第3章

资源和表述

REST 世界中的一切都被认为是资源，每个资源都由 URI 标识，使用统一的接口。处理资源时需要使用 POST、GET、PUT、DELETE 等 HTTP 方法，每个 HTTP 请求都是独立的，穿梭在资源请求者与提供者之间的则是资源的表述。

3.1 资源的本质

资源是一个很宽泛的概念，任何寄宿于 Web、可供操作的“事物”均可被视为资源，就其本质而言，任何足够重要并被引用的事物都可以是资源。一个苹果可以是一个资源，但显然人们不可能通过网络传输物质的苹果，像《星际迷航》中那种“远距传送”(teleportation)现阶段还只是科学幻想。但如果将这个苹果放到电商网站中售卖，通过一系列的操作，用户可以在几天之内收到快递送上门的实物。

信息系统中的资源一般是可以被保存到计算机里的虚拟资源，如电子文档，数据库的记录，或者算法的运行结果，这些被统称为“信息资源(information resource)”，因为它们的本质都是数据，故得以在网络中传输。

信息资源可以被看作是物理资源的一种抽象，可以被体现为经过持久化处理后保存到磁盘上的某个文件或者数据库中的某条记录，也可以是 Web 应用接受请求后采用某种算法计算得出的结果。但在某种意义上，这种资源抽象又具有物理含义，如有了电子支付以后，买东西时不再需要付纸质的钞票，通过一个二维码或者银行的手机客户端进行转账，“钱”这个物质就可以发生“转移”，在银行账户里的数字也就变了。

所以，人们完全可以用理解物理资源的方式理解将要在软件系统中通过服务操作的“资源”。面向资源的架构实际更接近面向对象的思想，其同样是把系统分解为一个个功能部件，即对象(object)。每个对象都有自己的类和方法(用于与其他对象交互)。这些对象对应的就是面向资源架构中的那些资源，只不过在 Web 世界里，这些资源必须通过 URI 标识。

资源是物理资源的抽象，它可以具有多种表现形式，这种资源的呈现形式被称作资源的表述。例如，一篇文章可以使用不含任何格式的 txt 文本形式表现，也可以使用 HTML、XML、JSON 等具有丰富格式的形式表现。HTTP 头部中的 Content-Type 字段描述的就是资源表述的格式。

例如，在电商平台的网站里，描绘一种苹果商品的资源数据如下所示。

```
…
skuid: 10020316405615,
name: '新鲜红富士苹果水果 3 千克装新鲜水果 3 千克',
skuidkey:'44793A379A42310B54A227146F902BD78493D9B4DC91B9A0',
href: '//item.jd.com/10020316405615.html',
src:'jfs/t1/124184/25/11457/153968/5f4cb164Eb3ae0e51/3d644848828b1c1f.jpg',
…
```

其中，src 代表的相对地址加上“https://img14.360buyimg.com/n0/”前缀就可以组成一个 URL，表示的是一幅显示给用户的苹果图像（图 3.1），这也可以被看作是资源的一种表述。

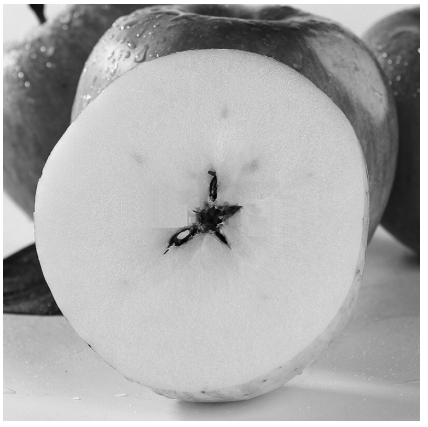

图 3.1　苹果的图像表述（京东平台）

这条数据库资源是一个 HTTP 资源，事实上也是一个信息资源，因为用户可以通过互联网将它逐个字符地发送出去。任何寄宿在 Web 服务器上，可以利用 HTTP 获取或者操作的“事物”均可以被称为资源。

资源请求者和资源拥有者只有在对事物的命名上达成一致以后才能针对这个事物实现相互通信。因此，每个资源必须拥有自己的唯一标识，互联网中使用 URL 和 URN 唯一标记一个资源，二者被统称为 URI。因为还需要定位，所以 URI 还应该具有“可寻址性（addressability）”，所以人们通常采用一个 URL 作为资源的标识，举例如下。

（1）http://www.abc.com/employees/c001（编号 c001 的员工）。

（2）http://www.abc.com/sales/2021/12/31（2021 年 12 月 31 日的销售额）。

（3）http://www.abc.com/orders/2021/q4（2021 年第 4 季度签订的订单）。

URL 为每个资源提供一个全球唯一的地址，将一个事物赋以 URL，它就会成为一个资源，如下面这条也是一个可用的资源。

```
http://t.weather.itboy.net/api/weather/city/101020100
```

这个 URL 代表的资源是上海市的实时天气，作为 URL，它的结构非常清晰：起始是提供资源的宿主网站 t.weather.itboy.net；然后是代表天气资源的 api/weather/；之后是表示单个城市资源的 city/，最后是城市代码 101020100。

向这个 API 发送一个 Get 请求，会得到如下 JSON 格式的反馈。

```
{
"message":"success……",
"status":200,
"date":"20210313",
"time":"2021-03-13 17:45:28",
"cityInfo":
{"city":"上海市",
"citykey":"101020100",
"parent":"上海",
"updateTime":"16:01"},
```

```
"data":
{"shidu":"56%","
pm25":55.0,"
pm10":66.0,"
quality":"良","
wendu":"14","
ganmao":"极少数敏感人群应减少户外活动",
"forecast":
[
{"date":"13",
"high":"高温 16℃",
"low":"低温 9℃",
"ymd":"2021-03-13",
"week":"星期六",
"sunrise":"06:07",
"sunset":"18:01",
"aqi":70,
"fx":"东北风",
"fl":"2 级",
"type":"阴",
"notice":"不要被阴云遮挡住好心情"
},
……//两周天气预报,此处省略后面 13 天

],
"yesterday":
{"date":"12",
"high":"高温 12℃",
"low":"低温 7℃",
"ymd":"2021-03-12",
"week":"星期五",
"sunrise":"06:08",
"sunset":"18:00",
"aqi":59,
"fx":"西北风",
"fl":"3 级",
"type":"阴",
"notice":"不要被阴云遮挡住好心情"}
}
}
```


不需赘述了,其中丰富的信息就是请求者希望得到的内容。

3.2 表述的本质

资源请求者实际上并不关心资源是什么,因为资源请求者从来看不到资源,资源请求者看到的永远只是资源的 URL 和表述。客户端应用与服务端的交互是通过资源的表述间接完成的,这体现了非常好的设计原则:“松耦合”与前后端分离。

在软件领域,“耦合”一般指软件组件之间的依赖程度,在一个“松耦合”的系统中,客户

端和远程服务并不知道也不需要知道对方是如何实现的，这样它们各自的实现就可以根据需要自行更改，而不必担心这种修改会破坏对方已有的实现。前后端分离已成为互联网项目开发的标准模式，前端展现所用到的数据都是由后端通过同步或异步接口的方式提供，前端只负责展现，后端则只负责处理逻辑与数据的存储。

通过资源的表述间接完成交互，实际上就是隔离了客户端与服务端（前端与后端），使请求服务方的操作不会直接影响服务提供者，而服务提供者也可以安全地分享自己的资源。

很多资源数据是变化的，如某地的气温数据，所以资源的表述实际是一段对资源（在某个特定时刻的）状态的描述，而客户端请求资源往往也是想得到资源的当前状态。另外，服务提供者也不必提供原始的或者完整的资源，只需要根据情况将资源（局部的或者完全的）用合适的格式以及结构表达出来，这就是“表述”。

因此，在客户端-服务器端之间转移的并不是资源本身，而是资源的表述。缩写词 REST 中的 state transfer 被翻译为“状态转移”，在客户端-服务器端之间转移的资源表述，就是对资源当前状态的某种合适的表达。

对资源的表述可以有多种形式，如 JSON/XML/HTML/纯文本等。服务器发送给客户端的资源，可以通过定义在 HTTP 中的标准的内容协商（content negotiation）机制来确定具体的格式。表 3.1 是罗伊·托马斯·菲尔丁在他的博士论文中对 REST 数据元素的总结。

表 3.1　REST 数据元素的总结

数 据 元 素	现代 Web 实例
资源	一个超文本引用的预期概念目标
资源标识符	URL、URN
表述	HTML 文档、JPEG 图片等
表述元数据	媒体类型、最后修改时间
资源元数据	源链接、替代物、变化
控制数据	if-modified-since、cache-control

表述的作用可以被归纳如下。

1. 表述可以描述资源状态

表述只负责提供数据，如前文提到的天气数据资源，如果 GET 这个地址的资源（http://t.weather.itboy.net/api/weather/city/101120101），会得如下信息。

```
{"message":"success……","status":200,"date":"20210316","time":"2021-03-16 22:18:59", "cityInfo":{"city":"济南市", "citykey":"101120101", "parent":"山东", "updateTime":"21:16"}, "data":{"shidu":"39%", "pm25":78.0, "pm10":448.0, "quality":"严重", "wendu":"7", "ganmao":"老年人病人应留在室内，停止体力消耗，一般人群避免户外活动", "forecast":[{"date":"16", "high":"高温 15℃", "low":"低温 5℃", "ymd":"2021-03-16", "week":"星期二", "sunrise":"06:21", "sunset":"18:19", "aqi":181, "fx":"东北风", "fl":"2 级", "type":"霾", "notice":"雾霾来袭，戴好口罩再出门"},……
```

这是一段 JSON 格式的表述，内容是当前时刻的某地天气数据。这段文字可能看起来不够直观，但经过浏览器的处理，可以得到类似如图 3.2 所示的形式。

图 3.2 表述内容的一种可视化呈现

一个资源可以有很多种表述，如政府的官方文档经常会有多个语言版本。有的资源既有整体概括性的表述，也有面面俱到的、细致化的表述。有一些 API 可以使用 JSON 和 XML 数据格式来表示同一数据，当这种情况发生时，客户端应该如何指定它想要的表述呢？有两种策略：第一种就是内容协商，客户端通过一个 HTTP 报头的值来区分这些表述；第二种就是为一个资源分配多个 URL，一个 URL 对应一种表述，如表 3.2 所示。

表 3.2 同一内容的 XML 格式和 JSON 格式的两种表述

XML	JSON

<PlaceSearchResponse>
<status>0</status>
<message>ok</message>
<result_type>poi_type</result_type>
<results>
<result>
<name>中国建设银行 24 小时自助银行(北京天通苑支行)</name>
<location>
<lat>40.06701</lat>
<lng>116.421094</lng>
</location>
<address>北京市昌平区立汤路 186 号龙德广场 F1</address>
<province>北京市</province>
<city>北京市</city>
<area>昌平区</area>
<detail>1</detail>
<uid>2bb80dfd86d8417a0b69d9ee</uid>
</result>
…
</results>
</PlaceSearchResponse>
	{
	"status":0,
	"message":"ok",
	"result_type":"poi_type",
	"results":[
	{
	"name":"中国建设银行 24 小时自助银行(北京天通苑支行)",
	"location":{
	"lat":40.06701,
	"lng":116.421094
	},
	"address":"北京市昌平区立汤路 186 号龙德广场 F1",
	"province":"北京市",
	"city":"北京市",
	"area":"昌平区",
	"street_id":"2bb80dfd86d8417a0b69d9ee",
	"detail":1,
	"uid":"2bb80dfd86d8417a0b69d9ee"
	},
	…]
	}

2. 往来穿梭的表述

人们通常认为表述是服务器发送给客户端的数据，这是由于在上网时，发送的大部分请求都是 GET 请求，访问互联网多数时候都在请求获取表述。但是实际上，在 POST、PUT 或者 PATCH 请求中，客户端也会向服务器端发送表述，服务器随后的工作就是改变资源状态，这种情况下请求者的表述反映的是他所期望的未来的表述。

当客户端为了创建一个新的资源而发起一个 POST 请求时，它会发送它所期望的新的资源内容。服务器端的工作就是创建这个资源或者拒绝创建这个资源。客户端的表述只是一个建议，服务器可以根据请求者的要求增加、修改，也可以什么都不做，或者忽略表述的某一部分。

服务器发送的表述用于描述资源当前的状态；客户端发送的表述则用于描述客户端希

望资源拥有的状态，这就是所谓将资源状态通过表述“移交”。

用户使用 GET 请求表述比较简单，例如，获取 IFTTT 上面的一个用户信息表述。

```
GET /ifttt/v1/user/info HTTP/1.1
Host: api.example-service.com
Authorization: Bearer b29a71b4c58c22af116578a6be6402d2
Accept: application/json
Accept-Charset: utf-8
Accept-Encoding: gzip, deflate
X-Request-ID: 434d757081c94013b1b28f2087d28a98
```

但如果用户是 POST 给服务器端一个请求，则可能得到不一样的结果。以用户请求授权认证为例。

```
POST /oauth2/token HTTP/1.1
Host: api.example-service.com
Content-Type: application/x-www-form-urlencoded

grant _ type = authorization _ code&code = 67a8ad40341224c1&client _ id =
83465ab42&client_secret=c4f7defe91df9b23&redirect_uri=https%3A//ifttt.com/
channels/service_id/authorize
```

用户提供自己的认证信息，希望得到网站授权的令牌。正常情况下，服务器会反馈如下信息。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{
  "token_type": "Bearer",
  "access_token": "b29a71b4c58c22af116578a6be6402d2"
}
```

但如果认证不通过，则会反馈如下结果。

```
HTTP/1.1 400 OK
Content-Type: application/json; charset=utf-8

{
  "error": "invalid_grant",
  "error_description": "The code or token used is not valid"
}
```

这里，400 状态码表示的含义是：从 IFTTT 传入的数据出现了问题，服务端提供了一个错误响应体以澄清出错的原因。

3.3　超媒体与 HATEOAS

超文本(hypertext)是用超链接的方法将各种不同空间的文字信息组织在一起的网状文本，其中的文字包含有可以链接到其他文档的地址，可以从当前阅读位置直接切换到超文本链接所指向的文本。

超媒体(hypermedia)是超级媒体的缩写,是一种采用非线性网状结构对块状多媒体信息(包括文本、图像、视频等)组织和管理的技术[1]。超媒体在本质上和超文本是一样的,只不过超链接技术诞生初期管理的对象是纯文本,所以叫作超文本,随着多媒体技术的兴起和发展,超链接技术的管理对象从纯文本扩展到了多媒体。为强调管理对象的变化,就产生了超媒体这个词。

1945 年,美国科学家 Vannevar Bush 在《大西洋月刊》上发表了一篇文章 *As We May Think*,提出一种信息机器的构想——Memex(图 3.3)。

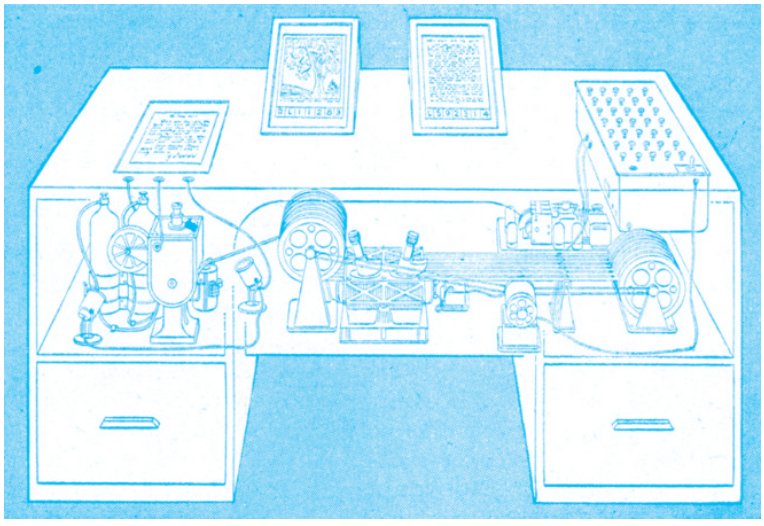

图 3.3 Memex 构想图

这种机器内部用微缩胶卷(microfilm)存储信息,也可以自动翻拍,以实现不断地向机器中添加新的信息;桌面上有阅读屏,该机器用投影放大、展示微缩胶卷中的内容;机器还有许多按钮,每一个按钮代表一个主题,只要按一下,相应的微缩胶卷就会显示出来。每一个胶卷内部还记录着其他相关胶卷的编号,用户可以方便地切换、阅读。

在 Bush 博士的设想中,这种机器可以与图书馆联网,通过某种机制将图书馆收藏的胶卷自动装载到本地。因此,只通过这一台机器就可以实现海量的信息检索。他将这种机器命名为 Memex,也就是 memory extender 这两个单词词首的组合,意思是“记忆的延伸”。这篇文章中关于信息切换的描述直接启发了“超文本协议”(hypertext)的发明。现在,人们在互联网网页上不同链接之间跳转访问,其源头都可以追溯到这篇文章。

超媒体体现了一种“关联”关系,即将资源关联到一起,这种关联实现了“1+1>2”的效果。善用超媒体技术,可以达成服务交互中的很多目的,故人们称为**超媒体策略**。

(1) 超媒体可以帮助服务器与客户端实现对话:服务器在发给客户端的文本中附加超链接,可以告知客户端如何进一步向服务器发起请求。

(2) 服务提供者可以在发给客户端的文本中附加广告信息的超链接,引导用户新的消费行为。

(3) 可以用超媒体写一个服务器提供的功能菜单,客户端可以从中自由选择,目前大多

[1] 温怀疆,何光威,史惠. 融媒体技术[M]. 北京:清华大学出版社, 2016.

数应用都是这样做的。

超链接将 Web 资源链接在一起，形成一张由多达数十亿计的 HTML 页面组成的网络，Web 则作为一个整体按照连通性原则运转。超链接可以有多种技术实现方式，可以被添加到图像上、文本上、按钮上等；在实际应用中，可以由开发者根据业务需求灵活地选择；Web 中的大多数链接是 HTML＜a＞标记和＜form＞标记这样的超链接形式，它们分别描述了针对资源的 GET、POST 等 HTTP 请求。

超媒体策略和超媒体技术有助于创建出更具灵活性的服务访问接口。但是超链接只是一种声明，它只是告知客户端服务器能做的事，最终还是要由客户端决定去不去访问这个链接。

超文本作为应用程序状态的引擎（hypertext as the engine of application state，HATEOAS）是 REST 的一项重要原则，罗伊·菲尔丁曾说过："如果应用程序状态的引擎（以及 API）不是由超文本驱动的，则它不能是 RESTful 的，也不能是 REST API"。借助 HATEOAS，应用程序服务器通过超媒体动态地提供信息，帮助客户端与网络应用程序交互，为 REST 资源返回的表述不仅包含数据，还包含指向相关资源的链接，只要能理解超媒体，REST 客户端几乎不需要其他额外知识就能与服务器交互。

例如，客户端向服务器端发送一个 GET 请求，客户端返回一个 JSON 表述的响应。

（1）请求。

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
Accept: application/vnd.acme.account+json
…
```

（2）响应。

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposit": "/accounts/12345/deposit",
            "withdraw": "/accounts/12345/withdraw",
            "transfer": "/accounts/12345/transfer",
            "close": "/accounts/12345/close"
        }
    }
}
```

注意：这个响应中包含了提示客户端后续可以执行的操作链接：存款、取款、转账及关闭账户。这些操作有可能引起客户端进一步的行动，引发新的状态转移，这就是"引擎"的

含义。

Github 的 API 就实现了 HATEOAS，用户请求 api.github.com 会得到一个 JSON 格式的列表，显示其所有可用的 API 地址如下。

```
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url": "https://github.com/settings/
connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url": "https://api.github.com/search/code? q={query}{&page,per
_page,sort,order}",
  "commit_search_url": "https://api.github.com/search/commits? q={query}
{&page,per_page,sort,order}",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  "events_url": "https://api.github.com/events",
  "feeds_url": "https://api.github.com/feeds",
  "followers_url": "https://api.github.com/user/followers",
  "following_url": "https://api.github.com/user/following{/target}",
  "gists_url": "https://api.github.com/gists{/gist_id}",
  "hub_url": "https://api.github.com/hub",
  "issue_search_url": "https://api.github.com/search/issues? q={query}{&page,
per_page,sort,order}",
  "issues_url": "https://api.github.com/issues",
  "keys_url": "https://api.github.com/user/keys",
  "label_search_url": "https://api.github.com/search/labels? q={query}
&repository_id={repository_id}{&page,per_page}",
  "notifications_url": "https://api.github.com/notifications",
  "organization_url": "https://api.github.com/orgs/{org}",
  "organization_repositories_url": "https://api.github.com/orgs/{org}/repos{?
type,page,per_page,sort}",
  "organization_teams_url": "https://api.github.com/orgs/{org}/teams",
  "public_gists_url": "https://api.github.com/gists/public",
  "rate_limit_url": "https://api.github.com/rate_limit",
  "repository_url": "https://api.github.com/repos/{owner}/{repo}",
  "repository_search_url": "https://api.github.com/search/repositories? q=
{query}{&page,per_page,sort,order}",
  "current_user_repositories_url": "https://api.github.com/user/repos{? type,
page,per_page,sort}",
  "starred_url": "https://api.github.com/user/starred{/owner}{/repo}",
  "starred_gists_url": "https://api.github.com/gists/starred",
  "user_url": "https://api.github.com/users/{user}",
  "user_organizations_url": "https://api.github.com/user/orgs",
  "user_repositories_url": "https://api.github.com/users/{user}/repos{? type,
page,per_page,sort}",
  "user_search_url": "https://api.github.com/search/users? q={query}{&page,
per_page,sort,order}"
}
```

OpenStack 也大量的使用到了这种设计，如下所示。

```
HTTP/1.1 200 OK
Content-Type: application/json

{"servers": [{
    "status": "ACTIVE",
    "links": [{
        "href": "http://192.168.10.111:8774/v2.1/e5ab2182bb984f3bb4773d4a83672549/
servers/95f684d4-0802-484e-b852-7ded35a8eeb5",
        "rel": "self"
    }, {
        "href": "http://192.168.10.111:8774/e5ab2182bb984f3bb4773d4a83672549/
servers/95f684d4-0802-484e-b852-7ded35a8eeb5",
        "rel": "bookmark"
    }],
    "image": {
        "id": "be4e8e37-226f-4784-b19d-a439400edca0",
        "links": [{
            "href": "http://192.168.10.201:8774/e5ab2182bb984f3bb4773d4a83672549/
images/be4e8e37-226f-4784-b19d-a439400edca0",
            "rel": "bookmark"
        }]
    },
    "flavor": {
        "id": "ed218eec-1e00-4ea9-93e7-f6e4e7c0ba93",
        "links": [{
            "href": "http://192.168.10.201:8774/e5ab2182bb984f3bb4773d4a83672549/
flavors/ed218eec-1e00-4ea9-93e7-f6e4e7c0ba93",
            "rel": "bookmark"
        }]
    },
    "id": "95f684d4-0802-484e-b852-7ded35a8eeb5",
    …
}]}
```

Spring 提供了对 HATEOAS 的支持，以简化开发者在 Spring 尤其是 Spring MVC 开发中创建遵循 HATEOAS 原则的 REST 表述的过程(图 3.4)，它所解决的核心问题是创建链接和组装表述。

下面介绍 Spring 的一个简单例子，即添加一个静态的链接。

```
public class WebSite extends EntityModel {
    private String name;
    public WebSite(String name) {
        this.name =name;
        add(new Link("https://www.google.com"));
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name =name;
```

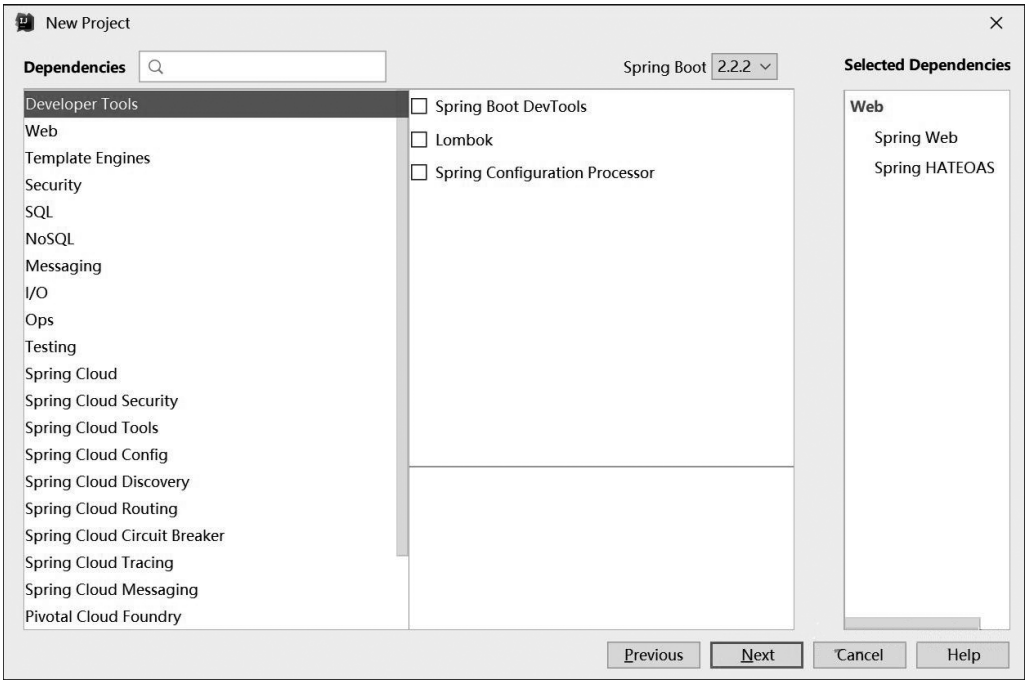

图 3.4 Spring 对 HATEOAS 的支持

```
    }
}
```

然后声明接口如下。

```
@GetMapping("/")
public EntityModel<WebSite>getGoogle() {
    WebSite webSite =new WebSite("Google");
    return webSite;
}
```

就可以返回以下结果。

```
{"name":"Google","_links":{"self":{"href":"https://www.google.com"}}}
```

Link 也支持模板的写法，如下所示。

```
Link link =new Link("/{segment}/something{? parameter}");
Map<String, Object>values =new HashMap<>();
values.put("segment", "path");
values.put("parameter", 42);
link.expand(values).getHref(); //返回/path/something? parameter=42
```

具体的开发案例见附录 A。

3.4 HTTP 的语义

在一个 Web RESTful 系统中，客户端和服务器端只能通过相互发送遵循预定义协议的消息来交互，这个协议就是 HTTP。客户端可以发送一些不同类型的 HTTP 消息与服务器

端交互。

每一个 HTTP 响应可以被分成 3 部分，如下所示。

(1) 状态码，亦称响应码。

其由三位数字组成，简要说明了请求目前的进展。响应码是客户端从响应中最先看到的信息，它奠定了响应剩余部分的基调。正确使用状态码能够给客户端以简明准确的信息。前面的示例中最常看到的状态码是 200(OK)，这是客户端所期盼的——这意味着一切进展顺利。

(2) 实体消息体(entity-body)，有时也被称为消息体。

这部分是一个采用某种数据格式书写成的文档，并且人们预期该文档是可以被客户端理解的。如果将 GET 请求理解成为获取表述而发起的请求，那么可以将实体消息体理解为客户端最终得到的表述(严格来说，整个 HTTP 响应都是“表述”，但是重要的信息通常都被记录在实体消息体中)。

(3) 响应报头。

响应报头的发送顺序排在状态码和实体消息体之间，通常是一系列用于描述实体消息体和 HTTP 响应的“键-值”对。

最重要的 HTTP 报头是 Content-Type，它向 HTTP 客户端说明了如何理解实体消息体。Content-Type 报头的值被称为实体消息体的媒体类型(media type)，媒体类型非常重要，它的值都具有特定的名称。就平时人们通过浏览器就能看到的 Web 信息而言，最常见的媒体类型是 text/html(针对 HTML 文档)。

HTTP 标准定义了 8 种不同类型的操作，除了之前介绍过的 GET、DELETE、POST、PUT 4 个最常用的操作外，下面两个方法是客户端在分析研究 API 时经常用到的。

(1) HEAD：获取服务器发送过来的报头信息(不是资源的表述)，这些报头信息是在服务器发送资源的表述时被一起发送过来的。

(2) OPTIONS：获取这个资源所能响应的 HTTP 方法列表。

另外两个定义在 HTTP 标准中的方法 CONNECT 和 TRACE 只被用于 HTTP 代理，所以暂且不对它们进行介绍。

第 9 个 HTTP 方法 PATCH 并没有被写进 HTTP 标准中，而是作为补充内容在 RFC 5789 中定义的。

PATCH 方法可以根据客户端提供的表述信息修改资源的部分状态。如果某些资源状态在提供的表述中没被提到，这些状态就保持不变；所以 PATCH 类似于 PUT，但允许对资源状态进行一些细粒度的改动，俗称“打补丁”。

总体来说，这 9 个方法确定了 HTTP 的基本协议语义。仅通过查看 HTTP 请求中所采用的方法就可以大概了解客户端要做什么了。

对 HTTP 操作的统一仅是完成了协议语义上的一致化，但对资源的操作的具体应用语义(application semantic)其实是无法统一的，因为资源可以是任何事物。向一个博客日志发送的 GET 请求和向一个股票代码发送的 GET 请求在协议语义上是一致的，但在拿到这个资源之前是很难知道资源的实际含义的。所以，无法仅通过使用 HTTP 来满足应用语义的统一要求，因为 HTTP 并没有定义任何应用语义。

1. GET

尽管可能完全不知道资源的应用语义，不明白资源能干什么，但 HTTP 的语义却很好地保持了一致。“获取一篇日志”和“获取一个股票报价”都应该被归为“获取一个资源的表述”，所以这两个请求都应该使用 HTTP GET 方法。

GET 请求中最常见的响应码是前文介绍过的 200(OK)。此外像 300(Moved Permanently)这样的重定向码也比较常见。

2. DELETE

当客户端想要删除一个资源时，它可以发送一个 DELETE 请求。客户端这时会希望服务器将资源销毁。当然，服务器没有义务来删除一些自己不希望删除的资源。

下面这个 HTTP 片段中，客户端要求删除一条信息。

```
DELETE /api/Messages/1234 HTTP/1.1
Host:https://developer.abc.com/
```

DELETE 请求成功发送后收到的状态码可能是 204(No Content，也就是“删除成功，我没有其他关于这个资源的信息描述了”)，如下所示。

```
HTTP/1.1 204 No Content
```

返回的状态码也可能是 200 (OK，也就是“删除成功，这里是关于它的一条消息”)或者 202(Accepted，也就是“收到，我稍后将删除这个资源”)。

如果客户端试图获取一个已经被删除的资源，那么服务器会返回错误响应码，通常是 404(Not Found)或者 410(Gone)，如下所示。

```
GET /api/Messages/1234 HTTP/1.1
Host:https://developer.abc.com/
HTTP/1.1 404 Not Found
```

很明显，DELETE 不是一个安全的方法。发送 DELETE 请求的效果不同于未发送请求。但是 DELETE 方法有另外一个很有用的特性：它是幂等的。一旦删除了一个资源，这个资源就消失了，资源状态也就永久性地改变了。再次发送同一条 DELETE 请求，可能会收到一个 404 错误，但是资源状态和第一次发送 DELETE 请求之后的状态是一致的：资源还是不存在的。这就是幂等性的好处，不管发送多少次同样请求，对资源状态的影响和发送一次请求时的影响是一样的。

幂等是一个很有用的特性，因为互联网不是一个可靠的网络。假设用户发送了一个 DELETE 请求，然后连接超时了，由于没有收到响应信息，所以用户无法确定之前的 DELETE 请求是否顺利完成。这时用户只需要再次发送 DELETE 请求并不断重试，直到收到响应信息为止。执行两次 DELETE 请求并不会比只执行一次造成更多的影响，HTTP DELETE 方法就相当于用零乘以一个资源。

3. POST

POST 方法有两种，第一种就是 POST-to-append，即向某个资源发送一条 POST 请求用于在该资源的下一级目录或结点中创建一个新的资源。在客户端发送一个 POST-to-append 请求时，它会在请求的实体消息体中添加资源的表述信息。

例如，使用 POST-to-append 通过一个新闻 API 发布一条消息，如下所示。

```
POSTnews/api/ HTTP/1.1
Content-Type: application/ json
{
"data" : [
{"title" : "Hangzhou sets stage for excellence",
"abstract" : "Asian Games venues completed as state-of-the-art facilities
promise to deliver exceptional event",
"content" : "With less than six months to go, preparations for the Hangzhou Asian
Games are in full swing……",
"datetime" : "2022-04-01 09:23"
}
]
}
```

对 POST-to-append 请求而言，最常见的响应码是 201(Created)，它用于告知客户端一个新的资源已经被创建成功，Location 报头用于告诉客户端这个新资源的 URL 地址；另一种常见的响应码是 202 (Accepted)，这表示服务器打算按照提供的表述信息创建一个资源，但是现在还没有真正创建完成。

POST 方法既不安全也不幂等，发送 5 次 POST 请求，会收到 5 条内容一模一样的消息，但它们却是 5 条独立的资源，因为具有不同的 URI。

除了用 POST“创建一个新的资源”之外，因为 POST 可以往服务器端发送内容，所以其被用来完成各种各样的工作，这是 POST 的第二种用法，被称为重载的 POST(overloaded POST)。

由于过去大部分浏览器只支持 GET/POST 方法，所以人们无法完美地实现 REST。对于这种情况，人们不得不将 PUT、DELETE、PATCH、LINK 和 UNLINK 等操作的用法混同为一个操作。

例如，POST 一个表单，然后在表单里加入一个名为 method 的隐藏字段，用于表示真正的方法，或者使用 X-HTTP-METHOD-OVERRIDE 头信息来重载 POST。

下面是一个 HTML 表单，其目的是编辑以前发布的商品描述。

```
<form method="POST" action="/merchant/items/1101">
  <textarea>
      A new description of goods.
  </textarea>
  <input type="submit" class="edit-description" value="Edit the description.">
</form>
```

在应用语义的语境中，“编辑商品描述”这个操作听起来像是一个 PUT 请求。但是 HTML 表单不能触发 PUT 请求，HTML 数据格式并不允许这么做，所以需要使用 POST 代替之。

这完全是合法的。因为 HTTP 规范中 POST 可以用于向数据处理流程提供表单提交结果的数据块。

这里“数据处理流程”可以无限扩展，用户可以将任何数据作为 POST 请求的一部分发送出去，不论是出于什么目的这都是合法的。

但这种用法下的 POST 方法并不真正表示“创建一个新的资源”，这将导致 POST 请求

实际上没有任何协议语义的一致性，使用户只能在应用语义的层面上理解它。

由于重载的 POST 请求可以用来完成任何工作，所以这种 POST 方法同样既不安全也不幂等。某个特定的重载的 POST 可能事实上是安全的，但是从 HTTP 协议层面考虑，仍然是不安全的。

这种用法显然带来了很多混乱，因此建议尽量不要使用重载的 POST。

4. PUT

PUT 方法用于修改资源状态。客户端一般会通过 GET 请求获取表述，然后对其进行修改，最后再将修改后的资源表述作为 PUT 请求的负载数据发送回去。例如，要修改一条消息的文本信息(将 abstract 字段的值修改以取代之前的内容)，内容如下。

```
PUTnews/api/q1w2e HTTP/1.1
Content-Type: application/json
{
"data" : [
{"title" : "Hangzhou sets stage for excellence",
"abstract" : "The 56 venues for the 19th Asian Games Hangzhou 2022 (Sept 10-25) and
the fourth Asian Para Games (Oct 9-15) have been finished on schedule, according
to the organizing committee.",
"content" : "With less than six months to go, preparations for the Hangzhou Asian
Games are in full swing……",
"datetime" : "2022-04-01 09:23"
}
]
}
```

服务器可以自由地拒绝一个 PUT 请求，理由可以是多种多样的，例如，实体消息类的意义不够明确，实体消息类试图修改服务器认为是只读的资源等。如果服务器决定接受一个 PUT 请求，那么它就会修改资源的状态，完成之后，通常会返回 200(OK)或者 204(No Content)状态码。

PUT 请求和 DELETE 请求一样是幂等的，发送 10 次同样的 PUT 请求，结果和只发 1 次请求的结果是一样的。

如果客户端知道新资源的 URL，那么它同样能够使用 PUT 新建一个资源。例如，想要发布一条新的消息，并且恰好还知道这条新消息的 URL，那么就可以用 PUT 操作来实现。

创建操作可以使用 POST 也可以使用 PUT，区别在于 POST 是作用在一个集合资源之上的(如/items)，而 PUT 操作是作用在一个具体资源之上的(如/items/12/3)。通俗点说，如果 URL 可以在客户端确定，那么就使用 PUT；如果是在服务端确定，那么就使用 POST，例如，使用数据库自增主键作为标识信息创建的资源，其标识信息只能由服务端提供，这个时候就必须使用 POST。

5. PATCH

“修改表述，然后通过 PUT 方法提交”是一个简单的规则。但是如果表述的信息量非常大，而需要修改的却只是资源状态中很小的一部分，这就可能造成极大的浪费；此外，PUT 规则还可能导致发生修改冲突。这时可以仅向服务器发送需要修改的部分数据文档，PATCH 方法就提供了这样的功能。

与将完整的表述信息通过 PUT 方法发送出去不同，用户可以建立一个特别的 diff 表

述，并将它作为 PATCH 请求的负载数据发送给服务器，如下所示。

```
PATCH /my/data HTTP/1.1
Host: example.org
Content-Length: 326
Content-Type: application/json-patch+json
If-Match: "abc123"
[
{ "op": "test", "path": "/a/b/c", "value": "foo" },
{ "op": "remove", "path": "/a/b/c" },
{ "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
{ "op": "replace", "path": "/a/b/c", "value": 42 },
{ "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
{ "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

对一个执行成功的 PATCH 请求而言，如果服务器想要向客户端发送数据（如已经更新的资源表），那么 200（OK）是最好的选择；而如果服务器仅仅想要表示执行已经成功，那么 204（No Content）就已经足够了。

PATCH 方法既不是安全的，也不能保证幂等，如果对同一个文档应用了两次 PATCH，可能会在第二次收到一个错误信息，但这并没有被定义在相关标准中。考虑到 PATCH 的协议语义，它跟 POST 一样是一个不安全的操作。

需要注意的是，由于 PATCH 方法是针对 Web API 而特别设计的扩展方法，并没有被定义在 HTTP 规范中，这也就意味着在工具支持方面，PATCH 方法及其所使用的 diff 文档提供的工具不如 PUT 方法丰富。

6. HEAD

HEAD 像 GET 方法一样安全，其可以被理解为轻量级 GET 方法。服务器处理 HEAD 方法的方式与 GET 方法类似，但是不需要发送实体消息体——只需要发送 HTTP 状态码和报头，如下所示。

```
HEADnews/api/ HTTP/1.1
Accept: application/json
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
ETag: "dd9b7c436ab247a7b69f355f2d57994c"
Last-Modified: Thu, 24Feb 2022 18:40:42 GMT
Date: Thu, 24Feb 2022 19:14:23 GMT
Connection: keep-alive
```

代替 GET 方法的 HEAD 方法并不会节约任何时间（服务器还是需要生成所有的 HTTP 报头），但是它确实能够节省带宽消耗。

7. OPTIONS

OPTIONS 请求是 HTTP 的原生探索机制。一个 OPTIONS 请求的返回结果包含一个 HTTP Allow 报头，这个报头展示了该资源所支持的所有 HTTP 方法。下面是一个 OPTIONS 请求例子。

```
OPTIONS /api/a1s2d3 HTTP/1.1
```

```
Host:https://example.com/
200 OK
Allow: GET PUT DELETE HEAD OPTIONS
```

已知资源所支持的 HTTP 方法后，用户可以方便地对该资源进行各种读写操作，OPTIONS 请求的意义便在于此。

8. HTTP 响应状态码(status codes)

REST 请求会遇到各种各样的情况，这些情况都需要通过 HTTP 状态码反映。

状态码是一个三位数字，被分成五个类别，每个类别都代表一种状态，具体见本书附录 B。

有时一些服务提供者会专门描述自己对状态码的定义，如 IFTTT 网站对服务请求的状态码约定如表 3.3 所示。

表 3.3 ITFFF 对状态码的说明

状　态	描　述
200	请求成功
400	从 IFTTT 传入的数据出现了问题。提供一个错误响应体以澄清出错的因
401	IFTTT 发送了一个无效的 OAuth 2.0 访问令牌
404	IFTTT 正试图访问一个不存在的 URL
500	应用逻辑中存在错误
503	请求的服务目前不可用，但 IFTTT 稍后会再试

3.5 操作资源

HTTP 方法够用么？从上文内容可以看出，使用已有的 POST、DELETE、PUT、GET 四种方法就可以增、删、改、查资源。

但在实际情况下人们需要做的操作往往并不仅局限于增、删、改、查，例如，要把一篇文章“置顶”，但是 HTTP 方法中没有一个和“置顶”操作相对应的方法，这时该怎么办呢？

REST 对类似问题的解决方案是创建一个新的资源。例如，上面的例子可以使用 PUT 方法实现，如下所示。

```
PUT /toparticles/123
```

通过创建一个新的资源(toparticles)，可以使用简单的 HTTP 方法实现一切操作。

再举一个例子，实现银行转账可以把账户看作是一个资源，在这个资源上的存储操作体现出来就是账户余额值的变化，转账涉及的就是两个账户余额的此消彼长。这在数据库中是一个典型的事务操作：张三给李四转账 100 元，实际上的操作分为两步，第一步，张三账户余额减去 100 元；第二步，李四账户余额增加 100 元。事务的作用就是要保证这两步要么全部成功，要么全部失败。

标准的 HTTP 操作并没有“增加”“减少”“转账”这些操作。如果用服务实现，可以设计

一个新资源“账户交易”，以之作为账户的从属资源，每次账户发生存取款时都在账户下面 POST 一个新的账户交易资源，用正负数值表示存款与取款，而真正的账户变动则由服务端的数据库事务操作完成。另外，还可以设计一个转账资源，资源表述中包括转账的目的账户，也以之作为账户的从属资源，每次账户向其他账户转账，就在账户下面 POST 一个新的转账资源，真正的转账操作也由服务端的数据库事务操作去完成。

在设计服务时需要把握一点：HTTP REST 接口应该是粗粒度的，不应该是暴露对后台数据库增、删、改、查的细粒度操作。

另外，设计良好的 API 会响应 GET 请求并返回一个超媒体说明文档，用这个文档来宣传自己，这些文档中的链接和表单阐明了客户端下一步所能发起的 HTTP 请求。而设计低劣的 API 则只会使用人类可读的文档来说明客户端能发起哪种 HTTP 请求。

本章习题

1. 如何理解资源的本质？
2. 如何理解表述的本质？
3. 如何理解资源与表述的关系？
4. 资源操作能否仅依赖 HTTP 方法？
5. 如果 HTTP 方法中没有需要的操作方法，这时该如何设计？
6. 何为重载的 POST 操作？为什么在开发 REST 架构风格的 API 时不建议使用重载的 POST？
7. HEAD 方法有什么用处？
8. 为什么需要善用状态码/响应码？
9. 如何理解超链接在 Web 中的重要作用？
10. 请以在电商网站购买商品的过程为例，列出可能涉及的资源、对资源的 HTTP 操作。

第4章

认识 RESTful 资源：以地图服务为例

地图是人们日常生活中一种常见的工具，《周礼·地官·土训》有“掌道地图，以诏地事”，也就是说手握地图，给人一种俯瞰天下的可能。信息时代，出现了卫星遥感影像，不但给地图制作提供了新的数据源，还可以把影像直接作为地理事物的表现形式；而北斗导航等卫星定位技术，可以将个体的位置直接关联到电子地图中。各种各样的地图以及附加在地图上的资源给人们的出行和生活带来很大方便，这里面也离不开 RESTful 风格的地图服务的推动。

本章以地图为例，帮助读者认识我们身边这些常见的 RESTful 资源。

4.1 基于位置的服务

随着 GPS、北斗等空间定位技术与移动网络的快速发展，持有移动终端的人们可以方便地采用定位技术获取自身当前所在位置，因此通过网络向定位设备提供信息的“基于位置的服务”(location based services，LBS)应运而生。国内 LBS 发展迅速，出现了百度、腾讯、高德等一批骨干服务商和国家地理信息服务平台(天地图)等一批专业的位置服务机构，它们都提供了基于位置的服务接口，构建了基于地图 API 的开发者生态。

(1) 百度地图 Web 服务 API(图 4.1)为开发者提供 HTTP/HTTPS 接口，开发者通过 HTTP/HTTPS 协议发起检索请求，获取 JSON 或 XML 格式的返回数据，以这些数据为基础开发基于 JavaScript、C#、C++、Java 等语言的地图应用。

(2) 高德地图 Web 服务 API(图 4.2)向开发者提供 HTTP 接口，开发者可通过该接口使用各类型的地理数据服务，返回结果同样支持 JSON 和 XML 格式。高德地图 Web 服务 API 对所有用户开放，但不同类型用户能够获取的数据范围有所不同。

(3) 腾讯地图 Web 服务 API(图 4.3)基于 HTTPS/HTTP 协议的数据接口，开发者可以使用任何客户端、服务器和开发语言，按照腾讯地图 Web 服务 API 规范，按需构建 HTTPS 请求并获取结果数据(目前支持 JSON/JSONP 方式返回)。

(4) 天地图 Web 服务 API(图 4.4)为用户提供 HTTP/HTTPS 接口，开发者可以通过这些接口使用各类型的地理信息数据服务，可以基于此开发跨平台的地理信息应用。

图 4.1　百度地图 Web 服务 API 首页

图 4.2　高德地图 Web 服务 API 首页