EMQ

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 协议简介 2

为什么 MQTT 是适用于物联网的最佳协议？ 3

MQTT 5.0 与 3.1.1 6

MQTT 服务器 6

MQTT 客户端 6

MQTT 发布/订阅模式介绍 **8**

什么是发布/订阅模式？ 8

MQTT 发布/订阅中的消息路由 9

MQTT 与 HTTP 请求响应 10

MQTT 与消息队列 10

创建 MQTT 连接 **11**

MQTT 连接的基本概念 11

MQTT 连接参数的使用 12

如何建立一个安全的 MQTT 连接？ 14

MQTT 主题与通配符 **16**

什么是 MQTT 主题？ 16

MQTT 主题通配符 16

不同场景中的主题设计 19

MQTT 主题常见问题及解答 21

MQTT 持久会话 **22**

什么是 MQTT 持久会话？ 22
MQTT Clean Session 的使用 23
MQTT 5.0 中的会话改进 27
关于 MQTT 会话的 Q&A 28
MQTT QoS 0, 1, 2 29
什么是 QoS 29
不同 QoS 的适用场景和注意事项 36
关于 MQTT QoS 的 Q&A 37
MQTT 保留消息 38
什么是 MQTT 保留消息？ 38
MQTT 保留消息的使用 39
关于 MQTT 保留消息的 Q&A 43
EMQX 中的 MQTT 保留消息 45
MQTT 遗嘱消息 47
演示遗嘱消息的使用 48
进阶使用场景 51
MQTT Keep Alive 52
MQTT Keep Alive 的机制流程与使用 52
Keep Alive 与遗嘱消息 53
如何在 EMQX 中使用 Keep Alive 54
结语 56

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。

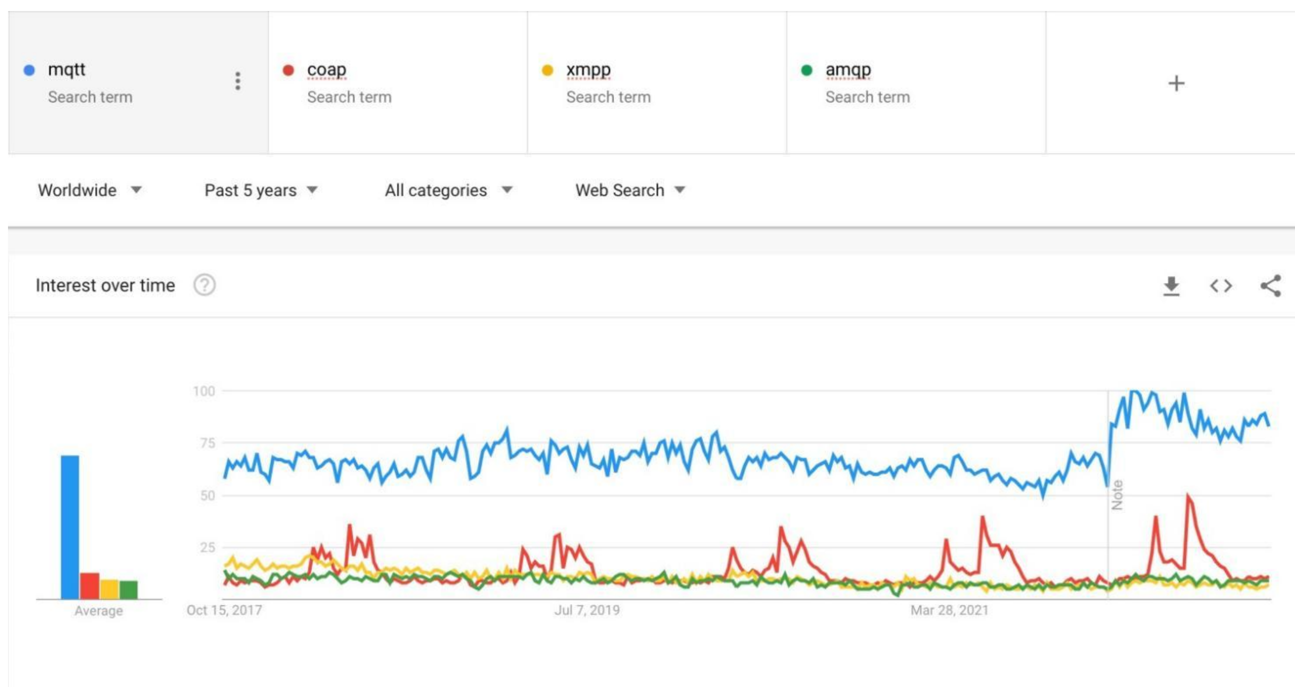
- MQTT 报文体积小且编解码容易，XMPP 基于繁重的 XML，报文体积大且交互繁琐。
- MQTT 基于发布订阅模式，相比 XMPP 基于 JID 的点对点消息路由更为灵活。
- MQTT 支持 JSON、二进制等不同类型报文。XMPP 采用 XML 承载报文，二进制必须 Base64 编码等处理。
- MQTT 通过 QoS 保证消息可靠传输，XMPP 主协议并未定义类似机制。

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。

MQTT X 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 MQTT X 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。

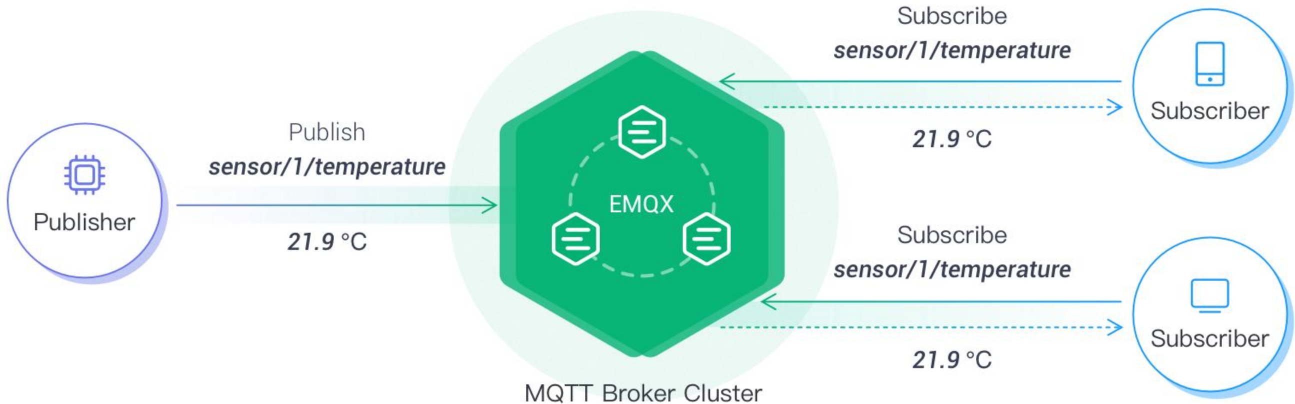
- **代理（Broker）**

  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、开发周期端，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?

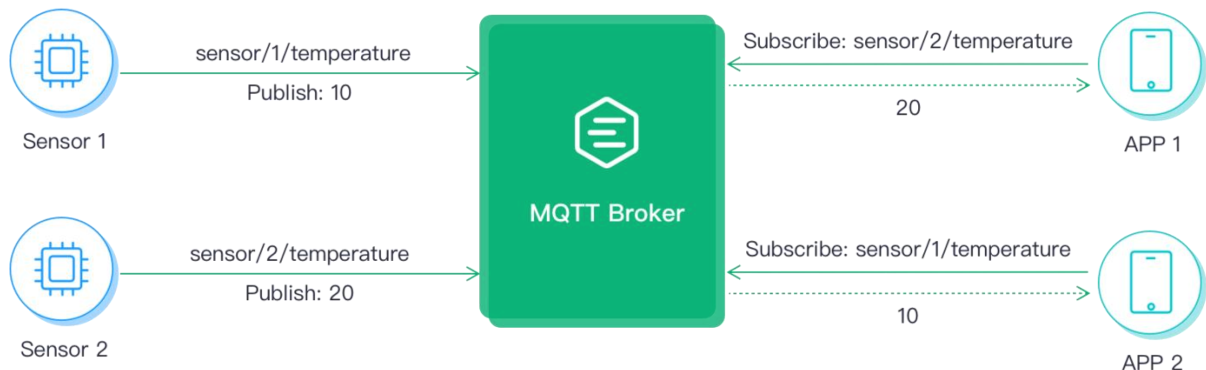
MQTT 主题本质上是一个 UTF-8 编码的字符串，是 MQTT 协议进行消息路由的基础。MQTT 主题类似 URL 路径，使用斜杠 `/` 进行分层：

```
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。

MQTT 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：

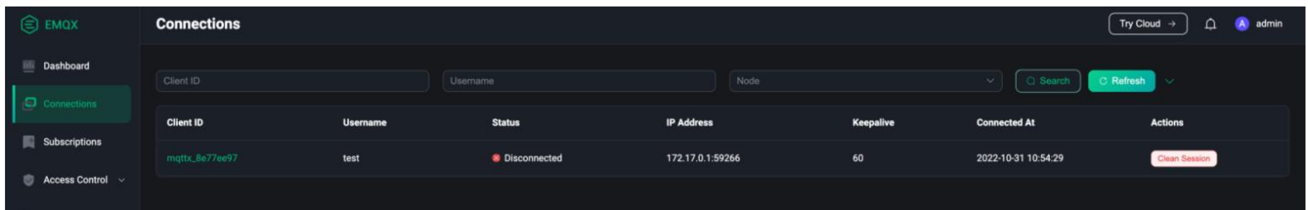
- 会话是否存在，即使会话状态其余部分为空。
- 已发送给客户端，但是还没有完成确认的 QoS 1 与 QoS 2 消息。
- 等待传输给客户端的 QoS 0 消息（可选），QoS 1 与 QoS 2 消息。
- 从客户端收到的，但是还没有完成确认的 QoS 2 消息，遗嘱消息和遗嘱延时间隔。

MQTT Clean Session 的使用

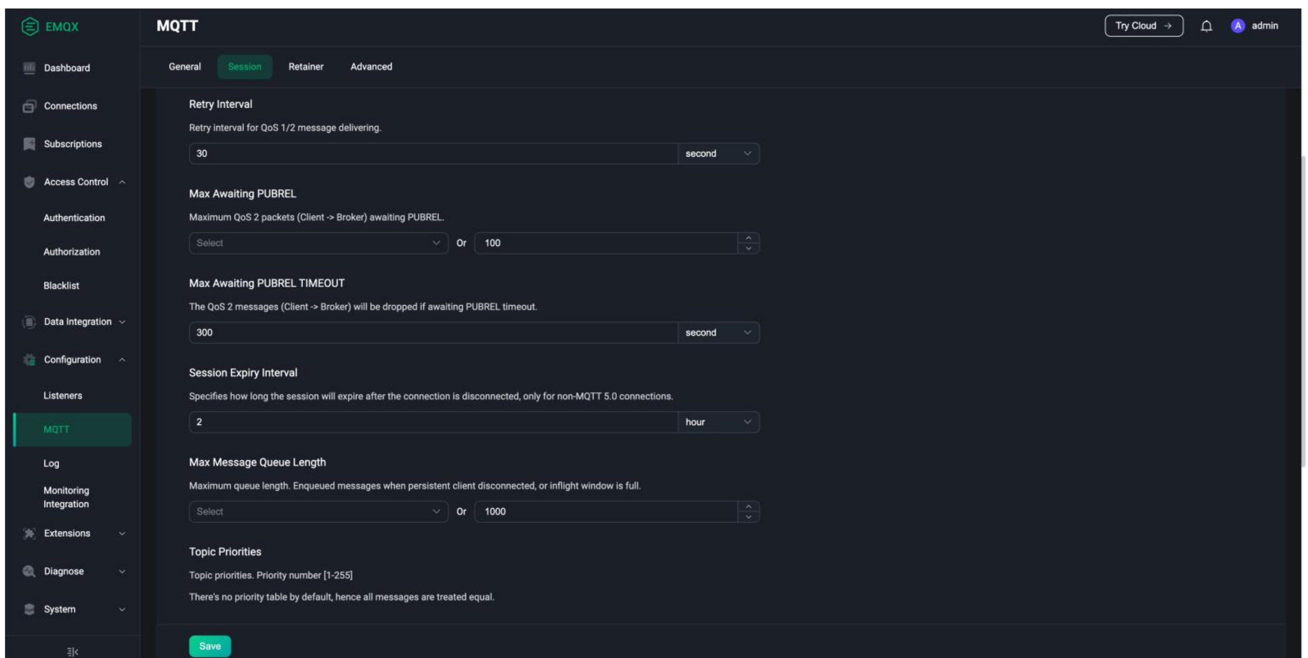
Clean Session 是用来控制会话状态生命周期的标志位，为 `true` 时表示创建一个新的会话，在客户端断开连接时，会话将自动销毁。为 `false` 时表示创建一个持久会话，在客户端断开连接后会话仍然保持，直到会话超时注销。

> 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。

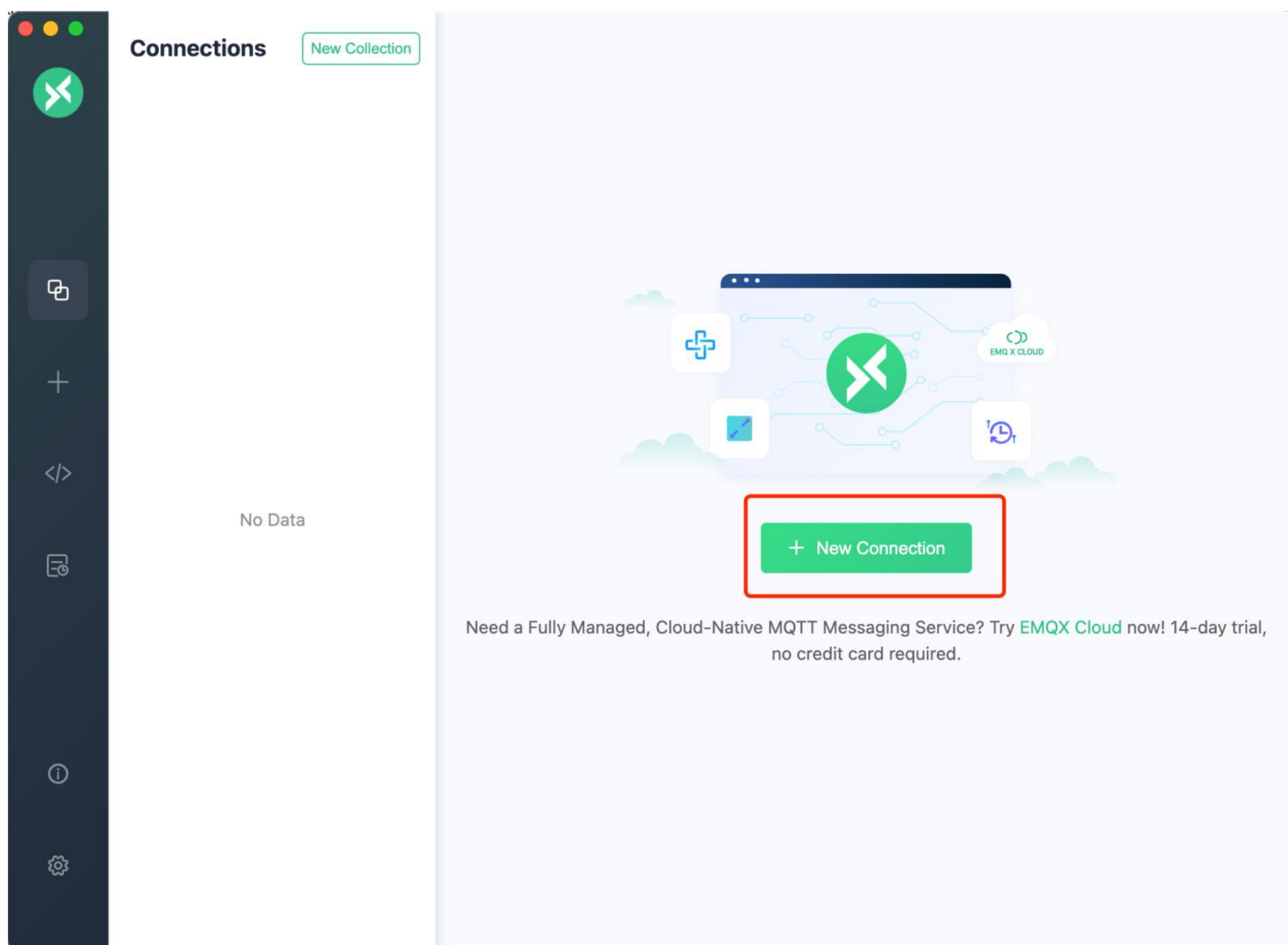


MQTT 3.1.1 没有规定持久会话应该在什么时候过期，如果仅从协议层面理解的话，这个持久会话应该永久存在。但在实际场景中这并不现实，因为它会非常占用服务端的资源，所以服务端通常不会完全遵循协议来实现，而是向用户提供一个全局配置来限制会话的过期时间。

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。

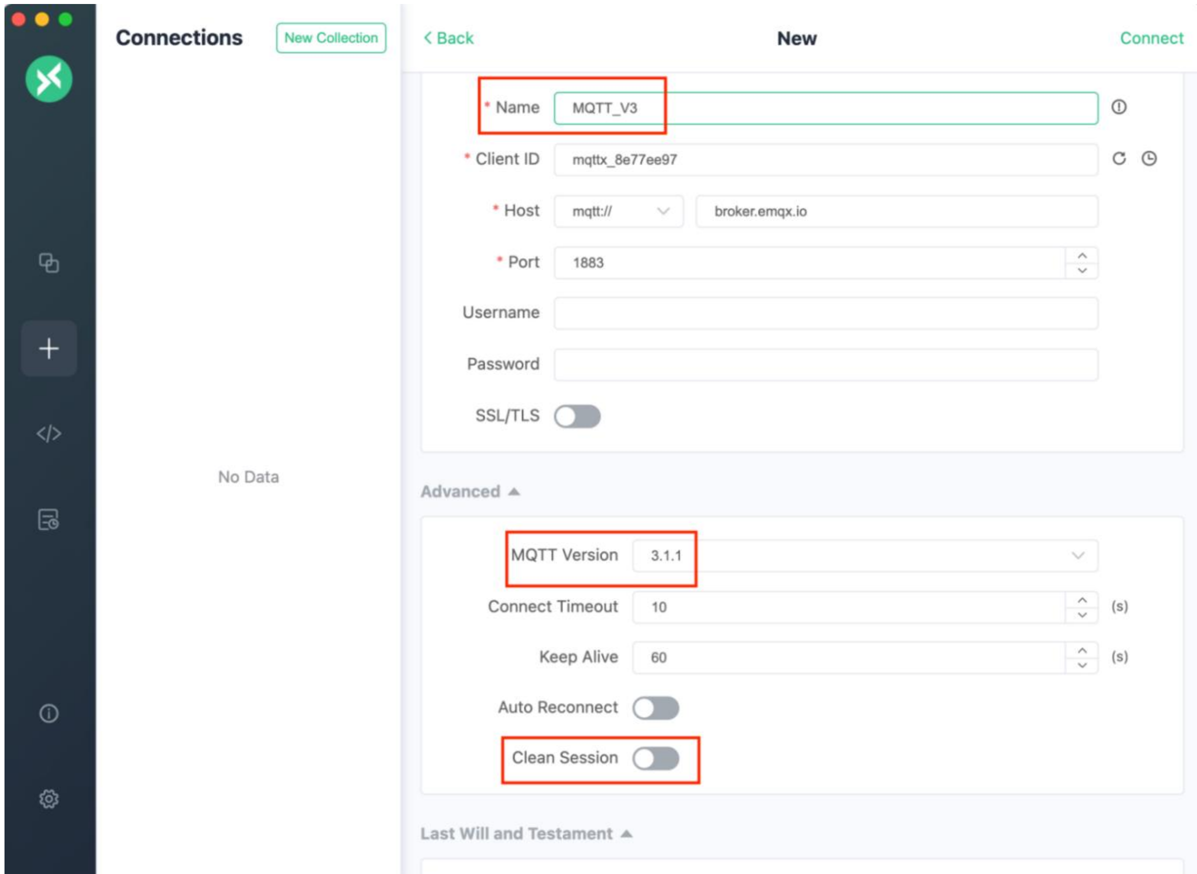
接下来我们使用开源的跨平台 MQTT 5.0 桌面客户端工具 – MQTT X 演示 Clean Session 的使用。

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

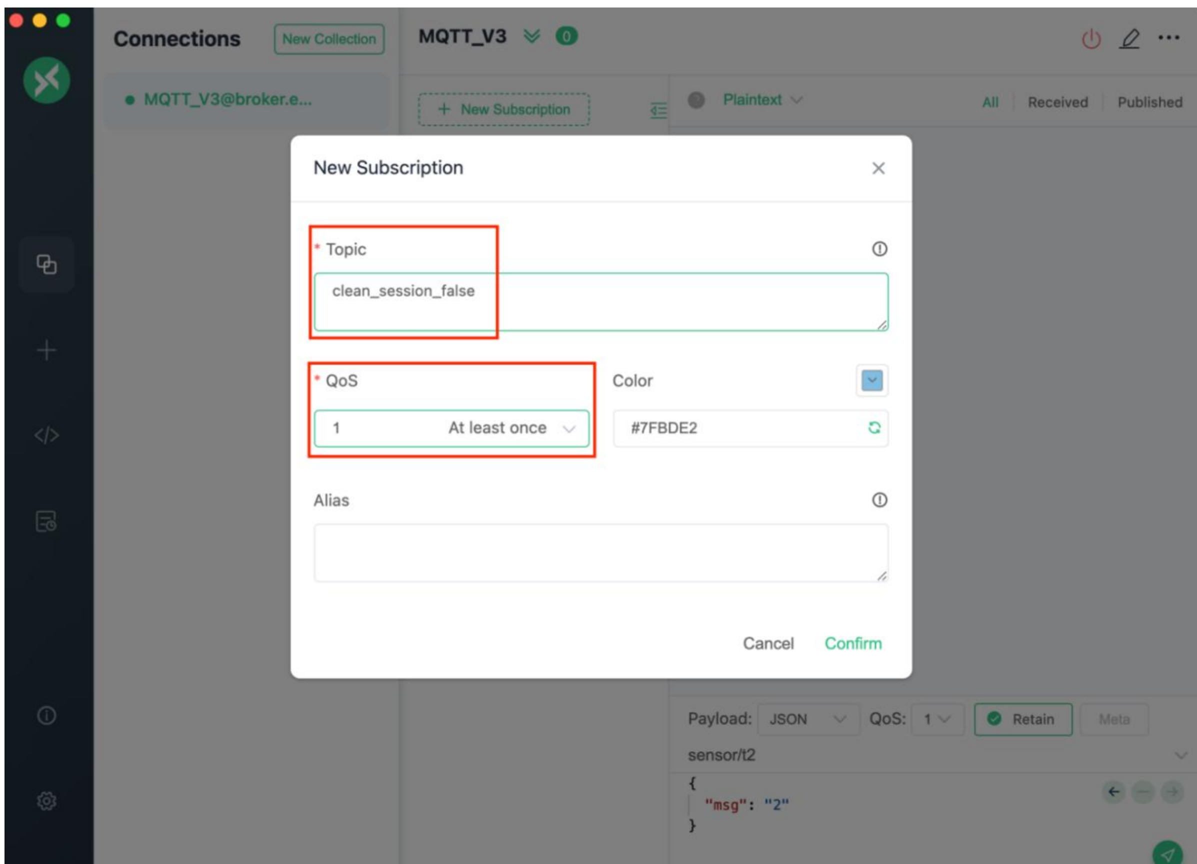


创建一个名为 `MQTT_V3` 的连接，Clean Session 为关闭状态（即为 false），MQTT 版本选择 3.1.1，然后点击右上角的 `Connect` 按钮。

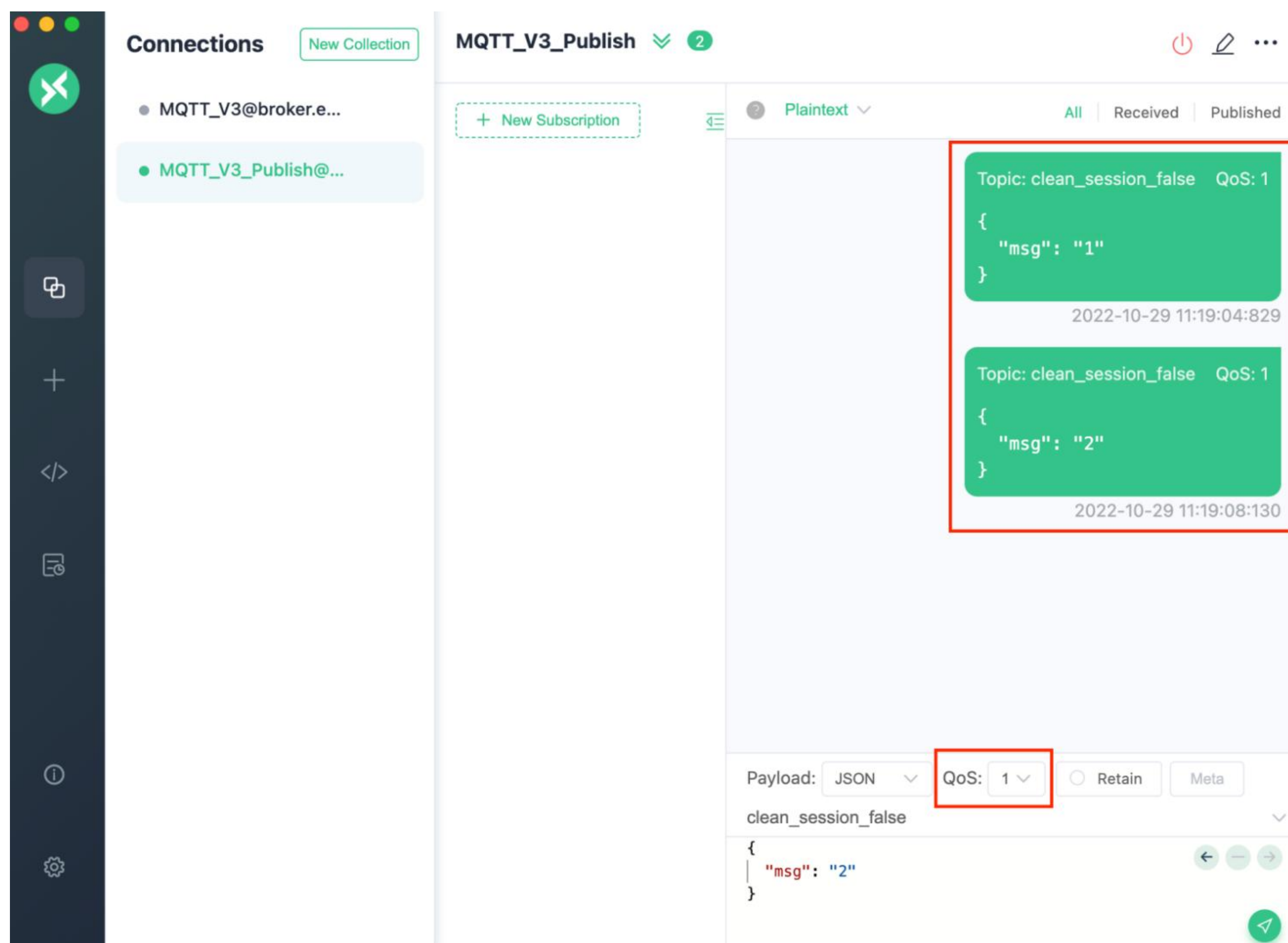
> 连接的服务器默认为 EMQ 提供的 免费的公共 MQTT 服务器。



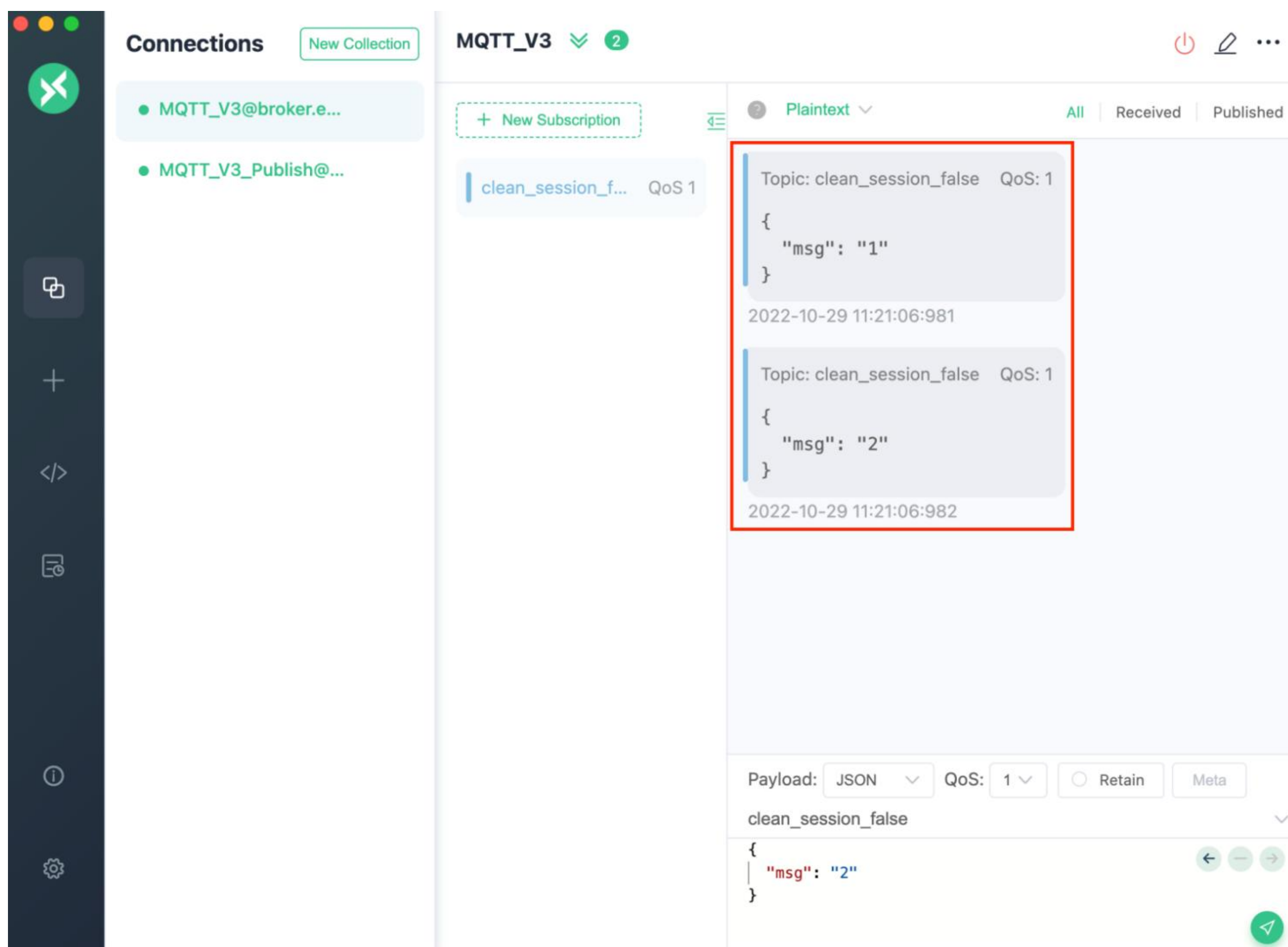
连接成功后订阅 clean_session_false 主题，且 QoS 设置为 1。



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。

MQTT 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。

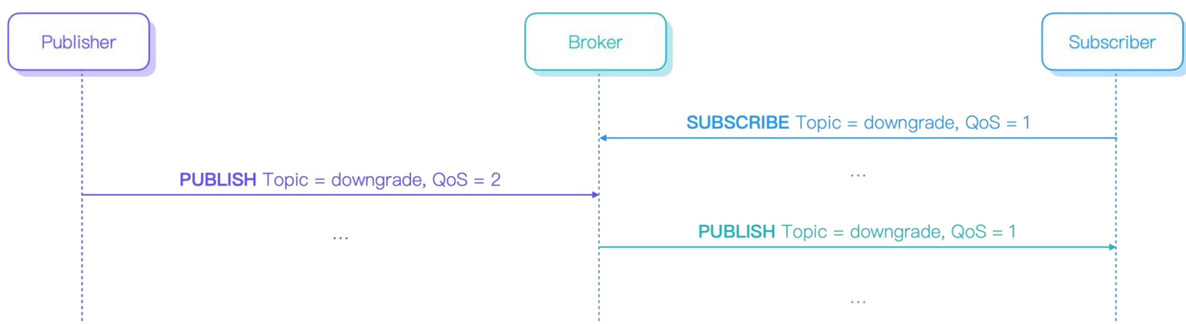
MQTT 定义了三个 QoS 等级，分别为：

- 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。



此处省略了完整的报文收发流程

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?

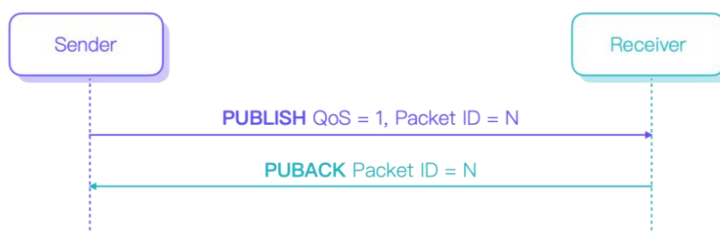
当我们使用 QoS 0 传递消息时，消息的可靠性完全依赖于底层的 TCP 协议。

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：

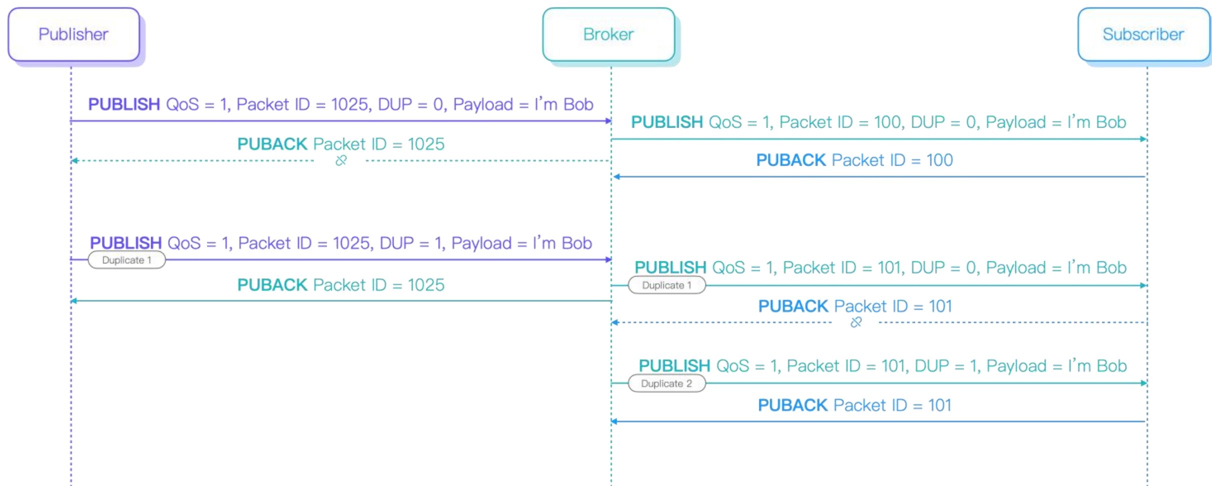
第一种情况，发送方由于没有收到 PUBACK 报文而重传了 PUBLISH 报文。此时，接收方收到的前后两个 PUBLISH 报文使用了相同的 Packet ID，并且第二个 PUBLISH 报文的 DUP 标志为 1，此时它确实是一个重复的消息。

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。

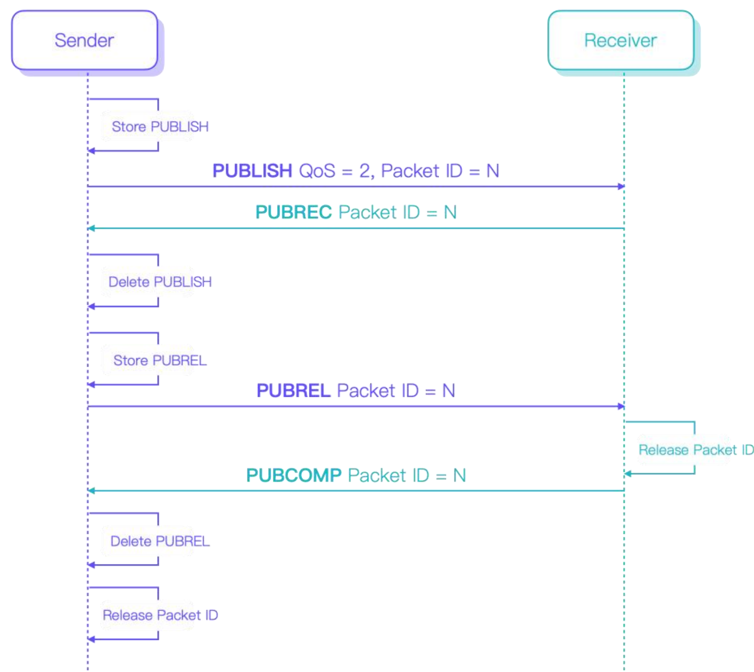
在下图表示的例子中，虽然发布者的本意只是发布一条消息，但对接收方来说，最终却收到了三条相同的消息：



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。

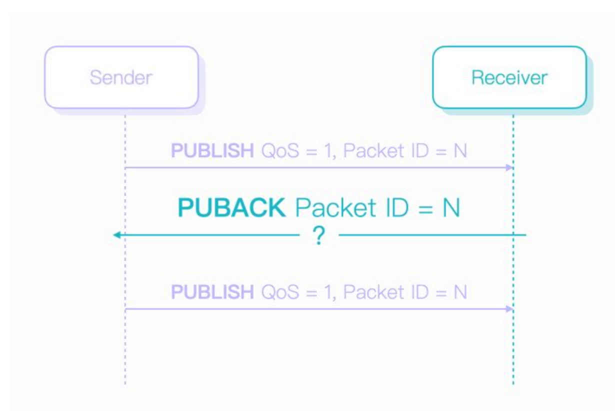
为什么 QoS 2 消息不会重复？

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。

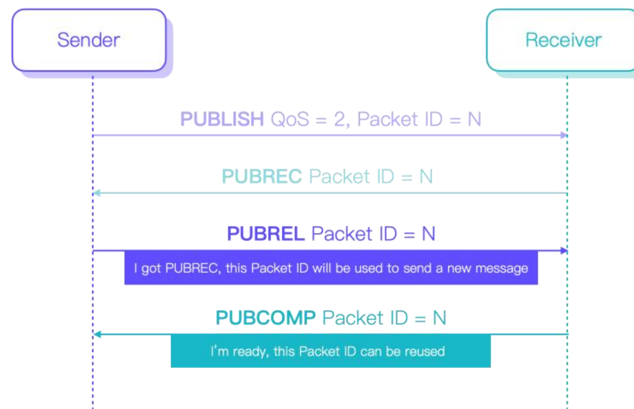
在我们更进一步之前，我们先快速回顾一下 QoS 1 消息无法避免重复的原因。

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。

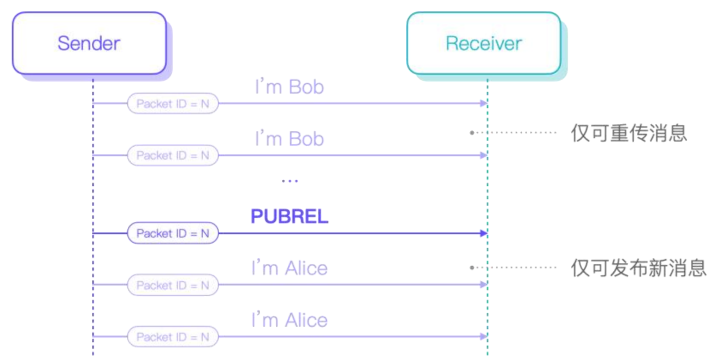


所以，消息去重的关键就在于，通信双方如何正确地同步释放 Packet ID，换句话说，不管发送方是重传消息还是发布新消息，一定是和对端达成共识了的。

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?

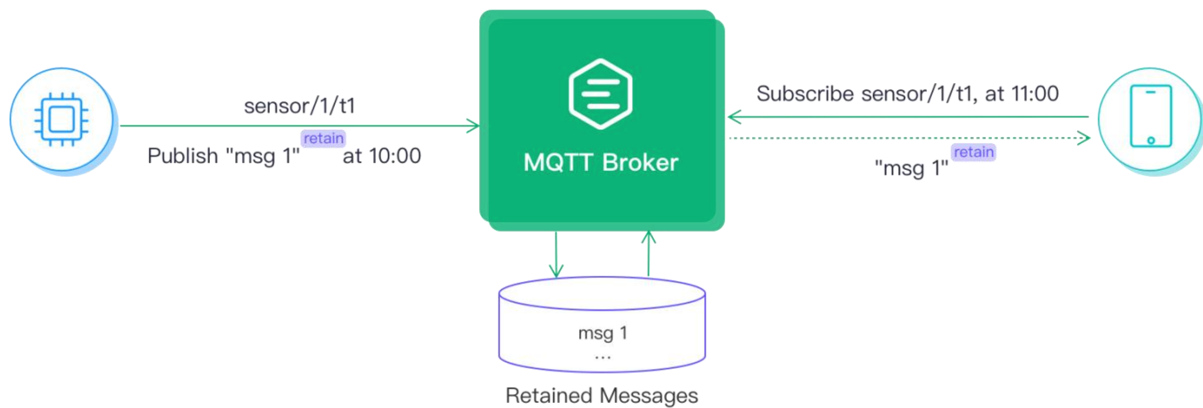
以 EMQX 为例，在相同的硬件配置下进行点对点通信，通常 QoS 0 与 QoS 1 能够达到的吞吐比较接近，不过 QoS 1 的 CPU 占用会略高于 QoS 0，负载较高时，QoS 1 的消息延迟也会进一步增加。而 QoS 2 能够达到的吞吐一般仅为 QoS 0、1 的一半左右。

MQTT 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。

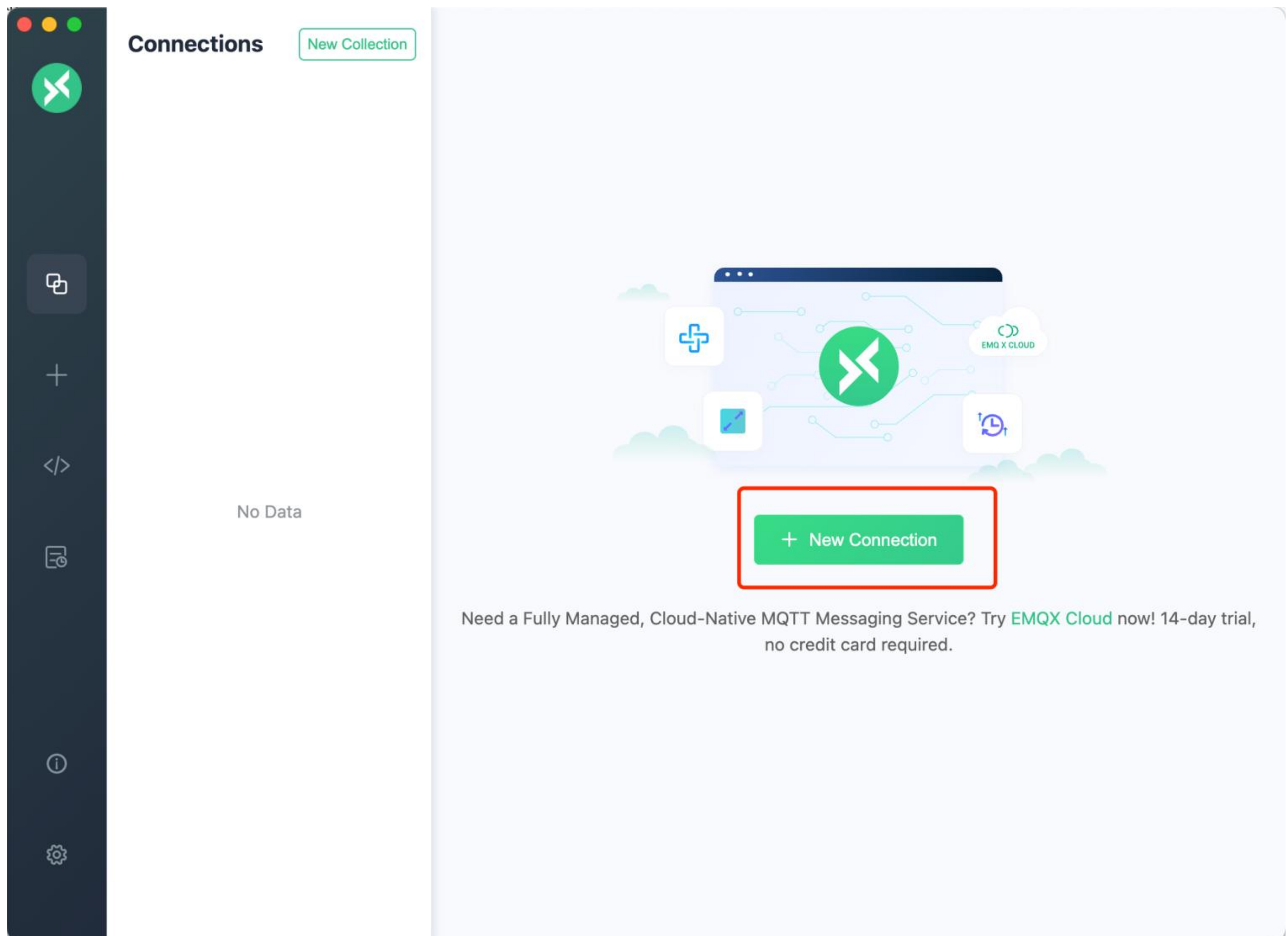
借助保留消息，新的订阅者能够立即获取最近的状态，而不需要等待无法预期的时间，例如：

- 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。

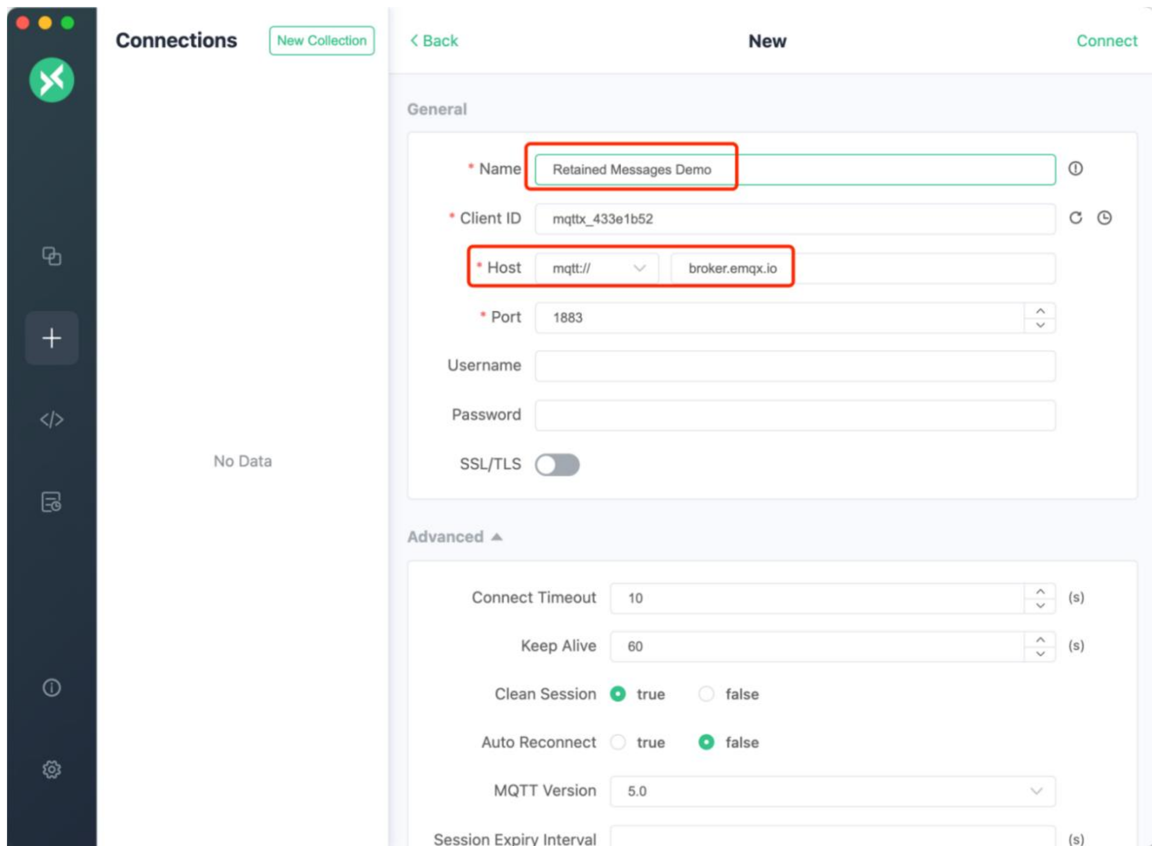
MQTT 保留消息的使用

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 – MQTT X 为例，演示如何使用 MQTT 保留消息。

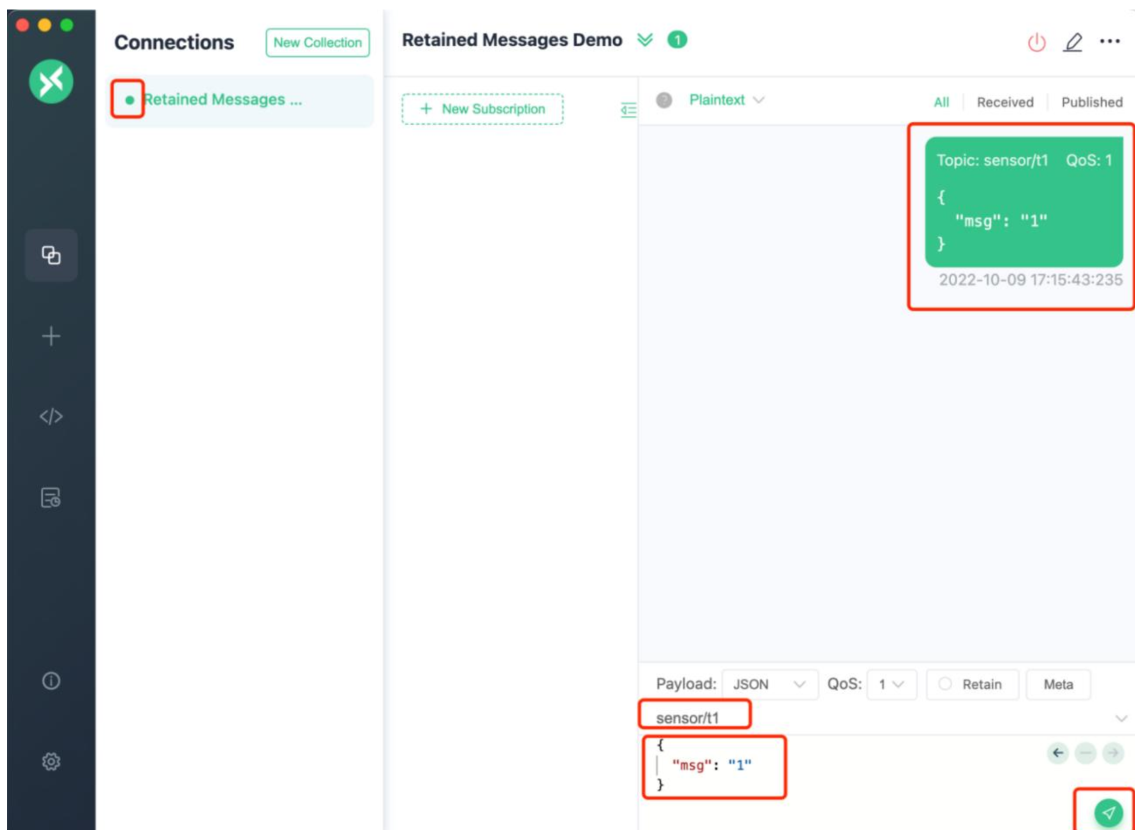
打开 MQTT X 后如下所示，需点击 New Connection 按钮创建一个 MQTT 连接。



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。

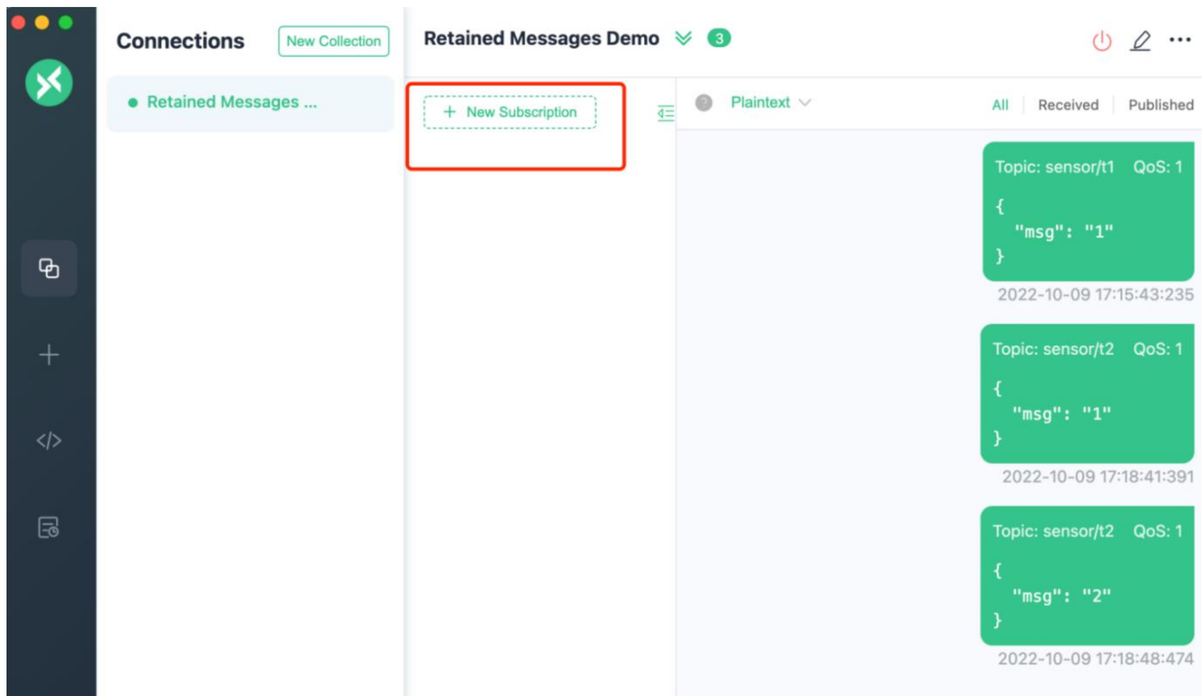


连接成功后将会看到连接名称旁边的状态为绿色。然后我们在右下角消息输入框向主题 **sensor/t1** 发送一条普通的消息。



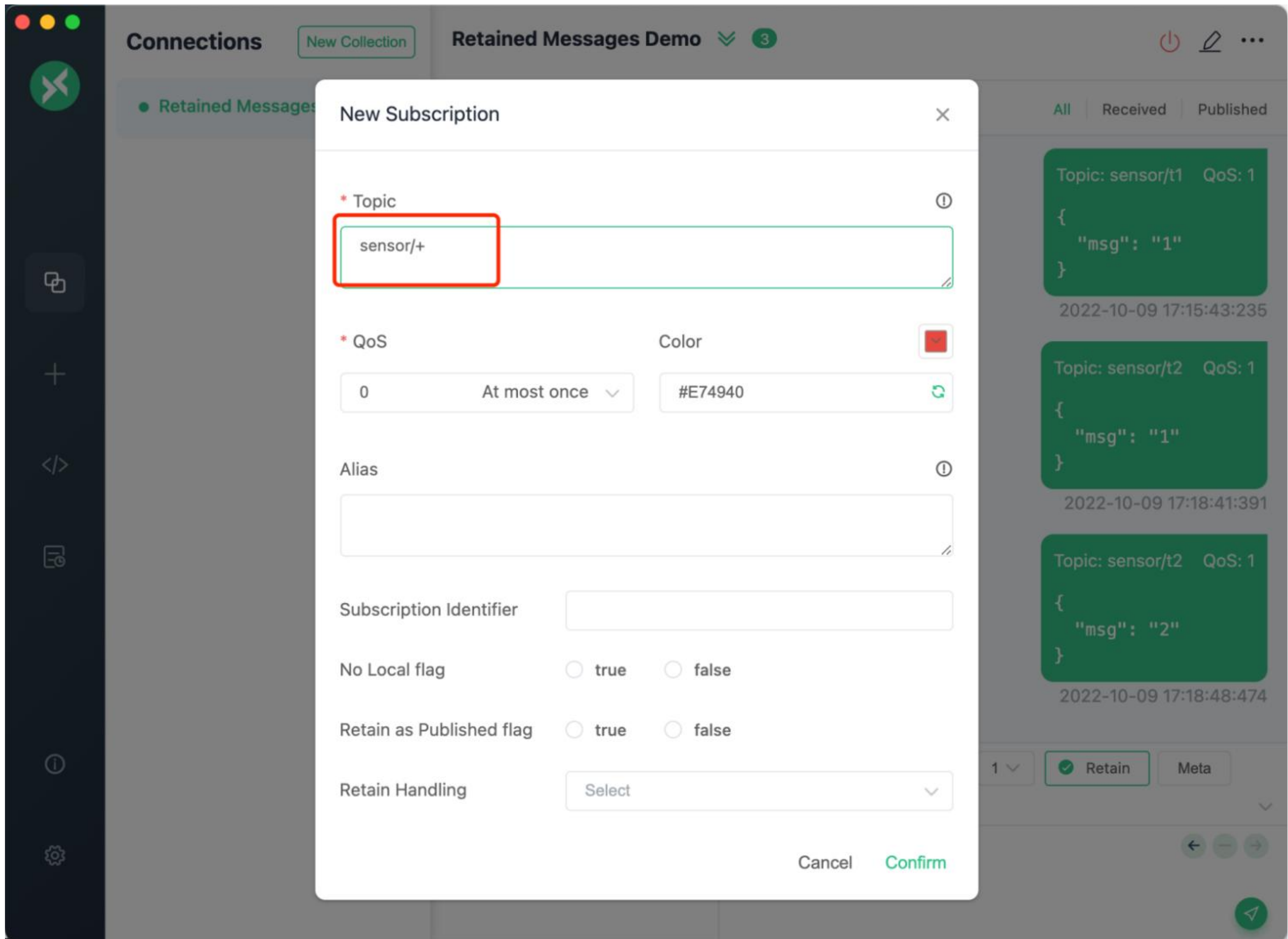
接下来我们选中右下角的 Retain 标记，并向主题 **sensor/t2** 发送两条保留消息。

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

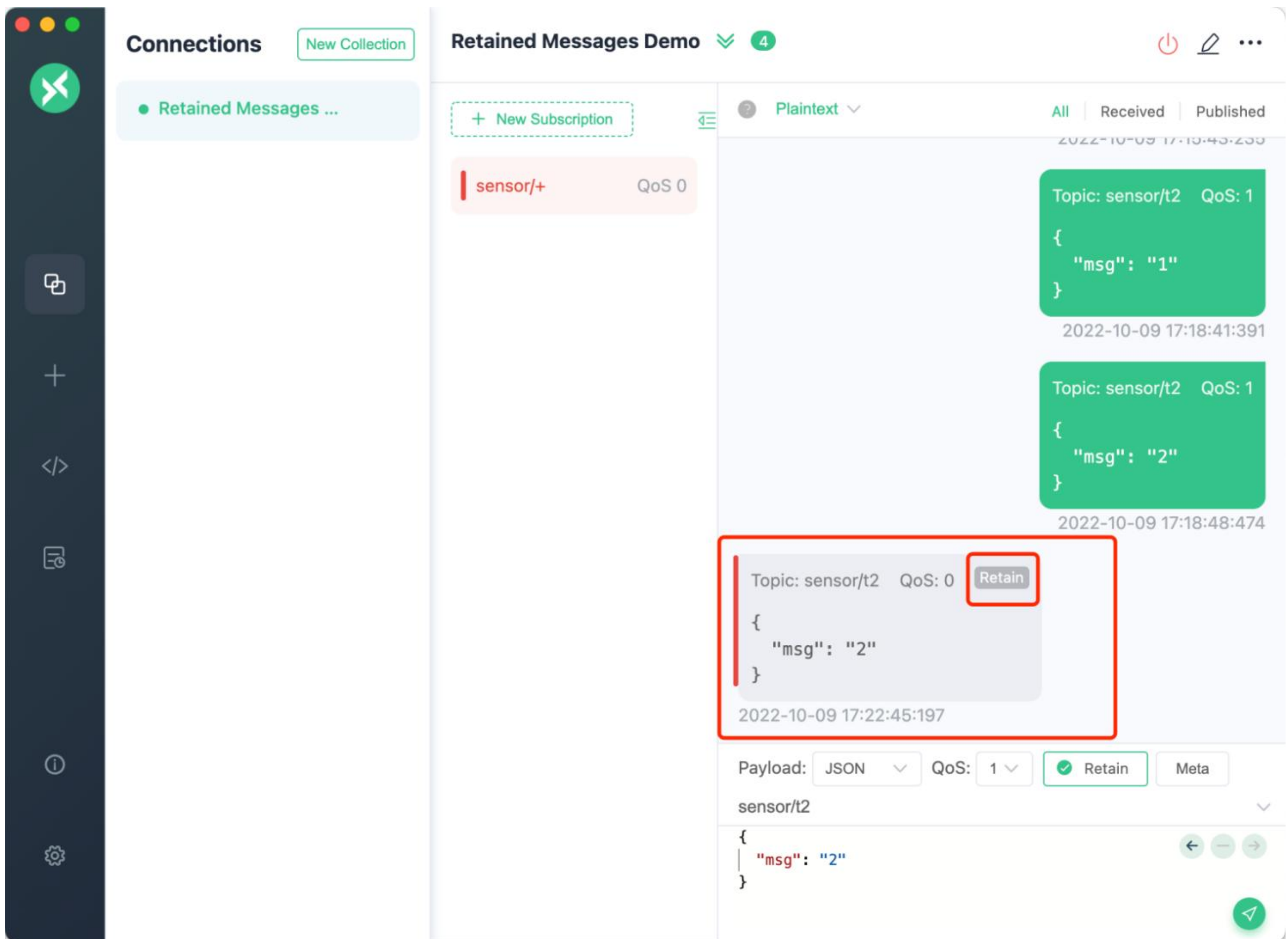


如下，我们订阅通配符主题 sensor/+，该通配符主题将会匹配主题 sensor/t1 及 sensor/t2。

> 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。

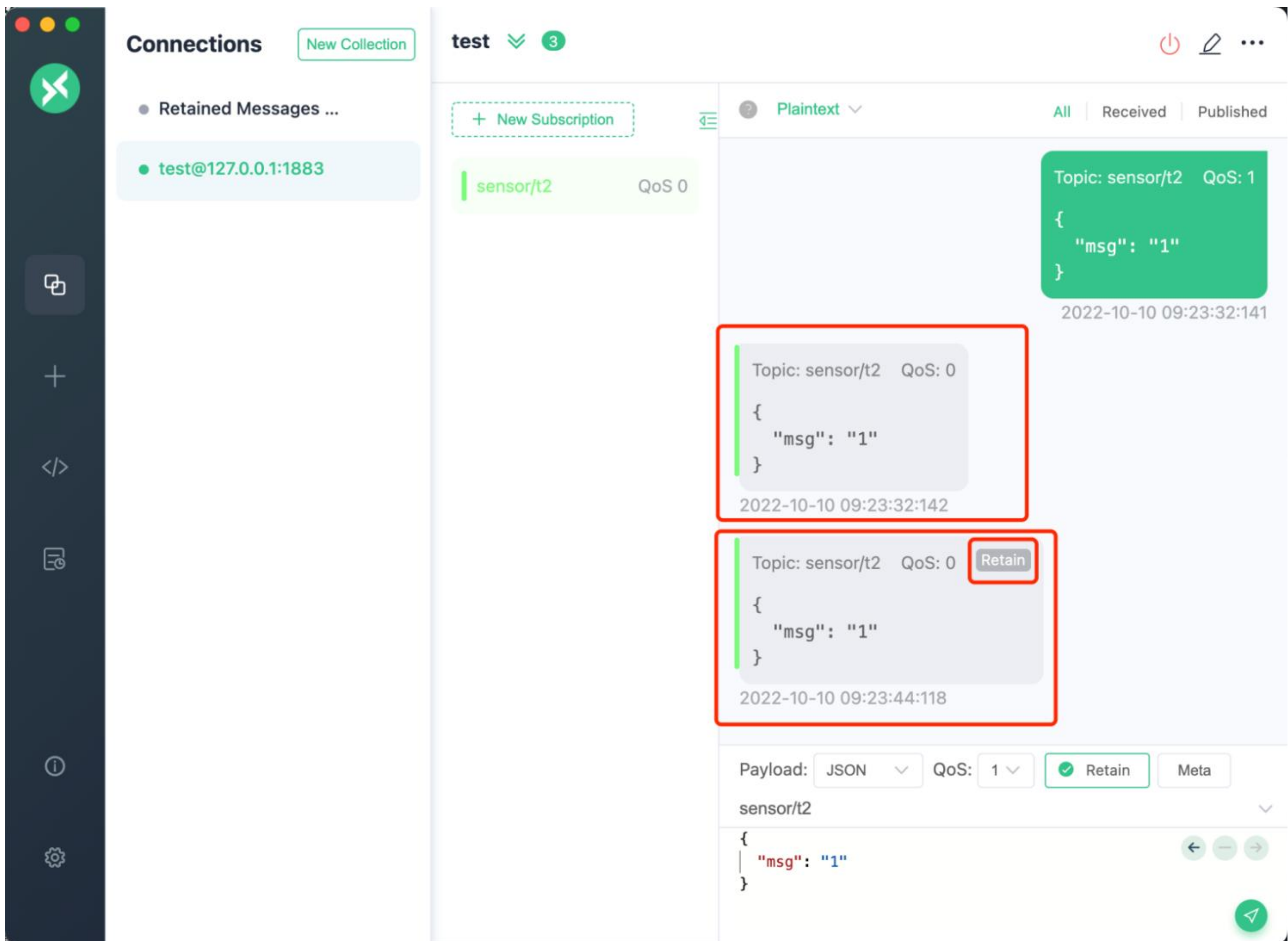


关于 MQTT 保留消息的 Q&A

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。

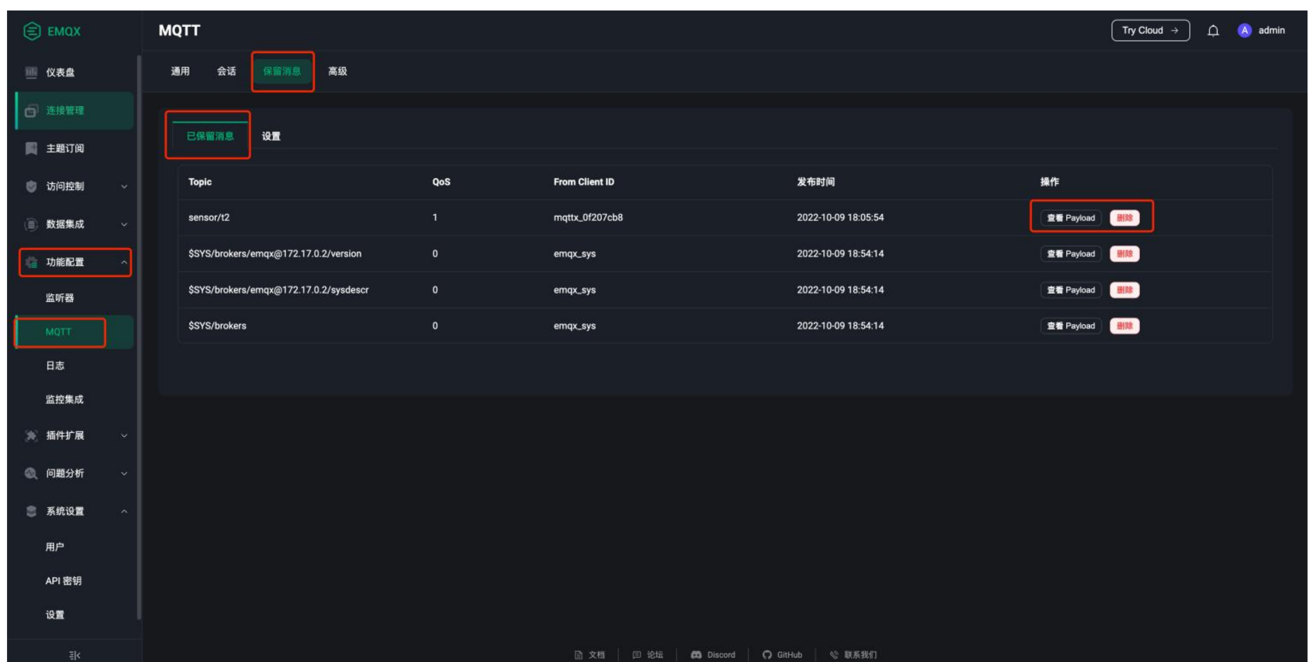
EMQX 5.0 支持在内置的 Dashboard 中查看、设置保留消息。感兴趣的读者可通过如下 Docker 命令安装 EMQX 5.0 开源版进行体验。

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

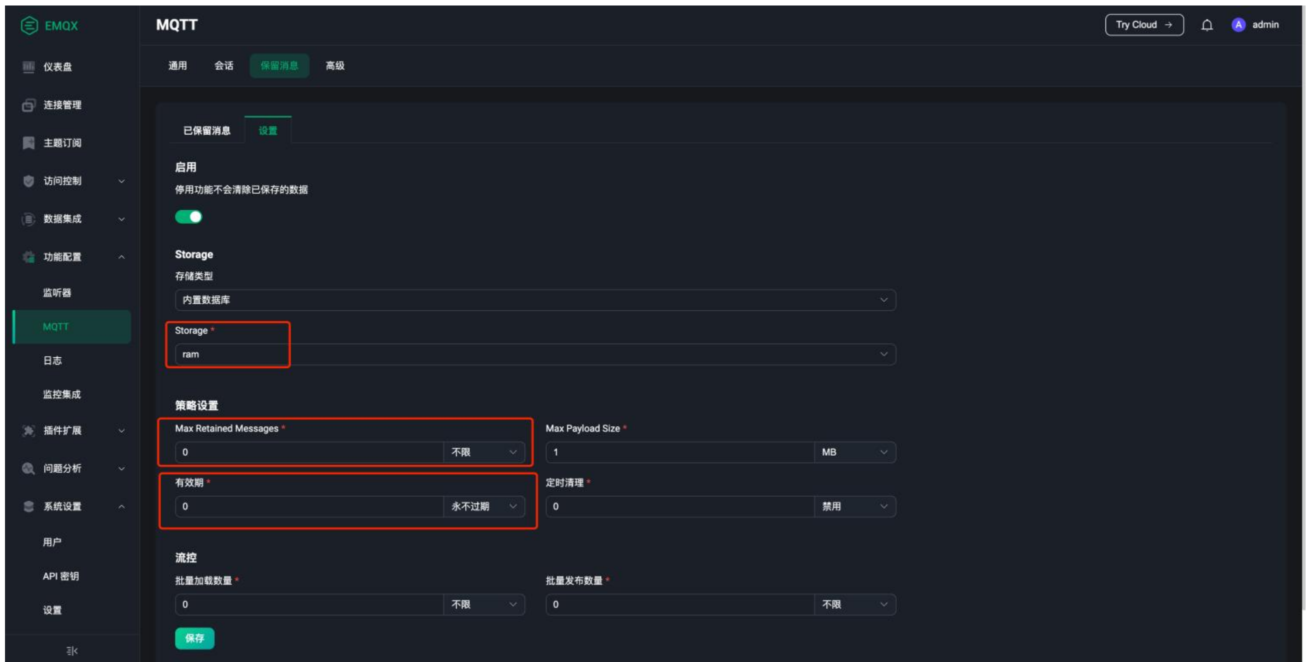
EMQX 安装成功后，使用浏览器访问 `http://127.0.0.1:18083/` 即可体验 EMQX 5.0 全新 Dashboard。

> 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。



MQTT 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 MQTT X 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"]}.
```

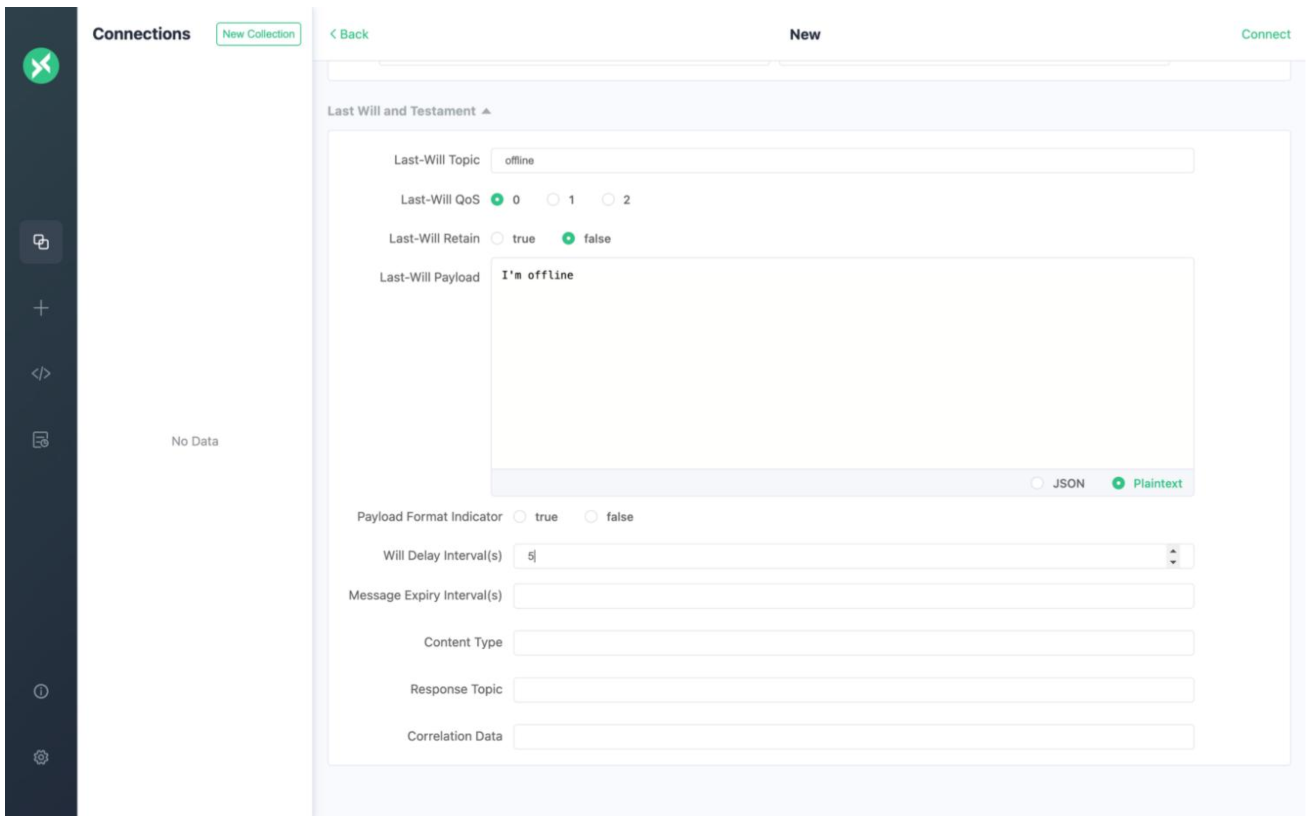
然后修改 `etc/emqx.conf` 中 `zone.internal.acl_deny_action` 配置项，将其设置为 ACL 检查拒绝时断开客户端连接：

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

完成以上修改后，我们启动 EMQX。

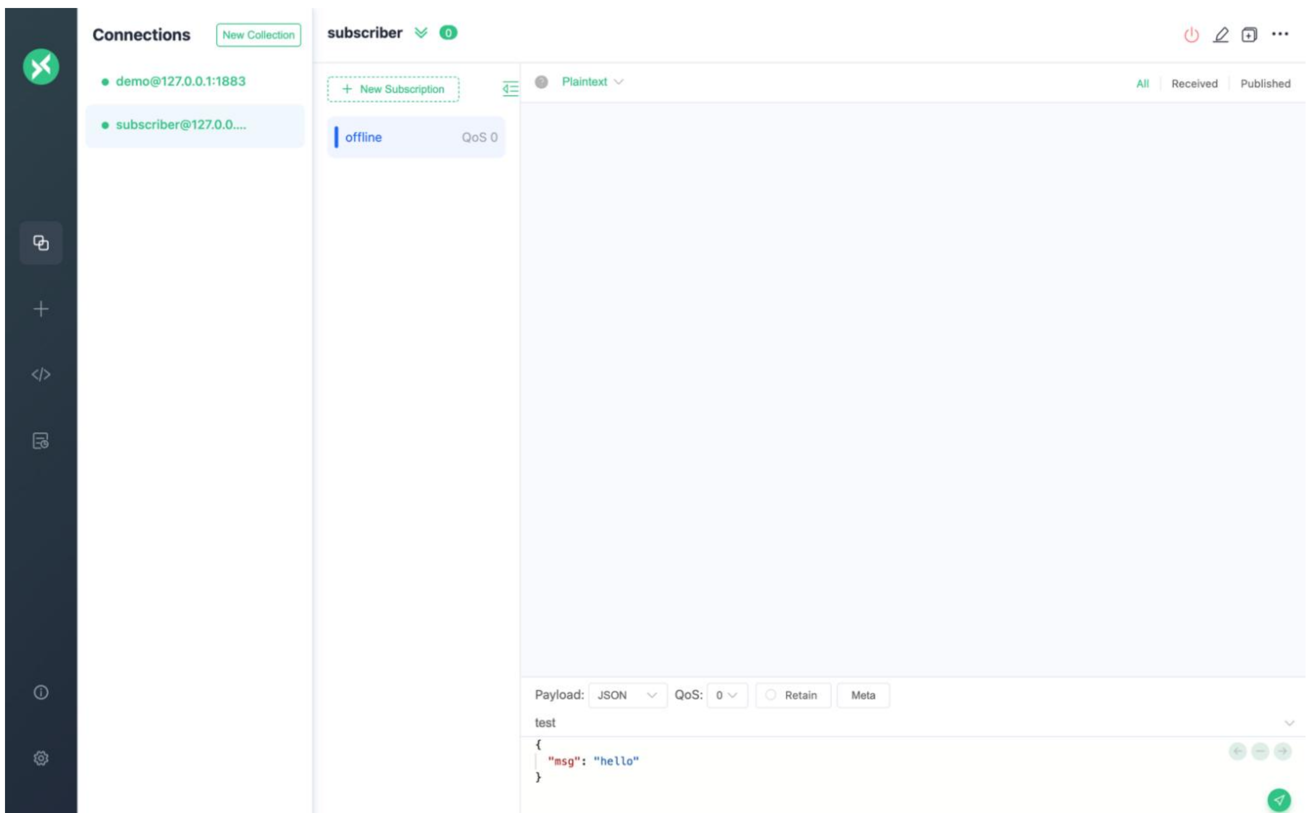
接下来，我们在 MQTT X 中新建一个名为 demo 的连接，Host 修改为 localhost，在 Advanced 部分选择 MQTT Version 为 5.0，并且将 Session Expiry Interval 设置为 10，确保会话不会在遗嘱消息发布前过期。

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。

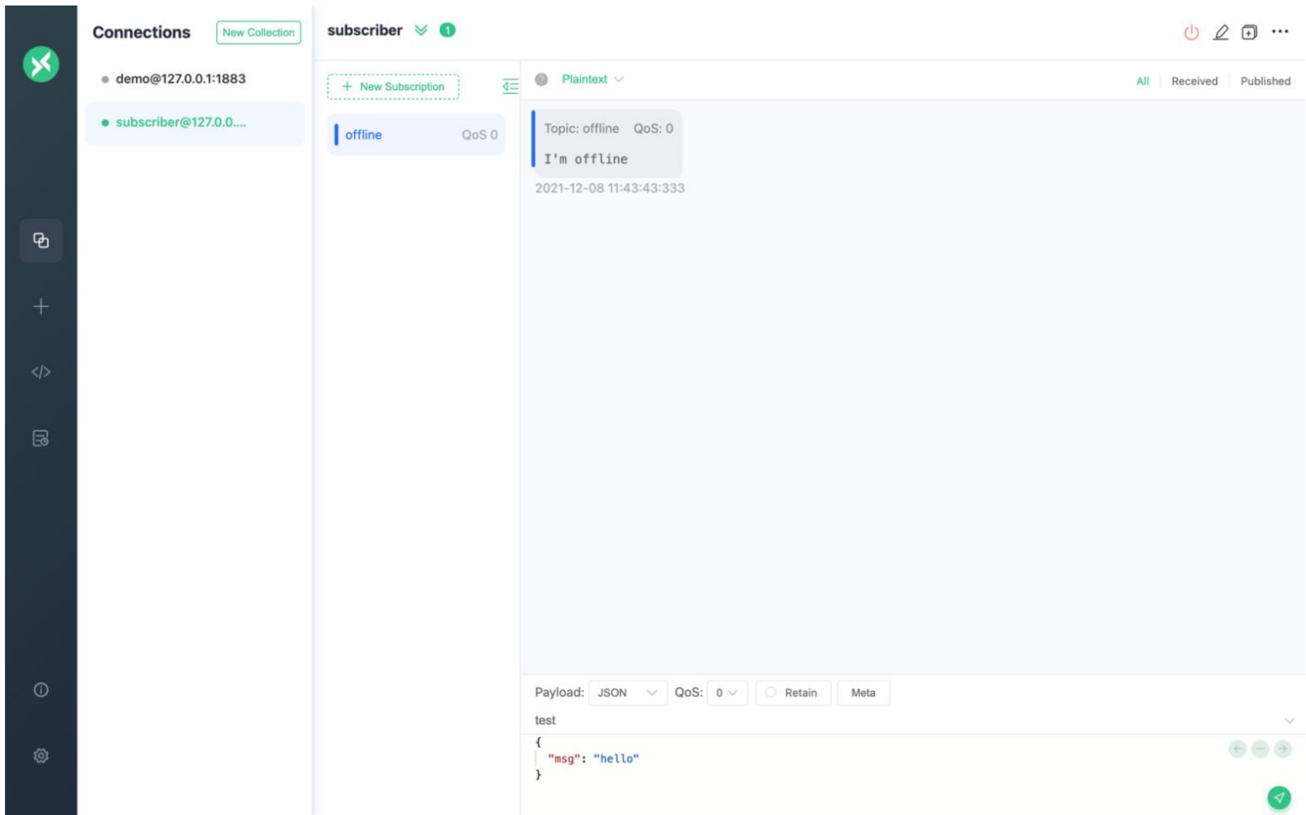


完成以上设置后，我们点击右上角的 Connect 按钮以建立连接。

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。

MQTT 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

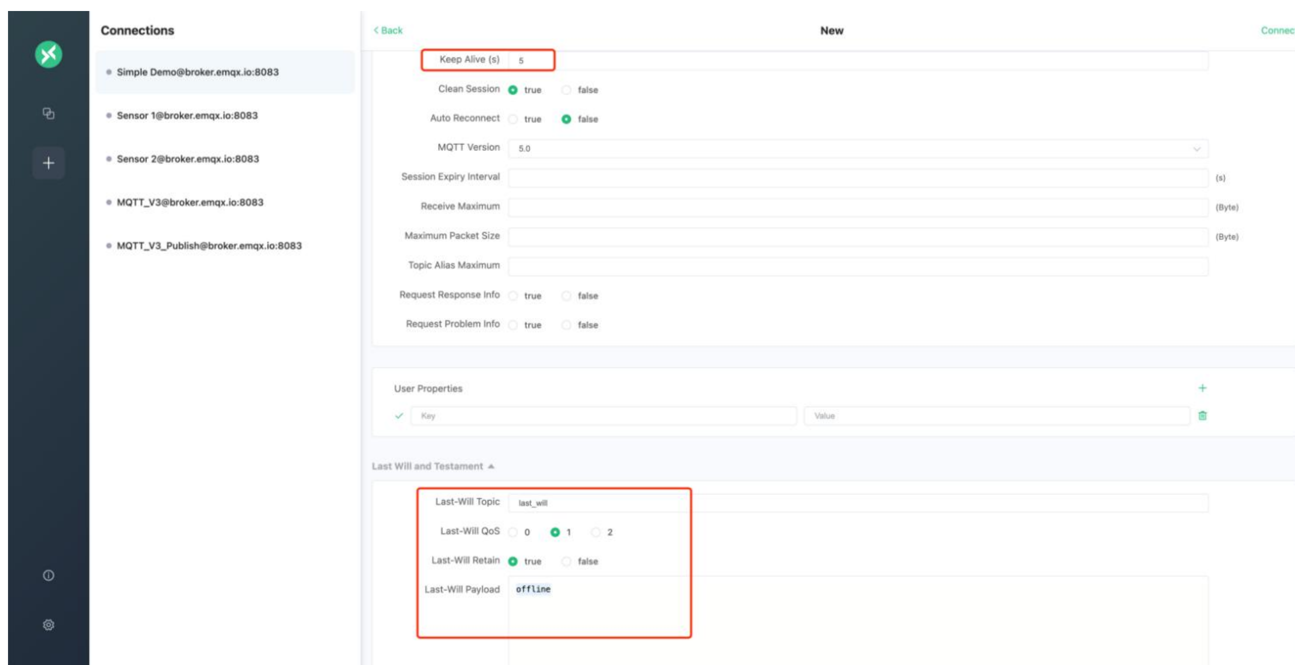
当 Broker 里存在半连接时，如果对应的客户端发起了重连或新的连接，则 Broker 会启动客户端接管机制：关闭旧的半连接，然后与客户端建立新的连接。

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。

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

©2013-2023 杭州映云科技有限公司版权所有