物联网中间件

第七章 RESTful Web Service

有关WSDL描述错误的是（）

- A）WSDL 指网络服务描述语言，用于描述网络服务
- B）WSDL 使用 XML 编写
- C）是一种 XML 文档
- D）WSDL 不能用于定位网络服务

下列哪些元素不属于WSDL命名空间（xmlns=http://schemas.xmlsoap.org/wsdl/）

- A）<definitions>
- B）<types>
- C）<message>
- D）<portType>
- E）<function>
- F）<binding>

参考材料

- 课本
  - 9.2，11，18
- **Architectural Styles and the Design of Network-based Software Architectures**
  - Roy Thomas Fielding's DISSERTATION
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
  - REST章节：Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)
- 网络上的材料

RESTful 架构

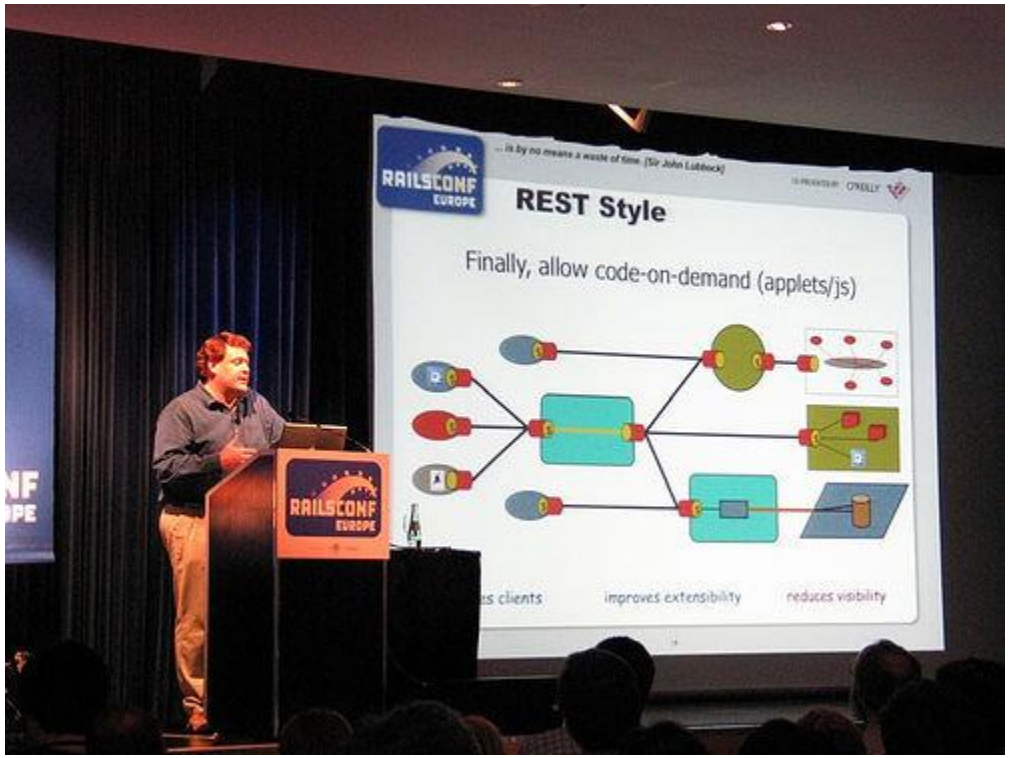
- RESTful 架构是目前流行的一种互联网应用架构。如果把网站，移动应用从服务器到前端，从整体上看作是一个软件，它就是一个层次清楚，功能强，扩展方便，适宜通信的架构规范。
- 在 REST 出现之前，程序间的网络通信架构采用的是远程过程调用（Remote Procedure Call，RPC），而后又在 RPC 基础上发展出来简单对象访问协议（Simple Object Access Protocol，SOAP），此后，出现了 REST。

- **REST 是一种面向资源的架构**，它能更好的**适应分布式下的系统架构设计**。对于开发者来说，越来越简单，越来越灵活。

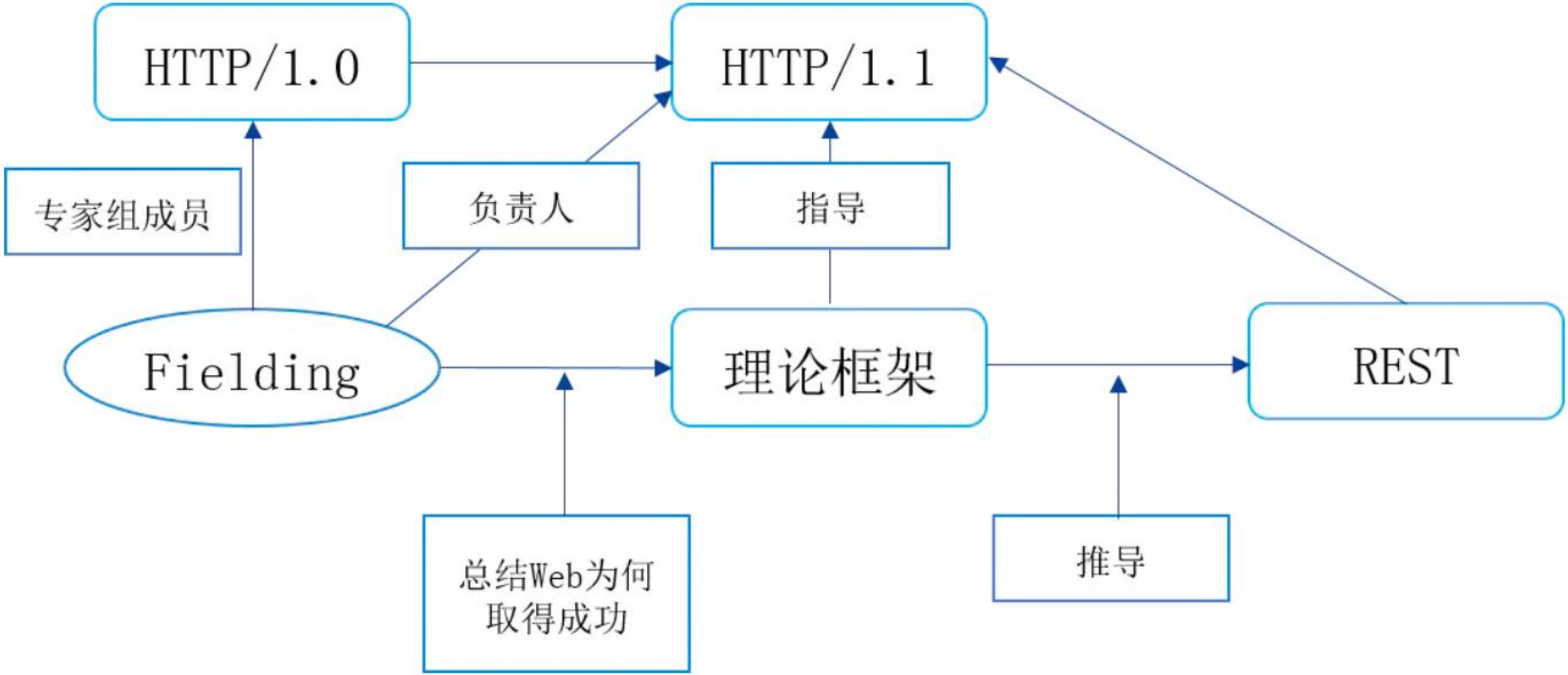
什么是 REST

- REST 是 “**Representational State Transfer**”的缩写，直译过来就是“**表述性状态转移**”
- 这个名词来源于 Roy Thomas Fielding 博士著名的论文《Architectural Styles and the Design of Network-based Software Architectures》(架构风格与基于网络的软件架构设计)。
- 该论文发表于2000年

Roy Thomas Fielding



- Roy Thomas Fielding在基于 REST 的约束上设计了 HTTP 协议。设计 REST 的目的，就是为了指导现代 Web 架构的设计与开发。他是 HTTP 协议（1.0 版和 1.1 版）的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席。


HTTP/1.0
HTTP/1.1
专家组成员
负责人
指导
Fielding
理论框架
REST
总结Web为何
取得成功
推导

REST由来

- REST -- REpresentational State Transfer
- 全称是 Resource Representational State Transfer：通俗来讲就是：资源在网络中以某种表现形式进行状态转移。
- 分解开来：
  - Resource：资源，即数据（前面说过网络的核心）。
  - Representational：某种表现形式，比如用JSON，XML，JPEG等；
  - State Transfer：状态变化。通过HTTP动词实现。

什么是资源（Resource）

- 任何能够被命名的信息都能够作为一个资源，它是对信息的核心抽象：一份文档、一张图片、一个与时间相关的服务(例如：“我现在城市的天气”)、一个包含其他资源的集合、一个非虚拟的对象(例如：用户)等等。
- 在设计具体 API 的时候，资源是业务系统里，抽象出来的一个业务对象。例如：用户(User)，订单(Order)，令牌(Token)等等。它允许随时间变化，输出不同值。
- 资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

表现层（Representation）

- "资源"是一种信息实体，它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。

- 比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

表现层（Representation）

- URI只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而URI应该只代表"资源"的位置。它的具体表现形式，应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

状态转化（State Transfer）

- 访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

- 互联网通信协议HTTP协议，是一个无状态协议。
- **如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。**

- 客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。
- 它们分别对应四种基本操作:
- **GET用来获取资源，**
- **POST用来新建资源（也可以用于更新资源），**
- **PUT用来更新资源，**
- **DELETE用来删除资源。**

REST由来

- 论文的第六章，作者解释了这个名词的由来：REST最初被 称作“HTTP 对象模型”，但是那个名称常常引起误解，使人们误以为它是一个 HTTP 服务器的实现模型。

- 这个名称“表述性状态转移”是有意唤起人们对于一个良好设计的 Web 应用如何运转的印象：一个由网页组成的网络(一个虚拟状态机)，用户通过选择链接(状态转移)在应用中前进，**导致下一个页面(代表应用的下一个状态)被转移给用户**，并且呈现给他们，以便他们来使用。

设计 REST 的动机

- "本文研究计算机科学两大前沿----软件和网络----的交叉点。长期以来，软件研究主要关注软件设计的分类、设计方法的演化，很少客观地评估不同的设计选择对系统行为的影响。而相反地，网络研究主要关注系统之间通信行为的细节、如何改进特定通信机制的表现，常常忽视了一个事实，那就是改变应用程序的互动风格比改变互动协议，对整体表现有更大的影响。**我这篇文章的写作目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。**"

- (This dissertation explores a junction on the frontiers of two research disciplines in computer science: software and networking. Software research has long been concerned with the categorization of software designs and the development of design methodologies, but has rarely been able to objectively evaluate the impact of various design choices on system behavior. Networking research, in contrast, is focused on the details of generic communication behavior between systems and improving the performance of particular communication techniques, often ignoring the fact that changing the interaction style of an application can have more impact on performance than the communication protocols used for that interaction. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture.)

设计 REST 的动机

- 第四章描述了设计 REST 的动机“为 Web 应该如何运转创建一种架构模型， 使之成为 Web 协议标准的指导框架”。
- 第五章从一个没有约束的空架构开始，不断的添加约束，从而使此架构进化为 Web 所需要的架构。

- 所以，**REST 是一组架构约束。**

- REST描述的是在网络中client和server的一种交互形式；REST本身不实用，实用的是如何设计 RESTful API（REST风格的网络接口）；

REST 约束

- **客户-服务器**
- **无状态**
- **缓存**
- **统一接口**
- **分层系统**
- **按需代码**
- 如果一个架构符合 REST约束，就称它为 RESTful 架构。

客户-服务器（Client-Server）

- 客户端、服务器分离

- 优点:
- 提高用户界面的便携性（操作简单）
- 提高可伸缩性（高性能，低成本）
- 允许组件分别优化（可以让服务端和客户端分别进行改进和优化）

无状态性

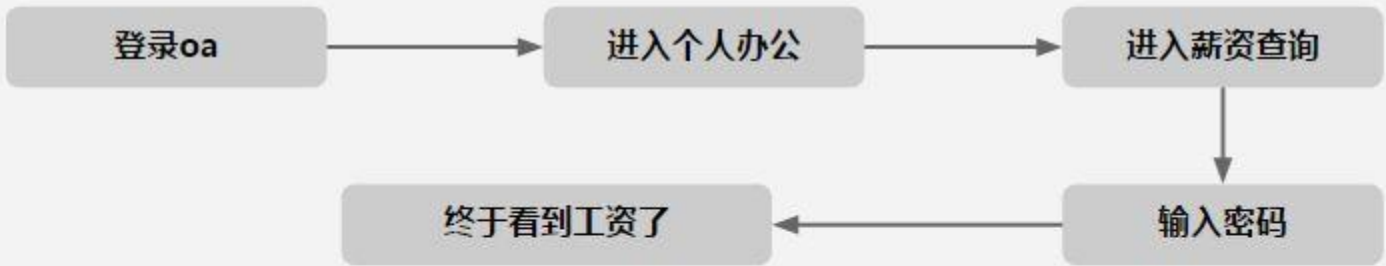
- 客户端与服务端的交互必须是无状态的，并在每一次请求中包含处理该请求所需的一切信息。服务端不需要在请求间保留应用状态

- 这种无状态通信原则，使得服务端能够理解独立的请求和响应。在多次请求中，同一客户端也不再需要依赖于同一服务器，**方便实现高可扩展和高可用性的服务端。**

有状态

- 无状态是相对于『有状态』而言的
- 我们平常接触到的网站，都是『有状态』的

如，在oa中查看基本工资：



后面的每一个状态，都依赖于前面的状态
没有一个url，能够直接定位到『张三』的『工资』

无状态

对每个资源的请求，都不依赖于其他资源或其他请求

每个资源，都是可寻址的，都有至少一个url能对其定位

- Application State
- Resource Stateless

RESTful 架构下，工资可以通过以下url查询：

张三工资 http://oa.company.com/salary/zhangsan
李四工资 http://oa.company.com/salary/lee4

无状态更加方便客户端使用服务器的资源或服务

- 有状态和无状态的区别，举个简单的例子说明一下。如查询员工的工资，如果查询工资是需要登录系统，进入查询工资的页面，执行相关操作后，获取工资的多少，则这种情况是有状态的，因为查询工资的每一步操作都依赖于前一步操作，只要前置操作不成功，后续操作就无法执行；如果输入一个url即可得到指定员工的工资，则这种情况是无状态的，因为获取工资不依赖于其他资源或状态，且这种情况下，员工工资是一个资源，由一个url与之对应，可以通过HTTP中的GET方法得到资源，这是典型的RESTful风格。

无状态（Stateless）

- 从客户端的每个请求要包含服务器所需要的所有信息

- 提高可见性（可以单独考虑每个请求）
- 提高了可靠性（更容易从局部故障中修复）
- 提高可扩展性（降低了服务器资源使用）

缓存（Cachable）

- 服务器返回信息必须被标记是否可以缓存，如果缓存，客户端可能会重用之前的信息发送请求。

- 优点:
- 减少交互次数
- 减少交互的平均延迟

统一的接口

- 这个才是REST架构的核心，统一的接口对于RESTful服务非常重要。客户端只需要关注实现接口就可以，接口的可读性加强，使用人员方便调用。

- 需要的付出的代价是，统一接口降低了效率，因为信息都使用标准化的形式来移交，而不能使用特定于应用的需求的形式。

分层系统（Layered System）

- 系统组件不需要知道与他交流组件之外的事情。封装服务，引入中间层。
- 优点： 限制了系统的复杂性
- 提高可扩展性

按需编码、可定制代码 Code-On-Demand（可选）

- 服务端可选择临时给客户端下发一些功能代码让客户端来执行，从而定制和扩展客户端的某些功能。
- 比如服务端可以返回一些 Javascript 代码让客户端执行，去实现某些特定的功能。
- 提示：REST架构中的设计准则中，只有按需编码为可选项。如果某个服务违反了其他任意一项准则，严格意思上不能称之为RESTful风格

- 优点： 提高可扩展性

HATEOAS

- Hypertext As The Engine Of Application State **应用状态引擎的超媒体**
- 它实际的意思是，**在资源的表述中，如果有其他资源的，则会提供相应的链接 URL，**使得用户不查文档，也知道如何获得相关的资源。
- Fielding 明确表示，**系统必须满足 HATEOAS 约束才能称为是符合 REST 风格的。**

HATEOAS

- **HATEOAS 会被称为状态引擎，因为它会引导状态的转移。**
- 在设计的理想状态中，使用 HATEOAS 的 REST 服务中，客户端可以通过服务器提供的资源的表达来智能地发现可以执行的操作。当服务器发生了变化时，客户端并不需要做出修改，因为资源的 URI 和其他信息都是动态发现的。
- **这样的设计，保证了客户端和服务器的实现之间是松耦合的。**客户端需要根据服务器提供的返回信息来了解所暴露的资源和对应的操作。当服务器端发生了变化时，如修改了资源的 URI，客户端不需要进行相应的修改。

- Github 的 API 就是这种设计，访问 http://api.github.com 会得到一个所有可用的 API 的信息列表，类似这样：
- {
- "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",
- ...
- }

REST成熟度模型

- Level 0：Web 服务只是使用 HTTP 作为传输方式，实际上只是远程方法调用（RPC）的一种具体形式。SOAP 和 XML-RPC 都属于此类。
- Level 1：Web 服务引入了资源的概念。每个资源有对应的标识符和表述。
- Level 2：Web 服务使用不同的 HTTP 方法来进行不同的操作，并且使用 HTTP 状态码来表示不同的结果。如：HTTP GET 方法来获取资源，HTTP DELETE 方法来删除资源。
- Level 3：Web 服务使用 HATEOAS。在资源的表述中包含了链接信息。客户端可以根据链接来发现可以执行的动作。

RESTful API的接口设计风格

- 基于“资源”
- 无状态
- URL语义清晰、明确
- URL中通常不出现动词，只有名词
- 使用HTTP的GET、POST、DELETE、PUT来表示对于资源的增删改查
- 使用JSON不使用XML

URI

- **一个资源具有一个或者多个标识。**这里说的标识就是统一资源标识符(Uniform Resource Identifier，URI)。统一资源定位符（Uniform Resource Locator，URL）则是 URI 的一种具体实现，是 URI 的子集。在通常的 API 设计中，直接使用 URL 来标示应用系统中的资源。
- 例如：
- https://www.sample.com/api/shops/10083 - 编号10083店铺的基本信息
- https://www.sample.com/api/users/2372/orders - 编号2372用户的所有订单信息

URL中通常不出现动词，只有名词

- URL的设计只要负责把资源通过合理方式暴露出来就可以了。
- 对资源的操作与它无关
- 操作是通过 HTTP动词来体现

- 所以REST 通过 URL暴露资源时，会强调不要在 URL中出现动词。

例子1

- 左边是错误的设计，而右边是正确的
- GET /rest/api/getDogs --> GET /rest/api/dogs 获取所有小狗
- GET /rest/api/addDogs --> POST /rest/api/dogs 添加一个小狗
- GET /rest/api/editDogs/:dog_id --> PUT /rest/api/dogs/:dog_id 修改一个小狗
- GET /rest/api/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一个小狗

例子1

- 左边是错误的设计，而右边是正确的
- GET /rest/api/getDogs --> GET /rest/api/dogs 获取所有小狗
- GET /rest/api/addDogs --> POST /rest/api/dogs 添加一个小狗
- GET /rest/api/editDogs/:dog_id --> PUT /rest/api/dogs/:dog_id 修改一个小狗
- GET /rest/api/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一个小狗
- 注意： :dog_id 是指代任意dog_id, 冒号是不安全字符不能出现在URL当中。复合URL语法的例子如下:
- DELETE /rest/api/dogs/1573 删除一个dog_id为1573小狗

- 左边的这种设计，很明显不符合REST风格，上面已经说了，URI只负责准确无误的暴露资源，而 getDogs/addDogs...已经包含了对资源的操作，这是不对的。相反右边却满足了，它的操作是使用标准的HTTP动词来体现。

- 左边的接口设计就沿袭了 RPC 的设计风格，表示 rest 服务提供了一系列方法，不符合 REST 规范
- 规范里建议的，URI中不要有动词。因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应该放在HTTP协议

例子2

- rest协议是面向资源的
- 假如要管理一些用户，那么将用户看作是一种资源：
- get /users/{userId} 获取userId对应的user信息
- post /users 创建一个新的user
- put /users/{userId} 更改userId对应的user信息
- delete /users/{userId} 删除userId对应的user

例子2

- rest协议是面向资源的
- 假如要管理一些用户，那么将用户看作是一种资源：
- get /users/{userId} 获取userId对应的user信息
- post /users 创建一个新的user
- put /users/{userId} 更改userId对应的user信息
- delete /users/{userId} 删除userId对应的user
- 注意：{userId}是指代任意复合URL语法的字符串，{}是不能直接出现在URL当中的。复合URL语法的例子如下：
- get /users/1573 获取userId 1573对应的user信息

- soap是面向服务的
- 还是管理用户，将对用户的操作看成服务:
- post /users/getUser
- post /users/creatUser
- post /users/updateUser
- post /users/deleteUser

例子3

- 如果某些动作是HTTP动词表示不了的。
- 比如网上汇款，从账户1向账户2汇款500元

- POST /accounts/1/transfer/500/to/2

例子3

- 你就应该把动作做成一种资源，正确的写法是把动词transfer改成名词transaction

- POST /transaction HTTP/1.1
  Host: 127.0.0.1

  from=1&to=2&amount=500.00

使用标准的 HTTP 方法

- RESTful API 使用标准的 HTTP 方法实现前后端的接口调用。对涉及到的资源，常用的操作就是增、删、改、查，类似对数据库记录的CRUD（Create，Read，Update，Delete)。使用的 HTTP 方法规则如下：
- **查询**GET： GET /users/{userId}
- **增加**POST： POST /users
- **全量修改**PUT： PUT /users/{userId} 即提供该用户的所有信息来修改

- **部分修改**PATCH：PATCH /users/{userId} 只提供需要的修改的信息
- **删除**DELETE：DELETE /users/{userId}
- 在修改的时，PUT 和 PATCH 区别在于 PUT 是全量修改，user 资源有多少信息，需要全部提供，而 PATCH 可以只修改手机或者邮箱，昵称，密码等信息。

- 将下列non-RESTful URL 转换为RESTful风格URL：
- GET /getAllUsers
-
- GET /createNewCompany
-
- GET /updateUserInfo
-
- GET /deleteUser?name=zhangsan

参考答案

- GET /Users
-
- POST /Company
-
- PUT /UserInfo
-
- DELETE /User?name=zhangsan
- DELETE /User/zhangsan

- http://example.com/api/user/delete/name=zhangsan //DELETE
- DELETE /users/{User?name=zhangsan}
- DELETE /User/{zhangsan}
-

URL语法

- https://www.ietf.org/rfc/rfc1738.txt

URL当中的参数

- Query parameter （查询参数）：
  - http://myexample.com/mystuff?paramA=2000&paramB=3000
- Matrix parameter（矩阵参数）：
  - http://myexample.com/mystuff;paramA=2000;paramB=3000
- Path parameter （路径参数）：
  - http://myexample.com/mystuff/2000/3000
- Basic path： http://myexample.com/mystuff 后面的都是参数

- HTTP 的方法中还有两个涉及到 REST：**HEAD 和 OPTIONS。**

- **HEAD 方法用于得到描述目标资源的元数据信息。**
- 例如，腾讯云的对象存储的API：HEAD Bucket 请求可以确认该存储是否存在，是否有权限访问。

- **OPTIONS 请求用来确定对某个资源必须具有怎样的约束。**
- 使用场景：客户端先使用 OPTIONS 询问服务端，应该采用怎样的 HTTP 方法以及自定义的请求报头，然后根据其约束发送真正的请求。

安全性和幂等性

- 在讨论 RESTful API 接口设计时，会提到两个基本的特性：**“安全性”和“幂等性”**。
- **安全性是指调用接口不对资源产生修改。**
- 幂等性是数学中的一个概念，表达的是N次变换与1次变换的结果相同
- **幂等性是指调用方法1次或N次对资源产生的影响结果都是相同的**
- 需要特别注意的是：这里幂等性指的是对资源产生的影响结果，而不是调用HTTP方法的返回结果。

常用 HTTP 方法的幂等性和安全性总结

-

http方法	安全	幂等
GET	是	是
POST	否	否
Head	是	是
PUT	否	是
DELETE	否	是
PATCH	否	否

- OPTONS、HEAD、GET 既是幂等也是安全的，不修改资源，多次调用对资源的影响是相同的。
- POST、PATCH 既不幂等也不安全，修改了资源，同时多次调用时，对资源影响是不同的，PATCH 的影响不同在于，每次的局部更新可能会导致资源不一样。
- PUT 是对资源的全量更新，多次更新总是对资源影响是一致的，所以它是幂等，但不安全。
- DELETE 用于删除资源，多次调用的情况下，都是删除了资源，所以它是幂等，但不安全。

- 保证 HEAD 和 GET 方法是安全的，不会对资源状态有所改变（污染）。

- 比如严格杜绝如下情况：
- GET /deleteProduct?id=1

为什么要在接口设计时，考虑幂等性?

- 在实际的业务流程场景下，我们可能会碰到下面一些问题:
- 订单创建接口，前端调用超时了，但服务端已经完成了订单的创建，然后前端显示失败，用户又点了一次。
- 用户完成了支付，服务端完成了扣钱操作，但前端超时了，用户不知道，又去支付了一次。
- 用户发起一笔转账业务，服务端已经完成了扣款，接口响应超时，调用方重试了一次。

- 以上类似的场景，需要在设计接口时考虑幂等性。我们可以借鉴微信支付的接口方案来实现这类场景需要的幂等性。在支付之前，需要调用一个接口生产预支付交易单，获得一个交易单号，随后再针对这个交易单号完成支付。**服务端确保一个交易单号只会被支付一次，这样就保证了支付过程的幂等性。**

状态码（Status Codes）

- 服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。
- 200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
- 201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
- 202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
- 204 NO CONTENT - [DELETE]：用户删除数据成功。

- 400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
- 401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
- 403 Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
- 404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
- 406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。

非REST架构软件（SOAP）和REST架构软件之间的的区别是啥呢？

- 其实区别就是在于对应的取资源的方式，SOAP用的是自己规定好的格式，而REST用的是大统一的URL格式。所以你想从SOAP软件中存取操作，必须事先知道它的读写格式；而REST，你只需要有一个浏览器就可以了。

REST风格的服务已经被广泛的普及和采纳，下面关于REST风格服务的描述，哪些是正确的：

- A）REST风格的服务，数据以HTTP协议交换，数据编码格式为JSON数据格式
- B）REST风格的服务，充分利用了HTTP的方法语义，实现服务的语义，如PUT, DELETE, GET, POST等
- C）REST风格的服务非常符合现代WEB应用的需要，能够实现有效的前后端分离模式
- D）REST风格的服务比SOAP-WebService更具有效率,并且具有更强大的语义表达能力
- E）REST服务建议把服务当作资源，每个资源都有唯一的URL标识，通过这个URL标识访问和控制REST服务

答疑

- 题目五：假设你是一个拥有1000辆车的自动驾驶出租车公司的技术负责人，你所负责的系统需要监视每个出租车的一切信息，包括位置、车速、乘客、目的地等等，系统还需要为出租车派发订单，规划路线等等。
- 这个系统就是一个大型的物联网。
- 这个系统中你应该选择什么样的中间件？为什么？（开放题）

- 开放题:
  - 分析问题的过程以及解决问题的方案复合逻辑
  - 没有明显的基础知识错误
  - 最好选择我们学习过的中间件

参考模板

- 该应用场景（该系统）有何特点：
  - 功能：功能复杂，系统支持的功能较多（系统需要监视每个出租车的一切信息，包括位置、车速、乘客、目的地等等，系统还需要为出租车派发订单，规划路线等等）
  - 计算能力：客户端（自动驾驶出租车）、服务器端都有比较强的计算能力
  - 网络环境：移动网络可能有一定的延迟，但是并不恶劣，不影响各种中间件的使用
  - 网络带宽、流量：应用场景中并没有限制
  - 实时性（网络延迟）：应用场景中并没有的要求
  - 是否需要支持异构系统：应用场景中并没有指定平台的类型，因此该系统中设备可能都属于同一平台，也可能分属于不同的系统

参考模板

- 根据应用场景的特点判断该选择哪种中间件:

- 功能：功能复杂，系统支持的功能较多（系统需要监视每个出租车的一切信息，包括位置、车速、乘客、目的地等等，系统还需要为出租车派发订单，规划路线等等）
  - 套接字只支持数据流的读写操作
  - RESTful Web Service 只支持HTTP动词，语义受限
  - RPC
  - SOAP

- 计算能力：客户端（自动驾驶出租车）、服务器端都有比较强的计算能力
  - 套接字
  - RPC
  - SOAP
  - RESTful

- 网络环境：移动网络可能有一定的延迟，但是并不恶劣，不影响各种中间件的使用
  - 套接字
  - RPC
  - SOAP
  - RESTful

- 网络带宽、流量：应用场景中并没有限制
  - 套接字
  - RPC
  - SOAP
  - RESTful

- 实时性（网络延迟）：应用场景中并没有的要求
  - 套接字
  - RPC
  - SOAP
  - RESTful

- 是否需要支持异构系统：应用场景中并没有指定平台的类型，因此该系统中设备可能都属于同一平台，也可能分属于不同的系统
- 同一平台：所有设备都采购自同一厂家
  - 套接字
  - RPC
  - SOAP
  - RESTful
- 不同平台：所有设备采购自不同厂家，未来还可能添加其他厂家的设备
  - 套接字（支持异构系统，但是开发调试很困难）
  - RPC（对异构系统的支持并不好）
  - SOAP
  - RESTful

注意

- 参考模板只是一种分析方式，不是具体的答案，也不是标准答案

- 开放题：
  - 分析问题以及解决问题的方案复合逻辑
  - 没有明显的基础知识错误
  - 最好选择我们学习过的中间件

不太好的答案

- 事务式中间件
- 消息中间件
- java Servlet
- Hadoop