RESTful

基本介绍及设计使用

序言：RESTful概述

- RESTful架构风格最初由Roy T. Fielding（HTTP/1.1协议专家组负责人）在其2000年的博士学位论文中提出。HTTP就是该架构风格的一个典型应用。从其诞生之日开始，它就因其可扩展性和简单性受到越来越多的架构师和开发者们的青睐。一方面，随着云计算和移动计算的兴起，许多企业愿意在互联网上共享自己的数据、功能；另一方面，在企业中，RESTful API（也称RESTful Web服务）也逐渐超越SOAP成为实现SOA的重要手段之一。时至今日，RESTful架构风格已成为企业级服务的标配。

- REST即Representational State Transfer的缩写，可译为"表现层状态转化”。

- REST最大的几个特点为：**资源**、**统一接口**、URI(统一资源标识符)和**无状态**。

序言：Fielding

- Fielding是一个非常重要的人，他是HTTP协议（1.0版和1.1版）的主要设计者、Apache服务器软件的作者之一、Apache基金会的第一任主席。所以，他的这篇论文一经发表，就引起了关注，并且立即对互联网开发产生了深远的影响。
- "本文研究计算机科学两大前沿----软件和网络----的交叉点。长期以来，软件研究主要关注软件设计的分类、设计方法的演化，很少客观地评估不同的设计选择对系统行为的影响。而相反地，网络研究主要关注系统之间通信行为的细节、如何改进特定通信机制的表现，常常忽视了一个事实，那就是改变应用程序的互动风格比改变互动协议，对整体表现有更大的影响。
- **我这篇文章的写作目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。**"

1. RESTful架构风格的特点
1.1.1资源

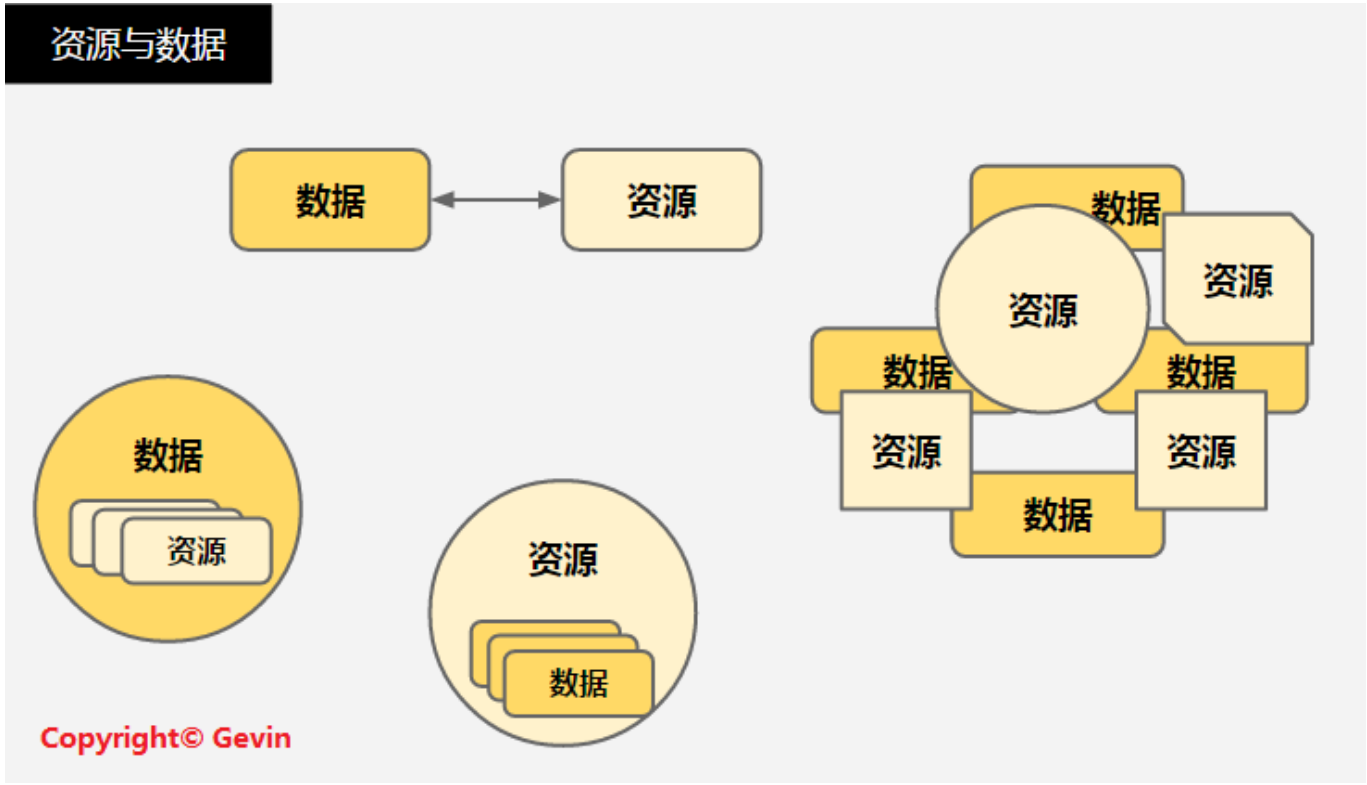
- 所谓“资源”，就是网络上的一个实体，或者说是网络上的一个具体信息。
- 它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。
- 资源总要通过某种载体反应其内容，文本可以用txt格式表现，也可以用HTML格式、XML格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现；JSON是现在最常用的资源表示格式。

1. RESTful架构风格的特点
1.1.1资源

- 资源是以json(或其他Representation)为载体的、面向用户的一组数据集，资源对信息的表达倾向于概念模型中的数据；
- 资源总是以某种Representation为载体显示的，即序列化的信息
- 常用的Representation是json(推荐)或者xml（不推荐）等
- Represntation 是REST架构的表现层
- 相对而言，数据（尤其是数据库）是一种更加抽象的、对计算机更高效和友好的数据表现形式，更多的存在于逻辑模型中

1. RESTful架构风格的特点
1.1.1资源

- 资源和数据关系如下：



1. RESTful架构风格的特点
1.1.2统一接口

- 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。
- 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。

1. RESTful架构风格的特点
1.1.3URI

- 可以用一个URI（统一资源定位符）指向资源，即每个URI都对应一个特定的资源。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或识别符。
- 一般的，每个资源至少有一个URI与之对应，最典型的URI即URL。

- URI与URL的区别：URI，是uniform resource identifier，统一资源标识符，用来唯一的标识一个资源。而URL是uniform resource locator，统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。

1. RESTful架构风格的特点
1.1.4无状态

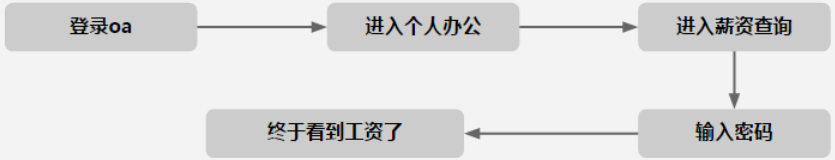
- 所谓无状态的，即所有的资源，都可以通过URI定位，而且这个定位与其他资源无关，也不会因为其他资源的变化而改变。
- 有状态和无状态的区别，举个简单的例子说明一下。如查询员工的工资，如果查询工资是需要登录系统，进入查询工资的页面，执行相关操作后，获取工资的多少，则这种情况是有状态的，因为查询工资的每一步操作都依赖于前一步操作，只要前置操作不成功，后续操作就无法执行；如果输入一个url即可得到指定员工的工资，则这种情况是无状态的，因为获取工资不依赖于其他资源或状态，且这种情况下，员工工资是一个资源，由一个url与之对应，可以通过HTTP中的GET方法得到资源，这是典型的RESTful风格。

1. RESTful架构风格的特点
1.1.4无状态

有状态

- 无状态是相对于『有状态』而言的
- 我们平常接触到的网站，都是『有状态』的

如，在oa中查看基本工资：



后面的每一个状态，都依赖于前面的状态
没有一个url，能够直接定位到『张三』的『工资』

无状态

对每个资源的请求，都不依赖于其他资源或其他请求

每个资源，都是可寻址的，都有至少一个url能对其定位

- **Application State**
- **Resource Stateless**

RESTful 架构下，工资可以通过以下url查询：

张三工资 http://oa.company.com/salary/zhangsan
李四工资 http://oa.company.com/salary/lee4

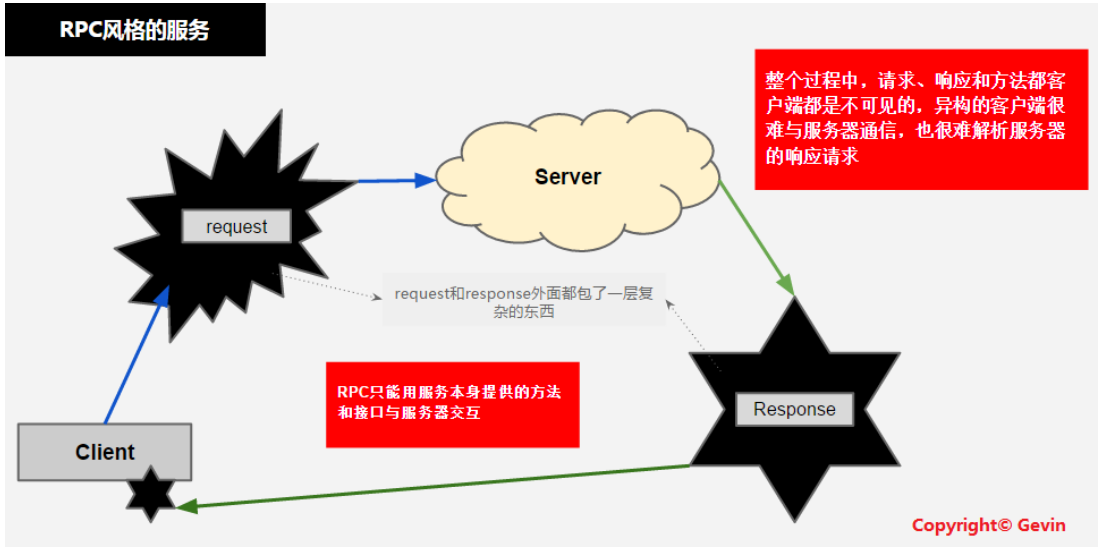
无状态更加方便客户端使用服务器的资源或服务

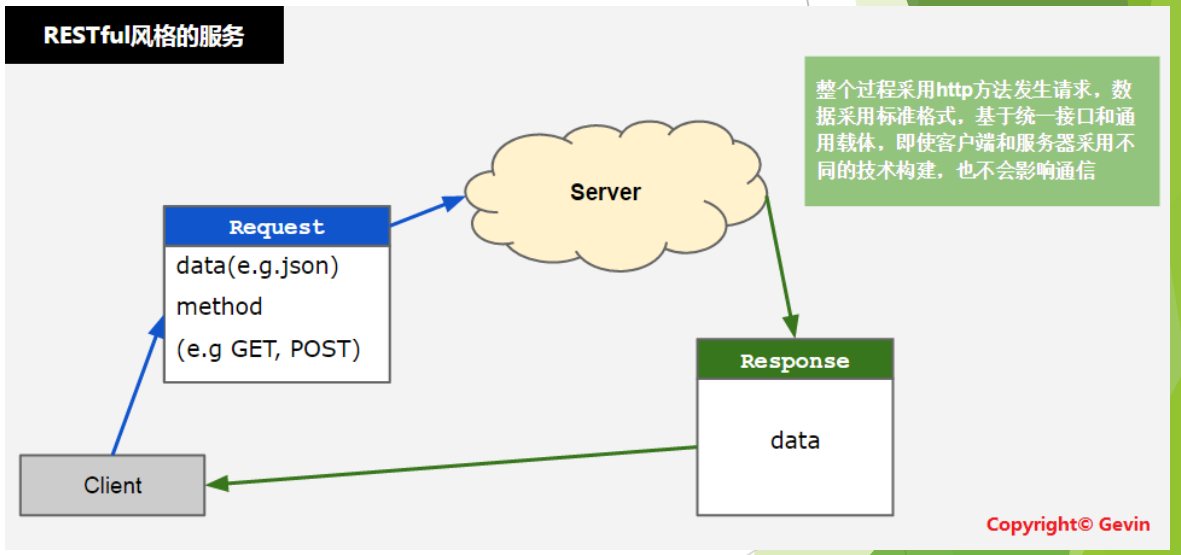
1. RESTful架构风格的特点
1.2 ROA、SOA、REST与RPC

- ROA即Resource Oriented Architecture，RESTful 架构风格的服务是围绕资源展开的，是典型的ROA架构（虽然“A”和“架构”存在重复，但说无妨），虽然ROA与SOA并不冲突，甚至把ROA看做SOA的一种也未尝不可，但由于RPC也是SOA，比较久远一点点论文、博客或图书也常把SOA与RPC混在一起讨论，因此，RESTful 架构风格的服务通常被称之为ROA架构，很少提及SOA架构，以便更加显式的与RPC区分。
- RPC风格曾是Web Service的主流，最初是基于XML-RPC协议（一个远程过程调用（remote procedure call，RPC)的分布式计算协议），后来渐渐被SOAP协议（简单对象访问协议（Simple Object Access Protocol））取代；RPC风格的服务，不仅可以用HTTP，还可以用TCP或其他通信协议。但RPC风格的服务，受开发服务采用语言的束缚比较大，如.NET框架中，开发web service的传统方式是使用WCF，基于WCF开发的服务即RPC风格的服务，使用该服务的客户端通常要用C#来实现，如果使用python或其他语言，很难实现可以直接与服务通信客户端；进入移动互联网时代后，RPC风格的服务很难在移动终端使用，而RESTful风格的服务，由于可以直接以json或xml为载体承载数据，以HTTP方法为统一接口完成数据操作，客户端的开发不依赖于服务实现的技术，移动终端也可以轻松使用服务，这也加剧了REST取代RPC成为web service的主导。

1. RESTful架构风格的特点
1.2 ROA、SOA、REST与RPC

- RPC与RESTful的区别如下面两个图所示：





1. RESTful架构风格的特点
1.3 本真REST与hybrid风格

- 通常开发者做服务相关的客户端开发时，使用的所谓RESTful服务，基本可分为本真REST和hybrid风格两类。本真REST即我上文阐述的RESTful架构风格，具有上述的4个特点，是真正意义上的RESTful风格；而hybrid风格，只是借鉴了RESTful的一些优点，具有一部分RESTful的特点，但对外依然宣称是RESTful风格的服务。（窃以为，正是由于hybrid风格服务混淆了RESTful的概念，才在RESTful架构风格提出了本真REST的概念，以为了划分界限:P）
- hybrid风格的最主流的用法是，使用GET方法获取资源，用POST方法实现资源的创建、修改和删除。hybrid风格之所以存在，据我了解有两种来源：一种情况是因为，某些开发者并没有真正理解何为RESTful架构风格，导致开发的服务貌合神离；而主流的原因是由于历史包袱 —— 服务本来是RPC风格的，由于上文提到的RPC的劣势及RESTful的优势，开发者在RPC风格的服务上又包装了一层RESTful的外壳，通常这层外壳只为获取资源服务，因此会按RESTful风格实现GET方法，如果客户端提出一些简单的创建、修改或删除数据的需求，则通过HTTP协议中最常用的POST方法实现相应功能。
- 因此，开发RESTful 服务，如果没有历史包袱，不建议使用hybrid风格。

2. 认证机制

- 由于OAUTH的严谨性和安全性，现在OAUTH已成为RESTful架构风格中最常用的认证机制，和RESTful架构风格一起，成为企业级服务的标配。
- 目前OAuth已经从OAuth1.0发展到OAuth2.0，但这二者并非平滑过渡升级，OAuth2.0在保证安全性的前提下大大减少了客户端开发的复杂性，因此，Gevin建议在实战应用中采用OAuth2.0认证机制。
- 现在网上关于OAuth的资料非常丰富，也有大量开源的第三方库实现了OAuth机制，不熟悉OAuth的同学从OAuth官网入手即可。

3. 如何设计RESTful API

3. 如何设计RESTful API
3.1 设计核心

- Request 和 Response （请求和响应）
- Serialization 和 Deserialization （序列化和反序列化）
- Validation （数据校验）
- Authentication 和 Permission （授权和许可）
- CORS （Cross-origin resource sharing，解决JS跨域问题）
- URL Rules （URL 规则）

3. 如何设计RESTful API
3.2 Request 和 Response

- RESTful API的开发和使用，无非是客户端向服务器发请求（request），以及服务器对客户端请求的响应（response）。本真RESTful架构风格具有统一接口的特点，即，使用不同的http方法表达不同的行为：
- 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

3. 如何设计RESTful API
3.3 Serialization 和 Deserialization

- Serialization 和 Deserialization即序列化和反序列化。RESTful API以规范统一的格式作为数据的载体，常用的格式为json或xml，以json格式为例，当客户端向服务器发请求时，或者服务器相应客户端的请求，向客户端返回数据时，都是传输json格式的文本，而在服务器内部，数据处理时基本不用json格式的字符串，而是native类型的数据，最典型的如类的实例，即对象（object），json仅为服务器和客户端通信时，在网络上传输的数据的格式，服务器和客户端内部，均存在将json转为native类型数据和将native类型数据转为json的需求，其中，将native类型数据转为json即为序列化，将json转为native类型数据即为反序列化。虽然某些开发语言，如Python，其原生数据类型list和dict能轻易实现序列化和反序列化，但对于复杂的API，内部实现时总会以对象作为数据的载体，因此，确保序列化和反序列化方法的实现，是开发RESTful API最重要的一步准备工作

3. 如何设计RESTful API
3.4 Validation

- Validation即数据校验，是开发健壮RESTful API中另一个重要的一环。仍以json为例，当客户端向服务器发出post, put或patch请求时，通常会同时给服务器发送json格式的相关数据，服务器在做数据处理之前，先做数据校验，是最合理和安全的前后端交互。如果客户端发送的数据不正确或不合理，服务器端经过校验后直接向客户端返回400错误及相应的数据错误信息即可。常见的数据校验包括：
- •数据类型校验，如字段类型如果是int，那么给字段赋字符串的值则报错
- •数据格式校验，如邮箱或密码，其赋值必须满足相应的正则表达式，才是正确的输入数据
- •数据逻辑校验，如数据包含出生日期和年龄两个字段，如果这两个字段的数据不一致，则数据校验失败
- 以上三种类型的校验，数据逻辑校验最为复杂，通常涉及到多个字段的配合，或者要结合用户和权限做相应的校验。Validation虽然是RESTful API 编写中的一个可选项，但它对API的安全、服务器的开销和交互的友好性而言，都具有重要意义，因此，Gevin建议，开发一套完善的RESTful API时，Validation的实现必不可少。

3. 如何设计RESTful API
3.5 Authentication 和 Permission

- Authentication指用户认证，Permission指权限机制，这两点是使RESTful API 强大、灵活和安全的基本保障。
- 常用的认证机制是Basic Auth和OAuth，RESTful API 开发中，除非API非常简单，且没有潜在的安全性问题，否则，认证机制是必须实现的，并应用到API中去。Basic Auth非常简单，很多框架都集成了Basic Auth的实现，自己写一个也能很快搞定，OAuth目前已经成为企业级服务的标配，其相关的开源实现方案非常丰富（更多）。

3. 如何设计RESTful API
3.6 CORS

- CORS即Cross-origin resource sharing，在RESTful API开发中，主要是为js服务的，解决javascript 调用 RESTful API时的跨域问题。
- 由于固有的安全机制，js的跨域请求时是无法被服务器成功响应的。现在前后端分离日益成为web开发主流方式的大趋势下，后台逐渐趋向指提供API服务，为各客户端提供数据及相关操作，而网站的开发全部交给前端搞定，网站和API服务很少部署在同一台服务器上并使用相同的端口，js的跨域请求时普遍存在的，开发RESTful API时，通常都要考虑到CORS功能的实现，以便js能正常使用API。
- 目前各主流web开发语言都有很多优秀的实现CORS的开源库，我们在开发RESTful API时，要注意CORS功能的实现，直接拿现有的轮子来用即可。
- *更多关于CORS的介绍，有兴趣的同学可以查看阮一峰老师的跨域资源共享 CORS详解*

3. 如何设计RESTful API
3.7 URL Rules

- RESTful API 是写给开发者来消费的，其命名和结构需要有意义。因此，在设计和编写URL时，要符合一些规范。

3. 如何设计RESTful API
3.7.1 Version your API

- 规范的API应该包含版本信息，在RESTful API中，最简单的包含版本的方法是将版本信息放到url中，如：
- /api/v1/posts/
- /api/v1/drafts/
- /api/v2/posts/
- /api/v2/drafts/
- 另一种优雅的做法是，使用HTTP header中的accept来传递版本信息，这也是GitHub API 采取的策略。

3. 如何设计RESTful API
3.7.2 Use nouns, not verbs

- RESTful API 中的url是指向资源的，而不是描述行为的，因此设计API时，应使用名词而非动词来描述语义，否则会引起混淆和语义不清。即：
- # Bad APIs
- /api/getArticle/1/
- /api/updateArticle/1/
- /api/deleteArticle/1/
- 上面四个url都是指向同一个资源的，虽然一个资源允许多个url指向它，但不同的url应该表达不同的语义，上面的API可以优化为：
- # Good APIs
- /api/Article/1/

- article 资源的获取、更新和删除分别通过 GET, PUT 和 DELETE方法请求API即可。试想，如果url以动词来描述，用PUT方法请求 /api/deleteArticle/1/ 会感觉多么不舒服。

3. 如何设计RESTful API

3.7.3 GET and HEAD should always be safe

- RFC2616已经明确指出，GET和HEAD方法必须始终是安全的。例如，有这样一个不规范的API:
- # The following api is used to delete articles
- # [GET]
- /api/deleteArticle?id=1
- 试想，如果搜索引擎访问了上面url会如何？
- 简单来说就是，GET和HEAD不能够授予除检索资源以外的功能。否则可能会产生不可预期的安全问题。

3. 如何设计RESTful API

3.7.4 Nested resources routing

- 如果要获取一个资源子集，采用 nested routing 是一个优雅的方式，如，列出所有文章中属于Gevin编写的文章：
- # List Gevin's articles
- /api/authors/gevin/articles/
- 获取资源子集的另一种方式是基于filter（见下面章节），这两种方式都符合规范，但语义不同：如果语义上将资源子集看作一个**独立的资源集合**，则使用 nested routing 感觉更恰当，如果资源子集的获取是出于**过滤**的目的，则使用filter更恰当。
- 至于编写RESTful API时到底应采用哪种方式，则仁者见仁，智者见智，语义上说的通即可。

3. 如何设计RESTful API
3.7.5 Filter

- 对于资源集合，可以通过url参数对资源进行过滤，如：
- # List Gevin's articles
- /api/articles?author=gevin
- 分页就是一种最典型的资源过滤。

3. 如何设计RESTful API

3.7.5 Pagination

- 对于资源集合，分页获取是一种比较合理的方式。如果基于开发框架（如Django REST Framework），直接使用开发框架中的分页机制即可，如果是自己实现分页机制，Gevin的策略是：
- 返回资源集合是，包含与分页有关的数据如下：
- {
- "page": 1, # 当前是第几页
- "pages": 3, # 总共多少页
- "per_page": 10, # 每页多少数据
- "has_next": true, # 是否有下一页数据
- "has_prev": false, # 是否有前一页数据
- "total": 27 # 总共多少数据
- }

3. 如何设计RESTful API
3.7.5 Pagination

- 当想API请求资源集合时，可选的分页参数为：
- 参数 含义
- page 当前是第几页，默认为1
- per_page 每页多少条记录，默认为系统默认值
- 另外，系统内还设置一个per_page_max字段，用于标记系统允许的每页最大记录数，当per_page值大于 per_page_max 值时，每页记录条数为 per_page_max。

3. 如何设计RESTful API
3.7.6 URL design tricks

- （1）Url是区分大小写的，这点经常被忽略，即：
- •/Posts
- •/posts
- 上面这两个url是不同的两个url，可以指向不同的资源
- （2）Back forward Slash (/)
- 目前比较流行的API设计方案，通常建议url以/作为结尾，如果API GET请求中，url不以/结尾，则重定向到以/结尾的API上去（这点现在的web框架基本都支持），因为有没有 /，也是两个url，即：
- •/posts/
- •/posts
- 这也是两个不同的url，可以对应不同的行为和资源

3. 如何设计RESTful API
3.7.6 URL design tricks

- （3）连接符 - 和 下划线 _
- RESTful API 应具备良好的可读性，当url中某一个片段（segment）由多个单词组成时，建议使用 - 来隔断单词，而不是使用 _，即：
- # Good
- /api/featured-post/

- # Bad
- /api/featured_post/

- 这主要是因为，浏览器中超链接显示的默认效果是，文字并附带下划线，如果API以_隔断单词，二者会重叠，影响可读性。

Thanks

- 参考文献：

[1].RESTful 概述http://www.ruanyifeng.com/blog/2011/09/restful

[2].RESTful 概述.http://blog.igevin.info/posts/restful-architecture-in-general/

[3].RESTful API 编写指南.http://blog.igevin.info/posts/restful-api-get-started-to-write/