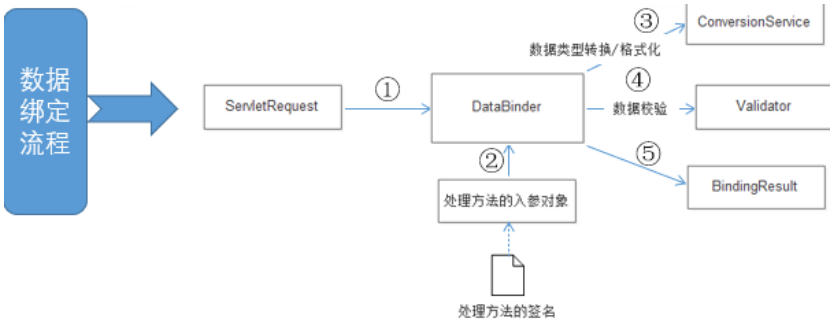
1. 绑定说明

> *在执行程序时，Spring MVC会根据客户端请求参数的不同，将请求消息中的信息以一定的方式转换并绑定到控制器类的方法参数中。这种将请求消息数据与后台方法参数建立连接的过程就是Spring MVC中的数据绑定。*
>
> *Spring MVC 中弯沉数据绑定的方式：*
> *在数据绑定过程中，Spring MVC框架会通过数据绑定组件（DataBinder）将请求参数串的内容进行类型转换，然后将转换后的值赋给控制器类中方法的形参，这样后台方法就可以正确绑定并获取客户端请求携带的参数了。接下来，将通过一张数据流程图来介绍数据绑定的过程。*



> 1. *Spring MVC将ServletRequest对象传递给DataBinder；*
> 2. *将处理方法的入参对象传递给DataBinder；*
> 3. *DataBinder调用ConversionService组件进行数据类型转换、数据格式化等工作，并将ServletRequest对象中的消息填充到参数对象中；*
> 4. *调用Validator组件对已经绑定了请求消息数据的参数对象进行数据合法性校验；*
> 5. *校验完成后会生成数据绑定结果BindingResult对象，Spring MVC会将BindingResult对象中的内容赋给处理方法的相应参数。*

1.1 绑定的机制

我们都知道，表单中请求参数都是基于 key=value 的。 SpringMVC 绑定请求参数的过程是通过把表单提交请求参数，作为控制器中方法参数进行绑定的。 例如：

```
<a href="account/findAccount?accountId=10">查询账户</a>
```

中请求参数是：

```
accountId=10
```

Controller实现：

```java
/**
* 查询账户
* @param accountId
* @return
*/
@RequestMapping("/findAccount")
public String findAccount(Integer accountId) {
    System.out.println("查询了账户。。。。"+accountId);
    return "success";
}
```

1.2 支持的数据类型

- 基本类型参数： 包括基本类型和 String 类型
- POJO 类型参数： 包括实体类，以及关联的实体类
- 数组和集合类型参数： 包括 List 结构和 Map 结构的集合（包括数组）

SpringMVC 绑定请求参数是自动实现的，但是要想使用，必须遵循使用要求。

1.3 使用要求

- 如果是基本类型或者 String 类型： 要求我们的参数名称和控制器中方法的形参名称保持一致(严格区分大小写)；可以使用@RequestParam 注解解决不一致问题。
- 如果是 POJO 类型（记作M），或者它的关联对象（记作N）： 要求表单中参数名称和 POJO 类的属性名称保持一致。并且控制器方法的参数类型是 POJO 类型（在表单中请求的参数如果是N类中的属性，那么这个属性的名称为：【对象.属性】，其中【对象】要和M类中的N类型的字段名一致，【属性】要和N类中属性名称一致）。
- 如果是集合类型,有两种方式：
  第一种： 要求集合类型的请求参数必须包装在 POJO中。在表单中请求参数名称要和 POJO 中集合属性名称相同。 给 List 集合中的元素赋值，使用下标。 给 Map 集合中的元素赋值，使用键值对。
  第二种： 接收的请求参数是 json 格式数据。需要借助一个注解实现。
- 数组类型
  示例：
  表单

```
<!--3个复选框的name属性值和类型均相同-->
<body>
    <form
action="${pageContext.request.contextPath
}/deleteUsers" method="post">
        <table width="20%" border=1>
            <tr><td>选择</td><td>用户名</td></tr>
            <tr><td><input name="ids" value="1"
type="checkbox"></td><td>tom</td></tr>
            <tr><td><input name="ids" value="2"
type="checkbox"></td><td>jack</td></tr>
            <tr><td><input name="ids" value="3"
type="checkbox"></td><td>lucy</td></tr>
        </table>
        <input type="submit" value="删除"/>
    </form>
</body>
```

```
//...省略向用户列表页面跳转方法
@RequestMapping("/deleteUsers")
public String deleteUsers(Integer[] ids) {
    if(ids !=null){
        for (Integer id : ids)
{System.out.println("删除了id为"+id+"的用户！");}
    }else{System.out.println("ids=null");}
    return "success";
}
```

注意: 它还可以实现一些数据类型自动转换。内置转换器全都在：
org.springframework.core.convert.support 包下。
有：
java.lang.Boolean -> java.lang.String : ObjectToStringConverter
java.lang.Character -> java.lang.Number : CharacterToNumberFactory
java.lang.Character -> java.lang.String : ObjectToStringConverter
java.lang.Enum -> java.lang.String : EnumToStringConverter
java.lang.Number -> java.lang.Character : NumberToCharacterConverter
java.lang.Number -> java.lang.Number : NumberToNumberConverterFactory
java.lang.Number -> java.lang.String : ObjectToStringConverter
java.lang.String -> java.lang.Boolean : StringToBooleanConverter
java.lang.String -> java.lang.Character : StringToCharacterConverter
java.lang.String -> java.lang.Enum : StringToEnumConverterFactory
java.lang.String -> java.lang.Number : StringToNumberConverterFactory
java.lang.String -> java.util.Locale : StringToLocaleConverter
java.lang.String -> java.util.Properties : StringToPropertiesConverter
java.lang.String -> java.util.UUID : StringToUUIDConverter
java.util.Locale -> java.lang.String : ObjectToStringConverter
java.util.Properties -> java.lang.String : PropertiesToStringConverter
java.util.UUID -> java.lang.String : ObjectToStringConverter
如遇特殊类型转换要求，需要我们自己编写自定义类型转换器。

1.4 @RequestParam注解

属性	说明
value	name 属性的别名，这里指参数的名字，即入参的请求参数名字，如 value="item_id"表示请求的参数中名字为 item_id 的参数的值将传入。如果只使用 vaule 属性，则可以省略 value 属性名。
name	指定请求头绑定的名称。
required	用于指定参数是否必须，默认是 true，表示请求中一定要有相应的参数。
defaultValue	默认值，表示如果请求中没有同名参数时的默认值。

使用示例：先用@RequestParam接收同名参数，后间接绑定到方法形参上

```
@RequestMapping("/selectUser")
public String
selectUser(@RequestParam(value="user_id")Integer id) {
     System.out.println("id="+id);
     return "success";
}
```

1.5 @PathParam注解（RESTful中）

略，到时候看相应的API 总结以下

类似于RequestParam，获取get请求中的参数；
但是在另一篇博客中有这样的例子（类似于PathVariable）：

```
url对应：localhost:8080/introduction/1/2/3/4/5

//在后端中接受模板变量
@Path("/introduction/{bookId}/{gg}/{version}/{plarform}/{v
ps}")
```

https://blog.csdn.net/xmh594603296/article/details/79566986

https://blog.csdn.net/u011410529/article/details/66974974

1.6 @PathVariable注解

指示方法参数应绑定到 URI 模板变量的注解。

Modifier and Type	Optional Element and Description
String	name The name of the path variable to bind to.
boolean	required Whether the path variable is required.
String	value Alias for name () .

示例：

```
<a href="http://localhost:8080/springmvc/hello/101"></a>
```

```
@RequestMapping("/hello/{id}")
    public String getDetails(@PathVariable(value="id")
String id){
    System.ou.println(id);
}
```

2. 使用示例

2.1 基本类型和 String 类型作为参数

jsp 代码：

```
<!-- 基本类型示例     -->
    <a href="account/findAccount?
accountId=10&accountName=zhangsan">查询账户</a>
```

控制器代码：

```
/**
  * 查询账户
  * @return
  */
@RequestMapping("/findAccount")
public String findAccount(Integer accountId,String
accountName) {
    System.out.println("查询了账
户。。。。"+accountId+","+accountName);
    return "success";
}
```

2.2 POJO 类型作为参数

实体类代码：

```
public class Account implements Serializable {

    private static final long serialVersionUID =
8405232260975148534L;

    private Integer id;
    private String name;
    private Float money;
    private Address address;

    //注意这个类，默认的有一个无参数的构造方法
    public Account() {

    }

    public Account(Integer id, String name, Float money) {
        this.id = id;
        this.name = name;
        this.money = money;
    }
    //getters and setters
}

public class Address implements Serializable {
      private String provinceName;
      private String cityName;
      //getters and setters
}
```

jsp 代码：

```
<form action="/account/saveAccount" method="get">
    账户名:<input type="text" name="name"/><br/>
    账户金额:<input type="text" name="money"/><br/>
    账户省份:<input type="text"
name="address.provinceName"/><br/>
    账户城市:<input type="text" name="address.cityName"/>
<br/>
    <input type="submit" value="存钱"/>
</form>
```

控制器代码：

```
/**
     * 保存账户
     * @param account
     * @return
     */
@RequestMapping("/saveAccount")
public String saveAccount(Account account) {
    System.out.println("保存了账户。。。。"+account);
    return "success";
}
```

2.3 POJO 类中包含集合类型参数

实体类代码：

```
/**
用户实体类
*/
public class User implements Serializable
{
    private String username;
    private String password;
    private Integer age;
    private List<Account> accounts;
    private Map<String, Account> accountMap;

    //getters and setters
    @Override
    public String toString()
    {
        return "User [username=" + username + ",
password=" +
            password + ", age="
            + age + ",\n accounts=" + accounts
            + ",\n accountMap=" + accountMap + "]";
    }
}
```

jsp 代码：

```
<!-- POJO 类包含集合类型演示    -->
<form action="account/updateAccount" method="post">
    用户名称：<input type="text" name="username"><br/>
    用户密码：<input type="password" name="password"><br/>
    用户年龄：<input type="text" name="age"><br/>
    账户 1 名称：<input type="text"
name="accounts[0].name"><br/>
```

```
    账户 1 金额：<input type="text"
name="accounts[0].money"><br/>
    账户 2 名称：<input type="text"
name="ac`counts[1].name"><br/>
    账户 2 金额：<input type="text"
name="accounts[1].money"><br/>
    账户 3 名称：<input type="text"
name="accountMap['one'].name"><br/>
    账户 3 金额：<input type="text"
name="accountMap['one'].money"><br/>
    账户 4 名称：<input type="text"
name="accountMap['two'].name"><br/>
    账户 4 金额：<input type="text"
name="accountMap['two'].money"><br/>
    <input type="submit" value="保存">
</form>
```

控制器代码：

```
/**
     更新账户
     @return
     */
@RequestMapping("/updateAccount")
public String updateAccount(User user)
{
    System.out.println("更新了账户。。。。" + user);
    return "success";
}
```

2.4 请求参数乱码问题

post 请求方式： 在 web.xml 中配置一个过滤器

```
<!-- 配置 springMVC 编码过滤器 针对POST请求乱码问题 -->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilt
er</filter-class>
    <!-- 设置过滤器中的属性值 -->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <!-- 启动过滤器 -->
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
```

```
<!-- 过滤所有请求 -->
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

get请求方式： tomcat 对 GET和POST请求处理方式是不同的,GET请求的编码问题，要改tomcat的server.xml 配置文件，如下：

```
<Connector connectionTimeout="20000" port="8080"
           protocol="HTTP/1.1" redirectPort="8443"/>
```

改为:

```
<Connector connectionTimeout="20000" port="8080"
           protocol="HTTP/1.1" redirectPort="8443"
           useBodyEncodingForURI="true"/>
```

如果遇到 ajax 请求仍然乱码，请把：

```
useBodyEncodingForURI="true"    改为    URIEncoding="UTF-8"
```

即可。

2.5 静态资源过滤问题

背景：在web.xml中写了 拦截所有请求，当然包括了静态资源，所以页面需要引用css或js的话，该请求也会被拦截——>示例：

在 springmvc 的配置文件中可以配置，静态资源不过滤：

```
<!-- 添加注解驱动 -->
<mvc:annotation-driven/>
<!--
通过mvc:resources设置静态资源，这样servlet就会处理这些静态资源，而不通过控制器
设置不过滤内容，比如:css,js,img 等资源文件
location指的是本地的真是路径，mapping指的是映射到的虚拟路径。
-->

<!-- location 表示路径，mapping 表示文件，**表示该目录下的文件以及子目录的文件-->
<mvc:resources location="/css/" mapping="/css/**"/>
<mvc:resources location="/images/" mapping="/images/**"/>
<mvc:resources location="/scripts/"
mapping="/javascript/**"/>
```

https://blog.csdn.net/qq_40594137/article/details/79112700

https://www.cnblogs.com/heliusKing/p/11405986.html

3. 自定义类型转换器

3.1 使用场景

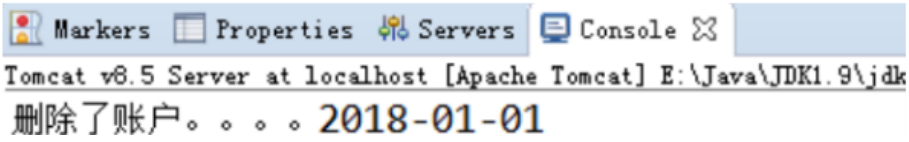
jsp代码：

```
<!--特殊情况类型转换问题-->
<a href="account/deleteAccount?date=2018-01-01">根据日期删除账户</a>
```

控制器代码：

```
/**
     * 删除账户
     * @param date
     * @return
     */
@RequestMapping("/deleteAccount")
public String deleteAccount(String date){
    System.out.println("删除日期是:"+date);
    return "success";
}
```

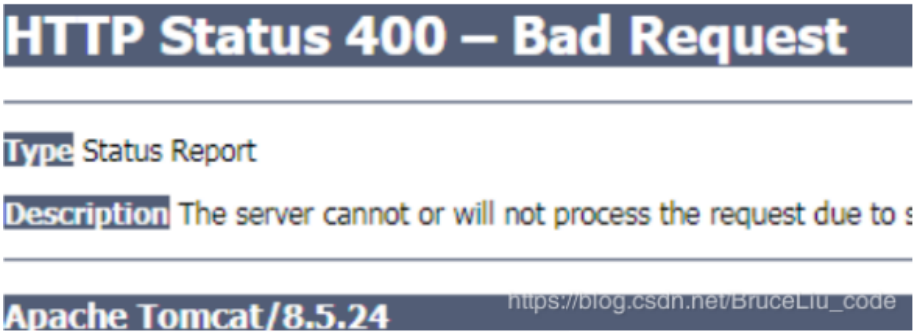
运行结果：



当我们把控制器中方法参数的类型改为 Date 时：

```
/**
     * 删除账户
     * @param date
     * @return
     */
@RequestMapping("/deleteAccount")
public String deleteAccount(Date date){
    System.out.println("删除日期是:"+date);
    return "success";
}
```

运行结果：



异常提示：Failed to bind request element: org.springframework.web.method.annotation.MethodArgumentTypeMismatchException: Failed to convert value of type 'java.lang.String' to required type 'java.util.Date'; nested exception is org.springframework.core.convert.ConversionFailedException: Failed to convert from type [java.lang.String] to type [java.util.Date] for value '2018-01-01'; nested exception is java.lang.IllegalArgumentException

3.2 使用步骤

第一步：定义一个类，实现 Converter 接口（org.springframework.core.convert.converter包下），该接口有两个泛型。

```
public class StringToDateConverter implements
Converter<String,Date> {

    /**
     * 字符串转日期
     * @param source
     * @return
     */
    @Override
    public Date convert(String source) {
        DateFormat format = null;
        try {
            if(StringUtils.isEmpty(source)) {
                throw new NullPointerException("请输入要转换
的日期");
            }
            format = new SimpleDateFormat("yyyy-MM-dd");
            Date date = format.parse(source);
            return date;
        } catch (Exception e) {
            throw new RuntimeException("输入日期有误");
        }
    }
}
```

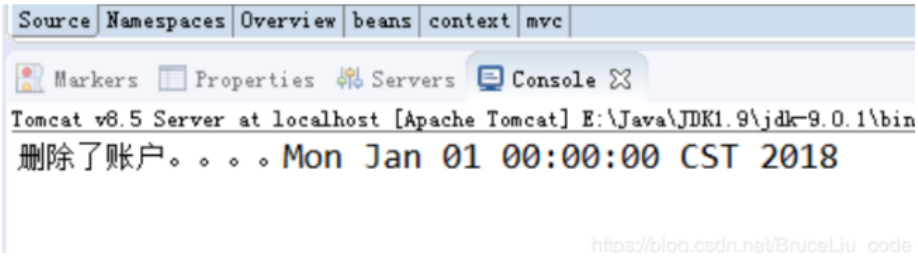
第二步：在 spring 配置文件中配置类型转换器。 spring 配置类型转换器的机制是，将自定义的转换器注册到类型转换服务中去。

```
<!-- 配置类型转换器工厂 -->
<bean id="converterService"
class="org.springframework.context.support.ConversionServi
ceFactoryBean">
    <!-- 给工厂注入一个新的类型转换器 -->
    <property name="converters">
        <array>
            <!-- 配置自定义类型转换器 -->
            <bean
class="com.bruceliu.converter.StringToDateConverter"/>
        </array>
    </property>
</bean>
```

第三步：在 annotation-driven 标签中引用配置的类型转换服务

```
<!-- 引用自定义类型转换器 -->
<mvc:annotation-driven conversion-
service="converterService"/>
```

运行结果：



4. 使用 ServletAPI 对象作为方法参数

SpringMVC 还支持使用原始 ServletAPI 对象作为控制器方法的参数。支持原始

ServletAPI 对象有：

```
1  HttpServletRequest
2  HttpServletResponse
3  HttpSession
4  java.security.Principal
5  Locale
6  InputStream
7  OutputStream
8  Reader
9  Writer
```

我们可以把上述对象，直接写在控制的方法参数中使用。部分示例代码：

jsp 代码：

```
<!--原始ServletAPI作为控制器参数-->
<a href="account/testServletAPI">测试访问 ServletAPI</a>
```

控制器中的代码：

```
1  /**
2  * 测试访问    testServletAPI
3  * @return
4  */
5  @RequestMapping("/testServletAPI")
6  public String testServletAPI(HttpServletRequest
request,HttpServletResponse response,
7  HttpSession session) {
8      System.out.println(request);
9      System.out.println(response);
10      System.out.println(session);
11      return "success";
12  }
```

执行结果：