**如何版本化你的 API？**

LinuxEden09-12 05:50

如何版本化 API 需要考虑各种实际业务场景，但是一个完备的 API 应该是：

* 和客户端交互的约定。API 需要确保稳定性，预先定义各种可能返回状态，包括各种异常。客户端无需考虑约定之外的情况。
* 向下兼容。在 API 没有变化的时候，API 实现的更新和升级，都应该确保原有客户端请求不出现问题。
* RESTful。API 设计应该能够遵照 RESTful 风格，通过 URI 来表示资源，通过 HTTP GET、POST、PUT、DELETE 等方法表示操作行为。

为了满足上述约定，版本化 API 不失为一种保持兼容性的好方法。版本化 API 的通常方式有：

**URI 中设置版本**

这种方式通常在 URI 中增加一段用于标识版本，例如 /v1 、 /v2 等。例如：

curl https://example.com/api/v2/lists/3

这种方式的优势在于版本信息很容易明显的看出来，可以通过浏览器直接访问。

**HTTP 头中设置版本**

这种方式的版本信息会放在 HTTP 的请求头中，通常会利用 Accept 字段，或者自定义一个字段。例如：

curl https://example.com/api/lists/3 \

-H 'Accept: application/vnd.example.v2+json'

这种方式的好处是当版本升级时，URI 保持不变，并且仅用于表示资源定位。

**没有版本**

版本化的目的是为了标识 API 的变化，如果 API 不会变化，或者每次都会重新扩展新的 API，这种情况下，就可以标识版本信息。例如：

curl https://example.com/api/lists/3

**一种折中方案**

前面提到了三种版本化 API 的方式，通常情况下需要针对自己业务的特殊性来挑选其中的一种方式。但是，在实际应用场景中，情况会更加复杂，API 的升级通常有两种情况：

1. 大版本更新，例如字段类型变更、数据对象变更等。这种情况下无法满足对客户端的向下兼容，因此需要修改版本号。
2. 小版本更新，例如增加可选参数、增加返回字段等。这种情况对于新客户端可以增加功能，对于老客户端仍然可以保持原有功能，可以不修改版本号。

因此，本文提出的折中方案是基于 URI 中的大版本号和 HTTP 头中的小版本号整合的方式。下面通过一个简单的示例来解释。

**用户管理平台**

一个常用的用户管理平台，提供以下 API，通过用户 ID 获取用户信息：

curl https://example.com/api/v1/user/1

...

{

"id": 1,

"name": "test",

"email": "test@example.com"

}

考虑以下两种变动情况：一种是用户 id 从数字变成了字符串，另一种是新增一个用户头像的值。

前者修改因为数据类型的变化，会导致客户端解析出现问题。因此这样的修改已经破坏了向下兼容性，此时就需要修改 API 的版本号。例如：

curl https://example.com/api/v2/user/1

...

{

"id": "1",

"name": "test",

"email": "test@example.com"

}

第二种情况，对于旧客户端来说，只是增加了不使用的字段，通常的 JSON 格式解析库都可以忽略这些不使用的字段。对于新客户端则可以读取新的字段。例如：

curl https://example.com/api/v2/user/1

...

{

"id": "1",

"name": "test",

"email": "test@example.com",

"avatar": "http://example.com/1.jpg"

}

这种情况下，基本可以做到向下兼容，因此可以算是“小版本升级”。针对小版本升级，可以将小版本号放到 HTTP 头中。例如：

curl https://example.com/api/v2/user/1 \

-H 'API-VERSION: 20170801'

...

{

"id": "1",

"name": "test",

"email": "test@example.com",

"avatar": "http://example.com/1.jpg"

}

**后端路由**

由于混合版本化的方式同时涉及到 URI 和 HTTP 头字段，前端代理（例如 HAProxy、nginx）可以通过这些特定版本号字段将请求代理到对应的后端应用。

例如，前端使用 HAProxy 进行多版本分发，可以针对 URI 和 HTTP 头定制 acl，然后再对这些 acl 进行组合，设置不同的 backend。

acl is\_v1 path\_beg /api/v1

acl is\_v2 path\_beg /api/v2

acl is\_version\_1 hdr(API-VERSION) 20170801

acl is\_version\_2 hdr(API-VERSION) 20170701

use\_backend old\_server if is\_v1 is\_version\_1

use\_backend new\_server if is\_v2 is\_version\_2

backend old\_server

...

backend new\_server

...

这样可以将 API 版本化规则应用到不同的后端，以保证向下兼容性。

**总结**

基于本文版本化 API 规则，将“大版本”应用在 URI 上，将“小版本”应用在 HTTP 头字段上。通常来说，如果 API 升级之后破坏了向下兼容性，就应该升级“大版本”号；如果 API 升级可以向下兼容，可以升级“小版本”号。

版本化 API 有很多不同的设计方式，本文仅是其中一种。实际应用时，还是要根据业务场景进行选择，包括 API 版本升级频率，API 稳定性等。通过 HAProxy、nginx 等代理服务，可以在确保向下兼容的情况下，由业务方决定老版本 API 的保留时间。