

题 目 一种基于微服务架构的PAAS平台

\_\_\_\_\_\_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\_\_\_\_\_\_\_\_\_\_\_刘其奇\_\_\_\_\_\_\_\_\_\_\_\_\_\_

企业导师\_\_\_\_\_\_\_\_\_\_\_薄宏剑\_\_\_\_\_\_\_\_\_\_\_\_\_\_

起止日期\_\_\_\_\_2017-01-20至2017-05-20\_\_\_\_

设计地点\_\_\_\_\_\_\_\_\_\_\_南京\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 

**一种基于微服务架构的PAAS平台**

**摘要**

伴随电信行业3G/4G等业务的增加，数据量出现了爆炸式的增长，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等分布式大数据技术。由于电信运营商自身结构复杂，产品和业务繁多，电信用户多，每天能产生大量的数据，因此产生了很多不同的hadoop/spark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。

华为大数据部门决定开发一个PAAS平台用来统一管理和调度hadoop/spark任务（以后称之为统一运行平台/Runtime-Platform/RP）。本文结合我所参与的华为大数据统一管理平台的经验，详细介绍了华为这个PAAS平台的设计。该项目采用了分布式的开发理念，选择了较为流行的微服务架构，并且使用轻量级的docker容器单独部署服务，因此，该项目具有极高的可用性和鲁棒性。

**关键词**：微服务；分布式；PAAS；大数据；docker

**Abstract**

With the telecommunications industry 3G / 4G and other business increases, the amount of data appeared 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 large distributed data technology. Because telecom operators are complex in their own structure, many products and services, many telecom users can produce a lot of data every day, resulting in a lot of different hadoop / spark needs, how to manage and schedule a large number of big data tasks has become a runner A headache for one thing.

Huawei Big Data Department decided to develop a PAAS platform for unified management and scheduling hadoop / spark task (later called the unified operating platform / Runtime-Platform / RP). This paper introduces the experience of Huawei's PAAS platform in combination with the experience of Huawei's large data unified management platform. The project uses a distributed development concept, selected a more popular micro service architecture, and the use of lightweight docker containers to deploy services alone, so the project has a high availability and robustness.

**Keywords**: Micro service; distributed; PAAS; big data; docker

目 录

[**一种基于微服务架构的PAAS平台** 1](#_Toc484204042)

[**摘要** 1](#_Toc484204043)

[**Abstract** 2](#_Toc484204044)

[第一章 绪论 5](#_Toc484204045)

[1.1引言 5](#_Toc484204046)

[1.2目前现状 5](#_Toc484204047)

[1.3软件架构发展现状 6](#_Toc484204048)

[1.4本文的研究目的和主要研究内容 6](#_Toc484204049)

[第二章 应用到的技术及原理 7](#_Toc484204050)

[2.1微服务 7](#_Toc484204051)

[2.1.1**概述** 7](#_Toc484204052)

[2.1.2**微服务的好处** 7](#_Toc484204053)

[2.1.3**微服务与SOA的比较** 8](#_Toc484204054)

[2.2远程调用过程（RPC）与具象状态传输（REST） 8](#_Toc484204055)

[2.2.1**RPC** 8](#_Toc484204056)

[2.2.2**REST** 9](#_Toc484204057)

[2.2.2**REST和RPC的选择** 10](#_Toc484204058)

[2.3业务流程模型和标记法BPMN 11](#_Toc484204059)

[2.3.1**概述** 11](#_Toc484204060)

[2.3.1**要素** 11](#_Toc484204061)

[2.3.2**BPMN示例** 12](#_Toc484204062)

[2.4Docker 13](#_Toc484204063)

[**概述** 13](#_Toc484204064)

[第三章 所使用的框架 15](#_Toc484204065)

[3.1Spring Boot 15](#_Toc484204066)

[3.1.1**概述** 15](#_Toc484204067)

[3.1.2**Spring Framework** 15](#_Toc484204068)

[3.1.3如何使用**Spring Boot** 15](#_Toc484204069)

[3.1.4**Spring Boot进阶使用** 18](#_Toc484204070)

[3.2Spring Cloud 20](#_Toc484204071)

[3.2.1**概述** 20](#_Toc484204072)

[3.2.2Spring Cloud的**主要项目** 20](#_Toc484204073)

[3.2.2如何使用Spring Cloud 21](#_Toc484204074)

[3.3camunda 24](#_Toc484204075)

[3.3.1**概述** 24](#_Toc484204076)

[3.3.2camunda**组件** 24](#_Toc484204077)

[3.3.3**使用camunda** 24](#_Toc484204078)

[3.4apache ignite 30](#_Toc484204079)

[3.4.1**概述** 30](#_Toc484204080)

[3.4.2**特点** 30](#_Toc484204081)

[3.4.3**使用apache ignite** 33](#_Toc484204082)

[第四章 具体实现 35](#_Toc484204083)

[4.1系统的模块划分 35](#_Toc484204084)

[4.1.1任务模板模块 35](#_Toc484204085)

[4.1.2任务实例模块 35](#_Toc484204086)

[4.1.3 BPMN引擎模块 36](#_Toc484204087)

[4.1.4 Ignite模块 37](#_Toc484204088)

[4.1.5 Eureka模块 37](#_Toc484204089)

[4.2模块的交互 37](#_Toc484204090)

[4.3任务流程 37](#_Toc484204091)

[4.4安全性 38](#_Toc484204092)

[4.5路由 38](#_Toc484204093)

[4.6测试 39](#_Toc484204094)

[4.6.1 单元测试 39](#_Toc484204095)

[4.6.2集成测试 39](#_Toc484204096)

[4.7开发和部署流程 39](#_Toc484204097)

[4.8开发和部署环境 39](#_Toc484204098)

[致谢 39](#_Toc484204099)

[参考文献： 41](#_Toc484204100)

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任务的调度问题，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目前现状

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：①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。②海量电信数据烟囱式分布，难以整合。③数据管理分散，难以共享。④数据获取过程繁琐，开发困难，对专家依赖高。具体来说：①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。②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。③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。④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。

1.4本文的研究目的和主要研究内容

本文主要研究UARP的几个核心模块的设计，以及如何实现微服务，这样的微服务有怎样的优缺点。

1. 应用到的技术及原理

2.1微服务

2.1.1**概述**

微服务就是一些协同工作的小而自治的服务：

1. 小，专注于一件事：随着业务的不断增加，代码库也不断地增加，久而久之，代码量会变得非常庞大，以至于想要修改一个小地方会变得极其困难。尽管我们做了清晰的模块化，但事实上模块的界限会便得越来越模糊，最后难以维护，相似功能的代码可能在代码库中不断地出现。Robert C. Martin提出了单一职责原则：“把因为相同原因变化的东西聚集到一起，把因为不同原因而变化的东西分离出来。”微服务的理念很好地应用到了这个内聚性的观点，微服务根据业务边界来确认服务的边界，这样每段代码都有自己合适的去处。Jon Eaves提出，一个微服务应该可以在两周的时间内完全重写。使用的服务越小，能够带来的独立的好处就越多，但是大量的服务也会使管理变得复杂。
2. 自治性：服务之间通过网络进行通信，从而加强了服务间的独立性，避免了耦合。对于一个服务，我们关注于暴露什么和隐藏什么，如果暴露太多东西会带来耦合性。

2.1.2**微服务的好处**

微服务的好处：

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。

2.1.3**微服务与SOA的比较**

面向服务架构(Server-Oriented Architecture)缩写为SOA，这是一种前几年很流行的架构，它与微服务之间有着很多的相似点。SOA最先开始提出了服务间通过网络进行通信。在单块系统的开发过程中，代码总会不自觉地渐渐耦合在一起，虽然可以通过架构师良好的规划和对开发人员严格的约束来避免这一现象，但是，由于代码的界限并不是严格明确的，导致边界处或多或少会出现冗余的代码。后来人们发现，通过网络进行调用会使得模块的边界便得非常地明确，而且模块间只需要维持外部接口不变，内部可以完全透明起来。SOA就是一个这样的架构，听起来他和微服务的实现方式完全相同，但是他们之间还是存在一些不同之处的：

1. SOA的服务大小并没有明确的规定，一般公司会使用较大粒度的方式按照业务分割服务，而微服务强调小而微的概念，相比SOA，微服务会将业务按照更小的粒度拆分为多个服务，而且微服务一般会使得单个服务能够在两周内重写。
2. ESB总线：正如之前所说，SOA一般按照业务的范围来划分服务，因此他会采用“总线”的方式来管理和规划服务之间的联系，在微服务中，单个业务可能由多个服务组成，因此并没有在业务级别的总线，但是还是会有类似注册中心的实现。

综上所述，微服务可能更像是对SOA的一种，但是它有着很严格和细致的规则。

2.2远程调用过程（RPC）与具象状态传输（REST）

2.2.1**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。

2.2.2**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：

1. 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。
2. 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。
3. 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，例如我们请求一个网络上的资源



图1 http请求

1. 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。
2. 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。

2.2.2**REST和RPC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。

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。

2.3业务流程模型和标记法BPMN

2.3.1**概述**

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。

2.3.1**要素**

BPMN通过图形来进行建模，它所用到的图形很少，但是可以很好地提供一个规范：

1. 流对象：流对象包括事件、活动、网关

事件由圆环表示，一般分为开始事件、结束事件、中间事件。

开始事件是流程的触发事件，只能用于捕获；

结束事件是流程结果的表现，只能用于抛出；

中间事件发生在开始事件和结束事件之间，可以是抛出或捕获。

https://upload.wikimedia.org/wikipedia/commons/9/99/BPMN-Event.jpg

图2 流对象

活动用圆角的矩形来表示，分为任务、子流程和事务。

其中事务是子流程的一种形式，事务所表示的活动需当作一个整体，任何一个其中的活动失败后就必须全部撤回。

https://upload.wikimedia.org/wikipedia/commons/0/06/BPMN-Activity.jpg

图3 活动

关口用菱形表示，用来表达条件、合并或者分流。



图4 关口

1. 连接：顺序流、消息流、关联

顺序流是默认的条件流，消息流用来表示跨过组织边界的消息，关联用来表示器物和文本流的关系，关联不需要标明方向。

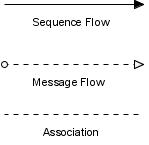


图5 连接

1. 泳道：池和道

泳道是对活动组织和分类事物，其中池表示主要参与者，道则是按照职能的归类。

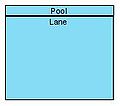


图6 泳道

1. 器物：数据对象、组、注释

这些对象相互组合就可以形成一个BPMN图。

2.3.2**BPMN示例**

**2.3.2.1简单的饥饿过程**

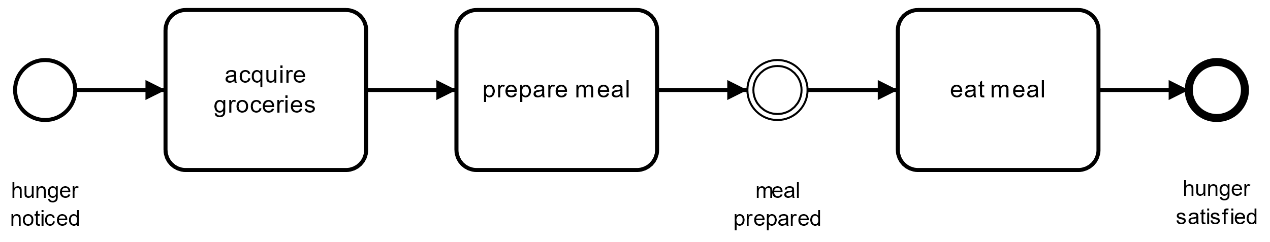


图7 饥饿过程的BPMN图

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。

BPMN并没有规定BPMN图的绘画方向，但是通常来说会按照从左到右的顺序来画BPMN图。

**2.3.2.1较为复杂的装货过程**

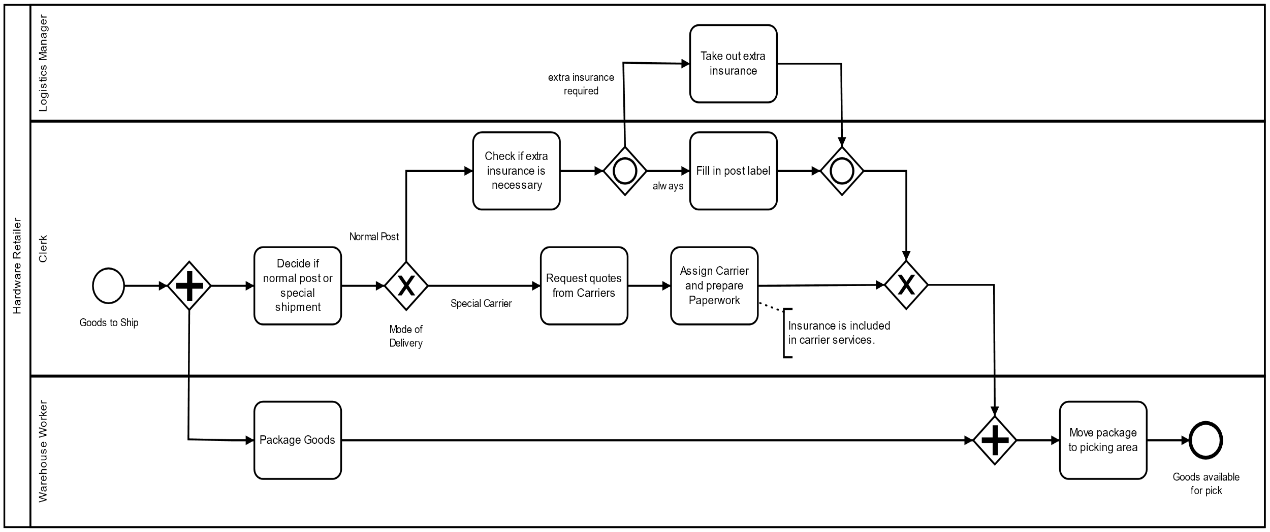


图8 装货过程的BPMN图

在上面这张图中你可以看到一个硬件零售商在用户订购的货物可以安全送达前必须做的准备步骤。

在这个例子中，我们为这个过程中涉及到的人们使用了一个池和不同的道，在这里自动意味着我们消除了人与人之间的沟通：我们假设他们以某种方式相互通信。如果我们有一个流程引擎来驱动这个过程，那么这个引擎会分配用户任务并且负责这些人之间的沟通。如果我们没有这样的流程引擎，但是想要明确地介绍所涉及到的人之间的沟通，我们就需要使用协作图。

2.4Docker

**概述**

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。

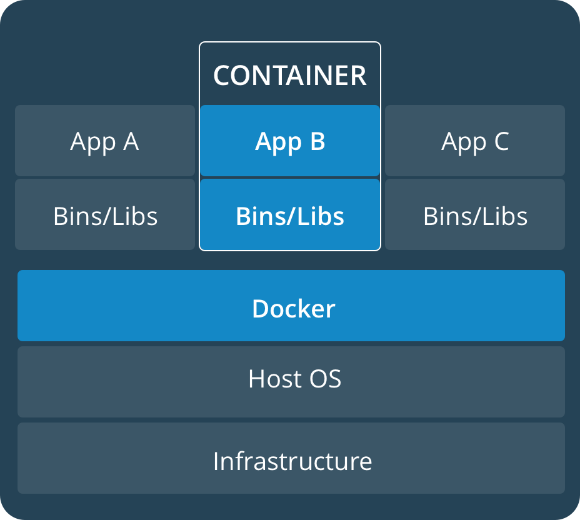
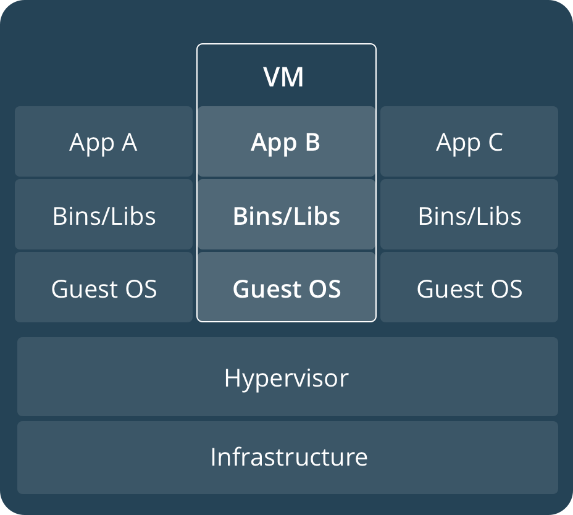
 

图9 docker与vm结构比较

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。

1. 所使用的框架

3.1Spring Boot

3.1.1**概述**

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：

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

3.1.2**Spring Framework**

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：

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

3.1.3如何使用**Spring Boot**

首先可以在spring官网上直接生成项目的初始文件，然后通过IDE导入即可。下图是通过自动生成的项目：

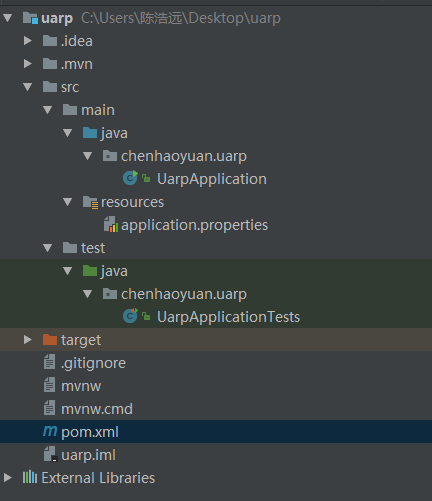


图10 Spring Boot项目结构

其中包含pom文件、一个基础的类、这个类对应的测试文件。

Pom文件中已经解决了项目中spring boot的依赖：

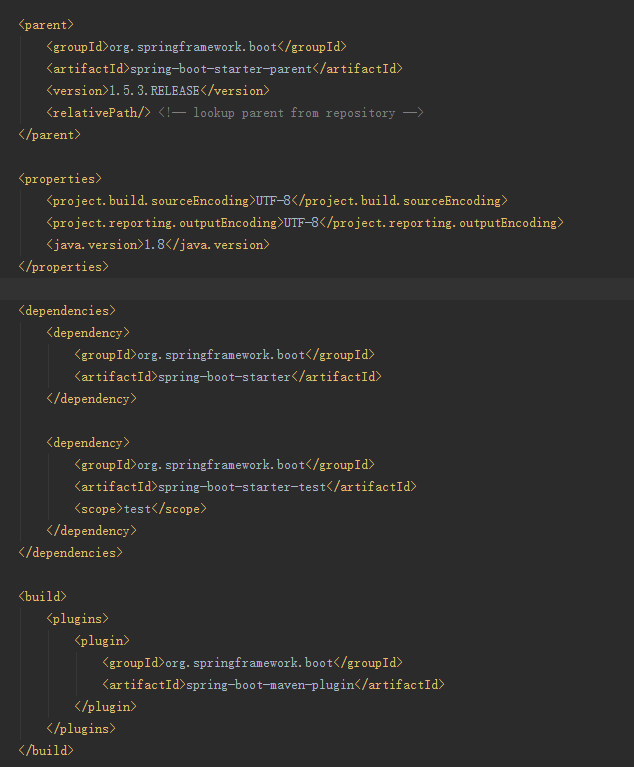


图11 Spring Boot 项目依赖

基础类中提供了最简单的启动main方法：

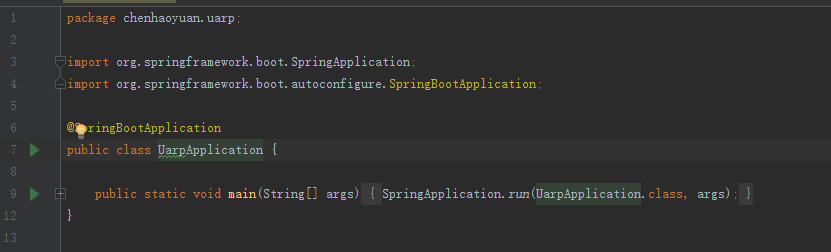


图12 Spring Boot启动代码

而测试类中自动引入了测试所需要的Junit和SpringBootTest：

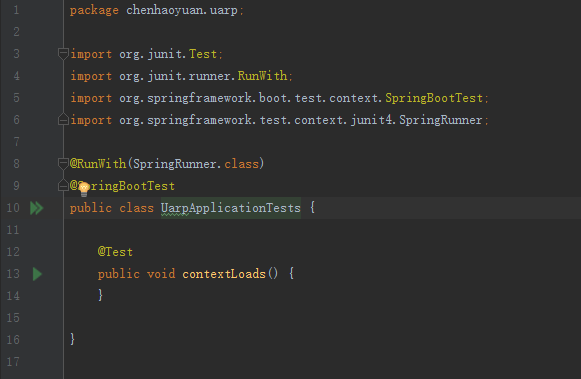


图13 Spring Boot测试代码

3.1.4**Spring Boot进阶使用**

为了完成应用程序，我们需要创建一个单独的Java文件。Maven默认会编译src/main/java下的源码，所以你需要创建那样的文件结构，并添加一个名为src/main/java/Example.java的文件：

import org.springframework.boot.\*;

import org.springframework.boot.autoconfigure.\*;

import org.springframework.stereotype.\*;

import org.springframework.web.bind.annotation.\*;

@RestController

@EnableAutoConfiguration

public class Example {

@RequestMapping("/")

String home() {

return "Hello World!";

}

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

SpringApplication.run(Example.class, args);

}

}

@RestController和@RequestMapping注解：Example类上使用的第一个注解是@RestController，这被称为构造型（stereotype）注解。它为阅读代码的人提供暗示（这是一个支持REST的控制器），对于Spring，该类扮演了一个特殊角色。在本示例中，我们的类是一个web @Controller，所以当web请求进来时，Spring会考虑是否使用它来处理。

@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。@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。

注：@RestController和@RequestMapping是Spring MVC中的注解（它们不是Spring Boot的特定部分），具体参考Spring文档的MVC章节。

@EnableAutoConfiguration注解

第二个类级别的注解是@EnableAutoConfiguration，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，并对Spring进行相应地设置。

Starters和Auto-Configuration：Auto-configuration设计成可以跟"Starters"一起很好的使用，但这两个概念没有直接的联系。你可以自由地挑选starters以外的jar依赖，Spring Boot仍会尽最大努力去自动配置你的应用。

配置Spring Boot：Spring Boot提倡基于Java的配置。尽管你可以使用XML源调用SpringApplication.run()，不过还是建议你使用@Configuration类作为主要配置源。通常定义了main方法的类也是使用@Configuration注解的一个很好的替补。虽然网络上有很多使用XML配置的Spring示例，但你应该尽可能的使用基于Java的配置，搜索查看enable\*注解就是一个好的开端。自动配置（Auto-configuration）是非侵入性的，任何时候你都可以定义自己的配置类来替换自动配置的特定部分。例如，如果你添加自己的DataSource bean，默认的内嵌数据库支持将不被考虑。

依赖注入：你可以自由地使用任何标准的Spring框架技术去定义beans和它们注入的依赖。简单起见，我们经常使用@ComponentScan注解搜索beans，并结合@Autowired构造器注入。

如果遵循以上的建议组织代码结构（将应用的main类放到包的最上层，即root package），那么你就可以添加@ComponentScan注解而不需要任何参数，所有应用组件（@Component, @Service, @Repository, @Controller等）都会自动注册成Spring Beans。

3.2Spring Cloud

3.2.1**概述**

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。

3.2.2Spring Cloud的**主要项目**

1. Spring Cloud Config:由git仓库支持的集中式外部配置管理。 配置资源直接映射到Spring`Environment`，但如果需要，可以使用非Spring应用程序。
2. Spring Cloud Bus: 用于将服务和服务实例以及分布式消息传递链接的事件总线。 用于在集群中传播状态更改（例如，配置更改事件）。
3. Spring Cloud for Cloud Foundry: 将应用程序与Pivotal Cloudfoundry集成。 提供服务发现实现，并且还可以轻松实现SSO和OAuth2保护的资源，还可以创建Cloudfoundry服务代理。
4. Spring Cloud Cloud Foundry Service Broker: 提供构建管理Cloud Foundry管理服务的服务代理的起点。
5. Spring Cloud Cluster: Zookeeper，Redis，Hazelcast，Consul的抽象和实现的选举模式和常见的有的状态模式。
6. Spring Cloud Consul：控制总线和配置。智能路由（Zuul）和客户端负载平衡（Ribbon），断路器（Hystrix）通过与Spring Cloud Netflix的集成提供。
7. Spring Cloud Security：基于Spring Boot和Spring安全性OAuth2，我们可以快速创建实现常见模式的系统，如单点登录，令牌中继和令牌交换。
8. Spring Cloud Sleuth：适用于Spring Cloud应用程序的分布式跟踪，与Zipkin，HTrace和基于对象（例如ELK）跟踪兼容。
9. Spring Cloud Data Flow：用于现代运行时可组合的微服务应用程序的云本地编排服务。 易于使用的DSL，拖放式GUI和REST API一起简化了基于微服务的数据管道的整体编排。
10. Spring Cloud Stream：一个轻量级的事件驱动的微服务框架来快速构建可以连接到外部系统的应用程序。 使用Apache Kafka或RabbitMQ在Spring Boot应用程序之间发送和接收消息的简单声明模型。
11. Spring Cloud Stream App Starters：Spring Cloud Stream应用程序启动器是基于Spring Boot的Spring集成应用程序，提供与外部系统的集成。
12. Spring Cloud Task：一种短期的微服务框架，用于快速构建执行有限数据处理的应用程序。 简单的声明，将功能和非功能功能添加到Spring Boot应用程序。
13. Spring Cloud Zookeeper：使用Apache Zookeeper进行服务发现和配置管理。
14. Spring Cloud Starters：Spring Boot-style启动项目，以缓解Spring Cloud消费者的依赖管理。

3.2.2如何使用Spring Cloud

在之前Spring Boot的基础上，向POM文件中添加Spring Cloud的依赖，然后使用Maven导入这些依赖。

<parent>

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

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

<version>1.5.2.RELEASE</version>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId></groupId>

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

</dependency>

<dependency>

<groupId></groupId>

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

</dependency>

</dependencies>

对于spring cloud的使用需要在一个application.yml中进行如下配置：

server:  
 port: 8888  
  
management:  
 context-path: /admin  
   
logging:  
 level:  
 com.netflix.discovery: 'OFF'  
 org.springframework.cloud: 'DEBUG'  
   
eureka:  
 instance:  
 leaseRenewalIntervalInSeconds: 10  
 statusPageUrlPath: /admin/info  
 healthCheckUrlPath: /admin/health  
  
spring:  
 cloud:  
 config:  
 server:  
 git:  
 uri: https://github.com/spring-cloud-samples/config-repo  
 basedir: target/config  
  
---  
spring:  
 profiles: cloud  
eureka:  
 password: password  
 instance:  
 hostname: ${vcap.application.uris[0]}  
 nonSecurePort: 80  
 client:  
 serviceUrl:  
 defaultZone: ${vcap.services.${PREFIX:}eureka.credentials.uri:http://user:${eureka.password:}@${PREFIX:}eureka.${application.domain:cfapps.io}}/eureka/

并在bootstrap.yml中进行如下配置：

spring:  
 application:  
 name: configserver  
encrypt:  
 failOnError: false  
 keyStore:  
 location: classpath:keystore.jks  
 password: ${KEYSTORE\_PASSWORD:foobar} # don't use a default in production  
 alias: test

bootstrap.yml在application.yml之前加载。

它通常用于以下内容：在使用Spring Cloud Config Server时，应该在bootstrap.yml中指定spring.application.name和spring.cloud.config.server.git.uri一些加密/解密信息。从技术上讲，bootstrap.yml由父Spring ApplicationContext加载。 那个父ApplicationContext在使用application.yml的那个之前加载。

3.3camunda

3.3.1**概述**

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。

3.3.2camunda**组件**

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：

1. camunda引擎 - 负责执行BPMN 2.0进程的核心组件；
2. REST API - REST API提供对运行进程的远程访问；
3. 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：

1. 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；
2. 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：

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

3.3.3**使用camunda**

Camunda BPM可以通过maven中央仓库来使用，但用来平台开发则必须添加以camunda的公有仓库到maven的配置文件：

<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>

之后需要添加如下依赖到pom文件：

<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:

package org.camunda.bpm.getstarted.loanapproval;

import org.camunda.bpm.engine.RuntimeService;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.annotation.Autowired;

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;

}

}

最后在camunda官网可以找到camunda引擎的API。

3.4apache ignite

3.4.1**概述**

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。

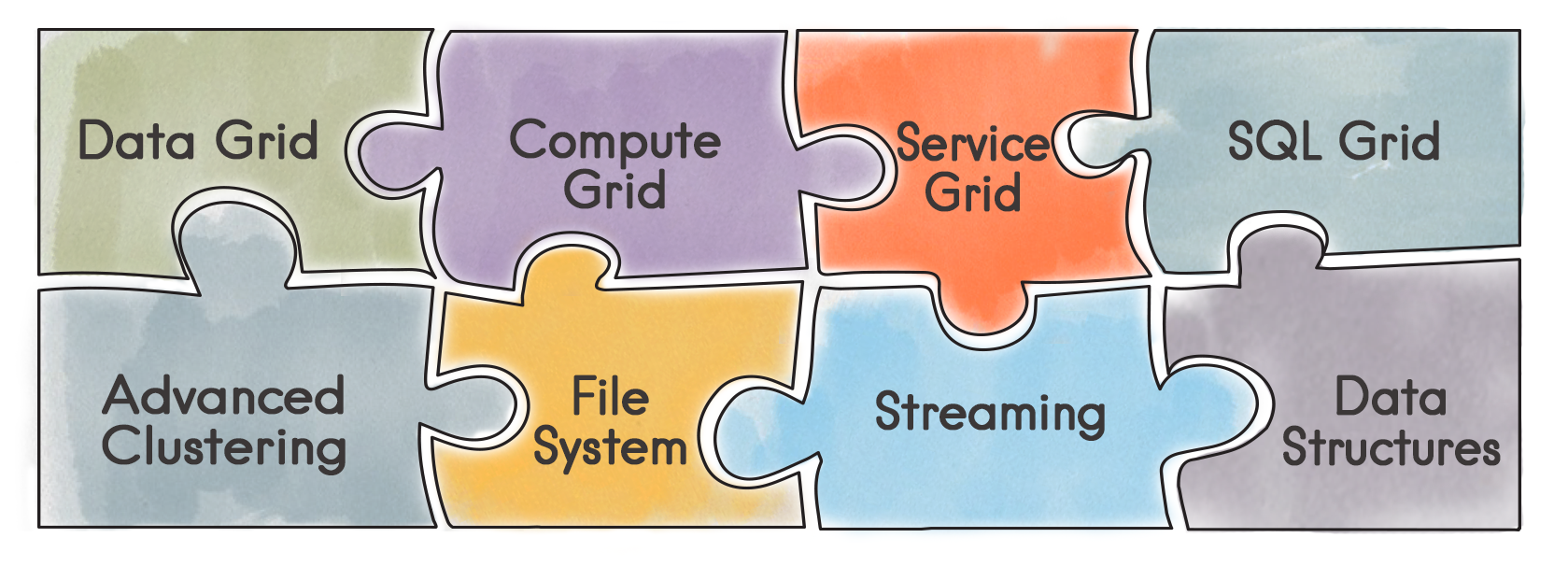


图14 apache ignite组成

Apache Ignite内存数据结构旨在为广泛的内存计算用例，从高性能计算到行业最先进的数据网格，高可用性服务网格和流式传输提供无与伦比的性能。

3.4.2**特点**

1. 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。

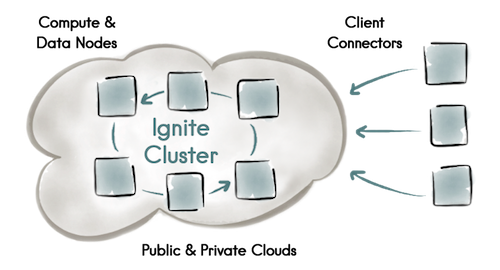


图15 apache ignite 集群

1. 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。

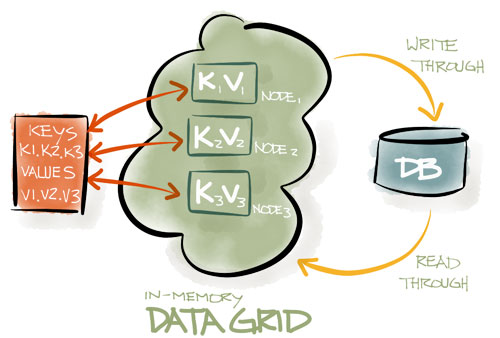


图16 apache ignite 数据网格

1. 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。

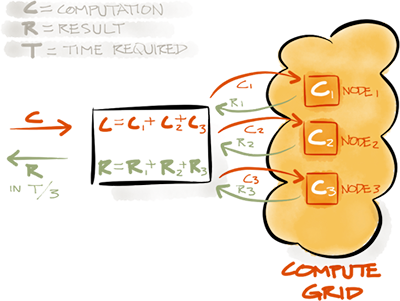


图17 apache ignite流媒体

1. 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18 apache ignite服务网格

1. 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。
2. 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。
3. 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。
4. 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。

3.4.3**使用apache ignite**

Apache ignite可以使用JAVA或其他语言来启动，也可以使用其分发的二进制版本通过命令行直接启动。Ignite可以通过配置使其作为存储节点或者普通节点，当它作为存储节点时，它即可以进行存储，也可以进行读取，而当它作为普通节点时，它就不会被用作存储，但是可以通过网络将要存储的内容存储在其他存储节点上。

当我们采用JAVA来启动它时，我们只需要向pom文件中添加：

<dependency>

<groupId>org.apache.ignite</groupId>

<artifactId>ignite-core</artifactId>

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

</dependency>

<dependency>

<groupId>org.apache.ignite</groupId>

<artifactId>ignite-spring</artifactId>

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

</dependency>

<dependency>

<groupId>org.apache.ignite</groupId>

<artifactId>ignite-indexing</artifactId>

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

</dependency>

之后就可以通过JAVA代码在系统内部启动了：

try (Ignite ignite = Ignition.start("examples/config/example-ignite.xml")) {

Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

// Iterate through all the words in the sentence and create Callable jobs.

for (final String word : "Count characters using callable".split(" "))

calls.add(word::length);

// Execute collection of Callables on the grid.

Collection<Integer> res = ignite.compute().call(calls);

// Add up all the results.

int sum = res.stream().mapToInt(Integer::intValue).sum();

System.out.println("Total number of characters is '" + sum + "'.");

}

当然，apache ignite也提供了通过JAVA bean来更好地操作ignite的接口。

1. 具体实现

4.1系统的模块划分

系统的模块划分：

系统主要分为任务模板模块、任务实例模块、BPMN引擎模块，以及需要Hadoop和Spark模块提供的接口。

4.1.1任务模板模块

或修改一个已有的模块，它会创建一个任务模板并保存到Ignite中，用以之后实例化为一个任务。任务模板是一系列小任务所编排得出的大的任务，它可能有多个Hadoop和Spark组成。

通过下面这个rest接口来创建任务模板

/template post 创建任务模板

在调用这个rest接口后，任务模板模块会收到这个rest请求，之后根据请求的内容创建一个任务模板，将任务模板存入Ignite中，如果成功创建模板，这个接口的返回类型为200 OK，返回内容为保存有任务模板id的json；如果不成功，则根据情况返回400或500错误码，并在返回内容中附加异常栈，其中400对应请求数据的格式错误，500对应服务器产生的错误。

目前的任务模板在创建时通过json中嵌入一个BPMN字段来实现，具体为：

{

taskTemplate: “BPMN”

}

在Ignite中，任务模板保持以BPMN的xml样式的形式存在。

同时这个模块也提供了：

1. /template get 获取任务模板

在这个接口中，get请求路径后需添加任务模板id作为资源，例如：

/template/id

1. /template delete 删除任务模板

在这个接口中，delete请求路径后需添加任务模板id作为资源，例如：

/template/id

如果成功删除则返回200 OK,如果任务模板不存在则返回404 NOT FOUND。

1. /templates get 获取全部的任务模板

通过这个接口，我们可以获取到全部的任务模板的id，具体返回的内容为：将存有所有任务模板id的数据转换为的json

｛

Templates: [templateID1,templateID2,等]

｝

4.1.2任务实例模块

将任务模板根据具体的需求实例化，相比任务模板，它更为具体的包括了具体的数据存放地点、输出地点、执行时间、是否循环等信息。

它通过下面这个接口来创建任务实例：

/task post 创建任务实例

通过对这个rest接口的请求体格式为：

｛

templateID: ”ID”,

local: “local”,

time: “time”,

cycles：“number”

｝

如果这个请求成功创建任务实例，则返回状态码200 OK，在返回的response体中以json的形式封装任务实例id，且对应在Ignite中存储任务实例，之后这个模块会自动调用BPMN引擎模块，并将任务实例id传给BPMN引擎。

这个模块同样提供了：

1. /task get 获取任务实例

在这个接口中，get请求路径后需添加任务实例id作为资源，例如：

/task/id

1. /tasks get 获取全部的任务实例

通过这个接口，我们可以获取到全部的任务实例的id，具体返回的内容为：将存有所有任务实例id的数据转换为的json

｛

tasks: [taskID1,taskID2,等]

｝

注意一点是，目前当前的平台并未提供删除已经运行中的任务实例的功能，但是这个功能会在以后的版本中逐渐增加。

4.1.3 BPMN引擎模块

这部分主要是将从任务实例模块接收到的任务实例按照其参数放入BPMN引擎中运行。

这部分的实现主要是通过开源的camunda引擎来实现，目前提供了两个rest接口：

1. /bpmn get

在这个接口中，get请求路径后需添加任务实例id作为资源，例如：

/bpmn/taskID

调用这个接口后，任务实例进入BPMN引擎运行，运行过程由camunda引擎调用专门写的JAVA类来实现。

1. /bpmn/state get

在这个接口中，get请求路径后需添加任务实例id作为资源，例如：

/bpmn/state/taskID

通过这个接口可以返回BPMN在执行单个任务时的状态，状态包括：运行时间、任务进度，以json的形式封装并返回。

对于要执行的hadoop/spark任务，需要开发人员自行编写一个JAVA类并开放rest接口，在BPMN引擎中调用的java类中通过rest客户端去调用hadoop/spark的rest接口。

4.1.4 Ignite模块

这个模块用于充当数据库进行存储和共享数据，在开发过程中随其他模块一起启动，开发结束后单独分离出来。

Ignite提供了通过JAVA启动的版本和单独启动的版本，同时Ignite分为客户端和存储端两种模式，在启用客户端模式时的ignite节点并不会存储数据，在开发过程中，ignite都启用存储模式，但在开发结束后，我们将ignite从系统中分离出来并通过docker部署，系统中仅保留ignite的客户端作为与ignite交互的接口，而在docker中部署的ignite作为存贮的集群。

4.1.5 Eureka模块

Eureka是Netflix服务发现的一种服务和客户端。这种服务是可以被高可用性配置的和部署,并且在注册的服务当中，每个服务的状态可以互相复制给彼此。Eureka模块提供Eureka Server的功能。

Eureka通过官方开放的端口可以注册服务和获取服务的信息，通过Eureka我们可以轻易的部署集群。Eureka可以为spring cloud提供每个服务运行的实例个数，以及各实例的信息。

4.2模块的交互

cli/Web部分通过Jersey和Spring框架提供的Rest客户端与整个系统交互，交互的数据格式为json

系统间的部分通过rest接口互相交互，通过Ignite共享数据，由Eureka负责服务的注册和发现，Eureka同样使用rest接口，交互的数据格式均为json。

4.3任务流程

最开始由用户通过cli/web创建一个任务模板。该模板可以直接实例化为任务实例或者作为母模板使用。实例化后产生任务实例进入BPMN引擎,该实例的表现格式为BPMN引擎所规定的XML格式，BPMN引擎分析任务实力的内容并按照实例的要求执行任务。在任务执行后，cli/web部分会收到通知，然后可以手动查询任务进度。

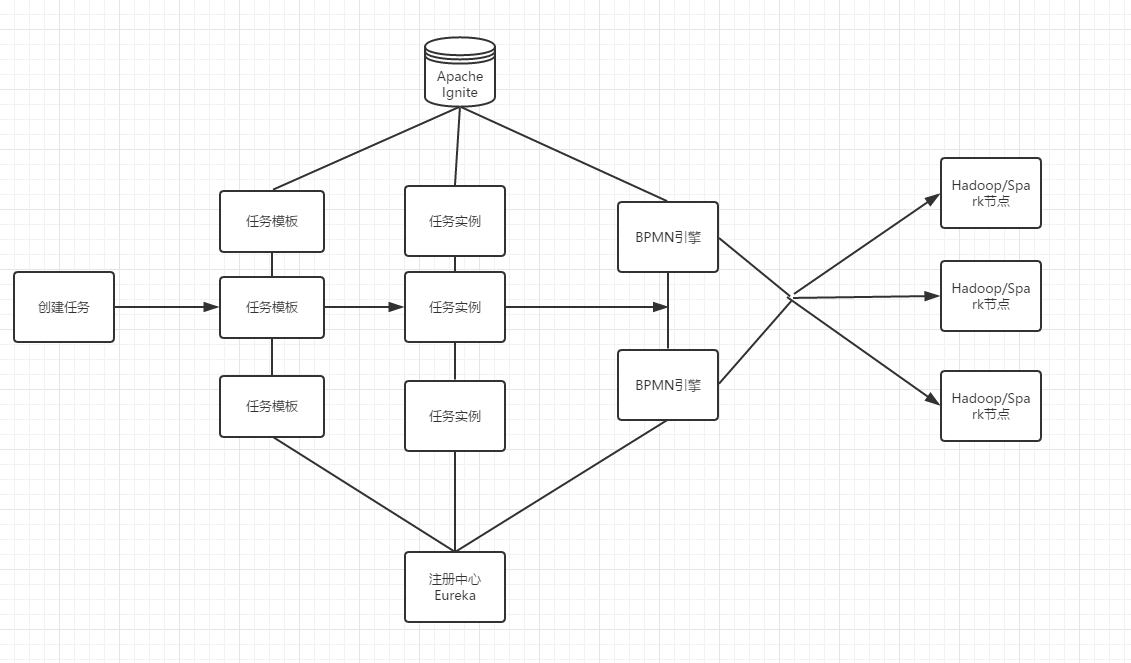


图19 uarp任务流程

4.4安全性

对于所有的rest接口，采用basic基础认证+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。

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。

4.5路由

由于我们的系统是分布式的结构，对于每个服务都存在多个实例，因此，我们通过一定的路由规则去选择使用哪个实例来运行当前请求。

目前，我们仅实现了随机的路由规则。我们通过Eureka注册中心获取请求对应的微服务的实例个数，之后通过随机数确定选择哪个实例来处理当前请求。

由于随机的规则并不能保证系统做到最优的负载均衡，因此，在之后的版本我们可能加入更优的路由规则。

4.6测试

4.6.1 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%

4.6.2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。

4.7开发和部署流程

1). 微服务开发通过maven引入Spring boot、Spring Cloud、ignite等package

2). 按照之前划分好的模块，每个模块单独开发单独测试

3). 通过配置Eureka将各个模块结合起来

4). 通过配置ignite将ignite单独分离出来

5). 编写DockerFile,把每个模块、ignite和Eureka都制作为单独的镜像

6). 编写bash脚本，自动化部署Docker镜像

7). 单独通过JAVA开发cli部分

8). 使用SpringMVC和angularJS开发web部分

4.8开发和部署环境

开发环境均在windows下，JAVA部分由IntelliJ IDEA开发，web前端部分由Visual Studio Code开发。

部署环境：web和cli部分部署在ubuntu的主机上，微服务模块、ignite、Eureka部署在ubuntu下的Docker内。

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.

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。

微服务就是采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2.1所示。

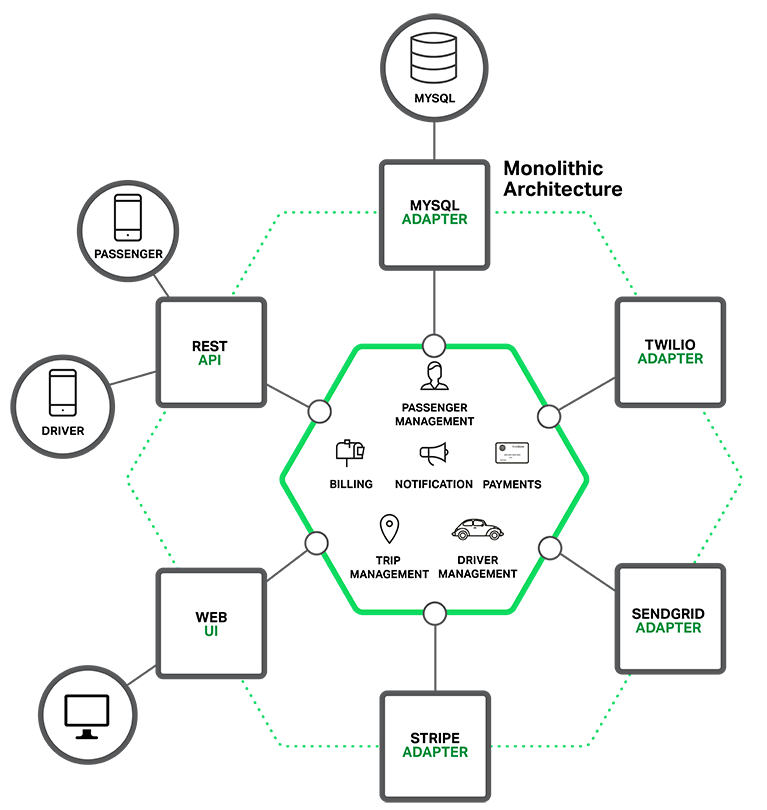


图2.1 微服务之间的联系

对于统一分析运行平台，我们需要有任务编排、上下文管理、按照任务流程进行运行、调用其他服务的功能，要有企业软件所需的安全性、容错性、高效性、易部署的特性，除此之外，还应该拥有可扩展性、可组合性的特性。