

下一代车联网设计与实现

从平台架构、数据采集到消息传输、安全通信的全方位技术手册



导读

目前，我国车联网行业处于与 5G 技术的深度融合时期。随着 5G 与 V2X 技术的发展成熟，未来的车联网产业必将打开新的成长空间。

车联网是物联网技术在交通系统领域的典型应用，车联网行业所涉及的相关技术领域的融合布局与协同发展，在某种程度上与物联网一脉相通。

作为一家开源物联网数据基础设施软件供应商，EMQ 多年来也为车联网领域的众多客户提供了云边端协同的物联基础设施软件，实现对人、车、路、云的统一连接，为整车制造商、T1 供应商、后市场服务商、出行服务公司等打造智能网联、自动驾驶和 V2X 等场景解决方案。

本白皮书旨在深入探讨和解决车联网领域中的关键问题，为车联网平台的设计与实现提供有力的指导。我们将根据 EMQ 在车联网领域的实践经验，从协议选择等理论知识，到平台架构设计等实战操作，与读者分享如何搭建一个可靠、高效、符合行业场景需求的车联网平台。

目录

01 车联网场景中的 MQTT 协议	1
MQTT 协议适合车联网吗？	1
相比于 MQTT，其他协议差距在哪里？	2
如何选择 MQTT 消息接入产品/服务？	3
常用技术方案	5
小结	7
02 千万级车联网 MQTT 消息平台架构设计	8
车联网的基础：数据采集与传递	8
千万级车联网消息平台架构设计	10
千万级消息接入测试	14
小结	18
03 TSP 平台场景中的 MQTT 主题设计	19
车联网 TSP 场景中对消息通道的需求	19
什么是 MQTT 协议的主题	20
基础概念	20
车联网 TSP 平台主题设计原则最佳实践	23
MQTT 协议主题设计在车联网场景中的应用	25
以 EMQX 进行车联网 TSP 平台主题设计	27
小结	29
04 QoS 设计：车联网平台消息传输质量保障	30

MQTT 协议中的 QoS 等级	30
车联网场景中的消息 QoS 设计	32
EMQX 基于 QoS 等级的消息传输保障	34
小结	34
05 车联网平台百万级消息吞吐架构设计	35
车联网场景消息吞吐设计的关联因素	35
EMQX+Kafka 构建百万级吞吐车联网平台	36
EMQX+InfluxDB 构建百万级吞吐车联网平台	40
小结	44
06 车联网通信安全之 SSL/TLS 协议	45
车联网安全通信 MQTTS 协议	46
构建安全认证体系典型架构	51
MQTTS 通信中单、双向认证的配置方式	52
常见 TLS 选项介绍	54
小结	58
07 国密在车联网安全认证场景中的应用	59
国密的分类	59
密码（GmSSL）证书与传统 SSL 证书对比	61
国密算法在车云通信中的应用	63
EMQ 基于国密算法的传输加加密认证集成方案	63
小结	74
08 实现车联网灵活数据采集	75

如何实现灵活数采	76
灵活数采方案剖析	77
更多可能	84
小结	85
09 车联网移动场景 MQTT 通信优化	86
移动设备网络迁移问题	86
车联网移动场景下的 MQTT 连接	88
如何改善移动网络下 MQTT 连接稳定性？	91
小结	95
10 MQTT over QUIC：下一代车联网消息传输标准协议	96
什么是 QUIC 协议	96
QUIC 协议的基本特性	96
MQTT over QUIC 介绍	97
MQTT over QUIC 与 MQTT over TCP/TLS 对比	98
MQTT over QUIC 如何优化车联网移动通信	100
MQTT over QUIC 在车联网中的更多应用	101
EMQX：首个实现 MQTT over QUIC 的 MQTT Broker	101
小结	103
11 云原生赋能智能网联汽车消息处理基础框架构建	104
传统车联网平台构建的挑战	104
云原生技术赋能新一代车联网消息处理	105
基于 Operator 的 EMQX 云原生框架	106

EMQX 在车联网场景中的云原生实践	108
小结	110
结语	111

01 | 车联网场景中的 MQTT 协议

MQTT 协议早已是物联网领域当之无愧的主流协议，其凭借轻巧高效、可靠安全、双向通讯等特性在诸多行业物联网平台搭建中得到了广泛的应用。那么，MQTT 协议在车联网场景中的应用情况是怎样的呢？

在本白皮书的第一章节，我们将从 EMQ 车联网行业用户实际案例经验出发，对比不同物联网通信协议在车联网平台搭建应用中的利弊，分享企业该如何基于自身情况选择合适的 MQTT 消息接入产品与服务，以及当前在数据传输安全、数据集成等方面的最热门技术方案。

MQTT 协议适合车联网吗？

整个车联网业务架构复杂，涉及多个通信环节，在本文中我们讨论的是车联网平台主要负责的云-端消息接入模块。

MQTT 是基于发布/订阅模式的物联网通信协议，具有简单易实现、支持 QoS、报文紧凑等特点，占据了物联网协议的半壁江山。

在车联网场景中，MQTT 依然能够胜任海量车机系统灵活、快速、安全接入，并保证复杂网络环境下消息实时性、可靠性，其主要应用优势如下：

1. 开放消息协议，简单易实现。市场上有大量成熟的软件库与硬件模组，可以有效降低车机接入难度和使用成本；
2. 提供灵活的发布订阅和主题设计，能够通过海量的 Topic 进行消息通信，应对各类车联网业务；
3. Payload 格式灵活，报文结构紧凑，可以灵活承载各类业务数据并有效减少车机网络流量；
4. 提供三个可选的 QoS 等级，能够适应车机设备不同的网络环境；
5. 提供在线状态感知与会话保持能力，方便管理车机在线状态并进行离线消息保留。

综上，如果配以具备海量车端连接、软实时、高并发数据吞吐以及多重安全保障能力的消息中间件产品，MQTT 协议无疑将为车联网平台的搭建带来极大便利。

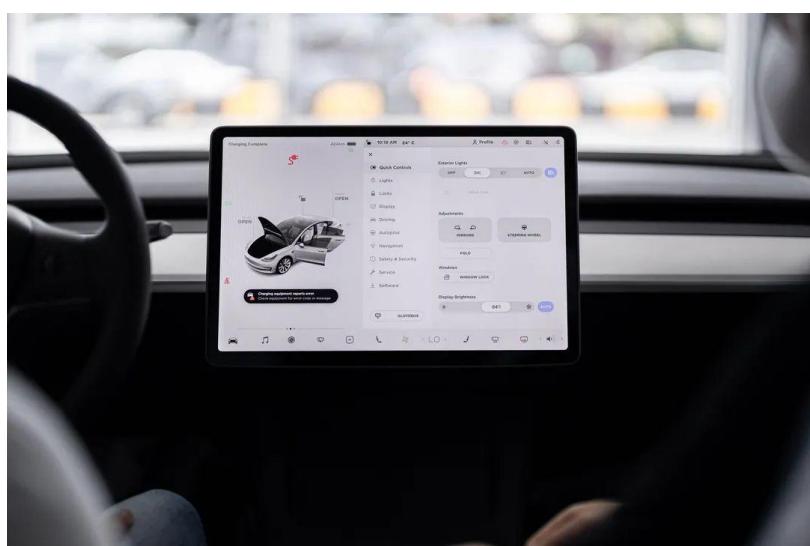
相比于 MQTT，其他协议差距在哪里？

目前为止大多数车联网客户首选的都是 MQTT 协议，我们也遇到过一些客户曾选择其他诸如私有 TCP、HTTP 协议，但从最终结果来看，MQTT 都是车联网场景下的最佳选择。

在没有接触过 MQTT 协议之前，华南某大型主机厂采用了私有化的 TCP 协议（ACP 协议）构建车联网服务平台。经过长周期的协议规范设计和开发，基本实现了车联网平台的主要功能。

但随着车联网业务场景的不断增加和车机数量的不断增长，私有化的 TCP 的弊端逐渐凸显：协议私有化定义与版本维护困难、所有的协议功能（如保活、断线重连、离线消息等）都需要定制开发，私有的协议也导致终端硬件适配都需要定制开发……**成本高、周期长，更新迭代慢等问题突出。**

随着 MQTT 协议生态不断完善和在车联网平台通讯协议选型中被广泛采用，该主机厂在新一代车联网平台的开发中开始采用 MQTT 协议，基于 EMQX 物联网接入平台为其提供的完善 MQTT 协议支持，不仅降低了开发成本、缩短了开发周期，同时实现了更多的功能场景和运维手段。



华东某大型主机厂现有一百多万的存量车机，之前的车联网平台采用私有的 TCP 协议构建，面对百万车机海量的消息通信，**私有化的 TCP 协议维护成本高，消息可靠性无保障，日常系统维护和功能扩展开发工作量大。**

随着 MQTT 协议在集团内部车联网平台广泛采纳，该主机厂也开始启动 MQTT 协议的改造升级工作，目前针对部分车型已经通过 OTA 升级的方式完成了升级，未来他们计划分阶段逐步完成所有车型的升级改造工作。

还有一个车企客户早期与我们接触过，但考虑到初期业务比较简单以及自身技术选型问题，最终使用了自建 HTTP 服务的形式接入车机。

随着业务发展，传统的请求-响应模式通信已经无法满足新增业务需求，同时随着功能与终端数量增多，整个平台通信量成倍增加，**使用 HTTP 接入出现了性能瓶颈**。该客户最终还是选择了 MQTT 作为接入协议，使用 EMQX 提供的数据接入方案很好地解决了之前的业务难题。

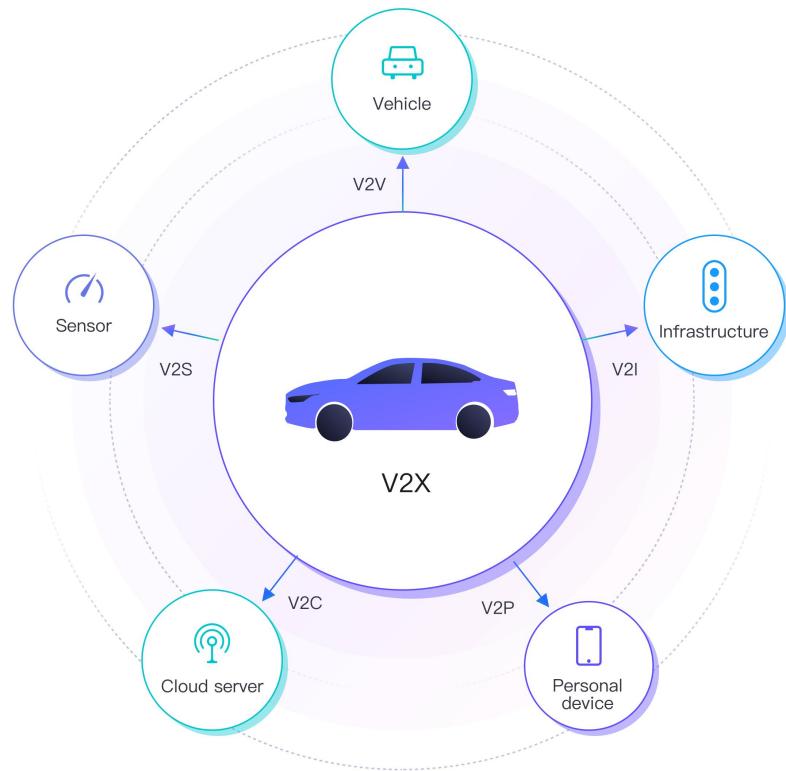
总体来看，私有协议具有封闭性、排他性等特点。其在制定初期即是为解决特定问题而设计，导致缺乏灵活性，在业务调整后往往难以满足新的需求，企业不得不在协议中加入更多的特性；又或者因为接入量的增多，私有协议 Server 端过早达到了性能和扩展性的瓶颈。

以上种种原因最终导致用户的工作重心从业务开发转移到接入层、中间件的开发，无形中增加了平台项目成本。由此，MQTT 协议顺理成章地成为了最适合车联网领域的主流协议。

如何选择 MQTT 消息接入产品/服务？

平台设计中，系统架构设计与产品选型是一个严谨的过程。用户首先要结合应用场景，评估产品功能是否满足业务需求，性能与可扩展性能否能够支撑平台短期的设计容量以及未来可能的增长；产品使用成本也是一个重要的考量，产品本身的成本、IaaS 基础设施、开发集成和维护工作这些都会影响客户的总体拥有成本；此外还应当结合产品全球化能力进行评估，对于有海外业务的项目，产品能否支持全球部署、是否满足各个地区的合规性、能否避免云计算提供

商锁定这些都是选择产品的依据。



EMQ 的车联网客户在选型过程中经常会与云计算提供商的物联网消息接入 SaaS 服务进行对比。相比之下 **EMQX 的优势主要在于私有部署和标准化能力** —— 支持私有部署到任意云平台，无平台锁定，提供标准 MQTT 协议，这也是车联网客户普遍看重的一点。

避免云计算供应商锁定有助于企业用户获得竞价优势，可以减少企业与云计算供应商中止合作关系而带来的影响。另一方面多云支持也可以充分使用不同云计算提供商的技术与商业优势，比如一些全球运营的企业可以在国内和海外选择不同的云计算提供商。

此外还有相当一部分客户看中 EMQX 较低的使用成本，根本原因是由于计费方式不同，往往业务规模越大，云计算提供商的接入服务成本越高。

不过从成本考量也有例外。之前曾有车企用户评估了云提供商的接入服务和 EMQX 后，考虑到自身的运维成本和风险后最终放弃私有部署，选择了云提供商的接入服务。

随着 EMQX 的全托管 MQTT 消息服务 EMQX Cloud 的上线，这个问题也已得到解决。通过

EMQX Cloud，现在用户可以在免除基础设施管理维护负担的同时，保持了同私有部署一致的成本预算清晰可控、跨云跨平台等优势，没有后顾之忧地开展车联网平台建设。

对于有私有部署需求的客户，EMQX 也有其独有的优势。EMQX 提供全球性的商业支持，较高的产品性能可带来海量连接和吞吐能力，规则引擎与数据桥接则提供快速集成能力。同时针对车联网领域，高可靠和易扩展架构能力以及云边一体的 V2X 信息交互能力也让 EMQX 在支持私有部署的同类产品中脱颖而出。



2018 年上汽大众在设计研发新一代车联网系统时，SC 部门考虑到新型车联网大并发、低时延、高吞吐的场景需求，参考了国内外主流新型车联网系统架构，最终采用了基于 MQTT 协议建设新一代车联网平台。

该项目中，MQTT 的特性与 EMQX 强大规则引擎数据集成能力和通用型总线能力有效满足了客户对复杂网络下消息实时性可靠性的要求，并解决了项目时间紧、任务重，需要快速开发对接的需求。

常用技术方案

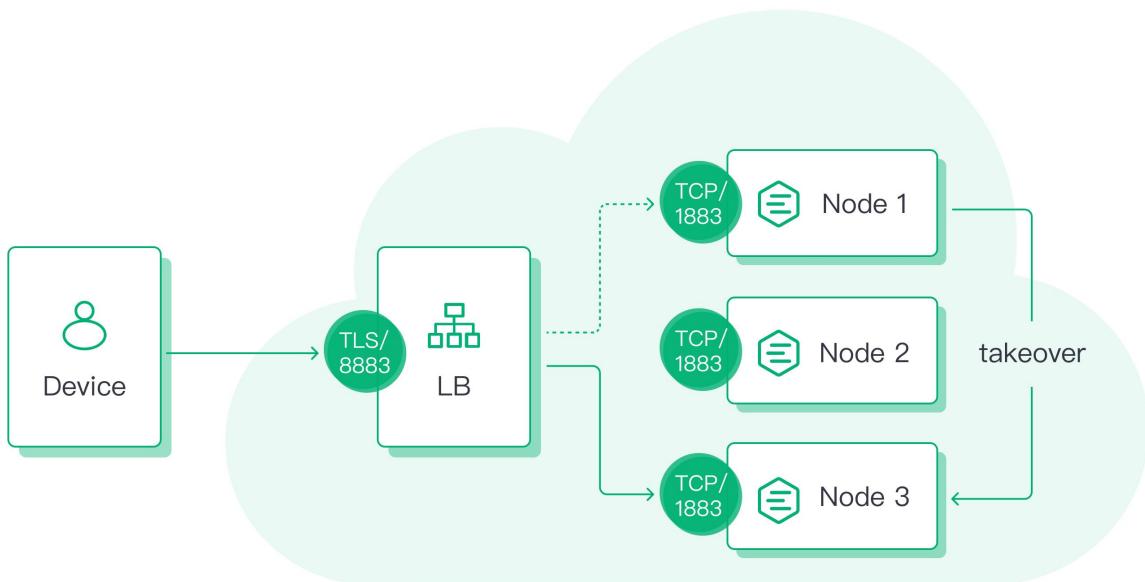
作为消息中间件，EMQX 提供了丰富且灵活的集成能力，且每个功能都提供了不同的技术方案以供用户选择，经过长期使用总结，比较热门的技术方案如下：

安全保障

在传输链路层上我们均推荐用户启用 TLS 加密传输，但是多数云计算提供商的负载均衡产品不支持 TLS 终结，生产部署时需要额外部署 HAProxy 等组件来卸载 TLS 证书。另外有部分客户需要国密算法 TLS 加密传输，我们也专门定制提供了方案。

TBox 接入最常见的是使用证书认证，EMQX 提供可扩展的认证链，支持第三方认证平台扩展（如 PKI 系统），基于用户名/密码的外置数据源和内部数据库认证其次。

此外绝大部分用户都启用了 EMQX 鉴权功能，为不同 TBox 终端分配相应的发布、订阅权限以有效保护数据安全。

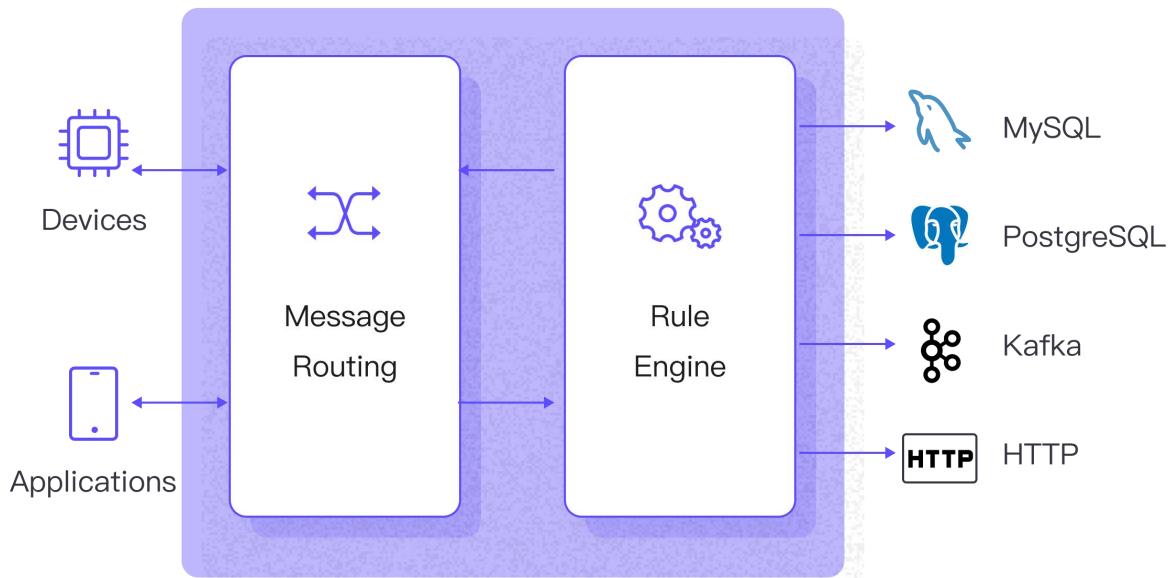


数据集成

将流经 EMQX 的海量车联网数据与业务系统连接是客户最重视的能力，EMQX 内置了规则引擎和数据桥接能力，可以将 MQTT 数据流式传输到 Kafka、各类 SQL / NoSQL / 时序数据库中，而实际项目中绝大多数客户都使用 Kafka 作为后端流处理组件。

Kafka 专注于数据的存储和读取，而 EMQX 则侧重于客户端和服务器之间的通信，EMQX 用来快速接收和处理来自大量物联网设备的消息，Kafka 可以收集并存储这些数据并将其发送给

后端程序来分析和处理，这个架构是目前应用最广的数据集成方案。



小结

目前汽车电子发展迅速，助推车联网行业技术升级，智能交通及汽车行业未来市场前景可观，可以预见有更多的车主消费者和汽车厂家将从中受益。基于完善的 MQTT 协议和 EMQX 强大的产品能力可以帮助车联网平台开发者快速构建健壮、灵活的车联网平台。

02 | 千万级车联网 MQTT 消息平台架构设计

随着整个汽车出行领域新四化（电气化、智能化、网联化和共享化）的推进，各个汽车制造商正逐步构建以智能驾驶和智能网联为核心的车联网系统。新一代的车联网系统对于底层消息采集、传输和处理的平台架构提出了更高的要求。

前一章节中我们已经提到，MQTT 协议是目前最适合车联网场景数据平台搭建的通信协议。基于此，我们将继续讨论车联网场景中的 MQTT 消息采集与传递，以及如何构建一个千万级车联网 MQTT 消息平台，以期为正在进行车联网业务的企业用户提供平台架构设计参考。

车联网的基础：数据采集与传递

车联网传输协议的演进

众所周知，车联网（vehicle-to-everything，V2X）是指车与云、车与网、车与车、车与路、车与人、车与传感设备等交互，实现车辆与公众网络通信的动态移动通信系统，是为了满足与车有关的每一个环节中的效率、安全、管理等元素而建立起的异构通信网络。而运行于其中的通信协议就成为车联网系统建设的关键和核心。



在车联网发展的历程中，主要有两种主流的通信技术，对车联网整体发展起到了推动作用：

DSRC(DeDICated Short Range CommunICation, 专用短程通信): 1992 年由美国材料试验学会 ASTM 针对 ETC 的业务场景研发而出，后经多年完善和迭代，演变为 IEEE(802.1X) 车联网通信技术标准。在相当长的一段时间里，DSRC 技术是国际汽车主流生产和消费市场使用的主流车联网通信协议。

C-V2X(Cellular Vehicle to Everything, 蜂窝车联网通信): C-V2X 依托现有的蜂窝基站，除了支持 PC5 的直连通信，RSU、车辆均可通过 4/5G 信道（采用 Uu 接口）与 V2X 平台相连，实现车路协同通信。较之 DSRC，C-V2X 技术上更优，它增强通信的安全性与保密性，支持高网络容量，可支持高带宽和大数据量需求。

DSRC 和 C-V2X 技术的竞争非常激烈，两者都希望能够成为主流车联网通信标准。目前，我国拥有最完善的 5G 通信网络的基础设施，因此更倾向于采用 C-V2X(LTE-V、5G-V2X)通信技术，通过 V2X 车路系统+单车智能系统的体系化建设，实现基于自动驾驶的新一代车联网架构。

消息平台建设对于车联网的意义

在车联网建设高速发展的今天，所有的主机厂已形成了一个共识：**车联网建设的目的不是为了联网而联网，也不是为了车载娱乐而联网，联网是为了数据。**有了车联网，就有了数据。有了数据，辅以完整的数据治理和应用体系，就有了一切。

而这个业务的目标数据，也仅仅限于车端的相关数据。在 V2X 框架中，需要解决车与车(V2V)、车与路(V2R)、车与网(V2I)、车与云(V2C)、车与人(V2H)等的互联互通，实现针对车、路、云、网、人的全面数据采集和分析。基于 5G 的 C-V2X 协议和通讯方式，为整个系统的建设提供基础能力保障。

从传统的 OTA 应用到智能座舱、高精地图适配、厘米级定位、车机端长连接、手机端消息采

集、车路云图、车路协同等众多新型智能应用场景，车联网业务对于消息平台和数据处理系统的需求已从原始的车云扩展为人-车-路-网-云的整体架构建设，也因此对整个消息平台的建设提出了更高的要求。

如何建设一个海量连接、高并发吞吐、低时延的消息通信和传输系统架构，来保证整个系统的泛在性、便利性、高可用性、可靠性、安全性和高并发性，就成为了基于自动驾驶和车路协同场景下新一代车联网系统建设的关键所在。

千万级车联网消息平台架构设计

接下来我们将以 EMQ 的车联网消息平台和数据处理整体解决方案为例，介绍如何构建一个千万级的车联网消息平台。



业务挑战

- 车机、路测单元和手机端系统安全接入

车端需要涵盖车机数据上报、POI 下发、推送文件、下发配置、推送消息、运营关怀等全新车联网业务，产生的海量消息 Topic 需要更加安全稳定的接入与传输实现消息订阅和发布。路端需要实现路测 RSU 的安全接入，消息采集和传输、地图数据的传输等。

- 大并发消息传递的实时性和可靠性

高精地图、厘米级定位、车路协同等应用场景均需要解决海量车路图消息的毫秒级低延时和高可靠的传输能力保障，需要消息处理平台具备高性能、低延时和高可靠支持千万连接和百万并发业务场景的能力。

- 丰富的应用场景集成

在以自动驾驶为核心的车联网系统中，需要使用消息平台对接各种基于人、路、图、云相关的应用对接。将车端数据通过消息平台同高精地图、厘米级定位、车路协同、手机端连接等应用场景进行连接，通过消息平台保障应用的消费供给，并提供高性能、低延时和高可靠的数据架构。

- 海量数据存储、处理和分发

来自于人、车、路、云、图、网的海量物联网数据被采集后，需要针对这些大规模实时数据流的接入、存储、处理、分发等环节进行全生命周期管理，为应用提供针对动态连续数据流的数据库支撑，支持应用深度使用车联网数据服务于消费者，进行商业决策。

整体解决方案

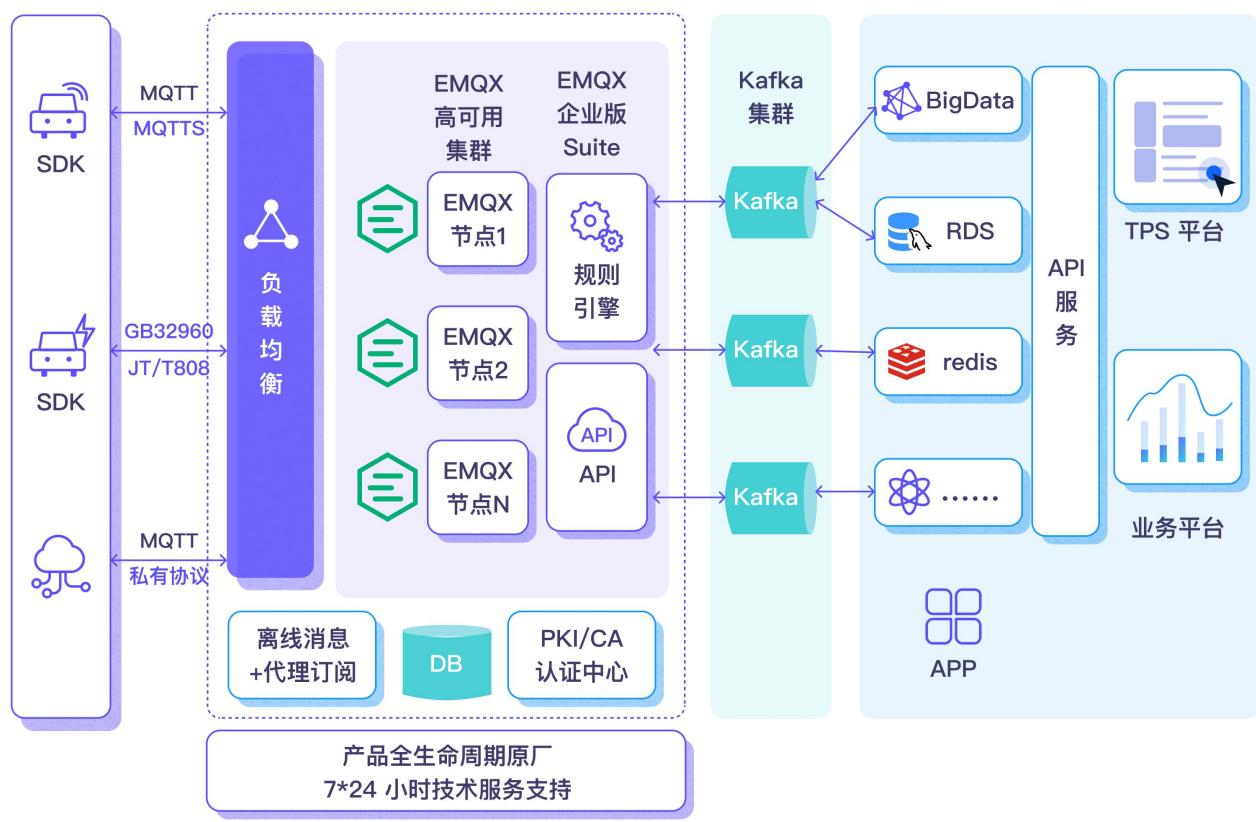
在方案中我们主要采用 EMQ 旗下的云原生分布式物联网接入平台 EMQX，实现车联网系统中车端和人、路端的数据连接、移动和处理。EMQX 一体化的分布式 MQTT 消息服务和强大的 IoT 规则引擎，可为高可靠、高性能的物联网实时数据移动、处理和集成提供基础能力底座，助力企业快速构建关键业务的 IoT 平台与应用。

- 针对车端的消息处理

EMQX 采用 MQTT 协议接入车联系统。车机端通过负载均衡与 EMQX 分布式集群进行连接，EMQX 的横向扩展能力可实现千万级车机连接和百万并发响应的数据通信能力。通过规则引擎，可一站式实现海量消息桥接消息队列、持久化入库、离线消息存储等能力，同时提供

丰富的 API 原子能力北向集成。

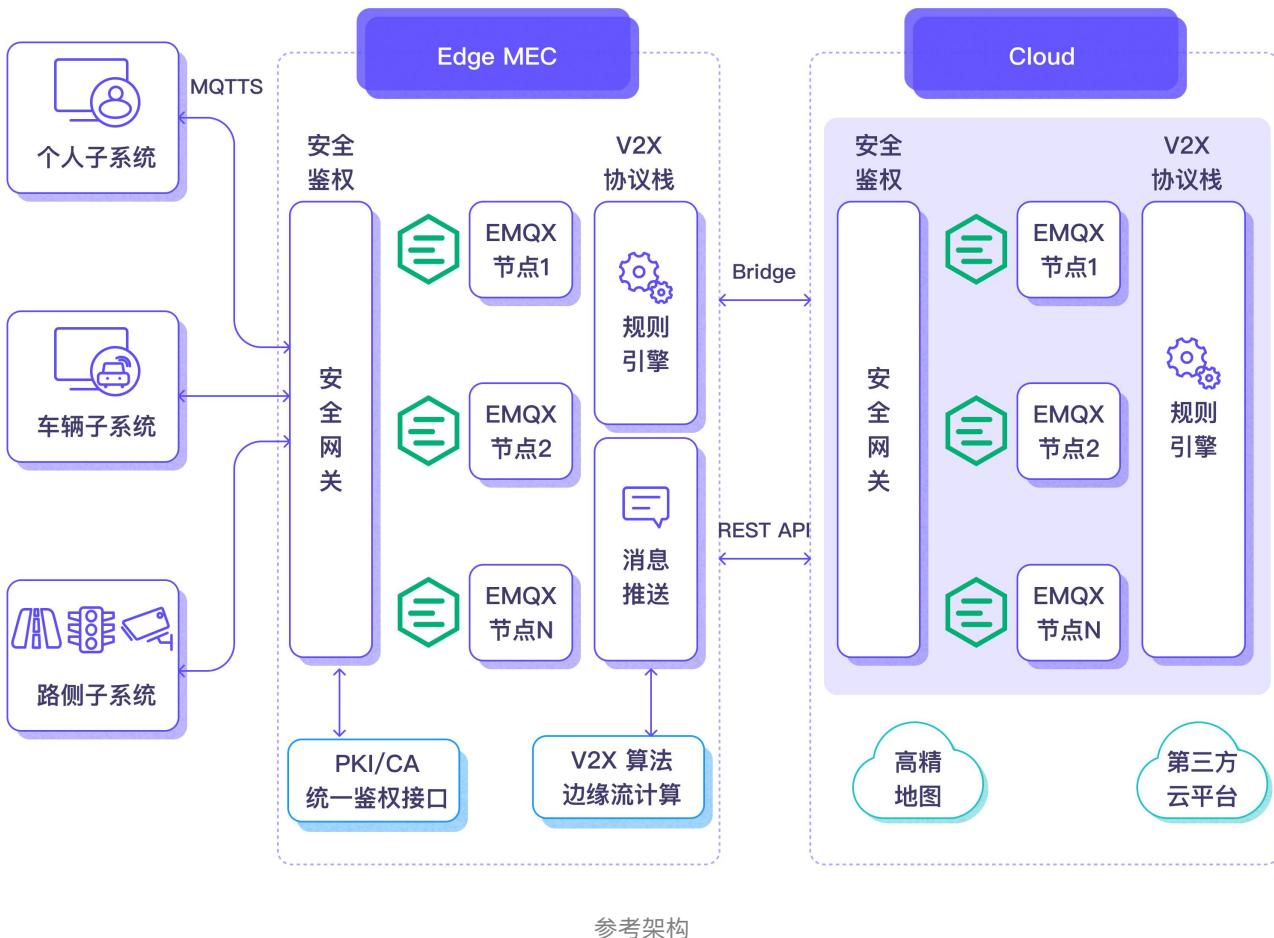
在安全方面，EMQX 不仅支持 TLS/DTLS 或国密 GMSSL 安全协议，保障系统可靠与稳定；还提供心跳监测、遗嘱消息、QoS 等级等多重保障机制，通过离线消息存储实现在复杂的网络环境下实时、安全、可靠的车机消息通信。



- 针对人、路端的消息处理

EMQX 为人、路端提供针对手机 APP、RSU 等终端的消息采集和处理平台。基于 5G 网络切片能力，通过个人终端和路侧单元的就近接入，实现超低时延的交通信息服务。通过 MQTT 等协议将人端、路侧设施感知到的路况信息推送到云控平台，通过云控平台融合 V2X 算法实现道路协同感知、安全提醒、远程协同控制等智能交通场景。

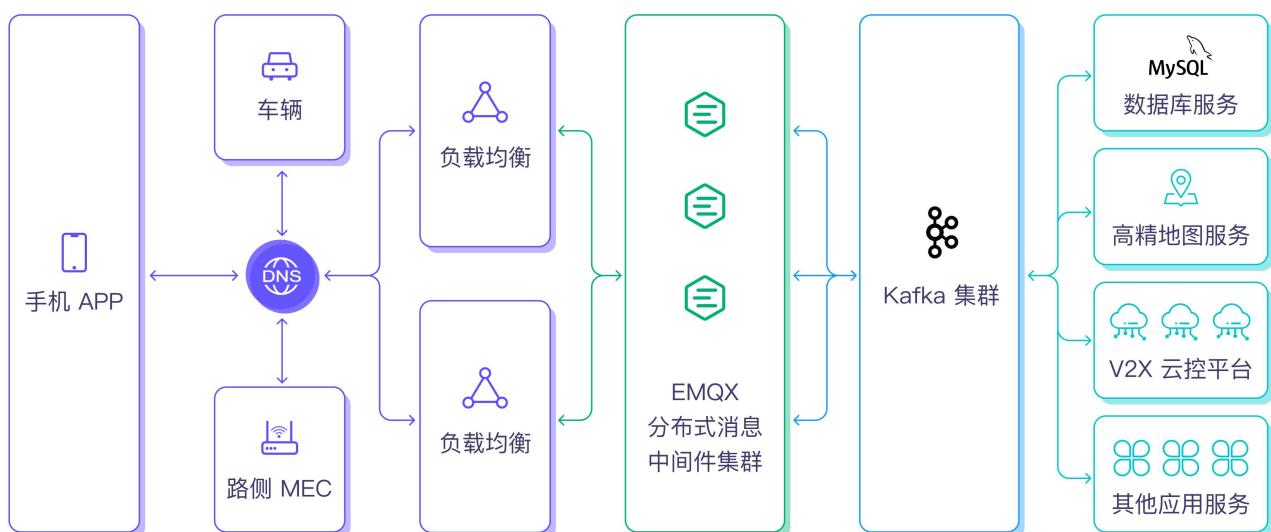
在安全方面，支持国际标准的 TLS/DTLS 加密或国密算法 GMSSL 加密，通过扩展基于 PKI/CA 证书认证体系保障人车路信息系协同的安全通信。



参考架构

- 千万消息接入框架模型

针对新一代车联网场景，EMQ 千万级连接规模和百万级并发的整体消息接入和数据处理平台
参考架构如下：



- **业务场景：**车联网体系中的车辆、手机 APP 端、路侧 RSU 等设备等通过 MQTT 接入，实现对千万量级的以上终端的并发接入能力。
- **系统架构：**终端设备通过 MQTT、HTTP 等协议接入，经过负载均衡组件连接至分布式消息平台 EMQX。通过分布式多集群部署满足千万并发连接需求，按照百万级消息吞吐能力，通过规则引擎对接 Kafka 集群实现数据的转发。车联网服务平台、高精地图服务、V2X 云控服务、定位服务和其他车辆网相关应用可以直接通过订阅 Kafka 数据进行消费，同时 EMQ 提供了 REST、MQTT 和 MQ 消息队列三种南向接口服务实现对车控（远程控制）消息的双向通信。

通过以上参考框架，EMQ 凭借 EMQX 云原生分布式物联网接入平台可实现车联网场景下的千万连接、百万并发的业务需求。

千万级消息接入测试

测试环境和目的

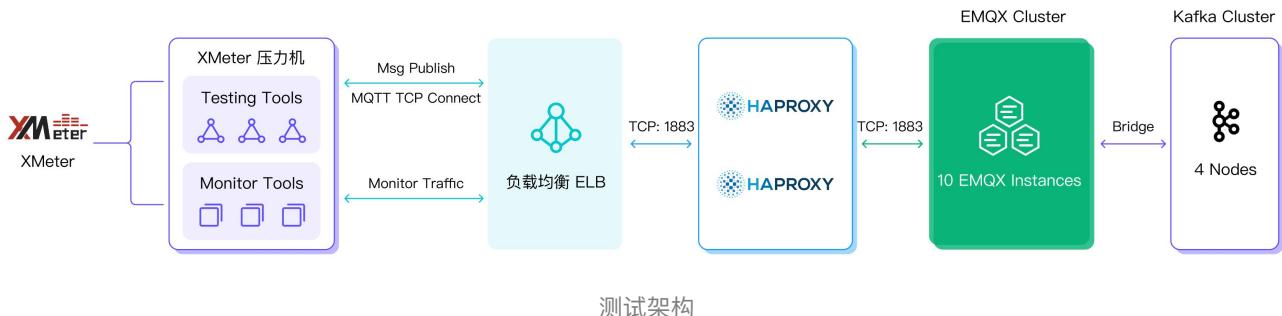
某车企计划在车联网场景下，基于测试环境验证 EMQX 集群的以下能力，为后续业务增长做相应的技术架构和能力支撑准备：

- 可支撑 1000 万并发连接，同时支持每秒 10 万~15 万、payload 为 100 字节的 QoS 0 消息通过规则引擎桥接到 Kafka；
- 1000 万并发连接订阅、消费 OTA 广播主题；
- 300 万用户同时连接不会造成集群雪崩，并测试连接所需时间。

另外，在完成上述所有测试后，继续探索在目前配置下 1000 万并发的同时可支持的最高消息发送并桥接转发至 Kafka 的吞吐量(根据 EMQX 集群资源使用情况提高客户端消息发送频率)，以及测试在 1000 万连接下满足 QoS 为 2 且平均响应时间在 50 毫秒内的最高消

息吞吐。

测试准备



测试架构

客户端通过 TLS 加密连接负载均衡 ELB，然后在 HAProxy 对客户端进行 TLS 终结，最后通过 TCP 连接至 EMQX 集群。通过在 HAProxy 上终结 TLS 的方式可以提高 EMQX 集群的支持能力，在这种部署模式下 EMQX 的处理能力和客户端直接通过 MQTT TCP 连接是完全一致的。另一方面，相比 MQTT TCP 连接，客户端通过 TLS 连接也需要消耗更多的资源，而本次测试规模为千万级，所需的测试机数量众多，为了减少所需测试资源的同时不影响对 EMQX 集群的测试目标，本次测试将直接使用 TCP 连接。

服务	数量	版本	OS	实例规格	CPU	RAM	网卡	端口
负载均衡云服务	1			网络型/大型II				18083/1883 /8081
EMQX节点	10	V4.3.4	Centos7.8	c6.16xlarge.2 普通块存储	64C	128G	1	18083/1883 /8081/8883
KAFKA云服务	4	2.3.0	Centos7.8	c6.4xlarge.2 超高IO块存储	16	32G	1	9092
XMeter压力测试控制节点	2	3.0	Centos7.8	c6.4xlarge.2	16	32G	1	443/80 /3000/8086 内网放通
XMeter压力测试节点	43	3.0	Centos7.8	c6.4xlarge.2	16	32G	5	内网放通

测试场景

序号	场景名称	描述	期望结果
1	千万连接+消息吞吐	1000 万 MQTT TCP 并发连接, 心跳间隔 200s。其中 700 万为背景连接(只连接不发送消息), 300 万活跃用户, 每个用户每隔 15S 上报一条 QoS 0 的消息, payload 为 100B。消息通过规则引擎桥接到 Kafka。先测试 1 小时, 通过后进行 24 小时稳定性测试	内网测试成功率为 100%, 无消息积压, CPU 和内存存在测试期间表现平稳, 没有大幅度的抖动。
2	消息广播	1000 万 MQTT TCP 并发连接, 所有连接均订阅同一个 OTA 广播主题(QoS 0, payload 为 100B)。模拟一个 MQTT 客户端每隔 10 分钟向该主题广播一条消息, 测试 30 分钟	内网测试成功率为 100%, 所有订阅客户端成功消费 3 条消息
3	300 万并发瞬时连接	300 万 MQTT 客户端同时发起连接, 测试所有连接完成所需时间	300 万客户端都成功连接, 集群不会雪崩
4	1000 万连接下最高消息吞吐探索	现有配置及 1000 万连接且桥接 kafka 下可达到的消息最高吞吐率 (Qos 0, payload 100B/1kB)	最高消息吞吐率达到后测试 2 小时, 内网测试成功率为 100%, 无消息积压, CPU 和内存存在测试期间表现平稳, 没有大幅度的抖动
5	平均响应时间下的 TPS	1000 万连接下, 消息为 QoS2、payload 100B, 平均响应时间在 50 毫米内支持的最高消息 TPS	能够达到不少于 20 万 TPS 的吞吐能力

测试结果

以下是本次测试的结果呈现：

序号	场景	平均响应时间	EMQ X 节点 CPU 使用率	EMQ X 节点 CPU IDLE	EMQ X 节点内存使用(G)	LB所需带宽(MB)
1	1千万连接+20万消息吞吐, QoS 0, payload 100B	1.5ms	31%~48% 平均 47%	37%~54% 平均 47%	Used: 27.7~42 Free: 78.2~92.5	45
2	1千万连接下的消息广播	100ms	最高 21%	最低 69%	Used 最高 32.3 Free 最低 87.9	200
3	300万客户端瞬时连接	3分钟完成连接	最高 25%	最低 63%	Used 最高 14.7 Free 最低 108.2	400
4	探索最高吞吐: 1千万连接+120万消息吞吐, QoS 0, payload 1kB	164.3ms	23%~64% 平均 46%	20%~64% 平均 43%	Used: 33~38 Free: 81.3~87.1	1350
5	1千万连接+QoS2 20万消息吞吐, payload 100B	51.4ms	3%~51% 平均 41%	31%~53% 平均 43%	Used: 22.2~29 Free: 91~98	95

如以上结果所示，在目前的部署架构下，可以满足该车企对于千万并发连接 +20 万消息桥接至 Kafka、消息广播及 300 万瞬时并发连接的验证需求。在探索测试中，1000 万连接下测试到最高 120 万消息 TPS(QoS 0、payload 1kB)，测试持续 10 小时 EMQX 集群稳定，CPU idle 最低至 20%，内存使用平稳。

由以上可知，EMQX 在车联网场景下支持千万连接性能表现突出，架构稳定可靠。

压力测试工具简介和使用

本次测试由于所需测试机数量多，管理复杂，故使用 EMQ 旗下商业版测试软件 XMeter 性能测试平台和 JMeter-MQTT 插件进行。

XMeter 是基于开源测试工具 JMeter 扩展的性能测试平台。针对物联网具有的接入规模大、弹性扩展要求、多种接入协议、混合场景等特点，XMeter 对 JMeter 进行了改造，可以支持大规模、高并发的性能测试，比如实现千万级别的 MQTT 并发连接和消息吞吐测试。除了测试 MQTT 协议之外，还可以支持 HTTP/HTTPS 等主流的应用的测试。

JMeter-MQTT 插件是由 XMeter 实现的开源 MQTT 性能测试插件，在众多的项目中得到了使用，目前是 JMeter 社区中流行度最高的 MQTT 插件。

- XMeter 官网和试用地址: <https://www.xmeter.net>
- XMeter MQTT 插件下载: <https://github.com/xmeter-net/mqtt-jmeter>
jmeter/tree/master/Download/v2.0.2
- JMeter 下载地址: <https://jmeter.apache.org>

小结

通过本文，我们介绍了基于云原生分布式物联网接入平台 EMQX 的千万级车联网 MQTT 消息平台架构设计，并验证了该架构在千万级并发连接场景环境下的性能表现，为车联网系统的消息数据平台建设提供了一种可能的设计参考。

03 | TSP 平台场景中的 MQTT 主题设计

在车联网生态中，TSP（Telematics Service Provider）平台在产业链中居于核心地位，上接汽车、车载设备制造商与网络运营商，下接内容提供商，是主机厂车辆与服务的核心数据连接平台。

随着智能汽车的发展和车主用户对应用场景需求的不断提升，主机厂对 **TSP 平台的设备与应用承载能力需求将不断增加。**



在之前的章节中我们提到，在车载设备与 TSP 平台数据交互协议选择上，MQTT 以其轻量化、易扩展、多种消息质量保证（QoS），以及通过发布订阅模式实现数据产生与数据消费系统解偶等优势成为目前各大主机厂的新一代 TSP 平台的首选协议。

在这一章节，我们将介绍**在车联网 TSP 平台搭建过程中，如何进行 MQTT 消息主题设计。**

车联网 TSP 场景中对消息通道的需求

车联网 TSP 场景中，MQTT 协议作为「车-平台-应用」之间的业务消息通道，不仅要保证车与应用之间消息可以双向互通互联，而且需要通过一定规则将不同类型的消息识别与分发。而 MQTT 协议中的主题就是这些消息的标签，也可以看作是业务通道。

在车联网场景中，可以把消息分为从**车-平台-应用的数据上行通道**以及**应用-平台-车的数据下行通道**；对于车联网 TSP 平台，不同数据方向意味着不同的业务类型，需要通过 MQTT 主题进行明确的区分与隔离。

- 从车端角度看：

在 TSP 平台中**车辆数据上报**是上行数据的主要业务类型。

随着车联网业务的不断丰富，如 T-box 等车载系统计算能力与通讯能力不断增强，车辆数据上报的业务场景、数据量及频率也不断增加。基于业务隔离、实时性与安全等需求，从车联网早期的一车一主题逐渐向一车多消息通道发展。

- 从应用侧角度看：

平台应用作为车辆数据接收与消费方，同时也会作为数据下发，指令下发的消息发送方。根据业务需求不同，消息发送类型也可以分为：

- 一对多消息：针对一些如车控等关键业务与高安全性要求的业务，需要针对每辆车提供一对一的消息通道。
- 消息广播：针对大规模的消息通知，配置更新场景，可以向平台所连设备发送大规模的消息广播。

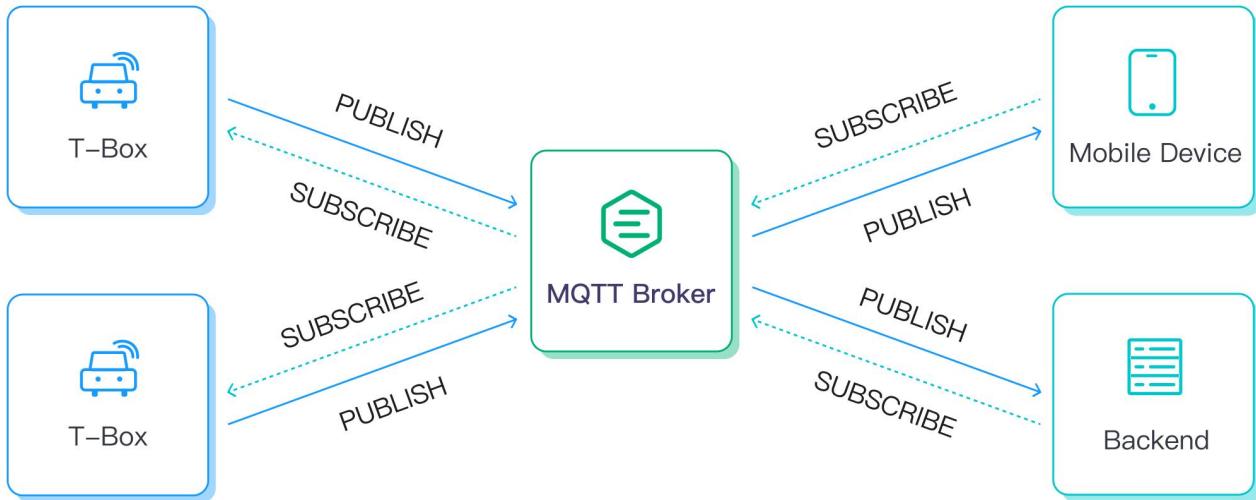
什么是 MQTT 协议的主题

基础概念

在 MQTT 协议通信机制中有三个角色：消息发布者（publisher）、代理服务器（broker）和

消息订阅者（subscriber）。

消息从发布者发送到代理服务器，然后被订阅者接收，而主题就是发布者与订阅者之间约定的消息通道。



发布者指定的主题发送消息，订阅者从指定的主题订阅接收消息，而 Broker 则起到按照主题接受并分发消息的代理人。在车联网 TSP 平台场景中，**车载设备、移动终端与业务应用都可以被看作是 MQTT 的客户端**。

根据业务不同与数据方向不同，车载设备、移动终端与业务应用的角色也会在发布者与订阅者之间切换。

主题的定义与规范

MQTT 协议中规定了主题是一段 UTF-8 编码的字符串，主题需要满足以下规则：

- 所有的主题名和主题过滤器必须至少包含一个字符。
- 主题名和主题过滤器是大小写敏感的。如：ACCOUNTS 和 Accounts 是不同的主题名。
- 主题名和主题过滤器可以包含空格字符。如：Accounts payable 是合法的主题名。
- 主题名或主题过滤器以前置或后置斜杠 / 区分。如：/finance 和 finance 是不同的。

- 只包含斜杠 / 的主题名或主题过滤器是合法的。
- 主题名和主题过滤器不能包含 null 字符(Unicode U+0000)。
- 主题名和主题过滤器是 UTF-8 编码字符串，除了不能超过 UTF-8 编码字符串的长度限制之外，主题名或主题过滤器的层级数量没有其它限制。

主题层级

MQTT 协议主题可以通过斜杠（"/" U+002F）将主题分割成多个层级；作为消息通道，客户端可以通过定义主题层级来实现对消息类型的细分；

例如：一个主机厂有多个车型，每个车型下面有多个车联网业务，我们在定义车机向对某个车型业务系统发消息时可以向<车型 A>/<车辆唯一标识>/<业务 X>主题发消息；

当然在 MQTT 世界中主题可以有很多层（MQTT 协议中没有限制层级数量），比如：<车型 A>/<车辆唯一标识(车架号)>/<业务 X>/<子业务 1>

这样，我们在定义车联网分层级的业务通道的时候可以按主题层级来设计。

通配符

MQTT 协议中订阅者的订阅的主题过滤器可以包含特殊的通配符，允许客户端一次订阅多个主题。

- 多层通配符

#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。如：订阅者可以通过订阅<车型 A>/# 接收到：

<车型 A>

<车型 A>/<车架号 1>

<车型 A>/<车架号 1>/<业务 X>

这几类主题的消息。

- 单层通配符

加号 ("+" U+002B) 用于单个主题层级匹配的通配符。如：订阅者可以通过订阅<车型 A>/+ 来接收

<车型 A>/<车架号 1>

<车型 A>/<车架号 2>

不同于多层通配符，使用单层通配符的时候无法匹配子层级的主题，比如：<车型 A>/<车架号 1>/<业务 X>的主题消息就无法接收到。

车联网 TSP 平台主题设计原则最佳实践

前文中我们提到在车联网场景中 MQTT 主题定义了业务与数据的通道，**主题定义的核心是区分业务场景。**

如何合理的定义主题，需要根据一定原则来设计。我们可以从以下几个维度来设计与定义主题：

根据业务数据方向区分

首先，数据的上下行方向不同决定了数据由谁产生，被谁消费。

在车联网场景中，车载设备到平台的数据上行通道与平台应用到车的下行数据需要通过主题分开。通过对上行、下行主题的设计区分，可以帮助设计、运维及业务人员快速定位场景、问题及相关干系方。

有些业务可能会同时用到上下行主题，比如车辆申请数据下发后平台下发数据，以及平台请求

车辆上班数据后车辆上报数据。这种情况下，由于 MQTT 协议的异步通讯机制，也需要对一个整体业务的上下行主题分别定义。

根据车型区分

在车联网场景中，不同车型意味着车辆产生的数据不完全相同，车机能力不完全相同，对接的业务应用也不尽相同。我们可以根据车型型号对差异化的车辆数据以及业务进行主题上的区分。

当然，同一个主机厂下的不同车型也会有相同的业务和数据，这些业务可以通过跨车型的主题来定义。

根据车辆区分

在车联网场景中，如车控等安全等级较高的业务场景往往需要一对一的主题作为数据通道。一方面通过主题来隔离车辆与车辆之间的业务信息，另一方面保证数据可以点对点的交互。

在主题设计中，有时需要将车辆的唯一标识符作为主题的一部分来实现一对一的消息通道。常见的方案有使用车辆 VIN 码作为主题的一部分。

根据用户区分

在实际使用场景中，也存在需要根据用户（而非车辆）实现车云的一对一的消息通道，此类需求经常发生在用户促销、运营、ToB 业务等场景中。

在主题设计时，常见的方案有两种，一是使用用户 ID 作为主题的一部分；二是通过人-车关系转换成车辆级主题，但由于消息时效性、车内用户登录状态等原因，此方案下生产端及消费端均需要添加额外的设计及处理，相对复杂。



根据研发环境区分

从项目工程实施角度出发，一般在主题设计时同时会添加环境变量，通过配置实现不同研发环境下的资源复用以及正确性检查。

根据数据吞吐量区分

由于业务的不同，不管是上行数据还是下行数据，数据的发送频率与报文大小都不尽相同。不同的数据吞吐量会影响到消费端的处理以及架构设计，比如我们在处理高频的车辆数据上报业务时往往要考虑应用层的消费能力，这时候可能要借助类似 Kafka 之类的高吞吐消息队列来进行数据缓冲，**防止应用消费不及时造成数据积压与数据丢失**。

所以在 MQTT 主题定义上，我们往往也需要对不同数据吞吐量的业务进行区分。

MQTT 协议主题设计在车联网场景中的应用

车辆数据主动上报

车载设备（T-box，车机等）作为车辆运行数据的收集者，基于固定频率将车内各类控制器、传感器等数据打包发送到平台端。此类数据一般可以按照上报数据的车型、车架号、业务数据类型等多个层级进行设计。

例如在用户同意的前提下，车辆在行驶过程中会将位置、车速、电量等信息按照固定频率上报云平台，云端应用基于这些数据，提供位置查找、超速提醒、电量提醒、地理围栏服务给终端用户使用。

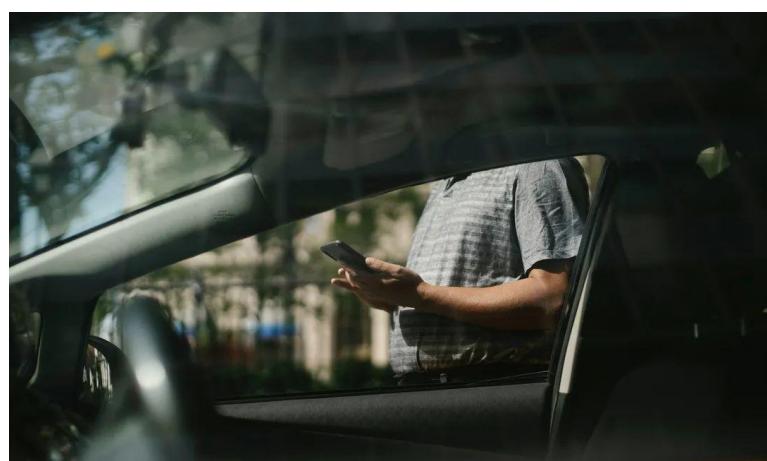
平台请求下发后车辆数据上报

当云平台需要获取车辆的最新状态及信息时，可以主动下发命令要求车辆上报数据。此类场景一般可以按照车架号、业务类型等层级进行主题设计。

例如在诊断场景下，平台通过 MQTT 下发诊断命令至车辆，当车内各设备完成诊断操作后，会将诊断数据打包后上报至云平台，车辆诊断工程师将根据采集到的诊断数据对于车况进行整体的分析及问题定位。

平台指令下发

车辆远程控制是车联网业务中最常见、最典型的场景，各主机厂均在手机 App 中提供各种遥控功能，例如远程启动、远程开车门、远程闪灯鸣笛等等。



此类场景下，手机 App 发送控制命令至云平台，平台应用经过权限检查、安全检查等一系列操作后，通过 MQTT 将命令下发至车辆执行，车辆端执行成功后，异步通知平台执行结果。

此类场景一般可以按照上行下行、车架号、业务类型、操作类型等多个层级进行主题设计。

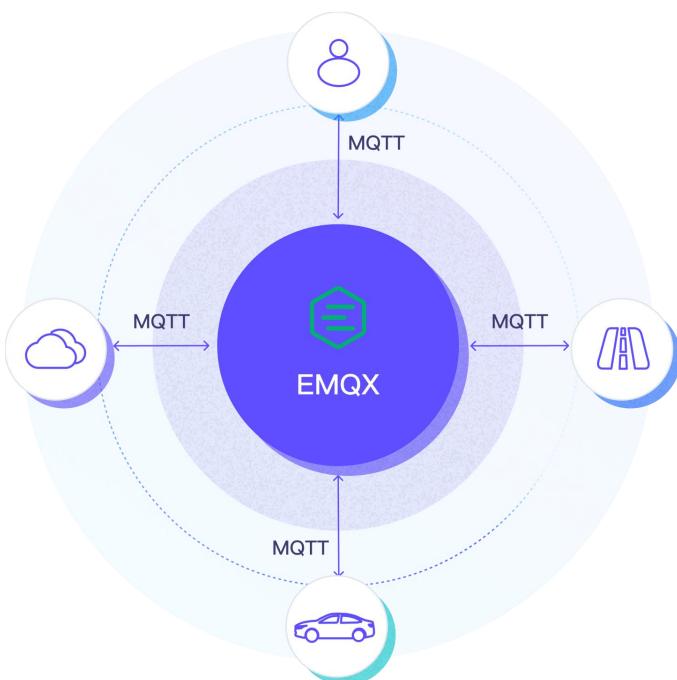
车辆客户端请求后平台数据下发

在 SDV（软件定义汽车）的大背景下，车内很多配置是可以做到动态变化的，例如数据采集规则、安全访问规则，所以车辆在点火启动后，会主动请求平台最新的相关配置，若两侧配置不一致，平台侧会下发最新的配置信息至车辆，车辆侧实时生效。

此类场景一般可以按照上行下行、车架号、业务类型等多个层级进行主题设计。

以 EMQX 进行车联网 TSP 平台主题设计

EMQX 作为全球领先的 MQTT 物联网消息中间件，基于分布式集群、大规模并发连接、快速低延时的消息路由等突出特性，能够有效处理车联网场景中高时效性业务需求，大幅度缩短端到端时延，为大规模车联网平台快速部署提供标准的 MQTT 服务。



EMQX 在车联网场景下的优势

- 海量主题支持

随着车联网场景中的业务不断增加，承载业务通道的主题数量也不断增加，尤其是包括车控场景所需要的一车一主题、一车多主题需求越来越大。在这种背景下，MQTT Broker 的主题数承载能力就成为了 TSP 平台的重要评估指标。

EMQX 在一开始的底层设计中就规划了对海量设备连接与海量主题支持的能力。常见的 16 核 32G 内存的 3 节点 EMQX 集群可以支持百万级主题同时运行，为 TSP 平台主题设计提供了灵活的设计空间。

- 强大规则引擎

EMQX 提供了内置的规则引擎，基于规则引擎可以提供对不同主题数据的查找、过滤、数据分拆以及对消息重新路由。使用规则引擎，我们可以在已有车载设备与应用主题建立好的场景下，通过创建新的路由规则与数据预处理规则对已有主题中的数据进行再处理。在车辆上市后，通过在平台侧定义新规则实现对新业务应用的支持。

在 EMQX 企业版中，规则引擎提供了数据持久化对接能力，可以通过规则引擎中的配置将不同主题中的数据直接对接不同持久化方案。比如对数据吞吐量比较高的数据可以通过规则引擎对接 Kafka、Apache Pulsar 等高吞吐消息队列进行数据缓冲；而车辆报警等小吞吐低时延主题数据可以直接对接应用，实现数据的快速路由消费。

- 代理订阅功能

EMQX 提供了代理订阅功能，客户端在连接建立时，不需要发送额外的 SUBSCRIBE 报文，便能自动建立用户预设的订阅关系。这样可以让平台侧直接管理车载设备的主题订阅关系，方便平台侧进行统一管理。

- 丰富的主题监控与慢订阅统计

EMQX 企业版提供了以主题为监控维度的运行数据监控，可以在 EMQX 可视化 Dashboard 中清晰看到主题下消息流入流出、丢弃的总数和当前速率。

自 4.4 版本起，EMQX 提供了对慢订阅的统计。该功能会追踪 QoS1 和 QoS2 消息到达 EMQX 后，完成消息传输全流程的时间消耗，然后采用指数移动平均算法，计算该订阅者的平均消息传输时延，之后按照时延高低对订阅者进行统计排名。

通过在 TSP 平台运营过程中不断监控各种主题的数据接收与消费情况，平台运营者就可以根据业务变化不断调整平台业务设计与应用设计，实现平台的不断优化扩展。

需要注意的事项

我们在使用 EMQX 作为车联网 TSP 平台 MQTT Broker 时，在设计主题的过程中需要注意以下几个问题：

- 通配符使用与主题数层级

由于 EMQX 采用主题树的数据结构对主题进行过滤匹配。在使用通配符来匹配多个主题的场景下，如果主题层级非常多，就会对 EMQX 产生比较大的资源消耗。所以在主题设计时，不建议层级太多，一般不建议超过 5 层。

- 主题与内存的消耗

由于在 EMQX 中主题数与主题长度主要与内存相关，我们在承载大量主题的同时也要重点监控 EMQX 集群内存的用量。

小结

随着 MQTT 协议在车联网业务中的广泛普及，车联网 TSP 平台的 MQTT 消息主题设计将是各主机厂与 TSP 平台方案供应商必须面对的课题。

我们结合多年 TSP 平台建设经验，针对车联网业务从多维度总结的 MQTT 主题设计思路，希望能够在平台前期设计与业务扩展阶段给行业同仁一些帮助与启发。

04 | QoS 设计：车联网平台消息传输质量保障

车联网场景下会产生海量数据，这些数据可以作为车辆诊断的基础，保障车辆安全稳定地运行；也可以与手机等基础设施进行联动，以提供更好的行车体验。

国家与行业也陆续出台了相关政策文件，如《汽车驾驶自动化分级》、《国家车联网产业标准体系建设指南》、《车联网信息服务数据安全技术要求》等，对车联网数据传输提出了更高要求。

通信的安全、稳定、可靠自始至终都是车联网亘古不变的话题，因此一套完善的数据传输保障方案也是车联网业务中不可忽视的一部分。

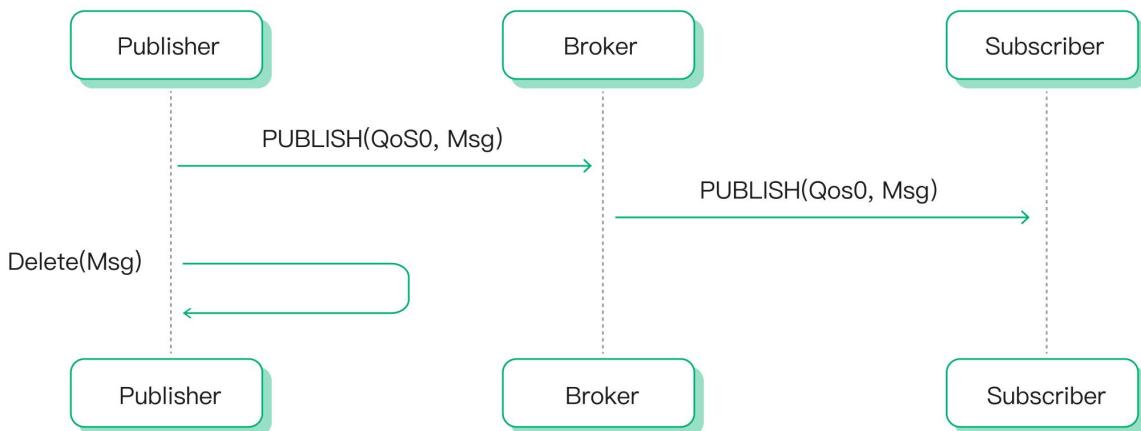
MQTT 协议中的 QoS 等级

作为现如今车联网行业数据通信协议的首选，MQTT 协议中规定了**消息服务质量**（Quality of Service，以下简称 QoS）。QoS 保证了在不同的网络环境下消息传递的可靠性，可作为车联网场景中保障消息可靠性传输的首要实现技术。

MQTT 设计了 3 个 QoS 等级：

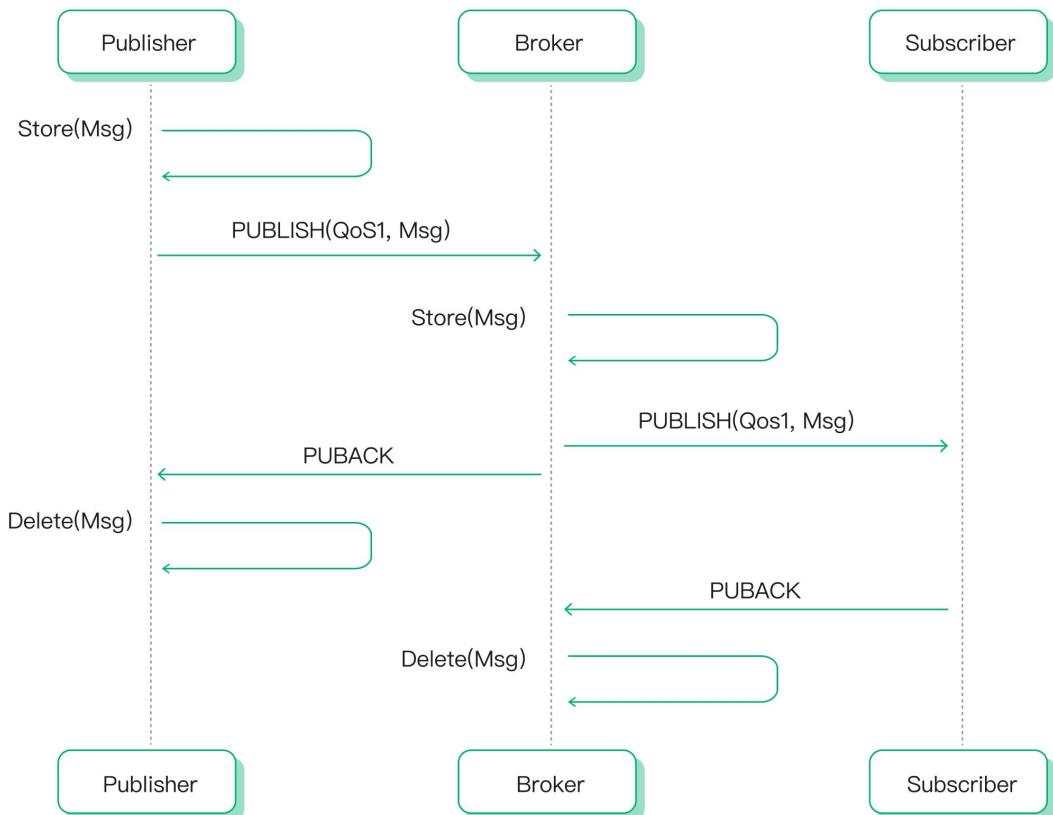
- QoS 0

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。Sender (可能是 Publisher 或者 Broker) 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。



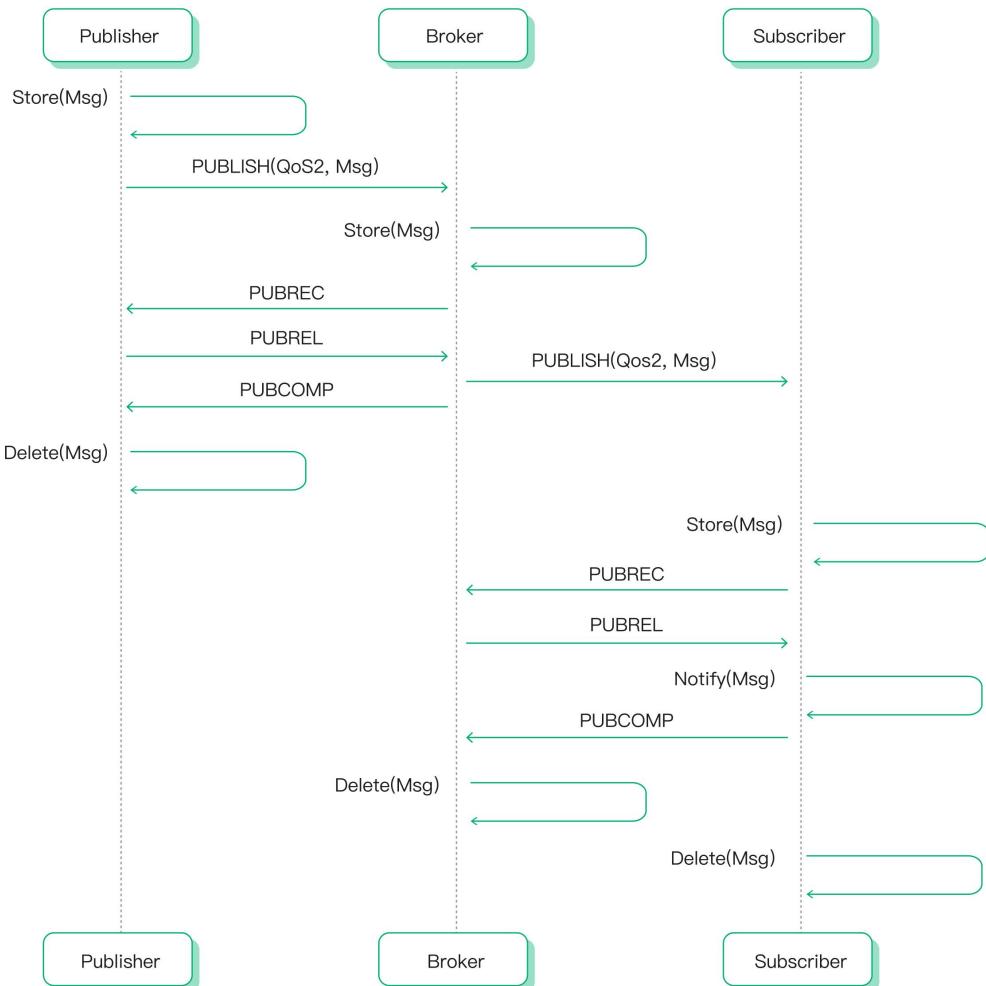
● QoS 1

消息传递至少 1 次。包含了简单的重发机制，Sender 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。



车联网场景中的消息 QoS 设计

首先需要明确的是 **QoS 级别越高，消息交互越复杂，系统资源消耗越大**，所以 QoS 等级不是设置的越高越好。应用程序可以根据自己的网络场景和业务需求，选择合适的 QoS 级别。

根据车联网信息服务相关数据的属性和特征，我们可以将其分为六类：基础属性类数据、车辆工控类数据、环境感知类数据、车控类数据、应用服务类数据和用户个人信息。

那么在不同的车联网场景中**如何选择 MQTT QoS 等级呢？**

- 以下情况下可以选择 QoS 0

可以接受消息偶尔丢失的场景下可以选择 QoS 0。

车联网提供的与娱乐相关的多媒体服务，如天气预报等数据等。还有部分涉车服务类数据，如车辆历史行车数据的上报、历史行车操作数据等。

- 以下情况下可以选择 QoS 1

车联网大部分场景都是选用 QoS 1，它实现了系统资源性能和消息实时性、可靠性最优化。

QoS 1 广泛运用于控车消息、行车上报数据（含新能源国标和企标）、交通安全管控类数据，和交通安全、道路安全相关的预警数据。

- 以下情况下可以选择 QoS 2

对于不能忍受消息丢失，且不希望收到重复的消息，数据完整性与及时性要求较高的场景，可以选择 QoS 2。

车联网场景中 QoS 2 的应用并不多，虽然其可以增加消息可靠性，但同时也使资源消耗和消息时延大幅增加。

QoS 2 主要运用于对数据完整性与及时性要求较高的银行、消防、航空等行业，有些主机厂的行车告警和车辆充电桩计费单消息会选择采用 QoS 2。

特别提醒

需要注意的是 MQTT 发布与订阅操作中的 QoS 代表了不同的含义，发布时的 QoS 表示消息发送到 MQTT Broker 使用的 QoS 等级，订阅时的 QoS 表示 MQTT Broker 向自己转发消息时可以使用的最大 QoS 等级。

需要保障发送与订阅的 QoS 一致，才能确保最终收到的消息是固定的 QoS 等级，否则会出现消费降级的情况。例如：A 发送的消息 QoS 为 2，B 订阅的消息 QoS 为 1，则最终接收到消息的 QoS 为 1。

EMQX 基于 QoS 等级的消息传输保障

为了更好地保障车联网过程中人-车-路-网-云之间数据传递的安全可靠，同时提高消息吞吐效率，减少网络波动带来的影响，云原生分布式物联网消息服务器 EMQX 在全面适配 QoS 信令交互的基础上，还设计了飞行窗口、消息队列、消息全链路追踪和离线消息存储等功能来提高消息可靠性。

飞行窗口的设计可允许多个未确认的 QoS 1 和 QoS 2 报文同时存在于网路链路上，消息队列则可以满足在消息链路中消息超出飞行窗口的同时对消息进行进一步存储，以满足客户端离线时未接收的消息或者未确认数据消息的存储需求。

飞行窗口同时也有 `upgrade_qos` 参数实现根据订阅强制升级 QoS 之类的功能，可实现 QoS 等级的一致性，确保不会出现消费降级的情况。

此外，EMQX 还可提供限制业务按区域接入实现不同的 QoS 等级、数据桥接 QoS 管理、MQTT-SN 协议 QoS 管理等能力，均为车联网场景下的消息可靠传输提供了有力保障。

下载体验：<https://www.emqx.com/zh/products/emqx>

小结

通过这一章节我们可以看到，MQTT 协议的 QoS 特性对于车联网场景下消息数据的安全传输具有重要意义。

作为完整支持 MQTT 协议标准的云原生分布式消息服务器，EMQX 在产品设计中也充分利用了 MQTT 协议的特性优势，为物联网平台与应用构建提供可靠的数据连接、移动、处理与集成。

05 | 车联网平台百万级消息吞吐架构设计

在之前的文章中，我们提到车联网 TSP 平台拥有很多不同业务的主题，并介绍了如何根据不同业务场景进行主题设计。

车辆会持续不断产生海量的消息，每一条通过车联网上报的数据都是非常珍贵的，其背后蕴藏着巨大的业务价值。因此我们构建的车辆 TSP 平台也通常需要拥有千万级主题和百万级消息吞吐能力。

传统的互联网系统很难支撑百万量级的消息吞吐。在本章节，我们将主要介绍如何针对百万级消息吞吐这一需求进行新一代车联网平台架构设计。

车联网场景消息吞吐设计的关联因素

车联网的消息分为上行和下行。

上行消息一般是传感器及车辆发出的告警等消息，把设备的信息发送给云端的消息平台。下行消息一般有远程控制指令集消息和消息推送，是由云端平台给车辆发送相应的指令。

在车联网消息吞吐设计中，我们需要重点考虑以下因素：

消息频率

车在行驶过程中，GPS、车载传感器等一直不停地在收集消息，为了收到实时的反馈信息，其上报接收的消息也是非常频繁的。上报频率一般在 100ms-30s 不等，所以当车辆数量达到百万量级时，平台就需要支持**每秒百万级的消息吞吐**。

消息包大小

整个消息包大小一般在 500B 到几十 KB 不等。当大量消息包同时上报时，需要车联网平台拥有更强的接收、发送大消息包的能力。

消息延时

车辆在行驶过程中，消息数据只能通过无线网络来进行传输。在大部分车联网场景下，**对车辆的时延要求是 ms 级别**。平台在满足百万级吞吐条件下，还需要保持低延时的消息传输。

Topic 数量和层级

在考虑百万级消息吞吐场景时，还需要针对消息 Topic 数量和 Topic 树层级进行规范设计。

Payload 编解码

当消息包比较大的时候，需要重点考虑**消息体的封装**。单纯的 JSON 封装在消息解析时不够高效，可以考虑采用 Avro、Protobuf 等编码格式进行 Payload 格式化封装。

对于百万级消息吞吐场景，基于 MQTT 客户端共享订阅消息或通过规则引擎实时写入关系型数据库的传统架构显然无法满足。

目前主流的架构选型有两种：一种是消息接入产品/服务+消息队列（Kafka、Pulsar、RabbitMQ、RocketMQ 等），另外一种是消息接入产品/服务+时序数据库（InfluxDB、Lindorm 等）来实现。

接下来我们将基于上述的关联因素和客户案例的最佳实践，以云原生分布式物联网消息服务器 EMQX 作为消息接入层，分别介绍这两种架构的实现方式。

EMQX+Kafka 构建百万级吞吐车联网平台

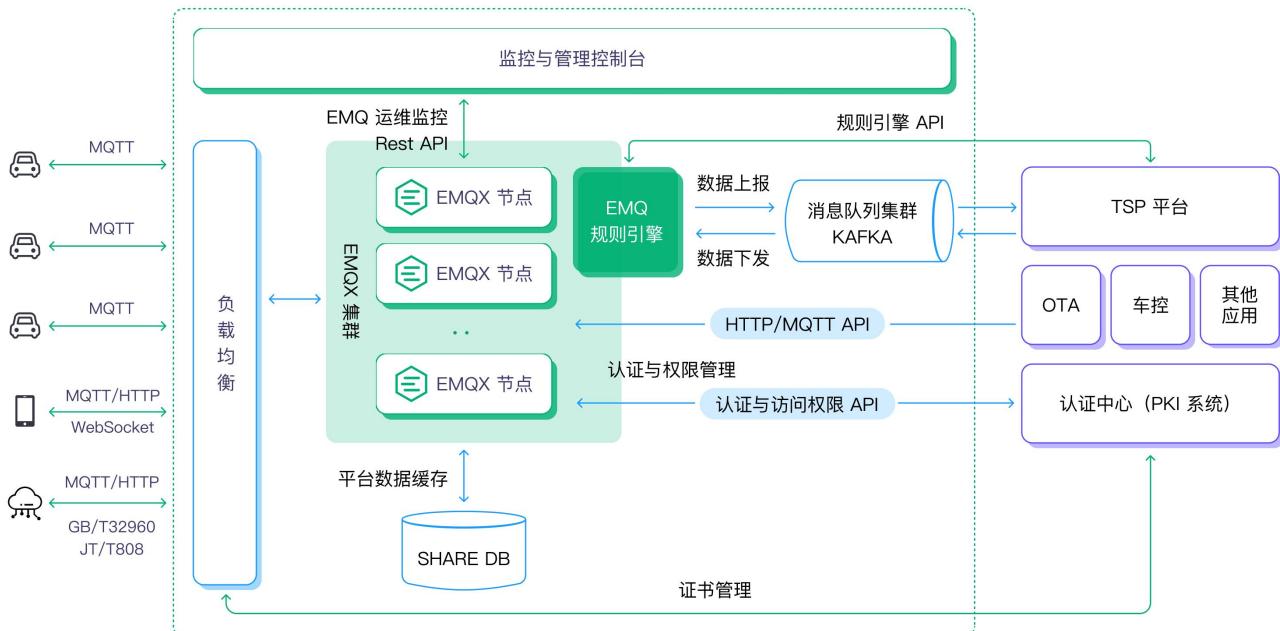
架构设计

Kafka 作为主流消息队列之一，具有持久化数据存储能力，可进行持久化操作，同时可通过将数据持久化到硬盘以及 replication 防止数据丢失。后端 TSP 平台或者大数据平台可以批量订阅想要的消息。

由于 Kafka 拥有订阅发布的能力，既可以从南向接收，把上报消息缓存起来；又可以通过北向的连接，把需要发送的指令通过接口传输给前端，用作指令下发。

我们以 Kafka 为例，构建 EMQX+Kafka 百万级吞吐车联网平台：

- 前端车机的连接与消息可通过公有云商提供的负载均衡产品用作域名转发，如果采用了 TLS/DTLS 的安全认证，可在云上建立四台 HAProxy/Nginx 服务器作为证书卸载和负载均衡使用。
- 采用 10 台 EMQX 组成一个大集群，把一百万的消息吞吐平均分到每个节点十万消息吞吐，同时满足高可用场景需求。
- 如有离线离线/消息缓存需求，可选用 Redis 作为存储数据库。
- Kafka 作为总体消息队列，EMQX 把全量消息通过规则引擎，转发给后端 Kafka 集群中。
- 后端 TSP 平台/OTA 等应用通过订阅 Kafka 的主题接收相应的消息，业务平台的控制指令和推送消息可通过 Kafka/API 的方式下发到 EMQX。



总体架构图

在这一方案架构中，EMQX 作为消息中间件具有如下优势，可满足该场景下的需求：

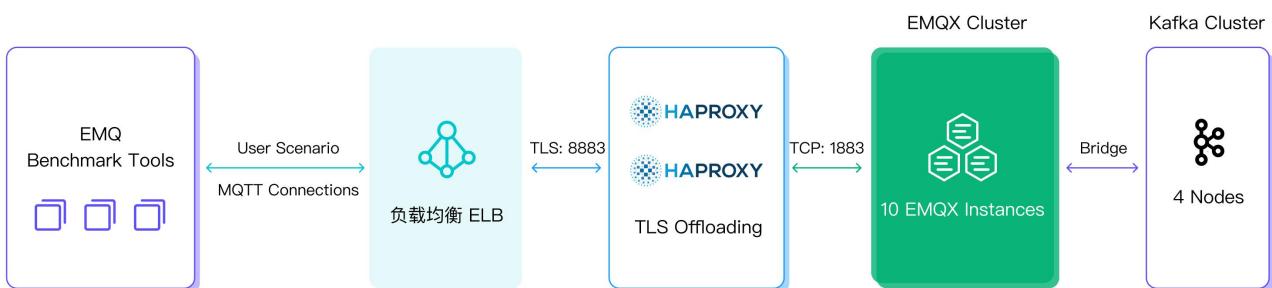
- 支持千万级车辆连接、百万级消息吞吐能力。
- 分布式集群架构，稳定可靠，支持动态水平扩展。
- 强大的规则引擎和数据桥接、持久化能力，支持百万级消息吞吐处理。
- 拥有丰富 API 与认证等系统能顺利对接。

百万吞吐场景验证

为了验证上述架构的吞吐能力，在条件允许的情况下，我们可以通过以下配置搭建百万级消息吞吐测试场景。压测工具可以选用 Benchmark Tools、JMeter 或 XMeter 测试平台。共模拟 100 万设备，每个设备分别都有自己的主题，每个设备每秒发送一次消息，持续压测 12 小时。

服务	数量	版本	操作系统	CPU	内存	云主机型号	内网网卡数量	开放端口
云商 LB	1							
HA (可选)	4	2.4.3	Centos 7.6	16核	32G	c6.4xlarge.2 存储: 普通	8	18083/1883/8883/8081
EMQX	10	企业版 v4.3.3	Centos 7.6	64核	128G	c6.16xlarge.2 存储: 普通	1	1883/8883
Kafka 云服务	4	2.3.0	Centos 7.6	16核	32G	c6.4xlarge.2 存储: 超高I/O	1	18083/1883/8883/8081

压测架构图如下：



性能测试部分结果呈现：



运行统计

命中次数		当前速度		最近5分钟执行速度	
3073438155 次		1003562.3 次/秒		1000057.62 次/秒	
预测启用后的执行次数					
节点	命中次数	当前速度	最大执行速度	最近5分钟执行速度	
emqx@10.167.93.0	307124950	100301.9	100392.8	99903.07	
emqx@10.167.94.114	307251308	100376.8	100527.9	99954.85	
emqx@10.167.95.132	307931443	100605.5	100797.5	100214.2	
emqx@10.167.93.18	307014777	100190.9	100797.6	99899.92	
emqx@10.167.94.174	307141040	100307.8	100457.5	99946.05	
emqx@10.167.95.166	307501254	100462.9	100630.2	100040.9	
emqx@10.167.94.13	307137673	100272.9	100816	99922.25	
emqx@10.167.94.36	307287851	100363.3	100839.7	100001.42	
emqx@10.167.94.39	307626639	100352.9	100913.6	100097.87	
emqx@10.167.92.66	307421220	100327.4	100905.7	100077.09	

EMQX 规则引擎统计

EMQX 规则引擎中可以看到每个节点速度为 10 万/秒的处理速度，10 个节点总共 100 万/秒的速度进行。

Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %	Producer Message/Sec	Summed Recent Offsets
_consumer_offsets	50	4	100	0	0	3	0	0.00	2
_trace	12	4	100	50	50	1	0	0.00	0
app_kafka	12	4	100	0	0	3	100	983249.32	4,093,378,274

Showing 1 to 3 of 3 entries

Previous Next

Kafka 管理界面统计

在 Kafka 中可以看到每秒 100 万的写入速度，并且一直持续存储。

EMQX+InfluxDB 构建百万级吞吐车联网平台

架构设计

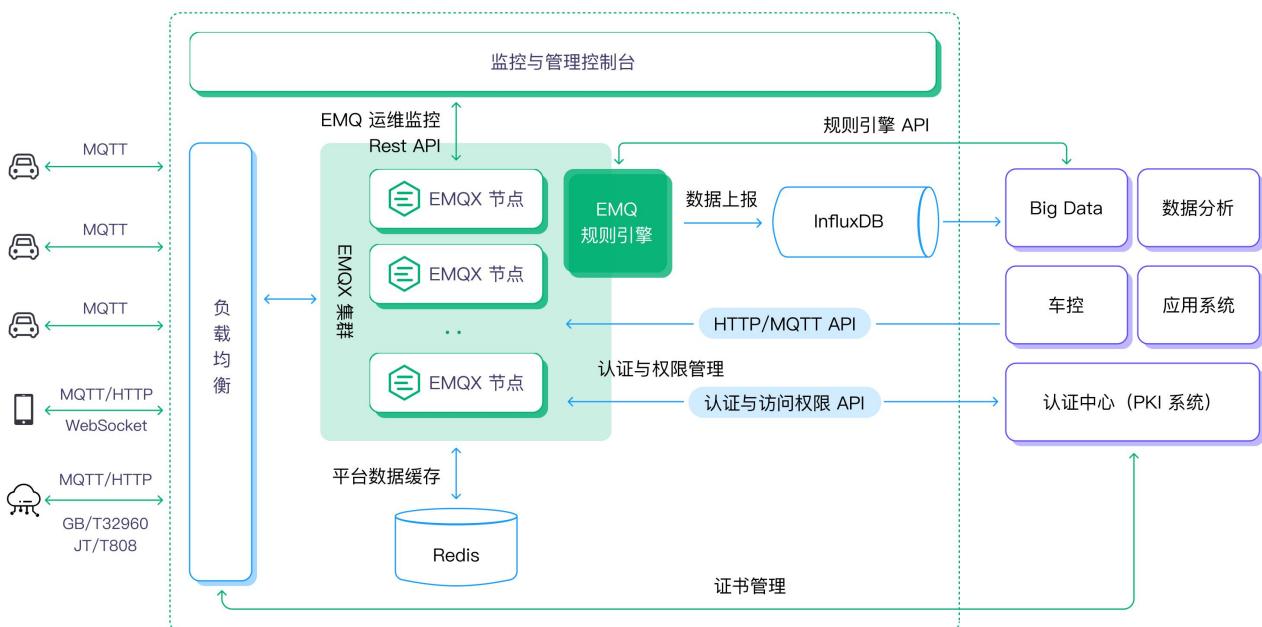
采用 EMQX+ 时序数据库的架构，同样可以构建百万级消息吞吐平台。在本文我们以

InfluxDB 时序数据库为例。

InfluxDB 是一个高性能的时序数据库，被广泛应用于存储系统的监控数据、IoT 行业的实时数据等场景。它从时间维度去记录消息，具备很强写入和存储性能，适用于大数据和数据分析。分析完的数据可以提供给后台应用系统进行数据支撑。

此架构中通过 EMQX 规则引擎进行消息转发，InfluxDB 进行消息存储，对接后端大数据和分析平台，可以更方便地服务于时序分析。

- 前端设备的消息通过云上云厂商的负载均衡产品用作域名转发和负载均衡。
- 本次采用 1 台 EMQX 作为测试，后续需要时可以采用多节点的方式，组成相应的集群方案（测试 100 万可以部署 10 台 EMQX 集群）。
- 如有离线离线/消息缓存需求，可选用 Redis 作为存储数据库。
- EMQX 把全量消息通过规则引擎转发给后端 InfluxDB 进行数据持久化存储。
- 后端大数据平台通过 InfluxDB 接收相应的消息，对其进行大数据分析，分析后再通过 API 的方式把想要的信息传输到 EMQX。



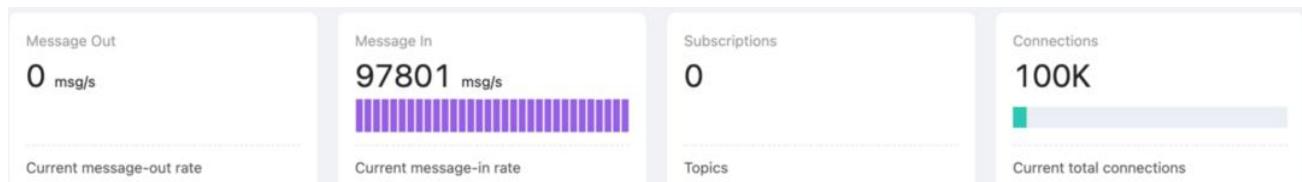
总体架构图

场景验证

如测试架构图中所示，XMeter 压力机模拟 10 万 MQTT 客户端向 EMQX 发起连接，新增连接速率为每秒 10000，客户端心跳间隔（Keep Alive）300 秒。所有连接成功后每个客户端每秒发送一条 QoS 为 1、Payload 为 200B 的消息，所有消息通过 HTTP InfluxDB 规则引擎桥过滤筛选并持久化发至 InfluxDB 数据库。

服务	数量	版本	操作系统	CPU	内存	云主机型号
EMQX	1	企业版 v4.3.5	Centos 7.8	32 核	64G	c6.8xlarge.2
InfluxDB	1	1.8.10	Centos 7.8	16 核	32G	c6.4xlarge.2
XMeter 管理机	2	3.2	Centos 7.8	8 核	16G	c6.2xlarge.2
XMeter 压力机	10	/	Centos 7.8	8核	16G	c6.2xlarge.2

测试结果呈现如下：



rule:850398

Metrics

matched: 344236893 matched
Current Speed: 0 matched/second
Maximum speed: 99501.2 matched/second
Last 5 minutes speed: 0 matched/second

Action metrics

data_to_influxdb
Success: 344236893
Fail: 0

EMQX 规则引擎统计

```
> use http
Using database http
> select count(*) from m1
name: m1
time count_val
---- -----
0    344236893
```

InfluxDB 数据库收到数据

Messages

received	344236893
dropped	344236893
retained	204
qos1.received	344236893

EMQX Dashboard 消息数统计

单台 EMQX 服务器实现了单台服务器 10 万 TPS 的消息吞吐持久化到 InfluxDB 能力。参考 EMQX+Kafka 架构的测试场景，将 EMQX 的集群节点扩展到 10 台，就可以支持 100 万的 TPS 消息吞吐能力。

小结

本章节介绍了车联网场景消息吞吐设计需要考虑的因素，同时提供了两种较为主流的百万级吞吐平台架构设计方案。面对车联网场景下日益增加的数据量，希望本文能够为相关团队和开发者在车联网平台设计与开发过程中提供参考。

06 | 车联网通信安全之 SSL/TLS 协议

在汽车出行愈加智能化的今天，我们可以实现手机远程操控车辆解锁、启动通风、查看车辆周围影像，也可以通过 OTA（空中下载技术）完成升级车机固件、更新地图包等操作，自动驾驶技术更是可以让车辆根据路面状况自动辅助实施转向、加速和制动。



然而，每项提升我们使用体验的功能，都有可能成为致命的安全漏洞。

腾讯安全科恩实验室曾向外界披露并演示过如何凭借 3/4G 网络或者 WiFi 网络，在远程无物理接触的情况下入侵智能汽车，实现对车辆信号灯、显示屏、门锁甚至是刹车的远程控制。不仅如此，攻击者甚至可以利用某个已知漏洞获取智能汽车的 Autopilot 控制权，对车辆行驶方向进行操控。

因此，我们在车联网平台构建时也应充分认识到**通信安全、身份认证、数据安全**的重要性，正确使用相关加密认证等技术手段来提供保障。

这一章节中我们将全面介绍 **SSL/TLS 协议在车联网通信安全中的应用**，希望能让大家对 SSL/TLS 的作用有更清晰直观的认识。此外，我们还将详细讲解 **SSL/TLS 的配置方式**，确保大家能正确使用 SSL/TLS，实现安全性保障。

车联网安全通信 MQTT 协议

MQTT 协议是在 MQTT 协议的基础上，封装了一层基于 SSL/TLS（传输层安全）的加密协议，它确保车机端和车联网平台通信是加密的。

但如果没有正确配置 SSL/TLS，依然会存在很多安全隐患。想要真正运用好 SSL/TLS，我们必须了解 SSL/TLS 解决了哪些问题，以及对 SSL/TLS 用到的密码技术有初步的认知。

通常情况下，通信过程需要具备以下四个特性，才能被认为是安全的，分别是：**机密性、完整性、身份认证和不可否认。**

机密性

机密性是安全通信的基础，缺少机密性任何窃听通信的人都可以轻而易举获取到你的诸如登录密码、支付密码等关键隐私信息。

实现机密性最常用的手段就是加密，这样窃听者只能得到加密后的毫无意义的一串数据，只有持有密钥的人才能将密文恢复成正确的原始信息。

根据密钥的使用方法，加密方式可以分为对称加密和非对称加密两种。对称加密是指加密和解密使用相同的密钥，非对称加密则是指加密和解密时使用不同的密钥。

对称加密由于通信双方要使用相同的密钥来进行加解密，所以必然会遇到密钥配送问题，即我需要对方能够解密我发送过去的密文，我就必须把我加密时使用的密钥告诉对方，但是我如何保证将密钥与对方同步的过程中密钥不会泄漏？这就是对称加密的密钥配送问题。

目前常用的解决方案是使用**非对称加密**和使用 **Diffie-Hellman 密钥交换算法**。

非对称加密的核心是生成一对密钥，一个是公钥，一个是私钥，公钥用于加密，它是公开的，可以派发给任何人使用，私钥用于解密，不参与通信过程，需要被妥善保管，这样就解决了密

钥配送问题。

Diffie-Hellman 密钥交换算法的核心思想则是通信双方交换一些公开的信息就能够计算出相同的共享密钥，而窃听者获得这些公开信息却无法计算出相同的密钥。

Diffie-Hellman 算法的一个好处是没有非对称加密的性能问题，非对称加密虽然解决了密钥配送问题，但非对称加密算法的运算速度远远不及对称加密算法，它们甚至能有几百倍的差距。

虽然保障了安全，但严重影响了通信的效率，丧失了实用性。

因此，实际应用时通常会将对称加密和非对称加密结合使用，即使用伪随机数生成器生成会话密钥后，用公钥进行加密并发送给对方，对方收到密文后使用私钥解密取出会话密钥，后续通信将完全使用该会话密钥。这样既解决了密钥配送问题，又解决了非对称加密带来的性能问题，这种方式通常又被称为**混合加密**。

完整性

仅仅具备机密性还不足以实现安全的通信，攻击者依旧可以篡改、伪造密文内容，而接收者既无法判断密文是否来自正确的发送者，也无法判断解密后的明文是否是未经篡改的。

尽管对加密之后的密文进行针对性篡改的难度有所上升，例如篡改之后明文的数据结构很有可能会遭到破坏，这种情况下接收者能够很轻易地拒绝这个明文。

但依然存在篡改之后正好使得解密得到的明文消息中，某些本身就具备随机属性的字段的值发生变化的概率，例如电机转速字段的值从 500 变为了 718，无非是几个比特位的变化，如果接收者正常接受这些消息，就可能带来意想不到的隐患。

因此，我们还需要在机密性的基础上进一步保证信息的完整性。

常见的做法就是**使用单向散列函数计算消息的散列值，然后将消息和散列值一起发送给接收者。**

单向散列函数能够确保消息中哪怕只有 1 比特的改变，也有很高的概率产生不同的散列值。这样接收者就可以计算消息的散列值，然后对比收到的散列值来判断数据是否被人篡改。

身份认证

但可惜的是，当攻击者同时伪造消息和对应的散列值时，接收者依然无法识破这个伪装。

因此，我们不仅需要确认消息的完整性，还需要确认消息是否来自合法的发送者，也就是说还需要对身份进行认证。

这个时候我们就需要用到**消息认证码**，消息认证码依然基于单向散列函数，但它的输入除了原本的消息以外，还包括了一个发送者与接收者之间共享的密钥。

由于消息认证码本身并不提供消息机密性的保证，所以在实际使用中，通常会将对称加密与消息认证码结合使用，以同时满足机密性、完整性和认证的要求，这种机制也被称作认证加密（AEAD）。在具体使用方面，产生了以下几种方案：

1. **Encrypt and MAC**: 先用对称密码将明文加密，再计算明文的 MAC 值，最后把二者拼接起来发给接收方。
2. **MAC then Encrypt**: 先计算明文的 MAC 值，然后将明文和 MAC 值同时用对称密码加密，加密后的密文发送给接收方。
3. **Encrypt then MAC**: 先用对称密码将明文加密，再后计算密文的 MAC 值，最后把二者拼接起来发给接收方。

在很长一段时间内，SSL/TLS 都采用了第二种方案，但事实上以上三种方案都已经陆续被验证为**存在安全漏洞**。SSL/TLS 历史上的 POODLE 和 Lucky 13 攻击都是针对 MAC then Encrypt 方案中的漏洞实现的。**目前业界推荐的安全方案是采用 AEAD 算法**，SSL/TLS 1.3 版本中也正式废除了其他加密方式，仅支持 AEAD 加密。

不可否认

现在，我们已经保证了消息的机密性，同时也能识别出伪装和篡改，但是由于消息认证码的核心是需要通信双方共享密钥，因此又引发了新的问题，即**无法对第三方证明以及无法防止否认**。

假设 Bob 接收了来自 Alice 的消息，想要向第三方证明这条消息的确是 Alice 发送的，就需要将原本只有两个人知道的密钥告诉给第三方，这显然会增加后续继续使用这个密钥通信的安全风险。同时，即便第三方拿到了密钥，也无法得出有效的结论，例如 Bob 可以宣称这条消息是由 Alice 构造的，因为 Alice 也持有相同的密钥。

因此，我们还需要引入**数字签名机制**，它的原理与非对称机密很像，又正好相反。数字签名需要发送者用私钥对消息施加签名，然后将消息与签名一并发送给接收者，接收者则使用对应的公钥验证签名，确认签名来自合法的发送者。

由于**只有持有私钥的人才能施加正确的签名**，这样发送者就无从否认了。而**公钥只是用来验证签名**，所以可以随意派发给任何人。

可能敏感的读者到这里心中已经有些疑问了，是的，取到公钥的人如何确认这个公钥的确来自自己期望的通信对象呢？如果攻击者伪装成发送者，并把自己的公钥给了接收者，那么就能在无需破解数字签名算法的前提下完成攻击。

我们已经陷入了一个死循环，数字签名是用来识别消息篡改、伪装以及否认的，但在此之前我们又必须从没有被伪装的发送者得到没有被篡改的公钥才行。

到了这一步，我们只能借助外力的帮助了，委托公认的可信第三方，也就是我们现在常说的认证机构或 CA，由它来给各个公钥施加签名，形成**公钥证书**。显而易见的是，认证机构需要努力确保自己的私钥不被窃取，以保证数字签名的有效性。

虽然认证机构的私钥依然有泄漏的概率，甚至认证机构本身也可能被攻击者伪装，我们依然无法获得绝对的安全，但提前信任几个已知的认证机构，总是比从全新的通信对象获取并信任他

的公钥要可靠的多。

以上这些密码技术，共同构成了现代安全通信领域的基石。

而 **SSL/TLS 作为目前世界上应用最广泛的密码通信方法**，综合运用了前面提到的对称加密、非对称加密、消息认证码、数字签名、伪随机数生成器等密码技术，来提供通信安全保障。

考虑到密码学技术是不断进步发展的，或者说目前看似可靠的加密算法，可能在第二天就会被宣告攻破，所以 SSL/TLS 并没有强制使用某一种密码技术，而是**提供了密码套件（Cipher Suite）这一机制**，当某项密码技术被发现存在弱点，可以随时像零件一样替换它，当然前提是客户端和服务端使用相同的密码技术。

这也延伸出了 SSL/TLS 的握手协议，协商使用的密码套件就是这一部分协议的主要工作之一。

想要 SSL/TLS 具备良好的安全性，就需要避免使用已经被攻破或者已经被验证为弱安全性的加密算法，要避免使用容易被预测的伪随机数生成器，要尽量保证各个算法具有近似的安全性（短板效应）。

因此，**如何正确选择密码套件，也成为了保障安全性的一个重要环节**。这里我也会对目前推荐的密码技术和加密算法进行一个简单的整理，希望可以帮助各位读者查漏补缺：

- 对称加密算法中 RC4、DES、3DES 都已经被认为是不安全的了，目前推荐使用的只有 AES 和 ChaCha20。ChaCha20 是 Google 设计的一种加密算法，如果 CPU 或软件不支持 AES 指令集，ChaCha20 可提供比 AES 更好的性能。
- AES 这类对称加密算法只能加密固定长度的明文，想要加密任意长度的明文，还需要用到分组模式。早期的 ECB、CBC、CFB、OFB 等分组模式已经被认定为存在安全漏洞，目前更推荐使用 GCM、CCM 和 Poly1305。
- 常用的非对称加密算法有 DH、RSA、ECC 这几种。由于 DH 和 RSA 都不具备前向安全

性，目前已经不推荐使用，TLS 1.3 中更是直接废除了 DH 和 RSA 算法，取而代之的是安全强度和性能都明显优于 RSA 的 ECC 算法，它有两个子算法，ECDHE 用于密钥交换，ECDSA 用于数字签名。但需要注意的是，由于 ECDHE/DHE 不提供身份验证，因此服务端应当启用对客户端证书的验证。

- 散列算法方面，我们熟知的 MD5 和 SHA-1 都已经被认定为不再可靠，不推荐继续使用。目前通常建议使用 SHA256 或更高版本。

在了解推荐使用的密码技术以后，也许我们想要修改客户端或服务端的密码套件配置，但此时我们可能会发现这些密码套件的名称还有点难以理解。

例如 `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`，其中 TLS 只是表示协议，ECDHE 表示密钥交换算法，ECDSA 表示身份认证算法，AES_256_CBC 表示批量加密算法，SHA384 表示消息认证码 MAC 算法。这通常是 TLS 1.2 中密码套件的命名格式，而到了 TLS 1.3 则又发生了一些变化。由于 TLS 1.3 只接受使用 ECDHE 算法进行密钥交换，并且使用 ECDSA 进行身份认证，因此它的密码套件名称可以精简成 `TLS_AES_256_GCM_SHA384` 这种格式。

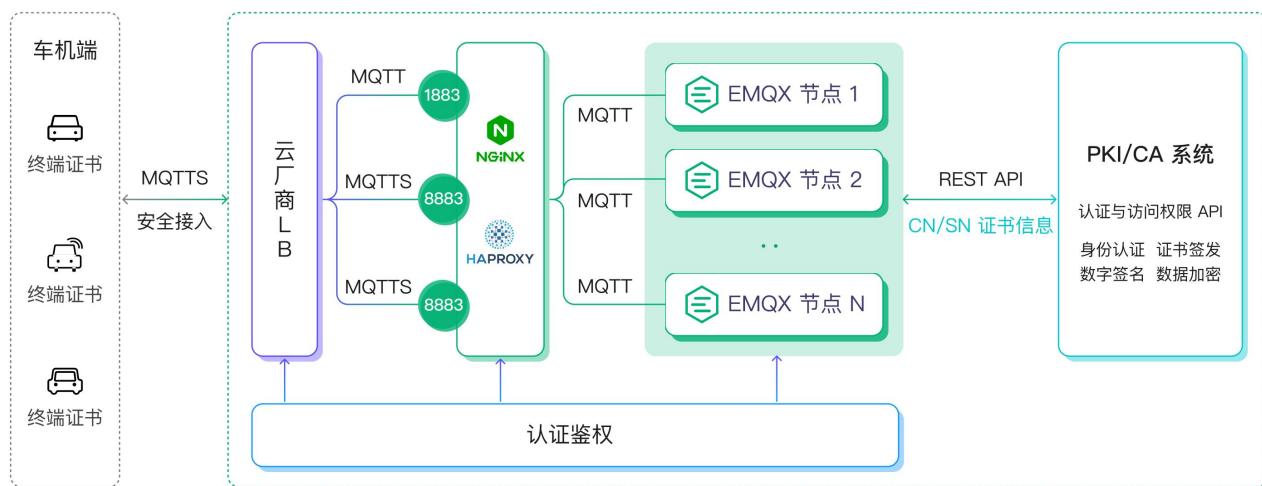
如果仅从安全性角度出发，个人建议使用

`TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384` 和 `TLS_AES_256_GCM_SHA384`。但考虑到目前仍有很多以 RSA 方式签发的证书正在使用，因此我们还需要根据自身情况来选择是否要继续使用 `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`。

构建安全认证体系典型架构

采用基于 PKI/CA 的数字证书体系是解决车联网安全关键的一步，也是大多数车企典型安全管理体系。其主要的设计思路如下：

1. 基于数字证书的身份标识：通过 PKI/CA 系统建立严谨的证书管理和使用规范，为车联网的应用和终端颁发数字证书，虚拟身份和真实身份进行绑定，解决身份标识和唯一性问题（可实现一机一密或一型一密）；
2. 所有数据交互时通过终端的身份唯一标识证明身份的真实性，防止第三方恶意入侵；
3. 基于数字证书安全功能，提供身份鉴别、身份认证、数据加解密、数字签名与验签等多种功能，满足车联网中 TSP、OTA 等多业务安全需求。



车联网平台安全通信交互流程，一般是将车机端申请终端证书，下载并完整安装后通过 MQTTS 安全协议与云端平台请求建立安全连接。

在云端我们可以选择在云厂商的负载均衡产品、基于 Nginx/HAProxy 自行搭建的 LB 层或是 MQTT Broker 层进行认证鉴权，同时通过 proxy_protocol v2 将车机端的 ID 信息、用户名密码及证书的 CN/SN 等信息通过调用 PKI/CA 统一认证接口进行唯一性认证，实现一机一密或一型一密的安全认证。

MQTTS 通信中单、双向认证的配置方式

SSL/TLS 连接认证认证的是对方身份，是否是可信的通信对象，认证的依据则是通信对象提供的证书。

通常情况下是由客户端对服务端的身份进行认证，也就是所谓的单向认证。那么双向认证顾名思义就是在单向认证的基础上，服务端对客户端的身份进行认证。

认证的原理其实非常简单，以单向认证为例，最简单的情况就是服务端在 SSL/TLS 握手阶段发送服务端证书，客户端验证该证书是否由受信任的 CA 机构签发，也就是使用受信任的 CA 证书中的公钥来验证服务端证书中的数字签名是否合法。

当然，大部分情况会比上述的稍微复杂一些，即服务端的证书不是由最顶层的 CA 机构直接签发的，而是由根 CA 机构对下层 CA 机构的公钥施加数字签名，形成中间 CA 证书，这样 的关系可能会多达几层，以尽可能保护根证书的安全。

大部分情况下常见 CA 机构的根 CA 证书和中间 CA 证书都已经内置在我们的操作系统中了，只有少数情况下需要自行添加信任的 CA 证书。

多级证书或者说证书链的认证过程会稍微复杂一些，但如果我们搞明白了前面说的证书签发逻辑，其实理解起来也很简单。

还是以单向认证为例，如果客户端只信任了根 CA 证书，那么服务端在握手阶段就需要发送服务端证书和根 CA 证书到服务端证书之间的所有中间 CA 证书。只有客户端拿到了完整的证书链，才能通过自己持有的根 CA 证书一层一层往下验证，缺少中间 CA 导致证书链不完整或者包含了错误的中间 CA，都会导致信任链中断而无法通过认证。

如果客户端除根 CA 证书以外，还持有一部分中间 CA 证书，那么在认证过程中，服务端还可以省略这些中间 CA 证书的发送，来提高握手效率。

因此，当我们配置单向认证时，需要在服务端指定服务端证书和中间 CA 证书（可选），以及服务端私钥文件。客户端则需要信任相应的根 CA 证书，信任的方式可以是在连接时指定或者通过证书管理工具将该根 CA 证书添加到信任列表。通常客户端库还提供了对端验证选项允许选择是否验证证书，关闭对端验证将在不验证证书的情况下直接创建加密的 TLS 连接。但这会带来中间人攻击的安全风险，因此**强烈建议启用对端验证**。

在启用对端验证后，客户端通常还会检查服务器证书中的域名（SAN 字段或 CN 字段）与自己连接的服务器域名是否匹配。如果域名不匹配，则客户端将拒绝对服务器进行身份验证或建立连接。

双向认证的配置方式只需要在单向认证的基础上，在服务端启用对端验证即表示启用双向认证以外，再参考服务端证书的配置方式正确配置客户端证书即可。

常见 TLS 选项介绍

当使用 EMQX 配置 SSL/TLS 连接时，通常会有 `certfile`、`keyfile` 等选项，为了帮助大家更好地了解这些选项的配置方式，接下来我们会对这些常见的 TLS 选项做一个简单的梳理和介绍：

- `certfile`，用于指定服务端或客户端证书和中间 CA 证书，需要指定多个证书时通常将它们简单地合并到一个证书文件中即可。
- `keyfile`，用于指定服务端或客户端私钥文件。
- `cacertfile`，用于指定 Root CA 证书，单向认证时客户端需要配置此选项以校验服务端证书，双向认证时服务端也需要配置此选项以校验客户端证书。
- `verify`，用于指定是否启用对端验证。客户端启用对端验证后通常还会去检查连接的服务器域名与服务器证书中的域名是否匹配。客户端与服务端同时启用则意味着这将是一个双向认证。
- `fail_if_no_peer_cert`，这是一个服务端的选项，通常在服务端启用对端验证时使用，设置为 `false` 表示允许客户端不发送证书或发送空的证书，相当于同时开启单向认证和双向认证，这会增加中间人攻击的风险。
- `versions`，指定支持的 TLS 版本。通信双方会在握手过程中，将 `versions` 选项中指定的版本发送给对方，然后切换至双方都支持的最高版本。同时也会基于该协议版本来协商密码套件。

- `ciphers`, 指定支持的密码套件。注意事项：避免使用前文提到的或其他被认定为弱安全性的密码套件，以及当使用包含 ECDSA 签名算法的密码套件时，需要额外注意自己的证书是否为 ECC 类型。
- `server_name_indication`, 服务器名称指示，这是一个客户端的选项。通常在客户端启用对端验证且连接的服务器域名与服务器证书中的域名不匹配时使用。例如服务器证书中的域名为 abc.com，而客户端连接的是 123.com，那么就需要客户端在连接时指定 `server_name_indication` 为 abc.com 表示自己信任该域名以通过证书检查。又或者将 `server_name_indication` 设置为 `disable` 来关闭此项检查，但这会增加中间人攻击的风险，通常并不建议这样做。

示例

为了便于演示，我们会使用 EMQX Broker（4.3.0 版本及以上

<https://github.com/emqx/emqx>）作为服务端，在 EMQX Broker 的控制台中使用 Erlang 的 `ssl:connect/3` 函数作为客户端。

- 单向认证

修改 EMQX Broker 配置如下：

```
# 监听端口我们使用默认的 8883
listener.ssl.external = 8883
# 服务端私钥文件
listener.ssl.external.keyfile = etc/certs/zhouzb.club/Nginx/2_zhouzb.club.key
# 证书捆绑包文件，包含了服务端证书和中间 CA 证书
listener.ssl.external.certfile = etc/certs/zhouzb.club/Nginx/1_zhouzb.club_bundle.crt
# 不开启对端验证
listener.ssl.external.verify = verify_none
```

```
# 支持 TLS 1.2 和 TLS 1.3
```

```
listener.ssl.tls_versions = tlsv1.3,tlsv1.2
```

```
# 服务端支持的密码套件
```

```
listener.ssl.external.ciphers = TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256,TLS_CHACHA20_POLY1305_SHA256,TLS_AES_128_CCM_SHA256,TLS_AES_128_CCM_8_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
```

启动 EMQX Broker 并进入控制台：

```
$ ./emqx console
```

使用 `ssl:connect/3` 函数连接：

```
%% 1. 指定用于验证服务端证书的 Root CA 证书
```

```
%% 2. 启用对端验证
```

```
%% 3. 仅支持 TLS 1.2
```

```
%% 4. 仅支持 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 这一个密码套件
```

```
ssl:connect("zhouzb.club", 8883,
            [{cacertfile, "etc/certs/zhouzb.club/DigiCertGlobalRootCA.crt.pem"},
             {verify, verify_peer},
             {versions, ['tlsv1.2']},
             {ciphers, ["TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"]}]).
```

- 双向认证

为了尽快进入正题，这里将继续使用服务端证书来充当客户端证书。**但是这存在严重的安全隐患，在生产环境中请勿这样使用！**

修改 EMQX Broker 配置如下：

```
# 监听端口我们使用默认的 8883
```

```
listener.ssl.external = 8883
```

```

# 服务端私钥文件
listener.ssl.external.keyfile = etc/certs/zhouzb.club/Nginx/2_zhouzb.club.key

# 证书捆绑包文件，包含了服务端证书和中间 CA 证书
listener.ssl.external.certfile = etc/certs/zhouzb.club/Nginx/1_zhouzb.club_bundle.crt

# 指定用于验证客户端证书的 Root CA 证书
listener.ssl.external.cacertfile = etc/certs/zhouzb.club/DigiCertGlobalRootCA.crt.pem

# 启用对端验证
listener.ssl.external.verify = verify_peer

# 要求客户端必须提供证书
listener.ssl.external.fail_if_no_peer_cert = true

# 支持 TLS 1.2 和 TLS 1.3
listener.ssl.tls_versions = tlsv1.3,tlsv1.2

# 服务端支持的密码套件
listener.ssl.external.ciphers = TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256,TLS_CHACHA20_POLY1305_SHA256,TLS_AES_128_CCM_SHA256,TLS_AES_128_CCM_8_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

```

启动 EMQX Broker 并进入控制台，然后使用 `ssl:connect/3` 函数连接：

```

%% 1. 指定用于验证服务端证书的 Root CA 证书
%% 2. 启用对端验证
%% 3. 仅支持 TLS 1.2
%% 4. 仅支持 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 这一个密码套件
ssl:connect("zhouzb.club", 8883,
            [{cacertfile, "etc/certs/zhouzb.club/DigiCertGlobalRootCA.crt.pem"},
             {certfile, "etc/certs/zhouzb.club/Nginx/1_zhouzb.club_bundle.crt"},
             {keyfile, "etc/certs/zhouzb.club/Nginx/2_zhouzb.club.key"},
             {verify, verify_peer},
             {versions, ['tlsv1.2']},
             {ciphers, ["TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"]}]).
```

小结

工业和信息化部印发的《车联网网络安全和数据安全标准系统建设指南》中明确指出，要构建**车联网网络安全和数据安全**的标准体系。

车联网领域的网络通信安全和数据安全将受到越来越多的关注，是车联网企业提高竞争力的关键影响因素之一。希望通过这一章节的内容，读者可以掌握 SSL/TLS 协议的使用方式，在实际业务场景中正确应用，实现车联网通信安全保障。

07 | 国密在车联网安全认证场景中的应用

国密即国家密码局认定的国产密码算法。通过自主可控的国产密码算法保护重要数据的安全，是有效提升信息安全保障水平的重要举措。

目前，我国在金融银行、教育、交通、通信、国防工业等各类重要领域的信息系统均已开始进行国产密码算法的升级改造。

随着汽车电动化、网联化、智能化融合发展，**车辆运行安全、数据安全和网络安全风险交织叠加，亟需加快建立健全车联网网络安全和数据安全保障体系，为车联网产业安全健康发展提供支撑。**

2022 年 2 月，工业和信息化部在现有国家车联网产业标准体系的基础上，组织编制了《车联网网络安全和数据安全标准体系建设指南》，其中已发布的 GB/T 37376-2019《交通运输数字证书格式》等国标文件中，凡涉及密码算法相关的内容，均考虑了国密的应用与实现。

本章节将详细介绍国密算法的分类及应用，以及如何使用 EMQX 实现国密证书集成，保障车联网信息安全。

国密的分类

为了保障在金融、医疗等领域保障信息传输安全，国家商用密码管理办公室制定了一系列密码标准，包括 SM1（SCB2）、SM2、SM3、SM4、SM7、SM9、祖冲之密码算法（ZUC）等。其中 SM1、SM4、SM7 是对称算法，SM2、SM9 是非对称算法，SM3 是哈希算法。

SM1 算法

SM1 算法是分组对称算法，分组长度为 128 位，密钥长度都为 128 比特，算法安全保密强度及相关软硬件实现性能与 AES 相当，算法不公开，仅以 IP 核的形式存在于芯片中。

采用该算法已经研制了系列芯片、智能 IC 卡、智能密码钥匙、加密卡、加密机等安全产品，广泛应用于电子政务、电子商务及国民经济的各个应用领域（包括国家政务通、警务通等重要领域）。

SM2 算法

SM2 算法是一种先进安全的公钥密码算法，在我们国家商用密码体系中被用来替换 RSA 算法。

SM2 算法就是 ECC 椭圆曲线密码机制，但在签名、密钥交换方面不同于 ECDSA、ECDH 等国际标准，而是采取了更为安全的机制。另外，SM2 推荐了一条 256 位的曲线作为标准曲线。

SM3 算法

SM3 是一种哈希算法，其算法本质是给数据加一个固定长度的指纹，这个固定长度就是 256 比特。用于密码应用中的数字签名和验证、消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。

SM4 算法

SM4 算法是一个分组算法，用于无线局域网产品。

该算法的分组长度为 128 比特，密钥长度为 128 比特。加密算法与密钥扩展算法都采用 32 轮非线性迭代构。解密算法与加密算法的结构相同，只是轮密钥的使用顺序相反，解密轮密钥是加密轮密钥的逆序。

SM7 算法

SM7 算法是一种分组密码算法，分组长度为 128 比特，密钥长度为 128 比特。SM7 的算法

文本目前没有公开发布。

SM9 算法

SM9 是基于对的标识密码算法，与 SM2 类似，包含四个部分：总则，数字签名算法，密钥交换协议以及密钥封装机制和公钥加密算法。

目前支持国密算法的软硬件密码产品包括 SSL 网关、数字证书认证系统、密钥管理系统、金融数据加密机、签名验签服务器、智能密码钥匙、智能 IC 卡、PCI 密码卡等多种类型。

但常用的操作系统、浏览器、网络设备、负载均衡设备等软硬件产品，仍然不支持国产密码算法。受到国密算法兼容性的制约，在 HTTPS 加密应用方面，国密算法的应用仍然比较滞后。

密码（GmSSL）证书与传统 SSL 证书对比

对比项目	ECC 加密算法	RSA 加密算法
密钥长度	256 位	2048 位
CPU 占用	较少	较高
内存占用	较少	较高
网络消耗	较低	较高
加密效率	较高	一般
破解难度	具有数学特性，破解难度大	难破解，但相对 ECC 理论上容易一些
抗攻击性	强	一般
可扩展性	强	一般
兼容范围	支持新版浏览器和操作系统， 但存在少数不支持的平台， 例如 cPanel	广泛支持

国密算法与传统算法对比

算法

传统 SSL 证书通常是 RSA 算法（2048 位），它是目前最有影响力和最常用的公钥加密算法，能抵抗已知的绝大多数密码攻击。但是随着密码技术的飞速发展，证实了 1024 位 RSA 算法存在着被攻击的风险，现已升级到 2048 位 RSA 算法。

现阶段的国密 SM2 证书采用的是 ECC 算法（256 位），由国家密码管理局于 2010 年 12 月发布，是我国自主设计的公钥密码算法，在椭圆曲线密码理论基础进行改进而来，其加密强度比 RSA 算法（2048 位）更高。

安全性能

虽然 RSA 算法在目前的 SSL 证书市场中依然占据着主流地位，但是随着计算机技术的发展，加上对因子分解的改进，对低位数的密钥攻击已成为可能。

目前基于 ECC 算法的 SM2 算法普遍采用 256 位密钥长度，它的单位安全强度相对较高，在工程应用中比较难以实现，破译或求解难度基本上是指数级的。因此，**ECC 算法可以用较少的计算能力提供比 RSA 算法更高的安全强度，而所需的密钥长度却远比 RSA 算法低。**

此外，为了不断提高安全强度，必须增加密钥长度，ECC 算法密钥长度增长速度较慢（例如：224-256-384），而 RSA 算法密钥长度则需呈倍数增长（例如：1024-2048-4096）。

传输速度

在通讯过程中，**更长的密钥意味着必须来回发送更多的数据以验证连接。** 256 位的 SM2 算法相对于 2048 位的 RSA 算法*，可以传输更少的数据，也就意味着更少的传输时间。

经国外有关权威机构测试，在 Web 服务器中采用 SM2 算法，**Web 服务器新建并发处理响应时间比 RSA 算法快十几倍。**

国密算法在设计时，RSA2048 是主流签名算法，所以这里暂不讨论 ECDSA 等算法。

国密算法在车云通信中的应用

国密算法在车云通信中主要用于对传输协议加解密：车机端作为发送端，一般数据都是用 SM4 对数据内容加密，使用 SM3 对内容进行摘要，再使用 SM2 对摘要进行签名；消息 Broker 作为接收端，先用 SM2 对摘要进行验签，验签成功后就做到了防抵赖，对发送过来的内容进行 SM3 摘要，确认生成的摘要和验签后的摘要是否一致，用于防篡改。

另外 SM4 在加密解密需要相同的密钥，可以通过编写密钥交换模块实现生成相同的密钥，用于 SM4 对称加密。

关于非对称算法还要注意几点：

1. 公钥是通过私钥产生的。
2. 公钥加密，私钥解密是加密的过程。
3. 私钥加密，公钥解密是签名的过程。

由于 SM4 加解密的分组大小为 128 比特，故对消息进行加解密时，若消息长度过长，需要进行分组，若消息长度不足，则要进行填充。

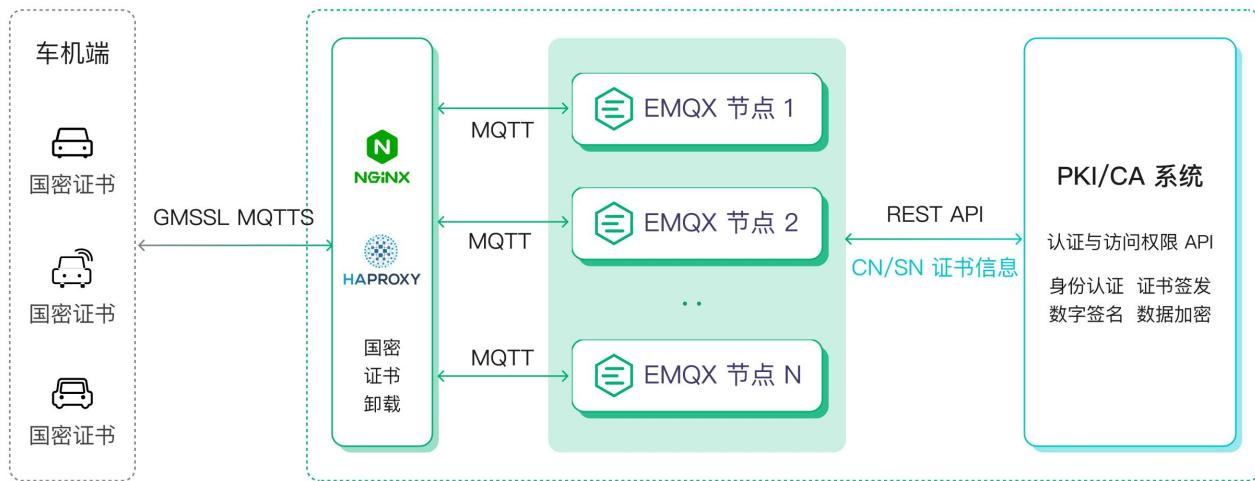
EMQ 基于国密算法的传输加认证集成方案

当前 EMQ 支持两种国密证书集成方案。

一种是在 EMQX 上通过插件的方式开发一个国密认证 Java Gateway；

另一种是通过 C 语言的 GmSSL SDK 对原生的 Nginx/HAProxy 两种主流的 LB 代理软件

进行编译扩展，使其具备 GmSSL 证书认证卸载的能力。我们更加推荐该方案。



下面我们将介绍如何使用两种代理软件解决国密证书支持。

Nginx 编译扩展 GmSSL

1. 解压 GmSSL: `tar -xvf gmssl_opt_xxx.tar.gz -C /usr/local`
2. 解压 nginx: `tar -xvf nginx-x.xxx.tar.gz`
3. 进入 `cd nginx-x.xxx` 目录
4. 编辑 `auto/lib/openssl/conf`, 将全部 `$OPENSSL/.openssl/` 修改为 `$OPENSSL/` 并保存
5. 编译配置

```
./configure \
--prefix=/usr/local/nginx \
--sbin-path=/usr/local/nginx/sbin/nginx \
--conf-path=/usr/local/nginx/conf/nginx.conf \
--error-log-path=/usr/local/nginx/log/nginx/error.log \
--http-log-path=/usr/local/nginx/log/nginx/access.log \
--pid-path=/var/run/nginx.pid \
--lock-path=/var/run/nginx.lock \
--http-client-body-temp-path=/usr/local/nginx/client_temp \
--http-proxy-temp-path=/usr/local/nginx/proxy_temp \
```

```
--http-fastcgi-temp-path=/usr/local/nginx/fastcgi_temp \
--http uwsgi-temp-path=/usr/local/nginx/uwsgi_temp \
--http scgi-temp-path=/usr/local/nginx/scgi_temp \
--without-http_gzip_module \
--with-http_ssl_module \
--with-http_realip_module \
--with-http_addition_module \
--with-http_sub_module \
--with-http_dav_module \
--with-http_flv_module \
--with-http_mp4_module \
--with-http_random_index_module \
--with-http_secure_link_module \
--with-http_stub_status_module \
--with-http_auth_request_module \
--with-threads \
--with-stream \
--with-stream_ssl_module \
--with-http_slice_module \
--with-mail \
--with-mail_ssl_module \
--with-file-aio \
--with-http_v2_module \
--with-openssl=/usr/local/gmssl \
--with-cc-opt="-I/usr/local/gmssl/include" \
--with-ld-opt="-Lm"
```

6. 编译安装：make install

7. /usr/local/nginx 即为生成的 nginx 目录

8. 配置 nginx.conf：进入到 /usr/local/nginx 目录，在 conf/nginx.conf 添加 :include
/usr/local/nginx/conf/mqtt_tcp.conf;

```

#user nobody;
worker_processes 1;

#error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;

#pid      logs/nginx.pid;

events {
    worker_connections 1024;
}

include /usr/local/nginx/conf/mqtt_tcp.conf;

http {
    include mime.types;
    default_type application/octet-stream;

    #log_format main '$remote_addr - $remote_user [$time_local] "$request" '
    #           '$status $body_bytes_sent "$http_referer" '
    #           '"$http_user_agent" "$http_x_forwarded_for"';

    access_log logs/access.log main;

    sendfile on;
    #tcp_nopush on;

    #keepalive_timeout 0;
    keepalive_timeout 65;

    #gzip on;

    server {
        listen 80;

```

添加 mqtt_tcp.conf 文件，内容如下：

- 单向认证

```

stream {
    log_format proxy '$remote_addr [$time_local] '
                    '$protocol $status $bytes_sent $bytes_received '
                    '$session_time "$upstream_addr" '
                    '"$upstream_bytes_sent" "$upstream_bytes_received" "$upstream_conne
ct_time"';

    access_log /usr/local/nginx/log/tcp-access.log proxy;
    open_log_file_cache off;
    upstream mqtt_tcp_server {
        server 192.168.0.239:1883;      #高可用均衡配置
        #server 172.17.0.4:1883;
    }
    server {
        listen 1883; #监听端口也可以使用 1883
        #ssl_verify_client on;

```

```

#      proxy_connect_timeout 150s;
#      proxy_timeout 350s;
#      proxy_next_upstream on;

proxy_pass mqtt_tcp_server; #反向代理地址

#      proxy_buffer_size 3M;
#tcp_nodelay on;
proxy_protocol on;
}

server {
    listen 8083 ssl;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:AES128-SHA:DES-CBC3-SHA:E
CC-SM4-CBC-SM3:ECDHE-SM4-GCM-SM3;
    ssl_verify_client off;
    ssl_certificate /usr/local/nginx/cert/Tsp_Server_Test_210906_sign.cer;
    ssl_certificate_key /usr/local/nginx/cert/Tsp_Server_Test_210906_sign.key;
    ssl_certificate_key /usr/local/nginx/cert/Tsp_Server_Test_210906_enc.key;
    ssl_certificate /usr/local/nginx/cert/Tsp_Server_Test_210906_enc.cer;
    proxy_pass mqtt_tcp_server;
    proxy_protocol on;
}
}

```

- 双向认证

注：双向认证比单向认证多了 `ssl_client_certificate`: 这个是 CA 证书，将签名 CA 和密钥 CA 合并到一个文件，同时 `ssl_verify_client` 设置为 `on`。

```

stream {
    log_format proxy '$remote_addr [$time_local] '
                    '$protocol $status $bytes_sent $bytes_received '
                    '$session_time "$upstream_addr" '
                    '"$upstream_bytes_sent" "$upstream_bytes_received" "$upstream_conne
ct_time"';
    access_log /usr/local/nginx/log/tcp-access.log proxy;
    open_log_file_cache on;
    upstream mqtt_tcp_server {

```

```

server 192.168.0.239:1883;      #高可用均衡配置
#server 172.17.0.4:1883;
}
server {
    listen 1883; #监听端口 也可以使用 1883
    #ssl_verify_client on;
    # proxy_connect_timeout 150s;
    # proxy_timeout 350s;
    # proxy_next_upstream on;
    proxy_pass mqtt_tcp_server; #反向代理地址
    # proxy_buffer_size 3M;
    #tcp_nodelay on;
    proxy_protocol on;
}
server {
    listen 8083 ssl;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:AES128-SHA:DES-CBC3-SHA:ECC-SM4-CBC-SM3:ECDHE-SM4-GCM-SM3;
    ssl_verify_client on;
    ssl_client_certificate /usr/local/nginx/cert/ca.pem;
    ssl_certificate /usr/local/nginx/cert/Tsp_Server_Test_210906_sign.cer;
    ssl_certificate_key /usr/local/nginx/cert/Tsp_Server_Test_210906_sign.key;
    ssl_certificate_key /usr/local/nginx/cert/Tsp_Server_Test_210906_enc.key;
    ssl_certificate /usr/local/nginx/cert/Tsp_Server_Test_210906_enc.cer;
    proxy_pass mqtt_tcp_server;
    proxy_protocol on;
}
}

```

9. 启动

在 /usr/local/nginx 下操作

启动: ./sbin/nginx

重启: ./sbin/nginx -s reload

停止: ./sbin/nginx -s stop

验证配置文件是否正确: ./sbin/nginx -t

HAProxy 集成 GmSSL 编译扩展

1. 解压 GmSSL: tar -xvf gmssl_opt_xxx.tar.gz -C /usr/local

2. 解压 HAProxy: tar -xvf haproxy_xxx.tar.gz

3. 进入 HAProxy 安装目录修改 (不要和后面编译生成的运行目录同一目录)

- 修改 makefile:

注释:

```
#OPTIONS_LDFLAGS += $(if $(SSL_LIB),-L$(SSL_LIB)) -lssl -lcrypto
```

新增:

```
OPTIONS_LDFLAGS += $(SSL_LIB)/libssl.aOPTIONS_LDFLAGS += $(SSL_LIB)/libcrypto.a  
-lm -lpthread -ldl
```

```
ifeq ($(USE_OPENSSL),)  
SSL_INC =  
SSL_LIB =  
# OpenSSL is packaged in various forms and with various dependencies.  
# In general -lssl is enough, but on some platforms, -lcrypto may be needed,  
# reason why it's added by default. Some even need -lz, then you'll need to  
# pass it in the "ADDLIB" variable if needed. If your SSL libraries are not  
# in the usual path, use SSL_INC=/path/to/inc and SSL_LIB=/path/to/lib.  
OPTIONS_CFLAGS += $(if $(SSL_INC),-I$(SSL_INC))  
#OPTIONS_LDFLAGS += $(if $(SSL_LIB),-L$(SSL_LIB)) -lssl -lcrypto  
OPTIONS_LDFLAGS += $(SSL_LIB)/libssl.a  
OPTIONS_LDFLAGS += $(SSL_LIB)/libcrypto.a -lm -lpthread -ldl
```

- 修改源码:

文件 src/ssl_sock.c 备注: 不能直接修改红色内容, 还需要更换位置, 按照修改的顺序

函数 `ssl_sock_put_ckch_into_ctx` 将以下代码

```

if (SSL_CTX_use_PrivateKey(ctx, ckch->key) <= 0) {
    memprintf(err, "%sunable to load SSL private key into SSL Context '%s'.\n", err &
& *err ? *err : "", path);
    errcode |= ERR_ALERT | ERR_FATAL; return errcode;
if (!SSL_CTX_use_certificate(ctx, ckch->cert)) {
    memprintf(err, "%sunable to load SSL certificate into SSL Context '%s'.\n", err &&
*err ? *err : "", path);
    errcode |= ERR_ALERT | ERR_FATAL;
    goto end;
}

```

修改为：

```

if (!SSL_CTX_use_certificate_file(ctx, path, SSL_FILETYPE_PEM)) {
    memprintf(err, "%sunable to load SSL certificate into SSL Context '%s'.\n", err &&
*err ? *err : "", path);
    errcode |= ERR_ALERT | ERR_FATAL;
    goto end;
}

if (SSL_CTX_use_PrivateKey_file(ctx, path, SSL_FILETYPE_PEM) <= 0) {
    memprintf(err, "%sunable to load SSL private key into SSL Context '%s'.\n", err &
& *err ? *err : "", path);
    errcode |= ERR_ALERT | ERR_FATAL; return errcode;
}

```

```

static int ssl_sock_put_ckch_into_ctx(const char *path, const struct cert_key_and_chain *ckch, SSL_CTX *ctx, char **err)
{
    int errcode = 0;
    STACK_OF(X509) *find_chain = NULL;

    if (!SSL_CTX_use_certificate_file(ctx, path, SSL_FILETYPE_PEM)) {
        memprintf(err, "%sunable to load SSL certificate into SSL Context '%s'.\n",
                err && *err ? *err : "", path);
        errcode |= ERR_ALERT | ERR_FATAL;
        goto end;
    }

    if (SSL_CTX_use_PrivateKey_file(ctx, path, SSL_FILETYPE_PEM) <= 0) {
        memprintf(err, "%sunable to load SSL private key into SSL Context '%s'.\n",
                err && *err ? *err : "", path);
        errcode |= ERR_ALERT | ERR_FATAL;
        return errcode;
    }

    if (ckch->chain) {
        find_chain = ckch->chain;
    }
}

```

顺序一定不要搞反了

4. 编译

编译前可能需要提前安装相关依赖：

```
yum install pcre-devel zlib-devel
```

```
make TARGET=linux31 USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1 USE_CRYPT_H=1  
USE_LIBCRYPT=1 SSL_INC=/root/gmssl/gmssl/include SSL_LIB=/root/gmssl/gmssl/lib
```

注：SSL_INC 和 SSL_LIB 指定 gmssl 解压的路径

5. 安装

```
make install PREFIX=/usr/local/haproxy
```

注：PREFIX=/usr/local/haproxy 是编译生成的运行目录，不要和安装目录同目录

6. 配置

- 证书准备：

将签名证书 pem 文件和签名私钥 pem 文件合并成 XXX_sig.pem，文件名必须以 sig.pem 结尾

```
[root@master01 cert]# ls  
ca.pem      root.cer      server_sig.pem          Tsp_Server_Test_210906_enc.key  Tsp_Server_Test_210906_sign.key  
issuer.cer  server_enc.pem  Tsp_Server_Test_210906_enc.cer  Tsp_Server_Test_210906_sign.cer  
[root@master01 cert]# cat Tsp_Server_Test_210906_sign.cer Tsp_Server_Test_210906_sign.key >> server_sig.pem
```

将加密证书 pem 文件和加密私钥 pem 文件合并成 XXX_enc.pem，文件名必须以 enc.pem 结尾

XXX_enc.pem 将被隐式加载，且必须放到 XXX_sig.pem 的相同目录下，比如：
`/usr/local/keystore/server_enc.pem`

需要双向认证的时候：CA 证书合并到一个文件（选做）

```
[root@master01 cert]# ls
ca.pem      root.cer      server_sig.pem          Tsp_Serv
issuer.cer  server_enc.pem Tsp_Server_Test_210906_enc.cer  Tsp_Serv
[root@master01 cert]# cat root.cer issuer.cer >>ca.pem
```

- HAProxy.conf:

```
global
  daemon
  ssl-default-bind-ciphers ECC-SM4-CBC-SM3:ECC-SM4-GCM-SM3
  #ssl-default-bind-options no-sslv3
  maxconn 256
  log 127.0.0.1 local7 info
defaults
  mode tcp
  log global
  option tcplog
  timeout connect 5000ms
  timeout client 50000ms
  timeout server 50000ms
  stats uri /status
  #stats auth zp:123456
frontend emqx_dashboard
  bind *:18083
  option tcplog
  mode tcp
  default_backend emqx_dashboard_back
frontend emqx_tcps
  bind *:8883 ssl crt /usr/local/haproxy/cert/server_sig.pem ca-file /usr/local/
  haproxy/cert/ca.pem verify required
  option tcplog
  mode tcp
  default_backend backend_emqx_tcp
frontend emqx_tcp
  bind *:1883
  option tcplog
  mode tcp
  default_backend backend_emqx_tcp
frontend frontend_emqx_ws
  bind *:8083
  option tcplog
```

```

#      option forwardfor
    mode tcp
    default_backend backend_emqx_ws
backend emqx_dashboard_back
    balance roundrobin
    server emqx_node_1 192.168.92.120:18083 check
backend backend_emqx_tcp
    mode tcp
balance roundrobin
    server emqx_node_1 192.168.92.120:1883 check-send-proxy send-proxy-v2-s
sl-cn
backend backend_emqx_ws
    mode http
    option forwardfor
    balance roundrobin
    server emqx_node_1 192.168.92.120:8083 check-send-proxy send-proxy-v2 c
heck inter 10s fall 2 rise 5

```

```

#stats auth zp:123456
frontend emqx_dashboard
    bind *:18083
    option tcplog
    mode tcp
    default_backend emqx_dashboard_back
frontend emqx_tps
    bind *:8883 ssl crt /usr/local/haproxy/cert/server_sig.pem ca-file /usr/local/haproxy/cert/ca.pem verify required
    option tcplog
    mode tcp
    default_backend backend_emqx_tcp
frontend emqx_tcp
    bind *:1883
    option tcplog
    mode tcp
    default_backend backend_emqx_tcp
frontend frontend_emqx_ws
    bind *:8083
    option tcplog
    option forwardfor
    mode tcp
    default_backend backend_emqx_ws

backend emqx_dashboard_back
    balance roundrobin
    server emqx_node_1 192.168.159.120:18083 check
backend backend_emqx_tcp
    mode tcp
    balance roundrobin
    server emqx_node_1 192.168.159.120:1883 check-send-proxy send-proxy-v2-ssl-cn
backend backend_emqx_ws
    mode http
    option forwardfor
    balance roundrobin
    server emqx_node_1 192.168.159.120:8083 check-send-proxy send-proxy-v2 check inter 10s fall 2 rise 5
root@master01:~# 

```

双向认证

- 启动测试

假如配置文件放在：/usr/local/haproxy/conf/ 下

启动命令：/usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/conf/haproxy.cfg

小结

本文为大家介绍了国密算法的基础背景知识以及其在车联网场景中的应用，同时介绍了 EMQ 基于国密算法的传输加密认证集成方案，通过本文提供的配置操作示例，**读者可以尝试在车联网平台中使用国密认证，进一步增加平台安全性。**

EMQ 车联网 GmSSL 集成解决方案紧密对接车联网产业对网络安全、数据安全的迫切需求，为车联网通信安全提供了有力保障。目前已经部署应用在车联网平台安全认证和 V2X 车路协同等多个车云安全通信场景。

08 | 实现车联网灵活数据采集

随着车联网与 5G 技术的融合以及车辆智能化的发展，车联网的数据采集需求呈现爆发式增长。传统的车辆数据采集主要用于车辆的远程监测和故障诊断。随着车辆应用的丰富和智能化水平的提高，车辆数据采集逐渐应用到更多的场景，如研发用数据采集、数据统计和分析、规则引擎与报警系统、车辆实时控制等。

现有的数据采集方案往往通过车载数据采集终端（T-BOX）固件中的采集功能或自行编写的采集程序进行车辆数据采集。通常采集程序所采集到的车身信息是固定且直接固化在车载终端上的。在智能车联网时代之前，采集的数据种类少、全量采集压力不大，这种做法是可行的。

随着车联网技术的发展，车辆整车网络构成也越来越复杂，可采集的车身信息多样化，全量采集数据量过大而且浪费宝贵的带宽资源，因此需要根据 TSP 应用的需求按需进行采集。此外，不同车型的汽车通常会有不同的数据，例如 CAN 总线的数据在不同车型上会有不同的 DBC 文件。固定采集程序无法移植，必须重新编写，并 OTA 升级采集程序。

总的来讲，固定采集程序存在以下问题：

- 采集的数据固定，无法灵活变更采集的项目。实际上随着车联网的发展，数据采集项目将根据应用呈现更多变化，固定的采集方式无法满足经常变动的真实需求。
- 采集信息解析配置固定，无法匹配新的车型或总线数据变化。例如，采集 CAN 总线数据的采集程序，无法变更 DBC 文件以匹配总线数据的变化。
- 扩展不易，新的传感器或总线协议需要重新开发。

注：DBC (Data Base CAN) 文件是由德国 Victor 公司发布的，它被用来描述单一 CAN 网络中各逻辑节点信息，依据该文件可以开发出监视和分析 CAN 网络中所有逻辑节点的运行状态。

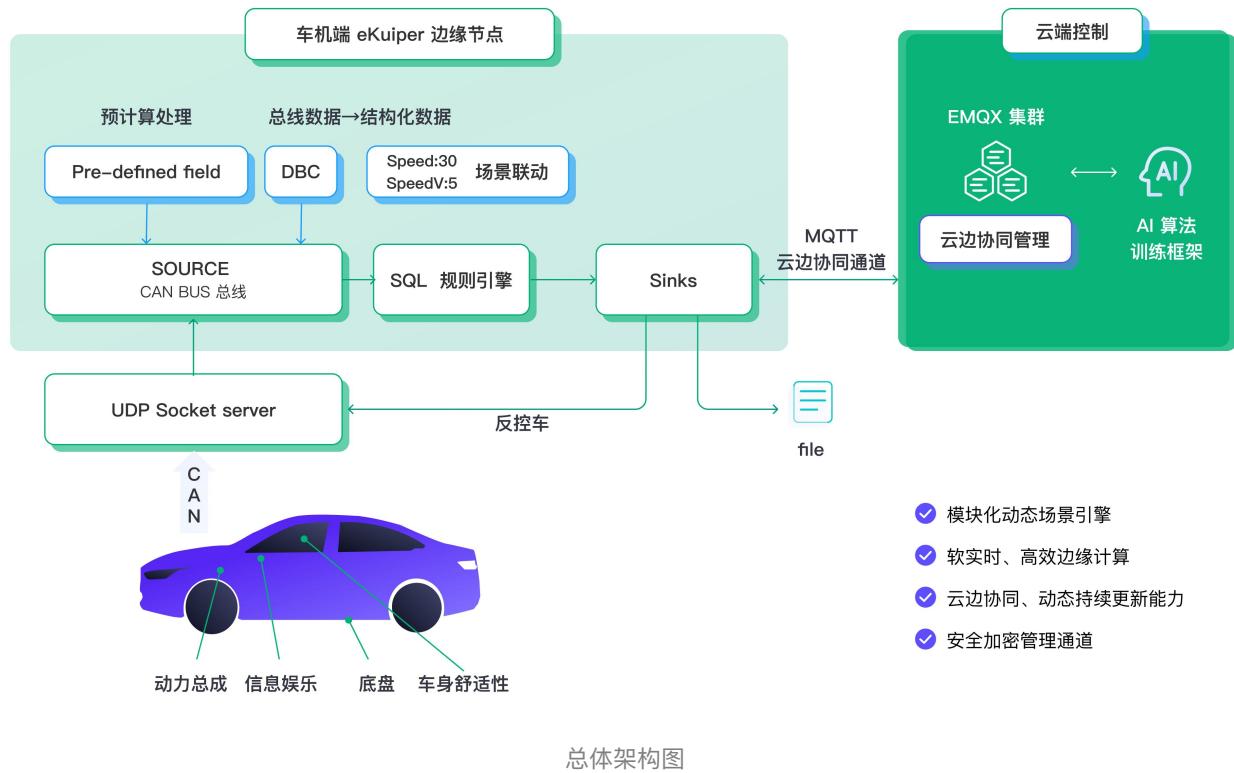
因此，业界迫切需要一种灵活的、支持弱网情况的数据采集方案。

如何实现灵活数采

针对固定数采程序缺陷，我们需要一个灵活数据采集引擎，并具备以下能力：

- 灵活数据埋点配置和规则，并可热更新和热启停数据采集规则。
- 多数据源对接和解析能力，例如 CAN 总线、HTTP 信号等。
- 灵活配置数据源解析的能力。以 CAN 总线为例，应当支持 DBC 文件的灵活加载和更新。
- 采集数据灵活分发的能力。可根据业务创建规则，将一部分数据本地保存，一部分数据回传云端。
- 弱网工况下，采集数据高效回传的能力。
- 足够轻量高效，从而可以运行在多种车型，包括车机资源受限的车型上。

基于大量的车联网用户案例和经验，EMQ 推出了基于 eKuiper 与 QUIC 协议的车云系统方案，实现了一套易部署可移植的车联网灵活数采方案。方案架构如下图所示。



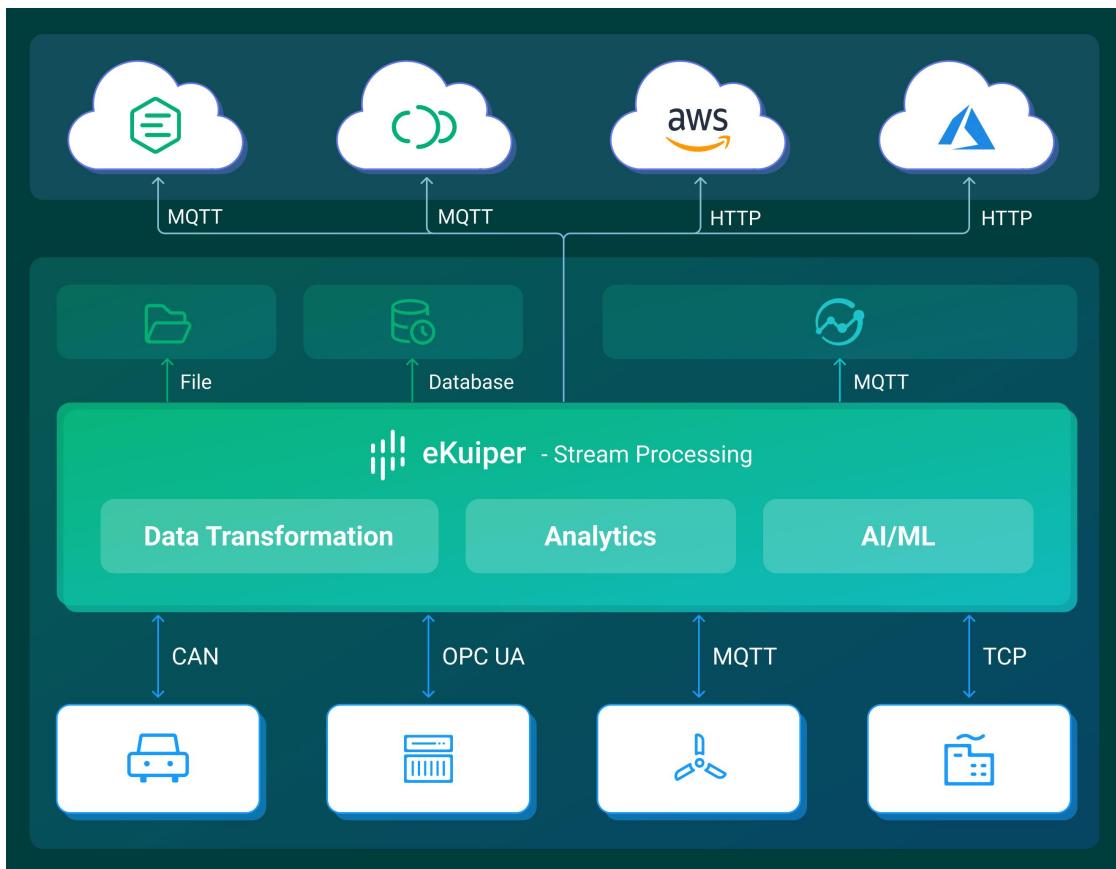
在该方案中，我们采用开源边缘流式处理引擎 eKuiper 实现车载终端上的灵活数据采集功能，采用大规模分布式物联网 MQTT 消息服务器 EMQX 实现采集数据的连接、移动和处理以及车云一体的控制指令交互。在前文中，我们已经详细介绍了基于 EMQX 的车联网消息平台的架构设计，这里不再赘述。接下来，我们将以 eKuiper 为例，介绍如何实现车联网灵活数采。

灵活数采方案剖析

LF Edge eKuiper（简称 eKuiper）是开源的超轻量数据分析和流式计算引擎。eKuiper 兼容性强，可适配 X86 和 ARM 等多种 CPU 架构的车机终端。软件较轻量且可裁剪，核心包和初始运行内存在 10MB 级别，支持高吞吐低时延的各种数据处理和计算。在方案中，以部署于车机端的 eKuiper 为核心，可以实现近实时的灵活数据采集和处理转发等功能。

如下图所示，eKuiper 具备了数据流的接入、处理和流转能力。南向部分，eKuiper 支持接入各种协议的数据流，例如 CAN 总线、MQTT 协议、Socket (TCP 或 UDP) 和 DDS 等。接入的数据可以在引擎内部根据用户定义的规则，进行数据的采集、转换、过滤和分析等数据

处理工作，之后再将采集或处理的结果发送到各种北向的目的地中，例如存到本地的文件、数据库中以便后续车载应用使用；或是通过 HTTP/MQTT 等协议发送到云端或 TSP 应用端进行处理。



在灵活数采的场景中，假设 eKuiper 已部署到车机中，要完成一个数采任务，一般只需要两个步骤：

1. 接入数据流
2. 建立采集规则

在 eKuiper 中，这两个步骤无需编写代码，可使用 SQL 语句或者可视化 Flow 编辑器进行配置。

数据流接入

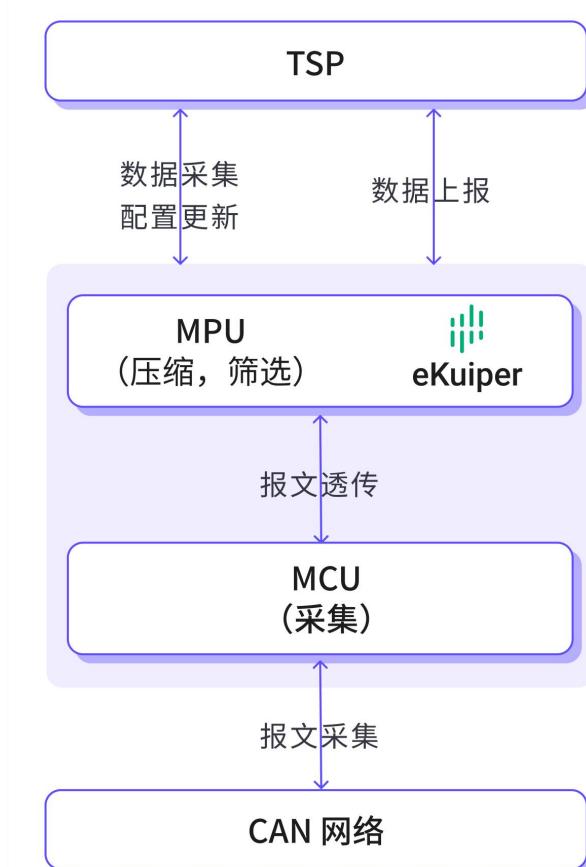
CAN (Controller Area Network) 是最常见的车联网总线网络。本文以接入和解析 CAN 数据为例，介绍 eKuiper 如何实现车载数据流的接入。

在 eKuiper 中提供了 CAN 数据源，其中主要实现了两个能力：

1. 连接协议
2. 根据 DBC 解码 CAN 报文

连接协议支持

若 eKuiper 可以直接连接 CAN 总线，则可通过 CAN 协议建立到车载总线的连接，获取总线数据。出于安全的原因，eKuiper 也经常被部署到与总线隔离的硬件上。例如，eKuiper 部署在 MPU 中；而在 MCU 中部署 CAN 总线连接应用，通过 TCP、UDP 或者 MQTT 协议等，将报文透传出来。eKuiper 同样支持通过这些协议进行连接，获取总线数据报文。



车端数据流向图

灵活 CAN 报文解码

我们从总线接收到的报文为二进制编码的数据，人类难以阅读。CAN DBC 是一种文本文件，用于 CAN 报文的描述文件。通过读取 DBC 的描述信息，我们可以把 CAN 报文的数据解析为物理值的信息。例如，一段 CAN ID 0x208 的数据 0x0500000000000000，根据 DBC 的描述信息可能解析为一系列信号，可表示为键值对或者 JSON 串 {"temperature":10, "voltage": 100} 等。

eKuiper 的 CAN 数据源可将数据解析为可读的键值对，这样编写数据采集规则的时候，可以直接选取可读的信号，大大简化采集逻辑的编写。CAN 报文解析功能具有极大的灵活性，可以动态地更新 DBC 文件以适配不同的车型或升级车载总线数据而无需编码。

CAN 报文解析的灵活性主要体现在如下方面：

- DBC 文件可配置，可热更新
- 支持多个 DBC 文件
- 支持 CAN FD 格式
- 支持白名单和 container ID 映射

基于灵活的报文解码支持，当总线数据结构改变或者更改车型时，仅需更新 DBC 文件即可适配。

在 eKuiper 中，流 (stream) 是用于定义数据接入的一个实体。我们可使用如下的 SQL 语句，定义一个接入 CAN 总线的数据流。

```
CREATE STREAM canDemo () WITH ( Type="can",  CONF_KEY="test",
SHARED="TRUE")
```

该语句定义了一个名为 canDemo 的流，其类型为 can，即接入 CAN 总线的数据源类型；CONF_KEY 表示接入配置定义在名为 test 的配置中，其中可配置使用的 DBC 文件地址等；

SHARED 设置为 true，表示使用该数据流的所有规则共享一份数据，确保解码只会进行一次。

该流将接入解析 CAN 总线数据，得到 JSON 数据流。接下来，应用开发人员可以在其上创建多条规则，定义如何采集数据。

接入扩展

随着汽车智能化程度的提高，车载的传感器和数据总线的数量和种类越来越多。eKuiper 提供了扩展机制，用户可以编写插件实现新的协议或私有协议的接入和解析。安装后的插件遵循使用逻辑，应用开发人员可以与使用原有的数据流类型相同的方法创建数据流。

灵活配置采集规则

前文中我们已经创建了连接 CAN 总线的数据流，接下来我们可以建立多个数据采集规则进行灵活的数采。本节介绍一些常见的采集规则。规则内容为 JSON 文本数据，可通过 REST API 等方式进行规则的动态下发管理，具体管理方法将在下一节介绍。

eKuiper 的规则分为两个部分，其中 SQL 用于编写业务逻辑，例如需要采集哪些数据、对数据做哪些处理；Actions 部分用于描述规则命中后执行的动作，例如存储到本地文件或者发送到云端 MQTT 的某个主题中。假设上一节创建的数据流 canDemo 中，总线中的数据解析为包含发动机相关数据如发动机转速（rpm）、进气温度（inletTemperature）、进气压力（inletPressure）以及电池相关数据，如电池电压（voltage），电池电流（current）等数据的键值对数据，如 `{"rpm":2000,"inletTemperature":230,"inletPressure":27,"voltage":15,"currency":2}`。下列配置的规则将针对这个总线数据进行采集。

1. 采集指定的信号

本规则可实时采集发动机的信号并发送到 MQTT topic collect 中。规则通过 SQL 语句中的 SELECT 子句定义了需要采集的数据点。

```
{
  "id": "ruleCollect",
  "sql": "SELECT rpm, inletTemperature, inletPressure FROM canDemo",
  "actions": [
    {
      "mqtt": {
        "server": "tcp://yourserver:1883",
        "topic": "collect"
      }
    }
  ]
}
```

2. 采集有变化的信号

某些信号可能变化周期比较长，全部采集的话大部分为重复值，占据存储和带宽。eKuiper 提供了内置的变化采集函数 `CHANGED_COLS`，可以仅采集信号数值变化的情况。下面的示例规则中，我们采集了电池的变化信息，并保存在本地文件中。

```
{
  "id": "ruleChangeCollect",
  "sql": "SELECT CHANGED_COLS(\"\", true, voltage, currency) FROM canDemo",
  "actions": [
    {
      "file": {
        "path": "/tmp/cell"
      }
    }
  ]
}
```

3. 根据事件采集

某些信号只有在特定的情况下才需要采集，例如碰撞后采集相关的数据。eKuiper 中可以灵活设置采集的条件。以下的规则中，当电池电压异常（不在 10 到 20 之间）的情况下，采集所有数据到 MQTT 的 Topic `exception` 中。

```
{
  "id": "ruleExpCollect",
  "sql": "SELECT * FROM canDemo WHERE voltage NOT BETWEEN 10 AND 20 ",
```

```

"actions": [
    "mqtt": {
        "server": "tcp://yourserver:1883",
        "topic": "exception"
    }
]
}

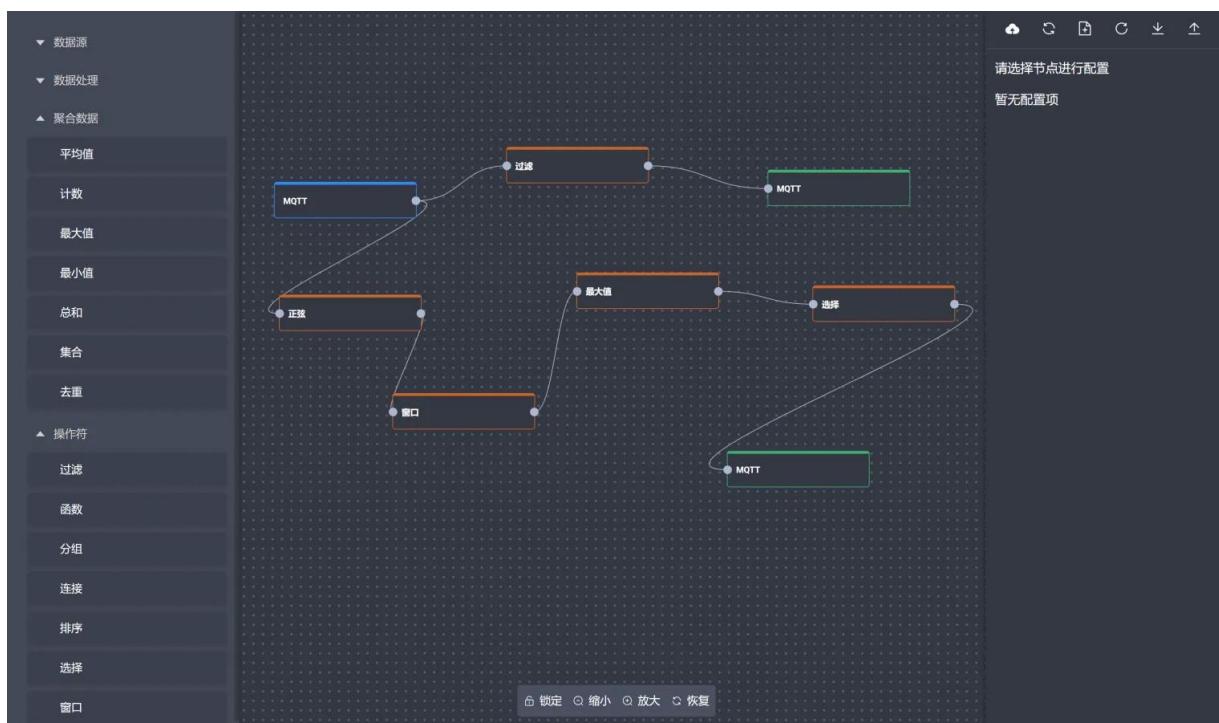
```

车云一体规则管理

规则构思完成后，需要进行动态下发和管理。EMQ 提供了车云一体的规则管理控制台，用户可以在里面进行规则的编写、下发和状态管理。管理控制台可以在云端集中管理多个车机边缘节点。

规则编写

管理控制台上提供了规则编写的图形界面。如下图所示，用户可以在界面上填入规则的 ID、SQL 和动作等。提交后，规则即可下发到对应车机节点。



规则编辑界面

我们也将提供规则编写的可视化 Flow 编辑器界面。用户可采用拖拽的方式编写自己的业务规则。

规则管理

eKuiper 中的规则都是可灵活管理的。规则可以热添加、热更新和热启停。在管理控制台中，用户可以查看规则的运行状态，进行规则的修改、启停、删除等操作。

ID	Start/Stop	Operations
ruleCollect	<button>Start</button>	
ruleSOC	<button>Start</button>	
ruleTF	<button>Start</button>	

规则管理界面

云端集中管理

通过云端可集中管理在车辆边缘端上的数据分析应用。

- 大规模在线车辆支持：基于 EMQX Enterprise，百万级别在线车辆支持
- 车辆在线自动更新应用：支持离线车辆应用更新、升级；车辆在线后自动更新应用，并报告状态
- 车辆离线后状态查询支持：车辆离线状态下可获取应用部署的最后状态
- HTTP Rest API 服务接口支持：同步 HTTP 接口，接入前端应用（web & mobile）

更多可能

eKuiper 作为一个通用的流式计算引擎，除了实现数据采集之外，还可以实现很多边缘计算功能，充分利用车载终端算力。例如，eKuiper 可支持下列功能：

- 数据变换和格式化，例如将传输信号由整型转换回浮点型，或者将信号格式化为目标系统要求的格式。
- 数据分析，例如计算一段时间内的平均值等统计值。
- SOA 服务调用，实现场景联动，例如根据车内温度自动开关空调。
- AI/ML 算法集成，例如根据采集到的信号识别用户的充电意图等。

更多功能欢迎读者们自行探索。

小结

车联网软硬件技术大发展的浪潮下，传统的固定数据采集方案难以应对层出不穷的采集需求。通过本章节介绍的基于 eKuiper 与 EMQX 的车云系统方案，可实现端到端的灵活数据采集需求。eKuiper 采用基于文本型业务处理应用下发，避免复杂 OTA 升级，帮助车联网企业实现灵活的数据采集以及高效的车云数据协同。

09 | 车联网移动场景 MQTT 通信优化

很多的车联网应用场景不但计算量巨大，而且对通信链路有非常强的低时延、低能耗和高可靠要求。传统的通信协议如 HTTP 等并不能同时满足以上要求。而作为目前物联网领域事实上的标准协议，MQTT 提供了 Pub/Sub 的消息模式，具备精简优良的协议设计，可以满足低延时和低功耗的需求，适用于资源有限的车机系统。

但不同于智能家居、机器人这类设备固定且网络环境稳定的场景，车联网中快速移动、场景切换快、网络情况复杂多变等特性，对 MQTT 协议在车端和服务端的应用提出了更高的要求。

本章节将深入分析车联网移动场景下 MQTT 消息传输面临的问题及产生原因，并利用 MQTT 协议特性对其加以解决和优化，帮助用户构建更稳定的车联网通信架构。

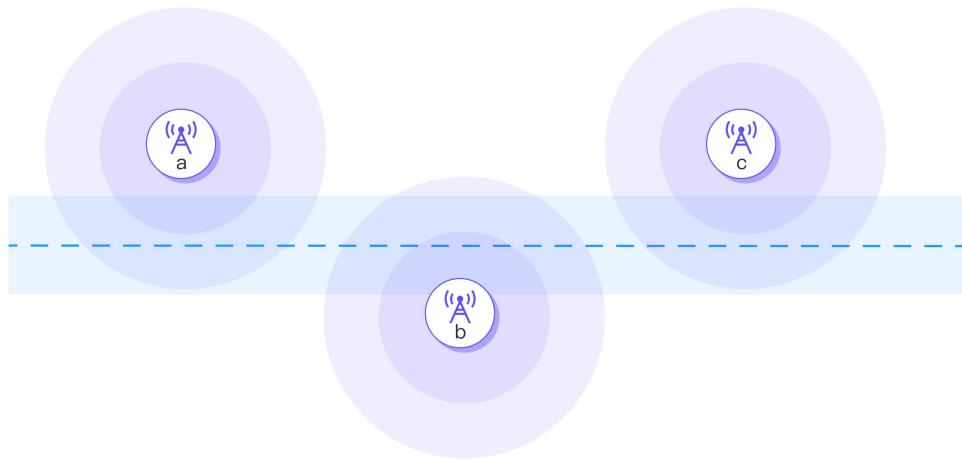
移动设备网络迁移问题

相信大家使用 4G 手机时都有过类似体验：进入地下室时信号强度突然变弱，虽然网络没有显示中断，但是实际使用体验就和断网一样；亦或是在一个很大区域的 WiFi 网络范围内移动，在不同的 AP（无线接入点）覆盖范围之间切换时也会有类似情况。这就是一个典型的移动设备导致的网络迁移问题。而在车联网中，由于车辆是高速移动，特别是在高速公路基站覆盖稀疏或穿过隧道的情况下，都会导致这种问题更加频繁地出现，从而引起车机端 MQTT 连接中断重连。

首先我们来看看车联网场景面对的网络现状：根据 2020 年底的数据，我国基站总数为 931 万个，其中 3G/4G 基站总数为 575 万个。但这些基站大多集中在城市区域，而在乡村，高速公路甚至是隧道内的信号覆盖就远没有城市那么全面。目前，针对高速公路和国道省道等区域的网络覆盖方案基本分为公网延伸覆盖和专网覆盖方案。

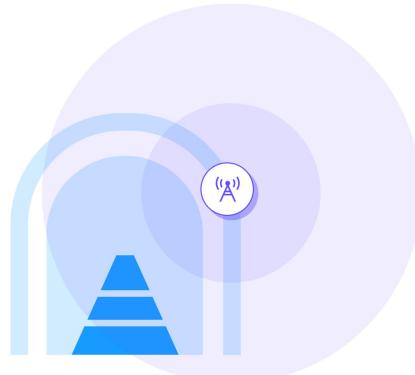
- 公网延伸覆盖：将路线区域与周边区域统一规划，使用常规基站蜂窝组网方式进行覆盖。

由于往往是直接将大网的网络资源延伸到高速公路上，所以也叫大网延伸覆盖。



公网基站覆盖示意图

- **专网覆盖：**针对特殊的点覆盖和线覆盖场景的特殊要求进行优化，配置特殊的频率、信令和功能进行异频组网。由于建设成本高，往往更多用于高速铁路沿线覆盖。专网与公网之间完全隔离，只有在特定出入口例如高速公路收费站才能进入或离开专网。



专网延伸覆盖示意图

可以发现，我们使用的网络是依靠通信从业者建设的一个个蜂窝基站提供的。而车辆在快速移动的过程中，位置更新频繁，经常会在多个基站覆盖范围之间切换。这导致其网络信令负荷大，基站切换频繁，最终将导致车载 4G 模块的网络链路中断。虽然专网覆盖可以通过采用 BBU+RRU 小区合并的技术来减少网际切换和同频干扰，进而解决这一问题，但由于专网方案建设成本高昂，所以实际场景里，车联网更多面对的还是第一种公网覆盖方案。

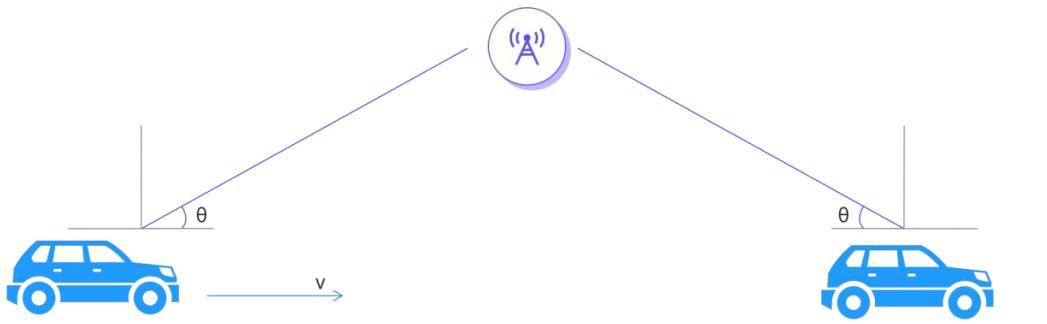
卡号	终端状态		终端IP	状态变更时间	APN	网络接入类型
	离线		10.191.0.137	2020-05-12 15:50:14	CMIOT	4G及其他
	在线		10.191.0.137	2020-05-12 13:52:03	CMIOT	4G及其他
	离线		10.141.29.109	2020-05-12 12:54:28	CMIOT	4G及其他
	在线		10.141.29.109	2020-05-12 12:04:14	CMIOT	4G及其他

从运营商提供的管理系统里查看 4G 连接的情况

于是，我们就会发现车机端的 4G 连接出现如上图所示的不断上下线的情况。

多普勒效应和隧道覆盖

除了基站覆盖带来的网络问题外，当车辆行驶速度很快的时候，也会由于多普勒效应造成延迟增加和丢包。车速越大，频偏越大，延迟越大，丢包的概率也越大。

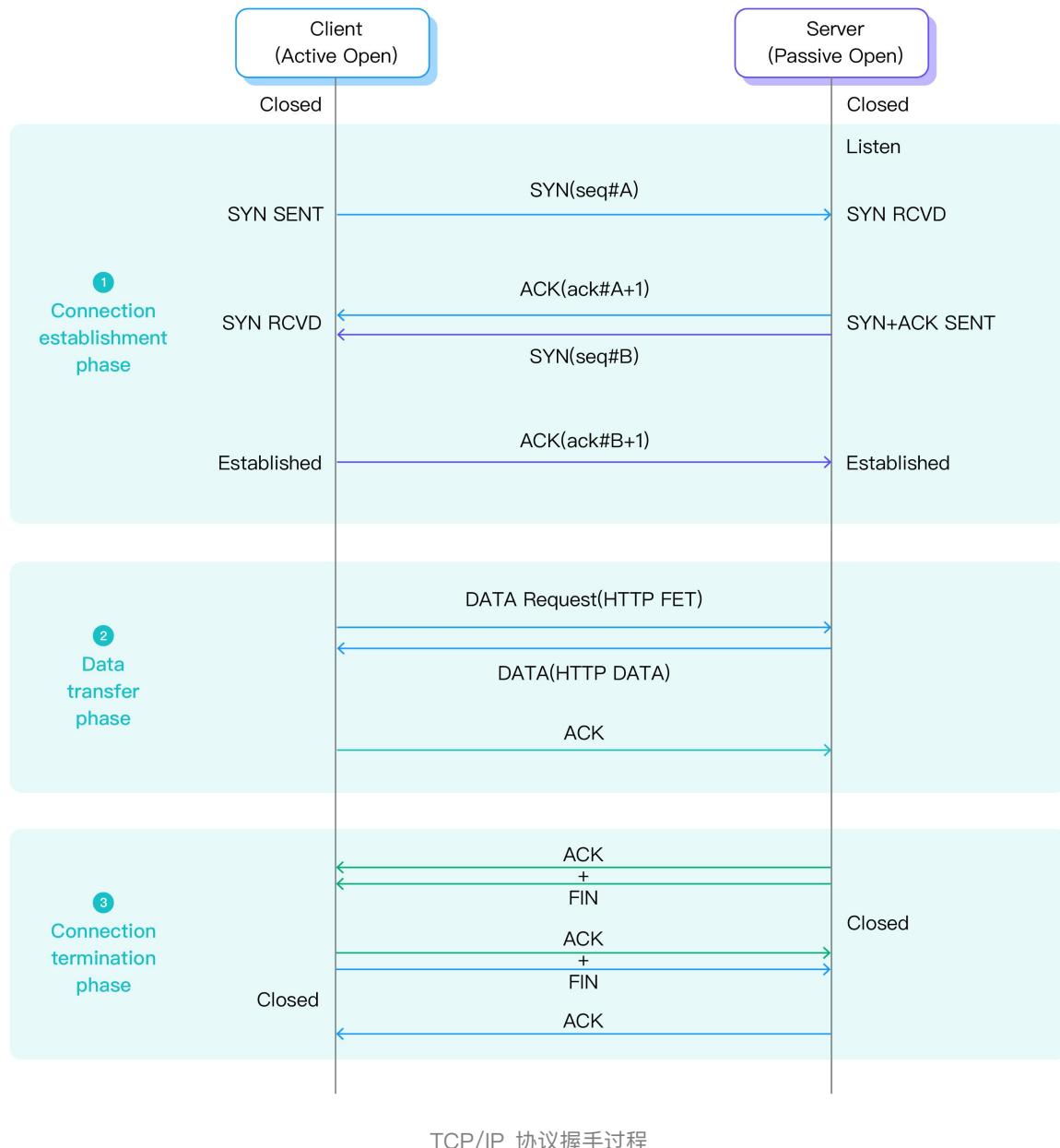


多普勒效应示意图

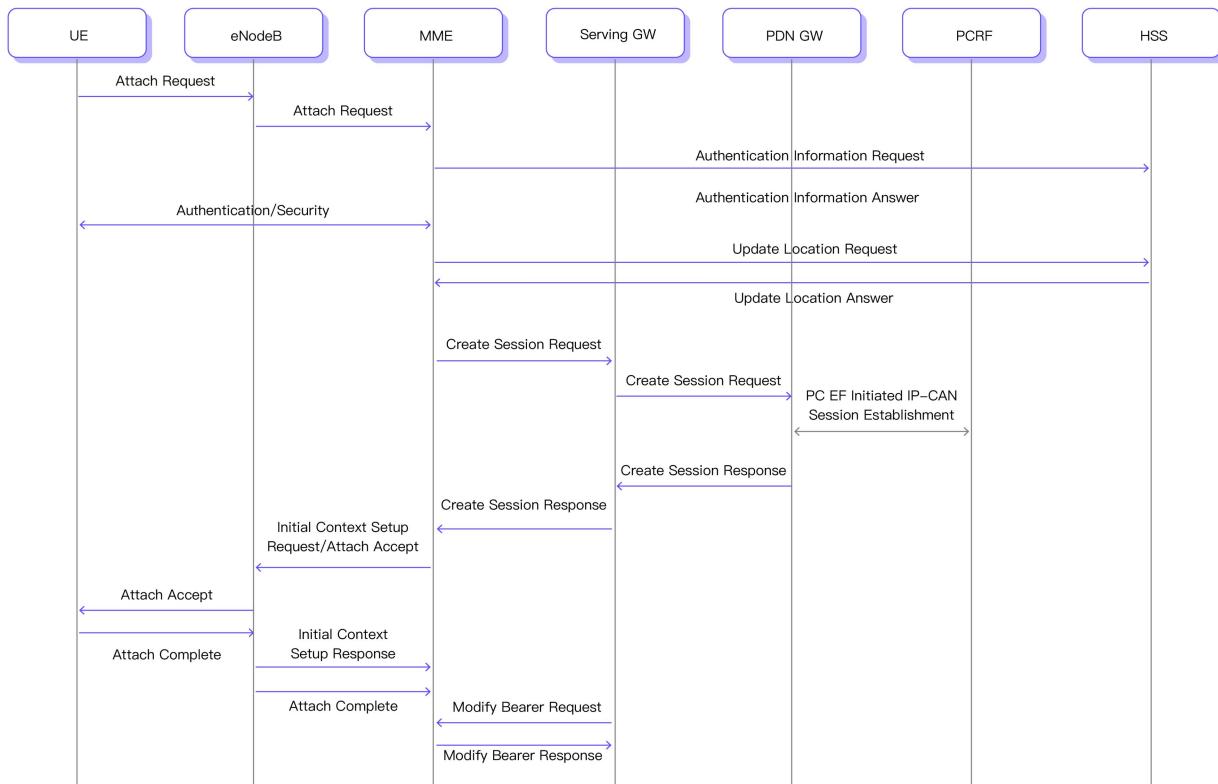
车联网移动场景下的 MQTT 连接

我们知道了车辆的网络情况，那么这些因素是如何影响车机端 MQTT 连接的呢？

众所周知，MQTT 连接也是基于 TCP/IP 协议栈。看到这里大家可能会有疑问：TCP/IP 协议栈里有连接保活机制，MQTT 协议里也有 Keep Alive 参数供连接重建恢复，哪怕基站切换导致了短暂的通信中断，但是等到进入下一个基站的范围，通信链路也很快就恢复了，那么为什么还会导致车辆设备 MQTT 连接的频繁离线呢？要回答这个疑问，我们需要结合 TCP/IP 和移动网络入网过程一起来分析。

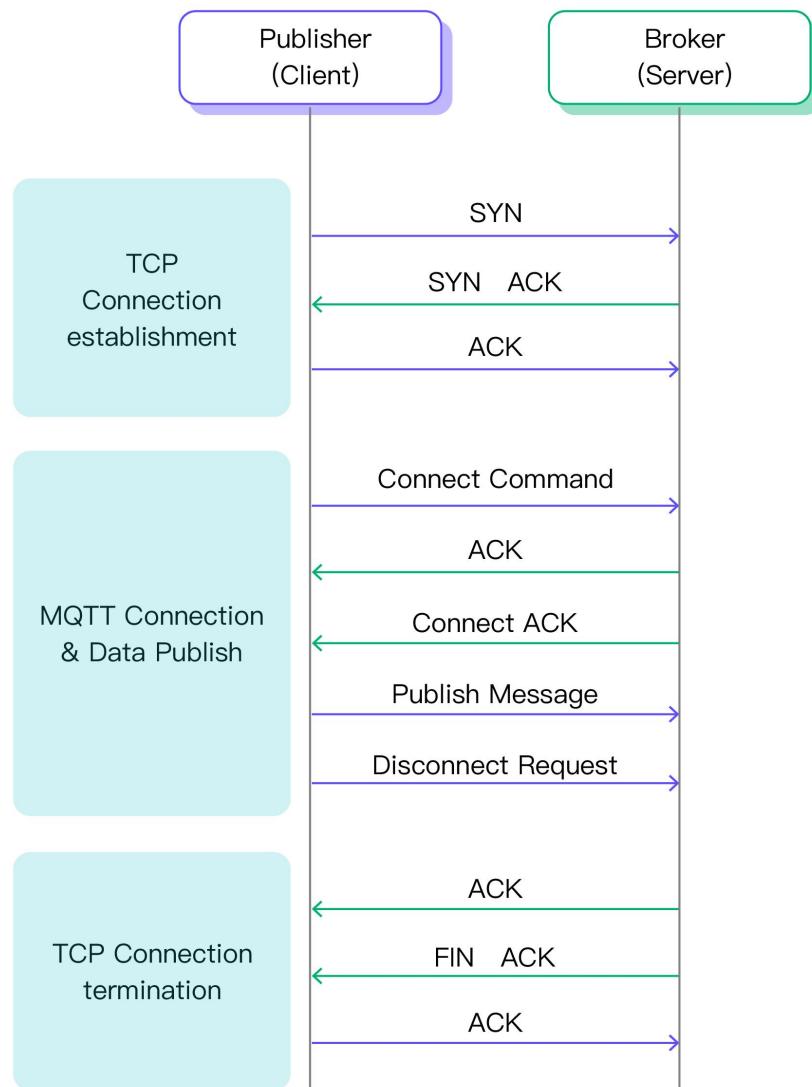


TCP/IP 协议诞生之初，主要针对的是稳定的有线网络，作为一个可靠传输协议，其内部有数据 ACK，能够进行数据重传和连接复用。但是，这一切都是基于 IP 地址不变的前提下，而在车联网场景里，基站切换是会导致车机端的 IP 地址变更的。每次车机 4G 模块进入新的基站覆盖范围时都会重新发起一次入网附着请求。



网络入网过程—UE 初始化附着到 UE-UTRAN 网络的过程

其中的协议细节这里不进行详细解释。由于目前我们仍然在使用 IPV4 标准，所以车机 4G 模块在重新入网过程中会向新搜寻到的 eNB 基站发送一个关键的信令 PDN (Packet Domain Network) 来请求为自己分配一个新的 IP 地址，而这个地址往往都是 NAT 地址，这也是 4G 终端开机即在线的技术的一环。此时还伴随着网络质量检测、APN 匹配等流程来判断终端使用的网络类型和推送网络路由以保证连通性。如果此时的边缘 eNB 基站没有针对车机端的 4G 卡和 PDN 信令进行对应优化，是无法获知其原先使用的 IP 地址的，那么这时候 IP 地址就发生了改变，需要进行 NAT 地址重绑定。而对于 MQTT 和 TCP/IP 这一类长链接协议来说，IP 地址变化后，TCP 服务端无法识别出现在客户端是否还是原先的客户端，所以 TCP 连接是必须要重新建立的，从而导致 MQTT 连接也必须重建。



TCP 连接和 MQTT 连接的关系

以上是一个正常的快速移动的车辆在蜂窝基站间正常切换会发生的过程。而实际情况中网络更加复杂，公网覆盖方案由于共享基站和接入网资源，若边缘基站负载过高还会发生 eNB 基站对 PDN 请求不响应等情况。网络侧对承载请求不响应，更不用说伪基站。此外地理环境和多普勒效应引起的多径效应和信号衰减都会导致延时增加和连接中断。

如何改善移动网络下 MQTT 连接稳定性？

清楚了问题的根源，接下来我们将借助 MQTT 协议的特性来解决上述问题，构建更稳定的车联网通信架构，避免因为连接重连和中断造成的数据丢失。

虽然 TCP/IP 部分无法改变，但 MQTT 协议提供了许多供配置的参数和消息 QoS 等级供我们配置。针对一些关键数据，比如车机端重要的状态变化和用户发出的请求，我们需要保证消息到达，这就需要我们使用 QoS 1/2。

Clean Session

首先，我们要解决 IP 更新导致 TCP 重连后客户端无法识别的问题。我们可以通过 MQTT 会话保持特性来解决。

关于 MQTT 会话状态可参考文章：<https://www.emqx.com/zh/blog/mqtt-session>。

MQTT 要求客户端与服务端在会话有效期内存储一系列与客户端标识（ClientID）相关联的状态，即会话状态。我们将从客户端向服务端发起 MQTT 连接请求开始，到连接中断直到会话过期为止的消息收发序列称为会话。因此，会话可能仅持续一个网络连接，也可能跨越多个网络连接存在。所以在这种网络切换的过程中，车机端每次连接使用相同的客户端标识，就可以让 MQTT Broker 在 TCP 连接重建的情况下，仍然可以识别到新连接是之前的客户端，从而将缓存的 QoS 消息重发，并应用之前的连接状态。

客户端使用会话保持的方式以 Java 为例：

```
public MQTTWriter(String address, String clientId, boolean cleanSession, int qos) throws MqttException {
    this.clientId = clientId;
    this.qos = qos;
    this.client = new MqttClient(address, clientId, persistence);
    MqttConnectOptions connOpts = new MqttConnectOptions();
    connOpts.setCleanSession(cleanSession); //置为 false 即为保留会话
    this.client.connect();
}
```

MQTT 5.0

基于这种网络连接频繁断开重连的情况，为了避免应用层频繁收到上下线事件，影响业务进行。

MQTT 5.0 也对协议进行了响应的优化：

Will Delay Interval（延时遗愿消息发布）：我们经常使用遗愿消息对客户端的下线进行追踪和告知。在这种情况下会频繁的收到遗愿消息。所以遗嘱时间间隔的一个重要用途就是避免在频繁的网络连接临时断开时发布遗嘱消息，因为客户端往往会很快重新连上网络并继续之前的会话。

Session Expiry Interval（会话过期间隔）：MQTT 3.1.1 未对会话保持时间做明确规定。如果使用 session 保持功能的客户端大量频繁上下线会造成 Broker 内存使用增加，最终影响服务高可用。所以 MQTT 5.0 也针对这种情况设计了会话过期时间。客户端可以在连接时使用这一特性设置自己的会话保持时间。

QoS 1/2

设置完会话保留状态，我们就可以使用 QoS 消息来保证消息的到达。

关于 QoS 的详解可参考文章：

<https://www.emqx.com/zh/blog/introduction-to-mqtt-qos>。

我们建议对于重要数据在车机端使用 QoS 1 进行发送，并且使用带有 QoS 重传功能和内置 QoS 消息窗口（队列）的 MQTT SDK。例如 [NanoSDK](#)，其具有异步确认、内置 QoS 消息队列、自动重发、高吞吐高消费能力等特点。

Broker QoS MsgQueue

QoS 消息在 Broker 端有内存持久化功能，除了客户端有内置的消息队列，Broker 也有一个 QoS 消息队列。如上文所述，车联网场景经常发生的基站切换导致连接重置，反映到 MQTT

连接就体现为 QoS 消息积压现象。客户端和服务端都会有未确认的消息积压在队列里。所以我们要根据实际情况设置消息队列的长度。

以 EMQX 为例，消息队列设置：

打开 emqx.conf

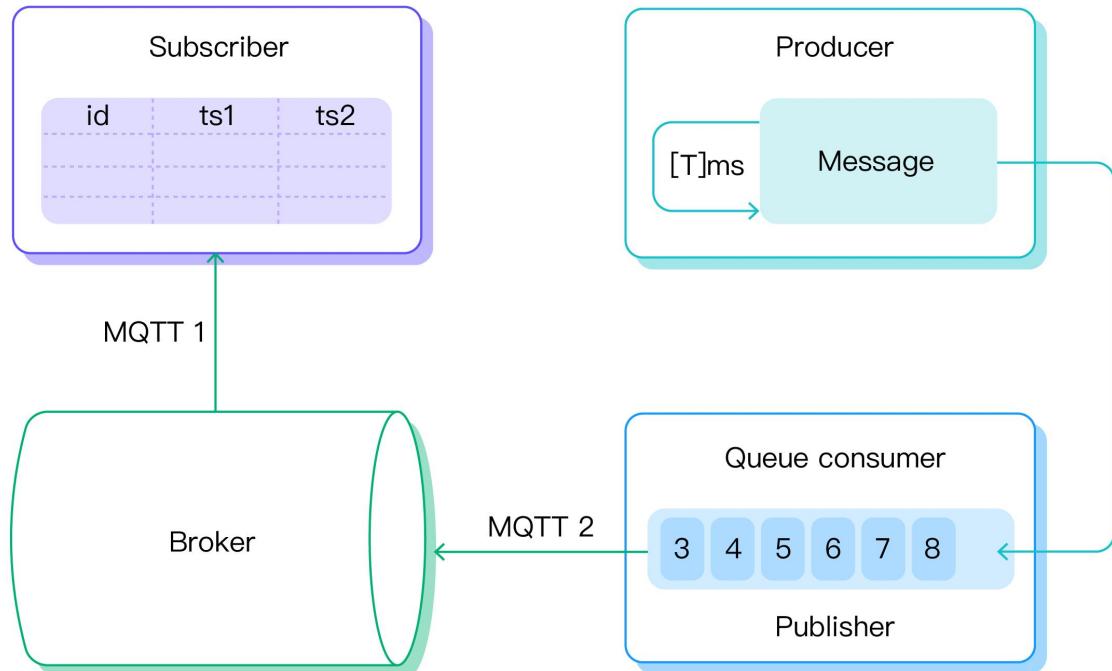
```
mqtt {  
    ## @doc Maximum QoS 2 packets (Client -> Broker) awaiting PUBREL.  
    max_awaiting_rel    = 100  
  
    ## @doc The QoS 2 messages (Client -> Broker) will be dropped if awaiting  
    ## PUBREL timeout.  
    await_rel_timeout    = 300s  
  
    ## @doc Maximum queue length. Enqueued messages when persistent client  
    ## disconnected, or inflight window is full.  
    max_mqueue_len      = 1000  
}
```

max_awaiting_rel 为接受 QoS 2 的消息队列长度。QoS 1 此项无限制。

await_rel_timeout 为 QoS 2 消息超时时间。

max_mqueue_len 为下发 QoS 1/2 的队列缓存长度

默认的 QoS 2 消息队列长度仅为 100，此处建议根据给客户端发布消息的频率和消费能力适当增加，一般考虑为 publisher 平均每秒产生消息的数量 *2 。此队列提供消费端一定的缓冲时间来完成重连后积压消息的消费。



小结

通过客户端和服务端对会话保持、QoS、客户端 ID 的配置和内置消息队列缓存等 MQTT 协议特性，我们可以在一定程度上解决高速移动带来的连接不稳定导致的数据丢失问题。

10 | MQTT over QUIC：下一代车联网消息传输标准协议

2022 年 6 月 11 日，IETF 正式颁布了 HTTP/3 RFC 技术标准文档，基于 UDP 的 QUIC 正式成为了传输层标准之一。自 5.0 版本起，EMQX 成为全球首个支持 MQTT over QUIC 的物联网 MQTT 消息服务器。针对前文我们提到的车联网移动场景下面临的消息传输挑战，MQTT over QUIC 是一种更优的解决方案。本章节将详细介绍为什么 MQTT over QUIC 是下一代车联网消息传输标准协议，以及它将如何赋能车联网场景。

什么是 QUIC 协议

QUIC（Quick UDP Internet Connections）是由谷歌公司开发的一种基于用户数据报协议（UDP）的传输层协议，旨在提高网络连接的速度和可靠性，以取代当前互联网基础设施中广泛使用的传输控制协议（TCP）。

QUIC 通过加密和多路复用技术来提供更高的安全性和更快的数据传输。它支持在单个连接上并行发送多个数据流，从而降低延迟并提高吞吐量。QUIC 还具有拥塞控制和流量控制等机制，以应对网络拥塞并保证数据传输的稳定性。

国际互联网工程任务组（IETF）已完成对 QUIC 的标准化，并且主流的 Web 浏览器和服务端正在逐步采用它。与 TCP 相比，QUIC 在高延迟和不稳定的网络环境中，如移动网络，可以显著提升网页加载速度并减少连接中断，使得网络体验更加流畅。

QUIC 协议的基本特性

相互独立的逻辑流

相互独立的逻辑流是 QUIC 的核心特性之一。它允许在单个连接上并行传输多个数据流，并

且每个流可以独立地处理。相比之下，TCP 只支持单数据流，需要按照发送顺序接收和确认每个报文。通过多路复用，应用程序可以更高效地发送和接收数据，并更好地利用网络带宽等资源。

一致安全性

QUIC 的另一个重要特性是它提供了端到端的安全保护。所有通过 QUIC 发送的数据都是默认加密的，并且不支持明文通信。这有助于防止数据被窃听和其他形式的攻击。QUIC 使用传输层安全协议（TLS）来建立和维护安全连接和端到端加密。

低延迟

QUIC 协议的设计目的是减少建立连接所需的延迟，以便在端点之间快速地发送和接收数据。对于移动网络这种高延迟的网络环境来说，这一点尤为重要。为了实现这个目标，QUIC 最小化了建立连接所需的往返次数，并且采用更小的报文来发送数据。传统的互联网协议通常存在延迟问题，例如美欧之间的往返时间有时可达 300 或 400 毫秒。

可靠性

QUIC 基于 UDP 但可提供可靠传输能力。类似于 TCP，它是一种面向连接的传输协议。QUIC 协议在数据传输过程中具有报文丢失恢复和重传功能，这可以确保数据的完整性和准确性。此外，QUIC 可以保证数据包按照发送顺序到达，避免因数据包乱序导致的数据错误。

消除 HOL 阻塞

QUIC 通过支持多个数据流来解决 HOL 阻塞问题。这使得来自不同应用的消息可以独立地传递，避免了因为等待其他应用而可能产生的延迟。

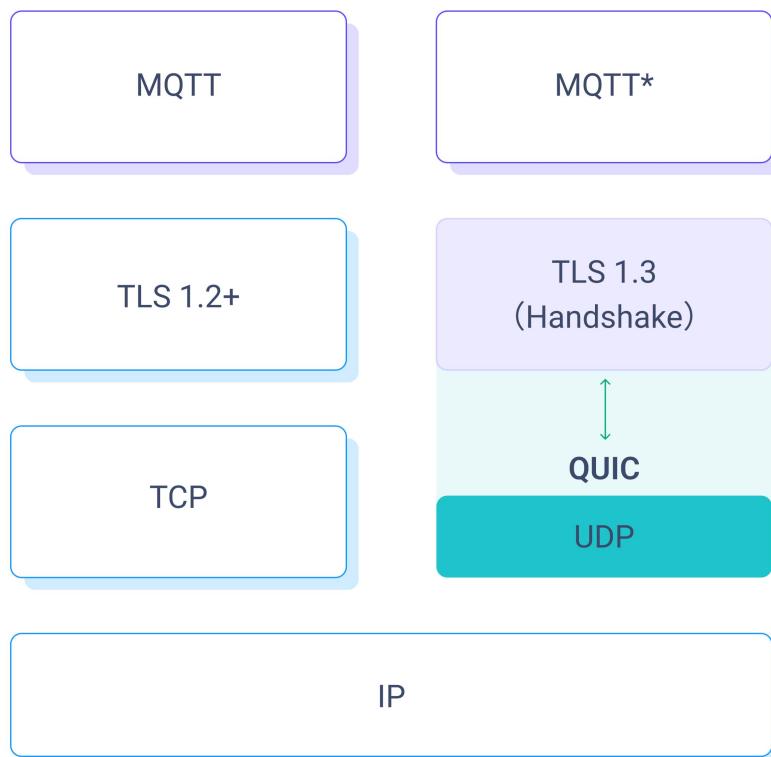
MQTT over QUIC 介绍

[MQTT](#) 是一种适用于低带宽、高延迟或不稳定网络环境的轻量级消息协议。它运行在应用层，主要用于机器对机器（M2M）通信和物联网场景。MQTT 采用发布/订阅模型，设备将消息发送到 Broker（即发布），其他设备根据主题接收这些消息（即订阅）。

对于 Web 应用而言，QUIC 专注于提高其性能和安全性，而 MQTT 则专为资源受限的网络环境提供轻量级和高效的消息传递解决方案。基于 QUIC 的 MQTT 可以显著提高性能并降低延迟，同时无需额外的 TLS 开销。由于大多数 QUIC 栈实现是在用户空间完成的，因此可以根据应用层的要求，自定义 QUIC 的数据传输，以适应不同的网络环境。

MQTT over QUIC 与 MQTT over TCP/TLS 对比

MQTT over TCP/TLS 指的是使用 TCP 作为传输层的 MQTT 协议。TCP 是一种可靠的、面向连接的协议，可确保数据包在设备之间的正确传递。TLS 是一种加密协议，通过加密两个端点之间传输的数据，为网络提供安全通信。通常情况下，TLS 作为 TCP 的上层协议使用，它使用 TCP 在两个端点之间建立和维护连接，并加密在该连接上传输的数据。



MQTT over QUIC 相比 MQTT over TCP/TLS 具有明显的优势：

连接建立：

- MQTT over TCP/TLS: MQTT over TCP/TLS 遵循 TLS1.2 规范，需要在 TCP 层和 TLS 层各进行一次握手。这意味着在应用层开始交换数据之前，需要进行两到三次往返通信。
- MQTT over QUIC: MQTT over QUIC 遵循 TLS1.3 规范，可以利用零或一次往返（0-RTT 或 1-RTT）握手快速建立连接，降低连接建立时的延迟。

延迟和性能：

- MQTT over TCP/TLS: 提供可靠的数据传输，但 TCP 的 HOL 阻塞问题和拥塞控制机制可能导致延迟增加和性能降低，尤其是在不可靠的网络环境下。
- MQTT over QUIC: 将 TCP 的可靠性与 UDP 的低延迟特性相结合。QUIC 的多路复用特性有助于最小化 HOL 阻塞问题，从而在有丢包或高延迟的网络环境下提高性能。

安全性：

- MQTT over TCP/TLS: 为了保证 MQTT 通信的安全，通常将其与 TLS 结合使用，TLS 提供了加密和认证功能。但是，这需要在连接建立和数据传输过程中增加额外的开销。
- MQTT over QUIC: QUIC 使用 TLS1.3 实现了内置加密，提供了安全的通信，无需额外的设置或开销。

客户端的连接迁移：

- MQTT over TCP/TLS: 如果 [MQTT 客户端](#)或服务器更换了 IP 地址或网络，那么现有的 TCP 连接就必须断开并重新建立，这会增加应用对异常处理的难度，容易出现各种因处理异常导致的 Bug。

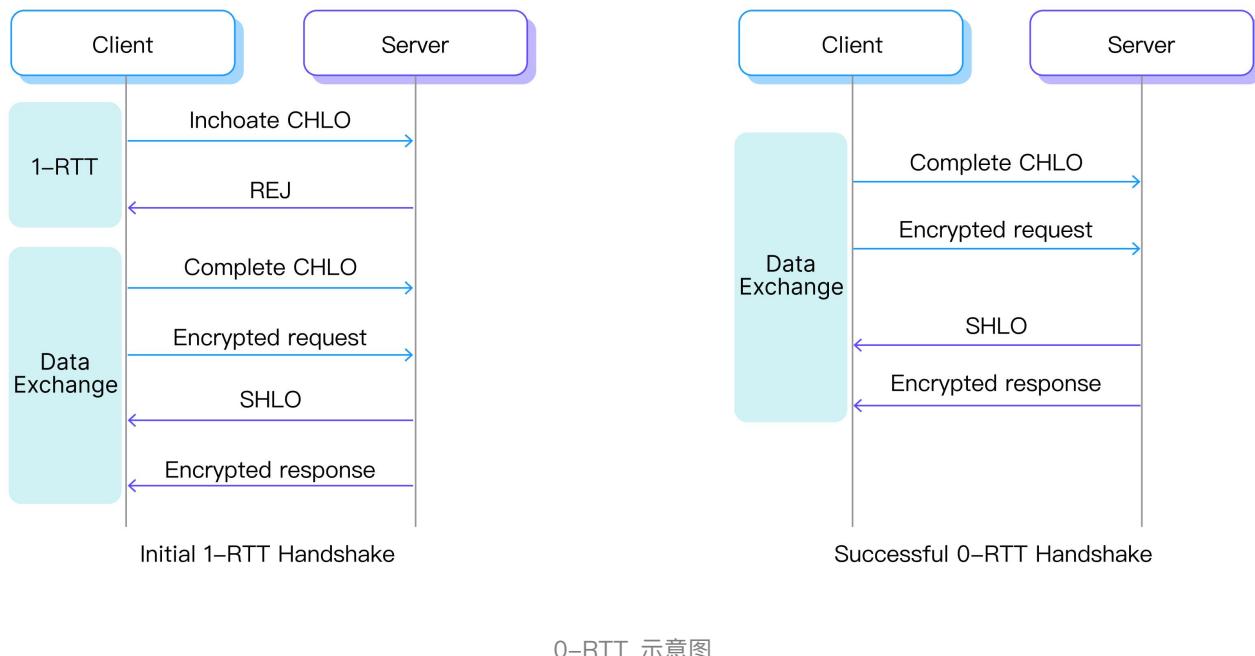
- MQTT over QUIC：支持连接平滑迁移，允许客户端或服务器在不影响正在进行的通信的情况下更换 IP 地址、端口或网络。

应用和支持：

- MQTT over TCP/TLS：已经得到了广泛的应用和支持，很多平台和编程语言都有 [MQTT broker](#)、客户端和库的实现。
- MQTT over QUIC：到目前为止，由于 QUIC 仍然是一种新兴的协议，因此 MQTT over QUIC 还没有得到广泛的应用和支持。

MQTT over QUIC 如何优化车联网移动通信

借助 QUIC 协议的地址迁移、流式多路复用、分路流控、更低的连接建立延迟，我们有望彻底解决车联网移动场景下的连接问题。



QUIC 能够侦测到地址改变，自动采用 0-RTT 的方式重建连接，从而使得客户端和服务端对于 IP 地址的变动无感知，这样就彻底避免了上文所说的一系列问题。

MQTT over QUIC 在车联网中的更多应用

在车联网场景下，MQTT over QUIC 可以带来很多优势，因为低延迟、可靠和安全的通信对各种应用来说都非常重要。由于 QUIC 结合了 TCP 和 UDP 的优点，并且提供了内置的加密，因此它可以显著提高基于 MQTT 的车联网应用的性能和安全性。

在车联网中使用 MQTT over QUIC 的场景包括：

- **车对基础设施（V2I）通信：**QUIC 的低延迟和可靠的数据传输可以提高车辆与基础设施组件（如交通信号灯、收费系统或智能停车系统等）之间的通信效率。
- **车联网（V2X）通信：**V2X 通信将车辆、基础设施和其他道路用户组合起来，旨在提高道路安全和交通效率。MQTT over QUIC 可以提供可靠的通信，并减少延迟，确保关键信息的及时交换。
- **车载资讯娱乐和远程诊断系统：**MQTT over QUIC 可以提高资讯娱乐系统的性能，实现更快的媒体流、导航更新和实时交通信息，同时确保通信安全。
- **车队管理和跟踪：**实时跟踪和管理车队需要车辆和管理系统之间的高效通信。MQTT over QUIC 可以提供可靠和安全的通信，实现车辆位置、诊断和驾驶行为的实时更新。
- **OTA 更新：**安全可靠的 OTA 更新对更新车辆固件和软件至关重要。MQTT over QUIC 可以提供必要的安全性和可靠性，无需中断车辆操作就可以传送这些更新。
- **应急响应：**在紧急情况下，可靠和快速的通信非常重要。MQTT over QUIC 可以确保及时安全地在应急车辆、响应团队和控制中心之间交换信息。

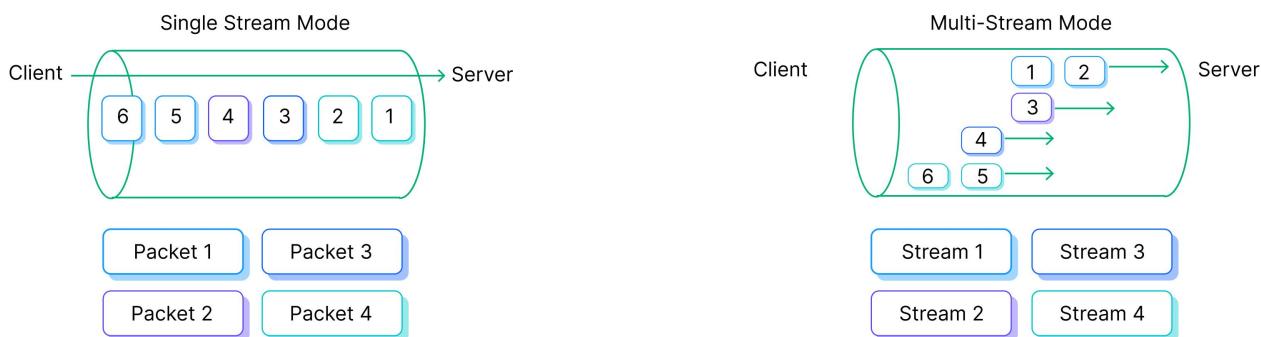
EMQX：首个实现 MQTT over QUIC 的 MQTT Broker

[EMQX](#) 是全球领先的开源 MQTT Broker，拥有高性能的实时消息处理引擎，为海量的物联网设备事件流处理提供动力。EMQX 从 5.0 版本开始支持 MQTT over QUIC，成为首个支持

MQTT over QUIC 的 MQTT Broker。不仅为现代复杂网络的 MQTT 消息传输提供了一种更高效安全的新方式，同时可以在某些场景下显著提高 MQTT 性能。

EMQX 支持将传输层替换为 QUIC 流，客户端可发起连接并创建双向流，从而实现更加高效可靠的通信。EMQX 支持两种操作模式：

- **单流模式**是一种基本模式，它将 MQTT 报文封装在一个双向的 QUIC 流中。该模式提供了快速握手、有序数据传输、连接恢复、0-RTT、客户端地址迁移以及增强的丢包检测和恢复等功能。这种模式使得客户端和 Broker 之间的通信更加快速和高效，同时保持有序，还能够快速恢复连接，并支持在不影响客户端通信的情况下迁移其本地地址。
- **多流模式**利用了 QUIC 的多路复用特性，允许 MQTT 报文在多个流中传输。这使得单个 [MQTT 连接](#)可以并行传输多个主题的数据且互不干扰。该模式还提供了多项优化，例如解耦连接控制和 MQTT 数据交换、避免 HOL 阻塞、分离上行和下行数据、优先处理不同类型的数据、提高并发性、增强鲁棒性、允许对数据流进行流量控制以及降低订阅延迟等。



使用 NanoSDK 客户端连接 MQTT over QUIC

[NanoSDK](#) 是基于 C 语言开发的第一个支持 MQTT over QUIC 的 SDK，完全兼容 EMQX 5.0。NanoSDK 的主要特点包括异步 I/O、将 MQTT 连接映射到 QUIC 流、低延迟的 0-RTT 握手以及多核并行处理等。

此外，EMQX 还为多种编程语言提供了客户端 SDK，以支持 MQTT over QUIC，包括：

- [NanoSDK-Python](#): NanoSDK 的 Python binding。
- [NanoSDK-Java](#): NanoSDK 的 Java JNA binding。
- [emqtt - Erlang MQTT 客户端](#): 使用 Erlang 开发的支持 QUIC 的 MQTT 客户端。

小结

目前，EMQX 企业版 5.1 版本中，MQTT over QUIC 已经具备投入生产能力，已在客户项目中进行了深度测试集成并获得了良好反馈。车联网用户可以使用 EMQX 5.1 获得更加高效、低延迟的物联网数据传输体验。随着 EMQ 在 MQTT over QUIC 标准化进程的积极推进，相信未来车联网及其他更多物联网行业的弱网与不固定网络通路场景下的消息传输问题都将得到进一步改善。

11 | 云原生赋能智能网联汽车消息处理基础框架构建

近年来，汽车产业向「电气化、智能化、网联化、共享化」快速演进，「软件定义汽车」模式和 SOA 理念在汽车研发和设计领域逐渐深入。无论是作为智能网联汽车云端底座的 TSP 平台、基于单车智能 ADAS 的自动驾驶体系，还是实现软件定义汽车的 SOA 框架，均需要更加灵活的软件开发、迭代、复用和运行架构保障。

云原生技术的快速发展和落地，大大改变了车联网应用传统的开发和运行方式。以其灵活、弹性、敏捷、自动化、高可用、易扩展等特性，为汽车行业智能网联和自动驾驶相关软件的开发和运行，提供了平台层面的助力，解决了车联网平台在新趋势下面临的上述挑战。

***云原生：**在 CNCF（云原生计算基金会）的定义中，云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。

本章节旨在深入分析云原生技术如何作用于车联网物联网基础设施构建，基于体系中最关键的车端消息采集、移动、处理和分析领域，结合 EMQ 相关数据基础设施软件，实现云原生的车联网基础设施架构。

传统车联网平台构建的挑战

传统车联网的消息处理框架在构建底层资源和运行平台端的整体框架时，往往采用本地数据中心虚拟机/物理机或云服务商虚拟机进行部署。此种模式在联网车辆快速增多、车端上传数据愈加复杂的场景下，通常会面临如下的痛点和挑战：

1. 各大主机厂若想将在私有数据中心提供车联网服务的模式转变为通过公有云提供车联网服务，传统的以虚拟机为应用承载方式的架构会过于沉重，无法平滑实现车联网混合云迁移

- 需求和混合云的部署模式；
2. 随着智能化和网联化的快速发展，车联网系统对于消息处理平台的软件迭代能力要求逐渐提高，传统的软件迭代模式无法应对智能网联对于车联网系统敏捷、灵活和快速的能力要求，针对纷繁复杂的消息处理需求和软件迭代也无法响应；
 3. 车联网系统作为主机厂同终端客户最重要的实时沟通系统，需要具备极高的可靠性、可用性和可支撑性。消息处理平台作为核心应用组件，应具备弹性的资源获取能力和自动化伸缩、运维等运营支撑能力。传统的巨石型应用架构和虚拟机部署模式无法满足消息处理平台弹性和自复位的能力要求。

云原生技术赋能新一代车联网消息处理

CNCF (Cloud Native Computing Foundation) 旗下项目中以容器编排系统 Kubernetes 最为核心和基础。Kubernetes 通过将应用程序的容器组合成逻辑单元，以便于管理与服务发现，其为开源系统，可以自由地部署在企业内部、私有云、混合云或公有云，方便用户做出自由选择。



越来越多的主机厂在业务平台的生产交付场景中，采用云原生技术打造以下能力，助力智能网联汽车的应用演进和发展。

- 统一部署：提供屏蔽底层资源型基础设施差异，一次构建，多处使用的能力
- 易于实施：提供 CaaI(Config as an Infrastructure)，即配置即设施的能力，达到配置即运行时效果的能力

- 弹性扩容：根据业务使用情况进行资源型资源的快速弹性伸缩，提供运行时伸缩，业务应用无感知的能力
- 监控告警：提供完善的监控告警体系，满足生产环境后期维护的可控性能力
- 版本迭代可控：提供风险可控的版本变更手段，包括版本追溯与回滚的能力

基于 Operator 的 EMQX 云原生框架

早期 EMQ 产品云原生部署采用的是 Helm 部署方式，Operator 模式的出现为实现自定义资源提供了标准的解决方案，解决了通用 Kubernetes 基础模型元素无法支撑不同业务领域下复杂自动化场景的痛点，为实现更加简单、高效的 EMQX 部署提供了全新的方式。

简单来说，Operator 模式是一组自定义控制器的集合以及由这些控制器所管理的一系列自定义资源，我们将不再关注于 Pod（容器）、ConfigMap 等基础模型元素，而是将它们聚合为一个应用或者服务。

Operator 通过控制器的协调循环来使自定义应用达到我们期望的状态，我们只需要关注该应用的期望状态，通过自定义控制器协调循环逻辑来达到 7*24 小时不间断的应用或者服务的声明周期管理。基于 Operator 的 EMQX 云原生框架，使得用户可以轻松基于 Kubernetes 的模式部署和运维 EMQX 集群。

Operator VS Helm

Operator 的管理不仅限于 Pod，也可以是多个资源（比如 SVC 域名等）。从这个角度上来说，Operator 跟 Helm 一样，也是具有编排能力的。从编排角度来看，Helm 与 Operator 有非常多的共性，很难对两者的作用进行区分。Helm 也可以完成分布式系统的部署。

那么 Operator 跟 Helm 又有什么样的区别呢？

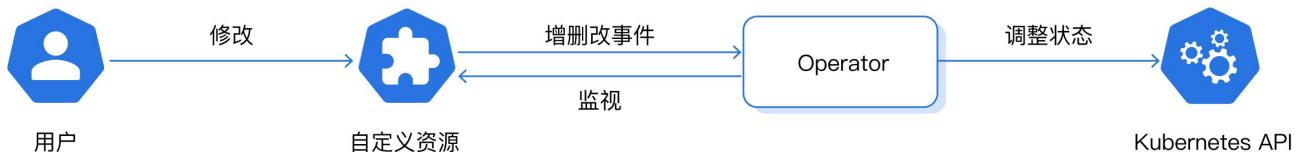
- Helm 的侧重点在于多种多个的资源管理，而对生命周期的管理主要包括创建更新和删除，

Helm 通过命令驱动整个的生命周期。

- Operator 对于资源的管理则不仅是创建和交付。由于其可以通过 watch 的方式获取相关资源的变化事件，因此可以实现高可用、可扩展、故障恢复等运维操作。因此 Operator 对于生命周期的管理包括创建、故障恢复、高可用、升级、扩容缩容、异常处理以及最终的清理等等。
- 如果把 Helm 比作 RPM，那么 Operator 就是 systemd。RPM 负责应用的安装、删除，而 systemd 则负责应用的启动、重启等等操作。

Operator 工作原理

Operator 使用自定义资源（CR）管理应用及其组件的自定义 Kubernetes 控制器，自定义资源 Kubernetes 中 API 扩展，自定义资源配置 CRD 会明确 CR 并列出 Operator 用户可用的所有配置，Operator 监视 CR 类型并且采取特定于应用的操作，确保当前状态与该资源的理想状态相符。



Operator 中主要有以下几种对象：

CRD: 自定义资源的定义，Kubernetes API 服务器会为你所指定的每一个 CRD 版本生成 RESTful 的资源路径。一个 CRD 其实就是定义自己应用业务模型的地方，可以根据业务的需求，完全定制自己所需要的资源对象，如 EMQX Broker、EMQX Enterprise 等这些都是可以被 Kubernetes 直接操作和管理的自定义的资源对象。

CR: 自定义资源，即 CRD 的一个具体实例，是具体的可管理的 Kubernetes 资源对象，可以对其进行正常的生命周期管理，如创建、删除、修改、查询等，同时 CR 对象一般还会包含运行时的状态，如当前的 CR 的真实的状态是什么，用于观察和判断，CR 对象的真正所处

于的状态。

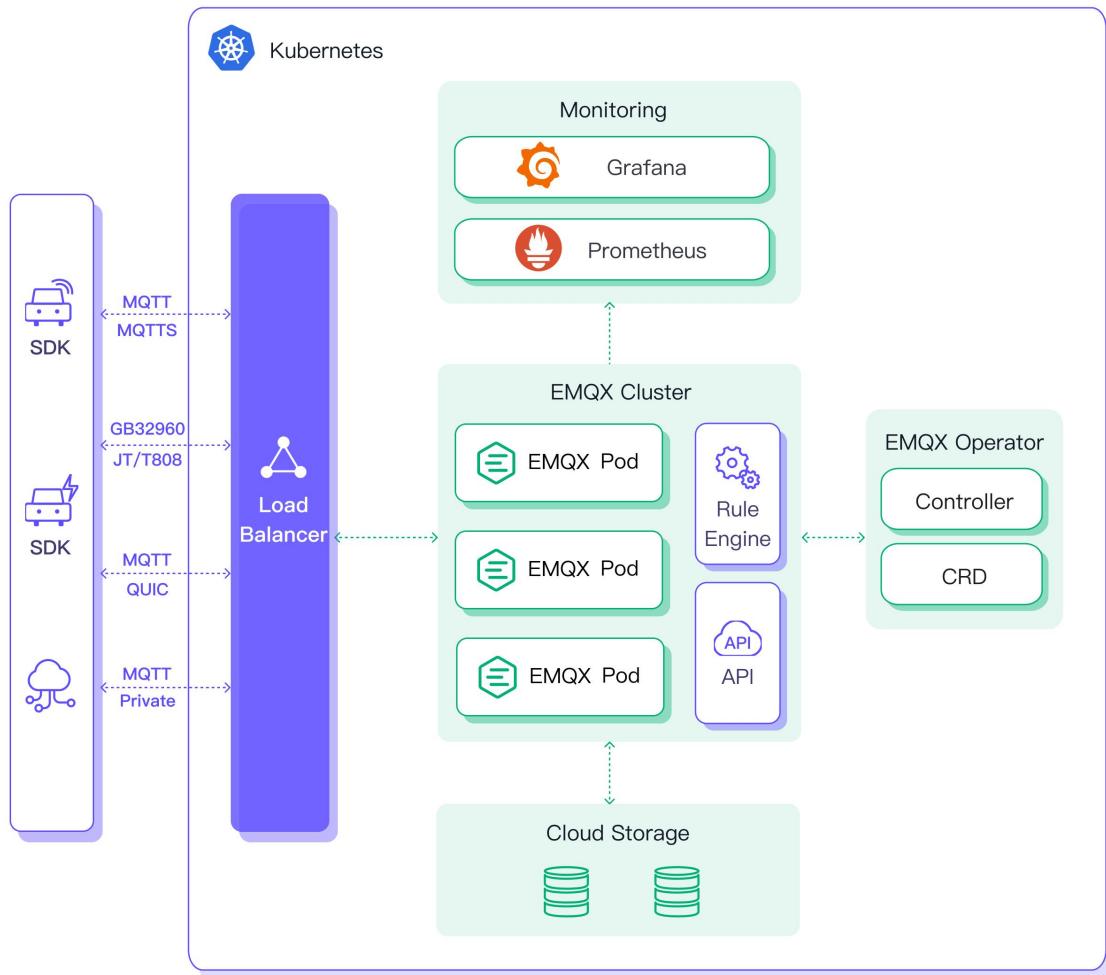
Controller: 其实就是控制器真正的用武之地了，它会循环处理工作队列中的动作，按照逻辑协调应用当前状态至期望状态。如观察一个 CR 对象被创建了之后，会根据实现的逻辑来处理 CR，让 CR 对象的状态以及 CR 对象所负责的业务逻辑慢慢向期望的状态上靠近，最终达到期望的效果。

举例来说：如果定义了一个 EMQX Broker 的 Operator，那在创建 EMQX Broker 的时候，就会一直协调和观察 EMQX Broker 真正的集群是否已创建好，以及每个节点的状态和可用性是否健康。一旦发现不符合期望的状态就会继续协调，始终保持基于事件的机制不断检查和协调，以保证期望的状态。

EMQX 在车联网场景中的云原生实践

基于 Operator 模式，我们提供了 EMQX Operator 来帮助客户在 Kubernetes 的环境上快速构建和管理 EMQX 集群。

EMQX Operator 是一个用来帮助用户在 Kubernetes 的环境上快速创建和管理 EMQX 集群的工具。它可以大大简化部署和管理 EMQX 集群的流程，将其变成一种低成本的、标准化的、可重复性的能力。



作为车联网的核心底层支撑组件，EMQX 可以通过 Kubernetes Operator 进行部署、管理和运维。通过基于云原生的消息处理平台，为车联网场景中的客户开发和运维部署带来了诸多好处：

- 无感和滚动更新：以云原生技术构建的车联网消息处理框架，可以轻松实现车联网应用的灰度发布，使得车联网系统升级迭代过程中无需中断服务，让车联网应用的使用者完全无感知，实现无感迭代和滚动更新，提升用户体验；
- 统一监控：基于云原生技术，车联网系统的运维者可轻松进行应用集群和节点的监控和管理。通过与 Prometheus 等监控工具的集成，可以轻松获取车联网中最重要的消息处理平台的运行信息，从而更直观清晰地了解业务运行情况，进行相应的业务迭代；
- 快速部署：云原生技术可以让车联网开发和运维人员快速部署和调整应用，无论是基于公有云还是私有云环境，均可以轻松部署相应的 EMQX 集群，实现对业务的支撑；

- 标准化镜像：EMQX 基于云原生环境提供了标准化的官方容器镜像，当用户系统通过镜像来进行 EMQX 的部署和构建时，可基于标准化镜像进行相应的工作，对于车联网环境中的快速发布和多次构建等需求提供了很好的支撑；
- 弹性伸缩：随着车联网应用的深入，整个消息处理框架所需要对接的应用逐步扩展和车联网规模的增大，消息处理平台对资源的弹性能力要求也越来越高，通过 Kubernetes 弹性灵活的资源支撑模式，可以针对应用的使用量进行资源获取、增加和释放，从而节省资源，降低运营成本；
- 快速迭代：基于云原生框架构建的车联网应用，可利用持续集成和持续交付流水线实现应用的即时更新和发布，支撑业务对于车联网快速迭代的需求。

小结

随着云原生理念在各行业的深入，我们相信云原生也将为车联网领域的平台构建与应用开发模式注入新的动力。未来 EMQ 将围绕对 EMQX 新版本特性的支持，不断完善迭代 EMQX Operator，致力于在云原生模式下提供更加丰富可靠的数据基础设施能力，服务车联网行业。

结语

随着智能网联汽车、智慧交通系统和物联网的普及，车联网将继续在我们的生活中扮演越来越重要的角色。

通过本白皮书，我们深入研究了 MQTT 协议、消息平台架构设计、通信安全、数据采集、消息传输优化等关键领域，还介绍了下一代车联网消息传输标准协议 MQTT over QUIC，展望未来车联网通信的发展方向。

我们希望为车联网领域的从业者所面临的设计、架构和安全等方面挑战提供全面的解决方案，帮助大家更好地理解、设计和实现车联网平台，推动智能交通系统的发展，为未来的出行方式带来更多创新和便利。

车联网的未来充满了无限可能性，我们期待看到各种各样的创新和应用，让我们的道路更安全、更高效、更智能。希望这本白皮书能够成为车联网领域的有益资源，为未来的发展提供坚实的基础和方向。



EMQ（杭州映云科技有限公司）是全球领先的物联网数据基础设施软件供应商，交付全球领先的开源、云原生 MQTT 消息服务器，为企业云边端的海量物联网数据提供高可靠、高性能的「实时连接、移动、处理与集成」。公司成立于 2017 年，旗舰产品 EMQX 拥有来自 50 多个国家的 500 多家企业用户，连接全球超过 2.5 亿台物联网设备。

访问官网：[连接物理世界与人工智能](#)

免费试用 EMQX Enterprise：<https://www.emqx.com/zh/try?product=enterprise>



获取更多技术干货



联系车联网解决方案专家