数据采集指导书

# 一、实验目标

本节旨在通过实际案例，展示如何利用多种数据采集技术，从不同数据源收集数据，以全面监测和评估建筑结构的健康状态，建筑结构不限于桥梁、高层建筑、大坝、隧道、机场等，本实验以桥梁为例。

# 二、数据采集方案

1.爬虫技术

**目的：**实时获取天气数据，评估其对桥梁结构的影响。

**方法：**

* 选择可靠的气象网站或API。
* 编写爬虫脚本，定时抓取建筑所在地天气数据。
* 将数据存储到本地或云端数据库。

2.物联网传感器

**目的：**实时监测桥梁结构的物理状态。

**方法：**

* 在桥梁关键部位安装温度、应力、位移等传感器。（不同建筑结构的传感器要求可不同）
* 通过物联网技术，实时传输数据至中心服务器。
* 数据处理和分析，以评估结构健康状态。

*说明：本实验使用代码模拟传感器发送数据，具体参见“附件1 传感器模拟”。*

3.日志数据采集

**目的：**保存和管理系统日志数据。

**方法：**

* 将系统产生的日志数据保存。
* 定期上传至数据存储服务器。

*说明：本实验采用txt文档日志，具体参见“附件2 日志读取示例”。*

4.关系型数据库

**目的：**长期保存和管理桥梁相关数据。

**方法：**

* 设计数据库结构，包括历史数据、传感器数据等。
* 定期备份和维护数据库。

# 三、实验指导

按上述数据采集方案设计对于某个建筑结构的健康监测系统，学会利用多种数据采集技术，得到建筑结构的物理信息。具体实现可参考示例代码。

## 3.1实验环境

·高级编程语言，如Python 3.x

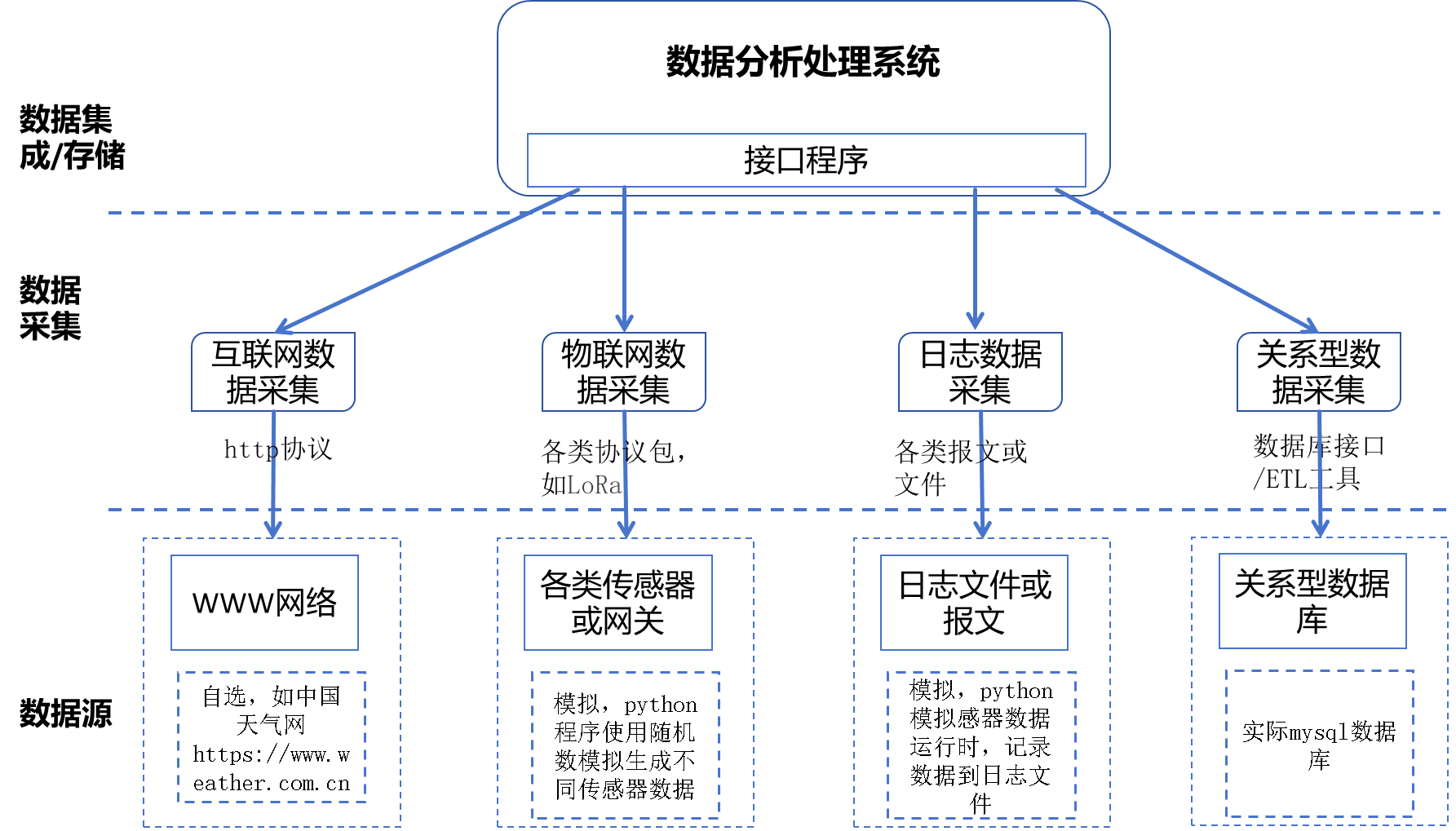
·关系型数据库（mysql等）

·网络连接（爬虫）

·物联网传感器（模拟）

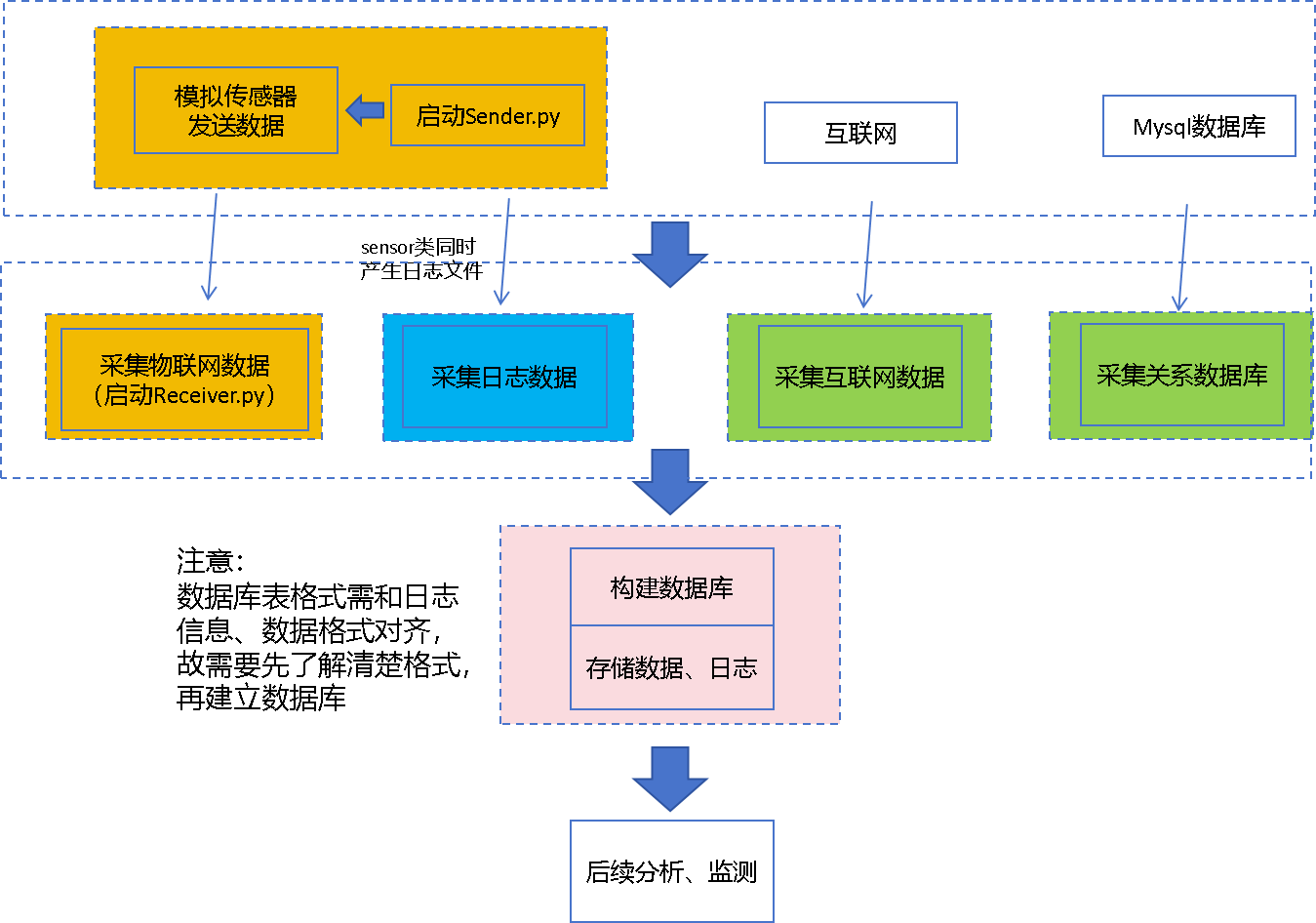
## 3.1实验内容

如下图：



## 3.2实验步骤

实验流程图下图：



### 3.2.1互联网数据采集

实现爬虫数据采集，可选择爬取天气、温湿度、光照强度、风力等影响建筑结构物理特性的参数。代码示例如下：

1. 导入模块

import requests

import re

import json

首先，我们需要导入一些 Python 模块：

·requests: 用于发送 HTTP 请求，获取网页内容。

·re: 用于处理正则表达式，从网页内容中提取我们需要的数据。

·json: 用于解析 JSON 格式的数据。

1. 定义城市代码

city\_codes = {

'北京': '101010100',

'上海': '101020100',

'成都': '101270101',

'嘉定': '101020500',

# 其他城市代码

}

这里我们定义了一个字典 city\_codes，它将城市名称映射到对应的天气代码。天气代码是访问天气网站时需要用到的参数。

1. 获取天气数据

def get\_weather\_from\_internet(city):

if city not in city\_codes:

raise ValueError(f"City code for {city} not found")

city\_code = city\_codes[city]

url = f'http://www.weather.com.cn/weather1d/{city\_code}.shtml'

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'}

timeout = 10

response = requests.get(url, headers=headers, stream=True, timeout=timeout)

if response.status\_code != 200:

raise ValueError("Failed to retrieve content")

pattern = re.compile(r'var observe24h\_data = (.\*?);')

match = pattern.search(response.text)

if match:

json\_str = match.group(1)

data = json.loads(json\_str)

latest\_data = data['od']['od2'][0]

temperature = latest\_data['od22']

humidity = latest\_data['od27']

weather = 1 # 0 sunny, 1 cloudy, 2 ... # 这个后面可以自己修改成类似的定义，或者从爬取内容中获取

return weather, temperature, humidity

else:

print("未找到温湿度信息")

return None, None

接下来，我们定义了一个函数 get\_weather\_from\_internet，用于从互联网获取指定城市的天气数据。

·参数: city 表示城市名称。

·检查城市代码: 首先检查传入的城市是否在 city\_codes 字典中，如果没有找到对应的代码，则抛出异常。

·构建 URL: 使用城市代码构建请求 URL，可选择其他url收集更为精确的数据

·设置请求头: 为了模拟浏览器访问，我们设置了 User-Agent 请求头。

·发送请求: 使用 requests.get 方法发送 HTTP 请求，并设置超时时间为 10 秒。

·检查响应状态码: 如果响应状态码不是 200，则表示请求失败，抛出异常。

·正则表达式匹配: 使用正则表达式从网页内容中提取包含天气数据的 JSON 字符串。

·解析 JSON 数据: 将提取到的 JSON 字符串解析为 Python 字典。

·提取温度和湿度: 从解析后的字典中提取温度和湿度信息。

·返回结果: 返回天气、温度和湿度信息。

1. 调用函数获取天气数据

city = '嘉定'

weather, temperature, humidity = get\_weather\_from\_internet(city)

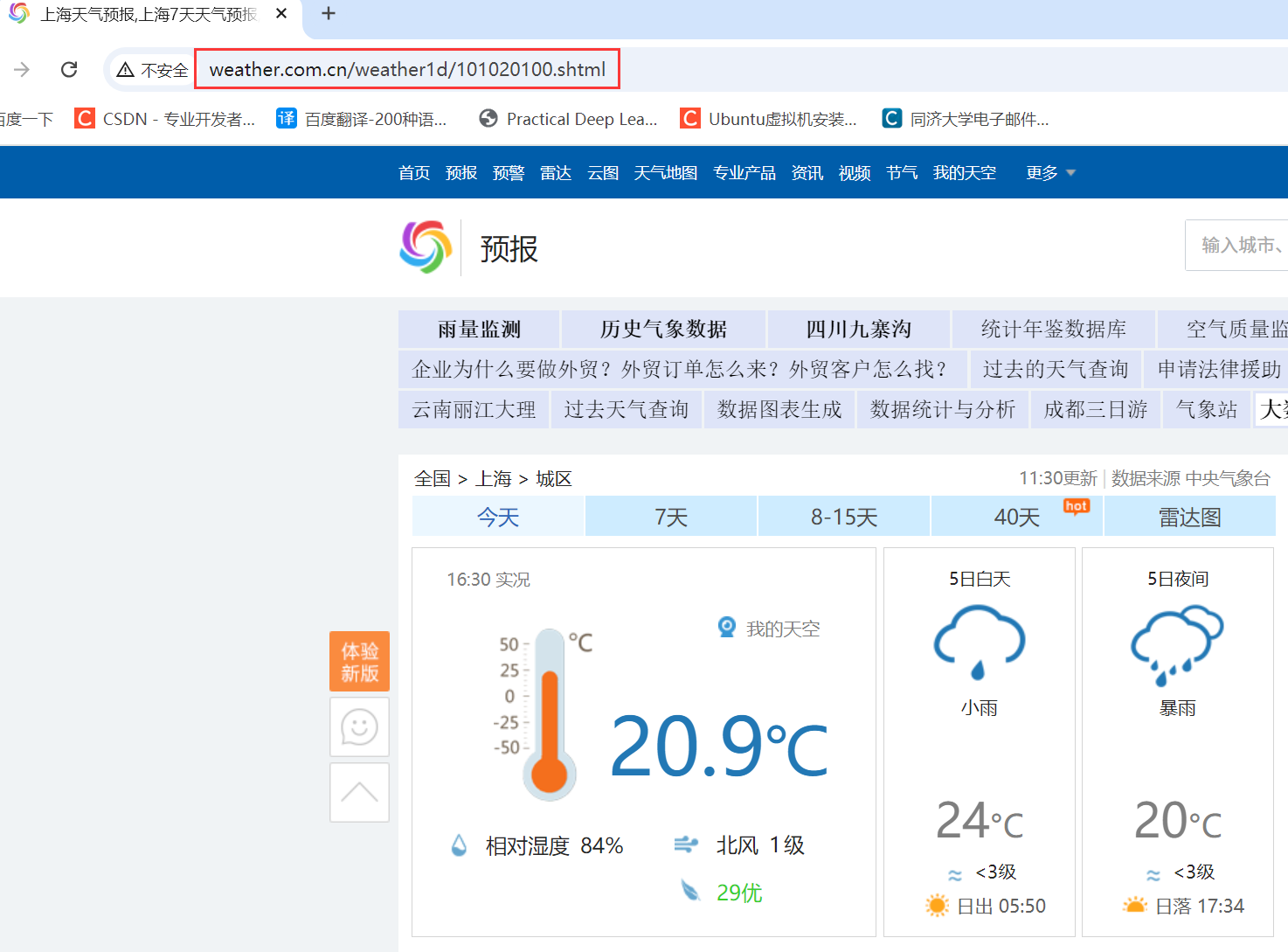
if temperature and humidity:

print(f'The temperature in {city} is {temperature}, humidity is {humidity}, weather is {weather}')

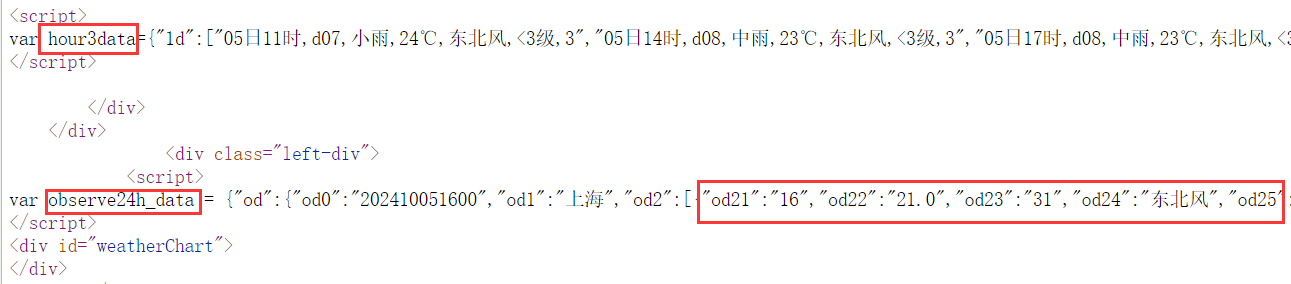
else:

print(f"Failed to retrieve temperature and humidity for {city}")

最后，我们调用 get\_weather\_from\_internet 函数，传入城市名称嘉定，获取该城市的天气数据，并打印结果。

下图为网页信息以及网页源码,我们可以查看源码确定好所需要爬取信息：





### 3.2.2物联网数据采集

模拟得到传感器数据，传感器数量不少于5个，实现模拟传感器得到具体流程更佳，如遵循zig-bee、CoAP、NB-IoT、LoRa等协议的传感器与网关设备的数据通信。

在物联网（IoT）系统中，传感器数据采集是至关重要的一环。为了帮助初学者理解这一过程，我们将模拟一个简单的物联网数据采集场景，并详细说明模拟的原理、如何运行模拟程序以及如何读取数据。

1.模拟原理

在实际的物联网系统中，传感器通常通过无线或有线方式与网关设备通信，网关设备再将数据传输到云端或本地服务器。为了模拟这一过程，我们可以使用以下步骤：

·传感器模拟: 使用软件或硬件模拟传感器，生成模拟数据。

·网关模拟: 模拟网关设备，接收传感器数据并进行初步处理。

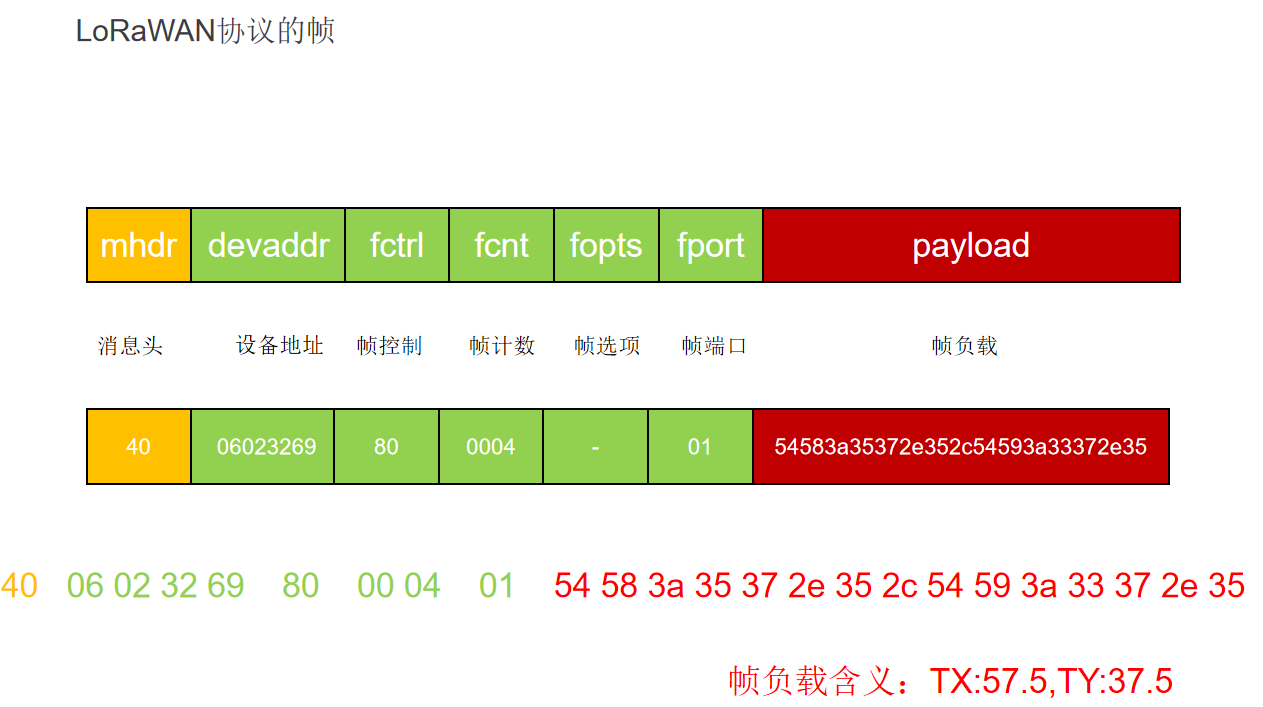
·数据传输: 模拟数据传输过程，将数据发送到云端或本地服务器。

·数据读取: 从云端或本地服务器读取数据。

2.模拟步骤

模拟原理中的数据传输、读取阶段，我们简化为函数输出协议帧；网关模拟，简化为对数据帧格式的处理。示例如下（代码见附件1）：

我们采用LoRaWAN协议的帧(也可以采用其他协议)，我们使用随机函数生成数据，将其以指定帧格式进行传输，而后接收，并进行处理（模拟网关）。



1. 导入模块

import struct

import random

import socket

import logging

import time

首先，我们导入了一些必要的Python模块：

·struct: 用于将数据打包成二进制格式。

·random: 用于生成随机数，模拟传感器数据。

·socket: 用于获取本机IP地址。

·logging: 用于记录日志。

·time: 用于控制时间间隔。

1. Sensor类

class Sensor:

def \_\_init\_\_(self, sensor\_type, sensor\_id): #输入传感器种类 传感器id

self.dev\_addr = self.get\_local\_ip() # 使用本机地址作为设备地址

self.fcnt = 0 # 帧计数器初始化为0

self.fport = 0x01 # 应用层协议

self.sensor\_type = sensor\_type # 传感器类型

self.sensor\_id = sensor\_id # 传感器编号

self.logger = self.setup\_logger(sensor\_id) # 设置日志记录器

self.status = self.generate\_random\_status() # 传感器状态，随机生成

def get\_local\_ip(self):

# 获取本机IP地址

hostname = socket.gethostname()

local\_ip = socket.gethostbyname(hostname)

return int(local\_ip.replace('.', ''))

def setup\_logger(self, sensor\_id):

# 设置日志记录器

logger = logging.getLogger(f"Sensor\_{sensor\_id}")

logger.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

file\_handler = logging.FileHandler(f"sensor\_{sensor\_id}.log")

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

return logger

def generate\_frame(self):

mhdr = 0x40 # Unconfirmed Data Up

fctrl = 0x80 # Frame Control

fopts = b'' # Frame Options

fport = self.fport

sensor\_data = self.generate\_random\_sensor\_data(self.sensor\_type)

frm\_payload = self.format\_sensor\_data(sensor\_data)

fhdr = struct.pack('>I', self.dev\_addr) + struct.pack('B', fctrl) + struct.pack('>H', self.fcnt) + fopts

frame = struct.pack('B', mhdr) + fhdr + struct.pack('B', fport) + frm\_payload

self.fcnt += 1 # 帧计数器递增

self.log\_sensor\_status(sensor\_data) # 记录日志

return frame

def format\_sensor\_data(self, sensor\_data):

if self.sensor\_type == "Temperature":

return f"T:{sensor\_data['temperature']}".encode('utf-8')

elif self.sensor\_type == "Pressure":

return f"P:{sensor\_data['pressure']}".encode('utf-8')

elif self.sensor\_type == "Tilt":

return f"TX:{sensor\_data['tilt\_x']},TY:{sensor\_data['tilt\_y']}".encode('utf-8')#倾斜传感器

else:

raise ValueError("Unsupported sensor type")

def generate\_random\_sensor\_data(self, sensor\_type):

if sensor\_type == "Temperature":

return {

"temperature": round(random.uniform(0, 50), 2)

}

elif sensor\_type == "Pressure":

return {

"pressure": round(random.uniform(0, 1000), 2)

}

elif sensor\_type == "Tilt":

return {

"tilt\_x": round(random.uniform(-90, 90), 2),

"tilt\_y": round(random.uniform(-90, 90), 2)

}

else:

raise ValueError("Unsupported sensor type")

def generate\_random\_status(self):

statuses = ["Normal", "Warning", "Fault"]

return random.choice(statuses)

def log\_sensor\_status(self, sensor\_data):

status\_message = f"Sensor ID: {self.sensor\_id}, Sensor Type: {self.sensor\_type}, Status: {self.status}, Data: {sensor\_data}"

self.logger.info(status\_message)

Sensor 类用于模拟传感器，这里定义了其属性、功能以及日志信息的记录方式。

其中，setup\_logger 方法用于设置日志记录器，将日志记录到文件中。每个传感器实例都有独立的日志文件。

generate\_frame 方法用于生成数据帧，生成数据帧后，帧计数器递增，并将传感器数据和状态记录到日志中。

generate\_random\_sensor\_data 方法用于生成随机传感器数据。根据传感器类型，生成相应的随机数值。

generate\_random\_status 方法用于生成随机传感器状态（如正常、警告、故障）。

log\_sensor\_status 方法用于将传感器状态和数据记录到日志文件中。

1. parse\_frame\_payload 函数（模拟网关解析帧）

def parse\_frame\_payload(frame):

# 提取 payload 部分

frm\_payload = frame[9:]

# 将 payload 从十六进制转换为字符串

payload\_str = frm\_payload.decode('utf-8')

return payload\_str

parse\_frame\_payload 函数用于解析数据帧中的有效载荷部分，并将其从十六进制转换为字符串。

1. 主函数调用

def main():

sensors = [

Sensor("Temperature", "TEMP\_001"),

Sensor("Temperature", "TEMP\_002"),

Sensor("Pressure", "PRESS\_001"),

Sensor("Pressure", "PRESS\_002"),

Sensor("Tilt", "TILT\_001")

]

while True:

for sensor in sensors:

frame = sensor.generate\_frame()

print(f"Generated Frame from {sensor.sensor\_id}: {frame.hex()}")

parsed\_payload = parse\_frame\_payload(frame)

print(f"Parsed Payload from {sensor.sensor\_id}: {parsed\_payload}")

time.sleep(5) # 每五秒更新一次传感器数据

if \_\_name\_\_ == "\_\_main\_\_":

main()

main 函数是程序的入口。它创建了多个传感器实例，并进入一个无限循环，每5秒生成一次数据帧，并解析有效载荷。这里也可以尝试使用多线程的方式，使得多个传感器同时运行，收发数据。

生成数据如下：

2024-09-02 20:14:49,461 - Sensor\_PRESS\_001 - INFO - Sensor ID: PRESS\_001, Sensor Type: Pressure, Status: Warning, Data: {'pressure': 341.24}

2024-09-02 20:14:54,468 - Sensor\_PRESS\_001 - INFO - Sensor ID: PRESS\_001, Sensor Type: Pressure, Status: Warning, Data: {'pressure': 600.62}

2024-09-02 20:14:59,470 - Sensor\_PRESS\_001 - INFO - Sensor ID: PRESS\_001, Sensor Type: Pressure, Status: Warning, Data: {'pressure': 783.58}

### 3.2.3关系型数据库

关系型数据库是用于存储和管理结构化数据的系统。在本实验中，可使用MySql等作为关系型数据库来存储历史数据、传感器数据及其他相关信息。通过关系型数据库，可以有效地组织和查询数据，支持复杂的数据分析和报告生成。

**数据库表的设计**

2024-09-02 20:14:59,470 - Sensor\_PRESS\_001 - INFO - Sensor ID: PRESS\_001, Sensor Type: Pressure, Status: Warning, Data: {'pressure': 783.58}

结合前面数据采集的例子，现在我们得到的数据有：时间戳、传感器名字、传感器id、传感器种类、传感器状态、数据。基于这些信息，我们可以设计以下数据库表：

1. sensor表

用途: 存储传感器的基本信息。

字段:

sensor\_id (主键): 传感器的唯一标识符。

sensor\_name: 传感器的名称。

sensor\_type: 传感器的类型（如温度、压力、倾斜）。

示例数据：

sensor\_id sensor\_name sensor\_type

PRESS\_001 Pressure 1 Pressure

TEMP\_001 Temperature 1 Temperature

TILT\_001 Tilt 1 Tilt

1. Sensor\_data表

用途: 存储传感器采集的数据。

字段:

data\_id (主键): 数据的唯一标识符。

sensor\_id (外键): 引用 sensors 表中的 sensor\_id。

timestamp: 数据采集的时间戳。

status: 传感器的状态（如正常、警告、故障）。

data: 传感器采集的具体数据（如温度、压力、倾斜角度）。

示例数据：

data\_id sensor\_id timestamp status data

1 PRESS\_001 2024-09-02 20:14:59 Warning {"pressure": 783.58}

2 TEMP\_001 2024-09-02 20:15:00 Normal {"temperature": 25.5}

3 TILT\_001 2024-09-02 20:15:01 Normal {"tilt\_x": 10.2, "tilt\_y": 20.3}

**数据库创建步骤**

1. 创建数据库

首先，我们需要创建一个数据库来存储传感器数据。可以使用以下SQL语句创建数据库：

CREATE DATABASE sensor\_data;

1. 创建sensor表

接下来，我们创建 sensors 表来存储传感器的基本信息：

USE sensor\_data;

CREATE TABLE sensors (

sensor\_id VARCHAR(50) PRIMARY KEY,

sensor\_name VARCHAR(100) NOT NULL,

sensor\_type VARCHAR(50) NOT NULL

);

1. 创建sensor\_data表

然后，我们创建 sensor\_data 表来存储传感器采集的数据：

CREATE TABLE sensor\_data (

data\_id INT AUTO\_INCREMENT PRIMARY KEY,

sensor\_id VARCHAR(50),

timestamp TIMESTAMP NOT NULL,

status VARCHAR(50) NOT NULL,

data JSON NOT NULL,

FOREIGN KEY (sensor\_id) REFERENCES sensors(sensor\_id)

);

**数据插入示例**

假设我们已经从传感器采集到以下数据：

2024-09-02 20:14:59,470 - Sensor\_PRESS\_001 - INFO - Sensor ID: PRESS\_001, Sensor Type: Pressure, Status: Warning, Data: {'pressure': 783.58}

1.插入传感器信息

INSERT INTO sensors (sensor\_id, sensor\_name, sensor\_type)

VALUES ('PRESS\_001', 'Pressure 1', 'Pressure');

2.插入传感器数据

INSERT INTO sensor\_data (sensor\_id, timestamp, status, data)

VALUES ('PRESS\_001', '2024-09-02 20:14:59', 'Warning', '{"pressure": 783.58}');

**数据查询示例**

我们可以使用SQL查询语句从数据库中获取传感器数据。例如，查询所有压力传感器的数据：

SELECT sd.timestamp, s.sensor\_name, sd.status, sd.data

FROM sensor\_data sd

JOIN sensors s ON sd.sensor\_id = s.sensor\_id

WHERE s.sensor\_type = 'Pressure';

**添加传感器位置信息**

我们也可以添加传感器位置信息，为了在数据库中添加传感器位置信息，我们需要对现有的数据库表进行一些修改。具体来说，我们需要在 sensors 表中添加一个新的字段来存储传感器的位置信息。

**1.修改sensors表结构**

首先，我们需要在 sensors 表中添加一个新的字段 location，用于存储传感器的位置信息。可以使用以下SQL语句来修改表结构：

ALTER TABLE sensors

ADD COLUMN location VARCHAR(100);

**2.更新现有数据**

如果 sensors 表中已经有数据，我们需要更新这些数据以包含传感器的位置信息。例如，假设我们有一个传感器 PRESS\_001 位于 "Room 101"，可以使用以下SQL语句更新数据：

UPDATE sensors

SET location = 'Room 101'

WHERE sensor\_id = 'PRESS\_001';

**3.插入新传感器数据时包含位置信息**

在插入新传感器数据时，我们需要确保包含传感器的位置信息。例如，插入一个新的温度传感器 TEMP\_002，位于 "Room 102"：

INSERT INTO sensors (sensor\_id, sensor\_name, sensor\_type, location)

VALUES ('TEMP\_002', 'Temperature 2', 'Temperature', 'Room 102');

### 3.2.4日志数据采集

日志数据采集是指收集和存储系统设备产生的日志信息。系统设备产生的日志数据被保存为文件，并定期上传至数据存储服务器。日志数据包括系统运行状态、错误信息、用户操作记录等，对于系统维护和故障排查至关重要。在本实验中，模拟系统设备产生日志文件，并读入上传。通过日志数据采集，可以跟踪系统运行状态，分析和解决潜在的问题。

我们将通过一个简单的例子，展示如何使用正则表达式从日志文件中提取所需信息，并将其解析为结构化数据。

1. 导入模块

import re

import ast

首先，我们导入了两个Python模块：

·re: 用于处理正则表达式，从日志文件中提取所需信息。

·ast: 用于将字符串形式的Python数据结构（如字典）转换为实际的Python对象。

1. 定义日志文件路径

log\_file\_path = 'Sensor\_TILT\_001.log'

这里我们定义了日志文件的路径，即 Sensor\_TILT\_001.log。这个文件包含了传感器 TILT\_001 的日志数据。

1. 定义正则表达式

log\_pattern = re.compile(

r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2},\d{3}) - ' # 时间戳

r'Sensor\_TILT\_001 - ' # 传感器名称

r'INFO - ' # 日志级别

r'Sensor ID: (\w+), ' # 传感器ID

r'Sensor Type: (\w+), ' # 传感器类型

r'Status: (\w+), ' # 状态

r'Data: ({.\*})' # 数据

)

我们使用正则表达式来匹配日志条目。正则表达式中的每个括号 () 表示一个捕获组，用于提取特定的信息，不同的日志信息用不同的格式匹配：

·时间戳: (\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2},\d{3})

·传感器名称: Sensor\_TILT\_001

·日志级别: INFO

·传感器ID: (\w+)

·传感器类型: (\w+)

·状态: (\w+)

·数据: ({.\*})

1. 读取日志文件

with open(log\_file\_path, 'r') as file:

log\_content = file.read()

我们使用 open 函数打开日志文件，并使用 read 方法读取文件内容。with 语句确保文件在使用完毕后自动关闭。

1. 解析日志内容

log\_entries = log\_pattern.findall(log\_content)

使用 re.compile 编译的正则表达式对象 log\_pattern 的 findall 方法，从日志内容中提取所有匹配的日志条目。findall 方法返回一个包含所有匹配结果的列表，每个匹配结果是一个元组，包含捕获组中的信息。

1. 输出日志条目

for entry in log\_entries:

timestamp, sensor\_id, sensor\_type, status, data\_str = entry

data = ast.literal\_eval(data\_str)

print(f"Timestamp: {timestamp}")

print(f"Sensor ID: {sensor\_id}")

print(f"Sensor Type: {sensor\_type}")

print(f"Status: {status}")

print(f"Data: {data}")

print("-" \* 40)

我们遍历 log\_entries 列表中的每个日志条目，并将其解包为 timestamp, sensor\_id, sensor\_type, status, 和 data\_str。然后，使用 ast.literal\_eval 将 data\_str 转换为实际的Python字典对象 data。最后，打印每个日志条目的详细信息，并在每条日志之间打印分隔线。

# 四、实验要求

**1.爬虫内容：**

·选择可靠的气象网站或API。

·编写爬虫脚本，定时抓取建筑所在地天气数据。

**2.物联网内容：**

·获得模拟传感器发送数据并解析。

·数据处理和分析，以评估结构健康状态。

·修改模拟器，模拟在建筑关键部位安装温度、应力、位移等传感器。

**3.日志数据内容：**

·获得模拟传感器产生的日志数据并保存。

·掌握日志文件的读取。

**4.关系型数据库内容：**

·设计数据库结构，包括历史数据、传感器数据等。

·可根据物联网实验内容建立数据库。

**5.数据存储:**

·以上数据设计相应数据库进行存储。

**6.实验报告：**

·撰写实验报告，包括但不限于步骤、结果、心得体会。

·提交源代码和数据库文件

**7.选做**

1）数据整合与分析：

·从传感器日志文件中读取数据，并将其存储到数据库中。

·根据传感器数据，判断建筑物的健康状态。

·设定健康状态的阈值，例如温度、压力、倾斜角度等。

·当传感器数据超过阈值时，判断建筑物处于“警告”或“故障”状态。

·当建筑物健康状态为“警告”或“故障”时，生成预警提示。

2）扩展要求（选做）：

·整合多个传感器的数据，综合判断建筑物的健康状态。

·分析历史传感器数据，判断建筑物的健康趋势。

# 五、注意事项

**网络连接稳定：**

确保网络连接稳定，以便获取天气数据。不稳定的网络连接可能导致数据获取失败或延迟。

**数据有效性和合理性：**

在模拟传感器数据时，注意数据的有效性和合理性。确保模拟的数据符合实际桥梁监测的需求。

**数据完整性和一致性：**

在处理数据库操作时，注意数据的完整性和一致性。确保数据在存储和查询过程中不会丢失或损坏。

# 六、实验扩展

真实传感器数据：

尝试将模拟的传感器数据替换为真实的物联网传感器数据。这将提高系统的实际应用价值。

数据可视化功能：

优化GUI界面，增加数据可视化功能，如绘制传感器数据的趋势图。这将帮助用户更直观地理解数据变化。

异常检测功能：

实现异常检测功能，当传感器数据超出正常范围时，自动发出警报。这将提高系统的预警能力。