类Flask框架实现

路由分组**

所谓路由分组，就是按照前缀分别映射。

需求
URL为/product/123456
需要将产品ID提取出来

分析
这个URL可以看做是一级分组路由，生成环境中够用了。
例如

```
product = Router('/product') # 匹配前缀/product，每一个路由实例保存一个前缀
product.get('/(?P<id>\d+)') # 匹配路径为/product/123456，路由实例再管理第二级匹配
```

常见的一级目录
/admin 后台管理
/product 产品
这些目录都是 / 根目录下的第一级， 暂且称为前缀prefix

> 前缀要求必须以 / 开头，但不能以分隔符结尾

```
# 下面的定义方法已经不能描述prefix和URL之间的隶属关系
@Router.get(r'^/$')
def index(request: Request):
    pass

@Router.route('^/python$')
def showpython(request: Request):
    pass
```

如何建立prefix和URL之间的隶属关系？
一个Prefix下可以有若干个URL，这些URL都是属于这个Prefix的。
不同前缀对应不同的Router实例管理，所有路由注册方法，都成了实例的方法，路由表实例自己管理。
App中需要保存不同前缀的Router实例，提供注册方法，把Router对象被管理起来。
`__call__`方法依然是WSGI回调入口，在其中遍历所有Router实例，将路径匹配代码全部挪到Router中并新建一个match方法。
match方法匹配就返回Response对象，否则返回None。

```
from webob import Response, Request
from webob.dec import wsgify
from wsgiref.simple_server import make_server
from webob.exc import HTTPNotFound
import re
```

```python
class Router:
    def __init__(self, prefix:str=''):
        self.__prefix = prefix.rstrip('/\\') # 前缀，例如/product
        self.__routetable = [] # 存三元组，列表，有序的

    def route(self, pattern, *methods): # 注册路由，装饰器
        def wrapper(handler):
            self.__routetable.append(
                (tuple(map(lambda x: x.upper(), methods)),
                 re.compile(pattern), handler)) # (方法元组,预编译正则对象,处理函数)
            return handler
        return wrapper

    def get(self, pattern):
        return self.route(pattern, 'GET')

    def post(self, pattern):
        return self.route(pattern, 'POST')

    def head(self, pattern):
        return self.route(pattern, 'HEAD')

    def match(self, request:Request):
        # 必须先匹配前缀
        if not request.path.startswith(self.__prefix):
            return None
        # 前缀匹配，说明就必须这个Router实例处理，后续匹配不上，依然返回None
        for methods, pattern, handler in self.__routetable:
            # not methods表示一个方法都没有定义，就是支持全部方法
            if not methods or request.method.upper() in methods:
                # 前提已经是以__prefix开头了，可以replace，去掉prefix剩下的才是正则表达式需要匹配的路
径
                matcher = pattern.match(request.path.replace(self.__prefix, '', 1))
                if matcher: # 正则匹配
                    # 动态为request增加属性
                    request.groups = matcher.groups() # 所有分组组成的元组，包括命名分组
                    request.groupdict = matcher.groupdict() # 命名分组组成的字典
                    return handler(request)


class App:
    _ROUTERS = [] # 存储所有一级Router对象

    # 注册
    @classmethod
    def register(cls, *routers:Router):
        for router in routers:
            cls._ROUTERS.append(router)

    @wsgify
    def __call__(self, request:Request):
        # 遍历_ROTERS，调用Router实例的match方法，看谁匹配
```

```
        for router in self._ROUTERS:
            response = router.match(request)
            if response: # 匹配返回非None的Router对象
                return response # 匹配则立即返回
        raise HTTPNotFound('<h1>你访问的页面被外星人劫持了</h1>')


# 创建Router对象
idx = Router()
py = Router('/python')

# 注册
App.register(idx, py)


@idx.get(r'^/$')
@idx.route(r'^/(?P<id>\d+)$') # 支持所有方法访问
def indexhandler(request):
    print(request.groups)
    print(request.groupdict)
    return '<h1>马哥教育欢迎你. magedu.com</h1>'

@py.get('^/(\w+)$')
def pythonhandler(request):
    res = Response()
    res.charset = 'utf-8'
    res.body = '<h1>Welcome to Magedu Python</h1>'.encode()
    return res


if __name__ == '__main__':
    ip = '127.0.0.1'
    port = 9999
    server = make_server(ip, port, App())
    try:
        server.serve_forever() # server.handle_request() 一次
    except KeyboardInterrupt:
        server.shutdown()
        server.server_close()
```

字典访问属性化

这是一个技巧
让kwargs这个字典，不使用[ ]访问元素，使用.点号访问元素，如同属性一样访问。

字典转属性类，如下

```
class AttrDict:
    def __init__(self, d:dict):
        self.__dict__.update(d if isinstance(d, dict) else {})

    def __setattr__(self, key, value):
```

```
        # 不允许修改属性
        raise NotImplementedError

    def __repr__(self):
        return "<AttrDict {}>".format(self.__dict__)

d = {'a':1, 'b':2}

do = AttrDict(d)
print(do.a, do.b)
print(do.__dict__)

# 下面语句抛异常
do.c = 1
do.a = 2
# 下面语句可以
do.__dict__['c'] = 1000
print(do.__dict__)
```

修改Router的match方法如下

```
class AttrDict:
    def __init__(self, d:dict):
        self.__dict__.update(d if isinstance(d, dict) else {})

    def __setattr__(self, key, value):
        # 不允许修改属性
        raise NotImplementedError

    def __repr__(self):
        return "<AttrDict {}>".format(self.__dict__)

    def __len__(self):
        return len(self.__dict__)

class Router:
    def match(self, request:Request):
        # 必须先匹配前缀
        if not request.path.startswith(self.__prefix):
            return None
        # 前缀匹配，说明就必须这个Router实例处理，后续匹配不上，依然返回None
        for methods, pattern, handler in self.__routetable:
            # not methods表示一个方法都没有定义，就是支持全部方法
            if not methods or request.method.upper() in methods:
                # 前提已经是以__prefix开头了，可以replace，去掉prefix剩下的才是正则表达式需要匹配的路
径
                matcher = pattern.match(request.path.replace(self.__prefix, '', 1))
                if matcher: # 正则匹配
                    # 动态为request增加属性
                    request.groups = matcher.groups() # 所有分组组成的元组，包括命名分组
                    request.groupdict = AttrDict(matcher.groupdict()) # 命名分组组成的字典被属性化
                    return handler(request)
```

```
@idx.get(r'^/$')
@idx.route(r'^/(?P<id>\d+)$') # 支持所有方法访问
def indexhandler(request):
    print(request.groups)
    print(request.groupdict)
    id = ''
    if request.groupdict:
        id = request.groupdict.id
    return '<h1>马哥教育欢迎你{}. magedu.com</h1>'.format(id)
```

正则表达式的化简***

问题
目前路由匹配使用正则表达式定义，不友好，很多用户不会使用正则表达式，能否简化？

分析
生产环境中，URL是规范的，不能随便书写，路径是有意义的，尤其是对restful风格。
所以，要对URL规范。
例如/product/111023564 ，这就是一种规范，要求第一段是业务，第二段是ID。

设计
路径规范化，如下定义
`/student/{name:str}/{id:int}`
类型设计，支持str、word、int、float、any类型。
还可以考虑一种raw类型，直接支持正则表达式，暂不考虑实现。
通过这样的定义，可以让用户定义简化了，也规范了，背后的转换由编程者实现。

类型	含义	对应正则
str	不包含/的任意字符。默认类型	[^/]+
word	字母和数字	\w+
int	纯粹数字，正负数	[+-]?\d+
float	正负号，数字，包含.	[+-]?\d+.\d+
any	包含/的任意字符	.+

建立如下的映射关系

```
# 类型字符串映射到正则表达式
TYPEPATTERNS = {
    'str'   : r'[^/]+',
    'word'  : r'\w+',
    'int'   : r'[+-]?\d+',
    'float' : r'[+-]?\d+\.\d+', # 严苛的要求必须是 15.6这样的形式
    'any'   : r'.+'
}
```

```
# 类型字符串到Python类型的映射
TYPECAST = {
    'str'   : str,
    'word'  : str,
    'int'   : int,
    'float' : float,
    'any'   : str
}
```

`/student/{name:str}/{id:int}`怎样转换？
使用`(\{([^\{\}]+):([^\{\}]+)\})`来search整个字符串，分别找到{name:str}和{id:int}并替换成目标的正则表达式。
id转变为命名分组的名字，int转变为对应的正则表达式，如下
`/student/{name:str}/xxx/{id:int}` => `/student/(?P<name>[^/]+)/xxx/(?P<id>[+-]?\d+)`

对于`/student/{name:str}`和`/student/{name:}`正则表达式修改为`/{([^{}:]+):?([^{}:]*)}`

先看一个正则替换sub函数使用的例子

```
import re

pattern = r'\d+'
repl = ''
src = 'welcome 123 to 456 magedu.com'

regex = re.compile(pattern)
dest = regex.sub(repl, src)

print(dest, end='----------------\n')


pattern = r'/{([^{}:]+):?([^{}:]*)}'
src = '/student/{name}/xxx/{id:int}'
print(src)
def repl(matcher):
    print(matcher.group(0))
    print(matcher.group(1))
    print(matcher.group(2)) # {name}或{name:}这个分组都匹配为''
    # (?P<name>...)
    return '/(?P<{}>{})'.format(
        matcher.group(1),
        'T' if matcher.group(2) else 'F'
    )

regex = re.compile(pattern)
dest = regex.sub(repl, src)
print(dest)
```

编写一个转换例程

```
import re
```

```
regex = re.compile('/{([^{}:]+):?([^{}:]*)}')

s = [
    '/student/{name:str}/xxx/{id:int}',
    '/student/xxx/{id:int}/yyy',
    '/student/xxx/5134324',
    '/student/{name:}/xxx/{id}',
    '/student/{name:}/xxx/{id:aaa}'
]

# /{id:int} => /(?P<id>[+-]?\d+)
# '/(?<{}>{})'.format('id', TYPEPATTERNS['int'])

TYPEPATTERNS = {
    'str': r'[^/]+',
    'word': r'\w+',
    'int': r'[+-]?\d+',
    'float': r'[+-]?\d+\.\d+',  # 严苛的要求必须是 15.6这样的形式
    'any': r'.+'
}

def repl(matcher):
    # print(matcher.group(0))
    # print(matcher.group(1))
    # print(matcher.group(2)) # {name}或{name:}这个分组都匹配为''
    return '/(?P<{}>{})'.format(
        matcher.group(1),
        TYPEPATTERNS.get(matcher.group(2), TYPEPATTERNS['str'])
    )

def parse(src:str):
    return regex.sub(repl, src)

for x in s:
    print(parse(x))
# 运行结果
/student/(?P<name>[^/]+)/xxx/(?P<id>[+-]?\d+)
/student/xxx/(?P<id>[+-]?\d+)/yyy
/student/xxx/5134324
/student/(?P<name>[^/]+)/xxx/(?P<id>[^/]+)
/student/(?P<name>[^/]+)/xxx/(?P<id>[^/]+)
```

将上面的代码合入Router类中

```
from webob import Response, Request
from webob.dec import wsgify
from wsgiref.simple_server import make_server
from webob.exc import HTTPNotFound
import re
```

```python
class AttrDict:
    def __init__(self, d:dict):
        self.__dict__.update(d if isinstance(d, dict) else {})

    def __setattr__(self, key, value):
        # 不允许修改属性
        raise NotImplementedError

    def __repr__(self):
        return "<AttrDict {}>".format(self.__dict__)

    def __len__(self):
        return len(self.__dict__)

class Router:
    ############## 正则转换
    __regex = re.compile(r'/{([^{}:]+):?([^{}:]*)}')

    TYPEPATTERNS = {
        'str': r'[^/]+',
        'word': r'\w+',
        'int': r'[+-]?\d+',
        'float': r'[+-]?\d+\.\d+',  # 严苛的要求必须是 15.6这样的形式
        'any': r'.+'
    }

    def __repl(self, matcher):
        # print(matcher.group(0))
        # print(matcher.group(1))
        # print(matcher.group(2)) # {name}或{name:}这个分组都匹配为''
        return '/(?P<{}>{})'.format(
            matcher.group(1),
            self.TYPEPATTERNS.get(matcher.group(2), self.TYPEPATTERNS['str'])
        )

    def __parse(self, src: str):
        return self.__regex.sub(self.__repl, src)

    ############ 实例
    def __init__(self, prefix:str=''):
        self.__prefix = prefix.rstrip('/\\') # 前缀，例如/product
        self.__routetable = [] # 存三元组，列表，有序的

    def route(self, pattern, *methods): # 注册路由，装饰器
        def wrapper(handler):
            # /student/{name:str}/xxx/{id:int}  =>
            # '/student/(?P<name>[^/]+)/xxx/(?P<id>[+-]?\\d+)'
            self.__routetable.append(
                (tuple(map(lambda x: x.upper(), methods)),
                 re.compile(self.__parse(pattern)), # 用户输入规则转换为正则表达式并编译
                 handler)
            ) # (方法元组,预编译正则对象,处理函数)
            return handler
```

```
        return wrapper

    def get(self, pattern):
        return self.route(pattern, 'GET')

    def post(self, pattern):
        return self.route(pattern, 'POST')

    def head(self, pattern):
        return self.route(pattern, 'HEAD')

    def match(self, request:Request):
        # 必须先匹配前缀
        if not request.path.startswith(self.__prefix):
            return None
        # 前缀匹配，说明就必须这个Router实例处理，后续匹配不上，依然返回None
        for methods, pattern, handler in self.__routetable:
            # not methods表示一个方法都没有定义，就是支持全部方法
            if not methods or request.method.upper() in methods:
                # 前提已经是以__prefix开头了，可以replace，去掉prefix剩下的才是正则表达式需要匹配的路
径
                matcher = pattern.match(request.path.replace(self.__prefix, '', 1))
                if matcher: # 正则匹配
                    # 动态为request增加属性
                    request.groups = matcher.groups() # 所有分组组成的元组，包括命名分组
                    request.groupdict = AttrDict(matcher.groupdict()) # 命名分组组成的字典被属性化
                    return handler(request)


class App:
    _ROUTERS = [] # 存储所有一级Router对象

    # 注册
    @classmethod
    def register(cls, *routers:Router):
        for router in routers:
            cls._ROUTERS.append(router)

    @wsgify
    def __call__(self, request:Request):
        # 遍历_ROTERS，调用Router实例的match方法，看谁匹配
        for router in self._ROUTERS:
            response = router.match(request)
            if response: # 匹配返回非None的Router对象
                return response # 匹配则立即返回
        raise HTTPNotFound('<h1>你访问的页面被外星人劫持了</h1>')


# 创建Router对象
idx = Router()
py = Router('/python')

# 注册
```

```
App.register(idx, py)


@idx.get(r'^/$')
@idx.route(r'^/{id:int}$') # 支持所有方法访问
def indexhandler(request):
    print(request.groups)
    print(request.groupdict)
    id = ''
    if request.groupdict:
        id = request.groupdict.id
    return '<h1>马哥教育欢迎你{}. magedu.com</h1>'.format(id)

@py.get('^/{id}$')
def pythonhandler(request):
    res = Response()
    res.charset = 'utf-8'
    res.body = '<h1>Welcome to Magedu Python</h1>'.encode()
    return res


if __name__ == '__main__':
    ip = '127.0.0.1'
    port = 9999
    server = make_server(ip, port, App())
    try:
        server.serve_forever() # server.handle_request() 一次
    except KeyboardInterrupt:
        server.shutdown()
        server.server_close()
```

如果要增加对捕获数据的类型，使用re.sub()就不行了。重写prase函数，如下

```
def parse(src:str):
    start = 0
    repl = ''
    types = {}

    matchers = regex.finditer(src)
    for i, matcher in enumerate(matchers):
        name = matcher.group(1)
        t = matcher.group(2)

        types[name] = TYPECAST.get(matcher.group(2), str)

        repl += src[start:matcher.start()] # 拼接分组前
        tmp = '/(?P<{}>{})'.format(
            matcher.group(1),
            TYPEPATTERNS.get(matcher.group(2), TYPEPATTERNS['str'])
        )
        repl += tmp # 替换
```

```
            start = matcher.end() # 移动
        else:
            repl += src[start:] # 拼接分组后的内容

        return repl, types
```

由此，重新修改源码

```
from webob import Response, Request
from webob.dec import wsgify
from wsgiref.simple_server import make_server
from webob.exc import HTTPNotFound
import re


class AttrDict:
    def __init__(self, d:dict):
        self.__dict__.update(d if isinstance(d, dict) else {})

    def __setattr__(self, key, value):
        # 不允许修改属性
        raise NotImplementedError

    def __repr__(self):
        return "<AttrDict {}>".format(self.__dict__)

    def __len__(self):
        return len(self.__dict__)

class Router:
    ############## 正则转换
    __regex = re.compile(r'/{([^{}:]+):?([^{}:]*)}')

    TYPEPATTERNS = {
        'str': r'[^/]+',
        'word': r'\w+',
        'int': r'[+-]?\d+',
        'float': r'[+-]?\d+\.\d+',  # 严苛的要求必须是 15.6这样的形式
        'any': r'.+'
    }

    TYPECAST = {
        'str': str,
        'word': str,
        'int': int,
        'float': float,
        'any': str
    }

    def __parse(self, src: str):
        start = 0
        repl = ''
```

```
        types = {}

        matchers = self.__regex.finditer(src)
        for i, matcher in enumerate(matchers):
            name = matcher.group(1)
            t = matcher.group(2)

            types[name] = self.TYPECAST.get(matcher.group(2), str)

            repl += src[start:matcher.start()]  # 拼接分组前
            tmp = '/(?P<{}>{})'.format(
                matcher.group(1),
                self.TYPEPATTERNS.get(matcher.group(2), self.TYPEPATTERNS['str'])
            )
            repl += tmp  # 替换

            start = matcher.end()  # 移动
        else:
            repl += src[start:]  # 拼接分组后的内容

        return repl, types

    ############ 实例
    def __init__(self, prefix:str=''):
        self.__prefix = prefix.rstrip('/\\') # 前缀，例如/product
        self.__routetable = [] # 存四元组，列表，有序的

    def route(self, rule, *methods): # 注册路由，装饰器
        def wrapper(handler):
            # /student/{name:str}/xxx/{id:int}  =>
            # '/student/(?P<name>[^/]+)/xxx/(?P<id>[+-]?\\d+)'
            pattern, trans = self.__parse(rule) # 用户输入规则转换为正则表达式
            self.__routetable.append(
                (tuple(map(lambda x: x.upper(), methods)),
                 re.compile(pattern), # 编译
                 trans,
                 handler)
            ) # (方法元组,预编译正则对象,类型转换,处理函数)
            return handler
        return wrapper

    def get(self, pattern):
        return self.route(pattern, 'GET')

    def post(self, pattern):
        return self.route(pattern, 'POST')

    def head(self, pattern):
        return self.route(pattern, 'HEAD')

    def match(self, request:Request):
        # 必须先匹配前缀
        if not request.path.startswith(self.__prefix):
```

```
            return None
        # 前缀匹配，说明就必须这个Router实例处理，后续匹配不上，依然返回None
        for methods, pattern, trans, handler in self.__routetable:
            # not methods表示一个方法都没有定义，就是支持全部方法
            if not methods or request.method.upper() in methods:
                # 前提已经是以__prefix开头了，可以replace，去掉prefix剩下的才是正则表达式需要匹配的路
径
                matcher = pattern.match(request.path.replace(self.__prefix, '', 1))
                if matcher: # 正则匹配
                    newdict = {}
                    for k,v in matcher.groupdict().items(): # 命名分组组成的字典
                        newdict[k] = trans[k](v)
                    # 动态为request增加属性
                    request.vars = AttrDict(newdict) # 属性化
                    return handler(request)


class App:
    _ROUTERS = [] # 存储所有一级Router对象

    # 注册
    @classmethod
    def register(cls, *routers:Router):
        for router in routers:
            cls._ROUTERS.append(router)

    @wsgify
    def __call__(self, request:Request):
        # 遍历_ROTERS，调用Router实例的match方法，看谁匹配
        for router in self._ROUTERS:
            response = router.match(request)
            if response: # 匹配返回非None的Router对象
                return response # 匹配则立即返回
        raise HTTPNotFound('<h1>你访问的页面被外星人劫持了</h1>')


# 创建Router对象
idx = Router()
py = Router('/python')

# 注册
App.register(idx, py)


@idx.get(r'^/$')
@idx.route(r'^/{id:int}$') # 支持所有方法访问
def indexhandler(request):
    id = ''
    if request.vars:
        id = request.vars.id
        print(type(id))
    return '<h1>马哥教育欢迎你{}. magedu.com</h1>'.format(id)
```

```
@py.get('^/{id}$')
def pythonhandler(request):
    if request.vars:
        print(type(request.vars.id))
    res = Response()
    res.charset = 'utf-8'
    res.body = '<h1>Welcome to Magedu Python</h1>'.encode()
    return res


if __name__ == '__main__':
    ip = '127.0.0.1'
    port = 9999
    server = make_server(ip, port, App())
    try:
        server.serve_forever() # server.handle_request() 一次
    except KeyboardInterrupt:
        server.shutdown()
        server.server_close()
```

Application中，可以使用字典保存所有Router实例。因为每一个Router实例的前缀不同，完全可以使用前缀为key，Router实例为value组成字典。这样，以后在`__call__`方法中，就不需要遍历列表了，只需要提取request的path的前缀就可以和字典的key匹配了，这样提高了效率。
目前这么做的原因是一级目录本身不是很多，就几个，所以不用字典也可以。

框架处理流程

客户端发来HTTP请求，被WSGI服务器处理后传递给App的`__call__`。
App中遍历已注册的Routers，Routers的match来判断是不是自己能处理，前缀匹配，就看注册的规则（当然规则被装饰器已经转换成了命名分组的正则表达式了）。
如果由某一个注册的正则表达式匹配，就把获取的参数放到request中，并调用注册时映射的handler给它传入request。
handler处理后，返回response。App中拿到这个response的数据，返回给最初的wsgi。
如果handler返回仅仅是数据，将这些数据填入一个HTML中，将新生成的HTML字符串返回客户端，这就是网页。这种技术就是模板技术。