

资深物联网专家10余年经验总结，让从业者少走弯路、少踩坑、少重复造轮子

详解MQTT协议、从0到1搭建物联网平台的方法  
总结物联网平台开发的设计模式和最佳实践

付强◎著

# 物联网 系统开发

## 从0到1构建IoT平台



机械工业出版社  
China Machine Press

# 物联网系统开发：从0到1构建IoT平台

付强 著

ISBN: 978-7-111-66240-2

本书纸版由机械工业出版社于2020年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

# 前言

## 为什么要写这本书

2011年我在硅谷的时候，曾经参与设计和开发了一个物联网平台。这个平台的目的是为各种物联网设备提供统一的通信接口，以及提供数据存储和分析功能，降低物联网设备商的开发和运营成本。不过由于物联网设备的异构性太强，同时平台的愿景过于超前，而当时物联网应用的发展包括资本的投入都远不及现在，这个项目不得不半途中止。

2015年，我在国内和朋友联合创办了一家物联网相关的公司。为了支撑公司的硬件产品，我们开发了一个提供统一通信和数据服务的物联网平台，不过吸取了之前的教训，这个平台只限于对同一组织（公司）里的多个产品提供支持。当时各大云服务商，比如阿里云，也提供了非常成熟的物联网套件，我们将这些物联网套件中的一些功能移植到了自研的物联网平台上。这个平台从技术层面很好地支持了公司从0到1、从1到N持续盈利的全流程。

在这个过程中我遇到过一些问题，也总结了一些非常有用的经验。在此期间，我也加入了一些物联网开发者的社区。在日常的技术

交流里，我发现一些开发人员对常用的物联网协议的理解是有问题的，对一些功能应该在协议层面解决还是在业务层面解决不是很清楚。我曾在互联网上搜索过相关的技术文章，发现系统性地讲解协议的规范和特性非常少，不是只对一个两个功能进行介绍，就是翻译协议规范，缺乏代码示例。

在这种情况下，我在GitChat码字专栏写了我的第一篇文章《MQTT协议快速入门》，详细地对物联网应用中最常见的MQTT协议的规范和特性进行了讲解，并对每一个特性附以丰富的代码示例。

加入专栏文章的读者交流群后，我又发现读者们还有很多关于设计、业务架构上的疑问，深入理解MQTT协议并不能解决这些问题。这让我意识到，在物联网行业，并不像Web开发那样有成熟的设计模式和框架可以使用，开发者往往都是从协议级别开始往上搭，重复地造轮子。

这时，我觉得有必要把我们在开发物联网平台中遇到的困难和总结的经验分享出来，从协议开始讲起，再覆盖物联网后台开发中常见的设计模式和最佳实践，让其他的物联网开发者少走一些弯路，少造一些轮子，进而更快速、高效地上线自己的产品。

## 读者对象

- 物联网应用开发者
- 物联网架构师
- 物联网平台开发者
- 对物联网感兴趣的开发人员
- 有一定经验的IM平台、移动推送平台开发人员
- 渴望学习更多物联网实际开发经验的人员

## 如何阅读本书

本书涵盖物联网应用开发80%的场景，理论和实战并重。本书内容分为三大部分。

第一部分（第1～2章）为物联网基础知识介绍，涵盖物联网的概念和常用协议。

第二部分（第3～5章）为MQTT协议详解，通过详尽的示例代码对MQTT协议的规范和特性进行讲解。

第三部分（第6～12章）为物联网平台开发实战，从0开始用开源的组件搭建一个名为“Maque IoT Hub”的物联网平台，在这个过程中

讲解物联网后台开发中常见的设计模式和最佳实践。

在最后补充有结语与附录。结语总结了本书讲到的相关系统与知识体系，附录介绍了运行Maque IoTHub的方法和步骤。

如果你对MQTT协议已经非常了解，可以直接从第三部分开始看起，第二部分可用作协议规范参考。

如果你是一名初学者，请务必从第1章的基础知识开始学习。

## 勘误和支持

除封面署名外，参加本书编写工作的还有：赵华振、李斌锋、邓斌、戚祥、于伟、皮文星、陈育春、陆正武、虞晓东、张恒汝、高喆、刘威、刘冉、付志涛、宗杰、王大平、李振捷、李波、张鹏、管西京、闫芳、王玉芹、王秀明、杨振珂。由于作者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。

书中有大量的实例代码，都可以从我的GitHub站点（<https://github.com/sufish>）下载。你也可以关注我在GitChat的专栏

（<https://gitbook.cn/gitchat/author/59ed8409991df70ecd5a0f8f>

），并加入专栏读者群交流。如果你有更多的宝贵意见，也欢迎发邮件到yfc@hzbook.com。期待能够得到你们的真挚反馈。

## 致谢

首先要感谢EMQ X的开发者和贡献者们，开发和维护一款强大的开源MQTT Broker非常不易。

感谢GitChat提供的平台，你们的引荐和帮助促成了本书的出版。

感谢机械工业出版社华章公司的杨福川老师和各位编辑，在这么长的时间中始终支持我写作，你们的鼓励和引导帮助我顺利完成全部书稿。

感谢我公司的全体同仁，是你们的共同努力才给我提供一个能够实践自己想法的机会和平台。

最后感谢关心我的家人，尤其是我的妻子和女儿，你们的支持是我完成本书的原动力！

谨以此书献给我最亲爱的家人，以及广大的物联网开发者！

付强

# 第一部分 物联网基础

- 第1章 什么是物联网
- 第2章 常见的物联网协议



## 第1章 什么是物联网

物联网（Internet of Things）这个概念读者应该不会陌生。物联网的概念最早于1999年被提出来，曾被称为继计算机、互联网之后，世界信息产业发展的第三次浪潮，到现在已经发展了20余年。如今，在日常生活中，我们已经可以接触到非常多的物联网产品，例如各种智能家电、智能门锁等，这些都是物联网技术比较成熟的应用。

物联网最早的定义是：把所有物品通过射频识别等信息传感设备与互联网连接起来，实现智能化识别和管理。当然，物联网发展到今天，它的定义和范围已经有了扩展与变化，下面是现代物联网具有的两个特点。

### 1. 物联网也是互联网

物联网，即物的互联网，属于互联网的一部分。物联网将互联网的基础设施作为信息传递的载体，即现代的物联网产品一定是“物”通过某种方式接入了互联网，而“物”通过互联网上传/下载数据，以及与人进行交互。举个通过手机App远程启动汽车的例子，当用户通过App完成启动操作时，指令从已接入互联网的手机发送到云端平台，云端平台找到已接入互联网的车端电脑，然后下发指令，车端电脑执行启动命令，并将执行的结果反馈到云端平台；同时，用户的这次操作

被记录在云端，用户可以随时从App上查询远程开锁记录历史。这就是一个典型的物联网场景，它是属于互联网应用的一种。“物”接入互联网，数据和信息通过互联网交互，同时数据和其他互联网应用一样汇聚到了云端。

再举一个例子，一个具有红外模块的手机，可以通过发送红外信号来开关客厅的电视机，这种应用在功能机时代十分常见，那么这个场景属于物联网应用吗？看起来很像，同样是用手机操纵一个物体，不过此时你的电视并没有接入互联网，你的手机可能也没有，手机和电视的交互数据没有汇聚到云端，所以这个场景不属于现代物联网场景。

## 2. 物联网的主体是“物”

前面说现代物联网应用是一种互联网应用，但是物联网应用和传统互联网应用又有一个很大的不同，那就是传统互联网生产和消费数据的主体是人，而现代物联网生产和消费数据的主体是物。

我们可以回想一下自己上网娱乐的日常：刷微博、写微博的是人，看微博的也是人；看短视频是人，拍短视频也是人；上淘宝买东西，下单的是人，收到订单进行发货的也是人；上在线教育网站学习，写课程的是人，学习课程的也是人。在传统互联网的应用场景中，生产的数据是和人息息相关的，人生产数据，也消费数据，互联

网平台在采集这些数据之后，将分析和汇总的结果也应用到人这个主体上，比如通过你的偏好推送新闻、商品等。

不过在现代物联网的应用场景下，情况就有所不同了。首先数据的生产方是“物”，比如智能设备或者传感器，数据的消费者往往也是“物”，这里举个例子。

在智慧农业的应用中，孵化室中的温度传感器将孵化室中的温度周期性地上传到控制中心。当温度低于一定阈值时，中心按照预设的规则远程打开加温设备。在这一场景中，数据的生产者是温度传感器，数据的消费者是加温设备，二者都是“物”，人并没有直接参与其中。


当然，在很多现代物联网的应用场景中，人作为个体，也会参与数据的消费和生产，比如在上面的例子中，打开加温设备的规则是人设置的，相当于生产了一部分数据。同时，在打开加温设备时，设备可能会通知管理人员，相当于消费了一部分数据。但是在大多数场景下，人生产和消费数据的频次和黏度是非常低的。例如，我可能会花3个小时来写一篇博客，但我只会花几分钟来设置温度的阈值规则；我可能会刷一下午的抖音，但不会花整个下午的时间一条条地看孵化室的温度记录，我只要在特定事件发生的时候能够收到一个通知就可以了。在这些场景下，数据的主体仍然是“物”。

这就是物联网和传统互联网最大的不同：数据的生产者和消费者主要是物，数据内容也是和“物”息息相关的。

## 1.1 物联网和人工智能

既然说到了物联网，那么这里有必要再提一下人工智能。

人工智能可谓近年来IT领域最火的词语之一。人工智能的概念是在1956年提出的，之前一直不温不火，直到最近几年才得以飞速发展，尤其是以神经网络为代表的深度学习，发展更为迅速。

 **提示**神经网络是深度学习中的一种非常重要的技术，它用类似于大脑神经元的架构来组织学习网络，在分类、计算机视觉方面的应用场景非常多。它的特点之一就是需要大量的数据进行训练。

纵观人工智能的发展路线，我们可以看到，人工智能的发展之所以能够突飞猛进，主要有以下两个原因。

- 硬件的发展使得深度学习神经网络的学习时间迅速缩短。
- 在大数据的时代，获取大量数据的成本变低。

事实上，第二个原因尤为重要，神经网络由于其特性，需要海量的数据进行学习，可供学习的有效数据量往往决定了最后训练出的神经网络的效果，甚至算法的重要性都可以排在数据量之后。

而物联网设备，比如智能家电、可穿戴设备等，每天都在产生海量的数据，这些数据经过处理和清洗后，都可以作为不错的训练数据反哺神经网络。同时，训练出来的神经网络又可以重新应用到物联网设备中，进而形成一个良性循环。这里举个例子，通过交通探头，我们可以采集大量的实时交通图片。经过处理，我们把图片“喂给”神经网络，比如SSD（Single Shot MultiBox Detector）。SSD先学会在图片中标注出人和汽车的位置，然后把模型部署到探头端，探头就可以利用深度学习的结果，实时分析人流和车流情况。



**提示** SSD是在物体识别中常用的一种神经网络。

图1-1所示为物联网应用人工智能方法进行数据采集-迭代的循环。通过物联网设备采集并训练数据，在数据中心完成训练后，将模型应用到物联网设备，并评估效果进行下一次迭代。

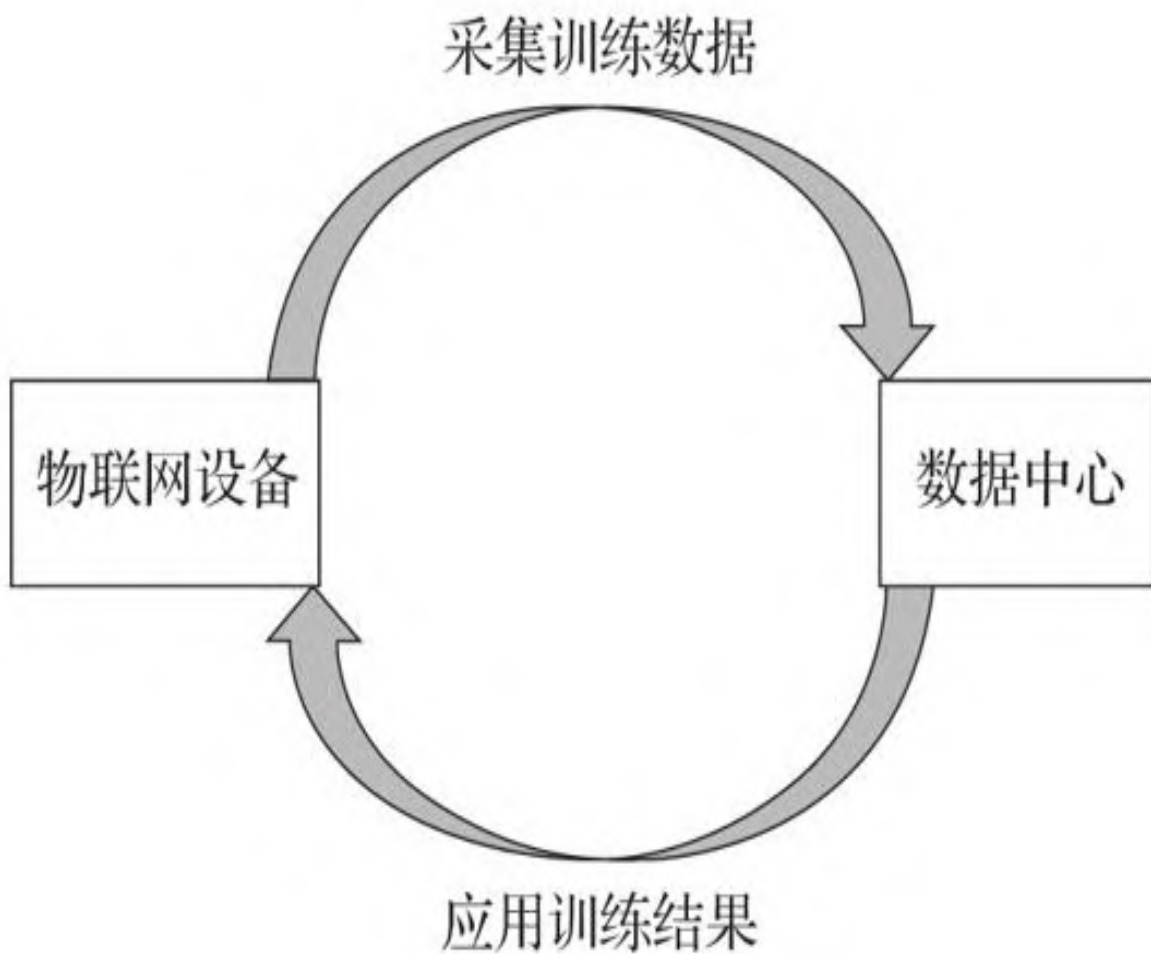


图1-1 采集-迭代的循环

物联网是人工智能落地的一个非常好的应用场景。随着人工智能的迅速发展，物联网这个同样在很多年前就提出的理论和技术，也会迎来新的春天。

目前，互联网数据入口渐渐朝几大巨头（例如阿里、腾讯）汇聚，规模较小的公司获取数据的代价越来越高，物联网这块还未完全开发的数据领域就显得尤为重要。

这也是本书侧重于物联网平台开发而略过前端设备开发的原因，因为前端设备最终会趋于相同，出现同质化竞争，而如何采集和使用好设备产生的海量数据，才是你是否具有竞争优势的决定性因素。



## 1.2 物联网的现状与前景

随着5G时代的来临，物联网的发展将会非常迅速。同时，物联网方向的新增融资也一直处于上升趋势。下面再从应用场景角度来谈一下物联网行业的发展前景。

物联网的应用场景非常广泛，包括：

- 智慧城市
- 智慧建筑
- 车联网
- 智慧社区
- 智能家居
- 智慧医疗
- 工业物联网

.....

在不同的场景下，物联网应用的差异非常大，终端和网络架构的异构性强，这意味着在物联网行业存在足够多的细分市场，这就很难

出现一家在市场份额上具有统治力的公司，同时由于市场够大，所以能够让足够多的公司存活。这种情况在互联网行业是不常见的，互联网行业的头部效应非常明显，市场绝大部分份额往往被头部的两家公司占据。

物联网模式相对于互联网模式来说更“重”一些。物联网的应用总是伴随着前端设备，这也就意味着用户的切换成本相对较高，毕竟拆除设备、重新安装设备比动动手指重新下载一个应用要复杂不少。这也就意味着，资本的推动力在物联网行业中相对更弱。如果你取得了先发优势，那么后来者想光靠资本的力量赶上或者将你挤出市场，那他付出的代价要比在互联网行业中大得多。

所以说，物联网行业目前仍然是一片蓝海，小规模公司在这个行业中也完全有能力和大规模公司同台竞争。在AI和区块链的热度冷却后，物联网很有可能会成为下一个风口。作为程序员，在风口来临之前，提前进行一些知识储备是非常有必要的。

下面我们将从协议开始学习，一步步搭建起一个完善的物联网平台。

## 第2章 常见的物联网协议

本章将简单介绍一些常见的物联网协议，包括物理层协议、数据链路层协议和应用层协议。

## 2.1 MQTT协议

MQTT协议（Message Queue Telemetry Transport，消息队列遥测传输协议）是IBM的Andy Stanford-Clark和Arcom的Arlen Nipper于1999年为了一个通过卫星网络连接输油管道的项目开发的。为了满足低电量消耗和低网络带宽的需求，MQTT协议在设计之初就包含了以下几个特点。

- 实现简单；
- 提供数据传输的QoS；
- 轻量、占用带宽低；
- 可传输任意类型的数据；
- 可保持的会话（Session）。

此后，IBM一直将MQTT协议作为一个内部协议在其产品中使用。直到2010年，IBM公开发布了MQTT 3.1版本。2014年，MQTT协议正式成为OASIS（结构化信息标准促进组织）的标准协议。随着多年的发展，MQTT协议的重点不再只是嵌入式系统，而是更广泛的物联网世界。

简单来说，MQTT协议有以下特性。

- 基于TCP协议的应用层协议;
- 采用C/S架构;
- 使用订阅/发布模式, 将消息的发送方和接受方解耦;
- 提供3种消息的QoS (Quality of Service) : 至多一次、最少一次、只有一次;
- 收发消息都是异步的, 发送方不需要等待接收方应答。

MQTT协议的架构由Broker和连接到Broker的多个Client组成, 如图2-1所示。

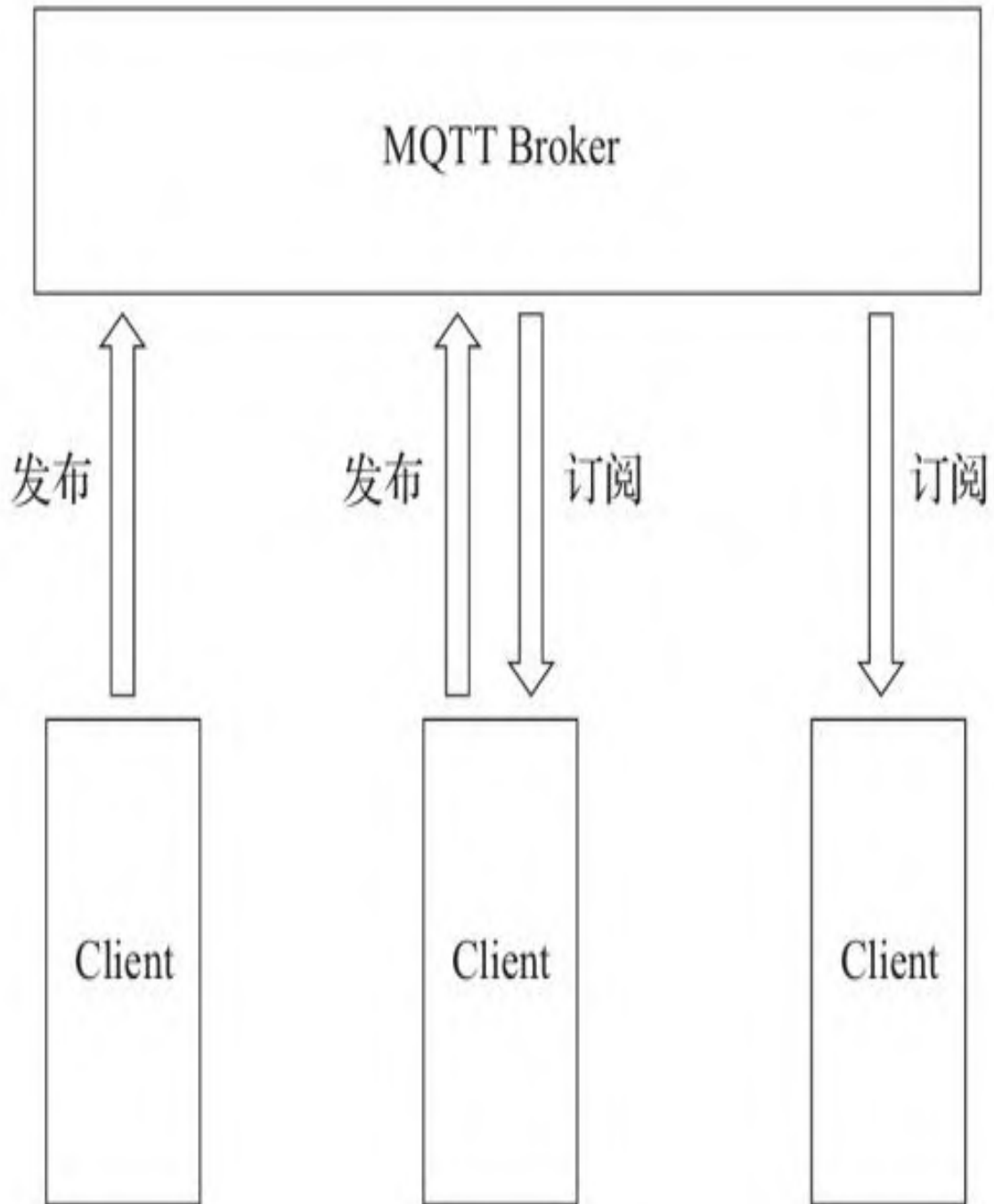


图2-1 MQTT协议的Broker和Client

MQTT协议可以为大量的低功率、工作网络环境不可靠的物联网设备提供通信保障。而它在移动互联网领域也大有作为，很多Android

App的推送功能都是基于MQTT协议实现的，一些IM的实现也是基于MQTT协议的。

MQTT协议可以说是目前运用最广的协议。下面的章节将对MQTT协议以及其特性进行详细的讲解。

## 2.2 MQTT-SN协议

MQTT-SN (MQTT for Sensor Network) 协议是MQTT协议的传感器版本。MQTT协议虽然是轻量的应用层协议，但是MQTT协议是运行于TCP协议栈之上的，TCP协议对于某些计算能力和电量非常有限的设备来说，比如传感器，就不太适用了。

MQTT-SN运行在UDP协议上，同时保留了MQTT协议的大部分信令和特性，如订阅和发布等。MQTT-SN协议引入了MQTT-SN网关这一角色，网关负责把MQTT-SN协议转换为MQTT协议，并和远端的MQTT Broker进行通信。MQTT-SN协议支持网关的自动发现。MQTT-SN协议的通信模型如图2-2所示。



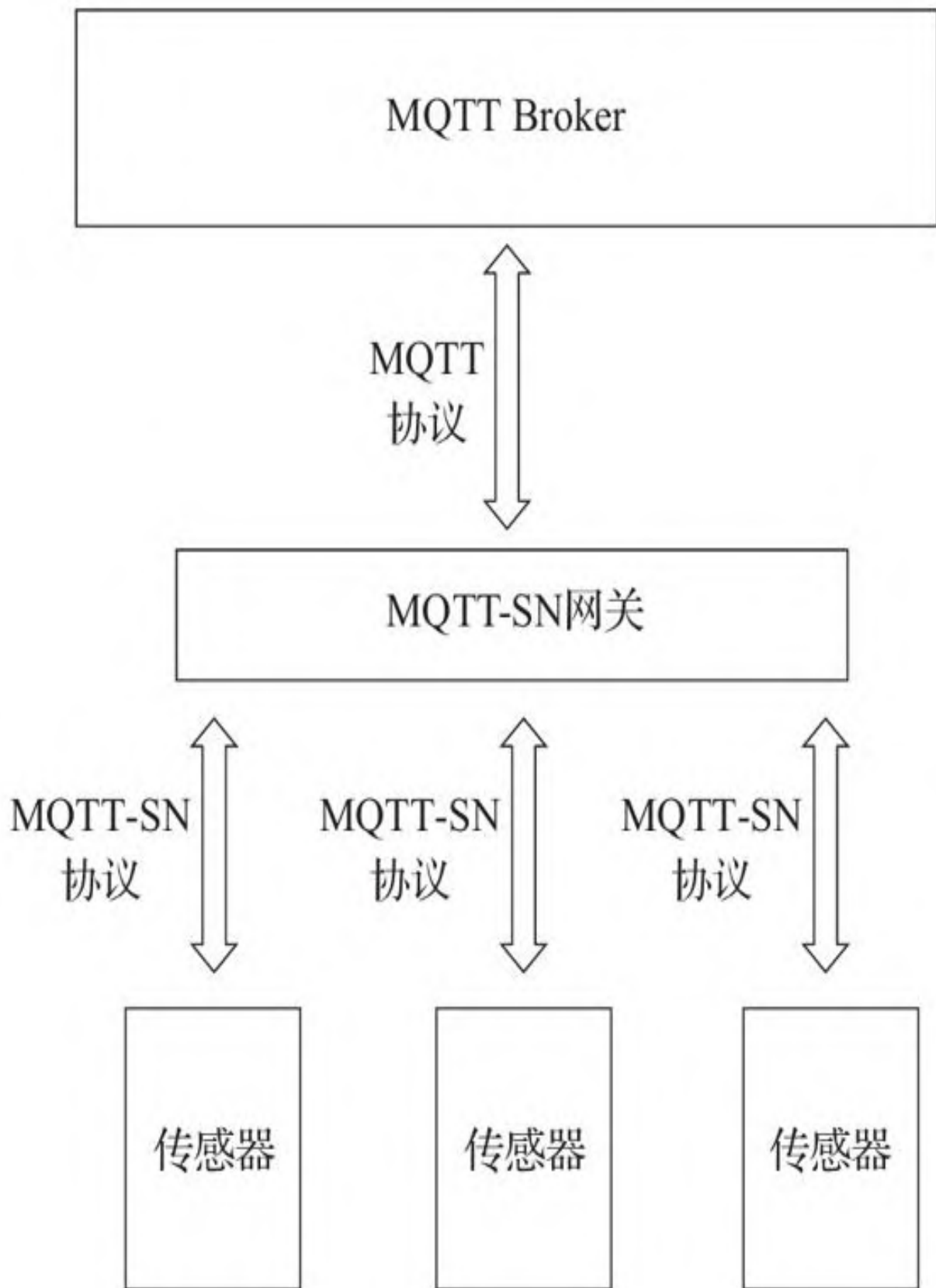


图2-2 MQTT-SN协议的通信模型

## 2.3 CoAP协议

CoAP (Constrained Application Protocol) 协议是一种运行在资源比较紧张的设备上的协议。和MQTT-SN协议一样，CoAP协议通常也是运行在UDP协议上的。

CoAP协议设计得非常小巧，最小的数据包只有4个字节。CoAP协议采用C/S架构，使用类似于HTTP协议的请求-响应的交互模式。设备可以通过类似于coap://192.168.1.150:5683/2ndfloor/temperature的URL来标识一个实体，并使用类似于HTTP的PUT、GET、POST、DELETE请求指令来获取或者修改这个实体的状态。

同时，CoAP提供一种观察模式，观察者可以通过OBSERVE指令向CoAP服务器指明观察的实体对象。当实体对象的状态发生变化时，观察者就可以收到实体对象的最新状态，类似于MQTT协议中的订阅功能。CoAP协议的通信模型如图2-3所示。

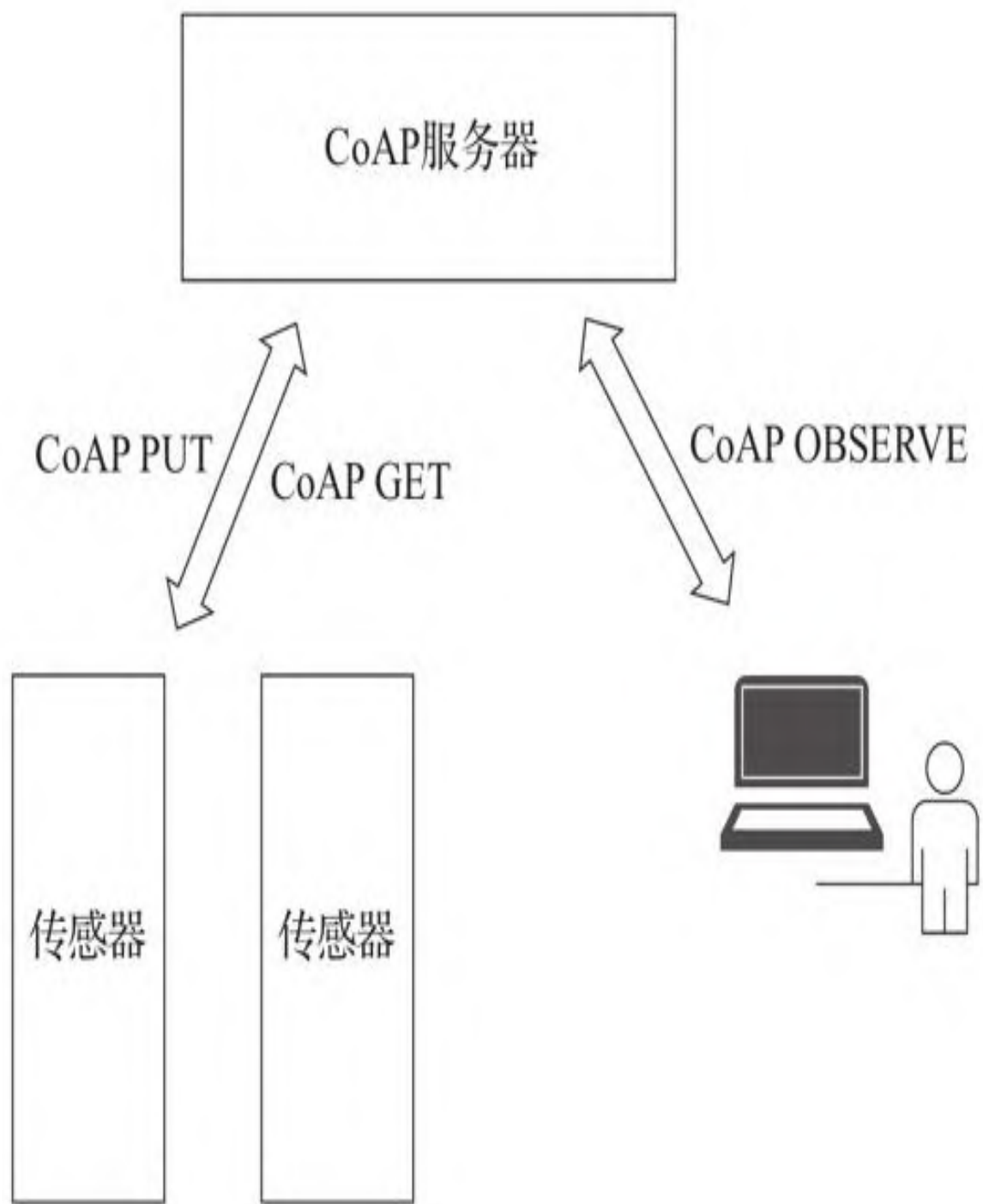


图2-3 CoAP协议的通信模型

我们会在第12章中对CoAP协议进行详细讲解。

## 2.4 LwM2M协议

LwM2M (Lightweight Machine-To-Machine) 协议是由Open Mobile Alliance (OMA) 定义的一套适用于物联网的轻量级协议。它使用RESTful接口, 提供设备的接入、管理和通信功能, 也适用于资源比较紧张的设备。LwM2M协议的架构如图2-4所示。

LwM2M协议底层使用CoAP协议传输数据和信令。而在LwM2M协议的架构中, CoAP协议可以运行在UDP或者SMS (短信) 之上, 通过DTLS (数据报传输层安全) 来实现数据的安全传输。

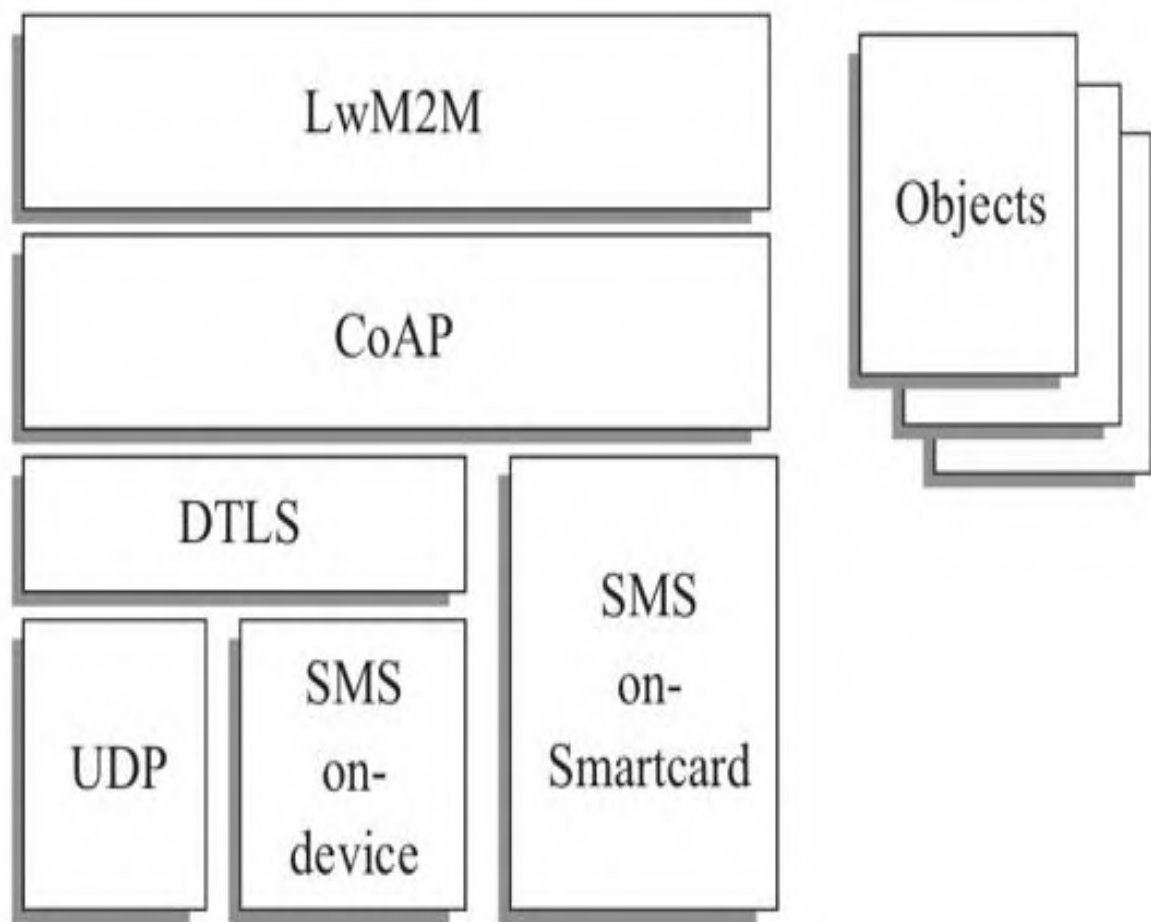


图2-4 LwM2M协议架构

在没有移动数据网络覆盖的地区，比如偏远地区的水电站，用短信作为信息传输的载体已经有比较长的历史了。

LwM2M协议架构主要包含3种实体——LwM2M Bootstrap Server、LwM2M Server和LwM2M Client。

LwM2M Bootstrap Server负责引导LwM2M Client注册并接入LwM2M Server，之后LwM2M Server和LwM2M Client就可以通过协议指定的接

口进行交互了。

## 2.5 HTTP协议

正如我们之前所讲，物联网也是互联网，HTTP这个在互联网中广泛应用的协议，在合适的环境下也可以应用到物联网中。在一些计算和硬件资源比较充沛的设备上，比如运行安卓操作系统的设备，完全可以使用HTTP协议上传和下载数据，就好像在开发移动应用一样。设备也可以使用运行在HTTP协议上的WebSocket主动接收来自服务器的数据。

我们会在第三部分讲解如何在物联网平台中使用HTTP协议。



## 2.6 LoRaWAN协议

LoRaWAN协议是由LoRa联盟提出并推动的一种低功率广域网协议，它和我们之前介绍的几种协议有所不同。MQTT协议、CoAP协议都是运行在应用层，底层使用TCP协议或者UDP协议进行数据传输，整个协议栈运行在IP网络上。而LoRaWAN协议则是物理层/数据链路层协议，它解决的是设备如何接入互联网的问题，并不运行在IP网络上。

说到设备如何接入互联网，我们很自然地想到4G、Wi-Fi，如果设备上有4G/Wi-Fi模块，或者支持以太网的网卡，就可以和其他联网终端，比如手机，以同样的方式接入互联网。

但是在某些情况下，4G或者Wi-Fi网络的覆盖非常困难，比如隧道施工的工程设备往往处于隧道深处几千米处，不可能用Wi-Fi或者4G网络覆盖。而工程设备经常在移动，使用有线网络与现场环境也不匹配。

LoRa (Long Range) 是一种无线通信技术，它具有使用距离远、功耗低的特点。在上面的场景下，用户就可以使用LoRaWAN技术进行组网，在工程设备上安装支持LoRa的模块。通过LoRa的中继设备将数据发往位于隧道外部的、有互联网接入的LoRa网关，LoRa网关再将数据

封装成可以在IP网络中通过TCP协议或者UDP协议传输的数据协议包（比如MQTT协议），然后发往云端的数据中心。

## 2.7 NB-IoT协议

NB-IoT (Narrow Band Internet of Things) 协议和LoRaWAN协议一样，是将设备接入互联网的物理层/数据链路层的协议。

和LoRA不同的是，NB-IoT协议构建和运行在蜂窝网络上，消耗的带宽较低，可以直接部署到现有的GSM网络或者LTE网络。设备安装支持NB-IoT的芯片和相应的物联网卡，然后连接到NB-IoT基站就可以接入互联网。而且NB-IoT协议不像LoRaWAN协议那样需要网关进行协议转换，接入的设备可以直接使用IP网络进行数据传输。

NB-IoT协议相比传统的基站，增益提高了约20dB，可以覆盖到地下车库、管道、地下室等之前信号难以覆盖的地方。

## 2.8 本章小结

本章简单介绍了一些常见的物联网协议，接下来进入实战的第一步，即MQTT协议详解与实战。

## 第二部分 MQTT协议详解与实战

- 第3章 MQTT协议基础
- 第4章 MQTT协议详解
- 第5章 MQTT协议实战

在第一部分中我们介绍了几种常用的物联网协议，目前的物联网通信协议并没有统一的标准。在这些协议中，MQTT协议（消息队列遥测传输协议）是目前应用最广泛的协议之一。可以这么说，MQTT协议之于物联网，就像HTTP协议之于互联网。目前，基本上所有开放云平台（比如，阿里云、腾讯云、青云等）都支持MQTT的接入，我们可以来看一下它们提供的物联网套件服务。

这些物联网套件服务对MQTT协议的支持都是第一位的。所以，想入门物联网，学习和了解MQTT协议是非常必要的，它解决了物联网中一个最基础的问题，即设备和设备、设备和云端服务之间的通信。

在接下来的几章里，我们将逐一学习MQTT协议的每一个特性及其最佳实践，并辅以实际的代码来进行讲解。其内容包括：

- MQTT协议数据包、数据收发流程详细解析;
- 如何在Web端和移动端正确地使用MQTT协议;
- 如何搭建自己的MQTT Broker;
- 如何增强MQTT平台的安全性;
- 使用MQTT协议设计和开发IoT产品和平台的最佳实践;
- MQTT 5.0的新特性。

最后，我们还会做一个“IoT+AI”的实战项目。

## 第3章 MQTT协议基础

MQTT协议是运行在TCP协议栈上的应用层协议，虽然MQTT协议的名称有Message和Queue两个词，但是它并不是像RabbitMQ那样的消息队列，这是初学者最容易搞混的一个问题。MQTT协议与传统的消息队列相比，有以下几个区别。

1) 传统消息队列在发送消息前必须先创建相应的队列。在MQTT协议中，不需要预先创建要发布的主题（可订阅的Topic）。

2) 传统消息队列中，未被消费的消息会被保存在某个队列中，直到有一个消费者将其消费。在MQTT协议中，如果发布一个没有被任何客户端订阅的消息，这个消息将被直接扔掉。

3) 传统消息队列中，一个消息只能被一个客户端获取。在MQTT协议中，一个消息可以被多个订阅者获取，MQTT协议也不支持指定消息被单一的客户端获取。

MQTT协议有几个不同的版本，目前支持和使用最广泛的版本是3.1.1。2017年8月，OASIS MQTT Technical Committee正式发布了用于Public Review的MQTT 5.0草案。2018年，MQTT 5.0正式发布。

MQTT 5.0在MQTT 3.1.1的基础上做了很多改变，并不向下兼容。  
本书以MQTT 3.1.1标准为主，同时也会讲到MQTT 5.0的新特性。



### 3.1 MQTT协议的通信模型

就像我们之前提到的，MQTT协议的通信是通过发布/订阅的方式来实现的，消息的发布方和订阅方通过这种方式进行解耦，它们之间没有直接连接，所以需要有一个中间方来对信息进行转发和存储。在MQTT协议里，我们称这个中间方为Broker，而连接到Broker的订阅方和发布方我们称之为Client。

一次典型的MQTT协议消息通信流程如图3-1所示。

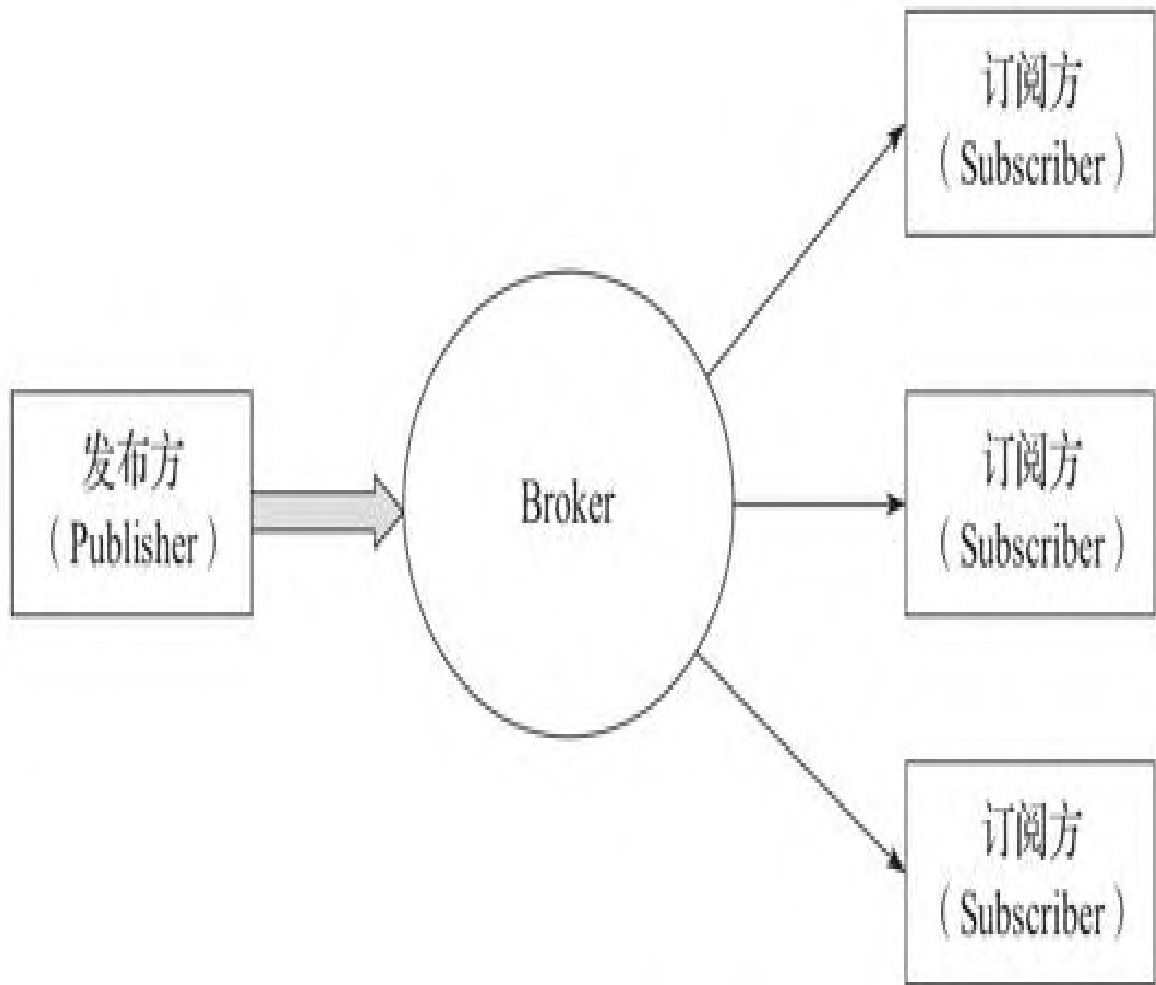


图3-1 MQTT协议消息通信流程

- 1) 发布方和订阅方都建立了到Broker的TCP连接。
- 2) 订阅方告知Broker它要订阅的消息主题 (Topic) 。
- 3) 发布方将消息发送到Broker，并指定消息的主题 (Topic) 。
- 4) Broker接收到消息以后，检查都有哪些订阅方订阅了这个主题 (Topic)，然后将消息发送到这些订阅方。

5) 订阅方从Broker获取该消息。

6) 如果某个订阅方此时处于离线状态，Broker可以先为它保存此条消息，当订阅方下次连接到Broker的时候，再将之前的消息发送到订阅方。

在本书的后续部分，我们将发送方称为Publisher，将订阅方称为Subscriber。

## 3.2 MQTT Client

任何终端，无论是嵌入式设备，还是服务器，只要运行了MQTT协议的库或者代码并连接了MQTT Broker，我们都称其为MQTT Client。Publisher和Subscriber都属于Client。一个Client是Pushlisher还是Subscriber，只取决于该Client当前的状态——是在发布消息还是在订阅消息。当然，一个Client可以同时是Publisher和Subscriber。

在大多数情况下，我们不需要自己按照MQTT的协议规范来实现一个MQTT Client，因为MQTT Client库在很多语言中都有实现，包括Android、Arduino、Ruby、C、C++、C#、Go、iOS、Java、JavaScript以及.NET等。如果你要查看相应语言的库实现，可以查看<https://github.com/mqtt/mqtt.github.io/wiki/libraries>。图3-2收集了MQTT Client在各种语言和平台上的实现。

# libraries

Raigeddas edited this page on 9 Nov 2019 · 74 revisions

Note: although there are a range of options available for developers interested in MQTT, not all of the client APIs listed below are current. Some are at an early or experimental stage of development, whilst others are stable and mature. Additionally, some may not provide full support for all of the features of the latest MQTT specification – for example, some may only support QoS 0, not include authentication, etc.

Check with the provider for the current status of your preferred language implementation; and remember to respect the licenses that different implementations are published under.

## Device-Specific

- [Arduino](#) (more information)
- [Espduino](#) (tailored Arduino library for the ESP8266)
- [mbed](#) (more information)
- [mbed](#) (simple port of the Arduino pubsubclient)
- [mbed](#) (native implementation)
- [mbed](#) (Paho Embedded C++ port) (more information)
- [mbed](#) (Paho Embedded C port) (more information)
- [Nanode](#)
- [Netduino](#)
- [M2MQTT](#) (works with .Net Micro Framework)

(see also [devices](#) page for more on hardware with built-in support)

## Actionscript

- [as3MQTT](#)

## Bash

- [see Shell Script, below](#)

图3-2 MQTT Client在各个平台上的实现

在本书中，我们将使用MQTT Client在NodeJS上进行代码演示和开发，首先你需要安装NodeJS，然后安装对应的MQTT协议包。

---

```
1. npm install mqtt -g
```

---

之后就可以在代码中使用MQTT Client的相关功能了。

### 3.3 MQTT Broker

搭建一个完整的MQTT协议环境，除了需要MQTT Client外，我们还需要一个MQTT Broker。如前文所述，Broker负责接收Publisher的消息，并将消息发送给相应的Subscriber，它是整个MQTT协议订阅/发布的核心。

在实际应用中，一个MQTT Broker还应该提供如下功能：

- 可以对Client的接入进行授权，并对Client进行权限控制；
- 可以横向扩展，比如集群，满足海量的Client接入；
- 有较好的扩展性，可以比较方便地接入现有业务系统；
- 易于监控，满足高可用性。

下面列举了几个比较常用的MQTT Broker。

#### (1) Mosquitto

Mosquitto是用C语言编写的一款开源MQTT Broker，单机配置和运行Mosquitto比较简单，官方并没有集群的解决方案。如果要扩展Mosquitto的功能，比如自定义的验证方式，实现过程比较复杂，对接现有的业务系统等也较为复杂，需要对Mosquitto代码比较熟悉。搭建

一个测试Broker或者单机验证功能环境，用Mosquitto还是比较合适的，如果想在生产系统中使用，则需要自行解决集群和扩展的问题。

## (2) EMQ X

EMQ X是由中国人开发并提供商业支持的MQTT Broker。EMQ X同时提供开源社区版和付费企业版。EMQ X是用Erlang语言编写的，官方提供集群解决方案，并可以通过编写插件的方式对Broker的功能进行扩展。EMQ X的开发社区比较活跃，青云提供的物联网套件就是基于EMQ X的。我在生产系统中使用EMQ X已经有几年了，EMQ X的性能在我看来是很不错的，唯一的缺点就是Erlang这门语言比较小众，而且学习曲线较陡，编写插件的时候可能需要重新学习一门语言。第5章会对EMQ X进行详细介绍。

## (3) HiveMQ

HiveMQ是用Java编写的MQTT Broker，支持集群，同时也可以通过插件的方式对功能进行扩展。Java语言的受众较广，功能扩展相对简单。不过HiveMQ是闭源的，只有付费企业版。

## (4) VerneMQ

VerneMQ是开源的，是用Erlang编写的MQTT Broker，且由一家位于瑞士的公司提供商业服务。VerneMQ同样支持集群，并可使用插件的



方式对功能进行扩展。

除了上面提到的几个常用的MQTT Broker，你还可以在<https://github.com/mqtt/mqtt.github.io/wiki/servers>找到更多的MQTT Broker。

除了自建Broker，我们还可以使用前面提到的阿里云、腾讯云、青云之类的云服务商提供的物联网云平台的MQTT协议服务。

如果只是抱着学习或者测试的目的，我们还可以使用一些公共的MQTT Broker，这里我们先使用一个公共的MQTT Broker（[mqtt.eclipse.org](https://mqtt.eclipse.org)）讲解和学习MQTT协议，在后面的章节里，再学习如何搭建一个MQTT Broker。

## 3.4 MQTT协议数据包格式

不同于HTTP协议，MQTT协议使用的是二进制数据包。MQTT协议的数据包非常简单，一个MQTT协议数据包由固定头（Fix Header）、可变头（Variable Header）、消息体（Payload）这3个部分依次组成。

- 固定头：存在于所有的MQTT协议数据包中，用于表示数据包类型及对应标识，表明数据包大小。

- 可变头：存在于部分类型的MQTT协议数据包中，具体内容由相应类型的数据包决定。

- 消息体：存在于部分MQTT协议数据包中，存储消息的具体数据。

这里我们首先看一下固定头，可变头和消息体将在讲解各种具体类型的MQTT协议数据包的时候详细讨论。

### 固定头格式

MQTT协议数据包的固定头的格式如图3-3所示。

Bit	7	6	5	4	3	2	1	0
字节1	MQTT 协议数据包类型				标识位 (Flag), 内容由数据包类型指定			
字节2	数据包剩余长度							

图3-3 MQTT协议数据包的固定头

(1) 数据包类型

MQTT协议数据包的固定头的第一个字节的高4位用于指定该数据包的类型。MQTT协议的数据包类型如表3-1所示。

表3-1 MQTT协议数据包类型

名称	值	方向	描述
Reserved	0	不可用	保留位
CONNECT	1	Client 到 Broker	Client 请求连接到 Broker
CONNACK	2	Broker 到 Client	连接确认
PUBLISH	3	双向	发布消息
PUBACK	4	双向	发布确认
PUBREC	5	双向	发布收到
PUBREL	6	双向	发布释放
PUBCOMP	7	双向	发布完成

(续)

名称	值	方向	描述
SUBSCRIBE	8	Client 到 Broker	Client 请求订阅
SUBACK	9	Broker 到 Client	订阅确认
UNSUBSCRIBE	10	Client 到 Broker	Client 请求取消订阅
UNSUBACK	11	Broker 到 Client	取消订阅确认
PINGREQ	12	Client 到 Broker	PING 请求
PINGRESP	13	Broker 到 Client	PING 应答
DISCONNECT	14	Client 到 Broker	Client 主动中断连接
Reserved	15	不可用	保留位

## (2) 数据包标识位

MQTT协议数据包的固定头的低4位用于指定数据包的标识位（Flag）。在不同类型的数据包中，标识位的定义是不一样的，每种数据包对应的标识位如表3-2所示。

表3-2 MQTT协议数据包的固定头的标识位含义

数据包	标识位	Bit 3	Bit 2	Bit 1	Bit 0
CONNECT	保留位	0	0	0	0
CONNACK	保留位	0	0	0	0
PUBLISH	MQTT 3.1.1 使用	DUP	QoS	QoS	RETAIN
PUBACK	保留位	0	0	0	0
PUBREC	保留位	0	0	0	0
PUBREL	保留位	0	0	0	0
PUBCOMP	保留位	0	0	0	0
SUBSCRIBE	保留位	0	0	0	0
SUBACK	保留位	0	0	0	0
UNSUBSCRIBE	保留位	0	0	0	0
UNSUBACK	保留位	0	0	0	0
PINGREQ	保留位	0	0	0	0
PINGRESP	保留位	0	0	0	0
DISCONNECT	保留位	0	0	0	0

注： DUP、QoS、RETAIN标识的含义将在后文进行详细讲解。

### （3）数据包剩余长度

从固定位的第二个字节开始，是用于标识当前数据包剩余长度的字段，剩余长度等于可变头长度加上消息体长度。

这个字段最少一个字节，最多4个字节。其中，每一个字节的最高位叫作延续位（Continuation Bit），用于标识在这个字节之后是否还有一个用于表示剩余长度的字节。剩下的低7位用于标识值，范围为0~127。

例如，剩余长度字段的第一个字节的最高位为1，那么意味着剩余长度至少还有1个字节，然后继续读下一个字节，下一个字节的最高位为0，那么剩余长度字段到此为止，一共2个字节。

不同长度的剩余长度字段可以标识的长度如表3-3所示。

表3-3 剩余长度字段可标识的数据包长度

字节数	可标识的最小长度	可标识的最大长度
1	0 ( 0x00 )	127 ( 0x7F )
2	128(0x80, 0x01)	16 383(0xFF, 0x7F)
3	16 384(0x80, 0x80, 0x01)	2 097 151(0xFF, 0xFF, 0x7F)
4	2 097 152(0x80, 0x80, 0x80, 0x01)	268 435 455 ( 0xFF, 0xFF, 0xFF, 0x7F )

所以，这4个字节最多可以标识的包长度为：

( 0xFF, 0xFF, 0xFF, 0x7F ) =268 435 455字节，即256MB，这是MQTT协议中数据包的最大长度。

### 3.5 本章小结

本章介绍了MQTT协议的通信模型，以及Client和Broker的概念，同时讲解了MQTT协议数据包的格式。接下来，将从建立MQTT协议连接开始，详细讲解MQTT协议的规范以及特性，并辅以实例代码。



## 第4章 MQTT协议详解

本章将从MQTT协议连接的建立开始，逐一讲解MQTT 3.1.1的各个特性，同时介绍MQTT 5.0的新特性。

## 4.1 建立到Broker的连接

Client在可以发布和订阅消息之前，必须先连接到Broker。

Client建立连接的流程如图4-1所示。

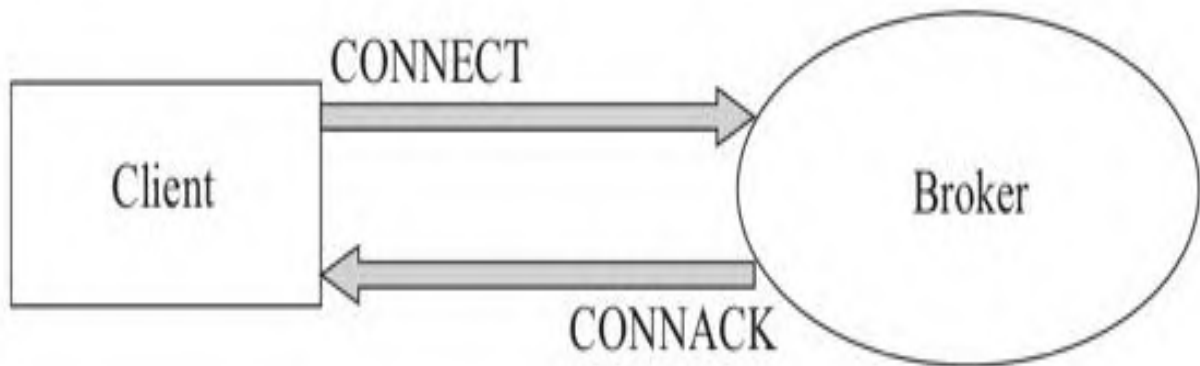


图4-1 Client建立连接的流程

1) Client向Broker发送一个CONNECT数据包；

2) Broker在收到Client的CONNECT数据包后，如果允许Client接入，则回复一个CONNACK包，该CONNACK包的返回码为0，表示MQTT协议连接建立成功；如果不允许Client接入，也回复一个CONNACK包，该CONNACK包的返回码为一个非0的值，用来标识接入失败的原因，然后断开底层的TCP连接。

### 4. 1. 1 CONNECT数据包

CONNECT数据包的格式如下。

#### 1. 固定头

CONNECT数据包的固定头格式如图4-2所示。

Bit	7	6	5	4	3	2	1	0
字节 1	1 (CONNECT)				保留			
字节 2	数据包剩余长度							

图4-2 CONNECT数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为1，代表CONNECT数据包。

#### 2. 可变头

可变头由4部分组成，依次为：协议名称、协议版本、连接标识和Keepalive。

协议名称是一个UTF-8编码字符串。在MQTT协议数据包中，字符串会有2个字节的前缀，用于标识字符串的长度。

协议名称的值固定为“MQTT”，加上前缀一共6个字节，如图4-3所示。

	值	7	6	5	4	3	2	1	0
字节1	0	0	0	0	0	0	0	0	0
字节2	4	0	0	0	0	0	1	0	0
字节3	'M'	0	1	0	0	1	1	0	1
字节4	'Q'	0	1	0	1	0	0	0	1
字节5	'T'	0	1	0	1	0	1	0	0
字节6	'T'	0	1	0	1	0	1	0	0

图4-3 协议名称字段

如果协议名称不正确，Broker会断开与Client的连接。

协议版本长度为1个字节，是一个无符号整数，MQTT 3.1.1的版本号为4，如图4-4所示。

	值	7	6	5	4	3	2	1	0
字节7	4	0	0	0	0	0	1	0	0

图4-4 协议版本号字段

连接标识长度为1个字节，字节中不同的位用于标识不同的连接选项，如图4-5所示。

	值	7	6	5	4	3	2	1	0
字节8		用户名标识	密码标识	遗嘱消息 Retain 标识	遗嘱消息 QoS 标识		遗嘱标识	会话清除标识	保留

图4-5 连接标识字段

每一个标识位的含义如下所示。

- 用户名标识（User Name Flag）：标识消息体中是否有用户名字段，长度为1bit，值为0或1；
- 密码标识（Password Flag）：标识消息体中是否有密码字段，长度为1bit，值为0或1；

- 遗愿消息Retain标识（Will Retain）：标识遗愿消息是否是Retain消息，长度为1bit，值为0或1；

- 遗愿消息QoS标识 (Will QoS)：标识遗愿消息的QoS，长度为2bit，值为0、1或2。

- 遗愿标识 (Will Flag)：标识是否使用遗愿消息，长度为1bit，值为0或1；

- 会话清除标识（Clean Session）：标识Client是否建立一个持久化的会话，长度为1bit，值为0或1。当Clean Session的标识设为0时，代表Client希望建立一个持久会话的连接，Broker将存储该Client订阅的主题和未接受的消息，否则Broker不会存储这些数据，并在建立连接时清除这个Client之前存在的持久会话所保存的数据。

可变头的最后2个字节代表连接的Keepalive，即连接保活设置，如图4-6所示。

	7	6	5	4	3	2	1	0
字节 9	Keepalive 高 8 位							
字节 10	Keepalive 低 8 位							

图4-6 连接保活字段

Keepalive代表一个单位为秒的时间间隔，Client和Broker之间在这个时间间隔之内至少要有一次消息交互，否则Client和Broker会认为它们之间的连接已经断开，我们会在4.4节中进行详细讲解。

### 3. 消息体

CONNECT数据包的消息体依次由5个字段组成：客户端标识符、遗嘱主题、遗嘱QoS、遗嘱消息、用户名和密码。除了客户端标识符外，其他4个字段都是可选的，由可变头里对应的连接标识来决定是否包含在消息体中。

这些字段有一个2个字节的前缀，用来标识字段值的长度，如图4-7所示。

字节	7	6	5	4	3	2	1	0
字节1	数据长度高8位							
字节2	数据长度低8位							
字节3	数据内容							

图4-7 MQTT协议的变长字段格式

· 客户端标识符 (Client Identifier) : Client Identifier是用来标识Client身份的字段, 在MQTT 3.1.1中, 这个字段的长度是1~23个字节, 而且只能包含数字和26个英文字母 (包括大小写), Broker通过这段来区分不同的Client。连接时, Client应该保证它的Identifier是唯一的, 通常我们可以使用UUID、唯一的设备硬件标识或者Android设备的DEVICE\_ID等作为Client Identifier的取值来源。

MQTT协议中要求Client连接时必须带上Client Identifier, 但也允许Broker在实现时接受Client Identifier为空的CONNECT数据包, 这时Broker会为Client分配一个内部唯一的Identifier。如果你需要使用持久性会话, 那就必须自己为Client设定一个唯一的Identifier。

· 用户名 (Username) : 如果可变头中的用户名标识为1, 那么消息体中将包含用户名字段, Broker可以使用用户名和密码对接入的Client进行验证, 只允许已授权的Client接入。注意, 不同的Client需要使用不同的Client Identifier, 但它们可以使用同样的用户名和密码进行连接。

· 密码 (Password) : 如果可变头中的密码标识为1, 那么消息体中将包含密码字段。



- 遗愿主题 (Will Topic) : 如果可变头中的遗愿标识为1, 那么消息体中将包含遗愿主题。当Client非正常地中断连接时, Broker将向指定的遗愿主题发布遗愿消息。

- 遗愿消息 (Will Message) : 如果可变头中的遗愿标识为1, 那么消息体中将包含遗愿消息。当Client非正常地中断连接时, Broker将向指定的遗愿主题发布由该字段指定的内容。

### 4. 1. 2 CONNACK数据包

当Broker收到Client的CONNECT数据包后，将检查并校验CONNECT数据包的内容，然后给Client回复一个CONNACK数据包。

CONNACK数据包的格式如下所示。

#### 1. 固定头

CONNACK数据包的固定头格式如图4-8所示。

Bit	7	6	5	4	3	2	1	0
字节 1	2 (CONNACK)				保留			
字节 2	2 (数据包剩余长度)							

图4-8 CONNACK的固定头

当固定头中的MQTT数据包的类型字段值为2时，则代表该数据包是CONNACK数据包。CONNACK数据包剩余长度固定为2。

#### 2. 可变头

CONNACK数据包的可变头为2个字节，由连接确认标识和连接返回码组成，如图4-9所示。

字节		7	6	5	4	3	2	1	0
字节1	连接确认标识	0	0	0	0	0	0	0	X
字节2	返回码	X	X	X	X	X	X	X	X

图4-9 CONNACK数据包的可变头

· 连接确认标识：连接确认标识的前7位都是保留的，必须设为0，最后一位是会话存在标识（Session Present Flag），值为0或1。当Client在连接时设置Clean Session=1，则CONNACK中的Session Present Flag始终为0；当Client在连接时设置Clean Session=0，那就有两种情况——如果Broker保存了这个Client之前留下的持久性会话，那么CONNACK中的Session Present Flag值为1；如果Broker没有保存该Client的任何会话数据，那么CONNACK中的Session Present Flag值为0。


连接返回码（Connect Return Code）：用于标识Client与Broker的连接是否建立成功。

连接返回码如表4-1所示。

表4-1 连接返回码

Return Code	连接状态
0	连接已建立
1	连接被拒绝，不允许的协议版本
2	连接被拒绝，Client Identifier 被拒绝
3	连接被拒绝，服务器不可用
4	连接被拒绝，错误的用户名或密码
5	连接被拒绝，未授权

这里重点讲一下Return Code 4和Return Code 5。Return Code 4在MQTT协议中的含义是用户名（Username）或密码（Password）的格式不正确，但是在大部分的Broker实现中，在使用错误的用户名或密码时，得到的返回码也是4。所以，这里我们认为4代表错误的用户名或密码。Return Code 5一般在Broker不使用用户名和密码而使用IP地址或者Client Identifier进行验证的时候使用，用来标识Client没有通过验证。

 **注意** Return Code 2代表Client Identifier格式不规范，比如长度超过23个字符、包含了不允许的字符等（部分Broker的实现在协议标准上做了扩展，比如允许超过23个字符的Client Identifier等）。

### 3. 消息体

CONNACK数据包没有消息体。

当Client向Broker发送CONNECT数据包并获得Return Code为0的CONNACK包后，就代表连接建立成功，可以发布和订阅消息了。

### 4.1.3 关闭连接

接下来我们看一下MQTT协议的连接是如何关闭的。MQTT协议的连接关闭可以由Client或Broker二者任意一方发起。

#### 1.Client主动关闭连接

Client主动关闭连接的流程非常简单，只需要向Broker发送一个DISCONNECT数据包就可以了。

DISCONNECT数据包固定头格式如图4-10所示。

Bit	7	6	5	4	3	2	1	0
字节 1	14 (DISCONNECT)				保留			
字节 2	0 (数据包剩余长度)							

图4-10 DISCONNECT数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为14，代表该数据包为DISCONNECT数据包。DISCONNECT的数据包剩余长度固定为0。

DISCONNECT数据包没有可变头和消息体。

在Client发送完DISCONNECT数据包之后，就可以关闭底层的TCP连接了，不需要等待Broker的回复，Broker也不会回复DISCONNECT数据包。

在这里，读者可能会有一个疑问，为什么需要在关闭TCP连接之前，发送一个与Broker没有交互的DISCONNECT数据包，而不是直接关闭底层的TCP连接？

这里涉及MQTT协议的一个特性，即Broker需要判断Client是否正常地断开连接。当Broker收到Client的DISCONNECT数据包的时候，会认为Client是正常地断开连接，那么它会丢弃当前连接指定的遗嘱消息。如果Broker检测到Client的TCP连接丢失，但又没有收到DISCONNECT数据包，它会认为Client是非正常断开连接，就会向在连接的时候指定的遗嘱主题发布遗嘱消息。

## 2. Broker主动关闭连接

MQTT协议规定Broker在没有收到Client的DISCONNECT数据包之前都应该保持和Client的连接，只有Broker在Keepalive的时间间隔里，没有收到Client的任何MQTT协议数据包时才会主动关闭连接。一些Broker的实现在MQTT协议上做了一些拓展，支持Client的连接管理，可以主动断开和某个Client的连接。

Broker主动关闭连接之前不需要向Client发送任何MQTT协议数据包，直接关闭底层的TCP连接就可以。



## 4.1.4 代码实践

接下来，我们将用代码来展示各种情况下MQTT协议连接的建立以及断开。

在这里，我们使用Node.js的MQTT库，请确保已安装Node.js，并通过`npm install mqtt--save`安装了MQTT库。

这里使用一个公共的Broker：`mqtt.eclipse.org`。

### 1. 建立持久会话的连接

首先引用MQTT库。

---

```
1. var mqtt = require('mqtt')
```

---

然后建立连接。

---

```
1. var client = mqtt.connect('mqtt://mqtt.eclipse.org', {  
2.   clientId: "mqtt_sample_id_1",  
3.   clean: false  
4. })
```

---

这里通过`clientId`选项指定Client Identifier，并通过`Clean`选项设定Clean Session为`false`，代表我们要建立一个持久会话的连

接。

接下来通过捕获connect事件将CONNACK数据包中的Return Code和Session Present Flag打印出来，然后断开连接。

---

```
1. client.on('connect', function (connack) {
2.   console.log('return code: ${connack.returnCode},
sessionPresent: ${connack.
  sessionPresent}')
3.   client.end()
```

---

完整的代码persistent\_connection.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_id_1",
4.   clean: false
5. })
6.
7. client.on('connect', function (connack) {
8.   console.log('return code: ${connack.returnCode},
sessionPresent: ${connack.
  sessionPresent}')
9.   client.end()
10. })
```

---

在终端上运行node persistent\_connection.js会得到以下输出。

---

```
return code: 0, sessionPresent: false
```

---

连接成功，因为是客户端标识符为“mqtt\_sample\_id\_1”的Client第一次建立连接，所以SessionPresent为false。

再次运行node persistent\_connection.js，输出就会变成SessionPresent。

---

```
return code: 0, sessionPresent: true
```

---

因为之前已经创建了一个持久会话，所以这次再使用同样的客户端标识符进行连接，得到的SessionPresent为true，表示会话已经存在了。

## 2. 建立非持久会话的连接

我们只需要将clean选项设为true，就可以建立一个非持久会话的连接了。完整的代码non\_persistent\_connection.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_id_1",
4.   clean: true
5. })
6.
7. client.on('connect', function (connack) {
8.   console.log('return code: ${connack.returnCode},
sessionPresent: ${connack.
sessionPresent}')
9.   client.end()
10. })
```

---

---

第4行代码将Clean Session设为true。

我们在终端上运行node persistent\_connection.js会得到以下输出。

---

```
return code: 0, sessionPresent: false
```

---

无论运行多少次，SessionPresent都会为false。

### 3. 使用相同的客户端标识符进行连接

接下来看一下如果两个Client使用相同的Client Identifier会发生什么事情。我们把代码稍微调整下，在连接成功时保持连接，然后捕获offline事件，在Client的连接被关闭时打印出来。

完整的代码identifcal.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_identical_1",
4. })
5.
6. client.on('connect', function (connack) {
7.   console.log('return code: ${connack.returnCode},
sessionPresent: ${connack.
sessionPresent}')
8. })
9.
10. client.on('offline', function () {
```

```
11.   console.log("client went offline")
12. })
```

---

从第10行代码开始，捕获Client的离线事件，并进行打印。

然后打开两个终端，分别运行node identifcal.js，这样就会看到在两个终端上不停地打印以下内容。

---

```
return code: 0, sessionPresent: false
client went offline
return code: 0, sessionPresent: false
client went offline
return code: 0, sessionPresent: false
.....
```

---

在MQTT协议中，两个Client使用相同的Client Identifier进行连接时，如果第二个Client连接成功，Broker会关闭与第一个Client的连接。

由于我们使用的MQTT库实现了断线重连功能，因此当连接被Broker关闭时，Client会尝试重新连接，结果就是这两个Client交替地把对方顶下线，我们就会看到上面所示的打印输出。因此，在实际应用中，一定要保证每一个设备使用的Client Identifier都是唯一的。

如果你观察到一个Client不停地上线和下线，那么就很有可能是由于Client Identifier冲突造成的。

在本节中，我们学习了MQTT连接关闭的过程，并且学习了连接建立和关闭的相关代码。在4.2节，我们将学习订阅和发布的概念，进而实现消息在Client之间的传输。

## 4.2 订阅与发布

上文介绍了MQTT基于订阅与发布的消息模型，MQTT协议的订阅与发布是基于主题的。一个典型的MQTT消息发送与接收的流程如图4-11所示。

- 1) ClientA连接到Broker。
- 2) ClientB连接到Broker，并订阅主题Topic1。
- 3) ClientA给Broker发送一个Publish数据包，主题为Topic1；
- 4) Broker收到ClientA的消息，发现ClientB订阅了Topic1，然后通过发送PUBLISH数据包的方式将消息转发到ClientB；
- 5) ClientB从Broker接收到该消息。

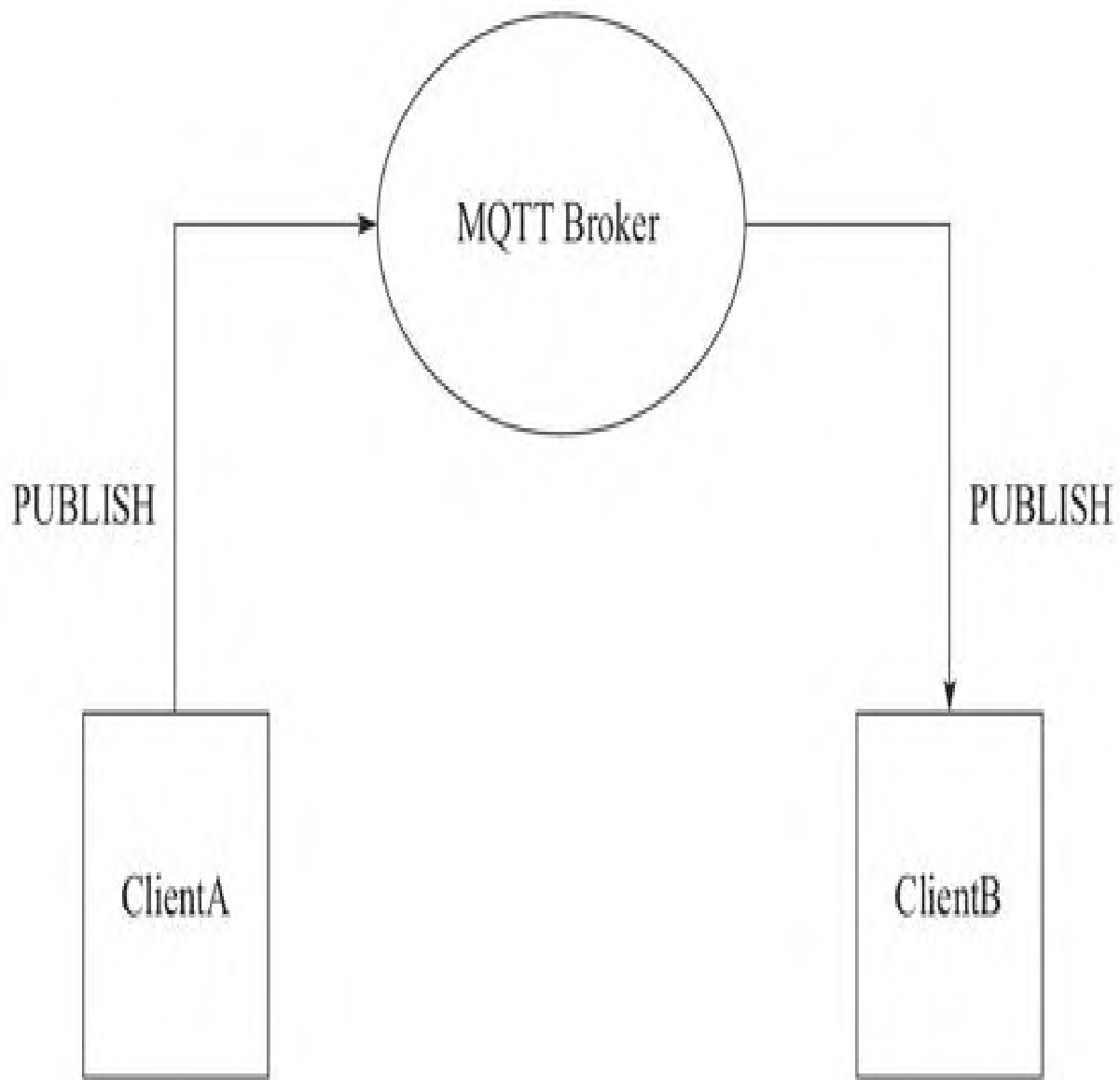


图4-11 MQTT消息的发送与接收流程

和传统的队列有点不同，如果ClientB在ClientA发布消息之后再订阅Topic1，那么ClientB就不会收到该消息。

MQTT协议通过订阅与发布模型对消息的发布者和订阅者进行解耦，发布者在发布消息时不需要订阅方也能连接到Broker，只要订阅



方之前订阅过相应主题，那么它在连接到Broker之后就可以收到发布方在它离线期间发布的消息。为了方便起见，在本书中我们称这种消息为离线消息。

接收离线消息需要Client使用持久会话，且发布时消息的QoS不小于1。

在继续学习前，我们有必要搞清楚两组概念：发布者（Publisher）和订阅者（Subscriber），发送方（Sender）和接收方（Receiver）。弄清这两组概念，我们才能更好地理解订阅和发布的流程以及QoS的概念。

## 1. Publisher和Subscriber

Publisher和Subscriber是相对于Topic来说的身份，如果一个Client向某个Topic发布消息，那么它就是Publisher；如果一个Client订阅了某个Topic，那么它就是Subscriber。在上面的例子中，ClientA是Publisher，ClientB是Subscriber。

## 2. Sender和Receiver

Sender和Receiver是相对于消息传输方向的身份，仍然用前面的例子做解释。

- 当ClientA发布消息时，它给Broker发送一条消息，那么ClientA是Sender，Broker是Receiver；

- 当Broker转发消息给ClientB时，Broker是Sender，ClientB则是Receiver。

Publisher/Subscriber、Sender/Receiver这两组概念最大的区别是Publisher和Subscriber只可能是Client，而Sender/Receiver有可能是Client，也有可能是Broker。

解释清楚这两组不同的概念之后，我们接下来看一下PUBLISH数据包。

### 4.2.1 PUBLISH数据包

PUBLISH数据包用于在Sender和Receiver之间传输消息数据，也就是说，当Publisher要向某个Topic发布一条消息的时候，Publisher会向Broker发送一个PUBLISH数据包；当Broker要将一条消息转发给订阅了某条主题的Subscriber时，Broker也会向Subscriber发送一条PUBLISH数据包。PUBLISH数据包的格式如下所示。

#### 1. 固定头

PUBLISH数据包的固定格式如图4-12所示。

Bit	7	6	5	4	3	2	1	0
字节 1	3 (PUBLISH)				DUP	QoS		保留
字节 2	数据包剩余长度							

图4-12 PUBLISH数据包的固定头

固定头中的MQTT协议数据包类型字段的值为3，代表该数据包是PUBLISH数据包。PUBLISH数据包固定头中的标识位（Flag）中有如下3个字段。

- 消息重复标识 (DUP Flag) : 长度为1bit, 值为0或1。当DUP Flag=1时, 代表该消息是一条重发消息, 因为Receiver没有确认收到之前的消息。这个标识只在QoS大于0的消息中使用。

- QoS: 长度为2bit, 值为0、1、2, 代表PUBLISH消息的服务质量级别。

- Retain标识 (Retain Flag) : 长度为1bit, 值为0或1。当Retain标识在从Client发送到Broker的PUBLISH消息中被设为1时, Broker应该保存该消息, 并且之后有任何新的Subscriber订阅PUBLISH消息中指定的主题时, 都会先收到该消息, 这种消息也被称为Retained消息; 当Retain标识在从Broker发送到Client的PUBLISH消息中被设为1时, 代表该消息是一条Retained消息。

## 2. 可变头

PUBLISH数据包的可变头由两个字段组成——主题名和包标识符 (Packet Identifier)。其中, Packet Identifier只会在QoS1和QoS2的PUBLISH数据包里出现, 我们在4.3节再详细讲解。

主题名是一个UTF-8编码的字符串, 它由两个前缀字节来辨识字符串的长度, 如图4-13所示。

字节	7	6	5	4	3	2	1	0
字节1	主题名长度高8位							
字节2	主题名长度低8位							
字节3...	主题名							

图4-13 主题名字段

由于只有2个字节标识主题名长度，所以主题名的最大长度为65535字节。

虽然主题名可以是长度从1~65535的任意字符串（可以包含空格），但是在实际项目中，我们最好还是遵循以下一些命名规则。

- 主题名称应该包含层级，不同的层级用“/”划分，比如，2楼201房间的温度感应器可以用主题：“home/2ndfloor/201/temperature”表示。

- 主题名称开头不要使用“/”，例如：  
“/home/2ndfloor/201/temperature”。

- 不要在主题中使用空格。

- 只使用ASCII字符。

- 主题名称在可读的前提下尽量短一些。
- 主题名称对大小写是敏感的，“Home”和“home”是两个不同的主题。
- 可以将设备的唯一标识加到主题中，比如：  
“warehouse/shelf/shelf1\_ID/status”。
- 主题尽量精确，不要使用泛用的主题，例如在201房间中有3个传感器，温度传感器、亮度传感器和湿度传感器，那么你应该使用3个主题名称：“home/2ndfloor/201/temperature”  
“home/2ndfloor/201/brightness”和“home/2ndfloor/201/humidity”，  
而不是让3个传感器都使用“home/2ndfloor/201”这个主题名。
- 以“\$”开头的主题属于Broker预留的系统主题，通常用于发布Broker的内部统计信息，比如“\$SYS/broker/clients/connected”。应用程序不要使用“\$”开头的主题收发数据。

### 3. 消息体

PUBLISH数据包的消息体就是该数据包要发送的数据，它可以是任意格式的数据，比如二进制数据、文本、JSON等。具体数据格式由应用程序定义。在实际生产中，我们可以使用JSON、Protocol Buffer等格式对数据进行编码。

消息体中数据的长度可以由固定头中的数据包剩余长度减去可变头的长度得到。

## 4.2.2 代码实践：发布消息

接下来写一小段代码，目的是向一个主题发布一条QoS为1的使用JSON编码的数据，然后退出。代码如下。

---

```
1. //publisher.js
2.
3. var mqtt = require('mqtt')
4. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
5.   clientId: "mqtt_sample_publisher_1",
6.   clean: false
7. })
8.
9. client.on('connect', function (connack) {
10.   if(connack.returnCode == 0){
11.     client.publish("home/2ndfloor/201/temperature",
JSON.stringify({current:
25})), {qos: 1}, function (err) {
12.       if(err == undefined) {
13.         console.log("Publish finished")
14.         client.end()
15.       }else{
16.         console.log("Publish failed")
17.       }
18.     })
19.   }else{
20.     console.log('Connection failed:
${connack.returnCode}')
21.   }
22. })
```

---

第11行代码表示向主题“home/2ndfloor/201/temperature”，发送一条QoS为1的消息，消息的内容是格式为JSON的字符串。



运行 “node publisher.js”，会得到以下输出。

---

```
Publish finished
```

---

### 4.2.3 订阅一个主题

ClientB想要接收ClientA发布到某个主题的消息，就必须先向Broker订阅这个主题，订阅一个主题的流程如图4-14所示。

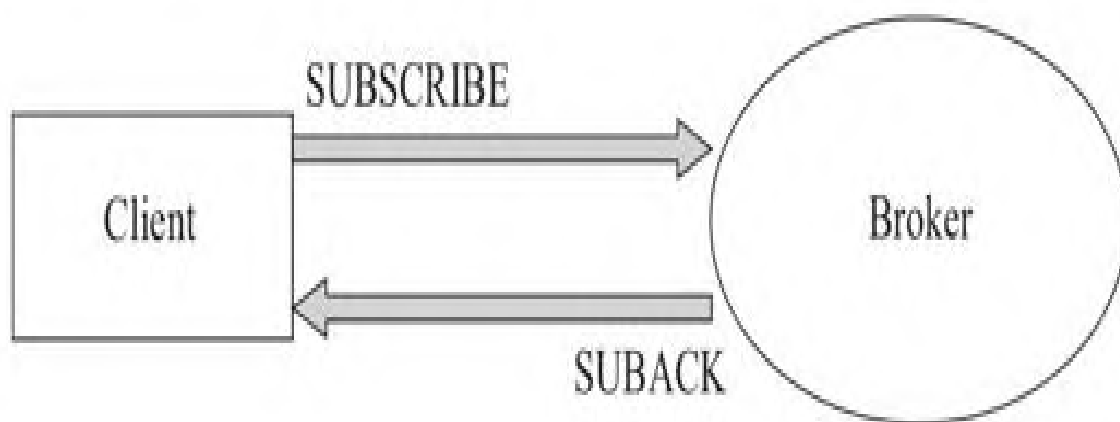


图4-14 Client的订阅流程

1) Client向Broker发送一个SUBSCRIBE数据包，其中包含Client想要订阅的主题以及其他参数。

2) Broker收到SUBSCRIBE数据包后，向Client发送一个SUBACK数据包作为应答。

接下来我们看一下数据包的具体内容。

#### 1. SUBSCRIBE数据包

(1) 固定头

SUBSCRIBE数据包的固定头格式如图4-15所示。

Bit	7	6	5	4	3	2	1	0
字节1	8 (SUBSCRIBE)				保留			
字节2	数据包剩余长度							

图4-15 SUBSCRIBE数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为8，代表该数据包是SUBSCRIBE数据包。

(2) 可变头

SUBSCRIBE数据包的可变头只包含一个两字节的包标识符，用来唯一标识一个数据包。数据包标识只需要保证从Sender到Receiver的一次消息交互中唯一即可。SUBSCRIBE数据包可变头的标识符格式如图4-16所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-16 SUBSCRIBE数据包可变头的包标识符格式

### （3）消息体

SUBSCRIBE数据包中的消息体由Client要订阅的主题列表构成。和PUBLISH数据包的主题名不同，SUBSCRIBE数据包中的主题名可以包含通配符，通配符包括单层通配符“+”和多层通配符“#”。使用包含通配符的主题名可以订阅满足匹配条件的所有主题。为了和PUBLISH数据包中的主题名进行区分，我们称SUBSCRIBE数据包中的主题名为主题过滤器（Topic Filter）。

单层通配符“+”：如之前所述，MQTT协议的主题名是具有层级概念的，不同的层级间用“/”分割，“+”可以用来指代任意一个层级。

例如：“home/2ndfloor/+/temperature”，可匹配：  
home/2ndfloor/201/temperature、  
home/2ndfloor/202/temperature；不可匹配：

home/2ndfloor/201/livingroom/temperature、  
home/3ndfloor/301/temperature。

多层通配符“#”：“#”和“+”的区别在于，“#”可以用来指定任意多个层级，但是“#”必须是Topic Filter的最后一个字符，同时必须跟在“/”后面，除非Topic Filter只包含“#”这一个字符。

例如：“home/2ndfloor/#”，可匹配：home/2ndfloor、  
home/2ndfloor/201、home/2ndfloor/201/temperature、  
home/2ndfloor/202/temperature、  
home/2ndfloor/201/livingroom/temperature；不可匹配：  
home/3ndfloor/301/temperature。



**注意** “#”是一个合法的Topic Filter，代表所有的主题；而  
“home#”不是一个合法的Topic Filter，因为“#”号需要跟在“/”后面。

每一个Topic Filter必须是一个UTF-8编码的字符串，在这个字符串后面紧跟着1个字节，用于描述订阅该主题的QoS。Topic Filter的格式如图4-17所示。

字节	7	6	5	4	3	2	1	0
字节1	长度高8位							
字节2	长度低8位							
字节3..N	主题过滤器							
	Reserved						QoS	
字节N+1	0	0	0	0	0	0	X	X

图4-17 Topic Filter的格式

QoS字节的最后2位用于标识QoS值，值为0、1或2。

消息体的主题列表按照上面的格式依次拼接即可。

## 2. SUBACK数据包

为了确认每一次的订阅，Broker在收到SUBSCRIBE数据包后都会回复一个SUBACK数据包作为应答。

### (1) 固定头

SUBACK数据包的固定头如图4-18所示。

Bit	7	6	5	4	3	2	1	0
字节1	9 (SUBACK)				保留			
字节2	数据包剩余长度							

图4-18 SUBACK数据包的固定头

固定头中的MQTT协议数据包类型字段的值为9，代表该数据包是SUBACK数据包。

(2) 可变头

SUBACK数据包的可变头只包含一个两字节的包标识符，其格式如图4-19所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-19 SUBACK数据包的可变头

(3) 消息体

SUBACK数据包包含一组返回码，返回码的数量和顺序与SUBSCRIBE数据包的订阅列表对应，用于标识订阅类别中每一个订阅项的订阅结果。

SUBACK数据包中每一个返回码为一个字节，如图4-20所示。

字节	7	6	5	4	3	2	1	0
字节1	返回码							

图4-20 返回码字段

返回码列表按照图4-20的格式依次拼接而成。

返回码的对应值如表4-2所示。

表4-2 返回码的对应值

返回码	含义
0	订阅成功，最大可用 QoS 为 0
1	订阅成功，最大可用 QoS 为 1
2	订阅成功，最大可用 QoS 为 2
128	订阅失败



返回码0~2代表订阅成功，同时Broker授予Subscriber不同的QoS等级，这个等级可能会与Subscriber在SUBSCRIBE数据包中要求的不一樣。

返回码128代表订阅失败，比如Client没有权限订阅某个主题，或者要求订阅的主题格式不正确等。

## 4.2.4 代码实践：订阅主题

接下来，试着写一下订阅并处理消息的代码。订阅主题为4.2.2节中代码实现的publisher.js，然后通过捕获“message”事件获取接收的消息并进行打印。

通常，在建立和Broker的连接后我们就可以开始订阅了，但这里有一个小小的优化，如果你建立的是持久会话的连接，那么Broker有可能已经保存了之前连接时订阅的主题，这样就没必要再发起SUBSCRIBE请求了。这个小优化在网络带宽或者设备处理能力较差时尤为重要。

完整的代码subscriber.js如下。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_subscriber_id_1",
4.   clean: false
5. })
6.
7. client.on('connect', function (connack) {
8.   if(connack.returnCode == 0) {
9.     if (connack.sessionPresent == false) {
10.       console.log("subscribing")
11.
12.       client.subscribe("home/2ndfloor/201/temperature", {
13.         qos: 1
14.       }, function (err, granted) {
```

```
14.         if (err !== undefined) {
15.             console.log("subscribe failed")
16.         } else {
17.             console.log('subscribe succeeded with
18.             ${granted[0].topic}, qos:
19.             ${granted[0].qos}')
20.         }
21.     })
22. }else {
23.     console.log('Connection failed:
24.     ${connack.returnCode}')
25. }
26. })
27. client.on("message", function (_, message, _) {
28.     var jsonPayload = JSON.parse(message.toString())
29.     console.log('current temperature is
30.     ${jsonPayload.current}')
31. })
```

---

第9行代码通过判断CONNACK的SessionPresent标识，来决定是否发起订阅，如果Session已经存在，则不再发起订阅。

第11行代码指定订阅主题“home/2ndfloor/201/temperature”，订阅的QoS等级为1。

在终端上运行“node subscriber.js”会得到以下输出。

---

```
subscribing
subscribe succeeded with home/2ndfloor/201/temperature,
qos: 1
```

---

第一次运行上述代码的时候，Broker上面没有保存这个Client的会话，所以需要进行订阅，现在点击“Ctrl+C”终止这段代码运行，然后重新运行，因为Broker上已经保存了这个Client的会话，不需要再订阅，所以我们也不会看到订阅相关的输出。

在4.2.5节中，我们运行过publisher.js，向“home/2ndfloor/201/temperature”这个主题发布过一个消息，但是这发生在subscriber.js订阅该主题之前，所以现在Subscriber不会收到任何消息，我们需要再运行一次publish.js，然后在运行subscriber.js的终端上会得到如下输出。

---

```
current temperature is 25
```

---

这样，我们就通过MQTT协议完成了一次点对点的消息传递，同时也验证了建立持久会话连接之后，Broker会保存Client的订阅信息。

### 4.2.5 取消订阅

Subscriber也可以取消对某些主题的订阅。取消订阅的流程如图4-21所示。

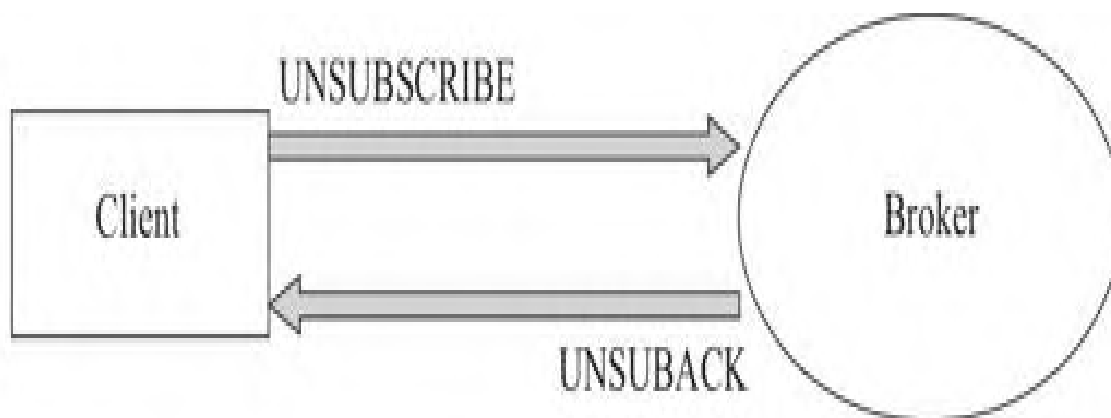


图4-21 取消订阅流程

(1) Client向Broker发送一个UNSUBSCRIBE数据包，其中包含Client想要取消订阅的主题。

(2) Broker收到UNSUBSCRIBE数据包后，向Client发送一个UNSUBACK数据包作为应答。

接下来看一下数据包的具体内容。

#### 1. UNSUBSCRIBE数据包

(1) 固定头

UNSUBSCRIBE数据包的固定头格式如图4-22所示。

Bit	7	6	5	4	3	2	1	0
字节1	10 (UNSUBSCRIBE)				保留			
字节2	数据包剩余长度							

图4-22 UNSUBSCRIBE数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为10，代表该数据包是UNSUBSCRIBE数据包。

(2) 可变头

UNSUBSCRIBE数据包的可变头只包含一个2字节的包标识符，包标识符的格式如图4-23所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-23 UNSUBSCRIBE数据包可变头中的包标识符

### (3) 消息体

UNSUBSCRIBE数据包包含要取消的主题过滤器（Topic Filter）列表，这些主题过滤和SUBSCRIBE数据包中的规则是一样的，不过不再包含QoS字段。其格式如图4-24所示。

字节	7	6	5	4	3	2	1	0
字节1	长度高8位							
字节2	长度低8位							
字节3..	主题过滤器							

图4-24 UNSUBSCRIBE数据包的消息体

UNSUBSCRIBE数据包的消息体的主题列表按照图4-24的格式依次拼接而成。

和订阅时不同，取消订阅时，主题名中的通配符并不起通配作用。取消订阅的主题名必须每个字符都和订阅时指定的主题名相同，这样才能被取消，举个例子。

订阅主题名为“home/2ndfloor/201/temperature”，取消订阅名为“home/+/201/temperature”，并不会取消之前的订阅。

同理，订阅的时候使用了通配符，取消订阅的时候也必须使用完全一样的主题名。

订阅主题名为“home/+/201/temperature”，取消订阅名为“home/+/201/temperature”，这样才能取消之前的订阅。

## 2. UNSUBACK数据包

Broker在收到UNSUBSCRIBE数据包后，会回复给Client一个UNSUBACK数据包作为响应。

### (1) 固定头

UNSUBACK数据包的固定头如图4-25所示。



Bit	7	6	5	4	3	2	1	0
字节1	11 (UNSUBACK)				保留			
字节2	2(数据包剩余长度)							

图4-25 UNSUBACK数据包的固定头

固定头中的MQTT协议数据包类型字段的值为10，代表该数据包是UNSUBACK数据包。UNSUBACK数据包中的固定头的数据包剩余长度字段固定为2。

## (2) 可变头

UNSUBACK数据包的可变头只包含一个2字节的包标识符，如图4-26所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-26 UNSUBACK可变头的包标识符

### (3) 消息体

UNSUBACK数据包没有消息体。

## 3. 代码实践：取消订阅

下面要完成的代码很简单，只需要在建立连接后取消之前订阅的主题。

完整的代码unsubscribe.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org', {
3.   clientId: "mqtt_sample_subscriber_id_1",
4.   clean: false
5. })
6.
7. client.on('connect', function (connack) {
8.   if (connack.returnCode == 0) {
9.     console.log("unsubscribing")
10.
11. client.unsubscribe("home/2ndfloor/201/temperature",
12. function (err) {
13.   if (err != undefined) {
14.     console.log("unsubscribe failed")
15.   } else {
16.     console.log("unsubscribe succeeded")
17.   }
18.   client.end()
19. })
20. } else {
21.   console.log('Connection failed:
22.   ${connack.returnCode}')
23. }
24. })
```

---

在终端上运行“node unsubscribe.js”，会得到以下输出。

---

```
unsubscribing
unsubscribe succeeded
```

---

这里取消了对“home/2ndfloor/201/temperature”的订阅，所以再次运行subscriber.js和publisher.js的时候，在运行subscribe.js的终端上就不会再有“home/2ndfloor/201/temperature”的打印信息了。如何使subscriber.js重新订阅这个主题呢？读者可以参考上文进行思考，然后自己动手实现。

在本节中，我们学习了MQTT协议发布、订阅消息的模型及其特性，并第一次实现了消息的点对点传输。接下来，我们将学习MQTT协议中的一个非常重要的特性——QoS等级。

## 4.3 QoS及其最佳实践

前文多次提到了QoS（Quality of Service），CONNECT数据包、PUBLISH数据包、SUBSCRIBE数据包中都有QoS的标识。那么MQTT协议提供的QoS是什么呢？

### 4.3.1 MQTT协议中的QoS等级

作为最初用来在网络带宽窄、信号不稳定的环境下传输数据的协议，MQTT协议设计了一套保证消息稳定传输的机制，包括消息应答、存储和重传。在这套机制下，MQTT协议还提供了3种不同层次的QoS。

- QoS0: At most once, 至多一次。
- QoS1: At least once, 至少一次。
- QoS2: Exactly once, 确保只有一次。

这三个层次都是什么意思呢？QoS是消息的发送方（Sender）和接收方（Receiver）之间达成的一个协议。

· QoS0: 表示Sender发送一条消息，Receiver最多能收到一次，也就是说Sender尽力向Receiver发送消息，如果发送失败，则放弃。

· QoS1: 表示Sender发送一条消息，Receiver至少能收到一次，也就是说Sender向Receiver发送消息，如果发送失败，Sender会继续重试，直到Receiver收到消息为止，但是因为重传的原因，Receiver可能会收到重复的消息。

· QoS2: 表示Sender发送一条消息, Receiver确保能收到且只收到一次, 也就是说Sender尽力向Receiver发送消息, 如果发送失败, 会继续重试, 直到Receiver收到消息为止, 同时保证Receiver不会因为消息重传而收到重复的消息。

注意, QoS是Sender和Receiver之间达成的协议, 不是Publisher和Subscriber之间达成的协议。也就是说, Publisher发布一条QoS1的消息, 只能保证Broker至少收到一次; 而对应的Subscriber能否至少收到一次这个消息, 还要取决于Subscriber在Subscribe的时候和Broker协商的QoS等级。

接下来, 看一下QoS0、QoS1和QoS2的机制, 并讨论一下什么是QoS降级。

### 4.3.2 QoS0

QoS0是最简单的一个QoS等级。在QoS0等级下，Sender和Receiver之间一次消息的传递流程如图4-27所示。

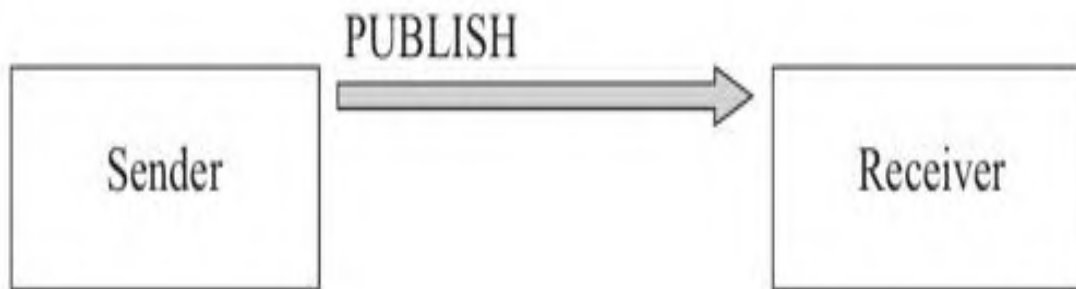


图4-27 QoS0等级下Sender和Receiver之间的消息传递流程

Sender向Receiver发送一个包含消息数据的PUBLISH数据包，然后不管结果如何，丢弃已发送的PUBLISH数据包，这样一条消息即可完成发送。

### 4.3.3 QoS1

QoS1要保证消息至少到达Receiver一次，所以这里有一个应答的机制。在QoS1等级下，Sender和Receiver之间一次消息的传递流程如图4-28所示。

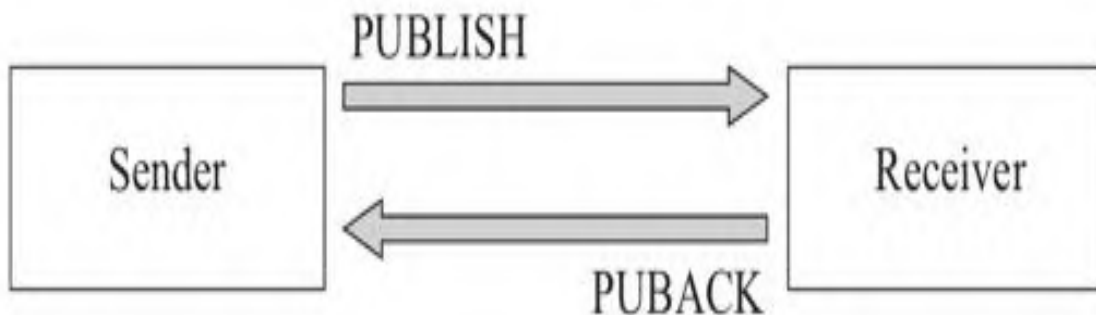


图4-28 QoS1等级下Sender和Receiver之间的消息传递流程

1) Sender向Receiver发送一个带有消息数据的PUBLISH数据包，并在本地保存这个PUBLISH数据包。

2) Receiver在收到PUBLISH数据包后，向Sender发送一个PUBACK数据包，PUBACK数据包没有消息体，只在可变头中有一个包标识（Packet Identifier），和它收到的PUBLISH数据包中的Packet Identifier一致。



3) Sender在收到PUBACK数据包之后，根据PUBACK数据包中的Packet Identifier找到本地保存的PUBLISH数据包，然后丢弃，这样一次消息发送完成。

4) 如果Sender在一段时间内没有收到PUBLISH数据包对应的PUBACK数据包，那么它会将PUBLISH数据包中的DUP标识设为1（代表的是重新发送PUBLISH数据包），然后重新发送该PUBLISH数据包。重复这个流程，直到Sender收到PUBACK数据包，然后执行第3步。

1. QoS>0时的PUBLISH数据包

当QoS为1或2时，PUBLISH数据包的可变头中将包含包标识符字段。包标识符长度为两个字节，如图4-29所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-29 包标识符

包标识符用来唯一标识一个MQTT协议数据包，但它并不要求全局唯一，只要保证在一次完整的消息传递过程中是唯一的就可以。

## 2. PUBACK数据包

### (1) 固定头

PUBACK数据包的固定头格式如图4-30所示。

Bit	7	6	5	4	3	2	1	0
字节1	4 (PUBACK)				保留			
字节2	2 (数据包剩余长度)							

图4-30 PUBACK数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为4，代表的是PUBACK数据包。PUBACK数据包的剩余长度字段值固定为2。

### (2) 可变头

PUBACK数据包的可变头只包含一个2字节的包标识符，如图4-31所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高 8 位							
字节2	包标识符低 8 位							

图4-31 包标识符

(3) 消息体

PUBACK数据包没有消息体。

#### 4.3.4 QoS2

QoS0和QoS1是相对简单的QoS等级，QoS2不仅要确保Receiver能收到Sender发送的消息，还要保证消息不重复。所以，它的重传和应答机制就要更复杂，同时开销也是最大的。

在QoS2等级下，Sender和Receiver之间一次消息的传递流程如图4-32所示。

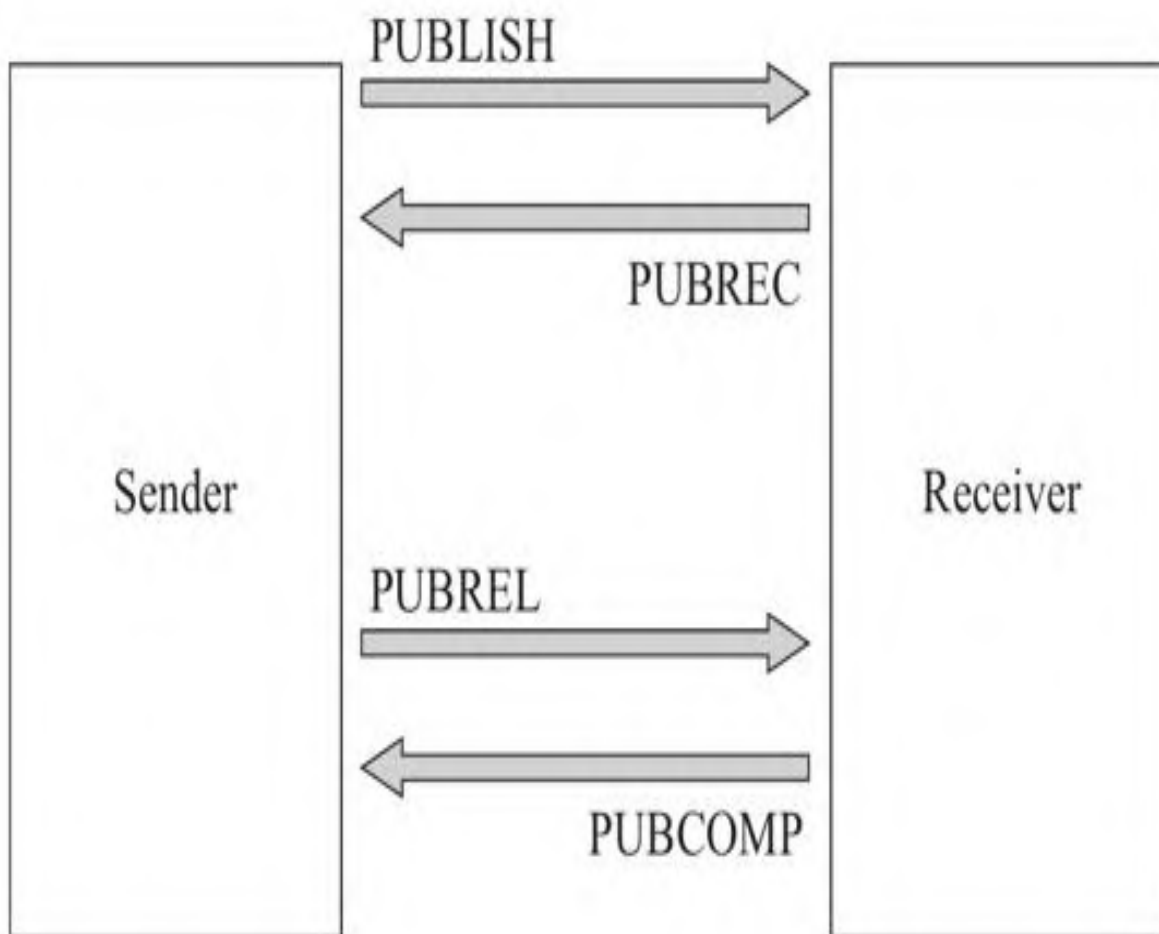


图4-32 QoS2等级下Sender和Receiver之间的消息传递流程

QoS2使用2套请求/应答流程（一个4段的握手）来确保Receiver收到来自Sender的消息，且不重复。

1) Sender发送QoS值为2的PUBLISH数据包，假设该数据包中Packet Identifier为P，并在本地保存该PUBLISH数据包。

2) Receiver收到PUBLISH数据包后，在本地保存PUBLISH数据包的Packet Identifier为P，并回复Sender一个PUBREC数据包。PUBREC数据包可变头中的Packet Identifier为P，没有消息体。

3) 当Sender收到PUBREC数据包后，它就可以安全地丢掉初始的Packet Identifier为P的PUBLISH数据包，同时保存该PUBREC数据包，并回复Receiver一个PUBREL数据包。PUBREL数据包可变头中的Packet Identifier为P，没有消息体；如果Sender在一定时间内没有收到PUBREC数据包，它会把PUBLISH数据包的DUP标识设为1，重新发送该PUBLISH数据包。

4) 当Receiver收到PUBREL数据包时，它可以丢弃掉保存的PUBLISH数据包的Packet Identifier P，并回复Sender一个PUBCOMP数据包。PUBCOMP数据包可变头中的Packet Identifier为P，没有消息体。

5) 当Sender收到PUBCOMP数据包，它会认为数据包传输已完成，会丢掉对应的PUBREC数据包。如果Sender在一定时间内没有收到PUBCOMP数据包，则会重新发送PUBREL数据包。

我们可以看到，在QoS2中，想要完成一次消息的传递，Sender和Receiver之间至少要发送4个数据包，所以说，QoS2是最安全，也是最慢的一种QoS等级。

1. PUBREC数据包

(1) 固定头

PUBREC数据包的固定头格式如图4-33所示。

Bit	7	6	5	4	3	2	1	0
字节1	5 (PUBREC)				保留			
字节2	2 (数据包剩余长度)							

图4-33 PUBREC数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为5，代表该数据包是PUBREC数据包。PUBREC数据包的剩余长度字段值固定为2。

(2) 可变头

PUBREC数据包的可变头只包含一个2字节的包标识符，如图4-34所示。



图4-34 PUBREC数据包可变头中的包标识符

(3) 消息体

PUBREC数据包没有消息体。

2. PUBREL数据包

(1) 固定头

PUBREL数据包的固定头格式如图4-35所示。

Bit	7	6	5	4	3	2	1	0
字节1	6 (PUBREL)				保留			
字节2	2 (数据包剩余长度)							

图4-35 PUBREL数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为6，代表该数据包是PUBREL数据包。PUBREL数据包的剩余长度字段值固定为2。

(2) 可变头

PUBREL数据包的可变头只包含一个2字节的包标识符，如图4-36所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-36 PUBREL数据包可变头中的包标识符

(3) 消息体



PUBREL数据包没有消息体。

3. PUBCOM数据包

(1) 固定头

PUBCOM数据包的固定头格式如图4-37所示。

Bit	7	6	5	4	3	2	1	0
字节 1	7 (PUBCOM)				保留			
字节 2	2 (数据包剩余长度)							

图4-37 PUBCOM数据包的固定头格式

固定头中的MQTT协议数据包类型字段的值为7，代表该数据包是PUBCOM数据包。PUBCOM数据包的剩余长度字段值固定为2。

(2) 可变头

PUBCOM数据包的可变头只包含一个2字节的包标识符，如图4-38所示。

字节	7	6	5	4	3	2	1	0
字节1	包标识符高8位							
字节2	包标识符低8位							

图4-38 PUBCOM数据包可变头中的包标识符

(3) 消息体

PUBCOM数据包没有消息体。

### 4.3.5 代码实践：使用不同的QoS发布消息

本节会实现一个发布端和一个订阅端，它们可以通过命令行参数来指定发布和订阅的QoS，同时，通过捕获packetsend和packetreceive事件，将发送和接收到的MQTT协议数据包的类型打印出来。

发布端的代码publish\_with\_qos.js如下所示。

---

```
1. var args = require('yargs').argv;
2. var mqtt = require('mqtt')
3. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
4.   clientId: "mqtt_sample_publisher_2",
5.   clean: false
6. })
7.
8. client.on('connect', function (connack) {
9.   if (connack.returnCode == 0) {
10.     client.on('packetsend', function (packet) {
11.       console.log('send: ${packet.cmd}')
12.     })
13.     client.on('packetreceive', function (packet) {
14.       console.log('receive: ${packet.cmd}')
15.     })
16.     client.publish("home/sample_topic",
JSON.stringify({data: 'test'}), {qos:
args.qos})
17.   } else {
18.     console.log('Connection failed:
${connack.returnCode}')
19.   }
20. })
```

---

第10~12行代码捕获packetsend事件，然后将发送的MQTT协议数据包的类型打印出来。

第13~15行代码捕获packetreceive事件，然后将收到的MQTT协议数据包的类型打印出来。

第16行代码使用命令行中指定的QoS来发布消息。

订阅端的代码subscribe\_with\_qos.js如下所示。

---

```
1. var args = require('yargs').argv;
2. var mqtt = require('mqtt')
3. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
4.   clientId: "mqtt_sample_subscriber_id_2",
5.   clean: false
6. })
7.
8.
9. client.on('connect', function (connack) {
10.   if (connack.returnCode == 0) {
11.     client.subscribe("home/sample_topic", {qos:
args.qos}, function () {
12.       client.on('packetsend', function (packet) {
13.         console.log('send: ${packet.cmd}')
14.       })
15.       client.on('packetreceive', function (packet) {
16.         console.log('receive: ${packet.cmd}')
17.       })
18.     })
19.   } else {
20.     console.log('Connection failed:
${connack.returnCode}')
21.   }
22. })
```

---

第11行代码使用命令行中指定的QoS进行订阅。


第12~14行代码捕获packet<sub>send</sub>事件，然后将发送的MQTT协议数据包的类型打印出来。

第15~17行代码捕获packet<sub>receive</sub>事件，然后将收到的MQTT协议数据包的类型打印出来。

在subscribe\_with\_qos.js中，Client每次连接到Broker都会按照参数指定的QoS重新订阅主题，订阅成功以后才开始捕获接收和发送的数据包，所以Client从连接后到重新订阅前收到的离线消息都不会被打印出来。

我们可以通过node publish\_with\_qos.js--qos=xxx和node subscribe\_with\_qos.js--qos=xxx来运行这两段node.js代码。

接下来，用不同的参数组合来运行这两段node.js代码，看看输出分别是什么。

 **注意** 需要先运行subscribe\_with\_qos.js再运行publish\_with\_qos.js，确保接收到的消息可以打印出来。

## 1. 发布使用QoS0，订阅使用QoS0

运行“node publish\_with\_qos.js--qos=0”输出为：

---

```
send: publish
```

---

运行 “node subscribe\_with\_qos.js--qos=0” 输出为:

---

```
receive: publish
```

---

Publisher到Broker, Broker到Subscriber都是用的QoS0。

## 2. 发布使用QoS1, 订阅使用QoS1

运行 “node publish\_with\_qos.js--qos=1” 输出为:

---

```
send: publish  
receive: puback
```

---

运行 “node subscribe\_with\_qos.js--qos=1” 输出为:

---

```
receive: publish  
send: puback
```

---

Publisher到Broker, Broker到Subscriber都是用的QoS1。

## 3. 发布使用QoS0, 订阅使用QoS1

运行 “node publish\_with\_qos.js--qos=0” 输出为:

---

```
send: publish
```

---

运行 “node subscribe\_with\_qos.js--qos=1” 输出为:

---

```
receive: publish
```

---

这里就有点奇怪了，很明显Broker到Subscriber使用的是QoS0，和Subscriber订阅时指定的QoS不一样。原因我们在4.3.6中会进行详细解释。

#### 4. 发布使用QoS1，订阅使用QoS0

运行 “node publish\_with\_qos.js--qos=1” 输出为:

---

```
send: publish  
receive: puback
```

---

运行 “node subscribe\_with\_qos.js--qos=0” 输出为:

---

```
receive: publish
```

---

和设定的一样，Publisher到Broker使用QoS1，Broker到Subscriber使用QoS0。Publisher使用QoS1发布消息，但是消息到Subscriber却是QoS0。也就是说，Subscriber有可能无法收到消息，这种现象被称为QoS的降级（QoS Degradation）。

## 5. 发布使用QoS2，订阅使用QoS2

运行 “node publish\_with\_qos.js--qos=2” 输出为：

---

```
send: publish
receive: pubrec
send: pubrel
receive: pubcomp
```

---

运行 “node subscribe\_with\_qos.js--qos=2” 输出为：

---

```
receive: publish
send: pubrec
receive: pubrel
send: pubcomp
```

---

可以看到，Publisher到Broker，Broker到Subscriber都是用的QoS2。



### 4.3.6 实际的Subscribe QoS

在上节的代码实践中，我们已经发现了在某些情况下，Broker在实际发送消息到订阅者时使用的QoS和订阅者在订阅主题时指定的QoS不一样。

这里有一个很重要的计算方法：在MQTT协议中，从Broker到Subscriber这段消息传递的实际QoS等级等于Publisher发布消息时指定的QoS等级和Subscriber在订阅时与Broker协商的QoS等级中最小的那一个。

$$\text{Actual Subscribe QoS} = \text{MIN}(\text{Publish QoS}, \text{Subscribe QoS})$$

这也就解释了“publish qos=0, subscribe qos=1”的情况下Subscriber的实际QoS为0，以及“publish qos=1, subscribe qos=0”时出现QoS降级的原因。

同样，如果Publish QoS为1，Subscribe QoS为2，或者Publish QoS为2，Subscribe QoS为1，那么实际Subscribe接收消息的QoS仍然为1。

理解了实际Subscriber QoS的计算方法，你才能更好地设计系统中Publisher和Subscriber使用的QoS。例如，如果你希望Subscriber

至少收到一次Publisher的消息，那么你要确保Publisher和Subscriber都使用不小于1的QoS。

## 4.3.7 QoS的最佳实践

### 1. QoS与会话

如果Client想接收离线消息，就必须在连接到Broker的时候指定使用持久会话（Clean Session=0），这样Broker才会存储Client在离线期间没有确认接收的QoS大于1的消息。

### 2. 如何选择QoS

以下情况可以选择QoS0：

- Client和Broker之间的网络连接非常稳定，例如一个通过有线网络连接到Broker的测试用Client；
- 可以接受丢失部分消息，比如一个传感器以非常短的间隔发布状态数据，那丢失一些数据也可以接受；
- 不需要离线消息。

以下情况可以选择QoS1：

- 应用需要接收所有的消息，而且可以接受并处理重复的消息；

- 无法接受QoS2带来的额外开销，QoS1发送消息的速度比QoS2快很多。

以下情况下可以选择QoS2：

- 应用必须接收所有的消息，而且应用在重复的消息下无法正常工作，同时你可以接受QoS2带来的额外开销。

实际上，QoS1是应用最广泛的QoS等级。QoS1表示发送消息的速度很快，而且能够保证消息的可靠性。虽然使用QoS1等级可能会收到重复的消息，但是在应用程序中处理重复消息，通常并不是件难事。在5.1节中，我们会看到如何在应用程序中对消息进行去重。

本节学习了MQTT协议在3种不同的QoS等级下消息传递的流程，并用代码进行验证。同时，我们也讨论了如何根据实际的应用场景来选择不同的QoS等级。在4.4节，我们将学习MQTT协议的另外两个特性——Retained消息和LWT。

## 4.4 Retained消息和LWT

本节我们将学习MQTT的Retain消息和LWT（Last Will and Testament）。

### 4.4.1 Retained消息

让我们来考虑一下这个场景。你有一个温度传感器，它每3个小时向一个主题发布当前温度。那么问题来了，有一个新的订阅者在它刚刚发布了当前温度之后订阅了这个主题，那么这个订阅端什么时候才能收到温度消息？

没错，和你想的一样，它必须等到3个小时以后，温度传感器再次发布消息的时候才能收到。在这之前，这个新的订阅者对传感器的温度数据一无所知。

那该怎么解决这个问题呢？

这时候就轮到Retained消息出场解决这个问题了。Retained消息是指在PUBLISH数据包中将Retain标识设为1的消息，Broker收到这样的PUBLISH数据包以后，将为该主题保存这个消息，当一个新的订阅者订阅该主题时，Broker会马上将这个消息发送给订阅者。

Retain消息有如下特点：

- 一个Topic只能有一条Retained消息，发布新的Retained消息将覆盖旧的Retained消息；

- 如果订阅者使用通配符订阅主题，那他会收到所有匹配主题的Retained消息；

- 只有新的订阅者才会收到Retained消息，如果订阅者重复订阅一个主题，那么在每次订阅的时候都会被当作新的订阅者，然后收到Retained消息。

当Retained消息发送到订阅者时，PUBLISH数据包中Retain的标识仍然是1。订阅者可以判断这个消息是否是Retained消息，以做出相应的处理。



**注意** Retained消息和持久会话没有任何关系。Retained消息是Broker为每一个主题单独存储的，而持久会话是Broker为每一个Client单独存储的。

如果你想删除某个主题的Retained消息，只要向这个主题发布一个Payload长度为0的Retained消息就可以了。

那么，开头我们提到的那个场景的解决方案就很简单了，温度传感器每3个小时向相应的主题发布包含当前温度的Retained消息，那么无论新的订阅者什么时候订阅这个主题，他都能收到温度传感器上一次发布的数据。

## 4.4.2 代码实践：发布和接收Retained消息

下面编写一个发布Retained消息的发布端，和一个接收消息的订阅端，订阅端在接收消息的时候将消息的Retain标识和内容打印出来。

发布端的代码publish\_retained.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_publisher_1",
4.   clean: false
5. })
6.
7. client.on('connect', function (connack) {
8.   if(connack.returnCode == 0){
9.     client.publish("home/2ndfloor/201/temperature",
JSON.stringify({current:
10.      25})), {qos: 0, retain: 1}, function (err) {
11.       if(err == undefined) {
12.         console.log("Publish finished")
13.         client.end()
14.       }else{
15.         console.log("Publish failed")
16.       }
17.     })
18.   }else{
19.     console.log('Connection failed:
${connack.returnCode}')
20.   }
```

---



第9行代码在发布时指定Retain标识为1。

订阅端的代码subscribe\_retained.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_subscriber_id_chapter_8",
4.   clean: false
5. })
6.
7. client.on('connect', function (connack) {
8.   if(connack.returnCode == 0) {
9.     if (connack.sessionPresent == false) {
10.      console.log("subscribing")
11.
12.      client.subscribe("home/2ndfloor/201/temperature", {
13.        qos: 0
14.      }, function (err, granted) {
15.        if (err != undefined) {
16.          console.log("subscribe failed")
17.        } else {
18.          console.log('subscribe succeeded with
19.          ${granted[0].topic}, qos:
20.          ${granted[0].qos}')
21.        }
22.      })
23.    }
24.  } else {
25.    console.log('Connection failed:
26.    ${connack.returnCode}')
27.  }
28. })
29.
30. client.on("message", function (_, message, packet) {
31.   var jsonPayload = JSON.parse(message.toString())
32.   console.log('retained: ${packet.retain},
33.   temperature: ${jsonPayload.
34.   current}')
```

---

第9行代码判断了CONNACK数据包中的Session Present标识，只有在第一次建立会话的时候才进行订阅，重复多次运行订阅端代码也只会触发一次订阅。

第28行代码是在收到消息的时候打印出消息的Retain标识。

我们首先运行“node publish\_retained.js”，再运行“node subscribe\_retained.js”，会得到如下输出：

---

```
retained: true, temperature: 25
```

---

当订阅端第一次订阅该主题的时候，Broker会将为该主题保存的Retained消息转发给订阅端，所以在Publisher发布之后订阅者再订阅主题也能收到Retained消息。

然后我们再运行一次“node publish\_retained.js”，在运行subscribe\_retained.js的终端会有如下输出：

---

```
retained: false, temperature: 25
```

---

由于此时订阅端已经订阅了该主题，Broker收到Retained消息以后，只保存该消息，然后按照正常的转发逻辑转发给订阅端，因此对

于订阅端来说，这只是一个普通的MQTT协议消息，所以Retain标识为0。

接着点击“Ctrl+C”，关闭subscribe\_retained.js，重新运行，此时因为Session已经存在，订阅端不会再重新订阅这个主题，终端不会有任何输出。由此可见，Retained消息只对新订阅的订阅者有效。

### 4.4.3 LWT

LWT全称为Last Will and Testament，也就是我们在连接Broker时提到的遗愿，包括遗愿主题、遗愿QoS、遗愿消息等。

顾名思义，当Broker检测到Client非正常地断开连接时，就会向Client的遗愿主题中发布一条消息。遗愿的相关设置是在建立连接时，在CONNECT数据包里面指定的。

- Will Flag：是否使用LWT。
- Will Topic：遗愿主题名，不可使用通配符。
- Will QoS：发布遗愿消息时使用的QoS等级。
- Will Retain：遗愿消息的Retain标识。
- Will Message：遗愿消息内容。

Broker在以下情况下认为Client是非正常断开连接的：

- 1) Broker检测到底层I/O异常；
- 2) Client未能在Keepalive的间隔内和Broker之间进行消息交互；

3) Client在关闭底层TCP连接前没有发送DISCONNECT数据包;

4) Broker因为协议错误关闭了和Client的连接, 比如Client发送了一个格式错误的MQTT协议数据包。

如果Client通过发布DISCONNECT数据包断开连接, 这属于正常断开连接, 不会触发LWT的机制。同时, Broker还会丢掉这个Client在连接时指定的LWT参数。

通常, 如果我们关心设备, 比如传感器的连接状态, 则可以使用LWT。在接下来的代码实践中, 我们会使用LWT和Retained消息实现对一个Client的连接状态监控。

#### 4.4.4 代码实践：监控Client连接状态

实现Client连接状态监控的原理很简单：

1) Client在连接时指定Will Topic为“client/status”，遗嘱消息为“offline”，Will Retain=1；

2) Client在连接成功后向同一个主题“client/status”发布一个内容为“online”的Retained消息。

那么，订阅者在任何时候订阅“client/status”，都会获取Client当前的连接状态。

Client.js的代码如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_publisher_chapter_8",
4.   clean: false,
5.   will:{
6.     topic : 'client/status',
7.     qos: 1,
8.     retain: true,
9.     payload: JSON.stringify({status: 'offline'})
10.  }
11. })
12.
13. client.on('connect', function (connack) {
14.   if(connack.returnCode == 0){
```

```
15.     client.publish("client/status",
JSON.stringify({status: 'online'}),
    {qos: 1, retain: 1})
16.   }else{
17.     console.log('Connection failed:
${connack.returnCode}')
18.   }
19. })
```

---

代码的第5~9行对Client的LWT进行了设置。

在第15行，Client在连接到Broker之后会向指定的主题发布一条消息。

用于监控Client连接状态的monitor.js代码如下所示。

---

```
1. var mqtt = require('mqtt')
2. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
3.   clientId: "mqtt_sample_subscriber_id_chapter_8_2",
4.   clean: false
5. })
6.
7. client.on('connect', function () {
8.   client.subscribe("client/status", {qos: 1})
9. })
10.
11. client.on("message", function (_, message) {
12.   var jsonPayload = JSON.parse(message.toString())
13.   console.log('client is ${jsonPayload.status}')
14. })
```

---

首先运行“node client.js”，然后运行“node monitor.js”，我们会得到以下输出：

---

```
client is online
```

---

在运行client.js的终端上，点击“Ctrl+C”终止client.js，之后在运行monitor.js的终端上会得到以下输出：

```
client is offline
```

---

重新运行“node client.js”，在运行monitor.js的终端上会得到以下输出：

```
client is online
```

---

点击“Ctrl+C”终止monitor.js，然后重新运行“node monitor.js”，会得到以下输出：

```
client is online
```

---

这样，我们就完美地监控了Client的连接状态。

本节我们学习了Retained消息和LWT，并利用这两个特性完成了对Client连接状态进行监控。接下来，我们将学习Keepalive和在移动端的连接保活。



## 4.5 Keepalive与连接保活

在生产环境下，特别是物联网这种无人值守的设备比较多的情况下，我们都希望设备能够自动从错误中恢复过来，比如在网络故障恢复以后，设备能够自动重新连接Broker。本节我们就来学习MQTT协议的Keepalive机制，以及连接保活的方式。

### 4.5.1 Keepalive

在4.4节中，我们提到Broker需要知道Client是否非正常地断开了和它的连接，以发送遗愿消息。实际上，Client也需要能够很快地检测到它和Broker的连接断开，以便重新连接。

MQTT协议是基于TCP协议的一个应用层协议，理论上TCP协议在连接断开时会通知上层应用，但是TCP协议有一个半打开连接的问题（Half-open Connection）。这里不会深入分析TCP协议，需要记住的是，在这种状态下，一端的TCP协议连接已经失效，但是另外一端并不知情，它认为连接依然是打开的，需要很长时间才能感知到对端连接已经断开，这种情况在使用移动网络或者卫星网络的时候尤为常见。

仅仅依赖TCP层的连接状态监测是不够的，于是MQTT协议设计了一套Keepalive机制。回忆一下，在建立连接的时候，我们可以传递一个Keepalive参数，它的单位为秒。MQTT协议约定：在 $1.5 \times \text{Keepalive}$ 的时间间隔内，如果Broker没有收到来自Client的任何数据包，那么Broker认为它和Client之间的连接已经断开；同样，在这段时间间隔内，如果Client没有收到来自Broker的任何数据包，那么Client也认为它和Broker之间的连接已经断开。

MQTT协议中设计了一对PINGREQ/PINGRESP数据包，当Broker和Client之间没有任何数据包传输时，我们可以通过PINGREQ/PINGRESP数据包满足Keepalive的约定和连接状态的侦测。

1. PINGREQ数据包

当Client在一个Keepalive时间间隔内没有向Broker发送任何数据包，比如PUBLISH数据包和SUBSCRIBE数据包时，它应该向Broker发送PINGREQ数据包，PINGREQ数据包的格式如下所示。

(1) 固定头

PINGREQ数据包的固定头如图4-39所示。

Bit	7	6	5	4	3	2	1	0
字节 1	12 (PINGREQ)				保留			
字节 2	0 (数据包剩余长度)							

图4-39 PINGREQ数据包的固定头

固定头中的MQTT协议数据包类型字段的值为12，代表该数据包是PINGREQ数据包。PINGREQ数据包的剩余长度字段值固定为0。

(2) 可变头

PINGREQ数据包没有可变头。

(3) 消息体

PINGREQ数据包没有消息体。

2. PINGRESP数据包

当Broker收到来自Client的PINGREQ数据包时，它应该回复Client一个PINGRESP数据包，PINGRESP数据包的格式如下所示。

(1) 固定头

PINGRESP数据包的固定头格式如图4-40所示。

Bit	7	6	5	4	3	2	1	0
字节1	13 (PINGRESP)				保留			
字节2	0 (数据包剩余长度)							

图4-40 PINGRESP数据包的固定头

固定头中的MQTT协议数据包类型字段的值为13，代表该数据包是PINGRESP数据包。PINGRESP数据包的剩余长度字段值固定为0。

(2) 可变头

PINGRESP数据包没有可变头。

### (3) 消息体

PINGRESP数据包没有消息体。

## 3. Keepalive的其他特性

Keepalive机制还有以下几点需要注意：

1) 如果在一个Keepalive时间间隔内，Client和Broker有过数据包传输，比如PUBLISH数据包，那Client就没有必要再使用PINGREQ数据包了，在网络资源比较紧张的情况下这点很重要；

2) Keepalive的值是由Client指定的，不同的Client可以指定不同的值；

3) Keepalive的最大值为18个小时12分15秒；

4) Keepalive的值如果设为0的话，代表不使用Keepalive机制。

## 4.5.2 代码实践

首先我们编写一段简单的Client代码，它会把发送和接收到的MQTT协议数据包类别打印出来。

完整的代码Keepalive.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var dateTime = require('node-datetime');
3. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
4.   clientId: "mqtt_sample_id_chapter_9",
5.   clean: false,
6.   Keepalive: 5
7. })
8.
9. client.on('connect', function () {
10.   client.on('packetsend', function (packet) {
11.
12.     console.log('${dateTime.create().format('H:M:S')}: send
13.     ${packet.cmd}')
14.   })
15.   client.on('packetreceive', function (packet) {
16.
17.     console.log('${dateTime.create().format('H:M:S')}:
18.     receive ${packet.cmd}')
19.   })
20. })
```

---

代码第6行把Keepalive的值设为5秒。

运行“node Keepalive.js”，我们会得到以下输出：

---

```
19:42:44: send pingreq
19:42:44: receive pingresp
19:42:49: send pingreq
19:42:49: receive pingresp
19:42:54: send pingreq
19:42:54: receive pingresp
.....
```

---

可以看到，每隔5秒就会有一个PINGREQ/PINGRESP数据包的交互。

然后再编写一段Client代码，这个Client每隔4秒发布一条消息，完整的代码Keepalive\_with\_publish.js如下所示。

---

```
1. var mqtt = require('mqtt')
2. var dateTime = require('node-datetime');
3. var client = mqtt.connect('mqtt://mqtt.eclipse.org',
{
4.   clientId: "mqtt_sample_id_chapter_9",
5.   clean: false,
6.   Keepalive: 5
7. })
8.
9. client.on('connect', function () {
10.   client.on('packetsend', function (packet) {
11.     console.log('${dateTime.create().format('H:M:S')}: send
${packet.cmd}')
12.   })
13.
14.   client.on('packetreceive', function (packet) {
15.     console.log('${dateTime.create().format('H:M:S')}:
receive ${packet.cmd}')
16.   })
17.
18.   setInterval(function () {
19.     client.publish("foo/bar", "test")
```

```
20.    }, 4 * 1000)  
21.  })
```

---

代码的第6行把Keepalive的值设为5秒。

代码的第18~20行，设置了定时器，每隔4秒做一个publish。

运行“node Keepalive\_with\_publish.js”，会得到以下输出：

---

```
19:54:37: send publish  
19:54:41: send publish  
19:54:45: send publish  
.....
```

---

正如之前所讲的那样，如果在一个Keepalive的时间间隔内，Client和Broker之间传输过数据包，那么就不会触发PINGREQ/PINGRESP数据包。



### 4.5.3 连接保活

Client的连接保活逻辑很简单，在检测到连接断开时再重新进行连接就可以了。大多数语言的MQTT Client都支持这个功能，并默认打开。不过如果是移动设备，比如在Android系统或者iOS系统的智能手机上使用MQTT Client，那情况就有所不同了。通常在移动端使用MQTT协议的时候会碰到一个问题：App被切入后台后，怎样才能保持与MQTT协议的连接并继续接收消息？接下来，我们就通过Android系统和iOS系统分别来讲一下。

#### 1. Android系统上的连接保活方式

在Android系统上，我们可以在一个Service中创建和保持MQTT协议连接，这样即使App被切入后台，这个Service还在运行，MQTT协议的连接还存在，就能接收消息。参考代码如下所示。

---

```
1. public class MQTTService extends Service{
2.     .....
3.     @Override
4.     public int onStartCommand(Intent intent, int flags,
int startId) {
5.         .....
6.         mqttClient.connect(...)
7.         .....
8.     }
9.     .....
10. }
```

---

接收到MQTT消息后，我们可以通过一些方式，比如广播通知App处理这些消息。

## 2. iOS系统上的连接保活方式

iOS系统的连接保活机制与Android系统的不同，在App被切入后台时，你没有办法在后台运行App的任何代码，所以无法通过MQTT协议的连接来获取消息。（当然，iOS系统提供了几种可以后台运行的方式，比如Download、Audio等，但如果你的App假借这些方式运行后台程序，是过不了审核的，所以这里只讨论正常情况）。

在iOS系统中的App切入后台后，正确接收MQTT协议消息的方式是：

- 1) Publisher发布一条或多条消息；
- 2) Publisher通过某种渠道（比如HTTP API）告知App的应用服务器，然后服务器通过苹果的APNs向对应的iOS订阅者推送一条消息；
- 3) 用户点击推送，App进入前台；
- 4) App重新建立和Broker的连接；
- 5) App收到Publisher刚刚发送的一条或多条消息。

App端的代码如下所示。

---

```
1. -(void)application:(UIApplication *)app
   didReceiveRemoteNotification:(NSDictionary *)userInfo {
2.     if([app applicationState] ==
   UIApplicationStateInactive) {
3.         [mqttClient connect]
4.     }
5. }...
```

---



**注意**实际上，当下国内主流的Android系统都有后台清理功能，App被切入后台后，它的服务，即使是前台服务（Foreground Service）也会很快地被杀掉，除非App被厂商或者用户加入白名单。所以在Android系统上最好还是利用厂商的推送通道，比如华为推送、小米推送等，即在App被切入后台时采用和iOS系统上一样的机制来接收MQTT协议的消息。

本节学习了MQTT协议的Keepalive机制，并了解了如何在移动端保持与MQTT协议的连接。到此为止，MQTT 3.1.1版本的所有特性就已经介绍完了，在4.6节中，我将讲解MQTT 5.0版本的一些新特性。

## 4.6 MQTT 5.0的新特性

在前面的章节里，我们使用的是MQTT 3.1.1版本，也是目前支持和使用最广泛的版本。2017年8月，OASIS MQTT Technical Committee正式发布了用于Public Review的MQTT 5.0草案。2018年，MQTT 5.0已正式发布，虽然目前支持MQTT 5.0的Broker和Client库还比较有限，但是作为MQTT未来的发展方向，我认为了解5.0的新特性还是很有必要的，也许看完本节的内容你马上就想迁移到MQTT 5.0了呢！

MQTT 5.0在MQTT 3.1.1的基础上做了很多改变，同时也不是向下兼容的。这里我挑了几个个人认为比较实用的新特性进行介绍。这些新特性能够解决在MQTT 3.1.1版本中较难处理的一些问题，例如。

- 用户属性 (User Properties)
- 共享订阅 (Shared Subscriptions)
- 消息过期 (Publication Expiry Interval)
- 重复主题
- Broker能力查询
- 双向DISCONNECT

作为MQTT 3.1.1的后续版本，为什么版本号直接变成了5.0呢？因为MQTT 3.1.1版本指定在连接的时候Protocol Version为4，所以后续版本只能使用5。

### 4.6.1 用户属性

MQTT 5.0中可以在PUBLISH、CONNECT和带有Return Code的数据包中夹带一个或多个用户属性（User Properties）数据。

- 在PUBLISH数据包中携带的用户属性由发送方的应用定义，随消息被Broker转发到消息的订阅方。

- CONNECT数据包和ACKs消息中也可以携带发送者自定义的用户属性数据。

在实际的项目中，我们除了关心收到的消息内容，往往也想知道这个消息来自谁。例如：ClientA收到ClientB发布的消息后，ClientA想给ClientB发送一个回复，这时ClientA必须知道ClientB订阅的主题才能将消息传递给ClientB。在MQTT 3.1.1中，我们通常是在消息数据中包含发布方的信息，比如它订阅的主题等。MQTT 5.0以后就可以把这些信息放在User Properties中了。

## 4.6.2 共享订阅

在MQTT 3.1.1和之前的版本里，订阅同一主题的订阅者都会收到来自这个主题的所有消息。例如你需要处理一个传感器数据，假设这个传感器上传的数据量非常大且频率很高，你没有办法启动多个Client分担处理该工作，则可以启动一个Client来接收传感器的数据，并将这些数据分配给后面的多个Worker处理。这个用于接收数据的Client就会是系统的瓶颈和单点故障之一。

通常，我们可以通过主题分片。比如，让传感器依次发布到/topic1……/topicN来变通地解决这个问题，但这仅仅解决了部分问题，同时也提高了系统的复杂度。

而在MQTT 5.0里面，MQTT可以实现Producer/Consumer模式了。多个Client（Consumer）可以一起订阅一个共享主题（Producer），来自这个主题的消息会依次均衡地发布给这些Client，实现订阅者的负载均衡。

这个功能在传统的队列系统，比如RabbitMQ里很常见。如果你不想升级到MQTT 5.0，其实EMQ X Broker在MQTT 3.1.1上已经支持这个功能了。

### 4.6.3 消息过期

假设你设计了一个基于MQTT协议的共享单车平台，用户通过平台下发一条开锁指令给一辆单车，但是不巧的是，单车的网络信号（比如GSM）这时恰好断了，用户摇了摇头走开去找其他单车了。过了2小时以后，单车的网络恢复了，它收到了2小时前的开锁指令，此时该怎么做？

为了处理这种情况，在MQTT 3.1.1和之前的版本中，我们往往是在消息数据里带一个消息过期（Publication Expiry Interval）时间，在接收端判断消息是否过期，这要求设备端的时间和服务端的时间保持一致。但对于一些电量不是很充足的设备，一旦断电，之后再启动，时间就会变得不准确，这样就会导致异常的出现。

MQTT 5.0版本考虑到了这个问题，其直接包含了消息过期功能，在发布的时候可以指定这个消息在多久后过期，这样Broker不会将已过期的离线消息发送到Client。



#### 4.6.4 重复主题

在MQTT 3.1.1和之前的版本里，PUBLISH数据包每次都需要带上发布的主题名，即便每次发布的都是同一个主题。

在MQTT 5.0中，如果你将一条PUBLISH的主题名设为长度为0的字符串，那么Broker会使用你上一次发布的主题。这样降低了多次发布到同一主题（往往都是这样）的额外开销，对网络资源和处理资源都有限的系统非常有用。

### 4.6.5 Broker能力查询

在MQTT 5.0中，CONNACK数据包包含了一些预定义的头部数据，用于标识Broker支持哪些MQTT协议功能，如表4-3所示。

表4-3 CONNACK数据包包含的预定义的头部数据

Pre-defined Header	数据类型	描述
Retain Available	Boolean	是否支持 Retained 消息
Maximum QoS	Number	Client 可以用于订阅和发布的最大 QoS
Wildcard available	Boolean	订阅时是否可以使用通配符主题
Subscription identifiers available	Boolean	是否支持 Subscription Identifier (MQTT 5.0 特性)
Shared Subscriptions available	Boolean	是否支持共享订阅
Maximum Message Size	Number	可发送的最大消息长度
Server Keepalive	Number	Broker 支持的最大 Keepalive 值

Client在连接之后就可以知道Broker是否支持自己要用到的功能，这对一些通用的MQTT设备生产商或者Client库的开发者很有用。

#### 4.6.6 双向DISCONNECT

在MQTT 3.1.1或之前的版本中，Client只有在主动断开时会向Broker发送DISCONNECT数据包。如果因为某种错误，Broker要断开和Client的连接，它只能直接断开底层TCP连接，而Client并不会知道自己连接断开的原因，也无法解决错误，只是简单地重新连接、被断开、重新连接……

在MQTT 5.0中，Broker在主动断开和Client的连接时也会发送DISCONNECT数据包。同时，从Client到Broker，以及从Broker到Client的CONNCT数据包中都会包含一个Reason Code，用于标识断开的原因，如表4-4所示。

表4-4 Reason Code的含义

Reason code	发送方	描述
0	Client 或 Broker	正常断开连接，不发送遗嘱消息
4	Client	正常断开连接，但是要求 Broker 发送遗嘱消息
129	Client 或 Broker	MQTT 数据包格式错误
135	Broker	请求未授权
143	Broker	主题过滤器格式正确，但是 Broker 不接收
144	Client 或 Broker	主题名格式正确，但是 Client 或者 Broker 不接收
153	Client 或者 Broker	消息体格式不正确
154	Broker	不支持 Retained 消息
155	Broker	QoS 等级不支持
158	Broker	不支持共享订阅
162	Broker	订阅时不支持通配符主题名

上面列举的是我认为能够解决MQTT 3.1.1中一些难题的新特性，如果你不想升级到MQTT 5.0，也不用担心，我们仍然可以使用上面提到的一些Workaround来解决这些问题。

## 4.7 本章小结

至此，MQTT协议的主要特性就介绍完了，所有的代码可以在<https://github.com/sufish/mqtt-sample>中找到。

在第5章里，我们会用一个“AI+IoT”的实战项目来演示如何在实际的项目中使用MQTT协议，并讲解一些MQTT协议在实际应用中的最佳实践。

## 第5章 MQTT协议实战

在本书的开头，我曾经提到过一个“AI+IoT”的应用场景，一个可以识别出人和车辆的交通探头，本章我们就来实现一个类似的功能。

当然，本书的内容重点不在AI上，所以我们会把AI端设计得尽量简单。

我们将实现一个基于Android系统的App，这个App可以识别出照片中的物体，并将照片和识别结果使用MQTT协议上传。同时，我们会实现一个基于Web的用户端，可以实时地看到App上传的照片和识别结果。

实际上，运行Android系统的物联网设备已经很常见了，有兴趣的读者可以了解一下Google的Android Things项目。

## 5.1 “AI+IoT” 项目实战

### 5.1.1 用TensorFlow在Android系统上进行物体识别

TensorFlow是Google推出的深度学习框架，它有一个可以在移动设备上使用的版本——TensorFlow Mobile，在我们的实战项目中可以使用TensorFlow Android版本。

TensorFlow Android是一个可以用训练好的网络模型进行推理的TensorFlow，为了能够识别图片中的物体，通常我们需要使用大量的标记好的照片作为训练数据，将其输入神经网络才能训练出满足我们需求的网络模型，不过在这里我们可以跳过这一步，Google开源了许多预先训练好的模型，其中就包括了可以从照片中识别出各种物体的模型，我们可以直接拿来使用。

最终这个Android App可以达到如图5-1所示的效果。从相册中选取一张照片，程序会识别出图片中的物体。在图5-1中，程序能识别出多个人物和沙发，并用方框标识出物体的位置，同时在方框的左上角标注出了物体的名称。

未插卡

18:59

obdemo

选取图片

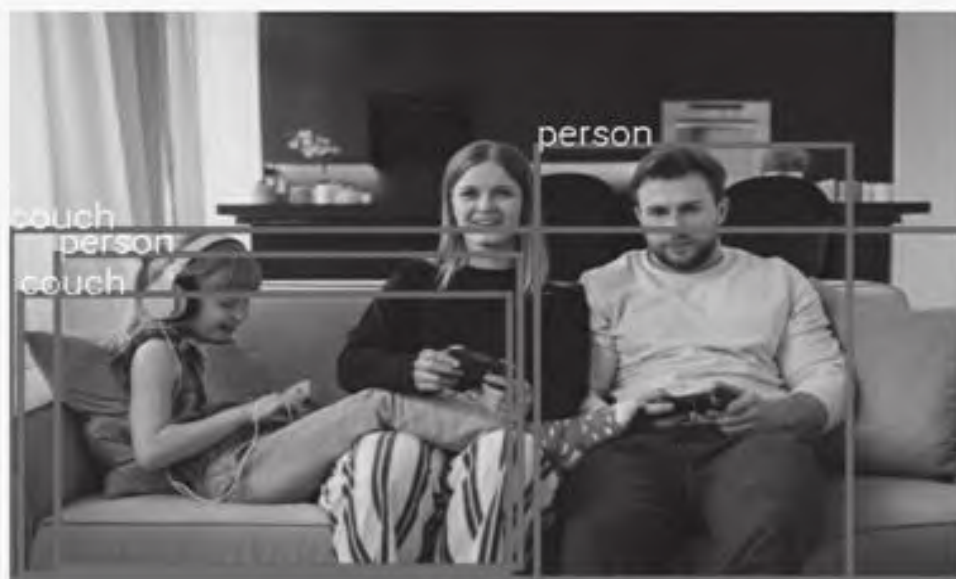




图5-1 物体识别效果

你可以在[https://github.com/sufish/object\\_detection\\_mqtt](https://github.com/sufish/object_detection_mqtt)找到这个App的全部代码。

## 5.1.2 如何在MQTT协议里传输大文件

能从照片中识别出物体的Android App需要将照片和识别结果通过MQTT协议发布出去，那么怎么使用MQTT协议传输类似图片这样的大文件呢？

我们前面提到过，一个MQTT协议数据包的消息体最大可以达到约256MB，所以对于传输图片的需求，最简单、直接的方式就是把图片数据直接包含在PUBLISH数据包里进行传输。这要求图片的大小不超过256MB，如果超过256MB，还需要对图片进行拆分和组装。

除此之外，还有一种做法：在发布数据前，先把图片上传到云端的某个图片存储里，使得PUBLISH数据包中只包含图片的URL，当订阅端接收这个数据后，再通过图片的URL来获取图片。这种做法较前面的做法有如下几个优点。

- 对订阅端来说，它可以在有需要的时候再下载图片数据，而第一种做法，每次都必须接收图片的全部数据。

- 这种做法可以处理大于256MB的文件，而第一种做法必须把文件分割为多个PUBLISH数据包，订阅端接收后再重新组合，非常烦琐。

- 节省带宽。如果图片数据直接放在PUBLISH数据包中，那么Broker就需要预留相对大的带宽。目前国内，带宽还是比较贵的。如果PUBLISH数据包中只包含URL，每一个PUBLISH数据包都很小，那么Broker的带宽需求就小多了。虽然上传图片也需要带宽，但是如果你使用云存储，比如阿里云OSS、七牛等，从它们那里购买上传和下载图片的带宽要便宜很多。同时，这些云存储服务商建设了很多CDN，通常上传和下载图片比直接通过PUBLISH数据包传输要快一些。

- 节约存储和处理能力。因为Broker需要存储Client未接收的消息，所以如果图片包含在PUBLISH数据包里面，Broker需要预留相当大的存储空间。如果用云存储，存储的成本比自建要便宜得多。

在大多数情况下，使用MQTT协议传输大文件我都建议采用后一种方式。在这个项目中，我们也采用这种方式传输照片。这里我们选择七牛作为文件的云存储服务商。

### 5.1.3 消息去重

在4.3节中我们提到，在MQTT协议的3种QoS等级中，QoS1是运用最广泛的QoS等级，因为它在保证消息可靠性的前提下，额外开销较QoS2少得多，性价比更高。

不过QoS1有一个问题，就是可能会收到重复的消息，所以需要在应用中手动对消息进行去重。

我们可以在消息数据中携带一个唯一的消息ID，通常是UUID。订阅端需要保存已接收消息的ID，当收到新消息的时候，通过消息的ID来判断是否是重复的消息，如果是，则丢弃。

### 5.1.4 最终的消息数据格式

综合前文所讲的两点，项目中的消息数据最终用如下格式进行编码：

```
{'id':<消息ID>, timestamp:<UNIX时间戳>, image_url:<图片>, objects:[图片中物体名称的数组]}
```

我们使用JSON格式对要传输的数据进行编码，其中id字段为一个唯一的消息ID，用这个字段进行消息去重，image\_url字段为照片上传到云存储之后的url。

## 5.1.5 代码实践：上传识别结果

### 1. 连接到Broker

首先在项目中引入Java的MQTT Client库，在build.gradle文件中加入以下代码。

---

```
1. repositories {
2.     maven {
3.         url
4.         "https://repo.eclipse.org/content/repositories/paho-
5.         snapshots/"
6.     }
7. }
8. dependencies {
9.     compile
10.    'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.0'
```

---

然后在App启动时连接到Broker。

---

```
1. String clientId = "client_" +
2. Settings.Secure.getString(getApplicationCont
3. ext().getContentResolver(), Settings.Secure.ANDROID_ID);
4. MqttAsyncClient = new
5. MqttAsyncClient("tcp://mqtt.eclipse.org:1883",
6. clientId,
7. new
8. MqttDefaultFilePersistence(getApplicationContext().
9. getApplicationInfo().dataDir));
10. mqttAsyncClient.connect(null, new
```

---

```

IMqttActionListener() {
5.         @Override
6.         public void onSuccess(IMqttToken
asyncActionToken) {
7.             runOnUiThread(new Runnable() {
8.                 @Override
9.                 public void run() {
10.                    Toast.makeText(getApplicationContext(),
"已连接到 Broker",
    Toast.LENGTH_LONG).show();
11.                }
12.            });
13.        }
14.
15.        @Override
16.        public void onFailure(IMqttToken
asyncActionToken, final Throwable
exception) {
17.            runOnUiThread(new Runnable() {
18.                @Override
19.                public void run() {
20.                    Toast.makeText(getApplicationContext(),
"连接 Broker 失败:" +
    exception.getMessage(), Toast.LENGTH_LONG).show();
21.                }
22.            });
23.        }
24.    });

```

---

第1行代码使用Android ID作为Client Identifier的一部分，这样可以保证在不同的终端上运行Client Identifier时不会冲突。

然后通过回调来获取连接成功或失败的事件，并在App界面上做出相应的展示。

## 2. 发布识别结果到对应主题

在发布识别结果之前，需要先将照片上传到七牛云存储，首先把七牛云存储的SDK引入到项目中，在build.gradle文件中加入以下代码。

---

```
1. compile 'com.qiniu:qiniu-android-sdk:7.3.+'
```

---

然后上传照片，在照片上传成功后，发布消息到相应的主题。

---

```
1. uploadManager.put(getBytesFromBitmap(image), null,
upToken, new
  UpCompletionHandler() {
2.      @Override
3.      public void complete(String key, ResponseInfo
info, JSONObject
response) {
4.          if(info.isOK()){
5.              JSONObject jsonMessage = new JSONObject();
6.              jsonMessage.put("id", randomUUID());
7.              jsonMessage.put("timestamp", timestamp);
8.              jsonMessage.put("objects", objects);
9.              jsonMessage.put("image_url", "http://" +
QINIU_DOMAIN + "\\\" +
response.getString("key"));
10. mqttAsyncClient.publish("front_door/detection/objects",
new
  MqttMessage(jsonMessage.toString().getBytes()));
11.      }
12.  }
13. }, null);
```

---

在代码的第1行开始上传图片，在上传图片成功的回调里，即代码的第5~9行，按照我们之前提到的消息格式进行组装。然后在第10



行，将消息发布到主题“front\_door/detection/objects”。

这样App发布识别结果的功能就基本完成了。

### 5.1.6 在浏览器中运行MQTT Client

接下来实现基于Web的用户端，接收Android App发来的照片和识别结果。

用户端实际上是运行在浏览器里的JavaScript程序，那么，如何在浏览器中运行MQTT Client并建立MQTT协议连接呢？

在目前主流的浏览器里，使用JavaScript直接打开一个TCP连接是不可能的（Socket API可以解决这个问题，但是浏览器对Socket API的支持还非常有限）。不过我们可以通过使用WebSocket的方式在浏览器里面使用MQTT协议，这种技术叫作MQTT over WebSocket，MQTT over WebSocket的实现原理是把MQTT协议数据包封装在WebSocket帧中进行发送，大多数的浏览器都支持WebSocket，如图5-2所示。

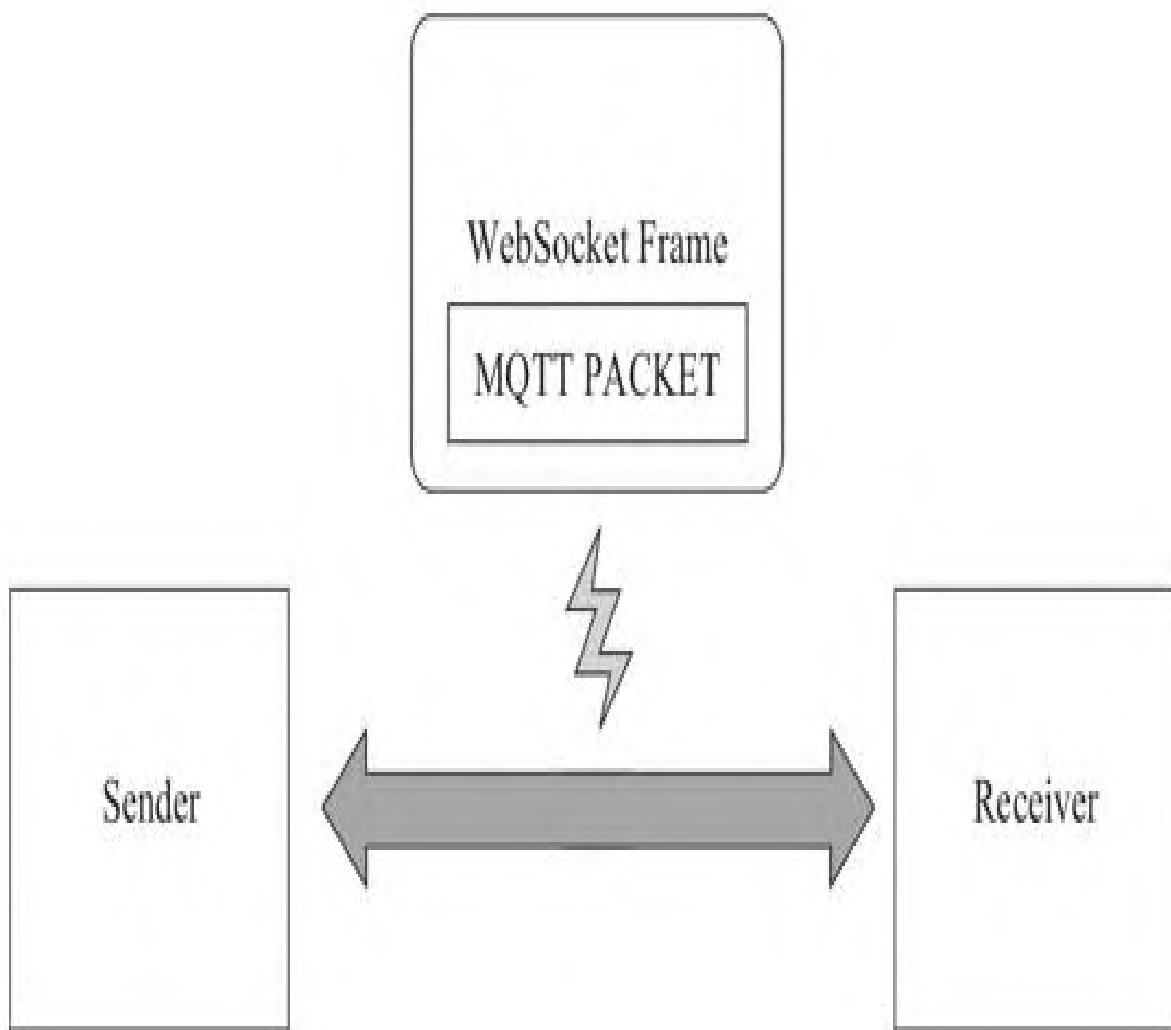


图5-2 使用WebSocket发送MQTT协议数据包

MQTT over WebSocket也需要Broker支持，不过目前大部分Broker都是支持的，包括我们现在使用的公共Broker。

## 5.1.7 代码实践：接收识别结果

### 1. 连接到Broker

首先需要在HTML页面中引入实现了MQTT over WebSocket的MQTT Client文件。

---

```
1. <script
  src="https://unpkg.com/mqtt@2.18.6/dist/mqtt.min.js">
</script>
```

---

然后建立到Broker的连接。

---

```
1. var client = mqtt.connect("ws://mqtt.eclipse.org/ws")
```

---

注意，这里Broker的URL中的协议部分变成了“ws”，同时path也变成了“/ws”。

在连接成功后，订阅对应的主题。

---

```
1. client.subscribe("front_door/detection/objects", {
2.     qos: 1
3. }, function (err) {
4.     if (err !== undefined) {
5.         console.log("subscribe failed")
6.     } else {
7.         console.log('subscribe succeeded')
```

---

```
8.         }  
9.     })
```

---

## 2. 处理并展示识别结果

在接收到发布自Android App的消息后，首先要根据消息数据中的ID字段进行去重。

---

```
1. var receivedMessages = new Set();  
2. client.on("message", function (_, payload) {  
3.     var jsonMessage = JSON.parse(payload.toString())  
4.     if(!receivedMessages.has(jsonMessage.id)){  
5.         receivedMessages.add(jsonMessage.id)  
6.         //接下来把结果显示在页面上  
7.     }  
8. })
```

---

这里只是简单地使用一个Set来保存已收到的消息ID。在实际项目中，可以用稍微复杂一点的数据结构，比如用支持Expiration的缓存来存储已收到的消息ID。

最后，把接收到的结果在页面上显示出来（这里使用Table显示）。

---

```
1. var date = new Date(jsonMessage.timestamp * 1000)  
2. $('#results tr:last').after('<tr>  
<td>${date.toLocaleString()}</td><td>  
    ${jsonMessage.objects}</td><td></td></tr>');
```

---

用户端的最终效果如图5-3所示。



图5-3 识别结果展示效果

你可以在[https://github.com/sufish/mqtt\\_browser](https://github.com/sufish/mqtt_browser)上找到全部代码。

## 5.1.8 搭建私有MQTT Broker

到目前为止，我们使用的都是一个公共的Broker，对于学习和演示来说是足够的。但是对于实际生产来说，我们需要一个私有、可控的Broker。

当然，我们可以选择像阿里云、青云这样的云计算服务商提供的MQTT Broker服务，云计算服务商的MQTT Broker服务是一个很好的选择，接入和配置也很简单，你只需要阅读相应的产品文档，照着步骤一步步来就可以了。

如果你因为某种原因无法使用公有云服务，或者你需要可控性、定制性更强的Broker，你也可以选择自行搭建MQTT Broker。

本书使用的Broker是EMQ X，使用它的理由有以下几点。

- 性能和可靠性：EMQ X是用Erlang语言编写的，在电信行业工作的读者可能了解，电信行业里很多核心的应用系统都是用Erlang编写的。

- 纵向扩展能力：在8核32G的主机上，可以容纳超过100万MQTT Client接入。



- 横向扩展能力：支持多机组成集群。

- 基于插件的功能扩展：官方提供了很多扩展插件用于与其他业务系统集成，如果你熟悉Erlang语言，也可以通过插件的方式自行扩展。

- 项目由商业公司开发和维护，并提供商业服务。

实际上，上面提到的青云IoT Hub就是基于EMQ X实现的，我在实际生产中也使用EMQ X很多年了，对它的性能和稳定性是相当认可的。读者朋友可以访问EMQ X的官网了解关于EMQ X的更多信息。

## 1. 安装Erlang runtime

在安装EMQ X之前，需要安装Erlang的Runtime，这里使用源码编译方式进行安装。

- (1) 下载Erlang/OTP 22.0源文件包。

- (2) 解压缩：`tar-xzf otp_src_22.0.tar.gz`。

- (3) `cd otp_src_22.0`。

- (4) `./configure`。

- (5) `make`。

(6) `sudo make install`。

安装完毕后在终端输入`erl`，即可进入Erlang的交互式Shell。

## 2. 安装EMQ X Broker

在EMQ X的官网[emqx.io](http://emqx.io)可以下载对应操作系统的EMQ X Broker二进制包，这里以ubuntu系统为例。

先下载MQTT Broker的二进制文件的zip包，然后解压缩到<EMQ X 安装目录>，完毕之后，在控制台运行：

---

```
<EMQ X安装目录>/emqx/bin/emqx start
```

---

如果命令行输出为“`emqx 3.2.0 is started successfully!`”，那说明EMQ X已经成功安装并运行了。

现在EMQ X Broker以守护进程的方式运行。

关闭Broker需要运行：

---

```
<EMQ X 安装目录>/emqx/bin/emqx stop
```

---

默认情况下，EMQTT Broker的MQTT协议端口是1883，WebSocket的端口是8083。

本书使用的版本为EMQ X Broker V 3.2.0，EMQ X Enterprise是EMQ X的付费版本，注意不要安装错了。

## 5.1.9 传输层安全

到目前为止，本书中的MQTT代码都是使用明文来传输MQTT协议数据包，包括含有username、password的CONNECT数据包。在实际的生产环境中，这样显然是不安全的。

MQTT协议支持传输层加密。在生产环境中，通常需要使用SSL来传输MQTT协议数据包，使用传输层加密的MQTT协议数据包被称为MQTTS（类似于HTTP之于HTTPS）。

我们使用的Public Broker支持MQTTS，你只需要在连接时修改一下Broker URL并将“mqtt”换成“mqtts”就可以了。

---

```
1. var client = mqtt.connect('mqtts://mqtt.eclipse.org')
```

---

EMQ X Broker也支持MQTTS，你可以在<EMQ X安装目录>/emqx/etc/certs下配置你的SSL证书。

EMQ X Broker自带一份自签署的证书，可以开箱即用地在8883端口使用MQTTS。但是因为自签署的证书，所以你需要关闭客户端的证书验证。

---

```
1. var client = mqtt.connect('mqtt://127.0.0.1:8883', {  
2.   rejectUnauthorized: false  
3. })
```

---

代码的第2行是关闭客户端的证书验证。

## 5.2 MQTT常见问题解答

本节整理了一些我经常被问到的与MQTT协议相关的问题和解答。

1) 目前MQTT 5.0会马上普及吗？

暂时不会，目前Broker以及Client实现的支持都还比较有限。

2) MQTT模块如何实现持续的超低功耗连接？

MQTT协议建立的是TCP长连接，所以功耗会高一些，如果满足不了低功耗的要求，还可以选择基于UDP的CoAP协议。

3) 如何正确地理解Retained消息？

Broker收到Retained消息后，会单独保存一份，再向当前的订阅者发送一份普通的消息（Retained标识为0）。当有新订阅者时，Broker会把保存的这条消息发给新订阅者（Retained标识为1）。

4) 怎么能让发送数据的一方快速收到指定设备的回应数据？

只要发送的数据Payload里包含发送方订阅的主题，接收方收到消息之后向这个主题发布一个消息，这样发送方就能收到了。

5) 部署好Broker后，怎么实现Broker与Client的通信？

根据使用的语言选择一种Client的实现就可以了，在相关网站可以找到一些主流语言的Client库。

6) 我的设备已经按照MQTT协议在发送数据，我在服务器部署的是Mosquitto代理，如何设置Mosquitto才能将我的设备数据打印出来？

在服务器端创建一个Subscriber并订阅相应主题，然后打印收到的消息。

7) 如果订阅者重复订阅一个主题，也会被当作新的订阅者。那何时会被当作旧的订阅者？

在下次主动订阅这个主题之前，都会被当作旧的订阅者。

8) 100台以内的设备使用MQTT协议，是自己搭建还是用各种云提供的物联网服务？

看价格，使用云服务一般比自建要便宜。

9) 有哪些开源的比较好的MQTT Broker？

我使用过的有EMQ X和Mosquitto，我推荐EMQ X。

10) MQTT必须在Linux系统上开发吗？

不用，各个OS都有现成的Client实现。

## 5.3 开发物联网应用，学会MQTT协议就够了吗

至此，我们已经详细讲解了MQTT协议的各个特性，并辅以代码和实战项目。我想读者们应该对MQTT协议已经有了相当多的了解。那是不是我们就已经准备好开发物联网应用了呢？

实际上，在我经常被读者们问到的问题中，除了关于MQTT协议本身的内容以及特性相关的问题之外，还有很多问题是关于物联网软件设计和架构方面的，比如：

- 我该如何管理我的设备和设备状态？
- 业务服务端应该怎么接收、处理和存储来自设备的数据？
- 我的设备数量很多，Broker端应该怎么架设来确保性能和可扩展性？
- 我的设备处理能力有限，除了使用MQTT协议外，还有没有其他选择？
- .....

这让我意识到，单单学会MQTT协议离设计一个成熟的物联网产品还有一段不小的距离。其实仔细想想，这也没什么不对的：拿Web开发



做一个类比，我们只学习了HTTP协议，就能够开发一个成熟的网站或者基于Web的服务吗？答案也是否定的。

回想一下我们是怎么学习Web开发的。

首先，我们会了解一下HTTP协议，然后选择一个框架，比如Java的Spring Boot、Python的Django、Ruby的Rails等。这些框架提供固定的模式，对软件进行高度的抽象和分层，并集成一些Web开发的最佳实践。你可以在Model层处理业务逻辑，用ORM来进行数据库操作，在Controller层处理输入、输出和跳转，在View层渲染HTML页面，这样一个网站和Web服务才能很快被开发出来。除了性能优化的时候，你几乎不用去想HTTP协议的细节。

回到物联网开发，抛开设备端的异构性，单说服务端的架构，它并不像Web开发领域有一个为人熟知的模式、架构或开发框架。开发者往往还是需要从协议这一层慢慢往上“搭积木”，学习曲线相对来说还是比较陡的。

## 1. 我的经历

2015年年中，我开始在物联网方向创业，我的第一个决定是先实现一个供业务系统和设备使用的物联网平台。当时阿里云的IoT平台已经上线，由于功能性和定制性方面暂时满足不了我的需求，最后还是决定自行开发。

我们自行开发的物联网平台实现了设备的管理和接入，设备数据的存储和处理，并抽象和封装了基于MQTT协议的数据传输（比如设备的数据上报和服务端的指令下发等），提供了业务服务端使用的服务端API，以及设备端使用的设备端SDK，业务服务器和设备不再需要处理数据传输和接入等方面的细节，它们甚至不知道数据是通过MQTT协议传输的，这一切对业务服务器和设备都是透明的。

这个平台很好地支持了业务服务端和设备端的快速迭代，也支撑着业务从0到1，从1到盈利的飞速发展。同时，我也在密切关注着各大云服务商（比如阿里云、AWS等）提供的IoT平台。在一些功能上，他们与我们的设计思路和实现逻辑是非常相似的，同时我也会把云IoT平台上的新功能或者更好的实践集成到自研的物联网平台上。

## 2. 接下来学什么

在研发物联网平台的过程中，我踩了很多“坑”，同时也积累了一些物联网平台在架构和设计模式等方面的经验。在本书的第三部分，我会把这些物联网平台架构以及设计方面的知识和经验分享出来，这应该可以覆盖物联网开发中80%的场景和大部分的设计和架构问题。

到2019年，阿里云IoT平台的功能已经非常强大，在第三部分中，我们将使用开源组件，从第一行代码开始，一步步地实现一个具有阿

里云IoT平台大部分功能的物联网平台。在这个过程中，我会穿插讲解在物联网开发中可以用到的模式和架构选择——Pros and Cons，以及一些最佳实践等。与前两部分侧重协议内容和理论不同，第三部分包含大量的实战代码，毕竟代码是程序员之间交流的最好语言。

## 5.4 本章小结

本章完成了一个简单的IoT+AI的实战项目，讲解了在实际开发中会遇到的问题：如何使用MQTT来传输大文件，如何对消息进行去重，如何通过WebSocket在浏览器中使用MQTT等。

我们可以看到，只是熟悉MQTT协议，离开发一个成熟的物联网产品还是有一定距离的。所以在第6章，我们将搭建一个物联网平台，进一步讲解物联网架构设计。

## 第三部分 实战：从0搭建一个IoT平台

- 第6章 准备工作台
- 第7章 设备生命周期管理
- 第8章 上行数据处理
- 第9章 下行数据处理
- 第10章 IotHub的高级功能
- 第11章 扩展EMQ X Broker
- 第12章 集成CoAP协议

从这部分开始，我们将一步一步搭建一个可以同时支持公司内部多个异构物联网产品的IoT平台，它包含了类似于阿里云IoT平台的云物联网平台的大部分功能。

我给这个物联网平台取名为Maque（麻雀）IotHub，寓意“麻雀虽小，五脏俱全”，简称IotHub。

## 第6章 准备工作台

在本章里，我们首先要配置好开发环境，安装必要的软件，然后搭建项目的代码框架。

## 6.1 安装需要的组件

首先准备好开发环境并安装IoTHub需要的各个开源组件。

### (1) MongoDB

MongoDB是一个基于分布式文件存储的数据库，我们可以把MongoDB作为物联网平台的主要数据存储工具。

大家可以在软件官网

<https://docs.mongodb.com/manual/installation/#mongodb-community-edition-installation-tutorials>找到MongoDB的安装文档，根据文档在对应的系统上安装和运行MongoDB。

### (2) Redis

Redis是一个高效的内存数据库，物联网平台会使用Redis来作为缓存。

请根据<https://redis.io/download>的文档在对应的系统上安装和运行Redis。

### (3) Node.js

Node.js是一个基于Chrome V8引擎的JavaScript运行环境，我们会使用Node.js来开发IoTHub的主要功能。

请根据<https://nodejs.org/en/download/>的文档在对应的系统上面安装Node.js。

#### (4) RabbitMQ

RabbitMQ是使用Erlang编写的AMQP Broker，IoTHub使用RabbitMQ作为队列系统来实现物联网平台内部以及物联网平台到业务系统的异步通信。

请根据<https://www.rabbitmq.com/download.html>中的说明在对应的系统上安装和运行RabbitMQ。

#### (5) Mosquitto MQTT Client

mosquitto\_sub/mosquitto\_pub是一对命令行的MQTT Client，我们可以用它们做一些简单的测试，mosquitto\_sub/mosquitto\_pub是随着Mosquitto broker一起安装的。

请根据<https://mosquitto.org/download/>中的说明在对应的系统上安装Mosquitto broker。



我们不会运行和使用Mosquitto broker，只使用随Mosquitto broker一起安装的Mosquitto MQTT Client。

## (6) EMQ X

如前面所说，IoT Hub将使用EMQ X实现MQTT/CoAP协议接入，并使用EMQ X的一些高级功能简化和加速开发。

如何安装EMQ X Broker已经在第3章介绍过了。

## 6.2 Maque IotHub的组成部分

安装完开发物联网平台需要的组件后，我们简单介绍一下IotHub的各个组成部分。

- Maque IotHub：我们要开发的物联网平台，简称IotHub。
- Maque IotHub Server API：Maque IotHub的服务端API，以RESTful API的形式将功能提供给外部业务系统调用，简称Server API。
- Maque IotHub Server：Maque IotHub的服务端，包含了Server API和IotHub服务端主要的功能代码，简称IotHub Server。
- Maque IotHub DeviceSDK：Maque IotHub提供的设备端SDK，设备通过调用SDK提供的API接入Maque IotHub，并与业务系统进行交互，简称DeviceSDK。

同时我们再定义如下两个实体。

- 设备应用代码：实现设备具体功能的代码，比如打开灯、在屏幕上显示温度等，它调用Maque IotHub DeviceSDK使用Maque IotHub提供的功能。它是IotHub DeviceSDK的“用户”。

- 业务系统：指实现特定物联网应用服务端的业务逻辑系统，它通过调用Maque IoT Hub Server API的方式控制设备、使用设备上报的数据，Maque IoT Hub为它屏蔽了与设备交互的细节。它是IoT Hub Server API的“用户”。

## 6.3 项目结构

在本书中我们会使用两个NodeJS的项目来进行开发，分别是物联网平台的服务端代码IotHub Server和设备端代码DeviceSDK。

### 6.3.1 IotHub Server

服务端代码以一个Express项目作为开始，Express是一个基于Node.js的轻量级Web开发框架，非常适合开发RESTful API，用来开发IotHub Server API非常方便。项目的代码结构如图6-1所示。

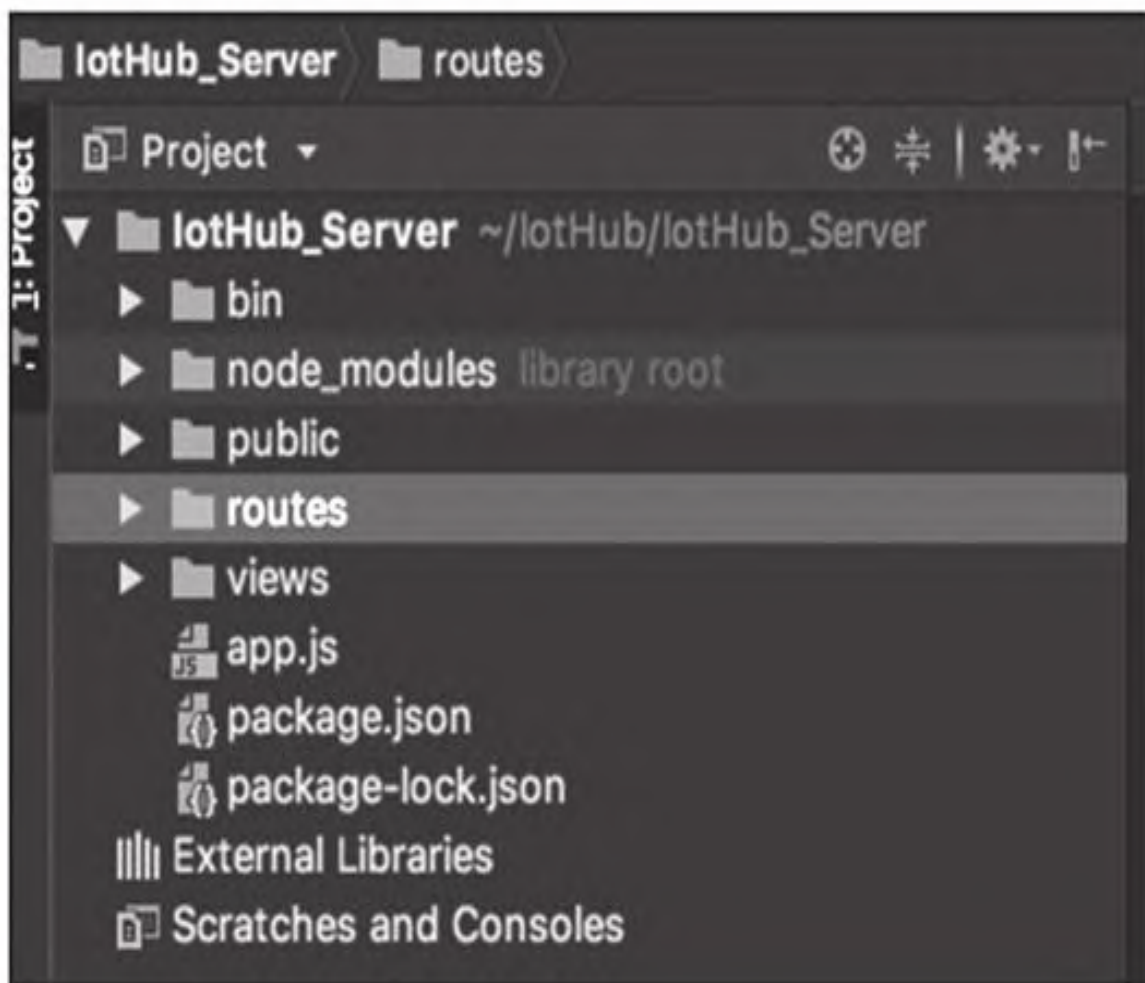


图6-1 IotHub\_Server的项目结构

这个项目包含了Maque IotHub Server API以及Maque IotHub服务端的一些其他功能。

## 6.3.2 IotHub DeviceSDK

设备端代码仍然使用Node.js，但这并不意味着要在设备上运行Node.js，物联网设备的异构性非常大，很难找到一个有广泛代表性的开发语言和平台。

JavaScript是一门表达力很强而且简单易学的语言，它的用户群体也很广，不管是前端程序员，还是后端开发者，都有所涉猎。使用它可以很容易表达DeviceSDK的设计思路，在了解设计思路以后，再移植到实际的物联网设备上就非常容易了。这就是为什么我们选择NodeJS开发整个IotHub的原因。

接下来我们验证一下EMQ X Broker是否已经配置正确，并可以接受MQTT协议连接了。

在NodeJS项目中的package.json中加入对MQTT协议的依赖：

---

```
1. "dependencies": {  
2.   "mqtt": "^2.18.8"  
3. }
```

---

然后运行“npm install”。

我们可以写一小段代码来测试一下MQTT协议连接。

---

```
1. //test_mqtt.js
2. var mqtt = require('mqtt')
3. var client = mqtt.connect('mqtt://127.0.0.1:1883')
4. client.on('connect', function (connack) {
5.   console.log('return code: ${connack.returnCode}')
6.   client.end()
7. })
```

---

如果不出意外，控制台会输出：“return code:0”。

重新把代码组织一下，把与MQTT协议相关的代码以及与Maque IotHub Server交互的代码进行封装，这里实现一个类“IotDevice”作为DeviceSDK的入口。

---

```
1. //iot_device.js
2. "use strict";
3. var mqtt = require('mqtt')
4. const EventEmitter = require('events');
5.
6. class IotDevice extends EventEmitter {
7.   constructor(serverAddress = "127.0.0.1:8883") {
8.     super();
9.     this.serverAddress = 'mqtts://${serverAddress}'
10.  }
11.
12.   connect() {
13.     this.client = mqtt.connect(this.serverAddress, {
14.       rejectUnauthorized: false
15.     })
16.     var self = this
17.     this.client.on("connect", function () {
18.       self.emit("online")
19.     })
20.     this.client.on("offline", function () {
21.       self.emit("offline")
22.     })
23.     this.client.on("error", function (err) {
24.       self.emit("error", err)
```



```
25.     })
26.   }
27.
28.   disconnect() {
29.     if (this.client !== null) {
30.       this.client.end()
31.     }
32.   }
33. }
34.
35.
36. module.exports = IotDevice;
```

---

这段代码做了如下几件事：

- 封装了MQTT Client的connect和disconnect。在代码的第12~15行和第28~31行分别提供了2个接口。

- 在第14行通过设定MQTT Broker地址为“mqttp://127.0.0.1:8883”的方式，在传输层使用SSL。EMQ X默认使用一个自签署的证书，所以我们需要设定“rejectUnauthorized:false”。

- 代码的第4行引入了Node.js的Events库，DeviceSDK通过Events与设备应用代码进行交互。比如在第18行，把设备上线的事件通过Events发布出去，设备应用代码可以通过捕获该Event的方式进行相应的处理。

调用DeviceSDK的设备应用代码示例如下。

---

```
1. var device = new IotDevice()
2. device.on("online", function () {
```

```
3.   console.log("device is online")
4.   device.disconnect()
5. })
6. device.on("offline", function () {
7.   console.log("device is offline")
8. })
9. device.connect()
```

---

代码的第2~4行捕获了设备的上线事件，在设备上线后就断开与IoT Hub的连接。

Maque IoT Hub DeviceSDK的项目结构如图6-2所示。

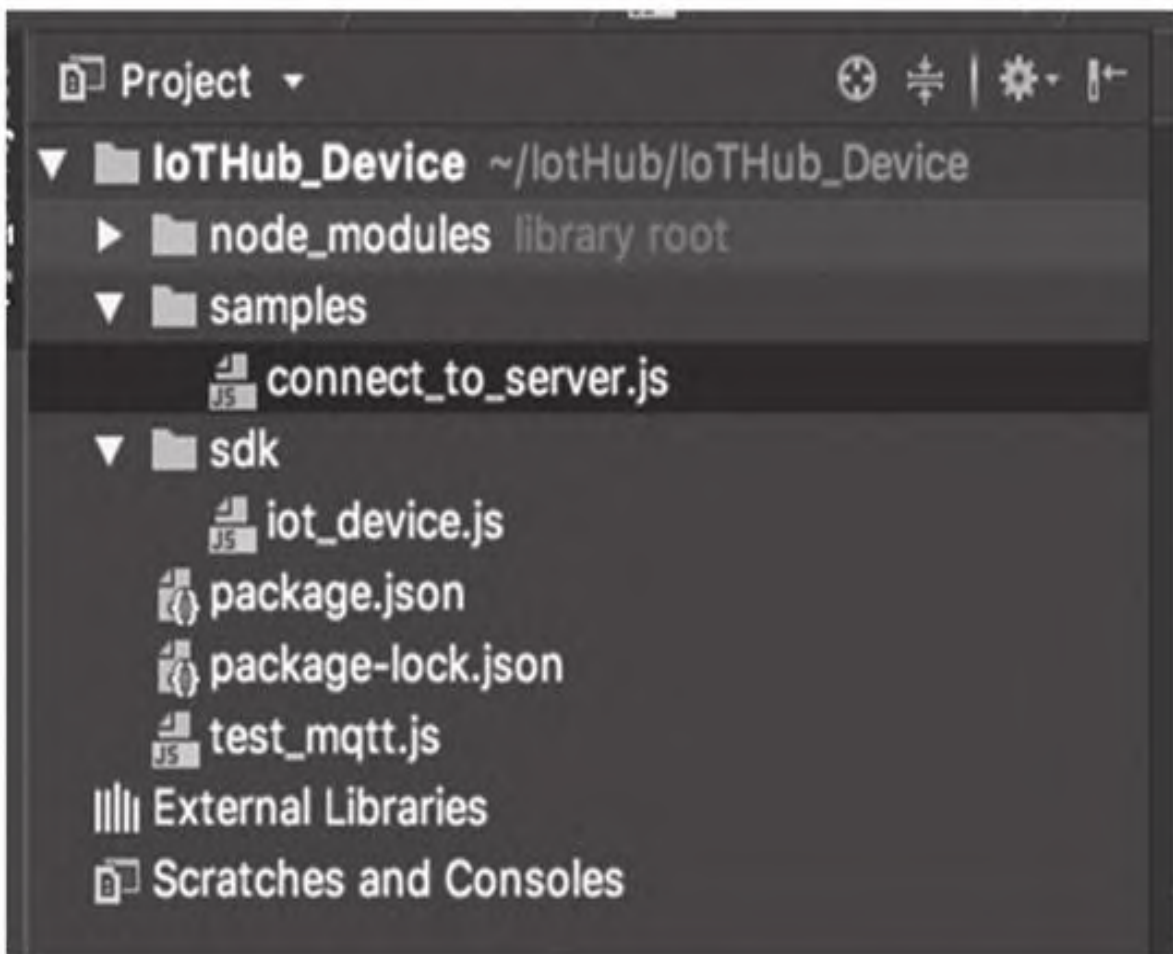


图6-2 IoTHub DeviceSDK的项目结构

- 在sdk目录中的是DeviceSDK的代码。
- 在samples目录中的是调用DeviceSDK的示例代码。

## 6.4 本章小结

工作台准备完毕！准备好开发环境后，下面就开始实现IoTHub的第一个主要功能：设备生命周期管理。

## 第7章 设备生命周期管理

本章将实现IoTHub的设备生命周期管理功能，包括设备接入、状态管理、设备的禁用与删除，以及权限管理等功能。

## 7.1 设备注册

EMQ X在默认的情况是允许匿名连接的，所以在前面的代码里，IotDevice类在连接MQTT Broker时没有指定username和password也能成功。

当然，我们肯定不希望任意一个设备都能连接上IotHub。一个设备接入IotHub的流程应该是这样的：首先需要在IotHub上注册一个设备，设备再通过由IotHub生成的username/password连接到IotHub，以实现一机一密。

### 7.1.1 设备三元组

阿里云IoT平台用一个三元组（ProductKey, DeviceName, Secret）来标识一个逻辑上的设备，ProductKey是指设备所属的产品，DeviceName用来标识这个设备的唯一名称，Secret是指这个设备连接物联网平台使用的密码。我认为这是一个很好的设计，因为即使在同一家公司内部，往往也会有多个服务于不同业务的物联网产品需要接入，所以用ProductKey对后续的主题名、数据存储和分发等进行区分是很有必要的。

IotHub将使用类似的三元组（ProductName, DeviceName, Secret）来标识逻辑上的一台设备。ProductName由业务系统提供，可以是一个有意义的ASCII字符串，DeviceName和Secret由IotHub自动生成，（ProductName, DeviceName）应该是全局唯一的。

这里我们约定，对一个设备（ProductName1, DeviceName1, Secret1）来说，它接入IotHub的username为“ProductName1/DeviceName1”，password为“Secret1”。

为什么说三元组标识的是逻辑上的一台设备而不是物理上的一台设备？比如说：移动应用接入Maque IotHub并订阅某个主题，假如有一个用户在多个移动设备上用同一个账号登录，他使用的应该是同一

个三元组，而他订阅的消息在每个设备上应该都能收到，那么在这种情况下，一个三元组实际上对应多个物理设备。我们后面再讲怎样区分物理设备。

用“/”做分隔符的理由这里先不作说明，在第9章讲到下行数据处理的部分再进行解释。



## 7.1.2 EMQ X的认证方式

EMQ X通过插件的方式提供了多种灵活的认证方式，包括文件、MySQL、Postgres等，你可以在<https://developer.emqx.io/docs/broker/v3/cn/plugins.html#找到> EMQ X自带的插件列表。

IoT Hub使用MongoDB作为数据存储，所以这里我们选择MongoDB认证插件。除了使用MongoDB认证外，我们还会使用JWT的认证方式来提供一种临时性的接入认证。

在启用认证插件前，我们需要关闭EMQ X的默认匿名认证。

编辑“<EMQ X安装目录>/emqx/etc/emqx.conf”，修改以下配置项。

---

```
allow_anonymous = false
```

---

然后重新启动EMQ X。

---

```
<EMQ X 安装目录>/emqx/bin/emqx restart
```

---

### 1. MongoDB认证

MongoDB的认证插件功能逻辑很简单：将设备的username/password存储在MongoDB的某个Collection中，当设备发起连接请求时，Broker再查找这个Collection，如果username/password能匹配上，则允许连接，否则拒绝连接。

在“<EMQ X安装目录>/emqx/etc/plugins/emqx\_auth\_mongo.conf”中可以对MongoDB认证进行配置，配置项很多，这里我们看几个关键的配置项。

- MongoDB地址：auth.mongo.server=127.0.0.1:27017。
- 用于认证的数据库：auth.mongo.database=mqtt，存储设备username和password的数据库，这里暂时用默认值。
- 用于认证的Collection：  
auth.mongo.auth\_query.collection=mqtt\_user，存储设备username和password的Collection，这里暂时使用默认值。
- password字段名：  
auth.mongo.auth\_query.password\_field=password。
- password加密方式：  
auth.mongo.auth\_query.password\_hash=plain，password字段的加密方式，这里选择不加密。

· 是否打开超级用户查询: `auth.mongo.super_query=off`, 设置为关闭。

· 是否打开权限查询: `auth.mongo.acl_query=off`, 这里我们暂时不打开Publish和Subscribe的权限控制。

我们可以在MongoDB中插入一条记录, 在MongoDB Shell中运行。

---

```
1. use mqtt
2. db.createCollection("mqtt_user")
3. db.mqtt_user.insert({username: "test", password:
"123456"})
```

---

然后加载MongoDB认证插件。

---

```
<EMQ X 安装目录>/emqx/bin/emqx_ctl plugins load
emqx_auth_mongo
```

---

不出意外的话, 控制台会输出以下代码。

---

```
Start apps: [emqx_auth_mongo]
Plugin emqx_auth_mongo loaded successfully.
```

---

然后, 运行之前用于测试Broker连接的代码`test_mqtt.js`, 会得到以下输出。

---

```
Error: Connection refused: Bad username or password
```

---

接下来，修改test\_mqtt.js代码，在连接时指定刚才存储在MongoDB中的username/password:test/123456。

---

```
1. var client = mqtt.connect('mqtt://127.0.0.1:1883', {
2.   username: "test",
3.   password: "123456"
4. })
```

---

重新运行test\_mqtt.js，如果输出为“return code:0”，说明基于MongoDB的认证方式已经生效了。

如果返回的是“Error:Connection refused:Bad username or password”，那么你需要检查下面几项设置。

- 插件的配置文件是否按照上文的方式进行配置。
- MongoDB插件是否成功加载，可通过运行“/emqx/bin/emqx\_ctl plugins list”查看。
- 对应的用户名和密码是否添加到MongoDB对应的Collection中。

我们可以通过运行“<EMQ X安装目录>/bin/emqx\_ctl plugins list”的方式查看插件列表，已加载的插件会显示active=true。

## 2. JWT (JSON Web Token) 认证

使用MongoDB认证插件已经能够满足我们对设备注册的需求，但是我在这里还想再引入一种新的认证方式：JWT认证。为什么呢？考虑以下两个场景：

- 在浏览器中，使用WebSocket方式进行接入时，你需要将接入Maque IotHub的username和password并传给前端的JavaScript代码，那么在浏览器的Console里就可以看见username和password，这非常不安全。如果使用JWT认证方式，你只需要将一段有效期很短的JWT传给前端的JavaScript代码，即使泄露了，可以操作的时间窗口也很短；

- 有时候你需要绕过注册设备这个流程来连接IotHub，EMQ X会在一些内部的系统主题上发布与Broker相关的状态信息，比如连接数、消息数等。如果你需要用Client连接到IotHub并订阅这些主题，先创建一个Device并不是很好的选择，在这种情况下，用JWT作为一次性的密码为这些系统内部的接入做认证就会非常方便。

JWT是一种基于JSON的、用于在网络上声明某种主张的令牌（Token），更详细的介绍可以参考相关资料。

EMQ X提供了JWT认证插件来提供使用JWT的认证方式，在“<EMQ X安装目录>/emqx/etc/plugins/emqx\_auth\_jwt.conf”中可以对JWT认证插件进行配置。

- JWT Secret:auth.jwt.secret=emqxsecret, 这里使用默认值, 在实际生产中你需要使用一个长且复杂的字符串。

- 是否开启Claim验证: auth.jwt.verify\_claims=on, 打开Claim验证。

- Claim验证字段: auth.jwt.verify\_claims.username=%u, 需要验证Claim中的username字段。

接下来, 修改test\_mqtt.js的代码。

---

```
1. var jwt = require('jsonwebtoken')
2. var password = jwt.sign({
3.   username: "jwt_user",
4.   exp: Math.floor(Date.now() / 1000) + 10
5. }, "emqxsecret")
6. var client = mqtt.connect('mqtt://127.0.0.1:1883', {
7.   username: "jwt_user",
8.   password: password
9. })
```

---

代码的第2~4行, 使用我们在EMQ X Broker中配置的JWT Secret “emqxsecret” 来签发一个JWT token, 它的Payload是 “jwt\_user”, 与连接Broker时使用的用户名一致才能通过验证。

代码的第4行将JWT Token的有效值设为10秒。

重新运行test\_mqtt.js，如果输出为“return code:0”，说明基于JWT的认证方式已经生效了。

如果返回的是“Error:Connection refused:Bad username or password”，则需要检查：

- 插件的配置文件是否已经按照上文中的方式进行配置；
- JWT插件是否成功加载，可通过运行“/bin/emqx\_ctl plugins list”查看；
- 是否是使用书中指定的Payload生成JWT。

### 3. 认证链

我们加载了MongoDB和JWT两个认证插件，EMQ X就可以用这两个插件组成的认证链对接入的Client进行认证。简单来说，设备既可以使用存储在MongoDB里的username和password接入EMQ X Broker，也可以使用JWT接入EMQ X Broker。

EMQ X在加载一个插件后，会把这个插件的名字写入“<EMQ X安装目录>/emqx/data/loaded\_plugins”，EMQ X在每次启动时都会自动加载这个文件里包含的插件，所以我们只需要手动加载一次这两个插件就可以了。

### 7.1.3 设备接入流程

接下来，我们定义IoT Hub中设备从注册到接入的流程。

1) 业务系统调用IoT Hub Server API的设备注册API，提供要注册设备的ProductName。

2) IoT Hub Server根据业务系统提供的参数生成一个三元组 (ProductName, DeviceName, Secret)，然后将该三元组存储到MongoDB，同时存储到MongoDB的还有该设备接入EMQ X的用户名“ProductName/DeviceName”。

3) IoT Hub Server API将生成的三元组返回给业务系统，业务系统应该保存这个三元组，以后调用IoT Hub Server API时需要使用。

4) 业务系统通过某种方式，例如烧写Flash，将这个三元组“写”到物联网设备上。

5) 设备应用代码调用DeviceSDK，传入三元组。

6) DeviceSDK使用

username:ProductName/DeviceName,password:Secret连接到EMQ X Broker。



7) EMQ X Broker到MongoDB里查询ProductName/DeviceName和Secret，如果匹配成功，则允许连接。

流程如图7-1所示。

## 7.1.4 Server API：设备注册

接下来，在IotHub\_Server项目里实现用IotHub Server API的设备注册API。

首先在MongoDB里创建一个名为IotHub的数据库存储设备信息。

### 1. 定义设备模型

我们使用mongoose执行MongoDB的相关操作，首先定义用于存储设备信息的Device模型，代码如下。

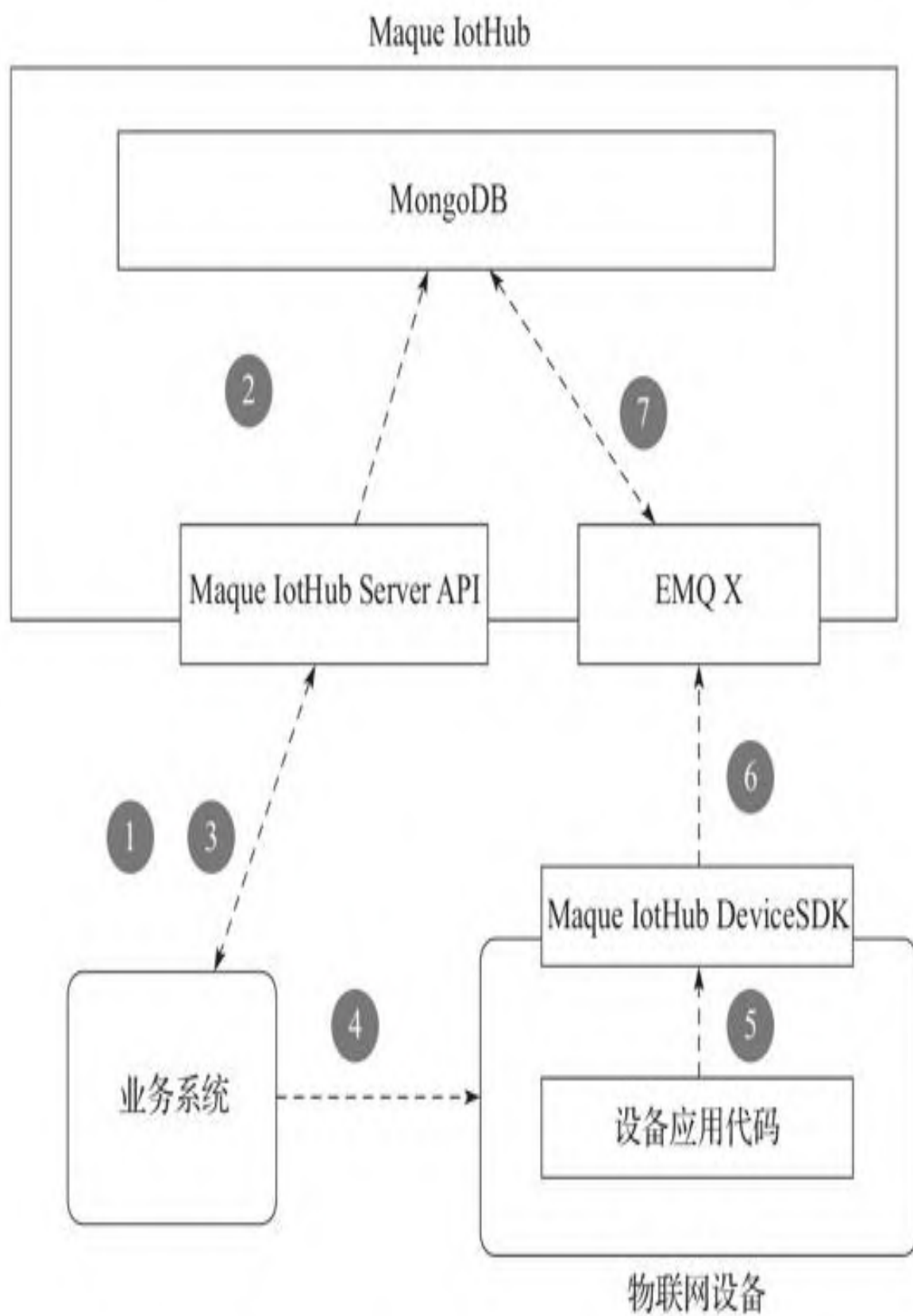


图7-1 设备注册流程

Mongoose是一个MongoDB的ORM。

---

```
1. // IotHub_Server/models/device.js
2. const deviceSchema = new Schema({
3.   //ProductName
4.   product_name: {
5.     type: String,
6.     required: true
7.   },
8.   //DeviceName
9.   device_name: {
10.    type: String,
11.    required: true,
12.  },
13.   //接入 EMQ X 时使用的 username
14.   broker_username: {
15.     type: String,
16.     required: true
17.   },
18.   //secret
19.   secret: {
20.     type: String,
21.     required: true,
22.   }
23. })
```

---

## 2. RESTful API实现

每次在生成新设备时，由系统自动生成DeviceName和Secret，DeviceName和Secret应该是随机且唯一的字符串，例如UUID，这里我们用shortid来生成稍短一点的随机唯一字符串，代码如下。

Shortid是一个NodeJS的库，可以生成短的随机的唯一字符串。

---

```
1. // routes/devices.js
2. ...
3. router.post("/", function (req, res) {
4.   var productName = req.body.product_name
5.   var deviceName = shortid.generate();
6.   var secret = shortid.generate();
7.   var brokerUsername = '${productName}/${deviceName}'
8.
9.   var device = new Device({
10.     product_name: productName,
11.     device_name: deviceName,
12.     secret: secret,
13.     broker_username: brokerUsername
14.   })
15.
16.   device.save(function (err) {
17.     if(err){
18.       res.status(500).send(err)
19.     }else{
20.       res.json({product_name: productName,
device_name: deviceName, secret:
secret})
21.     }
22.   })
23. })
24.
25. ...
```

---

接着我们将这个router挂载到路径 “/devices” 下，并连接到 MongoDB。

---

```
1. //app.js
2. ...
3.
mongoose.connect('mongodb://iot:iot@localhost:27017/iothub
', { useNewUrlParser:
true })
4. var deviceRouter = require('./routes/devices');
5. app.use('/devices', deviceRouter);
6. ...
```

---

运行“bin/www”启动Web服务器，然后在命令行用curl调用下面这个接口。

---

```
curl -d "product_name=IotApp" -X POST
http://localhost:3000/devices
```

---

输出应为

“{"product\_name":"IotApp","device\_name":"V5MyuncRK","secret":"GNxU20VYTZ"}”。

ProductName包含的字符是有限制的，不能包含“#” “/” “+”以及IoT Hub预留的一些字符，为了演示，这里跳过了输入参数的校验，但在实际项目中是需要加上的。

到这里，设备注册就成功了，我们需要记录下这个三元组。

## 7.1.5 调整EMQ X配置

我们需要按照IoTHub定义的数据库结构修改EMQ X MongoDB认证插件的配置，下面是需要在之前的配置上进行修改的项。

---

```
1. # 存储用户名和密码的 database
2. auth.mongo.database = iotHub
3.
4. # 存储用户名和密码的 collection
5. auth.mongo.auth_query.collection = devices
6.
7. # 密码字段
8. auth.mongo.auth_query.password_field = secret
9.
10. # 查询记录时的 selector
11. auth.mongo.auth_query.selector = broker_username=%u
```

---

编辑完成后需要重新加载MongoDB认证插件。

运行“<EMQ X安装目录>/emqx/bin/emqx\_ctl plugins reload emqx\_auth\_mongo”。

## 7.1.6 修改DeviceSDK

接下来在IoTHub\_Device项目里对DeviceSDK进行修改，接受三元组作为初始化参数。

---

```
1. // sdk/iot_device.js
2.
3. ...
4. class IotDevice extends EventEmitter {
5.   constructor({serverAddress = "127.0.0.1:8883",
productName, deviceName,
secret} = {}) {
6.     super();
7.     this.serverAddress = `mqtt://${serverAddress}`
8.     this.productName = productName
9.     this.deviceName = deviceName
10.    this.secret = secret
11.    this.username =
`${this.productName}/${this.deviceName}`
12.  }
13.  connect() {
14.    this.client = mqtt.connect(this.serverAddress, {
15.      rejectUnauthorized: false
16.      username: this.username,
17.      password: this.secret
18.    })
19.    ...
20.  }
21.
22.  ...
23. }
24. ...
```

---



然后，用刚才记录下的三元组作为参数调用DeviceSDK接入IoTHub。

---

```
1. // samples/connect_to_server.js
2. ...
3. var device = new IoTDevice({productName: "IotApp",
  deviceName: "V5MyuncRK",
  secret: "GNxU20VYTZ"})
4. device.connect()
5. ...
```

---

最后，运行“samples/connect\_to\_server.js”，我们会得到输出“device is online”。

这说明设备已经完成注册并成功接入了IoTHub。

## 7.1.7 Server API：设备信息查询

我们还需要几个接口完善注册流程。

### 1. 获取单个设备的信息

当业务系统查询设备信息的时候，我们并不需要把Device的所有字段都返回。首先定义返回内容。

---

```
1. // IotHub_Server/models/device.js
2. //定义 device.toJSONObject
3. deviceSchema.methods.toJSONObject = function () {
4.   return {
5.     product_name: this.product_name,
6.     device_name: this.device_name,
7.     secret: this.secret
8.   }
9. }
```

---

然后实现Server API接口。

---

```
1. // IotHub_Server/routes/devices.js
2. router.get("/:productName/:deviceName", function
(req, res) {
3.   var productName = req.params.productName
4.   var deviceName = req.params.deviceName
5.   Device.findOne({"product_name": productName,
"device_name": deviceName},
function (err, device) {
6.     if (err) {
7.       res.send(err)
8.     } else {
```

```
9.         if (device != null) {
10.             res.json(device.toJSONObject())
11.         } else {
12.             res.status(404).json({error: "Not Found"})
13.         }
14.     }
15. })
16. })
```

---

可以用curl调用这个接口看一下效果。

---

```
curl http://localhost:3000/devices/IotApp/V5MyuncRK
{"product_name":"IotApp","device_name":"V5MyuncRK","secret":"GNxU20VYTZ"}
```

---

## 2. 获取设备列表

这里我们实现一个接口，可以根据ProductName列出该产品下的所有设备。

---

```
1. // IotHub_Server/routes/devices.js
2. router.get("/:productName", function (req, res) {
3.     var productName = req.params.productName
4.     Device.find({"product_name": productName}, function
(err, devices) {
5.         if (err) {
6.             res.send(err)
7.         } else {
8.             res.json(devices.map(function (device) {
9.                 return device.toJSONObject()
10.            }))
11.        }
12.    }
13. })
14. })
```

---

可以用curl调用这个接口看看效果。

---

```
curl http://localhost:3000/devices/IotApp
[{"product_name":"IotApp","device_name":"V5MyuncRK","secret":"GNxU20VYTZ"}]
```

---

### 7.1.8 Server API：获取接入IotHub的一次性密码（JWT）

在前面使用JWT认证的示例代码里，我们是直接使用配置在IotHub中的secret进行签发，但在实际项目中，我们是不会把IotHub配置的JWT secret暴露出去的。我们可以提供一个接口，业务系统通过这个接口向IotHub申请一对用于临时接入IotHub的用户名和密码。

---

```
1. // IotHub_Server/routes/tokens.js
2.
3. var express = require('express');
4. var router = express.Router();
5. var shortid = require("shortid")
6. var jwt = require('jsonwebtoken')
7.
8. //这个值应该与EMQ X etc/plugins/emqx_auth_jwt.conf 中的
  保持一致
9. const jwtSecret = "emqxsecret"
10.
11. router.post("/", function (_, res) {
12.   var username = shortid.generate()
13.   var password = jwt.sign({
14.     username: username,
15.     exp: Math.floor(Date.now() / 1000) + 10 * 60
16.   }, jwtSecret)
17.   res.json({username: username, password: password})
18. })
19.
20. module.exports = router
```

---

代码的第12行是生成一个随机的字符串作为临时的用户名。

然后挂载这个接口。

---

```
1. // IoTHub_Server/app.js
2. var tokensRouter = require('./routes/tokens')
3. app.use('/tokens', tokensRouter)
```

---

通过这个接口，可以签发一个有效期为1分钟的  
username/password。

---

```
curl http://localhost:3000/tokens -X POST
{"username":"apmE_JPl1","password":"eyJhbGciOiJIUzI1NiIsI
nR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFwbUVfSlBsbCI6ImV4cCI6MTU2ODMxNjk2MSwia
WF0IjoxNTU4MzE2OTYxfQ.-SnqvBGdO3wjSu7IHR91Bo58gb-
VLFuQ28BeN6hlTLk"}
```

---

大家可能还发现，在ServerAPI中没有对调用者的身份进行认证和权限控制，也没有对输入参数进行校验，输出列表时也没有进行分页等处理，在实际的项目中，这些都是有必要的。但是，这些属于Web编程的范畴，大家应该都非常熟悉了，所以在此就跳过了，有需要的读者可以自行扩展。

## 7.1.9 完善细节

### 1. 添加数据库索引

由于IoTHub经常需要通过Device的product\_name和device\_name进行查询，所以我们需要在这两个字段上加上索引，在MongoDB shell里面输入如下代码。

---

```
1. use iotHub
2. db.devices.createIndex({
3.   "production_name" : 1,
4.   "device_name" : 1
5. }, { unique: true })
```

---

MongoDB插件在每次接入设备时都会使用broker\_name查询Devices Collection，所以我们也需要在broker\_name上加一个索引。

---

```
1. use iotHub
2. db.devices.createIndex({
3.   "broker_username" : 1
4. })
```

---

### 2. 使用持久化连接

细心的读者可能已经发现，DeviceSDK在连接到Broker的时候并没有指定Client Identifier。没错，到目前为止，我们使用的都是在连

接时自动分配的Client Identifier, 没有办法很好地使用QoS1和QoS2的消息。

Client Identifier是用来唯一标识MQTT Client的, 由于我们之前的设计保证了(ProductName, DeviceName)是全局唯一的, 因此一般来说用这个二元组作为Client Identifier就足够了。但是, 之前我也提到过, 在某些场景下, 可能会出现多个设备使用同样的设备三元组接入IoT Hub。综合这些情况, 我们可按下面的方法设计IoT Hub里的Client Identifier。

设备提供一个可选的Client ID标识自己, 可以是硬件编号、Android ID等, 如果设备提供Client ID, 那么使用ProductName/DeviceName/Client ID作为连接Broker的Client Identifier, 否则使用ProductName/DeviceName。根据这个规则对DeviceSDK进行修改。

---

```
1. // IoT Hub_Device/sdk/iot_devices.js
2. ...
3. class IoTDevice extends EventEmitter {
4.   constructor({serverAddress = "127.0.0.1:8883",
productName, deviceName,
secret, clientId} = {}) {
5.     super();
6.     this.serverAddress = `mqtt://${serverAddress}`
7.     this.productName = productName
8.     this.deviceName = deviceName
9.     this.secret = secret
10.    this.username =
`${this.productName}/${this.deviceName}`
```



```
11.      //根据 ClientID 设置
12.      if(clientID != null){
13.          this.clientIdentifier =
14.          '${this.username}/${clientID}'
15.      }else{
16.          this.clientIdentifier = this.username
17.      }
18.
19.      connect() {
20.          this.client = mqtt.connect(this.serverAddress, {
21.              rejectUnauthorized: false,
22.              username: this.username,
23.              password: this.secret,
24.              //设置 ClientID 和 clean session
25.              clientId: this.clientIdentifier,
26.              clean: false
27.          })
28.          ...
29.      }
30.      ...
```

---

然后，我们可以再运行一次samples/connect\_to\_server.js看下效果。

因为Node.js的MQTT库自带断线重连功能，所以这里就不用自己实现了。

### 3. 从环境变量中读取配置

根据The Twelve-Factor App的理念，从环境变量中读取配置项是一个非常好的实践，在我们的项目中有两个地方要用到配置：

- ServerAPI，比如mongoDB的地址；

· DeviceSDK端的samples里的代码会经常使用到预先注册的三元组 (ProductName, DeviceName, Secret) 。

这里我们使用NodeJS的dotenv库管理环境变量，它可以从.env文件中读取并设置环境变量。

The Twelve-Factor App，即12要素应用，是一个设计符合现代要求的应用的方法论和经验总结

首先，在IotHub\_Server项目中创建一个.env文件。

---

```
1. # IotHub_Server/.env
2. MONGODB_URL=mongodb://iot:iot@localhost:27017/iothub
3. JWT_SECRET=emqxsecret
```

---

然后，在IotHubServer启动的时候加载.env中预设的环境变量。

---

```
1. // IotHub_Server/app.js
2. require('dotenv').config()
```

---

连接MongoDB时使用环境变量中的配置。

---

```
1. // IotHub_Server/app.js
2. ...
3. mongoose.connect(process.env.MONGODB_URL, {
  useUrlParser: true })
4. ...
```

---

在接口中使用环境变量中的配置。

---

```
1. // IotHub_Server/routes/tokens.js
2. ...
3. const jwtSecret = process.env.JWT_SECRET
4. ...
```

---

同样，在IotHub\_Device项目中创建一个.env文件。

---

```
1. #IotHub_Device/samples/.env
2. PRODUCT_NAME=注册接口获取的 ProductName
3. DEVICE_NAME=注册接口获取的 DeviceName
4. SECRET=注册接口获取的 Secret
```

---

然后在代码中读取环境变量中的配置。

---

```
1. // IotHub_Device/samples/connect_to_server.js
2. require('dotenv').config()
3. var device = new IotDevice({
4.   productName: process.env.PRODUCT_NAME,
5.   deviceName: process.env.DEVICE_NAME,
6.   secret: process.env.SECRET
7. })
```

---

我们在本节中设计和实现了IotHub设备注册的所有功能，接下来将实现IotHub设备连接状态的监控功能。

## 7.2 设备连接状态管理

如何得知一个设备是在线或离线？这是一个经常会被问到的问题，这也是实际生产中非常必要的一个功能。那么它有哪些解决方案呢？

## 7.2.1 Poor man' s Solution

MQTT协议并没有在协议级别约定如何对Client的在线状态进行管理，但我们在第二部分里介绍过一个解决思路。

1) Client在连接成功时向TopicA发送一个消息，表示Client已经上线。

2) Client在连接时指定LWT，Client在离线时向TopicA发送一个Retained消息，表示已经离线。

3) 只要订阅TopicA就可以获取Client上线和离线的状态。

这个解决方案在实际上是可行的，但有一个问题，你始终需要保持一个接入Broker的Client来订阅TopicA，如果设备的数量高达十万甚至几十万，这个订阅TopicA的Client就很容易成为单点故障点，所以说这种解决方案的可扩展性比较差。

## 7.2.2 使用EMQ X的解决方案

EMQ X提供了丰富的管理功能和接口，我们可以利用EMQ X提供的这些功能来实现设备的连接状态管理。

### 1. 系统主题

EMQ X使用许多系统主题发布Broker内部的状态和事件，当Client Identifier为“`${clientid}`”的Client连接到节点名为“`${node}`”的EMQ X Broker时，EMQ X Broker会向系统主题

“`$SYS/brokers/${node}/clients/${clientid}/connected`”发布一条消息；当这个Client离线时，EMQ X会向系统主题

“`$SYS/brokers/${node}/clients/${clientid}/disconnected`”发布一条消息。

EMQ X Broker的节点名可以在“`<EMQ X安装目录>/emqx/etc/emqx.conf`”里进行配置。配置项为`node.name`，默认值为`emqx@127.0.0.1`。

你可以在

<https://developer.emqx.io/docs/broker/v3/cn/guide.html#sys>找到系统主题列表。

那么，我们只需要订阅

“\$SYS/brokers/+/clients/+/connected” 和

“\$SYS/brokers/+/clients/+/disconnected” 就可以获取到每个EMQX节点上所有Client的上线和离线事件。实现代码如下。

---

```
1. // IotHub_Device/samples/sys_topics.js
2. var mqtt = require('mqtt')
3. var jwt = require('jsonwebtoken')
4. require('dotenv').config()
5. var password = jwt.sign({
6.   username: "jwt_user",
7.   exp: Math.floor(Date.now() / 1000) + 10
8. }, process.env.JWT_SECRET)
9. var client = mqtt.connect('mqtt://127.0.0.1:1883', {
10.   username: "jwt_user",
11.   password: password
12. })
13. client.on('connect', function () {
14.   console.log("connected")
15.
16.   client.subscribe("$SYS/brokers/+/clients/+/connected")
17.   client.subscribe("$SYS/brokers/+/clients/+/disconnected")
18. })
19. client.on("message", function (_, message) {
20.   console.log(message.toString())
21. })
```

---

先运行 “sys\_topics.js”，随后运行

“connect\_to\_server.js”，接着关闭 “connect\_to\_server.js”，我们会看到以下输出。

---

```
{"clean_start":false,"clientid":"IotApp/V5MyuncRK","connack":0,"ipaddress":"127.0.0.1","Keepalive":60,"proto_name":"MQTT","proto_ver":4,"ts":1558335733,"username":"IotApp/V5MyuncRK"}
{"clientid":"IotApp/V5MyuncRK","username":"IotApp/V5MyuncRK","reason":"closed","ts":1558335752}
```

---

第1行是Client connected事件的信息，第2行是Client disconnected的信息，这些信息包含ClientID、IP地址、连接时间等，非常详细。

这种解决方案的缺点也很明显，和上面提到的一样，订阅这两个主题的Client很容易成为单点故障点。

## 2. 基于Hook系统的解决方案

我比较喜欢EMQ X的一点就是它设计了一套Hook系统，你可以通过这个Hook系统捕获Broker内部的事件并进行处理。EMQ X中Hook系统的定义如图7-2所示。



# 钩子(Hook)设计

## 钩子(Hook)定义

EMQ X 消息服务器在客户端上下线、主题订阅、消息收发位置设计了扩展钩子(Hook):

钩子	说明
client.authenticate	客户端认证
client.check_acl	客户端 ACL 检查
client.connected	客户端上线
client.subscribe	客户端订阅主题前
client.unsubscribe	客户端取消订阅主题
session.subscribed	客户端订阅主题后
session.unsubscribed	客户端取消订阅主题后
message.publish	MQTT 消息发布
message.deliver	MQTT 消息投递前
message.acked	MQTT 消息回执
client.disconnected	客户端连接断开

图7-2 EMQ X中Hook系统的定义

通常我们需要编写一个插件来捕获并处理这些事件，不过EMQ X自带了一个WebHook插件，它的原理很简单，当像Client上线或下线之类的事件发生时，EMQ X会把事件的信息发送到一个事先指定好的URL上，这样我们就可以进行处理了。

如何避免一个Web服务成为单点故障点，我想大家应该都很熟悉了，所以在这里我们可以使用基于WebHook的方式实现设备的在线状态管理。

### 3. 开启WebHook

首先需要编辑WebHook插件的配置文件，将回调的URL指向本地运行的Express应用。

---

```
1. #< EMQ X 安装目录>/emqx/etc/plugins/emqx_web_hook.conf
2. web.hook.api.url = http://127.0.0.1:3000/emqx_web_hook
```

---

然后重新加载WebHook插件。

---

```
< EMQ X 安装目录>/emqx/bin/emqx_ctl plugins load
emqx_web_hook
```

---

我们先简单实现一下WebHook的回调，只把EMQ X发送过来的信息打印出来。

---

```
1. // IotHub_Server/routes/emqx_web_hook.js
2. var express = require('express');
3. var router = express.Router();
4.
5. router.post("/", function (req, res) {
6.   console.log(req.body)
7.   res.status(200).send("ok")
8. })
9.
10. module.exports = router
```

---

然后挂载上面的router。

---

```
1. // IotHub_Server/app.js
2. var webHookeRouter = require('./routes/emqx_web_hook')
3. app.use('/emqx_web_hook', webHookeRouter)
```

---

运行“samples/connect\_to\_server.js”，然后将其关闭，我们可以发现，当Client连接时，EMQ X会把以下的JSON发送到指定的URL。

---

```
1. {
2.   action: 'client_connected',
3.   client_id: 'IotApp/V5MyuncRK',
4.   username: 'IotApp/V5MyuncRK',
5.   Keepalive: 60,
6.   ipaddress: '127.0.0.1',
7.   proto_ver: 4,
8.   connected_at: 1558338318,
```

```
9.    conn_ack: 0
10. }
```

---

当Client断开连接时，EMQ X会把以下JSON发送到指定的URL。

---

```
1. {
2.   action: 'client_disconnected',
3.   client_id: 'IotApp/V5MyuncRK',
4.   username: 'IotApp/V5MyuncRK',
5.   reason: 'closed'
6. }
```

---

connected\_at是指连接时刻的UNIX时间戳。

/emq\_web\_hook这个URL是在IotHub内部使用的，除了WebHook，外部是不应该能够访问的，本书为了让内容更紧凑，尽量跳过了这些属于Web编程范畴的内容，但在实际项目中，是需要考虑的。

在Client connect和disconnect事件中，包含了Client连接时使用的username，而username里面包含了（ProductName，DeviceName），所以我们可以通过这些信息定位到具体是哪一个设备已经连接或者没有连接，从而更新设备的连接状态。

### 7.2.3 管理设备的连接状态

验证了WebHook的功能后，我们可以开始设计设备连接状态管理功能了。

1) IotHub不保存某个设备在线与否这个boolean值，而是保存一个connection列表，这个列表包含了所有用这个设备的三元组接入的connection，connection的信息由WebHook捕获的client\_connected事件提供。

2) 当收到client\_connected的消息时，通过username里面的ProductName和DeviceName查找到Device记录，然后用ClientID查找Device的connection列表，如果不存在该ClientID的connection记录就新增一条connection记录；如果存在，则更新这条connection记录，状态为connected。

3) 当收到client\_disconnected的消息时，通过username里面的ProductName和DeviceName查找到Device记录，然后用ClientID查找Device的connection列表，如果存在该ClientID的connection记录，则更新这条connection记录，状态为disconnected。

4) 业务系统可以通过调用Server API的设备详情接口获取设备的连接状态。

通过这样的设计，我们不仅可以知道一个设备是否在线，还能知道其连接的具体信息。

## 1. Connection模型

我们定义一个Connection模型来存储连接信息。

---

```
1. // IotHub_Server/models/connection.js
2.
3. const connectionSchema = new Schema({
4.   connected: Boolean,
5.   client_id: String,
6.   Keepalive: Number,
7.   ipaddress: String,
8.   proto_ver: Number,
9.   connected_at: Number,
10.  disconnect_at: Number,
11.  conn_ack: Number,
12.  device: {type: Schema.Types.ObjectId, ref:
'Device'}
13. })
14.
15. const Connection = mongoose.model("Connection",
connectionSchema);
```

---

## 2. 实现回调

我们在Device类里用两个方法来实现对connect事件和disconnect事件的处理。

---

```
1. // IotHub_Server/models/device.js
2.
3. //connected
4. deviceSchema.statics.addConnection = function (event)
{
5.   var username_arr = event.username.split("/")
6.   this.findOne({product_name: username_arr[0],
device_name: username_
arr[1]}}, function (err, device) {
7.     if (err == null && device != null) {
8.       Connection.findOneAndUpdate({
9.         client_id: event.client_id,
10.        device: device._id
11.      }, {
12.        connected: true,
13.        client_id: event.client_id,
14.        Keepalive: event.Keepalive,
15.        ipaddress: event.ipaddress,
16.        proto_ver: event.proto_ver,
17.        connected_at: event.connected_at,
18.        conn_ack: event.conn_ack,
19.        device: device._id
20.      }, {upsert: true, useFindAndModify: false, new:
true}).exec()
21.    }
22.  })
23.
24. }
25. //disconnect
26. deviceSchema.statics.removeConnection = function
(event) {
27.   var username_arr = event.username.split("/")
28.   this.findOne({product_name: username_arr[0],
device_name: username_
arr[1]}}, function (err, device) {
29.     if (err == null && device != null) {
30.       Connection.findOneAndUpdate({client_id:
event.client_id, device:
device._id},
31.        {
32.          connected: false,
33.          disconnect_at: Math.floor(Date.now() /
1000)
34.        }, {useFindAndModify: false}).exec()
```

```
35.     }  
36.   })  
37. }
```

---

代码的第8~20行调用了findOneAndUpdate方法，这样的话，如果client\_id为event.client\_id的Connection记录不存在，则创建一条新的记录，否则更新已有的Connection记录。

其次，我们修改一下之前实现的回调，在收到对应事件信息时调用上面实现的两个方法。

---

```
1. //IotHub_Server/routes/emqx_web_hook  
2.  
3. router.post("/", function (req, res) {  
4.   if (req.body.action == "client_connected") {  
5.     Device.addConnection(req.body)  
6.   }else if(req.body.action == "client_disconnected"){  
7.     Device.removeConnection(req.body)  
8.   }  
9.   res.status(200).send("ok")  
10. })
```

---

然后，修改设备详情接口，在接口的返回中加上设备的Connection数据。首先，定义Connection的JSON返回格式。

---

```
1. // IotHub_Server/models/connection.js  
2. connectionSchema.methods.toJSONObject = function () {  
3.   return {  
4.     connected: this.connected,  
5.     client_id: this.client_id,  
6.     ipaddress: this.ipaddress,  
7.     connected_at: this.connected_at,  
8.     disconnect_at: this.disconnect_at
```



```
9.    }  
10. }
```

---

最后，通过接口查询device对应的connection后返回。

---

```
1. // IotHub_Server/routes/devices.js  
2. router.get("/:productName/:deviceName", function  
(req, res) {  
3.     var productName = req.params.productName  
4.     var deviceName = req.params.deviceName  
5.     Device.findOne({"product_name": productName,  
"device_name": deviceName}).  
exec(function (err, device) {  
6.         if (err) {  
7.             res.send(err)  
8.         } else {  
9.             if (device != null) {  
10.                 Connection.find({device: device._id},  
function (_, connections) {  
11.                     res.json(Object.assign(device.toJSONObject(), {  
12.                         connections: connections.map(function  
(conn) {  
13.                             return conn.toJSONObject()  
14.                         }))  
15.                     })))  
16.                 })  
17.             } else {  
18.                 res.status(404).json({error: "Not Found"})  
19.             }  
20.         }  
21.     })  
22. })
```

---

代码的第11~12行把device的JSON Object和Connection的JSON Object数组合并起来作为接口的返回结果。

代码完成后，运行

“IotHub\_Device/samples/connect\_to\_server.js”，然后调用设备详情接口“curl<http://localhost:3000/devices/IotApp/c-j0c-2qq>”。我们会得到以下输出。

---

```
{ "product_name": "IotApp", "device_name": "c-j0c-2qq", "secret": "m0PfE0DcNC", "connections": [ { "connected": true, "client_id": "IotApp/c-j0c-2qq", "ipaddress": "127.0.0.1", "connected_at": 1558354603 } ] }
```

---

关闭“connect\_to\_server.js”，再次调用设备详情接口

“curl<http://localhost:3000/devices/IotApp/c-j0c-2qq>”，我们会得到以下输出。

---

```
{ "product_name": "IotApp", "device_name": "c-j0c-2qq", "secret": "m0PfE0DcNC", "connections": [ { "connected": false, "client_id": "IotApp/c-j0c-2qq", "ipaddress": "127.0.0.1", "connected_at": 1558354603, "disconnect_at": 1558355260 } ] }
```

---

由于DeviceName是随机生成的，读者在使用curl调用接口时，应该使用在本地生成的DeviceName替代相应参数或路径。

这样我们就完成了设备状态管理，业务系统通过查询设备详情，就可以知道设备是否在线（有connected==true的connection记录），以及设备连接的一些附加信息。

本节我们设计并实现了IoTHub的设备连接状态管理功能，细心的读者可能已经发现，我们的解决方案是有一点瑕疵的。

- 存在性能问题。每次设备上下线，包括Publish/Subscribe等，EMQ X都会发起一个HTTP POST，这肯定是有损耗的，至于多大损耗以及能不能接受这个损耗，取决于你的业务和数据量。

- 由于Web服务是并发的，因此有可能出现在很短时间内发生的一对connect/disconnect事件，而disconnect会比connect先处理，从而导致设备的连接状态不正确；

- 设备下线时间我们取的是处理这个事件的时间，不准确。

在第三部分的后半部分，我们会实现一个基于RabbitMQ的插件来替换WebHook，在那个时候，我们再来尝试解决这些问题，在此之前，我们都将用WebHook完成功能验证。

## 7.3 设备的禁用与删除

有时候我们可能需要暂停某个设备接入IoT Hub的权限，或者彻底删除这个设备。

### 7.3.1 禁用设备

设备禁用的逻辑很简单，业务系统可以通过一个接口暂停设备的接入认证，被禁用的设备无法接入IotHub；业务系统也可以通过一个接口恢复设备的接入认证，使设备可以重新接入IotHub。这里还暗含着另外一个操作，即在禁用设备时，如果这个设备已经接入IotHub且是在线状态，那么需要将这个设备踢下线。接下来，我们来一步步实现这些功能。

#### 1. 设备接入状态

首先在Device模型里加一个字段，标识设备当前是否可以接入IotHub。

---

```
1. // IotHub_Server/models/Device.js
2. const deviceSchema = new Schema({
3.   //ProductName
4.   product_name: {
5.     type: String,
6.     required: true
7.   },
8.   //DeviceName
9.   device_name: {
10.    type: String,
11.    required: true,
12.  },
13.  //接入EMQ X时使用的username
14.  broker_username: {
15.    type: String,
```

```
16.     required: true
17.   },
18.   //secret
19.   secret: {
20.     type: String,
21.     required: true,
22.   },
23.
24.   //可接入状态
25.   status: String
26. })
```

---

在创建Device时，这个字段的默认值为active。

---

```
1. //IotHub_Server/routes/device.js
2. router.post("/", function (req, res) {
3.   ...
4.
5.   var device = new Device({
6.     product_name: productName,
7.     device_name: deviceName,
8.     secret: secret,
9.     broker_username: brokerUsername,
10.    status: "active"
11.  })
12.
13.  ...
14. })
```

---

在连接设备时，还要通过这个字段判断设备是否可以接入，我们需要配置MongoDB认证插件。

---

```
# <EMQ X 安装目录>/etc/plugins/emqx_auth_mongo.conf
auth.mongo.auth_query.selector = broker_username=%u,
status=active
```

---

## 2. 主动断开设备连接

当禁用设备时，如果设备已经连入了IotHub，我们应该主动断开和这个设备的连接，MQTT协议并没有在协议级别约定Broker如何管理Client的连接，不过EMQ X Broker提供了一套RESTful的API来监控和管理Broker，我们可以在

<https://developer.emqx.io/docs/broker/v3/cn/rest.html>看到这些API文档，在这些接口中有一个可以管理Client连接的接口，我们可以调用这个接口主动断开设备连接。

EMQ X对接口的调用者需要进行身份验证，EMQ X使用HTTP Basic的认证方式对API的调用者进行认证。在调用这些API之前，需要创建一个账号，在EMQ X中称为创建一个Application。

创建Application的命令格式为：`mgmt insert<AppId><Name>`，这里我们用iothub作为AppId，将Name设为MaqueIotHub。

---

```
<EMQ X 安装目录>/emqx/bin/emqx_ctl mgmt insert iothub
MaqueIotHub
```

---

控制台输出如下所示。

---

```
AppSecret:
Mjg3NDc3MDc1MjgxNTM3OTg2MTYwNjYwMjMyOTU2ODA1MTC
```

---

我们需要记录下这个AppSecret。

EMQ X的管理API是通过插件的方式实现的，默认的访问地址是“http(s)://<host>:8080/api/v3/”，我们可以在“<EMQ X安装目录>/emqx/etc/plugins/emqx\_management.conf”进行配置，这个插件是默认加载的。

然后把这些信息都放到.env文件中。

---

```
1. IotHub_Server/.env
2. EMQ_X_APP_ID=iothub
3. EMQ_X_APP_SECRET=Mjg3NDc3MDc1MjgxNTM3OTg2MTYwNjYwMjMyOTU2ODA1MTC
4. EMQ_X_API_URL=http://127.0.0.1:8080/api/v3/
```

---

断开与某个Client连接的RESTful API的代码如下。

---

```
DELETE api/v3/connections/${clientid}
```

---

需要提供Client的ClientID作为参数，在设备连接状态管理里面已经保存了设备的ClientID，所以按需调用这个接口就可以。

首先把对EMQ X API的调用进行封装。

---

```
1. // IotHub_Server/services/emqx_service.js
2. "use strict";
3. const request = require('request');
```

---



```
4.
5. class EMQ XService {
6.   static disconnectClient(clientId) {
7.     const apiUrl = '${process.env.EMQ
X_API_URL}/connections/${clientId}'
8.     request.delete(apiUrl, {
9.       "auth": {
10.        'user': process.env.EMQ X_APP_ID,
11.        'pass': process.env.EMQ X_APP_SECRET,
12.        'sendImmediately': true
13.      }
14.    }, function (error, response, body) {
15.      console.log('statusCode:', response &&
response.statusCode);
16.      console.log('body:', body);
17.    })
18.  }
19. }
20.
21. module.exports = EMQ XService
```

---

然后在Device类里提供一个disconnect方法。

---

```
1. // IotHub_Server/models/device.js
2.
3. deviceSchema.methods.disconnect = function() {
4.   Connection.find({device: this._id}).exec(function
(err, connections) {
5.     connections.forEach(function (conn) {
6.       emqxService.disconnectClient(conn.client_id)
7.     })
8.   })
9. }
```

---

代码的第4~7行遍历了这个设备的所有连接，并依次断开。

### 3. Server API：禁用和恢复设备

接下来实现两个Server API接口来设置设备的接入状态，即禁用和恢复。

### (1) 禁用设备

---

```
1. // IotHub_Server/routes/devices.js
2. ...
3. router.put("/:productName/:deviceName/suspend",
function (req, res) {
4.   var productName = req.params.productName
5.   var deviceName = req.params.deviceName
6.   Device.findOneAndUpdate({"product_name":
productName, "device_name":
deviceName},
7.     {status: "suspended"}, {useFindAndModify:
false}).exec(function (err,
device) {
8.     if (err) {
9.       res.send(err)
10.    } else {
11.      if (device != null) {
12.        device.disconnect()
13.      }
14.      res.status(200).send("ok")
15.    }
16.  })
17. })
```

---

代码的第11行，在更新设备的status字段后，便主动断开与设备的连接。

### (2) 恢复设备

---

```
1. // IotHub_Server/routes/devices.js
2. ...
```

```
3. router.put("/:productName/:deviceName/resume",
function (req, res) {
4.   var productName = req.params.productName
5.   var deviceName = req.params.deviceName
6.   Device.findOneAndUpdate({"product_name":
productName, "device_name":
deviceName},
7.     {status: "active"}, {useFindAndModify:
false}).exec(function (err) {
8.     if (err) {
9.       res.send(err)
10.    } else {
11.      res.status(200).send("ok")
12.    }
13.  })
14. })
```

---

这时我们可以来试一下，首先暂停设备的接入认证

“`curl http://localhost:3000/devices/IotApp/c-j0c-2qq/suspend-X PUT`”，然后运行

“`IotHub_Device/samples/connect_to_server.js`”，控制台的输出如下。

---

```
Error: Connection refused: Not authorized
...
```

---

然后恢复这个设备的接入认证

“`curl http://localhost:3000/devices/IotApp/c-j0c-2qq/resume-X PUT`”，控制台的输出如下。

---

```
device is online
```

---

这时我们再运行

“`curlhttp://localhost:3000/devices/IotApp/c-j0c-2qq/suspend-X PUT`”，在运行“`connect_to_server.js`”的终端上会有以下输出。

---

```
device is online
device is offline
Error: Connection refused: Not authorized
...
```

---

(1) 设备连接上IoT Hub后，输出“device is online”。

(2) 被禁用后，连接被IoT Hub断开，输出“device is offline”。

(3) Device SDK的自动重连功能触发，因为设备已被禁用，所以在连接时输出“Error:Connection refused:Not authorized”。

## 7.3.2 删除设备

完成了禁用设备的功能后，删除设备对我们来说就非常简单了，依葫芦画瓢就可以了，基本思路如下。

- 1) 删除设备。
- 2) 删除设备的连接信息。
- 3) 将设备的所有连接断开。

### Server API: 删除设备

---

```
1. // IotHub_Server/routes/device.js
2.
3. router.delete("/:productName/:deviceName", function
(req, res) {
4.   var productName = req.params.productName
5.   var deviceName = req.params.deviceName
6.   Device.findOne({"product_name": productName,
"device_name": deviceName}).
  exec(function (err, device) {
7.     if (err) {
8.       res.send(err)
9.     } else {
10.      if (device != null) {
11.        device.remove()
12.        device.disconnect()
13.        res.status(200).send("ok")
14.      } else {
15.        res.status(404).json({error: "Not Found"})
16.      }
17.    }
  })
}
```

```
18.    })  
19.  })
```

---

同时，在删除设备时，删除所有的连接信息。

---

```
1. // IotHub_Server/models/device.js  
2.  
3. deviceSchema.post("remove", function (device, next) {  
4.   Connection.deleteMany({device: device._id}).exec()  
5.   next()  
6. })
```

---

这里我们使用Mongoose的回调钩子（Hook）来完成这个功能，当一个Device被删除时，就会触发Connection的删除动作（代码第4行）。

我们可以运行“curl<http://localhost:3000/devices/IotApp/c-j0c-2qq-X> DELETE”来看下效果。

本节完成了设备的禁用和删除，设备从注册、接入到移除的流程就完整了，在第7.4节，我们将设计和实现设备的权限管理。

## 7.4 设备权限管理

设备权限管理是指对一个设备的Publish（发布）和Subscribe（订阅）权限进行控制，设备只能发布到它有发布权限的主题上，同时它也只能订阅它有订阅权限的主题。

## 7.4.1 为什么要控制Publish和Subscribe

我们可以考虑一下以下的场景。

场景1: ClientA订阅主题command/ClientA来接收服务器端的指令, 这时ClientB接入, 同时也订阅command/usernameA, 那么在服务器向ClientA下发指令时, ClientB也能收到, 同时ClientB也可以将收到的指令再Publish到command/ClientA, ClientA无法肯定指令是否来自正确的对象。

场景2: ClientA向主题data/ClientA发布数据, 这时ClientB接入, 它也向data/ClientA发布数据, 那么data/ClientA的订阅者无法肯定数据来源是否为ClientA。

在这两个场景下都存在安全性和数据准确性的问题, 但是如果我们能控制Client的权限, 让ClientA才能订阅command/ClientA, 同时也只有ClientA才能发布到data/ClientA, 那么刚才的问题就都不存在了。

这就是我们需要对设备的Publish权限和Subscribe权限进行控制的原因。



## 7.4.2 EMQ X的ACL功能

EMQ X的ACL（权限管理）功能也是由插件实现的，我们使用的MongoDB认证插件就包含ACL功能，在第7.1节设备注册的章节里我们暂时关闭了这个功能，现在只需打开就可以了。

MongoDB插件的ACL功能很简单，Client在Publish和Subscribe的时候会查询一个Collection，找到该Client对应的ACL记录，这条记录应该包含如下3个字段。

```
publish:["topic1","topic2",...]
```

```
subscribe:["subtop1","subtop2",...]
```

```
pubsub:["topic/#","topic1",...]
```

publish字段代表该Client可以Publish的主题列表，subscribe字段代表该Client可以Subscribe的主题列表，pubsub字段代表Client可以同时Subscribe和Publish的主题列表，列表里的主题名可以使用通配符+和#，接下来我们来验证一下。

### 1. 配置MongoDB认证插件的ACL功能

首先编辑<EMQ X安装目录

>/emqx/etc/plugins/emqx\_auth\_mongo.conf。

1) “auth.mongo.acl\_query=on”，打开ACL功能；

2) “auth.mongo.acl\_query.collection=mqtt\_acl”，在mqtt\_acl collection中存放ACL信息；

3) “auth.mongo.acl\_query.selector=username=%u”，指明查询ACL记录的条件，%u代表Client接入时使用的用户名。

然后重新加载插件。

---

```
<EMQ X 安装目录>/emqx/bin/emqx_ctl plugins reload  
emqx_auth_mongo
```

---

接着在MongoDB里创建mqtt\_acl collection，但是暂时不添加任何记录。

---

```
1. ## MongoDB Shell  
2.  
3. db.createCollection("mqtt_acl")
```

---

用代码对Publish的权限进行验证，首先实现一个Subscribe端，使用JWT接入。

---

```
1. // IotHub_Device/samples/test_mqtt_sub.js
2. var jwt = require('jsonwebtoken')
3. var mqtt = require('mqtt')
4. require('dotenv').config()
5. var username = "username"
6. var password = jwt.sign({
7.   username: username,
8.   exp: Math.floor(Date.now() / 1000) + 10
9. }, process.env.JWT_SECRET)
10. var client = mqtt.connect('mqtt://127.0.0.1:1883', {
11.   username: username,
12.   password: password
13. })
14. client.on('connect', function (connack) {
15.   console.log('return code: ${connack.returnCode}')
16.   client.subscribe("/topic1")
17. })
18.
19. client.on("message", function (_, message) {
20.   console.log(message.toString())
21. })
```

---

然后实现一个Publish端，用已注册的设备的信息接入。

---

```
1. // IotHub_Device/samples/test_mqtt_pub.js
2.
3. var mqtt = require('mqtt')
4. require('dotenv').config()
5.
6. var client = mqtt.connect('mqtt://127.0.0.1:1883', {
7.   username:
8.   ${process.env.PRODUCT_NAME}/${process.env.DEVICE_NAME},
9.   password: process.env.SECRET
10. })
11. client.on('connect', function (connack) {
12.   console.log('return code: ${connack.returnCode}')
13.   client.publish("/topic1", "test", console.log)
14. })
```

---

先运行“test\_mqtt\_sub.js”，然后再运行“test\_mqtt\_pub.js”，我们可以看到“test\_mqtt\_sub.js”输出“test”，说明EMQ X在查询不到对应的ACL记录时，对Client的Publish和Subscribe权限是不限制的。我们可以在“<EMQ X安装目录>/emqx/etc/emqx.conf”中对这一行为进行修改。

---

```
1. ## Allow or deny if no ACL rules matched.
2. ##
3. ## Value: allow | deny
4. acl_nomatch = deny
```

---

这个配置项是指在没有查询到对应的ACL记录的情况下，对Client的Publish/Subscribe的处理，这里我们把配置从默认的allow（允许）改成了deny（拒绝）。

然后我们在mqtt\_acl中插入一条记录。

---

```
"username" : <device的broker_username>,
"publish" : [],
"subscribe" : [],
"pubsub" : []
```

---

这条ACL记录限制了设备，不允许设备Publish和Subscribe任何主题。

然后再重新运行“test\_mqtt\_pub.js”，这时“test\_mqtt\_sub.js”不会再输出“test”，说明EMQ X已经对Client的Publish权限进行了限制。

这里的Publish操作并没有报错，从安全性的角度来说，忽略未授权的操作比返回错误信息要好，我们可以在“<EMQ X安装目录>/emqx/etc/emqx.conf”中对这一行为进行修改：

---

```
1. ## The action when acl check reject current operation
2. ##
3. ## Value: ignore | disconnect
4. ## Default: ignore
5. acl_deny_action = ignore
```

---

如果将acl\_deny\_action的值改为disconnect，Broker将会断开和发起未授权的Publish或Subscribe的Client的连接。

接着修改这条ACL记录，将publish字段改为“[/topic1]”，然后重新运行“test\_mqtt\_pub.js”，这时“test\_mqtt\_sub.js”又会输出“test”，说明EMQ X已经按照ACL记录Client的Publish进行了限制。

Subscribe权限验证就不重复实验了，背后的逻辑是相同的。

## 2. ACL缓存

启用了ACL之后，EMQ X在Client Publish和Subscribe时都会查询MongoDB进行验证，这会带来额外的开销，需要进行平衡和选择，因为安全性和效率是冲突的。在一个完全可信的环境里，你也可以选择不打开ACL来提升效率，这一切都取决于你的使用场景，在IoT Hub中，我们会保持ACL的开启。

上面说ACL会导致EMQ X在Publish和Subscribe时查询MongoDB的结论不是完全准确，因为默认情况下EMQ X会缓存查询的结果，我们在“<EMQ X安装目录>/emqx/etc/emqx.conf”中可以找到这个cache的相关配置：

---

```
1. ## Whether to enable ACL cache.
2. ##
3. ## If enabled, ACLs roles for each client will be
   cached in the memory
4. ##
5. ## Value: on | off
6. enable_acl_cache = on
7.
8. ## The maximum count of ACL entries can be cached for
   a client.
9. ##
10. ## Value: Integer greater than 0
11. ## Default: 32
12. acl_cache_max_size = 32
13.
14. ## The time after which an ACL cache entry will be
   deleted
15. ##
16. ## Value: Duration
17. ## Default: 1 minute
18. acl_cache_ttl = 1m
```

---

你可以设置是否开启ACL缓存，ACL缓存可为单个Client缓存的最大ACL记录数，以及ACL缓存的过期时间。

通常，我们应该打开ACL缓存，并根据你的使用场景设定一个合理的ACL过期时间，这样可以减少很多不必要的数据库查询操作，因为大多数时候设备的可订阅主题与查询的主题不会频繁变动。

### 7.4.3 集成EMQ X ACL

接下来，我们把EMQ X的ACL功能集成到我们现有的体系当中：

- 事先定义好设备可以订阅和发布的主题范围；
- 在注册设备时，生成设备的ACL记录；
- 在删除设备时，删除相应的ACL记录。

我们使用名为device\_acl的collection保存设备的ACL记录。

---

```
1. // IotHub_Server/models/device_acl.js
2.
3. var mongoose = require('mongoose');
4. var Schema = mongoose.Schema;
5.
6. const deviceACLSchema = new Schema({
7.   broker_username: String,
8.   publish: Array,
9.   subscribe: Array,
10.  pubsub: Array,
11. }, { collection: 'device_acl' })
12.
13. const DeviceACL = mongoose.model("DeviceACL",
deviceACLSchema);
14.
15. module.exports = DeviceACL
```

---



这里，我们可以定义一个方法返回某个设备可以订阅和发布的主题，在第8章和第9章讲到处理上行数据和下发指令时，再来规划设备可以订阅和发布的主题，所以目前暂时将设备的可订阅和可发布主题列表设为空。

---

```
1. // IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = []
4.   const subscribe = []
5.   const pubsub = []
6.   return {
7.     publish: publish,
8.     subscribe: subscribe,
9.     pubsub: pubsub
10.  }
11. }
```

---

然后在设备注册的时候，根据这个方法的返回值生成ACL记录。

---

```
1. // IotHub_Server/routes/devices.js
2. router.post("/", function (req, res) {
3.   ...
4.   device.save(function (err) {
5.     if (err) {
6.       res.status(500).send(err)
7.     } else {
8.       var aclRule = device.getACLRule()
9.       var deviceACL = new DeviceACL({
10.         broker_username: device.broker_username,
11.         publish: aclRule.publish,
12.         subscribe: aclRule.subscribe,
13.         pubsub: aclRule.pubsub
14.       })
15.       deviceACL.save(function () {
16.         res.json({product_name: productName,
17. device_name: deviceName,
```

```
    secret: secret}))
17.         })
18.     }
19. })
20. })
```

---

删除设备的时候也移除相应的ACL记录：

---

```
1. // IotHub_Server/models/device.js
2. deviceSchema.post("remove", function (device, next) {
3.     Connection.deleteMany({device: device._id}).exec()
4.     DeviceACL.deleteMany({broker_username:
device.broker_username}).exec()
5.     next()
6. })
```

---

由于EMQ X在验证设备权限的时候，会根据broker\_username字段来查询设备的ACL记录，因此这个字段还需要加上索引。

---

```
1. ## MongoDB Shell
2. use iotHub
3. db.devices.createIndex({
4.     "broker_username" : 1,
5. })
```

---

这样，IotHub的设备权限管理框架就完成了。

本节完成了设备权限管理框架，设备具体可订阅和可发布的主题会在后面的章节里讲解。到此，我们就完成了Maque IotHub的设备从

注册到删除的全生命周期管理，第7.4节将讨论一个和设备管理无关的内容——IoT Hub的扩展性。

## 7.5 给IotHub加一点扩展性

前面我们讨论了设备接入的抽象问题和生命周期的管理问题，在继续深入前，我们先讨论一下IotHub的扩展性问题，毕竟作为一个物联网平台，能够承载大量的设备接入是非常重要的，我想这也是大家非常关心的问题。

在这里，我把问题稍微改一下，从IotHub能容纳多少设备接入，改成IotHub能不能随着业务扩容来满足业务需求。也就是说，IotHub是否具有可扩展性？

现在IotHub的组成部分有：基于Express的Web API、MongoDB和EMQ X。如何扩展Web服务和搭建多节点的MongoDB在本节中就不赘述了，只要清楚Web服务和MongoDB都具备良好的扩展性就可以了。本节主要讨论如何扩展EMQ X Broker。

## 7.5.1 EMQ X的纵向扩展

纵向扩展是指单机如何更多地接入设备。根据EMQ X官网的信息，EMQ X号称单机可接入百万设备。

EMQ X消息服务器1.x版本的MQTT连接压力测试的峰值是130万，在一台8核心、32G内存的CentOS服务器上。

我不打算验证这个说法，因为可接入数量不仅取决于你的硬件、网络和功能（比如，如果加载了MongoDB插件，那么MongoDB的配置也会对Broker性能有影响），还取决于你的测试资源，你可能需要一些测试服务器，通过并发的方式来模拟大量Client的接入。这里只简单介绍如何Tuning和测试的方法。

EMQ X提供了《Tuning Guide》，里面详细列出了需要修改的配置项，主要包括：

- 修改操作系统参数，提高可打开的文件句柄数；
- 优化TCP协议栈参数；
- 优化Erlang虚拟机参数，提高Erlang Process限制；
- 修改EMQ X配置，提高最大并发连接数。

你可以在EMQ X官网找到全部的配置文档。

EMQ X还提供了一个压测工具EMQTT Benchmark，你需要配置运行压测工具的电脑和服务器，使单机能够创建更多的连接。如果你希望测试的并发数很大，可能还需要准备多个电脑或者服务器来运行压测工具。

你可以在[https://github.com/emqtt/emqtt\\_benchmark](https://github.com/emqtt/emqtt_benchmark)找到这个测试工具。

我的建议是加载所有需要的插件，启用MQTTS（如果业务需要的话），在与生产环境一致的硬件、操作系统和软件环境下进行测试。最好内存够大，因为大量的连接会占用较多内存。

在一个8核32GB的Ubuntu服务器上，在加载了全部的应用和扩展插件并使用MQTTS的情况下，我的数据是15~20万Client接入，系统的负载还比较平稳。

## 7.5.2 EMQ X的横向扩展

单机的容量不管怎么扩展，总是有上限的，EMQ X还支持由多个节点组成集群。随着业务的发展，我们可以接入新的EMQ X节点来扩容。理论上，横向扩展是无上限的。

这里我会简单展示一下在Manual方式下建立一两个节点的EMQ X集群，以及节点如何退出集群。

除了手动的方式以外，EMQ X还支持通过etcd、Kubernetes、DNS等方式实现集群自动发现，具体可以查看EMQ X的集群配置文档：  
<https://developer.emqx.io/docs/broker/v3/cn/cluster.html>。

我准备了两台运行Ubuntu18.04的服务器：

- EMQ X\_A，公网IP为52.77.224.83，内网IP为172.31.25.69；
- EMQ X\_B，公网IP为54.255.237.124，内网IP为172.31.22.189。

### 1. 加入集群

首先，在EMOX\_A和EMOX\_B上安装EMQ X 3.2.0。

然后，需要在这两台服务器上分别配置Erlang节点名，编辑“<EMQ X安装目录>/emqx/etc/emqx/emqx.conf”。

```
EMQ X_A:node.name=emqx@172.31.25.69。
```

```
EMQ X_B:node.name=emqx@172.31.22.189。
```

最后，启动EMQ X Broker。

在EMQ X\_B上运行“<EMQ X安装目录>/emqx/bin/emqx\_ctl cluster join emqx@172.31.25.69”，如果不出意外，会得到以下输出。

---

```
[EMQ X] emqx shutdown for join
Join the cluster successfully.
Cluster status: [{running_nodes,
['emqx@172.31.25.69','emqx@172.31.22.189']}]
```

---

这说明集群已经搭建成功了。

在EMQ X\_A上运行“emqx\_ctl cluster join emqx@172.31.22.189”效果是一样的。

接下来，我们来看一下集群的效果，用一个MQTT Client连接到EMQ X\_A并订阅一个主题。

---

```
mosquitto_sub -h 52.77.224.83 -t "/topic1" -v
```



---

用一个MQTT client连接到EMQ X\_B，并向这个主题发布一个消息。

---

```
mosquitto_pub -h 54.255.237.124 -t "/topic1" -m "test"
```

---

如果订阅端输出“/topic1 test”，就说明这个集群的工作是正常的。

节点间直接通过“Cookie”的方式认证，在“<EMQ X安装目录>/emqx/etc/emqx.conf”中使用“node.cookie=emqxsecretcookie”进行配置，集群中节点的Cookie需保持一致。

## 2. 退出集群

可以在节点上运行“emqx\_ctl cluster leave”退出集群。或者通过“emqx\_ctl cluster force-leave emqx@172.31.25.69”的方式强制EMQ X\_A退出集群。

## 7.6 本章小结

本章我们学习了如何横向和纵向扩展EMQ X Broker。

通常我们会在EMQ X集群前部署一个Load Balancer，所有的Client都使用Load Balancer的地址建立连接，Load Balancer再与集群里的各个节点建立连接并传输数据。

一般，我们可以从一个单节点或者双节点的EMQ X集群开始，使用压测工具测试配置是否满足业务的要求，然后随着业务的变化，再往这个集群里添加或者减少节点，进而实现EMQ X的扩展性。

我们可以看到，IoTHub的组件都是可扩展的，所以IoTHub也是可以扩展的，它具有很好的可扩展性。

本章实现了设备的全生命周期管理，为后续内容打下了基础，同时也验证了IoTHub的扩展性。接入来，我们一起进入第8章的学习。

## 第8章 上行数据处理

作为一个物联网平台，首先要做的就是可以接收并处理设备上传的数据，我们称之为上行数据。

本章将设计和实现IoTHub的上行数据处理模块，它应有如下功能。

- 存储上行数据：IoTHub接收设备端上传的数据，并将数据来源（设备的ProductName, DeviceName）、消息ID、消息类型、payload进行存储。
- 通知业务系统：当有新的上行数据到达时，IoTHub将通知并将上行数据发送给业务系统，业务系统可以自行处理这些数据，例如通知用户，将数据和其他业务数据融合后存储在业务系统的数据库等。
- 设备数据查询：业务系统可以通过IoTHub Server API查询某个设备上传的历史数据。

## 8.1 选择一个可扩展的方案

IotHub应该怎么接收来自设备的数据呢？

这也是我被问到的比较多的一个问题：“MQTT协议服务端怎么接收客户端发送的数据呢？”

这里我想先做一个说明，在MQTT协议的架构里，是没有“服务端”和“客户端”的概念的，只有Broker和MQTT Client，所以EMQ X说自己是一个MQTT Broker，而不是MQTT的Server。而服务端和客户端，是我们在MQTT协议的基础上构建的C/S结构的平台或者业务系统里面的概念，所以我们需要做一些抽象，让这两组不太相干的逻辑实体能做到匹配。

在我们的架构里，IotHub Server就是服务端，设备就是客户端，IotHub Server有一个最基础的功能就是接收设备的数据并进行存储，那么怎么实现呢？我们来看一下可能的几种方案。

### 8.1.1 完全基于MQTT协议的方案

这个方案使用MQTT协议框架内的实体来实现设备上行数据的接收功能。

像前面说的一样，MQTT协议架构里没有“服务端”和“客户端”，那么IotHub Server需要接收设备端的数据，它需要和设备一样，以MQTT Client的身份接入EMQ X Broker，订阅相关的主题来获取数据。

- 1) 设备端发布消息到特定主题，例如“data/client/:DeviceName”。
- 2) IotHub Server启动一个MQTT Client，接入EMQ X Broker，并订阅主题“data/client/+”。
- 3) IotHub Server的MQTT Client接收到消息后，将消息存入数据库。

这是一个可行的方案，但是依然存在单点故障问题：服务端的MQTT Client挂了怎么办？当数据量很大时，这个Client是否能够处理得过来，这会不会成为系统的瓶颈？

我们可以往前走一步，将设备进行分片。设备在发布的时候先随机生成一个1~20的随机整数SliceID（这里假设要分20片），然后设备将数据发布到“data/client/:SliceId/:DeviceName”。在IoT Hub Server端，可以启动最多20个MQTT Client，分别订阅

“data/client/1/+”到“data/client/20/+”这20个主题。通过这样的方式，设备在服务端将数据流从一个Client，分散到了最多20个Client，减少了这部分成为系统瓶颈的可能性。

这个解决方案仍然有一个问题，当IoT Hub Server端的某一个MQTT Client挂了以后，有一部分设备的数据上传就会受到影响，直到这个MQTT Client恢复为止。

我们还可以更进一步。EMQ X Broker支持一个共享订阅功能，多个订阅者可以订阅同一个主题，EMQ X Broker会按照某种顺序依次把消息分发给这些订阅者，在某种意义上实现订阅者负载均衡。

共享订阅的实现很简单，订阅者只需要订阅具有特殊前缀的主题即可，目前共享订阅支持2种前缀“\$queue/”和

“\$share/<group>/”，且支持通配符“#”和“+”，我们可以做个实验。

在两个终端上运行mosquitto\_sub-t '\$share/group/topic/+'，并在第三个终端上运行mosquitto\_pub-t "topic/1"-m "test"--repeat

10, 我们会发现在运行mosquitto\_sub的两个终端上会分别打印出test, 加起来一共10次。

- 1) \$share/group/topic/+中group可以为任何有意义字符串;
- 2) 在发布的时候不再需要加上共享订阅的前缀;
- 3) 这里为了方便验证, 将EMQ X设置为允许匿名登录;
- 4) 共享订阅有多种分发策略, 可以在<EMQ X安装目录>/emqx/etc/emqx.conf中修改配置项shared\_subscription\_strategy, 对分发的策略进行配置。

下面补充上述方案中涉及的几个参数。

- random: 默认值, 所有共享订阅者随机选择分发;
- round\_robin: 按照共享订阅者订阅的顺序分发;
- sticky: 分发给上次分发的订阅者;
- hash: 根据发布Client的ClientID进行分发。

如果使用共享订阅的方式实现服务端接收设备端数据, 我们就可以根据数据量动态的增添共享订阅者, 这样就不存在单点故障了, 也具有有良好的扩展性。唯一的缺点是, 这种方案引入了多个MQTT Client

这样额外的实体，提高了系统的复杂性，增加了开发、部署和运维监控的成本。



## 8.1.2 基于Hook的方案

这个方案会使用EMQ X的Hook机制实现设备上行数据的接收功能。

在第7.2节设备连接状态管理中，我们已经了解EMQ X的Hook设计，并用到了EMQ X自带的WebHook插件。和设备上线与下线一样，EMQ X会在收到Publish数据包时将Publish的信息通过Hook传递出来。

---

```
1. {
2.   "action":"message_publish",
3.   "from_client_id":"C_1492410235117",
4.   "from_username":"C_1492410235117",
5.   "topic":"world",
6.   "qos":0,
7.   "retain":true,
8.   "payload":"Hello world!",
9.   "ts":1492412774
10. }
```

---

这时，我们就可以对数据进行存储和处理，实现接收设备的上行数据。基于Hook的方案不用在Server端建立和管理连接到Broker的MQTT Client，系统复杂度要低一些，Maque IoT Hub会使用基于Hook的方案实现上行数据的接收。

基于共享订阅和基于Hook的方式都是在生产环境可以用的解决方案，Maque IoT Hub使用基于Hook的方式只是因为这样开发和部署要简

单一些，而不是因为基于Hook的方案相较于共享订阅有明显的、决定性的优势。

### 8.1.3 数据格式

接下来让我们来定义一下上行数据的格式，在IoTHub里，上行数据由两部分组成：负载和元数据。

- 负载：负载（Payload）是指消息所携带的数据本身，比如传感器在某一时刻的读数。负载可以是任意格式，例如JSON字符，或者二进制数据。它的格式由业务系统和设备之间约定，IoTHub不对负载进行解析，只负责在业务系统和设备之前传递负载数据并存储。这部分数据包含在Publish数据包的消息体中。

- 元数据：元数据（Metadata）是指描述消息的数据，包括消息发布者的身份（ProductName, DeviceName）、数据类型等，在IoTHub中，上行数据会使用QoS1，所以还需要在接收端对消息进行手动去重，那么在元数据中还会包含消息的唯一ID，在后面我们还会看到更多的元数据类型。元数据的内容是包含在Publish数据包的Topic Name里面的，IoTHub会对元数据进行解析，以便做后续的处理。

## 8.1.4 主题名规划

从这里开始，我们需要对IoTHub的设备订阅或者发布的主题名进行规划，设备会发布、订阅很多主题，这里暂时不一起规划完，而是一步一步地进行规划。

如8.1.3节所说，我们会把上行数据的元数据放在主题名里，那么设备发布的主题名格式为：

upload\_data/:ProductName/:DeviceName/:DataType/:MessageID，其中各个字段的含义如下所示。

- ProductName：设备的产品名。
- DeviceName：设备名。
- DataType：上传的数据类型，这个由业务系统和设备约定。比如传感器的温度数据，可以设置DataType为temperature，在主题名中添加这一层级的目的是使主题名尽量精确。（这是一个MQTT主题命名的最佳实践。）
- MessageID：每个消息的唯一ID。

假设来自设备的Publish数据包的主题名为：

---

```
upload_data/IotApp/ODrvBHNaY/temperature/5ce4e36de3522c03b48a8f7f,
```

---

那么通过解析这个主题名，IotHub就可以获取该条消息的元数据：消息为设备上传的数据，来自设备（IotAPP，ODrvBHNaY），数据类型为temperature，消息ID为5ce4e36de3522c03b48a8f7f。

## 8.1.5 上行数据存储

IotHub仍然使用MongoDB存储设备上传的数据，这里我们来定义一些用于存储设备上传数据的数据库模型。

---

```
1. //IotHub_Server/models/message.js
2.
3. var mongoose = require('mongoose');
4. var Schema = mongoose.Schema;
5.
6. const messageSchema = new Schema({
7.   message_id: String,
8.   product_name: String,
9.   device_name: String,
10.  data_type: String,
11.  payload: Buffer,
12.  sent_at: Number
13. })
14.
15. const Message = mongoose.model("Message",
messageSchema);
16.
17. module.exports = Message
```

---

因为payload可以是任意类型的数据，例如字符串或者二进制数据，所以这里将它定义为buffer类型。

消息可以根据message\_id或者（ProductName, Device）查询，所以需要创建相应的索引。

---

```
1. # MongoDB Shell
2. use iotHub
3. db.messages.createIndex({
4.   "production_name" : 1,
5.   "device_name" : 1
6. })
7. db.messages.createIndex({
8.   "message_id" : 1
9. })
```

---

### 8.1.6 通知业务系统

来自设备的数据到达IoTHub后，IoTHub需要通知对应的业务系统。

实际上有很多种方式可以在新的上行数据到达时通知业务系统，比如调用业务系统预先注册的回调URL、使用队列系统等，这属于软件层面的架构设计。本书选择了一种简单的方式进行演示，IoTHub将使用RabbitMQ进行通知，当有新的上行数据到达时，IoTHub会向相应的Exchange发布一条包含设备上行数据负载的消息。



### 8.1.7 上行数据查询

Server API可提供接口供业务系统查询存储在IotHub上的设备上  
行数据，我们可以通过MessageID (ProductName, DeviceName) 进行  
查询。

由于负载可以是任意二进制数据，所以通过HTTP接口返回payload  
内容时需要进行编码，Server AIP使用Base64进行负载与编码。

### 8.1.8 上行数据处理流程

综合上文的设计，我们可以画出IoTHub上行数据处理的流程，如图8-1所示。

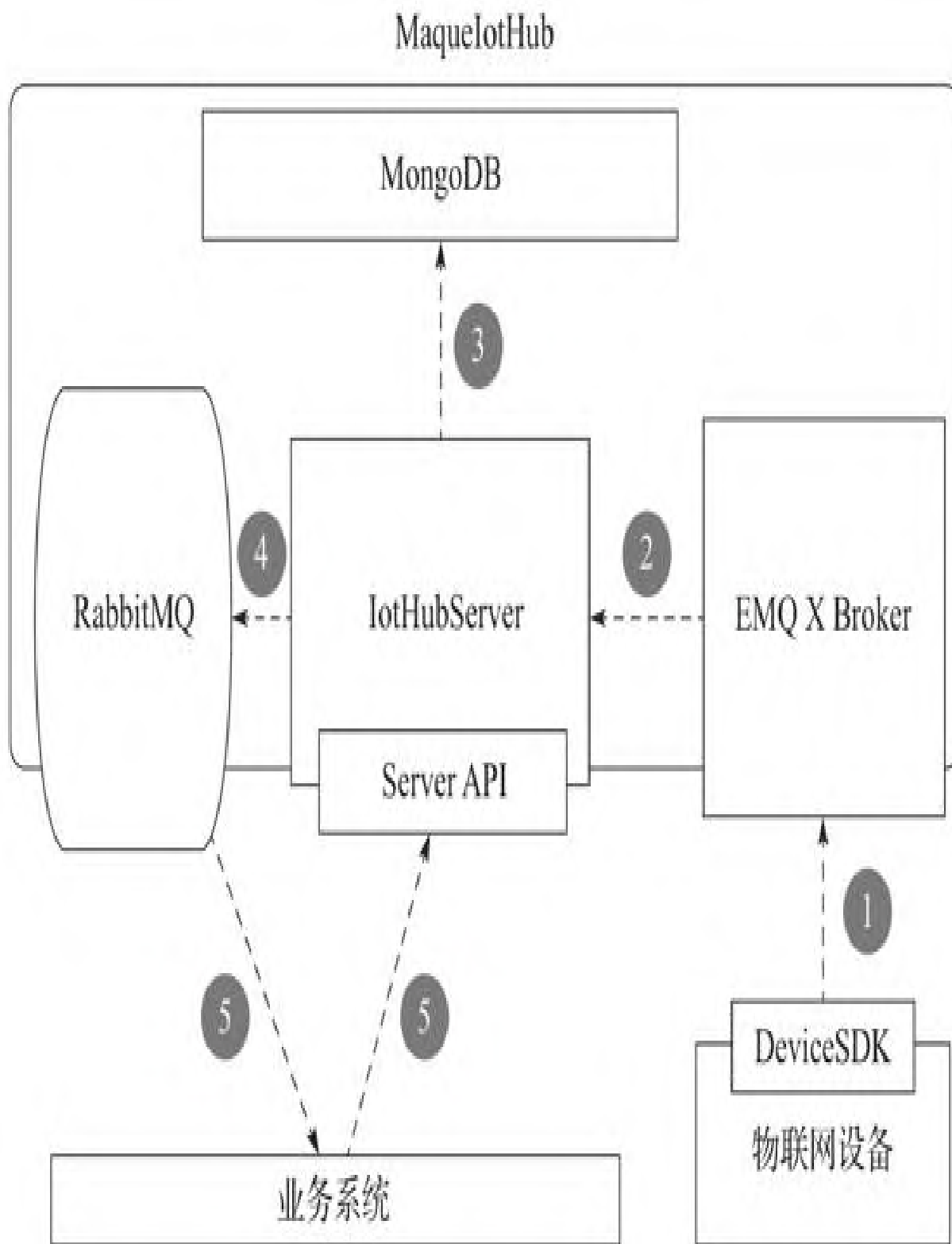


图8-1 IotHub的上行数据处理流程

1) 物联网设备调用DeviceSDK的接口将数据发布到  
“upload\_data/:ProductName/:DeviceName/:DataType/:MessageID”  
(MessageID由DeviceSDK生成)。

2) EMQ X Broker通过WebHook将消息传递给Iothub Server。

3) IotHub Server将消息存储到MongoDB。

4) IotHub Server将数据放入对应的RabbitMQ队列。

5) 业务系统从RabbitMQ获取新的上行数据，业务系统也可以调用  
Server API提供的接口查询设备的上行数据。

本节确定了IotHub上行数据处理的方案，并设计了上行数据处理的流程。接下来，我们将按照这个设计实现IotHub的上行数据处理功能。

## 8.2 实现上行数据处理功能

本节将实现IoTHub的上行数据处理功能，包括设备端和服务端的实现。

在设备端我们会实现数据上传的接口，在服务端我们会实现元数据提取、消息去重、存储等功能。

## 8.2.1 DeviceSDK的功能实现

### 1. 实现uploadData接口

DeviceSDK端的实现比较简单，实现一个uploadData方法供设备应用代码调用，按照第8.1节约定的主题名发布数据就可以了。在发布数据时，DeviceSDK负责为消息生成唯一的MessageID，生成的算法有很多选择，最简单的就是用UUID，在本书中，我们使用BSON的ObjectID作为MessageID。

BSON（Binary JSON）是一种类JSON的二进制存储格式，我们在其他章节还会用到BSON。

---

```
1. //IotHub_Device/sdk/iot_device.js
2.
3. const objectId = require('bson').ObjectID;
4. class IotDevice extends EventEmitter{
5.     ...
6.     uploadData(data, type="default"){
7.         if(this.client != null){
8.             var topic =
'upload_data/${this.productName}/${this.deviceName}/
${type}/${new ObjectId().toHexString()}'
9.             this.client.publish(topic, data, { qos: 1
10.         })
11.     }
12. }
```

---

代码的第8行，按照之前约定的格式拼接生成主题名。

## 2. 使用可持久化的Message Store

MQTT Client在发布QoS>1的消息时，会先在本地存储这条消息，等收到Receiver的ACK后，再删除这条消息。同时，在现在大部分的Client实现里，也会把还没有发布出去的消息缓存在本地，这样的话即使Client和Broker的连接因为网络问题断开，也可以继续调用publish方法，在恢复连接之后，再把这部分消息依次发布出去。这些消息被称为in-flight消息，用于存储in-flight消息的地方叫Message Store。

在DeviceSDK里，in-flight消息是存储在内存里的，这是有问题的：设备断电之后，in-flight消息就都丢了。所以我们需要可持久化的Message Store。

Node.js版的MQTT Client有几种支持可持久化的Message Store：mqtt-level-store、mqtt-nedb-store和mqtt-localforage-store，这里我们选择mqtt-level-store作为可持久化的Message Store。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. var levelStore = require('mqtt-level-store');
3. constructor({serverAddress = "127.0.0.1:8883",
productName, deviceName,
secret, clientID, storePath} = {}) {
4.     ...
5.     if(storePath != null) {
```

```
6.         this.manager = levelStore(storePath);
7.     }
8. }
9.
10. connect() {
11.     var opts = {
12.         rejectUnauthorized: false,
13.         username: this.username,
14.         password: this.secret,
15.         clientId: this.clientIdentifier,
16.         clean: false
17.     };
18.     if(this.manager != null){
19.         opts.incomingStore = this.manager.incoming
20.         opts.outgoingStore = this.manager.outgoing
21.     }
22.     this.client = mqtt.connect(this.serverAddress,
23.     ...
24. }
```

---

代码的第3行新增了storePath参数，代码的第6行用这个参数初始化了levelStore。

代码的第18~22行将之前初始化的levelStore传递给了MQTT Client，作为Message Store。

大多数语言的MQTT Client库都有类似的持久化Message Store实现，所以你在其他语言或者平台上开发时，需要找到或者实现对应的持久化Message Store。



## 8.2.2 IotHub Server的功能实现

IotHub Server在接收到上行数据时候需要做以下几步处理：

- 从主题名中提取出上行数据的元数据；
- 消息去重；
- 将消息进行存储；
- 通过RabbitMQ通知业务系统。

### 1. 配置WebHook的payload编码

前面提过，因为我们使用的是WebHook，所以需要对payload进行Base64编码，以应对payload是二进制的情况。我们首先需要对WebHook插件进行配置。

---

```
## <EMQ X 安装目录>/emqx/etc/plugins/emqx_web_hook.conf
web.hook.encode_payload = base64
```

---

然后运行“<EMQ X安装目录>/emqx/bin/emqx\_ctl plugins reload emqx\_web\_hook”。重新加载WebHook插件。

### 2. 提取元数据

接下来我们需要在类似

“upload\_data/IotApp/ODrvBHNaY/temperature/5ce4e36de3522c03b48a8f7f” 的主题名中将消息的元数据提取出来，这样的操作用正则表达式进行模式匹配是最好的，不太会写正则表达式的读者也不用担心，我们可以使用path-to-regexp按照预先定义好的规则生成对应的正则表达式。

path-to-regexp是一个Node.js包，可以根据预先设定的路径规则生成对应的、用于匹配的正则表达式，path-to-regexp的输出是一个通用的正则表达式，所以不用Node.js也可以用path-to-regexp按照主题名规则预先生成正则表达式，然后复制到你的代码里使用。在这里每次都重新生成正则表达式，这样的代码更好读一些。

---

```
1. //IotHub_Server/services/message_service.js
2. const pathToRegexp = require('path-to-regexp')
3. class MessageService {
4.     static dispatchMessage({topic, payload, ts} = {}){
5.         var dataTopicRule =
"upload_data/:productName/:deviceName/:dataType
/:messageId";
6.         const topicRegx = pathToRegexp(dataTopicRule)
7.         var result = null;
8.         if ((result = topicRegx.exec(topic)) != null)
{
9.             var productName = result[1]
10.            var deviceName = result[2]
11.            var dataType = result[3]
12.            var messageId = result[4]
13.            //接下来对上行数据进行处理
14.            ...
15.        }
16.    }
```

```
17. }  
18. module.exports = MessageService
```

---

这里我们新建了一个service类来处理消息的分发。

代码的第5行是我们定义路径的规则，第6行使用了path-to-regexp生成对应的正则表达式。

代码的第8行用这个正则表达式来匹配主题名，如果匹配成功，就可以从匹配结果的数组中依次提取productName、device Name、dataType和messageId的值。

### 3. 消息去重

消息去重的原理比较简单，我们使用Redis存储已接收到的消息的MessageID，当收到一条新消息时，先检查Redis，如果MessageID已存在，则丢弃该消息，如果MessageID不存在，则将该消息的MessageID存入Redis，然后进行后续的处理。

创建Redis连接。

---

```
1. //IotHub_Server/models/redis.js  
2. const redis = require('redis');  
3. const client =  
  redis.createClient(process.env.REDIS_URL);  
4. client.on("error", function (err) {  
5.   console.log("Error " + err);  
6. });  
7. module.exports = client;
```

---

检查messageId。

---

```
1. //IotHub_Server/services/message_service.js
2. const redisClient = require("../models/redis")
3.
4. class MessageService {
5.   ...
6.
7.   static checkMessageDuplication(messageId, callback)
8.   {
9.     var key = '/messageIDs/${messageId}'
10.    redisClient.setnx(key, "", function (err, res) {
11.      if (res == 1) {
12.        redisClient.expire(key, 60 * 60 * 6)
13.        callback.call(this, false)
14.      } else {
15.        callback.call(this, true)
16.      }
17.    })
18.  }
19.  ...
```

---

代码的第9~14行使用setnx命令往Redis插入由MessageID组成的key，如果setnx返回1，则说明key不存在，那么消息可以进行后续处理；如果返回0，则说明MessageID已经存在，应该要丢掉这个消息。

在这里我们给key设置了6个小时的有效期，理论上来说，你应该永久保存已接收到的MessageID，但是存储MessageID的空间不是无限的，它需要一种清理机制，在实际项目中，6个小时的有效期已经可以应对99.9%的情况，你也可以根据实际情况自行调整。

## 4. 消息存储

消息存储非常简单，这里我们仍然做一个封装。

---

```
1. //IotHub_Server/services/message_service.js
2.
3. static handleUploadData({productName, deviceName, ts,
payload, messageId,
dataType} = {}) {
4.     var message = new Message({
5.         product_name: productName,
6.         device_name: deviceName,
7.         payload: payload,
8.         message_id: messageId,
9.         data_type: dataType,
10.        sent_at: ts
11.    })
12.    message.save()
13. }
```

---

### 8.2.3 代码联调

现在，我们把之前的代码都合并到一起。

---

```
1. //IotHub_Service/services/message_service.js
2. static dispatchMessage({topic, payload, ts} = {}) {
3.     var dataTopicRule =
"upload_data/:productName/:deviceName/:dataType/:
messageId";
4.     const topicRegx = pathToRegexp(dataTopicRule)
5.     var result = null;
6.     if ((result = topicRegx.exec(topic)) != null) {
7.         this.checkMessageDuplication(result[4], function
(isDup) {
8.             if (!isDup) {
9.                 MessageService.handleUploadData({
10.                     productName: result[1],
11.                     deviceName: result[2],
12.                     dataType: result[3],
13.                     messageId: result[4],
14.                     ts: ts,
15.                     payload: Buffer.from(payload, 'base64')
16.                 })
17.             }
18.         })
19.     }
20. }
```

---

代码的第15行使用`Buffer.from(payload, 'base64')`进行Base64解码。

然后在WebHook接口里面调用`dispatchMessage`方法。

---

```
1. //IotHub_Server/routes/emqx_web_hook.js
2.
3. router.post("/", function (req, res) {
4.   switch (req.body.action){
5.     case "client_connected":
6.       Device.addConnection(req.body)
7.       break
8.     case "client_disconnected":
9.       Device.removeConnection(req.body)
10.      break;
11.    case "message_publish":
12.      messageService.dispatchMessage({
13.        topic: req.body.topic,
14.        payload: req.body.payload,
15.        ts: req.body.ts
16.      })
17.    }
18.    res.status(200).send("ok")
19.  })
```

---

最后，我们需要在设备的ACL列表中加入这个主题，使设备有权限发布数据到这个主题中。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = [
4.
5.     'upload_data/${this.productName}/${this.deviceName}/+/'
6.   ]
7.   ...
8. }
```

---

代码的第4行拼接的主题名使用了2个“+”通配符，这是ACL允许的。

这些步骤完成后，我们可以写一段代码来验证下。我们需要事先重新注册一个设备来上传数据（或者你需要手动在MongoDB里更新已有的设备ACL列表）。

---

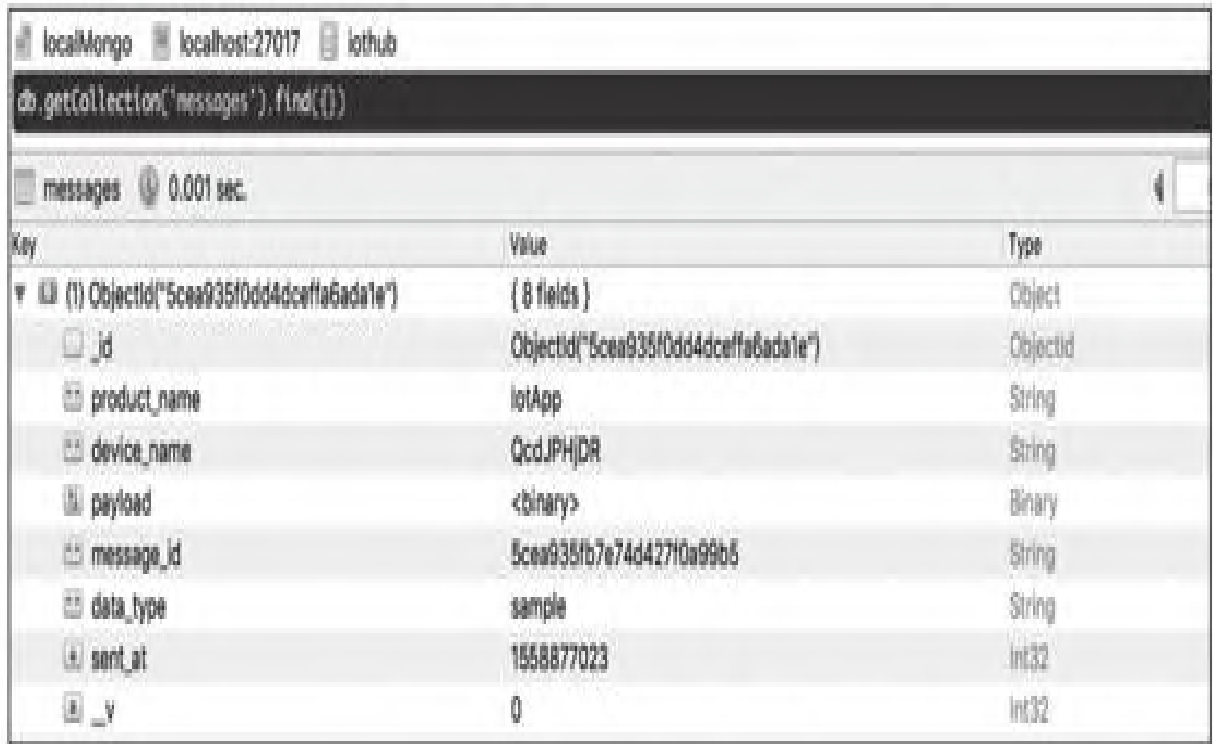
```
1. //IotHub_Device/samples/upload_data.js
2. var IotDevice = require("../sdk/iot_device")
3. require('dotenv').config()
4.
5. var device = new IotDevice({
6.   productName: process.env.PRODUCT_NAME,
7.   deviceName: process.env.DEVICE_NAME,
8.   secret: process.env.SECRET,
9.   clientID: path.basename(__filename, ".js"),
10.  storePath: '../tmp/${path.basename(__filename,
11.  ".js")}'
12. })
13. device.on("online", function () {
14.   console.log("device is online")
15. })
16. device.connect()
17. device.uploadData("this is a sample data", "sample")
```

---

在代码的第9~10行，我们使用JavaScript的文件名来命名Message Store和ClientID，这样的话sample目录下不同的JavaScript文件在运行时就不会产生冲突了。

运行upload\_data.js之后，查询IotHub数据库的Messages Collection，我们可以发现刚才发送的消息已经存进来了，如图8-2所示。





localMongo localhost:27017 lothub		
db.getCollection('messages').find({})		
messages 0.001 sec.		
Key	Value	Type
(1) ObjectId("5cea935f0dd4dceffa6ada1e")	{ 8 fields }	Object
_id	ObjectId("5cea935f0dd4dceffa6ada1e")	ObjectId
product_name	lotApp	String
device_name	QcdJPHJR	String
payload	<binary>	Binary
message_id	5cea935fb7e74d427f0a99b6	String
data_type	sample	String
sent_at	1568877023	Int32
_v	0	Int32

图8-2 保存到MongoDB的上行数据

检查IotHub\_Device/tmp/upload\_data目录，可以看到这个目录下生成了一些文件。这是用于持久化in-flight消息的文件。

## 8.2.4 通知业务系统

当上行数据到达IotHub时，IotHub可以通过RabbitMQ来通知并发送收到的上行数据给业务系统。

这里我们做一个约定：当有新的上行数据达到时，IotHub会向RabbitMQ名为`iothub.events.upload_data`的Direct Exchange的发送一条消息，RoutingKey为设备的ProductName。

这里我们使用`amqplib`作为RabbitMQ Client端实现。

关于RabbitMQ Routing相关的概念可以查看RabbitMQ Tutorials，在本书中就不赘述了。

我们仍然新增一个`service`类来处理通知业务系统的功能。

在程序启动时，初始化RabbitMQ Client，并确保对应的Exchange存在。

---

```
1. //IotHub_Server/services/notify_service.js
2. var amqp = require('amqplib/callback_api');
3. var uploadDataExchange = "iothub.events.upload_data"
4. var currentChannel = null;
5. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
6.   if (error0) {
7.     console.log(error0);
```

```
8.     } else {
9.         connection.createChannel(function (error1,
channel) {
10.             if (error1) {
11.                 console.log(error1)
12.             } else {
13.                 currentChannel = channel;
14.                 channel.assertExchange(uploadDataExchange,
'direct', {durable: true})
15.             }
16.         });
17.     }
18. });
```

---

然后实现通知业务系统的方法。

---

```
1. //IotHub_Server/services/notify_service.js
2. const bson = require('bson')
3. class NotifyService {
4.     static notifyUploadData(message) {
5.         var data = bson.serialize({
6.             device_name: message.device_name,
7.             payload: message.payload,
8.             send_at: message.sendAt,
9.             data_type: message.dataType,
10.            message_id: message.message_id
11.        })
12.        if(currentChannel != null) {
13.            currentChannel.publish(uploadDataExchange,
message.product_name, data, {
14.                persistent: true
15.            })
16.        }
17.    }
18. }
19. module.exports = NotifyService
```

---

代码的第13行使用BSON对上传数据的相关信息进行序列化后，再发送到相应的Exchange上，所以业务系统获取到这个数据以后需要先使用BSON反序列化。这是IotHub和业务系统之间的约定。

最后在接收到上行数据的时候调用这个方法。

---

```
1. //IotHub_Server/service/message_service.js
2. static handleUploadData({productName, deviceName, ts,
payload, messageId,
dataType} = {}) {
3.     var message = new Message({
4.         product_name: productName,
5.         device_name: deviceName,
6.         payload: payload,
7.         message_id: messageId,
8.         data_type: dataType,
9.         sent_at: ts
10.    })
11.    message.save()
12.    NotifyService.notifyUploadData(message)
13. }
```

---

接下来我们可以写一小段代码模拟业务系统从IotHub获取通知。

---

```
1. //IotHub_Server/business_sim.js
2. require('dotenv').config()
3. const bson = require('bson')
4. var amqp = require('amqplib/callback_api');
5. var uploadDataExchange = "iothub.events.upload_data"
6. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
7.     if (error0) {
8.         console.log(error0);
9.     } else {
10.        connection.createChannel(function (error1,
channel) {
```

---

```
11.         if (error1) {
12.             console.log(error1)
13.         } else {
14.             channel.assertExchange(uploadDataExchange,
15. 'direct', {durable: true})
16.             var queue = "iotapp_upload_data";
17.             channel.assertQueue(queue, {
18.                 durable: true
19.             })
20.             channel.bindQueue(queue, uploadDataExchange,
21. "IotApp")
22.             channel.consume(queue, function (msg) {
23.                 var data = bson.deserialize(msg.content)
24.                 console.log('received from
25. ${data.device_name}, messageId: ${data.
26. message_id},payload: ${data.payload.toString()}')
27.                 channel.ack(msg)
28.             })
29.         }
30.     });
31. }
```

---

运行business\_sim.js, 再运行

IotHub\_Device/samples/upload\_data.js, 可以看到在运行

business\_sim.js的终端上会输出:

---

```
received from QcdJPHjDR, messageId:
5ceb788f80124804aa1ea95b,payload: this is a sample data
```

---

这样就表示通知业务系统的功能已经完成了。

## 8.2.5 Server API历史消息查询

历史消息查询Server API的实现就很简单了，可以根据产品、设备和MessageID进行查询。

首先定义在接口中返回的消息的JSON格式。

---

```
1. //IotHub_Server/models/message.js
2. messageSchema.methods.toJSONObject = function () {
3.   return {
4.     product_name: this.product_name,
5.     device_name: this.device_name,
6.     send_at: this.send_at,
7.     data_type: this.data_type,
8.     message_id: this.message_id,
9.     payload: this.payload.toString("base64")
10.   }
11. }
```

---

代码的第9行对Payload进行了Base64编码，因为Payload有可能是二进制数据。然后实现查询的接口。

---

```
1. //IotHub_Server/routes/messages.js
2. var express = require('express');
3. var router = express.Router();
4. var Message = require('../models/message')
5.
6. router.get("/:productName", function (req, res) {
7.   var messageId = req.query.message_id
8.   var deviceName = req.query.device_name
9.   var productName = req.params.productName
10.   var query = {product_name: productName}
```

---

```
11.   if (messageId != null) {
12.       query.message_id = messageId
13.   }
14.   if (deviceName != null) {
15.       query.device_name = deviceName
16.   }
17.   Message.find(query, function (error, messages) {
18.       res.json({
19.           messages: messages.map(function (message) {
20.               return message.toJSONObject()
21.           })
22.       })
23.   })
24. })
25. module.exports = router
```

---

最后将接口挂载到对应的URL上。

---

```
1. //IotHub_Server/app.js
2. var messageRouter = require('./routes/messages')
3. app.use('/messages', messageRouter)
```

---

调用接口curl http://localhost:3000/messages/IotApp?  
device\_name\=QcdJPHjDR\&message\_id\=5ceb788f80124804aa1ea95b  
会有以下输出。

---

```
{ "messages":
  [{ "product_name": "IotApp", "device_name": "QcdJPHjDR", "data_
    _type": "sample", "message_id": "5ceb788f80124804aa1ea95b", "
    payload": "dGhpcyBpcyBhIHhXbXBsZSBkYXRh" }] }
```

---

要注意的是，接口返回的payload字段是用Base64编码的。

本节按照之前的设计方案，实现了IotHub的上行数据处理功能，包括IotHub Server和DeviceSDK的代码。8.3节将讨论并处理另外一种上行数据：设备状态上报。



## 8.3 设备状态上报

本节来讨论另外一种设备上行数据，即设备状态。

### 8.3.1 设备状态

8.2节实现了对设备上行数据的处理，假设我们有一台装有温度传感器的设备，那么它可以使用这个功能将每个时刻统计的温度数据上报到IoTHub，IoTHub会记录每一条温度数据并通知业务系统，业务系统可以自行存储温度数据，也可以使用IoTHub提供的接口来查询不同时刻的温度数据。

除了温度数据，设备可能还需要上报一些其他数据，比如当前使用的软件/硬件版本、传感器状态（有没有坏掉）、电池电量等，这些属于设备的状态数据，通常我们不会关心这些数据的历史记录，只关心当前状态，那么用前面实现的上报数据功能来管理设备的状态就有点不合适了。

IotHub需要对设备的状态进行单独处理，我们按如下方式设计IoTHub的设备状态管理功能。

- 1) 设备用JSON的格式将当前的状态发布到主题  
`update_status/:productName/:deviceName/:messageId`。
- 2) IoTHub将设备的状态用JSON的格式存储在Devices Collection中。

3) IotHub将设备的状态通知到业务系统，业务系统再做后续的处理，比如通知相关运维人员等。

4) IotHub提供接口供业务系统查询设备的当前状态。

为了对消息进行去重，设备状态消息也会带MessageID。

在这里我们做一个定义，设备上报的状态一定是单向的，状态只在设备端更改，然后设备上报到IotHub，最后由IotHub通知业务系统。

如果一个状态是业务系统、IotHub和设备端都有可能更改的，那么使用在10.7节将要讲的设备影子可能会更好。

如果业务系统需要记录设备状态的历史记录，那么使用前面实现的上行数据就可以了：把设备状态看作一般的上行数据。

### 8.3.2 DeviceSDK的实现

DeviceSDK的实现比较简单，只需要实现一个方法，将状态数据发布到指定的主题。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. updateStatus(status){
3.     if(this.client != null){
4.         var topic =
'update_status/${this.productName}/${this.deviceName}/
  ${new ObjectId().toHexString()}'
5.         this.client.publish(topic,
JSON.stringify(status), {
6.             qos: 1
7.         })
8.     }
9. }
```

---

### 8.3.3 IotHub Server的实现

IotHub Server的实现和前面实现上行数据处理的逻辑非常相似。

#### 1. 更新设备的ACL列表

首先需要将上报状态的主题加入设备的ACL列表。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = [
4.
5.     'upload_data/${this.product_name}/${this.device_name}/+/'
6.   ],
7.   'update_status/${this.product_name}/${this.device_name}/+'
8. ]
9. }
```

---

你可能需要重新注册一个设备进行后续测试（或者手动在MongoDB里更新已有的设备ACL列表）。

#### 2. 处理设备上报的状态

和处理上行数据一样，IotHub Server根据主题名判断该数据是否是上报的状态数据，然后从主题中提取出元数据。

---

```
1. //IotHub_Server/services/message_service.js
2. static dispatchMessage({topic, payload, ts} = {}) {
3.     var dataTopicRule =
"upload_data/:productName/:deviceName/:dataType/:
messageId";
4.     var statusTopicRule =
"update_status/:productName/:deviceName/:messageId"
5.     const topicRegx = pathToRegexp(dataTopicRule)
6.     const statusRegx = pathToRegexp(statusTopicRule)
7.     var result = null;
8.     if ((result = topicRegx.exec(topic)) != null) {
9.         //处理上报数据
10.        ...
11.    } else if ((result = statusRegx.exec(topic)) !=
null) {
12.        this.checkMessageDuplication(result[3],
function (isDup) {
13.            if (!isDup) {
14.                MessageService.handleUpdateStatus({
15.                    productName: result[1],
16.                    deviceName: result[2],
17.                    deviceStatus: new Buffer(payload,
'base64').toString(),
18.                    ts: ts
19.                })
20.            }
21.        })
22.    }
23. }
```

---

同样的，在处理状态数据之前，要先进行消息去重。

我们会在MessageService的handleUpdateStatus方法里根据上报状态的内容更新设备的状态。

首先在Device模型里面添加存储设备状态的字段。

---

```
1. //IotHub_Server/models/devices
2. const deviceSchema = new Schema({
3.   ...
4.   device_status: {
5.     type: String,
6.     default: "{}"
7.   },
8.   last_status_update: Number //最后一次状态更新的时间
9. })
```

---

然后实现handleUpdateStatus方法。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleUpdateStatus({productName, deviceName,
3.   deviceStatus, ts}) {
4.   Device.findOneAndUpdate({product_name:
5.     productName, device_name:
6.       deviceName,
7.       "$or": [{last_status_update: {"$exists": false}},
8.       {last_status_update:
9.         {"$lt": ts}}]}],
10.    {device_status: deviceStatus,
11.    last_status_update: ts}, {useFindAndModify:
12.      false}).exec()
13. }
```

---

虽然MQTT协议可以保证数据包是按序到达的，但是在WebHook并发处理时有可能会乱序，所以我们只更新时间更近的状态数据。这也是为什么我们在Device模型用一个last\_status\_update字段。

### 3. 通知业务系统

同上报数据一样，设备上报状态时IotHub也是通过RabbitMQ通知业务系统，IotHub会向RabbitMQ名为iothub.events.update\_status的

Direct Exchange发送一条消息，RoutingKey为设备的ProductName，消息格式依然是BSON。

---

```
1. //IotHub_Server/services/notify_service.js
2. var updateStatusExchange =
  "iothub.events.update_status"
3. static notifyUpdateStatus({productName, deviceName,
  deviceStatus}){
4.     var data = bson.serialize({
5.         device_name: deviceName,
6.         device_status: deviceStatus
7.     })
8.     if(currentChannel != null) {
9.         currentChannel.publish(updateStatusExchange,
productName, data, {
10.             persistent: true
11.         })
12.     }
13. }
```

---

然后在handleUpdateStatus调用这个方式来通知业务系统。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleUpdateStatus({productName, deviceName,
  deviceStatus, ts}) {
3.     Device.findOneAndUpdate({product_name:
productName, device_name:
  deviceName,
4.         "$or": [{last_status_update: {"$exists": false}},
{last_status_update:
  {"$lt": ts}}]},
5.     {device_status: deviceStatus,
last_status_update: ts}, {useFindAndModify:
  false}).exec(function (error, device) {
6.         if (device != null) {
7.             NotifyService.notifyUpdateStatus({
8.                 productName: productName,
9.                 deviceName: deviceName,
```



```
10.         deviceStatus: deviceStatus
11.     })
12. }
13. })
14. }
```

---

### 8.3.4 Server API: 查询设备状态

我们只需在设备详情接口里面返回状态字段就可以了。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.toJSONObject = function () {
3.   return {
4.     product_name: this.product_name,
5.     device_name: this.device_name,
6.     secret: this.secret,
7.     device_status: JSON.parse(this.device_status)
8.   }
9. }
```

---

### 8.3.5 代码联调

我们可以在business\_sim.js里加一小段代码，来验证整个流程。

---

```
1. //IotHub_Server/business_sim.js
2. var updateStatusExchange =
  "iothub.events.update_status"
3. channel.assertExchange(updateStatusExchange,
  'direct', {durable: true})
4. var queue = "iotapp_update_status";
5. channel.assertQueue(queue, {durable: true})
6. channel.bindQueue(queue, updateStatusExchange,
  "IotApp")
7. channel.consume(queue, function (msg) {
8.   var data = bson.deserialize(msg.content)
9.   console.log('received from ${data.device_name},
status: ${data.device_
  status}')
10.  channel.ack(msg)
11. })
```

---

然后写一小段代码，调用DeviceSDK的updateStatus方法来上报设备状态。

---

```
1. //IotHub_Device/samples/update_status.js
2. ...
3. device.connect()
4. device.updateStatus({lights: "on"})
```

---

运行IotHub\_Server/business\_sim.js，然后运行update\_status.js，在运行business\_sim.js的终端会输出received

from 60de4bqyu,status:{"lights":"on"}}, 最后调用设备详情接口  
curl<http://localhost:3000/devices/IotApp/60de4bqyu>, 输出如下。

---

```
{"product_name":"IotApp","device_name":"60de4bqyu","secret":"sVDhDJZhm7",  
"device_status":{"lights":"on"},"connections":  
[{"connected":true,"client_id":"IotApp/60de4bqyu/update_status","ipaddress":"127.0.0.1","connected_at":1558964672,  
"disconnect_at":1558964668}]}%
```

---

可以看到，设备状态上报的整个流程都已完成了。

### 8.3.6 为何不用Retained Message

在第二部分中，我曾提到过用Retained Message记录设备的状态是个不错的方案，但是这在Maque IotHub里目前是行不通的，我们看一下假设设备在向某个主题，如TopicA发送一条Retained Message表明自己的状态时，会发生什么：

- 1) 设备A向TopicA发送一条消息M，标记为Retained，QoS=1；
- 2) EMQ X Broker收到M，回复设备A PUBACK；
- 3) EMQ X为TopicA保存下Retained消息M\_retained；
- 4) EMQ X通过WebHook将消息传递给IotHub Server；
- 5) EMQ X发现没有任何Client订阅TopicA，丢弃M。

可以看到，由于在IotHub中使用的是基于Hook的方式来获取设备发布的消息，没有实际的Client订阅设备发布状态的主题，所以即使发送Retained Message，也只是白白浪费Broker的存储空间罢了。

那么设备需要在什么时候上报状态呢？这在DeviceSDK里面并没有强制的约定，不过我的建议是在设备每次开机时以及在状态发生变化时。

本节完成了设备状态上报功能。这样IotHub的上行数据处理功能就完整了，第8.4节将学习一种非常适合于物联网数据存储的数据库：时序数据库。

## 8.4 时序数据库

在前文中我们完成了IoTHub上行数据处理的功能。截至目前，IoTHub都是使用MongoDB作为数据存储，不过在物联网的应用中，在某些情况下，我们可能还会用到别的存储方案，比如说时序数据库，那么本节就来学习一下时序数据库。

### 8.4.1 时序数据

首先来看一下什么是时序数据。时序数据是一类按照时间维度进行索引的数据，它记录了某个被测量实体在一定时间范围内，每个时间点上的一组测试值。传感器上传的蔬菜大棚每小时的湿度和温度数据、A股中某支股票每个时间点的股价、计算机系统的监控数据等，都属于时序数据，时序数据有如下特点：

- 数据量较大，写入操作是持续且平稳的，而且写多读少；
- 只有写入操作，几乎没有更新操作，比如去修改大棚温度和湿度的历史数据，那是没有什么意义的；
- 没有随机删除，即使删除也是按照时间范围进行删除。删除蔬菜大棚08:35的温度记录没有任何实际意义，但是删除6个月以前的记录是有意义的；
- 数据实时性和时效性很强，数据随着时间的推移不断追加，旧数据很快失去意义；
- 大部分以时间和实体为维度进行查询，很少以测试值为维度查询，比如用户会查询某个蔬菜大棚某个时间段的温度数据，但是很少会去查询温度高于多少度的数据记录。



如果说你的业务数据符合上面的条件，比如你的业务数据属于监控、运维类，或者你的数据需要用折线图之类的进行可视化，那么你就可以考虑使用时序数据库。

## 8.4.2 时序数据库

时序数据库就是用来存储时序数据的数据库，时序数据库相较于传统的关系型数据库和非关系型数据库而言，专门优化了对时序数据的存储，开源的时序数据库有InfluxDB、OpenTSDB、TimeScaleDB等。本书使用InfluxDB数据库进行演示。

时序数据库有如下几个概念。

- Metric：度量，可以当作关系型数据库中的表（table）。
- Data Point：数据点，可以当作关系型数据库中的行（row）。
- Timestamp：时间戳，数据点生成时的时间戳。
- Field：测量值，比如温度和湿度。
- Tag：标签，用于标识数据点，通常用来标识数据点的来源，比如温度和湿度数据来自哪个大棚，可以当作关系型数据库表的主键。

图8-3所示为时序数据库概念示例，方便大家理解这几个概念。

- Vents：度量Metric，存储所有大棚的温度和湿度数据；
- Humidity和Temperature：测量值Field；

- Vent No.和Vent Section：Tag标签，标识测量值来自于哪个大棚；
- Time：时间戳Timestamp。

Time	Humidity	Temperature	Vent No.	Vent Section
2015-01-10 00:00:00	18%	31	#107	A
2015-01-10 01:00:00	16%	33	#107	A
2015-01-10 02:00:00	20%	35	#106	B
2015-01-10 03:00:00	22%	29	#110	D

图8-3 时序数据库概念示例

线框中的部分就是一个Data Point，它包含Timestamp、Field、Tag。

### 8.4.3 收集设备连接状态变化的数据

这里我们将IoTHub中的一些状态监控的数据存入InfluxDB来演示如何使用时序数据库。目前我们的数据中，设备连接状态变化（上线/下线）可以算作一种时序数据，我们将这个数据加入时序数据库，这样就可以看到设备连接状态的变化情况，对故障排查也很有帮助。

#### 1. 安装和运行InfluxDB

可以在<https://docs.influxdata.com/influxdb/v1.7/>找到InfluxDB的安装文档，我们使用的是InfluxDB 1.7.6版本，在对应的操作系统上安装并按照默认配置运行InfluxDB，默认情况下InfluxDB运行在“localhost:8086”。

然后，我们创建一个数据库来存储设备的连接状态变化记录。

1) 运行InfluxDB CLI: influx。

---

```
connected to http://localhost:8086 version 1.7.6
InfluxDB shell version: 1.7.6
Enter an InfluxQL query
```

---

2) 然后输入create database iothub，敲击回车键，这样数据库就创建好了。

## 2. 存储数据

接下来需要做的事情比较简单，IoT Hub在获取到设备上线和下线时间时，将对应的数据写入InfluxDB，这里我们使用InfluxDB的Node.js库Influx。

首先创建一个service类来实现设备连接记录的写入。

在InfluxDB中，Metric被称为Measurement。

---

```
1. //IoT_Hub_Server/services/influxdb_service.js
2.
3. const Influx = require('influx')
4. const influx = new Influx.InfluxDB({
5.   host: process.env.INFLUXDB,
6.   database: 'iothub',
7.   schema: [
8.     {
9.       measurement: 'device_connections',
10.      fields: {
11.        connected: Influx.FieldType.BOOLEAN
12.      },
13.      tags: [
14.        'product_name', 'device_name'
15.      ]
16.    }
17.  ]
18. })
19.
20. class InfluxDBService{
21.   static writeConnectionData({productName,
22.     deviceName, connected, ts}) {
23.     var timestamp = ts == null ?
24.       Math.floor(Date.now() / 1000) : ts
25.     influx.writePoints([
26.       {
27.         measurement: 'device_connections',
```

```
26.         tags: {product_name: productName,
device_name: deviceName},
27.         fields: {connected: connected},
28.         timestamp: timestamp
29.     }
30. ], {
31.     precision: 's',
32. }).catch(err => {
33.     console.error('Error saving data to InfluxDB!
${err.stack}')
34. })
35. }
36. }
37. module.exports = InfluxDBService
```

---

这里使用Measurement: 'device\_connections' 存储设备连接状态变化的数据, Tag为一个二元组 (ProductName, DeviceName), 可以唯一标识一台设备, Field是一个布尔值, 标识设备的连接状态。

然后在设备上线/下线时, 调用writeConnectionData方法, 将设备连接状态变化的数据写入InfluxDB。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.statics.addConnection = function (event)
{
3.     var username_arr = event.username.split("/")
4.     let productName = username_arr[0];
5.     let deviceName = username_arr[1];
6.     this.findOne({product_name: productName,
device_name: deviceName},
function (err, device) {
7.         if (err == null && device != null) {
8.             ...
9.             influxDBService.writeConnectionData({
10.                 productName: productName,
11.                 deviceName: deviceName,
12.                 connected: true,
```

```
13.         ts: event.connected_at
14.     })
15. }
16. })
17.
18. }
19.
20. deviceSchema.statics.removeConnection = function
(event) {
21.     var username_arr = event.username.split("/")
22.     let productName = username_arr[0];
23.     let deviceName = username_arr[1];
24.     this.findOne({product_name: productName,
device_name: deviceName},
    function (err, device) {
25.         if (err == null && device != null) {
26.             ...
27.             influxDBService.writeConnectionData({
28.                 productName: productName,
29.                 deviceName: deviceName,
30.                 connected: false
31.             })
32.         }
33.     })
34. }
```

---

接下来我们可以验证一下，运行

IotHub\_Device/samples/connect\_to\_server.js，等看到device is online的输出以后，按Ctrl+C组合键终止程序，重复操作几次之后运行influx-precision-s，然后查询device\_connections。

---

```
use iothub
select * from device_connections
```

---

这时，我们会看到，设备的连接状态变化已经被存入了InfluxDB。

---

```
Connected to http://localhost:8086 version 1.7.6
InfluxDB shell version: 1.7.6
Enter an InfluxQL query
> use iothub
Using database iothub
> select * from device_connections
name: device_connections
time          connected device_name product_name
----          -
1559046440 true      60de4bqyu  IotApp
1559046442 false     60de4bqyu  IotApp
1559046443 true      60de4bqyu  IotApp
1559046444 false     60de4bqyu  IotApp
>
```

---

influx-precision-s表明InfluxDB使用UNIX时间戳格式来显示time字段。

通常我们可以将这些监控数据可视化，但这已经超出了本书的范围，有兴趣的读者可以自行实现。



## 8.5 本章小结

本章设计和实现了IoTHub上行数据的全部处理功能，并展示了如何在IoTHub中使用时序数据库。接下来让我们开始实现另外一个非常重要的功能——下行数据处理。

## 第9章 下行数据处理

什么是下行数据？

在物联网应用中，下行数据一般有两种：第一种是需要同步的数据，比如平台需要把训练好的模型部署到前端的摄像头上，这样平台下发给设备的消息中就包含了模型数据的信息；第二种是指令，平台下发给设备，要求设备完成某种操作，比如共享单车的服务端下发给单车开锁的指令。

在IoTHub里，我们会把这两种下行数据统称为指令，因为第一种数据也可以当作是要求设备完成“同步数据”这个操作的指令。大多数情况下，设备在收到指令后都应该向业务系统回复指令执行的结果<sup>[1]</sup>，比如文件有没有下载完毕、继电器有没有打开等。是否回复以及如何回复应该由业务逻辑决定，这个是业务系统和设备之间的约定，IoTHub只负责将业务系统下发的指令发送到设备，同时将设备对指令的回复再传送回业务系统。IoTHub在实现一些内部的功能时，也会向设备发送一些内部的指令。

IotHub的指令下发有如下功能。

1) 业务系统可以通过IoTHub Server API提供的接口向指定的设备发送指令，指令可以包含任意格式的数据，比如字符串和二进制数

据。

2) 指令可设置过期时间，过期的指令将不会被执行。

3) 业务系统可在设备离线时下发指令，设备在上线以后可以接收到离线时由业务系统下发的指令。

4) 设备可以向业务系统回复指令的执行结果，IoT Hub会把设备的回复通知给业务系统，通知包括：哪个设备回复了哪条指令、回复的内容是什么等。

那么IoT Hub应该如何将指令下发给设备呢？

[1] 注意，不是回复指令已收到，因为使用QoS>1 的消息在MQTT 协议层面就已经保证设备一定能收到指令。

## 9.1 选择一个可扩展的方案

在IoTHub里，下行数据分为两种：

- 业务系统下发给设备的消息，比如需要同步的数据、需要执行的指令等，这些数据会经由IoTHub发送给设备；
- IoTHub和设备之间发送的消息，通常是为了实现IoTHub相关功能的内部消息。

就像在第8章上行数据处理开头说的那样，在MQTT协议架构里面没有“服务端”和“客户端”的概念，所谓的上行和下行只是我们在实现IoTHub时抽象出来的概念。从IoTHub Server发送到设备的数据是下行数据，反之就是上行数据；而对MQTT Broker来说，IoTHub Server和设备一样，都是MQTT Client。接下来我们看一下发送下行数据可能的一些方案。

### 9.1.1 完全基于MQTT协议的方案

这种方案的逻辑非常直接，IoT Hub Server以MQTT Client的身份接入EMQ X Broker，将数据发布到设备订阅的主题上。

- 设备端订阅某特定的主题，比如/cmd/ProductNameA/DeviceNameA。
- IoT Hub Server启动一个MQTT Client接入EMQ X Broker。
- 业务系统通过IoT Hub Server API的接口告知IoT Hub Server要发送的数据和设备。
- IoT Hub的MQTT Client将数据发布到/cmd/ProductNameA/DeviceNameA。

这是一个可行的方案，只不过仍然存在单点故障的问题，如果IoT Hub Server用于发布的MQTT Client挂了怎么办？

我们可以更进一步，IoT Hub Server可以同时启用多个用于发布的MQTT Client，这些Client可以从一个工作队列（比如RabbitMQ、Redis等）里获取要发布的消息，然后将其发布到对应的设备，在每次

IoTHub Server需要发送数据到设备时，只需要往这个队列里投递一条消息就可以了。具体流程如图9-1所示。

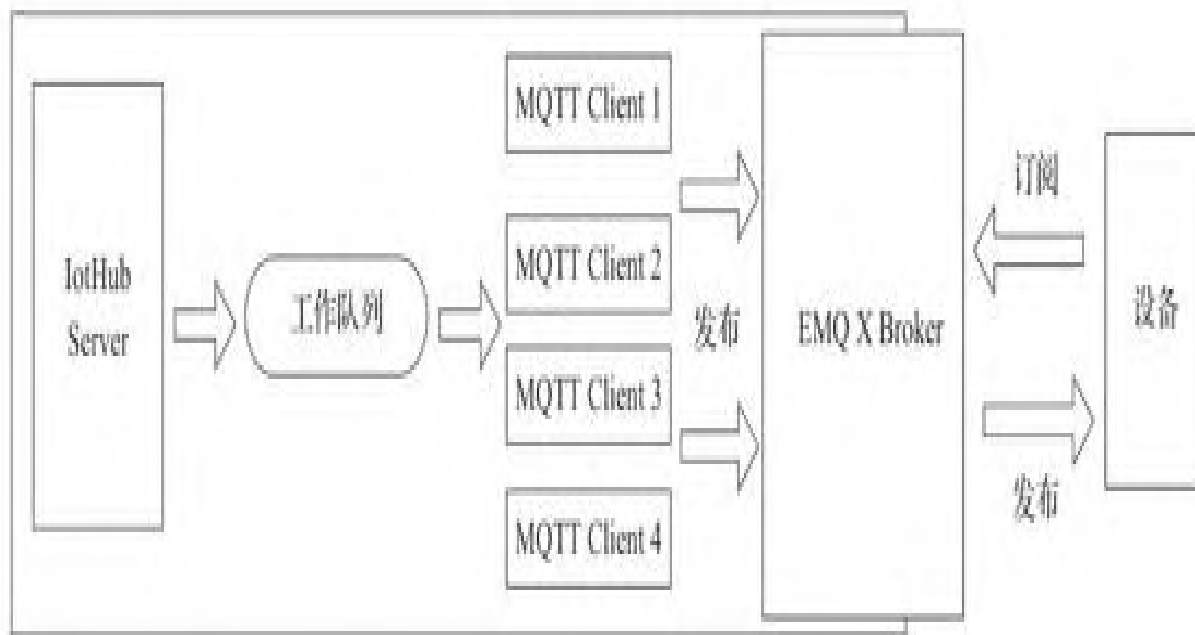


图9-1 完全基于MQTT协议的下行数据处理架构

现在这个方案就具有较好的扩展性了，也不存在单点故障。实际上我在几个项目中使用过这种方案，效果还是不错的。

这种方案的唯一缺点就是引入了多个MQTT Client这样额外的实体，提高了系统的复杂性，同时增加了开发、部署和运维监控的成本。

## 9.1.2 基于EMQ X RESTful API的方案

在设备生命周期管理的功能里，我们已经使用过了EMQ X的RESTful API了，EMQ X的RESTful API提供了一个接口，可以向某个主题发布消息。

**API定义：**POST api/v3/mqtt/publish。

API请求参数：

---

```
{
  "topic": "test_topic",
  "payload": "hello",
  "qos": 1,
  "retain": false,
  "client_id": "mqttjs_ab9069449e"
}
```

---

这种方案不需要维护多个用于发布的MQTT Client，在开发和部署上的复杂度要低一些，IoT Hub会使用该方案向设备发送下行数据，如图9-2所示。

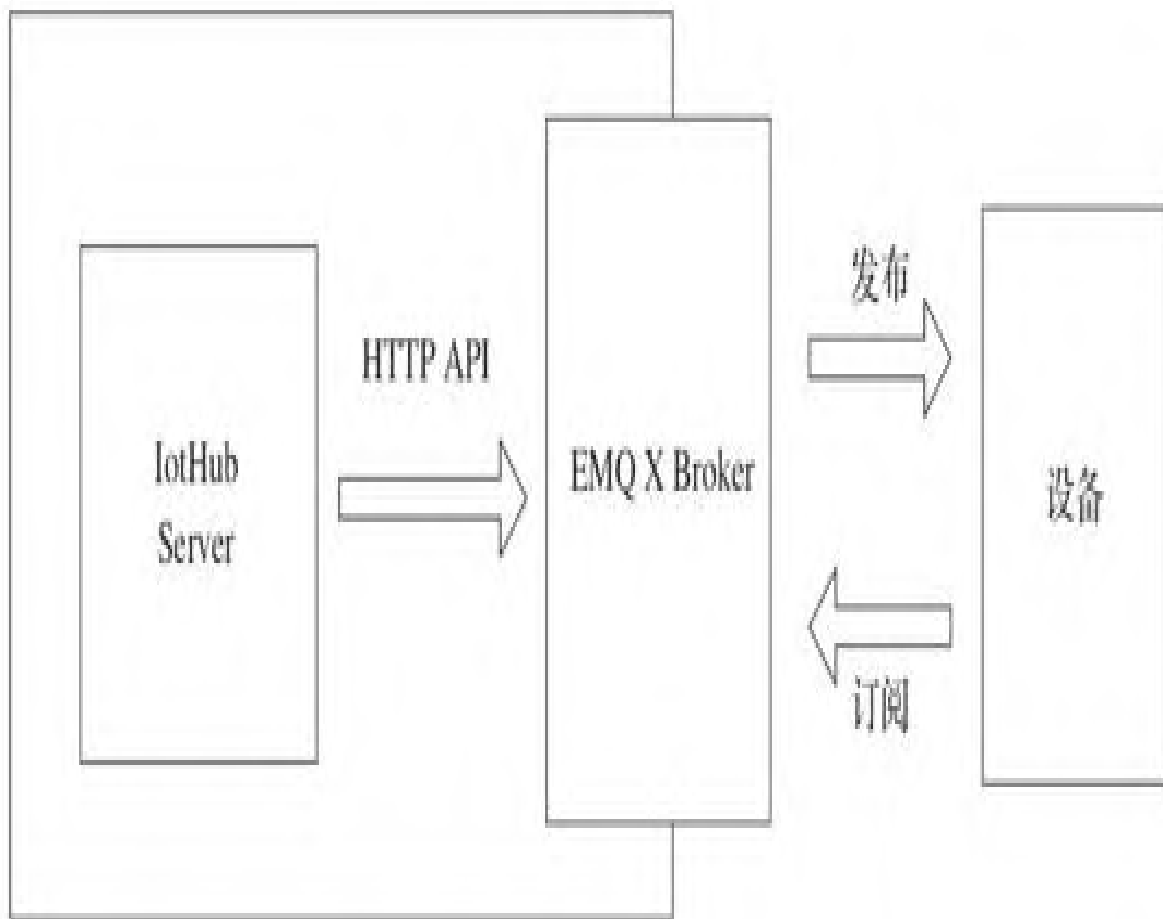


图9-2 基于RESTful API的方案



### 9.1.3 下行数据格式

和上行数据一样，指令也由元数据和负载组成，指令的元数据也是放在主题名中的，一般来说指令有如下元数据。

- ProductName、DeviceName：用于标识指令发送给哪个设备。
- MessageID：作为消息的唯一标识，用于消息去重，也用于标识指令，设备回复指令时会用到。
- 指令名称：用于标识指令的名称，比如单车开锁的指令可以叫作unlock。
- 指令类别：用于标识指令的类别，在后面的章节实现一些特殊的指令时会用到。
- 过期时间：有些指令具有时效性，设备不应该执行超过有效时间的指令，比如单车开锁的指令，假设单车的网络断开，一个小时后恢复了，那么单车会收到在断线期间用户发出的开锁指令，单车不应该执行这些指令，所以需要给开锁指令加一个过期时间，比如1分钟。

负载中包含了执行指令需要的额外数据，比如前面说的部署模型到前端摄像头的情景，指令的负载就应该是模型的数据，比如下载模

型的URL，指令名称可以称为update\_model。

我们可以这样理解IoTHub里面的指令，下发指令相当于远程在设备上执行一个函数 $f(x)$ ，那么在主题名里的指令名称就相当于函数名 $f$ ，指令的负载就相当于函数的参数 $x$ ，设备还会将 $f(x)$ 的返回值回复给业务系统或者IoTHub（如果 $f(x)$ 有返回值的话）。

## 9.1.4 主题名规划

和上行数据处理一样，IoT Hub会把指令的元数据放在主题名中，为了接收下发的指令，设备需要订阅以下主题：

`cmd/:ProductName/:DeviceName/:CommandName/:Encoding/:RequestID/:ExpiresAt`。

这个主题的第一层级代表的是指令的类别，目前固定为`cmd`，代表普通的下行指令，后面我们还会看到其他类型的指令。后面的各个层级代表一种指令的元数据，元数据的含义如下所示。

- `ProductName`和`DeviceName`：这两个元数据很好理解，代表接受指令的设备名称，设备用自己的`ProductName`和`DeviceName`进行订阅。

- `CommandName`：指令的名称，比如重启设备的名称叫作`reboot`。

- `Encoding`：指令数据的编码格式，由于IoT Hub提供给业务系统的接口是HTTP的，同时IoT Hub Server也是调用EMQ X的REST API发布指令，所以如果发布的指令携带的是二进制数据，就需要对这个二进制数据进行编码，让它变成一个字符串。当Device SDK接收到二进制

的指令数据时，需要按照相应的编码方式解析出原始的二进制数据。

Encoding可以有2种值：plain或Base64，其中plain代表未编码，指令数据为字符串的时候使用；Base64代表使用Base64编码，指令数据为二进制数据时使用。

- RequestID，指令的编号，有两层意义：第一，和上行数据的MessageID一样，用来做消息去重；第二，用于唯一标识一条指令，设备在回复指令时需要带上指令的RequestID。

- ExpiresAt：可选，指令的过期时间，格式为UNIX时间戳。如果指定了ExpiresAt，那么DeviceSDK在收到指令时就会检查当前时间是否大于ExpiresAt，如果是，就直接丢弃掉这条指令。

如果我们统一对Payload进行Base64编码的话就可以省去Encoding这个层级，这里增加这个层级的考虑是尽量减少设备端不必要的计算，如果发送的指令数据是ASCII字符串就不用再decode了。你可以根据具体情况来决定是否需要这个层级。

## 9.1.5 如何订阅主题

一般来说，按照MQTT协议的方式，DeviceSDK可以按如下方式订阅上面的主题。

---

```
1. client.on('connect', function (connack) {
2.   if(connack.returnCode == 0) {
3.     if (connack.sessionPresent == false) {
4.
5.       client.subscribe('cmd/${this.productName}/${this.DeviceName}/+//+/+/#', {
6.         qos: 1
7.       }, function (err, granted) {
8.         if (err != undefined) {
9.           console.log("subscribe failed")
10.        } else {
11.          console.log('subscribe succeeded with
12.            ${granted[0].topic}, qos:
13.            ${granted[0].qos}')
14.        }
15.      })
16.    }
17.  } else {
18.    console.log('Connection failed:
19.      ${connack.returnCode}')
20.  }
21.})
```

---

cmd/\${this.productName}/\${this.DeviceName}/+//+/+/#正好可以匹配接受指令用的主题，因为ExpiresAt这个层级是可选的，所以用放在最后的#号来匹配，我们以后在设计主题的时候，尽量把可选的层级放在最后面。

除了这种方式以外，我们还可以利用EMQ X的服务端订阅功能进行更高效、更灵活的订阅。服务端订阅指的是，当MQTT Client连接到EMQ X Broker时，EMQ X会按照预先定义好的规则自动为Client订阅主题。用这种方式设备不需要再发送subscribe，增加和减少设备订阅的主题也不需要改动设备的代码。

在后文中，我们会看到如何使用EMQ X的服务端订阅功能。

## 9.1.6 设备端消息去重

回想一下在处理上行数据时IotHub是怎样实现消息去重的？

IotHub会把已收到消息的MessageID存入Redis，每次收到新消息时都会拿新消息的MessageID去和已收到的MessageID进行比较，如果找到相同的，就丢弃收到的消息。同时对MessageID的存储进行时间限制，这样才不会让存储空间无限增大。

同样，在DeviceSDK端，我们也需要找到这样一个存储RequestID的缓存来帮助我们进行消息去重，这个缓存最好有以下的特性：

- key-value存储；
- 可以设置key的有效期；
- 可持久化，设备掉电也不会丢失已存储的RequestId。

本书使用Node.js实现DeviceSDK，这里会使用一个Node.js版本满足上述要求的缓存实现。

如果你是在其他语言或平台上进行设备端的开发，那么你还需要找到或者自己实现一个类似的缓存。

### 9.1.7 指令回复

当设备回复指令时，它需要向一个特定的主题发布一个消息。同样，我们把这个回复的元数据放在主题名中，把回复的数据放在消息的负载中：cmd\_resp/:ProductName/:DeviceName/:CommandName/:RequestID/:MessageID，其中，

- ProductName, DeviceName：标识回复来自于哪个设备；
- CommandName：所回复的指令的名称；
- RequestID：所回复的指令的RequestID；
- MessageID：因为回复本身也是一条上行数据，也需要MessageID来唯一标识自己，以实现消息去重。

本节设计了IoTHub的指令下发功能，并选择了发送下行数据的方案，接下来开始实现指令下发的服务端功能。



## 9.2 DeviceSDK端的实现

本节开始实现IotHub指令下发的DeviceSDK端的功能。首先进行消息去重，接着使用正则表达式提取出元数据，然后通过事件的方式将指令的数据传递给设备应用代码，最后提供一个接口供设备对指令进行回复。

## 9.2.1 消息去重

node-persist是一个Node.js提供的本地存储功能库，我们可用来存储已收到指令的RequestID。

首先，在构造函数中初始化存储。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. const storage = require('node-persist');
3. class IotDevice extends EventEmitter {
4.   constructor({serverAddress = "127.0.0.1:8883",
productName, deviceName,
secret, clientID, storePath} = {}) {
5.     ...
6.     storage.init({dir: '${storePath}/message_cache'})
7.     ...
8.   }
```

---

代码的第6行初始化了用于存储RequestID的缓存，基于文件message\_cache。

然后在IotDevice类里面实现一个方法，来检查RequestID是否重复。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. class IotDevice extends EventEmitter {
3.   ...
4.   checkRequestDuplication(requestID, callback) {
5.     var key = 'requests/${requestID}'
```

---

```
6.     storage.getItem(key, function (err, value) {
7.         if (value == null) {
8.             storage.setItem(key, 1, {ttl: 1000 * 3600 *
6}))
9.             callback(false)
10.        } else {
11.            callback(false)
12.        }
13.    })
14. }
15.
16. ...
17. }
```

---

## 9.2.2 提取元数据

和IotHubServer一样，DeviceSDK也使用pathToRegex生成正则表达式，从主题名中提取出元数据。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. class IotDevice extends EventEmitter {
3.     connect() {
4.         ...
5.         this.client.on("message", function (topic,
message) {
6.             self.dispatchMessage(topic, message)
7.         })
8.     }
9.
10.    dispatchMessage(topic, payload){
11.        var cmdTopicRule =
"cmd/:productName/:deviceName/:commandName/:encoding/
:requestID/:expiresAt?"
12.        var result
13.        if((result =
pathToRegex(cmdTopicRule).exec(topic)) != null){
14.            this.checkRequestDuplication(result[6],
function (isDup) {
15.                if (!isDup) {
16.                    self.handleCommand({
17.                        commandName: result[3],
18.                        encoding: result[4],
19.                        requestID: result[5],
20.                        expiresAt: result[6] != null ?
parseInt(result[6]) : null,
21.                        payload: payload
22.                    })
23.                }
24.
25.            })
26.        }
```

```
27.    }  
28.  
29.    ...  
30. }
```

---

代码的第14行使用从主题名中提取出的RequestID进行消息去重，如果消息不重复，则对消息进行处理（代码的第16行）。

因为expiredAt层级是可选的，所以用“:expiresAt?”来表示。

### 9.2.3 处理指令

DeviceSDK对指令的处理流程如下：

- 1) 检查指令是否过期；
- 2) 根据Encoding对指令数据进行解码；
- 3) 通过Emit Event的方式将指令传递给设备应用代码。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. class IotDevice extends EventEmitter {
3.     ...
4.     handleCommand({commandName, requestID, encoding,
payload, expiresAt}){
5.         if(expiresAt == null || expiresAt >
Math.floor(Date.now() / 1000)){
6.             var data = payload;
7.             if(encoding == "base64"){
8.                 data = Buffer.from(payload.toString(),
"base64")
9.             }
10.            this.emit("command",  commandName, data)
11.        }
12.    }
13.
14.    ...
15. }
```

---

代码的第5行对指令的有效期进行了判断，代码的第7~9行通过Encoding的值对指令数据进行了解码。

设备应用代码可以通过如下方式来获取指令的内容。

---

```
1. device.on("command", function(commandName, data){
2.     //处理指令
3. })
```

---

## 9.2.4 回复指令

是否回复指令，以及什么时候回复指令是由设备的应用代码来决定的，DeviceSDK做不到强制约定，但是可以提供帮助函数来屏蔽掉回复指令所需的细节。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. class IotDevice extends EventEmitter {
3.     ...
4.     handleCommand({commandName, requestID, encoding,
payload, expiresAt}) {
5.         if (expiresAt == null || expiresAt <
Math.floor(Date.now() / 1000)) {
6.             var data = payload;
7.             if (encoding == "base64") {
8.                 data = Buffer.from(payload, "base64")
9.             }
10.            var respondCommand = function (respData) {
11.                var topic =
'cmd_resp/${this.productName}/${this.deviceName}/
${commandName}/${requestID}/${new
ObjectId().toHexString()}'
12.                this.client.publish(topic, respData, {
13.                    qos: 1
14.                })
15.            }
16.            this.emit("command", commandName, data,
respondCommand)
17.        }
18.    }
19.
20.    ...
21. }
```

---



代码第10行创建了一个闭包，包含了回复这个指令的具体代码，并在command事件里把这个闭包传递给了设备应用代码（第16行），设备应用代码可以通过如下方式回复指令。

---

```
1. device.on("command", function(commandName, data,
2.     respondCommand) {
3.     //处理指令
4.     ...
5.     respondCommand("ok") //处理完毕后回复，可以带任何格式的数据，字符串或者二进制数据
6. })
```

---

这样IoT Hub内部的指令下发、回复的流程和细节对设备应用代码是完全透明的，符合我们对Device SDK的期望。

在非Node.js的语言环境，没有闭包的情况下，你也可以使用类似的编程技巧来完成，比如方法对象（Method Object）、匿名函数、内部类、Lambda、block等。

在Device SDK端，我们用node-persist存储RequestID，通过正则表达式对主题名进行模式匹配的方式提取出元数据，对指令的过期时间进行检查后，通过事件的方式将指令传递给设备应用代码，并使用闭包的方式提供接口供设备应用代码回复指令。

本节基本完成了指令下发Device SDK端的功能代码，接下来我们开始实现IoT Hub Server端的功能。

## 9.3 服务端的实现

本节开始实现指令下发的IotHub Server端。首先使用EMQ X的API发布消息，并提供指令下发接口供业务系统调用，然后使用EMQ X的服务器订阅功能，实现设备的自动订阅；当设备对指令进行回复以后，通过RabbitMQ将设备的回复通知到业务系统，最后将IotHub Server端的代码和DeviceSDK的代码进行联调。

### 9.3.1 更新ACL列表

由于设备端需要在回复指令的时候发布到主题

cmd\_resp/:ProductName/:DeviceName/:CommandName/:RequestID/:MessageID。所以我们需要先把这个主题添加到设备的ACL列表里。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = [
4.
5.     'upload_data/${this.product_name}/${this.device_name}/+/+',
6.     'update_status/${this.product_name}/${this.device_name}/+',
7.     'cmd_resp/${this.product_name}/${this.device_name}/+/+/+',
8.   ]
9.   ...
10. }
```

---

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB的ACL列表。

### 9.3.2 EMQ X发布功能

这里我们调用EMQ X Publish API接口，实现消息发布的功能，当IotHub Server需要向某个设备下发一个指令的时候会用到。

和前面用到的EMQ X RESTful API一样，我们将调用EMQ X Publish API的代码封装到service类里面。

---

```
1. //IotHub_Server/service/emqx_service
2. var shortid = require("shortid")
3. static publishTo({topic, payload, qos = 1, retained =
false})) {
4.     const apiUrl = '${process.env.EMQ
X_API_URL}/mqtt/publish'
5.     request.post(apiUrl, {
6.         "auth": {
7.             'user': process.env.EMQ X_APP_ID,
8.             'pass': process.env.EMQ X_APP_SECRET,
9.             'sendImmediately': true
10.        },
11.        json:{
12.            topic: topic,
13.            payload: payload,
14.            qos: qos,
15.            retained: retained,
16.            client_id: shortid.generate()
17.        }
18.    }, function (error, response, body) {
19.        console.log('statusCode:', response &&
response.statusCode);
20.        console.log('body:', body);
21.    })
22. }
```

---

IotHub支持离线指令下发，所以发布的QoS=1在调用EMQ X的 Publish API时还需要提供一个ClientID，这里随机生成一个就可以。

### 9.3.3 Server API：发送指令

IotHub Server API需要提供一个接口供业务系统向设备下发指令，业务系统可以通过调用这个接口向指定的设备下发一条指令，业务系统可以指定指令的名称和指令附带的数据。

首先在Device类中做一个封装，实现一个下发指令的方法。

---

```
1. //IotHub_Server/models/device.js
2. const ObjectId = require('bson').ObjectId;
3. deviceSchema.methods.sendCommand = function
({commandName, data, encoding,
  ttl = undefined}) {
4.   var requestId = new ObjectId().toHexString()
5.   var topic =
'cmd/${this.product_name}/${this.device_name}/${commandName}/
  ${encoding}/${requestId}'
6.   if (ttl != null) {
7.     topic = '${topic}/${Math.floor(Date.now() / 1000)
+ ttl}'
8.   }
9.   emqxService.publishTo({topic: topic, payload:
data})
10.  return requestId
11. }
```

---

注意这里如果指定了TTL（有效期，单位为秒）的话，是用当前的时间戳加上TTL作为指令的过期时间。

这里仍然使用BSON的ObjectID作为指令的RequestID。同时需要将这个RequestID返回给调用者。

业务系统在请求IoTHub给指定设备下发指令时，需要提供设备的ProductName、DeviceName、指令的名称、指令数据以及指令的TTL。如果指令数据为二进制数据，那么业务系统需要在请求前将指令数据使用Base64进行编码，并在请求时指明编码格式（Encoding）为Base64。

---

```
1. router.post("/:productName/:deviceName/command",
function (req, res) {
2.   var productName = req.params.productName
3.   var deviceName = req.params.deviceName
4.   Device.findOne({"product_name": productName,
"device_name": deviceName},
function (err, device) {
5.     if (err) {
6.       res.send(err)
7.     } else if (device != null) {
8.       var requestId = device.sendCommand({
9.         commandName: req.body.command,
10.        data: req.body.data,
11.        encoding: req.body.encoding || "plain",
12.        ttl: req.body.ttl != null ?
parseInt(req.body.ttl) : null
13.      })
14.      res.status(200).json({request_id: requestId})
15.    }else{
16.      res.status(404).send("device not found")
17.    }
18.  })
19. })
```

---

指令发送成功以后，IoT Hub会把指令的RequestID返回给业务系统，业务系统应该保存这个RequestID，以便在收到设备对指令的回复时进行匹配。

例如，用户可以远程让家里的路由器下载一个文件，并希望下载完成后在手机上能收到通知，那么业务系统在调用IoT Hub Server API下发指令到路由器后应该保存RequestID和用户ID，路由器下载后便回复该指令，业务系统收到后用回复里的RequestID去匹配它保存的RequestID和用户ID，如果匹配成功，则使用对应的用户ID通知用户。

因为我们需要用command参数拼接主题名，所以command参数不应该包含有“#” “/” “+” 以及IoT Hub预留的一些字符，这里为了演示，跳过了输入参数的校验，但是在实际项目中是需要加上的。



### 9.3.4 服务器订阅

在DeviceSDK里面没有任何订阅接收指令的主题代码，因为我们会使用EMQ X的服务器订阅来实现设备的自动订阅。

EMQ X的服务器订阅是在“<EMQ X安装目录>/emqx/etc/emqx.conf”里进行配置的，下面我把这些配置项讲解一下。

首先，打开EMQ X的服务器订阅功能。

---

```
module.subscription = on
```

---

然后配置需要自动订阅的主题名，以及QoS。

---

```
module.subscription.1.topic = topics
module.subscription.1.qos = 1
```

---

module.subscription.1.topic这个配置项支持两个占位符，%u代表Client接入时使用的username，%c代表Client接入时使用的Client Identifier。在IoTHub中，设备接入EMQ X Broker时使用的用户名为ProductName/DeviceName，那么这里我们就可以这样配置自动订阅的主题名。

---

```
module.subscription.1.topic = cmd/%u/+/+/+/#
```

---

目前，EMQ X只支持这种方式定义服务器订阅列表，如果你需要更灵活的配置方式，可用插件的方式扩展或者让设备进行自行订阅。这就是为什么使用ProductName/DeviceName作为设备接入Broker的username的原因了，这算一个小小的取巧。

如果你需要配置更多的订阅主题，可以这样做。

---

```
module.subscription.1.topic = xxx
module.subscription.1.qos = xx
module.subscription.2.topic = xxx
module.subscription.2.qos = xx
module.subscription.3.topic = xxx
module.subscription.3.qos = xx

...
```

---

配置完成以后需要重新启动EMQ X：<EMQ X安装目录  
>/emqx/bin/emqx restart。

我们可以使用EMQ X的Management Web Console验证服务器自动订阅是否生效。

EMQ X默认情况下会在<http://host:18083>启动一个Web服务器，可以通过Web UI来查看和管理EMQ X的状态，登录这个Web Console的默认用户名和密码是admin/public。

运行IotHub\_Device/samples/connect\_to\_server.js，然后访问<http://<host>:18083/#/subscriptions>，你会发现服务器订阅已经生效了，如图9-3所示。



图9-3 EMQ X管理后台查看订阅主题

因为这个主题是服务器自动订阅的，并不是由一个真实的MQTT Client去发起订阅，所以不会触发ACL校验，我们无须将这个主题存入设备的ACL列表中。

### 9.3.5 通知业务系统

指令处理的最后一步就是将设备对指令的回复再转发到业务服务器，和处理上行数据一样，IoT Hub使用RabbitMQ对业务系统进行通知，当IoT Hub收到设备对指令的回复时，会向名为 `iothub.events.cmd_resp`的Exchange发布一条消息，该消息包含 DeviceName、指令名、指令的RequestID及回复数据等内容。Exchange的类型为Direct，RoutingKey为设备的ProductName。

具体流程如下：

- 1) IoT Hub Server通过WebHook获取设备对指令的回复消息；
- 2) IoT Hub Server通过解析消息的主题名获取指令回复的元数据；
- 3) IoT Hub Server向对应的RabbitMQ Exchange发布指令的回复。
- 4) 业务系统从RabbitMQ获取指令回复。

首先在WebHook里添加对指令回复消息的处理，使用预先生成的正则表达式对主题名进行匹配。

---

```

1. /IotHub_Server/messages/message_service.js
2. static dispatchMessage({topic, payload, ts} = {}) {
3.     ...
4.     var cmdRespRule = "
(cmd_resp|rpc_resp)/:productName/:deviceName/:commandName/:requestId/:messageId"
5.     const cmdRespRegx = pathToRegexp(cmdRespRule)
6.     var result = null;
7.     if ((result = topicRegx.exec(topic)) != null) {
8.         ...
9.     } else if ((result = statusRegx.exec(topic)) !=
null) {
10.        ...
11.    } else if ((result = cmdRespRegx.exec(topic)) !=
null) {
12.        this.checkMessageDuplication(result[5],
function (isDup) {
13.            if (!isDup) {
14.                MessageService.handleCommandResp({
15.                    productName: result[1],
16.                    deviceName: result[2],
17.                    ts: ts,
18.                    command: result[3],
19.                    requestId: result[4],
20.                    payload: new Buffer(payload, 'base64')
21.                })
22.            }
23.        })
24.    }
25. }

```

---

然后在handleCommandResp方法里向RabbitMQ对应的Exchange发布一条包含指令回复内容的消息，这里仍然用BSON进行序列化。

---

```

1. //IotHub_Server/service/message_service.js
2. static handleCommandResp({productName, deviceName,
command, requestId, ts,
payload}) {
3.     NotifyService.notifyCommandResp({
4.         productName: productName,

```

```
5.         deviceName: deviceName,
6.         command: command,
7.         requestId: requestId,
8.         ts: ts,
9.         payload: payload
10.     })
11. }
12. //IotHub_Server/service/notify_service.js
13. static notifyCommandResp({productName, deviceName,
command, requestId, ts,
payload}){
14.     var data = bson.serialize({
15.         device_name: deviceName,
16.         command: command,
17.         request_id: requestId,
18.         send_at: ts,
19.         payload: payload
20.     })
21.     if(currentChannel != null){
22.         currentChannel.publish(commandRespExchange,
productName, data)
23.     }
24. }
```

---

9.3.6 代码联调

我们可以总结一下整个指令下发的流程，如图9-4所示。

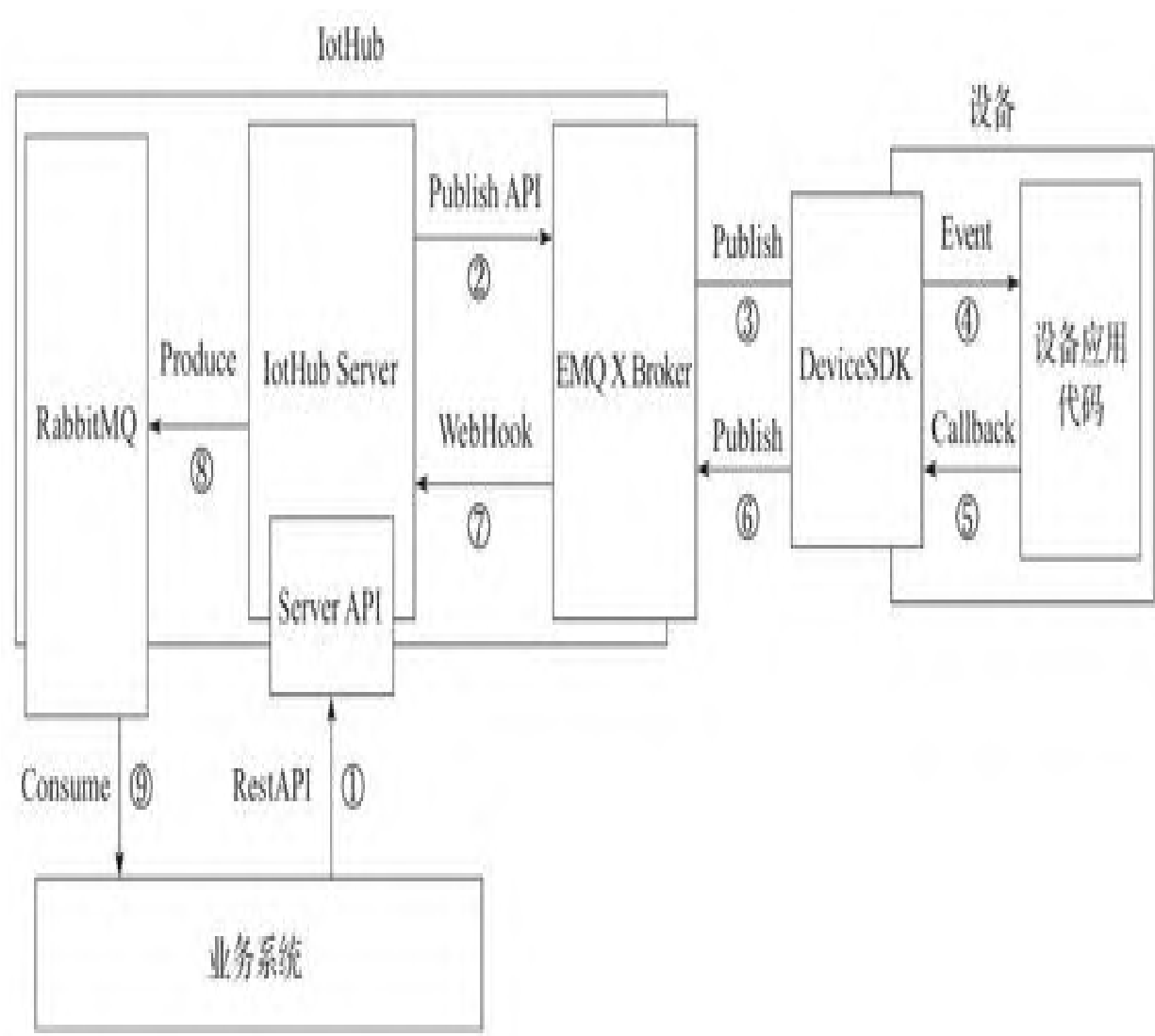


图9-4 下行数据处理流程

1) 业务系统调用Server API发送指令。

- 2) IotHub Server调用EMQ X的Publish API (RESTful)。
- 3) EMQ X Broker发布消息到设备订阅的主题。
- 4) DeviceSDK提取出指令的信息并通过Event的方式传递到设备应用代码。
- 5) 设备应用代码执行完指令要求的操作后，通过Callback (闭包) 的方式要求DeviceSDK对指令进行回复。
- 6) DeviceSDK发布包含指令回复的消息到EMQ X Broker。
- 7) EMQ X Broker通过WebHook将指令回复传递到IotHub Server。
- 8) IotHub Server将指令回复放入到RabbitMQ对应的队列中。
- 9) 业务系统从RabbitMQ的对应队列获得指令的回复。

接下来我们用代码验证这个流程。

现在开始我们把Server端的示例代码放在IotHub\_Server/samples下面。

我们先实现一段模拟业务系统的代码，它有如下功能：

- 调用IotHub Server API，向设备发送指令ping，指令数据为当前的时间戳，以二进制格式传输；



- 可以通过命令行参数指定指令的TTL，默认情况下指令无有效期限制；

- 从RabbitMQ中获取设备对指令的回复，并打印出来。

---

```
1. //IotHub_Server/samples/ping.js
2. require('dotenv').config({path: "../.env"})
3. const bson = require('bson')
4. const request = require("request")
5. var amqp = require('amqplib/callback_api');
6. var exchange = "iothub.events.cmd_resp"
7. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
8.   if (error0) {
9.     console.log(error0);
10.  } else {
11.    connection.createChannel(function (error1,
channel) {
12.      if (error1) {
13.        console.log(error1)
14.      } else {
15.        channel.assertExchange(exchange, 'direct',
{durable: true})
16.        var queue = "iotapp_cmd_resp";
17.        channel.assertQueue(queue, {
18.          durable: true
19.        })
20.        channel.bindQueue(queue, exchange,
process.env.TARGET_PRODUCT_NAME)
21.        channel.consume(queue, function (msg) {
22.          var data = bson.deserialize(msg.content)
23.          if(data.command == "ping") {
24.            console.log('received from
${data.device_name}, requestId:
${data.request_id},payload:
${data.payload.buffer.readUInt32BE(0)}')
25.          }
26.          channel.ack(msg)
27.        })
28.      }
29.    });
```

```

30.    }
31. });
32. const buf = Buffer.alloc(4);
33. buf.writeUInt32BE(Math.floor(Date.now())/1000, 0);
34. var formData = {
35.     command: "ping",
36.     data: buf.toString("base64"),
37.     encoding: "base64"
38. }
39. if(process.argv[2] != null){
40.     formData.ttl = process.argv[2]
41. }
42.
request.post('http://127.0.0.1:3000/devices/${process.env.
TARGET_PRODUCT_
NAME}/${process.env.TARGET_DEVICE_NAME}/command', {
43.     form: formData
44. }, function (error, response, body) {
45.     if (error) {
46.         console.log(error)
47.     } else {
48.         console.log('statusCode:', response &&
response.statusCode);
49.         console.log('body:', body);
50.     }
51. })

```

---

代码的第11~20行绑定queue到对应的Exchange，以接收来自设备的指令回复，第22行将接收到的BSON格式的消息反序列化，再从payload字段中读取指令的回复并打印出来。这里指令的回复是一个32位的整数。

代码的第40行使用命令行参数设定指令的有效期。

代码的第42行调用ServerAPI的发送指令接口向设备下发指令。指令数据为一个32位整数，值为当前时间的UNIX时间戳接下来实现一段

设备端应用代码，当接收到ping指令时，回复设备当前的时间戳，使用二进制格式进行传输。

---

```
1. //IotHub_Device/samples/pong.js
2. var IotDevice = require("../sdk/iot_device")
3. require('dotenv').config()
4. var path = require('path');
5.
6. var device = new IotDevice({
7.   productName: process.env.PRODUCT_NAME,
8.   deviceName: process.env.DEVICE_NAME,
9.   secret: process.env.SECRET,
10.  clientID: path.basename(__filename, ".js"),
11.  storePath: '../tmp/${path.basename(__filename,
12.  ".js")}'
13. })
14. device.on("online", function () {
15.   console.log("device is online")
16. })
17. device.on("command", function (command, data,
18.  respondCommand) {
19.   if (command == "ping") {
20.     console.log('get ping with:
21.     ${data.readUInt32BE(0)}')
22.     const buf = Buffer.alloc(4);
23.     buf.writeUInt32BE(Math.floor(Date.now())/1000, 0);
24.     respondCommand(buf)
25.   }
26. })
27. device.connect()
```

---

这两段代码是有实际意义的，即业务系统和设备可以通过一次指令的交互来了解它们之间数据传输的延迟状况（包括网络和IotHub处理的耗时）。

现在我们来运行上面的两段代码。

先运行`node ping.js`，然后运行`node pong.js`，我们可以看到以下输出。

---

```
## node ping.js
statusCode: 200
body: {"request_id":"5cf25cce5cb7dc80277d4641"}
received from HBG84L_M6, requestId:
5cf25cce5cb7dc80277d4641,payload:
1559387342
## node pong.js
device is online
get ping with: 1559387342
```

---

这说明设备可以接受离线消息并回复，业务系统也正确地接收了设备对指令的回复，设备回复里的RequestID和业务系统下发指令时的RequestID是一致的。

先运行`node ping.js 10`，设定指令有效期为10秒，然后在10秒内运行`node pong.js`，我们可以看到和第一步一致的输出。

先运行`node ping.js 10`，设定指令有效期为10秒，然后等待10秒，再运行`node pong.js`，控制台上不会有任何和指令相关的输出，说明指令的有效期设置是生效的。

到本节为止，IoTHub的下行数据处理功能就完成了。目前IoTHub已经可以正确地处理上行数据和下行数据了。第10章中将基于IoTHub

上行和下行数据处理的框架，做进一步的抽象处理，实现更高级的功能。

## 9.4 本章小结

本章讨论了几种可行的下行数据处理方案，并选择了基于EMQ X RESTful API的方案实现了IotHub的下行数据功能。我们可以看到，和上行数据处理一样，仍然使用主题名来携带消息的元数据。

## 第10章 IoTHub的高级功能

本章将开始设计和实现一些IoTHub更高维度抽象的功能。

## 10.1 RPC式调用

第9章实现了IotHub的指令下发功能。我们曾经把指令下发当作是一次函数 $y=f(x)$ 调用，在目前的实现中， $f(x)$ 的返回结果 $y$ 是通过异步方式告知调用者（业务系统）的，即业务系统调用下发指令接口，获得一个RequestID，设备对指令进行回复后，业务系统再从RabbitMQ队列中，使用RequestID获取对应的指令执行结果。

而RPC式调用是指当业务系统调用IotHub的发送指令接口后，IotHub会把设备对指令的回复内容直接返回给业务系统，而不再通过异步的方式（RabbitMQ）通知业务系统。程序执行的流程如图10-1所示。

- 1) 业务系统对Iot Hub Server API的下发指令接口发起HTTP Post请求。
- 2) IotHub Server调用EMQ X的指令接口。
- 3) EMQ X将指令发送到设备。
- 4) 设备执行完指令，将指令执行结果发送到EMQ X Broker。
- 5) EMQ X Broker将指令执行结果发送到IotHub Server。



6) IotHub Server API将指令结果放入HTTP Response Body中，完成对HTTP Post请求的响应。

这样，业务系统只用一次HTTP请求可以完成，检查Response Body就可以获取指令的执行结果了。

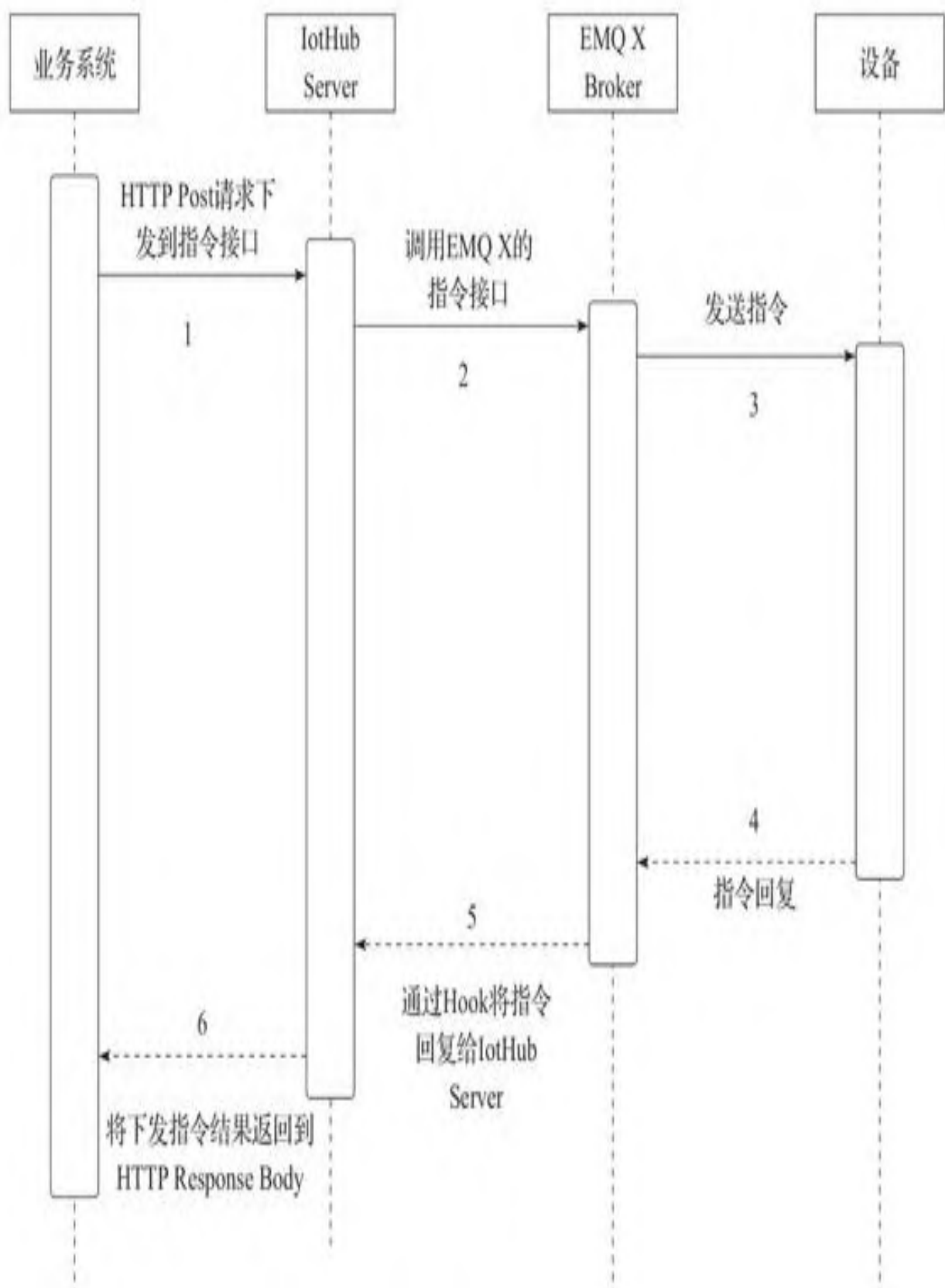


图10-1 RPC式调用流程

在RPC式调用中，如果设备在一定时间内没有对指令进行回复，那么IotHub Server API不会一直等待下去，而是在HTTP Response Body中放入错误信息（比如设备无响应）并返回给业务系统，所以指令一定是有有效期的，比如5秒。

通过这样的流程，一次RPC式调用就完成了。我们可以用这样的操作来执行一些简单的、时效性要求又比较高的指令。

## 10.1.1 主题规划

我们会使用一个特定的主题来发布RPC式调用的指令：

---

```
rpc/:ProductName/:DeviceName/:CommandName/:Encoding/:RequestID/:ExpiresAt。
```

---

可以看到，这个主题和之前用于下发指令的主题相比，除了第一个层级从cmd变成了rpc之外，其他层级都是一模一样的。因为RPC式调用也是一种下发指令的操作，所以，我们可以把下发指令的主题统一定义为：

---

```
:CommandType/:ProductName/:DeviceName/:CommandName/:Encoding/:RequestID/:ExpiresAt。
```

---

CommandType有两个可选值：“cmd”和“rpc”。

设备会把对RPC指令的回复发布到主题：

---

```
rpc_resp/:ProductName/:DeviceName/:CommandName/:RequestID/:MessageID。
```

---

同样，这个和之前回复指令的主题相比，除了第一个层级从“cmd\_resp”变成了“rpc\_resp”以外，其他层级都是一模一样的。

所以，我们可以把指令回复的主题统一定义为：

---

```
:RespType/:ProductName/:DeviceName/:CommandName/:RequestID/:MessageID。
```

---

RespType有两个可选值：“cmd\_resp”和“rpc\_resp”。

## 10.1.2 等待指令回复

业务系统在调用Server API下发RPC式指令后，IoT Hub Server需要等待设备对指令的回复，再把回复放入HTTP Response Body中，然后结束这次HTTP调用。我们可以使用Redis来帮助IoT Hub Server等待指令回复。

1) Server API的代码调用了EMQ X的Publish功能后，进一步调用Redis的Get指令来获取Redis中的key：“cmd\_resp/:RequestID”的value。如果value不为空，则将value作为指令的回复，返给业务系统；如果value为空，则需要等待一小段时间，比如10毫秒以后，重复上述操作。

2) IoT Hub Server在收到设备对RPC指令的回复以后，调用Redis的Set指令将回复的payload保存到Redis的key中：  
cmd\_resp/:RequestID。

3) 如果Server API在指定时间内仍然无法获取到key：“cmd\_resp/:RequestID”的value，则返回“错误”给业务系统。

### 10.1.3 服务端实现

首先，封装等待设备回复的过程，并将其放入一个Service类中。

---

```
1. //IotHub_Server/services/utils_service.js
2. const redisClient = require("../models/redis")
3. class UtilsService {
4.   static waitKey(key, ttl, callback) {
5.     var end = Date.now() + ttl * 1000
6.     function checkKey() {
7.       if (Date.now() < end) {
8.         redisClient.get(key, function (err, val) {
9.           if (val != null) {
10.            callback(val)
11.          } else {
12.            setTimeout(checkKey, 10)
13.          }
14.        })
15.      } else {
16.        callback(null)
17.      }
18.    }
19.  }
20.  checkKey()
21. }
22. }
23.
24. module.exports = UtilsService
```

---

waitKey方法接收TTL参数作为等待超时时间，单位为秒。每隔10毫秒检查一次设备的回复是否被放入Redis当中。

然后，修改Device类的sendCommand方法，使它可以发送RPC式指令。

---

```
1. //IotHub_Server/model/device.js
2. deviceSchema.methods.sendCommand = function
({commandName, data, encoding,
  ttl = undefined, commandType="cmd"}) {
3.   var requestId = new ObjectId().toHexString()
4.   var topic =
'${commandType}/${this.product_name}/${this.device_name}/
  ${commandName}/${encoding}/${requestId}'
5.   if (ttl != null) {
6.     topic = '${topic}/${Math.floor(Date.now() / 1000)
+ ttl}'
7.   }
8.   emqxService.publishTo({topic: topic, payload:
data})
9.   return requestId
10. }
```

---

sendCommand新增了一个参数commandType，根据这个参数来决定是发送普通指令还是RPC式指令（通过发布到不同的主题名）。

最后，在WebHook中处理RPC式指令回复，如果是RPC式指令的调用，那么将payload放入Redis对应的key中。

---

```
1. //IotHub_Server/service/message_service
2.
3. static dispatchMessage({topic, payload, ts} = {}) {
4.   ...
5.   var cmdRespRule = "
(cmd_resp|rpc_resp)/:productName/:deviceName/
:commandName/:requestId/:messageId"
6.   const cmdRespRegx = pathToRegexp(cmdRespRule)
7.   var result = null;
```

---



```

8.      ...
9.      else if ((result = cmdRespRegx.exec(topic)) !=
null) {
10.          this.checkMessageDuplication(result[6],
function (isDup) {
11.              if (!isDup) {
12.                  var payloadBuffer = new Buffer(payload,
'base64');
13.                  if (result[1] == "rpc_resp") {
14.                      var key = 'cmd_resp/${result[5]}';
15.                      redisClient.setex(key, 5, payload)
16.                  } else {
17.                      MessageService.handleCommandResp({
18.                          productName: result[2],
19.                          deviceName: result[3],
20.                          ts: ts,
21.                          command: result[4],
22.                          requestId: result[5],
23.                          payload: payloadBuffer
24.                      })
25.                  }
26.              }
27.          })
28.      }
29.  }

```

---

这里指令回复的主题的规则就变成了

“(cmd\_resp|rpc\_resp)/:productName/:deviceName/:commandName/:  
requestId/:messageId”，多了一个变量（第一个层级，指令类  
型），所以之前变量在result数组中的index要依次+1。

## 10.1.4 Server API: 发送RPC指令

我们在原有的下发指令接口上添加一个参数，用来表明是否使用RPC式调用。如果使用RPC式调用，那么最多等待设备5秒，同时将指令的有效期设为5秒。

---

```
1. //IotHub_Server/routes/devices.js
2. router.post("/:productName/:deviceName/command",
function (req, res) {
3.   var productName = req.params.productName
4.   var deviceName = req.params.deviceName
5.   var useRpc = (req.body.use_rpc == "true")
6.   Device.findOne({"product_name": productName,
"device_name": deviceName},
function (err, device) {
7.     if (err) {
8.       res.send(err)
9.     } else if (device != null) {
10.      var ttl = req.body.ttl != null ?
parseInt(req.body.ttl) : null
11.      if(useRpc){
12.        ttl = 5
13.      }
14.      var requestId = device.sendCommand({
15.        commandName: req.body.command,
16.        data: req.body.data,
17.        encoding: req.body.encoding || "plain",
18.        ttl: ttl,
19.        commandType: useRpc ? "rpc" : "cmd"
20.      })
21.      if (useRpc) {
22.        UtilsService.waitKey('cmd_resp/${requestId}',
ttl, function (val) {
23.          if(val == null){
24.            res.status(200).json({error: "device
timeout"})
}
```

```
25.             }else{
26.                 res.status(200).json({response:
27. val.toString("base64")})
28.             })
29.         } else {
30.             res.status(200).json({request_id: requestId})
31.         }
32.     } else {
33.         res.status(404).send("device not found")
34.     }
35. })
36. })
```

---

由于IoT Hub允许设备对指令回复二进制数据，所以在第26行把设备的回复进行base64编码以后再返回业务系统。

## 10.1.5 更新设备ACL列表

设备端需要将回复发布到主题

“rpc\_resp/:productName/:deviceName/:commandName/:requestId/:messageId”，所以需要把这个新的主题加入到设备的ACL列表里。

---

```
1. //IotHub_Server/models/devices
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = [
4.
5.     'upload_data/${this.product_name}/${this.device_name}/+/'+
6.     'update_status/${this.product_name}/${this.device_name}/+
7.     'cmd_resp/${this.product_name}/${this.device_name}/+/'+
8.     'rpc_resp/${this.product_name}/${this.device_name}/+/'+
9.   ]
10.   ...
11. }
```

---

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB的ACL列表。

## 10.1.6 更新服务器订阅列表

IotHub会将RPC式指令发送到

“rpc/:ProductName/:DeviceName/:CommandName/:Encoding/:RequestId/:ExpiresAt”，所以需要在EMQ X的服务器订阅列表里面添加这个主题。

---

```
## <EMQ X 安装目录>/emqx/etc/emqx.conf
module.subscription.1.topic = cmd/%u/+/+/+/#
module.subscription.1.qos = 1
module.subscription.2.topic = rpc/%u/+/+/+/#
module.subscription.2.qos = 1
```

---

然后，重启EMQ X Broker “<EMQ X安装目录>/emqx/bin/emqx restart”。

注意，这里不能用“+/%u/+/+/+/#”代替“rpc/%u/+/+/+/#”和“cmd/%u/+/+/+/#”，因为这样设备会订阅到其他不该订阅的主题。

## 10.1.7 DeviceSDK端实现

DeviceSDK的实现非常简单，只需要保证可以匹配到相应的RPC指令的主题名，并将回复发布到正确的主题上。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. dispatchMessage(topic, payload) {
3.     var cmdTopicRule = "
(cmd|rpc)/:productName/:deviceName/:commandName/
:encoding/:requestID/:expiresAt?"
4.     var result
5.     if ((result =
pathToRegexp(cmdTopicRule).exec(topic)) != null) {
6.         this.checkRequestDuplication(result[6],
function (isDup) {
7.             if (!isDup) {
8.                 self.handleCommand({
9.                     commandName: result[4],
10.                     encoding: result[5],
11.                     requestID: result[6],
12.                     expiresAt: result[7] != null ?
parseInt(result[7]) : null,
13.                     payload: payload,
14.                     commandType: result[1]
15.                 })
16.             }
17.
18.         })
19.     }
20. }
```

---

这里指令回复的主题规则变成了

“(cmd|rpc)/:productName/:deviceName/:commandName/:encoding/:

requestID/:expiresAt?”，多了一个变量（第一个层级，指令类型），所以之前变量在result数组中的index要依次+1。

然后在指令处理的代码中，将RPC式指令回复到相应的主题上。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. handleCommand({commandName, requestID, encoding,
payload, expiresAt,
commandType = "cmd"}) {
3.     if (expiresAt == null || expiresAt >
Math.floor(Date.now() / 1000)) {
4.         var data = payload;
5.         if (encoding == "base64") {
6.             data = Buffer.from(payload.toString(),
"base64")
7.         }
8.         var self = this
9.         var respondCommand = function (respData) {
10.             var topic =
'$${commandType}_resp/${self.productName}/${self.deviceName}/
${commandName}/${requestID}/${new
ObjectId().toHexString()}'
11.             self.client.publish(topic, respData, {
12.                 qos: 1
13.             })
14.         }
15.         this.emit("command", commandName, data,
respondCommand)
16.     }
17. }
```

---

对于设备应用代码来说，它并不知道指令是否是RPC式调用。不管是RPC式调用，还是普通的指令下发，设备应用代码的处理都是一样的：执行指令，然后回复结果。这是我们想要的效果。

## 10.1.8 代码联调

接下来，我们用代码来验证这个功能。仍然用之前ping/pong的例子进行演示，不过这次我们实现的是一个RPC式调用。

---

```
1. //IotHub_Server/samples/rpc_ping.js
2. require('dotenv').config({path: "../.env"})
3. const request = require("request")
4. const buf = Buffer.alloc(4);
5. buf.writeUInt32BE(Math.floor(Date.now())/1000, 0);
6. var formData = {
7.   command: "ping",
8.   data: buf.toString("base64"),
9.   encoding: "base64",
10.  use_rpc: true
11. }
12.
request.post('http://127.0.0.1:3000/devices/${process.env
.TARGET_PRODUCT_NAME}/
${process.env.TARGET_DEVICE_NAME}/command', {
13.  form: formData
14. },function (error, response, body) {
15.   if (error) {
16.     console.log(error)
17.   } else {
18.     console.log('statusCode:', response &&
response.statusCode);
19.     var result = JSON.parse(body)
20.     if(result.error != null){
21.       console.log(result.error)
22.     }else{
23.       console.log('response:',
Buffer.from(result.response, "base64").
readUInt32BE(0));
24.     }
25.   }
26. })
```



---

首先，运行 “IotHub\_Device/samples/pong.js”，然后运行 “IotHub\_Server/samples/rpc\_ping.js”，得到以下输出。

---

```
statusCode: 200  
response: 1559532366
```

---

这说明调用RPC接口已经正确获得设备对指令的回复。

然后关闭 “IotHub\_Device/samples/pong.js”，再运行 “IotHub\_Server/samples/rpc\_ping.js”，大概5秒后，得到如下输出。

---

```
statusCode: 200  
device timeout
```

---

这就说明了RPC式调用可以正确处理设备执行指令超时的情况。

本节完成了IotHub的RPC式调用功能。大家可以看到，使用RPC式调用，业务系统的代码会更少，逻辑更简单。不过，RPC式调用的缺点是它不能用于执行时间比较长的指令。

RPC式调用和我们之前实现的指令下发流程相比，就好像一个功能的同步接口和异步接口一样，按照实际情况使用就可以了。

## 10.2 设备数据请求

到目前为止，如果要把数据从服务端发送到设备端，只能使用指令下发的方式，这种方式相当于是Push的模式，即服务端主动推送数据到设备端。本节实现服务端到设备数据传输的另一种方式——Pull模式：设备数据请求。

之前的Push模式，数据的传输是由服务端触发的，而Pull模式是指由设备端主动向服务端请求数据（服务端包括业务系统和IoTHub）。

我在这里举一个很有意思的例子，比如：业务系统通过Push的方式把一些数据同步到设备中，一段时间后设备的存储故障，经过维修，换了一块新的存储，但是原有的本地数据已经丢失，这时设备就可以用Pull方式再把本地数据从业务系统主动同步过来。

由此可以发现，数据同步这个功能需要提供Push和Pull这两种语义的操作才完整。

在IoTHub里，一次设备主动数据请求的流程如图10-2所示。

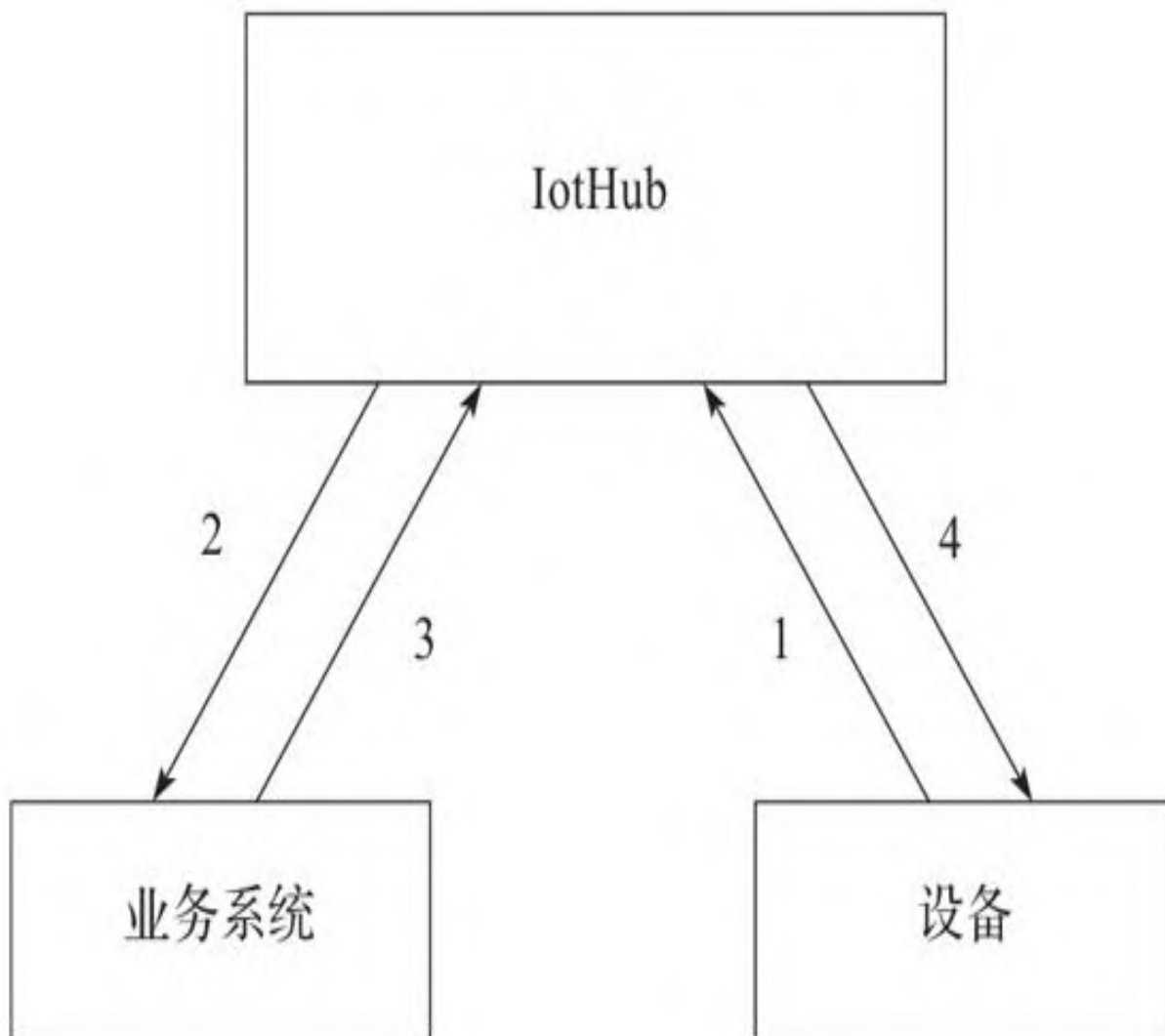


图10-2 数据请求流程

1) 设备发送数据请求到特定的主题:

“get/:ProductName/:DeviceName/:Resource/:MessageID”，其中Resource代表要请求的资源名称。

2) IotHub将请求的内容，包括DeviceName和Resource已经请求的Payload通过RabbitMQ发送给业务系统。

3) 业务系统调用指令下发接口，请求IoTHub将相应的数据下发给设备。

4) IoTHub将数据用指令的方式下发给设备，指令名称以及设备是否需要回复这个指令，由设备和业务系统约定，IoTHub不做强制要求。

我们可以把这个流程看作一次类似于HTTP Get的操作，主题中的Resource相当于Query的URL，DeviceName和消息的payload相当于查询参数，而ProductName相当于HostName，指示IoTHub把请求路由到对应的业务系统。

## 10.2.1 更新设备ACL列表

首先，需要把这个新增的主题加入设备的ACL列表。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.   const publish = [
4.
5.     'upload_data/${this.product_name}/${this.device_name}/+/'
6.   ,
7.     'update_status/${this.product_name}/${this.device_name}/+'
8.   ,
9.     'cmd_resp/${this.product_name}/${this.device_name}/+/'
10.   ,
11.     'rpc_resp/${this.product_name}/${this.device_name}/+/'
12.   ,
13.     'get/${this.product_name}/${this.device_name}/+'
14.   ]
15.   ...
16. }
```

---

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB的ACL列表。

## 10.2.2 服务端实现

服务端需要解析新的主题名，然后将相应的数据转发到业务系统。

---

```
1. //IotHub_Server/services/message_service.js
2. static dispatchMessage({topic, payload, ts} = {}) {
3.     ...
4.     var dataRequestTopicRule =
"get/:productName/:deviceName/:resource/
:messageId"
5.     const dataRequestRegx =
pathToRegexp(dataRequestTopicRule)
6.     var result = null;
7.     ...
8.     } else if((result = dataRequestRegx.exec(topic))
!= null){
9.         this.checkMessageDuplication(result[4],
function (isDup) {
10.             if(!isDup){
11.                 MessageService.handleDataRequest({
12.                     productName: result[1],
13.                     deviceName: result[2],
14.                     resource: result[3],
15.                     payload: payload
16.                 })
17.             }
18.         })
19.     }
20. }
```

---

Data Request相关的数据将会被发送到名为  
“iothub.events.data\_request” 的RabbitMQ Exchange中，Exchange

的类型为Direct, Routing key为ProductName。

---

```
1. //IotHub_Server/services/notify_service.js
2. var dataRequestRespExchange =
"iothub.events.data_request"
3. static notifyDataRequest({productName, deviceName,
resource, payload}){
4.     var data = bson.serialize({
5.         device_name: deviceName,
6.         resource: resource,
7.         payload: payload
8.     })
9.     if(currentChannel != null){
10.         currentChannel.publish(dataRequestRespExchange,
productName, data)
11.     }
12. }
1. //IotHub_Server/services/message_service.js
2. static handleDataRequest({productName, deviceName,
resource, payload}) {
3.     NotifyService.notifyDataRequest({
4.         productName: productName,
5.         deviceName: deviceName,
6.         resource: resource,
7.         payload: payload
8.     })
9. }
```

---

### 10.2.3 DeviceSDK端实现

设备端只需要实现向对应的主题发送消息就可以。当业务系统下发数据后，设备只需要把业务系统下发的数据当作一条正常的指令处理就可以。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. sendDataRequest(resource, payload = "") {
3.     if (this.client != null) {
4.         var topic =
'get/${this.productName}/${this.deviceName}/${resource}/
  ${new ObjectId().toHexString()}'
5.         this.client.publish(topic, payload, {
6.             qos: 1
7.         })
8.     }
9. }
```

---



## 10.2.4 代码联调

这里我们模拟设备向业务系统请求当前天气数据的场景。

首先，需要实现业务系统的代码。当收到设备Resource为weather的数据请求时，业务系统会下发名为weather的指令，指令数据为{temp:25,wind:4}。

---

```
1. //IotHub_Server/samples/resp_to_data_request.js
2. require('dotenv').config({path: "../.env"})
3. const bson = require('bson')
4. const request = require("request")
5. var amqp = require('amqplib/callback_api');
6. var exchange = "iothub.events.data_request"
7. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
8.   if (error0) {
9.     console.log(error0);
10.  } else {
11.    connection.createChannel(function (error1,
channel) {
12.      if (error1) {
13.        console.log(error1)
14.      } else {
15.        channel.assertExchange(exchange, 'direct',
{durable: true})
16.        var queue = "iotapp_data_request";
17.        channel.assertQueue(queue, {
18.          durable: true
19.        })
20.        channel.bindQueue(queue, exchange,
process.env.TARGET_PRODUCT_NAME)
21.        channel.consume(queue, function (msg) {
22.          var data = bson.deserialize(msg.content)
23.          if (data.resource == "weather") {
```

```

24.             console.log('received request for weather
from ${data.device_name}')
25.
request.post('http://127.0.0.1:3000/devices/${process.env
.
TARGET_PRODUCT_NAME}/${data.device_name}/command', {
26.             form: {
27.                 command: "weather",
28.                 data: JSON.stringify({temp: 25, wind:
4}),
29.             }
30.         },function (error, response, body) {
31.             if (error) {
32.                 console.log(error)
33.             } else {
34.                 console.log('statusCode:', response
&& response.statusCode);
35.                 console.log('body:', body);
36.             }
37.         })
38.     }
39.     channel.ack(msg)
40. })
41. }
42. });
43. }
44. });

```

---

在设备端发起对应的数据请求，并处理来自业务系统的相应指令。

---

```

1. //IotHub_Device/samples/send_data_request.js
2. var IotDevice = require("../sdk/iot_device")
3. require('dotenv').config()
4. var path = require('path');
5.
6. var device = new IotDevice({
7.     productName: process.env.PRODUCT_NAME,
8.     deviceName: process.env.DEVICE_NAME,
9.     secret: process.env.SECRET,

```

```
10.   clientID: path.basename(__filename, ".js"),
11.   storePath: '../tmp/${path.basename(__filename,
12.   ".js")}'
13. })
14. device.on("online", function () {
15.   console.log("device is online")
16. })
17. device.on("command", function (command, data) {
18.   if (command == "weather") {
19.     console.log('weather: ${data.toString()}')
20.     device.disconnect()
21.   }
22. })
23. device.connect()
24. device.sendDataRequest("weather")
```

---

先运行“resp\_to\_data\_request.js”，再运行  
“send\_data\_request.js”，得到如下输出。

---

```
## resp_to_data_request.js
received request for weather from yUNNHoQzv

## send_data_request.js
device is online
weather: {"temp":25,"wind":4}
```

---

本节完成了数据同步的最后一块拼图——Pull模式。设备数据请求是一个很有用的功能，10.3节将基于设备数据请求实现IoT Hub的NTP功能。

## 10.3 NTP服务

什么是NTP？这个我想大家都不陌生，NTP是同步网络中各个计算机时间的一种协议。在IoTHub中，保证设备和服务端的时间同步是非常重要的，比如指令的有效期设置就非常依赖设备和服务器间的时间同步，如果设备时间不准确，就有可能导致过期的指令仍然被执行。

通常情况下，设备上都应该运行一个NTP服务，定时地和NTP服务器进行时间同步（IoTHub服务器端也使用同样的NTP服务器进行时间同步），这样在大多数情况下可以保证设备和IoTHub服务器端的时间是同步的，除非设备掉电或者断网。

### 10.3.1 IotHub的NTP服务

某些嵌入式设备上，系统可能没有自带NTP服务，或者因为设备资源有限，无法运行NTP服务，这时候IotHub就需要基于现有的数据通道，实现一个类似于NTP服务器的时间同步功能，以满足上述情景下的设备与IotHub的时间同步。

IotHub的NTP服务实现流程如下。

- 1) 设备发起数据请求，请求NTP对时，请求中包含当前的设备时间deviceSendTime。
- 2) IotHub收到NTP对时的请求下，通过下发指令的方式将收到NTP对时请求的时间IotHubRecvTime，IotHub发送指令的时间IotHubSendTime，以及deviceTime发送到设备。
- 3) 设备收到NTP对时指令后，记录当前时间deviceRecvTime，然后通过公式  $(\text{IotHubRecvTime} + \text{IotHubSendTime} + \text{deviceRecvTime} - \text{deviceSendTime}) / 2$  获取当前的精确时间。时间的单位都为毫秒。

整个流程没有涉及业务系统，这里的数据请求和指令下发都只存在于IotHub和设备间，我们把这样的数据请求和指令都定义为IotHub的内部请求和指令，它们有如下特点：

- 数据请求的resource以\$开头；
- 指令下发的指令名以\$开头；
- payload格式统一为JSON。

这也就意味着业务系统不能发送以\$开头的指令；设备应用代码也不能通过sendDataRequest接口发送\$开头的请求；在调用时需要对输入参数进行校验。本节为了演示，跳过了输入参数校验的部分，不过在实际项目中，是不能漏掉这部分的。

## 10.3.2 DeviceSDK端实现

DeviceSDK要实现发送NTP对时请求的功能，NTP对时请求用数据请求的接口实现，这里我们约定NTP对时请求的Resource叫作\$ntp。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. sendNTPRequest() {
3.     this.sendDataRequest("$ntp",
JSON.stringify({device_time: Date.now()}))
4. }
```

---

DeviceSDK在收到IotHub下发的NTP对时指令时进行正确计算，这里我们约定NTP对时的下发指令叫作\$set\_ntp。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. handleCommand({commandName, requestID, encoding,
payload, expiresAt,
commandType = "cmd"}) {
3.     if (expiresAt == null || expiresAt >
Math.floor(Date.now() / 1000)) {
4.         ...
5.         if (commandName.startsWith("$")){
6.             if(commandName == "$set_ntp"){
7.                 this.handleNTP(payload)
8.             }
9.         } else {
10.             this.emit("command", commandName, data,
respondCommand)
11.         }
12.     }
13. }
```

---

在处理内部指令时，DeviceSDK不会通过“command”事件将内部指令的信息传递给设备应用代码。可能有的读者会觉得

“if(commandName.startsWith("\$"))”这个判断是多余的，毕竟后面还要按照指令名去对比，但是这个是有必要的，如果IoTHub的功能升级了，增加了新的内部命令，不做这个判断的话，当新的内部命令发给还未升级的DeviceSDK设备时，就会把内部命令暴露给设备应用代码。

最后计算当前的准确时间，再传递给设备应用代码。

---

```
1. //IoTHub_Device/sdk/iot_device.js
2. handleNTP(payload) {
3.     var time = Math.floor((payload.iothub_recv +
payload.iothub_send +
    Date.now() - payload.device_time) / 2)
4.     this.emit("ntp_set", time)
5. }
```

---

设备应用代码可以捕获“ntp\_set”事件来获取当前准确的时间。



### 10.3.3 服务端实现

服务端的实现很简单，收到NTP数据请求以后，将公式中需要的几个时间用指令的方式下发给设备。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleDataRequest({productName, deviceName,
resource, payload, ts}) {
3.     if(resource.startsWith("$")){
4.         if(resource == "$ntp"){
5.             this.handleNTP(payload, ts)
6.         }
7.     }else {
8.         NotifyService.notifyDataRequest(...)
9.     }
10. }
```

---

这里的检查“if(resource.startsWith(“\$”))”和DeviceSDK中类似，当IotHub弃用了某个内部数据请求时，如果不检查的话，使用还未升级的DeviceSDK设备可能会导致这个弃用的数据请求被转发至业务系统。

因为NTP要使用收到消息的时间，所以这里添加了ts参数。

接下来将NTP对时指令下发给设备。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleNTP({payload, ts, productName,
```

---

```
deviceName})) {  
  3.      var data = {  
  4.          device_time: payload.device_time,  
  5.          iothub_recv: ts * 1000,  
  6.          iothub_send: Date.now()  
  7.      }  
  8.      Device.sendCommand({  
  9.          productName: productName,  
 10.          deviceName: deviceName,  
 11.          data: JSON.stringify(data),  
 12.          commandName: "$set_ntp"  
 13.      })  
 14.  }
```

---

由于EMQ X WebHook传递过来的ts单位是秒，所以这里的计算会有误差，我们后面再解决这个问题。

### 10.3.4 代码联调

接下来我们写段代码来验证一下。

---

```
1. //IotHub_Device/samples/ntp.js
2. ...
3. device.on("online", function () {
4.     console.log("device is online")
5. })
6. device.on("ntp_set", function (time) {
7.     console.log('going to set time ${time}')
8. })
9. device.connect()
10. device.sendNTPRequest()
```

---

运行“ntp.js”，可以看到以下输出。

---

```
device is online
going to set time 1559569382108
```

---

那么，IotHub的NTP服务功能就完成了。

本节设计和实现了IotHub的NTP服务功能，这是一个非常有用的功能，在一些无法运行NTPClient的设备上显得尤为重要。

10.4节将设计和实现设备的分组和指令批量下发功能。

## 10.4 设备分组

到目前为止，IoT Hub一次只能对一个设备下发指令，假如业务系统需要对多台设备同时下发指令，那应该怎么做呢？我们可以将设备进行分组，业务系统可以通过指定指令的设备组，实现指令的批量下发。

来看一下这个场景，业务系统要关闭二楼所有的传感器（10个）。在目前的IoT Hub功能设计下，业务系统需要调用10次下发指令接口，这显然是不合理的。以MQTT协议的解决方案来说，这10个传感器都可以订阅一个topic，比如sensors/2ndfloor，只要往这个主题上发布一条消息就可以，不需要每个设备都发布一次。

这就是IoT Hub设备分组要实现的功能，业务系统可以通过IoT Hub Server API提供的接口，给设备设置一个或者多个标签，同时Server API提供接口，可以根据标签批量下发指令。这样就实现了类似于设备分组的功能，拥有相同的标签的设备就相当于属于同一分组。

这里有两个问题是设备分组功能需要解决的。

- 1) 设备如何订阅相应的标签主题。到目前为止，IoT Hub的设备端是通过EMQ X的服务器订阅功能完成订阅的，在设备分组的场景下，设备的标签是可以动态增加和删除的，所以无法使用EMQ X的服务器订阅

功能。那么我们就需要使用MQTT协议的subscribe和unsubscribe功能来完成标签的订阅。

2) 设备如何知道自己应该订阅哪些标签的主题。业务系统修改了设备的标签，IoT Hub需要将设备的标签信息告知设备，这样设备才能去subscribe和unsubscribe相应的标签主题。IoT Hub同时会使用Push和Pull模式来告知设备它的标签信息。

## 10.4.1 功能设计

我们会为设备添加标签，设备会根据标签的内容去订阅相应主题。为了确保标签的内容可以正确同步到设备，我们会设计标签同步的Push和Pull模式。

### 1. 标签字段

Device模型将新增一个tags字段，类型为数组，使一个设备可以拥有多个标签。

### 2. 标签信息同步

Push模式：当设备的标签信息发生变化，即业务系统调用Server API修改设备标签后，IoT Hub将设备标签数组通过指令下发给设备，指令名为\$set\_tags。

Pull模式：当设备连接到IoT Hub后，会发起一个Resource名为“\$tags”的数据请求，IoT Hub在收到请求后会将设备标签数组通过指令下发给设备，指令名为“\$set\_tags”。

结合Push和Pull模式，我们可以保证设备能够准确地获取自己的标签。

熟悉MQTT协议的读者可能会有一个疑问，根据MQTT协议的内容，还有一个更简单的方案：每次设备标签信息发生变化后，向一个设备相关的主题上发布一个Retained消息，里面包含标签信息不就可以了吗？这样无论设备在什么时候连接到IoT Hub都能获取到标签，不再需要Pull了。

按照MQTT协议，理论上这样是更优选择。但是实际情况和MQTT协议是有一点出入的：MQTT协议规定了如果Client不主动设置“clean\_session=true”，那么Broker应该永久为Client保存session，包括设备订阅的主题、未应答的QoS>1的消息等。但在实际情况中，Broker的存储空间是有限的，Broker不会永久保存session，大部分的Broker都会设置一个session过期时间，我们可以在“<EMQ X 安装目录>/emqx/etc/emqx.conf”里找到EMQ X client session过期时间的配置。

---

```
zone.external.session_expiry_interval = 2h
```

---

默认情况下，EMQ X Client Session的过期时间是2小时。换句话说，QoS1消息的保存时间是2小时，你可以根据项目的实际情况将其调整为更大的值。但是Broker的存储空间是有限的，session始终是需要有过期时间的，这是你在设计和架构中需要考虑到的。



注意

阿里云IoT的QoS1消息保存时间是7天。

假设我们修改了设备的标签以后，恰好设备离线超出了设置的session过期时间，那么设备就收不到标签相关的指令了。所以，这里我们加上Pull模式来保证设备能获取标签数据。

在MQTT协议架构里，Client是无法从Broker处获取自己订阅的主题的，所以设备需要在本地保存自己的标签，以便和“\$set\_tags”指令数据里面的标签进行对比，因此设备需要提供持久化的存储。

假设设备的存储坏了（这是不可避免的），存储的标签数据没有了，更换了存储标签重新接入以后，设备对比IoT Hub发来的标签数组，是无法知道它应该unsubscribe哪些标签的，所以设备可能会订阅到它不应该订阅的主题。这种情况下我的建议是设备使用新的ClientID接入。

在实际的项目中，我们一般会使用EMQ X Broker集群，如果设备的网络状态不是很稳定，有可能会出现标签指令乱序的情况，比如业务系统连续对一个设备的标签修改两次，结果第二次修改的指令比第一次修改的指令先到达，这样在设备端第一次修改的内容就会覆盖第二次修改的内容。

为了避免这种情况的发生，“\$set\_tags”指令会带一个标签信息的版本号tags\_version:



- 业务系统每次修改设备信息时，tags\_version会加1；

- 设备端收到“\$set\_tags”指令时，用指令里的tags\_version和本地保存的tags\_version对比，如果指令里的tags\_version大于本地保存的tags\_version，才会执行后续的处理。

这里的tags\_version只是用来应对MQTT Publish数据包未按照预定顺序到达设备时的情况，对于业务系统调用Server API对设备标签的并发修改，需要其他机制来应对，比如乐观锁，这和本书的主题无关，就暂行跳过，不再赘述和实现了。

### 3. 主题规划

这里我们约定设备通过标签接收下发指令的主题为：

tags/:ProductName/:tag/cmd/:CommandName/:Encoding/:RequestID/:ExpiresAt。

当设备收到“\$set\_tags”指令后，对比自己已订阅的标签和\$set\_tags指令数据里的标签数组，进而确定需要的subscribe和unsubscribe的主题。

## 10.4.2 服务端实现

### 1. 添加tags字段

在Device模型中，添加字段保存tags和tags\_version。

---

```
1. //IotHub_Server/models/device.js
2. const deviceSchema = new Schema({
3.   ...
4.   tags: {
5.     type: Array,
6.     default: []
7.   },
8.
9.   tags_version: {
10.    type: Number,
11.    default: 1
12.  }
13. })
```

---

在查询设备信息时需要返回设备的tags。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.toJSONObject = function () {
3.   return {
4.     product_name: this.product_name,
5.     device_name: this.device_name,
6.     secret: this.secret,
7.     device_status: JSON.parse(this.device_status),
8.     tags: this.tags
9.   }
10. }
```

---

## 2. 更新设备ACL列表

我们需要把设备订阅的标签主题加入设备的ACL列表中。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.     ...
4.     const subscribe =
5.     ['tags/${this.product_name}/+/cmd/+/+/+/#']
6. }
```

---

细心的读者可能会发现，这个主题名在tag这一层级也用了通配符，这样会允许Client订阅到不属于它的标签主题，但是在输出时设备对ACL是做了严格控制的，所以安全性还是可以得到保证的。这样的话每次修改设备标签时就不用修改设备的ACL列表，这是一种权衡。

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB的ACL列表。

## 3. 发送\$set\_tags指令

设备在连接到IotHub时会主动请求标签信息，离线的标签指令对设备来说没有意义，所以使用QoS0发送\$set\_tags指令。

首先需要在发送指令的方法上加上QoS参数。

---

```
1. deviceSchema.statics.sendCommand = function
  ({productName, deviceName, commandName,
   data, encoding = "plain", ttl = undefined, commandType =
   "cmd", qos = 1}) {
2.   var requestId = new ObjectId().toHexString()
3.   var topic =
   '${commandType}/${productName}/${deviceName}/${commandName}'
   '${encoding}/${requestId}'
4.   if (ttl != null) {
5.     topic = '${topic}/${Math.floor(Date.now() / 1000)
   + ttl}'
6.   }
7.   emqxService.publishTo({topic: topic, payload: data,
   qos: qos})
8.   return requestId
9. }
```

---

然后封装发送\$set\_tags指令的方法。

---

```
1. deviceSchema.methods.sendTags = function () {
2.   this.sendCommand({
3.     commandName: "$set_tags",
4.     data: JSON.stringify({tags: this.tags || [],
   tags_version: tags_version || 1}),
5.     qos: 0
6.   })
7. }
```

---

## 4. 处理设备标签数据请求

当设备发送resource名为\$tags的数据请求时，IoT Hub应该响应这个请求，并将当前设备的标签下发到设备，这里可以做一个小小的优化，在设备的标签数据请求中带上设备本地的tags\_version，只有服务端的tags\_version大于设备端的tags\_version时才下发标签指令。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleDataRequest({productName, deviceName,
resource, payload, ts}) {
3.     if (resource.startsWith("$")) {
4.         if (resource == "$ntp") {
5.             ...
6.         } else if (resource == "$tags") {
7.             Device.findOne({product_name: productName,
device_name: deviceName},
function (err, device) {
8.                 if (device != null) {
9.                     var data = JSON.parse(payload.toString())
10.                    if (data.tags_version <
device.tags_version) {
11.                        device.sendTags()
12.                    }
13.                }
14.            })
15.        }
16.    } else {
17.        ...
18.    }
19. }
```

---

## 5. Server API: 修改设备标签

Server API提供一个接口供业务系统修改设备的标签，多个标签名用逗号分隔。

---

```
1. //IotHub_Server/route/devices.js
2. router.put("/:productName/:deviceName/tags", function
(req, res) {
3.     var productName = req.params.productName
4.     var deviceName = req.params.deviceName
5.     var tags = req.body.tags.split(",")
6.     Device.findOne({"product_name": productName,
"device_name": deviceName},
function (err, device) {
```

```
7.     if (err != null) {
8.         res.send(err)
9.     } else if (device != null) {
10.        device.tags = tags
11.        device.tags_version += 1
12.        device.save()
13.        device.sendTags()
14.        res.status(200).send("ok")
15.    } else {
16.        res.status(404).send("device not found")
17.    }
18.
19. })
20. }
```

---

代码的第11行在每次修改标签后，将tags\_version加1。

## 6. Server API:批量指令下发

最后Server API需要提供接口，使业务系统可以按照标签批量下发指令。

---

```
1. //IotHub_Server/routes/tags.js
2. var express = require('express');
3. var router = express.Router();
4. const emqxService =
require("../services/emqx_service")
5. const ObjectId = require('bson').ObjectId;
6.
7. router.post("/:productName/:tag/command", function
(req, res) {
8.     var productName = req.params.productName
9.     var ttl = req.body.ttl != null ?
parseInt(req.body.ttl) : null
10.    var commandName = req.body.command
11.    var encoding = req.body.encoding || "plain"
12.    var data = req.body.data
13.    var requestId = new ObjectId().toHexString()
```

```
14.    var topic =  
    'tags/${productName}/${req.params.tag}/cmd/${commandName}  
    /  
    ${encoding}/${requestId}'  
15.    if (ttl != null) {  
16.        topic = '${topic}/${Math.floor(Date.now() / 1000)  
+ ttl}'  
17.    }  
18.    emqxService.publishTo({topic: topic, payload:  
data})  
19.    res.status(200).json({request_id: requestId})  
20. })  
21. module.exports = router
```

---

设备在回复批量下发的指令时，其流程和普通指令下发的流程一样，IoT Hub也会用同样的方式将设备对指令的回复传递给业务系统。不同的是，在批量下发指令时，针对同一个RequestID，业务系统会收到多个回复。

由于涉及多个设备的指令回复处理，批量指令下发无法提供RPC式的调用。

### 10.4.3 DeviceSDK端实现

设备在连接到IoT Hub时，需要主动请求标签数据，在收到来自服务端的标签数据时，需要对比本地存储的标签数据，然后subscribe或者unsubscribe对应的主题。

#### 1. 设备端的持久性存储

由于需要和服务端的标签进行对比，设备需要在本地使用持久化的存储来保存已订阅的标签。一般来说，DeviceSDK需要根据自身平台的特点来提供存储的接口。这里为了演示，我们使用存储in-flight消息的Message Store所使用的levelDB作为DeviceSDK的本地存储。这里我们把标签数据的存取进行封装。

---

```
1. //IoT Hub_Device/sdk/persistent_store.js
2. var level = require('level')
3. class PersistentStore {
4.   constructor(dbPath) {
5.     this.db = level('${dbPath}/device_db/')
6.   }
7.   getTags(callback) {
8.     this.db.get("tags", function (error, value) {
9.       if (error != null) {
10.         callback({tags: [], tags_version: 0})
11.       } else {
12.         callback(JSON.parse(value))
13.       }
14.     })
15.   }
}
```



```
16.  
17.   saveTags(tags) {  
18.     this.db.put("tags",  
Buffer.from(JSON.stringify(tags)))  
19.   }  
20.   close() {  
21.     this.db.close()  
22.   }  
23. }  
24.  
25. module.exports = PersistentStore;
```

---

然后在初始化时加载PersistentStore。

---

```
1. //IotHub_Device/sdk/iot_device.js  
2. const PersistentStore =  
require("../persistent_storage")  
3. constructor({serverAddress = "127.0.0.1:8883",  
productName, deviceName,  
secret, clientID, storePath} = {}) {  
4.   ...  
5.   this.persistent_store = new  
PersistentStore(storePath)  
6. }
```

---

## 2. 处理\$set\_tags指令

当收到IotHub下发的\$set\_tags指令时，DeviceSDK需要进行以下操作：

1) 将指令数据里的tags\_version和本地存储的tags\_version进行比较，如果指令的tags\_version不大于本地的tags\_version，忽略该指令，否则进入下一步；

2) 比较本地保存的tags和指令数据里的tags, 对本地有而指令里没有的tag, unsubscribe相应的主题;

3) 比较本地保存的tags和指令数据里的tags, 对本地没有而指令里有的tag, subscribe相应的主题;

4) 将指令里的tags和tags\_version存入本地存储。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. setTags(serverTags) {
3.     var self = this
4.     var subscribe = []
5.     var unsubscribe = []
6.     this.persistent_store.getTags(function
(localTags) {
7.         if (localTags.tags_version <
serverTags.tags_version) {
8.             serverTags.tags.forEach(function (tag) {
9.                 if (localTags.tags.indexOf(tag) == -1) {
10.
subscribe.push('tags/${self.productName}/${tag}/cmd/+/+/+
/#')
11.             }
12.         })
13.         localTags.tags.forEach(function (tag) {
14.             if (serverTags.tags.indexOf(tag) == -1) {
15.
unsubscribe.push('tags/${self.productName}/${tag}/cmd/+/+
/+/#')
16.             }
17.         })
18.         if(subscribe.length > 0) {
19.             self.client.subscribe(subscribe, {qos: 1},
function (err, granted) {
20.                 console.log(granted)
21.             })
22.         }
23.         if(unsubscribe.length > 0) {
```

```
24.         self.client.unsubscribe(unsubscribe)
25.     }
26.     self.persistent_store.saveTags(serverTags)
27. }
28. })
29.
30. }
```

---

代码的第7~11行将IotHub下发的标签和本地存储的标签相比较，如果某个标签不存在于本地存储的标签中，则subscribe相应的主题。

代码的第13~16行将本地存储的标签和IotHub下发的标签相比较，如果本地的某个标签不存在于IotHub下发的标签中，则unsubscribe相应的主题。

然后在接收到\$set\_tags指令时，调用setTags。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. handleCommand({commandName, requestID, encoding,
payload, expiresAt,
commandType = "cmd"}) {
3.     ...
4.     if (commandName.startsWith("$")) {
5.         payload = JSON.parse(data.toString())
6.         if (commandName == "$set_ntp") {
7.             this.handleNTP(payload)
8.         } else if (commandName == "$set_tags") {
9.             this.setTags(payload)
10.        }
11.    } else {
12.        ...
13.    }
14. }
15. }
```

---

### 3. “\$tags” 数据请求

在设备连接到IoTHub时，发起标签的数据请求。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. sendTagsRequest() {
3.     this.sendDataRequest("$tags")
4. }
5.
6. connect() {
7.     ...
8.     this.client.on("connect", function () {
9.         self.sendTagsRequest()
10.        self.emit("online")
11.    })
12.    ...
13. }
```

---

### 4. 处理批量下发指令

DeviceSDK在处理批量下发指令时，其流程和普通的指令下发没有区别，只需要匹配批量指令下发的主题名即可。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. dispatchMessage(topic, payload) {
3.     var cmdTopicRule = "
(cmd|rpc)/:productName/:deviceName/:commandName/:
encoding/:requestID/:expiresAt?"
4.     var tagTopicRule =
"tags/:productName/:tag/cmd/:commandName/:encoding/:
requestID/:expiresAt?"
5.     var result
6.     if ((result =
pathToRegexp(cmdTopicRule).exec(topic)) != null) {
7.         ...
8.     }else if ((result =
```

```
pathToRegexp(tagTopicRule).exec(topic)) != null) {
  9.      if (this.checkRequestDuplication(result[5])) {
10.          this.handleCommand({
11.              commandName: result[3],
12.              encoding: result[4],
13.              requestID: result[5],
14.              expiresAt: result[6] != null ?
parseInt(result[6]) : null,
15.              payload: payload,
16.          })
17.      }
18.  }
19. }
```

---

## 10.4.4 代码联调

### 1. 设备获取标签信息

我们写一段简单的设备应用代码。收到指令时，将指令的名称打印出来。

---

```
1. /IotHub_Device/samples/print_cmd.js
2. ...
3. device.on("online", function () {
4.   console.log("device is online")
5. })
6. device.on("command", function (command) {
7.   console.log('received cmd: ${command}')
8. })
9. device.connect()
```

---

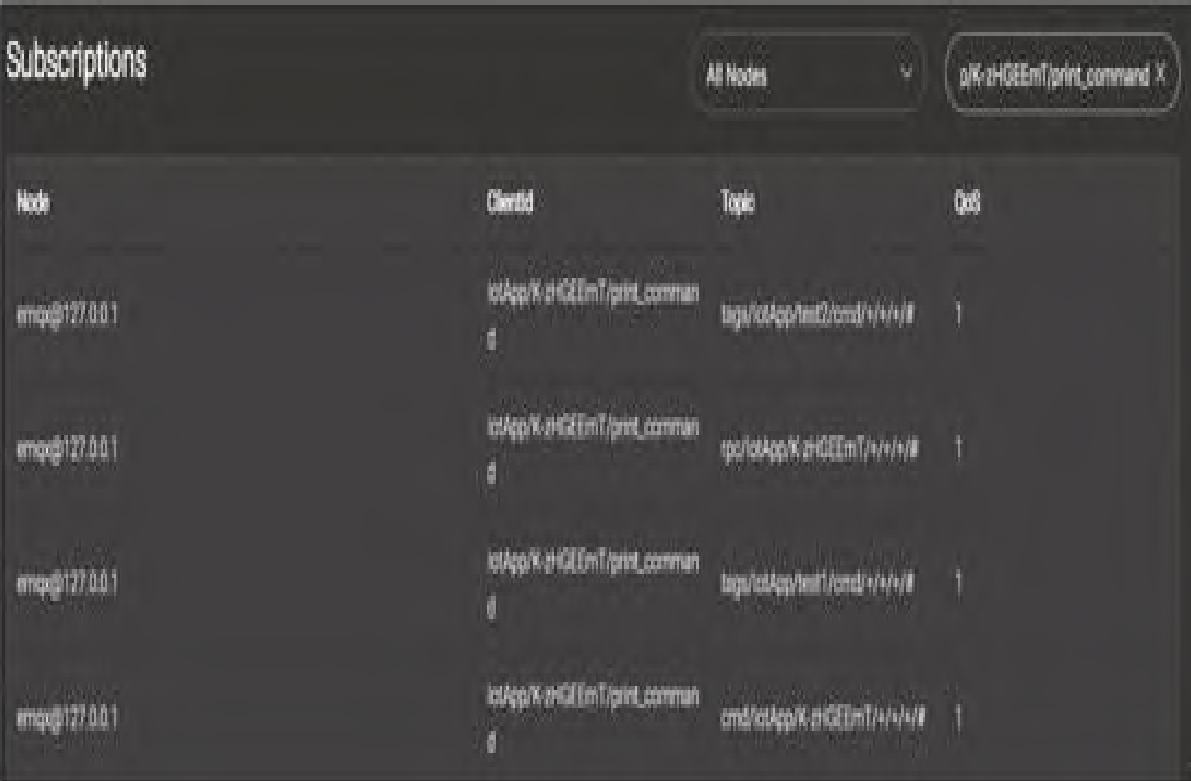
修改设备的标签为test1、test2。

---

```
curl -d "tags=test1,test2"
http://localhost:3000/devices/IotApp/K-zHGEEemT/tags -X PUT
```

---

然后运行“print\_command.js”，通过EMQ X Web Management Console检查Client的订阅情况，可以看到设备订阅了标签test1、test2对应的主题，如图10-3所示。



The screenshot shows the 'Subscriptions' page in the EMQ X Web Management Console. At the top, there is a dropdown menu set to 'All Nodes' and a search bar containing 'p/K-zHGEEt/print\_command X'. Below this is a table with four columns: 'Node', 'ClientId', 'Topic', and 'QoS'. The table lists four subscriptions, all from the same client 'p/K-zHGEEt/print\_command' to the same topic 'tags/IotApp/test1/cmd/+/#' with a QoS of 1.

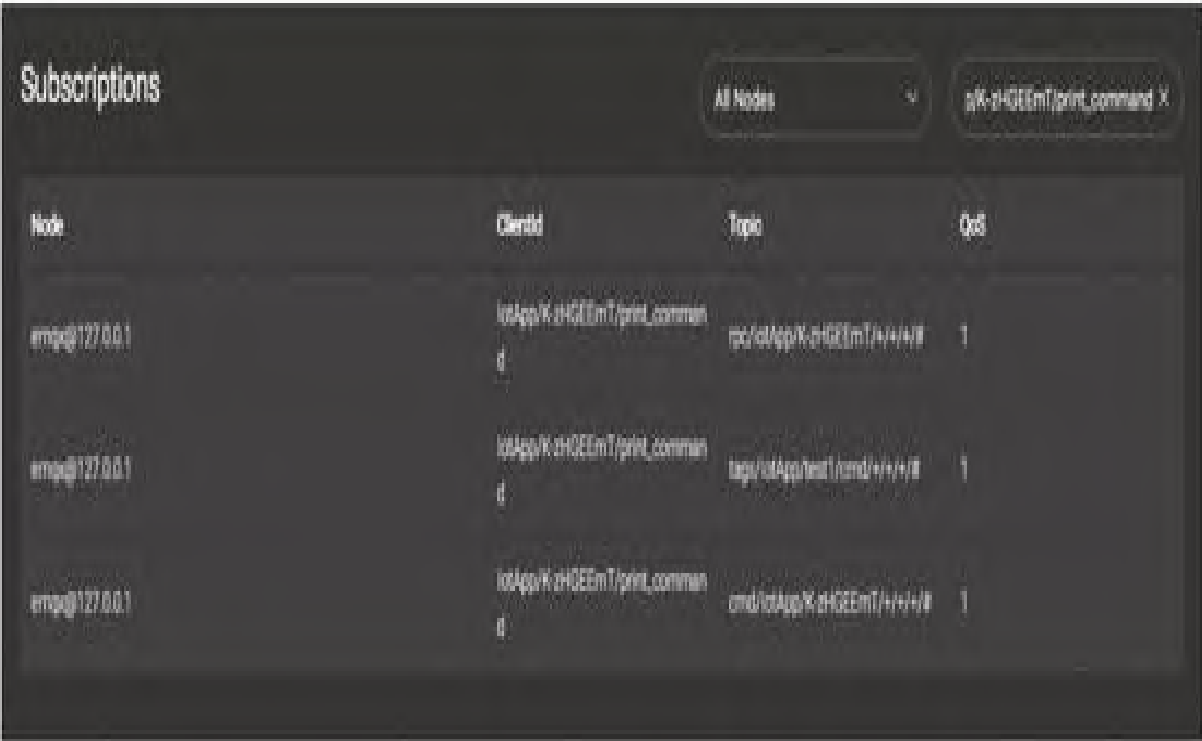
Node	ClientId	Topic	QoS
emqx@127.0.0.1	p/K-zHGEEt/print_command	tags/IotApp/test1/cmd/+/#	1
emqx@127.0.0.1	p/K-zHGEEt/print_command	tags/IotApp/test1/cmd/+/#	1
emqx@127.0.0.1	p/K-zHGEEt/print_command	tags/IotApp/test1/cmd/+/#	1
emqx@127.0.0.1	p/K-zHGEEt/print_command	tags/IotApp/test1/cmd/+/#	1

图10-3 Client的订阅列表

然后将设备的标签修改为“test1”。

```
curl -d "tags=test1"  
http://localhost:3000/devices/IotApp/K-zHGEEt/tags -X PUT
```

通过EMQ X Web Management Console检查Client的订阅情况，可以看到设备目前只订阅了标签test1对应的主题，如图10-4所示。



Node	ClientId	Topic	QoS
emsg@127.0.0.1	iotApp/K-zHGEmT/print_command	tcp/iotApp/K-zHGEmT/+/#	1
emsg@127.0.0.1	iotApp/K-zHGEmT/print_command	tags/iotApp/test1/cmd/+/#	1
emsg@127.0.0.1	iotApp/K-zHGEmT/print_command	cmd/iotApp/K-zHGEmT/+/#	1

图10-4 修改标签后的Client订阅列表

## 2. 批量下发指令

调用Server API向标签为test1的设备发送指令。

```
curl -d "command=echo"  
http://localhost:3000/tags/IotApp/test1/command -X POST
```

我们可以看到“print\_command.js”的输出如下所示。

```
device is online  
received cmd: echo
```



本节完成了设备分组的功能，这是在实际应用中非常常见和有用的功能。10.5节将实现M2M设备间通信。

## 10.5 M2M设备间通信

到目前为止，我们在MQTT协议上抽象出了服务端和设备端，数据的流向是从服务端（业务系统、IoT Hub）到设备端，或者从设备端到服务端。

在某些场景下，接入IoT Hub的设备可能还需要和其他接入的设备进行通信，例如管理终端通过P2P的方式查看监控终端的实时视频，在建立P2P连接之前，需要管理终端和监控终端进行通信，交换一些建立会话的数据。

两个不同的设备DeviceA、DeviceB作为MQTT Client接入EMQ X Broker，它们之间进行通信的流程很简单：DeviceA订阅主题TopicA，DeviceB订阅主题TopicB，如果DeviceA想向DeviceB发送信息，只需要向TopicB发布消息就可以了，反之亦然。

不过，IoT Hub和DeviceSDK需要对这个过程进行抽象和封装，DeviceSDK想对设备应用代码屏蔽掉MQTT协议层的细节，就需要做到如下功能：

- 设备间以DeviceName作为标识发送消息；

- 当DeviceA收到DeviceB的消息时，它知道这个消息是来自Device B的，可以通过DeviceB的DeviceName对DeviceB进行回复。

在IoT Hub Server端，需要控制设备间通信的范围，这里我们约定只有同一个ProductName下的设备可以相互通信。

## 10.5.1 主题名规划

为了接收其他设备发来的消息，设备会订阅主题

“m2m/:ProductName/:DeviceName/:SenderDeviceName/:MessageID”。

其中，元数据的含义如下：

- ProductName、DeviceName：和之前的使用方式一样，唯一标识一个设备（消息的接收方）；

- SenderDeviceName：消息发送方的设备名，表明消息的来源方，接收方在需要回复消息时使用；

- MessageID：消息的唯一ID，以便对消息进行去重。

也就是说，在设备间通信场景下，设备需要同时发布和订阅主题“m2m/:ProductName/:DeviceName/:SenderDeviceName/:MessageID”。

在两个设备开始通信前，发送方设备如何获取接收方设备的DeviceName，这就取决于设备和业务系统的业务逻辑了，业务系统可

以通过指令下发、设备主动数据请求等方式将这些信息告知发送方设备。

## 10.5.2 服务端实现

### 1. 更新设备ACL列表

我们需要将用于设备间通信的主题加入设备的ACL列表。

---

```
1. //Iothub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.     const publish = [
4.         ...
5.         'm2m/${this.product_name}/+/${this.device_name}/+'
6.     ]
7.     ...
8. }
```

---

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB的ACL列表。

### 2. 新增服务端订阅

接下来配置服务端订阅，让设备自动订阅这个主题。

---

```
## < EMQX 安装目录>/emqx/etc/emqx.config
module.subscription.3.topic = m2m/%u/+/+
module.subscription.3.qos = 1
```

---

然后运行“<EMQ X安装目录>/bin/emqx restart”。

## 10.5.3 DeviceSDK端实现

### 1. 发送消息

DeviceSDK实现一个方法，可以向指定的DeviceName发送消息。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. sendToDevice(deviceName, payload){
3.     if (this.client != null) {
4.         var topic =
5. 'm2m/${this.productName}/${deviceName}/${this.deviceName}
6. /
7. ${new ObjectId().toHexString()}'
8.         this.client.publish(topic, payload, {
9.             qos: 1
10.        })
11.    }
12. }
```

---

主题名里的:DeviceName层级使用消息接收方的

DeviceName; :SenderDeviceName层级使用发送方，即设备自己的

DeviceName; :ProductName层级使用发送方的ProductName，保证设备只能给属于同一ProductName的设备发送消息。

### 2. 接收消息

DeviceSDK需要处理来自设备间通信主题的消息，并用event的方式将消息和发送方传递给设备应用代码。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. dispatchMessage(topic, payload) {
3.     ...
4.     var m2mTopicRule =
"m2m/:productName/:deviceName/:senderDeviceName/:MessageI
D"
5.     var result
6.     var self = this
7.     ...
8.     else if ((result =
pathToRegexp(m2mTopicRule).exec(topic)) != null) {
9.         this.checkRequestDuplication(result[4],
function (isDup) {
10.             if (!isDup) {
11.                 self.emit("device_message", result[3],
payload)
12.             }
13.         })
14.     }
15.     ...
```

---

在代码的第11行，DeviceSDK将发送方的DeviceName和消息内容通过“device\_message”事件传递给设备应用代码。



## 10.5.4 代码联调

接下来我们写一段代码来验证这个功能，实现两个设备端互相发送ping/pong。

---

```
1. //IotHub_Device/samples/m2m_pinger.js
2. ...
3. device.on("online", function () {
4.     console.log("device is online")
5. })
6. device.on("device_message", function (sender,
payload) {
7.     console.log('received ${payload.toString()} from:
${sender}')
8.     setTimeout(function () {
9.         device.sendToDevice(sender, "ping")
10.    }, 1000)
11. })
12. device.connect()
13. device.sendToDevice(process.env.DEVICE_NAME2, "ping")
```

---

这段代码会在收到来自其他设备的消息时把消息内容和发送方打印出来，并在1秒后向对方发送消息“ping”；在代码开始运行时，代码将向一个指定的deviceName发送消息“ping”。

---

```
1. //IotHub_Device/samples/m2m_ponger.js
2. ...
3. var device = new IotDevice({
4.     productName: process.env.PRODUCT_NAME,
5.     deviceName: process.env.DEVICE_NAME2,
6.     secret: process.env.SECRET2,
7.     ...
```

```
8. })
9. device.on("online", function () {
10.   console.log("device is online")
11. })
12. device.on("device_message", function (sender,
payload) {
13.   console.log('received ${payload.toString()} from:
${sender}')
14.   setTimeout(function () {
15.     device.sendToDevice(sender, "pong")
16.   }, 1000)
17. })
18. device.connect()
```

---

这段代码会在收到来自其他设备的消息的时候把消息内容和发送方打印出来，并在1秒后向对方发送消息“pong”。

这里需要同时运行两个不同的设备端，所以新增了环境变量DEVICE\_NAME2和SECRET2来保存第二个设备的DeviceName和Secret。运行“m2m\_pinger.js和m2m\_ponger.js”，我们可以看到以下输出。

---

```
## m2m_pinger.js

received pong from: D_nSy7k7W
received pong from: D_nSy7k7W
received pong from: D_nSy7k7W
received pong from: D_nSy7k7W
...
## m2m_ponger.js

received ping from: M-lKbbY80
received ping from: M-lKbbY80
received ping from: M-lKbbY80
received ping from: M-lKbbY80
...
```

---

本节实现了IoTHub的设备间通信功能，这个功能很好地封装了底层细节，并对设备间的通信范围和权限进行了控制。10.6节将实现一个对于物联网应用来说不可或缺的功能：设备OTA升级。

## 10.6 OTA升级

OTA (Over-the-Air Technology) 一般叫作空中下载技术，在物联网应用里，设备一般都是通过OTA技术进行软件升级的，毕竟人工升级设备的成本太高了。

设备应用升级的类型可能会包括设备应用程序、固件、OS等，具体如何在设备上执行这些升级程序，各个设备的方法都不同，本书不在这方面进行论述。

不过IoTHub会对设备OTA升级的流程做一些约定，并做一定程度的抽象和封装，实现如下部分功能：

- 业务系统可以将升级的内容发送给设备，包括升级包和升级包的类型（应用、固件等）；
- IoTHub可以监控设备升级的进度，包括升级包下载的进度、安装是否成功等；
- 业务系统可以向IoTHub提供的接口查询设备的升级情况。

## 10.6.1 功能设计

### 1. 如何获取设备的软件版本号

设备的软件版本号可能包括设备应用程序、固件、操作系统等版本号。在上行数据处理功能中，IoT Hub提供了一个设备状态上报功能，在每次设备系统启动时，设备应该通过状态上报功能上报当前的软件版本号，类似如下代码。

---

```
1. var device = new IoTDevice(...)
2. device.connect()
3. device.updateStatus({app_ver: "1.1", os_ver: "9.0" })
```

---

具体版本的种类和格式，由业务系统和设备约定，IoT Hub不做强制约定。

### 2. 下发升级指令

业务系统可以通过IoT Hub的Server API获取当前的设备软件版本信息，按照业务需求决定哪些设备需要升级。业务系统还可以通过IoT Hub提供的接口向设备下发升级指令，OTA升级指令是一个IoT Hub内部使用的指令，指令数据包括：

- 将要升级的软件版本号；

- 此次升级的类型（应用、固件、OS等，由业务系统和设备约定）；
- 升级包的下载地址；
- 升级包的md5签名；
- 升级包的大小，单位为字节。

之前讲过，如果要传输较大的二进制文件，比如照片、软件升级包等，最好不要放到MQTT协议消息的Payload里面，而是将文件的URL放入Payload中，Client在收到消息以后，再通过URL去下载文件。

在进行OTA升级前，业务系统需要将升级包上传到一个设备可以访问的网络文件存储服务器中，并提供升级包的下载URL；同时需要提供升级包的md5签名，以免因为网络原因导致设备下载到不完整的升级包，进而导致升级错误。

### 3. 上报升级进度

当设备接收到升级指令后，应该按照次序执行以下操作：

- 1) 下载升级包；
- 2) 校验安装包的md5签名；

3) 执行安装/烧写;

4) 向IoTHub上报新的软件版本号。如果升级以后要重启设备应用,设备会在应用启动时,自动通过状态上报功能上报自己的软件版本号;如果升级以后不需要重启设备应用,那么设备应用代码应该使用状态上报功能上再报自己的软件版本号。

在上述流程中,设备需要上报升级进度,包括升级包下载的进度、升级中发生的错误等,设备上报的进度数据是JSON格式的,内容如下。

---

```
1. {
2.   type: "firmware",
3.   version: "1.1",
4.   progress: 70
5.   desc: "downloading"
6. }
```

---

- type: 代表此次升级的类型,比如固件、应用等;

- version: 代表此次升级的版本;

- progress: 当前的升级进度,由于只有在下载升级包时才能够保证设备应用是在运行的,所以progress只记录下载升级包的进度,取值为1~100。在安装升级包时,很多时候设备应用都是处于被关闭的状态,无法上报进度。同时progress也被当作错误码使用:-1代表下载失

败，-2代表签名校验识别失败，-3代表安装/烧写失败，-4代表其他错误导致的安装失败。

- desc：当前安装步骤的描述，也可以记录错误信息。

由于在软件包安装过程中，设备应用可能处于不可控状态，所以确定安装升级包是否成功的依据只有一个：检查设备状态中的软件版本号是否更新为期望的版本号，而只能依赖设备上报的进度数据。

#### 4. OTA升级流程

设备执行OTA升级的流程如图10-5所示（图中虚线代表该步骤可能会被重复执行多次）。



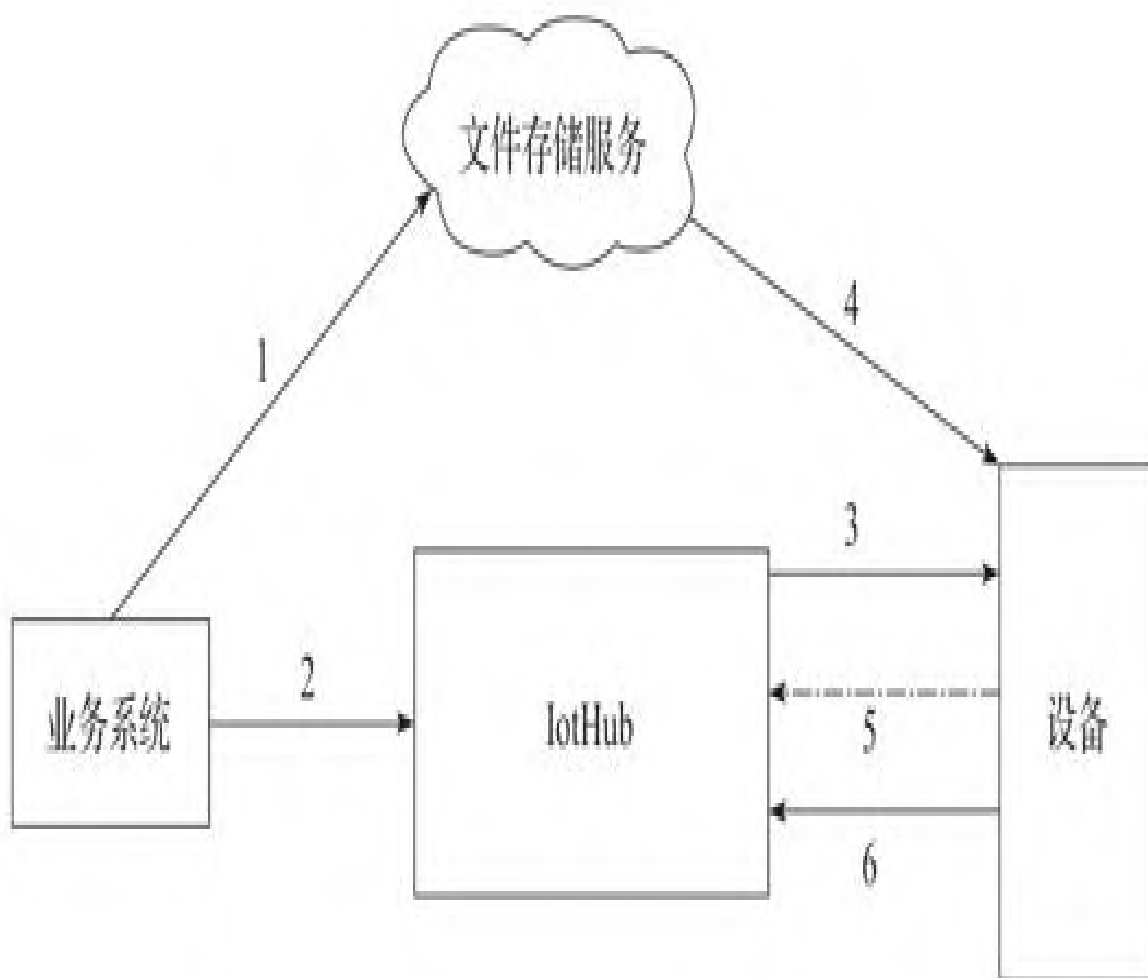


图10-5 OTA升级流程

1) 业务系统将升级文件上传到文件存储服务器，获得升级文件可下载的URL。

2) 业务系统调用IoT Hub的接口请求对设备下发OTA升级指令。OTA升级指令包含升级包URL等信息。

3) IoT Hub下发OTA指令到设备。

4) 设备通过指令数据中的升级文件URL从文件存储服务器下载升级文件。

5) 在下载和升级过程中，设备上报进度或错误信息。

6) 设备完成升级后，通过状态上报功能上报新的软件版本号。

## 10.6.2 服务端实现

### 1. 主题名规划

下发OTA升级指令使用已有的指令下发通道就可以了，但是我们需要增加一个主题名供设备上报升级进度。这里约定设备使用主题：

update\_ota\_status/:ProductName/:DeviceName/:messageID上报升级进度。

这样的话，这个新的主题就可以和我们之前使用的状态上报主题update\_status/:Product-ctName/:DeviceName/:MessageID统一为(update\_ota\_status|update\_status)/:ProductName/:DeviceName/:MessageID。

### 2. 更新设备ACL列表

我们需要将新的主题名加入Device的ACL列表。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.getACLRule = function () {
3.     const publish = [
4.         ...
5.         'update_ota_status/${this.product_name}/${this.device_name}'+',
6.     ]
```

```
7.    ...
8. }
```

---

你需要重新注册一个设备或者手动更新已注册设备存储在MongoDB中的ACL列表。

### 3. 下发OTA指令

这里我们使用“\$ota\_upgrade”作为OTA升级的指令名，同时支持单一设备下发和批量下发。

---

```
1. //IotHub_Server/services/ota_service.js
2. const Device = require("../models/device")
3. static sendOTA({productName, deviceName = null, tag =
null, fileUrl,
version, size, md5, type}) {
4.     var data = JSON.stringify({
5.         url: fileUrl,
6.         version: version,
7.         size: size,
8.         md5: md5,
9.         type: type
10.    })
11.    if (deviceName != null) {
12.        Device.sendCommand({
13.            productName: productName,
14.            deviceName: deviceName,
15.            commandName: "ota_upgrade",
16.            data: data
17.        })
18.    }else if(tag != null){
19.        Device.sendCommandByTag({
20.            productName: productName,
21.            tag: tag,
22.            commandName: "ota_upgrade",
23.            data: data
24.        })
```

```
25.     }
26. }
```

---

在代码的第11~17行，如果deviceName参数不为空，则向deviceName指定的设备下发OTA指令；代码的第18~24行，如果tag参数不为空，则向标签为tag的设备批量下发OTA指令。

为此，我们在Device类新增了一个静态方法sendCommandByTag。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.statics.sendCommandByTag =
function({productName, tag,
  commandName, data, encoding = "plain", ttl =
undefined,qos = 1}){
3.   var requestId = new ObjectId().toHexString()
4.   var topic =
'tags/${productName}/${tag}/cmd/${commandName}/${encoding
}/
  ${requestId}'
5.   if (ttl != null) {
6.     topic = '${topic}/${Math.floor(Date.now() / 1000)
+ ttl}'
7.   }
8.   emqxService.publishTo({topic: topic, payload: data,
qos: qos})
9. }
```

---

#### 4. 处理设备上报的升级进度

在IotHub中，我们把设备升级的进度放到Redis中进行存储，同时把这块业务逻辑放到一个Service类中。

---

```
1. //IotHub_Server/services/ota_service.js
2. const redisClient = require("../models/redis")
3. class OTAService{
4.   static updateProgress(productName, deviceName,
progress){
5.
redisClient.set('ota_progress/${productName}/${deviceName
}', JSON.
stringify(progress))
6.   }
7. }
8. module.exports = OTAService
```

---

然后在收到升级进度时调用这个方法。

---

```
1. //IotHub_Server/services/message_service.js
2. static dispatchMessage({topic, payload, ts} = {}) {
3.   ...
4.   var statusTopicRule=" "
(update_status|update_ota_status)/:productName/
:deviceName/:messag"Id"
5.   ...
6.   const statusRegx = pathToRegexp(statusTopicRule)
7.   ...
8.   var result = null;
9.   ...
10.  else if ((result = statusRegx.exec(topic)) !=
null) {
11.    this.checkMessageDuplication(result[4],
function (isDup) {
12.      if (!isDup) {
13.        if (result[1] "= "update_sta"us") {
14.          MessageService.handleUpdateStatus({
15.            productName: result[2],
16.            deviceName: result[3],
17.            deviceStatus: payload.toString(),
18.            ts: ts
19.          })
20.        } else if (result[1] "=
"update_ota_sta"us") {
21.          var progress =
JSON.parse(payload.toString())
```

```
22.             progress.ts = ts
23.             OTAService.updateProgress(result[2],
result[3], progress)
24.         }
25.     }
26. })
27. }
```

---

## 5. Server API: 执行OTA升级

业务系统可以调用IotHub接口向指定的设备下发OTA升级指令。

---

```
1. //IotHub_Server/routes/ota.js
2. var express = require('express');
3. var router = express.Router();
4. var Device = require("../models/device")
5. var OTAService = require("../services/ota_service")
6. router.post("/:productName/:deviceName", function
(req, res) {
7.     var productName = req.params.productName
8.     var deviceName = req.params.deviceName
9.     Device.findOne({product_name: productName,
device_name: deviceName},
    function (err, device) {
10.         if(err){
11.             res.send(err)
12.         }else if(device != null){
13.             OTAService.sendOTA({
14.                 productName: device.product_name,
15.                 deviceName: device.device_name,
16.                 fileUrl: req.body.url,
17.                 size: parseInt(req.body.size),
18.                 md5: req.body.md5,
19.                 version: req.body.version,
20.                 type: req.body.type
21.             })
22.             res.status(200).send("ok")
23.         }else{
24.             res.status(400).send("device not found")
25.         }
    }
```

```
26.    })
27. })
28. module.exports = router
```

---

读者应该注意到了，这个接口是向单一设备发送OTA升级指令的接口，通过标签批量下发的接口实现也很简单，在这里我们就跳过了。

## 6. Server API: 查询设备升级进度

业务系统可以查询某个设备的升级进度，首先把从Redis中读取升级进度的操作封装起来。

---

```
1. //IotHub_Server/services/ota_service.js
2. static getProgress(productName, deviceName, callback)
3. {
4.     redisClient.get('ota_progress/${productName}/${deviceName}', function
5.     (err, value) {
6.         if (value != null) {
7.             callback(JSON.parse(value))
8.         } else {
9.             callback({})
10.        }
```

---

然后在ServerAPI处调用这个方法。

---

```
1. //IotHub_Server/routes/ota.js
2. router.get("/:productName/:deviceName", function (req,
3. res) {
4.     var productName = req.params.productName
5.     var deviceName = req.params.deviceName
```



```
5.    OTAService.getProgress(productName, deviceName,  
function (progress) {  
6.        res.status(200).json(progress)  
7.    })  
8. })
```

---

## 10.6.3 DeviceSDK端实现

### 1. 上报升级进度

首先新增一个类来封装上报升级进度的操作。

---

```
1. //IotHub_Device/sdk/ota_progress.js
2. const ObjectId = require('bson').ObjectId;
3. class OTAProgress {
4.   constructor({productName, deviceName, mqttClient,
version, type}) {
5.     this.productName = productName
6.     this.deviceName = deviceName
7.     this.mqttClient = mqttClient
8.     this.version = version
9.     this.type = type
10.  }
11.
12.   sendProgress(progress) {
13.     var meta = {
14.       version: this.version,
15.       type: this.type
16.     }
17.     var topic =
'update_ota_status/${this.productName}/${this.deviceName}
/
${new ObjectId().toHexString()}'
18.     this.mqttClient.publish(topic,
JSON.stringify({...meta, ...progress}),
    {qos: 1})
19.   }
20.
21.   download(percent, desc = "download") {
22.     this.sendProgress({progress: percent, desc:
desc})
23.   }
24. }
```

```
25.   downloadError(desc = "download error"){
26.       this.download(-1, desc)
27.   }
28.
29.   checkMD5Error(desc = "check md5 error"){
30.       this.sendProgress({progress: -2, desc: desc})
31.   }
32.
33.   installError(desc = "install error"){
34.       this.sendProgress({progress: -3, desc: desc})
35.   }
36.
37.   error(desc = "error"){
38.       this.sendProgress({progress: -4, desc: desc})
39.   }
40. }
```

---

这个类提供了几个方法来封装OTA升级的各个节点上报进度的操作，设备应用代码只需要在相应的节点调用对应的方法就可以了。比如下载升级包时调用`progress.download(17)`，安装失败时调用`progress.installError()`。

## 2. 响应OTA指令

在收到`$ota_upgrade`指令以后，DeviceSDK代码需要把指令数据传递给设备应用代码。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. handleCommand({commandName, requestID, encoding,
payload, expiresAt,
commandType = "cmd"}) {
3.     ...
4.     if (commandName.startsWith("$")) {
5.         payload = JSON.parse(data.toString())
6.         if (commandName == "$set_ntp") {
```

```
7.         this.handleNTP(payload)
8.     } else if (commandName == "$set_tags") {
9.         this.setTags(payload)
10.    }else if(commandName == "$ota_upgrade"){
11.        var progress = new OTAProgress({
12.            productName: this.productName,
13.            deviceName: this.deviceName,
14.            mqttClient: this.client,
15.            version: payload.version,
16.            type: payload.type
17.        })
18.        this.emit("ota_upgrade", payload, progress)
19.    }
20.    }
21.    ...
22.    }
23.    }
```

---

DeviceSDK除了把OTA升级相关的数据通过“ota\_upgrade”事件传递给设备应用代码，还传递了一个OTAProgress对象，设备应用代码可以调用这个对象方法正确上报升级进度。

## 10.6.4 代码联调

接下来我们写一段代码来测试OTA升级功能。

首先实现一个设备端来模拟OTA升级。设备端收到OTA升级指令后会每隔2秒更新一次下载进度，当下载进度达到100%时，会再等待3秒，之后上报更新后的软件版本。我们用这样的方式来模拟一次成功的OTA升级流程。

---

```
1. //IotHub_Device/samples/ota_upgrade.js
2. ...
3. const currentVersion = "1.0"
4. device.on("online", function () {
5.     console.log("device is online")
6. })
7. device.on("ota_upgrade", function (ota, progress) {
8.     console.log(going to upgrade ${ota.type}:
9.     ${ota.url}, version=${ota.version})
10.    var percent = 0
11.    var performUpgrade = function () {
12.        console.log('download:${percent}')
13.        progress.download(percent)
14.        if(percent < 100){
15.            percent += 20
16.            setTimeout(performUpgrade, 2000)
17.        }else{
18.            setTimeout(function () {
19.                device.updateStatus({firmware_ver:
20.                ota.version})
21.            }, 3000)
22.        }
23.    }
24.    performUpgrade()
25. })
```

```
24. device.connect()
25. device.updateStatus({firmware_ver: currentVersion})
```

---

然后写一段代码模拟业务系统：通过Server API向设备发送OTA指令以后，每隔1秒查询一次设备的升级进度，当进度达到100%后，就检查设备的软件版本，如果设备的软件版本和预期一致，就说明设备已经升级成功。

---

```
1. //IotHub_Server/samples/perform_ota.js
2. ...
3. var otaData = {
4.   type: "firmware",
5.   url: "http://test.com/firmware/1.1.pkg",
6.   version: "1.1",
7.   size: 1000,
8.   md5: "abcd"
9. }
10. var progress = 0
11. var checkUpgradeProgress = function () {
12.   if (progress < 100) {
13.
14.     request.get('http://127.0.0.1:3000/ota/${process.env.TARGET_PRODUCT_NAME}/
15.     ${process.env.TARGET_DEVICE_NAME}', function (err, res,
16.     body) {
17.       if (!err && res.statusCode == 200) {
18.         var info = JSON.parse(body);
19.         if(info.version == otaData.version) {
20.           progress = info.progress
21.           console.log('current
22.           progress:${progress}%');
23.         }
24.         setTimeout(checkUpgradeProgress, 1000)
25.       }
26.     })
27.   } else {
28.
29.     request.get('http://127.0.0.1:3000/devices/${process.env.
```

```

TARGET_PRODUCT_
NAME}/${process.env.TARGET_DEVICE_NAME}', function (err,
res, body) {
25.     if (!err && res.statusCode == 200) {
26.         var info = JSON.parse(body);
27.         console.log('current
version:${info.device_status.firmware_ver}');
28.         if (info.device_status.firmware_ver ==
otaData.version) {
29.             console.log('upgrade completed');
30.         } else {
31.             setTimeout(checkUpgradeProgress, 1000)
32.         }
33.     }
34. })
35. }
36. }
37.
38. console.log("perform upgrade")
39.
request.post('http://127.0.0.1:3000/ota/${process.env.TAR
GET_PRODUCT_NAME}/
${process.env.TARGET_DEVICE_NAME}', {
40.   form: otaData
41. }, function (error, response) {
42.   if (error) {
43.       console.log(error)
44.   } else {
45.       console.log('statusCode:', response &&
response.statusCode);
46.       checkUpgradeProgress()
47.   }
48. })

```

---

先运行IotHub\\_Device/samples/ota\\_upgrade.js, 然后运行IotHub\\_Server/samples/perform\_ota.js, 可观察到如下输出。

---

```

## IotHub_Device/samples/ota_upgrade.js
device is online
going to upgrade firmware:

```

```
http://test.com/firmware/1.1.pkg, version=1.1
download:0
download:20
download:40
download:60
download:80
download:100
upgrade completed
## IotHub_Server/samples/perform_ota.js
perform upgrade
statusCode: 200
current progress:0%
current progress:0%
current progress:20%
current progress:20%
current progress:40%
current progress:60%
current progress:60%
current progress:80%
current progress:80%
current progress:100%
current version:1.0
current version:1.0
current version:1.1
upgrade completed
```

---

这说明OTA升级功能能够正常工作了。

本节完成了IotHubOTA升级的功能，OTA功能在大多数的物联网应用中都会用得到，IotHub的OTA功能为设备和业务系统封装了很多细节，并约定了流程。10.7节将设计和实现一个新的功能：设备影子。



## 10.7 设备影子

### 10.7.1 什么是设备影子

我最早是在AWS IoT上面看到设备影子，后来国内主流云服务上的IoT套件中都包含了设备影子。

我们首先来看一下各个平台对设备影子的描述。

#### (1) 阿里云

物联网平台提供设备影子功能，用于缓存设备状态。设备在线时，可以直接获取云端指令；设备离线时，上线后可以主动拉取云端指令。

设备影子是一个JSON文档，用于存储设备上报状态和应用程序期望状态信息。

每个设备有且只有一个设备影子，设备可以通过MQTT协议获取和设置设备影子，并同步状态。该同步可以是影子同步给设备，也可以是设备同步给影子。

#### (2) 腾讯云

设备影子文档是服务器端为设备缓存的一份状态和配置数据。它以JSON文本形式存储。

简单来说，设备影子包含了两种主要功能：服务端和设备端数据同步，设备端数据/状态缓存。

### （1）服务端和设备端数据同步

设备影子提供了一种在网络情况不稳定、设备上下线频繁的情况下，服务端和设备端稳定实现数据同步的功能。

这里要说明一下，IoT Hub之前实现的数据/状态上传、指令下发功能都是可以在网络情况不稳定的情况下，稳定实现单向数据同步的。

设备影子主要解决的是：当一个状态或者数据可以被设备和服务端同时修改时，在网络状态不稳定的情况下，如何保持其在服务端和设备端状态的一致性。当你需要双向同步时，就可以考虑使用设备影子了。例如，智能灯泡的开关状态既可以远程改变（比如通过手机App进行开关），也可以在本地图通过物理开关改变，那么就可以使用设备影子，使得这个状态在服务端和设备端保持一致。

### （2）设备端数据/状态缓存

设备影子还可以作为设备状态/数据在服务端的缓存，由于它保证了设备端和服务端的一致性，因此在业务系统需要获取设备上的某个

状态时，只需要读取服务端的数据就可以了，不需要和设备进行交互，实现了设备和业务系统的解耦。

## 10.7.2 设备影子的数据结构

像引用的阿里云和腾讯云的文档里说的那样，设备影子是一个JSON格式的文档，每个设备对应一个设备影子。下面是一个典型的设备影子。

---

```
1. {
2.   "state": {
3.     "reported": {
4.       "lights": "on"
5.     },
6.     "desired": {
7.       "lights": "off"
8.     }
9.   },
10.  "metadata": {
11.    "reported": {
12.      "lights": {
13.        "timestamp": 123456789
14.      }
15.    },
16.    "desired": {
17.      "lights": {
18.        "timestamp": 123456789
19.      }
20.    }
21.  },
22.  "version": 1,
23.  "timestamp": 123456789
24. }
```

---

state包含以下一些字段：

- reported: 指当前设备上报的状态，业务系统如果需要读取当前设备的状态，以这个值为准；

- desired: 指服务端希望改变的设备状态，但还未同步到设备上。

- metadata: 状态的元数据，内容是state中包含的状态字段的最后更新时间。

- version: 设备影子的版本。

- timestamp: 设备影子的最后一次修改时间。

### 10.7.3 设备影子的数据流向

阿里云和腾讯云的设备影子的数据流向大体一致，细节上略有不同，这里我总结和简化了一下，在IoTHub里，设备影子的数据流向可分为两个方向。

#### 1. 服务端向设备端同步

当业务系统通过服务端的接口修改设备影子后，IoTHub会向设备端进行同步，这个流程分为如下4步。

第一步，IoTHub向设备下发指令UPDATE\_SHADOW，指令中包含了更新后的设备影子文档。以前面的设备影子文档为例，其中最重要的部分是desired和version。

---

```
1. {
2.   "state": {
3.     ...
4.     "desired": {
5.       "lights": "off"
6.     }
7.   },
8.   ...
9.   "version": 1,
10.  ...
11. }
```

---

第二步，设备根据desired里面的值更新设备的状态，这里应该是关闭智能灯。

第三步，设备向IoT Hub回复状态更新成功的信息。

---

```
1. {
2.   "state": {
3.     "desired": null
4.   },
5.   "version": 1,
6. }
```

---

这里设备必须使用第二步得到version值，当IoT Hub收到这个回复时，要检查回复里的version是否和设备影子中的version一致。如果一致，那么将设备影子中reported里面字段的值修改为与desired对应的值，同时删除desired，并修改metadata里面相应的值，修改过后的设备影子文档如下。

---

```
1. {
2.   "state": {
3.     "reported": {
4.       "lights": "off"
5.     }
6.   },
7.   "metadata": {
8.     "reported": {
9.       "lights": {
10.        "timestamp": 123456789
11.      }
12.     }
13.   },
14.   "version": 1,
```

```
15.  "timestamp": 123456789
16. }
```

---

文档中desired字段被删除了，同时state中的reported字段从“{"lights":"on"}”变成了“{"lights":off}”。

如果不一致，则说明在此期间设备影子又被修改了，那么回到第一步，重新执行。

第四步，设备影子更新成功后，IoT Hub向设备回复一条消息SHADOW\_REPLY。

---

```
1. {
2.   status: "succss",
3.   "timestamp": 123456789,
4.   "version": 1
5. }
```

---

## 2. 设备端向服务端同步

设备端的流程有如下3步。

第一步，当设备连接到IoT Hub时，向IoT Hub发起数据请求，IoT Hub收到请求后会下发UPDATE\_SHADOW指令，执行一次服务端向设备端同步，设备需要记录下当前设备影子的version。



第二步，当设备的状态发生变化，比如通过物理开关关闭智能电灯时，IoT Hub发送REPORT\_SHADOW数据，包含第一步获得的version，代码如下。

---

```
1. {
2.   "state": {
3.     "reported": {
4.       "lights": "off"
5.     }
6.   },
7.   "version": 1
8. }
```

---

当IoT Hub收到这个数据后，检查REPORT\_SHADOW里的version是否和设备影子中的数据一致。

- 如果一致，那么用REPORT\_SHADOW里的reported值修改设备影子中reported的字段。

- 如果不一致，那么IoT Hub会下发指令UPDATE\_SHADOW，再执行一次服务端向设备端的同步。

## 10.7.4 服务端实现

服务端需要对设备影子进行存储。在业务系统修改设备影子时，需要将设备影子同步到设备端，同时还需要处理来自设备的设备影子同步消息，将设备端的数据同步到数据库中。

最后服务端还要提供接口供业务系统查询和修改设备影子。

### 1. 存储设备影子

我们在Device模型里新增一个字段“shadow”来保存设备影子，一个空的设备影子如下所示。

---

```
1. {  
2.   "state": {},  
3.   "metadata": {},  
4.   "version": 0  
5. }
```

---

按照上述代码设置这个字段的默认值。

---

```
1. const deviceSchema = new Schema({  
2.   ...  
3.   shadow: {  
4.     type: String,  
5.     default: JSON.stringify({  
6.       "state": {},  
7.       "metadata": {},
```

---

```
8.         "version":0
9.     })
10. }
11. })
```

---

## 2. 下发设备影子相关指令

IotHub需要向设备发送2种与设备影子相关的指令，一种是更新设备影子，这里使用指令名“\$update\_shadow”，另一种是成功更新设备影子后，对设备的回复信息，这里使用指令名“\$shadow\_reply”。发送这两条指令使用IotHub指令下发通道就可以了。

## 3. Server API:更新设备影子

IotHub提供一个接口供业务系统修改设备影子，它需要接收一个JSON对象“{desired:{key1=value1,...},version=xx}”作为参数，业务系统在调用时需要提供设备影子的版本，以避免业务系统用老版本数据覆盖当前的新版本数据。

---

```
1. //IotHub_Server/routes/devices.js
2. router.put("/:productName/:deviceName/shadow",
function (req, res) {
3.     var productName = req.params.productName
4.     var deviceName = req.params.deviceName
5.     Device.findOne({"product_name": productName,
"device_name": deviceName},
function (err, device) {
6.         if (err != null) {
7.             res.send(err)
8.         } else if (device != null) {
9.             if(device.updateShadowDesired(req.body.desired,
```

```
req.body.version)){  
10.         res.status(200).send("ok")  
11.     }else{  
12.         res.status(409).send("version out of date")  
13.     }  
14. } else {  
15.     res.status(404).send("device not found")  
16. }  
17. })  
18. })
```

---

如果业务系统提交的version大于当前的设备影子version，则更新设备影子的desired字段，以及相关的metadata字段，更新成功后向设备下发指令“\$update\_shadow”。

---

```
1. //IotHub_Server/models/device.js  
2. deviceSchema.methods.updateShadowDesired = function  
(desired, version) {  
3.     var ts = Math.floor(Date.now() / 1000)  
4.     var shadow = JSON.parse(this.shadow)  
5.     if (version > shadow.version) {  
6.         shadow.state.desired = shadow.state.desired || {}  
7.         shadow.metadata.desired = shadow.metadata.desired  
|| {}  
8.         for (var key in desired) {  
9.             shadow.state.desired[key] = desired[key]  
10.            shadow.metadata.desired[key] = {timestamp: ts}  
11.        }  
12.        shadow.version = version  
13.        shadow.timestamp = ts  
14.        this.shadow = JSON.stringify(shadow)  
15.        this.save()  
16.        this.sendUpdateShadow()  
17.        return true  
18.    } else {  
19.        return false  
20.    }  
21. }  
22. deviceSchema.methods.sendUpdateShadow= function(){
```

```
23.     this.sendCommand({
24.         commandName: "$update_shadow",
25.         data: this.shadow,
26.         qos: 0
27.     })
```

---

因为设备在连接时还会主动请求一次影子数据，所以这里使用 qos=0就可以了。

#### 4. 响应设备端影子消息

设备端会向IoTHub发送3种与设备影子相关的消息，IoTHub Server需要对这些消息进行回应：

- 设备主动请求设备影子数据，使用设备数据请求的通道，收到 resource 为 “\$shadow” 的数据请求；
- 设备更新完状态后向IoTHub回复的消息，这里我们使用上传数据的通道，将DataType设为 “\$shadow\_updated” ；
- 设备主动更新影子数据，这里我们使用上传数据的通道，将 DataType 设为 “\$shadow\_reporeted” 。

##### （1）影子数据请求

在收到resource名为 “\$shadow” 的数据请求后，IoTHub应该下发 “\$update\_shadow” 指令。

---

```

1. //IotHub_Server/services/message_service.js
2. static handleDataRequest({productName, deviceName,
resource, payload, ts}) {
3.     if (resource.startsWith("$")) {
4.         ...
5.     } else if (resource == "$shadow_updated") {
6.         Device.findOne({product_name: productName,
device_name: deviceName},
function (err, device) {
7.             if (device != null) {
8.                 device.sendUpdateShadow()
9.             }
10.        })
11.    }
12. }
13. ...
14. }

```

---

## (2) 设备状态更新完成以后的回复

在收到“DataType=“\$shadow\_updated””的上传数据后，IotHub应该按照数据的内容对设备影子进行更新。

---

```

1. //IotHub_Server/service/message_service.js
2. static handleUploadData({productName, deviceName, ts,
payload, messageId,
dataType} = {}) {
3.     if (dataType.startsWith("$")) {
4.         if (dataType == "$shadow_updated") {
5.             Device.findOne({product_name: productName,
device_name: deviceName},
function (err, device) {
6.                 if (device != null) {
7.                     device.updateShadow(JSON.parse(payload.toString()))
8.                 }
9.             })
10.        }
11.    } else {

```

```
12.      ...
13.    }
14.  }
```

---

更新时需要先检查回复的version，如果此时desired中的字段值为null，则需要在reported里面删除相应的字段，更新成功后需要回复设备。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.updateShadow = function
(shadowUpdated) {
3.   var ts = Math.floor(Date.now() / 1000)
4.   var shadow = JSON.parse(this.shadow)
5.   if (shadow.version == shadowUpdated.version) {
6.     if (shadowUpdated.state.desired == null) {
7.       shadow.state.desired = shadow.state.desired ||
{}
8.       shadow.state.reported = shadow.state.reported
|| {}
9.       shadow.metadata.reported =
shadow.metadata.reported || {}
10.      for (var key in shadow.state.desired) {
11.        if (shadow.state.desired[key] != null) {
12.          shadow.state.reported[key] =
shadowUpdated.state.desired[key]
13.          shadow.metadata.reported[key] = {timestamp:
ts}
14.        } else {
15.          delete(shadow.state.reported[key])
16.          delete(shadow.metadata.reported[key])
17.        }
18.      }
19.      shadow.timestamp = ts
20.      shadow.version = shadow.version + 1
21.      delete(shadow.state.desired)
22.      delete(shadow.metadata.desired)
23.      this.shadow = JSON.stringify(shadow)
24.      this.save()
25.      this.sendCommand({
```

```
26.         commandName: "$shadow_reply",
27.         data: JSON.stringify({status: "success",
timestamp: ts, version:
  shadow.version}),
28.         qos: 0
29.     })
30. }
31. } else {
32.     this.sendUpdateShadow()
33. }
34. }
```

---

如设备影子的version和上报的version不一致，IotHubServer需要再发起一次服务端向设备端的同步（代码第32行）。

### （3）设备主动更新影子

在收到DataType="\$shadow\_reported"的上传数据后，IotHub应该按照数据的内容对设备影子进行更新。

---

```
1. //IotHub_Server/services/message_service.js
2. static handleUploadData({productName, deviceName, ts,
payload, messageId,
  dataType} = {}) {
3.     if (dataType.startsWith("$")) {
4.         ...
5.         else if(datatype == "$shadow_reported"){
6.             Device.findOne({product_name: productName,
device_name: deviceName},
  function (err, device) {
7.                 if (device != null) {
8.
device.reportShadow(JSON.parse(payload.toString()))
9.                 }
10.            })
11.        }
12.    }
```



```
13.     ...
14. }
```

---

在更新设备影子时也需要检查version和null字段。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.reportShadow = function
(shadowReported) {
3.     var ts = Math.floor(Date.now() / 1000)
4.     var shadow = JSON.parse(this.shadow)
5.     if (shadow.version == shadowReported.version) {
6.         shadow.state.reported = shadow.state.reported ||
{}
7.         shadow.metadata.reported =
shadow.metadata.reported || {}
8.         for (var key in shadowReported.state.reported) {
9.             if (shadowReported.state.reported[key] != null)
{
10.                 shadow.state.reported[key] =
shadowReported.state.reported[key]
11.                 shadow.metadata.reported[key] = {timestamp:
ts}
12.             } else {
13.                 delete(shadow.state.reported[key])
14.                 delete(shadow.metadata.reported[key])
15.             }
16.         }
17.         shadow.timestamp = ts
18.         shadow.version = shadow.version + 1
19.         this.shadow = JSON.stringify(shadow)
20.         this.save()
21.         this.sendCommand({
22.             commandName: "$shadow_reply",
23.             data: JSON.stringify({status: "success",
timestamp: ts, version:
shadow.version}),
24.             qos: 0
25.         })
26.     } else {
27.         this.sendUpdateShadow()
28.     }
29. }
```

---

## 5. Server API: 查询设备影子

只需要在设备详情接口返回设备影子的数据就可以了。

---

```
1. //IotHub_Server/models/device.js
2. deviceSchema.methods.toJSONObject = function () {
3.   return {
4.     ...
5.     shadow: JSON.parse(this.shadow),
6.   }
7. }
```

---

## 10.7.5 DeviceSDK端实现

设备端需要处理来自IoT Hub Server的设备影子同步指令，同时在本地状态发生变化时，向IoT Hub Server发送相应的数据。

### 1. 设备影子数据请求

在设备连接到IoT Hub时，需要主动发起一个数据请求，请求设备影子的数据。

---

```
1. //IoT Hub Device/sdk/iot_device.js
2. this.client.on("connect", function () {
3.     self.sendTagsRequest()
4.     self.sendDataRequest("$shadow")
5.     self.emit("online")
6. })
```

---

### 2. 处理“\$update\_shadow”指令

DeviceSDK在处理“\$update\_shadow”指令时有两件事情要做：

第一，如果desired不为空，要将desired数据传递给设备应用代码；

第二，需要提供接口供设备应用代码在更新完设备状态后向IoT Hub Server回复。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. handleCommand({commandName, requestID, encoding,
payload, expiresAt,
commandType = "cmd"}) {
3.     ...
4.     else if (commandName == "$update_shadow") {
5.         this.handleUpdateShadow(payload);
6.     }
7.     ...
8. }
9.
10. handleUpdateShadow(shadow) {
11.     if (this.shadowVersion <= shadow.version) {
12.         this.shadowVersion = shadow.version
13.         if (shadow.state.desired != null) {
14.             var self = this
15.             var respondToShadowUpdate = function () {
16.                 self.uploadData(JSON.stringify({
17.                     state: {
18.                         desired: null
19.                     },
20.                     version: self.shadowVersion
21.                 })), "$shadow_updated")
22.             }
23.             this.emit("shadow", shadow.state.desired,
respondToShadowUpdate)
24.         }
25.     }
26. }
```

---

这里同样使用一个闭包封装对IotHub Server进行回复，并传递给设备应用代码（代码的第15~22行）

this.shadowVersion初始化为0。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. constructor(...) {
3.     ...
```

```
4.     this.shadowVersion = 0
5. }
```

---

### 3. 主动更新影子设备状态

设备可以上传DataType="\$shadow\_reported"的数据主动修改设备影子状态，下面我们通过一个reportShadow方法来完成这个操作。

---

```
1. //IotHub_Device/sdk/iot_device.js
2. reportShadow(reported) {
3.     this.uploadData(JSON.stringify({
4.         state: {
5.             reported: reported
6.         },
7.         version: this.shadowVersion
8.     }), "$shadow_reported")
9. }
```

---

### 4. 处理 "\$shadow\_reply" 指令

当IotHub Server根据设备上传的数据成功修改设备影子后，IotHub Server会下发 "\$shadow\_reply" 指令。这个指令的处理逻辑很简单，如果指令携带的version大于本地的version，那么就将本地的version更新为指令携带的version。

---

```
1. handleCommand({commandName, requestId, encoding,
payload, expiresAt,
commandType = "cmd"}) {
2.     ...
3.     else if (commandName == "$update_shadow") {
4.         this.handleUpdateShadow(payload);
```

```
5.          } else if (commandName == "$shadow_reply") {
6.          if (payload.version > this.shadowVersion &&
payload.status ==
"success") {
7.          this.shadowVersion = payload.version
8.          }
9.          }
10.         }
11.         ...
12.
13.     }
```

---

## 10.7.6 代码联调

接下来用代码测试IoTHub的设备影子功能，模拟下面场景：

- 1) 设备离线时，业务系统将设备影子状态设置为desired:  
`{lights:"on"};`
- 2) 设备上线后，业务系统更改设备影子状态为lights=on;
- 3) 业务系统再次查询设备影子，此时设备影子状态为reported:  
`{lights:"on"};`
- 4) 设备在3秒后主动将设备影子状态更改为reporeted:  
`{lights:"off"};`
- 5) 业务系统再次查询设备影子，此时设备影子状态为reported:  
`{lights:"off"}。`

---

```
1. //IotHub_Server/samples/update_shadow.js
2. require('dotenv').config({path: "../.env"})
3. const request = require("request")
4.
5. var deviceUrl =
'http://127.0.0.1:3000/devices/${process.env.TARGET_
PRODUCT_NAME}/${process.env.TARGET_DEVICE_NAME}';
6.
7. var checkLights = function () {
8.   request.get(deviceUrl
```

```
9.      , function (err, response, body) {
10.      var shadow = JSON.parse(body).shadow
11.      var lightsStatus = "unknown"
12.      if(shadow.state.reported &&
shadow.state.reported.lights){
13.          lightsStatus = shadow.state.reported.lights
14.      }
15.      console.log('current lights status is
${lightsStatus}')
16.      setTimeout(checkLights, 2000)
17.  })
18. }
19. request.get(deviceUrl
20. , function (err, response, body) {
21.     var deviceInfo = JSON.parse(body)
22.     request.put('${deviceUrl}/shadow', {
23.         json: {
24.             version: deviceInfo.shadow.version + 1,
25.             desired: {lights: "on"}
26.         }
27.     }, function (err, response, body) {
28.         checkLights()
29.     })
30. })
```

---

代码在更新设备影子接口后，每隔2秒检查一次当前设备的设备影子。

注意在调用接口修改设备影子前，先调用接口查询当前设备影子的version，以当前的version+1作为设备新的影子version（代码第24行）。

设备端的应用代码很简单，先运行  
“`IotHub_Server/samples/update_shadow.js`”，然后运行



“IoT\_Hub\_Device/samples/resp\_to\_shadow.js”，我们可以观察到以下输出。

---

```
### IoT_Hub_Server/samples/update_shadow.js
current lights status is unknown
current lights status is unknown
current lights status is unknown
current lights status is on
current lights status is off
current lights status is off

### IoT_Hub_Device/samples/resp_to_shadow.js
device is online
turned the lights on
turned the lights off
```

---

这说明IoT\_Hub的设备影子功能是按照预期在工作的。

本节完成了目前各大主流云IoT平台都具备的设备影子功能，在同步某些在服务端和设备端都会修改的状态时，使用设备影子是非常方便的。

10.8节将讨论如何对IoT\_Hub的状态进行监控。

## 10.8 IotHub的状态监控

作为一个服务多个产品的物联网平台，目前我们还缺少一个功能，即如何监控当前物联网平台的运行情况。这里要讨论的不是监控内存、硬盘I/O、网络等，也不是监控WebServer、MongoDB的运行状态，因为这些监控都已经有了非常成熟的方案了。我们要讨论的是如何监控EMQ X Broker的运行状态，比如在线设备的数量有多少，IotHub一共发送了多少消息、接收了多少消息等。

EMQ X Broker提供了两种对运行状态进行监控的方式，接下来分别对这两种方法进行阐述。

## 10.8.1 使用RESTful API

我们之前已经多次用到EMQ X的RESTful API了，EMQ X的RESTful API中也提供了获取Broker当前运行状态的接口。以连接数据统计为例，可以访问下面这个接口。

---

```
GET api/v3/stats/
```

---

RESTful API会返回各个节点的连接信息统计数据，格式如下。

---

```
1. {
2.   "code": 0,
3.   "data": [
4.     {
5.       "node": "emqx@127.0.0.1",
6.       "subscriptions/shared/max": 0,
7.       "subscriptions/max": 2,
8.       "subscribers/max": 2,
9.       "topics/count": 0,
10.      "subscriptions/count": 0,
11.      "topics/max": 1,
12.      "sessions/persistent/max": 2,
13.      "connections/max": 2,
14.      "subscriptions/shared/count": 0,
15.      "sessions/persistent/count": 0,
16.      "retained/count": 3,
17.      "routes/count": 0,
18.      "sessions/count": 0,
19.      "retained/max": 3,
20.      "sessions/max": 2,
21.      "routes/max": 1,
22.      "subscribers/count": 0,
23.      "connections/count": 0
```

```
24.    }  
25.    ]  
26. }
```

---

更多的监控数据接口可以参考EMQ X的文档。

那么我们可以定期（比如10分钟）调用监控API，获取相应的数据统计，并存储结果，比如放入之前使用的时序数据库。这样便于查询历史状态数据和状态数据可视化。

## 10.8.2 使用系统主题

除了使用调用REST API这种Pull模式，EMQ X还提供了一种Push模式获取运行状态的数据。EMQ X定义了一系列以“\$SYS”开头的系统主题，会周期性地向这些主题发布包含运行数据的MQTT协议消息。

在<https://developer.emqx.io/docs/broker/v3/cn/guide.html?highlight=%E7%B3%BB%E7%BB%9F%E4%B8%BB%E9%A2%98#sys>可以看到所有系统主题의列表。

我们可以运行一个MQTT Client去订阅相应的系统主题，当收到MQTT协议消息后，将数据存入时序数据库就可以了。

以当前连接数为例，EMQ X会把消息发向主题“\$SYS/brokers/:Node/stats/connections/count”，其中Node为EMQ X的节点名。如果想获取所有节点의连接数信息，只需要订阅“\$SYS/brokers+/stats/connections/count”。

系统主题也是支持共享订阅的，以当前连接数为例，我们可以启动多个MQTT Client，订阅“\$queue/\$SYS/brokers+/stats/connections/count”，那么这些

MQTTClient就会依次获得当前订阅数的消息，这样就实现了订阅端负载均衡，并避免了单点故障。

默认设置下，EMQ X会每隔1分钟向这些系统主题发布运行数据，这个时间周期是可以配置的。例如，在开发环境中，可以把这个时间间隔设置得短一点（比如2秒）。

---

```
### < EMQ X 安装目录>/emqx/conf/emqx.conf  
broker.sys_interval = 2s
```

---

但在生产环境中建议设置一个较大的值。

默认情况下，EMQ X只会向和节点运行在同一服务器上的MQTT Client订阅系统主题，我们可以通过修改“<EMQ X安装目录>/emqx/conf/acl.conf”来修改这个默认配置。

---

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

---

我们会以这种方案来实现IoTHub的状态监控。

### 10.8.3 EMQ X的Listener Zone

现在还剩下最后一个问题，IoT Hub的EMQ X Broker启用了Client认证，Client需要提供用户名和密码才能接入，那么这些订阅系统主题，接收Broker运行数据的MQTT Client也需要分配用户名和密码。

在现有的架构里，我们可以用JWT为这些MQTT Client生成一个临时的token作为接入的认证，那有没有办法在保持现有认证体系的前提下，让内部使用的Client安全地绕过认证呢？

EMQ X提供了监听器（Listener）Zone功能，监听器是指用于接收MQTT协议连接的server socket，比如可以配置EMQ X分别在端口1000和2000接受MQTT协议的连接，那么端口1000和2000就是两个监听器。Zone的意思是可以将监听器进行分组，不同的组应用不同的策略，比如通过端口1000接入的Client需要认证，而通过端口2000接入的Client则不需要认证。

EMQ X默认情况下提供两个Zone：External和Internal，分别对应接受外部MQTT协议连接和内部MQTT协议连接。External Zone的MQTTS (MQTT over SSL) 监听器监听的是8883端口，IoT Hub对外开放，供设备连接的就是这个监听器。我们可以在配置文件找到这个监听器的配置。

---

```
### < EMQ X 安装目录>/emqx/conf/emqx.conf  
listener.ssl.external = 8883
```

---

Internal Zone的MQTT（非SSL）监听器监听的是127.0.0.1的11883端口。

---

```
### < EMQ X 安装目录>/emqx/conf/emqx.conf  
listener.tcp.internal = 127.0.0.1:11883
```

---

如上文所说，每个Zone都可以单独设置一些策略，比如Internal Zone在默认情况下是允许匿名接入的。

---

```
### < EMQ X 安装目录>/conf/emqx.conf  
zone.internal.allow_anonymous = true
```

---

所以用于接收系统主题的MQTT Client可以连接到127.0.0.1:11883，这样就不需要用户名和密码了。当然，在实际生产中，你应该用例如防火墙的规则对外屏蔽掉Internal监听器的端口，只向外暴露External监听器的端口。



## 10.8.4 代码演示

这里我们用一段简单代码演示如何通过接受系统主题消息的方式，获取EMQ X当前的连接数，并将数据存入InfluxDB。首先通过共享订阅的方式订阅相应的系统主题。

---

```
1. //IotHub_Server/monitor.js
2. var mqtt = require('mqtt')
3. var client = mqtt.connect('mqtt://127.0.0.1:11883')
4. client.on('connect', function () {
5.
6. client.subscribe("$queue/$SYS/brokers/+/stats/connections
7. /count")
8. })
9. client.on('message', function (topic, message) {
10.   console.log('${topic}: ${message.toString()}')
11. })
```

---

运行这段代码，会得到如下输出。

---

```
$SYS/brokers/emqx@127.0.0.1/stats/connections/count: 1
$SYS/brokers/emqx@127.0.0.1/stats/connections/count: 1
...
```

---

这里我们已经获得了当前的EMQ X的连接数，接下来定义连接数在InfluxDB里的存储。

---

```

1. //IotHub_Service/services/influxdb_service.js
2. const Influx = require('influx')
3. const influx = new Influx.InfluxDB({
4.   host: process.env.INFLUXDB,
5.   database: 'iothub',
6.   schema: [
7.     ...
8.     {
9.       measurement: 'connection_count',
10.      fields:{
11.        count: Influx.FieldType.INTEGER
12.      },
13.      tags:["node_name"]
14.    }
15.  ]
16. })
17. class InfluxDBService {
18.   ...
19.   static writeConnectionCount(nodeName, count) {
20.     influx.writePoints([
21.       {
22.         measurement: 'connection_count',
23.         tags: {node_name: nodeName},
24.         fields: {count: count},
25.         timestamp: Math.floor(Date.now() / 1000)
26.       }
27.     ], {
28.       precision: 's',
29.     }).catch(err => {
30.       console.error('Error saving data to InfluxDB!
31.       ${err.stack}')
32.     })
33.   }

```

---

在存储时我们使用nodeName作为tag，所以需要从主题名里提取出 nodeName。

---

```

1. //IotHub_Server/monitor.js
2. client.on('message', function (topic, message) {

```

```
3.   var result
4.   var countRule =
pathToRegexp("$SYS/brokers/:nodeName/stats/connections/co
unt")
5.   if((result = countRule.exec(topic)) != null){
6.       InfluxDbService.writeConnectionCount(result[1],
parseInt(message.toString()))
7.   }
8. })
```

---

运行“monitor.js”，然后查询InfluxDB，我们会得到如下输出。

---

```
influx
> use iothub
Using database iothub
> select * from connection_count
name: connection_count
time                count node_name
----                -
1559970858000000000 1      emqx@127.0.0.1
1559970860000000000 1      emqx@127.0.0.1
...
```

---

这说明当前连接数的信息已经被成功记录到InfluxDB，如果需要记录更多的运行状态数据，可以按照同样的方法进行拓展。

## 10.9 本章小结

本章利用之前实现的上行数据处理和下行数据处理功能实现了一些更高维度的抽象功能，现在我们完成了Maque IoTHub MQTT相关的全部功能。在第11章中，我们将学习如何编写插件实现扩展EMQ X的功能。

## 第11章 扩展EMQ X Broker

在IotHub中，我们使用了EMQ X自带的WebHook插件，IotHub Server通过使用WebHook插件获取设备的上下线事件和发布的数据。从开发和演示功能的角度看，这个插件是没问题的，但是若要在生产环境中使用，你应该注意以下几个问题。

- WebHook缺乏对身份的校验，EMQ X在Post到指定的WebHook URL时，没有带上任何的身份认证信息，所以IotHub没有办法知道消息是否真的来自EMQ X。

- 性能的损耗，在每次设备上下线和发布数据时，EMQ X都会发起一个HTTP请求：建立连接、发送数据、关闭连接，这部分的开销对EMQ X的性能有可见的影响。这还不是最糟的，对于那些我们不关注的事件，比如设备订阅、设备取消订阅、送达等发生的时候EMQ X依然向WebHook URL发起一个请求，这完全是性能的浪费。

- 顽健性，假设IotHub的Web服务因为某种原因宕机了，在修复好之前，EMQ X获取的上行数据都会丢失。

这也就意味着，IotHub需要一个定制化更强的Hook机制：

- 能够对消息和事件的提供者进行验证；

- 只有在IoTHub感兴趣的事件发生时，才触发Hook机制；
- 用可持久化的队列来解耦消息和事件的提供者（EMQ X）、消费者（IoTHub Server），同时保证在IoTHub Server不可用期间，消息和事件不会丢失。

EMQ X自带了很多插件，不过暂时没有满足上述需求的插件。但是EMQ X的架构提供了很好的可扩展性，我们可以自行编写一个插件实现上述功能。接下来，我们将编写一个基于RabbitMQ的Hook插件。

- 可配置触发Hook的事件。
- 当事件发生时，插件将事件和数据放入RabbitMQ的可持久化的Queue中，保证事件数据不会丢失。

RabbitMQ有一套完整的接入认证和ACL功能，所以我们可以通过使用RabbitMQ的认证体系进行身份验证，保证事件来源的可靠性（只会来自EMQ X）。

## 11.1 EMQ X的插件系统

EMQ X插件机制的逻辑其实很简单，首先EMQ X定义了很多Hook（钩子），如表11-1所示。

表11-1 EMQ X的Hook定义

Hook	说明
client.authenticate	连接认证
client.check_acl	ACL 校验
client.connected	客户端上线
client.disconnected	客户端连接断开
client.subscribe	客户端订阅主题
client.unsubscribe	客户端取消订阅主题
session.created	会话创建
session.resumed	会话恢复
session.subscribed	会话订阅主题后
session.unsubscribed	会话取消订阅主题后
session.terminated	会话终止
message.publish	MQTT 消息发布
message.deliver	MQTT 消息进行投递
message.acked	MQTT 消息回执
message.dropped	MQTT 消息丢弃

开发者可以通过编写插件的方式在这些Hook上注册处理函数，在插件内部可以调用EMQ X中的数据和函数，在这些处理函数里面执行自定义的业务逻辑就可以对EMQ X的功能进行扩展。



我们之前使用的MongoDB认证插件、JWT认证插件、WebHook插件和我们将要开放的RabbitMQ Hook都是使用的这样的机制。

### 11.1.1 Erlang语言

EMQ X是用Erlang编写的，所以我们开发插件也必须使用Erlang。Erlang对很多人来说还比较陌生，对电信行业的从业者来说可能要相对熟悉一些。多年前，我在诺基亚工作的时候曾使用过Erlang，后来在编写EMQTT（EMQ X 3.0之前被称为EMQTT）插件的时候又用了一段时间。Erlang是我使用过的表达力最强的一种语言，强过现在的高级脚本式语言，比如Ruby、Python、Node.js等。但Erlang的学习曲线比较陡，本节我会简单介绍Erlang语言的一些特性，但是仅限于在插件编写中用到的部分。

我不期望读者在短短几节就能学会Erlang语言，但是我仍然强烈推荐大家有空去学习一下Erlang语言，这个语言让我第一次有写程序犹如在写诗的感觉。

如果要学习Erlang，我建议大家看由Erlang之父编写的《Erlang程序设计（第2版）》。

## 11.1.2 安装Erlang编译工具

在之前的章节里，我们已经安装好了Erlang的Runtime，EMQ X 3.0需要Erlang/OTP-R21+进行编译，所以要确保你安装的Erlang Runtime版本是符合要求的，你可以在终端运行“erl”，如果得到下面的输出，那么说明Erlang Runtime是被正确安装的。

---

```
Erlang/OTP 22 [erts-10.4.1] [source] [64-bit] [smp:8:8]
[ds:8:8:10] [async-threads:1] [hipe] [dtrace]

Eshell V10.4.1 (abort with ^G)
1>
```

---

我们将要编写的插件会依赖Erlang的RabbitMQ Client库，这个库有一部分是用Elixir编写的，Elixir是运行在Erlang虚拟机上的一种语言，类似Scala之于Java。所以还需要安装elixir，同时为了保证能够正确的编译EMQ X，请确保你系统上的GNU Make为4.0之后的版本。

读者朋友可以在Elixir官网找到Elixir的安装文档。

编译EMQ X还需要Erlang的编译工具Rebar3，最简单的方式是通过源码编译的方式安装。

---

```
$ git clone https://github.com/erlang/rebar3.git
$ cd rebar3
```

```
$ ./bootstrap
```

---

然后将生成的rebar3 binary加入“\$PATH”就可以了。

本节介绍了EMQ X的插件系统，并准备了开发环境，接下来将介绍Erlang语言的一些特性。

## 11.2 我们会用到的Erlang特性

本节将学习一部分Erlang的语言特性，目的是让读者能够更好理解后文编写的插件代码。

## 11.2.1 Erlang简介

Erlang诞生于1987年，由爱立信的CS-Lab开发。Erlang是一种动态类型的、函数式的语言，主要是为了处理并行、分布式应用而设计，所以Erlang内建了轻量级的进程模型，让编写并发应用变得非常容易。

运行Erlang程序需要有一个类似JVM的虚拟机，11.1节已经安装了这个虚拟机。Erlang提供了一个交互式脚本解析器，我们可以运行`erl`打开这个解析器，并在里面运行Erlang语句。

---

```
Erlang/OTP22 [erts-10.4.1] [source] [64-bit] [smp:8:8]
[ds:8:8:10] [async-threads:1] [hipe] [dtrace]

EshellV10.4.1 (abortwith ^G)
1>1+1.
2
```

---

这里Erlang计算了`1+1`的值，输出结果为`2`。注意，一条Erlang语句的结束符是英文句号“`.`”，和我们写英文文章时的句子结束符是一样的。

## 11.2.2 变量和赋值

Erlang中的“变量”一定是用大写字符开始的，比如大写X就是一个合法的“变量”，而小写x就不是一个合法的“变量”，我们可以用“变量赋值”。

---

```
2>X=10.  
10
```

---

这里“变量赋值”是加了引号的，原因是在Erlang的世界里，其实没有变量和赋值这两个概念。

我们可以尝试改变X的值。

---

```
3>X=1.  
**exceptionerror: nomatchofrighthandsidevalue1
```

---

在X被第一次“赋值”后，我们就无法再改变它的值了，所以说Erlang里面的变量实际上是一次性赋值变量。本书仍然会用变量来指代X，不过你要记住它是不可变的。

在尝试修改X值时，我们得到一个错误提示：“no match of right hand side value 1”，提示的意思是等式两边的值不匹配，而

不是你不能改变X的值，这是为什么呢？

因为“=”在Erlang中并不是赋值符，Erlang中的“=”非常类似我们在数学课程中使用的“=”，只是用于表达一个等式，这是Erlang和其他语言非常不同的一个地方。如果你在Erlang里面使用“=”号，那代表你想让Erlang计算一个等式或者方程式，Erlang会尽力给出方程的解，如果方程是无解的，就会抛出no match的异常。

1) “X=10.” Erlang计算这个等式的时候，为了让这个等式成立，所以给X绑定了一个值10，作为方程的解。所以，X的值就变成了10。

2) “X=1.” Erlang计算这个等式，发现X为10，等式无法成立，所以抛出异常。

这是一个非常重要的概念，理解了Erlang中“=”的含义后，就能理解Erlang语言中一半的内容了。

由于Erlang的变量实际是不可变的，所以在进行并行编程时，不会出现多个Erlang进程修改同一变量的情况，这就意味着没有竞争，也就没有锁了，这就是用Erlang编写并行程序很简单的原因之一。



## 11.2.3 特殊的Erlang数据类型

### 1. 原子

前面说了小写的x在Erlang中不是一个合法的“变量”，那么x是什么呢？在Erlang中，以小写字母开头的元素被称为原子，比如x、monday、failed都是原子，你可以将原子理解为Ruby里的符号，C和Java里面的枚举类型。只不过在Erlang中不需要预先定义原子就可以使用。

### 2. 元组

元组类似于C语言里的结构体，例如“{20, 50}”就是一个合法的Erlang元组，可以用于表达一个点的坐标。不像C语言的结构体，Erlang元组里面的成员字段是可以没有名字的。在Erlang里，通常会用原子标识元素的含义，增加程序的可读性，比如标识一个三维坐标的元组可以表示为：“P={{x, 100}, {y, 80}, {z, 170}}”。

### 3. 记录

记录是元组的另外一种形式，它可以用一个名字标注元组的各个元素，不过在使用记录前，需要先声明记录。

---

```
-record(record_name, {  
    field1="default",  
    field2=1,  
    field3  
})
```

---

上面的代码声明了一个名为record\_name的记录，它包含3个字段，其中field1和field2具有一个默认值。声明一个记录后，就可以创建对应的记录了。

---

```
#record_name{field1="new",field3=100}
```

---

## 4. 列表

Erlang中的列表类似于JavaScript、Ruby等动态语言中的数组，可以存放任何类型的值，用“[]”表示。

---

```
L=[x,20,{a,3.0}].
```

---

也可以用“|”来拼接列表。

---

```
L1=[100, 200|L].
```

---

## 11.2.4 模式匹配

模式匹配是Erlang中的一个关键概念，就像我在前面说的一样，“=”在Erlang中表达一个等式，其实就是让Erlang去执行一个模式匹配，Erlang会尽力让等式两边的值匹配，以元组“{{x, 100}, {y, 80}, {z, 170}}”为例，我们可以写出如下等式。

---

```
1> {X, {Y, Y}, {Z, Z}}={{x, 100}, {y, 80}, {z, 170}}.
{{x,100},{y,80},{z,170}}
2>X.
{x,100}
3>Y.
80
4>Z.
170
5>
```

---

可以看到，为了让等式“{X, {y, Y}, {z, Z}}={{x, 100}, {y, 80}, {z, 170}}”成立，Erlang将X的值绑定为100，将Y的值绑定为80，将Z的值绑定为170，这里相当于用模式匹配获取了元组各个字段的值。

编写Erlang语言程序，大部分工作是在写模式匹配，能看懂模式匹配，就基本能看懂Erlang代码了。

## 11.2.5 模块与函数

Erlang的源文件是以.erl结尾的，一个.erl文件就是一个模块，在一个模块里可以定义多个函数，还可以用显式的export方式使方法可以被其他模块使用。

---

```
-module(module1).  
-export([func/0]).  
  
func()->  
    ...
```

---

上面的代码声明了一个名为module1的模块，模块里定义了一个名为func()的方法，并将这个方法export了出去，那么在其他地方调用func()方法的方式如下所示。

---

```
module1:func().
```

---

export语句中的“func/0”函数名后面的数字代表的是函数的参数数量，在Erlang里面可以有同名函数，而且同名函数可以拥有一样的参数数量。

---

```
func()->  
    ...  
func(X,80)->
```

```
...  
func(X,Y) ->  
...
```

---

有意思的是，Erlang在决定调用哪个同名函数的时候，是通过对参数列表进行模式匹配来确定的，以上面的例子为例：func(100,200)将调用第三个func()函数，而func(100,80)会调用第二个func()函数。这样就不用像其他语言一样，在func(X,Y)的函数体里面写“if Y==80”这样的分支了。

## 11.2.6 宏定义

在Erlang中可以使用以下方式定义宏。

---

```
-define(MACRO_NAME, XXX)
```

---

宏定义好之后，可以通过以下方式使用宏。

---

```
?MACRO_NAME
```

---

## 11.2.7 OTP

OTP (Open Telecom Platform, 开发电信平台) 是Erlang一个框架和库的集合, 类似于Java的J2EE容器, Erlang程序利用OTP比较快速地开发大规模的分布式系统, EMQ X就是运行在OTP上的Erlang程序。我们在编写插件时, 也会使用到OTP的一些功能。

本节简单介绍了Erlang语言的一些特性, 这仅仅是Erlang的冰山一角, 并不能让你马上学会编写Erlang程序, 目的是便于你阅读和理解接下我们要写的插件代码。

## 11.3 搭建开发和编译环境

本节将开始搭建EMQ X插件开发的环境。



### 11.3.1 下载和编译EMQ X

EMQ X的插件需要调用EMQ X内部的一些函数，所以没有办法单独编译EMQ X插件，而是需要将插件代码放到EMQ X源码中一起编译。所以首先要搭建EMQ X的编译环境。EMQ X的代码需要从EMQ X的Release Project下载，因为IotHub是基于EMQ X 3.2.0开发的，所以我们要编译版本为3.2.0的EMQ X。

---

```
git clone -b v3.2.0 https://github.com/emqx/emqx-rel.git
```

---

先编译一次EMQ X，确认编译环境搭建成功。

---

```
cd emqx-rel  
make
```

---

如果你看到如下输出，则代表编译成功了。

---

```
==> Resolved emqx-3.2.0  
==> Including Erts from /usr/local/lib/erlang  
==> release successfully created!
```

---

运行`emqx-rel/_build/emqx/rel/emqx/bin/emqx console`，这是EMQ X的控制台模式（和“`emqx start`”不同），EMQ X会启动一个交

互式的Erlang控制台方便开发和调试，如果看到如下输出就说明EMQ X 编译成功了。

---

```
EMQ X Broker v3.2.0 is running now!  
Eshell V10.3 (abort with ^G)
```

---

然后关闭emqx console，进行下一步。

## 11.3.2 使用插件模板

EMQ X提供了一个模板插件`emqx-plugin-template`，其可以用作编写插件的参考。这个插件功能是在事件发生时，例如`client.connected`，将事件的内容打印出来。在编写插件前，我们先通过这个插件模板来学习如何编译插件和代码结构。

插件在EMQ X是作为一个编译器的依赖库和EMQ X的核心代码一起进行编译的。EMQ X的Makefile在默认情况下已经把`emqx-plugin-template`作为一个依赖了，我们可以在`emqx-rel/rebar.config`里找到与`emqx-plugin-template`相关的配置项。

---

```
1. {deps,  
2.   [ emqx  
3.     ...  
4.     , emqx_plugin_template  
5.   ].
```

---

但是在默认情况下，这个插件并不会随`emqx`发布，为了在`emqx`里面使用这个插件，我们还需要修改`rebar.config`，在`rebar.config`中加入以下内容。

---

```
1. {relx,  
2.   ...  
3.   [ kernel  
4.     ...
```

---

```
5.      , {emqx_psk_file, load}
6.      , {emqx_plugin_template, load}
7.      ]
```

---

我们在编译插件时，需要在这个`rebar.config`中加入类似的关于插件的配置。

在EMQ X的编译工具里，编译器依赖是由一个git地址指定的，所以要使用一个git仓库来保存插件的代码。

然后运行`make`，等待编译结束。

EMQ X编译完成后可以在`emqx-rel/_build/emqx/lib`下找到编译完成的`emqx-plugin-template`，然后把整个`emqx_plugin_template`目录复制到`<EMQ X安装目录>/emqx/lib`下（注意不是编译得到的EMQ X目录），以`console`模式启动EMQ X：`<EMQ X安装目录>/emqx/bin/emqx console`，然后加载模板插件。

---

```
< EMQ X 安装目录>/emqx/bin/emqx_ctl plugins load
emqx_plugin_template
```

---

接着运行一个MQTT Client：`mosquitto_sub-t"test/pc"`，我们可以在EMQ X console上打印出图11-1所示的信息。

```
Eshell V10.3.2 (abort with ^G)
```

```
(emp@127.0.0.1)1>
```

```
(mysql@127.0.0.1)1>
```

```
(emp@127.0.0.1)1>
```

```
(mosq127.0.0.1)> Client(mosq/ByzH1CH9e9LcLXoKy) authenticate, Password: undefined
```

```
Session(mosq/9yzH1CH9e9LcLXoKcy) created: [{clean_start,true},
{binding,local},
{client_id,
<<"mosq/9yzH1CH9e9LcLXoKcy">>},
{username,undefined},
{expiry_interval,0},
{created_at,{1560,152000,298263}},
{conn_pid,<0.1811.0>},
{next_pkt_id,1},
{max_subscriptions,0},
{subscriptions,#{}},
{upgrade_qos,false},
{inflight,
{emqx_inflight,32,{0,nil}}},
{retry_interval,20000},
{queue_len,0},
{awaiting_rel,#{}},
{max_awaiting_rel,100},
{wait_rel_timeout,300000}]
```

```
Client(mosq/9yzH1CH9e9LCLXoKcy) connected, connack: 0, conn_attrs:#{clean_start =>
true,
client_id =>
<<"mosq/9yzH1CH9e9LCLXoKcy">>,
conn_mod =>
enqx_connection,
connected_at =>
{1560,
152000,
297931},
credentials =>
#{auth_result =>
success,
client_id =>
<<"mosq/9yzH1CH9e9LCLXoKcy">>,
mountpoint =>
undefined,
peername =>
{{127,
```

图11-1 EMQ X console打印出的信息

这不是出错信息，而是Client触发的，如connected、subscribed等事件的打印。

这样我们就可以编译和使用emqx-plugin-template了，接下来我们看一下emqx-plugin-template的代码结构。

### 11.3.3 插件的代码结构

在成功地进行一次编译以后，`emqx-rel`项目会把依赖的代码都检出到`emqx-rel/deps`目录下，我们可以在`emqx-rel/deps/emqx_plugin_template`目录中找到模板插件的所有代码，如图11-2所示。

图11-2中的序号1、2、3分别表示：插件的代码、插件的配置文件、插件的编译配置（比如依赖等）。

插件的配置文件、插件的编译配置我们会在后面讲到。我们先看一下插件的代码结构。

EMQ X的插件和EMQ X一样，是一个运行在OTP上面的应用，这个应用的入口是`emqx_plugin_template_app.erl`，如图11-3所示。

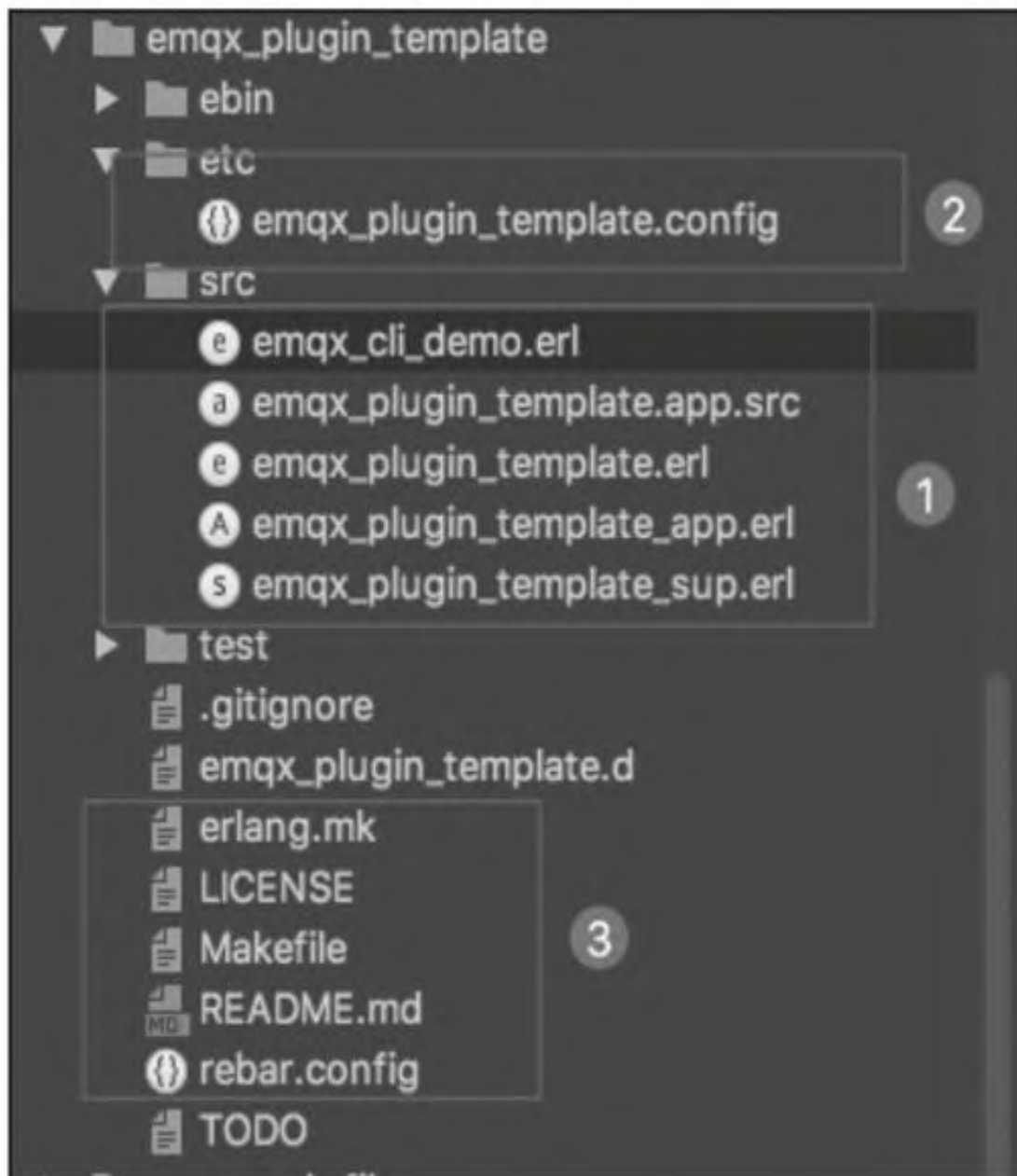


图11-2插件的代码结构



```
%% See the License for the specific language governing permissions and
%% limitations under the License.

-module(emqx_plugin_template_app).

-behaviour(application).

-emqx_plugin(?MODULE).

-export([ start/2
         , stop/1
         ]).

start(_StartType, _StartArgs) ->
  ① {ok, Sup} = emqx_plugin_template_sup:start_link(),
  ② emqx_plugin_template:load(application:get_all_env()),
  {ok, Sup}.

stop(_State) ->
  emqx_plugin_template:unload().
```

图11-3 emqx\_plugin\_template\_app.erl的代码

这个应用主要完成如下两个功能：

- 1) 运行App的监视器（序号1）；
- 2) 加载插件的主要功能代码（序号2）。

插件的主要功能代码在emqx\_plugin\_template.erl里，如图11-4所示。

```

, on_message_publish/2
, on_message_deliver/3
, on_message_acked/3
, on_message_dropped/3
)).

%% Called when the plugin application start
load(Erv) ->
    emqx:hook('client.authenticate', fun ?MODULE:on_client_authenticate/2, [Env]),
    emqx:hook('client.check_acl', fun ?MODULE:on_client_check_acl/5, [Env]),
    emqx:hook('client.connected', fun ?MODULE:on_client_connected/4, [Env]),
    emqx:hook('client.disconnected', fun ?MODULE:on_client_disconnected/3, [Env]),
    emqx:hook('client.subscribe', fun ?MODULE:on_client_subscribe/3, [Env]),
    emqx:hook('client.unsubscribe', fun ?MODULE:on_client_unsubscribe/3, [Env]),
    emqx:hook('session.created', fun ?MODULE:on_session_created/3, [Env]),
    emqx:hook('session.resumed', fun ?MODULE:on_session_resumed/3, [Env]),
    emqx:hook('session.subscribed', fun ?MODULE:on_session_subscribed/4, [Env]),
    emqx:hook('session.unsubscribed', fun ?MODULE:on_session_unsubscribed/4, [Env]),
    emqx:hook('session.terminated', fun ?MODULE:on_session_terminated/3, [Env]),
    emqx:hook('message.publish', fun ?MODULE:on_message_publish/2, [Env]),
    emqx:hook('message.deliver', fun ?MODULE:on_message_deliver/3, [Env]),
    emqx:hook('message.acked', fun ?MODULE:on_message_acked/3, [Env]),
    emqx:hook('message.dropped', fun ?MODULE:on_message_dropped/3, [Env]).

on_client_authenticate(Credentials = #{client_id := ClientId, password := Password},
    io:format("Client(~s) authenticate, Password:-p -n", [ClientId, Password]),
    {stop, Credentials#{auth_result => success}}.

on_client_check_acl(#{client_id := ClientId}, PubSub, Topic, DefaultACLResult, _Env)

```

图11-4 emqx\_plugin\_template.erl的代码

插件的功能很简单：

1) 在插件启动的时候，注册钩子函数，在相应的事件发生时，触发钩子函数（序号1）；

2) 钩子函数的实现，在事件发生时打印事件的内容（序号2）。

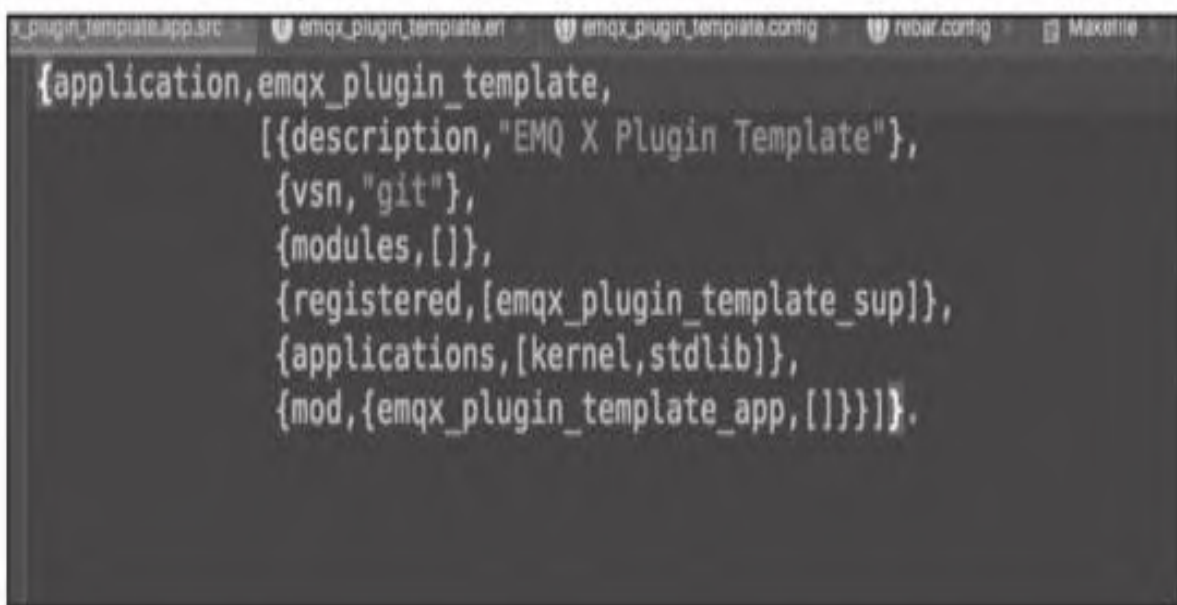
然后是监视器的代码emqx\_plugin\_template\_sup.erl，如图11-5所示。

```
-module(emqx_plugin_template_sup).  
  
-behaviour(supervisor).  
  
-export([start_link/0]).  
  
-export([init/1]).  
  
start_link() ->  
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).  
  
init([]) ->  
    {ok, { {one_for_all, 0, 1}, []} }.
```

图11-5 emqx\_plugin\_template\_sup.erl的代码

这部分代码基本上是固定的，唯一要注意的是划线部分的代码设置了OTP监控App的策略，一个OTP App会存在多个并行的Worker，one\_for\_all代表如果一个Worker因为某种异常崩溃了，就重启所有的Worker，可选的设置还有one\_for\_one，代表只重启崩溃的Worker。

最后是插件App的描述文件emqx\_plugin\_template\_app.erl，如图11-6所示。



```
{application,emqx_plugin_template,
  [{description,"EMQ X Plugin Template"},
   {vsn,"git"},
   {modules,[]},
   {registered,[emqx_plugin_template_sup]},
   {applications,[kernel,stdlib]},
   {mod,{emqx_plugin_template_app,[]}}]}.

```

图11-6 emqx\_plugin\_template\_app.erl的代码

它主要是描述App的名字、加载的模块等。

emqx\_cli\_demo.erl用于添加自定义console命令，暂时用不到，在这里就跳过了。

### 11.3.4 修改模板插件

我们可以尝试修改一下代码，比如让Client连接时，打印用户名和密码。

---

```
1. %% emqx_plugin_template.erl
2. on_client_authenticate(Credentials = #{client_id :=
  ClientId, password :=
  Password}, _Env) ->
3.   io:format("Modified: Client(~s) authenticate,
  Password:~p ~n", [ClientId,
  Password]),
4.   {stop, Credentials#{auth_result => success}}.
```

---

运行make重新编译，用同样的方法将编译出来的插件复制到“<EMQ X安装目录>/emqx/lib”，然后重新启动EMQ X console，用任意的MQTT Client连接到EMQ X，我们会看到以下输出。

---

```
(emqx@127.0.0.1)1> Modified:
Client(mosq/lRZIa6iizQ43roYR8P) authenticate,
Password:undefined
```

---

这说明我们修改的代码已经生效了。

当插件代码用git的方式被引用进来以后，我们就可以在本地进行修改而不用将修改提交到git仓库再进行编译了。

EMQ X的插件不支持热加载，所以修改了插件代码以后需要重启EMQ X。

本节介绍了插件的代码结构，以及如何编译并运行插件，接下来开始正式编写基于RabbitMQ的Hook插件：`emqx_rabbitmq_hook`。

## 11.4 实现基于RabbitMQ的Hook插件：emqx-rabbitmq-hook

和前面说的一样，emqx-rabbitmq-hook插件会在一些事件发生时，比如设备连接、发布消息时，将事件的数据发送到RabbitMQ指定的exchange中。



### 11.4.1 代码结构

在开发的时候我们可以直接在“emqx-rel/deps”创建一个目录“emqx\_rabbitmq\_hook”来存放emqx-rabbitmq-hook插件的代码，如图11-7所示。

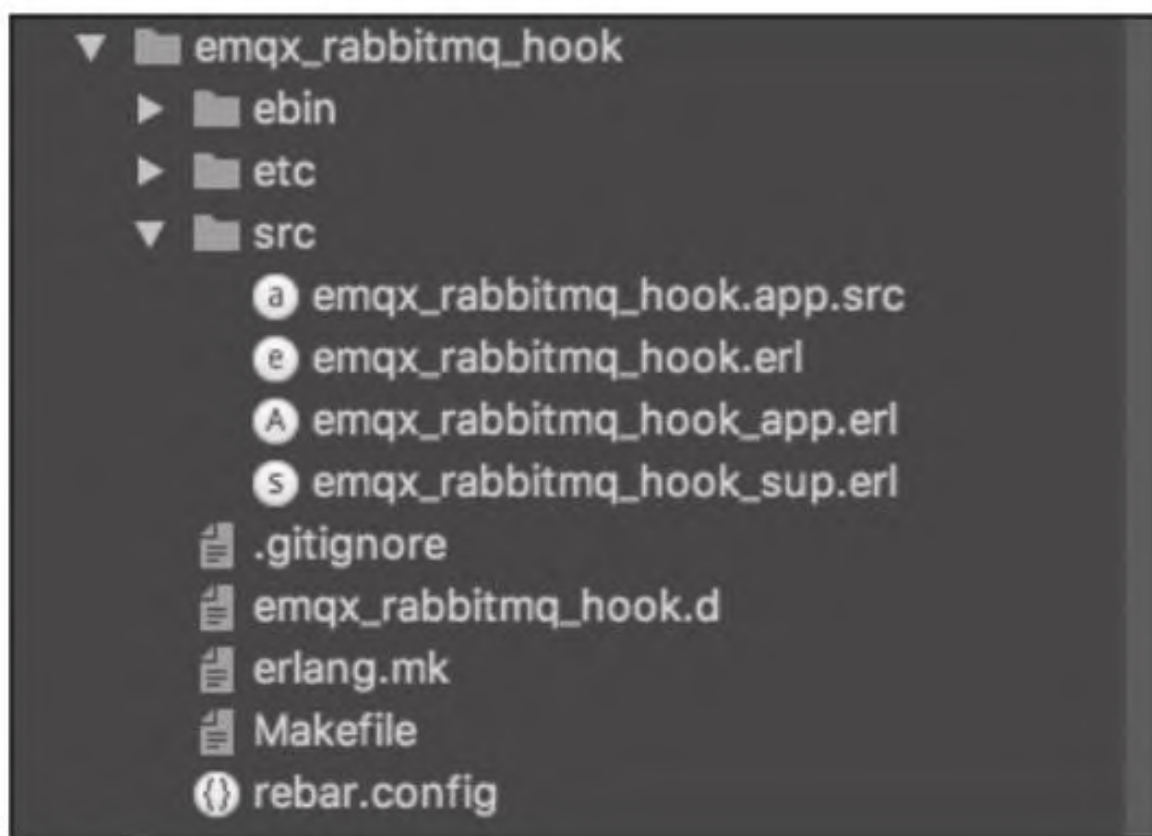


图11-7 emqx\_rabbitmq\_hook的代码结构

初始的代码结构基本和“emqx-plugin-template”一致。

## 11.4.2 建立RabbitMQ连接和连接池

我们需要在插件启动的时候建立和RabbitMQ的连接，同时我们希望用一个连接池对插件的RabbitMQ连接进行管理，首先在插件的rebar.config文件中添加相应的依赖。

---

```
1. ### emqx_rabbitmq_hook/rebar.config
2.
3. {deps, [
4.   {amqp_client, "3.7.15"},
5.   {ecpool, "0.3.0"} ,
6.   ...
7. ]}.
8. {erl_opts, [debug_info]}.
```

---

amqp\_client是Erlang的RabbitMQ Client包，ecpool是一个Erlang的连接池实现。

接着在监控器启动的时候，初始化连接池。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook_sup.erl
2. init([]) ->
3.   application:set_env(amqp_client, prefer_ipv6,
4.     false),
5.   PoolSpec = ecpool:pool_spec(?APP, ?APP,
6.     emqx_rabbitmq_hook_cli, [{pool_
7.       size, 10}, {host, "127.0.0.1"}, {port, 5672}]),
8.   {ok, [{one_for_one, 10, 100}, [PoolSpec]]}.
```

---

这里我们先把连接池大小、RabbitMQ Server的地址端口等配置先写到代码里，后面再把这些配置项放到配置文件里。

连接池需要在emqx\_rabbitmq\_hook\_cli模块中实现RabbitMQ的连接功能。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook_cli.erl
2. connect(Opts) ->
3.     ConnOpts = #amqp_params_network{
4.         host = proplists:get_value(host, Opts),
5.         port = proplists:get_value(port, Opts)
6.     },
7.     {ok, C} = amqp_connection:start(ConnOpts),
8.     {ok, C}.
```

---

### 11.4.3 处理client.connected事件

这里我们做一个约定，默认情况下emqx-rabbitmq-hook插件会把事件数据发送到名为mqtt.events的exchange中，exchange的类型为direct，事件的数据将用BSON进行编码。首先引入对BSON的依赖。

---

```
1. ### emqx_rabbitmq_hook/rebar.config
2.
3. {deps, [
4.     ....
5.     {bson_erlang, "0.3.0"}
6. ]}.
7. {erl_opts, [debug_info]}
```

---

然后在插件启动的时候进行初始化。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. load(Env) ->
3.     emqx_rabbitmq_hook_cli:ensure_exchange(),
4.     emqx:hook('client.connected', fun ?
MODULE:on_client_connected/4, [Env]),
5.     ...
```

---

代码的第3行用来确保“mqtt\_events”已在RabbitMQ中创建。

代码的第4行用于调用内部函数emqx:hook注册处理client.connected事件的钩子函数。

emqx\_rabbitmq\_hook\_cli:ensure\_exchange方法的实现如下。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook_cli.erl
2. ensure_exchange() ->
3.   ensure_exchange(<<"mqtt.events">>).
4.
5. ensure_exchange(ExchangeName) ->
6.   ecpool:with_client(?APP, fun(C) ->
ensure_exchange(ExchangeName, C) end).
7.
8. ensure_exchange(ExchangeName, Conn) ->
9.   {ok, Channel} = amqp_connection:open_channel(Conn),
10.   Declare = #'exchange.declare'{exchange =
ExchangeName, durable = true},
11.   #'exchange.declare_ok'{} =
amqp_channel:call(Channel, Declare),
12.   amqp_channel:close(Channel).
```

---

代码的第6行从连接池里取出一个连接，然后调用第三个  
ensure\_exchange，在RabbitMQ中声明对应的exchange。

处理client.connected事件的钩子函数实现如下。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. on_client_connected("#{client_id := ClientId, username
:= Username}, ConnAck,
ConnInfo, _Env) ->
3.   {IpAddr, _Port} = maps:get(peername, ConnInfo),
4.   Doc = {
5.     client_id, ClientId,
6.     username, Username,
7.     Keepalive, maps:get(Keepalive, ConnInfo),
8.     ipaddress, iolist_to_binary(ntoa(IpAddr)),
9.     proto_ver, maps:get(proto_ver, ConnInfo),
10.    connected_at,
emqx_time:now_ms(maps:get(connected_at, ConnInfo)),
11.    conn_ack, ConnAck
```

```
12.    },
13.
emqx_rabbitmq_hook_cli:publish(bson_binary:put_document(D
oc), <<"client.
connected">>),
14.    ok.
```

---

在事件发生时，向exchange中发布一条routing\_key为client.connected的消息。注意，这里使用emqx\_time:now\_ms获取了以毫秒为单位的消息到达时间，比使用WebHook获取的ts更加精确。

<<"client.connected">>代表用一个字符串生成的二进制数据。

emqx\_rabbitmq\_hook\_cli:publish的实现如下，仍然是从连接池中取一个连接进行操作。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. publish(Payload, RoutingKey) ->
3.    publish(<<"mqtt.events">>, Payload, RoutingKey).
4.
5. publish(ExchangeName, Payload, RoutingKey) ->
6.    ecpool:with_client(?APP, fun(C) ->
publish(ExchangeName, Payload,
RoutingKey, C) end).
7.
8. publish(ExchangeName, Payload, RoutingKey, Conn) ->
9.    {ok, Channel} = amqp_connection:open_channel(Conn),
10.    Publish = #'basic.publish'{exchange = ExchangeName,
routing_key = RoutingKey},
11.    Props = #'P_basic'{delivery_mode = 2},
12.    Msg = #amqp_msg{props = Props, payload = Payload},
13.    amqp_channel:cast(Channel, Publish, Msg),
14.    amqp_channel:close(Channel).
```

---

## 11.4.4 处理client.disconnected事件

这个事件的处理和client.connected事件的处理方法类似，不过需要过滤掉client因为用户名和密码没有通过认证的client.disconnected。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. on_client_disconnected({}, auth_failure, _Env) ->
3.   ok;
4.
5. on_client_disconnected({client_id := ClientId,
username := Username},
ReasonCode, _Env) ->
6.   Reason = if
7.     is_atom(ReasonCode) ->
8.       ReasonCode;
9.     true ->
10.      unknown
11.   end,
12.   Doc = {
13.     client_id, ClientId,
14.     username, Username,
15.     disconnected_at, emqx_time:now_ms(),
16.     reason, Reason
17.   },
18.   emqx_rabbitmq_hook_cli:publish(bson_binary:put_document(Doc), <<"client.
disconnected">>),
19.   ok.
```

---

我们通过参数的模式匹配，没通过认证的client.disconnected事件会落入第一个on\_client\_disconnected函数中，不作任何处理。

## 11.4.5 处理message.publish事件

在处理这个事件时，需要过滤掉来自系统主题的Publish事件。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. on_message_publish(Message = #message{topic =
<<"$SYS/", _/binary>>}, _Env) ->
3.   {ok, Message};
4.
5. on_message_publish(Message = #message{topic = Topic,
flags = #{retain :=
Retain}}, _Env) ->
6.   Username = case maps:find(username,
Message#message.headers) of
7.     {ok, Value} -> Value;
8.     _ -> undefined
9.   end,
10.  Doc = {
11.    client_id, Message#message.from,
12.    username, Username,
13.    topic, Topic,
14.    qos, Message#message.qos,
15.    retained, Retain,
16.    payload, {bin, bin, Message#message.payload},
17.    published_at,
emqx_time:now_ms(Message#message.timestamp)
18.  },
19.
emqx_rabbitmq_hook_cli:publish(bson_binary:put_document(D
oc),
<<"message.publish">>),
20.  {ok, Message}.
```

---

同样，这里使用参数的模式匹配，来自系统主题的“message.publish”事件会落入第一个on\_message\_publish函数中，



不作任何处理。

这里使用`emqx_time:now_ms`函数获取到消息发布的时间（以毫秒为单位），这样可以解决之前NTP服务中以秒为单位而导致的计时不够精确的问题。

## 11.4.6 编译插件

在11.4.4节中，我们已经学会了如何编译插件。不过有一点需要注意，如果新增了一个插件，那么这个插件就只能和一同编译出来的emqx binaries一起发布使用，不能只是把插件的binary复制到已经安装好的emqx的plugins目录下，这样插件是无法使用的。

但是修改一个已发布的插件代码，编译以后就无须再发布一次emqx binaries了，只需要将插件的binary复制过来就可以。

首先需要把我们已经编写完成的代码提交到一个git仓库，这里我把代码提交到了[https://github.com/sufish/emqx\\_rabbitmq\\_hook](https://github.com/sufish/emqx_rabbitmq_hook)的分支“rebar3”上，然后在emqx-rel的rebar.config上添加对emqx\_rabbitmq\_hook的依赖。

---

```
1. {deps,  
2.   [  
3.     ...  
4.     , {emqx_rabbitmq_hook, {git,  
   "https://github.com/sufish/emqx_rabbitmq_  
   hook", {branch, "rebar3"}}}  
5.   ]}.  
6.  
7. relx,  
8.   [  
9.     , {release, {emqx, git_describe},  
10.      [ ...
```

```
11.      , {emqx_rabbitmq_hook, load}
12.      ]}
```

---

运行make，进行编译。

编译完成后，可发布的emqx binaries和插件会被放到“emqx-rel/\_build/emqx/rel/emqx/”目录下，里面包含了完整的EMQ X文件和目录结构。运行这个目录下的EMQ X就可以测试刚编写的插件了。

运行emqx：“emqx-rel/\_build/emqx/rel/emqx/bin/emqx console”；加载emqx-rabbitmq-hook：“emqx-rel/\_build/emqx/rel/emqx/bin/emqx\_ctl plugins load emqx\_rabbitmq\_hook”。不出意外，可以得到如下输出。

---

```
Start apps: [emqx_rabbitmq_hook]
Plugin emqx_rabbitmq_hook loaded successfully.
```

---

我们可以写一段RabbitMQ Client代码测试一下emqx\_rabbitmq\_hook插件。

---

```
1. require('dotenv').config()
2. const bson = require('bson')
3. var amqp = require('amqplib/callback_api');
4. var exchange = "mqtt.events"
5. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
6.   if (error0) {
7.     console.log(error0);
8.   } else {
```

```
9.      connection.createChannel(function (error1,
channel) {
10.          if (error1) {
11.              console.log(error1)
12.          } else {
13.              var queue = "iothub_client_connected";
14.              channel.assertQueue(queue, {
15.                  durable: true
16.              })
17.              channel.bindQueue(queue, exchange,
"client.connected")
18.              channel.consume(queue, function (msg) {
19.                  var data = bson.deserialize(msg.content)
20.                  console.log('received:
${JSON.stringify(data)}')
21.                  channel.ack(msg)
22.              })
23.          }
24.      });
25.  }
26. });
```

---

运行这段代码，接着使用任意的MQTT Client连接到Broker，比如：mosquitto\_sub-t “test/pc，我们会得到以下输出。

---

```
received:
{"client_id":"mosq/Rmkn7f4VZyUbeduN1t","username":null,"Keepalive":60,"ipaddress":"127.0.0.1","proto_ver":4,"connected_at":1560250142384,"conn_ack":0}
```

---

## 11.4.7 插件的配置文件

### 1. config配置文件

插件的配置文件是放在emqx\_rabbitmq\_hook/etc/下的，默认情况下是一个Erlang风格的.config文件，这种配置文件实际上就是Erlang的源文件，内容是一个Erlang的列表，如下所示。

---

```
1. %% emqx_rabbitmq_hook/etc/emqx_rabbitmq_hook.config
2. [
3.   {emqx_rabbitmq_hook, [{enabled, true}]}
4. ].
```

---

EMQ X在启动时会加载这个列表，在插件里，我们可以通过下面的方式读取到这个列表里元素的值。

---

```
(emqx@127.0.0.1)1>
application:get_env(emqx_rabbitmq_hook, enabled).
{ok,true}
```

---

这种风格的配置文件对Erlang用户来说没什么问题，但是对非Erlang的用户来说，可读性还是稍差了一点。EMQ X 3.0后来提供了非Erlang格式的.conf配置文件，在第7章到10章中已经学过。

---

```
xxx.yy.zz = vvv
```

---

这种配置文件需要配置一个映射规则，在EMQ X启动时通过cuttlefish转换成上面的Erlang列表。接下来我们看看如何操作。

cuttlefish是一个可以将Erlang风格的.config配置文件转换为非Erlang风格的.conf配置文件的Erlang库。

## 2. conf配置文件

如果我们要使用上面所说的这种非Erlang风格的.conf配置文件对插件进行配置，那么需要定义一个映射规则，把xxx.yy.zz=vvv这样的键值对映射到Erlang风格的.config配置文件中的Erlang列表中。以是否监听client.connected事件的配置为例，首先新增配置文件。

---

```
1. ### emqx_rabbitmq_hook/etc/emqx_rabbitmq_hook.conf
2. hook.rabbitmq.client.connected = on
```

---

然后增加对应的映射规则。

---

```
1. %% emqx_rabbitmq_hook/priv/emqx_rabbitmq_hook.schema
2. {mapping, "hook.rabbitmq.client.connected",
  "emqx_rabbitmq_hook.client_
  connected", [
3.   {default, on},
4.   {datatype, flag}
5. ]}.

```

---

映射规则文件其实也是一个Erlang源文件，上面的代码将配置项hook.rabbitmq.client.connected映射成了下面的Erlang列表，并指定它的默认值和类型。

---

```
1. [  
2.   {emqx_rabbitmq_hook, [{client_connected, on}]}  
3. ].
```

---

最后需要在emqx-rabbitmq-hook的Makefile里面指明用cuttlefish对配置文件和映射文件进行处理。

---

```
1. emqx_rabbitmq_hook/Makefile  
2. app.config::  
3.   ./deps/cuttlefish/cuttlefish -l info -e etc/ -c  
   etc/emqx_rabbitmq_hook.  
   conf -i priv/emqx_rabbitmq_hook.schema -d data
```

---

重新编译后，运行emqx-rel/\_build/emqx/rel/emqx/bin/emqx console，在控制台中输入application:get\_env(emqx\_rabbitmq\_hook,client\_connected).，就可以获取这个配置项的值了。

---

```
emqx@127.0.0.1)1> application:get_env(emqx_rabbitmq_hook,  
client_connected).  
{ok,true}
```

---

### 3. 更复杂的映射

在映射某些配置项时，我们还需要写一点Erlang代码，比如配置发布事件的exchange名时，RabbitMQ Erlang Client接受的exchange参数是二进制串，比如<<mqtt.events>>，而从.conf配置文件只能读取到字符串值，所以需要再做一个转化。

---

```
1. %% emqx_rabbitmq_hook/priv/emqx_rabbitmq_hook.schema
2. {mapping, "hook.rabbitmq.exchange",
  "emqx_rabbitmq_hook.exchange", [
3.   {default, "mqtt.events"},
4.   {datatype, string}
5. ]}.
6. {translation, "emqx_rabbitmq_hook.exchange", fun(Conf)
->
7.
  list_to_binary(cuttlefish:conf_get("hook.rabbitmq.exchang
  e", Conf))
8. end}.
```

---

代码的第6行调用了list\_to\_binary函数将字符串转换为二进制串。

连接池ecpool的初始化方法接受的是一个配置项列表，所以需要  
将配置文件中的key-value对转换成一个列表。

---

```
1. %% emqx_rabbitmq_hook/priv/emqx_rabbitmq_hook.schema
2. {translation, "emqx_rabbitmq_hook.server", fun(Conf)
->
3.   Pool = cuttlefish:conf_get("hook.rabbitmq.pool",
  Conf),
4.   Host = cuttlefish:conf_get("hook.rabbitmq.host",
  Conf),
5.   Port = cuttlefish:conf_get("hook.rabbitmq.port",
  Conf),
```

---



```
6.    [{pool_size, Pool},
7.      {host, Host},
8.      {port, Port}
9.    ]
10. end}.
```

---

修改完配置映射文件后，需要重新进行编译。我们可以按照上面的规则继续添加更多的配置项。最终的配置文件 `emqx_rabbitmq_hook.conf` 内容如下。

---

```
1. ##rabbitmq server相关配置
2. hook.rabbitmq.host = 127.0.0.1
3. hook.rabbitmq.port = 5672
4. hook.rabbitmq.username = guest
5. hook.rabbitmq.password = guest
6.
7. ## rabbitmq重连时间间隔
8. hook.rabbitmq.reconnect = 3
9.
10. ##连接池大小
11. hook.rabbitmq.pool = 10
12.
13. ## 是否为client.connected事件注册hook
14. hook.rabbitmq.client.connected = on
15.
16. ## 是否为client.disconnected事件注册hook
17. hook.rabbitmq.client.disconnected = on
18.
19. ## 是否为message.publish事件注册hook
20. hook.rabbitmq.message.publish = on
21.
22. ## 发布消息的exchange名字
23. hook.rabbitmq.exchange = mqtt.events
```

---

## 11.4.8 应用配置项

配置文件准备完毕后，我们就可以在代码里读取这些配置项，然后根据配置项的值进行相应的操作。

### 1. 初始化连接池

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook_sub.erl
2. init([]) ->
3.   {ok, PoolOpts} = application:get_env(?APP, server),
4.   PoolSpec = ecpool:pool_spec(?APP, ?APP,
5.   emqx_rabbitmq_hook_cli, PoolOpts),
6.   {ok, {{one_for_one, 10, 100}, [PoolSpec]}}.
```

---

### 2. 设置exchange

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook_cli.erl
2. ensure_exchange() ->
3.   {ok, ExchangeName} = application:get_env(?APP,
4.   exchange),
5.   ensure_exchange(ExchangeName).
6.
7. publish(Payload, RoutingKey) ->
8.   {ok, ExchangeName} = application:get_env(?APP,
9.   exchange),
10.  publish(ExchangeName, Payload, RoutingKey).
```

---

### 3. 根据配置注册hook函数

先实现一个根据配置进行hook注册的方法。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. hookup(Event, ConfigName, Func, InitArgs) ->
3.   case application:get_env(?APP, ConfigName) of
4.     {ok, true} -> emqx:hook(Event, Func, InitArgs);
5.     _ -> ok
6.   end.
```

---

然后在插件加载时调用这个方法。

---

```
1. %% emqx_rabbitmq_hook/src/emqx_rabbitmq_hook.erl
2. load(Env) ->
3.   ...
4.   hookup('client.connected', client_connected, fun ?
MODULE:on_client_
  connected/4, [Env]),
5.   hookup('client.disconnected', client_disconnected,
fun ?MODULE:on_client_
  disconnected/3, [Env]),
6.   hookup('message.publish', message_publish, fun ?
MODULE:on_message_
  publish/2, [Env]).
```

---

重新编译后，加载插件，修改几个配置项以后再重新加载插件，观察配置项是否生效。

本节实现了emqx\_rabbitmq\_hook的全部功能，你可以在[https://github.com/sufish/emqx\\_rabbitmq\\_hook.git](https://github.com/sufish/emqx_rabbitmq_hook.git)找到插件的全部代码。

11.5节将介绍把插件emqx\_rabbitmq\_hook集成到IoTHub中。

## 11.5 使用emqx-rabbitmq-hook

本节将用emqx-rabbitmq-hook插件替换IoTHub之前使用的WebHook插件。

## 11.5.1 发布emqx-rabbitmq-hook插件

虽然我的电脑已经有了可运行的emqx-rabbitmq-hook插件，但是为了在别的系统和机器上使用这个插件，必须要先发布这个插件。

EMQ X插件的代码需要用一个git仓库存放，我们已经把代码放到了[https://github.com/sufish/emqx\\_rabbitmq\\_hook](https://github.com/sufish/emqx_rabbitmq_hook)的rebar3分支，然后编译EMQ X和emqx-rabbitmq-hook插件。

---

```
git clone -b v3.2.0 https://github.com/emqx/emqx-rel.git
```

---

编辑emqx-rel项目的rebar.config文件，加入emqx-rabbitmq-hook。

---

```
1. {deps,  
2.   [  
3.     ...  
4.     , {emqx_rabbitmq_hook, {git,  
5.       "https://github.com/sufish/emqx_rabbitmq_  
6.       hook", {branch, "rebar3"}}}  
7.   ]}.  
8.   relx,  
9.   [  
10.    , {release, {emqx, git_describe},  
11.    [ ...  
12.    , {emqx_rabbitmq_hook, load}  
13.    ]}  
14. ]}
```

---

然后运行make。

在第11.4节中，我们讲过新增的EMQ X插件不能单独发布，需要和编译生产的EMQ X binaries一起发布。目录emqx-rel/\_build/emqx/rel/emqx/包含了完整的EMQ X文件和目录结构，以及emqx-rabbitmq-hook插件，你可以将这个目录打包，解压缩到任意你想安装EMQ X的位置，这样插件就算发布成功了。

当然，之前我们安装的EMQ X就不能用了，关闭旧的EMQ X，然后把旧的EMQ X的配置文件emqx.conf、etc/plugins/emqx\_auth\_mongo.conf、etc/plugins/emqx\_auth\_jwt.conf、etc/plugins/emqx\_web\_hook.conf复制到新的EMQ X安装目录的对应位置。然后在新的EMQ X安装目录中运行bin/emqx start。这样，新的包含emqx-rabbitmq-hook插件EMQ X Broker就可以运行了。

由于使用了新的EMQ X，因此我们需要重新申请EMQ X App账号，运行用于调用EMQ X的Rest API。

---

```
bin/emqx_ctl mgmt insert iotHub MaqueIotHub
```

---

将得到的secret放入“IotHub\_Server/.env”文件中。

最后加载IotHub需要使用的插件。

---

```
bin/emqx_ctl plugins load emqx_auth_mongo  
bin/emqx_ctl plugins load emqx_auth_jwt  
bin/emqx_ctl plugins load emqx_rabbitmq_hook
```

---



## 11.5.2 集成emqx-rabbitmq-hook

IoT Hub Server现在需要从RabbitMQ对应的exchange中获取事件并  
进行处理，我们启动一个RabbitMQ Client读取消息并进行相应的处  
理。

---

```
1. //IoT Hub_Server/event_handler.js
2. ...
3. var addHandler = function (channel, queue, event,
handlerFunc) {
4.   var exchange = "mqtt.events"
5.   channel.assertQueue(queue, {
6.     durable: true
7.   })
8.   channel.bindQueue(queue, exchange, event)
9.   channel.consume(queue, function (msg) {
10.     handlerFunc(bson.deserialize(msg.content))
11.     channel.ack(msg)
12.   })
13. }
14. amqp.connect(process.env.RABBITMQ_URL, function
(error0, connection) {
15.   if (error0) {
16.     console.log(error0);
17.   } else {
18.     connection.createChannel(function (error1,
channel) {
19.       if (error1) {
20.         console.log(error1)
21.       } else {
22.         addHandler(channel,
"iothub_client_connected", "client.connected",
function (event) {
23.           event.connected_at =
Math.floor(event.connected_at / 1000)
24.           Device.addConnection(event)
```

```
25.         })
26.         addHandler(channel,
"iothub_client_disconnected", "client.disconnected",
    function (event) {
27.             Device.removeConnection(event)
28.         })
29.         addHandler(channel, "iothub_message_publish",
"message.publish",
    function (event) {
30.             messageService.dispatchMessage({
31.                 topic: event.topic,
32.                 payload: event.payload.buffer,
33.                 ts: Math.floor(event.published_at /
1000)
34.             })
35.         })
36.     }
37. });
38. }
39. });
```

---

这里为了兼容WebHook，将事件中以微秒为单位的时间转换成了秒，这样做是为了方便同样的代码在WebHook和emqx\_rabbitmq\_hook之间切换。如果需要更高精度的时间，读者可以自行修改代码。

运行“node event\_handler.js”，再运行第7～11章中的测试代码，你会发现IoT Hub在更换Hook插件后仍然可以正常工作。

现在必须保持event\_handler.js运行，IoT Hub才能正常工作。

### 11.5.3 IotHub的全新架构

在使用RabbitMQ Hook后，处理MQTT Broker事件的功能就从运行Server API的Web服务中剥离出去了，现在IotHub由各个相对独立的模块组成，如图11-8所示。

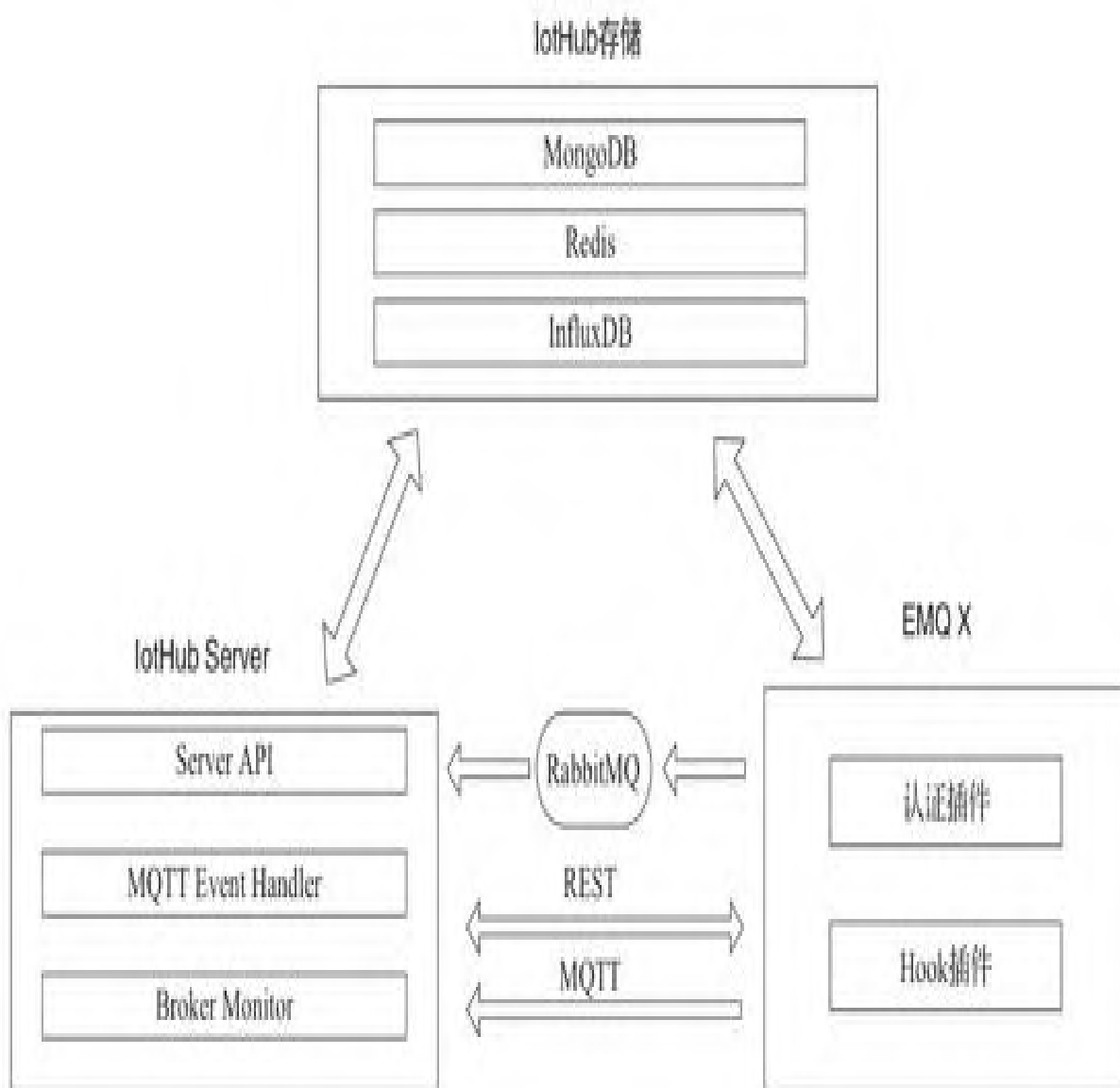


图11-8 IotHub的架构

- Server API: 对外提供IotHub服务的RESTful API服务, 通过运行 `bin/www` 启动。

- MQTT Event Handler: IotHub的核心模块, 处理上行和下行数据的逻辑, 通过运行 `node event_handler.js` 启动。

- Broker Monitor: 监控MQTT Broker运行状态的模块, 通过运行 `node monitor.js` 启动。

为了方便启动这些服务, 我们可以使用Foreman管理这些服务, 首先安装foreman。

---

```
npm install -g foreman
```

---

然后添加Procfile。

---

```
1. api: ./bin/www
2. event_handler: node event_handler.js
3. monitor: node monitor.js
```

---

最后就可以通过Foreman启动所有的服务了。

---

```
cd IotHub_Server
nf start
```

---

细心的读者可能发现了，在目前的代码实现里，当IoTHub向设备发布消息时，也会触发message.publish事件，并被Hook插件发布到RabbitMQ里。如果你不希望这样，可以自行扩展RabbitMQ Hook插件，设置一些主题规则（可以是主题名、正则表达式或者通配符主题名），在message.publish事件匹配到设置的主题规则时，跳过后续的处理。

当然，不做这个优化也不会影响现有功能。

## 11.6 本章小结

本章我们实现了一个基于Rabbitmq的EMQ X Hook插件：  
emqx\_rabbitmq\_hook，并把这个插件集成到了IotHub中，有效解决了原来使用WebHook的一些问题。至此，IotHub的大部分功能和架构就都完成了。接下来，我们将学习另外一种物联网协议——CoAP，并在IotHub中使用它。

## 第12章 集成CoAP协议

本章我们会介绍另外一种物联网协议——CoAP协议，并将这个协议的接入整合到Iothub中。

在物联网开发者社区里，我看到过这样的问题：我的设备运行MQTT协议时，资源很紧张，发布消息很慢；我的传感器只是发布数据，也要跑MQTT协议吗？

最初，MQTT协议被设计用于在计算资源和网络资源有限的环境下运行，它在大多数的环境下运行都很稳定，不过MQTT协议需要建立一个TCP长连接，并需要在固定的时间间隔发送心跳包，对于一些使用电池供电的小型设备而言，能耗还是比较明显的。对于一些采集设备的终端而言，其大部分时间是往上发布数据，建立一个可以用于反控的TCP连接，可能也有些多余。

如果你的应用场景和上面描述的类似，那么还可以选择CoAP协议（Constrained Application Protocol），协议的详细内容可以查看CoAP协议的RFC文档，本章就不逐条解读协议规范了。

我们可以用一句话来概括CoAP协议：CoAP协议是一个建立在UDP协议之上的弱化版的HTTP协议。这就很好理解了，如果设备只需要上传

数据，那么完全可以调用服务器的HTTP接口。如果运行HTTP协议对你的设备来说功耗太大，那么使用CoAP协议就可以解决这个问题。



## 12.1 CoAP协议简介

与MQTT协议的Client-Broker-Client模型不同，CoAP协议和Web都是C/S架构的，设备是Client，接收设备、发送数据的就是Server。所以CoAP协议也被称为“The Web of Things Protocol（物联网协议）”。

## 12.1.1 CoAP协议的消息模型

一条CoAP协议消息由以下三部分组成。

- 二进制头 (Header)
- 消息选项 (Options)
- 负载 (Payload)

CoAP协议的消息设计得非常紧凑，消息的最小长度为4个字节，每个消息都有一个唯一的ID，便于消息的追踪和去重。

CoAP协议的消息可分为两种：一种是需要被确认的消息CON (Confirmable message)，一种是不需要确认的消息NON (Non-confirmable message)。

### 1. CON

CON是一种可靠消息，当接受方收到CON消息时，需要回复发送方，如果发送方没有收到接受方的回复，则不停地重新发送消息。

当Server正常处理完CON消息时，应该向Client回复ACK，ACK中包含了与CON消息一样的ID，如图12-1所示。

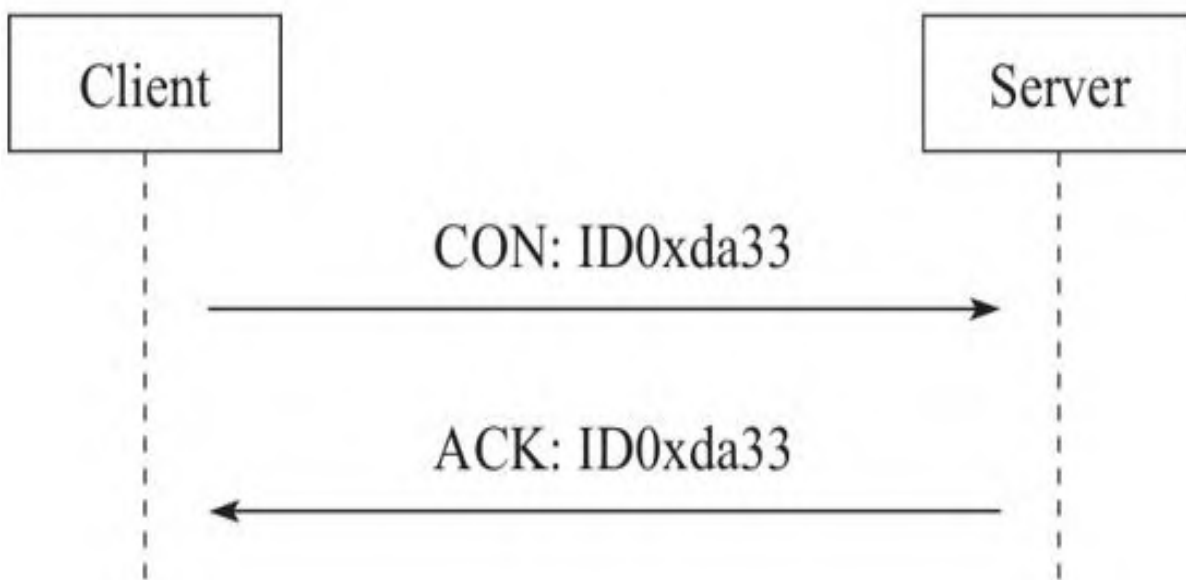


图12-1 对CON消息回复ACK

如果Server无法处理Client的CON消息，Server可以向Client回复RST消息，Client收到RST消息之后，将不再等待ACK，如图12-2所示。

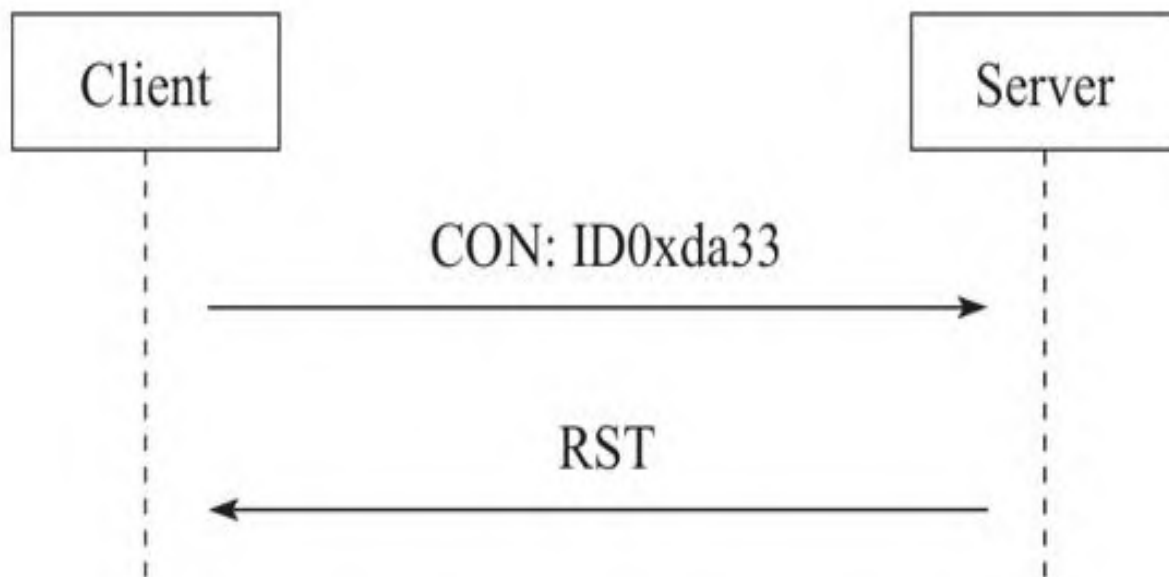


图12-2 对CON消息回复RST

## 2. NON

NON是一种不可靠消息，这种消息不需要接收方的回复，如图12-3所示。



图12-3 NON消息不需要回复

通常可以用这种消息传输类似于传感器读数之类的数据。

### 12.1.2 CoAP协议请求/应答机制

CoAP协议的请求/应答是建立在前面讲到的CON和NON的基础上的，CoAP协议的请求和HTTP协议的请求非常相似，其包含GET、POST、PUT、DELETE 4种方法和请求的URL。图12-4展示了一个典型的CoAP协议的请求/应答流程。

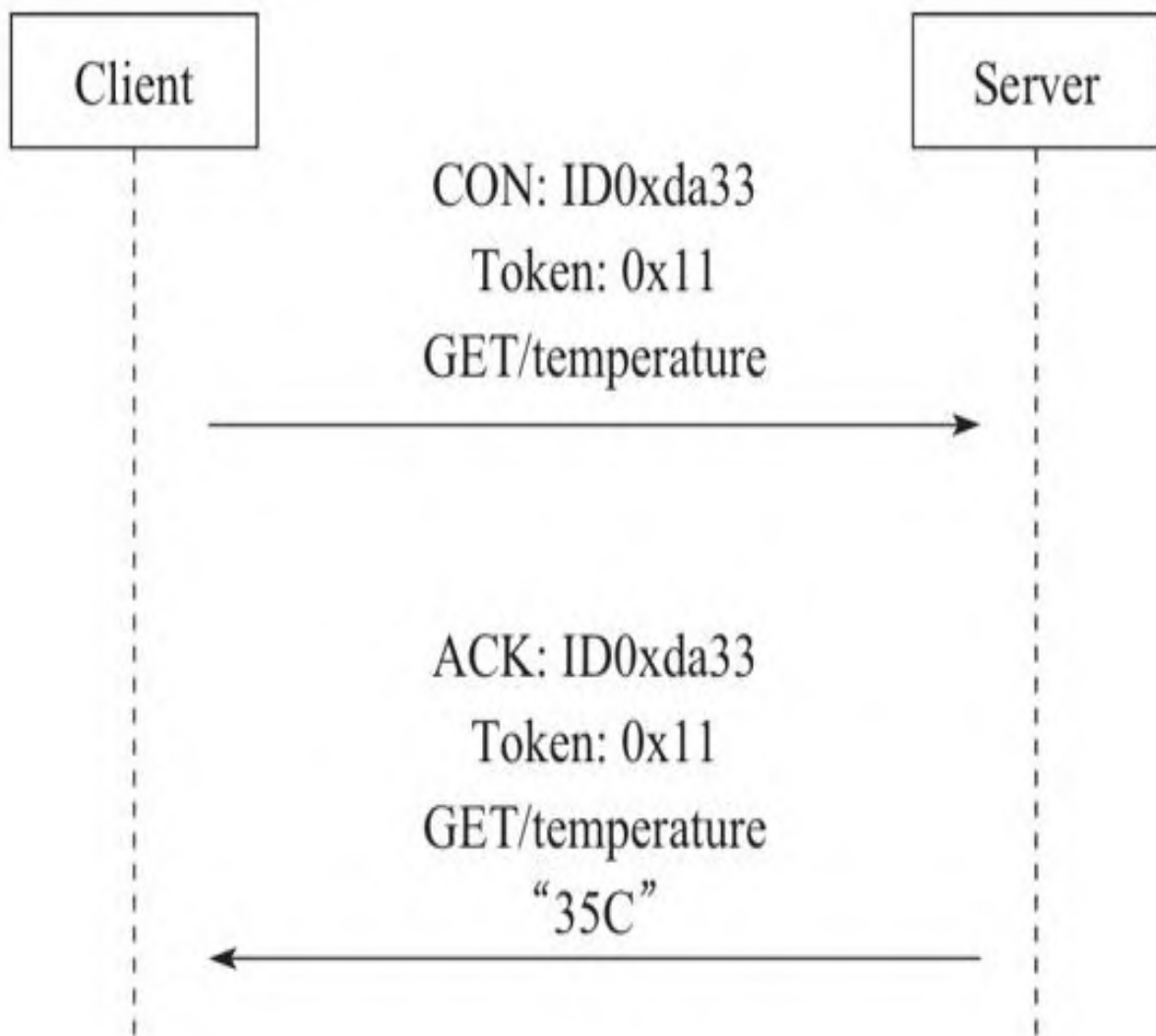


图12-4 典型的CoAP协议的请求/应答流程

请求也可以用NON消息发送，如果请求是用NON发送的，那么Server端也会用NON来回复Client，如图12-5所示。

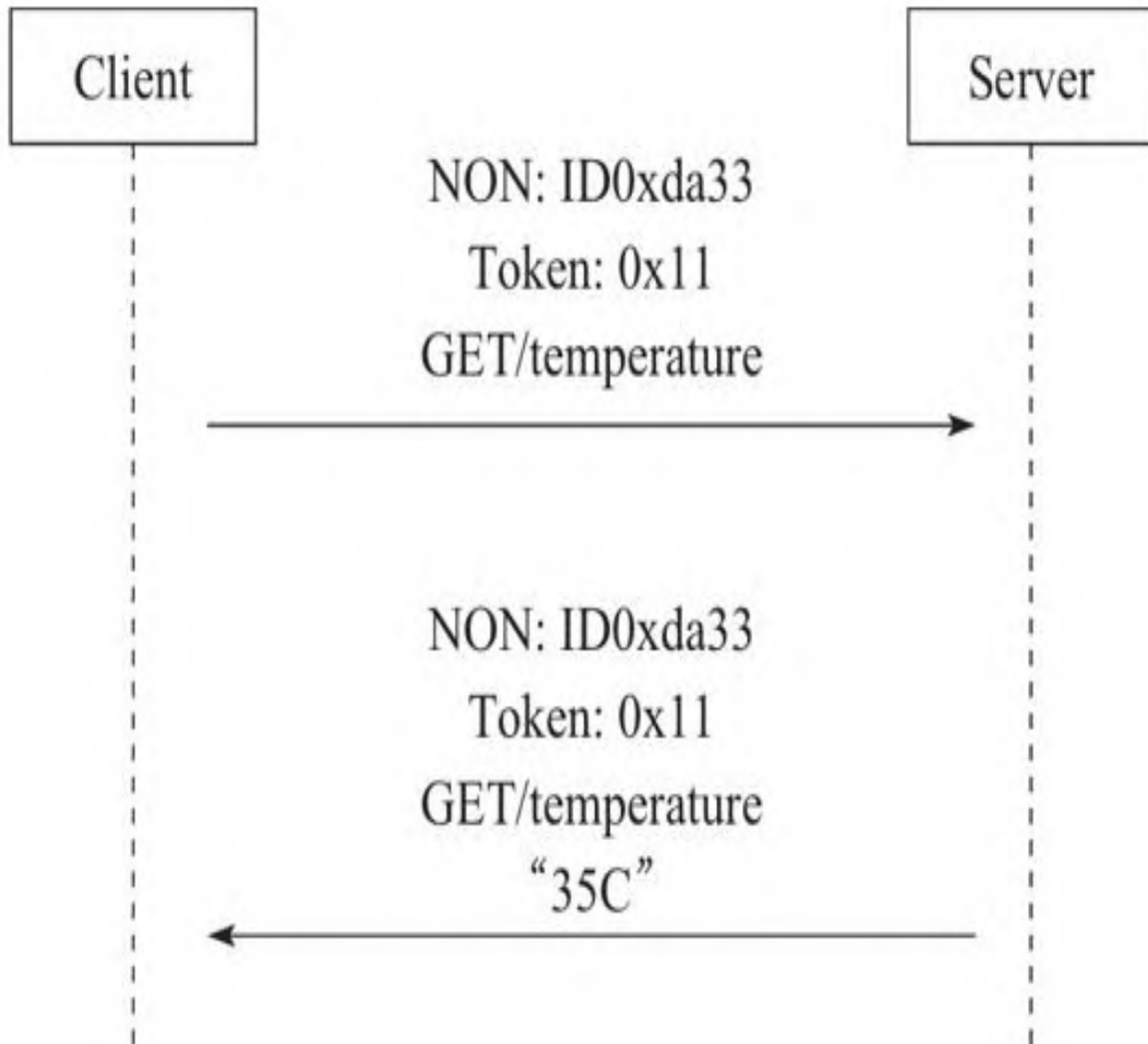


图12-5 使用NON消息的请求/应答流程

大家可能已经注意到，请求和回复中都带了一个Token字段，Token的作用是匹配请求和应答。

如果Server没有办法在收到Client请求时立刻给Client应答，它可以先回复一个空的ACK消息，当Server准备好对Client进行应答时，再向Client发送一个包含同样Token字段的CON消息，这类似于一种异步应答机制，如图12-6所示。

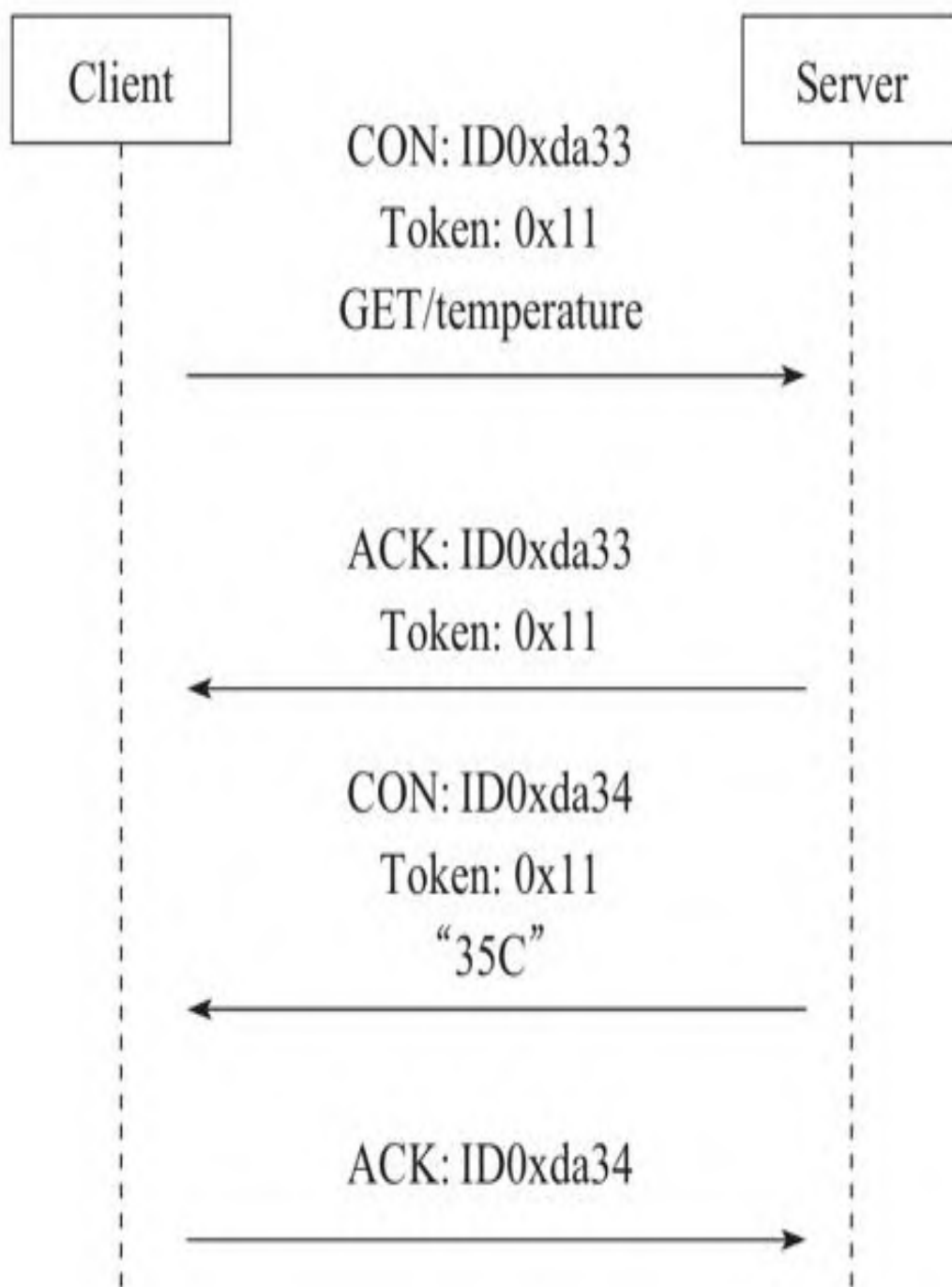


图12-6 “异步”的CoAP协议请求/应答流程



### 12.1.3 CoAP OBSERVE

OBSERVE是CoAP协议的一个扩展，Client可以向Server请求“观察”一个资源，当这个资源状态发生变化时，Server将通知Client该资源的当前状态。利用这个机制，我们可以实现类似于MQTT协议的Subscribe机制，对设备进行反控。

Client可以指定“观察”时长，当超过这个时长后，Server将不再就资源的状态变化向Client发送通知。

在NAT转换后，尤其是在3G/4G的网络下，从Server到Client的UDP传输是很不可靠的，所以我不建议单纯用CoAP协议做数据收集之外的事情，比如设备控制等。

### 12.1.4 CoAP HTTP Gateway

由于CoAP协议和HTTP协议有很大的相似性，因此通常可以用一个Gateway将CoAP协议转换成HTTP协议，便于接入已有的基于Web的系统，如图12-7所示。

Client通过CON命令发送的请求被Gateway转换成了HTTPGET请求并发给Web Server，Web Server的HTTP Response被Gateway转换成CoAP协议的ACK并发送给Client。

本节我们学习了CoAP协议的内容和特性，可以看到，CoAP协议是一个轻量的、类似于HTTP协议的协议，在第12.2节中，我们将会把CoAP协议集成到IoTHub中。

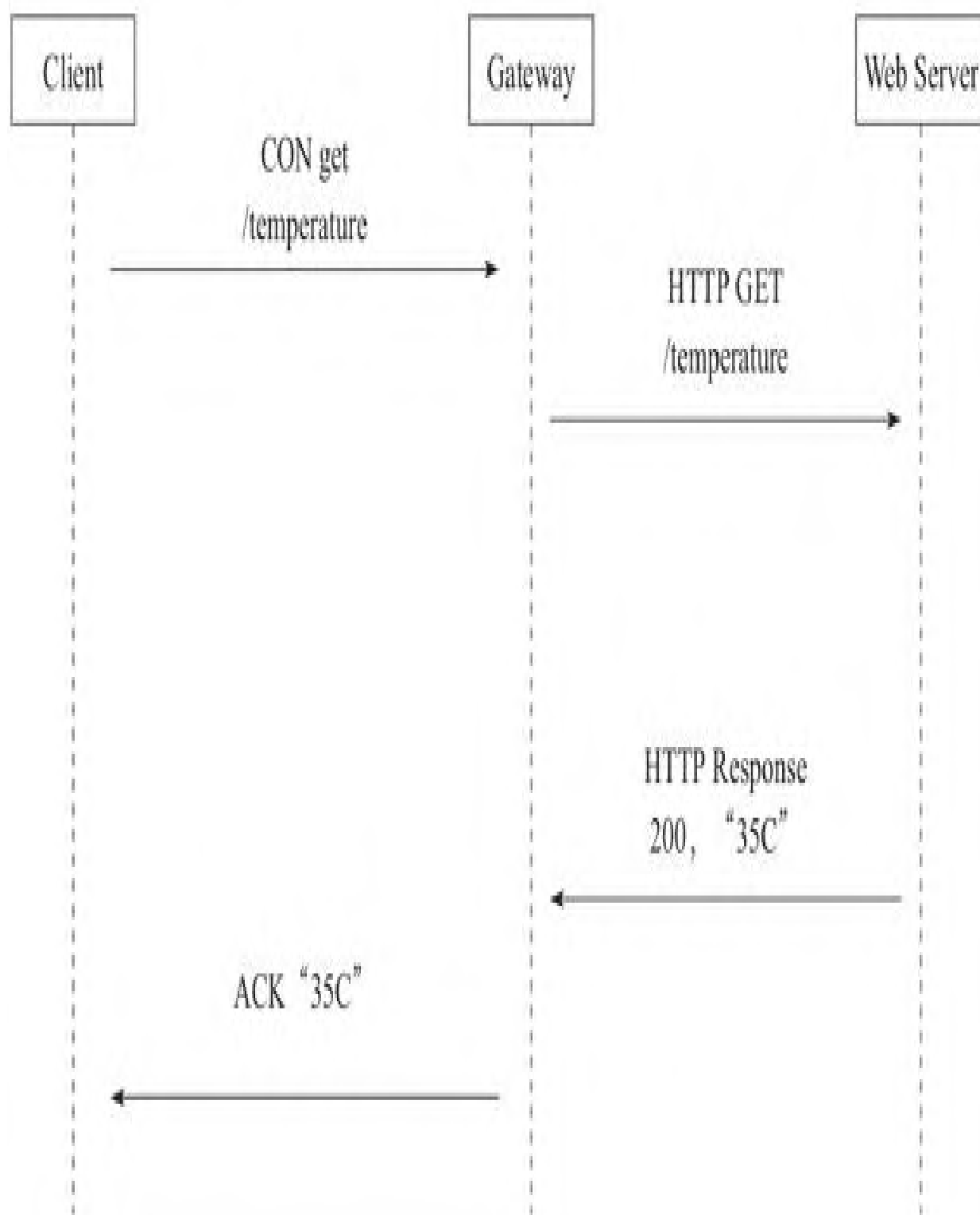


图12-7 CoAP HTTP Gateway

## 12.2 集成CoAP协议

本节中，我们会将CoAP协议接入IotHub，IotHub的CoAP包含以下功能：

- 允许设备用CoAP协议接入，并上传数据和状态；
- DeviceSDK仍然需要向设备应用屏蔽底层的协议细节；
- CoAP设备使用与MQTT设备相同的认证和权限系统。

由于我只建议用CoAP协议实现数据上传功能，因此这里只实现了上行数据的功能。

### 12.2.1 EMQ X的CoAP插件

EMQ X提供了emqx\_coap插件，可用于CoAP协议的接入，这个插件其实是一个CoAP Gateway，与12.1节中提到的CoAP HTTP Gateway类似，不过它会把CoAP请求按照一定规则转换成MQTT的Publish/Subscribe，如图12-8所示。

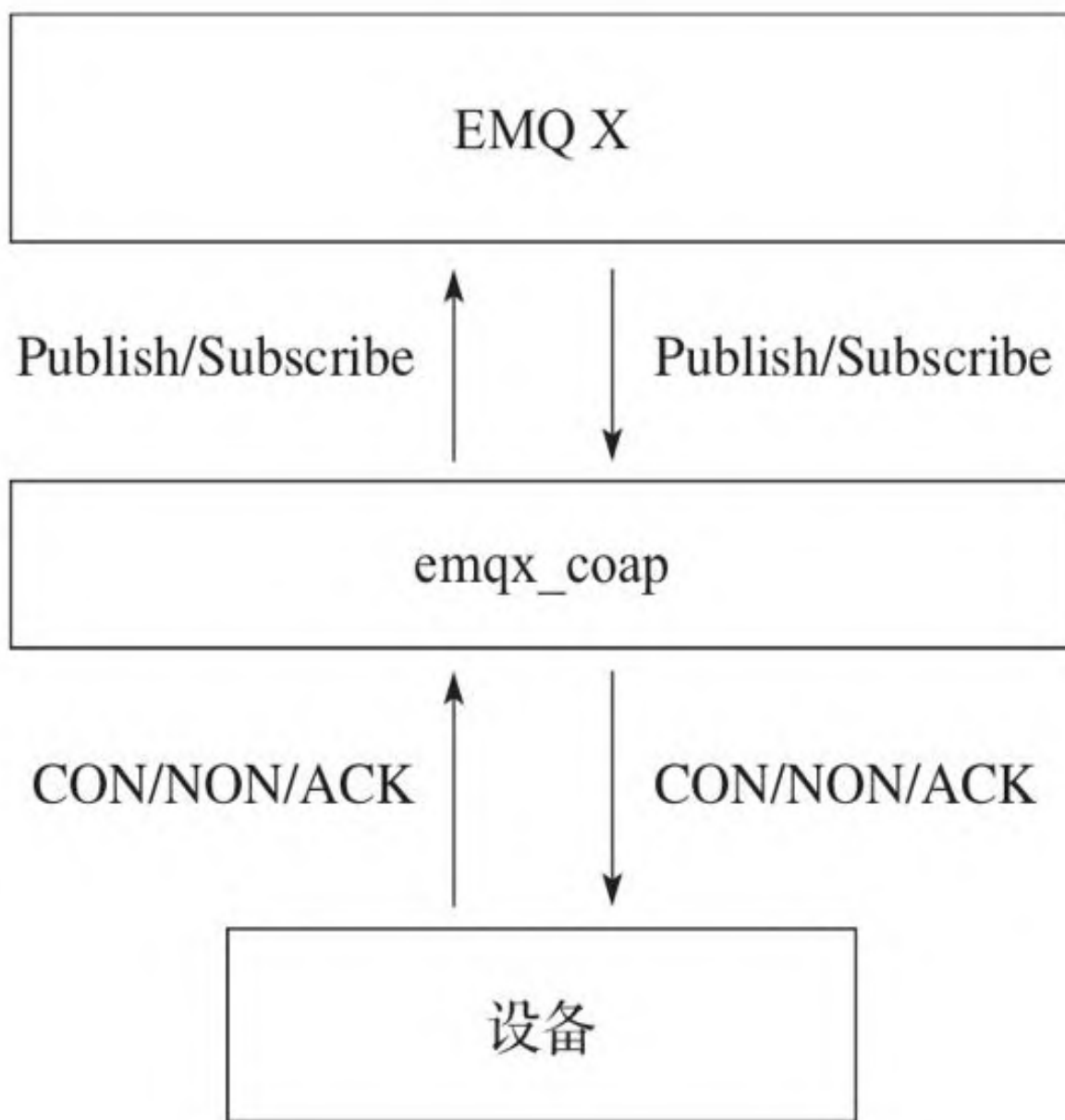


图12-8 EMQ X的CoAP插件功能

CoAP请求被转换成MQTT Publish的示例如下所示。

方法：PUT

URL: coap://<host>:<port>/mqtt/<topic>?c=<client\_id>&u=<username>&p=<password>

例如，CoAP请求PUT “coap://127.0.0.1:5683/mqtt/topic/test?c=c1&u=u1&p=p1” 会被转换成主题为 “topic/test” 的MQTT Publish消息，使用的username/password为u1/p1，ClientID为c1。

CoAP请求被转换成Subscribe请求的示例如下所示。

方法: GET

URL: coap://<host>:<port>/mqtt/<topic>?c=<client\_id>&u=<username>&p=<password>

例如，CoAP请求 “GET coap://127.0.0.1:5683/mqtt/topic/test?c=c1&u=u1&p=p1” 会被转换成主题为 “topic/test” 的MQTT Subscribe消息，使用的username/password为u1/p1，ClientID为c1。

在本书中，我们只关心Publish消息的转换。

我们可以通过运行 “<EMQ X安装目录>/emqx/bin/emqx\_ctl plugins load emqx\_coa” 加载emqx\_coap插件，在默认配置下，插件使用端口5683接收CoAP协议数据。

## 12.2.2 CoAP设备端代码

这里，我们可以使用Node.js的CoAP Client库node-coap，然后在DeviceSDK中新建一个IotCoAPDevice类作为CoAP设备接入的入口。

---

```
1. //IotHub_DeviceSDK/sdk/iot_coap_device.js
2. class IotCoAPDevice {
3.   constructor({serverAddress = "127.0.0.1",
serverPort = 5683, productName,
deviceName, secret, clientID} = {}) {
4.     this.serverAddress = serverAddress
5.     this.serverPort = serverPort
6.     this.productName = productName
7.     this.deviceName = deviceName
8.     this.secret = secret
9.     this.username =
'${this.productName}/${this.deviceName}'
10.    if (clientID != null) {
11.      this.clientIdentifier =
'${this.username}/${clientID}'
12.    } else {
13.      this.clientIdentifier = this.username
14.    }
15.  }
16. }
```

---

我们仍然使用现有设备的ProductName和DeviceName进行接入认证。

上传数据和状态的代码很简单，按照转换规则去构造CoAP请求就可以了。

---



---

```
1. //IotHub_DeviceSDK/sdk/iot_coap_device.js
2. class IotCoAPDevice {
3.     ...
4.     publish(topic, payload) {
5.         var req = coap.request({
6.             hostname: this.serverAddress,
7.             port: this.serverPort,
8.             method: "put",
9.             pathname: 'mqtt/${topic}',
10.            query:
11.            'c=${this.clientIdentifier}&u=${this.username}&p=${this.secret}'
12.        })
13.        req.end(Buffer.from(payload))
14.    }
15.    uploadData(data, type){
16.        var topic =
17.        'upload_data/${this.productName}/${this.deviceName}/${type}/${new ObjectId().toHexString()}'
18.        this.publish(topic, data)
19.    }
20.    updateStatus(status){
21.        var topic =
22.        'update_status/${this.productName}/${this.deviceName}/${new ObjectId().toHexString()}'
23.        this.publish(topic, JSON.stringify(status))
24.    }
```

---

### 12.2.3 代码联调

我们可以写一小段代码来测试IoTHub的CoAP的功能。

---

```
1. //IoTHub_DeviceSDK/samples/upload_coap.js
2. var IotCoapDevice = require("../sdk/iot_coap_device")
3. require('dotenv').config()
4. var path = require('path');
5. var device = new IotCoapDevice({
6.   productName: process.env.PRODUCT_NAME,
7.   deviceName: process.env.DEVICE_NAME,
8.   secret: process.env.SECRET,
9.   clientID: path.basename(__filename, ".js"),
10. })
11. device.updateStatus({coap: true})
12. device.uploadData("this is a sample data",
    "coapSample")
```

---

运行“node upload\_coap.js”，然后检查MongoDB里面对应设备的状态数据和消息数据，你会发现CoAP在正常工作。

## 12.2.4 CoAP协议的连接状态

CoAP协议是基于UDP的，按理来说是没有连接的，但是，如果我们查看对应设备的连接状态，就会发现对应的设备下多了一个已连接的connection，而upload\_coap.js早就执行完毕退出了，这是为什么呢？

---

```
curl http://localhost:3000/devices/IotApp/H9rTa3uSm
...
{"connected":true,"client_id":"IotApp/H9rTa3uSm/upload_coap",
"ipaddress":"127.0.0.1","connected_at":1560432017}
...
```

---

因为经过emqx\_coap的转换之后，EMQ X Broker认为这是一个MQTT Client接入并发布了数据，所以会保留一个MQTT connection，而upload\_coap.js执行完毕退出时，并没有发送DISCONNECT数据包，所以这个MQTT connection的状态是已连接的。

那什么时候这个连接会变成未连接呢？按照MQTT协议的规范，当超过 $1.5 * \text{keep\_alive}$ 的时间间隔没有收到来自该连接的消息时，就会认为该连接已关闭。

我们在“<EMQ X安装目录>/emqx/etc/plugins/emqx\_coap.conf”中可以配置keep\_alive的值，默认为120秒。

---

```
coap.keep_alive = 120s
```

---

等待超过180秒后，再次查询设备的连接状态。

---

```
curl http://localhost:3000/devices/IotApp/H9rTa3uSm
...
{"connected":false,"client_id":"IotApp/H9rTa3uSm/upload_c
oap","ipaddress":"127.
0.0.1","connected_at":1560432017,"disconnect_at":15604321
97}
...
```

---

这个连接的状态已变为未连接。

## 12.3 本章小结

本章我们学习了CoAP协议，并配置了EMQ X Broker使其支持CoAP，然后用现有的IotHub的设备体系支持CoAP设备的接入和数据上传。至此，IotHub的所有功能和代码就完成了。

# 结语 我们学到了什么

如果你耐着性子从开头看到这里，那么祝贺你，比你更熟悉MQTT协议的技术人员应该不多了，你现在应该已经准备好构建一个属于自己的物联网平台了。

在本书里，我们不仅学习了MQTT协议的规范以及各种特性，还从零搭建了一个支持MQTT/CoAP协议的物联网平台。

## 1. MQTT协议的规范和特性

我们详细学习了MQTT协议v3.1.1的规范和全部特性，并结合代码进行了演示，你可以将本书作为MQTT协议的参考，在工作和学习中遇到相关问题时进行查阅。

同时，我们也介绍了MQTT 5.0的相关特性，现在支持MQTT 5.0的Broker也比较多，在合适的条件下，也可以考虑在实际项目中使用MQTT 5.0。

## 2. 从Client-Broker-Client到Client-Server

我们通过抽象，将MQTT协议的Client-Broker-Client模式转换成了Client-Server模式。对设备而言，它通过调用DeviceSDK，不用再关心底层的数据传输细节，只需要向服务器发送数据和处理服务器下发的数据即可。对于业务系统来说，它通过调用IoT Hub提供的API，不需要再建立到Broker的连接，只处理设备上报的数据和下发数据到设备即可。至于数据是用MQTT协议还是CoAP协议以及MQTT Broker在哪里等问题，对Client-Server模式都是透明的。我们主要是通过以下两点完成这个抽象的。

### （1）主题规划

在IoT Hub中，我们定义了一系列主题，我们把主题当作描述消息内容的元数据字段使用。这是很关键的一点，如果我们把消息的描述放入Payload，那么IoT Hub的业务逻辑就和设备应用代码的逻辑耦合到了一起。要记住，在MQTT协议或者任何类似队列的系统里，用Payload判断消息的类型都是anti-pattern。在这样的系统里，用主题名或者队列名进行消息类型的判断，同一种类型的消息应该使用同样的主题名或者队列名。

IoT Hub利用EMQ X的Hook机制，在处理上行数据时，设备Publish的MQTT协议消息中的主题实际上是没有任何真实的MQTT Client订阅的，EMQ X Broker不再通过主题名将消息路由给其他的Client，而是

将消息交给IoTHub Server进行处理，这样就更像一个C/S模式的服务器，而不是Broker了。

在处理下行数据时，主题名除了描述消息内容外，还必需要有路由的功能。

## （2）作为中间件的IoTHub

像IoTHub这样的物联网平台的一个很重要的设计思路是：作为业务系统和设备之间的中间件，IoTHub可通过复用业务逻辑来简化和加快物联网应用的开发；屏蔽业务系统和设备应用代码底层的协议细节，并提供一些常用的基础功能，如OTA升级、指令下发、数据上报、设备认证与管理、设备分组、影子设备、NTP服务等。

这样一来，基于IoTHub开发一套物联网应用，就只需要关注于业务逻辑的实现。如果你的公司要开发新的物联网应用，都可以基于IoTHub，业务系统可以复用IoTHub的Server API，设备端也可以直接复用DeviceSDK代码，如果设备换了硬件平台，只需将DeviceSDK移植到相应的语言上。

## 3. EMQ X的高级功能



本书使用了EMQ X的相关功能，这些功能是MQTT协议中没有指定的，它们简化了IotHub的开发，扩展了IotHub的功能。

灵活的Client认证：EMQ X提供了多种认证机制，本书使用的是MongoDB和JWT，你也可以根据自己的需求更换为其他认证方式，多种认证方式可以组成认证链。如果自带的认证方式无法满足你的需求，你还可以通过编写插件来进行扩展。

建议保留JWT认证方式。

基于插件的Hook功能：通过Hook机制，IotHub Server不再需要通过订阅的方式获取设备的消息，除了自带的Hook插件，我们还学习了如何编写插件，并实现了一个基于RabbitMQ的Hook插件。

管理和监控：通过使用EMQ X提供的API，可以在不建立MQTT协议连接的前提下发布数据，同时也可以对设备的连接进行管理，强制关闭设备连接；通过订阅EMQ X提供的系统主题，可以对Broker的运行状态进行监控。

服务器订阅：通过使用服务器订阅，我们简化了设备端的代码，并且减少了设备端需要发送的Subscribe消息。

当然，使用设备端订阅时，在设备每次上线的时候订阅相应主题也是可行的，不过要注意，这样做有两个缺点：

第一，大多数时候，设备上线时发送的Subscribe数据包都是多余的；

第二，重复订阅主题对Retain消息的处理是有干扰的，每次订阅的时候都会收到主题上的Retain消息，这与Retain消息设计的初衷是相悖的。

如果这两个缺点对你来说没有什么影响，那你可以根据你的需求选择和使用设备端订阅。

## 4. 不只是MQTT

MQTT协议是目前最流行的物联网协议，但它并不是在任何情况下都是最优的。IoTHub除了支持MQTT协议外，还支持CoAP协议。

## 5. 离用于生产环境还有多远

### （1）完成剩下的70%代码

一般来说，在一个软件项目中，只有30%的代码是用来完成业务功能的，而其他70%的代码都在做错误处理。在本书中，因为篇幅有限，我基本跳过了这70%的错误处理代码，把内容集中在了功能设计和实现上。所以，要在生产环境使用这套代码，还需要补上错误处理部分。

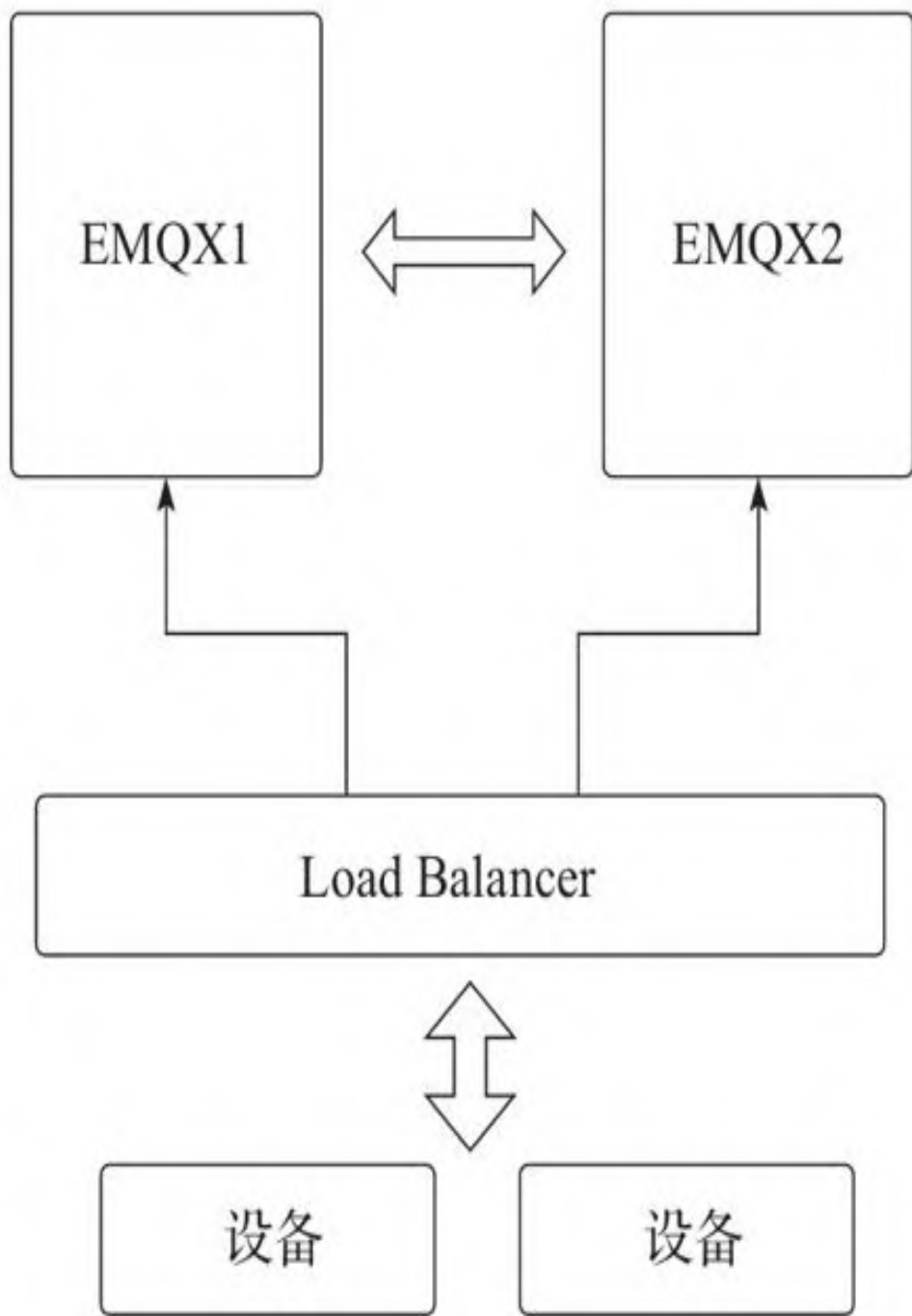
NodeJS并不是我常用和熟悉的语言，我的初衷是选择一门流行的、简单的语言表达IoTHub的设计思路，所以代码实现可能不是最优的。但是，IoTHub的架构和功能的设计思路应该都表达清楚了，你可以根据实际情况，把IoTHub移植到你熟悉的语言上，并对功能进行裁剪或扩展。

DeviceSDK也需要根据你的实际情况移植到对应的语言上，毕竟使用NodeJS的物联网终端还是比较少的。

## （2）横向扩展

如果在生产环境中部署IoTHub，还需要考虑其横向扩展性，IoTHub在设计之初就考虑到了这点，它的每一个组成模块都是可以横向扩展的。

EMQ X可以组成集群，下图是一个双节点EMQ X集群的推荐部署方式。



· Monitor: monitor使用的是共享订阅，所以可以启动多个Monitor进程实现负载均衡。

- Event\_Handler: RabbitMQ是支持多消费者的,可以启动多个Event\_Handler进程实现负载均衡。

- Server API: 可以像横向扩展任意Web服务的方式那样进行扩展。

- Redis: 可以组成Redis Cluster。

- MongoDB: 可以组成MongoDB Replica Set。

- InfluxDB: 可以组成InfluxDB cluster。

- RabbitMQ: 可以组成RabbitMQ Cluster。

## 附录 如何运行Maque IoTHub

### A.1 安装依赖软件

首先，确保你的电脑已安装并运行以下软件。

- MongoDB
- Redis
- InfluxDB
- RabbitMQ

以上软件可以使用默认配置运行。

### A.2 IoTHub的代码

IoTHub的代码都托管在GitHub：IoTHubServer：

[https://github.com/sufish/IotHub\\_Server](https://github.com/sufish/IotHub_Server)、IoTHub DeviceSDK：

[https://github.com/sufish/IotHub\\_Device](https://github.com/sufish/IotHub_Device)、emqx\_rabbitmq\_hook：

[https://github.com/sufish/emqx\\_rabbitmq\\_hook](https://github.com/sufish/emqx_rabbitmq_hook)。

## A.3 安装运行EMQ X

按照第11章中的步骤安装包含emqx\_rabbitmq\_hook插件的EMQ X Broker。

## A.4 配置EMQ X

以下是需要配置的EMQ X配置项：

<EMQ X安装目录>/emqx/etc/emqx.conf:

---

```
allow_anonymous = false
zone.external.mqueue_store_qos0 = false
module.subscription = on
module.subscription.1.topic = cmd/%u/+//+/#
module.subscription.1.qos = 1
module.subscription.2.topic = rpc/%u/+//+/#
module.subscription.2.qos = 1
module.subscription.3.topic = m2m/%u/+//
module.subscription.3.qos = 1
```

---

如果是在开发环境中，建议修改如下两项：

---

```
enable_acl_cache = off
acl_deny_action = disconnect
```

---

第一项是让ACL列表的改动可以马上生效，第二项是能方便地发现ACL权限错误的情况。

<EMQ X安装目录>/emqx/etc/plugins/emqx\\_auth\\_jwt.js:

---

```
auth.jwt.verify_claims = on
auth.jwt.verify_claims.username = %u
```

---

<EMQ X安装目录>/emqx/etc/plugins/emqx\\_auth\\_mongo.js:

---

```
auth.mongo.database = iothub
auth.mongo.auth_query.collection = devices
auth.mongo.auth_query.password_field = secret
auth.mongo.auth_query.password_hash = plain
auth.mongo.auth_query.selector = broker_username=%u,
status=active
auth.mongo.super_query = off
auth.mongo.acl_query.collection = device_acl
auth.mongo.acl_query.selector = broker_username=%u
```

---

<EMQ X安装目录>/emqx/etc/plugins/emqx\\_web\\_hook.js:

---

```
web.hook.api.url = http://127.0.0.1:3000/emqx_web_hook
web.hook.encode_payload = base64
```

---

然后运行 “<EMQ X安装目录>/emqx/bin/emqx start”。

## A.5 加载插件



---

```
<EMQ X 安装目录>/bin/emqx_ctl plugins load emqx_auth_mongo  
<EMQ X 安装目录>/bin/emqx_ctl plugins load emqx_auth_jwt
```

---

选择使用WebHook。

---

```
<EMQ X 安装目录>/bin/emqx_ctl plugins load emqx_web_hook
```

---

选择使用RabbitMQ Hook。

---

```
<EMQ X 安装目录>/bin/emqx_ctl plugins load  
emqx_rabbitmq_hook
```

---

Web Hook和RabbitMQ Hook只能二选一。

## A.6 IotHub Server

---

```
git clone https://github.com/sufish/IotHub_Server  
cd IotHub_Server  
npm install  
cp .env.sample .env
```

---

根据你的环境和配置修改.env。

---

```
###运行IotHub Server需要的配置项  
#API端口  
PORT=3000  
#Mongodb地址
```

```
MONGODB_URL=mongodb://iot:iot@localhost:27017/iothub
#需要与<EMQ X 安装目录>/emqx/etc/emqx_auth_jwt.conf中一致
JWT_SECRET=emqxsecret

#使用EMQ X REST API的账号，可通过bin/emqx_ctl mgmt insert
<APP_ID> Maque IotHub添加
EMQ_X_APP_ID=
EMQ_X_APP_SECRET=

#EMQ X REST API地址
EMQ_X_API_URL=http://127.0.0.1:8080/api/v3/

#Redis
REDIS_URL=redis://127.0.0.1:6379

#RabbitMQ
RABBITMQ_URL=amqp://localhost

#InfluxDB
INFLUXDB=localhost

###运行Sample Code需要的环境变量
TARGET_DEVICE_NAME=#设备名DeviceName
TARGET_PRODUCT_NAME=#设备的产品名ProductName
```

---

最后运行“nf start”。

## A.7 DeviceSDK

---

```
git clone https://github.com/sufish/IotHub_Device
cd IotHub_Device
npm install
cd samples
cp .env.sample .env
```

---

根据你的环境和配置修改.env。

---

### 运行sample code需要的环境变量

#设备的ProductName、DeviceName和Secret调用IotHub Server API  
的对应接口创建

PRODUCT\_NAME=

DEVICE\_NAME=

SECRET=

#需要与<EMQ X 安装目录>/etc/emqx\_auth\_jwt.conf中一致

JWT\_SECRET=emqxsecret

##执行设备间通信示例代码时，对端设备的DeviceName和Secret调用  
IotHub Server API的对应接口创建

DEVICE\_NAME2=

SECRET2=

---

环境变量配置好后，我们运行IotHub\_Server/samples和  
IotHub\_Device/samples里面的示例代码，具体请查看各节内容。