一、概念

1.1 flask概念

> flask是一款基于Werkzeug 和jinja2 的微服务框架。只保留了核心功能werkzeug和jinjia2功能，支持灵活第三方扩展组件，可以根据自己的需求搭建，非常的

1.2 Werkzeug介绍

> Werkzeug 是一个 **Python 的 WSGI（Web Server Gateway Interface）工具库**，它为开发者提供了构建 Web 应用所需的底层基础设施和实用工具。作为 Python Web 开发领域的重要工具，Werkzeug 以其灵活性、模块化设计和丰富的功能著称，广泛应用于 Flask 等知名框架的底层实现中

1.3 jinja介绍

> “Jinja” 是一个 **Python 的模板引擎**，通常指 **Jinja2**（即 Jinja 的第二代版本），它被广泛用于 Web 开发中动态生成 HTML、XML 或其他文本格式的文件。Jinja2 的设计灵感来源于 Django 模板语言，但功能更强大且更灵活，是 Flask 等 Python Web 框架的默认模板引擎

1.4 一些协议

1. **WSGI** (Web Server Gateway Interface)

- **定义**：

> WSGI是Python官方定义的同步接口标准，用于规范Web服务器与Web应用程序/框架之间的通信。

- **核心作用**：
  - 解耦Web服务器（如Nginx）和Web框架（如Django、Flask）。
  - 提供统一的接口，使不同的服务器和框架可以自由组合。
- **特点**：
  - 同步模式：每个请求按顺序处理，适合传统的HTTP请求-响应模型。
  - 中间件支持：通过WSGI中间件可以扩展功能（如身份验证、日志记录）。
- 适用场景：
  - 同步框架（如Flask、Django的传统模式）。
  - 部署生产环境时，常与uWSGI或Gunicorn等服务器结合使用。

2. **uwsgi**

- **定义**：

> uwsgi是uWSGI服务器自定义的二进制通信协议，用于服务器与Web框架之间的高效数据传输。

- **核心作用**：
  - 作为uWSGI服务器与前端服务器（如Nginx）之间的通信协议。
  - 相比HTTP协议，uwsgi更高效，支持更多功能（如进程间通信、文件传输）。
- **特点**：
  - 二进制协议：比文本协议（如HTTP）更轻量，传输速度更快。
  - 灵活性：支持多种传输方式（TCP、Unix Socket等）。

- **与uWSGI的关系**：
  - uWSGI是实现了WSGI协议的服务器软件。
  - uwsgi是uWSGI服务器内部使用的协议，用于与前端服务器通信。

3. **ASGI** (Asynchronous Server Gateway Interface)

- **定义**：

> ASGI是WSGI的异步扩展，用于处理现代Web中的高并发和实时通信需求（如WebSocket、HTTP/2）。

- **核心作用**：
  - 支持异步编程模型（基于Python的asyncio）。
  - 兼容传统HTTP请求和实时通信协议（如WebSocket）。
- **特点**：
  - 异步模式：可同时处理多个请求，适合高并发场景。
  - 协议兼容性：支持HTTP、HTTP/2、WebSocket等。
- **适用场景**：
  - 异步框架（如FastAPI、Starlette）。
  - 实时应用（如聊天室、在线游戏、实时数据推送）。

4. **uWSGI**

- **定义**：

> uWSGI是一个高性能的Web服务器/应用服务器，支持WSGI、ASGI和uwsgi协议。

- **核心作用**：
  - 运行Python Web应用（如Django、Flask）。
  - 通过uwsgi协议与Nginx等前端服务器通信。
- **特点**：
  - 支持多种协议：WSGI、ASGI、HTTP、FastCGI等。
  - 高度可配置：支持进程/线程管理、缓存、负载均衡、动态扩展等。
  - 生产环境常用：适合部署高流量的Web应用。

二、核心使用

2.1 路由

> 一种是使用默认route， 一种是使用蓝图来注册理由

2.1.1 route：

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

2.1.2 蓝图

- **为啥使用蓝图**
  - 拆分应用，可以分模块注册，非常的灵活，是一个针对大模型的理想方案
  - 可以使用同一个URL前缀或者子域，不需要手动添加
  - 蓝图支持定义请求钩子（如 `before_request`）和错误处理函数（如 `@bp.errorhandler(404)`
- **使用**
  - **创建蓝图**

```
from flask import Blueprint, render_template, redirect, url_for

# 创建蓝图对象，名称为 'auth'
blog_bp = Blueprint('auth', __name__,
                    template_folder='templates',  # 模板路径（相对于当前文件）
                    static_folder='static') # 静态文件路径

@auth_bp.route('/login')
def login():
    return render_template('auth/login.html')

@auth_bp.route('/logout')
def logout():
    return redirect(url_for('auth.login'))

@auth_bp.route('/register')
def register():
    return render_template('auth/register.html')
```

  - **注册蓝图**

```
# 文件: app.py
from flask import Flask
from auth.routes import auth_bp

app = Flask(__name__)

# 注册蓝图，设置 URL 前缀为 '/auth'
app.register_blueprint(auth_bp, url_prefix='/auth')

if __name__ == '__main__':
    app.run(debug=True)
```

  - **错误处理**

```
@auth_bp.errorhandler(404)
def auth_not_found(e):
    return render_template('auth/404.html')
```

2.1.3 路由变量规则

- **概念**

> 通过把 URL 的一部分标记为 `<variable_name>` 就可以在 URL 中添加变量。标记的 部分会作为关键字参数传递给函数。通过使用 `<type:variable_name>` ，可以 选择性的加上一个转换器，为变量指定规则

- **例子**

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % escape(username)

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return 'Subpath %s' % escape(subpath)
```

- **规则类型**

类型	含义
`string`	（缺省值） 接受任何不包含斜杠的文本
`int`	接受正整数
`float`	接受正浮点数
`path`	类似 `string` ，但可以包含斜杠
`uuid`	接受 UUID 字符串

2.1.4 设置HTTP方法

> 通过route函数methods 方法指定

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

2.1.5 URL重定向

- **最基本重定向**

> 使用 `redirect()` 函数可以将用户重定向到指定的 URL

```
from flask import Flask, redirect

app = Flask(__name__)

# 外部路由
@app.route('/external')
def external_redirect():
    return redirect('https://www.baidu.com')

# 内部路由
@app.route('/')
def index():
    return 'Hello, World!'

@app.route('/internal')
def internal_redirect():
    return redirect(url_for('index'))  # 重定向到 index 路由
```

- **动态传参**

```
@app.route('/user/<username>')
def user_profile(username):
    return f'User: {username}'

@app.route('/redirect-to-user')
def redirect_to_user():
    return redirect(url_for('user_profile', username='Alice'))
```

- **重定向到上一个页面**

```
from flask import request

@app.route('/back')
def go_back():
    return redirect(request.referrer or url_for('index'))
```

- **注意**

> 使用蓝图的时候重定向，url_for('bpname.viewfunc'),bpname注册蓝图名称，viewfunc指该蓝图的视图函数

示例

```
from flask import Blueprint, render_template, redirect, url_for

# 创建蓝图对象，名称为 'auth'
blog_bp = Blueprint('auth', __name__,
                    template_folder='templates',  # 模板路径（相对于当前文件）
                    static_folder='static') # 静态文件路径
```

```python
@auth_bp.route('/login')
def login():
    return render_template('auth/login.html')

@auth_bp.route('/logout')
def logout():
    return redirect(url_for('auth.login'))
```

2.1.6 错误处理

> 使用 redirect()函数可以重定向到错误页面。也可以使用 abort()可以 更早退出请求，并返回错误代码

```python
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

2.2 操作请求数据

> Flask 中由全局 对象 request来提供请求信息

2.2.1 request全局对象是怎么保证线程安全的?

> Flask 的全局对象（如 `request`、`g`）并不是真正的全局变量，而是 `LocalProxy` **代理对象**

- **Local，LocalStack，LocalProxy介绍**
  - **Local**

    > local 是一个线程/上下文隔离的容器，用于存储于当前线程或上下文相关的数据。每个flask的请求上下文都有自己打的独立副本，不会相互干扰

    ```python
    from werkzeug.local import Local

    local = Local()
    local.user = "Alice"  # 当前线程/上下文的 user 值为 "Alice"

    def another_context():
        local.user = "Bob"  # 在另一个线程/上下文中，user 值为 "Bob"
        print(local.user)  # 输出: Bob

    another_context()
    print(local.user)  # 输出: Alice（主线程的值未变）
    ```

  - **LocalStack**

> LocalStack 是一个基于Local的栈结构，运行咋系统一个线程/上下文中管理多个值。通过LIFO（后进先出）规则管理数据

```
from werkzeug.local import LocalStack

stack = LocalStack()
stack.push("A")
stack.push("B")
print(stack.top)  # 输出: B

stack.pop()
print(stack.top)  # 输出: A
```

- **LocalProxy**

> LocalProxy 是一个代理对象，用于透明访问Local或LocalStack中的数据。简化了对隔离数据的访问，避免显示操作Local和LocalStack

```
from werkzeug.local import Local, LocalProxy

local = Local()
local.user = "Alice"

# 创建 LocalProxy 代理
user = LocalProxy(lambda: local.user)

print(user)  # 输出: Alice
```

- **总结**

> 每次有新的请求到达时，Flask 会创建一个新的请求上下文，并将这个上下文推入一个栈中。由于使用了 `LocalStack`，每个线程都有自己的栈实例，因此每个线程只能访问到属于自己的那个请求上下文。这样就确保了不同线程之间的请求不会相互干扰

2.2.2 request的具体操作

- **获取请求方法和查询参数**

```
@app.route('/method', methods=['GET', 'POST'])
def show_method():
    return f'This request was made using {request.method}'

@app.route('/query')
def get_query():
    name = request.args.get('name', 'Anonymous')  # 获取参数，若不存在则返回默认值
    age = request.args.get('age', 0, type=int)    # 指定类型转换
    return f'Hello, {name}! Your age is {age}.'
```

- **获取表单数据**

```python
@app.route('/form', methods=['POST'])
def handle_form():
    username = request.form.get('username')
    password = request.form.get('password')
    return f'Username: {username}, Password: {password}'
```

- **获取 JSON 数据**

> 客户端发送的是 JSON 数据（`Content-Type: application/json`），可以通过 `request.get_json()` 或 `request.json` 获取

```python
@app.route('/api/data', methods=['POST'])
def api_data():
    data = request.get_json()  # 或直接使用 request.json
    return f'Received data: {data["name"]}, {data["age"]}'
```

- **处理文件上传**

> 文件上传通过 `multipart/form-data` 编码实现

```python
@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        return 'No file uploaded.'
    file = request.files['file']
    if file.filename == '':
        return 'No file selected.'
    file.save(f'uploads/{file.filename}')  # 保存文件到指定路径
    return 'File uploaded successfully!'
```

- **获取请求头**

```python
@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')  # 获取 User-Agent
    return f'Your User-Agent is {user_agent}'
```

- **获取cookie**

> 户端发送的 Cookie 存储在 `request.cookies` 中

```python
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    return {}
```

2.3 操作响应数据

> 在flask中，响应对象(Response)是通过视图函数返回给客户端的数据结构，flask会根据视图函数的返回值自动将其转换为合法的HTTP响应对象

响应对象的类型

返回值类型	`Content-Type`	示例
字符串	`text/html`	`return "Hello"`
字典	`application/json`	`return {"key": "value"}`
`jsonify` 返回值	`application/json`	`return jsonify({"key": "value"})`
HTML 模板	`text/html`	`return render_template("index.html")`
流式数据	指定的 `mimetype`	`return Response(generate(), mimetype='text/plain')`
文件	根据文件类型自动推断	`return send_file("file.txt")`

2.4 session（会话）

2.4.1 核心机制

- **基于 Cookie 的实现**：
  - **Session ID**：Flask 为每个用户生成唯一的 Session ID（通过 `os.urandom` 或 `secrets` 模块生成），存储在客户端的 Cookie 中。
  - **加密存储**：Session 数据（如字典）在服务器端被加密后，以 Base64 编码的形式嵌入 Cookie 中。例如，`session['user_id'] = 123` 会被加密为类似 `abc123xyz` 的字符串存储在 Cookie 中。
  - **签名验证**：每次请求时，Flask 会验证 Cookie 中的 Session 数据是否被篡改（通过 `secret_key` 加密签名）
- **服务器端存储**（可选）：
  - 默认情况下，Session 数据加密后存储在客户端的 Cookie 中。但可以通过 **Flask-Session** 扩展，将 Session 数据存储在服务器端（如 Redis、数据库等），进一步提升安全性。

2.4.2 基本操作

```python
from flask import Flask, session

# 初始化
app = Flask(__name__)
app.secret_key = 'slsdaljdljdlj'  # 设置加密密钥（必须）

# 设置Session
@app.route('/login')
def login():
    session['user_id'] = 123  # 存储用户ID
    session['username'] = 'john_doe'  # 存储用户名
    return 'Logged in!'
```

```python
# 获取Session
@app.route('/profile')
def profile():
    user_id = session.get('user_id')  # 安全获取数据
    if user_id:
        return f'Welcome, user {user_id}!'
    return 'Not logged in.', 403

#删除Session
@app.route('/logout')
def logout():
    session.pop('user_id', None)  # 删除指定键
    session.clear()  # 清除所有 Session 数据
    return 'Logged out!'
```

2.4.3 session的高级配置

- **设置session持久化时间**

```python
from datetime import timedelta

app.permanent_session_lifetime = timedelta(days=7)  # 设置为7天
```

- 使用flas-sesion存储到服务端

```python
from flask import Flask
from flask_session import Session
import redis

app = Flask(__name__)
app.config['SESSION_TYPE'] = 'redis'  # 使用 Redis 存储
app.config['SESSION_REDIS'] = redis.from_url('redis://localhost:6379')  #
Redis 连接
Session(app)
```

2.4.4 flash 消息闪现

> flask提供了一个非常有用的flash()函数，它可以用来“闪现”需要提示给用户的消息，比如当用户登录成功后显示“欢迎回来！”。在视图函数调用flash()函数，传入消息内容，flash（）函数把消息存储在session中，我们需要在模板中使用全局函数get_flashed_messages()获取消息并将它显示出来

- 视图

> flash提交数据

```python
from flask import flash, redirect, url_for

@app.route('/submit', methods=['POST'])
def submit():
    # 模拟表单验证
    if success:
        flash('操作成功! ', category='success')  # 添加成功消息
    else:
        flash('操作失败，请重试。', category='error')  # 添加错误消息
    return redirect(url_for('result'))  # 重定向到目标页面
```

- 模板

> 获取消息

```python
<!-- 基模板 base.html -->
{% for category, message in get_flashed_messages(with_categories=True) %}
  <div class="alert alert-{{ category }}">
    {{ message }}
  </div>
{% endfor %}
```

2.5 信号（signals）

2.5.1 核心概念

> flask的signals是基于事件驱动的通信模型（使用时需要安装 Blinke），注册一个signals，然后有函数绑定这个signals，触发这个信号，会自动执行这个函数，默认是同步执行，除非搭配celery。可以在不修改核心逻辑的情况下插入额外功能（如日志记录、异常处理、缓存更新等）。合理使用信号可以显著提升代码的可维护性和灵活性，但也需注意其同步执行的局限性

- **事件触发**

> 当 Flask 应用中的某个事件发生时（如请求开始、模板渲染完成、异常抛出等），会自动触发一个信号

- **信号订阅**

> 开发者可以定义一个函数（称为信号处理函数），并将其绑定到某个信号上。当信号触发时，所有订阅该信号的处理函数会被自动调用

- **解耦与灵活性**

> 信号机制将事件的触发者（如 Flask 框架）和响应者（如开发者自定义的函数）解耦，使得代码更模块化、可维护性更高

2.5.2 flask信号用途

1. 常见内置信号

- **日志记录**

> 在请求结束后记录日志

```python
from flask import request_finished

def log_request(sender, response, **extra):
    print(f"Request finished with status {response.status_code}")

request_finished.connect(log_request, app)
```

- **异常处理**

> 在请求异常时记录错误信息

```python
from flask import got_request_exception

def log_exception(sender, exception, **extra):
    app.logger.error(f"An exception occurred: {exception}")

got_request_exception.connect(log_exception, app)
```

- **模板渲染监控**

> 在模板渲染后执行操作（如统计页面访问量）：

```python
from flask import template_rendered

def track_template_rendering(sender, template, context, **extra):
    print(f"Rendered template: {template.name}")

template_rendered.connect(track_template_rendering, app)
```

- **消息闪现**

> 在调用 `flash()` 函数时触发：

```python
from flask import message_flashed

def log_flash_message(sender, message, category, **extra):
    print(f"Flashed message: {message} (Category: {category})")

message_flashed.connect(log_flash_message, app)
```

2. 自定义信号

> 开发者可以定义自己的信号，用于跨模块通信或业务逻辑解耦：

- **定义信号**

```python
from flask import signals

# 创建一个自定义信号
user_registered = signals.signal('user-registered')
```

- **绑定处理函数**

```
def send_welcome_email(sender, user, **extra):
    print(f"Sending welcome email to {user.email}")

user_registered.connect(send_welcome_email, app)
```

- **触发信号**

```
# 在用户注册后发送信号
user_registered.send(app, user=new_user)
```

2.6 上下文

2.6.1 请求上下文

> Flask 在处理请求时，会自动 推入 请求上下文。视图函数，错误处理钩子函数与其他在请求当中运行的函数可以访问指向当前请求的请求对象代理对象 `request`，Flask 使用了 `request`与 `session`代理对象来访问请求对象, 简化了操作

1.生命周期

> 请求到达->before_request->view->after_request->销毁，概况来说就是请求到达就自动创建了request上下文，请求处理完成了就销毁request上下文

2.手动创建

> 通过`test_request_context()`

```
#删除Session
@app.route('/logout')
def logout():
    session.pop('user_id', None)  # 删除指定键
    session.clear()  # 清除所有 Session 数据
    return 'Logged out!'

with app.test_request_context("/logout"):
    logout()
```

3.核心对象

> request，session

```
@app.route('/logout')
def logout():
    print(request.headers)
    session.pop('user_id', None)  # 删除指定键
    session.clear()  # 清除所有 Session 数据
    return 'Logged out!'
```

2.6.2 应用上下文

> 应用上下文在请求、命令行命令以及其他活动期间保持对应用层面数据的追踪。相比起将应用上下文传递到每个函数当中，Flask 使用了 `current_app`和 `g` 这两个代理对象来访问应用上下文

1.生命周期

> 请求到达->before_request->view->after_request->弹出，概况来说就是请求到达就自动创建了application上下文，请求处理完成了就弹出application上下文，通常来说，一个应用上下文与请求上下文拥有一致的生命周期。

2.手动推入

> 使用`app_context()`才能提前使用

```
def create_app():
    app = Flask(__name__)

    with app.app_context():
        init_db()

    return app
```

3.核心对象

> g和current_app

2.7 生命周期

1.简化步骤

- wsgi ,请求封装后给web后台应用
- 创建请求和应用上下问
- 中间件
- 路由处理，交给视图函数
- 视图函数处理
- 创建响应体
- 中间件
- 清理请求上下文和应用上下问

2.详细步骤

- WSGI 服务器调用 Flask 对象，该对象调用 .
- 将创建一个对象。这会将 WSGI dict 转换为对象。它还会创建一个对象。environAppContext
- 应用程序上下文被推送，
- 信号已发送
- 请求上下文被推送，这 makes 和 available。
- 会话将打开，并使用应用程序的 、 的实例加载任何现有会话数据。
- 该 URL 将与在应用程序设置期间向装饰器注册的 URL 规则进行匹配。如果没有匹配项，则错误 - 通常是 404， 405 或 redirect - 被存储以供以后处理。
- 信号已发送。
- 将调用任何修饰的函数。
- 将调用任何修饰的函数。如果 这些函数返回一个值，它会立即被视为响应。
- 如果 URL 在几个步骤前与路由不匹配，则现在会引发该错误。
- 与匹配的 URL 关联的修饰视图函数 ， 并返回要用作响应的值。

- 如果到目前为止任何步骤引发了异常，并且存在与异常类或 HTTP 错误代码匹配的修饰函数，则为调用以处理错误并返回响应。
- 返回响应值的任何内容 - before request 函数、视图或 error 处理程序，该值将转换为 Object。
- 调用任何修饰的函数，然后清除。
- 任何装饰函数都会被调用，这些函数可以修改 响应对象。
- 会话将被保存，并使用应用程序的 .
- 信号已发送。
- 如果到目前为止任何步骤引发了异常，并且它未由错误处理程序处理 函数，它现在已处理。HTTP 异常被视为响应，其 对应的状态码，其他异常将转换为通用的 500 响应。 信号已发送。
- 响应对象的 status、headers 和 body 将返回到 WSGI 服务器。
- 将调用任何修饰的函数。
- 信号已发送。
- 请求上下文已弹出，并且不再是 可用。
- 将调用任何修饰的函数。
- 信号已发送。
- 应用程序上下文已弹出，并且不再 可用。
- 信号已发送。

2.8 请求钩子

- `before_first_request`

  > 第一次请求到达的时候触发，用于一次性初始操作

- `before_request`

  > 每次请求前触发(在路由匹配后、视图函数执行前)，一般用于验证，限流

- `after_request`

  > 每次请求后执行（仅在视图函数返回响应，且没有发生异常后执行），对响应添加处理

- `teardown_request`

  > 每个请求结束后（无论是否发生异常），可以用来如关闭数据库连接、释放锁

```
from flask import Flask, request, session, g

app = Flask(__name__)

@app.before_first_request
def before_first_request():
    print("第一个请求前执行：初始化数据库连接")

@app.before_request
def before_request():
    # 登录验证
    if 'user_id' not in session and request.endpoint != 'login':
```

```
        return redirect('/login')
    # 记录请求时间
    g.start_time = time.time()

@app.after_request
def after_request(response):
    # 添加响应头
    response.headers['X-Processed-By'] = 'Flask'
    return response

@app.teardown_request
def teardown_request(exception):
    # 关闭数据库连接
    db.close()
```

2.9 常用扩张库

2.9.1 flask-sqlalchemy

> ORM连接数据库

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)

@app.route('/')
def index():
    user = User.query.filter_by(username='admin').first()
    return f'Hello, {user.username}!'

if __name__ == '__main__':
    db.create_all()  # 创建数据库表
    app.run()
```

2.9.2 flask-login

> 用户认证和会话管理功能

```
from flask import Flask, redirect, url_for, render_template
from flask_login import LoginManager, UserMixin, login_user, login_required,
logout_user

app = Flask(__name__)
app.secret_key = 'your-secret-key'
login_manager = LoginManager(app)

class User(UserMixin):
    def __init__(self, id):
        self.id = id
```

```python
@login_manager.user_loader
def load_user(user_id):
    return User(user_id)

@app.route('/login', methods=['POST'])
def login():
    user = User(1)  # 模拟用户
    login_user(user)
    return redirect(url_for('protected'))

@app.route('/protected')
@login_required  # 限制未登录用户访问
def protected():
    return 'Logged in!'

@app.route('/logout')
def logout():
    logout_user()
    return 'Logged out!'

if __name__ == '__main__':
    app.run()
```

2.9.3 flask-wtf

> 处理表单验证和 CSRF 保护，简化表单创建和验证

```python
from flask import Flask, render_template, redirect, url_for
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Login')

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        return f'Logged in as {form.username.data}'
    return render_template('login.html', form=form)

if __name__ == '__main__':
    app.run()
```

2.9.4 Flask-RESTful

> 构建 RESTful 风格的 API，简化资源管理

```
from flask import Flask
from flask_restful import Api, Resource, reqparse

app = Flask(__name__)
api = Api(app)

# 模拟数据库
users = {
    1: {'name': 'Alice'},
    2: {'name': 'Bob'}
}

class User(Resource):
    def get(self, user_id):
        return users[user_id], 200

    def post(self, user_id):
        parser = reqparse.RequestParser()
        parser.add_argument('name')
        args = parser.parse_args()
        users[user_id] = {'name': args['name']}
        return users[user_id], 201

api.add_resource(User, '/user/<int:user_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

2.9.5 flask-admin

> 快速构建功能强大的后台管理界面

```
from flask import Flask
from flask_admin import Admin
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
db = SQLAlchemy(app)

# 初始化 Flask-Admin
admin = Admin(app, name='My Admin', template_mode='bootstrap3')

# 定义数据库模型
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

# 创建数据库表
with app.app_context():
    db.create_all()
```

```
# 注册模型视图
from flask_admin.contrib.sqla import ModelView
admin.add_view(ModelView(User, db.session))

if __name__ == '__main__':
    app.run(debug=True)
```

2.9.6 flask-session

> 添加服务器存储session

```
from flask import Flask
from flask_session import Session
import redis

app = Flask(__name__)
app.config['SESSION_TYPE'] = 'redis'  # 使用 Redis 存储
app.config['SESSION_REDIS'] = redis.from_url('redis://localhost:6379')  # Redis 连接
Session(app)
```

2.9.7 Flask-CORS

> 解决跨域资源共享（CORS）问题，允许跨域请求

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)  # 允许所有域名跨域访问

@app.route('/')
def hello():
    return 'Hello from a CORS-enabled endpoint!'

if __name__ == '__main__':
    app.run()
```

2.10 web常见的问题

2.10.1 XSS（跨站脚本攻击）防御

1.概念

> XSS 攻击通过向网页注入恶意脚本，窃取用户信息或破坏页面功能

2.解决办法

- **1. 使用 Jinja2 模板自动转义**

  > Flask 默认使用 **Jinja2 模板引擎**，它会自动对变量进行 HTML 转义，防止恶意脚本执行。

```
# 示例：自动转义用户输入
@app.route('/')
def index():
    user_input = "<script>alert('XSS')</script>"
    return render_template('index.html', user_input=user_input)
```

```
<!-- Jinja2 会自动转义 HTML 标签 -->
<p>{{ user_input }}</p>
```

- **2. 手动转义**

> 前段手动转义

- **3. 输入验证与过滤**

> 对用户输入进行严格验证，拒绝非法内容。

```
from flask import request

@app.route('/submit', methods=['POST'])
def submit():
    user_input = request.form['user_input']
    if not is_valid_input(user_input):  # 自定义验证函数
        return "Invalid input", 400
    return "Input received", 200

def is_valid_input(input):
    # 限制输入长度、正则匹配等
    return len(input) < 100 and not re.search(r'<script>', input)
```

- **4. 设置 Content Security Policy (CSP)**

> 通过 CSP 限制页面加载的资源，减少 XSS 风险。

```
@app.after_request
def apply_csp(response):
    response.headers['Content-Security-Policy'] = "default-src 'self';
script-src 'self'"
    return response
```

- **5. 使用 `Markup` 类标记安全内容**

```
from flask import Markup

@app.route('/')
def index():
    user_input = "<strong>Safe content</strong>"
    return render_template('index.html', user_input=Markup(user_input))
```

2.10.2 CORS（跨域资源共享）配置

1.概念

> CORS 攻击源于浏览器的同源策略被绕过，Flask 可通过 `flask-cors` 扩展轻松处理跨域请求

2.解决办法

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)  # 允许所有路由的跨域请求

@app.route('/api/data')
def get_data():
    return jsonify({'data': 'Hello, CORS!'})
```

2.10.3 CSRF（跨站请求伪造）防护

> CSRF 攻击利用用户身份发起恶意请求，Flask 可通过 `Flask-WTF` 扩展实现防护

```
from flask import Flask, request, session
from flask_wtf.csrf import CSRFProtect

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'  # 必须设置密钥
csrf = CSRFProtect(app)
```

表单

```
<form method="POST">
    {{ form.csrf_token }}
    <!-- 其他表单字段 -->
    <input type="submit" value="Submit">
</form>
```

2.11 三种会话管理

2.11.1 cookie

> 用cookies来实现会话管理时，用户的相关信息或者其他我们想要保持在每个请求中的信息，都是放在cookies中,而cookies是由客户端来保存，每当客户端发出新请求时，就会稍带上cookies,服务端会根据其中的信息进行操作

- **优点**：
  - **简单易用**：浏览器原生支持，开发成本低。
  - **自动携带**：无需手动处理，适合传统多页应用（MPA）。
  - **安全性**：可通过 `HttpOnly`、`Secure` 和 `SameSite` 标志防止 XSS 和 CSRF。
- **缺点**：
  - **跨域限制**：受同源策略限制，跨域场景需配置 CORS。
  - **存储大小限制**：单个 Cookie 通常不超过 4KB。
  - **依赖服务器**：需服务器维护 Session 数据，增加资源消耗

2.11.2 session

> 当用户登录后，服务器会生成一个唯一的 Session ID，并通过 **Cookie** 将其发送到客户端。后续请求中，浏览器会自动将该 Cookie（包含 Session ID）发送回服务器，服务器通过 Session ID 找到对应的 Session 数据

- **优点**：
  - **安全性高**：敏感数据存储在服务器，仅传输 Session ID。
  - **灵活控制**：可随时修改或销毁会话（如强制登出）。
  - **兼容性强**：适合传统 Web 应用和需要频繁交互的场景。
- **缺点**：
  - **有状态设计**：需服务器存储会话数据，扩展性差（需分布式 Session 管理）。
  - **跨域问题**：跨域场景需额外配置（如 CORS 或 SSO）。
  - **性能开销**：服务器需频繁查询会话数据，可能成为瓶颈

2.11.3 jwt

> 首先用户发出登录请求，服务端根据用户的登录请求进行匹配，如果匹配成功，将相关的信息放入payload中，利用算法，加上服务端的密钥生成token，这里需要注意的是secret_key很重要，如果这个泄露的话，客户端就可以随机篡改发送的额外信息，它是信息完整性的保证。生成token后服务端将其返回给客户端，客户端可以在下次请求时，将token一起交给服务端，一般是说我们可以将其放在Authorization首部中，这样也就可以避免跨域问题

- **优点**：
  - **无状态**：服务器无需存储会话数据，适合分布式系统和微服务。
  - **跨域支持**：可通过 HTTP 头传递 Token，天然支持跨域。
  - **自包含性**：Token 包含用户信息，减少数据库查询。
  - **移动端友好**：适合单页应用（SPA）和移动端接口。
- **缺点**：
  - **安全性风险**：Token 存储在客户端，易受 XSS 攻击（需配合 `HttpOnly` 和 `Secure`）。
  - **无法主动失效**：Token 一旦签发，需等到过期或使用黑名单机制（如 Redis 缓存）。
  - **性能开销**：Token 体积较大，增加请求头大小（尤其在频繁请求时）