spring cloud

# 准备

Eclipse安装spring boot插件：spring tool suite(sts)

# 官方Demo

1、参考介绍文档http://dockone.io/article/510

2、github下载源码https://github.com/kbastani/spring-cloud-microservice-example

3、下载docker tool https://get.daocloud.io/

4、导入eclipse，安装Spring Tool Suite (STS) for Eclipse 插件

5、window必须打开bois的虚拟化功能 vt-x

6、使用Docker Quickstart Terminal 进入到项目根目录，然后命令mvn clean install构建项目；

7、构建成功后，进入docker项目，后台运行 docker-compose up –d 启动服务

8、docker-machine ssh登录到镜像中，通过ifconfig 命令查看镜像ip，通常是192.168.99.xxx

9、在cmd下设置环境变量：set DOCKER\_HOST=tcp://镜像ip:2376 例如：

set DOCKER\_HOST=tcp://192.168.99.100:8761

10、浏览器输入：http://镜像ip:8761/ 例如http://192.168.99.100:8761/

11、可以直接访问 http://192.168.99.100:10000

tips:

1、后台运行 docker-compose up -d

2、登录到某个镜像：docker-machine ssh 镜像名

3、本地http连接镜像中的系统

首先通过echo $(echo \"$(echo $DOCKER\_HOST)\"|

\sed 's/tcp:\/\//http:\/\//g'|

\sed 's/[0-9]\{4,\}/8761/g'|

\sed 's/\"//g')

得到连接地址，类似http://192.168.99.101:8761/

然后浏览器中访问（我遇到的情况是开始死活访问不了，第二天就可以了。。。不知道什么原因——应该是启动太慢了，将近10分钟）

4、要用官方的maven仓库，不要用杭研的;

5、docker相关的内容在宿主机的 /var/lib/docker 目录下（sudo root账号查看和操作）

问题：

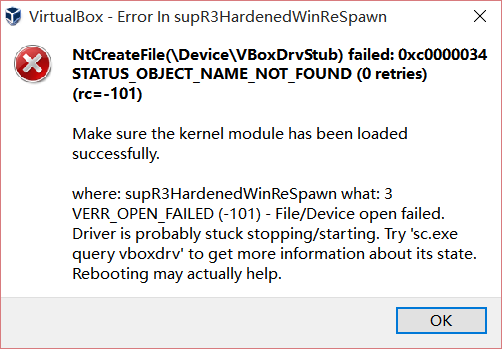
1、 mvn clean install -e 进行编译构建时，抛异常

无效的目标版本 1.8

解决：保持maven/系统默认/ spring-cloud-microservice-example 的pom文件 三者中指定的jdk版本一致，并1.7及以上

2、windows上需要开启bois vt-v模式

3、



解决方法：

CMD运行sc query vboxdrv发现

C:\windows\system32>sc query vboxdrv

[SC] EnumQueryServicesStatus:OpenService 失败 1060:

指定的服务未安装。

进入%VBOX\_MSI\_INSTALL\_PATH%\drivers\vboxdrv目录

右击VBoxDrv.inf 选择安装

CMD继续运行sc query vboxdrv发现

C:\windows\system32>sc query vboxdrv

SERVICE\_NAME: vboxdrv

TYPE : 1 KERNEL\_DRIVER

STATE : 1 STOPPED

WIN32\_EXIT\_CODE : 1077 (0x435)

SERVICE\_EXIT\_CODE : 0 (0x0)

CHECKPOINT : 0x0

WAIT\_HINT : 0x0

继续CMD输入sc start vboxdrv

C:\windows\system32>sc start vboxdrv

SERVICE\_NAME: vboxdrv

TYPE : 1 KERNEL\_DRIVER

STATE : 4 RUNNING

(STOPPABLE, NOT\_PAUSABLE, IGNORES\_SHUTDOWN)

WIN32\_EXIT\_CODE : 0 (0x0)

SERVICE\_EXIT\_CODE : 0 (0x0)

CHECKPOINT : 0x0

WAIT\_HINT : 0x0

PID : 0

FLAGS :

再重新启动虚拟机看看。是不是已经可以进去了。

4、

Failed to execute goal com.spotify:docker-maven-plugin:0.3.258:build (default) on project users-microservice: Exception caught: java.util.concurrent.ExecutionException: com.spotify.docker.client.shaded.javax.ws.rs.ProcessingException: org.apache.http.conn.HttpHostConnectException: Connect to localhost:2375 [localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed: Connection refused: connect -> [Help 1]

原因：docker-maven-plugin构建时默认会连接本地localhost:2375 ，但是我们的docker在虚拟机上，需要想办法把本地和虚拟机连接打通。

解决（好像也没用）

Yes, the plugin should default to the default local socket if DOCKER\_HOST is not set - this is an upstream bug

you just need to   
export DOCKER\_HOST=unix:///private/var/tmp/docker.sock

即用Docker Quickstart Terminal 输入上面命令，然后再试（同样在终端中使用mvn命令进行编译构建），不行就重启电脑

# 常用命令

docker-machine start default 启动镜像

docker-machine ssh 登录到镜像

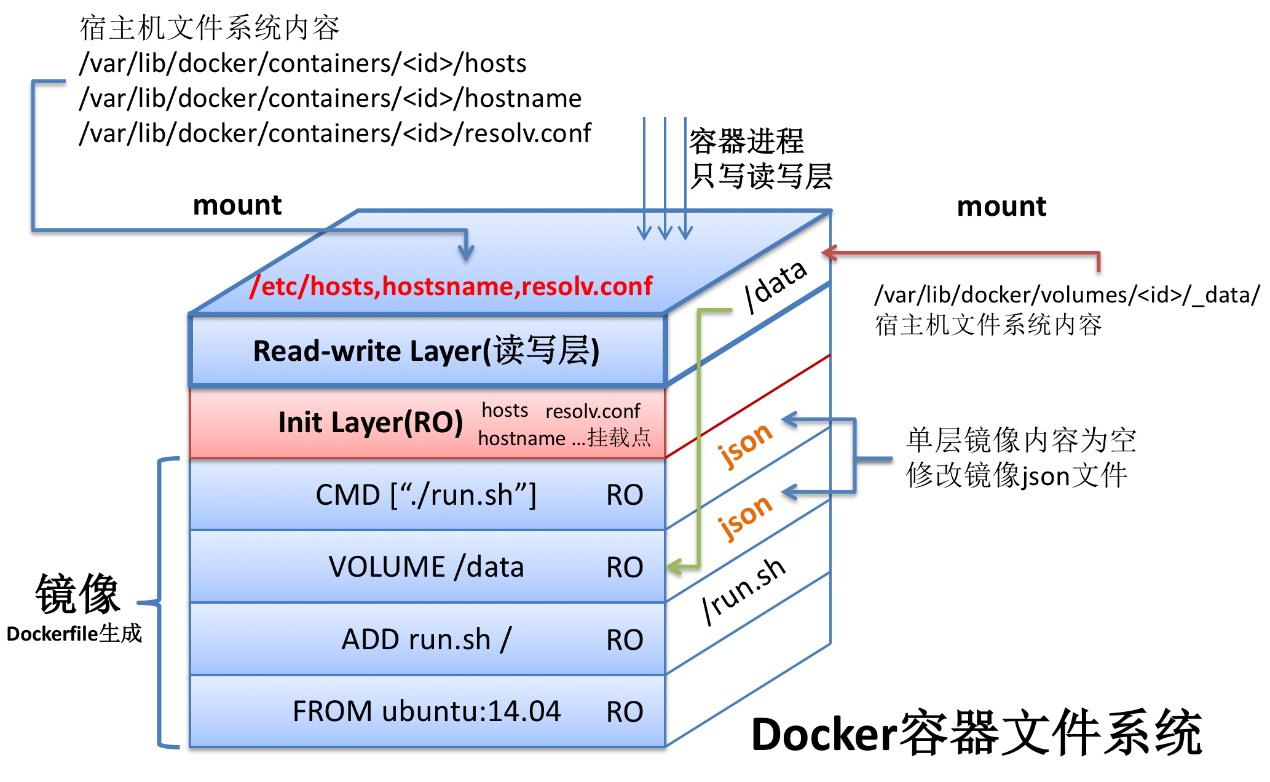
docker宿主机（Windows下即虚拟机）切换root账号: sudo -i

我的docker账号：lizhaosheng

# Kbastani解析

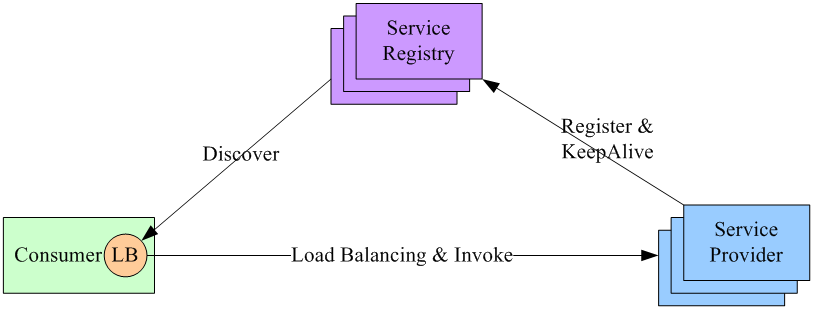
# 理解

## Docker



## Eureka

服务注册，包括Eureka Server/ Eureka Client，client又分两种：服务提供端和服务消费端（注意，在微服务语义下，服务提供和服务消费其实都是内部子系统，也就是微服务解决的是子系统之间的问题，并不包括外部用户和客户端，那是网关进来的请求了）。所以大致是下面这样的：



* Service Registry即Eureka Server，负责服务的注册发现；
* Service Provider 服务提供端，比如某个子系统的Controller；
* Consumer 服务消费端，即服务调用方，比如某个子系统的一个功能需要调用另一个子系统提供的服务；
* LB 负载均衡；

使用@EnableEurekaServer来声明当前系统为eureka服务注册发现系统，

使用@EnableDiscoveryClient来声明当前系统为eureka的客户端系统。

# 部署war包配置

<https://qbgbook.gitbooks.io/spring-boot-reference-guide-zh/content/IX.%20%E2%80%98How-to%E2%80%99%20guides/74.1.%20Create%20a%20deployable%20war%20file.html>

### 创建一个可部署的war文件

产生一个可部署war包的第一步是提供一个SpringBootServletInitializer子类，并覆盖它的configure方法。这充分利用了Spring框架对Servlet 3.0的支持，并允许你在应用通过servlet容器启动时配置它。通常，你只需把应用的主类改为继承SpringBootServletInitializer即可：

@SpringBootApplication

public class Application extends SpringBootServletInitializer {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(Application.class);

}

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

SpringApplication.run(Application.class, args);

}

}

下一步是更新你的构建配置，这样你的项目将产生一个war包而不是jar包。如果你使用Maven，并使用spring-boot-starter-parent（为了配置Maven的war插件），所有你需要做的就是更改pom.xml的packaging为war：

<packaging>war</packaging>

如果你使用Gradle，你需要修改build.gradle来将war插件应用到项目上：

apply plugin: 'war'

该过程最后的一步是确保内嵌的servlet容器不能干扰war包将部署的servlet容器。为了达到这个目的，你需要将内嵌容器的依赖标记为provided。

如果使用Maven：

<dependencies>

<!-- … -->

<dependency>

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

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

<scope>provided</scope>

</dependency>

<!-- … -->

</dependencies>

如果使用Gradle：

dependencies {

// …

providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

// …

}

如果你使用[Spring Boot构建工具](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#build-tool-plugins)，将内嵌容器依赖标记为provided将产生一个可执行war包，在lib-provided目录有该war包的provided依赖。这意味着，除了部署到servlet容器，你还可以通过使用命令行java -jar命令来运行应用。

**注**：查看Spring Boot基于以上配置的一个[Maven示例应用](http://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-traditional/pom.xml)。

# My Demo

**组件**

Eureka – 服务注册（做集群）

Zuul – 网关

Ribbon – 服务发现

Feign – 服务发现

Hystrix – 容错

Turbine – 容错（集群）

Rabbitmq – 消息队列

Redis – 缓存

Mybatis – dao

Mysql – db

Spring session – 分布式session方案

Springfox – api自动文档化

Docker – 部署

由于docker水比较深，暂时不涉及docker相关

**一些重要的可注入对象**

**RestTemplate –** ribbon服务调用对象，根据url或者服务id调用远程服务

@Autowired

**private** RestTemplate restTemplate;

需要在项目的spring boot启动类中加入：

@SpringBootApplication

@EnableDiscoveryClient

@RibbonClient(name = "demo-service", configuration = DemoRibbonConfiguration.**class**)

**public** **class** Application {

@Bean

@LoadBalanced

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Application.**class**, args);

}

}

注意：@RibbonClient(name = "demo-service", configuration = DemoRibbonConfiguration.**class**)中的name指的是被调用的服务名，也就是当前配置仅对调用名为demo-service的服务起作用！！除非DemoRibbonConfiguration能被自动扫描（@SpringBootApplication

）到上下文，则当前应用的所有客户端都共享该配置。

**DiscoveryClient** **–** 服务注册客户端对象，可以获取服务列表

@Autowired

DiscoveryClient discoveryClient;

涉及项目：

Puppet——父级项目，下面都是Puppet的子项目，该项目提供统一的构建方式，也是parent项目

Config——统一配置

Discovery——服务注册及发现

Gateway——网关

Demo——demo

**Spring 注解扫描路径**

Spring boot 默认扫描路径为@SpringBootApplication 注解所在目录及其子目录，[可以通过@SpringBootApplication(scanBasePackages={"com.netease.kolibri","com.netease.numen"})](mailto:可以通过@SpringBootApplication(scanBasePackages=%7b%22com.netease.kolibri%22,%22com.netease.numen%22%7d)) 这样的方式来改变扫描包路径

两种方式连接服务（包括配置中心）

1、通过服务id

spring:

cloud:

config:

discovery:

service-id: config

2、通过固定url

spring:

cloud:

config:

uri: http://localhost:8888

注意，通过服务id方式需要加入依赖

<dependency>

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

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

</dependency>

**随机端口**

server:

port: ${PORT:${SERVER\_PORT:0}}

# 首先若存在环境变量（命令行或配置文件）存在PORT则使用PORT，若不存在继续检查是否存在SERVER\_PORT变量，存在则使用，否则随机（0）分配一。

**随机值**

${random.value}

**默认eureka实例id**

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}

如myhost:myappname:8080

可以通过下面属性改变

*application.yml*

eureka:

instance:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

若两个实例使用了相同的instanceId，则后面实例会覆盖前面

**Discovery 集群**

<https://segmentfault.com/a/1190000006149891>

defaultZone为下面格式：<http://hostname:port/eureka/>

若需要进行身份验证，则<http://user:password@localhost:8761/eureka>

application.yml

spring:

profiles: discovery1

server:

port: ${PORT:8761}

eureka:

instance:

hostname: discovery1

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

defaultZone: http://discovery2:8762/eureka/,http://discovery3:8763/eureka/

---

spring:

profiles: discovery2

server:

port: ${PORT:8762}

eureka:

instance:

hostname: discovery2

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

defaultZone: http://discovery1:8761/eureka/,http://discovery3:8763/eureka/

---

spring:

profiles: discovery3

server:

port: ${PORT:8763}

eureka:

instance:

hostname: discovery3

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

defaultZone: http://discovery1:8761/eureka/,http://discovery2:8762/eureka/

boot

bootstrap.yml

spring:

application:

name: discovery

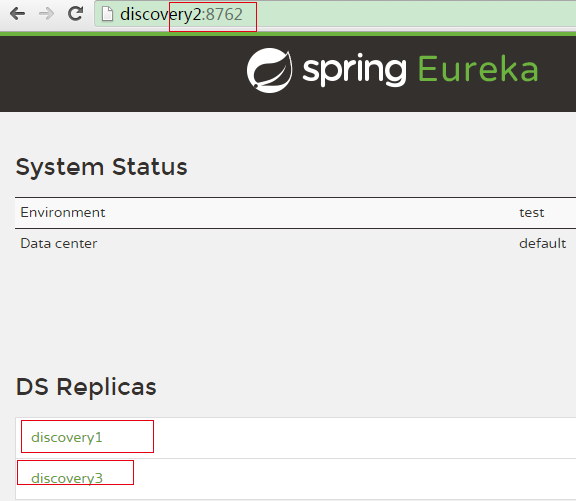
profiles:

active: discovery1 #default active profile

# spring.cloud.config.discovery.enabled: true

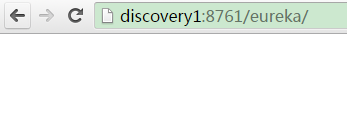
存在问题：

从集群中的某个实例界面点击链接跳转到另一个实例界面url不正确，如下



点击discovery1后跳转地址是<http://discovery1:8761/eureka/>

实际应该是<http://discovery1:8761/>



若config的客户端需要通过eureka使用serviceId来访问config server，需要设置

spring:

cloud:

config:

discovery:

service-id: config

enabled: **true # don’t forget it**

不要忘记设置enabled: true

**difference of application.yml and bootstrap.yml**

**bootstrap.yml is loaded before application.yml.**

It is typically used for the following:

* when using Spring Cloud Config Server, you should specify spring.application.name and spring.cloud.config.server.git.uri inside bootstrap.yml
* some encryption/decryption information

Technically, bootstrap.yml is loaded by a parent Spring ApplicationContext. That parent ApplicationContext is loaded before the one that uses application.yml.

另外，spring.cloud.config相关也需要放到bootstrap.yml中

**远程application.yml 和 本地application.yml**

远程从config server中获取的配置信息或覆盖本地配置信息

**服务实例设置**

eureka:

instance:

hostname: config

instance-id: ${spring.cloud.client.hostname}:${spring.application.name}:${server.port}-${random.value}

preferIpAddress: **true**

特别要注意preferIpAddress属性，默认false，若为true，则请求服务时，从eureka中获取到的host为ip形式，否则为hostname的值（默认主机名，在没有设置host的网络中容易请求失败）。强烈建议所有discoveryclient都设置preferIpAddress: **true**，不会有错

**Hystrix**

启用hystrix首先添加依赖

<!-- hystrix & Turbine -->

<dependency>

<groupId>com.netflix.hystrix</groupId>

<artifactId>hystrix-javanica</artifactId>

</dependency>

然后在启动类添加@EnableCircuitBreaker

@HystrixCommand(fallbackMethod = "methonName")不能直接作用Feign接口，可以加在调用feign接口的方法上

当被注解的方法抛出异常时，就会触发。

**Filter**

按照下面方式定义过滤器即可

@Component("loggerFilter ")

**public** **class** LoggerFilter **implements** Filter

**Spring session**

启动

Redis必须支持config命令，否则ERR unknown command 'CONFIG'

若config命令被禁用，需要配置bean

@Bean

**public** **static** ConfigureRedisAction configureRedisAction() {

**return** ConfigureRedisAction.***NO\_OP***;

}

zuul网关中需要配置  
zuul:

sensitiveHeaders: "\*"

## Freemarker

// 下面的注解会自动配置freemarker，导致自定义的spring.freemarker 相关配置不生效

//@EnableHystrixDashboard

不能通过**extends** FreeMarkerProperties来自定义freemarker的配置。

Spring Boot 会从以下路径寻找静态文件：

* /META-INF/resources/
* /resources/
* /static/
* /public/

可以通过配置

spring:

freemarker:

template-loader-path:

- /views

- classpath:/views/

suffix: .ftl

来改变，上面配置了两个路径，第一个搜索当前项目下webapp/views 目录；第二个是类路径/views/下

**注意：freemarker和RestController不能共用，因为**RestController**中包括@ResponseBody注解！！！，如下：**

@Controller

@ResponseBody

**public** **@interface** RestController

@Controller

@ResponseBody

**public** **@interface** RestController

可见@RestController包括@ResponseBody的所以要使用Freemarker

，则不用@RestController，直接用@Controller

静态资源

启动类上加入@EnableWebMvc，默认静态资源根目录为src/main/webapp

如果使用@EnableWebMvc了，那么就会自动覆盖了官方给出的/static, /public, META-INF/resources, /resources等存放静态资源的目录。而将静态资源定位于src/main/webapp。当需要重新定义好资源所在目录时，则需要主动添加下面的配置类，来Override addResourceHandlers方法。

@Configuration

@EnableWebMvc

@ComponentScan

public class WebConfig extends WebMvcConfigurerAdapter {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/\*\*").addResourceLocations("/");

}

}

I think stating "Using src/main/webapp is not recommended" can be misinterpreted by the reader. According to the documentation "Do not use the src/main/webapp directory *if your application will be packaged as a jar*", so you should mention that as well.

**Spring-boot-autoconfigure**

这个jar中都是spring boot默认设置的配置，比如org.springframework.autoconfigure.redis.RedisProperties，默认从application.yml文件中加载spring.redis 开头的属性。这样仅需要在配置文件设置属性值即可使用

# Ribbon

@RibbonClient(name = "foo", configuration = HiRibbonConfiguration.class)

这里是声明一个RibbonClient对象，这个对象名为foo，配置内容是HiRibbonConfiguration（可以在这里定义各种自定义bean）。

默认情况下ribbon会根据服务名去获取RibbonClient对象，也就是说请求foo服务时，会去取名为”foo”的RibbonClient对象，若不存在名为”foo”的RibbonClient对象，则获取默认的名为”default” RibbonClient对象。

所以，一般情况下@RibbonClient(name = "foo", configuration = HiRibbonConfiguration.class)是对调用某个服务的配置，上述配置仅对调用foo服务起作用，调用其他服务是默认的RibbonClient对象起作用。

要设置一个对所有服务生效的RibbonClient对象，需要让spring实例化HiRibbonConfiguration。也就是，可以在HiRibbonConfiguration类上加入注解@Configuration，并spring能扫描到其所在的包（@SpringBootApplication(scanBasePackages= 指定HiRibbonConfiguration所在的包路径）。

原理是：每次获取服务时，都会根据服务名去获取对应的RibbonClient对象，然后根据HiRibbonConfiguration的配置去实例化ILoadBalancer对象和其他定义的对象，若没有定义ILoadBalancer对象或者其他对象，则使用默认的。若定义了ILoadBalancer对象或者其他对象，则不会生成默认的对象。

默认bean在

spring-cloud-netflix-core-1.1.7.BUILD-SNAPSHOT-sources.jar中的org.springframework.cloud.netflix.ribbon.RibbonClientConfiguration代码。

画个图比较明白：

IClientConfig/IRule/IPing/ServerList/ILoadBalancer/ServerListFilter/RibbonLoadBalancerContext/ServerIntrospector 等bean

AppContext(Spring context)

Foo1ServiceContext

Foo2ServiceContext

AppBaseConfigure

Foo1BaseConfigure

DefaultBaseConfigure

Foo1BaseConfigure

Foo1

Foo2

Foo3

如图所示：在应用内部存在两种上下文，应用上下文（即spring根上下文）和服务上下文（每一个服务一个独立的上下文），服务上下文作为应用上下文的子上下文，可以访问应用上下文中的实例，反之不行。因此若在服务上下文中获取不到指定实例，则将到应用上下文中查找。

初始化时会根据配置情况生成IClientConfig/IRule/IPing/ServerList/ILoadBalancer/ServerListFilter/RibbonLoadBalancerContext/ServerIntrospector 等bean 并放置到对应的上下文中。放置规则是：spring 扫描到的@RibbonClient注解指定的configuration对象中定义的bean实例放入到应用上下文（@RibbonClient(name = "foo", configuration = HiRibbonConfiguration.class) 这个时候name是无意义的，甚至无需写@RibbonClient注解，只需要HiRibbonConfiguration类中加入类似@Configuration @Component 注解让spring发现即可）；其他spring扫描不到的@RibbonClient注解则生成name对应服务的上下文，并将其中的configuration指定类里定义的bean放入到服务上下文。在查找过程中，比如调用服务Foo1查找其中的ILoadBalancer，先查看Foo1ServiceContext中有没有ILoadBalancer类型的bean，没有则到AppContext中找。又比如服务Foo3，因为没有配置任何Foo3的东西，所以Foo3的上下文是空的，将使用AppContext中的。

另外，AppBaseConfigure中定义的bean会覆盖（不共存）DefaultBaseConfigure

中定义的相同类型bean

关键类：NamedContextFactory.createContext (String name)

另外需要关注的一个地方是：没有参数的DynamicServerListLoadBalancer构造的负载均衡实例是不会动态刷新的，需要传入IClientConfig clientConfig参数（可以参考RibbonClientConfiguration）。

# 参考

<http://dockone.io/article/510>

<http://www.infoq.com/cn/articles/basis-frameworkto-implement-micro-service>

<https://cloud.spring.io/spring-cloud-netflix/#quick-start>

<http://blog.loli.io/2016/02/21/spring-cloud-netflix-01/>

<http://my.oschina.net/openforce/blog/680941>

<http://sail-y.github.io/2016/03/21/Spring-cloud%E9%A1%B9%E7%9B%AE%E5%AE%9E%E8%B7%B5/>

<http://www.cnblogs.com/skyblog/category/774535.html>

<http://kubecloud.io/guide-spring-cloud-config/>

http://kubecloud.io/apigatewaypattern/

<http://www.xyuu.cn/spring-cloud-netflix-zhcn.html>