目录

HTTP后台端：RESTful API接口设计

- 最新版本：`v3.0`
- 更新时间：`20190530`

简介

整理过HTTP知识总结后，继续去整理`HTTP`的后台相关的技术。在服务器后台进行设计API接口时，目前最流行的风格（原则/标准/规范）就是`RESTful`，常简称为`REST`接口。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

Gitbook源码

- crifan/http_restful_api: HTTP后台端：RESTful API接口设计

如何使用此Gitbook源码去生成发布为电子书

详见：crifan/gitbook_template: demo how to use crifan gitbook template and demo

在线浏览

- HTTP后台端：RESTful API接口设计 book.crifan.com
- HTTP后台端：RESTful API接口设计 crifan.github.io

离线下载阅读

- HTTP后台端：RESTful API接口设计 PDF
- HTTP后台端：RESTful API接口设计 ePub
- HTTP后台端：RESTful API接口设计 Mobi

版权说明

鸣谢

感谢我的老婆陈雪的包容理解和悉心照料，才使得我`crifan`有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

更多其他电子书

本人`crifan`还写了其他`100+`本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

RESTful API简介

服务器后台设计API接口时，目前最流行的风格（原则/标准/规范）就是 `RESTful` ，往往简称为 `REST` 。

其中 `REST` = `REpresentational State Transfer`

- `REST` 直译：表现层状态转移
- `REST` 核心含义：无状态的资源
  - 资源的变化（CURD）都是通过操作去实现的
    - 资源可以用 URI 表示
    - 用不同的URI和方法，表示对资源的不同操作
      - 典型的：
        - `GET` ： 获取资源
        - `POST` ： 新建资源
        - `PUT` ： 更新资源
        - `DELETE` ： 删除资源

`REST` 接口设计的特点/要求

- 接口形式统一=Uniform Interface
- 无状态=Stateless
- 可缓存=Cacheable
- 客户端服务器架构=Client-Server
- 分层设计=Layered System
- [可选]按需执行=COD(Code on Demand)
  - 解释见：What is the code-on-demand constraint? - The RESTful cookbook

RESTful的通俗理解

借用某人的总结：

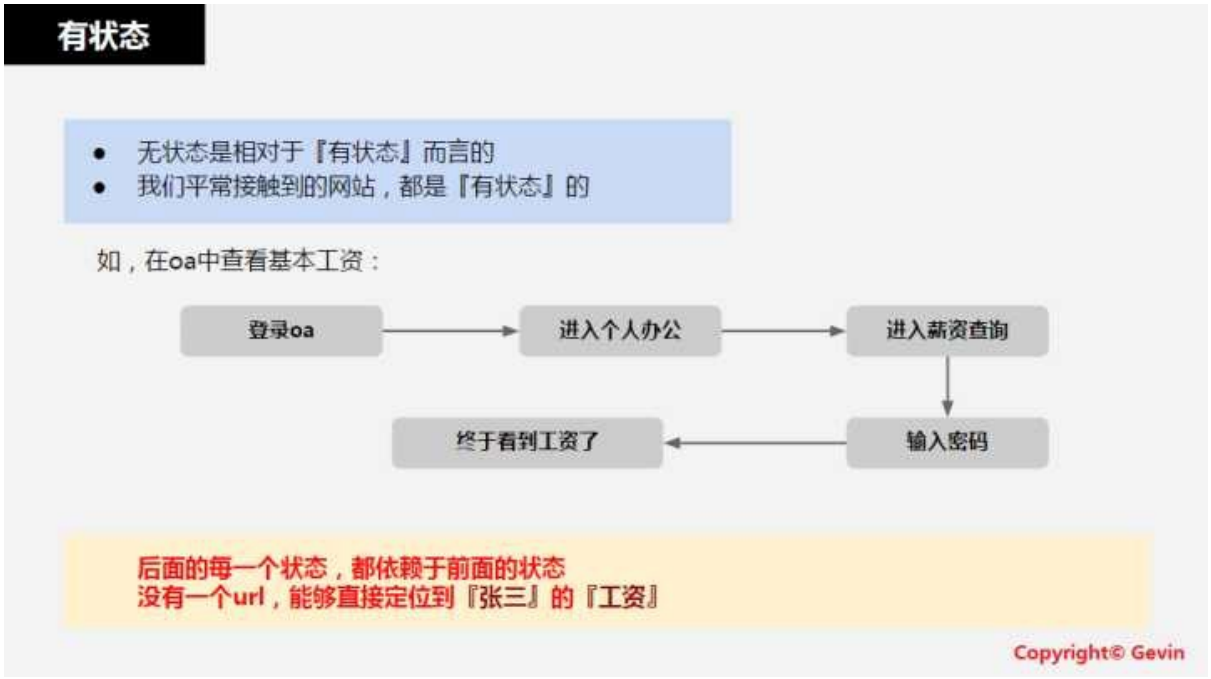
- 看 `url` 就知道要什么
- 看 `http method` 就知道干什么
- 看 `http status code` 就知道结果如何

其他类型的接口设计风格(含RESTful)

- `ROA` = `Resource Oriented Architecture`
- `RPC` = `Remote Procedure Call`
- `SOA` = `Simple Object Access Protocol`
- `REST` = `REpresentational State Transfer`

关于无状态的解释

有状态



无状态

RESTful API常见形式举例

下面找些常见的RESTful的API供参考和有个直观的概念：

RESTful的订单的API

HTTP 方法	行为	示例
GET	获取资源的信息	http://example.com/api/orders
GET	获取某个特定资源的信息	http://example.com/api/orders/123
POST	创建新资源	http://example.com/api/orders
PUT	更新资源	http://example.com/api/orders/123
DELETE	删除资源	http://example.com/api/orders/123

RESTful的客户的API

```
POST http://www.example.com/customers
POST http://www.example.com/customers/12345/orders

GET http://www.example.com/customers/12345
GET http://www.example.com/customers/12345/orders
GET http://www.example.com/buckets/sample

PUT http://www.example.com/customers/12345
PUT http://www.example.com/customers/12345/orders/98765
PUT http://www.example.com/buckets/secret_stuff

DELETE http://www.example.com/customers/12345
DELETE http://www.example.com/customers/12345/orders
DELETE http://www.example.com/bucket/sample
```

RESTful的待办事项TodoList的API

HTTP 方法	URL	动作
GET	http://[hostname]/todo/api/v1.0/tasks	检索任务列表
GET	http://[hostname]/todo/api/v1.0/tasks/[task_id]	检索某个任务
POST	http://[hostname]/todo/api/v1.0/tasks	创建新任务
PUT	http://[hostname]/todo/api/v1.0/tasks/[task_id]	更新任务
DELETE	http://[hostname]/todo/api/v1.0/tasks/[task_id]	删除任务

更多细节详见：【整理】TodoList待办事项：常被用于解释一个概念和框架如何应用）

RESTful API通用设计规则

对于RESTful的API设计，有些通用的设计规则。其实也可以叫做HTTP的各种方法的典型用法。

通用设计规则

资源：往往对应着后台系统中对象，比如一个用户User，一个待办事项todo item，一个任务Task等等

用对应的接口表示要对资源进行何种操作，想要实现什么目的：

HTTP Verb=HTTP Method=HTTP方法	操作类型=CRUD	可能返回的状态码	是否幂等 Idempotent	是否安全Safe
OPTIONS	询问该接口/端点支持哪些方法	200 OK	是	是

POST	Create创建	• 正常
		  ◦ 201 (Created)
		• 异常
		  ◦ 404 (Not Found)
		  ◦ 409 (Conflict)
			否	否

GET	Read读取	• 正常：
		  ◦ 200 (OK)
		• 异常：
		  ◦ 404 (Not Found)
			是	是

HEAD	同GET但是返回BODY为空	同GET	是	是

PUT	Replace替换	• 正常：
		  ◦ 200 (OK)
		• 异常
		  ◦ 405 (Method Not Allowed)
		  ◦ 204 (No Content)
		  ◦ 404 (Not Found)
			是	否

PATCH	Update/Modify更新	• 正常：
		  ◦ 200 (OK)
		• 异常
		  ◦ 405 (Method Not Allowed)
		  ◦ 204 (No Content)
		  ◦ 404 (Not Found)
			否	否

DELETE	Delete删除	• 正常：
		  ◦ 200 (OK)
		  ◦ 204 (No Content)
		• 异常：
		  ◦ 404 (Not Found)
		  ◦ 405 (Method Not Allowed)
			是	否

举例：RESTful的某个类似于外卖的项目的API

此处给出之前做过一个项目的RESTful的API，供参考：

- 用户
  - 获取用户ID：支持多个参数，根据参数不同返回对应的值
    - `GET /v1.0/open/userId?type=phone&phone={phone}`
    - `GET /v1.0/open/userId?type=email&email={email}`
    - `GET /v1.0/open/userId?type=facebook&facebookUserId={facebookUserId}`
  - 获取用户信息
    - `GET /v1.0/users/{userId}`
  - 修改用户信息
    - `PUT /v1.0/users/{userId}/info`
  - 修改密码
    - `PUT /v1.0/users/{userId}/password`
- 订单
  - 获取订单任务信息
    - `GET /v1.0/tasks/{taskId}/users/{userId}`
  - 发布任务
    - `POST /v1.0/tasks/users/{userId}`
  - 发单人确认任务信息
    - `PUT /v1.0/tasks/{taskId}/users/{userId}/confirmInfo`

常见问题

在具体设计接口时，会遇到一些具体的某些接口，到底应该用哪个 `HTTP` 的 `方法` 等方面的问题，整理如下供参考：

更新资源到底应该用 `PATCH` 还是 `PUT` ？

对于更新一个资源，很多人都会遇到这个问题，到底应该用 `PATCH` 还是 `PUT` ？

现解释如下：

HTTP的官方规范定义中，其实是：

HTTP方法	主要目的	传入参数	额外说明或注意事项
`PUT`	把某个资源的整体的信息替换掉	该资源的 全部 的字段	换言之：当某些字段没有传的话，则直接设置为 `null`
`PATCH`	把某个资源的部分信息更新掉	该资源的（你想要更新数据的那）部分的字段	`PATCH` 是在 `PUT` 之后才提出来，进入官方规范的。目的就是，只更新你传了值的那些字段，保留其他字段的已有的值

最佳实践==大家的实际做法

只不过，现在大家的常见的，实际的做法往往是：

用 `PUT` 实现了 `PATCH` 的效果：

使用 `PUT` ，但只传递 `部分` 字段，然后更新相应的字段

举例说明

有一个user用户，信息是：

```
{
  "name": "zhangsan",
  "email": "xxx@x.com"
}
```

按照PUT的规范来说，如果想要更新这个用户的邮箱，则需要传递：

```
PUT /user/some_user_id

{
  "name": "zhangsan",
  "email": "yyy@y.com"
}
```

否则如果传递：

```
PUT /user/some_user_id

{
  "email": "yyy@y.com"
}
```

REST标准中PUT就会认为，你没有其他字段，包括name，则会去设置name为null

-> 这样才是原本想要的replace替换的效果

而如果只是想要更新用户邮箱，应该去用PATCH，传递：

```
PATCH /user/some_user_id

{
  "email": "yyy@y.com"
}
```

然后才可以实现：只更新email字段，保留其他字段的已有的值

但是现在实际上大部人都是为了省事，也可以说作为最佳实践，用 `PUT` 代替 `PATCH`，传入

```
PUT /user/some_user_id

{
  "email": "yyy@y.com"
}
```

去实现（只）更新email邮箱，而保留其他（比如用户名name）的效果。

`POST` 创建资源成功后是返回 `200` 还是 `201`？

虽然按照REST协议来说，POST了新建的，应该返回201

但是为了统一处理，以及：很多开发者未必能很好的处理201，所以：

对于操作成功的，都正常返回200即可。

然后在response中返回对应的信息

比如：

新建的对象的详细信息

或者只是新建对象的id

结论：

- HTTP协议规范：`POST` 创建成功后，应该返回201，以及对应的新的资源的 `id`
  - 用户再通过新建资源的 `id`，去获取资源详情
- 最佳实践：大家实际的常见做法则是，直接返回新建资源的所有的详情的 `json`，其中包括 `id`

`DELETE` 应该返回什么？

综合官网和其他文档后，`REST` 服务器端对于 `DELETE` 请求应该返回什么 `状态码`，以及具体返回什么值，详细解释如下：

- `202` = `Accepted`：表示接受了此删除请求
  - 但是暂时还没执行
  - 或者目前执行删除动作，但是还没有完成
    - 之后（服务器端）会完成删除动作
- `204` = `No Content`：已经执行了删除动作，内容被删除了，没有内容了
  - response的body中是空的
  - 注意：如果要实现 `HATEOAS` 的REST的接口，则尽量避免DELETE返回 `204`
    - 详见：rest - RESTful API: Delete Entity - What should I return as result? - Stack Overflow
- `200` = `OK`：内容已删除，同时返回body信息，包含具体的删除的详情
  - response的body中包含额外关于删除的相关信息
    - 比如之前帖子中提到的，删除了具体哪个内容，已删除的状态，甚至删除了符合对应条件的多个内容等等

目前自己所理解的最佳实践是第三种：

- 返回状态码：`200` = `OK`
- 返回什么值：根据之前的最佳实践：code、message、data，应该返回：

```
{
  "code": 200,
  "message": "Question deleted for id 5c1777e1cc6df4563adf4a50",
  "data": {
    "deletedCount": 1
  }
}
```

且更加细节的做法是：当检测到想要删除一个已经删除的内容，则提示已经删除了：

```
{
  "code": 200,
  "message": "Question already deleted for id 5c1777e1cc6df4563adf4a50",
  "data": {
    "deletedCount": 0
  }
}
```

详见帖子

【已解决】Flask的REST接口的最佳实践中DELETE应该返回什么

HATEOAS

`HATEOAS` = `Hypermedia As The Engine Of Application State` = `超媒体即应用状态引擎`

谈到 `REST` 时，往往会提到这个 `HATEOAS` 。

什么是 `HATEOAS` ？

或者说：

- 为何会有 `HATEOAS` ？
- `HATEOAS` 有什么好处或作用？

找个例子来比喻，就容易理解了：

举例解释 `HATEOAS`

比如有个 `用户` 的对象，或者说资源，定义是：

```
class Customer {
    String name;
}
```

而普通的 `REST` ， `GET` 后返回的信息是：

```
{
    "name" : "Alice"
}
```

而简单点的 `HATEOAS` 则返回是：

```
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

好处是：

客户端，不需要再去问提供了接口的服务器端，就可以通过此 `HATEOAS` 返回的信息中知道一些额外的信息：

- `rel` ： 表示 `relationship` 关系。此处的 `self` 指的是就是对象 `Customer` 自己本身。
  - 而更加复杂点的情况中，可能会包含其他的对象，比如 `"rel":"customer"`
- `href` ： 当前对象的完整的 `url` 地址。

由此可以看出：

如果后台接口支持，或者说实现了 `HATEOAS` 这套标准（规范），那么：

调用接口的前端（移动端等），就可以像：

用户通过点击页面的 `href` 的链接地址，而跳转到其他网页，实现浏览网页的过程了。

-> 让调用 `REST` 的api就可以实现，类似于用户浏览网页的从一个页面跳转到另外一个页面的过程了

-> 而这种超链接方式的api用于告诉用户：该资源的只允许哪些操作（比如 `GET`，`POST`），以及不允许哪些操作（比如 `DELETE`）

-> 从而达到方便用户更加清楚使用你的接口的目的

其他 `HATEOAS` 实例

- Eve (Python)
- RESTHeart (Java)

关于 `HATEOAS` 的最佳实践：不用 `HATEOAS`

但是 `HATEOAS` 的缺点也很明显：

就把简单的返回的信息，搞的更加复杂了。

也因此实际在开发 `REST` 的api过程中，至少我是很少采用这个规范的。

当然，也有和我持同样观点的，比如这位

-》这样会让前端解析API时，倒是变得更加复杂了。显得多此一举和增加复杂度了。

而之前自己在折腾选择好的Flask的REST API的框架期间，本来觉得 `eve` 不错，后来就是由于发现 `eve` 默认使用 `HATEOAS`，把返回的 `json` 搞的太复杂，而放弃 `eve` 的。

顺带提及一点的是：

针对于 `HATEOAS` 标准，也还有是别人会用的。

所以一些流行的 `REST` 的框架中，有些也是内置支持了 `HATEOAS`。

比如：`Flask` 的 `REST` 框架：

- eve
- ripozo
- flask-marshmallow

另外，再贴出来一个复杂点的 `HATEOAS` 的例子，仅供了解：

```
{
    "content" : [ {
        "price" : 499.00,
        "description" : "Apple tablet device",
        "name" : "iPad",
        "links" : [ {
            "rel" : "self",
            "href" : "http://localhost:8080/product/1"
        } ],
        "attributes" : {
            "connector" : "socket"
        }
    }, {
        "price" : 49.00,
        "description" : "Dock for iPhone/iPad",
        "name" : "Dock",
        "links" : [ {
            "rel" : "self",
            "href" : "http://localhost:8080/product/3"
        } ],
        "attributes" : {
            "connector" : "plug"
        }
    } ],
```

```
    "links": [ {
        "rel": "product.search",
        "href": "http://localhost:8080/product/search"
    } ]
}
```

RESTful API工具和库

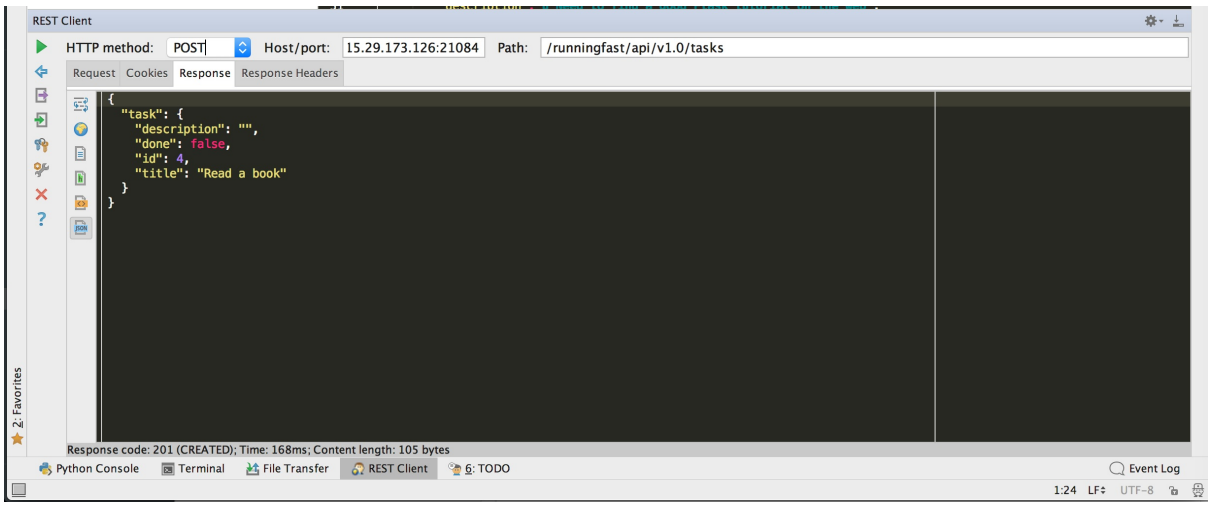
下面介绍和RESTful API开发、设计时相关的一些工具、库。

RESTful的API测试工具

Postman

详见另外的教程：API开发利器：Postman

PyCharm中的Restful API测试工具



详见：【整理】flask restful api 测试工具

Chrome插件：`Advanced REST client`

详见：［记录］chrome的websocket插件：Advanced REST client

Restful的API开发设计工具和框架

Python语言

在用Python语言去实现服务端的`REST`接口时，常见的框架和库是：

- Django + DRF= `Django REST Framework`
- Flask + Flask-RESTful

Javascript

NodeJS的ExpressJS

PHP

Slim

Ruby

Sinatra

.NET

Nancy - Lightweight Web Framework for .net

如何写API接口文档

如果你是后台API开发人员，往往会为了写清晰的API接口文档而发愁 此处，自己的建议和经验是：

- 方法1: 用 `markdown` 写API文档
- 方法2: 用 `Postman` 生成API文档
- 方法3: 用 `Swagger` 写（设计API接口的同时就可以生成出）API文档
  - 并可生成对应的后台和前端的代码
  - 剩下只需要编写业务逻辑代码即可

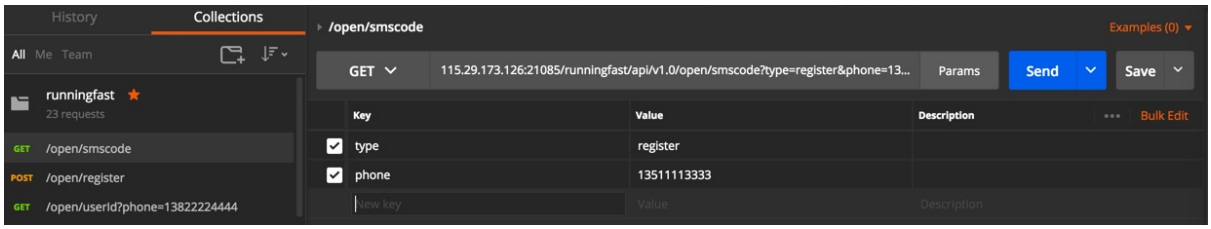
后来发现其他还有一些API文档工具，比如：

- docute
- ShowDoc
- 小幺鸡 接口文档管理工具
  - 也支持API接口调试

下面详细介绍这些写API文档的不同方法。

用Markdown写API文档

举例： 一个GET方法，用于获取验证码的接口： 在postman中已经调试完毕：



然后去（推荐）有道云笔记中编写markdown：

```
# API接口

## 注册

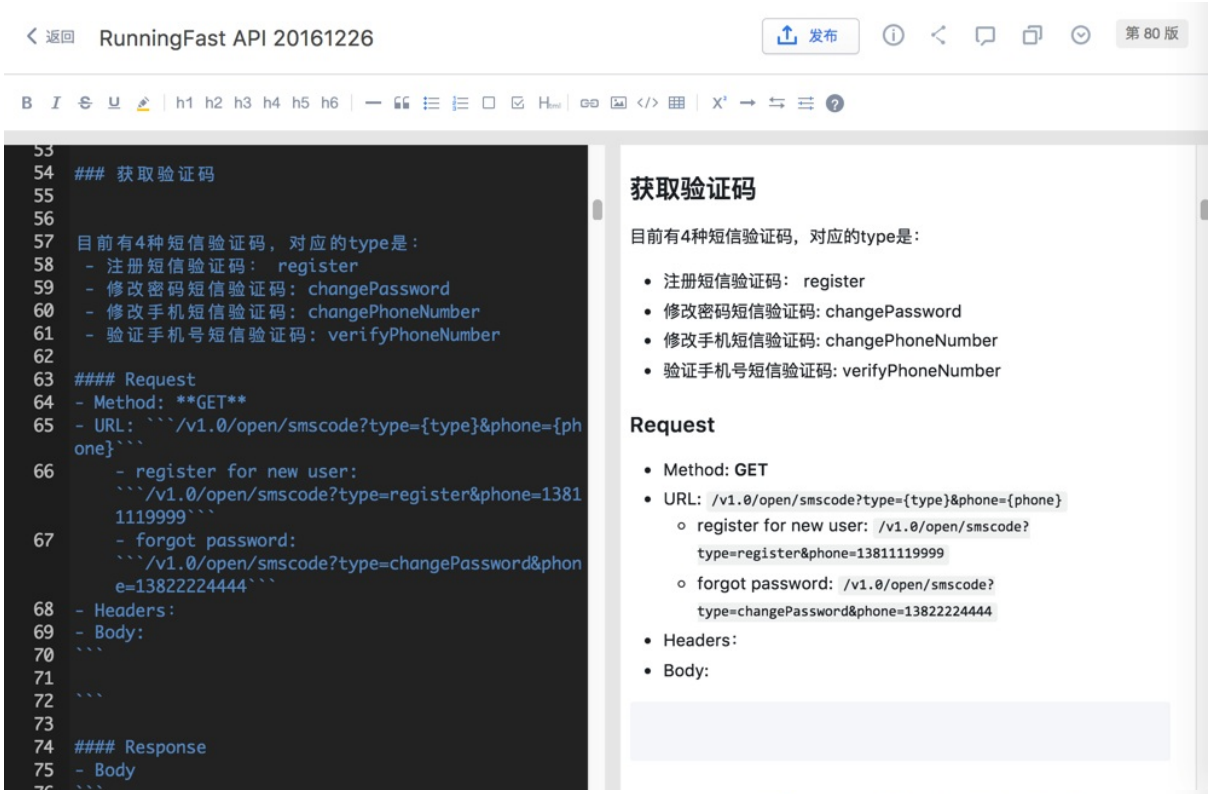
### 获取验证码

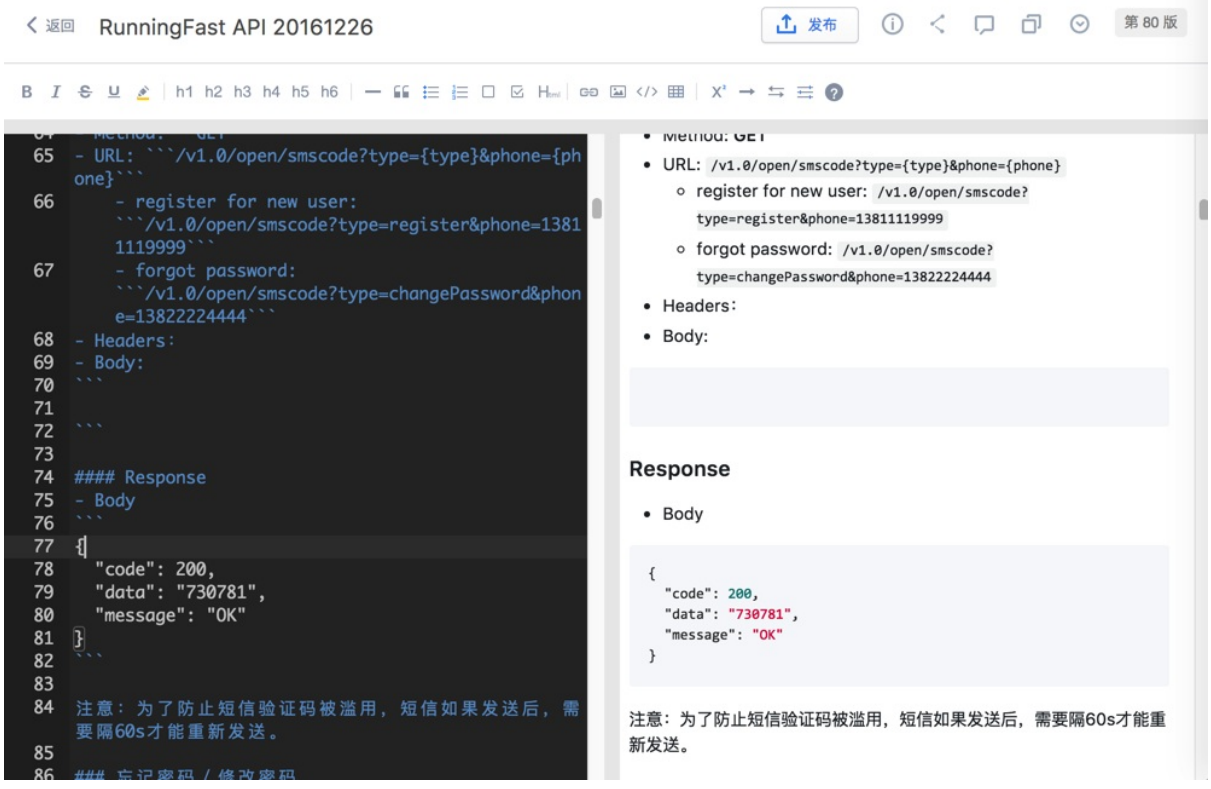
目前有4种短信验证码，对应的type是：
- 注册短信验证码： register
- 修改密码短信验证码：changePassword
- 修改手机短信验证码：changePhoneNumber
- 验证手机号短信验证码：verifyPhoneNumber
#### Request
- Method: **GET**
- URL: ```/v1.0/open/smscode?type={type}&phone={phone}```
    - register for new user: ```/v1.0/open/smscode?type=register&phone=13811119999```
    - forgot password: ```/v1.0/open/smscode?type=changePassword&phone=13822224444```
- Headers:
- Body:
```
```

#### Response
- Body
```
{
  "code": 200,
  "data": "730781",
  "message": "OK"
}
```

注意：为了防止短信验证码被滥用，短信如果发送后，需要隔60s才能重新发送。
```

对应的效果：





另外，再举个有request也有response的POST的例子：

```
### 创建新用户
#### Reuqest

- Method: **POST**
- URL: ```/v1.0/open/register```
- Headers: Content-Type:application/json
```

- Body:

```
{
    "phone" : "13511112222",
    "smsCode" : "730781",
    "email" : "crifan@webonn.com",
    "firstName" : "crifan",
    "lastName" : "Li",
    "password" : "654321",
    "facebookUserId" : "123907074803456"
}
```

Response

- Body

```
{
  "code": 200,
  "data": {
    "avatarUrl": "",
    "createdAt": "2016-10-24T20:39:46",
    "curRole": "IdleNoRole",
    "email": "crifan@webonn.com",
    "errandorRating": 0,
    "facebookUserId": "123907074803456",
    "firstName": "crifan",
    "id": "user-4d51faba-97ff-4adf-b256-40d7c9c68103",
    "isOnline": false,
    "lastName": "Li",
    "location": {
      "createdAt": null,
      "fullStr": null,
      "id": null,
      "latitude": null,
      "longitude": null,
      "shortStr": null,
      "updatedAt": null
    },
    "locationId": null,
    "password": "654321",
    "phone": "13511112222",
    "shareCodeCount": 0,
    "updatedAt": "2016-10-24T20:39:46"
  },
  "message": "new user has created"
}
```

markdown生成文档的效果：

< 返回 RunningFast API 20161226

创建新用户

Reuqest

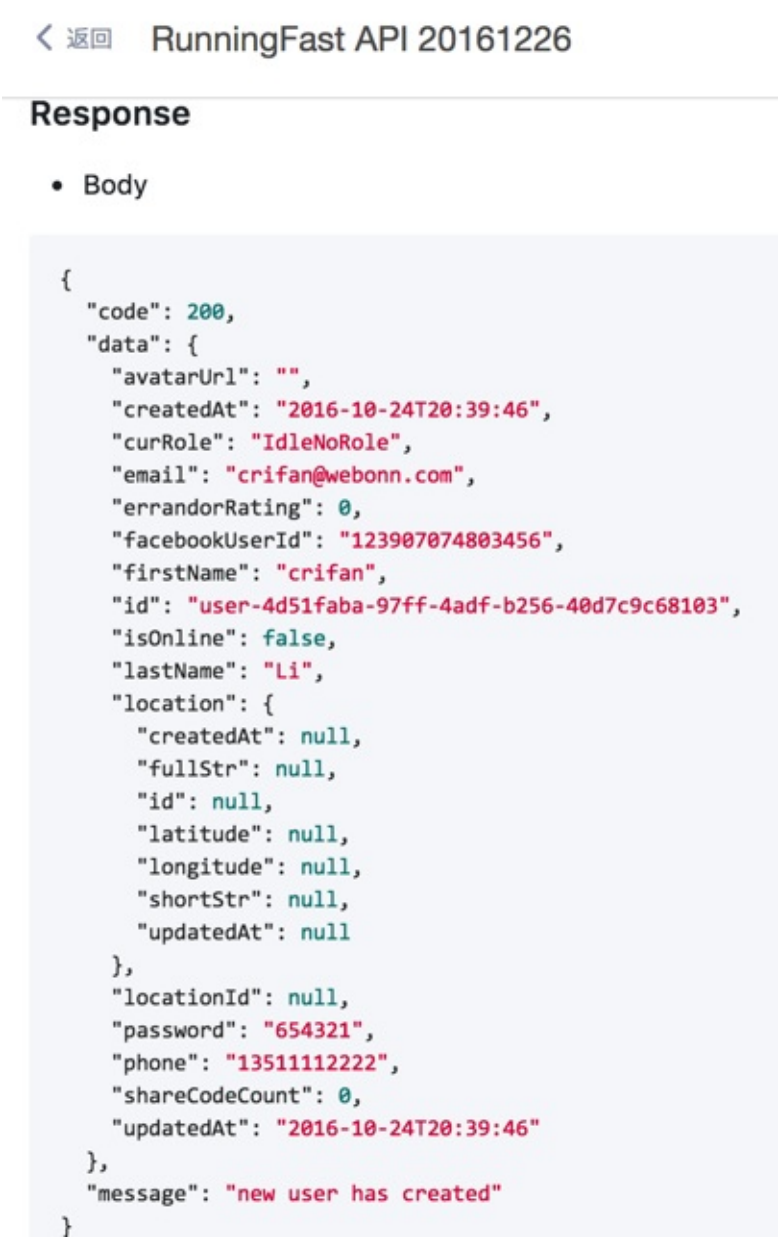
- Method: **POST**
- URL: `/v1.0/open/register`
- Headers： Content-Type:application/json
- Body:

```
{
    "phone" : "13511112222",
    "smsCode" : "730781",
    "email" : "crifan@webonn.com",
    "firstName" : "crifan",
    "lastName" : "Li",
    "password" : "654321",
    "facebookUserId" : "123907074803456"
}
```

Response

- Body

```
{
  "code": 200,
  "data": {
    "avatarUrl": "",
    "createdAt": "2016-10-24T20:39:46"
```



所以后续其他接口，均可参考上面的GET/POST等接口的写法，去写出对应的markdown的源文件，生成API文档后，效果还是不错的。

当然，也可以用其他Markdown编辑器去写md文件，去生成对应API文档。

另外，再附上，在写具体单个API接口之前的声明的部分：

```
# 文档说明
## 服务器API地址
前缀：
```http://115.29.173.126:21084/runningfast/api```

完整的API地址为：```前缀```+```具体接口路径```

比如，获取验证码都接口为：
```http://115.29.173.126:21084/runningfast/api``` + ```/v1.0/open/smscode```

->

```http://115.29.173.126:21084/runningfast/api/v1.0/open/smscode```
```
```

````
## 调用接口说明
- 如果参数格式是==JSON==的话：提交request请求时必须添加header头：==Content-Type:application/json==
- 请求中是否要包含头信息：==Authorization:{accesstoken}==
    - 接口中==包含==```/open/```的：不需要添加
    - 接口中==不包含==```/open/```：需要添加
        - 说明该接口都需要对应的权限才可以访问，所以需要在请求中包含头信息：```Authorization:{accesstoken}```
    - 当access token无效或者已过期时，返回：
```
{
  "code": 401,
  "message": "invalid access token: wrong or expired"
}
```

- 所有的接口的返回形式都是统一为：
    - 正常返回
```
{
  "code": 200,
  "message": "OK",
  "data": 某种类型的数据，比如字符串，数字，字典等等
}
```
    - 错误返回

```
{
  "code": 具体的错误码,
  "message": "具体的错误信息字符串"
}
```
````

文档效果：

文档说明

服务器API地址

前缀： `http://115.29.173.126:21084/runningfast/api`

完整的API地址为： `前缀` + `具体接口路径`

比如，获取验证码都接口为： `http://115.29.173.126:21084/runningfast/api` + `/v1.0/open/smscode`

->

`http://115.29.173.126:21084/runningfast/api/v1.0/open/smscode`

调用接口说明

- 如果参数格式是JSON的话：提交request请求时必须添加header头：Content-Type:application/json
- 请求中是否要包含头信息：Authorization:{accesstoken}
  - 接口中包含 `/open/` 的：不需要添加
  - 接口中不包含 `/open/` ：需要添加
    - 说明该接口都需要对应的权限才可以访问，所以需要在请求中包含头信息： `Authorization:{accesstoken}`
  - 当access token无效或者已过期时，返回：

```
{
```

返回 RunningFast API 20161226 编辑

- 接口中包含 `/open/` 的：不需要添加
- 接口中不包含 `/open/` ：需要添加
  - 说明该接口都需要对应的权限才可以访问，所以需要在请求中包含头信息： `Authorization:{accesstoken}`
- 当access token无效或者已过期时，返回：

```
{
  "code": 401,
  "message": "invalid access token: wrong or expired"
}
```

- 所有的接口的返回形式都是统一为：
  - 正常返回

```
{
  "code": 200,
  "message": "OK",
  "data": 某种类型的数据，比如字符串，数字，字典等等
}
```

```
- 错误返回
```

```
{
  "code": 具体的错误码,
  "message": "具体的错误信息字符串"
}
```

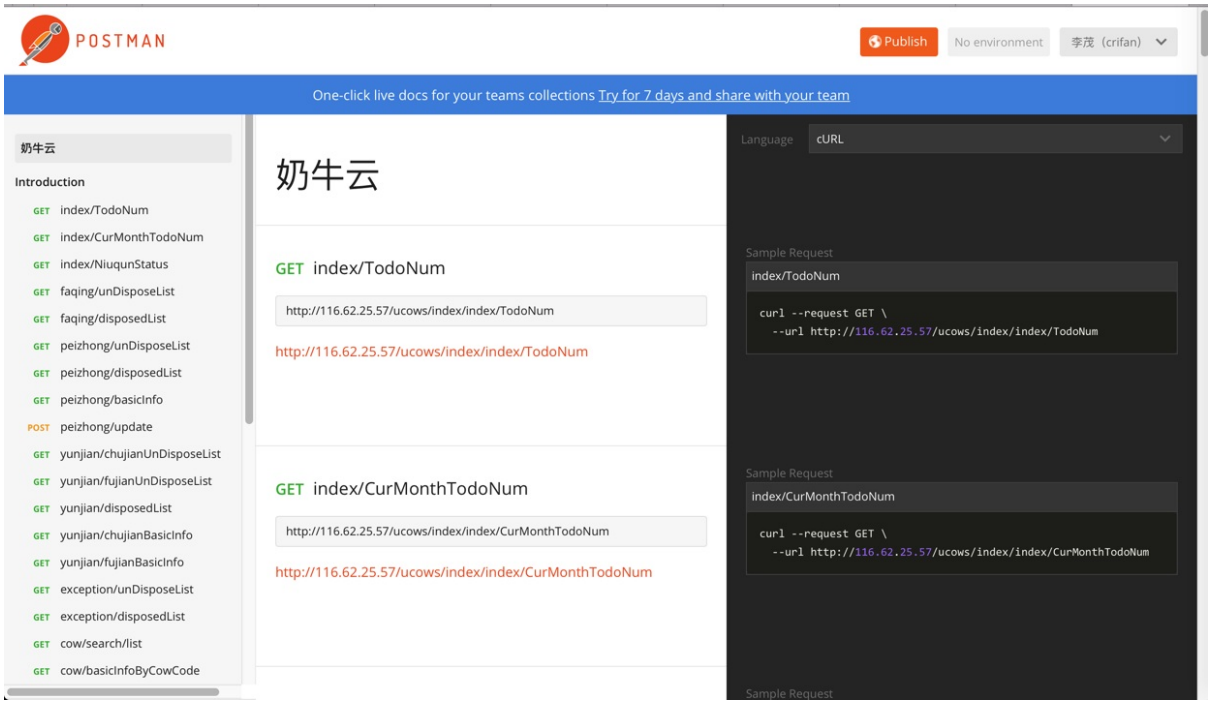
- 优点：简单易上手
- 缺点：后续API更新后，需要及时更新markdown的文档内容

用Postman生成API文档

步骤：

1. Collection
2. 鼠标移动到某个Collection
3. 点击 三个点
4. Publish Docs
5. Publish
6. Public URL
7. 别人打开这个Public URL即可查看API文档

效果：

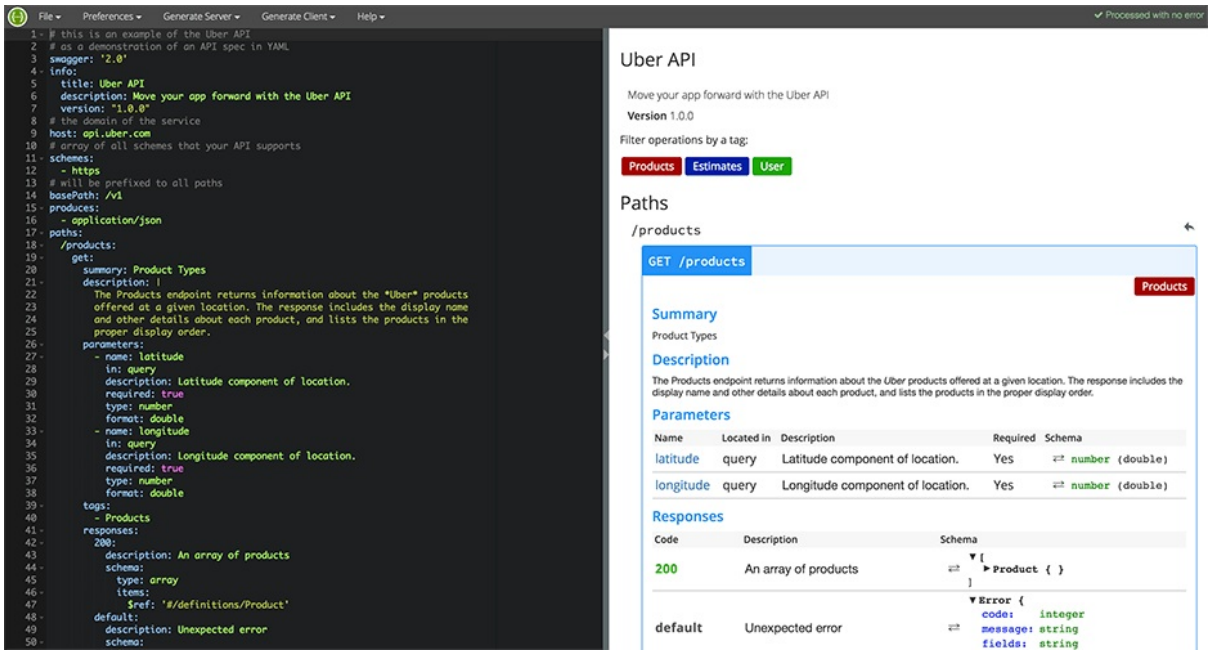


详见教程：API开发利器：Postman

- 优点：
  - 方便
    - 因为本身往往已用Postman调试接口，调试完毕后，即可发布
  - 及时更新文档
    - 同理，在后台代码更新后，用Postman调试无误后，即可再次点击发布即可，无须手动修改API文档
  - 美观
    - Postman生成的在线的API文档已足够清晰和美观
- 缺点：
  - 必须依赖于在Postman中调试接口

用Swagger写（设计API接口的同时就可以生成出）API文档

效果： API Development Tools | Swagger Editor | Swagger



详见： 【整理】swagger OpenAPI

- 优点：
  - 设计API接口的同时就是编写好了API文档
    - 因为有对应的工具可以直接生成API文档
  - 另外可以同时生成服务器端和客户端的代码
    - 剩下的只需要自己编写业务逻辑即可，支持N多种编程语言
  - 美观
    - 生成的API文档层次够清晰，够美观
- 缺点：
  - 必须用swagger去设计和编写API文档

RESTful API的心得和经验

此处把之前折腾过的RESTful的一些心得和经验整理出来，供参考。

不好的API设计风格

所有的资源的操作类型，都用 `POST`

听说过，有些偷懒的人，有坏习惯的人，竟然为了省事而去：把所有的资源的操作类型都用 `POST`

而不去管，实际上应该用 `GET` 、 `PUT` 、 `DELETE` 等操作。

不论你之前是否有这个坏习惯，都不要继续再有这种坏习惯和坏做法了。

而是应该根据资源的内容和操作的目的，分别用对应的 `HTTP` 的合适的方法： `GET` 、 `POST` 、 `PUT` 、 `DELETE`

在接口中添加GET/UPDATE等动词

比如：

- `GET /getUser`
- `POST /updateUser`
- `POST /cowfarm/cowfarmemp/new`
- `POST /cowfarm/cowfarmemp/update`
- `POST /cowfarm/cowfarmemp/delete/{id}`
  - 且返回值中包含了data和对应的字段

实际上不应该在接口中加这些动词，而应该通过接口的HTTP方法，GET/UPDATE，来表示接口的含义，比如改为：

- `GET /user`
  - 获取一个用户的信息
- `PUT /user`
  - 更新用户的信息
- `POST /cowfarm/employee`
  - 表示 新建 一个农场的雇员/员工
- `PUT /cowfarm/employee`
  - 表示 更新 农场雇员/员工的信息
- `DELETE /cowfarm/employee`
  - body中包含json参数
    - `{"id": xxx}`
  - 表示 删除 用户
  - 且返回值中，message，code应该正常返回，data就没必要返回了。

非改动资源的操作却设计为POST/PUT等方法

对于没有新增/更新/删除等去改动和影响资源的操作，HTTP的方法却设计为POST/PUT等

- 很多公司的后台开发人员，为了偷懒省事，所有的接口都用POST，包括本应该用GET的接口
- 或者是，对API接口设计规范不了解，把仅仅是获取、查询资源，不会改动资源的接口设计成POST

比如，不好的做法：

- 通过id获取task信息： `POST /task`
  - body参数： `{ "id": "1234" }`
- 查询出符合条件的任务： `POST /task/query`

    - 参数放在body中 `{"keyword":"xxx", "start": 0, "limit": 10}` 应该改为正常的做法：
- `GET /task/{id}`
    - 或： `GET /task?id=1234`
- `GET /task/query?keyword=xxxx&start=0&limit=10`
    - 其中GET的query string在客户端调用API接口时，往往不是手动加上去
    - 而是传递一个字典变量，然后用相关的encode函数去编码出来的
    - 详见：HTTP知识总结
    - 这样才能确保参数中包含特殊字符，服务器端也能正常接受，比如：
        - 空格 -> `%20`
        - 中文"李茂" -> 被UTF-8编码为 -> `%e6%9d%8e%e8%8c%82`

RESTful API接口返回数据的格式和风格

常见返回的数据的格式一般用JSON。

对应返回的内容，常见的做法是：

- `code`：http的status code
  - 如果有自己定义的额外的错误，那么也可以考虑用自己定义的错误码
- `message`：对应的文字描述信息
  - 如果是出错，则显示具体的错误信息
  - 否则操作成功，一般简化处理都是返回OK
- `data`
  - 对应数据的json字符串
    - 如果是数组，则对应最外层是[]的 `list`
    - 如果是对象，则对应最外层是{}的 `dict`

比如之前某项目中设计的返回的数据格式：

1. code是200 创建用户 `POST /v1.0/open/register` 返回：

```
{
 "code": 200,
 "message": "new user has created",
 "data": {
   "id": "user-4d51faba-97ff-4adf-b256-40d7c9c68103",
   "firstName": "crifan",
   "lastName": "Li",
   "password": "654321",
   "phone": "13511112222",
   "createdAt": "2016-10-24T20:39:46",
   "updatedAt": "2016-10-24T20:39:46"
   ......
 }
}
```

2. code是401

```
{
 "code": 401,
 "message": "invalid access token: wrong or expired"
}
```

分页= `paging` = `pagination`

写 `REST` 接口时，常会遇到一种情况是：

前端（web，移动端等）页面需要分多页，列出相关数据，供查看、编辑等操作。

随便截个某个管理后台中的某个数据的列表的图，好让大家有个更直观的理解：

而后台接口不可能，也不合适，一次性返回所有数据，而合理的做法和最佳实践是：分页

英文的说法一般是：`paging` = `pagination`

即每次返回一页数据，而想要获取更多数据，或下一页数据，则继续传入不同参数值即可

分页请求时的参数

而设计分页的API时，前端传入的参数中，最常见的，最重要的参数，就是：

- 当前是第几页= `page` = `cur_page` = `current_page` = `curPageNum`
- 每一页的个数= `per_page` = `numPerPage` =每一页的大小= `page_size` = `pageSize`

也有另外一种说法：

- 从哪个开始的= `start` ：
  - 另外一种叫法是：偏移量是多少= `offset`
- 返回的个数限制是多少= `limit`

总结一下就是：关于表示当前从哪里开始，要返回多少数据，有如下几类表示方法：

- 页数表示法：
  - `page` + `per_page`
    - `cur_page` + `per_page`
  - `page` + `page_size`
    - `page` + `pageSize`
    - `curPageNum` + `pageSize`
    - `curPageNum` + `numPerPage`

- 偏移量表示法：
  - `start` + `limit`
  - `offset` + `limit`

对应着前端页面 `GET` 请求所访问的典型的url是：

- `accounts?page=5&per_page=10`
- `accounts?limit=100&offset=300`
- `accounts?curPageNum=2&numPerPage=20`

分页返回的数据

分页返回的结果中，往往也包含了请求中所包含的基本参数，比如 `当前是第几页` 和 `每一页的个数` 。

另外肯定还要包含真正所需要的数据：

- `类型` ： 一般都是数组 `list`
- `命名方式` ： 有多种方案可选择：
  - 不固定：
    - 每个不同分页接口用自己的名字：一般采用对应的数据对象的名字，比如：
      - 用复数：比如 `tasks`
      - 用列表：比如 `taskList`
  - 固定：
    - 用固定的字段表示返回的结果：比如 `results` ， `items` ，

有时候，为了更加方便前端页面显示，比如希望知道：

- 总的个数有多少
- 是否还有下一页
  - 以此来控制页面上的下一页按钮是否可以点击，如果不可点击一般颜色采用灰色，否则显示正常的颜色

这种时候，往往还会加上一些其他相关的参数，比如 `总页数` ， `总个数` ， `是否还有下一页` ， `是否还有前一页` 等等。

这类相关参数，加上之前的基本参数，总结起来大概有这些：

- `page` = `cur_page` = `current_page` = `curPageNum` ： 当前是第几页
  - 说明：一般从 `0` 或 `1` 开始
  - 或：
    - `start` ： 从哪个开始的
      - 说明：一般从 `0` 开始
      - = `offset` ： （离最开始的）偏移量是多少
- `per_page` = `numPerPage` ： 每一页的个数
  - 说明：常见的值有 `10` ， `20` 等等
  - = `每一页的大小` = `page_size` = `pageSize`
  - 或： `返回的个数限制是多少` = `limit`
- `has_prev` = `hasPrev` ： 是否还有前一页
- `has_next` = `hasNext` ： 是否还有后一页
- `pages` = `totalPageNum` ： 总的页数
- `total` = `totalNum` = `counts` ： 总数=总个数=符合当前分页查询条件所返回的总个数
- `items` ： 真正的数据的列表

相关帖子

详见： 【已解决】Flask-Restful中如何设计分页的API

甚至还有些人会返回前一页和后一页的url：

- previous：前一页的url
- next：后一页的url

比如：

```
HTTP 200 OK
{
    "count": 1023
    "next": "https://api.example.org/accounts/?page=5",
    "previous": "https://api.example.org/accounts/?page=3",
    "results": [
       …
    ]
}

HTTP 200 OK
{
    "count": 1023
    "next": "https://api.example.org/accounts/?limit=100&offset=500",
    "previous": "https://api.example.org/accounts/?limit=100&offset=300",
    "results": [
       …
    ]
}
```

举例

下面给出一些实际的例子。

获取task任务的分页数据

下面给出之前某个 `Python` 的 `Flask + SQLAlchemy` 项目中查询一个 `task` =任务的分页查询的相关代码：

```
    curPageTaskList = None
    taskPagination = None

    if curRole == UserRole.Initiator:
        taskPagination = Task.query.filter_by(initiatorId=userId).paginate(
            page=curPageNum,
            per_page=numPerPage,
            error_out=False)
    elif curRole == UserRole.Errandor:
        taskPagination = Task.query.filter_by(errandorId=userId).paginate(
            page=curPageNum,
            per_page=numPerPage,
            error_out=False)

    paginatedTaskList = taskPagination.items

    paginatedTaskDict = {}
    for curIdx, eachTask in enumerate(paginatedTaskList):
        paginatedTaskDict[eachTask.id] = marshal(eachTask, task_fields)

    respPaginatedTaskInfoDict = {
        "curPageNum"    : taskPagination.page,
        "totalPageNum"  : taskPagination.pages,
        "numPerPage"    : taskPagination.per_page,
        "hasPrev"       : taskPagination.has_prev,
        "hasNext"       : taskPagination.has_next,
        "totalTaskNum"  : taskPagination.total,
        'tasks'         : paginatedTaskDict
    }
```

以及返回的结果：

```
{
    "code": 200,
    "message": "get task/orders ok",
    "data": {
        "curPageNum": 2,
        "hasNext": false,
        "hasPrev": true,
        "numPerPage": 10,
        "tasks": {
            "task-10b01105-ec53-41bb-810e-720ab468bdf7": {......},
            "task-da013992-e7aa-4ae9-8b6f-bdf621b9fbaa": {......},
            "task-f3c0c660-e7f5-4583-bab2-23c7006dadc4": {......},
            "task-f7a4d0df-3142-444b-a962-83660acd447f": {......}
        },
        "totalNum": 14,
        "totalPageNum": 2
    }
}
```

相关解释：

- 分页的变量选择用：`curPageNum` + `numPerPage`
- 返回的数据的列表：名字用 `tasks`，表示task任务的复数，真正返回的数据的列表

获取question题目的分页数据

代码：

```
class QuestionAPI(Resource):

    def get(self):
        log.info("QuestionAPI GET")

        respDict = {
            "code": 200,
            "message": "Get question ok",
            "data": {}
        }

        parser = reqparse.RequestParser()
        # parameters for get question list
        parser.add_argument('pageNumber', type=int, default 1, help "page number for get question list")
        parser.add_argument('pageSize', type=int, default settings.QUESTION_PAGE_SIZE,
                            help="page size for get question list")
        ...
        parser.add_argument('checkpointList', type=str, help="checkpoint for get question list")
        '''

        parsedArgs = parser.parse_args()
        log.debug("parsedArgs=%s", parsedArgs)

        if not parsedArgs:
            return genRespFailDict(BadRequest, "Fail to parse input parameters")

        '''
            # get question list
            pageNumber = parsedArgs["pageNumber"]
            pageSize = parsedArgs["pageSize"]
            if pageNumber < 1:
                return genRespFailDict(BadRequest.code, "Invalid pageNumber %d" % pageNumber)

            findParam = {}

            '''

            checkpointList = parsedArgs["checkpointList"]
            if checkpointList:
```

```
            maxFilterItemCount = 3
            filterItems = checkpointList.strip(', ').split(',')
            if len(filterItems) > maxFilterItemCount:
                return genRespFailDict(
                    BadRequest.code,
                    '{} items found in checkpoint parameter list, of which the max count is 3'.format(
                        len(filterItems))
                )
            checkPointFilterParam = [{'checkpoint.{}'.format(index): int(param)}
                                     for index, param in enumerate(filterItems)]
            if "$and" in findParam:
                findParam["$and"].extend(checkPointFilterParam)
            else:
                findParam["$and"] = checkPointFilterParam

        '''

        sortBy = "question_number"
        log.debug("findParam=%s", findParam)
        sortedQuestionsCursor = collectionEvaluationQuestion.find(findParam).sort(sortBy, pymongo.ASCENDING)
        totalCount = sortedQuestionsCursor.count()
        log.debug("search question: %s -> totalCount=%s", findParam, totalCount)
        if totalCount == 0:
            respData = {}
        else:
            # Note: for debug
            # follow will cause error:
            # pymongo.errors.InvalidOperation cannot set options after executing query
            # foundAllQuestions = list(sortedQuestionsCursor)
            # log.debug("foundAllQuestions=%s", foundAllQuestions)

            totalPageNum = int(totalCount / pageSize)
            if (totalCount % pageSize) > 0:
                totalPageNum += 1
            if pageNumber > totalPageNum:
                return genRespFailDict(BadRequest.code, \
                    "Current page number %d exceed max page number %d" % \
                    (pageNumber, totalPageNum))

            skipNumber = pageSize * (pageNumber - 1)
            limitedQuestionsCursor = sortedQuestionsCursor.skip(skipNumber).limit(pageSize)
            questionList = list(limitedQuestionsCursor)
            removeObjIdList = []
            for eachQuestion in questionList:
                eachQuestion = filterQuestionDict(eachQuestion)
                removeObjIdList.append(eachQuestion)

            hasPrev = False
            if pageNumber > 1:
                hasPrev = True
            hasNext = False
            if pageNumber < totalPageNum:
                hasNext = True

            respData = {
                "questionList": removeObjIdList,
                "curPageNum": pageNumber,
                "numPerPage": pageSize,
                "totalNum": totalCount,
                "totalPageNum": totalPageNum,
                "hasPrev": hasPrev,
                "hasNext": hasNext,
            }

        respDict["data"] = respData
        return jsonify(respDict)
```

请求：

```
GET /question?pageNumber=1&pageSize=10&checkpointList=73,83,85
```

响应：

```
{
    "code": 200,
    "data": {
        "curPageNum": 1,
        "hasNext": false,
        "hasPrev": false,
        "numPerPage": 10,
        "questionList": [
            {
                "_id": "5c628227bfaa44aa7b2f56a5",
                "active": "Y",
                "audio": "",
                ...
            },
            {
                "_id": "5c628261bfaa44aa7b2f56aa",
                "active": "Y",
                "audio": "",
                ...
            },
            ...
        ],
        "totalNum": 3,
        "totalPageNum": 1
    },
    "message": "Get question ok"
}
```


获取book列表的分页数据

请求：

```
GET /list.vpage?page_size=20&current_page=3
```

响应，返回的json：


```
{
    "success": true,
    "total_page": 15,
    "total": 300,
    "data": [
        {
            "id": "PBP_10300000138949",
            "name": "Cool Cat",
            ...
        },
        ...
        {
            "id": "PBP_10300000181801",
            "name": "Red Ben",
            ...
        }
    ],
    "current_page": 3,
    "page_size": 20,
    "selected_lexiler": "BR200L-300L",
    "selected_tag": ""
}
```

其他RESTful API心得

调用api时要有http的前缀，否则出错 `Error Domain`

关于其他（比如移动）端去调用REST的api时，要注意一点的是：

记得url地址写全了，最前面要加上 `http` 或 `https` 的前缀， 比如：

`http://115.29.173.126:21084/runningfast/api/v1.0/open/smscode`

否则，如果漏了，变成：

`115.29.173.126:21084/runningfast/api/v1.0/open/smscode`

就会报错：

`FAILURE: Error Domain=NSURLErrorDomain Code=-1002 unsupported URL` 或 `URL not supported`

详见： 【已解决】iOS端用Alamofire访问Flask的rest的api出错：Error Domain=NSURLErrorDomain Code=-1002

api中是否一定要加版本号?

如果是为了设计长期稳定的API接口，则最好是加上版本号v1.0这种写法 `http://[hostname]/todo/api/v1.0/` 但是往往中小型项目不需要这么长期维护和不需要迭代太多版本，则可以考虑不需要版本号，则可以写成：`http://[hostname]/todo/api/` 即可。

另外了解到： 有些的做法是把API的版本号v1，放到request header中。 ->github就是这么做的：Media Types | GitHub Developer Guide

设计Restful的接口时，尽量用复数，且统一

即，用 `/artists` 而不要用 `/artist`

如果有多个对象，用模块化逻辑，嵌套资源去设计接口

举例：

获取某个（内部id为8的）歌手的所有的专辑： `GET /artists/8/albums`

是否一定要严格按照Restful的规则，用对应的http的method实现对应的功能?

一般来说，不用非常严格的依照规则，尤其是

- UPDATE/PATCH 去更新修改资源
  - 往往为了简化，用PUT表示 更新修改资源即可

不过，有些项目，对方本身就要求设计接口时，严格按照Restful的规则来设计，这时最好用UPDATE/PATCH去更新修改资源。

其他的，见上面的表格总结，典型的是：

- `GET` 获取资源
- `POST` 新建资源
- `PUT` 更新/修改 资源
- `DELETE` 删除资源

POST时body中无参数时不应该添加 `Content-Type:application/json` 的Header

比如 `POST` 时,如果没有Body内容参数传递时，Header中就不要包含 `Content-Type:application/json`

否则，某些服务器就会返回错误。

比如Flask的Flask-Restful的接口就会自动返回：

```
code = 0 message = Failed to decode JSON object: No JSON object could be decoded
```

-》其意思是，你指定了

```
Content-Type:application/json
```

所以框架就会去尝试从Body中找JSON字符串，去解析参数

但是发现Body是空的，没有可用的JSON字符串待解析，所以报这个错误。

附录

下面列出相关参考资料。

参考资料

- 怎样用通俗的语言解释什么叫 REST，以及什么是 RESTful？ - 计算机网络 - 知乎
- 使用 Python 和 Flask 设计 RESTful API — Designing a RESTful API with Python and Flask 1.0 documentation
- restful update put When to use PUT or POST - The RESTful cookbook
- PUT Versus POST > RESTful APIs in the Real World Course 1 | KnpUniversity
- http - PUT vs. POST in REST - Stack Overflow
- HTTP Methods for RESTful Services
- REST API Tutorial
- What is REST?
- RFC 5789 - PATCH Method for HTTP
- RESTful Services Quick Tips
- RESTful API 设计指南 - 阮一峰的网络日志
- 设计一个务实的RESTful API
- RESTful API 编写指南
- API Creation - Full Stack Python
- REST best practices
- Methods
- When to use PUT or POST - The RESTful cookbook
- HTTP Methods [RESTful APIs Verbs] – REST API Tutorial
- Understanding REST
- What is REST?
- HTTP状态码 - 维基百科，自由的百科全书
- List of HTTP status codes - Wikipedia
- Overview of RESTful API Description Languages - Wikipedia
- Representational state transfer - Wikipedia
- Stateless protocol - Wikipedia
- HATEOAS - Wikipedia
- 表现层状态转换 - 维基百科，自由的百科全书
- 【已解决】iOS端用Alamofire访问Flask的rest的api出错：Error Domain=NSURLErrorDomain Code=-1002
- 【已解决】选择好的Flask的REST API的框架
- Understanding HATEOAS
- HATEOAS - Wikipedia
- What is HATEOAS and why is it important? - The RESTful cookbook
- REST HATEOAS教程(二)：HATEOAS规范
- 不要被名字吓到-RESTful、HATEOAS、Spring boot之整合 - 简书
- 使用 Spring HATEOAS 开发 REST 服务
- Why I Hate HATEOAS
- vertical-knowledge/ripozo: A tool for quickly creating REST/HATEOAS/Hypermedia APIs in python
- marshmallow-code/flask-marshmallow: Flask + marshmallow for beautiful APIs
- HATEOAS Driven REST APIs – REST API Tutorial
- pyeve/eve: REST API framework designed for human beings
- rest - Create request with POST, which response codes 200 or 201 and content - Stack Overflow
- Question about HTTP status 200 and 201 - Ruby on Rails / APIs - Code School Forum
- HTTP Status: 201 Created vs. 202 Accepted, by Ben Ramsey
- rest - RESTful API: Delete Entity - What should I return as result? - Stack Overflow
- RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
- Pagination - Django REST framework
- 【已解决】Flask-Restful中如何设计分页的API

- awesome-python-cn/README.md at master · jobbole/awesome-python-cn
- What is the code-on-demand constraint? - The RESTful cookbook
- How do I let users log into my RESTful API? - The RESTful cookbook
- php - RESTful API methods; HEAD & OPTIONS - Stack Overflow
- HTTP/1.1: Method Definitions