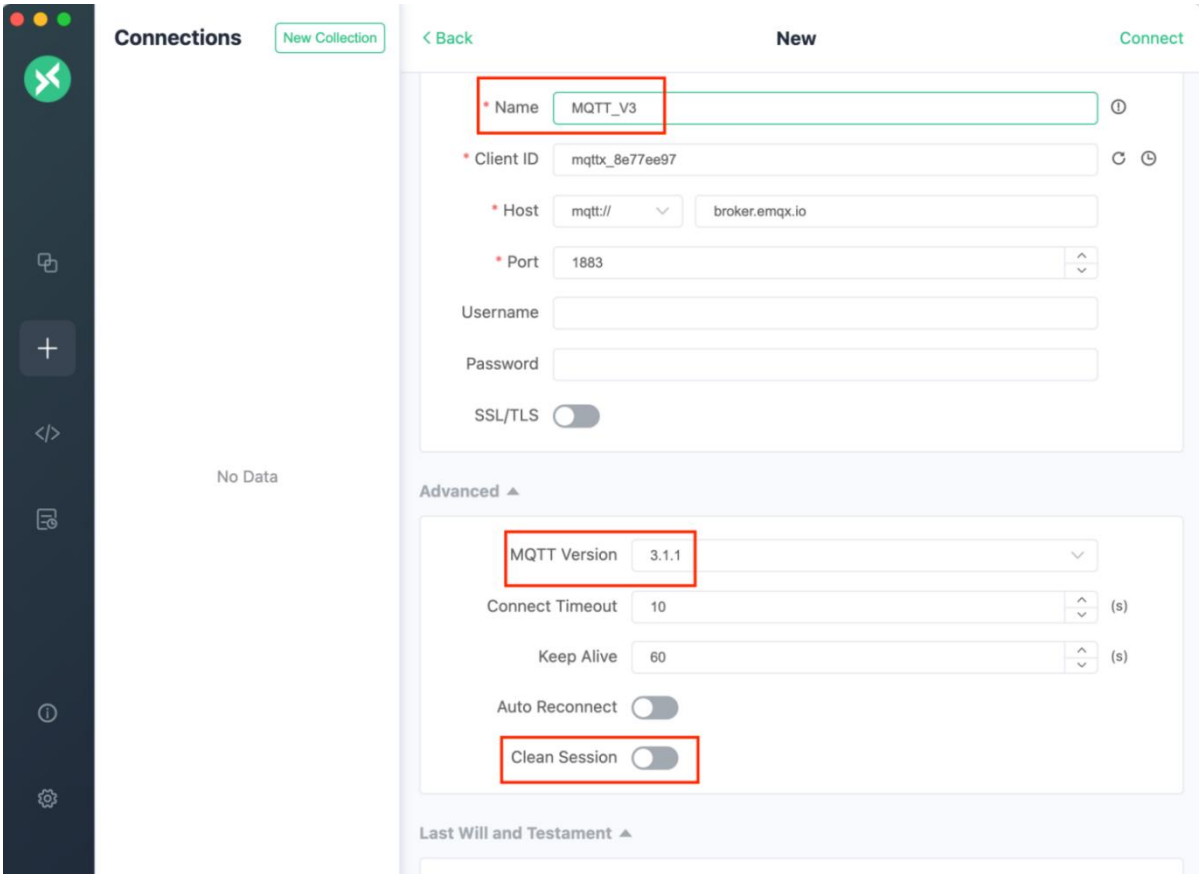
接下来我们使用开源的跨平台 MQTT 5.0 桌面客户端工具 – MQTTX 演示 Clean Session 的使用。

打开 MQTTX 后如下所示，点击 **New Connection** 按钮创建一个 MQTT 连接。

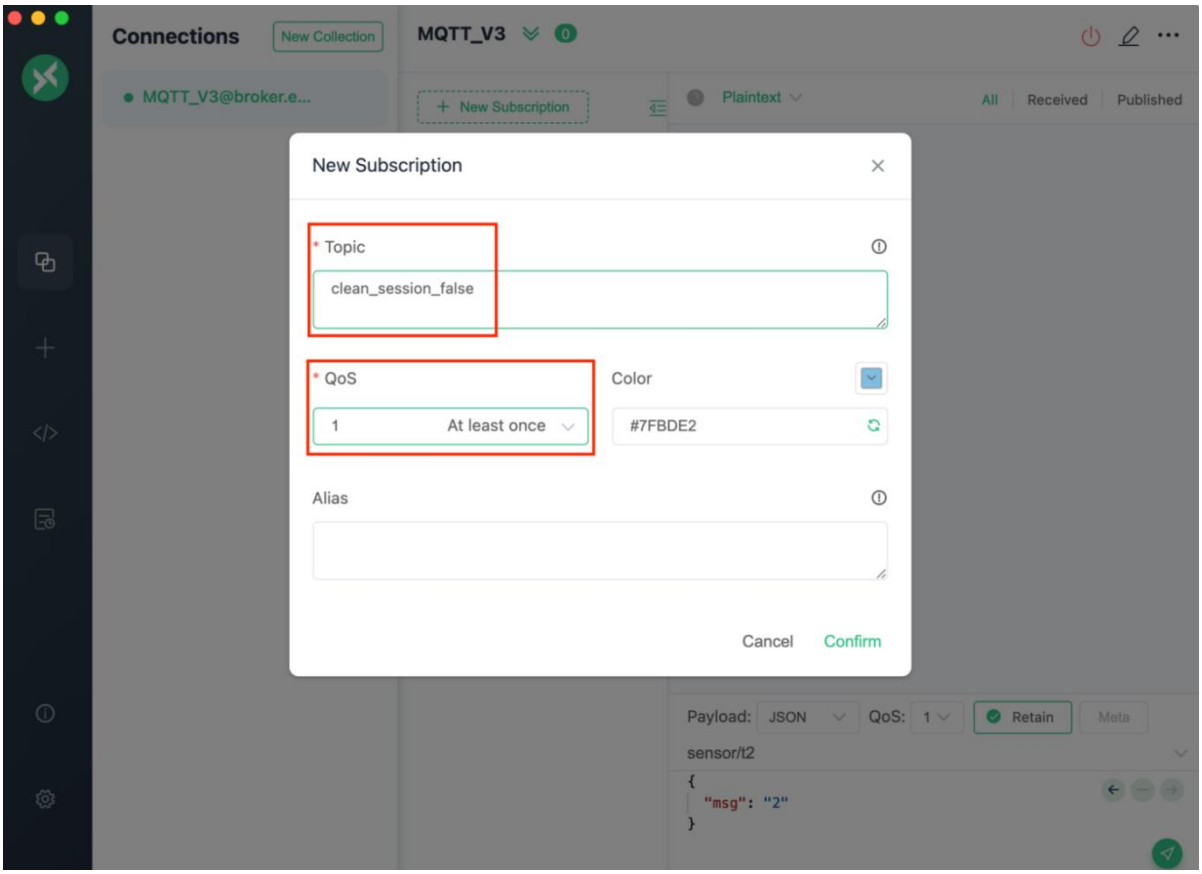



创建一个名为 **MQTT_V3** 的连接，Clean Session 为关闭状态（即为 false），MQTT 版本选择 3.1.1，然后点击右上角的 **Connect** 按钮。

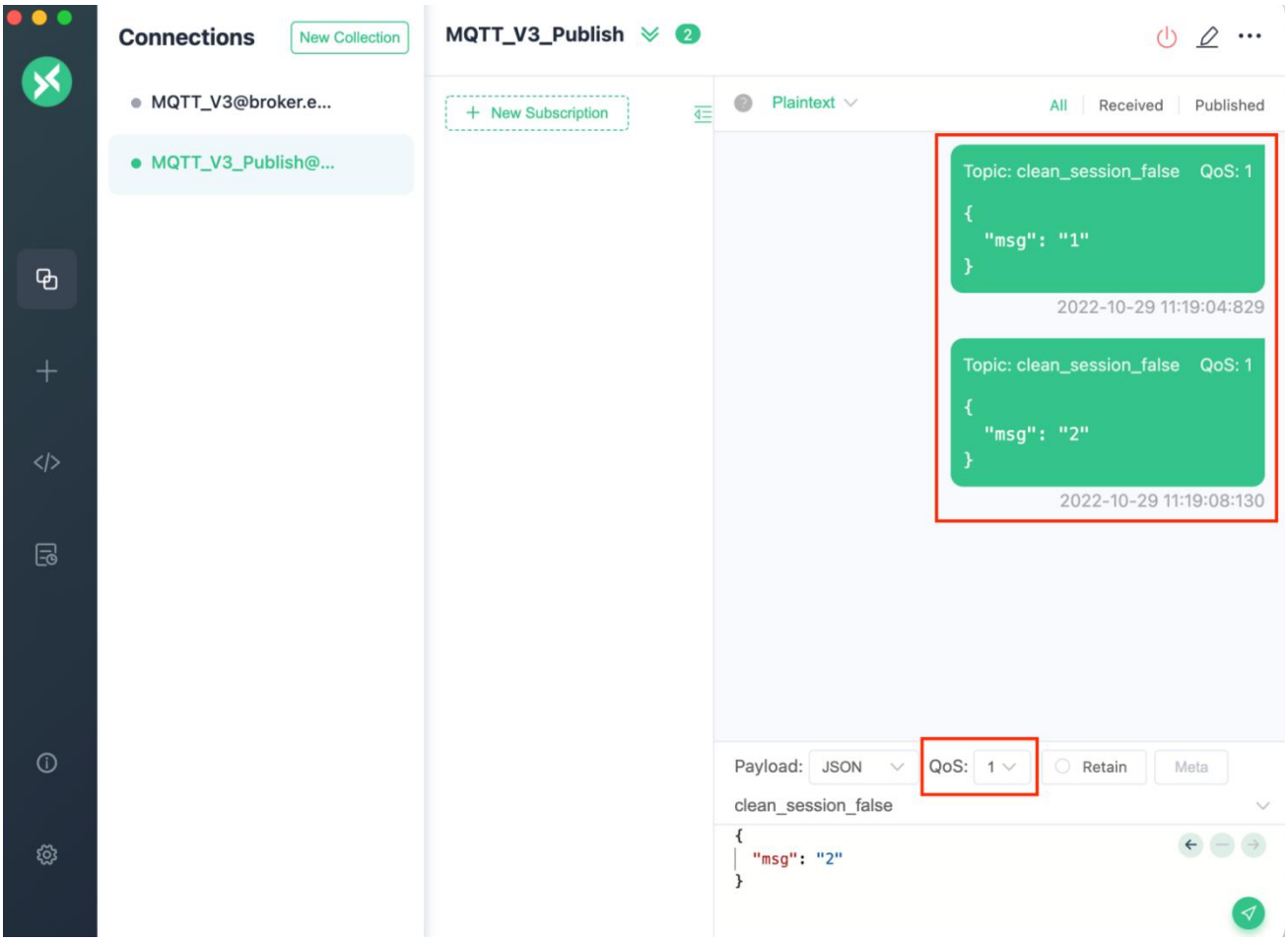
> 连接的服务器默认为 EMQ 提供的 免费的公共 MQTT 服务器。



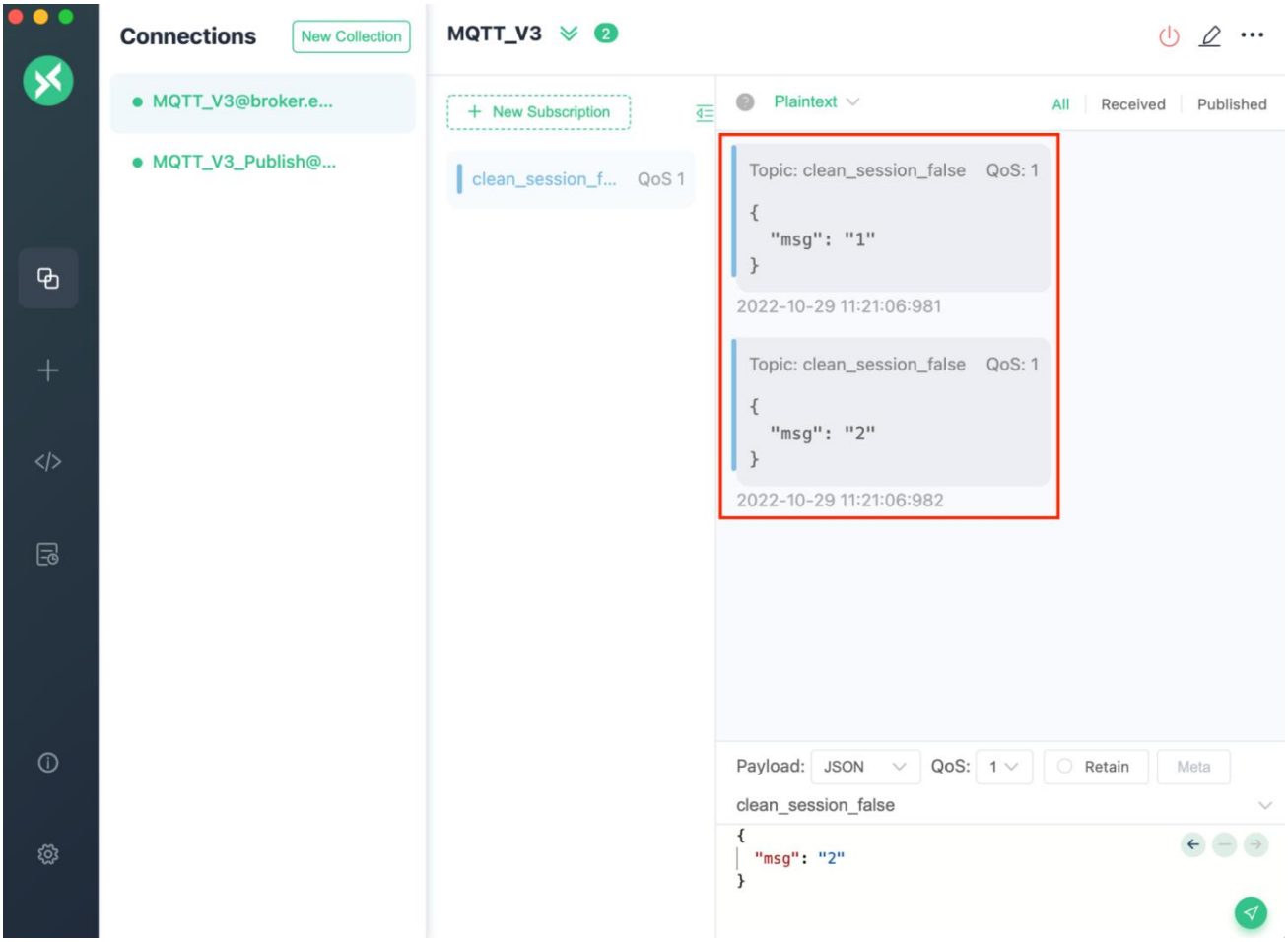

连接成功后订阅 **clean_session_false** 主题，且 QoS 设置为 1。



订阅成功后，点击右上角的断开连接按钮。然后，创建一个名为 **MQTT_V3_Publish** 的连接，MQTT 版本同样设置为 3.1.1，连接成功后向 **clean_session_false** 主题发布两条 QoS 1 消息。




然后选中 MQTT_V3 连接，点击连接按钮连接至服务器，将会成功接收到两条离线期间的消息。




## MQTT 5.0 中的会话改进

MQTT 5.0 中将 Clean Session 拆分成了 Clean Start 与 Session Expiry Interval。Clean Start 用于指定连接时是创建一个全新的会话还是尝试复用一个已存在的会话，Session Expiry Interval 用于指定网络连接断开后会话的过期时间。

Clean Start 为 **true** 时表示必须丢弃任何已存在的会话，并创建一个全新的会话；为 **false** 时表示必须使用与 Client ID 关联的会话来恢复与客户端的通信（除非会话不存在）。

Session Expiry Interval 解决了 MQTT 3.1.1 中持久会话永久存在造成的服务器资源浪费问题。设置为 0 或未设置，表示断开连接时会话即到期；设置为大于 0 的数值，则表示会话在网络连接关闭后会保持多少秒；设置为 **0xFFFFFFFF** 表示会话永远不会过期。

# 关于 MQTT 会话的 Q&A

## 当会话结束后，保留消息还存在么？

MQTT 保留消息不是会话状态的一部分，它们不会在会话结束时被删除。

## 客户端如何知道当前会话是被恢复的会话？

MQTT 协议从 v3.1.1 开始，就为 CONNACK 报文设计了 Session Present 字段。当服务器返回的该字段值为 1 时，表示当前连接将会复用服务器保存的会话。客户端可通过该字段值决定在连接成功后是否需要重新订阅。

## 使用持久会话时有哪些建议？

- 不能使用动态 Client ID，需要保证客户端每次连接的 Client ID 都是固定的。
- 根据服务器性能、网络状况、客户端类型等合理评估会话过期时间。设置过长会占用更多的服务端资源，设置过短会导致未重连成功会话就失效。
- 当客户端确定不再需要会话时，可使用 Clean Session 为 true 进行重连，重连成功后再断开连接。如果是 MQTT 5.0 则可在断开连接时直接设置 Session Expiry Interval 为 0，表示连接断开后会话即失效。

# QoS 0, 1, 2

## 什么是 QoS

很多时候，使用 MQTT 协议的设备都运行在网络受限的环境下，而只依靠底层的 TCP 传输协议，并不能完全保证消息的可靠到达。因此，MQTT 提供了 QoS 机制，其核心是设计了多种消息交互机制来提供不同的服务质量，来满足用户在各种场景下对消息可靠性的要求。

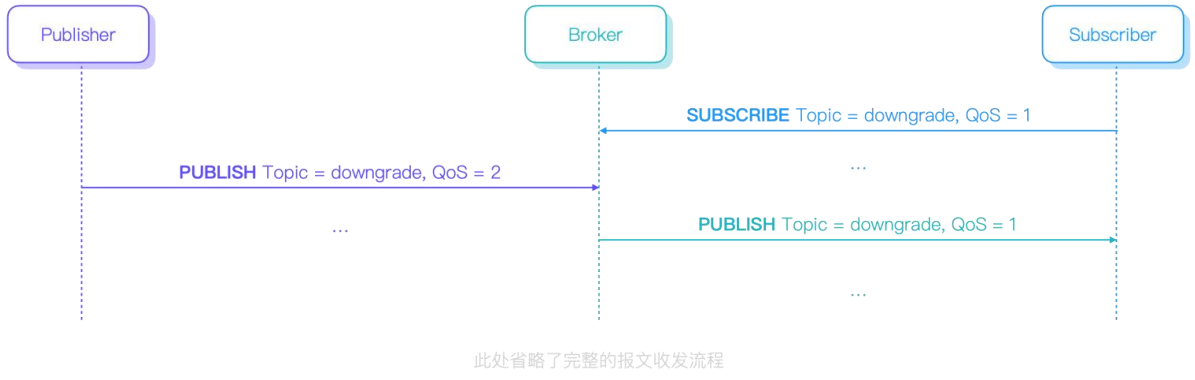
MQTT 定义了三个 QoS 等级，分别为：

- QoS 0，最多交付一次。
- QoS 1，至少交付一次。
- QoS 2，只交付一次。

其中，使用 QoS 0 可能丢失消息，使用 QoS 1 可以保证收到消息，但消息可能重复，使用 QoS 2 可以保证消息既不丢失也不重复。QoS 等级从低到高，不仅意味着消息可靠性的提升，也意味着传输复杂程度的提升。

在一个完整的从发布者到订阅者的消息投递流程中，QoS 等级是由发布者在 PUBLISH 报文中指定的，大部分情况下 Broker 向订阅者转发消息时都会维持原始的 QoS 不变。不过也有一些例外的情况，根据订阅者的订阅要求，消息的 QoS 等级可能会在转发的时候发生降级。

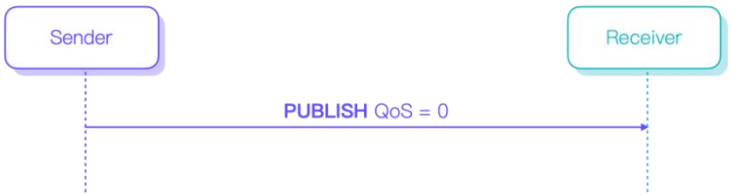
例如，订阅者在订阅时要求 Broker 可以向其转发的消息的最大 QoS 等级为 QoS 1，那么后续所有 QoS 2 消息都会降级至 QoS 1 转发给此订阅者，而所有 QoS 0 和 QoS 1 消息则会保持原始的 QoS 等级转发。



接下来，让我们来看看 MQTT 中每个 QoS 等级的具体原理。

## QoS 0 – 最多交付一次

QoS 0 是最低的 QoS 等级。QoS 0 消息即发即弃，不需要等待确认，不需要存储和重传，因此对于接收方来说，永远都不需要担心收到重复的消息。




### 为什么 QoS 0 消息会丢失?

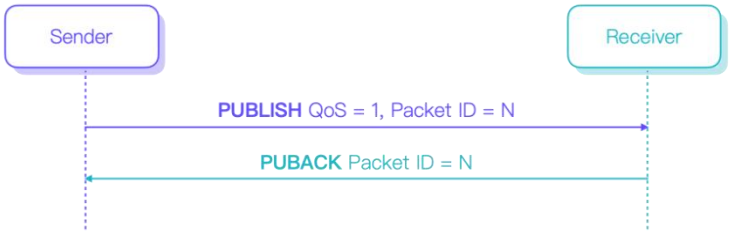
当我们使用 QoS 0 传递消息时，消息的可靠性完全依赖于底层的 TCP 协议。

而 TCP 只能保证在连接稳定不关闭的情况下消息的可靠到达，一旦出现连接关闭、重置，仍有可能丢失当前处于网络链路或操作系统底层缓冲区中的消息。这也是 QoS 0 消息最主要的丢失场景。

## QoS 1 – 至少交付一次

为了保证消息到达，QoS 1 加入了应答与重传机制，发送方只有在收到接收方的 PUBACK 报文以后，才能认为消息投递成功，在此之前，发送方需要存储该 PUBLISH 报文以便下次重传。

QoS 1 需要在 PUBLISH 报文中设置 Packet ID，而作为响应的 PUBACK 报文，则会使用与 PUBLISH 报文相同的 Packet ID，以便发送方收到后删除正确的 PUBLISH 报文缓存。



## 为什么 QoS 1 消息会重复?

对于发送方来说，没收到 PUBACK 报文分为以下两种情况：

1. PUBLISH 未到达接收方
2. PUBLISH 已经到达接收方，接收方的 PUBACK 报文还未到达发送方

在第一种情况下，发送方虽然重传了 PUBLISH 报文，但是对于接收方来说，实际上仍然仅收到了一次消息。

但是在第二种情况下，在发送方重传时，接收方已经收到过了这个 PUBLISH 报文，这就导致接收方将收到重复的消息。


虽然重传时 PUBLISH 报文中的 DUP 标志会被设置为 1，用以表示这是一个重传的报文。但是接收方并不能因此假定自己曾经接收过这个消息，仍然需要将其视作一个全新的消息。

这是因为对于接收方来说，可能存在以下两种情况：

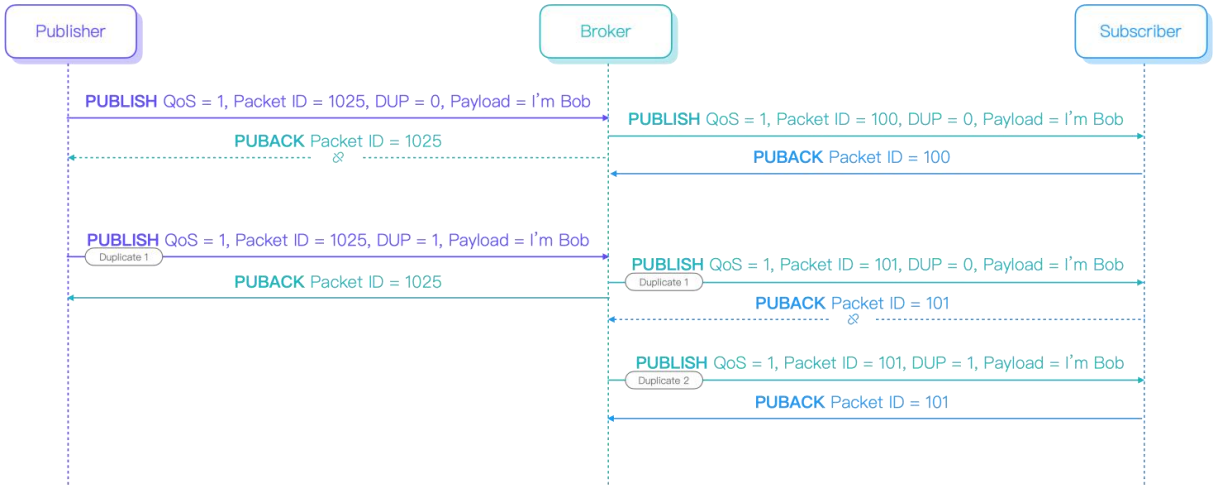

第一种情况，发送方由于没有收到 PUBACK 报文而重传了 PUBLISH 报文。此时，接收方收到的前后两个 PUBLISH 报文使用了相同的 Packet ID，并且第二个 PUBLISH 报文的 DUP 标志为 1，此时它确实是一个重复的消息。

第二种情况，第一个 PUBLISH 报文已经完成了投递，1024 这个 Packet ID 重新变为可用状态。发送方使用这个 Packet ID 发送了一个全新的 PUBLISH 报文，但这一次报文未能到达对端，所以发送方后续重传了这个 PUBLISH 报文。这就使得虽然接收方收到的第二个 PUBLISH 报文同样是相同的 Packet ID，并且 DUP 为 1，但确实是一个全新的消息。

由于我们无法区分这两种情况，所以只能让接收方将这些 PUBLISH 报文都当作全新的消息来处理。因此当我们使用 QoS 1 时，消息的重复在协议层面上是无法避免的。

甚至在比较极端的情况下，例如 Broker 从发布方收到了重复的 PUBLISH 报文，而在将这些报文转发给订阅方的过程中，再次发生重传，这将导致订阅方最终收到更多的重复消息。

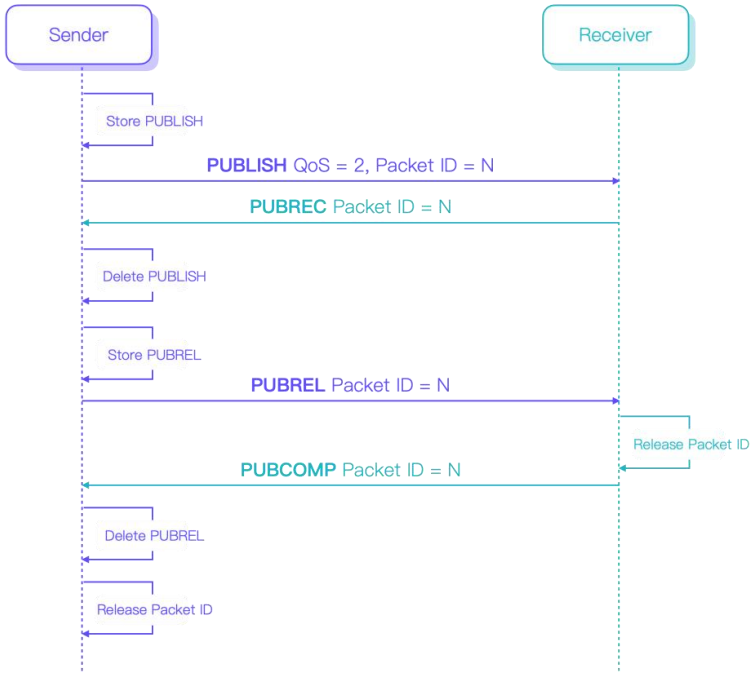
在下图表示的例子中，虽然发布者的本意只是发布一条消息，但对接收方来说，最终却收到了三条相同的消息：




以上，就是 QoS 1 保证消息到达带来的副作用。

## QoS 2 – 只交付一次

QoS 2 解决了 QoS 0、1 消息可能丢失或者重复的问题，但相应地，它也带来了最复杂的交互流程和最高的开销。每一次的 QoS 2 消息投递，都要求发送方与接收方进行至少两次请求/响应流程。




1. 首先，发送方存储并发送 QoS 为 2 的 PUBLISH 报文以启动一次 QoS 2 消息的传输，然后等待接收方回复 PUBREC 报文。这一部分与 QoS 1 基本一致，只是响应报文从 PUBACK 变成了 PUBREC。

2. 当发送方收到 PUBREC 报文，即可确认对端已经收到了 PUBLISH 报文，发送方将**不再需要重传**这个报文，并且也**不能再重传**这个报文。所以此时发送方可以删除本地存储的 PUBLISH 报文，然后发送一个 PUBREL 报文，通知对端自己准备将本次使用的 Packet ID 标记为可用了。与 PUBLISH 报文一样，我们需要确保 PUBREL 报文到达对端，所以也需要一个响应报文，并且这个 PUBREL 报文需要被存储下来以便后续重传。

3. 当接收方收到 PUBREL 报文，也可以确认在这一次的传输流程中不会再有重传的 PUBLISH 报文到达，因此回复 PUBCOMP 报文表示自己也准备好将当前的 Packet ID 用于新的消息了。

4. 当发送方收到 PUBCOMP 报文，这一次的 QoS 2 消息传输就算正式完成了。在这之后，发送方可以再次使用当前的 Packet ID 发送新的消息，而接收方再次收到使用这个 Packet ID 的 PUBLISH 报文时，也会将它视为一个全新的消息。

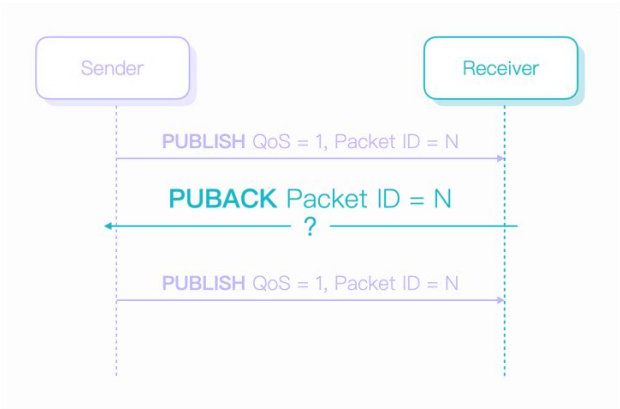
## 为什么 QoS 2 消息不会重复?

QoS 2 消息保证不会丢失的逻辑与 QoS 1 相同，所以这里我们就不再重复了。

与 QoS 1 相比，QoS 2 新增了 PUBREL 报文和 PUBCOMP 报文的流程，也正是这个新增的流程带来了消息不会重复的保证。

在我们更进一步之前，我们先快速回顾一下 QoS 1 消息无法避免重复的原因。

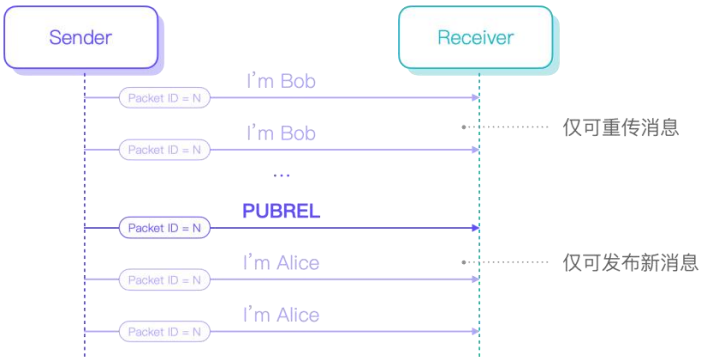
当我们使用 QoS 1 消息时，对接收方来说，回复完 PUBACK 这个响应报文以后 Packet ID 就重新可用了，也不管响应是否确实已经到达了发送方。所以就无法得知之后到达的，携带了相同 Packet ID 的 PUBLISH 报文，到底是发送方因为没有收到响应而重传的，还是发送方因为收到了响应所以重新使用了这个 Packet ID 发送了一个全新的消息。



所以，消息去重的关键就在于，通信双方如何正确地同步释放 Packet ID，换句话说，不管发送方是重传消息还是发布新消息，一定是和对端达成共识了的。

而 QoS 2 中增加的 PUBREL 流程，正是提供了帮助通信双方协商 Packet ID 何时可以重用的能力。


QoS 2 规定，发送方只有在收到 PUBREC 报文之前可以重传 PUBLISH 报文。一旦收到 PUBREC 报文并发出 PUBREL 报文，发送方就进入了 Packet ID 释放流程，不可以再使用当前 Packet ID 重传 PUBLISH 报文。同时，在收到对端回复的 PUBCOMP 报文确认双方都完成 Packet ID 释放之前，也不可以使用当前 Packet ID 发送新的消息。




因此，对于接收方来说，能够以 PUBREL 报文为界限，凡是在 PUBREL 报文之前到达的 PUBLISH 报文，都必然是重复的消息；而凡是在 PUBREL 报文之后到达的 PUBLISH 报文，都必然是全新的消息。

一旦有了这个前提，我们就能够在协议层面完成 QoS 2 消息的去重。

# 不同 QoS 的适用场景和注意事项

## QoS 0

QoS 0 的缺点是可能会丢失消息，消息丢失的频率依赖于你所处的网络环境，并且可能使你错过断开连接期间的消息，不过优点是投递的效率较高。

所以我们通常选择使用 QoS 0 传输一些高频且不那么重要的数据，比如传感器数据，周期性更新，即使遗漏几个周期的数据也可以接受。

## QoS 1

QoS 1 可以保证消息到达，所以适合传输一些较为重要的数据，比如下达关键指令、更新重要的有实时性要求的状态等。

但因为 QoS 1 还可能会导致消息重复，所以当我们选择使用 QoS 1 时，还需要能够处理消息的重复，或者能够允许消息的重复。

在我们决定使用 QoS 1 并且不对其进行去重处理之前，我们需要先了解，允许消息的重复，可能意味着什么。

如果我们不对 QoS 1 进行去重处理，我们可能会遭遇这种情况，发布方以 1、2 的顺序发布消息，但最终订阅方接收到的消息顺序可能是 1、2、1、2。如果 1 表示开灯指令，2 表示关灯指令，我想大部分用户都不会接受自己仅仅进行了开灯然后关灯的操作，结果灯在开和关的状态来回变化。


## QoS 2

QoS 2 既可以保证消息到达，也可以保证消息不会重复，但传输成本最高。如果我们不愿意自行实现去重方案，并且能够接受 QoS 2 带来的额外开销，那么 QoS 2 将是一个合适的选择。通常我们会在金融、航空等行业场景下会更多地见到 QoS 2 的使用。

# 关于 MQTT QoS 的 Q&A

## 如何为 QoS 1 消息去重?

在我们介绍 QoS 1 的时候讲到, QoS 1 消息的重复在协议层面上是无法避免的。所以如果我们想要对 QoS 1 消息进行去重，只能从业务层面入手。

一个比较常用且简单的方法是, 在每个 PUBLISH 报文的 Payload 中都带上一个时间戳或者一个单调递增的计数，这样上层业务就可以根据当前收到消息中的时间戳或计数是否大于自己上一次接收的消息中的时间戳或计数来判断这是否是一个新消息。

## 何时向后分发 QoS 2 消息?

我们已经了解到，QoS 2 的流程是非常长的，为了不影响消息的实时性，我们可以在第一次收到 PUBLISH 报文时, 就启动消息的向后分发。当然一旦开始向后分发, 后续收到在 PUBREL 报文之前到达的 PUBLISH 报文，都不能再重复分发操作，以免消息重复。

## 不同 QoS 的性能有差距么?

以 EMQX 为例，在相同的硬件配置下进行点对点通信，通常 QoS 0 与 QoS 1 能够达到的吞吐比较接近，不过 QoS 1 的 CPU 占用会略高于 QoS 0，负载较高时，QoS 1 的消息延迟也会进一步增加。而 QoS 2 能够达到的吞吐一般仅为 QoS 0、1 的一半左右。

# 遗嘱消息

遗嘱消息是 MQTT 为那些可能出现**意外断线**的设备提供的将**遗嘱**优雅地发送给第三方的能力。意外断线包括但不限于：

- 因网络故障或网络波动，设备在保持连接周期内未能通讯，连接被服务端关闭
- 设备意外掉电
- 设备尝试进行不被允许的操作而被服务端关闭连接，例如订阅自身权限以外的主题等

遗嘱消息可以看作是一个简化版的 PUBLISH 消息，他也包含 Topic, Payload, QoS 等字段。遗嘱消息会在设备与服务端连接时，通过 CONNECT 报文指定，然后在设备意外断线时由服务端将该遗嘱消息发布到连接时指定的遗嘱主题（Will Topic）上。这也意味着服务端必须在回复 CONNACK 之前完成遗嘱消息的存储，以确保之后任一时刻发生意外断线的情况，服务端都能保证遗嘱消息被发布。

以下为遗嘱消息在 MQTT 5.0 和 MQTT 3.1 & 3.1.1 的差异：

| | MQTT 5.0 | MQTT 3.1 & 3.1.1 |
|---|---|---|
| Will Retain | Yes | Yes |
| Will QoS | Yes | Yes |
| Will Flag | Yes | Yes |
| Will Properties | Yes | No |
| Will Topic | Yes | Yes |
| Will Payload | Yes | Yes |

Will Retain、Will QoS、Will Topic 和 Will Payload 的用处与普通 PUBLISH 报文基本一致，这里不再赘述。

唯一值得一提的是 Will Retain 的使用场景，它是保留消息与遗嘱消息的结合。如果订阅该遗嘱主题（Will Topic）的客户端不能保证遗嘱消息发布时在线，那么建议为遗嘱消息设置 Will Retain，避免订阅端错过

遗嘱消息。

Will Flag 通常是 MQTT 协议实现方关心的字段，它用于标识 CONNECT 报文中是否会包含 Will Properties、Will Topic 等字段。

最后一个是 MQTT 5.0 新增的 Will Properties 字段，属性本身也是 MQTT 5.0 的一个新特性，不同类型的报文有着不同的属性，例如 CONNECT 报文有会话过期间隔（Session Expiry Interval）、最大报文长度（Maximum Packet Size）等属性，SUBSCRIBE 报文则有订阅标识符（Subscription Identifier）等属性。

Will Properties 中的消息过期间隔（Message Expiry Interval）等属性与 PUBLISH 报文中的用法基本一致，只有一个遗嘱延迟间隔（Will Delay Interval）是遗嘱消息特有的属性。

遗嘱延迟间隔顾名思义，就是在连接断开后延迟一段时间才发布遗嘱消息。它的一个重要用途就是避免在设备因网络波动短暂断开连接，但能够快速恢复连接继续提供服务时发出遗嘱消息，并对遗嘱消息订阅方造成困扰。

需要注意的是，具体延迟多久发布遗嘱消息，除了遗嘱延迟间隔，还受限于会话过期间隔，取决于两者谁先发生。所以当我们将会话过期间隔设置为 0 时，即会话在网络连接关闭时过期，那么不管遗嘱延迟间隔的值是多少，遗嘱消息都会在网络连接断开时立即发布。

## 演示遗嘱消息的使用

接下来我们使用 EMQX 和 MQTTX 来演示一下遗嘱消息的实际使用。

为了实现 MQTT 连接被异常断开的效果，我们需要调整一下 EMQX 的默认 ACL 规则与相关配置项：

首先在 **etc/acl.conf** 中添加以下 ACL 规则，表示拒绝本机客户端连接发布 test 主题。注意需要加在所有默认 ACL 规则之前，以确保这条规则能成功生效：

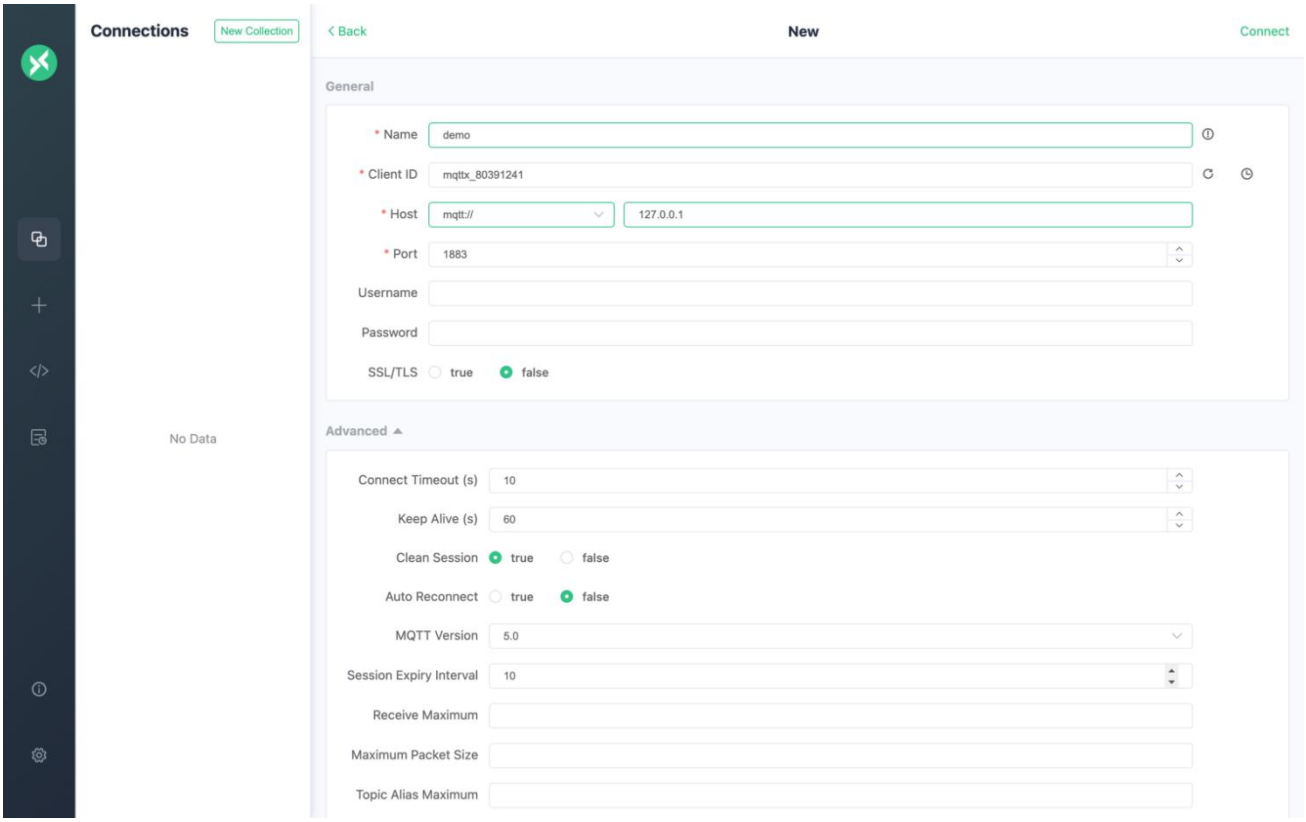
```
1.  {deny, {ipaddr, "127.0.0.1"}, publish, ["test"]}.
```

然后修改 **etc/emqx.conf** 中 **zone.internal.acl_deny_action** 配置项，将其设置为 ACL 检查拒绝时断开客户端连接：

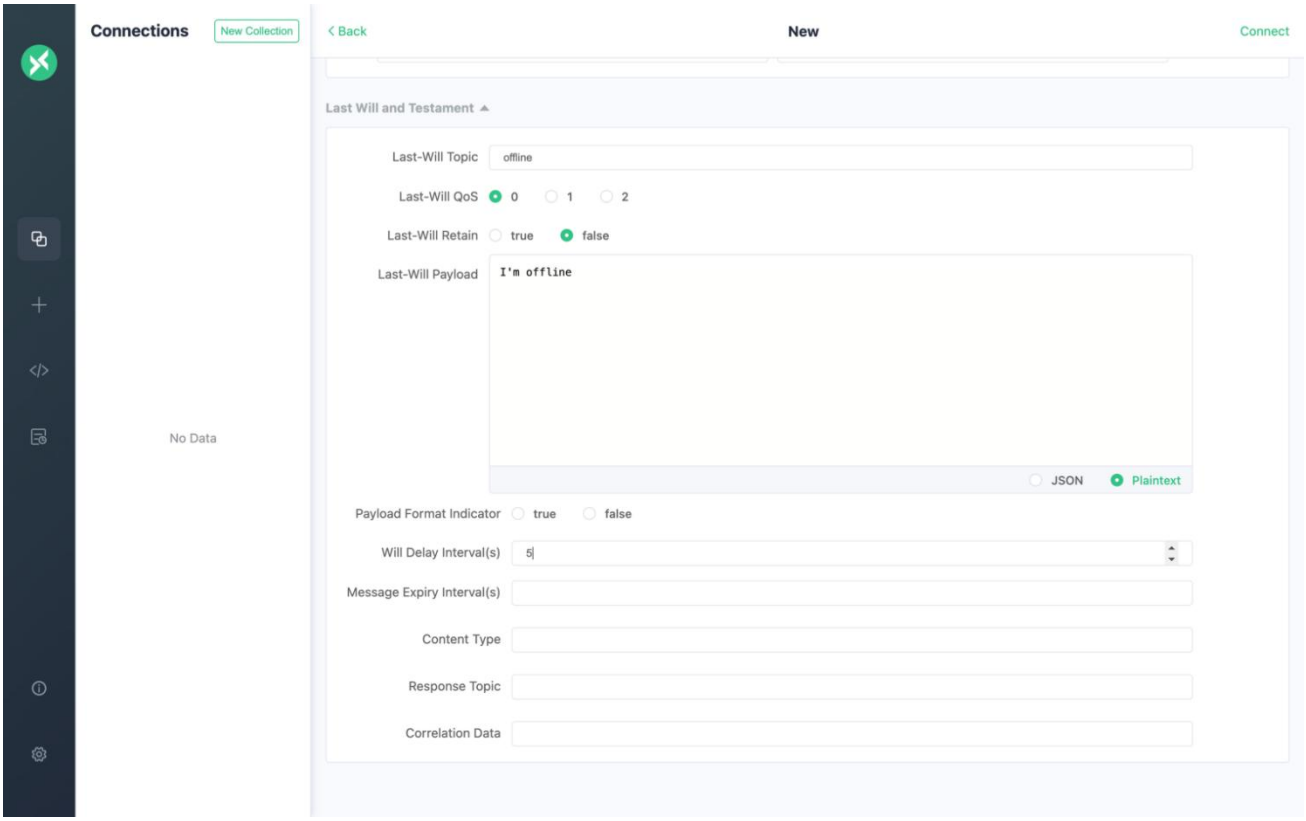
```
1.  zone.internal.acl_deny_action = disconnect
```

完成以上修改后，我们启动 EMQX。

接下来，我们在 MQTTX 中新建一个名为 demo 的连接，Host 修改为 localhost，在 Advanced 部分选择 MQTT Version 为 5.0，并且将 Session Expiry Interval 设置为 10，确保会话不会在遗嘱消息发布前过期。

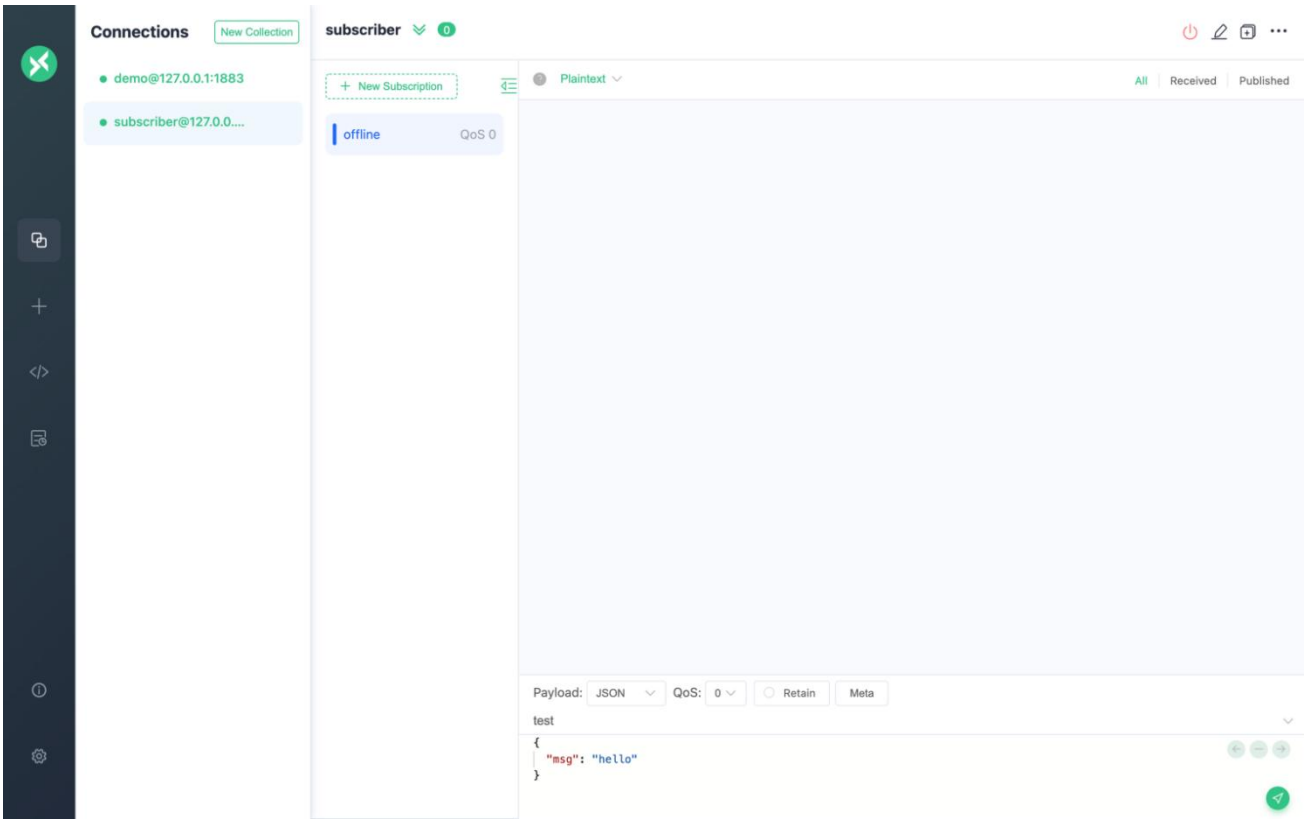



然后在 Lass Will and Testament 部分将 Last-Will Topic 设置为 offline，Last-Will Payload 设置为 **I'm offline**，Will Delay Interval (s) 设置为 5。

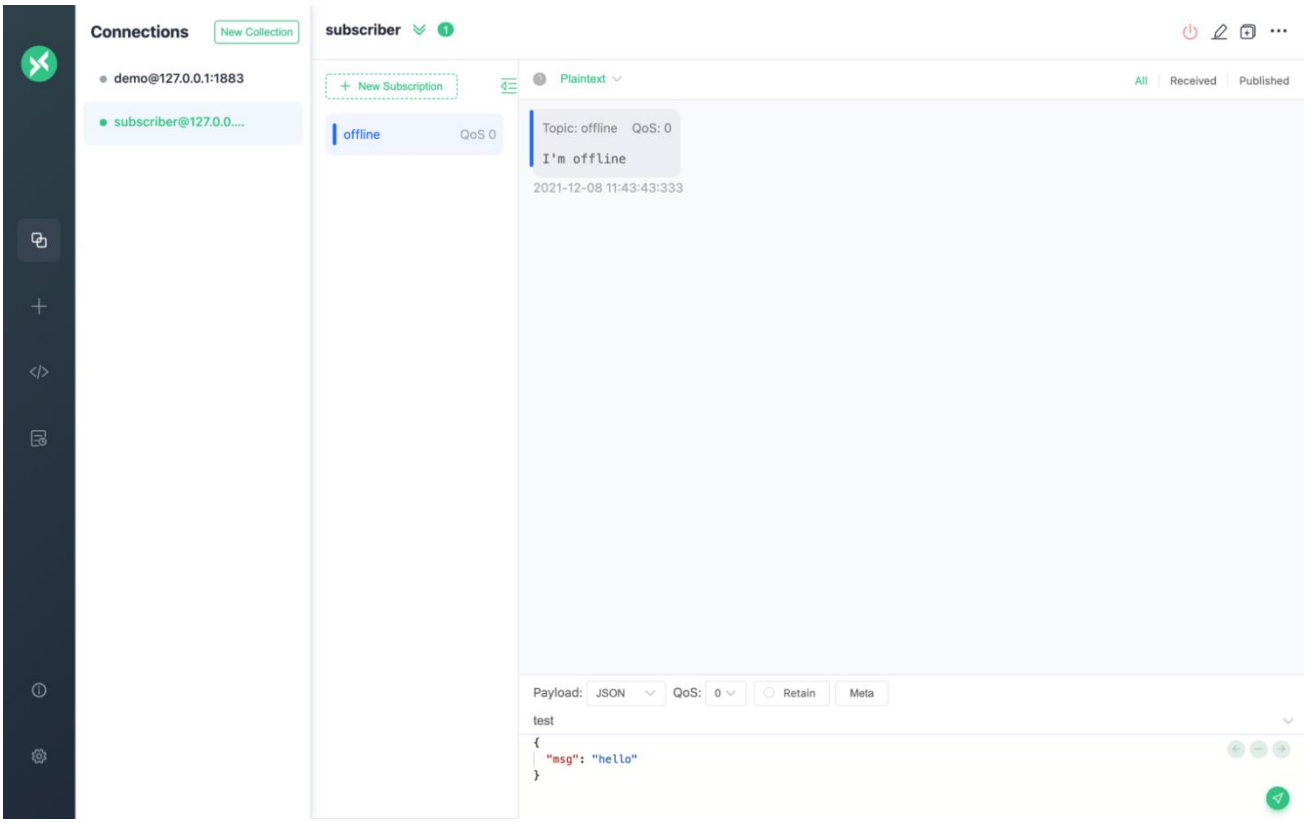



完成以上设置后，我们点击右上角的 Connect 按钮以建立连接。

然后我们再创建一个名为 subscriber 的客户端连接，并订阅 offline 主题。



接下来我们回到 demo 连接中，发布一个 Topic 为 test 的任意内容消息，这时连接会被断开，耐心等待五秒钟，我们将看到 subscriber 连接收到了一条内容为 **I'm offline** 的遗嘱消息。




## 进阶使用场景

这里介绍一下如何将 Retained 消息与 Will 消息结合起来进行使用。

1. 客户端 A 遗嘱消息内容设定为 **offline**，该遗嘱主题与一个普通发送状态的主题设定成同一个 **A/status**。

2. 当客户端 A 连接时，向主题 **A/status** 发送内容为 **online** 的 Retained 消息，其它客户端订阅主题 **A/status** 的时候，将获取到 Retained 消息为 **online**。

3. 当客户端 A 异常断开时，系统自动向主题 **A/status** 发送内容为 **offline** 的消息，其它订阅了此主题的客户端会马上收到 **offline** 消息；如果遗嘱消息设置了 Will Retain，那么此时如果有新的订阅 **A/status** 主题的客户端上线，也将获取到内容为 **offline** 的遗嘱消息。

# Keep Alive

## 为什么需要 Keep Alive

MQTT 协议是承载于 TCP 协议之上的，而 TCP 协议以连接为导向，在连接双方之间，提供稳定、有序的字节流功能。 但是，在部分情况下，TCP 可能出现半连接问题。所谓半连接，是指某一方的连接已经断开或者没有建立，而另外一方的连接却依然维持着。在这种情况下，半连接的一方可能会持续不断地向对端发送数据，而显然这些数据永远到达不了对端。为了避免半连接导致的通信黑洞，MQTT 协议提供了 **Keep Alive** 机制，使客户端和 MQTT 服务器可以判定当前是否存在半连接问题，从而关闭对应连接。

## MQTT Keep Alive 的机制流程与使用

### 启用 Keep Alive

客户端在创建和 MQTT Broker 的连接时，只要将连接请求协议包内的 Keep Alive 可变头部字段设置为非 0 值，就可以在通信双方间启用 **Keep Alive** 机制。 Keep Alive 为 0~65535 的一个整数，代表客户端发送两次 MQTT 协议包之间的最大间隔时间。

而 Broker 在收到客户端的连接请求后，会检查可变头部中的 Keep Alive 字段的值，如果有值，则 Broker 将会启用 **Keep Alive** 机制。

### MQTT 5.0 Server Keep Alive

在 MQTT 5.0 标准中，引入了 Server Keep Alive 的概念，允许 Broker 根据自身的实现等因素，选择接受客户端请求中携带的 Keep Alive 值，或者是覆盖这个值。如果 Broker 选择覆盖这个值，则需要将新值设置在连接确认包(**CONNACK**) 的 Server Keep Alive 字段中，客户端如果在连接确认包中读取到了 Server Keep Alive，则需要使用该值，覆盖自己之前的 Keep Alive 的值。

### Keep Alive 机制流程

#### 客户端流程

在连接建立后，客户端需要确保，自己任意两次 MQTT 协议包的发送间隔不超过 Keep Alive 的值，如果

客户端当前处于空闲状态，没有可发送的包，则可以发送 **PINGREQ** 协议包。

当客户端发送 **PINGREQ** 协议包后，Broker 必须返回一个 **PINGRESP** 协议包，如果客户端在一个可靠的时间内，没有收到服务器的 **PINGRESP** 协议包，则说明当前存在半连接、或者 Broker 已经下线、或者出现了网络故障，这个时候，客户端应当关闭当前连接。

**Broker 流程**

在连接建立后，Broker 如果没有在 Keep Alive 的 1.5 倍时间内，收到来自客户端的任何包，则会认为和客户端之间的连接出现了问题，此时 Broker 便会断开和客户端的连接。

如果 Broker 收到了来自客户端的 **PINGREQ** 协议包，需要回复一个 **PINGRESP** 协议包进行确认。

**客户端接管机制**

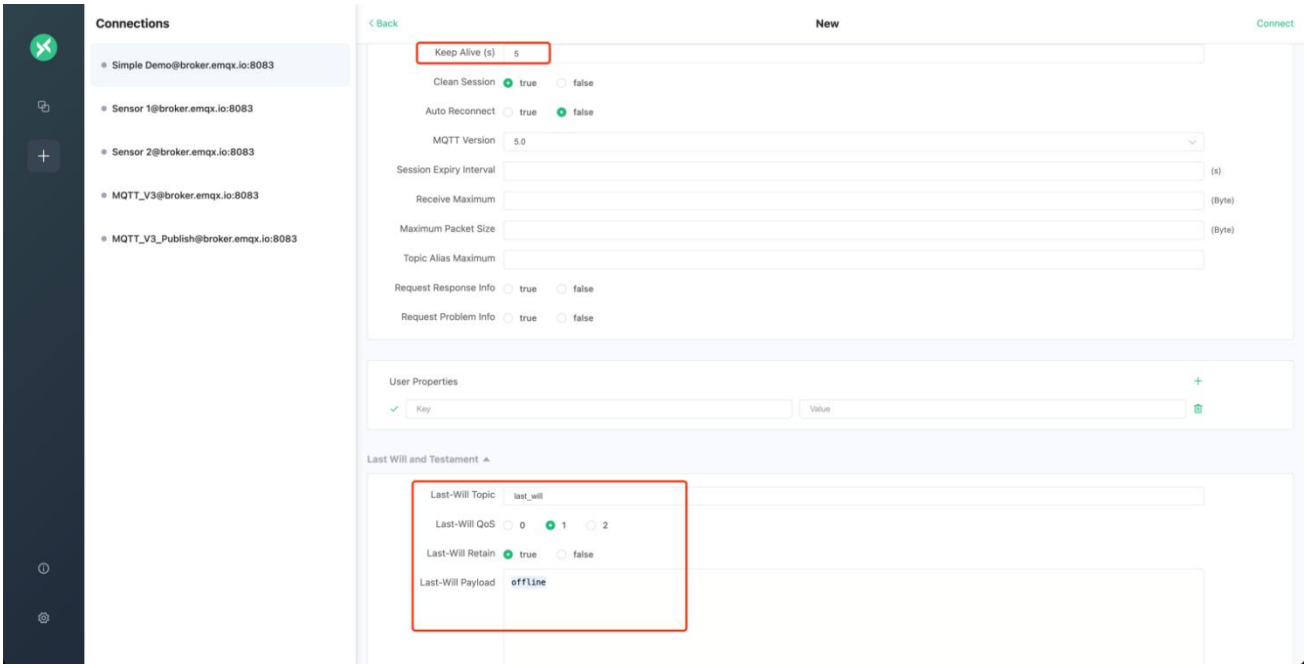
当 Broker 里存在半连接时，如果对应的客户端发起了重连或新的连接，则 Broker 会启动客户端接管机制：关闭旧的半连接，然后与客户端建立新的连接。

这种机制保证了客户端不会因为 Broker 里存在的半连接，导致无法进行重连。

## Keep Alive 与遗嘱消息

Keep Alive 通常还可以与遗嘱消息结合使用，通过遗嘱消息，设备可将自己的意外掉线情况及时通知第三方。

如下图，该客户端连接时设置了 Keep Alive 为 5 秒，并且设置了遗嘱消息。那么当服务器 7.5 秒 (1.5 倍 Keep Alive）内未收到该客户端的任何报文时，即会向 `last_will` 主题发送 Payload 为 `offline` 的遗嘱消息。




## 如何在 EMQX 中使用 Keep Alive

在 EMQX 中，用户可以通过配置来自定义 **Keep Alive** 机制的行为，主要配置字段有:

1. zone.${zoneName}.server_keepalive
2. server_keepalive 类型 默认值 整型 无

如果没有设置这个值，则 EMQX 会按照客户端创建连接时的 Keep Alive 的值，来控制 **Keep Alive** 的行为。

如果设置了这个值，则 Broker 会对该 zone 下面所有的连接，强制启用 **Keep Alive** 机制，并且会使用这个值，覆盖客户端连接请求中的值。

1. zone.${zoneName}.keepalive_backoff
2. keepalive_backoff 类型 默认值 浮点数 0.75

MQTT 协议中要求 Broker 在 1.5 倍 Keep Alive 时间内，如果没有收到客户端的任何协议包，则认定客户端断开了连接。

而在 EMQX 中，我们引入了退让系数（keepalive backoff），并将这个系数通过配置暴露出来，方便用户更灵活的控制 Broker 端的 **Keep Alive** 行为。

在引入退让系数后，EMQX 通过下面的公式来计算最大超时时间:

1. Keepalive * backoff * 2

backoff 默认值为 0.75，因此在用户不修改该配置的情况下，EMQX 的行为完全符合 MQTT 标准。

更多相关内容请参见 EMQX 配置文档。

**WebSocket 连接时设置 Keep Alive**

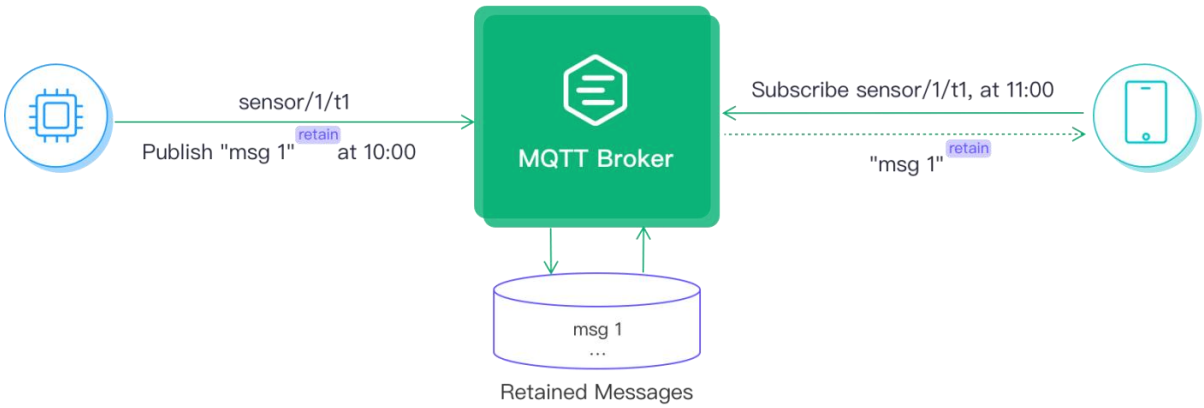
EMQX 支持客户端通过 WebSocket 接入，当客户端使用 WebSocket 发起连接时，只需要在连接参数中设置上 keepalive 的值即可， 具体见使用 WebSocket 连接 MQTT 服务器。

# 保留消息

## 什么是 MQTT 保留消息?

发布者发布消息时，如果 Retained 标记被设置为 true，则该消息即是 MQTT 中的保留消息（Retained Message）。MQTT 服务器会为每个主题存储最新一条保留消息，以方便消息发布后才上线的客户端在订阅主题时仍可以接收到该消息。

如下图，当客户端订阅主题时，如果服务端存在该主题匹配的保留消息，则该保留消息将被立即发送给该客户端。




## 何时使用 MQTT 保留消息?

发布订阅模式虽然能让消息的发布者与订阅者充分解耦，但也存在一个缺点，即订阅者无法主动向发布者请求消息。订阅者何时收到消息完全依赖于发布者何时发布消息，这在某些场景中就产生了不便。

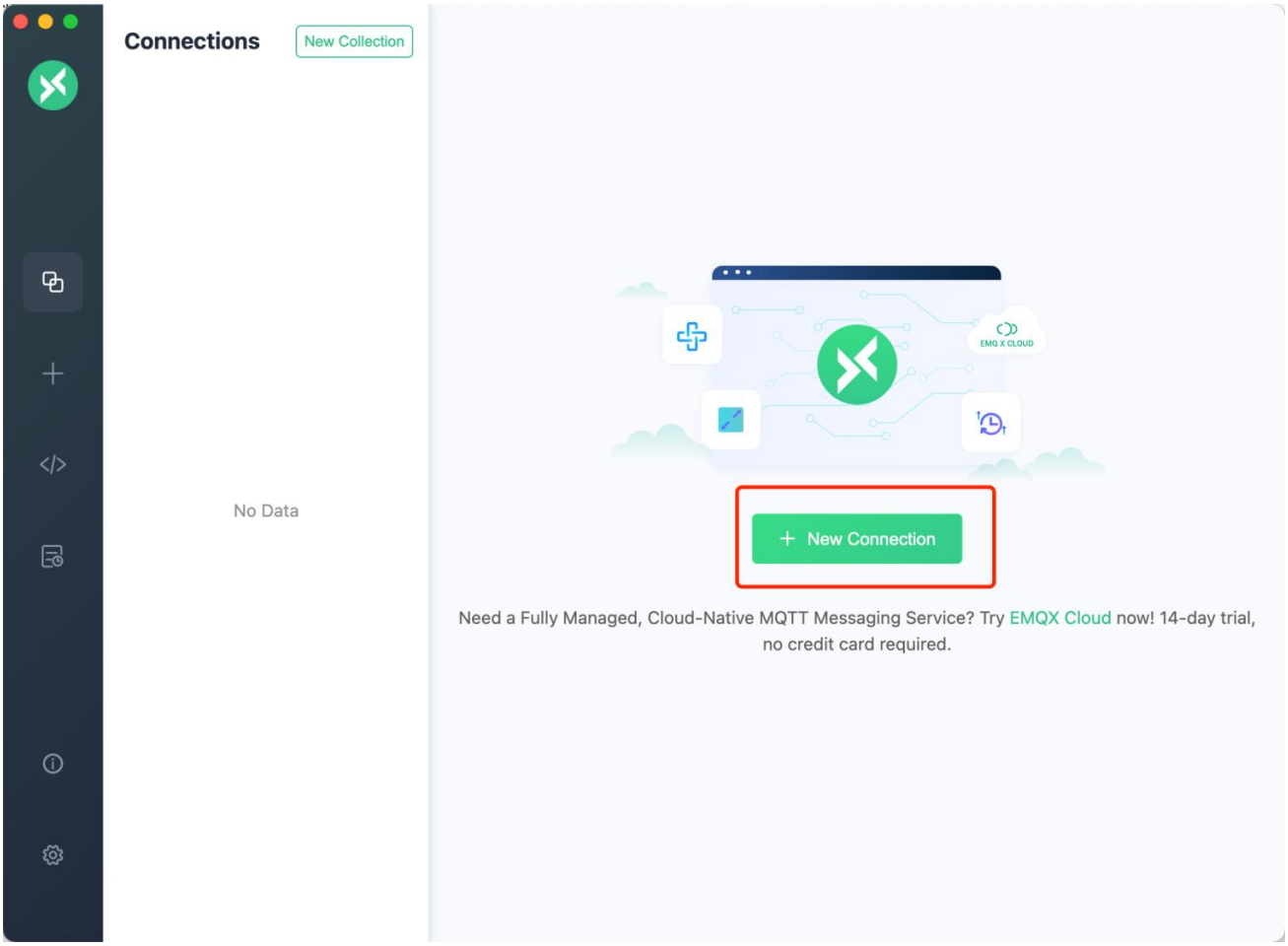
借助保留消息，新的订阅者能够立即获取最近的状态，而不需要等待无法预期的时间，例如：

- 智能家居设备的状态只有在变更时才会上报，但是控制端需要在上线后就能获取到设备的状态；
- 传感器上报数据的间隔太长，但是订阅者需要在订阅后立即获取到最新的数据；
- 传感器的版本号、序列号等不会经常变更的属性，可在上线后发布一条保留消息告知后续的所有订阅者。

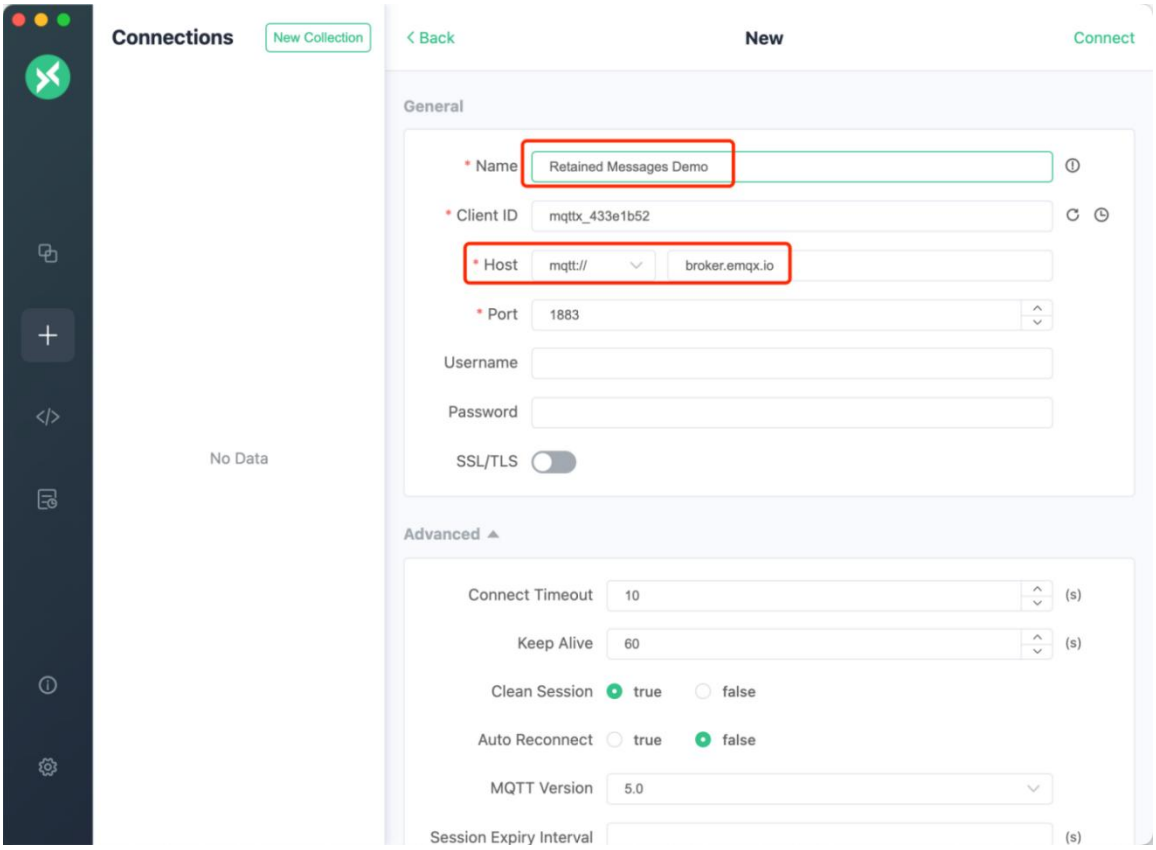
# MQTT 保留消息的使用

若要使用 MQTT 保留消息，只需在消息发布时将 Retained 状态设置为 true 即可。接下来我们以开源的跨平台 MQTT 5.0 桌面客户端工具 – MQTTX 为例，演示如何使用 MQTT 保留消息。

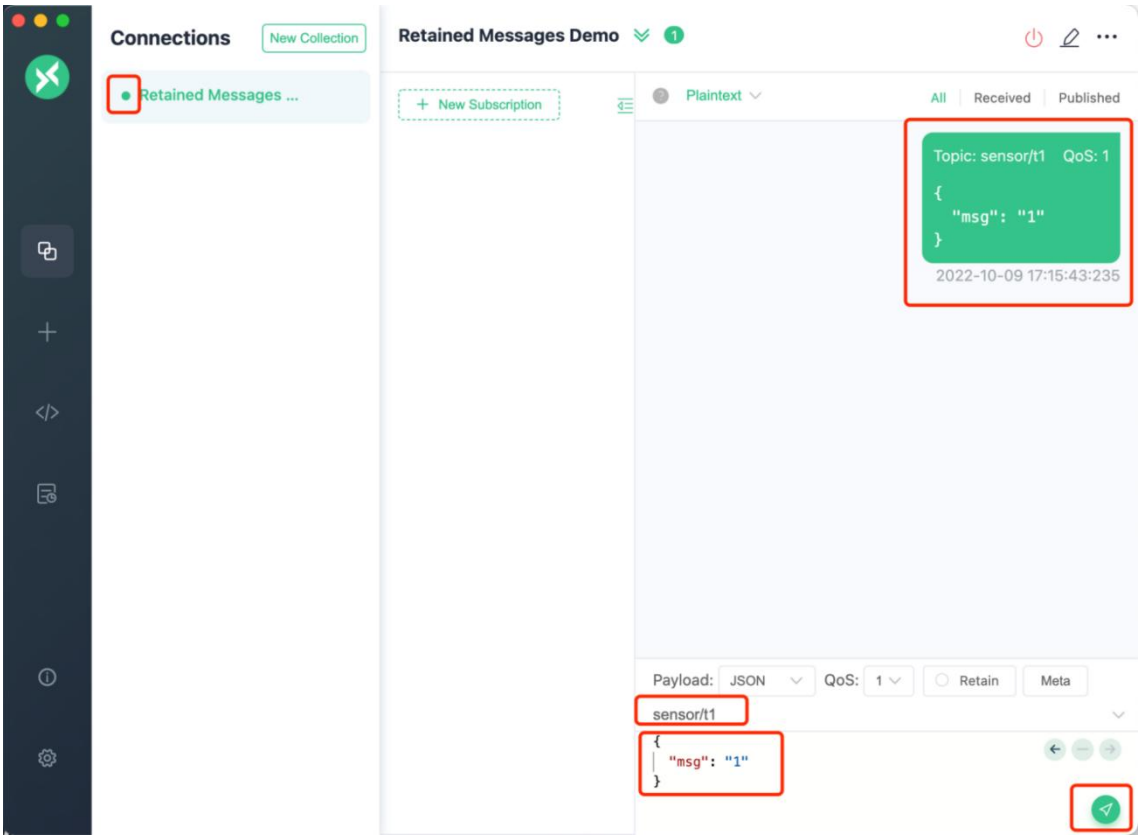
打开 MQTTX 后如下所示，需点击 **New Connection** 按钮创建一个 MQTT 连接。



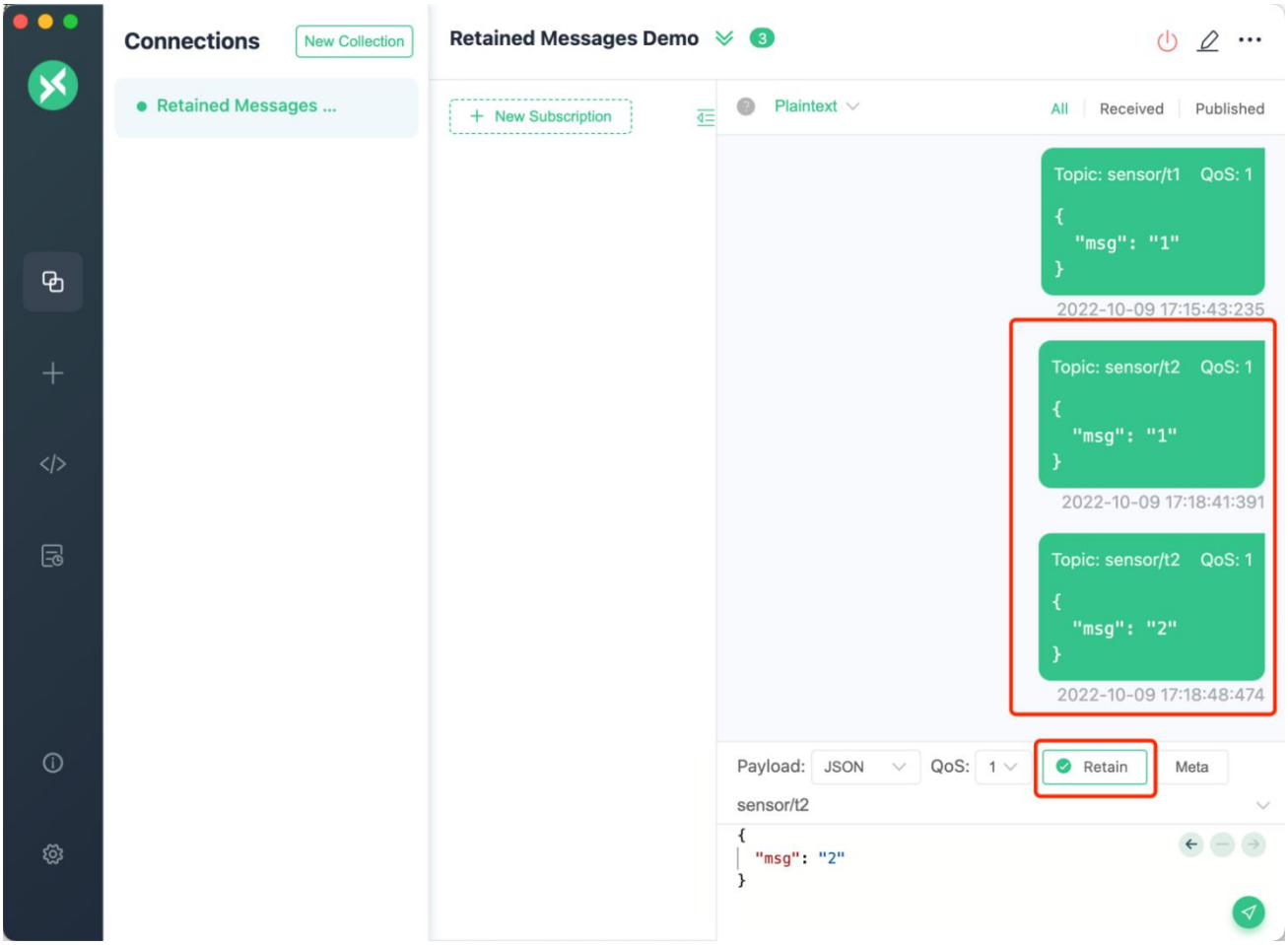

创建页面如下，我们只需填写一个连接名称（Name），其他参数保持默认。Host 将默认为 EMQX Cloud 提供的公共 MQTT 服务器。连接参数填写完成后，点击右上角的 **Connect** 按钮创建 MQTT 连接。



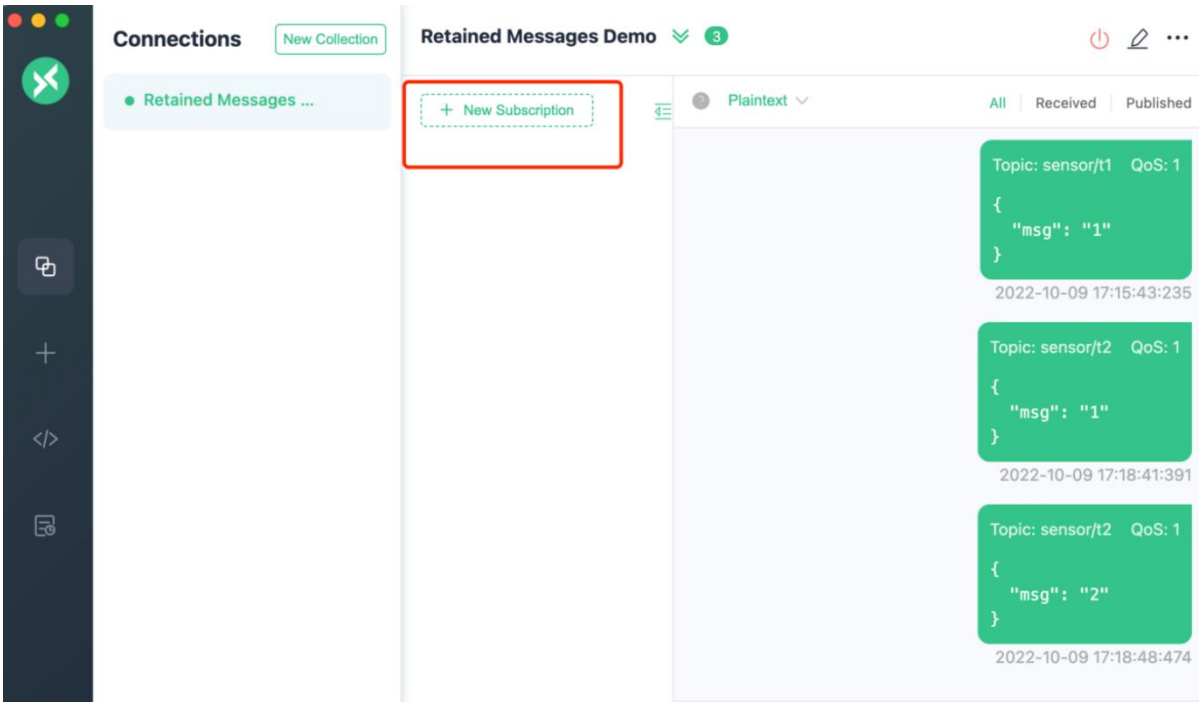

连接成功后将会看到连接名称旁边的状态为绿色。然后我们在右下角消息输入框向主题 **sensor/t1** 发送一条普通的消息。



接下来我们选中右下角的 Retain 标记，并向主题 **sensor/t2** 发送两条保留消息。

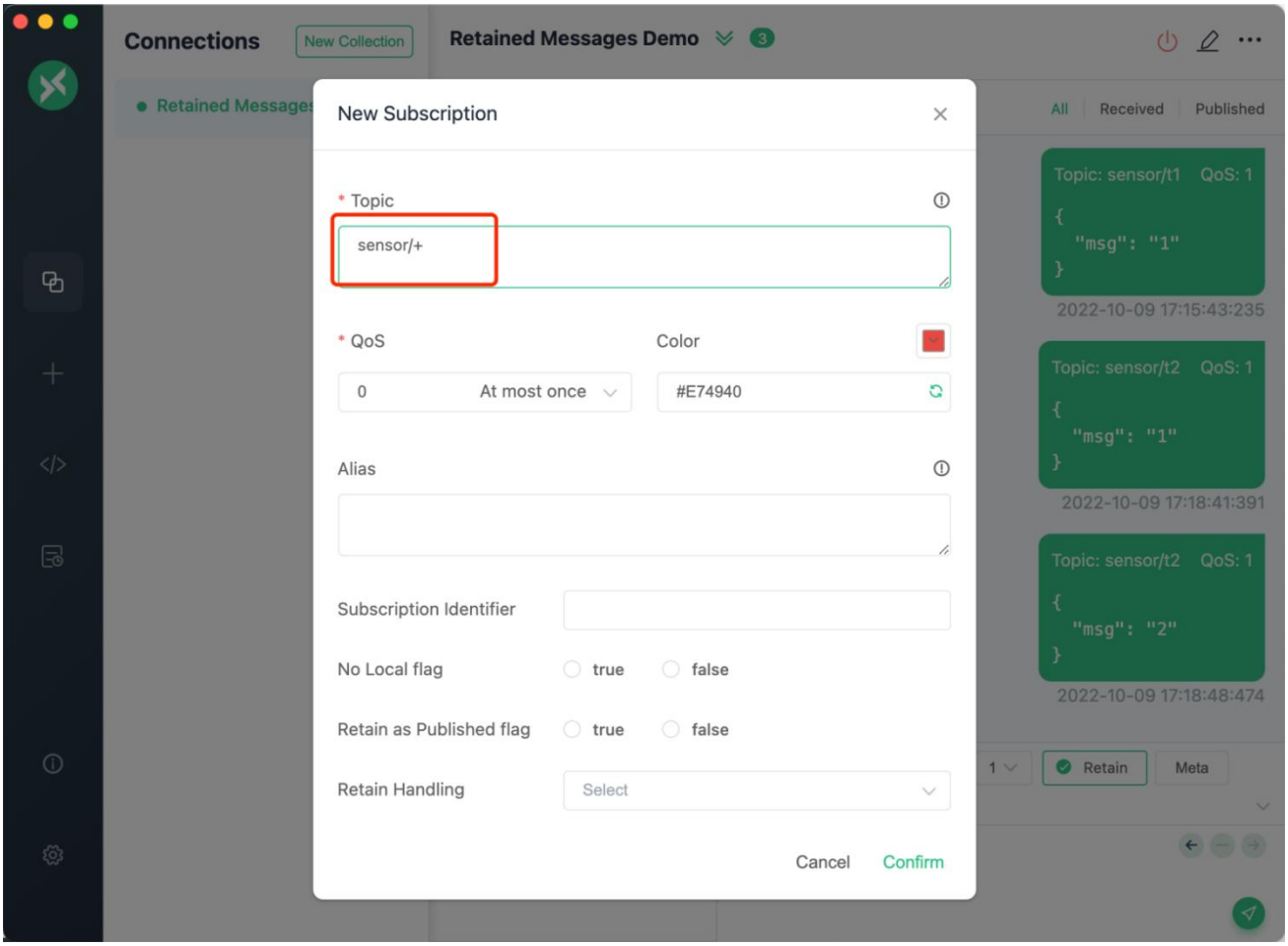



然后点击页面中间的 **New Subscription** 按钮创建订阅。

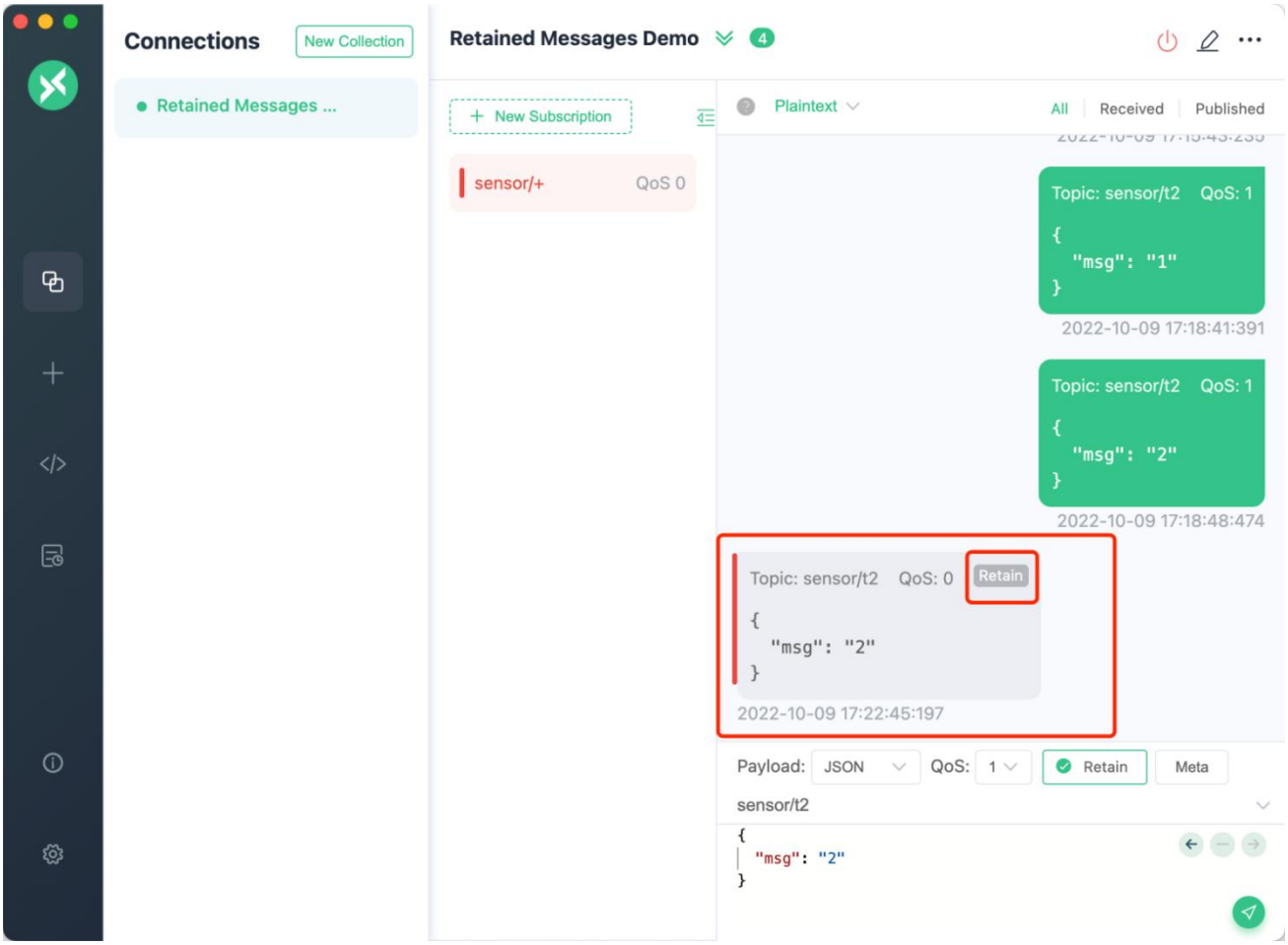


如下，我们订阅通配符主题 **sensor/+**，该通配符主题将会匹配主题 **sensor/t1** 及 **sensor/t2**。

> 关于通配符主题的更多细节，请查看博客通过案例理解 MQTT 主题与通配符。




最后，我们将会看到该订阅能成功收到第二条保留消息，**sensor/t1** 的普通消息及 **sensor/t2** 的第一条保留消息都未收到。可见 MQTT 服务器只会为每个主题存储最新一条保留消息。

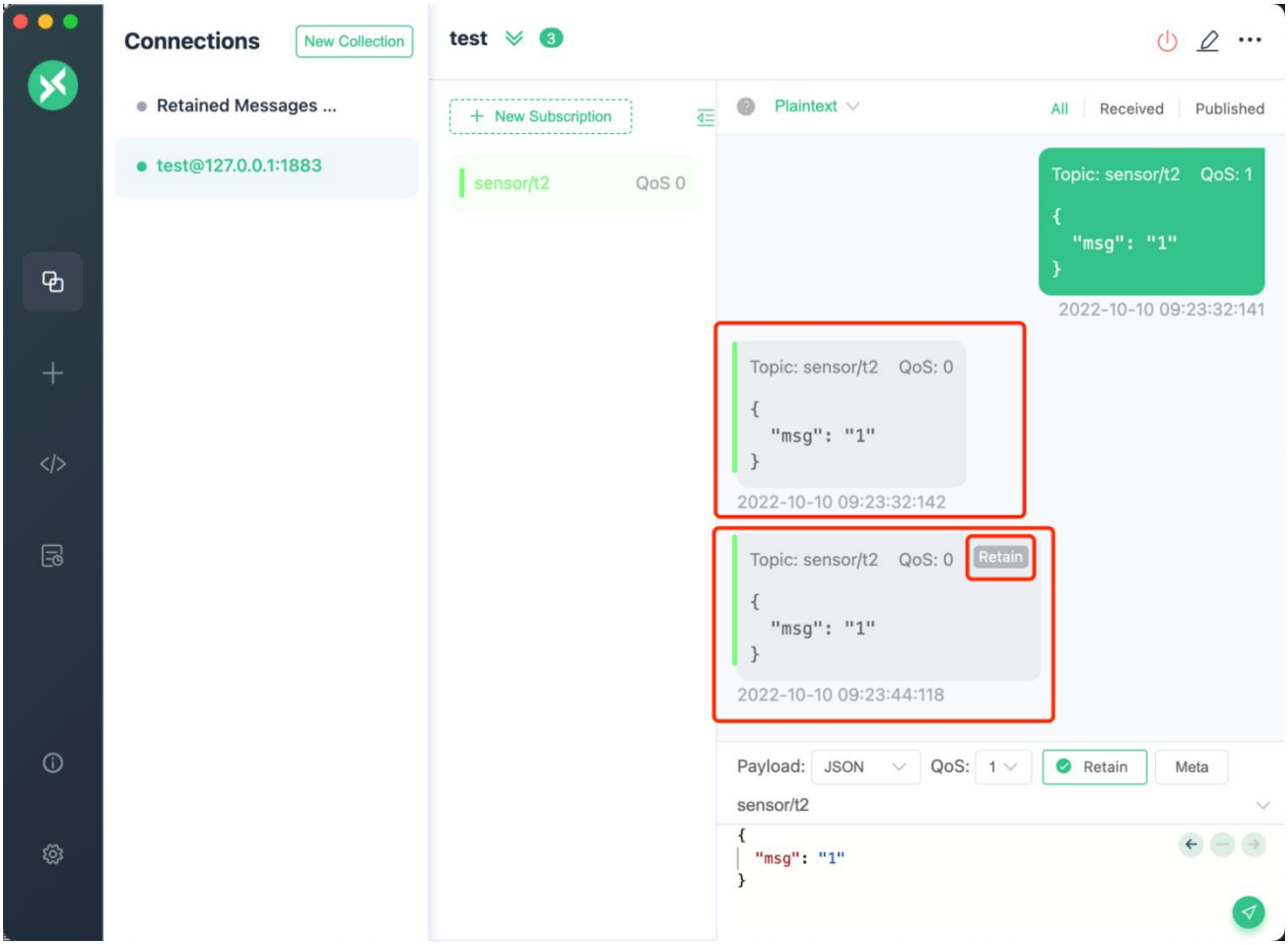



# 关于 MQTT 保留消息的 Q&A

## 如何判断一条消息是否是保留消息?

当客户端订阅了有保留消息的主题后，即会收到该主题的保留消息，可通过消息中的保留标志位判断是否是保留消息。需要注意的是，在保留消息发布前订阅主题，将不会收到保留消息。**需要待保留消息发布后，重新订阅该主题，才会收到保留消息**。

如下图，我们先订阅主题 **sensor/t2**，然后向该主题发布一条保留消息，该订阅会立即收到一条消息，但是该消息并不是保留消息。当我们删除该订阅，再次重新订阅 **sensor/t2** 主题时，立即收到了刚刚发布的保留消息。




## 保留消息将保存多久？如何删除？

服务器只会为每个主题保存最新一条保留消息，保留消息的保存时间与服务器的设置有关。若服务器设置保留消息存储在内存，则 MQTT 服务器重启后消息即会丢失；若存储在磁盘，则服务器重启后保留消息仍然存在。

保留消息虽然存储在服务端中，但它并不属于会话的一部分。也就是说，即便发布这个保留消息的会话已结束，保留消息也不会被删除。删除保留消息有以下几种方式：

- 客户端往某个主题发送一个 Payload 为空的保留消息，服务端就会删除这个主题下的保留消息；
- 在 MQTT 服务器上删除，比如 EMQX MQTT 服务器提供了在 Dashboard 上删除保留消息的功能；
- MQTT 5.0 新增了消息过期间隔属性，发布时可使用该属性设置消息的过期时间，不管消息是否为保留消息，都将会在过期时间后自动被删除。

# EMQX 中的 MQTT 保留消息

EMQX 是一款全球下载量超千万的大规模分布式物联网 MQTT 服务器，自 2013 年在 GitHub 发布开源版本以来，获得了来自 50 多个国家和地区的 20000 余家企业用户的广泛认可，累计连接物联网关键设备超过 1 亿台。

EMQX 5.0 版本通过一个 23 节点的集群达成了 1 亿 MQTT 连接 + 每秒 100 万消息吞吐，这使得 EMQX 5.0 成为目前为止全球最具扩展性的 MQTT 服务器。

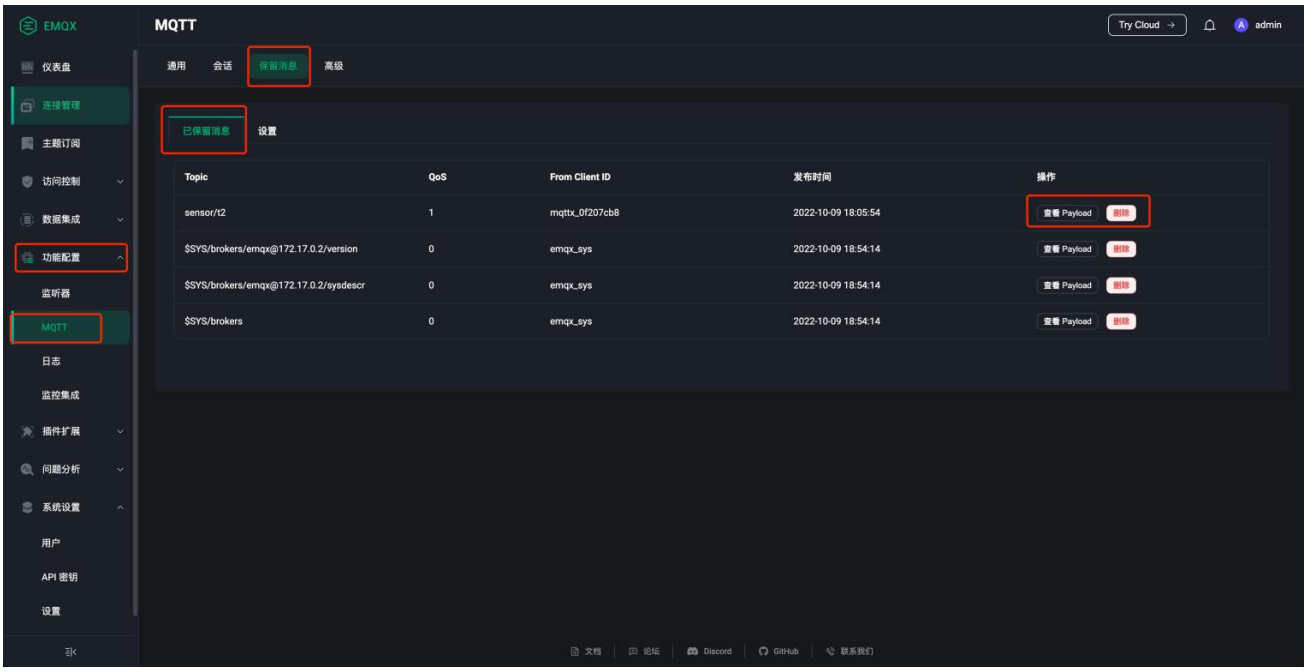
EMQX 5.0 支持在内置的 Dashboard 中查看、设置保留消息。感兴趣的读者可通过如下 Docker 命令安装 EMQX 5.0 开源版进行体验。

```
1. docker run -d --name emqx -p 1883:1883 -p 8083:8083 -p 8084:8084 -p 8883:8883 -p 18083:18083 emqx/emqx:latest
```

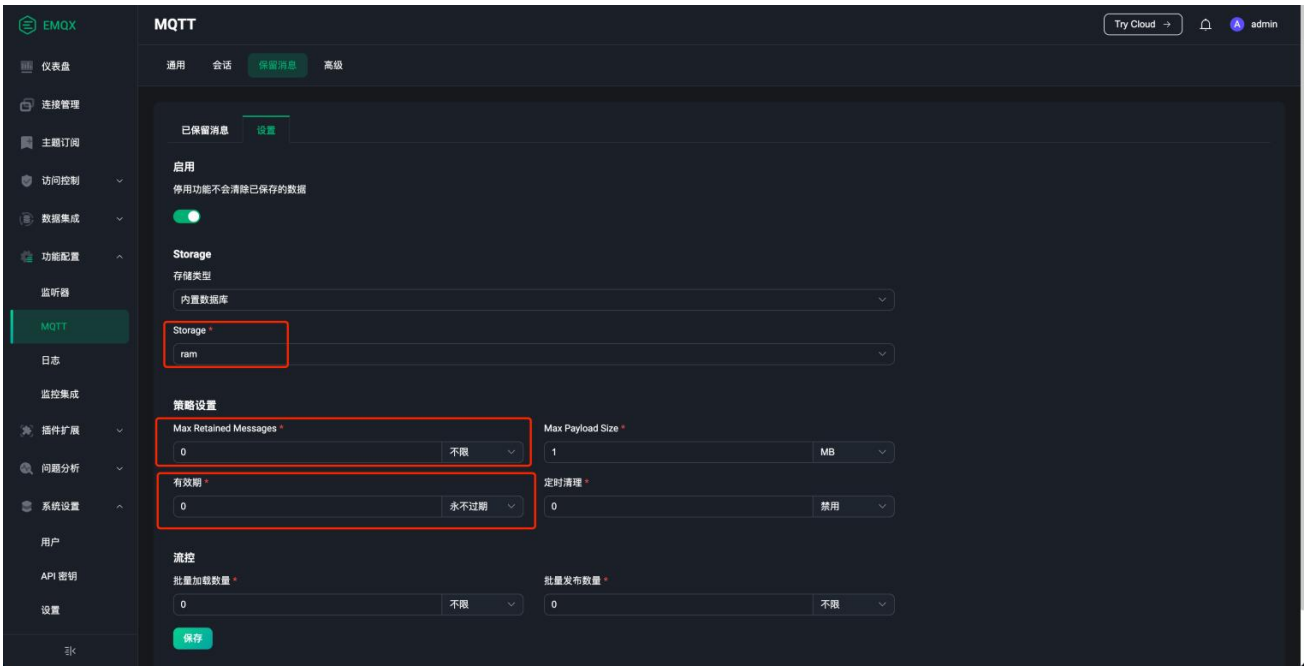
EMQX 安装成功后，使用浏览器访问 **http://127.0.0.1:18083/** 即可体验 EMQX 5.0 全新 Dashboard。

> 默认用户名为 admin，密码为 public

登录成功后，可在左侧菜单 **System -> Settings** 中修改显示语言为中文。如下图，可点击**功能配置->MQTT** 菜单查看已保留的消息列表，同时也可以查看保留消息的 Payload 或者删除某条保留消息。



点击保留消息下的设置菜单，可看到 EMQX 支持在 Dashboard 中设置保留消息的存储类型（内存或磁盘）、最大保留消息数、保留消息有效期等参数，点击保存后所有更改将会立即生效。



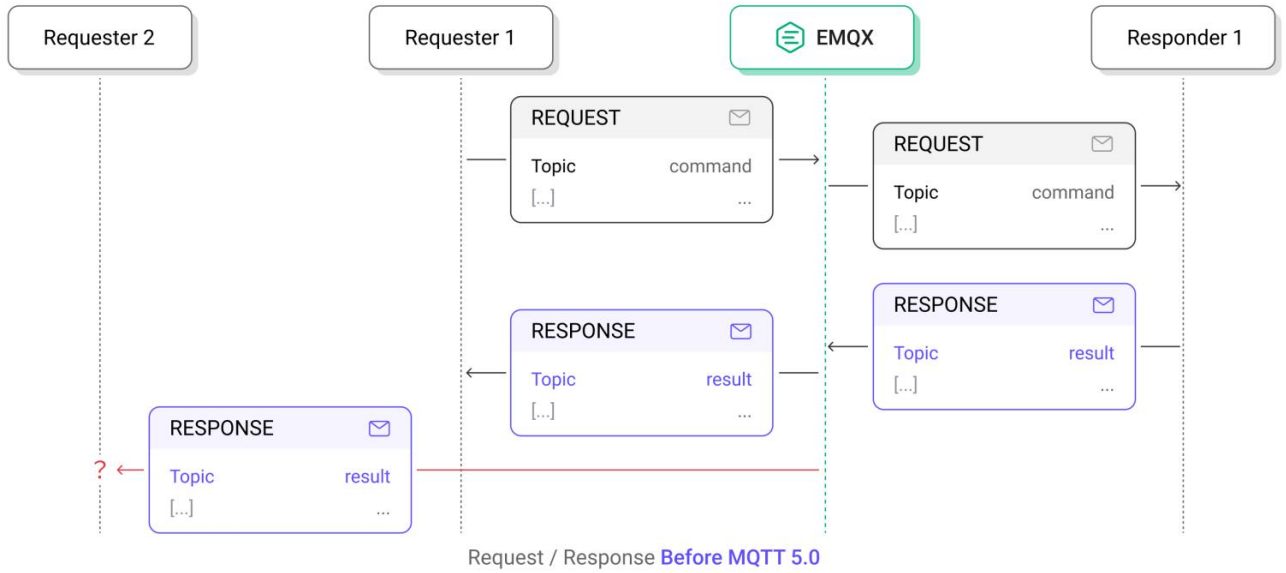
# 请求/响应

## MQTT 5.0 之前的“请求/响应”

MQTT 的发布订阅机制，使消息的发送者和接收者完全解耦，所以消息可以被异步地传递。但这也带来了一个问题，即便是使用了 QoS 1 或 2 的消息，发布端也只能确保消息到达服务端，而无法知晓订阅端最终是否收到了消息。在执行一些请求或命令时，发布端还会想要知道对端的执行结果。最直接的做法是，让订阅端为请求返回响应。

在 MQTT 中，这不难实现，只需要通信双方提前协商好请求主题和响应主题，然后订阅端在收到请求后向响应主题返回响应即可。这也是 MQTT 5.0 之前的客户端普遍采用的做法。

但在这个方案中，响应主题必须提前确定，无法灵活地变更。当存在多个不同的请求方时，由于只能订阅相同的响应主题，所以所有请求方都会收到响应，并且它们无法分辨响应是否属于自己：




存在多个请求方时，容易出现响应混乱的问题

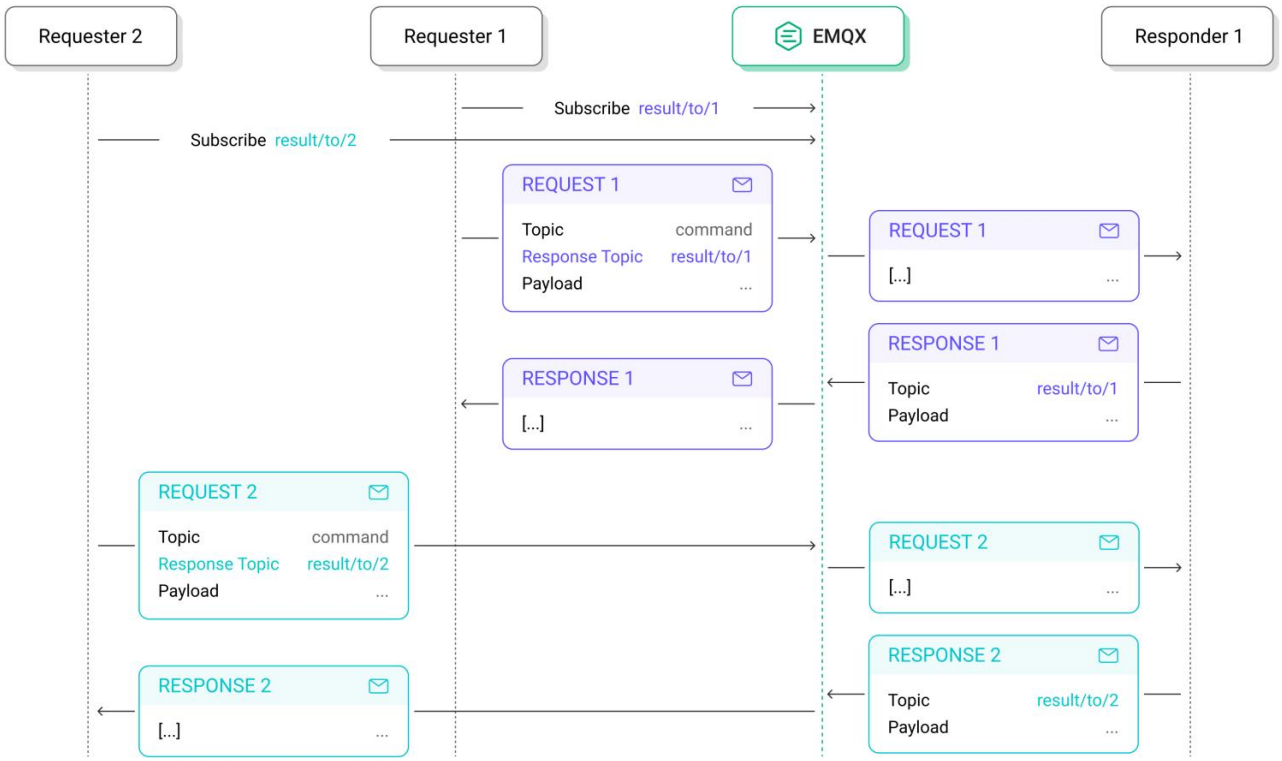
虽然有不少办法可以避免这一问题，但这也导致各个厂商可能有着完全不同的实现，这使用户在集成不同厂商设备时的难度和工作量极大增加。

为了解决以上问题，MQTT 5.0 引入了响应主题（Response Topic）、关联数据（Correlation Data）和响应信息（Response Information）这些属性使 MQTT 中的“请求/响应”机制标准化、规范化。

# MQTT 5.0 的“请求/响应”如何运作?

## 响应主题

在 MQTT 5.0 中，请求方可以在请求消息中指定一个自己期望的响应主题（Response Topic）。响应方根据请求内容采取适当的操作后，向请求中携带的响应主题发布响应消息。如果请求方订阅了该响应主题，那么就会收到响应。




请求方可以将自己的 Client ID 作为响应主题的一部分，这可以有效避免不同的请求方不小心使用了相同的响应主题而造成冲突。

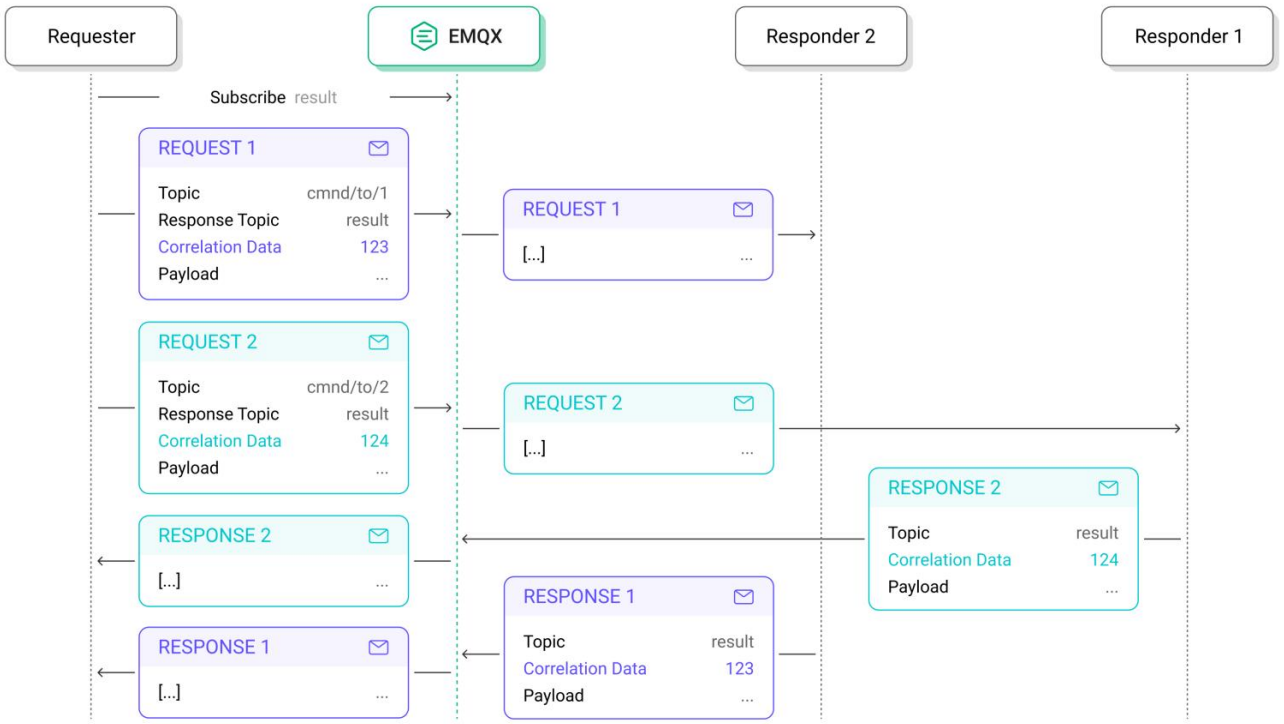
## 关联数据

请求方还可以在请求中携带关联数据（Correlation Data），响应方必须在响应中将关联数据**原封不动**地返回，请求方因此可以识别响应所属的原始请求。

这可以避免在响应方没有按请求顺序返回响应或者由于网络连接断开导致丢失了某个响应（QoS 0）时，请求方不正确地关联响应与原始请求。

另一方面，请求方可能需要与多个响应方交互，譬如我们通过手机控制家中的各种智能设备，关联数据可

以让请求方仅订阅一个响应主题就能管理从多个响应方异步返回的响应。

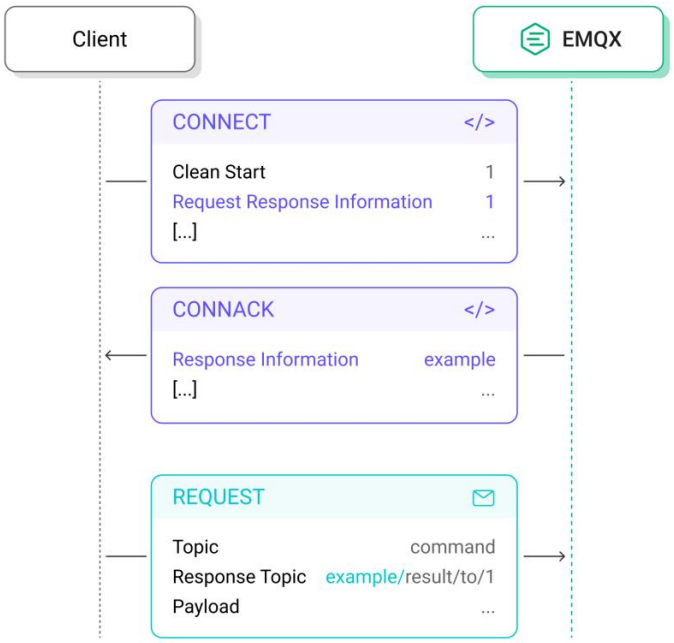



在以上“请求/响应”的过程中，MQTT 服务端不会对响应主题、关联数据进行任何更改，它仅起到转发的作用。

## 响应信息

出于安全考虑，MQTT 服务端通常会限制客户端可以发布和订阅的主题。请求方可以指定一个随机的响应主题，但无法保证自己有订阅该主题的权限，也无法保证响应方有向该响应主题发布消息的权限。

所以 MQTT 5.0 还引入了响应信息（Response Information）属性，通过在 CONNECT 报文中将请求响应信息(Request Response Information）标识符设置为 1，客户端可以请求服务端在 CONNACK 报文中返回响应信息。客户端可以将响应信息的内容作为响应主题的某个特定部分，以便通过服务端的权限检查。




MQTT 并未进一步约定这部分的细节，比如响应信息的内容格式以及客户端如何根据响应信息创建响应主题，所以不同服务端和客户端的实现可能有所不同。例如，服务端可以使用响应信息“FRONT,mytopic”既指示响应主题中特定部分的具体内容，又指示该特定部分在响应主题中的位置，也可以与客户端提前约定如何使用该特定部分，然后使用响应信息“mytopic”仅指示该特定部分的具体内容。

以智能家居场景为例，智能设备不会跨用户使用，我们可以让 MQTT 服务端将设备所属用户的 ID 作为响应信息返回，客户端统一使用该用户 ID 作为响应主题的前缀。MQTT 服务端只需要确保这些客户端在它们会话的生命周期内，都拥有该用户 ID 开头的主题的发布和订阅权限即可。

## MQTT “请求/响应”的使用建议

以下是在 MQTT 中使用“请求/响应”时的一些建议，遵循这些建议将有助于你实施最佳实践：

- MQTT 的 QoS 1 或 2 只能确保消息到达服务端，如果想要确认消息是否到达订阅端，可以借助“请求/响应”机制。
- 在发送请求前订阅响应主题以免错过响应。
- 确保响应方和请求方拥有发布和订阅响应主题的必要权限，响应信息（Response Information）可以帮助我们构建符合权限要求的响应主题。
- 存在多个请求方时，请求方需要使用不同的响应主题以免响应混淆，使用 Client ID 作为主题的一部

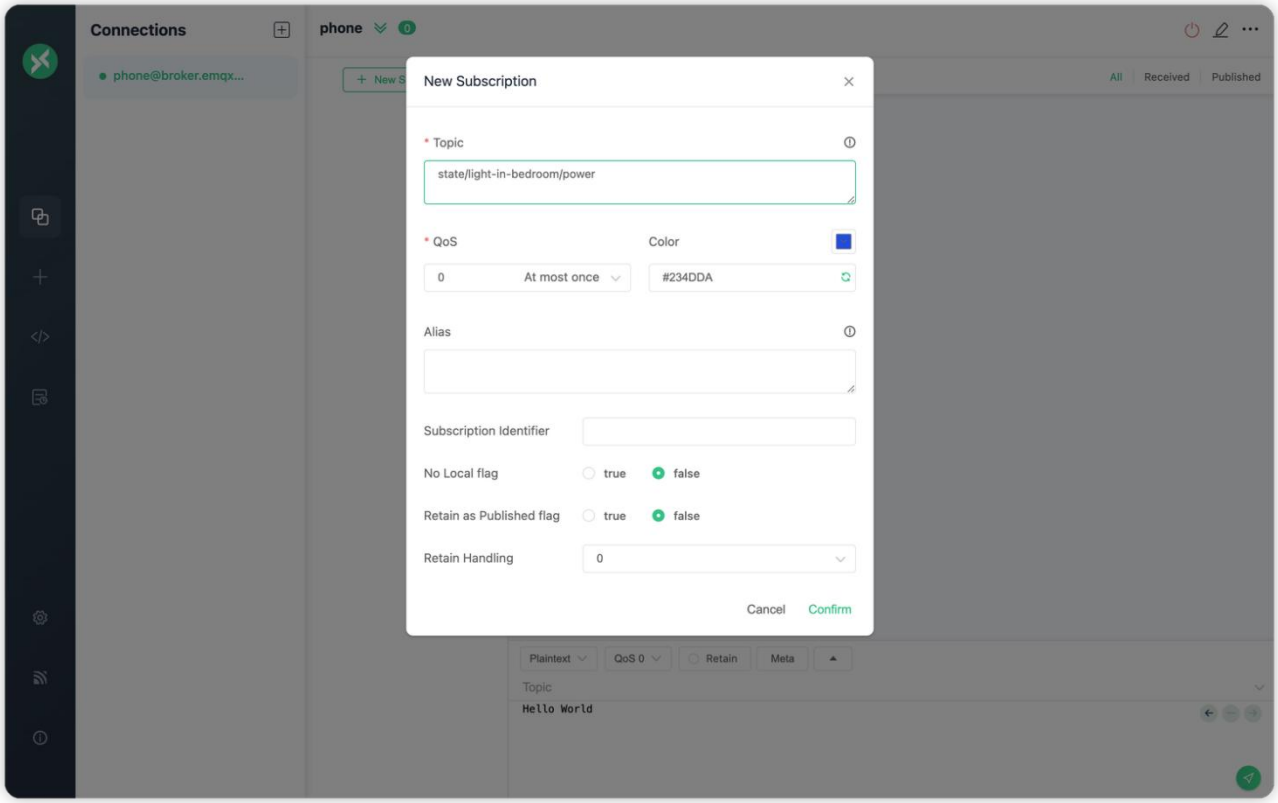
分是常见的做法。

- 存在多个响应方时，请求方最好在请求中设置关联数据（Correlation Data）以免响应混淆。
- 遗嘱消息也可以使用“请求/响应”，只需要在连接时为遗嘱消息设置响应主题即可。这可以帮助客户端知道在自己离线期间遗嘱消息是否被消费，以便做出适当的调整。

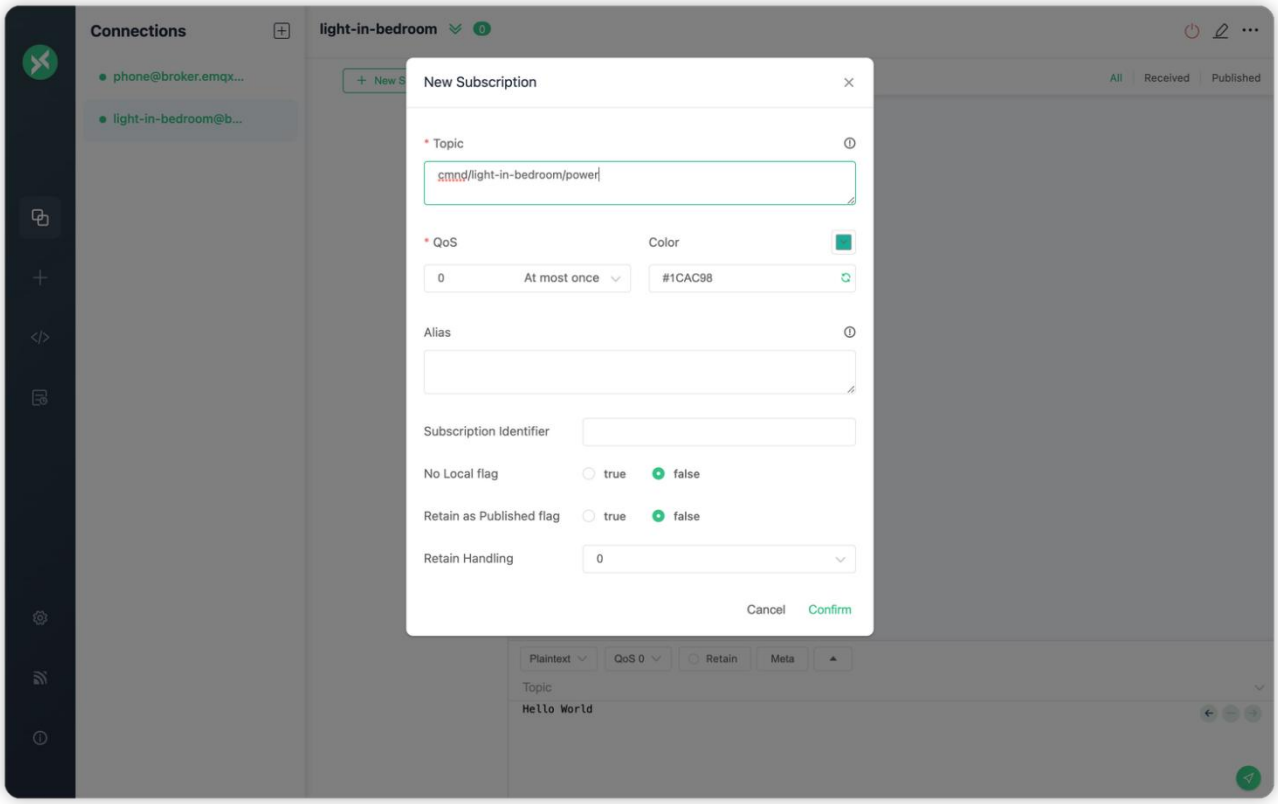
## 示例

接下来我们将使用 MQTTX 模拟使用手机远程控制卧室灯开启并接收响应的场景。

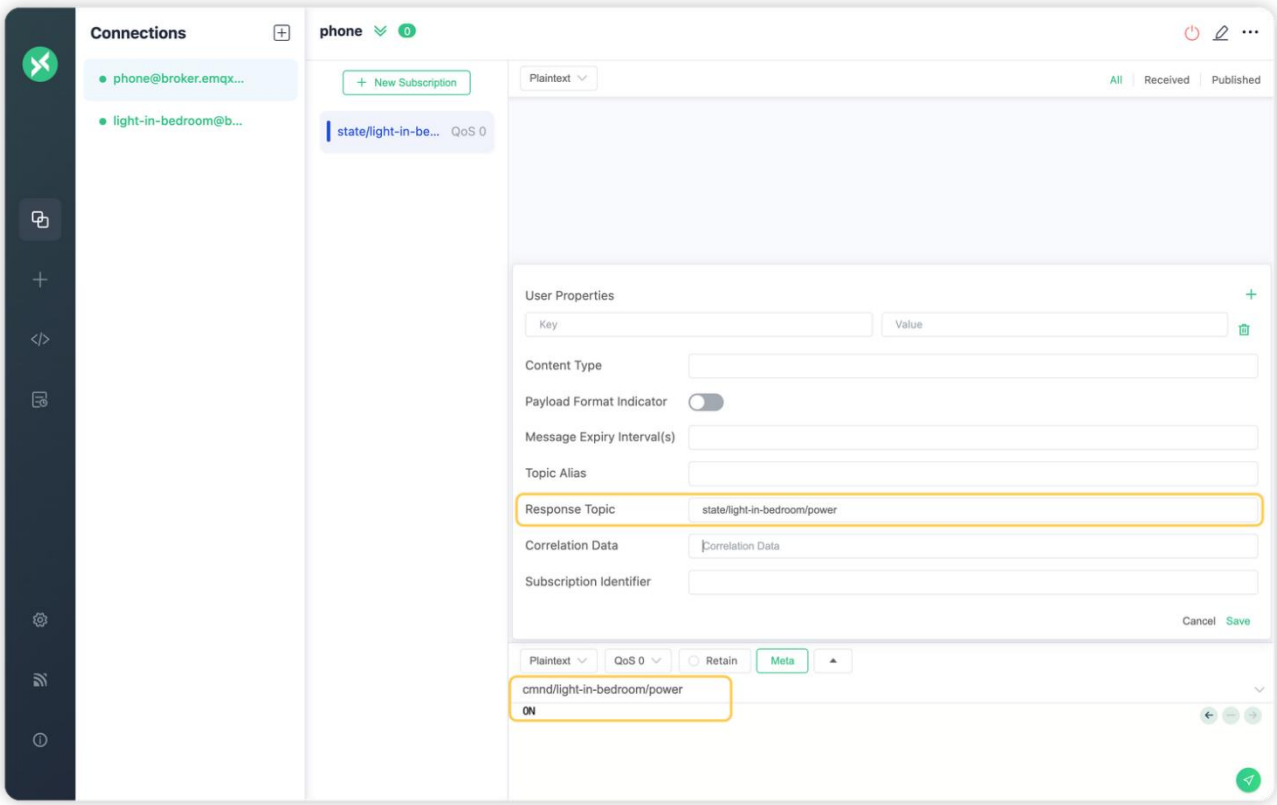
安装并打开 MQTTX，首先向 公共 MQTT 服务器 发起一个客户端连接用于模拟手机，并订阅响应主题 `state/light-in-bedroom/power`：



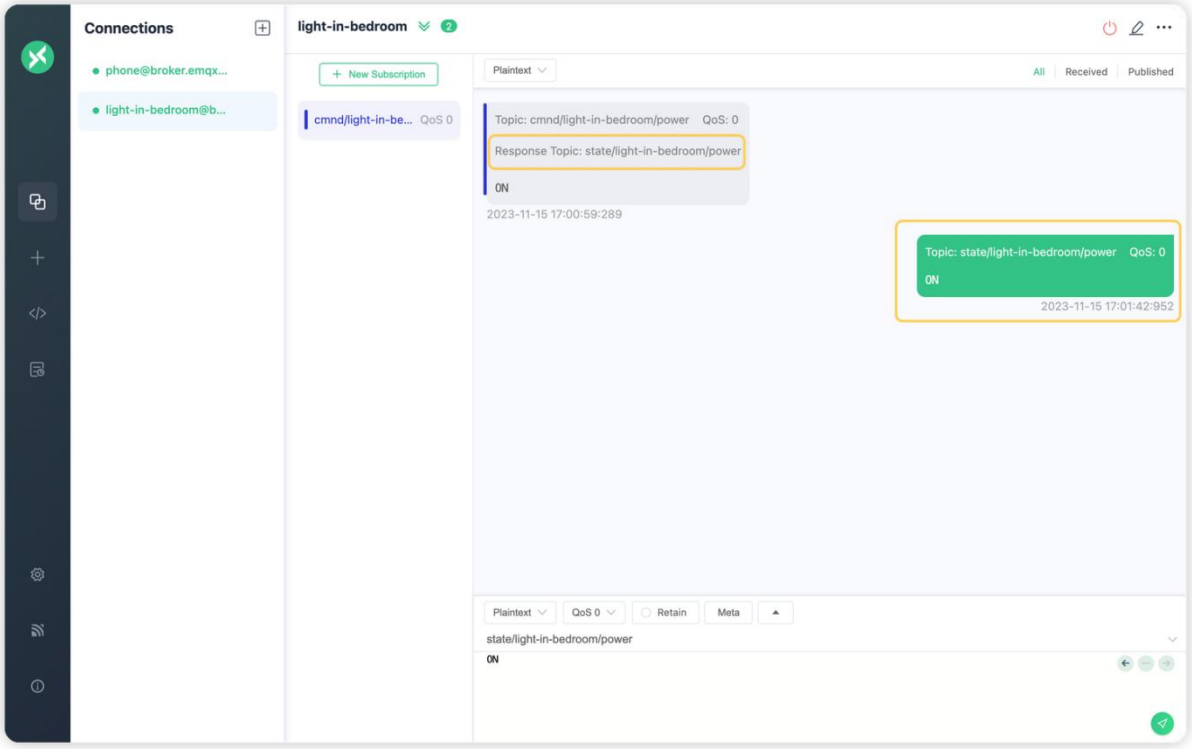

创建一个新的客户端连接用于模拟智能灯，并订阅请求主题 `cmnd/light-in-bedroom/power`：



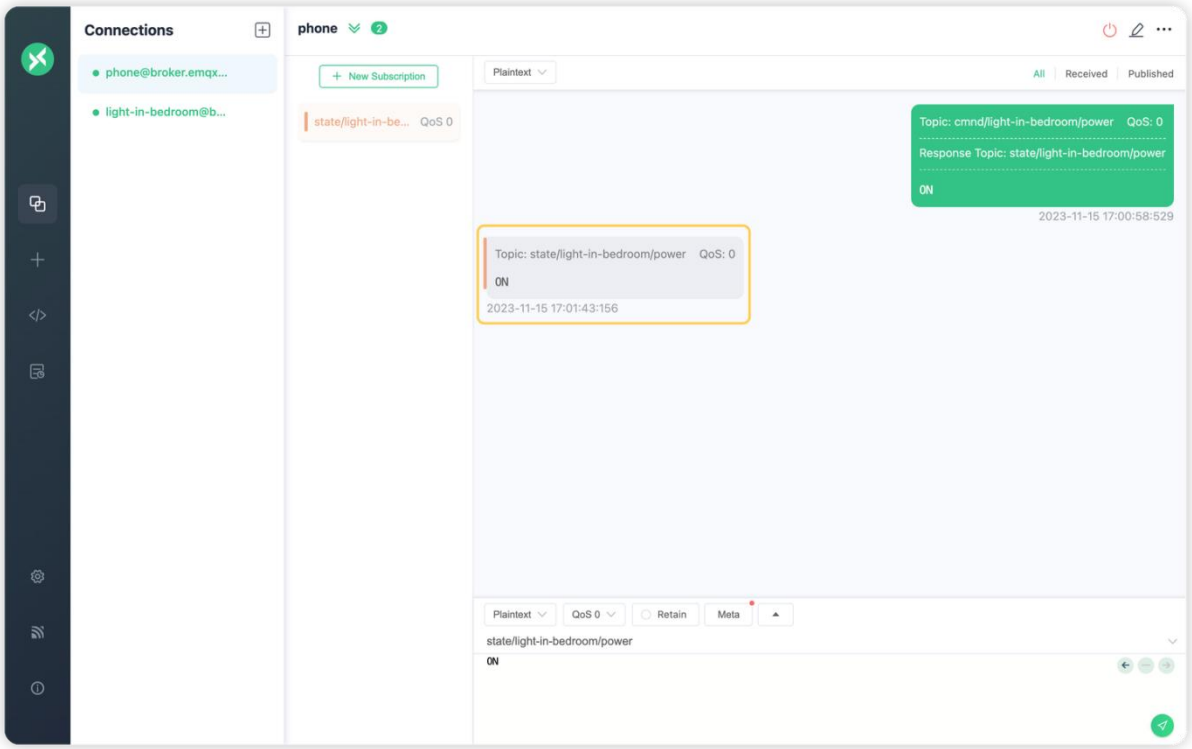

回到请求客户端，向请求主题 **cmnd/light-in-bedroom/power** 发送一条指定了响应主题的开灯指令：



在响应客户端中，我们可以看到收到的消息中携带了响应主题，那么接下来，我们就可以按照指令要求执行开灯操作，然后将灯的最新状态通过该响应主题返回：




最终，请求客户端将收到此响应，并且根据响应消息的内容，我们能够知道现在灯已经成功打开：



这是一个非常简单的示例，你还可以尝试增加发布者或者响应者的数量，体会在这些情况下应该如何设计请求主题和响应主题。

另外，我们在 emqx/MQTT-Features-Example 项目中提供了“请求/响应”的 Python 示例代码，你可以作为参考。

# 用户属性

## 什么是用户属性

用户属性（User Properties）其实是一种自定义属性，允许用户向 MQTT 消息添加自己的元数据，传输额外的自定义信息以扩充更多应用场景。

它由一个用户自定义的 UTF-8 的键/值对数组组成，并在消息属性字段中配置，只要不超过最大的消息大小，可以使用无限数量的用户属性来向 MQTT 消息添加元数据，并在发布者、MQTT 服务器和订阅者之间传递信息。

如果你熟悉 HTTP 协议的话，该功能与 HTTP 的 Header 的概念非常类似。用户属性有效地允许用户扩展 MQTT 协议，并且可以出现在所有消息和响应中。因为用户属性是由用户定义的，它们只对该用户的实现有意义。

## 为什么需要使用用户属性

MQTT 3 的协议扩展性能力较差，用户属性其实就是为了解决这个问题，它支持在消息中传递任何信息，确保了用户可扩展标准协议的功能。

对于选择和配置不同的消息类型，用户属性可以在客户端与 MQTT 服务器之间，或者客户端和客户端之间发送。在连接客户端中配置用户属性时，只能在 MQTT 服务器上接收，无法在客户端中接收。如果在发送消息的时候配置用户属性，则可以在其它客户端中接收。常用的有以下两种用户属性配置。

### 连接客户端的用户属性

当客户端与 MQTT 服务器发起连接时，服务器可以预先定义好一些需要并且可以使用到的元数据信息，即用户属性，当连接成功后，MQTT 服务可以拿到连接发送过来的相关信息进行使用，因此连接客户端的用户属性依赖于 MQTT 服务器。

### 消息发布的用户属性

消息发布时的用户属性可能是较为常用的，因为它们可以在客户端与客户端之间进行元数据信息传递。比

如可以在发布时添加一些常见的信息：消息编号，时间戳，文件，客户端信息和路由信息等属性。

除上述较为常用的用户属性设置外，还可以在订阅 Topic 时，取消订阅时，断开连接时配置用户属性。

# 用户属性的使用

## 文件传输

MQTT 5 的用户属性，可扩展为使用其进行文件传输，而不是像之前的 MQTT 3 中将数据放到消息体的 Payload 中，用户属性使用键值对的方式。这也意味着文件可以保持为二进制，因为文件的元数据在用户属性中。例如：

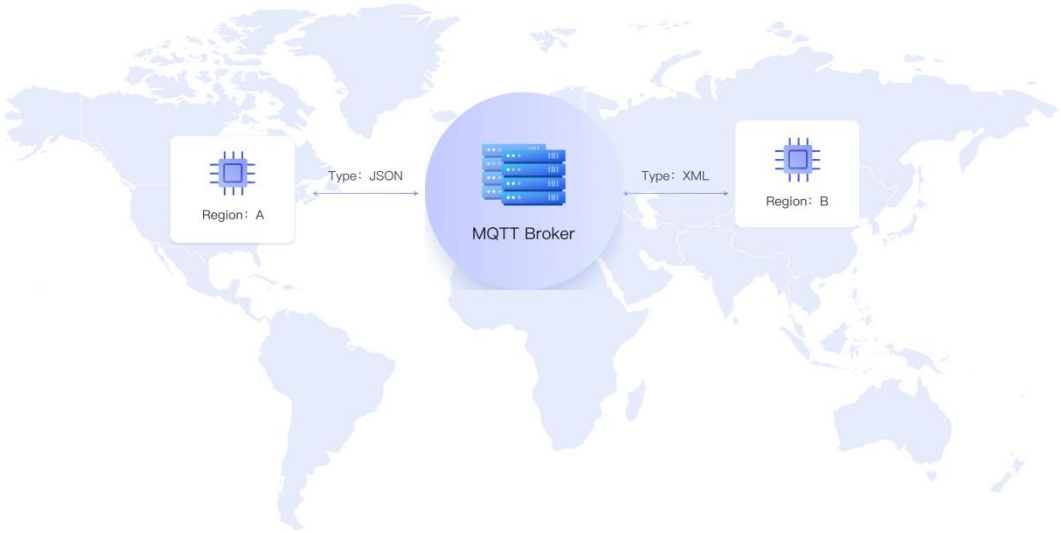
```
{
"filename": "test.txt",
"content": "xxxx"
}
```

## 资源解析

当客户端连接到 MQTT 服务器后，不同的客户端、供应商平台或系统存在着不同的方式传递消息数据，消息数据的格式可能都存在着一些结构差异。还有一些客户端是分布在不同的地域下。比如：地域 A 的设备发送的消息格式是 JSON 的，地域 B 的设备发送的是 XML 的，此时服务器接收到消息后可能需要一一进行判断和对比，找到合适的解析器来进行数据解析。

此时为了提高效率和减少计算负载，我们可以利用用户属性功能来添加数据格式信息和地域信息，当服务器接收到消息后，可以使用用户属性中提供的元数据来进行数据解析操作。并且当区域 A 的客户端订阅接收到来自区域 B 的客户端消息时，也能快速的清楚特定的消息的来自于哪个区域等，从而使的消息具有了可追溯性。

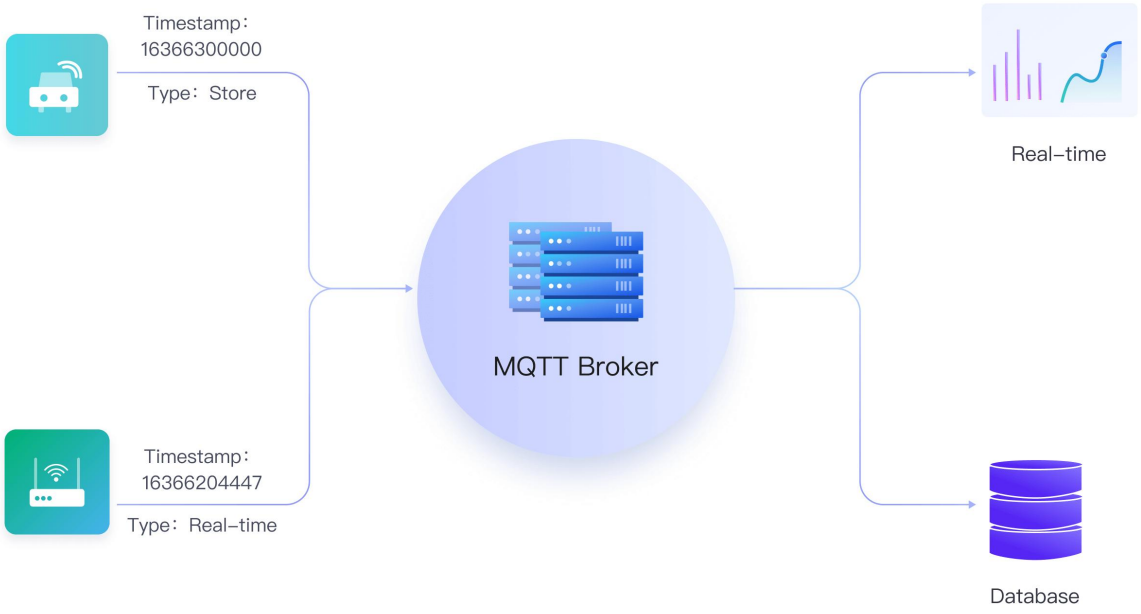
```
{
"region": "A",
"type": "JSON"
}
```




## 消息路由

我们还可以使用用户属性来做应用层级别的路由。如上所述，存在着不同的系统和平台，每个区域存在着不同的设备，多个系统可能收到同一个设备的消息，有些系统需要将数据进行实时的展示，另一个系统可能将这些数据进行时序存储。因此 MQTT 服务器可以通过上报消息中配置的用户属性来确定将消息分发到存储消息的系统还是展示数据的系统。

```
1. {
2. "type": "real-time",
3. "timestamp": 1636620444
4. }
```



# 在客户端中配置用户属性

我们以 JavaScript 环境为例，使用 MQTT.js 客户端来进行编程。

> 注意： 在连接客户端时需指定 MQTT 的版本 **protocolVersion** 为 5。

## 连接

我们在连接时的 options 中设置 properties 的 User Properties 属性，添加 type 和 region 属性。连接成功后，MQTT 服务器将收到这个用户自定义的信息。

```
// connect options
const OPTIONS = {
    clientId: 'mqtt_test',
    clean: true,
    connectTimeout: 4000,
    username: 'emqx',
    password: 'public',
    reconnectPeriod: 1000,
    protocolVersion: 5,
    properties: {
        userProperties: {
            region: 'A',
            type: 'JSON',
        },
    },
}
const client = mqtt.connect('mqtt://broker.emqx.io', OPTIONS)
```

## 发布消息

连接成功后发布消息，发布消息的配置中设置用户属性，并且监听消息接收。在 publish 函数中，我们配置 user properties 属性，并在接收消息的函数中打印 packet。

```
 client.publish(topic, 'nodejs mqtt test', {
    qos: 0,
```

```
    retain: false,
    properties: {
      userProperties: {
      region: 'A',
      type: 'JSON',
      },
    },
}, (error) => {
    if (error) {
      console.error(error)
    }
})
client.on('message', (topic, payload, packet) => {
    console.log('packet:', packet)
    console.log('Received Message:', topic, payload.toString())
})
```

此时我们看到控制台中已经打印并输出了刚才发送时所配置的用户属性。

```
packet: Packet {
  cmd: 'publish',
  retain: false,
  qos: 0,
  dup: false,
  length: 56,
  topic: '/nodejs/mqtt',
  payload: <Buffer 6e 6f 64 65 6a 73 20 6d 71 74 74 20 74 65 73 74>,
  properties: {
    userProperties: [Object: null prototype] { region: 'A', type: 'JSON' }
  }
}
Received Message: /nodejs/mqtt nodejs mqtt test
```

# 主题别名

## 什么是主题别名

主题别名（Topic Alias）是 MQTT v5.0 中新加入的与主题名（topic）相关的特性。它允许用户将主题长度较长且常用的主题名缩减为一个双字节整数来降低发布消息时的带宽消耗。

它是一个双字节整数，并将作为属性字段，编码在 `PUBLISH` 报文中可变报头部分。并且在实际应用中，将受到 `CONNECT` 报文和 `CONNACK` 报文中“主题别名最大长度”属性的限制。只要不超过该限制，任何主题名，都可以使用此特性缩减为编码长度 2 字节的整数。

## 为什么使用主题别名

在使用 MQTT v3 协议时。如果客户端在某次连接中需要发布大量相同主题的消息，那么在每一条 `PUBLISH` 报文中写入相同的主题名，就造成了客户端和服务端之间带宽资源的浪费。同时对于服务端而言，每次对相同主题名的 UTF-8 字符串进行解析，都是对计算资源的浪费。

设想这样的场景，在位置 A 以固定频率报告温度湿度的传感器：

- 使用 `/position/A/temperature` 作为该位置温度消息的主题（长度 23 字节）；
- 使用 `/position/A/humidity` 作为该位置湿度消息的主题（长度 20 字节）。

除去第一次发布消息外，之后的每个 `PUBLISH` 报文，都需要将“主题名”这个已经传递过的信息再次通过网络传输。即便抛开客户端和服务端之间额外的带宽消耗不言，对服务端来说，面对成千上万的传感器发布的大量消息，对每个客户端的每条消息，都要将同样的主题名字符串进行解析，这将造成了计算资源的浪费。

此时使用 MQTT v5 中的主题别名特性，就可以有效降低资源消耗。当客户端或服务端发布频率较高，且主题名长度较大的情景下，使用主题别名可以将每条消息中主题名的带宽消耗缩减为 2 字节，同时因为计算机处理整数的效率高于处理字符串的效率，对于客户端或服务端在报文解析时消耗的计算资源也有了一定的节约。

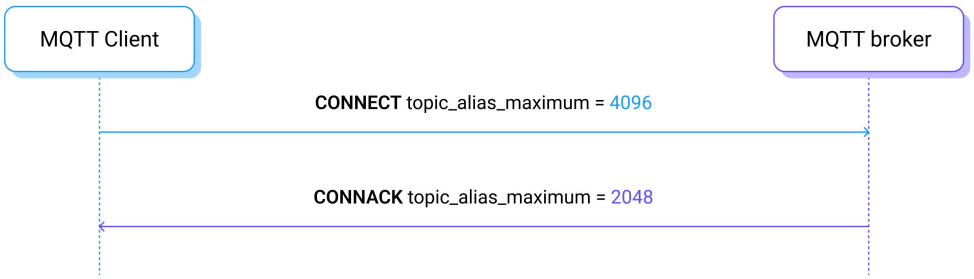
# 怎样使用主题别名

## 主题别名生命周期和作用范围

该值由客户端和服务端**各自维护**，且生命周期和作用范围仅限于当前连接。连接断开后需要再次使用主题别名需要重新建立**主题别名<=>主题名**映射关系。

## 主题别名最大值（Topic Alias Maximum）

在 MQTT 客户端和服务端使用主题别名进行发布消息前，需要对可以使用的最大主题别名长度进行约定。这部份信息交换将在 **CONNECT** 报文和 **CONNACK** 报文中完成。“主题别名最大值”也将以报文属性的形式，用双字节整数值编码在 **CONNECT** 和 **CONNACK** 报文的可变报头中。 客户端的 **CONNECT** 报文中"主题别名最大值"指示了本客户端在此次连接中服务端可以使用的最大主题别名数量；同样地，服务端发送的 **CONNACK** 报文中，也通过此值表明了当前连接中对端（客户端）可以使用的最大主题别名数量。



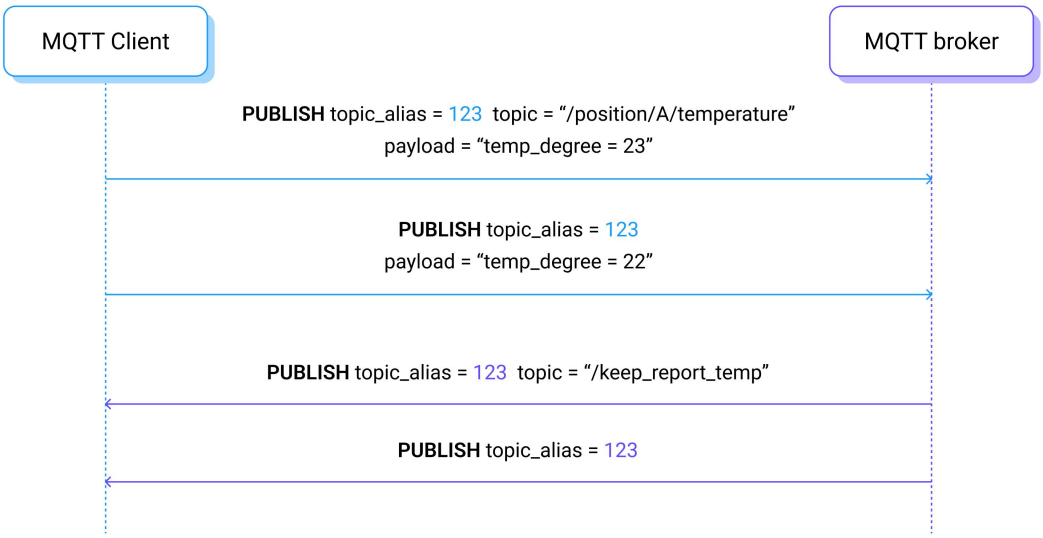

双端各自设置对端可以使用的最大主题别名数量

## 设置与使用主题别名

客户端（或服务端）在发送 **PUBLISH** 报文时，可以在可变报头的属性部分，用一个字节，值为 **0x23** 的标识符指示接下来 2 字节将是主题别名值。

但主题别名值不允许为 0，也不允许大于服务端（客户端）发送的 **CONNACK**（**CONNECT**）报文中设置的主题别名最大值。

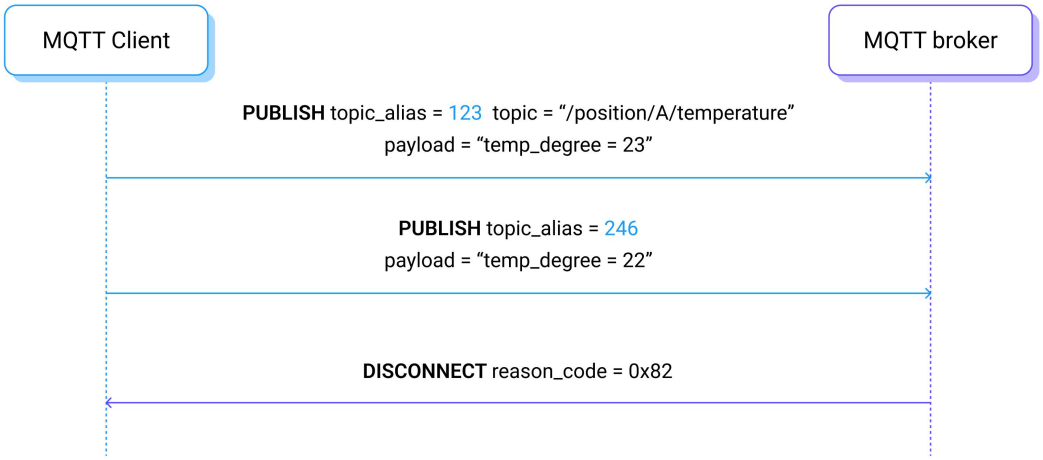
对端接收到带有主题别名值和非空主题名的 **PUBLISH** 报文后，将建立主题别名和主题名的映射关系，在此之后发送的 **PUBLISH** 报文中，便可以仅用长度 2 字节的主题别名发布消息，对端将使用通过之前建立的**主题别名<=>主题名**映射关系来处理消息中的主题。并且由于这一映射关系由双端各自维护，所以客户端与服务端可以使用值相同的主题别名互相发布消息。




MQTT Client 与 MQTT Broker 分别设置 topic_alias

## 使用未设置的主题别名

PUBLISH 报文中使用的主题别名值如果在此前的报文中未进行设置，即对端并未建立当前主题别名到某个主题名的映射关系，而此条报文的可变报头中主题名字段为空，对端将使用包含原因码（REASON_CODE）为 0x82 的 DISCONNECT 报文断开网络连接。

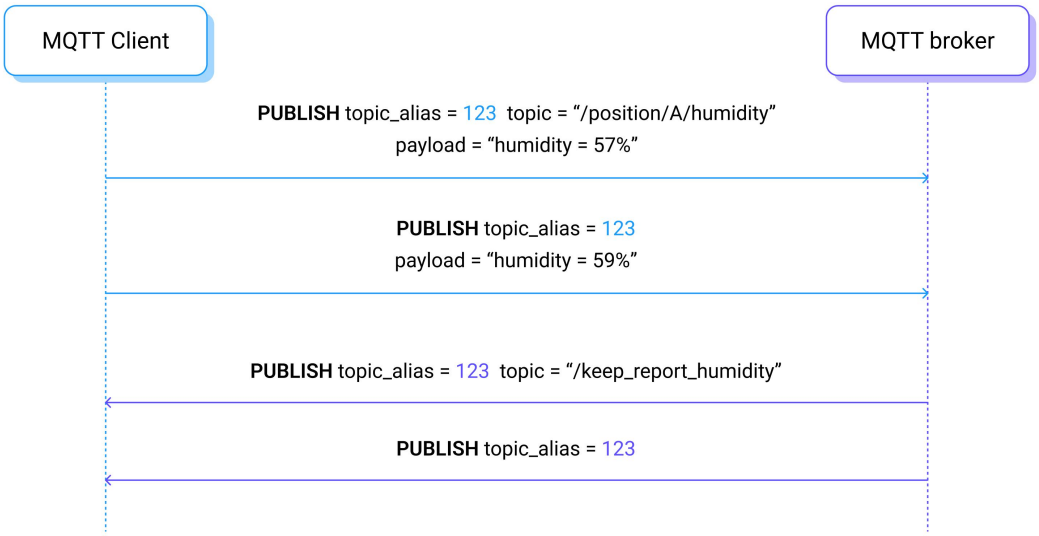



使用未建立映射关系的主题别名

## 重置主题别名

当对端已经根据本次连接中某个 PUBLISH 报文创建了一个主题别名<=>主题名的映射关系时，可以在下一次

发送 **PUBLISH** 报文时使用同样的主题别名值和非空的主题名来更新这个主题别名值到主题名的映射关系。




MQTT Client 与 MQTT Broker 分别更新主题别名所对应的主题名

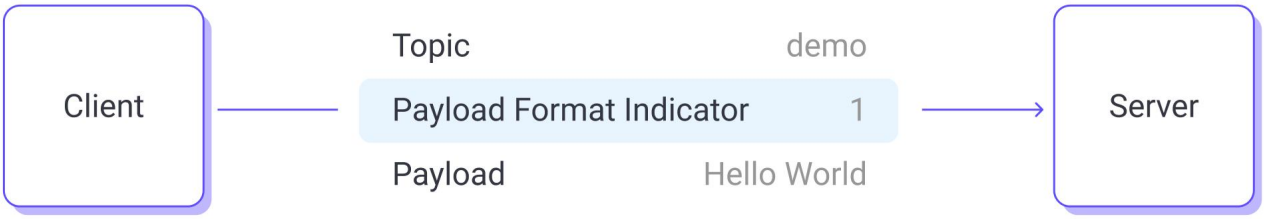
# 载荷格式指示与内容类型

## 什么是载荷格式指示（Payload Format Indicator）

Payload Format Indicator 是 MQTT 5.0 引入的一个全新属性，用来指示 MQTT 报文中有效载荷的格式。但 CONNECT、SUBSCRIBE 与 UNSUBSCRIBE 报文中有效载荷的格式都是固定不变的，所以实际上只有 PUBLISH 报文和 CONNECT 报文中的遗嘱消息需要声明其有效载荷的格式。

如果 Payload Format Indicator 的值为 0 或者没有指定这个属性，表示当前有效载荷是未指定的字节流；而如果这个属性的值为 1，则表示当前有效载荷是 UTF-8 编码的字符数据。

这允许接收者在无需解析具体内容的前提下检查有效载荷的格式，例如服务端可以检查有效载荷是否是一个有效的 UTF-8 字符串，避免将格式不正确的应用消息分发给订阅者。不过考虑到这个操作对服务端带来的负担和实际能够取得的收益，这通常是一个可选的行为。




## 什么是内容类型（Content Type）

Content Type 也是 MQTT 5.0 引入的一个全新属性，与 Payload Format Indicator 类似，它同样仅存在于 PUBLISH 报文和 CONNECT 报文的遗嘱消息中。

Content Type 的值是一个 UTF-8 编码的字符串，用来描述应用消息的内容，这可以帮助接收端了解如何解析应用消息的有效载荷。例如，消息的内容是一个 JSON 对象，那么 Content Type 可以被设置为 "json"。

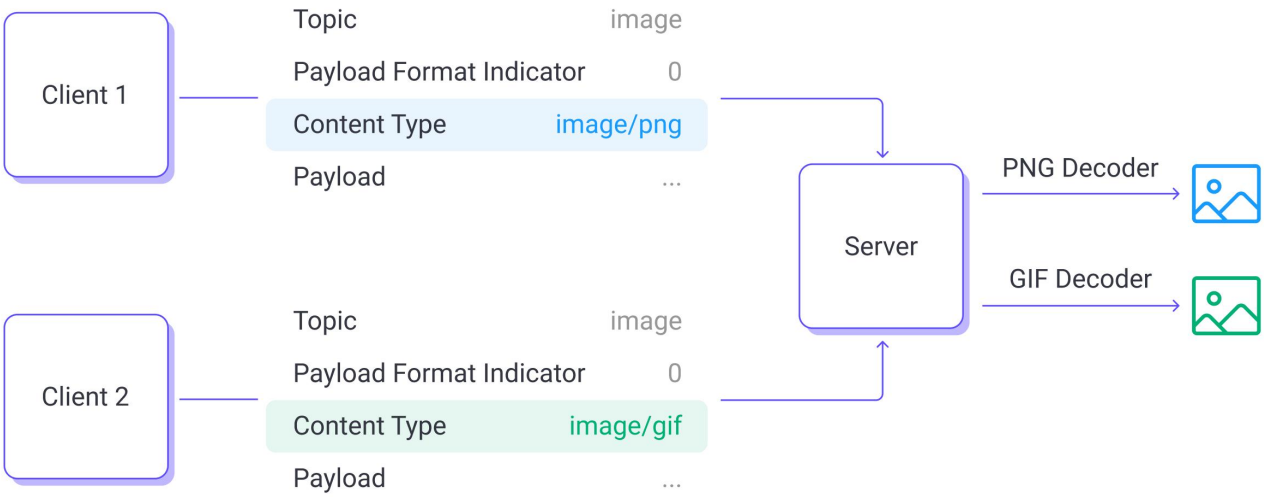
这个字符串的具体内容完全由发送端和接收端决定，在消息的整个传输过程中，服务端不会使用这个属性来验证消息内容的格式是否正确，它只负责将这个属性原封不动地转发给订阅者。

所以只要接收端能够理解，你甚至可以用“cocktail”来描述 JSON 类型。但为了避免造成不必要的困扰，

通常我们更推荐使用已知的 MIME 类型来描述消息内容，例如 `application/json`、`application/xml` 等等。

Content Type 在需要支持多种数据类型的场景中非常有用。比如当我们在聊天软件中向对方发送图片，图片可能有 png，gif，jpeg 等多种格式，如何向对端指示我们发送的二进制数据所对应的图片格式?

在 5.0 之前，我们可能会选择在主题中包含图片格式，比如 `to/userA/image/png`。但显然，随着支持的图片格式的增加，系统中的主题也会泛滥成灾。而在 5.0 中，我们只需要将 Content Type 属性设置为 `image/png` 即可。




## Payload Format Indicator 与 Content Type 必须一起使用吗

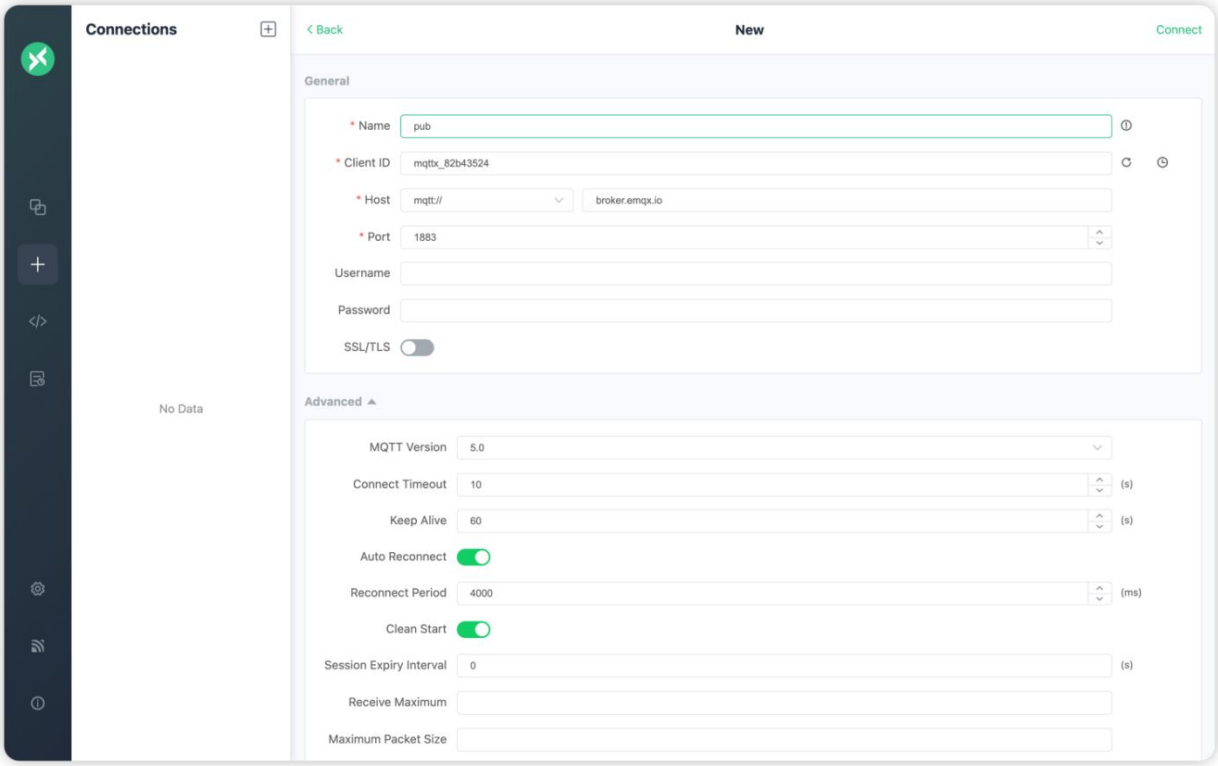
Payload Format Indicator 和 Content Type 是否需要同时使用，主要取决于我们的应用场景。

对于订阅端来说，他可以根据 Content Type 属性的值来判断消息的内容应该是 UTF-8 字符串还是二进制数据，所以 Payload Format Indicator 属性的意义不大。

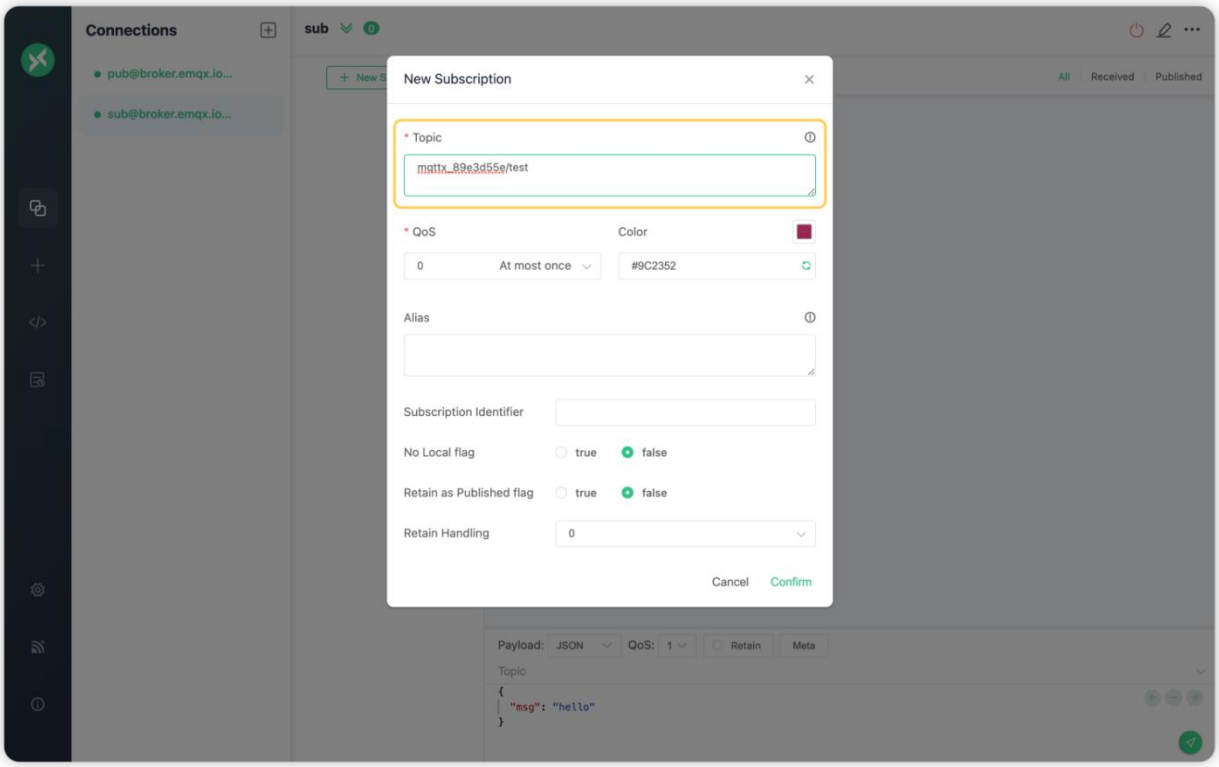
不过对于服务端来说，他并不了解 Content Type 的值的含义，所以如果我们希望服务端检查消息的有效载荷是否符合 UTF-8 编码规范，就必须借助 Payload Format Indicator 属性。

## 演示

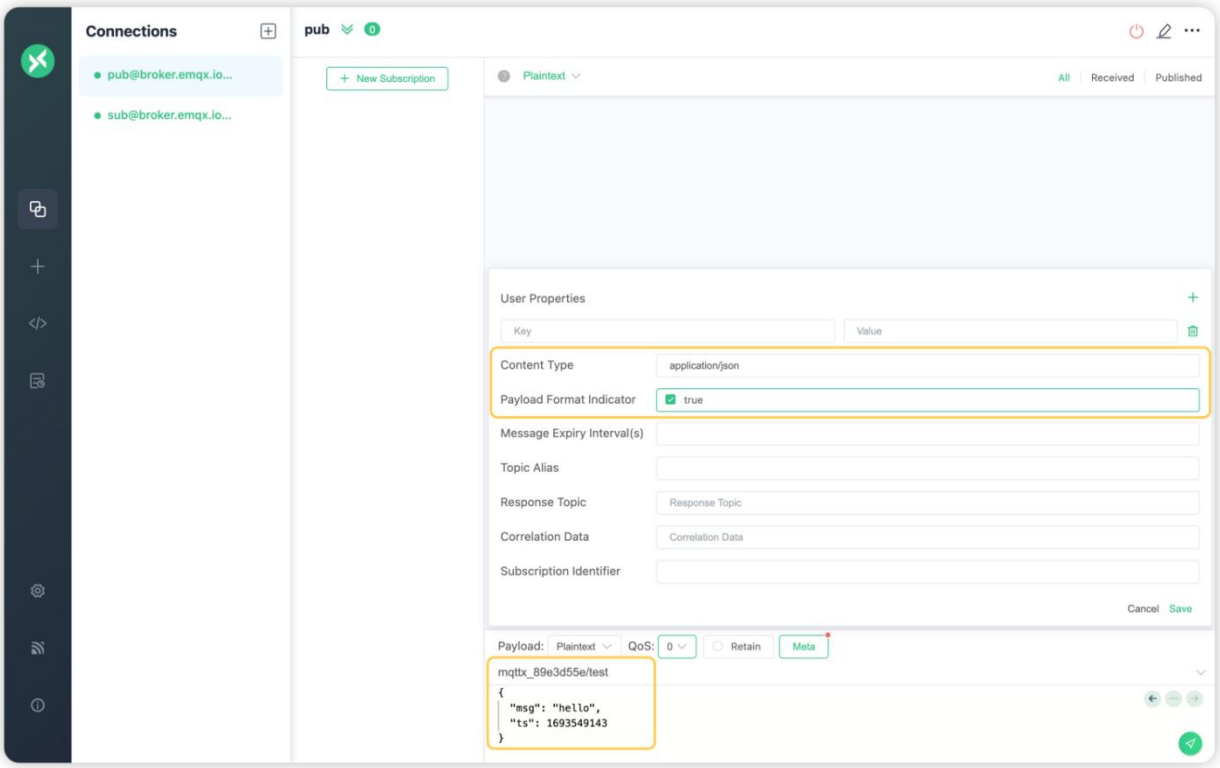
1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个名为 `pub` 的客户端连接用于发布消息，并且连接到免费的 公共 MQTT 服务器：

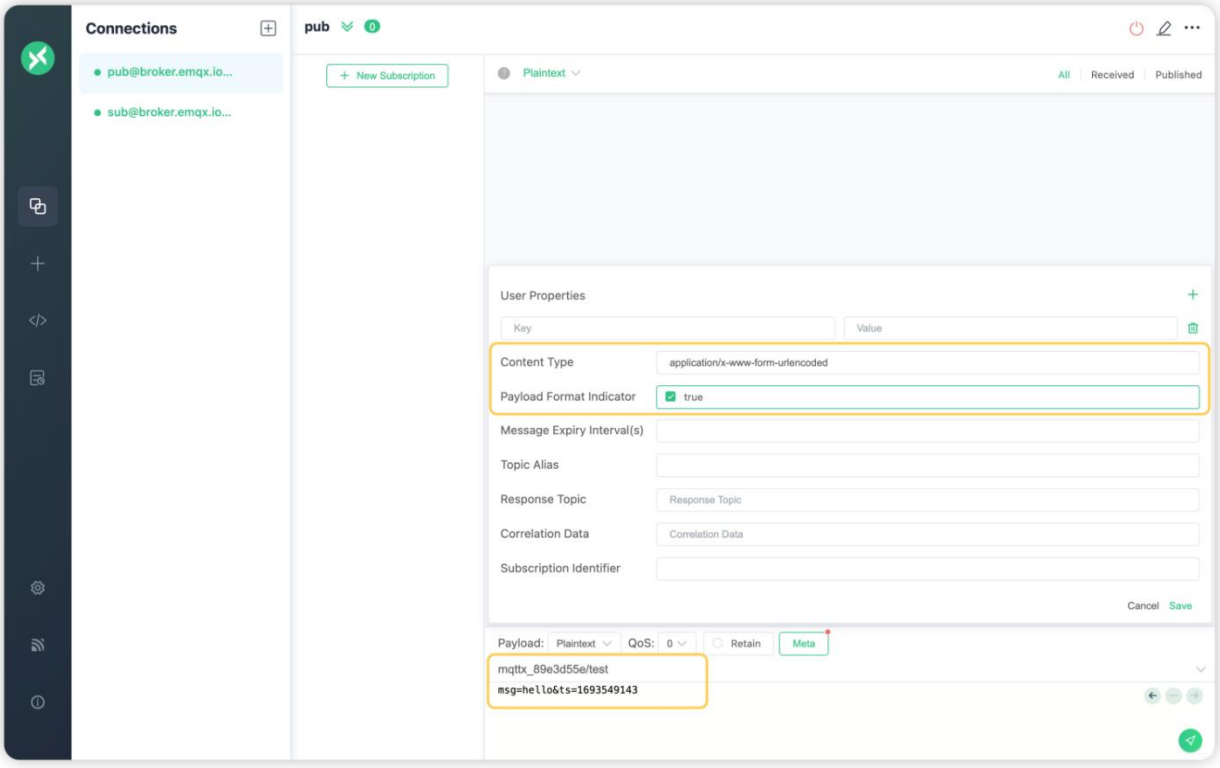



3. 以相同的方法创建一个名为 **sub** 的客户端连接并使用 Client ID 作为前缀订阅主题 **mqttx_89e3d55e/test**：

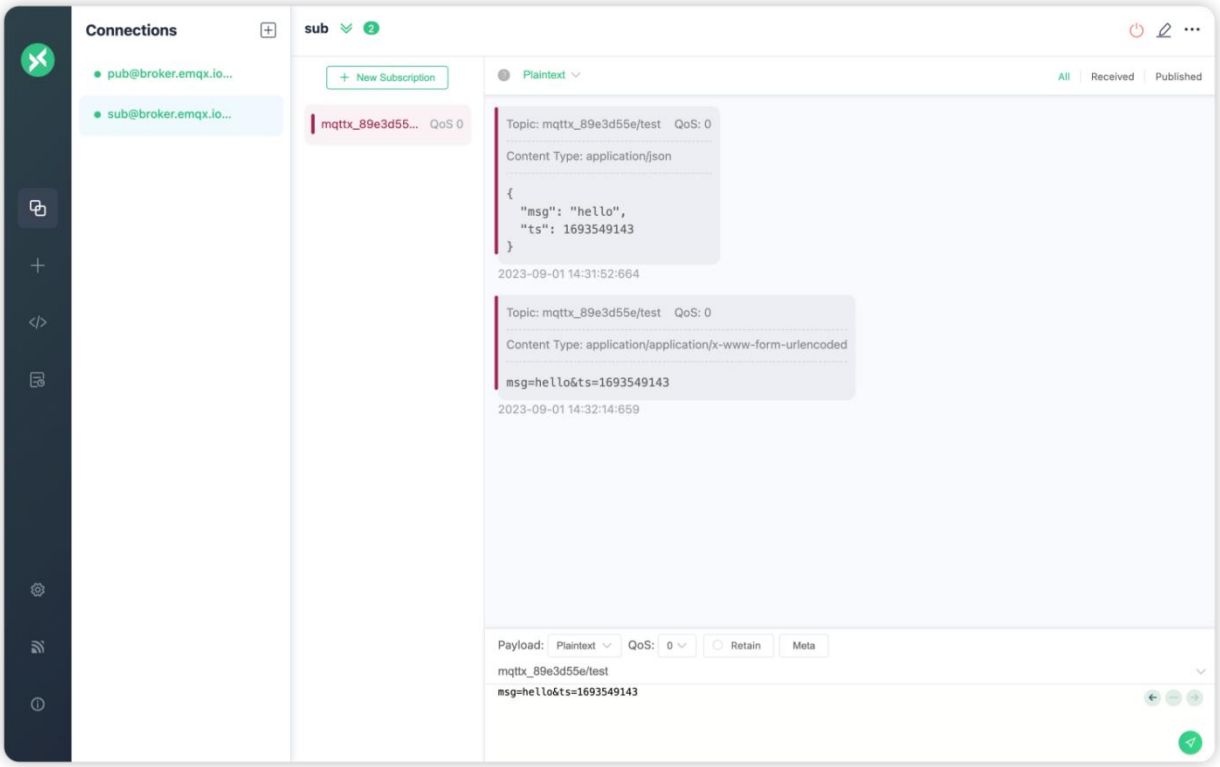


4. 然后回到 **pub** 客户端，点击消息栏的 Meta 按钮，将 Payload Format Indicator 设置为 **true**，Content Type 设置为 **application/json** 向主题 **mqttx_89e3d55e/test** 发布一条 JSON 格式的消息，然后修改为 **application/x-www-form-urlencoded** 再向同一主题发布一条 form 表单格式的消息：

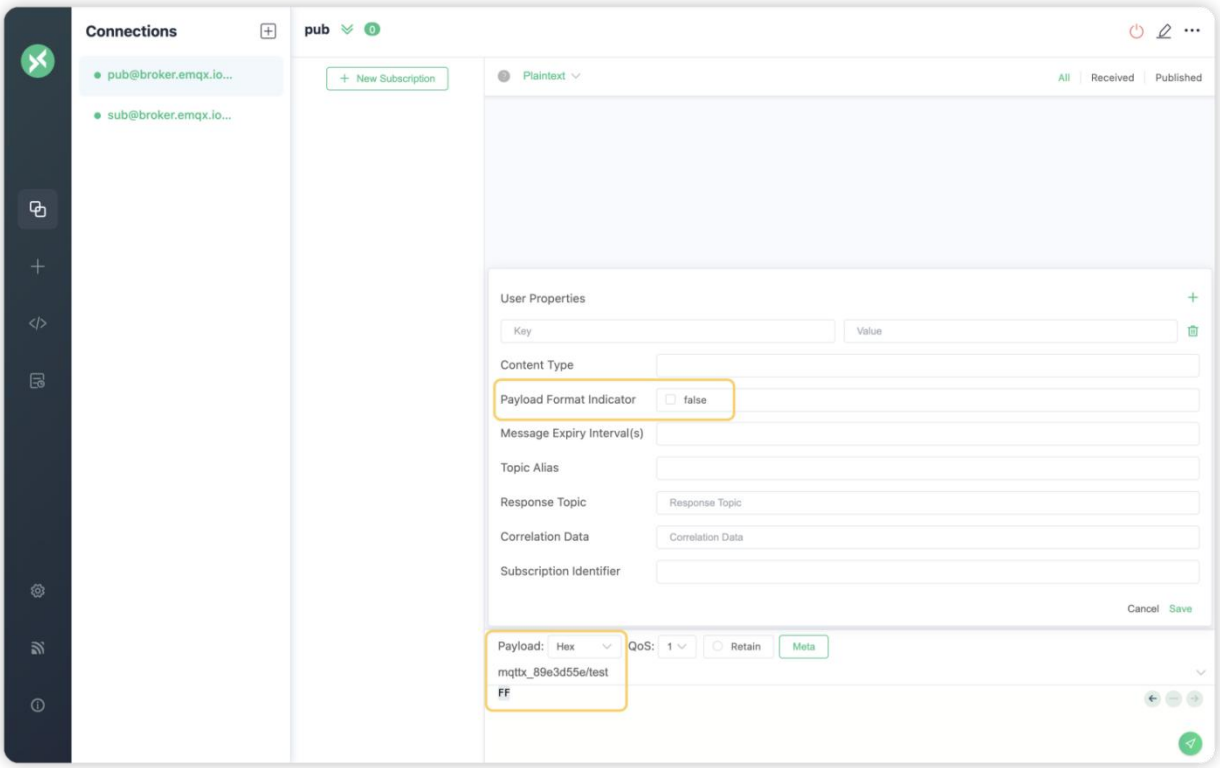


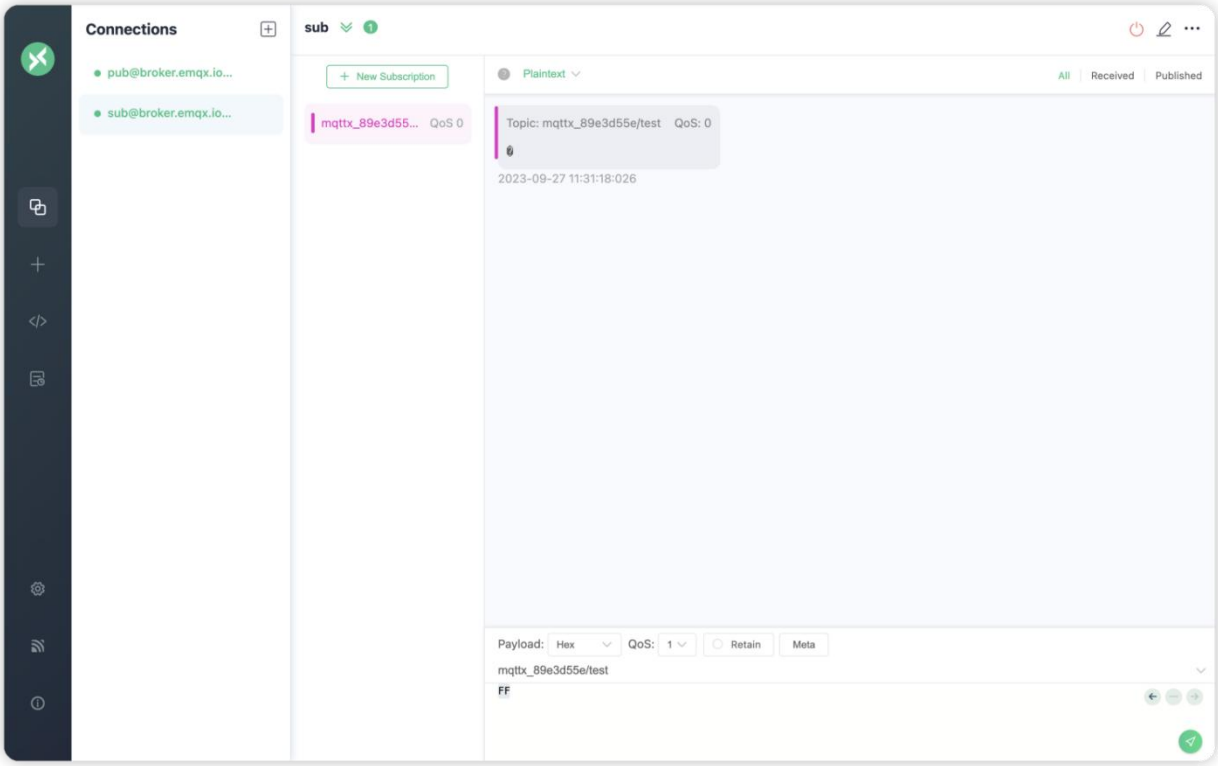
5. 消息中的 Content Type 将原封不动地转发给订阅端，因此订阅端可以根据 Content Type 的值得知应该如何解析 Payload 中的内容：




6. 回到发布端，将 Payload Format Indicator 设置为 false，并将 Payload 的编码格式改为 Hex，然后输入 FF 作为 Payload 内容并发送，0xFF 是一个典型的非 UTF-8 字符：



7. 虽然显示为乱码，但订阅端确实接收到了我们刚刚发送的 Payload 为 0xFF 的消息，这是因为出于性能考虑，EMQX 目前并未检查 Payload 格式：




在终端界面，我们还可以使用命令行工具 MQTTX CLI 来完成以上操作，我们可以使用以下命令订阅主题：

```
mqttx sub -h 'broker.emqx.io' -p 1883 -t 'random-string/demo' --output-mode clean
```

然后使用以下命令在发布消息时设置 Payload Format Indicator 和 Content Type 属性：

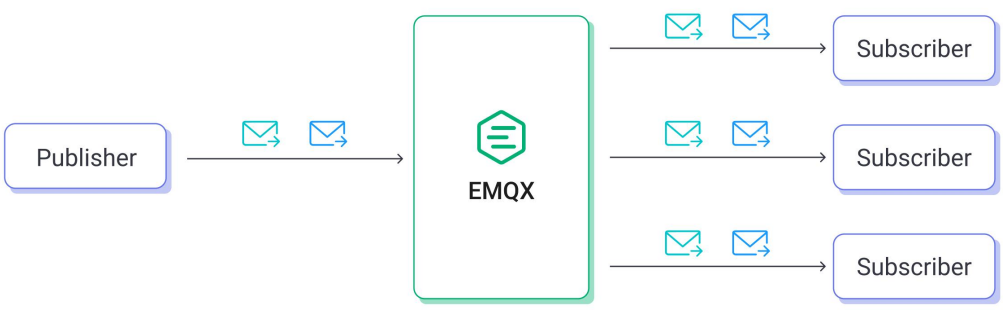
```
mqttx pub -h 'broker.emqx.io' -p 1883 -t 'random-string/demo' \
--payload-format-indicator \
--content-type 'application/json' \
-m '{"msg": "hello"}'
```

以上就是 MQTT 5.0 中 Payload Format Indicator 和 Content Type 属性的使用方法，你还可以在 这里 获取它们的 Python 示例代码。

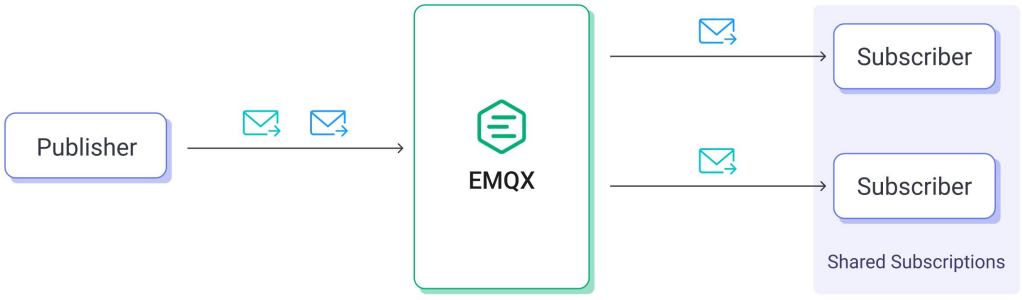
# 共享订阅

## 什么是共享订阅

在普通的订阅中，我们每发布一条消息，所有匹配的订阅端都会收到该消息的副本。当某个订阅端的消费速度无法跟上消息的生产速度时，我们没有办法将其中一部分消息分流到其他订阅端中来分担压力。这使订阅端容易成为整个消息系统的性能瓶颈。




所以 MQTT 5.0 引入了共享订阅特性，它使得 MQTT 服务端可以在使用特定订阅的客户端之间均衡地分配消息负载。这表示，当我们有两个客户端共享一个订阅时，那么每个匹配该订阅的消息都只会有一个副本投递给其中一个客户端。




共享订阅不仅为消费端带来了极佳的水平扩展能力，使我们可以应对更高的吞吐量，还为其带来了高可用性，即使共享订阅组中的一个客户端断开连接或发生故障，其他客户端仍然可以继续处理消息，在必要时还可以接管原先流向该客户端的消息流。

## 共享订阅如何工作

使用共享订阅，我们不需要对客户端的底层代码进行任何改动，只需要在订阅时使用遵循以下命名规范的

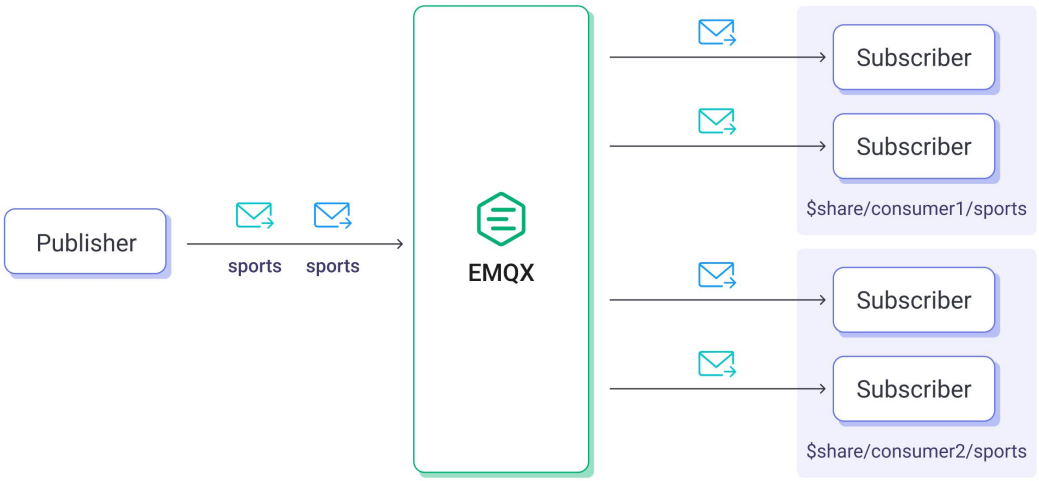
主题即可：

```
1.  $share/{Share Name}/{Topic Filter}
```

其中 **$share** 是一个固定的前缀，以便服务端知道这是一个共享订阅主题。**{Topic Filter}** 则是我们实际想要订阅的主题。

中间的 **{Share Name}** 是一个由客户端指定的字符串，表示当前共享订阅使用的共享名。很多时候，**{Share Name}** 这个字段也会被叫作 Group Name 或者 Group ID，这确实会更容易理解一些。

需要共享同一个订阅的一组订阅会话，必须使用相同的共享名。所以 **$share/consumer1/sport/#** 和 **$share/consumer2/sport/#** 属于不同的共享订阅组。当一个消息同时与多个共享订阅组使用的过滤器匹配时，服务端会在每个匹配的共享订阅组中选择一个会话发送该消息的副本。这在某个主题的消息有多个不同类型的消费者时非常有用。

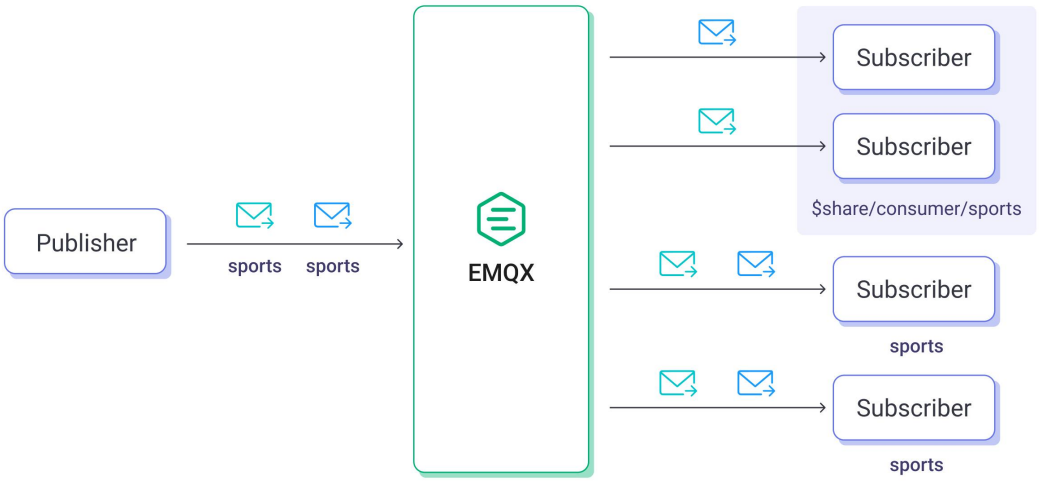



但是，两个订阅的共享名 **{Share Name}** 相同，并不表示它们一定是相同的共享订阅。只有 **{Share Name}/{Topic Filter}** 才能唯一地标识一个共享订阅组，下面这些订阅主题均属于不同的共享订阅组：

- **$share/consumer1/sport/tennis/+**
- **$share/consumer2/sport/tennis/+**
- **$share/consumer1/sport/#**
- **$share/comsumer1/finance/#**

共享订阅和普通订阅互不影响，当某个消息同时与共享订阅和普通订阅匹配时，服务端会向每个匹配的普通订阅的客户端发送该消息的副本，同时向每个匹配的共享订阅组中的其中一个会话发送该消息的副本。

如果这些订阅来自同一个客户端，那么这个客户端可能会收到该消息的多个副本。




## 共享订阅的负载均衡策略

共享订阅的核心在于服务端如何在客户端之间分配消息负载。比较常见的负载均衡策略有以下几种：

- 随机（Random），在共享订阅组内随机选择一个会话发送消息。
- 轮询（Round Robin），在共享订阅组内按顺序选择一个会话发送消息，循环往复。
- 哈希（Hash），基于某个字段的哈希结果来分配。
- 粘性（Sticky），在共享订阅组内随机选择一个会话发送消息，此后保持这一选择，直到该会话结束再重复这一过程。
- 本地优先（Local），随机选择，但优先选择与消息的发布者处于同一节点的会话，如果不存在这样的会话，则退化为普通的随机策略。

**随机** 和 **轮询** 这两种策略实现的均衡效果较为接近，所以它们在应用场景上的区别不大，但 **随机** 策略实际的均衡效果通常还会受到服务端采用的随机算法的影响。

在实际应用中，消息之间可能存在关联，比如属于同一张图片的多个分片显然不适合分发给多个订阅者。在这种情况下，我们就需要基于 Client ID 或者 Topic 的 **哈希** 策略来选择会话。这可以保证来自同一个发布端或者主题的消息始终由共享订阅组中的同一个会话处理。当然，**粘性** 策略也有相同的效果。

**本地优先** 策略比 **随机** 策略更合适在集群中使用，优先选择本地订阅端的策略可以有效降低消息的延迟。不过使用这一策略的前提是我们可以确保发布端和订阅端比较均衡地分布在每个节点上，以免不同订阅端

上的消息负载差别过大。

## MQTT 3.1.1 客户端如何使用共享订阅

早在 MQTT 5.0 发布之前，EMQX 就已经设计了一个共享订阅的方案，并且被许多用户采用。与 MQTT 5.0 的标准方案大同小异，我们在 MQTT 3.1.1 中约定以下格式的主题作为共享订阅主题：

```
1.  $queue/{Topic File}
```

前缀 **$queue** 表示这是一个共享订阅主题，**{Topic Filter}** 则是我们实际想要订阅的主题。它等效于 MQTT 5.0 中的 **$share/queue/{Topic Filter}**，即共享名固定为 **queue**。所以这一方案不支持使用相同主题过滤器的多个共享订阅组。

由于 MQTT 5.0 约定的共享订阅主题格式 **$share/{Share Name}/{Topic Filter}** 在 MQTT 3.1.1 中也是一个完全合法的主题，而共享订阅的逻辑完全在 MQTT 服务端中实现，客户端只需要修改订阅的主题内容即可。所以即便是仍在使用 MQTT 3.1.1 的设备，现在也可以直接使用 MQTT 5.0 才提供的共享订阅功能。

## 共享订阅使用场景

以下是几个典型的共享订阅的使用场景：

- 后端消费能力与消息的生产能力不匹配时，我们可以借助共享订阅让更多的客户端一起分担负载。
- 系统需要保证高可用性，特别是在大量消息流入的关键业务上，我们可以通过共享订阅来避免单点故障。
- 消息的流入量可能会在未来快速增长，需要消费端能够水平扩展，我们可以通过共享订阅来提供高扩展性。

## 共享订阅使用建议

### 在共享订阅组内使用相同的 QoS

MQTT 虽然允许一个共享订阅组内的会话使用不同的 QoS 等级，但这可能会使消息在投递给同一个组内的不同会话时存在不同的质量保证。相应地，在出现一些问题的时候，我们的调试也将变得困难重重。所

以我们最好在共享订阅组内使用相同的 QoS。

### 合理地设置会话过期时间

持久会话与共享订阅一起使用是非常常见的。但需要注意，即便共享订阅组中的某个客户端离线，但只要它的会话与订阅仍在存在时，MQTT 服务端仍然会向此会话分发消息。考虑到客户端可能因为故障等原因长时间离线，如果会话的过期时间过长，那么这段时间内将有很多消息因为被投递给离线客户端而无法得到处理。

一个更好的选择可能是，一旦订阅端离线，即便会话没有过期，MQTT 服务端在分配消息负载时也不再考虑这个订阅端。虽然与普通订阅的行为不同，但这是 MQTT 协议允许的。

## 演示

为了更好地展示共享订阅的效果，这里我们直接使用 MQTTX CLI 这个 MQTT 命令行客户端工具来进行演示。

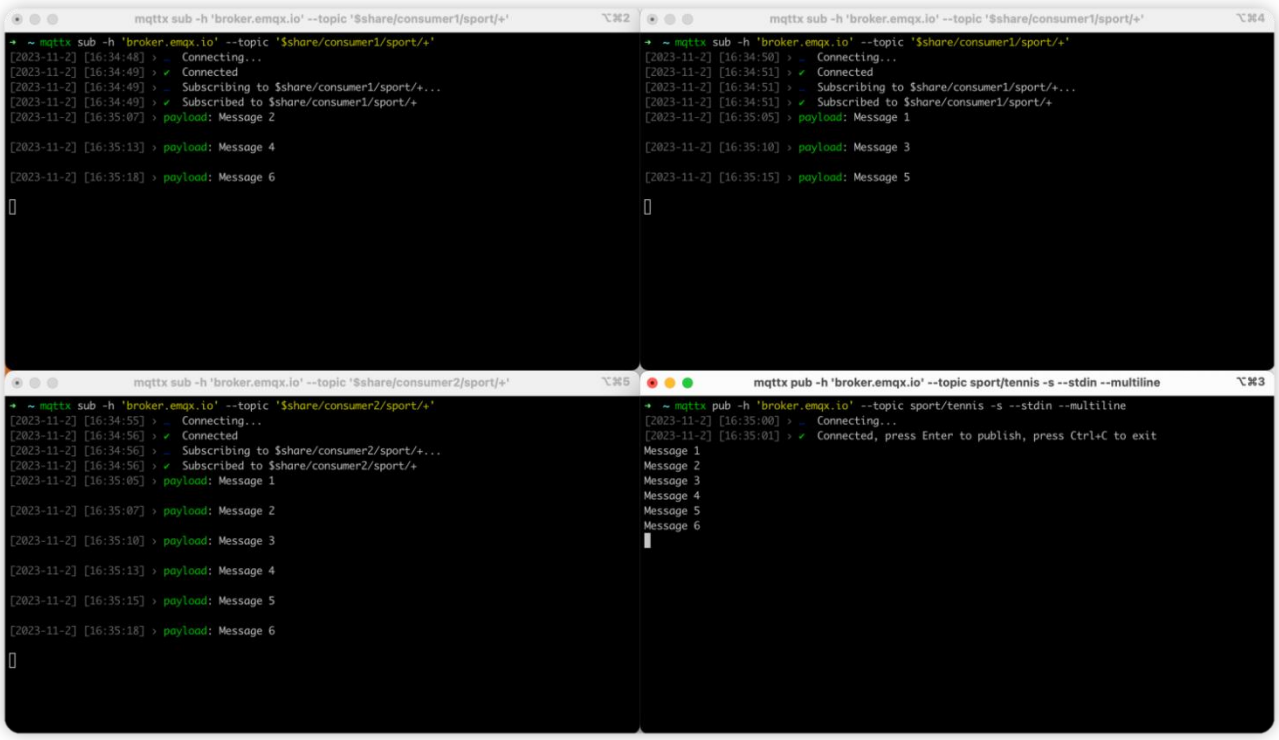
启动三个终端窗口，使用以下命令创建三个连接至 免费的公共 MQTT 服务器 的客户端并分别订阅主题 **$share/consumer1/sport/+**、**$share/consumer1/sport/+** 和 **$share/consumer2/sport/+**：

```
1.  mqttx sub -h 'broker.emqx.io' --topic '$share/consumer1/sport/+'
```

然后启动一个新的终端窗口，使用以下命令向主题 **sport/tennis** 发布 6 条消息。这里我们使用了 **--multiline** 选项，以每次键入回车的方式发送多条消息：

```
1.  mqttx pub -h 'broker.emqx.io' --topic sport/tennis -s --stdin --multiline
```

EMQX 默认为共享订阅使用的负载均衡策略为 Round Robin，所以我们将看到 **consumer1** 组内的两个订阅端交替收到我们发布的消息，而共享订阅组 **consumer2** 中只有一个订阅端，所以它将收到所有消息：




这只是一个非常简单的示例，你还可以尝试随时加入或退出共享订阅组，观察 EMQX 是否及时地按最新的订阅分配负载，或者自行安装 EMQX 然后观察不同负载均衡策略的表现。

# 订阅选项

## 订阅选项

在 MQTT 中，一个订阅由一个 主题过滤器 和对应的订阅选项组成。所以理论上，我们可以为每个订阅都设置不同的订阅选项。

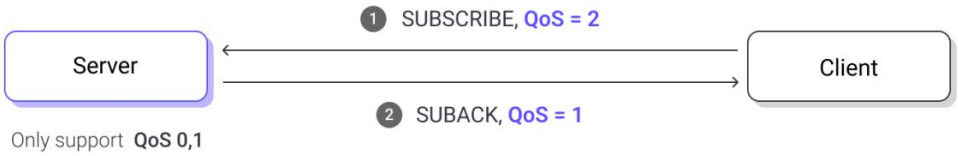
MQTT 5.0 提供了 4 个订阅选项，分别是 QoS、No Local、Retain As Published、Retain Handling，而 MQTT 3.1.1 则仅提供了 QoS 这一个订阅选项。不过这些 MQTT 5.0 新增的订阅选项的默认行为，仍与 MQTT 3.1.1 保持一致，如果你正准备从 MQTT 3.1.1 升级到 MQTT 5.0，这会非常地友好。

现在，让我们一起看看这些订阅选项的作用吧。

### QoS

QoS 是最常用的一个订阅选项，它表示服务端在向订阅端发送消息时可以使用的最大 QoS 等级。

客户端可能会在订阅时指定一个小于 2 的 QoS，因为它的实现不支持 QoS 1 或者 QoS 2。而如果服务端支持的最大 QoS 小于客户端订阅时请求的最大 QoS，那么显然服务端将无法满足客户端的要求，这时服务端就会通过订阅的响应报文（SUBACK）告知订阅端最终授予的最大 QoS 等级，订阅端可以自行评估是否接受并继续通信。




一个简单的计算公式：

> 1. 服务端最终授予的最大 QoS = min（ 服务端支持的最大 QoS，客户端请求的最大 QoS ）

但是，我们在订阅时请求的最大 QoS，并不能限制发布端发布消息时使用的 QoS。当我们订阅时请求的最大 QoS，小于消息发布时的 QoS 时，为了尽可能地投递消息，服务端不会忽略这些消息，而是会在转发时对这些消息的 QoS 进行降级处理。

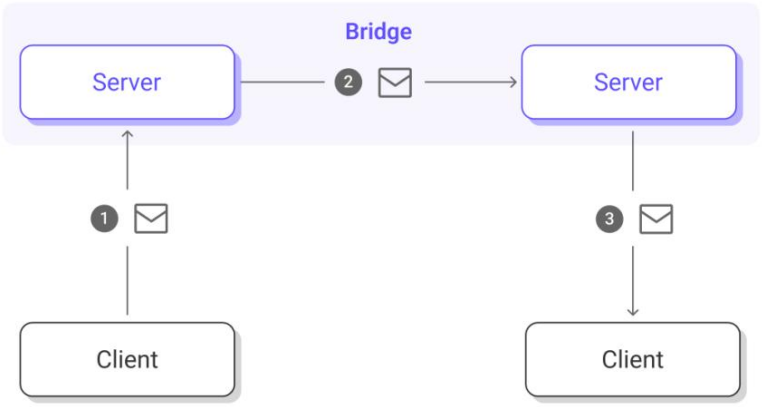

同样，我们也有一个简单的计算公式：

> 1. 消息被转发时的 QoS = min（ 消息原始的 QoS，服务端最终授予的最大 QoS ）

## No Local

No Local 只有 0 和 1 两个可取值，为 1 表示服务端不能将消息转发给发布这个消息的客户端，为 0 则相反。

这个选项通常被用在桥接场景中。桥接本质上是两个 MQTT Server 建立了一个 MQTT 连接，然后相互订阅一些主题，Server 将客户端的消息转发给另一个 Server，而另一个 Server 则可以将消息继续转发给它的客户端。




那么最简单的一个例子，我们假设两个 MQTT Server 分别是 Server A 和 Server B，它们分别向对方订阅了 `#` 主题。现在，Server A 将一些来自客户端的消息转发给了 Server B，而当 Server B 查找匹配的订阅时，Server A 也会位于其中。如果 Server B 将消息转发给了 Server A，那么同样 Server A 在收到消息后又会把它们再次转发给 Server B，这样就陷入了无休止的转发风暴。

而如果 Server A 和 Server B 在订阅 `#` 主题的同时，将 No Local 选项设置为 1，就可以完美地避免这个问题。

## Retain As Published

Retain As Published 同样只有 0 和 1 两个可取值，为 1 表示服务端在向此订阅转发应用消息时需要保持消息中的 Retain 标识不变，为 0 则表示必须清除。

Retain As Published 与 No Local 一样，同样也是主要适用于桥接场景。我们知道当服务端收到一条保留消息时，除了将它存储起来，还会将它像普通消息一样转发给当前已经存在的订阅者，并且在转发时会清除消息的 Retain 标识。

这在桥接场景下带来了一些问题。我们继续沿用前面的设定，当 Server A 将保留消息转发给 Server B 时，由于消息中的 Retain 标识已经被清除，Server B 将不会知道这原本是一条保留消息，自然不会再存储它。这就导致了保留消息无法跨桥接使用。

那么在 MQTT 5.0 中，我们可以让桥接的服务端在订阅时将 Retain As Published 选项设置为 1，来解决这个问题。


## Retain Handling

Retain Handling 这个订阅选项被用来向服务端指示当订阅建立时，是否需要发送保留消息。

我们知道默认情况下，只要订阅建立，那么服务端中与订阅匹配的保留消息就会下发。

但某些时候，客户端可能并不想接收保留消息，比如客户端在连接时复用了会话，但是客户端无法确认上一次连接中是否成功创建了订阅，所以它可能会再次发起订阅。如果订阅已经存在，那么可能保留消息已经被消费过了，也可能服务端已经在会话中缓存了一些离线期间到达的消息，这时客户端可能并不希望服务端发布保留消息。

另外，客户端也可能在任何时刻都不想收到保留消息，即使是第一次订阅。比如我们将开关状态作为保留消息发送，但对某个订阅端来说，开关事件将触发一些操作，那么在这种情况下不发送保留消息是很有用的。

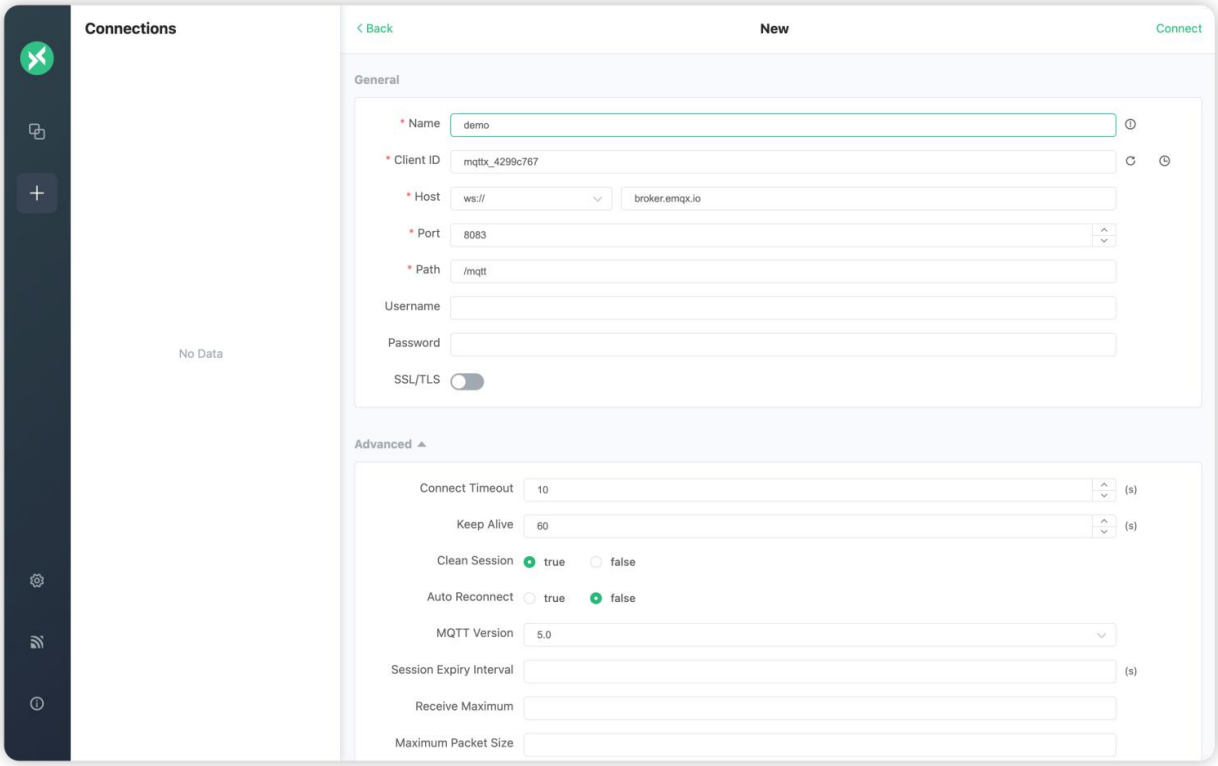
这三种不同的行为，我们可以通过 Retain Handling 来选择。

- 将 Retain Handling 设置为 0，表示只要订阅建立，就发送保留消息；
- 将 Retain Handling 设置为 1，表示只有建立全新的订阅而不是重复订阅时，才发送保留消息；
- 将 Retain Handling 设置为 2，表示订阅建立时不要发送保留消息。

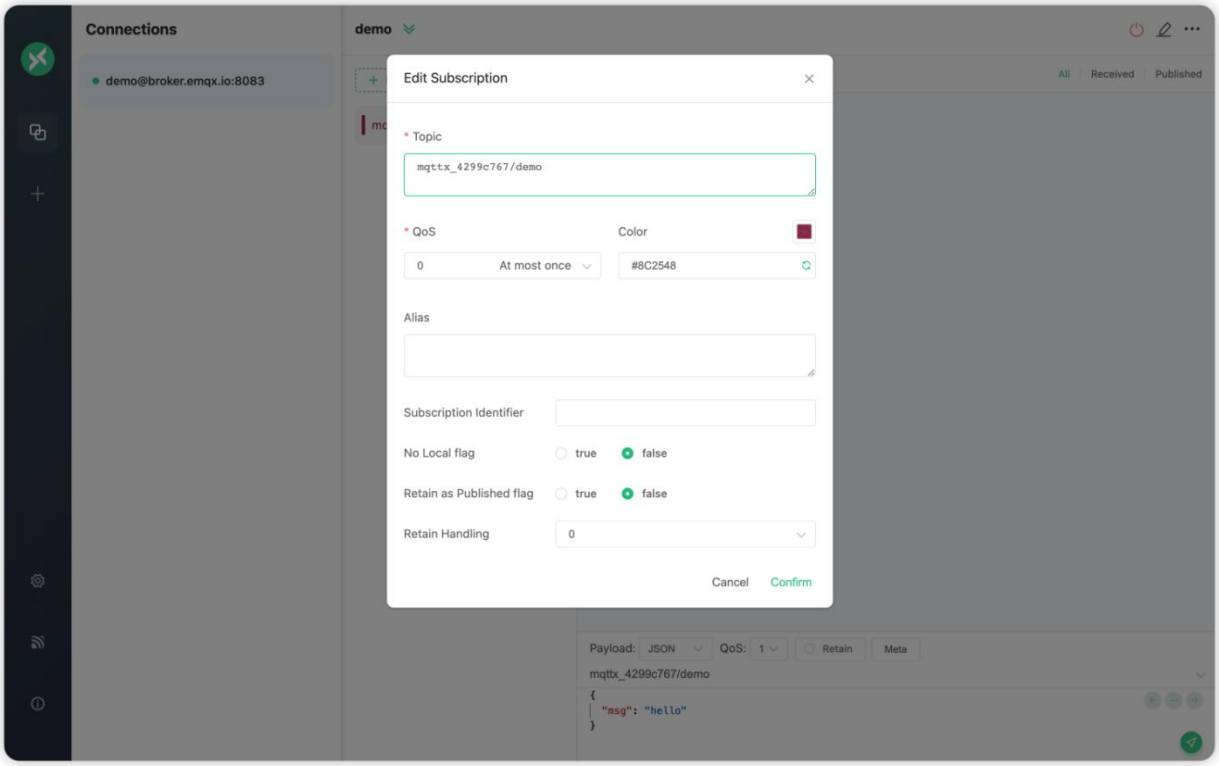
# 演示

## 订阅选项 QoS 的演示

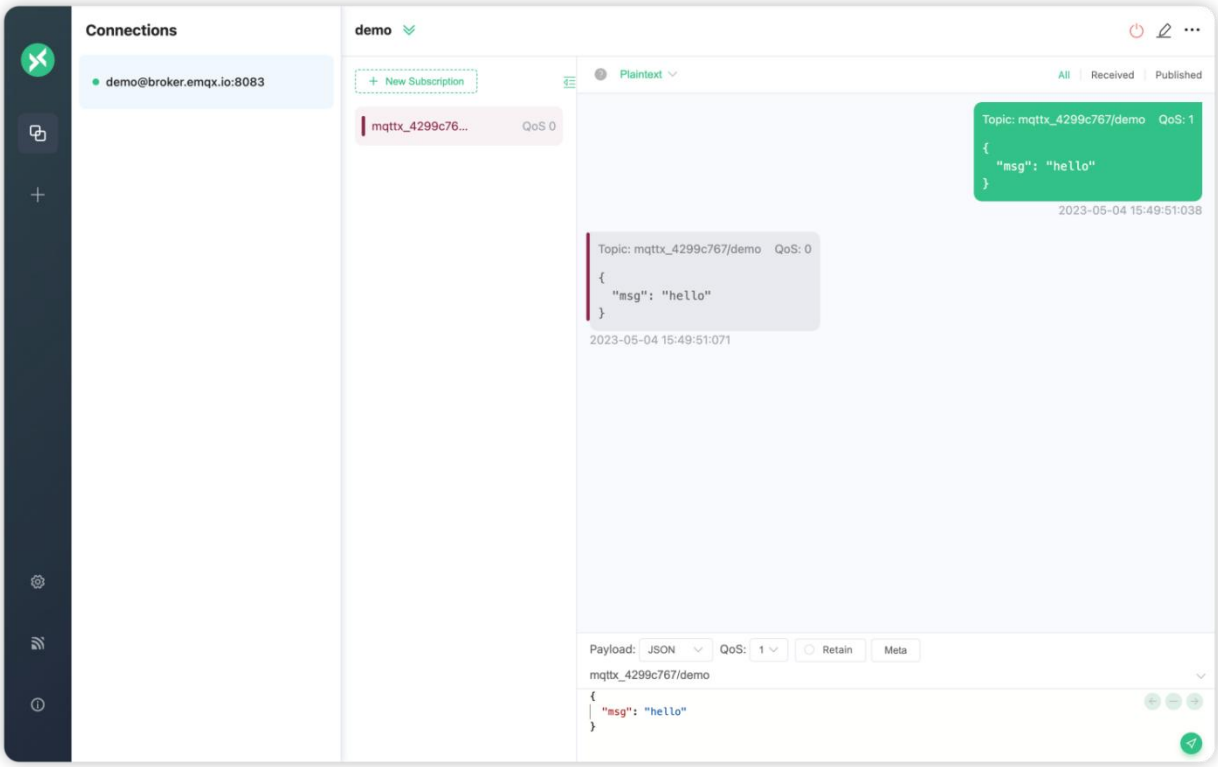
1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个使用 WebSocket 的 MQTT 连接，并且连接免费的 公共 MQTT 服务器：




3. 连接成功后，我们订阅主题 **mqttx_4299c767/demo**，并指定 QoS 为 0。由于公共服务器可能同时被很多人使用，为了避免主题与别人重复，我们可以将 Client ID 作为主题前缀：

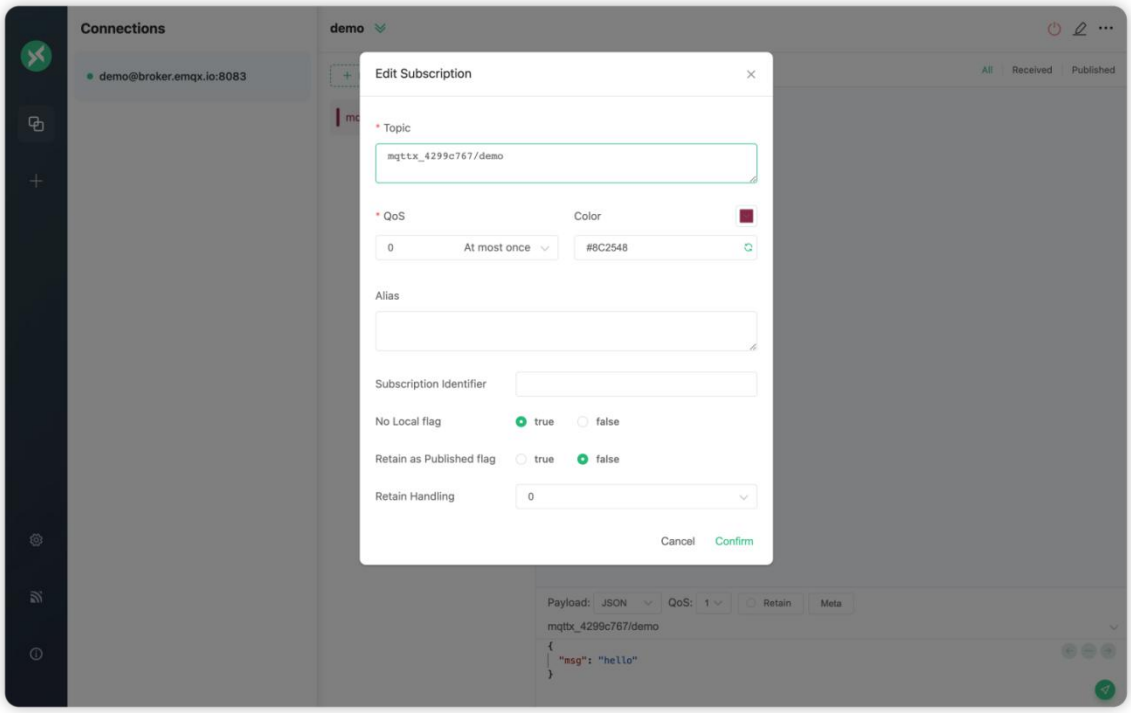



4. 订阅成功后，我们向主题 **mqttx_4299c767/demo** 发布一条 QoS 1 消息，这时我们将看到，我们发出的是 QoS 1 消息，但收到的却是 QoS 0 消息，这说明发生了 QoS 降级：

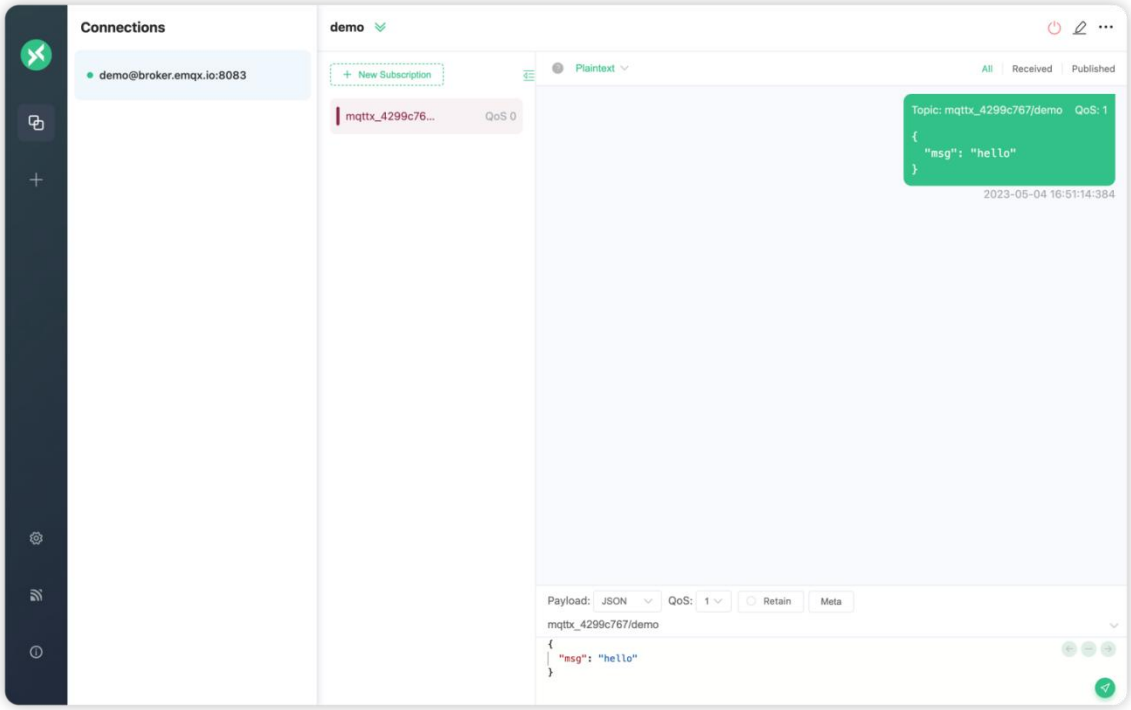


## 订阅选项 No Local 的演示

1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个使用 WebSocket 的 MQTT 连接，并且连接免费的 公共 MQTT 服务器。
3. 连接成功后，我们订阅主题 **mqttx_4299c767/demo**，并且将 No Local 设置为 true：

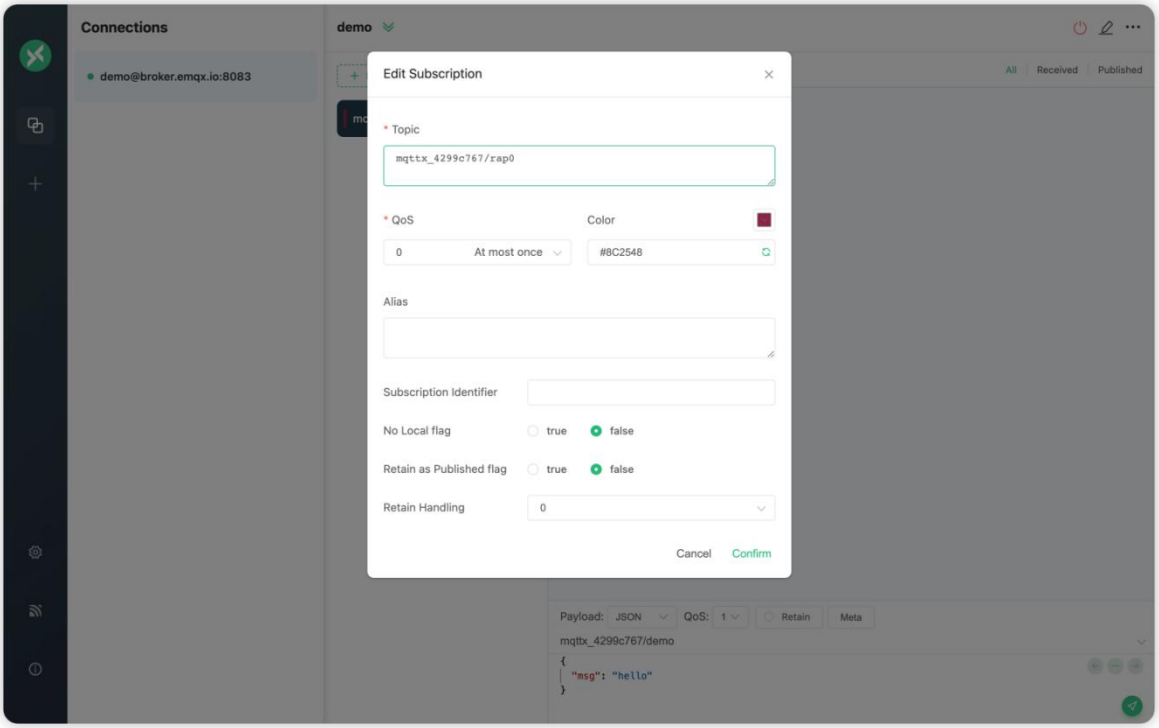


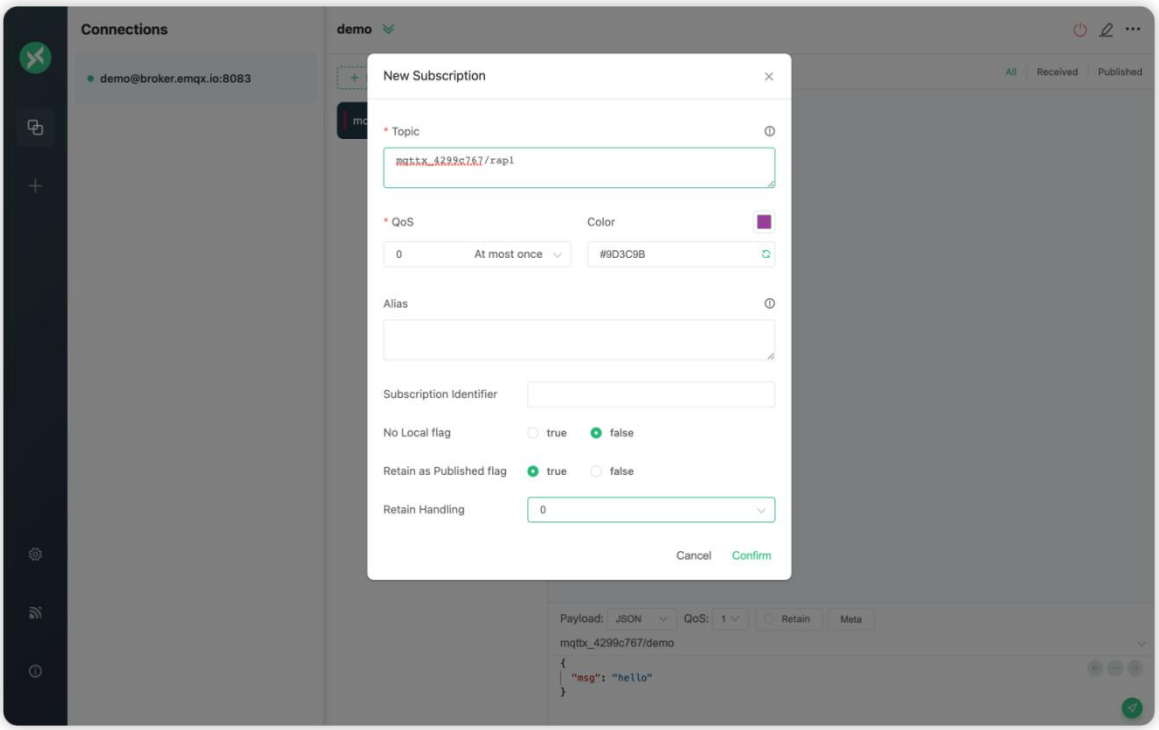

4. 订阅成功后，与前面 QoS 的演示一样，我们还是由订阅端自己来发布消息，但这一次我们会发现订阅端将无法收到消息：



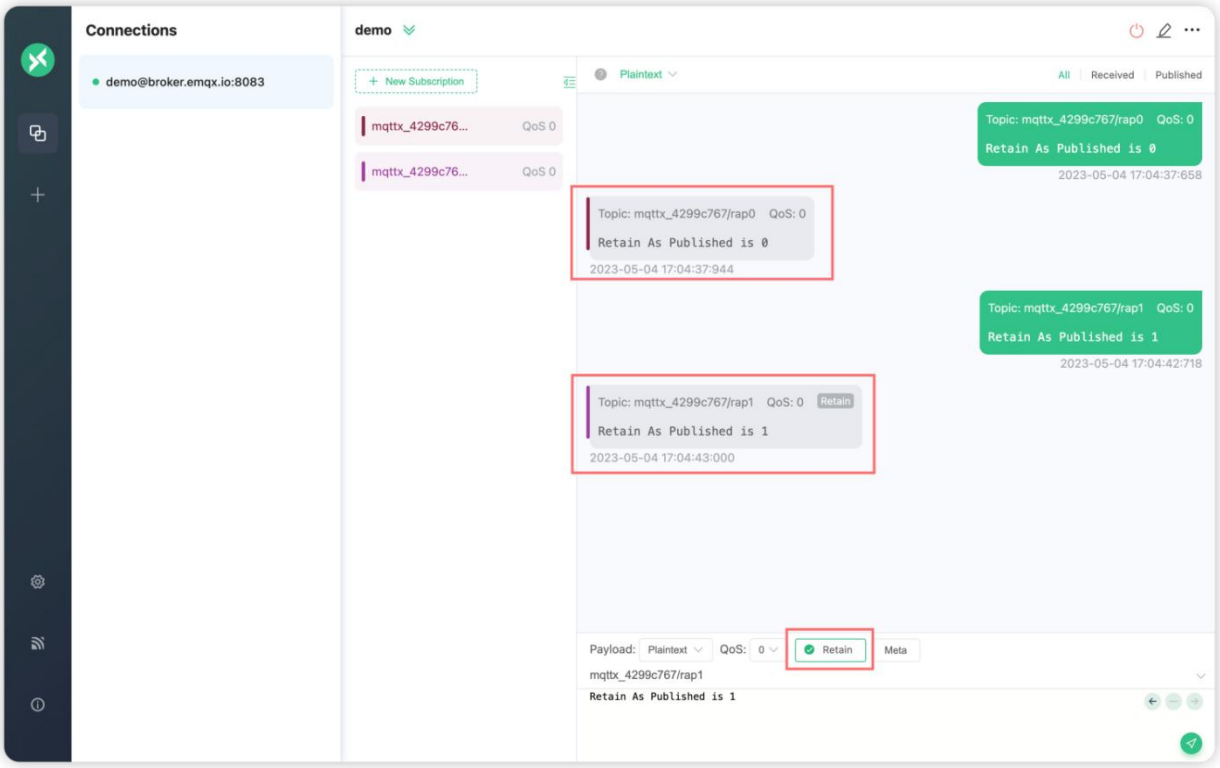
## 订阅选项 Retain As Published 的演示

1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个使用 WebSocket 的 MQTT 连接，并且连接免费的 公共 MQTT 服务器。
3. 连接成功后，我们先订阅主题 **mqttx_4299c767/rap0**，并且将 Retain As Published 设置为 false，然后订阅主题 **mqttx_4299c767/rap1**，并且将 Retain As Published 设置 true：



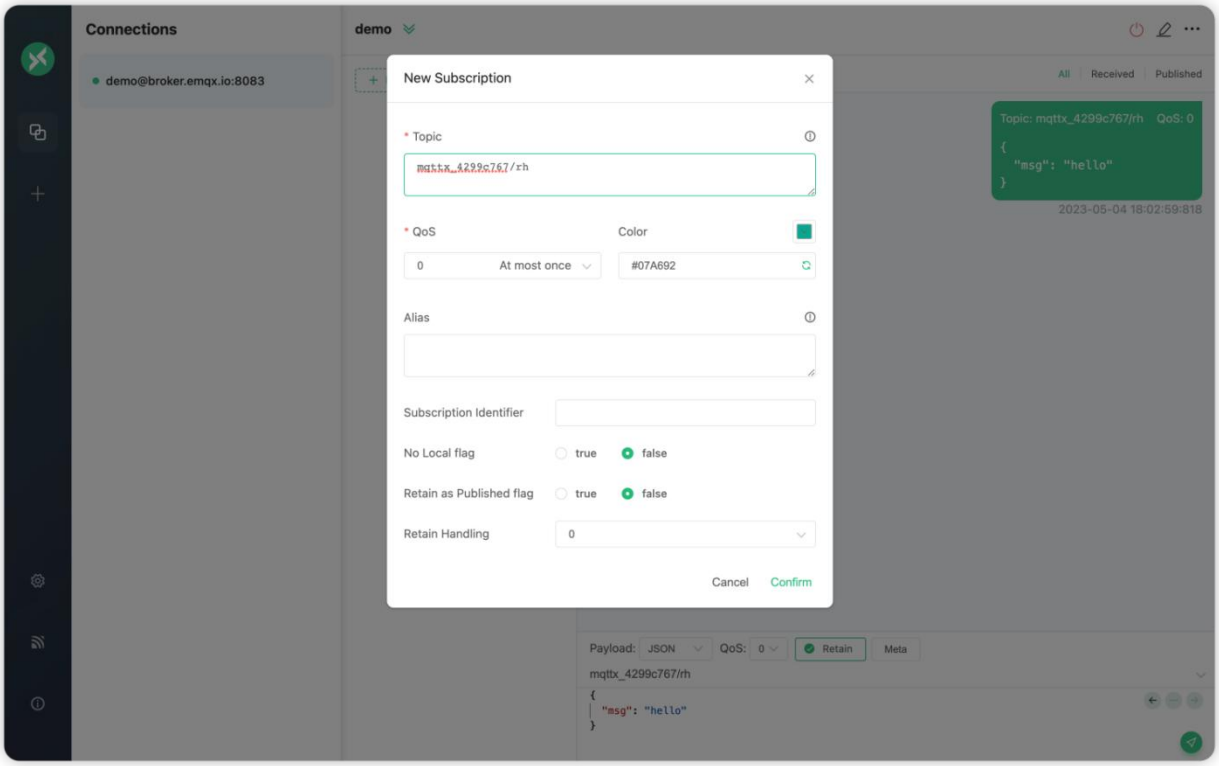



4. 订阅成功后，我们分别向主题 `mqttx_4299c767/rap0` 和 `mqttx_4299c767/rap1` 发布一条保留消息，我们将看到前者收到的消息中 Retain 标识被清除，而后者收到的消息中 Retain 标识被保留：

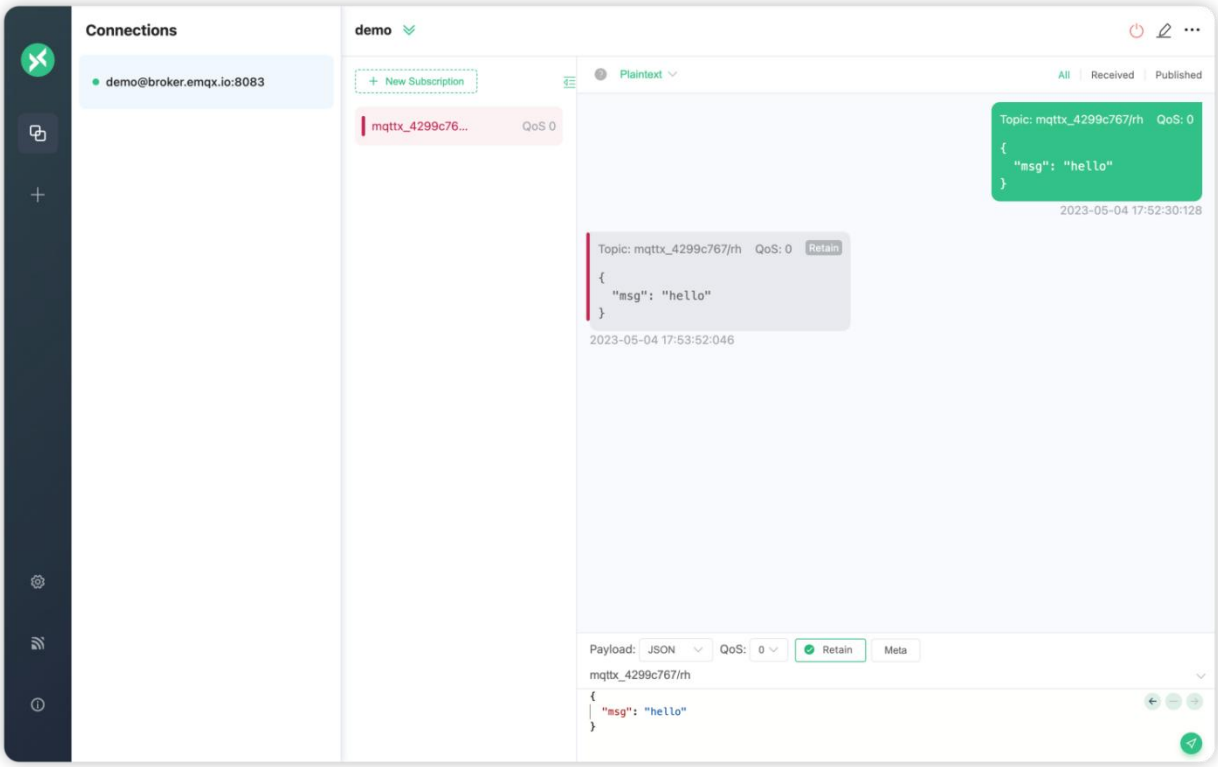



## 订阅选项 Retain Handling 的演示

1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个使用 WebSocket 的 MQTT 连接，并且连接免费的 公共 MQTT 服务器。
3. 连接成功后，我们先向主题 `mqttx_4299c767/rh` 发布一条保留消息。然后订阅主题 `mqttx_4299c767/rh`，并且将 Retain Handling 设置为 0：

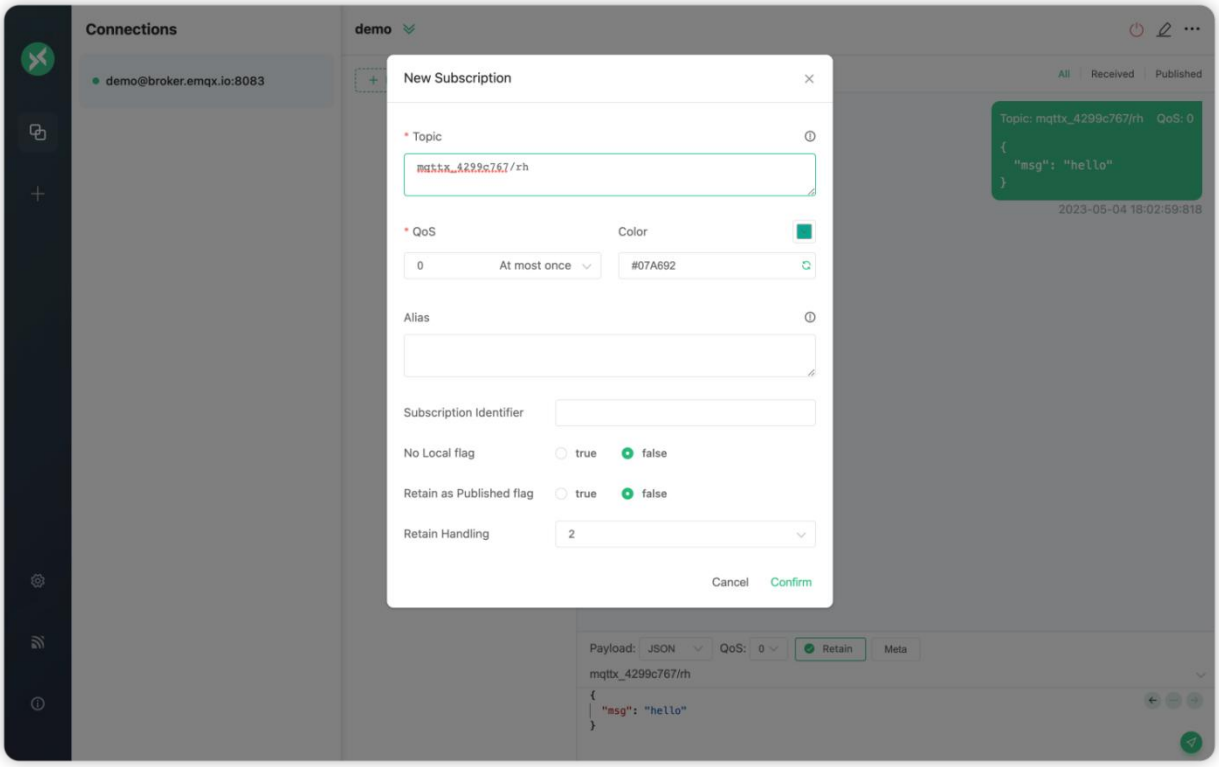



4. 订阅成功后，我们将收到服务端发送的保留消息：




5. 取消当前订阅，重新订阅主题 **mqttx_4299c767/rh**，并且将 Retain Handling 设置为 2。不过这一次订

阅成功后，我们将不会收到服务端发送的保留消息：




在 MQTTX 中，我们没有办法演示 Retain Handling 设置为 1 时的效果。不过你可以在 这里 获取订阅选项的 Python 示例代码。

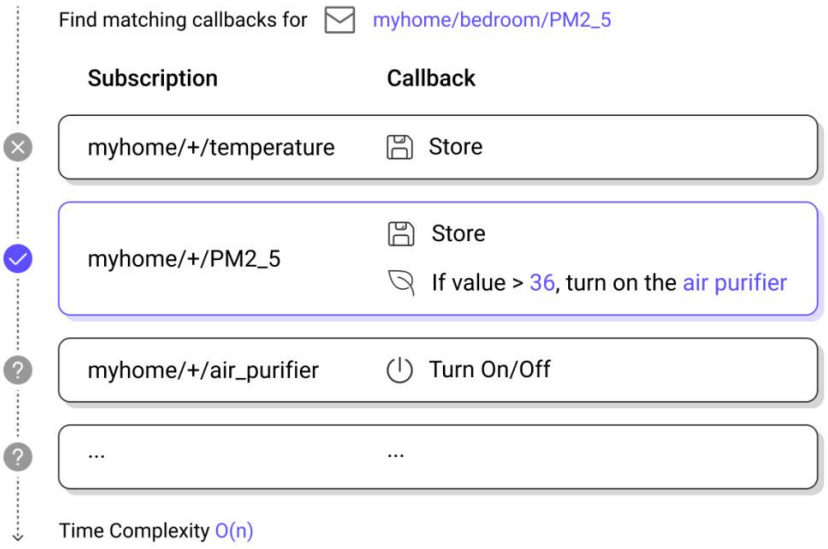
# 订阅标识符

## 为什么需要订阅标识符

在大部分 MQTT 客户端的实现中，都会通过回调机制来实现对新到达消息的处理。

但是在回调函数中，我们只能知道消息的主题名是什么。如果是非通配符订阅，订阅时使用的主题过滤器将和消息中的主题名完全一致，所以我们可以直接建立订阅主题与回调函数的映射关系。然后在消息到达时，根据消息中的主题名查找并执行对应的回调函数。

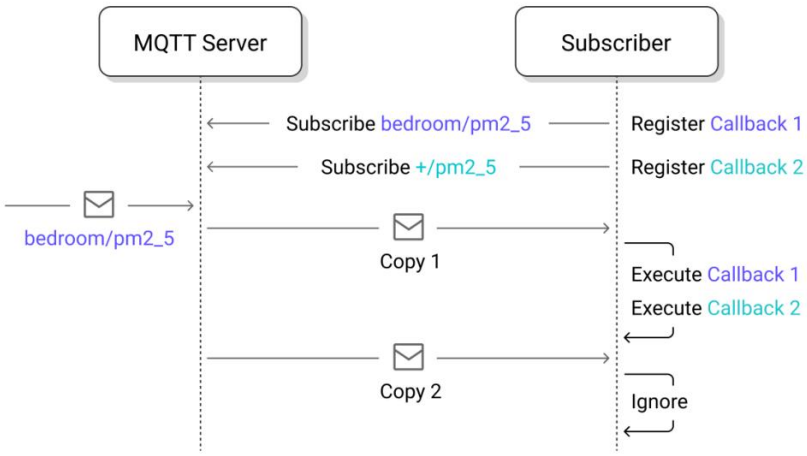
但如果是通配符订阅，消息中的主题名和订阅时的主题过滤器将是两个不同的字符串，我们只有将消息中的主题名与原始的订阅挨个进行主题匹配，才能确定应该执行哪个回调函数。这显然极大地影响了客户端的处理效率。




另外，因为 MQTT 允许一个客户端建立多个订阅，那么当客户端使用通配符订阅时，一条消息可能同时与一个客户端的多个订阅匹配。

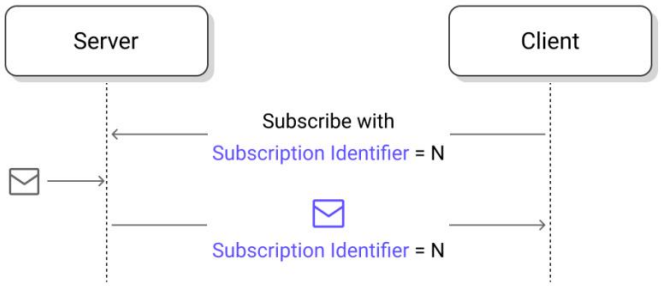
MQTT 允许服务端为这些重叠的订阅分别发送一次消息，也允许服务端只为这些重叠的订阅发送一条消息。

当服务端采用前一种实现时，客户端必须额外对这些消息进行去重才能保证回调不会被重复执行：



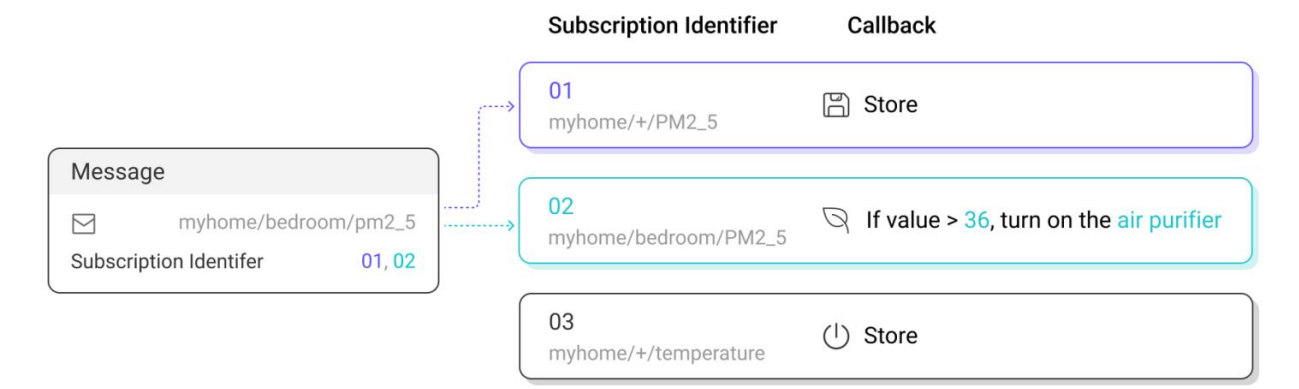

## 订阅标识符的工作原理

为了解决这个问题，MQTT 5.0 引入了订阅标识符。它的用法非常简单，客户端可以在订阅时指定一个订阅标识符，服务端则需要存储该订阅与订阅标识符的映射关系。当有匹配该订阅的 PUBLISH 报文要转发给此客户端时，服务端会将与该订阅关联的订阅标识符随 PUBLISH 报文一并返回给客户端。




如果服务端选择为重叠的订阅分别发送一次消息，那么每个 PUBLISH 报文都应该包含与订阅相匹配的订阅标识符，而如果服务端选择为重叠的订阅只发送一条消息，那么 PUBLISH 报文将包含多个订阅标识符。

客户端只需要建立订阅标识符与回调函数的映射，就可以通过消息中的订阅标识符得知这个消息来自哪个订阅，以及应该执行哪个回调函数。

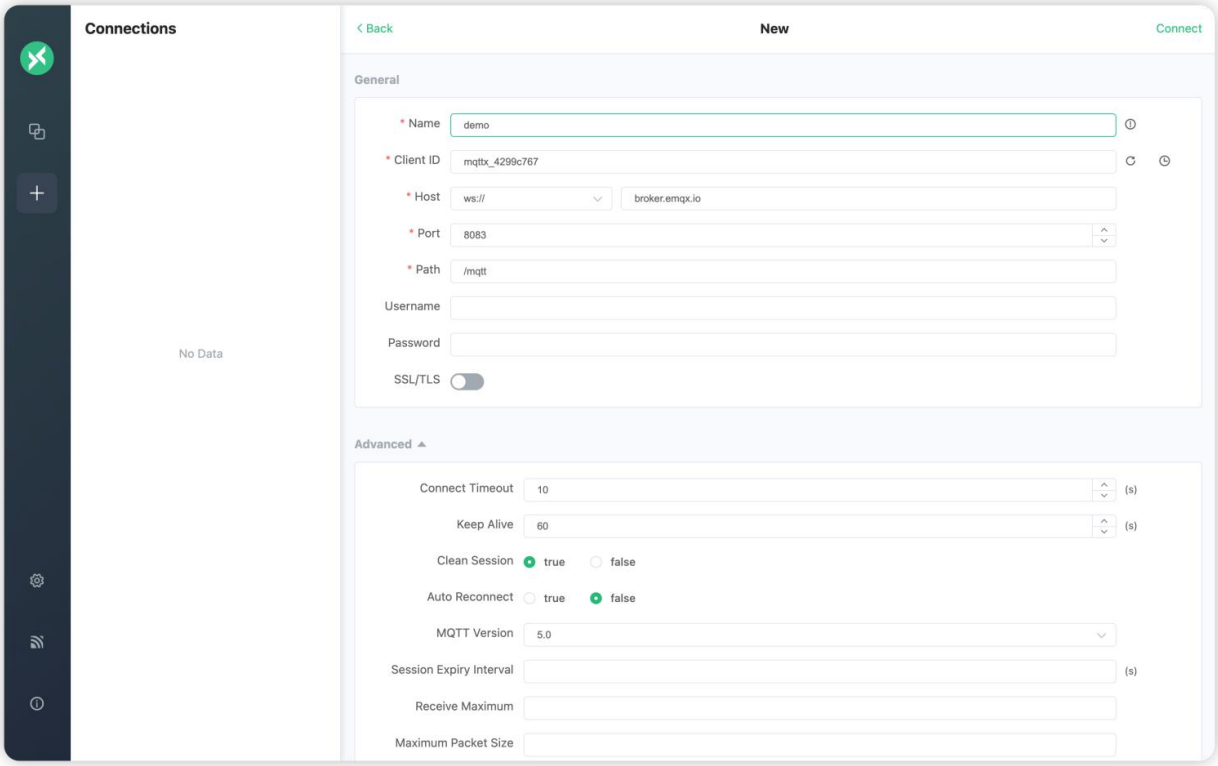



在客户端中，订阅标识符并不属于会话状态的一部分，将订阅标识符和什么内容进行关联，完全由客户端决定。所以除了回调函数，我们也可以建立订阅标识符与订阅主题的映射，或者建立与 Client ID 的映射。后者在转发服务端消息给客户端的网关中非常有用。当消息从服务端到达网关，网关只要根据订阅标识符就能够知道应该将消息转发给哪个客户端，而不需要重新做一次主题的匹配和路由。

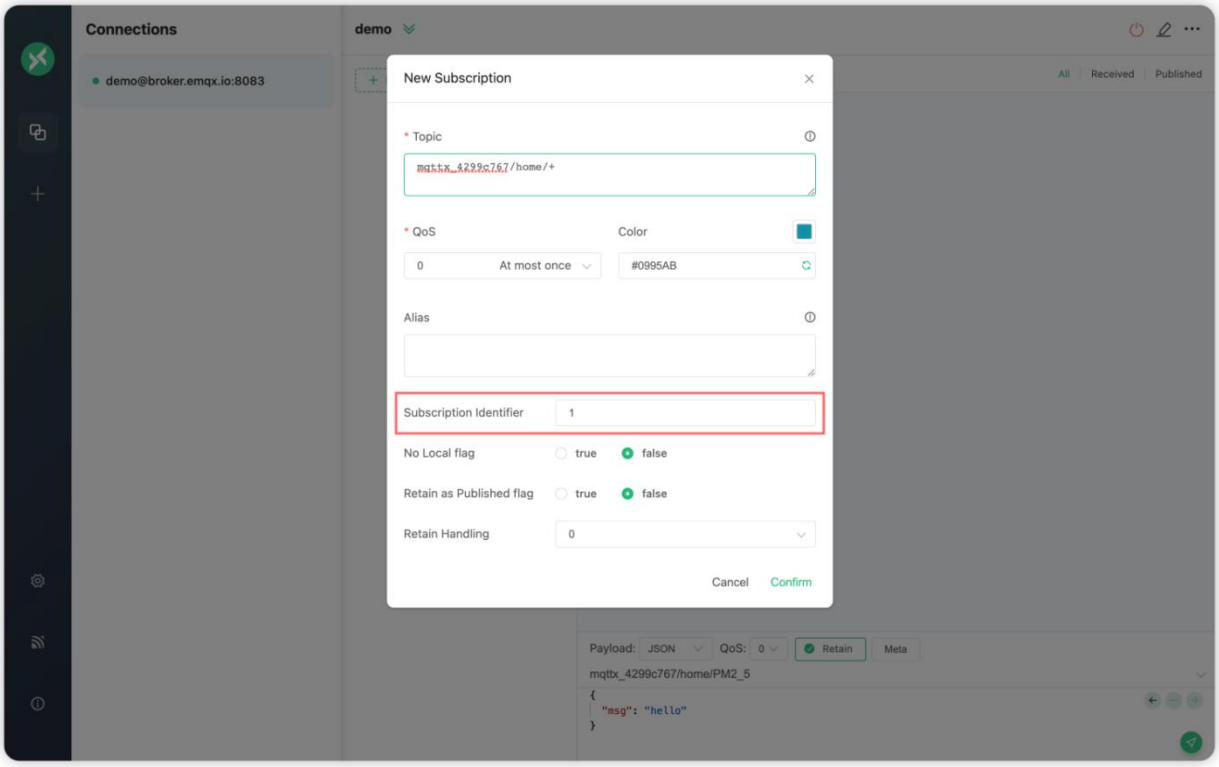
一个订阅报文只能包含一个订阅标识符，如果一个订阅报文中有多个订阅请求，那么这个订阅标识符将同时和这些订阅相关联。所以请尽量确保将多个订阅关联至同一个回调是您有意为之的。

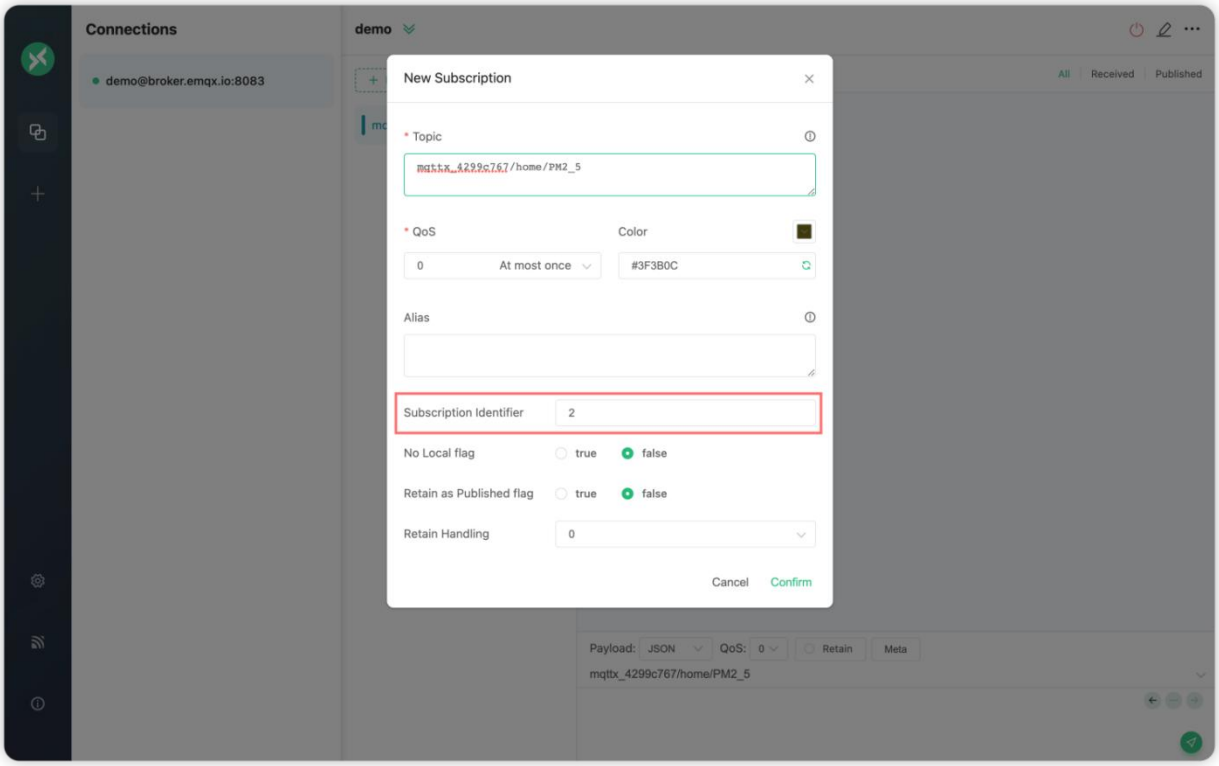
## 如何使用订阅标识符

1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个使用 WebSocket 的 MQTT 连接，并且连接免费的 公共 MQTT 服务器：

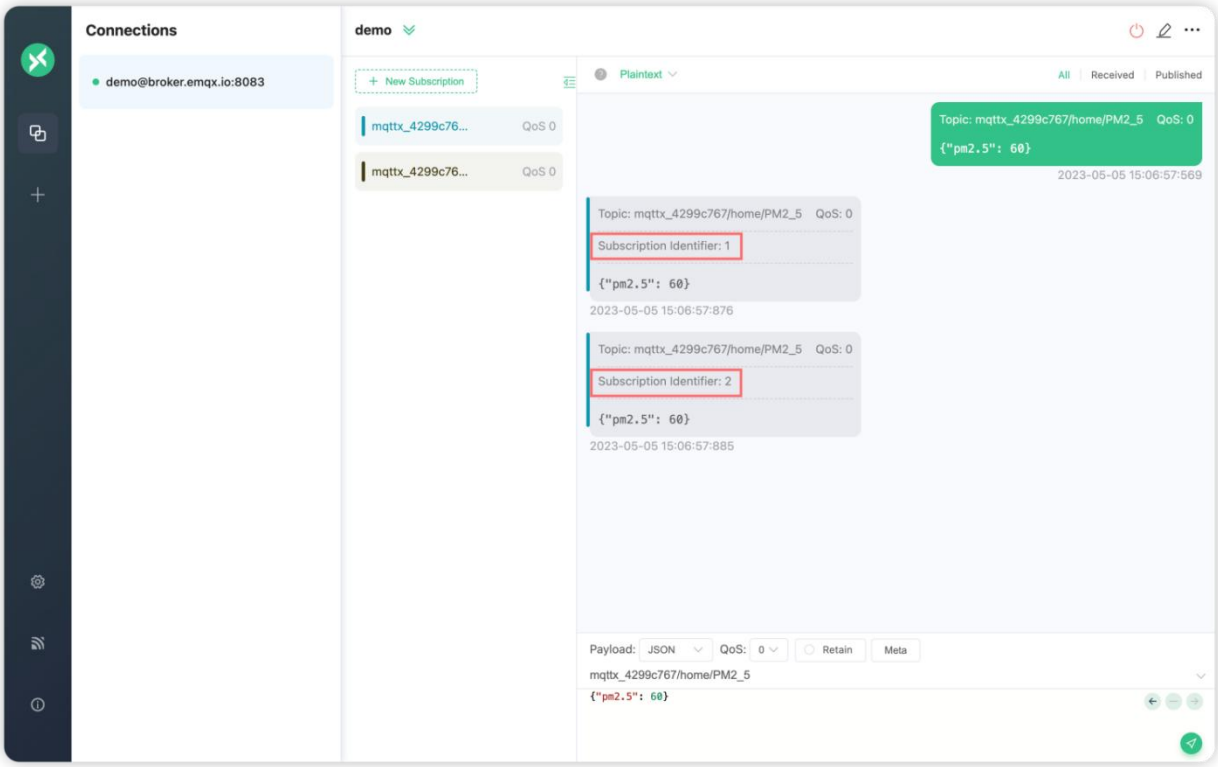



3. 连接成功后，我们先订阅主题 **mqttx_4299c767/home/+**，并指定 Subscription Identifier 为 1，然后订阅主题 **mqttx_4299c767/home/PM2_5**，并指定 Subscription Identifier 为 2。由于公共服务器可能同时被很多人使用，为了避免主题与别人重复，这里我们将 Client ID 作为主题前缀：

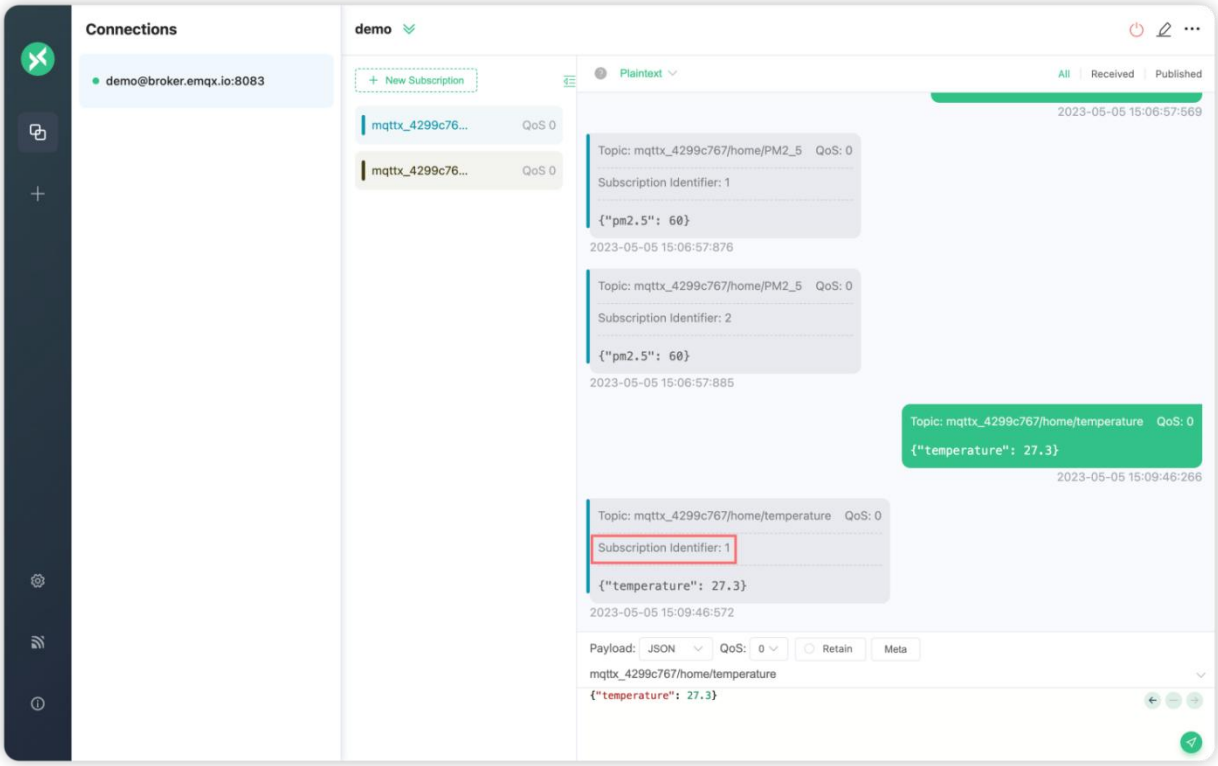





4. 订阅成功后，我们向主题 **mqttx_4299c767/home/PM2_5** 发布一条消息。我们将看到当前客户端收到了两条消息，消息中的 Subscription Identifier 分别为 1 和 2。这是因为 EMQX 的实现是为重叠的订阅分别发送一条消息：



5. 而如果我们向主题 **mqttx_4299c767/home/temperature** 发布一条消息，我们将看到收到消息中的 Subscription Identifier 为 1：




到这里，我们通过 MQTTX 演示了如何为订阅设置 Subscription Identifier。如果你仍然好奇如何根据 Subscription Identifier 来触发不同的回调，可以在 这里 获取 Subscription Identifier 的 Python 示例代码。

# 消息过期间隔

## 什么是消息过期间隔?

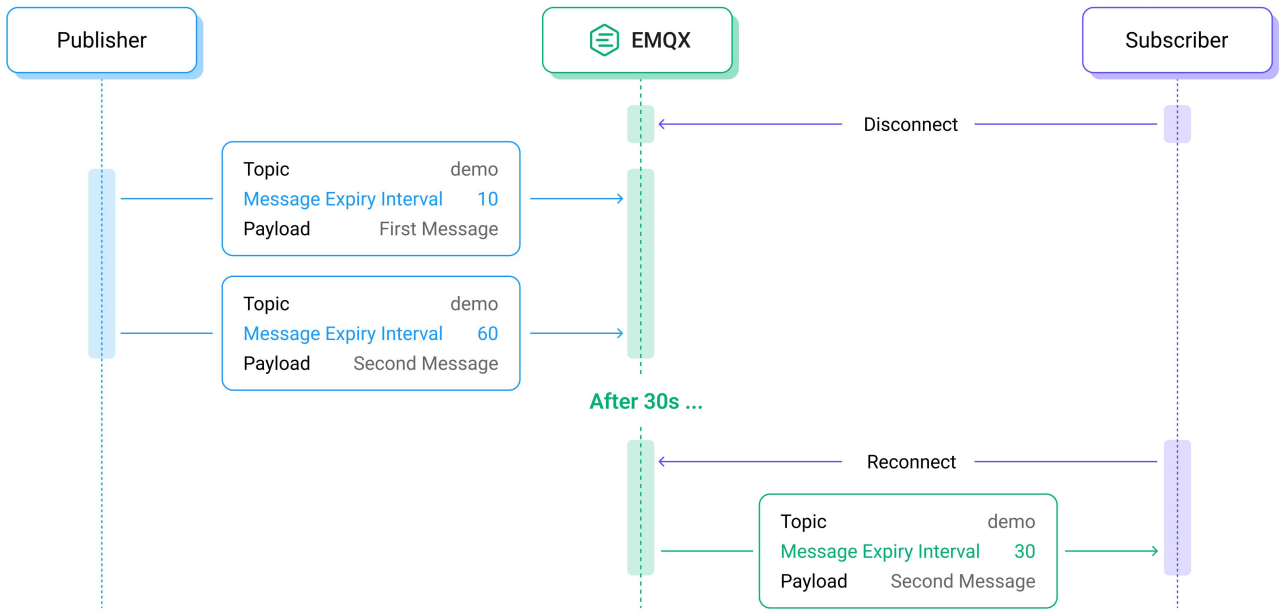
消息过期间隔是 MQTT 5.0 引入的一个新特性，它允许发布端为有时效性的消息设置一个过期间隔，如果该消息在服务端中停留超过了这个指定的间隔，那么服务端将不会再将它分发给订阅端。默认情况下，消息中不会包含消息过期间隔，这表示该消息永远不会过期。

MQTT 的持久会话可以为离线客户端缓存尚未发送的消息，然后在客户端恢复连接时发送。但如果客户端离线时间较长，可能有一些寿命较短的消息已经没有必要必须发送给客户端了，继续发送这些过期的消息，只会浪费网络带宽和客户端资源。

以联网汽车为例，我们可以向车辆发送建议车速使它能够在绿灯期间通过路口，这类消息通常仅在车辆到达下一个路口之前有效，生命周期非常短暂。而前方拥堵提醒这类消息的生命周期则会更长一些，一般会在半小时到 1 小时内有效。

如果客户端在发布消息时设置了过期间隔，那么服务端在转发这个消息时也会包含过期间隔，但过期间隔的值会被更新为服务端接收到的值减去该消息在服务端停留的时间。

这可以避免消息的时效性在传递的过程中丢失，特别是在桥接到另一个 MQTT 服务器的时候。



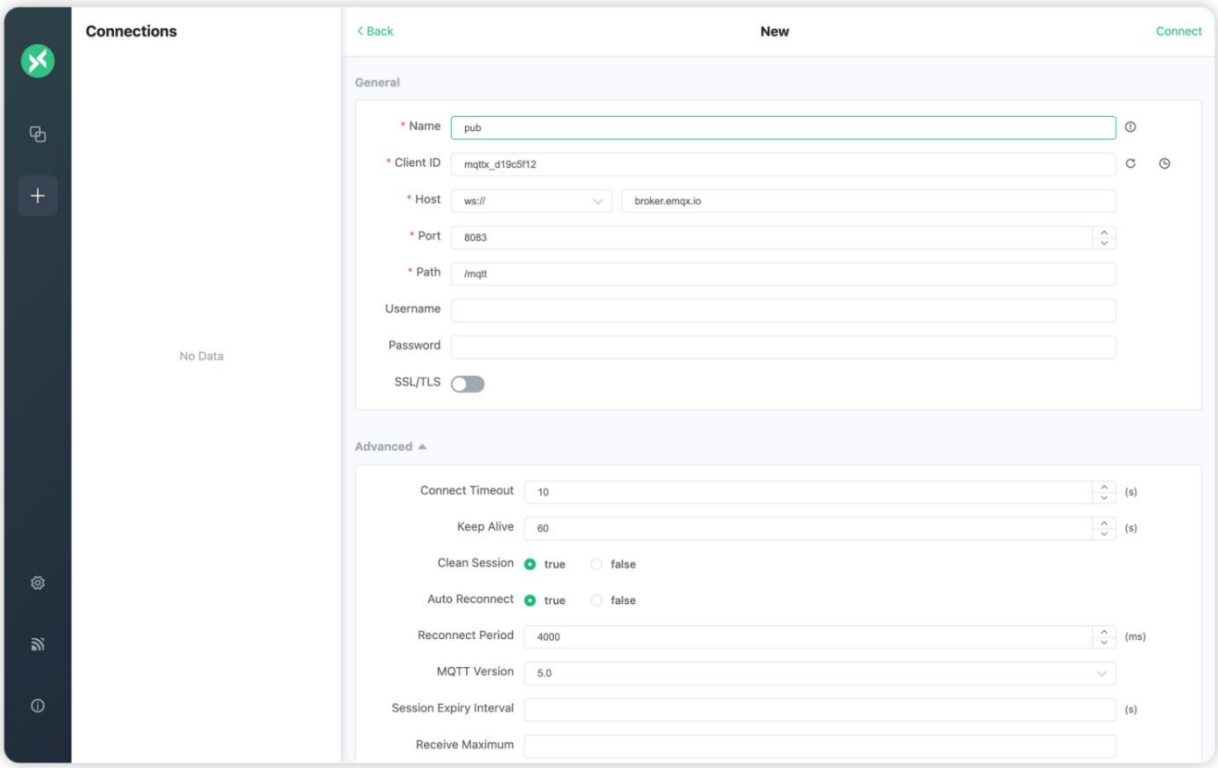
## 何时使用消息过期间隔?

消息过期间隔非常适合在以下场景下使用:

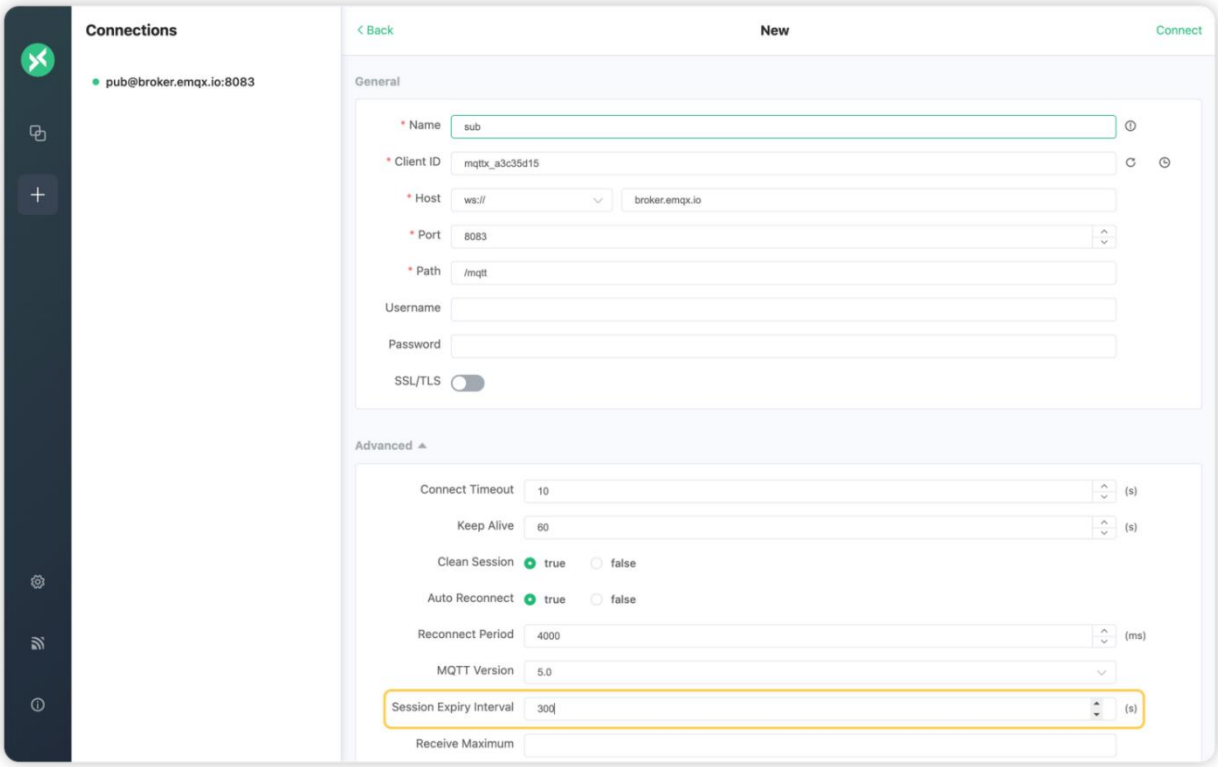
1. 与时间强绑定的消息。比如优惠还剩最后两小时这个消息，如果用户在两个小时后才收到它，不会有任何的意义。
2. 周期性告知最新状态的消息。仍然以道路拥堵提醒为例，我们需要周期性向车辆发送拥堵的预计结束时间，这个时间会随最新的道路情况而发生变化。所以当最新的消息到达后，之前还未发送的消息也没有必要继续发送了。此时消息的过期间隔将由我们实际的发送周期决定。
3. 保留消息。相比于需要再次发送 Payload 为空的保留消息来清除对应主题下的保留消息，为其设置过期时间然后由服务器自动删除显然更加方便，这也可以有效避免保留消息占用过多的存储资源。

## 演示

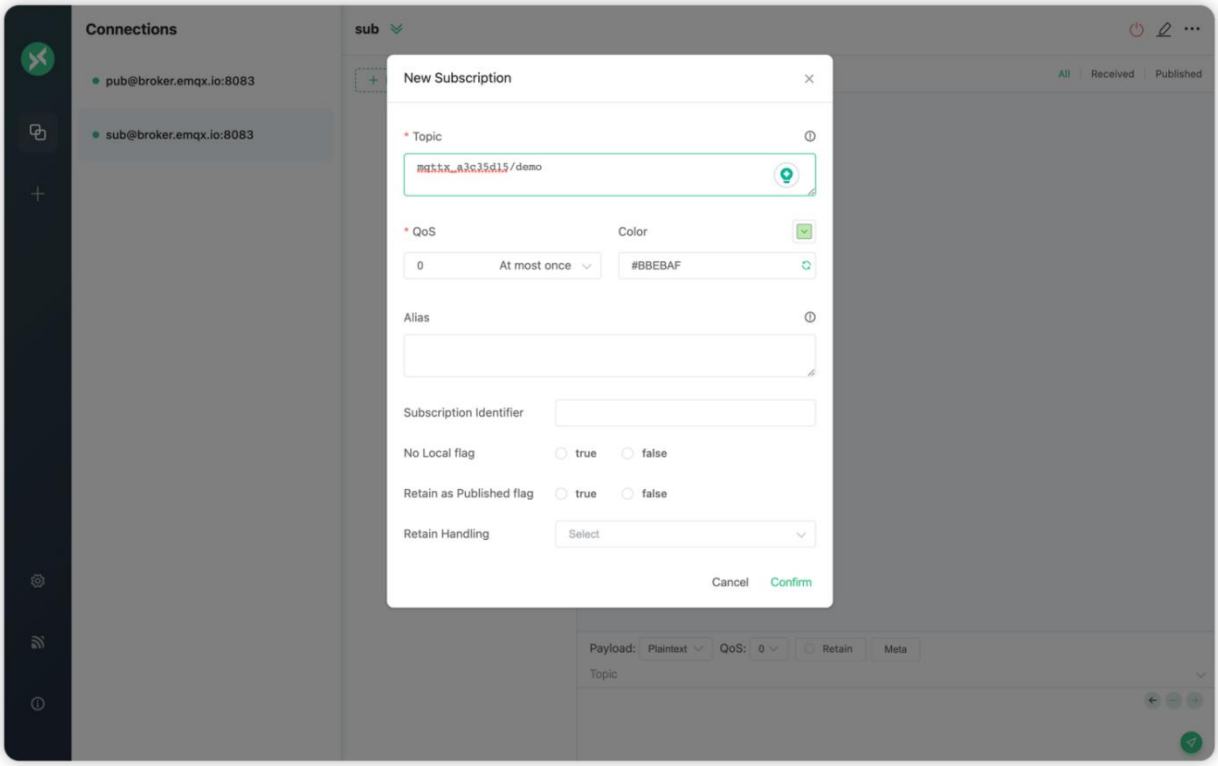
1. 在 Web 浏览器上访问 MQTTX Web。
2. 创建一个名为 `pub` 的客户端连接用于发布消息，并且连接到免费的 公共 MQTT 服务器:



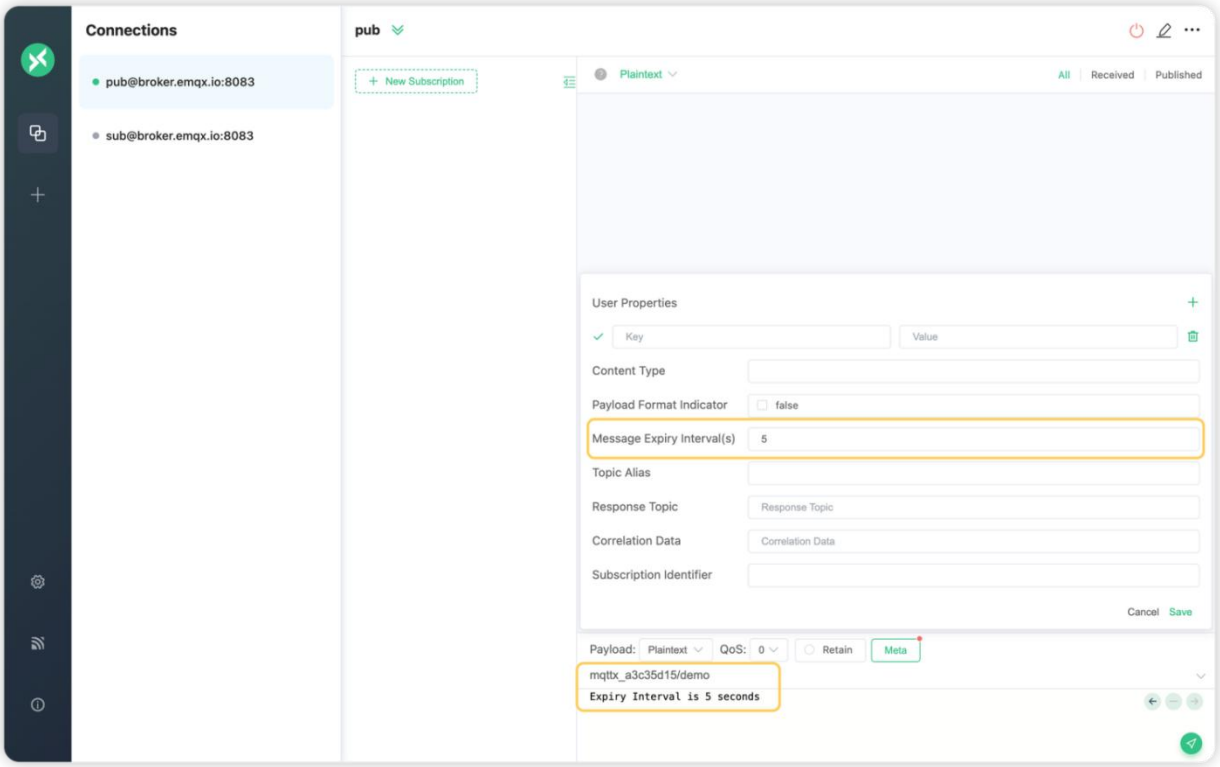
3. 新建一个名为 **sub** 的客户端连接用于订阅，并将 Session Expiry Interval 设置为 300 秒表示这将是一个持久会话：

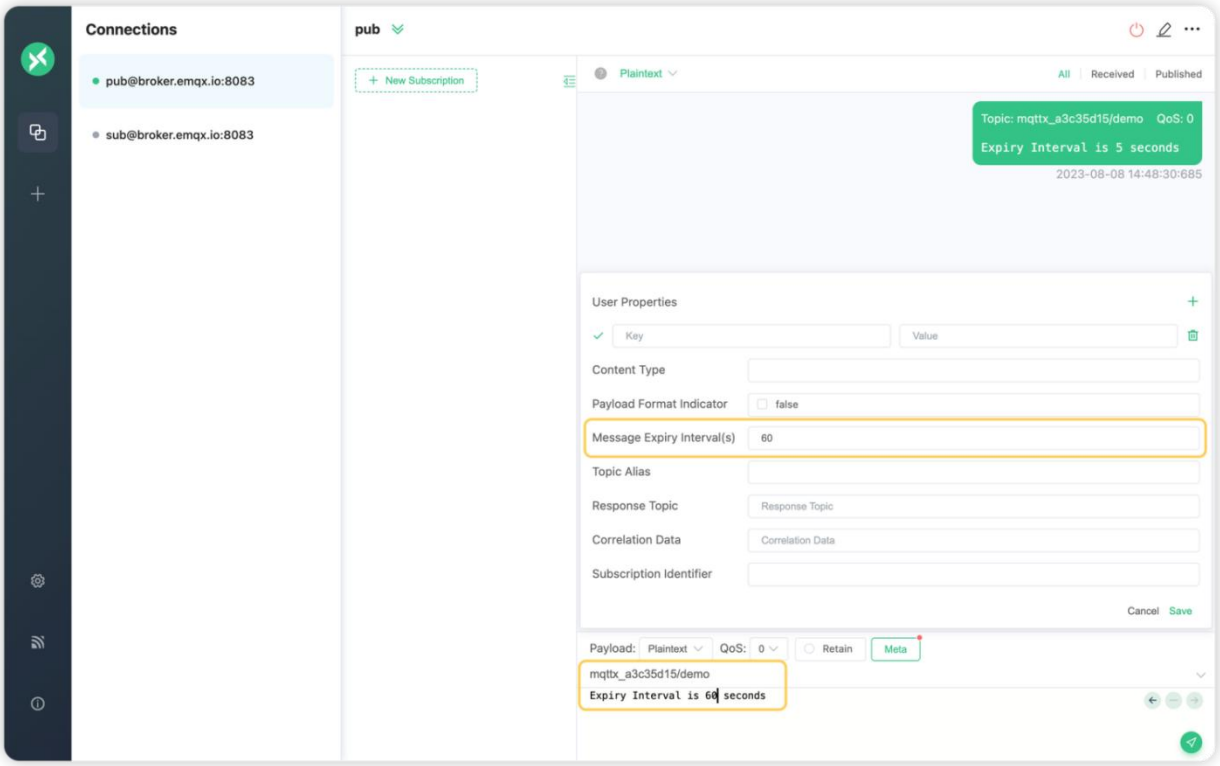



4. 连接成功后，我们订阅主题 **mqttx_a3c35d15/demo**，使用 Client ID 作为主题前缀可以有效避免与公共服务器中其他客户端使用的主题重复：



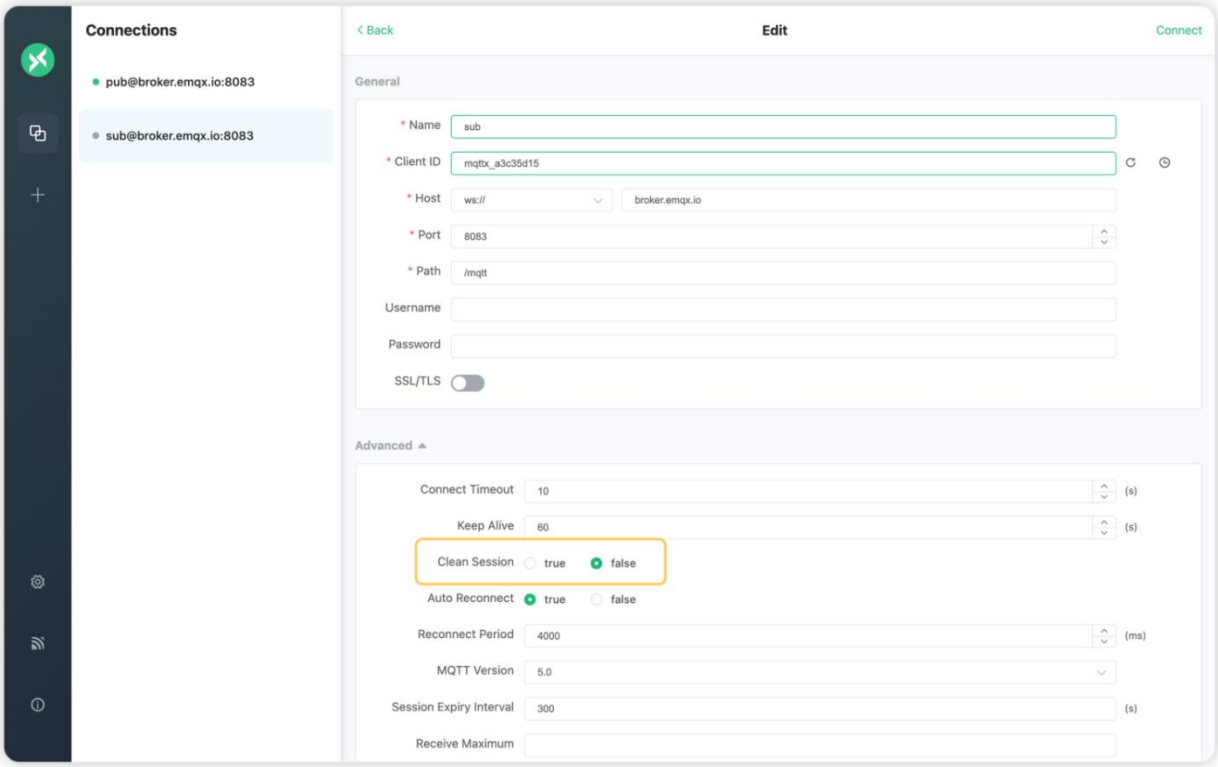
5. 订阅成功后，我们断开 **sub** 客户端与服务器的连接，然后切换到 **pub** 客户端，向主题 **mqttx_a3c35d15/demo** 发布以下两条 Message Expiry Interval 分别为 5 秒和 60 秒的消息：

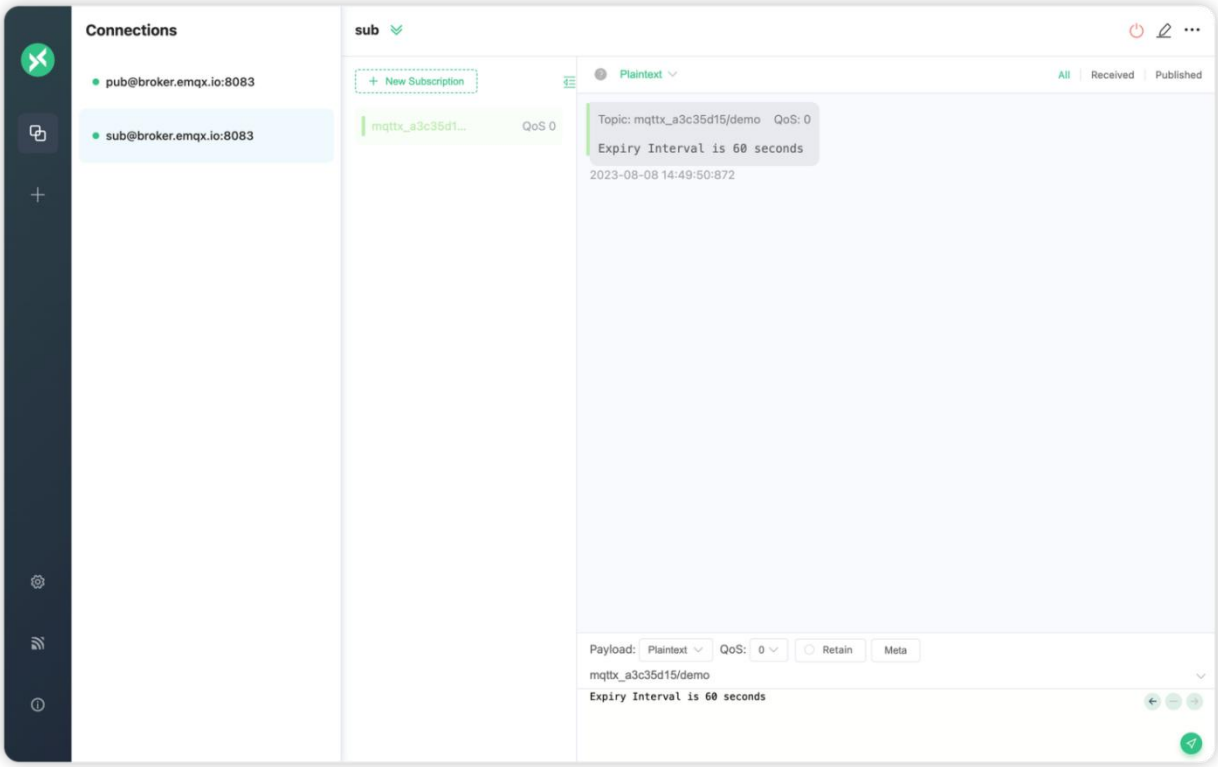



6. 发布完成后，切换到 sub 客户端，将 Clean Session 设置为 false 表示想要恢复之前的会话，然后等待至少 5 秒再重新连接。我们将看到 sub 只收到了过期时间为 60 秒的消息，因为此时另一条消

息已经过期：




以上就是 Message Expiry Interval 的用法与效果，你还可以在 这里 获取 Message Expiry Interval 的 Python 示例代码。

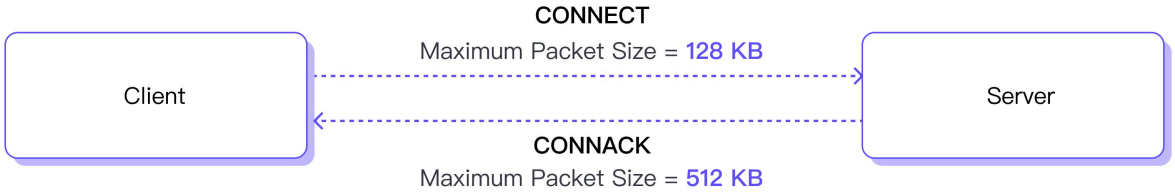
# 最大报文大小

## 什么是最大报文大小（Maximum Packet Size）

MQTT 报文的理论最大长度为 268435456 字节，也就是 256 MB。但显然，不仅仅是资源受限的客户端，一些作为边缘网关运行的 MQTT 服务端，可能也无法处理这一长度的报文。

考虑到不同客户端对报文的处理能力可能有着较大差异，发送过大的报文不仅可能影响对端的正常业务处理，甚至可能直接压垮对端。所以，我们需要使用 Maximum Packet Size 属性来协商客户端和服务端各自能够处理的最大报文长度。

客户端首先在 CONNECT 报文中通过 Maximum Packet Size 来指定允许服务端给自己发送的报文的最大长度，而服务端则会在 CONNACK 报文中同样通过 Maximum Packet Size 来指定允许客户端给自己发送的报文的最大长度。




一旦连接建立，双方就必须遵循这一约定来发送消息。任何一方都不允许发送超过约定长度限制的报文，否则接收方就会返回 Reason Code 为 0x95 的 DISCONNECT 报文然后关闭网络连接。

但需要注意，如果客户端在 CONNECT 报文中设置了遗嘱消息，这可能在其不自知的情况下使得 CONNECT 报文超过了服务端允许的最大报文长度。这时服务端将返回 Reason Code 为 0x95 的 CONNACK 报文然后关闭网络连接。

## 发送方如何在 Maximum Packet Size 限制下工作

对客户端来说，不管是发布还是订阅，作为主动发送的一方，它都可以将一个报文拆分多个发送来避免超过长度限制。

但对服务端来说，它只负责转发消息，并不能决定消息的大小。所以如果它发现准备转发的消息的大小超

过了客户端能够接受的最大值，那么它只能丢弃这个消息。如果当前是共享订阅，那么服务端除了丢弃以外，还可以选择把消息发送给组内其他能够接收此消息的客户端。

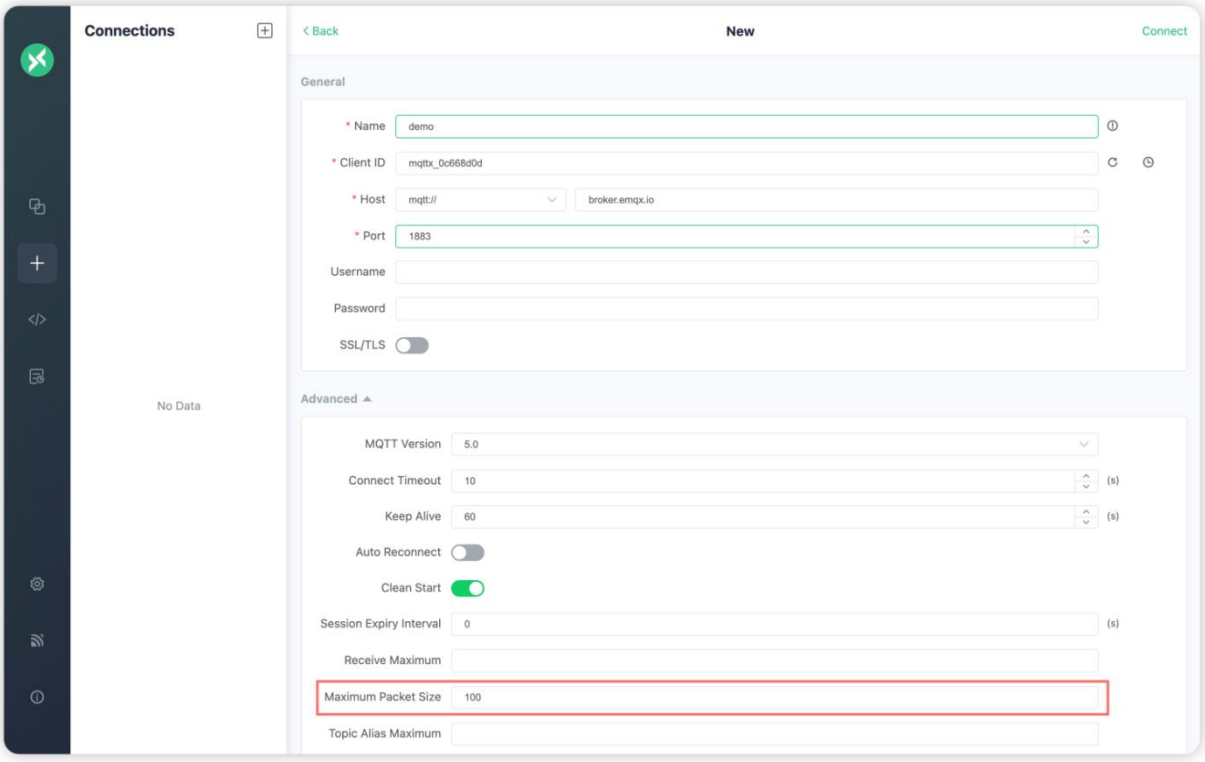
除了上面提到的两种策略，不管是客户端还是服务端，他们都可以在一定程度上裁剪报文的内容来减少长度。我们知道响应方可以在 CONNACK、PUBACK 这些响应报文中包含 User Property 和 Reason String 这两个属性来向对端传递更多信息。

但响应报文超过最大长度限制的可能性也正是由这两个属性带来的。显然，传输它们的优先级低于确保协议流程正常进行，所以响应方可以在报文长度超出限制时，从报文中移除这两个属性来尽可能保证响应报文被正常发送。

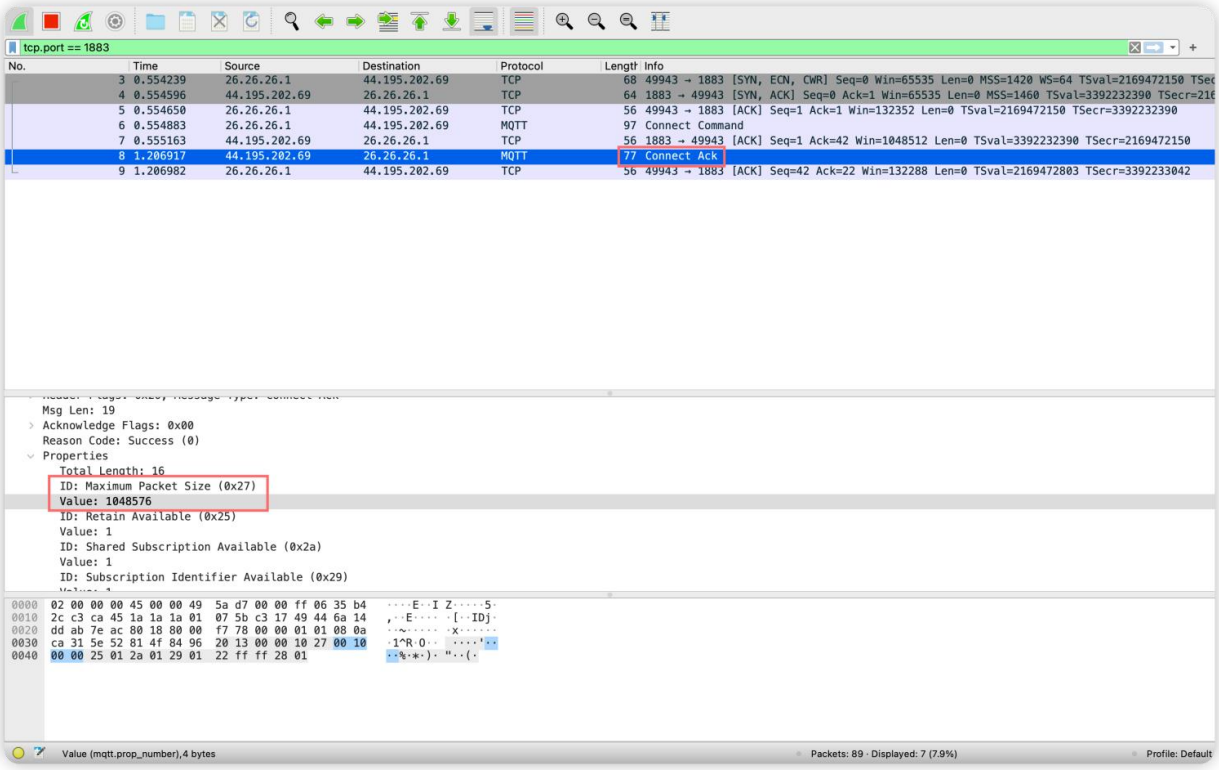
需要注意的是，这里仅仅指响应报文，PUBLISH 报文不在此列。对于 PUBLISH 报文来说，User Property 属于消息的一部分，服务端不能为了确保消息投递而尝试将它从报文中移除。

## 使用示例

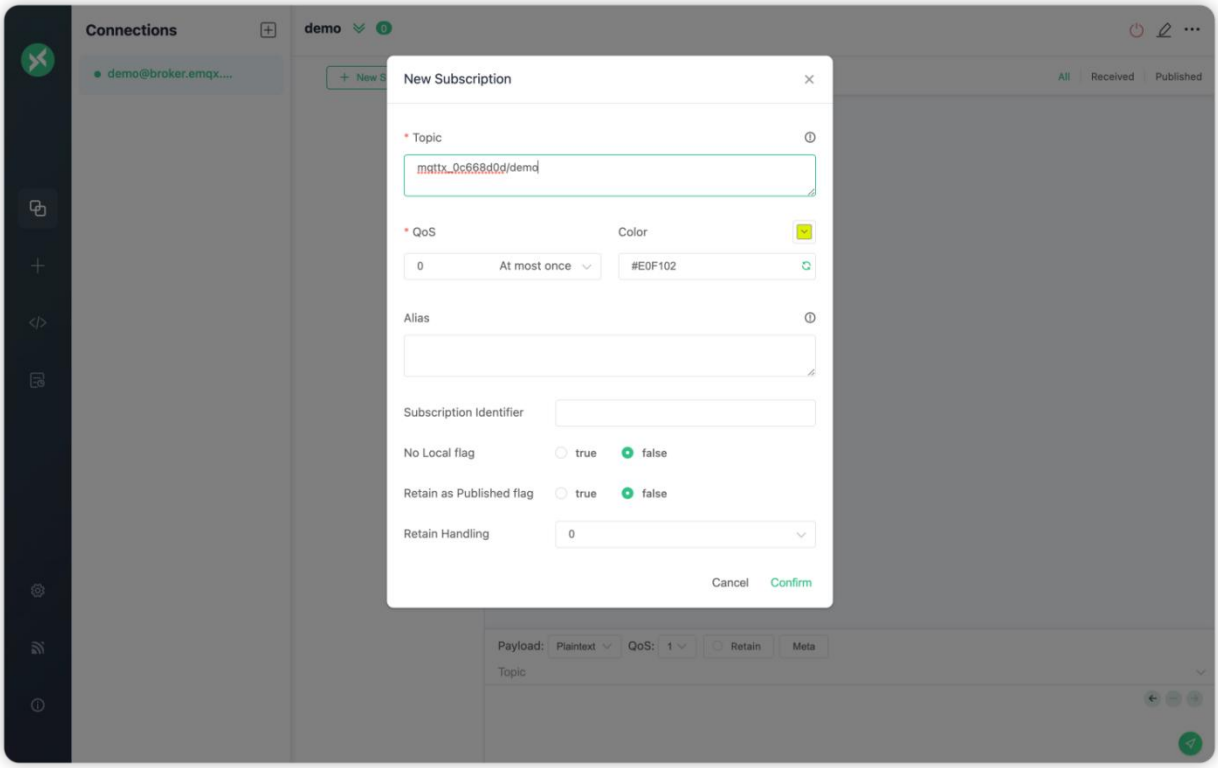
1. 打开已经安装在本地的 MQTTX
2. 创建一个 MQTT 连接，设置 Maximum Packet Size 为 100，然后连接至免费的 公共 MQTT 服务器：



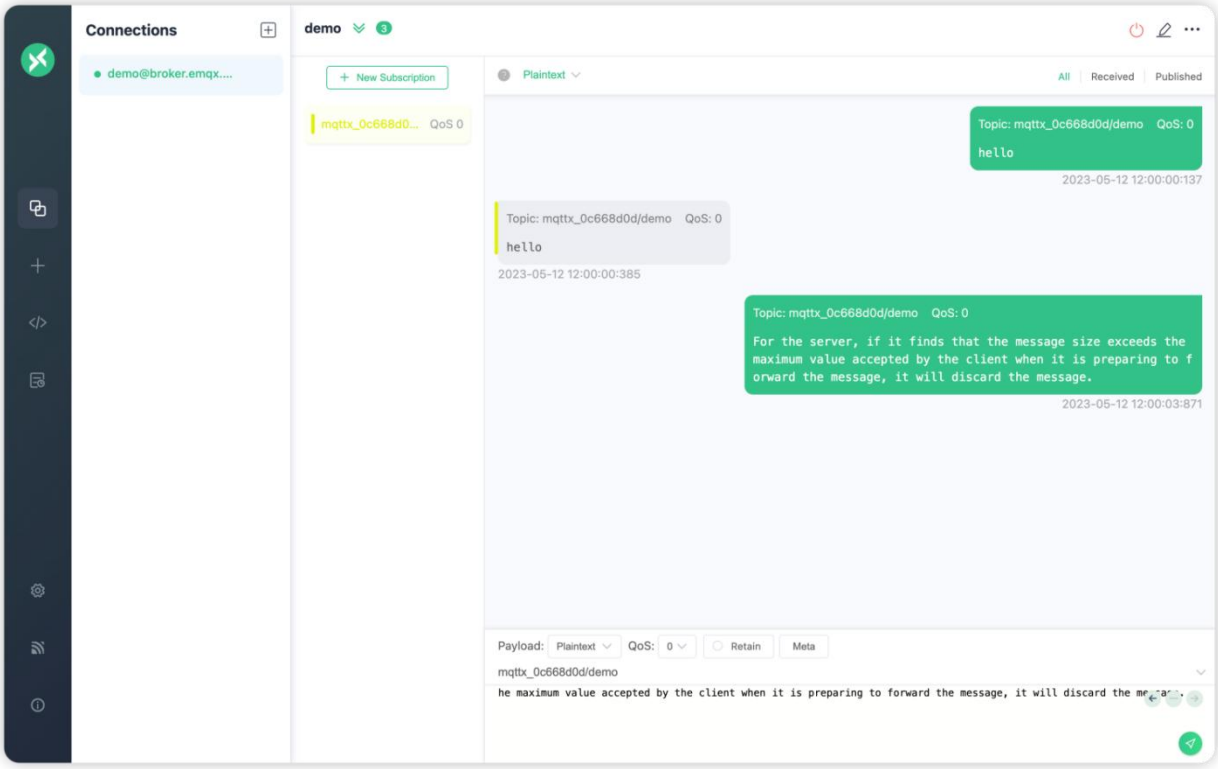
3. 连接成功后我们可以通过 Wireshark 抓包工具看到，服务端返回的 CONNACK 报文中 Maximum Packet Size 属性的值为 1048576，也就是说客户端每次只能向公共 MQTT 服务器发送最多 1 KB 的报文：




4. 回到 MQTTX 中，我们订阅 `mqttx_0c668d0d/demo` 主题：



5. 然后向 `mqttx_0c668d0d/demo` 主题分别发布长度为 5 字节和 172 字节的两条消息，我们将看到最终只会收到长度为 5 字节的消息，另一条超过 100 字节长度限制的消息没有被服务端转发：



# 原因码（Reason Code）

Reason Code 在 MQTT 中的主要作用是为客户端和服务端提供更详细的反馈。比如我们可以在 CONNACK 报文中将用户名或密码错误对应的 Reason Code 反馈给客户端，这样客户端就能够知道自己无法连接的原因。

## MQTT 3.1.1 中的 Reason Code

虽然 MQTT 3.1.1 就已经支持了 Reason Code，但它并没有定义太多可用的 Reason Code。

在仅有的两个支持 Reason Code 的报文中，CONNACK 报文只有 5 个用于指示失败的 Reason Code，SUBACK 报文则仅仅只有一个用于指示失败的 Reason Code，无法进一步指示订阅失败的原因。而对于发布、取消订阅这些不支持 Reason Code 的操作，我们更是连操作是否成功都无法知晓。这不仅对于开发调试非常地不友好，也不利于实现健壮的代码。

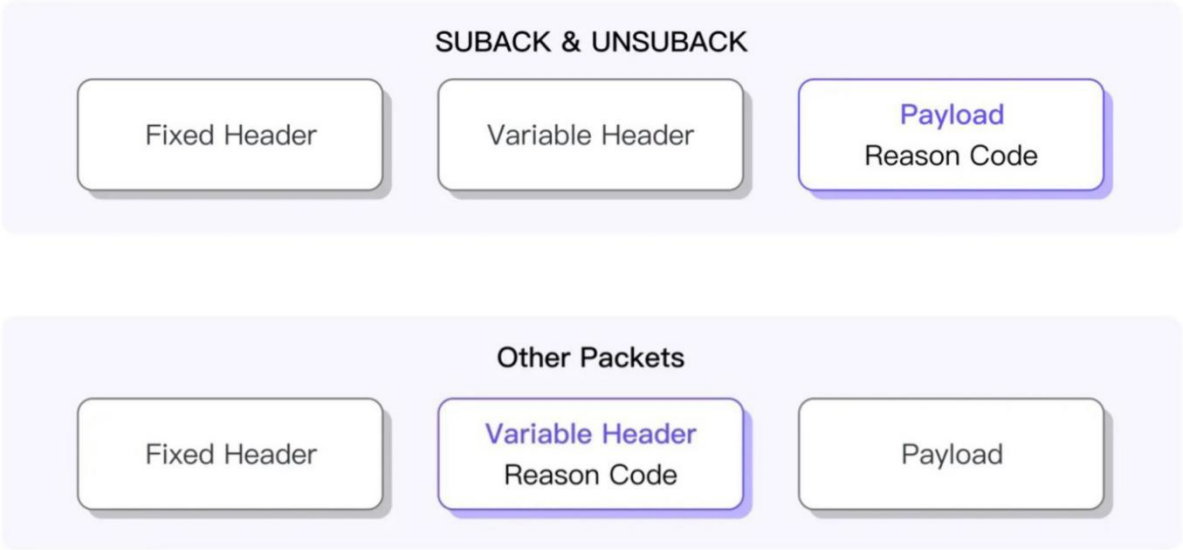
## MQTT 5.0 中的 Reason Code

所以在 MQTT 5.0 中，可用的 Reason Code 被扩充到了 43 个，并且规定了小于 0x80 的 Reason Code 用于表示成功，大于等于 0x80 的 Reason Code 则用于表示失败。而不再像 MQTT 3.1.1 那样，小于 0x80 的 Reason Code 也可能表示失败。这使得客户端能够更轻松地判断操作是否成功。

另外 MQTT 5.0 中支持 Reason Code 的报文也扩展到了：CONNACK、PUBACK、PUBREC、PUBREL、PUBCOMP、SUBACK、UNSUBACK、DISCONNECT 以及 AUTH。现在，我们不仅可以知道消息发布是否成功，还可以知道失败的原因，例如当前不存在匹配的订阅者、或者无权向这个主题发布消息等等。

| | Reason Codes in MQTT 3.1.1 | Reason Codes in MQTT 5.0 |
|---|---|---|
| CONNACK | ✓ | ✓ |
| DISCONNECT | ✗ | ✓ |
| PUBACK | ✗ | ✓ |
| PUBREC | ✗ | ✓ |
| PUBREL | ✗ | ✓ |
| PUBCOMP | ✗ | ✓ |
| SUBACK | ✓ | ✓ |
| UNSUBACK | ✗ | ✓ |
| AUTH | — | ✓ |

大部分报文都只会包含一个 Reason Code，除了 SUBACK 和 UNSUBACK。这是因为 SUBSCRIBE 和 UNSUBSCRIBE 报文可以包含多个主题过滤器，而每个主题过滤器都必须有一个对应的 Reason Code 来指示其操作结果，所以 SUBACK 和 UNSUBACK 报文也需要能够包含多个 Reason Codes。这也是为什么其他报文中的 Reason Code 都位于可变报头，而 SUBACK 和 UNSUBACK 的 Reason Code 则位于载荷部分。




在本章节最后的 Reason Code 速查表中，我们详细地解释了 MQTT 5.0 每个 Reason Code 的含义和使用场景，您可以自行查阅。

### 向客户端指示连接断开的原因

在 MQTT 3.1 和 3.1.1 中，DISCONNECT 报文只能由客户端发布。所以当客户端违反某些限制时，服务端只能直接关闭网络连接，而无法向客户端传递更多信息，这导致调查连接断开原因变得困难重重。

而在 MQTT 5.0 中，服务端可以在关闭网络连接之前向客户端发送 DISCONNECT 报文，而客户端则可以通过 DISCONNECT 报文中的 Reason Code 了解连接被断开的原因，比如报文过大、服务器正忙等等。

## Reason String

Reason String 是 MQTT 5.0 对 Reason Code 的一个补充，它是一个为诊断而设计的人类可读的字符串。虽然 Reason Code 已经能够指示大部分的错误原因，但对于开发者或运维人员来说，可能仍然缺少更直观的上下文信息。

比如当服务端通过 Reason Code 向客户端指示主题过滤器不合法（0x8F）时，开发者仍然无从知晓具体的原因，是主题层级过多？还是包含了不被服务端接受的字符？而如果服务端可以返回一个内容类似于 "The number of topic levels exceeds the limit, the maximum is 10." 的 Reason String，那么开发者很快就能够知道原因并做出调整。

在实际使用中，Reason String 的内容取决于客户端和服务端的具体实现，所以一个实现正确的接收端不应该尝试解析 Reason String 的内容，推荐的使用方式包括但不限于在抛出异常时使用 Reason String，或者将它写入日志。

最后，Reason String 是一个可选的功能，是否会收到 Reason String 取决于对端是否支持。

> MQTT 5.0 Reason Code 速查表：
> https://www.emqx.com/zh/blog/mqtt5-new-features-reason-code-and-ack

# 增强认证

## 什么是增强认证

增强认证是 MQTT 5.0 新引入的认证机制。事实上，我们用认证框架来形容它更为适合，因为它允许我们套用各种比密码认证更加安全的身份验证方法。

不过更安全，另一方面则意味着更复杂，这类身份验证方法例如 SCRAM 通常都要求一次以上的认证数据往返。这导致由 CONNECT 与 CONNACK 报文提供的一次往返的认证框架变得不再适用，所以 MQTT 5.0 专门为此新增了 AUTH 报文，它能够支持任意次数的认证数据的往返。这使得我们可以将质询–响应风格的 SASL 机制引入到 MQTT 中。

## 增强认证解决了什么问题

在我们谈论这个问题之前，我们需要知道，为什么密码认证仍然不够安全?

事实上，即使我们已经使用加盐与哈希的方式来存储密码，尽可能提升了密码存储的安全性。但是为了完成认证，客户端不得不在网络中明文传输密码，这就使密码有了被泄漏的风险。即使我们使用 TLS 加密了通信，也仍有可能因为使用了较低的 SSL 版本、不够安全的密码套件、不合法的 CA 证书等等原因导致被攻击者窃取到密码这类敏感数据。

另外，简单的密码认证只能让服务端验证客户端的身份，却不能让客户端验证服务端的身份，这使得攻击者有机会冒充服务端来获取客户端发送的敏感数据。而这就是我们通常所说的中间人攻击。

而通过增强认证，我们可以选择使用 SASL 框架下的安全性更强的认证方法，它们有些可以避免在网络中传输密码，有些可以让客户端和服务端互相验证对方的身份，有些则两者皆备，这仍取决于我们最终选择的认证方法。

## 常见的可用于增强认证的 SASL 机制

### DIGEST-MD5

DIGEST-MD5 是在简单认证安全层（SASL）框架下的一种身份验证机制。它基于 MD5（Message Digest 5）散列算法，使用质询-响应机制来验证客户端和服务器之间的身份。它的优点在于客户端不需要在网络上传输明文密码。

简单来说，就是当客户端请求访问受保护资源时，服务端将返回一个 Challenge，其中包含了一次性的随机数和一些必要参数，客户端需要使用这些参数加上自己持有的用户名密码等数据，生成一个响应并返回给服务端，服务端将使用完全相同的方式生成期望的响应，然后与收到的响应进行比较，如果两者匹配，则身份验证通过。这免去了密码因为遭到网络窃听而泄漏的风险，并且由于连接时使用的是一次性的随机数，所以也增强了对重放攻击的防御能力。

但需要注意，DIGEST-MD5 只提供了服务端对客户端的身份验证，但没有提供客户端对服务端的身份验证，所以它并不能防止中间人攻击。另外，由于 MD5 目前已经不再安全，所以更推荐使用 SHA-256 这类抗碰撞能力更强的哈希函数来替代它。

### SCRAM

SCRAM 同样是 SASL 框架下的一种身份验证机制，它的核心思想与 DIGEST-MD5 类似，同样是使用一次性的随机数要求客户端生成响应，所以客户端同样无需在网络上传输明文密码。但与 DIGEST-MD5 不同的是，SCRAM 引入了盐值（Salt）和迭代次数（Iterations），并且使用了 SHA-256、SHA-512 这些更安全的哈希算法，这带来了更高的安全性，使 SCRAM 能够更加安全地存储密码，并且减少被离线攻击、重放攻击或其他攻击破解的风险。

另外，SCRAM 使用了更复杂的质询-响应流程，它增加了一个服务端向客户端发送证明的过程，客户端可以通过这个证明来确认服务端是否持有正确的密码，这就实现了客户端对服务端的身份验证，降低了中间人攻击的风险。

当然，SCRAM 使用的 SHA256 等哈希算法，也在性能上带来了一些额外的开销，这可能会对一些资源受限的设备造成一定的影响。

### Kerberos

Kerberos 引入了一个可信的第三方 Kerberos 服务器来提供身份验证服务，Kerberos 服务器向通过验证的用户授予票据，用户再使用票据访问资源服务器。这带来的一个好处是用户只要通过一次身份验证，就可以获得多个系统和服务的访问权限，即实现了单点登录（SSO）的功能。

Kerberos 服务器授予的票据的生命周期是有限的，客户端只能在有限时间的内使用这个票据访问服务，这可以避免因票据泄漏而导致的安全问题。当然，虽然较短的有效期可以有效地提高安全性，但在使用的便利性上可能不太友好，我们需要自行权衡这两者。

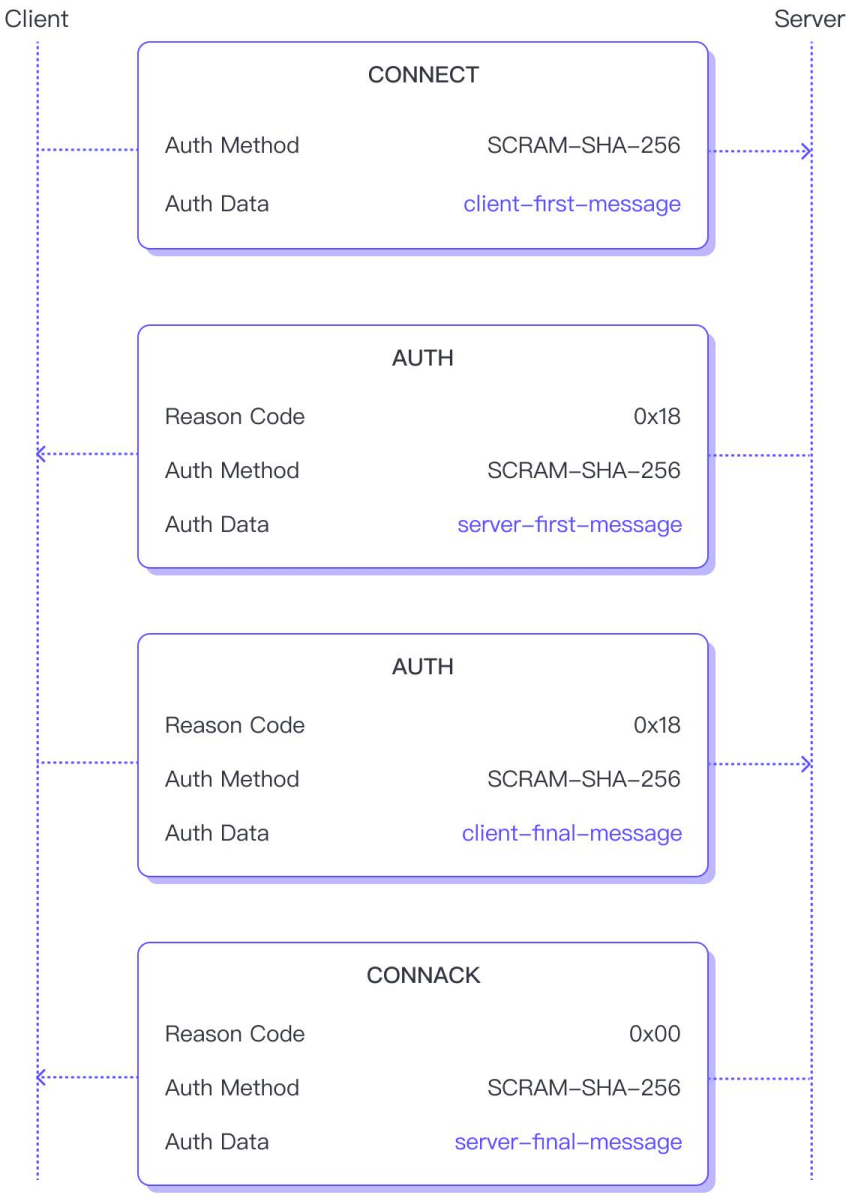
Kerberos 的核心是对称加密算法，服务端使用本地存储的密码哈希加密认证数据，然后返回给客户端。客户端对自己持有的密码进行哈希然后解密这些认证数据，这样的好处是无需在网络上明文传输密码，又能够让服务端和客户端相互验证对方都持有正确的密码。以这种通过对称加密交换数据的方式，服务端和客户端还能够安全地完成会话密钥的共享，这个密钥可以被用于后续通信数据的加密，以提供对通信数据的安全保护。

Kerberos 在提供较强安全性的同时，也带来了相当的复杂性，它的实现和配置都存在一定的门槛，另外多达六次的握手对于网络延迟和可靠性也提出了比较高的要求，所以通常 Kerberos 主要在企业的内网环境中使用。

## 增强认证在 MQTT 中是如何运行的

以 SCRAM 机制为例，我们来看一下在 MQTT 中增强认证是如何进行的。至于 SCRAM 的具体原理本文不作展开，在这里，我们只需要知道，SCRAM 需要传递四次消息才能完成认证：

- client-first-message
- server-first-message
- client-final-message
- server-final-message




首先，客户端仍然需要发送 CONNECT 报文来发起认证，只是需要将 Authentication Method 属性设置为 SCRAM-SHA-256 表示想要使用 SCRAM 认证，其中 SHA-256 表示准备使用的哈希函数，同时使用 Authentication Data 属性存放 client-first-message 的内容。Authentication Method 决定了服务端应该如何解析和处理 Authentication Data 中的数据。

如果服务端不支持 SCRAM 认证，或者发现 client-first-message 的内容不合法，那么它将返回包含指示认证失败原因的 Reason Code 的 CONNACK 报文，然后关闭网络连接。

反之服务端就会继续进行下一步：返回一个 AUTH 报文，并且将 Reason Code 设置为 0x18，表示继续认证。报文中的 Authentication Method 将与 CONNECT 报文相同，而 Authentication Data 属性将包含 server-first-message 的内容。

客户端在确认 server-first-message 的内容无误后，同样返回一个 Reason Code 为 0x18 的 AUTH 报文，Authentication Data 属性将包含 client-final-message 的内容。

在服务端确认 client-final-message 的内容无误后，服务端就已经完成了对客户端身份的验证。所以这次服务端将不再是返回 AUTH 报文，而是返回一个 Reason Code 为 0 的 CONNACK 报文以表示认证成功，并通过报文中的 Authentication Data 属性传递最终的 server-final-message。客户端需要根据这个消息的内容来验证服务端的身份。

如果服务端的身份通过了验证，那么客户端就可以开始订阅主题或者发布消息了，而如果没有通过验证，客户端将会发送 DISCONNECT 报文来终止这一次的连接。

# 控制报文

## 什么是 MQTT 控制报文?

MQTT 控制报文是 MQTT 数据传输的最小单元。MQTT 客户端和服务端通过交换控制报文来完成它们的工作，比如订阅主题和发布消息。

MQTT 目前定义了 15 种控制报文类型，如果按照功能进行分类，我们可以将这些报文分为连接、发布、订阅三个类别：

| Connect | Publish | Subscribe |
| --- | --- | --- |
| CONNECT | PUBLISH | SUBSCRIBE |
| CONNACK | PUBACK | SUBACK |
| DISCONNECT | PUBREC | UNSUBSCRIBE |
| AUTH Only MQTT 5.0 | PUBREL | UNSUBACK |
| PINGREQ | PUBCOMP | |
| PINGRESP | | |

其中，CONNECT 报文用于客户端向服务端发起连接，CONNACK 报文则作为响应返回连接的结果。如果想要结束通信，或者遇到了一个必须终止连接的错误，客户端和服务端可以发送一个 DISCONNECT 报文然后关闭网络连接。

AUTH 报文是 MQTT 5.0 引入的全新的报文类型，它仅用于增强认证，为客户端和服务端提供更安全的身份验证。

PINGREQ 和 PINGRESP 报文用于连接保活和探活，客户端定期发出 PINGREQ 报文向服务端表示自己仍然活跃，然后根据 PINGRESP 报文是否及时返回判断服务端是否活跃。

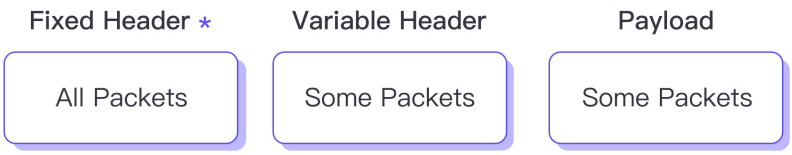
PUBLISH 报文用于发布消息，余下的四个报文分别用于 QoS 1 和 2 消息的确认流程。

SUBSCRIBE 报文用于客户端向服务端发起订阅，UNSUBSCRIBE 报文则正好相反，SUBACK 和 UNSUBACK 报文分别用于返回订阅和取消订阅的结果。

# MQTT 报文格式

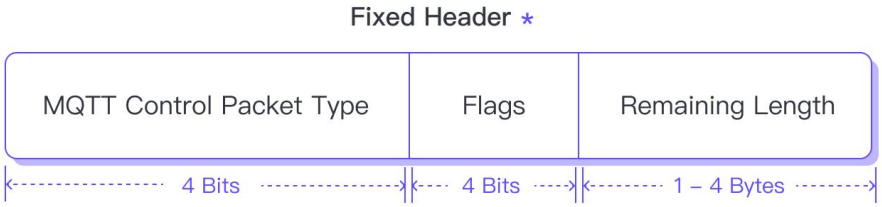
在 MQTT 中，无论是什么类型的控制报文，它们都由固定报头、可变报头和有效载荷三个部分组成。

固定报头固定存在于所有控制报文中，而可变报头和有效载荷是否存在以及它们的内容则取决于具体的报文类型。例如用于维持连接的 PINGREQ 报文就只有一个固定报头，用于传递应用消息的 PUBLISH 报文则完整地包含了这三个部分。




## 固定报头

固定报头由报文类型、标识位和报文剩余长度三个字段组成。



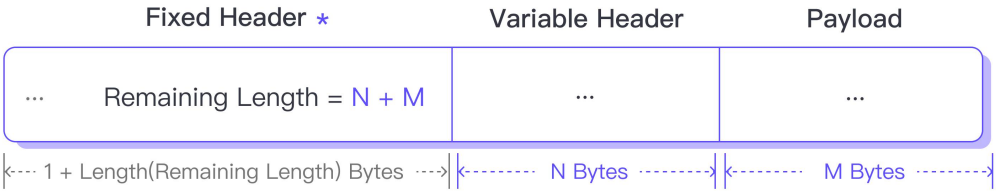

报文类型位于固定报头第一个字节的高 4 位，它是一个无符号整数，很显然，它表示当前报文的类型，例如 1 表示这是一个 CONNECT 报文，2 表示 CONNACK 报文等等。详细的映射关系可以参阅 MQTT 5.0 规范 – MQTT 控制报文类型。事实上，除了报文类型和剩余长度这两个字段，MQTT 报文剩余部分的内容基本都取决于具体的报文类型，所以这个字段也决定了接收方应该如何解析报文的后续内容。

固定报头第一个字节中剩下的低 4 位包含了由控制报文类型决定的标识位。不过到 MQTT 5.0 为止，只有 PUBLISH 报文的这四个比特位被赋予了明确的含义：

- Bit 3：DUP，表示当前 PUBLISH 报文是否是一个重传的报文。
- Bit 2,1：QoS，表示当前 PUBLISH 报文使用的服务质量等级。
- Bit 0：Retain，表示当前 PUBLISH 报文是否是一个保留消息。

其他所有的报文中，这 4 位都仍是保留的，即它们是一个固定的，不可随意变更的值。

最后的剩余长度指示了当前控制报文剩余部分的字节数，也就是可变报头和有效载荷这两个部分的长度。所以 MQTT 控制报文的总长度实际上等于固定报头的长度加上剩余长度。




**可变字节整数**

但固定报头长度并不是固定的，为了尽可能地减少报文大小，MQTT 将剩余长度字段设计成了一个可变字节整数。

在 MQTT 中，存在很多长度不确定的字段，例如 PUBLISH 报文中的 Payload 部分就用来承载实际的应用消息内容，而应用消息的长度显然是不固定的。所以我们需要一个额外的字段来指示这些不定长内容的长度，以便接收端正确地解析。

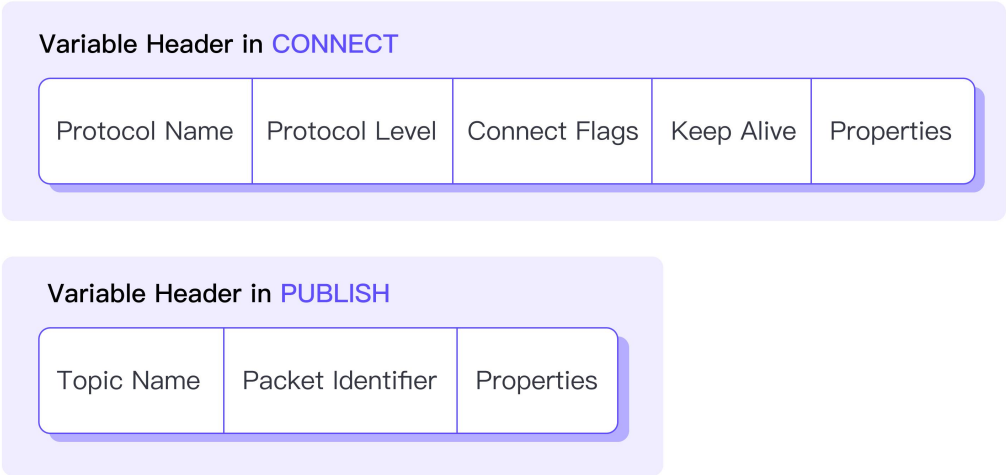
一个 2 兆大小，也就是总共 2,097,152 个字节的应用消息，我们就需要一个 4 字节长度的整数才能够指示它的长度。但并不是所有的应用消息都有这么大，更多情况下是几 KB 甚至几个字节。用一个 4 字节长度的整数来指示一个总共 2 个字节长度的应用消息，显然是过于浪费了。

所以 MQTT 的可变字节整数就被设计出来了，它将每个字节中的低 7 位用于编码数据，最高的有效位用于指示是否还有更多的字节。这样，长度小于 128 字节时可变字节整数只需要一个字节就可以指示。可变字节整数的最大长度为 4 个字节，所以最多可以指示长度为（2^28 – 1）字节，也就是 256 MB 的数据。

| Digits | From | To |
| --- | --- | --- |
| 1 | 0 (0x00) | 127 (0x7F) |
| 2 | 128 (0x80, 0x01) | 16,383 (0xFF, 0x7F) |
| 3 | 16,384 (0x80, 0x80, 0x01) | 2,097,151 (0xFF, 0xFF, 0x7F) |
| 4 | 2,097,152 (0x80, 0x80, 0x80, 0x01) | 268,435,455 (0xFF, 0xFF, 0xFF, 0x7F) |

## 可变报头

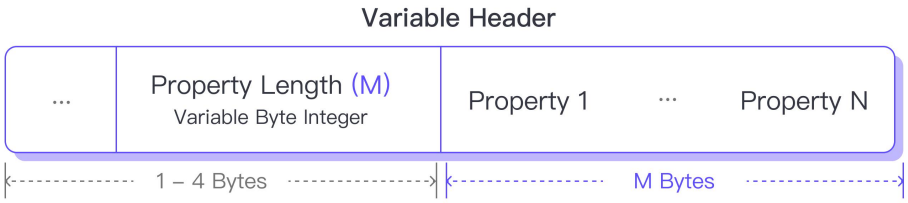
可变报头的内容取决于具体的报文类型。例如 CONNECT 报文的可变报头按顺序包含了协议名、协议级别、连接标识、Keep Alive 和属性这五个字段。PUBLISH 报文的可变报头则按顺序包含了主题名、报文标识符和属性这三个字段。




需要注意这里提到的顺序，可变报头中字段出现的顺序必须严格遵循协议规范，因为接收端只会按照协议规定的字段顺序进行解析。我们也不能随意地遗漏某个字段，除非是协议明确要求或允许的。例如，在 CONNECT 报文的可变报头中，如果协议名之后直接就是连接标识，那么就会导致报文解析失败。而在 PUBLISH 报文的可变报头中，报文标识符就只有在 QoS 不为 0 的时候才能存在。

### 属性

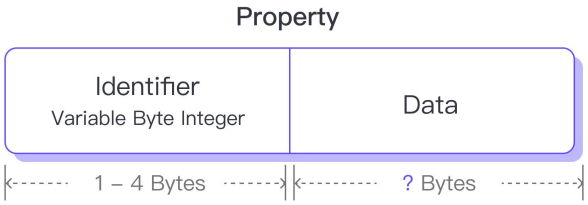
属性是 MQTT 5.0 引入的一个概念。属性字段基本上都是可变报头的最后一部分，由属性长度和紧随其后的一组属性组成，这里的属性长度指的是后面所有属性的总长度。




所有的属性都是可选的，因为它们通常都有一个默认值，如果没有任何属性，那么属性长度的值就为 0。

每个属性都由一个定义了属性用途和数据类型的标识符和具体的值组成。不同属性的数据类型可能不同，比如一个是双字节长度的整数，另一个则是 UTF-8 编码的字符串，所以我们需要按照标识符所声明的数

据类型对属性进行解析。




属性之间的顺序可以是任意的，这是因为我们可以根据标识符知道这是哪个属性，以及它的长度是多少。

属性通常都是为了某个专门的用途而设计的，比如在 CONNECT 报文中就有一个用于设置会话过期时间的的 Session Expiry Interval 属性，但显然我们在 PUBLISH 报文中就不需要这个属性。所以 MQTT 也严格定义了属性的使用范围，一个合法的 MQTT 控制报文中不应该包含不属于它的属性。

包含标识符、属性名、数据类型和使用范围的完整 MQTT 属性列表，请参阅 MQTT 5.0 Specification – Properties。

### 有效载荷

最后是有效载荷部分。我们可以将报文的可变报头看作是它的附加项，而有效载荷则用于实现这个报文的核心目的。

比如在 PUBLISH 报文中，Payload 用于承载具体的应用消息内容，这也是 PUBLISH 报文最核心的功能。而 PUBLISH 报文的可变报头中的 QoS、Retain 等字段，则是围绕着应用消息提供一些额外的能力。

SUBSCRIBE 报文也是如此，Payload 包含了想要订阅的主题以及对应的订阅选项，这也是 SUBSCRIBE 报文最主要的工作。

# 结语

作为物联网领域通信协议的事实标准，MQTT 的使用已成为每个物联网开发者必须掌握的技能。通过本电子书，希望读者能够对 MQTT 协议拥有更加深入的认知，在物联网开发中更加得心应手。

**免费 MQTT 消息云服务，三秒极速创建部署**：立即体验

更多关于 MQTT 的内容请点击：

- MQTT 在各类客户端的应用实践
- MQTT 入门系列视频教程

# 杭州映云科技有限公司

全球领先的物联网数据基础设施软件供应商

获取更多
物联网技术干货

产品咨询
专家演示预约

官网：www.emqx.com

邮箱：contact@emqx.io

电话：400-696-5502

Bilibili：EMQ 映云科技