

题 目 基于微服务架构的统一分析运行平台的设与实现

软件学院 院（系） 软件工程 专业

学 号 71113333

学生姓名 陈浩远

校内导师 刘奇其

企业导师 薄宏剑

起止日期 2017年1月 ～ 2017年6月

设计地点 华为技术有限公司南京研究所

基于微服务架构的统一分析运行平台的设计与实现

# 摘 要

随着电信行业3G/4G等业务规模的增大，运营商手中的数据量呈现出了爆炸式的增长，数据的规模已经达到了PB级别。海量数据带来了大量的信息，同时也迫使电信行业为了处理这些数据而不得不使用hadoop/spark等分布式大数据处理技术。由于电信运营商本身构造复杂，产品和业务繁多，电信用户多，每天能产生大批的数据，因此产生了很多不同的大数据任务需求，对于如何管理和调度众多的大数据任务也成为了运行商头疼的一件事。

为了实现对众多任务的统一调度和管理，解决运营商在数据分析中面临着的挑战，本文首先对企业软件的发展现状做了调查，结合对统一分析运行平台的需求分析，设计了该平台的总体方案，对系统的各模块进行了详细的设计，并针对企业平台所需的安全性、容错性、高效性等进行了分析，给出了统一运行平台的实现细节，针对统一运行平台所需的特性进行了测试，最后给出了统一运行平台的部署方案。

关键词：微服务；分布式；大数据；docker

A PAAS Platform Based on Micro Service Architecture

# ABSTRACT

With the telecom industry 3G / 4G and other business increases, the amount of data in the hands of operators there has been explosive growth, the size of the data has reached the PB level. Massive data has brought a lot of information, but also forced the telecommunications industry in order to deal with these data had to use hadoop / spark and other distributed large data processing technology. Because telecom operators have their own complex structure, many products and services, many telecom users can produce a lot of data every day, resulting in a lot of different large data task requirements, how to manage and schedule a large number of large data tasks have become runners A headache for one thing.

In order to realize the unified scheduling and management of many tasks and solve the challenges faced by operators in the data analysis, this paper firstly investigates the development of enterprise software, and analyzes the requirements of the unified analysis and operation platform. The overall program, the system of the modules were designed in detail, and for the enterprise platform for the security, fault tolerance, efficiency and other analysis, given the implementation of a unified platform for the details of the platform for the unified operation of the required The characteristics of the test, and finally gives a unified platform for the deployment of the program.

Keywords: Micro service; distributed; big data; docker

目 录

[摘 要 i](#_Toc484477561)

[ABSTRACT ii](#_Toc484477562)

[第一章 绪 论 1](#_Toc484477563)

[1.1 引言 1](#_Toc484477564)

[1.2 运营商大数据现状 1](#_Toc484477565)

[1.3 企业软件架构发展现状 2](#_Toc484477566)

[1.3 本文研究目的和研究内容 2](#_Toc484477567)

[1.4 论文组织结构 3](#_Toc484477568)

[第二章 系统需求分析 4](#_Toc484477569)

[2.1 系统功能需求 4](#_Toc484477570)

[2.1.1 业务编排 4](#_Toc484477571)

[2.1.2 上下文 5](#_Toc484477572)

[2.1.3 任务实例管理 6](#_Toc484477573)

[2.2 系统特性需求 6](#_Toc484477574)

[2.2.1 安全性 6](#_Toc484477575)

[2.2.2 容错性（鲁棒性） 7](#_Toc484477576)

[2.2.3 高效性 7](#_Toc484477577)

[2.2.4 扩展性 7](#_Toc484477578)

[2.2.4 部署难度 7](#_Toc484477579)

[第三章 系统设计 8](#_Toc484477580)

[3.1 系统架构设计 8](#_Toc484477581)

[3.2 服务交互 10](#_Toc484477582)

[3.2.1 RPC 10](#_Toc484477583)

[3.2.2 REST 11](#_Toc484477584)

[3.2.3 RPC和REST的选择 12](#_Toc484477585)

[3.3 微服务模块划分 14](#_Toc484477586)

[3.3.1 任务模板模块 14](#_Toc484477587)

[3.3.2 任务实例模块 17](#_Toc484477588)

[3.3.3 BPMN引擎模块 20](#_Toc484477589)

[3.3.4 Ignite模块 22](#_Toc484477590)

[3.3.5 Eureka模块 24](#_Toc484477591)

[3.4 系统特性的实现 26](#_Toc484477592)

[3.4.1 安全性 26](#_Toc484477593)

[3.4.2 容错性（鲁棒性） 26](#_Toc484477594)

[3.4.3 高效性 26](#_Toc484477595)

[3.4.4 扩展性 26](#_Toc484477596)

[3.4.5 部署难度 27](#_Toc484477597)

[第四章 系统设计 28](#_Toc484477598)

[4.1 具体实现 28](#_Toc484477599)

[致 谢 28](#_Toc484477600)

[参考文献 29](#_Toc484477601)

# 第一章 绪 论

## 1.1 引言

随着上网用户带宽和流量的增加，运营商对用户产生数据分析和处理的需求也越来越大，本文旨在解决电信行业存在的大量的hadoop和spark等业务的调度问题，以分布式的结构，提供一个提供电信分析应用所需的丰富的、高性能的分析引擎，为统一分析开发平台、统一数据治理平台提供统一的运行环境和运维功能。将单独部署的的多种不同的分析引擎，按照运营商自身的业务场景和它们所需的模型进行整合，协助运营商解决以往的运行环境过于简单的问题。

之所以采用微服务，是为了避免传统的分层架构带来的诸多问题，微服务也整合了过去十年来的新概念和技术，因此可以避开许多面向服务的架构中的陷阱。

在多年前，很多组织就开始尝试使用更细粒度的架构来实现更快的交付，结果发现其带来了更好的可扩展性，增强了团度的自治。微服务就是一个这样的快速发展的主题，尽管它不是一个新的想法，但它受了多年来的各种各样的架构的影响，因此，它在当前阶段具有很好的前瞻性和挑战性。

## 1.2 运营商大数据现状

全球信息数据量的迅猛增长成为大数据产业发展的基础。市场调研机构IDC预计，未来全球数据总量年增长率将维持在50%左右，到2020年，全球数据总量将达到40ZB。其中，我国数据量将达到8.6ZB，占全球的21%左右。另外，截至去年12月，我国网民规模达6.88亿人。其中，手机网民规模达6.20亿人，移动互联网流量呈现爆发式增长。工信部统计显示，去年全年移动互联网接入的流量超过400万TB，同比增长了103%。随着全球数据总量以及移动流量的不断提升，运营商的大数据任务不断增加，因此有必要开发一个统一运行平台来对这些任务进行编排和调度。

电信行业迈入数字化精细运营时代，大数据成为使能和引擎核心引擎。大数据既是运营商数字化转型的使能和引擎、同时也是数字化服务的重要内容。在技术上，大数据技术不断地演进和成熟；商业上，数据是运营资产；运营上，大数据作为新思维、新工具可以驱动运营效率的提升和创新；组织上，大数据是组织变革的切入点。

根据Accenture、Gartner趋势分析报告，2014年大数据从技术驱动转变为商业驱动，企业经营优化和数据资产变现成为主要诉求，2017年基于大数据的商业模式创新(Big Data As A Service、数据交易）将会爆发式增长。数据处理技术经过2010-2012年的疯狂发展，将会逐步标准化、云化，成为类似于 RDBMS的通用的IT软件基础设施，但大规模并行计算运维 仍然会成为关键壁垒。

运营商当前在大数据应用时面临如下问题：

* 工具不统一，多厂家异构集成后难以将工具功能集成到端到端解决方案，最大化呈现业务价值。
* 海量电信数据烟囱式分布，难以整合。
* 数据管理分散，难以共享。
* 数据获取过程繁琐，开发困难，对专家依赖高。

具体来说：

* 开发环境：多厂商、多平台造成的多种不同的开发环境，分散了开发资源；关联业务功能/模块被切割，额外增加集成成本；难以形成统一的、相关联的管控机制。
* 运行环境：较多应用运行在不同的环境中，资源利用率不高；跨环境集成降低了运行效率；缺乏统一管理能力。
* 生命周期管理：没有统一的运维管理手段，无法开展多租户管理，面向市场开放能力。
* 调度管理：调度计划难以管理；调度进程难以监控；调度策略难以改进优化。

## 1.3 企业软件架构发展现状

从50-60年代开始，软件技术主要为高级数据/控制结构，具体来说就是函数的重用，为了解决代码的重复出现，这一阶段是编程语言高速发展的阶段。随后70年代，出现了面向结构的架构，这种架构采用自顶向下、逐步求精的设计方法；任何的程序都可以由顺序选择和循环三种控制结构来实现；处理的对象为数值数据。到80年代则出现了面向对象的架构，对象是人们要进行研究的任何事 物，对于同类的事物，可以由一个对象来表示；通过对象抽象，可以大大减轻设 计人员的工作量，简化实际的模型；以数据对象为基础构筑，增加代码可重用性。90年代出现了面向组件的架构，面向组件技术建立在对象技术之上的进一步发展；组件技术的主要目标是粗粒度组件重用；复用性、灵活性大大提高。到04年，SOA已经成为了各大厂商流行的架构风格，SOA引入了服务的概念，它将系统的扩展变得更为灵活，更能响应软件业务的变化，它是基于业务总线ESB的一种架构。而几年来，微服务则成为各大厂商争相关注的对象，相比SOA，微服务不再关注业务总线，它将服务的概念引入了每一个小模块，而且更为依赖网络通信。

## 1.3 本文研究目的和研究内容

本文主要围绕华为的统一分析运行平台的设计和实现，介绍了微服务架构的特性，以及如何通过Spring Cloud实现微服务。该平台解决了运营商对大量任务缺少统一的管理调度系统的问题，根据现有的软件体系结构，解决了企业软件所需的高性能、高容错、高安全性的需求，保障了系统的长期稳定的运行，并且对以后的开发和维护有着较好的支持性。

## 1.4 论文组织结构

本论文主要介绍了基于微服务架构的统一分析运行平台的设计与实现，主要内容组织结构如下：

第一章 绪论，通过对运营商现状和软件结构的发展，介绍了该论文的研究目的和研究内容。

第二章 系统需求分析,分别对功能需求和特性需求进行了详细的分析，明确了统一分析运行平台所要达成的目标。

第三章 系统设计，主要分为架构设计和模块设计，详细的介绍了整个系统的架构和各个模块的架构，以及为了实现特性需求而针对特性需求的设计。

第四章 系统实现与测试，着重介绍了统一分析运行平台的实现，其中包含所使用的各种框架和开发中遇到的一些问题。

第五章 工作总结与展望，对论文内容进行了总结，以及对当前版本的统一分析运行平台的不足提出了一些日后可以实行的优化改进建议。

# 第二章 系统需求分析

## 2.1 系统功能需求

随着电信业务的不断拓展，电信资费越来越低，电信行业产生了越来越多的数据，传统使用人工单独调用大数据任务的方法已经不能满足电信行业所需的大量任务，大量的任务存在与不同的机器上，使得对任务的管理变得越来越困难，大量的机器上运行着早已不再需要的任务，严重浪费了机器的性能和管理人员的精力。为了提高对机器的利用率，降低人力成本，运营商急需一个可以统一管理和调度的平台。

对于统一分析运行平台，需要有业务编排、上下文、按照业务流程进行运行、调用其他服务的功能，在这里我们将多个业务通过编排后组成的业务流称之为任务模板，对于任务模板，统一运行平台要有运行多个任务模板的能力，并且要有对任务模板的管理功能。

### 2.1.1 业务编排

将多个业务按照一定的逻辑和顺序编排在一起。这是统一分析运行平台的关键性功能，他可以将多个业务组合在一起，相比以往需要运维人员逐个执行的业务在这里可以轻松的一次性执行。有了这个功能就可以更好的利用运算平台的性能，并且提供相比人工而言更复杂的管理功能。

要求业务编排需要实现以下的功能：

* 顺序执行：以串行的方式按照编排的顺序逐个执行业务，只有当前一个业务结束才可以开始下一个业务。如图2.1所示，这张图展示由一个人饥饿引发的简单过程。结果是必须有人去购买材料和做饭，之后有人吃了饭，他的饥饿会得到满足。这些业务之间的关系就是顺序执行的关系。

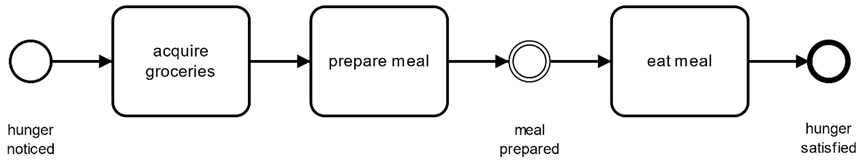


图2.1饥饿过程的顺序执行

* 条件执行：在当前业务后面跟着两个分支，在执行完当前业务之后，判断该处的条件，如果满足条件则接下来进入一个分支，如果不满足条件则进入另一个分支。如图2.2所示，这是一个在线商店从收到订单到用户收到物品之间的一个装货送货的过程，在这中间有很多个条件执行的业务，例如在第一个有效的业务中，决定是要使用普通方式配送还是特殊方式配送的方式，如果满足普通配送的条件则向上进入普通配送的流程，如果条件不满足普通配送的条件则向右进入特殊配送的流程。

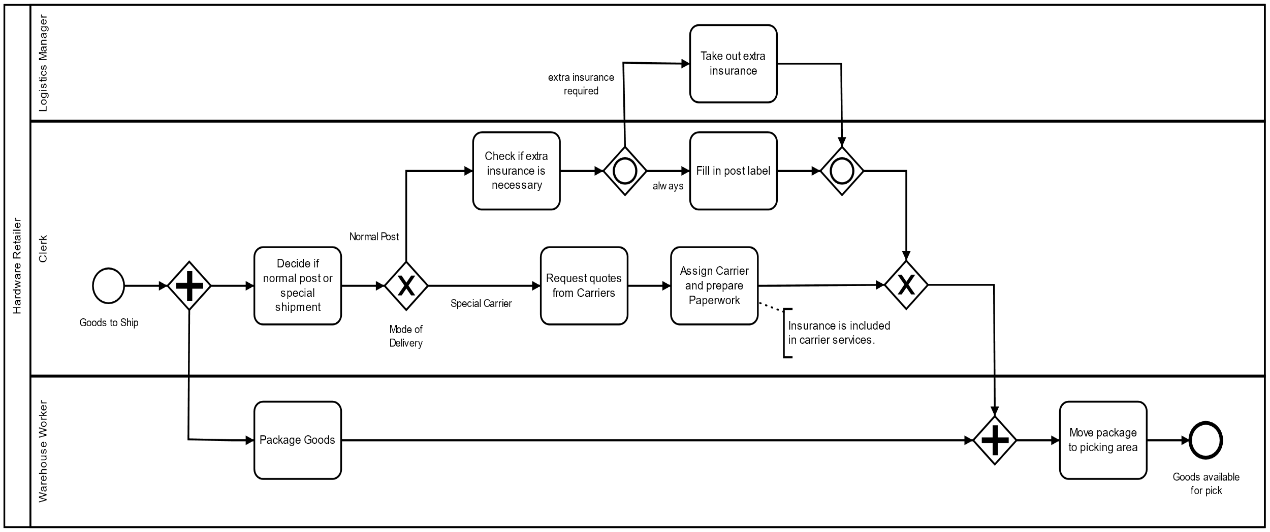


图2.2一个装货过程

* 并行执行：多个业务在同一时刻一起执行，处于并行关系的两个业务之间并不需要互相等待，他们可以一同执行。
* 定时事件：某个业务可能并不是按照顺序或者条件的逻辑作为开始标记，而是以一个时间作为自己的开始条件。如图2.3所示，这是烤蛋糕的过程，执行完检查蛋糕是否做好的业务后进入一个条件判断，如果条件判断为没有做好则定时五分钟后再执行下一个业务。

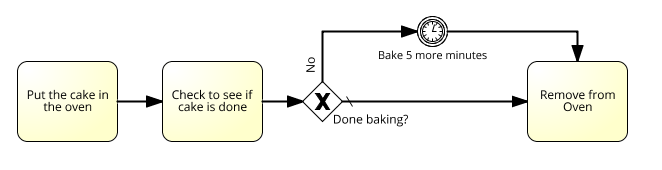


图2.3一个烤蛋糕的过程

对于业务编排之后形成的东西，我们称之为任务模板，统一分析运行平台需要有对任务模板的管理功能，具体分为：创建任务模板、修改任务模板、删除任务模板、获取全部任务模板、单独查询任务模板。

### 2.1.2 上下文

因为有时候并不止是单纯的把任务模板丢到引擎中开始执行，部分任务模板需要有一些环境变量才可以执行，有些业务可能需要传入一定的参数，例如：有些特定的hadoop业务需要在运行时指定结果的存放路径；有些业务有多种执行模式，需要依据环境变量才能判断按照哪种模式执行。

统一分析运行平台提供一个上下文的机制，可以向上下文环境中传入任意的参数，这些参数能够让引擎中的业务在执行的过程中获取，并且要对外提供一个创建上下文、修改上下文、查询上下文以及删除上下文的功能。

### 2.1.3 任务实例管理

在统一分析运行平台中，任务模板是有多个业务组成的一个业务流，而任务实例则是任务模板的一个实例化，在只有一个业务需要执行的情况下，我们也要求它先组成一个任务模板，实例化为任务实例后才能放入引擎中去运行。因此，任务实例就是统一分析运行平台的最小执行单元。

因为统一分析运行平台中会同时运行大量的任务实例，因此需要有对任务实例的管理功能。具体分为：创建任务实例、修改任务实例、删除任务实例、获取全部任务实例、单独查询任务实例。

## 2.2 系统特性需求

随着软件架构的发展，企业软件逐渐从单体应用向分布式的应用过渡，对于企业软件的各种特性要求也越来越多。针对统一运行平台而言，由于该平台是大量业务的管理调度平台，因此该平台对企业有着很高的重要性，并且该平台也会承担大量的数据访问。对于该平台，我们对他的安全性、容错性、高效性、扩展性、部署难度等都有较高的要求。

### 2.2.1 安全性

随着时间的推移，越来越多的大型系统被曝出安全漏洞，这些漏洞往往会导致很多严重而又危险的事情。例如，棱镜计划就是美国国家安全局存在着的安全漏洞，最终导致了整个国家的安全都受到威胁。统一运行平台对公司来说同样是极其重要的一个平台，因此我们在设计系统时一定要尽可能地保证他的安全性。具体需求如下：

* 身份认证和授权：对于整个系统，我们要求它拥有认证和授权机制，并且可以对不同的用户设置不同的权限。对于分布式系统的单个节点，我们要求该节点只对需要直接交互的其他节点进行认证，其余外在系统和平台内不与其直接交互的节点认证均不通过（即进行细粒度的授权）。
* 数据安全：虽然授权机制能在一定程度上保证统一运行平台的安全性，但是一旦操作系统被黑客攻破，我们的敏感数据和密钥都面临着巨大的威胁，因此，我们要求对密钥和敏感信息进行加密处理。
* 防火墙：华为要求所开发的软件必须明确其开放端口，对于不需要的端口我们会全部关闭，以保证系统没有后门也没有多余的端口暴露漏洞。
* 日志：系统要求对所有涉及到数据和登陆的操作都有日志记录。完善的日志虽然起不到预防攻击的作用，但是可以在发生难以预测的事情时重现当时的情景。同样，我们需要在日志中剔除敏感信息。
* 网络隔离：对于分布式系统而言，并不是所有的节点都需要同样的网络环境，我们可以通过将节点分配到不同的网段，这可以进一步避免内部节点的暴露，提高整个系统的安全性。
* 操作系统：操作系统要求选用打好安全漏洞布丁的操作系统，并关闭操作系统的不必要的服务，定期检查该操作系统曝出的安全漏洞，在操作系统出现漏洞的情况下及时做出响应。

### 2.2.2 容错性（鲁棒性）

分布式系统相比单体系统而言，其中一个重要的有点就是他可以对系统的容错性进行很好的设计和控制。系统的容错性是指当系统发生故障时，系统还能保证运行和数据安全，并且可以从错误中恢复过来，其中的故障可能是系统和软件在运行中出现的运行时错误，也可能是机房的硬件故障，甚至是断网断电的情况。对统一分析运行平台来说，要保证部分节点出故障的情况下，平台还能保证功能的完整性，对于故障节点所涉及到的数据可以完整地保存起来，以备之后的恢复。

### 2.2.3 高效性

这里的高效性是指对整个系统的性能需求，是这个系统要能够承受预估范围内的访问数据量。对于统一分析运行平台而言，其在交付后可能会同时运行上万个的任务。我们需要在能预知大概数据量的情况下，不对机器做过多的浪费。

### 2.2.4 扩展性

分布式系统相比单体应用而言，他可以针对特定的模块进行多个扩展。要求统一分析运行平台的多个模块能够按照所需的性能进行扩展，并且要求可以在运行时对模块进行扩展，扩展之后的模块要在一分钟内能够加入系统并分担一部分数据量。

### 2.2.4 部署难度

初次部署要求提供完整详细的文档和脚本，要求系统易于部署。而在之后需要修改平台代码的情况下，我们只需要重新部署这行代码所涉及到的这个节点。

# 第三章 系统设计

## 3.1 系统架构设计

传统的单体结构的应用很难应对每秒上万次的调用，对于应用的容错性也很难保障，应用内一旦出现一个致命性的错误将导致整个应用的停止运行，这对运行商来说将会带来巨大的损失，基于微服务架构的应用可以完美的解决这一问题。

微服务就是使用网络来进行分割模块，每个模块都是一个小的服务，这些小服务彼此间只通过网络进行交互，单独的微服务完全拥有自治的能力，因此微服务的架构可以带来：

* 技术异构性：因为服务之间通过网络进行协作，除此之外，服务之间完全独立，因此可以在不同的服务内使用不同的框架，甚至是不同的语言，为了达到微服务的正确运行，只需要统一服务间的网络协议和数据格式。但是，在一个组织内使用过多的语言也会导致组织难以招聘人才和找不到合适的人交接工作。
* 弹性：在单块系统中，如果一个组件不能用，整个系统都会受到影响，但是在微服务中，我们可以轻易的将一个服务部署成分布式的。
* 扩展：在单体的软件中，想要实现分布式的扩展，则必须将整个系统进行扩展，这意味着及时系统中只有一个小模块存在性能问题，我们也要浪费大量的机器去扩展它。而微服务具有天生的可扩展性，可以针对需要扩展的微服务进行扩展。
* 简化部署：在大型的单体程序中，即使只修改了一行代码，也需要把整个程序进行重新部署，但是在微服务中，我们只需要重新部署这行代码所涉及到的这个微服务即可。
* 与组织结构相匹配：微服务可以有效的将模块分离出来，这也使得整个项目组团队可以分割成在各地的小型团队，服务的所有权也可以随意的在团队间转移，能够保证所有相近服务的团队都在一起工作。
* 可组合性：微服务最大的好处就是可以对各个服务进行编排重组，不同于以往的单纯的Web、PC端程序，人们有更多的选择使用同一个功能。微服务中会开放很多接口供外部调用，当情况发生改变时，可以使用不同的方式构建应用。
* 对于可替代性的优化：在单块的系统中，我们想要修改一写历史遗留的问题时，常常要思考这次修改所需要付出的代价有多大，因此，跟多公司可能还在维护者很多年前就该淘汰的代码，这些代码没人能看得懂，甚至每人敢动他。在微服务上，我们基本可以保证一个服务可以在两周时间内重写，这也就意味着我们想要替换掉这个服务时所付出的代价并不是那么大。

微服务之间的完全通过网络进行了解耦，服务之间的联系就如图3.1所示。

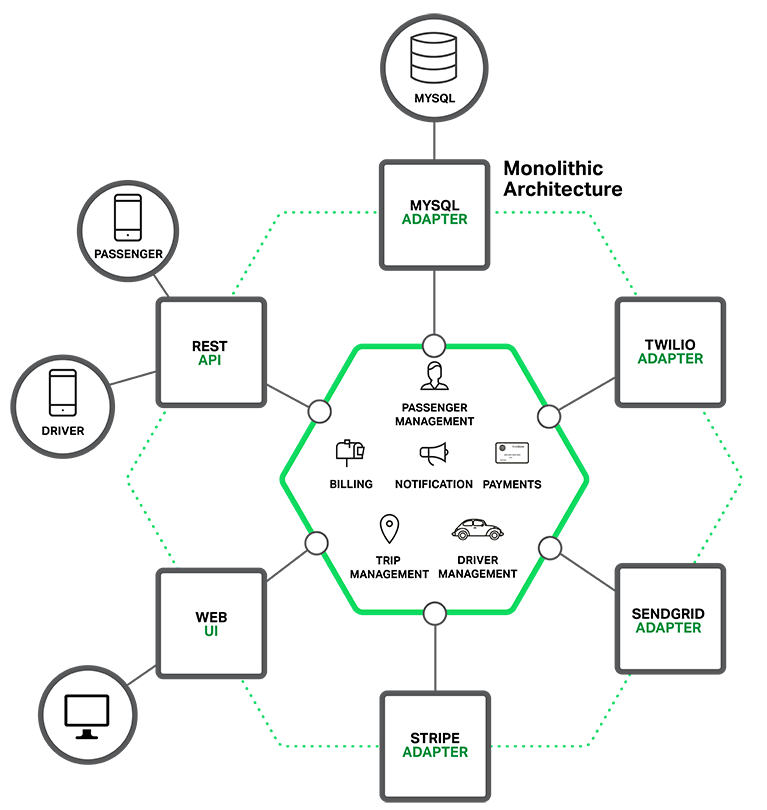


图3.1 微服务之间的联系

综上所述，统一分析运行平台采用微服务架构作为系统架构。

在统一分析运行平台中，我们的架构入图3.2所示，划分为任务模板模块、任务实例模块、BPMN引擎模块、Ignite模块和Eureka模块五个模块，其中负责和用户交互的模块为任务模板模块和任务实例模块，对于调用相应业务的功能由BPMN引擎模块完成。

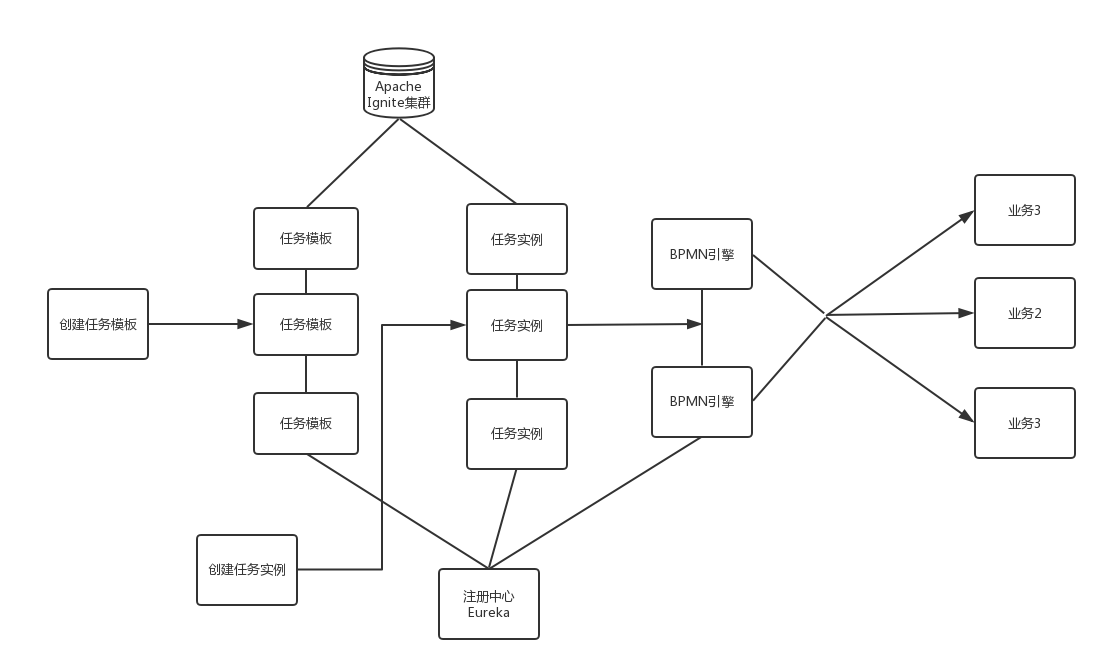


图3.2 系统整体架构

## 3.2 服务交互

对于微服务而言，服务之间通过网络进行交互，常采用的方式为RPC或者REST。下文将通过对比RPC和REST的优缺点，确定在统一分析运行平台中所选择的交互方式。

### 3.2.1 RPC

在分布式计算中，远程过程调用(RPC)是当一个计算机程序引发另一个地址上的程序执行,并且在这一过程中没有涉及到程序的代码细节,就好像它是一个正常的本地调用,无需程序员显式编码远程交互的细节。也就是程序员不管是远程调用还是本地调用都写相同的代码。这是一个典型的client-server交互，调用者是客户端，执行者是服务器，通常通过请求-相应消息传递系统来实现。

RPC的具体实现有很多，如JAVA的RMI、或者通过暴露一个SOAP接口来实现，不同的RPC实现所用的消息格式不同，使用的技术栈也不同，但是作为调用的双方则必须使用相同的格式和技术。

因此，RPC可能带来技术上的耦合，比如RMI只能用于JAVA的平台，因此这就要求被调用者和调用者的双方都选择在JAVA平台上开发，这就对技术选型造成了一定的限制。另一方面，由于RPC像本地调用一样，所以想要修改一个方法则必须对本地和远程的代码都进行修改，这是开发过程中C/S端耦合的一部分。

另外，RPC虽然用起来和本地调用一样，因为RPC将远程调用的中间过程给隐藏起来了，但是这也带来了很多不确定性，比如性能上，不同的RPC调用的性能不同，而且还有网络通信所需要的时间，这就可能导致不明真相的开发人员把远程调用当然本地调用来使用，最后造成性能上的问题，甚至会由于网络的问题而出现各种各样的错误。

### 3.2.2 REST

REST是Representational State Transfer的缩写，它是由Roy Thomas Fielding博士在2000年提出来的一种架构风格。Roy Fielding也是http规范的主要作者之一。

REST是一种设计风格，REST本身有很多的约束和限制，但在这里，我们仅将他和RPC进行对比。REST的关键点为：

* 资源与URI：任何被引用到的事物都可以看作一个资源，例如在URI中，我们的每一个地址都可以看作一个资源，这个地址的命名可以很好的描述这是一个怎样的资源。
* 统一资源接口：这个接口预先定义了可能对资源的操作，放在http里面则对应着http的动词，例如：GET一般表明获取资源，它是安全的，它的操作是幂等的；POST表明更新资源，它意味着将要去修改服务器上的资源，它的操作不是幂等的；PUT一般表明要去创建一个资源，它是不安全的，但是是幂等的；DELETE则是要删除一个资源，是不安全的，但是是幂等的。
* 资源的表述：资源的表述基本上是指用怎样的数据格式去形容这个资源，对外表现为资源的具体呈现，用来进行传递资源。比如文本我们可以使用json或xml，图片我们可以使用PNG或JPG展现，如图3.3所示，这是一个使用json作为资源表述格式的rest请求。



图3.3 http请求

* 资源的链接：这一点奠定了互联网下一代获取资源的方式，就是我们的下一个要访问的资源链接都可以在上一个资源处找到，这也就是超媒体即应用状态引擎的概念，但是目前我们的微服务应用还不能很好的实现这一点。
* 状态的转移：REST是一种无状态通信原则，他在服务端并不会有状态，服务端只维护资源，而客户端维护状态，状态转移就是由客户端维护如何进入下一个状态。

HTTP实现了很多REST的功能，因此HTTP通常会用于构建RESTful软件，我们只需要在使用HTTP时按照REST对资源的定义来实现，就可以基本地实现符合REST风格的软件。

另外HTTP与RPC并不冲突，RPC可以通过HTTP来实现，例如SOAP，但是这并没有完全用到HTTP的统一资源接口等功能。

### 3.2.3 RPC和REST的选择

由上可知，REST的功能和易用性都比RPC要强，但是我们不能说RPC一无是处，一方面部分RPC框架可以自动的生成桩代码，这使得使用RPC的工作量将要比REST小一些，另一方面，基于http的REST的每次请求都需要完好的封装http协议，而http是建立在TCP上层的协议，而且很多的http客户端框架并不能很好的实现http的长连接，这就导致在一次http请求之后，TCP连接会断开，而再次请求还需要重新建立http请求，这就使得在有低时延需求的情景下，http比RPC的时延可能要高。

在系统开发之前，我们针对REST和RPC进行了一些相关的测试，测试方法如下：

* 选择相同的配置的四台机器作为测试机，其中两台用来测试REST的接收和发送，另外两台测试RPC的接收和发送。
* 使用Java RMI实现RPC；使用jersey实现rest服务端，通过封装JAVA内置类HttpURLConnection实现REST客户端；RPC和REST服务端均只在接受到调用后累加器加一。
* 分别为RPC和REST编写测试函数，对两种调用方式的用时进行统计。

RPC客户端代码如下：

publicclass Program {

publicstaticvoid main(String[] args){

try{

//调用远程对象，注意RMI路径与接口必须与服务器配置一致

RPCService rPCService=(RPCService)Naming.lookup("rmi://10.163.14.19:6600/RPCTest");

int counter;

long start = System.currentTimeMillis();

for(int I;I <= 20000;i++){

counter = rPCService.add();

}

long end = System.currentTimeMillis();

System.out.println("Took : " + ((end - start) / 1000));

}

}catch(Exception ex){

ex.printStackTrace();

}

}

}

RPC服务端代码如下：

//此为远程对象的实现类，须继承UnicastRemoteObject

publicclass RPCServiceImpl extends UnicastRemoteObject implements RPCService {

public RPCServiceImpl() throws RemoteException {

counter = 0;

super();

}

int counter;

@Override

Int add() throws RemoteException {

counter++;

return counter;

}

}

其中，在客户端代码中对RPC服务端进行了20000次调用，并统计了20000次调用的时间。

REST服务端代码：

@Path({"test "})

public class RestTest

{ @GET

@Path({"add"})

public int doGet(

)

{

counter++;

return counter;

}

}

REST客户端代码：

URL url = new URL("http:// 10.163.14.31:8080/test/add");

URLConnection rulConnection = url.openConnection();

HttpURLConnection httpUrlConnection = (HttpURLConnection) rulConnection;

httpUrlConnection.setRequestMethod("GET");

httpUrlConnection.setDoOutput(true);

httpUrlConnection.setDoInput(true);

httpUrlConnection.setUseCaches(false);

java.io.EOFException)

httpUrlConnection.setRequestProperty("Content-type", "application/x-java-serialized-object");

httpUrlConnection.connect();

OutputStream outStrm = httpUrlConnection.getOutputStream();

int counter;

long start = System.currentTimeMillis();

for(int I;I <= 20000;i++){

ObjectOutputStream objOutputStrm = new ObjectOutputStream(outStrm);

objOutputStrm.writeObject();

objOutputStm.flush();

objOutputStm.close();

}

long end = System.currentTimeMillis();

System.out.println("Took : " + ((end - start) / 1000));

最终测试结果如表3.1所示，其中RPC和REST均交换机器再次测试了一次，由表可以看出，以这种方式实现的RPC比REST的性能稍好，但是两者的差距并不是很大。

表3.1 RPC和REST性能测试

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 测试次数 | 总用时 | 平均时间 |
| RPC | 20000 | 18.860s | 0.943ms |
| REST | 20000 | 20.076s | 1.0038ms |
| RPC | 20000 | 17.779s | 0.88895ms |
| REST | 20000 | 20.430s | 1.0215ms |

因此，我建议，在通常的微服务开发中，选择REST风格的模式，这样更利于开发和维护；而在有低延迟需求的微服务开发中选用RPC来做通信。在本文所介绍的统一分析运行平台中，低延迟并不是我们所要保证的重点内容，因此我们选择更为规范的REST作为我们服务间的交互方式，同时选择json作为交互内容的载体。

## 3.3 微服务模块划分

为了实现业务编排、上下文和task管理的功能性需求，将系统的模块划分：任务模板模块、任务实例模块、BPMN引擎模块、Ignite模块和Eureka模块。

### 3.3.1 任务模板模块

任务模板模块负责对任务模板的管理，任务模板就是指一连串的编排好的业务流程，在统一分析运行平台中以BPMN的形式表示。正如前面需求分析中提到的需求，我们再任务模板模块中实现这些需求：创建任务模板、修改任务模板、删除任务模板、获取全部任务模板、单独查询任务模板。按照需求分析，我们应当提供对指定模板的修改功能，但是考虑到修改模板可以通过先删除模板再创建模板来实现，而实际过程中对修改的要求并不高，所以修改任务模板的功能并不需要在任务模板模块实现。

该模块对外开放的rest接口为表3.2所示。

表3.2 任务模板模块的REST接口

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 请求方法 | 返回类型 | 功能 |
| /template | post | json | 创建任务模板 |
| /template/id | get | json | 获取指定任务模板 |
| /template/id | delete | json | 删除指定任务模板 |
| /templates | get | json | 获取全部任务模板的id |

该模块需要依赖Ignite模块存储业务流程，在Ignite中创建一个名为“Template”的cache块，该cache块可供任务模板模块和任务实例模块访问。其中key为该流程的id，value为该流程的BPMN。

当创建模板时，首先调用/template接口的post方法，要求在请求的body中以json的形式封装BPMN的标准xml，如果任务模板创建成功则返回200 OK，并在返回的body中附带所创建的任务模板的id；如果创建失败则根据情况返回相应的状态码，具体如表3.3所示。接收的json格式为：

｛

“BPMN”: “<?xml version=\"1.0\" encoding=\"UTF-8\"?>

<definitions xmlns=\"http://www.omg.org/spec/BPMN/20100524/MODEL\" xmlns:bpmndi=\"http://www.omg.org/spec/BPMN/20100524/DI\" xmlns:omgdi=\"http://www.omg.org/spec/DD/20100524/DI\" xmlns:omgdc=\"http://www.omg.org/spec/DD/20100524/DC\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" id=\"sid-38422fae-e03e-43a3-bef4-bd33b32041b2\" targetNamespace=\"http://bpmn.io/bpmn\" exporter=\"http://bpmn.io\" exporterVersion=\"0.10.1\">

<process id=\"Process\_1\" isExecutable=\"false\">

<startEvent id=\"StartEvent\_1\" name=\"hunger noticed\">

<outgoing>SequenceFlow\_1</outgoing>

</startEvent>

<task id=\"Task\_1\" name=\"choose recipe\">

<incoming>SequenceFlow\_1</incoming>

<outgoing>SequenceFlow\_2</outgoing>

</task>

<exclusiveGateway id=\"ExclusiveGateway\_1\" name=\"desired dish?\" gatewayDirection=\"Diverging\">

<incoming>SequenceFlow\_2</incoming>

</exclusiveGateway>

<sequenceFlow id=\"SequenceFlow\_1\" name=\"\" sourceRef=\"StartEvent\_1\" targetRef=\"Task\_1\" />

<sequenceFlow id=\"SequenceFlow\_2\" sourceRef=\"Task\_1\" targetRef=\"ExclusiveGateway\_1\" />

</process>

</definitions>”

｝

其中xml的部分需要对引号进行转义处理，该xml可以通过界面的拖拽生成。

表3.3 创建任务模板时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 创建成功 | 该任务模板的id | json | {”id”: ” 42fb3c84da067494”} |
| 400 Bad Request | 请求格式有问题 | 返回请求体的具体问题 | json | {“reason”: ”formatException”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

创建任务模板的具体流程为：任务模板模块接收到请求，取出请求body中的xml，将此xml作为value,计算该xml的md5值并作为这个业务流程的id，将id和value存入Ignite模块的“Template”这个cache块中，最后将此id封装成json格式后返回。

当获取指定任务模板时，需要对/template/id接口发起一个get请求，如果任务模板查询成功则返回200 OK，并在返回的body中附带所查询的任务模板的信息；如果创建失败则根据情况返回相应的状态码，具体如表3.4所示。

表3.4 查询指定任务模板时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 查询成功 | 该任务模板的id和具体内容 | json | {”id”: ” 42fb3c84da067494”,”BPMN”: “完整的BPMN的xml，此处略”} |
| 404 Not Found | 任务模板不存在 | 指定id的任务模板不存在 | json | {“reason”: ” 42fb3c84da067494 NOT FOUND”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

当删除指定任务模板时，需要对/template/id接口发起一个delete请求，如果任务模板删除成功则返回200 OK，并在返回的body中附带所删除的任务模板的id；如果删除失败则根据情况返回相应的状态码，具体如表3.5所示。

表3.5 删除指定任务模板时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 删除成功 | 该任务模板的id和具体内容 | json | {”id”: ” 42fb3c84da067494” } |
| 404 Not Found | 任务模板不存在 | 指定id的任务模板不存在 | json | {“reason”: ” 42fb3c84da067494 NOT FOUND”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

当获取全部任务模板时，需要对/templates接口发起一个get请求，如果任务模板查询成功则返回200 OK，并在返回的body中附带所有任务模板的id；如果查询失败则根据情况返回相应的状态码，具体如表3.6所示。

表3.6 获取全部任务模板的id时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 查询成功 | 所有任务模板的id | json | {”ids”: [” 42fb3c84da067494”,” 43b47fd5eed9eceb”,” 77391bb04c2f8291”] } |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

### 3.3.2 任务实例模块

任务实例模块负责对任务实例进行管理。因为在需求中我们提出了上下文的定义，因此，我们通过用任务模板表示编排好的业务流程，而任务实例则是附带有上下文的业务流程。在这个模块中，任务实例一旦创建完成，就会按照上下文进入运行环节，如果是即刻执行的业务流程就会直接发送给BPMN引擎模块，如果是有定时的任务则会在任务实例模块中等待，等到开始时间则会发送给BPMN引擎模块。我们在任务实例模块中需要实现：创建任务实例、查询任务执行状态、获取全部任务实例。因为终止正在运行中的任务实例所涉及到的逻辑较为复杂，因此终止任务执行在当前版本并未实现。此外由于任务实例模块和BPMN模块之间采用了异步通信的方式，因此，任务状态只有三种：任务还未发送给BPMN模块；BPMN模块正在处理任务；任务完成。为了实现和BPMN模块之间的异步调用，任务实例模块需为BPMN模块单独开放一个用来回调的rest接口。

综上所述，该模块开放的rest接口如表3.7所示。

表3.7 任务实例模块的REST接口

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 请求方法 | 返回类型 | 功能 |
| /task | post | json | 创建任务实例 |
| /task/id | get | json | 查询指定任务实例的运行状态 |
| /tasks | get | json | 获取全部任务实例 |
| /status | post | json | 供BPMN模块回调修改状态 |

该模块需要依赖Ignite模块存储业务流程，在Ignite中创建一个名为“Task”的cache块，该cache块可供任务实例模块访问。其中key为该task的id，value为该任务实例的内容。

当创建任务实例时，首先调用/task接口的post方法，要求在请求的body中以json的形式封装任务模板的id和上下文，在目前的版本，上下文包括：是否即刻开始、开始时间、任务运行中的环境变量、是否循环执行、循环执行的次数，其中，如果即刻开始任务则不需要提供开始时间，不循环执行也不需要提供循环次数。如果任务实例创建成功则返回200 OK，并在返回的body中附带所创建的任务模板的id；如果创建失败则根据情况返回相应的状态码，具体如表3.8所示。接收的json格式为：

｛

“templateId”: “42fb3c84da067494”,

“timer”: true,

“startTime”: 1496246400,

“loop”: true,

“loopTimes”: 2,

“environment”:

{

“localtion”: “/var/digData/task1”,

“number”: 20

}

｝

其中，开始时间格式为以秒为单位的时间戳。

表3.8 创建任务实例时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 创建成功 | 所创建的任务实例的id | json | {”id”: “77391bb04c2f8291” } |
| 400 Bad Request | 请求格式有问题 | 返回请求体的具体问题 | json | {“reason”: ”formatException”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

当查询指定任务实例时，需要对/task/id接口发起一个get请求，如果任务实例查询成功则返回200 OK，并在返回的body中附带所查询的任务实例的信息；如果创建失败则根据情况返回相应的状态码，具体如表3.9所示。

表3.9 查询指定任务实例时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 查询成功 | 该任务实例的id和具体内容 | json | {”id”: ” 77391bb04c2f8291”,”templateId”: “42fb3c84da067494”, “timer”:true,  “startTime”: 1496246400,  “loop”: true,  “loopTimes”: 2, “environment”:{  “localtion”: “/var/digData/task1”,  “number”: 20,  ”status”: “await”}} |
| 404 Not Found | 任务实例不存在 | 指定id的任务实例不存在 | json | {“reason”: ” 77391bb04c2f8291 NOT FOUND”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

当获取全部任务实例时，需要对/tasks接口发起一个get请求，如果任务实例查询成功则返回200 OK，并在返回的body中附带所有任务实例的id；如果查询失败则根据情况返回相应的状态码，具体如表3.10所示。

表3.10 获取全部任务实例的id时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 查询成功 | 所有任务实例的id | json | {”ids”: [” 42fb3c84da067494”,” 43b47fd5eed9eceb”,” 77391bb04c2f8291”] } |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

当BPMN模块调用/status接口修改任务实例状态时，需要对/status接口发起一个post请求，如果状态修改成功则返回200 OK，并在返回的body中附带所有任务实例的id；如果修改失败则根据情况返回相应的状态码，具体如表3.11所示。

表3.11 修改task状态时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 修改成功 | 无 | - | - |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “IgniteLackException”} |

### 3.3.3 BPMN引擎模块

BPMN引擎模块用来处理和调用相应的业务流程。该模块仅与任务实例模块之间有显式的交互，它不与任务模板模块交互，它也不需要ignite为他提供存储环境。

标准业务流程模型和标记法（BPMN）将为企业提供以图形符号理解其内部业务流程的能力，并将使组织能够以标准方式传达这些流程。此外，图形符号将有助于了解组织之间的绩效协作和业务交易。这将确保企业了解自身和参与者的业务，并使组织能够快速适应新的内部和B2B业务环境。业务流程管理计划（BPMI）开发了BPMN，自2005年它与对象管理团队（OMG）合并以后，BPMN开始由OMG维护。BPMN 2.0版于2011年1月发布。

在这里，我们选用BPMN来表示业务流程可以很好的满足规范性的标准，而且BPMN作为图形符合的标记法，他可以通过界面的拖拽来生成业务流程，这对使用者来说更为友好，而且在BPMN2.0的标准中，BPMN增加了生成对应xml的功能，因此，我们采用了BPMN作为统一分析运行平台的业务流程的表示方法。

因此BPMN引擎模块提供的功能可分为：接收业务流程；执行业务流程；执行结束后回调任务实例模块。

该模块对外开放的rest接口为表3.12所示。

表3.12 BPMN引擎模块的REST接口

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 请求方法 | 返回类型 | 功能 |
| /bpmn | post | json | 将业务流程放入BPMN引擎中运行 |
| /bpmn/taskId | get | json | 获取指定task的状态 |

虽然在当前版本的统一分析运行平台中并没有提供查询详细的task状态的功能，只提供了和BPMN模块无关的三个状态，但是在之后的版本中会对这个功能进行完善，使统一分析运行平台可以查询详细的task状态。

当调用/bpmn接口时，要求在请求的body体中附加BPMN，这个BPMN是经过任务实例模块处理的BPMN，其中附加有业务流的环境变量。具体的json可如下表示：

｛

“BPMN”: “<?xml version=\"1.0\" encoding=\"UTF-8\"?>

<definitions xmlns=\"http://www.omg.org/spec/BPMN/20100524/MODEL\" xmlns:bpmndi=\"http://www.omg.org/spec/BPMN/20100524/DI\" xmlns:omgdi=\"http://www.omg.org/spec/DD/20100524/DI\" xmlns:omgdc=\"http://www.omg.org/spec/DD/20100524/DC\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" id=\"sid-38422fae-e03e-43a3-bef4-bd33b32041b2\" targetNamespace=\"http://bpmn.io/bpmn\" exporter=\"http://bpmn.io\" exporterVersion=\"0.10.1\">

<process id=\"Process\_1\" isExecutable=\"false\">

<startEvent id=\"StartEvent\_1\" name=\"hunger noticed\">

<outgoing>SequenceFlow\_1</outgoing>

</startEvent>

<task id=\"Task\_1\" name=\"choose recipe\">

<incoming>SequenceFlow\_1</incoming>

<outgoing>SequenceFlow\_2</outgoing>

</task>

<exclusiveGateway id=\"ExclusiveGateway\_1\" name=\"desired dish?\" gatewayDirection=\"Diverging\">

<incoming>SequenceFlow\_2</incoming>

</exclusiveGateway>

<sequenceFlow id=\"SequenceFlow\_1\" name=\"\" sourceRef=\"StartEvent\_1\" targetRef=\"Task\_1\" />

<sequenceFlow id=\"SequenceFlow\_2\" sourceRef=\"Task\_1\" targetRef=\"ExclusiveGateway\_1\" />

</process>

</definitions>”

｝

其中xml的部分需要对引号进行转义处理，该xml是最开始用户传给任务模板的BPMN经过任务实例模块处理的xml。

表3.13业务流程进入BPMN引擎时返回的状态码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 意义 | 返回body内容 | body格式 | 样例 |
| 200 OK | 任务成功开始 | 该任务实例的id | json | {”id”: ” 77391bb04c2f8291”} |
| 400 Bad Request | 请求格式有问题 | 返回请求体的具体问题 | json | {“reason”: ”formatException”} |
| 401 Unauthorized | 请求认证失败 | 认证失败的原因,在安全部分的实现处详细介绍 | json | {“reason”: “certificateLackException”} |
| 500 Internal Server Error | 服务器错误 | 具体出错的原因 | json | {“reason”: “CamundaException”} |

### 3.3.4 Ignite模块

Ignite模块负责为任务模板模块和任务实例模块提供存储的功能。

该模块采用了apache基金会开源的项目Apache Ignite，Apache Ignite是一种高性能，集成和分布式内存平台，用于实时计算和处理大规模数据集，比传统的基于磁盘或闪存技术的速度更快。

Ignite具有如下特点：

* 先进的集群：ignite节点可以自动发现其他节点。这时得在有需要扩大集群规模时不需要重启整个集群。开发人员还可以利用Ignite的混合云支持，从而建立私有云和公有云之间的连接，如亚马逊网络服务，为他们提供最好的两个世界。图3.4所示为Ignite构建集群时的架构图。

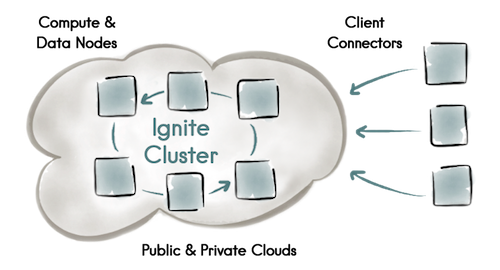


图3.4 Ignite集群

* 数据网格（JCache）：Ignite数据网格是一个内存中分布式密钥值存储，可以被视为分布式分区哈希映射，每个集群节点拥有整个数据的一部分。 这样，我们添加的集群节点越多，我们可以缓存的数据越多。与其他键值存储区域不同，Ignite使用可插入散列算法来确定数据的局部性。 每个客户端可以通过将密钥插入散列函数来确定密钥属于哪个节点，而不需要任何特殊的映射服务器或名称节点。Ignite数据网格支持本地，复制和分区数据集，并允许使用标准SQL语法在这些数据集之间自由交叉查询。 Ignite支持用于查询内存数据的标准SQL，包括支持分布式SQL连接。

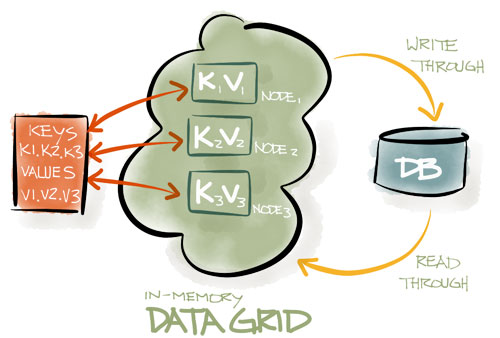


图3.5 Ignite数据网格

* 流媒体和CEP：Ignite流式传输允许以可扩展和容错的方式处理连续不间断的数据流。 数据可以注入Ignite的速度可能非常高，并且在中等大小的集群上容易超过每秒数百万事件。实时数据通过数据流传送。 我们已经为JMS 1.1，Apache Kafka，MQTT，Twitter，Apache Flume和Apache Camel提供了流媒体，并且我们每个版本都不断添加新的。

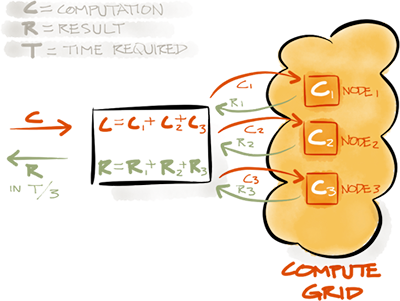


图3.6 apache ignite流媒体

* 服务网格：服务网格允许在群集上部署任意用户定义的服务。 您可以实现和部署任何服务，如自定义计数器，ID生成器，分层映射等。Ignite允许您控制在每个集群节点上部署多少服务实例，并自动确保所有服务的正确部署和容错。



图3.7 apache ignite服务网格

* Ignite文件系统：Ignite文件系统（IGFS）是一种内存中文件系统，允许在现有缓存基础架构上处理文件和目录。 IGFS可以作为纯内存文件系统工作，也可以委托给另一个充当缓存层的文件系统（例如各种Hadoop文件系统实现）。此外，IGFS还提供了API来对文件系统数据执行映射减少任务。
* 分布式数据结构：Ignite以分布式的方式支持复杂的数据结构：队列和集合：普通，有界，并行的，非并行的；原子类型：AtomicLong和AtomicReference；闭锁；ID发生器。
* 分布式消息传递：分布式消息传递允许所有节点之间的基于主题的集群范围的通信。 具有指定消息主题的消息可以分发给已订阅该主题的所有节点或子组。Ignite消息传递基于发布订阅范例，其中发布者和订阅者通过共同的主题连接在一起。 当其中一个节点向主题T发送消息A时，它将在已订阅T的所有节点上发布。
* 分布式事件：分布式事件允许应用程序在分布式网格环境中发生各种事件时接收通知。 您可以自动收到在集群内的本地或远程节点上发生的任务执行，读取，写入或查询操作的通知。

根据Ignite的诸多特性，我们选择Ignite而不是数据库作为我们的存储模块。Ignite的存储是基于cache块的，在cache块中可以选择用key-value的方式存储，Ignite还提供了直接存贮对象的功能，同时可以很方便地通过Ignite的接口将对象反序列化出来。

按照之前对任务模板模块和任务实例模块的划分，我们在Ignite中创建两个cache块：“Template”和“Task”。

### 3.3.5 Eureka模块

Eureka是Netflix服务发现的一个开源的实现。Eureka是我们组件集群的关键，Eureka可以提供一个供其他服务注册和发现对方的功能，所有的服务都会注册到这个Eureka上面，因此，所有的服务都可以通过Eureka来获取彼此的地址。

如图3.8所示，其中有四个任务模板服务注册到Eureka模块上面，其中任何一个任务模板服务都可以通过询问Eureka模块发现其他的三个任务模板服务。同理，当有别的服务注册到Eureka模块上时，也可以通过询问Eureka模块来发现他们的地址。

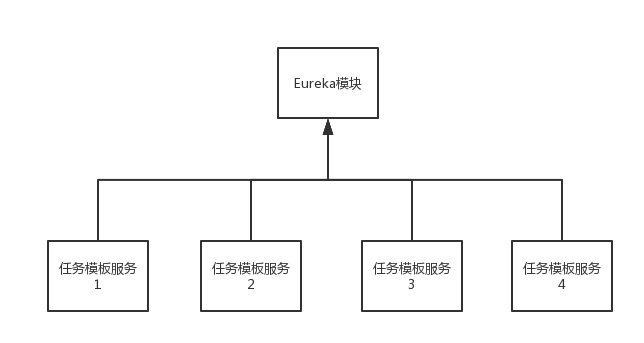


图3.8 apache ignite服务网格

在统一分析运行平台中，我们以app表示一种微服务，例如任务模板模块、任务实例模块和BPMN引擎模块都是一种微服务，而实体（instance）则表示为一个微服务的实例，例如，我们在图3.8中启动了4个任务模板服务，其中所启动每个服务都称之为一个实体。Eureka的目的就是让任务模板模块可以发现所有的任务实例模块的实体，任务实例模块也可以发现所有的BPMN引擎模块的实体。因此，他开放的rest接口如表3.14所示。

表3.14 Eureka模块的REST接口

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 请求方法 | 返回类型 | 功能 |
| /eureka/v2/apps/appID | post | json | 注册一个app到eureka上 |
| /eureka/v2/apps/appID/instanceID | delete | json | 从Eureka上删除一个app的实体 |
| /eureka/v2/apps | get | json | 查询全部的实体 |
| /eureka/v2/apps/appID | get | json | 查询当前appID的全部实体 |
| /eureka/v2/apps/appID/instanceID | get | json | 查询指定的实体 |

Eureka模块可以使微服务实体之间可以相互发现，那么，该调用哪个实体就需要一定的路由规则，在当前版本的统一分析运行平台中，我们的路由策略为随机调用，即在所有的同种服务的实体中随机选择一个实体去调用。例如：任务实例模块需要调用BPMN引擎模块，那么他通过Eureka模块可以发现一共有n个BPMN引擎模块的实体，那么他就会随机选择一个BPMN引擎模块实体去调用。

## 3.4 系统特性的实现

### 3.4.1 安全性

对于所有的rest接口，我们采用http基本认证+ssl证书认证。

其中basic通过在http请求的header中增加 Authorization 字段，并且要求header中的用户名和密码结合在一起通过base64编码后传输。而ssl由于则会要求服务端提供证书，并且在第一次请求后将证书传至客户端，客户端完全信任证书并不对证书进行验证，这一步并不是为了确保客户端和正确的服务器建立连接，而是为了保证在网络交互的过程中信息不被恶意抓取。由于在HTTPs的传输过程中，header也经过了ssl加密，因此，Authorization 字段不会被抓包等手段恶意获取。

在basic中，我们引入了多用户的机制，对于不同的服务，我们针对特定的用户进行授权，保证了用户不会越权操作。

针对ignite模块，一方面我们通过ignite提供的认证功能保证集群的正确建立，另一方面，因为我们通过docker集群来组件ignite集群，因此我们可以通过kubernetes提供的网络隔离和DNS发现功能来将属于同一集群的ignite隔离开来。

对于docker容器，我们严格限制开放的网络端口。

### 3.4.2 容错性（鲁棒性）

因为我们采用了微服务架构，并且利用Eureka模块的服务注册和服务发现机制实现了每个服务都拥有多个备份的特定，在单个服务实体崩溃掉后，其他的服务实体还是能够保证系统的正常运行的。因为任务模板和任务实例都存储在ignite中，因此，即使这两个模块的实体崩溃并不会使系统的数据丢失。而Apache Ignite本身有具有多机备份和冗余的功能，因此部分Ignite节点崩溃也不会使Ignite中的数据丢失。

### 3.4.3 高效性

在开发的过程中，在保证安全和功能的实现上，尽可能地选择高性能的实现方式，并且在开发结束后需要通过系统测试来检测整个系统中每个模块所需的实体数量，例如BPMN引擎模块比任务模板模块需要更多的运行时间和机器性能，因此，我们可以在部署阶段部署更多的BPMN引擎实体，而任务模块实体相对较少地部署。

### 3.4.4 扩展性

微服务具有天生的可扩展性，可以针对需要扩展的微服务进行扩展（见3.4.3）。如果在运行阶段需要对服务进行扩展，即部署更多的app实体，我们只需要正确的配置Eureka模块的地址，使新配置的实体能够注册到Eureka上面，那么在下次调用该模块时就能发现这个实体了。例如：在统一分析运行平台工作一段时间后，我们将一个新的BPMN模块的实体注册到了Eureka上面，那么在下次任务实例模块需要调用BPMN引擎模块时，它就能发现这个新加入的BPMN引擎实体，并根据自身的规则去选择该调用的实体。

### 3.4.5 部署难度

开发人员需将每个服务都封装到一个Docker容器中，并为运维人员提供一个部署脚本将这些Docker容器安装和启动起来。

Docker是世界上领先的软件容器平台。开发人员使用Docker来消除与同事的代码协作时的“我机器上的工作”问题。运营商使用Docker在独立的容器中并行运行和管理应用程序，以获得更好的计算密度。企业使用Docker构建灵活的软件传送管道，可以更快，更安全地运行新功能，并且对于Linux和Windows Server应用程序都具有很高的安全性。

使用容器，使一件软件运行所需的一切都被打包成隔离的容器。与VM不同，容器不捆绑完整的操作系统-只需要使软件工作所需的库和设置。这使得系统高效，轻便，并保证无论软件在哪里部署将始终运行相同。

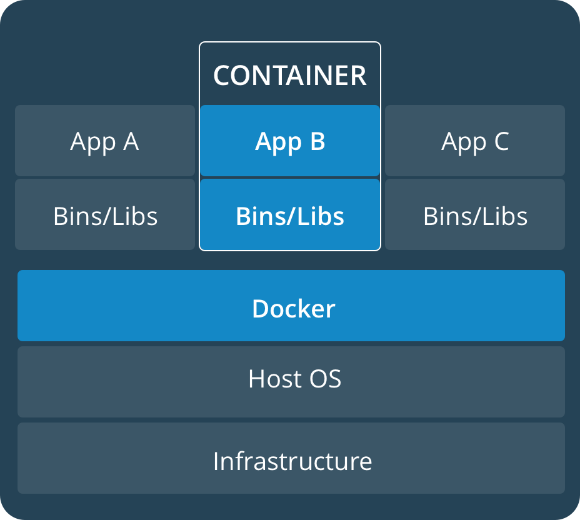
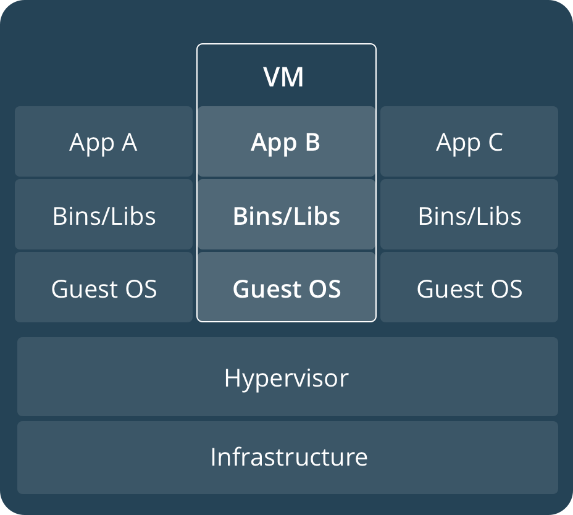
 

图3.8 docker与vm结构比较

容器是应用层的一个抽象，它将代码和依赖项集成在一起。多个容器可以在同一台机器上运行，并与其他容器共享操作系统内核，每个容器都以用户空间中的隔离进程运行。容器占用的空间少于虚拟机（容器映像的大小通常为几十MB），并且几乎立即启动。

# 第四章 系统实现与测试

## 4.1 工具类的实现

### 4.1.1 REST客户端

该类封装了HttpURLConnection类，针对统一分析运行平台所需的REST功能进行了封装，因为在设计部分设计了REST请求需要有http基本认证和ssl证书认证，因此我们封装的REST客户端需要实现这个功能。

对于http基本认证，由于HttpURLConnection并未提供这个功能，根据HTTP1.0规范，我们需要在请求的header中增加一个“Authorization”字段。将用户名和密码用冒号拼接起来后再使用Base64编码，之后与“Basic”用空格连接起来就是这个字段的值。因为Base64是可逆编码，因此我们可以在 我们通过以下代码实现：

HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();

String encoded = Base64.getEncoder().encodeToString((username+":"+password).getBytes(StandardCharsets.UTF\_8));

connection.setRequestProperty("Authorization", "Basic "+encoded);

对于ssl加密，标准的HTTPs浏览器会通过可信的认证机构认证服务器所持有的证书，但因为统一分析运行平台并不一定运行在互联网的环境中，因此对于ssl加密的实现并不应该依赖认证机构的认证。因此，通过一下代码实现：

// Create a trust manager that does not validate certificate chains

TrustManager[] trustAllCerts = new TrustManager[]{

new X509TrustManager() {

public java.security.cert.X509Certificate[] getAcceptedIssuers() {

return null;

}

public void checkClientTrusted(

java.security.cert.X509Certificate[] certs, String authType) {

}

public void checkServerTrusted(

java.security.cert.X509Certificate[] certs, String authType) {

}

}

};

// Install the all-trusting trust manager

try {

SSLContext sc = SSLContext.getInstance("SSL");

sc.init(null, trustAllCerts, new java.security.SecureRandom());

HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

} catch (Exception e) {

}

// Now you can access an https URL without having the certificate in the truststore

try {

URL url = new URL("https://hostname/index.html");

} catch (MalformedURLException e) {

}

在这段代码中，我们自己实现了TrustManager这个接口，在我们的实现中，REST客户端会接受并信任所有的ssl证书。

### 4.1.2 REST服务端

这个类使用了Jersey来构建REST接口。

Jersey是一个实现JAXA-RS的REST框架。Jersey提供他门自己扩展JAXA-RS的工具包的API以进一步简化RESTful服务和客户端开发。Jersey同样公开了许多接口供开发者们扩展Jersey以适合他们的需求。Jersey具有以下的特点：

* 追踪JAXA-RS的API并且提供参考GlassFilsh实现的高质量的稳定版本。
* 提供扩展Jersey的API并且为用户和开发者们构建一个社区。
* 让使用JAVA和JAVA虚拟机开发RESTful Web服务便得容易。

在使用Jersey开发RESTful服务时同样需要为web容器提供一个web.xml来配置servlet调用顺序和filter等。具体的文件如下：

<servlet>

<servlet-name>Jersey REST Service</servlet-name>

<servlet-class>

com.sun.jersey.spi.container.servlet.ServletContainer

</servlet-class>

<init-param>

<param-name>com.sun.jersey.config.property.packages</param-name>

<param-value>sample.hello.resources</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Jersey REST Service</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

之后需要在资源处以注解的形式标记出路径、请求方法和返回内容的载体，如下是一个使用TEXT作为返回类型，接口路径为/hello的GET请求的定义方法，该接口返回的内容为”Hello Jersey”。

@Path("/hello")

public class HelloResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String sayHello() {

return "Hello Jersey";

}

}

### 4.1.3 编码解码包

由于统一分析运行平台中有很多涉及到编解码的地方，因此将编解码的类总结起来放在这个包中，具体包括MD5加密、Base64编码、Base64解码等。实现如下：

Base64：

public class base64 {

/\*\*

\* BASE64解密

\*

\* @param key the String to be decrypted

\* @return byte[] the data which is decrypted

\* @throws Exception

\*/

public static byte[] decryptBASE64(String key) throws Exception {

return Base64.decode(key,Base64.DEFAULT);

}

/\*\*

\* BASE64加密

\*

\* @param key the String to be encrypted

\* @return String the data which is encrypted

\* @throws Exception

\*/

public static String encryptBASE64(byte[] key) throws Exception {

return Base64.encodeToString(key, Base64.DEFAULT);

}

}

md5:

public class md5 {

public static byte[] encryptMD5(byte[] data) throws Exception {

MessageDigest md5 = MessageDigest.getInstance("MD5");

md5.update(data);

return md5.digest();

}

}

aes:

public class aes {

public static final String ALGORITHM = "AES";

public static byte[] decryptBASE64(String key) throws Exception {

return Base64.decode(key, Base64.DEFAULT);

}

public static String encryptBASE64(byte[] key) throws Exception {

return Base64.encodeToString(key, Base64.DEFAULT);

}

private static Key toKey(byte[] key) throws Exception {

//DESKeySpec dks = new DESKeySpec(key);

//SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);

//SecretKey secretKey = keyFactory.generateSecret(dks);

// 当使用其他对称加密算法时，如AES、Blowfish等算法时，用下述代码替换上述三行代码

SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);

return secretKey;

}

public static byte[] decrypt(byte[] data, String key) throws Exception {

Key k = toKey(decryptBASE64(key));

Cipher cipher = Cipher.getInstance(ALGORITHM);

cipher.init(Cipher.DECRYPT\_MODE, k);

return cipher.doFinal(data);

}

public static byte[] encrypt(byte[] data, String key) throws Exception {

Key k = toKey(decryptBASE64(key));

Cipher cipher = Cipher.getInstance(ALGORITHM);

cipher.init(Cipher.ENCRYPT\_MODE, k);

return cipher.doFinal(data);

}

public static String initKey() throws Exception {

return initKey(null);

}

public static String initKey(String seed) throws Exception {

SecureRandom secureRandom = null;

if (seed != null) {

secureRandom = new SecureRandom(decryptBASE64(seed));

} else {

secureRandom = new SecureRandom();

}

KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);

kg.init(secureRandom);

SecretKey secretKey = kg.generateKey();

return encryptBASE64(secretKey.getEncoded());

}

}

## 4.2 模块实现

### 4.2.1 任务模板模块

该模块基于Spring Boot作为运行中的框架，Spring Cloud作为微服务的框架。

Spring Boot使得创建一个独立的基于Spring应用框架的软件变得容易，你只需要去运行它。特点是：

* 创建一个独立的Spring应用；
* 直接嵌入Tomcat,Jetty或者Undertow（不再需要部署WAR文件）；
* 提供一个开始的POM文件简化Maven配置；
* 只要有可能就自动配置Spring；
* 提供指标、健康检查、具体的配置来指示生产就绪；
* 绝对没有代码生成和不需要配置XML。

Spring Franmework为现代的基于JAVA的企业应用提供一个全面的编程和配置模型。Spring的一个关键因素是在应用层面的基础设施支持：Spring侧重于企业应用程序的“管道”，以便团队可以专注于应用程序级的业务逻辑，而不需要与特定部署环境的不必要的联系。特点是：

* 依赖注入；
* 包括Spring的声明事务管理的面向切片编程；
* Spring MVC web应用和RESTful web服务框架；
* 对JDBC、JPA、JMS的基础支持。

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线，一次性令牌，全局锁，领导选举，分布式 会话，集群状态）。分布式系统的协调导致样板模式，并且使用Spring Cloud开发人员可以快速站起来实现这些模式的服务和应用程序。 他们将在任何分布式环境中运行良好，包括开发人员自己的笔记本电脑，裸机数据中心，以及Cloud Foundry等托管平台。

Spring Cloud基于Spring Boot，通过提供一堆库，可以在添加到类路径时增强应用程序的行为。可以利用基本的默认行为快速入门，然后在需要时，可以配置或扩展以创建自定义解决方案。

在任务模板模块，我们首先引入POM文件引入任务模板所需的依赖：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>chenhaoyuan</groupId>

<artifactId>uarp</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>uarp-template</name>

<description>task template service for uarp </description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.3.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

在这个模块中，我们需要连接Ignite并将任务模板存入其中。因此，需要Ignite的配置文件ignite.xml如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:util="http://www.springframework.org/schema/util"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd">

<!-- Datasource for sample in-memory H2 database. -->

<bean id="h2-example-db" class="org.h2.jdbcx.JdbcDataSource">

<property name="URL" value="jdbc:h2:tcp://localhost/mem:ExampleDb" />

<property name="user" value="sa" />

</bean>

<bean abstract="true" id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">

<!-- Set to true to enable distributed class loading for examples, default is false. -->

<property name="peerClassLoadingEnabled" value="true"/>

<!-- Enable task execution events for examples. -->

<property name="includeEventTypes">

<list>

<!--Task execution events-->

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_STARTED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_FINISHED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_FAILED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_TIMEDOUT"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_SESSION\_ATTR\_SET"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_REDUCED"/>

<!--Cache events-->

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_PUT"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_READ"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_REMOVED"/>

</list>

</property>

<!-- Explicitly configure TCP discovery SPI to provide list of initial nodes. -->

<property name="discoverySpi">

<bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">

<property name="ipFinder">

<!--

Ignite provides several options for automatic discovery that can be used

instead os static IP based discovery. For information on all options refer

to our documentation: http://apacheignite.readme.io/docs/cluster-config

-->

<!-- Uncomment static IP finder to enable static-based discovery of initial nodes. -->

<!--<bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">-->

<bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">

<property name="addresses">

<list>

<!-- In distributed environment, replace with actual host IP address. -->

<value>127.0.0.1:47500..47509</value>

</list>

</property>

</bean>

</property>

</bean>

</property>

</bean>

</beans>

在这段代码中，我们配置了另一个Ignite节点的地址，只要当前Ignite能够发现对方Ignite节点，这个Ignite就可以加入到整个Ignite集群中。

此外，我们需要将BPMN存入Ignite中：

Ignite ignite = Ignition.ignite();

IgniteCache<Integer, String> cache = ignite.cache(“Template”);

cache.put(templateId, bPMNString);

这段代码中的bPMNString就是BPMN由/template post接口传入的BPMN文件，templateId通过调用编码解码包中的md5类的加密函数来生成。

该模块通过一下代码实现对外开放rest接口：

public class TemplateRest {

@Path("/template")

@POST

@Produces(APPLICATION\_JSON\_TYPE)

public String creatTemplate() {

//此处省略的代码为上面Ignite的实现代码

}

}

在这个模块中，我们需要通过如下方式启动Spring环境：

@SpringBootApplication

public class UarpApplication {

public static void main(String[] args) {

SpringApplication.run(UarpApplication.class, args);

}

}

最后需要在application.yml的配置文件中配置Eureka的路径，以供该模块注册和发现其他节点：

server:

port: 8761

eureka:

client:

registerWithEureka: false

fetchRegistry: false

server:

waitTimeInMsWhenSyncEmpty: 0

这里需要根据具体的生产和部署环境中Eureka的地址添加。

### 4.2.2 任务实例模块

该模块同样依赖Spring Boot和Spring Cloud为核心框架，使用Ignite作为存放task的地方，因此给出pom依赖为：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>chenhaoyuan</groupId>

<artifactId>uarp</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>uarp-task</name>

<description>task service for uarp </description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.3.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

在这个模块中，我们需要连接Ignite并将任务实例存入其中。因此，需要Ignite的配置文件ignite.xml如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:util="http://www.springframework.org/schema/util"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd">

<!-- Datasource for sample in-memory H2 database. -->

<bean id="h2-example-db" class="org.h2.jdbcx.JdbcDataSource">

<property name="URL" value="jdbc:h2:tcp://localhost/mem:ExampleDb" />

<property name="user" value="sa" />

</bean>

<bean abstract="true" id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">

<!-- Set to true to enable distributed class loading for examples, default is false. -->

<property name="peerClassLoadingEnabled" value="true"/>

<!-- Enable task execution events for examples. -->

<property name="includeEventTypes">

<list>

<!--Task execution events-->

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_STARTED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_FINISHED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_FAILED"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_TIMEDOUT"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_SESSION\_ATTR\_SET"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_TASK\_REDUCED"/>

<!--Cache events-->

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_PUT"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_READ"/>

<util:constant static-field="org.apache.ignite.events.EventType.EVT\_CACHE\_OBJECT\_REMOVED"/>

</list>

</property>

<!-- Explicitly configure TCP discovery SPI to provide list of initial nodes. -->

<property name="discoverySpi">

<bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">

<property name="ipFinder">

<!--

Ignite provides several options for automatic discovery that can be used

instead os static IP based discovery. For information on all options refer

to our documentation: http://apacheignite.readme.io/docs/cluster-config

-->

<!-- Uncomment static IP finder to enable static-based discovery of initial nodes. -->

<!--<bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">-->

<bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">

<property name="addresses">

<list>

<!-- In distributed environment, replace with actual host IP address. -->

<value>127.0.0.1:47500..47509</value>

</list>

</property>

</bean>

</property>

</bean>

</property>

</bean>

</beans>

和任务模板模块一样，Ignite在这里可以发现到整个集群，并且他和任务模板模块的集群相同。

此外，我们需要将BPMN存入Ignite中：

Ignite ignite = Ignition.ignite();

IgniteCache<Integer, String> cache = ignite.cache(“Task”);

cache.put(taskId, taskString);

这段代码中的taskString就是Task类通过实现toString()的方法来生成。Task类是根据用户从接口处传入的环境变量生成的。

该模块通过一下代码实现对外开放rest接口：

public class TemplateRest {

@Path("/template")

@POST

@Produces(APPLICATION\_JSON\_TYPE)

public String creatTemplate() {

//此处省略的代码为上面Ignite的实现代码

}

}

由于任务实例模块和任务模板模块在实现的逻辑上极为相似，所以，Spring的启动代码和配置文件就不再罗列了。

### 4.2.3 BPMN引擎模块

在这个模块中，我们还是和前面一样在Jersey创造的web容器中启动，运行环境由Spring Boot提供。在这里，我们使用的核心部分是开源的BPMN引擎：Camunda。

camunda BPM平台是工作流和过程自动化的灵活框架。它的核心是在Java虚拟机中运行的本机BPMN 2.0进程引擎。 它可以嵌入到任何Java应用程序和任何运行时容器中。它与Java EE 6集成，是Spring框架的完美匹配。在流程引擎之上，我们可以从一堆工具中进行人力工作流管理，操作和监控。

camunda BPM平台提供了一组丰富的组件围绕BPM生命周期。

流程实现和执行：

* camunda引擎 - 负责执行BPMN 2.0进程的核心组件；
* REST API - REST API提供对运行进程的远程访问；
* Spring，CDI集成 - 编程模型集成，允许开发人员编写与运行进程交互的Java应用程序。

流程设计：

* camunda建模者 - 一个独立的桌面应用程序，允许业务用户和开发人员设计和配置流程；
* camunda周期 - 启用参与项目的业务和IT方之间的基于BPMN 2.0的往返。 允许使用任何带有camunda BPM的BPMN 2.0建模工具。

流程操作：

* camunda引擎 - JMX和高级运行时集装箱集成用于过程引擎监控；
* camunda驾驶舱 - 用于过程操作的Web应用程序工具；
* camunda admin - 用于管理用户，组和访问权限的Web应用程序。

为了能够在项目中使用camunda引擎，我们需要在maven的配置中增加camunda仓库的配置：

<profiles>

<profile>

<id>camunda-bpm</id>

<repositories>

<repository>

<id>camunda-bpm-nexus</id>

<name>camunda-bpm-nexus</name>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

<url>https://app.camunda.com/nexus/content/groups/public</url>

</repository>

</repositories>

</profile>

</profiles>

<activeProfiles>

<activeProfile>camunda-bpm</activeProfile>

</activeProfiles>

之后在pom文件中添加：

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.camunda.bpm.getstarted</groupId>

<artifactId>loanapproval-spring</artifactId>

<version>0.1.0-SNAPSHOT</version>

<packaging>war</packaging>

<properties>

<camunda.version>7.6.0</camunda.version>

<spring.version>3.1.2.RELEASE</spring.version>

</properties>

<!-- import Camunda BOM to ensure correct versions of Camunda projects -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.camunda.bpm</groupId>

<artifactId>camunda-bom</artifactId>

<version>${camunda.version}</version>

<scope>import</scope>

<type>pom</type>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.camunda.bpm</groupId>

<artifactId>camunda-engine</artifactId>

</dependency>

<dependency>

<groupId>org.camunda.bpm</groupId>

<artifactId>camunda-engine-spring</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.3.171</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-jdk14</artifactId>

<version>1.7.13</version>

</dependency>

</dependencies>

</project>

要想在项目中使用camunda作为BPMN引擎，则需要将camunda引擎作为Spring Bean添加到spring的配置文件中：

<bean id="dataSource" class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy">

<property name="targetDataSource">

<bean class="org.springframework.jdbc.datasource.SimpleDriverDataSource">

<property name="driverClass" value="org.h2.Driver" />

<property name="url"

value="jdbc:h2:mem:process-engine;DB\_CLOSE\_DELAY=1000" />

<property name="username" value="sa" />

<property name="password" value="" />

</bean>

</property>

</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<bean id="processEngineConfiguration" class="org.camunda.bpm.engine.spring.SpringProcessEngineConfiguration">

<property name="processEngineName" value="engine" />

<property name="dataSource" ref="dataSource" />

<property name="transactionManager" ref="transactionManager" />

<property name="databaseSchemaUpdate" value="true" />

<property name="jobExecutorActivate" value="false" />

</bean>

<bean id="processEngine" class="org.camunda.bpm.engine.spring.ProcessEngineFactoryBean">

<property name="processEngineConfiguration" ref="processEngineConfiguration" />

</bean>

<bean id="repositoryService" factory-bean="processEngine" factory-method="getRepositoryService" />

<bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService" />

<bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />

<bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />

<bean id="managementService" factory-bean="processEngine" factory-method="getManagementService" />

接下来启动项目时就可以在控制台看到camunda启动的log了。

在代码中，我们通过如下的方式来使用camunda:

public class Starter implements InitializingBean {

@Autowired

private RuntimeService runtimeService;

public void afterPropertiesSet() throws Exception {

runtimeService.startProcessInstanceByKey("loanApproval");

}

public void setRuntimeService(RuntimeService runtimeService) {

this.runtimeService = runtimeService;

}

}

而这个模块的REST接口的实现和任务模板模块基本相似，不再过多描述了。

### 4.2.4 Eureka模块

这个模块是由Spring Cloud提供的，它并不需要Ignite，但是因为是要作为Eureka模块对外提供注册发现服务的，因此需在pom文件中添加Eureka的依赖：

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Camden.BUILD-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<start-class>eurekademo.EurekaApplication</start-class>

<java.version>1.7</java.version>

<docker.image.prefix>springcloud</docker.image.prefix>

</properties>

<build>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.2.3</version>

<configuration>

<imageName>${docker.image.prefix}/${project.artifactId}</imageName>

<dockerDirectory>src/main/docker</dockerDirectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<!-- defined in spring-cloud-starter-parent pom (as documentation hint),

but needs to be repeated here -->

<configuration>

<requiresUnpack>

<dependency>

<groupId>com.netflix.eureka</groupId>

<artifactId>eureka-core</artifactId>

</dependency>

<dependency>

<groupId>com.netflix.eureka</groupId>

<artifactId>eureka-client</artifactId>

</dependency>

</requiresUnpack>

</configuration>

</plugin>

<plugin>

<groupId>pl.project13.maven</groupId>

<artifactId>git-commit-id-plugin</artifactId>

<configuration>

<failOnNoGitDirectory>false</failOnNoGitDirectory>

</configuration>

</plugin>

<plugin>

<!--skip deploy (this is just a test module) -->

<artifactId>maven-deploy-plugin</artifactId>

<configuration>

<skip>true</skip>

</configuration>

</plugin>

</plugins>

</build>

启动该模块只需要启动Spring Boot，Eureka会在Spring Boot启动的时候跟随启动：

@SpringBootApplication

@EnableEurekaServer

public class EurekaApplication {

public static void main(String[] args) throws Exception {

SpringApplication.run(EurekaApplication.class, args);

}

}

### 4.2.5 Ignite模块

最后是关于Ignite模块的实现，由于Ignite本身就提供了单独启动的版本，因此这部分的代码不需要我们来写，我们只需要配置好就可以了。但是由于最终部署在docker集群上并使用kubernetes作为虚拟网关，因此，Ignite自带的组播功能就无法使用了，kubernetes虽然没有提供组播的功能，但是其提供了一个DNS发现其他节点的功能，因此我们需要重新实现Ignite节点发现的类。代码如下：

public class KubernetesPodIpFinder extends TcpDiscoveryVmIpFinder {

private static final int DNS\_LOOKUP\_TIMEOUT\_MILLIS = 1000;

/\*\*

\* Grid logger.

\*/

@LoggerResource

private IgniteLogger log;

@GridToStringInclude

private String containerPortName;

@GridToStringInclude

private String serviceName;

public KubernetesPodIpFinder() {

super(true);

}

public KubernetesPodIpFinder(boolean shared) {

super(shared);

}

/\*\*

\* Parses provided container port name.

\*

\* @param containerPortName the container port name to use in lookup queries.

\* @throws IgniteSpiException

\*/

@IgniteSpiConfiguration(optional = true)

public synchronized void setContainerPortName(String containerPortName) throws IgniteSpiException {

this.containerPortName = containerPortName;

}

/\*\*

\* Parses provided service name.

\*

\* @param serviceName the service name to use in lookup queries.

\* @throws IgniteSpiException

\*/

@IgniteSpiConfiguration(optional = true)

public synchronized void setServiceName(String serviceName) throws IgniteSpiException {

this.serviceName = serviceName;

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public synchronized Collection<InetSocketAddress> getRegisteredAddresses() {

// provision DNS resolver for usage with SRV records

final DnsSrvResolver resolver = DnsSrvResolvers.newBuilder()

.retainingDataOnFailures(true)

.dnsLookupTimeoutMillis(DNS\_LOOKUP\_TIMEOUT\_MILLIS)

.build();

// resolve configured addresses

final Collection<InetSocketAddress> inets = new CopyOnWriteArrayList<>();

final String fqdn = "\_" + containerPortName + ".\_tcp." + serviceName;

log.debug("Looking up SRV records with FQDN [" + fqdn + "].");

final List<LookupResult> nodes = resolver.resolve(fqdn);

for (LookupResult node : nodes) {

inets.add(InetSocketAddress.createUnresolved(node.host(), node.port()));

}

log.debug("Found " + nodes.size() + " nodes.");

return inets;

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public String toString() {

return S.toString(KubernetesPodIpFinder.class, this, "super", super.toString());

}

}

## 4.3 测试

### 4.3.1 单元测试

单元测试一般是由开发人员来完成的，在我们的系统中，单元测试主要是通过Junit和SpringBootTest来完成，Junit可以提供测试时常用的断言、多用例测试，而SpringBootTest可以提供单个类运行中的Spring的运行环境。

但是由于Spring运行时环境的复杂性和项目的很多依赖，并不能保证SpringBootTest能够提供足够的测试环境的支持，对于SpringBootTest无法提供的被反射的类，我们通过mockito来模拟那些类，来实现单元测试的解耦。

在本项目中单元测试的代码覆盖率为90%。

### 4.3.2 集成测试

集成测试时，我们通过一个测试类将每个服务都启动一个实例，然后对每个服务开放的rest接口进行测试，能够以预期的步骤运行则认为测试通过。

# 第五章 工作总结与展望

## 5.1 工作总结

本文通过分析运营商所面临着的大数据问题开始，引出了统一分析运行平台。之后通过对统一分析运行平台的需求分析，明确了该平台所要达到的目标，然后根据需求对系统进行了架构设计，最后根据系统的架构设计对系统进行了实现并给出了实现过程中所遇到的问题的解决方案。

论文通过以这样一个方式展示了微服务开发的流程和遇到问题时处理问题的方法，最终，得出一个具有的高性能、高容错、高安全性的统一分析运行平台，解决了运营商对大量业务流程的编排和调度的难题。

## 5.2 工作展望

本文实现了统一分析运行平台的运行，但是，该平台目前依然有很大的改进空间。例如：

* 在本文中，选择哪个微服务实体作为将要调用的服务采用了随机选择的方式，其实这样的方式并不能很好地对所有的微服务实体都做到负载均衡；
* 目前外部可以通过任务实例模块查询的状态只有开始与否、结束与否这样简单的应答，将来会增加对任务实例的详细状态的查询，在执行业务流程时可以直接返回一个具体的执行进度。
* 最后，统一分析运行平台的业务流程引擎模块采用了开源的引擎，这对于商业公司而言在引擎问题时可能并不能很好地解决，因此，之后可能会开发自己的流程编排引擎。

# 致 谢

值此论文完成之际，谨向所有关心和支持我的人们致以诚挚的谢意！

首先，我要衷心地感谢我的校内导师刘奇其教授和企业导师薄宏剑工程师。从论文选题、内容和整体结构的确定，到直至最后定稿，两位导师都以极其负责的态度给予悉心指导，为我提出了许多宝贵的意见和建议，使我获益良多。导师有严肃的科学态度，严谨的治学精神和精益求精的工作作风，这些都是我所需要学习的，感谢导师给予了我这样一个学习机会，谢谢!在此，谨向导师致以最诚挚的感谢！此外，还要感谢在企业实习时团队的成员。没有他们的帮助和共同努力，就没有项目的圆满成功，也就不会有本文的形成。在此，向他们表示衷心的感谢!

# 参考文献

1. 陈春霞. 基于容器的微服务架构的浅析[J]. 信息系统工程, 2016 (3): 95-96.
2. Fielding R T. Architectural styles and the design of network-based software architectures[D]. University of California, Irvine, 2000.
3. 唐文宇. 面向 SOA 架构微服务的安全系统的设计与实现[D]. 南京大学, 2016.
4. Abrams S, Kunze J, Loy D. An emergent micro-services approach to digital curation infrastructure[J]. International Journal of Digital Curation, 2010, 5(1): 172-186.
5. Namiot D, Sneps-Sneppe M. On micro-services architecture[J]. International Journal of Open Information Technologies, 2014, 2(9).
6. 徐鹏, 陈思, 苏森. 互联网应用 PaaS 平台体系结构[J]. 北京邮电大学学报, 2012, 35(1): 120-124.
7. Newman S. Building microservices[M]. " O'Reilly Media, Inc.", 2015.
8. 王克苑, 张维勇, 王建新. SSL 安全性分析研究[J]. 合肥工业大学学报: 自然科学版, 2004, 27(1): 87-91.
9. 王珊, 王会举, 覃雄派, 等. 架构大数据: 挑战, 现状与展望[J]. 计算机学报, 2011, 34(10): 1741-1752.
10. Brooks F P, Li Q. 人月神话[M]. 中国电力出版社, 2003.