REST 简介

REST 是一个术语的缩写，representational state transfer，中文直译「表征状态转移」。REST 是一套风格约定，RESTful 是它的形容词形式；比如一套实现了 REST 风格的接口，可以称之为 RESTful 接口。

REST 对请求的约定

REST 用来规范应用如何在 HTTP 层与 API 提供方进行数据交互，是一种交互规约定；客户端通过向服务器端发送 HTTP（s）请求，接收服务器的响应，完成一次 HTTP 交互。这个交互过程中，REST 架构约定两个重要方面就是 HTTP 请求的所采用方法，以及请求的链接。在请求层面，REST 规范可以简单粗暴抽象成以下两个规则：

- **请求 API 的 URL 表示用来定位资源；**
- **请求的 METHOD 表示对这个资源进行的操作；**

以下将以这两个规则为基础，描述如何构造一个符合 REST 规范的请求。

（1）API 的 URL

URL 用来定位资源，跟要进行的操作区分开，这就意味这 URL 不该有任何动词；

```
//下面示例中的 get、create、search 等动词，都不应该出现在 REST 架构的后端接口路径中。在以前，这些接口中的动名词通常对应后台的某个函数。比如：
/api/getUser
/api/createApp
/api/searchResult
/api/deleteAllUsers
```

```
//当我们需要对单个用户进行操作时，根据操作的方式不同可能需要下面的这些接口：
/api/getUser （用来获取某个用户的信息，还需要以参数方式传入用户 id 信息）
/api/updateUser （用来更新用户信息）
/api/deleteUser （用来删除单个用户）
/api/resetUser （重置用户的信息）
```

```
//更有甚者，可能在更新用户不同信息时，提供不同的接口，比如：
/api/updateUserName
/api/updateUserEmail
/api/updateUser
```

这样的弊端在于：首先加上了动词，肯定是使 URL 更长了；其次对一个资源实体进行不同的操作就是一个不同的 URL，造成 URL 过多难以管理。其实当你回过头看「URL」这个术语的定义时，更能理解这一点。URL 的意思是统一资源定位符，这个术语已经清晰的表明，一个 URL 应该用来定位资源，而不应该掺入对操作行为的描述。在 REST 架构的链接应该是这个样子：

- URL 中不应该出现任何表示操作的动词，链接只用于对应资源；
- URL 中应该单复数区分，推荐的实践是永远只用复数；比如 GET /api/users 表示获取用户的列表；如果获取单个资源，传入 ID，比如 /api/users/123 表示获取单个用户的信息；
- 按照资源的逻辑层级，对 URL 进行嵌套，比如一个用户属于某个团队，而这个团队也是众多团队之一；那么获取这个用户的接口可能是这样：

```
GET /api/teams/123/members/234 表示获取 id 为 123 的小组下，id为234 的成员信息
```

```
//按照类似的规则，可以写出如下的接口
/api/teams （对应团队列表）
/api/teams/123 （对应 ID 为 123 的团队）
/api/teams/123/members （对应 ID 为 123 的团队下的成员列表）
/api/teams/123/members/456 （对应 ID 为 123 的团队下 ID 未 456 的成员）
```

（2）API 请求的方法

在很多系统中，几乎只用 GET 和 POST 方法来完成了所有的接口操作；这个行为类似于全用 DIV 来布局。实际上，我们不只有GET 和 POST 可用，在 REST 架构中，有以下几个重要的请求方法：GET，POST，PUT，PATCH，DELETE。这几个方法都可以与对数据的 CRUD 操作对应起来。

【Read】资源的读取，用 GET 请求

```
//GET 应当实现为一个安全方法。用于获取数据而不应该产生副作用。
GET /api/users  （ 表示读取用户列表）
```

【Created】资源的创建，用 POST 方法

POST 是一个非幂等的方法，多次调用会造成不同效果；

> 如果对服务器资源的多次请求与一次请求造成的副作用是一样的的话，那这个请求方法可以被认为是幂等。

比如下面的请求会在服务器上创建一个 name 属性为 'John Snow' 的用户；多次请求就会创建多个这样的用户。

```
POST /api/users
{
  "name": "John Snow"
}
```

【Update】资源的更新。用于更新的 HTTP 方法有两个，PUT 和 PATCH。

他们都应当被实现为幂等方法，即多次同样的更新请求应当对服务器产生同样的副作用。PUT 和 PATCH 有各自不同的使用场景；PUT 用于更新资源的全部信息，在请求的 body 中需要传入修改后的全部资源主体；而 PATCH 用于局部更新，在 body 中只需要传入需要改动的资源字段。

```
//设想服务器中有以下用户资源 /api/users/123
{
 "id": 123,
 "name": "Original",
 "age": 20
}
```

```
//当我们往后台发送更新请求时，PATCH 和 PUT 造成的效果是不一样。
PUT /api/users/123
{
 "name": "PUT Update"
}
```

```
//上述 PUT 请求操作后的内容是：
{
 "id": 123,
 "name": "PUT Update"
}
//可以观察到，资源原有的 age 字段被清除掉了。
```

```
//而如果改用 PATCH 的话
PATCH /api/users/123
{
 "name": "PATCH Update"
}
```

```
//更新后的内容是：
{
 "id": 123,
 "name": "PATCH Update",
 "age": 20
}
//请求中指定的 name 属性被更新了，而原有的 age 属性则保持不变。
```

PATCH 的作用在于如果一个资源有很多字段，在进行局部更新时，只需要传入需要修改的字段即可。否则在用 PUT 的情况下，你不得不将整个资源模型全都发送回服务器，造成网络资源的极大浪费。

【Delete】，资源的删除，相应的请求 HTTP 方法就是 DELETE。这个也应当被实现为一个幂等的方法。如:

```
DELETE /api/users/123
//用于删除服务器上 ID 为 123 的资源，多次请求产生副作用都是，是服务器上 ID 为 123 的资源不存在。
```

（3）分页、过滤

REST 风格的接口地址，表示的可能是单个资源，也可能是资源的集合；当我们需要访问资源集合时，设计良好的接口应当接受参数，允许只返回满足某些特定条件的资源列表。

```
//比如支持以 offset 和 limit 参数来进行分页；
GET /api/users?offset=0&limit=20
```

```
//支持提供关键词进行搜索，以及排序
GET /api/users?keyword=john&sort=age
```

```
//支持根据字段进行过滤
GET /api/users?gender=male
```

设计合适的 API URL，以及选择合适的请求方法，可以语义化的描述一个 HTTP 请求的操作。

当我们都熟悉且遵循这样的规范后，基本可以看到一个 REST 风格的接口就知道如何使用这个接口进行 CRUD 操作了。比如下面这面这个接口就表示搜索 ID 为 123 的图书馆的书，并且书的信息里包含关键字「game」，返回前十条满足条件的结果。

```
GET /api/libraries/123/books?keyword=game&sort=price&limit=10&offset=0
```

```
//同样，下面这个请求的意思也就很明显了吧。
PATCH /api/companies/123/employees/234
{
    "salary": 2300
}
```