选择了一个开源的Spring Cloud实战项目来说明应该如何使用Spring Cloud。

项目介绍

功能点

paascloud（ **https://github.com/paascloud/paascloud-master** ）是模拟商城，有完整的购物流程、后端运营平台对前端业务的支撑，和对项目的运维，有各项的监控指标和运维指标

技术栈
核心框架：springcloud Edgware全家桶，Vye全家桶
安全框架：Spring Security Spring Cloud Oauth2
分布式任务调度：elastic-job
持久层框架：MyBatis、通用Mapper4、Mybatis_PageHelper
数据库连接池：Alibaba Druid
日志管理：Logback 前端框架：Vue全家桶以及相关组件
三方服务： 邮件服务、阿里云短信服务、七牛云文件服务、钉钉机器人服务、高德地图API

目录结构说明

微服务架构目录形式

如果是采用Maven来进行构建的话，按照Maven分布式工程的基本架构来进行创建。

```
 microservice
---- microservice-common
---- microservice-modules
---- microservice-provider
---- microservice-provider-api
---- microservice-consumer
```

其中microservice为父工程模块，来定义整个工程的依赖版本，因为Spring Cloud中各个组件的版本都是有对应关系的，同时在根pom中定义项目中的模块。

```
<groupId>com.liuzm.paascloud</groupId>
<artifactId>paascloud-master</artifactId>
```

```
<version>1.0</version>
<modules>
    <module>paascloud-eureka</module>
    <module>paascloud-discovery</module>
    <module>paascloud-monitor</module>
    <module>paascloud-zipkin</module>
    <module>paascloud-gateway</module>
    <module>paascloud-provider-api</module>
    <module>paascloud-common</module>
    <module>paascloud-provider</module>
    <module>paascloud-generator</module>
</modules>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <springfox.version>2.6.1</springfox.version>
    <commons-codec.version>1.10</commons-codec.version>
    <jjwt.version>0.7.0</jjwt.version>
    <codehaus-jackson.version>1.9.13</codehaus-jackson.version>
    <mybatis-starter-version>1.2.0</mybatis-starter-version>
    <mybatis.plus.version>2.0.7</mybatis.plus.version>
    <druid.version>1.0.29</druid.version>
    <mapper.version>3.4.0</mapper.version>
    <springframework.version>4.3.8.RELEASE</springframework.version>
    </properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
 </dependency>
</dependencies>
```

在下一级目录的Maven子工程中，要添加父pom的信息，

```
<parent>
    <groupId>com.liuzm.paascloud</groupId>
    <artifactId>paascloud-master</artifactId>
    <version>1.0</version>
</parent>
```

同理，在三级目录的maven子工程要依赖于二级maven工程的pom,这样利用Maven的依赖传递去控制所使用的构件的版本。

microservices-common包为公共模块，来定义一些公共的组件和配置类。

microservices-moudles是微服务架构中基础组件的代指（比如注册中心，配置中心，监控中心等等）。

provider为服务提供者，提供api供其它微服务使用，这两部分的api最好分包存放。

provider-api是提供api给其它微服务使用，这样做的目的是隐藏api的实现细节，重新定义两个微服务之间交互所需要使用的数据对象，应该要服务微服务建模时划分的限界上下文。

consumer为服务消费者，用于消费内部实现的微服务。实际上，提供者同时也可能是其它一些微服务的消费者，但是通常来说provider在命名定义的优先级更高，当使用其它微服务提供的api时，需要在pom中引入依赖，然后在services层实现自己服务的时候利用依赖注入注入其它微服务的api，利用Feign就像是像在本地操作数据库那样操作其它微服务的数据。以下是三步的演示。

```
<dependency>
    <groupId>com.liuzm.paascloud.provider.api</groupId>
    <artifactId>paascloud-provider-mdc-api</artifactId>
</dependency>
```

```
@Resource
private MdcProductQueryFeignApi mdcProductQueryFeignApi;
@Resource
private MdcProductFeignApi mdcProductFeignApi;
```

```
ProductDto product = mdcProductService.selectById(cartItem.getProductId());
```

此项目的目录结构

此项目选择了eureka作为注册中心，spring-cloud-config作为配置中心，hystrix dashboard和spring-boot-admin作为监控中心，netflix zuul作为网关，zipkin为日志采集中心。这些基础模块作为微服务架构中的服务治理部分。mdc、omc、opc、tpc、uac作为业务模块。

```
├─paascloud-master----------------------------父项目，公共依赖
|  |
|  ├─paascloud-eureka--------------------------微服务注册中心
|  |
|  ├─paascloud-discovery-----------------------微服务配置中心
|  |
|  ├─paascloud-monitor-------------------------微服务监控中心
|  |
|  ├─paascloud-zipkin--------------------------微服务日志采集中心
|  |
|  ├─paascloud-gateway-------------------------微服务网关中心
|  |
|  ├─paascloud-provider
|  |  |
|  |  ├─paascloud-provider-mdc-----------------数据服务中心
|  |  |
|  |  ├─paascloud-provider-omc-----------------订单服务中心
|  |  |
|  |  ├─paascloud-provider-opc-----------------对接服务中心
|  |  |
|  |  ├─paascloud-provider-tpc-----------------任务服务中心
|  |  |
|  |  └─paascloud-provider-uac-----------------用户服务中心
|  |
|  ├─paascloud-provider-api
|  |  |
|  |  ├─paascloud-provider-mdc-api-------------数据服务中心API
|  |  |
|  |  ├─paascloud-provider-omc-api-------------订单服务中心API
|  |  |
|  |  ├─paascloud-provider-opc-api-------------对接服务中心API
|  |  |
|  |  ├─paascloud-provider-tpc-api-------------任务服务中心API
|  |  |
|  |  ├─paascloud-provider-sdk-api-------------可靠消息服务API
|  |  |
|  |  └─paascloud-provider-uac-api-------------用户服务中心API
```

```
|   |
|   ├─paascloud-common
|   |   |
|   |   ├─paascloud-common-base------------------公共POJO基础包
|   |   |
|   |   ├─paascloud-common-config----------------公共配置包
|   |   |
|   |   ├─paascloud-common-core------------------微服务核心依赖包
|   |   |
|   |   ├─paascloud-common-util------------------公共工具包
|   |   |
|   |   ├─paascloud-common-zk--------------------zookeeper配置
|   |   |
|   |   ├─paascloud-security-app-----------------公共无状态安全认证
|   |   |
|   |   ├─paascloud-security-core----------------安全服务核心包
|   |   |
|   |   └─paascloud-security-feign---------------基于auth2的feign配置
|   |
|   ├─paascloud-generator
|   |   |
|   |   ├─paascloud-generator-mdc----------------数据服务中心Mybatis Generator
|   |   |
|   |   ├─paascloud-generator-omc----------------数据服务中心Mybatis Generator
|   |   |
|   |   ├─paascloud-generator-opc----------------数据服务中心Mybatis Generator
|   |   |
|   |   ├─paascloud-generator-tpc----------------数据服务中心Mybatis Generator
|   |   |
|   |   └─paascloud-generator-uac----------------数据服务中心Mybatis Generator
```

组件使用说明

通过代码简单介绍Spring Cloud的组件应该如何使用，注意，这一切都是在Spring Boot的基础项目之上进行的操作。

Eureka

首先启动注册中心

- 添加maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- 在启动类上添加 `@EnableEurekaServer` 注解
- 启动项目即可

接下来就可以把自己写的微服务添加到注册中心，

- 添加maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- 启动类上添加注解 `@EnableDiscoveryClient`
- 添加配置为注册中心的地址并且配置自己的实例名:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://root:root@paascloud-eureka:8761/eureka/
  instance:
    instance-id:
${spring.application.name}:${spring.cloud.client.ipAddress}:${server.port}
```

  也可以设置一些连接参数如心跳时间等等。

这样在eureka提供的web界面上就可以看到注册的服务和它们的状态。

更高级的使用方法比如eureka的高可用部署模式以及提供的API这里不赘述。

config

同样也要启动配置中心

- 首先添加maven依赖

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
```

- 在启动类添加注解 `@EnableConfigServer`
- 配置配置仓库的地址

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/paascloud/paascloud-config-repo.git
          username:
          password:
          search-paths: /*
          default-label: ${spring.profiles.active}
          basedir: /data/config/paascloud-config-repo
```

  配置仓库是配置中心搜寻服务配置的地方，这里存放了所有的配置文件，配置中心根据客户端配置的文件名寻找它的配置文件并且发送给它。

- 启动即可

也可以将配置中心作为一个微服务同样注册到注册中心，这样在微服务中就不需要些配置中心的实际地址而只写逻辑名，假设使用了这种方法再将微服务接入到配置中心。

- 首先添加依赖

注意，只要添加了此项依赖，如果不进行配置的话也会在本地8888端口寻找配置文件。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

- 新建bootstrap的配置文件，配置配置中心所在的位置，这里直接写了逻辑名。所寻找的配置文件的名字是applicationname-label-profile。

```
spring:
  cloud:
    config:
      fail-fast: true
      discovery:
        service-id: paascloud-discovery
        enabled: true
      label: ${spring.profiles.active}
      profile: ${spring.profiles.active}
      username: admin
      password: admin
```

高级应用是利用git的webhook功能和Spring Cloud Bus实现当配置刷新的时候自动更新程序的配置而不用将应用重新部署。

Monitor

监控中心使用了Hystrix的Dashboard监控微服务之间的调用关系，Hystrix的Turbine监控集群微服务的调用健康关系，spring-boot-admin监控单个服务的健康状态。

- 添加依赖

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server</artifactId>
    </dependency>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server-ui</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
```

```
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server-ui-hystrix</artifactId>
    </dependency>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server-ui-turbine</artifactId>
    </dependency>
```

- 添加注解

```
@EnableTurbine
@EnableHystrixDashboard
@EnableCircuitBreaker
@EnableAdminServer
```

- 启动

  就可以在这三个监控组件自带的前端界面上看到我们系统中微服务的健康情况。

  对于客户端的使用,spring-boot-admin的使用比较简单，添加依赖注解然后配置admin server的位置即可，hystrix可以集成在feign中使用，在后面介绍。

网关

网关也是一个单独的项目，这里使用Netflix的zuul

- 首先添加注解

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

- 启动类添加注解 `@EnableZuulProxy`
- 启动

  这样是一个空的网关，需要在配置文件中添加路由规则（即访问路径和服务实例之间的对应关系，支持硬编码和服务化的路由规则），此外要实现权限、错误控制等功能需要添加filter来实现，这样实现网关虽然在功能的实现上会简单，但是会造成网关变成一个庞然大物，所以在设计网关的具体功能时还是要仔细考虑。

Feign

Spring Cloud使用Netflix的Hystrix做错误处理，Ribbon做负载均衡，而Feign就是这两者的综合，推荐在微服务之间的调用都使用Feign client代理，这样只需要很少的配置就可以实现两大功能。具体的使用说明如下:

- 添加依赖

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>
```

- 在provider-api中有一个这样的方法

```
@Component
public class MdcAddressQueryFeignHystrix implements MdcAddressQueryFeignApi {

    @Override
    public Wrapper<AddressDTO> getById(final Long addressId) {
        return null;
    }
}
```

而自身有一个服务需要使用远程服务的api，所以使用 `@FeignClient` 注解，value代表要调用的服务实例，如果这个服务实例名会有多个会自动做负载均衡（Ribbon实现），fallback代表如果调用失败会调用此函数进行服务降级（Hystrix实现）。

```
@FeignClient(value = "paascloud-provider-mdc", configuration =
OAuth2FeignAutoConfiguration.class, fallback = MdcAddressQueryFeignHystrix.class)
public interface MdcAddressQueryFeignApi {

    /**
     * Select by id wrapper.
     *
     * @param addressId the address id
     *
     * @return the wrapper
     */
    @PostMapping(value = "/api/address/getById/{addressId}")
    Wrapper<AddressDTO> getById(@PathVariable("addressId") Long addressId);
}
```

这是最基本的实现，可以通过Hystrix设置超时的时间和重试的次数，对于服务降级需要根据实际的需求进行设计，服务降级可以进行多层，推荐服务降级的最后一层是不依赖于网络的，这样能够保证起码能够返回一些东西不至于被卡死。

链路追踪与Zipkin

Zipkin是一个收集信息的工具，它需要单独启动，可以和数据库配合做所搜集信息的持久化，这里不在赘述。

在每一个微服务中，

- 添加依赖

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-sleuth</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-zipkin</artifactId>
        </dependency>
```

- 配置zipkin server所在的位置，默认是本地的9411

  Sleuth会将调用链路中的信息进行收集并且发送到zipkin，在Zipkin提供的面板中可以看到微服务之间的调用关系。