RESTful概念及其最佳实践

周淳威

HTTP基础

在最常见的场景下[1]，现代互联网架构于HTTP协议[2]之上，且自从Internet这一概念被发明以来一直如此。HTTP协议的风靡很大程度上要归结于它是一种灵活而通用的应用层协议：尽管它最初的设计目的以及命名都旨在用于传输网页内容，但它所能做到的远不止于此。

HTTP的设计本身决定了HTTP具有出色的灵活性。HTTP将网络传输抽象为请求（Request）与响应（Response）两种行为，向服务器发起的一个请求对应服务端的一个响应。HTTP认为，服务端保存着资源（Resources），而发起请求的过程，就是客户端向服务端指出自己的资源需求的过程；服务端返回响应，也就是将客户端所请求的资源交给客户端。资源是一种十分抽象的概念，我们可以用其指代任何请求者所需要而服务端拥有的文件、资料、信息、数据等；正因如此，HTTP协议可以超越最初的传输网页数据的限制，成为一种通用的数据传输协议。

在HTTP/1.1版本及以前，HTTP本身是一种纯文本协议，其协议自身不包含任何二进制数据。HTTP/1.1标准[3]（如无特殊说明，下文均指HTTP/1.1）规定请求的格式如下：

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

HTTP的格式定义是相当直观的。作为客户端，想要向服务端请求资源，那么显然需要告诉服务端：

- 自己需要获得（GET）资源
- 自己想要的到底是什么资源，它在服务端的位置（Path）如何（/）
- 自己所使用的协议版本（HTTP/1.1），以保证服务端能理解客户端消息的含义。

上述三个要素构成请求的报文（Message）的第一行（Request line）。随后的行构成报文的头部（Headers），它采用`Name: value`的格式，表示的是关于此次请求的元数据（metadata，亦即"关于数据的数据"，即用于描述某些data的特征、性质等的data），描述了这次请求自身的一些特征，服务端可以根据这些元数据来对返回的资源做相应的控制（当然，亦可以忽略）。例如，`Host`指定这次请求需要访问目标服务器上名为`developer.mozilla.org`的站点（Site），而非其他的站点（此种设计的目的是为了提高服务器的利用率（utilization），使得一台物理服务器与一个IP地址上可以同时提供多个网站，使用该头部的值即可进行区分，而不必每个网站都需要使用单独的服务器机器与IP地址）；`Accept-Language: fr`表示此次请求希望获取资源的法语版本；但服务端当然可以忽略这些头部，例如，如果它所拥有的资源没有法语版本，它便可以不执行`Accept-Language`的请求。

头部的设计是HTTP灵活性的一大来源。HTTP标准只规定了头部的格式，以及一组和HTTP协议自身有关的标准头部。除此之外，我们可以任意定义自己的头部（即，在发起请求时，通过编程加入我们自定义的头部；在服务端处理请求时，可以编程读取自定义的头部），用于传达我们所需要的额外信息，这就大大扩展了协议的灵活性。

HTTP规定响应的报文格式如下：

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html...
```

响应的报文格式略有不同，但仍然非常直观。响应的首行（Response line）用于表示服务端执行这次请求的状态，包括协议版本，状态码（status code）以及状态码的文字表述（OK）；在上面的例子中，这一行表示服务端成功执行了客户端发来的请求。随后是本次响应的头部，它们主要描述随着这次响应返回的资源的元数据，如资源的大小（Content-Length，单位为byte）与类型[4]（Content-Type）。在响应报文中，我们通常期望得到所请求的资源，后者被放置在响应的包体（Body）中。包体可以是任意的数据，例如"Content-Type"，那么显然我们就需要一种标识来区分头部和包体，此即上例中第5行的空行（实际上是CRLF（`\r\n`），HTTP标准规定使用CRLF作为换行符[5]）。HTTP标准规定首行与头部必须连续排列，随后用一个空行分割，而后是包体数据。

上文对HTTP的请求与响应模型做了基本的阐述。顾名思义，我们容易认为只有响应才需要包含包体数据，但实际上并非如此。请求报文同样可以包含包体数据，因而实际上请求报文和响应报文之间最大的差距是首行格式不同。什么样的情况下请求报文也需要包含包体呢？最典型的场景是我们需要向服务端提交数据。例如，购买火车票时向服务端发起的请求就应该包含我们的个人信息，以供服务端进行验证。换言之，虽然服务端拥有资源，但很多时候访问这些资源首先需要客户端提交一定的信息；因此请求报文的包体也是相当重要的。在这样的情况下，我们有必要对首行进行一些修改，以告知服务端在请求报文中包含数据，而并非单纯的获取（GET）资源，如下所示：

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

我们需要在首行表明我们随请求提交（POST）了数据，而非单纯获取（GET）资源（理论上而言，GET请求也可以包含包体，但绝大部分情况下，服务端会忽略随GET请求提交的包体）。从上面的表述中我们可以看到，尽管同样是请求，但不同的请求可能希望对服务端资源执行不同的操作，因此HTTP协议中对这一动词进行了标准化，称为HTTP方法（Method），它用于告知服务端这一请求想对服务端资源执行的操作，包括`GET|POST|PUT|PATCH|DELETE`等等。还可以注意到，`Content-Type`的值目前看起来

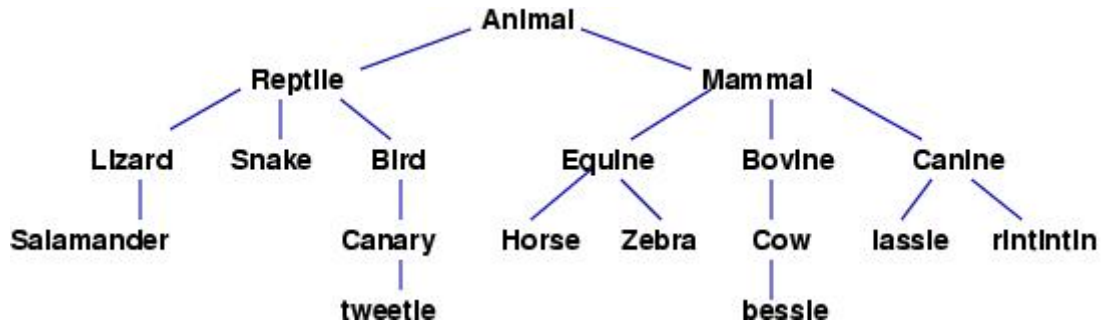
并不好理解，包体的数据格式也有些奇特，这里我们暂且按下不表。实际上，HTTP对包体的内容没有任何限制（只需要服务端程序可以理解即可），这也是HTTP灵活性的另一来源。

Motivation

HTTP为我们提供了一种便捷的访问服务端资源的方式，以及一个清晰的网络模型。但HTTP标准对于服务端应该提供什么样的资源（如：资源的内容、位置等）并没有任何规定；再加上HTTP本身的灵活性，通过HTTP进行数据的传输便显得有些混乱。然而，在计算机科学中，某种程度上，混乱便意味着低效；反之，如果在某一领域具有统一的标准，那么该领域往往可以围绕标准良性发展。RESTful便是在这样的动力下出现的。

REST[6]，全称为Representational State Transfer，或译为表现层状态转换，是由Roy Thomas Fielding于2000年提出的一种**风格**，换言之其并不是强制性的标准。它的名称直观看来有些不好理解，但如果我们理解了它的目的，那么其命名所要传达的意图其实是非常清晰的。

如前所述，HTTP认为服务端保有资源，客户端可以通过请求来获取资源。但服务端应当如何提供资源呢？要理解这一问题，我们首先要思考的就是，客户端应当如何**找到**服务端的资源？“名不正而言不顺”，**命名**（Naming）是计算机系统的关键思想；只有我们给予资源一个名称，我们才有可能通过该名称找到对应的资源。在HTTP请求报文中，首行的Path部分即是服务端资源的名称。某种程度上，它可以理解为服务器硬盘上的一个路径，与Unix风格的路径十分类似，例如`/home/user/downloads`。若如此解释（Interpret）路径，我们就可以通过HTTP来命名服务器上存储的文件资源。但实际上，路径并不仅仅可以用于表示文件的存储位置，它也可以被解释为一种抽象的层次表示，用以命名任何组织为树形结构的资源。例如，一个图书馆的服务器可以提供如下的路径：`/books/computer_science/modern_operating_system`，表明电子版《Modern Operating System》在图书馆服务器中的逻辑层次结构（`books`分类下的`computer_sienece`子分类），但实际上该书的电子版并不一定存储在服务器硬盘上名为`books`的目录下名为`computer_science`的子目录中。由此我们可以发现，路径这一命名方法提供了很大的灵活性，但与此同时，缺乏标准的路径结构，也容易引发混乱。除此之外，如前所述，客户端可以通过指定HTTP请求方法，来对服务端上的资源执行一定的操作。尽管HTTP标准规定了一组动词，但实际上关于如何解释这些动词，并无强制性的规定。



> 如图，树状结构即是典型的层次（Hierarchy）结构。

REST提议（propose）了一种标准的风格，用于解决上述两大问题。而我们不难发现，上述两个问题，实际上是一种对服务端资源的进一步抽象（Abstraction）。对上述问题订立的标准，给予了服务端的资源一种更具体、更容易理解的表现形式，而不仅仅是一个抽象的概念，换言之即是规定了资源的表示方法（Representation），此即REST这一直观看来不太好理解的名称的用意。

REST

简而言之，REST主要包括一种标准的资源路径格式，以及一组标准的HTTP方法的语义（Semantic）定义，分别用于解决前述的两大问题。

路径的本质是命名，REST规定一个资源的名称分为两部分，即该资源所在的集合名称，以及该资源自身在该集合中的唯一标识（ID）。又由于我们需要使用路径的形式来表述这一命名，因此我们可以直接得到一个资源在REST中的典型路径：

```
/books/1
```

它表示一本书的数据（即资源个体）在一个符合REST风格（即RESTful）服务器上的路径。显然，它所属的集合是书籍，`books`表示集合名称，而`1`则是它自身在集合中的唯一ID。二者相结合即可唯一命名该资源。

集合当然可以具有层次结构，例如：

```
/books/literature/1
```

表示该ID为1的书籍所属的集合是`literature`，而后者所属的集合是`books`。REST规定集合名称需要是一个名词，且如果可能的话，使用名词的复数形式。由于名称由集合名和ID两部分组成，因此显然地，在不同的集合中可以存在相同的ID。

REST的另一贡献是为不同的HTTP方法规定了标准的语义。我们知道，HTTP方法指定了请求试图对资源执行的操作，而广义上来说，资源并不仅仅包括集合中的个体，集合本身显然也是一种资源。因此，REST将操作分为两类：对于个体的操作与对于集合的操作。REST主要利用HTTP标准中规定的`GET|POST|PUT|PATCH|DELETE`五种方法，它们分别具有对个体与对集合的语义。基本上，我们可以将对于数据的操作概括为**CRUD（Create, Retrieve, Update, Delete）**。换言之，REST便是利用HTTP协议，提出了一种标准的CRUD语义格式，具体如下表所示：

方法	路径	语义	响应包体
GET	/books	获取集合中的个体资源列表	一个列表，其中包含集合中的每个个体的数据
POST	/books	根据请求报文包体中的数据创建该集合下的新资源（通常为新的个体）	新的资源的数据
PUT	/books	N/A （即不合法）	N/A
PATCH	/books	N/A	N/A
DELETE	/books	N/A	N/A
GET	/books/1	获取集合中指定ID的个体的数据	集合中指定ID的个体的数据
POST	/books/1	N/A	N/A
PUT	/books/1	使用请求报文包体中的数据**完整替换**（可理解为删除后新建）服务端已有的该个体资源的数据	替换后的新资源个体的数据
PATCH	/books/1	使用请求报文包体中的数据**部分替换**（即：只替换包体中给出的部分，未给出的部分仍按原状）服务端已有的该个体资源的数据	替换后的新资源个体的数据
DELETE	/books/1	删除集合中指定ID的个体	空

RESTful最佳实践

REST随着前后端分离的开发模式的扩张迅速成为数据型后端开发的事实标准。但于2000年提出的REST，在很多场景下并不能满足日益复杂的应用的数据需求。在工业界长期的实践中，RESTful经历了大量的改进，形成了许多最佳实践（Best Practice）。本节旨在基于REST设计思想的前提下，介绍笔者在RESTful后端开发过程中所采取的一些实践经验。

理解RESTful语义

REST在提出时所规定的这一组语义，在很多场景下已经不能满足后世应用的数据需求。但若要找出目前我们所了解的语义的问题，我们首先需要对这组语义设计时的一些考量做进一步的阐述。

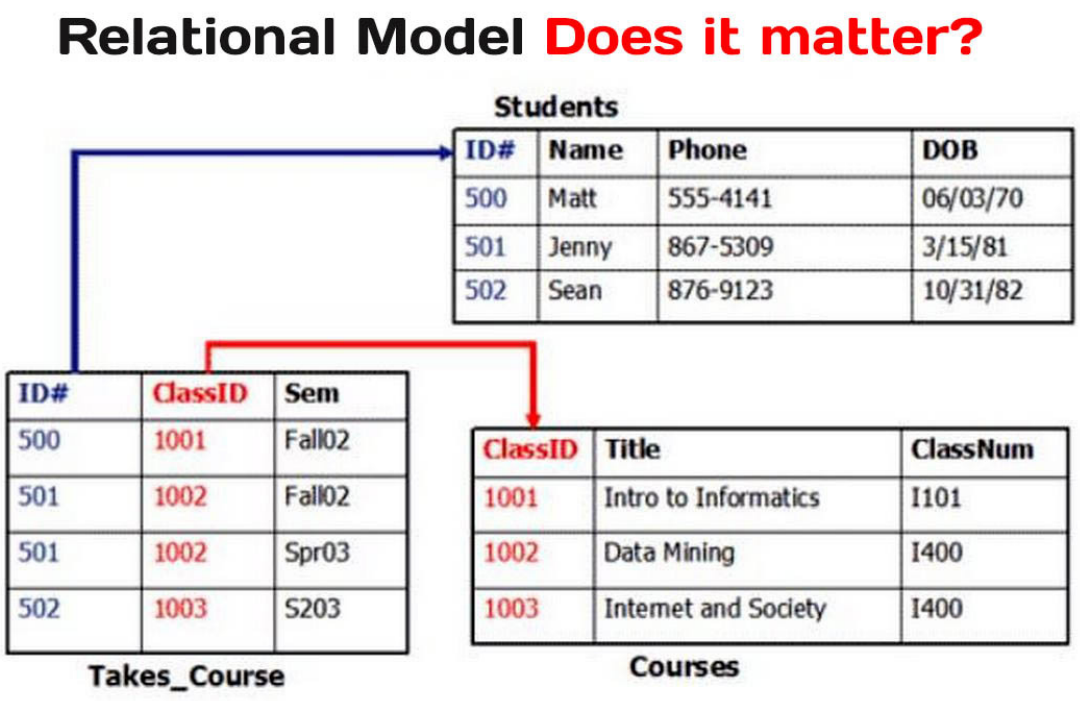
直观上不太好理解的可能是POST/PUT/PATCH的语义。它们（在有效的前提下）共同的特点是服务端应在响应包体中返回被影响个体的**完整**数据。一种常见的问题是，客户端已在POST时提交了数据，为什么还需要返回新资源的数据呢？这是因为客户端所提交的数据和服务端中存储的实际数据在内容上常常有所不同。例如，一个集合中资源的ID常常应当由服务端控制，若要创建一个新的资源，其ID应当由服务端分配而非客户端指定，以避免许多问题，例如多个客户端同时添加资源所产生的数据竞争（Race Condition[7]）等。那么，在这样的情况下，客户端不可能提前得知它需要创建的新资源的ID，这一数据只能由服务端在响应包体中回传给客户端。此种**输入输出数据形式的不一致性**导致了RESTful设计上的许多考量，我们后续进一步阐述。

关系模型

> No Man Is An Island —— John Donne

关系模型（Relational Model[8]）是目前广泛应用的用于描述数据的一种思想；尽管它常常与关系型数据库（Relational Database）及SQL密不可分，但它的本质其实是对现实世界中的数据进行建模的一种通用思想。限于篇幅原因，我们只用较为直观的方式介绍关系模型。

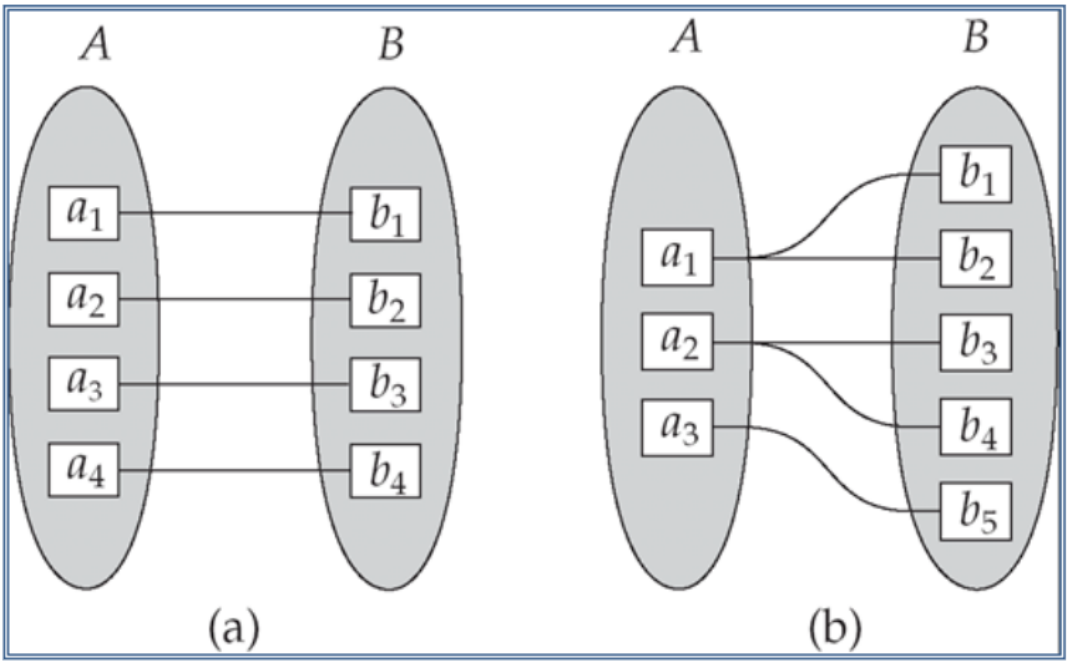
关系模型认为，现实世界由对象与对象间的关系组成。对象是客观上或逻辑上独立的实体，例如学校、学生；关系表示不同对象之间的关联，在不同的对象间，存在不同的关系，构成了现实世界。例如，教师与课程之间存在教授关系，学生与课程之间存在修读关系，等等。关系模型通过将一类对象的共同特征（features）抽取为实体集（Entity Set，集合中的不同个体对于集合中的features拥有不同的取值）来建模现实世界中的对象；再通过定义不同对象间的关系，从而描述现实世界中的数据。



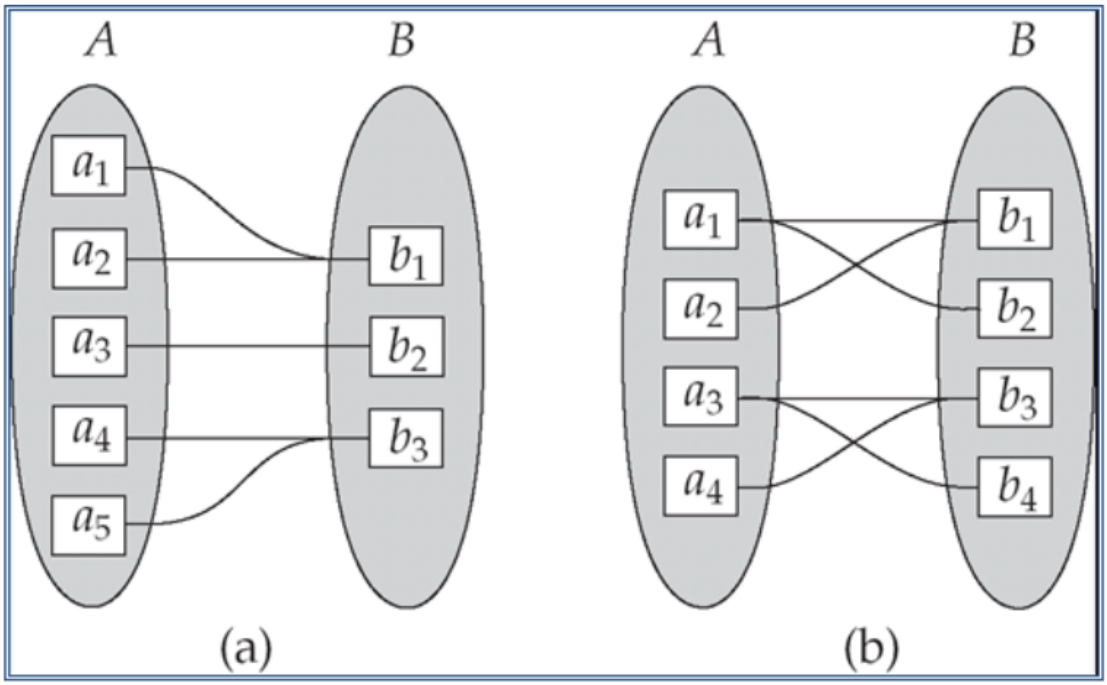
> 典型的关系模型

关系模型中的关系是一种较为抽象的表述，简单起见，我们只介绍二元关系，这样的关系只涉及到两种不同的对象。关系模型中的关系摒弃了一切现实中的关联与交互，它关注的是两类对象间的**对应**关系，主要分为如下四种：

- 一对一：一个对象与另一个对象一一对应；例如，一名学生拥有一个学号，一个学号对应一个学生
- 一对多/多对一：一个对象与多个对象相对应（或反之）；例如：一名学生只能属于一个班级，但一个班级可以拥有多名学生
- 多对多：多个对象与多个对象相对应；例如：一名学生可以修读多门课程，而一门课程同时也可以有多名学生



One to one　　One to many



Many to one　　Many to many

了解了关系模型，我们不难发现REST的设计思想与关系模型表达数据的方式存在高度的重合。原因是非常直观的。我们习惯于使用关系模型来描述数据和存储数据，HTTP只不过是用于传输这些数据，因此构建在HTTP之上的REST自然要适应关系型数据的需求。这也从另一个角度解释了RESTful处于表现层（Representation），它只是提出了一种表示数据的方式，而并未颠覆数据的本质结构。

初探RESTful包体

如前所述，关系模型非常适合用表格的形式来表示，而具体到实现层面，数据表（Table）亦是基于关系模型的数据库系统提供的基本抽象。但表格这种二维形式，非常不适合在网络中进行传输。因此，我们需要一种可序列化（Serializable）的形式来表达关系模型数据，以便在网络中进行传输。

键值对模型（Key Value Model）是一种通过键值对（`Name: value`）来表达数据的思想，HTTP头部即是典型的键值对模型。 键值对模型认为每个键代表数据的一项特征，不同的数据对于相同键的取值不同。键值对模型具有线性、容易在网络上传输的优势。对键值对模型进行简单扩展，我们即可使用它来表达关系模型下的数据：

- 一个对象由一组键值对组成
- 使用列表来表示复数个对象的集合
- 键的值可以是另一个对象或者列表

第一项扩展，使得键值对模型可以用于表示关系模型中一个表格的单行数据；第二、三项扩展，使得键值对模型可以用于表示对象与对象间的关系，前者使得键值对模型可以用于表示一对一关系或多对一关系，后者使得键值对模型可以用于表示一对多关系或多对多关系。

键值对模型（包括上述扩展）只是一种思想，要将其用于具体的数据传输，我们仍然需要标准进行规范。有了这样的规范后，我们就可以用它们来表示REST风格下的响应数据。在目前的RESTful实践中，最常见的标准是JSON[9]，但事实上也可以是任何其他标准，只要其采用键值对模型，且满足上述的三项扩展即可。

```
// JSON示例
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

我们后续还会进一步讨论包体的问题。

命名：ID

在之前的介绍中，我们看到ID就是一个连续的整数，这是REST中ID最常见的形式。但实际上，ID的本质是在相同集合中不同个体的唯一标识符，因此我们可以使用任何具有这样性质的线性形式。一种常见的替代形式是UUID[10]。它的本质上是具有强唯一性的字符串。相较于连续整数，UUID的优势在于其具有更好的防碰撞性。例如，若使用连续整数，那么任何客户端只需要连续GET请求`/books/1` `/books/2` … `/books/100`即可获取ID为1-100的全部数据；而UUID并非连续的，因此可以一定程度上防止这样的恶意行为。此外也可以根据实际数据的性质使用其他的ID，例如身份证号码、学号等等。

ID的形式选择是典型的tradeoff。以连续整数与UUID为例，前者实现简单，而且存储所需的空间更小，性能通常较好，但抗碰撞性很差；后者抗碰撞性强，但需要更大的空间来存储，且性能相对较差。因此需要根据业务的实际需求进行选择。

命名：多级命名与嵌套命名

我们现在开始考虑如何扩展REST以满足现代应用对数据的更高需求。一种常见的需求是提高命名的可读性。在介绍REST时，我们曾介绍过类似`/books/literature/1`这样的路径结构，这就是典型的层次结构。层次结构是提高命名可读性的有效方法，但如果我们决定需要采用层次结构，就需要考虑避免二义性。具体来说，如果如下两种路径结构同时存在，将会产生歧义：

```
/books/{title}
/books/literature/{title}
```

{title}表示此处使用书籍的标题作为ID。由于路径需要交由服务端程序来处理，一个含义清晰无歧义的路径结构是必需的。但问题在于，具体书籍的标题和此处用作分类层级的“literature”一样，均是字符串。那么若恰好有某本书的title也为“literature”，则会产生歧义。常见的实践是，若决定采用层次结构，则不提供如上的第一种路径，或避免采用title作为ID，而使用连续整数等方案，总而言之，必须消除ID和路径层次间产生歧义的可能性。

如前所述，我们通常通过RESTful接口来提供关系型数据服务。在这样的情况下，一个很自然的需求是对由关系产生的子集合进行操作，而不是直接针对完整的集合进行操作。考虑一个选课系统，每名学生可以选择修读多门课程，学生与课程之间为多对多关系；那么，对于某名特定的学生，他/她所选修的课程就构成完整的课程集合的一个子集，且这一子集是通过选修这一关系确定的。如果我们可以直接针对一名学生所选修的课程集合进行CRUD等操作，无疑能大大降低接口的使用成本。

嵌套（Nesting）指的是将一个完整的结构作为另一个具有相同形式但规模/层次更高的结构的子结构的设计思想。具体而言，我们可以在REST的命名结构上进行嵌套。例如，要实现上述的需求，我们可以如此定义路径：

```
/students/{studentID}/courses/{courseID}
```

在此处，我们把一个完整的REST路径结构`/courses/{courseID}`嵌入到了一个更高层次的路径中，这应用了嵌套的思想。由于这样的路径结构并非REST规范的一部分，我们可以自行定义扩展的语义。例如，`/students/{studentID}/courses`表示特定学生选修课程的子集合，我们可以规定对它的GET请求只返回该学生选修的课程，而不返回其他课程。我们还可以规定对该路径的POST请求表示将已有的课程添加到该选修关系中，而非创建新课程……总之，需要从中学习到的思想是，我们可以通过扩展REST的路径结构，来获得接口上更强的表达能力。

命名/语义：非REST路径结构

从更高的层次来看，REST的设计思路是通过路径对资源进行命名。某些场景下，我们只需要学习这一思想，但具体的RESTful路径结构反而降低了我们的接口的可读性。笔者遇到的一种典型场景是，一个系统中，每个用户有一份用户资料一一对应；那么，在设计访问用户资料的接口路径时，如果还遵循标准REST结构如`/userprofiles/ID`就显得有些过于累赘了——该接口其实并不需要访问一个集合，或者说它所访问的集合中有且仅有一个元素。REST所建议的集合名称+集合内ID的命名方法在这样的场景下显得可读性较低，且REST的标准语义也有悖于我们的需求，例如，我们显然不希望用户通过GET `/userprofiles`就能获取完整的用户资料集合的列表；也不希望用户通过POST `/userprofiles`这样的请求创建新的用户资料；除非用户注销帐号，否则我们也不希望用户通过DELETE `/userprofiles/id`的请求删除任何用户资料……

笔者依据自身的经验，将这样的场景称为单例（Singleton）接口（这一名称来源于设计模式中的单例模式[11]，指的是某个类的对象有且仅有一个的场景），它使用`/cur_{name}`这样的形式来表达一组特定的语义，前缀cur表示“current”。例如，路径`/cur_userprofile`即表示访问当前登录用户的用户资料的接口。

笔者在实践中所采用的单例接口语义如下：

方法	路径	语义	响应包体
GET	/cur_userprofile	获取当前登录用户的用户资料（单个对象）	当前登录用户的用户资料
POST	/cur_userprofile	若当前登录用户的用户资料不存在，则使用包体数据创建新的用户资料；否则完整替换已有的用户资料	创建或替换后的用户资料
PUT	/cur_userprofile	完整替换	替换后的用户资料
PATCH	/cur_userprofile	部分替换	替换后的用户资料
DELETE	/cur_userprofile	依据实际需求而定，可禁止用户发起该请求，也可表示删除当前登录用户的用户资料	空

语义：分页(Paging)/过滤(Filtering)/排序(Sorting)/投影(Projection)

如今看来，REST所规定的对于集合的GET方法语义有严重的问题，我们接下来介绍一系列针对**对于集合的GET方法**的语义改进设计。一方面，现代应用的数据量普遍较大，若一次GET请求就要获取一个集合中的全部数据，不论从服务端机器性能还是从网络传输还是从客户端处理的角度而言都是无法接受的，这就要求我们一次只能返回一小批数据（如同书的一页），这一技术称为**分页（Paging）**；另一方面，现代应用对于某个集合的数据列表通常也会提出更高的需求，典型的需求包括根据一定的条件筛选数据，而非获取任何个体，这一技术称为**过滤（Filtering）**；另一种典型的需求是对返回的列表中的数据条目根据一定的规则（如时间）进行**排序（Sorting）**。

以上三种需求常常共同出现。在此种情况下，我们一般期待的行为是，首先对完整集合进行过滤获得子集合，随后对子集合进行排序，最后从排序后的集合中取出一页数据作为响应返回。当然，要实现这样的行为，一种可能的方式是后端不提供这样的语义，将所有数据全部发往前端，由前端进行处理，但如前所述，如此会面临严重的性能/网络传输/数据安全问题；因此我们必须在后端引入这样的语义。

REST本身的表达能力并不足以表达上述的任何一种需求，我们需要通过其他的方式扩展REST，以让请求可以描述其分页/过滤/排序的需求。我们先前已经了解到，通过改进路径结构可以增强REST的表达能力，但此种途径属于命名方式上的改进，与我们这里的需求并不一致。因而我们接下来介绍扩展REST表达能力的重要方法：**使用查询字符串（Query String [12]）表达复杂的数据需求。**

HTTP链接乃是URL [13] 的一个子集，而后者规定了在URL末尾可以附有一段特殊的字符串，用于指定和所指定的资源相关联的一组参数。我们可以利用这一语义来表达我们的分页/过滤/排序需求。查询字符串的格式如下：

```
https://example.com/path/to/page?name=ferret&color=purple
```

标准规定以`?`后为查询字符串，它也采用键值对模型，以`&`分隔不同的键值对。我们可以通过定义特定的键名称，来表达我们的数据需求。笔者通常定义如下参数名：

名称	取值	语义	默认值
page	正整数，从1开始	要获取的数据的页号	1
size	正整数	每页的数据条目上限	10
sort	字符串，依据实际数据而定	通常为`+键名`或`-键名`，表示按照返回的数据列表中的某个键进行升序或降序排列	预先定义的某种顺序
fields	以逗号分隔的键名列表，如`id,name,title`	投影数据，详见下文	全部键名，即无投影
其他	依据实际需求而定	过滤参数，可根据实际需要定义，例如`name=ferret`与`color=purple`表示筛选`name`为`ferret`且`color`为`purple`的数据条目	根据实际需求而定

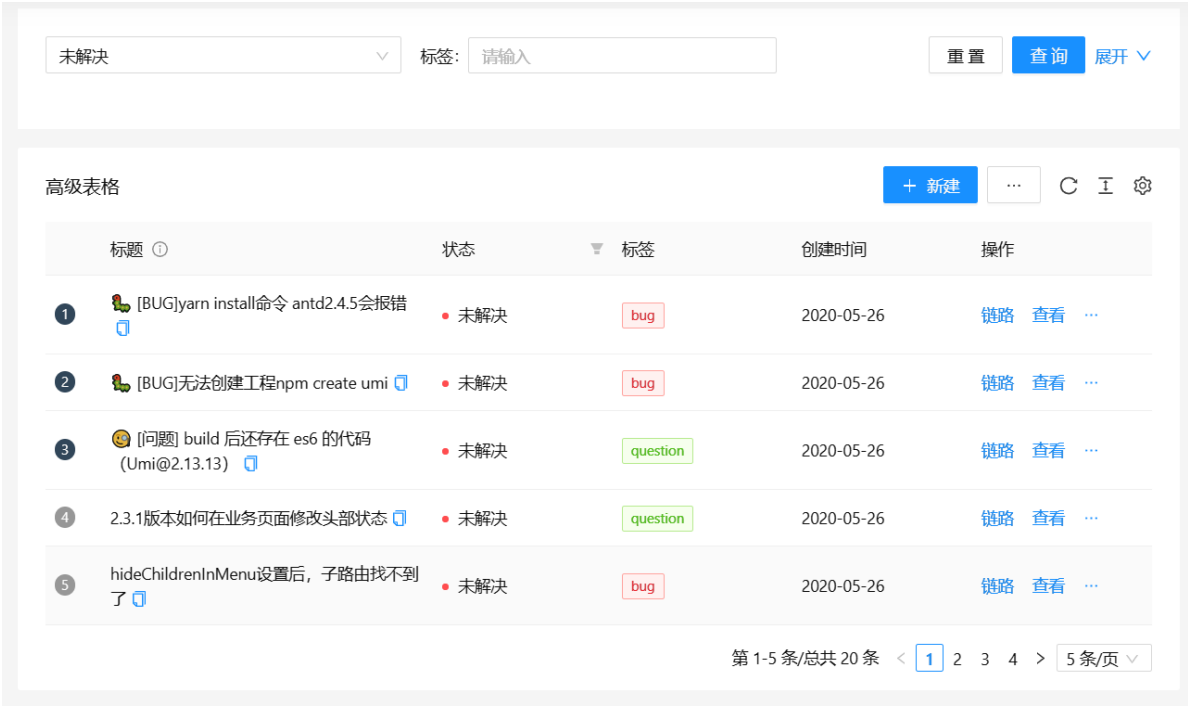
需要注意的是，出于上述多种原因的考虑，不论客户端是否指明自身需要分页数据，服务端都应当强制执行数据分页，这可以通过定义参数的默认值来实现。换言之，若客户端发起的请求并未指定`page`参数，则默认返回第1页数据。

分页启用后，每次请求返回的包体（以下如无特殊情况均使用JSON）格式也需要格外注意。我们可能认为，既然未分页前，每次均返回所有数据条目的列表，那么分页后只需返回一页的列表即可，这种设计思路是错误的。在分页的场景下，随之而来的一个问题是客户端如何判断有多少页数据可供获取？它不可能通过遍历`page`的取值来获得这一信息，这无论从客户端还是从服务端的角度都是不可接受的。我们注意到，总可用页数其实也是一种元数据，它是关于分页这一行为的。所以我们可以改进分页接口的返回数据格式，除了数据列表外，还返回关于分页的元数据，例如：

```
{
    "count": 1024,
    "result": [
        ...
    ]
}
```

上面的JSON中，`count`表示集合的大小，客户端只需将其除以每页大小并向上取整即可计算出可用的总页数，而所请求的页的数据则在`result`中。

客户端常常组合分页、过滤、排序功能来实现一个表格组件，这在各种后台管理应用中是非常常见的需求，如下图[14]所示。



如前所述，实际的分页/过滤/排序应当在服务端直接完成。那么，这样的表格组件应当如何与服务端交互呢？笔者认为，前端的表格组件实际上只需要负责处理与用户的交互，接受用户的输入（如过滤或页号选择等），然后将其转换为对应的查询参数，向服务端发起请求，随后展示服务端返回的一页数据即可。例如，在上图的表格中，表格只需要使用`count/5`计算出总共有4页数据，但实际上此时它只获取了一页数据，若用户点击页号2，则表格向后端发起`?page=2`的请求，获取并展示第2页数据。

表格组件催生了另一种数据需求，即所谓的**投影（Projection）**。现代应用中，一个对象常常有很多不同的属性，但当它在列表中进行展示时，我们通常并不需要显示它的所有属性。例如上图中表格只展示了每个问题的标题、状态、标签和创建时间，大量的其他数据并不需要在表格中展示，而只需要在每个条目的详情视图中展示即可，如问题的具体内容。这种**只获取对象的特定属性集合而非获取完整属性**的需求称为投影。为了实现这一需求，最简单的方法是服务端仍然发送对象的完整数据，由客户端挑选其关注的属性进行展示；但如此的代价是在网络中发送了大量不必要的数据，浪费网络带宽、降低了用户体验。因此我们可以通过查询参数定义一次请求所需要的属性集合，如上表中的`fields`参数。若请求指定了该参数，则返回的列表中每条数据只会包含指定的属性，其他属性则被过滤了。

语义：批处理（Batching）

在了解了REST语义后很容易想到的一个问题是，为什么POST请求一次只能创建一个资源？这个问题是值得探讨的。在任何请求/响应的网络模型中，一个很重要的参数是RTT（Round-Trip Time），即请求从客户端发送到服务端，与响应从服务端发送到客户端的时间之和；它并不包括服务端处理请求的时间。换言之，即使存在处理请求时间为0的服务端，一个请求也至少需要等待RTT的时间。显然，请求数

越多，那么RTT所带来的时间浪费就越多。理解了RTT的存在，我们就很容易理解REST所规定的POST只能创建单个资源的语义的局限性：如果我们需要创建多个资源，那么就需要等待多个RTT。

显然，降低RTT所带来的不必要的冗余的方法就是改进POST请求的语义，规定其请求包体中可以携带一个列表，其中每一个条目表示一个需要创建的新资源的数据；服务端根据该列表中的数据创建对应个数的新资源，随后返回新创建的资源的列表。此种设计称为**批处理（Batching）**。

语义/包体：错误处理（Error Handling）

在前面的叙述中，我们只描述了一切正常的情况下的各种行为和语义，但实际上，客户端与服务端交互的过程中可能产生大量的错误，包括但不限于：

- 客户端请求了不存在的资源
- 客户端请求了无权访问的资源
- 客户端指定的查询参数不正确
- 客户端提供的数据格式不正确
- 服务端程序执行过程中出错崩溃
- ……

一旦错误发生，在RESTful的实践中，常见的处理方法是忽略或告知客户端。有些错误是可以被忽略的，例如客户端指定了一个未定义的查询参数，那么服务端只需忽略即可。有些错误是服务端无法忽略的，例如客户端请求的资源不存在、POST的数据不正确等等。在后一种情况下，服务端必须以某种方式通知客户端错误已发生，它无法正常执行客户端的请求。

在描述HTTP协议时，我们了解到响应报文的首行包括HTTP状态码（status code），它指示了服务端处理请求的状态结果。HTTP标准规定了一系列状态码[15]，我们在此不过多讲述。但我们需要了解的是，HTTP状态码包含错误指示状态码，因此在出错时，我们可以通过设定特定的HTTP状态码来告知客户端本次请求处理过程中出现了问题。例如，HTTP标准规定404状态码表示服务端找不到所请求的资源，403代表客户端的权限不足，400代表请求本身的格式、数据等存在问题等。

REST的设计思路是尽可能利用现有的协议和标准，如HTTP和URL。因此，使用HTTP状态码作为错误指示，直到现在也仍然是一种相当流行的方案（需要注意，即使状态码出错，响应报文也可以返回包体数据，可以用包体数据提供关于错误的详情。）。但这种方案也是有局限性的，主要的问题在于在较为复杂的应用中，可能出现的错误情况较多也较为具体（与应用本身的数据有关），而HTTP状态码一方面数量过少，另一方面它所能表达的含义也过于宽泛。因此一种更加常见的实践是，使用一个通用的HTTP出错状态码指示出错（如400，不一律使用200是因为大部分客户端使用的工具对于异常状态码都会有特殊的处理，这可以让错误处理代码更容易编写，也更容易区分正常与异常请求），随后在包体中返回我们自行定义的状态码，例如：

```
{
    "status": 65536,
    "msg": "商品暂时缺货"
}
```

包体：输入输出异构

在讨论为何POST/PUT/PATCH请求的响应报文中需要返回受影响的资源的数据时，我们指出，部分数据如ID不应由客户端指定，而应当由服务端确定，随后服务端可以在响应报文中包含完整的数据，以供客户端了解完整的状态。笔者将这种客户端输入数据的格式/内容与服务端返回数据的格式/内容不一致的情况称为**输入输出异构**。

一种常见的异构来源于服务端生成的数据。典型的例子是ID、**时间审计（Time Auditing）**与**用户审计（User Auditing）**数据。时间审计在现代应用中是极为常见的需求。它表示，对于每个资源个体，我们需要在数据中保存该资源的创建时间（Creation Time）与上一次更新时间（Last Modification Time）。这两项数据可用于各种用途，例如统计和数据分析、权限控制、定时任务等。这两项数据必须

由服务端跟踪管理，而不能任由客户端指定，否则它们就将失去审计功能。但与ID相同，尽管它们不能由客户端指定，但却需要返回给客户端以供客户端读取使用，因此它们不能包含在输入中，但却需要包含在输出中。用户审计的实践更为复杂，但思想与实践审计类似。一种常见的情况是需要保存资源的创建者，例如一个订单的数据中需要保存哪位用户创建了该订单；那么这一数据显然只能由服务端生成，否则任何客户端都可以冒用其他用户的信息。

另一种常见的异构与关系模型有关。考虑这样的情况：现有学生和班级两个实体集，学生和班级为多对一关系。那么，如何存储一名学生所属的班级信息呢？在关系模型中，最常见的方法是在学生实体集中加入一条名为`class`的属性，其取值为学生所属的班级的ID。因此，如果直接将学生的数据使用JSON表示，可能得到如下的形式：

```
{
    "id": 1,
    "name": "Jack",
    "class": 1
}
```

这表示，该ID为1的学生属于ID为1的班级。由于这两个ID属于不同的集合，因此是可以重合的。这样的数据表示带来的问题是，如果客户端需要该学生的班级详细数据，它将需要再发起一个到`/class/1`的请求来获取所需的数据，这会增大客户端代码的复杂度与网络冗余。更好的解决方案是我们将当前所要序列化为JSON的对象的关联对象一并取出并序列化，如下所示：

```
{
    "id": 1,
    "name": "Jack",
    "class": {
        "id": 1,
        "school": "SEIEE",
        "name": "F1803700"
    }
}
```

这样的设计增强了数据对于客户端的可利用性，降低了客户端代码的复杂度，因此是一种**便于输出**的关系数据表示形式。

然而，这一形式并非没有缺点，现在假设这名学生需要更换班级，我们可以向`/students/1`发起一个PATCH请求，修改`class`属性。然而，我们马上会遇到一个问题：`class`属性的取值是什么？如果我们沿用上面的设计的话：

```
{
    "class": {
        "id": 2,
        "school": "SEIEE",
        "name": "F1802800"
    }
}
```

这一设计看似可行，实则语义非常模糊。考虑这样的情况，ID为2的班级实际的数据是：

```
{
    "id": 2,
    "school": "SE",
    "name": "F1801800"
}
```

那么，在为学生更新班级时，如果`class`中指定的班级数据与已有的该班级数据不一致的话，难道我们应当一并更新班级的数据吗？如果答案是应该，那么这显然会带来无数语义上和实践上的问题和错误。如果答案是不应该，那么我们为什么还要随着`class`传递这么详细的数据呢？它除了增大网络开销之外没有任何补益。

我们需要注意，在此处，我们更新的实际上是**关系**，而非**关系所关联的对象**，因此更合适的方案是指出新的关系对方的ID即可，而不需要也不应包含对方的详细数据，即：

```
{
    "class": 2
}
```

这样的形式在很多情况下并不适合数据输出，但相反的是，它非常适合用于输入关系数据，因此是一种**便于输入**的关系数据表示形式。因此我们在设计RESTful接口时，需要注意到可能的数据异构情况，并灵活运用数据异构来增强接口的易用性。

包体：其他编码

键值对模型可以较好地表达一般的数据，但现有的符合键值对模型的规范如JSON、XML等通常更多地关注文本数据的表达，而较难使用它们来传递一般的二进制数据。这在某些场景下不甚方便，例如客户端需要向服务端上传文件等。对于这样的需求，一种可以考虑的方案是使用Base64编码[16]，将二进制数据转换为文本数据，但其代价是数据的大小将固定增大1/3，这在上传较大的文件时是不可接受的。

对于上传文件的需求，较为理想的解决方案是使用其他的包体编码方法。目前应用最广泛的是`multipart/form-data`编码，具体规则可以参考这篇文章，它还介绍了其他的一些包体编码方法可以参考。它的优势在于它符合键值对模型，而又可以传输二进制数据，因此可以用于同时上传文件和指定文件的元数据。若需要使用该编码，需要注意客户端使用的网络请求库与服务端框架对该编码的支持情况。

安全：认证（Authentication）与授权（Authorization）

在任何多用户的系统中，权限都是一个永恒的话题。我们常常需要根据不同用户的身份等信息，限制其能进行的操作、所能看到的数据的范围等等。要完成这样的需求，我们必须完成两个连续的步骤，即**认证（Authentication）**与**授权（Authorization）**。这两个术语常常一同出现，但它们所指的是不同而又连续的两个范畴。

HTTP是一种无状态协议，换言之，对于两个不同的请求，如果不做额外的处理，服务端无法识别这两个请求是否由同一用户或客户端发出。显然，在这样的情况下，我们无法识别不同的用户，因为即使两个请求由同一用户发出，对于服务端而言也是完全不同的。因此，为了让服务端能够识别不同用户发出的连续请求，我们就需要在请求中带上一种凭据（Credential），它应当具有如下性质：

- 不同用户的凭据不同，以便根据凭据区分用户
- 只有服务端可以生成凭据，用户自身无法生成自己或其他用户的凭据
- 几乎不可能通过枚举的方式得到其他用户的凭据
- 凭据在用户登陆时通过校验用户名密码等方式生成，并可通过某种方式使其失效

签发凭据以及通过凭据识别用户的过程称为**认证（Authentication）**。常见的凭据形式包括Token、SessionID与JWT。不同的认证模式区别主要与这一凭据的形式、传递方式、更新方式有关，但共同点在于首次认证都需要用户的用户名和密码，或者其他相等的方式来核实用户身份，然后签发认证凭据。

Token认证和Session认证本质上是相同的。它们的思想都是在使用用户名密码核实用户身份后，生成一个随机性强、难以碰撞的长字符串作为后续请求的凭据。若后续请求中带有这个凭据，则可以认为请求的发起者就是先前登陆的用户。它们的区别在于凭据的存储和传递方式。在Token认证下，客户端应当自行保存Token，在随后的请求中通过GET参数或如下Header：

```
GET /endpoint
Authorization: Token <token_value>
```

来传递token。而Session认证方式是后端直接把SessionID作为Token，用户登陆后设置名为`SESSION`或其他类似名称的Cookie[17]，随后的请求中由浏览器自动发送相应的Cookie。相较于Token认证，Session认证可以免去开发者管理token的麻烦，且绝大部分后端框架都有默认支持，实现简单，但问题在于Session认证是基于Cookie的，而在配置不当的服务器上这可能会留下CSRF攻击漏洞。

Token认证和Session认证的主要缺点在于不论是Token还是SessionID的信息量都很少，或者说它们与随机字符串没有本质上的区别，因而后端在验证一个token是否有效时就必须引入数据库，通过数据库来判断token是否有效、属于哪个用户。这么做的结果是这两种认证方式的扩展性（scale）都很差，因为在集群环境下频繁的数据库访问将给数据库带来很大的压力，而且存在单点故障的风险。那么解决方案是很自然的，就是增加token中的信息量，把用户认证所必须的数据，如用户id、用户名等也编码到token中，再通过校验/加密机制防止恶意签发和修改，这就是JWT[18]的思想。限于篇幅原因，我们在此不赘述JWT的具体规范。

JWT认证传递token的方式与Token认证类似，也是通过特定的Header，如下所示：

```
GET /endpoint
Authorization: Bearer <jwt_value>
```

JWT认证的主要优势在于，由于Token中信息量大大增加，已经包含了后端中认证或者访问所需的用户数据，所以我们不再需要频繁查询数据库以获取用户信息，显著降低了数据库访问量，非常适合分布式环境。但JWT相较于Token或Session认证的主要缺点在于它无法有效地吊销。后者因为每次认证都需要查询数据库，那么只需要删除数据库中对应的条目，也就等价于token被吊销了，而JWT的出发点就是避免数据库访问，无法实现这样的吊销方式。一般情况下，JWT的唯一过期方式就是指定时间后到期，在没达到JWT中所规定的到期时间之前，哪怕用户已经登出，JWT依然有效。所以一个长期有效的JWT是非常危险的。

笔者在实践中常用的认证方案是Session+短有效期JWT相结合，以同时利用Session的实时性与JWT的高性能。其过程如下：

- 首次认证：通过用户名和密码认证，服务端指示浏览器设置`SESSION` Cookie，并返回JWT `accessToken`。服务端返回的`accessToken`有效期为10分钟。客户端应部署定时检查并刷新JWT的逻辑。

- 后续认证：客户端首先检查`accessToken`是否即将或已经过期，若是，首先尝试向服务端要求刷新`accessToken`。在刷新时，服务端需要使用Session来进行认证，成功则签发新的`accessToken`，否则说明session已过期或被注销，要求用户重新登录。刷新成功或`accessToken`未过期的情况下，使用`accessToken`发起实际请求。
- 对于安全性或实时性要求较高的接口，应当一律使用Session进行认证，甚至要求用户二次输入密码进行认证。

授权（Authorization）指的是根据认证步骤中识别到的用户信息，来判定用户权限的过程。在笔者的实践中，用户权限主要有两种表现形式：

- 路径权限，即对于特定路径，若用户无权访问该路径，则直接拒绝用户发起的访问该路径的请求
- 范围权限，即对于特定的集合，根据用户的权限不同，预先过滤该集合的数据，形成用户可访问的子集，随后的分页/过滤/排序/投影需求均在前述子集上进行

我们需要根据应用的实际需求使用如上两种权限。

前瞻：RESTful的局限性与新的接口范式

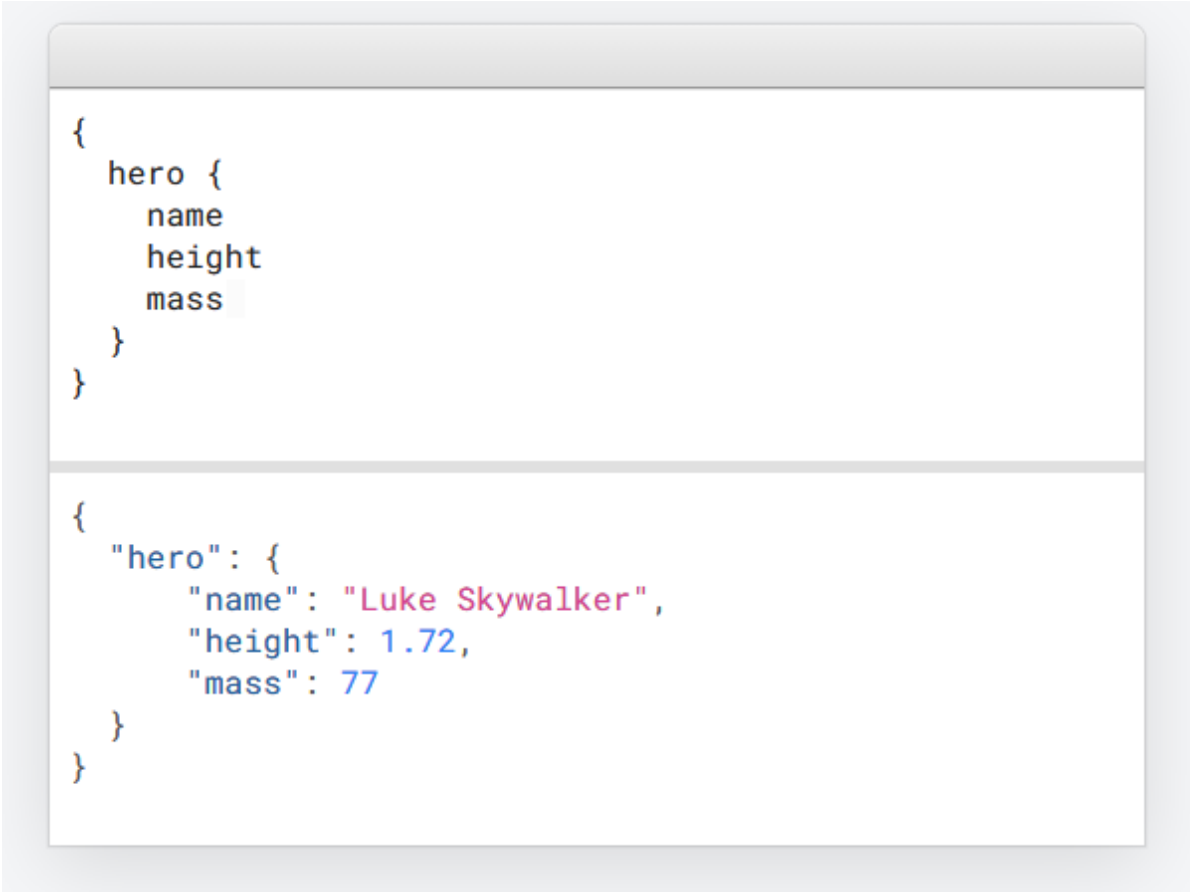
计算机系统亘古不变的问题是复杂性的控制。我们需要采取一系列的方法控制系统的复杂度，以降低开发和维护系统的心智成本、减少错误的产生。上述介绍的RESTful的各种实践，实际上都在不断地增加REST的复杂度，以通过复杂度的增加来换取更强的表达能力。但这样的方法很容易使系统的复杂度过高，直到我们无法承受。

根本的问题在于，REST所秉持的利用已有HTTP和URL标准的思想，在很大程度上已经无法适应现代应用的需求。虽然增加了以上种种复杂性，以期增强接口的表达能力，但仍然无法解决RESTful遇到的种种问题：

- 必须同时存在大量的接口路径
- 数据表示中存在各种无意义的枚举常量，必须通过额外的文档进行定义
- 一旦有新的数据需求，就往往需要定义新的接口

RESTful的另一根本问题在于，它在设计上就认为**主导数据需求的是服务端**，客户端应当根据服务端所提供的数据来开发应用。因此一旦客户端有新的数据需求，服务端往往就需要提供新的接口。这是REST模式无法解决的问题。

为了解决这些问题，最根本的办法是开发新的接口范式。由Facebook发明并首先应用的GraphQL[19]，目前正方兴未艾。它的核心思想是彻底颠覆RESTful中以服务端为主导的数据提供模式，而取代以**客户端描述其数据需求**，服务端执行客户端的数据需求的模式。GraphQL的核心贡献是设计了一套系统的数据查询语言（即QL，Query Language），以供客户端描述数据需求，而非囿于表达能力欠缺的路径与URL参数，例如：



上方为一个GraphQL查询，它描述了所需要的数据的“骨架”，而服务端则负责用实际的数据填充骨架并返回给客户端，即下方的JSON。

目前，RESTful仍然占据接口范式的主导地位，GraphQL离这一目标仍有很大的距离，主要有以下几点原因：

- RESTful更容易学习，时间成本相对较低
- RESTful长期以来积累了大量的经验、工具与库
- GraphQL的查询更复杂，因而对于服务端的计算性能要求较高，同时客户端很容易发起大量复杂的查询以瘫痪服务器
- GraphQL除了Facebook官方的Node.js实现外，其他实现或多或少存在性能/标准上的问题，库的支持程度不够理想
- 尽管GraphQL在设计上大大增强了查询数据的能力，但在GraphQL下，对数据的变更或写入变得更加麻烦；且GraphQL在设计上没有更好地解决分页/过滤/排序的需求

尽管如此，GraphQL仍然提供了非常诱人的表达能力与光明的前景，值得我们深入学习了解。

---

1. https://en.wikipedia.org/wiki/WebSocket ↩
2. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview ↩
3. https://tools.ietf.org/html/rfc2616 ↩
4. https://en.wikipedia.org/wiki/MIME ↩
5. HTTP标准的规定意在统一不同操作系统之间对换行符的规定，详见 https://zh.wikipedia.org/wiki/%E6%8F%9B%E8%A1%8C ↩

6. https://en.wikipedia.org/wiki/Representational_state_transfer ↩

7. https://stackoverflow.com/questions/34510/what-is-a-race-condition ↩

8. https://en.wikipedia.org/wiki/Relational_model ↩

9. https://en.wikipedia.org/wiki/JSON ↩

10. https://en.wikipedia.org/wiki/Universally_unique_identifier UUID有多个版本，各自面向不同的用途，详见：https://stackoverflow.com/questions/20342058/which-uuid-version-to-use ↩

11. https://en.wikipedia.org/wiki/Singleton_pattern ↩

12. https://en.wikipedia.org/wiki/Query_string ↩

13. https://en.wikipedia.org/wiki/URL#Syntax ↩

14. https://procomponents.ant.design/components/table ↩

15. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes ↩

16. https://en.wikipedia.org/wiki/Base64 ↩

17. https://en.wikipedia.org/wiki/HTTP_cookie ↩

18. https://www.ruanyifeng.com/blog/2018/07/json_web_token-tutorial.html ↩

19. https://graphql.org/ ↩