什么是SpringEL?

Spring3中引入了Spring表达式语言—SpringEL,SpEL是一种强大,简洁的装配Bean的方式,他可以通过运行期间执行的表达式将值装配到我们的属性或构造函数当中,更可以调用JDK中提供的静态常量,获取外部Properties文件中的的配置

为什么要使用SpringEL?

我们平常通过配置文件或Annotaton注入的Bean,其实都可以称为静态性注入,试想一下,若然我Bean A中有变量A,它的值需要根据Bean B的B变量为参考,在这场景下静态注入就对这样的处理显得非常无力,而Spring3增加的SpringEL就可以完全满足这种需求,而且还可以对不同Bean的字段进行计算再进行赋值,功能非常强大

如何使用SpringEL?

SpringEL从名字来看就能看出,和EL是有点关系的,SpringEL的使用和EL表达式的使用非常相似,EL表达式在JSP页面更方便的获取后台中的值,而SpringEL就是为了更方便获取Spring容器中的Bean的值,EL使用${},而SpringEL使用#{}进行表达式的声明。

使用SpringEL注入简单值

```
public class TestSpringEL {
    /*
     * @Value注解等同于XML配置中的<property/>标签,
     * SpringEL同样支持在XML<property/>中编写
     */
    // 注入简单值,输出num为5
    @Value("#{5}")
    private Integer num;
    // 注入ID为testConstant的Bean
    @Value("#{testConstant}")
    private TestConstant Constant;
    // 注入ID为testConstant Bean中的STR常量/变量
    @Value("#{testConstant.STR}")
    private String str;
}
```

使用SpringEL调用方法

```
public class TestSpringEL {
    /*
     * TestConstant类中有两个方法重载,
     * 返回值为String类型
     */

    // 调用无参方法
    @Value("#{testConstant.showProperty}")
    private String method1;

    // 有参接收字符串的方法
    @Value("#{testConstant.showProperty('Hello')}")
    private String method2;

    /*
     * 若然希望方法返回的String为大写
     */
    @Value("#{testConstant.showProperty().toUpperCase()}")
    private String method3;
```

```
    /*
     * 若使用method3这种方式,若然showProperty返回为null,
     * 将会抛出NullPointerException,可以使用以下方式避免
     */
    @Value("#{testConstant.showProperty()?.toUpperCase}")
    private String method4;

    /*
     * 使用?.符号代表若然左边的值为null,将不执行右边方法,
     * 读者可以灵活运用在其他场景,只要左边可能返回null,
     * 即可使用上面示例中的?.
     */
}
```

SpringEL调用静态类或常量

```
public class TestSpringEL {

    /*
     * 注入JDK中的工具类常量或调用工具类的方法
     */

    // 获取Math的PI常量
    @Value("#{T(java.lang.Math).PI")
    private double pi;

    // 调用random方法获取返回值
    @Value("#{T(java.lang.Math).random()}")
    private double ramdom;

    // 获取文件路径符号
    @Value("#{T(java.io.File).separator}")
    private String separator;
}
```

SpringEL运算

```
public class TestSpringEL {

    /*
     * 使用SpringEL进行运算及逻辑操作
     */

    // 拼接字符串
    @Value("#{testConstant.nickname + ' ' + testConstant.name}")
    private String concatString;

    // 对数字类型进行运算,testConstant拥有num属性
    @Value("#{ 3 * T(java.lang.Math).PI + testConstant.num}")
    private double operation;

    // 进行逻辑运算
    @Value("#{testConstant.num > 100 and testConstant.num <= 200}")
    private boolean logicOperation;

    // 进行或非逻辑操作
    @Value("#{ not testConstant.num == 100 or testConstant.num <= 200}")
    private boolean logicOperation2;

    // 使用三元运算符
    @Value("#{testConstant.num > 100 ? testConstant.num : testConstant.num + 100}")
    private Integer logicOperation3;
```

```
}
```

SpringEL使用正则表达式

```
public class TestSpringEL {

    // 验证是否邮箱地址正则表达式
    @Value("#{testConstant.STR matches '\w+([\.-]?\w+)*@\w+([\.-]?\w+)*
(\.\w{2,3})+'}")
    private boolean regularExpression;
}
```

SpringEL操作集合

```
public class TestSpringEL {

    /*
     * TestConstant类中拥有名为testList的List变量, 和名为testMap的Map
     */

    // 获取下标为0的元素
    @Value("#{testConstant.testList[0]}")
    private String str;

    // 获取下标为0元素的大写形式
    @Value("#{testConstant.testList[0]?.toUpperCase()}")
    private String upperStr;

    // 获取map中key为hello的value
    @Value("#{testConstant.testMap['hello']}")
    private String mapValue;

    // 根据testList下标为0元素作为key获取testMap的value
    @Value("#{testConstant.testMap[testConstant.testList[0]]}")
    private String mapStrByTestList;
}
```

Spring操作外部Properties文件

```
<!-- 首先通过applicaContext.xml中<util:properties>增加properties文件 -->
<!-- 注意需要引入Spring的util schemea命名空间和注意id属性,id属性将在SpringEL中使用 -
->

<util:properties id="test" location="classpath:application.properties"/>
```

```
public class TestSpringEL {

    // 注意test为xml文件中声明的id
    @Value("#{test['jdbc.url']}")
    private String propertiesValue;
}
```

SpringEL查询筛选集合和投影

```
public class TestSpringEL {

    /*
     * 声明City类,有population属性 testContants拥有名叫cityList的City类List集合
     */

    // 过滤testConstant中cityList集合population属性大于1000的全部数据注入到本属性
    @Value("#{testConstant.cityList.?[population > 1000]}")
    private List<City> cityList;
```

```
    // 过滤testConstant中cityList集合population属性等于1000的第一条数据注入到本属性
    @Value("#{testConstant.cityList.^[population == 1000]}")
    private City city;

    // 过滤testConstant中cityList集合population属性小于1000的最后一条数据注入到本属性
    @Value("#{testConstant.cityList.$[population < 1000]}")
    private City city2;

    /*
     * 首先为city增加name属性,代表城市的名称
     */

    /*
     * 假如我们在过滤城市集合后只想保留城市的名称,
     * 可以使用如下方式进行投影
     */
    @Value("#{testConstant.cityList.?[population > 1000].![name]}")
    private List<String> cityName;
}
```

优点:

SpringEL功能非常强大,在Annotation的方式开发时可能感觉并不强烈,因为可以直接编写到源代码来实现SpringEL的功能,但若然是在XML文件中进行配置,SpringEL可以弥补XML静态注入的不足,从而实现更强大的注入

缺点:

SpringEL在使用时仅仅是一个字符串,不易于排错与测试,也没有IDE检查我们的语法,当出现错误时较难检测

笔者实际应用:

笔者开发的项目当中比较频繁的使用SpringEL,例如通过SpringEL获取外部properties中的值,又或者项目当中的数据字典亦是使用SpringEL的一个场景,我们抽象出一个Param类的集合,通过SpringEL集合筛选和投影获取我们想要的字段参数添加到我们的程序逻辑当中(笔者项目中的Spring Security亦使用SpringEL,但本文章不加以叙述)

总结:

Spring3.0让人为之惊艳的非SpringEL莫属,为我们的注入提供了另一种强大的形式,传统注入能做到的事情,和做不到的事情,SpringEL一概能完成,但在项目当中并不适宜大量使用SpringEL,适当的技术方在适当的位置,才能更好的完成事情

```
package SpringEl;

import org.apache.commons.io.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Component;
```

```
    @Configuration
//扫描目录下配置 从java开始如目录为 com.springEl.此处填写 com.springEl
    @ComponentScan("SpringEl")
//读取配置 默认起始地址为 resources
    @PropertySource("test.properties")
    public class BeanSpringEl {
        @Value("Bean") //name = "bean"
        private String name;
        @Value("#{systemProperties['os.name']}") //取操作系统版本
        private String systemName;
        @Value("#{T(java.lang.Math).random() * 100.0}") //随机数计算值
        private Double randomDouble;
        @Value("#{anotherBean.anotherBean}") //取AnotherBean类中，anotherBean元素。
        private String bean;
        @Value("https://www.baidu.com") //访问 百度 地址 获取Html
        private Resource testUrl;
        @Value("test.txt")
        private Resource testFile;//访问 resources地址下的 test.txt;
        @Value("${system.name}")
        private String sysName;//读取 resources地址下 test.properties内的system.name
属性

        @Autowired
        private Environment environment; //获得容器中的环境变量，也可取出
test.properties下的属性

        @Bean //必须配置 使用该方式 取得test.properties下的数据
        public static PropertySourcesPlaceholderConfigurer propertyConfigure(){
            return new PropertySourcesPlaceholderConfigurer();
        }

        public  void say(){
            System.out.println(name);
            System.out.println(systemName);
            System.out.println(randomDouble);
            System.out.println(bean);
            try {
                System.out.println(IOUtils.toString(testUrl.getInputStream()));
                System.out.println(IOUtils.toString(testFile.getInputStream()));
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
            System.out.println(environment.getProperty("system.boot"));
            System.out.println(sysName);
        }
    }
```

Spring表达式语言(简称SpEL)是一种强大的表达式语言，支持在运行时查询和操作对象图。语言语法类似于Unified EL，但提供了额外的功能，最明显的是方法调用和基本字符串模板功能

简单来说，SpEL表达式可以实现调用对象的属性和方法(静态的也可以)，还支持表达式的编写。

SpEL表达式的格式：#{表达式}

```
<?xml version="1.0" encoding="UTF-8"?>
<!--约束文件，切记先有约束后有本地变量  -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
```

```
       xsi:schemaLocation="
    http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util.xsd">

    <bean id="bd2" class="cn.code404.domain.BasicDate"></bean>

    <!--1、SpEL表达式的数据类型 支持  -->
    <bean id="bn1" class="cn.code404.domain.BasicNum">
        <!--SpEL表达式  -->
        <property name="num1" value="#{1}"></property>
        <property name="num2" value="#{10*23.3}"></property>
        <property name="num3" value="#{10/23.3}"></property>
        <property name="num4" value="#{'abc'}"></property>
        <property name="date" value="#{bd2}"></property>
    </bean>

    <!--2、SpEL表达式运算符的支持  -->
    <bean id="bo1" class="cn.code404.domain.BasicOp">
        <!--算术运算符  -->
        <property name="num" value="#{2/3}"></property>
        <!--字符串连接运算符  -->
        <property name="msg" value="#{bd2.name+user1.id}"></property>
        <!-- 比较运算符 -->
        <property name="res" value="#{3 gt 4}"></property>
        <!--三目运算符的支持  -->
        <property name="sex" value="#{user1.id gt 3?'男人':'女孩'}"></property>
        <!-- <property name="name" value="#{bd2.name!=null?bd2.name:'默认值'}">
</property> -->
        <!--简写的三目  -->
        <property name="name" value="#{bd2.name?:'默认值'}"></property>
    </bean>

    <!--3、SpEL表达式的方法的调用  -->
    <bean id="bm1" class="cn.code404.domain.BasicMethod">
        <property name="msg" value="#{'abcdefg'}"></property>
        <!--长度  -->
        <property name="size" value="#{'abcdefg'.length()}"></property>
        <!--转换为大写  -->
        <property name="msgUpper" value="#{'abcdefg'.toUpperCase()}"></property>
        <!--支持方法的链式调用，${}不支持  -->
        <property name="index" value="#{'abcdefg'.toUpperCase().indexOf('E')}">
</property>
        <!--调用实例方法  -->
        <property name="content" value="#{bo1.getTime()}"></property>
        <!--调用静态属性  -->
        <property name="pi" value="#{T(java.lang.Math).PI}"></property>
        <!--调用静态方法  -->
        <property name="isStudy" value="#{T(java.lang.Math).random()*2}"></property>
</property>
        <!--静态方法的传值  -->
        <property name="food" value="#{T(org.qf.domain.BasicOp).food(bo1.name)}">
</property>
    </bean>

    <!--4、使用正则表达式  -->
    <bean id="stu" class="cn.code404.domain.Student">
    <property name="isMatch" value="#{'123a5678' matches '^[0-9]{8}$'}">></property>
    </bean>

    <!--5、对集合的查询操作  -->
    <util:list id="emp_2">
    <bean class="cn.code404.collection.Emp" p:name="小王" p:zw="CEO" p:age="30">
</bean>
```

```
    <bean class="cn.code404.collection.Emp" p:name="大张" p:zw="CEO" p:age="40">
</bean>
    <bean class="cn.code404.collection.Emp" p:name="小白" p:zw="CEO" p:age="50">
</bean>
    <bean class="cn.code404.collection.Emp" p:name="老高" p:zw="CEO" p:age="60">
</bean>
    </util:list>
    <bean class="cn.code404.collection.ElModel" id="el">
    <!--1、返回符合条件的第一个元素  -->
    <!-- <property name="emp" value="#{emp_2.^[age lt 50]}"></property> -->
    <!--2、返回符合条件的最后一个元素  -->
    <property name="emp" value="#{emp_2.$[age lt 50]}"></property>
    <!--3、返回符合条件的多个元素(组成集合返回)  -->
    <property name="emps" value="#{emp_2.?[age gt 30 ]}"></property>
    </bean>

</beans>
```

单元测试类：

```
public class SpELTest {

    //SpEL表达式的数据类型的支持
    @Test
    public void test1(){
        ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext1.xml");
        System.out.println(context.getBean("bn1"));
    }
    //SpEL表达式的运算符的支持
    @Test
    public void test3(){
        ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext1.xml");
        System.out.println(context.getBean("bo1"));
    }
    //SpEL表达式对方法的支持
    @Test
    public void test4(){
        ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext1.xml");
        BasicMethod bMethod=(BasicMethod) context.getBean("bm1");
        System.out.println(bMethod.getSize());
        System.out.println(bMethod.getMsgUpper());
        System.out.println(bMethod.getIndex());

        //调用实例方法
        System.out.println(bMethod.getContent());
        //调用静态属性
        System.out.println(bMethod.getPi());
        //调用静态方法
        System.out.println(bMethod.getIsStudy());
        System.out.println(bMethod.getFood());



    }
    //SpEL表达式对正则的支持
    @Test
    public void test5(){
        ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext1.xml");
        Student student=(Student) context.getBean("stu");
        System.out.println(student.getIsMatch());
    }
```

```
    //SpEL的集合支持
    @Test
    public void test7(){
        ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext1.xml");
        ElModel model=(ElModel) context.getBean("el");
        System.out.println(model.getEmp().getName());
        System.out.println(model.getEmps().size());
    }
}
```

Spring EL表达式使用详解补充:

什么是Spring EL表达式

Spring EL 表达式是Spring表达式语言，支持在xml和注解中使用表达式，类似于JSP的EL，JSTL表达式语言。Spring开发中我们会经常涉及到调用各种资源的情况，包含普通文件、网址、正则表达式、系统变量、其他Bean的一些属性、配置文件、集合等等，我们就可以使用Spring的表达式语言实现资源的注入。

试想，我们平常通过注解或xml配置文件方式注入的Bean或Bean属性,其实都是静态注入,如果，Bean A中的一个成员变量m的值需要参考Bean B中的成员变量n的值，这种情况静态注入就显得无力。而Spring EL表达式就完全可以满足我们的种种动态的需求，甚至还能进行一些计算，功能非常强大。

使用Spring表达式语言，我们在项目中不需要手动管理Spring表达式的相关的接口和实例，只需要直接编写Spring表达式，Spring就会自动解析并转换表达式。

Spring EL的格式为 #{ SpEL expression } 。Spring表达式主要写在注解 @Value的参数中，它的作用是通过spring把值注入给某个属性。

下面以注解的方式列举一些Spring表达式的常用用法。xml配置文件中也是同样的用法。

注入字面值

表达式支持各种类型的字面值。字符串或字符类型的字面值需要使用单引号包括，其他类型字面值直接写就行。#{值}

注入操作系统（OS）的属性

Spring EL表达式还可以获取操作系统的属性，我们可以注入到需要的变量中，示例如下：

User

```
@Data
@Component
public class User {
    //注入操作系统的属性
    @Value("#{systemProperties['os.name']}")
```

```
    private String OSName;

    //注入操作系统的属性
    @Value("#{systemProperties['file.encoding']}")
    private String fileEncoding;
}
```

从容器中获取对象信息，并和手动获取的操作系统属性进行对比

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
System.out.println(applicationContext.getBean("user"));

        System.out.println("========手动获取信息=======");
        Properties properties = System.getProperties();
        System.out.println("os.name:" + properties.getProperty("os.name"));
        System.out.println("file.encoding:" +
properties.getProperty("file.encoding"));
```

运行结果如下：

```
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
User(OSName=Windows 10, fileEncoding=UTF-8)   ========手动获取信息=======
os.name:Windows 10   file.encoding:UTF-8 Process finished with exit code 0
```

构造器

在Spring EL表达式中，也使用new关键字来调用构造器，如果new的是java.lang包下的类的对象，可以省略包名。如果是自定义的类或者非java.lang包下的类，类名需要写全限定名。

User

```
@Data
@Component
public class User {

    //调用Computer的两个参数的构造方法，为User的Computer属性注入Computer对象
    @Value("#{new com.ls.entity.Computer('白色',66)}")
    private Computer computer;

    //注入新new的StringBuffer对象
    @Value("#{new StringBuffer('hello world!')}")
    private StringBuffer stringBuffer;
}
```

获取user对象并打印，结果如下：

```
信息: Loading properties file from class path resource [db2.properties]
User(computer=Computer(color=白色, price=66), stringBuffer=hello world!)     Process
finished with exit code 0
```

Elvis运算符

Spring EL表达式支持Elvis运算符，语法是变量?:默认值 意思是当某变量不为 null 或不表达空的意思（如空字符串）的时候使用该变量，当该变量为

null 或表达空的意思的时候使用指定的默认值。

Computer

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class Computer {

    //string1赋值白色字符串
    @Value("#{'白色'}")
    private String string1;

    //string2赋值空字符串
    @Value("#{''}")
    private String string2;

    //string3不赋值，默认为null
    private String string3;
}
```

User

```
@Data
@Component
public class User {

    //如果Spring容器中id为computer对象的string1属性为空字符串或null则赋值为默认值哈哈，否则赋值为computer对象的string1属性
    @Value("#{computer.string1?:'哈哈'}")
    private String username1;

    @Value("#{computer.string2?:'哈哈2'}")
    private String username2;

    @Value("#{computer.string3?:'哈哈3'}")
    private String username3;
}
```

#this和#root

#this和#root代表了表达式上下文的对象，我们可以提前定义一个上下文根对象，这样就可以使用#root来引用这个根对象。而且#root被定义始终引用上下文根对象。通过跟队形可以向表达式公开一个全面的自定义量。#this则根据当前求值环境的不同而变化。#this来表示当前的对象. 常用于集合的过滤，下面的例子中，#this即每次循环的值。

Computer

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class Computer {

    private String brand;

    @Value("#{{1,2,3,4,5}}")
    private List<Integer> computerIds;
}
```

User

```java
@Data
@Component
public class User {
    //#root用法演示
    static {
        //手动创建解析器来解析表达式

        //造一个电脑对象，用于将其存入上下文对象的根对象进行演示
        Computer computer = new Computer();
        computer.setBrand("华硕");

        //获取上下文对象
        StandardEvaluationContext context = new StandardEvaluationContext();
        //将computer存入上下文根对象，以在别的地方使用Spring EL表达式#root来获取在此存入的computer对象
        context.setRootObject(computer);

        //手动创建一个解析器
        ExpressionParser parser = new SpelExpressionParser();
        //Spring EL表达式，取出上下文根对象，由于根对象为我们手动存的电脑，所以可以获取其brand属性值
        String statement = "#root.brand";
        //解析表达式
        Expression expression = parser.parseExpression(statement);
        //获取解析后的结果
        String result = expression.getValue(context, String.class);
        //打印结果
        System.out.println("取出根对象computer的brand属性为:"+result);
    }

    //#this用法演示
    /*
    集合.?[expression]：是一种语法，本文后面即有介绍，目的是选择符合条件的元素
    #this即代表遍历集合时每次循环的值，此处意思是遍历容器中id为computer对象的computerIds属性（前提是此属性是一个集合），选出集合中大于2的所有元素生成一个新的集合，并将新的集合注入给user对象的userIds属性.
     */
    @Value("#{computer.computerIds.?[#this>2]}")
    private List<Integer> userIds;
}
```

获取computer和user对象并打印，结果如下：

```
八月 01, 2019 4:38:13 下午 org.springframework.beans.factory.config.PropertiesFactoryBean loadProperties
信息: Loading properties file from class path resource [db2.properties]
取出根对象computer的brand属性为:华硕
Computer(brand=null, computerIds=[1, 2, 3, 4, 5])
User(userIds=[3, 4, 5])

Process finished with exit code 0
```

操作符（运算符）

表达式中支持各种运算符，运算规则和Java规则类似。常用运算符如下：

```
//instanceof使用
@Value("#{1 instanceof T(Integer)}")
private boolean instanceofTest; //true
@Value("#{computer instanceof T(com.ls.entity.Computer)}")
private boolean instanceofTest2; //true

/*
between使用,判断一个数据是否在两个数据之间
格式：a between {m,n}
m的值必须在n前面，也就是m必须比n小
m和n不能交换顺序，否则虽然不会报错但是无法获得正确的比较结果
 */
@Value("#{3 between {2,5}}")
private boolean betweenTest;     //true
@Value("#{'ab' between {'aa','ac'}}")
private boolean betweenTest2;     //true

//正则表达式匹配，我们可以使用正则表达式和 matches关键字匹配数据,只有完全匹配的时候返回true,match前的字符串不能为null否则报错
@Value("#{'35' matches '\\d+'}")
private boolean regExp; //true
@Value("#{computer.str1 matches '\\w+'}")
private boolean regExp2; //true
```

注意：空值的处理，假设有非空值val，那么表达式 val > null 恒为真，这一点需要注意。

安全导航运算符

这是来自Groovy的一个功能，语法是?. ，当然有些语言也提供了这个功能。当我们对对象的某个属性求值时，如果该对象本身为空，就会抛出空指针异常，如果使用安全导航运算符，空对象的属性就会简单的返回空。

Computer

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class Computer {
    //注入字符串str1
    @Value("#{'str1'}")
    private String str1;
    //注入空字符串
    @Value("#{''}")
    private String str2;
    //不赋值默认为 null
    private String str3;
}
```

User

```java
@Data
@Component
public class User {

    //安全导航运算符 ?. 避免空指针异常，如果调用对象为null则直接返回null，避免了空指针异常
    @Value("#{computer.str1?.concat('abc')}")
    private String str1;

    @Value("#{computer.str2?.concat('abc')}")
    private String str2;

    @Value("#{computer.str3?.concat('abc')}")
    private String str3;
}
```

获取computer和user对象并打印，结果如下：

```
八月 01, 2019 5:48:49 下午 org.springframework.beans.factory.config.PropertiesFactoryBean loadProperties
信息: Loading properties file from class path resource [db2.properties]
Computer(str1=str1, str2=, str3=null)
User(str1=str1abc, str2=abc, str3=null)

Process finished with exit code 0
```

我们看到computer.str3为null并没有报空指针异常，而是直接给str3注入了null，另外两项不为null的都正常执行，后面拼接了字符串abc。

从数组、List、Map集合取值

我们可以使用Spring EL表达式从数组，list集合，set集合或map中取值。

- 数组和列表可以使用方括号引用对应索引的元素，list[index]。
- Map类型可以使用方括号引用键对应的值，map[key]。

Computer

```java
@Data
@Component
public class Computer {

    //给数组赋值
    @Value("#{{1,2,3,4}}")
    private Integer[] ids;

    //给List集合赋值
    @Value("#{{'list1','list2','list3'}}")
    private List<String> hobbiesList;

    //给List集合赋值
    @Value("#{{'set1','set3','set2'}}")
    private Set<String> hobbiesSet;

    @Value("#{{'key1':'value1','key2':'value2'}}")
    private Map<String,String> map;
}
```

User

```java
@Data
@Component
public class User {

    //取出数组0号索引的值
    @Value("#{computer.ids[0]}")
    private Integer id;

    //取出List集合中索引为0的值
    @Value("#{computer.hobbiesList[0]}")
    private String hobbyList;

    //取出Set集合中索引为0的值,set集合也可以通过索引取值，没想到吧
    @Value("#{computer.hobbiesSet[0]}")
    private String hobbySet;

    //取出map集合中key为key1的值
    @Value("#{computer.map['key1']}")
    private String MapStr;
}
```

获取computer和user对象并打印，结果如下：

```
Computer(ids=[1, 2, 3, 4], hobbiesList=[list1, list2, list3], hobbiesSet=[set1, set3, set2], map={key1=value1, key2=value2}) User(id=1, hobbyList=list1, hobbySet=set1, MapStr=value1)
```

集合投影

我们可以在Spring EL表达式中，将一个集合中所有元素的某属性抽取出来，组成一个新集合。语法是![投影表达式]。（其实在上面我们已经使用过了。。。-_-）

Computer

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class Computer {
    private int price;
    private String brand;
}
```

User

```
@Data
@Component
public class User {

    //new 三个Computer对象给集合赋值
    @Value("#{{new com.ls.entity.Computer(88,'联想'),new
com.ls.entity.Computer(77,'弘基'),new com.ls.entity.Computer(99,'华硕')}}")
    private List<Computer> computers;

    //将computers集合中所有computer的brand属性投影到brands集合中，组成新集合
    @Value("#{user.computers.![#this.brand]}")
    private List<String> brands;

    //将computers集合中所有computer的price属性值乘以10投影到 prices集合中，组成新集
合
    @Value("#{user.computers.![#this.price * 10]}")
    private List<String> prices;
}
```

输出：

```
Computer(price=0, brand=null)
User(computers=[Computer(price=88, brand=联想), Computer(price=77, brand=弘基),
Computer(price=99, brand=华硕)], brands=[联想, 弘基, 华硕], prices=[880, 770, 990])
```

Spring的#和$的区别

在spring中有#的使用也有$的使用，那么这两个分别是做什么的呢?

${key名称}:

1、用户获取外部文件中指定key的值;

2、可以出现在xml配置文件中，也可以出现在注解@Value中;

3、一般用户获取数据库配置文件的内容信息等。

#{表达式}：

1、SpEL表达式的格式，如上所述。

2、可以出现在xml配置文件中，也可以出现在注解@Value中

3、可以任意表达式，支持运算符等。

SpEL：Spring Expression Language，spring的一套表达式，主要应用在IOC容器进行对象属性的注入。格式为：#{表达式}

在使用的时候也允许#{‘${key}’}这样使用。

比如：

@Value(“#{‘${jdbc.url}’}”)

private String jdbcUrl;