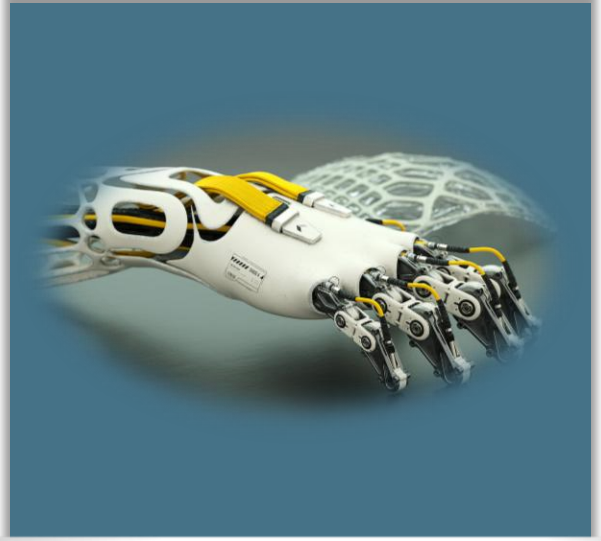
智能感知与物联网

授课人：王闻博

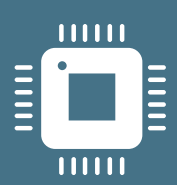
Email: wenbo_wang@kust.edu.cn

昆明理工大学 机电工程学院

2026年5月08日-5月20日

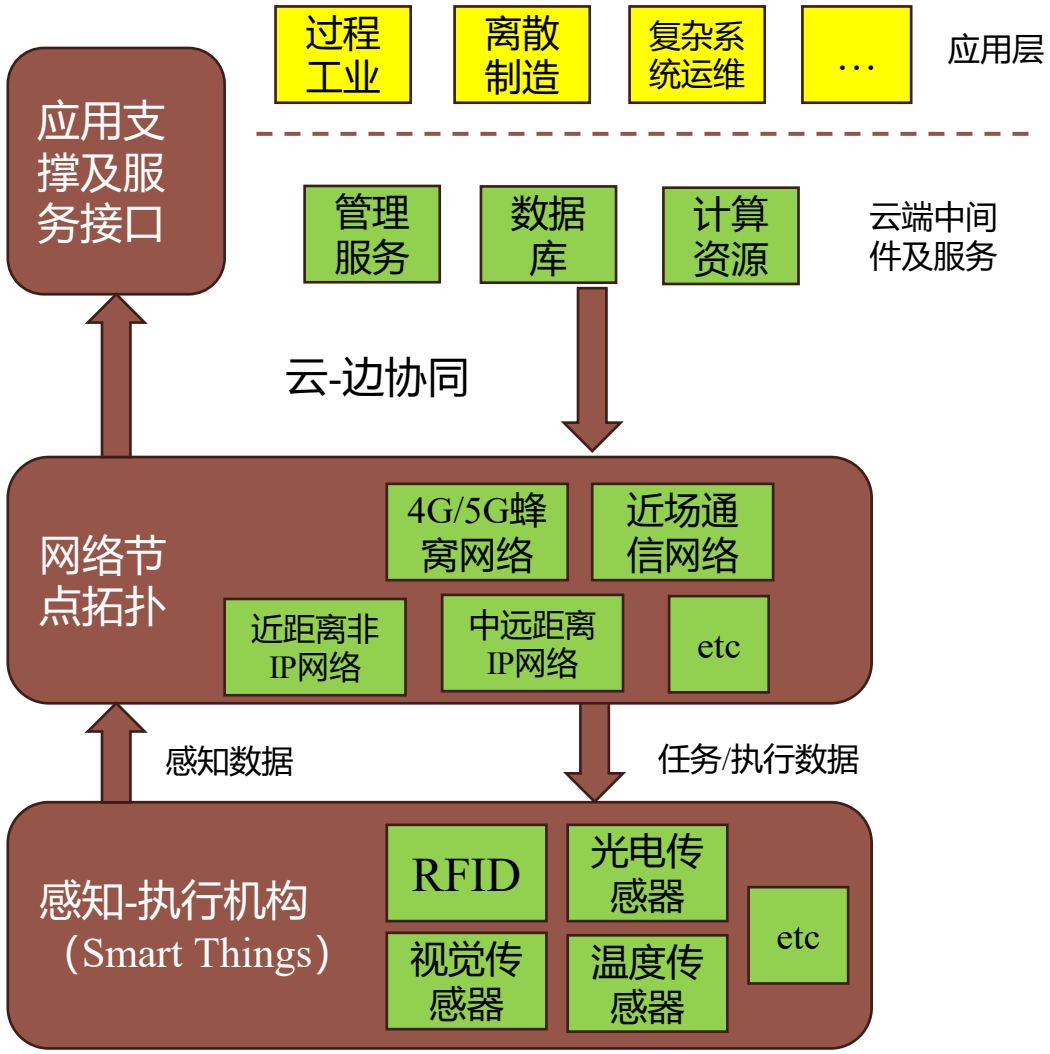


第五章
基于云的计算与决策




前章回顾：云计算在中的物联网的位置和角色

物理-软件实体



协议层抽象

数据和信息层：管理系统及数据库、智能处理、云计算，etc…

服务驱动层（中间件）

IoT接入网络层

通信层：IP无线网络、非IP无线网络、有线网络，etc…

设备和资源层：传感器、电源管理，etc…

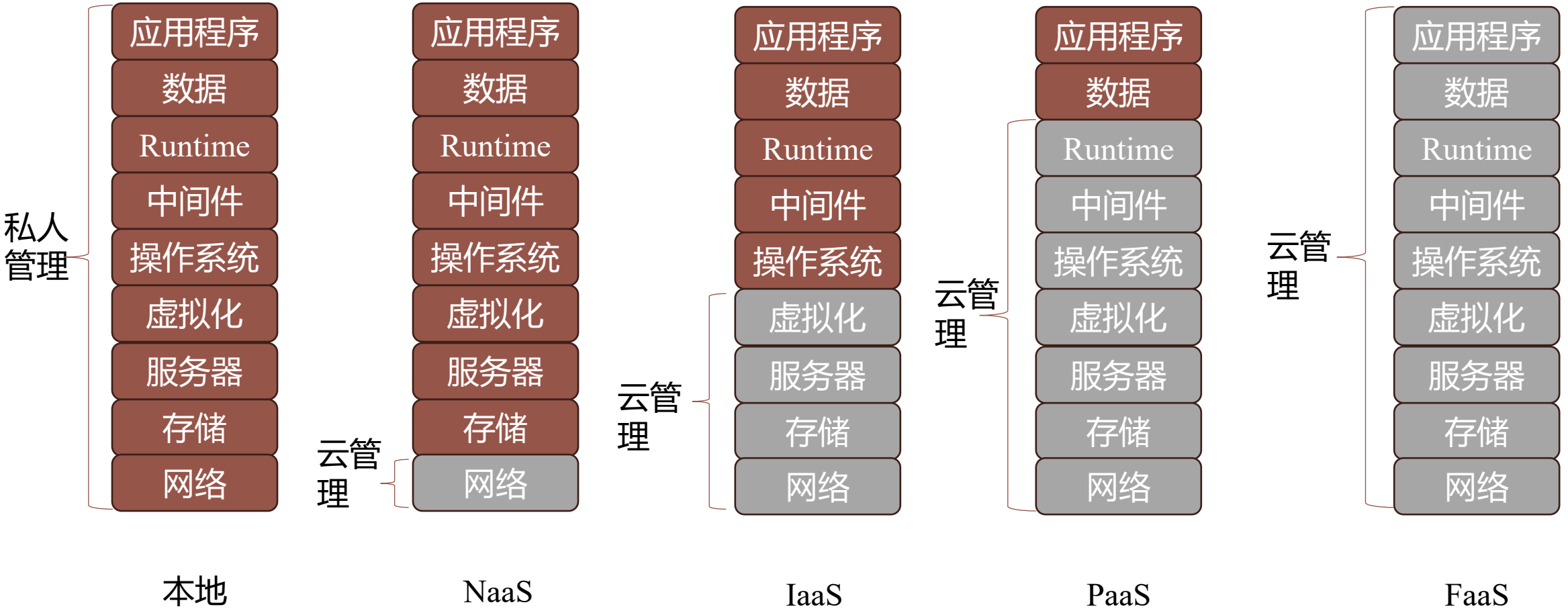
“云”的定义

- 从用户角度而言
  - 一种按用量付费的“资源服务”模式，计算发生在远端网络平台而非本地主机上；
  - 用户通过资源配置UI，以“个体-云资源”的点对点交互形式，即可从网络访问获得便捷可用的计算、服务器（主机）、存储、应用软件、服务等资源；
  - 用户通过配置，从资源共享池中获得相应尺度的资源服务，而无需了解底层架构。
- 从云服务提供商角度而言
  - 云计算是一种大规模的、分布式的、并行的计算资源组织架构；
  - 云计算基于虚拟化（Virtualization）技术，通过资源切片（Slicing）的形式提供集中式的通用资源的弹性配置和释放的功能；
  - 云计算服务应当具有高扩展性、高可靠性、廉价性，这些特性通常依托负载均衡、热备份冗余、资源自动化管理（效用计算，Utility Computing）等技术实现。

云计算的服务分类

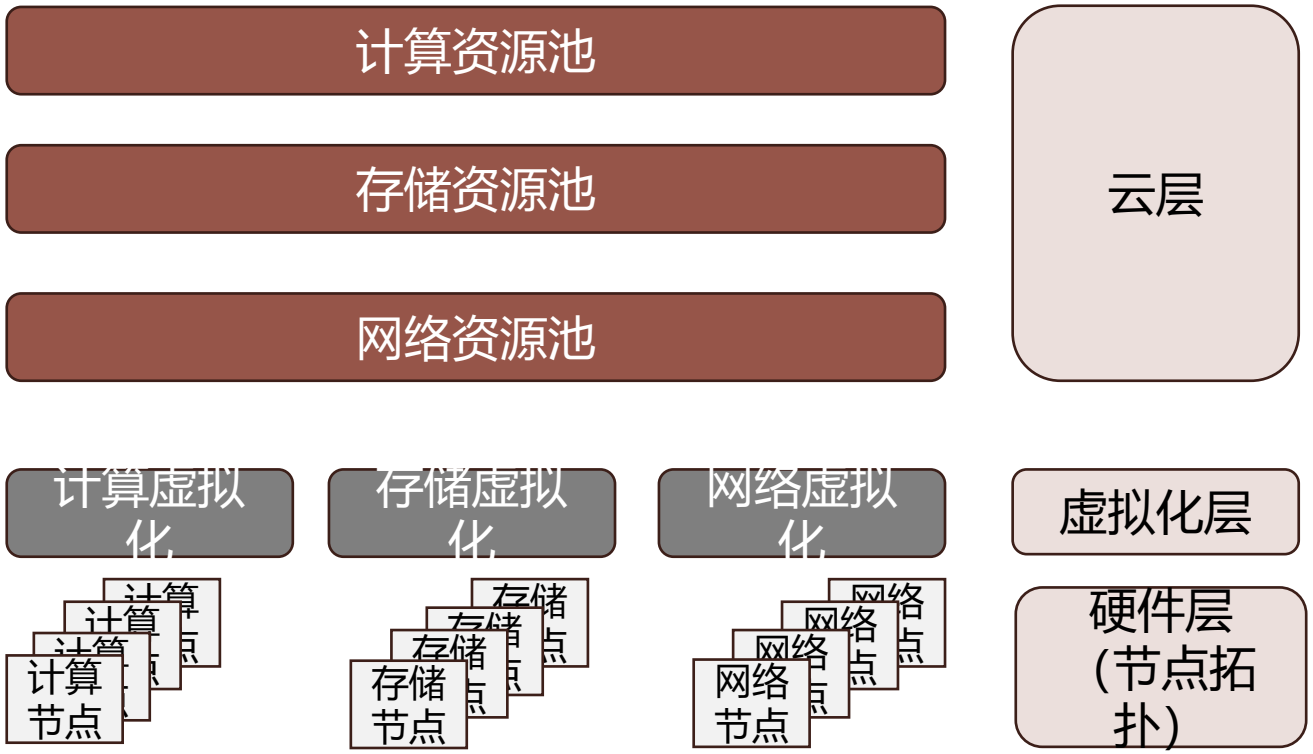
- **网络即服务（Networking as a Service, NaaS）**
  - 通过网络虚拟化技术（Network Function Virtualization），将网络控制层与网络数据层，以及网络硬件层解耦，在数据层以上，提供部署于云端的**底层网络功能**，如虚拟交换机、虚拟防火墙和虚拟中间件等，典型服务如Amazon CloudFront。
- **基础设施即服务（Infrastructure as a Service, IaaS）**
  - 云端提供可扩展的**底层硬件**（计算、存储等）服务，并提供软件框架构建客户端虚拟机。典型的IaaS如Amazon的AWS。
- **平台即服务（Platform as a Service, PaaS）**
  - 云端提供和维护**底层硬件和底层软件**功能，如操作系统和开发工具等应用运行环境。用户使用PaaS托管他们的私有应用程序。典型服务如微软的Azure云。
- **软件即服务（Software as a Service, SaaS）**
  - 供应商通过瘦客户端、Web浏览器向终端用户提供自身应用程序及服务。供应商提供应用表示（UI）、应用管理和应用服务（如在线电子邮箱）。典型的SaaS如Microsoft 365，金山WPS、Google App，Dropbox等。
- **功能即服务（Function as a Service, FaaS）**
  - 即“无服务计算”框架，无需配置和管理服务器而在云中直接运行代码。典型服务如阿里云函数计算（Function Compute）。

不同云模型管理间的差异



注：Runtime即运行时，提供虚拟机（如Java的JVM）、垃圾回收和即时编译功能，通过抽象化硬件差异实现程序的一次编写，多平台运行。

云计算的一般部署结构



Kubernetes：一个开源PaaS云服务搭建框架

- 提供容器（Docker）编排服务引擎。
- 提供可伸缩集群（由Master和Worker节点组成）管理服务。

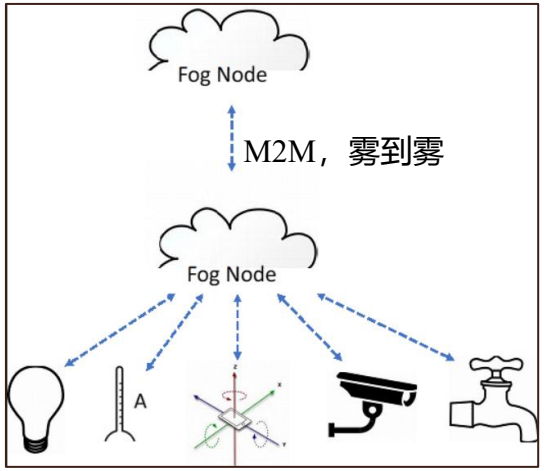
Kubernetes组件：

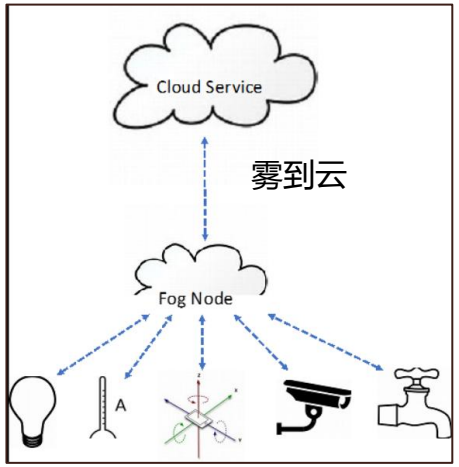
- 使用容器集合（称为Pod）管理容器化应用（共享硬件资源）。
- 通过Kube-Controller Manager组件协调和管理Kubernetes集群的资源，执行自动扩展、故障检测等工作。
- 通过Kube-Scheduler组件完成集群间资源调度和配置Pod的部署位置。
- 通过etcd组件提供分布式配置（键值）存储和管理，存储集群状态与配置信息。
- 通过Kube-API Server组件提供授权、访问控制注册、管理请求处理等服务，并写入etcd。

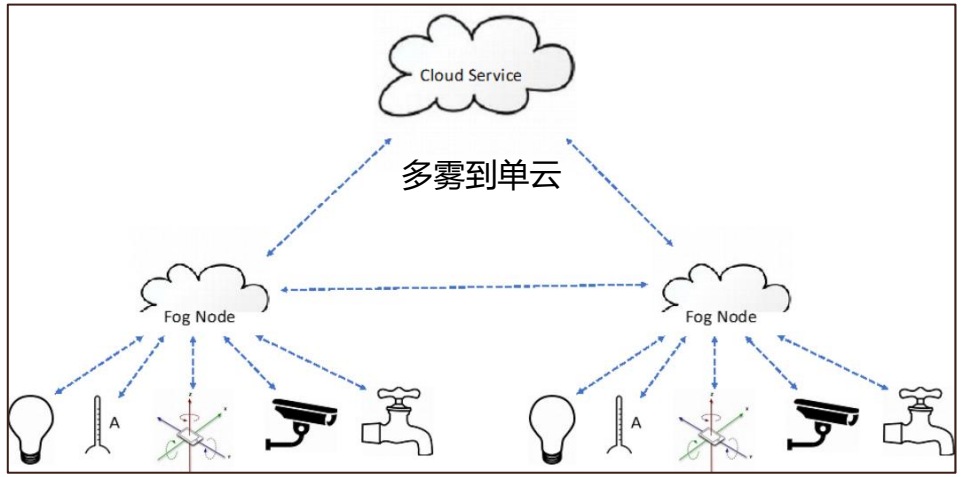
雾计算的定义和网络拓扑

（回顾）边缘节点是IoT网络中靠近数据生成位置的终端设备，雾节点处于边缘与云计算中心之间，通常部署在网络基础设施（如网关、路由器、基站或本地服务器）中。

- 雾计算
  - 对比边缘设备不参与云基础架构，雾计算的组织来自雾节点与其他雾节点或虚拟承载（Overlay）云服务共享计算框架的API和通信标准。雾计算可以看做是靠近传感器的云计算。
- 雾网络和雾拓扑
  - 雾网络的分布式计算结构采用云计算范式。其最大的特点是允许分层结构设计。





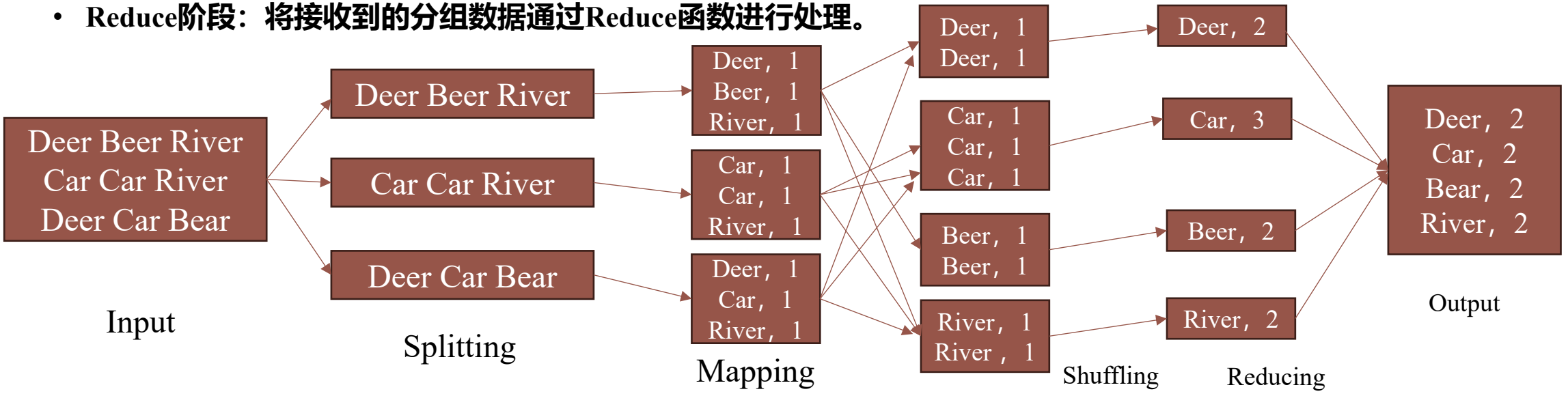


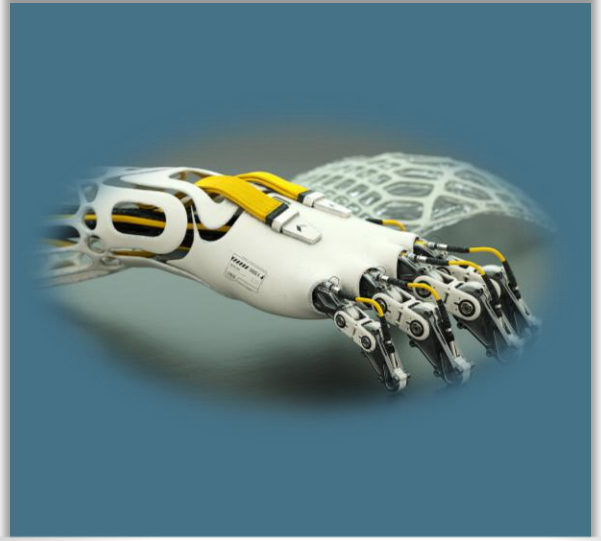
雾计算的核心功能

- **雾计算的功能核心**
  - **中间层处理**：在靠近数据源的网络边缘侧提供计算、存储和网络服务，但覆盖范围比单一“边缘节点”更广（例如使用局域网网关覆盖一个园区、多个工厂或城市区域）。
  - **任务协同**：对来自多个边缘节点的数据进行汇总、预处理或复杂分析，再将关键结果上传至云端。
  - **动态扩展性**：可根据需求灵活扩展，支持分布式协作（如多个雾节点形成“雾网络”）。
- **雾计算的典型应用场景**
  - 智慧花卉大棚：边缘节点直接控制灌溉阀门（实时响应土壤湿度监测信号），雾节点汇聚并分析大棚内多个区域的水肥、光照、温度等数据并预测病虫害。
  - 车联网：车载边缘设备处理紧急制动，雾节点（**路边单元，Roadside Unit**）协调多车辆避撞路径。
- **与边缘节点任务的对比：**
  - 边缘计算处理高度本地化、低延迟任务（如设备控制），雾计算处理跨边缘节点的协作任务（如数据融合）。

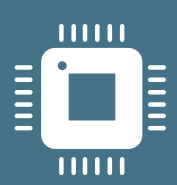
面向云（雾）端的大规模计算框架

- **MapReduce框架**
  - **一种弹性计算编程模式，用于处理超大规模数据集（大于1TB）的作业调度。**
  - **MapReduce通过将一个大数据处理作业拆分为部署在多个节点上的多个小作业，将单一作业映射到并行作业的节点上。**
- **MapReduce的处理阶段**
  - **Map阶段：对输入分片（Input Split），并解析成Key-Value对。**
  - **Shuffle阶段：对Map输出的Key-Value对进行分区处理，保证同一个key的数据发送到同一个Reduce。**
  - **Reduce阶段：将接收到的分组数据通过Reduce函数进行处理。**





第五章
基于云的计算与决策

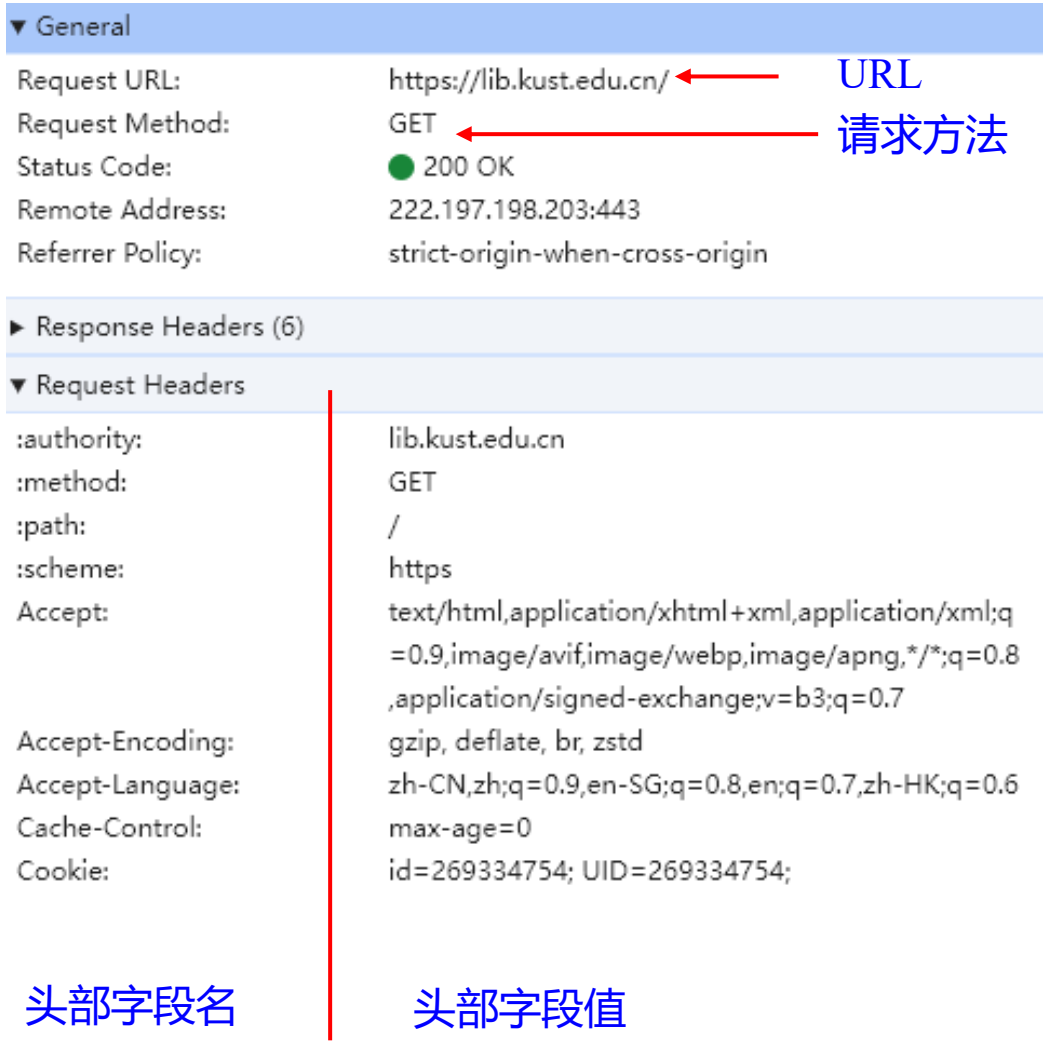



数据的准备：从边缘端到云端的数据传输协议解析和格式定义

- 基于TCP/IP的应用层协议：HTTP（超文本传输）协议
  - 基于应用程序Client-Service模式：“客户端”（应用程序）主动向“服务器”（应用程序）发起连接和数据请求（如HTML、XML和JSON等格式的数据）。
  - 服务器不能主动向客户端推送数据。
  - 使用统一资源标识符（Uniform Resource Identifier，URI）传输数据和建立连接。
  - HTTP是一种基于TCP/IP的无连接、无状态协议。
    - 无连接：每次连接只处理一个请求；服务器处理完客户端的请求，并收到应答后，即断开连接。
    - 无状态：对事物处理上下文没有记忆。
  - 客户端通过不同的请求方法（GET、POST、PUT、DELETE等）表明对Request-URI指定的资源的不同操作。

HTTP协议客户端请求消息报文

- **HTTP请求的8种方法**
  - **(1) OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。**
  - **(2) GET：向特定的资源发出请求。**
  - **(3) HEAD：向服务器索要与GET请求相一致的响应，在不传输响应内容的情况下，获取包含在响应消息头中的元信息。**
  - **(4) POST：向指定资源位置（URL）提交数据进行处理请求（如提交表单或者上传文件）。数据包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。**



HTTP协议客户端请求消息报文（续）

- HTTP请求的8种方法（续）
  - (5) PUT：向指定资源位置上传其最新内容
    - PUT由客户端指定URI，对已有资源进行更新。
    - POST由服务器决定新资源的URI，对资源进行创建/添加操作。
  - (6) DELETE：请求服务器删除Request-URI 所标识的资源。
  - (7) TRACE：回显服务器收到的请求，主要用于测试或诊断。
  - (8) CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

- 客户端代码举例（Python的requests库）

```
7   import requests
8
9   # 发送GET请求
10  search_query = "物联网数据解析"
11
12  # 指定URL
13  url = f"https://www.baidu.com/s?wd={search_query}"
14
15  print('发送GET请求 ')
16  response = requests.get(url)
17  print('回复状态码: ', response.status_code,
18        ', 状态码原因: ', response.reason)
19  print('服务器回复: ')
20  print(response.content)
21
22  print('收到资源文本（HTML）解析: ')
23  print(response.text)
```

HTTP协议服务器响应消息报文格式

- 以GET昆工图书馆主页地址为例

```
< HTTP/1.1 200 OK
< Server: RUMS
< Date: Sat, 04 May 2024 08:20:46 GMT
< Content-Type: text/html;charset=UTF-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: JSESSIONID=052F94B71BCC70D856553C2ECE7082DB; Path=/; HttpOnly
< Vary: Accept-Encoding
< Content-Language: zh-CN
<
<!DOCTYPE html><html><head>



<title>昆明理工大学图书馆</title><META Name="keywords" Content="昆明理工大学
图书馆" />


<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<meta name="viewport" content="width=device-width, initial-scale=1.0,minimum-
scale=1.0, maximum-scale=1.0">
<link href="css/style.css" rel="stylesheet" type="text/css"><link href="css/p
ublice.css" rel="stylesheet" type="text/css"><script type="text/javascript" s
rc="js/jquery.min.js"></script><script type="text/javascript" src="js/SuperSl
iden.js"></script><script type="text/javascript" src="js/jquery.soChange.js">
```

(1) 状态行

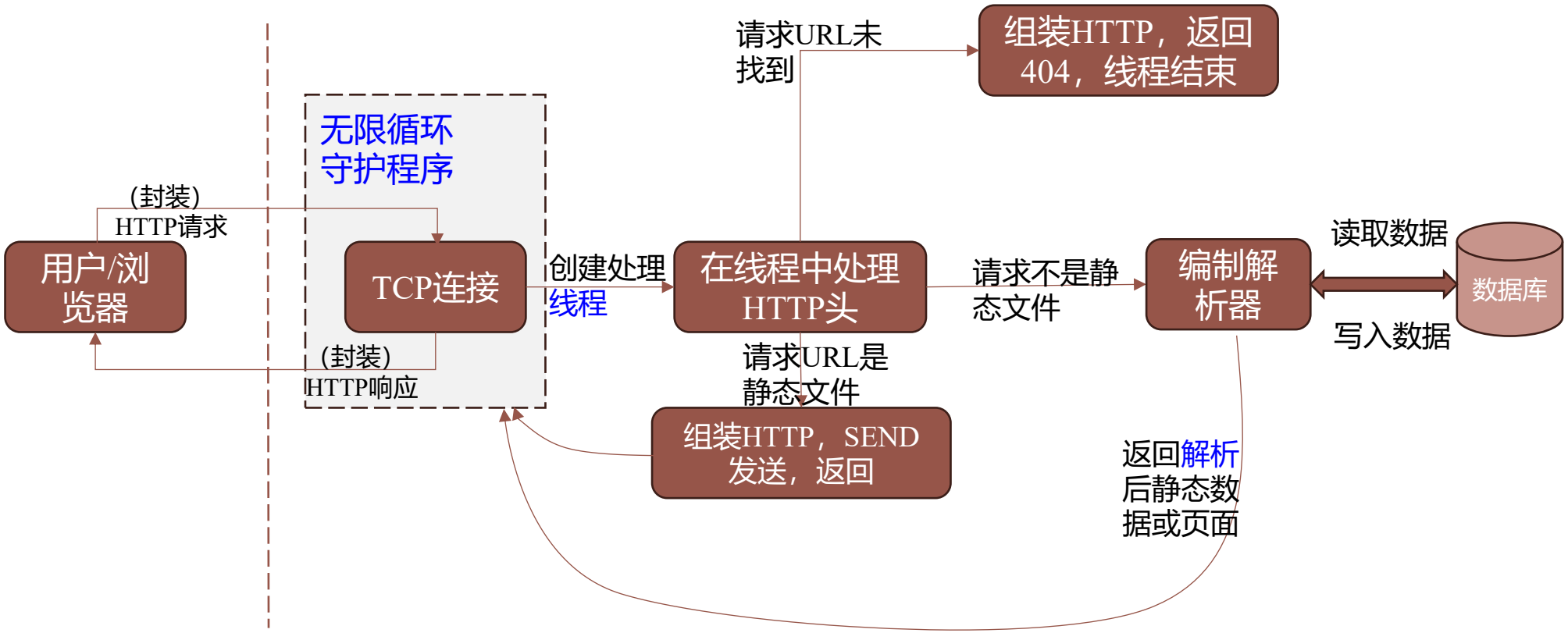
(2) 消息报头

(3) 空行

(4) 响应正文

HTTP服务器端应用的设计

- **HTTP服务端（Web服务端）的常见框架**
  - **Apache Http Server，Nginx，Django等。**
- **HTTP服务端的逻辑结构**



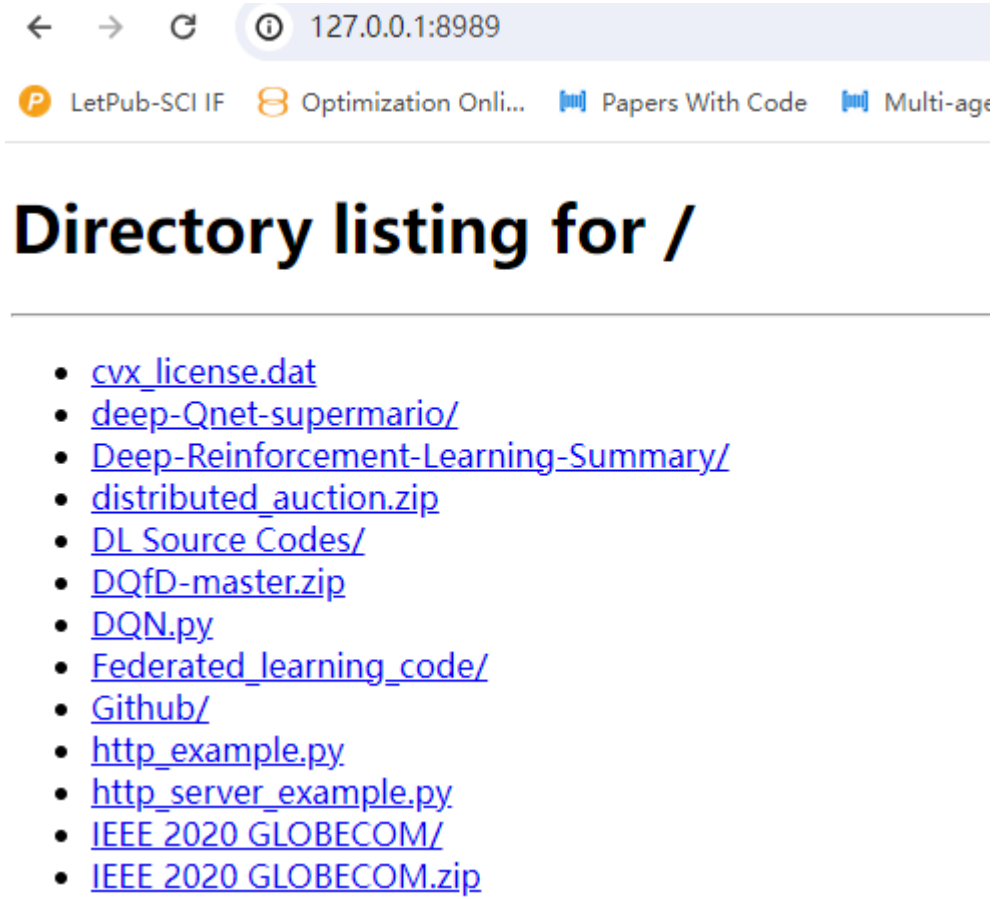
HTTP服务器端应用程序的简例

- 服务端代码举例（Python的http库）
  - 在虚拟地址的8989端口创建简单HTTP服务
  - 此服务器将服务于当前目录下的文件，但未实现URI解析功能

```
8   import http.server
9   import socketserver
10
11  # 设置IP和端口号
12  PORT = 8989
13  server_address = ('127.0.0.1', PORT)
14
15  Handler = http.server.SimpleHTTPRequestHandler
16
17  with socketserver.TCPServer(server_address, Handler) as httpd:
18      print("Serving at port", PORT)
19      httpd.serve_forever()
```

问题：在客户端如何通过POST对服务端发送数据，并在服务端进行响应，在当前根目录生成记录文件？

- 简易HTTP服务器部署在教师个人开发目录下的Web浏览器响应结果



HTTP协议下的客户端-服务端数据推送

- 客户端示例代码（Python）

```python
@author: wbwan
"""

import requests
import json

# 准备要发送的数据
sensor_data = {
    'temperature': '25',
    'key2': 'value2'
}
# 将数据转换为JSON格式
json_data = json.dumps(sensor_data)

# 发送POST请求到服务器
response = requests.post('http://127.0.0.1:8989',
                         data=json_data, headers={'Content-Type': 'application/json'})

# 输出服务器的响应内容
print(response.text)
```

- 服务端示例代码（Python，部分）

```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

class SensorDataHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        # 解析请求的内容长度
        content_length = int(self.headers['Content-Length'])
        # 读取POST过来的数据
        post_data = self.rfile.read(content_length)
        # 将字节数据解码为字符串
        post_data_str = post_data.decode('utf-8')
        # 解析JSON数据（如果客户端发送的是JSON）
        try:
            data = json.loads(post_data_str)
        except json.JSONDecodeError:
            self.send_error(400, "Bad Request: Invalid JSON format")
            return

        # 在这里处理数据，例如记录到文件
        with open('tmp_sensor_data.txt', 'a') as record_file:
            record_file.write(f"{data}\n")

        # 发送响应状态码
        self.send_response(200)
        # 发送响应头
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        # 发送响应内容（可选）
        response = json.dumps({'message': 'Data received and recorded.'})
        self.wfile.write(bytes(response, "utf8"))
        return

# 设置服务器地址和端口
server_address = ('127.0.0.1', 8989)
httpd = HTTPServer(server_address, SensorDataHTTPRequestHandler)
print("Server started on port 8080...")
httpd.serve_forever()
```

HTTP协议与IoT数据通信

- **HTTP协议的优势**
  - **开发便捷性：快速原型设计阶段使用基于Restful API的HTTP原型开发，以简化测试。**
- **HTTP协议在IoT场景下的劣势**
  - **高开销的数据包头部（Header Overhead）**
    - **HTTP请求/响应的头部信息庞大（如Cookie、User-Agent等字段），而IoT设备通常仅需传输少量数据（如传感器读数）。**
  - **基于传输层TCP协议的可靠性连接需要付出额外的资源代价**
    - **HTTP依赖TCP协议，需三次握手建立连接，且维护连接状态会消耗资源，导致延迟高、能耗高。**
  - **请求-响应模式不灵活**
    - **HTTP严格遵循客户端主动请求-服务器响应的模式。而IoT场景通常需要数据实时推送（如警报通知）、多对多通信（如设备群组协同）等。**

（插曲）物联网数据封装形式简介：JSON

JSON：JavaScript Object Notation

- 一种轻量级数据交换格式（人类可阅读），支持UTF-8编码。
- 解析库：json（Python），cJSON（C/C++ API）。
- 语法特征：
  - 数据在“键:值”对中表示，由逗号隔开
    - 例子：“flag”: true, “name”: ”1号喷涂机器人”。
  - 中括号保存数组
    - 例子：“factory_location”：[{“location”: “昆明”}, {“location”: “曲靖”}]。
  - 大括号保存对象
    - 例子：{“name”: “AGV1”, “battery_level”: 50, “coordinates”:{“x”: 100, “y”: 150} }

物联网数据封装形式简介：JSON（续）

JSON：JavaScript Object Notation

- **面向对象的数据表示：JSON可以直接表示对象（在JavaScript等语言中通常称为“对象”或“字典”），其中对象的属性以键值对的形式存储。**
- **嵌套结构：JSON支持嵌套的对象和数组结构，这可以很好地表示复杂的数据关系。例如，一个对象可以包含另一个对象或数组作为其属性的值。**

示例文件：

```
{
  "name": "张三",
  "age": 30,
  "department": "产线运维",
  "email": "zhangsan@example.com",
  "isManager": false,
  "project": ["产线数字孪生", "MES系统集成", "IoT中间件开发"],
  "project_state": {
    "产线数字孪生": 30,
    "MES系统集成": 75,
    "IoT中间件开发": 45
  },
  "projects": [
    {
      "projectName": "产线数字孪生",
      "startDate": "2024-01-01",
      "endDate": "2024-011-30",
      "status": "进行中"
    },
    {
      "projectName": "MES系统集成",
      "startDate": "2023-11-01",
      "endDate": "2024-7-31",
      "status": "进行中"
    }
  ]
}
```

物联网数据封装形式简介：XML

XML：eXtensible Markup Language

- 使用标记结构（树形结构）来表示数据项，是一种标记语言，除UTF-8编码外还支持其他编码。
- 解析库：xml（Python），TinyXML （C/C++ API）。
- 语法特征：
  - 元素：由开始标签、结束标签和元素内容组成。元素内容可能包含其他元素、文本或实体引用。
  - 属性：是元素的组成部分（放在标签中间）。属性由名称和值组成，名称和值之间用等号（=）连接，且属性值必须加引号（单引号或双引号均可）。
  - 命名空间：命名空间用于避免元素名称冲突，通过前缀来标识。命名空间声明通常在根元素上，可以使用xmlns属性定义。

物联网数据封装形式简介：XML（续）

XML的简单示例

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <bk:bookstore xmlns:bk="http://example.com/bookstore">
3       <bk:book>
4           <bk:title>XML Developer's Guide</bk:title>
5           <bk:author>Gamalda, Erik T.</bk:author>
6           <bk:isbn>0-07-139246-X</bk:isbn>
7           <bk:price>44.95</bk:price>
8           <bk:edition>1</bk:edition>
9       </bk:book>
10      <bk:book>
11          <bk:title>Midnight Rain</bk:title>
12          <bk:author>Pareto, N.C.</bk:author>
13          <bk:isbn>0-9658536-9-0</bk:isbn>
14          <bk:price>5.95</bk:price>
15          <bk:edition>1</bk:edition>
16      </bk:book>
17  </bk:bookstore>
```

文件说明：

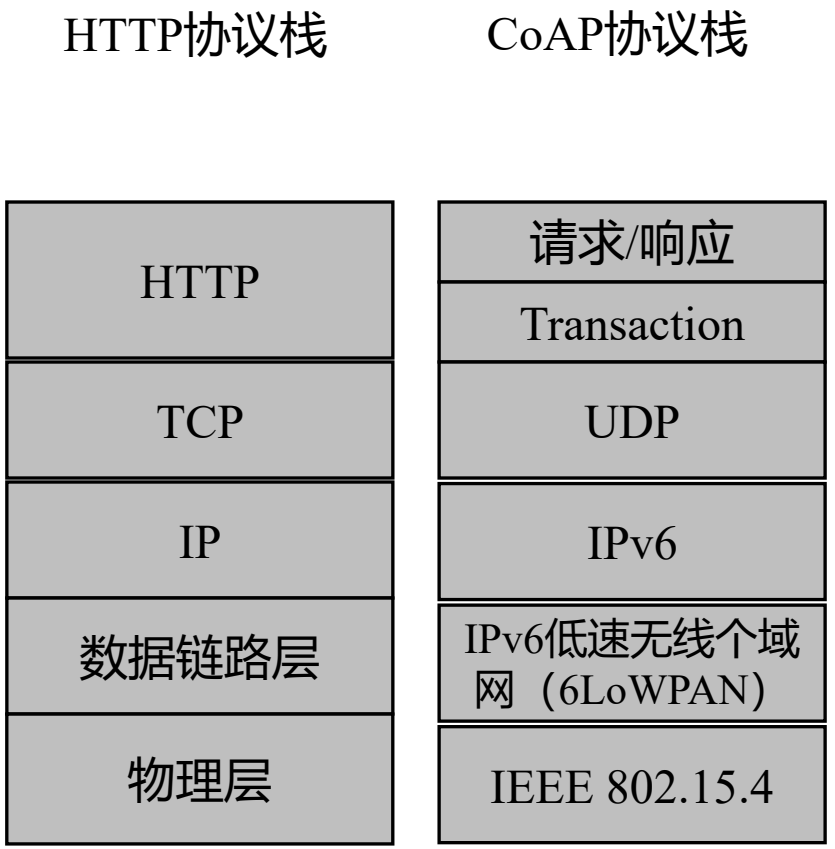
- 行1：XML文件声明，用于定义XML的版本和编码方式。
- 行2：定义了一个命名空间bk，它关联到URI http://example.com/bookstore。这个URI通常是一个标识符，并不一定指向实际的网络资源。前缀“bk”表示该元素或属性属于这个命名空间。
- 行3-9：定义第一个<bk:book>元素，它包含多个子元素，如<bk:title>, <bk:author>, <bk:year>, <bk:price>等。
- 行10-16：定义第二个<bk:book>子元素。

基于TCP/IP的应用层协议：CoAP（受限应用协议）

- CoAP：Constrained Application Protocol，受限应用协议
  - 是简化了HTTP协议的RESTful API（参见附录1）。与HTTP共享上下文、语法和用法，允许代理桥接到HTTP对话。
  - 支持观察者模式：允许客户端订阅服务端资源，并在资源变化时收到通知。
  - 异步通信：通过不同的消息模式（CON、NON、ACK等四种）支持请求-响应、订阅-发布等交互模式。
  - 是一种建立在UDP上的无连接、无状态协议。
    - 无连接：每次连接只处理一个请求；服务器处理完客户端的请求，并收到应答后，即断开连接。
    - 无状态：每个请求都是独立的，对事物处理上下文没有记忆。
  - 仍然基于典型的Web架构
    - 使用统一资源标识符（Uniform Resource Identifier，URI）传输数据和建立连接，典型URI形式如coap://host[:port]/[path][?query]。
    - 客户端支持不同的请求方法（GET、POST、PUT、DELETE），表明对Request-URI指定的资源的不同操作。

CoAP（受限应用协议，续）：CoAP与HTTP的比较

- 协议栈不同：CoAP基于UDP，有请求/响应层和事务层两个基本层；HTTP基于TCP
  - 请求/响应层：负责接收基于RESTful API的查询。REST查询以CON（可确认消息）或NON（不可确认消息）为载体，REST响应以ACK消息为载体。
  - 事务（Transaction）层：使用四种消息之一处理终端消息交换。支持多播和拥塞控制。
- CoAP兼容RESTful语义，但只支持一下四种请求方式： POST、GET、PUT、DELETE。
- CoAP的效率（功耗、transaction字节消耗）可达到HTTP的64倍
  - CoAP报头开销最小为4字节，HTTP通常有几百字节。
  - CoAP由于基于UDP，其连接、传输开销以及延迟都要远小于需要维持TCP连接的HTTP协议。



CoAP系统中的参与者

CoAP终端：CoAP消息的来源于目的地

- 客户端（Clients）：资源请求的发起端，响应的目的端。通常是资源受限的设备，如传感器节点等。客户端向服务器发送请求（GET/获取、POST/创建、PUT/更新和DELETE/删除），并接收服务器响应。
- 服务器（Servers）：资源请求的目的端，响应的发起端。服务器负责处理客户端的请求，并返回相应的响应。在IoT场景中，服务器通常是一个云服务或边缘Sink节点，用于收集、存储和分析来自各个客户端的数据。
- 代理（Proxies）和中介（Intermediary）：由客户端委托代理（或中介）来处理和转发客户端与服务器之间的通信。代理可以作为中间节点，接收客户端的请求，然后将其转发给服务器，并将服务器的响应返回给客户端。中介是同时充当服务器和中介，并指向源服务器的终端。中介提供更广泛的功能，如数据转换、协议转换、安全性增强等。代理是一种中介。
- 源服务器：给定资源所在的服务器。
- 观察/监视器（Observer）：客户端通过向服务器发送带有Observe选项的GET请求来订阅某种资源。一旦订阅的资源状态发生变化，服务器会主动通知所有订阅了该资源的客户端。观察器模式使得客户端能够被动地接收服务器推送的数据，而无需不断轮询服务器。
  - 比较：在HTTP中，客户端需要定期向服务器发送请求以获取最新的资源状态。

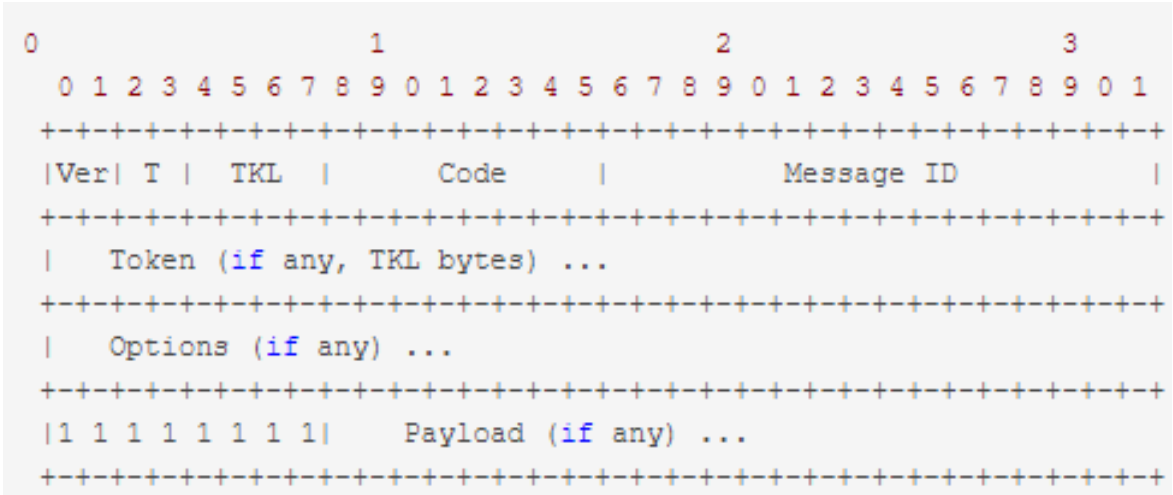
CoAP的消息种类

CoAP的客户端通过消息种类确定连接类型

- CoAP的消息类型定义是对UDP协议的可靠性补偿。
- **可确认（CON）消息**：Confirmable，来自客户端，要求在一个随机ACK_TIMEOUT时间间隔中接收回复的ACK确认消息。
- **不可确认（NON）**：Non-Confirmable，来自客户端，是不需要服务端ACK的广播消息或即发即弃（fire-and-forget）消息，适合高频次传感器数据上报。
- **确认（ACK）消息**：Acknowledgement， 来自服务器，是对相应CON消息的确认响应。ACK消息可以附加在其他数据上传输。
- **重置（RST）消息**：Reset，来自服务器，是对相应CON消息的反馈。指示已经收到CON消息，但上下文丢失。 RST消息可以附加在其他数据上传输。

CoAP的消息格式

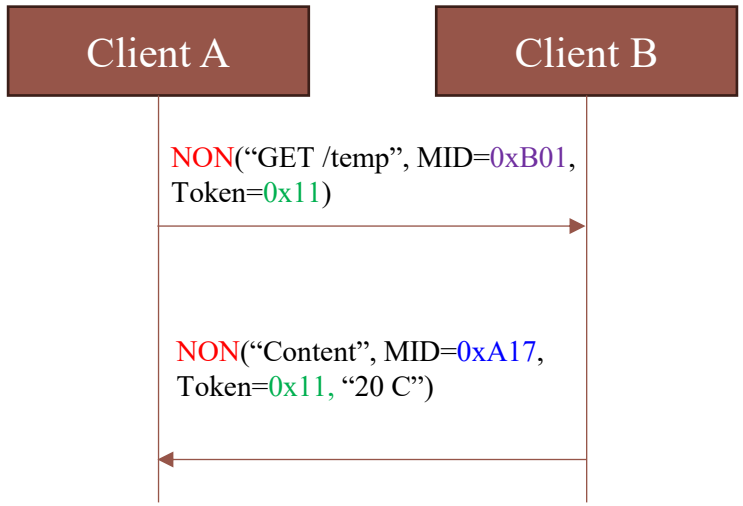
- **CoAP的消息报文格式（报头共4字节）**
  - **Ver：版本，2bit位。**
  - **T：消息种类， 2bit位（CON：0，NON：1，ACK：2，RST：3）。**
  - **TKL：令牌长度（ToKen Length），指示可变长度令牌字段（Token）的长度（0-8bit位）。**
  - **Code：代码（8bit位），分为（a）请求码——指示请求方法（GET，POST、PUT，DELETE）；（b）响应码——指示请求的处理结果（成功、客户端错误、服务器错误）。**
  - **Message ID（MID）：消息ID（16位），用于检测消息重复，并将确认/重置类型的消息与可确认/不可确认类型的消息进行匹配。同一会话中ID固定，会话结束后回收复用。**
  - **Token：长度由TKL决定（0-8bit位），用于标识请求和响应之间的关联：客户端在请求中携带一个Token，服务器在响应中必须原样返回该Token，使客户端能将响应与对应请求关联。**
  - **Options：传递额外参数和附加信息，如端口号、主机号；以及URI路径、内容类型、指定“观察者”模式、指定是否为块传输（Block选项用于大文件分片传输，如固件升级等场景。）**
  - **Payload：实际数据，如传感器信息等，以0xFF分隔符起始。**



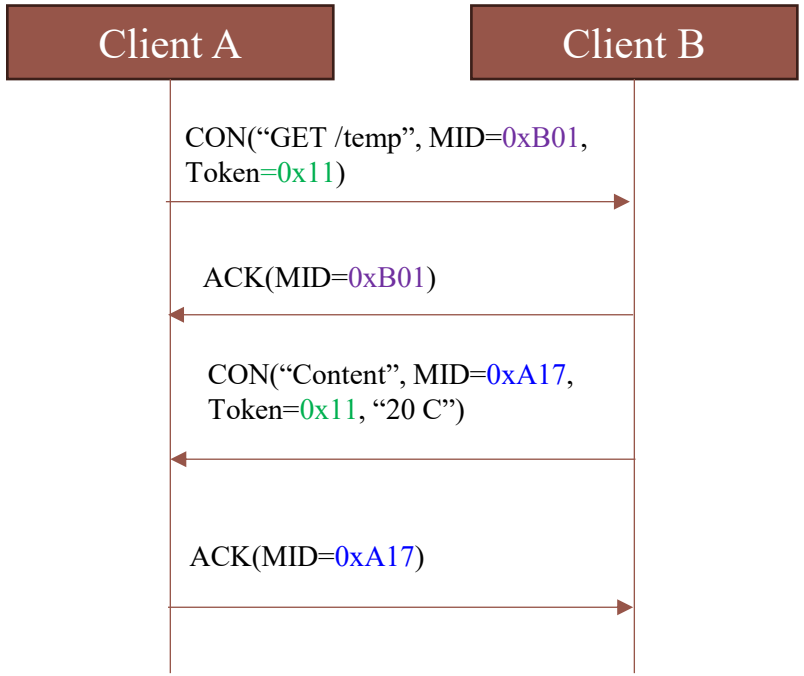
CoAP的不同消息的传递-确认过程

- **不同消息类别的传递-确认过程**

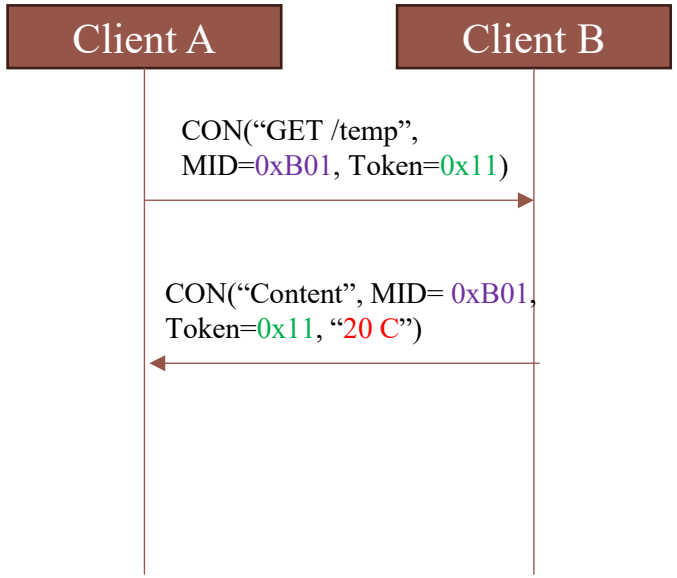
不可确认的请求和响应



可确认的请求和响应

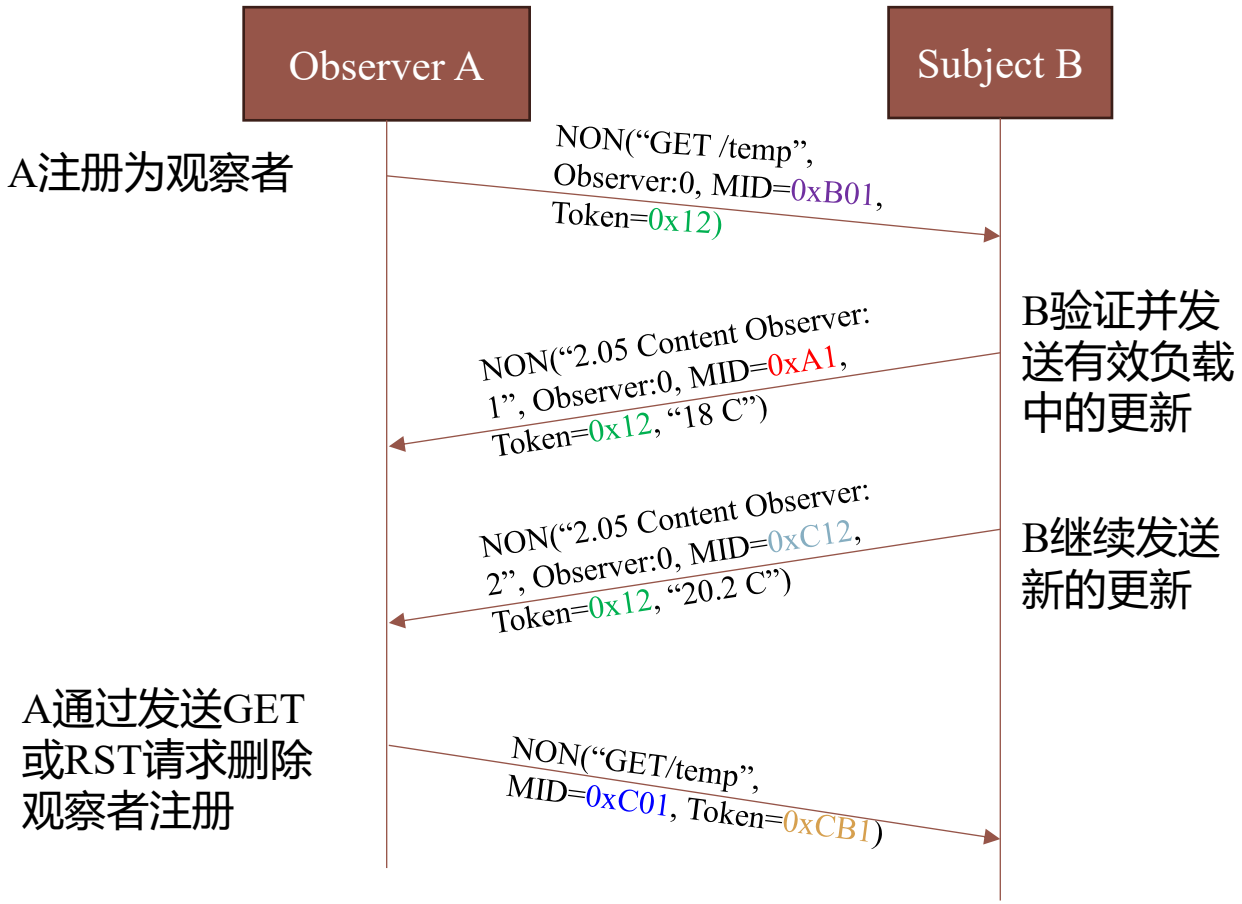


可确认的请求和响应（简化形式：Piggyback ACK with data）



- MID和Token的差异比较（见中间图）
  - MID为**会话级标识**：同一会话中固定（例如一次设备配网过程），会话结束后可回收复用；
  - Token为**逻辑事务标识**：可跨越多个消息或会话，例如订阅资源变更时，通过Token持续跟踪状态。

CoAP的订阅-发布模式：观察者（Observer）



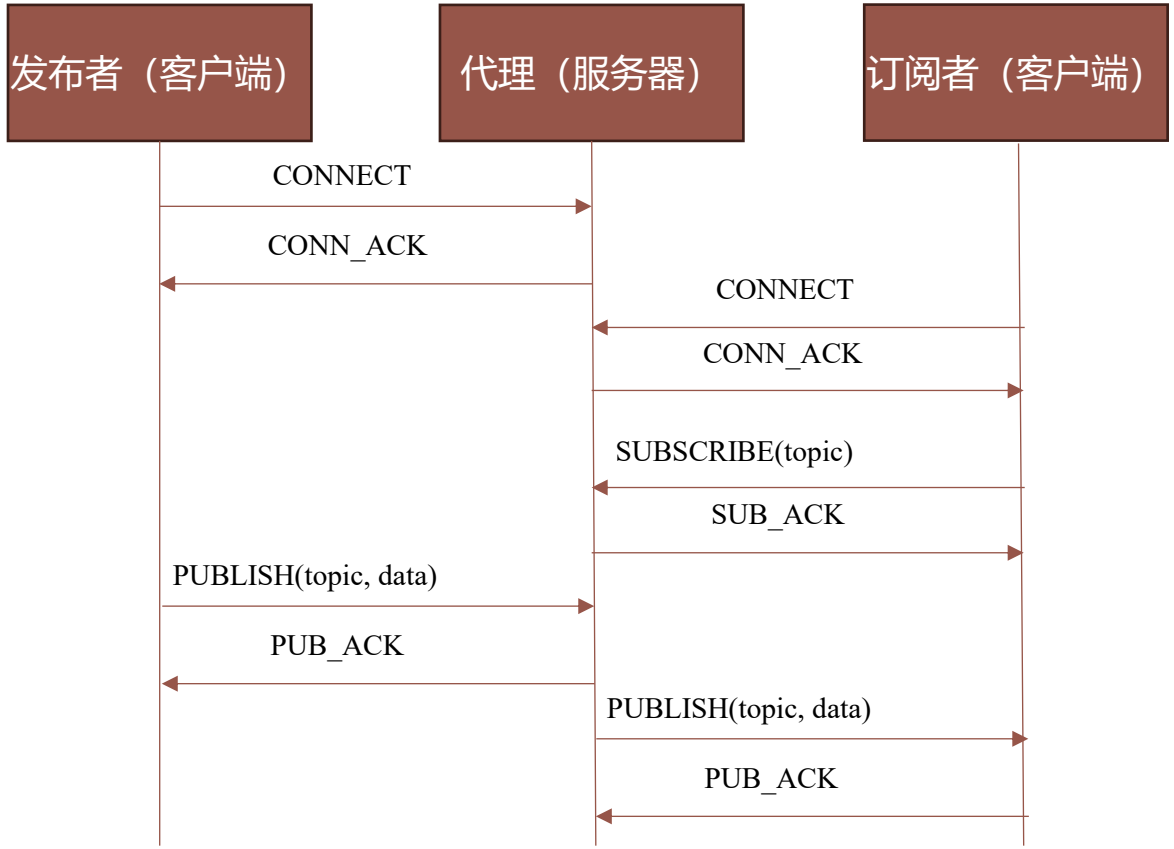
- **模式的其他相关说明**
  - **CoAP通过CON-ACK的请求-响应模式能够为UDP传输添加超时重传机制（超时重传参数：ACK_Timeout，UML时序图略）。**
  - **CoAP的响应代码（参见消息格式Code部分介绍）**
    - **2.01：创建；2.02删除；2.04：更改；2.05：内容；4.04：未找到（资源）；4.05：方法未被允许。**
  - **CoAP允许缓存模式，通过消息头中的响应代码控制是否缓存。由参数Max_Age设置消息在刷新前的缓存时间。缓存工作一般由代理（Proxy）执行。**
- **当前可用CoAP库（Python）**
  - **CoAPthon/CoAPthon3，aiocoap，pycoap等。**

基于TCP/IP的应用层协议：MQTT（消息队列遥测传输）

- MQTT（Message Queueing Telemetry Transport）协议特性：
  - 基于代理的发布/订阅模式：MQTT提供一对多的消息发布。客户端可以订阅特定的主题（或通配符主题），当有消息发布到这个主题时，所有订阅了该主题的客户端都会收到消息。
  - 基于TCP/IP网络连接：MQTT使用TCP/IP作为其传输协议，提供有序、无损、双向连接，保证了消息的可靠传输。
  - 支持QoS服务质量等级设定：MQTT支持对消息设置不同的服务质量等级（QoS 0，1，2），根据消息的重要性不同设置不同的服务质量等级。这有助于确保重要消息的可靠传递。
  - 使用will遗嘱（LWT）机制：MQTT使用will遗嘱机制来通知客户端异常断线，使得在客户端意外断开连接时，服务器能够采取相应的措施。
  - 支持心跳机制：通过心跳报文保持与服务器的连接，确保客户端与服务器之间的连接保持活跃。

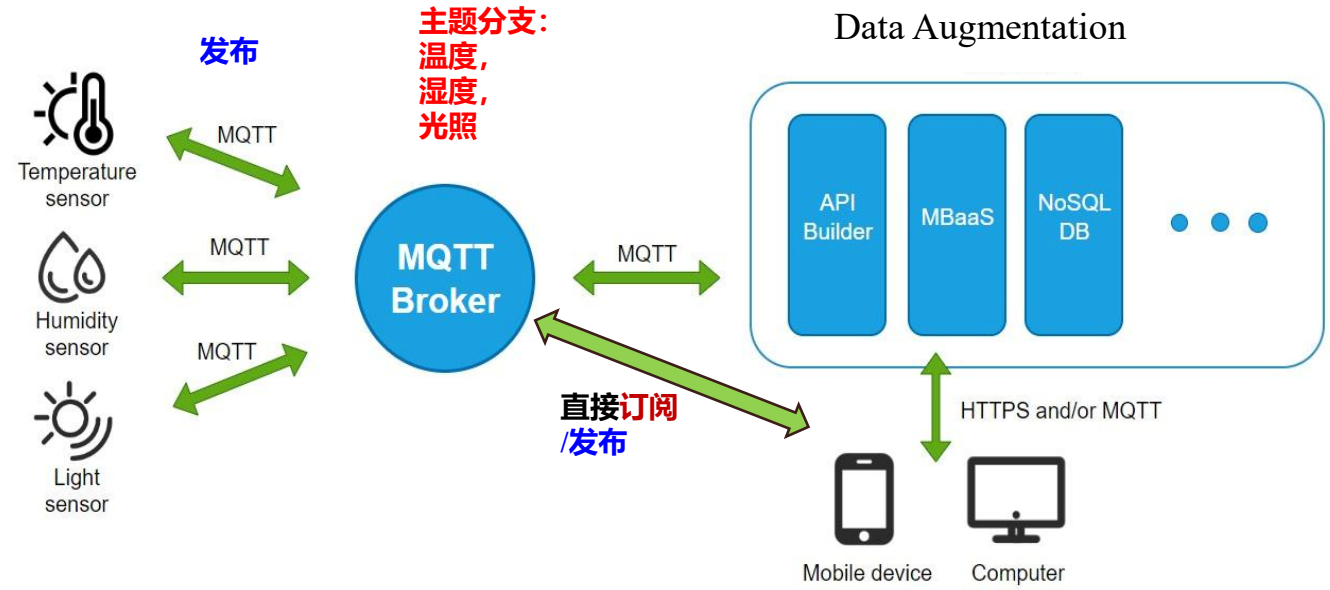
MQTT的通信参与者身份与消息模式

- **MQTT的参与者**
  - **客户端（Client）：**
    - **客户端可以是发布者（Publisher），负责发送消息；**
    - **也可以是订阅者（Subscriber），负责接收消息；或者同时兼具两者角色。**
  - **代理（Broker，服务器）：代理充当消息中转站的角色，客户端通过代理进行消息的发布和订阅。**



MQTT的消息模式

- **MQTT的消息模式：发布-订阅（pub/sub）**
  - **数据生产端（发布者）和消费者（订阅者）并不直接交互。发布者和订阅者之间通过代理进行解耦。**
    - **解耦：客户端不知道/识别任何物理标识符，如IP地址或端口。**
  - **MQTT系统是一个中心化（星型）结构。代理负责连接客户端并提供数据过滤**
    - **过滤：代理提供主题分支过滤和内容过滤；客户端也可以实现自己的数据解析和过滤器（如类型过滤）。**



MQTT的通信请求-响应方式

MQTT的不同请求消息的格式

- CONNECT（客户端到服务器）
  - 用以启动会话。
  - 重要字段：clientID (必选，其他可选), clearSession（断开连接后丢弃所有订阅和未确认消息），cleanStart（丢弃之前的任何会话并创建一个新会话），UserName, password, lastWillTopic（遗嘱消息的主题），lastWillQoS（遗嘱消息的服务质量等级），lastWillMessage（遗嘱载荷），lastWillRetain, keepalive（心跳保活时间间隔/秒）。
- CONNACK：CONNECT返回码（服务器到客户端）
  - 0-成功连接，1-5拒绝连接（原因不同）。

MQTT的不同请求消息的格式（续）

- PUBLISH（客户端-发布者到服务器）
  - 用以将数据发布到主题分支。
  - 重要字段（必选）：packetID, topicName, qos, returnFlag, payload, dupFlag。
- SUBSCRIBE（客户端-订阅者到服务器）
  - 订阅数据包的有效负载至少包含一对UTF-8编码的主题ID和QoS级别。
  - 重要字段：（必选）packetID, {topic_1, qos_1}, (可选) {topic_2, qos_2, …}。

MQTT的通信请求-响应方式（续）

其他次要消息

- **PUBACK：发布确认。**
- **PUBREC，PUBREL，PUBCOMP：与QoS-2服务等级相关（见下页）。**
  - **PUBREC（Publish Received）：**接收方（服务端或客户端）对QoS 2的PUBLISH消息的**首次确认**，表示已接收但暂未处理消息；
  - **PUBREL（Publish Release）：**送方（发布者）对PUBREC的响应，表示允许释放消息资源并进入最终确认阶段；
  - **PUBCOMP（Publish Complete）：**作用：接收方对PUBREL的**最终确认**，标志着QoS 2流程完成。
- **SUBACK：订阅确认（服务器到客户端）。**

其他次要消息（续）

- **UNSUBSCRIBE：请求取消订阅（客户端到服务器）。**
- **UNSUBACK：取消订阅确认（服务器到客户端）。**
- **PINGREQ：心跳请求（客户端到服务器）。**
- **PINGRESP：心跳响应（服务器到客户端）。**
- **DISCONNECT：中断连接（客户端到服务器）。**

MQTT的独有模式：服务质量等级、遗嘱和心跳

MQTT的服务质量等级

- **QoS-0（非保证传输，At Most Once）**
  - 最低服务等级，只发1次，不等待接收端确认，没有发送端重传。
  - 在不需要消息队列（如有线连接或有严格带宽限制）时使用。
  - 传输效率高，典型场景：传感器周期性上报环境数据（如温湿度）；日志记录等容忍偶发丢失的非关键业务；
  - 缺点：可靠性最低——网络不稳定时易丢失消息。
- **QoS-1（保证传输，At Least Once）**
  - 保证消息至少送达至接收端1次。消息可能不止重传1次（即订阅者/代理可能收到重复消息），发送端通过PUBLISH发送消息后等待接收端的PUBACK确认，接收方使用PUBACK响应确认，若超时未收到则重传。
  - 是MQTT的默认服务质量等级。资源消耗中等，因为需维护重传队列，可能增加延迟。
  - 典型场景：工业自动化中需可靠传输但可容忍冗余数据的场景。

MQTT的独有模式：服务质量等级、遗嘱和心跳（续）

MQTT的服务质量等级

- **QoS-2（对应用程序的有保证服务，Exactly Once）**
  - QoS最高等级。保证接收端只接收1次，通过唯一消息ID确保消息不丢失、不重复。
  - 握手顺序（四次）：
    - (Publisher端) PUBLISH请求 → (Broker端)
    - (Broker端) PUBREC响应（Publish Received，发布收到）→ (Publisher端)
    - (Publisher端) PUBREL响应（Publish Release，发布释放）→ (Broker端)
    - (Broker端) PUBCOMP响应（Publish Complete，发布完成）→ (Publisher端)
  - （优点）最高可靠性：无丢失且无重复，适用于关键业务（如医疗设备传输）。
  - 缺点：资源开销大，四次交互导致延迟高，消耗更多带宽和存储。

MQTT的独有模式：服务质量等级、遗嘱和心跳（续）

发布与订阅的QoS匹配

- **MQTT代理在转发消息时，最终QoS取发布端QoS和订阅端QoS的较低值。例如：**
  - 发布端QoS=2，订阅端QoS=1，则转发QoS=1；
  - 发布端QoS=0，订阅端QoS=2 ，则转发QoS=0。

MQTT的遗嘱机制

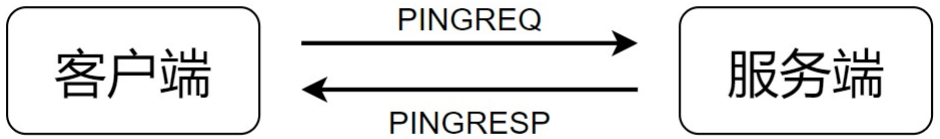
- **最后的遗嘱和遗言（Last Will and Testament，LWT）**
  - 客户端在与代理的连接保持阶段设置并发送遗嘱消息，以便在连接不正常断开时，由代理将LWT广播给所有订阅该主题的所有其他客户端。

MQTT的心跳（见左图）

- **防止TCP连接的半开启状态（连接一端在网络中消失，另一端仍然处于ESTABLISHED状态）的保活机制：**
  - 客户端定时向服务端发送心跳请求（PINGREQ），
  - 服务端收到心跳请求后回复一条心跳响应（PINGRESP），确认连接保持。

MQTT的遗嘱机制关键步骤

- **连接阶段：客户端在CONNECT报文中声明遗嘱参数，包括**
  - 遗嘱主题（Will Topic，如device/status，用于接收离线通知的订阅路径）、遗嘱内容（Will Payload，传递断线状态或指令）、oS级别（Will QoS，定义消息传输可靠性）、保留标志（Will Retain）。
- **运行时**
  - 客户端正常在线时，可主动向遗嘱主题发送online状态消息并设置保留标志，覆盖原有遗嘱内容；
  - 客户端异常断线时（如网络故障、心跳超时、协议错误），Broker立即或在延迟后发布遗嘱消息。
- **断线判断条件**
  - 网络I/O故障或心跳超时未响应；客户端未发送DISCONNECT报文即关闭连接；服务端因协议违规主动断开连接等。



MQTT的发布-订阅模式与CoAP的发布-订阅模式的比较

比较维度	MQTT（基于主题订阅）	CoAP（基于Observe选项）
传输协议	基于TCP（可靠、面向连接）	基于UDP（轻量、无连接）
代理角色	必须依赖中央代理（Broker）路由消息	可直连，或通过代理（如CoAP-HTTP代理）
消息格式	固定头部+可变负载（主题名明确分层）	紧凑二进制+选项字段（如URI路径、Observe）
REST兼容性	❌ 专注于消息传递，不与REST直接映射	✓ 兼容HTTP语义（GET/ POST/ PUT/ DELETE）
多播支持	依赖代理单播分发（无原生多播支持）	支持UDP多播（如局域网设备群控）
QoS机制	- 明确三个QoS等级（0/1/2） - 提供端到端可靠性保证	- Confirmable/Non-confirmable消息 - 基础重传机制

MQTT的服务端部署和客户端编排

MQTT的服务端部署（OASIS MQTT/结构化信息标准促进组织官网：https://mqtt.org/mqtt-specification/）

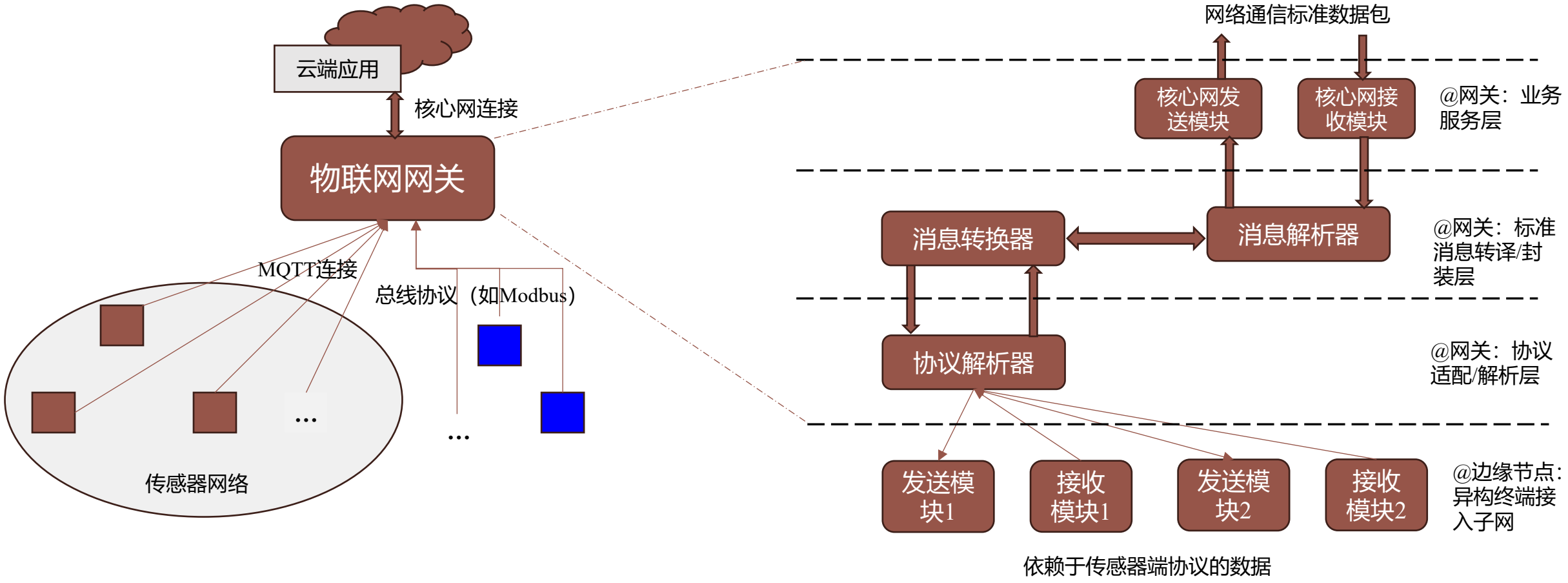
- 常见服务端架构Mosquitto （Windows下示例）
  - Eclipse Mosquitto for MQTT 5.0, 3.1.1, and 3.1 (https://github.com/eclipse-mosquitto/mosquitto)。
  - Client-Proxy-Client结构：
    - （1）通过安装目录中“mosquitto.conf”的配置文件，包括IP地址、端口，连接max_keepalive等参数设定服务器配置。
    - （2）在命令行中通过“mosquitto -c mosquitto.conf”命令启动服务端。
    - （3）可以在命令行中通过mosquitto_sub、 mosquito_pub命令启动客户端，订阅或发布给定topic（略）
- 常见的客户端开发库（Python）
  - Eclipse paho-mqtt库安装命令： pip install paho-mqtt。

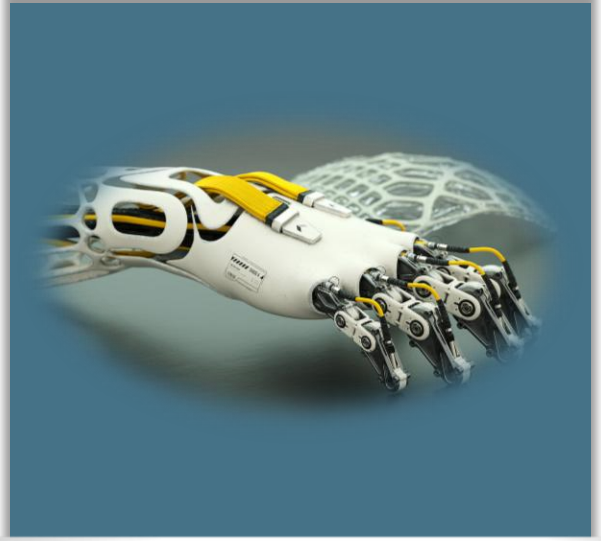
基于paho的简单MQTT客户端（发布者发布到test/topic）

```
8   # Import package
9   import paho.mqtt.client as mqtt
10
11  # Define Variables
12  MQTT_HOST = "127.0.0.1"
13  MQTT_PORT = 1883
14  MQTT_KEEPALIVE_INTERVAL = 45 #seconds
15  MQTT_TOPIC = "test/topic"
16  MQTT_MSG = "hello MQTT!"
17
18
19  # 定义publish事件回调函数
20  def on_publish(client, userdata, mid):
21      print ("Message Published...")
22
23  # 初始化MQTT Client
24  mqttc = mqtt.Client()
25
26  # 注册publish回调函数
27  mqttc.on_publish = on_publish
28
29  # 连接MQTT Broker（代理）
30  mqttc.connect(MQTT_HOST, MQTT_PORT, MQTT_KEEPALIVE_INTERVAL)
31
32  # 将对应topic信息发布到MQTT代理
33  mqttc.publish(MQTT_TOPIC,MQTT_MSG)
34
35  # 关闭连接
36  mqttc.disconnect()
```

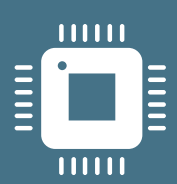
物联网协议转译模块（网关/中间件）

- 一种可能的设计：协议转换模块坐落在网关





第五章
基于云的计算与决策



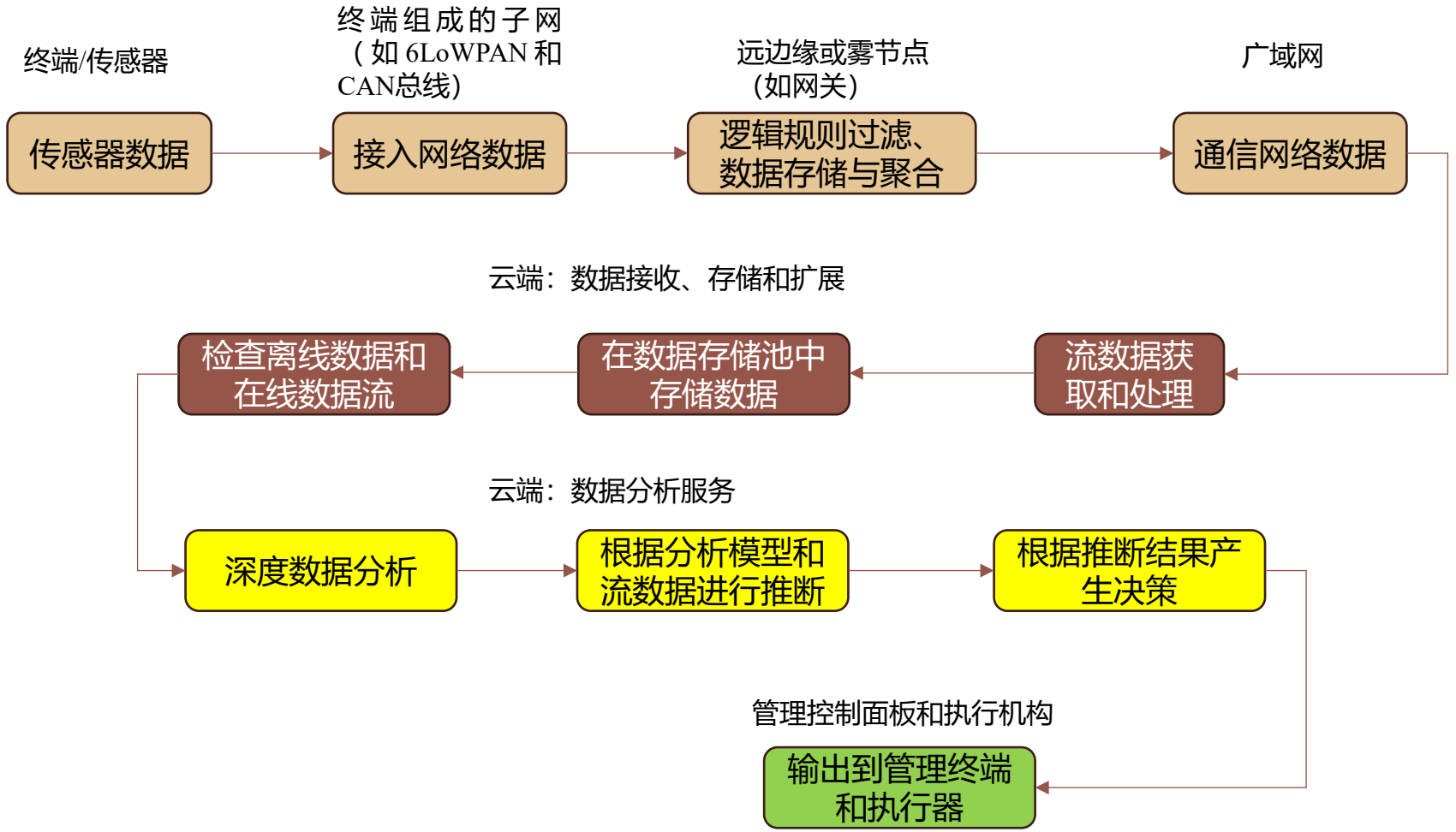

云上的数据处理：数据结构化

- **数据处理工作的一般流程：**
  - **数据流连接：将多个数据流（如不同传感器的数据流）合并成一个单一流。**
  - **预处理：对异步数据流中的丢失数据、乱码数据、无序数据等异常数据进行识别和处理。**
  - **数据结构化：结构化意味着将数据转译为可预测的数据格式（如SQL关系模型）。**
  - **数据归档：流格式数据落库（如常见的MongoDB）。**
  - **批处理查询：利用数据库做大规模的数据查询。**
  - **基于大数据的离线分析：如机器学习模型的训练。**
  - **根据离线分析和模型构建进行系统状态预测与控制。**

回忆：工业大数据的特点

- **工业大数据：工业领域产品全生命周期中产生的多源异构数据集合，往往无法在一定时间范围内用常规软件工具进行捕捉、管理和处理。**
- **工业大数据的特点**
  - **海量性（Volume）**：单台设备每秒可产生数百测点数据（如风机测点可达500个/秒），大型企业数据总量往往达EB级。
  - **多模态（Multiple Modality）**：数据类型多样，类型包括结构化（数据库记录）、半结构化（日志文件）和非结构化（设计图纸、视频）。这是由工业生产社会化属性决定的。
  - **实时性**：工业大数据的重要应用场景是实时监测、预警、控制等。生产现场往往要求毫秒级实时响应，典型如设备故障预测需在数据生成后立即分析。
  - **强关联性（Strong-Relevance）**：工业现场数据间存在复杂的显性和隐性机理关联（如零部件状态与设备寿命），需跨环节整合分析。
  - **闭环性（Closed-loop）**：工业大数据的分析和使用需支撑“感知-分析-反馈-控制”的动态优化闭环，例如通过基于大数据分析的实时工艺参数调整提升良品率。

物联网数据管道：一个一般化的例子



基于云的数据分析-控制响应的必要模块：

- **业务规则引擎**：定义数据处理行为并产生执行结果。
- **流数据处理引擎**：基于流数据进行离线化注入（到数据库）和在线化处理（产线规划引擎）。
- **数据分析引擎**：进行模型的生成和训练，以及接入其他深度数据分析方法。
- **信号与控制规划-回传**：生成控制信号并回传到边缘端或执行机构。

云上的数据结构化处理：从远边缘/雾端开始

- **基于远边缘端网关的数据预处理——服务需求：**
  - **网关上的串口数据服务：将串口IO数据（如RS-232、RS-485、现场总线等接入协议下的报文数据）转译成网络IO兼容数据。**
  - **网关上的通信管理服务：管理以太网（TCP/IP）连接、非IP连接（Zigbee等）的报文接收和向上转发。**
  - **网关上的数据缓存服务：基于内存型数据库（如Redis数据库）建立数据缓冲池。**
    - 注意1：由于数据生命周期的限制，网关通常不需要将数据永久存储。
    - 注意2：由于多协议解析器的存在，内存上的数据缓存池的存取需要遵循多进程/线程读写时的ACI（D）基本特性：即**原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）**，内存数据不保证**持久性（Durability）**。
- **网关服务程序的一般框架：**
  - **网关上的每个数据/通信服务以单独进程（或线程）实现，进程间通信酌情使用管道、信号量、消息队列、共享内存或套接字通信实现。**
  - **不同协议报文可依靠基于Redis内存数据库为代表的缓冲区，进行存储-转译/重写报文-发送的操作。**

流数据从雾端网关到云端的聚合

- 工业生产环境中的实时IoT数据往往是有缺陷的，在数据聚合时应注意处理
  - **数据丢失**：由传感器故障、掉电甚至断网导致的数据流中的部分数据缺失。
  - **数据格式错误**：由传感器故障、传输错误或编解码错误导致的数据流中的部分数据不正确。
  - **数据失序**：数据可能从多个路径流向云端，由于延迟等作用导致到达顺序被打乱。
- 自有流数据聚合引擎的设计原则
  - 处理能力能够随事件的增长而扩展；
  - 提供向其他模块（如数据库模块）发布的接口API；
  - 提供可伸缩性的规则处理能力。
  - 提供对分布式数据仓库（如Hadoop Distributed File System，HDFS）的支持。

云上的数据存储：自有数据库的选择

- **数据库管理系统（DataBase Management System，DBMS）的分类：**
  - **关系型数据库（relational database）：数据库默认的标准，使用SQL查询语言，且有着预定义的模式。**
    - 通过二维表（也称为关系）来组织和存储数据，表与表之间通过主键和外键建立关联；
    - 使用结构化查询语言（SQL）进行数据操作，包括数据的增删改查等；
    - 具有明确定义的数据模式（schema），这意味着数据的结构和关系在数据库设计之初就已经确定，并严格遵循，从而确保了数据的完整性和一致性。
  - **非关系型数据库（non-relational database）：非传统型的数据库，支持无模式数据（Schema-less，不限表结构的数据）的存储。典型产品如MongoDB（以JSON格式存储）和Redis（以Key-Value快速存取数据）。**
  - **小型数据库（small database）：定位于嵌入式系统（边缘节点）使用环境，如SQLite。**
    - 小型数据库直接集成到应用程序中，与宿主程序共享进程资源，无需独立运行的数据库服务。但仍支持ACID事务（原子性、一致性等）、标准SQL语法（如JOIN、索引、触发器等）。

选择自有数据库：注意服务端的需求特性

- 规划好事务服务模块和数据库服务模块连接的通信标准：
  - ODBC（开放式数据库互联）协议连接；
  - JDBC（Java程序数据库连接）协议连接；
  - TCP/IP连接。
- 在关系型和非关系型数据库间做出选择：
  - 是否需要支持海量数据（典型如搜索引擎）并发？（是：非关系型数据库）
  - 是否需要处理非结构化数据（如图片、文本、视音频等不便以二维表形式存在的数据）和半结构化数据，如以XML和JSON格式描述的，不同数据片带有一定结构但又不如关系型数据一致严格定义的数据）？（是：非关系型数据库）
  - 其他：如对不用编程语言的支持、数据压缩方式、能否支持水平扩展等，依不同数据库特性而定。

一般数据库的部署：以MySQL为例

- **云端SaaS或PaaS服务下一般可以选定服务商镜像通过Web GUI部署并配置**
  - **右图：百度云服务（地址：https://console.bce.baidu.com/gaiadb/#/gaiadb/cluster/create）下部署MySQL兼容数据库的Web客户端展示。**

一般数据库的部署：以MySQL为例（续）

- 私有云或服务器端的情况（以Ubuntu环境下的典型命令行指令为例，Windows环境略）
  - 安装：sudo apt-get install mysql-server，安装后设置root账号密码。
  - 启动：service mysql start （停止：stop；重启：restart）。
  - 登录（在terminal命令行中）： mysql –u root –p [password]。
- 服务器端的可调配置：
  - mysql-server提供一些常见配置选项，一般保存在配置文件（Linux： etc/my.cnf；Windows：mysql的安装路径/my.ini）中:
  - 端口号：[mysqld]字段，默认端口号3306；
  - 数据文件存储路径：[datadir];
  - 服务端使用的字符集（uft-8等），允许最大并发连接数，默认存储引擎等。

数据库的创建、调试和开发：以MySQL为例

- **数据库的创建和本地调试**
  - 在快速开发过程中，常将数据库的创建（相关的MySQL语句集）放在一个*.sql脚本文件里，通过mysql的source命令执行创建相应数据库和表。
  - 数据库的服务器端调试工具：一般需要安装mysql-client命令行环境或其他如MySQL Workbench等图形化工具进行本地开发和管理。
- **服务器端事务程序模块的MySQL嵌入：**
  - C/C++开发环境下的MySQL官方接口：安装MySQL Connector/ C++或Connector/C驱动，一般由头文件 #include <mysql/mysql.h> 提供接口定义。
  - C/C++下的ODBC接口：安装MySQL Connector/ODBC驱动，一般由头文件#include <sql.h>； #include <sqlext.h>； #include<sqltypes.h>提供接口。

服务器端事务程序模块的MySQL嵌入：Python开发环境

- **服务器端事务程序模块的MySQL嵌入：Python[1]**
  - **Python开发环境下的MySQL官方接口： mysql-connector-python库**
    - **安装命令：（conda的激活环境下） pip install mysql-connector-python**
  - **Python开发环境下的ODBC接口：pymysql库或pyodbc库**
    - **安装命令： pip install pymysql**
    - **其他依赖：安装MySQL Connector/ODBC驱动。**
- **一般开发原则**
  - **服务器端事务程序模块的MySQL嵌入需围绕事务原子性-一致性-隔离性-持久性保障（Atomicity、Consistency、Isolation、Durability）、代码健壮性设计及性能优化展开。**
  - **代码健壮性包括但不限于：**并发（如多线程模式下的）死锁预防与排查；事务块设计模式下的事务封装与异常处理等。
  - **性能优化包括但不限于**：实时数据处理（如结合流式计算框架Apache Kafka + Flink ）；高并发写入；嵌入式MySQL的轻量化集成等。

1：数据库的Python嵌入基本操作案例见后页。

SQL数据库的基本读写操作

- **关系数据库的数据模式**
  - **基本表**：实际存储在数据库中的关系表，由二维表格式的键（Key）-值（Value）组成。
  - **视图**：是一种逻辑上的（虚拟的）关系表，由其他基本表和视图导出生成。数据库只存放视图的定义而不存放数据，它保存了一个SQL查询，同时可以像普通表一样被查询。
- **对数据库表的基本操作语句**
  - 创建：CREATE TABLE…（后面接数据表的表头定义，包括主键和外键，Primary Key & Foreign Key），格式为<表名>（ <新列名><数据类型>[完整性约束条件],…, PRIMARY KEY （键名））；
  - 修改：ALTER TABLE <表名>[ADD<新列名><数据类型>[完整性约束条件]];

    （或） [DROP<完整性约束名>];

    （或） [MODIFY<列名><数据类型>];
  - 删除：DROP TABLE <表名>。

SQL数据库的基本读写操作（续）

- 对数据库基本表的常用操作语句
  - 插入：INSERT INTO 基本表名（字段名 [，字段名]… VALUE(常量 [,常量]…);

    或 INSERT INTO 基本表名（列表名）SELECT 查询语句;
  - 删除：DELETE FROM 基本表名 [WHERE 条件表达式];
  - 更新/修改：UPDATE 基本表名 SET 列名=值表达式 [WHERE 条件表达式][WITH CHECK OPTION];
- 对数据库视图的操作
  - 创建：CREATE VIEW 视图名（列表名） AS SELECT 查询语句;
  - 删除：DROP VIEW 视图名。
- SQL查询操作请参见附录2

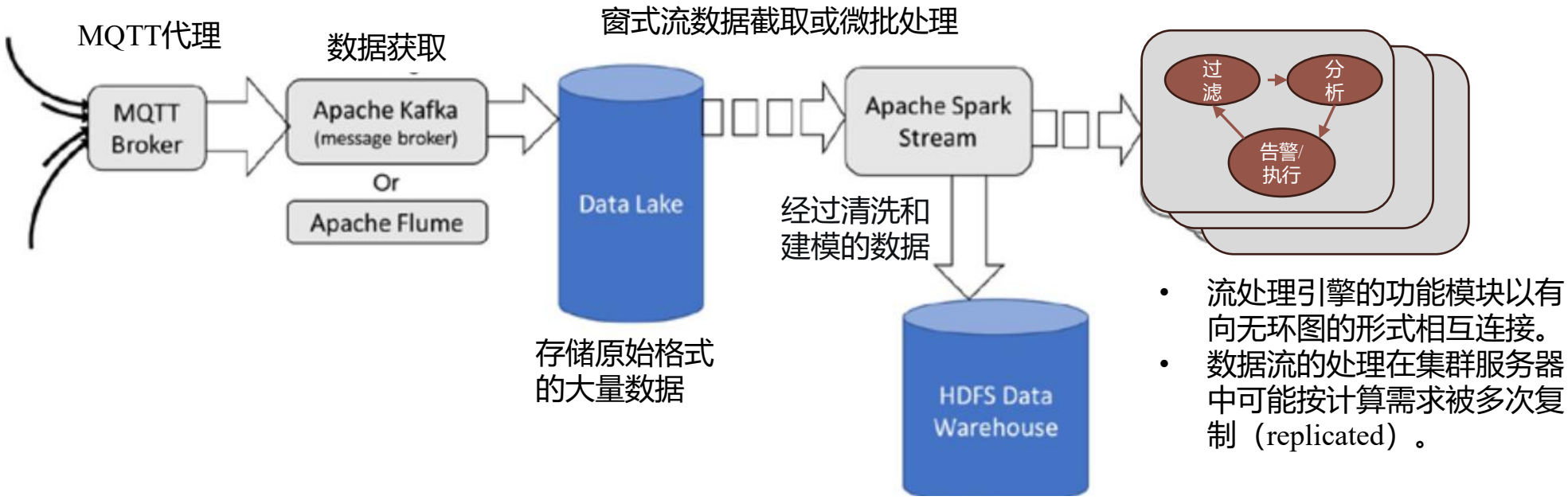
数据库操作在程序语言中的嵌入：Python示例

```python
import mysql.connector
from mysql.connector import Error

# 连接到MySQL服务器
try:
    connection = mysql.connector.connect(
        host='localhost',  # 你的MySQL服务器地址
        user='username',  # 你的MySQL用户名
        password='password',  # 你的MySQL密码
        # 由于我们要创建数据库，所以先不连接到具体的数据库
    )

    if connection.is_connected():
        print('Connected to MySQL server')

        # 创建名为'test'的数据库
        cursor = connection.cursor()
        cursor.execute('DROP DATABASE IF EXISTS test;') #如果已存在则删除库
        cursor.execute('CREATE DATABASE test DEFAULT character set utf8 collate utf8_bin;')
        print('Database "test" created successfully.')

        # 切换到新创建的数据库
        connection = mysql.connector.connect(
            host='localhost',
            user='your_username',
            password='your_password',
            database='test'
        )

        # 在'test'数据库中创建'sensor_data'表格
        cursor = connection.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS sensor_data (
                id INT NOT NULL AUTO_INCREMENT COMMENT '数据标签',
                name VARCHAR(255) COMMENT '传感器名',
                time_stamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                temperature FLOAT
                PRIMARY KEY(id)
            );
        ''')
        print('Table "sensor_data" created successfully.')

except Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print('MySQL connection is closed.')
```

在Python中嵌入数据库操作主要通过数据库连接库实现，支持主流关系型数据库（如MySQL、PostgreSQL、SQLite）和非关系型数据库（如MongoDB）。

分布式流数据处理引擎：Apache Spark简介

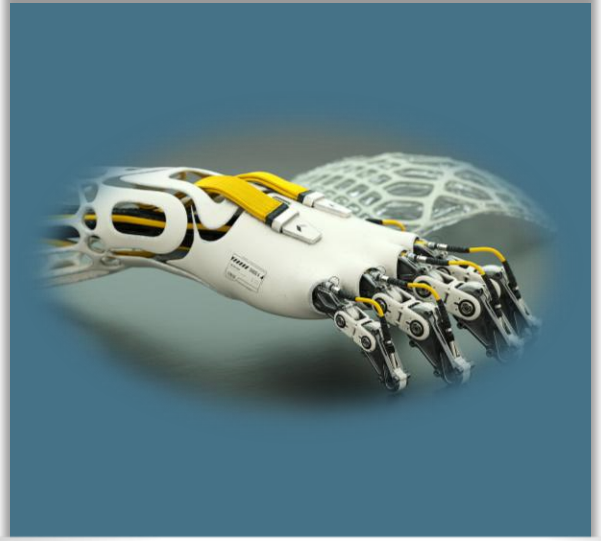
- **Apache Spark是目前最流行的分布式流数据处理引擎**
  - **Spark Streaming：Apache Spark中负责流处理的核心组件。它提供了实时数据流处理功能，并支持微批处理和Structured Streaming两种模式。**
  - **Spark Streaming从数据源（Apache Kafka、Flume等，如从Kafka代理服务器获取的各种MQTT消息）接收实时数据流，并进行处理和分析。**
- **Apache Spark的数据处理流程**



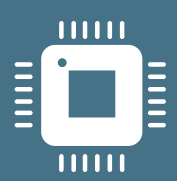
- 流处理引擎的功能模块以有向无环图的形式相互连接。
- 数据流的处理在集群服务器中可能按计算需求被多次复制（replicated）。

Apache Spark简介（续）

- **Apache Spark的核心功能**
  - **Apache Spark提供了Java、Scala、Python和SQL等程序语言的API，方便用户选择熟悉的编程语言进行操作。这使得Spark成为了一个非常灵活和易用的数据处理工具。**
  - **Spark 提供了Spark SQL作为核心组件，用于将SQL的结构化查询转化成RDD（Resilient Distributed Datasets，弹性分布式数据集）在云端的分布式节点上进行查询的并行式处理。**
  - **Spark 提供了图形化分析引擎和机器学习库：如MLlib（机器学习库）和GraphX（图形处理库），这些库提供了丰富的算法和工具，可以帮助用户解决各种复杂的数据处理问题。**
  - **Mlib库涵盖了常见的统计机器学习算法：**
    - 分类算法：包括逻辑回归、决策树、随机森林、梯度提升树等，适用于二分类和多分类任务。
    - 回归算法：如线性回归、Ridge回归、Lasso回归等。
    - 聚类算法：例如K均值聚类、高斯混合模型等。
    - 推荐算法：协同过滤算法，如基于用户的协同过滤和基于物品的协同过滤。



第五章
基于云的计算与决策

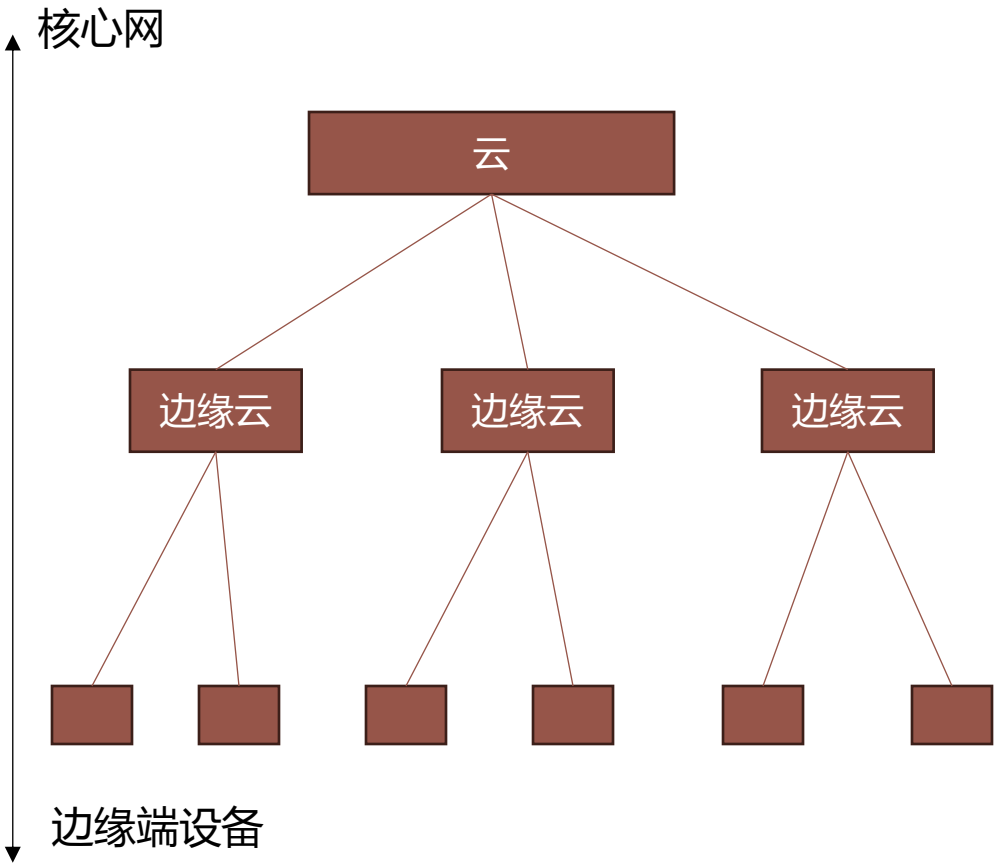



边缘-云的协同框架：引入“边缘云”的概念

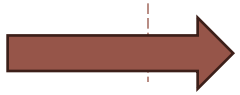
- **边缘云的引入**
  - 边缘云是云计算向网络边缘侧（靠近数据源或用户终端）的延伸，通过将云服务能力（计算、存储、网络）下沉至边缘节点，实现与中心云的深度协同，形成“中心云-边缘云-终端设备”的三层分布式架构。
- **边缘云与雾节点的异同**

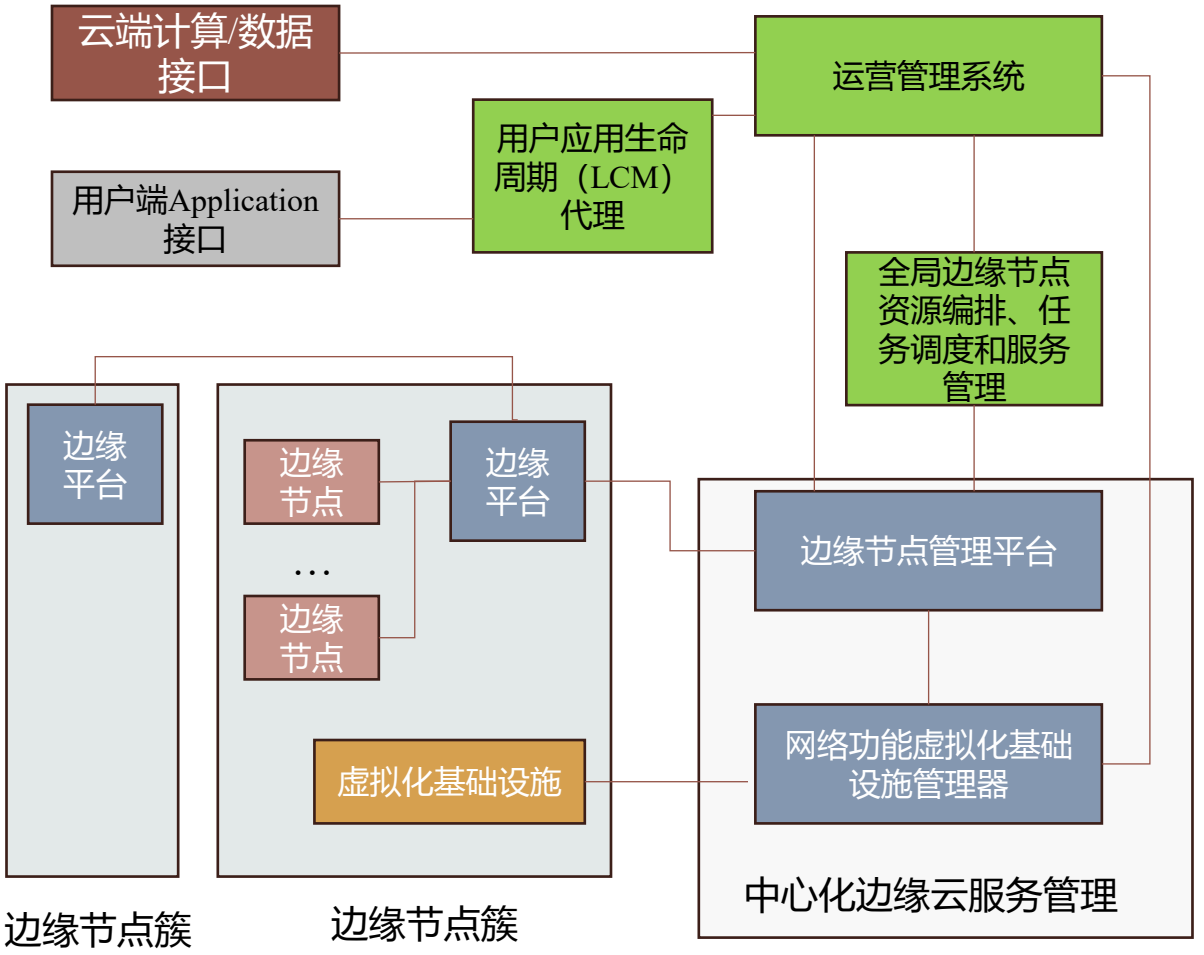
维度	边缘云	雾节点
部署位置	网络最边缘（如基站、智能终端侧），直接接触数据源	位于边缘设备与中心云之间的中间层（如企业网关、路侧单元）
覆盖范围	局部区域（如工厂车间、单个基站覆盖范围）	区域性或广域覆盖（如城市级物联网网络）
架构特性	集中式管理，通常由运营商或云服务商统一部署	分布式网络结构，节点间可自主协同
硬件能力	计算资源较强（如配备GPU的微型数据中心），支持容器化与微服务	中等算力（如工业级网关），侧重协议转换与数据聚合

边缘-云的协同框架：引入“边缘云”的概念



云边协同中设备的地理关系



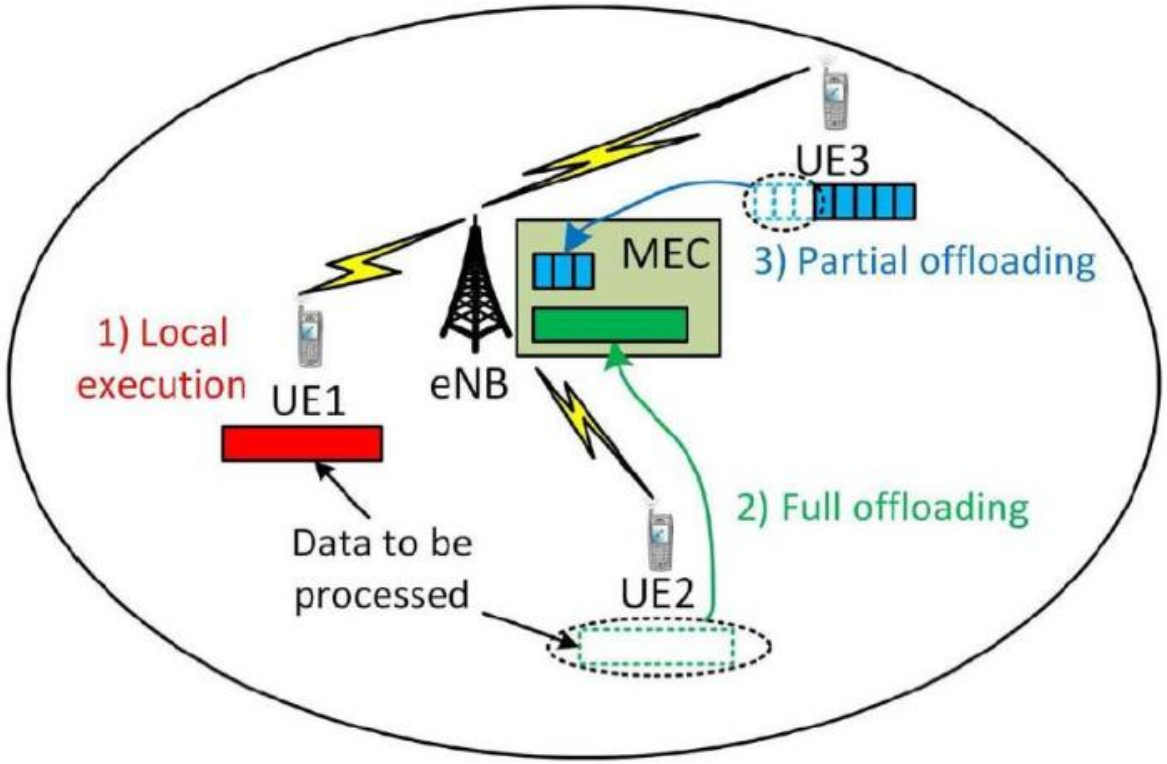


欧洲电信标准化协会（ETSI）定义的云边协同参考框架

云边协同技术场景（云端向边缘端计算卸载）

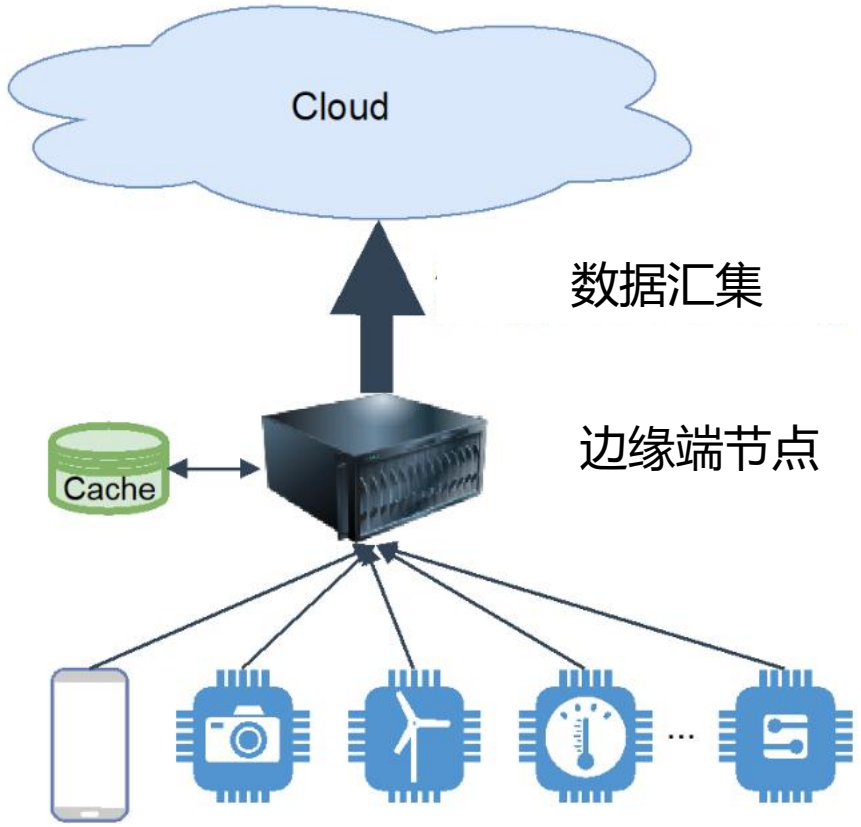
云边协同中的计算卸载是指将原本由云端处理的任务动态分配至边缘节点，以优化资源利用、降低延迟并提升系统效率。典型任务卸载场景如下：

- **1）不卸载（None Offloading）**
  - **云端本地执行（Local execution）**
- **2) 部分任务卸载（Partial Offloading）**
- **3）全部任务卸载（Full Offloading）**
- **任务卸载前提**
  - **任务可卸载（可分割）；**
  - **数据量可知/可估计**
    - **典型任务1：视频和图像编码；**
    - **典型任务2：在线游戏流式渲染。**
  - **分割后子任务依赖性明确**
    - **并行任务；**
    - **串行任务（有向无环图描述）。**

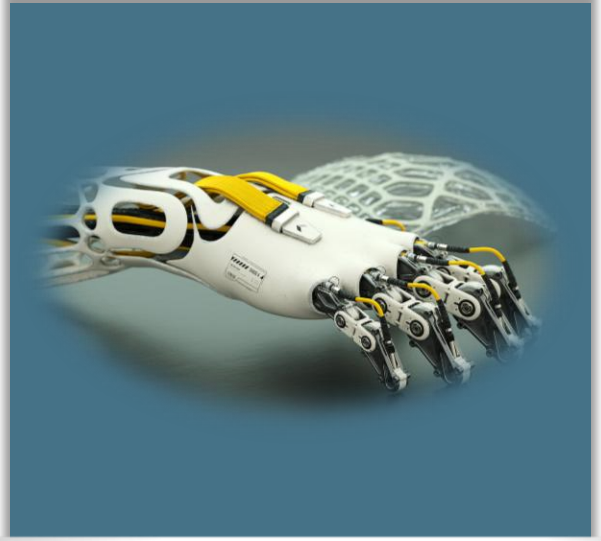


云边协同技术场景（边缘端向云端计算汇聚）

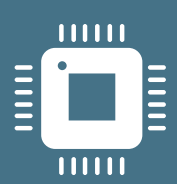
- 在“边缘向云汇聚的”场景下，边缘端用于
  - 数据预处理（Pre-processing）
  - 数据缓存（Caching）
- 如前所述，数据经IoT网关向云端（企业端）的数据池/数据仓库汇集。
- 更多讨论参见：
  - [9] Binayak Kar, Widhi Yahya, Ying-Dar Lin and Asad Ali, “Offloading Using Traditional Optimization and Machine Learning in Federated Cloud–Edge–Fog Systems: A Survey”, in *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, pp. 1199-1226, Second quarter 2023.



IoT终端智能设备（传感器，执行器等）



第五章
基于云的计算与决策




附录1 RESTful API简介

附录2 SQL常见查询操作

附录1：RESTful API

- RESTful API中的REST接口是一种基于类HTTP协议的Web服务接口，它遵循REST（Representational State Transfer）架构风格：
  - 以资源为基础设计：REST接口使用URL来定位资源。每个资源都有一个唯一的URL，这个URL通常反映了资源的结构和层次。
  - 统一接口：接口定位资源，请求类型定位操作。
  - 接口的无状态性：每个请求都是独立的，服务器不会在请求之间保留任何状态信息。
  - 可缓存性：对于不经常变化的资源，REST接口支持缓存，以提高性能。
  - 分层系统设计：允许通过代理服务器、网关等中间件实现系统的扩展性和安全性。

附录1：RESTful API（续）

- **统一的接口：一个完整的URL组成由以下几个部分构成：**

  URI = scheme "://" host ":" port "/" path ["?" query]["#" fragment]

  - **scheme: 指底层用的协议，如http、https、coap等；**
  - **host: 服务器的IP地址或者域名；**
  - **port: 端口，CoAP默认为5683端口；**
  - **path: 访问资源的路径。通常一个RESTful API的path组成如下：**

    /{version}/{resources}/{resource_id}/{subresources}/{subresource_id}；
  - **query: 查询字符串，为发送给服务器的参数；**
  - **fragment: 判断标识符，定位到页面的资源。**

附录1：RESTful API（续）

- **统一的请求方法（操作由HTTP方法指定）：**
  - **GET /collection：从服务器查询资源的列表（数组）；**
    - **GET /collection/resource：从服务器查询单个资源；**
  - **POST /collection：在服务器创建新的资源；**
  - **PUT /collection/resource：更新服务器资源；**
  - **DELETE /collection/resource：从服务器删除资源path: 访问资源的路径。**
  - **RESTful API要求在URL上都以名词的方式出现，以清晰地表示所暴露的资源，并避免动词（如query，add，update，delete）出现。**

附录2：SQL常见查询操作

- 统一语句格式：SELECT-FROM-WHERE
  - SELECT[ALL | DISTINCT]<目标列表达式>[,<目标列表达式>…
    FROM<表名或视图名>[,<表名或视图名>]
    [WHERE<条件表达式>] [GROUP BY<列名 1> [HAVING<条件表达式>]]
    [ORDER BY<列名 2>[ASC | DESC]…
    - SELECT … FROM …语句是必须的；
    - HAVING条件表达式必须与GROUP BY搭配使用。
    - WHERE语句是关系代数中的选择谓词，其中可以使用包括集合、逻辑、算数、比较运算符等多种运算符。
- IoT数据分析中的常用查询
  - 简单查询：在一个表中查找子字段
    - 举例：SELECT SequenceID，SensorName，SensorValue FROM SensorTable WHERE Factory=‘Plant1’;
  - 连接查询：在一个查询中涉及两个以上的表
    - 举例：检索连接到控制中心1的设备：SELECT DeviceNO, DeviceName, FROM DeviceTable, ControllerTable WHERE DeviceTable. DeviceNO = ControllerTable. DeviceNO AND ControllerTable.CotrollerNo=‘C1’.

附录2：SQL常见查询操作（续）

- IoT数据分析中的常用查询（续）
  - 子查询：在一个SELECT-FROM-WHERE语句中嵌入另一个SELECT-FROM-WHERE查询模块。
  - 分组查询：在WHERE子句后加入GROUP BY用于将结果集按照一个或多个列进行分组
    - 注意：在此使用聚合函数，如SUM，COUNT、AVG对每个分组进行单值输出；
    - 举例：在名为Temperature的表单里查询某个时间段内的给定传感器sensor_id在某天24小时内的感应温度平均值：

```
SELECT sensor_id, AVG(sensor_temperature)
FROM Temperature WHERE sensor_time BETWEEN '2024-01-01 00:00:00' AND '2024-01-01 24:00:00'
GROUP BY sensor_id;
```

作业（2025-05-19）：请提交打印版

- 题1：简述工业物联网从边缘到云端的软件结构框架，分别从数据流向和层次化服务模块两个角度入手。
- 题2：结合课本第8章以及课件内容，简述联邦学习在云-边协同框架下的训练规则和数据流（画出数据流图）。
- 题3：对比MQTT和CoAP协议，列出它们在参与者、数据请求模式、以及信息确认方法等方面的异同。
- 题4（编程实现）：在个人电脑上尝试部署MQTT Mosquitto服务，将其配置在localhost（127.0.0.1）：1883。使用Python的paho库编写发布者和订阅者的服务端，以默认QoS服务等级发布消息。要求：至少发布一则消息“Hello MQTT！”到test/topic下，发布-订阅结构为{“id”: “随机整数”}的消息10条到同一自定义topic，并以JSON形式在订阅者当前目录存为“result.json”文件。（程序及输入-输出结果以打印文档的形式提交）

结语（Q&A）

- 参考文献

- [1] 曾凡太等，物联网之智——智能硬件开发与智慧城市建设，机械工业出版社，2020.
- [2] Perry Lea, 中国移动设计院北京分院译，物联网系统架构设计与边缘计算，机械工业出版社，2021.
- [3] 郭斌等，智能物联网导论，机械工业出版社，2022.
- [4] 连志安，物联网嵌入式开发实战，清华大学出版社，2021
- [5] Sebastian Raschka等著，陈斌译，Python机器学习，机械工业出版社，2021.
- [6] 朱文伟，李建英，Linux C/C++服务器开发实践，清华大学出版社，2022.
- [7] 詹姆·布尔塔著，卢浩，任鸿等译，Python架构模式，机械工业出版社，2024.
- [8] 褚华，霍秋艳主编，软件设计师教程，清华大学出版社，2018.
- [9] Binayak Kar, Widhi Yahya, Ying-Dar Lin and Asad Ali, “Offloading Using Traditional Optimization and Machine Learning in Federated Cloud–Edge–Fog Systems: A Survey”, in *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, pp. 1199-1226, Second quarter 2023.