**Web API高级应用-路由**

* **什么是路由?**

路由Route是Web API框架响应用户的URI路径请求，并将请求导航到对应的Action的过程。因此，我们可以通过配置路由，来达到对客户端的访问控制。

**NOTE:**

**路由对字母的大小写不敏感。**

* **路由有哪些类型的配置？**
* 路由模板

routes.MapHttpRoute(

name: "API Default",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

* 属性路由

[Route("books/{id}/author")]

public string GeAuthorById(int id) { ... }

* **同一个项目可以同时使用路由模板和属性路由吗？**

可以。路由模板对于项目中的有效范围是全部的控制器Controllers，可以说，它是全局的。但由于它的全局性，便限制了它的灵活性。因此，便有了属性路由。属性路由的配置是在控制器内部的，因此，它可以针对控制器的业务逻辑做专一的路由控制，而不影响其他控制器。

* **如何使路由生效？**

我们可以在配置Web API对象的时候调用Routes.MapHttpRoute方法来注册路由模板，调用类 System.Web.Http.HttpConfigurationExtensions的扩展方法 MapHttpAttributeRoutes使能属性路由。

* 在WebApiConfig.cs文件中注册路由：

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Attribute routing.

config.MapHttpAttributeRoutes();

// Convention-based routing.

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

}

* 在Service服务的启动事件中使能路由：

protected void OnStart()

{

// Pass a delegate to the Configure method.

GlobalConfiguration.Configure(WebApiConfig.Register);

}

* **HTTP Methods**

Web API可以基于HTTP的请求方式（GET、POST、DELETE、PUT）来刷选Action。默认情况下，Web API在不区分字母大小写的情况下通过控制器的Action名称(以GET、POST、DELETE、PUT为开头)来匹配Action。例如，控制器有一个名为“PutBooks”的Action能够匹配HTTP的PUT请求。

我们通过添加附加属性可以为Action匹配任意的HTTP请求：

* [HttpDelete]
* [HttpGet]
* [HttpHead]
* [HttpOptions]
* [HttpPatch]
* [HttpPost]
* [HttpPut]

下面的Action可以通过POST请求进行导航：

[HttpPost]

public HttpResponseMessage CreateBook(Book book) {}

针对其他HTTP请求，包括非标准化的请求，可以通过属性AcceptVerbs进行添加：

[AcceptVerbs("MKCOL","DELETE")]

public void MakeCollection() { }

* **路由属性**

我们可以向控制器的Action添加Route属性来设置该Action匹配的路由：

public class BooksController : ApiController

{

[Route("api/books/{id}")]

public Book FindBookById(int id){}

[Route("api/books/all")]

public Book[] PostAll(){}

}

* **路由前缀**

上面的BooksController定义的两个Action添加的路由属性都有相同的部分“api/books”，我们可以向类添加路由前缀属性RoutePrefix来省略相同的部分：

[RoutePrefix("api/books")]

public class BooksController : ApiController

{

[Route("")]

public IEnumerable<Book> Get() {}

[Route("{id}")]

public Book FindBookById(int id){}

[Route("all")]

public Book[] PostAll(){}

}

由于RoutePrefix是全局的，因此默认情况下所有的Action都会被应用该路由前缀。当然，我们通过符号“~”取消RoutePrefix的作用：

[Route("~/api/authors/{id:int}/books")]

public IEnumerable<Book> GetById(int id) {}

当然，路由前缀可以包含占位符变量：

[RoutePrefix("api/author/{id:int}/books/")]

public class BooksController : ApiController

{

[Route("")]

public IEnumerable<Book> GetById(int id) {}

}

* **路由约束**

RouteConstraints可以让我们在路由模板中规定参数的使用范围，其语法格式为“{Parameter:constraint}”：

[Route("api/books/{id:int}")]

public Book GetBookById(int id) {}

[Route("api/books/{author}")]

public User GetBookByAuthor(string author) {}

上面的例子中，第一个Action只有在URI路径相对应的占位符上是个整型字符串才能被匹配，否则，第二个Action将被匹配。

下面是常见的路由约束：

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| alpha | Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z) | {x:alpha} |
| bool | Matches a Boolean value. | {x:bool} |
| datetime | Matches a **DateTime** value. | {x:datetime} |
| decimal | Matches a decimal value. | {x:decimal} |
| double | Matches a 64-bit floating-point value. | {x:double} |
| float | Matches a 32-bit floating-point value. | {x:float} |
| guid | Matches a GUID value. | {x:guid} |
| int | Matches a 32-bit integer value. | {x:int} |
| length | Matches a string with the specified length or within a specified range of lengths. | {x:length(6)} {x:length(1,20)} |
| long | Matches a 64-bit integer value. | {x:long} |
| max | Matches an integer with a maximum value. | {x:max(10)} |
| maxlength | Matches a string with a maximum length. | {x:maxlength(10)} |
| min | Matches an integer with a minimum value. | {x:min(10)} |
| minlength | Matches a string with a minimum length. | {x:minlength(10)} |
| range | Matches an integer within a range of values. | {x:range(10,50)} |
| regex | Matches a regular expression. | {x:regex(^\d{3}-\d{3}-\d{4}$)} |

我们可以对同一个Action设置多个约束：

[Route("api/books/{id:int:min(1)}")]

public Book GetBookById(int id) {}

* 自定义路由约束

以一个非零的整型约束为例，通过下面的步骤，我们可以自定义路由约束：

* 自定义一个继承接口IHttpRouteConstraint的类：

public class NonZeroConstraint : IHttpRouteConstraint

{

public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName, IDictionary<string, object> values, HttpRouteDirection routeDirection)

{

object value;

if (values.TryGetValue(parameterName, out value) && value != null)

{

long longValue;

if (value is long)

{

longValue = (long)value;

return longValue != 0;

}

string valueString = Convert.ToString(value, CultureInfo.InvariantCulture);

if (Int64.TryParse(valueString, NumberStyles.Integer,

CultureInfo.InvariantCulture, out longValue))

{

return longValue != 0;

}

}

return false;

}

}

* 向WebApiConfig.cs文件中注册路由：

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Convention-based routing.

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

//Register Custom RouteConstraint

var constraintResolver = new

DefaultInlineConstraintResolver();

constraintResolver.ConstraintMap.Add("nonzero", typeof(NonZeroConstraint));

config.MapHttpAttributeRoutes(constraintResolver);

}

}

* 使用自定义路由约束：

[Route("{id:nonzero}")]

public HttpResponseMessage GetNonZero(int id) {}

* **配置可选的占位符参数和缺省值**

我们可以添加“?”字符来让URI路径对应的占位符参数可选，但必须设置默认值：

[Route("api/books/{id:int? ")]

public Book GetBookById(int id = 0) {}

或者

[Route("api/books/{id:int = 0")]

public Book GetBookById(int id) {}

* **路由名称**

在Web API在，每一个路由都有一个名称。路由名称对于产生链接是非常有用的。我们可以向HTTP响应添加基于路由名称的链接。为了让每一个路由都有自己的名称，我们可以通过名称属性Name赋予路由专一的名称：

public class BooksController : ApiController

{

[Route("api/books/{id}", Name="GetBookById")]

public Bookt GetBook(int id) {}

[Route("api/books")]

public HttpResponseMessage Post(Book book)

{

// Validate and add book to database (not shown)

var response = Request.CreateResponse(HttpStatusCode.Created);

// Generate a link to the new book and set the Location header in the response.

string uri = Url.Link("GetBookById", new { id = book.BookId });

response.Headers.Location = new Uri(uri);

return response;

}

}

* **路由匹配顺序**

对于相同的URI路由路径的Action，.NET框架会按照以下的规则进行顺序匹配：

1. 比较Action路由顺序属性的值，值越小，优先级越高。默认的值为0，即最高级。
2. 在顺序属性优先级相同的情况下，框架会根据URI占位符的参数进行优先级排序，下面的优先级由高到低排序：

* 字面声明
* 带约束的占位参数变量
* 不带约束的占位参数变量
* 带约束的占位通配符参数变量
* 不带约束的占位通配符参数变量

**以下面例子为例：**

[RoutePrefix("orders")]

public class OrdersController : ApiController

{

[Route("{id:int}")] // constrained parameter

public HttpResponseMessage Get(int id) {}

[Route("details")] // literal

public HttpResponseMessage GetDetails() {}

[Route("pending", RouteOrder = 1)]

public HttpResponseMessage GetPending() {}

[Route("{customerName}")] // unconstrained parameter

public HttpResponseMessage GetByCustomer(string customerName) {}

[Route("{\*date:datetime}")] // wildcard

public HttpResponseMessage Get(DateTime date) {}

}

路由顺序如下：

1. orders/details
2. orders/{id}
3. orders/{customerName}
4. orders/{\*date}
5. orders/pending

即便pending是字面声明，但它的路由顺序属性设置为1，高于其他的默认值0，因此它的优先级最低。