http://gityuan.com/2015/06/21/http-restful/#12-统一资源接口

RESTful 是一种非常流行的软件架构，或者说设计风格而非新的技术标准。提供了一组设计原则和约束条件，主要用于客户端与服务器的交互。RESTful架构更简洁，更有层次，更易于实现缓存等机制。

1.理解RESTful

RESTful, 全称Representational State Transfer。REST通常基于使用HTTP，URI，和XML以及HTML这些现有的广泛流行的协议和标准。要理解RESTful概念，需要明白下面的概念：

1.1 资源与URI

REST全称是表述性状态转移，表述指的就是资源。资源通过`URI(Uniform Resource Identifier)`来标示。URI的设计应该遵循可寻址性原则，具有自描述性。

这里以github网站为例，给出一些还算不错的URI：

- https://github.com/git
- https://github.com/git/git/blob/master/block-sha1/sha1.h
- https://github.com/git/git/pulls
- https://github.com/git/git/pulls?state=closed

关于URI设计技巧：

- 使用 _ 或 - 来让URI可读性更好，例如`http://www.github.com/blog/translate-reward-plan`。
- 使用 / 来表示资源的层级关系，例如上面的`/git/git/blob/master/block-sha1/sha1.h`
- 使用 ? 用来过滤资源，例如`/git/pulls?state=closed`用来表示git项目的所有推入请求中已经关闭的请求。
- 使用,或;表示同级资源关系，例如`/git/sha1/compare/ef7b53d18;bd638e8c1`

1.2 统一资源接口

RESTful架构应该遵循统一接口原则，统一接口包含了一组受限的预定义的操作，所有资源的访问接口应该使用标准的HTTP方法如GET，PUT，POST，DELETE，并遵循这些方法的语义。

如果按照HTTP方法的语义来暴露资源，那么接口将会拥有安全性和幂等性的特性，例如GET和HEAD请求都是安全的，无论请求多少次，都不会改变服务器状态。而GET、HEAD、PUT和DELETE请求都是幂等的，无论对资源操作多少次，结果总是一样的，后面的请求并不会产生比第一次更多的影响。

下面列出了GET，DELETE，PUT和POST的典型用法:

GET

安全且幂等 获取表示 变更时获取表示（缓存） 200（OK）- 表示已在响应中发出 204（无内容）- 资源有空表示 301（Moved Permanently）- 资源的URI已被更新 303（See Other）- 其他（如，负载均衡） 304（not modified）- 资源未更改（缓存） 400 （bad request）- 指代坏请求（如，参数错误） 404 （not found）- 资源不存在 406 （not acceptable）- 服务端不支持所需表示 500 （internal server error）- 通用错误响应 503 （Service Unavailable）- 服务端当前无法处理请求

POST

不安全且不幂等 使用服务端管理的（自动产生）的实例号创建资源 创建子资源 部分更新资源 如果没有被修改，则不过更新资源（乐观锁） 200（OK）- 如果现有资源已被更改 201（created）- 如果新资源被创建 202（accepted）- 已接受处理请求但尚未完成（异步处理） 301（Moved Permanently）- 资源的URI被更新 303（See Other）- 其他（如，负载均衡） 400（bad request）- 指代坏请求 404 （not found）- 资源不存在 406 （not acceptable）- 服务端不支持所需表示 409 （conflict）- 通用冲突 412 （Precondition Failed）- 前置条件失败（如执行条件更新时的冲突） 415 （unsupported media type）- 接受到的表示不受支持 500 （internal server error）- 通用错误响应 503 （Service Unavailable）- 服务当前无法处理请求

PUT

不安全但幂等 用客户端管理的实例号创建一个资源 通过替换的方式更新资源 如果未被修改，则更新资源（乐观锁） 200 （OK）- 如果已存在资源被更改 201 （created）- 如果新资源被创建 301（Moved Permanently）- 资源的URI已更改 303 （See Other）- 其他（如，负载均衡 ） 400 （bad request）- 指代坏请求 404 （not found）- 资源不存在 406 （not acceptable）- 服务端不支持所需表示 409 （conflict）- 通用冲突 412 （Precondition Failed）- 前置条件失败（如执行条件更新时的冲突） 415 （unsupported media type）- 接受到的表示不受支持 500 （internal server error）- 通用错误响应 503 （Service Unavailable）- 服务当前无法处理请求

DELETE
不安全但幂等 删除资源 200 （OK）- 资源已被删除 301 （Moved Permanently）- 资源的URI已更改 303 （See Other）- 其他，如负载均衡 400 （bad request）- 指代坏请求 404 （not found）- 资源不存在 409 （conflict）- 通用冲突 500 （internal server error）- 通用错误响应 503 （Service Unavailable）- 服务端当前无法处理请求

接下来再按一些实践中的常见问题

- POST和PUT在创建资源的区别：所创建的资源的名称(URI)是否由客户端决定。 例如为为博客增加一个android的分类，生成的路径就是分类名/categories/android，那么就可以采用PUT方法。
- 客户端不一定都支持这些HTTP方法：较古老的基于浏览器的客户端，只能支持GET和POST两种方法。妥协的解决方法，通过隐藏参数_method=DELETE来传递真实的请求方法等措施来规避。
- 统一资源接口对URI的意义：统一资源接口要求使用标准的HTTP方法对资源进行操作，所以URI只应该来表示资源的名称，而不应该包括资源的操作，如下是一些不符合统一接口要求的URI:
    - GET /getUser/1
    - POST /createUser
    - PUT /updateUser/1
    - DELETE /deleteUser/1

正确写法应该是 /User/1，不应该包含动词，具体的动作由请求方法来体现。

1.3 资源的表述

资源的表述是指对资源在特定时刻的状态的描述，客户端通过HTTP方法可以获取资源，更准确说是资源的表述而已。 资源在外界的具体呈现，可以有多种表述形式，在客户端和服务端之间传送的也是资源的表述，而不是资源本身。 例如文本资源可以采用html、xml、json等格式，图片可以使用PNG或JPG展现出来。

资源的表述包括数据和描述数据的元数据，例如，HTTP头”Content-Type”就是这样一个元数据属性。通过HTTP内容协商，客户端可以通过Accept头请求一种特定格式的表述，服务端则通过Content-Type告诉客户端资源的表述形式。

1.4 资源的链接

REST是使用标准的HTTP方法来操作资源的，但仅仅因此就理解成带CURD的Web数据库架构就太过于简单了。这种反模式忽略了一个核心概念：”超媒体即应用状态引擎”。 超媒体是什么?当你浏览Web网页时，从一个连接跳到一个页面，再从另一个连接跳到另外一个页面，就是利用了超媒体的概念：把一个个把资源链接起来.

要达到这个目的，就要求在表述格式里边加入链接来引导客户端。在《RESTful Web Services》一书中，作者把这种具有链接的特性成为连通性。下面我们具体来看一些例子。

下面展示的是github获取某个组织下的项目列表的请求，可以看到在响应头里边增加Link头告诉客户端怎么访问下一页和最后一页的记录。 而在响应体里边，用url来链接项目所有者和项目地址。

```
# Request
GET https://api.github.com/orgs/github/repos HTTP/1.1
Accept: application/json

# Response
HTTP/1.1 Status: 200 OK
Link: <https://api.github.com/orgs/github/repos?page=2>; rel="next",
      <https://api.github.com/orgs/github/repos?page=3>; rel="last"
Content-Type: application/json; charset=utf-8

[
  {
    "id": 1296269,
    "owner": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "somehexcode",
      "url": "https://api.github.com/users/octocat"
    },
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "description": "This your first repo!",
    "private": false,
    "fork": false,
    "url": "https://api.github.com/repos/octocat/Hello-World",
    "html_url": "https://github.com/octocat/Hello-World",
    ...
  }
]
```

又例如下面这个例子，创建订单后通过链接引导客户端如何去付款。

```
201 Created
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <cost>3.00</cost>
  <next xmlns="http://example.org/state-machine"
     rel="http://starbucks.example.org/payment"
     url="https://starbucks.example.org/payment/order/1234"
     type="application/xml">
</order>
```

上面的例子展示了如何使用超媒体来增强资源的连通性。很多人在设计RESTful架构时，使用很多时间来寻找漂亮的URI，而忽略了超媒体。所以，应该多花一些时间来给资源的表述提供链接，而不是专注于”资源的CRUD”。

1.5 状态的转移

REST原则中的无状态通信原则，并不是说客户端应用不能有状态，而是指服务端不应该保存客户端状态。

应用状态与资源状态

客户端负责维护应用状态，而服务端维护资源状态。客户端与服务端的交互必须是无状态的，并在每一次请求中包含处理该请求所需的一切信息。服务端不需要在请求间保留应用状态，只有在接受到实际请求的时候，服务端才会关注应用状态。这种无状态通信原则，使得服务端和中介能够理解独立的请求和响应。 在多次请求中，同一客户端也不再需要依赖于同一服务器，方便实现高可扩展和高可用性的服务端。

但有时候我们会做出违反无状态通信原则的设计，例如利用Cookie跟踪某个服务端会话状态，常见的像J2EE里边的JSESSIONID。这意味着，浏览器随各次请求发出去的Cookie是被用于构建会话状态的。当然，如果Cookie保存的是一些服务器不依赖于会话状态即可验证的信息（比如认证令牌），这样的Cookie也是符合REST原则的。

应用状态的转移

状态转移到这里已经很好理解了，“会话”状态不是作为资源状态保存在服务端的，而是被客户端作为应用状态进行跟踪的。客户端应用状态在服务端提供的超媒体的指引下发生变迁。服务端通过超媒体告诉客户端当前状态有哪些后续状态可以进入。

这些类似”下一页”之类的链接起的就是这种推进状态的作用——指引你如何从当前状态进入下一个可能的状态。这些类似”下一页”之类的链接起的

就是这种推进状态的作用——指引你如何从当前状态进入下一个可能的状态。

总结

本文从资源的定义、获取、表述、关联、状态变迁等角度，试图快速理解RESTful架构背后的概念。RESTful架构与传统的RPC、SOAP等方式在理念上有很大的不同，希望本文能对各位理解REST有所帮助。