做出一个好的API设计很难。API表达的是你的数据和你的数据使用者之间的契约。打破这个契约将会招致很多愤怒的邮件，和一大堆伤心的用户-因为他们手机上的App不工作了。而文档化只能达到一半的效果，并且也很难找到一个愿意写文档的程序员。

你所能做的最重要一件事来提高服务的价值就是创建一个API。因为随着其他服务的成长，有这样一个API会使你的服务或者核心应用将有机会变成一个平台。环顾一下现有的这些大公司：Facebook，Twitter，Google， Github，Amazon，Netflix等。如果当时他们没有通过API来开放数据的话，也不可能成长到如今的规模。事实上，整个行业存在的唯一目的就是消费所谓平台上的数据。

**你的API越容易使用，那么就会有越多的人去用它。**

本文提到的这些原则，如果你的API能严格按照这些原则来设计，使用者就可以知道它接下来要做什么，并且能减少大量不必要的疑惑或者是愤怒的邮件。我已经把所有内容都整理到不同的主题里了，你无需按顺序去阅读它。

定义

这里有一些非常重要的术语，我将在本文里面一直用到它们：

* **资源**：一个对象的单独实例，如一只动物
* **集合**：一群同种对象，如动物
* **HTTP**：跨网络的通信协议
* **客户端**：可以创建HTTP请求的客户端应用程序
* **第三方开发者**：这个开发者不属于你的项目但是有想使用你的数据
* **服务器**：一个HTTP服务器或者应用程序，客户端可以跨网络访问它
* **端点**：这个API在服务器上的URL用于表达一个资源或者一个集合
* **幂等**：无边际效应，多次操作得到相同的结果
* **URL段**：在URL里面已斜杠分隔的内容

数据设计与抽象

规划好你的API的外观要先于开发它实际的功能。首先你要知道数据该如何设计和核心服务/应用程序会如何工作。如果你纯粹新开发一个API，这样会比较容易一些。但如果你是往已有的项目中增加API，你可能需要提供更多的抽象。

有时候一个集合可以表达一个数据库表，而一个资源可以表达成里面的一行记录，但是这并不是常态。事实上，你的API应该尽可能通过抽象来分离数据与业务逻辑。这点非常重要，只有这样做你才不会打击到那些拥有复杂业务的第三方开发者，否则他们是不会使用你的API的。

当然你的服务可能很多部分是不应该通过API暴露出去的。比较常见的例子就是很多API是不允许第三方来创建用户的。

动词

显然你了解GET和POST请求。当你用浏览器去访问不同页面的时候，这两个是最常见的请求。POST术语如此流行以至于开始侵扰通俗用语。即使是那些不知道互联网如何工作的人们也能“post”一些东西到朋友的Facebook墙上。

这里至少有四个半非常重要的HTTP动词需要你知道。我之所以说“半个”的意思是PATCH这个动词非常类似于PUT，并且它们俩也常常被开发者绑定到同一个API上。

* GET (选择)：从服务器上获取一个具体的资源或者一个资源列表。
* POST （创建）： 在服务器上创建一个新的资源。
* PUT （更新）：以整体的方式更新服务器上的一个资源。
* PATCH （更新）：只更新服务器上一个资源的一个属性。
* DELETE （删除）：删除服务器上的一个资源。

还有两个不常用的HTTP动词：

* HEAD ： 获取一个资源的元数据，如数据的哈希值或最后的更新时间。
* OPTIONS：获取客户端能对资源做什么操作的信息。

一个好的RESTful API只允许第三方调用者使用这四个半HTTP动词进行数据交互，并且在URL段里面不出现任何其他的动词。

一般来说，GET请求可以被浏览器缓存（通常也是这样的）。例如，缓存请求头用于第二次用户的POST请求。HEAD请求是基于一个无响应体的GET请求，并且也可以被缓存的。

版本化

无论你正在构建什么，无论你在入手前做了多少计划，你核心的应用总会发生变化，数据关系也会变化，资源上的属性也会被增加或删除。只要你的项目还活着，并且有大量的用户在用，这种情况总是会发生。

请谨记一点，API是服务器与客户端之间的一个公共契约。如果你对服务器上的API做了一个更改，并且这些更改无法向后兼容，那么你就打破了这个契约，客户端又会要求你重新支持它。为了避免这样的事情，你既要确保应用程序逐步的演变，又要让客户端满意。那么你必须在引入新版本API的同时保持旧版本API仍然可用。

注：如果你只是简单的增加一个新的特性到API上，如资源上的一个新属性或者增加一个新的端点，你不需要增加API的版本。因为这些并不会造成向后兼容性的问题，你只需要修改文档即可。

随着时间的推移，你可能声明不再支持某些旧版本的API。申明不支持一个特性并不意味着关闭或者破坏它。而是告诉客户端旧版本的API将在某个特定的时间被删除，并且建议他们使用新版本的API。

一个好的RESTful API会在URL中包含版本信息。另一种比较常见的方案是在请求头里面保持版本信息。但是跟很多不同的第三方开发者一起工作后，我可以很明确的告诉你，在请求头里面包含版本信息远没有放在URL里面来的容易。

**分析**

所谓API分析就是持续跟踪那些正为人使用的API的版本和端点信息。而这可能就跟每次请求都往数据库增加一个整数那样简单。有很多的原因显示API跟踪分析是一个好主意，例如，对那些使用最广泛的API来说效率是最重要的。

第三方开发者通常会关注API的构建目的，其中最重要的一个目的是你决定什么时候不再支持某个版本。你需要明确的告知开发者他们正在使用那些即将被移除的API特性。这是一个很好的方式在你准备删除旧的API之前去提醒他们进行升级。

当然第三方开发者的通知流程可以以某种条件被自动触发，例如每当一个过时的特性上发生10000次请求时就发邮件通知开发者。

API根URL

无论你信不信，API的根地址很重要。当一个开发者接手了一个旧项目（如进行代码考古时）。而这个项目正在使用你的API，同时开发者还想构建一个新的特性，但他们完全不知道你的服务。幸运的是他们知道客户端对外调用的那些URL列表。让你的API根入口点保持尽可能的简单是很重要的，因为开发者很可能一看到那些冗长而又复杂的URL就转身而走。

这里有两个常见的URL根例子：

* https://example.org/api/v1/\*
* https://api.example.com/v1/\*

如果你的应用很庞大或者你预期它将会变的很庞大，那么将API放到子域下通常是一个好选择。这种做法可以保持某些规模化上的灵活性。

但如果你觉得你的API不会变的很庞大，或是你只是想让应用安装更简单些（如你想用相同的框架来支持站点和API），将你的API放到根域名下也是可以的。

让API根拥有一些内容通常也是个好主意。Github的API根就是一个典型的例子。从个人角度来说我是一个通过根URL发布信息的粉丝，这对很多人来说是有用的，例如如何获取API相关的开发文档。

同样也请注意HTTPS前缀，一个好的RESTful API总是基于HTTPS来发布的。

端点

一个端点就是指向特定资源或资源集合的URL。

如果你正在构建一个虚构的API来展现几个不同的动物园，每一个动物园又包含很多动物，员工和每个动物的物种，你可能会有如下的端点信息：

* https://api.example.com/v1/**zoos**
* https://api.example.com/v1/**animals**
* https://api.example.com/v1/**animal\_types**
* https://api.example.com/v1/**employees**

针对每一个端点来说，你可能想列出所有可行的HTTP动词和端点的组合。如下所示，请注意我把HTTP动词都放在了虚构的API之前，正如将同样的注解放在每一个HTTP请求头里一样。（下面的URL就不翻译了，我觉得没啥必要翻）

* GET /zoos: List all Zoos (ID and Name, not too much detail)
* POST /zoos: Create a new Zoo
* GET /zoos/ZID: Retrieve an entire Zoo object
* PUT /zoos/ZID: Update a Zoo (entire object)
* PATCH /zoos/ZID: Update a Zoo (partial object)
* DELETE /zoos/ZID: Delete a Zoo
* GET /zoos/ZID/animals: Retrieve a listing of Animals (ID and Name).
* GET /animals: List all Animals (ID and Name).
* POST /animals: Create a new Animal
* GET /animals/AID: Retrieve an Animal object
* PUT /animals/AID: Update an Animal (entire object)
* PATCH /animals/AID: Update an Animal (partial object)
* GET /animal\_types: Retrieve a listing (ID and Name) of all Animal Types
* GET /animal\_types/ATID: Retrieve an entire Animal Type object
* GET /employees: Retrieve an entire list of Employees
* GET /employees/EID: Retreive a specific Employee
* GET /zoos/ZID/employees: Retrieve a listing of Employees (ID and Name) who work at this Zoo
* POST /employees: Create a new Employee
* POST /zoos/ZID/employees: Hire an Employee at a specific Zoo
* DELETE /zoos/ZID/employees/EID: Fire an Employee from a specific Zoo

在上面的列表里，ZID表示动物园的ID， AID表示动物的ID，EID表示雇员的ID，还有ATID表示物种的ID。让文档里所有的东西都有一个关键字是一个好主意。

为了简洁起见，我已经省略了所有API共有的URL前缀。作为沟通方式这没什么问题，但是如果你真要写到API文档中，那就必须包含完整的路径（如，GET http://api.example.com/v1/animal\_type/ATID）。

请注意如何展示数据之间的关系，特别是雇员与动物园之间的多对多关系。通过添加一个额外的URL段就可以实现更多的交互能力。当然没有一个HTTP动词能表示正在解雇一个人，但是你可以使用DELETE一个动物园里的雇员来达到相同的效果。

过滤器

当客户端创建了一个请求来获取一个对象列表时，很重要一点就是你要返回给他们一个符合查询条件的所有对象的列表。这个列表可能会很大。但你不能随意给返回数据的数量做限制。因为这些无谓的限制会导致第三方开发者不知道发生了什么。如果他们请求一个确切的集合并且要遍历结果，然而他们发现只拿到了100条数据。接下来他们就不得不去查找这个限制条件的出处。到底是ORM的bug导致的，还是因为网络截断了大数据包？

**尽可能减少那些会影响到第三方开发者的无谓限制。**

这点很重要，但你可以让客户端自己对结果做一些具体的过滤或限制。这么做最重要的一个原因是可以最小化网络传输，并让客户端尽可能快的得到查询结果。其次是客户端可能比较懒，如果这时服务器能对结果做一些过滤或分页，对大家都是好事。另外一个不那么重要的原因是（从客户端角度来说），对服务器来说响应请求的负载越少越好。

过滤器是最有效的方式去处理那些获取资源集合的请求。所以只要出现GET的请求，就应该通过URL来过滤信息。以下有一些过滤器的例子，可能是你想要填加到API中的：

* ?limit=10: 减少返回给客户端的结果数量（用于分页）
* ?offset=10: 发送一堆信息给客户端（用于分页）
* ?animaltypeid=1: 使用条件匹配来过滤记录
* ?sortby=name&order=asc:  对结果按特定属性进行排序

有些过滤器可能会与端点URL的效果重复。例如我之前提到的GET /zoo/ZID/animals。它也同样可以通过GET /animals?zoo\_id=ZID来实现。独立的端点会让客户端更好过一些，因为他们的需求往往超出你的预期。本文中提到这种冗余差异可能对第三方开发者并不可见。

无论怎么说，当你准备过滤或排序数据时，你必须明确的将那些客户端可以过滤或排序的列放到白名单中，因为我们不想将任何的数据库错误发送给客户端。

状态码

对于一个RESTful API来说很重要的一点就是要使用HTTP的状态码，因为它们是HTTP的标准。很多的网络设备都可以识别这些状态码，例如负载均衡器可能会通过配置来避免发送请求到一台web服务器，如果这台服务器已经发送了很多的50x错误回来。这里有大量的HTTP状态码可以选择，但是下面的列表只给出了一些重要的代码作为一个参考：

* 200  OK – [GET] 客户端向服务器请求数据，服务器成功找到它们
* 201  CREATED – [POST/PUT/PATCH]   客户端向服务器提供数据，服务器根据要求创建了一个资源
* 204  NO CONTENT – [DELETE]   客户端要求服务器删除一个资源，服务器删除成功
* 400  INVALID REQUEST – [POST/PUT/PATCH]   客户端向服务器提供了不正确的数据，服务器什么也没做
* 404  NOT FOUND – [\*]   客户端引用了一个不存在的资源或集合，服务器什么也没做
* 500  INTERNAL SERVER ERROR – [\*]   服务器发生内部错误，客户端无法得知结果，即便请求已经处理成功

状态码范围

1xx范围的状态码是保留给底层HTTP功能使用的，并且估计在你的职业生涯里面也用不着手动发送这样一个状态码出来。

2xx范围的状态码是保留给成功消息使用的，你尽可能的确保服务器总发送这些状态码给用户。

3xx范围的状态码是保留给重定向用的。大多数的API不会太常使用这类状态码，但是在新的超媒体样式的API中会使用更多一些。

4xx范围的状态码是保留给客户端错误用的。例如，客户端提供了一些错误的数据或请求了不存在的内容。这些请求应该是幂等的，不会改变任何服务器的状态。

5xx范围的状态码是保留给服务器端错误用的。这些错误常常是从底层的函数抛出来的，并且开发人员也通常没法处理。发送这类状态码的目的是确保客户端能得到一些响应。收到5xx响应后，客户端没办法知道服务器端的状态，所以这类状态码是要尽可能的避免。

**预期的返回文档**

当使用不同的HTTP动词向服务器请求时，客户端需要在返回结果里面拿到一系列的信息。下面的列表是非常经典的RESTful API定义：

* GET /collection: 返回一系列资源对象
* GET /collection/resource: 返回单独的资源对象
* POST /collection: 返回新创建的资源对象
* PUT /collection/resource: 返回完整的资源对象
* PATCH /collection/resource: 返回完整的资源对象
* DELETE /collection/resource: 返回一个空文档

请注意当一个客户端创建一个资源时，她们常常不知道新建资源的ID（也许还有其他的属性，如创建和修改的时间戳等）。这些属性将在随后的请求中返回，并且作为刚才POST请求的一个响应结果。

认证

服务器在大多数情况下是想确切的知道谁创建了什么请求。当然，有些API是提供给公共用户（匿名用户）的，但是大部分时间里也是代表某人的利益。

OAuth2.0提供了一个非常好的方法去做这件事。在每一个请求里，你可以明确知道哪个客户端创建了请求，哪个用户提交了请求，并且提供了一种标准的访问过期机制或允许用户从客户端注销，所有这些都不需要第三方的客户端知道用户的登陆认证信息。

还有OAuth1.0和xAuth同样适用这样的场景。无论你选择哪个方法，请确保它为多种不同语言/平台上的库提供了一些通用的并且设计良好文档，因为你的用户可能会使用这些语言和平台来编写客户端。

内容类型

目前，大多数“精彩”的API都为RESTful接口提供JSON数据。诸如Facebook，Twitter，Github等等你所知的。XML曾经也火过一把（通常在一个大企业级环境下）。这要感谢SOAP，不过它已经挂了，并且我们也没看到太多的API把HTML作为结果返回给客户端（除非你在构建一个爬虫程序）。

只要你返回给他们有效的数据格式，开发者就可以使用流行的语言和框架进行解析。如果你正在构建一个通用的响应对象，通过使用一个不同的序列化器，你也可以很容易的提供之前所提到的那些数据格式（不包括SOAP）。而你所要做的就是把使用方式放在响应数据的接收头里面。

有些API的创建者会推荐把.json, .xml, .html等文件的扩展名放在URL里面来指示返回内容类型，但我个人并不习惯这么做。我依然喜欢通过接收头来指示返回内容类型（这也是HTTP标准的一部分），并且我觉得这么做也比较适当一些。

超媒体API

超媒体API很可能就是RESTful API设计的将来。超媒体是一个非常棒的概念，它回归到了HTTP和HTML如何运作的“本质”。

在非超媒体RESTful API的情景中，URL端点是服务器与客户端契约的一部分。这些端点必须让客户端事先知道，并且修改它们也意味着客户端可能再也无法与服务器通信了。你可以先假定这是一个限制。

时至今日，英特网上的API客户端已经不仅仅只有那些创建HTTP请求的用户代理了。大多数HTTP请求是由人们通过浏览器产生的。人们不会被哪些预先定义好的RESTful API端点URL所束缚。是什么让人们变的如此与众不同？因为人们可以阅读内容，可以点击他们感兴趣的链接，并浏览一下网站，然后跳到他们关注的内容那里。即使一个URL改变了，人们也不会受到影响（除非他们事先给某个页面做了书签，这时他们回到主页并发现原来有一条新的路径可以去往之前的页面）。

超媒体API概念的运作跟人们的行为类似。通过请求API的根来获得一个URL的列表，这个列表里面的每一个URL都指向一个集合，并且提供了客户端可以理解的信息来描述每一个集合。是否为每一个资源提供ID并不重要（或者不是必须的），只要提供URL即可。

一个超媒体API一旦具有了客户端，那么它就可以爬行链接并收集信息，而URL总是在响应中被更新，并且不需要如契约的一部分那样事先被知晓。如果一个URL曾经被缓存过，并且在随后的请求中返回404错误，那么客户端可以很简单的回退到根URL并重新发现内容。

在获取集合中的一个资源列表时会返回一个属性，这个属性包含了各个资源的完整URL。当实施一个POST/PATCH/PUT请求后，响应可以被一个3xx的状态码重定向到完整的资源上。

JSON不仅告诉了我们需要定义哪些属性作为URL，也告诉了我们如何将URL与当前文档关联的语义。正如你猜的那样，HTML就提供了这样的信息。我们可能很乐意看到我们的API走完了完整的周期，并回到了处理HTML上来。想一下我们与CSS一起前行了多远，有一天我们可能再次看到它变成了一个通用实践让API和网站可以去使用相同的URL和内容。

文档

老实说，即使你不能百分之百的遵循指南中的条款，你的API也不是那么糟糕。但是，如果你不为API准备文档的话，没有人会知道怎么使用它，那它真的会成为一个糟糕的API。

让你的文档对那些未经认证的开发者也可用

不要使用文档自动化生成器，即便你用了，你也要保证自己审阅过并让它具有更好的版式。

不要截断示例中请求与响应的内容，要展示完整的东西。并在文档中使用高亮语法。

文档化每一个端点所预期的响应代码和可能的错误消息，和在什么情况下会产生这些的错误消息

如果你有富余的时间，那就创建一个控制台来让开发者可以立即体验一下API的功能。创建一个控制台并没有想象中那么难，并且开发者们（内部或者第三方）也会因此而拥戴你。

另外确保你的文档能够被打印。CSS是个强大的工具可以帮助到你。而且在打印的时候也不用太担心边侧栏的问题。即便没有人会打印到纸上，你也会惊奇的发现很多开发者愿意转化成PDF格式进行离线阅读。

勘误：原始的HTTP封包

因为我们所做的都是基于HTTP协议，所以我将展示给你一个解析了的HTTP封包。我经常很惊讶的发现有多少人不知道这些东西。当客户端发送一个请求道服务器时，他们会提供一个键值对集，先是一个头，紧跟着是两个回车换行符，然后才是请求体。所有这些都是在一个封包里被发送。

服务器响应也是同样的键值对集，带两个回车换行符，然后是响应体。HTTP就是一个请求/响应协议；它不支持“推送”模式（服务器直接发送数据给客户端），除非你采用其他协议，如Websockets。

当你设计API时，你应该能够使用工具去查看原始的HTTP封包。Wireshark是个不错的选择。同时，你也该采用一个框架/web服务器，使你能够在必要时修改某些字段的值。

**Example HTTP Request**

1. POST /v1/animal HTTP/1.1
2. Host: api.example.org
3. Accept: application/json
4. Content-Type: application/json
5. Content-Length: 24
6. {
7. "name": "Gir",
8. "animal\_type": 12
9. }

**Example HTTP Response**

1. HTTP/1.1 200 OK
2. Date: Wed, 18 Dec 2013 06:08:22 GMT
3. Content-Type: application/json
4. Access-Control-Max-Age: 1728000
5. Cache-Control: no-cache
6. {
7. "id": 12,
8. "created": 1386363036,
9. "modified": 1386363036,
10. "name": "Gir",
11. "animal\_type": 12
12. }

wenz出处：https://www.cnblogs.com/moonz-wu/p/4211626.html