第5章 物联网网关协议基础

5.1 物联网网关

网关作为一种中间设备，能够促进两个使用不同协议的网络段在数据传输过程中完成协议转换，所有数据在路由之前都需要经过网关。网关通常由路由器和调制解调器组成，在网络边缘实现并管理从该网络内部或外部定向经过的所有数据。除了负责协议转换之外，网关还能够对数据进行主动采集和传输、对数据进行解析以及对数据进行过滤和汇聚、存储有关主机网络内部路径的信息以及其他网络的路径。

如图 5-1 所示，在物联网中，网关主要存在于网络层。物联网可以简单地划分为三层结构，即感知层、网络层与应用层。感知层是智能物体和感知网络的集合体，主要由 RFID 芯片、GPS 接收设备、传感器、智能测控设备等感知设备组成，用于采集和捕获外界环境或物品的状态信息；网络层则是数据传输的主要载体，其组成部分有各种私有网络、网络管理系统、有线和无线通信网、互联网和云计算平台等，用于信息的传输和通信。

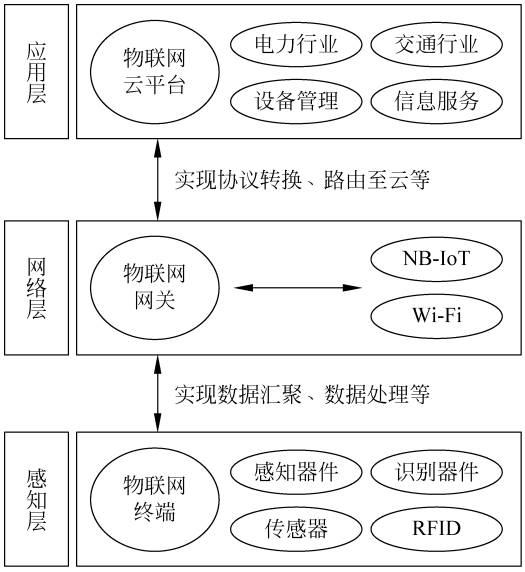

图 5-1 物联网网关的位置

物联网网关聚合来自感知层的数据，并在向高层发送之前处理传感器数据、进行协议转换等。物联网网关将物联网中的各种连接类型转换为标准类型，通过在边缘处预处理数据，可以有效地缩短响应时间。此外，物联网网关还承担着作为物联网设备第一道防线的责任，其通过对互联网和其他外部访问进行分析，有效地保障了物联网数据的安全性。

5.2 HTTP

5.2.1 HTTP 介绍

1. HTTP 基本介绍

HTTP(HyperText Transfer Protocol，超文本传输协议)的发展归功于万维网(World

Wide Web，WWW)协会和 Internet 工程任务组(Internet Engineering Task Force，IETF)的合作。在同一时期，一系列请求评论(Request For Comments，RFC)文件也被发布了。在这一系列文件中，RFC 1945 定义了 HTTP 1.0 版本，RFC 2616 定义了 HTTP 1.1 版本。HTTP 1.1 版本也是今天使用最普遍的一个版本。

HTTP 是用于从 WWW 服务器传输超文本到本地浏览器的传送协议。它的作用是保障超文本文档在被计算机传输时的正确性和高速度。同时，它也有指定传输文档的功能，例如可以指定传输的部分或者指定内容显示的次序，例如，当内容有文本和图形时，就可以指定图形的显示次序比文本高。

2. HTTP 的特点

HTTP 的主要特点有：

(1) 无连接。每次连接仅处理一个请求，完成请求后即断开。该方式可以节省因频繁建立和关闭连接所需要的资源和时间、提升并行处理能力，但随着网页内容愈加丰富可能造成效率较低。需要指出的是，持续连接策略可以适当弥补该不足。

(2) 无状态。每个请求相对独立，不需要额外的资源记忆处理前序事务的历史信息。该方式可以有效避免连接占用，但也可能造成重复传输。需要指出的是，在实际的网页应用中，常结合 Cookies 等措施提升用户体验。

(3) 灵活可靠。灵活是指有多种类型的数据对象可以被 HTTP 传输，由 Content-Type 字段标记了数据对象的具体类型；可靠是指其依赖于可靠的 TCP，较 UDP 来讲可靠性更强。

(4) 兼容性佳。不仅支持客户端/服务器(Client/Server，C/S)架构，也支持浏览器/服务器(Browser/Server，B/S)架构。

3. HTTPS

当信息传递于 Web 浏览器和网站服务器间时，可以应用 HTTP。HTTP 有一定的缺陷，不适合应用于敏感信息的传输。因为它发送内容的方式是以明文的形式，没有对数据进行加密的举措，所以一旦有攻击者截取 Web 浏览器和网站服务器之间的传输报文，那么传输的消息就会被读懂以至于造成信息泄露的危害。

当要传输卡号、密码等信息时，就必须克服此缺陷，所以安全套接字层超文本传输协议(HTTPS)在 HTTP 的基础上加入 SSL。SSL 拥有加密浏览器和服务器间的通信的功能可以根据服务器的证书来实现对服务器身份的验证。

HTTPS 和 HTTP 主要的区别有：

(1) 安全方面。HTTP 的信息传输是明文的方式，而 HTTPS 的信息传输基于 SSL 加密传输协议，因此保证了安全性。

(2) 花销方面。HTTPS 要申请安全证书，此证书一般需要收费。

(3) 端口方面。两者的连接方式和默认端口不同，HTTP 的默认端口号是 80，HTTPS 的默认端口号是 443。

(4) 主要特点。HTTP 的连接是无状态的，所以它的特点是连接过程简单；HTTPS 的特点是安全性高，原因是其由 SSL＋HTTP 所构建，能够实现加密传输和身份认证的网络。

5.2.2 HTTP 的原理

1. 协议架构

如图 5-2 所示，HTTP 承载于 TCP 之上，HTTPS 则是在 TCP 之上增加了 TLS 或 SSL

的协议层。HTTP 和 HTTPS 默认的端口号分别为 80 和 443。

HTTP 是基于 C/S 架构的。作为 HTTP 客户端，浏览器向 HTTP 服务端即 Web 服务器发送请求是需要通过 URL 的。与发布/订阅机制不同，服务器不能向客户端推送消息，除非接收到客户端发送的请求。

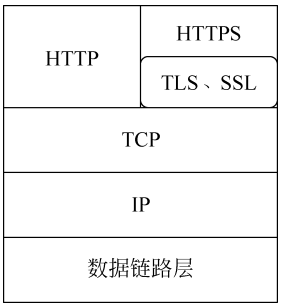

图 5-2　协议架构

2. 工作流程

一个事务的意思是一次 HTTP 操作，要经历的工作过程分为以下几个步骤。

(1) 实现客户机与服务器之间连接的建立；

(2) 完成步骤(1)后，客户机会向服务器发送请求，请求报文的格式为：统一资源定位符(URL)、协议版本号，之后的内容为 MIME 信息，其中有请求修饰符、客户机信息以及其余可能出现的内容；

(3) 当服务器接收到步骤(2)中发送的请求后，就会发出对应的响应信息，响应信息的格式是一个状态行，其中含有信息的协议版本号、一个代码(用于指示成功或错误)，之后的内容为 MIME 信息，例如服务器信息、实体信息以及其余可能出现的内容；

(4) 客户端接收到步骤(3)中的信息后，就会使用浏览器以实现在用户显示屏上的显示，完成以上步骤之后，客户机和服务器之间的连接就会自动断开。

一旦以上步骤中存在错误，那么客户端就会接收到返回的错误信息，输出方式是通过显示器输出。虽然以上步骤较为烦琐，但用户不需要做过多干预，因为上述步骤是由 HTTP 自动进行的，用户要做的事项仅有单击某个超链接，再等待显示屏上的信息显示。

5.2.3　HTTP 请求及响应

1. HTTP 请求

HTTP 的请求消息由请求行、消息报头和请求正文三部分组成，也分别称为请求行、请求头和请求体。

(1) 请求行由请求方法、资源地址(URL)和 HTTP 版本三部分组成，中间使用空格间隔，结尾换行。HTTP 1.1 中共定义了 8 种方法来表明对指定的资源的不同操作方式，所有请求方法区分大小写，应使用大写形式，如表 5-1 所示。

表 5-1　请求方法及功能

方 法 名 称	功　　能
GET	用于请求服务器返回指定资源
PUT	用于请求服务器更新指定资源
POST	用于请求服务器新增资源或执行特殊操作； POST 请求可能会导致新资源的建立或已有资源的修改
DELETE	用于请求服务器删除指定资源，如删除对象等
HEAD	用于请求服务器资源头部
OPTIONS	请求查询服务器的性能或与资源相关的选项和需求
TRACE	回显服务器之前收到的请求，主要作用是测试或诊断
CONNECT	HTTP 1.1 中预留给能够将连接改为管道方式的代理服务器

(2) 请求头由多种标识和对应内容组成，标识与对应内容之间使用冒号隔开，每个标识

内容的末尾需要换行。请求头与请求体之间通常需要空一行以表示请求头的结束。常见的请求头标识如表 5-2 所示，在 HTTP 1.1 中，所有的请求头，除 Host 外，都是可选的。

表 5-2　请求头标识

标　识	含　义
Accept	指定可以被客户端接收的内容类型
Accept-Charset	代表能够被浏览器接受的字符编码集
Accept-Encoding	指定能够被浏览器支持的 Web 服务器返回内容压缩编码类型
Accept-Language	浏览器可接受的语言
Accept-Ranges	可以请求网页实体的一个或者多个子范围字段
Authorization	HTTP 授权的授权证书
Cache-Control	指定请求和响应遵循的缓存机制
Connection	表示是否需要持久连接(HTTP 1.1 默认进行持久连接)
Cookie	HTTP 请求发送时，全部保存在该请求域名下的 Cookie 值都会被共同发送到 Web 服务器
Content-Length	请求的内容长度
Content-Type	请求的与实体对应的 MIME 信息
Date	请求发送的日期和时间
Expect	请求的特定的服务器行为
From	发出请求的用户的 E-mail
Host	指定请求的服务器的域名和端口号
If-Match	只有请求内容与实体相匹配才有效
If-Unmodified-Since	只在实体在指定时间之后未被修改才请求成功
Max-Forwards	限制信息通过代理和网关传送的时间
Pragma	用于包含实现特定的指令
Proxy-Authorization	连接到代理的授权证书
Range	只请求实体的一部分，指定范围
User-Agent	User-Agent 的内容包含发出请求的用户信息
Via	通知中间网关或代理服务器地址、通信协议
Warning	关于消息实体的警告信息

(3) 请求体为用户的主要数据，当请求行的请求方法为 GET 时，请求体为空。

2. HTTP 响应

与请求消息格式相对应，HTTP 响应消息也分为状态行、消息报头、响应正文这三部分，即响应行、响应头和响应体。

(1) 响应行由 HTTP 版本、状态码以及状态码的文本描述这三部分组成，中间使用空格来间隔，结尾换行。状态码的文本描述对于该状态码进行了简短的描述。状态码由 3 个数字组成，第一个数字对响应的类别进行了定义，其有 5 种可能取值，分别代表 5 种响应。常见的状态码及对应含义如表 5-3 所示。

① 1xx：指示信息——表示服务器已接收到请求，需要继续处理。

表 5-3　状态码第一个数字为 1

状态码	英文名称	中 文 描 述
100	Continue	继续。表示客户端应继续其请求，服务器已收到请求的一部分，正在等待其余部分

续表

状态码	英文名称	中 文 描 述
101	Switching Protocols	切换协议。客户端请求服务器切换协议，服务器已确认并准备更换。只能切换到更高级的协议，例如，切换到 HTTP 的新版本协议

② 2xx：成功——表示服务器已成功接收、理解、完成客户端的请求，如表 5-4 所示。

表 5-4　状态码第一个数字为 2

状态码	英文名称	中 文 描 述
200	OK	请求成功。服务器已成功处理了请求，一般用于 GET 与 POST 请求
201	Created	已创建。请求成功并且服务器创建了新的资源
202	Accepted	已接受。服务器已经接受请求，但尚未处理完成
203	Non-Authoritative Information	非授权信息。服务器成功处理了请求，但返回的 meta 信息不在原始的服务器中，而是一个副本
204	No Content	无内容。服务器成功处理了请求，但未返回任何内容。在未更新网页的情况下，可确保浏览器继续显示当前文档
205	Reset Content	重置内容。服务器成功处理了请求，但未返回任何内容，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域
206	Partial Content	部分内容。服务器成功处理了部分 GET 请求

③ 3xx：重定向——要完成请求，客户端必须进行更进一步的操作，如表 5-5 所示。

表 5-5　状态码第一个数字为 3

状态码	英 文 名 称	中 文 描 述
301	Moved Permanently	永久移动。客户端请求的资源已被永久地移动到新 URI，服务器返回信息会包括新的 URI，并且浏览器会自动定向到新 URI。今后客户端发送任何新的请求时都应使用新的 URI
302	Found	临时移动。与 301 类似，但资源只是临时被移动。客户端以后发送请求时应继续使用原有 URI
303	See Other	查看其他地址。与 301 类似，表示客户端应当使用 GET 和 POST 请求查看其他的地址
304	Not Modified	未修改。自从上次客户端请求后，所请求的资源未修改过。服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供的头信息指出客户端希望只返回在指定日期之后修改的资源
305	Use Proxy	使用代理。表示客户端所请求的资源必须通过代理访问
306	Unused	已经被废弃的 HTTP 状态码
307	Temporary Redirect	临时重定向。与 302 类似，客户端应使用 GET 请求重定向

④ 4xx：客户端错误——客户端请求中出现语法错误或者服务器无法实现客户端的请求，如表 5-6 所示。

表 5-6　状态码第一个数字为 4

状态码	英 文 名 称	中 文 描 述
400	Bad Request	客户端请求的语法错误，服务器无法理解
401	Unauthorized	请求要求用户的身份认证，一般出现在需要登录的网页
402	Payment Required	保留，将来使用
403	Forbidden	服务器理解客户端的请求，但是拒绝执行此请求
404	Not Found	服务器找不到客户端所请求的资源（网页）。通过此代码，网站设计人员可对“您所请求的资源无法找到”页面进行个性化设计
405	Method Not Allowed	服务器禁止执行客户端请求中的某些方法
406	Not Acceptable	服务器无法根据客户端请求的内容特性完成请求响应
407	Proxy Authentication Required	请求要求代理的身份认证，与 401 类似，但客户端应当使用代理进行授权
408	Request Time-out	服务器等待客户端发送的请求时间过长，超时
409	Conflict	服务器在处理请求时发生了冲突，在处理客户端的 PUT 请求时，服务器可能会返回此代码，响应中必须包含有关冲突的信息
410	Gone	客户端请求的资源已经不存在。410 不同于 404，如果资源以前存在而现在被永久删除则可使用 410 代码，网站设计人员可通过 301 代码指定资源的新位置
411	Length Required	客户端发送的请求信息缺少 Content-Length，服务器无法处理该请求
412	Precondition Failed	服务器不满足客户端请求信息的某一先决条件
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果服务器只是暂时无法处理，则会包含一个 Retry-After 的响应信息
414	Request-URI Too Large	客户端请求的 URI（通常为网址）过长，服务器无法处理
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式
416	Requested range not satisfiable	客户端请求的范围无效
417	Expectation Failed	服务器无法满足 Expect 的请求头的要求

⑤ 5xx：服务器端错误——客户端的请求合法，但是服务器未能实现，如表 5-7 所示。

表 5-7　状态码的第一个数字为 5

状态码	英 文 名 称	中 文 描 述
500	Internal Server Error	服务器内部出现错误，无法完成请求
501	Not Implemented	服务器不具备完成请求的功能，无法完成请求
502	Bad Gateway	作为网关或代理的服务器，从远端服务器接收到一个无效的请求
503	Service Unavailable	由于超载或系统维护，服务器暂时无法处理客户端的请求。延时的长度可包含在服务器的 Retry-After 头信息中
504	Gateway Time-out	作为网关或代理的服务器，没有及时从远端服务器获取请求
505	HTTP Version not supported	服务器不支持请求的 HTTP 的版本，因而无法完成处理

（2）HTTP 响应头的格式与请求头相同，并且也需要在末尾空一行表示响应头的结束，

常见的响应头标识如表 5-8 所示。

表 5-8　常见的响应头标识

标　识	含　义
Allow	表示服务器支持的请求方法(如 GET、POST 等)
Content-Encoding	表示文档编码的方法
Content-Length	表示文档内容的长度。只有当浏览器使用持久 HTTP 连接时才需要这个数据
Content-Type	表示文档属于什么 MIME 类型。Servlet 中 Content-Type 默认值为 text/plain，但通常需要显式地将该值指定为 text/html
Date	表示消息发送的时间，时间的描述格式由 RFC 822 定义。例如，Date：Sat，06 May 2017 12：16：56 GMT
Expires	表示文档缓存的保留时间，超过该值会认为文档已经过期，从而不再保留缓存
Last-Modified	表示文档最后一次改动的时间
Location	表示客户应当去哪个位置提取文档
Refresh	表示浏览器应该在多久之后刷新文档，以秒作为单位
Server	服务器名称。Servlet 一般不设置这个值，而是由 Web 服务器自己设置
Set-Cookie	非常重要的 header，它可以将 cookie 发送到客户端浏览器，每写入一个 cookie 都会生成一个 Set-Cookie
WWW-Authenticate	表示在 Authorization 头中应提供的授权信息的类型
P3P	用于跨域设置 Cookie，这样可以解决 iframe 跨域访问 cookie 的问题

(3) 响应体就是服务器返回的资源的内容，即整个 HTML 文件。

5.2.4　示例

HTTP 自从 1990 年问世以来，经过了数次完善和改进，目前数十种编程语言中均封装了便于进行 HTTP 调用的 API，本例使用 Python 模拟发送 HTTP 请求并解析响应内容，简单介绍与 HTTP 有关的部分模块。

1. 模块介绍

urllib 与 urllib2 均是 Python 中用于实现网络请求的标准库。在 Python 3 中，urllib2 不再保留，它的内容迁移到了 urllib 模块中。urllib 的主要作用是从指定的 URL 获取数据以及对 URL 字符串进行格式化处理。

urllib 中包含 4 个模块，分别是 urllib. request、urllib. error、urllib. parse 与 urllib. robotparser。在模拟 HTTP 请求的过程中，主要使用的是 urllib. request 与 urllib. parse 两个模块，其余模块与网络爬虫有关。其中，urllib. request 是 HTTP 请求模块，只需要调用相关的库方法，并给相关方法传入 URL 以及其他额外的参数，就可以实现发送 HTTP 网络请求；urllib. parse 则是一个工具模块，提供了很多 URL 的处理方法，如拆分、解析、合并等。

Python 同时封装了一个名为 http 的库，其中用于开发 HTTP 的模块包括 http. client、http. server、http. cookies 以及 http. cookiejar。其中，http. client 是一个底层的 HTTP 客户端，被更高层的 urllib. request 模块所使用；http. server 包含基于 SocketServer 的基本 HTTP 服务器的类；http. cookies 与 http. cookiejar 用于实现对 Cookie 的管理。

2. 实践过程

1) 发起 HTTP 请求

首先模拟浏览器发起一个 HTTP 请求，使用 urllib. request 模块中的 urlopen()函数，

函数原型如下：

```
urllib.request.urlopen(url, data = None, [timeout, ] * , cafile = None, capath = None, context = None)
```

url：String 类型的请求链接，这个是必传参数，其他都是可选参数。

data：bytes 类型的内容。使用 data 参数时，会以 POST 请求方式提交表单。

timeout：请求超时时间，单位是秒。

cafile，capath：CA 证书和 CA 证书的路径，如果使用 HTTPS 则需要用到。

context：本参数必须是 ssl.SSLContext 类型，用来指定 SSL 设置。

该函数也可以单独传入 urllib.request.Request 对象作为参数。该函数返回结果是一个 http.client.HTTPResponse 对象。

使用 urlopen()函数即可实现抓取网页、设置请求超时、提交数据等功能，但如果请求中需要加入请求头、指定请求方式等信息，就必须利用 urllib.request.Request 对象来构建一个请求，将其作为 urllib.request.urlopen()的参数传入，urllib.request.Request 的构造方法如下：

```
urllib.request.Request(url, data = None, headers = {}, origin_req_host = None, unverifiable = False, method = None)
```

url：String 类型的请求链接。这个是必传参数，其他都是可选参数。

data：bytes 类型的内容。使用 data 参数时，会以 POST 请求方式提交表单。

headers：指定发起的 HTTP 请求的头部信息。headers 是一个字典。除了在 Request 的构造函数中添加外，还可以通过调用 Request 的 add_header()方法来添加请求头。

origin_req_host：表示请求方的 host 或者 IP 地址。

unverifiable：表示该请求无法验证，默认值是 False。例如，请求一个网页中的图片时，用户并没有权限来自动抓取图像，此时应将 unverifiable 的值设置为 True。

method：发起 HTTP 请求的方式，如 GET、POST、DELETE、PUT 等。

2）解析响应内容

通过 urllib.request.urlopen()函数成功发送 HTTP 请求之后，便可获取到一个 http.client.HTTPResponse 类型的对象。可以使用变量 resp 接收返回结果，通过对 resp 的解析即可获得需要的内容。以下是对 resp 常用的部分解析操作。

```
#获取 HTTP 版本号，返回 10 表示 HTTP 1.0, 11 表示 HTTP 1.1
resp.version

#获取响应码
resp.status
resp.getcode()

#获取响应描述字符串
resp.reason

#获取实际请求的页面 url(防止重定向时使用)
resp.geturl()
```

```
#获取特定响应头的信息
resp.getheader(name = "Content - Type")

#获取响应头信息,返回二元元组列表
resp.getheaders()

#获取响应头信息,返回字符串
resp.info()

#读取响应体,需进行解码
resp.readline().decode('utf - 8')
resp.read().decode('utf - 8')
```

5.3 MQTT 协议

5.3.1 MQTT 协议介绍

1. MQTT 协议基本介绍

MQTT(Message Queuing Telemetry Transport,消息队列遥测传输)协议是一种消息协议,基于 ISO 标准下的发布/订阅范式,是运行于 TCP/IP 协议栈之上的应用层协议,所以理论上只要应用能支持 TCP/IP 协议栈,那么也同样能够支持 MQTT。它可以支持连接远程设备的实时可靠的消息服务,即使是处在代码量极为稀少和带宽有限的情况下。基于此优势,众多计算能力不足,或者处于低带宽、不可靠网络环境的远程传感器及控制设备可以通过 MQTT 获得良好的性能保障。

2. MQTT 协议的特点

由于 MQTT 低开销和即时性的特点,使得其在物联网领域中得到广泛的应用。此协议的主要特性有:

(1) 发布/订阅的消息模式。拥有一对多的消息发布功能,能实现应用程序耦合的解除,与 HTTP 和 CoAP 采用的请求/响应机制差别较大。

(2) 对负载内容屏蔽的消息传输机制。能够针对不当言论等,对消息订阅者所接收到的内容进行部分屏蔽。

(3) MQTT 协议使用开销很小的小型传输,1 字节控制报头,2 字节心跳报文,以此实现了数据传输和协议交换的最小化,从而减少了网络流量的占用。

(4) 遗言机制(Last Will)和遗嘱机制(Testament)。当客户端异常中断时会自动实现对应的告知,即通知同一主题下发送遗言或遗嘱的设备已经断开了连接。

(5) 对于消息传输,MQTT 提供了以下三种 QoS(Quality of Service,质量服务)。

① 至多一次:此级别有消息丢失或者重复的可能性,在消息发布的过程中,对于底层 TCP/IP 网络是完全的依赖。因为可靠性较低,所以一般应用于对于数据丢失一次读记录敏感度不高的环境。这是因为在这种情况下,将会有第二次发送在不久后到达,例如设备传感器。

② 至少一次:此级别保证消息可以到达,但不能阻止消息重复的情况。

③ 只有一次:保证消息能够到达且只到达一次,适用于消息重复或丢失会导致不正确

结果的应用环境，例如计费系统。

3. 优缺点

作为一款极受欢迎的轻量级传输协议，MQTT 主要具有以下几点优势：

(1) 具有极佳的轻量化性能，适用于绝大多数的受限网络；

(2) 用户能够灵活选择具有给定功能的服务质量；

(3) 经过了 OASIS 技术委员会的标准化；

(4) 协议简洁轻巧，数据冗余量低；

(5) 能在处理器和内存资源有限的嵌入式设备中运行，因为其支持所有平台，所以几乎能将全部联网物品和互联网建立起连接。

但是 MQTT 同时也在以下几方面有待改进：

(1) 基于 TCP 的连接，功耗较高；

(2) 缺乏加密功能；

(3) 服务器端实现难度大，虽然已经有了 C++版本的服务端组件，但是并不开源。

5.3.2 MQTT 协议的原理

1. 协议架构

如图 5-3 所示，MQTT 协议实现的基本组成有消息的发布者(Publisher)、代理(Broker)以及订阅者(Subscriber)三部分。在这几种角色中，客户端为发布者和订阅者，服务器端为代理，发布者也可以担任订阅者的角色。

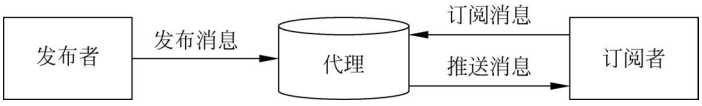

图 5-3 MQTT 协议的基本组成

MQTT 客户端是一个应用程序或者设备，在客户端中应用 MQTT 协议，其具有以下基本功能：

(1) 建立到服务器的网络连接；

(2) 发布信息，此信息可能会被其他客户端订阅；

(3) 订阅信息，此信息为其他客户端发布的；

(4) 退订或删除信息，此信息属于应用程序发送的；

(5) 实现与服务器之间连接的断开。

“消息代理”是 MQTT 服务器的另一种称谓，其处于消息发布者和订阅者之间，基本功能如下：

(1) 接受网络连接，此网络连接来自客户；

(2) 接收应用信息，此应用信息由客户发布；

(3) 处理订阅和退订请求，此请求来自客户端；

(4) 转发应用消息，转发给符合条件的已订阅客户端；

(5) 关闭来自客户端的网络连接消息传输。

MQTT 能够为客户端与服务器提供一个有序的、无损的、基于字节流的双向传输通道。在其上传输的消息的主要组成部分是主题(Topic)和负载(Payload)。其中，主题即消息的

类型，负载即消息的内容。订阅者订阅了某一主题之后，便会收到该主题对应的负载内容。

MQTT 的订阅包含主题筛选器(Topic Filter)和 QoS，若应用数据是借助 MQTT 网络发送的，MQTT 将关联起和其有关的服务质量和主题。

会有一个会话(Session)形成于客户端与服务器成功建立连接后，多个订阅包含于一个会话之中，不同的主题筛选器可以应用于不同的订阅。会话不拘泥于一种连接方式，其可以跨越连续的多个客户端与服务器之间的网络连接，也能够仅存在于一个网络内。

2. 数据包结构

如图 5-4 所示，根据 MQTT 协议，MQTT 数据包的组成部分有以下几个：

(1) 固定头(Fixed Header)：作用为指示数据包类型及其分组类标识，存在于所有 MQTT 数据包中；

(2) 可变头(Variable Header)：可变头是否存在及其具体内容是由数据包类型决定的，与固定头不同的是，可变头并不是存在于所有 MQTT 数据包中，其仅存在于部分 MQTT 数据包中；

(3) 消息体(Payload)：含义为客户端收到的具体内容，同可变头一样，其仅存在于部分 MQTT 数据包中。

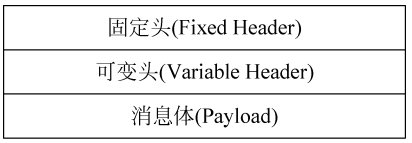

图 5-4　MQTT 数据包结构

前文中已提到，MQTT 的固定头部只有 1 字节，正是这一特点，实现了数据传输和协议交换的最小化。MQTT 的固定头部结构如图 5-5 所示，主要包含以下 5 部分。

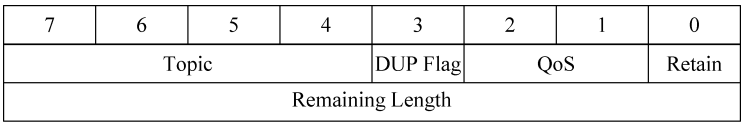

图 5-5　MQTT 的固定头部结构

(1) Topic：即消息类型，使用 4 位二进制数表示，其中 0 和 15 位置属于保留待用，共 14 种消息事件类型，如表 5-9 所示。

表 5-9　不同消息类型的含义和作用

数值	名　字	含　义	作　用
1	CONNECT	连接服务器	客户端到服务器端的网络连接建立后，客户端发送给服务器端的第一个报文必须是 CONNECT 报文
2	CONNACK	确认连接请求	服务器端发送 CONNACK 报文响应从客户端收到的 CONNECT 报文，服务器端发送给客户端的第一个报文必须是 CONNACK
3	PUBLISH	发布消息	从客户端向服务器端或者服务器端向客户端传输一个应用消息
4	PUBACK	发布确认	对 QoS 1 等级的 PUBLISH 报文的响应
5	PUBREC	发布收到	对 QoS 2 等级的 PUBLISH 报文的响应，它是 QoS 2 等级协议交换的第二个报文
6	PUBREL	发布释放	对 PUBREC 报文的响应，它是 QoS 2 等级协议交换的第三个报文

续表

数值	名　字	含　义	作　用
7	PUBCOMP	发布完成	对 PUBREL 报文的响应，它是 QoS 2 等级协议交换的第四个也是最后一个报文
8	SUBSCRIBE	订阅主题	客户端向服务器端发送 SUBSCRIBE 报文，用于创建一个或多个订阅
9	SUBACK	订阅确认	服务器端向客户端发送 SUBACK 报文，作用是确认其是否已收到且正在处理的 SUBSCRIBE 报文，SUBACK 报文包含一个返回码清单，它们指定了 SUBSCRIBE 请求的每个订阅被授予的最大 QoS 等级
10	UNSUBSCRIBE	取消订阅	客户端发送 UNSUBSCRIBE 报文给服务器端，用于取消订阅主题
11	UNSUBACK	取消订阅确认	服务器端发送 UNSUBACK 报文给客户端，用于确认收到 UNSUBSCRIBE 报文
12	PINGREQ	心跳请求	客户端发送 PINGREQ 报文给服务器端，用于在没有任何其他控制报文从客户端发给服务器时，告知服务器端客户端还“活”着，同时请求服务端发送响应确认它还“活”着，由此使用网络以确认网络连接没有断开
13	PINGRESP	心跳响应	服务器端发送 PINGRESP 报文响应客户端的 PINGREQ 报文，表示服务器端还“活”着
14	DISCONNECT	断开连接	DISCONNECT 报文是客户端发给服务器端的最后一个控制报文，表示客户端正常断开连接

(2) DUP Flag：一个打开标志，保证消息可靠传输，默认为“0”，只占用 1 字节，表示第一次发送。当值为“1”时，表示当前消息先前已经被传送过。

(3) QoS：即服务质量，由 2 位二进制数表示，如表 5-10 所示。

表 5-10　服务质量

数　值	二进制表示	含　义
0	00	至多一次，发完即丢弃
1	01	至少一次，需要确认回复
2	10	只有一次，需要确认回复
3	11	暂无含义，保留待用

(4) Retain：此标识的含义是发布保留，也就是说代表着此次推送的信息会被服务器保留，当出现了新的订阅者时，就将取出最新的一个 Retain＝1 的消息推送，如果没有，则推迟至当前订阅者后释放。

(5) Remaining Length：消息体的总大小是由固定头的第二字节保存的，可以扩展此字段，扩展的最大字节数受限，最大字节数为 4 字节。保存的原理可以概括为长度保存在每一字节中的前 7 位，标识位于最后一位。若长度不足，那么最后一位就会显示为“1”，此时若要保存完整信息，就要采用 2 字节。

由于数据包类型的不一致，可变头的内容也是不相同的。用作数据包的标识是一种较为典型的应用，这时的可变头含有两个字段：主题名称(Topic Name)和数据包标识符(Packet Identifier)。主题名称标识有效数据发布的信息通道，数据包标识符字段则仅出现

在 QoS 级别为 1 或 2 的 PUBLISH 数据包中。

消息体的作用是包含 4 种类型数据包的具体消息，这 4 种类型的数据包分别为 CONNECT、SUBSCRIBE、SUBACK、UNSUBSCRIBE，对应的具体介绍如下。

(1) CONNECT，主要的消息体内容为订阅的 Topic、Message、客户端的 ClientID 以及用户名和密码；

(2) SUBSCRIBE，消息体内容为 QoS 及一系列需要订阅的主题；

(3) SUBACK，消息体内容是服务器的确认和回复，对象是 SUBSCRIBE 申请的主题和 QoS；

(4) UNSUBSCRIBE，消息体内容是需要订阅的主题。

5.3.3 示例

MQTT 官方网站(https://mqtt.org/)中提供了基于多种语言的 API 供开发者使用，本例介绍在 Windows 中使用 Python 实现 MQTT 客户端功能的相关知识。在此之前，需要进行环境配置。

1. 环境配置

首先安装 MQTT 代理：EMQ X Broker 安装过程因平台而有差异，参照 https://www.emqx.io/docs/zh/v4.3/getting-started/install.html 官方网站教程即可。成功启动 EMQ 后，可通过浏览器访问 http://localhost：18083 admin/public 进入 EMQ 控制台，在工具→WebSocket 模块方便地进行客户端连接、主题订阅、消息接收、消息发布等测试和调试工作。

在命令行中使用指令 pip install paho-mqtt 安装 MQTT 客户端的支持库 paho-mqtt，其能够让应用程序简单方便地连接到 MQTT 代理进行消息发布、订阅主题和消息接收。出现如图 5-6 所示内容时表示安装成功，本次安装的版本为 paho-mqtt-1.6.1。

```
Successfully built paho-mqtt
Installing collected packages: paho-mqtt
Successfully installed paho-mqtt-1.6.1
```

图 5-6 安装成功示意图

2. 创建步骤

创建 MQTT 客户端的一般步骤为：

(1) 创建一个客户端实例；

(2) 使用任一 connect*()方法连接到 MQTT 代理；

(3) 调用任一 loop*()方法保持与 MQTT 代理通信；

(4) 使用 subscribe()方法订阅一个主题并接收消息；

(5) 使用 publish()方法向 MQTT 代理发布消息；

(6) 使用 disconnect()方法中断与 MQTT 代理的连接。

3. 函数介绍

(1) 构造方法。

```
Client(client_id="", clean_session=True, userdata=None, protocol=MQTTv311, transport="tcp")
```

client_id：将采用唯一客户端 ID 字符串来实现和 MQTT 代理的连接。若为 0 或者为 None，则分配的 ID 是随机生成的，此时 clean_session 参数一定要是 True。

clean_session：布尔值类型，用来确定客户端类型。如果为 True，当断开连接时，

MQTT 代理将移除该客户端的所有信息；如果为 False，客户端则为持久客户端，当断开连接时，订阅信息和消息队列将被 MQTT 保存；当断开连接时，客户端不会丢弃自己发送的消息。调用 connect()或者 reconnect()方法将导致重新发送消息，只有使用 reinitialise()方法可以将客户端重置为初始状态。

userdata：该参数是用户定义的数据，此数据可以为任意类型，并且将被传递给回调函数，存在一定延迟的更新方法是调用 user_data_set()方法。

protocol：一种 MQTT 协议版本，它是供客户端使用的，可以是 MQTTv31 或 MQTTv311。

transport：一种传输形式，默认为 tcp，但如果想通过套接字传输给 MQTT，那就应该修改成 websocket。

(2) connect * ()函数。例如：

```
connect(host, port = 1883, keepalive = 60, bind_address = "")
```

该函数为阻塞函数，代表客户端连接 MQTT 代理。

host：代表了代理的主机名或 IP 地址。

port：连接服务的端口号，默认为 1883。

keepalive：心跳检测时长。

bind_address：绑定此客户端本地网络的 IP 地址。

```
connect_async(host, port = 1883, keepalive = 60, bind_address = "")
```

connect_async()与 loop_start()函数结合使用以非阻塞的形式进行连接，在调用 loop_start()函数之前，连接不会完成。

(3) loop * ()函数。例如：

```
loop(timeout = 1.0)
```

该函数定期调用处理事件。

timeout：最大阻塞的秒数。

```
loop_start()
loop_stop(force = False)
```

loop_start() / loop_stop()函数实现了网络循环的线程接口，在执行 connect * ()函数之前或者之后调用一次 loop_start()函数，后台会自动运行一个线程调用 loop()函数，这样就释放了主线程去执行其他工作，避免发生阻塞，这个调用也处理重新连接到代理。调用 loop_stop()函数则停止后台线程。

```
loop_forever()
```

通过阻塞式网络循环处理事件，它有自动重连的功能，不会返回，除非客户端调用 disconnect()函数。

(4) subscribe()函数。例如：

```
subscribe(topic, qos = 0)
```

该函数订阅一个或多个主题。

topic：消息发布的主题，不能为 None 或者空字符。

qos：消息的服务质量等级，必须为 0、1 或 2，默认为 0。

该方法有三种不同的调用方式：

```
# 1.参数为字符串和整数,订阅 topic1
subscribe("my/topic1", 2)
# 2.参数为字符串和整数元组,订阅 topic2
subscribe(("my/topic2", 1))
# 3.参数为字符串和整数元组的列表,订阅 topic1 和 topic2
# 单次调用多个主题,比多次调用 subscribe()函数更有效
# 下面这行代码相当于先后执行 1 和 2
subscribe([("my/topic1", 2), ("my/topic2", 1)])
```

(5) publish()函数。例如：

```
publish(topic, payload = None, qos = 0, retain = False)
```

该函数表示客户端向 MQTT 代理发送一条消息。

topic：消息发布的主题，不能为 None 或者空字符。

payload：发送的消息内容，如果没有赋值或者赋值为 None，则将使用零长度的消息。传递 int 或者 float 型数据将会被转换为该数字的字符串，如果想发送真正的 int 或者 float 型数据，使用 struct.pack()函数去创建。

qos：消息的服务质量等级，必须为 0、1 或 2。

retain：设置为 True，MQTT 代理保留最后一条消息，以便分发给消息发布后的订阅者。

(6) disconnect()函数。例如：

```
disconnect()
```

该函数表示彻底与 MQTT 代理断开，使用该函数断开连接不会让代理发送遗嘱消息。

5.4 LwM2M 协议

5.4.1 LwM2M 协议介绍

1. LwM2M 协议基本介绍

LwM2M(Lightweight Machine-To-Machine)协议是由 OMA(Open Mobile Alliance)提出并定义的一种适用于资源有限终端设备管理的轻量级物联网协议，可以用于快速部署客户端、服务器模式的物联网业务。

LwM2M 协议为物联网设备的管理和应用建立了一套标准，提供了轻便小巧的安全通信接口及高效的数据模型，以实现 M2M 设备管理和服务支持。

2. LwM2M 协议的特点

LwM2M 协议主要具有以下几个突出的特点。

(1) LwM2M 协议采用了风格更加简洁易懂的 REST 架构，在降低开发复杂性的同时提高了系统的可伸缩性。为了更好地适用于资源有限的终端设备，LwM2M 协议舍弃了传统 HTTP 数据传输的方式，选择了更加轻便的 CoAP 来完成消息和数据传递。

(2) LwM2M 协议定义了一个以资源为基本单位的数据模型，结构紧凑高效，同时又具有极佳的扩展性。

(3) LwM2M 协议支持 C/S 架构,主要的实体有 LwM2M 服务器和 LwM2M 客户端。

5.4.2 LwM2M 协议的原理

1. LwM2M 协议的架构

LwM2M 协议的架构如图 5-7 所示。服务器(LwM2M Server)部署在 M2M 服务供应商处或网络服务供应商处,客户端(LwM2M Client)部署在各个 LwM2M 设备上,LwM2M 引导服务器(Bootstrap Server)或智能卡(Smart Card)用于对客户端完成初始的引导。

这些实体之间定义了 4 个接口。

(1) 引导接口(Bootstrap):用于向 LwM2M 客户端提供注册到 LwM2M 服务器的访问信息、客户端支持的资源信息等必要信息。这些引导信息可以由生产厂家预先存储在设备中,也可以通过 LwM2M 引导服务器或者智能卡提前写入设备。

(2) 客户端注册接口(Client Registration):使 LwM2M 客户端与 LwM2M 服务器互联,将 LwM2M 客户端的相关信息存储在 LwM2M 服务器上。客户端只有在完成注册之后才可以与服务器之间进行通信。

(3) 设备管理与服务实现接口(Device Management and Service Enablement):LwM2M 服务器作为主控方,向客户端发送指令,客户端对指令做出回应并将回应消息发送给服务器。

(4) 信息上报接口(Information Reporting):允许 LwM2M 服务器端向客户端订阅资源信息,客户端接收订阅后按照约定的模式向服务器端报告自己的资源变化情况。

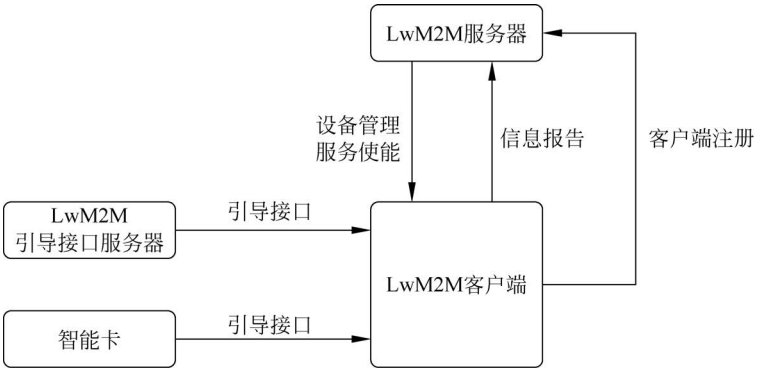

图 5-7 LwM2M 协议的架构

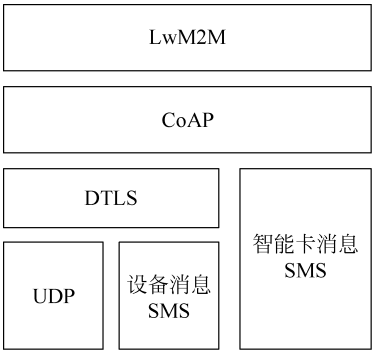

图 5-8 底层协议支撑

LwM2M 属于应用层协议,为了更加适应终端设备的轻量化要求,其使用 CoAP 作为传输层协议。CoAP 使用了基于 UDP 的 DTLS 安全传输协议,与 TCP 相比,其更适合运用在网络资源有限及无法确保设备始终在线的环境里。此外 CoAP 本身的消息结构非常简单,对报文进行了压缩,主要部分可以做到特别小巧,无须占用过多资源,如图 5-8 所示。

2. 数据模型

为了适应资源受限的终端设备,LwM2M 的数据模型同样必须足够简单。LwM2M 协议定义了一个以资源为

基本单位的数据模型，所有信息都可以抽象为资源以提供访问。每个资源可以携带数值，可以指向地址，以表示 LwM2M 客户端中每一项可用的信息。LwM2M 客户端可以拥有任意数量的资源。

资源必须存在于对象实例中，对象是逻辑上用于特定目的的一组资源的集合，一个对象可以有多个实例，对象的定义格式如下。

Name （对象名称）	Object ID （对象 ID）	Instances （实例数量）	Mandatory （强制性）	Object URN （对象统一资源名称）

其中，对象 ID 为 16 位无符号整型数据，实例数量可选单一实例（Single）或多个实例（Multiple），强制性可以选择强制（Mandatory）或可选（Optional），URN 格式为 urn：oma：LwM2M：{oma，ext，x}：{Object ID}。

LwM2M 协议预先定义了 8 个对象，如表 5-11 所示。

表 5-11　LwM2M 协议的 8 个对象

对 象 名 称	ID	含　义
Security（安全对象）	0	负载的安全模式，一些算法/密钥，服务器的短 ID 等信息
Server（服务器对象）	1	服务器的短 ID，注册的生命周期，观察的最小/最大周期，绑定模型等信息
Access Control（访问控制对象）	2	每个对象的访问控制权限
Device（设备对象）	3	设备的制造商、型号、序列号、电量、内存等信息
Connectivity Monitoring（连通性监控对象）	4	网络制式、链路质量、IP 地址等信息
Firmware（固件对象）	5	固件包，包的 URI、状态、更新结果等信息
Location（位置对象）	6	经纬度、海拔、时间戳等信息
Connectivity Statistics（连通性统计对象）	7	收集期间的收发数据量，包的大小等信息

与对象相同，一个资源也可以有多个实例，资源的定义格式为：

ID （资源 ID）	Name （资源名称）	Operations （操作类型）	Instances （实例数量）	Mandatory （强制性）	Type （类型）	Description （描述）

其中，操作类型可选 R（Read）、W（Write）或 E（Execute）；类型可选 String、Integer、Float、Boolean、Opaque、Time、ObjInk none。

5.4.3　示例

1. 关键代码

LwM2M 协议的主要开源实现有以下几个。

（1）OMA LwM2M DevKit：提供可视化界面与 LwM2M 服务器交互；

（2）Eclipse Leshan：基于 Java 语言，提供了 LwM2M 服务器与 LwM2M 客户端的实现；

（3）Eclipse Wakaama：基于 C 语言，提供了 LwM2M 服务器与 LwM2M 客户端的实现；

(4) AVSystem Anjay：基于C语言，提供了LwM2M客户端的实现。

以下对Wakaama开源协议栈预定义的一些结构体和函数进行介绍。

1) lwm2m_data_t

```
typedef enum                                    //枚举类型
{
    LWM2M_TYPE_UNDEFINED = 0,                   //对应0
    LWM2M_TYPE_OBJECT,                          //对应1
    LWM2M_TYPE_OBJECT_INSTANCE,                 //对应2
    LWM2M_TYPE_MULTIPLE_RESOURCE,               //对应3
    LWM2M_TYPE_STRING,                          //对应4
    LWM2M_TYPE_OPAQUE,                          //对应5
    LWM2M_TYPE_INTEGER,                         //对应6
    LWM2M_TYPE_FLOAT,                           //对应7
    LWM2M_TYPE_BOOLEAN,                         //对应8
    LWM2M_TYPE_OBJECT_LINK                      //对应9
} lwm2m_data_type_t;
typedef struct _lwm2m_data_t lwm2m_data_t;//用于存储数据

struct _lwm2m_data_t
{
    lwm2m_data_type_t type;                     //数据类型
    uint16_t    id;                             //数据ID
    union
    {
        bool                asBoolean;          //转换为bool类型
        int64_t             asInteger;          //转换为int64_t类型
        double              asFloat;            //转换为double浮点数类型
        struct
        {
            size_t          length;
            uint8_t *       buffer;
        } asBuffer;                             //转换为Buffer类型
        struct
        {
            size_t          count;
            lwm2m_data_t *array;
        } asChildren;                           //转换为数组或对象类型
        struct
        {
            uint16_t objectId;
            uint16_t objectInstanceId;
        } asObjLink;                            //转换为ObjLink类型
    } value;
};
```

lwm2m_data_t是协议栈定义的一种标准数据类型，用于存储各种协议中可能用到的数据，其中数据类型与成员之间的对应关系如下。

```
LWM2M _ TYPE _ OBJECT, LWM2M _ TYPE _ OBJECT _ INSTANCE,  LWM2M _ TYPE _ MULTIPLE _ RESOURCE:
value.asChildren
```

```
LWM2M_TYPE_STRING, LWM2M_TYPE_OPAQUE: value.asBuffer
LWM2M_TYPE_INTEGER,LWM2M_TYPE_TIME: value.asInteger
LWM2M_TYPE_FLOAT: value.asFloat
LWM2M_TYPE_BOOLEAN: value.asBoolean
```

2) lwm2m_object_t

```
typedef struct _lwm2m_object_t lwm2m_object_t;
struct _lwm2m_object_t
{
    struct _lwm2m_object_t *    next;           //仅供内部调用
    Uint16_t                    objID;          //当前对象的 ID
    Lwm2m_list_t * instanceList;                //对象实例的列表
    Lwm2m_read_callback_t       readFunc;       //read()回调函数
    lwm2m_write_callback_t      writeFunc;      //write()回调函数
    lwm2m_execute_callback_t    executeFunc;    //execute()回调函数
    lwm2m_create_callback_t     createFunc;     //create()回调函数
    lwm2m_delete_callback_t     deleteFunc;     //delete()回调函数
    lwm2m_discover_callback_t   discoverFunc;   //discover()回调函数
    void * userData;                    //用户自定义的数据,可存储任意数据类型和大小的指针
};
```

其中,objID 即 Object ID,用于标识当前对象的 ID; instanceList 是对象实例的列表; userData 是用户自定义的数据,可以存储任意数据类型和大小的指针,一般是跟该对象密切相关的信息。

lwm2m_object_t 抽象了客户端实体中的资源,各种资源通过其对应的 URI 进行定位,并在定位之后通过特定的方法对其进行操作,每种方法通常对应一个或一组函数。

LwM2M 定义了资源的标准操作方法,分别对应 6 个事件回调函数(Callback),这些函数在对象收到对应方法请求时会被调用,包括 read()、write()、discover()、create()、delete()、execute(),并且对于特定的标准对象,LwM2M 协议文档规定了每种方法的响应流程。但是对于自定义的对象,可以只实现其中的某种或某几种方法,也可以根据需求自行约定方法的响应方式。6 个回调函数分别如下。

```
typedef uint8_t ( * lwm2m_read_callback_t) (uint16_t instanceId, int * numDataP, lwm2m_data_t
** dataArrayP, lwm2m_object_t * objectP);                //read()回调函数
typedef uint8_t ( * lwm2m_discover_callback_t) (uint16_t instanceId, int * numDataP, lwm2m_
data_t ** dataArrayP, lwm2m_object_t * objectP);        //discover()回调函数
typedef uint8_t ( * lwm2m_write_callback_t) (uint16_t instanceId, int numData, lwm2m_data_t *
dataArray, lwm2m_object_t * objectP);                   //write()回调函数
typedef uint8_t ( * lwm2m_execute_callback_t) (uint16_t instanceId, uint16_t resourceId,
uint8_t * buffer, int length, lwm2m_object_t * objectP); //execute()回调函数
typedef uint8_t ( * lwm2m_create_callback_t) (uint16_t instanceId, int numData, lwm2m_data_t
* dataArray, lwm2m_object_t * objectP);                 //create()回调函数
typedef uint8_t ( * lwm2m_delete_callback_t) (uint16_t instanceId, lwm2m_object_t *
objectP);                                               //delete()回调函数
```

这些回调函数根据命名,分别作为实体接收到对应的方法的请求后所触发的动作入口。其中各个参数的含义如下。

instanceId: 触发该次事件对象实例的 ID;

dataArrayP：由该次操作返回的数据组成的链表，由用户函数填入，返回给发送方；

numDataP：指出 dataArrayP 中含有的 lwm2m_data_t 数目，由用户函数填入，返回给发送方；

objectP：触发该次事件的 Object 的引用，由协议栈填入；

numData：指明 dataArray 中包含的 lwm2m_data_t 的数目；

dataArray：指向该次事件发生时，接收到的 lwm2m_data_t 数据；

resourceId：触发该次事件的资源 ID；

buffer：指向该次事件发生时，接收到的普通数据；

length：指明 buffer 的长度。

3）lwm2m_context_t

```
typedef struct
{
#ifdef LWM2M_CLIENT_MODE                          //客户端
    lwm2m_client_state_t state;
    char *                    endpointName;       //端点名
    char *                    msisdn;             //用于识别客户的唯一号码
    char *                    altPath;
    lwm2m_server_t *          bootstrapServerList; //当前服务器列表
    lwm2m_server_t *          serverList;         //当前连接服务器列表
    lwm2m_object_t *          objectList;         //当前 object 列表,包含所有管理数据
    lwm2m_observed_t *        observedList;       //当前 observed 列表
#endif
#ifdef LWM2M_SERVER_MODE                          //服务器
    lwm2m_client_t *          clientList;         //所有连接的客户端列表
    lwm2m_result_callback_t   monitorCallback;    //打印当前状态
    void *                    monitor;            //指向 lwm2m_context_t 的镜像
#endif
#ifdef LWM2M_BOOTSTRAP_SERVER_MODE                //启动服务器
    lwm2m_bootstrap_callback_t bootstrapCallback;
    void *                    bootstrap;
#endif
    uint16_t                  nextMID;            //用于监视 Resource
    lwm2m_transaction_t *     transactionList;    //业务列表,供代理服务器使用
    void *                    userData;           //用户自定义的数据
} lwm2m_context_t;
```

lwm2m_context_t 抽象了一个正在运行的服务器、客户端或者启动服务器的实体。其中参数的含义如下。

bootstrapServerList：当前代理服务器列表；

serverList：当前连接服务器列表；

objectList：当前 object 列表，包括所有管理数据；

observedList：当前 observed 列表；

clientList：所有连接客户端列表；

monitorCallback：打印当前状态；

monitor：指向 lwm2m_context_t；

nextMID：供监视 Resource 使用；

transactionList：业务列表，供代理服务器使用。

4）command_desc_t

```
typedef struct
{
    char *            name;       //命令名
    char *            shortDesc;  //命令的简要描述
    char *            longDesc;   //命令的完整描述
    command_handler_t callback;
    void *            userData;   //用户自定义的数据
} command_desc_t;
```

command_desc_t 的功能是处理命令行的操作。当使用命令行输入命令时，callback 会被调用。命令和功能可以根据个人需求添加和修改。最终这些信息都会保存到 lwm2m_context_t 结构体中。

5）lwm2m_client_t

```
typedef struct _lwm2m_client_object_
{
    struct _lwm2m_client_object_ * next;          //与 lwm2m_list_t::next 一致
    uint16_t                       id;            //与 lwm2m_list_t::id 一致
    lwm2m_list_t *                 instanceList;  //实例列表
} lwm2m_client_object_t;

typedef struct _lwm2m_observation_
{
    struct _lwm2m_observation_ *   next;          //与 lwm2m_list_t::next 一致
    uint16_t                       id;            //与 lwm2m_list_t::id 一致
    struct _lwm2m_client_ *        clientP;       //连接的客户端
    lwm2m_uri_t                    uri;           //统一资源标识符
    lwm2m_status_t                 status;
    lwm2m_result_callback_t        callback;
    void *                         userData;      //用户自定义数据
} lwm2m_observation_t;

typedef struct _lwm2m_client_
{
    struct _lwm2m_client_ *        next;          //与 lwm2m_list_t::next 一致
    uint16_t                       internalID;    //与 lwm2m_list_t::id 一致
    char *                         name;          //客户端名
    lwm2m_binding_t                binding;
    char *                         msisdn;        //用于标识客户的唯一号码
    char *                         altPath;
    bool                           supportJSON;
    uint32_t                       lifetime;      //存活时间
    time_t                         endOfLife;     //终止时间
    void *                         sessionH;
    lwm2m_client_object_t *        objectList;
    lwm2m_observation_t *          observationList;//记录对资源的观察情况
```

```
} lwm2m_client_t;
```

lwm2m_client_t 描述了远程客户端的基本信息，其中实体 lwm2m_observation_t 用于在服务器端中记录对应客户端资源的观察情况；sessionH 成员是客户端和服务器的会话记录。

6）lwm2m_server_t

```
typedef struct _lwm2m_server_
{
    struct _lwm2m_server_ * next;         //与 lwm2m_list_t::next 一致
    uint16_t                secObjInstID; //与 lwm2m_list_t::id 一致
    uint16_t                shortID;      //服务器的 ID, 对于 BootStrap Server,该值可能为 0
    time_t                  lifetime;     //注册的生存时间,以秒为单位
    //填 0 代表默认值 86400 秒,也可用作启动服务器的延迟时间
    time_t                  registration; //上次注册的时间,以秒为单位
    //或表示启动服务器的客户端延迟时间结束
    lwm2m_binding_t         binding;      //客户端与服务器端的连接方式
    void *                  sessionH;
    lwm2m_status_t          status;
    char *                  location;
    bool                    dirty;
    lwm2m_block1_data_t *   block1Data;   //用于处理 block1 数据的缓冲区
    //应当用 list 替换,以支持服务器的多个 block1 传输
} lwm2m_server_t;
```

lwm2m_server_t 描述了远程服务器实体，这个实体应当只被客户端实体用来记录远端服务器的信息。其中 lwm2m_binding_t 和 lwm2m_status_t 分别记录了与该服务器的通信方式以及与该服务器的通信状态。

2. 流程介绍

使用 LwM2M 完成个人需求，其本质上就是添加一个对象，并完善其对应的一些回调函数，具体流程如下：

（1）根据源码风格添加 object_objectname.c 文件；

（2）在 object_objectname.c 文件中添加 objectname_data_t 结构体；

（3）在 object_objectname.c 文件中添加 prv_res2tlv()函数；

（4）根据实际需求在 object_objectname.c 文件中添加 prv_ objectname_read()、prv_objectname_write()、prv_objectname_execute()、prv_objectname_create()、prv_objectname_delete()、prv_objectname_discover()等函数，供服务器回调使用；

（5）在 object_objectname.c 文件添加 display_object_objectname()函数，供打印使用；

（6）在 object_objectname.c 文件中添加 get_object_objectname()函数，供 userData 初始化；

（7）在 object_objectname.c 文件中添加 free_object_objectname()函数，供 userData 释放；

（8）在 object_objectname.c 文件的 main()函数中添加 objArray［LWM2M_objectname_OBJECT_ID］，其中 LWM2M_ objectame _OBJECT_ID 是标识每个 object 唯一 ID 的宏定义；

(9) 在 main 函数中添加 free_object_objectname()函数；

(10) 在 prv_display_objects()函数中添加 display_object_objectname()函数；

(11) 在 lwm2mclient.h 中添加函数声明；

(12) 在 CMakeLists SOURCES 变量中添加 object_objectname.c。

5.5 Modbus 协议

5.5.1 Modbus 协议介绍

1. Modbus 协议基本介绍

Modbus 协议是一种串行通信协议，它广泛应用于工业控制领域。控制器以及其他设备之间可以通过 Modbus 协议实现通信。Modbus 协议定义了一个控制器能识别的消息结构。它对控制器与其他设备之间请求访问和应答回应的过程进行了描述，并对错误检测和记录的规范、报文字段和内容的公共格式做出了明确的规定。按照报文格式的不同，可将其分为 Modbus-RTU、Modbus-ASCII、Modbus-TCP，前两者在串行通信控制网络中应用较多，例如 RS-485、RS-232 等，而后者主要应用于基于以太网 TCP/IP 通信的控制网络中。

2. Modbus 协议的特点

Modbus 协议属于应用层协议，凭借着其开放性、高可靠性、高效简单性、免费等优点，成为了工业领域通信协议的业界标准，是工业现场电子设备之间常用的连接方式。其主要具有以下几个特点。

(1) Modbus 通信结构为一对多的主从查询模式，即主从(Master-Slave)模式。主控设备方作为主节点，将其所使用的协议称为 Modbus Master；被控设备方作为从节点，将其使用的协议称为 Modbus Slave。

(2) Modbus 网络上从节点可以有多个，但主节点有且只有一个。主节点按照通信协议对从节点发出通信请求，从节点收到主节点的请求后，响应后再向主节点回复应答消息。

(3) Modbus 协议是一个标准的、开放的协议，用户可以免费使用。目前，共有超过 400 家厂家、超过 600 种产品支持 Modbus 协议。

(4) Modbus 对如 RS-232、RS-485 等电气接口支持良好，其还可以在各种介质上传输，如双绞线、光纤、无线等。

(5) Modbus 的帧格式简单、紧凑且通俗易懂。这使得用户可以很容易地上手使用，也使得厂商开发变得更简单。

5.5.2 Modbus 协议的原理

1. Modbus 协议的架构

Modbus 的工作方式是请求/应答。如图 5-9 所示，在 Modbus 架构中，主站(Master)只有一个，从站

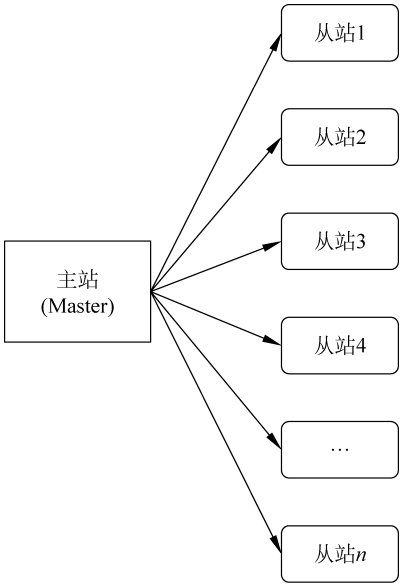

图 5-9 Modbus 协议的架构示意图

(Slave)有多个,每次通信都是由主站向从站先发送指令,形式可以是向所有从站的广播或是向特定从站的单播。然后从站对指令进行响应,并按要求应答,或者报告异常。当主站没有向从站发送请求时,从站不会自己发出数据,并且从站和从站之间不能直接通信。

在使用 TCP 通信时,主站为客户端,主动建立连接;从站为服务器端,等待连接。此过程中主站与从站之间的关系如图 5-10 所示。

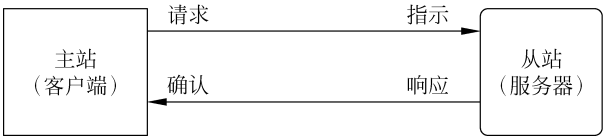

图 5-10　TCP 通信中主从站关系

Modbus 消息服务在 TCP/IP 网络上连接的设备之间提供客户机/服务器通信,主要有 4 种类型消息。

(1) 请求:客户端在网络上发送用于启动事务的消息;

(2) 确认:客户端接收到的响应消息;

(3) 指示:服务器收到的请求消息;

(4) 响应:服务器发送的响应消息。

2. 数据单元

Modbus 协议定义了一个简单协议数据单元(PDU),它与基础通信层无关。在特定总线或网络上,Modbus 协议映射能够在应用数据单元(ADU)上引入一些附加域。

如图 5-11 所示,应用数据单元由以下 4 部分组成。

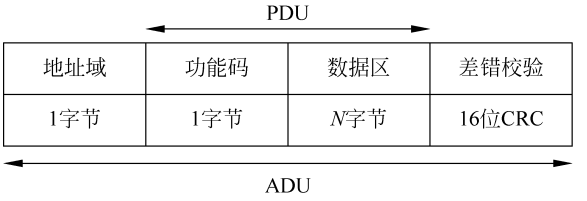

图 5-11　数据单元示意图

(1) 地址域:位于通信信息帧的第一字节,可以是 0x00～0xFF 范围内的值。每个从站具有唯一的地址码,在通信过程中会对地址码进行检查,只有地址码符合的从站才能响应回送信息。其中,0xFF 为广播地址。

(2) 功能码:占用 1 字节,由客户端发起请求,用于向服务器指示将执行哪种操作。常用功能码如表 5-12 所示。

表 5-12　常用功能码

功能码	名　　称	功　　能
0x01	读线圈状态	读位(读 *N* 位)—读从机线圈寄存器,位操作
0x02	读输入离散量	读位(读 *N* 位)—读离散输入寄存器,位操作
0x03	读多个寄存器	读整型、字符型、状态字、浮点型(读 *N* 个字)—读保持寄存器,字节操作
0x04	读输入寄存器	读整型、状态字、浮点型(读 *N* 个字)—读输入寄存器,字节操作

续表

功能码	名　　称	功　　能
0x05	写单个线圈	写位(写 1 位)—写线圈寄存器,位操作
0x06	写单个保持寄存器	写整型、字符型、状态字、浮点型(写 1 个字)—写保持寄存器,字节操作
0x0F	写多个线圈	写位(写 N 位)—强置一串连续逻辑线圈的通断
0x10	写多个保持寄存器	写整型、字符型、状态字、浮点型(写 N 个字)—把具体的二进制值装入一串连续的保持寄存器

(3) 数据区：数据区的内容为二进制数,可以是数据开关量或模拟量的输入/输出,也可以是寄存器、参考地址等。这些数据的参考地址在综合控制装置中均从 1 开始,但在通信过程中参考地址则是从 0 开始,所以读写地址 N 时使用的实际地址数据为 $N-1$。

(4) 差错校验码：差错校验区域使用循环冗余码(CRC),包含 2 字节。用于检验通信数据传送过程中的信息是否有误,错误的数据可以放弃,这样增加了系统的安全和效率。发送设备会计算一次 CRC,并将其放置于发送信息帧的尾部。接收设备在接收消息之后,再根据接收到的信息重新计算一次 CRC,比较计算得到的 CRC 是否与接收到的 CRC 相符。如果两者不相符,则表明出错,该信息应当放弃。

5.5.3　示例

Python 中封装了三个与 Modbus 开发相关的库：modbus_tk、pymodbus 以及 MinimalModbus。其中,modbus_tk 支持完整 Modbus 协议栈的实现,支持 Modbus-TCP 与 Modbus-RTU 两种格式；pymodbus 使用 twisted(一个 Python 的异步库)实现了 Modbus 的完整协议,因而支持异步通信；MinimalModbus 则只支持 Modbus-RTU 的格式。

modbus_tk 的下载地址为 https://pypi.org/project/modbus_tk/,以下使用 modbus_tk 模拟一个 Modbus Master,对 Modbus Slave 进行操控。

1. 导入功能包

```
import modbus_tk.modbus_tcp as mt
import modbus_tk.defines as md
```

2. 建立连接

与远程 Slave 建立连接,监听 502 端口,并设置超时时间为 5 秒。

```
master = mt.TcpMaster("192.168.2.20", 502)   //IP 地址和端口
master.set_timeout(5.0)                      //超时时间,单位为秒
```

3. 具体操控

之后即可通过 execute()方法进行具体操控,该方法原型如下。

```
execute(slave, function_code, starting_address, quantity_of_x, output_value)
```

该方法中各参数的含义如下。

slave：Slave 的编号,范围为 1～247,为 0 时表示广播所有的 Slave；

function_code：功能码；

starting_address：开始地址；

quantity_of_x：寄存器/线圈的数量；

output_value：一个整数或可迭代的值，例如一个整数数组。

常见操作如下。

```
#取到的所有寄存器的值
val = master.execute(slave = 1, function_code = md.READ_HOLDING_REGISTERS, starting_address = 1,
quantity_of_x = 3, output_value = 5)
#获取第一个寄存器的值
val[0]
#从地址 0 开始,读取 16 个保持寄存器
master.execute(1, md.READ_HOLDING_REGISTERS, 0, 16)
#从地址 0 开始,读取 16 个输入寄存器
master.execute(1, md.READ_INPUT_REGISTERS, 0, 16)
#从地址 0 开始,读取 16 个线圈寄存器
master.execute(1, md.READ_COILS, 0, 16)
#从地址 0 开始,读取 16 个离散输入寄存器
master.execute(1, md.READ_DISCRETE_INPUTS, 0, 16)
#单个读写寄存器操作
#从地址 0 开始,向保持寄存器写入内容 21
master.execute(1, md.WRITE_SINGLE_REGISTER, 0, output_value = 21)
master.execute(1, md.READ_HOLDING_REGISTERS, 0, 1)
#从地址 0 开始,向线圈寄存器写入内容 0(位操作)
master.execute(1, md.WRITE_SINGLE_COIL, 0, output_value = 0)
master.execute(1, md.READ_COILS, 0, 1)
#多个寄存器读写操作
#从地址 0 开始,向 4 个保持寄存器依次写入内容 20,21,22,23
master.execute(1, md.WRITE_MULTIPLE_REGISTERS, 0, output_value = [20,21,22,23])
master.execute(1, md.READ_HOLDING_REGISTERS, 0, 4)
#从地址 0 开始,将 4 个线圈寄存器全部写入 0
master.execute(1, md.WRITE_MULTIPLE_COILS, 0, output_value = [0,0,0,0])
master.execute(1, md.READ_COILS, 0, 4)
```

5.6 本章小结

本章首先介绍了物联网网关在网络层次中的位置及应用场景，在此基础上，从协议简介、协议特点、基本原理、重要性质等几方面针对物联网中常见的几种网关协议 MQTT、HTTP、LwM2M、Modbus 进行了介绍，并对每一种网关协议给出了进行编程实践的案例指导。

5.7 课后习题

1. 知识点考查

(1) 典型的物联网网关协议有哪些？它们的特点是什么？

(2) MQTT 在传输层采用哪种协议，为什么？请仿照 MQTT 客户端的创建方法，通过查找资料，尝试建立简单的 MQTT 服务器。

(3) HTTPS 中的 S 指什么？它与 HTTP 主要有什么区别？

(4) LwM2M 主要是针对什么应用场景设计的?

(5) Modbus 协议采用了什么架构?

2. 拓展阅读

[1] 于海飞,张爱军. 基于 MQTT 的多协议物联网网关设计与实现[J]. 国外电子测量技术,2019,38(11): 45-51.

[2] 曾灶荣,陈德基,肖杨. 基于区块链和 MQTT 的物联网通信协议[J]. 电子技术,2022,51(5): 15-19.

[3] 叶欣,陈文艺,赵健. 基于 Matlab 物联网网关的 Modbus 协议实现[J]. 测控技术,2013,32(2): 77-80.