SpringCloud

——黎昭声 2016/12/14

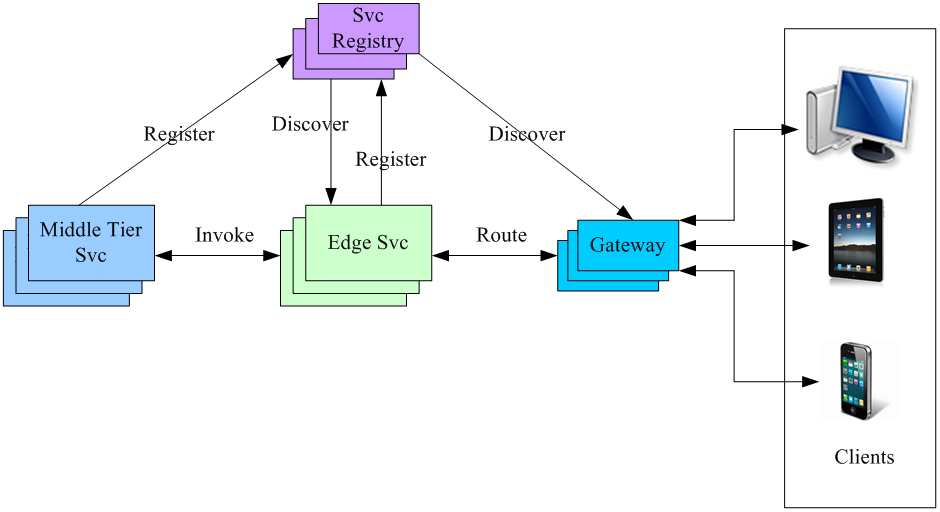
# 前言：BACKGROUD

个人理解微服务主要是为解决业务的复杂度和灵活度问题的，并不是主要为了解决代码复杂度和开发上的难题。因此，需要基于业务的角度来进行微服务化，而不是为了微服务化而微服务化。在构建一个微服务框架的过程中，不仅仅需要考虑到开发的痛点，好的架构应该满足平衡产品链中各个角色的利益点。例如产品经理对架构的诉求是能够支撑产品的快速迭代，开发对架构的诉求是对修改关闭对扩展开放，测试对架构的诉求是能够提高产品质量，运维对架构的诉求是快速定位响应线上问题等。因此对待微服务化的问题上应该三思而后行。

SpringCloud是一个Spring团队提供的微服务化方案。[何为微服务？](http://dockone.io/article/394)我不懂，请google自行理解。那还说啥呢？就说说对SpringCloud的理解和应用。

SpringCloud的技术基础是Spring Boot和Netflix的相关技术，具体来说就是Spring Boot的自动化加载技术和Netflix的服务注册发现、负载均衡、ZUUL网关技术等技术栈的合体。Netflix是一个视频流服务提供商，自己搞了个微服务方案，很强很流行。Spring Cloud目前仍然不算一个成熟的方案，在快速的迭代开发当中，好处就是提供了一整套涉及各个方面的完整解决方案。

下图是一个简化的微服务框架：



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



-来自 http://dockone.io/article/510-

下面我们就这个图使用一个demo进行简单的讲解各个角色。

# 微服务应用角色：DEMO

## Parent: Spring Cloud公用依赖

在开始discovery程序前，先定义parent pom文件，引入一些公共的jar

<parent>

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

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

<version>Brixton.BUILD-SNAPSHOT</version>

<relativePath/>

<!-- lookup parent from repository -->

</parent>

<!-- spring cloud -->

<!-- 服务注册，也就是所有eureka客户端都需要该jar进行服务注册 -->

<dependency>

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

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

</dependency>

<!-- 配置文件/注解 注解配置、配置中心相关需要使用 -->

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

</dependency>

<!-- 服务选择负载均衡，两种不同的使用方式feign是在ribbon之上的封装 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- web/rest rest风格的controller层-->

<dependency>

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

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

<!-- spring boot的基础包，但是因为已经被包含在spring-boot-starter-data-rest中，这里就不进行重复引入

<dependency>

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

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

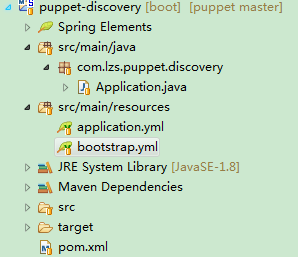
</dependency>

-->

上面包括了服务注册发现、配置相关、web层相关以及负载均衡相关的公共jar，当然也可以根据需要再添加或删除一些。

\*\*\*starter\*\*\* 是spring boot中的一些为了更好自动化配置定义的包，目的就是减少配置，只需要引入jar包并进行极少或者不需要任何配置，就可使用相应的组件。

## Discovery: 服务注册与提供



**pom.xml**

首先需要引入依赖：

<dependency>

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

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

**启动类**

编写启动类，非常简单，如下：

@SpringBootApplication

@EnableEurekaServer

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

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

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

}

}

因为spring boot内置了tomcat容器，因此不需要进行容器的配置并打成war放到容器中，直接启动主类即可。基本上所有的spring boot程序都是这样的启动方式：

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

@SpringBootApplication表明这是一个spring boot应用，主要是为了让spring boot能够根据依赖引入的各种starter进行自动配置。@SpringBootApplication包含了

@Configuration

@EnableAutoConfiguration

@ComponentScan

这几个注解，因此能够自动将依赖中的相应组件加载进来。

@EnableEurekaServer表明这是discovery服务端应用，接受Eureka客户端的服务注册请求和服务获取请求。

**配置文件**

yml文件和properties类似，只是在呈现方式上更加直观结构化，并且yml文件配置加载是有上下先后顺序的。更多详情请google

Spring Cloud中，bootstrap优先application加载，并且这两个文件名都是特殊的文件名（感觉，没做过严谨的测试）。

**bootstrap.yml**

spring:

application:

name: discovery

profiles:

active: discovery1 #default active profile

这里定义了应用名和激活的配置profile标识（下面会讲到）

**application.yml**

spring:

profiles: discovery1

server:

port: ${PORT:8761}

eureka:

instance:

hostname: discovery1

preferIpAddress: **true** # default false, but strong recommend to set true

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

preferIpAddress: **true**

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

preferIpAddress: **true**

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

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

---

spring:

profiles: discovery

server:

port: ${PORT:8761}

eureka:

instance:

hostname: discovery

client:

registerWithEureka: **false**

fetchRegistry: **false**

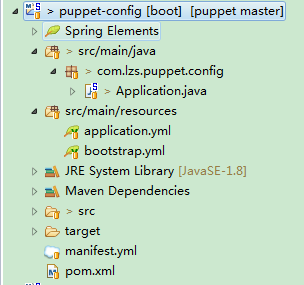
serviceUrl:

defaultZone: http://localhost:${server.port}/eureka/

上面有4种配置，通过---来分隔，相当于maven中的不同profile文件。我们之前在bootstrap.yml中激活的是profile=discovery的配置，这个配置中应用启动端口是server.port=8761，eureka实例hostname=discovery。${PORT:8761}表达式意思是：若存在参数PORT则使用PORT值，否则使用:号后面的值。hostname默认是主机名（计算机名），用于呈现，作用不大。eureka.instance还有许多的设置，都是一些服务实例信息，例如ip地址、实例id等。eureka.client 是服务注册和获取相关的配置，例如上面的registerWithEureka=false 表示自身不注册到eureka服务端（因为当前是discovery服务端，当然不注册自己到自己），默认true。fetchRegistry表示是否从eureka服务器获取注册信息，同理这里也是不需要的，默认true。defaultZone就比较重要了，是设置eureka服务器所在的地址，查询服务和注册服务都需要依赖这个地址。但是这个地址只对eureka客户端起作用，服务端不需要，所以这里设置defaultZone是无意义的。

另外的profile=discovery123 三种配置是集群的配置方式，例子中配置了3个discovery的服务端集群，服务端之间会定时进行信息同步。在集群模式下需要将自身注册到服务注册表registerWithEureka=true 以及能够感知获取服务fetchRegistry=true 并设置了不同的defaultZone，访问到另两个eureka服务器。

## Config: 配置中心



**pom.xml**

引入依赖：

<!-- 配置中心服务端 -->

<dependency>

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

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

</dependency>

**启动类**

@SpringBootApplication

@EnableDiscoveryClient

@EnableConfigServer

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

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

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

}

}

和discovery基本差不多，区别是@EnableConfigServer。表明当前应用是配置中心服务器。@EnableDiscoveryClient注解表示这是一个eureka客户端（和@EnableEurekaServer服务端对应）。应用启动后会根据相应的配置，将当前应用服务注册到eureka服务端，配置就是eureka.instance/eureka.client（见下面）。会有一些默认的配置，因此即使不设置eureka.instance/eureka.client也是可以正常使用的。

**配置文件**

**bootstrap.yml**

spring:

application:

name: config

**application.yml**

server:

port: ${PORT:0}

eureka:

instance:

hostname: config

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

preferIpAddress: **true** # default false, but strong recommend to set true

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

defaultZone: http://discovery:8761/eureka/ #,http://discovery1:8761/eureka/,http://discovery2:8762/eureka/,http://discovery3:8763/eureka/

spring:

cloud:

config:

server:

git:

uri: https://github.com/lizhaosheng/springcloud-demo.git

searchPaths: config-repo

大致和discovery差不多，注意到eureka.instance.instanceId 的值是一个很长的表达式。instanceId是服务实例注册id，要求不同的服务id必须不同，否则前面的相同id的服务将被后面的覆盖。instanceId值表达式中${spring.application.name}即bootstrap.yml文件中的对应值，${server.port}即application.yml的对应值，${random.value}是系统随机变量。可见，在yml文件中可以使用前面或者比当前文件先加载的文件中的值。

eureka.instance.preferIpAddress=true 表示在服务注册时优先使用ip地址（更多的是显示时以及获取服务信息时），这时即使设置了hostname，实际获取（代码或者eureka服务端界面看到的）到的hostname也是ip，默认preferIpAddress=false，hostname 不设置则是主机名。

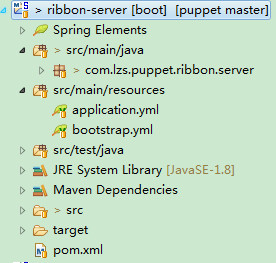
defaultZone设置服务注册以及获取的服务端地址（也就是discovery），注解掉的红色部分是集群的配置方式。

spring.cloud.config.server是配置中心服务器的配置，这里配置git方式获取，uri是git仓库地址，searchPaths是搜索路径。

**扩展：**

所有设置了从配置中心服务器获取配置的应用，在启动后会首先连接到配置中心服务器下载配置文件，并将下载的配置文件和本地的配置文件进行合并，然后再根据合并的配置文件开始程序的初始化的。合并同名配置本地配置优先远程配置。

## Ribbon-server: 服务提供



**pom.xml**

引入依赖

<parent>

<groupId>com.lzs.puppet</groupId>

<artifactId>puppet-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

不需要额外的依赖引入，只需要设置parent为我们之前创建的parent即可（里面已经加入了公用的依赖包，特别是spring-cloud-starter-eureka包就是eureka客户端所需要的包，满足ribbon-server使用）

**启动类**

@RestController

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** SayHelloApplication {

**private** **static** Logger *log* = LoggerFactory.*getLogger*(SayHelloApplication.**class**);

@RequestMapping(value = "/greeting")

**public** String greet() {

*log*.info("Access /greeting");

List<String> greetings = Arrays.*asList*("hi 12", "hello 34", "good bye 56");

Random rand = **new** Random();

**int** randomNum = rand.nextInt(greetings.size());

**return** greetings.get(randomNum);

}

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

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

}

}

注解@EnableDiscoveryClient表示这是一个eureka客户端，讲自身作为服务注册到服务端，为其他应用提供服务，也可从eureka服务端获取指定服务或服务列表。@RestController是一个REST风格MVC注解，包含了@Controller和@ResponseBody，因此需要注意使用这个注解请求返回都是ajax json格式消息包。

代码中响应请求/greeting并返回字符串问候语，供ribbon-client客户端服务消费调用。

**配置文件**

**bootstrap.yml**

spring:

application:

name: ribbon-server

cloud:

config:

discovery:

enabled: **true**

serviceId: config

name: manage

profile: test

这里启用了配置中心发现功能spring.cloud.config，将会从配置中心获取配置。配置中心以名为config的服务形式提供（discovery.enabled=true/discovery.serviceId=config），并指定获取的配置文件名为manage-test。注意name是必须的，另外config配置必须是在bootstrap.yml中而不能写在application.yml或其他配置文件（因为bootstrap被最先加载，而配置必须最先加载）。配置文件的获取方式如下：

资源文件的命名规则由以下的三个参数确定：

* {application}映射到应用的spring.application.name属性
* {profile}映射到应用的spring.profiles.active属性，可以用来区分环境，比如dev，test，produce等等
* {label}映射到svn/Git服务器的commit id,分支名称或者tag，默认值为master

对应文件名和规则如下（配置文件的根目录就是config配置中心的searchPaths配置的目录）：

/{application}-{profile}.yml

/{application}-{profile}.properties

**application.xml**

server:

port: ${PORT:0}

eureka:

instance:

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

hostname: ribbon-server

preferIpAddress: **true** # default false, but strong recommend to set true

metadataMap:

turbine-cluster: MAIN # 集群名，给turbine使用，表明当前hystrix信息是属于哪个集群的

env: dev

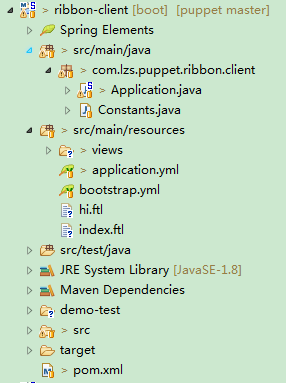
client:

serviceUrl:

defaultZone: http://discovery:8761/eureka/ # ,http://discovery1:8761/eureka/,http://discovery2:8762/eureka/,http://discovery3:8763/eureka/

和之前config基本一致，metadataMap是一些自定义数据，将会被当做服务实例信息注册到服务端。

## Ribbon-client: 服务消费



**pom.xml**

和ribbon-server一致

**启动类**

@SpringBootApplication(scanBasePackages = { "com.lzs.puppet.ribbon.client","com.lzs.puppet.base.redis","com.lzs.puppet.base.route"})

@EnableDiscoveryClient

@Controller

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

@LoadBalanced

@Bean

RestTemplate restTemplate() {

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

}

@Autowired

RestTemplate restTemplate;

@RequestMapping("/serv")

@ResponseBody

**public** String serv(@RequestParam(value = "name", defaultValue = "Artaban") String name) {

String greeting = **this**.restTemplate.getForObject("http://ribbon-server/greeting", String.**class**);

**return** String.*format*("%s, %s!", greeting, name);

}

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

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

}

}

注解和ribbon-server基本一致，这里使用@Controller而不是@RestController，所以在方法中若要返回ajax json需要加入@ResponseBody。

另外注解@SpringBootApplication，加上了参数scanBasePackages。指定spring注解扫描路径（默认扫描当前所在类的类路径以及子路径）。

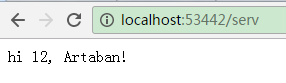
最重要的是在这个类中，我们通过@LoadBalanced和@Bean注解在spring上下文中生成了一个负载均衡对象restTemplate。该对象负责从eureka服务端获取服务列表，并根据负载均衡规则（有默认规则）选择某个服务实例。

在方法serv中，通过该对象restTemplate进行服务调用：

String greeting = **this**.restTemplate.getForObject("http://ribbon-server/greeting", String.**class**);

getForObject方法参数1是服务地址，该地址中的域名部分就是被调用服务的id（也就是服务名，例子中的ribbon-server）,后面greeting是该服务应用上的请求url。参数2是返回值类型，这里是返回字符串。

结合ribbon-server在52286端口，ribbon-client在53442端口，在浏览器中输入<http://localhost:53442/serv> ，得到结果如下：



可见ribbon-client调用了ribbon-server的服务并返回。

**配置文件**

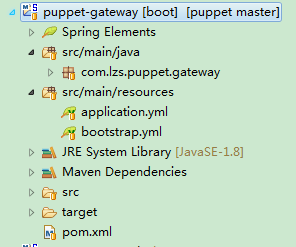
**bootstrap.yml**

和ribbon-server一样，注意服务名不同

**application.xml**

和ribbon-server一样，注意服务名不同

## Gateway: 网关



**pom.xml**

引入依赖

<parent>

<groupId>com.lzs.puppet</groupId>

<artifactId>puppet-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<!-- spring cloud zuul-->

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-netflix-sidecar</artifactId>

</dependency>

</dependencies>

添加了和网关相关的一些jar（只需要spring-cloud-netflix-sidecar，这个jar貌似也都包含了spring-cloud-starter-zuul的jar）

**启动类**

@SpringBootApplication

@EnableSidecar

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

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

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

}

}

**配置文件**

**bootstrap.yml**

spring:

application:

name: gateway

cloud:

config:

discovery:

enabled: **true**

serviceId: config

**application.xml**

server:

port: 10000 # 外侧端口号

sidecar:

port: 8000 # 内侧端口号，也就是系统内和其他应用通信的端口

endpoints: # 一些内置请求，通过这些请求可以查询当前网关信息

restart:

enabled: **true**

shutdown:

enabled: **true**

health:

sensitive: **true** # remenber to set it false in prod environment

eureka:

instance:

hostname: gateway

instanceId: ${spring.cloud.client.hostname}:${spring.application.name}:${server.port}

preferIpAddress: **true** # default false, but strong recommend to set true

client:

serviceUrl:

defaultZone: http://discovery:8761/eureka/ #,http://discovery1:8761/eureka/,http://discovery2:8762/eureka/,http://discovery3:8763/eureka/

zuul:

sensitiveHeaders: "\*" # session can be transfered from gateway to applications

ignoredServices: '\*' # all services are ignored except those defined below.

routes:

manage:

path: /manage/\*\*

serviceId: demo-manage

# stripPrefix: false # default true /hello/login -> /login , if false /hello/login -> /hello/login

client-api:

path: /client-api/\*\*

serviceId: demo-client-api

# stripPrefix: false # default true /hello/login -> /login , if false /hello/login -> /hello/login

ribbon-client:

path: /ribbon-client/\*\*

serviceId: ribbon-client

# stripPrefix: false # default true /hello/login -> /login , if false /hello/login -> /hello/login

# 超时时长设置

#hystrix:

# command:

# default:

# execution:

# isolation:

# thread:

# timeoutInMilliseconds: 1500

最关键的是zuul.routes配置，看配置能看懂就不详细说了，要注意的是被注释掉的stripPrefix，若false表示网关请求到服务的url路径包括服务名在前面，true则请求到服务实例的url上没有服务名。ignoredServices表示忽略的服务（可通配符）不处理，但除去配置在routes路由表中的服务，通常设置’\*’忽略所有服务，只处理路由表中指定的。

Zuul内部也是使用的ribbon负载均衡（下面会讲解）来获取服务实例。

# 自动化配置组件：SOLUTION

## Jdbc: 数据源-mysql

**自动化配置**

使用spring boot 默认的自动化配置mysql非常简单，只需要两步

1. 引入相应依赖

<!-- jdbc & mybatis -->

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>1.1.1</version>

</dependency>

1. 配置数据库访问信息

spring:

datasource:

driverClassName: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/test

username: name

password: password

配置信息可以放在当前应用的application.yml里，也可以统一放在配置中心的application.yml中。

出现问题无法启动？检查下配置错没错。

**自定义配置**

相对第一种，自定义配置需要多一步，就是定义相关的配置bean和数据库实例bean。

首先是数据库连接信息配置类，

Properties类如下：

@ConfigurationProperties(prefix = PuppetDataSourceProperties.***PREFIX***)

**public** **class** PuppetDataSourceProperties {

**public** **final** **static** String ***PREFIX*** = "jdbc";

**private** String driverClassName;

**private** String url;

**private** String username;

**private** String password;

**private** **int** maxActive = 100;

**private** **int** maxIdle = 8;

**private** **int** minIdle = 8;

**private** **int** initialSize = 10;

**private** String validationQuery;

**private** **boolean** testOnBorrow = **false**;

**private** **boolean** testOnReturn = **false**;

}

@ConfigurationProperties会从类路径的配置文件中加载配置，这里指定了配置的前缀是jdbc，也就是说需要提供配置jdbc.url jdbc.username 等（自动化配置中数据库的配置前缀是spring.datasource）。然后需要一个根据这些配置生成必要的bean的配置类

Configuration类如下：

@Configuration

@EnableConfigurationProperties(PuppetDataSourceProperties.**class**)

**public** **class** DataSourceConfiguration {

@Autowired

**private** PuppetDataSourceProperties properties;

**private** org.apache.tomcat.jdbc.pool.DataSource dataSource;

@Bean(destroyMethod = "close")

**public** DataSource dataSource() {

**this**.dataSource = **new** org.apache.tomcat.jdbc.pool.DataSource();

**this**.dataSource.setDriverClassName(properties.getDriverClassName());

**this**.dataSource.setUrl(properties.getUrl());

**if** (properties.getUsername() != **null**) {

**this**.dataSource.setUsername(properties.getUsername());

}

**if** (properties.getPassword() != **null**) {

**this**.dataSource.setPassword(properties.getPassword());

}

**this**.dataSource.setInitialSize(properties.getInitialSize());

**this**.dataSource.setMaxActive(properties.getMaxActive());

**this**.dataSource.setMaxIdle(properties.getMaxIdle());

**this**.dataSource.setMinIdle(properties.getMinIdle());

**this**.dataSource.setTestOnBorrow(properties.isTestOnBorrow());

**this**.dataSource.setTestOnReturn(properties.isTestOnReturn());

**this**.dataSource.setValidationQuery(properties.getValidationQuery());

**return** **this**.dataSource;

}

@PreDestroy

**public** **void** close() {

**if** (**this**.dataSource != **null**) {

**this**.dataSource.close();

}

}

}

@Configuration表明这是一个配置类，里面存着一个或多个@Bean注解方法需要spring在运行时定义并创建bean。@EnableConfigurationProperties实际上和@Bean的作用相当，专门用于创建@ConfigurationProperties注解类的bean。所以我们才可以将该bean注入（@Autowired **private** PuppetDataSourceProperties properties;）

**使用**

在需要的地方直接注入@Autowired DataSource dataSource; 即可

## Mybatis: 数据访问层

**自动化配置**

和datasource的自动化配置过程相似，需要引入依赖包并编写配置信息，如下：

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>1.1.1</version>

</dependency>

配置文件

mybatis:

mapper-locations:

- classpath:/com/lzs/puppet/demo/\*\*/dao/\*.xml

mapper-locations是数组配置，classpath:是搜索类路径，非 classpath是哪个为根目录我没测试出来也google不到，有时间可以看看源码。

同样配置信息可以放配置中心

**自定义配置**

和datasource类似，在spring cloud中自定义配置一般都差不多。先编写自己的配置属性类（Properties，大部分也可以直接使用spring boot内置的属性类，比如mybatis内置属性类是MybatisProperties），然后编写bean定义创建类（Configration）

Configuration类如下：

@Configuration

@EnableTransactionManagement

**public** **class** MybatisConfiguration **implements** TransactionManagementConfigurer {

@Autowired

DataSource dataSource;

/\*\*

\* 使用spring boot内置的MybatisProperties，当然也可以自己定义

\*/

@Autowired

MybatisProperties mybatisProperties;

@Bean

**public** SqlSessionFactory sqlSessionFactoryBean(DataSource ds) **throws** Exception{

SqlSessionFactoryBean bean = **new** SqlSessionFactoryBean();

bean.setDataSource(dataSource);

// //添加插件

// bean.setPlugins(new Interceptor[]{pageHelper});

// XML文件中的model类的基包路径

bean.setTypeAliasesPackage(mybatisProperties.getTypeAliasesPackage());

//添加XML目录

bean.setMapperLocations(mybatisProperties.resolveMapperLocations());

**return** bean.getObject();

}

@Bean

**public** SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {

**return** **new** SqlSessionTemplate(sqlSessionFactory);

}

@Bean

@Override

**public** PlatformTransactionManager annotationDrivenTransactionManager() {

**return** **new** DataSourceTransactionManager(dataSource);

}

}

看到了熟悉的@Configuration和@Bean组合，创建mybatis必要的bean（所有在Configuration创建的bean通常都会覆盖默认bean，不显式创建的其他bean都使用默认自动化生成）。

另外这里没有编写自定义的properties类，而是直接注入了默认创建的properties类（@Autowired MybatisProperties mybatisProperties;）。其他大部分组件也可以这么干。

**使用**

在mapper接口上加入注解，如下例子：

@Mapper

**public** **interface** StaffDao

## Redis: 缓存

**自动化配置**

还是一样的套路先引入依赖包并编写配置信息，如下：

<dependency>

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

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

</dependency>

配置文件

spring:

redis:

database: 0 # schema

host: 127.0.0.1

port: 6379

password: ZZdfawt4e # Login password of the redis server.

timeout: 200

pool.max-active: 100

pool.max-idle: 20

pool.max-wait: 50

pool.min-idle: 1

transactionSupport: **false**

同样配置信息可以放配置中心

**自定义配置**

这里我将properties和configuration都重新定义了

Properties类如下：

@ConfigurationProperties(prefix = PuppetRedisProperties.***PREFIX***)

**public** **class** PuppetRedisProperties **extends** RedisProperties{

**public** **final** **static** String ***PREFIX*** = "redis";

/\*\* 是否支持事务，默认支持\*/

**private** **boolean** transactionSupport = **true**;

**public** **boolean** isTransactionSupport() {

**return** transactionSupport;

}

**public** **void** setTransactionSupport(**boolean** transactionSupport) {

**this**.transactionSupport = transactionSupport;

}

}

虽然继承了默认的RedisProperties类，但是重新定义了配置前缀为redis，添加配置项transactionSupport。这样我们不仅可以设置和默认配置相同的配置项，也可以添加自己特殊需求的配置项。

配置文件如下：

redis:

database: 0 # schema

host: 127.0.0.1

port: 6379

password: ZZdfawt4e # Login password of the redis server.

timeout: 200

pool.max-active: 100

pool.max-idle: 20

pool.max-wait: 50

pool.min-idle: 1

transactionSupport: **false**

可见spring.redis 前缀换成了redis前缀，并且多出了一个默认配置没有的配置项transactionSupport，

Configuration类如下：

@Configuration

@EnableCaching

**public** **class** RedisConfiguration {

@Bean(name = "puppetRedisProperties")

**public** PuppetRedisProperties redisProperties() {

**return** **new** PuppetRedisProperties();

}

@Bean(name = "puppetRedisTemplate")

**public** PuppetRedisTemplate redisTemplate(RedisConnectionFactory factory) {

PuppetRedisTemplate template = **new** PuppetRedisTemplate();

template.setConnectionFactory(factory);

setSerializer(template); // 设置序列化工具

template.afterPropertiesSet();

**return** template;

}

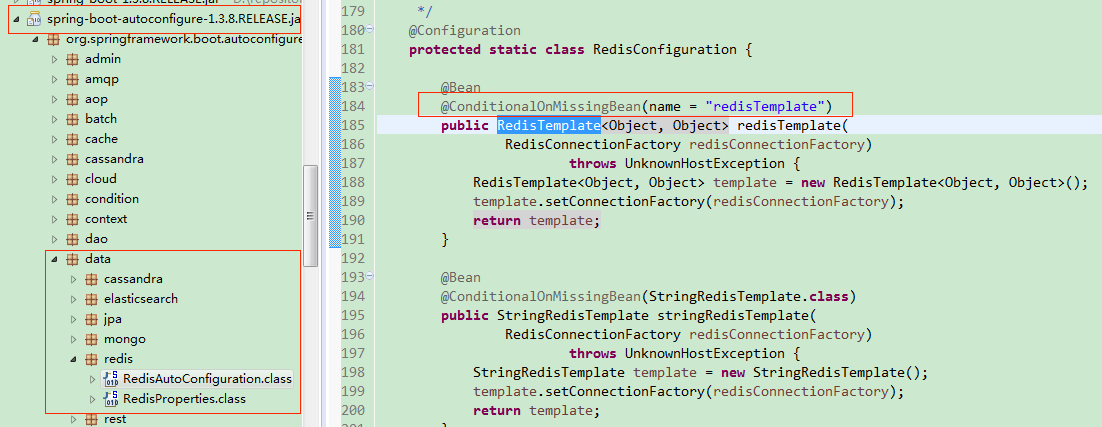
可以看到创建的bean类型是PuppetRedisTemplate，是我们自定义的类

**public** **class** PuppetRedisTemplate **extends** RedisTemplate<String, Object> {

@Autowired

**private** PuppetRedisProperties puppetRedisProperties;

继承的是RedisTemplate，结果是怎样的呢？先看看默认的相关bean



在**Spring-boot-autoconfigure**包中是常用组件自动化配置加载实现，找到redis的configuration可以看到RedisTemplate的bean注解@ConditionalOnMissingBean(name = "redisTemplate")。也就是当不存在名为redisTemplate的bean时将会创建。所以我们的自定义configuration中@Bean(name = "puppetRedisTemplate")结果就是不会覆盖默认的redisTemplate bean！也就是将会创建两个redisTemplate的实例（PuppetRedisTemplate **extends** RedisTemplate）。

另外注意到properties是通过@Bean创建的，其实可以通过之前datasource中说过的更方便的方式

@EnableConfigurationProperties(PuppetRedisProperties.**class**)

**使用**

在需要的地方直接注入@Autowired RedisTemplate redisTemplate; （或者在自定义方案中注入 @Autowired PuppetRedisTemplate redisTemplate;）即可

注意spring-boot-starter-redis使用的是logback日志，若项目中使用的是slf4j+log4j则需要进行排除，如下

<dependency>

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

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

<!-- 项目中使用的是slf4j+log4j 和logback冲突 -->

<exclusions>

<exclusion>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-core</artifactId>

</exclusion>

<exclusion>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

</exclusion>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

</exclusion>

</exclusions>

</dependency>

## Rabbitmq: 消息队列

暂时没用过

<http://www.jianshu.com/p/048e954dab40>

## Freemark: 视图模板

**自动化配置**

引入依赖

<dependency>

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

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

</dependency>

默认模板配置

spring:

freemarker:

template-loader-path:

- /views # webapp/views/

- classpath:/views/

suffix: .ftl

模板路径有两种搜索方式：文件系统和类路径方式。注意文件系统方式的根目录是/src/main/webapp

**自定义配置**

包依赖和配置文件与之前的类似

在freemark这里将properties类和configuration类合二为一，如下：

@Configuration

@ConfigurationProperties(prefix = "freemarker")

**public** **class** FreeMarkerConfiguration **extends** FreeMarkerProperties{

@Bean

**public** FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = **new** FreeMarkerConfigurer();

applyProperties(configurer);

**return** configurer;

}

**private** **void** applyProperties(FreeMarkerConfigurer factory) {

factory.setTemplateLoaderPaths(getTemplateLoaderPath());

factory.setPreferFileSystemAccess(isPreferFileSystemAccess());

factory.setDefaultEncoding(getCharsetName());

Properties settings = **new** Properties();

settings.putAll(getSettings());

factory.setFreemarkerSettings(settings);

}

}

本质上和上面介绍过的组件是一样，也都是properties接收配置文件的配置项，configuration创建必要的bean，这是这里将这两个类合并在一起了。通过注解

@Configuration

@ConfigurationProperties(prefix = "freemarker")

将当前类声明为一个configuration类也是一个properties类，同时定义了前缀freemarker。然后继承了默认FreeMarkerProperties类的所有配置项。

**使用**

直接返回字符串视图名即可

注意controller层不能使用@RestController 注解，因为该注解包含@ResponseBody 返回json格式的数据，并非字符串视图名。

注意@EnableHystrixDashboard注解会自动配置freemarker，导致spring.freemarker 相关配置不生效

另外从默认自动化配置类可以看到

@EnableConfigurationProperties(FreeMarkerProperties.**class**)

**public** **class** FreeMarkerAutoConfiguration {

FreeMarkerProperties类bean默认被创建不能覆盖，因此最好别写自己的properties类了，搞出问题自己慢慢查。

## Session: 分布式Session

Spring session 需要redis组件配合一起才能使用的，所以需要先配置好redis（当然spring session也支持其他方式的存储，没研究过其他的）。

**自动化配置**

引入依赖

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session</artifactId>

</dependency>

在启动类上加入注解

@EnableRedisHttpSession

**自定义配置**

上面的第一步还是需要做的，其他需要自定义的可以参考[这里](http://www.infoq.com/cn/articles/Next-Generation-Session-Management-with-Spring-Session)。我给出我的一个配置

@EnableRedisHttpSession

**public** **class** SpringSessionInitializer **extends** AbstractHttpSessionApplicationInitializer {

**public** SpringSessionInitializer() {

**super**(RedisConfiguration.**class**);

}

@Bean

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

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

}

}

因为用的是自定义的redis，所以需要自定义的SpringSessionInitializer，将redis配置设置关联起来。

另外创建了ConfigureRedisAction Bean，是因为在我的环境中使用的redis不支持（禁用）config命令，而spring session初始化的时候默认会调用redis的一些高级管理命令，比如config命令。这里创建了一个ConfigureRedisAction bean，并告诉spring初始化的时候不进行任何redis操作（ConfigureRedisAction.***NO\_OP***）。

## Consult: 总结

从以上的几个常用组件可以看出，spring cloud（或者说spring boot）中通用组件可以通过两种方式配置加载：自动化/自定义，并且不同组件的配置加载方式大同小异。对于自动化配置通常是三个个步骤：引入相关依赖包（通常是\*\*\*starter\*\*\*）；编写模板化配置信息；在需要的地方直接注入对应的bean即可。对于自定义配置通常是：引入相关依赖包（通常是\*\*\*starter\*\*\*）；编写模板化或自定义properties配置信息；编写configuration类覆盖默认实现bean（注意有些bean需要指定特定的beanname才能覆盖，详细可以去看看jar包**Spring-boot-autoconfigure**）；在需要的地方直接注入对应的bean即可。

所有的组件都有大量的配置项，spring boot会为我们设置了默认值，实际使用中仅需要修改覆盖部分常用的配置，需要了解更多的配置项可以到**Spring-boot-autoconfigure**包中查看相关的properties类，或者google([Common application properties](http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html))。自动化配置组件Configuration类（[Auto-configuration classes](http://docs.spring.io/spring-boot/docs/current/reference/html/auto-configuration-classes.html)）

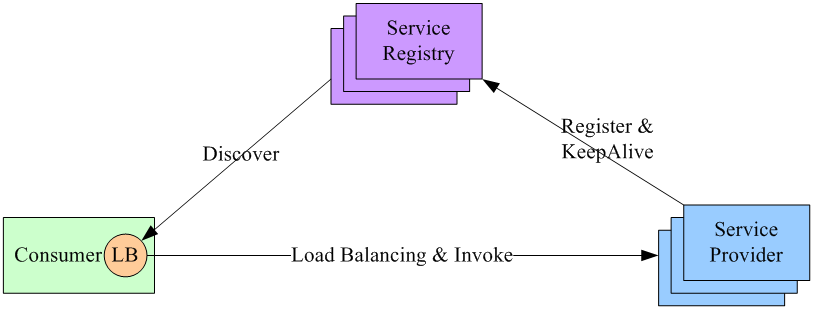
# 关键技术：ANALYSIS

## SpringBoot自动化配置

## Eureka服务注册和感知

微服务最重要的就是功能模块服务化。在Spring Cloud中一个服务实例就是一个运行的程序应用，服务实例和服务实例之间通过HTTP协议或者其他轻量级通信协议进行通信。为了达到灵活扩展和服务之间的解耦合，需要一个中间的服务注册表用于注册和查询服务实例。Spring Cloud Discovery就是这样的一个角色

服务注册发现组件，Spring Cloud Discovery使用的是Netflix的技术——Eureka。Eureka分为服务端和客户端（如没有特别指明，下面所有的客户端指的是eureka客户端，而不是手机客户端等）。服务端即服务注册表，客户端即服务实例（即提供自身为服务，也可能消费其他服务）。首先存在一个服务端，客户端启动时，会根据配置信息，将自身服务信息（包括服务名、实例url地址等）注册到服务端，即声明了一个服务。当请求服务时，客户端首先向服务端查询指定服务的所有服务实例，然后根据一定的规则（轮询、随机等）获取到具体服务实例信息，并根据对应的url请求服务。如下图：



注意：在Eureka中消费者也是服务提供者，上图中Consumer和Service Provider都是客户端。

服务端和不同的客户端都是独立的项目进程。一个服务实例实质上就是一个运行的程序进程（ip+port定位）

## Ribbon客户端负载均衡

## Config配置中心

# 实践：EXAMPLE

## 传统Spring MVC项目迁移

## 扩展SpringCloud服务过滤增强

## 静态文件处理

# 参考

<http://www.infoq.com/cn/articles/Next-Generation-Session-Management-with-Spring-Session>