# 1.0服务端框架

## 服务端框架使用说明

首先是一个概念：

我希望一【**类】**业务称为一个server。一个server下面若干service。负责各种细分业务

开启一个服务需要以下几步:

|  |
| --- |
| *# coding=utf-8* **from** server\_core.server **import** Server **from** server\_impl.user\_server **import** user\_server **from** server\_impl.room\_server **import** room\_server   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  server = Server(**"cwl server"**, **"01"**)  server.start(mode=**"light"**)   *# 加载 service* user\_server.UserServer(server)  room\_server.RoomServer(server)   server.run() |

1. 从**from** server\_core.server **import** Server 加载Server类
2. 从**from** server\_impl.user\_server **import** user\_server。import所以你需要的所有service。这些需要由你来决定
3. server = Server(**"cwl server"**, **"01"**) 这里两个参数分别是服务名和服务ID。目前该值对服务无影响
4. server.start(mode=**"light"**)

mode可选值：**epoll，epoll\_et，select，light**

决定了网络IO读取行为。分别为IO复用中的epoll lt, epoll et, select 和 无IO复用。

此函数会根据mode值判断当前操作系统是否支持。不支持则会自己选一个

1. 加载service。需要构造若干 server的实例。后面有具体说明
2. server.run() 进入服务端主循环

上面有个遗留问题是Server和Service是何物

在util包中，我写了一个工具server\_creator.py。他会根据server\_temple下面的模板生成业务代码

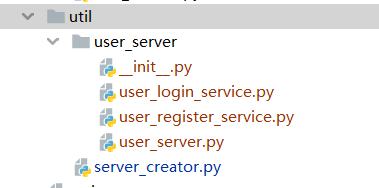
|  |
| --- |
| **if** \_\_name\_\_ == **'\_\_main\_\_'**:sc = ServerCreator()  sc.load\_config({  **"server\_name"**: **"user"**,  **"service\_list"**: [  {**"service\_name"**: **"login"**},  {**"service\_name"**: **"register"**},  ]  })  sc.create() |

每个server的生成都需要调用上面三个函数。Load\_config参数是一个字典

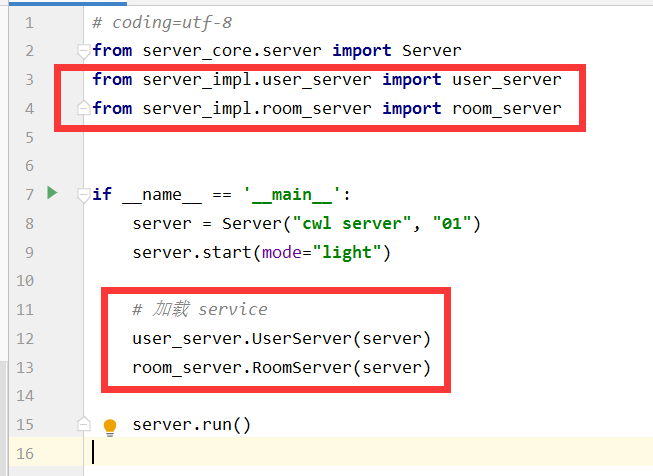
“server\_name”代表生成的服务名称

“service\_list”是一个列表，代表需要生产的service信息

其中只有一个项 “service\_name” 表示service名词。如果有多个单词请用空格 隔开。



它会生成一个包。包内有唯一的server后缀文件。你需要把他拷贝到某个地方。并在start后，主循环run前引入并加载这个服务。加载地方如下



在service中提供了三个函数。分别后缀pretreatment，run，aftertreatment

并有三个参数controller, req, res

controller提供了一系列工具，后面再提。Req,res方便包含了请求和响应的信息

框架会按顺序调用这三个函数。你需要在其中完善你的业务逻辑

在第一个函数中，我希望能做一些参数的检查

|  |
| --- |
| **def** user\_login\_service\_pretreatment(controller, req, res):  req.check\_contain\_string(**"username"**)  req.check\_contain\_string(**"password"**) |

参数检测的方法在req和res中都有

|  |
| --- |
| **def** check\_contain\_string(self, key):  # 判断是否存在字符串类型的key  pass **def** check\_contain\_int(self, key, min\_val=None, max\_val=None):  # 判断是否存在字符串类型的 key，如果min\_val,max\_val不为空还会判断  # 改值范围是否正确  pass |

在第二个函数中

|  |
| --- |
| **def** user\_login\_service\_run(controller, req, res):  **if not** req.parse\_success **or not** req.content:  # log something  **return** Log().debug(**"login service: "** + str(req.msg))   *# 获取参数* username = req.content[**"username"**]  password = req.content[**"password"**]  *# 处理业务* login\_success = False   user\_list = {  **"netease1"**: **"123456"**,  **"netease2"**: **"123456"**,  **"netease3"**: **"123456"**,  }  **if** username **in** user\_list **and** password == user\_list[username]:  login\_success = True   *# 设置返回 dict* res.content = {  **"ret"**: 0,  **"login\_success"**: login\_success  } |

他首先会对前面参数预检查的结果做判断。决定是否继续运行

在req.content中你可以获得所需的参数，这是一个字典。

最后你需要在最后设置res.content为一个字典。表示你的返回参数。

框架会为你把字符流协议包解析。锁定到某一个service.然后把参数解析好的字典给你。最后把你最后设置的res.content的字典发回发送者。

每一个service都有一个handler\_id。每一个数据包也有一个handler\_id。决定了某一个数据包会命中哪一个service。

|  |
| --- |
| **class** UserLoginService:   **def** \_\_init\_\_(self):  **if not** hasattr(config, **"USER\_LOGIN\_SERVICE"**):  **raise** Exception(**"config file service id not define"**)  self.handler\_id = config.USER\_LOGIN\_SERVICE  ... |

另外每一个service中的异常会被一个外层try捕捉并打日志。单个service异常不会导致服务崩溃

客户端：

消息包格式为：

【4字节消息内容长度】【4字节句柄】【消息内容】

1. 内容长度是一个4字节的int，表示后面消息内容的长度。
2. 句柄是一个4字节int，表示需要调用哪一个service
3. 消息内容是数据的序列化字符串。目前版本只有json的序列化。如果消息内容按json解析失败框架会阻止service被调用。如果需要更换序列化方法，只需要在Request中更换新的序列化工具类即可
4. 以上数据框架做了网络字节序的处理

符合以上要求的客户端会被该框架识别。

关于controller的使用：

## 服务端框架实现说明

