12 RestTemplate 用法详解

更新时间：2019-06-19 17:56:05

> 学习要注意到细处，不是粗枝大叶的，这样可以逐步学习、摸索，找到客观规律。
>
> ——徐特立

上篇文章带大家学习了一下基本的微服务环境搭建，由 provider 提供服务， consumer 通过 DiscoveryClient 先去 eureka 上获取 provider 的服务的地址，获取到地址之后再去调用相关的服务。在服务的调用过程中，使用到了一个工具，叫做 RestTemplate，RestTemplate 是由 Spring 提供的一个 HTTP 请求工具。在上文的案例中，开发者也可以不使用 RestTemplate ，使用 Java 自带的 HttpUrlConnection 或者经典的网络访问框架 HttpClient 也可以完成上文的案例，只是在 Spring 项目中，使用 RestTemplate 显然更方便一些。在传统的项目架构中，因为不涉及到服务之间的调用，大家对 RestTemplate 的使用可能比较少，因此，本文我们就先来带领大家来学习下 RestTemplate 的各种不同用法，只有掌握了这些用法，才能在微服务调用中随心所欲地发送请求。

RestTemplate 简介

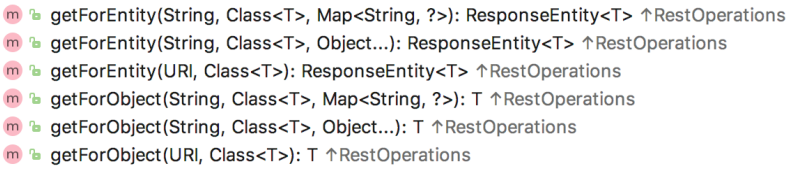
RestTemplate 是从 Spring3.0 开始支持的一个 HTTP 请求工具，它提供了常见的REST请求方案的模版，例如 GET 请求、POST 请求、PUT 请求、DELETE 请求以及一些通用的请求执行方法 exchange 以及 execute。RestTemplate 继承自 InterceptingHttpAccessor 并且实现了 RestOperations 接口，其中 RestOperations 接口定义了基本的 RESTful 操作，这些操作在 RestTemplate 中都得到了实现。接下来我们就来看看这些操作方法的使用。

用法实战

在开始下面的案例之前，我们需要先创建一个工程，命名为 RestTemplate 。和上篇文章的项目结构一样，在 RestTemplate 中，我们也分别创建子项目 eureka 、provider 以及 consumer ，将 provider 和 consumer 分别注册到 eureka 上面去。这个具体的步骤大家可以参考上篇文章，本文我就不赘述了，这是我们的准备工作。

GET 请求

做好了准备工作，先来看使用 RestTemplate 发送 GET 请求。在 RestTemplate 中，和 GET 请求相关的方法有如下几个：



这里的方法一共有两类，getForEntity 和 getForObject，每一类有三个重载方法，下面我们分别予以介绍。

getForEntity

既然 RestTemplate 发送的是 HTTP 请求，那么在响应的数据中必然也有响应头，如果开发者需要获取响应头的话，那么就需要使用 getForEntity 来发送 HTTP 请求，此时返回的对象是一个 ResponseEntity 的实例。这个实例中包含了响应数据以及响应头。例如，在 provider 中提供一个 HelloController 接口，HelloController 接口中定义一个 sayHello 的方法，如下：

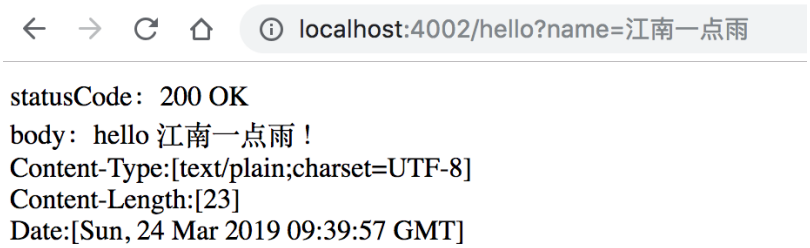
```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello(String name) {
        return "hello " + name + " !";
    }
}
```

在 consumer 中定义一个 UseHelloController 的类，再定义一个 `/hello` 接口，在接口中调用 provider 提供的服务，如下：

```
@RestController
public class UseHelloController {
    @Autowired
    DiscoveryClient discoveryClient;
    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/hello")
    public String hello(String name) {
        List<ServiceInstance> list = discoveryClient.getInstances("provider");
        ServiceInstance instance = list.get(0);
        String host = instance.getHost();
        int port = instance.getPort();
        String url = "http://" + host + ":" + port + "/hello?name={1}";
        ResponseEntity<String> responseEntity = restTemplate.getForEntity(url, String.class, name);
        StringBuffer sb = new StringBuffer();
        HttpStatus statusCode = responseEntity.getStatusCode();
        String body = responseEntity.getBody();
        sb.append("statusCode：")
                .append(statusCode)
                .append("</br>")
                .append("body：")
                .append(body)
                .append("</br>");
        HttpHeaders headers = responseEntity.getHeaders();
        Set<String> keySet = headers.keySet();
        for (String s : keySet) {
            sb.append(s)
                    .append(":")
                    .append(headers.get(s))
                    .append("</br>");
        }
        return sb.toString();
    }
}
```


关于 DiscoveryClient 那一段本文先不做讨论，主要来看 getForEntity 方法。第一个参数是 url，url 中有一个占位符 {1} ,如果有多个占位符分别用 {2} 、{3} … 去表示，第二个参数是接口返回的数据类型，最后是一个可变长度的参数，用来给占位符填值。在返回的 ResponseEntity 中，可以获取响应头中的信息，其中 getStatusCode 方法用来获取响应状态码， getBody 方法用来获取响应数据， getHeaders 方法用来获取响应头，在浏览器中访问该接口，结果如下：



当然，这里参数的传递除了这一种方式之外，还有另外两种方式，也就是 getForEntity 方法的另外两个重载方法。

第一个是占位符不使用数字，而是使用参数的 key，同时将参数放入到一个 map 中。map 中的 key 和占位符的 key 相对应，map 中的 value 就是参数的具体值，例如还是上面的请求，利用 map 来传递参数，请求方式如下：

```
Map<String, Object> map = new HashMap<>();
String url = "http://" + host + ":" + port + "/hello?name={name}";
map.put("name", name);
ResponseEntity<String> responseEntity = restTemplate.getForEntity(url, String.class, map);
```

这种方式传参可能看起来更直观一些。

第二个是使用 Uri 对象，使用 Uri 对象时，参数可以直接拼接在地址中，例如下面这样：

```
String url = "http://" + host + ":" + port + "/hello?name="+ URLEncoder.encode(name,"UTF-8");
URI uri = URI.create(url);
ResponseEntity<String> responseEntity = restTemplate.getForEntity(uri, String.class);
```

但需要注意的是，这种传参方式，参数如果是中文的话，需要对参数进行编码，使用 `URLEncoder.encode` 方法来实现。

getForObject

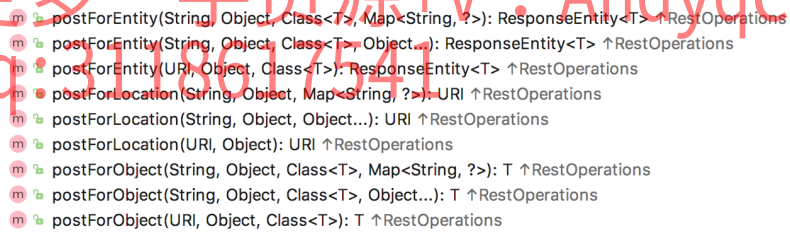
getForObject 方法和 getForEntity 方法类似，getForObject 方法也有三个重载的方法，参数和 getForEntity 一样，因此这里我就不重复介绍参数了，这里主要说下 getForObject 和 getForEntity 的差异，这两个的差异主要体现在返回值的差异上， getForObject 的返回值就是服务提供者返回的数据，使用 getForObject 无法获取到响应头。例如，还是上面的请求，利用 getForObject 来发送 HTTP 请求，结果如下：

```
String url = "http://" + host + ":" + port + "/hello?name=" + URLEncoder.encode(name, "UTF-8");
URI uri = URI.create(url);
String s = restTemplate.getForObject(uri, String.class);
```

注意，这里返回的 s 就是 provider 的返回值，如果开发者只关心 provider 的返回值，并不关系 HTTP 请求的响应头，那么可以使用该方法。

POST 请求

和 GET 请求相比，RestTemplate 中的 POST 请求多了一个类型的方法，如下：



可以看到，post 请求的方法类型除了 postForEntity 和 postForObject 之外，还有一个 postForLocation。这里的方法类型虽然有三种，但是这三种方法重载的参数基本是一样的，因此这里我还是以 postForEntity 方法为例，来剖析三个重载方法的用法，最后再重点说下 postForLocation 方法。

postForEntity

在 POST 请求中，参数的传递可以是 key/value 的形式，也可以是 JSON 数据，分别来看：

1. 传递 key/value 形式的参数

首先在 provider 的 HelloController 类中再添加一个 POST 请求的接口，如下：

```
@PostMapping("/hello2")
public String sayHello2(String name) {
    return "Hello " + name + " !";
}
```

然后在 consumer 中添加相应的方法去访问，如下：

```
@GetMapping("/hello5")
public String hello5(String name) {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/hello2";
    MultiValueMap map = new LinkedMultiValueMap();
    map.add("name", name);
    ResponseEntity<String> responseEntity = restTemplate.postForEntity(url, map, String.class);
    return responseEntity.getBody();
}
```

在这里， postForEntity 方法第一个参数是请求地址，第二个参数 map 对象中存放着请求参数 key/value，第三个参数则是返回的数据类型。当然这里的第一个参数 url 地址也可以换成一个 Uri 对象，效果是一样的。这种方式传递的参数是以 key/value 形式传递的，在 post 请求中，也可以按照 get 请求的方式去传递 key/value 形式的参数，传递方式和 get 请求的传参方式基本一致，例如下面这样：

```
@GetMapping("/hello6")
public String hello6(String name) {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/hello2?name={1}";
    ResponseEntity<String> responseEntity = restTemplate.postForEntity(url, null, String.class,name);
    return responseEntity.getBody();
}
```

此时第二个参数可以直接传一个 null。

2. 传递 JSON 数据

上面介绍的是 post 请求传递 key/value 形式的参数，post 请求也可以直接传递 json 数据，在 post 请求中，可以自动将一个对象转换成 json 进行传输，数据到达 provider 之后，再被转换为一个对象。具体操作步骤如下：

首先在 RestTemplate 项目中创建一个新的maven项目，叫做 commons ，然后在 commons 中创建一个 User 对象，如下：

```
public class User {
    private String username;
    private String address;
    //省略getter/setter
}
```

然后分别在 provider 和 consumer 的 pom.xml 文件中添加对 commons 模块的依赖，如下：

```
<dependency>
    <groupId>com.justdojava</groupId>
    <artifactId>commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

这样，在 provider 和 consumer 中就都能使用 User 对象了。首先在 provider 中创建一个添加用户的接口，如下：

```
@Controller
@ResponseBody
public class UserController {
    @PostMapping("/user")
    public User hello(@RequestBody User user) {
        return user;
    }
}
```

这里的接口很简单，只需要将用户传来的 User 对象再原封不动地返回去就行了，然后在 consumer 中添加一个接口来测试这个接口，如下：

```
@GetMapping("/hello7")
public User hello7() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/user";
    User u1 = new User();
    u1.setUsername("牧码小子");
    u1.setAddress("深圳");
    ResponseEntity<User> responseEntity = restTemplate.postForEntity(url, u1, User.class);
    return responseEntity.getBody();
}
```

看到这段代码有人要问了，这不和前面的一样吗？是的，唯一的区别就是第二个参数的类型不同，这个参数如果是一个 MultiValueMap 的实例，则以 key/value 的形式发送，如果是一个普通对象，则会被转成 json 发送。

postForObject

postForObject 和 postForEntity 基本一致，就是返回类型不同而已，这里不再赘述。

postForLocation

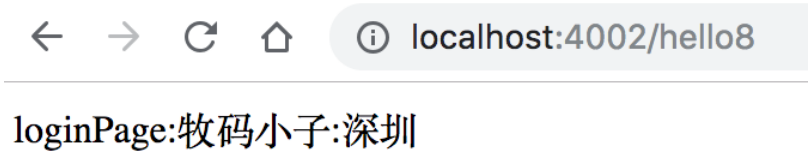
postForLocation 方法的返回值是一个 Uri 对象，因为 POST 请求一般用来添加数据，有的时候需要将刚刚添加成功的数据的 URL 返回来，此时就可以使用这个方法，一个常见的使用场景如用户注册功能，用户注册成功之后，可能就自动跳转到登录页面了，此时就可以使用该方法。例如在 provider 中提供一个用户注册接口，再提供一个用户登录接口，如下：

```
@RequestMapping("/register")
public String register(User user) throws UnsupportedEncodingException {
    return "redirect:/loginPage?username=" + URLEncoder.encode(user.getUsername(),"UTF-8") + "&address=" + URLEncoder.encode(user.getAddres
s(),"UTF-8");
}
@GetMapping("/loginPage")
@ResponseBody
public String loginPage(User user) {
    return "loginPage:" + user.getUsername() + ":" + user.getAddress();
}
```

这里一个注册接口，一个是登录页面，不过这里的登录页面我就简单用一个字符串代替了。然后在 consumer 中来调用注册接口，如下：

```
@GetMapping("/hello8")
public String hello8() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/register";
    MultiValueMap map = new LinkedMultiValueMap();
    map.add("username", "牧码小子");
    map.add("address", "深圳");
    URI uri = restTemplate.postForLocation(url, map);
    String s = restTemplate.getForObject(uri, String.class);
    return s;
}
```

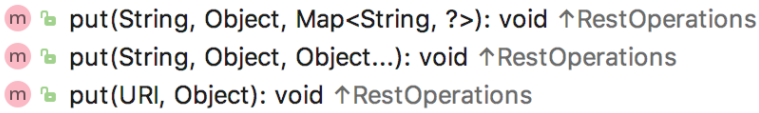
这里首先调用 postForLocation 获取 Uri 地址，然后再利用 getForObject 请求 Uri，结果如下：



注意：**postForLocation** 方法返回的 **Uri** 实际上是指响应头的 **Location** 字段，所以，**provider** 中 **register** 接口的响应头必须要有 **Location** 字段（即请求的接口实际上是一个重定向的接口），否则 **postForLocation** 方法的返回值为**null**，初学者很容易犯这个错误，如果这里出错，大家可以参考下我的源代码。

PUT 请求

只要将 GET 请求和 POST 请求搞定了，接下来 PUT 请求就会容易很多了，PUT 请求本身方法也比较少，只有三个，如下：



这三个重载的方法其参数其实和 POST 是一样的，可以用 key/value 的形式传参，也可以用 JSON 的形式传参，无论哪种方式，都是没有返回值的，我这里就举两个例子给大家参考下：

首先在 provider 的 UserController 中添加如下两个数据更新接口：

```
@PutMapping("/user/name")
@ResponseBody
public void updateUserByUsername(User User) {
    System.out.println(User);
}
@PutMapping("/user/address")
@ResponseBody
public void updateUserByAddress(@RequestBody User User) {
    System.out.println(User);
}
```

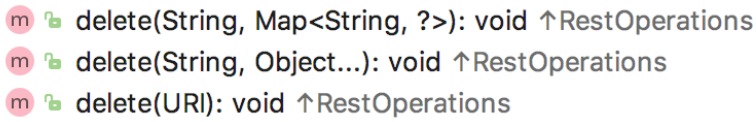
这里两个接口，一个接收 key/value 形式的参数，另一个接收 JSON 参数。因为这里没有返回值，我直接把数据打印出来就行了。接下来在 consumer 中添加接口调用这里的服务，如下：

```
@GetMapping("/hello9")
public void hello9() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url1 = "http://" + host + ":" + port + "/user/name";
    String url2 = "http://" + host + ":" + port + "/user/address";
    MultiValueMap map = new LinkedMultiValueMap();
    map.add("username", "牧码小子");
    map.add("address", "深圳");
    restTemplate.put(url1, map);
    User u1 = new User();
    u1.setAddress("广州");
    u1.setUsername("江南一点雨");
    restTemplate.put(url2, u1);
}
```

访问 /hello9 接口，即可看到 provider 上有日志打印出来，这里比较简单，我就不再演示。

DELETE 请求

和 PUT 请求一样，DELETE 请求也是比较简单的，只有三个方法，如下：



不同于 POST 和 PUT，DELETE 请求的参数只能在地址栏传送，可以是直接放在路径中，也可以用 key/value 的形式传递，当然，这里也是没有返回值的。我也举两个例子：

首先在 provider 的 UserController 中添加两个接口，如下：

```
@DeleteMapping("/user/{id}")
@ResponseBody
public void deleteUserById(@PathVariable Integer id) {
    System.out.println(id);
}
@DeleteMapping("/user/")
@ResponseBody
public void deleteUserByUsername(String username) {
    System.out.println(username);
}
```

两个接口，一个的参数在路径中，另一个的参数以 key/value 的形式传递，然后在 consumer 中，添加一个方法调用这两个接口，如下：

```
@GetMapping("/hello10")
public void hello10() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url1 = "http://" + host + ":" + port + "/user/{1}";
    String url2 = "http://" + host + ":" + port + "/user/?username={username}";
    Map<String,String> map = new HashMap<>();
    map.put("username", "牧码小子");
    restTemplate.delete(url1, 99);
    restTemplate.delete(url2, map);
}
```

这里参数的传递和 GET 请求基本一致，我就不再赘述了。

其他

设置请求头

有的时候我们会有一些特殊的需求，例如模拟 cookie ，此时就需要我们自定义请求头了。自定义请求头可以通过拦截器的方式来实现（下篇文章我们会详细的说这个拦截器）。定义拦截器、自动修改请求数据、一些身份认证信息等，都可以在拦截器中来统一处理。具体操作步骤如下：

首先在 provider 中定义一个接口，在接口中获取客户端传来的 cookie 数据，如下：

```
@GetMapping("/customheader")
public String customHeader(HttpServletRequest req) {
    return req.getHeader("cookie");
}
```

这里简单处理，将客户端传来的 cookie 拿出来后再返回给客户端，然后在 consumer 中添加如下接口来测试：

```
@GetMapping("/hello11")
public void hello11() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/customheader";
    restTemplate.setInterceptors(Collections.singletonList(new ClientHttpRequestInterceptor() {
        @Override
        public ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution execution) throws IOException {
            HttpHeaders headers = request.getHeaders();
            headers.add("cookie","justdojava");
            return execution.execute(request,body);
        }
    }));
    String s = restTemplate.getForObject(url, String.class);
    System.out.println(s);
}
```


这里通过调用 RestTemplate 的 setInterceptors 方法来给它设置拦截器，拦截器也可以有多个，我这里只有一个。在拦截器中，将请求拿出来，给它设置 cookie ，然后调用 execute 方法让请求继续执行。此时，在 `/customheader` 接口中，就能获取到 cookie了。

通用方法 exchange

在 RestTemplate 中还有一个通用的方法 exchange。为什么说它通用呢？因为这个方法需要你在调用的时候去指定请求类型，即它既能做 GET 请求，也能做 POST 请求，也能做其它各种类型的请求。如果开发者需要对请求进行封装，使用它再合适不过了，举个简单例子：

```java
@GetMapping("/hello12")
public void hello12() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    String url = "http://" + host + ":" + port + "/customheader";
    HttpHeaders headers = new HttpHeaders();
    headers.add("cookie","justdojava");
    HttpEntity<MultiValueMap<String,String>> request =  new HttpEntity<>(null,headers);
    ResponseEntity<String> responseEntity = restTemplate.exchange(url, HttpMethod.GET, request, String.class);
    System.out.println(responseEntity.getBody());
}
```

这里的参数和前面的也都差不多，注意就是多了一个请求类型的参数，然后创建一个 HttpEntity 作为参数来传递。HttpEntity 在创建时候需要传递两个参数，第一个上文给了一个 null ，这个参数实际上就相当于 POST/PUT 请求中的第二个参数，有需要可以自行定义。HttpEntity 创建时的第二个参数就是请求头了，也就是说，如果使用 exchange 来发送请求，可以直接定义请求头，而不需要使用拦截器。

小结

本文主要向大家介绍了 RestTemplate 这样一个 HTTP 请求工具类的常见用法，一些比较冷门的用法本文并未涉及，读者有兴趣可以自行查找资料学习。由于 Spring、SpringMVC、Spring Boot、Spring Cloud 这些家族成员一脉相承，因此在 SpringMVC 中支持良好的 RESTful 风格的接口在后续的各个组件中都继续支持，在微服务接口设计时，大部分接口也都满足 RESTful 风格，使用 RestTemplate 则可以非常方便地发送 RESTful 风格的请求，因此这个工具的使用是我们后面学习的基础，常见的用法一定要熟练掌握。

本文作者：纯洁的微笑、江南一点雨

11 服务注册与消费
13 服务请求负载均衡