### 什么是Rest

REST（RepresentationalStateTransfer）是RoyFielding提出的一个描述互联系统架构风格的名词。REST定义了一组体系架构原则，您可以根据这些原则设计以系统资源为中心的Web服务，包括使用不同语言编写的客户端如何通过HTTP处理和传输资源状态。

为什么称为REST？Web本质上由各种各样的资源组成，资源由URI唯一标识。浏览器将展示出该资源的一种表现方式，或者一种表现状态。如果用户在该页面中定向到指向其它资源的链接，则将访问该资源，并表现出它的状态。这意味着客户端应用程序随着每个资源表现状态的不同而发生状态转移，也即所谓REST。

附：[REST定义](https://www.ibm.com/developerworks/cn/webservices/ws-restful/)、[REST与SOAP的比较](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/index.html)

### REST成熟度的四个层次

* 第一个层次的Web服务只是使用HTTP作为传输方式，实际上只是远程方法调用（RPC）的一种具体形式。SOAP和XML-RPC都属于此类。
* 第二个层次的Web服务引入了资源的概念。每个资源有对应的标识符和表达。
* 第三个层次的Web服务使用不同的HTTP方法来进行不同的操作，并且使用HTTP状态码来表示不同的结果。如HTTPGET方法来获取资源，HTTPDELETE方法来删除资源。
* 第四个层次的Web服务使用HATEOAS。在资源的表达中包含了链接信息。客户端可以根据链接来发现可以执行的动作。

其中第三个层次建立了CRUD操作与HTTP方法之间的一对一映射。根据此映射：

（1）若要在服务器上创建资源，应该使用POST方法。

（2）若要检索某个资源，应该使用GET方法。

（3）若要更改资源状态或对其进行更新，应该使用PUT方法。

（4）若要删除某个资源，应该使用DELETE方法。

### HTTP请求的方法

（1）GET：通过请求URI得到资源

（2）POST：用于添加新的内容

（3）PUT：用于修改某个内容，若不存在则添加

（4）DELETE：删除某个内容

（5）OPTIONS：询问可以执行哪些方法

（6）HEAD：类似于GET，但是不返回body信息，用于检查对象是否存在，以及得到对象的元数据

（7）CONNECT：用于代理进行传输，如使用SSL

（8）TRACE：用于远程诊断服务器

### HTTP请求的常见状态码

（1）成功Successful2xx：此类状态码标识客户端的请求被成功接收、理解并接受。常见如200（OK）、204（NoContent）。

（2）重定向Redirection3xx：这个类别的状态码标识用户代理要做出进一步的动作来完成请求。常见如301（MovedPermanently）、302（MovedTemprarily）。

（3）客户端错误ClientError4xx：4xx类别的状态码是当客户端象是出错的时使用的。常见如400（BadRequest）、401（Unauthorized）、403（Forbidden）、404（NotFound)。

（4）服务器错误ServerError5xx：响应状态码以5开头表示服务器知道自己出错或者没有能力执行请求。常见如500（InternalServer Error）、502（BadGateway）、504（GatewayTimeout）。

HTTP1.0、HTTP1.1 和 HTTP2.0 的区别：

<https://www.cnblogs.com/heluan/p/8620312.html>

### RestTemplate

Spring官网对它的介绍如下：

[RestTemplate](https://docs.spring.io/spring/docs/5.1.9.RELEASE/spring-framework-reference/integration.html#rest-resttemplate):TheoriginalSpringRESTclientwithasynchronous,templatemethodAPI.

RestTemplate是一个同步的RestAPI客户端。

在项目中，当我们需要远程调用一个HTTP接口时，经常会用到RestTemplate这个类。这个类是Spring框架提供的一个工具类。简化了发起HTTP请求以及处理响应的过程，并且支持REST。为什么说简化了了呢？

1）使用java.net包下的URLConnection建立连接：

String result= "";

BufferedReaderin = null;

try {

String urlNameString= url +"?" + param;

URL realUrl= new URL(urlNameString);

// 打开和URL之间的连接

URLConnectionconnection = realUrl.openConnection();

// 设置通用的请求属性

connection.setRequestProperty("accept","\*/\*");

connection.setRequestProperty("connection","Keep-Alive");

connection.setRequestProperty("user-agent",

"Mozilla/4.0(compatible; MSIE 6.0; Windows NT 5.1;SV1)");

// 建立实际的连接

connection.connect();

// 获取所有响应头字段

Map<String,List<String>> map = connection.getHeaderFields();

// 遍历所有的响应头字段

for(String key : map.keySet()) {

System.out.println(key+ "--->" + map.get(key));

}

// 定义 BufferedReader输入流来读取URL的响应

in =new BufferedReader(newInputStreamReader(

connection.getInputStream()));

String line;

while ((line = in.readLine())!= null) {

result += line;

}

} catch (Exception e) {

…

}

// 使用finally块来关闭输入流

finally{

// 关闭流

}

2）使用RestTemplate

ResponseEntity<SsoUrlPrm>result =

restTemplate.getForEntity(requestPathUrl,SsoUrlPrm.class);

#### 对外开放的接口

|  |  |
| --- | --- |
| DELETE | delete |
| GET | getForObject |
| getForEntity |
| HEAD | headForHeaders |
| OPTIONS | optionsForAllow |
| POST | postForLocation |
| postForObject |
| PUT | put |
| any | exchange |
| execute |

使用RestTemplate来调用接口

#### 1、新建一个配置类，配置RestTemplate的Bean

/\*\*

\* RestTemplate配置模板

\*

\* @author like

\*/

@Configuration

public class RestTemplateConfig {

@Bean

public RestTemplate restTemplate(ClientHttpRequestFactory factory) {

return new RestTemplate(factory);

}

@Bean

public ClientHttpRequestFactory simpleClientHttpRequestFactory() {

SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();

factory.setReadTimeout(5000);//单位为ms

factory.setConnectTimeout(5000);//单位为ms

return factory;

}

}

注意：如果在编译器中提示factory不能做自动注入，那应该是跟其他类有冲突，有多个ClientHttpRequestFactory，把参数的名称嗨一下就行。

多种传输和接收参数的方式：

#### 2、postForObject方法

post请求，并返回一个Object对象。

第1个参数：请求的url地址

第2个参数：其实是HttpEntity，这个类主要有3种构造方法：

* new HttpEntity(请求体)
* new HttpEntity(请求头)
* new HttpEntity(请求体,请求头)

第3个参数：返回的结果类型，这里String.class表示返回结果是一个字符串。

第4个参数：参数值，这里有Map和可变参数两种形式（通常用不到，数据通常放在Json里就全部传输过去了）

使用json来传递和接收数据：

1）首先引入fastJson的pom依赖。

2）在实现类中注入RestTemplate。

3）创建一个ExpressionDomain对象，将这个对象转化成JSONObject。使用json来传递数据。

ExpressionDomain expressionDomain=new ExpressionDomain("hello","hasaki","win");

JSONObject jsonObj = (JSONObject) JSONObject.toJSON(expressionDomain);

//设置请求头

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON\_UTF8);

headers.add("Accept", MediaType.APPLICATION\_JSON.toString());

//请求体

HttpEntity<String> formEntity = new HttpEntity<String>(jsonObj.toString(), headers);

//发起请求

String jsonResult = restTemplate.postForObject("http://localhost:8081/findDataByReflection" , formEntity, String.class);

//将Json字符串解析成对象

Response resp = JSON.parseObject(jsonResult, new TypeReference<Response>() {});

}

接收端：

用@ResponseBody注解一个参数，用于自动解析Json为对象。返回的Response也是一个对象，添加@ResponseBody注解，将返回Json字符串。解析的时候将Json字符串解析成对象即可。



#### 3、postForEntity

和getForEntity原理是一样的，下面会讲到。

#### 4、getForObject()方法

get请求，并返回一个对象。这里有3种方法：

第1个参数：请求的url地址。

第2个参数：返回的结果类型，这里String.class表示返回结果是一个字符串。

第3个参数：参数值，Map和可变参数两种形式。

通过Map传参数的方式：

可以使用Map来封装请求参数，并作为getForObject的第三个参数，同时修改url如下，map中的”1”会替换url中的{1}。

Map map = new HashMap();

map.put("1", "hello");

map.put("2", "world");

String result = restTemplate

.getForObject("http://localhost:8081/getIds?param1={1}&param2={2}", String.class,map);

接口端：

@RequestMapping(value = "/getIds", method = RequestMethod.GET)

public @ResponseBody String getIds(String param1, String param2) {

return param1 + param2;

}

通过可变参数的方式：

直接将要传递的值放到getForObject方法的参数结尾，数量不限，它会按顺序替换{1}和{2}

String result = restTemplate

.getForObject("http://localhost:8081/getIds?param1={1}&param2={2}", String.class, "hello", "world");

#### 5、getForEntity方法

getForEntity和getForObject的用法是一样的，只是其返回结果是一个ResponseEntity，其中包含了更多的响应信息，比如：

ResponseEntity response = restTemplate.getForEntity("http://localhost:8081/getIds",String.class);

response.getHeaders(); //响应头

response.getStatusCode(); //响应码

response.getBody(); //响应体，即前面的result

1、简单使用

提供高度封装的接口，可以让我们非常方便地进行RestAPI调用。常见的方法如下：



方法大致分为三组：

* getForObject——optionsForAllow分为一组，这类方法是常规的RestAPI（GET、POST、DELETE）方法调用；
* exchange：接收一个RequestEntity参数，可以自己设置HTTPmethod，URL，headers和body。返回ResponseEntity。
* execute：通过callback接口，可以对请求和返回做更加全面的自定义控制。

一般情况下，使用第一组和第二组方法就够了。

创建RestTemplate

创建RestTemplate时需要一个ClientHttpRequestFactory，通过这个请求工厂，我们可以统一设置请求的超时时间，设置代码以及一些其他细节。

@Bean

publicRestTemplaterestTemplate(ClientHttpRequestFactoryfactory){

RestTemplaterestTemplate=newRestTemplate(factory);

returnrestTemplate;

}

@Bean

publicClientHttpRequestFactorysimpleClientHttpRequestFactory(){

SimpleClientHttpRequestFactoryfactory=newSimpleClientHttpRequestFactory();

factory.setReadTimeout(5000);

factory.setConnectTimeout(15000);

//设置代理

//factory.setProxy(null);

returnfactory;

}

通过上面的配置之后，直接在代码中注入RestTemplate就可以使用了。

1.2接口调用

1）普通接口调用

Map<String,String>vars=Collections.singletonMap("hotel","42");

//通过GET方式调用，返回一个String值，还可以给URL变量设置值（也可通过uriTemplateHandler这个属性自定义）

Stringresult=restTemplate.getForObject(

"https://example.com/hotels/{hotel}/rooms/{hotel}",String.class,vars);

Stringurl="http://127.0.0.1:8080/hello";

JSONObjectparam=newJSONObject();

//restTemplate会根据params的具体类型，调用合适的HttpMessageConvert将请求参数写到请求体body中，并在请求头中添加合适的content-type；

//也会根据responseType的类型（本列子中是JSONObject），设置head中的accept字段，当响应返回的时候再调用合适的HttpMessageConvert进行响应转换

ResponseEntity<JSONObject>responseEntity=restTemplate

.postForEntity(url,params,JSONObject.class);

intstatusCodeValue=responseEntity.getStatusCodeValue();

HttpHeadersheaders=responseEntity.getHeaders();

JSONObjectbody=responseEntity.getBody();

2）添加Header和Cookie

有时候，需要在请求中的Head中添加值或者将某些值通过cookie传给服务端，那么上面这种调用形式就不太满足要求了。

UriComponentsuriComponents=UriComponentsBuilder.fromHttpUrl("127.0.0.1:8080").

path("/test").build(true);

URIuri=uriComponents.toUri();

RequestEntity<JSONObject>requestEntity=RequestEntity.post(uri).

//添加cookie(这边有个问题，假如我们要设置cookie的生命周期，作用域等参数我们要怎么操作)

header(HttpHeaders.COOKIE,"key1=value1").

//添加header

header(("MyRequestHeader","MyValue")

accept(MediaType.APPLICATION\_JSON).

contentType(MediaType.APPLICATION\_JSON).

body(requestParam);

ResponseEntity<JSONObject>responseEntity=restTemplate.exchange(requestEntity,JSONObject.class);

//响应结果

JSONObjectresponseEntityBody=responseEntity.getBody();

3）文件上传

上面的两个例子基本能覆盖平时开发的大多数功能。下面是文件上传的例子。

publicObjectuplaod(@RequestBodyJSONObjectparams)throwsException{

finalStringurl="http://localhost:8888/hello/m3";

//设置请求头

HttpHeadersheaders=newHttpHeaders();

headers.setContentType(MediaType.MULTIPART\_FORM\_DATA);

//设置请求体，注意是LinkedMultiValueMap

FileSystemResourceresource1=newFileSystemResource("D:\\dir1\\ss\\pic1.jpg");

FileSystemResourceresource2=newFileSystemResource("D:\\dir1\\ss\\pic2.jpg");

MultiValueMap<String,Object>form=newLinkedMultiValueMap<>();

form.add("file",resource1);

form.add("file",resource2);

form.add("param1","value1");

HttpEntity<MultiValueMap<String,Object>>files=newHttpEntity<>(form,headers);

JSONObjects=restTemplate.postForObject(url,files,JSONObject.class);

returns;

}

上述代码上传了两张本地图片，通过下面代码可以顺利接收；

@RequestMapping("/m3")

publicObjectfileUpload(@RequestParam("file")MultipartFile[]files,HttpServletRequestrequest)throwsException{

//携带的其他参数可以使用getParameter方法接收

Stringparam1=request.getParameter("param1");

Responseresponse=newResponse();

if(files==null){

response.failure("文件上传错误,服务端未拿到上传的文件！");

returnresponse;

}

for(MultipartFilefile:files){

if(!file.isEmpty()&&file.getSize()>0){

StringfileName=file.getOriginalFilename();

//参考FileCopyUtils这个工具类

file.transferTo(newFile("D:\\"+fileName));

logger.info("文件:{}上传成功...",fileName);

}

}

response.success("文件上传成功");

returnresponse;

}

但是我们发现上面的上传代码中，上传文件的类必须使用FileSystemResource。有时我们会碰到这种情况：文件我们会从文件服务下载到内存中一个InputStream的形式存在，那此时在使用FileSystemResource就不行了。

当然，我们使用讨巧一点的办法也是可以的：先将下载下来的InputStream保存到本地，然后再读取到FileSystemResource，上传后再删除本地临时文件。

但是总觉得这个方法不够完美。最后发现有个同事已经写了相关的实现。这边就直接拿来用了。

//自己实现了一个Resource

publicclassInMemoryResourceextendsByteArrayResource{

privatefinalStringfilename;

privatefinallonglastModified;

publicInMemoryResource(Stringfilename,Stringdescription,byte[]content,longlastModified){

super(content,description);

this.lastModified=lastModified;

this.filename=filename;

}

@Override

publiclonglastModified()throwsIOException{

returnthis.lastModified;

}

@Override

publicStringgetFilename(){

returnthis.filename;

}

}

调整后的上传代码：

@PostMapping("/m3")

publicObjectm3(@RequestBodyJSONObjectparams)throwsException{

finalStringurl="http://localhost:8888/hello/m3";

//设置请求头

HttpHeadersheaders=newHttpHeaders();

headers.setContentType(MediaType.MULTIPART\_FORM\_DATA);

//设置请求体，注意是LinkedMultiValueMap

//下面两个流从文件服务下载，这边省略（注意最后关闭流）

InputStreamfis1=

InputStreamfis2=

InMemoryResourceresource1=newInMemoryResource("file1.jpg","description1",FileCopyUtils.copyToByteArray(fis1),System.currentTimeMillis());

InMemoryResourceresource2=newInMemoryResource("file2.jpg","description2",FileCopyUtils.copyToByteArray(fis2),System.currentTimeMillis());

MultiValueMap<String,Object>form=newLinkedMultiValueMap<>();

form.add("file",resource1);

form.add("file",resource2);

form.add("param1","value1");

HttpEntity<MultiValueMap<String,Object>>files=newHttpEntity<>(form,headers);

JSONObjects=restTemplate.postForObject(url,files,JSONObject.class);

returns;

}

2、其他设置

2.1拦截器

RestTemplate可以设置拦截器做一些统一处理。这个功能感觉和SpringMVC类似。配置也简单：

classMyInterceptorimplementsClientHttpRequestInterceptor{

@Override

publicClientHttpResponseintercept(HttpRequestrequest,byte[]body,ClientHttpRequestExecutionexecution)throwsIOException{

logger.info("enterinterceptor...");

returnexecution.execute(request,body);

}

}

@Bean

publicRestTemplaterestTemplate(ClientHttpRequestFactoryfactory){

RestTemplaterestTemplate=newRestTemplate(factory);

MyInterceptormyInterceptor=newMyInterceptor();

List<ClientHttpRequestInterceptor>list=newArrayList<>();

list.add(myInterceptor);

restTemplate.setInterceptors(list);

returnrestTemplate;

}

2.2ErrorHandler配置

用来对调用错误对统一处理。

publicclassMyResponseErrorHandlerextendsDefaultResponseErrorHandler{

@Override

publicbooleanhasError(ClientHttpResponseresponse)throwsIOException{

returnsuper.hasError(response);

}

@Override

publicvoidhandleError(ClientHttpResponseresponse)throwsIOException{

HttpStatusstatusCode=HttpStatus.resolve(response.getRawStatusCode());

if(statusCode==null){

thrownewUnknownHttpStatusCodeException(response.getRawStatusCode(),response.getStatusText(),

response.getHeaders(),getResponseBody(response),getCharset(response));

}

handleError(response,statusCode);

}

@Override

protectedvoidhandleError(ClientHttpResponseresponse,HttpStatusstatusCode)throwsIOException{

switch(statusCode.series()){

caseCLIENT\_ERROR:

HttpClientErrorExceptionexp1=newHttpClientErrorException(statusCode,response.getStatusText(),response.getHeaders(),getResponseBody(response),getCharset(response));

logger.error("客户端调用异常",exp1);

throwexp1;

caseSERVER\_ERROR:

HttpServerErrorExceptionexp2=newHttpServerErrorException(statusCode,response.getStatusText(),

response.getHeaders(),getResponseBody(response),getCharset(response));

logger.error("服务端调用异常",exp2);

throwexp2;

default:

UnknownHttpStatusCodeExceptionexp3=newUnknownHttpStatusCodeException(statusCode.value(),response.getStatusText(),

response.getHeaders(),getResponseBody(response),getCharset(response));

logger.error("网络调用未知异常");

throwexp3;

}

}

}

@Bean

publicRestTemplaterestTemplate(ClientHttpRequestFactoryfactory){

RestTemplaterestTemplate=newRestTemplate(factory);

MyResponseErrorHandlererrorHandler=newMyResponseErrorHandler();

restTemplate.setErrorHandler(errorHandler);

List<HttpMessageConverter<?>>messageConverters=restTemplate.getMessageConverters();

//通过下面代码可以添加新的HttpMessageConverter

//messageConverters.add(new);

returnrestTemplate;

}

### 总结

通过RestTemplate，可以非常方便的进行RestAPI调用。但是在Spring5中已经不再建议使用RestTemplate，而是建议使用WebClient。WebClient是一个支持异步调用的Client。