1. http协议
2. **http四个特点**

**1)基于TCP/IP协议**

http协议是基于TCP/IP协议之上的应用层协议。

**2)基于请求－响应模式**

HTTP协议规定,请求从客户端发出,最后服务器端响应该请求并 返回。换句话说,肯定是先从客户端开始建立通信的,服务器端在没有 接收到请求之前不会发送响应

**3)无状态保存**

HTTP是一种不保存状态,即无状态(stateless)协议。HTTP协议 自身不对请求和响应之间的通信状态进行保存。也就是说在HTTP这个 级别,协议对于发送过的请求或响应都不做持久化处理。

使用HTTP协议,每当有新的请求发送时,就会有对应的新响应产 生。协议本身并不保留之前一切的请求或响应报文的信息。这是为了更快地处理大量事务,确保协议的可伸缩性,而特意把HTTP协议设计成 如此简单的。

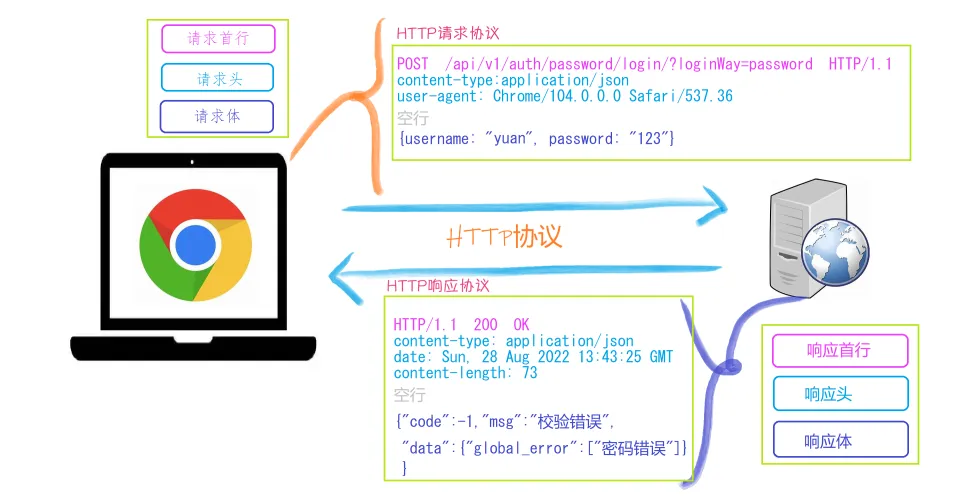
**4)短连接**

HTTP1.0默认使用的是短连接。浏览器和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。

HTTP/1.1起，默认使用长连接。要使用长连接，客户端和服务器的HTTP首部的Connection都要设置为keep-alive，才能支持长连接。

HTTP长连接，指的是复用TCP连接。多个HTTP请求可以复用同一个TCP连接，这就节省了TCP连接建立和断开的消耗。

1. **什么是请求头请求体,响应头响应体**



请求首行: 请求方式,请求路径,请求协议

请求头:

在发送请求到服务器时，请求头会包含诸如请求的类型（GET、POST等）、所需的响应格式、客户端可以接受的语言、Cookie等信息。这些信息帮助服务器更好地理解和处理请求。

请求头为服务器提供了额外的上下文信息，以便服务器可以根据这些信息来定制化响应。例如，通过Accept-Language请求头，服务器可以决定向用户返回英文或中文网页。通过User-Agent，服务器可以判断请求来自哪种类型的设备（手机、桌面计算机等），并据此提供适当的响应。

请求体:

存放核心数据,例如post提交表单的数据,**注意get请求没有请求体**,他的数据是挂载在URL地址?后面传递过去.

1. **URL地址包括什么**

一个完整的URL包括：协议、ip、端口、路径、参数

例如： [https://www.baidu.com/s?wd=yuan](https://www.baidu.com/s?wd=yuan" \t "_blank) 其中https是协议，www.baidu.com 是IP，端口默认80，/s是路径，参数是wd=yuan

1. **get请求和post请求是什么**

● GET提交的数据会放在URL之后，以?分割URL和传输数据，参数之间以&相连，如EditBook?name=test1&id=123456. POST方法是把提交的数据放在HTTP包的请求体中.

● GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制

1. **Content-Type是什么**

Content-Type 是一个 HTTP 头部，用于指示资源的 MIME 类型（媒体类型）。在 HTTP 请求中，当你发送数据给服务器（如通过 POST 或 PUT 请求）时，Content-Type 告诉服务器发送的数据是什么格式。同样，在 HTTP 响应中，它告诉客户端实际返回的内容类型是什么。这样，客户端就能正确地解析和处理这些数据。

Content-Type 格式通常包括类型和子类型，如 type/subtype，后面可以跟着参数，比如字符集。这里有一些常见的例子：

text/html; charset=UTF-8：表示发送的内容是 HTML 文本，并且字符集是 UTF-8。

application/json：表示发送的内容是 JSON 格式。

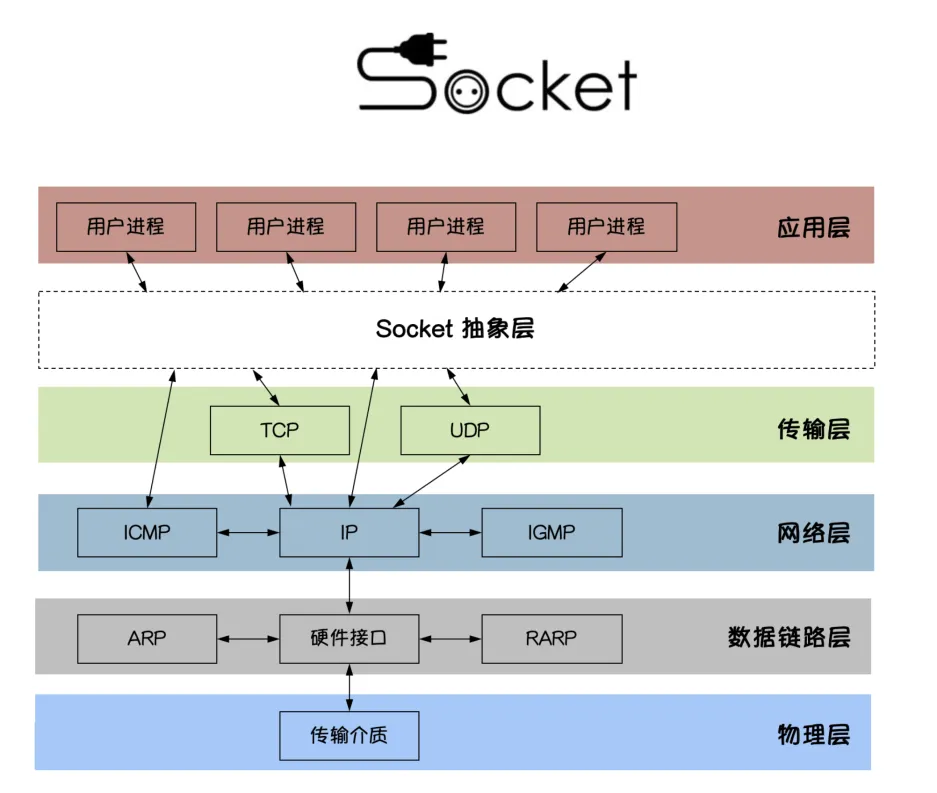
multipart/form-data：用于文件上传场景，允许发送表单数据和文件。

application/x-www-form-urlencoded：这是 HTML 表单提交时默认的编码类型，表单数据会被编码为键值对，类似于 URI 查询字符串。

服务器和客户端通过 Content-Type 头部相互沟通，确保发送和接收的内容以正确的方式被处理。例如，如果服务器返回的 Content-Type 是 application/json，客户端就会预期接收到的内容是 JSON 格式，并可以使用 JSON 解析器来处理这些数据。

通俗来说,content-type是告诉服务器应该用何种数据格式进行反解出原始数据.

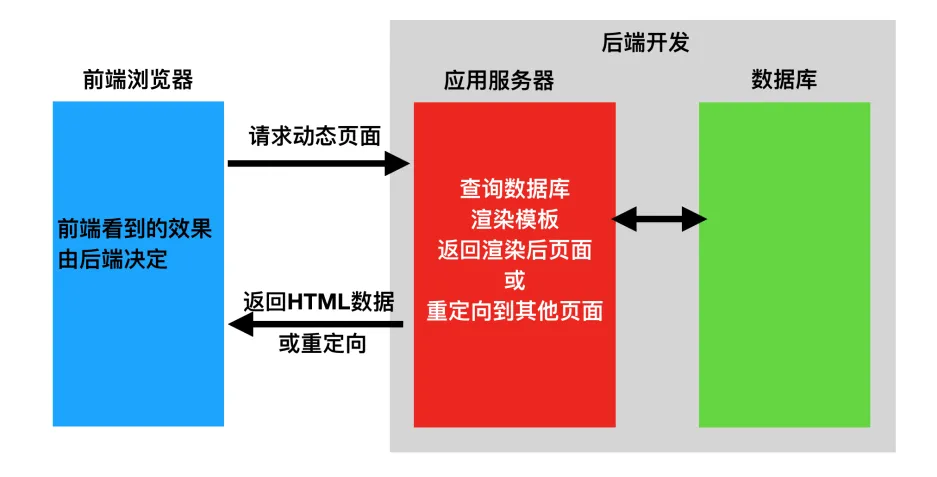
1. Socket



1. 什么是前后端分离

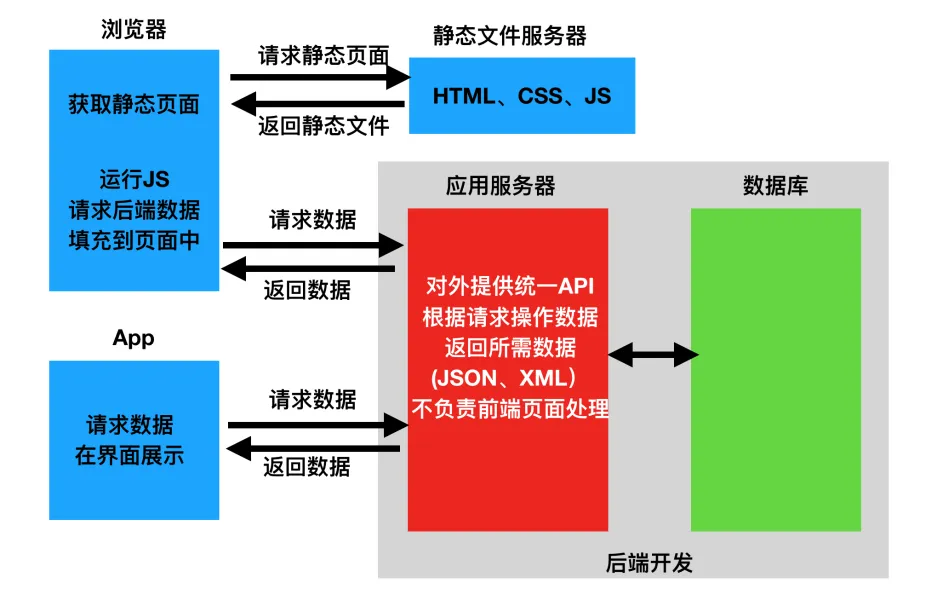
**前后端不分离模式 [客户端看到的内容和所有界面效果都是由服务端提供出来的。]**

在传统的网站开发模式中，前端和后端代码通常紧密结合在一起，服务器不仅处理数据和业务逻辑，还负责生成和返回HTML页面。客户端（浏览器）主要负责展示从服务器接收到的HTML内容。这种模式下，前后端的修改和更新往往需要同时进行，相互之间的依赖性很高。



**前后端分离模式 [把前端的界面效果(html，css，js分离到另一个服务端，python服务端只需要返回数据即可)]**

前后端分离后，前端变得独立，专注于用户界面和用户体验的构建，通常使用HTML、CSS和JavaScript等技术开发单页应用（SPA）。后端则专注于数据处理、业务逻辑的实现以及安全性等，通常通过RESTful API、GraphQL等接口与前端通信。



1. 什么是API?

应用程序编程接口（Application Programming Interface，API接口），就是应用程序对外提供了一个操作数据的入口，这个入口可以是一个函数或类方法，也可以是一个url地址或者一个网络地址。当客户端调用这个入口，应用程序则会执行对应代码操作，给客户端完成相对应的功能。

当然，api接口在工作中是比较常见的开发内容，有时候，我们会调用其他人编写的api接口，有时候，我们也需要提供api接口给其他人操作。由此就会带来一个问题，api接口往往都是一个函数、类方法、或者url或其他网络地址，不断是哪一种，当api接口编写过程中，我们都要考虑一个问题就是这个接口应该怎么编写？接口怎么写的更加容易维护和清晰，这就需要大家在调用或者编写api接口的时候要有一个明确的编写规范！！！

为了在团队内部形成共识、防止个人习惯差异引起的混乱，我们都需要找到一种大家都觉得很好的接口实现规范，而且这种规范能够让后端写的接口，用途一目了然，减少客户端和服务端双方之间的合作成本。

目前市面上大部分公司开发人员使用的接口实现规范主要有：restful、RPC。

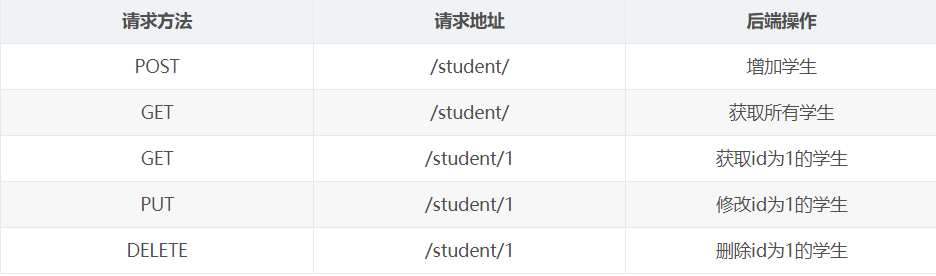
REST全称是Representational State Transfer，中文意思是表述（编者注：通常译为表征）性状态转移。 它首次出现在2000年Roy Fielding的博士论文中。

RESTful是一种专门为Web 开发而定义API接口的设计风格，尤其适用于前后端分离的应用模式中。

关键：面向资源开发

这种风格的理念认为后端开发任务就是提供数据的，对外提供的是数据资源的访问接口，所以在定义接口时，客户端访问的URL路径就表示这种要操作的数据资源。

对于数据资源分别使用POST、DELETE、GET、UPDATE等请求动作来表达对数据的增删查改。而不再是通过url的路径来区分,即student/add,student/delete等等.



restful规范是一种通用的规范，不限制语言和开发框架的使用。事实上，我们可以使用任何一门语言，任何一个框架都可以实现符合restful规范的API接口。

1. 什么是uvicorn

Uvicorn 是一个轻量级的ASGI服务器，用于运行Python的异步Web应用。它基于异步编程模型，可以高效处理大量并发请求，是运行像FastAPI这样的现代Web应用的理想选择。



命令行启动: uvicorn main:app --reload

直接方式:

if \_\_name\_\_ == '\_\_main\_\_':

import uvicorn

uvicorn.run("main:app", host="127.0.0.1", port=8080, debug=True, reload=True)

1. FastAPI的基本流程

（1）导入 FastAPI。

（2）创建一个 app 实例。

（3）编写一个路径操作装饰器（如 @app.get(“/”)）。

（4）编写一个路径操作函数（如上面的 def root(): …）

（5）定义返回值

（6）运行开发服务器（如 uvicorn main:app --reload）

1. 路径操作装饰器

**fastAPI支持的请求方式:**

@app.get()

@app.post()

@app.put()

@app.patch()

@app.delete()

@app.options()

@app.head()

@app.trace()

路径操作装饰器的参数:

**path:** 字符串，定义了路由的路径。例如，在 @app.get("/items") 中，"/items" 是路径。

**status\_code:** 整数，指定响应的HTTP状态码。默认情况下，大多数操作会返回 200，但你可以通过这个参数改变为其他状态码，例如 status\_code=201。

**tags:** 列表，用于组织API文档中的操作。通过为不同的路由设置标签，你可以在自动生成的文档中将它们分类。

**summary:** 字符串，提供操作的简短摘要，使API文档更加清晰。

**description:** 字符串，用于提供操作的详细描述。这比 summary 提供更多的信息，有助于理解API的用途。

**response\_model:** Pydantic模型，声明响应的格式。这告诉FastAPI预期的输出数据类型，使其能自动生成文档，并对输出数据进行验证。

**responses:** 字典，用于额外声明除默认外的其他响应类型和消息，允许你为不同的HTTP状态码定义响应模型。

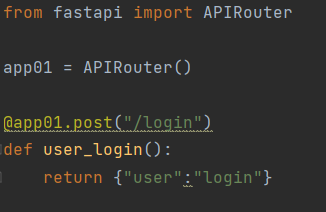
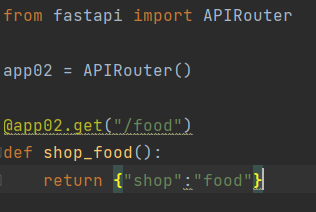
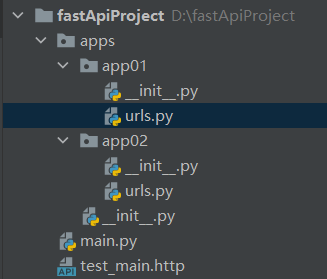
**dependencies:** 列表，包含依赖项。这些依赖项在路由函数执行前运行，可以用于请求前验证、权限检查等。

**deprecated:** 布尔值，如果设置为True，标记该操作为弃用。这对于向用户指示计划在将来版本中移除的API部分非常有用。

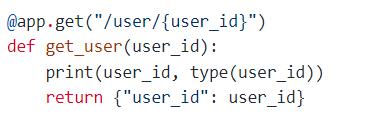
**operation\_id:** 字符串，用于为操作设置特定的ID。这在自动生成的API文档中用于唯一标识每个操作。

**methods:** 仅在 app.api\_route() 装饰器中使用，它允许你为同一个路径指定多个HTTP方法。

**路由分发**

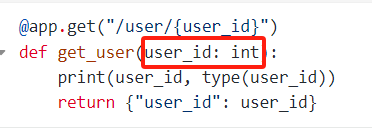


1. 请求与响应
2. **路径参数**



路径参数 user\_id 的值将作为参数 user\_id 传递给你的函数。请注意,在路径url中传递的任何参数给路径函数都是一个字符串.

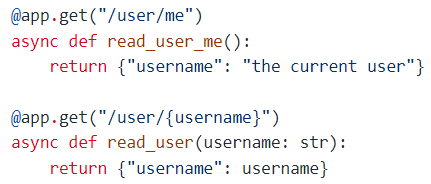
路径参数与位置参数一一对应.



在这个例子中，user\_id 被声明为 int 类型。这就将路径参数进行限定约束,将传过来的字符进行类型转换.

这将为你的函数提供编辑器支持，包括错误检查、代码补全等等。

**注意**



在创建路径操作时，你会发现有些情况下路径是固定的。

比如 /users/me，我们假设它用来获取关于当前用户的数据.

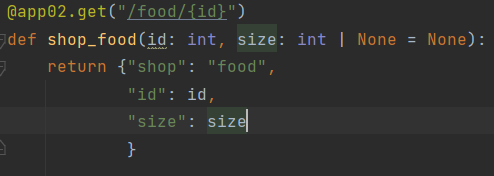
然后，你还可以使用路径 /user/{username} 来通过用户名 获取关于特定用户的数据。

由于路径操作是按顺序依次运行的(从上到下进行匹配)，你需要确保路径 /user/me 声明在路径 /user/{username}之前.否则，/user/{username} 的路径还将与 /user/me 相匹配，"认为"自己正在接收一个值为 "me" 的 username 参数。

1. **查询参数**

在路径函数中声明不属于路径参数的其他函数参数时，它们将被自动解释为"查询字符串"参数，就是 url? 之后用&分割的 key-value 键值对。

例如:xxx是路径参数,与路径函数中的位置参数xxx绑定,而其他的位置参数a,b就是查询参数.



表示size它的类型要么是int类型,要么是None,它有默认参数None.

另外size:int|None也可以写成size:Union[int,None]

其中的Union要导入typing模块,即form typing import Union

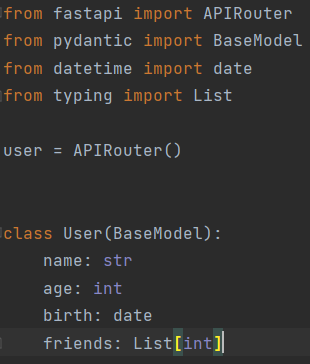
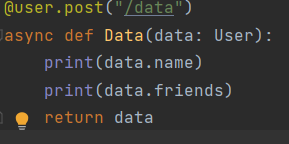
Optional 是Union的一个简化,当数据类型中有可能是None时,比如有可能是int也有可能是None，则Optional[int], 相当于Union[int, None]

Optional也是要导入的,即form typing import Optional

1. **请求体数据**

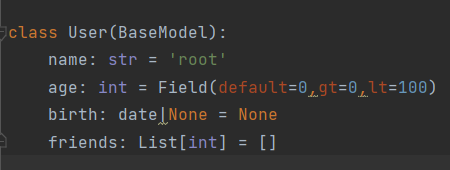
FastAPI 基于 Pydantic ，Pydantic 主要用来做类型强制检查（校验数据）。不符合类型要求就会抛出异常。

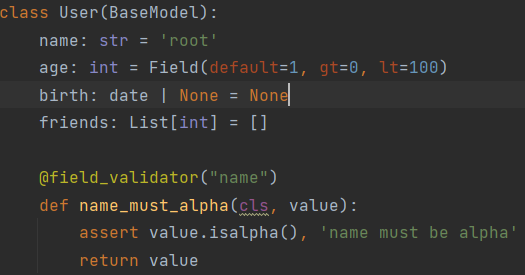
对于 API 服务，支持类型检查非常有用，会让服务更加健壮，也会加快开发速度，因为开发者再也不用自己写一行一行的做类型检查。

也就是说请求体的数据格式必须按照User类的形式.

扩展写法



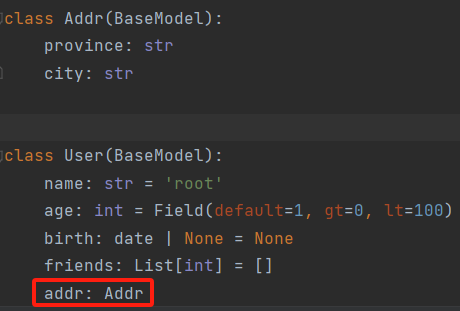


其中的@field\_validator()是对某个字段进行校验.

在 Pydantic 中，@field\_validator 是一个装饰器，用于对模型的一个或多个字段进行自定义验证。通过这个装饰器，你可以为指定的字段定义验证逻辑，如果输入数据不符合这些逻辑，可以抛出异常来指示验证失败。

使用 @field\_validator 时，你需要指定至少一个字段名，可以在类定义中直接将它应用于一个验证方法上。这个方法将接收字段的值作为输入，并根据你的逻辑对其进行检查。如果一切正常，它可以返回原始值或修改后的值；如果检查失败，应抛出一个异常。

@field\_validator 允许你详细控制数据验证过程，非常适合处理复杂的验证需求或当内置的验证逻辑不足以覆盖你的需求时。例如，你可以使用它来检查一个字符串字段是否包含特定的子字符串，或者验证一个数字是否在某个范围内。

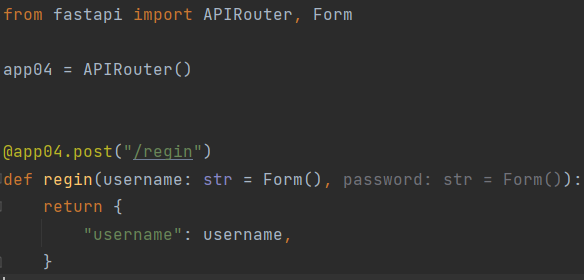
还可以进行嵌套.

注意:默认情况请求体格式是JSON格式

1. **Form表单数据**

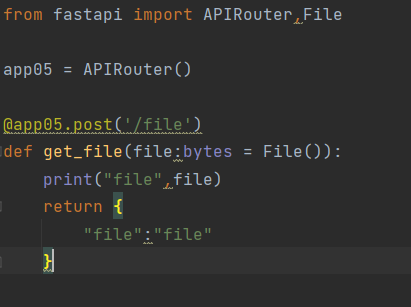
之前的pydantic是第三方包,不属于fastAPI的,与其是并列关系.

而Form组件是依赖于fastAPI的.

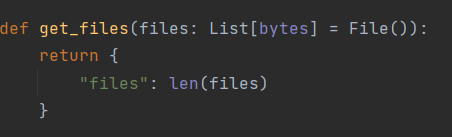


说明username,password是form表单其中的变量.

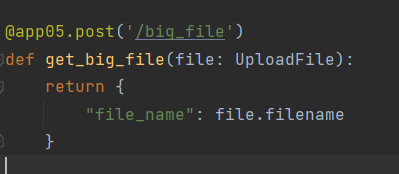
1. 文件上传



file是接受字节流的文件对象.



上面是多个文件.



File

File类型用于读取文件内容为 bytes或 str。

当你使用 File 作为操作函数的参数类型时，文件的内容将直接被读取到内存中。这适用于小文件，因为它们可以快速被加载处理。

使用 File 类型时，你不会接收到文件的元数据，比如文件名或者文件类型。

UploadFile

UploadFile 类型是 FastAPI 特有的，提供了更多的操作文件的方法，特别适用于大文件，因为它支持异步的文件写入和读取，可以减少内存消耗。

使用 UploadFile 时，你可以访问文件的元数据，如文件名（filename）、内容类型（content\_type）等。

UploadFile 提供了保存文件到磁盘的方法，这对于需要处理上传的文件并将它们保存在服务器上的场景非常有用。

它还支持异步的读写操作，对于不想阻塞主线程处理其他请求的同时处理文件上传的场景非常合适。

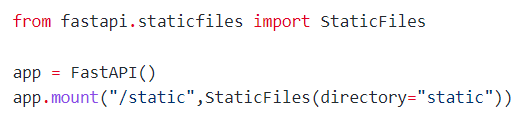
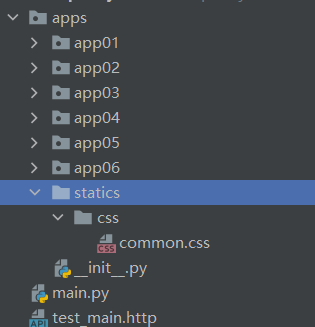
Pydantic校验只能对请求体数据进行校验,路径参数和查询参数不行.

1. Request对象

有些情况下我们希望能直接访问Request对象。例如我们在路径操作函数中想获取客户端的IP地址，需要在函数中声明Request类型的参数，FastAPI 就会自动传递 Request 对象给这个参数，我们就可以获取到 Request 对象及其属性信息，例如 header、url、cookie、session 等。



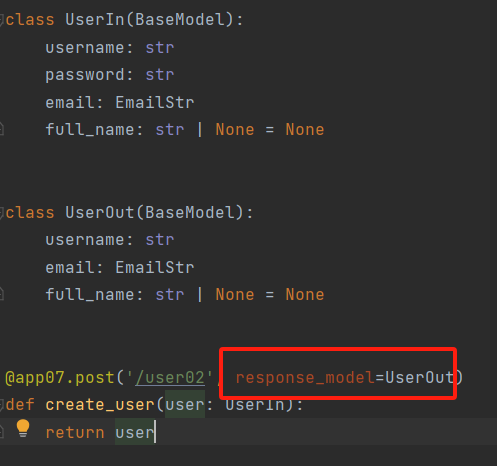
1. 请求静态文件



通过url:127.0.0.1:8080/static/css/common.css访问到静态资源.开放一个窗口给前端访问静态资源.

1. 响应模型
2. **response\_model**

FastAPI 提供了 response\_model 参数，声明 return 响应体的模型,注意response\_model 是路径操作的参数，并不是路径函数的参数.



在 FastAPI 中，response\_model 是一个非常重要的特性，它允许你定义操作函数（也称为路由函数）的响应模型。这个响应模型用于：

数据转换：自动将你的输出数据转换（或序列化）为定义的模型格式。这意味着，无论你的内部数据结构如何，最终返回给客户端的都将是符合 response\_model 定义的格式。

数据验证：在数据被发送给客户端之前验证数据符合模型的定义。这有助于确保返回的数据是有效和一致的，增加了API的健壮性。

输出限制：限制返回数据的字段。如果你的模型中包含了不希望暴露给客户端的敏感数据或内部信息，response\_model 可以帮助你仅仅返回那些你希望返回的数据。例如密码password

1. **response\_model\_exclude\_unset**

排除未设置的值.

使用路径操作装饰器的 response\_model 参数来定义响应模型，特别是确保私有数据被过滤掉。使用 response\_model\_exclude\_unset 来仅返回显式设定的值。

除了response\_model\_exclude\_unset以外，还有response\_model\_exclude\_defaults和response\_model\_exclude\_none，我们可以很直观的了解到他们的意思，不返回是默认值的字段和不返回是None的字段。

还有response\_model\_exclude = {“xxx”,”yyy”} 排除xxx,yyy字段

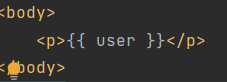
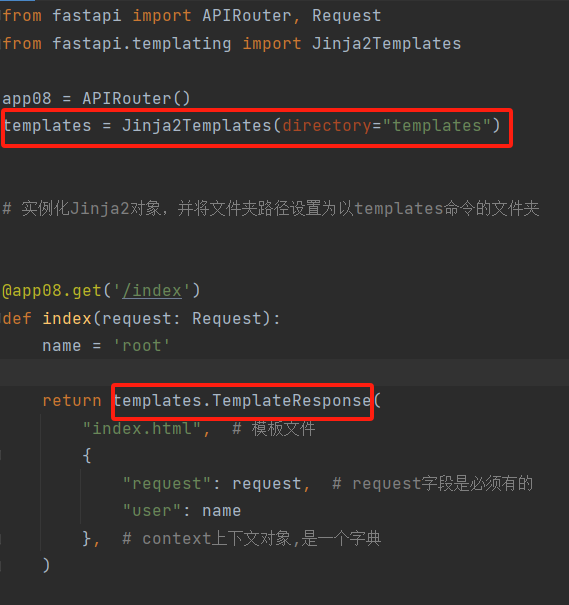
response\_model\_include= {“zzz”} 只显示zzz字段.

1. 响应状态码

from fastapi import status

1. jinja2模板

百分之90是跟Django的模板语言是相同的



{{ 变量名 }}

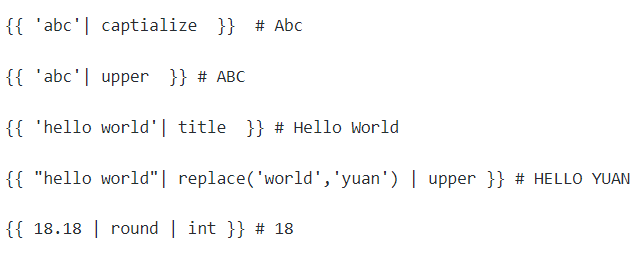
{{ 列表.0 }} {{ 列表.1}}... 在模板中索引是用’.’来索引.

{{ 字典.key }} 同理字典也是用’.’来索引.

过滤器



只需要在变量后⾯使⽤管道(|)分割，多个过滤器可以链式调⽤，前⼀个过滤器的输出会作为后⼀个过滤器的输⼊。



控制结构:

{% if %} {% else %} {% endif %}

{% for book in books %}

<p>{{ book }}</p>

{% endfor %}

1. tortoise ORM

pip install tortoise-orm

pip install aiomysql



main.py



settings.py

TORTOISE\_ORM = {

'connections': {

'default': {

# 'engine': 'tortoise.backends.asyncpg', PostgreSQL

'engine': 'tortoise.backends.mysql', # MySQL or Mariadb

'credentials': {

'host': '127.0.0.1',

'port': '5500',

'user': 'root',

'password': 'root',

'database': 'fastapi',

'minsize': 1,

'maxsize': 5,

'charset': 'utf8mb4',

"echo": True

},

},

},

'apps': {

'models': {

'models': ['models','aerich.models'],

'default\_connection': 'default',

}

},

'use\_tz': False,

'timezone': 'Asia/Shanghai'

}

注意,还有加上aerich.models

模型迁移:

pip install aerich

aerich是一种ORM迁移工具，需要结合tortoise异步orm框架使用。

命令:aerich init -t settings.TORTOISE\_ORM # TORTOISE\_ORM配置的位置)

初始化完会在当前目录生成一个文件：pyproject.toml和一个文件夹：migrations

● pyproject.toml：保存配置文件路径，低版本可能是aerich.ini

● migrations：存放迁移文件

命令:aerich init-db

1. 此时数据库中就有相应的表格

2. 如果TORTOISE\_ORM配置文件中的models改了名，则执行这条命令时需要增加--app参数，来指定你修改的名字

**更新模型并重新进行迁移**

修改model类，重新生成迁移文件,比如添加一个字段

命令:aerich migrate [--name] (标记修改操作) # aerich migrate --name add\_column

重新执行迁移,写入数据库

命令:aerich upgrade

回到上一个版本

命令:aerich downgrade

查看历史迁移记录

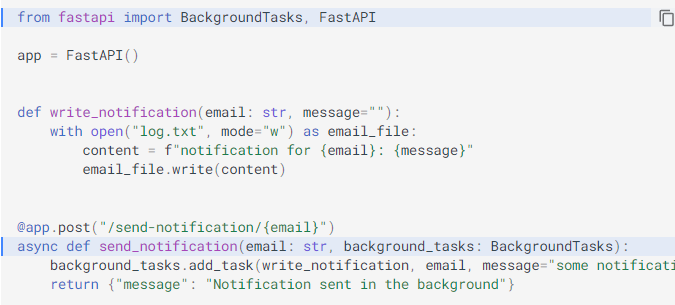
命令:aerich history

1. restful规范



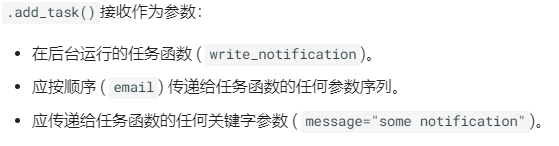
1. BackgroundTasks 后台任务

BackgroundTasks是 FastAPI 框架的一个功能，它允许我们在响应已经返回给客户端后继续处理任务，而无需等待任务完成。这意味着这些任务将在后台异步执行，而不会阻塞主请求——响应循环。这在处理一些非实时关键的任务时非常有用，比如发送电子邮件、日志记录、消息通知等



首先,创建后台任务函数.

然后,添加后台任务通过.add\_task()对象添加.

.add\_task()对象接受的参数有:

阻塞问题:

有时候我们的后台函数中也可能包含阻塞操作，例如网络请求,因为后台函数里的网络请求阻塞了，会导致整个请求也被阻塞

要解决这个问题，可以通过线程、进程或协程来执行后台任务

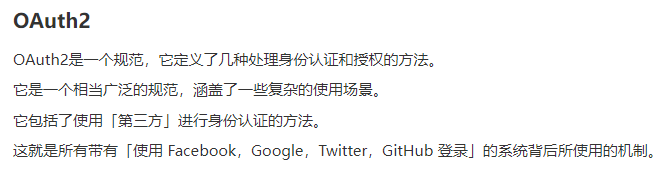




1. fastapi响应文件下载



1. FastAPI安全性



OAuth2授权模式:1.授权码授权模式;2.隐式授权模式,3.密码授权模式4.客户端凭证授权模式

1. 依赖注入

什么是依赖注入?

在依赖注入中，依赖项（也称为组件或服务）不是在代码内部创建或查找的，而是由外部系统提供给组件。这种方式有助于降低组件之间的耦合度，使系统更加灵活、可维护和可测试。

依赖注入非常适合在业务逻辑复用的场景使用，可以有效地减少代码重复，除此之外，在进行权限校验、身份验证、共享数据库连接等场景也非常适用。

通俗的理解： 依赖注入是一种软件设计模式，用于管理不同模块之间的依赖关系。在依赖注入中，一个对象不会直接创建或者获取它所依赖的对象，而是通过外部传递来实现。这种方式使得代码更加灵活、可维护、可测试。

当该接口被调用的时候，先调用此依赖项函数，传递合适的参数，依赖函数执行，返回结果再传递给路由函数的参数，如 user\_id。

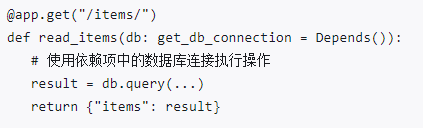
FastAPI支持依赖注入通过以下方式：

使用参数注解： 你可以在路由处理程序函数的参数上使用Python的类型注解，告诉FastAPI你需要什么依赖项。FastAPI将根据类型自动查找或创建这些依赖项。

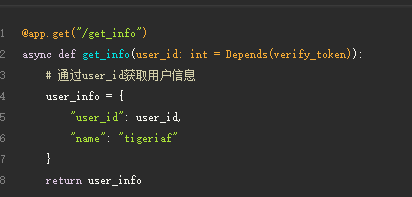
依赖注入容器： FastAPI内部使用一个依赖注入容器来管理依赖项。这个容器会在运行时解析参数注解，自动处理依赖项的创建和生命周期管理。

1. 创建依赖项函数： 首先，你需要创建一个Python函数，它将包含你想要注入的依赖项的逻辑。这个函数通常会返回一个对象，该对象将在路由处理程序中使用。

传入依赖项函数:



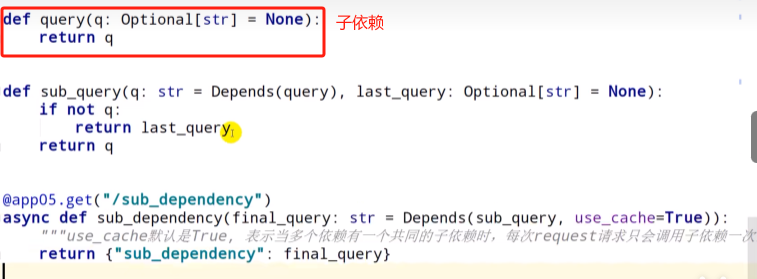
传入依赖项函数返回值:



1. 以类的形式来创建依赖项



子依赖 (嵌套依赖):



user\_cache默认是True,表示当多个依赖有一个共同的子依赖时,每次request请求只会调用子依赖一次,多次调用将从缓存中获取.

前面都是在路径函数里面导入依赖,作为函数的参数.

在路径操作装饰器导入依赖

一般是有些依赖项不返回值,对于这种情况,就不必要在路径函数的参数使用Depends了.注意,就算这些依赖项会返回值,但他们的值也不会传递给路径操作函数.

@app.get(“/items”,dependencies = [Depends(xxx),Depends(xxx)])

def item():

pass

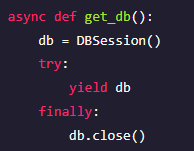
全局依赖:

在app创建中, app = FastAPI(dependencies = [Denpends(verify\_token)]) 类似中间件的作用.

带yield的依赖

想要依赖项完成后执行一些额外的步骤,需要将return替换为yield,并在之后编写额外的逻辑步骤.

例如数据库依赖项yield



1. 规范目录结构

├── app // 应用程序目录

│ ├── api // API接口目录

│ │ └── v1 // 版本1的API接口

│ │ ├── apis // API相关接口

│ │ ├── base // 基础信息接口

│ │ ├── menus // 菜单相关接口

│ │ ├── roles // 角色相关接口

│ │ └── users // 用户相关接口

│ ├── controllers // 控制器目录

│ ├── core // 核心功能模块

│ ├── log // 日志目录

│ ├── models // 数据模型目录

│ ├── schemas // 数据模式/结构定义

│ ├── settings // 配置设置目录

│ └── utils // 工具类目录

├── deploy // 部署相关目录

│ └── sample-picture // 示例图片目录

└── web // 前端网页目录

├── build // 构建脚本和配置目录

│ ├── config // 构建配置

│ ├── plugin // 构建插件

│ └── script // 构建脚本

├── public // 公共资源目录

│ └── resource // 公共资源文件

├── settings // 前端项目配置

└── src // 源代码目录

├── api // API接口定义

├── assets // 静态资源目录

│ ├── images // 图片资源

│ ├── js // JavaScript文件

│ └── svg // SVG矢量图文件

├── components // 组件目录

│ ├── common // 通用组件

│ ├── icon // 图标组件

│ ├── page // 页面组件

│ ├── query-bar // 查询栏组件

│ └── table // 表格组件

├── composables // 可组合式功能块

├── directives // 指令目录

├── layout // 布局目录

│ └── components // 布局组件

├── router // 路由目录

│ ├── guard // 路由守卫

│ └── routes // 路由定义

├── store // 状态管理(pinia)

│ └── modules // 状态模块

├── styles // 样式文件目录

├── utils // 工具类目录

│ ├── auth // 认证相关工具

│ ├── common // 通用工具

│ ├── http // 封装axios

│ └── storage // 封装localStorage和sessionStorage

└── views // 视图/页面目录

├── error-page // 错误页面

├── login // 登录页面

├── profile // 个人资料页面

├── system // 系统管理页面

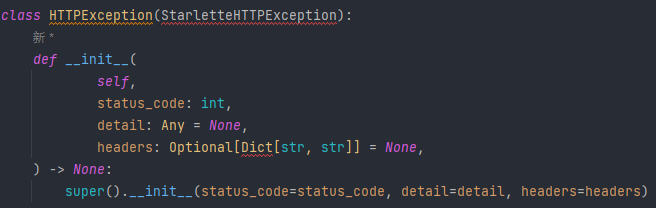
└── workbench // 工作台页面

1. fastapi错误处理

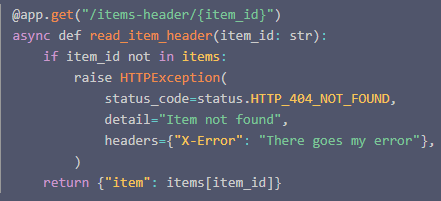
FastAPI 提供了一个装饰器 @app.exception\_handler(exception\_type)，用于定义异常处理器。通过该装饰器，我们可以指定要处理的异常类型，并在异常发生时执行相应的逻辑。

1. HTTPException

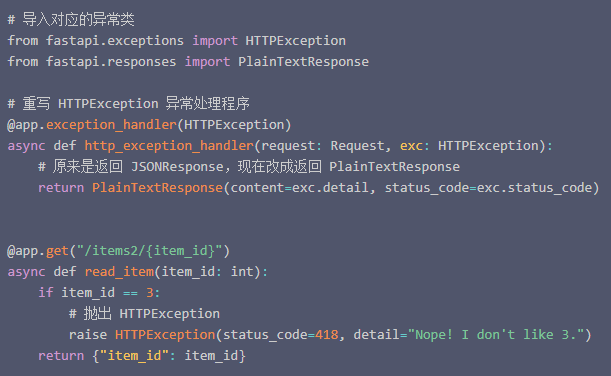
将带有错误的HTTP响应(状态码和响应信息)返回给客户端.



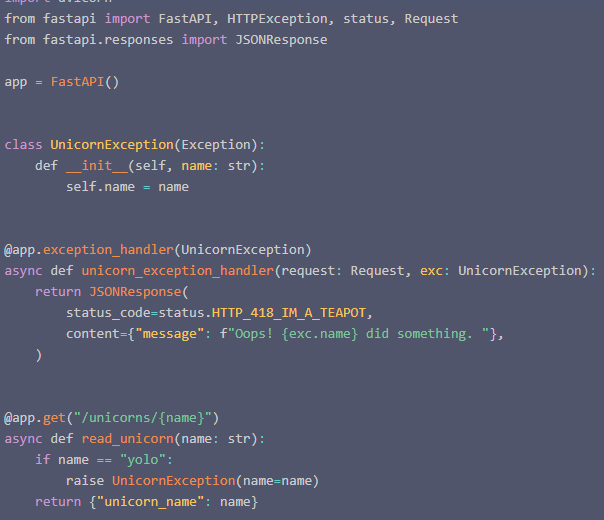
例子:



1. 重写HTTPException



1. 自定义异常类

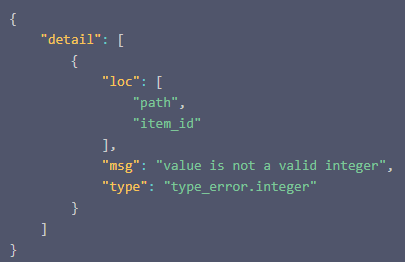


1. 重写请求验证异常类

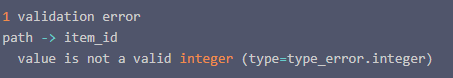


当客户端发送的请求包含无效数据时,即不符合路径函数里面的参数类型声明.会抛出RequestValidationError.

默认为

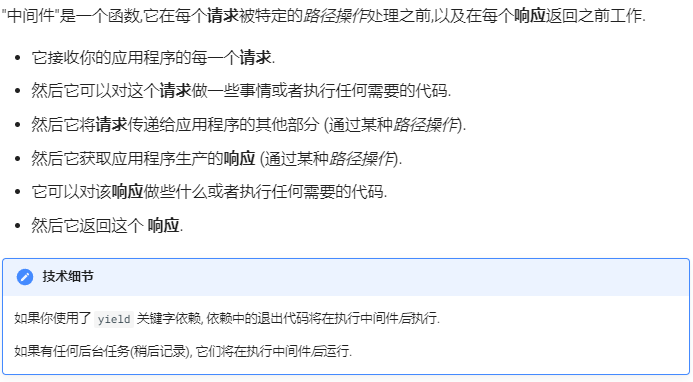


重写后:



1. 中间件

Fastapi中的中间件是一个函数，用于处理每一请求路径操作之前，以及每个响应返回之前工作。



1. 通过app.middleware(“http”)装饰器创建中间件.

@app.middleware(“http”)

async def middleware(request: Request, call\_next): #必须用async

#before

response = await call\_next(request) #调用下一个中间件或路由操作函数

#after

return response

1. 通过继承BaseHTTPMiddleware创建中间件

from starlette.middleware.base import BaseHTTPMiddleware

from fastapi import FastAPI, Request

app = FastAPI()

class MyMiddleware(BaseHTTPMiddleware):

def \_\_init\_\_(self,app,header\_value:str):

super().\_\_init\_\_(app)

self.header\_value = header\_value

#dispatch必须实现

async def dispatch(self,request:Request,call\_next):

#before

response = await call\_next(request)

#after

retrun response

#将中间件添加到主程序中

app.add\_middleware(MyMiddleware,header\_value=”middleware”)

#app.add\_middleware()操作来引入已定义的中间件，接收两个参数，

#第一个参数为中间件的类，第二个参数为要传递给中间件的参数。

1. 多个中间件的处理顺序

例如有两个中间件MyMiddleware1与MyMiddleware2

app.add\_middleware(MyMiddleware1)

app.add\_middleware(MyMiddleware2)

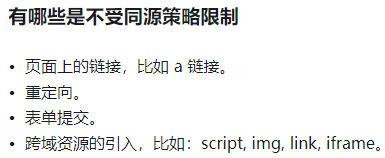
那么外层是MyMiddleware2,里层是MyMiddleware1.其顺序是与中间件注册顺序有关,越晚注册就越先执行,同时也是最后收尾执行响应.

1. fastapi已有常用的中间件:
2. HTTPSRedirectMiddleware: 将 HTTP 请求重定向到 HTTPS。这个中间件会检查请求的协议，如果是 HTTP，则自动将请求重定向到相应的 HTTPS 地址;
3. TrustedHostMiddleware: 强制所有传入请求都具有正确设置的 Host 标头，以防止 HTTP 主机标头攻击。
4. GZipMiddleware: 用于在响应中压缩内容，以减小传输大小。这有助于提高应用程序的性能，特别是在处理大量文本或数据时。
5. CORSMiddleware: 用于处理跨域资源共享（CORS）请求。CORS 是一种浏览器机制，允许 Web 页面从不同的域请求不同域的资源。(常用)
6. 跨域资源共享CORS

当一个资源从与该资源本身所在的服务器不同的域、协议或端口请求一个资源时，资源会发起一个跨域 HTTP 请求。

为什么会出现跨域呢？因为同源策略。

同源策略是浏览器的一个安全功能，不同源的客户端脚本在没有明确授权的情况下，不能读写对方资源。同源策略就是指必须在同一个协议，域名，端口号下，而且三者必须一致的



from fastapi.middleware.cors import CORSMiddleware

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"], #允许的来源

allow\_credentials=True, #是否允许携带凭证（例如，使用 HTTP 认证、Cookie 等）

allow\_methods=["\*"], #允许的 HTTP 方法,get,post等等

allow\_headers=["\*"], # 允许的 HTTP 头信息

expose\_headers=["\*"], # 允许前端访问的额外响应头

max\_age=600, # 请求的缓存时间，以秒为单位

)

1. 测试用例

以test\_开头新建.py文件用于测试.

test\_demo.py

from fastapi.testclient import TestClient

from main import app #main.py里的主程序app

client = TestClient(app) #先pip install pytest

def test\_run\_task():

#不能async def ,因为测试用例是按顺序跑

#函数名用”test\_”开头是pytest规范.

response = client.post(url=””)

assert response.status\_code == 200

assert response.json() == {“message”:”ok”}

运行: 进入终端执行 pytest

1. 异步编程
   1. 协程

协程不是计算机提供的,计算机有(进程,线程).协程是程序员人为创造的.可以理解为微线程.用一个线程在代码中来回切换游走地运行.

例子:

普通的函数

main.py

def func1():

print(1)

...

print(2)

def func2():

print(3)

...

print(4)

func1()

func2()

普通的就是按顺序依次执行函数func1(),func2()

而如果是协程,那么可以在函数之间来回切换运行.(只有一个线程来完成这个事情)

实现协程功能的方法:

greenlet 早期模块 第三方模块

yield关键字

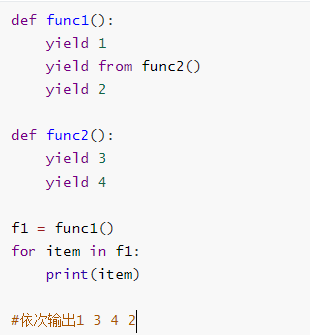
asyncio装饰器 (python3.4引入)

async,await关键字(python 3.5引入) (推荐)

1.1使用greenlet实现协程:



1.2通过yield关键字实现协程:(了解即可,没什么意义)



yield的机制是暂停运行过程输出当前结果并保留状态，状态包括上次终止的位置和终止时的数值。下一次next()时从上一次终止的地方开始

1.3 asyncio内置模块:



注意:遇到IO阻塞自动切换.

1.4 async & await关键字



* 1. 协程的意义

简而言之,就是充分利用线程,不让线程闲着.

* 1. 事件循环

可以理解为死循环,去检测并执行某些代码.

#伪代码

任务列表 = [ 任务1,任务2,任务3....]

while True:

可执行的任务列表 , 已完成的任务列表 = 去任务列表中检查所有任务,返回”可执行”和”已完成”的任务.

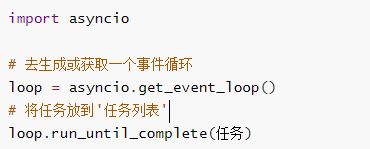
for 就绪任务 in 可执行的任务列表:

执行已就绪的任务

for 已完成的任务 in 已完成的任务列表:

在任务列表中移除 已完成的任务

如果 任务列表 中的任务都完成,则退出循环.



* 1. 快速上手

协程函数: 定义函数时 async def 函数名

协程对象: 执行协程函数得到的协程对象

import asyncio

async def func():

pass

result = func()

#注意,执行协程函数创建协程对象时,函数内部代码是不会执行

#loop.asyncio.get\_event\_loop()

#loop.run\_until\_complete(result)

asyncio.run(result) #python3.7之后

asyncio.run()等价于loop.asyncio.get\_event\_loop()与loop.run\_until\_complete()

* 1. await

await 后面接 可等待的对象(协程对象,future对象,task对象)

简而言之就是”等”.等待await 后的对象返回,才继续向下执行.

在等待的时候,线程会切换到另一个任务执行.

* 1. Task对象



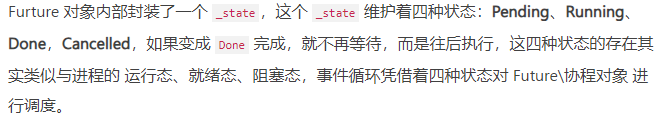
示例:



* 1. asyncio.Future对象

task继承future

task对象内部await结果处理基于future对象来的.



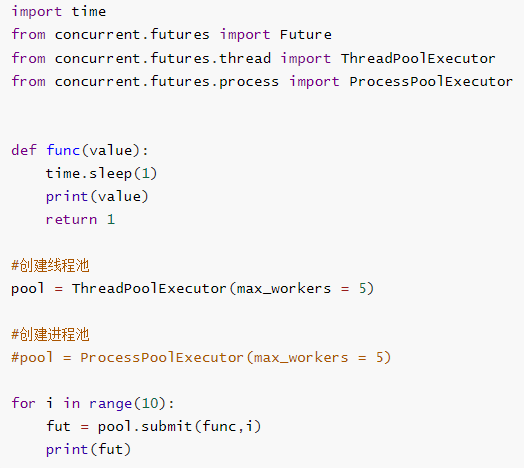
示例:



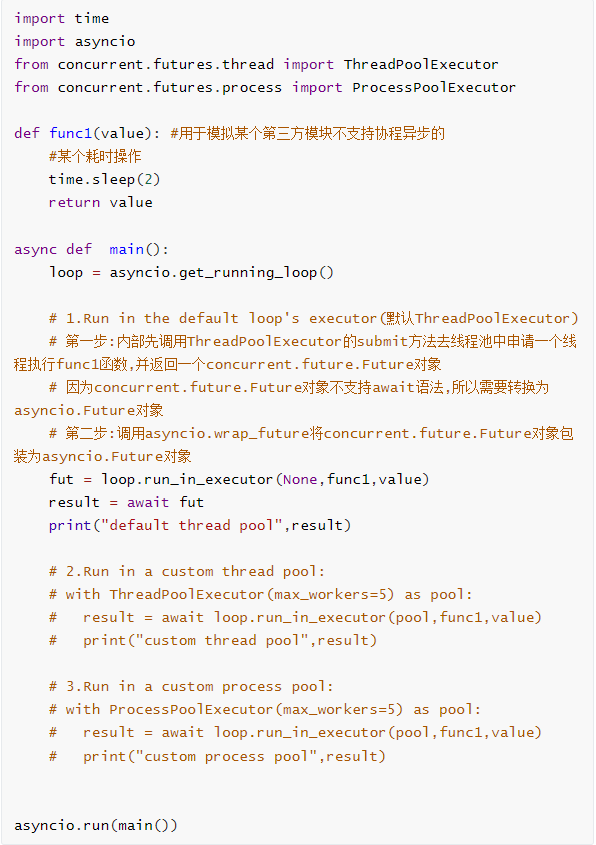
* 1. concurrent.futures.Future对象

coroutines.futures.Future 常用于 多进程、多线程实现并发。使用线程池,进程池实现异步操作.

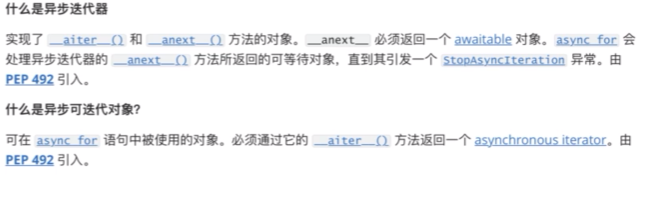
示例:



基于协程的异步和基于进程池,线程池的异步也可以结合起来使用,常用于异步编程时,遇到某个第三方模块,它不支持协程异步时使用.



* 1. 异步迭代器

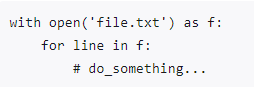




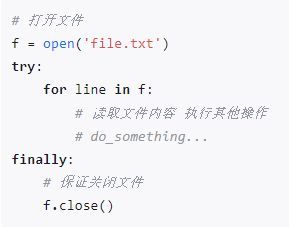
async for语句必须写在协程函数里面,不然报错

* 1. 异步上下文管理器

python上下文管理器:



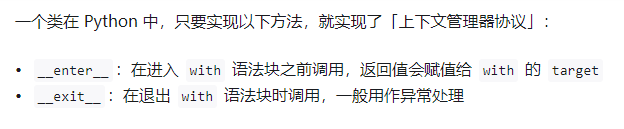
with语句等同于下面的功能.可以保证最后能释放文件资源



那么with如何运行的呢?



这是with语法格式,可以看到with后面的表达式是上下文管理



异步上下文管理器:



* 1. uvloop

