****

**专业学位硕士学位论文**

**学位类别：工程硕士**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **题目：** | **微服务架构模式研究及其在车联网平台中的应用** | | | |
|  | | | |
| **英文并列题目：** | **Research on Microservices architecture and** | | | |
| **it’s application in IOV platform** | | | |
|  | | | |
| **研究生：** | **郝昕悦** | **专业名称：** | **物联网技术** | |
| **研究方向：** | **计算机技术** | | | |
| **导师①姓名：** | **李志华** | **职称：** | **副教授** | |
| **导师②姓名：** | **林其光** | **职称：** | **高级研究员** | |
| **学位授予日期：** |  | | | |
|  |  | **答辩委员会主席：** | |  |

**江 南 大 学**

**地址：无锡市蠡湖大道1800号**

二○○捌年叁月

**独 创 性 声 明**

**本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含本人为获得江南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。**

|  |  |  |  |
| --- | --- | --- | --- |
| **签名：** |  | **日期：** |  |

**关于论文使用授权的说明**

**本学位论文作者完全了解江南大学有关保留、使用学位论文的规定：江南大学有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅，可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文，并且本人电子文档的内容和纸质论文的内容相一致。**

**保密的学位论文在解密后也遵守此规定。**

|  |  |  |  |
| --- | --- | --- | --- |
| **签名：** |  | **导师签名：** |  |
|  |  | **日期：** |  |

摘 要

随着IT业务的发展，软件功能的不断的迭代，系统变得越来越复杂。无论从开发角度，还是运维角度都面临着严重的问题，单体系统的迭代变的举步维艰。随着微服务技术的兴起和成熟，让业界看到了希望。虽然微服务或者微服务架构这一个概念提出已经有很长一段时间了，但在最近却开始频繁地出现在各种场合。微服务和微服务架构是一种特定的软件程序设计思想，它是将大型软件拆分为多个独立可部署的服务组合而成的一整台方案。虽然微服务或者微服务架构的确切定义还存在争议，但并不妨碍其在众多企业的实际应用中被实践和应用。并体现出了可靠性、扩展能力以及安全性等优势。

本文从微服务的技术演进历程出发，介绍了微服务的架构理念和核心概念，通过和单体应用优缺点的比较，提出了软件系统在不同发展阶段的架构选择的一些见解。微服务的治理是微服务架构的核心内容。因为随着微服务系统的不断复杂化，对海量服务接口的管理、服务的发现和注册、性能调优、以及容错隔离方面都需要精耕细作。在第二章里，文章将针对微服务治理中每一个技术要点进行详细的阐述，特别是微服务的容错降级和服务限流方面做了较为深入的研究，分析了常见的算法和实现细节。但是微服务在工业产品中的真正落地还需要很多辅助工作，其中包含微服务框架、服务拆分，服务编排和自动化运维等等，这些内容也在这一章节中进行论述。

在第三和四章节通过实战项目车联网应用平台来实践微服务架构。介绍了怎样从一个已有的单体应用系统重构成一个基于微服务架构的应用系统。其中包含相关技术的选型，服务和数据库的拆分，前后端代码的实现以及数据模型设计等具体内容。

微服务架构是单体应用架构的延伸，而不是替代品。微服务架构应该在软件发展过程中特定的时期被引进。目前微服务框架仍有其局限性，配套工具的不完善，系统复杂度增加等等内容将会在最后一节中阐述。

**关键词**：微服务；服务治理；限流；熔断；车联网

Abstract

With the development of IT services and the continuous iteration of software functions, the system becomes more and more complex. Both from the angle of development and the angle of operation and maintenance are faced with serious problems, and the iteration of the single system is difficult. With the rise and maturity of micro - service technology, the industry has seen the hope. Although the concept of microservice or microservice architecture has been proposed for a long time, it has recently begun to appear frequently in various situations. Micro service and micro service architecture is a specific software programming idea. It is a whole solution that split large software into multiple independent and deployable service combinations. Although the precise definition of micro service or microservice architecture is still controversial, it does not prevent it from being practiced and applied in the practical application of many enterprises. It also embodies the advantages of reliability, expansion and security.

Starting from the technological evolution process of micro service, this paper introduces the concept and core concept of micro service, and puts forward some views on the selection of software architecture in different stages of development through comparing with the advantages and disadvantages of single application. The governance of micro services is the core of the micro service architecture. Because with the micro service system has become more complicated, the massive service interface management, service discovery and registration, performance tuning, and fault isolation are intensive and meticulous farming. In the second chapter, the article will elaborate on every technology point of micro service governance, especially the fault tolerance and service flow limitation of micro service, and analyze the common algorithm and implementation details. However, the real landing of micro services in industrial products still needs a lot of auxiliary work, including micro service framework, service split, service orchestration, automated operation and maintenance, etc. these contents are also discussed in this chapter.

In the third and fourth chapter, the micro service architecture is practiced by the practical platform of the actual vehicle networking application. This paper introduces how to reconstitute an application system based on a microservice architecture from an existing single application system. It includes the selection of related technology, the separation of service and database, the realization of front and back code and the design of data model.

The micro service architecture is an extension of a single application architecture, not a substitute. The micro service architecture should be introduced at a specific time in the development process of the software. At present, the micro service framework still has its limitations, the imperfect supporting tools, the increasing complexity of the system and so on will be elaborated in the last section.m.

**Keywords:**micro service; service governance,Rate Limit; fallback;IOV

目 录

[摘 要 3](#_Toc523727034)

[第一章 绪论 8](#_Toc523727035)

[1.1 本课题的研究目的与意义 8](#_Toc523727036)

[1.2 国内外车联网平台的现状 9](#_Toc523727037)

[1.3 本文的内容安排 9](#_Toc523727038)

[2 第二章 微服务架构研究 11](#_Toc523727039)

[2.1 什么是微服务架构 11](#_Toc523727041)

[2.2 微服务与SOA架构 11](#_Toc523727042)

[2.2.1 微服务与SOA的一些观点 13](#_Toc523727043)

[2.2.2 正确的理解微服务与SOA 14](#_Toc523727044)

[2.3 微服务架构与单体应用架构 16](#_Toc523727045)

[2.3.1 重新认识单体应用 16](#_Toc523727046)

[2.3.2 单体应用的优点 17](#_Toc523727047)

[2.3.3 单体应用的缺点 17](#_Toc523727048)

[2.3.4 微服务架构的优点 18](#_Toc523727049)

[2.3.5 微服务架构的缺点 19](#_Toc523727050)

[2.4 微服务的治理 19](#_Toc523727051)

[2.4.1 为什么要服务治理 20](#_Toc523727052)

[2.4.2 服务治理的技术演进历程 20](#_Toc523727053)

[2.4.3 API网关 23](#_Toc523727054)

[2.4.4 服务发现 23](#_Toc523727055)

[2.4.5 服务注册 27](#_Toc523727056)

[2.4.6 服务降级 29](#_Toc523727057)

[2.4.7 服务的限流 30](#_Toc523727058)

[2.4.8 服务优先级调度 33](#_Toc523727059)

[2.4.9 集群容错 34](#_Toc523727060)

[2.4.10 服务上线审批下线通知机制 35](#_Toc523727061)

[2.5 微服务的自动化测试 35](#_Toc523727062)

[2.6 微服务的跟踪 35](#_Toc523727063)

[2.6.1 跟踪原理 35](#_Toc523727064)

[2.6.2 抽样收集 36](#_Toc523727065)

[2.7 微服务和DevOps 36](#_Toc523727066)

[2.8 配置中心 37](#_Toc523727067)

[2.9 服务监控 37](#_Toc523727068)

[2.10 本章小结 38](#_Toc523727069)

[3 第三章 车联网项目需求分析 39](#_Toc523727070)

[3.1 现存的问题 39](#_Toc523727072)

[3.2 设计目标 39](#_Toc523727073)

[3.2.1 关键需求属性 39](#_Toc523727074)

[3.2.2 关键质量属性 40](#_Toc523727075)

[4 第四章 车联网架构项目实战应用 41](#_Toc523727076)

[4.1 设计原则 41](#_Toc523727077)

[4.1.1 弹性架构 41](#_Toc523727078)

[4.1.2 避免单点系统 41](#_Toc523727079)

[4.1.3 可模块化测试 41](#_Toc523727080)

[4.1.4 标准化原则 41](#_Toc523727081)

[4.1.5 可靠性原则 42](#_Toc523727082)

[4.1.6 高性能、大容量原则 42](#_Toc523727083)

[4.2 系统设计 42](#_Toc523727084)

[4.2.1 系统架构图 42](#_Toc523727085)

[4.2.2 逻辑分层 43](#_Toc523727086)

[4.2.3 微服务技术选型 45](#_Toc523727087)

[4.3 平台实现 47](#_Toc523727088)

[4.3.1 业务拆分逻辑 47](#_Toc523727089)

[4.3.2 前端的实现 47](#_Toc523727090)

[4.3.3 后端服务的实现 54](#_Toc523727091)

[4.3.4 数据库设计 70](#_Toc523727092)

[4.3.5 服务治理 71](#_Toc523727093)

[4.3.6 服务的限流 73](#_Toc523727094)

[4.3.7 服务的熔断 74](#_Toc523727095)

[4.3.8 配置中心 76](#_Toc523727096)

[4.3.9 发布与部署 78](#_Toc523727097)

[4.3.10 系统安全性解决方案 79](#_Toc523727098)

[4.3.11 质量 81](#_Toc523727099)

[5 第五章 主要结论与展望 90](#_Toc523727100)

[5.1 主要结论 90](#_Toc523727102)

[5.2 展望 91](#_Toc523727103)

[致 谢 91](#_Toc523727104)

[参考文献 92](#_Toc523727105)

[附录 94](#_Toc523727106)

[5.3 数据库设计 94](#_Toc523727107)

[5.3.1 车辆基础信息表(TB\_VEHICLE) 94](#_Toc523727108)

[5.3.2 车辆型号型表(TB\_VEHICLE\_MODEL) 104](#_Toc523727109)

[5.3.3 组织信息表(TB\_ORGANIZATION) 109](#_Toc523727110)

[5.3.4 车辆品牌表(TB\_VEHICLE\_BRAND) 117](#_Toc523727111)

[5.3.5 车辆轨迹信息表(TB\_VEHICLE\_TRACK\_RECORD) 119](#_Toc523727112)

# 第一章 绪论

## 本课题的研究目的与意义

什么的是微服务架构，怎样做好微服务架构是目前业内对于平台架构风格非常流行的讨论话题。

2012年，微服务的概念首次被提出，提出人是作为面向对象分析与设计、uml、模式等领域的专家Martin Fowler，他也是敏捷开发方法的创始人之一。微服务架构在业务、自动化部署、终端智能和分散的语言和数据控制方面具有一些共同的新特性。这也是一种在云中部署应用程序和服务的新技术。随着云计算技术的进步和服务的增长，微服务架构越来越受到关注，虽然有很多不同的论据，但是微服务架构模型对于敏捷部署和复杂企业应用程序的实现提供了很大的帮助。

一般来说我们的开发逻辑是模块化的设计思想，在程序完成后，将打包并部署到特定的应用程序上。每个特定的格式取决于相应的语言和框架。例如，java的web应用程序通常包装为war格式，部署在tomcat或者jetty，其他的一些java应用程序会被打包成一个jar包的格式。此类应用程序开发风格非常普遍，易于调试，它只需要简单地运行这个应用程序。并且通过与UI连接的一些工具，可以完成端到端测试。打包的应用程序被会复制到服务器端，通过在负载均衡器后端运行多个副本，可以很容易地实现应用程序扩展。在刚开始的时候，这种应用程序运行良好，但随着时间的推移，简单的应用程序将逐渐变得复杂。几年之后，原本简单的小应用程序就会变成一个庞然大物，这导致开发团队非常痛苦，开发和部署困难重重，主要的问题是应用程序过于复杂，以至于任何单个开发人员都无法完全掌握它。从而导致修复错误并正确添加新功能将是非常困难和耗时的。同时，应用程序越大，启动时间越长，大部分时间都花在等待上，生产效率会受到很大的影响。此外，在同一个应用系统中在不同模块发生资源冲突时，扩展将会非常困难。而可靠性会是这类架构最头痛的问题。当所有模块都运行在一个进程中，任何一个模块中的一个缺陷，比如内存泄露，将会有可能影响到整个进程，导致进程崩溃推出。

微服务处理结构模式则可以很好的解决上述问题。微服务架构的思想不是开发一个复杂巨大的应用，而是将应用分解为若干小的、互相连接的微服务。除了各微服务能够单独进行部署与规模伸缩之外，每项服务还具备牢固的模块边界，甚至允许我们在不同的服务当中使用不同的编程语言进行代码编写。另外，各服务亦可由不同团队负责管理。

本论文研究的目的，就是通过对微服务架构风格的研究，在对一些中小型软件系统向大型系统架构演进的过程中，提供一种可选的架构风格模式。分析其优点和弊端，以及指明在实施过程中需要注意的问题。并通过一个案例：车联网应用平台，来具体实现微服务架构。

## 国内外车联网平台的现状

“微服务架构”概念的提出已经有很长一段时间了，但在最近几年却开始频繁地出现。微服务架构是一种特定的软件应用程序设计方式，它将大型软件拆分为多个独立可部署服务组合而成的套件方案。虽然这种架构风格的确切定义还存在争议，但并不妨碍其在众多企业的实际应用中被实践，并体现出了具备通用特征的业务功能、自动化部署、端点智能化以及对语言与数据的离散化控制能力。

在2014年，Sam Newman，Martin Fowler在ThoughtWorks的一位同事，出版了一本新书《Building Microservices》。该书描述了如何按照Microservice架构模式设计及搭建一个具有良好扩展性并可持续开发的系统。除此之外，该书还将基于该模式的系统演化流程与Continuous Delivery等当前甚为流行的开发流程结合在了一起，使得Microservice架构模式看起来非常具有吸引力。基于这些原因，该架构模式迅速被业界所熟知，并在多个产品中被尝试着使用。

实际上目前很多大公司已经采用了微服务技术作为技术Web服务，如Google、Amazon等。在Java语言中重要的两个以SOAP技术开始的网络服务框架XFire和Axis也把基于REST的微服务作为自己的另一种选择。

## 本文的内容安排

1. 微服务的概念、历史和由来。

追本溯源，了解微服务的历史和发展过程，指导后续的理论研究。

1. 研究微服务架构的特性。

也许我们无法给微服务架构风格出具一条确切的定义，但我们却可以根据该架构表现出的各类共同特性对其加以描述。

1. 研究微服务架构与传统单体应用架构的优缺点对比

虽然微服务架构相较与传统的单体应用架构会有一些优势，但是，它也和其它技术一样，在某些特定的场景下有其自身的不足。在这一小节中将会为读者在做架构选择的时候提供详尽的理论依据。

1. 微服务架构的核心功能

微服务的核心优势需要有一些相应的技术进行支撑，在这一小节，将介绍和研究实现这些功能的算法和逻辑。

1. 微服务和运维

微服务架构的复杂程度大大高于单体应用，动辄数十台，甚至上百台机器的集群。传统的运维方式已经很难满足微服务管理的需求了。自动化运维，以及DevOps这些理念和实施方式将如何更好的配置微服务架构的实施是微服务架构能否成功的关键。

1. 项目落地

在一个完整的物联网项目（车联网平台）中使用微服务理念来构建整体架构。

# 第二章 微服务架构研究



## 什么是微服务架构

微服务架构概念的提出已经有很长一段时间了，但在最近几年却开始频繁地热炒。微服务架构是一种特定的软件应用程序设计方式，它将大型软件拆分为多个独立可部署服务组合而成的套件方案。虽然这种架构风格的确切定义还存在争议，但并不妨碍其在众多企业的实际应用中被实践，并体现出了具备通用特征的业务功能、自动化部署、端点智能化以及对语言与数据的离散化控制能力。Docker 作为一种开源的应用容器引擎，帮助开发者将他们的应用以及依赖打包到一个可移植的容器中，便于应用的部署和扩展。而随之产生的微容器概念和微服务正好相辅相成，通过 Docker 封装的应用可以轻松运行在以扩容能力见长的云计算平台上。

实际上，从业界的讨论来看，微服务本身并没有一个严格的定义。不过，ThoughtWorks的首席科学家——马丁·福勒（Martin Fowler）先生，对微服务的这段描述，似乎更加通俗易懂：微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通（通常是基于HTTP的RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。另外，应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

## 微服务与SOA架构

微服务架构思想最重要的一个重点就是要将业务系统彻底的组件化和服务化，原有的单一业务系统将被拆分为多个独立的开发、设计、运营的应用系统。这些小型应用程序通过服务进行交互和集成，甚至从前端web UI到控制层、逻辑层、数据库访问、数据库的每个小应用程序都是完全独立的。除了完成其业务功能本身之外的每一个小应用程序，关键还需要消耗外部暴露于其他应用程序服务，同时它本身也将拥有对服务的外部发布的能力。

微服务架构风格是一种将单个应用程序开发为一套小型服务的方法，每个应用程序都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。这些服务是围绕业务功能构建的，并且可以通过完全自动化的部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，使用不同的数据存储技术。

微服务体系结构不需要是独立的函数或独立的资源，就像普通的服务一样。说微服务需要匹配业务功能是完全正确的。可是，这仍然意味着如果性能模型粒度的设计是错误的，那么我们必须付出很多代价。如果你阅读了Fowler的整篇文章，你会发现这些指导方针是非常实用的。当您决定将所有组件组合在一起时，开发人员需要非常确信这些组件将会改变，并且规模将会改变。服务粒度越粗，越难符合规则。服务的粒度越细，就越灵活，以减少更改和负载的影响。然而，利弊的权衡是非常复杂的，我们需要在配置和资金模型的基础上考虑基础设施的成本。

另外，SOA和微服务之间的核心区别在于大小和范围。正如“微”这个词所暗示的那样，它必须比SOA更小。微服务是一个会更加小型独立部署单元。谨防非常小的微服务反模式-纳米服务。SOA可以是单一的，也可以由多个微服务组成。马丁·福勒说，他喜欢把SOA看作是一套微型服务。

综上所述，微服务架构和SOA架构的主要区别如下：

* 第一，应用本身暴露出来的服务是和应用一起部署的，服务本身并不单独部署，服务本身就是业务组件已有的接口能力发布和暴露出来的。从这里我们就看到一个关键，我们在进行单个应用组件设计的时候，本身在组件内部就会有很大接口的设计和定义，这些接口我们可以根据和外部组件协同的需求将其发布为上面类型的微服务，而如果不需要对外协同我们完全可以走内部API接口访问模式提高效率（微服务有基于Http的，也有基于RPC的，通常RPC接口调用性能更高，适合用于内部服务的调用）。
* 第二，微服务架构本身来源于互联网的思路，因此组件对外发布的服务强调了采用HTTP Rest API的方式来进行。这个也可以看到在互联网开放能力服务平台基本都采用了Http API的方式进行服务的发布和管理。从这个角度来说，组件拥有朝外部暴露的能力才需要发布为微服务，其本身也是一种封装后的粗粒度服务。而不是将组件内部的所有业务规则和逻辑，组件本身的底层数据库CRUD操作全部朝外部发布。否则将极大的增加服务的梳理而难以进行整体服务管控和治理。
* 第三，围绕着业务领域组件来创建应用是微服务的核心思想。这些就应用可独立地进行开发、管理和优化。在分散的组件中使用微服务开源框架使部署、管理和服务功能交付变得更加简单。

### 微服务与SOA的一些观点

对于了解过SOA的人来说，第一次看到微服务这个概念肯定会有所疑惑：为何有了SOA还要提微服务呢？等到简单看完微服务的介绍后，可能很多人更加困惑了：这不就是SOA吗？关于SOA和微服务的关系和区别，大致可以分为下面几个典型的观点：

#### 微服务是SOA的实现方式

如下图所示，这种观点认为SOA是一种架构理念，而微服务是SOA理念的一种具体实现方法。例如，微服务就是使用HTTP RESTFul协议来实现ESB的SOA；使用SOA来构建单个系统就是微服务和微服务就是更细粒度的SOA。



#### 微服务是去掉ESB后的SOA

如图所示，这种观点认为传统SOA架构最广为人诟病的就是庞大、复杂、低效的ESB，因此将ESB去掉，改为轻量级的HTTP实现，就是微服务。



#### 微服务是一种和SOA相似但是本质上不同的架构理念

如下图所示，这种观点认为微服务的SOA只是有点类似，但是本质是不同的架构设计理念。相似点在于下图中交叉的地方，就是两者都关注“服务”，都是通过服务的拆分来解决可扩展问题。本质上不同的地方在于几个核心理念的差异：是否有ESB、服务的粒度、架构设计的目标等等。



### 正确的理解微服务与SOA

以上观点看似都有一定的道理，但是都有点差别，到底哪个才是准确的呢？单纯从概念上是难以分辨的，我来对比一下SOA的微服务的一些具体做法，再来看看到底哪一种观点更加符合实际情况。

* 服务粒度

从整体上来说，SOA的服务粒度要粗一些，而微服务的服务粒度要细一些。例如，对一个大型企业来说，“员工系统系统”就是一个SOA架构中的服务；而如果采用微服务架构，则“员工管理系统”会被拆分为更多的服务，比如“员工信息管理”、“员工考勤管理”、“员工假期管理”和“员工福利管理”等更多服务。

* 服务通信

SOA采用了ESB作为服务间通信的关键组件，负责服务定义、服务路由、消息转换、消息传递，总体上是重量级的实现。微服务推荐使用同一的协议和格式，例如，RESTful协议、RPC协议、无须ESB这样的重量级实现。Martin Fowler将微服务架构服务通信理念称为“Smart endpoints and dumb pipes”，简单翻译为“聪明的终端，愚蠢的管道”。之所以用“愚蠢”两字，其实就是与ESB对比的，因为ESB太强大了，既知道每个服务的协议类型（例如，是XML还是JSON），还知道每个数据的格式（例如，是2017-01-01还是20170101），而微服务的“dumb pipes”仅仅做消息传递，对消息格式和内容一无所知。

* 服务交付

SOA对服务的交付并没有特殊的要求，因为SOA更多考虑的是兼容已有的系统；微服务的架构理念要求“快速交付”，相应的要求采取自动化测试、持续集成、自动化部署等敏捷开发相关的最佳实践。如果没有这些基础能力支撑，微服务的规模一旦变大（例如，超过20个微服务），整体就难以达到快速交付的要求，这也是很多企业在实行微服务时踩过的一个明显的坑，就是系统拆分为微服务后，部署成本呈指数及上升。

* 应用场景

SOA更加适合于庞大、复杂、异构的企业级系统，这也是SOA诞生的背景。这类系统的典型特征就是很多系统已经发展多年，采用不同的企业级技术，有的是内部开发，有的是外部购买，无法完全推倒重来或者进行大规模的优化和重构。因为成本和影响太大，只能采用兼容的方式进行处理，而承担兼容任务的就是ESB。

微服务更加适合于快速、轻量级、基于Web的互联网系统，这类系统业务变化快，需要快速尝试，快速交付；同时基础就是基于Web，虽然开发技术可能差异很大（例如，java、C++，.NET等），但对外的接口基本都提供HTTP RESTful风格的接口，无须考虑在接口层进行类似SOA的ESB那样的处理。

综上所述，我将SOA和微服务对比如下：

|  |  |  |
| --- | --- | --- |
| 对比纬度 | SOA | 微服务 |
| 服务粒度 | 粗 | 细 |
| 服务通信 | 重量级 | 轻量级 |
| 服务交付 | 慢 | 快 |
| 应用场景 | 企业级 | 互联网 |

因此，我们可以看到，SOA和微服务本质上是两种不同的架构设计理念，只是在“服务”这个点上有交集而已，因此两者的关系应该是上面的第三种观点。其实Martin Fowler在他的微服务文章中的三个词：small、lightweight、automated，基本上浓缩了微服务的精华，也是微服务与SOA的本质区别所在。

通过前面详细的分析和比较，似乎微服务本质上就是一种比SOA要优秀很多的架构模式，那是否意味着我们都应该把架构重构为微服务呢？其实不然，SOA和微服务是两种不同理念的架构模式，并不存在优劣，只是应用场景不同而已。我们介绍SOA时候提到其产生历史背景是因为企业的IT服务系统庞大而又复杂，改造成本很高，但业务上有要求其互通，因此才会提出SOA这种解决方案。如果，我们将微服务的架构模式生搬硬套到企业级IT服务系统总，这些IT服务系统的改造成本可能远远超出实施SOA的成本。

## 微服务架构与单体应用架构

单体应用架构常常是我们在讨论微服务架构的时候拿来比较的另一种架构。那是不是微服务架构就一定好于单体应用架构呢？下面，我们将对两种架构的优缺点进行详细的分析和比较。我们会看到，单体应用架构和微服务架构其实是软件系统由小到大，由简单到复杂过程中所采用的，他们是在不同的软件系统发展阶段中的选择。

### 重新认识单体应用

我们重新来认识一下单体应用架构。通俗地讲，“单体应用（monolith application）”就是将应用程序的所有功能都打包成一个独立的单元，如果是使用Java语言编写的程序的话，可以是JAR、WAR、EAR或其它格式，如果是C#或者C++写的程序的话，那也许是一个可以执行的exe文件或应用。就像下图一样，单体应用虽然内部也有可能有多个模块构成，但是它的物理部署就是一个独立的程序。而右边，微服务，则有很多可以独立部署和运行的完整运行单元。

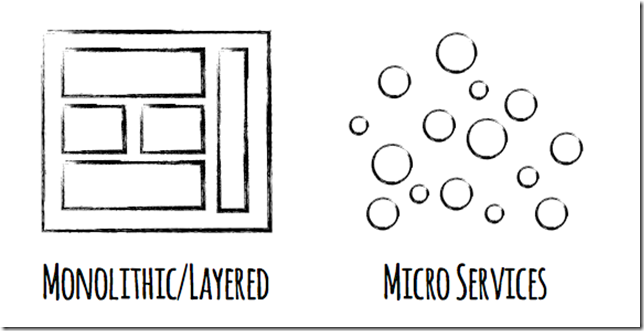


图 2‑1

### 单体应用的优点

单体应用作为最传统的架构方式有着如下优点：

* 为人所熟知：现有的大部分工具、应用服务器、框架和脚本都是这种应用程序；
* IDE友好：像NetBeans、Eclipse、IntelliJ这些开发环境都是针对开发、部署、调试这样的单个应用而设计的；
* 便于共享：单个归档文件包含所有功能，便于在团队之间以及不同的部署阶段之间共享；
* 易于测试：单体应用一旦部署，所有的服务或特性就都可以使用了，这简化了测试过程，因为没有额外的依赖，每项测试都可以在部署完成后立刻开始；
* 容易部署：只需将单个归档文件复制到单个目录下；
* 开发效率高：只在一个项目中开发，省去跨进程开发的协议指定，联调，开发等待等问题。

### 单体应用的缺点

迄今为止，单体应用架构已经帮助我们解决了很多问题，同时未来还会继续发挥其重要的作用。但是，不管如何优化，单体应用最终都会因为系统复杂性的增加，团队壮大，性能问题的凸显，应用范围扩展等出现问题。下面列举了一些单体应用的缺点：

* 不够灵活：对应用程序做任何细微的修改都需要将整个应用程序重新构建、重新部署。开发人员需要等到整个应用程序部署完成后才能看到变化。如果多个开发人员共同开发一个应用程序，那么还要等待其他开发人员完成了各自的开发。这降低了团队的灵活性和功能交付频率；
* 妨碍持续交付：单体应用可能会比较大，构建和部署时间也相应地比较长，不利于频繁部署，阻碍持续交付。在移动应用开发中，这个问题会显得尤为严重；
* 受技术栈限制：对于这类应用，技术是在开发之前经过慎重评估后选定的，每个团队成员都必须使用相同的开发语言、持久化存储及消息系统，而且要使用类似的工具，无法根据具体的场景做出其它选择；
* 技术债务：“不坏不修（Not broken，don’t fix）”，这在软件开发中非常常见，单体应用尤其如此。系统设计或写好的代码难以修改，因为应用程序的其它部分可能会以意料之外的方式使用它。随着时间推移、人员更迭，这必然会增加应用程序的技术债务。

### 微服务架构的优点

而随着业务需求的快速发展变化，稳定性、灵活性、可扩展性等非功能需求需求不断增长，迫切需要一种更加快速高效的软件架构方式。微服务架构就很好的契合了这种架构需求。在微服务架构中单体应用被拆分成多个更小的服务，每个服务有自己的程序，独立部署，然后共同组成一个应用程序群。微服务的微并不是针对代码行数而言，而是针对业务需求，把服务的范围限定到单个功能，提供更细粒度的服务。微服务有如下特点：

* 领域驱动设计：应用程序功能分解可以通过Eric Evans在《领域驱动设计》中明确定义的规则实现；每个团队负责与一个领域或业务功能相关的全部开发；团队拥有全系列的开发人员，具备用户界面、业务逻辑和持久化存储等方面的开发技能；
* 单一职责原则：每个服务应该负责该功能的一个单独的部分，这是SOLID原则之一；
* 明确发布接口：每个服务都会发布一个定义明确的接口，而且保持不变；服务消费者只关心接口，而对于被消费的服务没有任何运行依赖；
* 独立部署、升级、扩展和替换：每个服务都可以单独部署及重新部署而不影响整个系统。这使得服务很容易升级，每个服务都可以沿着《Art of Scalability》一书定义的X轴和Z轴进行扩展；
* 可以异构/采用多种语言：每个服务的实现细节都与其它服务无关，这使得服务之间能够解耦，团队可以针对每个服务选择最合适的开发语言、持久化存储、工具和方法；
* 轻量级通信：服务通信使用轻量级的通信协议，例如，同步的REST，异步的AMQP、STOMP、MQTT等。

### 微服务架构的缺点

同样微服务架构也并非那么完美，我们在做技术选型时，也应该充分考虑它的不足之处：

* 运维成本过高：微服务通常会有大量的服务进程，在添加了负载均衡与消息中间件后，进程的数量还会持续增加。运维与编排所有这些服务是个令人望而却步的任务。实现所有这些需求需要高质量的监控与运维基础设施。常常需要监控几百台服务器，几百个进程。磁盘、CPU、内存不等等，还要保证性能。这真是一个非常棘手的任务。这个过程对运维的要求会非常高，实施自动化运维是必然的选择。
* 多版本维护：服务依赖于彼此间的接口进行通信。改变一个服务的接口会对其他服务造成影响，修改了契约一方的语法或是语义，那么所有其他服务都需要理解这个改变。在微服务环境下，这意味着简单的交叉改变会导致很多不同的组件发生变化，这些组件需要以一种协调一致的方式进行发布。 一般来说可以通过向后兼容的方式避免这些变化的发生，不过实际情况却是业务驱动的需求让我们没法实现分阶段的发布。
* 分布式系统的复杂性：作为一种分布式系统，微服务引入了复杂性和其他若干问题，比如说网络延迟、容错性、消息序列化、不可靠的网络、异步、版本化、应用层中的负载变化等等。
* 异步带来的问题：微服务常常会使用异步编程、消息与并行，如果要求某个操作必须是同步且具有事务的，那么这就非常复杂了，这要求我们得管理好相关联的ID以及分布式事务，将各种动作绑定在一起。
* 测试的难度增大：使用微服务架构时，测试是另一个需要考虑的问题，因为无论是手工测试还是自动化测试，我们都很难以一致的方式重现环境。当添加了异步与动态消息负载后，要测试以这种风格构建的系统就难上加难了，同时也无法对将要发布到生产的各种服务抱有信心。 我们可以测试每个服务，不过在这种动态环境下，非常微妙的行为都会从服务间的交互中产生出来，这是难以做到可视化的，也不易想清楚，更不必说全面的测试了。
* 初期开发效率下降：当我们面对紧张的研发工期时，使用微服务架构就像是一种噩梦。开发人员再也不能像开发单体应用的时候从头写到尾，但必须新建微服务的项目，制定消息交换的协议等等，大大降低了开发效率，也使研发内测或者单元测试变得复杂。

## 微服务的治理

我们可认为SOA governance（服务治理），是企业为了保证事情顺利完成而实施的过程，其中包括架构原则、治理规程、最佳实践、以及规律。服务治理指的是用来管理大量微服务接口的方式和实现的过程。

### 为什么要服务治理

用微服务架构的系统发展到最后常常会有成百上千个服务接口和数十乃至上百个独立的应用系统。这么大一个庞大的系统将会面临业务和技术方面的需求，特别是非功能性方面的需求。

随着系统越做越大以及业务需求的发展，系统的微服务会越来越多。这对如何协调线上运行的各个服务，保障服务的稳定性来说是一个不小的挑战。也是系统服务架构和运维人员所需要面对的关键问题。随着业务规模的不断扩大，小服务资源浪费等问题逐渐显现，服务需要占用数百台服务器，但是有些小服务占用的资源又比较少，所以需要根据数据进行容量规划，合理分配各个服务的资源占用，以提高服务器的资源利用率。当线上业务发生故障或者流量突变时，同样需要对故障业务做服务降级、流量控制等，以快速应对这样的情况。

在开发过程中，随着服务的越来越多，大量的服务与服务之间依赖变得错综复杂。开发人员，甚至是架构师都很难记清楚一个服务有哪些依赖，导致修改或者发版的时候会出现很多问题。服务的上下线也会出现很多问题。为了规范服务的上下线，在服务发布前需要走服务发布的流程，由架构师或者项目经理对需要上线的服务做进行审核，审核通过的才能够上线。为了满足上述这些需求，需要有一个统一的服务治理框架对服务进行统一管控和管理。

另外，在大型企业中，特别是在企业数字化转型的过程中，遗留下来的传统应用非常复杂，如果需要升级就必须要解耦，用以降低系统改造和演进的复杂度。主要的核心系统应该专注于稳定性高和变化小的业务。而一些不是核心功能的系统可以通过外部的系统或者第三方系统来继承。通过一些系统的拆分，把能力开放出来等技术手段来满足敏捷调整的需求。

### 服务治理的技术演进历程

#### 第一代服务治理 SOA Governance

最早的微服务架构是以IBM为首的SOA提供商推出的针对企业IT系统的服务治理框架，这个框架主要关注在企业IT系统中那些异构系统的质量、发布审批流程和服务的建模、开发、测试。同时关注这些异构系统的全生命周期管理。

这一代的服务治理，我们称之为SOA Service Governance。可以认为是第一代的服务治理框架。SOA Governance的定位是面向企业IT系统异构服务的治理和服务生命周期管理。它是专门为SOA服务治理而生，SOA和微服务的区别在上文已经提到。传统的SOA Governance或者说SOA服务治理主要包含四部分内容：

* 服务建模：验证功能需求与业务需求，发现和评估当前服务，服务建模和性能需求，开发治理规划。
* 服务组装：创建服务更新计划，创建和修改服务以满足所有业务需求，根据治理策略评估服务，批准组装完成。
* 服务部署：确保服务的质量，措施包括功能测试、性能测试和满足度测试，批准服务部署。
* 服务管理：在整个生命周期内管理和监控服务，跟踪服务注册表中的服务，根据服务质量等级协议（SLA）上报服务的性能KPI数据进行服务质量管理。

第一代的传统的SOA Governance的主要缺点如下：

* 分布式服务框架的发展，内部服务框架需要统一，服务治理也需要适应新的架构，能够由表及里，对服务进行细粒度的管控。
* 微服务架构的发展和业务规模的扩大，导致服务规模量变引起质变，服务治理的特点和难点也随之发生变化。
* 缺少服务运行时动态治理能力，面对突发的流量高峰和业务冲击，传统的服务治理在响应速度、故障快速恢复等方面存在不足，无法更敏捷应对业务需求。

#### 第二代服务治理分布式服务框架

随着电子商务和移动互联网近十年的快速发展，诞生了全新服务治理理念，它基于电子商务平台的统一分布式服务框架，并且聚焦于对内部同构服务的线上治理，同时保障线上服务的运行质量。和传统的基于第一代的IT架构服务治理相比较，第二代的理念在服务架构的开发模式、部署规模、业务特点等做了较大的改变，新的架构理念的重点也从线下转移到了线上服务的质量保障。

这一代的服务治理更加关注互联网业务，其重点在对内部采用统一服务框架服务化的业务运行态以及敏捷治理体系的粒度大小。

* 治理范围：基于统一分布式服务框架开发的业务服务，与协议本身无关，治理的可以是SOA服务，也可以是基于内部服务框架私有协议开发的各种服务。
* 治理策略：针对互联网业务的特点，例如突发的流量高峰、网络延时、机房故障等，重点针对大规模跨机房的海量服务进行运行态治理，保障线上服务的高SLA，满足用户的体验。常用的治理策略包括服务的限流降级、服务迁入迁出、服务动态路由和灰度发布等。

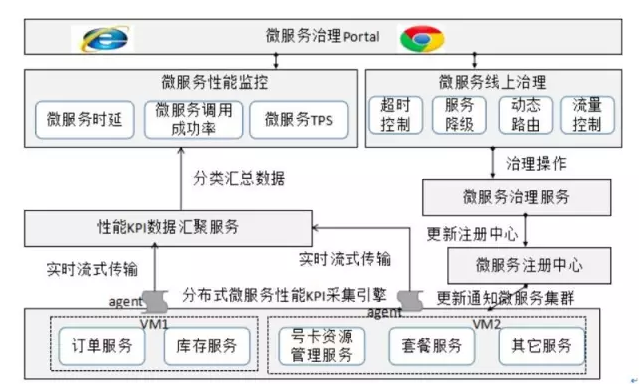


图 2‑2

#### 第三代以云计算为核心的云端微服务治理架构

近几年，随着微服务和云计算技术的快速发展，以亚马逊aws为首的基于微服务架构云服务化的云端服务治理体系诞生，微服务自治是它的核心理念，通过自动化调度的弹性和敏捷，逐渐消除人工的服务治理，达到全面自动化。这种微服务架构的治理可以实现微服务在一定程度上的自治，比如服务独立打包、独立部署、独立升级、独立扩容、独立降级等等。它把传统云计算的弹性伸缩、单点故障迁移、服务健康度管理和自动容量规划等措施和微服务治理相结合，逐步实现微服务的自治。

随着云计算的发展以及DevOps的逐渐盛行，基础设施的服务化（IaaS）为大规模、批量流水线式软件交付提供了便利，亚马逊做为全球最大的云计算解决方案提供商，特别是在微服务云化开发和治理方面积累了很多宝贵的经验，主要包含如下内容：

* 把全公司的开发环境统一服务化，统一运行平台，快速高效服务开发；
* 规定所有后端应用必须服务化，系统由多项服务化组件构成。
* 规定所有的服务必须共享、原子化、重用。
* 各自管理自己的服务。包括负责服务开发、测试、部署和治理，运维整个生命周期支撑。
* 开发的服务需要高度自动化和DevOps支持，一键式服务部署和回退。
* 超大规模支持：后台几十万个服务，成千上万开发者同时使用，平均每秒钟有1-2个服务部署。
* 基于容器化部署的微服务架构。

### API网关

系统拆分为微服务后，内部的微服务之间是互联互通的，相互之间的访问是点对点的。如果外部系统想调用系统的某个功能，也采取点对点的方式，则外部系统会非常“头大”。因为在外部系统看来，它不需要也没有办法理解这么多微服务的职责分工和边界，它只会关注它需要的能力，而不会关注这个能力应该由哪个微服务提供。除此之外，外部系统访问系统还涉及到安全和权限相关的限制，如果外部系统直接访问某个微服务，则意味着每个微服务都要自己实现安全和权限的功能，这样做不但工作量大，而且都是重复的工作。

综合上面的分析，微服务需要一个统一的API网关，负责外部系统的访问操作。API网关是外部系统的访问接口，所有的外部系统接入系统都需要通过API网关，主要包括接入鉴权（是否允许接入）、权限控制（可以访问哪些功能）、传输加密、请求路由、流量控制等功能。

### 服务发现

在我们现代基于云的微服务应用中，服务实例的网络地址是动态分配的。而且由于自动扩展，服务实例的配置也经常变化。为了满足客户端向服务发送请求的需要，必须要实现服务发现这一机制。

微服务架构中有两种主要的服务发现模式：客户端服务发现（client-side discovery）和服务器端服务发现（server-side discovery）。

#### client-side discovery

当使用client-side discovery的时候，client端负责决定可用的服务实例网络地址，以及和他们相关的负载均衡。client端向服务注册表（service registry）发送一个请求的时候，服务注册列表是一个存储了可用服务实例的数据库。Client端使用一种负载均衡算法去选择一个可用的服务实例来响应这个请求，下图展示了这种模式的架构：

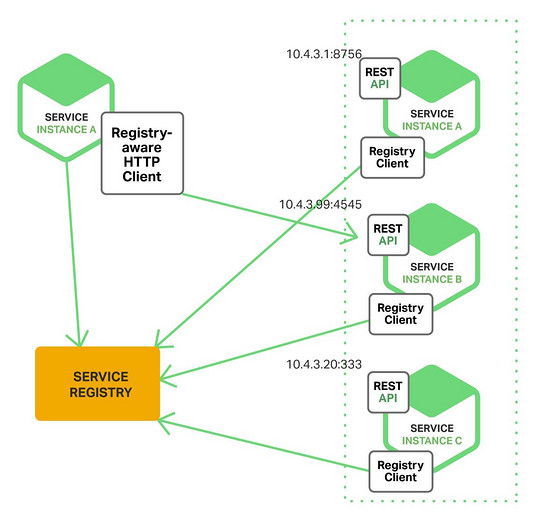


图 2‑3

每当一个服务实例被系统启动的时候，它的IP地址会被写到存储网络地址的注册表上；当服务实例结束运行的时候，再从这个注册表中删除。这个服务实例的注册表是通过心跳机制来实现动态刷新。

著名的Netflix OSS给我们提供了一个client-side discovery模式的好例子。Netflix Eureka是一个服务注册的列表，使用的REST API用来管理服务实例的注册和查询可用的实例。Netflix Ribbon是一个IPC Client，它和Eureka一起处理可用服务实例的负载均衡策略。

client-side discovery模式有它的优势同时也有其缺点。这种模式相对简单和直接，除了服务注册表，其它动态的部分就没有了。同时，由于Client端存储可用的服务实例列表，所以它可以做到非常智能的、明确的负载均衡决策，比如一直用hash算法。这种模式的一个重大缺点在于，服务注册表都是实现在客户端的，所以必须为每一种语言的客户端框架实现各自的客户端服务发现逻辑。从而造成一定的冗余。

#### server-side discovery

下图展示了这种模式的架构：

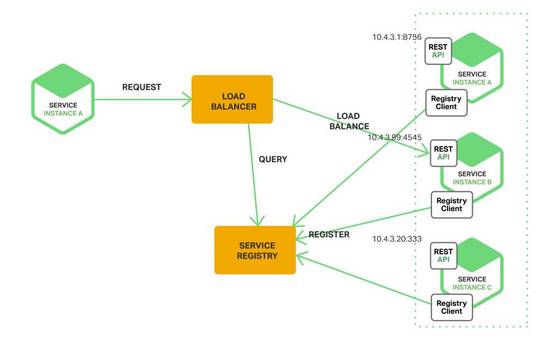


图 2‑4

server-side discovery模式和client-side discovery不同，client端通过负载均衡器向某一个服务发送请求，这个负载均衡器会查询存储在上面的服务注册列表，并将请求路由到可用的服务实例上。通过服务端的服务发现，服务实例在服务注册表上被注册和注销。亚马孙公司的的AWS的ELB（Elastic Load Blancer）就是使用的服务器发现策略的路由器。这个负载均衡起通常被用来均衡互联网的外部流量和流向VPC（Virtual Private Cloud）的流量。当一个客户端通过AWS的ELB发送请求（HTTP或TCP）的时候，使用的是域名系统，ELB会均衡这些注册的EC2实例或ECS（EC2 Container Service）容器的流量，如果没有另外的服务注册列表，EC2实例和ECS容器会在ELB上进行注册。我们常见的HTTP服务器和类似Nginx、Nginx Plus的负载均衡器也可以被用做服务器端服务发现负载均衡器。举个例子：我们可以使用Consul Template来动态配置Nginx的反向代理，Consul Template可以定期从Consul服务注册表中的存储数据中生成任意的配置文件，每当文件有变化时，会运行一个shell命令。同时，Consul Template可以生成一个配置反向代理的nginx.conf文件，然后运行一个指令，使Nginx去重新加载它的配置。此外，还有一个更复杂的实现，可以通过HTTP协议或DNS去动态地重新配置Nginx。比如，Kubernetes和Marathon这些部署环境会在集群中的每个主机上运行一个代理，这个代理承担了server-side discovery中的负载均衡器的角色。为了向一个服务发送一个请求，客户端需要使用host的IP地址和服务分配的端口，通过代理路由这个请求。这个proxy会直接将请求发送到集群上可用的服务实例中。

Server-side discovery模式和Client-side discovery模式一样，也是优势、缺陷并存。Server-side discovery模式最大的好处在于服务发现的细节不保存在客户端中，客户端只需要向负载均衡器发送请求，而不需要关心客户端使用哪一种语言和框架但是，这种模式也有一些其他问题，这个负载均衡器会是一个单点系统，未来有可能成为系统的稳定性隐患，需要监控和管理。

#### 服务注册列表

服务器注册列表是服务发现中的非常重要的部分。服务器注册列表是一个包含了服务实例的网络地址的小型数据库。它必须是高可用和最新的。客户端可以缓存从服务器注册列表处获得的IP地址。但是，这些地址信息最终因为过期而失效，客户端会找不到真实的地址。所以，服务器注册列表由多台机器或者一个服务器集群组成，并且会通过应用协议来保持最终一致性。

正如上面提到的，Netflix Eureka是一个服务注册表的好例子。它提供了一个REST API用来注册和查询服务实例。一个服务实例通过POST请求来注册自己的网络位置，每隔30秒要通过一个PUT请求重新注册。注册表中的一个条目会因为一个HTTP DELETE请求或实例注册超时而被删除，客户端通过一个HTTP GET请求来检索注册的服务实例。

同样在Netflix，通过在每个EC2的可用区中，运行多个Eureka服务器实现其高可用。每一个运行在亚马逊EC2实例上的Eureka服务器都有一个弹性的IP地址。域名系统文本记录（DNS TEXT RECORDS）用来存储Eureka集群配置，实际上是从可用区到Eureka服务网络地址列表的映射，当一个Eureka服务器启动时，会向域名系统发送请求，检索Eureka集群的配置、定位节点，并且给自己分配一个未占用的IP地址。Eureka客户端（服务和服务客户端）查询DNS去寻找Eureka服务器的网络地址。客户端想使用这个可用区内的Eureka服务器，如果没有可用的Eureka服务器，客户端会用另一个可用区内的Eureka服务器。

其它常见的服务注册例子包括：

* Etcd：一个高可用，分布式，一致的key-value存储，用来共享配置和服务发现。Kubernetes和Cloudfoundry都使用了etcd；
* Consul：一个发现和配置服务的工具。客户端可以利用它提供的API，注册和发现服务。Consul可以执行监控检测来实现服务的高可用；
* Apache Zookeeper：一个常用的，为分布式应用设计的高可用协调服务，最开始Zookeeper是Hadoop的子项目，现在已经顶级项目了。
* 一些系统，比如Kubernetes，Marathon和AWS没有一个明确的服务注册组件，这项功能是内置在基础设置中的。

### 服务注册

服务注册表是存储服务注册信息的。服务实例可以从服务注册表中注册和注销服务，注册就是添加数据，注销就是删除数据。服务的注册和注销主要有两种，一是通过服务实例自己操作服务注册表，就是self-registration模式。另一种则是通过第三方组件专门来维护服务实例，服务只和这个组件通信。这种方式就是third-party registration模式。

#### 自注册模式

在自注册（self-registration）模式中，服务器注册表中注册和注销服务地址是由服务实例自己添加或者删除。一般情况下，为了防止注册信息过期，每个注册过的服务实例会定时发送心跳请求。下图展示了这种自注册模式的架构：

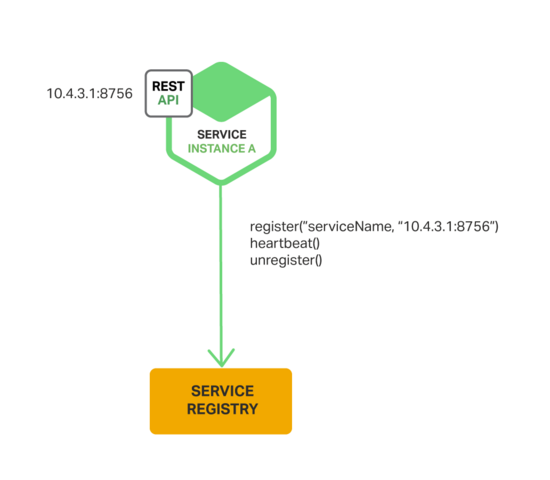


图 2‑5

来自于Netflix的OSS Eureka服务是通过这种方式注册服务的一个例子。服务实例注册和注销的所有问题都由Eureka服务端来处理。Spring Cloud提供了包括服务发现在内的很多模式，使Eureka的服务实例自动注册功能的实现大大简化。只要使用Spring的@EnableEurekaClient注解，就可以轻松注册Java的配置类，并且自动注册在注册表中。自注册模式最大的优点是实现起来比较简单，不需要其它组件。但是缺点在于，服务实例和服务注册表相对应，对于每种语言，都必须要实现一遍注册代码。

#### 第三方注册模式

在第三方注册（third-party registration）模式中，微服务实例是通过另一个子系统组件服务注册到服务表中，这个注册服务我们称之为注册器（service registrar）。注册器通过轮询或订阅事件去跟踪运行中服务实例的变化，一旦发现在环境中出现一个新可用的服务实例时，就会自动到把这个服务实例的地址信息等相关信息注册到服务注册表中。也可以通过轮训检测方式把断开的服务实例注销，下图展示了这种模式的架构。

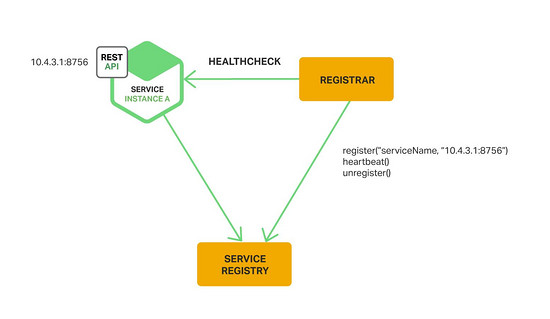


图 2‑6

第三方注册中注册器的一个例子是开源的Registrator项目。像Docker容器也能用Registrator自动注册和注销，同时Registrator也支持etcd和Consul等服务注册。Netflix的OSS Prana也是另外一个开源注册器的实现。，它主要用于非JVM语言编写的服务，OSS Prana是一个和服务实例配合的“双轮”应用。Prana会在Netflix的Eureka上注册和注销实例。注册器是一个部署环境的内置组件，由Autoscaling Group创建的EC2实例可以被ELB自动注册。而自动化容器操作的开源平台Kubernetes的服务也可以自动注册。

通过第三方注册（third-party registration）的方法主要的优点在于解耦，把服务实例和服务注册表两个业务分离。由于第三方注册有独立的服务，所以不需要为每个语言和框架都去实现服务注册逻辑。同样，如果是一个独立的服务，那么就会有单点的问题，未来有可能成为系统的稳定性隐患。这种情况可以通过集群部署的技术来提高稳定性。

### 服务降级

当大促、秒杀或者业务出现高峰时，为了保证核心服务的正常服务，我们往往会停掉一些不太核心的业务来保证核心服务，例如商品的评论、论坛的帖子的浏览功能或者查询粉丝积分相关的功能。还有一种场景就是某些服务因为某种原因不可用，但是还要希望有后续处理，做流程放通。以用户购买商品为例，如果用户登录鉴权服务不能正常工作，需要做业务放通，记录订单信息，允许用户继续购物，而不是简单的返回失败。

服务降级主要包括屏蔽降级和容错降级两种策略，通过者两种策略的服务降级功能，就可以满足上述两种场景的需求。

#### 屏蔽降级

这种方式是通过运维人员或者开发人员决策，运维人员或者开发人员需要在外界的触发条件达到某个临界值时，对某类或者某个服务进行强制降级。

它的处理流程如下所示：

1. 开发人员或者运维人员以管理员身份登录服务治理控制台，管理员具备服务治理的全套权限。
2. 开发人员或者运维人员选择服务降级菜单，在服务降级的界面中选择屏蔽降级这个功能。
3. 在服务查询结果界面选择需要降级的服务，并且指定具体的降级策略：返回null、返回指定异常还是执行本地Mock接口实现。
4. 服务治理界面可以通过服务注册中心客户端SDK，将屏蔽降级的指令和相关策略内容发送到服务注册中心。
5. 服务注册中心接收到上一步下发下来的屏蔽降级消息后，可以用消息的形式下分别发给服务提供者集群和服务消费者集群。
6. 服务消费者集群接收到屏蔽降级消息的通知之后，获取其中的相关内容，对本地缓存的服务订阅信息进行修改。当后续需要进行远程服务调用时，会检查是否有屏蔽降级的策略，如果匹配成功，则执行屏蔽降级逻辑，从而不发起远程服务调用。
7. 服务提供者集群接收到屏蔽降级消息的通知之后，获取其中的相关内容，对本地的服务发布缓存信息进行修改，把与之对应的服务降级属性更新为屏蔽降级。
8. 上述操作成功之后，服务注册中心将返回降级成功的应答消息，由服务治理门户界面展示。
9. 运维人员或者开发人员可以通过查询服务提供者列表来查看服务状态。
10. 服务注册中心返回服务状态为屏蔽降级状态。

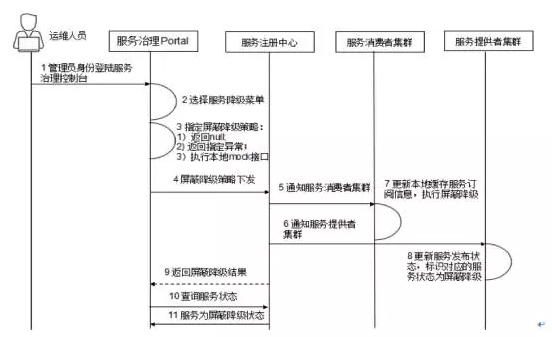


图 2‑7

#### 容错降级

这种策略是当非核心的服务不可用时，可以绕过这些故障业务，核心服务的继续运行，从而保证关键路径可以通过。



图 2‑8

### 服务的限流

限流可以认为服务降级的一种，限流就是限制系统的输入和输出流量已达到保护系统的目的。一般来说系统的吞吐量是可以被测算的，为了保证系统的稳定运行，一旦达到的需要限制的阈值，就需要限制流量并采取一些措施以完成限制流量的目的。比如：延迟处理，拒绝处理，或者部分拒绝处理等等。

常见的限流算法有：计数器、漏桶和令牌桶算法。

#### 计数器

计数器是最简单粗暴的算法。比如某个服务最多只能每秒钟处理100个请求。我们可以设置一个1秒钟的滑动窗口，窗口中有10个格子，每个格子100毫秒，每100毫秒移动一次，每次移动都需要记录当前服务请求的次数。内存中需要保存10次的次数。可以用数据结构LinkedList来实现。格子每次移动的时候判断一次，当前访问次数和LinkedList中最后一个相差是否超过100，如果超过就需要限流了。[引证：8]

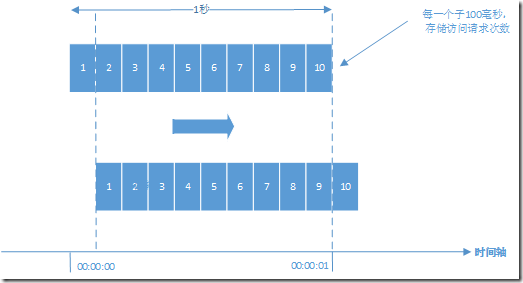


图 2‑9

很明显，当滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑，限流的统计就会越精确。[引证：8]

#### 漏桶算法

漏桶算法即leaky bucket是一种非常常用的限流算法，可以用来实现流量整形（Traffic Shaping）和流量控制（Traffic Policing）。贴了一张维基百科上示意图帮助大家理解：

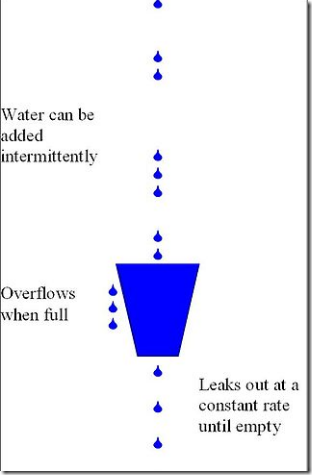


图 2‑10

漏桶算法的主要概念如下：

* 一个固定容量的漏桶，按照常量固定速率流出水滴；
* 如果桶是空的，则不需流出水滴；
* 可以以任意速率流入水滴到漏桶；
* 如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

漏桶算法比较好实现，在单机系统中可以使用队列来实现（.Net中TPL DataFlow可以较好的处理类似的问题，你可以在这里找到相关的介绍），在分布式环境中消息中间件或者Redis都是可选的方案。[引证：8]

#### 令牌桶算法

令牌桶算法是一个存放固定容量令牌（token）的桶，按照固定速率往桶里添加令牌。令牌桶算法基本可以用下面的几个概念来描述：

* 令牌将按照固定的速率被放入令牌桶中。比如每秒放10个。
* 桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃或拒绝。
* 当一个n个字节大小的数据包到达，将从桶中删除n个令牌，接着数据包被发送到网络上。
* 如果桶中的令牌不足n个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么缓冲区等待）。

如下图：

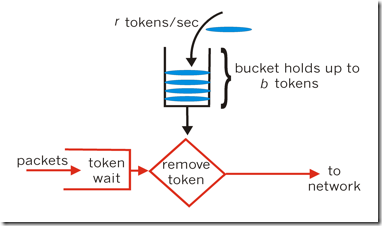


图 2‑11

令牌算法是根据放令牌的速率去控制输出的速率，也就是上图的to network的速率。to network我们可以理解为消息的处理程序，执行某段业务或者调用某个RPC。[引证：8]

#### 漏桶和令牌桶的比较

令牌桶可以在运行时控制和调整数据处理的速率，处理某时的突发流量。放令牌的频率增加可以提升整体数据处理的速度，而通过每次获取令牌的个数增加或者放慢令牌的发放速度和降低整体数据处理速度。而漏桶不行，因为它的流出速率是固定的，程序处理速度也是固定的。整体而言，令牌桶算法更优，但是实现更为复杂一些。[引证：8]

### 服务优先级调度

当系统的计算资源有限时，为了保证核心服务或者是高优先级的服务能够正常运行并保障服务的SLA，除了做一些熔断的功能之外，还可以降低一些非核心服务或者低优先级服务的调度频率，从而降低系统资源的占用，保障系统的整体运行平稳。

微服务治理框架可以在服务发布的时候指定服务的优先级，以Dubbo为例，如果用户没有指定，采用默认优先级策略。服务的优先级可以采用传统的低、中、高三级配置策略，同时可以为每个级别的优先级配置执行比例。在Dubbo中，服务发布通过扩展priority属性的方式指定优先级，服务提供者将优先级属性注册到服务注册中心，客户端消费者也会收到通知，客户端会根据不同的优先级在调用服务时采用不用的处理策略。好的服务治理框架也支持动态的修改服务的优先级。

### 集群容错

系统拆分为微服务后，单个微服务故障的概率变小了，故障影响的范围也在减少，但是微服务的节点数量大大增加。从整体上来看，系统某个微服务出现故障的概率会大大增加，如果不及时处理故障，故障扩散开来就会导致看起来系统中很多服务节点都故障了，因此需要微服务能够自动应对这种出错场景，及时进行处理。否则，如果节点一故障就需要人工处理，投入人力大，处理速度慢；而一旦处理速度慢，则故障就很快扩散，所以我们需要服务容错的能力。

服务调用的客户端根据配置的路由策略选择某个目标地址发起远程服务调用后，如果发生了远程服务调用异常，则需要客户端服务框架进行容错处理，往往是在一定重试次数之后，重新选择路由。因为配置的服务端服务地址一般不会只一个，如果一个访问不通会换其他的尝试。集群容错的功能是服务治理框架自己完成，对上层透明。

常用的集群容错策略如下：

* Failover策略:服务调用失败自动切换策略指的是当发生RPC调用异常时，重新选路，查找下一个可用的服务提供者。通常可以配置失败切换的最大次数和间隔周期，以防止E2E服务调用时延过大。
* Failback策略：在很多业务场景中，消费者需要能够获取到服务调用失败的具体信息，通过对失败错误码等异常信息的判断，决定后续的执行策略，例如非幂等性的服务调用。
* Failcache策略:Failcache策略是失败自动恢复的一种，在实际项目中它的应用场景如下：服务有状态路由，必须定点发送到指定的服务提供者。当发生链路中断、流控等服务暂时不可用时，服务框架将消息临时缓存起来，等待周期T，重新发送，直到服务提供者能够正常处理该消息。对时延要求不敏感的服务。系统服务调用失败，通常是链路暂时不可用、服务流控、GC挂住服务提供者进程等，这种失败不是永久性的失败，它的恢复是可预期的。如果消费者对服务调用时延不敏感，可以考虑采用自动恢复模式，即先缓存，再等待，最后重试。通知类服务：例如通知粉丝积分增长、记录接口日志等，对服务调用的实时性要求不高，可以容忍自动恢复带来的时延增加。
* Failfast策略：在业务高峰期，对于一些非核心的服务，希望只调用一次，失败也不再重试，为重要的核心服务节约宝贵的运行资源。此时，快速失败是个不错的选择。

### 服务上线审批下线通知机制

当微服务系统不断的变大，团队也到了一定的规模，团队的组织架构也会按照业务线进行炒粉。会划分成平台组、各种业务产品组等不同研发团队，一些团队甚至跨多地协同开发和运维。服务的上线和下线必须要有严格的流程来把控，一旦不合格的服务上线并被消费者消息，再想下线就非常困难了。

除了服务的上线，服务的下线也同样重要。虽然有些服务调用频次不高，业务量也不大。但是如果贸然下线，很有可能导致依赖它的消费者业务调用失败，给服务提供商造成损失。这个时候就需要知道一个服务所有的服务依赖，而这个问题常常是在研发和测试阶段不能知道的。在微服务门户上可以找到这些服务的所有客户端消费者，这样在服务下线的时候，就可以有充分的把握。

## 微服务的自动化测试

微服务将原本大一统的系统拆分为多个独立运行的“微”服务，微服务之间的接口数量大大增加，并且微服务提倡快速交付，版本周期短，版本更新频繁。如果每次更新都靠人工回归整个系统，则工作量大，效率低下，达不到“快速交付”的目的，因此必须通过自动化测试系统来完成绝大部分测试回归的工作。

自动化测试涵盖的范围包括代码级的单元测试、单个系统级的集成测试、系统间的接口测试，理想情况是每类测试都自动化。如果因为团队规模和人力的原因无法全面覆盖，至少要做到接口测试自动化。

## 微服务的跟踪

随着业务的发展以及系统规模的越来越大，微服务之间的调用关系也变得越来越错综复杂。往往一个客户端发起的请求在后端系统中会经过多个不同的微服务，从而形成一条复杂的分布式服务调用链路。在每条链路中任何一个节点出现延迟或者异常的时候，都有可能导致客户端访问最后的失败。这时候，如果可以对每一个请求，特别是记录每一个访问节点的状态，时长有跟踪计划的话，将可以帮助我们快速发现的定位问题以及监控分析每条请求链路上的性能瓶颈等。

### 跟踪原理

微服务架构中的服务跟踪在理论上比较简单，它主要通过下面两个关键点老实现：

* 为了实现对客户端请求的跟踪，当客户端请求到达某一个服务入口时，服务需要判断是否已经存在跟踪的唯一标识：TraceID。如果已经有了，则不再创建，说明这个请求不是客户端请求的第一个服务。如果没有则需要服务跟踪框架为该客户端请求创建TraceID。同时，这个请求在微服务系统内部流转的时候，服务跟踪框架之中可以使用Http的头信息来保持传递该唯一标识，只要返回给请求方为止。通过TraceID的记录，我们就能将所有请求链关联起来。
* TraceID是标识整个一次客户端请求的。如果要统计各个处理单元的时间延迟，或者说走了哪些处理单元的话，就要使用SpanID了。每一个Span，代表一次处理单元。当达到每一个处理单元的时候，跟踪框架可以在本地TraceID的基础上，添加一个Span开始。当处理单元结束处理的时候，可以添加Span结束。通过记录开始Span和结束Span的时间戳，就能统计出该Span的时间延迟。

### 抽样收集

当微服务越来越多，成百上千的时候，如果每一次调用的日志都给记录并存储下来会占用大量的存储空间，而其中大部分数据都是没有使用价值的。在统计服务接口性能的业务场景中，就需要少量的样本数据就能满足业务需求。一些开源的微服务框架，比如Spring Cloud Sleuth就提供了这样的功能。它可以用通过配置一些抽样的策略来确定日志是否需要给存储下来。默认情况下，它可以通过设置百分比（默认是10%）来执行抽样过程。另外也可以自定义抽样算法。

## 微服务和DevOps

相比大一统的系统，微服务需要部署的节点增加了几倍甚至十几倍，微服务部署的频率也会大幅提升，综合计算下来，微服务部署的次数是单体应用系统部署次数的几十倍。这么大量的部署操作，如果继续采用人工手工处理，需要投入大量的人力，且容易出错，因此需要自动化部署的系统来完成部署操作。自动化部署系统包括版本管理、资源管理（例如，机器管理、虚拟机管理）、部署操作、回退操作等功能。

Devops，即开发测试和部署运维的一体化，是业内说到互联网或者微服务架构常常会说到的话题。单体应用发展到一定阶段，会拆分为多个小应用后，虽然这样做的好处是可以松耦合和可扩展，但是如果拆分的服务越来越多的话，服务之间本身的集成和部署运维的复杂度就越发独显。任何一个服务，当它依赖的外部其它应用服务越多的时候，整个集成、部署和联调测试的过程就越复杂。如果还是和原来维护单体应用一样使用人工维护的话，由于操作的服务较多，工作量和准确度都不能保证。

服务化开发谈的最多的是单个服务的持续集成，包括配置环境集成，自动打包部署，自动化的冒烟测试等。微服务架构并不是要另辟蹊径，仍然需要做好单个服务本身的持续集成，同时在这个基础上更进一步，推广到多个服务的打包部署和服务之间间的集成。Devops的思路就是其中的核心思想，它希望能够打破传统研发和运维的界限，把运维融入到开发中，从而实现开发设计到部署运维的一体化。

在微服务架构中，单个服务本身需要在独立的进程里面运行，各个服务之间在部署的时候都是一个单独的进程，服务之间是进程级别的隔离。我们的微服务一般都会有几十个、上百个甚至更多，为了保证隔离性，不可能部署在几十台物理机上。我们可以用虚拟机，但是当几十个进程都完全用独立的虚拟机也是不现实的，而这个问题的解决刚好就是利用轻量级的Docker容器来做这个事情，每个Docker是独立的容器刚好又完全做到进程级别的隔离，资源占用率又最小，这些特点刚好满足微服务架构的开发测试和自动化部署。

部署的问题解决之后就需要考虑一个统一的服务管控和治理平台，按照当前微服务架构的整体思路，虽然单个服务的实现、测试和发布仍然是在服务内部完成，但是这些服务暴露的服务本身的调用策略，服务本身的安全，注册和发现，降级和限流等仍然需要一个统一的SOA服务管理平台来完成。成熟的方案有Spring Could和阿里的Dubbo框架。

## 配置中心

微服务的节点数量非常多，通过人工登录每台机器手工修改，效率低，容易出错。特别是在部署或者排障时，需要快速增删改查配置，人工操作的方式显然是不行的。除此之外，有的运行期配置需要动态修改并且所有节点即时生效，微服务需要一个统一的配置中心来管理所有微服务节点的配置。

配置中心包括配置版本管理（例如，同样的微服务，有10个节点是给移动用户服务的，有20个节点是给联通用户服务的，配置项都一样，配置值不一样）、增删改查配置、节点管理、配置同步、配置推送等功能。

## 服务监控

系统拆分为微服务后，节点的数量大大增加，导致需要监控的机器、网络、进程、接口调用数等监控对象的数量大大增加；同时，一旦发生故障，我们需要快速根据各类型信息来定位故障。这两个目标如果靠人力去完成是不现实的。举个例子：我们收到用户投诉说业务有问题，如果此时采取人工的方式去搜集、分析信息，可能把几十个节点的日志打开一遍就需要十几分钟了，因此需要服务监控系统来完成微服务监控系统来完成微服务节点的监控。

服务监控的主要作用有：

* 实时搜集信息并进行分析，避免故障后再来分析，减少了处理时间。
* 服务监控可以在实时分析的基础上进行预警，在问题萌芽的阶段发觉并预警，降低了问题的影响范围和时间

## 本章小结

随着单体项目的越来越大，需求越来越多，从单体架构过度到微服务架构变得势在必行。但是微服务架构并不是单体架构的替代品，而是在软件系统发展到不同阶段所需要采用的不同的系统架构。和单体架构相比，微服务架构同样有缺点和限制。由于微服务架构的复杂性，服务治理变得非常重要，它直接影响了微服务实施的成败。服务治理是微服务架构在实施过程中遇到的问题的一组解决方案，有解决可用性和可靠性的，也有解决流程方面问题的。同时，微服务框架由于其部署和运行时的复杂性，线上的问题常常很难排查，这就需要和运维配合起来，更多的利用自动化工具或者开源项目来开展运维工作。

# 第三章 车联网项目需求分析

从本章开始，我们将以一个实战项目为例，使用微服务架构的思想构建一个车联网应用平台，这个平台命名为V+平台。这个平台并不是从无到有，而是基于一个已有的单体应用进行的重构。从而可以更好的理解微服务架构和单体应用架构的优劣。



## 现存的问题

车联网系统已经运营了多年，其问题也逐渐暴露出来：

* 不够灵活：对应用程序做任何细微的修改都需要将整个应用程序重新构建、重新部署。开发人员需要等到整个应用程序部署完成后才能看到变化。如果多个开发人员共同开发一个应用程序，那么还要等待其他开发人员完成了各自的开发。这降低了团队的灵活性和功能交付频率；
* 妨碍持续交付：单体应用可能会比较大，构建和部署时间也相应地比较长，不利于频繁部署，阻碍持续交付。在移动应用开发中，这个问题会显得尤为严重；
* 受技术栈限制：对于这类应用，技术是在开发之前经过慎重评估后选定的，每个团队成员都必须使用相同的开发语言、持久化存储及消息系统，而且要使用类似的工具，无法根据具体的场景做出其它选择；
* 技术债务：“不坏不修（Not broken，don’t fix）”，这在软件开发中非常常见，单体应用尤其如此。系统设计或写好的代码难以修改，因为应用程序的其它部分可能会以意料之外的方式使用它。随着时间推移、人员更迭，这必然会增加应用程序的技术债务。

## 设计目标

系统的设计目标除了要解决上述遇到的问题之外，还需要对未来平台2-3年的发展进行规划。

### 关键需求属性

* 不存在单点故障，系统支持横向扩展。
* 支持大用户并发访问；
* 周期性报告在指定周期完成前生成；
* 支持海量数据存储，提供大数据分析基础；
* 会导致系统不稳定的故障，要在10分钟内通知到运维人员；
* 支撑百万级物联网设计的接入

### 关键质量属性

关键质量属性主要包含高性能、可靠性和可扩展性。这一节主要介绍了V+系统在这三个方面的要求和理解，具体的实现方案将会在质量一节有详细的描述。

* 高性能

影响最终产品的性能因素非常多，其中软件因素如下：

1. 架构不合理导致的性能问题。
2. 编码实现不合理导致的性能问题，例如锁的不恰当使用导致性能瓶颈。

硬件原因如下：

1. 服务器硬件配置太低导致的性能问题。
2. 带宽、磁盘的IOPS等限制导致的IO操作性能差。
3. 测试环境被共用导致被测试的软件产品受到影响。

“性能是设计出来的，而不是测试出来的”。尽管影响产品性能的因素非常多，但是架构的性能模型合理与否对性能的影响非常大。如果过一个产品的架构设计的不好，无论开发如何努力，都很难开发出一个高性能、高可用的软件产品。

* 可靠性

用软件系统规模越做越大越复杂，其可靠性越来越难保证。应用本身对系统运行的可靠性要求也越来越高。V+系统作为物联网系统虽然没有类似航天、金融等系统对可开行要求那么高，但是如果涉及到告警、故障、能耗等相关业务，对系统可靠性的高要求也是车联网系统的核心竞争力之一。

* 可扩展性

随着系统的规模越来越大，系统的维护性就可能会变得越来越差，有时可能会出现牵一发而动全身的情况。如果需要修改某个功能的代码，或者添加某项功能，会耗费大量的人力和时间。同时，一个不起眼的小小的业务修改有时候也会对性能造成重大的影响。这种情况下，高可扩展性的、低耦合的应用程序就变得非常重要了。下文主要通过业务的扩展和吞吐量的扩展来描述V+系统的可扩展性。

# 第四章 车联网架构项目实战应用



## 设计原则

### 弹性架构

应用系统追求灵活性，以便随时应对真实世界的业务变化。系统压力是一个重要的变化维度，是否能够随着系统压力的增减而动态的调整系统的某一部分体现了架构是否具有足够的弹性。

系统投入使用后，将面临业务流量不断增长的压力。系统处理能力也需要具备同步伸缩的能力，即通过增强系统单机性能或者增加系统的节点资源来满足不断增长的业务流量的需要。应满足系统资源的横向扩展和纵向扩展的需要，能在逐渐增加系统资源的情况下不断增长相应的处理能力。为了保证系统在将来能够得到充分的利用，投资不被浪费，系统要具有充分、灵活的规模伸缩能力和业务扩展能力。以便于系统将来的扩容与升级，并充分考虑接口的标准化。

开发一个能够水平扩展的系统可能是个挑战，尤其是随着参与节点的数量增至三倍或四倍，更是困难重重。所有设计决策务必要牢记水平扩展这一条准则。即，所有的子系统都能够水平扩展。

### 避免单点系统

单点系统最有可能造成系统整体故障和性能瓶颈。对于对可用性要求很高的大型应用系统来说，单点问题是不可接受的。常规的解决单点问题的方式主要有主备切换和负载均衡的方式。

### 可模块化测试

子系统的可测试性是衡量架构优劣的重要指标之一。也是未来架构或者子系统重构的坚实基础。主要需要做到两点：

1. 合理的划分子系统，通过消息中间件或者RPC调用进行系统隔离。子系统内部功能高内聚，子系统间低耦合。
2. 系统的边界定义清晰，数据交换需要有严格的定义文档。

### 标准化原则

系统的总体设计充分参照国际上的规范、标准，支持国内外目前所流行的主流网络体系结构和网络运行系统，采用国际上成熟的模式。 同时会根据行业具体需求探索并制定符合行业发展需求的相关标准及规范。

### 可靠性原则

对于客户来说，必须满足高可用要求，系统要求7\*24小时不间断地运行，因为系统服务停止即意味着客户的服务质量下降、服务声誉下降和经济损失。所以系统在设计上保证了系统的高可用性，防止单点故障。

### 高性能、大容量原则

本系统需要应对数以万计的车载终端并发接入，在设计过程中必须充分考虑海量的并发容量需求，同时为了减少服务器的投入，降低成本，需要提高单节点服务器的运行性能，实现单节点支撑并发车载终端接入数量的最大化。具体要求系统在处理每次业务的各个环节上，都能够充分利用系统资源，从而加快处理速度，降低资源开销，最大化的提升系统的吞吐量。

## 系统设计

### 系统架构图

系统整体架构如下图，由于是车联网整体架构，所以还包含设备的接入、数据实时和异步处理，以及大数据等相关内容。而和微服务架构相关的从下到上是：IOT服务治理层、IOT基础服务层、IOT业务服务层。

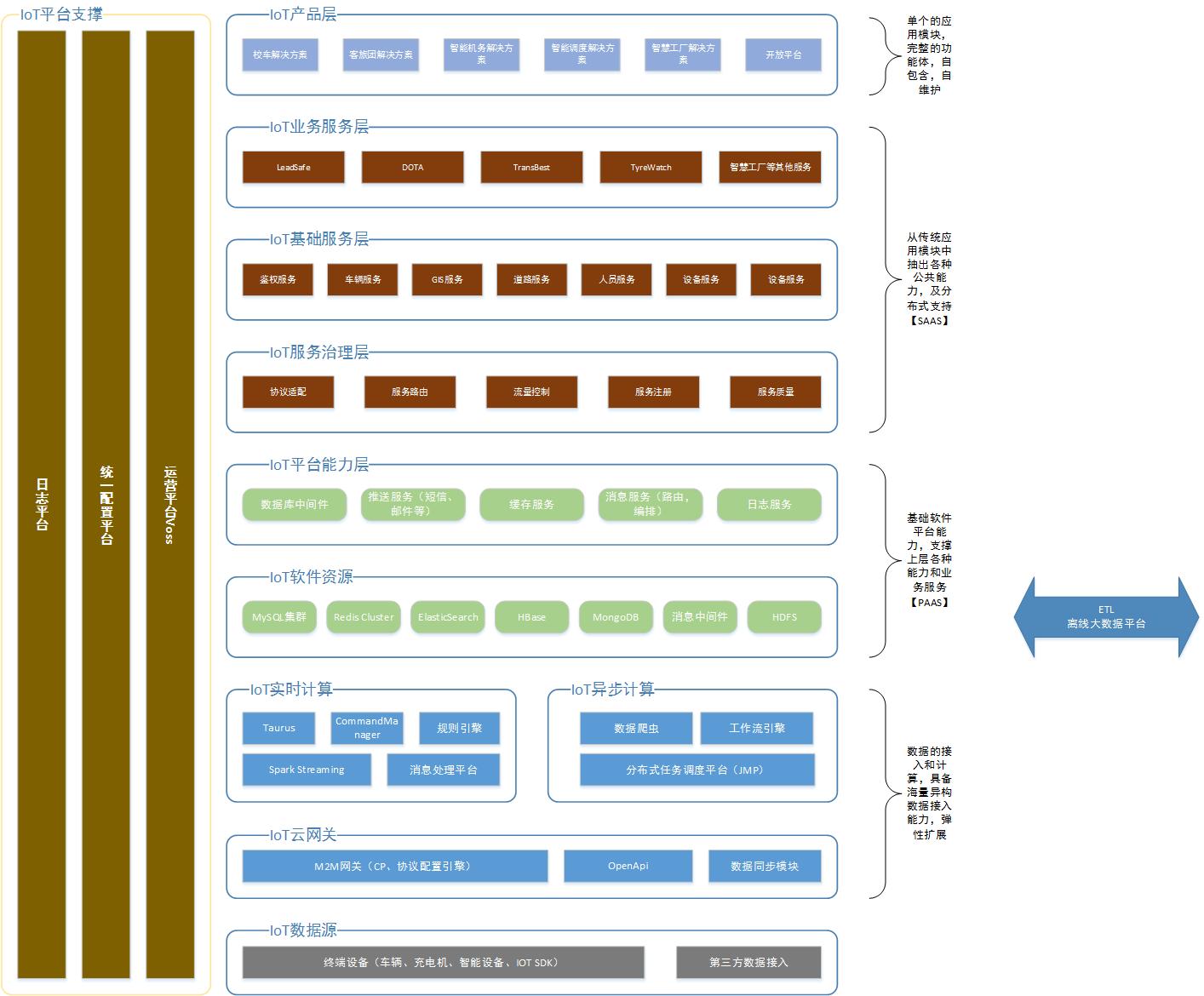


Figure 0‑1

### 逻辑分层

V+架构按照逻辑分层主要分为：接入和计算层，平台PaaS层，平台SaaS层，专业产品层，平台支撑层。每个层内部还分不同的系统。

#### 接入和计算层

接入和计算层主要负责数据的接入和计算，具备海量异构数据的接入能力和弹性扩展能力。

##### IOT云网关

IOT云网关主要负责设备的接入。不同的模块适配不同的接入方式。未来如果涉及到MQTT的协议接入，还会适配MQTT的服务器。

经过IOT云网关处理之后数据处理会被推送到消息中间件kafka中。平台的实时计算层会接收到数据进行实时处理。

##### M2M云网关

云网关主要负责常规设备数据的接入。包括设备鉴权，设备状态维护，协议解析、下发等功能。IOT云网关将整合MQTT服务器，接入MQTT协议数据。

##### 实时计算

实时计算层是对IOT数据的实时计算和实时存储，对实时性要求比较高的业务可以在这一层进行计算。在v1.0的架构中，这部分功能是使用VL和DTF来实现。在v2.0架构中将使用Pisces和Aries来实现。Pisces和Aries的架构设计会在单独的文档中描述。实时计算层的数据来源是kafka。具体的数据流图会在数据架构章节中描述。下面主要介绍实时计算层中最重要的三个：Pisces、Aries和CommandCenter。

##### 异步计算

异步计算有别于实时计算，类似于离线计算，用于计算非实时性的业务。比如，判断车辆的上下线，终端命令超时计算、定时任务、一些日报表统计等等。

#### 平台PaaS层

平台PaaS层是平台的基础能力，用于支撑上层各种能力和业务服务。平台PaaS主要包含IOT平台能力层和IOT软件资源层。

##### 软件资源层

IOT软件资源层是平台会用到的最基础的软件集合。包含关系型数据库、缓存组件、消息中间件、分布式文件存储等。所有的基础软件资源都有横向扩展的能力，为上层业务提供最原始的能力支撑。

##### 平台能力层

IOT平台能力层是对资源层的一个封装。它屏蔽了操作某些软件资源的复杂度。其中主要包含缓存服务、日志服务、消息服务等等。

缓存服务，上层业务可以使用提供了Api的缓存服务来操作RedisCluster，而使用者不用关心RedisCluster部署在哪里，或者缓存的Key存储在哪台服务器上。

IOT平台能力层提供了一组和业务无关的、提供基础资源访问的抽象接口，为上层业务提供PaaS层面上的服务。

#### 平台SaaS层

平台SaaS是从传统应用模块中抽出各种通用服务，依托于服务治理框架，实现分布式支持。目前是用Dubbo框架来实现整个微服务框架。

##### 服务治理层

服务治理层主要的功能包括：服务注册与发现、服务监控、服务调用控制、服务编排等功能。

##### 基础服务层

基础服务层提供和业务相关的、通用的业务接口。比如车辆基础信息服务、人员基础信息服务、位置服务、鉴权服务等等。这些服务通常会被很多专业产品使用。

##### 业务服务层

业务服务层是一组和专业产品强相关的服务接口。这些接口会把一些功能按照专业产品的划分组合在一起，提供服务的接口。比如，获取车辆能耗信息的接口，它会根据车辆不同的能源类型返回不同的数据。

业务服务层会按照leadsafe、dota、transbest等这样的维护来分别部署，已达到专业产品独立部署、独立发版的要求。

#### 专业产品层

专业产品层是一组解决方案的组合。每个解决方案就是一个子系统（Web网站），可以独立开发，独立部署。每个解决方案是由多个业务服务层的服务组合而来。每一个解决方案和业务服务是一对多的关系。

#### 平台支撑层

支撑平台运营的工具类子系统都属于这一层，主要包含运营平台，日志平台和统一配置平台。

### 微服务技术选型

#### 选型准则

对于技术选型，我个人有很多标准，其中下面三项是最重要的：

* 生产级

我们选择的技术栈是要解决实际业务问题和上生产抗流量的（选择不慎可能造成生产级事故），而不是简单做个 POC 或者 Demo 展示，所以生产级（Production Ready），可运维（Ops Ready），可治理，成熟稳定的技术才是我们的首选；

* 一线互联网公司落地产品

我们会尽量采用在一线互联网公司落地并且开源的，且在社区内形成良好口碑的产品，它们已经在这些公司经过流量冲击，坑已经基本被填平，且被社区接受形成一个良好的社区生态（本文附录部分会给出所有推荐使用或参考的开源项目的 GitHub 链接）。

* 开源社区活跃度

GitHub 上的 stars 的数量是一个重要指标，同时会参考其代码和文档更新频率（尤其是近年），这些指标直接反应开源产品的社区活跃度或者说生命力。

另外，对于不同业务体量和团队规模的公司，技术选型标准往往是不同的，创业公司的技术选型和 BAT 级别公司的技术选型标准可能完全不同。本文主要针对日流量千万以上，研发团队规模不少于 50 人的公司，如果小于这个规模我建议认真评估是否真的需要采用微服务架构。考虑到 Java 语言在国内的流行度和我个人的背景经验，本文主要针对采用 Java 技术栈的企业。本文也假定自建微服务基础架构，有些产品其实有对应的云服务可以直接使用，自建和采用云服务各有利弊，架构师需要根据场景上下文综合权衡。

#### 服务框架选型

服务框架是一个比较成熟的领域，有太多可选项。Spring Boot/Cloud[附录 12.1] 由于 Spring 社区的影响力和 Netflix 的背书，目前可以认为是构建 Java 微服务的一个社区标准，Spring Boot 目前在 GitHub 上有超过 20k 星。

基于 Spring 的框架本质上可以认为是一种 RESTful 框架（不是 RPC 框架），序列化协议主要采用基于文本的 JSON，通讯协议一般基于 HTTP。RESTful 框架天然支持跨语言，任何语言只要有 HTTP 客户端都可以接入调用，但是客户端一般需要自己解析 payload。目前 Spring 框架也支持 Swagger 契约编程模型，能够基于契约生成各种语言的强类型客户端，极大方便不同语言栈的应用接入，但是因为 RESTful 框架和 Swagger 规范的弱契约特性，生成的各种语言客户端的互操作性还是有不少坑的。

Dubbo是阿里多年构建生产级分布式微服务的技术结晶，服务治理能力非常丰富，在国内技术社区具有很大影响力，目前 github 上有超过 16k 星。Dubbo 本质上是一套基于 Java 的 RPC 框架，当当 Dubbox 扩展了 Dubbo 支持 RESTful 接口暴露能力。

Dubbo 主要面向 Java 技术栈，跨语言支持不足是它的一个弱项，另外因为治理能力太丰富，以至于这个框架比较重，完全用好这个框架的门槛比较高，但是如果你的企业基本上投资在 Java 技术栈上，选 Dubbo 可以让你在服务框架一块站在较高的起点上，不管是性能还是企业级的服务治理能力，Dubbo 都做的很出色。新浪微博开源的 Motan（GitHub 4k stars）也不错，功能和 Dubbo 类似，可以认为是一个轻量裁剪版的 Dubbo。

## 平台实现

### 业务拆分逻辑

应用微服务架构需要对业务进行拆分，在V+平台中，我们将对业务模块做垂直拆分，每一个模块为一个专业产品，分成：基础信息模块，安全模块，能耗模块，调度模块等等。每一个专业产品有独立的前端、后端服务、数据表设计等等，可以独立部署。

### 前端的实现

整体Web架构采用前后端分离的方式。前端采用AngularJs来进行模块化的开发。通过Nginx来向后端的Dubbo服务发送http请求。

前端也和服务框架一样，使用专业产品维度进行拆分。这样的好处是，各个专业产品组可以独立开发、部署和运维。

#### 前端部署图示意图

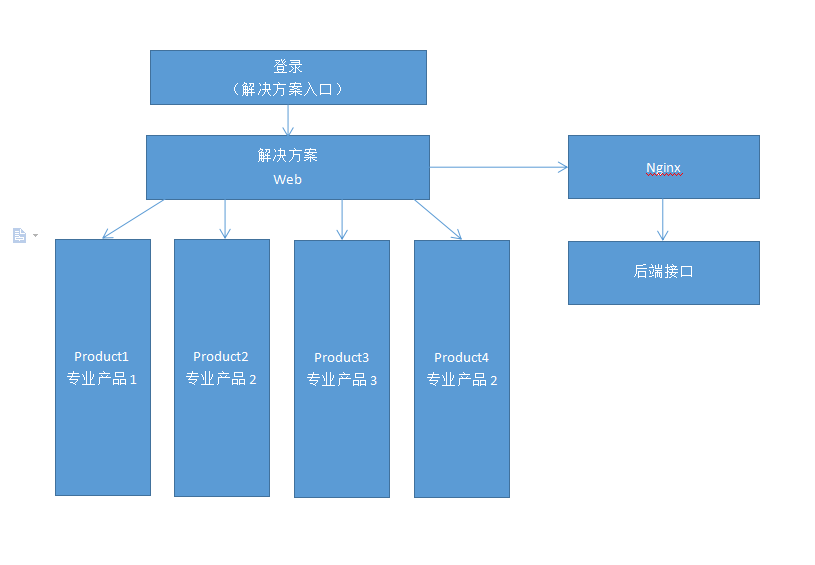


Figure 0‑2

#### 专业产品目录结构

前端代码的目录也应该按照微服务架构的拆分方式构建，在web目录下新建products目录，用来放各置专业产品。专业产品目录规范以标准的产品名命名。结构图如下：

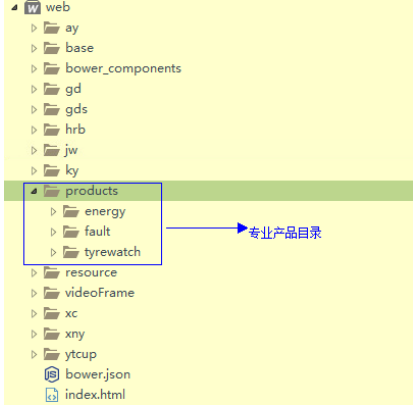


Figure 0‑3

每个专业产品目录下都有pages、resource和route文件。

1. 故障监控专业产品

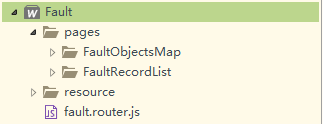


Figure 0‑4

1. 能耗专业产品

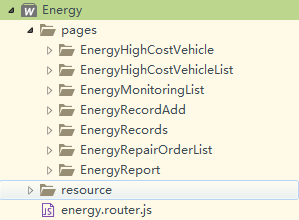


Figure 0‑5

#### 页面命名规范

* 针对通用页面：{产品前缀}{页面名}
* 针对项目页面：{产品前缀}{页面名}-{项目简写}

如图：

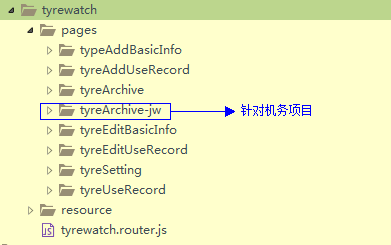


Figure 0‑6

#### 代码命名规范

全局变量的命名加上专业产品名前缀，以免不同项目下同名变量冲突。例如：tyrewatch专业产品中全局变量 Foo，加上前缀tyre，$rootScope. tyreFoo = {};

#### 产品路由(router)

每个专业产品拥有自己的router.js配置。命名规范：{产品名}.router.js，如下图，以轮胎管理为例：

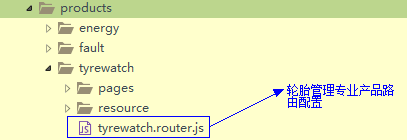


Figure 0‑7

路由内容示例：以对象数组形式定义。router里面的配置节点不能与其他router里的配置节点重名。



Figure 0‑8

#### 产品级数据接口配置(dataConfig.json)

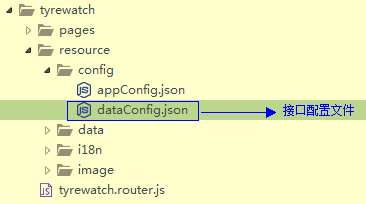


Figure 0‑9

如上图，product字段，表示此配置项数据产品级的。对应的值为{产品名}，此字段为必须项。否则无法被主框架识别是否为产品级的配置。

dataUrl字段，产品级数据接口的URL。对应产品自己的后端接口。此字段如果没有，默认会取项目级的dataUrl。



Figure 0‑10

#### 专业产品页面资源引用proload.js

页面的代码和模板都通过proload.js加载到主框架中。

products对象

变量list:[{产品配置1},{产品配置2},….] 对象数组。项目中用到哪些专业产品都在此配置。

数组中的产品配置节点(JSON对象)：

{

pid:产品名,

url:引用的url,同目录下使用相对地址。产品独立部署时使用绝对地址。

}

proload.js内容如图：



Figure 0‑11

proload.js文件引用，注意引用顺序。放置在angular的主文件之后，并且在app.js之前。如图：

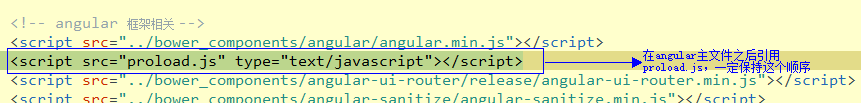


Figure 0‑12

proload.js文件引用，注意引用顺序。放置在config.js的配置文件之后，并且在app.js之前。如图：

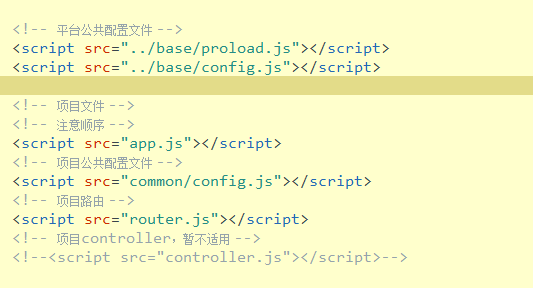


Figure 0‑13

#### 产品级数据接口配置获取

每个产品拥有独立的数据接口配置文件。在app.js中加入产品级的数据接口配置文件引用。每个产品的配置都会在此获取。如下图：

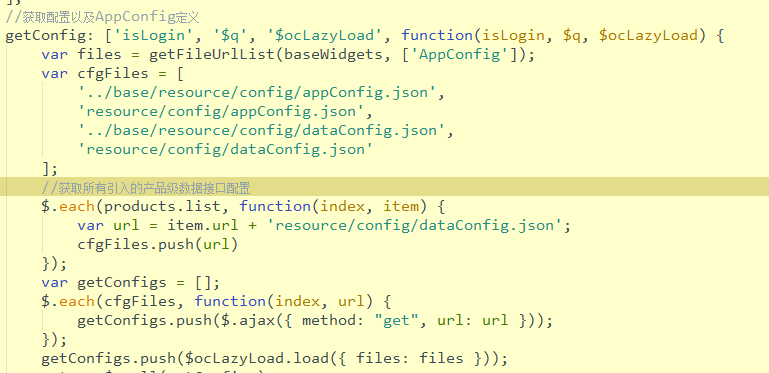


Figure 0‑14

app.js中产品级配置与主配置合并，如下图：

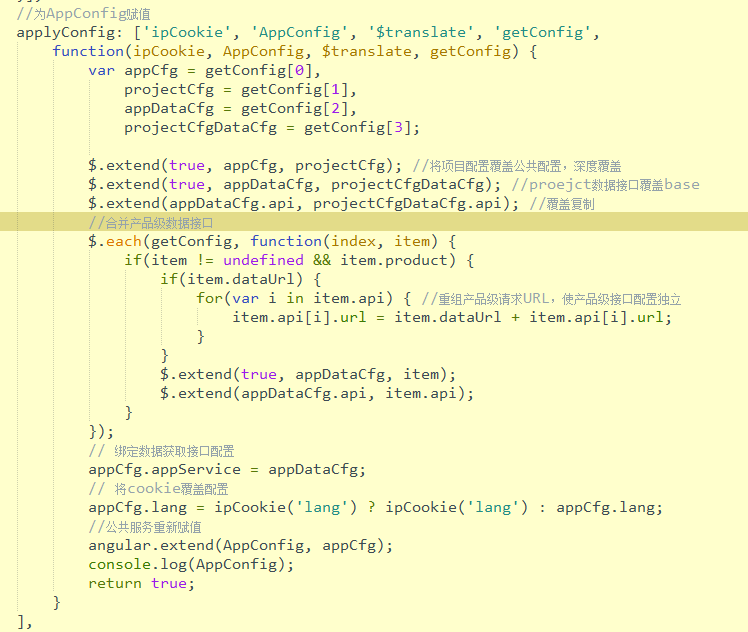


Figure 0‑15

### 后端服务的实现

经过技术选型分析，后端服务采用Spring boot、Spring cloud Zuul、 zookeeper、dubobo、mybatis、druid的技术栈实现。

#### 相关技术介绍

###### Spring boot

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

* 依赖Jar如下：

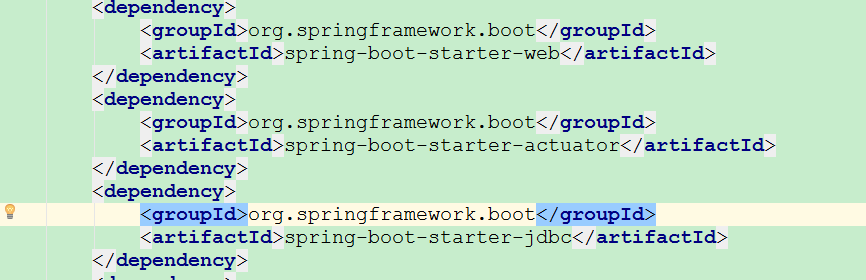


Figure 0‑16

* Spring boot常用注解
* @RestController 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者，它继承自@Controller注解。4.0之前的版本，[spring](http://lib.csdn.net/base/javaee) MVC的组件都使用@Controller来标识当前类是一个控制器servlet。
* @RequestMapping 注解提供路由信息。它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到 home 方法。
* @EnableAutoConfiguration类级别的注解，这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于 spring-boot-starter-web 添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。Starter POMs和Auto-Configuration：设计auto-configuration的目的是更好的使用"Starter POMs"，但这两个概念没有直接的联系。你可以自由地挑选starter POMs以外的jar依赖，并且Spring Boot将仍旧尽最大努力去自动配置你的应用 。
* @Configuration等价 与XML中配置beans；用@Bean标注方法等价于XML中配置bean
* @SpringBootApplication注解等价于以默认属性使用 @Configuration,@EnableAutoConfiguratio和@ComponentScan 。
* @ConfigurationProperties类添加JSR-303 javax.validation约束注解。
* @Profiles Spring Profiles提供了一种隔离应用程序配置的方式，并让这些配置只能在特定的环境下生效。任何@Component或@Configuration都能被@Profile标记，从而限制加载它的时机。
* @ResponseBody表示该方法的返回结果直接写入HTTP response body中一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。比如异步获取json数据，加上@responsebody后，会直接返回json数据。
* @Component：泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。一般公共的方法我会用上这个注解。
* @AutoWired byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上（required=false）时，就算找不到bean也不报错。
* @RequestParam用在方法的参数前面
* @PathVariable路径变量。
* @ControllerAdvice包含@Component。可以被扫描到,统一处理异常。
* @ExceptionHandler（Exception.class）用在方法上面表示遇到这个异常就执行以下方法。
* 通过@value注解来读取application.properties里面的配置，一般常用配置文件都在application.properties。

##### Dubbo

阿里巴巴开源RPC服务框架Dubbo,引入Dubbo目前前期是为了解决专业产品夸服务调用，后期主要是为了实现粒度更细的服务化做铺垫，Dubbo详细使用见链接：http://dubbo.io/。

* 依赖Jar如下：

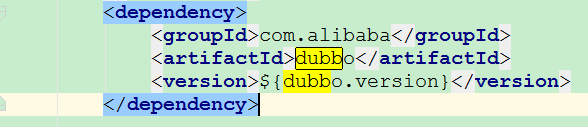


Figure 0‑17

* Dubbo总体架构

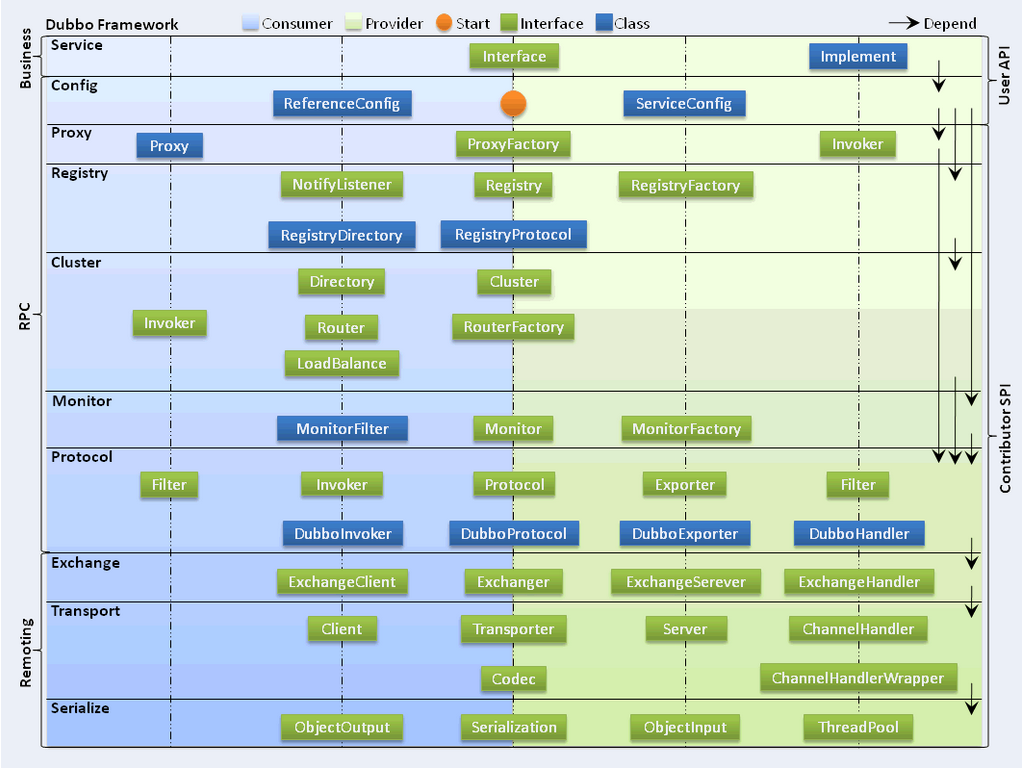


Figure 0‑18

* 服务注册

对于服务提供方，它需要发布服务，而且由于应用系统的复杂性，服务的数量、类型也不断膨胀；对于服务消费方，它最关心如何获取到它所需要的服务，而面对复杂的应用系统，需要管理大量的服务调用。而且，对于服务提供方和服务消费方来说，他们还有可能兼具这两种角色，即既需要提供服务，有需要消费服务。通过将服务统一管理起来，可以有效地优化内部应用对服务发布/使用的流程和管理。服务注册中心可以通过特定协议来完成服务对外的统一。Dubbo提供的注册中心有如下几种类型可供选择：

* Multicast注册中心
* Zookeeper注册中心
* Redis注册中心
* Simple注册中心
* 目前我们采用的Zookeeper注册
* 监控平台Dubbo Admin

去github官网：<https://github.com/alibaba/dubbo>构建了一个dubbo Admin注册到Zookeeper即可；

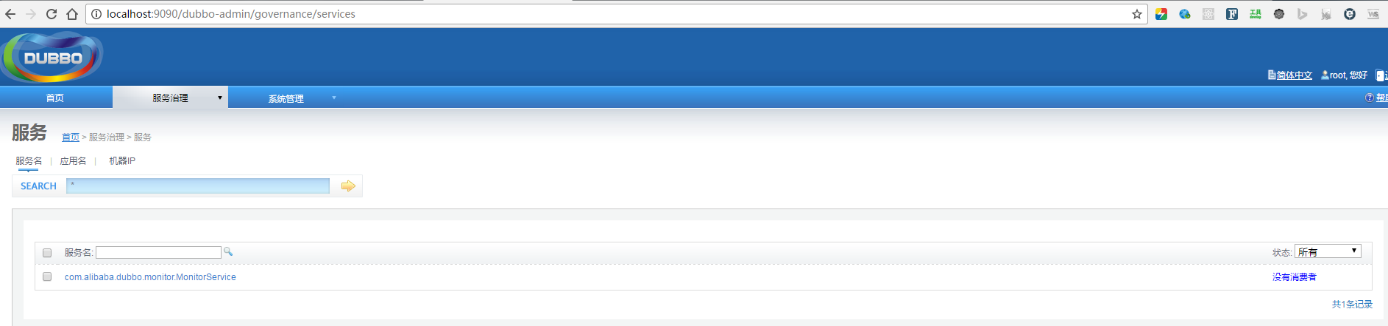


Figure 0‑19

##### Lombok

由于Lombok通过注解申明,编译期动态生成相关实体的GET和SET结构体方法，所以相应的IDE需要配置插件

* 依赖Jar如下：

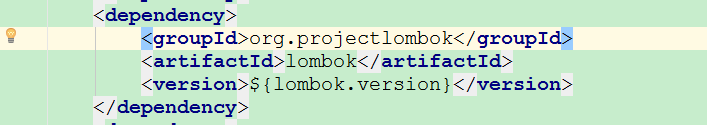


Figure 0‑20

* Idea配置

1. 找到file里面的settings

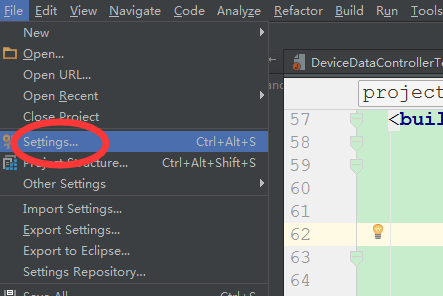


Figure 0‑21

1. 找到pluging点击选择浏览资源

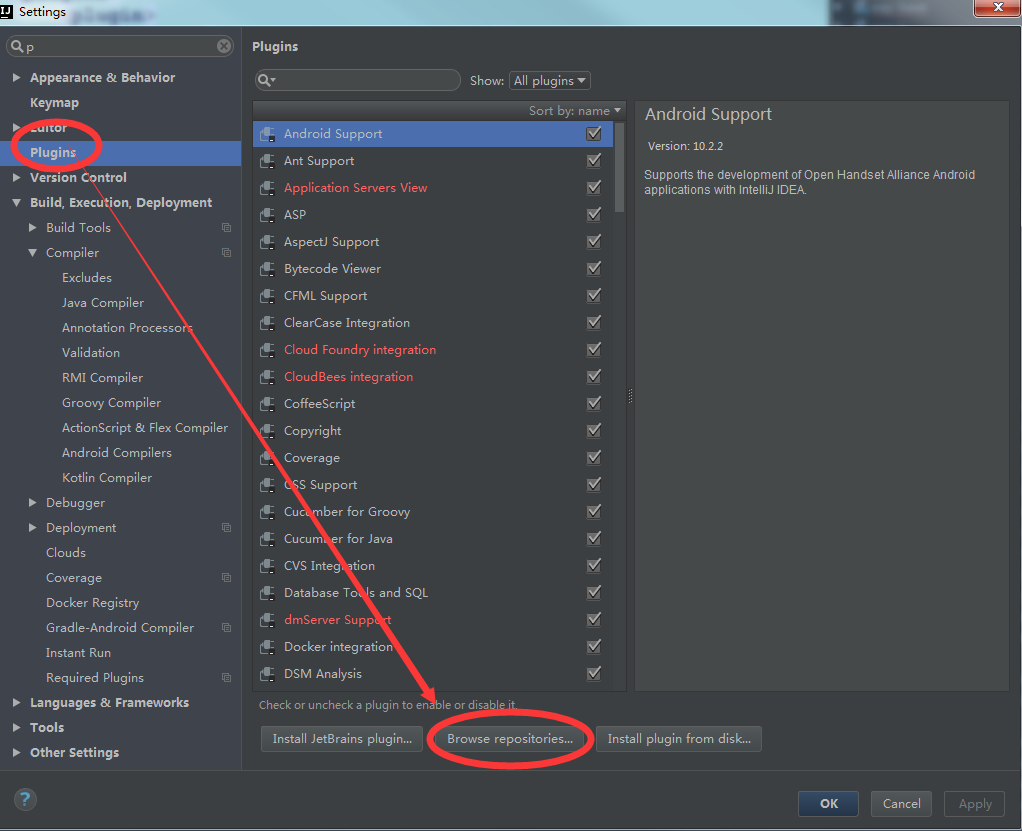


Figure 0‑22

1. 搜索lombok点击安装

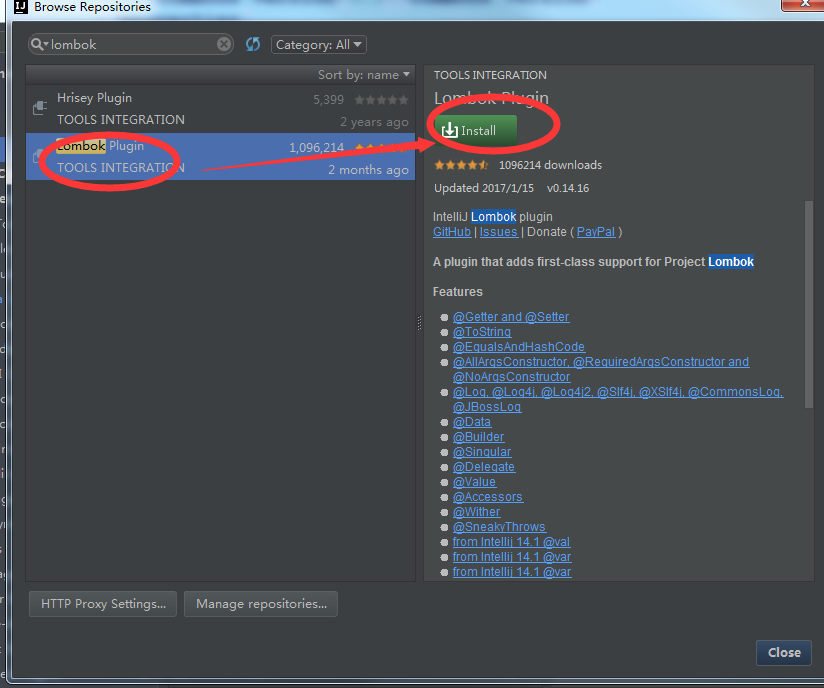


Figure 0‑23

1. 安装后点击重启Intellij Idea

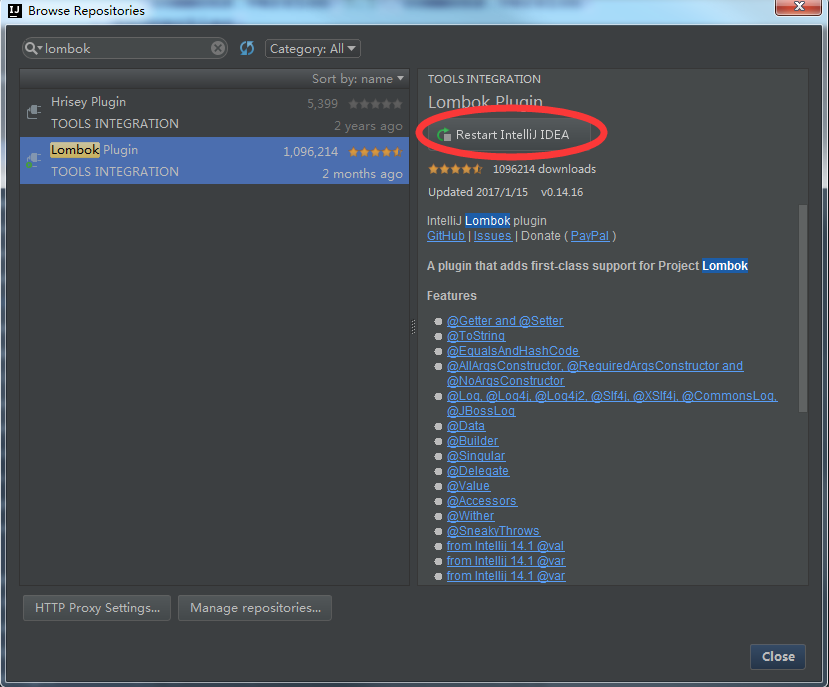


Figure 0‑24

* Eclipse配置

1. 下载lombok <http://projectlombok.org/download.html>
2. 点击安装：



Figure 0‑25

如果eclipse没有安装到默认目录，那么需要点击Specify选择eclipse的安装文件，然后Install即可完成安装

1. 如果程序还在报错，那么点击在eclipse的Project选项的clean

Lombok常用注解API

Lombok的特色是根据annotation创建一些代码，以减少重复代码的数量，它提供了以下几个annotation：

* @Getter和@Setter：为属性创建getter和setter
* @EqualsAndHashCode：实现equals()方法和hashCode()方法
* @ToString：实现toString()方法
* @Data：上述3个annotation的和，会创建getter setter equals hashCode和toString （最实用）
* @Cleanup：关闭流
* @Synchronized：对象同步
* @SneakyThrows：抛出异常
* @Log4j: log4j日志声明

##### Swagger

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器以同样的速度来更新。文件的方法，参数和模型紧密集成到服务器端的代码，允许API来始终保持同步。Swagger 让部署管理和使用功能强大的API从未如此简单。

* 依赖Jar如下：

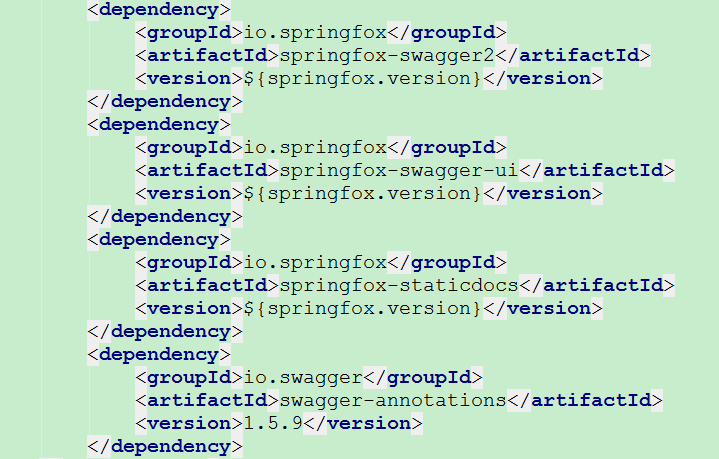


Figure 0‑26

* 注入Swagger相关Configuration



Figure 0‑27

* Swagger常用注解API介绍：
* @Api：用在类上，说明该类的作用
* @ApiOperation：用在方法上，说明方法的作用
* @ApiImplicitParams：用在方法上包含一组参数说明
* @ApiImplicitParam：用在@ApiImplicitParams注解中，指定一个请求参数的各个方面 paramType：参数放在哪个地方
* @Api：用在类上，说明该类的作用
* @ApiOperation：用在方法上，说明方法的作用
* @ApiImplicitParams：用在方法上包含一组参数说明
* @ApiImplicitParam：用在@ApiImplicitParams注解中，指定一个请求参数的各个方面 paramType
* @ApiResponses：用于表示一组响应
* @ApiResponse：用在@ApiResponses中，一般用于表达一个错误的响应信息 code
* @ApiModel：描述一个Model的信息（这种一般用在post创建的时候，使用@RequestBody这样的场景，请求参数无法使用@ApiImplicitParam注解进行描述的时候）
* @ApiModelProperty：描述一个model的属性
* @ApiResponses：用于表示一组响应
* @ApiResponse：用在@ApiResponses中，一般用于表达一个错误的响应信息 code
* @ApiModel：描述一个Model的信息（这种一般用在post创建的时候，使用@RequestBody这样的场景，请求参数无法使用@ApiImplicitParam注解进行描述的时候）
* @ApiModelProperty：描述一个model的属性
* Swagger-ui管理路径效果如下：

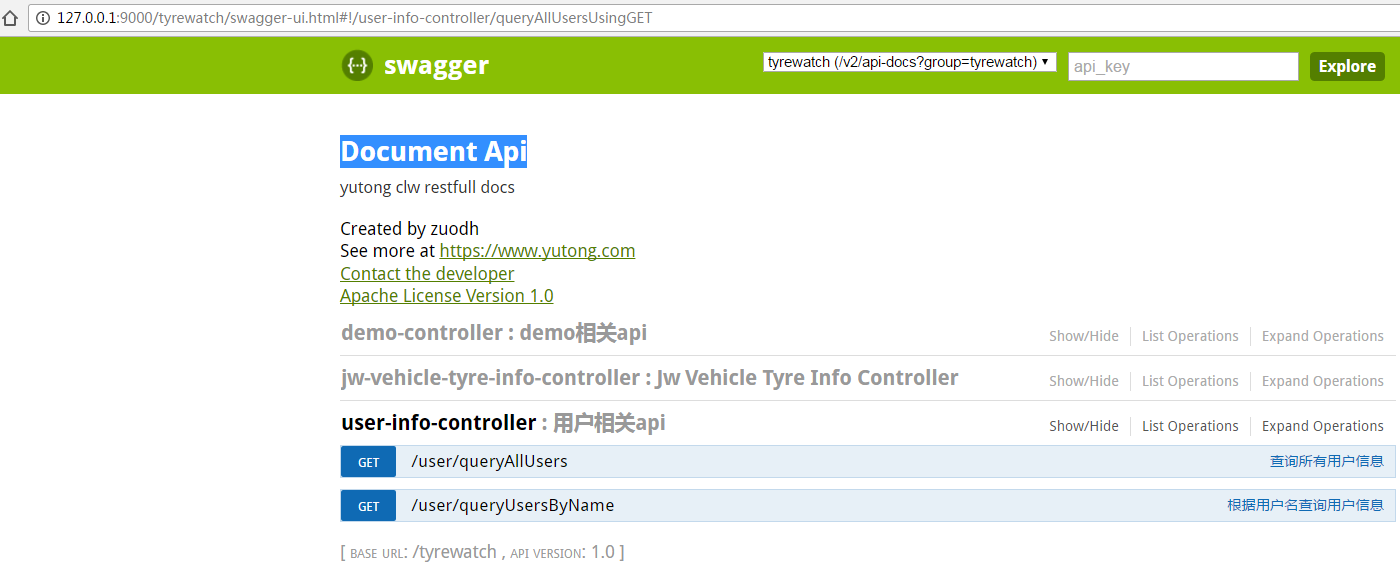


Figure 0‑28

##### Hystrix

[Hystrix](https://github.com/Netflix/Hystrix) 是世界最大在线影片租赁服务商Netflix开源，针对分布式系统的延迟和容错库。该库由Java写成，项目源于Netflix API团队在2011年启动的弹性工程项目。项目在github上发布至今，已经有接近三千颗星，只有少数优秀的开源项目才能享受到千星级别的待遇，Hystrix成功可见一斑。

* 依赖Jar如下：

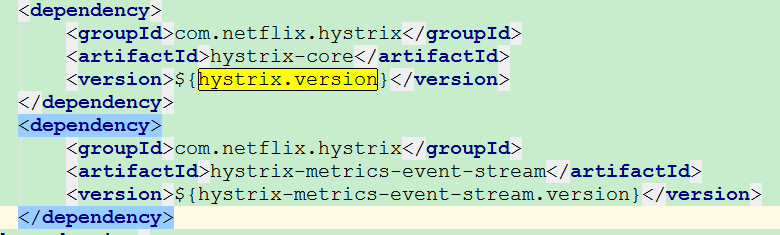


Figure 0‑29

* 为什么使用Hystrix

在大中型 [分布式系统](http://www.cxyclub.cn/Tag/fenbushixitong.html) 中，通常系统很多依赖( [HTTP](http://www.cxyclub.cn/Tag/http.html) ,hession,Netty,Dubbo等)，如下图:

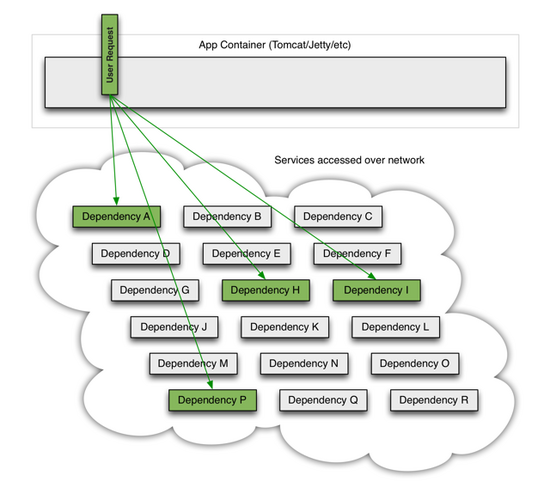


Figure 0‑30

在高并发访问下,这些依赖的稳定性与否对系统的影响非常大,但是依赖有很多不可控问题:如网络连接缓慢，资源繁忙，暂时不可用，服务脱机等。

* Hystrix可以完成如下功能：
* Hystrix使用命令模式HystrixCommand(Command)包装依赖调用逻辑，每个命令在单独线程中/信号 [授权](http://www.cxyclub.cn/Tag/shouquan.html) 下执行
* 提供熔断器组件,可以自动运行或手动调用,停止当前依赖一段时间(10秒)，熔断器默认 [错误](http://www.cxyclub.cn/Tag/cuowu.html)率阈值为50%,超过将自动运行。
* 可配置依赖调用 [超时](http://www.cxyclub.cn/Tag/chaoshi.html) 时间,超时时间一般设为比99.5%平均时间略高即可.当调用超时时，直接返回或执行fallback逻辑。
* 为每个依赖提供一个小的线程池（或信号），如果线程池已满调用将被立即拒绝，默认不采用排队.加速失败判定时间。
* 依赖调用结果分:成功，失败（抛出 [异常](http://www.cxyclub.cn/Tag/yichang.html) ），超时，线程拒绝，短路。 请求失败(异常，拒绝，超时，短路)时执行fallback(降级)逻辑。
* 提供近实时依赖的统计和监控

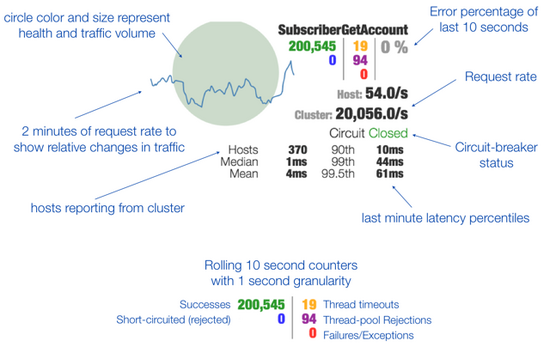


Figure 0‑31

* 支持异步执行。支持并发请求缓存。自动批处理失败请求。

#### 开发架构

平台后端开发架构如下图：

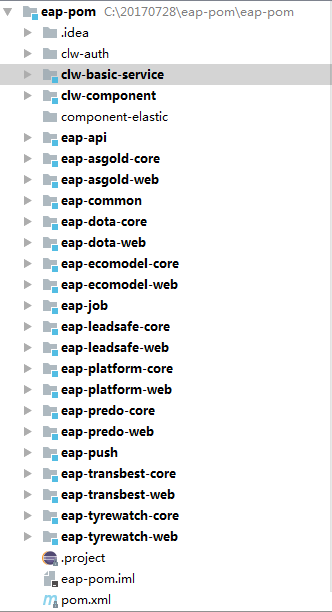


Figure 0‑32

后端主要通过RPC服务的方式暴露接口，供前端调用。开发架构按照专业产品拆分，每一个专业产品会是一个单独的微服务。

每个专业产品都有core和web两个基础项目构成，如安全类专业产品eap-leadsafe-core和eap-leadsafe-web。Core子项目主要实现所属专业产品项目的业务逻辑部分并且提供基于RPC的服务方式，而web子项目主要实现mvc controller这一层的相关内容，并且实现基于HTTP的服务方式。Eap-api项目是各个专业产品项目的接口定义。

#### 子系统的实现

如平台整体架构图，V+平台有很多子系统构成，而每个专业产品都是一个独立的微服务，每个微服务中都有若干服务接口。下文以车辆基础信息模块和能耗业务模块为例介绍微服务的具体实现。

##### 车辆基础信息模块实现

车辆基础信息模块是一组车辆基础信息（如：车牌号、车辆VIN号、轮胎、发动机型号等等）的接口组合，它以微服务的形式对外提供服务。

如下图，首先需要在eap-platform-core子项目中添加接口的定义。eap-platform-core和其他专业产品的子项目不同，platform定义为平台的基础子项目，它的一些方法和接口会被其他的专业产品使用。



Figure 0‑33

接口定义完成之后，需要对其进行实现。车辆的基础信息比较简单，基本都是用数据库中直接获取，如果遇到性能问题，可以考虑使用Redis进行数据缓存。简单的实现代码如下：

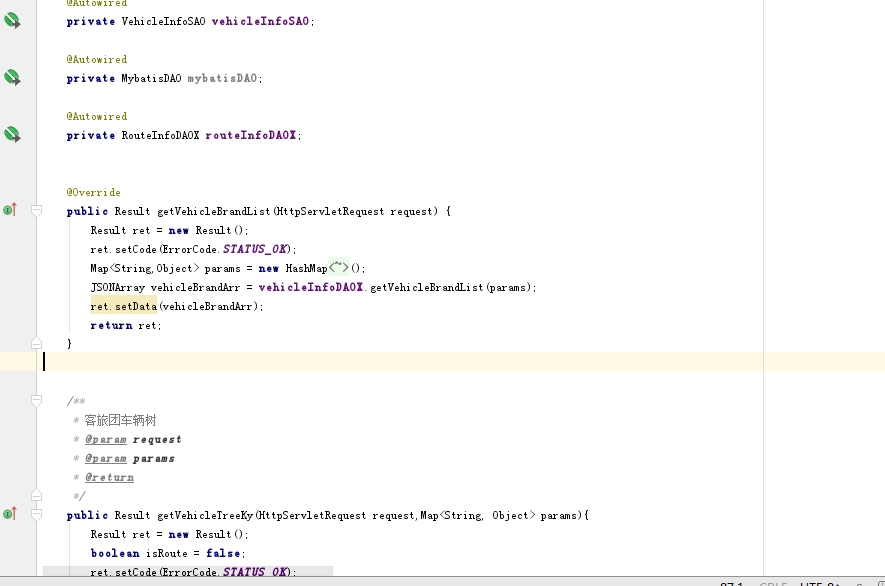


Figure 0‑34

### 数据库设计

由于整个数据库设计比较庞大，由于本文只是展示了车辆基础信息服务，所以只列出了基础信息相关的数据表结构。

#### ER图

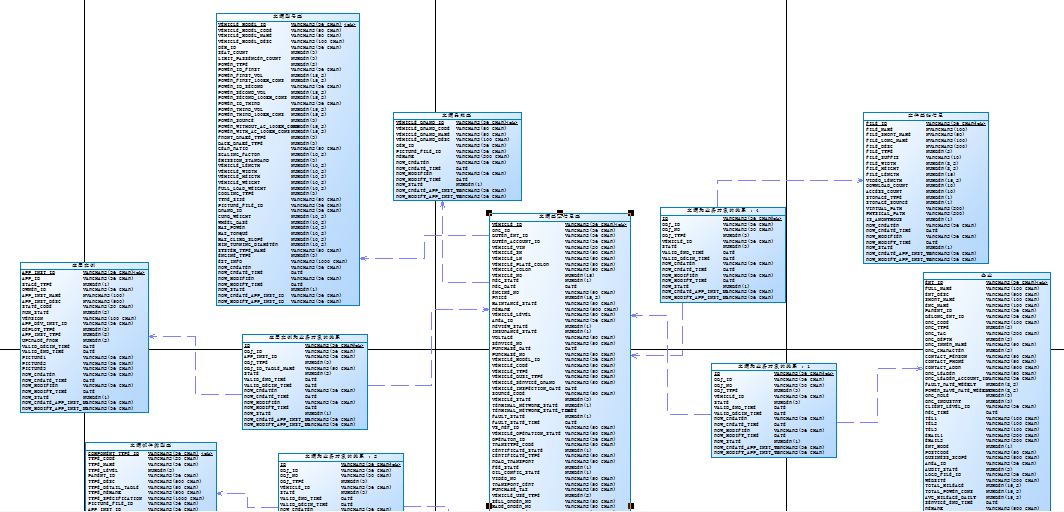


Figure 0‑35

#### 基础数据表介绍

|  |  |  |
| --- | --- | --- |
| **表名** | **表名** | **描述** |
| TB\_VEHICLE | 车辆表 | 车辆基础信息表 |
| TB\_ORGNIZATION | 组织表 | 车辆所归属的组织，是一个树状结构 |
| TB\_TERMINAL | 终端表 | 车上安装的终端的信息，和车辆表是一对多的关系。一个车辆可以安装多个终端。 |
| TB\_VEHICLE\_BRAND | 车辆品牌表 | 车辆的品牌，类似宇通，金龙，奔驰等。 |
| TB\_VEHICLE\_MODEL | 车辆型号表 | 车辆的型号，和车辆表和品牌表都是一对多的关系 |

### 服务治理

#### 服务提供者

由于使用了Dubbo框架，所以当系统启动起来之后，会有一个服务治理的管理界面，方便我们进行服务治理。其中有一些监控和配置信息。

下图展示了我们系统成功起来以后，在“提供者”一栏出现了一个服务，就是我们系统的服务。列表中显式了服务的IP、权重、类型、状态等信息，还有一些编辑和禁用的操作功能。

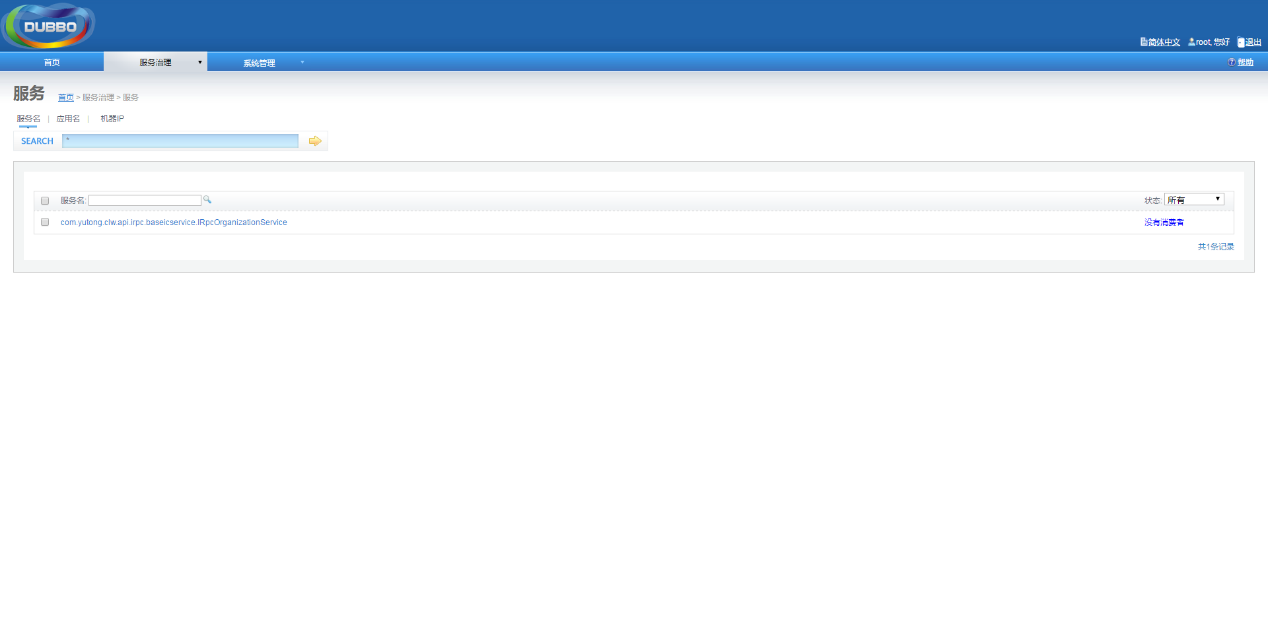


Figure 0‑36

选择上图中的服务链接，点击进行就能看到服务提供的接口列表。我们的例子中只实现了一个接口，组织相关的接口。下图显示了接口的详细信息。

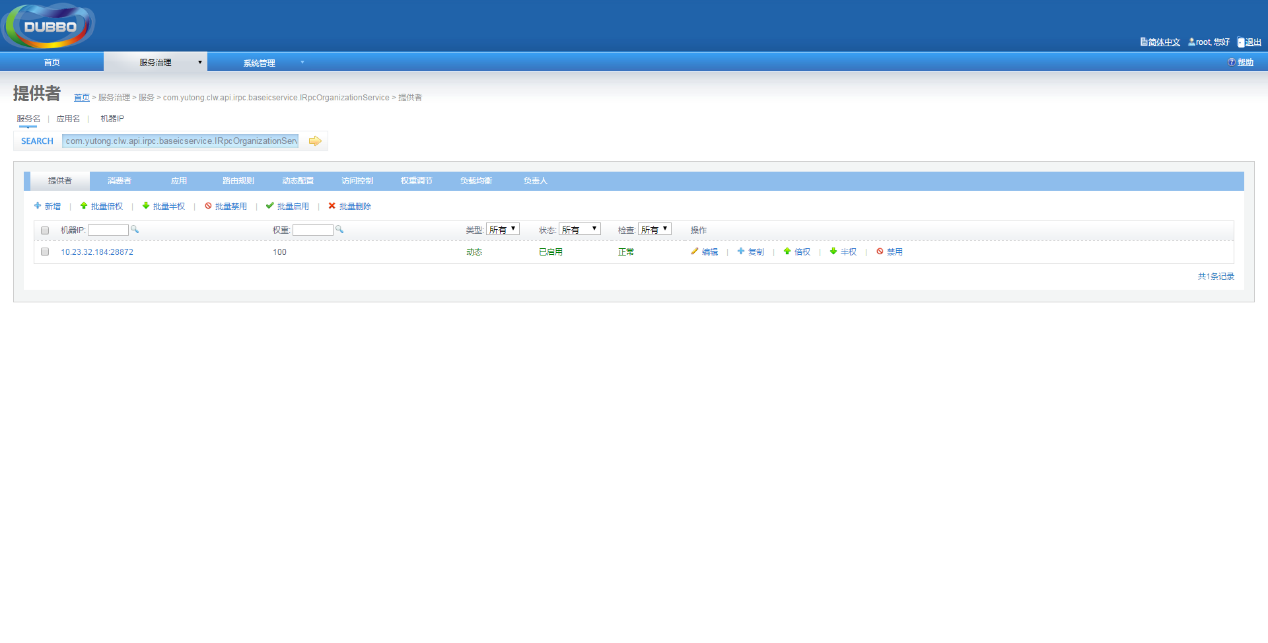


Figure 0‑37

#### 服务消费者

当有客户端调用的时候，就能在消费者页面中看到消费者的信息，如下图：



Figure 0‑38

#### 路由规则

路由规则允许管理员对客户端和服务端的访问路由做配置。比如，某个服务只允许某个IP的客户端使用。如下图：



Figure 0‑39

### 服务的限流

Dubbo框架并没有直接提供限流的解决方案，直接使用上文研究的Guava框架实现。Guava框架实现了令牌算法的限流方案。

#### Guava框架介绍

Guava是一个Google开源项目，包含了若干被Google的Java项目广泛依赖的核心库，其中的RateLimiter提供了令牌桶算法实现：平滑突发限流（SmoothBursty）和平滑预热限流（SmoothWarmingUp）实现。

#### 限流实现

在每个服务中心设置限流的开关，然后可以在服务的配置文件中进行配置开启或者关闭，甚至是限流的策略。

可以在需要实现限流的代码中添加如下片段：

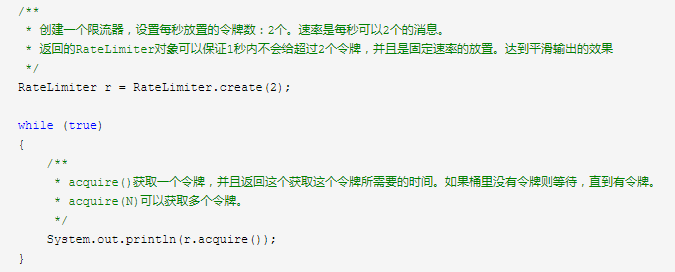


Figure 0‑40

### 服务的熔断

在dubbo中对于消费者的保护提供了actives进行并发控制保护，但是功能相对薄弱。在V+平台的微服务实现中，采用了Hystrix框架对微服务进行熔断策略处理。

#### Hystrix简介

Hystrix源于Netflix，该库可以对那些在访问远程系统、服务、第三方库或者其他方法调用时所产生的延迟或者故障提供更强大的容错能力。说白了就是提供了一种简洁、灵活和强大的框架去处理依赖异常，一个服务挂了不影响其他的服务正常运行，也是微服务概念中熔断，降级的具体框架实现。

它主要的功能如下：

* 对第三方组件进行访问（特别是网络访问）时出现的延迟或者的异常进行保护和控制。
* 在复杂的分布式系统中，停止级联故障。
* 快速失败，快速恢复。
* 可以优雅的降级。
* 可以提供近实时的监控、告警和操作控制。

#### 使用Hystrix实现熔断

Hystrix把每个潜在的依赖性进行包装，把每一个依赖项都与另一个依赖项隔离开来，当潜在的问题发生时能限制它的资源，并在调用发生异常时能够决定如何应答给调用方，从而快速的恢复处理，不影响下面的调用。Hystrix可以配置两种策略，异常和超时，那么只要有一样触发，或者线程被占满，就可以马上进入我们预设的异常处理逻辑。

实现代码例子如下：

import com.netflix.hystrix.HystrixCommand;

import com.netflix.hystrix.HystrixCommandGroupKey;

import com.netflix.hystrix.HystrixCommandProperties;

import java.util.concurrent.TimeUnit;

/\*\*

\* 使用HystrixCommand封装执行方法。继承Hystrix命令基类，泛型参数即调用方法的返回参数

\*/

public class HelloWorldCommand extends HystrixCommand<String>

{

private final String name;

/\*\*

\* 定义构造函数，参数即被包装方法的输入参数

\* @param name

\*/

public HelloWorldCommand(String name)

{

//定义命令组 和 方法调用超时时间

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))

.andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(5000)));

this.name = name;

}

/\*\*

\* 封装业务逻辑的方法体，在这里执行真正的业务逻辑。

\* @return

\* @throws Exception

\*/

@Override

protected String run() throws Exception

{

//例子中，模拟执行超时

if (name.equals("TimeoutCommand"))

{

TimeUnit.SECONDS.sleep(6);

}

//例子中，模拟出现异常

if (name.equals("ExceptionCommand"))

{

throw new Exception("ExceptionCommand");

}

//例子中，模拟正常返回

return "Hello " + name + "!";

}

/\*\*

\* 降级方法定义，即，run方法中出现异常后应该执行的方法。包括例子中的超时。

\* @return

\*/

@Override

protected String getFallback()

{

return "Command failed!";

}

}

### 配置中心

#### 配置中心简介

随着程序功能的日益复杂，程序的配置日益增多：各种功能的开关、参数的配置、服务器的地址等等，同时对程序配置的期望值也越来越高：配置修改后实时生效，分环境、分集群管理配置，完善的权限、审核机制等等。因此，平台采用开源的系统配置Apollo来构建车联网的配置中心。

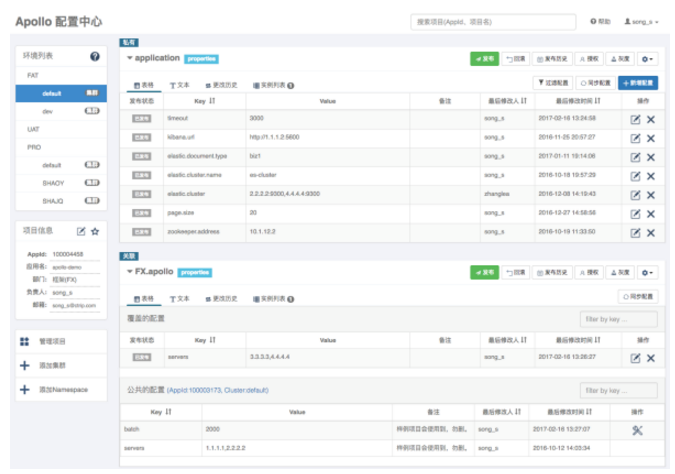


Figure 0‑41

#### 配置中心功能

1. 统一管理不同环境、不同集群的配置

Apollo提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。同一份代码部署在不同的集群，可以有不同的配置，比如zk的地址等。通过命名空间（namespace）可以很方便的支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖。

1. 配置修改实时生效（热发布）

用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最新的配置，并通知到应用程序。

1. 版本发布管理

所有的配置发布都有版本概念，从而可以方便的支持配置的回滚。

1. 灰度发布

支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例。

1. 权限管理、发布审核、操作审计

应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。所有的操作都有审计日志，可以方便的追踪问题。

1. 客户端配置信息监控

可以方便的看到配置在被哪些实例使用

1. 提供Java和.Net原生客户端

提供了Java和.Net的原生客户端，方便应用集成。支持Spring Placeholder和Annotation，方便应用使用（需要Spring 3.1.1+）。同时提供了Http接口，非Java和.Net应用也可以方便的使用

1. 提供开放平台API

Apollo自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。不过Apollo出于通用性考虑，对配置的修改不会做过多限制，只要符合基本的格式就能够保存。在我们的调研中发现，对于有些使用方，它们的配置可能会有比较复杂的格式，如xml, json，需要对格式做校验。还有一些使用方如DAL，不仅有特定的格式，而且对输入的值也需要进行校验后方可保存，如检查数据库、用户名和密码是否匹配。对于这类应用，Apollo支持应用方通过开放接口在Apollo进行配置的修改和发布，并且具备完善的授权和权限控制

1. 部署简单

配置中心作为基础服务，可用性要求非常高，这就要求Apollo对外部依赖尽可能地少。目前唯一的外部依赖是MySQL，所以部署非常简单，只要安装好Java和MySQL就可以让Apollo跑起来。Apollo还提供了打包脚本，一键就可以生成所有需要的安装包，并且支持自定义运行时参数

### 发布与部署

#### 系统打包

对通过Maven对应用程序进行打包，包的格式是zip文件。打包内部结构效果图如下：

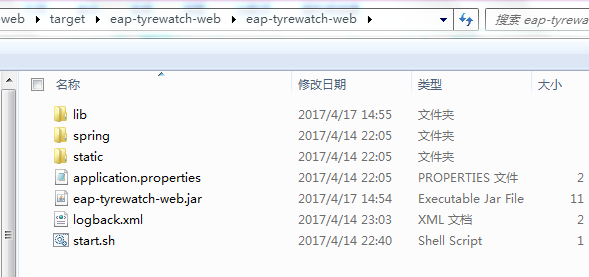


Figure 0‑43

* Lib下面是工程依赖的jar；
* Static放的是前端代码；
* Spring下面是放的spring相关配置；
* Start.sh是启动脚本；

#### 发布部署

微服务程序是运行在Linux上的，所以首先需要准备Linux环境，并且确认有安装Java8及以上的环境，由于使用SpringBoot开发，服务使用内置的Web容器，所以不需要安装如Tomcat这类Web容器。下面以轮胎服务为例简单介绍部署的基本流程。

1. 安装：把打包文件放置在linux服务器上并解压eap-tyrewatch-web.zip；
2. 配置：进入解压目录，在配置文件application.properties里把相应的数据库、redis、等等信息替换为实际的环境；如果遇到端口冲突相应的需要修改端口；
3. 启动：环境配置完成后即可启动：sh start.sh eap-tyrewatch-web.jar；
4. 验证：通过url：<http://ip:port/tyrewatch/health>查看服务的健康信息；

### 系统安全性解决方案

#### 安全方案概述

整个系统安全主要涉及到：系统级安全、功能性安全、数据安全等。

#### 系统级安全

类似于IP访问控制、连接时段、登录数控制这类系统级安全防护采用硬件防火墙来实现。防火墙最基本的功能是确保网络流量的合法性，并在此前提下将网络的流量快速的从一条链路转发到另外的链路上去。防火墙将网络上的流量通过相应的网络接口接收上来，按照OSI协议栈的七层结构顺序上传，在适当的协议层进行访问规则和安全审查，然后将符合通过条件的报文从相应的网络接口送出，而对于那些不符合通过条件的报文则予以阻断。

#### 功能性安全

功能性安全主要通过功能权限来实现。所谓功能权限就是有或者没有做某种操作的权限，具体表现形式就是你看不到某个菜单或按钮，当然也有的是把菜单或按钮灰掉的形式。实际上它的实现机制比表面上看到的要复杂得多，比如：我们从浏览器访问过一个地址之后，实际上这个URL就会在历史中存在，这时就会存在一种可能，有的人虽然没有权限，但是他知道怎么访问的URL，如果他再有一定的技术基础，那么通过猜测，有时候就可以得到真正的操作的URL，这个时候如果操作权限限制做得不到位，那么他就可能做他非授权的事项，所以操作权限一般都在显示界面做一次控制，过滤没有权限的操作菜单或按钮，另外在真正执行操作时，还要进行一次权限检查，确保控制非授权访问。

平台通过通用的功能权限设计可以有效的控制功能的安全性。

#### 数据安全性

##### 数据权限

所谓数据权限，就是有或者没有对某些数据的访问权限，具体表现形式就是当某用户有操作权限的时候，但不代表其对所有的数据都有查看或者管理权限。数据权限有两种表现形式：一种是行权限、另外一种是列权限。所谓行权限，就是限制用户对某些行的访问权限，比如：只能对本人、本部门、本组织的数据进行访问；也可以是根据数据的范围进行限制，比如：合同额大小来限制用户对数据的访问。所谓列权限，就是限制用户对某些列的访问权限，比如：某些内容的摘要可以被查阅，但是详细内容就只有VIP用户才能查看。

平台通过通用的数据权限设计可以有效的控制数据的安全性。

##### 使用安全的密码策略

V+把密码策略摆在所有安全配置的第一位，健壮的密码是安全的第一步，建议密码含有多种数字字母组合并9位以上。并且会有密码策略的定期巡检，检查看是否有不符合密码要求的账号。

##### 数据库备份策略

V+的数据库备份有完整的策略。BSP数据库系统常规备份使用rman周日进行完全备份，周一至周六进行差异增量备份。

##### 数据库权限

Oracle是多用户系统，它允许许多用户共享系统资源。为了保证数据库系统的安全，数据库管理系统配置了良好的安全机制。

* 建立系统级的安全保证

系统级特权是通过授予用户系统级的权利来实现，系统级的权利(系统特权)包括：建立表空间、建立用户、修改用户的权利、删除用户等。系统特权可授予用户，也可以随时回收。系统级的权限只授予数据库DBA。

* 建立对象级的安全保证

对象级特权通过授予用户对数据库中特定的表、视图、序列等进行操作(查询、增、删改)的权利来实现。

* 建立用户级的安全保证

用户级安全保障通过用户口令和角色机制(一组权利)来实现。引入角色机制的目的是简化对用户的授权与管理。做法是把用户按照其功能分组，为每个用户建立角色，然后把角色分配给用户，具有同样角色的用户有相同的特权。

##### 防止SQL注入

主要通过如下措施来防止SQL注入：

* 数据库权限控制，只给访问数据库的web应用功能所需的最低权限帐户。
* 把危险的和不必要的存储过程删除
* 使用参数化的SQL，使用绑定变量

### 质量

#### 可扩展性

V+会面对未来百万级物联网设备的接入，无论是业务、吞吐量还是性能对需要具备很强的可扩展性。一般变化范围会集中在以下几点。

##### 业务的扩展性

##### 可扩展的通信协议

物联网业务的扩展很大部分最后会落地到通信协议的扩展。从整个数据流可以看出从CP解析层只有的通信协议都是使用 Protobuf和Json这样的数据传输格式。这种传输协议非常有利于数据项的扩展，往往只需要极少的开发量就能实现协议的修改。

##### 业务处理逻辑的增加

需求的变化是永恒不变的。对于新业务的添加除了单个应用程序本身代码的解耦程度之外，我们更关注进程级别的业务扩展性。在V+中，系统基本都是通过kafka或者RabbitMQ来进行数据传输和业务链式处理的。所以遇到新业务的开发处理可以在本身已有的进程上修改代码之外，还可能再产生一个新的消费者进行数据消费，这种方式一般在遇到性能问题，或者需要并行处理的时候使用。

##### 吞吐量的可扩展

###### 实时计算的扩展

V+的实时计算主要在Pisces和Aries上。Pisces是一个基于Spark Streaming的实时计算框架，计算资源的分配依赖于Hadoop Yarn。Yarn的计算资源是可以实时动态调整的，一旦YARN的整体资源不够，可以通过添加机器的方式实现。Aries是一个Java开发的应用程序，作为一个应用程序整体来说，是无状态的。也就是说，可以通过消息中间件把消息分发给Aries处理，多个Aries可以并行处理。如果性能出现问题可以通过添加机器的方式实现吞吐量的提升。

###### 数据存储的性能和容量扩展

数据存储是整个系统吞吐量提升的关键，无论实际计算的架构有多完美，只要数据存储出现瓶颈就会导致整个系统出现性能问题，吞吐量上不去。下面列出了V+系统的主要存储和扩展方式：

* Oracle：当数据量增大的时候，虽然有分区表可以在磁盘IO不是瓶颈的时候提升性能，但是IO还是在一个存储上面，当整体IO都维持在较高水平的时候，分区表的作用就不是很明显。目前的方案是将这部分数据迁移到大数据存储上，例如HBase、ElasticSearch。
* Redis：采用Redis Cluster架构，可以方便的进行横向扩展，无论是IO还是容量，都可以采用这样的方式。
* HBase：运行在Hadoop HDFS上，HDFS分身具备横向扩展能力。HBase可以充分利用其能力实现IO和容量扩展。
* ElasticSearch：它本身就是无中心架构，可以轻松的通过添加机器实现。原有的索引也能rebalance到新的机器上。
* Ceph：分布式的对象存储服务器。天生具备扩展能力。

#### 可伸缩性

可伸缩性及系统的弹性，V+所有的核心子系统都可以集群部署，根据接入数据多少和访问量的多少进行动态调配。上文中，可扩展性以及下文中的可靠性中的内容中基本描述了具体的实现。

#### 可靠性（高可用性）

V+系统需要满足99.95%的高可用性。V+系统的核心模块都能实现负载均衡，非核心模块使用主备切换来实现高可用。

##### 微服务高可用

微服务采用Dubbo框架，主要通过集群容错和负载均衡来实现微服务的高可用。

###### 集群容错

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。

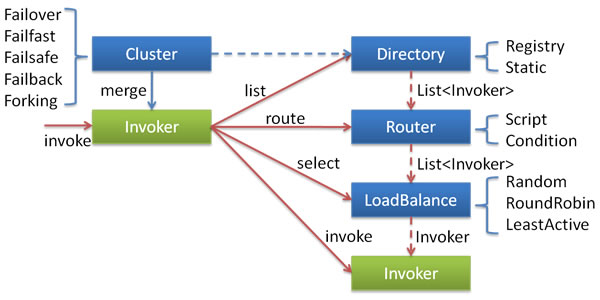


Figure 0‑44

各节点关系：

* 这里的 Invoker 是 Provider 的一个可调用 Service 的抽象，Invoker 封装了 Provider 地址及 Service 接口信息
* Directory 代表多个 Invoker，可以把它看成 List<Invoker> ，但与 List 不同的是，它的值可能是动态变化的，比如注册中心推送变更
* Cluster 将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
* Router 负责从多个 Invoker 中按路由规则选出子集，比如读写分离，应用隔离等
* LoadBalance 负责从多个 Invoker 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

###### 集群容错模式

* Failover Cluster

失败自动切换，当出现失败，重试其它服务器 1。通常用于读操作，但重试会带来更长延迟。

* Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

* Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

* Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

* Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

* Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错 2。通常用于通知所有提供者更新缓存或日志等本地资源信息。

###### 负载均衡

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 random 随机调用。

###### 负载均衡策略

* Random LoadBalance

随机，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

* RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

* LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

* ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

##### 数据存储的高可用

###### Oracle数据库的高可用

V+系统中唯一的关系型数据里Oracle采用RAC技术来实现数据库的高可用。Oracle RAC的核心是共享磁盘子系统，集群中所有节点必须能够访问所有数据、重做日志文件、控制文件和参数文件，数据磁盘必须是全局可用的，允许所有节点访问数据库，每个节点有它自己的重做日志和控制文件，但是其他节点必须能够访问它们以便在那个节点出现系统故障时能够恢复。

Oracle RAC 运行于集群之上，为 Oracle 数据库提供了最高级别的可用性、可伸缩性和低成本计算能力。如果集群内的一个节点发生故障，Oracle 将可以继续在其余的节点上运行。

###### HBase的高可用

HBase的数据存储在HDFS上，数据存储可靠性和高可用依赖于HDFS。V+1.0由于没有配置NameNode的HA所以由单点的故障，但是在新的V+架构中将采用NameNode的高可用架构，从而彻底解决Hadoop集群的单点故障问题。

###### ElasticSearch的高可用

V+的ElasticSearch是集群部署，集群中有一个Master节点和多个Data节点。如果一个Master节点宕机，则其他的节点会选举出另一个Master节点。

存储在ElasticSearch中的数据是可以设置备份数量的，默认是2个。一份数据会存储在两台不同的机器上。从而保证了高可用。

###### Redis集群的高可用

V+的Redis架构采用Redis官方的Cluster集群架构。在物理部署上采用3主3从，六台机器的部署方式。如果某一台Master宕机，则对应的Slave会成为Master节点提供服务。下图为Redis集群架构。

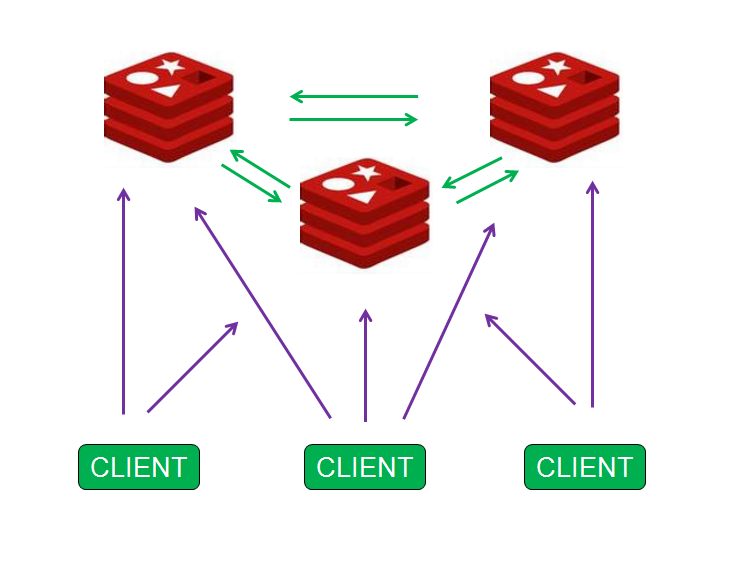


Figure 0‑45

##### 其他应用系统的高可用

应用系统的主要通过消息中间件负载均衡的方式来实现高可用。少数的没有负载均衡的应用则采用zookeeper来实现主备切换。

###### 使用消息中间件

CP的接入层和解析层使用kafka来进行解耦，也是高可用的一种实现。接入层和解析层都是集群部署，当解析层中某一台机器宕机，原本由宕机的服务器处理的消息，通过kafka，则会被其他的解析层的服务器处理。

同样的，解析层到Pisces，再到Aries，都是用kafka或者RabbitMQ来实现消息的分发，一个节点的宕机，不会影响数据的完整性和可靠性，也实现了高可用。

###### 使用zookeeper

对于那些只能单机运行的子系统则可以通过zookeeper实现主备切换，如果主节点宕机，和zookeeper失去联系，同样连在zookeeper上的另一个节点就会替换原来主节点的功能，升级为主节点。

#### 可维护性

系统的可维护性是衡量一个系统的可修复性和可改进性的难易程度。所谓可修复性是指在系统发生故障后能够排除故障予以修复，并返回到原来正常运行状态的可能性。

##### 子系统的划分

提高应用系统可维护性的重要的措施就是分割系统，将系统中的业务逻辑尽可能分割为相互独立、容易理解的子系统。V+将整个业务处理的流程主要分解为：接入、解析、分析计算、消息处理等多个步骤，每个步骤成为一个子系统独立运行和部署，之间，使用消息中间件解耦。如果每一步骤出现问题，可以方便的定位、修复或替换，不影响其他流程上的业务，减少故障对其他子系统的影响。

##### 日志平台

V+有功能较为齐全的日志平台可以权限分析系统的异常情问题和运行情况。每个子系统都必须使用统一的日志格式进行日志记录。日志平台把这些日志统一收集到日至平台的大数据存储中，供后续业务的查询和分析。

* 业务日志：记录业务数据或业务流程，便于后续分析业务异常和系统执行情况。
* 异常日志：当系统出现异常的事情，可以通过日志分析异常，发现问题。
* 性能日志：通过埋点实时记录系统的性能参数、队列长度等性能日志，用于分析宕机、预判异常、提前告警。
* 调用链日志：记录数据或者请求走过的业务模块，用于分析业务异常、性能问题。

##### 完善的监控体系

V+通过zabbix全面监控系统每一个子系统的服务器情况、进程情况、端口等监控项，同时也结合业务日志对每个子系统的业务运行情况进行监控，做到全方位、无盲区。Zabbix也配置成和短信系统、邮件系统相连，发现问题及时通知到相关人员。

#### 安全性

##### 接入安全

V+通信平台对外通信的信息安全性主要是基于外部通信协议中规定的安全策略来实现，内部通信协议与内部数据之间的安全，则是依靠系统内部的设计机制。目前通信平台暴露在外网的服务端口主要是用于提供车载终端接入的监听服务端口，其他的设备大多数都是在内网间进行，安全性要求较低。当前车载终端的监听服务端口主要是采用了一些基本的个性化定制协议来实现通信数据安全，对于数据库的数据安全则是通过对局部敏感数据的加密来实现。当前通信平台的设计过程中主要考虑到以下相关部分的安全策略。

###### 拒绝服务攻击防范

除了常规的安全考虑外，通信类服务器还要考虑端口句柄耗尽等风险，需要与服务器端口建立的socket连接进行有效判断，如果一个socket从建立连接开始，30s内未有数据流动，则认为该socket可能是一个无效socket，要及时关闭，并释放句柄，以避免因拒绝服务攻击，系统的句柄被耗尽而导致其他终端无法连接。

###### 数据加密

当前通信平台与车载终端的通信协议规定中，支持对上下行数据的加密，根据协议帧的状态位来识别数据是否加密，如果加密则按照特定的解密算法解析

###### 令牌验证

车载终端连入通信平台后，通信平台会在车载终端注册验证成功之后，生成一个全局唯一的令牌码，后续车载终端登陆时，直接通过该令牌完成与中心系统的身份安全校验工作。

###### 服务器与客户端双证书验证

当前的通信协议规范中支持对于服务器与客户端进行双证书验证，但是考虑到目前系统安全需求尚未达到该级别，故并未在业务层面实现，后续随着业务的发展，遇到有此类高安全需求时，可以进行双证书验证的具体实现。

##### 数据安全

###### 应用认证

V+平台对每个应用都有认证和鉴权，应用如果想访问V+的数据，首先要经过平台的应用认证，才可以获取V+的数据。

###### 用户认证

无论是手机App用户还是Web网页用户都需要通过后端服务的鉴权，用户的密码是通过MD5加密后存储在数据库中。

###### 多租户的系统设计

V+系统采用多租户的系统设计，每个企业的数据逻辑隔离。一般情况下一个企业的用户不会看见其他企业的数据。

###### 功能权限和数据权限

每个用户登录之后，平台会赋予其功能权限和数据权限。保证用户只能看到背授权的功能和数据。

# 第五章 主要结论与展望



## 主要结论

本文主要研究了微服务以及微服务在车辆网行业中应用的相关理论和技术，特别在微服务的服务治理方面进行了大量深入的研究。并在调研的基础上形成需求方案，设计并实现基于微服务技术的车联网平台。

本文首先对微服务的概念、历史和由来阐述了整个微服务理念和框架的发展历程。然后通过微服务特性的研究、与传统单体应用的优缺点对比、以及微服务的核心功能来更加深入的探讨微服务理念和架构在特定的软件开发阶段的可行性和先进性。同时，也阐述了自动化运维对于微服务架构的重要性。

之后，作者根据软件工程的设计理论，进行总体设计、概要设计、详细设计，完成系统的的详细和全面的设计，为系统的实现、部署打下基础。最后，通过部署系统的部署实施环境，解决系统功能模块开发、业务数据处理、系统部署和测试问题，为系统开发的完成把好关键环节关。

通过本课题的研究和系统开发工作锤炼，使论文作者对微服务的应用理论和技术体系有了更加深入的理解，对车联网技术平台的开发和应用有了更加深入的掌握和应用，进一步提高了大型物联网平台开发工作的能力，特别是当前不断深入的信息化建设，随着规模的扩大和应用要求的不断提高，更加明确了发展和努力的方向。

课题研究和系统开发基本完成后，进行了软件系统的部署与实施，解决系统运作和业务数据处理和管理等中存在的许多问题，借助测试工具和理论实现对系统的功能模块和业务流程的完善，并根据用户的意见，考虑用户的广泛性还对系统的操作界面进行简化，实现系统成功的部署并开始试运行。

在对课题相关研究的基础上，一方面总结了系统需求分析、概要设计、实现与完善等环节的工作，并对存在的问题进行了认真全面地分析，整理了相关的资料，根据研究思路组织文字撰写论文；另一方面通过系统的测试和试运行，根据用户的反馈之后完成了系统的更新和完善，使系统具有相当的易操作性和使用性，基本实现了本课题的研究初期设立的目标，解决了随着系统复杂度不断增加而带来的开发、部署、线上运维等系统化问题。

## 展望

虽然已经完成了课题相关的研究和系统的设计实现工作，但是受限于技术及制度等因素，课题和系统的发展仍存在一些问题仍未得到很好地解决。

1. Dubbo框架只是实现了服务治理，其他组件需要另外整合以实现对应的功能，比如：分布式配置可以使用淘宝的diamond、百度的disconf来实现分布式配置管理。服务跟踪：可以使用京东开源的Hydra；批量任务，可以使用当当开源的Elastic-Job。
2. Dubbo的RPC来实现服务间调用的一些痛点：
   1. 服务提供方与调用方接口依赖方式太强：调用方对提供方的抽象接口存在强依赖关系，需要严格的管理版本依赖，才不会出现服务方与调用方的不一致导致应用无法编译成功等一系列问题；
   2. 服务对平台敏感，难以简单复用：通常我们在提供对外服务时，都会以REST的方式提供出去，这样可以实现跨平台的特点。在Dubbo中我们要提供REST接口时，不得不实现一层代理，用来将RPC接口转换成REST接口进行对外发布。所以当当网在dubbox（基于Dubbo的开源扩展）中增加了对REST支持。
3. 使用Docker容器来部署微服务可以更加发挥微服务架构的优势。由于微服务常常成百上千个，虽然不可能一个服务一台机器，但是数十个服务部署在一台机器上回相互影响。并且，部署修改配置的工作量巨大。Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。有了Docker，就可以很好的对微服务进行隔离，也有助于自动化部署。

# 致 谢

时光飞逝，三年多的研究生生活即将画上一个圆满的句号，我将要结束江南大学三年的在职研究生生活，借这次论文即将完成之际，我向期间对我作出悉心指导的老师们，同学，和朋友致以最真诚的感谢。

首先要感谢我的导师李志华老师，他不但在学习，科研上给予我很大的帮助，还在论文写作，规划，提炼论题方面给予我很大的鼓励，本文从开题，撰写，修改到定稿，都倾注了老师大量的心血，李志华严谨的作风，扎实的教学学风都令我受益匪浅，你的言传身教将会成为学习的榜样，书山有路勤为径，学海无涯苦作舟，至此再次向李老师表达崇高的敬意与真挚的感谢。感谢所有我的授课老师们，让我学到了很多在工作中学不到的理论知识，对我的工作帮助巨大。感谢我的同学，在这期间，我们一起学习生活，一起奋斗，让我再次体验到了大学生涯的感觉。

最后，衷心感谢导师在百忙之中评阅论文与参与答辩论文，再次表示感谢。

# 参考文献

[1]韩秋君;丁岳伟,SaaS模式下新型认证方案的设计与分析[J],计算机工程,2011-07

[2]吴昌雨;李云松;刘青;王善勤,基于微服务架构的物联网应用基础框架设计[J];宿州学院学报,2015-07

[3]郭栋;王伟;曾国荪,一种基于微服务架构的新型云件PaaS平台[J],信息网络安全,2015-11

[4]纽曼,微服务设计,人民邮电出版社,2016-04

[5]翟永超,Spring Cloud微服务实战,电子工业出版社,2017-04

[6]王磊,微服务架构与实践,电子工业出版社,2015-11

[7]托马斯·埃尔, SOA架构：服务和微服务分析及设计,机械工业出版社,2017-12

[8]郝昕悦，谈谈高并发系统的限流，2017-05

# 附录



## 数据库设计

下表列出了车联网平台最基础的基础信息表：

### 车辆基础信息表(TB\_VEHICLE)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **代码** | **名称** | **数据类型** | **主要的** | **强制** | **注释** |
| VEHICLE\_ID | VEHICLE\_ID | VARCHAR2(36 CHAR) | TRUE | TRUE |  |
| ORG\_ID | ORG\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 所属组织 |
| BUYER\_ENT\_ID | BUYER\_ENT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 购买企业 |
| BUYER\_ACCOUNT\_ID | BUYER\_ACCOUNT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 购买用户 |
| VEHICLE\_VIN | VEHICLE\_VIN | VARCHAR2(20 CHAR) | FALSE | TRUE | 车辆VIN号 |
| VEHICLE\_SN | VEHICLE\_SN | VARCHAR2(50 CHAR) | FALSE | FALSE | 车工号 |
| VEHICLE\_LN | VEHICLE\_LN | VARCHAR2(50 CHAR) | FALSE | FALSE | 车牌号 |
| VEHICLE\_PLATE\_COLOR | VEHICLE\_PLATE\_COLOR | VARCHAR2(50 CHAR) | FALSE | FALSE | 车牌颜色 |
| VEHICLE\_COLOR | VEHICLE\_COLOR | VARCHAR2(50 CHAR) | FALSE | FALSE | 车身颜色 |
| VEHICLE\_NO | VEHICLE\_NO | NUMBER(15) | FALSE | TRUE | 自增长 |
| REG\_STATE | REG\_STATE | NUMBER(1) | FALSE | TRUE | 注册状态 0 未注册 1已注册 |
| REG\_DATE | REG\_DATE | DATE | FALSE | FALSE | 注册日期 |
| ENGINE\_NO | ENGINE\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 发动机号 |
| PRICE | PRICE | NUMBER(15,2) | FALSE | FALSE | 售价 |
| MAINTANCE\_STATE | MAINTANCE\_STATE | VARCHAR2(50 CHAR) | FALSE | FALSE | 二级维护状态 |
| REMARK | REMARK | VARCHAR2(500 CHAR) | FALSE | FALSE | 备注 |
| VEHICLE\_LEVEL | VEHICLE\_LEVEL | VARCHAR2(50 CHAR) | FALSE | FALSE | 等级 |
| AREA\_ID | AREA\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 区域 |
| REVIEW\_STATE | REVIEW\_STATE | NUMBER(1) | FALSE | FALSE | 审核状态 0 未审核 1已审核 |
| INSURANCE\_STATE | INSURANCE\_STATE | NUMBER(1) | FALSE | FALSE | 投保状态 0 未投保 1已投保 |
| VOLTAGE | VOLTAGE | VARCHAR2(50 CHAR) | FALSE | FALSE | 电压 |
| SERVICE\_NO | SERVICE\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 运营证号 |
| PURCHASE\_DATE | PURCHASE\_DATE | DATE | FALSE | FALSE | 购买日期 |
| PURCHASE\_NO | PURCHASE\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 购置证号 |
| VEHICLE\_MODEL\_ID | VEHICLE\_MODEL\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 车辆型号ID |
| VEHICLE\_CODE | VEHICLE\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆编号，内部编码，保证企业内部唯一 |
| VEHICLE\_TYPE | VEHICLE\_TYPE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆类型，JT415标准中的5.4.9章节，参照运输行业车型编码表 |
| VEHICLE\_BUSI\_TYPE | VEHICLE\_BUSI\_TYPE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆业务分类 |
| VEHICLE\_SERVICE\_BRAND | VEHICLE\_SERVICE\_BRAND | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆服务品牌 |
| VEHICLE\_INSPECTION\_DATE | VEHICLE\_INSPECTION\_DATE | DATE | FALSE | FALSE | 车辆年检日期 |
| SOURCE\_CODE | SOURCE\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆来源 |
| VEHICLE\_STATE | VEHICLE\_STATE | NUMBER(3) | FALSE | FALSE | -1 未知 10 制造中 20 配送中 30 运营中 40 故障中 50 空闲中 |
| TERMINAL\_NETWORK\_STATE | TERMINAL\_NETWORK\_STATE | NUMBER(1) | FALSE | TRUE | 终端网络状态 -1未知 0 不在线 1在线 |
| TERMINAL\_NETWORK\_STATE\_TIME | TERMINAL\_NETWORK\_STATE\_TIME | DATE | FALSE | FALSE | 终端网络状态更新时间 |
| FAULT\_STATE | FAULT\_STATE | NUMBER(1) | FALSE | FALSE | 故障状态 |
| FAULT\_STATE\_TIME | FAULT\_STATE\_TIME | DATE | FALSE | FALSE | 故障状态时间 |
| VS\_REF\_ID | VS\_REF\_ID | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆状态参考值id，对应车辆状态参考值信息表（TB\_VSTATUS\_REF）中的AUTO\_ID |
| VEHICLE\_OPERATION\_STATE | VEHICLE\_OPERATION\_STATE | VARCHAR2(50 CHAR) | FALSE | FALSE | 运营状态 |
| OPERATOR\_ID | OPERATOR\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 运营机构ID |
| TRANSTYPE\_CODE | TRANSTYPE\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 运输行业类别编码，JT415标准中的5.2.1章节 |
| CERTIFICATE\_STATE | CERTIFICATE\_STATE | NUMBER(1) | FALSE | FALSE | 运输行业证照状态，JT415标准中的5.5.2章节,1有效，0无效 |
| CERTIFICATE\_TYPE | CERTIFICATE\_TYPE | VARCHAR2(50 CHAR) | FALSE | FALSE | 运输行业证照类别，JT415标准中的5.5.1章节 |
| ROAD\_TRANSPORT | ROAD\_TRANSPORT | VARCHAR2(50 CHAR) | FALSE | FALSE | 道路运输证号 |
| FEE\_STATE | FEE\_STATE | NUMBER(1) | FALSE | FALSE | 缴费状态 1:未欠费 0：欠费 |
| OIL\_CONFIG\_STATE | OIL\_CONFIG\_STATE | NUMBER(1) | FALSE | FALSE | 是否油耗配置：0：否 1：是 |
| VIDEO\_NO | VIDEO\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 视频连接编号 |
| TRANSPORT\_CERT | TRANSPORT\_CERT | VARCHAR2(50 CHAR) | FALSE | FALSE | 道路运输证 |
| PURCHASE\_TAX | PURCHASE\_TAX | VARCHAR2(50 CHAR) | FALSE | FALSE | 购置税（附加费） |
| VEHICLE\_USE\_TYPE | VEHICLE\_USE\_TYPE | NUMBER(2) | FALSE | FALSE | 车辆用途  01 客运  02 公交  03 旅游  04 团体  05 校车 |
| SELL\_ORDER\_NO | SELL\_ORDER\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 销售订单号 |
| MADE\_ORDER\_NO | MADE\_ORDER\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 生产订单号 |
| CHASSIS\_ORDER\_NO | CHASSIS\_ORDER\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 底盘订单号 |
| SELL\_DATE | SELL\_DATE | DATE | FALSE | FALSE | 销售日期 |
| TERMINAL\_SETUP\_TYPE | TERMINAL\_SETUP\_TYPE | NUMBER(1) | FALSE | FALSE | 终端安装 方式 1 前装 2后装 |
| PICTURE\_FILE\_ID | PICTURE\_FILE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 照片 |
| SERVER\_TOTAL\_MILEAGE | SERVER\_TOTAL\_MILEAGE | NUMBER(15,2) | FALSE | FALSE | 服务端总里程 |
| SERVER\_TOTAL\_MILEAGE\_TIME | SERVER\_TOTAL\_MILEAGE\_TIME | DATE | FALSE | FALSE | 服务端总里程更新时间 |
| CURRENT\_DRIVER\_ID | CURRENT\_DRIVER\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| CURRENT\_CONDUCTOR\_ID | CURRENT\_CONDUCTOR\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| CURRENT\_ROUTE\_ID | CURRENT\_ROUTE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| CURRENT\_TRIP\_ID | CURRENT\_TRIP\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| SERVICE\_END\_DATE | SERVICE\_END\_DATE | DATE | FALSE | FALSE | 服务结束时间 |
| PRODUCTION\_DATE | PRODUCTION\_DATE | DATE | FALSE | FALSE | 出厂时间 |
| LOCK\_STATE | LOCK\_STATE | NUMBER(1) | FALSE | FALSE | 锁车状态 见VehicleLockState枚举 |
| CURRENT\_LIMIT\_SPD\_THRESHOLD | CURRENT\_LIMIT\_SPD\_THRESHOLD | NUMBER(8,2) | FALSE | FALSE | 当前限速阈值 |
| MGMT\_STATE | MGMT\_STATE | NUMBER(1) | FALSE | FALSE | 管理状态 1：正常 2：吊销 |
| DVR3G\_STATE | DVR3G\_STATE | NUMBER(1) | FALSE | FALSE |  |
| DVR3G\_STATE\_TIME | DVR3G\_STATE\_TIME | DATE | FALSE | FALSE |  |
| VEHICLE\_INDUSTRY\_ID | VEHICLE\_INDUSTRY\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 行业 |
| BASIC\_INFO\_UPDATE\_TIME | BASIC\_INFO\_UPDATE\_TIME | DATE | FALSE | FALSE | 基础信息更新时间 |
| ENGINE\_TYPE | ENGINE\_TYPE | NUMBER(3) | FALSE | FALSE | 1 潍柴 2 玉柴 ESI 3 玉柴 ECI |
| APP\_INST\_ID | APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| CONS\_SOURCE | CONS\_SOURCE | NUMBER(1) | FALSE | TRUE | 油/（气）耗来源，1-终端上报；2-油箱油量；3-人工录入 |
| CURB\_WEIGHT | CURB\_WEIGHT | NUMBER(5) | FALSE | FALSE | 整备质量,0~50000(包含0和50000) |
| LOADED\_WEIGHT | LOADED\_WEIGHT | NUMBER(5) | FALSE | FALSE | 满载总质量（吨），0~50000（包含0和50000） |
| LIMIT\_PASSENGER\_COUNT | LIMIT\_PASSENGER\_COUNT | NUMBER(3) | FALSE | FALSE | 最大限员，核载人数，0~500（包含0和500） |
| WHEEL\_BASE | WHEEL\_BASE | NUMBER(5) | FALSE | FALSE | 轴距（mm），只可输入0~50000之间的数字，保留整数 |
| LICENSE\_DATE | LICENSE\_DATE | DATE | FALSE | FALSE | 上牌日期 |
| FUELTANK\_VOLUME | FUELTANK\_VOLUME | NUMBER(7,2) | FALSE | FALSE | 油箱油量，0~5000之间的数字，保留2位小数 |
| ENGINE\_MODEL | ENGINE\_MODEL | VARCHAR2(36 CHAR) | FALSE | TRUE | 发动机型号 |
| ENGINE\_STANDARD\_ROTATE | ENGINE\_STANDARD\_ROTATE | NUMBER(5) | FALSE | FALSE | 发动机标准转速，默认为1500 |
| ENGINE\_RATED\_POWER | ENGINE\_RATED\_POWER | NUMBER(3) | FALSE | FALSE | 发动机额定功率，0~700（包含0和700） |
| ANNUAL\_INSPECTION\_STATE | ANNUAL\_INSPECTION\_STATE | NUMBER(1) | FALSE | FALSE | 年度审验状态，0-未年审；1-年审合格；2-年审不合格 |
| TIRE\_ROLLING\_RADIUS | TIRE\_ROLLING\_RADIUS | NUMBER(15,2) | FALSE | FALSE | 轮胎滚动半径（米）,小数，长度10，不能为0 |
| DRIVING\_LICENSE\_NO | DRIVING\_LICENSE\_NO | VARCHAR2(30 CHAR) | FALSE | FALSE | 行驶证号,长度不超过30位 |
| BUSI\_SCOPE | BUSI\_SCOPE | VARCHAR2(50 CHAR) | FALSE | FALSE | 经营范围,道路运输电子政务平台 证照分类5.2.4；长度不超过50位 |
| ROW\_CREATER | ROW\_CREATER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建人 |
| ROW\_CREATE\_TIME | ROW\_CREATE\_TIME | DATE | FALSE | TRUE | 行创建时间 |
| ROW\_MODIFIER | ROW\_MODIFIER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改人 |
| ROW\_MODIFY\_TIME | ROW\_MODIFY\_TIME | DATE | FALSE | TRUE | 行最后修改时间 |
| ROW\_STATE | ROW\_STATE | NUMBER(1) | FALSE | TRUE | 行状态 |
| ROW\_CREATE\_APP\_INST\_ID | ROW\_CREATE\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建应用ID |
| ROW\_MODIFY\_APP\_INST\_ID | ROW\_MODIFY\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改应用ID |
| BATTERY\_OEM\_ID | BATTERY\_OEM\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 电池OEM ID |
| BATTERY\_TYPE | BATTERY\_TYPE | VARCHAR2(50 CHAR) | FALSE | FALSE | 电池类型 |
| MOVEMENT\_STATE | MOVEMENT\_STATE | NUMBER(1) | FALSE | FALSE | 1 行驶 0 停驶 |
| SEAT\_COUNT | SEAT\_COUNT | NUMBER(3) | FALSE | FALSE | 座位数 |
| MILEAGE\_SOURCE | MILEAGE\_SOURCE | NUMBER(1) | FALSE | FALSE | 里程来源 1：终端上报 2：手工录入 |
| ELEC\_CONS\_SOURCE | ELEC\_CONS\_SOURCE | NUMBER(1) | FALSE | FALSE | 电耗来源 1：终端上报 2：手工录入 |
| V\_FAULT\_STATE | V\_FAULT\_STATE | NUMBER(2) | FALSE | FALSE | 车辆故障状态 1：正常 2：故障中 3：维修中 |
| V\_FAULT\_STATE\_TIME | V\_FAULT\_STATE\_TIME | DATE | FALSE | FALSE | 车辆故障状态时间 |
| V\_FAULT\_LST\_NOR\_TIME | V\_FAULT\_LST\_NOR\_TIME | DATE | FALSE | FALSE | 车辆故障最后一次恢复正常的时间 |

### 车辆型号型表(TB\_VEHICLE\_MODEL)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **代码** | **名称** | **数据类型** | **主要的** | **强制** | **注释** |
| VEHICLE\_MODEL\_ID | VEHICLE\_MODEL\_ID | VARCHAR2(36 CHAR) | TRUE | TRUE | 车辆型号ID |
| VEHICLE\_MODEL\_CODE | VEHICLE\_MODEL\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆型号编码 |
| VEHICLE\_MODEL\_NAME | VEHICLE\_MODEL\_NAME | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆型号名称 |
| VEHICLE\_MODEL\_DESC | VEHICLE\_MODEL\_DESC | VARCHAR2(100 CHAR) | FALSE | FALSE | 车辆型号描述 |
| OEM\_ID | OEM\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | OEM厂家ID |
| SEAT\_COUNT | SEAT\_COUNT | NUMBER(3) | FALSE | FALSE | 座位数 |
| LIMIT\_PASSENGER\_COUNT | LIMIT\_PASSENGER\_COUNT | NUMBER(3) | FALSE | FALSE | 最大限员 |
| POWER\_TYPE | POWER\_TYPE | NUMBER(2) | FALSE | FALSE | 能源类型 |
| POWER\_ID\_FIRST | POWER\_ID\_FIRST | VARCHAR2(36 CHAR) | FALSE | FALSE | 第一能源类型 关联 TB\_VEHICLE\_POWER |
| POWER\_FIRST\_VOL | POWER\_FIRST\_VOL | NUMBER(15,2) | FALSE | FALSE | 第一能源容量 |
| POWER\_FIRST\_100KM\_CONS | POWER\_FIRST\_100KM\_CONS | NUMBER(15,2) | FALSE | FALSE | 第一能源容量百公里能耗 |
| POWER\_ID\_SECOND | POWER\_ID\_SECOND | VARCHAR2(36 CHAR) | FALSE | FALSE | 第二能源类型 关联 TB\_VEHICLE\_POWER |
| POWER\_SECOND\_VOL | POWER\_SECOND\_VOL | NUMBER(15,2) | FALSE | FALSE | 第二能源容量 |
| POWER\_SECOND\_100KM\_CONS | POWER\_SECOND\_100KM\_CONS | NUMBER(15,2) | FALSE | FALSE | 第二能源容量百公里能耗 |
| POWER\_ID\_THIRD | POWER\_ID\_THIRD | VARCHAR2(36 CHAR) | FALSE | FALSE | 第三能源类型 关联 TB\_VEHICLE\_POWER |
| POWER\_THIRD\_VOL | POWER\_THIRD\_VOL | NUMBER(15,2) | FALSE | FALSE | 第三能源容量 |
| POWER\_THIRD\_100KM\_CONS | POWER\_THIRD\_100KM\_CONS | NUMBER(15,2) | FALSE | FALSE | 第三能源容量百公里能耗 |
| POWER\_SOURCE | POWER\_SOURCE | NUMBER(3) | FALSE | FALSE | 能源来源 |
| POWER\_WITHOUT\_AC\_100KM\_CONS | POWER\_WITHOUT\_AC\_100KM\_CONS | NUMBER(15,2) | FALSE | FALSE | 不开空调百公里能耗 |
| POWER\_WITH\_AC\_100KM\_CONS | POWER\_WITH\_AC\_100KM\_CONS | NUMBER(15,2) | FALSE | FALSE | 开空调百公里能耗 |
| FRONT\_BRAKE\_TYPE | FRONT\_BRAKE\_TYPE | NUMBER(3) | FALSE | FALSE | 前制动类型 |
| BACK\_BRAKE\_TYPE | BACK\_BRAKE\_TYPE | NUMBER(3) | FALSE | FALSE | 后制动类型 |
| GEAR\_RATIO | GEAR\_RATIO | VARCHAR2(50 CHAR) | FALSE | FALSE | 传动比 |
| SCALING\_FACTOR | SCALING\_FACTOR | NUMBER(10,2) | FALSE | FALSE | 折标系数 |
| EMISSION\_STANDARD | EMISSION\_STANDARD | NUMBER(3) | FALSE | FALSE | 排放标准 |
| VEHICLE\_LENGTH | VEHICLE\_LENGTH | NUMBER(10,2) | FALSE | FALSE | 车长 |
| VEHICLE\_WIDTH | VEHICLE\_WIDTH | NUMBER(10,2) | FALSE | FALSE | 车宽 |
| VEHICLE\_HEIGTH | VEHICLE\_HEIGTH | NUMBER(10,2) | FALSE | FALSE | 车高 |
| VEHICLE\_WEIGHT | VEHICLE\_WEIGHT | NUMBER(10,2) | FALSE | FALSE | 车重 |
| FULL\_LOAD\_WEIGHT | FULL\_LOAD\_WEIGHT | NUMBER(10,2) | FALSE | FALSE | 满载总质量 |
| COOLING\_TYPE | COOLING\_TYPE | NUMBER(3) | FALSE | FALSE | 冷却方式 |
| TYRE\_SIZE | TYRE\_SIZE | VARCHAR2(50 CHAR) | FALSE | FALSE | 轮胎规格 |
| PICTURE\_FILE\_ID | PICTURE\_FILE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 图片ID |
| BRAND\_ID | BRAND\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 品牌ID |
| CURB\_WEIGHT | CURB\_WEIGHT | NUMBER(10,2) | FALSE | FALSE | 整备质量 |
| WHEEL\_BASE | WHEEL\_BASE | NUMBER(10,2) | FALSE | FALSE | 轴距 |
| MAX\_POWER | MAX\_POWER | NUMBER(10,2) | FALSE | FALSE | 最大功率 |
| MAX\_TORQUE | MAX\_TORQUE | NUMBER(10,2) | FALSE | FALSE | 最大扭矩 |
| MAX\_CLIMB\_SLOPE | MAX\_CLIMB\_SLOPE | NUMBER(10,2) | FALSE | FALSE | 最大爬坡度 |
| MIN\_TURNING\_DIAMETER | MIN\_TURNING\_DIAMETER | NUMBER(10,2) | FALSE | FALSE | 最小转弯直径 |
| SYSTEM\_TYPE\_NAME | SYSTEM\_TYPE\_NAME | VARCHAR2(50 CHAR) | FALSE | FALSE | 系统类型名称 |
| ENGINE\_TYPE | ENGINE\_TYPE | NUMBER(3) | FALSE | FALSE | 1 潍柴 2 玉柴 ESI 3 玉柴 ECI |
| EXT\_INFO | EXT\_INFO | VARCHAR2(1000 CHAR) | FALSE | FALSE | 扩展信息 |
| ROW\_CREATER | ROW\_CREATER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建人 |
| ROW\_CREATE\_TIME | ROW\_CREATE\_TIME | DATE | FALSE | TRUE | 行创建时间 |
| ROW\_MODIFIER | ROW\_MODIFIER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改人 |
| ROW\_MODIFY\_TIME | ROW\_MODIFY\_TIME | DATE | FALSE | TRUE | 行最后修改时间 |
| ROW\_STATE | ROW\_STATE | NUMBER(1) | FALSE | TRUE | 行状态 |
| ROW\_CREATE\_APP\_INST\_ID | ROW\_CREATE\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建应用ID |
| ROW\_MODIFY\_APP\_INST\_ID | ROW\_MODIFY\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改应用ID |

### 组织信息表(TB\_ORGANIZATION)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **代码** | **名称** | **数据类型** | **主要的** | **强制** | **注释** |
| ORG\_ID | ORG\_ID | VARCHAR2(36 CHAR) | TRUE | TRUE |  |
| FULL\_NAME | FULL\_NAME | VARCHAR2(100 CHAR) | FALSE | FALSE | 组织全名 |
| ORG\_DESC | ORG\_DESC | VARCHAR2(500 CHAR) | FALSE | FALSE | 组织描述 |
| SHORT\_NAME | SHORT\_NAME | VARCHAR2(100 CHAR) | FALSE | FALSE | 组织短名 |
| ENG\_NAME | ENG\_NAME | VARCHAR2(100 CHAR) | FALSE | FALSE | 英文名 |
| PARENT\_ID | PARENT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 上一级组织ID |
| BELONG\_ENT\_ID | BELONG\_ENT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 归属的企业ID |
| ORG\_PATH | ORG\_PATH | VARCHAR2(500 CHAR) | FALSE | TRUE | 区域路径 |
| ORG\_CODE | ORG\_CODE | VARCHAR2(100 CHAR) | FALSE | FALSE | 组织编号 |
| ORG\_TYPE | ORG\_TYPE | NUMBER(2) | FALSE | FALSE | 组织类型 |
| ORG\_TAG | ORG\_TAG | VARCHAR2(200 CHAR) | FALSE | FALSE | 组织标签 |
| ORG\_DEPTH | ORG\_DEPTH | NUMBER(2) | FALSE | FALSE | 组织层次 |
| ORG\_INNER\_NAME | ORG\_INNER\_NAME | VARCHAR2(50 CHAR) | FALSE | FALSE | 组织内部名称 |
| ORG\_CHARACTER | ORG\_CHARACTER | NUMBER(2) | FALSE | FALSE | 组织性质 |
| CONTACT\_PERSON | CONTACT\_PERSON | VARCHAR2(50 CHAR) | FALSE | FALSE | 联系人 |
| CONTACT\_PHONE | CONTACT\_PHONE | VARCHAR2(50 CHAR) | FALSE | FALSE | 联系电话 |
| CONTACT\_ADDR | CONTACT\_ADDR | VARCHAR2(500 CHAR) | FALSE | FALSE | 联系地址 |
| ORG\_LEADER | ORG\_LEADER | VARCHAR2(50 CHAR) | FALSE | FALSE | 组织负责人 |
| ORG\_LEADER\_ACCOUNT\_ID | ORG\_LEADER\_ACCOUNT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 组织负责人账号 |
| FAULT\_RATE\_WEEKLY | FAULT\_RATE\_WEEKLY | NUMBER(8,2) | FALSE | FALSE | 周故障率 |
| POWER\_SAVE\_RATE\_WEEKLY | POWER\_SAVE\_RATE\_WEEKLY | NUMBER(8,2) | FALSE | FALSE | 周节能率 |
| ORG\_ROLE | ORG\_ROLE | NUMBER(3) | FALSE | FALSE | 组织角色 |
| ORG\_INDUSTRY | ORG\_INDUSTRY | NUMBER(3) | FALSE | FALSE | 所属行业 |
| CLIENT\_LEVEL\_ID | CLIENT\_LEVEL\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 客户级别ID |
| REG\_TIME | REG\_TIME | DATE | FALSE | FALSE | 入网时间 |
| TEL1 | TEL1 | VARCHAR2(100 CHAR) | FALSE | FALSE | 电话1 |
| TEL2 | TEL2 | VARCHAR2(100 CHAR) | FALSE | FALSE | 电话2 |
| TEL3 | TEL3 | VARCHAR2(100 CHAR) | FALSE | FALSE | 电话3 |
| EMAIL1 | EMAIL1 | VARCHAR2(200 CHAR) | FALSE | FALSE | 邮件1 |
| EMAIL2 | EMAIL2 | VARCHAR2(200 CHAR) | FALSE | FALSE | 邮件2 |
| ENT\_MODE | ENT\_MODE | NUMBER(1) | FALSE | FALSE | 模式一，模式二，模式三 |
| FAX | FAX | VARCHAR2(50 CHAR) | FALSE | FALSE | 传真 |
| POSTCODE | POSTCODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 邮编 |
| BUSINESS\_SCOPE | BUSINESS\_SCOPE | VARCHAR2(500 CHAR) | FALSE | FALSE | 经营范围 |
| AREA\_ID | AREA\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 区域 |
| AUDIT\_STATE | AUDIT\_STATE | NUMBER(2) | FALSE | FALSE | 企业状态：待审核，已经审核 |
| LOGO\_FILE\_ID | LOGO\_FILE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | LOGO |
| WEBSITE | WEBSITE | VARCHAR2(200 CHAR) | FALSE | FALSE | 网站 |
| TOTAL\_MILEAGE | TOTAL\_MILEAGE | NUMBER(15,2) | FALSE | FALSE | 总里程 |
| TOTAL\_POWER\_CONS | TOTAL\_POWER\_CONS | NUMBER(15,2) | FALSE | FALSE | 总能耗 |
| AVG\_MILEAGE\_DAILY | AVG\_MILEAGE\_DAILY | NUMBER(15,2) | FALSE | FALSE | 日平均里程 |
| SERVICE\_END\_TIME | SERVICE\_END\_TIME | DATE | FALSE | FALSE | 最近服务到期时间 |
| VEHICLE\_COUNT | VEHICLE\_COUNT | NUMBER(8) | FALSE | FALSE | 注册车辆数：每日统计一次 |
| REMARK | REMARK | VARCHAR2(500 CHAR) | FALSE | FALSE | 备注 |
| APP\_INST\_ID | APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| ROW\_CREATER | ROW\_CREATER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建人 |
| ROW\_CREATE\_TIME | ROW\_CREATE\_TIME | DATE | FALSE | TRUE | 行创建时间 |
| ROW\_MODIFIER | ROW\_MODIFIER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改人 |
| ROW\_MODIFY\_TIME | ROW\_MODIFY\_TIME | DATE | FALSE | TRUE | 行最后修改时间 |
| ROW\_STATE | ROW\_STATE | NUMBER(1) | FALSE | TRUE | 行状态 |
| ROW\_CREATE\_APP\_INST\_ID | ROW\_CREATE\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建应用ID |
| ROW\_MODIFY\_APP\_INST\_ID | ROW\_MODIFY\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改应用ID |
| ENT\_BUSI\_LICENSE | ENT\_BUSI\_LICENSE | VARCHAR2(50 CHAR) | FALSE | FALSE | 经营许可证号 |
| IS\_CAL\_RT | IS\_CAL\_RT | NUMBER(1) | FALSE | FALSE | 是否计算完好率，0：否，1：是 |
| IS\_CAL\_ENERGY | IS\_CAL\_ENERGY | NUMBER(1) | FALSE | FALSE | 是否考核能耗，0：否，1：是 |
| ENTERPRISE\_TYPE\_CFG | ENTERPRISE\_TYPE\_CFG | NUMBER(3) | FALSE | FALSE | 企业类型 A类 B类 C类 等等 |

### 车辆品牌表(TB\_VEHICLE\_BRAND)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **代码** | **名称** | **数据类型** | **主要的** | **强制** | **注释** |
| VEHICLE\_BRAND\_ID | VEHICLE\_BRAND\_ID | VARCHAR2(36 CHAR) | TRUE | TRUE | 车辆品牌ID |
| VEHICLE\_BRAND\_CODE | VEHICLE\_BRAND\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆品牌编码 |
| VEHICLE\_BRAND\_NAME | VEHICLE\_BRAND\_NAME | VARCHAR2(50 CHAR) | FALSE | FALSE | 车辆品牌名称 |
| VEHICLE\_BRAND\_DESC | VEHICLE\_BRAND\_DESC | VARCHAR2(100 CHAR) | FALSE | FALSE | 车辆品牌描述 |
| OEM\_ID | OEM\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | OEM厂家ID |
| PICTURE\_FILE\_ID | PICTURE\_FILE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 图片ID |
| REMARK | REMARK | VARCHAR2(200 CHAR) | FALSE | FALSE | 备注 |
| ROW\_CREATER | ROW\_CREATER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建人 |
| ROW\_CREATE\_TIME | ROW\_CREATE\_TIME | DATE | FALSE | TRUE | 行创建时间 |
| ROW\_MODIFIER | ROW\_MODIFIER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改人 |
| ROW\_MODIFY\_TIME | ROW\_MODIFY\_TIME | DATE | FALSE | TRUE | 行最后修改时间 |
| ROW\_STATE | ROW\_STATE | NUMBER(1) | FALSE | TRUE | 行状态 |
| ROW\_CREATE\_APP\_INST\_ID | ROW\_CREATE\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建应用ID |
| ROW\_MODIFY\_APP\_INST\_ID | ROW\_MODIFY\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改应用ID |

### 车辆轨迹信息表(TB\_VEHICLE\_TRACK\_RECORD)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **代码** | **名称** | **数据类型** | **主要的** | **强制** | **注释** |
| ID | ID | NUMBER(20) | FALSE | TRUE | 自增长ID |
| VEHICLE\_ID | VEHICLE\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| VEHICLE\_VIN | VEHICLE\_VIN | VARCHAR2(20 CHAR) | FALSE | TRUE |  |
| TERMINAL\_ID | TERMINAL\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 终端ID |
| SIMCARD\_NO | SIMCARD\_NO | VARCHAR2(36 CHAR) | FALSE | FALSE | SIM卡号 |
| GPS\_VALID | GPS\_VALID | NUMBER(1) | FALSE | FALSE | gps是否有效 |
| GPS\_LATITUDE | GPS\_LATITUDE | NUMBER(15,8) | FALSE | FALSE | 经度 |
| GPS\_LONGITUDE | GPS\_LONGITUDE | NUMBER(15,8) | FALSE | FALSE | 纬度 |
| GPS\_ALTITUDE | GPS\_ALTITUDE | NUMBER(15,8) | FALSE | FALSE | 海拔 |
| GPS\_DIRECTION | GPS\_DIRECTION | NUMBER(15,2) | FALSE | FALSE | 方向 |
| GPS\_SPEED | GPS\_SPEED | NUMBER(15,2) | FALSE | FALSE | GPS速度 |
| GPS\_STATE | GPS\_STATE | NUMBER(1) | FALSE | FALSE | GPS状态 |
| HAS\_EVENT | HAS\_EVENT | NUMBER(1) | FALSE | FALSE | 是否有事件 |
| EVENT\_CODE\_LIST | EVENT\_CODE\_LIST | VARCHAR2(1000 CHAR) | FALSE | FALSE | 事件列表 |
| VEHICLE\_SPEED | VEHICLE\_SPEED | NUMBER(15,2) | FALSE | FALSE | 车辆速度 |
| DRIVER\_ID | DRIVER\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 驾驶员ID |
| DRIVER\_NAME | DRIVER\_NAME | VARCHAR2(50 CHAR) | FALSE | FALSE | 驾驶员姓名 |
| DRIVER\_CARD\_CODE | DRIVER\_CARD\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 驾驶员卡的编号 |
| DRIVER\_LICENSE\_NO | DRIVER\_LICENSE\_NO | VARCHAR2(50 CHAR) | FALSE | FALSE | 驾驶证号码 |
| CONDUCTOR\_CARD\_CODE | CONDUCTOR\_CARD\_CODE | VARCHAR2(50 CHAR) | FALSE | FALSE | 司乘卡的编号 |
| UTC\_TIME | UTC\_TIME | NUMBER(14) | FALSE | FALSE | UTC时间 |
| TERMINAL\_TIME | TERMINAL\_TIME | DATE | FALSE | FALSE | 终端时间 |
| DAY\_KEY | DAY\_KEY | NUMBER(8) | FALSE | FALSE | 终端时间-日期 |
| ALARM\_CODE | ALARM\_CODE | VARCHAR2(64 CHAR) | FALSE | FALSE | 报警编号 |
| DEVICE\_FAULT | DEVICE\_FAULT | VARCHAR2(128 CHAR) | FALSE | FALSE | 设备故障状态，定义见附录设备故障列表。 |
| ALARM\_STATE | ALARM\_STATE | VARCHAR2(256 CHAR) | FALSE | FALSE | 报警状态 |
| ALARM\_STATE\_EXT | ALARM\_STATE\_EXT | VARCHAR2(256 CHAR) | FALSE | FALSE | 报警状态扩展位 |
| VEHICLE\_STATE | VEHICLE\_STATE | VARCHAR2(256 CHAR) | FALSE | FALSE | 车辆信号状态 |
| VEHICLE\_STATE\_EXT | VEHICLE\_STATE\_EXT | VARCHAR2(256 CHAR) | FALSE | FALSE | 车辆信号状态扩展位 |
| ENGINE\_ROTATE\_SPEED | ENGINE\_ROTATE\_SPEED | NUMBER(15, 2) | FALSE | FALSE | 发动机转速 |
| ENGINE\_WATER\_TEMPER | ENGINE\_WATER\_TEMPER | NUMBER(15, 2) | FALSE | FALSE | 发动机水温 |
| ENGINE\_TORQUE | ENGINE\_TORQUE | NUMBER(15, 2) | FALSE | FALSE | 发动机扭矩 |
| ENGINE\_TORQUE\_PERCENT | ENGINE\_TORQUE\_PERCENT | NUMBER(15, 2) | FALSE | FALSE | 发动机转矩百分比 |
| ENGINE\_FIRE\_STATE | ENGINE\_FIRE\_STATE | NUMBER(1) | FALSE | FALSE | 发动机点火状态 |
| ENGINE\_RUN\_DURATION | ENGINE\_RUN\_DURATION | NUMBER(15, 2) | FALSE | FALSE | 发动机总运行时常 |
| ENGINE\_COOLANT\_TEMPER | ENGINE\_COOLANT\_TEMPER | NUMBER(15, 2) | FALSE | FALSE | 发动机冷却液温度 |
| ENGINE\_OIL\_TEMPER | ENGINE\_OIL\_TEMPER | NUMBER(15, 2) | FALSE | FALSE | 机油温度 |
| ACC\_STATE | ACC\_STATE | NUMBER(1) | FALSE | FALSE | ACC状态 |
| PULSE\_SPEED | PULSE\_SPEED | NUMBER(15, 2) | FALSE | FALSE | 脉冲车速 |
| PULSE\_MILEAGE | PULSE\_MILEAGE | NUMBER(15, 2) | FALSE | FALSE | 脉冲里程 |
| TOTAL\_MILEAGE | TOTAL\_MILEAGE | NUMBER(15, 2) | FALSE | FALSE | 总里程 |
| GPS\_TOTAL\_MILEAGE | GPS\_TOTAL\_MILEAGE | NUMBER(15, 2) | FALSE | FALSE | GPS总里程 |
| VSS\_TOTAL\_MILEAGE | VSS\_TOTAL\_MILEAGE | NUMBER(15, 2) | FALSE | FALSE | VSS总里程 |
| SPEED\_SOURCE | SPEED\_SOURCE | NUMBER(1) | FALSE | FALSE | 速度来源 |
| FUEL\_CONSUMPTION\_INSTANT | FUEL\_CONSUMPTION\_INSTANT | NUMBER(15, 2) | FALSE | FALSE | 瞬时油耗 |
| FUEL\_CONSUMPTION\_TOTAL | FUEL\_CONSUMPTION\_TOTAL | NUMBER(15, 2) | FALSE | FALSE | 累计油耗 |
| GAS\_CONSUMPTION\_INSTANT | GAS\_CONSUMPTION\_INSTANT | NUMBER(15, 2) | FALSE | FALSE | 瞬时气耗 |
| GAS\_CONSUMPTION\_TOTAL | GAS\_CONSUMPTION\_TOTAL | NUMBER(15, 2) | FALSE | FALSE | 累计气耗 |
| E\_CONSUMPTION\_INSTANT | E\_CONSUMPTION\_INSTANT | NUMBER(15, 2) | FALSE | FALSE | 瞬时电耗 |
| E\_CONSUMPTION\_TOTAL | E\_CONSUMPTION\_TOTAL | NUMBER(15, 2) | FALSE | FALSE | 累计电耗 |
| FUEL\_CONSUMPTION\_METER | FUEL\_CONSUMPTION\_METER | NUMBER(15, 2) | FALSE | FALSE | 计量仪油耗 |
| FUEL\_PRESSURE | FUEL\_PRESSURE | NUMBER(15, 2) | FALSE | FALSE | 机油压力 |
| FUEL\_VOLUME | FUEL\_VOLUME | NUMBER(15, 2) | FALSE | FALSE | 油量 |
| POWER\_STATE | POWER\_STATE | NUMBER(1) | FALSE | FALSE | 供电状态(0:外部电源 1:电池) |
| POWER\_BATTERY\_VOLTAGE | POWER\_BATTERY\_VOLTAGE | NUMBER(15, 2) | FALSE | FALSE | 供电电池电压 |
| POWER\_EXT\_VOLTAGE | POWER\_EXT\_VOLTAGE | NUMBER(15, 2) | FALSE | FALSE | 供电外部电压 |
| IMG\_HANDLE\_PROCESS | IMG\_HANDLE\_PROCESS | NUMBER(1) | FALSE | FALSE | 图像实时处理过程 |
| AREA\_STATE | AREA\_STATE | NUMBER(1) | FALSE | FALSE | 区域状态 |
| MSG\_ID | MSG\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 短消息ID：有此项，短消息需要应答；无此项，短消息不需要应答 |
| MSG\_RESP\_ID | MSG\_RESP\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE | 短信应答ID |
| MSG\_TYPE | MSG\_TYPE | NUMBER(1) | FALSE | FALSE | 消息类型 1 = 普通消息，2 = 禁用，3 = 提示/告警消息，4 = 打印消息，其他保留； |
| MSG\_INFO | MSG\_INFO | VARCHAR2(300 CHAR) | FALSE | FALSE | 短消息内容 |
| MSG\_SEQ | MSG\_SEQ | VARCHAR2(4 CHAR) | FALSE | FALSE | 终端上行实时序列号 |
| SPEED\_RATIO | SPEED\_RATIO | NUMBER(15, 2) | FALSE | FALSE | 速比 |
| GEAR | GEAR | NUMBER(15, 2) | FALSE | FALSE | 档位 |
| ACC\_PADEL\_POSITION | ACC\_PADEL\_POSITION | NUMBER(15, 2) | FALSE | FALSE | 油门踏板位置 |
| AIR\_IN\_TEMPER | AIR\_IN\_TEMPER | NUMBER(15, 2) | FALSE | FALSE | 进气温度 |
| AIR\_PRESSURE | AIR\_PRESSURE | NUMBER(15, 2) | FALSE | FALSE | 大气压力 |
| OVERLOAD\_FLAG | OVERLOAD\_FLAG | NUMBER(1) | FALSE | FALSE | 超载标识 |
| RSSI2G | RSSI2G | NUMBER(1) | FALSE | FALSE | 2G信号强度 |
| RSSI3G | RSSI3G | NUMBER(1) | FALSE | FALSE | 3G信号强度 |
| RSSI\_WIRELESS | RSSI\_WIRELESS | NUMBER(15, 2) | FALSE | FALSE | 无线通信网络信号强度 |
| PASSENGER\_NUM | PASSENGER\_NUM | NUMBER(15, 2) | FALSE | FALSE | 乘客人数 |
| TRIP\_CODE | TRIP\_CODE | VARCHAR2(36 CHAR) | FALSE | FALSE | 行程编号 |
| DINSTANCE\_PERCENT | DINSTANCE\_PERCENT | NUMBER(15, 2) | FALSE | FALSE | 站间行驶百分比 |
| SITE\_CODE | SITE\_CODE | VARCHAR2(36 CHAR) | FALSE | FALSE | 站点编号 |
| ROUTE\_CODE | ROUTE\_CODE | VARCHAR2(36 CHAR) | FALSE | FALSE | 线路编码 |
| ROUTE\_INFO | ROUTE\_INFO | VARCHAR2(30 CHAR) | FALSE | FALSE | 线路信息 |
| CHAR\_OEFFICIENT\_STATE | CHAR\_OEFFICIENT\_STATE | NUMBER(2) | FALSE | FALSE | 特征系数校正状态，00表示特征系数未校正；01表示特征系数校正中；02表示特征系数已校正 |
| DOOR\_ALARM | DOOR\_ALARM | VARCHAR2(10 CHAR) | FALSE | FALSE | 门状态告警 |
| ANALOG\_VALUE | ANALOG\_VALUE | VARCHAR2(64 CHAR) | FALSE | FALSE | 模拟量 |
| TERMINAL\_SLEEP\_STATE | TERMINAL\_SLEEP\_STATE | VARCHAR2(20 CHAR) | FALSE | FALSE | 终端休眠状态 |
| TERMINAL\_SUB\_MSG\_FLAG | TERMINAL\_SUB\_MSG\_FLAG | VARCHAR2(50 CHAR) | FALSE | FALSE | 终端订阅信息标识 |
| GNSS\_SATELLITE\_COUNT | GNSS\_SATELLITE\_COUNT | NUMBER(2) | FALSE | FALSE | GNSS卫星数量 |
| VEHICLE\_LOCK\_STATE | VEHICLE\_LOCK\_STATE | VARCHAR2(10 CHAR) | FALSE | FALSE | 锁车业务状态 |
| SD\_CARD\_VOLUME | SD\_CARD\_VOLUME | NUMBER(15, 2) | FALSE | FALSE | SD卡容量 |
| SD\_CARD\_VOLUME\_AVAILABLE | SD\_CARD\_VOLUME\_AVAILABLE | NUMBER(15, 2) | FALSE | FALSE | SD卡可用容量 |
| DEVICE\_STATE | DEVICE\_STATE | VARCHAR2(64 CHAR) | FALSE | FALSE | 设备状态 |
| ORG\_ID | ORG\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| ENT\_ID | ENT\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| APP\_INST\_ID | APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | FALSE |  |
| ROW\_CREATER | ROW\_CREATER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建人 |
| ROW\_CREATE\_TIME | ROW\_CREATE\_TIME | DATE | FALSE | TRUE | 行创建时间 |
| ROW\_MODIFIER | ROW\_MODIFIER | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改人 |
| ROW\_MODIFY\_TIME | ROW\_MODIFY\_TIME | DATE | FALSE | TRUE | 行最后修改时间 |
| ROW\_STATE | ROW\_STATE | NUMBER(1) | FALSE | TRUE | 行状态 |
| ROW\_CREATE\_APP\_INST\_ID | ROW\_CREATE\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行创建应用ID |
| ROW\_MODIFY\_APP\_INST\_ID | ROW\_MODIFY\_APP\_INST\_ID | VARCHAR2(36 CHAR) | FALSE | TRUE | 行最后修改应用ID |