ISC微服务

架构设计说明书

版本：1.4

编制人：王孝坤

目录

[ISC微服务 1](#_Toc6837222)

[架构设计说明书 1](#_Toc6837223)

[一、 微服务背景介绍 3](#_Toc6837224)

[二、 微服务框架技术选型 5](#_Toc6837225)

[三、 ISC微服务架构设计 6](#_Toc6837226)

[四、 用户认证设计 10](#_Toc6837227)

[五、 服务鉴权设计 12](#_Toc6837228)

[六、 ISC微服务架构搭建的项目结构 14](#_Toc6837229)

[七、 开发调试环境部署图设计 15](#_Toc6837230)

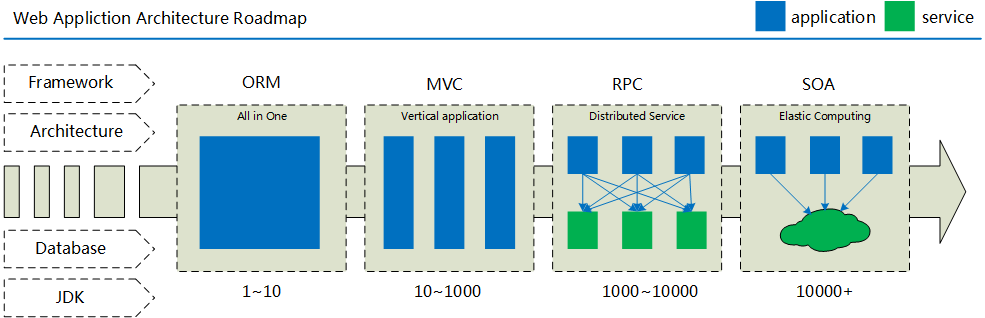
[八、 集成环境部署图设计 16](#_Toc6837231)

[九、 其他 17](#_Toc6837232)

[十、 版本修订历史记录 20](#_Toc6837233)

# 微服务背景介绍

互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，分布式服务框架在这种情况下诞生的。将核心业务抽取出来，作为独立的服务，使前端应用能更快速和稳定的响应。



1. 单体架构，在小微企业比较常见，典型代表就是一个应用、一个数据库、一个web容器就可以跑起来。

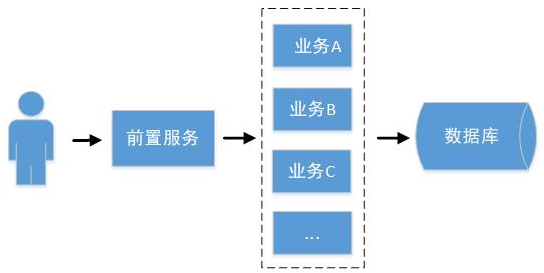
在单体架构中，技术选型非常灵活，优先满足快速上线的要求，也便于快速跟进市场。

下面是单体架构的架构图：



1. 垂直架构，在单体架构发展一段时间后，业务模式得到了认可，业务逻辑处理量大起来，并发访问量不断增长，就会对原有的业务进行拆分，比如说：后台系统、前端系统、业务模块系统等。

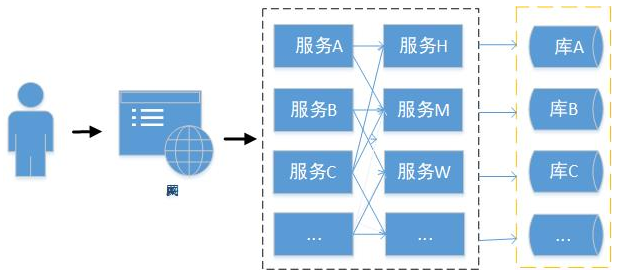
下面是垂直架构的架构图：



1. 服务化架构，垂直子系统会变的越来越多，系统和系统之间的调用关系呈指数上升的趋势。在这样的背景下，SOA代表面向服务的架构，将应用程序根据不同的职责划分为不同的模块，不同的模块直接通过特定的协议和接口进行交互。当流量过大时通过水平扩展相应的组件来支撑，所有的组件通过交互来满足整体的业务需求。

服务化架构是一套松耦合的架构，服务的拆分原则是服务内部高内聚，服务之间低耦合。

下面是服务化架构图：



1. 大规模服务化之前，应用可能只是通过RMI或Hessian等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过F5等硬件进行负载均衡。
2. **当服务越来越多时，服务URL配置管理变得非常困难，F5硬件负载均衡器的单点压力也越来越大。**

此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。

并通过在消费方获取服务提供方地址列表，实现软负载均衡，降低对F5硬件负载均衡器的依赖，也能减少部分成本。

1. **当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。**

这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

1. **接着服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？**

为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。

其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

综上所述需要引入微服务架构，在这个阶段可以使用dubbo或者springcloud来服务治理。微服务架构是 SOA 架构思想的一种扩展，更加强调服务个体的独立性、拆分粒度更小。

微服务架构解决的第一个问题就是：服务与服务之间的解耦。在业务高速发展的时候，服务组件也会相应的不断增加。服务和服务之间有着复杂的相互调用关系，经常有服务A调用服务B，服务B调用服务C和服务D ...，随着服务化组件的不断增多，服务之间的调用关系成指数级别的增长。

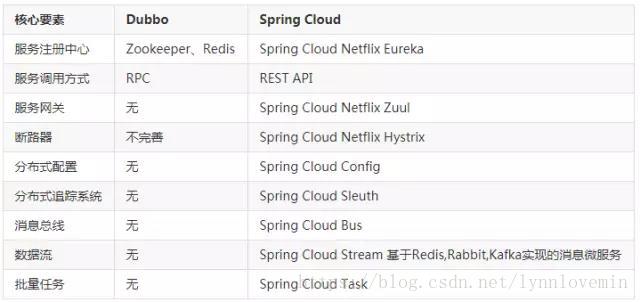
# 微服务框架技术选型

1. Spring Cloud 是什么

SpringCloud是一系列框架的有序集合。它利用SpringBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用SpringBoot的开发风格做到一键启动和部署。SpringCloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过SpringBoot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

1. Spring Cloud和Dubbo对比

Dubbo 只是实现了服务治理，而 Spring Cloud 实现了微服务架构的方方面面，服务治理只是其中的一个方面。下面通过一张图对其进行比较：



1. Spring Cloud 优缺点

其主要优点有：

* + 集大成者，SpringCloud 包含了微服务架构的方方面面
  + SpringCloud来源于Spring，质量、稳定性、持续性都可以得到保证
  + 约定优于配置，基于注解，没有配置文件
  + SpirngCloud天然支持Spring Boot，更加便于业务落地
  + Spring Cloud发展非常的快，从16年开始接触的时候相关组件版本为1.x，到现在已发布的2.x系列
  + 轻量级组件，SpringCloud 整合的组件大多比较轻量级，且都是各自领域的佼佼者
  + 开发简便，SpringCloud 对各个组件进行了大量的封装，从而简化了开发
  + 开发灵活，SpringCloud 的组件都是解耦的，开发人员可以灵活按需选择组件

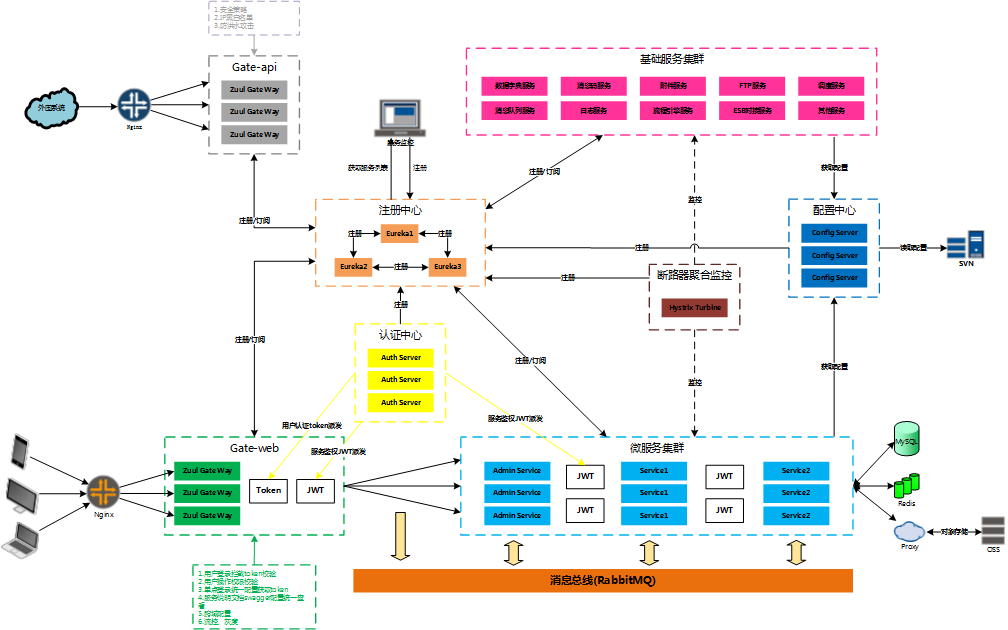
接下来，我们看下它的缺点：

* 项目结构复杂，每一个组件或者每一个服务都需要创建一个项目。
* 部署门槛高，项目部署需要配合 Docker /k8s等容器技术进行集群部署，而要想深入了解 Docker，学习成本高。

综上所述：SpringCloud的优势是显而易见的，搭建微服务架构SpringCloud是一个不错的选择。

# ISC微服务架构设计

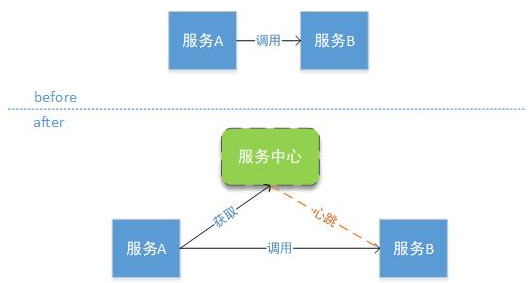
ISC微服务架构是基于SpringCloud框架进行设计，如图所示：



* 1. **注册中心Eureka**

Eureka是Netflix开源的一款提供服务注册和发现的产品，它提供了完整的Service Registry和Service Discovery实现。

Eureka就是一个服务中心，将所有的可以提供的服务都注册到它这里来管理，其它各调用者需要的时候去注册中心获取，再进行调用。避免了服务之间的记录url的直接调用，方便后续的水平扩展、故障转移等。如下图：



服务中心这么重要的组件一但挂掉将会影响全部服务，因此需要搭建Eureka集群来保持高可用性。随着系统的流量不断增加，需要根据情况来扩展某个服务，Eureka内部已经提供均衡负载的功能，只需要增加相应的服务端实例既可。在系统的运行期间某个实例挂了怎么办？Eureka内容有一个心跳检测机制，如果某个实例在规定的时间内没有进行通讯则会自动被剔除掉，避免了某个实例挂掉而影响服务。

**因此使用了Eureka就自动具有了注册中心、负载均衡、故障转移的功能。**

* 1. **负载均衡**

客户端请求的负载均衡：

所有的客户端请求通过网关(zuul/gateway)访问后台的服务。它可以使用一定的路由配置来判断某一个URL由哪个服务来处理。并从Eureka获取注册的服务来转发请求。网关将一个请求发送给某一个服务的应用的时候，如果一个服务启动了多个实例，就会通过**Ribbon(Netflix组件)**来通过一定的负载均衡策略来发送给某一个服务实例。

服务之间请求的负载均衡：

服务之间如果需要相互访问，可以使用RestTemplate，也可以使用**Feign(Netflix组件)**客户端访问。它默认会使用Ribbon来实现负载均衡。

* 1. **配置中心**

随着微服务不断的增多，每个微服务都有自己对应的配置文件。在研发过程中有开发环境、测试环境、UAT环境、生产环境，因此每个微服务又对应至少三四个不同环境的配置文件。这么多的配置文件，如果需要修改某个公共服务的配置信息，难免会产生混乱，这个时候就需要引入Spring Cloud另外一个组件：Spring Cloud Config。

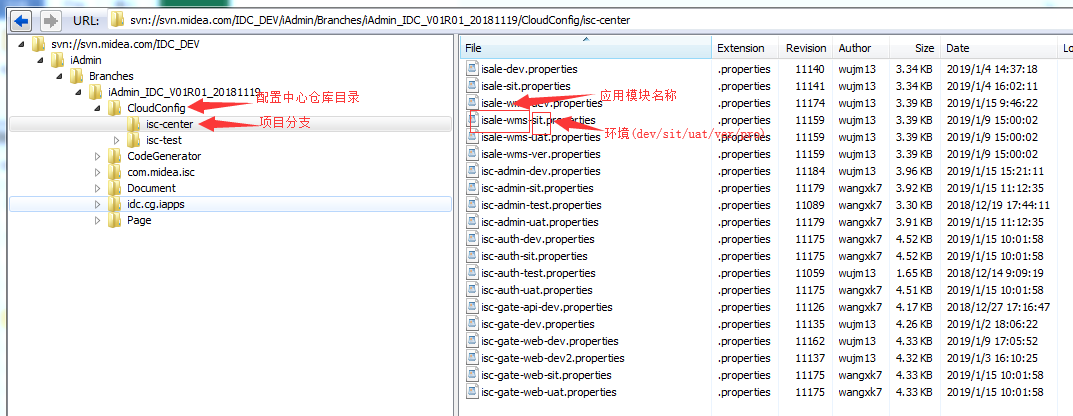
* Spring Cloud Config是一个**解决分布式系统的配置管理方案**。它包含了Client和Server两个部分，Server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，Client通过接口获取数据、并依据此数据初始化自己的应用。
* 如果服务运行期间改变配置文件，服务是不会得到最新的配置信息，需要解决这个问题就需要引入Refresh。可以在服务的**运行期间重新加载配置文件**。
* 当所有的配置文件都存储在配置中心的时候，配置中心就成为了一个非常重要的组件。如果配置中心出现问题将会导致灾难性的后果，因此对配置中心也需要做集群支持**高可用性**。
* 配置中心的地址一定要放在bootstrap.properties中，这个配置文件是由“根”上下文优先加载，可以保证程序启动之初就感知到远程配置中心的存在，并从远程获取配置，随后继续启动系统。而application.properties是由子上下文加载，加载顺序低于前者，如果配置中心地址放在这里，并且你远程配置了一些启动相关的必要参数，那么，你的程序很可能由于缺少参数而启动失败。

当前配置文件以svn作为仓储，配置文件获取规则如下：

spring.cloud.config.label 指明远程仓库的项目分支

spring.cloud.config.name 指明当前应用名称，不配置默认为spring.application.name

spring.cloud.config.profile 指明当前环境阶段(dev/sit/uat/ver/pro)



* 1. **网关(web/api)**

在微服务架构模式下，后端服务的实例数一般是动态的，对于客户端而言很难发现动态改变的服务实例的访问地址信息。

目前两个主流的网关组件如下：

Spring Cloud Zuul路由提供动态路由，监控，弹性，安全等的边缘服务。Zuul是Netflix出品的一个基于JVM路由和服务端的负载均衡器。它的具体作用就是服务转发，接收并转发所有内外部的客户端调用。使用Zuul可以作为资源的统一访问入口，同时也可以在网关做一些权限校验等类似的功能。

Spring Cloud Gateway是Spring官方基于Spring 5.0，Spring Boot 2.0和Project Reactor等技术开发的网关，Spring Cloud Gateway旨在为微服务架构提供一种简单而有效的统一的API路由管理方式。Spring Cloud Gateway作为Spring Cloud生态系中的网关，目标是替代Netflix ZUUL，其不仅提供统一的路由方式，并且基于Filter链的方式提供了网关基本的功能，例如：安全，监控/埋点，和限流等。

技术选型：spring-cloud-Gateway是spring-cloud的一个子项目。而zuul则是netflix公司的项目，只是spring将zuul集成在spring-cloud中使用而已。spring gateway使用基于netty异步io；zuul使用servlet 3，每个请求一个线程，同步Servlet，多线程阻塞模型。而且基本可以肯定Spring Cloud Finchley版本的gateway比zuul 1.x系列的性能和功能整体要好。所以本架构选则spring gateway作为网关组件。

ISC微服务架构提供两个不同用途的网关

Gate-web：

1. 前端服务请求的统一入口
2. 单点登录统一配置，获取token
3. 校验token
4. 用户服务调取权限校验
5. 服务接口说明文档swagger统一查看
6. 跨域配置
7. 流控、灰度

Gate-api

1. 外围系统请求的统一入口
2. 安全策略制定
3. IP黑白名单
4. 防洪水攻击
   1. **认证中心**
      1. 用户认证
5. 用户登录校验、token生成
6. 用户登录信息分布式存储和更新
7. 用户服务权限获取
   * 1. 服务鉴权
        1. 服务发现和注册到数据库，生成secret
        2. RSA加密算法的公钥派发
        3. 服务校验和JWT派发(带加密签名的令牌)
        4. 通过消息总线定时推送服务间的访问权限
   1. **平台管理服务(Admin Service)**

新一代权限管理平台的后台服务，包括但不限于以下功能模块：应用管理、语言管理、国家地区管理、菜单管理、操作管理、用户管理、角色管理、服务管理、菜单授权、操作授权、服务授权等等。

* 1. **基础服务集群**

**数据字典服务**：下拉值的统一配置和获取，支持多语言

**消息码服务**：错误码的统一配置和获取，支持展示形式控制和多语言配置，同时配套提供前端Vue框架的封装(选用)

**附件服务**：提供基于美的云盘的附件存储和下载，支持基于项目的用户空间配置，同时配套提供前端Vue框架的封装(选用)

**FTP服务**：提供可配置的FTP服务，支持高并发、断点续传、监控等等的文件传输功能

**调度服务**：支持动态的job配置，调度监控等功能

**消息队列服务**：统一消息队列中间件使用的服务，实现消息队列的持久化、重发机制、高可用集群的统一配置等

**日志服务**：提供服务间互调的日志存储和查询，便于定位问题和监控

**流程引擎服务**：提供流程引擎的统一对接，支持按项目的流程对接配置，统一管理流程引擎的基础连接配置和日志跟踪记录等

**ESB对接服务**：基于老系统的数据对接而衍生的服务

* 1. **微服务集群**

基于服务内部高内聚、服务间低耦合的设计理念进行业务模块的划分，建议将业务功能完整且独立的模块划分为一个服务，例如：客户关系管理(CRM)、仓储管理(WMS)、订单管理(OMS)、售后管理(Service)等等，服务内部功能做强做大，并向其他服务提供统一的服务接口。

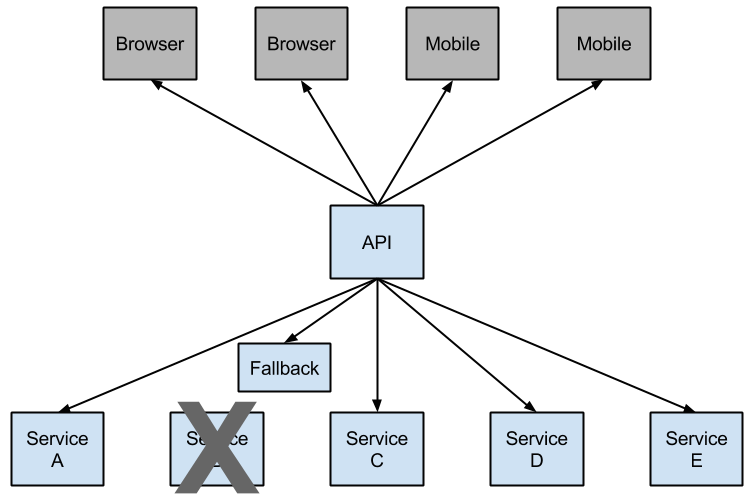
* 1. **断路器聚合监控**

1. 雪崩效应

在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用，在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。为了解决这个问题，业界提出了断路器模型。

1. 熔断机制(断路器)

一个微服务的超时失败可能导致瀑布式连锁反映，下图中，Hystrix通过自主反馈实现的断路器， 防止了这种情况发生。



图中的服务B因为某些原因失败变得不可用，所有对服务B的调用都会超时。当对B的调用失败达到一个特定的阀值(5秒之内发生20次失败是Hystrix定义的缺省值), 链路就会被处于open状态， 之后所有所有对服务B的调用都不会被执行， 取而代之的是由断路器提供的一个表示链路open的Fallback消息。Hystrix提供了相应机制，可以让开发者定义这个Fallbak消息。

open的链路阻断了瀑布式错误， 可以让被淹没或者错误的服务有时间进行修复。这个fallback可以是另外一个Hystrix保护的调用, 静态数据，或者合法的空值.。Fallbacks可以组成链式结构，所以，最底层调用其它业务服务的第一个Fallback返回静态数据。

1. 断路器套件

Hystrix，监控和断路器

Hystrix Dashboard，监控面板，提供了一个界面来实时查看接口的运行状态和调用频率等。

Turbine，监控聚合，使用Hystrix监控，我们需要打开每一个服务实例的监控信息来查看。而Turbine可以帮助我们把所有的服务实例的监控信息聚合到一个地方统一查看。

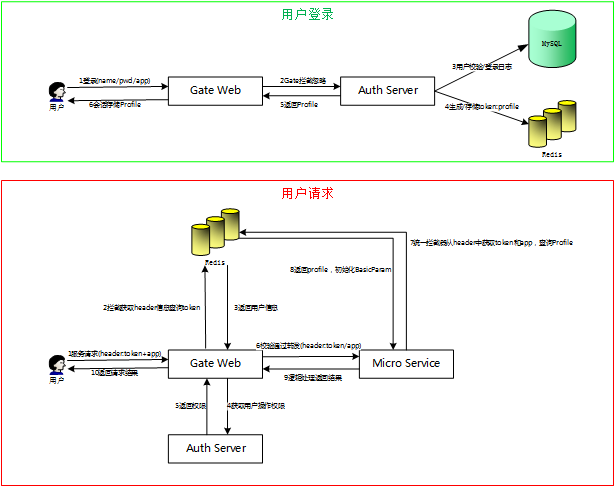
具体的服务只配置断路器hystrix和健康监控actuator依赖即可，平台提供基于hystrix dashboard和turbine的聚合监控组件实现hystrix的仪表盘聚合查看。

* 1. **服务器监控**

利用Spring Boot Admin 来监控各个独立服务的运行状态以及配合运维部门的雅典娜自动化平台和阿格斯一体化监控实现服务器状态和日志的监控。

# 用户认证设计

* 1. **用户认证授权(非SSO)**

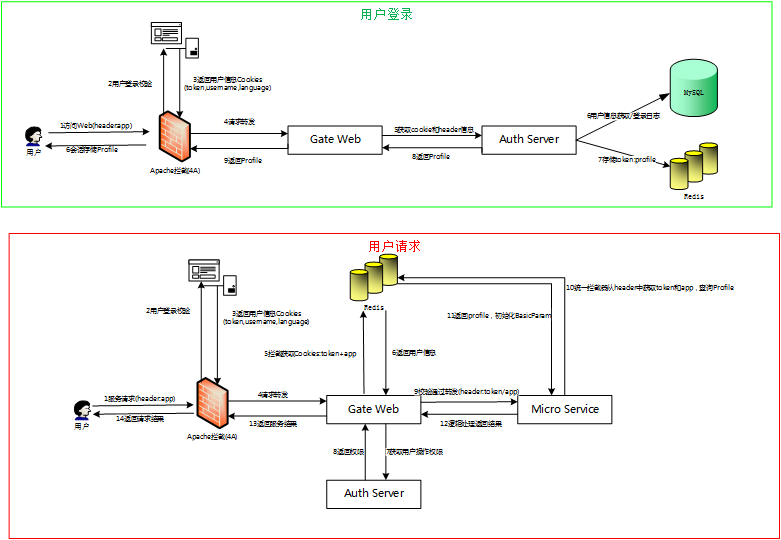


用户登录：

1. 前端输入用户名和密码(md5加密)，调用auth/weblogin服务
2. Gate-web不拦截登录请求，即不校验token
3. Auth Server通过Common库验证用户名和密码并记录登录日志
4. 校验通过后生成唯一标志token，并根据用户信息初始化Profie，然后存储到分布式缓存Redis中(key为app+token)，供其他用户请求时校验和获取用户信息
5. Auth Server返回Profile到前端
6. 前端将Profile缓存在当前会话中

用户请求(登录后的服务请求)：

1. 前端发起请求，请求头Header需要添加已经缓存的token和app
2. Gate-web设置过滤器处理请求，获取请求头信息token和app
3. 根据token和app从redis获取已登录用户的信息，同时校验是否登录或者过期
4. 根据用户信息调用Auth-server获取用户的操作权限
5. 根据返回的权限信息判断当前用户是否有权限继续调用服务
6. 校验通过后，将token和app添加到网关的请求中再转发到指定服务
7. 服务添加了统一请求拦截器，获取请求头的token和app后从redis获取profile
8. 将profile初始化到入参的基类BasicParam中 ，这样此次服务的方法就可以获取到当前登录用户的所有信息
9. 完成逻辑处理返回结果到Gate-web
10. Gate-web将结果返回前端
    1. **用户认证授权(SSO)**



用户登录：

与非SSO的区别在于ISC架构不提供用户登录界面，直接发送具体请求，由4A的Apache拦截并跳转到登录界面，登录成功后4A将token和用户信息存储在cookies中，前端的请求头Header中同样要带app，Gate-web通过cookies获取token、用户名和语言，通过header获取app，其他操作同非SSO登录。

**具体的服务不再区分SSO和非SSO登录，由Gate-web完成这个工作，具体服务只需要根据Header中的token和app获取当前登录用户信息即可，当然这个步骤也已经由框架完成，服务只要关注自己的业务逻辑代码即可。**

用户请求(登录后的服务请求)：

与非SSO的区别在于token的获取不需要前端通过header传递，直接从cookies中获取，前端只需要在请求头中添加app。

**单点登录后切换app，其实对于后端来说会重新走一次用户信息的获取和保存操作，后端的用户信息存储是以app进行分割的，这个也是有必要的，因为不同app的登录附带信息不一样。**

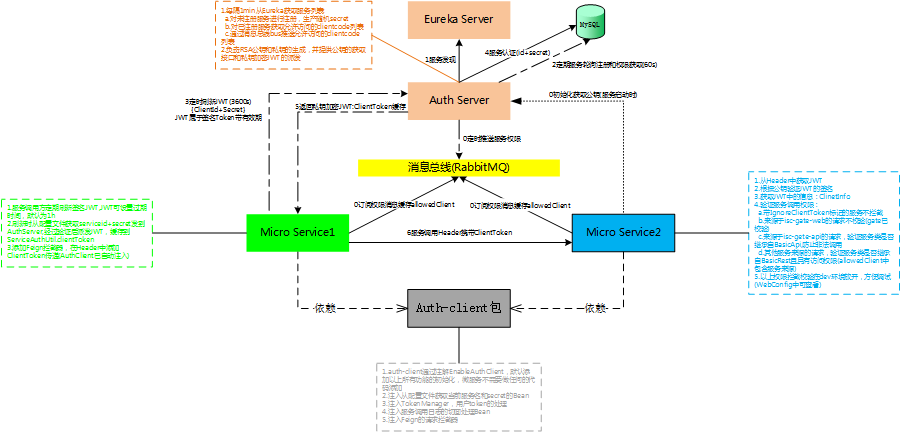
# 服务鉴权设计

注册中心维护了整个体系中的所有服务，每个服务都是一个可以独立运作的个体，且服务间是需要互相访问的，而这种访问不能没有限制，不然将产生数据安全性的问题，固需要设计一套基于服务间的鉴权机制。本架构选择JWT技术进行服务鉴权的设计。

这里简单介绍一下JWT(JSON Web Token)：

* JWT是一种基于JSON的、用于在网络上声明某种主张的令牌（token）。JWT通常由三部分组成: 头信息（header）, 消息体（payload）和签名（signature）。头信息指定了该JWT使用的签名算法，消息体包含了JWT的意图，签名则通过私有的key计算而成。
* JWT常常被用作保护服务端的资源（resource），客户端通常将JWT通过HTTP的Authorization header发送给服务端，服务端使用自己保存的key计算、验证签名以判断该JWT是否可信。
* 应用场景：最适合的应用场景就是“开票”，或者“签字”。放到系统集成的场景中，JWT更适合一次性操作的认证。JWT适合一次性的命令认证，颁发一个有效期极短的JWT，即使暴露了危险也很小，由于每次操作都会生成新的JWT，因此也没必要保存JWT，真正实现无状态。
* 在Web应用中，别把JWT当做session使用，绝大多数情况下，传统的cookie-session机制工作得更好。

基于JWT的服务鉴权设计如图：



本图以Micro Service1向Micro Service2发起请求为例子展示一次普通的服务请求过程中发生了什么事情。

1. **各个服务的初始化操作如下：**

Auth Server：

* 每隔1min从Eureka获取服务列表:

1. 对未注册服务进行注册，生成随机secret，各个服务需要将在secret存放到自己的配置文件中
2. 对已注册服务获取允许访问的clientcode列表
3. 通过消息总线bus推送允许访问的clientcode列表

* 负责RSA公钥和私钥的生成，并提供公钥的获取接口和私钥加密JWT的派发

Micro Service1：

* 服务调用方定期刷新签名JWT,JWT可在Auth Server设置过期时间，默认为1h
* 刷新时从配置文件获取serviceid+secret发到Auth Server,经过验证后派发加密的数字签名JWT，缓存到ServiceAuthUtil.clientToken
* 添加Feign拦截器，负责在Header中添加JWT传递(AuthClient已自动注入)

Micro Service2：

* 服务启动时通过调用Auth-Server获取和初始化获取解密JWT所需要的公钥
* 订阅Auth Server定期发布的权限消息并缓存到服务本地的allowedClient

1. **初始化后一次服务调用的鉴权过程如下：**

Micro Service1：

通过Feign拦截器在请求Header中添加本地缓存的JWT传递给服务提供方

Micro Service2：

* 从Header中获取JWT
* 根据公钥验证JWT的签名
* 获取JWT中的信息：ClinetInfo
* 验证服务调用权限：

1. 带IgnoreClientToken标记的服务不拦截
2. 来源于isc-gate-web的请求不校验(gate已校验)
3. 来源于isc-gete-api的请求，验证服务类是否继承自BasicApi,防止非法调用
4. 其他服务来源的请求，验证服务类是否继承自BasicRest且具有访问权限(allowedClient中包含服务来源)

* 以上权限拦截校验在dev环境放开，方便调试(WebConfig中可查看)

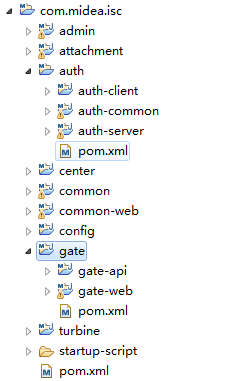
1. **黑盒设计，Auth-client包的封装**

所有微服务模块只需要了解服务鉴权的设计而不需要添加任何代码去支持这个设计，这个设计的实现由Auth-client包统一提供，且在springboot启动类中通过一个注解引入即可。

* auth-client通过注解EnableAuthClient，默认添加以上所有功能的初始化，微服务不需要做任何的代码添加
* 注入从配置文件获取当前服务名和secret的Bean
* 注入TokenManager，用户token的处理
* 注入服务调用日志的切面处理Bean
* 注入Feign的请求拦截器

# ISC微服务架构搭建的项目结构

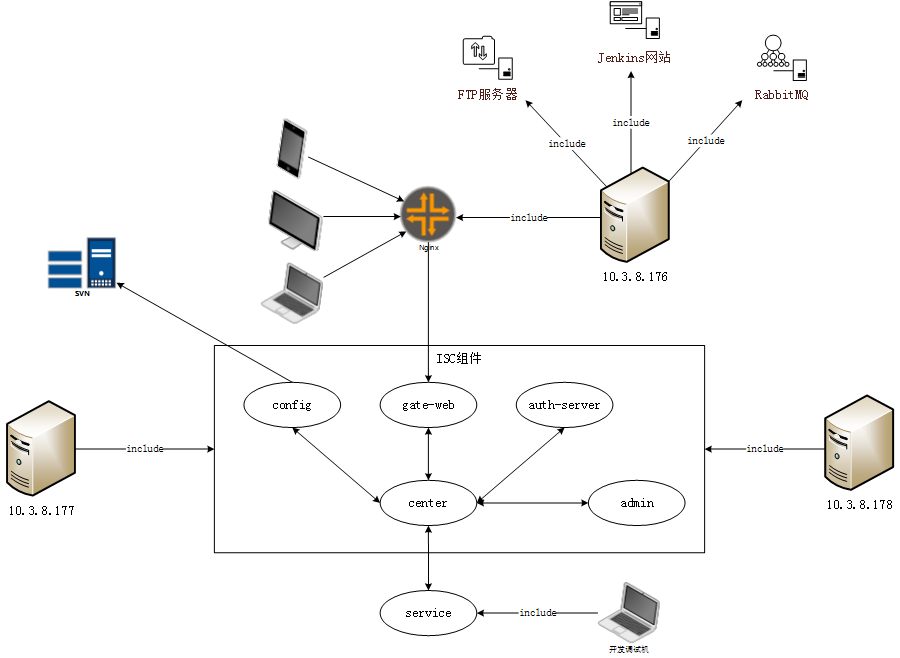
* 1. **IDE项目结构图**



* 1. **项目结构说明图**

****

# 开发调试环境部署图设计



在整个微服务架构体系中，存在统一的服务组件：注册中心、配置中心、网关、认证中心、管理平台、公共服务等。这些组件不需要在开发调试机部署，开发人员只需要关心自己的业务服务组件，但业务服务组件的开发又离不开统一服务组件的访问，因此需要将统一服务组件集中部署，使他们和业务服务组件共用一个注册中心以此可以达到互相访问的目的，并需确保我们的三台远程调试服务器与开发机网络互通(基于容器云的部署无法支持，容器云内部部署的容器为虚拟IP，注册中心的服务只能内部互通，外部无法访问)。

前端的请求统一通过nginx来分发跳转到gate-web，由gate-web将请求转发到指定服务组件。这里涉及多台开发机针对同一个应用注册到center进行调试，由于注册中心的负载均衡作用导致请求无法准确落入指定开发机进行调试，这里可以在开发阶段通过修改spring.application.name来唯一标志一个服务应用，同时修改前端调试的url来达到前后端联动的调试目的。

**服务器10.3.8.176包含以下功能：**

搭建FTP服务器：主要用于后续存放服务组件的jar包供容器云通过FTP方式拉取生成镜像

搭建Jenkins网站：主要用于持续集成ISC架构的服务组件

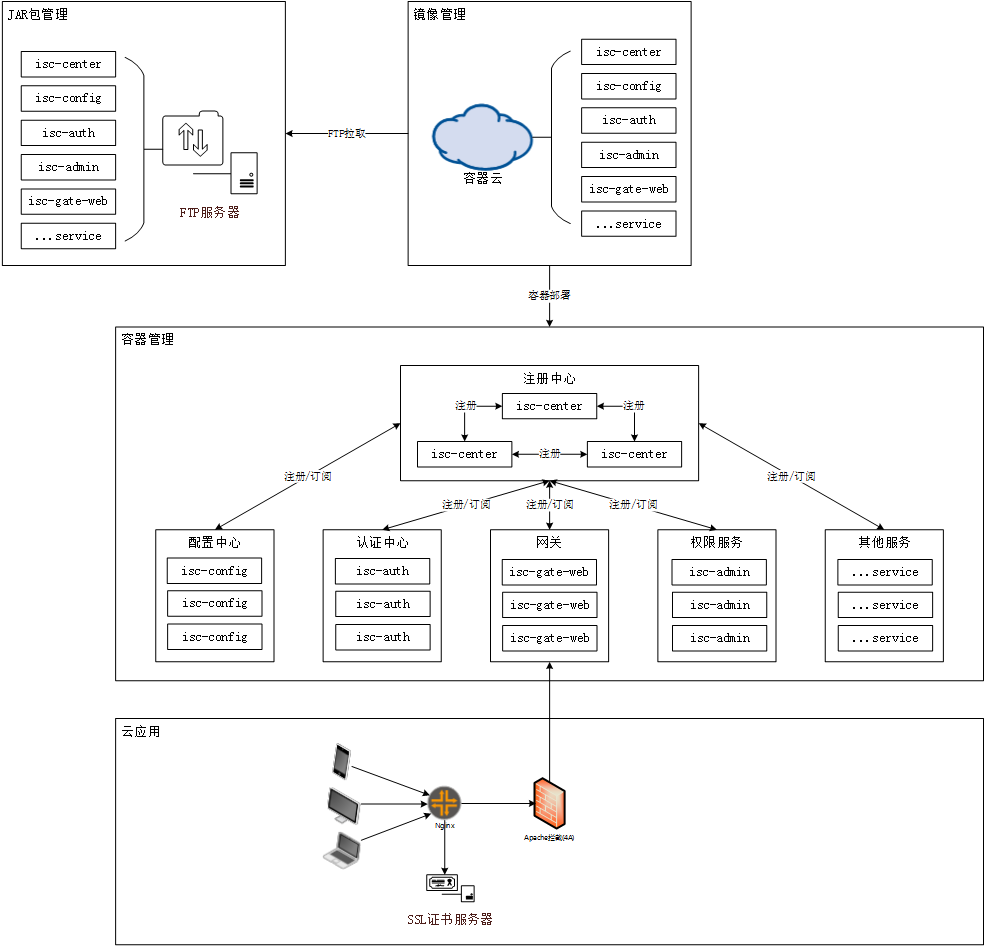
搭建RabbitMQ中间件：主要用于springcloud的消息总线和后续的消息队列服务组件

搭建Nginx容器：主要用于负载均衡到网关组件

**服务器10.3.8.177和服务器10.3.8.178：**

这两台主要作为统一服务组件的节点服务器

# 集成环境部署图设计



集成环境(SIT/UAT/VER/PRO)建议使用容器云进行部署，可以简单实现服务水平扩容，简化部署的过程，不用申请服务器，不用配置服务器的环境等等。容器云是由运维搭建的平台，使用docker和k8s等技术实现的容器管理平台。

整个部署过程主要分为四个模块：

JAR包管理：使用Jenkins持续集成应用服务的打包并存放到FTP服务器进行管理

镜像管理：通过FTP获取JAR包并在容器云上进行基础的dockerfile配置后生成镜像，有容器云统一管理

容器管理：通过镜像进行容器部署，每个服务默认部署三个容器做负载均衡处理

云应用：使用Httpd应用进行4A单点登录的拦截，使用Nginx应用进行前端请求的代理并作为访问入口进行证书的安装

# 其他

* 1. **注册中心的集群部署问题**
     1. host方式

需要配置主机host名称，不建议使用，但可以实现单机的集群

* + 1. IP Address方式

设置preferIpAddress: true，必须使用不同的IP作为host，同一个ip不行，意味着不能使用单机做集群

* 1. **注册中心的自我保护和心跳检测**

注册中心的自我保护：

默认情况下，当eureka server在一定时间内没有收到实例的心跳，便会把该实例从注册表中删除（默认是90秒），但是，如果短时间内丢失大量的实例心跳，便会触发eureka server的自我保护机制。

比如在开发测试时，需要频繁地重启微服务实例，但是我们很少会把eureka server一起重启（因为在开发过程中不会修改eureka注册中心），当一分钟内收到的心跳数大量减少时，会触发该保护机制。可以在eureka管理界面看到Renews threshold和Renews(last min)，当后者（最后一分钟收到的心跳数）小于前者（心跳阈值）的时候，触发保护机制，会出现红色的警告。

Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 会将这些实例保护起来，让这些实例不会过期，但是在保护期内如果服务刚好这个服务提供者非正常下线了，此时服务消费者就会拿到一个无效的服务实例，此时会调用失败，对于这个问题需要服务消费者端要有一些容错机制，如重试，断路器等。

**此时我们可以使用eureka.server.enable-self-preservation=false来关闭保护机制，这样可以确保注册中心中不可用的实例被及时的剔除，但是正式环境下不建议关闭保护机制**。

心跳检测问题：

在开发过程中，我们常常希望Eureka Server能够迅速有效地踢出已关停的节点，但是新手由于Eureka自我保护模式，以及心跳周期长的原因，常常会遇到Eureka Server不踢出已关停的节点的问题。解决方法如下：

* + - 1. Eureka Server端：配置关闭自我保护，并按需配置Eureka Server清理无效节点的时间间隔。
         1. eureka.server.enable-self-preservation # 设为false，关闭自我保护
         2. eureka.server.eviction-interval-timer-in-ms # 清理间隔（单位毫秒，默认是60\*1000）
      2. Eureka Client端：配置开启健康检查，并按需配置续约更新时间和到期时间
         1. eureka.client.healthcheck.enabled # 开启健康检查（需要spring-boot-starter-actuator依赖）
         2. eureka.instance.lease-renewal-interval-in-seconds # 续约更新时间间隔（默认30秒）
         3. eureka.instance.lease-expiration-duration-in-seconds # 续约到期时间（默认90秒）
  1. **配置中心热部署问题**

这个主要是实现配置项修改后，各个服务不用重启即可获取最新配置值的功能。

1. 单个服务更新配置文件

如果服务运行期间改变配置文件，服务是不会得到最新的配置信息，需要解决这个问题就需要引入Refresh。可以在服务的运行期间重新加载配置文件，实现步骤如下：

* 客户端pom添加依赖spring-boot-starter-actuator
* 客户端获取配置的类上添加@RefreshScope注解
* application.properties中配置如下暴露出所有actuator端点:

management.endpoints.web.exposure.include=\*

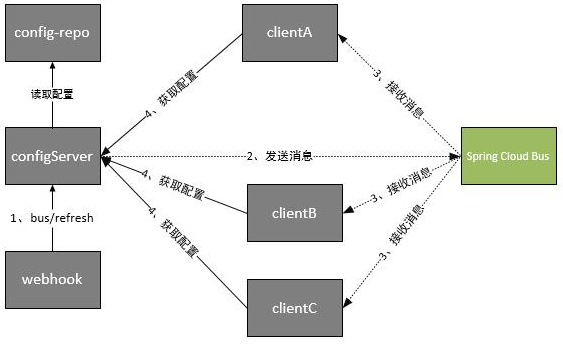
* 修改配置仓库中的相关客户端的配置项，调用相关客户端实例的/actuator/refresh接口，就实现了不重启应用的配置热更新
* 此种热更新和配置中心服务端没什么关系

此种方式缺点是：一般微服务实例会分布式部署的，服务实例会有多个，如果更改了客户端的某个配置，需要把客户端的/actuator/refresh接口挨个调用遍才可以，很显然是不合理的

1. spring cloud bus优化动态更新

Spring Cloud Bus 将分布式的节点用轻量的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。这里要讲述的是用Spring Cloud Bus实现通知微服务架构的配置文件的更改。

设计图如下：



实现步骤如下：

* 服务端pom添加依赖（使用rabbitmq，也可以使用kafka），

spring-boot-starter-actuator和

spring-cloud-starter-bus-amqp。

* application.properties里要配置相应的rabbitmq信息
* 配置服务启动后暴露出了actuator/bus-refresh端点(POST请求)，可由配置仓库自动触发或者由后台主动调用。

也可以指定刷新哪个服务/actuator/bus-refresh/customers:8000，这样消息总线上的微服务实例就会根据 destination 参数的值来判断是否需要要刷新。其中，customers:8000指的是各个微服务的 ApplicationContext ID。

destination 参数也可以用来定位特定的微服务。例如：

/actuator/bus-refresh/customers:\*\*，这样就可以触发 customers 微服务所有实例的配置刷新。

* 客户端pom文件同样添加spring-cloud-starter-bus-amqp依赖，

application.properties中同样添加rabbitmq配置信息

* 客户端获取配置的类上添加@RefreshScope注解，@Value的值就可以自动刷新
  1. **集成共享部门的网关**

共享部门eva平台的网关提供了灰度策略、流控策略，这些是目前的ISC微服务架构暂未研究实现的功能，同时也集成了4A单点登录功能，固考虑使用共享部门的网关功能。

相比于现有架构的网关，缺少了统一swagger接口文档的查看、统一登录信息获取和权限校验，统一请求头usertoken、clienttoken、app封装传递，统一profile过期刷新等功能。

统一swagger接口文档的查看：属于扩展功能，不包含不影响总体架构

统一登录信息获取和权限校验：需要对后台框架进行适配，将用户token的获取放到兼容到具体服务的包中

统一请求头usertoken、app、clienttoken封装传递：需要对后台框架进行适配，其中usertoken、app可以通过代码兼容， clienttoken是用来获取服务请求的来源，当前架构中的服务都会有clienttoken，但共享的网关没有这部分信息，目前共享正在修改

统一profile过期刷新：已将profile的过期时间统一由权限中心控制并保存在profile中，每次用户请求后，由具体服务通过common-web包中的拦截器获取profile中的过期时长进行更新。

**当前适配的原则是保持网关的无缝切换，即同时支持自有网关和共享网关。**

启用共享网关且打开登录校验的用户认证流程如图：



启用共享网关但不打开登录校验的用户认证流程：

* + 1. 前端校验非单点登录时会跳转到自有登录界面，登录时通过调用auth/weblogin实现登录并获取token
    2. 用户请求时将在header中添加token，后台已兼容从header中获取token来获取用户信息

# 版本修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 作者 | 内容 | 版本 |
| 2019-01-18 | 王孝坤 | ISC微服务架构设计说明 | V1.0 |
| 2019-02-22 | 王孝坤 | 补充说明配置中心热部署的设计 | V1.1 |
| 2019-03-07 | 王孝坤 | 添加共享网关适配的说明 | V1.2 |
| 2019-03-08 | 王孝坤 | 添加注册中心的自我保护和心跳检测说明 | V1.3 |
| 2019-04-22 | 王孝坤 | 添加断路器聚合监控的相关说明 | V1.4 |