微服务架构技术方案

文档修订记录表

（状态：C―创建，A—增加，M—修改，D—删除，R—发布）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 版本 | 状态 | 简要说明（内容和范围） | 更改人 | 更改日期 | 审批人 | 审批日期 |
| V0.1 | C |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

[术语 4](#_Toc479669218)

[架构的选择 4](#_Toc479669219)

[微服务架构 4](#_Toc479669220)

[微服务的优点 7](#_Toc479669221)

[微服务的挑战 9](#_Toc479669222)

[设计原则 11](#_Toc479669223)

[领域模型 12](#_Toc479669224)

[配置中心 13](#_Toc479669225)

[注册中心 14](#_Toc479669226)

[网关 15](#_Toc479669227)

[数据中心 16](#_Toc479669228)

[监控中心 17](#_Toc479669229)

[服务领域 18](#_Toc479669230)

[逻辑架构 19](#_Toc479669231)

[依赖关系 19](#_Toc479669232)

[数据架构 21](#_Toc479669233)

[web视角 21](#_Toc479669234)

[服务端视角 23](#_Toc479669235)

[客户端视角 23](#_Toc479669236)

[网关视角 24](#_Toc479669237)

[注册中心视角 24](#_Toc479669238)

[配置中心视角 25](#_Toc479669239)

[技术架构 26](#_Toc479669240)

[统一的接口 26](#_Toc479669241)

[web前端 26](#_Toc479669242)

[服务 26](#_Toc479669243)

[服务定义和组织 26](#_Toc479669244)

[服务生命周期管理 27](#_Toc479669245)

[服务治理 27](#_Toc479669246)

[服务编制（Orchestration）与服务编排（Choreography） 27](#_Toc479669247)

[负载均衡 28](#_Toc479669248)

[客户端负载 28](#_Toc479669249)

[服务端负载 29](#_Toc479669250)

[特殊需求 30](#_Toc479669251)

[一致性 31](#_Toc479669252)

[可用性 31](#_Toc479669253)

[扩展性 31](#_Toc479669254)

[容错性 32](#_Toc479669255)

[技术选型 32](#_Toc479669256)

[服务消费方(Service Consumer) 32](#_Toc479669257)

[服务提供方(Service Provider) 33](#_Toc479669258)

[注册中心(Register Center) 33](#_Toc479669259)

[配置中心(Config Center) 33](#_Toc479669260)

[网关(Gateway) 33](#_Toc479669261)

[数据中心(Data Center) 33](#_Toc479669262)

[监控中心(Monitor Center) 33](#_Toc479669263)

# 术语

微服务：微服务架构的简称，定义为一种特定的架构解决方案。

# 架构的选择

公司已建和在建系统的技术方案存在多样性，基本上都基于项目组内部的自行选择。其中，既存在老系统架构的沿用，也存在新技术的小规模应用。在分析互联网行业所选用技术的成熟应用和发展趋势上，确定采用基于微服务架构的技术解决方案。

# 微服务架构

微服务定义为一种软件架构，提出一个应用由一系列微小的服务组成，每个微服务运行在各自独立的进程中，相互之间通过轻量级的通信协议协作。所有微服务围绕业务功能构建，可以全自动的独立部署。这些微服务使集中管理降到最小化，因此可以采用不同的编程语言和数据存储技术实现。

与微服务架构对应是单体架构，也是已有和在建系统常规采用的架构。单体架构在实际应用中存在以下几个显著的痛点：

* 整体停机



应用系统中当某个服务发现设计缺陷、发生致命异常、进行功能升级，而其他服务均正常的情况下，单体架构中需要对整个应用系统进行启停操作，导致正常的服务也不能使用。

* 资源浪费



应用系统中某个服务达到性能瓶颈或者可用性阈值时，可以采用集群的部署方式。单体架构采用整体扩容的方式，而不是针对瓶颈点进行扩容，导致资源的浪费。

* 重复建设



单体架构下的多系统基本以业务部门或项目的形式划分，分别独自进行建设而缺少业务上或技术上的交集。业务领域存在很多共性和特性，共性的内容放在各系统一方面导致了不必要的重复建设，另一方面在数据一致性也存在问题。

## 微服务的优点

* 强大的功能模块划分

传统软件架构和微服务都提倡模块化的划分方式，以此降低开发的复杂度和依赖性。传统的软件架构各模块之间相互依赖，统一部署，随着业务复杂度的提升，模块之间的耦合性越来越强。而微服务提倡的模块划分方式屏蔽了模块间的依赖，具有自包含的完整功能，每个微服务都不依赖其他特定的微服务。



单体架构的多个模块存在一定的依赖关系和耦合性。微服务架构的多个模块不存在依赖关系。

* 独立的部署

传统软件架构虽然各模块可以独立的开发、调试，但是需要依赖于其他的模块，并且必须采用统一部署的方式。而微服务因为各自高度的独立性，可以独立部署在不同的进程中，每一个微服务的部署升级都不会影响到其他微服务。



单体架构下的多个模块打包为一个jar或者war，部署在同一个进程中。微服务架构下的多个模块打包为不同的jar或者war，部署在不同的进程中。

* 技术多样性

传统软件架构各模块需要使用统一的编程语言，而微服务中的每个模块可以使用不同的技术、不同的编程语言。



## 微服务的挑战

* 分布式

传统的系统一般在进程内、方法级调用即可实现一个功能，而分布式系统需要进程间、系统间、物理机器间的调用才能实现一个功能，由此会带来额外的风险。单机、单DB、单进程早已不能满足很多应用场景的需要，在发展的历程中，基本都遵循单机单进程—>单机多进程—>多机多进程的方式。分布式问题在现有企业级应用的架构中普遍存在，我们一方面要深入的分析业务功能在分布式架构下可能产生的各种问题，另一方面要采用合适的方法、工具、手段对整个软件生命周期的各个阶段做好支撑和管理。

* 最终一致性

不同的业务领域在一致性方面有不同要求，强一致性是金融领域的传统需求，但同时也是性能的拦路虎，而采用最终一致性，一方面可以在一个可接受的时间范围内保证最终的一致性，另一方面可以显著提高系统性能。在最终一致性方案下，一个功能的完整实现需要多个步骤，每个步骤相互独立，任何一个步骤失败都会导致功能失败，因此需要针对每个步骤可能出现的任何异常情况做额外的处理，给系统带来一定的复杂性。微服务采用分布式的部署方式，在最终一致性方面面对更大的挑战。

* 操作复杂性

微服务因粒度定义更细，服务更多，进程更多，会给分布式系统带来数量级增长的开发、部署、运维难度。首先，微服务架构需要采用稳定的协议，隐藏所有的实现细节，只将数据契约和网络协议暴露出来。其次，在解耦的前提下，服务的定义和拆分是一个循序渐进的过程，不能在初期一次性定义过多的服务。

# 设计原则

没有银弹

架构的设计是在平衡中做选择，在实践中促发展。没有任何一个架构可以完美解决所有问题。微服务架构不是与传统架构绝对的对立，相互之间存在借鉴的关系。

促进DevOps闭环

人的因素所造成的系统风险远大于技术造成的风险，需求、设计、开发、测试、运维之间的“墙”需要被打掉。

低耦合、高内聚

软件架构设计的常用原则。传统架构中以横向分层的方式实现，微服务架构提倡领域驱动划分，模块化和层次化是基本的实施路径。

面对挑战、拥抱变化

技术、业务会根据发展产生不同的特性，领域、模块、层次也是在分分合合中发展。

# 领域模型

软件系统提供的各种功能对内可以采用不同的技术实现，对外可以表现为不同的形式，总体来说，可以归纳为服务化的思想，在各种新老系统的设计理念中都体现了服务化。围绕服务化，可以衍生出各种具有抽象性和共性的技术领域。



## 配置中心



系统使用的一部分数据以配置的形式体现，例如：日志、数据库、JVM等相关的参数。配置信息经历了文件、数据库、缓存的演变。分散的配置信息给管理维护带来极大的复杂性，采用配置中心作为集中统一的管理可以带来极大的便利。

## 注册中心



现有的企业应用中，系统间的协作已经普遍存在。在无注册中心的情况下，当某个被依赖系统维护、停机、升级时，不能较好的通知其他的依赖系统；在系统协作关系复杂时，不能较好的理清网状的协作关系。注册中心也是采用集中管理维护的思想，以改进系统庞大、服务增长之后分散管理的缺点。

## 网关



系统间协作的直接调用在单机、集群、分布式的环境下都会面临一系列的安全问题、路由问题、监控问题、可用性问题等。网关的出现即是针对性的解决这些问题，网关作为业务系统的前置，不仅仅提供硬件负载所支持的路由和转发功能，还可以根据技术和业务需求进行定制。

## 数据中心



数据是所有系统运行、业务逻辑处理的核心。根据不同的要求，数据会以不同形式体现。例如，日志信息以文件的形式体现，持久化数据以数据库的形式体现，高频数据以缓存的形式体现。



数据处理的过程中会存在分散-->集中或者集中-->分散的过程。微服务架构中优先关注应用系统内数据的处理过程。

## 监控中心



硬件环境的监控已经存在成熟的产品，而软件系统因设计的差异性，所采用的监控手段、监控能达到的粒度各不相同。微服务架构下，各个技术领域能保证持续稳定运行的前提条件就依赖于监控中心的有效性。业务系统在遵循统一设计的前提下，可以较好的保证监控的快速实施，以达到高效支撑、持续运营的目的。

## 服务领域



基础服务：技术相关的逻辑抽象，为服务间的协作提供基础支撑。

共享服务：业务功能共性的逻辑抽象，封装企业各个维度的核心数据资源和业务规则，避免重复建设，统一业务规范，支撑业务功能的快速交付。

业务服务：面向不同的业务部门、业务人员，根据不同特点和发展需求，提供定制化的功能。

服务分层后的协作关系如下：

* 上层可以调用下层，比如共享服务调用基础服务，也可以跨层调用，比如业务服务直接调用基础服务。
* 同层方面，业务服务可以互相调用，组成更粗粒度服务，共享服务和基础服务都是细分服务，相互之间垂直正交，不允许相互调用。

# 逻辑架构



## 依赖关系

各领域模型在不同状态下对其他领域模型的依赖如以下列表所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 注册中心 | 配置中心 | 网关 | 服务提供方 | 服务消费方 | 数据中心 |
| 注册中心 | 启动时 | - | No | No | No | No | No |
| 运行时 | - | No | No | No | No | No |
| 宕机时 | - | - | - | - | - | - |
| 配置中心 | 启动时 | Weak | - | No | No | No | No |
| 运行时 | Weak | - | Weak | No | No | No |
| 宕机时 | - | - | - | - | - | - |
| 网关 | 启动时 | Force | Force | - | No | No | No |
| 运行时 | Weak | Weak | - | Weak | No | No |
| 宕机时 | - | - | - | - | - | - |
| 服务提供方 | 启动时 | Force | Force | No | - | No | Force |
| 运行时 | Weak | Weak | No | - | No | Force |
| 宕机时 | - | - | - | - | - | - |
| 服务消费方 | 启动时 | Force | Force | No | No | - | Force |
| 运行时 | Weak | Weak | Force | Weak | - | Force |
| 宕机时 | - | - | - | - | - | - |
| 数据中心 | 启动时 | No | Force | No | No | No | - |
| 运行时 | No | Weak | No | No | No | - |
| 宕机时 | - | - | - | - | - | - |

各领域依靠网络进行协作，依赖关系可以分为：force-强依赖；weak-弱依赖，可以自动恢复；no-无依赖。

配置中心、注册中心作为基础设施为其他域提供支撑；网关抽取管理、监控、控制等共性，使之与实际的业务代码解耦；服务提供方和服务消费方即是业务代码，只处理业务相关的逻辑。

# 数据架构

## web视角



1. web发起请求
2. 网管验证ticket是否存在
3. ticket不存在，web跳转登录
4. web发起登录请求
5. 网关调用认证服务
6. 认证成功，认证服务返回token
7. 网管返回tocket至web
8. web发起申请ticket请求
9. 网管调用认证服务，申请ticket
10. 认证服务返回ticket
11. 网关把ticket写入cookie
12. 网关返回cookei至web
13. web发起业务请求
14. 网关调用认证服务，验证ticket是否合法
15. 认证服务返回ticket验证结果
16. 网关判断ticket是否合法
17. ticket合法，网关调用授权服务验证权限
18. 授权服务返回权限验证结果
19. 网关判断权限是否合法
20. 权限合法，网关调用业务服务
21. 业务服务返回业务响应
22. 网关返回业务响应至web

## 服务端视角



1. 服务向注册中心注册、刷新、注销服务信息
2. 服务从配置中心获取配置信息，配置中心向服务推送配置信息
3. 网关转发请求至服务
4. 多服务间存在调用的关系

## 客户端视角



1. 客户端从注册中心获取服务信息
2. 客户端从配置中心获取配置信息
3. 客户端向网关发起请求

## 网关视角



1. 网关从注册中心获取服务信息
2. 网关从配置中心获取配置信息，配置中心推送配置信息至网关
3. 客户端发起请求到网关
4. 网关转发请求到服务端

## 注册中心视角



1. 服务向注册中心注册、刷新、注销服务信息
2. 客户端从注册中心获取服务信息
3. 网关从注册中心获取服务信息
4. 配置中心向注册中心注册、刷新、注销配置服务

## 配置中心视角



1. 服务从配置中心获取配置信息，配置中心向服务推送配置信息
2. 客户端从配置中心获取配置信息
3. 网关从配置中心获取配置信息，配置中心推送配置信息至网关
4. 配置中心向注册中心注册、刷新、注销配置服务

# 技术架构

## 统一的接口

微服务架构提倡技术多样性，以服务为边界，对外提供统一的接口，内部的实现可以采用各种不同技术。统一的接口涉及到：通信协议、接口定义、消息格式。通信协议采用HTTP，以符合restful风格的形式定义接口，消息格式采用json。选用的方案具备编程语言无关性的特点，也是通用的互联网标准。统一的接口作为原语可以简化人员、组织之间的理解。

## web前端

web前端采用ReactJS框架，UI组件借鉴AntD并在此之上进行定制化开发。

## 服务

### 服务定义和组织

服务名采用统一的命名规范，可以采用所见即所得的英文命名。服务名原则上具有全局唯一性，但考虑到实际的场景，唯一性可以限制为某个领域范围之内。服务表示为一种资源，在服务数量增长之后，所有资源以平级的方式组织，不利于识别和管理，因此需要以URI的形式对服务分组。例如：系统A的服务资源可以表示为：/SystemA/ServiceA，/SystemA/ServiceB。

### 服务生命周期管理

服务生命周期仅包含服务本身是否可用，包括：注册、更新、注销。服务提供方与注册中心协作完成整个生命周期的管理。

### 服务治理

在服务提供的业务功能基础上，可以根据技术需求或者业务需求进行定制化的管理。例如：服务的负载策略、服务的访问权限。网关可以根据配置中心的实时配置信息进行动态的服务治理。

### 服务编制（Orchestration）与服务编排（Choreography）

服务可以分为原子服务和组合服务。原子服务可以独立提供功能，在演变中可能会抽取出其他的原子服务。组合服务需要多个服务协调提供功能，一般为多个原子服务间的协调。原子服务和组合服务在粒度定义上的差异，会产生不同划分。

组合服务可以分为串行执行和并行执行两种情况。

## 负载均衡

在分布式环境下，负载均衡可以提高资源利用率、最大化吞吐量、最小化响应时间、避免过载。负载均衡可以采用硬件也可以采用软件，本章节主要针对软件负载。

负载均衡有两种常规解决方案：客户端负载和服务端负载。

### 客户端负载



每个客户端获取并维护服务ip列表，通过指定的算法向服务端发起请求。客户端负载适用于内部服务的调用，负载程序集成在客户端程序中。

### 服务端负载



在多个服务之上提取出中间层，可以是网关、代理等。由中间层负责所有请求的接入、负载算法、请求转发。服务端负载可以适用于所有的场景。

客户端负载适用的场景、负载算法较简单，对客户端程序的耦合性较强，升级与扩展具有局限性。服务端负载适用所有场景，可以基于系统监控、业务规则做算法的扩展，但需要重点关注高可用性。

### 特殊需求

动态伸缩

服务端随着系统压力的增减会进行节点的新增或删除。

容错

服务端存在宕机的风险和恢复的能力。

灰度加载

在特殊技术要求和业务需求情况下，会产生服务的差异化。



针对特殊需求的算法规则，可以在配置中心进行配置，网关和配置中心之间进行同步。

## 一致性

强一致性要求的业务，原则上不做一致性拆分，也不做服务的拆分。弱一致性要求的业务，提供最终一致性的解决方案，提供补偿机制和回滚机制。

## 可用性

配置中心、注册中心、网关、数据中心、业务服务全部采用集群的方式部署。一般情况下，通过负载分散压力到集群中各个节点。在单节点失效的情况下，屏蔽失效节点，其他有效节点分担失效节点的压力。

## 扩展性

高并发情况下，集群负载过高或者达到阈值，集群支持横向扩展。资源冗余情况下，集群支持收缩。

## 容错性

屏蔽失效节点，对外服务无影响，外部不可见。同时，可以在服务粒度进行容错处理。

## 技术选型

|  |  |
| --- | --- |
| Technology | Version |
| JDK | 1.8 |
| SpringBoot | 1.5.2 |
| SpringCloud | Dalston.RC1 |
| SpringCloud-Config | 1.2.3 |
| SpringCloud-Consul | 0.7.5 |
| NetFlix-Zuul | 1.3.0 |
|  |  |

### 服务消费方(Service Consumer)

服务消费方可以从两个层次划分：以UI形式提供给使用人员，即是前端程序；在CS模式中，作为Client端的软件系统。前端程序采用ReactJS作为开发框架；Client端和Server端采用相同的SpringBoot框架。

### 服务提供方(Service Provider)

服务提供方即业务逻辑处理程序，基于SpringBoot框架开发。

### 注册中心(Register Center)

Consul

### 配置中心(Config Center)

Spring Cloud Config

### 网关(Gateway)

Zuul

### 数据中心(Data Center)

根据DataBase、NoSQL、Cache选型的不同，分别定制不同的产品方案。

### 监控中心(Monitor Center)

先基于各个域自身提供的监控功能进行，后搭建统一的微服务架构软件系统监控平台。