

## Nacos - 服务发现和配置管理

### 教学目标

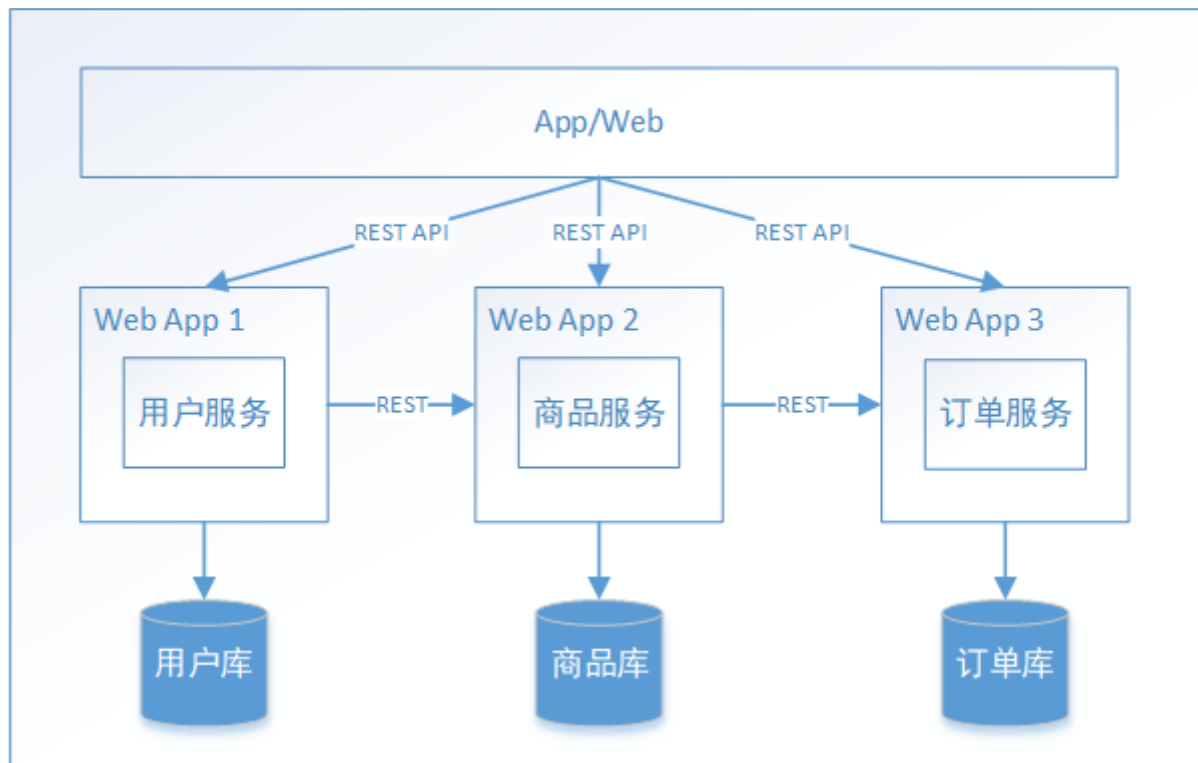
- 1) 能够理解微服务架构的特点
- 2) 能够理解服务发现的流程
- 3) 能够说出Nacos的功能
- 4) 掌握Nacos的安装方法
- 5) 掌握RESTful服务发现开发方法
- 6) 掌握Dubbo服务发现开发方法
- 7) 理解Nacos服务发现的数据模型
- 8) 能够掌握Nacos配置管理方法
- 9) 掌握Nacos扩展配置方法

## 1 理解服务发现

### 1.1 微服务架构

为适应企业的业务发展，提高软件研发的生产力，降低软件研发的成本，软件架构也作了升级和优化，将一个独立的系统拆分成若干小的服务，每个小服务运行在不同的进程中，服务与服务之间采用RESTful、RPC等协议传输数据，每个服务所拥有的功能具有独立性强的特点，这样的设计就实现了单个服务的高内聚，服务与服务之间的低耦合效果，这些小服务就是微服务，基于这种方法设计的系统架构即微服务架构。

下图是基于微服务架构的电商系统：



特点：

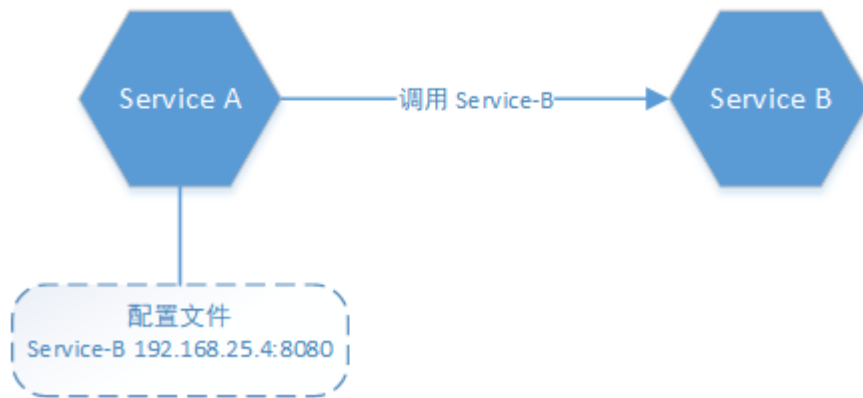
- 1、服务层按业务拆分为一个一个的微服务。
- 2、微服务的职责单一。
- 3、微服务之间采用RESTful、RPC等轻量级协议传输。
- 4、有利于采用前后端分离架构。

## 1.2 理解服务发现

### 1.2.1 测试环境

在微服务架构中，整个系统会按职责能力划分为多个服务，通过服务之间协作来实现业务目标。这样在我们的代码中免不了要进行服务间的远程调用，服务的消费方要调用服务的生产方，为了完成一次请求，消费方需要知道服务生产方的网络位置(IP地址和端口号)。

我们的代码可以通过读取配置文件的方式读取服务生产方网络位置，如下：



我们通过Spring boot技术很容易实现：

## 1、创建nacos-discovery父工程

pom.xml如下：

```
<?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>com.itheima.nacos</groupId>
    <artifactId>nacos-discovery</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.1.3.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

</project>
```

## 2、Service B（服务生产者）

创建服务提供者 nacos-restful-provider。

pom.xml如下：

```
<?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">
  <parent>
    <artifactId>nacos-discovery</artifactId>
    <groupId>com.itheima.nacos</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>nacos-restful-provider</artifactId>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

  </dependencies>

</project>
```

Service B是服务的生产方，暴露/service服务地址，实现代码如下：

### 1、创建Controller

```
package com.itheima.nacos.provider.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class RestProviderController {
    @GetMapping(value = "/service") //暴露服务
    public String service(){
        System.out.println("provider invoke");
        return "provider invoke";
    }
}
```

### 2、创建启动类

```
@SpringBootApplication
public class SpringRestProviderBootstrap {
    public static void main(String[] args) {
        SpringApplication.run(SpringRestProviderBootstrap.class, args);
    }
}
```

配置文件：

创建application.yml，内容如下：

```
server :  
    port : 56010
```

### 3、Service A ( 服务消费者 )

创建nacos-restful-consumer 服务消费工程。

pom.xml如下：

```
<?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">  
    <parent>  
        <artifactId>nacos-discovery</artifactId>  
        <groupId>com.itheima.nacos</groupId>  
        <version>1.0-SNAPSHOT</version>  
    </parent>  
    <modelVersion>4.0.0</modelVersion>  
  
    <artifactId>nacos-restful-consumer</artifactId>  
  
    <dependencies>  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-web</artifactId>  
        </dependency>  
    </dependencies>  
  
</project>
```

实现代码：

#### 1、创建controller

```
package com.itheima.nacos.consumer.controller;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
import org.springframework.web.client.RestTemplate;  
  
@RestController  
public class RestConsumerController {  
    @Value("${provider.address}")  
    private String providerAddress;  
  
    @GetMapping(value = "/service")
```

```
public String service(){
    RestTemplate restTemplate = new RestTemplate();
    //调用服务
    String providerResult = restTemplate.getForObject("http://" + providerAddress +
"/service",String.class);
    return "consumer invoke | " + providerResult;
}

}
```

## 2、创建启动类

```
@SpringBootApplication
public class SpringRestConsumerBootstrap {
    public static void main(String[] args) {
        SpringApplication.run(SpringRestConsumerBootstrap.class, args);
    }
}
```

配置文件：

创建application.yml，内容如下：

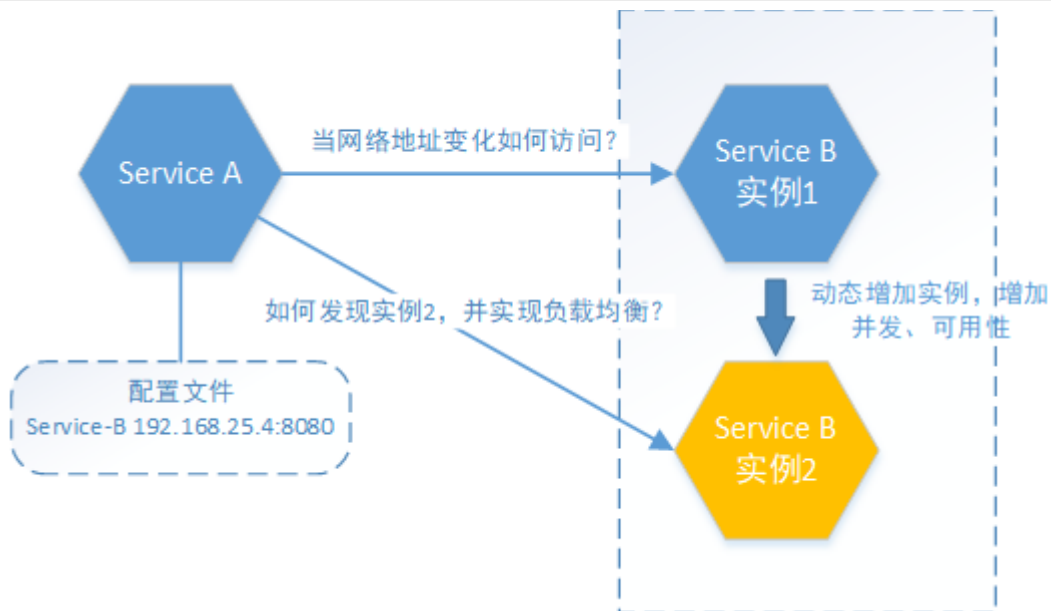
```
server.port = 56020
# 服务生产方地址
provider.address = 127.0.0.1:56010
```

访问<http://127.0.0.1:56020/service>，输出以下内容：

```
consumer invoke | provider invoke
```

### 1.2.2 服务发现流程

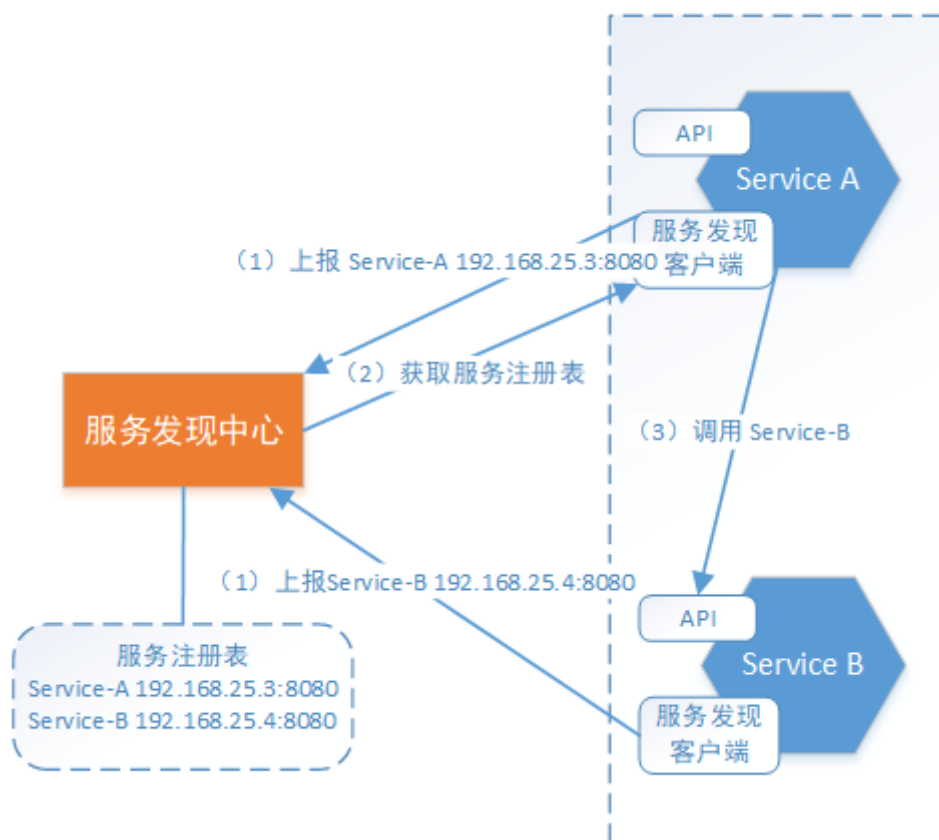
上边的例子看上去很完美，但是，仔细考虑以下，此方案对于微服务应用而言行不通。首先，微服务可能是部署在云环境的，服务实例的网络位置或许是动态分配的。另外，每一个服务一般会有多个实例来做负载均衡，由于宕机或升级，服务实例网络地址会经常动态改变。再者，每一个服务也可能应对临时访问压力增加新的服务节点。正如下图所示：



基于以上的问题，服务之间如何相互发现？服务如何管理？这就是服务发现的问题了。

服务发现就是服务消费方通过服务发现中心智能发现服务提供方，从而进行远程调用的过程。

如下图：



上图中服务实例本身并不记录服务生产方的网络地址，所有服务实例内部都会包含**服务发现客户端**。

(1) 在每个服务启动时会向**服务发现中心**上报自己的网络位置。这样，在服务发现中心内部会形成一个**服务注册表**，**服务注册表**是服务发现的核心部分，是包含所有服务实例的网络地址的数据库。

(2) 服务发现客户端会定期从服务发现中心同步服务注册表，并缓存在客户端。

(3) 当需要对某服务进行请求时，服务实例通过该注册表，定位目标服务网络地址。若目标服务存在多个网络地址，则使用负载均衡算法从多个服务实例中选择出一个，然后发出请求。

总结，在微服务环境中，由于服务运行实例的网络地址是不断动态变化的，服务实例数量的动态变化，因此无法使用固定的配置文件来记录服务提供方的网络地址，必须使用动态的服务发现机制用于实现微服务间的相互感知。各服务实例会上报自己的网络地址，这样服务中心就形成了一个完整的服务注册表，各服务实例会通过服务发现中心来获取访问目标服务的网络地址，从而实现服务发现的机制。

## 2 Nacos 服务发现

### 2.1 Nacos简介

#### 2.1.1 服务发现产品对比

目前市面上用的比较多的服务发现中心有：Nacos、Eureka、Consul和Zookeeper。

对比项目	Nacos	Eureka	Consul	Zookeeper
一致性协议	支持AP和CP模型	AP模型	CP模型	CP模型
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	Keep Alive
负载均衡策略	权重/metadata/Selector	Ribbon	Fabio	-
雪崩保护	有	有	无	无
自动注销实例	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	TCP
监听支持	支持	支持	支持	支持
多数据中心	支持	支持	支持	不支持
跨注册中心同步	支持	不支持	支持	不支持
SpringCloud集成	支持	支持	支持	不支持
Dubbo集成	支持	不支持	不支持	支持
k8s集成	支持	不支持	支持	不支持

从上面对比可以了解到，Nacos作为服务发现中心，具备更多的功能支持项，且从长远来看Nacos在以后的版本会支持SpringCloud+Kubernetes的组合，填补 2 者的鸿沟，在两套体系下可以采用同一套服务发现和配置管理的解决方案，这将大大的简化使用和维护的成本。另外，Nacos 计划实现 Service Mesh，也是未来微服务发展的趋势。

#### 2.1.2 Nacos简介



Nacos是阿里的一个开源产品，它是针对微服务架构中的服务发现、配置管理、服务治理的综合型解决方案。



官方介绍是这样的：

Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您实现动态服务发现、服务配置管理、服务及流量管理。Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos 是构建以“服务”为中心的现代应用架构的服务基础设施。

官网地址：<https://nacos.io>

### 2.1.3 Nacos特性

Nacos主要提供以下四大功能：

#### 1. 服务发现与服务健康检查

Nacos使服务更容易注册，并通过DNS或HTTP接口发现其他服务，Nacos还提供服务的实时健康检查，以防止向不健康的主机或服务实例发送请求。

#### 2. 动态配置管理

动态配置服务允许您在所有环境中以集中和动态的方式管理所有服务的配置。Nacos消除了更新配置时重新部署应用程序，这使配置的更改更加高效和灵活。

#### 3. 动态DNS服务

Nacos提供基于DNS 协议的服务发现能力，旨在支持异构语言的服务发现，支持将注册在Nacos上的服务以域名的方式暴露端点，让三方应用方便的查阅及发现。

#### 4. 服务和元数据管理

Nacos 能让您从微服务平台建设的视角管理数据中心的所有服务及元数据，包括管理服务的描述、生命周期、服务的静态依赖分析、服务的健康状态、服务的流量管理、路由及安全策略。

这里1、3、4说明了服务发现的功能特性。

## 2.2 安装Nacos Server

### 2.2.1 预备环境准备

Nacos 依赖 [Java](#) 环境来运行。如果您是从代码开始构建并运行Nacos，还需要为此配置 [Maven](#)环境，请确保是在以下版本环境中安装使用：

1. 64 bit OS，支持 Linux/Unix/Mac/Windows，推荐选用 Linux/Unix/Mac。
2. 64 bit JDK 1.8+；[下载](#) & [配置](#)。
3. Maven 3.2.x+；[下载](#) & [配置](#)。

### 2.2.2 下载源码或者安装包

你可以通过源码和发行包两种方式来获取 Nacos。

#### 从 Github 上下载源码方式

```
git clone https://github.com/alibaba/nacos.git
cd nacos/
mvn -Prelease-nacos clean install -U
ls -al distribution/target/

// change the $version to your actual path
cd distribution/target/nacos-server-$version/nacos/bin
```

## 下载编译后压缩包方式

您可以从 [最新稳定版本](#) 下载 `nacos-server-$version.zip` 包，本教程使用nacos-server-1.1.3版本。

下载地址：<https://github.com/alibaba/nacos/releases>

下载后解压：

```
unzip nacos-server-$version.zip 或者 tar -xvf nacos-server-$version.tar.gz
cd nacos/bin
```

## 2.2.3 启动服务器

nacos的默认端口是8848，需要保证8848默认端口没有被其他进程占用。

进入安装程序的bin目录：

**Linux/Unix/Mac启动方式：**

启动命令(standalone代表着单机模式运行，非集群模式)：

```
sh startup.sh -m standalone
```

如果您使用的是ubuntu系统，或者运行脚本报错提示[[符号找不到，可尝试如下运行：

```
bash startup.sh -m standalone
```

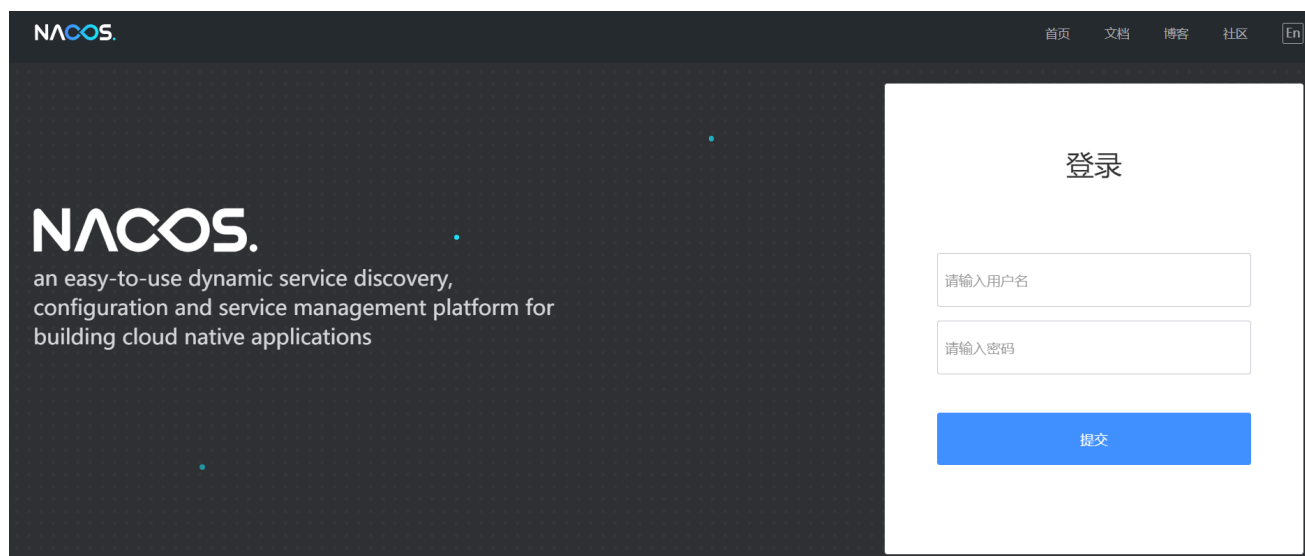
**Windows启动方式：**

启动命令：

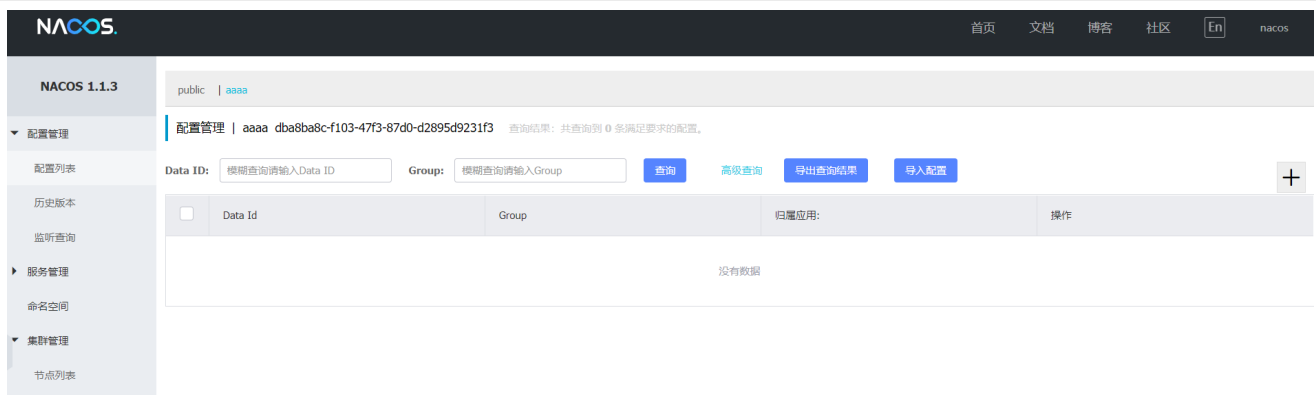
```
cmd startup.cmd
```

或者双击startup.cmd运行文件。

启动成功，可通过浏览器访问 <http://127.0.0.1:8848/nacos>，打开如下nacos控制台登录页面：



使用默认用户名：nacos，默认密码：nacos 登录即可打开主页面。



## 2.2.4 外部mysql数据库支持

单机模式时nacos默认使用嵌入式数据库实现数据的存储，若想使用外部mysql存储nacos数据，需要进行以下步骤：

- 1. 安装数据库，版本要求：5.6.5+，mysql 8 以下
- 2. 初始化mysql数据库，新建数据库nacos\_config，数据库初始化文件：\${nacoshome}/conf/nacos-mysql.sql
- 3. 修改\${nacoshome}/conf/application.properties文件，增加支持mysql数据源配置（目前只支持mysql），添加mysql数据源的url、用户名和密码。

```
spring.datasource.platform=mysql

db.num=1
db.url.0=jdbc:mysql://11.162.196.16:3306/nacos_config?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
db.user=nacos_devtest
db.password=youdontknow
```

## 2.3 RESTful服务发现

### 2.3.1 测试环境

Spring Cloud是一套微服务开发框架集合，包括微服务开发的方方面面，Spring Cloud是一套微服务开发的标准，集成了很多优秀的开源框架，比如有名的Netflix公司的众多项目。

Spring Cloud 项目地址：<https://spring.io/projects/spring-cloud>

本测试环境采用阿里开源的Spring Cloud Alibaba微服务开发框架，Spring Cloud Alibaba是阿里巴巴公司基于Spring Cloud标准实现一套微服务开发框架集合，它和Netflix一样都是Spring Cloud微服务开发实现方案。

Spring Cloud Alibaba项目地址：<https://github.com/alibaba/spring-cloud-alibaba>

通过Spring Cloud Alibaba实现解决：

- 1、服务发现客户端从服务发现中心获取服务列表
- 2、服务消费方通过负载均衡获取服务地址

在nacos-discovery父工程中添加依赖管理

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.1.3.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Greenwich.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2.1.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

分别在服务提供及服务消费工程中添加依赖，此依赖的作用是服务发现

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

### 2.3.2 服务注册

在服务提供工程中配置nacos服务发现相关的配置：

服务提供：

```
spring:
  application:
    name: nacos-restful-provider
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

启动nacos

启动服务提供

观察nacos服务列表，nacos-restful-provider注册成功



服务名称：每个服务在服务注册中心的标识，相当于Java中的类名。

服务实例：网络中提供服务的实例，具有IP和端口，相当于Java中的对象，一个实例即为运行在服务器上的一个进程。

### 2.3.3 服务发现

在服务消费工程中配置nacos服务发现相关的配置：

服务消费：

```
spring:
  application:
    name: nacos-restful-consumer
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

修改Controller中远程调用的代码：

```
//服务id即注册中心中的服务名
private String serviceId="nacos-restful-provider";

@Autowired
LoadBalancerClient loadBalancerClient;

@GetMapping(value = "/service")
public String service(){
    RestTemplate restTemplate = new RestTemplate();
    //调用服务
    String providerResult = restTemplate.getForObject("http://" + providerAddress +
"/service",String.class);
    ServiceInstance serviceInstance = loadBalancerClient.choose(serviceId);
    URI uri = serviceInstance.getUri();
    String providerResult = restTemplate.getForObject(uri+"/service",String.class);
    return "consumer invoke | " + providerResult;
}
```

执行流程：

- 1、服务提供方将自己注册到服务注册中心
- 2、服务消费方从注册中心获取服务地址

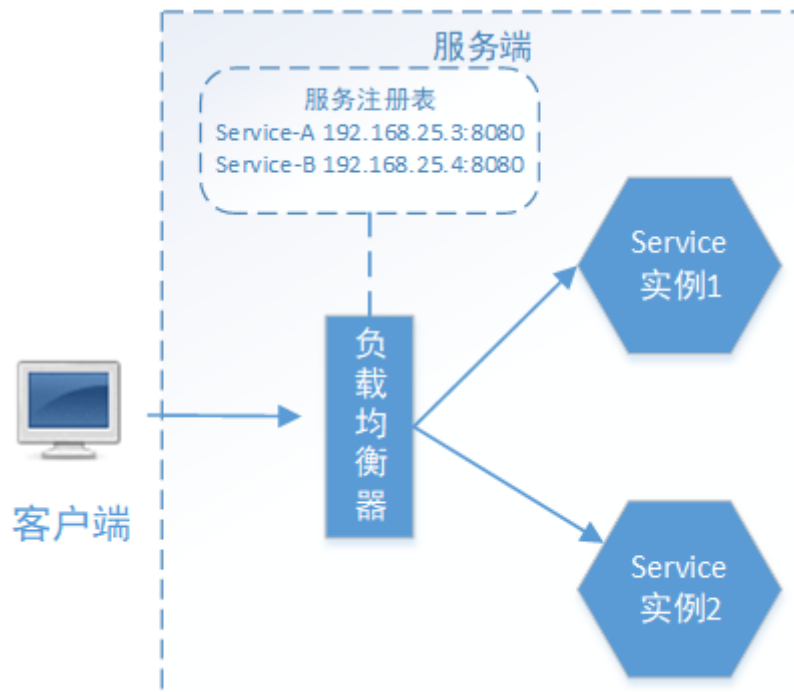
### 3、进行远程调用

#### 2.3.4 负载均衡

在RESTful服务发现的流程中，ServiceA通过负载均衡调用ServiceB，下边来了解一下**负载均衡**：

**负载均衡**就是将用户请求（流量）通过一定的策略，分摊在多个服务实例上执行，它是系统处理高并发、缓解网络压力和进行服务端扩容的重要手段之一。它分为**服务端负载均衡**和**客户端负载均衡**。

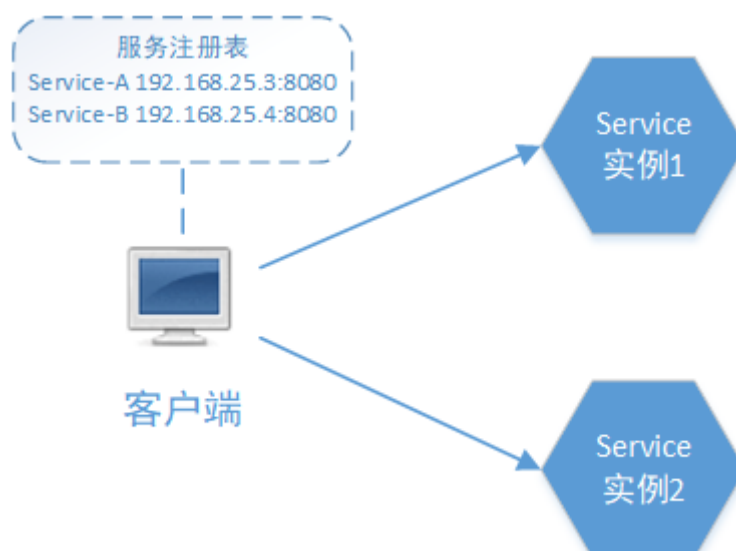
**服务器端负载均衡：**



在负载均衡器中维护一个可用的服务实例清单，当客户端请求来临时，负载均衡服务器按照某种配置好的规则(**负载均衡算法**)从可用服务实例清单中选取其一去处理客户端的请求。这就是服务端负载均衡。

例如Nginx，通过Nginx进行负载均衡，客户端发送请求至Nginx，Nginx通过负载均衡算法，在多个服务器之间选择一个进行访问。即在服务器端再进行负载均衡算法分配。

**客户端服务负载均衡：**



上边使用的LoadBalancerClient就是一个客户端负载均衡器，具体使用的是Ribbon客户端负载均衡器。Ribbon在发送请求前通过负载均衡算法选择一个服务实例，然后进行访问，这是客户端负载均衡。即在客户端就进行负载均衡的分配。

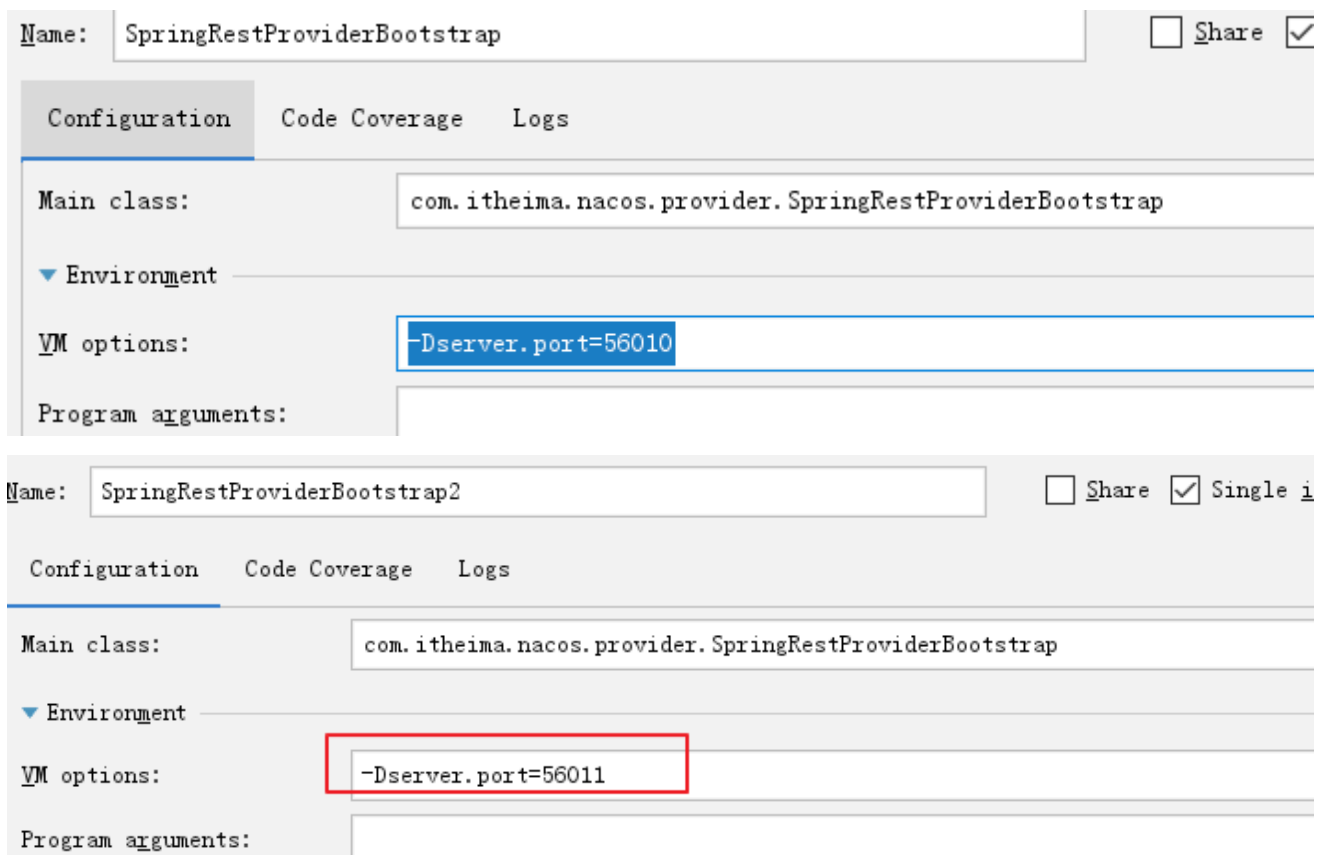
Ribbon是一个客户端负载均衡器，它的责任是从一组实例列表中挑选合适的实例，如何挑选？取决于**负载均衡策略**。

Ribbon核心组件IRule是负载均衡策略接口，它有如下实现，大家仅做了解：

- RoundRobinRule(默认):轮询，即按一定的顺序轮换获取实例的地址。
- RandomRule:随机，即以随机的方式获取实例的地址。
- AvailabilityFilteringRule: 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务,以及并发的连接数量超过阈值的服务,然后对剩余的服务列表按照轮询策略进行访问;
- WeightedResponseTimeRule: 根据平均响应时间计算所有服务的权重,响应时间越快,服务权重越大,被选中的机率越高;  
刚启动时,如果统计信息不足,则使用RoundRobinRule策略,等统计信息足够时,会切换到WeightedResponseTimeRule
- RetryRule: 先按照RoundRobinRule的策略获取服务,如果获取服务失败,则在指定时间内会进行重试,获取可用的服务;
- BestAvailableRule: 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务,然后选择一个并发量最小的服务;
- ZoneAvoidanceRule: 默认规则,复合判断server所在区域的性能和server的可用性选择服务器;

#### 准备测试环境：

启动多个服务提供方进程，为保证端口不冲突，通过启动参数配置端口，并启动这两个进程。



The image shows two IDE configuration windows for a project named 'SpringRestProviderBootstrap'. Both windows have tabs for 'Configuration', 'Code Coverage', and 'Logs'. The 'Configuration' tab is selected in both.

**Top Window:**

- Name: SpringRestProviderBootstrap
- Share: ☐ (unchecked)
- Main class: com.itheima.nacos.provider.SpringRestProviderBootstrap
- Environment: (expanded)
- VM options: -Dserver.port=56010
- Program arguments: (empty)

**Bottom Window:**

- Name: SpringRestProviderBootstrap2
- Share: ☐ (unchecked)
- Single instance: ☒ (checked)
- Main class: com.itheima.nacos.provider.SpringRestProviderBootstrap
- Environment: (expanded)
- VM options: -Dserver.port=56011
- Program arguments: (empty)

可通过下面方式在服务消费方的 配置文件中修改默认的负载均衡策略：

```
nacos-restful-provider:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

`nacos-restful-provider` 即服务提供方的服务名称。

`com.netflix.loadbalancer.RandomRule`：负载均衡类路径。

### 2.3.5 小结

服务注册与发现流程：

- 1、服务提供方将自己注册到服务注册中心
- 2、服务消费方从注册中心获取服务地址
- 3、通过客户端负载均衡器进行远程调用

## 2.4 Dubbo服务发现

Dubbo是阿里巴巴公司开源的RPC框架，在国内有着非常大的用户群体，但是其微服务开发组件相对Spring Cloud来说并不那么完善。

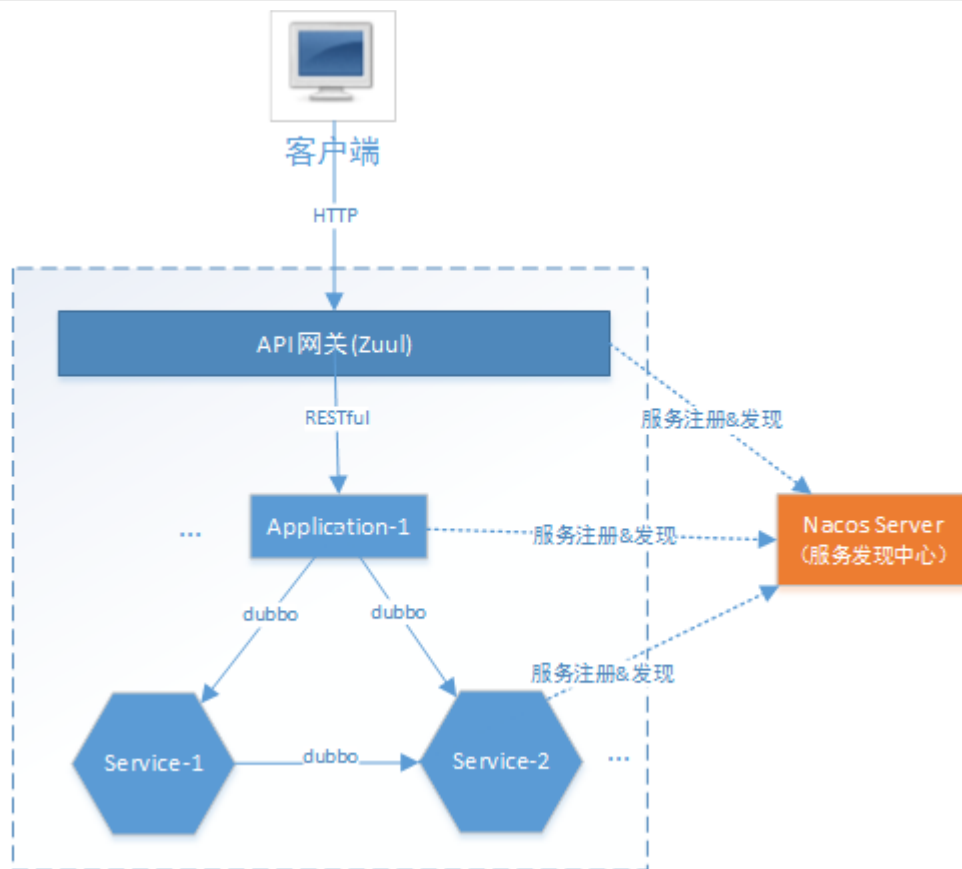
Spring Cloud Alibaba微服务开发框架集成了Dubbo，可实现微服务对外暴露Dubbo协议的接口，Dubbo协议相比RESTful协议速度更快。

RPC：RPC是远程过程调用（Remote Procedure Call）的缩写形式，调用RPC远程方法就像调用本地方法一样，非常方便，后边案例讲解具体过程。

### 2.4.1 Dubbo服务架构

下图是微服务采用Dubbo协议的系统架构图：





组件说明：

- 1、客户端：前端或外部系统
- 2、API网关：系统唯一入口，路由转发
- 3、application-1：应用1，前端提供Http接口，接收用户的交互请求
- 4、service-1：微服务1，提供业务逻辑处理服务
- 5、service-2：微服务2，提供业务逻辑处理服务

交互流程：

- 1、网关负责客户端请求的统一入口，路由转发，前端通过网关请求后端服务。
- 2、网关收到前端请求，转发请求给应用。
- 3、应用接收前端请求，调用微服务进行业务逻辑处理
- 4、微服务为应用提供业务逻辑处理的支撑，为应用提供Dubbo协议接口

优势分析：

此架构同时提供RESTful和Dubbo接口服务，应用层对前端提供RESTful接口，RESTful是互联网通用的轻量级交互协议，方便前端接入系统；微服务层向应用层提供Dubbo接口，Dubbo接口基于RPC通信协议速度更快。

本架构采用阿里开源的Nacos，集服务发现和配置中心于一身，支持RESTful及Dubbo服务的注册。

## 2.4.2 测试环境

父工程：仍然使用nacos-discovery。

application1：使用nacos-restful-consumer。

service1微服务：需要新建

service2微服务：需要新建

api网关：后边的课程内容讲解。

## 2.4.3 service2微服务

service2对外暴露dubbo协议的接口，考虑远程接口可能会被其它多个服务调用，这里将service2的接口单独抽出api工程，service2微服务工程的结构如下：



service2-api：存放接口，独立成一个工程方便被其它服务工程依赖。

service2-server：存放接口实现，即dubbo服务的实现部分。

### 2.4.3.1 定义service2-api

#### 1、创建service2工程

```
<?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">
    <parent>
        <artifactId>nacos-discovery</artifactId>
        <groupId>com.itheima.nacos</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>nacos-dubbo-service2</artifactId>
    <packaging>pom</packaging>

</project>
```

#### 2、创建service2-api工程

```
<?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">
        <parent>
            <artifactId>nacos-dubbo-service1</artifactId>
            <groupId>com.itheima.nacos</groupId>
            <version>1.0-SNAPSHOT</version>
        </parent>
        <modelVersion>4.0.0</modelVersion>

        <artifactId>service2-api</artifactId>

    </project>
```

### 3、定义接口

```
package com.itheima.microservice.service2.api;

public interface Service2Api {
    public String dubboService2();
}
```

#### 2.4.3.2 定义service2-server

##### 1、创建service2-server工程

```
<?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">
    <parent>
        <artifactId>nacos-dubbo-service1</artifactId>
        <groupId>com.itheima.nacos</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>service2-server</artifactId>
    <dependencies>
        <dependency>
            <groupId>com.itheima.nacos</groupId>
            <artifactId>service2-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        </dependency>
        <dependency>

            <groupId>com.alibaba.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-dubbo</artifactId>
</dependency>
</dependencies>

</project>
```

## 2、定义接口实现

注意：使用@org.apache.dubbo.config.annotation.Service标记dubbo服务

```
package com.itheima.microservice.service2.service;

import com.itheima.microservice.service2.api.Service2Api;

@org.apache.dubbo.config.annotation.Service
public class Service2ApiImpl implements Service2Api {

    public String dubboService2() {
        return "dubboService2";
    }
}
```

## 3、定义启动类

```
package com.itheima.microservice.service2;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Service2Bootstrap {
    public static void main(String[] args) {
        SpringApplication.run(Service2Bootstrap.class, args);
    }
}
```

## 4、定义配置文件bootstrap.yml

```
server:
  port: 56040 #启动端口 命令行注入

spring:
  application:
    name: dubbo-service2
  main:
    allow-bean-definition-overriding: true # Spring Boot 2.1 需要设定
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

```
dubbo:
  scan:
    # dubbo 服务扫描基准包
    base-packages: com.itheima.microservice.service2.service
  protocol:
    # dubbo 协议
    name: dubbo
    # dubbo 协议端口
    port: 20891
  registry:
    address: nacos://127.0.0.1:8848
  application:
    qos-enable: false #dubbo运维服务是否开启
  consumer:
    check: false #启动时就否检查依赖的服务
```

## 5、启动service2-server

启动成功观察nacos的服务列表

## 6、bootstrap.yml配置说明

bootstrap.yml内容中以dubbo开头的为dubbo服务 的配置：

- `dubbo.scan.base-packages`：指定 Dubbo 服务实现类的扫描基准包，将 `@org.apache.dubbo.config.annotation.Service` 注解标注的 service 暴露为 dubbo 服务。
- `dubbo.protocol`：Dubbo 服务暴露的协议配置，其中子属性 `name` 为协议名称，`port` 为 dubbo 协议端口可以指定多协议，如：`dubbo.protocol.rmi.port=1099`
- `dubbo.registry`：Dubbo 服务注册中心配置，其中子属性 `address` 的值 `"nacos://127.0.0.1:8848"`，说明 dubbo 服务注册到 nacos 相当于原生 dubbo 的 xml 配置中的 `<dubbo:registry address="10.20.153.10:9090" />`

bootstrap.yml 内容的上半部分为 Spring Cloud 的相关配置：

- `spring.application.name`：Spring 应用名称，用于 Spring Cloud 服务注册和发现。

该值在 Dubbo Spring Cloud 加持下被视作 `dubbo.application.name`，因此，无需再显式地配置 `dubbo.application.name`

- `spring.cloud.nacos.discovery`：Nacos 服务发现与注册配置，其中子属性 `server-addr` 指定 Nacos 服务器主机和端口

## 2.4.4 application1调用service2

根据 dubbo 服务的架构图，本章节将 nacos-restful-consumer 作为 application1，实现 application1 调用 service2。

### 2.4.4.1 引用service2

在nacos-restful-consumer工程中引用service2依赖

在pom.xml中引入service2-api的依赖

```
<dependency>
  <groupId>com.itheima.nacos</groupId>
  <artifactId>service2-api</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

引入 spring-cloud-starter-dubbo依赖，它会根据接口生成代理对象

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-dubbo</artifactId>
</dependency>
```

#### 2.4.4.2 实现远程调用

修改nacos-restful-consumer工程的RestConsumerController：

```
@org.apache.dubbo.config.annotation.Reference
private Service2Api service2Api;

@GetMapping(value = "/service2")
public String service2(){
    //远程调用service2
    String providerResult = service2Api.dubboService2();
    return "consumer dubbo invoke | " + providerResult;
}
```

Note：注意：这里的 @Reference 注解是 org.apache.dubbo.config.annotation.Reference

测试：

请求：<http://127.0.0.1:56020/service2>

显示：consumer dubbo invoke | dubboService2 表明service2调用成功。

### 2.4.5 service1微服务

service1采用和service2相同的工程结构。

本节实现service1对外暴露dubbo接口，并用实现service1调用service2。

#### 2.4.5.1 定义service1-api

1、创建service1工程

```
<?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">
  <parent>
    <artifactId>nacos-discovery</artifactId>
    <groupId>com.itheima.nacos</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>nacos-dubbo-service1</artifactId>
  <packaging>pom</packaging>

</project>
```

## 2、创建service1-api工程

```
<?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">
  <parent>
    <artifactId>nacos-dubbo-service1</artifactId>
    <groupId>com.itheima.nacos</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>service1-api</artifactId>

</project>
```

## 3、定义接口

```
package com.itheima.microservice.service1.api;

public interface Service1Api {
    public String dubboService1();
}
```

### 2.4.5.2 定义service1-server

#### 1、创建service1-server工程

由于实现service1调用service2，这里需要引入 service2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">
  <parent>
    <artifactId>nacos-dubbo-service1</artifactId>
    <groupId>com.itheima.nacos</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>service1-server</artifactId>
  <dependencies>
    <dependency>
      <groupId>com.itheima.nacos</groupId>
      <artifactId>service1-api</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.itheima.nacos</groupId>
      <artifactId>service2-api</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-starter-dubbo</artifactId>
    </dependency>
  </dependencies>
</project>
```

## 2、定义接口实现

```
package com.itheima.microservice.service1.service;

import com.itheima.microservice.service1.api.Service1Api;
import com.itheima.microservice.service2.api.Service2Api;
import org.apache.dubbo.config.annotation.Reference;

@org.apache.dubbo.config.annotation.Service
public class Service1ApiImpl implements Service1Api {

    @Reference
    Service2Api service2Api;

    public String dubboService1() {
        String s = service2Api.dubboService2();
    }
}
```





```
        return "dubboService1|"+s;  
    }  
}
```

### 3、定义启动类

```
package com.itheima.microservice.service1;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class Service1Bootstrap {  
    public static void main(String[] args) {  
        SpringApplication.run(Service1Bootstrap.class, args);  
    }  
}
```

### 4、定义配置文件bootstrap.yml

```
server:  
  port: 56030 #启动端口 命令行注入  
  
spring:  
  application:  
    name: dubbo-service1  
  main:  
    allow-bean-definition-overriding: true # Spring Boot 2.1 需要设定  
  cloud:  
    nacos:  
      discovery:  
        server-addr: 127.0.0.1:8848  
  dubbo:  
    scan:  
      # dubbo 服务扫描基准包  
      base-packages: com.itheima.microservice.service1.service  
    protocol:  
      # dubbo 协议  
      name: dubbo  
      # dubbo 协议端口  
      port: 20881  
    registry:  
      address: nacos://127.0.0.1:8848  
    application:  
      qos-enable: false #dubbo运维服务是否开启  
    consumer:  
      check: false #启动时就否检查依赖的服务
```

### 5、启动service1-server

启动成功观察nacos的服务列表

### 2.4.5.3 application1调用service1

参考 application1调用service2的方法实现。

1、在application1引入 service1-api的依赖

略

2、在application1的controller中调用service1接口

```
@Reference
private Service1Api service1Api;

@GetMapping(value = "/service3")
public String service1(){
    String providerResult = service1Api.dubboService1();
    return "consumer dubbo invoke | " + providerResult;
}
```

3、测试，请求<http://127.0.0.1:56020/service3>

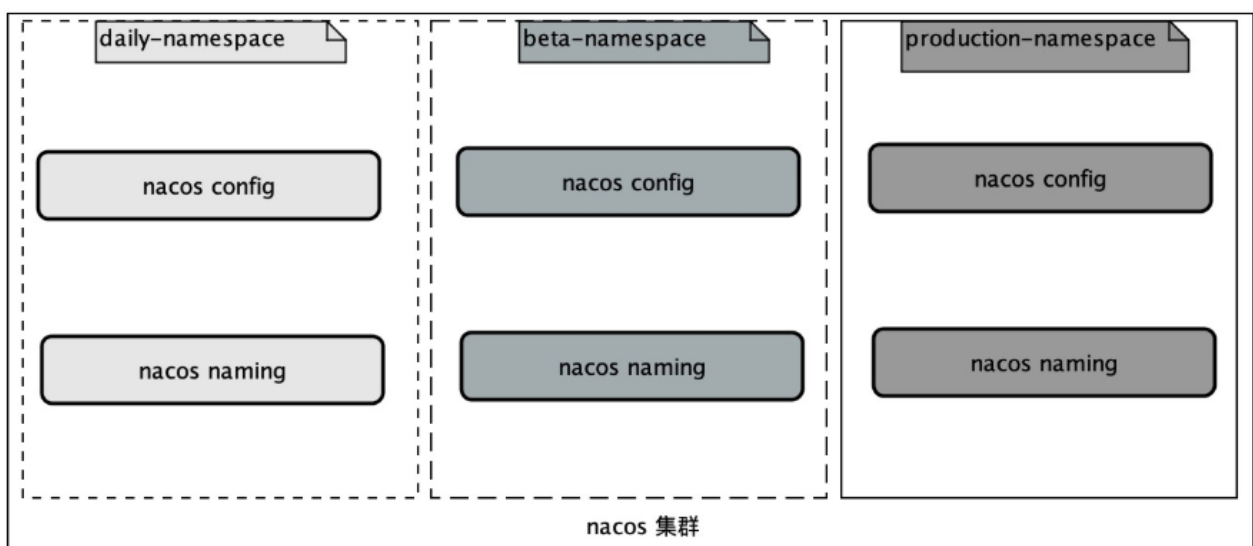
显示：consumer dubbo invoke | dubboService1 | dubboService2，表明调用service1成功，service1调用service2成功。

## 2.5 服务发现数据模型

### 2.5.1 Namespace 隔离设计

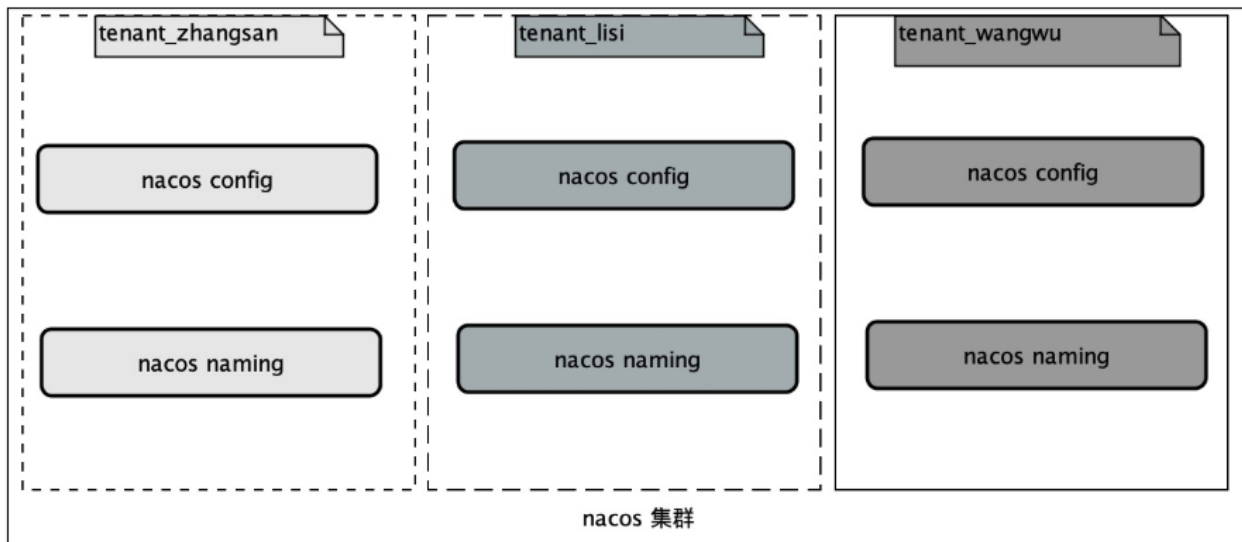
**命名空间(Namespace)**用于进行租户粒度的隔离，Namespace 的常用场景之一是不同环境的隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

- 从一个租户(用户)的角度来看，如果有多套不同的环境，那么这个时候可以根据指定的环境来创建不同的 namespace，以此来实现多环境的隔离。例如，你可能有开发，测试和生产三个不同的环境，那么使用一套 nacos 集群可以分别建以下三个不同的 namespace。如下图所示：



单个用户使用 nacos client，可通过不同的 namespace 来做不同环境下的配置/服务 数据隔离

- 从多个租户(用户)的角度来看，每个租户(用户)可能会有自己的 namespace,每个租户(用户)的配置数据以及注册的服务数据都会归属到自己的 namespace 下，以此来实现多租户间的数据隔离。例如超级管理员分配了三个租户，分别为张三、李四和王五。分配好了之后，各租户用自己的账户名和密码登录后，创建自己的命名空间。如下图所示：

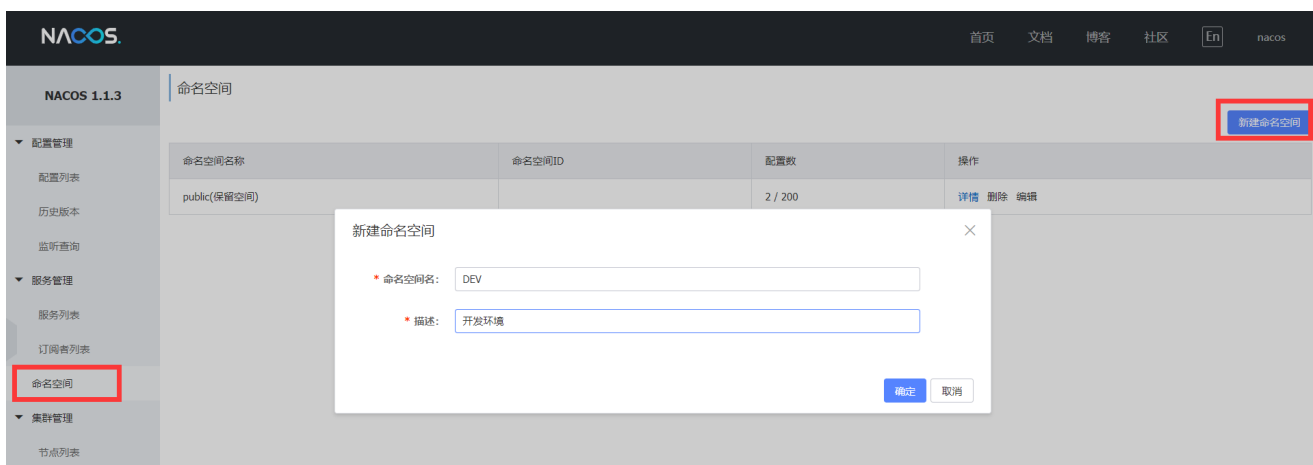


多个用户使用 nacos client，可通过自己租户下的 namespace 来初始化，不同租户下的不同 namespace 是不可见的。

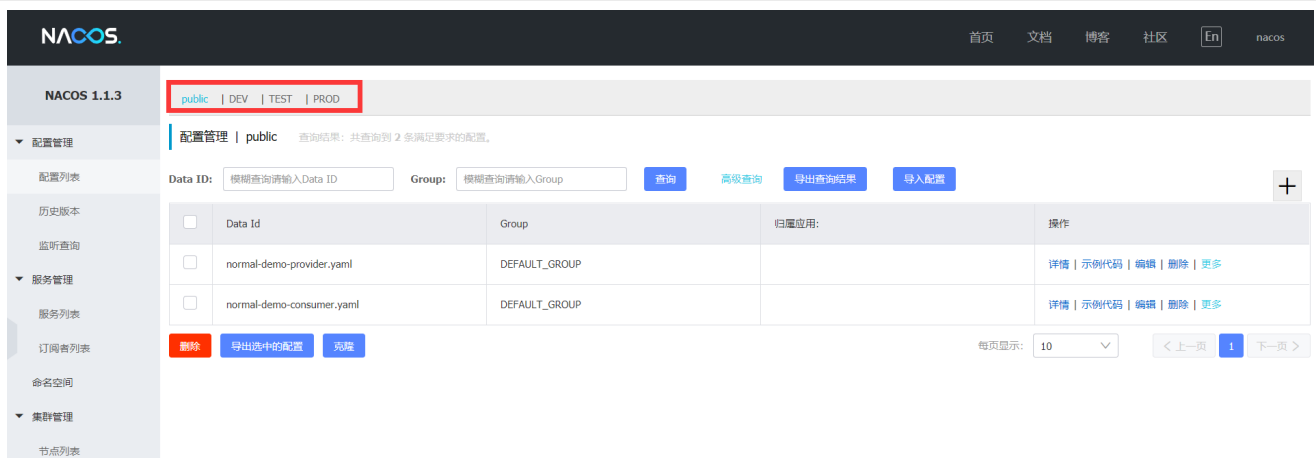
**注意:** 在此教程编写之时，nacos多租户(用户)功能还在规划中。

## 2.5.2 命名空间管理

前面已经介绍过，命名空间(Namespace)是用于隔离多个环境的（如开发、测试、生产），而每个应用在不同环境的同一个配置（如数据库数据源）的值是不一样的。因此，我们应针对企业项目实际研发流程、环境进行规划。如某软件公司拥有开发、测试、生产三套环境，那么我们应该针对这三个环境分别建立三个namespace。



建立好所有namespace后，在配置管理与服务管理模块下所有页面，都会包含用于切换namespace(环境)的tab按钮，如下图：



Note: namespace 为 **public** 是 nacos 的一个保留空间，如果您需要创建自己的 namespace，不要和 **public** 重名，以一个实际业务场景有具体语义的名字来命名，以免带来字面上不容易区分自己是哪一个 namespace。

Note：在编写程序获取配置集时，指定的namespace参数一定要填写**命名空间ID**，而不是名称

### 2.5.3 数据模型

Nacos在经过阿里内部多年生产经验后提炼出的数据模型，则是一种服务-集群-实例的三层模型，这样基本可以满足服务在所有场景下的数据存储和管理。



nacos服务发现的数据模型如下：

#### 服务

对外提供的软件功能，通过网络访问预定义的接口。

## 服务名

服务提供的标识，通过该标识可以唯一确定要访问的服务。

## 实例

提供一个或多个服务的具有可访问网络地址（IP:Port）的进程，启动一个服务，就产生了一个服务实例。

## 元信息

Nacos数据（如配置和服务）描述信息，如服务版本、权重、容灾策略、负载均衡策略、鉴权配置、各种自定义标签(label)，从作用范围来看，分为服务级别的元信息、集群的元信息及实例的元信息。

## 集群

服务实例的集合，服务实例组成一个默认集群，集群可以被进一步按需求划分，划分的单位可以是虚拟集群，相同集群下的实例才能相互感知。

通过数据模型可知：

应用通过Namespace、Service、Cluster(DEFAULT)的配置，描述了该服务向哪个环境（如开发环境）的哪个集群注册实例。

例子如下：

指定namespace的id：a1f8e863-3117-48c4-9dd3-e9ddc2af90a8（注意根据自己环境设置namespace的id）

指定集群名称：DEFAULT表示默认集群，可不填写

```
spring:
  application:
    name: transaction-service
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:7283 # 注册中心地址
        namespace: a1f8e863-3117-48c4-9dd3-e9ddc2af90a8 # 开发环境
        cluster-name: DEFAULT # 默认集群，可不填写
```

Note: 集群作为实例的隔离，相同集群的实例才能相互感知。

Note: namespace、cluster-name若不填写都将采取默认值，namespace的默认是public命名空间，cluster-name的默认值为DEFAULT集群。

## 3 Nacos配置管理

### 3.1 理解配置中心

#### 3.1.1 什么是配置

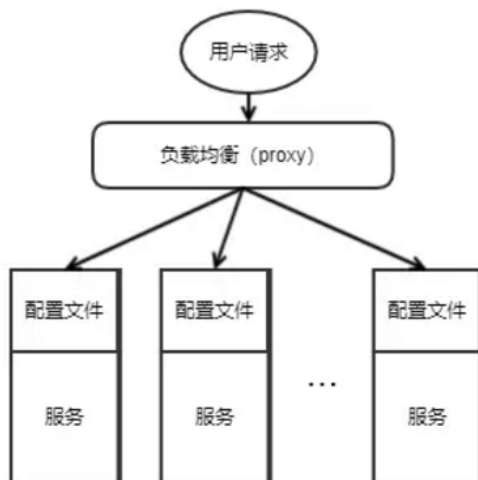
应用程序在启动和运行的时候往往需要读取一些配置信息，配置基本上伴随着应用程序的整个生命周期，比如：数据库连接参数、启动参数等。

配置主要有以下几个特点：

- **配置是独立于程序的只读变量**
  - 配置对于程序是只读的，程序通过读取配置来改变自己的行为，但是程序不应该去改变配置
- **配置伴随应用的整个生命周期**
  - 配置贯穿于应用的整个生命周期，应用在启动时通过读取配置来初始化，在运行时根据配置调整行为。  
比如：启动时需要读取服务的端口号、系统在运行过程中需要读取定时策略执行定时任务等。
- **配置可以有多种加载方式**
  - 常见的有程序内部hard code，配置文件，环境变量，启动参数，基于数据库等
- **配置需要治理**
  - 同一份程序在不同的环境（开发，测试，生产）、不同的集群（如不同的数据中心）经常需要有不同的配置，所以需要有完善的环境、集群配置管理

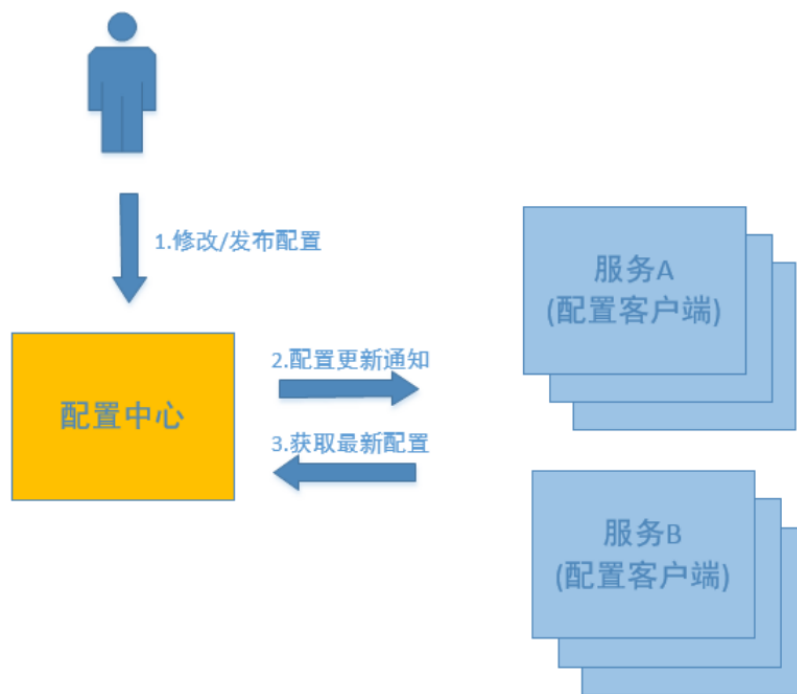
### 3.1.2 什么是配置中心

在微服务架构中，当系统从一个单体应用，被拆分成分布式系统上一个一个服务节点后，配置文件也必须跟着迁移（分割），这样配置就分散了，不仅如此，分散中还包含着冗余，如下图：



下图显示了配置中心的功能，配置中心将配置从各应用中剥离出来，对配置进行统一管理，应用自身不需要自己去管理配置。



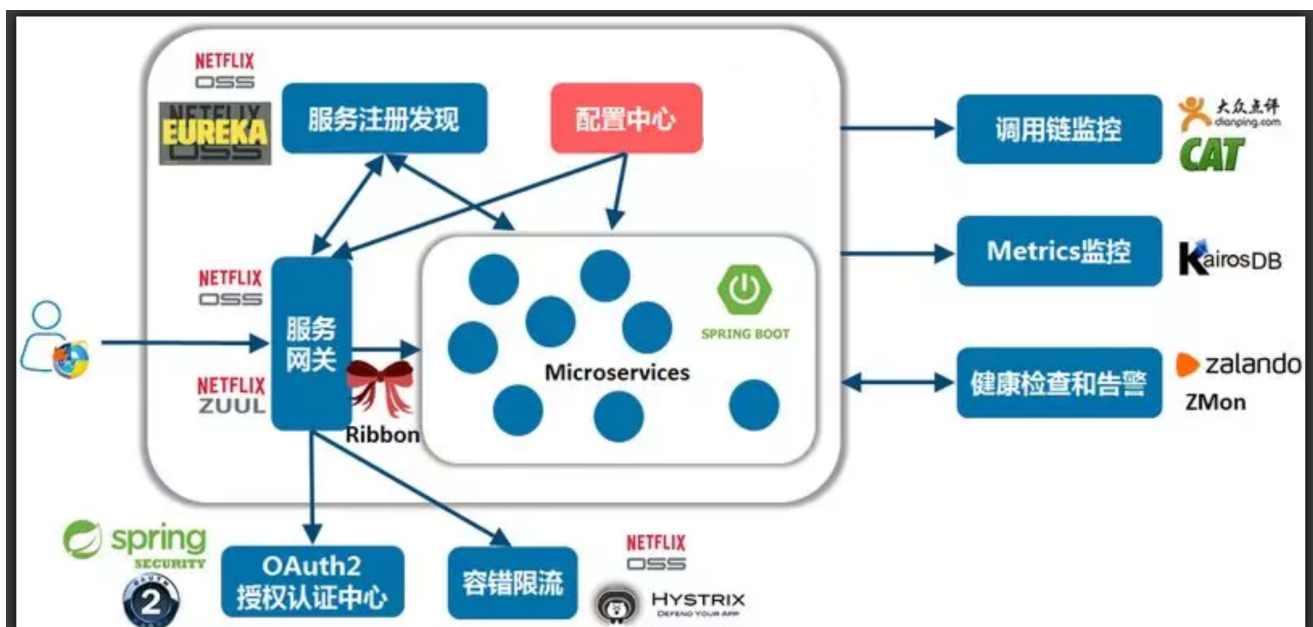


配置中心的服务流程如下：

- 1、用户在配置中心更新配置信息。
- 2、服务A和服务B及时得到配置更新通知，从配置中心获取配置。

**总的来说，配置中心就是一种统一管理各种应用配置的基础服务组件。**

在系统架构中，配置中心是整个微服务基础架构体系中的一个组件，如下图，它的功能看上去并不起眼，无非就是配置的管理和存取，但它是整个微服务架构中不可或缺的一环。



总结一下，在传统巨型单体应用纷纷转向细粒度微服务架构的历史进程中，配置中心是微服务化不可缺少的一个系统组件，在这种背景下中心化的配置服务即配置中心应运而生，一个合格的配置中心需要满足如下特性：

- 配置项容易读取和修改

- 分布式环境下应用配置的可管理性，即提供远程管理配置的能力
- 支持对配置的修改的检视以把控风险
- 可以查看配置修改的历史记录
- 不同部署环境下应用配置的隔离性

### 3.1.3 主流配置中心对比

目前市面上用的比较多的配置中心有：Spring Cloud Config、Apollo、Nacos和Disconf等。

由于Disconf不再维护，下面主要对比一下Spring Cloud Config、Apollo和Nacos。

对比项目	Spring Cloud Config	Apollo	Nacos
配置实时推送	支持(Spring Cloud Bus)	支持(HTTP长轮询1s内)	支持(HTTP长轮询1s内)
版本管理	支持(Git)	支持	支持
配置回滚	支持(Git)	支持	支持
灰度发布	支持	支持	不支持
权限管理	支持(依赖Git)	支持	不支持
多集群	支持	支持	支持
多环境	支持	支持	支持
监听查询	支持	支持	支持
多语言	只支持Java	主流语言，提供了Open API	主流语言，提供了Open API
配置格式校验	不支持	支持	支持
单机读(QPS)	7(限流所致)	9000	15000
单击写(QPS)	5(限流所致)	1100	1800
3节点读(QPS)	21(限流所致)	27000	45000
3节点写(QPS)	5(限流所致)	3300	5600

从配置中心角度来看，性能方面Nacos的读写性能最高，Apollo次之，Spring Cloud Config依赖Git场景不适合开放的大规模自动化运维API。功能方面Apollo最为完善，nacos具有Apollo大部分配置管理功能，而Spring Cloud Config不带运维管理界面，需要自行开发。Nacos的一大优势是整合了注册中心、配置中心功能，部署和操作相比Apollo都要直观简单，因此它简化了架构复杂度，并减轻运维及部署工作。

综合来看，Nacos的特点和优势还是比较明显的，下面我们一起进入Nacos的世界。

## 3.2 Nacos配置管理



### 3.2.1 发布配置

首先在nacos发布配置，nacos-restful-consumer服务从nacos读取配置。

浏览器访问 <http://127.0.0.1:8848/nacos>，打开nacos控制台，并点击菜单配置管理->配置列表：

在Nacos添加如下的配置：

nacos-restful-consumer:

```
Namespace: public
Data ID:   nacos-restful-consumer.yaml
Group  :   DEFAULT_GROUP
配置格式:  YAML
配置内容:  common:
            name: application1 config
```

## 新建配置

\* Data ID:

\* Group:

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

\* 配置内容:   
 ? : 

```
1 common:
2   name: application1 config
```

NACOS

首页 文档 博客 社区

NACOS 1.1.3

配置管理

配置列表

历史版本

监听查询

服务管理

public

配置管理 | public 查询结果：共查询到 1 条满足要求的配置。

Data ID:  Group:

<input type="checkbox"/>	Data Id	Group	归属应用:	操作
<input type="checkbox"/>	nacos-restful-consumer.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示</a>

### 3.2.2 获取配置

要想从配置中心获取配置添加nacos-config的依赖：

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

在bootstrap.yml添加配置：

```
spring:
  cloud:
    nacos:
      config:
        server-addr: 127.0.0.1:8848 # 配置中心地址
        file-extension: yaml
        group: DEFAULT_GROUP
```

注意：要使用配置中心就要在bootstrap.yml中来配置，bootstrap.yml配置文件的加载顺序要比application.yml要优先。

在nacos-restful-consumer工程的controller中增加获取配置的web访问端点/configs，通过标准的spring @Value方式。

```
@Value("${common.name}")
private String common_name;

@GetMapping(value = "/configs")
public String getvalue(){
    return common_name;
}
```

基于上面的例子，若要实现配置的动态更新，只需要进行如下改造：

```
// 注入配置文件上下文
@Autowired
private ConfigurableApplicationContext applicationContext;

@GetMapping(value = "/configs")
public String getConfigs(){
    return applicationContext.getEnvironment().getProperty("common.name");
}
```

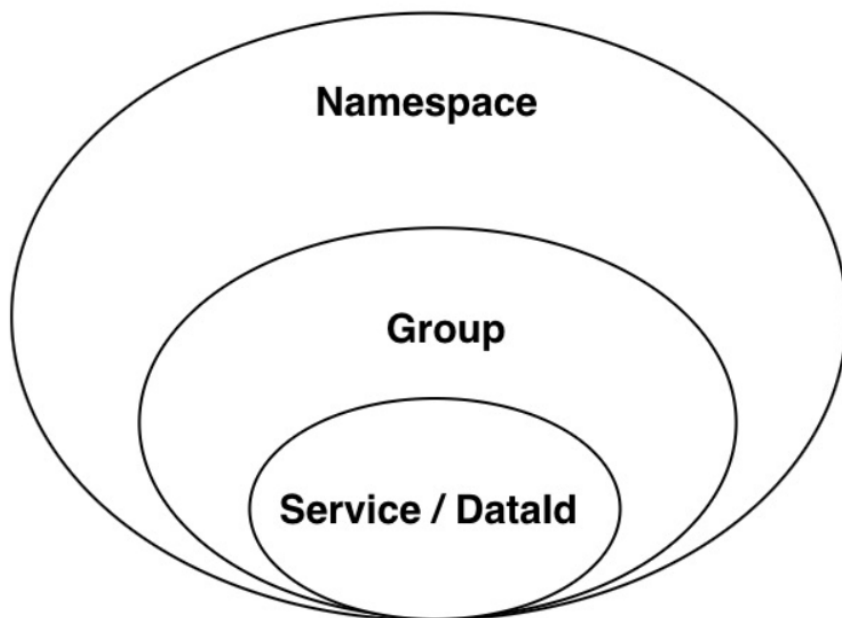
我们通过nacos控制台更新common.name的配置值，再次访问web端点/configs，发现应用程序能够获取到最新的配置值，说明spring-cloud-starter-alibaba-nacos-config 支持配置的动态更新。

Note 可以通过配置spring.cloud.nacos.config.refresh.enabled=false来关闭动态刷新

### 3.2.3 配置管理模型

对于Nacos配置管理，通过Namespace、group、Data ID能够定位到一个配置集。

## Nacos data model



#### 配置集(Data ID)

在系统中，一个配置文件通常就是一个**配置集**，一个配置集可以包含了系统的各种配置信息，例如，一个配置集可能包含了数据源、线程池、日志级别等配置项。每个配置集都可以定义一个有意义的名称，就是配置集的ID即Data ID。

#### 配置项

**配置集**中包含的一个个配置内容就是**配置项**。它代表一个具体的可配置的参数与其值域，通常以 key=value 的形式存在。例如我们常配置系统的日志输出级别（logLevel=INFO|WARN|ERROR）就是一个配置项。

#### 配置分组(Group)

配置分组是对配置集进行分组，通过一个有意义的字符串（如 Buy 或 Trade）来表示，不同的配置分组下可以有相同的配置集（Data ID）。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT\_GROUP。配置分组的常见场景：可用于区分不同的项目或应用，例如：学生管理系统的配置集可以定义一个group为：STUDENT\_GROUP。

#### 命名空间(Namespace)

命名空间（namespace）可用于进行不同环境的配置隔离。例如可以隔离开发环境、测试环境和生产环境，因为它们的配置可能各不相同，或者是隔离不同的用户，不同的开发人员使用同一个nacos管理各自的配置，可通过namespace隔离。不同的命名空间下，可以存在相同名称的配置分组(Group)或配置集。

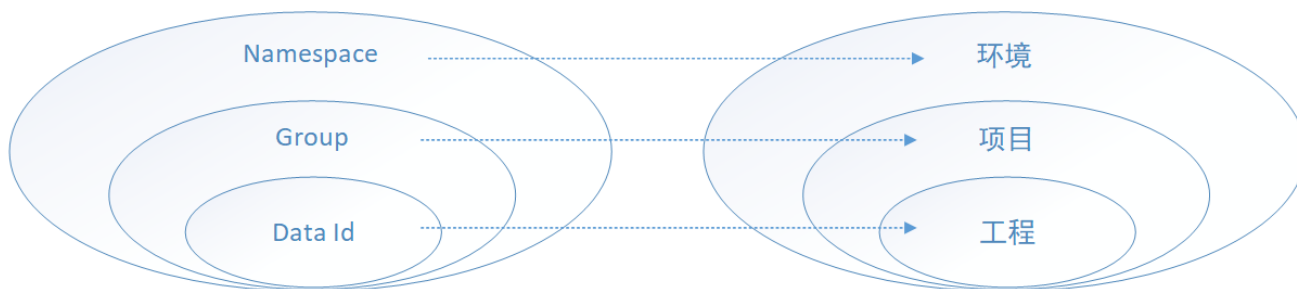
#### 最佳实践

Nacos抽象定义了Namespace、Group、Data ID的概念，具体这几个概念代表什么，取决于我们把它们看成什么，这里推荐给大家一种用法，如下图：

Namespace：代表不同**环境**，如开发、测试、生产环境。

Group：代表某**项目**，如XX医疗项目、XX电商项目

DataId：每个项目下往往有若干个**工程**，每个配置集(DataId)是一个工程的**主配置文件**



**获取某配置集的代码：**

获取配置集需要指定：

- 1、nacos服务地址，必须指定
- 2、namespace，如不指定默认public

在config中指定namespace，例子如下：

```
config:
  server-addr: 127.0.0.1:8848 # 配置中心地址
  file-extension: yaml
  namespace: a1f8e863-3117-48c4-9dd3-e9ddc2af90a8 # 开发环境
  group: DEFAULT_GROUP # xx业务组
```

- 3、group，如不指定默认 DEFAULT\_GROUP

见上边第2点的例子。

- 4、dataId，必须指定，名称为应用名称+配置文件扩展名

```
spring:
  application:
    name: nacos-restful-consumer
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
# namespace: alf8e863-3117-48c4-9dd3-e9ddc2af90a8
# cluster-name: DEFAULT
      config:
        server-addr: 127.0.0.1:8848 # 配置中心地址
        file-extension: yaml
```

nacos-restful-consumer.yaml

## 3.3 自定义扩展的 Data Id 配置

### 3.3.1 ext-config扩展配置

Spring Cloud Alibaba Nacos Config可支持自定义 Data Id 的配置。一个完整的配置案例如下所示：

```
spring:
  application:
    name: service2
  cloud:
    nacos:
      config:
        server-addr: 127.0.0.1:8848 # 配置中心地址
        file-extension: yaml
# namespace: alf8e863-3117-48c4-9dd3-e9ddc2af90a8 # 开发环境
        group: DEFAULT_GROUP
# config external configuration
# 1、Data Id group: 组名,refresh: 动态刷新
        ext-config[0]:
          data-id: ext-config-common01.yaml
          group: COMMON_GROUP
          refresh: true
        ext-config[1]:
          data-id: ext-config-common02.yaml
          group: COMMON_GROUP
          refresh: true
```

可以看到:

- 通过 `spring.cloud.nacos.config.ext-config[n].data-id` 的配置方式来支持多个 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.ext-config[n].group` 的配置方式自定义 Data Id 所在的组，不明确配置的话，默认是 `DEFAULT_GROUP`。

- 通过 `spring.cloud.nacos.config.ext-config[n].refresh` 的配置方式来控制该 Data Id 在配置变更时，是否支持应用中可动态刷新，感知到最新的配置值。默认是不支持的。

Note : `spring.cloud.nacos.config.ext-config[n].data-id` 的值必须带文件扩展名，文件扩展名既可支持 properties，又可以支持 yaml/yml。此时 `spring.cloud.nacos.config.file-extension` 的配置对自定义扩展配置的 Data Id 文件扩展名没有影响。

测试：

配置ext-config-common01.yaml：

\* Data ID: `ext-config-common01.yaml`

\* Group: `COMMON_GROUP`

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容 ? :

```
1 common:
2   name: zhangsan
3   addr: beijing
```

配置ext-config-common02.yaml：



\* Data ID: ext-config-common02.yaml

\* Group: COMMON\_GROUP

[更多高级选项](#)

描述: null

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容 ? :

```
1 common:
2   name: lisi
3   addr: zhengzhou
```

编写测试代码：

```
@GetMapping(value = "/configs")
public String getvalue(){
    String name = applicationContext.getEnvironment().getProperty("common.name");
    String address = applicationContext.getEnvironment().getProperty("common.addr");
    return name+address;
}
```

重启nacos-restful-consumer工程

访问：<http://127.0.0.1:56020/configs>

通过测试发现：

扩展配置优先级是 `spring.cloud.nacos.config.ext-config[n].data-id` 其中 n 的值越大，优先级越高。

通过内部相关规则(应用名、扩展名)自动生成相关的 Data Id 配置的优先级最大。

### 3.3.2 案例

案例需求如下：

1、抽取servlet公用的配置到独立的配置文件，配置文件内容如下：

```
#HTTP格式配置
spring:
  http:
    encoding:
      charset: UTF-8
```

```
force: true
enabled: true
messages:
  encoding: UTF-8

#tomcat头信息和访问路径配置
server:
  tomcat:
    remote_ip_header: x-forwarded-for
    protocol_header: x-forwarded-proto
  servlet:
    context-path: /a
    use-forward-headers: true
```

2、如果有context-path的个性配置可以单独设置

实现如下：

1、抽取servlet公用的配置到独立的配置文件

在nacos-restful-consumer中添加扩展配置文件，下图中红色框内为指定的配置文件名称：

```
config:
  server-addr: 127.0.0.1:8848 # 配置中心地址
  file-extension: yaml
  namespace: alf8e863-3117-48c4-9dd3-e9ddc2af90a8 # 开发环境
  group: DEFAULT_GROUP
  # config external configuration
  # 1、Data Id group: 组名, refresh: 动态刷新
  ext-config[0]:
    data-id: ext-config-common01.yaml
    group: COMMON_GROUP
    refresh: true
  ext-config[1]:
    data-id: ext-config-common02.yaml
    group: COMMON_GROUP
    refresh: true
  ext-config[2]:
    data-id: ext-config-http.yaml
    group: COMMON_GROUP
    refresh: true
```

在nacos中添加扩展配置文件，下图中红色框内是servlet公用的配置内容。



\* Data ID: ext-config-http.yaml

\* Group: COMMON\_GROUP

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容 ? :

```
1 #HTTP格式配置
2 spring:
3   http:
4     encoding:
5       charset: UTF-8
6       force: true
7       enabled: true
8   messages:
9     encoding: UTF-8
10
```

发布配置，重启nacos-restful-consumer

访问<http://127.0.0.1:56020/a/configs>验证配置是否生效

2、如果有context-path的个性配置可以单独设置

这里我们需要对nacos-restful-consumer的context-path设置为根路径，如何实现？

方案1：修改ext-config-http.yaml中context-path为根路径，但是此文件定义为公用配置文件，其它服务可使用了，此方案不可行。

方案2：在nacos-restful-consumer.yaml中定义context-path为根路径，因为nacos-restful-consumer.yaml单独属于nacos-restful-consumer工程，且nacos-restful-consumer.yaml的优先级比ext-config-http.yaml高，所以此方案可行。

在nacos中配置nacos-restful-consumer.yaml，如下：

\* Data ID:

\* Group:

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容 ? :

```
1 common:
2   name: application1 config
3 server:
4   servlet:
5     context-path: /
```

重启nacos-restful-consumer工程。

测试，请求：<http://127.0.0.1:56020/configs>

## 4 总结

Nacos用来干什么？

Nacos是阿里巴巴公司开源的项目，它用来实现配置中心和服务注册中心。

什么是服务发现？

在微服务架构中一个业务流程需要多个微服务通过网络接口调用完成业务处理，服务消费方从服务注册中心获取服务提供方的地址，从而进行远程调用，这个过程叫做服务发现。

服务发现的流程是什么？

- 1、服务发现的客户端从服务注册中心获取服务列表
- 2、服务消费方通过客户端负载均衡获取服务实例地址，进行远程调用。

什么是配置中心？

在微服务架构中为了统一管理各各微服务的配置信息专门设置配置中心，配置中心就是一种统一管理各种应用配置的基础服务组件。

配置中心的应用流程是什么？

- 1、发布配置，将配置信息发布到配置中心。

2、获取配置，配置中心客户端得到配置中心的通知，从配置中心获取配置。

Spring Cloud是什么？

Spring Cloud是一套微服务开发框架集合，包括微服务开发的方方面面，Spring Cloud是一套微服务开发的标准，集成了很多优秀的开源框架，比如有名的Netflix公司的众多项目。

Spring Cloud Alibaba是什么？

Spring Cloud Alibaba是阿里巴巴公司基于Spring Cloud标准实现一套微服务开发框架集合，它和Netflix一样都是Spring Cloud微服务开发实现方案。

Dubbo服务开发流程是什么？

1、定义api工程。

方便其它服务原来api工程，远程调用dubbo服务。

2、定义api实现工程。

service实现类使用 `@org.apache.dubbo.config.annotation.Service` 注解标记为dubbo服务。

3、服务消费方开发

引入api工程依赖

使用 `org.apache.dubbo.config.annotation.Reference` 注解注入service，发起远程调用