软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2022,33(9):3271−3296 [doi: 10.13328/j.cnki.jos.006383]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# REST API 设计分析及实证研究*

周芯宇[1,2], 陈 伟[1,2], 吴国全[1,2,3], 魏 峻[1,2,3]

[1](中国科学院 软件研究所, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 陈伟, E-mail: wchen@otcaix.iscas.ac.cn

**摘 要**: REST API 已成为访问和使用 Web 服务的重要途径, 为开发基于服务架构的应用系统提供了可复用接口. 但是, REST API 的设计质量参差不齐, 因此有效、合理的设计指导规范对于规范和提高 REST API 设计质量具有现实意义和应用价值. 首先, 基于 REST API 的本质内涵, 建立了一个多维度、两层次的 REST API 设计指导规范分类体系 RADRC (REST API design rule catalog), 并对当前主流的 25 条设计指导规范进行分类. 其次, 针对已有规范提出相应的检测方法, 并实现了 REST API 设计指导规范遵循情况的分析与检测工具 RESTer. 最后, 使用 RESTer 开展 REST API 设计实证研究, 分析了 APIs.guru 收录的近 2 000 个真实 REST API 的文档, 从中分析提取相应的 REST API 信息, 进一步检测并统计当前 REST API 的设计特征和设计指导规范遵循情况. 研究发现不同应用类别的 REST API 在资源和操作模式上存在差异, 使得不同类别 REST API 在设计规则和总体架构方面各有特点. 实证研究结果有助于深入了解当前 REST API 及其设计规则的特征、现状和不足, 对于提高 REST API 设计质量和改进设计指导规范具有实际意义.

**关键词**: REST API; 设计指导规范; 分类体系; API 描述文档; 实证研究

**中图法分类号**: TP311

中文引用格式: 周芯宇, 陈伟, 吴国全, 魏峻. REST API设计分析及实证研究. 软件学报, 2022, 33(9): 3271–3296. http://www.jos.org.cn/1000-9825/6383.htm

英文引用格式: Zhou XY, Chen W, Wu GQ, Wei J. REST API Design Analysis and Empirical Study. Ruan Jian Xue Bao/Journal of Software, 2022, 33(9): 3271–3296 (in Chinese). http://www.jos.org.cn/1000-9825/6383.htm

## REST API Design Analysis and Empirical Study

ZHOU Xin-Yu[1,2], CHEN Wei[1,2], WU Guo-Quan[1,2,3], WEI Jun[1,2,3]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

**Abstract**: As an important way to access and use web services, REST API provides a technical means for developing and implementing service-oriented architecture-based application systems. However, REST API's design quality varies, so practical and reasonable design guidelines are essential for standardizing and improving REST API design quality. First of all, based on the connotation of REST API, a multi-dimensional, two-layered REST API design guideline classification framework REST API design rule catalog (RADRC) is established. Twenty-five popular design guidelines are classified based on RADRC. Secondly, a REST API design guideline compliance inspection tool, namely RESTer, is implemented. Finally, RESTer is employed to conduct an empirical study on current REST API design by analyzing nearly 2 000 real-world REST API documents from APIs.guru. RESTer analyzes the documents and extracts REST API design information for characterizing REST API design and inspecting compliance with the design guidelines. The empirical study finds

* 基金项目: 国家重点研发计划 (2017YFB1400602); 国家自然科学基金重点项目 (61732019); 并行与分布处理国防科技重点实验室基金一般项目 (61421102000402)

收稿时间: 2021-01-22; 修改时间: 2021-03-31; 采用时间: 2021-05-25; jos 在线出版时间: 2021-08-03

that REST APIs of different application categories vary in resources and operation modes, making different categories REST APIs have the characteristics in terms of design guidelines and overall architecture. The empirical study results help understand the characteristics, status quo, and shortcomings of current REST APIs and their adoptions of design guidelines, which is practically significant to improve REST API design quality and design guidelines.

**Key words**: REST API; design guideline; catalog; API document; empirical study

面向服务的体系架构 (service oriented architecture, SOA)[1], 特别是微服务 (microservice) 架构[2], 已成为互联网应用的新兴架构并逐步得到广泛应用. 基于微服务架构的应用系统由一组独立的、分布式的微服务组成, 微服务之间通过接口交互和协作, 实现 Web 应用系统的业务逻辑和功能. 表述性状态转移 REST (representational state transfer)[3]是一种软件架构设计风格, 在国内外信息技术和互联网行业的众多企业中得到广泛应用, 例如, 百度、阿里、腾讯、Google、Facebook 和 Twitter 等. REST 主要应用于 Web 服务及其 API (application programming interface) 设计, 基于 HTTP 协议、通过 HTTP 标准方法请求和操作网络资源实体; 它规范了 Web 服务接口调用方式, 使服务请求和响应更加有序和统一.

遵循 REST 架构风格的 API 即为 REST API. 当前, REST API 已成为访问和使用 Web 服务的主要方式, 也为开发和实现基于 Web 服务的应用系统提供了技术途径. 例如, ProgrammableWeb[4]作为最大的 Web API 目录服务, 收录的 API 从 2010 年的不到 2 000 个迅速增长到 2020 年的超过 23 000 个, 数量增长了 10 倍以上.

但是, REST 作为一种架构风格, 需要开发人员对 REST API 的设计自行做出某些决策, 由此导致 REST API 的设计质量参差不齐. 低质量的 REST API 设计会带来一些糟糕的后果, 例如, REST API 的可读性较差会导致其使用、扩展和维护困难, 无法吸引更多的系统开发人员集成和使用; REST API 的资源粒度设计不合理会导致服务不可重用和难以组合 (设计粒度过粗). 因此, 有效、合理的设计指导规范对于规范和提高 REST API 设计质量具有现实意义和应用价值.

当前 REST API 在设计质量及其分析评价方面的主要问题包括:

(1) REST API 的设计需要考虑资源划分粒度、HTTP 标准方法的使用以及其他方面的最佳实践 (best practice) 等. 尽管学术界和工业界针对 REST API 的设计已经提出了各种各样的指导原则 (principle)[3]、规则 (rule)[5]和设计 (反) 模式[6,7], 但是这些规范和原则分散, 缺少系统的组织和归纳, 没有形成较为系统和全面的 REST API 设计指导规范分类体系.

(2) 自动化的 REST API 设计分析和评价十分必要, 能够与 RESTful 应用的测试和质量评估等活动一起, 改进 Web 服务的开发质量. 因此, 如何实现自动化的方法, 从不同维度出发对 REST API 设计指导规范的符合程度进行检测和分析十分重要.

(3) REST API 的设计和实现与其所在的应用领域密切相关, 不同应用领域中 REST API 的设计实现可能会呈现出不同的特点. 分析和评价 REST API 的设计质量必须综合考虑其所属应用类型, 使分析和评价结果更有针对性.

针对上述问题, 本文首先对已有的 REST API 设计原则、规范和最佳实践进行梳理, 基于 REST API 的本质内涵, 建立了一个多维度、两层次的 REST API 设计指导规范分类体系, 即 RADRC (REST API design rule catalog). RADRC 包括资源、HTTP 交互、HTTP 消息和非功能属性 4 个大的维度, 每个维度又进一步细分为若干子类, 涉及该维度的不同方面. 通过归纳总结, RADRC 对 25 条设计指导规范进行归类, 这些指导规范从不同层次和粒度对 REST API 的设计给出指导性约束和建议.

接着, 本文针对 RADRC 中的设计指导规范提出检测方法, 分析和检测目标 REST API 对于设计指导规范的遵循情况 (如: 遵循了哪些规则, 遵循的比例等). 本文设计实现了一个基于 REST API 说明文档 (REST API document, RAD) 分析的设计指导规范检测工具——RESTer (REST API design evaluator). RESTer 分析目标 REST API 的 RAD 并构建语法树, 然后从语法树上的不同节点获取与设计指导规范相关的 REST API 的必要信息, 最后逐一检测当前 REST API 对于设计指导规范的遵循情况.

最后, 本文使用 RESTer 对近 2 000 个公共开放的真实 REST API RAD 进行实证研究, 这些 RAD 对应的 REST API 归属于 16 个应用类别的 64 个子类. 实证研究分析总结了当前互联网上的 REST API 在设计质量方面的现状和特

点, 尤其是不同应用类别的 REST API 在设计指导规范的遵循方面的差异. 实证研究发现: (1) 当前 REST API 在资源设计方面的规范化程度较高; (2) HTTP 方法的正确应用还有待进一步提高; (3) 云平台类、工具类和信息获取类 REST API 的特征与设计指导规范遵循情况差异较大. 本工作针对各个应用类别 REST API 的分析结果可以作为参考基准, 用于后续 REST API 设计的分析评估和对比.

本文工作的主要贡献如下.

(1) 构建了一个 REST API 设计指导规范分类体系 RADRC. RADRC 基于 REST API 的本质内涵, 从 4 个维度和多个子方面对当前主流的 25 条 REST API 设计指导规范进行分类, 并提出检测方法.

(2) 设计实现了 REST API 设计指导规范自动检测工具 RESTer. 工具基于 RAD 分析提取 REST API 信息, 实现 RADRC 中 23 条规范的逐一检测, 能够自动检测大规模 REST API 对于设计指导规范的遵循情况.

(3) REST API 设计实证研究, 分析了近 2000 个真实 REST API 的 RAD. 研究发现不同应用类别的 REST API 在资源和操作模式上存在差异, 使得不同类别 REST API 在设计规则和总体架构方面各有特点. 研究结果有助于深入了解当前 REST API 及其设计规则的特征、现状和不足, 对于提高 REST API 设计质量和研究改进设计规则具有实际意义.

本文第 1 节概要介绍 REST API 相关概念和背景知识. 第 2 节构建了 REST API 设计指导规范分类体系 RADRC. 第 3 节详细介绍基于 REST API 说明文档的设计指导规范检测方法 RESTer. 第 4 节使用 RESTer 对大规模 REST API 开展实证研究, 分析了不同应用类别 REST API 的设计特征以及对设计指导规范的遵循情况. 第 5 节讨论了论文工作的局限性. 第 6 节简要综述了 REST API 领域的相关研究, 第 7 节总结全文.

## 1 相关概念及背景知识

### 1.1 REST API

REST 由 Fielding 最早提出[3], 是一种基于 HTTP 协议的软件架构风格. HTTP 协议定义了 9 种标准方法 (如表 1 所示), 规定了客户端与服务器之间的信息交互方式. Fielding 发现对资源的 4 种常用操作方法 (增删改查, CRUD) 可以直接映射为 HTTP 协议的标准方法, 如图 1 所示. 由于与 HTTP 协议特性相吻合, REST 风格越来越多的应用于 Web 服务设计, 规范了 Web 服务接口调用方式, 使服务请求响应更加统一和有序.

表 1 HTTP 标准方法

| HTTP方法 | 描述 |
|---|---|
| GET | 请求指定的页面信息, 并返回响应实体 |
| POST | 向指定资源提交数据处理请求, 可能会导致新的资源的建立或已有资源的修改 |
| PUT | 从客户端向服务器传送的数据取代指定的文档的内容 |
| PATCH | 是对 PUT 方法的补充, 用来对已知资源进行局部更新 |
| DELETE | 请求服务器删除指定的资源 |
| CONNECT | HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器 |
| OPTIONS | 客户端查看服务器的允许请求方法以及性能描述信息 |
| TRACE | 回显服务器收到的请求, 主要用于测试或诊断 |
| HEAD | 类似于 GET 请求, 用于获取报头, 响应中没有具体的响应体内容 |

REST API 泛指通过 HTTP 协议进行通信、资源以网络实体的形式呈现, 并使用 HTTP 标准方法对资源进行操作的 Web 服务的 API. 图 2 是 REST API 的示意图, 服务器端管理的资源以网络实体的形式体现, 并以统一资源标识符 (unified resource identity, URI) 唯一标识, 因此资源与 REST API 中的路径一一对应. 例如, 图 2 中资源 A 用“/A”来唯一标识, 并且路径层级用来表示资源之间的层级关系, 如“/A/a”表示资源 A 的子资源 a. 客户端使用 HTTP 标准方法发起服务请求, 不同方法对应资源的不同操作, 每个操作称为一个端点 (end point), 图 2 中“/A/a”这一路径有 4 个端点. 在图 2 中, 客户端使用 GET 方法发起请求, 对应的是资源 a 的查找, 使用 DELETE 方法是删除资源 a.

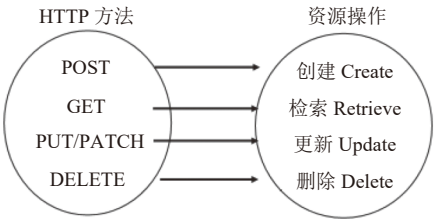



图 1 HTTP 方法与资源操作间的映射

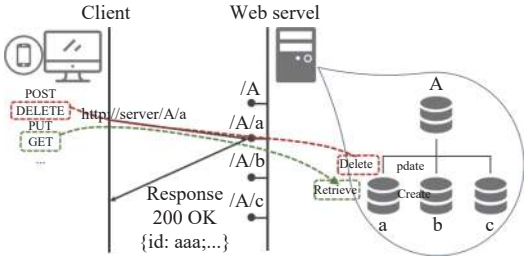



图 2 REST API 示意图

REST API 与传统 Web 服务接口最大的不同在于, 传统接口的设计以方法操作为核心, 即 URL 本身在一定程度上体现了操作方法的语义. 例如, “/getAByID?ID=23”“/deleteAByID?ID=23”等表示“基于 ID 获取/删除 ID 为 23 的 A 实体实例”; 而 REST API 以资源为核心, 服务设计侧重资源的呈现, 比如对于资源 a, 仅有“A/a”一个 URI 就足矣, 分别使用 HTTP 方法 GET 和 DELETE 对特定资源 a 的获取和删除.

### 1.2 REST 架构风格及成熟度模型

REST 是一种架构风格, 它包含了对于组件、连接器和数据的约束, 文献 [3] 对其的定义可概括为 6 条基本原则[3], 满足这些原则的架构被称之为符合 REST 架构风格.

(1) 客户端-服务器原则. REST 架构风格基于客户端-服务器架构, 背后的约束原则是关注点分离. 通过将用户界面功能 (user interface functionality) 移入客户端, 服务器端只做数据存储, 简化了服务器组件, 实现客户端和服务器端的解耦. 因此, 在保证通信接口 (即 REST API) 不变的前提下, 客户端与服务器端均能独立演进.

(2) 无状态原则. 客户端到服务器的所有请求都必须包含理解该请求的全部信息, 不能利用任何存储在服务器端的上下文, 会话状态全部保存在客户端而不是服务器端. 无状态给架构带来了 3 方面的好处: 可见性, 可靠性和可伸缩性.

(3) 缓存原则. 该原则要求请求响应中的数据被标记为可缓存或者不可缓存. 基于缓存机制, 客户端可以为相同请求重用响应数据. 缓存有助于减少交互, 降低平均延迟, 提高访问效率和性能.

(4) 系统分层架构原则. 该原则约束系统按层次组织, 每一层仅为上层提供服务并使用下层所提供的服务, 由此减少跨越多层的耦合, 提高可重用性.

(5) 代码可扩展性原则. 服务器能够向客户端传输可执行代码来临时扩展或自定义客户端功能, 例如 Java Applet 和客户端脚本等, 由此, 减少需要预先实现的功能数量, 简化客户端开发, 也能改善系统的可扩展性.

(6) 统一接口原则. 该原则是 REST 架构风格的核心特征, 也是 REST API 设计的最根本指导原则. 统一接口约束客户端和服务器之间的接口定义, 实现与接口背后服务的解耦, 增强独立演进能力. 具体包括:

① 基于资源, 使用 URI 作为资源标识符, 资源是到一组实体的概念性映射, 任何能够被命名的信息都能够作为一个资源;

② 通过表现层操纵资源, 当客户端持有资源的表示 (包括附加的任何元数据) 时, 在具有相应权限的前提下, 它有足够的信息来修改或删除服务器上的资源;

③ 自描述消息, 每个消息都包含足够的信息以描述如何处理该消息. 例如, 可以通过头文件 content-type (媒体类型) 来指定要调用的媒体类型 (media type) 解析器;

④ 作为应用程序状态引擎的超媒体 (hypermedia as the engine of application state, HATEOAS), 在资源的表达中包含了链接信息, 客户端可以根据链接来发现可以执行的下一步操作.

Richardson 成熟度模型[8]是对上述设计原则的具体化. 在 Richardson 成熟度模型中, (1) Level 0 级别并不是真正意义上的 REST API, 只是基于远程过程调用实现的 Web 服务, HTTP 仅作为传输层协议, 参数放在消息体中, 以 POST 方式提交到某个 URI 上的服务端点; (2) Level 1 是 REST API 的起始级别, 将服务抽象成资源, 每个资源都由唯一的 URI 单独标识, 而不再是将所有请求发送到单一公开服务端点. 请求的资源放在 URI 中, 而不是作为

参数放在 header 中, 但是仍然使用单一 HTTP 方法 (大多为 POST 方法) 进行通信; (3) Level 2 级别使用 HTTP 协议提供的几个标准方法对资源进行不同的操作, 比如使用 GET 方法进行资源获取操作、POST 方法进行资源创建; (4) Level 3 是对 HATEOAS 的实现, 它建立起资源以及资源操作间的联系, 提供给用户相关操作的链接.

REST 架构风格基本原则以及 Richardson 成熟度模型是 REST API 设计的核心准则和基本依据, 本文提出的基于 4 个维度的 REST API 设计指导规范分类体系的维度划分依据是 REST API 的基本设计原则, 同时具体规范内容总结整理自基本设计原则、成熟度模型以及已有的相关研究工作 (详见第 2 节).

### 1.3 REST API 说明文档及 OpenAPI 规范

传统 API 文档描述接口的功能、参数和返回类型等信息, 是接口的使用说明. 与之类似, REST API 说明文档同样需要准确、全面、清晰的描述一组 REST API 提供的所有请求方法以及服务端信息等, 具体内容应包括服务的基本介绍、服务器描述信息、附加的文档、所使用的安全验证方案等; 其中最为重要的是 REST API 的访问请求路径, 用来描述每个请求路径所提供的具体服务, 包括不同请求方法下对应的操作以及操作所需的属性, 请求响应的不同状态信息等.

OpenAPI specification (OAS) 是一个编程语言无关的 REST API 描述文档规范[9], 使人和计算机无需分析源代码或访问服务即可以发现和理解服务的功能. OAS 最早由 Swagger[10]在 2011 年提出, 2014 年发布了 Swagger 2.0 规范, 在 2015 年时 Swagger 将其贡献给了 OAI (OpenAPI initiative), Swagger 2.0 正式更名为 OAS 2.0. OAI 2017 年发布了 OAS 3.0, 截至当前, OAS 的最新版本为 3.1.0.

OAS 2 发展到 OAS 3, 在格式和内容方面都有所变化. 如图 3 所示, OAS 2 中的 schemes、host 和 basePath 分别用来描述通信协议、服务器信息和基础路径. 以“https://example.com/petstore-v1/users”为例, “https”表示使用安全 HTTP 协议, 服务器为“example.com”, 基础路径为“petstore-v1”. 上述信息在 OAS 3 中通过 servers 组件统一描述. OAS 2 中的属性 (parameters)、响应信息 (responses)、描述 (definitions) 和安全方案 (security definition) 在 OAS 3 中都被整合到组件 (components) 中. 同时, 该组件还包括示例 (examples)、请求体 (requestBodies)、头文件 (headers)、相关资源链接 (links) 和回调信息 (callbacks). links 是 OAS 3 新增部分, 用来遵循 HATEOAS 原则, 即在响应中包含当前请求相关资源相关操作的链接, 建立起服务间的联系, 以方便用户下一步操作; callbacks 也是新增内容, 描述用来执行回调操作的 URL. OAS 2 规范中的 consumes 和 produces 用来描述服务提供资源的表现形式. 可以看到, 无论是 OAS 2 还是 OAS 3, 路径 (paths) 都是核心内容, 包含了 API 提供服务的 URL 集合, 每个路径下包括不同的操作 (operation); 每个操作称为一个端点 (end point), 包括基本介绍、属性使用规则、响应状态集合等. 总体而言, OAS 3.0 规范将 OAS 2.0 中的信息进行了整合与完善, 去除掉一些被重复描述的组件, 并增加了一些能体现 RESTful 特性的组件, 如 links.

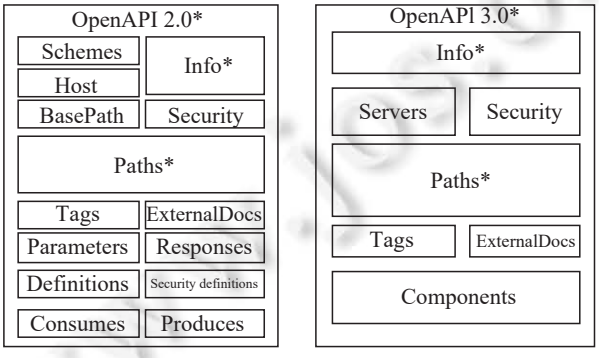



图 3 OAS 规范结构

除 OAS 外还有诸如基于 RAML[11]和 RSDL[12]的 REST API 描述文档, 但是 OAS 的结构清晰、表达的 API 信息丰富全面, 已经成为目前最主流的 RAD 描述规范, 遵循 OAS 的 RAD 也越来越多. 例如, APIs.guru[13]目前收录了 1754

个 REST API 以及 3 110 个遵循 OAS 的 REST API 说明文档, 其中 80% 符合 OAS 2, 其余遵循 OAS 3.

## 2 REST API 设计指导规范分类

REST API 本质上是以资源为核心、以 HTTP 方法为资源操作途径、以 HTTP 消息为数据载体的 Web 服务接口. 本文调研分析了当前 REST API 设计相关的研究工作, 以及工业界在 REST API 设计开发方面总结提出的最佳实践, 从资源 (A)、HTTP 交互 (B)、HTTP 消息 (C) 和非功能设计 (D) 4 个大的维度归纳整理设计规范, 构建了一个 REST API 设计指导规范分类体系 RADRC. 维度划分的依据来自 REST API 设计基本原则[3], 具体的, “统一接口原则”要求 REST API 的设计以资源为核心, 与 RADRC 中的资源设计 (A) 紧密相关; “统一接口原则”还要求 REST API 通过表现层操纵资源、遵循 HATEOAS, 以及 REST API 的消息是自描述的, 这分别与 RADRC 中的 HTTP 交互 (B) 和 HTTP 消息 (C) 两个维度紧密相关; 此外, “无状态原则”和“缓存原则”重点关注于 REST API 的非功能设计, 与 RADRC 的非功能设计 (D) 维度相关.

RADRC 的每个维度细分为若干子类, 涉及该维度的不同方面. 为了确保每一条指导规范的客观性以及典型性, 本文中 RADRC 每个维度上的每条指导规范均来源于对已有 REST API 的相关研究和文献书籍中规范的归纳整理. 由于 REST 在 2000 年被提出, 因此本文以 2000 年以后的研究成果作为文献检索范围, 选取了“REST” “RESTful”“规则 (rule)”“设计相关 (design, pattern, practice) ”“分析测试 (analysis, test) ”以及“综述 (review) ”等多个关键词, 将其组合构成文献检索的词组从学术搜索引擎以及 IEEE、ACM、Spinger 和 Elsevier 等文献数据库进行检测, 并人工对检索到的文献集合进一步分析过滤, 得到 17 篇与 REST API 设计相关的文献. 随后将文献中涉及到的规范进行归纳整理, 最终构建出本文的 REST API 设计指导规范分类体系中所包含的 25 条设计指导规范, 如表 2 所示, 每条设计规范的具体来源参照对应的参考文献.

### 2.1 资源设计

资源是 REST 架构的核心, REST API 的操作对象也是资源, 资源通过 URI 唯一标识. 资源设计规范进一步划分为 URI 格式 (A1)、URI 语义 (A2)、URI 路径 (A3) 和 URI 属性 (A4) 几个方面, 符合 REST API 提出之初的统一接口原则, 是其中基于资源原则的细粒度、具体化的规范.

(A1) URI 格式从资源命名的书写格式角度给出设计规范, 包括: 不以“/”结尾 (A1-1) 、使用连接符“-”而不是下划线“_”(A1-2) 作为连接符、全部使用小写字母 (A1-3) 等. 这些指导规范使 URI 的格式更加简洁明了, 可读性更高.

(A2) URI 语义主要从资源命名的语义相关性 (A2-1) 和层次关系 (A2-2) 两个方面给出设计规范. 资源是概念和实体等的模型, 一组 REST API 操作的资源应该属于相同或相近的语义范畴, 同时 URI 是以“/”表达层次结构, 因此以“/”分隔的相邻资源在语义上应具有层次关系. 这些规范使得 REST API 语义信息更加清晰合理.

(A3) URI 路径从规范路径包含的信息角度给出一些规范和最佳实践, 包括域名的设计 (A3-3, A3-4) 和资源操作相关的词语如何在路径中使用 (A3-1, A3-2).

(A4) URI 属性对于资源查询时查询属性的使用场景给出建议 (A4-1).

### 2.2 HTTP 交互

REST 架构基于 HTTP 协议和标准方法访问和操控资源, 是 REST 架构风格约束中统一接口原则的子原则之一. HTTP 交互维度的设计规范主要用以规范如何使用 HTTP 标准方法进行资源请求和操作, 以及如何使用响应状态码来正确表示服务器对于请求的响应结果.

(B1) 请求方法从 HTTP 标准方法使用的方面提出规范, 即必须使用 HTTP 协议定义的方法 (B1-1). 其次, 应该按照 HTTP 方法的语义正确使用, 不能仅用 GET 和 POST 方法实现所有对资源的 CRUD 操作 (B1-2), 这也与 Richardson 成熟度模型 Level 2 的定义相吻合.

(B2) 响应状态码规范了服务端对请求响应的状态表示, 总体原则应该遵循对标准状态码的使用 (B2-1), 便于对响应结果的理解和处理. 响应状态码的使用规范还可以进一步细分其使用前提和场景[5], 考虑到每个状态码的特定性, 因此不在本文建立的设计指导规范分类中展开.

表 2 REST API 设计指导规范分类

| 维度 | 方面 | 指导规范描述 |
|---|---|---|
| 资源设计 (A) | URI格式 (A1) | (1) URI不以“/”结尾[5,7,14,15]<br>(2) URI使用“-”而不是“_”作为连接符[5,7,14,15]<br>(3) URI路径全部使用小写[5,7,14,15]<br>(4) URI中不包含文件扩展名[5,7,14,15] |
| | URI语义 (A2) | (1) URI中的结点符合真实语义的上下文关系[7]<br>(2) URI使用“/”表示层次关系[5,7,15] |
| | URI路径 (A3) | (1)使用资源名词(动名词)为资源命名[5−7,14,15]<br>(2) CRUD方法名称不在URI中使用[5,7,14,15]<br>(3) API路径不包含“api”字样[5,15]<br>(4) API在顶级域名中加入“api”子域名[5,15] |
| | URI属性 (A4) | (1) 使用查询属性进行集合与存储类型资源的查询结果分页和过滤[5] |
| HTTP交互 (B) | 请求方法 (B1) | (1)使用HTTP协议定义的标准方法来对资源进行操作[5,14,15]<br>(2)正确使用GET和POST方法, 否则会导致对请求消息及其意图的错误表达, 破坏协议的透明性[5,6,14,15] |
| | 响应状态码 (B2) | (1)使用HTTP协议定义的响应码来标识响应状态[5,6,15] |
| HTTP消息 (C) | 消息头 (C1) | (1) 使用“Content-Type”声明请求和响应消息中的数据类型[5,6,15]<br>(2) 在HTTP头中使用“accept”支持媒体类型协商[5,6,14,15].<br>(3) 无状态原则, 请求使用“keys”或“tokens”等头文件进行用户身份验证, 响应不使用“set-cookie”“cookie”通过消息头传送状态信息[6] |
| | 消息体 (C2) | (1) 消息数据类型支持JSON格式, 同时以“application/json”作为消息头“Content-Type”的值来表明消息体格式[5,14,15]<br>(2) 请求对应的响应体中包含自链接, 以指示给定资源的关联和操作[5,6,14,15] |
| 非功能设计 (D) | 版本 (D1) | (1) 在HTTP头中声明选择的版本信息[7,16]<br>(2) 路径中避免出现版本(也有一些建议在顶级域名中描述版本信息)[15,16]<br>(3) 避免在查询参数中使用版本号[16,17] |
| | 安全 (D2) | (1) 使用OAuth安全协议用于保护资源[5,15] |
| | 缓存 (D3) | (1) 尽量使用缓存, “Cache-Control”“Expires” 和“Date”用于消息响应头, 以鼓励缓存[5,6,15]<br>(2) 响应头中使用“ETag”或“Last-Modified”来支持协商缓存[5] |

### 2.3 HTTP 消息

HTTP 消息是服务器和客户端交互的信息载体和表现形式. 该维度主要针对 REST API 统一接口原则中的消息自描述能力提出设计规范, 包括消息头 (C1) 和消息体 (C2) 两个子类, 使得每个消息都包含足够的信息以描述如何进行消息处理.

(C1) 消息头主要包含各种形式的元数据, 用来描述和规范 HTTP 请求和响应, 例如: 如何设置数据类型 (C1-1), 如何选择媒体类型 (C1-2), 以及如何正确使用消息头, 避免违背 REST 架构风格的无状态原则 (C1-3).

(C2) 消息体包含了请求和响应的具体内容, 如请求的方法和属性, 响应的资源表示, 以及下一步的资源操作链接等. 总体而言, 消息格式推荐支持 JSON 格式 (C2-1); 其次, 最为重要的是请求对应的响应体中应该包含链接, 以指示给定资源的关联和操作 (C2-2), 这与成熟度模型的 Level 3 对应, 是 HATEOAS 的具体表现.

### 2.4 非功能设计

非功能设计维度涵盖了除上述维度之外其他方面的设计规范和最佳实践, 本文建立的设计指导规范分类体系目前包含了版本 (D1)、安全 (D2) 和缓存 (D3) 这 3 个方面的内容.

(D1) REST API 随着服务的演化也在不断演进, 可能存在多个版本. 设计指导规范推荐通过 HTTP 消息头声

明的方式来确定对应的 API 版本 (D1-1), 以及避免在 URI 路径中出现版本信息 (D1-2).

(D2) 很多 REST API 面向特定的客户端和用户开放, 因此需要设计安全机制以避免私有信息和重要信息的泄露, 可以使用 OAuth (open authorization) 来提供安全机制 (D2-1).

(D3) 为了提高性能, REST API 支持设置缓存机制在客户端缓存响应信息, 这也是对 REST 架构风格可缓存原则的支持和实现. 缓存主要通过设置 HTTP 响应头得以实现 (D3-1, D3-2).

RADRC 包括了主流的、在已有工作中经常出现的 REST API 设计规范和最佳实践. 尽管 RADRC 的 4 个维度覆盖了 REST API 的内涵和根本原则, 但它更多的是一个分类框架, 而不是一个完备的规则目录. 一方面, REST API 的设计规则和最佳实践随着技术和应用的发展不断变化, 并非一成不变; 另一方面本文工作主要针对 REST API 设计在技术方面的规则进行总结归纳, 其他涉及与 API 治理及供给等其他方面的规则与实践[18], 并不在本文讨论范围之内. 此外, RADRC 中的指导规范并非强制的、必须遵循的, 除了与 REST 架构风格基本原则相关的必要约束外, 更多是以推荐和最佳实践的方式存在, 而且不同的设计规划在不同的应用类别和领域中的应用也不尽相同, 这可以从本文后续的 REST API 设计实证研究中看到 (详见第 4 节).

## 3 REST API 设计检测方法

RADRC 中的设计规范一方面可以指导 REST API 的设计, 另一方面可以检测目标 REST API 的设计是否遵循这些规范. 本文对 RADRC 中的指导规范提出相应的检测方法. 例如, RADRC 中规范 A1-2 的检测方法可以通过分析 URI 字符串实现, 如果目标 REST API 的某个资源 $r$ 的 URI 包含下划线“_”, 则认为当前 API 的资源 $r$ 的 URI 格式设计违背了规范 A1-2; 反之, 如果不存在“_”或者使用了“-”, 则认为遵循 A1-2.

需要说明的是, 不同设计规范需要不同检测和判断依据, 主要存在 3 类情况. (1) 有些规范 (如 A1-2) 通过静态分析 REST API 的相关信息即可判断是 (或否) 遵循设计规范. (2) 此外, 还有一些规范需要结合具体的应用上下文信息才能做出判断. 例如, 在不结合上下文信息的前提下, 对于规范 A3-1 的判断仅限于是否使用名词或动词对资源命名, 而难以判断命名是否合适. (3) 必须依赖于动态访问和响应结果分析, 例如, 在没有访问 REST 服务并获得真实响应的前提下, 通过静态分析, 对于 B2-1 规范的判断仅限于是否使用了标准的状态响应码, 而对于响应码使用的语义是否正确 (比如使用“404”表示一个服务器端错误) 则很难判断.

本文主要通过静态分析 REST API 相关元素实现设计指导规范遵循情况的检测, 其中对于需要结合上下文或动态访问的规范检测, 一方面主要从语法方面判断是否采用了规范建议的内容, 而不检测具体的语义和实际的动态响应. 例如, 对于 B2-1 的判断只检测 RAD 中是包含了正确的响应状态码, 而不检测状态码的具体使用情况. 本文设计实现了基于 RAD 静态分析的 REST API 设计指导规范检测工具 RESTer. 该工具依据表 2 中每条规范的检测依据实现了对 25 条规范中 23 条的分析检测.

具体的, 第 3.1 节介绍 RESTer 的工作流程以及各模块之间的关系, 第 3.2 节介绍具体的规范检测方法实现, 第 3.3 节对本文方法进行正确性以及效率的验证.

### 3.1 REST API 设计指导规范检测工具: RESTer

RESTer 检测 API 的流程如图 4 所示, 以满足 OAS 的 RAD 为基本输入, 主要由 REST API 文档解析模块和设计指导规范分析模块两部分组成. RESTer 首先对输入的 RAD 进行初步格式校验, 以确认其符合 OAS 2 或 3 规范.

接着, 符合 OAS 规范的 RAD 将进入文档解析模块. 该模块基于 swagger validator-badge[19]实现, swagger-parse 将说明文档解析为 JSON 结构, swagger-model 根据 OAS 规范定义了 REST API 的语法树层级结构以及各结点的内容格式. 本文使用以上两个库对说明文档解析, 包括词法检查以及语义分析; 随后创建组件结点, 并将一些引用组件链接到实际组件结点上, 从而构建出 REST API 语法树. 文档解析模块的作用在于进一步基于 OAS 规范对说明文档进行验证, 为后续分析评价模块构建了准确可用的 REST API 语法树, 并从语法树中提取 API 的类别组件作为类别信息, 为之后生成对比基准以及实现对 REST API 的针对性提供依据.

分析评价模块设计实现了表 3 所列的各条规范的检测算法, 从 REST API 语法树中提取相关组件结点 (表 3 中的检测对象), 逐条检测当前 REST API 对设计指导规范的实现情况, 具体检测方法实现在第 3.2 节进行介绍. 随

后根据 REST API 对规则的实现情况对其进行评价, 考虑到不同类别的 API 有自己的领域特征, 实现情况可能有较大差异, 使用统一的评价方式可能会有失偏颇, 因此本文根据 API 的类别信息统计了不同类别 REST API 对于各规范的平均遵循度, 作为该类别 API 的评价基准 (领域平均实现情况, 在第 4.3 节对领域实现特性进一步分析), 通过对比分析得到被测 API 在其所属应用类别内的 RESTful 程度, 以实现针对性的 RESTful 评价.

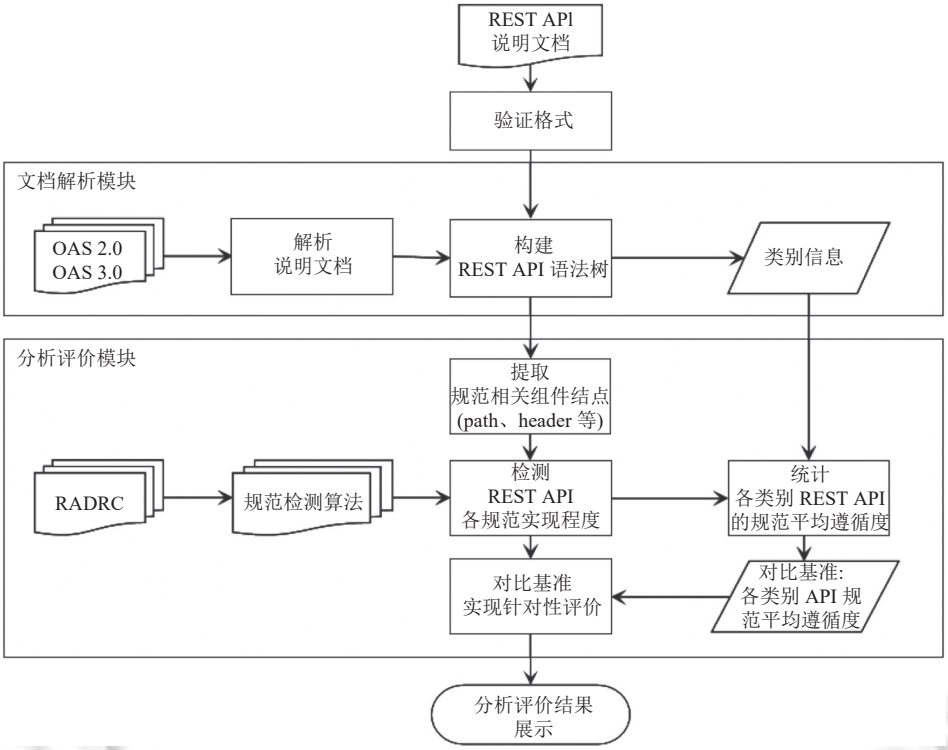



图 4 RESTer 工作流程

此外, RESTer 还分析和统计了目标 REST API 的基本信息, 包括: API 名称、所属应用类别、RAD 所遵循的 OAS 版本以及统计提供的资源数, 其中资源数以路径数和端点数作为度量值.

### 3.2 规范检测方法实现

表 3 是 RADRC 中各条规范的判断依据以及算法实现, 使用的方法包括基于字符串匹配的方法、语义分析和统计等, 对于 REST API 的检测是基于上述几种方法及其组合实现的.

● 基于字符串匹配的方法

部分 REST API 设计指导规范的检测可以通过检查目标对象是否包括特定的字符或字符串来实现. 例如, 检测是否遵循规范 A3-2 需要判断路径中不出现执行 CRUD 语义的动词. 为了达到这一目标我们预先构建一个具有 CRUD 语义的动词集合 $V$={get, create, add, …}. 针对某一 REST API $R$ 的某一路径 $p$, RESTer 首先根据“/”划分 $p$ 的路径层级, 去除路径属性部分; 然后对每一个层级进行分词处理, 得到一组词集合 $W_p$. 另一方面, 基于 $W_p$ 和 $V$, RESTer 检测两者交集是否为空, 如果 $W_p \cap V = \emptyset$ 则认为当前路径 $p$ 遵循 A3-2, 否则违背该规范.

● 基于统计的方法

以规范 A2-2 为例, 它要求 REST API 使用“/”对资源进行层级划分来表现资源的结构设计情况, 但是应用类别和领域的不同决定了 REST API 的资源设计具有各自特征, 无法采用统一的标准进行度量和判断. 鉴于此, RESTer 统计并计算每个 API 的平均路径层级数, 例如“school/student/{name}”的层级数为 3. 进一步的, RESTer 以每个应用类别中所有 REST API 的总体平均层级数作为基准, 通过与该应用类别的基准对比来评价被测 REST

API 对于该规范的实现情况.

● 基于语义分析的方法

本文基于 WordNET (https://wordnet.princeton.edu/) 和 Stanford's CoreNLP (https://stanfordnlp.github.io/CoreNLP/) 实现语义分析. Stanford's CoreNLP 是流行的英文自然语言库, 提供分词、词性标注、词法分析等功能, 本文使用该库获得 URI 中各结点 (按照层级划分) 的词根 (lemma) 以及词性, 以此标记各结点. WordNET 是一个被广泛使用的英文词汇数据库, 它将不同词性 (名词、动词和形容词等) 的词映射到一个或多个词义上, 建立词索引; 并为词义相近的词建立同义词集 (synsets), 词集之间使用语义指针表示彼此的关系, 比如上下位词 (hypernym-hyponym)、整体和部分 (holonym-meronym) 等关系, 以此来判断 URI 结点间是否存在语义关联性.

表 3 REST API 设计指导规范检测方法

| 规范编号 | 基本方法 | 检测对象 | 检测方法 |
|---|---|---|---|
| A1-1 | 字符串匹配 | path | 路径字符串尾部字符是否为"/" |
| A1-2 | 字符串匹配 | path | 路径字符串是否包含"_", 不包含"-" |
| A1-3 | 字符串匹配 | path | 将路径字符串转化为小写后与原字符串进行一致性对比 |
| A1-4 | 字符串匹配 | path | 路径字符串不包括文件后缀名子串, 如".json, .html, .js, .php, .xml, .gif, .jpg, .txt, .png, .java, .jsp, .asp" |
| A2-1 | 语义分析 | path | 以"/"将路径字符串分隔为多个结点(去除属性值), 判断相邻两个结点间的语义关联 |
| A2-2 | 字符串匹配+统计 | path | 检测和统计路径中"/"的出现次数 |
| A3-2 | 字符串匹配 | path | 对路径字符串进行CRUD方法名及相似语义的关键词检索 |
| A3-3 | 字符串匹配 | servers, host, basepath | 对检测对象进行子串"api"的存在检测 |
| A3-4 | 字符串匹配 | path, servers, host, basepath | 检测是否存在子串"api" |
| A4-1 | 字符串匹配 | parameter | 针对属性中的查询属性(即in:query)进行关键词模糊匹配, 关键词列表为"limit, offset, page, range, page size, page start index, before, after" |
| B1-1 | 字符串匹配+统计 | operation | 判断是否存在非HTTP标准方法的动词, 统计使用的各类HTTP方法 |
| B1-2 | 字符串匹配+统计 | path, operation | 对违反了A3-2规范的路径进一步检测, 根据违规动词的语义信息是否于HTTP标准方法的语义相同, 判断其HTTP方法使用的正确性 |
| B2-1 | 统计 | operation | 是否有响应状态的描述, 统计状态码使用频数及各类状态码的使用率 |
| C1-1 | 字符串匹配 | header | 检测响应中的头文件(header)是否包含"content-type"头文件 |
| C1-2 | 字符串匹配 | parameter | 检测属性中的头文件属性(即in:header)是否包含"accept"属性 |
| C1-3 | 字符串匹配 | parameter | 对属性名进行关键词"key, token, authorization"模糊匹配 |
| C2-2 | 字符串匹配 | responseSchema, links | 检测响应体模板描述中是否有包含"link"的属性(OAS2), 检测响应中是否包含"links"元素(OAS3) |
| D1-1 | 字符串匹配 | parameter | 检测属性中的头文件属性(即in:header)是否包含"version"属性 |
| D1-2 | 正则表达式匹配 | path | 正则表达式匹配"v(ers?\|ersion)?[0-9.]+ ", 其中语义版本号的正则表达式为"v(ers?\|ersion)?[0-9.]+(-?(alpha\|beta\|rc)([0-9.]+\+?[0-9]?\|[0-9]?))?" |
| D1-3 | 字符串匹配 | parameter | 检测属性中的查询属性(即in:query)是否包含"version"属性 |
| D2-1 | 字符串匹配+统计 | securityDefinitions, securityScheme | 检测是否使用该组件对安全方案进行描述, 并统计使用了何种安全方案 |
| D3-1 | 字符串匹配 | header | 检测响应头文件中是否包含"Cache-Control", "Expires" 或"Date" |
| D3-2 | 字符串匹配 | header | 检测响应头文件中是否包含"ETag"或"Last-Modified" |

### 3.3 规范检测方法验证

● 正确性验证

为了验证规则检测方法的正确性, 本文对实验数据集 (第 4.1 节) 进行随机采样, 抽取了 10% 的 RAD (即 181 个) 并对其中所描述的 REST API 对设计规范的遵循情况进行人工检测, 然后与 RESTer 的检测结果进行对比.

本文使用准确率和召回率两个指标对 RESTer 进行评价, 计算公式如下:

$$Accuracy = \frac{TP+TN}{TP+TN+FN+FP} \tag{1}$$

$$Recall = \frac{TP}{TP+FN} \tag{2}$$

其中, *TP* 为真实符合某一规范并且 RESTer 检测结果也符合的路径/说明文档数; *FN* 为真实符合某一规范而检测结果为不符合的路径/说明文档数; *FP* 为真实不符合某一规范而检测结果为符合的路径/说明文档数; *TN* 为真实不符合某一规范并且检测结果也不符合的路径/说明文档数. 需要说明的是, 表 3 中规范检测对象是针对路径 (path) 的规范检测算法 (比如 A1-1、A1-2 等), 本文进行验证时从路径级别进行验证, 因为一个 RAD 包含一个及以上路径; 其余规范 (比如 A3-3、D2-1 等) 采用说明文档级别进行验证. 验证结果如表 4 所示, 可以看到我们的规范检测方法具备较高准确性, 个别规范检测方法的召回率偏低, 比如规范 A2-1 检测路径结点间是否具有符合真实上下级语义的层级关系, 由于部分 API 提供的资源抽象程度较低, 有些使用了专有名词以及组合名词为资源命名, 检测方法很难实现全覆盖.

表 4 规范检测方法正确性验证

| 规范编号 | A1 | | | | A2 | | A3 | | | A4 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 1 | 2 | 2 | 3 | 4 | 1 |
| 准确率 | 1.00 | 1.00 | 1.00 | 0.98 | 0.82 | 1.00 | 0.88 | 0.96 | 0.98 | 0.99 |
| 召回率 | 1.00 | 1.00 | 1.00 | 1.00 | 0.43 | 1.00 | 0.91 | 0.95 | 1.00 | 0.98 |
| 规范编号 | C1 | | | C2 | D1 | | | D2 | D3 | |
| | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 1 | 1 | 2 |
| 准确率 | 1.00 | 1.00 | 0.94 | 0.92 | 1.00 | 0.95 | 1.00 | 1.00 | 1.00 | 1.00 |
| 召回率 | 1.00 | 1.00 | 0.88 | 0.78 | 1.00 | 0.96 | 1.00 | 1.00 | 1.00 | 1.00 |

- 时间开销

同时, 本文对 RESTer 的检测效率也进行了验证 (Win10, Intel i7, 16 GB 内存环境下), 平均每个 RAD 的检测时间为 2.3 s, 其中对路径的语义检测较为耗时, 平均为 0.13 s/路径.

## 4 REST API 设计实证研究

本文对真实、公开的大规模 REST API 的设计特征进行实证研究, 以期深入了解当前 REST API 的设计现状, 分析归纳 REST API 对设计指导规范遵循情况. 本文分应用类别统计了的 REST API 设计情况, 其目的在于研究分析不同应用类别对于 REST API 设计指导规范的遵循和应用是否存在差异, 以及这些差异是否由业务特征决定的; 另一方面, 分应用类别的大量 REST API 的设计研究结果可以作为对比基准, 为该行业 REST API 设计提供参考依据, 用于指导后续 REST API 的设计.

本文实证研究工作将回答以下研究问题.

问题 1 (RQ1). 当前互联网上大量实际 REST API 的总体设计特征, 以及对于 REST API 设计指导规范的遵循程度如何?

问题 2 (RQ2). 不同领域和应用类别的 API 的 RESTful 程度如何, 是否与应用领域的特征相关, 以及导致这些特征的原因是什么?

为了回答上述问题, 本文以 APIs.guru 上的近 2 000 个 REST API RAD 为目标对象, 使用 RESTer 逐一检测, 最后对检测结果进行统计和分析.

### 4.1 数据集

本文实验数据来自 APIs.guru 收集的开放 REST API RAD, 均遵循 OAS 2 或 OAS 3. 截至 2020 年 5 月, APIs.guru 共收录了 3 108 个 REST API RAD, 其中 609 个遵循 OpenAPI 3, 剩余 2 499 个遵循 OAS 2.

实验数据集的预处理包括: (1) 过滤掉数据集中同一 REST API 的多个版本 RAD, 只保留同一 API 的最新版本 RAD, 得到 1 940 个 RAD (562 个遵循 OAS 3, 1 378 个遵循 OAS 2); (2) 随后, 我们对这 1 940 个 RAD 进一步检查, 移除其中没有资源信息的 RAD, 共计 122 个 (121 个遵循 OAS 2, 1 个遵循 OAS 3). 最终, 用于本文实证研究的 REST API RAD 共计 1 818 个.

数据集中大部分 RAD (1 665 个) 都包含了 REST API 所属应用类别信息. 例如, Github.com 为 collaboration 类, azure.com 为 cloud 类. 剩余没提供类别信息的 RAD, 我们从 ProgramableWeb[4]上获取类别信息. ProgramableWeb 收录了数量更多的包括 REST API 在内的多种类型的 Web API, 但是缺少相应的 API 说明文档. 经统计, 1 818 个 REST API 文档分别属于 64 个类别, 包括: cloud、data、security、collaboration 等, 其中 cloud 类别数量最多, 包含 1 200 个 RAD, 主要来自 4 个最主流的公有云平台, 即 googleapi、amazonaws、azure、microsoft. 但是这些云平台提供的 REST API 并不都是 cloud 类, 比如 googleapi 平台的 analytics API 的类别是 analytics, reseller API 的类别是 enterprise, 而 identitytoolkit API 的类别是 security 等.

由于有些类别下的 API 数量较少, 为了使我们的实验验证结果具备一定的概括性同时不丢失类别特征信息, 我们首先根据类别和功能的相似性, 将 64 个类别合并为 16 个大类, 随后, 根据每个大类的具体规则检测结果进一步抽象, 分析出更具概括性的应用相关的 REST API 的设计特征 (第 4.3 节). 这 16 个大类的具体划分信息如表 5 所示, 其中子类信息列中括号内的数表示该类别包含的 REST API 数, 并以其中包含 RAD 数量最多的类别名称作为该大类的代表进行命名. 例如, ipass、iot、backend 和 cloud 均为平台类型, 且 cloud 类别包含的 RAD 数量最多, 因此将上述类别合并为一个大类, 仍以 cloud 作为类别名称. 需要说明的是, 由于有的 REST API 属于多个类别, 例如, Github 既属于 collaboration 类和 developer tools 类, 所以各个类别的 REST API 总数之和大于 1 818.

表 5 REST API 文档分类统计

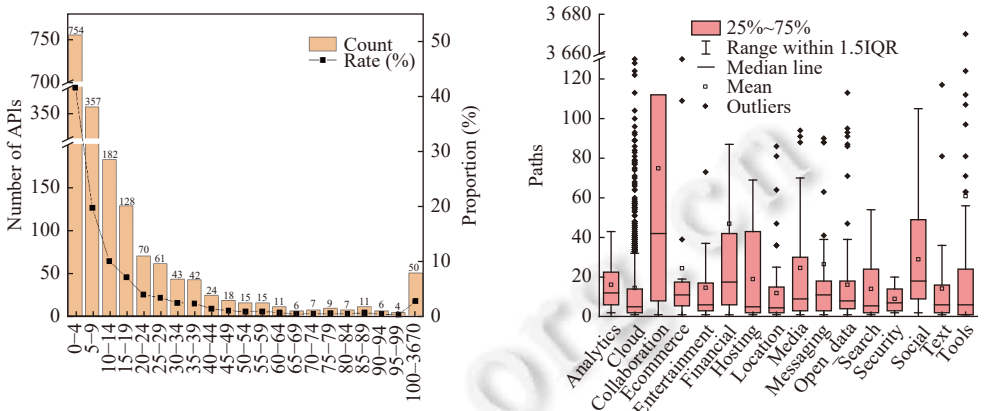
| 编号 | 大类名称 | 子类信息 | RAD数量 |
|---|---|---|---|
| 1 | analytics | extraction (1), machine learning (10), analytics (13) | 24 |
| 2 | hosting | accounts (1), domain (1), hosting (5) | 7 |
| 3 | search | 3d (1), healthcare (1), charity (1), medical (1), food (1), government (3), genetics (3), education (5), search (14) | 30 |
| 4 | security | fitness (1), verification (2), monitoring (3), security (14) | 20 |
| 5 | messaging | chat (1), email (11), telecom (11), messaging (16) | 39 |
| 6 | collaboration | jobs (3), enterprise (10), collaboration (12) | 25 |
| 7 | social | customer relationship management (1), customer relation (3), social (21) | 24 |
| 8 | text | reporting (1), documents (1), transcription (2), text (31) | 35 |
| 9 | location | time management (3), transport (27), location (32) | 62 |
| 10 | finance | finance (34) | 34 |
| 11 | ecommerce | payment (2), business (4), payment (13), marketing (14), ecommerce (42) | 75 |
| 12 | entertainment | sports (1), entertainment (60) | 61 |
| 13 | media | media (63) | 63 |
| 14 | tools | application (1), advertising (2), support (2), application development (3), management (1), web site management (1), project management (1), tools (33), developer tools (80) | 124 |
| 15 | open data | data (4), database (6), storage (10), open data (93) | 113 |
| 16 | cloud | ipass (2), iot (10), backend (16), cloud (1200) | 1 228 |

### 4.2 REST API 设计指导规范遵循情况

#### 4.2.1 资源设计

REST API 涉及资源的数量在一定程度上反映了 REST API 功能的丰富程度和复杂度. 本文基于 REST API 包含的 URI(路径) 数统计所有目标 REST API 的资源总体分布情况. 本文将 REST API 包含的资源数划分为不同区间, 0–100 之间的区间长度为 5, 100 以上为单独一个区间, 得到 21 个分布区间, 即 [0, 4][5, 9][10, 14]…[100, +∞), 然后统计目标 REST API 分布在各个区间中的数量.

统计结果如图 5(a) 所示, REST API 的数量分布随着包含资源数量的增加总体呈下降趋势. 可以看到拥有 5 个以下资源的 API 有 754 个, 占比最多 (41.45%), 其中只有 1 个路径的 API 占 17.7%; 其次是包含 5–9 个资源和 10–14 个资源的 REST API 数量, 分别占 19.63% 和 10%. 其中的一个例外是包含 100 个及以上资源的 REST API 数量为 50 个, 占总数的 2.69%, 排在第 7 位, 超过了拥有 30–99 个资源的 REST API 的数量.




(a) REST API 资源数 (b) 各类别 API 资源分布图

图 5 REST API 资源统计分布

本文统计了所有 REST API 的资源数, 平均为 19.9, 中位数为 6, 且大多数 (78%) REST API 的资源路径数小于平均值. 进一步的, 我们根据第 4.1 节中的应用类别统计了每个类别 REST API 拥有的资源, 如图 5(b) 所示, 平均包含资源最多的类别是 collaboration 类, 数量为 74.9, 并且该类的整体资源数分布跨度明显大于其他类. Collaboration 类别包含了以 Github、Bitbucket 为代表的在线代码仓库和开发环境、Trello 协作工具等服务, 此类 REST API 提供了包括对用户、代码仓库、提交、评论等资源的操作, 涉及的资源数量和类型众多, 因此属于该类别的 REST API 平均包含的资源最多. 其余各类别 API 的资源中位数均在 20 以下, 资源平均数均超过中位数, 这是由于各类几乎都存在少数资源数特别多 (离群点) 的 API, 比如 tools 类的资源中位数为 6, 其平均数达到了 61, 这是由于该类中有个别 API 包含了大量资源, 例如, 其中最多的 Microsoft-graph API 的资源数有 3 670 个.

● URI 格式

本文检测分析了规范 A1-1 到 A1-4 的遵循情况. 考虑到每个 REST API $R$ 通常有多于一条路径, 即 $path(R) \geqslant 1$. 我们首先使用 RESTer 检测 $R$ 的每条路径是否遵循某条规范 (如 A1-1), 然后统计 $R$ 对该规范整体遵循情况, 即 $C(R, rule)=n/N$, 其中 $n$ 为遵循规范的路径数, $N$ 为 $R$ 的路径总数. 因此, $C(R, rule)$ 的取值范围是 [0, 1], 例如, 某 REST API 有 10 条路径, 其中 8 条遵循了规范 A1-1, 那么该 API 对于规范 A1-1 的整体遵循情况为 80%.

目标 REST API 对于 URI 格式设计指导规范的遵循情况的统计结果如图 6 所示. 大多数 API 都遵循 URI 格式设计规范 A1-1, A1-2 和 A1-4. A1-3 规范“路径命名应该小写”的遵循情况存在较大差异, 与其他 3 条规范相比, 只有 387/1818 (21.29%) 个 REST API 完全遵守 A1-3, 而大部分 (1082/1818, 59.52%) REST API 完全没有遵循该规范. 进一步分析发现违背 A1-3 规范的 API 路径多数表达了较复杂的操作语义, 难以简单映射到 HTTP 协议的标准方法, 由此采用传统软件编码的“驼峰式”命名格式, 把 REST API 作为类似传统的代码函数加以命名, 例如, “exportEvaluatedExamples”“authorizedCertificates”, 而符合 REST API 设计指导规范的命名格式应为“export-evaluated-examples”“authorized-certificates”.

在 URI 格式规范中, REST API 对于 A1-4 (URI 中应该不包含文件扩展名) 的遵循程度最好, 总体上有 97.69% 的 API 实现了该规范. 进一步对少数违背该规范的 REST API 进行分析, 统计其出现的后缀文件名发现, 出现的后缀绝大多数都是声明文件为 JSON 格式 (587 个路径), 其他极少数的格式包括 html (6 个)、js (5 个)、xml (4 个)、

php (4 个)、gif (3 个)、jpg (2 个) 和 txt (1 个), 这也在一定程度说明出 JSON 目前已成为网络资源的主流数据格式. 但是, 对于资源格式的声明不应该出现在路径命名中, 而是应该在 HTTP 消息头文件的 content-type 中设置.

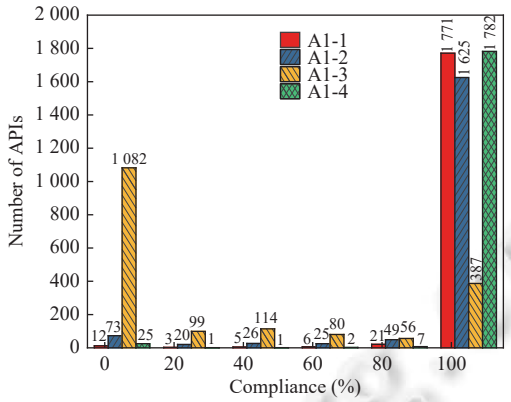



图 6 URI 格式相关指导规范遵循情况

● URI 语义

REST API 的 URI 路径使用“/”划分层级来表示资源的结构, 路径语义呈现服务提供的具体内容. 本文统计整体及各个应用类别的 API 的路径平均层级数, 计算方法如式 (3) 所示: $h_{ij}$ 和 $p_i$ 分别表示第 $i$ 个 API 的第 $j$ 个 URI 路径的层级数和路径总数, $n$ 表示 REST API 总数 (总体或分类别).

$$AvgH(all/domain)=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{p_i}\sum_{j=1}^{p_i}h_{ij}\right) \tag{3}$$

统计结果如图 7(a) 所示, 所有 API 的平均层级数为 4.8. 从所属应用类别来看, cloud 类 REST API 的平均层级数最高 (6.27), 这是由于云计算服务平台提供的服务业务逻辑较复杂, 比如 Azure 的平均层级数为 9.5, 它包括的路径命名如“/subscriptions/{subscriptionId}/providers/Microsoft.ApiManagement/service”和“/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ApiManagement/service/{serviceName}/certificates/{certificateId}”, 导致资源 URI 的结构层级高. 除此之外, 其余类别的 API 路径平均层级数大多不超过 3 层. 我们可以使用该结果作为 REST API 资源层级设计的参考依据, 对于非云计算平台 (cloud) 类的 API, 其资源层级数应该不超过 3 较为合理.

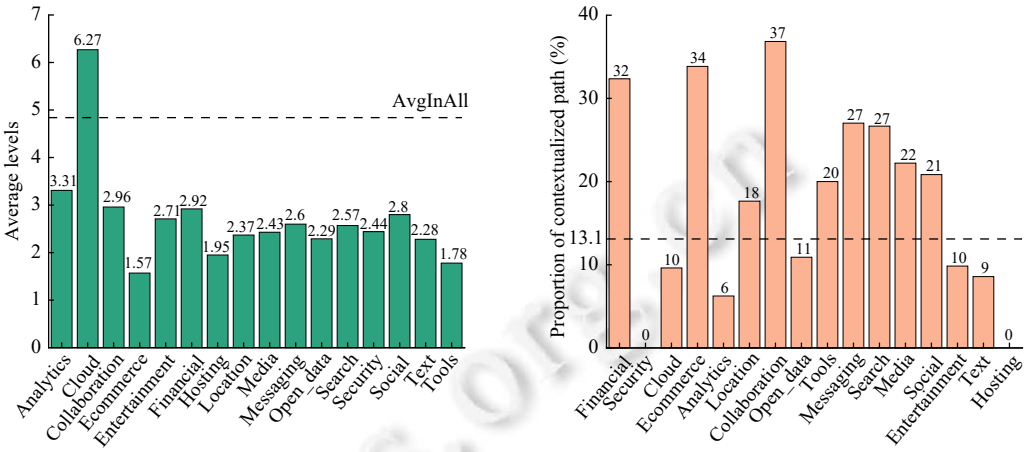
对于 REST API 的路径层级间的语义关系 (A2-1), 本文使用 Stanford coreNLP 和 WordNet 检测路径中是否存在符合真实语义的上下位关系, 利用 WordNet 中近义词集 (Sysnet) 间的上下位关系 (hypernym-hyponym) 和成员关系 (part, member, substance), 构建 4 层下位词义集树, 从而对层级间的语义关系进行检测, API 中有路径的层级间存在这种上下位关系, 则认为该 API 存在具有上下文语义关系的路径 (Contextualized Path), 对各领域的 API 均进行检测, 检测结果如图 7(b) 所示. 经统计 13.1% 的 REST API 检测到了语义路径, 比如 zuora.com (https://www.zuora.com/products/billing/)“管理工具”中的路径“/v1/transactions/invoices/accounts/{account-key}”, “/v1/transactions/payments/accounts/{account-key}”等均具有符合真实语义的上下级关系, 比如“transaction/invoices”, “transaction/payments”. REST API 的路径设计应该符合真实的语义关系, 对提高 API 的可读性具有重要意义, 使 API 更容易被理解以及组合重用.

● URI 路径

REST API 以资源为核心, 且其中大部分资源应以名词命名. 本文首先统计采用名词进行 URI 资源命名的情况, 结果如图 8(a) 所示. 可以看到, 各应用类别中, 平均 80% 以上的 API 使用名词进行资源命名, 完全使用名词进行资源命名的 API 占比均在 50% 以上.

本文进一步分析未使用名词命名资源的 REST API 及路径, 统计 URI 中出现的动词词频. 如图 8(b) 所示, 横

坐标表示出现纵坐标所列动词的 URI 路径数, 图中所列动词表达的资源操作语义都可以映射到 HTTP 协议实现的请求方法中. 例如, HTTP GET 方法可以实现 URI 路径中包含“get”“read”等动词的操作语义, POST 方法可以实现“create”“add”“new”等表达的资源操作语义, 使用 DELETE 方法能够实现“delete”“remove”等. 由此可见, 这些 API 没有很好的遵循设计指导规范, 仍然延续了以操作为核心的传统网络服务风格.




(a) 平均层级数分布 (b) 语义路径实现情况分布

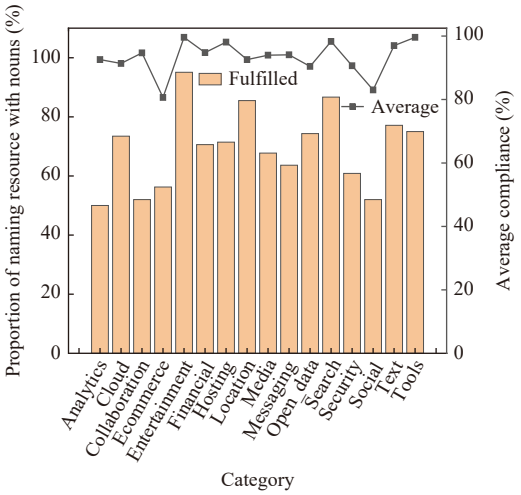
图 7 URI 资源层级分布统计

资源命名的规范违背分为两种情况: (1) 正确使用了 HTTP 方法, 但是在 URI 的命名中使用了多余的动词, 例如 amazonaws.com 的 image builder 服务 (https://aws.amazon.com/image-builder) 的 URI“/DeleteImagePipeline”、“/UpdateImagePipeline”分别使用 DELETE 方法和 PUT 方法发起请求, 只需去掉 URI 中的操作动词“delete”和“update”即符合 REST 设计规范 (A3-2); (2) 既没有正确使用 HTTP 方法, 在 URI 中也使用了违规的动词, 以 Google compute API (https://cloud.google.com/compute/docs/reference/rest/v1) 为例, “xxx/addAssociation”(使用 POST 方法), “xxx/getAssociation”(使用 GET 方法), “xxx/removeAssociation”(使用 POST 方法), 不符合设计规范 (A3-2, B1-1), 可以设计为一个以资源 association 为核心的 URI“xxx/association”, 分别使用 HTTP 的 POST、GET、DELETE 方法映射到上述 3 个接口的实现逻辑.

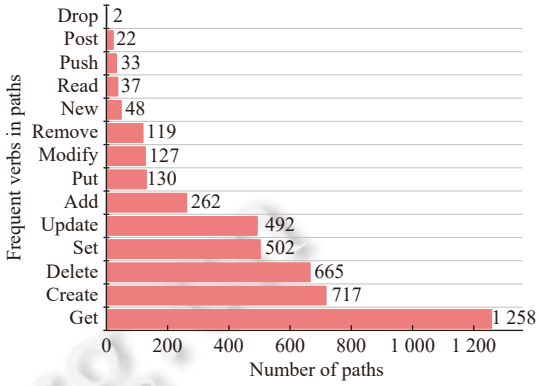
本文统计分析了 URI 路径中出现“api”的情况, 对应的是设计规范 A3-3 和 A3-4. 规范 A3-3 实现情况的统计通过分析 RAD 中的域名 (host)、基础路径 (basepath), 以及 OAS3 规范下的服务器信息 (servers). 统计结果如图 8(c) 所示, cloud 类别 API 对于该条规范的实现率较低, 只有 googleapis.com 在域名中出现了“api”. 其余各类别 API 对于该条规范的实现率都较高. 统计结果显示绝大多数 REST API (1674/1818, 92.08%) 遵循规范 A3-4.

● URI 属性

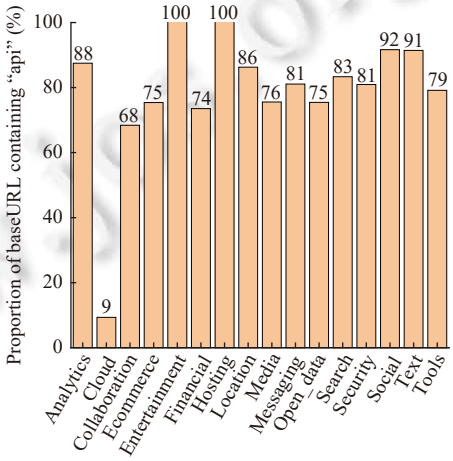
本文统计了查询属性中具有分页、过滤等功能性参数的使用情况, 该条设计规范 (A4-1) 适用于返回大量资源的 API, 需要对响应结果进行一定操作 (例如分页或条件过滤). 经统计, 180 个 (9.9%)API 使用了功能性查询属性. 图 9(a) 统计了每类 REST API 使用功能性查询属性的占比 (%), 可以看到 collaboration、media、social、open_data、search、hosting 和 location 等类别有较多 API 使用了功能性属性, 这些应用类别的共同特征在于查询结果返回的资源数量较多; 与之相反, cloud、security、tools 类别服务的资源概念较抽象, 且重点在于对资源的操作而非查询, 因此返回的网络实体资源少, 无需过滤和分页等功能性属性. 功能性属性的具体实现方式 (即属性名设计) 的统计如图 9(b) 所示, “limit”、“page”和“offset”限制返回资源的长度以及指定返回第几页的属性使用最为频繁, 其次是限定每页显示数量的属性, 如“per_page”、“pagesize”等, 还有以时间和序号等作为查询条件的属性也较为常见, 如“ending_before”和“starting_after”等, 用以对响应结果进行过滤.




(a) “资源名词性命名” 实现情况分布




(b) URI 动词频次图




(c) “基础路径包含 api 字样” 实现情况分布

图 8 URI 路径相关指导规范遵循情况分布

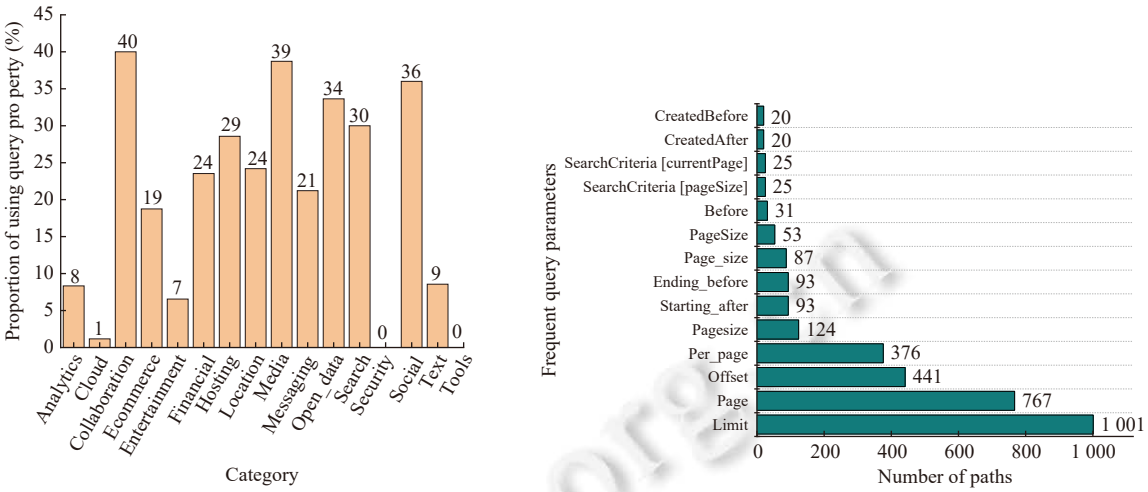
4.2.2 HTTP 交互

● 请求方法

经统计, HTTP GET 和 POST 方法使用最为频繁, 分别占 55.55% 和 26.40%. 随着 REST 架构风格应用的推广, 其他方法的语义和使用效果也逐步被认可和接受, 使用其他 HTTP 方法操作和管理资源的情况也逐渐增多, 使用 PUT、DELETE 和 PATCH 添加、删除和更新资源分别占 7.09%、7.02% 和 3.45%. HEAD 和 OPTIONS 这个方法极少使用, 分别只占 0.44% 和 0.05%. 这是因为: (1) HEAD 的操作效果与 GET 类似, 区别在于只获取响应头而不获取消息体, 客户端可以使用该方法来检测目标资源是否存在和有效, 或者只读取消息头的元数据而减少网络传输; 由于 GET 的功能更强, 因此在一般情况下更倾向于使用 GET 而非 HEAD; Openbanking (www.openbanking.org.uk) API 返回开放 ATM、银行门店信息、账户信息, 使用了 HEAD 方法来支持检测资源是否有效. (2) OPTIONS 用以获取目标资源允许的操作 (例如是否可以使用 DELETE 和 PUT 方法), 在 REST API 说明文档足够清晰和详细的前提下, 对于各个资源运行的操作也是固定可知的, 因此对于 OPTIONS 方法的实现和支持的 REST API 数量极为有限. Apidapp (https://apidapp.com/) 是一个金融类 API, 提供一些电子钱包的服务, 它将 OPTIONS 方法应用

到每一个 URI 上, 使用户在使用该 API 时能够预先了解资源的操作类型.

各类别 REST API 的 HTTP 方法使用情况统计如图 10 所示, 可以看到不同应用对于方法的使用不尽相同.




(a) “使用功能性查询参数” 实现情况分布 (b) 功能性查询参数使用频次图

图 9 URI 属性指导规范遵循情况分布

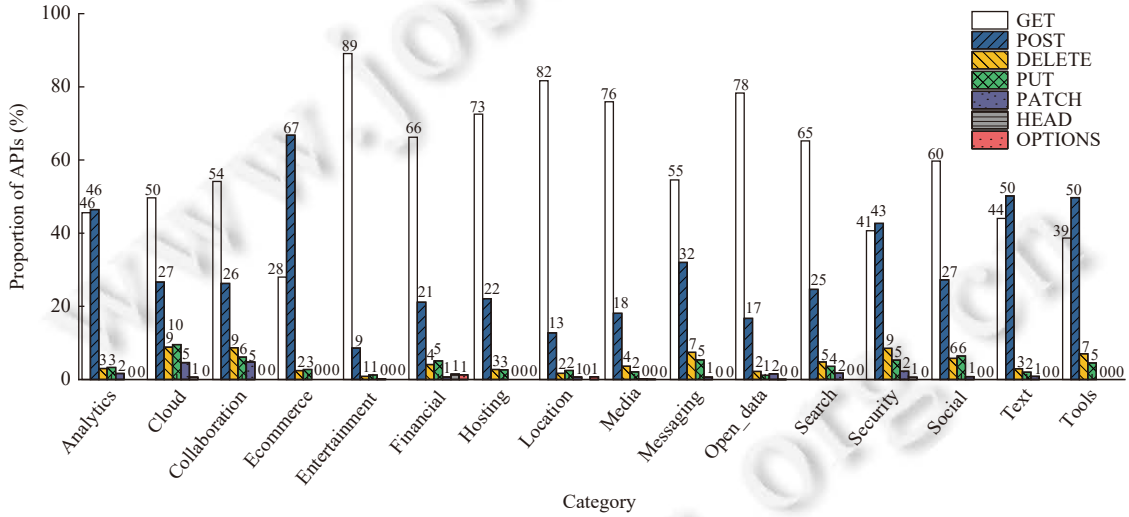



图 10 HTTP 方法使用情况分布

(1) cloud 和 collaboration 类别的 REST API 使用的 HTTP 方法类型最为多样, 且 DELETE、PUT 和 PATCH 方法的使用率在所有类别中最高. 这两类服务主要包括了公有云资源服务和以代码仓库为核心的协同开发等, 例如 collaboration 类中的 Github 如使用 PATCH 来编辑发布、评论, 更新用户信息等.

(2) eCommerce 类的 POST 方法使用占比最高, 该类别的服务提供较多的资源创建活动, 此外其中也存在违背 REST 规范 B1-2 的 REST API. 例如, eBay (https://www.ebay.com/) 大量使用 POST 方法设计服务, 比如“/shopping_cart/remove_item”“/shopping_cart/add_item”, 一方面这些路径中不应该使用代表 CRUD 操作语义的动词; 另一方面应该设计以资源为核心的 URI, 如“/shopping-cart/item”, 然后使用 DELETE 和 POST 方法实现向购物车中添加和删除商品. 与之类似的还有 adyen.com (https://www.adyen.com/) 的 AccountService 中的 URI “/createAccountHolder”“/getAccountHolder”等, 均使用 POST 方法实现.

(3) Entertainment、financial、hosting、location、media、open_data、search 几类应用均以提供不同类型的信

息服务为主, 因此大多数服务设计都主要使用了 GET 方法.

针对 HTTP 方法的误用情况, 本文检测的依据是 URI 中出现的动词 (短语) 与其使用的 HTTP 方法在语义上是否一致. 例如, 如果存在一个 URI“/delete-user”且对应的 HTTP 方法是 POST 而不是 DELETE, 那么则认为该 URI 存在 HTTP 方法的误用, 具体的是对 POST 方法的误用. 虽然该方法无法检测路径中不出现动词的 URI 是否存在 HTTP 方法的误用. 但是仍能够在一定程度上反映出各类型 HTTP 方法的误用情况以及各领域 REST API 对 HTTP 方法的误用情况, 如图 11 所示. 经统计, 出现动词的 4 385 个路径中, 有 2 318 个 (52.3%) 路径误用了 HTTP 方法, 其中 2 126 个 (48.5%) 路径误用了 POST 方法, 158 个 (3.6%) 路径误用了 GET 方法, 还有少数 (33 个) 误用了 PUT 方法. 图 11 反映出各领域 API 对 HTTP 方法的误用情况, 可以看到多数应用类别中误用最多的是 POST 方法, 只有少数 API 误用了 GET 方法, 比如 namsor.com (https://v2.namsor.com/NamSorAPIv2) “人名分类器”误使用 GET 方法进行删除操作, 应该使用 DELETE 方法, 违反 REST 规范.

● 响应状态码

REST 使用标准 HTTP 状态码用来描述服务响应的状态, REST API 的 RAD 应该描述每一个端点的响应状态 (包括成功、失败等) 和对应的解释以及措施等, 每种响应状态使用不同的状态码标识.

本文统计了 APIs.guru 收录的 REST API 中响应状态码总体使用情况. “2XX”表示请求成功状态的响应码, 共有 9 种成功状态码被使用, 其中次数最多的是 200 (43 932 次), 表示请求已成功, 请求所希望的响应头或数据体随此响应返回. “4XX”与“2XX”的使用总数相差无几, 表示客户端请求错误状态, 代表客户端的错误妨碍了服务器端的处理. “4XX”的每一种不同状态码代表不同的错误状态或错误原因. “4XX”的状态码种类最多, 共出现 54 种, 其中使用次数较多的“48X”频繁出现在 amazonaws.com 中, 例如, 使用次数最多 (7 148 次) 的 480 表示服务器暂时不可用, 481 表示呼叫的服务不存在, 482 表示检测到循环等. 此外, 使用率较高的是常见的“40X”状态码, 比如 400 表示不合法的请求, 401 表示需要用户身份验证, 404 表示找不到请求的资源等. “5XX”的状态码表示服务器错误, 统计中有 54 种状态码被使用, 但使用的次数均较少, 最多的是出现 3 060 次的 500, 表示服务器中断错误. “1XX”只有 101 被使用了 9 次, 在该响应之后, 服务器将会切换协议 (比如更低版本的 HTTP 协议).

本文还分类别统计了 REST API 响应状态码的使用情况. 首先统计单个 API 各端点中各类状态码的使用率, 比如一个 API 有 8 个端点, 8 个端点均使用了“2XX”响应码, 4 个端点有“4XX”响应码, 则该 API 对“2XX”和“4XX”响应状态码的使用率分别为 100% 和 50%. 然后计算各应用类别 API 对各类响应码的平均使用率, 结果如图 12 所示. 可以看到各应用类别对 2XX 的使用率都很高, 几乎每个端点都有成功状态的描述, 而错误状态 (“4XX”和“5XX”) 的使用率并不高. 因此, REST API 设计需要加强对各种响应结果的考虑, 对于错误状态的描述更有助于提高 API 的可用率, 有助于实现错误请求的描述以为用户发起合法请求提供更多知道信息.

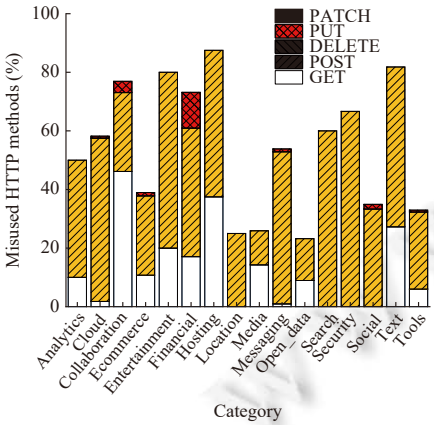



图 11 HTTP 方法误用分布

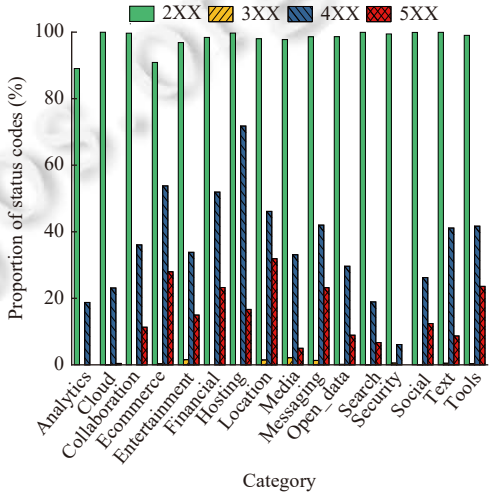



图 12 HTTP 响应状态码使用统计

4.2.3 HTTP 消息

● 消息头

本文通过分析 RAD 统计 REST API 在响应消息头中使用 Content-Type (C1-1)、Accept (C1-2) 和身份验证 (C1-3) 的情况, 结果如图 13 所示.

REST API 在消息头“Content-Type”中声明消息体的数据类型, 但是当前几乎所有 RAD 都不包含消息头中媒体类型的设置信息, 经统计, 1 818 个 RAD 只有 2 个描述了这一信息. RAD 应该更加详尽准确地描述 API 对于元数据 (包含消息头) 的设计.

API 通过声明“Accept”使用户可以设置希望接收的数据类型, 但是多数 RAD 中没有包含该信息, 少数 RAD 虽然包含“Accept”信息, 但也仅支持单一的期望数据类型. 例如, Github.com 提供的部分服务中仅声明了期望 JSON 一种类型.

REST 请求在消息头中使用 keys、Authorization 或 tokens 等进行用户身份验证, 主要是为了支持 REST API 的安全机制以及保证 REST API 的无状态原则. 例如, cloud 类中的 amazonaws.com 在消息头中使用 “X-Amz-Security-Token”实现安全认证. 可以从图 13 中看到各领域对此规范的实现率并不高, 深入分析有以下原因: 首先, 实验基于对 RAD 的分析, 但是并非所有的 REST API RAD 都包含这一内容, 比如 GitHub 使用头文件“Authorization”实现安全认证, 然而并没有在 RAD 中进行描述; 其次有些 API 是开放使用的并不需要用户身份认证, 因此没有这方面的设计实现.

● 消息体

本文主检测 REST API 对 HATEOAS 原则 (C2-2) 的实现与否, 即在响应中是否存在当前请求服务的下一步相关资源以及操作的链接. 对于 OAS 2.0 规范, 我们对 RAD 响应体的模板描述进行模糊匹配, 检测是否存在包含“link”的属性, 以此实现该规范的检测; 而对于 OAS 3.0 规范, 规范针对 HATEOAS 原则新增了“links”元素, 本文通过检测是否存在该元素来实现此规范的检测. 实验结果如图 14 所示, 可以看到对于 HATEOAS 原则的实现度依旧很低, 其中 cloud 类的实现度最高, Azure 中的部分 API 通过响应消息中的“nextLink”自定义属性来实现该规范, tools 类中的 microsoft-graph API (https://graph.microsoft.com/v1.0) 通过“links”元素提供下一步可行操作. 根据 Richardson 成熟度模型, RESTful 的最高层级 (level3) 是实现 HATEOAS 原则, 从统计结果来看, 目前大多数 REST API 仍处于 level 2.

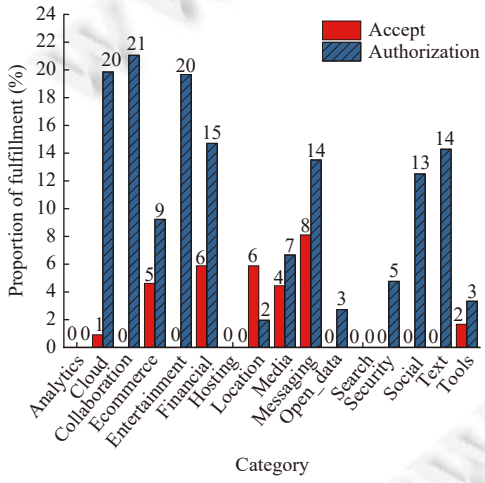



图 13 HTTP 消息头设置媒体类型和安全认证机制情况分布

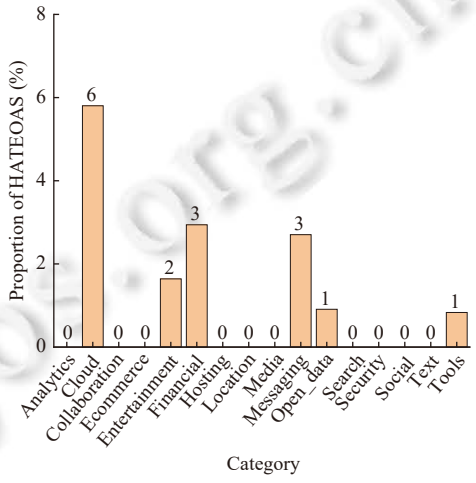



图 14 REST API 的 HATEOAS 原则实现分布

4.2.4 非功能设计

● 安全

OAS 3 中的安全方案有: APIKey、HTTP、OAuth2 以及 OpenIdConnect; OAS 2 中的方案有 APIKey、basic、

OAuth2. 本文主要检测并统计了上述安全机制在 REST API 中的使用情况. 如图 15 所示, 平均 83% 的 REST API RAD 声明了对安全方案的使用, 其中 tools、analytics、cloud、entertainment 和 security 几个类别中的 REST API 使用安全机制的比例均大于平均值, 这些类别中的服务普遍涉及大量资源操作, 并且这些资源具有私有性和非公开访问 (或付费访问) 的特点, 用户身份验证和访问权限控制十分重要. 进一步分析发现, 这些安全机制中 OAuth2 和 APIKey 使用最为广泛, 前者占 67.8%, 后者为 29%.

● 版本

版本信息是 REST API 一个重要组成部分, API 中有 4 种可能的策略来声明版本信息: 在请求头文件中声明, 在路径 (path) 中表示, 在域名信息中表示, 通过查询参数声明. 本文设计指导规范 D1 规范了 API 版本信息的声明位置: 建议在消息头中描述版本信息; 路径中不出现版本号, 避免破坏资源语义层次; 可以在域名信息中进行描述; 避免在路径的查询属性中描述版本信息.

据统计有 61.9% 的 RAD 声明了版本信息, 分应用类别的统计结果如图 16 所示. cloud 类中大多数版本信息出现在查询参数中, 进一步分析发现 Azure.com 包含的 RAD 大量使用查询参数“api-version”来标明版本信息, 并有示例显示 Azure 使用更新的日期来标识版本信息, 比如“2017-09-07-privatepreview”. 通过查询参数声明版本信息使得服务端必须先通过解析请求查询属性获得版本信息后才能提供对应的服务; 并且版本信息属于资源不相关属性, 破坏了路径的完整性和资源名词性. 除 cloud 类别外, 其他类别中多数 REST API 在路径中声明 API 的版本信息, 比如“/v1/userinfo”“v1/repo”, 在路径中描述版本信息同样被认为是违反 REST 规范的 (D1-2); 但是在 URL 中声明服务的版本信息是最简明的方式, 因此可以将版本信息描述设置在服务端信息 (OAS 3 规范下的 server 元素) 或域名信息 (OAS 2 规范下的 host 元素和 basePath 元素) 中, 避免路径中的冗余同时保证了路径的资源名词性. 尽管已有的设计规范建议在头文件中声明版本信息 (D1-1), 但实际应用中目前很少有 API 实现.

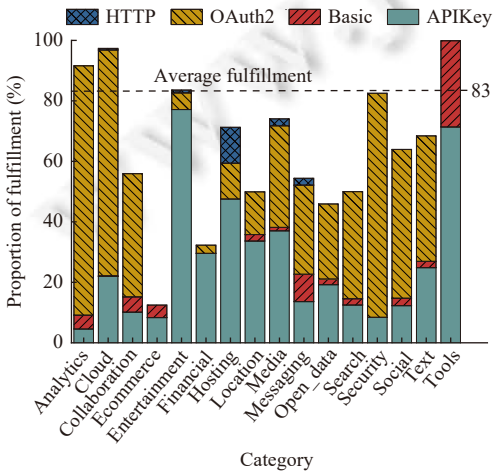



图 15 安全机制应用统计

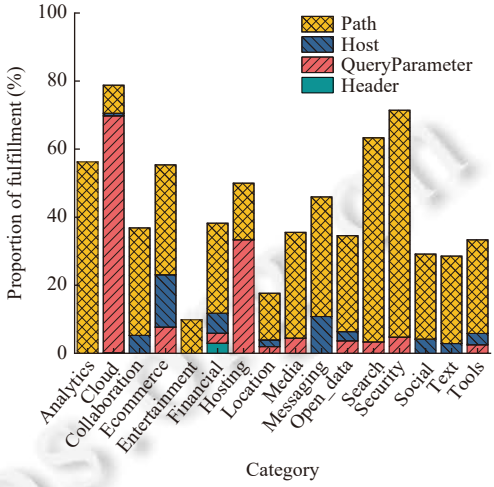



图 16 API 版本声明统计

进一步的, 本文统计分析了出现在路径 (path) 和域名 (host) 中的版本描述模式, 有不到 1/3 的 API 使用了语义版本号 (semantic versioning), 比如 Googleapi 中出现在路径中的版本信息使用“v1alpha”“v1beta2”等来描述.

● 缓存

Web 服务的缓存机制可以分为强缓存和协商缓存, 强缓存通过响应头文件 cache-control、date 或者 expires 等实现 (D3-1); 在协商缓存机制 (D3-2) 中, 服务器通过响应头文件 etag 或 last-modified 告知客户端资源状态, 下次请求时客户端通过头文件 If-None-Match 或 If-None-Match 携带资源状态值, 以此进行资源状态的对比协商是否返回 (新的) 资源. 本文对两种缓存方式的使用做了统计, 如图 17 所示, 纵轴表示某领域中实现了该规范的 API 占比, 如果某 API 的 RAD 中有在某个响应中描述了对应头文件, 则该 API 实现了这一规范. 统计结果显示只有少数

(68/1818) RAD 中有缓存机制的描述, 其中协商缓存有 61 个占绝大多数. 实验发现只有少数 RAD 中有对响应头文件的描述, 有些 API 可能使用了缓存机制但并没有进行描述, 需要获取真实响应进一步分析, 因此这一实验并不能全面反映 REST API 对缓存机制的使用情况, 但该实验在一定程度上反映了两种缓存机制的使用分布, 即 REST API 更偏向使用协商缓存机制.

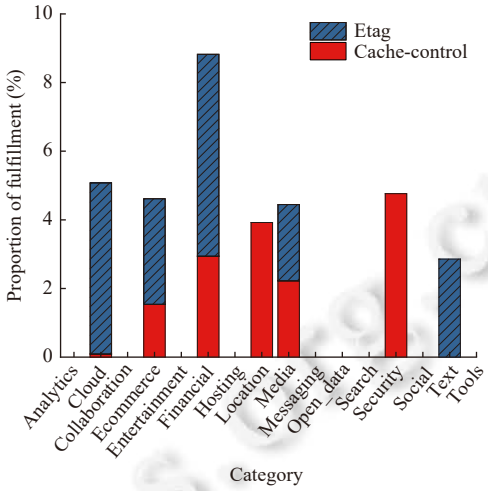



图 17 REST API 缓存使用情况分布

### 4.3 应用相关的 REST API 设计特征分析

本节主要从不同应用类别的视角分析 REST API 的设计特征, 尤其是对设计规范的遵循情况. 其中最有典型特征的应用类别包括: cloud、collaboration、tools 和其他以信息检索为主的应用类别.

#### 4.3.1 云平台类 REST API

云平台类 REST API 的典型代表包括大型公有云平台: googleapi, amazonaws, azure 和 microsoft. 总体而言, 此类 API 提供的服务复杂, 可操作的资源类型多样, 在资源路径数量方面占显著比重; 另一方面, 可操作资源之间的层级和组织结构导致此类 REST API 的平均路径层级数较大. 具体的, 云平台类 REST API 设计具有以下特征.

(1) 在总体结构方面, 云平台类 REST API 的资源路径数平均为 14.52, 每条路径平均包含 1.47 个端点. API 的平均层级为 6.27, 最大层级达到 19.

(2) 在 URI 格式方面, 此类 API 对于路径使用小写这一规范 (A1-3) 的违背率高, 仅有 4.15% 的 REST API 完全遵守, 而完全违背的 REST API 占 81.5%, 其命名通常延续了传统编程命名风格的“驼峰式”格式.

(3) 在 URI 路径方面, 部分 API 仍然在命名中使用了表示操作的动词, 73.47% 的 API 在资源命名中遵循规范 A3-2, 一方面是没有严格遵循 REST 架构风格, 使用单一的 HTTP 方法实现对同一资源的多个操作 (详见第 4.2.2 节); 另一方面是由于服务功能的复杂性而无法映射到 HTTP 的标准方法上, 此类情况需结合应用语义具体分析.

(4) 在 HTTP 方法方面, 平台类 REST API 违背 HTTP 方法正确使用的现象十分普遍. 例如, amazonaws 中 POST 方法使用占比大, 接口设计没有遵循 REST 架构风格原则, 延续了传统网络服务以操作为核心的设计模式, 将路径命名模式为“action=xxx”“target=xxx”, 使用 POST 方法发起请求. Googleapi 中经常出现“setxxx/getxxx”的接口设计, 同样不符合 REST 以资源为核心的命名原则.

(5) 在 URI 属性方面, 平台类应用中查询和返回大量资源的场景不多, 极少用到功能性属性对响应结果作进一步处理, 所以仅有 1.17% 的 API 使用了诸如分页和查询过滤的功能性属性.

(6) 在非功能属性方面, 97.50% 的平台类 REST API 提供了安全机制, 其中以 OAuth 2 为主 (76.6%), 相对于其他几种安全机制, OAuth 2 有更高的安全等级. 77.9% 的 API 有对版本信息的描述, 该类别 REST API 存在较多的版本更迭. 云平台 API 对于版本声明的方式并不统一, 并存在违背设计指导规范的情况. 例如 Azure.com 的 RAD 大量使用查询参数“api-version”声明版本, 不符合 REST API 设计风格; Amazonaws 平台下有少数 API 在路径中描

述了版本信息; Googleapis 的大多数 API 都在路径中描述了版本信息; 还有极少数的 API 在域名 (包括基础路径) 中描述版本信息.

4.3.2 协同开发与工具类 REST API

协同开发与工具包括了 collaboration 和 tools 两大类的 REST API, 包括协作平台 (如 GitHub), 以及针对网页、项目、公司业务等的管理工具 (如 Trello 项目管理工具). 总体而言, 协同开发与工具类的 RESTful 程度较高, 支持一些对资源的常见操作 (如: 增删改查), HTTP 方法的使用较为标准, 对于各个设计指导规范的遵循程度总体较好. 此类 API 的设计具有以下特征.

(1) 在总体结构方面, collaboration 和 tools 两类 API 是典型的资源密集型, 平均包含的资源数量分别达到了 74.9 和 61 (图 5(b)).

(2) 在资源设计方面, 这两类 API 的 URI 中不包含文件扩展名, 采用名词为资源命名的平均实现率达到 94.7%, 少数 (30%) 的 API 在路径中声明版本. 尽管同属资源密集型, 两类 API 在查询属性方面的特征极为不同, collaboration 类别的 API 较多的 (40%) 使用了查询属性, 而 tools 类别的 API 则没有使用查询属性 (图 9(a)). 这是因为虽然两类 API 都包含了种类较多的资源, 但是 collaboration 类别的 API 进一步包括了大量的资源实例, 例如 GitHub 和 Bitbucket 等都提供了数量众多的代码仓库、提交、评论等资源实例, 因此使用查询属性进行过滤和定位十分必要.

(3) 在 HTTP 方法方面, 这两类 API 对于各种 HTTP 方法的支持比较全面.

4.3.3 信息服务类 REST API

信息服务类 REST API 是指以信息检索为主的服务对外提供的接口, 包括了 entertainment、financial、hosting、location、media、open_data、search 几个类别下的 API, 如: 体育赛事时间表 (如: Sportsdata, https://sportsdata.io/)、日历信息查询 (如: Google calendar API, https://developers.google.com/calendar/v3/reference/calendars)、地理位置获取 (如: Gisgraphy, https://services.gisgraphy.com/static/leaflet/index.html)) 等. 总体而言, 信息服务类 REST API 的 RESTful 程度较高, 且大多遵循“检索-响应”的资源操作模式, 大多仅支持对 REST API 所代表的服务端资源的读操作, 因此, 总体上此类 API 使用 GET 方法的比例较大.

(1) 在 URI 路径方面, 不出现动词的平均实现率达到 95.5%, RESTful 程度较高.

(2) 在 URI 属性方面, 信息服务类应用中查询和返回大量资源的场景十分典型, 同时由于资源数量大使得基于条件的查询需求明显, 因此较多的使用了功能性查询属性 (比如: limit, page 等) 来限制响应信息的展示.

(3) 在 HTTP 方法方面, 该类 API 多提供资源获取类服务, 因此 GET 方法使用占比最高达到 77%.

### 4.4 小 结

本文深入分析了 APIs.guru 收录的上千个 REST API RAD, 通过大规模实证研究对当前 REST API 设计特征以及设计指导规范的遵循情况得出以下结论.

(1) REST API 的数量分布随着包含资源数量的增加总体呈下降趋势, 大多数 API 包含少量的资源, 其中的特例是 collaboration 和 tools 类别的 REST API.

(2) 当前 REST API 对于资源设计的相关规范遵循程度较高, 尤其是 A1-1, A1-2 和 A1-4. A1-3 的遵循程度较低, 主要问题在于使用“驼峰式”命名格式为资源和属性命名, 而设计规范使用小写和分隔符“-”格式的命名.

(3) REST API 的路径中存在不正确使用动词 (及短语) 的情况, 表达的语义能够完全映射到 HTTP 的标准方法, 应该使用“单一资源路径, 多个操作端点”的方式实现对同一资源的不同操作.

(4) HTTP 的 GET 和 POST 方法使用占比最高, 包括 PUT、PATCH 以及 DELETE 等方法也有使用. HTTP 方法误用情况较为普遍, 以 POST 方法的误用最为常见, 如使用 POST 实现资源的删除和查询, 应该使用正确的方法实现资源的 CRUD 操作.

(5) REST API 对于响应状态码的使用较为普遍, 状态码的类型多样, 远远多于已有设计规则[5]中提到的状态码, 其中种类最多的是表示客户端和服务器端错误的“4XX”与“5XX”. 但是, 从 RAD 分析来看, 错误状态码 (“4XX”和“5XX”) 的使用率不高, 很多 API 只有对成功状态的描述. 因此, 建议 REST API 设计加强对各种响应结果的解释和描述, 有助于错误的理解和诊断定位.

(6) OAuth2 是目前使用最为广泛的安全验证方式, 同时也是我们建议使用的安全协议, APIKey 也拥有较高的使用频率.

(7) 尽管推荐在 REST API 的消息头声明版本, 但是实际中多数 API 在路径中声明 API 的版本信息, 比如“/v1/userinfo”和“v1/repo”, 以使 API 更加简答明了, 利于使用[17]. 由此可以看出, API 版本声明的设计规范意见并不统一, 几种方式各有利弊[17].

本文发现不同应用类别的 API 对于 RESTful 设计的实现程度具有各自的特征和差异, 这些差异主要是由 REST API 的资源情况和操作模式决定的.

(1) 资源操作和管理复杂的云平台类 API 的 RESTful 程度较低, 延续传统网络服务的设计模式, 以操作为核心, 违背 REST 架构以资源为核心的命名原则.

(2) 信息服务类 API 的 RESTful 程度最高, 其操作较为单一, 通常是对资源的“读”模式, 因此 GET 方法占比最大;由于返回资源结果较多, 因此较多的使用了功能性查询属性来限制响应信息的展示.

(3) 协同开发与工具类提供对网页、项目、应用等的开发和管理服务, 一方面涉及的资源种类和数量多, 另一方面资源的各自操作均有涉及. 因此, 这些 API 对于各种 HTTP 方法的使用比较全面, 相较平台类 API 容易实现 RESTful 以资源为核心的设计方式, RESTful 程度整体较高.

最后, 本文研究发现, 尽管符合 OAS 的 RAD 包含了 REST API 的多方面元素和信息, 但是在响应状态码、HATEOAS 以及其他非功能属性的设置声明方面仍然欠缺. 因此, 本文建议一方面加强 RAD 文档信息的描述能力, 尽可能多的覆盖 REST API 在 HTTP 交互方面的信息; 另一方面, 建议提供动态分析和测试的方法来弥补静态分析的不足和局限性, 进一步加强 REST API 分析检测能力. 此外, REST API 的设计规范和最佳实践也并非一成不变, 一方面有特定的适用场景; 另一方面也随着 Web 服务、应用系统和应用领域的发展, 新的设计规范也会不断涌现, 而已有的规范也存在过时的可能, 因此分析规范的遵循情况必须结合相应的场景, 采用发展的眼光来分析和评价.

## 5 讨 论

本文方法通过静态分析遵循 OAS 的 RAD 实现, 具有一定的局限性, 需要考虑以下方面的问题.

(1) 方法的适用性. 尽管本文以遵循 OAS 的 RAD 为输入, 但并非必要条件, 仍可以采用类似 D2Spec[20]的方法从其他类型和格式的 REST API 文档 (如 html, RAML[11]) 中获取 API 的元素和数据进行后续的规范检测.

(2) RESTer 无法检测需要分析 API 响应的设计规范, 必须采用动态的方法, 通过构造和发送实际的 API 请求来获取响应. 本文后续工作将参考和结合 REST API 测试用例生成方面的研究[21−26], 实现基于动态访问的规范检测, 作为本文工作的有力补充.

(3) 本文方法以 RAD 与实际 REST API 一致为假设前提, 可能会存在两者不一致的情况, 导致分析检测的结果与真实 API 的情况不相符. 本文以 APIs.guru 收录的 REST API RAD 为数据集, 一方面它是当前最大的 REST API 目录, 而且在已有的研究和文献中多次被使用;另一方面, 大多数的 RAD 是 API 的官方版本, 因此在文档质量方面有一定的保障. 此外, 本文对大规模数据集上的检测结果做了进一步分析和统计, 即使存在少量的不一致情况, 对于总体和平均结果的影响也十分有限, 研究结果能够反映出 REST API 在设计方面的总体特征.

(4) 本文实验数据集有较好的代表性, 在应用领域方面覆盖了 64 个小类和 16 个大类, 包括了广泛使用的、流行度较高的平台和服务的 REST API, 且 API 文档的数量接近 2 000 个, 远远多于已有相关工作中目标数据的规模, 未来工作也将进一步获取和分析更多的 REST API.

## 6 相关工作

REST API 已成为基于服务架构应用系统的重要组成部分, 针对 REST API 的研究工作主要集中在设计分析、自动化测试和演化维护方面.

### 6.1 REST API 设计分析

早期 Web 服务和 Web API 的实证研究与分析[27−29]发现, REST 架构风格的相关原则和特性并没有得到很好

的遵循, Web 服务的 RESTful 程度不高. 随着一系列设计指导规范的提出[5], 本文研究结果表明近年来 Web 服务 API 的 RESTful 程度越来越高, 较好地遵循了设计指导规范.

当前已有一些针对 REST API 设计规则及其实现程度的分析研究. Rodríguez 等人[14]分析了大量移动应用 HTTP 请求数据, 基于 5 条最佳实践 (本文规范体系覆盖了这 5 条最佳实践) 度量和评价移动应用中 API 的 RESTful 程度, 结果显示多数 API 达到了 Richardson 成熟度模型的 Level 2 级, 但并没有完全实现基于 HTTP 协议的标准化的服务接口. Petrillo 等人[15]的研究工作只针对 cloud 类别的 REST API, 从 URI 设计、请求方法使用、响应状态码使用、消息头以及其他 5 个维度进行最佳实践总结, 人工分析了 3 个云平台对于 REST API 设计最佳实践的遵循情况, 认为尽管只遵循了 2/3 的设计规则, 这些平台接口设计的规范化和 RESTful 程度可以接受; 本文的规范分类维度覆盖了这项研究工作, 去除了一些针对性的最佳实践并对一些最佳实践进行抽象总结出规范, 同时对其实现自动检测, 并且本文的数据集覆盖了全类别不只针对 cloud 类别. Haupt 等人[30]提出一个 REST API 结构分析框架, 基于 Swagger 描述文档分析了 APIs.guru 上 286 个 REST API 的总体结构, 包括: 资源、HTTP 方法、链接和路径. SODA-R[6]是针对 8 个 REST 反模式和 5 个模式的基于启发式的半自动检测方法, 通过检测 12 个 REST API 发现了其中在消息头、格式和协议等方面存在违背自描述特征 (self-descriptiveness) 的反模式, 以及在超媒体和缓存方面的反模式. DOLAR[7]定义了 5 对语言相关的 (反) 模式并检测发现了 15 个 REST API 文档中的 URI 设计语法问题, 所提出的设计模式本文规范分类体系已覆盖. Neumann 等人[16]人工分析了 500 个 REST 服务的 API 的技术特征和对于 REST 架构风格的遵循程度, 结果显示仅有极少数 API 完全遵循所有的 REST 原则, 但是部分设计最佳实践还是得到了较为普遍的应用.

上述研究与本文工作最为相关, 但仍存在以下不同之处. (1) REST API 设计规则、(反) 模式和最佳实践分散在上述各个研究工作中, 导致其分析和检测的侧重点各有不同; 本文工作归纳梳理了各维度的设计指导规范, 因此在设计指导规范的检测能力和覆盖度方面具有优势; 同时本文实现了全自动的规范检测. (2) 研究结论的差异性, 由于时间差异, 本文研究发现越来越多的 REST API 能够较好的遵循 REST 架构风格和设计指导规范, 这与早期研究工作的结论有所不同, 由此也说明了 Web 服务的 RESTful 程度不断提高. (3) 本文工作区别于已有研究的一个显著特征在于, 考虑了不同应用领域类别对 REST API 设计的影响, 分析讨论了不同类别 REST API 在遵循设计指导规范方面的特征和差异. 此外, 本文工作基于 1 818 个真实 REST API, 其研究目标规模远远大于原有工作.

### 6.2 REST API 测试

黑盒测试是当前 REST API 测试的主流方法, 其中的关键问题在于如何生成有效、多样的测试用例, 达到较好的测试覆盖度, 并基于测试断言 (test oracle) 发现被测 REST API 的问题. 现有工作主要基于 REST API 说明文档分析生成测试用例, 包括: (1) 使用 RAD 中的缺省参数值、样例参数值和随机值作为测试用例输入[21]; (2) 分析推断 API 端点 (操作) 之间隐含的顺序和依赖, 从而产生正确的测试用例执行顺序[22,26]; (3) 基于属性规范和约束生成输入参数[23]; (4) 基于参数间依赖约束分析生成输入参数[24]. 为了提高测试覆盖度, 多种 Data fuzzing[25]技术被用来生成测试用例数据, 当前对于 REST API 的测试覆盖标准包括路径、操作、参数 (值)、内容类型、响应状态码等[31]. 针对缺少 REST API 测试断言的问题, Segura 等人[32]提出 REST API 蜕变测试 (metamorphic testing) 方法, 基于 REST API 输出的蜕变关系来发现和检测存在的问题. 类似的, Godefroid 等人[33]提出针对 REST API 的差分回归测试, 其核心思想也是通过比较回归前后相同测试用例的测试结果来发现问题.

除黑盒测试外, EvoMaster[34]是为数不多的一种 REST API 白盒测试方法, 它采用基于搜索的测试技术, 以测服务代码覆盖度最大化为优化目标进行测试.

总体而言, REST API 测试方法通过实际的 API 访问和调用来发现被测服务在运行中的错误, 与基于 RAD 静态分析的设计指导规范检测互为补充. 进一步的, 基于测试也能够更加细致的发现目标 API 在 HTTP 交互维度方面对于设计规范的遵循程度, 如 HTTP 方法和响应状态码的正确使用等.

### 6.3 REST API 演化维护

REST API 的版本演化, 尤其是版本间的不兼容变化 (breaking changes), 可能导致依赖它的其他应用和服务的

不可用. RADA[35]基于 OAS 文档识别 API 版本演化中弃用 (deprecated) 的 API 元素, 并分析其影响的操作. 实证研究使用 RADA 分析了 APIs.guru 上 1 818 个 API 文档, 发现其中确实存在与 API 版本演化相关的问题.

尽管遵循规范的 REST API 文档对于理解、使用和测试 REST API 十分重要, 但是维护和创建 RAD 仍需要一定成本和代价. D2Spec[20]分析 REST 在线文档, 基于启发式规则识别和抽取与 OAS 相关的数据元素, 并组合构建 REST API RAD.

## 7 总 结

随着 REST 架构的兴起, REST API 成为访问和重用 REST 服务的主要途径, 遵循 REST API 设指导规范对于提高接口质量, 支持接口的维护、管理和应用至关重要. 本文首先建立了一个 REST API 设计指导规范分类体系 RADRC 对当前主流的设计规则进行分类;其次实现了基于 REST API 描述文档静态分析的设计指导规范分析与检测工具 RESTer; 最后使用 RESTer 开展 REST API 设计实证研究. 实证研究结果有助于深入了解当前 REST API 及其设计规则的特征、现状和不足, 对于提高 REST API 设计质量和研究改进设计指导规范具有实际意义. 未来工作将从 3 个方面展开: (1) 结合动态分析和测试的方法提高设计指导规范的检测能力; (2) 更加广泛和全面的整理归纳 REST API 的设计指导规范, 并分析其合理性和适用性; (3) 针对更多的真实 REST API 进行检测分析, 进一步评价方法的广泛适用性.

**References:**


[1] Tan W, Fan YS, Ghoneim A, Hossain MA, Dustdar S. From the service-oriented architecture to the Web API economy. IEEE Internet Computing, 2016, 20(4): 64–68. [doi: 10.1109/MIC.2016.74]

[2] Di Francesco P, Lago P, Malavolta I. Architecting with microservices: A systematic mapping study. Journal of Systems and Software, 2019, 150: 77–97. [doi: 10.1016/j.jss.2019.01.001]

[3] Fielding RT. Architectural style and the design of network-based software architecture [Ph.D. Thesis]. Irvine: University of California, 2000.

[4] ProgrammableWeb. API directory. 2020. http://www.programmableweb.com

[5] Masse M. Rest API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media Inc., 2011.

[6] Palma F, Dubois J, Moha N, Guéhéneuc YG. Detection of REST patterns and antipatterns: A heuristics-based approach. In: Proc. of the 12th Int'l Conf. on Service-oriented Computing. Paris: Springer, 2014. 230–244. [doi: 10.1007/978-3-662-45391-9_16]

[7] Palma F, Gonzalez-Huerta J, Moha N, Guéhéneuc YG, Tremblay G. Are RESTful APIs well-designed? Detection of their linguistic (anti) patterns. In: Proc. of the 13th Int'l Conf. on Service-oriented Computing. Goa: Springer, 2015. 171–187. [doi: 10.1007/978-3-662-48616-0_11]

[8] Richardson L. Richardson Maturity model-steps toward the glory of REST. 2020. https://martinfowler.com/articles/richardsonMaturityModel.html

[9] OpenAPI specification. 2020. https://github.com/OAI/OpenAPI-Specification

[10] Swagger. API development for everyone. 2020. http://swagger.io/

[11] RAML workgroup. RESTful API modeling language specification version 1.0. 2020. https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/

[12] Robie J, Cavicchio R, Sinnema R, Wilde E. RESTful service description language (RSDL), describing RESTful services without tight coupling. Balisage Series on Markup Technologies, 2013, 10: 6–9.

[13] APIs. guru. 2020. https://apis.guru/

[14] Rodríguez C, Baez M, Daniel F, Casati F, Trabucco JC, Canali L, Percannella G. REST APIs: A large-scale analysis of compliance with principles and best practices. In: Proc. of the 16th Int'l Conf. on Web Engineering. Lugano: Springer, 2016. 21–39. [doi: 10.1007/978-3-319-38791-8_2]

[15] Petrillo F, Merle P, Moha N, Guéhéneuc YG. Are REST APIs for cloud computing well-designed? An exploratory study. In: Proc. of the 14th Int'l Conf. on Service-Oriented Computing. Banff: Springer, 2016. 157–170. [doi: 10.1007/978-3-319-46295-0_10]

[16] Neumann A, Laranjeiro N, Bernardino J. An analysis of public REST Web service APIs. IEEE Trans. on Services Computing, 2021, 14(4): 957–970. [doi: 10.1109/TSC.2018.2847344]

[17] RESTful API versioning insights. 2020. https://blog.restcase.com/restful-api-versioning-insights/

[18] Gamez-Diaz A, Fernandez P, Ruiz-Cortes A. An analysis of RESTful APIs offerings in the industry. In: Proc. of the 15th Int'l Conf. on Service-oriented Computing. Malaga: Springer, 2017. 589–604. [doi: 10.1007/978-3-319-69035-3_43]

[19] Swagger Validator Badge. 2020. https://github.com/swagger-api/validator-badge

[20] Yang JQ, Wittern E, Ying ATT, Dolby J, Tan L. Towards extracting web API specifications from documentation. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 454–464. [doi: 10.1145/3196398.3196411]

[21] Ed-Douibi H, Izquierdo JLC, Cabot J. Automatic generation of test cases for REST APIs: A specification-based approach. In: Proc. of the 22nd Int'l Enterprise Distributed Object Computing Conf. (EDOC). Stockholm: IEEE, 2018. 181–190. [doi: 10.1109/EDOC.2018.00031]

[22] Viglianisi E, Dallago M, Ceccato M. RESTTESTGEN: Automated black-box testing of RESTful APIs. In: Proc. of the 13th Int'l Conf. on Software Testing, Validation and Verification (ICST). Porto: IEEE, 2020. 142–152. [doi: 10.1109/ICST46399.2020.00024]

[23] Karlsson S, Čaušević A, Sundmark D. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In: Proc. of the 13th Int'l Conf. on Software Testing, Validation and Verification (ICST). Porto: IEEE, 2020. 131–141. [doi: 10.1109/ICST46399.2020.00023]

[24] Martin-Lopez A, Segura S, Ruiz-Cortés A. RESTest: Black-box constraint-based testing of RESTful Web APIs. In: Proc. of the 18th Int'l Conf. on Service-oriented Computing. Dubai: Springer, 2020. 459–475. [doi: 10.1007/978-3-030-65310-1_33]

[25] Godefroid P, Huang BY, Polishchuk M. Intelligent REST API data fuzzing. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Virtual: ACM, 2020. 725–736. [doi: 10.1145/3368089.3409719]

[26] Atlidakis V, Godefroid P, Polishchuk M. RESTler: Stateful REST API fuzzing. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 748–758. [doi: 10.1109/ICSE.2019.00083]

[27] Maleshkova M, Pedrinaci C, Domingue J. Investigating Web APIs on the world wide Web. In: Proc. of the 8th IEEE European Conf. on Web Services. Ayia Napa: IEEE, 2010. 107–114. [doi: 10.1109/ECOWS.2010.9]

[28] Renzel D, Schlebusch P, Klamma R. Today's top "RESTful" services and why they are not RESTful. In: Proc. of the 13th Int'l Conf. on Web Information Systems Engineering. Paphos: Springer, 2012. 354–367. [doi: 10.1007/978-3-642-35063-4_26]

[29] Bülthoff F, Maleshkova M. RESTful or RESTless–Current state of today's top Web APIs. In: Proc. of the European Semantic Web Conf. Crete: Springer, 2014. 64–74. [doi: 10.1007/978-3-319-11955-7_6]

[30] Haupt F, Leymann F, Scherer A, Vukojevic-Haupt K. A framework for the structural analysis of REST APIs. In: Proc. of the 2017 IEEE Int'l Conf. on Software Architecture (ICSA). Gothenburg: IEEE, 2017. 55–58. [doi: 10.1109/ICSA.2017.40]

[31] Martin-Lopez A, Segura S, Ruiz-Cortés A. Test coverage criteria for RESTful web APIs. In: Proc. of the 10th ACM SIGSOFT Int'l Workshop on Automating TEST Case Design, Selection, and Evaluation. Tallinn: ACM, 2019. 15–21. [doi: 10.1145/3340433.3342822]

[32] Segura S, Parejo JA, Troya J, Ruiz-Cortés A. Metamorphic testing of RESTful Web APIs. IEEE Trans. on Software Engineering, 2018, 44(11): 1083–1099. [doi: 10.1109/TSE.2017.2764464]

[33] Godefroid P, Lehmann D, Polishchuk M. Differential regression testing for REST APIs. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Virtual: ACM, 2020. 312–323. [doi: 10.1145/3395363.3397374]

[34] Arcuri A. RESTful API automated test case generation with EvoMaster. ACM Trans. on Software Engineering and Methodology, 2019, 28(1): 3. [doi: 10.1145/3293455]

[35] Yasmin J, Tian Y, Yang JQ. A first look at the deprecation of RESTful APIs: An empirical study. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Adelaide: IEEE, 2020. 151–161. [doi: 10.1109/ICSME46990.2020.00024]

**周芯宇**(1996－), 女, 硕士, 主要研究领域为软件工程.

**吴国全**(1979－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 软件测试与维护, 面向服务的计算.

**陈伟**(1980－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为智能软件工程, 服务计算, 网络分布式计算.

**魏峻**(1970－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 分布式系统, 服务计算.