

# Table of Contents

Introduction	1. 1
第一章 基础	1. 2
1. 1 什么是nacos	1. 2. 1
1. 2 nacos的功能和生态	1. 2. 2
1. 3 nacos专业名词	1. 2. 3
1. 4 nacos架构	1. 2. 4
1. 5 nacos的安装启动	1. 2. 5
第二章 实践	2. 1
2. 1 NacosClient	2. 1. 1
2. 2 Nacos集成Spring	2. 1. 2
2. 3 Nacos集成SpringBoot	2. 1. 3
2. 4 Nacos集成SpringCloud	2. 1. 4
2. 5 Nacos集成dubbo	2. 1. 5
2. 6 技术探讨OR学习总结	2. 1. 6
第三章 源码解析	3. 1
3. 1 核心特性	3. 1. 1
第四章 总结期许	3. 2
第五章 技术畅想	3. 3

- 写作背景
- 本书简介

## 写作背景

这本书是本人写的第一本gitbook书，大概耗费一个下午 将gitbook给安装。因为最近在研究nacos这个配置分发和注册中心，所以就想以此为题写一本书。希望之后的自己看到后，能够专心的把这本书给写出。

希望本书的写作内容，能给后来者一些启发，本书不仅仅是包含内容干活，更是面向个人的学习习惯触发。因为观念不同，可能我们看框架的视角不同，可做参考，但不可作为依赖。

本书基于Nacos2.2.3进行编写，如有错误的地方欢迎指正。

## 本书简介

首先，我们学习一门技术最好的方案就是从官方网站开始，下面是nacos的官方网站。

The screenshot shows the Nacos official website at nacos.io/zh-cn/docs/what-is-nacos.html. The page has a header with navigation links like Home, Documentation, Enterprise Edition, Free Trial, Architecture & Principles, Blog, Community, and Control Panel Examples. A search bar and language selection (En) are also present. The main content area features a sidebar with sections for Nacos (What is Nacos?, Introduction, Concepts, Architecture, Quick Start, Nacos, Nacos Spring, Nacos Spring Boot, Nacos Spring Cloud, Nacos Docker, Nacos Dubbo, Nacos k8s, Nacos Sync, User Guide), and a main content block titled 'Overview' (概览) which introduces Nacos as a dynamic naming and configuration service.

其中包含了其基本的介绍和与其它项目的集成，接下来我们将会从基本的概念和架构开始，先进行了解其基本的作用和实现，再与其他项目集成，最后我们再深挖源码，下面就开始我们的学习之旅吧！

- 1.1 什么是nacos?

## 1.1 什么是nacos?

一门技术的兴起一定会有它自己独特的功能，正如缓存、数据库、消息队列等一系列的中间件一样，nacos也有它自己独特的起源。

nacos主要是阿里的开源产品，伴随的是阿里的生产实践以及借鉴其他的注册中心而有的孵化品，在官网上我们可以看到有篇关于阿里巴巴服务注册中心产品的发展回顾，《[阿里巴巴服务注册中心产品ConfigServer 10年技术发展回顾](#)》，这篇文档的大概意思是在阿里的业务拓展下，最初的服务注册发现产品Eureka不再符合阿里的业务，于是在2018年左右，阿里开始了自研服务注册中心的道路，最开始也是借鉴Eureka的设计理念，往后推进时也添加上了一些自己的思考和阿里线上的具体实践，一步步的迭代，从最初的SDK，到单机版，再到集群一步步的解决了服务注册发现方面的一些问题，然后就形成了我们今天所看到的从ConfigServer进化而来的nacos。

	ConfigServer	Eureka
2008年	V1.0：单机版，定义了服务发现的领域模型	
2009年初	V1.5：应用和ConfigServer集群发布解耦	
2009年7月	V2.0：基于客户端模式同步数据，支持集群部署	
2010年底	V2.5：优化集群间数据同步模式，申请国家专利。	
2012年9月1号		Eureka1.0正式开源
2012年底	V3.0：支持session和data分层部署	
2014年	V3.5：支持异地多活等细分场景	
2015年		Eureka2.0架构升级方案公布
2017年	V4.0：支持data分片能力	
2018年7月		Eureka2.0架构升级宣布停止

在分布式系统中，有三个特性一致性、可用性、分区容错性，而注册中心必然处于分布式系统中，那么它必然要满足[CAP原则](#)。而Eureka和ConfigServer则是同属于AP类型的注册中心，他们两个在之后的业务拓展中

拥有着相似的阻碍，而阿里巴巴则将ConfigServer的技术架构和生产环境的发现融合到了开源产品nacos中，继往开来在云原生、微服务的时代继续着发光发热。

- 1.2 nacos功能和生态

- 1.2.1 特性
- 1.2.2 生态

上一节我们讨论了nacos的发展及一些基本的注册中心知识，这一节我们将对于它的功能和生态进行进一步的探讨。

## 1.2 nacos功能和生态

对于我们学习技术来说，最重要的就是这门新的技术有什么样的功能，和他的生态是否强大。功能决定了它的业务适用性，任何的技术都是为业务而生。而生态则是技术的后备支持，比如说漏洞维护，功能新增等一系列的技术支持。很显然Nacos是阿里巴巴的产品，而且可以与多种技术进行集成，由此决定了它在微服务中的适用性。

### 1.2.1 特性

nacos的官网是这么说的，它是一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

#### 1、服务发现和服务健康检测

服务发现和服务健康检测基本上是每一个注册中心都需要进行保证的功能。Nacos采用的是基于DNS和RPC的服务发现。支持传输层层面的健康检测，也就是我们常用的ping命令或者tcp的一系列指令，同时也延迟应用层自定义的健康检查。

对于复杂的云环境还提供了agent 上报模式和服务端主动检测2种健康检查模式。

#### 2、动态配置服务

动态配置可以让我们的配置文件中心化、外部化和动态化的管理所有环境的应用配置和服务配置，可以更加方便的管理服务，让服务进行弹性的拓展更加的简单。换句话说就是我们的配置文件不再由应用管理，而是交由一个中心化的应用进行管理，方便了配置文件的更改和版本追踪，最终还是服务于应用。

### 3、动态DNS服务（DDNS）

通过支持权重路由，动态DNS服务能让您轻松实现中间层负载均衡、更灵活的路由策略、流量控制以及简单数据中心内网的简单DNS解析服务。动态DNS服务还能让您更容易地实现以DNS协议为基础的服务发现，以消除耦合到厂商私有服务发现API上的风险。

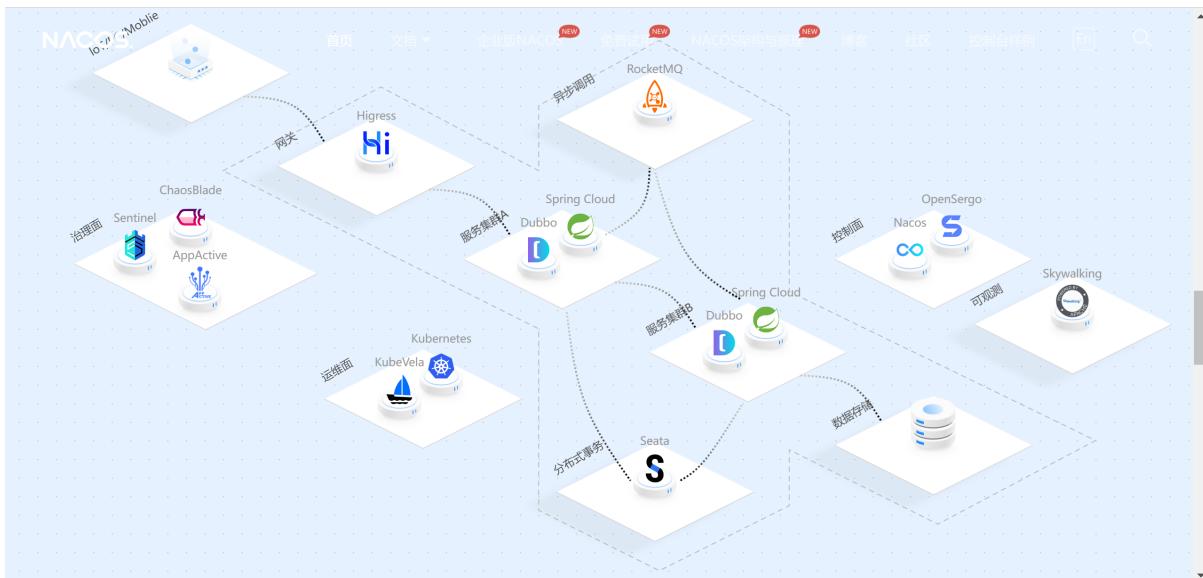
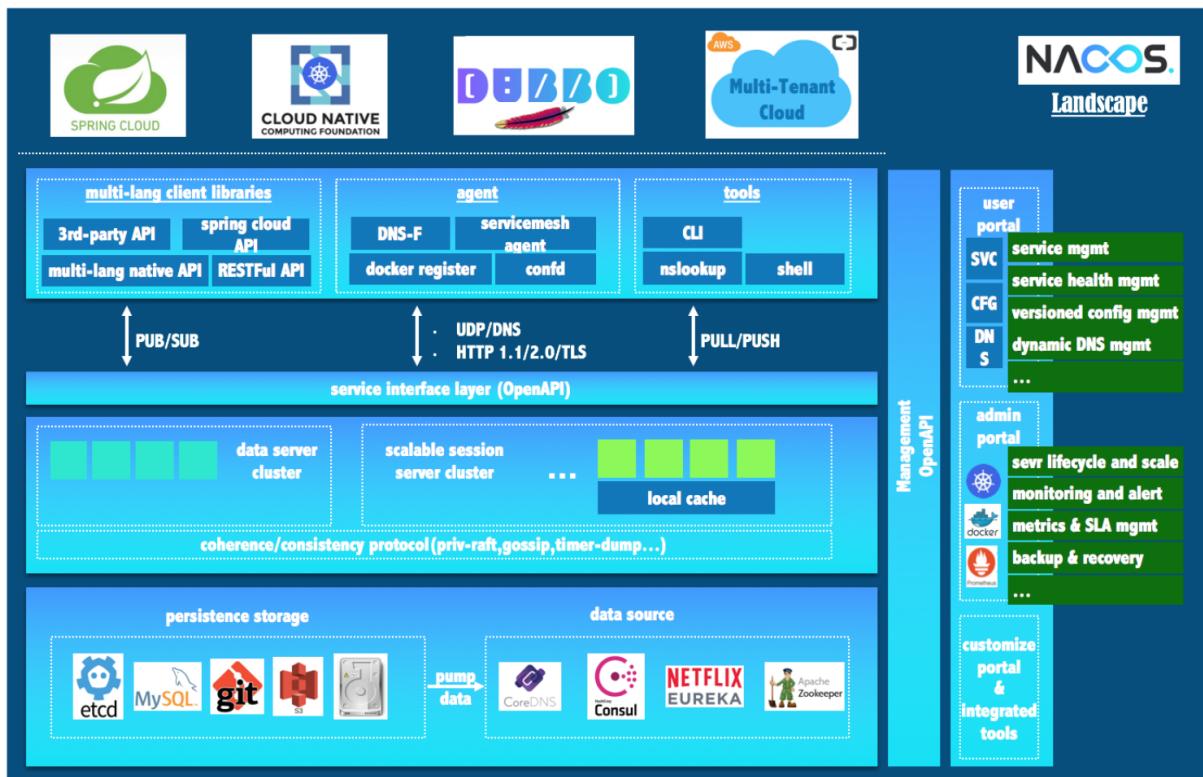
### 4、服务及其元数据管理

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、服务的 SLA 以及最首要的 metrics 统计数据。

## 1.2.2 生态

nacos的生态十分强大，它可以很方便的与一些第三方框架进行集成，具体的在此不再阐述，之后会在项目中进行一一的展现。而在此我想说的是，对于一个服务的生态，我们需要关心的是这个技术的使用方向以及使用的广度。

对于本书的nacos来说，他有非常多的使用方案，包括配置分发，服务发现等，因为背靠阿里巴巴这个巨大的生态圈，也使得它的更新迭代也是非常快的，我们在使用的初期仅需要去掌握使用即可，等到我们使用的次数多了，再去探讨核心功能的实现才是最好的‘食用方式’。下图为Nacos的生态图：



- 1.3 Nacos的专业名词
  - 1.3.1 地域
  - 1.3.2 可用区
  - 1.3.3 接入点
  - 1.3.4 命名空间
  - 1.3.5 配置
  - 1.3.6 配置管理
  - 1.3.7 配置项
  - 1.3.8 配置集
  - 1.3.9 配置集 ID
  - 1.3.10 配置分组
  - 1.3.11 配置快照
  - 1.3.12 服务
  - 1.3.13 服务名
  - 1.3.14 服务注册中心
  - 1.3.15 服务发现
  - 1.3.16 元信息
  - 1.3.17 应用
  - 1.3.18 服务分组
  - 1.3.19 虚拟集群
  - 1.3.20 实例
  - 1.3.21 权重
  - 1.3.22 健康检查
  - 1.3.23 健康保护阈值

上一节我们说到了Nacos的一些特性和它的生态圈，如果不出意外的话，或者作者未来的几年依然在这个行业，一定会慢慢的把生态圈上的所有给更新完毕。接下来我们将要来学习下Nacos的一些专业术语

## 1.3 Nacos的专业名词

这一部分主要取自Nacos的官方文档，因为对于专业名词来说，官网给的才是最准确的，其中也添加上了一点自己的见解。

### 1.3.1 地域

物理的数据中心，资源创建成功之后就不能再次更换。

### 1.3.2 可用区

同一地域内，电力和网络互相独立的物理区域。同一可用区内，实例的网络延迟较低。

### 1.3.3 接入点

地域的某个服务的入口域名。

### 1.3.4 命名空间

用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。这个我们之后的使用会比较多，主要用于隔绝环境，方便我们在不同环境下的开发工作。

### 1.3.5 配置

在系统开发过程中，开发者通常会将一些需要变更的参数、变量等从代码中分离出来独立管理，以独立的配置文件的形式存在。目的是让静态的系统工件或者交付物（如 WAR，JAR 包等）更好地和实际的物理运行环境进行适配。配置管理一般包含在系统部署的过程中，由系统管理员或者运维人员完

成。配置变更是调整系统运行时的行为的有效手段。这里的配置文件有很多格式，text, yml, properties等，已经可以符合我们的日常配置文件的格式。

### 1.3.6 配置管理

主要是对上面的配置进行一系列的操作，后面我们会进行探讨其源码的具体实现。系统配置的编辑、存储、分发、变更管理、历史版本管理、变更审计等所有与配置相关的活动。

### 1.3.7 配置项

一个具体的可配置的参数与其值域，通常以 param-key=param-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。

### 1.3.8 配置集

一组相关或者不相关的配置项的集合称为配置集。在系统中，一个配置文件通常就是一个配置集，包含了系统各个方面的配置。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。

### 1.3.9 配置集 ID

Nacos 中的某个配置集的 ID。配置集 ID 是组织划分配置的维度之一。Data ID 通常用于组织划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 通常采用类 Java 包（如 com.taobao.tc.refund.log.level）的命名规则保证全局唯一性。此命名规则非强制。

### 1.3.10 配置分组

Nacos 中的一组配置集，是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade ）对配置集进行分组，从而区分 Data ID 相同的配置集。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：不同的应用或组件使用了相同的配置类型，如 database\_url 配置和 MQ\_topic 配置。

### 1.3.11 配置快照

Nacos 的客户端 SDK 会在本地生成配置的快照。当客户端无法连接到 Nacos Server 时，可以使用配置快照显示系统的整体容灾能力。配置快照类似于 Git 中的本地 commit，也类似于缓存，会在适当的时机更新，但是并没有缓存过期（expiration）的概念。可以说是本地的缓存，用于缓解 Nacos 出现问题无法访问时可以生效，保证服务的正常运行。

### 1.3.12 服务

通过预定义接口网络访问的提供给客户端的软件功能。例如在大型网购系统中的订单系统、消费券系统、会员系统等。

### 1.3.13 服务名

服务提供的标识，通过该标识可以唯一确定其指代的服务。

### 1.3.14 服务注册中心

存储服务实例和服务负载均衡策略的数据库，可以将自己自身的服务信息提交到注册中心，其他的服务模块想要调用时则可以进行注册中心寻找调用的信息，注册中心也可以在此时做一层负载均衡，将请求分发到算力充足或者

业务不是很繁忙的服务上面去。

### 1.3.15 服务发现

在计算机网络上，（通常使用服务名）对服务下的实例的地址和元数据进行探测，并以预先定义的接口提供给客户端进行查询。

### 1.3.16 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。

### 1.3.17 应用

用于标识服务提供方的服务的属性。

### 1.3.18 服务分组

不同的服务可以归类到同一分组。

### 1.3.19 虚拟集群

同一个服务下的所有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。

### 1.3.20 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。

### 1. 3. 21 权重

实例级别的配置。权重为浮点数。权重越大，分配给该实例的流量越大。

### 1. 3. 22 健康检查

以指定方式检查服务下挂载的实例（Instance）的健康度，从而确认该实例（Instance）是否能提供服务。根据检查结果，实例（Instance）会被判断为健康或不健康。对服务发起解析请求时，不健康的实例（Instance）不会返回给客户端。

### 1. 3. 23 健康保护阈值

为了防止因过多实例（Instance）不健康导致流量全部流向健康实例（Instance），继而造成流量压力把健康实例（Instance）压垮并形成雪崩效应，应将健康保护阈值定义为一个 0 到 1 之间的浮点数。当域名健康实例数（Instance）占总服务实例数（Instance）的比例小于该值时，无论实例（Instance）是否健康，都会将这个实例（Instance）返回给客户端。这样做虽然损失了一部分流量，但是保证了集群中剩余健康实例（Instance）能正常工作。

## • 1.4 Nacos架构

上一节我们着重的了解了下Nacos的一些官方名词，接下来我们将继续深入Nacos的架构继续探讨。这时可能有人会疑惑了，为什么架构这么重要的部分只介绍了一节，因为官网已经给了一本300多页的书来讲Nacos的架构，产品的开发人员肯定比我更加的了解技术架构，所以在此就不再过多的阐述，就大概的讲述一下我对于软件架构的学习方法。

# 1.4 Nacos架构

对于一门技术的架构的话，我们如果学习，还是需要从官方入手，如果是比较火的技术，官方一般都会出一个文档关于研发这个软件的过程，以及致力于解决什么问题。



如上图所示，这本书是由阿里出的，所以内容也不会太差，我们在学习的初期，一般是不会太多的去关注架构。大概是在使用之后，我们需要对于开源软件有我们自己的DIY时才会去关注软件的架构，然后对于模块进行魔改，然后进行内测上线。

## 《Nacos架构&原理》

- 1.5 Nacos的安装和启动
  - 1.5.1 单机部署启动
  - 1.5.2 docker部署启动

这一节我们来探讨一下软件的安装和启动，在我们之前的学习中，我们可能倾向于去将手动下载，然后解压安装，但是随着技术的迭代，越来越多的虚拟技术的出现使得软件安装部署也有了新的花样。

假如我们来假设这样一个场景，我们需要部署一个集群，这样的话如果是在我们自己测试的环境还好说，通过暴力进行解决，或者说也可以通过写一个 shell脚本进行，但是如果说是在不同的主机上，这样就会变得很麻烦，于是乎就有了Docker, k8s等一系列的实现方案，我们可以在虚拟的容器内进行部署，然后也可以动态的进行上线下线的控制。对于Docker的学习还是有一定的必要的，之后可以了解一下这里只是简单的介绍一下之后的软件安装模式。

## 1.5 Nacos的安装和启动

我们这部分还是根据Nacos的官网来进行，有什么需要注意的细节我会一一的点明。

因为Nacos是基于JAVA实现的，所以最基础的要JAVA的环境，这一点是必不可少的，一下的安装都基于Linux完成，原因的话可以下去自己去了解。

### 1.5.1 单机部署启动

我们就从最基础的开始安装，只演示一次，之后我们会只使用docker安装。因为学到这里就已经默认已经具备独立安装JDK的条件了，我们主要来进行安装Nacos。

首先我们要先下载适合我们的软件包，下载地址我们一般是在github上找稳定的版本，我选择的是最新的

Nacos，选择第一个进行下载。

The vulnerability only affects port 7848 (by default), which is typically used as the communication port for Nacos cluster inter-raft protocol and does not handle client requests. Therefore, the risk can be controlled by disabling requests from outside of Nacos clusters (e.g. by limiting or not exposing the port) in older versions.

Detail:

- #10318 Fix import problem when disable auth.
- #10542 Add classes whitelist for HessianSerializer.

▼ Assets 4

		142 MB	May 25
)nacos-server-2.2.3.tar.gz			
)nacos-server-2.2.3.zip		142 MB	May 25
Source code (zip)			May 25
Source code (tar.gz)			May 25

Smile 40 Like 10 Share 8 50 people reacted

下载成功后，我们将其上传至Linux中，也可以通过linux的curl命令下载，但是一般很慢不建议。

```
#解压  
tar -zxvf nacos-server-2.2.3.tar.gz nacos
```

修改配置，nacos的配置是在application.properties中

```
***** Spring Boot Related Configurations *****  
#这些一般用默认的就可以  
### Default web context path:  
server.servlet.contextPath=/nacos  
### Include message field  
server.error.include-message=ALWAYS  
### Default web server port:  
server.port=8848  
  
***** Network Related Configurations *****  
### If prefer hostname over ip for Nacos server addresses in cluster  
# nacos.inetutils.prefer-hostname-over-ip=false  
  
### Specify local server's IP:  
# nacos.inetutils.ip-address=  
  
***** Config Module Related Configurations *****  
### If use MySQL as datasource:  
### Deprecated configuration property, it is recommended to use `spring.datasource.platform`  
# spring.datasource.platform=mysql  
# spring.sql.init.platform=mysql  
  
### Count of DB:  
# db.num=1  
  
#数据库的配置，我就用我本机的mysql，也不再进行安装了  
db.url.0=jdbc:mysql://10.102.46.60:3306/nacos?characterEncoding=utf8  
db.user.0=root  
db.password.0=123456
```

```
### Connection pool configuration: hikariCP
db.pool.config.connectionTimeout=30000
db.pool.config.validationTimeout=10000
db.pool.config.maximumPoolSize=20
db.pool.config.minimumIdle=2

***** Naming Module Related Configurations *****

### If enable data warmup. If set to false, the server would accept
# nacos.naming.data.warmup=true

### If enable the instance auto expiration, kind like of health check
# nacos.naming.expireInstance=true

### Add in 2.0.0
### The interval to clean empty service, unit: milliseconds.
# nacos.naming.clean.empty-service.interval=60000

### The expired time to clean empty service, unit: milliseconds.
# nacos.naming.clean.empty-service.expired-time=60000

### The interval to clean expired metadata, unit: milliseconds.
# nacos.naming.clean.expired-metadata.interval=5000

### The expired time to clean metadata, unit: milliseconds.
# nacos.naming.clean.expired-metadata.expired-time=60000

### The delay time before push task to execute from service changed
# nacos.naming.push.pushTaskDelay=500

### The timeout for push task execute, unit: milliseconds.
# nacos.naming.push.pushTaskTimeout=5000
```

```
### The delay time for retrying failed push task, unit: milliseconds.
# nacos.naming.push.pushTaskRetryDelay=1000

### Since 2.0.3
### The expired time for inactive client, unit: milliseconds.
# nacos.naming.client.expired.time=180000

***** CMDB Module Related Configurations *****
### The interval to dump external CMDB in seconds:
# nacos.cmdb.dumpTaskInterval=3600

### The interval of polling data change event in seconds:
# nacos.cmdb.eventTaskInterval=10

### The interval of loading labels in seconds:
# nacos.cmdb.labelTaskInterval=300

### If turn on data loading task:
# nacos.cmdb.loadDataAtStart=false

***** Metrics Related Configurations *****
### Metrics for prometheus
#management.endpoints.web.exposure.include=*

### Metrics for elastic search
management.metrics.export.elastic.enabled=false
#management.metrics.export.elastic.host=http://localhost:9200

### Metrics for influx
management.metrics.export.influx.enabled=false
```

```

#management.metrics.export.influx.db=springboot
#management.metrics.export.influx.uri=http://localhost:8086
#management.metrics.export.influx.auto-create-db=true
#management.metrics.export.influx.consistency=one
#management.metrics.export.influx.compressed=true

***** Access Log Related Configurations *****
### If turn on the access log:
server.tomcat.accesslog.enabled=true

### The access log pattern:
server.tomcat.accesslog.pattern=%h %l %u %t "%r" %s %b %D %{User-Agent}i

### The directory of access log:
server.tomcat.basedir=file:.

***** Access Control Related Configurations *****
### If enable spring security, this option is deprecated in 1.2.0:
#spring.security.enabled=false

### The ignore urls of auth
nacos.security.ignore.urls=/,/error,/**/*.css,/**/*.js,/**/*.html,/

### The auth system to use, currently only 'nacos' and 'ldap' is supported
nacos.core.auth.system.type=nacos

### If turn on auth system:
nacos.core.auth.enabled=false

### Turn on/off caching of auth information. By turning on this switch, the
nacos.core.auth.caching.enabled=true

```

```

### Since 1.4.1, Turn on/off white auth for user-agent: nacos-server
nacos.core.auth.enable.userAgentAuthWhite=false

### Since 1.4.1, worked when nacos.core.auth.enabled=true and nacos.core.auth.type=nacos
### The two properties is the white list for auth and used by identity
nacos.core.auth.server.identity.key=
nacos.core.auth.server.identity.value=

### worked when nacos.core.auth.system.type=nacos
### The token expiration in seconds:
nacos.core.auth.plugin.nacos.token.cache.enable=false
nacos.core.auth.plugin.nacos.token.expire.seconds=18000
### The default token (Base64 String):
nacos.core.auth.plugin.nacos.token.secret.key=


### worked when nacos.core.auth.system.type=ldap, {0} is Placeholder
#nacos.core.auth.ldap.url=ldap://localhost:389
#nacos.core.auth.ldap.basedc=dc=example,dc=org
#nacos.core.auth.ldap.userDn=cn=admin,${nacos.core.auth.ldap.basedc}
#nacos.core.auth.ldap.password=admin
#nacos.core.auth.ldap.userdn=cn={0},dc=example,dc=org
#nacos.core.auth.ldap.filter.prefix=uid
#nacos.core.auth.ldap.case.sensitive=true

***** Istio Related Configurations ****#
### If turn on the MCP server:
nacos.istio.mcp.server.enabled=false

***** Core Related Configurations ****#
### set the WorkerID manually

```

```

# nacos.core.snowflake.worker-id=

#### Member-MetaData
# nacos.core.member.meta.site=
# nacos.core.member.meta.adweight=
# nacos.core.member.meta.weight=


#### MemberLookup
#### Addressing pattern category, If set, the priority is highest
# nacos.core.member.lookup.type=[file,address-server]
## Set the cluster list with a configuration file or command-line arguments
# nacos.member.list=192.168.16.101:8847?raft_port=8807,192.168.16.101:8847?raft_port=8807
## for AddressServerMemberLookup
# Maximum number of retries to query the address server upon initialization
# nacos.core.address-server.retry=5
## Server domain name address of [address-server] mode
# address.server.domain=jmenv.tbsite.net
## Server port of [address-server] mode
# address.server.port=8080
## Request address of [address-server] mode
# address.server.url=/nacos/serverlist

***** JRaft Related Configurations *****

```

```

#### Sets the Raft cluster election timeout, default value is 5 seconds
# nacos.core.protocol.raft.data.election_timeout_ms=5000
#### Sets the amount of time the Raft snapshot will execute periodically
# nacos.core.protocol.raft.data.snapshot_interval_secs=30
#### raft internal worker threads
# nacos.core.protocol.raft.data.core_thread_num=8
#### Number of threads required for raft business request processing
# nacos.core.protocol.raft.data.cli_service_thread_num=4

```

```
### raft linear read strategy. Safe linear reads are used by default
# nacos.core.protocol.raft.data.read_index_type=ReadOnlySafe

### rpc request timeout, default 5 seconds
# nacos.core.protocol.raft.data.rpc_request_timeout_ms=5000

***** Distro Related Configurations *****

### Distro data sync delay time, when sync task delayed, task will
# nacos.core.protocol.distro.data.sync.delayMs=1000

### Distro data sync timeout for one sync data, default 3 seconds.
# nacos.core.protocol.distro.data.sync.timeoutMs=3000

### Distro data sync retry delay time when sync data failed or timed out
# nacos.core.protocol.distro.data.sync.retryDelayMs=3000

### Distro data verify interval time, verify synced data whether exists
# nacos.core.protocol.distro.data.verify.intervalMs=5000

### Distro data verify timeout for one verify, default 3 seconds.
# nacos.core.protocol.distro.data.verify.timeoutMs=3000

### 加载快照数据失败时，分区数据加载重试的延迟时间，默认为 30 秒。
# nacos.core.protocol.distro.data.load.retryDelayMs=30000

### 启用以支持 prometheus 服务发现
#nacos.prometheus.metrics.enabled=true

### Since 2.3
***** Grpc Configurations *****

## sdk grpc(between nacos server and client) configuration
```

```
## Sets the maximum message size allowed to be received on the serv
#nacos.remote.server.grpc.sdk.max-inbound-message-size=10485760
```

```
## 设置发送 keepalive ping 之前无读取活动的时间（毫秒）。典型的默认值是
#nacos.remote.server.grpc.sdk.keep-alive-time=7200000
```

```
## 设置发送 keepalive ping 后等待读取活动的时间（毫秒）。默认为 20 秒。
#nacos.remote.server.grpc.sdk.keep-alive-timeout=20000
```

```
## 设置时间（毫秒），指定允许客户端配置的最长保持连接时间。典型的默认值是
#nacos.remote.server.grpc.sdk.permit-keep-alive-time=300000
```

```
## cluster grpc(inside the nacos server) configuration
#nacos.remote.server.grpc.cluster.max-inbound-message-size=10485760
```

```
## 设置发送 keepalive ping 之前无读取活动的时间（毫秒）。典型的默认值是
#nacos.remote.server.grpc.cluster.keep-alive-time=7200000
```

```
## 设置发送 keepalive ping 后等待读取活动的时间（毫秒）。默认为 20 秒。
#nacos.remote.server.grpc.cluster.keep-alive-timeout=20000
```

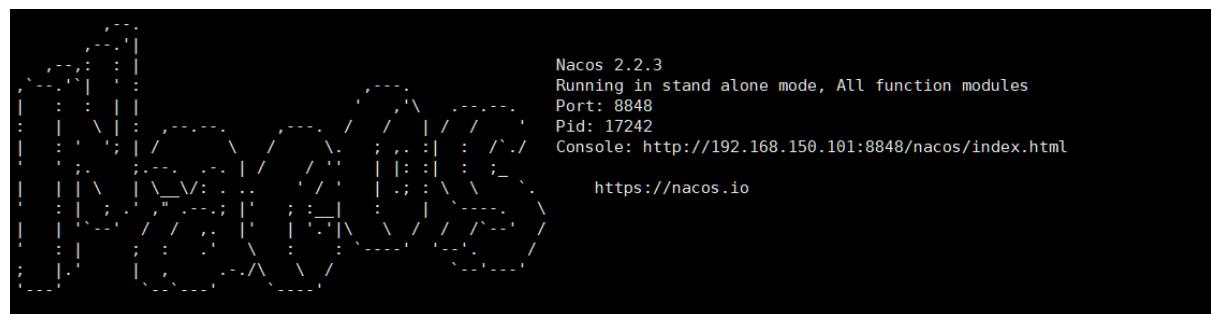
```
## 设置时间（毫秒），指定允许客户端配置的最长保持连接时间。典型的默认值是
#nacos.remote.server.grpc.cluster.permit-keep-alive-time=300000
```

我们一开始仅需关注权限认证和数据库配置的配置文件即可，后续的参数配置在使用时再进行探讨。接下来我们来进行启动nacos。我们先不开启认证，先进行体验，后续再进行开启认证。

如果启动过程中出现libstdc++.so.6: cannot open shared object file: No such file or directory, 大概是共享库有所缺失, 我们可以通过一位网友的做法来进行解决。解决方案

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

```
[root@hadoop-father bin]# sh startup.sh -m standalone
/opt/jdk1.8.0_201/bin/java -Djava.ext.dirs=/opt/jdk1.8.0_201/jre/lib/ext:/opt/jdk1.8.0_201/lib/ext -Xms512m -Xmx512m -Xmn256m -Dnacos.standalone=true -Dnacos.member.list= -Xloggc:/soft/nacos/nacos_gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileMaxSize=100M -Dloader.path=/soft/nacos/nacos/plugins,/soft/nacos/nacos/plugins/health,/soft/nacos/nacos/plugins/cmdb,/soft/nacos/nacos/plugins(selector -Dnacos.home=/soft/nacos/nacos -jar /soft/nacos/nacos/target/nacos-server.jar --spring.config.additional-location=file:/soft/nacos/nacos/conf/ --logging.config=/soft/nacos/nacos/conf/nacos-logback.xml --server.max-http-header-size=524288
nacos is starting with standalone
nacos is starting, you can check the /soft/nacos/nacos/logs/start.out
[root@hadoop-father bin]#
```



启动成功，然后我们进行访问。

此时我们可以先自己进行稍微体验一下，再然后我们将开始docker部署。

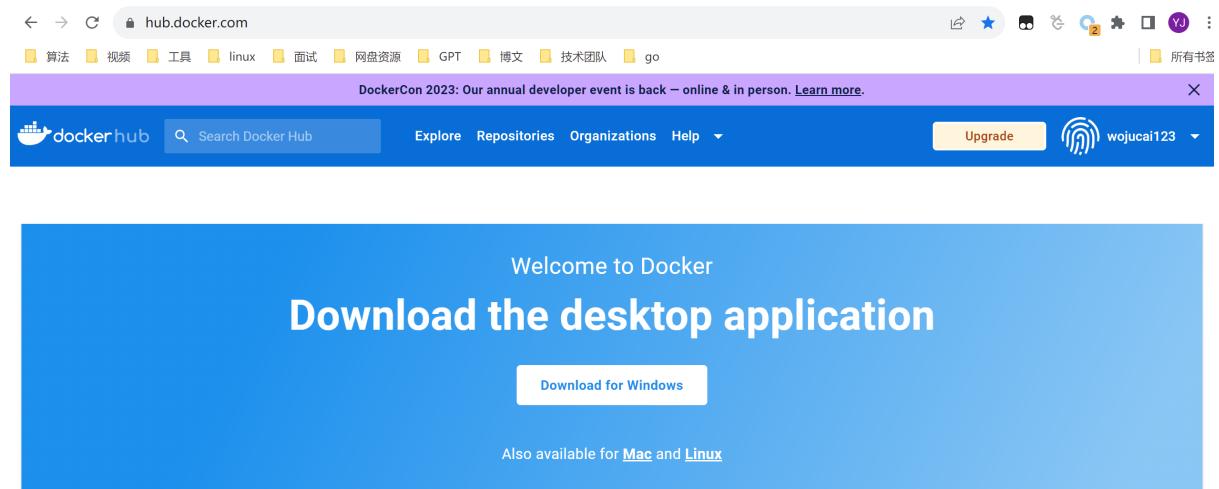
## 1.5.2 docker部署启动

如果你学习过docker的话就会大概知道我们要安装一个应用的话需要先下载镜像，然后通过镜像来进行容器的获取。[dockerhub](#)是一个可以下载镜像的网站，当然国内也有自己的镜像，你也可以选择自己搭一个镜像托管的网站或是自己写一个镜像(不推荐，毕竟前人已经实现了)，由于官网经常不能用，所以我们采用代理的方式来解决，

[dockerproxy](#)，目前来说做的最好的集成代理。

当然国内也有响应的镜像，[阿里云](#)、[网易云](#)，我们先要进行搜索nacos的镜像，然后安装。

我们首先进入dockerhub的官方网站，然后去搜索我们要的镜像，一般来说，对于开源的软件都有已经构建好的镜像，我们不需要自己再去构建。



然后我们来搜索nacos

The screenshot shows the Docker Hub search results for 'nacos'. The search bar at the top contains 'nacos'. Below it, there are filters for 'Products' (Images, Extensions, Plugins), 'Trusted Content' (Docker Official Image, Verified Publisher, Sponsored OSS), and 'Operating Systems' (Linux, Windows). The main results list two items:

- nacos/nacos-server** - 10M+ pulls, 430 stars. By nacos. Updated 4 months ago. This project contains a Docker image meant to facilitate the deployment of Nacos. Tags: Linux, unknown, x86-64, arm64, unknown.
- nacos/nacos-peer-finder-plugin** - 1M+ pulls, 2 stars. By nacos. Updated 2 years ago. scale plugin for nacos k8s. Tags: Linux, x86-64.

The URL in the address bar is <https://hub.docker.com/search?q=nacos>.

第一个就是我们需要的应用软件。

The screenshot shows the Docker Hub page for the **nacos/nacos-server** image. The URL in the address bar is <https://hub.docker.com/r/nacos/nacos-server>. The page has tabs for 'Overview' (selected) and 'Tags'. A red arrow points from the 'Tags' tab to the text '这个是标签对应着不同的nacos版本' (This is the tag corresponding to different Nacos versions). Another red arrow points from the 'Docker Pull Command' section to the text '这个是拉取镜像的命令' (This is the command to pull the image).

**nacos/nacos-server** ★  
By nacos • Updated 4 months ago  
This project contains a Docker image meant to facilitate the deployment of Nacos.  
Image  
Overview Tags  
**Nacos Docker**  
docker pulls 17M  
这个是使用文档，我们之后会很多的操作都是依靠它的  
This project contains a Docker image meant to facilitate the deployment of Nacos.  
中文  
Project directory  
• build: Nacos makes the source code of the docker image  
Docker Pull Command  
docker pull nacos/nacos-server

我们先去Tags中找到我们需要的版本往下看文档我们就会看到对应的启动命令，这里还是推荐先去大概的了解docker再来进行下面的安装步骤。

```
# 拉取镜像命令
docker pull nacos/nacos-server:v2.2.3

# 启动命令,因为在2.x版本后开启了grpc, 所以我们还要开启9849端口
#1、不配置数据库的启动
docker run --name nacos-test1 -e MODE=standalone -p 8848:8848 -d nacos/nacos-server:v2.2.3

# 注意要运行sql文件, 没有sql数据库会报错

#2、配置数据库
docker run \
--name nacos-test2 \
-e MODE=standalone \
-e SPRING_DATASOURCE_PLATFORM=mysql \
-e MYSQL_SERVICE_HOST=10\.102\.46\.60 \
-e MYSQL_SERVICE_DB_NAME=nacos \
-e MYSQL_SERVICE_USER=root \
-e MYSQL_SERVICE_PASSWORD=123456 \
-e MYSQL_SERVICE_DB_PARAM=characterEncoding=utf8\&connectTimeout=1000 \
-p 8848:8848 \
-p 9848:9848 \
-d nacos/nacos-server:v2.2.3
```

NACOS 2.2.3

配置管理

配置列表

历史版本

监听查询

服务管理

命名空间

集群管理

配置管理

public

创建配置 Data ID 已开启默认模糊查询 Group 已开启默认模糊查询 默认模糊匹配  查询 +

高级查询 导入配置

查询到 0 条满足要求的配置。

	Data Id	Group	归属应用	操作
没有数据				

我们可以根据文档的下面的参数表格来书写我们想要的配置项，也可以执行 docker命令进入容器内部进行修改，我比较推荐前者。

还有更方便的安装方式docker-compose, 这里就不进行过多的阐述，读者可以下去自行了解，总的来说docker的功能十分强大，它还支持内部的分配ip和网段进行网络的划分，有必要去好好的学习一下，这里我推荐一下自己的入门书籍《Docker 一从入门到实践》，今天的介绍就到此为止。

- 2.1 NacosClient

上一章我们大概了解了nacos的一些特性和基础的安装配置，下来我们将对于nacos的使用来进行讲解，对于我们来说一门技术的使用要么是对外进行提供SDK，要么是直接提供通信接口，最后便是与各种开发框架进行集成，nacos也不例外，他有着自己的client，也有和各种开源框架的集成。

我们的开发环境选择的IDEA，采用的nacos的版本为v2.2.3，因为我是根据nacos的官网开始的，所以我们可以跟随官网的例子，将步骤走一下，来领略下nacos的强大，以下的学习我们来通过[nacos-example](#)来进行学习和体验。

注意JAVA的版本最好使用1.8，不然可能会出现一系列的错误。

## 2.1 NacosClient

首先是我们需要引入nacos-client，这个是客户端SDK，我们需要导入maven坐标，我们采用JUNIT4来进行测试功能。

```
<properties>
    <!-- 2.2.3版本以上支持纯净版客户端 -->
    <nacos.version>2.2.3</nacos.version>
</properties>

<dependencies>
    <dependency>
        <!--这里我们就不采用纯净版的了-->
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>${nacos.version}</version>
    </dependency>
</dependencies>
```

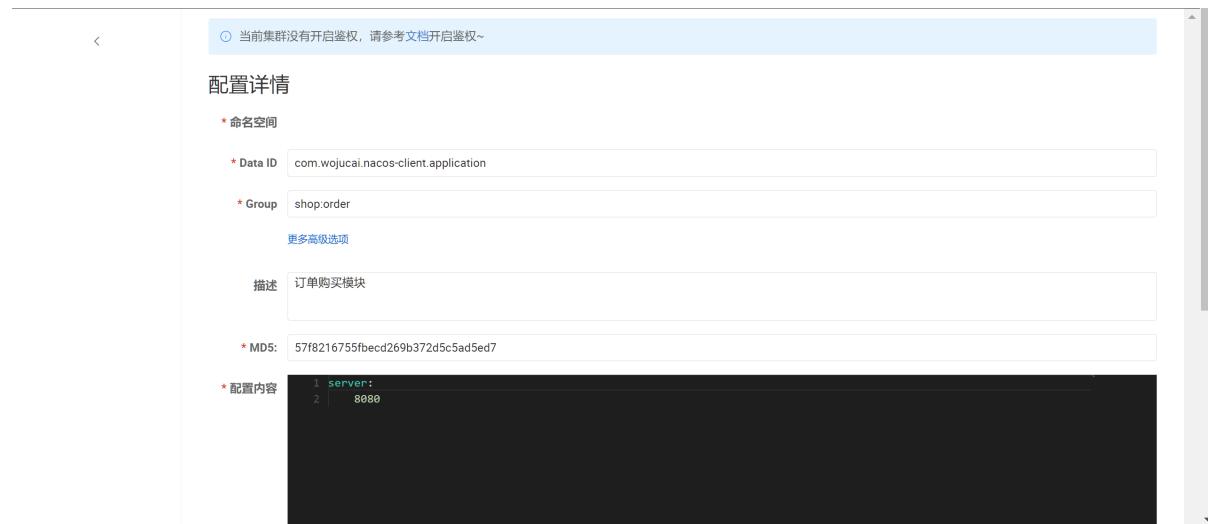
根据文档来看，我们在使用的时候需要先获取ConfigService，所以我们首先进行获取这个类，我们把它放在初始化的代码中，这样我们就可以在每一次启动的时候获取到ConfigService。我们首先需要创建dataId和groupId，如果有需要的话我们还可以新建一个namespace，主要用于配置文件的分组和隔离。

dataId一般是填写包名，保证配置文件的唯一性。

group一般是写产品名:模块名，为了区分不同模块的配置文件。

namespace一般是为了区分的不同的产品，也就是我们所说的项目。

下面是我创建的我自己的配置命名。



@Before

```
public void init() throws NacosException {  
    // nacos获取配置  
    String serverAddr = "192.168.150.101:8848";  
    Properties properties = new Properties();  
    properties.put(PropertyKeyConst.SERVER_ADDR, serverAddr);  
    configService = NacosFactory.createConfigService(properties);  
}
```

然后我们在使用完毕的时候再进行关闭。

```
@After  
public void after() throws NacosException {  
    configService.shutdown();  
}
```

然后我们来进行测试功能，各项功能皆是根据文档来的，所以我们就直接统一的贴上代码，也没有什么需要其他的注意的点，需要注意的我们会在这一章的最后一节进行统一解释。

```
package org.example;

import com.alibaba.nacos.api.NacosFactory;
import com.alibaba.nacos.api.PropertyKeyConst;
import com.alibaba.nacos.api.config.ConfigService;
import com.alibaba.nacos.api.config.listener.Listener;
import com.alibaba.nacos.api.exception.NacosException;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.util.Properties;
import java.util.concurrent.Executor;

/**
 * @description: 测试nacos
 * @author: xuyujie
 * @date: 2023/09/24
 */
public class TestNacosConfig {

    ConfigService configService;

    String dataId = "com.wojucai.nacos-client.application";
    String group = "shop:order";
    long timeout = 3000;

    @Before
    public void init() throws NacosException {
        // nacos获取配置
    }
}
```

```

        String serverAddr = "192.168.150.101:8848";
        Properties properties = new Properties();
        properties.put(PropertyKeyConst.SERVER_ADDR, serverAddr);
        // 账号密码,配置auth后启用, 后面会介绍
        // properties.put(PropertyKeyConst.USERNAME, "nacos");
        // properties.put(PropertyKeyConst.PASSWORD, "nacos");
        // 命名空间
        //properties.put(PropertyKeyConst.NAMESPACE, "ccf91393-bf7e-4b7c-a5d0-0016270ecb33");
        configService = NacosFactory.createConfigService(properties);
    }

    @After
    public void after() throws NacosException {
        configService.shutDown();
    }

    /**
     * 测试发布
     * 用于通过程序自动发布 Nacos 配置, 以便通过自动化手段降低运维成本。
     * 当配置不存在时会创建配置, 当配置已存在时会更新配置。
     *
     */
    @Test
    public void testPublishConfig() throws NacosException {
        // dataId = "com.wojucuai.nacos-client.application2";
        // group = "shop:order2";
        boolean yaml = configService.publishConfig(dataId, group,
                "8081", "yaml");
        System.out.println(yaml);
    }

    @Test

```

```
public void testUpdateConfig() throws NacosException {
    boolean yaml = configService.publishConfig(dataId, group,
        "8083", "yaml");
    System.out.println(yaml);
}

/**
 * dataId      string      配置 ID, 采用类似 package.class (如com.taobao.taobaoke.refund.log)
 * group       string      配置分组, 建议填写产品名:模块名 (Nacos:Test)
 * timeout     long        读取配置超时时间, 单位 ms, 推荐值 3000。
 * @throws NacosException
 */
@Test
public void testGetConfig() throws NacosException {
    String config = configService.getConfig(dataId, group, timeout);
    System.out.println(config);
}

/**
 * dataId
 * string
 * 配置 ID, 采用类似 package.class (如com.taobao.taobaoke.refund.log)
 * group
 * string
 * 配置分组, 建议填写产品名: 模块名 (如 Nacos:Test) 保证唯一性。
 * listener
 * Listener
 * 监听器, 配置变更进入监听器的回调函数。
 */
@Test
public void testAddListener() throws NacosException {
    configService.addListener(dataId, group, new Listener() {
```

```
    @Override
    public Executor getExecutor() {
        return null;
    }

    @Override
    public void receiveConfigInfo(String s) {
        System.out.println(s);
    }
});

// 移除监听器
// configService.removeListener(dataId, group, new Listener()
//     @Override
//     public Executor getExecutor() {
//         return null;
//     }
//
//     @Override
//     public void receiveConfigInfo(String s) {
//         System.out.println(s);
//     }
// });
// 守护线程
while (true) {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
// 取消监听
}
```

```
/*
 * 用于通过程序自动删除 Nacos 配置
 */
@Test
public void testRemoveConfig() throws NacosException {
    boolean removeConfig = configService.removeConfig(dataId, {
        System.out.println(removeConfig);
    }
}
```

上面是一些基本的和配置文件相关的使用，下面我们来看一下服务发现注册的api。

这里有几个比较重要的实例对象Cluster、Instance和Service。

我们先来看Cluster

```
public class Cluster implements Serializable {  
    /**  
     * 序列化ID  
     */  
    private static final long serialVersionUID = -7196138840047197;  
  
    /**  
     * 属于的服务的名字  
     */  
    private String serviceName;  
  
    /**  
     * 集群名  
     */  
    private String name;  
  
    /**  
     * 集群的健康检查  
     */  
    private AbstractHealthChecker healthChecker = new Tcp();  
  
    /**  
     * 在集群中注册的默认端口号  
     */  
    private int defaultPort = 80;  
  
    /**  
     * 在集群中默认健康检查的端口  
     */  
    private int defaultCheckPort = 80;
```

```
 /**
 * 是否使用实例的端口做健康检查
 */
private boolean useIPPort4Check = true;

/**
 * 元数据
 */
private Map<String, String> metadata = new HashMap<>();

}
```

然后我们再来看一下Instance

```
public class Instance implements Serializable {  
  
    private static final long serialVersionUID = -74290631056729197L;  
  
    /**  
     * 实例ID  
     */  
    private String instanceId;  
  
    /**  
     * 实例的IP  
     */  
    private String ip;  
  
    /**  
     * 实例的端口号  
     */  
    private int port;  
  
    /**  
     * 实例的权重  
     */  
    private double weight = 1.0D;  
  
    /**  
     * 实例的健康状态  
     */  
    private boolean healthy = true;  
  
    /**  
     * 实例是否接收请求  
     */
```

```
 */
private boolean enabled = true;

/**
 * 是否是短暂的实例
 *
 * @since 1.0.0
 */
private boolean ephemeral = true;

/**
 * 实例的集群名
 */
private String clusterName;

/**
 * 实例的服务名
 */
private String serviceName;

/**
 * 元数据
 */
private Map<String, String> metadata = new HashMap<>();
}
```

然后我们来看一下Service这个类

```
/**  
 * 我们引入了一个 "服务-->集群-->实例" 模型,  
 * 其中服务存储了一个集群列表, 而集群则包含一个实例列表。  
 */  
public class Service implements Serializable {  
  
    private static final long serialVersionUID = -3470985546826874L;  
  
    /**  
     * 服务名  
     */  
    private String name;  
  
    /**  
     * 保护阈值  
     */  
    private float protectThreshold = 0.0F;  
  
    /**  
     * 这个服务的应用名  
     */  
    private String appName;  
  
    /**  
     * 服务分组名, 用于将服务分为不同的组。  
     */  
    private String groupName;  
  
    /**  
     * 元数据  
     */
```

```
    private Map<String, String> metadata = new HashMap<>();  
}
```

我们先来注册一个服务，然后注册集群，然后注册群组，来体验一下不同的api。

然后就是我们测试的全部代码

```
/**  
 * @description:测试nacos服务  
 * @author: xuyujie  
 * @date: 2023/09/24  
 */  
  
public class TestNacosService {  
  
    NamingService namingService;  
  
    NamingMaintainService maintainService;  
  
    String appName = "shop";  
  
    String serviceName = "order-service";  
  
    String groupName = "MQ_topic";  
  
    @Before  
    public void init() throws NacosException {  
        String serverAddr = "192.168.150.101:8848";  
        Properties properties = new Properties();  
        properties.put(PropertyKeyConst.SERVER_ADDR, serverAddr);  
        //        properties.put(PropertyKeyConst.USERNAME, "nacos");  
        //        properties.put(PropertyKeyConst.PASSWORD, "nacos");  
        //        properties.put(PropertyKeyConst.NAMESPACE, "ccf91393-bf7e-  
        namingService = NacosFactory.createNamingService(properties);  
        maintainService = NacosFactory.createMaintainService(properties);  
    }  
  
    /**
```

```
* 创建服务
*/
@Test
public void testRegisterInstance2() throws NacosException {
    // 注册服务
    Service service = new Service();
    service.setAppName(appName);
    service.setName(serviceName);
    service.setGroupName(groupName);
    service.setProtectThreshold(1);
    Map<String, String> serviceMetadata = new HashMap<>();
    service.setMetadata(serviceMetadata);
    maintainService.createService(service, new NoneSelector());
    Service service1 = maintainService.queryService("order-serv
    // 保证服务是在线状态
    while (true) {

    }
}

/**
 * 创建集群
*/
@Test
public void testRegisterCluster() throws NacosException {
    Map<String, String> serviceMetadata = new HashMap<>();
    // 注册实例到集群
    namingService.registerInstance(serviceName+1,groupName,"11
    // 保证服务是在线状态
    while (true) {

    }
}
```

```

}

/**
 * 注册实例
 */
@Test
public void testRegisterInstance() throws NacosException {
    Map<String, String> serviceMetadata = new HashMap<>();
    // 注册实例到集群
    namingService.registerInstance(serviceName+1,groupName,"11
    // 保证服务是在线状态
    while (true) {

    }
}
}

```

要注意在注册的时候要保证线程是存活状态，只有线程存活，对应的nacos上的实例才会存活。如我现在启动两个线程，对应的nacos是如下图。

The screenshot shows the Nacos 2.2.3 web interface. The left sidebar has a '服务列表' (Service List) section selected. The main content area displays a table titled '服务列表' (Service List) with one entry:

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阀值	操作
order-service1	MQ_topic	2	2	2	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom of the page, there are pagination controls: '每页显示: 10' (Items per page: 10), '< 上一页' (Previous page), '1' (Page 1, highlighted in blue), and '下一页 >' (Next page).

当我把服务关闭时，又会变成下图

NACOS 2.2.3

当前集群没有开启鉴权, 请参考[文档](#)开启鉴权~

## 服务列表

public

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
order-service1	MQ_topic	0	0	0	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

每页显示: 10 < 上一页 1 下一页 >

以上便是我们这节测试它的一些Client功能。

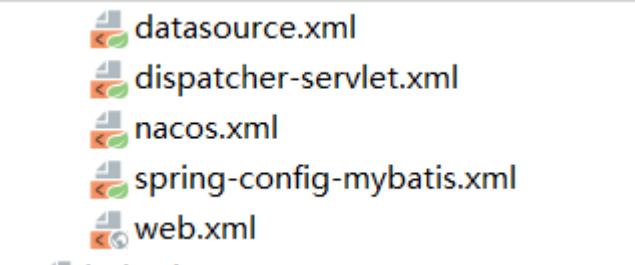
- 2.2 Nacos集成Spring
  - 2.2.1 Nacos配置数据库
  - 2.2.2 Spring配置注发布到Nacos
  - 2.2.3 Spring集成nacos实现配置监听
  - 2.2.4 Spring集成多数据项配置
  - 2.2.5 nacos在POJO对象属性生效
  - 2.2.6 spring注册服务到nacos
  - 2.2.7 总结

## 2.2 Nacos集成Spring

下面我将对于nacos集成Spring来做进一步的学习，我们来进行参照Nacos官方给的example来继续学习。

### 2.2.1 Nacos配置数据库

这个项目是nacos-spring-example的一个子项目，主要用于使用nacos来配置数据库，在WEB-INF文件夹下也有所有的配置文件，这是spring-web的基本的配置文件，我们大概来一个个了解下



datasource.xml  
dispatcher-servlet.xml  
nacos.xml  
spring-config-mybatis.xml  
web.xml

首先是datasource.xml，主要用于配置和数据库相关的属性。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/beans">

    <!--配置数据源和线程池-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataS
        init-method="init" destroy-method="close">
        <property name="url" value="${datasource.url}" />
        <property name="username" value="${datasource.username}" />
        <property name="password" value="${datasource.password}" />
        <property name="initialSize" value="${datasource.initial-s:}>
        <property name="maxActive" value="${datasource.max-active}" />
    </bean>

    <!-- 配置事务管理器-->
    <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTran
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!--事务注解的驱动-->
    <tx:annotation-driven transaction-manager="txManager" />
</beans>
```

然后是dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- 开启Spring mvc 的一系列配置视图转发， json序列化等等 -->
    <mvc:annotation-driven/>

    <!-- 自动的声明一些上下文注解， 像@Autowired -->
    <context:annotation-config/>

    <!-- 配置组件扫描的路径 -->
    <context:component-scan base-package="com.alibaba.nacos.example">
        <!-- 导入其他的配置文件 -->
        <import resource="nacos.xml"/>
        <import resource="datasource.xml"/>
        <import resource="spring-config-mybatis.xml"/>
    </beans>
```

再然后是nacos.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:nacos="http://nacos.io/schema/nacos"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://nacos.io/schema/nacos
    http://nacos.io/schema/nacos.xsd">

    <!-- 开启nacos的注解驱动 -->
    <nacos:annotation-driven/>
    <!-- 配置nacos的服务地址,这里没有详细的配置一些组和命名空间。这里还
    <nacos:global-properties server-addr="127.0.0.1:8848" />

    <!--
        Nacos 控制台添加配置:
        Data ID:
            datasource.properties
        Group:
            DEFAULT_GROUP
        配置内容示例:
            datasource.url=jdbc:mysql://localhost:3306/test?useSSL=
            datasource.username=root
            datasource.password=root
            datasource.initial-size=10
            datasource.max-active=20
        -->
        <!-- 配置数据源 -->
        <nacos:property-source data-id="datasource.properties"/>
    </beans>
```

## spring-config-mybatis.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 配置mybatis的数据源 -->
    <bean class="org.mybatis.spring.SqlSessionFactoryBean"
          id="sqlSessionFactory">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 配置mybatis -->
    <bean class="org.mybatis.spring.mapper.MapperFactoryBean"
          id="userMapper">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
        <property name="mapperInterface" value="com.alibaba.nacos.example.mapper.UserMapper"/>
    </bean>
</beans>
```

## web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" metadata-complete="true" version="3.0">
    <!-- 配置视图转发 -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>

```

如果不配置直接进行启动就会报错，如下图：

```

29-Sep-2023 10:40:31.208 警告 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.context.support.XmlWebApplicationContext.refres
29-Sep-2023 10:40:31.209 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.beans.factory.support.DefaultListableBeanFactory
29-Sep-2023 10:40:31.209 严重 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.servlet.DispatcherServlet.initServletBean Co
    org.springframework.beans.factory.BeanDefinitionStoreException: Invalid bean definition with name 'dataSource' defined in Servl
        at org.springframework.beans.factory.config.PlaceholderConfigurerSupport.doProcessProperties(PlaceholderConfigurerSupport.j
        at org.springframework.context.support.PropertySourcesPlaceholderConfigurer.processProperties(PropertySourcesPlaceholderCon
        at org.springframework.context.support.PropertySourcesPlaceholderConfigurer.postProcessBeanFactory(PropertySourcesPlaceholder
        at org.springframework.context.support.AbstractApplicationContext.invokeBeanFactoryPostProcessors(AbstractApplicationContex
        at org.springframework.context.support.AbstractApplicationContext.invokeBeanFactoryPostProcessors(AbstractApplicationContex
        at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:464) <6 internal l
        at javax.servlet.GenericServlet.init(GenericServlet.java:158) <15 internal lines>
        at java.management/com.sun.jmx.interceptor.DefaultMBeanServerInterceptor.invoke(DefaultMBeanServerInterceptor.java:809)
        at java.management/com.sun.jmx.mbeanserver.JmxMBeanServer.invoke(JmxMBeanServer.java:801) <7 internal lines>
        at java.management/com.sun.jmx.interceptor.DefaultMBeanServerInterceptor.invoke(DefaultMBeanServerInterceptor.java:809)
        at java.management/com.sun.jmx.mbeanserver.JmxMBeanServer.invoke(JmxMBeanServer.java:801)
        at java.management/com.sun.jmx.remote.security.MBeanServerAccessController.invoke(MBeanServerAccessController.java:468)
        at java.management.rmi/javax.management.remote.rmi.RMIClientImpl.doOperation(RMIClientImpl.java:1466)
        at java.management.rmi/javax.management.remote.rmi.RMIClientImpl$PrivilegedOperation.run(RMIClientImpl.java:1307) <1
        at java.management.rmi/javax.management.remote.rmi.RMIClientImpl.doPrivilegedOperation(RMIClientImpl.java:1406)
        at java.management.rmi/javax.management.remote.rmi.RMIClientImpl.invoke(RMIClientImpl.java:827) <17 internal lines>
Caused by: java.lang.IllegalArgumentException: Could not resolve placeholder 'datasource.url' in string value "${datasource.url}"
        at org.springframework.util.PropertyPlaceholderHelper.parseStringValue(PropertyPlaceholderHelper.java:173)
        at org.springframework.util.PropertyPlaceholderHelper.replacePlaceholders(PropertyPlaceholderHelper.java:125)
        at org.springframework.core.env.AbstractPropertyResolver.doResolvePlaceholders(AbstractPropertyResolver.java:190)

```

按照提示我们需要在控制台添加配置，我们添加下配置，然后来看效果。

The screenshot shows the Nacos configuration interface. A new configuration named 'datasource.properties' is being created under the group 'DEFAULT\_GROUP'. The configuration content is as follows:

```
1 datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false
2 datasource.username=root
3 datasource.password=123456
4 datasource.initial-size=10
5 datasource.max-active=20
```

启动成功，说明我们的应用已经从nacos上拉取下来了配置文件。

The screenshot shows the IntelliJ IDEA Services tool window. It displays logs for a Tomcat 9.0.50 server. The logs show the application starting up and successfully loading the Nacos configuration.

```
java.net.URLClassLoader@79e2c065
JM.Log:INFO Log root path: C:\Users\Xuyujie\logs\
Fri Sep 29 11:45:44 GMT+08:00 2023 ParallelWebappClassLoader
context: ROOT
delegate: false
-----> Parent ClassLoader:
java.net.URLClassLoader@79e2c065
JM.Log:INFO Set nacos log path: C:\Users\Xuyujie\logs\nacos
11:45:44.262 [RMI TCP Connection(3)-127.0.0.1] INFO com.alibaba.nacos.client.identify.CredentialWatcher - [] [] [] No credential found
29-Sep-2023 11:45:44.364 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.context.support.XmlWebApplicationContext.postProcessAft
29-Sep-2023 11:45:44.457 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.context.support.XmlWebApplicationContext.postProcessAft
29-Sep-2023 11:45:44.459 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.context.support.XmlWebApplicationContext.postProcessAft
29-Sep-2023 11:45:44.472 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstant
11:45:44.571 [RMI TCP Connection(3)-127.0.0.1] DEBUG org.apache.ibatis.logging.LogFactory - Logging initialized using "Class org.apache.ibatis
11:45:44.887 [RMI TCP Connection(3)-127.0.0.1] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-1} init
11:45:44.892 [RMI TCP Connection(3)-127.0.0.1] DEBUG org.mybatis.spring.SqlSessionFactoryBean - Property 'configuration' or 'configLocation' n
11:45:44.943 [RMI TCP Connection(3)-127.0.0.1] DEBUG org.mybatis.spring.SqlSessionFactoryBean - Property 'mapperLocations' was not specified o
29-Sep-2023 11:45:44.998 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.method.annotation.RequestMappingHandlerMapp
11:45:45.404 [RMI TCP Connection(3)-127.0.0.1] INFO com.alibaba.nacos.spring.context.event.LoggingNacosConfigMetadataEventListener - Nacos Con
29-Sep-2023 11:45:45.411 信息 [RMI TCP Connection(3)-127.0.0.1] org.springframework.web.servlet.DispatcherServlet.initServletBean FrameworkServ
[2023-09-29 11:45:45.429] Artifact nacos-spring-config-datasource-example:war exploded: Artifact is deployed successfully
[2023-09-29 11:45:45.429] Artifact nacos-spring-config-datasource-example:war exploded: Deploy took 3,704 milliseconds
29-Sep-2023 11:45:51.319 信息 [Catalina-utility-1] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [E:\IntelliJ ID
29-Sep-2023 11:45:51.354 信息 [Catalina-utility-1] org.apache.catalina.startup.HostConfig.deployDirectory Web应用程序目录[E:\IntelliJ IDEA2020(64
```

查询到了数据库中的值。

The screenshot shows a browser window with the URL 'localhost:8080/user?id=1'. The page displays the JSON response from the database query.

```
{"id":1,"name":"Nacos"}
```

以上便是Spring和nacos集成实现的最基本的拉取配置的实例。

## 2. 2. 2 Spring配置注发布到Nacos

Spring的应用程序本身也是向外提供服务的，所以它也可以作为一个实例将配置文件发布到nacos中，供其他应用程序进行调用。接下来我们要接触的实例是nacos-spring-config-example。

配置文件基本和上一部分无异，我们主要关注如何把这个服务给注册到Nacos中。

奥妙就在于NacosConfiguration这个类，我们需要通过这个类来向nacos发布配置。

```
// 保证这个类为Spring的配置类， 可以被Spring管理。
@Configuration
// nacos的服务器的一系列配置项
@EnableNacosConfig(globalProperties = @NacosProperties(serverAddr =
/**
 * Document: https://nacos.io/zh-cn/docs/quick-start-spring.html
 * <p>
 * Nacos 控制台添加配置：
 * <p>
 * Data ID: example
 * <p>
 * Group: DEFAULT_GROUP
 * <p>
 * 配置内容: useLocalCache=true
 */
// 配置属性所属的数据源
@NacosPropertySource(dataId = "example", autoRefreshed = true)
public class NacosConfiguration {

}
```

然后就是控制属性的类了

```
// 声明控制器
@Controller
@RequestMapping("config")
public class ConfigController {

    // 注入nacos的配置服务，主要用于向nacos进行发布配置
    @NacosInjected
    private ConfigService configService;
    // 监听nacos的值， 如果更新的话也可以自动更新
    @NacosValue(value = "${useLocalCache:false}", autoRefreshed = true)
    private boolean useLocalCache;

    // 获取值
    @RequestMapping(value = "/get", method = GET)
    @ResponseBody
    public boolean get() {
        return useLocalCache;
    }

    // 向nacos发布配置
    @RequestMapping(method = POST)
    @ResponseBody
    public ResponseEntity<String> publish(@RequestParam String dataId,
                                            @RequestParam(defaultValue = "dev") String group,
                                            @RequestParam String configName,
                                            boolean result = false);
    try {
        result = configService.publishConfig(dataId, group, configName);
    } catch (NacosException e) {
        return new ResponseEntity<String>("Publish Fail:" + e.getMessage());
    }
}
```

```

if (result) {
    return new ResponseEntity<String>("Publish Success", HttpStatus.OK);
}
return new ResponseEntity<String>("Publish Fail, Retry", HttpStatus.OK);
}

```

我们将容器启动后，发现访问后返回的是false，这是因为它具有默认值，然后我们按照上文指定的DataId去修改并发布配置，我们会发现一个新的世界。

美观输出 □  
true

然后我们来使用api工具来测试下发布配置，我们来把它重新改为false。

Add a description

METHOD: POST SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
http://localhost:8080/config length: 28 byte(s)

HEADERS: Content-Type: application/x-www-form-urlencoded  
+ Add header    Add authorization

QUERY PARAMETERS

BODY: dataId: example  
content: useLocalCache=false  
+ Add form parameter    application/x-www-form-urlencoded

Request preview

200

HEADERS: Content-Type: text/plain; charset=ISO-8859-1  
Content-Length: 15  
Date: Fri, 29 Sep 2023 04:22:59 GMT  
Keep-Alive: timeout=20  
Connection: keep-alive

BODY: Publish Success

可以看到发布成功，那我们就不再看具体的结果了。总结就是，我们可以通过注解@EnableNacosConfig(globalProperties = @NacosProperties(serverAddr = "192.168.150.101:8848"))

来进行配置nacos的服务器属性，然后他就能和nacos服务器通信了，然后我们通过@NacosPropertySource(dataId = "example", autoRefreshed = true)

为对应的配置类绑定dataId因为dataId是唯一的，当然我们还可以配置group等，具体的属性可以自己去观看对应的注解类。

然后我们使用@NacosInjected来注入ConfigService用于发布配置，使用@NacosValue来绑定属性对应的配置值。

## 2. 2. 3Spring集成nacos实现配置监听

首先就是配置文件，对于这个项目来说的话，配置文件也是只有web.xml和dispatcherServlet-servlet.xml两个文件，基本与上述配置无异，这里是讲述监听这个模块的，我们着重的来看一下如何实现监听。

和上面的一样要先配置对应的注册中心服务器的地址。

```
@Configuration  
@EnableNacosConfig(globalProperties = @NacosProperties(serverAddr :  
public class AdminConfiguration {  
  
}  
◀ ▶
```

```
// 实现服务监听的类
@Service
public class AdminServiceImpl implements AdminService {

    private static final Logger LOGGER = LoggerFactory.getLogger(AdminService.class);
    // dataId
    private static final String ADMIN_DATA_ID = "admin.json";
    // groupId
    private static final String ADMIN_GROUP_ID = "spring-listener";

    private volatile Admin admin;

    // 这个用于监听，获取json的数据的值
    @NacosConfigListener(dataId = ADMIN_DATA_ID, groupId = ADMIN_GROUP_ID)
    public void onReceived(String content) {
        LOGGER.info("onReceived(String) : {}", content);
    }

    /**
     * <p>
     * Nacos 控制台添加配置：
     * <p>
     * Data ID: admin.json
     * <p>
     * Group: spring-listener
     * <p>
     * 配置内容：
     * {
     *     "username": "admin",
     *     "password": "123456"
     * }
    
```

```
 */
// 可以直接转化为JSON
@NacosConfigListener(dataId = ADMIN_DATA_ID, groupId = ADMIN_GROUP_ID)
public void onReceived(Admin admin) {
    LOGGER.info("onReceived(Admin) : {}", admin);
    this.admin = admin;
}

@Override
public Admin getAdmin() {
    return admin;
}
}
```

我们来向nacos的页面添加json数据，然后来看看是否可以反序列化为JAVA对象。

```
{
    "username": "admin",
    "password": "123456"
}
```

可以获取到值，然后我们再来看一下控制台是否有监听的内容：

```
22:36:46.063 [pool-2-thread-1] INFO com.alibaba.nacos.example.spring.service.impl.AdminServiceImpl - onReceived(String) : {
    "username": "admin",
    "password": "123456"
}
22:36:46.176 [pool-1-thread-1] INFO com.alibaba.nacos.example.spring.service.impl.AdminServiceImpl - onReceived(Admin) : Admin{username='admin', password='123456'}
```

这里也有，由此我们便知道了配置绑定的作用，可以实时的刷新配置文件。

这个配置监听主要有两点：

其一是注解@NacosConfigListener (dataId = ADMIN\_DATA\_ID, groupId = ADMIN\_GROUP\_ID, converter = AdminConverter.class)，用于声明监听器，以及转化对象，由此便可以进行对应的监听了。

其二是对应的JAVA对象必须是volatile修饰的，因为这样的话，一旦值被修改就会马上刷新到主内存中，从而保证配置是最新的配置。

## 2. 2. 4 Spring集成多数据项配置

在上面我们只是简单的对于单个数据进行了配置，下来我们将了解一下多数据配置。nacos-spring-config-multi-data-ids-example用于集成了数据库和缓存Redis的用法。

首先它的配置没有多大的变更，依然是dispatcherServlet-servlet.xml, web.xml, cache.xml, datasource.xml。

这次的话我们需要在控制台上配备多个数据。

主要的作用类是NacosConfiguration，这个类用于实现多个配置注册。

```
package com.alibaba.nacos.example.spring;

import com.alibaba.nacos.api.annotation.NacosProperties;
import com.alibaba.nacos.spring.context.annotation.config.EnableNacosConfig;
import com.alibaba.nacos.spring.context.annotation.config.NacosProperties;
import com.alibaba.nacos.spring.context.annotation.config.NacosPropertySource;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableNacosConfig(globalProperties = @NacosProperties(serverAddr =
    "127.0.0.1:8848"))

@NacosPropertySources({
    /*
     * Nacos 控制台添加配置:
     * Data ID: app.properties
     * Group: multi-data-ids
     * 配置内容: app.user.cache=false
     */
    @NacosPropertySource(dataId = "app.properties", groupId = "multi-data-ids",
        autoRefreshed = true)
})

/*
 * 1. 本地安装 MySQL
 * 2. Nacos 控制台添加配置:
 * Data ID: datasource.properties
 * Group: multi-data-ids
 * 配置内容示例:
 *     spring.datasource.url=jdbc:mysql://localhost:3306/test?use
 *     spring.datasource.username=root
 *     spring.datasource.password=root
 *     spring.datasource.initial-size=10
 */
```

```
*   spring.datasource.max-active=20
*/
@NacosPropertySource(dataId = "datasource.properties", groupId

/*
 * 1. 本地安装 Redis
 * 2. Nacos 控制台添加配置:
 * Data ID: redis.properties
 * Group: multi-data-ids
 * 配置内容示例:
 *   spring.redis.host=localhost
 *   spring.redis.password=20190101
 *   spring.redis.timeout=5000
 *   spring.redis.max-idle=5
 *   spring.redis.max-active=10
 *   spring.redis.max-wait=3000
 *   spring.redis.test-on-borrow=false
*/
@NacosPropertySource(dataId = "redis.properties", groupId = "mi
})
public class NacosConfiguration {

}
```

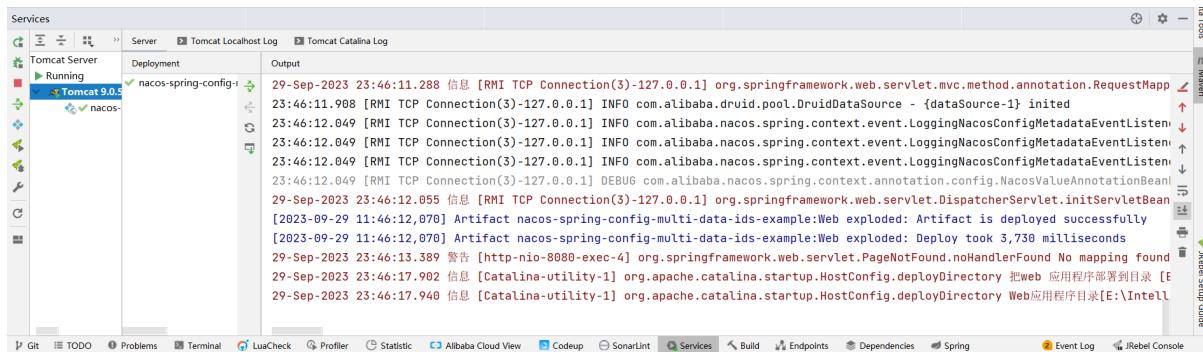
按照需求我们将数据发布到nacos注册中心上，然后我们进行应用的启动，观察是否可以启动成功。

[redis](#)和[mysql](#)如果有需要的话，我们还可以按照我们先前教的使用Docker来进行安装，也会省去很多的麻烦。

```
docker run -p 6379:6379 --name some-redis -d redis
```

然后我们配置并启动，后进行观察。

注意在启动的时候，数据库的依赖要和连接驱动的版本一致。



这个主要是通过配置文件控制是否开启缓存，与上一个也是类似也不进行多讲，这里主要讲述一个思想，也就是配置是多个数据源，但是是在一个分组里面，比如像Redis等一系列通用的配置是可以放在一个分组里面进行重复使用的。

而多数据源正式通过注解实现的@NacosPropertySources，这个注解里面可以存放来自不同数据源的配置信息，之后如果使用nacos的时候可能会很常用到。对于一些比较雷同的配置就不再做过多的解释了，如果有不懂的话可以在评论区进行留言，看到的话会进行回复。

## 2. 2. 5nacos在POJO对象属性生效

接下来我们要看的项目是nacos-spring-config-pojo-example这个小实例，这个实例讲述的是nacos如何在pojo对象实例上生效，进而做到实时的更新pojo实例对象属性。

这个项目的配置和前几个无异因为都是Spring项目，我们着重来看配置类PromotionConfiguration，这个类是配置类声明了一个Bean对象交由Spring管理。

```
@Configuration  
@EnableNacosConfig(globalProperties = @NacosProperties(serverAddr :  
public class PromotionConfiguration {  
  
    @Bean  
    public Promotion promotion() {  
        return new Promotion();  
    }  
  
}
```

Promotion对象则是被加上了nacos的注解

@NacosConfigurationProperties、@NacosProperty(value = "desc")

、@NacosIgnore。这三个注解。

第一个注解则是声明nacos配置的数据源和分组Id，第二个注解是为属性绑定实例的属性，可以起别名，第三个注解则是进行忽略这个POJO属性，忽略从 NacosConfigurationProperties 来的属性对象。

```
@NacosConfigurationProperties(dataId = "promotion.properties", group = "group1")
public class Promotion {

    private long sku;

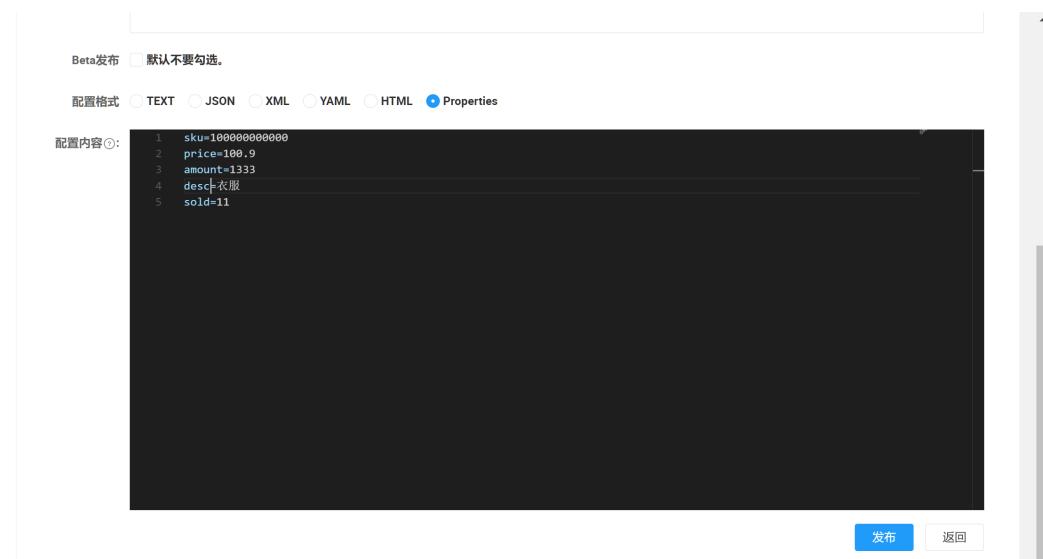
    private double price;

    private int amount;

    @NacosProperty(value = "desc")
    private String description;

    @NacosIgnore
    private int sold;
}
```

我们来在控制台添加对应的pojo属性，然后启动看是否符合我们自己的预期。



请注意一点，因为对于有的属性我们已经起了别名，那么对于之前的那个名称就会失效，所以一定要保证属性照应，而且因为加上了自动刷新的注解，所以对于属性的更新也是会更新的。

```
美观输出 □
{"sku":10000000000, "price":100.9, "amount":1333, "description":"衣服", "sold":0}
```

这样的话对于属性上的注解我们也进行了大概的演示。

## 2. 2. 6spring注册服务到nacos

通过前面的几个应用，我们大概了解了属性配置的一些内容，接下来我们将了解spring的应用获取注册到nacos中的服务，算是服务发现模块的，应用的名称是nacos-spring-discovery-example，基本的配置合并前几节无异，我们也是直接来看如何实现的。

首先就是NacosConfiguration这个类，配置了一定的属性注解。

```
@Configuration
// 和上面不同这个是开启了服务发现。
@EnableNacosDiscovery(globalProperties = @NacosProperties(serverAddres...
public class NacosConfiguration {
```

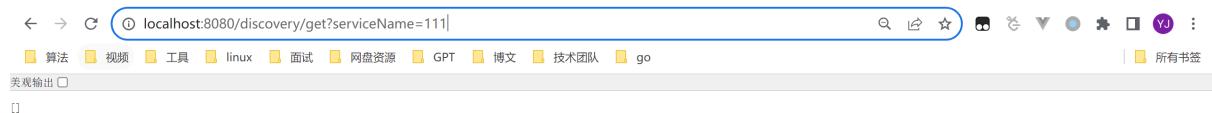
}



然后便是DiscoveryController，控制器获取所有的服务属性。

```
@Controller  
@RequestMapping("discovery")  
public class DiscoveryController {  
  
    // 注入nacos独有的服务  
    @NacosInjected  
    private NamingService namingService;  
  
    @RequestMapping(value = "/get", method = GET)  
    @ResponseBody  
    public List<Instance> get(@RequestParam String serviceName) thi  
        return namingService.getAllInstances(serviceName);  
    }  
}
```

然后我们来进行启动查看。



可以看到无服务，然后我们来注册一个，再来观察。

这里暂时有点问题，使用client时可以正常获取使用nacos的Spring注入时就无法获取，可能是因为版本不对应的问题，之后我再尝试下。

由此我们就将nacos和Spring的集成大概算是学习完毕了，之后我们将开启新的篇章。

## 2.2.7总结

这一章大概是结束了，体验了nacos的核心特性，使用了注册配置和注册服务，总的来说配置的东西挺多的，再下来的集成项目中，需要我们配置的东西会逐渐变少，加油。

- 2.3 Nacos集成SpringBoot
  - 2.3.1 SpringBoot属性配置
  - 2.3.2 SpringBoot集成配置数据库
  - 2.3.3 SpringBoot 服务发现
  - 2.3.4 总结

## 2.3 Nacos集成SpringBoot

SpringBoot相当于是Spring的升级版，集成了各种的自动配置，同时内嵌了tomcat容器，这样的话也方便了部署和测试，相较于上一章冗余的配置，这一章会更加的简介。

### 2.3.1 SpringBoot属性配置

因为对于SpringBoot来说，一些基本的配置已经被配置好了，我们只需要去更新必要的配置，如nacos的服务地址。然后直接使用即可，使用的方法也和Spring差不多。这个项目是nacos-spring-boot-config-example

首先我们来看一下application.properties这个文件：

```
# 修改nacos的配置的地址
nacos.config.server-addr=192.168.150.101:8848
# 下面是Spring应用的健康审查
# endpoint http://localhost:8080/actuator/nacos-config
# health http://localhost:8080/actuator/health
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

然后我们来看下控制层基本上就可以了，可以发现controller和之前的Spring的配置无太大的差别。

```
@Controller
@RequestMapping("config")
public class ConfigController {

    @NacosValue(value = "${useLocalCache:false}", autoRefreshed = true)
    private boolean useLocalCache;

    @RequestMapping(value = "/get", method = GET)
    @ResponseBody
    public boolean get() {
        return useLocalCache;
    }
}
```

然后是启动类, 基本上无太大的差别

```
@SpringBootApplication
@NacosPropertySource(dataId = "example", autoRefreshed = true)
public class NacosConfigApplication {

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

当我们修改值后再进行访问就会得到我们修改的值

美观输出

true

## 2.3.2 SpringBoot集成配置数据库

这一小节对应的实例是nacos-spring-boot-config-mysql-example这个小项目。最主要的是在启动类添加了@NacosPropertySource(dataId = "mysql.properties")这个注解，然后就会从nacos中拉取对应的文件内容。

```
@SpringBootApplication
@NacosPropertySource(dataId = "mysql.properties")
public class SpringBootMySQLApplication {

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

我们需要先在nacos中创建mysql.properties的配置文件，然后才能启动应用，不然会报错。

```
2023-09-30 14:10:26.524 INFO 11676 --- [           main] o.apache.catalina.core.StandardService : Stopping service [Tomcat]
2023-09-30 14:10:26.539 INFO 11676 --- [           main] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2023-09-30 14:10:26.543 ERROR 11676 --- [           main] o.s.b.d.LoggingFailureAnalysisReporter :

*****
APPLICATION FAILED TO START
*****

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:
 If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
 If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

Process finished with exit code 1
```

mysql.properties

```

spring.datasource.url=jdbc:mysql://localhost:3306/user?useUnicode=1&characterEncoding=utf8
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.initial-size=10
spring.datasource.max-active=20

```

Run: SpringBootMySQLApplication

Console Actuator

```

2023-09-30 14:13:55.128 INFO 21924 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...
]
2023-09-30 14:13:55.175 INFO 21924 --- [main] org.hibernate.Version : HHH000412: Hibernate Core {5.2.17.Final}
2023-09-30 14:13:55.176 INFO 21924 --- [main] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2023-09-30 14:13:55.204 INFO 21924 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2023-09-30 14:13:55.271 INFO 21924 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2023-09-30 14:13:55.539 INFO 21924 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-09-30 14:13:55.745 INFO 21924 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2023-09-30 14:13:55.952 INFO 21924 --- [main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.error.ErrorMvcConfig
2023-09-30 14:13:55.971 WARN 21924 --- [main] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure your mappings to disable this behavior
2023-09-30 14:13:55.992 INFO 21924 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[],methods=[GET]}" onto public com.alibaba.nacos.expo.core.entity.User com.alibaba.nacos.expo.core.controller.UserController.findAll()
2023-09-30 14:13:55.994 INFO 21924 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[error],produces=[text/html]}" onto public org.springframework.http.ResponseEntity<com.alibaba.nacos.expo.core.entity.User> com.alibaba.nacos.expo.core.controller.UserController.error()
2023-09-30 14:13:55.995 INFO 21924 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[error]}" onto public org.springframework.http.ResponseEntity<com.alibaba.nacos.expo.core.entity.User> com.alibaba.nacos.expo.core.controller.UserController.error()
2023-09-30 14:13:56.011 INFO 21924 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
2023-09-30 14:13:56.012 INFO 21924 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/] onto handler of type [class org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
2023-09-30 14:13:56.168 INFO 21924 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2023-09-30 14:13:56.161 INFO 21924 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Bean with name 'dataSource' has been autodetected for JMX exposure
2023-09-30 14:13:56.166 INFO 21924 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Located MBean 'dataSource': registering with JMX server as MBean [com.alibaba.nacos:service=DataSourceConfig, type=JMX]
2023-09-30 14:13:56.175 INFO 21924 --- [main] .LoggingNacosConfigMetadataEventListener : Nacos Config Metadata : dataId='mysql.properties', groupId='1', revision=1
2023-09-30 14:13:56.204 INFO 21924 --- [main] o.s.b.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-09-30 14:13:56.206 INFO 21924 --- [main] c.a.n.e.s.b.SpringBootMySQLApplication : Started SpringBootMySQLApplication in 2.711 seconds (JVM running for 3.001)

```

通过http请求查询后可以获取到对应的值



## 2.3.3 SpringBoot 服务发现

对应于Spring的升级版，这里和前几节差不多，我们首先需要修改的就是配置文件的服务器的地址

```
nacos.discovery.server-addr=192.168.150.101:8848
```

然后通过控制层去查看所有的服务信息，从而完成服务发现的步骤

```
@Controller  
@RequestMapping("discovery")  
public class DiscoveryController {  
  
    @NacosInjected  
    private NamingService namingService;  
  
    @RequestMapping(value = "/get", method = GET)  
    @ResponseBody  
    public List<Instance> get(@RequestParam String serviceName) thi  
        return namingService.getAllInstances(serviceName);  
    }  
}
```

通过NamingService查询所有的服务的实例，我们来进行启动查询。



这里依旧是获取不到信息，可能是因为版本的原因。

## 2.3.4 总结

总的来说，相较于Spring，SpringBoot有了很多的优化，既不用我们再去配置web容器，也用我们去书写繁杂的xml注解，方便了我们的开发工作，是的开发变得更加的方便，更加的易用起来了。

对于Spring和SpringBoot都有一点就是服务发现时发现不了，但是对应版本的client实现却能找到，应该是版本的影响。

```
[main] INFO com.alibaba.nacos.client.naming - init new ips(1) service: DEFAULT_GROUP@example -> [{"ip":"11.11.11.11","port":80,"weight":1.0,"healthy":true,"instanceId":null,"ipStr":"11.11.11.11"}]
[main] INFO com.alibaba.nacos.client.naming - current ips:(1) service: DEFAULT_GROUP@example -> [{"ip":"11.11.11.11","port":80,"weight":1.0,"healthy":true,"instanceId":null,"ipStr":"11.11.11.11"}]
[Instance{instanceId='null', ip='11.11.11.11', port=80, weight=1.0, healthy=true, enabled=true, ephemeral=true, clusterName='DEFAULT', serviceName='DEF...}]
[Thread-4] WARN com.alibaba.nacos.common.http.HttpClientBeanHolder - [HttpClientBeanHolder] Start destroying common HttpClient
[Thread-1] WARN com.alibaba.nacos.common.notify.NotifyCenter - [NotifyCenter] Start destroying Publisher
[Thread-1] WARN com.alibaba.nacos.common.notify.NotifyCenter - [NotifyCenter] Destruction of the end
[Thread-4] WARN com.alibaba.nacos.common.http.HttpClientBeanHolder - [HttpClientBeanHolder] Destruction of the end
```

- 2.4 Nacos集成SpringCloud
  - 2.4.1 Nacos配置SpringCloud
  - 2.4.2 Nacos配置多个数据源
  - 2.4.3 nacos服务发现

上一章我们学习了集成SpringBoot，其实和对应的Spring没有太多的变化，基本上还是配置和服务发现。

下面我们将来了解一下SpringCloud，对于单体应用可能不满足大型的互联网架构，因此越来越多的应用转换到了微服务，对应的有SpringCloud 和 SpringCloud Alibaba，我们主要针对的是第二个，因为nacos本身也是阿里生态里的组件，我们将探讨nacos在SpringCloud中的使用。

## 2.4 Nacos集成SpringCloud

首先便是依赖文件发生变化，变化为alibaba下面的组件依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
    <version>0.2.1.RELEASE</version>
</dependency>
```

其次便是配置文件的变化，之前是application.properties，现在是微服务，所以变成了bootstrap.properties，

### 2.4.1 Nacos配置SpringCloud

这个小项目是nacos-spring-cloud-config-example，我们需要配置一下几项。

```
# nacos的服务地址
spring.cloud.nacos.config.server-addr=192.168.150.101:8848

# 应用名
spring.application.name=example
# Config Type: properties(Default Value) \ yaml \ yml
# 配置文件的拓展名
spring.cloud.nacos.config.file-extension=properties
#spring.cloud.nacos.config.file-extension=yaml

# Map Nacos Config: example.properties

# Create the config Of nacos firstly?you can use one of the following ways
## Create Config By OpenAPI
### Create Config By OpenAPI
# curl -X POST 'http://127.0.0.1:8848/nacos/v1/cs/configs' -d 'dataId=example.properties&group=DEFAULT_GROUP&content=useLocalCache=true'
### Get Config By OpenAPI
# curl -X GET 'http://127.0.0.1:8848/nacos/v1/cs/configs?dataId=example.properties&group=DEFAULT_GROUP'

## Create Config By Console
### Login the console of Nacos: http://127.0.0.1:8848/nacos/index.html
### Data ID: example.properties
### Group: DEFAULT_GROUP
### Content: useLocalCache=true
```

Controller

```
@RestController
@RequestMapping("/config")
// 配置这个类中的属性是可以进行自动刷新的
@RefreshScope
public class ConfigController {

    @Value("${useLocalCache:false}")
    private boolean useLocalCache;

    /**
     * http://localhost:8080/config/get
     */
    @RequestMapping("/get")
    public boolean get() {
        return useLocalCache;
    }
}
```

然后我们按照提示创建配置(这里我们已经在前面几节创建过了)，然后我们启动观察，是否可以实现。

美观输出   
false

然后我们再去修改，再次进行查询



主要是spring.application.name=example和  
spring.cloud.nacos.config.file-extension=properties通过两个配置项  
从而锁定配置文件。

## 2. 4. 2 Nacos配置多个数据源

nacos-spring-cloud-config-multi-data-ids-example

从Spring那里我们大概了解到了，nacos可以同时注册多个数据，那么对于SpringCloud来说肯定也是需要这个需求的，那么我们接下来将进一步的学习怎么来使用。

bootstrap.properties文件

```
spring.application.name=multi-data-ids-example

spring.cloud.nacos.config.server-addr=192.168.150.101:8848

spring.cloud.nacos.config.ext-config[0].data-id=app.properties
spring.cloud.nacos.config.ext-config[0].group=multi-data-ids
spring.cloud.nacos.config.ext-config[0].refresh=true

spring.cloud.nacos.config.ext-config[1].data-id=datasource.properties
spring.cloud.nacos.config.ext-config[1].group=multi-data-ids

spring.cloud.nacos.config.ext-config[2].data-id=redis.properties
spring.cloud.nacos.config.ext-config[2].group=multi-data-ids
```

采用了如上的配置方法，之后我们也可以按照它这个来进行拓展书写，因为我们之前已经配置过了，所以在这里我们就直接进行启动。

启动成功，首先我们不开启redis缓存，来进行查询。



查询到了对应的值，此时的app.user.cache=false，是没有开启缓存的，所以缓存没数据，然后我们来开启缓存。

```
@Service
@RefreshScope
public class UserServiceImpl implements UserService {

    private static final Logger LOGGER = LoggerFactory.getLogger(UserService.class);

    private final UserRepository userRepository;

    private final RedisTemplate redisTemplate;

    @Value("${app.user.cache}")
    private boolean cache;

    @Autowired
    public UserServiceImpl(UserRepository userRepository, RedisTemplate redisTemplate) {
        this.userRepository = userRepository;
        this.redisTemplate = redisTemplate;
    }

    @Override
    public User findById(Long id) {
        LOGGER.info("cache: {}", cache);

        if (cache) {
            Object obj = redisTemplate.opsForValue().get(key(id));
            if (obj != null) {
                LOGGER.info("get user from cache, id: {}", id);
                return (User)obj;
            }
        }
    }
}
```

```

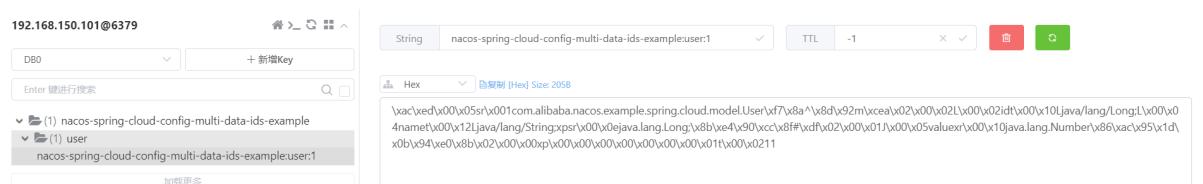
User user = userRepository.findById(id).orElse(null);
if (user != null) {
    if (cache) {
        LOGGER.info("set cache for user, id: {}", id);
        redisTemplate.opsForValue().set(key(id), user);
    }
}

return user;
}

private String key(Long id) {
    return String.format("nacos-spring-cloud-config-multi-data-%d", id);
}

```

可以看到缓存的值已经有了。



## 2. 4. 3 nacos服务发现

nacos-spring-cloud-discovery-example, 里面有两个小的子项目，分别扮演者生产者和消费者的例子。

我们首先来看consumer

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.alibaba.nacos</groupId>
            <artifactId>nacos-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-client</artifactId>
</dependency>
</dependencies>
```

配置文件application.properties

```
server.port=8080
spring.application.name=service-consumer
spring.cloud.nacos.discovery.server-addr=192.168.150.101:8848
```

然后就是比较主要的类了

```
@SpringBootApplication
// 开启服务发现的客户端
@EnableDiscoveryClient
public class NacosConsumerApplication {
    // 负载均衡
    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(NacosConsumerApplication.class, args);
    }
    // 测试的控制类
    @RestController
    public class TestController {

        private final RestTemplate restTemplate;

        @Autowired
        public TestController(RestTemplate restTemplate) {this.restTemplate = restTemplate; }

        @RequestMapping(value = "/echo/{str}", method = RequestMethod.GET)
        public String echo(@PathVariable String str) {
            return restTemplate.getForObject("http://service-provider:8080/echo/" + str, String.class);
        }
    }
}
```

## 生产者

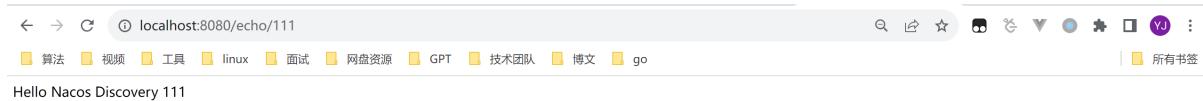
```
server.port=8070
spring.application.name=service-provider
spring.cloud.nacos.discovery.server-addr=192.168.150.101:8848
```

```
@SpringBootApplication
// 开启服务发现客户端
@EnableDiscoveryClient
public class NacosProviderApplication {

    public static void main(String[] args) {
        SpringApplication.run(NacosProviderApplication.class, args);
    }
    // 调用的服务
    @RestController
    class EchoController {
        @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            return "Hello Nacos Discovery " + string;
        }
    }
}
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.alibaba.nacos</groupId>
            <artifactId>nacos-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-client</artifactId>
</dependency>
```

然后我们来启动生产者和消费者，然后看看服务发现是否能够生效。



确实调用了对应的服务，并把结果给返回了，然后我们来看一下nacos的控制台。

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
service-provider	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
order-service1	MQ_topic	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
service-consumer	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

在这里有一个小的特性便是动态的dns，这里究竟是怎么实现的呢，我们之后的源码解析时会提到，这里我们只是简单的知道有这个很高级的功能即可。

我们对于SpringCloud的使用就基本上告一段落了。

- 2.5 Nacos集成dubbo

## 2.5 Nacos集成dubbo

通过上面的学习我们大概了解了nacos的两大功能，然后我们就要看一下nacos和dubbo的结合使用，这个项目是nacos-dubbo-example。众所周知，dubbo是远程功能调用，那么这就会涉及三个概念，调用方，提供方和接口

首先是调用方，也被称为消费者。

```
// 开启dubbo
@EnableDubbo
// 配置文件的路径
@PropertySource(value = "classpath:/consumer-config.properties")
public class DemoServiceConsumerBootstrap {

    // 远程服务的注入
    @DubboReference(version = "${demo.service.version}")
    private DemoService demoService;

    // 等待注入后的调用
    @PostConstruct
    public void init() {
        for (int i = 0; i < 10; i++) {
            System.out.println(demoService.sayName("Nacos"));
        }
    }

    // 主程序, 用于创建上下文
    public static void main(String[] args) throws IOException {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
        // 启动类
        context.register(DemoServiceConsumerBootstrap.class);
        // 刷新上下文容器
        context.refresh();
        context.close();
    }
}
```

对应的配置文件

```
## Dubbo Application info
dubbo.application.name = dubbo-consumer-demo

## Nacos registry address
dubbo.registry.address = nacos://192.168.150.101:8848
#dubbo.registry.address = nacos://127.0.0.1:8848?namespace=5cbb70a5-xxx-xxx-xxx-d43479ae6
#dubbo.registry.parameters.namespace=5cbb70a5-xxx-xxx-xxx-d43479ae6

# @Reference version
demo.service.version= 1.0.0

dubbo.application.qosEnable=false
```

## 生产者

```
// 开启dubbo
@EnableDubbo(scanBasePackages = "com.alibaba.nacos.example.dubbo.se
// 配置文件
@PropertySource(value = "classpath:/provider-config.properties")
public class DemoServiceProviderBootstrap {
    // 主程序
    public static void main(String[] args) throws IOException {
        AnnotationConfigApplicationContext context = new AnnotationC
        context.register(DemoServiceProviderBootstrap.class);
        context.refresh();
        System.out.println("DemoService provider is starting...");
```

然后就是接口

```
public interface DemoService {  
    String sayName(String name);  
}
```

基本上是这样，调用方只拥有接口，而提供方拥有接口和具体的实现，两者通过dubbo进行调用，而nacos则是起一个注册中心的角色，调用方需要根据注册中心的内容去寻找具体的调用实现，dubbo的默认的注册中心就是nacos。

然后我们来进行测试，确实进行调用。

```
[2021-07-23 09:24:51,051 main INFO o.s.i.d.u.DubboConsumerApp$DubboConsumerApp$ConsumerConsumer: [Dubbo] Dubbo API  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos  
Service [name :demoService , port : 20880] sayName("Nacos") : Hello,Nacos
```

而nacos上也有了提供方的信息

consumers:com.alibaba.nacos.example.dubbo.service.DemoService:1.0.0:	DEFAULT_GROUP	0	0	0	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
order-service1	MQ_topic	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
providers:com.alibaba.nacos.example.dubbo.service.DemoService:1.0.0:	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
dubbo-provider-demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

- 2.9 技术探讨OR学习总结
  - 2.9.1 关于日志的问题
  - 2.9.2 关于Mysql的报错
  - 2.9.3 关于开启权限认证

## 2.9 技术探讨OR学习总结

前面我们大概学习了nacos与各种框架的结合使用，大多数是参照官网的，这里也是我想告诉大家的一点，学习最快的方式就是参照官网，然后先会使  
用，然后了解实现，再到深入源码。

### 2.9.1 关于日志的问题

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
```

### 2.9.2 关于Mysql的报错

```
MySQLNonTransientConnectionException: Could not create connection
to database server.
```

mysql驱动5和8不兼容，换一个驱动即可。

### 2.9.3 关于开启权限认证

因为大多数的nacos是线上服务（暴露在内网的线上服务），那么最好还是  
开启权限认证比较好。那么开启的方式官网也已经告诉我们了。

权限认证文档清晰明了，如果开启了验证那么要在使用服务前先进行权限的  
校验，我们来进行docker的开启鉴权的测试。

为了系统安全起见，新版本（Nacos 2.2.1）删除了以下环境变量的默认值，请在启动时自行添加，否则启动时会报错。

1. NACOS\_AUTH\_IDENTITY\_KEY
2. NACOS\_AUTH\_IDENTITY\_VALUE
3. NACOS\_AUTH\_TOKEN

首先的话我们需要移除docker容器

```
# 停止容器
docker stop b59ea55b8973
# 移除容器
docker rm b59ea55b8973
# 启动新的容器
docker run \
--name nacos-quick \
-e MODE=standalone \
-e NACOS_AUTH_ENABLE=true \
-e NACOS_AUTH_TOKEN=VGhpc0lzTXlDdXN0b21TZWNyZXRLZXkwMTIzNDU2Nzg \
-e NACOS_AUTH_IDENTITY_KEY=nacos \
-e NACOS_AUTH_IDENTITY_VALUE=nacos \
-p 8848:8848 \
-p 9848:9848 \
-d nacos/nacos-server:v2.2.3
```

这样就配置了认证之后的操作都要先输入账号密码才能访问， 我们以dubbo为示例来继续演示。

未添加权限

```
[30/09/23 03:43:25 GMT+08:00] DubboShutdownHook INFO nacos.NacosRegistry: [DUBBO] Unregister: dubbo://10.102.46.60:20880/com.alibaba.nacos.example.dubbo.service.DemoService?anyhost=true&ap
[30/09/23 03:43:25 GMT+08:00] DubboShutdownHook WARN nacos.NacosRegistry: [DUBBO] Failed to unregister url dubbo://10.102.46.60:20880/com.alibaba.nacos.example.dubbo.service.DemoService?anyhost=true&ap
java.lang.IllegalStateException Create breakpoint : Failed to unregister dubbo://10.102.46.60:20880/com.alibaba.nacos.example.dubbo.service.DemoService?anyhost=true&ap
    at org.apache.dubbo.registry.support.FailbackRegistry.unregister(FailbackRegistry.java:267)
    at org.apache.dubbo.registry.support.AbstractRegistry.destroy(AbstractRegistry.java:489)
    at org.apache.dubbo.registry.support.FailbackRegistry.destroy(FailbackRegistry.java:408)
    at org.apache.dubbo.registry.support.RegistryManager.destroyAll(RegistryManager.java:105)
    at org.apache.dubbo.config.deploy.DefaultApplicationDeployer.destroyRegistries(DefaultApplicationDeployer.java:1101)
    at org.apache.dubbo.config.deploy.DefaultApplicationDeployer.postDestroy(DefaultApplicationDeployer.java:885)
    at org.apache.dubbo.rpc.model.ApplicationModel.onDestroy(ApplicationModel.java:269)
    at org.apache.dubbo.rpc.model.ScopeModel.destroy(ScopeModel.java:108)
    at org.apache.dubbo.config.DubboShutdownHook.doDestroy(DubboShutdownHook.java:67)
    at org.apache.dubbo.config.DubboShutdownHook.run(DubboShutdownHook.java:62)
Caused by: org.apache.dubbo.rpc.RpcException Create breakpoint : Failed to unregister dubbo://10.102.46.60:20880/com.alibaba.nacos.example.dubbo.service.DemoService?anyhost=true&ap
    at org.apache.dubbo.registry.nacos.NacosRegistry.doUnregister(NacosRegistry.java:194)
    at org.apache.dubbo.registry.support.FailbackRegistry.unregister(FailbackRegistry.java:254)
    ... 9 more
Caused by: ErrCode:500, ErrMsg:Request nacos server failed:
    at com.alibaba.nacos.client.naming.remote.nnc.NamingNncClientProxy.requestToServer(NamingNncClientProxy.java:270)
```

会在注册的时候报错，这个时候我们就需要进行添加账号密码

```
nacos://192.168.150.101:8848?username=nacos&password=nacos
```

这样我们就会鉴权成功。

- 3.1 核心特性
  - 3.1.1 DNS服务
  - 3.1.2 服务发现
  - 3.1.3 权重管理
  - 3.1.4 打标管理
  - 3.1.5 优雅上下线
  - 3.1.6 在线编辑
  - 3.1.7 历史版本
  - 3.1.8 一键回滚
  - 3.1.9 灰度发布
  - 3.1.10 推送轨迹

之前我们已经对于nacos的使用已经有了一定的了解，接下来我们将对于一些核心的特性来进行学习，首先便是它的动态DNS实现

## 3.1 核心特性

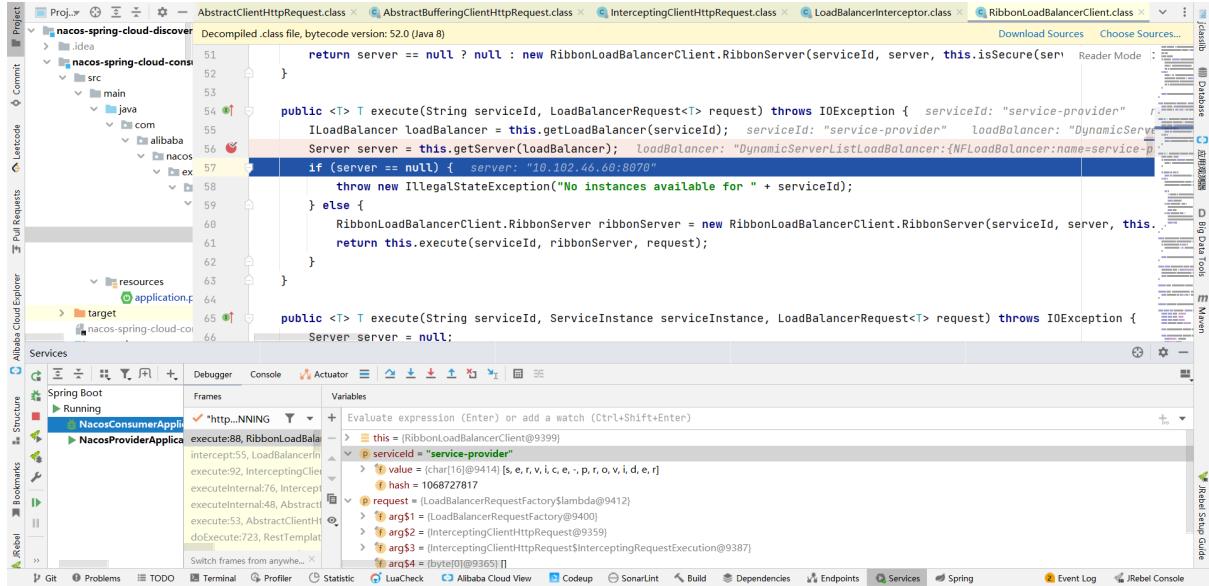
### 3.1.1 DNS服务

我们以SpringCloud的请求为例，来看一下这个DNS是如何实现的，首先我们来看一下这段代码

```
@RequestMapping(value = "/echo/{str}", method = RequestMethod.GET)
public String echo(@PathVariable String str) {
    return restTemplate.getForObject("http://service-provider/echo/" + str, String.class);
}
```

理论上来说，这段代码使用了service-provider这个域名，应该去请求这个域名对应的服务器，那么下来是怎么做的呢，通过debug我们发现，它首先将域名当作服务名取出，然后通过ribbon的负载均衡取出了这个服务名对应

的服务器地址。



那么此时我有个猜想就是，如果没有了负载均衡，他还会具有动态路由的特性吗？我们将负载均衡去掉试一下。

```
java.net.UnknownHostException Create breakpoint : service-provider
    at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:196) ~[na:1.8.0_291]
    at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:162) ~[na:1.8.0_291]
    at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:394) ~[na:1.8.0_291]
    at java.net.Socket.connect(Socket.java:606) ~[na:1.8.0_291]
    at java.net.Socket.connect(Socket.java:555) ~[na:1.8.0_291]
    at sun.net.NetworkClient.doConnect(NetworkClient.java:180) ~[na:1.8.0_291]
    at sun.net.www.http.HttpClient.openServer(HttpClient.java:463) ~[na:1.8.0_291]
    at sun.net.www.http.HttpClient.openServer(HttpClient.java:558) ~[na:1.8.0_291]
    at sun.net.www.http.HttpClient.<init>(HttpClient.java:242) ~[na:1.8.0_291]
    at sun.net.www.http.HttpClient.New(HttpClient.java:339) ~[na:1.8.0_291]
    at sun.net.www.http.HttpClient.New(HttpClient.java:357) ~[na:1.8.0_291]
    at sun.net.protocol.http.HttpURLConnection.getNewHttpClient(HttpURLConnection.java:1226) ~[na:1.8.0_291]
    at sun.net.protocol.http.HttpURLConnection.plainConnect0(HttpURLConnection.java:1162) ~[na:1.8.0_291]
    at sun.net.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:1056) ~[na:1.8.0_291]
    at sun.net.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:990) ~[na:1.8.0_291]
    at org.springframework.http.client.SimpleBufferingClientHttpRequest.executeInternal(SimpleBufferingClientHttpRequest.java:76) ~[spring-w
    at org.springframework.http.client.AbstractBufferingClientHttpRequest.executeInternal(AbstractBufferingClientHttpRequest.java:48) ~[spri
```

出现了报错，未知域名的错误，这里大概就可以猜出来了，它通过负载均衡在rest请求的时候将对应的服务名更改为对应的ip地址进而再请求的时候去请求对应的服务器。

接下来我们来继续看一下，那么nacos是什么时候和负载均衡关联到一起的呢？

我们通过debug，我们发现了最后是LoadBlancer把服务Id封装成了ILoadBalancer，然后通过Spring的对象工厂通过名称和类型的方式找到已经装配的对象

```
public <T> T getInstance(String name, Class<T> type) {  
    AnnotationConfigApplicationContext context = this.getConte  
    return BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
```

}

当取到对应的ILoadBalancer对象后，通过它得到了对应的Server，然后Server中则是包含了元数据信息，如果没有找到对应的服务的话，那么就按照原始的域名进行访问。

```
protected Server getServer(ILoadBalancer loadBalancer) {  
    return loadBalancer == null ? null : loadBalancer.chooseSer  
}
```

在这里我们得到一个结论，服务并不是在使用的时候才拉取的nacos服务，而是在装配的时候已经封装了对应的nacos的元数据对象为ILoadBalancer。

现在呢就找到了最主要的类自动装配的类RibbonNacosAutoConfiguration

```
@Configuration  
@EnableConfigurationProperties  
@ConditionalOnBean({SpringClientFactory.class})  
@ConditionalOnRibbonNacos  
@ConditionalOnNacosDiscoveryEnabled  
@AutoConfigureAfter({RibbonAutoConfiguration.class})  
@RibbonClients(  
    defaultConfiguration = {NacosRibbonClientConfiguration.class}  
)  
public class RibbonNacosAutoConfiguration {  
    public RibbonNacosAutoConfiguration() {  
    }  
}
```

提到了自动装配就不得不提一下SpringBoot的自动装配，在springboot的自动装配过程中，最终会加载 `META-INF/spring.factories` 文件，而加载的过程是由 `SpringFactoriesLoader` 加载的。从CLASSPATH下的每个Jar包中搜寻所有 `META-INF/spring.factories` 配置文件，然后将解析properties文件，找到指定名称的配置后返回。需要注意的是，其实这里不仅仅是会去 ClassPath路径下查找，会扫描所有路径下的Jar包，只不过这个文件只会在 Classpath下的jar包中。

我们来看一下nacos的自动装配的factories文件

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
    org.springframework.cloud.alibaba.nacos.NacosDiscoveryAutoConfigu  
    org.springframework.cloud.alibaba.nacos.ribbon.RibbonNacosAutoCor  
    org.springframework.cloud.alibaba.nacos.endpoint.NacosDiscoveryEr  
    org.springframework.cloud.alibaba.nacos.discovery.NacosDiscoveryC
```

总共是四个NacosDiscoveryAutoConfiguration、  
RibbonNacosAutoConfiguration、  
NacosDiscoveryEndpointAutoConfiguration、  
NacosDiscoveryClientAutoConfiguration

我们从第一个看起NacosDiscoveryAutoConfiguration

```
// 声明为配置类
@Configuration
// 使得使用@ConfigurationProperties的类生效
@EnableConfigurationProperties
// nacos的服务发现是开启的
@ConditionalOnNacosDiscoveryEnabled
// 是否开启了这个属性，如果没有配置就是默认开启的
@ConditionalOnProperty(
    value = {"spring.cloud.service-registry.auto-registration.enabled"}, // 配置属性名
    matchIfMissing = true // 如果没有配置，则认为是true
)
// 在类之后进行自动配置，为什么要在这两个类之后进行装配呢， 因为要按照使用顺序
@AutoConfigureAfter({AutoServiceRegistrationConfiguration.class, AutoServicePropertiesConfiguration.class})
public class NacosDiscoveryAutoConfiguration {
    public NacosDiscoveryAutoConfiguration() {
    }

    // NacosServiceRegistry 服务的Bean，用于注册NacosRegistration。
    @Bean
    public NacosServiceRegistry nacosServiceRegistry(NacosDiscoveryProperties nacosDiscoveryProperties) {
        return new NacosServiceRegistry(nacosDiscoveryProperties);
    }

    // 封装的NacosRegistration
    @Bean
    @ConditionalOnBean({AutoServiceRegistrationProperties.class})
    public NacosRegistration nacosRegistration(NacosDiscoveryProperties nacosDiscoveryProperties) {
        return new NacosRegistration(nacosDiscoveryProperties, context);
    }

    // 自动装配的NacosAutoServiceRegistration
    @Bean
    public NacosAutoServiceRegistration nacosAutoServiceRegistration(NacosDiscoveryProperties nacosDiscoveryProperties) {
        return new NacosAutoServiceRegistration(nacosDiscoveryProperties, context);
    }
}
```

```
@ConditionalOnBean({AutoServiceRegistrationProperties.class})
public NacosAutoServiceRegistration nacosAutoServiceRegistration() {
    return new NacosAutoServiceRegistration(registry, autoServiceProperties);
}
```

RibbonNacosAutoConfiguration 用于集成Ribbon实现负载均衡

```
// 声明为配置类
@Configuration
// @EnableConfigurationProperties注解应用到你的@Configuration时， 任何
@EnableConfigurationProperties
// 有这个类
@ConditionalOnBean({SpringClientFactory.class})
//
@ConditionalOnRibbonNacos
// 是否开启服务发现
@ConditionalOnNacosDiscoveryEnabled
// 在Ribbon配置以后再进行自动装配
@AutoConfigureAfter({RibbonAutoConfiguration.class})
// 设置客户端的配置类
@RibbonClients(
    defaultConfiguration = {NacosRibbonClientConfiguration.class}
)
public class RibbonNacosAutoConfiguration {
    public RibbonNacosAutoConfiguration() {
    }
}
```

```
    @Bean  
    @ConditionalOnMissingBean  
    public ServerList<?> ribbonServerListIClientConfig, NacosDiscoveryProperties nacosDiscoveryProperties) {  
        NacosServerList serverList = new NacosServerList(nacosDiscoveryProperties);  
        serverList.initWithNacosConfig(config);  
        return serverList;  
    }
```

在ribbon的配置类中将nacos的服务进行初始化到ServerList。

```
@Configuration  
@ConditionalOnClass({HttpRequest.class})  
@RibbonAutoConfiguration.ConditionalOnRibbonRestClient  
protected static class RibbonClientHttpRequestFactoryConfiguration {  
    @Autowired  
    private SpringClientFactory springClientFactory;  
  
    protected RibbonClientHttpRequestFactoryConfiguration() {  
    }  
  
    @Bean  
    public RestTemplateCustomizer restTemplateCustomizer(final  
        return (restTemplate) -> {  
            restTemplate.setRequestFactory(ribbonClientHttpRequestFactory());  
        };  
    }  
  
    @Bean  
    public RibbonClientHttpRequestFactory ribbonClientHttpRequestFactory() {  
        return new RibbonClientHttpRequestFactory(this.springClientFactory);  
    }  
}
```

在RibbonClientHttpRequestFactoryConfiguration这个类中可以看到，ribbon的http请求工厂使用了SpringClientFactory，然后所有的rest请求就交予了它，所以便会有之后的请求拦截，然后实现DNS将域名兑换为相应的IP地址，然后在通过重组的ip地址进行请求，并返回响应。

NacosDiscoveryEndpointAutoConfiguration，用于 nacos 发现、获取 nacos 属性和订阅服务的端点

```
@Configuration  
 @ConditionalOnClass({Endpoint.class})  
 @ConditionalOnNacosDiscoveryEnabled  
 public class NacosDiscoveryEndpointAutoConfiguration {  
     public NacosDiscoveryEndpointAutoConfiguration() {  
     }  
  
     @Bean  
     @ConditionalOnMissingBean  
     @ConditionalOnEnabledEndpoint  
     public NacosDiscoveryEndpoint nacosDiscoveryEndpoint(NacosDiscoveryProperties nacosDiscoveryProperties) {  
         return new NacosDiscoveryEndpoint(nacosDiscoveryProperties);  
     }  
 }
```

NacosDiscoveryClientAutoConfiguration 服务发现客户端自动配置

```
@Configuration
// 开了服务发现
@ConditionalOnNacosDiscoveryEnabled
// 在这两个类之前进行装配
@AutoConfigureBefore({SimpleDiscoveryClientAutoConfiguration.class})
public class NacosDiscoveryClientAutoConfiguration {
    public NacosDiscoveryClientAutoConfiguration() {
    }

    // 这个类用于描述当前注册服务的元数据信息
    @Bean
    @ConditionalOnMissingBean
    public NacosDiscoveryProperties nacosProperties() {
        return new NacosDiscoveryProperties();
    }

    // 服务发现客户端，用于获取nacos的服务的信息
    @Bean
    public DiscoveryClient nacosDiscoveryClient(NacosDiscoveryProperties discoveryProperties) {
        return new NacosDiscoveryClient(discoveryProperties);
    }

    // 注册配置监听器到Spring
    @Bean
    // 没有Nacoswatch这个类
    @ConditionalOnMissingBean
    // 是否开启
    @ConditionalOnProperty(
        value = {"spring.cloud.nacos.discovery.watch.enabled"}, 
        matchIfMissing = true
    )
}
```

```
public NacosWatch nacosWatch(NacosDiscoveryProperties nacosDiscoveryProperties) {
    return new NacosWatch(nacosDiscoveryProperties);
}
```

那么动态DNS就看到这里。

### 3.1.2 服务发现

### 3.1.3 权重管理

### 3.1.4 打标管理

### 3.1.5 优雅上下线

### 3.1.6 在线编辑

### 3.1.7 历史版本

### 3.1.8 一键回滚

### 3.1.9 灰度发布

### 3.1.10 推送轨迹