

10 | 数据绑定：如何自动转换传入的参数？

2023-04-03 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。今天我们继续手写 MiniSpring，这节课我们讨论传入参数的转换问题。

上节课，我们已经基本完成了对 Dispatcher 的扩展，用 HandlerMapping 来处理映射关系，用 HandlerAdapter 来处理映射后具体方法的调用。


在处理请求的过程中，我们用 ServletRequest 接收请求参数，而获取参数用的是 `getParameter()` 方法，它的返回值是 String 字符串，这也意味着无论是获取字符串参数、数字参数，还是布尔型参数，它获取到的返回值都是字符串。而如果要把请求参数转换成 Java 对象，就需要再处理，那么每一次获取参数后，都需要显式地编写大量重复代码，把 String 类型的参数转换成其他类型。显然这不符合我们对框架的期望，我们希望框架能帮助我们自动处理这些常规数据格式的转换。

再扩大到整个访问过程，后端处理完毕后，返回给前端的数据再做返回，也存在格式转换的问题，传入传出两个方向我们都要处理。而这节课我们讨论的重点是“传入”方向。

传入参数的绑定

我们先考虑传入方向的问题：请求参数怎么和 Java 对象里的属性进行自动映射？

这里，我们引入 `WebDataBinder` 来处理。这个类代表的是一个内部的目标对象，用于将 `Request` 请求内的字符串参数转换成不同类型的参数，来进行适配。所以比较自然的想法是这个类里面要持有一个目标对象 `target`，然后还要定义一个 `bind()` 方法，通过来绑定参数和目标对象，这是 `WebDataBinder` 里的核心。


 复制代码

```
1     public void bind(HttpServletRequest request) {
2         PropertyValues mpvs = assignParameters(request);
3         addBindValues(mpvs, request);
4         doBind(mpvs);
5     }
```

通过 `bind` 方法的实现，我们可以看出，它主要做了三件事。

1. 把 `Request` 里的参数解析成 `PropertyValues`。
2. 把 `Request` 里的参数值添加到绑定参数中。
3. 把两者绑定在一起。

你可以看一下 `WebDataBinder` 的详细实现。

 复制代码

```
1     package com.minis.web;
2
3     import java.util.Map;
4     import javax.servlet.http.HttpServletRequest;
5     import com.minis.beans.PropertyValues;
6     import com.minis.util.WebUtils;
7
8     public class WebDataBinder {
9         private Object target;
10        private Class<?> clz;
11        private String objectName;
12        public WebDataBinder(Object target) {
```

```

13         this(target, "");
14     }
15     public WebDataBinder(Object target, String targetName) {
16         this.target = target;
17         this.objectName = targetName;
18         this.clz = this.target.getClass();
19     }
20     //核心绑定方法，将request里面的参数值绑定到目标对象的属性上
21     public void bind(HttpServletRequest request) {
22         PropertyValues mpvs = assignParameters(request);
23         addBindValues(mpvs, request);
24         doBind(mpvs);
25     }
26     private void doBind(PropertyValues mpvs) {
27         applyPropertyValues(mpvs);
28     }
29     //实际将参数值与对象属性进行绑定的方法
30     protected void applyPropertyValues(PropertyValues mpvs) {
31         getPropertyAccessor().setPropertyValues(mpvs);
32     }
33     //设置属性值的工具
34     protected BeanWrapperImpl getPropertyAccessor() {
35         return new BeanWrapperImpl(this.target);
36     }
37     //将Request参数解析成PropertyValues
38     private PropertyValues assignParameters(HttpServletRequest request) {
39         Map<String, Object> map = WebUtils.getParametersStartingWith(request, "");
40         return new PropertyValues(map);
41     }
42     protected void addBindValues(PropertyValues mpvs, HttpServletRequest request)
43     }
44 }

```

从这个实现方法里可以看出，先是调用了 `assignParameters()`，把 Request 里的参数换成内存里的一个 map 对象，这一步用到了底层的 WebUtils 工具类，这个转换对我们来说比较简单。而最核心的方法是 `getPropertyAccessor().setPropertyValues(mpvs);`，这个 `getPropertyAccessor` 则是内置了一个 `BeanWrapperImpl` 对象，内部包含了 target。由名字可以看出它是 Bean 的包装实现类，把属性 map 绑定到目标对象上去。

有了这个大流程，我们再来探究一下一个具体的参数是如何转换的，我们知道 Request 的转换都是从字符串转为其他类型，所以我们可以定义一个通用接口，名叫 `PropertyEditor`，内部提供一些方法可以让字符串和 Object 之间进行双向灵活转换。

```

1 package com.minis.beans;
2
3 public interface PropertyEditor {
4     void setAsText(String text);
5     void setValue(Object value);
6     Object getValue();
7     Object getAsText();
8 }

```

现在我们来定义两个 PropertyEditor 的实现类：CustomNumberEditor 和 StringEditor，分别处理 Number 类型和其他类型，并进行类型转换。你可以看一下 CustomNumberEditor 的相关源码。

```

1 package com.minis.beans;
2
3 import java.text.NumberFormat;
4 import com.minis.util.NumberUtils;
5 import com.minis.util.StringUtils;
6
7 public class CustomNumberEditor implements PropertyEditor{
8     private Class<? extends Number> numberClass; //数据类型
9     private NumberFormat numberFormat; //指定格式
10    private boolean allowEmpty;
11    private Object value;
12    public CustomNumberEditor(Class<? extends Number> numberClass, boolean allowE
13        this(numberClass, null, allowEmpty);
14    }
15    public CustomNumberEditor(Class<? extends Number> numberClass, NumberFormat n
16        this.numberClass = numberClass;
17        this.numberFormat = numberFormat;
18        this.allowEmpty = allowEmpty;
19    }
20    //将一个字符串转换成number赋值
21    public void setAsText(String text) {
22        if (this.allowEmpty && !StringUtils.hasText(text)) {
23            setValue(null);
24        }
25        else if (this.numberFormat != null) {
26            // 给定格式
27            setValue(NumberUtils.parseNumber(text, this.numberClass, this.numberFormat)
28        }
29        else {

```


```

30     setValue(NumberUtils.parseNumber(text, this.numberClass));
31 }
32 }
33 //接收Object作为参数
34 public void setValue(Object value) {
35     if (value instanceof Number) {
36         this.value = (NumberUtils.convertNumberToTargetClass((Number) value,
37     }
38     else {
39         this.value = value;
40     }
41 }
42 public Object getValue() {
43     return this.value;
44 }
45 //将number表示成格式化串
46 public Object getAsText() {
47     Object value = this.value;
48     if (value == null) {
49         return "";
50     }
51     if (this.numberFormat != null) {
52         // 给定格式.
53         return this.numberFormat.format(value);
54     }
55     else {
56         return value.toString();
57     }
58 }
59 }

```

整体实现也比较简单，在内部定义一个名为 value 的域，接收传入的格式化 text 或者 value 值。如果遇到的值是 Number 类型的子类，比较简单，就进行强制转换。这里我们用到了一个底层工具类 NumberUtils，它提供了一个 NumberUtils.parseNumber(text, this.numberClass, this.numberFormat) 方法，方便我们在数值和文本之间转换。

你可以看下 StringEditor 实现的相关源代码。

 复制代码

```

1 package com.minis.beans;
2
3 import java.text.NumberFormat;
4 import com.minis.util.NumberUtils;

```




```

5 import com.minis.util.StringUtils;
6
7 public class StringEditor implements PropertyEditor{
8     private Class<String> strClass;
9     private String strFormat;
10    private boolean allowEmpty;
11    private Object value;
12    public StringEditor(Class<String> strClass,
13                        boolean allowEmpty) throws IllegalArgumentException {
14        this(strClass, "", allowEmpty);
15    }
16    public StringEditor(Class<String> strClass,
17                        String strFormat, boolean allowEmpty) throws IllegalArgum
18        this.strClass = strClass;
19        this.strFormat = strFormat;
20        this.allowEmpty = allowEmpty;
21    }
22    public void setAsText(String text) {
23        setValue(text);
24    }
25    public void setValue(Object value) {
26        this.value = value;
27    }
28    public String getAsText() {
29        return value.toString();
30    }
31    public Object getValue() {
32        return this.value;
33    }
34 }

```

StringEditor 的实现类就更加简单了，因为它是字符串本身的处理，但它的构造函数有些不一样，支持传入字符串格式 strFormat，这也是为后续类型转换格式留了一个“口子”。

有了两个基本类型的 Editor 作为工具，现在我们再来看关键的类 BeanWapperImpl 的实现。

 复制代码

```

1 package com.minis.web;
2
3 import java.lang.reflect.Field;
4 import java.lang.reflect.InvocationTargetException;
5 import java.lang.reflect.Method;

```

```

6 import com.minis.beans.PropertyEditor;
7 import com.minis.beans.PropertyEditorRegistrySupport;
8 import com.minis.beans.PropertyValue;
9 import com.minis.beans.PropertyValues;
10
11 public class BeanWrapperImpl extends PropertyEditorRegistrySupport {
12     Object wrappedObject; //目标对象
13     Class<?> clz;
14     PropertyValues pvs; //参数值
15     public BeanWrapperImpl(Object object) {
16         registerDefaultEditors(); //不同数据类型的参数转换器editor
17         this.wrappedObject = object;
18         this.clz = object.getClass();
19     }
20     public void setBeanInstance(Object object) {
21         this.wrappedObject = object;
22     }
23     public Object getBeanInstance() {
24         return wrappedObject;
25     }
26     //绑定参数值
27     public void setPropertyValues(PropertyValues pvs) {
28         this.pvs = pvs;
29         for (PropertyValue pv : this.pvs.getPropertyValues()) {
30             setPropertyValue(pv);
31         }
32     }
33     //绑定具体某个参数
34     public void setPropertyValue(PropertyValue pv) {
35         //拿到参数处理器
36         BeanPropertyHandler propertyHandler = new BeanPropertyHandler(pv.getName(
37             //找到对该参数类型的editor
38             PropertyEditor pe = this.getDefaultEditor(propertyHandler.getPropertyClz(
39             //设置参数值
40             pe.setAsText((String) pv.getValue());
41             propertyHandler.setValue(pe.getValue());
42         }
43         //一个内部类，用于处理参数，通过getter()和setter()操作属性
44         class BeanPropertyHandler {
45             Method writeMethod = null;
46             Method readMethod = null;
47             Class<?> propertyClz = null;
48             public Class<?> getPropertyClz() {
49                 return propertyClz;
50             }
51             public BeanPropertyHandler(String propertyName) {
52                 try {
53                     //获取参数对应的属性及类型
54                     Field field = clz.getDeclaredField(propertyName);


```

```

55         propertyClz = field.getType();
56         //获取设置属性的方法, 按照约定为setXxxx ()
57         this.writeMethod = clz.getDeclaredMethod("set" +
58         propertyName.substring(0, 1).toUpperCase() + propertyName.substring(1), prope
59         //获取读属性的方法, 按照约定为getXxxx ()
60         this.readMethod = clz.getDeclaredMethod("get" +
61         propertyName.substring(0, 1).toUpperCase() + propertyName.substring(1), prope
62         } catch (Exception e) {
63             e.printStackTrace();
64         }
65         //调用getter读属性值
66         public Object getValue() {
67             Object result = null;
68             writeMethod.setAccessible(true);
69             try {
70                 result = readMethod.invoke(wrappedObject);
71             } catch (Exception e) {
72                 e.printStackTrace();
73             }
74             return result;
75         }
76         //调用setter设置属性值
77         public void setValue(Object value) {
78             writeMethod.setAccessible(true);
79             try {
80                 writeMethod.invoke(wrappedObject, value);
81             } catch (Exception e) {
82                 e.printStackTrace();
83             }
84         }
85     }
86 }

```

这个类的核心在于利用反射对 Bean 属性值进行读写，具体是通过 setter 和 getter 方法。但具体的实现，则有赖于继承的 PropertyEditorRegistrySupport 这个类。我们再来看看 PropertyEditorRegistrySupport 是如何实现的。

 复制代码

```

1 package com.minis.beans;
2
3 import java.math.BigDecimal;
4 import java.math.BigInteger;
5 import java.util.HashMap;
6 import java.util.LinkedHashMap;
7 import java.util.Map;

```



```


8 public class PropertyEditorRegistrySupport {
9     private Map<Class<?>, PropertyEditor> defaultEditors;
10    private Map<Class<?>, PropertyEditor> customEditors;
11    //注册默认的转换器editor
12    protected void registerDefaultEditors() {
13        createDefaultEditors();
14    }
15    //获取默认的转换器editor
16    public PropertyEditor getDefaultEditor(Class<?> requiredType) {
17        return this.defaultEditors.get(requiredType);
18    }
19    //创建默认的转换器editor, 对每一种数据类型规定一个默认的转换器
20    private void createDefaultEditors() {
21        this.defaultEditors = new HashMap<>(64);
22        // Default instances of collection editors.
23        this.defaultEditors.put(int.class, new CustomNumberEditor(Integer.class,
24        this.defaultEditors.put(Integer.class, new CustomNumberEditor(Integer.class,
25        this.defaultEditors.put(long.class, new CustomNumberEditor(Long.class, fa
26        this.defaultEditors.put(Long.class, new CustomNumberEditor(Long.class, tr
27        this.defaultEditors.put(float.class, new CustomNumberEditor(Float.class,
28        this.defaultEditors.put(Float.class, new CustomNumberEditor(Float.class,
29        this.defaultEditors.put(double.class, new CustomNumberEditor(Double.class
30        this.defaultEditors.put(Double.class, new CustomNumberEditor(Double.class
31        this.defaultEditors.put(BigDecimal.class, new CustomNumberEditor(BigDecir
32        this.defaultEditors.put(BigInteger.class, new CustomNumberEditor(BigInteg
33        this.defaultEditors.put(String.class, new StringEditor(String.class, true
34    }
35    //注册客户化转换器
36    public void registerCustomEditor(Class<?> requiredType, PropertyEditor proper
37        if (this.customEditors == null) {
38            this.customEditors = new LinkedHashMap<>(16);
39        }
40        this.customEditors.put(requiredType, propertyEditor);
41    }
42    //查找客户化转换器
43    public PropertyEditor findCustomEditor(Class<?> requiredType) {
44        Class<?> requiredTypeToUse = requiredType;
45        return getCustomEditor(requiredTypeToUse);
46    }
47    public boolean hasCustomEditorForElement(Class<?> elementType) {
48        return (elementType != null && this.customEditors != null && this.customE
49    }
50    //获取客户化转换器
51    private PropertyEditor getCustomEditor(Class<?> requiredType) {
52        if (requiredType == null || this.customEditors == null) {
53            return null;
54        }
55        PropertyEditor editor = this.customEditors.get(requiredType);
56

```

```
57         return editor;
58     }
59 }
60
```

从这段源码里可以看到，PropertyEditorRegistrySupport 的核心实现是 createDefaultEditors 方法，它里面内置了大量基本类型或包装类型的转换器 Editor，还定义了可以定制化的转换器 Editor，这也是 WebDataBinder 能做不同类型转换的原因。不过我们目前的实现，只支持数字和字符串几个基本类型的转换，暂时不支持数组、列表、map 等格式。

现在，我们已经实现了一个完整的 WebDataBinder，用来绑定数据。我们接下来将提供一个 WebDataBinderFactory，能够更方便、灵活地操作 WebDataBinder。

 复制代码

```
1 package com.minis.web;
2
3 import javax.servlet.http.HttpServletRequest;
4
5 public class WebDataBinderFactory {
6     public WebDataBinder createBinder(HttpServletRequest request, Object target,
7         WebDataBinder wbd = new WebDataBinder(target, objectName);
8         initBinder(wbd, request);
9         return wbd;
10    }
11    protected void initBinder(WebDataBinder dataBinder, HttpServletRequest request) {
12    }
13 }
```

有了上面一系列工具之后，我们看怎么使用它们进行数据绑定。从前面的讲解中我们已经知道，这个 HTTP Request 请求最后会找到映射的方法上，也就是通过 RequestMappingHandlerAdapter 里提供的 handleInternal 方法，来调用 invokeHandlerMethod 方法，所以我们从这个地方切入，改造 invokeHandlerMethod 方法，实现参数绑定。

 复制代码

```
1     protected void invokeHandlerMethod(HttpServletRequest request,
```

```

2 HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
3     WebDataBinderFactory binderFactory = new WebDataBinderFactory();
4     Parameter[] methodParameters =
5     handlerMethod.getMethod().getParameters();
6     Object[] methodParamObjs = new Object[methodParameters.length];
7     int i = 0;
8     //对调用方法里的每一个参数，处理绑定
9     for (Parameter methodParameter : methodParameters) {
10         Object methodParamObj = methodParameter.getType().newInstance();
11         //给这个参数创建WebDataBinder
12         WebDataBinder wdb = binderFactory.createBinder(request,
13 methodParamObj, methodParameter.getName());
14         wdb.bind(request);
15         methodParamObjs[i] = methodParamObj;
16         i++;
17     }
18     Method invocableMethod = handlerMethod.getMethod();
19     Object returnObj = invocableMethod.invoke(handlerMethod.getBean(), method
20 response.getWriter().append(returnObj.toString());
21 }

```

在 `invokeHandlerMethod` 方法的实现代码中，`methodParameters` 变量用来存储调用方法的所有参数，针对它们进行循环，还有一个变量 `methodParamObj`，是一个新创建的空对象，也是我们需要进行绑定操作的目标，`binderFactory.createBinder` 则是创建了 `WebDataBinder`，对目标对象进行绑定。整个循环结束之后，`Request` 里面的参数就绑定了调用方法里的参数，之后就可以被调用。


我们从这个绑定过程中可以看到，循环过程就是按照参数在方法中出现的次序逐个绑定的，所以这个次序是很重要的。

客户化转换器

现在我们已经实现了 `Request` 数据绑定过程，也提供了默认的 `CustomNumberEditor` 和 `StringEditor`，来进行数字和字符串两种类型的转换，从而把 `ServletRequest` 里的请求参数转换成 `Java` 对象里的数据类型。但这种默认的方式比较固定，如果你希望转换成自定义的类型，那么原有的两个 `Editor` 就没办法很好地满足需求了。

因此我们要继续探讨，如何支持自定义的 `Editor`，让我们的框架具有良好的扩展性。其实上面我们看到 `PropertyEditorRegistrySupport` 里，已经提前准备好了客户化转换器的地方，你

可以看下代码。

 复制代码

```
1 public class PropertyEditorRegistrySupport {
2     private Map<Class<?>, PropertyEditor> defaultEditors;
3     private Map<Class<?>, PropertyEditor> customEditors;
```


我们利用客户化 Editor 这个“口子”，新建一个部件，把客户自定义的 Editor 注册进来就可以了。

我们先在原有的 WebDataBinder 类里，增加 registerCustomEditor 方法，用来注册自定义的 Editor，你可以看一下相关代码。

 复制代码

```
1 public void registerCustomEditor(Class<?> requiredType, PropertyEditor propertyEd
2     getPropertyAccessor().registerCustomEditor(requiredType, propertyEditor);
3 }
```

在这里，可以自定义属于我们自己的 CustomEditor，比如在 com.test 包路径下，自定义 CustomDateEditor，这是一个自定义的日期格式处理器，来配合我们的测试。

 复制代码

```
1 package com.test;
2
3 import java.text.NumberFormat;
4 import java.time.LocalDate;
5 import java.time.LocalDateTime;
6 import java.time.ZoneId;
7 import java.time.format.DateTimeFormatter;
8 import java.util.Date;
9 import com.minis.beans.PropertyEditor;
10 import com.minis.util.NumberUtils;
11 import com.minis.util.StringUtils;
12
13 public class CustomDateEditor implements PropertyEditor {
14     private Class<Date> dateClass;
15     private DateTimeFormatter datetimeFormatter;
16     private boolean allowEmpty;
```


```

17     private Date value;
18     public CustomDateEditor() throws IllegalArgumentException {
19         this(Date.class, "yyyy-MM-dd", true);
20     }
21     public CustomDateEditor(Class<Date> dateClass) throws IllegalArgumentException
22         this(dateClass, "yyyy-MM-dd", true);
23     }
24     public CustomDateEditor(Class<Date> dateClass,
25         boolean allowEmpty) throws IllegalArgumentException {
26         this(dateClass, "yyyy-MM-dd", allowEmpty);
27     }
28     public CustomDateEditor(Class<Date> dateClass,
29         String pattern, boolean allowEmpty) throws IllegalArgumentException {
30         this.dateClass = dateClass;
31         this.datetimeFormatter = DateTimeFormatter.ofPattern(pattern);
32         this.allowEmpty = allowEmpty;
33     }
34     public void setAsText(String text) {
35         if (this.allowEmpty && !StringUtils.hasText(text)) {
36             setValue(null);
37         }
38         else {
39             LocalDate localdate = LocalDate.parse(text, datetimeFormatter);
40             setValue(Date.from(localdate.atStartOfDay(ZoneId.systemDefault()).toInstant
41         )
42     }
43     public void setValue(Object value) {
44         this.value = (Date) value;
45     }
46     public String getAsText() {
47         Date value = this.value;
48         if (value == null) {
49             return "";
50         }
51         else {
52             LocalDate localDate = value.toInstant().atZone(ZoneId.systemDefault()).toLocalDate
53             return localDate.format(datetimeFormatter);
54         }
55     }
56     public Object getValue() {
57         return this.value;
58     }
59 }

```


程序也比较简单，用 `DateTimeFormatter` 来转换字符串和日期就可以了。

接下来我们定义一个 `WebBindingInitializer`，其中有一个 `initBinder` 实现方法，为自定义的 `CustomEditor` 注册做准备。

 复制代码

```
1 public interface WebBindingInitializer {
2     void initBinder(WebDataBinder binder);
3 }
```

下面，我们再实现 `WebBindingInitializer` 接口，在实现方法 `initBinder` 里，注册自定义的 `CustomDateEditor`，你可以看下相关代码。

 复制代码

```
1 package com.test;
2
3 import java.util.Date;
4 import com.minis.web.WebBindingInitializer;
5 import com.minis.web.WebDataBinder;
6
7 public class DateInitializer implements WebBindingInitializer{
8     public void initBinder(WebDataBinder binder) {
9         binder.registerCustomEditor(Date.class, new CustomDateEditor(Date.class, "yyyy
10     }
11 }
12
```

通过上述实现可以看到，我们自定义了“yyyy-MM-dd”这样一种日期格式，也可以根据具体业务需要，自定义其他日期格式。

然后，我们要使用它们，回到 `RequestMappingHandlerAdapter` 这个类里，新增 `WebBindingInitializer` 的属性定义，调整原有的 `RequestMappingHandlerAdapter(WebApplicationContext wac)` 这个构造方法的具体实现，你可以看下调整后的代码。

 复制代码

```
1 public RequestMappingHandlerAdapter(WebApplicationContext wac) {
2     this.webBindingInitializer = (WebBindingInitializer)
```



```
3 this.wac.getBean("webBindingInitializer");
4 }
```

其实也就是增加了 webBindingInitializer 属性的设置。

然后再利用 IoC 容器，让这个构造方法，支持用户通过 applicationContext.xml 配置 webBindingInitializer，我们可以在 applicationContext.xml 里新增下面这个配置。

```
1 <bean id="webBindingInitializer" class="com.test.DateInitializer">
2                                     </bean>
```

[复制代码](#)

最后我们只需要在 BeanWrapperImpl 实现类里，修改 setPropertyValue(PropertyValue pv) 这个方法的具体实现，把最初我们直接获取 DefaultEditor 的代码，改为先获取 CustomEditor，如果它不存在，再获取 DefaultEditor，你可以看下相关实现。

```
1 public void setPropertyValue(PropertyValue pv) {
2     BeanPropertyHandler propertyHandler = new BeanPropertyHandler(pv.getName(
3     PropertyEditor pe = this.getCustomEditor(propertyHandler.getPropertyClz())
4     if (pe == null) {
5         pe = this.getDefaultEditor(propertyHandler.getPropertyClz());
6     }
7
8     pe.setAsText((String) pv.getValue());
9     propertyHandler.setValue(pe.getValue());
10 }
```

[复制代码](#)

改造后，就能支持用户自定义的 CustomEditor，增强了扩展性。同样的类型，如果既有用户自定义的实现，又有框架默认的实现，那用户自定义的优先。

到这里，传入参数的处理问题我们就探讨完了。

小结

这节课，我们重点探讨了 MVC 里前后端参数的自动转换，把 Request 里的参数串自动转换成调用方法里的参数对象。

为了完成传入参数的自动绑定，我们使用了 WebDataBinder，它内部用 BeanWrapperImpl 对象，把属性值的 map 绑定到目标对象上。绑定的过程中，要对每一种数据类型分别进行格式转换，对基本的标准数据类型，由框架给定默认的转换器，但是对于别的数据类型或者是文化差异很大的数据类型，如日期型，我们可以通过 CustomEditor 机制让用户自定义。

通过数据的自动绑定，我们不用再通过 request.getParameter() 方法手动获取参数值，再手动转成对象了，这些 HTTP 请求里的参数值就自动变成了后端方法里的参数对象值，非常便利。实际上后面我们会看到，这种两层之间的数据自动绑定和转换，在许多场景中都非常有用，比如 Jdbc Template。所以这节课的内容需要你好好消化，灵活运用。

完整源代码参见 <https://github.com/YaleGuo/minis>

课后题

学完这节课的内容，我也给你留一道思考题。我们现在的实现是把 Request 里面的参数值，按照内部的次序隐含地自动转成后台调用方法参数对象中的某个属性值，那么可不可以使用一个手段，让程序员手动指定某个调用方法的参数跟哪个 Request 参数进行绑定呢？欢迎你在留言区和我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (10)



lmnsds

2023-05-16 来自北京

WebDataBinder类决定了本文的效果：

- controller函数只能是一个自定义对象类型，且只能有一个参数
- request中的参数不能名不能是参数对象中没有的成员名称，否则会报错

作者回复: 对。



Imnsds

2023-05-16 来自北京

`http://localhost:8080/test4?name=yourname&id=2&birthday=2023-05-16`

可以使用如上url进行测试

"/test4"对应的controller method 要以User为参数。User是定义在test.entity下的类。



风轻扬

2023-04-15 来自北京

这一讲，学到了以下知识点：

- 1)、如何将请求参数封装到实体类的字段上
- 2)、用接口，不要用实现类，增加扩展性

Spring框架真是应了现在很流行的一句服务标语：把困难留给自己，把方便带给客户。

另外，有2个问题，请教老师：

- 1)、为什么要翻译成客户化转换器呢？翻译成自定义转换器是不是更容易理解一点？
- 2)、BeanWrapperImpl中的getValue方法中,是一个笔误吗？应该是readMethod.setAccessible吧？另外,正常情况下,对外提供的set、get方法都是public的,不需要setAccessible为true了吧？

作者回复: 翻译问题各自有习惯，你的翻译挺好。其实术语用英文原文更好，我线下课所有术语名词都是英语。

setAccessible是笔误，copy后没改。是不是public都用一下是个习惯。

共 2 条评论 >



袁帅

2023-04-10 来自北京

@RequestMapping("/test3")

```
public String doTest3(Integer a) {}
```

参数是Integer methodParameter.getType().newInstance(); 这行会报错啊

java.lang.NoSuchMethodException: java.lang.Integer.<init>()

作者回复: 简单类型是有问题的，你看一下复杂对象的测试。简单类型和多参数，都是线下班的扩展

练习。



不是早晨，就是黄昏

2023-04-09 来自河南

所以我们应该怎么测试呢，在浏览器中输入的地址中参数应该是什么，应该给以下测试程序，来梳理整个流程，否则很乱

作者回复：感谢建议，我们后面把测试写得明细一点。



梦某人

2023-04-07 来自河北

目前这个似乎并不能完成对基本类型的转换，反而似乎要处理的是复合类型的转换？当然也可能是我代码和理解存在问题。

目前逻辑上是：Adapter 中对参数进行处理，对于每个参数都有一个 WebDataBinder 进行处理，而这个类在做类型绑定的时候，主要是通过 BeanWrapperImpl 类来进行处理，此时，每个 WebDataBinder 和 BeanWrapperImpl 内的 clazz 指向的都是这个参数的类，基本类型在这里会是一个 String、Long 之类的。在 BeanWrapperImpl 的 setPropertyValue 方法中，主要是借助于由请求转换而来的 PropertyValue 类，这个 PropertyValue 主要有 name 和 value 是请求中的请求名和参数，并调用了 BeanPropertyHandler 以 PropertyValue 的 name 值进行处理。BeanPropertyHandler 首先根据请求名找到这个请求参数的类里面对应名称的 Field，再根据 Field 获取对应的 clazz，然后使用 Editor 的 getValue 来进行类型转换，使用 set 方法进行赋值，然后使用对应属性的 get 进行取值操作。

但是基本类型没有 setString 和 getString 之类的情况。。。。所以他反而没办法处理相对基础的类型。

解决方案应该是考虑配置默认类型的 writemethod 和 readmethod，，优化 BeanPropertyHandler 类。

现在给我的感觉就是卡在了两头中间，类型转换可以完成，也能从请求中取到值，可以根据方法的参数列表构建对应的类型，但是中间基础类型的绑定的这一块是卡住无法处理的，就是基本类型是有问题的。

最后是思考题，如果要处理顺序问题，那么应该是在 adapter 中处理方法参数上的标记，根据注解或者标记来调整顺序。比如设置一个注解，指定同一类型的不同参数的名称，或者是指定顺序。然后在 adapter 中拿到方法后，根据注解重排这个方法的参数列表顺序。

作者回复: 你的思考是对的, 目前处理不了简单类型, 也处理不了多参数。前几年的线下课这是作为学院扩展练习布置的。你考虑到了, 想得很细, 学思练, 一定会收益很大。



C.

2023-04-04 来自江苏

```
this.webBindingInitializer = (WebBindingInitializer) this.webApplicationContext.getBean("webBindingInitializer");
```

这段代码错误的原因是上一章节的DispatcherServlet文件

```
protected void refresh() {  
    // 初始化 controller  
    initController();  
  
    initHandlerMappings(this.webApplicationContext);  
    initHandlerAdapters(this.parentApplicationContext);  
}
```

这个地方传递的容器不正确导致的, 因为这两个容器一个是配置文件applicationContext.xml解析bean容器, 一个是包扫描controller的bean容器。而webBindingInitializer这个bean的定义在applicationContext.xml配置文件中, 所以传入webApplicationContext这个容器对象是获取不到的。改为parentApplicationContext就可以正确执行下去。可能后面章节还会改, 但是本章节的内容结束出现了这个问题。

作者回复: 赞! 这是我喜欢的教学相长。学手艺就要这么眼脑手结合在一起, 自己动手, 而不是简单听讲, 光看光听是学不会的。你的这个问题, 后面都有改。



C.

2023-04-04 来自江苏

```
this.webBindingInitializer = (WebBindingInitializer) this.wac.getBean("webBindingInitializer");
```

这段代码在这个章节结束执行报错, No bean,而且出现了这个情况getBean了好多次Hello WorldBean。

```
webBindingInitializer bean created. com.minis.test.DateInitializer : com.minis.test.DateInitializer@305bb0d  
handle properties for bean: webBindingInitializer
```

bean registered..... webBindingInitializer
Context Refreshed...
com.minis.test.controller.HelloWorldBean bean created. com.minis.test.controller.HelloWorldBean : com.minis.test.controller.HelloWorldBean@4852cf8d
handle properties for bean: com.minis.test.controller.HelloWorldBean
bean registered..... com.minis.test.controller.HelloWorldBean
[2023-04-04 02:29:06,605] 工件 mini-spring:war: 工件已成功部署
[2023-04-04 02:29:06,605] 工件 mini-spring:war: 部署已花费 553 毫秒
com.minis.test.controller.HelloWorldBean bean created. com.minis.test.controller.HelloWorldBean : com.minis.test.controller.HelloWorldBean@71e1b60b
handle properties for bean: com.minis.test.controller.HelloWorldBean
bean registered..... com.minis.test.controller.HelloWorldBean
com.minis.test.controller.HelloWorldBean bean created. com.minis.test.controller.HelloWorldBean : com.minis.test.controller.HelloWorldBean@4ab383d7
handle properties for bean: com.minis.test.controller.HelloWorldBean
bean registered..... com.minis.test.controller.HelloWorldBean
com.minis.test.controller.HelloWorldBean bean created. com.minis.test.controller.HelloWorldBean : com.minis.test.controller.HelloWorldBean@6962cf89
handle properties for bean: com.minis.test.controller.HelloWorldBean
bean registered..... com.minis.test.controller.HelloWorldBean
com.minis.test.controller.HelloWorldBean bean created. com.minis.test.controller.HelloWorldBean : com.minis.test.controller.HelloWorldBean@1256dc01
handle properties for bean: com.minis.test.controller.HelloWorldBean
bean registered..... com.minis.test.controller.HelloWorldBean



马儿

2023-04-04 来自四川

感觉这节课的内容还是有点难懂.目前自己梳理了一下逻辑。大概有以下几个问题，希望老师能抽空解答一下

- 1.现在的代码似乎不能够解析基本类型，只能解析复杂类型。按照代码逻辑应该是将所有属性绑定到一个复杂类型中去，如果方法参数是基本类型就会报错NoSuchField
- 2.如果是复杂类型controller中每个参数都通过createbinder创建了WebDataBinder，但是WebDataBinder#BeanWrapperImpl每次都重新创建了BeanWrapperImpl对象，导致初始注册的CustomEditor在后续注册的时候并没有生效。

作者回复: 用心了，高质量问题，你的理解都是对的。对基本类型的支持和多参数支持，这是扩展练习。MiniSpring是供学习用的，当时的线下班需要学员过程中补充更多特性。这就是我强调的眼脑手

结合。我一直认为，编程是手工活儿，有点像学木匠，站在一边看师傅打家具是不能真掌握的这门手艺的，要自己动手。赞你。



门窗小二

2023-04-03 来自福建

增加一个类似requestParam的注解

作者回复: 对。不过要进一步想想在程序哪个地方修改才能支持这个注解。

