1 接口路径设计

1.1 接口设计原则

1.1.1 URI指向的是唯一的资源对象

**示例:** 指向ID为youyou的Account对象

1. GET http://~/$version/accounts/youyou

1.1.2 URI可以隐式指向唯一的集合列表

**示例:** 隐式地指向trades list 集合

1. GET http://~/$version/trades/(list)
2. 等同于
3. GET http://~/$version/trades

1.1.3 聚合资源必须通过父级资源操作

**示例:** Profile是User的聚合资源，User有一个唯一且私有的Profile资源，只能通过User操作Profile。

1. 更新user\_id为123456的Profile资源
2. PUT http://~/$version/users/123456/profiles
3. Request Body:
4. {
5. "full\_name": "youyou",
6. "state": "Beijing",
7. "title": "Software engineer"
8. }

1.1.4 组合资源要避免资源路径嵌套

**示例:** 一个系统里面包含多个 applications，一个 application 又包含多个 users。那获取 user 资源的路径应该是怎样的？

下面是一个路径嵌套的例子：

1. GET http://~/$version/systems/:systemId/applications/:applicationId/users/:userId

这样做是不合理的，它会让你的接口变得越来越混乱和缺少灵活性。正确的做法是：

1. GET http://~/$version/systems/:systemId
2. GET http://~/$version/applications/:applicationId
3. GET http://~/$version/users/:userId/

1.2 Http Methods

| **HTTP Operation** | **Description** |
| --- | --- |
| GET | 获取，查找 |
| POST | 新增创建 |
| PUT | 更新 |
| PATCH | 部分更新 |
| DELETE | 删除 |

1.3 URL组成

1. 网络协议(*HTTP*, *HTTPS*)
2. 服务器地址
3. 版本
4. 接口名称
5. ?参数列表
6. GET https://github.com/v1/trades

**版本**

当服务被更多其他系统使用的时候，服务的可用性和上下兼容变得至关重要。被外部系统依赖的服务在升级时是一个非常麻烦的事情，既要发布新的接口，又要保留旧的接口留出时间让调用者去升级。在URL中加入Version标示能很好地解决上下兼容(新老版本共存)问题。

**示例1:** URL中新增了Path parameter

v1版本

1. GET http://~/v1/trades?user\_id=123456

v2版本

1. GET http://~/v2/:user\_id/trades

示例1中的user\_id参数在v2版本被加入到path parameter中，使用$version保证了v1和v2接口的共存。

**示例2:** 数据接口发生变化

v1版本

1. GET http://~/v1/accounts/youyou
2. Response Body:
3. {
4. "user\_name": "youyou",
5. "e\_mail": "youyou@gmail.com",
6. "state": "Beijing",
7. "title": "Software engineer"
8. }

v2版本

1. GET http://~/v2/accounts/youyou
2. Response Body:
3. {
4. "user\_name": "youyou",
5. "e\_mail": "youyou@gmail.com",
6. "profile": {
7. "state": "Beijing",
8. "title": "Software engineer"
9. }
10. }

示例2中的接口返回数据结构已经发生了变化。使用$version保证了v1和v2接口的共存。

1.4 URL定义限制

1. 不使用大写字母
2. 使用中线-代替下划线\_
3. 参数列表应该被encode过

1.5 接口分类

1.5.1 资源对象的CURD操作

1. GET http://~/$version/trades 获取trades列表
2. GET http://~/$version/trades/:id 根据id获取单个trade
3. POST http://~/$version/trades 创建trade
4. PUT http://~/$version/trades/:id 根据id更新trade
5. PATCH http://~/$version/trades/:id 根据id部分更新trade
6. DELETE http://~/$version/trades/:id 根据id删除trade

1.5.2 服务型接口

使用services标识，根据服务的属性选择http方法。

1. http://~/services/$version/server-name

1.5.3 系统设置

使用settings标识，根据服务的属性选择http方法。

1. http://~/settings/$version/server-name

**示例1:** 搜索

1. GET http://~/services/$version/search?q=filter?category=file

**示例2:** 任务队列操作

1. PUT http://~/services/$version/queued/jobs 往任务队列里面添加一个新的任务
2. DELETE http://~/services/$version/queued/jobs/:id 根据id删除任务

**示例3:** 更改界面语言环境

1. PUT http://~/settings/$version/gui/lang
2. {
3. "lang": "zh-CN"
4. }

**区分**

1. Microservices

Microservices是一个全新的概念，它主要的观点是将一个大型的服务系统分解成多个微型系统。每个微型系统都能独立工作，并且提供各种不同的服务。独立运行的特点使微型系统之间不会产生相互影响，其中的一个微型系统宕机并不会牵连到其他的微型系统。这种架构使分布式系统的节点数量大大提升。因为RESTful服务是无状态的，所以这种分解并不会带来状态共享的问题。

1. 路由规则(逻辑)

当我们需要对不同属性的接口做路由规则的时候，按功能划分接口是一个很好的方案。例如：我们要对系统设置接口设置增加更严格的调用限制。

1.6 缓存

网络接口相对于堆栈接口来说数据传输极其不稳定，尽可能地减少数据传输不仅能控制这种风险还能减少流量。使用缓存还能有效地提高后台的吞吐量。   
后台在响应请求时使用响应头E-Tag或Last-Modified来标记数据的版本，前台在发送请求时将数据版本通过请求头If-None-Match帮助后台判断缓存的使用。

**Request Header**

1. If-None-Match: 2390239059405940

**Response Header**

1. E-Tag: 2390239059405940
2. Last-Modified: 2014-04-05T14:30Z

1.7 Bookmarker

在实际的环境中，有大量的查询需求是相同的。将这些搜索需求标签化能降低使用难度也可以达到重用的目的。

**示例1:** 查找状态为关闭的订单

**普通方式**

1. GET http://~/$version/trades?status=closed&sorting=-created\_at

**Bookmarker**

1. GET http://~/$version/trades#recently\_closed

或

1. GET http://~/$version/trades/recently\_closed

1.8 HATEOAS

HATEOAS通过Web Linking的方式来描述程序的状态信息   
Link 主要包含以下属性:

| **Property** | **Description** |
| --- | --- |
| rel | 关联内容 |
| href | URL |
| type | 媒体类型 |
| method | Http Method |
| title | 标题 |
| arguments | 参数列表 |
| value | 返回值 |

Rel 可能为以下值:

| **Value** | **Description** |
| --- | --- |
| next | 下一步 |
| prev | 上一步 |
| first | 第一步，最前 |
| last | 最后一步，最后 |
| source | 来源 |
| self | 资源自身，相对于this |

Web Linking 可以通过两种方式传递至客户端:

**Http Header**

1. Link: <http://~/$version/trades?page\_no=10>; rel="next", <http://~/$version/trades?page\_no=19>; rel="last"

**Http JSON Body**

1. {
2. "links": [
3. {
4. "rel": "next",
5. "href": "http://~/$version/trades?page\_no=1"
6. },
7. {
8. "rel": "last",
9. "href": "http://~/$version/trades?page\_no=19"
10. }
11. ]
12. }

**示例1:** 用户注册业务

1. 用户填写E-Mail与密码
2. 完善用户资料

**Register Request**

1. POST http://~/$version/accounts
2. Headers:
3. Accept: application/json
4. Content-Type: application/json;charset=utf-8
5. Body:
6. {
7. "username": "youyou@gmail.com",
8. "e\_mail": "youyou@gmail.com",
9. "password": "balabala"
10. }

**Register Response**

1. Headers:
2. Content-Type: application/json;charset=utf-8
3. Status: 201 Created
4. Body:
5. {
6. "uri": "http://~/$version/accounts/youyou",
7. "identity": "youyou",
8. "created\_at": "2014-04-05T14:30Z",
9. "links": [
10. {
11. "rel": "next",
12. "href": "http://~/$version/accounts/youyou/profiles",
13. "method": "POST",
14. "title": "Editing Profiles",
15. "arguments": "status=editing"
16. }
17. ]
18. }

**Profile Request**

1. POST http://~/$version/accounts/youyou/profiles
2. Headers:
3. Accept: application/json
4. Content-Type: application/json;charset=utf-8
5. Body:
6. {
7. "full\_name": "youyou",
8. "state": "Beijing",
9. "title": "Software engineer"
10. }

**Profile Response**

1. Headers:
2. Content-Type: application/json;charset=utf-8
3. Status: 201 Created
4. Body:
5. {
6. "uri": "http://~/$version/accounts/youyou/profiles",
7. "identity": "youyou",
8. "created\_at": "2014-04-05T14:30Z"
9. }

**HATEOAS**

HATEOAS是Hypermedia as the Engine of Application State的缩写形式，中文意思为：超媒体应用状态引擎。它的核心思想是使用超媒体表达应用状态，与hypertext-driven思想是一致的。在此之前，我们大多数的程序业务控制在前台完成。例如：我们会在前台做注册流程，我们在前台判定下一步应该做什么，可以做什么。当使用HATEOAS时，这些状态流程控制都在应用程序的后台完成。我们使用超媒体来表达前台做完某一步骤之后可以做哪些? 这样一来，前台的任务就变得相当简单了，前台需要处理的是理解状态表述，数据收集和结果显示。

1.9 分页

**Request**

1. GET http://~/$version/trades?page=10&pre\_page=100

**Response**

Link Header

1. Link: <http://~/$version/trades?page=11&pre\_page=100>; rel="next", <http://~/$version/trades?page=19&pre\_page=100>; rel="last"

JSON Body

1. {
2. "links": [
3. {
4. "rel": "next",
5. "href": "http://~/$version/trades?page=11&pre\_page=100"
6. },
7. {
8. "rel": "last",
9. "href": "http://~/$version/trades?page=19&pre\_page=100"
10. }
11. ]
12. }

2 安全

2.1 调用限制

为保证服务的可用性应对服务进行调用过载保护

**Response Headers**

1. X-RateLimit-Limit: 3000 调用量的最大限制
2. X-RateLimit-Reset: 1403162176516 调用限制重置时间
3. X-RateLimit-Remaining: 299 剩余的调用量

2.2 安全验证

RESTful服务使用Oauth2的方式进行调用授权，使用http请求头Authorization设置授权码; 必须使用User-Agent设置客户端信息, 无User-Agent请求头的请求应该被拒绝访问。

**Request Header**

1. User-Agent: Data-Server-Client
2. Authorzation: Bearer 383w9JKJLJFw4ewpie2wefmjdlJLDJF

**使用Oauth2授权**

Oauth2的参与者为：客户端，资源所有者，授权服务器，资源服务器。客户端先从资源所有者得到授权码之后使用授权码从授权服务器得到token，再使用token调用资源服务器获取经过资源所有者授权使用的资源。这种授权方式的特点有：

* 1. 资源所有者可以随时撤销授权许可
  2. 可以通过撤销token拒绝客户端的调用
  3. 资源服务器可以拒绝客户端的调用

通过这三种方式可以做到对资源的严格保护。资源的访问权限也把握在资源所有者的手中，而不是资源服务器。

当然，Oauth2授权框架也允许受信任的客户端直接使用token调用资源服务器获取资源。这种灵活性完全取决于客户端类型和对资源的保护程度。

**授权码放在Http Header中**

1. WEB服务器对访问做记录已经成为了行业的一个标准，访问记录不仅可以用来做访问量统计还能用来做访问特征分析。互联网广告平台就是利用访问记录来做精准营销的。如果token(授权码)包含在URL中就有很大的安全风险。
2. 包含在URL中的token串可能被进行重定向传递。通过这两种方式入侵者可以不通过授权而使用泄漏的授权码访问那些受保护的数据，会造成数据泄漏的风险。

以Tomcat为例，访问日志为:

1. 127.0.0.1 - - [24/Jun/2014:14:38:04 +0800] "GET /v1/accounts/youyou?token=dgdreLJLJLER798989erJKJK HTTPS/1.1" 200 343

通过对访问日志的提取，很容易得到token信息。

3 数据设计

3.1 交互原则

1. 查询，过滤条件使用query string。
2. 用来描述数据或者请求的元数据放Header中，例如 X-Result-Fields。
3. Content body 仅仅用来传输数据。
4. 数据要做到拿来就可用的原则，不需要“拆箱”的过程。
5. 使用ISO-8601格式表达时间字段，例如: 2019-05-31T14:30Z。

3.2 结构

使用JSON格式传输数据，在http请求头和响应头申明Content-Type。返回的数据结构应该做到尽可能简单，不要过于包装。响应状态应该包含在响应头中！

**Request**

1. Accept: application/json
2. Content-Type: application/json;charset=UTF-8

**Response**

1. Content-Type: application/json;charset=UTF-8

**错误的做法**

1. {
2. "status": 200,
3. "data": {
4. "trade\_id": 1234,
5. "trade\_name": "Bala bala"
6. }
7. }

**正确的做法**

1. Response Headers:
2. Status: 200
3. Response Body:
4. {
5. "trade\_id": 1234,
6. "trade\_name": "Bala bala"
7. }

**示例1:** 创建User对象

1. POST http://~/$version/users
2. Request
3. headers:
4. Accept: application/json
5. Content-Type: application/json;charset=UTF-8
6. body:
7. {
8. "user\_name": "Andy Ai"
9. }
10. Response
11. status: 201 Created
12. headers:
13. Content-Type: application/json;charset=UTF-8
14. body:
15. {
16. "uri": "http://~/$version/users/1234",
17. "identity": 1234,
18. "created\_at": "2014-04-05T14:30Z",
19. "links": [
20. {
21. "rel": "next",
22. "href": "http://~/gui/users/1234"
23. }
24. ]
25. }

**为什么是JSON?**

JSON 是一种可以跨平台高扩展的轻量级的数据交换格式。易于人阅读和编写，同时也易于机器解析和生成。

3.3 属性定义限制

1. 不能使用大写(大小写友好)
2. 使用下划线\_命名(连接两个单词)
3. 属性和字符串值必须使用双引号""

3.4 提取部分字段

无状态服务器应该允许客户端对数据按需提取。在请求头使用X-Result-Fields指定数据返回的字段集合。   
例如：trade 有trade\_id, trade\_name, created\_at 三个属性，客户端只需其中的trade\_id与trade\_name属性。

**Request Header**

1. X-Result-Fields: trade\_id,trade\_name

3.5 子对象描述

数据里面的子对象使用URI描述不应该被提取，除非用户指定需要提取子对象

**示例:** trade里面的order对象   
**错误的做法**

1. {
2. "trade\_id": "123456789",
3. "full\_path": null,
4. "order": {
5. "order\_id": "987654321"
6. }
7. }

**正确的做法**

1. {
2. "trade\_id": "123456789",
3. "order": "http://~/$version/orders/987654321"
4. }

应用指定提取子对象，需要在请求头声明X-Expansion-Fields

**Request**

1. X-Expansion-Fields: true

**为什么要客户端指定提取子对象时才提取?**

懒模式服务能够最大程度地节省运算资源。虽然与客户端交互的次数有所增加，但是能做到按需提取，按需响应，这也是响应式设计的一大特点。客户端的用户行为模式无法真实地模拟，也就无法确定哪些资源需要做到一次性推送，让客户端按需使用是一个不错的方式。

**关于空字段**

应该在返回结果里面剔除空字段，因为null值传输到客户端并没有实际的含义，反而增加了占用空间。

**Tips**

使用HTTP Header时，优先使用合适的标准头属性。用X-作为前缀自定义一个头属性，例如: X-Result-Fields。

4 状态码&错误处理

4.1 应用状态码

| **Code** | **HTTP Operation** | **Body Contents** | **Description** |
| --- | --- | --- | --- |
| 102 Processing | GET, POST, PUT, DELETE, PATCH | 处理状态的信息 | 当前请求正在处理 |
| 200 Ok | GET, PUT | 资源 | 操作成功 |
| 201 Created | POST, PUT | 资源, 元数据 | 对象创建成功 |
| 202 Accepted | POST, PUT, DELETE, PATCH | 处理信息 | 请求已经被接受 |
| 204 No Content | DELETE, PUT, PATCH | N/A | 操作已经执行成功，但是没有返回数据 |
| 301 Moved Permanently | GET | link | 资源已被移除 |
| 303 See Other | GET | link | 重定向 |
| 304 Not Modified | GET | N/A | 资源没有被修改 |
| 400 Bad Request | GET, POST, PUT, DELETE, PATCH | 错误提示 | 参数列表错误(缺少，格式不匹配) |
| 401 Unauthorized | GET, POST, PUT, DELETE, PATCH | 错误提示 | 未授权 |
| 403 Forbidden | GET, POST, PUT, DELETE, PATCH | 错误提示 | 访问受限，授权过期 |
| 404 Not Found | GET, POST, PUT, DELETE, PATCH | 错误提示 | 资源，服务未找到 |
| 405 Method Not Allowed | GET, POST, PUT, DELETE, PATCH | 错误提示 | 不允许的http方法 |
| 406 Not Acceptable | GET, POST, PUT, DELETE, PATCH | 错误提示 | 媒体内容不符合要求 |
| 408 Request Timeout | GET, POST, PUT, DELETE, PATCH | 错误提示 | 请求超时 |
| 409 Conflict | GET, POST, PUT | 错误提示 | 资源冲突，重复的资源 |
| 415 Unsupported Media Type | GET, POST, PUT, DELETE, PATCH | 错误提示 | 不支持的数据(媒体)类型 |
| 422 Unprocessable Entity | GET, POST, PUT, PATCH | 错误提示 | 请求格式正确，但是由于含有语义错误，无法响应。 |
| 423 Locked | GET, POST, PUT, DELETE, PATCH | 错误提示 | 当前资源被锁定 |
| 429 Too Many Requests | GET, POST, PUT, DELETE, PATCH | 错误提示 | 请求过多被限制 |
| 500 Internal Server Error | GET, POST, PUT, DELETE, PATCH | 错误提示 | 系统内部错误 |
| 501 Not Implemented | GET, POST, PUT, DELETE, PATCH | 错误提示 | 接口未实现 |

4.2 容器状态码

容器状态码是指http容器的状态码，Web应用不应该使用或限制使用

| **Code** | **HTTP Operation** | **Body Contents** | **Description** |
| --- | --- | --- | --- |
| 303 | GET | link | 静态资源被移除,应用限制使用 |
| 503 | GET, POST, PUT, DELETE, PATCH | text body | 服务器宕机 |

4开头的错误用来表达来自于客户端的错误，例如: 未授权，参数缺失。5开头的错误用来表达服务端的错误，例如: 在连接外部系统(DB)发生的IO错误。

4.3 错误信息格式

错误信息应该包含下列内容:

1. 错误标题 message, 必须
2. 错误代码 error code, 必须
3. 错误信息 error message, 必须
4. 资源 resource, 可选
5. 属性 field, 可选
6. 文档地址 document, 可选

Error Code 尽可能做到简洁明了，提取异常的关键字并且使用下划线\_把它们连接起来。

**示例:** 调用频率超过限制，Response:

1. Headers:
2. Content-Type: application/json;charset=UTF-8
3. X-RateLimit-Limit: 3000
4. X-RateLimit-Reset: 1403162176516
5. X-RateLimit-Remaining: 0
6. {
7. "message": "Message title",
8. "errors": [
9. {
10. "code": "rate\_limit\_exceeded",
11. "message": "Too Many Requests. API rate limit exceeded",
12. "document": "https://developer.github.com/v3/gists/"
13. }
14. ]
15. }