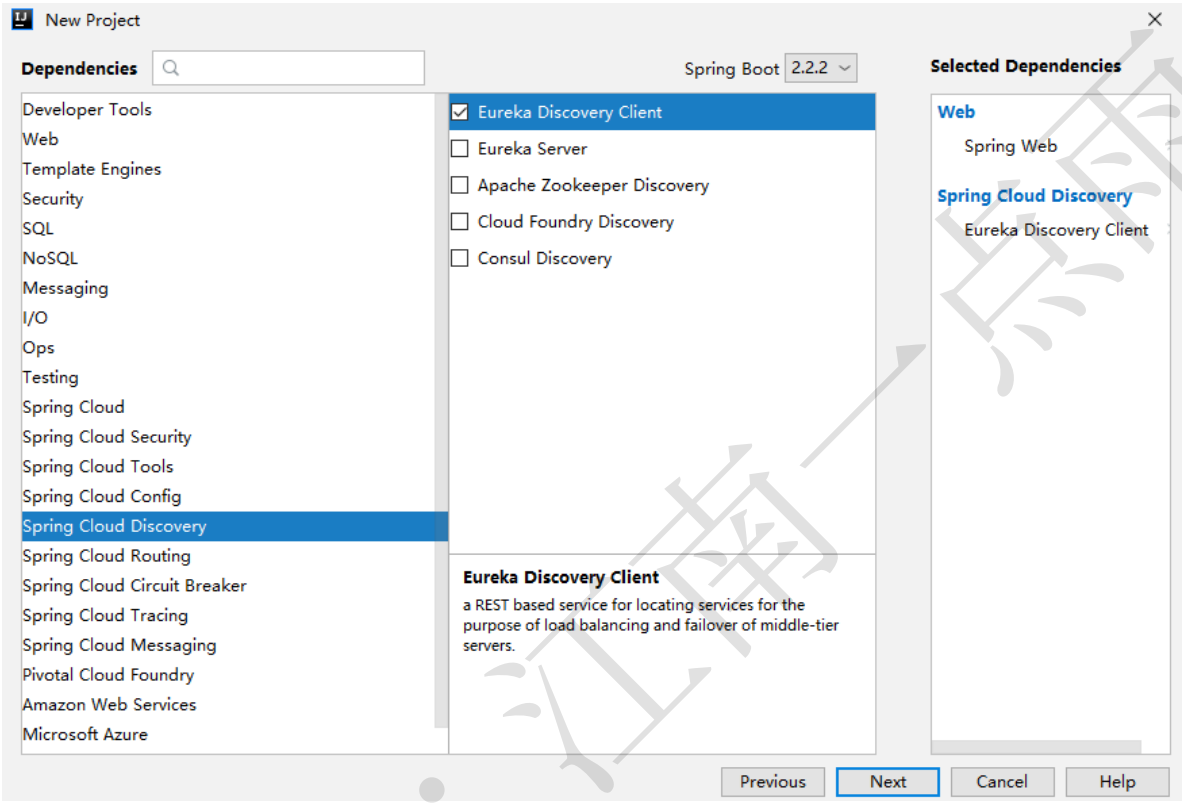
6.1 服务注册

服务注册就是把一个微服务注册到 Eureka Server 上，这样，当其他服务需要调用该服务时，只需要从 Eureka Server 上查询该服务的信息即可。

这里我们创建一个 provider，作为我们的服务提供者，创建项目时，选择 Eureka Client 依赖，这样，当服务创建成功后，简单配置一下，就可以被注册到 Eureka Server 上了：



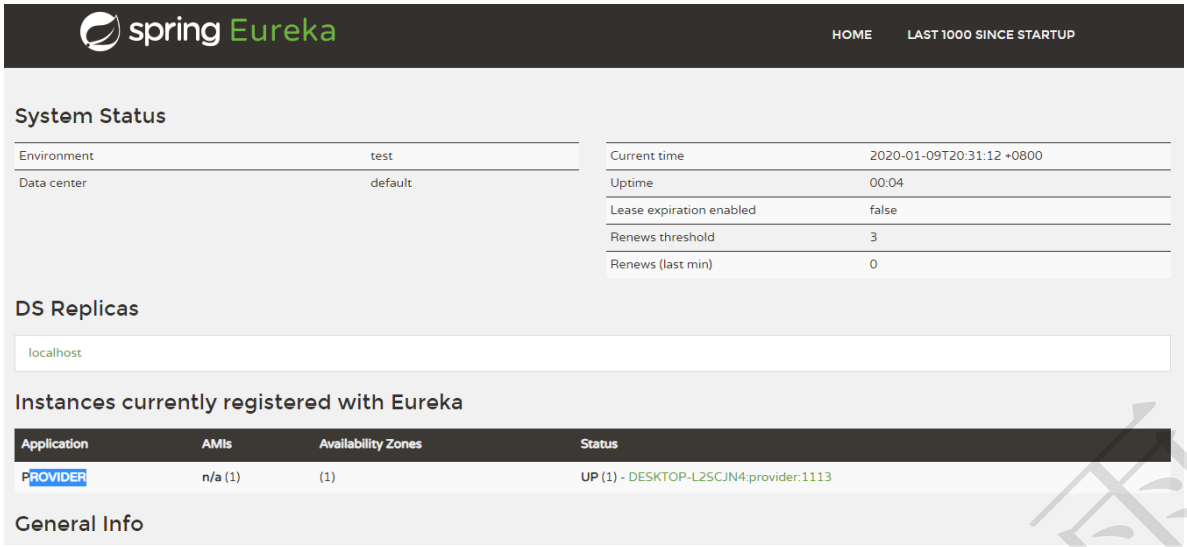
项目创建成功后，我们只需要在 application.properties 中配置一下项目的注册地址即可。注册地址的配置，和 Eureka Server 集群的配置很像。配置如下：

```
spring.application.name=provider
server.port=1113
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

三行配置，分别表示当前服务的名称、端口号以及服务地址。

接下来，启动 Eureka Server，待服务注册中心启动成功后，再启动 provider。

两者都启动成功后，浏览器输入 http://localhost:1111，就可以查看 provider 的注册信息：



6.2 服务消费

6.2.1 基本用法

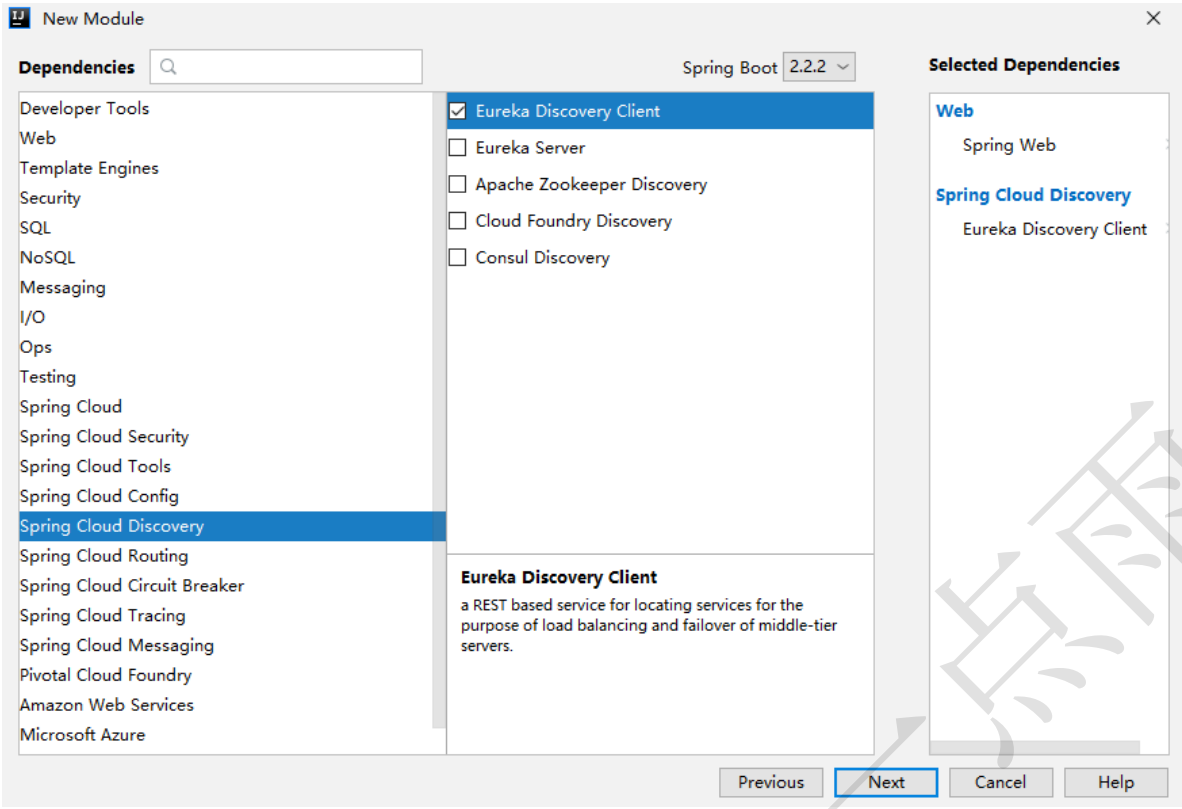
首先在 provider 中提供一个接口，然后创建一个新的 consumer 项目，消费这个接口。

在 provider 中，提供一个 hello 接口，如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello javaboy";
    }
}
```

接下来，创建一个 consumer 项目，consumer 项目中，去消费 provider 提供的接口。consumer 要能够获取到 provider 这个接口的地址，他就需要去 Eureka Server 中查询，如果直接在 consumer 中写死 provider 地址，意味着这两个服务之间的耦合度就太高了，我们要降低耦合度。首先我们来看一个写死的调用。

创建一个 consumer 项目，添加 web 和 eureka client 依赖：



创建完成后，我们首先也在 application.properties 中配置一下注册信息：

```
spring.application.name=consumer
server.port=1115
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

配置完成后，假设我们现在想在 consumer 中调用 provider 提供的服务，我们可以直接将调用写死，就是说，整个调用过程不会涉及到 Eureka Server。

```
@GetMapping("/hello1")
public String hello1() {
    HttpURLConnection con = null;
    try {
        URL url = new URL("http://localhost:1113/hello");
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

这是一段利用了 HttpUrlConnection 来发起的请求，请求中 provider 的地址写死了，意味着 provider 和 consumer 高度绑定在一起，这个不符合微服务的思想。

要改造它，我们可以借助 Eureka Client 提供的 DiscoveryClient 工具，利用这个工具，我们可以根据服务名从 Eureka Server 上查询到一个服务的详细信息，改造后的代码如下：

```
@Autowired
DiscoveryClient discoveryClient;
@GetMapping("/hello2")
public String hello2() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
            .append(host)
            .append(":")
            .append(port)
            .append("/hello");
    HttpURLConnection con = null;
    try {
        URL url = new URL(sb.toString());
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

注意，DiscoveryClient 查询到的服务列表是一个集合，因为服务在部署的过程中，可能是集群化部署，集合中的每一项就是一个实例。

这里我们可以稍微展示一下集群化部署。

首先，修改 provider 中的 hello 接口：

```
@RestController
public class HelloController {
    @Value("${server.port}")
    Integer port;
    @GetMapping("/hello")
    public String hello() {
        return "hello javaboy:" + port;
    }
}
```

因为我一会会启动多个 provider 实例，多个 provider 实例的端口不同，为了区分调用时到底是哪一个 provider 提供的服务，这里在接口返回值中返回端口。

修改完成后，对 provider 进行打包。provider 打包成功之后，我们在命令行启动两个 provider 实例：

```
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=1113
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=1116
```

启动完成后，检查 Eureka Server 上，这两个 provider 是否成功注册上来。

注册成功后，在 consumer 中再去调用 provider，DiscoveryClient 集合中，获取到的就不是一个实例了，而是两个实例。这里我们可以手动实现一个负载均衡：

```
int count = 0;
@GetMapping("/hello3")
public String hello3() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get((count++) % list.size());
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
            .append(host)
            .append(":")
            .append(port)
            .append("/hello");
    HttpURLConnection con = null;
    try {
        URL url = new URL(sb.toString());
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

在从集合中，获取数据时，通过一个小小举动，就可以实现线性负载均衡。

6.2.2 升级改造

从两个方面进行改造：

1. Http 调用
2. 负载均衡

Http 调用，我们使用 Spring 提供的 RestTemplate 来实现。

首先，在当前服务中，提供一个 RestTemplate 的实例：

```
@SpringBootApplication
public class ConsumerApplication {

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

    @Bean
    RestTemplate restTemplateOne() {
        return new RestTemplate();
    }

}
```

然后，在 Http 调用时，不再使用 HttpUrlConnection，而是直接使用 RestTemplate：

```
@Autowired
RestTemplate restTemplate;
@Autowired
DiscoveryClient discoveryClient;
@GetMapping("/hello2")
public String hello2() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
            .append(host)
            .append(":")
            .append(port)
            .append("/hello");
    String s = restTemplate.getForObject(sb.toString(), String.class);
    return s;
}
```

用 RestTemplate，一行代码就实现了 Http 调用。

接下来，使用 Ribbon 来快速实现负载均衡。

首先，我们需要给 RestTemplate 实例添加一个 @LoadBalanced 注解，开启负载均衡：

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

此时的 RestTemplate 就自动具备了负载均衡的功能。

此时的调用代码如下：

```
@Autowired
@Qualifier("restTemplate")
RestTemplate restTemplate;
@GetMapping("/hello3")
public String hello3() {
    return restTemplate.getForObject("http://provider/hello", String.class);
}
```

Java 中关于 Http 请求的工具实际上非常多，自带的 HttpUrlConnection，古老的 HttpClient，后起之秀 OkHttp 等，除了这些之外，还有一个好用的工具--RestTemplate，这是 Spring 中就开始提供的 Http 请求工具，不过很多小伙伴们可能是因为 Spring Cloud 才听说它。今天我们就来聊一聊这个 RestTemplate。

6.3 RestTemplate

RestTemplate 是从 Spring3.0 开始支持的一个 Http 请求工具，这个请求工具和 Spring Boot 无关，更和 Spring Cloud 无关。RestTemplate 提供了常见的 REST 请求方法模板，例如 GET、POST、PUT、DELETE 请求以及一些通用的请求执行方法 exchange 和 execute 方法。

RestTemplate 本身实现了 RestOperations 接口，而在 RestOperations 接口中，定义了常见的 RESTful 操作，这些操作在 RestTemplate 中都得到了很好的实现。

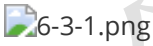
6.3.1 GET

首先我们在 provider 中定义一个 hello2 接口：

```
@GetMapping("/hello2")
public String hello2(String name) {
    return "hello " + name;
}
```

接下来，我们在 consumer 去访问这个接口，这个接口是一个 GET 请求，所以，访问方式，就是调用 RestTemplate 中的 GET 请求。

可以看到，在 RestTemplate 中，关于 GET 请求，一共有如下两大类方法：


6-3-1.png

这两大类方法实际上是重载的，唯一不同的，就是返回值类型。

getForObject 返回的是一个对象，这个对象就是服务端返回的具体值。getForEntity 返回的是一个 ResponseEntity，这个ResponseEntity 中除了服务端返回的具体数据外，还保留了 Http 响应头的数据。

```
@GetMapping("/hello4")
public void hello4() {
    String s1 = restTemplate.getForObject("http://provider/hello2?name={1}",
String.class, "javaboy");
    System.out.println(s1);
    ResponseEntity<String> responseEntity =
restTemplate.getForEntity("http://provider/hello2", String.class, "javaboy");
    String body = responseEntity.getBody();
    System.out.println("body:"+body);
    HttpStatus statusCode = responseEntity.getStatusCode();
    System.out.println("HttpStatus:"+statusCode);
```

```
    int statusCodeValue = responseEntity.getStatusCodeValue();
    System.out.println("statusCodeValue:"+statusCodeValue);
    HttpHeaders headers = responseEntity.getHeaders();
    Set<String> keySet = headers.keySet();
    System.out.println("--------------header-----------");
    for (String s : keySet) {
        System.out.println(s+":"+headers.get(s));
    }
}
```

这里大家可以看到，getForObject 直接拿到了服务的返回值，getForEntity 不仅仅拿到服务的返回值，还拿到 http 响应的状态码。然后，启动 Eureka Server、provider 以及 consumer ，访问 consumer 中的 hello4 接口，既可以看到请求结果。

看清楚两者的区别之后，接下来看下两个各自的重载方法，getForObject 和 getForEntity 分别有三个重载方法，两者的三个重载方法基本都是一致的。所以，这里，我们主要看其中一种。三个重载方法，其实代表了三种不同的传参方式。

```
@GetMapping("/hello5")
public void hello5() throws UnsupportedEncodingException {
    String s1 = restTemplate.getForObject("http://provider/hello2?name={1}",
String.class, "javaboy");
    System.out.println(s1);
    Map<String, Object> map = new HashMap<>();
    map.put("name", "zhangsan");
    s1 = restTemplate.getForObject("http://provider/hello2?name={name}",
String.class, map);
    System.out.println(s1);
    String url = "http://provider/hello2?name=" + URLEncoder.encode("张三", "UTF-
8");
    URI uri = URI.create(url);
    s1 = restTemplate.getForObject(uri, String.class);
    System.out.println(s1);
}
```

这就是我们说的三种不同的传参方式。

6.3.2 POST

首先在 provider 中提供两个 POST 接口，同时，因为 POST 请求可能需要传递 JSON，所以，这里我们创建一个普通的 Maven 项目作为 commons 模块，然后这个 commons 模块被 provider 和 consumer 共同引用，这样我们就可以方便的传递 JSON 了。

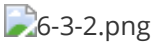
commons 模块创建成功后，首先在 commons 模块中添加 User 对象，然后该模块分别被 provider 和 consumer 引用。

然后，我们在 provider 中，提供和两个 POST 接口：

```
@PostMapping("/user1")
public User addUser1(User user) {
    return user;
}
@PostMapping("/user2")
public User addUser2(@RequestBody User user) {
    return user;
}
```

这里定义了两个 User 添加的方法，两个方法代表了两种不同的传参方式。第一种方法是以 key/value 形式来传参，第二种方法是以 JSON 形式来传参。

定义完成后，接下来，我们在 consumer 中调用这两个 POST 接口。



可以看到，这里的 post 和前面的 get 非常像，只是多出来了三个方法，就是 postForLocation，另外两个 postForObject 和 postForEntiy 和前面 get 基本一致，所以这里我们主要来看 postForObject，看完之后，我们再来看这个额外的 postForLocation。

```
@GetMapping("/hello6")
public void hello6() {
    MultiValueMap<String, Object> map = new LinkedMultiValueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    User user = restTemplate.postForObject("http://provider/user1", map,
User.class);
    System.out.println(user);
    user.setId(98);
    user = restTemplate.postForObject("http://provider/user2", user,
User.class);
    System.out.println(user);
}
```

post 参数到底是 key/value 形式还是 json 形式，主要看第二个参数，如果第二个参数是 MultiValueMap ，则参数是以 key/value 形式来传递的，如果是一个普通对象，则参数是以 json 形式来传递的。

最后再看看一下 postForLocation 。有的时候，当我执行完一个 post 请求之后，立马要进行重定向，一个非常常见的场景就是注册，注册是一个 post 请求，注册完成之后，立马重定向到登录页面去登录。对于这种场景，我们就可以使用 postForLocation。

首先我们在 provider 上提供一个用户注册接口：

```
@Controller
public class RegisterController {
    @PostMapping("/register")
    public String register(User user) {
        return "redirect:http://provider/loginPage?username=" +
user.getUsername();
    }

    @GetMapping("/loginPage")
    @ResponseBody
    public String loginPage(String username) {
        return "loginPage:" + username;
    }
}
```

注意，这里的 post 接口，响应一定是 302，否则 postForLocation 无效。

注意，重定向的地址，一定要写成绝对路径，不要写相对路径，否则在 consumer 中调用时会出问题

```
@GetMapping("/hello7")
public void hello7() {
    MultiValueMap<String, Object> map = new LinkedMultiValueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    URI uri = restTemplate.postForLocation("http://provider/register", map);
    String s = restTemplate.getForObject(uri, String.class);
    System.out.println(s);
}
```

这就是 postForLocation，调用该方法返回的是一个 Uri，这个 Uri 就是重定向的地址（里边也包含了重定向的参数），拿到 Uri 之后，就可以直接发送新的请求了。

6.3.3 PUT

PUT 请求比较简单，重载的方法也比较少。

我们首先在 provider 中提供一个 PUT 接口：

```
@PutMapping("/user1")
public void updateUser1(User user) {
    System.out.println(user);
}
@PutMapping("/user2")
public void updateUser2(@RequestBody User user) {
    System.out.println(user);
}
```

注意，PUT 接口传参其实和 POST 很像，也接受两种类型的参数，key/value 形式以及 JSON 形式。

在 consumer 中，我们来调用该接口：

```
@GetMapping("/hello8")
public void hello8() {
    MultiValueMap<String, Object> map = new LinkedMultiValueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    restTemplate.put("http://provider/user1", map);
    User user = new User();
    user.setId(98);
    user.setUsername("zhangsan");
    user.setPassword("456");
    restTemplate.put("http://provider/user1", user);
}
```

consumer 中的写法基本和 post 类似，也是两种方式，可以传递两种不同类型的参数。

6.3.4 DELETE

DELETE 也比较容易，我们有两种方式来传递参数，key/value 形式或者 PathVariable（参数放在路径中），首先我们在 provider 中定义两个 DELETE 方法：

```
@DeleteMapping("/user1")
public void deleteUser1(Integer id) {
    System.out.println(id);
}
@DeleteMapping("/user2/{id}")
public void deleteUser2(@PathVariable Integer id) {
    System.out.println(id);
}
```

然后在 consumer 中调用这两个删除的接口：

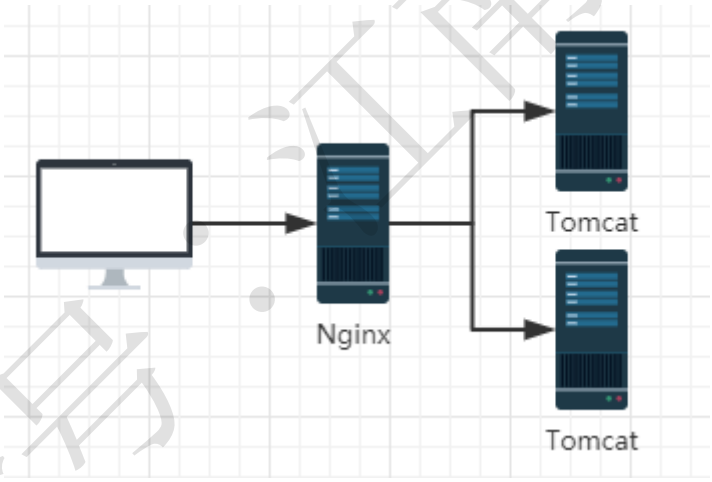
```
@GetMapping("/hello9")
public void hello9() {
    restTemplate.delete("http://provider/user1?id={1}", 99);
    restTemplate.delete("http://provider/user2/{1}", 99);
}
```

delete 中参数的传递，也支持 map，这块实际上和 get 是一样的。

6.4 客户端负载均衡

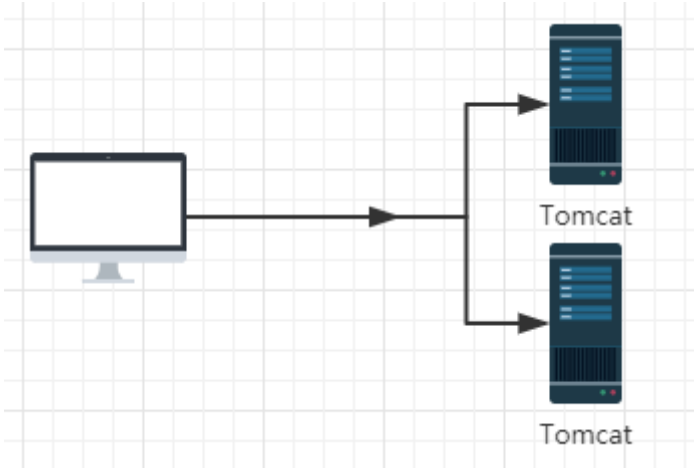
客户端负载均衡就是相对服务端负载均衡而言的。

服务端负载均衡，就是传统的 Nginx 的方式，用 Nginx 做负载均衡，我们称之为服务端负载均衡：



这种负载均衡，我们称之为服务端负载均衡，它的一个特点是，就是调用的客户端并不知道具体是哪一个 Server 提供的服务，它也不关心，反正请求发送给 Nginx，Nginx 再将请求转发给 Tomcat，客户端只需要记着 Nginx 的地址即可。

客户端负载均衡则是另外一种情形：



客户端负载均衡，就是调用的客户端本身是知道所有 Server 的详细信息的，当需要调用 Server 上的接口的时候，客户端从自身所维护的 Server 列表中，根据提前配置好的负载均衡策略，自己挑选一个 Server 来调用，此时，客户端知道它所调用的是哪一个 Server。

在 RestTemplate 中，要想使用负载均衡功能，只需要给 RestTemplate 实例上添加一个 @LoadBalanced 注解即可，此时，RestTemplate 就会自动具备负载均衡功能，这个负载均衡就是客户端负载均衡。

6.5 负载均衡原理

在 Spring Cloud 中，实现负载均衡非常容易，只需要添加 @LoadBalanced 注解即可。只要添加了该注解，一个原本普普通通做 Rest 请求的工具 RestTemplate 就会自动具备负载均衡功能，这个是怎么实现的呢?

整体上来说，这个功能的实现就是三个核心点:

1. 从 Eureka Client 本地缓存的服务注册信息中，选择一个可以调用的服务
2. 根据 1 中所选择的服务，重构请求 URL 地址
3. 将 1、2 步的功能嵌入到 RestTemplate 中