doi: 10.3969/j.issn.0490-6756.2019.03.009

基于 Metadata 的监控数据 API 设计与实现

冯 骐, 马晨辉, 沈富可
(华东师范大学信息化办公室, 上海 200062)

摘 要: 传统监控系统虽然提供了 API 接口,然而监控系统之间 API 风格各异,缺乏标准,应用程序调用比较困难,不利于数据可视化的推动. 文章提出了一种基于 Metadata 的监控数据 REST API 设计方法,根据监控数据的特点进行数据抽象,以 Metadata 的理念进行数据描述,并通过 REST 对 API 进行规范. 从而使得前端数据展示能够标准化和自动化,并给出了实际的应用示例,验证了设计的可行性.

关键词: 监控; API; Metadata; REST
中图分类号: TP311.5 **文献标识码**: A **文章编号**: 0490-6756(2019)03-0431-06

Design and implementation of REST API for monitoring data based on Metadata

FENG Qi, *MA Chen-Hui*, *SHEN Fu-Ke*
(Information Technology Service Center of East China Nomal University, Shanghai 200062, China)

Abstract: This paper proposes a Metadata-based monitoring data REST API design method, in which data abstraction and data description is based on the characteristics of monitoring data and metadata concept respectively, APIs specified with REST make it possible for the monitoring data presentations to be standardized and automated, some practical application examples are given to validate the feasibility of the proposed design method.

Keywords: Monitor; API; Metadata; REST

1 引 言

应用编程接口 API(Application Programming Interface)就是软件系统不同组成部分衔接的约定. 它有助于降低系统各部分的相互依赖,提高组成单元的内聚性,降低组成单元间的耦合程度,从而提高系统的维护性和扩展性. 对于 WEB 开发而言,当下的一种趋势是前后端分离技术[1]. 即后端服务只提供数据 API,前端通过 JS 等方式从后端的 API 中加载数据,动态生成 WEB 网页. 这种模式能够清晰的实现前后端的解耦和分工,从而提高开发效率.

REST[2](Representational State Transfer) 是一种 WEB API 的设计规范,由 RoyThomas Fielding 在其博士论文中首次提出. 它将请求的具体内容抽象为资源,通过 HTTP 协议的基本操作来进行资源表现的状态转化. 符合 REST 设计规范的 API 我们称之为 REST API .

Metadata[3]即元数据,意为用于描述数据的数据. 它使信息的描述和分类可以实现格式化,使得数据结构标准而有序,从而利于自动化[4]程序的处理.

监控系统是最为重要的运维基础设施之一. 它通过采集各个业务,系统内的监控指标,进行统一的数据展示和告警. 业内知名的监控系统,诸如 Zabbix 等都提供了其自身的监控数据 API 接口[5]. 然而这些 API 接口的风格,数据标准,设计

收稿日期: 2018-07-01
作者简介: 冯骐 (1989-), 男, 上海人, 硕士,工程师, 研究领域为软件定义网络, 自动化运维, 信息化基础设施架构等.
作者简介: 沈富可. E-mail: fkshen@ecnu.edu.cn

模式都各自不同,第三方前端——例如 Grafana[6] 不得不以分别适配不同的后端数据源,解析不同的 API 来构建统一的 dashboard. 然而 Grafana 仅是对不同的数据源进行解析和展示,并没有进行二次的 API 封装供外部读取. 因此其数据展示还是被限定在 Grafana 内部,难以与其他业务集成,无法进一步灵活的应用其数据.

本文提出一种基于 Metadata 的监控数据 REST API 设计,使得前端在进行数据展示时只需要适配这一种标准化的数据源即可. 其背后的数据整合由 API 提供者预处理[7],并隐藏起来不向前端暴露. 因此在该设计下,前端的监控数据展示能够简单,高效的进行开发和集成,并且能够伴随后端的数据更新自动适配. 从而显著降低监控数据可视化的开发周期和成本,进而推动运维可视化的落地,并有效提升运维能力.

2 API 设计

2.1 设计理念

API 的设计原则在于标准化,然而数据的类型繁多,数据风格各异,对其分别指定标准并不现实,也难以落地实现. 尽管 API 是一种约定,然而过多的约定势必造成灵活性和扩展性的缺失,过多的标准也势必让标准无法推行,最终反而变成"不标准"的异类. 然而过于灵活的设计也会让 API 变得难以结构化,不利于应用程序调用. 如何权衡是 API 设计时的一大考验.

引入 Metadata 理念是一个可行的方向,根据监控数据的特点,我们可以对监控数据的 Metadata 进行描述,形成数据描述的规范而不规范数据本身. 由于监控类的数据结构简单,特点清晰,我们可以很容易的借助 Metadata 对其进行结构化描述,而不至过于繁琐. 同时借助 REST 规范对 API 的模式进行约束,从而形成结构化的,易于调用的监控数据 API .

2.2 总体设计

根据 REST 规范的建议,API 设计应该尽量实现 HATEOAS[8] (Hypermedia As The Engine Of Application State). 其理念是客户端与服务器的交互完全在 API 由超媒体动态提供,客户端无需事先了解如何与数据或者服务器交互,从而实现客户端与服务端的部分解耦. 当服务端 API URL 发生了调整时,客户端可以通过服务器提供的资源的表达来智能地发现,而不需要人工介入修改.

基于 HATEOAS 的理念,我们在 API 中提供了 URL 之间的拓扑关系,以供客户端进行推导,见图 1.

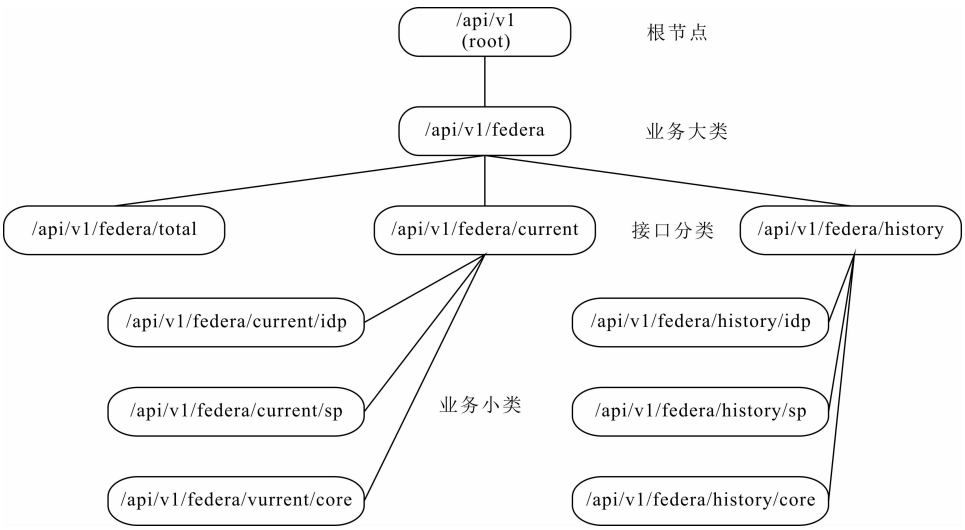

图 1 API URL 拓扑关系
Fig. 1 API URL topological relation

整个树形拓扑中,只有最终的叶节点是我们提供数据的 API 接口. 其他节点都是引导节点,当访问引导节点时,节点会返回它的邻居节点关系来引导前端找到最终的叶节点. 我们以 /api/v1/federa 为例,当访问改路径地址时,引导节点邻居关系的代码如下.

```
{
  "parent": {
```

```
        "apiAddr": "https://example.com/api/v2/",
        "displayName": "API 根目录",
        "type": "root"
      },
      "children": [
        {
          "apiAddr": "https://example.com/api/v2/federa/total/",
          "displayName": "联盟总体状态",
          "type": "end"
        },
        {
          "apiAddr": "https://example.com/api/v2/federa/current/",
          "displayName": "联盟当前状态",
          "type": "mid"
        },
        {
          "apiAddr": "https://example.com/api/v2/federa/history/",
          "displayName": "联盟历史状态",
          "type": "mid"
        }
      ]
    }
```

如拓扑所示，从 API 的根节点进入之后，我们首先对监控的业务大类进行分组. 这里的业务大类分组取决于实际的需要，例如我们可以按照被监控的业务类型来区分，监控服务器的放在一块，监控交换机的放在一块. 也可以根据业务的管理模式来分，网络中心的业务放一块，校园卡中心的业务放一块等等. 在数据展示的 Dashboard 中，这通常相当于一级菜单，API 根据业务的实际需要来抽取数据进行大类分组. 在图 1 的示例中，federa 即分组的业务大类.

在每个业务大类下，我们又进一步的划分了业务小类进行分组. 例如对于交换机业务大类，我们可以根据交换机的部署楼宇，划分为 A 楼，B 楼等多个小类；对于服务器业务大类，我们可以根据实际的业务集群，划分为邮件集群，数据库集群等多个小类；在数据展示的 Dashboard 中，这通常相当于二级菜单，API 根据业务的实际需要来抽取数据进行小类分组. 在图 1 的示例中，业务小类被分为 idp，sp，core 三部分.

对于业务的监控数据，我们设计了三个数据接口，分别提供 Current 数据，History 数据和 Total 数据. Current 即只返回当前状态的监控数据，这些数据可以用于实时的展示业务的当前状态并予以标识；History 则返回一段时间的历史数据，这些数据可以用于业务历史趋势的绘图，展示一段时间的总体业务状态. Total 是根据业务小类的分组，对 Current 数据的封装统计. 这是为了减少前端的 API 请求次数和数值逻辑计算，从而提升 Dashboard 的首屏响应速度.

2.3　请求模式

根据 REST 规范，资源应该通过 URL 来标识. 因此我们通过 URL 来标识业务组，如下所示.

```
/api/v2/{sericeGroup}/current/{service}
/api/v2/{sericeGroup}/history/{service}
/api/v2/{sericeGroup}/total
```

同样的，根据 REST 规范，从服务端获取资源应当使用 GET 方法. 因此 API 的请求 Method 均为 GET .

History 接口所返回的数据量较大，且通常只是针对单个业务的历史数据进行查询和展示. 因此，对于 History 接口，提供基于 GET 方法的查询条件过滤.

```
/api/v2/{ sericeGroup }/history/{ service }?entityID={entityID}
```

2.4　数据结构

数据的结构风格各异，但是从根本上讲，所有的数据都可以由以下三种类型组成：

(1) 映射：即 mapping，也就是 key－value 形式的数据. 也称之为哈希(hash)或者字典(dict). 映射的集合，也就是对象(object).

(2) 数组：即 array，也就是若干个数据按顺序放在一起. 也称之为序列(sequence)或者列表(list).

(3) 标量：即 scalar，表示单个的元素，比如一个数字(number)或者一个字符串(string)

基于 Metadata[9] 的理念，在数据结构中，我们应该避免直接使用数据本身作为 mapping 的 key，而使用 metadata 作为 key 来描述这个数据，将数据本身作为 value 进行描述. 例如，某个描述延迟的指标，他的 metric 名为 url. latency，数据的值为 100. 那么我不应该采用如下所示的数据结构：

```
{
  "url.latency":100
}
```

而应该采取 metadata 方式来描述数据. 例如采取以下的描述方式:

```
{
    "name": "url. latency",
    "displayName": "HTTP 请求延迟",
    "value": 100,
    "status": 1
}
```

这里对象中的 name, displayName, value, status 等 key 都是 metadata,他们只描述数据的类别. 数据本身全部以 value 的方式体现. 最终组成一个数据对象,数据对象或者多个数据对象组成的数组再作为 value 放在上一层级的 metadata key 中. 例如某个业务的监控可能由多个监控共同组成,它的监控状态就可以这样来描述:

```
"status": [
  {
"name": "url. latency",
"displayName": "HTTP 请求延迟",
"value": 100,
"status": 1
  },
  {
"name": "url. http_status_code",
"displayName": "HTTP 请求状态码",
"value": 200,
"status": 1
  }
]
```

请求延迟,请求状态码等多个指标,分别体现在各自的对象中,并组成一个对象数组,作为 status 这个 key 的 value. 其中,对象中的 status 字段用于描述指标的告警状态,其取值范围和告警含义表 1 所示:

表 1 告警状态

Tab. 1 Alarm status

value	level
0	INFO
1	OK
2	WARNING
3	CRITICAL

这些状态除了直接在 current 接口中提供之外,也会在经过统计计算后在 Total 中提供用于首屏展示.

我们对业务对象再进行类似的抽象和标准化,以 current 接口为例,其部分数据示例如下所示.

```
[
  {
    "entityID": "https://idp. ecnu. edu. cn/idp/shibboleth",
    "displayName": "华东师范大学",
    "descriptions": [
      {
        "name": "checkUrl",
        "displayName": "网页探测地址",
         "value": "https://idp. ecnu. edu. cn/idp/profile/Status"
      }
    ],
    "status": [
      {
        "name": "url. latency",
         "displayName": "HTTP 请求延迟",
        "value": 64,
        "status": 1
      },
     ……
    ]
  },
  ……
]
```

由于无论监控数据的指标如何变化,metadata 都不会发生变更,所以数据结构的 key 字段也都不会发生变更. 变化只是数组的长度. 这样就实现了 API 数据结构的标准化,也是 less is more 理念的体现.

3 应用示例

本章以上海市教育身份认监控服务为例,针对 IDP、SP 和 Core 三类服务的监控数据,通过对总览、当前状态以及历史状态三类接口的调用,进行后端的 API 实现和前端的设计开发.

3.1 后端架构

API 的后端设计,要考虑能兼容多个不同的后端数据源,对不同风格的监控数据进行统一的封装,最终以统一的标准化接口提供给前端. 为了提

高 API 的响应速度，我们对 Current 和 Total 的数据进行了缓存[10]. 整体架构如图 2 所示.

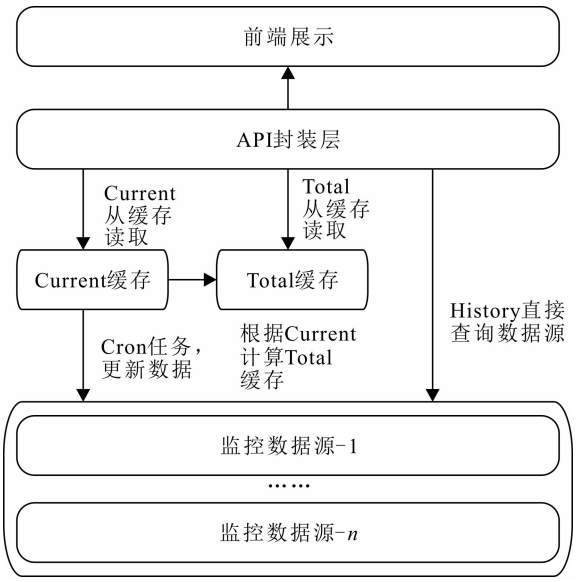

图 2　后端架构
Fig. 2　Backend architecture

在上海市教育身份认监控服务这个案例中，我们采用 Golang[11]进行 API 开发. 其中后端数据源是监控框架 Open-Falcon，因此可以通过调用其数据 API 进行数据读取；缓存我们利用了 Golang 的“sync. Map” 模块来实现带锁的 Map，从而进行安全的数据缓存操作；WEB 服务我们使用了 beego 框架来实现 API 的发布. 整个 API 后端设计上无状态，可以任意的进行横向扩展来满足大规模高性能的需要.

我们通过 wrk 工具对单个后端性能进行了测试，使用 4 进程模拟了 100 个连接对 API 各接口压测了 30 s，其测试结果表 2 所示.

表 2　性能测试

Tab. 2　Performance test

API	Requests/sec	Transfer/sec	Latency—Avg
api/v2/federa/total	17172	11.64 MB	5.85 ms
api/v2/federa/current/idp	3628	118.53 MB	27.60 ms
api/v2/federa/current/sp	8100	36.46 MB	12.41 ms
api/v2/federa/current/core	14303	21.23 MB	7.03 ms
api/v2/federa/history/idp?entityID=DS	843	62.76 MB	119.81 ms

如表 2 所示，history 数据量较大，且受限于后端数据源的吞吐率，其 qps 和延迟性能都相对较低，但是 history 数据不是高频访问，也不做前端的首屏展示，因此该吞吐量已经可以满足需求. 而 current 和 total 都经过了缓存，其性能和延迟都得到了大幅的提升. 其中前端首屏访问所需要的 total 接口，单后端性能达到了 17000+ 以上且延迟低于 10 ms. 能够很好的支持前端的首屏高速响应支持. 由于无状态的设计架构，当请求量进一步增加时，可以简单的通过横向扩展来直接提升后端性能.

3.2　前端架构

前端架构主要采用 Html+Javascript 技术进行页面设计[12]，首页被设定为总览页面，展示当前所有服务监控数据的总览情况，并能通过导航与链接进入查看各类数据的页面；针对 IDP、SP 和 Core 三类服务，需要分别设计三个页面，用来展示三类监控数据的历史状态与当前状态. 整体页面结构如图 3 所示.

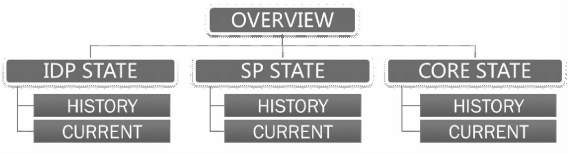

图 3　前端架构
Fig. 3　Frontend architecture

3.2.1 *数据请求和解析*　为了获取接口数据，前端页面需要向后端发送 Http 请求，为了实现局部刷新和快速响应[13]，我们在开发中主要使用 Ajax 技术来实现对后端接口的请求. 关键代码如下.

```
$.ajax({
url: " https://url: port/…/federa/current/idp",
type: "GET",
cache: false,
……
});
```

其中，url 参数是请求接口的地址，代码示例中是 IDP 服务当前状态的接口，需要调用其它接口只需在这里更换接口地址；type 参数为请求方式，当前为 get；cache 参数指定的是否在浏览器中保留缓存，为了获取实时最新数据，这里设置为 false.

当成功从接口中获取数据后，我们须根据具体需求来对获取到的数据进行解析. 以 IDP 监控数据为例，对于监控的当前状态数据，需要针对多个实体(Entity)的每项监控指标进行展现，根据接口定义的标准，将当前状态分为 4 种类型，0 表示普通信息(info)，1 表示正常(ok)，2 表示警告(warning)，3 表示严重错误(critical)，使用 JavaScript 进

行数据解析.关键代码如下:

```
$.each(c, function(i, e) {
    switch (e.status) {
    case 0:_ch.info++;break;
    case 1:_ch.ok++;break;
    case 2:_ch.warning++;break;
    case 3:_ch.critical++;break;
    }
});
```

对于历史型数据,根据接口标准,可以直接获取到数组型的历史数据值.关键代码如下:

```
var c = [];
c = rs[2].historyValues;
```

3.2.2 数据可视化渲染 在解析从接口获取到的各类数据后,再通过可视化的方式来展现数据.其效果如图 4 所示.

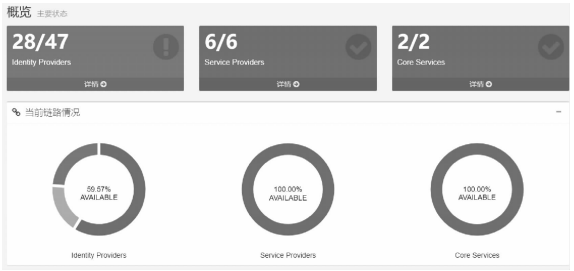

图 4 前端结构设计

Fig. 4 Frontend structural design

4 结 论

传统监控系统的 API 结构各异,前端可视化需要匹配多种后端系统接口,并需要伴随接口更新调整前端代码.这导致监控数据的前端可视化实现复杂,运维困难.本文基于 Metadata 理念,设计并实现了监控数据的 REST API.该设计结构清晰,易于调用,并且支持 HATEOAS,前端能够据此完全自动化的感知后端 API 的变更.最后给出了具体的实现示例,表明基于该 API 设计的监控数据可视化前端结构简单,易于开发,非常适合大屏 Dashboard 展示.后续我们将进一步完善 API 设计和应用,让它在更多的场景下发挥它的价值.

参考文献:

[1] 喻莹莹, 李新, 陈远平. 前后端分离的终端自适应动态表单设计 [J]. 计算机系统应用, 2018, 4: 70.

[2] 夏凌云, 龚文涛. 基于 Web Service 和 REST API 的智慧校园 SOA 基础框架设计 [J]. 微型电脑应用, 2016, 9: 51.

[3] Gardner H J, Karia R, Manduchi G. A web-based, dynamic metadata interface to MDSplus [J]. Fusion Eng Des, 2007, 83: 448.

[4] 刘韵洁, 张娇, 黄韬, 等. 面向服务定制的未来网络架构 [J]. 重庆邮电大学学报: 自然科学版, 2018, 30: 1.

[5] 贾宝军, 徐雷, 郭玉华, 等. 跨数据中心的统一监控研究与实现 [J]. 电信科学, 2016, 3: 1.

[6] 潘庆和. 一种通用分布式数据抓取系统的设计与实现 [J]. 哈尔滨商业大学学报: 自然科学版, 2016, 3: 307.

[7] 刘云, 袁浩恒. 数据挖掘中并行离散化数据准备优化 [J/OL]. 四川大学学报: 自然科学版, 2018, 55: 993.

[8] 李彬峰, 陈建国, 朱明敏. 基于 REST 架构的电子村务平台的设计与实现 [J]. 现代计算机, 2016, 15: 61.

[9] 吴琰, 唐小明. 基于 HBase 的分布式空间数据库技术 [J]. 吉林大学学报: 理学版, 2016, 54: 1355.

[10] 陈维兴, 张天娇, 林家泉, 等. 基于移动 agent 的机坪机会传输控制方法 [J]. 江苏大学学报: 自然科学版, 2018, 39: 309.

[11] 刘艳平. Go 语言实现数据库驱动的方法 [J]. 计算机与现代化, 2018, 1: 113.

[12] 颜仁喆, 高晓阳, 李红岭, 等. 移动式葡萄田间远程监控系统设计与研制 [J]. 四川大学学报: 自然科学版, 2017, 54: 1177.

[13] 吕国勇, 史祥龙. 基于嵌入式 Linux 和 Ajax 技术的 Web 异步交互设计 [J]. 计算机应用, 2013, 33(S1): 247.

引用本文格式:

中 文: 冯骐, 马晨辉, 沈富可. 基于 Metadata 的监控数据 API 设计与实现 [J]. 四川大学学报: 自然科学版, 2019, 56: 431.

英 文: Feng Q, Ma C H, Shen F K. Design and implementation of REST API for monitoring data based on Metadata [J]. J Sichuan Univ: Nat Sci Ed, 2019, 56: 431.