Spring Cloud 入门 之 Eureka 篇（一）

一、前言

Spring Cloud 是一系列框架的有序集合。它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。

本篇介绍 Spring Cloud 入门系列中的 Eureka，实现快速入门。

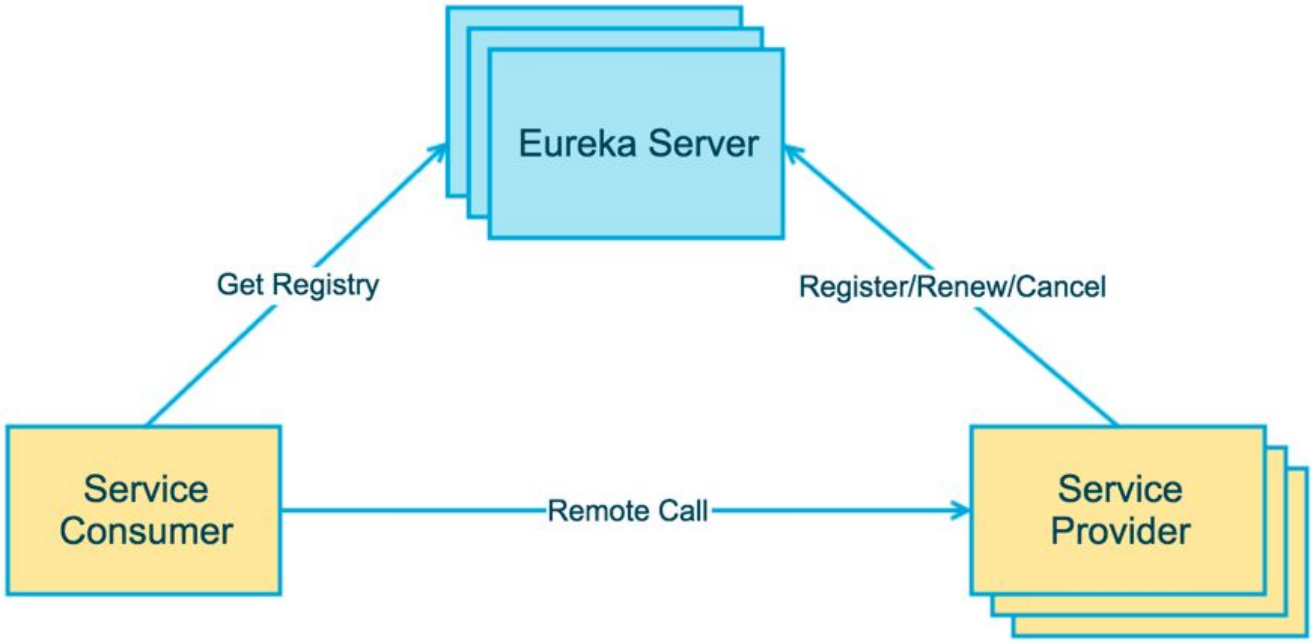
二、简单介绍

Eureka 是 Netflix 的子模块，它是一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

服务注册和发现对于微服务架构而言，是非常重要的。有了服务发现和注册，只需要使用服务的标识符就可以访问到服务，而不需要修改服务调用的配置文件。该功能类似于 Dubbo 的注册中心，比如 Zookeeper。

Eureka 采用了 CS 的设计架构。Eureka Server 作为服务注册功能的服务端，它是服务注册中心。而系统中其他微服务则使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。

其运行原理如下图：



由图可知，Eureka 的运行原理和 Dubbo 大同小异， Eureka 包含两个组件： Eureka Server 和 Eureka Client。

Eureka Server 提供服务的注册服务。各个服务节点启动后会在 Eureka Server 中注册服务，Eureka Server 中的服务注册表会存储所有可用的服务节点信息。

Eureka Client 是一个 Java 客户端，用于简化 Eureka Server 的交互，客户端同时也具备一个内置的、使用轮询负载算法的负载均衡器。在应用启动后，向 Eureka Server 发送心跳（默认周期 30 秒）。如果 Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，Eureka Server 会从服务注册表中将该服务节点信息移除。

三、搭建注册中心

创建 Spring Boot 项目，名为 eureka-server，进行如下操作：

3.1 添加依赖

```
<dependencyManagement>
  <dependencies>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Dalston.SR1</version>
                <type>pom</type>
                <scope>import</scope>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.9.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
        </dependency>
  </dependencies>
</dependencyManagement>

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

Spring Boot 与 **SpringCloud** 有版本兼容关系，如果引用版本不对应，项目启动会报错。

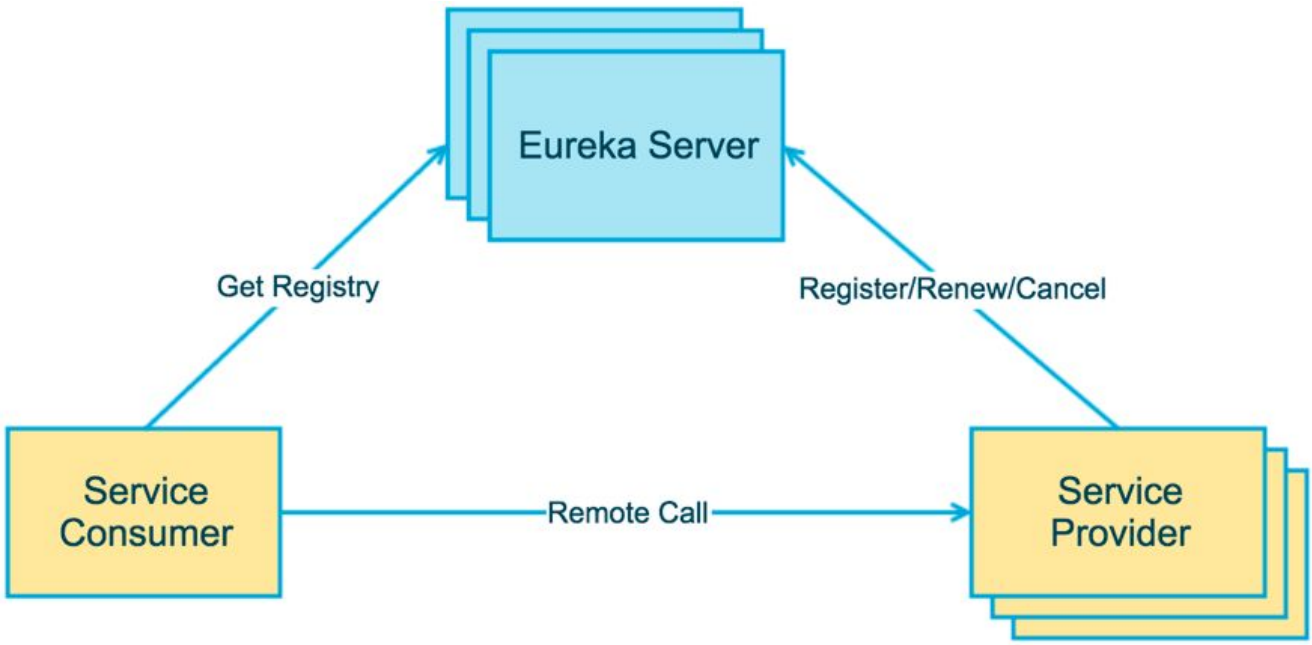
3.2 application.yml 配置参数

```
server:
    port: 9000

eureka:
    instance:
        hostname: localhost   # eureka 实例名称
    client:
        register-with-eureka: false # 不向注册中心注册自己
        fetch-registry: false       # 是否检索服务
        service-url:
            defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/  # 注册中心访问地
址
```

3.3 开启注册中心功能

在启动类上添加 **@EnableEurekaServer** 注解。

至此，准备工作完成，启动项目完成后，浏览器访问 http://localhost:9000 ，查看 Eureka 服务监控界面，如下图：



通过该网址可以查看注册中心注册服务的相关信息。当前还没有服务注册，因此没有服务信息。

补充：**http://localhost:9000** 是 **Eureka** 监管界面访问地址，
而 **http://localhost:9000/eureka/** **Eureka** 注册服务的地址。

四、实战演练

了解 Eureka 的环境搭建后，我们需要进行实战直观的感受 Eureka 的真正作用，这样才能清楚掌握和学习 Eureka 。

我们再创建两个 Spring Boot 项目，一个名为 user-api ，用于提供接口服务，另一个名为 user-web，用于调用 user-api 接口获取数据与浏览器交互。

服务实例	端口	描述
eureka	9000	注册中心（Eureka 服务端）
user-api	8081	服务提供者（Eureka 客户端）
user-web	80	服务消费者，与浏览器端交互（Eureka 客户端）

3.1 user-api 项目部分代码（服务提供）

由于文章内容针对 **Eureka** 的入门和使用，因此只张贴重要代码。具体代码请浏览下文提供的源码地址。

1. 添加依赖：

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

<!-- eureka 客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

1. 配置参数：

```
server:
    port: 8081

spring:
    application:
        name: user-api

eureka:
    instance:
        instance-id: user-api-8081
        prefer-ip-address: true # 访问路径可以显示 IP
    client:
        service-url:
            defaultZone: http://localhost:9000/eureka/  # 注册中心访问地址
```

注意：**http://localhost:9000/eureka/** 就是第三节中配置的注册中心的地址。

1. 服务接口：

```
public interface UserService {
    public User getById(Integer id);
}

@Service
public class UserServiceImpl implements UserService {

        private static Map<Integer,User> map;

        static {
                map = new HashMap<>();
                for (int i=1; i<6; i++) {
                        map.put(i, new User(i,"test" +i , "pwd" + i));
                }
        }

        @Override
        public User getById(Integer id) {
                return map.get(id);
        }

}
```

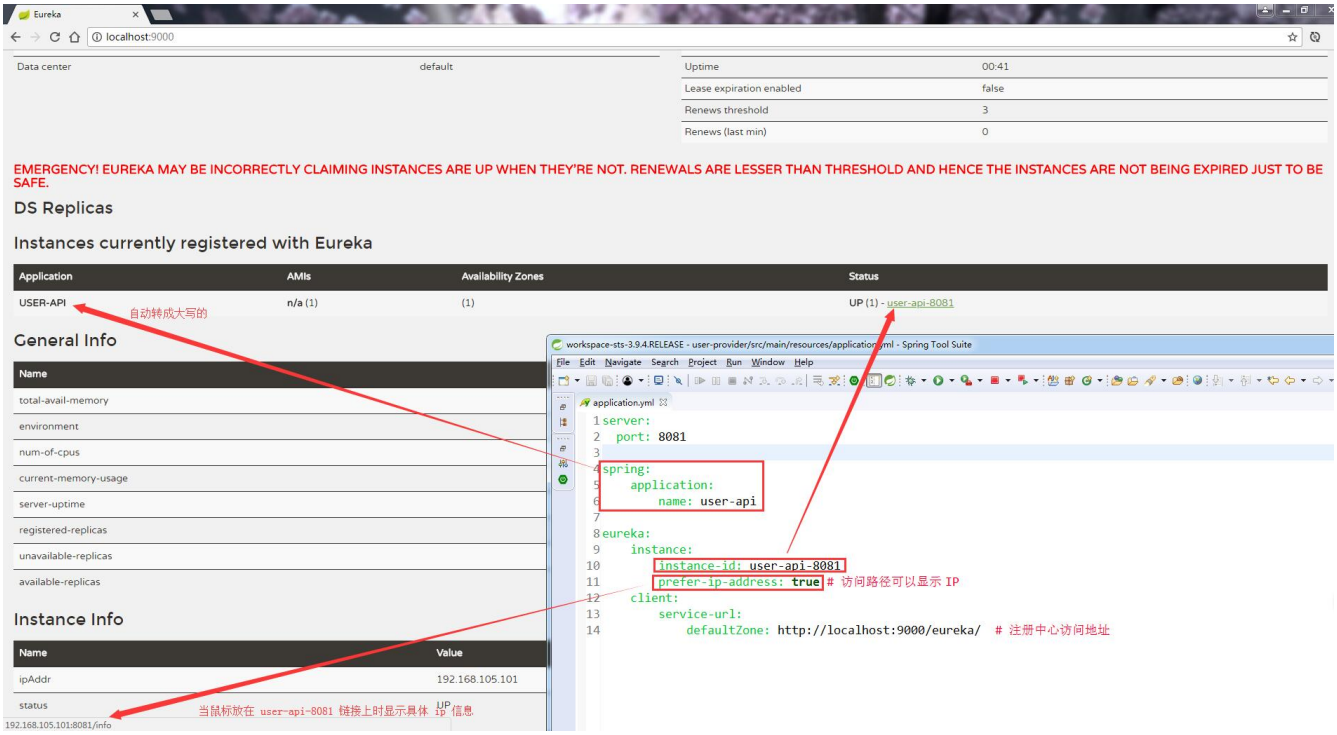
```
@RestController
@RequestMapping("/provider/user")
public class UserController {

        @Autowired
        private UserService userService;

        @RequestMapping("/get/{id}")
        public User get(@PathVariable("id") Integer id) {
                return this.userService.getById(id);
        }
}
```

注意：该 **controller** 是给 **user-web** 使用的（内部服务），不是给浏览器端调用的。

1. 开启服务注册功能：

在启动类上添加 **@EnableEurekaClient** 注解。

启动项目完成后，浏览器访问 http://localhost:9000 查看 Eureka 服务监控界面，如下图：



从图可知，user 相关服务信息已经注册到 Eureka 服务中了。

补充：在上图中，我们还看到一串红色的字体，那是因为 **Eureka** 启动了自我保护的机制。当 **EurekaServer** 在短时间内丢失过多客户端时（可能发生了网络故障），**EurekaServer** 将进入自我保护模式。进入该模式后，**EurekaServer** 会保护服务注册表中的信息不被删除。当网络故障恢复后，**EurekaServer** 会自动退出自我保护模式。

3.2 user-web 项目部分代码（服务消费）

1. 添加依赖：

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

<!-- eureka 客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

1. 配置参数：

```
server:
  port: 80

spring:
  application:
    name: user-web

eureka:
  client:
    register-with-eureka: false # 不向注册中心注册自己
    fetch-registry: true        # 是否检索服务
    service-url:
      defaultZone: http://localhost:9000/eureka/  # 注册中心访问地址
```

1. 客户端：

```
@Configuration
public class RestConfiguration {

        @Bean
        @LoadBalanced
        public RestTemplate getRestTemplate() {
                return new RestTemplate();
        }
}
```

```
@RestController
@RequestMapping("/user")
public class UserController {

        @Autowired
        private RestTemplate restTemplate;

//      @RequestMapping("get/{id}")
//      public User get(@PathVariable("id") Integer id) throws Exception {
//              // 没有使用 Eureka 时，uri 为消息提供者的地址，需要指定 ip 和
端口
//              return restTemplate.getForObject(new
URI("http://localhost:8081/provider/user/get/" + id), User.class);
//      }

    @Autowired
        private DiscoveryClient client;

        @RequestMapping("get/{id}")
        public User get(@PathVariable("id") Integer id) throws Exception {

                List<ServiceInstance> list =
this.client.getInstances("USER-API");
                String uri = "";
            for (ServiceInstance instance : list) {
                if (instance.getUri() != null &&
!"".equals(instance.getUri())) {
                        uri = instance.getUri().toString();
                        break;
                }
            }
                return restTemplate.getForObject(uri +
"/provider/user/get/" + id, User.class);
        }

}
```

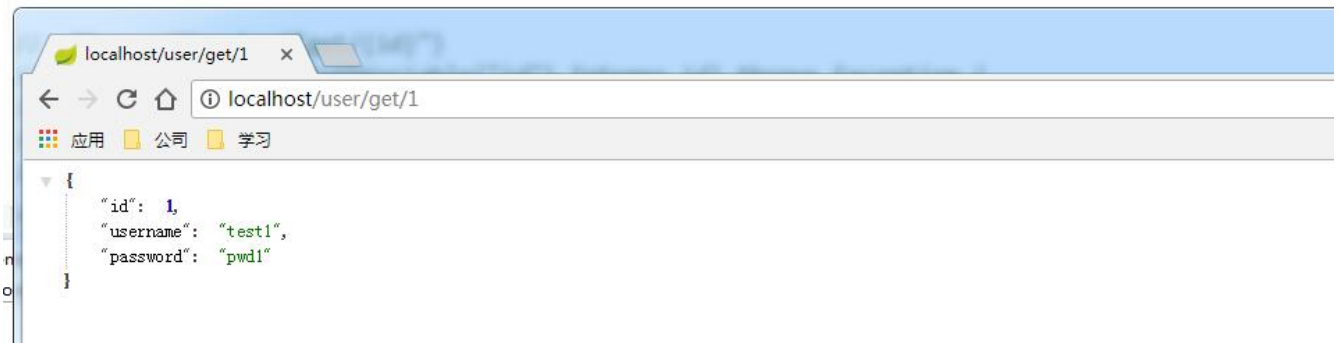
注意：此处只是为了体现服务发现的效果，实际开发中不使用 **DiscoveryClient** 查询服务进行调用！至于如何进行服务发现和调用请读者等待和浏览后续发布的 **Ribbon** 文章。

1. 开启服务发现功能：

在启动类上添加 **@EnableDiscoveryClient** 注解。

启动项目后，使用浏览器访问 user-web 项目接口，运行结果如下：

```
@Autowired
private DiscoveryClient client;

@RequestMapping("get/{id}")
public User get(@PathVariable("id") Integer id) throws Exception {

    List<ServiceInstance> list = this.client.getInstances("USER-API");
    String uri = "";
    for (ServiceInstance instance : list) {
        if (instance.getUri() != null && !"".equals(instance.getUri())) {
            uri = instance.getUri().toString();
            break;
        }
    }
    return restTemplate.getForObject(uri + "/provider/user/get/" + id, User.class);
}
```



至此，Eureka 的服务注册和发现演示完毕。

本系列的后续文章会基于该案例进行介绍和实战演练。

四、Eureka 集群

Eureka 作为注册中心，保存了系统服务的相关信息，如果注册中心挂掉，那么系统就瘫痪了。因此，对 Eureka 做集群实现高可用是必不可少的。

本次测试使用一台机器部署 Eureka 集群，通过名字和端口区分不同的 eureka 服务。

EUREKA 名称	端口号
eureka01	9001
eureka02	9002

1. 由于使用一台机器，使用两个名称还需要修改 C:\Windows\System32\drivers\etc 下的 host 文件，添加如下配置：

```
127.0.0.1  eureka01
127.0.0.1  eureka02
```

1. application.yml 文件需要进行如下修改：

```
server:
    port: 9001

eureka:
    instance:
        hostname: eureka01   # eureka 实例名称
    client:
        register-with-eureka: false # 不向注册中心注册自己
        fetch-registry: false       # 表示自己就是注册中心
        service-url:
            defaultZone:
http://eureka01:9001/eureka/,http://eureka02:9002/eureka/
```

两个 eureka 服务实例的配置文件修改方式类似，将名称和端口进行修改即可。

1. 服务注册的项目中，将 eureka.client.service-url.defaultZone 改成集群的 url 即可。

启动效果如下图：

!\](http://images.extlight.com/springcloud-eureka-05.jpg)

五、Eureka 与 Zookeeper 的区别

两者都可以充当注册中心的角色，且可以集群实现高可用，相当于小型的分布式存储系统。

5.1 CAP 理论

CAP 分别为 consistency(强一致性)、availability(可用性) 和 partition toleranc(分区容错性)。

理论核心：一个分布式系统不可能同时很好的满足一致性、可用性和分区容错性这三个需求。因此，根据 CAP 原理将 NoSQL 数据库分成满足 CA 原则、满足 CP 原则和满足 AP 原则三大类：

```
CA：单点集群，满足一致性，可用性的系统，通常在可扩展性上不高
CP：满足一致性，分区容错性的系统，通常性能不是特别高
AP：满足可用性，分区容错性的系统，通过对一致性要求较低
```

简单的说：CAP 理论描述在分布式存储系统中，最多只能满足两个需求。

5.2 Zookeeper 保证 CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟前的注册信息，但不能接受服务直接挂掉不可用了。因此，服务注册中心对可用性的要求高于一致性。

但是，zookeeper 会出现一种情况，当 master 节点因为网络故障与其他节点失去联系时，剩余节点会重新进行 leader 选举。问题在于，选举 leader 的时间较长，30 ~ 120 秒，且选举期间整个 zookeeper 集群是不可用的，这期间会导致注册服务瘫痪。在云部署的环境下，因网络问题导致 zookeeper 集群失去 master 节点的概率较大，虽然服务能最终恢复，但是漫长的选举时间导致注册服务长期不可用是不能容忍的。

5.3 Eureka 保证 AP

Eureka 在设计上优先保证了可用性。EurekaServer 各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和发现服务。

而 Eureka 客户端在向某个 EurekaServer 注册或发现连接失败时，会自动切换到其他 EurekaServer 节点，只要有一台 EurekaServer 正常运行，就能保证注册服务可用，只不过查询到的信息可能不是最新的。

除此之外，EurekaServer 还有一种自我保护机制，如果在 15 分钟内超过 85% 的节点都没有正常的心跳，那么 EurekaServer 将认为客户端与注册中心出现网络故障，此时会出现一下几种情况：

```
EurekaServer 不再从注册列表中移除因为长时间没有收到心跳而应该过期的服务

EurekaServer 仍然能够接收新服务的注册和查询请求，但不会被同步到其他节点上

当网络稳定时，当前 EurekaServer 节点新的注册信息会同步到其他节点中
```

因此，Eureka 可以很好的应对因网络故障导致部分节点失去联系的情况，而不会向 Zookeeper 那样是整个注册服务瘫痪。