REST，以及RESTful的讲解

原创 九师兄 于 2018-04-21 18:59:08 发布 174086 收藏 818

文章目录

1.背景

1.传统下的API 接口

http是目前在互联网上使用最多的协议，没有之一。

可是http的创始人一直都觉得，在过去10几年来，所有的人都在错误的使用Http.这句话怎么说呢?

如果说你要删除一个数据，以往的做法通常是 `delete/{id}`

如果你要更新一个数据，可能是Post数据放Body，然后方法是 `update/{id}`， 或者是artichle/{id}?method=update

这种做法让Roy Fielding很暴燥，他觉得这个世界不该这样的，所有的人都在误解而且在严重错误的误解Http的设计初衷，好比是发明了火药却只用它来做烟花爆竹 。

那么正确的使用方式是什么呢？如果你要看Rest各种特性，你恐怕真的很难理解Rest， 但是如果你看错误的使用http的人倒底儿了哪些错，什么是Rest就特别容易理解了 。

七宗罪的第一条，混乱。

一万个人心里有一万个Url的命名规则，Url是统一资源定位符，重点是资源。而很多人却把它当成了万金油，每一个独立的虚拟的网页都可以随意使用，各种操作都能够迭加。这是混乱的来源之一。
比如：

```
https://localhost:8080/myweb/getUserById?id=1
https://localhost:8080/myweb/user/getById?id=1
https://localhost:8080/myweb/x/y?id=1
```

第二条，贪婪。

有状态 和 无状态 全部 混在一起 。特别是在购物车或者是登录的应用中，经常刷新就丢失带来的用户体验简直棒棒哒。每一个请求并不能单独的响应一些功能，很多的功能混杂在一起里。这是人性贪婪的本质，也是各种 Hack 的起源，只要能够把问题解决掉，总会有人用他认为最方便的方式去解决问题，比如说汽车门把手坏掉了直接系根绳子当把手，emmmm这样确实很棒啊。

第三条，无序。

返回的结果往往是很随意，各种错误信息本来就是用Http的状态码构成的，可是很多人还是喜欢把错误信息返回在返回值中。最常见的就是Code和Message，当然对于这一点，我个人是保留疑问的，我的观点是，Http本身的错误和服务器的内部错误还是需要在不断层面分开的，不能混在一起。可是在大神眼里并非如此。

那么怎么解决这些问题呢?

强迫症患者的福音就是 先颁规则， 第一个规则就是明确Url是什么，该怎么用 。就是所有的 Url本质 来讲，都应该 是一种资源 。一个独立的Url地址，就是对应一个独一无二的资源。怎么样？这种感觉是不是棒棒哒？一个冰淇淋，一个老师，一间房子，在Url上对应的都是一个资源，不会有多余的Url跟他对应，也不会表示有多个Url地址

注意，这里点的是Url地址，并不是单独的参数，他就是一个 `/room/{room_id}` 这样的东西，举个栗子, `/room/3242` 这就表示3242号房间 。这是一个清爽的世界啊，你想想，之前的Url是什么都要，我开房，可能是 `/open/room/3242` 我要退房可能是 `/exit/3242/room` ，我要打理房间，可能是 `room/3242?method=clean`. 够了！这些乱七八糟的东西全够了，让世界回归清爽的本质， 一间房，就是/room/3242 没有别的Url地址了。

在过去的混乱世界里，经常用的就是 Get和Post 。如果不是因为Get不支持大数据传输，我想连Post都不会有人使用。（想像一下Roy Fielding在愤怒的对着电脑屏幕喊，Http的Method一共有八个，你们为毛只逮着Get一只羊的毛薅薅薅薅薅）。

而对资源最常见的操作是什么？ CRUD ，对不对，就是创建，读，更新，删除。再看Http的Method？是不是非常完美？其实也怪Fielding老爷子一开始命名不准确，如果刚开始就是把 Get方法叫做Read ， Put方法叫做Update ， Post叫做Create这该多好 。。。

你用一个Get，大家又发现没什么限制没什么所谓，又很难理解Put和Post的差别，法无禁止即可为啊，呃，老爷子不要瞪我，我瞎说的。总之，这四种方法够不够你浪？你有本身找出来更多的对资源的操作来啊，我还有4个Method没用过呢。如果这4个真的不够了，有什么问题，大不了我再重新更改http协议啊。其实简单说，对于Rest理解到这里就够了。后续的东西，都是在这一条基础上空想出来的，比强迫症更强迫症，当然，无状态我是百分百支持的。以上的各种表述可能不太准确，也纯属是我的意淫和各种小道资料，并未考据，但是凭良心讲，我是早就看不惯黑暗年代里的Url命名风格了，所以当时最早接触到Rest的时候，瞬间就找到了真爱，我靠，这不就是我一直想要的答案吗？ 但是我一直想的仅仅是命名规范，从来没有把自己的思考角度放在一个url就是一个资源，所有的操作都是对资源的更改而言的角度上啊。 所以你能理解到的程度，更多的就是在于你要弄清楚你要解决的什么问题，如果你的问题只是理解Rest，恐怕你很理解，如果你的问题是怎么解决Url混乱的问题，你反而很快能弄懂了~

对比

```
https://localhost:8080/myweb/getDogs --> GET /rest/api/dogs 获取所有小狗狗
https://localhost:8080/myweb/addDogs --> POST /rest/api/dogs 添加一个小狗狗
https://localhost:8080/myweb/updateDogs/:dog_id --> PUT /rest/api/dogs/:dog_id 修改一个小狗狗
https://localhost:8080/myweb/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一个小狗狗
```

左边是我们写的，而且后台我们可能会写出很多返回值，而且各种各样的，比如
https://localhost:8080/myweb/addDogs

```
操作成功 或者 1
```

或者

```
操作失败 或者 0
```

这还要我们自己去解析，还要前端和后端去协商你返回的0是啥意识啊。但是REST返回值是标准的，比如

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: xxx

{
   "url" : "/api/categories/1",
   "label" : "Food",
   "items_url" : "/api/items?category=1",
   "brands" : [
         {
            "label" : "友臣",
            "brand_key" : "32073",
            "url" : "/api/brands/32073"
         }, {
            "label" : "乐事",
            "brand_key" : "56632",
            "url" : "/api/brands/56632"
         }
         ...
   ]
}
```

格式固定，第一行永远是操作失败或者成功的状态码，第二行是返回的类型，第三行内容的长度，第五行开始是内容。

这样我只需写一个程序解析返回的信息就可以了，可以重用，但是我们上面传统的不仅仅要协商，还有有不同的解析程序，稍微改变，就不能正常使用了。所以rest的明显更加通用。

列子2

```
1、获取文章

请求：
GET /blog/post/{postId} HTTP/1.1

```

```
响应：
HTTP/1.1 200 OK
{
    "title": "foobar",
    "content": "foobar",
    "comments": ["", "", ""]
}

2、发布文章

请求：
POST /blog/post HTTP/1.1
{
    "title": "foobar",
    "content": "foobar",
    "comments": ["", "", ""]
}

响应：
HTTP/1.1 201 CREATED
```

规则

```
GET    用来获取资源，
POST   用来新建资源（也可以用于更新资源），
PUT    用来更新资源，
DELETE  用来删除资源
```

例子

```
DELETE http://api.qc.com/v1/friends: 删除某人的好友 （在http parameter指定好友id）
POST http://api.qc.com/v1/friends: 添加好友UPDATE
http://api.qc.com/v1/profile: 更新个人资料

```

概念

REST 是面向资源的，这个概念非常重要，而资源是通过 **URI** 进行暴露。
URI 的设计只要负责把资源通过合理方式暴露出来就可以了。对资源的操作与它无关，操作是通过 HTTP动词来体现，所以REST 通过 URI 暴露资源时，会强调不要在 URI 中出现动词。

比如：左边是错误的设计，而右边是正确的

```
GET /rest/api/getDogs --> GET /rest/api/dogs 获取所有小狗狗
GET /rest/api/addDogs --> POST /rest/api/dogs 添加一个小狗狗
GET /rest/api/editDogs/:dog_id --> PUT /rest/api/dogs/:dog_id 修改一个小狗狗
GET /rest/api/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一个小狗狗
```

REST很好地利用了**HTTP**本身就有的一些特征，如**HTTP**动词、**HTTP**状态码、**HTTP**报头等等
REST API 是基于 HTTP的，所以你的API应该去使用 HTTP的一些标准。这样所有的HTTP客户端（如浏览器）才能够直接理解你的API（当然还有其他好处，如利于缓存等等）。REST 实际上也非常强调应该利用好 HTTP本来就有的特征，而不是只把 HTTP当成一个传输层这么简单了。

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: xxx

{
   "url" : "/api/categories/1",
   "label" : "Food",
   "items_url" : "/api/items?category=1",
   "brands" : [
         {
            "label" : "友臣",
            "brand_key" : "32073",
            "url" : "/api/brands/32073"
         }, {
```

```
            "label" : "乐事",
            "brand_key" : "56632",
            "url" : "/api/brands/56632"
        }
        ...
    ]
}
```

看这个响应，包含了http里面的状态码等信息。还会有http的一些报头。

```
Authorization 认证报头
Cache-Control 缓存报头
Cnotent-Type  消息体类型报头
......
```

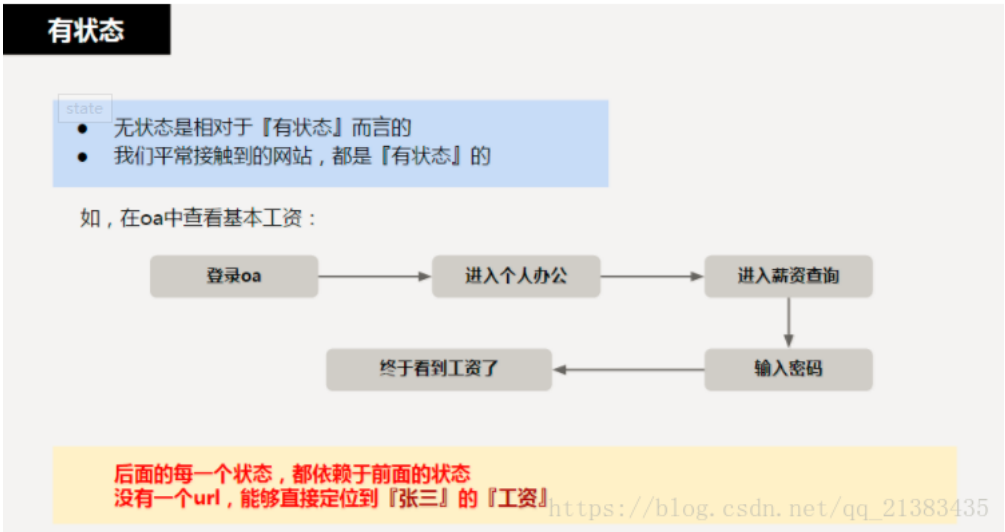
REST 系统的特征

1. 客户-服务器（**Client-Server**），提供服务的服务器和使用服务的客户需要被隔离对待。
2. 无状态（**Stateless**），来自客户的每一个请求必须包含服务器处理该请求所需的所有信息。换句话说，服务器端不能存储来自某个客户的某个请求中的信息，并在该客户的其他请求中使用。
3. 可缓存（**Cachable**），服务器必须让客户知道请求是否可以被缓存。（Ross：更详细解释请参考 理解本真的REST架构风格 以及 StackOverflow 的这个问题 中对缓存的解释。）
4. 分层系统（**Layered System**），服务器和客户之间的通信必须被这样标准化：允许服务器和客户之间的中间层（Ross：代理，网关等）可以代替服务器对客户的请求进行回应，而且这些对客户来说不需要特别支持。
5. 统一接口（**Uniform Interface**），客户和服务器之间通信的方法必须是统一化的。（Ross：GET,POST,PUT.DELETE, etc）
6. 支持按需代码（**Code-On-Demand，可选**），服务器可以提供一些代码或者脚本（Ross：Javascrpt，flash，etc）并在客户的运行环境中执行。这条准则是这些准则中唯一不必必须满足的一条。（Ross：比如客户可以在客户端下载脚本生成密码访问服务器。）

详细解释

无状态（**Stateless**）

所谓无状态的，即所有的资源，都可以通过URI定位，而且这个定位与其他资源无关，也不会因为其他资源的变化而改变。有状态和无状态的区别，举个简单的例子说明一下。如查询员工的工资，如果查询工资是需要登录系统，进入查询工资的页面，执行相关操作后，获取工资的多少，则这种情况是有状态的，因为查询工资的每一步操作都依赖于前一步操作，只要前置操作不成功，后续操作就无法执行；如果输入一个url即可得到指定员工的工资，则这种情况是无状态的，因为获取工资不依赖于其他资源或状态，且这种情况下，员工工资是一个资源，由一个url与之对应，可以通过HTTP中的GET方法得到资源，这是典型的RESTful 风格。



统一接口（Uniform Interface）

RESTful架构风格规定，数据的元操作，即CRUD(create, read, update和delete,即数据的增删查改)操作，分别对应于HTTP方法：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源，这样就统一了数据操作的接口，仅通过HTTP方法，就可以完成对数据的所有增删查改工作。

即：

```
GET（SELECT）：从服务器取出资源（一项或多项）。
POST（CREATE）：在服务器新建一个资源。
PUT（UPDATE）：在服务器更新资源（客户端提供完整资源数据）。
PATCH（UPDATE）：在服务器更新资源（客户端提供需要修改的资源数据）。
DELETE（DELETE）：从服务器删除资源。
```

演化

https://zhuanlan.zhihu.com/p/30396391?group_id=937244108725641216

优点&缺点

优点 是因为他对uri进行了限制，只用于定义资源。这样看起来比较容易理解。尤其是对简单的对象的增删改查，很好理解。

缺点 是因为这种限制，导致设计uri变得复杂了。尤其是复杂的关系，操作，资源集合， 硬性套用rest原则设计非常困难 。在rest基础上的HATEOAS，返回的json里增加了相应的关系和url。这也同样带来问题。好处是对简单的关系，的确可以通过url进一步处理。但对复杂的关系和操作，HATEOAS并不能胜任描述。 反而在单纯的数据中增加了一堆垃圾信息。

是什么?

REST是一个 标准，一种规范 ，遵循REST风格可以使开发的接口通用，便于调用者理解接口的作用。

参考：
https://www.zhihu.com/question/28557115
https://blog.igevin.info/posts/restful-architecture-in-general/