Spring注解驱动开发

项目准备

直接新建一个 `maven` 项目, `pom.xml` 如下

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ddf.spring</groupId>
    <artifactId>spring-annotation</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <!-- https://mvnrepository.com/artifact/javax.inject/javax.inject -->
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.44</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.10</version>
        </dependency>

    </dependencies>

</project>
```

一、容器

1. AnnotationConfigApplicationContext

1.1 @Configuration配置类

@Configuration 该注解标注在某个类上，则可以被 `spring` 识别为一个配置类，配置类的作用等同于以前配置文件版的 `xml` .具体需要在配置类中配置什么属性或起到什么作用，这就是另外注解的支持了，@Configuration只是保证你可以在该类中配置的功能能够被解析,如同一个空的 `xxx.xml` 配置文件,这是一切功能型注解生效的前提!

1.2 @Bean注解注入 `bean`

`@Bean` 该注解可以标注在某个方法上，它实现的意义是为 `IOC` 中容器注入 `bean` ，此时 `bean` 的名称为方法名称， `bean` 的类型为方法返回类型.以下为向容器中注入一个 `type` 为 `User` 的 `bean` ，该 `bean` 的名称为 `user` ,默认都是单实例 `bean`

```
@Bean
public User user() {
    return new User();
}
```

等同于

```
<bean id="user" class="xxx.xxx.User"/>
```

1.3 创建AnnotationConfigApplicationContext

- 创建一个 `Bean` 对象 `User.java`

```java
package com.ddf.spring.annotation.entity;

/**
 * @author DDf on 2018/7/19
 */
public class User {

    private Integer id;
    private String userName;
    private String password;
    private String tel;

    public User() {
        System.out.println("User创建完成.............");
    }

    public User(Integer id, String userName, String password, String tel) {
        this.id = id;
        this.userName = userName;
        this.password = password;
        this.tel = tel;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }
}
```

- 新建一个配置类 `AnnotationConfiguration`

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @author DDf on 2018/7/19
 */
@Configuration
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * @return
     */
    @Bean
    public User user() {
        return new User();
    }
}
```

- 新建主启动类 `Application`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------");
        // 获取当前IOC中所有bean的名称
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }
    }
}
```

打印如下,可以看到除了最后一行我们注入的 `User` 以外，容器启动后会默认注入一些 `bean` ，但这些 `bean` 默认都是懒加载的，通过打印可以看出来，这是在容器已经初始化完成之后，获取预定义 `bean` 的名称然后调用 `getBean()` 方法之后获取才创建的，懒加载就是针对单实例的而且是在第一次过去才会被创建，而默认情况下都是容器启动后就会完成创建

```
----------------------IOC容器初始化-------------------------
User创建完成.............
----------------------IOC容器初始化完成-------------------------
bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$1906ac13
bean name:user, type: class com.ddf.spring.annotation.entity.User
```

2. Bean的导入

2.1 @ComponentScan包扫描

该注解需要配合 `@Configuration` 标识的配置类上使用才会生效，它的作用是指定 `spring` 的扫描包，则在扫描路径下的被特定注解标识的类就会被自动纳入到 `IOC` 中，如我们熟知的 `@Repository` 、 `@Service` 、 `@Controller` 等，而不需要每个都使用 `@Bean` 注解一个个注入。
*属性详解

value	指定要扫描的包
excludeFilters	指定扫描的时候按照什么规则排除那些组件
includeFilters	指定扫描的时候只需要包含哪些组件，如果需要生效需要加入useDefaultFilters = false属性
FilterType.ANNOTATION	按照注解
FilterType.ASSIGNABLE_TYPE	按照给定的类
FilterType.ASPECTJ	使用ASPECTJ表达式
FilterType.REGEX	使用正则指定
FilterType.CUSTOM	使用自定义规则

- 添加一个 `UserController.java` 来测试排除注解类型为 `@Controller` 的类到容器中

```java
package com.ddf.spring.annotation.controller;

import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * 该类的作用是为了测试排除所有@Controller注解的类不需要注册到IOC容器中,
 * 详见{@link com.ddf.spring.annotation.configuration.AnnotationConfiguration}
 */
@Controller
public class UserController {
    public UserController() {
        System.out.println("UserController创建完成.................");
    }
}
```

- 添加一个 `UserSerivce` ，使用 `@Service` 注解标注来测试扫描注入

```java
package com.ddf.spring.annotation.service;

import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/19
 * 默认@Scope为singleton，IOC容器启动会创建该bean的实例，并且以后再次使用不会重新创建新的实例
 */
@Service
public class UserService {
    public UserService() {
        System.out.println("UserService创建完成...................");
    }
}
```

- 添加一个实现了 `TypeFilter` 接口的类，来自定义过滤规则 `ExcludeTypeFilter.java`

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.core.io.Resource;
import org.springframework.core.type.AnnotationMetadata;
import org.springframework.core.type.ClassMetadata;
import org.springframework.core.type.classreading.MetadataReader;
import org.springframework.core.type.classreading.MetadataReaderFactory;
import org.springframework.core.type.filter.TypeFilter;

import java.io.IOException;

/**
 * @author DDf on 2018/7/19
 */
public class ExcludeTypeFilter implements TypeFilter {
    @Override
    public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {
        // 获取当前类注解的信息
        AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();
        // 获取当前正在扫描的类的类信息
        ClassMetadata classMetadata = metadataReader.getClassMetadata();
        // 获取当前类资源（类的路径）
        Resource resource = metadataReader.getResource();

        String className = classMetadata.getClassName();
        System.out.println("自定义扫描类规则当前扫描类为： "+ className);
        // 这一块是return true 还是return false是有讲究的，比如把这个规则当做excludeFilters，那么返回true，则匹配的会过滤掉,如果把这个规则
        // 应用到includeFilters,如果返回true，则是会加入到容器中
        if (className.contains("ExcludeFilter")) {
            return true;
        }
        return false;
    }
}
```

- 添加一个符合 `ExcludeTypeFilter.java` 规则的 `@Service` 类，这个类本身应该会被纳入到容器中，在下面会演示如何根据自定义规则不纳入到容器中

```java
package com.ddf.spring.annotation.service;

import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/19
 * 该类的作用请参考{@link com.ddf.spring.annotation.configuration.ExcludeTypeFilter}是为了测试自定义规则决定是否导入某些bean
 */
@Service
public class ExcludeFilterService {
    public ExcludeFilterService() {
        System.out.println("ExcludeFilterService创建完成.............");
    }
}
```

修改 `AnnotationConfiguration.java`

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.service.IncludeFilterService;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type=FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * @return
     */
    @Bean
    public User user() {
        return new User();
    }
}
```

运行主启动类 `Application.java` ，可以看到多注入了名称为 `userService` 的 `bean` ，标注了 `@Controller` 的 `UserController` 被排除纳入，同样满足 `ExcludeTypeFilter.java` 定义规则的标注了 `@Service` 的 `ExcludeFilterService.java` 也被排除在外

```
----------------------IOC容器初始化-------------------------
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
七月 19, 2018 11:53:53 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
User创建完成..............
----------------------IOC容器初始化完成-------------------------
bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$19c06e9b
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:user, type: class com.ddf.spring.annotation.entity.User
```

2.2 @Scope Bean的作用域

`Spring` 初始化容器组件默认的 `scope` 都是 `singleton` ,即单实例。在容器初始化的时候就会把所有单实例的 `bean` 初始化创建出来，以后每次在使用的时候不会重新创建一个对象，可以通过 `@Scope` 来修改默认的作用域，作用域有以下属性

singleton	默认属性，单实例
prototype	启动创建，每次使用都是创建一个新的实例
request	基于`HttpServletRequest`，每次请求创建一个实例
session	基于`HttpSession`,每个`session`作用域下使用同一个实例

- 新建一个 `Service` 使用 `@Scope("prototype")` 注解修饰， `PrototypeScopeService.java`

```java
package com.ddf.spring.annotation.service;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/21
 * 测试@Scope的作用域为prototype，每次使用该bean都会重新生成一个实例
 */
@Service
@Scope("prototype")
public class PrototypeScopeService {
    public PrototypeScopeService() {
        System.out.println("PrototypeScopeService创建完成。。。。。。。");
    }
}
```

- 使用上面建立的 `UserService.java` 来作为对比测试

```java
package com.ddf.spring.annotation.service;

import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/19
 * 默认@Scope为singleton，IOC容器启动会创建该bean的实例，并且以后再次使用不会重新创建新的实例
 */
@Service
public class UserService {
    public UserService() {
        System.out.println("UserService创建完成...................");
    }
}
```

- 修改主启动类 `Application.java` ，添加测试 `@Scope` 的方法 `testPrototypeScopeService`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.PrototypeScopeService;
import com.ddf.spring.annotation.service.UserService;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------");
        // 获取当前IOC中所有bean的名称
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);

    }


    /**
     * 测试@Scope bean的作用域
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("-----------------------测试@Scope-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
    }
}
```

- 测试结果如控制台打印所示

```
----------------------IOC容器初始化-------------------------
七月 21, 2018 6:52:36 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Sat Jul 21
18:52:36 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
七月 21, 2018 6:52:37 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
User创建完成..............
----------------------IOC容器初始化完成-------------------------
bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$5ee69aa7
PrototypeScopeService创建完成。。。。。。。
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:user, type: class com.ddf.spring.annotation.entity.User

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束-------------------------
```

2.3 @Lazy 单实例Bean的懒加载

默认情况下所有的单实例bean都会在容器启动的时候创建出来，使用的时候不会直接获取不会重复创建，但是可以通过 `@Lazy` 注解来让容器启动的时候不创建某些类，而是等待第一次获取使用该类的时候才创建该Bean对象.

- 创建 `LazyBeanService.java` ，使用 `@Lazy` 和 `@Service` 注解修饰来懒加载注入

```java
package com.ddf.spring.annotation.service;

import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/21
 * 测试单实例bean的@Lazy IOC容器启动的时候不创建该bean，只有到第一次获取的时候才创建
 * 但当第一次使用这个bean的时候再创建，以后使用不再创建
 */
@Service
@Lazy
public class LazyBeanService {
    public LazyBeanService() {
        System.out.println("LazyBeanService创建完成.............");
    }
}
```

- 修改主启动类`Application.java`，增加测试懒加载的方法`testLazyBeanService`,需要注意的是主启动类在以前的代码中为了测试方便，在启动后获取了所有预定义的`bean`的名称然后获取打印了出来，为了更清楚的测试懒加载这一块的`getBean()`需要注释掉，因为这里会牵扯到获取创建的问题

```
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.LazyBeanService;
import com.ddf.spring.annotation.service.PrototypeScopeService;
import com.ddf.spring.annotation.service.UserService;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        /*for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }*/

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);

    }


    /**
     * 测试@Scope bean的作用域
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("-----------------------测试@Scope-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("---------------测试单实例bean的@Lazy懒加载----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1? : " + (lazyBeanService == lazyBeanService1));
    }
}
```

- 启动后查看控制台日志,`LazyBeanService`直到第一次获取的时候才被创建，并且是单实例

```
----------------------IOC容器初始化-------------------------
七月 21, 2018 9:29:55 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Sat Jul 21
21:29:55 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
七月 21, 2018 9:29:55 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
User创建完成..............
----------------------IOC容器初始化完成-------------------------

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束-------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------

Process finished with exit code 0
```

2.4 @Conditional 动态判断是否导入Bean

`@Conditional` 注解可以和导入 `Bean` 的注解组合使用，导入 `Bean` 的注解的作用是为容器中添加一个 `Bean` ，而且只要使用了注解就无条件的导入，而 `@Conditional` 注解可以指定一个类，该类必须实现 `org.springframework.context.annotation.Condition` 接口，重写 `matches` 方法返回true，则可以正常导入组件，但是如果返回false，则不能导入组件。而且该 `Condition` 类中还可以额外注册一些组件，这对于某些组件是否导入是根据某些条件导入的，而且导入之后还需要附加某些组件来说是很有用处的。

> 现在来模拟一个场景，有两个需要动态导入的 `Bean` ，一个是 `DevelopmentBean` ，该 `Bean` 有一个附加 `Bean` 为 `DevelopmentBeanLog` ，同时还有一个 `Bean` 名为 `ProductionBean` ，该 `Bean` 同样也有一个附加 `Bean` 名为 `ProductionBeanLog` 。现在的需求是根据不同的环境，如果是 `dev` 环境则注册的 `DevelopmentBean` 会生效，同时导入 `DevelopmentBeanLog` 。如果当前环境是 `prd` ，则注册的 `ProductionBean` 会生效，同时会导入 `ProductionBeanLog` 。环境的切换应该由外部配置文件来生效，但是为了方便测试，也为了不改代码就能够测试到两种不同的环境，所以使用当前时间的分钟，如果使偶数，则 `dev` 环境生效，如果使奇数，则 `prd` 环境生效;
>
> - 创建 `DevelopmentBean.java`

```
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/21
 */
public class DevelopmentBean {
    public DevelopmentBean() {
        System.out.println("DevelopmentBean创建完成.............");
    }
}
```

- 创建 `DevelopmentBeanLog.java`

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/21
 */
public class DevelopmentBeanLog {
    public DevelopmentBeanLog() {
        System.out.println("DevelopmentBeanLog创建完成.............");
    }
}
```

- 创建 `ProductionBean.java`

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/21
 */
public class ProductionBean {
    public ProductionBean() {
        System.out.println("ProductionBean创建完成.............");
    }
}
```

- 创建 `ProductionBeanLog.java`

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/21
 */
public class ProductionBeanLog {
    public ProductionBeanLog() {
        System.out.println("ProductionBeanLog创建完成.................");
    }
}
```

- 创建 `dev` 环境生效的判断条件和导入组件类 `DevelopmentProfileCondition.java`

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.type.AnnotatedTypeMetadata;

import java.time.LocalDateTime;

/**
 * @author DDf on 2018/7/21
 * 创建根据条件来判断是否导入某些组件，该类需要配合@Condition注解，@Condition注解需要用在要导入容器的地方，与导入组件注解组合使用，如果当前类
 * 返回true，则可以导入组件，反之，则不能。
 */
public class DevelopmentProfileCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        // 获取到bean定义的注册类(可以获取bean，注册bean，删除预定义的bean名称)
        BeanDefinitionRegistry registry = context.getRegistry();
        // 获取IOC容器使用的beanfactory（可以获取bean的定义信息，可以获取到bean的定义注册类）
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        // 可以获取到环境变量
        ConfigurableEnvironment environment = (ConfigurableEnvironment) context.getEnvironment();
        // 根据当前时间的分钟动态切换环境变量的值
        LocalDateTime localDateTime = LocalDateTime.now();
        int minute = localDateTime.getMinute();
        String profile;
        if (minute % 2 == 0) {
            profile = "dev";
        } else {
            profile = "prd";
        }
        System.out.println("DevelopmentProfileCondition profile: " + profile);
        // 如果是dev环境，并且当前IOC容器中未定义ProductionBean则返回true，同时注册一个DevelopmentBeanLog
        if ("dev".equals(profile)) {
            if (!registry.containsBeanDefinition("ProductionBean")) {
                RootBeanDefinition devServiceLogBean = new RootBeanDefinition(
                        "com.ddf.spring.annotation.bean.DevelopmentBeanLog");
                registry.registerBeanDefinition("DevelopmentBeanLog", devServiceLogBean);
                return true;
            }
        }
        return false;
    }
}
```

- 创建 `prd` 环境生效的判断条件和导入组件类 `ProductionProfileCondition.java`

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.type.AnnotatedTypeMetadata;

import java.time.LocalDateTime;

/**
 * @author DDf on 2018/7/21
 * 创建根据条件来判断是否导入某些组件，该类需要配合@Condition注解，@Condition注解需要用在要导入容器的地方，与导入组件注解组合使用，如果当前类
 * 返回true，则可以导入组件，反之，则不能。
 */
public class ProductionProfileCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        // 获取到bean定义的注册类(可以获取bean，注册bean，删除预定义的bean名称)
        BeanDefinitionRegistry registry = context.getRegistry();
        // 获取IOC容器使用的beanfactory（可以获取bean的定义信息，可以获取到bean的定义注册类）
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        // 可以获取到环境变量
        ConfigurableEnvironment environment = (ConfigurableEnvironment) context.getEnvironment();
        // 根据当前时间的分钟动态切换环境变量的值
        LocalDateTime localDateTime = LocalDateTime.now();
        int minute = localDateTime.getMinute();
        String profile;
        if (minute % 2 == 0) {
            profile = "dev";
        } else {
            profile = "prd";
        }
        System.out.println("ProductionProfileCondition profile: " + profile);
        // 如果是prd环境，并且当前IOC容器中未定义DevelopmentBean则返回true，同时注册一个ProductionBeanLog
        if ("prd".equals(profile)) {
            // 如果是prd环境，并且当前IOC容器中未定义DevelopmentBean则返回true，同时注册一个DevelopmentBeanLog
            if (!registry.containsBeanDefinition("DevelopmentBean")) {
                RootBeanDefinition prdServiceLogBean = new RootBeanDefinition(
                        "com.ddf.spring.annotation.bean.ProductionBeanLog");
                registry.registerBeanDefinition("prdServiceLog", prdServiceLogBean);
                return true;
            }
        }
        return false;
    }
}
```

- 修改主配置类 `AnnotationConfiguration` ，使用 `@Bean` 注解和 `@Condition` 注解分别标注对应的类

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.DevelopmentBean;
import com.ddf.spring.annotation.bean.ProductionBean;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type=FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * @return
     */
    @Bean
    public User user() {
        return new User();
    }


    /**
     * 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean DevelopmentBean() {
        System.out.println("-------------------------测试@Conditional------------------");
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean ProductionBean() {
        System.out.println("-------------------------测试@Conditional------------------");
        return new ProductionBean();
    }
}
```

- 运行主启动类 `Application.java`，控制台效果如下，通过控制台可以看到,虽然在配置类中定义了 `ProductionBean()` 和 `DevelopmentBean()` 两个方法，但是根据当前生效的环境，最终只会有一个被成功注入，另外一个就被忽略了，而且也可以看到在 `Condition` 类中注册的类比方法体中注册的要早。

```
----------------------IOC容器初始化-------------------------
七月 21, 2018 10:54:48 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Sat Jul 21
22:54:48 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
DevelopmentProfileCondition profile: dev
ProductionProfileCondition profile: dev
七月 21, 2018 10:54:48 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
User创建完成..............
DevelopmentBeanLog创建完成..............
-------------------------测试@Conditional------------------
DevelopmentBean创建完成.............
----------------------IOC容器初始化完成-------------------------
bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$75d3a279
LazyBeanService创建完成..............
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:user, type: class com.ddf.spring.annotation.entity.User
bean name:DevelopmentBeanLog, type: class com.ddf.spring.annotation.bean.DevelopmentBeanLog
bean name:DevelopmentBean, type: class com.ddf.spring.annotation.bean.DevelopmentBean

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束-------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------

Process finished with exit code 0
```

2.5 @Import导入组件

> 该注解需要标注在带有 `@Configuration` 的类上，@Import的 `value` 属性可以直接指定一个普通的 `java` 类，即可直接导入组件，也可以实现 `ImportSelector` 接口来导入一组组件，也可以实现 `ImportBeanDefinitionRegistrar` 接口来获得当前注解信息，通过 `BeanDefinitionRegistry` 来注册组件，这种方式可以指定导入的组件ID

2.5.1 @Import

- 新建一个普通的类 `ImportBean.java` ，用来实验直接通过 `@Import` 导入

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/30
 */
public class ImportBean {
    public ImportBean() {
        System.out.println("ImportBean创建完成（测试@Import导入组件）.........");
    }
}
```

- `Configuration` 类参考2.5.4

2.5.2 `@Import + ImportSelector`

- 新建一个 `CustomImportSelector.java` 实现 `ImportSelector` 接口

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.type.AnnotationMetadata;

import java.util.Set;

/**
 * @author DDf on 2018/7/30
 * 测试和@Import一起使用通过ImportSelector来导入组件
 */
public class CustomImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        Set<String> annotationTypes = importingClassMetadata.getAnnotationTypes();
        // 做个简单的判断，如果标注了@Import注解的类上还有指定的另外一个注解，则导入一些组件,不可以自定义bean的名称
        if (annotationTypes.contains("org.springframework.context.annotation.Import")) {
            return new String[]{"com.ddf.spring.annotation.bean.ImportSelectorBean"};
        }
        return new String[0];
    }
}
```

- `Configuration` 类参考2.5.4

2.5.3 `@Import + ImportBeanDefinitionRegistrar`

- 新建一个 `CustomImportBeanDefinitionRegistrar.java` 实现 `ImportBeanDefinitionRegistrar` 接口

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
import org.springframework.core.type.AnnotationMetadata;

import java.util.Set;

/**
 * @author DDf on 2018/7/31
 * 测试使用@Import注解结合ImportBeanDefinitionRegistrar
 */
public class CustomImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionRegistry) {
        Set<String> annotationTypes = annotationMetadata.getAnnotationTypes();
        // 做个简单的判断，如果标注了@Import注解的类上还有指定的另外一个注解，则导入一些组件
        if (annotationTypes.contains("org.springframework.context.annotation.Import")) {
            // 通过BeanDefinitionRegistry导入一个组件
            RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(
                    "com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean");
            // 通过BeanDefinitionRegistry导入的组件可以自定义bean的名称
            beanDefinitionRegistry.registerBeanDefinition("importBeanDefinitionRegistrarBean", rootBeanDefinition);
        }
    }
}
```

- `Configuration` 类参考2.5.4

2.5.4 修改 `AnnotationConfiguration.java` 配置类，在前面的基础上加上 `@Import` 的注解

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.DevelopmentBean;
import com.ddf.spring.annotation.bean.ImportBean;
import com.ddf.spring.annotation.bean.ProductionBean;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 * @Import 导入组件，可以直接导入普通类，或者通过ImportSelector接口或者ImportBeanDefinitionRegistrar接口来自定义导入
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     *
     * @return
     */
    @Bean
    public User user() {
        return new User();
    }


    /**
     * 测试@Conditional 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean DevelopmentBean() {
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean ProductionBean() {
        return new ProductionBean();
    }
}
```

2.5.5 演示结果

- 启动主类 `Application.java` ，打印如下,可以看到 `ImportBean.java ImportSelectorBean ImportBeanDefinitionRegistrarBean` 在IOC容器初始化的时候创建成功。

```
----------------------IOC容器初始化------------------------
八月 01, 2018 5:43:40 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Wed Aug 01
17:43:40 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportBeanDefinitionRegistrar
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportSelector
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryPrototypeBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactorySingletonBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportSelectorBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
DevelopmentProfileCondition profile: prd
ProductionProfileCondition profile: prd
八月 01, 2018 5:43:40 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
ImportBean创建完成（测试@Import导入组件）..........
ImportSelectorBean创建完成，测试@Import通过ImportSelector接口导入组件
User创建完成..............
ProductionBeanLog创建完成.......测试@Condition............
ProductionBean创建完成......测试@Condition.......
ImportBeanDefinitionRegistrarBean创建完成，测试@Import接口通过ImportBeanDefinitionRegistrar接口注入组件。。。。。
----------------------IOC容器初始化完成------------------------

bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$d7abd5ba
LazyBeanService创建完成..............
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:com.ddf.spring.annotation.bean.ImportBean, type: class com.ddf.spring.annotation.bean.ImportBean
bean name:com.ddf.spring.annotation.bean.ImportSelectorBean, type: class com.ddf.spring.annotation.bean.ImportSelectorBean
bean name:user, type: class com.ddf.spring.annotation.entity.User
bean name:prdServiceLog, type: class com.ddf.spring.annotation.bean.ProductionBeanLog
bean name:ProductionBean, type: class com.ddf.spring.annotation.bean.ProductionBean
bean name:importBeanDefinitionRegistrarBean, type: class com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean

----------------------测试@Scope开始------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。
```

```
PrototypeScopeService创建完成。。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束------------------------



---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------

Process finished with exit code 0
```

2.6 通过 `FactoryBean` 接口注入组件

> 通过FactoryBean工厂注册组件，该类本身需要注册到IOC容器中，如在类上标注 `@Component` 等或者使用 `@Bean` 注解,实际在IOC中注册的组件为FactoryBean中接口的方法来决定而非 `FactoryBean` 类本身，通过这种方式创建的组件可以 `isSingleton` 方法来决定组件是否为单实例，如果为单实例则同时该 `Bean` 是以懒加载的方式注册的

- 创建 `FactorySingletonBean.java` ，用来演示单实例bean

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/7/31
 * 该类用于测试使用FactoryBean来注册单实例组件
 */
public class FactorySingletonBean {
    public FactorySingletonBean() {
        System.out.println("FactorySingletonBean创建完成。。。。，测试通过FactoryBean来注册单实例组件。。。。");
    }
}
```

- 创建 `FactorySingletonBean` 的 `FactoryBean` 接口, `FactorySingletonBeanConfiguration.java`

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.FactorySingletonBean;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/1
 * 通过FactoryBean工厂注册组件，该类本身需要注册到IOC容器中
 * 实际在IOC中注册的组件为FactoryBean中接口的方法来决定
 */
// @Component
public class FactorySingletonBeanConfiguration implements FactoryBean<FactorySingletonBean> {
    /**
     * 要注册的组件
     * @return
     */
    @Override
    public FactorySingletonBean getObject() {
        return new FactorySingletonBean();
    }

    /**
     * 要注册的组件类型
     * @return
     */
    @Override
    public Class<?> getObjectType() {
        return FactorySingletonBean.class;
    }

    /**
     * 要注册的组件是否是单实例
     * @return
     */
    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

- 创建 `FactoryPrototypeBean.java`，用来演示非单实例 `bean`

```java
package com.ddf.spring.annotation.service;

/**
 * @author DDf on 2018/8/1
 * 该类用于测试使用FactoryBean来注册每个请求重新创建组件
 */
public class FactoryPrototypeBean {
    public FactoryPrototypeBean() {
        System.out.println("FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。");
    }
}
```

- 创建 `FactoryPrototypeBean` 的 `FactoryBean` 接口 `FactoryPrototypeBeanConfiguration.java`

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.FactoryPrototypeBean;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/1
 * 通过FactoryBean工厂注册组件，该类本身需要注册到IOC容器中
 * 实际在IOC中注册的组件为FactoryBean中接口的方法来决定
 */
// @Component
public class FactoryPrototypeBeanConfiguration implements FactoryBean<FactoryPrototypeBean> {

    /**
     * 要注册的组件
     * @return
     */
    @Override
    public FactoryPrototypeBean getObject() {
        return new FactoryPrototypeBean();
    }

    /**
     * 要注册的组件类型
     * @return
     */
    @Override
    public Class<?> getObjectType() {
        return FactoryPrototypeBean.class;
    }

    /**
     * 要注册的组件是否是单实例
     * @return
     */
    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

- 修改 `AnnotationConfiguration.java`，使用配置类的方式将 `FactoryPrototypeBeanConfiguration` 和 `FactorySingletonBeanConfiguration` 两个 `FactoryBean` 接口注册到容器中

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.DevelopmentBean;
import com.ddf.spring.annotation.bean.ImportBean;
import com.ddf.spring.annotation.bean.ProductionBean;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 * @Import 导入组件，可以直接导入普通类，或者通过ImportSelector接口或者ImportBeanDefinitionRegistrar接口来自定义导入
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     *
     * @return
     */
    @Bean
    public User user() {
        return new User();
    }


    /**
     * 测试@Conditional 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean DevelopmentBean() {
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean ProductionBean() {
        return new ProductionBean();
    }


    /**
     * 使用FactoryBean工厂来注册组件
     * @return
     */
    @Bean
    public FactoryPrototypeBeanConfiguration factoryPrototypeBeanConfiguration() {
        return new FactoryPrototypeBeanConfiguration();
    }

    /**
     * 使用FactoryBean工厂来注册组件
```

```
     * @return
     */
    @Bean
    public FactorySingletonBeanConfiguration factorySingletonBeanConfiguration() {
        return new FactorySingletonBeanConfiguration();
    }
}
```

- 修改主启动类`Application.java`，用来验证`FactoryBean`创建`bean`的作用域

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.*;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);

    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1? : " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }


    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
```

```
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }
}
```

- 启动测试,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的,因为在 `IOC` 容器创建的时候并没有创建，但是在获得所有定义的bean打印的时候才创建

```
----------------------IOC容器初始化------------------------
八月 01, 2018 6:06:46 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Wed Aug 01
18:06:46 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportBeanDefinitionRegistrar
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportSelector
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryPrototypeBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactorySingletonBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportSelectorBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
DevelopmentProfileCondition profile: dev
ProductionProfileCondition profile: dev
八月 01, 2018 6:06:46 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition
信息: Overriding bean definition for bean 'factoryPrototypeBeanConfiguration' with a different definition: replacing [Generic
bean: class [com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration]; scope=singleton; abstract=false;
lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null;
factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in file [D:\dev-tools\idea_root\spring-
annotation\target\classes\com\ddf\spring\annotation\configuration\FactoryPrototypeBeanConfiguration.class]] with [Root bean:
class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false;
factoryBeanName=annotationConfiguration; factoryMethodName=factoryPrototypeBeanConfiguration; initMethodName=null;
destroyMethodName=(inferred); defined in com.ddf.spring.annotation.configuration.AnnotationConfiguration]
八月 01, 2018 6:06:46 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition
信息: Overriding bean definition for bean 'factorySingletonBeanConfiguration' with a different definition: replacing [Generic
bean: class [com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration]; scope=singleton; abstract=false;
lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null;
factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in file [D:\dev-tools\idea_root\spring-
annotation\target\classes\com\ddf\spring\annotation\configuration\FactorySingletonBeanConfiguration.class]] with [Root bean:
class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false;
factoryBeanName=annotationConfiguration; factoryMethodName=factorySingletonBeanConfiguration; initMethodName=null;
destroyMethodName=(inferred); defined in com.ddf.spring.annotation.configuration.AnnotationConfiguration]
八月 01, 2018 6:06:47 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
ImportBean创建完成（测试@Import导入组件）..........
ImportSelectorBean创建完成，测试@Import通过ImportSelector接口导入组件
User创建完成...............
DevelopmentBeanLog创建完成.......测试@Condition........
DevelopmentBean创建完成.....测试@Condition........
ImportBeanDefinitionRegistrarBean创建完成，测试@Import接口通过ImportBeanDefinitionRegistrar接口注入组件。。。。。
----------------------IOC容器初始化完成------------------------

bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
```

```
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$7e17d64c
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册单实例组件。。。。
bean name:factoryPrototypeBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactoryPrototypeBean
FactorySingletonBean创建完成。。。。，测试通过FactoryBean来注册单实例组件。。。。
bean name:factorySingletonBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactorySingletonBean
LazyBeanService创建完成.............
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。。
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:com.ddf.spring.annotation.bean.ImportBean, type: class com.ddf.spring.annotation.bean.ImportBean
bean name:com.ddf.spring.annotation.bean.ImportSelectorBean, type: class com.ddf.spring.annotation.bean.ImportSelectorBean
bean name:user, type: class com.ddf.spring.annotation.entity.User
bean name:DevelopmentBeanLog, type: class com.ddf.spring.annotation.bean.DevelopmentBeanLog
bean name:DevelopmentBean, type: class com.ddf.spring.annotation.bean.DevelopmentBean
bean name:importBeanDefinitionRegistrarBean, type: class com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。。
PrototypeScopeService创建完成。。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束-------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------


----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册单实例组件。。。。
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册单实例组件。。。。
单实例factorySingletonBean==factorySingletonBean1?true
Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?false
----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------


Process finished with exit code 0
```

3. `Bean` 的生命周期

> bean的生命周期指的是bean创建---初始化----销毁的过程
> 我们可以自定义初始化和销毁方法；容器在bean进行到当前生命周期的时候来调用我们自定义的初始化和销毁方法
> 构造（对象创建）
> 单实例：在容器启动的时候创建对象
> 多实例：容器不会管理这个bean；容器不会调用销毁方法；

3.1 `@Bean` 指定初始化方法和销毁方法

- 修改之前创建的 `User.java` ，增加初始化和销毁方法，方法体与方法名称自定义即可

```java
package com.ddf.spring.annotation.entity;

/**
 * @author DDf on 2018/7/19
 */
public class User {

    private Integer id;
    private String userName;
    private String password;
    private String tel;

    public void init() {
        System.out.println("User创建后调用初始化方法..........");
    }

    public void destory() {
        System.out.println("User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法......");
    }

    public User() {
        System.out.println("User创建完成...通过@Bean的initMethod调用初始化方法............");
    }

    public User(Integer id, String userName, String password, String tel) {
        this.id = id;
        this.userName = userName;
        this.password = password;
        this.tel = tel;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }
}
```

- 修改主配置类 `AnnotationConfiguration.java` ,使用 `@Bean` 注册 `User` 的时候指定初始化和销毁方法

> initMethod 指定Bean创建后调用的初始化方法
> destroyMethod 指定Bean在销毁后会调用的方法

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.service.*;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 * @Import 导入组件，可以直接导入普通类，或者通过ImportSelector接口或者ImportBeanDefinitionRegistrar接口来自定义导入
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * initMethod 指定Bean创建后调用的初始化方法
     * destroyMethod 指定Bean在销毁后会调用的方法
     * @return
     */
    @Bean(initMethod = "init", destroyMethod = "destory")
    public User user() {
        return new User();
    }


    /**
     * 测试@Conditional 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean DevelopmentBean() {
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean ProductionBean() {
        return new ProductionBean();
    }


    /**
     * 使用FactoryBean工厂来注册组件
     * @return
     */
    @Bean
    public FactoryPrototypeBeanConfiguration factoryPrototypeBeanConfiguration() {
        return new FactoryPrototypeBeanConfiguration();
    }

    /**
     * 使用FactoryBean工厂来注册组件
     * @return
```

```
     */
    @Bean
    public FactorySingletonBeanConfiguration factorySingletonBeanConfiguration() {
        return new FactorySingletonBeanConfiguration();
    }
}
```

3.2 `InitializingBean` 和 `DisposableBean` 接口

> 将要创建指定初始化和销毁方法的类实现初始化接口 `InitializingBean`，如果需要指定销毁方法实现 `DisposableBean` 接口

- 创建一个实现了 `InitializingBean` 和 `DisposableBean` 接口的 `Bean`

```
package com.ddf.spring.annotation.service;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

/**
 * @author DDf on 2018/8/1
 * 实现InitializingBean接口的afterPropertiesSet方法在Bean被创建后会自动调用该初始化方法
 * 实现DisposableBean接口的destroy方法会在容器会Bean被销毁时时调用该方法
 */
public class InitAndDisposableBean implements InitializingBean, DisposableBean {
    public InitAndDisposableBean() {
        System.out.println("InitAndDisposableBean创建完成。。。。。。。。。。。。");
    }
    @Override
    public void destroy() throws Exception {
        System.out.println("InitAndDisposableBean容器销毁，实现DisposableBean接口调用销毁方法.....");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("InitAndDisposableBean创建后实现InitializingBean调用初始化方法。。。。。。。。。。。。");
    }
}
```

3.3 `@PostConstruct` 和 `@PreDestroy` 注解

> 在需要指定初始化和销毁方法的Bean里创建对应的初始化和销毁方法，使用对应注解标注即可。
> 使用@PostConstruct指定Bean的初始化方法
> 使用@PreDestroy指定Bean销毁后调用的方法

- 创建 `PostConstructAndPreDestoryBean.java`

```java
package com.ddf.spring.annotation.service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

/**
 * @author DDf on 2018/8/1
 * 使用JSR250注解实现Bean的初始化方法和销毁方法调用
 */
public class PostConstructAndPreDestoryBean {
    public PostConstructAndPreDestoryBean() {
        System.out.println("PostConstructAndPreDestoryBean创建完成.......");
    }

    /**
     * 使用@PostConstruct指定Bean的初始化方法
     */
    @PostConstruct
    public void init() {
        System.out.println("PostConstructAndPreDestoryBean创建完成，使用@PostConstruct注解来调用初始化方法。。。。");
    }


    /**
     * 使用@PreDestroy指定Bean销毁后调用的方法
     */
    @PreDestroy
    public void destroy() {
        System.out.println("PostConstructAndPreDestoryBean容器销毁，使用@PreDestroy注解来指定调用销毁方法。。。。");
    }
}
```

3.4 修改主启动类 `Application.java`

以上三种形式，现在统一修改主启动类，然后集中验证,修改主启动类，在主程序最后关闭 `IOC` 容器

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.*;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1? : " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }
```

```
    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }
}
```

3.5 验证结果

运行主启动类，打印如下，可以看到容器在IOC容器创建后同时调用了初始化方法，并且在IOC容器关闭的时候调用了对应 `Bean` 的销毁方法

```
----------------------IOC容器初始化------------------------
八月 01, 2018 11:08:03 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Wed Aug 01 23:08:03 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportBeanDefinitionRegistrar
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportSelector
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryPrototypeBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactorySingletonBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportSelectorBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.InitAndDisposableBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
DevelopmentProfileCondition profile: dev
ProductionProfileCondition profile: dev
八月 01, 2018 11:08:04 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
UserService创建完成...................
ImportBean创建完成（测试@Import导入组件）..........
ImportSelectorBean创建完成，测试@Import通过ImportSelector接口导入组件
User创建完成...通过@Bean的initMethod调用初始化方法............
User创建后调用初始化方法..........
DevelopmentBeanLog创建完成.......测试@Condition........
DevelopmentBean创建完成.....测试@Condition........
InitAndDisposableBean创建完成。。。。。。。。。。。。
InitAndDisposableBean创建后实现InitializingBean调用初始化方法。。。。。。。。。。。。
PostConstructAndPreDestoryBean创建完成.......
PostConstructAndPreDestoryBean创建完成，使用@PostConstruct注解来调用初始化方法。。。。
ImportBeanDefinitionRegistrarBean创建完成，测试@Import接口通过ImportBeanDefinitionRegistrar接口注入组件。。。。。
----------------------IOC容器初始化完成------------------------

bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$7081dc62
LazyBeanService创建完成..............
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:com.ddf.spring.annotation.bean.ImportBean, type: class com.ddf.spring.annotation.bean.ImportBean
bean name:com.ddf.spring.annotation.bean.ImportSelectorBean, type: class com.ddf.spring.annotation.bean.ImportSelectorBean
```

```
bean name:user, type: class com.ddf.spring.annotation.entity.User
bean name:DevelopmentBeanLog, type: class com.ddf.spring.annotation.bean.DevelopmentBeanLog
bean name:DevelopmentBean, type: class com.ddf.spring.annotation.bean.DevelopmentBean
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
bean name:factoryPrototypeBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactoryPrototypeBean
FactorySingletonBean创建完成。。。。，测试通过FactoryBean来注册单实例组件。。。。
bean name:factorySingletonBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactorySingletonBean
bean name:initAndDisposableBean, type: class com.ddf.spring.annotation.bean.InitAndDisposableBean
bean name:postConstructAndPreDestoryBean, type: class com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
bean name:importBeanDefinitionRegistrarBean, type: class com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService创建完成。。。。。。。
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束-------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------


----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
单实例factorySingletonBean==factorySingletonBean1?true
Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?false
----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------

八月 01, 2018 11:08:04 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Wed Aug 01
23:08:03 CST 2018]; root of context hierarchy
PostConstructAndPreDestoryBean容器销毁，使用@PreDestroy注解来指定调用销毁方法。。。。
InitAndDisposableBean容器销毁，实现DisposableBean接口调用销毁方法.....
User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法.....

Process finished with exit code 0
```

3.6 BeanPostProcessor接口

> 参见`org.springframework.beans.factory.config.BeanPostProcessor`，该接口可以在每个`bean`的初始化方法调用之前和初始化方法调用之后执行指定的方法，
> `postProcessBeforeInitialization()`在每个bean创建之后的初始化方法之前调用，`postProcessAfterInitialization`在每个bean的初始化方法执行之后被调用,执行实际千万和前面提到的初始化方法和销毁方法区分开。该方法通常用户修改预定义的`bean`的属性值，可以实现该接口进行覆盖。更详细参见章节`扩展原理之1. BeanFactoryPostProcessor接口`

- 新建`CustomBeanPostProcessor.java`实现`BeanPostProcessor`接口

```java
package com.ddf.spring.annotation.configuration;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/2
 * BeanPostProcessor接口为bean的后置处理器，用来在bean的初始化前后做一些工作，需要将该类加入到容器中。
 * 需要理解的是，这个会在每个bean的生命周期内都会生效
 * postProcessBeforeInitialization()方法是在bean的初始化方法之前调用
 * postProcessAfterInitialization()方法是在bean的初始化方法之前调用之后执行
 *
 * 原理：
 * 遍历得到容器中所有的BeanPostProcessor；挨个执行beforeInitialization，
 * * 一但返回null，跳出for循环，不会执行后面的BeanPostProcessor.postProcessorsBeforeInitialization
 * *
 * * BeanPostProcessor原理
 * * populateBean(beanName, mbd, instanceWrapper);给bean进行属性赋值
 * * initializeBean
 * * {
 * * applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
 * * invokeInitMethods(beanName, wrappedBean, mbd);执行自定义初始化
 * * applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
 * *}
 *
 * Spring底层对 BeanPostProcessor 的使用：
 * * bean赋值，注入其他组件，@Autowired，生命周期注解功能，@Async,xxx BeanPostProcessor;
 */
@Component
public class CustomBeanPostProcessor implements BeanPostProcessor {

    /**
     * 在每个bean创建之后的初始化方法之前调用
     * @param bean 当前实例化的bean
     * @param beanName bean的名称
     * @return 返回实例化的bean或者可以对对象进行再封装返回
     * @throws BeansException
     */
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【" + bean + "】");
        return bean;
    }


    /**
     * 在每个bean的初始化方法执行之后被调用
     * @param bean 当前实例化的bean
     * @param beanName bean的名称
     * @return 返回实例化的bean或者可以对对象进行再封装返回
     * @throws BeansException
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【" + bean + "】");
        return bean;
    }
}
```

- 运行主启动类 `Application.java`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.*;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1? : " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }
```

```
    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }
}
```

- 控制台打印如下，仔细观察 `BeanPostProcessor` 接口方法的执行时机，是在每个 `bean` 的初始化方法之前和之后两个地方呗调用

```
----------------------IOC容器初始化-------------------------
八月 02, 2018 10:21:08 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Thu Aug 02 22:21:08 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ApplicationContextUtil
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomBeanPostProcessor
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportBeanDefinitionRegistrar
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportSelector
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryPrototypeBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactorySingletonBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportSelectorBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.InitAndDisposableBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
DevelopmentProfileCondition profile: prd
ProductionProfileCondition profile: prd
八月 02, 2018 10:21:08 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【org.springframework.context.event.EventListenerMethodProcessor@77be656f】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【org.springframework.context.event.EventListenerMethodProcessor@77be656f】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【org.springframework.context.event.DefaultEventListenerFactory@221af3c0】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【org.springframework.context.event.DefaultEventListenerFactory@221af3c0】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$a202c61@23a5fd2】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$a202c61@23a5fd2】
UserService创建完成...................
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.service.UserService@78a2da20】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.service.UserService@78a2da20】
ImportBean创建完成（测试@Import导入组件）..........
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportBean@dd3b207】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportBean@dd3b207】
ImportSelectorBean创建完成，测试@Import通过ImportSelector接口导入组件
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportSelectorBean@551bdc27】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportSelectorBean@551bdc27】
User创建完成...通过@Bean的initMethod调用初始化方法.............
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.entity.User@564718df】
User创建后调用初始化方法..........
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.entity.User@564718df】
ProductionBeanLog创建完成.......测试@Condition.............
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ProductionBeanLog@18a70f16】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ProductionBeanLog@18a70f16】
ProductionBean创建完成......测试@Condition.......
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ProductionBean@62e136d3】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ProductionBean@62e136d3】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration@4206a205】
```

```
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration@4206a205】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration@c540f5a】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration@c540f5a】
InitAndDisposableBean创建完成。。。。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.InitAndDisposableBean@4d826d77】
InitAndDisposableBean创建后实现InitializingBean调用初始化方法。。。。。。。。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.InitAndDisposableBean@4d826d77】
PostConstructAndPreDestoryBean创建完成.......
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean@44a664f2】
PostConstructAndPreDestoryBean创建完成，使用@PostConstruct注解来调用初始化方法。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean@44a664f2】
ImportBeanDefinitionRegistrarBean创建完成，测试@Import接口通过ImportBeanDefinitionRegistrar接口注入组件。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean@7f9fcf7f】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean@7f9fcf7f】
----------------------IOC容器初始化完成-------------------------

bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$a202c61
bean name:customBeanPostProcessor, type: class com.ddf.spring.annotation.configuration.CustomBeanPostProcessor
LazyBeanService创建完成.............
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.service.LazyBeanService@2c34f934】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.service.LazyBeanService@2c34f934】
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@12d3a4e9】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@12d3a4e9】
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:com.ddf.spring.annotation.bean.ImportBean, type: class com.ddf.spring.annotation.bean.ImportBean
bean name:com.ddf.spring.annotation.bean.ImportSelectorBean, type: class com.ddf.spring.annotation.bean.ImportSelectorBean
bean name:user, type: class com.ddf.spring.annotation.entity.User
bean name:prdServiceLog, type: class com.ddf.spring.annotation.bean.ProductionBeanLog
bean name:ProductionBean, type: class com.ddf.spring.annotation.bean.ProductionBean
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@240237d2】
bean name:factoryPrototypeBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactoryPrototypeBean
FactorySingletonBean创建完成。。。。，测试通过FactoryBean来注册单实例组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactorySingletonBean@25a65b77】
bean name:factorySingletonBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactorySingletonBean
bean name:initAndDisposableBean, type: class com.ddf.spring.annotation.bean.InitAndDisposableBean
bean name:postConstructAndPreDestoryBean, type: class com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
bean name:importBeanDefinitionRegistrarBean, type: class com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean

----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@548a102f】
```

```
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@548a102f】
PrototypeScopeService创建完成。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@5762806e】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@5762806e】
PrototypeScopeService prototype scope作用域是否相等: false
----------------------测试@Scope结束------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------


----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@17c386de】
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@5af97850】
单实例factorySingletonBean==factorySingletonBean1?true
Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?false
----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------

PostConstructAndPreDestoryBean容器销毁，使用@PreDestroy注解来指定调用销毁方法。。。。
InitAndDisposableBean容器销毁，实现DisposableBean接口调用销毁方法...........
User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法......
八月 02, 2018 10:21:08 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Thu Aug 02
22:21:08 CST 2018]; root of context hierarchy

Process finished with exit code 0
```

4. `Bean` 的属性赋值

4.1 `@Value` 与 `@PropertySource`

> `@Value` 可以标注在类的字段上，以表示该该字段赋值，可以直接使用目标值，支持 `SPEl` ，也支持外部配置文件加载
> `@PropertySource` 可以标注在类上， `value` 可以指定一个资源文件，则可以导入该文件以便解析读取

- 在 `src/main/resources` 下新建一个配置文件 `User.properties` ，用来给 `User` 初始化赋值

```
user.id=1
user.userName=ddf
user.password=123456
user.tel=18356785555
```

- 修改 `User.java` ，引入配置文件和给属性赋值

```java
package com.ddf.spring.annotation.entity;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/7/19
 * 使用@PropertySource引入外部配置文件
 * @Component或@Configuration将该类交由IOC容器保管（这里有一个很奇怪的地方，明明使用了@Bean在配置类中已经注入了这个Bean,但是如果
 * 这里只使用了@PropertySource依然无法对User赋值，所以这里需要再加上一个@Component，很奇怪）
 */
@PropertySource("classpath:User.properties")
@Component
public class User {
    @Value("${user.id}")
    private Integer id;
    @Value("${user.userName}")
    private String userName;
    @Value("${user.password}")
    private String password;
    @Value("${user.tel}")
    private String tel;
    @Value("用户数据")
    private String defaultMessage;

    public void init() {
        System.out.println("User创建后调用初始化方法.....");
    }

    public void destory() {
        System.out.println("User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法.....");
    }

    public User() {
        System.out.println("User创建完成...通过@Bean的initMethod调用初始化方法.....");
    }

    public User(Integer id, String userName, String password, String tel, String defaultMessage) {
        this.id = id;
        this.userName = userName;
        this.password = password;
        this.tel = tel;
        this.defaultMessage = defaultMessage;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
```

```
    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }

    public String getDefaultMessage() {
        return defaultMessage;
    }

    public void setDefaultMessage(String defaultMessage) {
        this.defaultMessage = defaultMessage;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", userName='" + userName + '\'' +
                ", password='" + password + '\'' +
                ", tel='" + tel + '\'' +
                ", defaultMessage='" + defaultMessage + '\'' +
                '}';
    }
}
```

- 修改 `Application.java`，增加测试代码 `testPropertySourceValue`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.*;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);
        // 测试@PropertySource和@Value属性赋值
        testPropertySourceValue(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1？: " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }
```

```
    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }

    /**
     * 测试通过@PropertySource和@Value注解来对属性进行赋值
     * @param applicationContext
     */
    public static void testPropertySourceValue(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试@PropertySource和@Value赋值开始----------------");
        User user = applicationContext.getBean(User.class);
        System.out.println("user属性为:  " + user.toString());
        System.out.println("---------------测试@PropertySource和@Value赋值结束----------------\n");

    }
}
```

- 运行类，日志如下

```
----------------------IOC容器初始化-------------------------
八月 04, 2018 7:14:19 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5197848c: startup date [Sat Aug 04 19:14:19 CST 2018]; root of context hierarchy
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.Application
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ApplicationContextUtil
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomBeanPostProcessor
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportBeanDefinitionRegistrar
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.CustomImportSelector
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.DevelopmentProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ExcludeTypeFilter
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.configuration.ProductionProfileCondition
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.controller.UserController
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.entity.User
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.DevelopmentBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.ExcludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactoryPrototypeBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.FactorySingletonBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ImportSelectorBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.IncludeFilterService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.InitAndDisposableBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.LazyBeanService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBean
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.bean.ProductionBeanLog
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.PrototypeScopeService
自定义扫描类规则当前扫描类为:  com.ddf.spring.annotation.service.UserService
八月 04, 2018 7:14:19 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition
信息: Overriding bean definition for bean 'user' with a different definition: replacing [Generic bean: class [com.ddf.spring.annotation.entity.User]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in file [J:\dev-tools\idea-root\spring-annotation\target\classes\com\ddf\spring\annotation\entity\User.class]] with [Root bean: class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=annotationConfiguration; factoryMethodName=user; initMethodName=init; destroyMethodName=destory; defined in com.ddf.spring.annotation.configuration.AnnotationConfiguration]
DevelopmentProfileCondition profile: dev
ProductionProfileCondition profile: dev
八月 04, 2018 7:14:19 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【org.springframework.context.event.EventListenerMethodProcessor@b7f23d9】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【org.springframework.context.event.EventListenerMethodProcessor@b7f23d9】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【org.springframework.context.event.DefaultEventListenerFactory@69b794e2】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【org.springframework.context.event.DefaultEventListenerFactory@69b794e2】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$c59c92dd@4d339552】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$c59c92dd@4d339552】
解析的字符串：你好 Windows 10 我是 360
com.ddf.spring.annotation.configuration.ApplicationContextUtil@45018215
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.ApplicationContextUtil@45018215】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.ApplicationContextUtil@45018215】
User创建完成...通过@Bean的initMethod调用初始化方法.....
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【User{id=1, userName='ddf', password='123456', tel='18356785555', defaultMessage='用户数据'}】
User创建后调用初始化方法.....
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【User{id=1, userName='ddf', password='123456', tel='18356785555', defaultMessage='用户数据'}】
UserService创建完成.....
```

```
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.service.UserService@47d90b9e】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.service.UserService@47d90b9e】
ImportBean创建完成（测试@Import导入组件）..........
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportBean@1184ab05】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportBean@1184ab05】
ImportSelectorBean创建完成，测试@Import通过ImportSelector接口导入组件
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportSelectorBean@3aefe5e5】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.ImportSelectorBean@3aefe5e5】
DevelopmentBeanLog创建完成.......测试@Condition........
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.DevelopmentBeanLog@149e0f5d】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.DevelopmentBeanLog@149e0f5d】
DevelopmentBean创建完成.....测试@Condition........
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.DevelopmentBean@1b1473ab】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.DevelopmentBean@1b1473ab】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration@ef9296d】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactoryPrototypeBeanConfiguration@ef9296d】
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration@1c93084c】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.configuration.FactorySingletonBeanConfiguration@1c93084c】
InitAndDisposableBean创建完成。。。。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.InitAndDisposableBean@7ce3cb8e】
InitAndDisposableBean创建后实现InitializingBean调用初始化方法。。。。。。。。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.InitAndDisposableBean@7ce3cb8e】
PostConstructAndPreDestoryBean创建完成.......
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean@22a637e7】
PostConstructAndPreDestoryBean创建完成，使用@PostConstruct注解来调用初始化方法。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean@22a637e7】
ImportBeanDefinitionRegistrarBean创建完成，测试@Import接口通过ImportBeanDefinitionRegistrar接口注入组件。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean@6fe7aac8】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean@6fe7aac8】
----------------------IOC容器初始化完成-------------------------

bean name:org.springframework.context.annotation.internalConfigurationAnnotationProcessor, type: class
org.springframework.context.annotation.ConfigurationClassPostProcessor
bean name:org.springframework.context.annotation.internalAutowiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalRequiredAnnotationProcessor, type: class
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor
bean name:org.springframework.context.annotation.internalCommonAnnotationProcessor, type: class
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor
bean name:org.springframework.context.event.internalEventListenerProcessor, type: class
org.springframework.context.event.EventListenerMethodProcessor
bean name:org.springframework.context.event.internalEventListenerFactory, type: class
org.springframework.context.event.DefaultEventListenerFactory
bean name:annotationConfiguration, type: class
com.ddf.spring.annotation.configuration.AnnotationConfiguration$$EnhancerBySpringCGLIB$$c59c92dd
bean name:applicationContextUtil, type: class com.ddf.spring.annotation.configuration.ApplicationContextUtil
bean name:customBeanPostProcessor, type: class com.ddf.spring.annotation.configuration.CustomBeanPostProcessor
bean name:user, type: class com.ddf.spring.annotation.entity.User
LazyBeanService创建完成.............
BeanPostProcessor的postProcessBeforeInitialization方法执行，当前bean【com.ddf.spring.annotation.service.LazyBeanService@1700915】
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.service.LazyBeanService@1700915】
bean name:lazyBeanService, type: class com.ddf.spring.annotation.service.LazyBeanService
PrototypeScopeService创建完成。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@21de60b4】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@21de60b4】
bean name:prototypeScopeService, type: class com.ddf.spring.annotation.service.PrototypeScopeService
bean name:userService, type: class com.ddf.spring.annotation.service.UserService
bean name:com.ddf.spring.annotation.bean.ImportBean, type: class com.ddf.spring.annotation.bean.ImportBean
bean name:com.ddf.spring.annotation.bean.ImportSelectorBean, type: class com.ddf.spring.annotation.bean.ImportSelectorBean
```

```
bean name:DevelopmentBeanLog, type: class com.ddf.spring.annotation.bean.DevelopmentBeanLog
bean name:DevelopmentBean, type: class com.ddf.spring.annotation.bean.DevelopmentBean
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@c267ef4】
bean name:factoryPrototypeBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactoryPrototypeBean
FactorySingletonBean创建完成。。。。，测试通过FactoryBean来注册单实例组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactorySingletonBean@30ee2816】
bean name:factorySingletonBeanConfiguration, type: class com.ddf.spring.annotation.bean.FactorySingletonBean
bean name:initAndDisposableBean, type: class com.ddf.spring.annotation.bean.InitAndDisposableBean
bean name:postConstructAndPreDestoryBean, type: class com.ddf.spring.annotation.bean.PostConstructAndPreDestoryBean
bean name:importBeanDefinitionRegistrarBean, type: class com.ddf.spring.annotation.bean.ImportBeanDefinitionRegistrarBean

-----------------------测试@Scope开始-------------------------
默认单实例bean UserService是否相等 true
PrototypeScopeService创建完成。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@7a69b07】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@7a69b07】
PrototypeScopeService创建完成。。。。。。。。
BeanPostProcessor的postProcessBeforeInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@5e82df6a】
BeanPostProcessor的postProcessAfterInitialization方法执行，当
前bean【com.ddf.spring.annotation.service.PrototypeScopeService@5e82df6a】
PrototypeScopeService prototype scope作用域是否相等: false
-----------------------测试@Scope结束-------------------------


---------------测试单实例bean的@Lazy懒加载开始----------------------
lazyBeanService==lazyBeanService1? : true
---------------测试单实例bean的@Lazy懒加载结束----------------------


----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@3f197a46】
FactoryPrototypeBean创建完成....，测试通过FactoryBean来注册Prototype组件。。。。
BeanPostProcessor的postProcessAfterInitialization方法执行，当前bean【com.ddf.spring.annotation.bean.FactoryPrototypeBean@636be97c】
单实例factorySingletonBean==factorySingletonBean1?true
Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?false
----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------


---------------测试@PropertySource和@Value赋值开始----------------
user属性为： User{id=1, userName='ddf', password='123456', tel='18356785555', defaultMessage='用户数据'}
---------------测试@PropertySource和@Value赋值结束----------------

PostConstructAndPreDestoryBean容器销毁，使用@PreDestroy注解来指定调用销毁方法。。。。
InitAndDisposableBean容器销毁，实现DisposableBean接口调用销毁方法.....
User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法.....
```

5. `Bean` 的依赖注入

> 如何将一个组件自动注入到另外一个组件中呢?

5.1 `@Autowired`

> `@Autowired` 可以可以标注在构造器，方法，参数，字段上，默认按照 `Bean` 的类型去 `IOC` 容器中去寻找组件，如果有且一个组件被找到则成功注入，如果有多个组件被找到，则再使用默认名称去获取 `Bean` ，`Bean` 的默认名称为类首字母缩写。也可以配合 `@Qualifier` 注解明确指定注入哪个 `Bean` ，还可以在注册 `Bean` 的地方使用 `@Primary` ，在不指定 `@Qualifier` 的情况下，默认注入@Primary修饰的Bean。如果 `Bean` 不一定存在，可以使用属性 `required=false` ，则 `Bean` 不存在也不会抛出异常

- 编写一个用来注入到另外一个类中的 `Bean AutowiredBean.java` 并使用 `@Component` 注入到容器中

```java
package com.ddf.spring.annotation.bean;

import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/6
 */
@Component
public class AutowiredBean {
    public AutowiredBean() {
        System.out.println("AutowiredBean创建完成。。。。。。。。。。。。");
    }
}
```

- 修改 `AnnotationConfiguration.java`，增加方法 `autowiredBean2()`,再次注入 `AutowiredBean`，并取名 `autowiredBean2` 区分

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.*;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 * @Import 导入组件，可以直接导入普通类，或者通过ImportSelector接口或者ImportBeanDefinitionRegistrar接口来自定义导入
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * initMethod 指定Bean创建后调用的初始化方法
     * destroyMethod 指定Bean在销毁后会调用的方法
     * @return
     */
    @Bean(initMethod = "init", destroyMethod = "destory")
    public User user() {
        return new User();
    }


    /**
     * 测试@Conditional 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean developmentService() {
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean productionService() {
        return new ProductionBean();
    }


    /**
     * 使用FactoryBean工厂来注册组件
     * @return
     */
    @Bean
    public FactoryPrototypeBeanConfiguration factoryPrototypeBeanConfiguration() {
        return new FactoryPrototypeBeanConfiguration();
    }

    /**
     * 使用FactoryBean工厂来注册组件
     * @return
```

```
     */
    @Bean
    public FactorySingletonBeanConfiguration factorySingletonBeanConfiguration() {
        return new FactorySingletonBeanConfiguration();
    }


    /**
     * 注册一个实现InitializingBean, DisposableBean接口来指定Bean的初始化和销毁方法的Bean
     * @return
     */
    @Bean
    public InitAndDisposableBean initAndDisposableBean() {
        return new InitAndDisposableBean();
    }

    /**
     * 创建一个通过JSR250 @PostConstruct指定初始化方法/@PreDestroy指定销毁方法的Bean
     * @return
     */
    @Bean
    public PostConstructAndPreDestoryBean postConstructAndPreDestoryBean() {
        return new PostConstructAndPreDestoryBean();
    }


    /**
     * 注入AutowiredBean，名称为autowiredBean2，并将该bean作为默认依赖注入的首选
     * @return
     */
    @Bean
    @Primary
    public AutowiredBean autowiredBean2() {
        return new AutowiredBean();
    }
}
```

- 新建一个 `AutowiredService.java` 并使用 `@Service` 注解交由容器管理，然后注入 `AutowiredBean`

```java
package com.ddf.spring.annotation.service;

import com.ddf.spring.annotation.bean.AutowiredBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/8/6
 * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
 * * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
 * * 默认注入@Primary修饰的Bean，如果Bean不一定存在，可以使用属性required=false，则Bean不存在也不会抛出异常
 * 可以标注在构造器，方法，参数，字段上
 *
 * @Resource 默认使用名称注入Bean,可以使用name属性指定具体要注入的Bean的名称，不支持@Primary，不支持required=false,不支持@Primary
 * @Inject 使用类型去注入，支持@Primary，不支持required=false
 *
 *
 */
@Service
public class AutowiredService {
    @Autowired
    private AutowiredBean autowiredBean;
    public AutowiredBean getAutowiredBean() {
        return autowiredBean;
    }

    @Autowired
    @Qualifier("autowiredBean")
    private AutowiredBean qualifierAutowiredBean;
    public AutowiredBean getQualifierAutowiredBean() {
        return qualifierAutowiredBean;
    }

    public AutowiredService() {
        System.out.println("AutowiredService创建完成。。。。。。。。。。。。");
    }
}
```

- 测试见 5.3 综合演示测试结果

5.2 `@Resource` 和 `@Inject`

> 这两个注解也可以完成依赖注入，并且是 `java` 规范提供的。区别为 `@Resource` 默认是使用名称来注入组件的，并且不支持 `@Primary`，也不支持 `required=false`，有一个属性名为name可以指定要注入的Bean的名称； `@Inject` 和 `@Autowired` 类似，按照类型注入，支持 `@Primary`，不支持 `required=false`

- 修改 `AutowiredService.java` ,加入 `@Resource` 和 `@Inject` 的注入代码

```java
package com.ddf.spring.annotation.service;

import com.ddf.spring.annotation.bean.AutowiredBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import javax.inject.Inject;

/**
 * @author DDf on 2018/8/6
 * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
 * * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
 * * 默认注入@Primary修饰的Bean，如果Bean不一定存在，可以使用属性required=false，则Bean不存在也不会抛出异常
 * 可以标注在构造器，方法，参数，字段上
 *
 * @Resource 默认使用名称注入Bean,可以使用name属性指定具体要注入的Bean的名称，不支持@Primary，不支持required=false,不支持@Primary
 * @Inject 使用类型去注入，支持@Primary，不支持required=false
 *
 *
 */
@Service
public class AutowiredService {
    @Autowired
    private AutowiredBean autowiredBean;
    public AutowiredBean getAutowiredBean() {
        return autowiredBean;
    }

    @Autowired
    @Qualifier("autowiredBean")
    private AutowiredBean qualifierAutowiredBean;
    public AutowiredBean getQualifierAutowiredBean() {
        return qualifierAutowiredBean;
    }

    @Resource(name = "autowiredBean")
    private AutowiredBean resourceAutowiredBean;
    public AutowiredBean getResourceAutowiredBean() {
        return resourceAutowiredBean;
    }


    @Resource(name = "autowiredBean2")
    private AutowiredBean resourceAutowiredBean2;
    public AutowiredBean getResourceAutowiredBean2() {
        return resourceAutowiredBean2;
    }

    @Inject
    private UserService userService;
    public UserService getUserService() {
        return userService;
    }

    public AutowiredService() {
        System.out.println("AutowiredService创建完成。。。。。。。。。。。。");
    }
}
```

- 测试见 5.3 综合演示测试结果

5.3 综合演示测试结果

- 修改主启动类 `Application.java`，增加测试依赖注入的方法 `testAutowired()`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.AutowiredService;
import com.ddf.spring.annotation.service.LazyBeanService;
import com.ddf.spring.annotation.service.PrototypeScopeService;
import com.ddf.spring.annotation.service.UserService;
import com.ddf.spring.annotation.bean.AutowiredBean;
import com.ddf.spring.annotation.bean.FactoryPrototypeBean;
import com.ddf.spring.annotation.bean.FactorySingletonBean;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.entity.User;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Primary;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }

        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);
        // 测试@PropertySource和@Value属性赋值
        testPropertySourceValue(applicationContext);
        // 测试@Autowired注入多个相同类型的Bean
        testAutowired(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
```

```
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1？: " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }


    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }

    /**
     * 测试通过@PropertySource和@Value注解来对属性进行赋值
     * @param applicationContext
     */
    public static void testPropertySourceValue(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试@PropertySource和@Value赋值开始----------------");
        User user = applicationContext.getBean(User.class);
        System.out.println("user属性为： " + user.toString());
        System.out.println("---------------测试@PropertySource和@Value赋值结束----------------\n");

    }

    /**
     * 测试在IOC容器中存在两个相同类型的Bean,但是Bean的名称不一致
     * 在这种情况下，使用@Autowired将该Bean注入到另外一个容器中
     * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
     * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
     * 默认注入哪个Bean {@link AutowiredService}
     * @param applicationContext
     */
    public static void testAutowired(ApplicationContext applicationContext) {
        System.out.println("\n--------------测试autowired注入多个相同类型的类开始-----------------");
        AutowiredBean autowiredBean = (AutowiredBean) applicationContext.getBean("autowiredBean");
        AutowiredBean autowiredBean2 = (AutowiredBean) applicationContext.getBean("autowiredBean2");
        System.out.println("autowiredBean: " + autowiredBean);
        System.out.println("autowiredBean2: " + autowiredBean2);
        System.out.println(autowiredBean == autowiredBean2);

        /**
         * 这里已做更改，修改了默认注入 {@link com.ddf.spring.annotation.configuration.AnnotationConfiguration.autowiredBean2}
         */
        AutowiredService autowiredService = applicationContext.getBean(AutowiredService.class);
        AutowiredBean autowiredServiceBean = autowiredService.getAutowiredBean();
        System.out.println("使用@Primay后AutowiredService默认注入bean: " + autowiredServiceBean);

        AutowiredBean autowiredServiceBean2 = autowiredService.getQualifierAutowiredBean();
        System.out.println("使用@Qualifier明确注入Bean: " + autowiredServiceBean2);

        // 使用@Resource注入
        AutowiredBean resourceAutowiredBean = autowiredService.getResourceAutowiredBean();
```

```
        System.out.println("使用@Resource注入autowiredBean: " + resourceAutowiredBean);
        AutowiredBean resourceAutowiredBean2 = autowiredService.getResourceAutowiredBean2();
        System.out.println("使用@Resource注入autowiredBean2: " + resourceAutowiredBean2);

        // 使用@Inject注入
        UserService userService = autowiredService.getUserService();
        System.out.println("使用@Inject注入UserService:  " + userService);

        System.out.println("--------------测试autowired注入多个相同类型的类开始-----------------\n");
    }
}
```

- 打印日志如下,不相关日志已过滤

```
// 过滤。。。
--------------测试autowired注入多个相同类型的类开始-----------------
autowiredBean: com.ddf.spring.annotation.bean.AutowiredBean@77e4c80f
autowiredBean2: com.ddf.spring.annotation.bean.AutowiredBean@1d119efb
false
使用@Primay后AutowiredService默认注入bean: com.ddf.spring.annotation.bean.AutowiredBean@1d119efb
使用@Qualifier明确注入Bean: com.ddf.spring.annotation.bean.AutowiredBean@77e4c80f
使用@Resource注入autowiredBean: com.ddf.spring.annotation.bean.AutowiredBean@77e4c80f
使用@Resource注入autowiredBean2: com.ddf.spring.annotation.bean.AutowiredBean@1d119efb
使用@Inject注入UserService:  com.ddf.spring.annotation.service.UserService@4b5d6a01
--------------测试autowired注入多个相同类型的类开始-----------------
```

6. `@Profile` 根据环境注入 `Bean`

> `@Profile` 注解，Spring为我们提供的可以根据当前环境，动态的激活和切换一系列组件的功能，加了环境标识的bean，只有这个环境被激活的时候才能注册到容器中。默认是default环境，写在配置类上，只有是指定的环境的时候，整个配置类里面的所有配置才能开始生效，没有标注环境标识的bean在，任何环境下都是加载的；

演示场景：

> 加入有一个日志管理接口类，在这个接口下实现不同的具体日志实现方式，根据不同的环境来注入不同的日志实现类。如，当前有一个接口类 `Slf4jBean` ，有 `Log4jBean` 和 `LogbackBean` 两个实现，现决定如果使 `dev` 环境，则使用 `Lo4jBean` 来管理日志，如果 `prd` 环境，则使用 `LogbackBean` ；

- 新建日志接口类 `Slf4jBean.java`

```
package com.ddf.spring.annotation.bean;

/**
 * @author DDf on 2018/8/8
 * 测试根据不同的profile来动态切换注册不同的类，该类为接口，使用接口接收参数，实际注入值为接口实现类
 */
public interface Slf4jBean {
    void info(String str);
}
```

- 新建日志实现类 `Log4jBean.java`

```
package com.ddf.spring.annotation.bean;

/**
 * @author DDf on 2018/8/8
 */
public class Log4jBean implements Slf4jBean {

    @Override
    public void info(String str) {
        System.out.println(this.getClass().getName() + ": " + str);
    }
}
```

- 新建另一个日志实现类 `LogbackBean.java`

```
package com.ddf.spring.annotation.bean;

/**
 * @author DDf on 2018/8/8
 */
public class LogbackBean implements Slf4jBean {
    @Override
    public void info(String str) {
        System.out.println(this.getClass().getName() + ": " + str);
    }
}
```

- 新建一个配置类 `ProfileConfiguration.java` ，使用 `@Profile` 注解在方法上根据不同环境注入不同的类
  其实这一块完全可以使用之前一直使用的配置类 `AnnotationConfiguration` ，只不过切换环境牵扯到多次的重启 `IOC` 容器，而之前的测试大多都是在容器启动时测试的，因此有大量的打印在控制台上，如果继续使用这个配置类，会在大量的日志里看到当前的测试，会非常不易观察，因此这一块单独出来用一个新的配置类

```
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.Log4jBean;
import com.ddf.spring.annotation.bean.LogbackBean;
import com.ddf.spring.annotation.bean.Slf4jBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

/**
 * @author DDf on 2018/8/8
 * 配置类2，因为一些特殊的测试需要重新启动IOC容器，在原来的测试类上有太多测试代码，重新启动后，有太多的打印影响观看，
 * 所以这里单独独立出来一个配置类，实际情况中一个配置类是完全可以的
 */
@Configuration
public class ProfileConfiguration {

    /**
     * 使用接口的形式根据环境注入接口的实现类
     * 如果当前环境是dev，Log4jBean
     * @return
     */
    @Bean
    @Profile("dev")
    public Slf4jBean log4jBean() {
        return new Log4jBean();
    }

    /**
     * 使用接口的形式根据环境注入接口的实现类
     * 如果当前环境是prd，则注入LogbackBean
     * @return
     */
    @Bean
    @Profile("prd")
    public Slf4jBean logbackBean() {
        return new LogbackBean();
    }
}
```

- 修改主启动类 `Application.java` ，增加测试 `@profile` 的方法 `testProfile`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.service.AutowiredService;
import com.ddf.spring.annotation.service.LazyBeanService;
import com.ddf.spring.annotation.service.PrototypeScopeService;
import com.ddf.spring.annotation.service.UserService;
import com.ddf.spring.annotation.bean.AutowiredBean;
import com.ddf.spring.annotation.bean.FactoryPrototypeBean;
import com.ddf.spring.annotation.bean.FactorySingletonBean;
import com.ddf.spring.annotation.bean.Slf4jBean;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.ProfileConfiguration;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        printBeans(applicationContext);
        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);
        // 测试@PropertySource和@Value属性赋值
        testPropertySourceValue(applicationContext);
        // 测试@Autowired注入多个相同类型的Bean
        testAutowired(applicationContext);
        // 测试@Profile根据环境注入Bean
        testProfile(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 打印当前IOC容器中所有预定义的Bean
     * @param applicationContext
     */
    private static void printBeans(AnnotationConfigApplicationContext applicationContext) {
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
```

```
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1？: " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }


    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }

    /**
     * 测试通过@PropertySource和@Value注解来对属性进行赋值
     * @param applicationContext
     */
    public static void testPropertySourceValue(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试@PropertySource和@Value赋值开始----------------");
        User user = applicationContext.getBean(User.class);
        System.out.println("user属性为:  " + user.toString());
        System.out.println("---------------测试@PropertySource和@Value赋值结束----------------\n");

    }

    /**
     * 测试在IOC容器中存在两个相同类型的Bean,但是Bean的名称不一致
     * 在这种情况下，使用@Autowired将该Bean注入到另外一个容器中
     * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
     * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
     * 默认注入哪个Bean {@link AutowiredService}
     * @param applicationContext
     */
    public static void testAutowired(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试autowired注入多个相同类型的类开始-----------------");
        AutowiredBean autowiredBean = (AutowiredBean) applicationContext.getBean("autowiredBean");
        AutowiredBean autowiredBean2 = (AutowiredBean) applicationContext.getBean("autowiredBean2");
        System.out.println("autowiredBean: " + autowiredBean);
        System.out.println("autowiredBean2: " + autowiredBean2);
        System.out.println(autowiredBean == autowiredBean2);

        /**
         * 这里已做更改，修改了默认注入 {@link com.ddf.spring.annotation.configuration.AnnotationConfiguration.autowiredBean2}
```

```
         */
        AutowiredService autowiredService = applicationContext.getBean(AutowiredService.class);
        AutowiredBean autowiredServiceBean = autowiredService.getAutowiredBean();
        System.out.println("使用@Primay后AutowiredService默认注入bean: " + autowiredServiceBean);

        AutowiredBean autowiredServiceBean2 = autowiredService.getQualifierAutowiredBean();
        System.out.println("使用@Qualifier明确注入Bean: " + autowiredServiceBean2);

        // 使用@Resource注入
        AutowiredBean resourceAutowiredBean = autowiredService.getResourceAutowiredBean();
        System.out.println("使用@Resource注入autowiredBean: " + resourceAutowiredBean);
        AutowiredBean resourceAutowiredBean2 = autowiredService.getResourceAutowiredBean2();
        System.out.println("使用@Resource注入autowiredBean2: " + resourceAutowiredBean2);

        // 使用@Inject注入
        UserService userService = autowiredService.getUserService();
        System.out.println("使用@Inject注入UserService:  " + userService);

        System.out.println("--------------测试autowired注入多个相同类型的类开始----------------\n");
    }


    /**
     * 测试根据激活的Profile来根据环境注册不同的Bean
     * 切换profile有两种方式:
     * 1. 在java虚拟机启动参数加 -Dspring.profiles.active=test
     * 2. 如下演示，使用代码切换，可以将切换的变量放在配置文件，spring-boot配置文件即是这种方式
     * @param applicationContext
     */
    public static void testProfile(AnnotationConfigApplicationContext applicationContext) {
        System.out.println("------------------测试@Profile-------------------------");
        // 重新新建一个IOC容器
        applicationContext = new AnnotationConfigApplicationContext();
        // 注册配置类
        applicationContext.register(ProfileConfiguration.class);
        // 设置当前激活的环境profile
        applicationContext.getEnvironment().setActiveProfiles("dev");
        // 刷新容器
        applicationContext.refresh();

        // 使用接口获得实际接口实现类的注入Bean
        Slf4jBean devLogBean = applicationContext.getBean(Slf4jBean.class);
        devLogBean.info("测试环境");

        applicationContext = new AnnotationConfigApplicationContext();
        applicationContext.register(ProfileConfiguration.class);
        applicationContext.getEnvironment().setActiveProfiles("prd");
        applicationContext.refresh();

        Slf4jBean prdLogBean = applicationContext.getBean(Slf4jBean.class);
        prdLogBean.info("生产环境");

        System.out.println("------------------测试@Profile-------------------------");
    }
}
```

- 打印日志如下，可以看到首先使用的 `dev` 环境，然后打印的是 `com.ddf.spring.annotation.bean.Log4jBean: 测试环境`，然后切换为 `prd` 环境，打印的是 `com.ddf.spring.annotation.bean.LogbackBean: 生产环境`

```
.......省略其它不相关日志.............
------------------测试@Profile-------------------------
PostConstructAndPreDestoryBean容器销毁，使用@PreDestroy注解来指定调用销毁方法。。。。
InitAndDisposableBean容器销毁，实现DisposableBean接口调用销毁方法...........
User销毁后调用销毁方法....通过@Bean的destoryMethod指定销毁方法......
com.ddf.spring.annotation.bean.Log4jBean: 测试环境
com.ddf.spring.annotation.bean.LogbackBean: 生产环境
------------------测试@Profile-------------------------
```

7. 扩展 `Aware` 接口

> 使用相关的Aware接口可以直接获取到Spring底层设计的容器相关的类和属性

```
package com.ddf.spring.annotation.configuration;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.EmbeddedValueResolverAware;
import org.springframework.stereotype.Component;
import org.springframework.util.StringValueResolver;

/**
 * @author DDf on 2018/8/2
 * 使用Aware可以获得的属性
 * ApplicationContextAware 可以获取到IOC容器
 * BeanNameAware 可以获取到当前bean的名称
 * EmbeddedValueResolverAware
 */
@Component
public class ApplicationContextUtil implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {
    private ApplicationContext applicationContext;
    private String beanName;

    /**
     * 获得当前的bean的名称
     * @param name
     */
    @Override
    public void setBeanName(String name) {
        this.beanName = name;
    }

    /**
     * 获得当前的IOC容器
     * @param applicationContext
     * @throws BeansException
     */
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
        System.out.println(this.applicationContext.getBean(beanName));
    }

    /**
     * 字符串解析器
     * @param resolver
     */
    @Override
    public void setEmbeddedValueResolver(StringValueResolver resolver) {
        String resolveStringValue = resolver.resolveStringValue("你好 ${os.name} 我是 #{20*18}");
        System.out.println("解析的字符串："+resolveStringValue);
    }
}
```

二、 `AOP`

> 这一章节开始讲述有关 `AOP` 相关的注解内容，即面向切面变成， `AOP` 可以很方便的在某个方法某个业务逻辑前后，无侵入式的加入逻辑代码或者日志或者权限等

1. `@Aspect` 自定义切面

> 该注解可以用来标注一个类，则该类会被识别成一个 `AOP` 类，同时该类必须也需要交由容器管理，如使用 `@Component` ，切面包含了一系列的注解来定义切面语法如 `@Pointcut` ，
> 定义了切面方法的执行时机，如 `@Before`

@EnableAspectJAutoProxy	开启基于注解的aop模式,如果使用aop则必须在配置类上加入这个注解
@Aspect	表明当前是一个切面类，需要配合@Component等类似功能注解，交由容器管理
@Pointcut	定义一个切面语法，该语法直接影响到当前类的实际作用范围
@Before	在目标方法执行之前执行
@After	在目标方法执行结束之后执行
@AfterReturning	在结果值返回之后，可以在这里获取到方法的返回值,如果在结果值返回之前出现i异常，则这个方法不会被执行
@AfterThrowing	如果方法出现异常，则执行该方法
@Around	环绕通知，在使用了以上注解，再使用这个之后，发现方法正常执行后，结果值不能正确获取，方法出现异常，依然能执行到@AfterReturning，而且控制台异常被屏蔽，这个不做详细演示

- 使用 `@Aspect` 定义一个切面类,并使用 `@Component` 将该类交由容器管理,定义切面语法

```java
package com.ddf.spring.annotation.configuration;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;

import java.util.Arrays;

/**
 * @author DDf on 2018/8/10
 *
 * 标注一个AOP切面类
 * @Aspect 表明当前是一个切面类
 * @Component 必须将切面类交由容器管理，同时切面类也仅对同在容器中的类提供支持，仅仅满足切面语法不在类却不在容器中也是不行的，
 *
 * 还有一个很重要的地方，切面类的语法不能包含本切面类，如果本切面类也满足自己定义的切面语法，会出现问题
 *
 *
 */
@Aspect
@Component
public class LogAspect {

    /**
     * 使用@Pointcut来定义一个通用的切面表达式，方法体不会被执行，而别的方法切面类可以直接引用当前方法来复用表达式
     * 切面语法：
     * 首先是固定写法execution(访问修饰符 返回值 类路径 方法(参数1,参数2等))
     *
     * *的省略原则
     * 如果有一层是希望不加以限制的，则可以使用*代替，而如果连续的两个位置都不加以限制，则可以直接使用一个*号
     *
     * 包与子包
     * 如果类路径只限制到某个包下所有类则直接包名然后.*即可，但如果指定到某个包，某个包下面本身有类，而包下面又有子包，
     * 则可以指定到当前包之后使用两个.然后再*,如..*
     *
     * 参数的省略
     * 如果明确指定两个参数并且指定类型，则可以(string,string)
     * 但是如果一个参数固定，另外一个不加以限制，则可以(string,*)
     * 如果参数类型和数量都不加以限制，则可以(..)
     *
     * 如下示例，则拦截com.ddf.spring.annotation.service包以及所有子包下的所有public的方法（返回值不限，参数不限）
     *
     */
    @Pointcut("execution(public * com.ddf.spring.annotation.service..*.*(..))")
    public void pointcut() {};

    /**
     * 在目标方法执行之前执行，获取参数列表可以获取到Body参数，可以获取到Body参数，可以获取到Body参数!
     * @param joinPoint
     */
    @Before("pointcut()")
    public void beforeLog(JoinPoint joinPoint) {
        // 可以获取到Body参数，可以获取到Body参数，可以获取到Body参数!
        Object[] args = joinPoint.getArgs();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(methodName + "开始执行,参数列表" + Arrays.asList(args));
    }


    /**
     * 在目标方法执行结束之后执行
     * @param joinPoint
     */
    @After("pointcut()")
    public void afterLog(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(methodName + "结束执行,参数列表" + Arrays.asList(args));
    }
```

```
    /**
     * 在结果值返回之后，可以在这里获取到方法的返回值,如果在结果值返回之前出现i异常，则这个方法不会被执行
     * @param joinPoint 这个参数要保证在第一位
     * @param result 返回结果
     */
    @AfterReturning(value = "pointcut()", returning = "result")
    public void afterReturning(JoinPoint joinPoint, Object result) {
        Object[] args = joinPoint.getArgs();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(methodName + "结束执行,参数列表" + Arrays.asList(args) + ", 返回结果: " + result);
    }


    /**
     * 如果方法出现异常，则执行该方法
     * @param joinPoint
     * @param exception
     */
    @AfterThrowing(value="pointcut()",throwing="exception")
    public void logException(JoinPoint joinPoint,Exception exception){
        Object[] args = joinPoint.getArgs();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(methodName + "结束执行,参数列表" + Arrays.asList(args) + ", 出现异常: " + exception);
    }
}
```

- 修改 `UserService.java`，加入两个验证方法，一定要希望被拦截的目标类也必须被 `Spring` 管理

```
package com.ddf.spring.annotation.service;

import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/7/19
 * 默认@Scope为singleton，IOC容器启动会创建该bean的实例，并且以后再次使用不会重新创建新的实例
 */
@Service
public class UserService {
    public UserService() {
        System.out.println("UserService创建完成.....................");
    }

    public String welcome(String userName) {
        return "Hello " + userName;
    }

    public String welcomeException(String userName) {
        throw new RuntimeException("出现异常");
    }
}
```

- 修改主配置类 `AnnotationConfiguration.java`，加入 `@EnableAspectJAutoProxy`，开启基于注解的 `AOP` 代理

```java
package com.ddf.spring.annotation.configuration;

import com.ddf.spring.annotation.bean.*;
import com.ddf.spring.annotation.entity.User;
import org.springframework.context.annotation.*;
import org.springframework.stereotype.Controller;

/**
 * @author DDf on 2018/7/19
 * @Configuration 表明当前类是一个配置类
 * @ComponentScan 指定扫描的包路径，并且配置了excludeFilters来排除注解类型为@Controller的不纳入容器中，
 * 排除符合自定义ExcludeTypeFilter类中规则的类
 * @Import 导入组件，可以直接导入普通类，或者通过ImportSelector接口或者ImportBeanDefinitionRegistrar接口来自定义导入
 *
 * @EnableAspectJAutoProxy 开启基于注解的aop模式
 */
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
@EnableAspectJAutoProxy
public class AnnotationConfiguration {

    /**
     * 注入一个Type为User（方法返回值）的bean，bean的名称为user（方法名）
     * initMethod 指定Bean创建后调用的初始化方法
     * destroyMethod 指定Bean在销毁后会调用的方法
     * @return
     */
    @Bean(initMethod = "init", destroyMethod = "destory")
    public User user() {
        return new User();
    }


    /**
     * 测试@Conditional 满足{@link DevelopmentProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({DevelopmentProfileCondition.class})
    public DevelopmentBean developmentService() {
        return new DevelopmentBean();
    }


    /**
     * 满足{@link ProductionProfileCondition} 这个类的条件返回true则当前Bean能够成功注入，反之不能
     *
     * @return
     */
    @Bean
    @Conditional({ProductionProfileCondition.class})
    public ProductionBean productionService() {
        return new ProductionBean();
    }


    /**
     * 使用FactoryBean工厂来注册组件
     * @return
     */
    @Bean
    public FactoryPrototypeBeanConfiguration factoryPrototypeBeanConfiguration() {
        return new FactoryPrototypeBeanConfiguration();
    }
```

```
    /**
     * 使用FactoryBean工厂来注册组件
     * @return
     */
    @Bean
    public FactorySingletonBeanConfiguration factorySingletonBeanConfiguration() {
        return new FactorySingletonBeanConfiguration();
    }


    /**
     * 注册一个实现InitializingBean, DisposableBean接口来指定Bean的初始化和销毁方法的Bean
     * @return
     */
    @Bean
    public InitAndDisposableBean initAndDisposableBean() {
        return new InitAndDisposableBean();
    }

    /**
     * 创建一个通过JSR250 @PostConstruct指定初始化方法/@PreDestroy指定销毁方法的Bean
     * @return
     */
    @Bean
    public PostConstructAndPreDestoryBean postConstructAndPreDestoryBean() {
        return new PostConstructAndPreDestoryBean();
    }

    /**
     * 注入AutowiredBean，名称为autowiredBean2，并将该bean作为默认依赖注入的首选
     * @return
     */
    @Bean
    @Primary
    public AutowiredBean autowiredBean2() {
        return new AutowiredBean();
    }
}
```

- 修改主启动类 `Application.java` ,增加测试方法 `testAspect`

```java
package com.ddf.spring.annotation;

import com.ddf.spring.annotation.bean.AutowiredBean;
import com.ddf.spring.annotation.bean.FactoryPrototypeBean;
import com.ddf.spring.annotation.bean.FactorySingletonBean;
import com.ddf.spring.annotation.bean.Slf4jBean;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.ProfileConfiguration;
import com.ddf.spring.annotation.entity.User;
import com.ddf.spring.annotation.service.AutowiredService;
import com.ddf.spring.annotation.service.LazyBeanService;
import com.ddf.spring.annotation.service.PrototypeScopeService;
import com.ddf.spring.annotation.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        printBeans(applicationContext);
        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);
        // 测试@PropertySource和@Value属性赋值
        testPropertySourceValue(applicationContext);
        // 测试@Autowired注入多个相同类型的Bean
        testAutowired(applicationContext);
        // 测试@Profile根据环境注入Bean
        testProfile(applicationContext);
        // 测试AOP
        testAspect(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 打印当前IOC容器中所有预定义的Bean
     * @param applicationContext
     */
    private static void printBeans(AnnotationConfigApplicationContext applicationContext) {
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }
    }


    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n----------------------测试@Scope开始-------------------------");
```

```
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1？: " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }



    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }

    /**
     * 测试通过@PropertySource和@Value注解来对属性进行赋值
     * @param applicationContext
     */
    public static void testPropertySourceValue(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试@PropertySource和@Value赋值开始----------------");
        User user = applicationContext.getBean(User.class);
        System.out.println("user属性为:  " + user.toString());
        System.out.println("---------------测试@PropertySource和@Value赋值结束----------------\n");

    }

    /**
     * 测试在IOC容器中存在两个相同类型的Bean,但是Bean的名称不一致
     * 在这种情况下，使用@Autowired将该Bean注入到另外一个容器中
     * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
     * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
     * 默认注入哪个Bean {@link AutowiredService}
     * @param applicationContext
     */
    public static void testAutowired(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试autowired注入多个相同类型的类开始-----------------");
        AutowiredBean autowiredBean = (AutowiredBean) applicationContext.getBean("autowiredBean");
        AutowiredBean autowiredBean2 = (AutowiredBean) applicationContext.getBean("autowiredBean2");
        System.out.println("autowiredBean: " + autowiredBean);
        System.out.println("autowiredBean2: " + autowiredBean2);
        System.out.println(autowiredBean == autowiredBean2);
```

```
    /**
     * 这里已做更改，修改了默认注入 {@link com.ddf.spring.annotation.configuration.AnnotationConfiguration.autowiredBean2}
     */
    AutowiredService autowiredService = applicationContext.getBean(AutowiredService.class);
    AutowiredBean autowiredServiceBean = autowiredService.getAutowiredBean();
    System.out.println("使用@Primay后AutowiredService默认注入bean: " + autowiredServiceBean);

    AutowiredBean autowiredServiceBean2 = autowiredService.getQualifierAutowiredBean();
    System.out.println("使用@Qualifier明确注入Bean: " + autowiredServiceBean2);

    // 使用@Resource注入
    AutowiredBean resourceAutowiredBean = autowiredService.getResourceAutowiredBean();
    System.out.println("使用@Resource注入autowiredBean: " + resourceAutowiredBean);
    AutowiredBean resourceAutowiredBean2 = autowiredService.getResourceAutowiredBean2();
    System.out.println("使用@Resource注入autowiredBean2: " + resourceAutowiredBean2);

    // 使用@Inject注入
    UserService userService = autowiredService.getUserService();
    System.out.println("使用@Inject注入UserService:  " + userService);

    System.out.println("--------------测试autowired注入多个相同类型的类结束-----------------\n");
}


/**
 * 测试根据激活的Profile来根据环境注册不同的Bean
 * 切换profile有两种方式:
 * 1. 在java虚拟机启动参数加 -Dspring.profiles.active=test
 * 2. 如下演示，使用代码切换，可以将切换的变量放在配置文件，spring-boot配置文件即是这种方式
 * @param applicationContext
 */
public static void testProfile(AnnotationConfigApplicationContext applicationContext) {
    System.out.println("\n------------------测试@Profile开始-------------------------");
    // 重新新建一个IOC容器
    applicationContext = new AnnotationConfigApplicationContext();
    // 注册配置类
    applicationContext.register(ProfileConfiguration.class);
    // 设置当前激活的环境profile
    applicationContext.getEnvironment().setActiveProfiles("dev");
    // 刷新容器
    applicationContext.refresh();

    // 使用接口获得实际接口实现类的注入Bean
    Slf4jBean devLogBean = applicationContext.getBean(Slf4jBean.class);
    devLogBean.info("测试环境");

    applicationContext = new AnnotationConfigApplicationContext();
    applicationContext.register(ProfileConfiguration.class);
    applicationContext.getEnvironment().setActiveProfiles("prd");
    applicationContext.refresh();

    Slf4jBean prdLogBean = applicationContext.getBean(Slf4jBean.class);
    prdLogBean.info("生产环境");

    System.out.println("------------------测试@Profile结束-------------------------\n");
}


/**
 * 测试AOP
 * @param applicationContext
 */
public static void testAspect(ApplicationContext applicationContext) {
    System.out.println("\n--------------------测试AOP开始----------------------------");

    UserService userService = applicationContext.getBean(UserService.class);
    userService.welcome("ddf");
    userService.welcomeException("ddf");

    System.out.println("--------------------测试AOP结束----------------------------\n");
}
```

```
}
```

- 启动主启动类 `Application.java` ，查看控制台日志如下，可以看到, `UserService` 的 `welcome` 方法顺利执行 `@Before` 、 `@After` 、 `AfterReturning` 方法，并且接收到了返回值，但是 `welcomeException` 方法出现了异常，只执行了 `@Before` 方法和 `@After` 方法， `AfterReturning` 方法没有执行取而代之执行了 `@AfterThrowing` 方法

```
// 省略其它不相干日志
--------------------测试AOP开始----------------------------
welcome开始执行,参数列表[ddf]
welcome结束执行,参数列表[ddf]
welcome结束执行,参数列表[ddf]，返回结果: Hello ddf
welcomeException开始执行,参数列表[ddf]
welcomeException结束执行,参数列表[ddf]
welcomeException结束执行,参数列表[ddf]，出现异常: java.lang.RuntimeException: 出现异常
Exception in thread "main" java.lang.RuntimeException: 出现异常
 at com.ddf.spring.annotation.service.UserService.welcomeException(UserService.java:20)
 at com.ddf.spring.annotation.service.UserService$$FastClassBySpringCGLIB$$851f891e.invoke(<generated>)
 at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
 at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:738)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:157)
 at org.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor.invoke(MethodBeforeAdviceInterceptor.java:52)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
 at org.springframework.aop.aspectj.AspectJAfterAdvice.invoke(AspectJAfterAdvice.java:47)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
 at org.springframework.aop.framework.adapter.AfterReturningAdviceInterceptor.invoke(AfterReturningAdviceInterceptor.java:52)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
 at org.springframework.aop.aspectj.AspectJAfterThrowingAdvice.invoke(AspectJAfterThrowingAdvice.java:62)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
 at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:92)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
 at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:673)
 at com.ddf.spring.annotation.service.UserService$$EnhancerBySpringCGLIB$$7f50f9a0.welcomeException(<generated>)
 at com.ddf.spring.annotation.Application.testAspect(Application.java:206)
 at com.ddf.spring.annotation.Application.main(Application.java:41)

Process finished with exit code 1
```

2. 事务控制

本节演示如何使用注解版的事务，一般使用事务的前提为:

1. 必须有数据源连接 `DataSource`
2. 必须有事务管理器，并且将 `DataSource` 注入到事务管理器中

2.1 配置数据源与事务支持

I. 使用 `mysql` 和 `druid` 来连接数据库,在之前的 `pom.xml` 中已经导入依赖
II. 创建数据库,sql需要分开执行，先建立数据库，然后进入到数据库在执行建表语句

```
CREATE DATABASE `spring-annotation` CHARACTER SET utf8 COLLATE utf8_general_ci;

USE `spring-annotation`
DROP TABLE IF EXISTS PERSON;
CREATE TABLE PERSON(
 ID INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
 NAME VARCHAR(64),
 BIRTH_DAY DATE,
 TEL VARCHAR(32),
  ADDRESS VARCHAR(200)
);
```

III. 配置数据源
I. 在 `classpath` 下新建一个数据库连接信息的配置文件 `jdbc.properties`

```
jdbc.name=druid
jdbc.userName=root
jdbc.password=123456
jdbc.url=jdbc:mysql://localhost:3306/spring-annotation?characterEncoding=utf8&useSSL=true
jdbc.driverClassName=com.mysql.jdbc.Driver
```

II. 创建一个保存配置文件信息的类 `DataSourceConnection.java`

```java
package com.ddf.spring.annotation.bean;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/13
 */
@Component
@PropertySource("classpath:jdbc.properties")
public class DataSourceConnection {
    @Value("${jdbc.name}")
    private String name;
    @Value("${jdbc.userName}")
    private String userName;
    @Value("${jdbc.password}")
    private String password;
    @Value("${jdbc.driverClassName}")
    private String driverClassName;
    @Value("${jdbc.url}")
    private String url;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getDriverClassName() {
        return driverClassName;
    }

    public void setDriverClassName(String driverClassName) {
        this.driverClassName = driverClassName;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    @Override
    public String toString() {
        return "DataSourceConnection{" +
                "name='" + name + '\'' +
                ", userName='" + userName + '\'' +
                ", password='" + password + '\'' +
                ", driverClassName='" + driverClassName + '\'' +
```

```
                ", uri='" + uri + '\'' +
                '}';
    }
}
```

III.新建一个专门用户管理数据连接与数据相关的配置类，需要明确一点的是，配置类在同一个项目中可以存在多个，但是配置类必须被 `@CompontScan` 扫描到才可以，所以可以项目中有一个主配置类，其他的配置类只要标注 `@Configuration` 并且满足主配置类配置的 `@CompontScan` 扫描规则即可，，就会被自动识别. `@EnableTransactionManagement` 这个注解标书在配置类上，则开启了基于注解版事务的支持，否则注解无效。这个注解最好放在主配置类上，这样更能方便的看到主配置类就能明确到当前项目开启了哪些功能注解，当然这个放在自己的配置类也是可以的。

主配置类加注解 `@EnableTransactionManagement`，其余省略与前面不变

```
@Configuration
@ComponentScan(value = "com.ddf.spring.annotation", excludeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION, value = Controller.class),
        @ComponentScan.Filter(type = FilterType.CUSTOM, classes = ExcludeTypeFilter.class)
})
@Import(value = {ImportBean.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
@EnableAspectJAutoProxy
@EnableTransactionManagement
public class AnnotationConfiguration {
    //.............
}
```

新建配置类 `TransactionalConfiguration.java`，注入数据库连接 `DruidDataSource`，事务支持 `PlatformTransactionManager` 以及数据库操作简化支持 `JdbcTemplate`

```java
package com.ddf.spring.annotation.configuration;

import com.alibaba.druid.pool.DruidDataSource;
import com.ddf.spring.annotation.bean.DataSourceConnection;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;

/**
 * @author DDf on 2018/8/14
 * @Configuration 表名当时是一个配置类，如果自定义的扫描路径下包含了这个类，则该类会被自动识别成一个配置类
 * @EnableTransactionManagement 开启一个基于注解的事务,已标注在主配置类
 */
@Configuration
public class TransactionalConfiguration {
    /**
     * 向容器中注入DruidDataSource，属性来自于DataSourceConnection，
     * DataSourceConnection会自动从IOC容器中获取
     * 目前采用的main函数的写法，手动指定一个主配置类，因为不是web环境，当前配置类没有指定扫描包，而是在主配置类上指定的，
     * 所以当前类的这个方法的DataSourceConnection可能会提示没有这个bean，不用理会，如果是web环境就不会有问题了
     * @param dataSourceConnection
     * @return
     */
    @Bean
    public DruidDataSource druidDataSource(DataSourceConnection dataSourceConnection) {
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setName(dataSourceConnection.getName());
        druidDataSource.setUsername(dataSourceConnection.getUserName());
        druidDataSource.setPassword(dataSourceConnection.getPassword());
        druidDataSource.setUrl(dataSourceConnection.getUrl());
        // druidDataSource可以不指定driverClassName，会自动根据url识别
        druidDataSource.setDriverClassName(dataSourceConnection.getDriverClassName());
        return druidDataSource;
    }

    /**
     * 将数据源注入JdbcTemplate，再将JdbcTemplate注入到容器中，使用JdbcTemplate来操作数据库
     * @param druidDataSource
     * @return
     */
    @Bean
    public JdbcTemplate jdbcTemplate(DruidDataSource druidDataSource) {
        return new JdbcTemplate(druidDataSource);
    }


    /**
     * 将数据源注入PlatformTransactionManager，这是一个接口，使用DataSourceTransactionManager的实现来管理事务
     * @param druidDataSource
     * @return
     */
    @Bean
    public PlatformTransactionManager transactionManager(DruidDataSource druidDataSource) {
        return new DataSourceTransactionManager(druidDataSource);
    }
}
```

2.2 测试与总结

I. 新建一个 `Person` 类

```java
package com.ddf.spring.annotation.entity;

import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

/**
 * @author DDf on 2018/8/14
 */
public class Person implements RowMapper<Person> {
    private Integer id;
    private String name;
    private Date birthDay;
    private String address;
    private String tel;

    public Person() {
    }

    public Person(Integer id, String name, Date birthDay, String address, String tel) {
        this.id = id;
        this.name = name;
        this.birthDay = birthDay;
        this.address = address;
        this.tel = tel;
    }

    public Person(String name, Date birthDay, String address, String tel) {
        this.name = name;
        this.birthDay = birthDay;
        this.address = address;
        this.tel = tel;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthDay() {
        return birthDay;
    }

    public void setBirthDay(Date birthDay) {
        this.birthDay = birthDay;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getTel() {
        return tel;
```

```
    }

    public void setTel(String tel) {
        this.tel = tel;
    }

    @Override
    public Person mapRow(ResultSet resultSet, int i) throws SQLException {
        Person person = new Person();
        person.setId(resultSet.getInt("id"));
        person.setName(resultSet.getString("name"));
        person.setBirthDay(resultSet.getDate("birthDay"));
        person.setAddress(resultSet.getString("address"));
        person.setTel(resultSet.getString("tel"));
        return person;
    }

    @Override
    public String toString() {
        return "Person{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", birthDay=" + birthDay +
                ", address='" + address + '\'' +
                ", tel='" + tel + '\'' +
                '}';
    }
}
```

II. 新建 `PersonService`，增加添加person的方法，一个不使用事务，一个使用注解事务

```java
package com.ddf.spring.annotation.service;

import com.ddf.spring.annotation.Application;
import com.ddf.spring.annotation.entity.Person;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * @author DDf on 2018/8/14
 */
@Service
public class PersonService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 不使用事务的方法保存一个person
     * @param person
     * @return
     */
    public int add(Person person) {
        String sql = "INSERT INTO PERSON(NAME, BIRTH_DAY, TEL, ADDRESS) VALUES (?, ?, ?, ?)";
        int i = jdbcTemplate.update(sql, person.getName(), person.getBirthDay(), person.getTel(), person.getAddress());
        if (i == 1) {
            throw new RuntimeException("抛出异常");
        }
        return i;
    }


    /**
     * 使用事务控制来保存一个person，在保存结束后抛出异常，看是否会回滚
     * @param person
     * @return
     */
    @Transactional(rollbackFor = Exception.class)
    public int addWithTransactional(Person person) {
        String sql = "INSERT INTO PERSON(NAME, BIRTH_DAY, TEL, ADDRESS) VALUES (?, ?, ?, ?)";
        int i = jdbcTemplate.update(sql, person.getName(), person.getBirthDay(), person.getTel(), person.getAddress());
        if (i == 1) {
            throw new RuntimeException("抛出异常");
        }
        return i;
    }
}
```

III. 修改 `Application.java`，测试数据库连接是否正常以及测试person添加

```
/**
 * 测试连接数据源
 */
public static void testDruidDataSource(ApplicationContext applicationContext) {
    System.out.println("\n--------------------获取DruidDataSource数据库连接开始---------------------------");
    DataSourceConnection dataSourceConnection = applicationContext.getBean(DataSourceConnection.class);
    System.out.println(dataSourceConnection);
    DruidDataSource druidDataSource = applicationContext.getBean(DruidDataSource.class);
    DruidPooledConnection connection = null;
    try {
        connection = druidDataSource.getConnection();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    System.out.println(connection);
    System.out.println("--------------------获取DruidDataSource数据库连接结束---------------------------\n");
}


/**
 * 测试增加一个Person
 * 使用事务和不使用事务对比
 * @param applicationContext
 */
public static void testAddPerson(ApplicationContext applicationContext) {
    System.out.println("\n--------------------测试增加一个Person开始-------------------------");
    PersonService personService = applicationContext.getBean(PersonService.class);
    Calendar calendar = new GregorianCalendar();
    calendar.set(1992, 4, 29);
    personService.add(new Person("no_transactional", calendar.getTime(), "上海市", "18356789999"));

    personService.addWithTransactional(new Person("with_transactional", calendar.getTime(), "上海市", "18356789999"));

    System.out.println("--------------------测试增加一个Person结束-------------------------\n");
}
```

完整版如下

```java
package com.ddf.spring.annotation;

import com.alibaba.druid.pool.DruidDataSource;
import com.alibaba.druid.pool.DruidPooledConnection;
import com.ddf.spring.annotation.bean.*;
import com.ddf.spring.annotation.configuration.AnnotationConfiguration;
import com.ddf.spring.annotation.configuration.ProfileConfiguration;
import com.ddf.spring.annotation.entity.Person;
import com.ddf.spring.annotation.entity.User;
import com.ddf.spring.annotation.service.*;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.sql.SQLException;
import java.time.LocalDate;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

/**
 * @author DDf on 2018/7/19
 */
public class Application {
    public static void main(String[] args) {
        System.out.println("-----------------------IOC容器初始化-------------------------");
        // 创建一个基于配置类启动的IOC容器，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别，如果主配置类扫描包的路径下包含其他配置类，则其他配置类可以被自动识别
        AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(AnnotationConfiguration.class);
        System.out.println("-----------------------IOC容器初始化完成-------------------------\n");
        // 获取当前IOC中所有bean的名称,即使是懒加载类型的bean也会获取到
        printBeans(applicationContext);
        // 测试@Scope bean的作用域
        testPrototypeScopeService(applicationContext);
        // 测试单实例bean的@Lazy懒加载
        testLazyBeanService(applicationContext);
        // 测试FactoryBean接口导入单实例与Prototype作用域的组件
        testFactoryBeanPrototypeBean(applicationContext);
        // 测试@PropertySource和@Value属性赋值
        testPropertySourceValue(applicationContext);
        // 测试@Autowired注入多个相同类型的Bean
        testAutowired(applicationContext);
        // 测试@Profile根据环境注入Bean
        testProfile(applicationContext);
        // 测试AOP
        testAspect(applicationContext);
        // 测试获取数据库连接
        testDruidDataSource(applicationContext);
        // 测试事务
        testAddPerson(applicationContext);

        // 销毁容器
        applicationContext.close();
    }


    /**
     * 打印当前IOC容器中所有预定义的Bean
     * @param applicationContext
     */
    private static void printBeans(AnnotationConfigApplicationContext applicationContext) {
        String[] definitionNames = applicationContext.getBeanDefinitionNames();
        // 打印当前IOC中对应名称的bean和bean的类型
        for (String name : definitionNames) {
            // 这个会影响到测试懒加载的效果，如果需要测试懒加载，这行代码需要注释掉，因为getBean方法一旦调用则会初始化
            Object bean = applicationContext.getBean(name);
            System.out.println("bean name:" + name + ", type: " + bean.getClass());
        }
    }
```

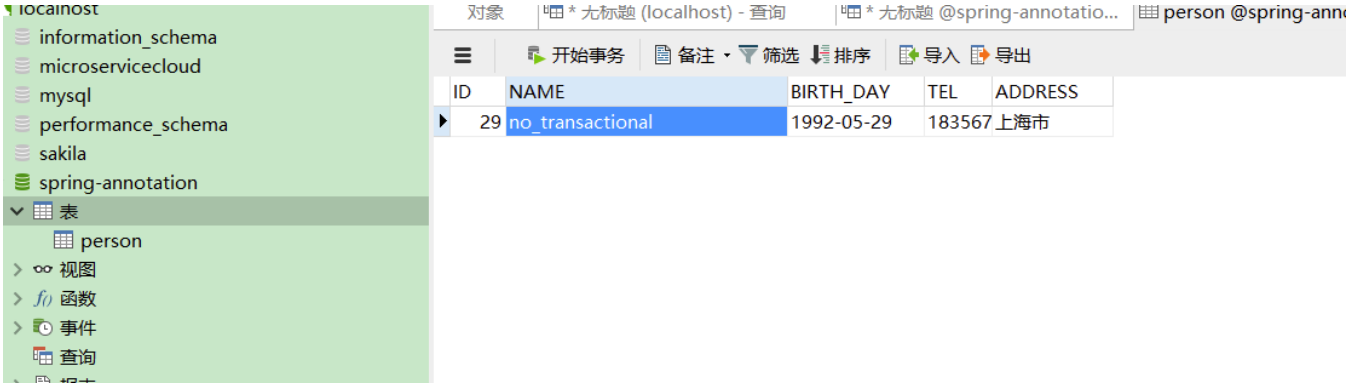
```
    /**
     * 测试@Scope bean的作用域
     *
     * @param applicationContext
     */
    public static void testPrototypeScopeService(ApplicationContext applicationContext) {
        System.out.println("\n-----------------------测试@Scope开始-------------------------");
        UserService userService = (UserService) applicationContext.getBean("userService");
        UserService userService1 = applicationContext.getBean(UserService.class);
        System.out.println("默认单实例bean UserService是否相等 " + (userService == userService1));

        PrototypeScopeService prototypeScopeService = applicationContext.getBean(PrototypeScopeService.class);
        PrototypeScopeService prototypeScopeService1 = applicationContext.getBean(PrototypeScopeService.class);
        System.out.println("PrototypeScopeService prototype scope作用域是否相等: " + (prototypeScopeService ==
prototypeScopeService1));
        System.out.println("-----------------------测试@Scope结束-------------------------\n");
    }

    /**
     * 测试单实例bean的懒加载，只有等使用的时候再创建实例。
     * IOC容器启动后不会创建该bean的实例，如果是在该方法中才创建这个bean的实例，并且获得的两个bean是同一个的话，则测试通过。
     */
    public static void testLazyBeanService(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试单实例bean的@Lazy懒加载开始----------------------");
        LazyBeanService lazyBeanService = applicationContext.getBean(LazyBeanService.class);
        LazyBeanService lazyBeanService1 = applicationContext.getBean(LazyBeanService.class);
        System.out.println("lazyBeanService==lazyBeanService1？: " + (lazyBeanService == lazyBeanService1));
        System.out.println("---------------测试单实例bean的@Lazy懒加载结束----------------------\n");
    }


    /**
     * 测试通过FactoryBean接口导入单实例与Prototype作用域的组件,根据打印可以看出FactoryBean创建的单实例Bean都是懒加载的
     * @param applicationContext
     */
    public static void testFactoryBeanPrototypeBean(ApplicationContext applicationContext) {
        System.out.println("\n----------测试通过FactoryBean注册单实例和Prototype作用域的组件开始----------");
        FactorySingletonBean factorySingletonBean = applicationContext.getBean(FactorySingletonBean.class);
        FactorySingletonBean factorySingletonBean1 = applicationContext.getBean(FactorySingletonBean.class);

        FactoryPrototypeBean factoryPrototypeBean = applicationContext.getBean(FactoryPrototypeBean.class);
        FactoryPrototypeBean factoryPrototypeBean1 = applicationContext.getBean(FactoryPrototypeBean.class);

        System.out.println("单实例factorySingletonBean==factorySingletonBean1?" + (factorySingletonBean==factorySingletonBean1));

        System.out.println("Prototype作用域factoryPrototypeBean==factoryPrototypeBean1?" +
(factoryPrototypeBean==factoryPrototypeBean1));
        System.out.println("----------测试通过FactoryBean注册单实例和Prototype作用域的组件结束----------\n");
    }

    /**
     * 测试通过@PropertySource和@Value注解来对属性进行赋值
     * @param applicationContext
     */
    public static void testPropertySourceValue(ApplicationContext applicationContext) {
        System.out.println("\n---------------测试@PropertySource和@Value赋值开始----------------");
        User user = applicationContext.getBean(User.class);
        System.out.println("user属性为:  " + user.toString());
        System.out.println("---------------测试@PropertySource和@Value赋值结束----------------\n");

    }

    /**
     * 测试在IOC容器中存在两个相同类型的Bean,但是Bean的名称不一致
     * 在这种情况下，使用@Autowired将该Bean注入到另外一个容器中
     * @Autowired 默认使用Bean的类型去匹配注入，如果找到多个相同类型的Bean,则使用默认名称去获取Bean,Bean的默认名称为类首字母缩写
     * 也可以配合@Autowired配合@Qualifier注解明确指定注入哪个Bean，还可以在注入Bean的地方使用@Primary，在不指定@Qualifier的情况下，
     * 默认注入哪个Bean {@link AutowiredService}
     * @param applicationContext
     */
```

```java
public static void testAutowired(ApplicationContext applicationContext) {
    System.out.println("\n--------------测试autowired注入多个相同类型的类开始-----------------");
    AutowiredBean autowiredBean = (AutowiredBean) applicationContext.getBean("autowiredBean");
    AutowiredBean autowiredBean2 = (AutowiredBean) applicationContext.getBean("autowiredBean2");
    System.out.println("autowiredBean: " + autowiredBean);
    System.out.println("autowiredBean2: " + autowiredBean2);
    System.out.println(autowiredBean == autowiredBean2);

    /**
     * 这里已做更改，修改了默认注入 {@link com.ddf.spring.annotation.configuration.AnnotationConfiguration.autowiredBean2}
     */
    AutowiredService autowiredService = applicationContext.getBean(AutowiredService.class);
    AutowiredBean autowiredServiceBean = autowiredService.getAutowiredBean();
    System.out.println("使用@Primay后AutowiredService默认注入bean: " + autowiredServiceBean);

    AutowiredBean autowiredServiceBean2 = autowiredService.getQualifierAutowiredBean();
    System.out.println("使用@Qualifier明确注入Bean: " + autowiredServiceBean2);

    // 使用@Resource注入
    AutowiredBean resourceAutowiredBean = autowiredService.getResourceAutowiredBean();
    System.out.println("使用@Resource注入autowiredBean: " + resourceAutowiredBean);
    AutowiredBean resourceAutowiredBean2 = autowiredService.getResourceAutowiredBean2();
    System.out.println("使用@Resource注入autowiredBean2: " + resourceAutowiredBean2);

    // 使用@Inject注入
    UserService userService = autowiredService.getUserService();
    System.out.println("使用@Inject注入UserService:  " + userService);

    System.out.println("--------------测试autowired注入多个相同类型的类结束-----------------\n");
}


/**
 * 测试根据激活的Profile来根据环境注册不同的Bean
 * 切换profile有两种方式：
 * 1. 在java虚拟机启动参数加 -Dspring.profiles.active=test
 * 2. 如下演示，使用代码切换，可以将切换的变量放在配置文件，spring-boot配置文件即是这种方式
 * @param applicationContext
 */
public static void testProfile(AnnotationConfigApplicationContext applicationContext) {
    System.out.println("\n------------------测试@Profile开始-------------------------");
    // 重新新建一个IOC容器
    applicationContext = new AnnotationConfigApplicationContext();
    // 注册配置类
    applicationContext.register(ProfileConfiguration.class);
    // 设置当前激活的环境profile
    applicationContext.getEnvironment().setActiveProfiles("dev");
    // 刷新容器
    applicationContext.refresh();

    // 使用接口获得实际接口实现类的注入Bean
    Slf4jBean devLogBean = applicationContext.getBean(Slf4jBean.class);
    devLogBean.info("测试环境");

    applicationContext = new AnnotationConfigApplicationContext();
    applicationContext.register(ProfileConfiguration.class);
    applicationContext.getEnvironment().setActiveProfiles("prd");
    applicationContext.refresh();

    Slf4jBean prdLogBean = applicationContext.getBean(Slf4jBean.class);
    prdLogBean.info("生产环境");

    System.out.println("------------------测试@Profile结束-------------------------\n");
}


/**
 * 测试AOP
 * @param applicationContext
 */
public static void testAspect(ApplicationContext applicationContext) {
```

```
        System.out.println("\n--------------------测试AOP开始----------------------------");

        UserService userService = applicationContext.getBean(UserService.class);
        userService.welcome("ddf");
        try {
            userService.welcomeException("ddf");
        } catch (Exception e) {
        }

        System.out.println("--------------------测试AOP结束----------------------------\n");
    }


    /**
     * 测试连接数据源
     */
    public static void testDruidDataSource(ApplicationContext applicationContext) {
        System.out.println("\n--------------------获取DruidDataSource数据库连接开始----------------------------");
        DataSourceConnection dataSourceConnection = applicationContext.getBean(DataSourceConnection.class);
        System.out.println(dataSourceConnection);
        DruidDataSource druidDataSource = applicationContext.getBean(DruidDataSource.class);
        DruidPooledConnection connection = null;
        try {
            connection = druidDataSource.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println(connection);
        System.out.println("--------------------获取DruidDataSource数据库连接结束----------------------------\n");
    }


    /**
     * 测试增加一个Person
     * 使用事务和不使用事务对比
     * @param applicationContext
     */
    public static void testAddPerson(ApplicationContext applicationContext) {
        System.out.println("\n--------------------测试增加一个Person开始-------------------------");
        PersonService personService = applicationContext.getBean(PersonService.class);
        Calendar calendar = new GregorianCalendar();
        calendar.set(1992, 4, 29);
        try {
            personService.add(new Person("no_transactional", calendar.getTime(), "上海市", "18356789999"));
        } catch (Exception e) {}
        try {
            personService.addWithTransactional(new Person("with_transactional", calendar.getTime(), "上海市", "18356789999"));
        } catch (Exception e) {}
        System.out.println("--------------------测试增加一个Person结束-------------------------\n");
    }
}
```

IIII. 运行主启动类，可以看到日志如下，省略不相关其它日志,数据库连接正常连接，然后测试保存的方法里每一个都抛出了异常，但是查看数据库记录，只有第一个用户 `no_transactional` 保存了下来，另外一条记录回滚并没有保存下来

```
-------------------获取DruidDataSource数据库连接开始---------------------------
DataSourceConnection{name='druid', userName='root', password='123456', driverClassName='com.mysql.jdbc.Driver',
url='jdbc:mysql://localhost:3306/spring-annotation?characterEncoding=utf8&useSSL=true'}
八月 14, 2018 1:09:14 下午 com.alibaba.druid.pool.DruidDataSource info
信息: {dataSource-1,druid} inited
com.mysql.jdbc.JDBC4Connection@36cda2c2
-------------------获取DruidDataSource数据库连接结束---------------------------


-------------------测试增加一个Person开始------------------------
add开始执行,参数列表[Person{id=null, name='no_transactional', birthDay=Fri May 29 13:09:14 CST 1992, address='上海市',
tel='18356789999'}]
add结束执行,参数列表[Person{id=null, name='no_transactional', birthDay=Fri May 29 13:09:14 CST 1992, address='上海市',
tel='18356789999'}]
add结束执行,参数列表[Person{id=null, name='no_transactional', birthDay=Fri May 29 13:09:14 CST 1992, address='上海市',
tel='18356789999'}], 出现异常: java.lang.RuntimeException: 抛出异常
addWithTransactional开始执行,参数列表[Person{id=null, name='with_transactional', birthDay=Fri May 29 13:09:14 CST 1992,
address='上海市', tel='18356789999'}]
addWithTransactional结束执行,参数列表[Person{id=null, name='with_transactional', birthDay=Fri May 29 13:09:14 CST 1992,
address='上海市', tel='18356789999'}]
addWithTransactional结束执行,参数列表[Person{id=null, name='with_transactional', birthDay=Fri May 29 13:09:14 CST 1992,
address='上海市', tel='18356789999'}], 出现异常: java.lang.RuntimeException: 抛出异常
-------------------测试增加一个Person结束------------------------
```



三、扩展原理

1. `BeanFactoryPostProcessor` 接口

> BeanFactoryPostProcessor
>
> > PropertyResourceConfigurer // 配置文件方式覆盖bean属性
> > BeanDefinitionRegistryPostProcessor // 注册更多的bean
>
> BeanPostProcessor 直接对属性改变，建议使用这个接口

```java
/*
 * Copyright 2002-2012 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

/**
 * Allows for custom modification of an application context's bean definitions,
 * adapting the bean property values of the context's underlying bean factory.
 *
 * <p>Application contexts can auto-detect BeanFactoryPostProcessor beans in
 * their bean definitions and apply them before any other beans get created.
 *
 * <p>Useful for custom config files targeted at system administrators that
 * override bean properties configured in the application context.
 *
 * <p>See PropertyResourceConfigurer and its concrete implementations
 * for out-of-the-box solutions that address such configuration needs.
 *
 * <p>A BeanFactoryPostProcessor may interact with and modify bean
 * definitions, but never bean instances. Doing so may cause premature bean
 * instantiation, violating the container and causing unintended side-effects.
 * If bean instance interaction is required, consider implementing
 * {@link BeanPostProcessor} instead.
 *
 * @author Juergen Hoeller
 * @since 06.07.2003
 * @see BeanPostProcessor
 * @see PropertyResourceConfigurer
 */
public interface BeanFactoryPostProcessor {

 /**
  * Modify the application context's internal bean factory after its standard
  * initialization. All bean definitions will have been loaded, but no beans
  * will have been instantiated yet. This allows for overriding or adding
  * properties even to eager-initializing beans.
  * @param beanFactory the bean factory used by the application context
  * @throws org.springframework.beans.BeansException in case of errors
  */
 void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;

}
```

I. 可以获取到所有的 `Bean` 预定义，但绝不是初始化，说明这个时候的 `Bean` 只是一些定义信息，可以获取到 `beanFactory` ，而beanFactory就是根据预定义的 `Bean` 信息去进行 `Bean` 的初始化，所以这里允许修改 `Bean` 的预定义的信息，以达到在 `bean` 被初始化之前做一些属性的改变等.如果是希望对 `bean` 的属性进行改变,推荐使用接口 `BeanPostProcessor` 代替

II. `org.springframework.beans.factory.config.PropertyResourceConfigurer` 是 `BeanFactoryPostProcessor` 的子类接口，允许从属性文件的方式来配置 `bean` 的值，属性值可以在读取后通过覆盖进行转换,见方法 `convertPropertyValue` , 该接口提供了两个实现

- PropertyOverrideConfigurer ,for "beanName.property=value" style overriding
- PropertyPlaceholderConfigurer, for replacing "${...}" placeholders

III. `org.springframework.beans.factory.support.BeanDefinitionRegistryPostProcessor`，这个接口扩展自接口`BeanFactoryPostProcessor`,这个接口不仅可以获取已经预定义的`bean`信息，更多的用于可以在`bean`被实例化之前注册更多的`bean`定义，然后反向注册到`beanFactory`中，这样就可以实例化更多的`Bean`

五、注解版整合Spring+SpringMVC

1. 可行性与整合思路研究

1.1 ServletContainerInitializer与SpringServletContainerInitializer

依赖于Servlet3.0的`ServletContainerInitializer`特性，在`Servlet3.0`章节中，已经说过`ServletContainerInitializer`类型的类放在`\META-INF\services\javax.servlet.ServletContainerInitializer`这个文件下，把实现了`ServletContainerInitializer`类的全名写入文件内容里。会在容器启动的时候就调用文件里的类的`onStartup()`方法。我们引入了`Spring-web.jar`包之后，打开该jar包下的源码会这个路径的文件下会发现里面的内容为`org.springframework.web.SpringServletContainerInitializer`，打开这个类的源码会发现这个类就是实现了`ServletContainerInitializer`。而且该类通过注解`@HandlesTypes(WebApplicationInitializer.class)`将所有`WebApplicationInitializer`的所有子类都导入到了onStartup的参数`Set`中,`onStarupup()`的方法核心内容为遍历所有传入的`WebApplicationInitializer`类型的类，然后判断将不是接口和抽象类的实现放入到一个`List`中，加入完成之后，再进行排序，然后再遍历这个`List`获取所有的`WebApplicationInitializer`回调自己的`onStartup`方法。相关源码如下

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {
 @Override
 public void onStartup(Set<Class<?>> webAppInitializerClasses, ServletContext servletContext)
   throws ServletException {

  List<WebApplicationInitializer> initializers = new LinkedList<WebApplicationInitializer>();

  if (webAppInitializerClasses != null) {
   for (Class<?> waiClass : webAppInitializerClasses) {
    // Be defensive: Some servlet containers provide us with invalid classes,
    // no matter what @HandlesTypes says...
    if (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&
      WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
     try {
      initializers.add((WebApplicationInitializer) waiClass.newInstance());
     }
     catch (Throwable ex) {
      throw new ServletException("Failed to instantiate WebApplicationInitializer class", ex);
     }
    }
   }
  }

  if (initializers.isEmpty()) {
   servletContext.log("No Spring WebApplicationInitializer types detected on classpath");
   return;
  }

  servletContext.log(initializers.size() + " Spring WebApplicationInitializers detected on classpath");
  AnnotationAwareOrderComparator.sort(initializers);
  for (WebApplicationInitializer initializer : initializers) {
   initializer.onStartup(servletContext);
  }
 }
}
```

1.2 WebApplicationInitializer

现在来看一下导入的`WebApplicationInitializer`，可以看到这就是一个接口，定义了一个方法`onStartup()`方法。该类的接口与`ServletContainerInitializer`本身类似，只不过该接口并没有实现`ServletContainerInitializer`这个接口，所以并不具备容器一旦初始化就被调用的功能。而`Spring`通过很巧妙的方式使用`SpringServletContainerInitializer`这个类来作为初始化类，并将所有实现了`WebApplicationInitializer`接口的类传递进来，然后调用`WebApplicationInitializer`类的`onStartup()`方法。这样对于使用者来说，只需要实现`WebApplicationInitializer`接口就可以，如果我们需要有多个初始化配置类，只需要用多个类去实现`WebApplicationInitializer`接口即可，而不需要复杂的在路径文件内容里，写入自定义实现的类全路径。

```
public interface WebApplicationInitializer {=
 void onStartup(ServletContext servletContext) throws ServletException;
}
```

1.3 WebApplicationInitializer的默认抽象实现

三个默认抽象实现

> WebApplicationInitializer
>
> > AbstractContextLoaderInitializer
> >
> > > AbstractDispatcherServletInitializer
> > >
> > > > AbstractAnnotationConfigDispatcherServletInitializer

1.3.1 AbstractContextLoaderInitializer

`AbstractContextLoaderInitializer` 定义了 `onStartup()` 调用 `registerContextLoaderListener()` 。然后调用 `createRootApplicationContext()` ，添加Spring根容器的 `Listener` ，`createRootApplicationContext()` 这是一个抽象方法，因此是留给我们实现的

1.3.2 AbstractDispatcherServletInitializer

`AbstractDispatcherServletInitializer` 继承了 `AbstractContextLoaderInitializer` ，在完成 `AbstractContextLoaderInitializer` 的步骤后，然后调用 `registerDispatcherServlet()` 方法来 `new` 一个 `SpringMVC` 的前端控制器 `DispatcherServlet` ,然后调用方法 `getServletApplicationContextInitializers()` 来创建 `SpringMVC` 的 `WEB` 容器，这个方法返回 `Null` 。 `DispatcherServlet` 在这时还只是一个普通的类，在往下开始通过代码 `ServletRegistration.Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);` 来将 `DispatcherServlet` 注册成一个 `Servlet` .名称默认为 `dispatcher` ，而该 `Servlet` 的 `mapping` 则通过方法 `getServletMappings()` 获取，
这个方法又是一个抽象方法，我们可以重写父类的这个方法来自定义映射规则；
因此可以结合 `AbstractContextLoaderInitializer` ，在这里只添加 `Spring` 的核心容器，如 `service, dataSource` ，而在 `AbstractDispatcherServletInitializer` 里添加 `controllers, viewResolver, HandlerMapping` 等和 `web` 相关的 `bean`

1.3.3 AbstractAnnotationConfigDispatcherServletInitializer

`AbstractAnnotationConfigDispatcherServletInitializer` 继承 `AbstractDispatcherServletInitializer` ，它重写了父类的 `createRootApplicationContext()` 和 `createServletApplicationContext()` 方法，这两个方法内部分别通过获取配置类来创建Spring的
根容器和Web容器，创建根容器的配置类通过 `getRootConfigClasses()` 这个方法获取，创建web容器的配置类通过 `getServletConfigClasses()`
方法获取，因此又留给我们自定义

1.4 思路总结

综上所述，如果我们要基于配置类的方法来自定义创建 `Spring` 和 `SpringWebmvc` 的容器，则需要继承 `AbstractAnnotationConfigDispatcherServletInitializer`
这个类即可，而通过重写 `getRootConfigClasses()` 这个方法来指定IOC容器的配置类， `getServletConfigClasses()` 这个方法来指定WEB容器的配置类，然后复写 `getServletMappings()` 方法指定 `DispatcherServlet` 的映射规则

2. 项目整合

使用 `maven` 创建 `web` 工程

2.1 `pom.xml` 内容如下

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ddf.springmvc</groupId>
    <artifactId>springmvc-annotation</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>4.3.12.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <!-- https://mvnrepository.com/artifact/javax.inject/javax.inject -->
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.44</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.10</version>
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.9.5</version>
        </dependency>

        <dependency>
```

```
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.9.5</version>
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.9.5</version>
        </dependency>
    </dependencies>
</project>
```

2.2 IOC 容器配置

注意： 一般配置类需要生效，需要加入注解 `@Configuration` ，但当前这个类，后面会通过编码来作为主配置类，所以不需要这个注解

新建 `RootApplicationContextConfig.java` ,用来扫描除 `@Controller` 以外的注解

```
package com.ddf.springmvc.annotation.configuration.ioc;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.stereotype.Controller;
/**
 * @author DDf on 2018/8/16
 * 指定Spring根容器的配置类，排除web相关的bean
 *
 */
@ComponentScan(value = "com.ddf.springmvc.annotation",
        excludeFilters = {@ComponentScan.Filter(value = Controller.class)})
public class RootApplicationContextConfig {

    public RootApplicationContextConfig() {
        System.out.println("RootApplicationContextConfig配置类被读取。。。。。。。。。。。。。。");
    }
}
```

2.3 WEB 容器配置类

NVC CONFIG

注意： 一般配置类需要生效，需要加入注解 `@Configuration` ，但当前这个类，后面会通过编码来作为主配置类，所以不需要这个注解

创建web容器配置类，只扫描controller,并且配置与web相关的bean.

而web功能一般需要做定制才能使用，如视图解析器，消息转换器，静态资源映射，默认静态资源处理，拦截器.

这里扩展一下，如果需要定制 `MVC` 的功能，可以继承类 `WebMvcConfigurerAdapter` ,并且在该类上使用注解 `@EnableWebMvc` ，则可以完全接管与定义 `SpringMVC` 。

这个注解，再做一个延伸注意，在使用 `SpringBoot` 作为框架的项目中，如果要注册 `web` 组件，只需要继承 `WebMvcConfigurerAdapter` 该类，并将该类通过 `@Configuration` 注解标识为配置类即可，

千万不要轻易加注解 `@EnableWebMvc` 。这个注解是完全接管MVC的定制，而 `SpringBoot` 默认做的定制则会完全失效。当然如果需要完全定制的话，则还是需要使用这个注解，要分清楚；

注意

因为如下配置牵扯到了添加拦截器，所以要提前把拦截器相关的代码先贴出来

- 新建 `RequestContext.java` ,目的是把所有的请求参数封装到这个类中

```java
package com.ddf.springmvc.annotation.configuration.util;

import org.springframework.stereotype.Component;
import org.springframework.web.context.annotation.RequestScope;

import java.util.HashMap;
import java.util.Map;

/**
 * @author DDf on 2018/8/17
 * 将当前请求的参数放在RequestContext这个类中
 */
@Component
@RequestScope
public class RequestContext {

    private Map<String, Object> paramsMap = new HashMap<>();

    public Map<String, Object> getParamsMap() {
        return paramsMap;
    }
}
```

- 新建拦截器代码 `RequestContextInterceptor.java`

```java
package com.ddf.springmvc.annotation.configuration.interceptor;

import com.ddf.springmvc.annotation.configuration.util.RequestContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Enumeration;

/**
 * @author DDf on 2018/8/17
 * 拦截器，将所有当前请求的参数封装到RequestContext中
 */
@Component
public class RequestContextInterceptor implements HandlerInterceptor {
    @Autowired
    private RequestContext requestContext;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        Enumeration<String> parameterNames = request.getParameterNames();
        while (parameterNames.hasMoreElements()) {
            String element = parameterNames.nextElement();
            requestContext.getParamsMap().put(element, request.getParameter(element));
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)
throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws
Exception {

    }
}
```

- 新建配置类 `ServletApplicatioonContextConfig.java`

```java
package com.ddf.springmvc.annotation.configuration.web;

import com.ddf.springmvc.annotation.configuration.interceptor.RequestContextInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.stereotype.Controller;
import org.springframework.web.servlet.config.annotation.*;

import java.util.List;

/**
 * @author DDf on 2018/8/16
 * 创建web容器配置类，只扫描controller,并且配置与web相关的bean，
 * controllers，viewResolver，HandlerMapping等
 * useDefaultFilters默认为true，包含了@Componet，@Service，@Repository，@Controller等，
 * 因此需要设置为false，不需要扫描那么多类型
 *
 * 所有与SpringMVC相关的配置请参考
 * https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config
 */
@ComponentScan(value = "com.ddf.springmvc.annotation", includeFilters = {
        @ComponentScan.Filter(value = Controller.class)
}, useDefaultFilters = false)
@EnableWebMvc
public class ServletApplicatioonContextConfig extends WebMvcConfigurerAdapter {

    @Autowired
    private RequestContextInterceptor requestContextInterceptor;

    public ServletApplicatioonContextConfig() {
        System.out.println("ServletApplicatioonContextConfig配置类被读取。。。。。。。。。。。。。");
    }


    /**
     * 为了支持转json，则需要导入jackson的包，否则返回对象时会提示找不到转换器，而不能成功转换为json。
     * 导入jsckson包就可以，未找到源码在哪里处理的这一块，导包就可以了
     * @param converters
     */
    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        super.configureMessageConverters(converters);
    }

    /**
     * Add a resource handler for serving static resources based on the specified URL path
     * * patterns. The handler will be invoked for every incoming request that matches to
     * * one of the specified path patterns.
     * 因为在web环境中最终所有的请求都会被dispatcherServlet拦截，然后由于配置了DefaultServletHandling,所以该映射如果
     * dispatcher处理不了，就会交由默认的servlet当做静态资源来处理。但是通过 addResourceHandlers可以直接添加静态资源的映射，
     * 即前端发送的请求如果与静态资源映射请求相匹配，那么就可以明确本次是请求静态资源，然后再该映射配置的静态资源地址直接获得资源
     *
     * @param registry
     */
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        System.out.println("addResourceHandlers.............");
        // 所有/static/** 相匹配的请求都到addResourceLocations指定的路径下获得资源
        // 当前项目请求/static/index.html则会找到/resource/static/index.html这个文件
        registry.addResourceHandler("/static/**").addResourceLocations("/static/",
                "classpath:static/");
    }

    /**
     * 通过调用new CorsRegistration(pathPattern);该类的config默认在实例化的如果没有配置允许的http方法，则会是所有的http方法，
     * 没有找到方法，如何自定义添加指定允许的http方法
     * @param registry
     */
    @Override
```

```
    public void addCorsMappings(CorsRegistry registry) {
        System.out.println("addCorsMappings.................");
        // 这种默认是允许指定请求的所有的http方法
        registry.addMapping("/**");
    }

    /**
     *配置不经过handler的视图解析器，根据映射地址直接找指定视图，如下配置根据configureViewResolvers默认的视图解析的前缀和后缀，
     * 则配置了项目默认的初始化界面为/views/index.html
     * @param registry
     */
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        System.out.println("addViewControllers...............");
        registry.addViewController("/").setViewName("index");
    }


    /**
     * 配置SpringMVC的视图解析器
     * <bean id="viewResolver"
     * class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     * <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
     * <property name="prefix" value="/views/" />
     * <property name="suffix" value=".html" />
     * </bean>
     * @param registry
     */
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        System.out.println("configureViewResolvers...................");
        registry.jsp("/views/", ".html");
    }


    /**
     * 开启静态资源的访问，否则所有的静态页面都不能访问
     * 即当静态资源的请求被SpringMVC拦截后并没有找到对应的映射，这时候应该将请求交给默认的Servlet去处理页面
     * 类似于原配置文件中 的
     * <mvc:default-servlet-handler/>
     * @param configurer
     */
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        System.out.println("configureDefaultServletHandling.....................");
        configurer.enable();
    }


    /**
     * 注册拦截器列表
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        System.out.println("addInterceptors...................");
        registry.addInterceptor(requestContextInterceptor).addPathPatterns("/**");
        super.addInterceptors(registry);
    }
}
```

2.4 实现 `AbstractAnnotationConfigDispatcherServletInitializer`

基于注解版容器配置

在前面两个章节，已经对 `IOC` 和 `web` 的容器做了配置，现在需要的就是对容器进行应用这些配置

新建 `MyWebAppInitializer.java`

```java
package com.ddf.springmvc.annotation.configuration;

import com.ddf.springmvc.annotation.configuration.ioc.RootApplicationContextConfig;
import com.ddf.springmvc.annotation.configuration.web.ServletApplicatioonContextConfig;
import org.springframework.web.SpringServletContainerInitializer;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

import javax.servlet.ServletContainerInitializer;
import javax.servlet.ServletContext;
import java.util.Set;

/**
 * @author DDf on 2018/8/16
 *
 * 基于注解版的SpringMVC的容器的配置，请参考官方文档
 * https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-servlet-config
 *
 * 请参考org.springframework:spring-web的jar包下在META-INF/services下有一个文件叫javax.servlet.ServletContainerInitializer，里面配置的一个类
 * {@link SpringServletContainerInitializer}。这个类实现了{@link ServletContainerInitializer},所以同样具备
 * 容器一启动就会调用该类的{@link SpringServletContainerInitializer#onStartup(Set, ServletContext)}方法（Servlet3.0+机制）
 * 同时该类通过注解{@link javax.servlet.annotation.HandlesTypes}将所有实现{@link WebApplicationInitializer}
 * 接口的类都传入到onStartup()方法的第一个参数Set中,遍历所有的WebApplicationInitializer，把所有不是接口和子类的WebApplicationInitializer的添加到列表中，
 * 然后排序之后重新回调列表中的每个webApplicationInitializer的onStartup()方法
 *
 * webApplicationInitializer接口有三个子抽象类实现
 * 1. AbstractContextLoaderInitializer定义了启动之后创建registerContextLoaderListener。然后调用createRootApplicationContext()
 * 添加Spring根容器的Listener,createRootApplicationContext()这是一个抽象方法，因此是留给我们实现的
 *
 * 2. AbstractDispatcherServletInitializer继承了AbstractContextLoaderInitializer，在完成1的步骤后，然后调用
 * registerDispatcherServlet()方法来new一个SpringMVC的前端控制器DispatcherServlet,然后调用方法getServletApplicationContextInitializers
 * 来创建SpringMVC的WEB容器，这个方法返回Null，。DispatcherServlet在这时还只是一个普通的类，在往下开始通过代码
 * ServletRegistration.Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);
 * 来将DispatcherServlet注册成一个Servlet.名称默认为dispatcher，而该Servlet的mapping则通过方法getServletMappings()获取，
 * 这个方法又是一个抽象方法，我们可以重写父类的这个方法来自定义映射规则；
 * 因此可以结合1，在1里只添加Spring IOC的核心容器，如service，dataSource，而
 * 在2里添加controllers，viewResolver，HandlerMapping等和web相关的bean
 *
 * 3. AbstractAnnotationConfigDispatcherServletInitializer继承AbstractDispatcherServletInitializer，它重写了父类的
 * createRootApplicationContext()和createServletApplicationContext()方法，这两个方法内部分别通过获取配置类来创建Spring的
 * 根容器和Web容器，创建根容器的配置类通过getRootConfigClasses()这个方法获取，创建web容器的配置类通过getServletConfigClasses()
 * 方法获取，因此又留给我们自定义
 *
 *
 * ============================================================
 * 综上所述，如果我们要基于配置类的方法来自定义创建Spring和SpringWebmvc的容器，则需要继承AbstractAnnotationConfigDispatcherServletInitializer
 * 这个类即可，而通过重写getRootConfigClasses()这个方法来指定IOC容器的配置类，getServletConfigClasses()这个方法
 * 来指定WEB容器的配置类，然后复写getServletMappings()方法指定DispatcherServlet的映射规则
 *
 */
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    /**
     * 指定Spring根容器的配置类
     * @return
     */
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] {RootApplicationContextConfig.class};
    }

    /**
     * 指定WEB容器的配置类
     * @return
     */
    @Override
```

```
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {ServletApplicatioonContextConfig.class};
    }

    /**
     * 配置SpringMVC的DispatcherServlet的映射规则
     * / 拦截所有请求，包括静态资源，但不包括JSP
     * /* 拦截所有资源，包含静态资源，包含JSP，一般JSP无法显示就是因为这里配置出错
     * @return
     */
    @Override
    protected String[] getServletMappings() {
        System.out.println(getServletName() + "映射规则为: " + "/");
        return new String[] {"/"};
    }
}
```

2.5 基本测试

在完成了上述步骤之后，完成了对容器的基本定制，配置类是否能正常加载，需要测试

1. 新建 `HelloService.java`

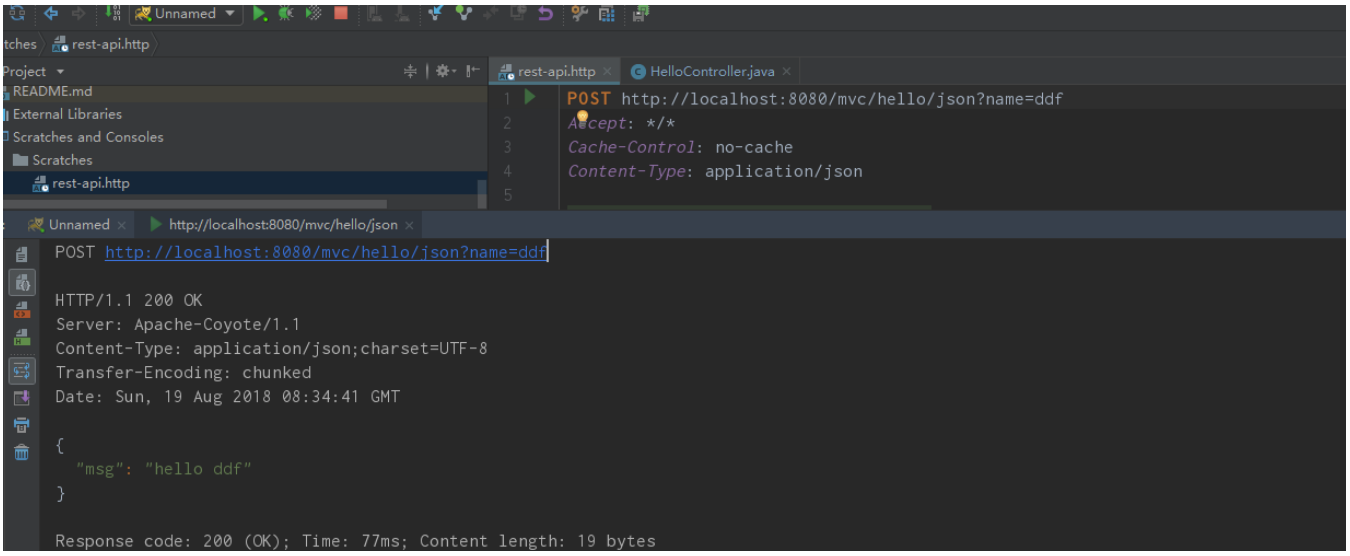
```
package com.ddf.springmvc.annotation.service;

import org.springframework.stereotype.Service;

/**
 * @author DDf on 2018/8/17
 */
@Service
public class HelloService {

    public String hello(String name) {
        return "hello " + name;
    }
}
```

1. 新建 `HelloController.java`

```java
package com.ddf.springmvc.annotation.controller;

import com.ddf.springmvc.annotation.service.HelloService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

/**
 * @author DDf on 2018/8/17
 */
@RestController
@RequestMapping("hello")
public class HelloController {
    @Autowired
    private HelloService helloService;
    @Autowired
    private ApplicationContext applicationContext;

    @RequestMapping("/string")
    public String string(@RequestParam String name) {
        return helloService.hello(name);
    }

    /**
     * 需要导入jackson的包，则会返回json
     * @param name
     * @return
     */
    @RequestMapping("/json")
    public Map json(@RequestParam String name) {
        Map<String, Object> map = new HashMap<>();
        map.put("msg", helloService.hello(name));
        return map;
    }


    /**
     * 在当前的环境中有两个ApplicationContext,一个是根据RootApplicationContextConfig建立的Root ApplicationContext，
     * 一个是根据ServletApplicatioonContextConfig创建的ApplicationContext。 Root ApplicationContext是跟容器，是与spring相关的
     * bean都被管理在这个容器中，而与web相关的容器都被管理在ApplicationContext中（这一块没有强制要求，根据代码会有不同的结果，但推荐这种分
离的写法）
     *
     * 两个ApplicationContext都是由org.springframework.web.context.support.AnnotationConfigWebApplicationContext 实现的
     */
    @RequestMapping("application")
    public void application() {
        System.out.println(applicationContext.getClass());
        System.out.println(applicationContext.getParent().getClass());
        System.out.println(applicationContext==applicationContext.getParent());

        // 这个会打印所有在web容器中注册的bean
        String[] names = applicationContext.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }

        System.out.println("root application context。。。。。。。。。。。。。。。。。。。。。。。");

        // 这个会打印所有在Root IOC容器中注册的bean
        String[] names1 = applicationContext.getParent().getBeanDefinitionNames();
        for (String name : names1) {
            System.out.println(name);
        }
    }
}
```

访问 `{ip}:{port}/{application}/hello/string?name=ddf`，返回 `hello ddf`
访问 `{ip}:{port}/{application}/hello/json?name=ddf`，返回

```
{
  "msg": "hello ddf"
}
```



3. 扩展定制

以上已经完成了基本的整合，现在按照实际使用情况，会持续的往框架中添加扩展组件

3.1 数据源与事务与JdbcTemplate

- 在classpath下新建 `application.properties` 配置文件

```
jdbc.name=druid
jdbc.userName=root
jdbc.password=123456
jdbc.url=jdbc:mysql://localhost:3306/spring-annotation?characterEncoding=utf8&useSSL=true
jdbc.driverClassName=com.mysql.jdbc.Driver
```

- 创建专门用来存取数据源相关的类 `DataSourceConnection.java`

```java
package com.ddf.springmvc.annotation.configuration.jdbc;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

/**
 * @author DDf on 2018/8/13
 * 解析数据连接的参数类
 */
@PropertySource("classpath:application.properties")
@Component
public class DataSourceConnection {
    @Value("${jdbc.name}")
    private String name;
    @Value("${jdbc.userName}")
    private String userName;
    @Value("${jdbc.password}")
    private String password;
    @Value("${jdbc.driverClassName}")
    private String driverClassName;
    @Value("${jdbc.url}")
    private String url;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getDriverClassName() {
        return driverClassName;
    }

    public void setDriverClassName(String driverClassName) {
        this.driverClassName = driverClassName;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    @Override
    public String toString() {
        return "DataSourceConnection{" +
                "name='" + name + '\'' +
                ", userName='" + userName + '\'' +
                ", password='" + password + '\'' +
```

```
                ", driverClassName='" + driverClassName + '\'' +
                ", url='" + url + '\'' +
                '}';
    }
}
```

- 修改主配置类 `RootApplicationContextConfig.java`

```java
package com.ddf.springmvc.annotation.configuration.ioc;

import com.alibaba.druid.pool.DruidDataSource;
import com.ddf.springmvc.annotation.configuration.jdbc.DataSourceConnection;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Primary;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

/**
 * @author DDf on 2018/8/16
 * 指定Spring根容器的配置类，排除web相关的bean
 *
 * @EnableTransactionManagement 开启基于注解的事务支持
 *
 */
@ComponentScan(value = "com.ddf.springmvc.annotation",
        excludeFilters = {@ComponentScan.Filter(value = Controller.class)})
@EnableTransactionManagement
public class RootApplicationContextConfig {

    public RootApplicationContextConfig() {
        System.out.println("RootApplicationContextConfig配置类被读取。。。。。。。。。。。。。");
    }


    /**
     * 向容器中注入DataSource，属性来自于DataSourceConnection，实际注入的是DataSource的DruidDataSource实现
     * DataSourceConnection会自动从IOC容器中获取
     * @Primary 如果要注入通过类型获取的Bean类型为DataSource， 则默认获取druidDataSource。防止注入多个不同名DataSource其它使用地方报错
     * @param dataSourceConnection
     * @return
     */
    @Bean
    @Primary
    public DataSource druidDataSource(DataSourceConnection dataSourceConnection) {
        System.out.println(dataSourceConnection);
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setName(dataSourceConnection.getName());
        druidDataSource.setUsername(dataSourceConnection.getUserName());
        druidDataSource.setPassword(dataSourceConnection.getPassword());
        druidDataSource.setUrl(dataSourceConnection.getUrl());
        // druidDataSource可以不指定driverClassName，会自动根据url识别
        druidDataSource.setDriverClassName(dataSourceConnection.getDriverClassName());
        return druidDataSource;
    }

    /**
     * 将数据源注入JdbcTemplate，再将JdbcTemplate注入到容器中，使用JdbcTemplate来操作数据库
     * @param druidDataSource
     * @return
     */
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource druidDataSource) {
        return new JdbcTemplate(druidDataSource);
    }


    /**
     * 将数据源注入PlatformTransactionManager，这是一个接口，使用DataSourceTransactionManager的实现来管理事务
     * 注意，如果想要生效，一定要在配置类加@EnableTransactionManagement注解
     * @param druidDataSource
     * @return
     */
```

```
    @Bean
    public PlatformTransactionManager transactionManager(DataSource druidDataSource) {
        return new DataSourceTransactionManager(druidDataSource);
    }
}
```

3.2 全局异常处理

使用 `HandlerExceptionResolver` 来处理全局异常，每当项目中有异常时该类都会尝试解决，我们可以在这个类中将消息封装成 `json` 返回给前端。同时消息可以使用 `MessageSource` 来支持国际化处理。因此我们可以自定义一个异常类 `GlobalException` 继承 `NestedRuntimeException` ，该异常类支持可以直接将消息当做文本返回，也可以支持传入定义的 `code` ，然后将 `code` 转换成国际化资源文件中定义的消息然后再返回给前端，国际化资源文件支持占位符。国际化资源文件中 `Property key` 对应着异常类中的 `code` ，而所有需要使用的 `code` 我们可以统一定义在一个类里。而为了防止一个类中的 `code` 随时间越来越大，我们可以分模块定义 `code` ，而统一定义一个接口返回 `code` 。异常类中接收这个接口，其它所有定义 `code` 的类必须实现这个接口

- 定义返回 `code` 统一接口类 `GlobalExceptionCodeResolver`

```
package com.ddf.springmvc.annotation.configuration.exception;

/**
 * @author DDf on 2018/8/18
 * 为自定义异常类消息代码定义统一接口，所以定义消息代码的类必须实现这个接口
 */
public interface GlobalExceptionCodeResolver {
    String getCode();
}
```

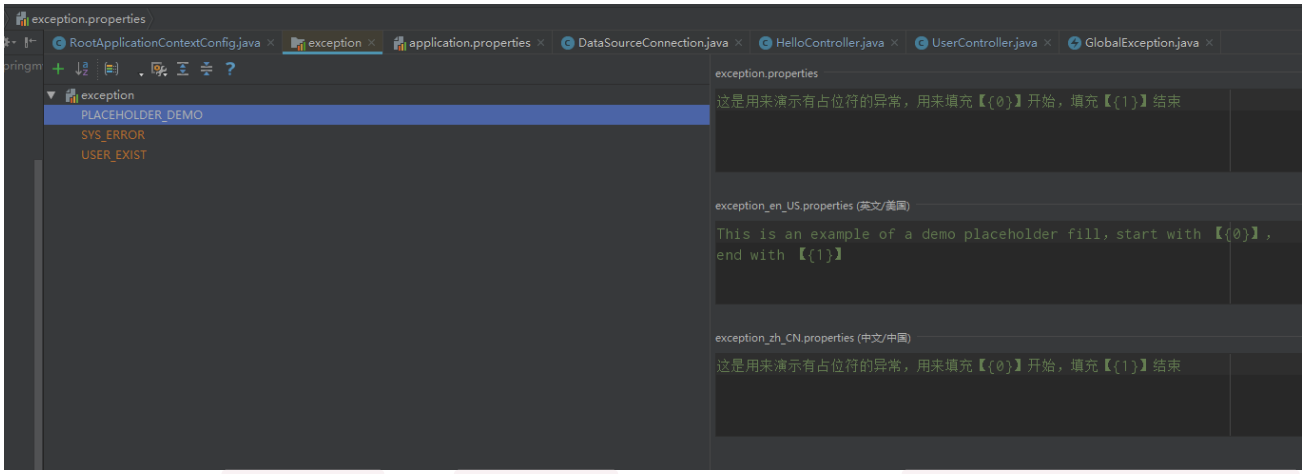
- 使用枚举类定义具体的 `code` ，需要实现接口 `GlobalExceptionCodeResolver`

```
/**
 *
 */
package com.ddf.springmvc.annotation.configuration.exception;

/**
 * 定义异常消息的代码，get方法返回实际值，这个值需要在exception.properties、exception_zh_CN.properties、
 * exception_en_US中配置，请根据实际情况在对应的Locale资源文件中配置，至少配置exception.properties
 * @author DDF 2017年12月1日
 *
 */
public enum GlobalExceptionEnum implements GlobalExceptionCodeResolver {
 SYS_ERROR("SYS_ERROR"),
 PLACEHOLDER_DEMO("PLACEHOLDER_DEMO"),
 USER_EXIST("USER_EXIST")

 ;

 private String code;

 GlobalExceptionEnum (String code) {
  this.code = code;
 }

 @Override
 public String getCode() {
  return code;
 }
}
```

- 自定义异常类 `GlobalException` ，继承 `NestedRuntimeException`

```java
package com.ddf.springmvc.annotation.configuration.exception;

import org.springframework.core.NestedRuntimeException;

import java.util.HashMap;
import java.util.Map;

/**
 * @author DDf on 2018/8/18
 * 自定义异常类
 */
public class GlobalException extends NestedRuntimeException {
    private static final long serialVersionUID = 1L;
    private String code;
    private String message;
    // 可替换消息参数，消息配置中不确定的值用大括号包着数组角标的方式，如{0} 占位，抛出异常的时候使用带params的构造函数赋值，即可替换
    private Object[] params;

    public GlobalException(String msg) {
        super(msg);
        this.code = msg;
    }

    public Map<String, String> toMap() {
        Map<String, String> map = new HashMap<>();
        map.put("code", code);
        map.put("message", message);
        return map;
    }

    /**
     * @param enumClass
     * @param params
     */
    public GlobalException(GlobalExceptionCodeResolver codeResolver, Object... params) {
        super(codeResolver.getCode());
        this.code = codeResolver.getCode();
        this.params = params;
    }

    public GlobalException(GlobalExceptionCodeResolver codeResolver) {
        super(codeResolver.getCode());
        this.code = codeResolver.getCode();
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    @Override
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public Object[] getParams() {
        return params;
    }

    public void setParams(Object[] params) {
        this.params = params;
    }
}
```

- 定义异常 `i18N` 文件,在 `classpath:/exception/` 下, `baseName` 取名为 `exception` 。目前添加一个默认的，一个中文的，一个英文的



- 将异常资源文件指定给 `MessageSource` ，并将 `MessageSource` 交由容器管理，修改主配置类 `RootApplicationContextConfig.java`

```java
package com.ddf.springmvc.annotation.configuration.ioc;

import com.alibaba.druid.pool.DruidDataSource;
import com.ddf.springmvc.annotation.configuration.jdbc.DataSourceConnection;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Primary;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

/**
 * @author DDf on 2018/8/16
 * 指定Spring根容器的配置类，排除web相关的bean
 *
 * @EnableTransactionManagement 开启基于注解的事务支持
 *
 */
@ComponentScan(value = "com.ddf.springmvc.annotation",
        excludeFilters = {@ComponentScan.Filter(value = Controller.class)})
@EnableTransactionManagement
public class RootApplicationContextConfig {

    public RootApplicationContextConfig() {
        System.out.println("RootApplicationContextConfig配置类被读取。。。。。。。。。。。。。。");
    }

    /**
     * 注册解析国际化资源文件的ResourceBundleMessageSource
     * @return
     */
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource resourceBundleMessageSource = new ResourceBundleMessageSource();
        // 指定资源文件，可以多个使用setBasenames指定多个，从classpath下读取
        resourceBundleMessageSource.setBasename("/exception/exception");
        System.out.println("注册MessageSource：  " + resourceBundleMessageSource);
        return resourceBundleMessageSource;
    }

    /**
     * 用户解析和格式化JSON的对象，在这里全局注册一个，用的时候直接注入即可。
     * @return
     */
    @Bean
    @Primary
    public ObjectMapper objectMapper() {
        return new ObjectMapper();
    }

    /**
     * 向容器中注入DataSource，属性来自于DataSourceConnection，实际注入的是DataSource的DruidDataSource实现
     * DataSourceConnection会自动从IOC容器中获取
     * @Primary 如果要注入通过类型获取的Bean类型为DataSource， 则默认获取druidDataSource。防止注入多个不同名DataSource其它使用地方报错
     * @param dataSourceConnection
     * @return
     */
    @Bean
    @Primary
    public DataSource druidDataSource(DataSourceConnection dataSourceConnection) {
        System.out.println(dataSourceConnection);
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setName(dataSourceConnection.getName());
```

```
        druidDataSource.setUsername(dataSourceConnection.getUserName());
        druidDataSource.setPassword(dataSourceConnection.getPassword());
        druidDataSource.setUrl(dataSourceConnection.getUrl());
        // druidDataSource可以不指定driverClassName，会自动根据url识别
        druidDataSource.setDriverClassName(dataSourceConnection.getDriverClassName());
        return druidDataSource;
    }

    /**
     * 将数据源注入JdbcTemplate，再将JdbcTemplate注入到容器中，使用JdbcTemplate来操作数据库
     * @param druidDataSource
     * @return
     */
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource druidDataSource) {
        return new JdbcTemplate(druidDataSource);
    }


    /**
     * 将数据源注入PlatformTransactionManager，这是一个接口，使用DataSourceTransactionManager的实现来管理事务
     * 注意，如果想要生效，一定要在配置类加@EnableTransactionManagement
     * @param druidDataSource
     * @return
     */
    @Bean
    public PlatformTransactionManager transactionManager(DataSource druidDataSource) {
        return new DataSourceTransactionManager(druidDataSource);
    }
}
```

- 新建异常处理类 `GlobalExceptionHandler` 实现 `HandlerExceptionResolver` 接口

```java
package com.ddf.springmvc.annotation.configuration.exception;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Locale;
import com.ddf.springmvc.annotation.configuration.ioc.RootApplicationContextConfig;

/**
 * @author DDf on 2018/8/18
 * 处理全局异常，将异常消息使用json返回到前端
 * 该类需要交由容器管理，为方便统一在主配置类看使用了哪些组件，没有在本类使用注解，
*  需要在主配置中注册组件
 */
public class GlobalExceptionHandler implements HandlerExceptionResolver {
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private MessageSource messageSource;

    @Override
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
{
        response.setContentType("application/json;charset=UTF-8");
        Locale locale = request.getLocale();
        GlobalException globalException;
        if (ex instanceof GlobalException) {
            globalException = (GlobalException) ex;
        } else {
            globalException = new GlobalException(ex.getMessage());
            globalException.setCode(ex.getMessage());
        }
        response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        // 使用messageSource将code到资源文件中根据locale解析成对应的消息文件，并填充占位符。getMessage()重载了多个方法，
        // 如果不存在可以抛异常，也可以 给一个默认值，这里使用了给默认值的处理，默认值即是code
        globalException.setMessage(messageSource.getMessage(globalException.getCode(), globalException.getParams(),
                globalException.getCode(), locale));
        try {
            response.getWriter().write(objectMapper.writeValueAsString(globalException.toMap()));
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

- 在主配置类 `RootApplicationContextConfig` 注册组件 `GlobalExceptionHandler`

```java
package com.ddf.springmvc.annotation.configuration.ioc;

import com.alibaba.druid.pool.DruidDataSource;
import com.ddf.springmvc.annotation.configuration.exception.GlobalExceptionHandler;
import com.ddf.springmvc.annotation.configuration.jdbc.DataSourceConnection;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Primary;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

/**
 * @author DDf on 2018/8/16
 * 指定Spring根容器的配置类，排除web相关的bean
 *
 * @EnableTransactionManagement 开启基于注解的事务支持
 *
 */
@ComponentScan(value = "com.ddf.springmvc.annotation",
        excludeFilters = {@ComponentScan.Filter(value = Controller.class)})
@EnableTransactionManagement
public class RootApplicationContextConfig {

    public RootApplicationContextConfig() {
        System.out.println("RootApplicationContextConfig配置类被读取。。。。。。。。。。。。。。");
    }

    /**
     * 注册解析国际化资源文件的ResourceBundleMessageSource
     * @return
     */
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource resourceBundleMessageSource = new ResourceBundleMessageSource();
        // 指定资源文件，可以多个使用setBasenames指定多个，从classpath下读取
        resourceBundleMessageSource.setBasename("/exception/exception");
        System.out.println("注册MessageSource：  " + resourceBundleMessageSource);
        return resourceBundleMessageSource;
    }

    /**
     * 将处理全局异常的Bean容器管理
     * @return
     */
    @Bean
    public GlobalExceptionHandler globalExceptionHandler() {
        return new GlobalExceptionHandler();
    }

    /**
     * 用户解析和格式化JSON的对象，在这里全局注册一个，用的时候直接注入即可。
     * @return
     */
    @Bean
    @Primary
    public ObjectMapper objectMapper() {
        return new ObjectMapper();
    }

    /**
     * 向容器中注入DataSource，属性来自于DataSourceConnection，实际注入的是DataSource的DruidDataSource实现
     * DataSourceConnection会自动从IOC容器中获取
```

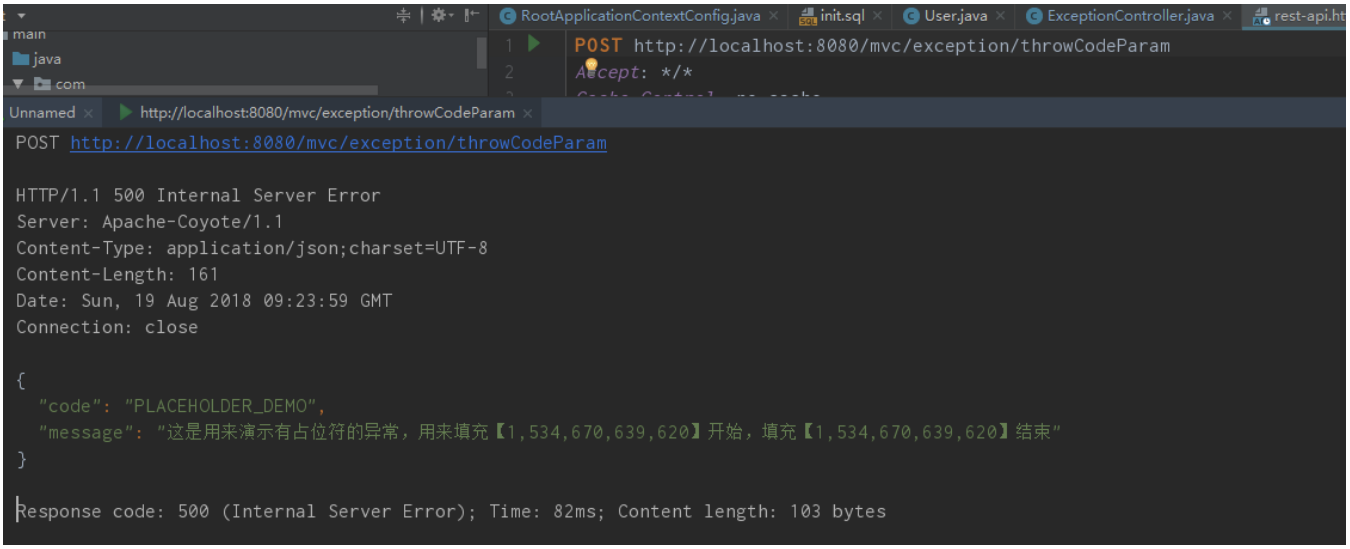
```
     * @Primary 如果要注入通过类型获取的Bean类型为DataSource，则默认获取druidDataSource。防止注入多个不同名DataSource其它使用地方报错
     * @param dataSourceConnection
     * @return
     */
    @Bean
    @Primary
    public DataSource druidDataSource(DataSourceConnection dataSourceConnection) {
        System.out.println(dataSourceConnection);
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setName(dataSourceConnection.getName());
        druidDataSource.setUsername(dataSourceConnection.getUserName());
        druidDataSource.setPassword(dataSourceConnection.getPassword());
        druidDataSource.setUrl(dataSourceConnection.getUrl());
        // druidDataSource可以不指定driverClassName，会自动根据url识别
        druidDataSource.setDriverClassName(dataSourceConnection.getDriverClassName());
        return druidDataSource;
    }

    /**
     * 将数据源注入JdbcTemplate，再将JdbcTemplate注入到容器中，使用JdbcTemplate来操作数据库
     * @param druidDataSource
     * @return
     */
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource druidDataSource) {
        return new JdbcTemplate(druidDataSource);
    }


    /**
     * 将数据源注入PlatformTransactionManager，这是一个接口，使用DataSourceTransactionManager的实现来管理事务
     * 注意，如果想要生效，一定要在配置类加@EnableTransactionManagement
     * @param druidDataSource
     * @return
     */
    @Bean
    public PlatformTransactionManager transactionManager(DataSource druidDataSource) {
        return new DataSourceTransactionManager(druidDataSource);
    }
}
```

3.3 测试事务与异常

3.3.1 异常演示

新建异常演示类 `ExceptionController.java`

```java
package com.ddf.springmvc.annotation.controller;

import com.ddf.springmvc.annotation.configuration.exception.GlobalException;
import com.ddf.springmvc.annotation.configuration.exception.GlobalExceptionEnum;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author DDf on 2018/8/18
 */
@Controller
@RequestMapping("exception")
public class ExceptionController {

    /**
     * 抛出非GlobalException的异常也可以处理
     */
    @RequestMapping("throwString")
    public void throwString() {
        throw new RuntimeException("运行出错");
    }

    /**
     * 抛出GlobalException，code需要在实现了GlobalExceptionCodeResolver接口的类中定义，该code还需要在资源文件exception.properties中定义
     * 对应的value来作为消息解析 /resources/exception/exception.properties exception_en_US.properties exception_ch_CN.properties
     */
    @RequestMapping("throwCode")
    public void throwCode() {
        throw new GlobalException(GlobalExceptionEnum.SYS_ERROR);
    }


    /**
     * 抛出GlobalException，code需要在实现了GlobalExceptionCodeResolver接口的类中定义，该code还需要在资源文件exception.properties中定义
     * 对应的value来作为消息解析 /resources/exception/exception.properties exception_en_US.properties exception_ch_CN.properties
     * ,资源文件中对应key的value可以使用{0}{1}占位符，然后在构造函数中传入替换的值，即可替换
     */
    @RequestMapping("throwCodeParam")
    public void throwCodeParam() {
        throw new GlobalException(GlobalExceptionEnum.PLACEHOLDER_DEMO, System.currentTimeMillis(),
                System.currentTimeMillis());
    }
}
```



3.3.2 事务演示

此章节对数据库操作前面主配置类中注入的 `JdbcTemplate`

- 在上述注入的数据库连接建立表,数据库的建立与数据表的建立需要分开执行，数据库建立后，进入对应的数据库执行建表语句

```
CREATE DATABASE `spring-annotation` CHARACTER SET utf8 COLLATE utf8_general_ci;

USE spring-annotation
DROP TABLE IF EXISTS USER;
CREATE TABLE USER(
 ID INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
 NAME VARCHAR(64),
 PASSWORD VARCHAR(64),
 BIRTH_DAY DATE,
 TEL VARCHAR(32),
  ADDRESS VARCHAR(200)
);
```

- 创建实体类 `User.java`

```java
package com.ddf.springmvc.annotation.entity;

import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

/**
 * @author DDf on 2018/8/19
 */
public class User implements RowMapper {
    private Integer id;
    private String name;
    private String password;
    private Date birthDay;
    private String address;
    private String tel;

    public User() {
    }

    public User(Integer id, String name, String password, Date birthDay, String address, String tel) {
        this.id = id;
        this.name = name;
        this.password = password;
        this.birthDay = birthDay;
        this.address = address;
        this.tel = tel;
    }

    public User(String name, String password, Date birthDay, String address, String tel) {
        this.name = name;
        this.password = password;
        this.birthDay = birthDay;
        this.address = address;
        this.tel = tel;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthDay() {
        return birthDay;
    }

    public void setBirthDay(Date birthDay) {
        this.birthDay = birthDay;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
```

```
    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public User mapRow(ResultSet resultSet, int i) throws SQLException {
        User user = new User();
        user.setId(resultSet.getInt("id"));
        user.setName(resultSet.getString("name"));
        user.setPassword(resultSet.getString("password"));
        user.setBirthDay(resultSet.getDate("BIRTH_DAY"));
        user.setAddress(resultSet.getString("address"));
        user.setTel(resultSet.getString("tel"));
        return user;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", password='" + password + '\'' +
                ", birthDay=" + birthDay +
                ", address='" + address + '\'' +
                ", tel='" + tel + '\'' +
                '}';
    }
}
```

- 用户操作，需要判断重复，先定义用户存在的异常消息，先定义用户存在的代码，修改 `GlobalExceptionEnum.java`

```java
/**
 *
 */
package com.ddf.springmvc.annotation.configuration.exception;

/**
 * 定义异常消息的代码，get方法返回实际值，这个值需要在exception.properties、exception_zh_CN.properties、
 * exception_en_US中配置，请根据实际情况在对应的Locale资源文件中配置，至少配置exception.properties
 * @author DDF 2017年12月1日
 *
 */
public enum GlobalExceptionEnum implements GlobalExceptionCodeResolver {
 SYS_ERROR("SYS_ERROR"),
 PLACEHOLDER_DEMO("PLACEHOLDER_DEMO"),
 USER_EXIST("USER_EXIST")

 ;

 private String code;

 GlobalExceptionEnum (String code) {
  this.code = code;
 }

 @Override
 public String getCode() {
  return code;
 }
}
```

- 在异常资源文件中定义用户存在的代码对应的异常消息
  默认exception.properties

```
USER_EXIST=用户【{0}】已经存在
```

中文exception_zh_CN.properties

```
USER_EXIST=用户【{0}】已经存在
```

英文exception_en_US.properties

```
USER_EXIST=User of name [{0}] is exist
```

- 创建操作用户的数据层 `UserDao.java`

```java
package com.ddf.springmvc.annotation.configuration.dao;

import com.ddf.springmvc.annotation.configuration.exception.GlobalException;
import com.ddf.springmvc.annotation.configuration.exception.GlobalExceptionEnum;
import com.ddf.springmvc.annotation.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Map;

/**
 * @author DDf on 2018/8/19
 */
@Repository
public class UserDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void addUser(User user) {

        String select = "SELECT * FROM USER WHERE NAME = ?";
        List<Map<String, Object>> list = jdbcTemplate.queryForList(select, user.getName());
        if (list != null && list.size() > 0) {
            throw new GlobalException(GlobalExceptionEnum.USER_EXIST, user.getName());
        }

        String sql = "INSERT INTO USER(NAME, PASSWORD, BIRTH_DAY, TEL, ADDRESS) VALUES (?, ?, ?, ?, ?)";
        jdbcTemplate.update(sql, user.getName(), user.getPassword(), user.getBirthDay(), user.getTel(), user.getAddress());
    }

    public List<User> list() {
        String sql = "SELECT * FROM USER ";
        return jdbcTemplate.query(sql, new User());
    }
}
```

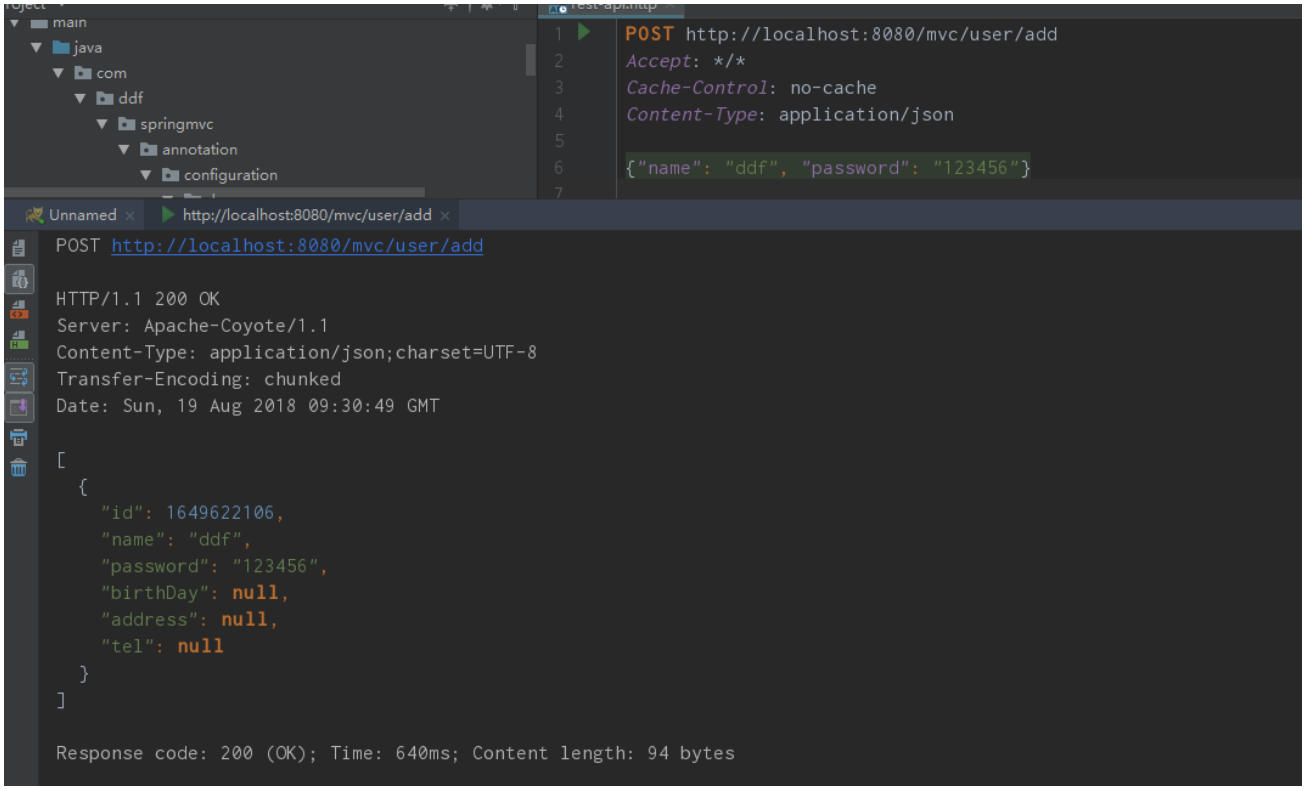
- 创建用户业务层 `UserService.java`

```java
package com.ddf.springmvc.annotation.service;

import com.ddf.springmvc.annotation.configuration.dao.UserDao;
import com.ddf.springmvc.annotation.configuration.exception.GlobalException;
import com.ddf.springmvc.annotation.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

/**
 * @author DDf on 2018/8/19
 */
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    @Transactional
    public void addUser(User user) {
        userDao.addUser(user);
    }

    @Transactional(readOnly = true)
    public List<User> list() {
        return userDao.list();
    }
}
```

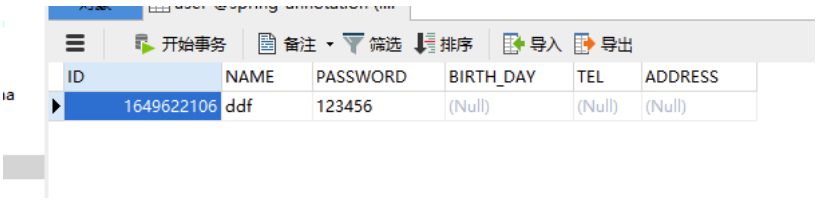
- 创建用户调用层 `UserController.java`

```java
package com.ddf.springmvc.annotation.controller;

import com.ddf.springmvc.annotation.entity.User;
import com.ddf.springmvc.annotation.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

/**
 * @author DDf on 2018/8/19
 */
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/add")
    public List<User> addUser(@RequestBody User user) {
        userService.addUser(user);
        return userService.list();
    }

    @RequestMapping("/list")
    public List<User> list() {
        return userService.list();
    }
}
```

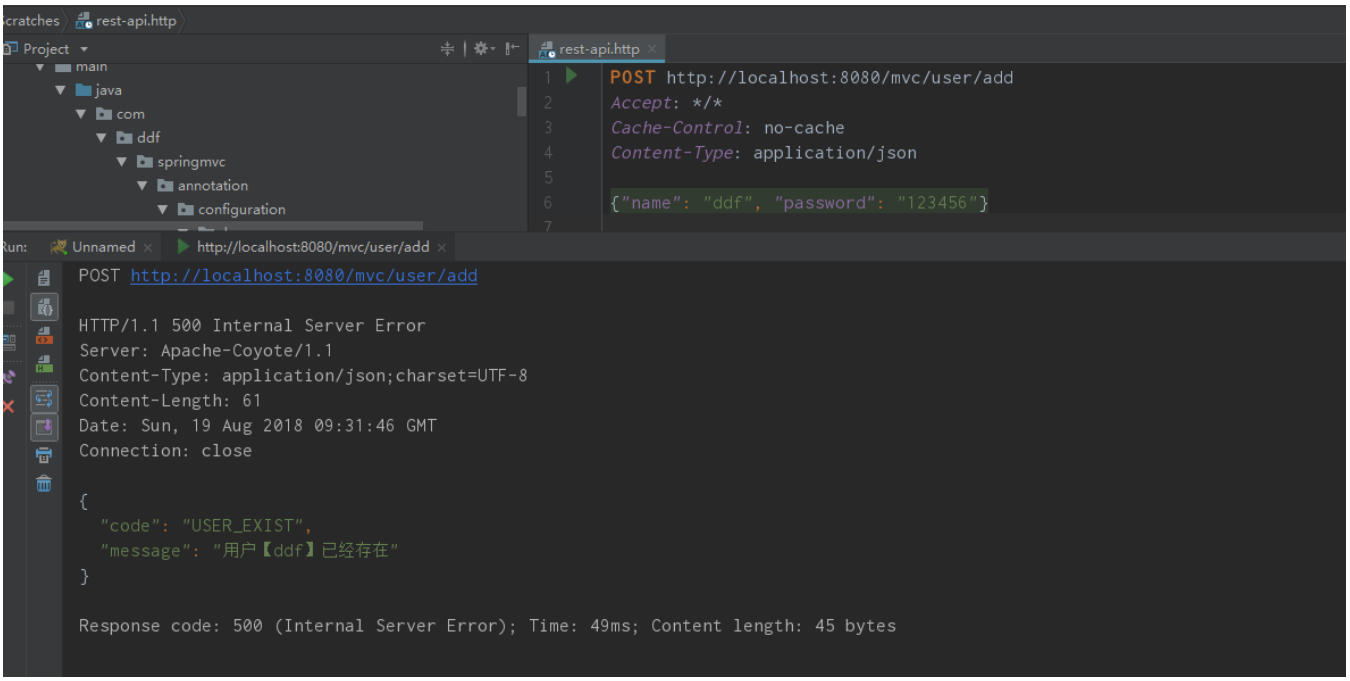
- 演示，添加用户，因为添加用户做了重复性判断，因此保存两次即可，既可以测试事务，又可以看到异常
  第一次调用成功，



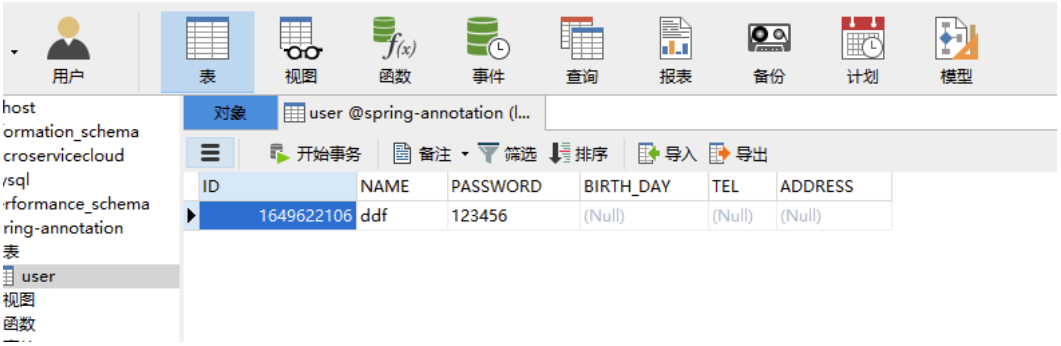
查看数据库，增加了一条数据



再次调用，提示用户已经存在，而且用户名作为占位符被填充了



再次查看数据库，表里依然还是之前的那条数据


用户
表
视图
函数
事件
查询
报表
备份
计划
模型
host
ormation_schema
croservicecloud
/sql
rformance_schema
ring-annotation
表
user
视图
函数
对象
user @spring-annotation (l...
开始事务
备注
筛选
排序
导入
导出
ID
NAME
PASSWORD
BIRTH_DAY
TEL
ADDRESS
1649622106
ddf
123456
(Null)
(Null)
(Null)