Spring HATEOAS - 参考指南

Oliver Gierke, Greg Turnquist, Jay Bryant

2021-09-13

Table of Contents

该项目提供了一些 API,以简化在使用 Spring 特别是 Spring MVC 时遵循 HATEOAS 原理的 REST 表示形式的过程.
它试图解决的核心问题是创建链接和表示组装.

Chapter 1. 前言

1.1. Migrating to Spring HATEOAS 1.0

对于1.0版本，我们借此机会重新评估了 0.x
分支所做的一些设计和包结构选择。并且已经获得了大量的反馈，主要版本的碰撞似乎是重构它们的最自然的地方。

1.1.1. 改变

包结构的最大变化是引入超媒体类型注册 API ,以支持 Spring HATEOAS 中的其他媒体类型. 这导致客户端和服务器 API(分别命名的包)以及 mediatype 包中的媒体类型实现的明确分离.

将代码库升级到新 API 的最简单方法是使用 migration script. 在我们开始之前 ,这些都已经了解了这些变化.

表示模型

ResourceSupport/Resource/Resources/PagedResources
这些类从未被真正恰当地命名过,毕竟,这些类型实际上并不代表资源,而是表示模型.可以通过超媒体信息和提供的内容来丰富它们. 这是新名称映射到旧名称的方式:

- ResourceSupport 现在是 RepresentationModel
- Resource 现在是 EntityModel
- Resources 现在是 CollectionModel
- PagedResources 现在是 PagedModel

因此, ResourceAssembler 已重命名为 RepresentationModelAssembler 它的 toResource(…) 和 toResources(…) 方法被重命名为 toModel(…) 和 toCollectionModel(…) . 名称的变化也反映在 TypeReferences 包含的类中.

- RepresentationModel.getLinks() 现在暴露一个 Links 实例 (在 List<Link> 上) 因为它暴露了其他 API,以使用各种策略来连接和合并不同的 Links 实例. 同样 ,它会自动绑定泛型类型,以允许向实例添加链接的方法返回实例本身.
- LinkDiscoverer APIs 已移至 client 包下.

- LinkBuilder 和 EntityLinks APIs 已移至 server 包下.
- ControllerLinkBuilder 已移到 server.mvc 中,不建议使用 WebMvcLinkBuilder 替代.
- RelProvider 已重命名为 LinkRelationProvider 并返回 LinkRelation 实例而不是 Strings.
- VndError 已移至 mediatype.vnderror 包下.

1.1.2. 迁移脚本

您可以在应用程序根目录放置 一个脚本
,该脚本会将所有导入语句和静态方法引用更新为在源代码仓库中移动的 Spring HATEOAS 类型. 只需下载该文件,然后从您的项目根目录中运行它即可. 默认情况下,它将检查所有 Java 源文件,并用新的替换旧的 Spring HATEOAS 类型引用.

Example 1. Sample application of the migration script

```
$ ./migrate-to-1.0.sh

Migrating Spring HATEOAS references to 1.0 for files : *.java

Adapting ./src/main/java/…
…

Done!
```

请注意,该脚本不一定能够完全修复所有更改,但应涵盖最重要的重构.

现在,您可以检测 Git 客户端中的文件所做的更改,并进行适当的提交.
如果您发现未迁移的方法或类型引用, 请在问题跟踪器中打开一个票据.

1.1.3. 从 1.0 M3 迁移到 1.0 RC1

- Link.andAffordance(…) 的 Affordance 详细信息已移至 Affordances. 现在 ,要手动构建 Affordance 实例,请使用 Affordances.of(link).afford(…). 另请注意,Affordances 暴露了新的 AffordanceBuilder 类型,以使其流式使用.

有关详细信息,请参见 功能可见性.

- `AffordanceModelFactory.getAffordanceModel(…)` 现在接收 `InputPayloadMetadata` 和 `PayloadMetadata` 实例,而不是 `ResolvableTypes` 实例 ,以允许基于非类型的实现. 自定义媒体类型的实现必须相应地进行调整.
- 如果 HAL Forms 的属性值遵循规范中定义的默认属性,则现在不呈现属性属性. 即如果先前将 `required` 显式设置为 `false`,则现在我们省略了 `required` 条目. 现在,对于使用 `PATCH` 作为HTTP方法的模板,我们现在也只强制不需要使用它们.

Chapter 2. Fundamentals

本节介绍 Spring HATEOAS 的基础知识及其基本域抽象.

2.1. Links

超媒体的基本思想是用超媒体元素丰富资源的表示. 最简单的形式是链接.
它们指示客户端可以导航到特定资源. 相关资源的语义是在所谓的链接关系中定义的.
您可能已经在 HTML 文件的标题中看到了这一点:

Example 2. A link in an HTML document

```
<link href="theme.css" rel="stylesheet" type="text/css" />
```

如您所见,链接指向资源 theme.css 并指示它是样式表. 链接通常包含其他信息
,例如资源将返回的媒体类型. 但是,链接的基本构建块是其引用和关系.

Spring HATEOAS 使您可以通过其不可变的 Link 值类型来处理链接.
它的构造函数同时接受超文本引用和链接关系,后者默认为 IANA 链接 self. 在Link
关系阅读有关后者的更多信息.

Example 3. Using links

```
Link link = Link.of("/something");
assertThat(link.getHref()).isEqualTo("/something");
assertThat(link.getRel()).isEqualTo(IanaLinkRelations.SELF);

link = Link.of("/something", "my-rel");
assertThat(link.getHref()).isEqualTo("/something");
assertThat(link.getRel()).isEqualTo(LinkRelation.of("my-rel"));
```

Link 暴露了 RFC-8288 中定义的其他属性. 您可以通过在 Link 实例上调用相应的
wither 方法来设置它们.

在 Spring MVC 的构建链接和 Spring WebFlux 的构建链接中找到有关如何创建指向 在

Spring MVC 中创建 links 和 在 Spring WebFlux 中建立链接
控制器的链接的更多信息.

2.2. URI 模板

对于 Spring HATEOAS Link,超文本引用不仅可以是 URI,而且可以是根据 RFC-6570 的 URI 模板. URI 模板包含所谓的模板变量,并允许扩展这些参数. 这样
,客户端就可以将参数化的模板转换为 URI,而不必了解最终 URI 的结构
,它只需要知道变量的名称即可.

Example 4. Using links with templated URIs

```
Link link = Link.of("/{segment}/something{?parameter}");
assertThat(link.isTemplated()).isTrue(); ①
assertThat(link.getVariableNames()).contains("segment", "parameter"); ②

Map<String, Object> values = new HashMap<>();
values.put("segment", "path");
values.put("parameter", 42);

assertThat(link.expand(values).getHref()) ③
    .isEqualTo("/path/something?parameter=42");
```

① Link 实例指示已被模板化,即它包含URI模板.

② 它暴露了模板中包含的参数

③ 它允许扩展参数.

可以手动构造 URI 模板,稍后再添加模板变量.

Example 5. Working with URI templates

```
UriTemplate template = UriTemplate.of("/{segment}/something")
  .with(new TemplateVariable("parameter", VariableType.REQUEST_PARAM);

assertThat(template.toString()).isEqualTo("/{segment}/something{?parameter
}");
```

2.3. Link 关系

为了指示目标资源与当前的关系,使用了一种所谓的链接关系. Spring HATEOAS 提供了 LinkRelation 类型,可以轻松地创建基于 String 的实例.

2.3.1. IANA link 关系

Internet 分配号码授权机构包含一组 预定义的链接关系. 可以通过 IanaLinkRelations 引用它们.

Example 6. Using IANA link relations

```
Link link = Link.of("/some-resource"), IanaLinkRelations.NEXT);

assertThat(link.getRel()).isEqualTo(LinkRelation.of("next"));
assertThat(IanaLinkRelation.isIanaRel(link.getRel())).isTrue();
```

2.4. 表示模型

为了轻松创建丰富的超媒体表示,Spring HATEOAS 提供了一组在其根处具有 RepresentationModel 的类. 它基本上是用于收集链接的容器
,并具有方便的方法将其添加到模型中. 以后可以将模型呈现为各种媒体类型格式
,这些格式将定义超媒体元素在表示中的外观. 有关此的更多信息,请查看媒体类型.

Example 7. The RepresentationModel class hierarchy

```
class RepresentationModel
class EntityModel
class CollectionModel
class PagedModel

EntityModel -|> RepresentationModel
CollectionModel -|> RepresentationModel
PagedModel -|> CollectionModel
```

使用 RepresentationModel 的默认方法是创建其子类,以包含该表示应包含的所有属性

,创建该类的实例,填充这些属性,并使用链接对其进行充实.

Example 8. A sample representation model type

```
class PersonModel extends RepresentationModel<PersonModel> {

  String firstname, lastname;
}
```

要让 `RepresentationModel.add(…)` 返回其自身的实例,必须进行通用的输入 .现在可以像这样使用模型类型:

Example 9. Using the person representation model

```
PersonModel model = new PersonModel();
model.firstname = "Dave";
model.lastname = "Matthews";
model.add(Link.of("https://myhost/people/42"));
```

如果您从 Spring MVC 或 WebFlux 控制器返回了这样的实例,并且客户端发送了一个 `Accept` 头集设置为 `application/hal+json`,则响应将如下所示:

Example 10. The HAL representation generated for the person representation model

```
{
  "_links" : {
    "self" : {
      "href" : "https://myhost/people/42"
    }
  },
  "firstname" : "Dave",
  "lastname" : "Matthews"
}
```

2.4.1. 模型表示子资源

对于由单个对象或概念支持的资源,存在便利的 `EntityModel` 类型.
不必为每个概念创建自定义模型类型,您只需重用一个已经存在的类型并将其实例包装到 `EntityModel` 中即可.

Example 11. Using `EntityModel` to wrap existing objects

```
Person person = new Person("Dave", "Matthews");
EntityModel<Person> model = EntityModel.of(person);
```

2.4.2. 模型表示集合资源

对于概念上是集合的资源,可以使用 `CollectionModel`. 它的元素可以是简单对象,也可以是 `RepresentationModel` 实例.

Example 12. Using `CollectionModel` to wrap a collection of existing objects

```
Collection<Person> people = Collections.singleton(new Person("Dave",
"Matthews"));
CollectionModel<Person> model = CollectionModel.of(people);
```

Chapter 3. Server-side support

3.1. 在 Spring MVC 中创建 links

虽然我们现在已经有了 domain 词汇表,但挑战仍然存在:如何以不脆弱的方式创建实际的 URI,以将其包装到 Link 实例中. 现在,我们将不得不在各处复制 URI 字符串.
这样做是脆弱且不可维持的.

假设您已按照以下方式实现了 Spring MVC 控制器:

```
@Controller
class PersonController {

  @GetMapping("/people")
  HttpEntity<PersonModel> showAll() { … }

  @GetMapping(value = "/{person}", method = RequestMethod.GET)
  HttpEntity<PersonModel> show(@PathVariable Long person) { … }
}
```

我们在这里看到两个约定. 第一个是通过控制器方法的 @GetMapping 注解暴露的集合资源,该集合的各个元素作为直接子资源暴露. 集合资源可以在一个简单的 URI (如刚刚所示)或更复杂的 URI (例如 /people/{id}/addresses)中暴露.
假设您想要链接到所有人的集合资源。按照上面的方法会导致两个问题:

- 要创建绝对 URI,您需要查找协议,主机名,端口,Servlet 地址和其他值. 这很麻烦,并且需要手动编写字符串连接代码.
- 您可能不希望将 /people 连接到基本 URI 之上,因为这样就必须在多个位置维护信息. 如果更改映射,则必须更改所有指向该映射的客户端.

Spring HATEOAS 现在提供了一个 WebMvcLinkBuilder,使您可以通过指向控制器类来创建链接. 以下示例显示了如何执行此操作:

```
import static org.sfw.hateoas.server.mvc.WebMvcLinkBuilder.*;

Link link = linkTo(PersonController.class).withRel("people");

assertThat(link.getRel()).isEqualTo(LinkRelation.of("people"));
assertThat(link.getHref()).endsWith("/people");
```

`WebMvcLinkBuilder` 使用 Spring 的 `ServletUriComponentsBuilder` 来从当前请求中获取基本 URI 信息. 假设您的应用程序在 `localhost:8080/your-app` 上运行,那么这正是您要在其上构建其他部分的 URI. 现在, 构建器将检查给定控制器类的根映射,从而以 `localhost:8080/your-app/people` 结尾. 您还可以构建更多的嵌套链接. 以下示例显示了如何执行此操作:

```
Person person = new Person(1L, "Dave", "Matthews");
//                 /person                 /     1
Link link =
linkTo(PersonController.class).slash(person.getId()).withSelfRel();
assertThat(link.getRel(), is(IanaLinkRelation.SELF.value()));
assertThat(link.getHref(), endsWith("/people/1"));
```

该构建器还允许创建 URI 实例以进行构建(例如,响应头值):

```
HttpHeaders headers = new HttpHeaders();
headers.setLocation(linkTo(PersonController.class).slash(person).toUri());

return new ResponseEntity<PersonModel>(headers, HttpStatus.CREATED);
```

3.1.1. 建立指向方法的链接

您甚至可以建立指向方法的链接,或创建虚拟控制器方法调用. 第一种方法是将 `Method` 实例传递给 `WebMvcLinkBuilder`. 以下示例显示了如何执行此操作:

```
Method method = PersonController.class.getMethod("show", Long.class);
Link link = linkTo(method, 2L).withSelfRel();

assertThat(link.getHref()).endsWith("/people/2"));
```

这仍然有点令人不满意,因为我们必须首先获取一个 `Method` 实例,该实例会引发异常,并且通常很麻烦. 至少我们不重复映射.
更好的方法是在控制器代理上对目标方法进行伪方法调用,我们可以使用 `methodOn(…)` 帮助器创建该方法. 以下示例显示了如何执行此操作:

```
Link link =
linkTo(methodOn(PersonController.class).show(2L)).withSelfRel();

assertThat(link.getHref()).endsWith("/people/2");
```

`methodOn(…)` 创建记录方法调用的控制器类的代理,
并在为方法的返回类型创建的代理中暴露它.这样就可以流式地表达我们希望获得映射的方法.但是,在使用这种技术可以获得的方法上有一些限制

- 返回类型必须能够代理,因为我们需要暴露对其的方法调用.
- 通常忽略传递给方法的参数(通过 `@PathVariable` 引用的参数除外,因为它们构成了 URI).

3.2. 在 Spring WebFlux 中建立链接

TODO

3.3. 功能可见性

> The affordances of the environment are what it offers … what it provides or furnishes, either for good or ill. The verb 'to afford' is found in the dictionary, but the noun 'affordance' is not. I have made it up.
>
> — James J. Gibson, The Ecological Approach to Visual Perception (page 126)

基于 REST 的资源不仅提供数据,还提供一些功能.
一个具有灵活性的服务的最后一个要素是有关如何使用各种功能的详细内容.
由于功能可见性是与链接相关联的,因此 Spring HATEOAS 提供了一个 API,可以将所需的尽可能多的相关方法附加到链接上. 就像您可以通过指向 Spring MVC 控制器方法来创建链接一样(请参见在 在 Spring MVC 中创建 links 中构建链接以了解详细信息),您…

下面的代码显示了如何 **self** 链接

Example 13. Connecting affordances to GET /employees/{id}

```
@GetMapping("/employees/{id}")
public EntityModel<Employee> findOne(@PathVariable Integer id) {

  Class<EmployeeController> controllerClass = EmployeeController.class;

  // Start the affordance with the "self" link, i.e. this method.
  Link findOneLink =
linkTo(methodOn(controllerClass).findOne(id)).withSelfRel(); ①

  // Return the affordance + a link back to the entire collection
resource.
  return EntityModel.of(EMPLOYEES.get(id), //
      findOneLink //

.andAffordance(afford(methodOn(controllerClass).updateEmployee(null, id)))
②

.andAffordance(afford(methodOn(controllerClass).partiallyUpdateEmployee(nu
ll, id)))); ③
}
```

① 创建 **self** 链接.

② 将 `updateEmployee` 方法与 `self` 链接相关联.

③ 将 `partiallyUpdateEmployee` 方法与 `self` 链接相关联.

使用 `.andAffordance(afford(…))`,您可以使用控制器的方法将 `PUT` 和 `PATCH` 操作连接到 `GET` 操作. 想象一下,上面提供的相关方法如下所示:

Example 14. updateEmpoyee method that responds to PUT /employees/{id}

```
@PutMapping("/employees/{id}")
public ResponseEntity<?> updateEmployee( //
    @RequestBody EntityModel<Employee> employee, @PathVariable Integer id)
```

Example 15. partiallyUpdateEmployee method that responds to PATCH /employees/{id}

```
@PatchMapping("/employees/{id}")
public ResponseEntity<?> partiallyUpdateEmployee( //
    @RequestBody EntityModel<Employee> employee, @PathVariable Integer id)
```

使用 `afford(…)` 方法指向这些方法将导致 Spring HATEOAS
分析请求正文和响应类型并捕获元数据,以允许不同的媒体类型实现使用该信息将该信息转换为输入和输出的描述.

3.3.1. 手动创建 affordances

尽管是注册链接功能的主要方式,但可能有必要手动构建其中的一些功能. 这可以通过使用 `Affordances` API 来实现:

Example 16. Using the Affordances API to manually register affordances

```
var methodInvocation = methodOn(EmployeeController.class).all();

var link = Affordances.of(linkTo(methodInvocation).withSelfRel()) ①

    .afford(HttpMethod.POST) ②
    .withInputAndOutput(Employee.class) //
    .withName("createEmployee") //

    .andAfford(HttpMethod.GET) ③
    .withOutput(Employee.class) //
    .addParameters(//
        QueryParameter.optional("name"), //
        QueryParameter.optional("role")) //
    .withName("search") //

    .toLink();
```

① 首先,从一个 `Link` 实例创建 `Affordances` 实例,然后创建一个用于描述能力的上下文.

② 每种功能都应以其应支持的HTTP方法开始. 然后,我们将类型注册为有效负载描述,并显式命名提供. 后者可以省略,并且默认名称将从HTTP方法和输入类型名称继承. 这有效地创建了与创建的 `EmployeeController.newEmployee(…)` 指针相同的能力

③ 建立下一个 `affordance` 是为了反映指向 `EmployeeController.search(…)` 的指针发生了什么. 在这里,我们将 `Employee` 定义为创建的响应的模型,并显式注册 `QueryParameters`.

`Affordances` 由媒体类型特定的 `Affordances` 模型支持,该模型将一般 `Affordances` 元数据转换为特定的表示形式. 请确保检查 媒体类型 部分中有关 `affordances` 的部分,以查找有关如何控制该元数据的显示量的更多详细信息.

3.4. 转发报头处理

当您的应用程序位于代理,负载均衡器之后或在云中时,最常使用 RFC-7239 forwarding headers. 实际接收 Web 请求的节点是基础结构的一部分,并将请求转发到您的应用程序.

您的应用程序可能正在 localhost:8080 上运行,但对于外界,您应该位于 reallycoolsite.com(和网络的标准端口80)上. 通过使代理包括额外的 header (许多人已经这样做了),Spring HATEOAS 可以正确地生成链接,因为它使用 Spring Framework 功能来获取原始请求的基本 URI

任何可以根据外部输入更改根 URI 的内容都必须得到适当的保护.
这就是默认情况下 **disabled** 转发 header 处理的原因.
您必须使它能够运行.
如果您要部署到云中或控制代理和负载均衡器的配置中,那么您肯定会希望使用此功能.

要启用转发 header 处理,您需要在应用程序中注册 Spring 的 Spring MVC 的 ForwardedHeaderFilter(详细信息)或 Spring WebFlux 的 ForwardedHeaderTransformer(详细信息). 在 Spring Boot 应用程序中 ,这些组件可以简单地声明为 Spring Bean,如此处所述.

Example 17. Registering a ForwardedHeaderFilter

```
@Bean
ForwardedHeaderFilter forwardedHeaderFilter() {
    return new ForwardedHeaderFilter();
}
```

这将创建一个处理所有 X-Forwarded-… header 的 servlet 过滤器. 并且它将在 servlet 处理程序中正确注册.

对于 Spring WebFlux 应用程序,reactive 对应的是 ForwardedHeaderTransformer:

Example 18. Registering a ForwardedHeaderTransformer

```
@Bean
ForwardedHeaderTransformer forwardedHeaderTransformer() {
    return new ForwardedHeaderTransformer();
}
```

这将创建一个转换响应式 Web 请求,处理 X-Forwarded-… header 的函数. 并且它将在 WebFlux 中正确注册.

有了上面显示的配置,传递 X-Forwarded-… header 的请求将看到那些反映在生成的链接中:

Example 19. A request using X-Forwarded-… headers

```
curl -v localhost:8080/employees \
    -H 'X-Forwarded-Proto: https' \
    -H 'X-Forwarded-Host: example.com' \
    -H 'X-Forwarded-Port: 9001'
```

Example 20. 带有生成考虑这些标题的链接的相应响应

```
{
  "_embedded": {
    "employees": [
      {
        "id": 1,
        "name": "Bilbo Baggins",
        "role": "burglar",
        "_links": {
          "self": {
            "href": "https://example.com:9001/employees/1"
          },
          "employees": {
            "href": "https://example.com:9001/employees"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "https://example.com:9001/employees"
    },
    "root": {
      "href": "https://example.com:9001"
    }
  }
}
```

3.5. 使用 EntityLinks interface

当前未为 Spring WebFlux 应用程序提供 EntityLinks 及其各种实现. EntityLinks SPI 中定义的合同最初是针对 Spring Web MVC 的,不考虑 Reactor 类型.
开发支持响应式编程的可比合同仍在进行中.

到目前为止,我们已经通过指向 Web 框架实现(即 Spring MVC 控制器)创建了链接,并检查了映射. 在许多情况下,这些类实质上是读取和写入由模型类支持的表示形式.

EntityLinks 接口现在暴露一个 API,以根据模型类型查找 Link 或 LinkBuilder. 这些方法实质上返回的链接要么指向集合资源(例如 /people),要么指向 item 资源(例如 /people/1). 以下示例显示如何使用 EntityLinks:

```
EntityLinks links = …;
LinkBuilder builder = links.linkFor(Customer.class);
Link link = links.linkToItemResource(Customer.class, 1L);
```

通过在 Spring MVC 配置中激活 @EnableHypermediaSupport,可以通过依赖注入来使用 EntityLinks. 这将导致注册 EntityLink 的各种默认实现. 最基础的是检查 SpringMVC 控制器类的 ControllerEntityLinks. 如果要注册自己的 EntityLinks 实现,请查看本节.

3.5.1. 基于 Spring MVC controllers 的 EntityLinks

激活实体链接功能将导致检查当前 ApplicationContext 中可用的所有 Spring MVC 控制器的 @ExposesResourceFor(…) 注解. 注解暴露了控制器管理的模型类型. 除此之外,我们假设您遵守以下 URI 映射设置和约定:

- 类型级别 @ExposesResourceFor(…) 声明控制器暴露其集合和 items 资源的实体类型.
- 表示集合资源的类级别的基本映射.
- 附加的方法级别映射,该映射扩展了映射以将标识符附加为附加的路径段.

以下示例显示了支持 EntityLinks 的控制器的实现:

```
@Controller
@ExposesResourceFor(Order.class) ①
@RequestMapping("/orders") ②
class OrderController {

  @GetMapping ③
  ResponseEntity orders(…) { … }

  @GetMapping("{id}") ④
  ResponseEntity order(@PathVariable("id") … ) { … }
}
```

① 控制器表示它正在暴露实体 `Order` 的集合和 `items` 资源.

② 其收集资源显示在 `/orders` 下

③ 该收集资源可以处理 `GET` 请求. 在方便时,为其他 `HTTP` 方法添加更多方法.

④ 一种附加的控制器方法,用于处理带有路径变量的从属资源,以暴露 `items` 资源,即单个 `Order`.

有了此功能,当您在 `Spring MVC` 配置中启用 `EntityLinks` `@EnableHypermediaSupport` 时,可以按以下方式创建到控制器的链接:

```
@Controller
class PaymentController {

  private final EntityLinks entityLinks;

  PaymentController(EntityLinks entityLinks) { ①
    this.entityLinks = entityLinks;
  }

  @PutMapping(…)
  ResponseEntity payment(@PathVariable Long orderId) {

    Link link = entityLinks.linkToItemResource(Order.class, orderId); ②
    …
  }
}
```

① 注入由 `EntityLinks` 在您的配置中提供的 `EntityLink`.

② 使用 API 通过使用实体类型而不是控制器类来构建链接.

如您所见,您可以引用管理 `Order` 实例的资源,而无需显式引用 `OrderController`.

3.5.2. EntityLinks API 细节

从根本上讲,`EntityLinks` 允许构建 `LinkBuilder` 和 `Link` 实例以收集实体类型的集合资源和 items 资源. 以 `linkFor…` 开头的方法将为您生成 `LinkBuilder` 实例,以供您扩展和扩充其他路径段,参数等.以 `linkTo` 开头的方法将生成完全准备好的 `Link` 实例.

虽然对于收集资源而言,提供一个实体类型就足够了,但指向 items 资源的链接将需要提供一个标识符. 通常如下所示:

Example 21. Obtaining a link to an item resource

```
entityLinks.linkToItemResource(order, order.getId());
```

如果您发现自己重复了这些方法调用,则可以将标识符提取步骤提取到可重用的 `Function` 中

,以在不同的调用中重用:

```
Function<Order, Object> idExtractor = Order::getId; ①

entityLinks.linkToItemResource(order, idExtractor); ②
```

① 标识符提取已外部化,因此可以保存在一个字段中或保持不变.

② 使用提取器的链接查找.

TypedEntityLinks

由于控制器的实现通常围绕实体类型进行分组,因此您经常会在整个控制器类中使用相同的提取器功能(详细信息请参见 EntityLinks API 细节). 通过获取一次提供提取器的 TypedEntityLinks 实例,我们可以进一步集中标识符提取逻辑 ,以便实际查找根本不再需要处理提取.

Example 22. Using TypedEntityLinks

```
class OrderController {

  private final TypedEntityLinks<Order> links;

  OrderController(EntityLinks entityLinks) { ①
    this.links = entityLinks.forType(Order::getId); ②
  }

  @GetMapping
  ResponseEntity<Order> someMethod(…) {

    Order order = … // lookup order

    Link link = links.linkToItemResource(order); ③
  }
}
```

① 注入 `EntityLinks` 实例.

② 表示您要使用特定的标识符提取器功能查找 `Order` 实例.

③ 根据唯一的 `Order` 实例查找商品资源链接.

3.5.3. EntityLinks 作为 SPI

`@EnableHypermediaSupport` 创建的 `EntityLinks` 实例的类型为 `DelegatingEntityLinks`,它将依次拾取 `ApplicationContext` 中可作为 bean 使用的所有其他 `EntityLinks` 实现. 它已注册为 primary bean, 因此通常在您注入 `EntityLink` 时始终是唯一的注入候选对象. `ControllerEntityLinks` 是设置中将包含的默认实现,但是用户可以自由实现和注册自己的实现. 使那些对 `EntityLinks` 实例可用的对象可用于注入是将实现注册为 Spring bean 的问题.

Example 23. Declaring a custom EntityLinks implementation

```
@Configuration
class CustomEntityLinksConfiguration {

  @Bean
  MyEntityLinks myEntityLinks(…) {
    return new MyEntityLinks(…);
  }
}
```

这种数据可扩展性的一个例子是 Spring Data REST 的 RepositoryEntityLinks ,它使用存储库映射信息来创建指向由 Spring Data 存储库支持的资源的链接. 同时 ,它甚至暴露了其他类型资源的其他查找方法. 如果要使用这些,只需显式注入 RepositoryEntityLinks.

3.6. 表示模型汇编器

由于必须在多个地方使用从实体到表示模型的映射,因此创建一个负责这样做的专用类是有意义的 . 转换包含非常自定义的步骤,但也包含一些样板步骤:

1. 模型类的实例化
2. 添加一个具有 rel of self 关系的链接,指向要渲染的资源

Spring HATEOAS 现在提供了 RepresentationModelAssemblerSupport 基类 ,该基类有助于减少您需要编写的代码量. 以下示例显示了如何使用它:

```
class PersonModelAssembler extends
RepresentationModelAssemblerSupport<Person, PersonModel> {

  public PersonModelAssembler() {
    super(PersonController.class, PersonModel.class);
  }

  @Override
  public PersonModel toModel(Person person) {

    PersonModel resource = createResource(person);
    // … do further mapping
    return resource;
  }
}
```

`createResource(…)` 是您编写的代码,用于在给定 `Person` 对象的情况下实例化 `PersonModel` 对象.
它应该只专注于设置属性,而不是填充链接.

像前面的示例中一样设置类,可以为您带来以下好处:

- 有几个 `createModelWithId(…)` 方法可让您创建资源的实例,并为它添加一个带有 `self` 属性的 `Link`, 该链接的 `href` 由配置的控制器的请求映射加上实体的 `ID(for example, /people/1)`.
- 资源类型通过反射实例化,并且需要一个无参数的构造函数.
  如果要使用专用的构造函数或避免反射性能开销,则可以重写 `instantiateModel(…)`.

然后,您可以使用汇编器来组装 `RepresentationModel` 或 `CollectionModel`.
以下示例创建 `PersonModel` 实例的 `CollectionModel`

```
Person person = new Person(…);
Iterable<Person> people = Collections.singletonList(person);

PersonModelAssembler assembler = new PersonModelAssembler();
PersonModel model = assembler.toModel(person);
CollectionModel<PersonModel> model = assembler.toCollectionModel(people);
```

3.7. 表示模型处理器

有时,您需要在 `assembled` 超媒体表示后对其进行调整和调整.

一个完美的例子是,当您有一个负责处理订单履行的控制器,但您需要添加与付款相关的链接.

想象一下,让您的 `Order` 系统生产这种类型的超媒体:

```
{
  "orderId" : "42",
  "state" : "AWAITING_PAYMENT",
  "_links" : {
    "self" : {
      "href" : "http://localhost/orders/999"
    }
  }
}
```

您希望添加一个链接,以便客户可以付款,但不想将有关 `PaymentController` 的详细信息混入 `OrderController` 中.

您可以像下面这样编写 `RepresentationModelProcessor`,而不用污染 `Order` 系统的详细信息:

```
public class PaymentProcessor implements
RepresentationModelProcessor<EntityModel<Order>> { ①

  @Override
  public EntityModel<Order> process(EntityModel<Order> model) {

    model.add( ②

Link.of("/payments/{orderId}").withRel(LinkRelation.of("payments")) //
            .expand(model.getContent().getOrderId()));

    return model; ③
  }
}
```

① 该处理器将仅应用于 `EntityModel<Order>` 对象.

② 通过添加无条件链接来处理现有的 `EntityModel` 对象.

③ 返回 `EntityModel` 以便可以将其序列化为请求的媒体类型.

在您的应用程序中注册处理器:

```
@Configuration
public class PaymentProcessingApp {

  @Bean
  PaymentProcessor paymentProcessor() {
    return new PaymentProcessor();
  }
}
```

现在,当您发布 `Order` 的超媒体表示时,客户端会收到以下信息:

```
{
  "orderId" : "42",
  "state" : "AWAITING_PAYMENT",
  "_links" : {
    "self" : {
      "href" : "http://localhost/orders/999"
    },
    "payments" : { ①
      "href" : "/payments/42" ②
    }
  }
}
```

① 您会看到此链接的关系已插入 `LinkRelation.of("payments")`

② URI由处理器提供.

这个例子很简单,但是您可以轻松地:

- 使用 `WebMvcLinkBuilder` 或 `WebFluxLinkBuilder` 构造到您的 `PaymentController` 的动态链接.
- 注入必要的服务,以有条件地添加由状态驱动的其他链接 (e.g. `cancel`, `amend`) .
- 利用诸如 `Spring Security` 之类的跨领域服务根据当前用户的上下文添加 ,删除或修改链接.

同样,在此示例中,`PaymentProcessor` 更改提供的 `EntityModel<Order>`.
您也可以将其替换为另一个对象. 请注意,API 要求返回类型等于输入类型.

3.8. 使用 `LinkRelationProvider` API

构建链接时,通常需要确定要用于链接的关系类型. 在大多数情况下,关系类型与(域)类型直接关联. 我们封装了详细的算法,以在LinkRelationProvider API 之后查找关系类型,该 API 使您可以确定单个资源和集合资源的关系类型.
查找关系类型的算法如下:

1. 如果使用 `@Relation`, 则使用注解中配置的值.

2. 如果不是,我们默认为不使用大写字母的简单类名,再加上集合 `rel` 的附加列表. `List for the collection rel`.
3. 如果 `EVO inflector` 位于类路径中,则使用复数算法提供的多个单资源 `rel`.
4. 用 `@ExposesResourceFor` 注解的 `@Controller` (有关详细信息,请参见 `使用 EntityLinks interface`) 透明地查找注解中配置的类型的关系类型,以便您可以使用 `LinkRelationProvider.getItemResourceRelFor(MyController.class)` 并获取 `domain` 暴露.

使用 `@EnableHypermediaSupport` 时,`LinkRelationProvider` 自动显示为 `Spring Bean`. 您可以通过实现接口并依次将其暴露为 `Spring bean` 来插入自定义提供程序.

Chapter 4. 媒体类型

4.1. HAL - Hypertext Application Language

JSON Hypertext Application Language 或 HAL 是不基于任何 Web 栈的最简单,使用最广泛的超媒体类型之一.

这是 Spring HATEOAS 采用的第一种基于规范的媒体类型.

4.1.1. 构建 HAL representation 模型

从 Spring HATEOAS 1.1 开始, 我们提供了专用的 HalModelBuilder, 该模型允许通过 HAL 惯用 API 创建 RepresentationModel 实例. 这些是其基础假设:

1. HAL representation 可以由构建 representation 中包含的 domain 字段的任意对象(实体)支持.
2. representation 可以通过各种嵌入文档来丰富, 这些文档可以是任意对象, 也可以是 HAL representation 本身(即包含嵌套的嵌入和链接). .
3. 某些特定的 HAL 模式(例如预览)可以直接在 API 中使用, 这样设置 representation 的代码读起来就像你在按照这些习语描述 HAL representation 一样. .

这是使用的 API 的示例: :

```
// An order
var order = new Order(…); ①

// The customer who placed the order
var customer = customer.findById(order.getCustomerId());

var customerLink = Link.of("/orders/{id}/customer") ②
  .expand(order.getId())
  .withRel("customer");

var additional = …

var model = HalModelBuilder.halModel(order)
  .preview(new CustomerSummary(customer)) ③
  .forLink(customerLink) ④
  .embed(additional) ⑤
  .link(Link.of(…, IanaLinkRelations.SELF));
  .build();
```

① 我们设置一些 domain 类型. 在本示例中,order 和 customer 具有一定的关系.

② 我们准备了一个链接, 指向将公开 customer 详细信息的资源

③ 我们通过提供应该在 _embeddable 子句中呈现的有效负载开始构建预览.

④ 我们通过提供目标链接来结束预览. 它被透明地添加到 _links 对象中,
它的链接关系被用作上一步中提供的对象的键.

⑤ 可以添加其他对象以显示在 _embedded 下. 它们所列的键来自对象关系设置.
它们可以通过 @Relation 或专用的 LinkRelationProvider 自定义它们 (see 使用 LinkRelationProvider API for details).

```
{
  "_links" : {
    "self" : { "href" : "…" }, ①
    "customer" : { "href" : "/orders/4711/customer" } ②
  },
  "_embedded" : {
    "customer" : { … }, ③
    "additional" : { … } ④
  }
}
```

① 提供 self 链接.

② 通过 `….preview(…).forLink(…)` 添加 `customer` 链接.

③ 提供的预览对象.

④ 通过 `….embed(…)` 提供其他元素.

在 HAL 中, `_embedded` 也用于表示顶级集合.
通常将它们归类为根据对象类型得出的链接关系. 即 HAL中的订单清单如下所示:

```
{
  "_embedded" : {
    "orders : [
      … ①
    ]
  }
}
```

① `Individual order documents go here.`

创建这样的表示很简单:

```
Collection<Order> orders = …;

HalModelBuilder.emptyHalDocument()
  .embed(orders);
```

也就是说, 如果 `Order` 为空, 则无法扩展链接关系以显示在 `_embedded` 内部, 因此,
如果集合为空, 则文档将保持为空.

如果您希望显式地传递一个空集合, 则可以将类型传递给采用 `Collection` 的 `….embed(…)` 方法的重载. 如果传递给该方法的集合为空,
则将导致使用其链接关系从给定类型扩展的字段呈现.

```
HalModelBuilder.emptyHalModel()
  .embed(Collections.emptyList(), Order.class);
  // or
  .embed(Collections.emptyList(), LinkRelation.of("orders"));
```

将创建以下更明确的表示形式.

```
{
  "_embedded" : {
    "orders" : []
  }
}
```

4.1.2. 配置链接渲染

在HAL中,`_links` 是一个 JSON 对象. 属性名称是 link relations,
每个值都是https://tools.ietf.org/html/draft-kelly-json-hal-07#section-4.1.1[链接对象或链接对象数组].

对于具有两个或多个链接的给定链接关系,规范在表示形式上很明确:

Example 24. HAL document with two links associated with one relation

```
{
  "_links": {
    "item": [
      { "href": "https://myhost/cart/42" },
      { "href": "https://myhost/inventory/12" }
    ]
  },
  "customer": "Dave Matthews"
}
```

但是,如果给定关系只有一个链接,则说明是不明确的. 您可以将其渲染为单个对象或单个 `item` 数组.

默认情况下,Spring HATEOAS 使用最简洁的方法并呈现如下所示的单链接关系:

Example 25. HAL document with single link rendered as an object

```
{
  "_links": {
    "item": { "href": "https://myhost/inventory/12" }
  },
  "customer": "Dave Matthews"
}
```

一些用户更喜欢在使用 HAL 时不在数组和对象之间切换.他们更喜欢这种类型的渲染:

Example 26. HAL with single link rendered as an array

```
{
  "_links": {
    "item": [{ "href": "https://myhost/inventory/12" }]
  },
  "customer": "Dave Matthews"
}
```

如果您希望自定义此策略,则要做的就是将 `HalConfiguration` bean注入应用程序配置中.有多种选择.

Example 27. Global HAL single-link rendering policy

```
@Bean
public HalConfiguration globalPolicy() {
    return new HalConfiguration() //
            .withRenderSingleLinks(RenderSingleLinks.AS_ARRAY); ①
}
```

① 通过将所有单链接关系呈现为数组来覆盖 Spring HATEOAS 的默认设置.

如果您只想重写某些特定的链接关系,则可以创建如下的 `HalConfiguration` bean:

Example 28. Link relation-based HAL single-link rendering policy

```
@Bean
public HalConfiguration linkRelationBasedPolicy() {
    return new HalConfiguration() //
            .withRenderSingleLinksFor( //
                    IanaLinkRelations.ITEM, RenderSingleLinks.AS_ARRAY) ①
            .withRenderSingleLinksFor( //
                    LinkRelation.of("prev"), RenderSingleLinks.AS_SINGLE);
②
}
```

① 始终将 `item` link 关系呈现为数组.

② 当只有一个链接时,将 `prev` 链接关系呈现为一个对象.

如果这些都不符合您的需求,则可以使用 Ant 样式的路径模式:

Example 29. Pattern-based HAL single-link rendering policy

```
@Bean
public HalConfiguration patternBasedPolicy() {
    return new HalConfiguration() //
            .withRenderSingleLinksFor( //
                    "http*", RenderSingleLinks.AS_ARRAY); ①
}
```

① 将以 `http` 开头的所有链接关系呈现为数组.

基于模式的方法使用 Spring 的 `AntPathMatcher`.

所有这些 `HalConfiguration` 凋零都可以组合形成一项全面的策略. 确保对您的 API 进行广泛测试,以免出现意外情况.

4.1.3. 链接标题国际化

HAL为其链接对象定义 `title` 属性. 可以使用 Spring 的资源包抽象和名为 `rest-messages` 的资源包来填充这些标题,以便客户端可以在其 UI 中直接使用它们.

该捆绑包将自动设置,并在 HAL 链接序列化期间使用.

要定义链接的标题,请使用密钥模板 `_links.$relationName.title` ,如下所示:

Example 30. A sample rest-messages.properties

```
_links.cancel.title=Cancel order
_links.payment.title=Proceed to checkout
```

这将导致以下 HAL 表示形式:

Example 31. A sample HAL document with link titles defined

```
{
  "_links" : {
    "cancel" : {
      "href" : "…"
      "title" : "Cancel order"
    },
    "payment" : {
      "href" : "…"
      "title" : "Proceed to checkout"
    }
  }
}
```

4.1.4. 使用 CurieProvider API

Web Linking RFC 描述了注册和扩展链接关系类型. 已注册的 rels 是在 IANA registry of link relation types 中注册的众所周知的字符串.
不希望注册关系类型的应用程序可以使用扩展 rel URI. 每个都是唯一标识关系类型的 URI. rel URI 可以序列化为紧凑URI或 Curie. 例如,如果 ex 定义为 example.com/rels/{rel}.,则 ex:persons curie 代表链接关系类型 example.com/rels/persons.
如果使用 curies,则基本URI必须存在于响应范围中.

默认的 RelProvider 创建的 rel 值是扩展关系类型,因此,必须是 URI,这会导致很多开销. CurieProvider API 可以解决此问题:它使您可以将基 本URI 定义为 URI 模板

,并定义代表该基本 URI 的前缀. 如果存在 CurieProvider,则 RelProvider 会在所有 rel 值之前添加居里前缀. 此外,还会将 curies 链接自动添加到HAL资源.

以下配置定义了默认的 curie 提供程序:

```
@Configuration
@EnableWebMvc
@EnableHypermediaSupport(type= {HypermediaType.HAL})
public class Config {

  @Bean
  public CurieProvider curieProvider() {
    return new DefaultCurieProvider("ex", new
UriTemplate("https://www.example.com/rels/{rel}"));
  }
}
```

请注意,现在 ex: 前缀会自动出现在所有未向IANA注册的 rel 值之前,例如 ex:orders. 客户可以使用 curies 链接将 Curie 解析为完整表单. 以下示例显示了如何执行此操作:

```
{
  "_links": {
    "self": {
      "href": "https://myhost/person/1"
    },
    "curies": {
      "name": "ex",
      "href": "https://example.com/rels/{rel}",
      "templated": true
    },
    "ex:orders": {
      "href": "https://myhost/person/1/orders"
    }
  },
  "firstname": "Dave",
  "lastname": "Matthews"
}
```

由于 CurieProvider API的目的是允许自动创建居里文件

,因此每个应用程序范围只能定义一个 `CurieProvider` bean.

4.2. HAL-FORMS

HAL-FORMS 旨在向 HAL media type 添加运行时 form 支持 .

> HAL-FORMS "looks like HAL." However, it is important to keep in mind that HAL-FORMS is not the same as HAL—the two should not be thought of as interchangeable in any way.
>
> — Mike Amundsen, HAL-FORMS spec

要启用此媒体类型,请将以下配置放入代码中:

Example 32. HAL-FORMS enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.HAL_FORMS)
public class HalFormsApplication {

}
```

每当客户端提供带有 `application/prs.hal-forms+json` 的 `Accept` 请求头时 ,您都可以期待这样的事情:

Example 33. HAL-FORMS sample document

```
{
  "firstName" : "Frodo",
  "lastName" : "Baggins",
  "role" : "ring bearer",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/employees/1"
    }
  },
  "_templates" : {
    "default" : {
      "method" : "put",
      "contentType" : "",
      "properties" : [ {
        "name" : "firstName",
        "required" : true
      }, {
        "name" : "lastName",
        "required" : true
      }, {
        "name" : "role",
        "required" : true
      } ]
    },
    "partiallyUpdateEmployee" : {
      "method" : "patch",
      "contentType" : "",
      "properties" : [ {
        "name" : "firstName",
        "required" : false
      }, {
        "name" : "lastName",
        "required" : false
      }, {
        "name" : "role",
        "required" : false
      } ]
    }
  }
}
```

查看 HAL-FORMS 规范 以了解 **_templates** 属性的详细信息. 阅读有关 Affordances

API 的信息,以通过这些额外的元数据来扩充您的控制器.

对于单项(EntityModel)和聚合根集合(CollectionModel),Spring HATEOAS使其呈现与 HAL 文档相同.

4.2.1. 定义 HAL-FORMS 元数据

HAL-FORMS 允许描述每个表单字段的标准. Spring HATEOAS
可以通过调整输入和输出类型的模型类型并在其上使用注解来自定义它们.

Attribute	Description
readOnly	如果该属性没有设置方法,则设置为 true.如果存在 ,请在访问器或字段上明确使用 Jackson 的 @JsonProperty(Access.READ_ONLY).默认情况下不呈现,因此默认为 false.
regex	可以通过在字段或类型上使用 JSR-303 的 @Pattern 注解进行自定义 .对于后者,该模式将用于声明为该特定类型的每个属性.默认情况下不呈现.
required	可以使用 JSR-303 的 @NotNull 进行自定义.默认情况下不呈现 ,因此默认为 false.使用 PATCH 作为方法的模板将自动将所有属性设置为不需要.

对于无法手动注解的类型,可以通过应用程序上下文中存在的 HalFormsConfiguration bean注册自定义模式.

```
@Configuration
class CustomConfiguration {

  @Bean
  HalFormsConfiguration halFormsConfiguration() {

    HalFormsConfiguration configuration = new HalFormsConfiguration();
    configuration.registerPatternFor(CreditCardNumber.class, "[0-9]{16}");
  }
}
```

此设置将使 CreditCardNumber 类型的表示模型属性的 HAL-FORMS 模板属性声明值为 [0-9]{16} 的正则表达式字段.

4.2.2. 表单属性的国际化

HAL-FORMS 包含用于人工解释的属性,例如模板的标题或属性提示. 可以使用 Spring 的资源包支持和默认情况下由 Spring HATEOAS 配置的 `rest-messages` 资源包来定义和国际化它们.

Template titles

要定义模板标题,请使用以下模式: `_templates.$affordanceName.title`. 请注意,在 HAL-FORMS 中,如果模板名称是唯一的,则它是默认名称.
这意味着您通常必须使用能力描述的本地或完全限定的输入类型名称来限定键.

Example 34. Defining HAL-FORMS template titles

```
_templates.default.title=Some title ①
_templates.putEmployee.title=Create employee ②
Employee._templates.default.title=Create employee ③
com.acme.Employee._templates.default.title=Create employee ④
```

① 使用默认键作为标题的全局定义.

② 使用实际可得名称作为关键字的标题的全局定义. 除非在创建提供者时明确定义,否则默认为 `$httpMethod + $simpleInputTypeName`.

③ 本地定义的标题将应用于名为 `Employee` 的所有类型.

④ 使用完全限定的类型名称的标题定义.

使用实际可负担名称的密钥优先于默认密钥.

属性提示

还可以通过 Spring HATEOAS 自动配置的 `rest-messages` 资源包来解决属性提示.
这些密钥可以全局,本地或完全定义,并且需要一个 `._prompt` 链接到实际的属性密钥

Example 35. Defining prompts for an email property

```
firstName._prompt=Firstname ①
Employee.firstName._prompt=Firstname ②
com.acme.Employee.firstName._prompt=Firstname ③
```

① 所有名为 `firstName` 的属性都将呈现 `Firstname`,而与声明它们的类型无关.

② 类型为 `Employee` 的 `firstName` 属性将提示 `Firstname`.

③ `com.acme.Employee` 的 `firstName` 属性将获得分配的 `Firstname` 提示.

定义了模板标题和属性提示的示例文档将如下所示:

Example 36. 带有国际化模板标题和属性提示的 *HAL-FORMS* 文档示例

```
{
  …,
  "_templates" : {
    "default" : {
      "title" : "Create employee",
      "method" : "put",
      "contentType" : "",
      "properties" : [ {
        "name" : "firstName",
        "prompt" : "Firstname",
        "required" : true
      }, {
        "name" : "lastName",
        "prompt" : "Lastname",
        "required" : true
      }, {
        "name" : "role",
        "prompt" : "Role",
        "required" : true
      } ]
    }
  }
}
```

4.3. HTTP Problem Details

Problem Details for HTTP APIs 是一种媒体类型，用于在 HTTP 响应中携带机器可读的错误详细信息，以避免需要为 HTTP API 定义新的错误响应格式.

HTTP Problem Details 定义了一组 JSON 属性，这些属性包含其他信息以向 HTTP 客户端描述错误详细信息. 在 RFC document 的相关部分中找到有关这些属性的更多详细信息.

您可以通过在 Spring MVC Controller 中使用 Problem 媒体类型 domain 类型来创建这样的 JSON 响应:

Reporting problem details using Spring HATEOAS' Problem type

```
@RestController
class PaymentController {

    @PutMapping
    ResponseEntity<?> issuePayment(@RequestBody PaymentRequest request) {

        PaymentResult result = payments.issuePayment(request.orderId,
request.amount);

        if (result.isSuccess()) {
            return ResponseEntity.ok(result);
        }

        String title = messages.getMessage("payment.out-of-credit");
        String detail = messages.getMessage("payment.out-of-credit.details", //
                new Object[] { result.getBalance(), result.getCost() });

        Problem problem = Problem.create() ①
                .withType(OUT_OF_CREDIT_URI) //
                .withTitle(title) ②
                .withDetail(detail) //

.withInstance(PAYMENT_ERROR_INSTANCE.expand(result.getPaymentId())) //
                .withProperties(map -> { ③
                    map.put("balance", result.getBalance());
                    map.put("accounts", Arrays.asList( //
                            ACCOUNTS.expand(result.getSourceAccountId()), //
                            ACCOUNTS.expand(result.getTargetAccountId()) //
                    ));
                });

        return ResponseEntity.status(HttpStatus.FORBIDDEN) //
                .body(problem);
    }
}
```

① 首先，使用公开的工厂方法创建 `Problem` 的实例. .

② 您可以定义由媒体类型定义的默认属性的值，例如 使用 Spring 国际化功能的 URI 类型，标题和详细信息（请参见上文）.

③ 可以通过 Map 或显式对象添加自定义属性（请参见下文）.

要将专用对象用于自定义属性，请声明一个类型，创建并填充其实例，然后通过

….withProperties(…) 或在通过 Problem.create(…) 创建实例时将其移入 Problem 实例.

Using a dedicated type to capture extended problem properties

```
class AccountDetails {
    int balance;
    List<URI> accounts;
}

problem.withProperties(result.getDetails());

// or

Problem.create(result.getDetails());
```

这将导致如下所示的响应:

A sample HTTP Problem Details response

```
{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30,
  "accounts": ["/account/12345",
               "/account/67890"]
}
```

4.4. Collection+JSON

Collection+JSON 是向IANA批准的媒体类型 application/vnd.collection+json 注册的JSON规范.

> Collection+JSON is a JSON-based read/write hypermedia-type designed to support management and querying of simple collections.
>
> — Mike Amundsen, Collection+JSON spec

Collection+JSON 提供了一种统一的方式来表示单个 item 资源和集合.要启用此媒体类型

,请将以下配置放入代码中:

Example 37. Collection+JSON enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.COLLECTION_JSON)
public class CollectionJsonApplication {

}
```

此配置将使您的应用程序响应具有 `application/vnd.collection+json` 的 `Accept` 请求头的请求,如下所示.

规格中的以下示例显示了一个 `item`

Example 38. Collection+JSON single item example

```
{
  "collection": {
    "version": "1.0",
    "href": "https://example.org/friends/", ①
    "links": [   ②
      {
        "rel": "feed",
        "href": "https://example.org/friends/rss"
      },
      {
        "rel": "queries",
        "href": "https://example.org/friends/?queries"
      },
      {
        "rel": "template",
        "href": "https://example.org/friends/?template"
      }
    ],
    "items": [  ③
      {
        "href": "https://example.org/friends/jdoe",
        "data": [  ④
          {
            "name": "fullname",
            "value": "J. Doe",
            "prompt": "Full Name"
          },
          {
            "name": "email",
            "value": "jdoe@example.org",
            "prompt": "Email"
          }
        ],
        "links": [ ⑤
          {
            "rel": "blog",
            "href": "https://examples.org/blogs/jdoe",
            "prompt": "Blog"
          },
          {
            "rel": "avatar",
            "href": "https://examples.org/images/jdoe",
            "prompt": "Avatar",
            "render": "image"
```

```
          }
        ]
      }
    ]
  }
}
```

① `self` 链接存储在文档的 `href` 属性.

② 该文档的顶部 `links` 部分包含集合级链接(减去 `self` 链接)

③ `items` 部分包含数据集合,由于这是一个单项文档,因此只有一个条目.

④ `data` 部分包含实际内容. 它由属性组成.

⑤ 该 `items` 的各个 `links`.

先前的片段已从规格中删除. 当 Spring HATEOAS 呈现 `EntityModel` 时,它将:

- 将 `self` 链接置于文档的 `href` 属性和 `items` 级 `href` 属性.
- 将模型的其余链接放在顶级 `links` 和 `items` 级别 `links` 中.
- 从EntityModel中提取属性并将其转换为…

呈现资源集合时,文档几乎相同,除了在 `items` JSON数组项中有多个条目,每个条目一个.

Spring HATEOAS 更具体地将:

- 将整个集合的 `self` 放入顶级 `href` 属性.
- `CollectionModel` 链接(减去 `self`)将放入顶级 `links`.
- 每个 `items` 级别的 `href` 都将包含 `CollectionModel.content` 集合中每个条目的相应 `self` 链接.
- 每个 `items` 级 `links` 将包含 `CollectionModel.content` 每个条目的所有其他链接..

4.5. UBER - 交换代表的统一依据

UBER is an experimental JSON spec

> The UBER document format is a minimal read/write hypermedia type designed to support simple state transfers and ad-hoc hypermedia-based transitions.
>
> — Mike Amundsen, UBER spec

UBER 提供了一种统一的方式来表示单个 item 资源和集合.要启用此媒体类型,请将以下配置放入代码中:

Example 39. UBER+JSON enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.UBER)
public class UberApplication {

}
```

此配置将使您的应用程序使用 Accept 请求头 application/vnd.amundsen-uber+json 响应请求,如下所示:

Example 40. UBER sample document

```
{
  "uber" : {
    "version" : "1.0",
    "data" : [ {
      "rel" : [ "self" ],
      "url" : "/employees/1"
    }, {
      "name" : "employee",
      "data" : [ {
        "name" : "role",
        "value" : "ring bearer"
      }, {
        "name" : "name",
        "value" : "Frodo"
      } ]
    } ]
  }
}
```

此媒体类型以及规范本身仍在开发中.如果您在使用票时遇到问题,请随时 open a ticket.

UBER media type 媒体类型与 **Uber Technologies Inc.** 公司没有任何关联.

4.6. ALPS - Application-Level Profile Semantics

ALPS 是一种媒体类型,用于提供有关另一个资源的基于配置文件的元数据.

> An ALPS document can be used as a profile to explain the application semantics of a document with an application- agnostic media type (such as HTML, HAL, Collection+JSON, Siren, etc.). This increases the reusability of profile documents across media types.
>
> — Mike Amundsen, ALPS spec

ALPS 不需要特殊激活.相反,您可以 "build" Alps 记录,并从 Spring MVC 或 Spring WebFlux Web 方法返回它,如下所示:

Example 41. Building an Alps record

```
@GetMapping(value = "/profile", produces = ALPS_JSON_VALUE)
Alps profile() {

  return Alps.alps() //
      .doc(doc() //
          .href("https://example.org/samples/full/doc.html") //
          .value("value goes here") //
          .format(Format.TEXT) //
          .build()) //
      .descriptor(getExposedProperties(Employee.class).stream() //
          .map(property -> Descriptor.builder() //
              .id("class field [" + property.getName() + "]") //
              .name(property.getName()) //
              .type(Type.SEMANTIC) //
              .ext(Ext.builder() //
                  .id("ext [" + property.getName() + "]") //
                  .href("https://example.org/samples/ext/" +
property.getName()) //
                  .value("value goes here") //
                  .build()) //
              .rt("rt for [" + property.getName() + "]") //

.descriptor(Collections.singletonList(Descriptor.builder().id("embedded").
build())) //
              .build()) //
          .collect(Collectors.toList()))
      .build();
}
```

- 本示例利用 PropertyUtils.getExposedProperties() 提取有关域对象属性的元数据.

此片段已插入测试数据.它会生成如下所示的JSON:

Example 42. ALPS JSON

```
{
  "version": "1.0",
  "doc": {
    "format": "TEXT",
    "href": "https://example.org/samples/full/doc.html",
    "value": "value goes here"
  },
  "descriptor": [
    {
      "id": "class field [name]",
      "name": "name",
      "type": "SEMANTIC",
      "descriptor": [
        {
          "id": "embedded"
        }
      ],
      "ext": {
        "id": "ext [name]",
        "href": "https://example.org/samples/ext/name",
        "value": "value goes here"
      },
      "rt": "rt for [name]"
    },
    {
      "id": "class field [role]",
      "name": "role",
      "type": "SEMANTIC",
      "descriptor": [
        {
          "id": "embedded"
        }
      ],
      "ext": {
        "id": "ext [role]",
        "href": "https://example.org/samples/ext/role",
        "value": "value goes here"
      },
      "rt": "rt for [role]"
    }
  ]
}
```

您可以手动编写它们,而不必将每个字段"自动”链接到域对象的字段. 也可以使用 Spring Framework 的消息包和 MessageSource 界面.
这使您能够将这些值委托给特定于区域设置的消息束,甚至可以使元数据国际化.

4.7. 基于社区的媒体类型

由于能够 创建自己的媒体类型 , 因此社区做出了许多努力来构建其他媒体类型.

4.7.1. JSON:API

- Specification
- Media type designation: application/vnd.api+json
- Latest Release
  - Reference documentation
  - API documentation
- Current Snapshot
  - Reference documentation
  - API documentation
- Source
- Project Lead: Kai Toedter

Maven coordinates

```
<dependency>
    <groupId>com.toedter</groupId>
    <artifactId>spring-hateoas-jsonapi</artifactId>
    <version>{see project page for current version}</version>
</dependency>
```

Gradle coordinates

```
implementation 'com.toedter:spring-hateoas-jsonapi:{see project page for current version}'
```

如果要快照版本，请访问项目页面以获取更多详细信息.

4.7.2. Siren

- Specification
- Media type designation: application/vnd.siren+json
- Reference documentation
- javadocs
- Source
- Project Lead: Ingo Griebsch

Maven coordinates

```
<dependency>
    <groupId>de.ingogriebsch.hateoas</groupId>
    <artifactId>spring-hateoas-siren</artifactId>
    <version>{see project page for current version}</version>
    <scope>compile</scope>
</dependency>
```

Gradle coordinates

```
implementation 'de.ingogriebsch.hateoas:spring-hateoas-siren:{see project page
for current version}'
```

4.8. 注册自定义媒体类型

Spring HATEOAS 允许通过第三方可以实现的一组 SPI 集成对自定义媒体类型的支持.
这样的实现的构造块是:

1. 某种形式的 Jackson ObjectMapper 定制. 在最简单的情况下,这就是 Jackson 模块的实现.
2. LinkDiscoverer 实现,以便客户端支持能够检测生成的表示形式中的链接.
3. 一些配置基础结构将允许 Spring HATEOAS 查找自定义实现并选择其配置.

4.8.1. 自定义媒体类型配置

Spring HATEOAS 通过扫描应用程序上下文中是否有 `HypermediaMappingInformation` 接口的任何实现，来获取自定义媒体类型的实现。每种媒体类型都必须实现此接口，以便：

- 应用于 `WebClient`，`WebTestClient` 或 `RestTemplate` 实例。
- 支持从 Spring Web MVC 和 Spring WebFlux 控制器提供该媒体类型的服务。

定义自己的媒体类型看起来像这样简单：

```
@Configuration
public class MyMediaTypeConfiguration implements
HypermediaMappingInformation {

  @Override
  public List<MediaType> getMediaTypes() {
    return MediaType.parse("application/vnd-acme-media-type") ①
  }

  @Override
  public Module getJacksonModule() {
    return new Jackson2MyMediaTypeModule(); ②
  }

  @Bean
  MyLinkDiscoverer myLinkDiscoverer() {
    return new MyLinkDiscoverer(); ③
  }
}
```

① 配置类返回它支持的媒体类型。这适用于服务器端和客户端方案。.

② 它重写 `getJacksonModule()`
以提供自定义序列化程序来创建特定于媒体类型的表示形式。

③ 它还声明了用于客户端支持的自定义 `LinkDiscoverer` 实现，
以提供进一步的客户端支持。

Jackson 模块通常为表示模型类型 `RepresentationModel`,`EntityModel`,`CollectionModel` 和 `PagedModel` 声明 `Serializer` 和 `Deserializer` 实现。

如果您需要进一步自定义 Jackson 的 ObjectMapper(如自定义的 HandlerInstantiator), 则可以替代重写 configureObjectMapper(…).

之前的参考文档提到了实现 MediaTypeConfigurationProvider 接口并将其注册到 spring.factories 中. 这不是必需的. 该 SPI 仅用于 Spring HATEOAS 提供的现成的媒体类型. 只需要实现 HypermediaMappingInformation 接口并将其注册为 Spring bean.

4.8.2. 建议

实现媒体类型表示的首选方法是通过提供与预期格式匹配的类型层次结构,并且可以由 Jackson 进行序列化. 在为 RepresentationModel 注册的序列化器和反序列化器实现中 ,将实例转换为特定于媒体类型的模型类型,然后为这些实例查找 Jackson 序列化器.

默认情况下支持的媒体类型使用与第三方实现相同的配置机制.因此,值得研究 mediatype 包中的实现. 请注意, 内置媒体类型实现通过 @EnableHypermediaSupport 激活后, 将其配置类包保持私有. 自定义实现可能应该公开这些配置, 以确保用户可以从其应用程序包中导入那些配置类.

Chapter 5. 配置

本节描述了如何配置 Spring HATEOAS.

5.1. 使用 @EnableHypermediaSupport

要使 RepresentationModel 子类型根据各种超媒体表示类型的规范来呈现,可以通过 @EnableHypermediaSupport 激活对特定超媒体表示格式的支持. 注解将 HypermediaType 枚举作为其参数. 当前,我们支持 HAL 以及默认渲染.
使用注解会触发以下内容:

- 它注册了必要的 Jackson 模块,以超媒体特定格式呈现 EntityModel 和 CollectionModel.
- 如果 JSONPath 在类路径上,它将自动注册一个 LinkDiscoverer 实例,并在 JSON 查找 rel 链接(请参阅使用 使用 LinkDiscoverer 实例).
- 默认情况下,它启用 entity links 并自动选择 EntityLinks 实现并将它们捆绑到可以自动装配的 DelegatingEntityLinks 实例中.
- 它会自动拾取 ApplicationContext 中的所有 RelProvider 实现 ,并将它们捆绑到可以自动装配的 DelegatingRelProvider 中. 它注册提供程序以考虑 domain 以及 Spring MVC 控制器上的 @Relation. 如果 EVO inflector 位于类路径上,则使用库中实现的复数算法继承集合 rel 值(请参见 使用 LinkRelationProvider API).

5.1.1. 明确启用对专用 Web 堆栈的支持

默认情况下,@EnableHypermediaSupport 将反射性地检测到您正在使用的 Web 应用程序堆栈,并挂接到为这些组件注册的 Spring 组件中,以启用对超媒体表示的支持. 但是 ,在某些情况下,您只希望明确激活对特定堆栈的支持. 例如. 如果基于 Spring WebMVC 的应用程序使用 WebFlux 的 WebClient 进行传出请求 ,并且该请求不应该与超媒体元素一起使用, 则可以通过在配置中显式声明 WebMVC 来限制要启用的功能:

Example 43. Explicitly activating hypermedia support for a particular web stack

```
@EnableHypermediaSupport(…, stacks = WebStack.WEBMVC)
class MyHypermediaConfiguration { … }
```

Chapter 6. 客户端支持

本节介绍 Spring HATEOAS 对客户端的支持.

6.1. Traverson

Spring HATEOAS 提供了用于客户端服务遍历的 API.它受 Traverson JavaScript library 的启发.以下示例显示了如何使用它:

```
Map<String, Object> parameters = new HashMap<>();
parameters.put("user", 27);

Traverson traverson = new
Traverson(URI.create("http://localhost:8080/api/"), MediaTypes.HAL_JSON);
String name = traverson
    .follow("movies", "movie", "actor").withTemplateParameters(parameters)
    .toObject("$.name");
```

您可以通过将 Traverson 实例指向 REST 服务器并配置要设置为 Accept 请求头的媒体类型来设置它. 然后,您可以定义要发现和遵循的关系名称. 关系名称可以是简单名称,也可以是 JSONPath 表达式(以 $ 开头).

然后,示例将参数映射传递给 Traverson 实例. 这些参数用于扩展遍历过程中找到的 URI(已模板化). 通过访问最终遍历的表示来结束遍历. 在前面的示例中,我们评估一个 JSONPath 表达式来访问演员的名字.

前面的示例是遍历的最简单版本,其中 rel 值是字符串,并且在每个 hop 处都应用相同的模板参数.

在每个级别上,还有更多的选项可以自定义模板参数. 以下示例显示了这些选项.

```
ParameterizedTypeReference<EntityModel<Item>> resourceParameterizedTypeReference
= new ParameterizedTypeReference<EntityModel<Item>>() {};

EntityModel<Item> itemResource = traverson.//
    follow(rel("items").withParameter("projection", "noImages")).//
    follow("$._embedded.items[0]._links.self.href").//
    toObject(resourceParameterizedTypeReference);
```

静态 `rel(…)` 函数是定义 `single Hop` 的便捷方法.使用 `.withParameter(key, value)` 可以简化指定 URI 模板变量的过程.

`.withParameter()` 返回可链接的新 `Hop` 对象.
您可以根据需要将任意多个 `.withParameter` 串在一起. 结果是单个 hop 定义. 以下示例显示了一种方法:

```
ParameterizedTypeReference<EntityModel<Item>>
resourceParameterizedTypeReference = new
ParameterizedTypeReference<EntityModel<Item>>() {};

Map<String, Object> params = Collections.singletonMap("projection",
"noImages");

EntityModel<Item> itemResource = traverson.//
    follow(rel("items").withParameters(params)).//
    follow("$._embedded.items[0]._links.self.href").//
    toObject(resourceParameterizedTypeReference);
```

您还可以使用 `.withParameters(Map)` 加载整个参数 `Map` .

`follow()` 是可链接的,这意味着您可以将多个 hop 串在一起,如前面的示例所示. 您可以放置多个基于字符串的 `rel` 值 (`follow("items", "item")`),也可以放置具有特定参数的单个 hop .

6.1.1. EntityModel<T> vs. CollectionModel<T>

到目前为止显示的示例演示了如何避免 Java 的类型擦除,以及如何将单个 JSON 格式的资源转换为 `EntityModel<Item>` 对象. 但是,如果您得到像 `\_embedded` HAL

集合这样的集合,该怎么办? 只需稍作调整即可完成此操作,如以下示例所示:

```
CollectionModelType<Item> collectionModelType =
    TypeReferences.CollectionModelType<Item>() {};

CollectionModel<Item> itemResource = traverson.//
    follow(rel("items")).//
    toObject(collectionModelType);
```

该资源没有获取单个资源,而是将一个集合反序列化为 `CollectionModel`.

6.2. 使用 `LinkDiscoverer` 实例

使用启用了超媒体的表示形式时,常见的任务是在其中找到具有特定关系类型的链接. Spring HATEOAS 为默认表示渲染或 HAL 提供了 `LinkDiscoverer` 接口的基于 JSONPath 的实现. 使用 `@EnableHypermediaSupport` 时
,我们会自动将支持配置的超媒体类型的实例作为 Spring Bean 暴露.

另外,您可以按以下步骤设置和使用实例:

```
String content = "{'_links' :  { 'foo' : { 'href' : '/foo/bar' }}}";
LinkDiscoverer discoverer = new HalLinkDiscoverer();
Link link = discoverer.findLinkWithRel("foo", content);

assertThat(link.getRel(), is("foo"));
assertThat(link.getHref(), is("/foo/bar"));
```

6.3. 配置 WebClient 实例

如果您需要配置 `WebClient` 来说明超媒体, 这很容易. 需要 `HypermediaWebClientConfigurer`, 如下所示:

Example 44. Configuring a WebClient yourself

```
@Bean
WebClient.Builder hypermediaWebClient(HypermediaWebClientConfigurer
configurer) { ①
 return configurer.registerHypermediaTypes(WebClient.builder()); ②
}
```

① 在你的 `@Configuration` 类中，获取一个 `HypermediaWebClientConfigurer` bean 副本 Spring HATEOAS registers.

② 创建 `WebClient.Builder` 后，使用 configurer 注册 hypermedia types.

`HypermediaWebClientConfigurer` 的功能是向 `WebClient.Builder` 注册所有正确的编码器和解码器. 要使用它，您需要将构建器注入到应用程序中的某个位置，然后运行 `build()` 方法生成 `WebClient`.

如果您使用的是 Spring Boot，则还有另一种方法：`WebClientCustomizer`.

Example 45. Letting Spring Boot configure things

```
@Bean ④
WebClientCustomizer
hypermediaWebClientCustomizer(HypermediaWebClientConfigurer configurer) {
①
    return webClientBuilder -> { ②
        configurer.registerHypermediaTypes(webClientBuilder); ③
    };
}
```

① 当创建一个 Spring bean 时，请获取一个 Spring HATEOAS `HypermediaWebClientConfigurer` bean 的副本.

② 使用 Java 8 lambda 表达式定义 `WebClientCustomizer`.

③ 在函数调用内部，应用 `registerHypermediaTypes` 方法.

④ 将整个内容作为 Spring bean 返回，Spring Boot 可以自动获取并将其应用于其自动配置的 `WebClient.Builder` bean.

在这个阶段，每当需要具体的 `WebClient` 时，只需将 `WebClient.Builder` 注入代码中，然后使用 `build()`. `WebClient` 实例将能够使用超媒体进行交互.

6.4. 配置 WebTestClient 实例

使用启用了超媒体的表示形式时，常见的任务是使用 `WebTestClient` 执行各种测试.

要在测试案例中配置 `WebTestClient` 的实例，请查看以下示例：

Example 46. Configuring WebTestClient when using Spring HATEOAS

```
@Test // #1225
void webTestClientShouldSupportHypermediaDeserialization() {

  // Configure an application context programmatically.
  AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
  context.register(HalConfig.class); ①
  context.refresh();

  // Create an instance of a controller for testing
  WebFluxEmployeeController controller =
context.getBean(WebFluxEmployeeController.class);
  controller.reset();

  // Extract the WebTestClientConfigurer from the app context.
  HypermediaWebTestClientConfigurer configurer =
context.getBean(HypermediaWebTestClientConfigurer.class);

  // Create a WebTestClient by binding to the controller and applying the
hypermedia configurer.
  WebTestClient client =
WebTestClient.bindToApplicationContext(context).build().mutateWith(configu
rer); ②

  // Exercise the controller.
  client.get().uri("http://localhost/employees").accept(HAL_JSON) //
      .exchange() //
      .expectStatus().isOk() //
      .expectBody(new
TypeReferences.CollectionModelType<EntityModel<Employee>>() {}) ③
      .consumeWith(result -> {
        CollectionModel<EntityModel<Employee>> model =
result.getResponseBody(); ④

        // Assert against the hypermedia model.

assertThat(model.getRequiredLink(IanaLinkRelations.SELF)).isEqualTo(Link.o
f("http://localhost/employees"));
        assertThat(model.getContent()).hasSize(2);
      });
}
```

① 注册使用了 `@EnableHypermediaSupport` 注解的配置类来开启 HAL 支持.

② 使用 `HypermediaWebTestClientConfigurer` 可以应用超媒体支持.

③ 使用 Spring HATEOAS 的 `TypeReferences.CollectionModelType` helper 请求 `CollectionModel<EntityModel<Employee>>` 响应.

④ 在获得 Spring HATEOAS 格式的 "body" 之后, 对其进行断言!

`WebTestClient` 是不可变的值类型, 因此您不能更改它. . `HypermediaWebClientConfigurer` 的返回值必须捕获才能使用.

如果您使用的是 Spring Boot, 则还有其他选项, 例如:

Example 47. Configuring WebTestClient when using Spring Boot

```
@SpringBootTest
@AutoConfigureWebTestClient ①
class WebClientBasedTests {

    @Test
    void exampleTest(@Autowired WebTestClient.Builder builder, @Autowired
HypermediaWebTestClientConfigurer configurer) { ②
        client = builder.apply(configurer).build(); ③

        client.get().uri("/") //
                .exchange() //
                .expectBody(new TypeReferences.EntityModelType<Employee>()
{}) ④
                .consumeWith(result -> {
                    // assert against this EntityModel<Employee>!
                });
    }
}
```

① 这是 Spring Boot 的测试注解，它将为该测试类配置 WebTestClient.Builder.

② 自动将 Spring Boot 的 WebTestClient.Builder 插入到 builder 并将 Spring HATEOAS 的 configurer 作为方法参数. .

③ 使用 HypermediaWebTestClientConfigurer 注册对超媒体的支持.

④ 表示您希望使用 TypeReferences 返回 EntityModel<Employee> .

同样，您可以使用与前面的示例类似的断言.

还有许多其他方式来测试案例. WebTestClient 可以绑定到控制器，函数和 URL. 本部分并不表示所有内容. 相反，这为您提供了一些入门示例. 重要的是，通过应用HypermediaWebTestClientConfigurer，可以更改 WebTestClient 的任何实例以处理超媒体.

6.5. 配置 RestTemplate 实例

如果要创建自己的 RestTemplate 副本（配置为使用超媒体），则可以使用

HypermediaRestTemplateConfigurer:

Example 48. Configuring RestTemplate yourself

```
/**
 * Use the {@link HypermediaRestTemplateConfigurer} to configure a {@link
RestTemplate}.
 */
@Bean
RestTemplate hypermediaRestTemplate(HypermediaRestTemplateConfigurer
configurer) { ①
    return configurer.registerHypermediaTypes(new RestTemplate()); ②
}
```

① 在 @Configuration 类中，获取 HypermediaRestTemplateConfigurer bean Spring HATEOAS registers 的副本.

② 创建 RestTemplate 之后，使用配置器应用超媒体类型.

您可以自由地将此模式应用于所需的任何 RestTemplate 实例，无论是创建注册的 bean 还是在定义的服务内部.

如果您使用的是 Spring Boot，则还有另一种方法.

通常，Spring Boot 摆脱了在应用程序上下文中注册 RestTemplate bean的概念.

- 当与不同的服务通信时，您通常需要不同的凭据.
- 当 RestTemplate 使用底层连接池时，您会遇到其他问题.
- 用户通常需要不同的实例，而不是单一的 bean.

为了弥补这一点，Spring Boot 提供了 RestTemplateBuilder. 这个自动配置的 bean 让您可以定义各种用于构造 RestTemplate 实例的 bean. 您请求RestTemplateBuilder bean，执行其 build() 方法，然后应用 final 设置(如凭据等).

注册基于超媒体的消息转换器，添加以下代码:

Example 49. Letting Spring Boot configure things

```
@Bean ④
RestTemplateCustomizer
hypermediaRestTemplateCustomizer(HypermediaRestTemplateConfigurer
configurer) { ①
    return restTemplate -> { ②
        configurer.registerHypermediaTypes(restTemplate); ③
    };
}
```

① 创建 Spring Bean 时，请获取 Spring HATEOAS 的 `HypermediaRestTemplateConfigurer` Bean的副本.

② 使用 Java 8 lambda 表达式定义 `RestTemplateCustomizer`.

③ 在函数调用内部，应用 `registerHypermediaTypes` 方法.

④ 将整个内容作为 Spring bean 返回，Spring Boot 可以自动获取并将其应用于其自动配置的 `RestTemplateBuilder`.

在这个阶段，只要您需要一个具体的 `RestTemplate`，只需将 `RestTemplateBuilder` 注入到代码中，并使用 `build()`. `RestTemplate` 实例将能够使用超媒体进行交互.