Day 9 - 编写API

374次阅读

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入http://localhost:9000/blog/123，就可以看到id为123的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取http://localhost:9000/api/blogs/123，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个@api来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

```
@api
@get('/api/users')
def api_get_users():
    users = User.find_by('order by created_at desc')
    # 把用户的口令隐藏掉:
    for u in users:
        u.password = '******'
    return dict(users=users)
```

所以，@api这个decorator只要编写好了，就可以把任意的URL处理函数变成API调用。

新建一个apis.py，编写@api负责把函数的返回结果序列化为JSON：

```
def api(func):
    @functools.wraps(func)
    def _wrapper(*args, **kw):
        try:
            r = json.dumps(func(*args, **kw))
        except APIError, e:
            r = json.dumps(dict(error=e.error, data=e.data, message=e.message))
        except Exception, e:
            r = json.dumps(dict(error='internalerror', data=e.__class__.__name__, message=e.message))
        ctx.response.content_type = 'application/json'
        return r
    return _wrapper
```

@api需要对Error进行处理。我们定义一个APIError，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为internalerror。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还

需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入http://localhost:9000/api/users，就可以看到返回的JSON：