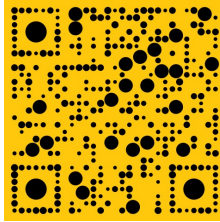


Spring 入门

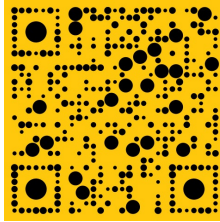
javaboy

江南一点雨

www.javaboy.org

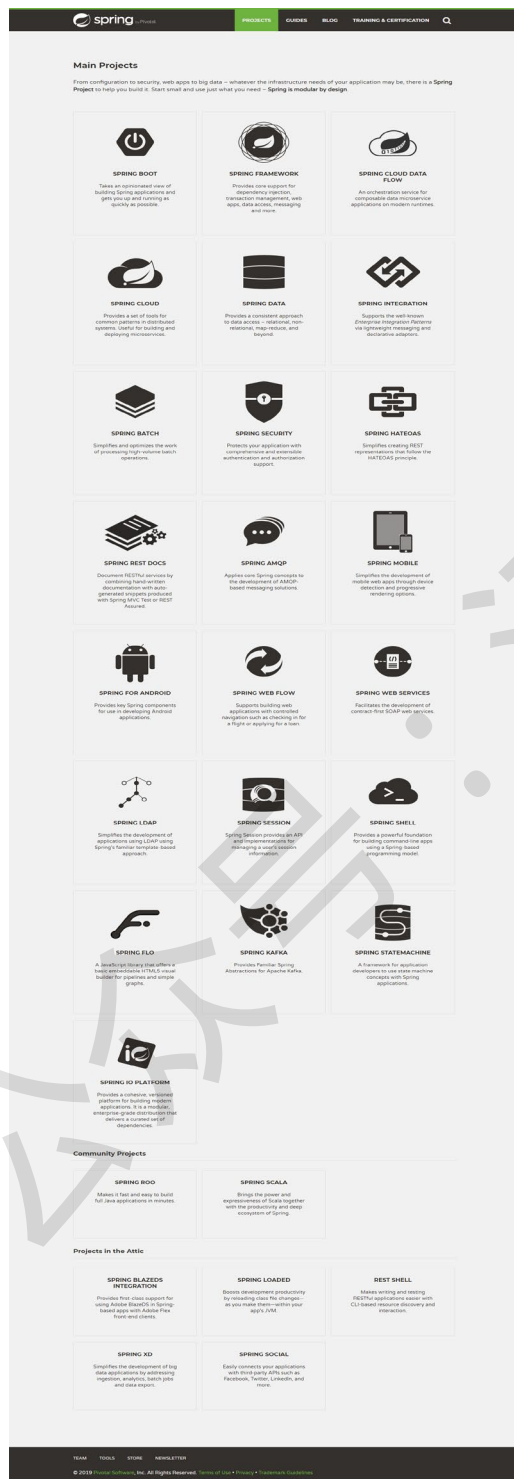


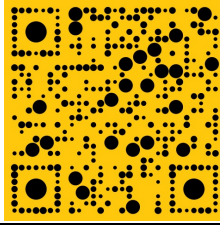
1. Spring 简介.....	2
2. Spring 下载.....	3
3.1 loc.....	4
3.1.1 loc 概念.....	4
3.1.2 loc 初体验.....	5
3.2 Bean 的获取.....	7
3.3 属性的注入.....	8
3.3.1 构造方法注入.....	8
3.3.2 set 方法注入.....	9
3.3.3 p 名称空间注入.....	10
3.3.4 外部 Bean 的注入.....	10
3.4 复杂属性的注入.....	14
3.4.1 对象注入.....	14
3.4.2 数组注入.....	14
3.4.3 Map 注入.....	15
3.4.4 Properties 注入.....	15
3.5 Java 配置.....	18
3.6 自动化配置.....	20
3.6.1 准备工作.....	20
3.6.2 Java 代码配置自动扫描.....	21
3.6.3 XML 配置自动化扫描.....	22
3.6.4 对象注入.....	23
3.7 条件注解.....	24
3.7.1 条件注解.....	24
3.7.2 多环境切换.....	26
3.8 其他.....	30
3.8.1 Bean 的作用域.....	30
3.8.2 id 和 name 的区别.....	31
3.8.3 混合配置.....	32
4. Aware 接口.....	33
5.1 Aop.....	34
5.1.1 Aop 的实现.....	35
5.2 动态代理.....	35
5.3 五种通知.....	37
5.4 XML 配置 Aop.....	45
6. JdbcTemplate.....	47
6.1 准备.....	47
6.2 Java 配置.....	49
6.3 XML 配置.....	52
7. 事务.....	54
7.1 XML 配置.....	56
7.2 Java 配置.....	57



1. Spring 简介

我们常说的 Spring 实际上是指 Spring Framework，而 Spring Framework 只是 Spring 家族中的一个分支而已。那么 Spring 家族都有哪些东西呢？





Spring 是为了解决企业级应用开发的复杂性而创建的。在 Spring 之前，有一个重量级的工具叫做 EJB，使用 Spring 可以让 Java Bean 之间进行有效的解耦，而这个操作之前只有 EJB 才能完成，EJB 过于臃肿，使用很少。Spring 不仅仅局限于服务端的开发，在测试性和松耦合方面都有很好的表现。

一般来说，初学者主要掌握 Spring 四个方面的功能：

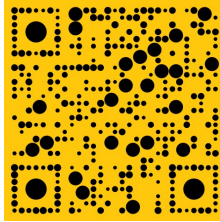
- IoC/DI
- AOP
- 事务
- JdbcTemplate
















2. Spring 下载

正常来说，我们在项目中添加 Maven 依赖就可以直接使用 Spring 了，如果需要单独下载 jar，下载地址如下：

- <https://repo.spring.io/libs-release-local/org/springframework/spring/5.2.1.RELEASE/>

下载成功后，Spring 中的组件，大致上提供了如下功能：



	spring-aop-4.3.7.RELEASE.jar	提供 AOP（面向切面编程）的实现	2017/3/1 8:32	Executable Jar File	372
	spring-aspects-4.3.7.RELEASE.jar	提供对 AspectJ 框架的整合	2017/3/1 8:35	Executable Jar File	58
	spring-beans-4.3.7.RELEASE.jar	提供控制反转的基础实现	2017/3/1 8:31	Executable Jar File	745
	spring-context-4.3.7.RELEASE.jar	spring-context 是在 Ioc 基础功能上继续扩展服务	2017/3/1 8:31	Executable Jar File	1,113
	spring-context-support-4.3.7.RELEASE.jar	spring-context-support 是对 spring-context 的扩展	2017/3/1 8:31	Executable Jar File	183
	spring-core-4.3.7.RELEASE.jar	这个是 spring 的核心组件	2017/3/1 8:31	Executable Jar File	1,093
	spring-expression-4.3.7.RELEASE.jar	spring 对表达式语言的支持	2017/3/1 8:32	Executable Jar File	258
	spring-instrument-4.3.7.RELEASE.jar		2017/3/1 8:32	Executable Jar File	8
	spring-instrument-tomcat-4.3.7.RELEASE.jar		2017/3/1 8:42	Executable Jar File	11
	spring-jdbc-4.3.7.RELEASE.jar	提供了对 Jdbc 模板的支持	2017/3/1 8:32	Executable Jar File	418
	spring-jms-4.3.7.RELEASE.jar	提供对 JMS 通信的支持	2017/3/1 8:42	Executable Jar File	283
	spring-messaging-4.3.7.RELEASE.jar	对消息服务的支持，例如搭配 JMS、搭配 AMQP、搭配 WebSocket 等	2017/3/1 8:42	Executable Jar File	201
	spring-orm-4.3.7.RELEASE.jar	整合第三方 ORM 框架（例如 Hibernate）时使用	2017/3/1 8:34	Executable Jar File	466
	spring-oxm-4.3.7.RELEASE.jar		2017/3/1 8:33	Executable Jar File	84
	spring-test-4.3.7.RELEASE.jar	这个是对单元测试的支持	2017/3/1 8:35	Executable Jar File	585
	spring-tx-4.3.7.RELEASE.jar	对事务的支持	2017/3/1 8:32	Executable Jar File	261
	spring-web-4.3.7.RELEASE.jar	对 web 开发的支持	2017/3/1 8:34	Executable Jar File	799
	spring-webmvc-4.3.7.RELEASE.jar		2017/3/1 8:34	Executable Jar File	895
	spring-webmvc-portlet-4.3.7.RELEASE.jar		2017/3/1 8:34	Executable Jar File	173
	spring-websocket-4.3.7.RELEASE.jar	对 WebSocket 的支持	2017/3/1 8:35	Executable Jar File	447

3.1 Ioc

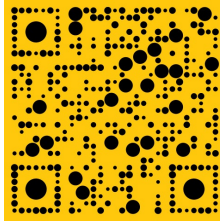
3.1.1 Ioc 概念

Ioc（Inversion of Control），中文叫做控制反转。这是一个概念，也是一种思想。控制反转，实际上就是指对一个对象的控制权的反转。例如，如下代码：

```
public class Book {
    private Integer id;
    private String name;
    private Double price;
    //省略 getter/setter
}

public class User {
    private Integer id;
    private String name;
    private Integer age;

    public void doSth() {
        Book book = new Book();
        book.setId(1);
        book.setName("故事新编");
        book.setPrice((double) 20);
    }
}
```



```
}  
}
```

在这种情况下，Book 对象的控制权在 User 对象里边，这样，Book 和 User 高度耦合，如果在其他对象中需要使用 Book 对象，得重新创建，也就是说，对象的创建、初始化、销毁等操作，统统都要开发者自己来完成。如果能够将这些操作交给容器来管理，开发者就可以极大的从对象的创建中解脱出来。

使用 Spring 之后，我们可以将对象的创建、初始化、销毁等操作交给 Spring 容器来管理。就是说，在项目启动时，所有的 Bean 都将自己注册到 Spring 容器中去（如果有必要的话），然后如果其他 Bean 需要使用到这个 Bean，则不需要自己去 new，而是直接去 Spring 容器去要。

通过一个简单的例子看下这个过程。

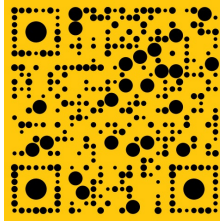
3.1.2 loc 初体验

首先创建一个普通的 Maven 项目，然后引入 spring-context 依赖，如下：

```
<dependencies>  
  <dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-context</artifactId>  
    <version>5.1.9.RELEASE</version>  
  </dependency>  
</dependencies>
```

接下来，在 resources 目录下创建一个 spring 的配置文件（注意，一定要先添加依赖，后创建配置文件，否则创建配置文件时，没有模板选项）：

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://www.springframework.org/schema/bean
```



```
s http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
</beans>
```

在这个文件中，我们可以配置所有需要注册到 Spring 容器的 Bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
s http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="org.javaboy.Book" id="book"/>
</beans>
```

class 属性表示需要注册的 bean 的全路径，id 则表示 bean 的唯一标记，也可以 name 属性作为 bean 的标记，在超过 99% 的情况下，id 和 name 其实是一样的，特殊情况下不一样。

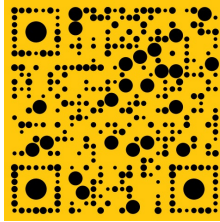
接下来，加载这个配置文件：

```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication
ionContext("applicationContext.xml");
    }
}
```

执行 main 方法，配置文件就会被自动加载，进而在 Spring 中初始化一个 Book 实例。此时，我们显式的指定 Book 类的无参构造方法，并在无参构造方法中打印日志，可以看到无参构造方法执行了，进而证明对象已经在 Spring 容器中初始化了。

最后，通过 getBean 方法，可以从容器中去获取对象：

```
public class Main {
    public static void main(String[] args) {
```



```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat
ionContext("applicationContext.xml");
Book book = (Book) ctx.getBean("book");
System.out.println(book);
}
}
```

加载方式，除了 ClassPathXmlApplicationContext 之外（去 classpath 下查找配置文件），另外也可以使用 FileSystemXmlApplicationContext，FileSystemXmlApplicationContext 会从操作系统路径下去寻找配置文件。

```
public class Main {
    public static void main(String[] args) {
        FileSystemXmlApplicationContext ctx = new FileSystemXmlApplic
ationContext("F:\\workspace5\\workspace\\spring\\spring-ioc\\src\\ma
in\\resources\\applicationContext.xml");
        Book book = (Book) ctx.getBean("book");
        System.out.println(book);
    }
}
```

3.2 Bean 的获取

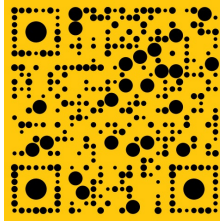
在上一小节中，我们通过 ctx.getBean 方法来从 Spring 容器中获取 Bean，传入的参数是 Bean 的 name 或者 id 属性。除了这种方式之外，也可以直接通过 Class 去获取一个 Bean。

```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat
ionContext("applicationContext.xml");
        Book book = ctx.getBean(Book.class);
        System.out.println(book);
    }
}
```

这种方式有一个很大的弊端，如果存在多个实例，这种方式就不可用，例如，

xml 文件中存在两个 Bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
<bean class="org.javaboy.Book" id="book"/>
<bean class="org.javaboy.Book" id="book2"/>
</beans>
```

此时，如果通过 Class 去查找 Bean，会报如下错误：

```
Exception in thread "main" org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'org.javaboy.Book' available: expected single matching
bean but found 2: book,book2
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBean(DefaultListableBeanFactory.java:1144)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:411)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:344)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:337)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1123)
    at org.javaboy.Main.main(Main.java:9)
```

所以，一般建议使用 name 或者 id 去获取 Bean 的实例。

3.3 属性的注入

在 XML 配置中，属性的注入存在多种方式。

3.3.1 构造方法注入

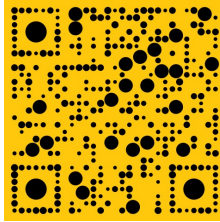
通过 Bean 的构造方法给 Bean 的属性注入值。

1. 第一步首先给 Bean 添加对应的构造方法：

```
public class Book {
    private Integer id;
    private String name;
    private Double price;

    public Book() {
        System.out.println("-----book init-----");
    }

    public Book(Integer id, String name, Double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
```



```
}  
}
```

2.在 xml 文件中注入 Bean

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
s http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <bean class="org.javaboy.Book" id="book">  
        <constructor-arg index="0" value="1"/>  
        <constructor-arg index="1" value="三国演义"/>  
        <constructor-arg index="2" value="30"/>  
    </bean>  
</beans>
```

这里需要注意的是，constructor-arg 中的 index 和 Book 中的构造方法参数一一对应。写的顺序可以颠倒，但是 index 的值和 value 要一一对应。

另一种构造方法中的属性注入，则是通过直接指定参数名来注入：

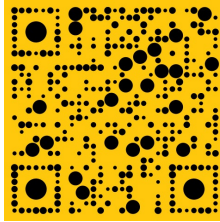
```
<bean class="org.javaboy.Book" id="book2">  
    <constructor-arg name="id" value="2"/>  
    <constructor-arg name="name" value="红楼梦"/>  
    <constructor-arg name="price" value="40"/>  
</bean>
```

如果有多个构造方法，则会根据给出参数个数以及参数类型，自动匹配到对应的构造方法上，进而初始化一个对象。

3.3.2 set 方法注入

除了构造方法之外，我们也可以通过 set 方法注入值。

```
<bean class="org.javaboy.Book" id="book3">  
    <property name="id" value="3"/>  
    <property name="name" value="水浒传"/>
```



```
<property name="price" value="30"/>
</bean>
```

set 方法注入，有一个很重要的问题，就是属性名。很多人会有一种错觉，觉得属性名就是你定义的属性名，这个是不对的。在所有的框架中，凡是涉及到反射注入值的，属性名统统都不是 Bean 中定义的属性名，而是通过 Java 中的内省机制分析出来的属性名，简单说，就是根据 get/set 方法分析出来的属性名。

3.3.3 p 名称空间注入

p 名称空间注入，使用的比较少，它本质上也是调用了 set 方法。

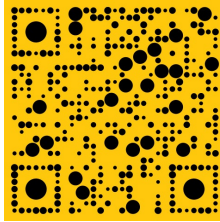
```
<bean class="org.javaboy.Book" id="book4" p:id="4" p:bookName="西游记"
p:price="33"></bean>
```

3.3.4 外部 Bean 的注入

有时候，我们使用一些外部 Bean，这些 Bean 可能没有构造方法，而是通过 Builder 来构造的，这个时候，就无法使用上面的方式来给它注入值了。

例如在 OkHttp 的网络请求中，原生的写法如下：

```
public class OkHttpMain {
    public static void main(String[] args) {
        OkHttpClient okHttpClient = new OkHttpClient.Builder()
            .build();
        Request request = new Request.Builder()
            .get()
            .url("http://b.hiphotos.baidu.com/image/h%3D300/sign=ad628627aacc7cd9e52d32d909032104/32fa828ba61ea8d3fcd2e9ce9e0a304e241f5803.jpg")
            .build();
        Call call = okHttpClient.newCall(request);
        call.enqueue(new Callback() {
            @Override
```



```
public void onFailure(@NotNull Call call, @NotNull IOException e) {
    System.out.println(e.getMessage());
}

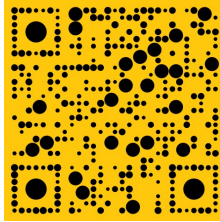
@Override
public void onResponse(@NotNull Call call, @NotNull Response response) throws IOException {
    FileOutputStream out = new FileOutputStream(new File("E:\\123.jpg"));
    int len;
    byte[] buf = new byte[1024];
    InputStream is = response.body().byteStream();
    while ((len = is.read(buf)) != -1) {
        out.write(buf, 0, len);
    }
    out.close();
    is.close();
}
});
}
```

这个 Bean 有一个特点，OkHttpClient 和 Request 两个实例都不是直接 new 出来的，在调用 Builder 方法的过程中，都会给它配置一些默认的参数。这种情况，我们可以使用 静态工厂注入或者实例工厂注入来给 OkHttpClient 提供一个实例。

1.静态工厂注入

首先提供一个 OkHttpClient 的静态工厂：

```
public class OkHttpUtils {
    private static OkHttpClient OkHttpClient;
    public static OkHttpClient getInstance() {
        if (OkHttpClient == null) {
            OkHttpClient = new OkHttpClient.Builder().build();
        }
        return OkHttpClient;
    }
}
```



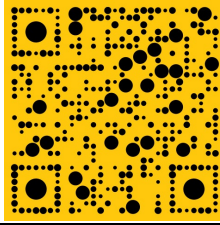
```
}  
}
```

在 xml 文件中，配置该静态工厂：

```
<bean class="org.javaboy.OkHttpUtils" factory-method="getInstance" id="okHttpClient"></bean>
```

这个配置表示 OkHttpUtils 类中的 getInstance 是我们需要的实例，实例的名字就叫 okHttpClient。然后，在 Java 代码中，获取到这个实例，就可以直接使用了。

```
public class OkHttpMain {  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication  
ionContext("applicationContext.xml");  
        OkHttpClient okHttpClient = ctx.getBean("okHttpClient", OkHttp  
pClient.class);  
        Request request = new Request.Builder()  
            .get()  
            .url("http://b.hiphotos.baidu.com/image/h%3D300/sign=a  
d628627aacc7cd9e52d32d909032104/32fa828ba61ea8d3fcd2e9ce9e0a304e241f  
5803.jpg")  
            .build();  
        Call call = okHttpClient.newCall(request);  
        call.enqueue(new Callback() {  
            @Override  
            public void onFailure(@NotNull Call call, @NotNull IOExcep  
tion e) {  
                System.out.println(e.getMessage());  
            }  
            @Override  
            public void onResponse(@NotNull Call call, @NotNull Respon  
se response) throws IOException {  
                FileOutputStream out = new FileOutputStream(new File("E:\\123.jpg"));  
                int len;  
                byte[] buf = new byte[1024];  
                InputStream is = response.body().byteStream();  
                while ((len = is.read(buf)) != -1) {
```



```
        out.write(buf, 0, len);
    }
    out.close();
    is.close();
}
});
}
}
```

2.实例工厂注入

实例工厂就是工厂方法是一个实例方法，这样，工厂类必须实例化之后才可以调用工厂方法。

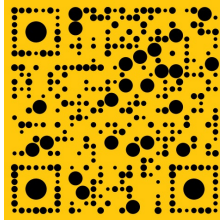
这次的工厂类如下：

```
public class OkHttpUtils {
    private OkHttpClient OkHttpClient;
    public OkHttpClient getInstance() {
        if (OkHttpClient == null) {
            OkHttpClient = new OkHttpClient.Builder().build();
        }
        return OkHttpClient;
    }
}
```

此时，在 xml 文件中，需要首先提供工厂方法的实例，然后才可以调用工厂方法：

```
<bean class="org.javaboy.OkHttpUtils" id="okHttpUtils"/>
<bean class="okhttp3.OkHttpClient" factory-bean="okHttpUtils" factory-
method="getInstance" id="okHttpClient"></bean>
```

自己写的 Bean 一般不会使用这两种方式注入，但是，如果需要引入外部 jar，外部 jar 的类的初始化，有可能需要使用这两种方式。



3.4 复杂属性的注入

3.4.1 对象注入

```
<bean class="org.javaboy.User" id="user">
    <property name="cat" ref="cat"/>
</bean>
<bean class="org.javaboy.Cat" id="cat">
    <property name="name" value="小白"/>
    <property name="color" value="白色"/>
</bean>
```

可以通过 xml 注入对象，通过 ref 来引用一个对象。

3.4.2 数组注入

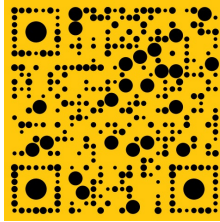
数组注入和集合注入在 xml 中的配置是一样的。如下：

```
<bean class="org.javaboy.User" id="user">
    <property name="cat" ref="cat"/>
    <property name="favorites">
        <array>
            <value>足球</value>
            <value>篮球</value>
            <value>乒乓球</value>
        </array>
    </property>
</bean>
<bean class="org.javaboy.Cat" id="cat">
    <property name="name" value="小白"/>
    <property name="color" value="白色"/>
</bean>
```

注意，array 节点，也可以被 list 节点代替。

当然，array 或者 list 节点中也可以是对象。

```
<bean class="org.javaboy.User" id="user">
    <property name="cat" ref="cat"/>
    <property name="favorites">
        <list>
```



```
<value>足球</value>
<value>篮球</value>
<value>乒乓球</value>
</list>
</property>
<property name="cats">
  <list>
    <ref bean="cat"/>
    <ref bean="cat2"/>
    <bean class="org.javaboy.Cat" id="cat3">
      <property name="name" value="小花"/>
      <property name="color" value="花色"/>
    </bean>
  </list>
</property>
</bean>
<bean class="org.javaboy.Cat" id="cat">
  <property name="name" value="小白"/>
  <property name="color" value="白色"/>
</bean>
<bean class="org.javaboy.Cat" id="cat2">
  <property name="name" value="小黑"/>
  <property name="color" value="黑色"/>
</bean>
```

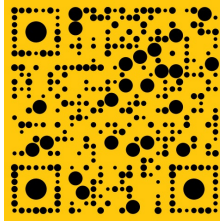
注意，即可以通过 ref 使用外部定义好的 Bean，也可以直接在 list 或者 array 节点中定义 bean。

3.4.3 Map 注入

```
<property name="map">
  <map>
    <entry key="age" value="99"/>
    <entry key="name" value="javaboy"/>
  </map>
</property>
```

3.4.4 Properties 注入

```
<property name="info">
  <props>
    <prop key="age">99</prop>
    <prop key="name">javaboy</prop>
  </props>
</property>
```

```
</props>
</property>
```

以上 Demo, 定义的 User 如下:

```
public class User {
    private Integer id;
    private String name;
    private Integer age;
    private Cat cat;
    private String[] favorites;
    private List<Cat> cats;
    private Map<String, Object> map;
    private Properties info;

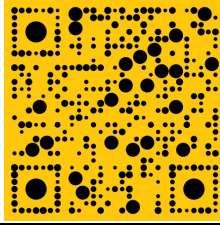
    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", age=" + age +
            ", cat=" + cat +
            ", favorites=" + Arrays.toString(favorites) +
            ", cats=" + cats +
            ", map=" + map +
            ", info=" + info +
            '}';
    }

    public Properties getInfo() {
        return info;
    }

    public void setInfo(Properties info) {
        this.info = info;
    }

    public Map<String, Object> getMap() {
        return map;
    }

    public void setMap(Map<String, Object> map) {
        this.map = map;
    }
}
```



```
}

public List<Cat> getCats() {
    return cats;
}

public void setCats(List<Cat> cats) {
    this.cats = cats;
}

public String[] getFavorites() {
    return favorites;
}

public void setFavorites(String[] favorites) {
    this.favorites = favorites;
}

public User() {
}

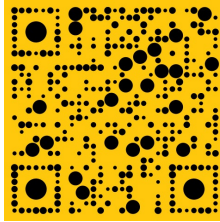
public User(Integer id, String name, Integer age, Cat cat) {
    this.id = id;
    this.name = name;
    this.age = age;
    this.cat = cat;
}

public Cat getCat() {
    return cat;
}

public void setCat(Cat cat) {
    this.cat = cat;
}

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 Integer getAge() {  
    return age;  
}  
  
public void setAge(Integer age) {  
    this.age = age;  
}  
}
```

3.5 Java 配置

在 Spring 中，想要将一个 Bean 注册到 Spring 容器中，整体上来说，有三种不同的方式。

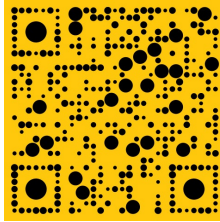
- XML 注入，如前文所说
- Java 配置（通过 Java 代码将 Bean 注册到 Spring 容器中）
- 自动化扫描

这里我们来看 Java 配置。

Java 配置这种方式在 Spring Boot 出现之前，其实很少使用，自从有了 Spring Boot，Java 配置开发被广泛使用，因为在 Spring Boot 中，不使用一行 XML 配置。

例如我有如下一个 Bean：

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



```
}  
}
```

在 Java 配置中，我们用一个 Java 配置类去代替之前的
applicationContext.xml 文件。

```
@Configuration  
public class JavaConfig {  
    @Bean  
    SayHello sayHello() {  
        return new SayHello();  
    }  
}
```

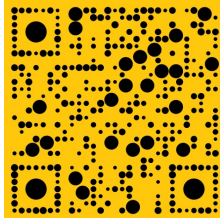
首先在配置类上有一个 @Configuration 注解，这个注解表示这个类不是一个普通类，而是一个配置类，它的作用相当于 applicationContext.xml。然后，定义方法，方法返回对象，方法上添加 @Bean 注解，表示将这个方法的返回值注入的 Spring 容器中去。也就是说，@Bean 所对应的方法，就相当于 applicationContext.xml 中的 bean 节点。

既然是配置类，我们需要在项目启动时加载配置类。

```
public class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ctx = new AnnotationConfig  
        ApplicationContext(JavaConfig.class);  
        SayHello hello = ctx.getBean(SayHello.class);  
        System.out.println(hello.sayHello("javaboy"));  
    }  
}
```

注意，配置的加载，是使用 AnnotationConfigApplicationContext 来实现。

关于 Java 配置，这里有一个需要注意的问题:Bean 的名字是什么？



Bean 的默认名称是方法名。以上面的案例为例，Bean 的名字是 sayHello。

如果开发者想自定义方法名，也是可以的，直接在 @Bean 注解中进行过配

置。如下配置表示修改 Bean 的名字为 javaboy:

```
@Configuration
public class JavaConfig {
    @Bean("javaboy")
    SayHello sayHello() {
        return new SayHello();
    }
}
```

3.6 自动化配置

在我们实际开发中，大量的使用自动配置。

自动化配置既可以通过 Java 配置来实现，也可以通过 xml 配置来实现。

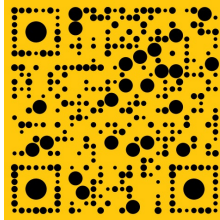
3.6.1 准备工作

例如我有一个 UserService，我希望在自动化扫描时，这个类能够自动注册到 Spring 容器中去，那么可以给该类添加一个 @Service，作为一个标记。

和 @Service 注解功能类似的注解，一共有四个：

- @Component
- @Repository
- @Service
- @Controller

这四个中，另外三个都是基于 @Component 做出来的，而且从目前的源码来看，功能也是一致的，那么为什么要搞三个呢？主要是为了在不同的类上面添加时方便。



- 在 Service 层上，添加注解时，使用 @Service
- 在 Dao 层，添加注解时，使用 @Repository
- 在 Controller 层，添加注解时，使用 @Controller
- 在其他组件上添加注解时，使用 @Component

@Service

```
public class UserService {  
    public List<String> getAllUser() {  
        List<String> users = new ArrayList<>();  
        for (int i = 0; i < 10; i++) {  
            users.add("javaboy:" + i);  
        }  
        return users;  
    }  
}
```

添加完成后，自动化扫描有两种方式，一种就是通过 Java 代码配置自动化扫描，另一种则是通过 xml 文件来配置自动化扫描。

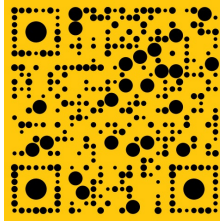
3.6.2 Java 代码配置自动扫描

@Configuration

```
@ComponentScan(basePackages = "org.javaboy.javaconfig.service")  
public class JavaConfig {  
}
```

然后，在项目启动中加载配置类，在配置类中，通过 @ComponentScan 注解指定要扫描的包（如果不指定，默认情况下扫描的是配置类所在的包下载 Bean 以及配置类所在的包下的子包下的类），然后就可以获取 UserService 的实例了：

```
public class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ctx = new AnnotationConfig  
        ApplicationContext(JavaConfig.class);  
        UserService userService = ctx.getBean(UserService.class);  
        System.out.println(userService.getAllUser());  
    }  
}
```



```
}  
}
```

这里有几个问题需要注意：

1.Bean 的名字叫什么？

默认情况下，Bean 的名字是类名首字母小写。例如上面的 UserService，它的实例名，默认就是 userService。如果开发者想要自定义名字，就直接在 @Service 注解中添加即可。

2.有几种扫描方式？

上面的配置，我们是按照包的位置来扫描的。也就是说，Bean 必须放在指定的扫描位置，否则，即使你有 @Service 注解，也扫描不到。

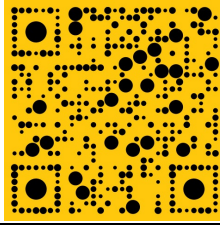
除了按照包的位置来扫描，还有另外一种方式，就是根据注解来扫描。例如如下配置：

```
@Configuration  
@ComponentScan(basePackages = "org.javaboy.javaconfig",useDefaultFilters = true,excludeFilters = {@ComponentScan.Filter(type = FilterType.ANNOTATION,classes = Controller.class)})  
public class JavaConfig {  
}
```

这个配置表示扫描 org.javaboy.javaconfig 下的所有 Bean，但是除了 Controller。

3.6.3 XML 配置自动化扫描

```
<context:component-scan base-package="org.javaboy.javaconfig"/>
```



上面这行配置表示扫描 `org.javaboy.javaconfig` 下的所有 Bean。当然也可以按照类来扫描。

XML 配置完成后，在 Java 代码中加载 XML 配置即可。

```
public class XMLTest {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat
ionContext("applicationContext.xml");
        UserService userService = ctx.getBean(UserService.class);
        List<String> list = userService.getAllUser();
        System.out.println(list);
    }
}
```

也可以在 XML 配置中按照注解的类型进行扫描：

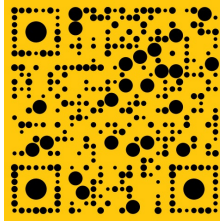
```
<context:component-scan base-package="org.javaboy.javaconfig" use-de
fault-filters="true">
    <context:exclude-filter type="annotation" expression="org.springf
ramework.stereotype.Controller"/>
</context:component-scan>
```

3.6.4 对象注入

自动扫描时的对象注入有三种方式：

1. `@Autowired`
2. `@Resources`
3. `@Injected`

`@Autowired` 是根据类型去查找，然后赋值，这就有一个要求，这个类型只可以有一个对象，否则就会报错。`@Resources` 是根据名称去查找，默认情况下，定义的变量名，就是查找的名称，当然开发者也可以在 `@Resources` 注解中手动指定。所以，如果一个类存在多个实例，那么就应该使用 `@Resources`



去注入，如果非常使用 @Autowired，也是可以的，此时需要配合另外一个注解，@Qualifier，在 @Qualifier 中可以指定变量名，两个一起用（@Qualifier 和 @Autowired）就可以实现通过变量名查找到变量。

```
@Service
public class UserService {

    @Autowired
    UserDao userDao;
    public String hello() {
        return userDao.hello();
    }

    public List<String> getAllUser() {
        List<String> users = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            users.add("javaboy:" + i);
        }
        return users;
    }
}
```

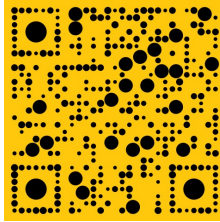
3.7 条件注解

条件注解就是在满足某一个条件的情况下，生效的配置。

3.7.1 条件注解

首先在 Windows 中如何获取操作系统信息？Windows 中查看文件夹目录的命令是 dir，Linux 中查看文件夹目录的命令是 ls，我现在希望当系统运行在 Windows 上时，自动打印出 Windows 上的目录展示命令，Linux 运行时，则自动展示 Linux 上的目录展示命令。

首先定义一个显示文件夹目录的接口：



```
public interface ShowCmd {  
    String showCmd();  
}
```

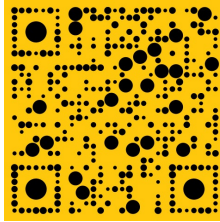
然后，分别实现 Windows 下的实例和 Linux 下的实例：

```
public class WinShowCmd implements ShowCmd {  
    @Override  
    public String showCmd() {  
        return "dir";  
    }  
}  
  
public class LinuxShowCmd implements ShowCmd {  
    @Override  
    public String showCmd() {  
        return "ls";  
    }  
}
```

接下来，定义两个条件，一个是 Windows 下的条件，另一个是 Linux 下的条件。

```
public class WindowsCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
        return context.getEnvironment().getProperty("os.name").toLowerCase().contains("windows");  
    }  
}  
  
public class LinuxCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
        return context.getEnvironment().getProperty("os.name").toLowerCase().contains("linux");  
    }  
}
```

接下来，在定义 Bean 的时候，就可以去配置条件注解了。



```
@Configuration
public class JavaConfig {
    @Bean("showCmd")
    @Conditional(WindowsCondition.class)
    ShowCmd winCmd() {
        return new WinShowCmd();
    }

    @Bean("showCmd")
    @Conditional(LinuxCondition.class)
    ShowCmd linuxCmd() {
        return new LinuxShowCmd();
    }
}
```

这里，一定要给两个 Bean 取相同的名字，这样在调用时，才可以自动匹配。

然后，给每一个 Bean 加上条件注解，当条件中的 matches 方法返回 true 的时候，这个 Bean 的定义就会生效。

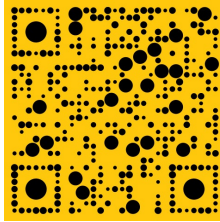
```
public class JavaMain {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new AnnotationConfig
        ApplicationContext(JavaConfig.class);
        ShowCmd showCmd = (ShowCmd) ctx.getBean("showCmd");
        System.out.println(showCmd.showCmd());
    }
}
```

条件注解有一个非常典型的使用场景，就是多环境切换。

3.7.2 多环境切换

开发中，如何在 开发/生产/测试 环境之间进行快速切换？Spring 中提供了 Profile 来解决这个问题，Profile 的底层就是条件注解。这个从 @Profile 注解的定义就可以看出来：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
```



```
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {

    /**
     * The set of profiles for which the annotated component should be
     * registered.
     */
    String[] value();

}

class ProfileCondition implements Condition {

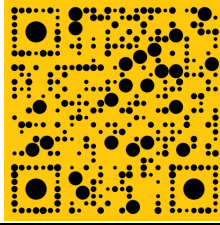
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMet
adata metadata) {
        MultiValueMap<String, Object> attrs = metadata.getAllAnnotati
onAttributes(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(Profiles.
of((String[]) value))) {
                    return true;
                }
            }
            return false;
        }
        return true;
    }

}
```

我们定义一个 DataSource:

```
public class DataSource {
    private String url;
    private String username;
    private String password;

    @Override
    public String toString() {
        return "DataSource{" +
            "url='" + url + '\'' +
```



```
        ", username='" + username + '\\'' +
        ", password='" + password + '\\'' +
        '}'';
    }

    public String getUrl() {
        return url;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

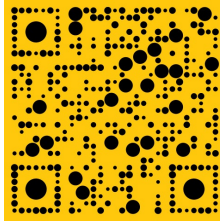
    public String getPassword() {
        return password;
    }

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

然后，在配置 Bean 时，通过 @Profile 注解指定不同的环境：

```
@Bean("ds")
@Profile("dev")
DataSource devDataSource() {
    DataSource dataSource = new DataSource();
    dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/dev");
    dataSource.setUsername("root");
    dataSource.setPassword("123");
    return dataSource;
}

@Bean("ds")
@Profile("prod")
```



```
DataSource prodDataSource() {  
    DataSource dataSource = new DataSource();  
    dataSource.setUrl("jdbc:mysql://192.158.222.33:3306/dev");  
    dataSource.setUsername("jklDasjfk1");  
    dataSource.setPassword("jfsdjflkajkld");  
    return dataSource;  
}
```

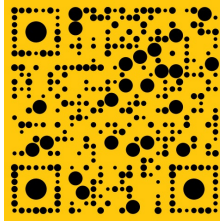
最后，在加载配置类，注意，需要先设置当前环境，然后再去加载配置类：

```
public class JavaMain {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ctx = new AnnotationConfig  
ApplicationContext();  
        ctx.getEnvironment().setActiveProfiles("dev");  
        ctx.register(JavaConfig.class);  
        ctx.refresh();  
        DataSource ds = (DataSource) ctx.getBean("ds");  
        System.out.println(ds);  
    }  
}
```

这个是在 Java 代码中配置的。环境的切换，也可以在 XML 文件中配置，如下配置在 XML 文件中，必须放在其他节点后面。

```
<beans profile="dev">  
    <bean class="org.javaboy.DataSource" id="dataSource">  
        <property name="url" value="jdbc:mysql:///devdb"/>  
        <property name="password" value="root"/>  
        <property name="username" value="root"/>  
    </bean>  
</beans>  
<beans profile="prod">  
    <bean class="org.javaboy.DataSource" id="dataSource">  
        <property name="url" value="jdbc:mysql://111.111.111.111/devd  
b"/>  
        <property name="password" value="jsdfaklfj789345fjsd"/>  
        <property name="username" value="root"/>  
    </bean>  
</beans>
```

启动类中设置当前环境并加载配置：



```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat
ionContext();
        ctx.getEnvironment().setActiveProfiles("prod");
        ctx.setConfigLocation("applicationContext.xml");
        ctx.refresh();
        DataSource dataSource = (DataSource) ctx.getBean("dataSource
");
        System.out.println(dataSource);
    }
}
```

3.8 其他

3.8.1 Bean 的作用域

在 XML 配置中注册的 Bean，或者用 Java 配置注册的 Bean，如果我多次获取，获取到的对象是否是同一个？

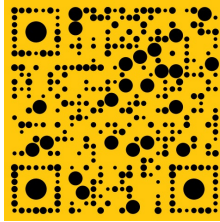
```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat
ionContext("applicationContext.xml");
        User user = ctx.getBean("user", User.class);
        User user2 = ctx.getBean("user", User.class);
        System.out.println(user==user2);
    }
}
```

如上，从 Spring 容器中多次获取同一个 Bean，默认情况下，获取到的实际上是同一个实例。当然我们可以自己手动配置。

```
<bean class="org.javaboy.User" id="user" scope="prototype" />
```

通过在 XML 节点中，设置 scope 属性，我们可以调整默认的实例个数。

scope 的值为 singleton（默认），表示这个 Bean 在 Spring 容器中，是以单



例的形式存在，如果 scope 的值为 prototype，表示这个 Bean 在 Spring 容器中不是单例，多次获取将拿到多个不同的实例。

除了 singleton 和 prototype 之外，还有两个取值，request 和 session。这两个取值在 web 环境下有效。这是在 XML 中的配置，我们也可以在 Java 中配置。

```
@Configuration
public class JavaConfig {
    @Bean
    @Scope("prototype")
    SayHello sayHello() {
        return new SayHello();
    }
}
```

在 Java 代码中，我们可以通过 @Scope 注解指定 Bean 的作用域。

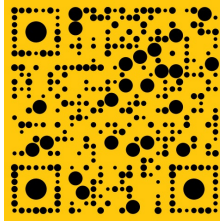
当然，在自动扫描配置中，也可以指定 Bean 的作用域。

```
@Repository
@Scope("prototype")
public class UserDao {
    public String hello() {
        return "userdao";
    }
}
```

3.8.2 id 和 name 的区别

在 XML 配置中，我们可以看到，即可以通过 id 给 Bean 指定一个唯一标识符，也可以通过 name 来指定，大部分情况下这两个作用是一样的，有一个小小区别：

name 支持取多个。多个 name 之间，用 , 隔开：



```
<bean class="org.javaboy.User" name="user,user1,user2,user3" scope="prototype"/>
```

此时，通过 user、user1、user2、user3 都可以获取到当前对象：

```
public class Main {  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication  
ionContext("applicationContext.xml");  
        User user = ctx.getBean("user", User.class);  
        User user2 = ctx.getBean("user2", User.class);  
        System.out.println(user);  
        System.out.println(user2);  
    }  
}
```

而 id 不支持有多个值。如果强行用，隔开，它还是一个值。例如如下配置：

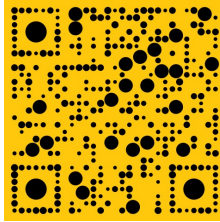
```
<bean class="org.javaboy.User" id="user,user1,user2,user3" scope="pr  
ototype" />
```

这个配置表示 Bean 的名字为 user,user1,user2,user3，具体调用如下：

```
public class Main {  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication  
ionContext("applicationContext.xml");  
        User user = ctx.getBean("user,user1,user2,user3", User.clas  
s);  
        User user2 = ctx.getBean("user,user1,user2,user3", User.clas  
s);  
        System.out.println(user);  
        System.out.println(user2);  
    }  
}
```

3.8.3 混合配置

混合配置就是 Java 配置+XML 配置。混用的话，可以在 Java 配置中引入 XML 配置。



```
@Configuration
@ImportResource("classpath:applicationContext.xml")
public class JavaConfig {
}
```

在 Java 配置中，通过 `@ImportResource` 注解可以导入一个 XML 配置。

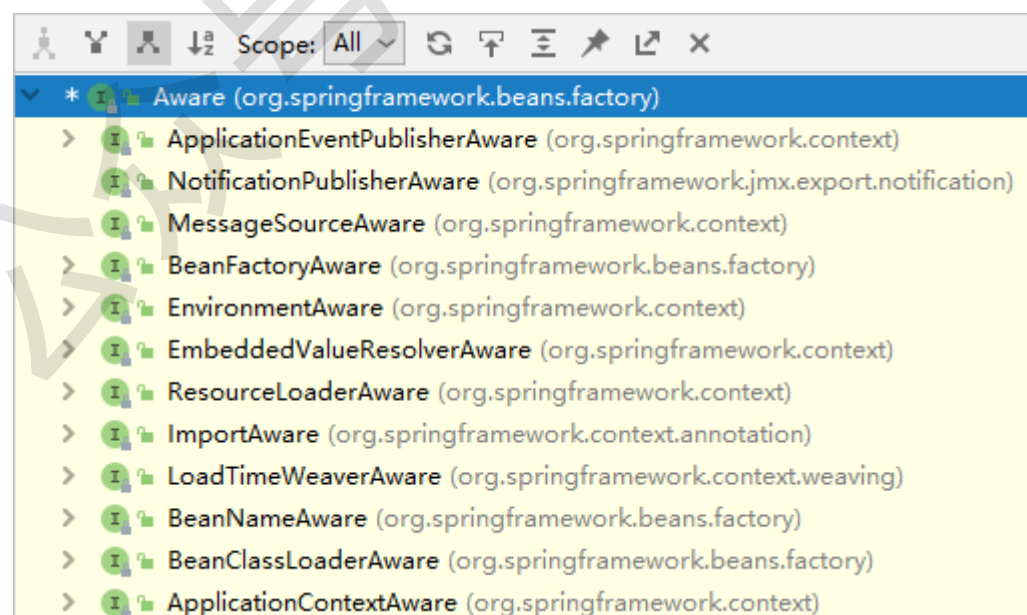
4. Aware 接口

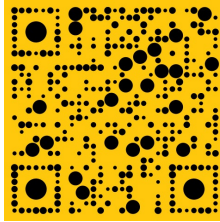
Aware 接口，从字面上理解就是感知捕获。单纯的一个 Bean 是没有知觉的。

在 3.6.4 节的场景中，之所以 UserDao 能够注入到 UserService，有一个前提，就是它两个都是被 Spring 容器管理的。如果直接 new 一个 UserService，这是没用的，因为 UserService 没有被 Spring 容器管理，所以也不会给它里边注入 Bean。

在实际开发中，我们可能会遇到一些类，需要获取到容器的详细信息，那就可以通过 Aware 接口来实现。

Aware 是一个空接口，有很多实现类：





这些实现的接口，有一些公共特性：

1. 都是以 Aware 结尾
2. 都继承自 Aware
3. 接口内均定义了一个 set 方法

每一个子接口均提供了一个 set 方法，方法的参数就是当前 Bean 需要感知的内容，因此我们需要在 Bean 中声明相关的成员变量来接受这个参数。接收到这个参数后，就可以通过这个参数获取到容器的详细信息了。

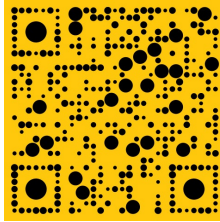
```
@Component
public class SayHello implements ApplicationContextAware {
    private ApplicationContext applicationContext;
    public String sayHello(String name) {
        //判断容器中是否存在某个 Bean
        boolean userDao = applicationContext.containsBean("userDao333");
        System.out.println(userDao);
        return "hello " + name;
    }
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

5.1 Aop

Aop (Aspect Oriented Programming)，面向切面编程，这是对面向对象思想的一种补充。

面向切面编程，就是在程序运行时，不改变程序源码的情况下，动态的增强方法的功能，常见的使用场景非常多：

1. 日志



2. 事务
3. 数据库操作
4.

这些操作中，无一例外，都有很多模板化的代码，而解决模板化代码，消除臃肿就是 Aop 的强项。

在 Aop 中，有几个常见的概念：

概念	说明
切点	要添加代码的地方，称作切点
通知（增强）	通知就是向切点动态添加的代码
切面	切点+通知
连接点	切点的定义

5.1.1 Aop 的实现

在 Aop 实际上集基于 Java 动态代理来实现的。

Java 中的动态代理有两种实现方式：

- cglib
- jdk

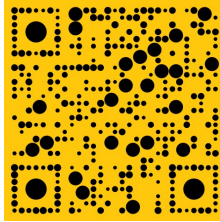
5.2 动态代理

基于 JDK 的动态代理。

1.定义一个计算器接口：

```
public interface MyCalculator {  
    int add(int a, int b);  
}
```

2.定义计算机接口的实现：



```
public class MyCalculatorImpl implements MyCalculator {
    public int add(int a, int b) {
        return a+b;
    }
}
```

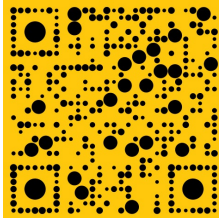
3.定义代理类

```
public class CalculatorProxy {
    public static Object getInstance(final MyCalculatorImpl myCalculator) {
        return Proxy.newProxyInstance(CalculatorProxy.class.getClassLoader(), myCalculator.getClass().getInterfaces(), new InvocationHandler() {
            /**
             * @param proxy 代理对象
             * @param method 代理的方法
             * @param args 方法的参数
             * @return
             * @throws Throwable
             */
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                System.out.println(method.getName()+"方法开始执行啦...");

                Object invoke = method.invoke(myCalculator, args);
                System.out.println(method.getName()+"方法执行结束啦...");

                return invoke;
            }
        });
    }
}
```

Proxy.newProxyInstance 方法接收三个参数，第一个是一个 classloader，第二个是代理多项实现的接口，第三个是代理对象方法的处理器，所有要额外添加的行为都在 invoke 方法中实现。



5.3 五种通知

Spring 中的 Aop 的通知类型有 5 种：

- 前置通知
- 后置通知
- 异常通知
- 返回通知
- 环绕通知

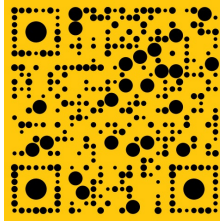
具体实现，这里的案例和 5.2 中的一样，依然是给计算器的方法增强功能。

首先，在项目中，引入 Spring 依赖（这次需要引入 Aop 相关的依赖）：

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
  </dependency>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.9.5</version>
  </dependency>
</dependencies>
```

接下来，定义切点，这里介绍两种切点的定义方式：

- 使用自定义注解
- 使用规则



其中，使用自定义注解标记切点，是侵入式的，所以这种方式在实际开发中不推荐，仅作为了解，另一种使用规则来定义切点的方式，无侵入，一般推荐使用这种方式。

自定义注解

首先自定义一个注解：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Action {
}
```

然后在需要拦截的方法上，添加该注解，在 add 方法上添加了 @Action 注解，表示该方法将会被 Aop 拦截，而其他未添加该注解的方法则不受影响。

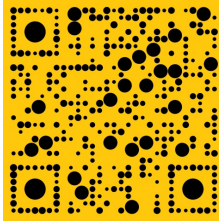
```
@Component
public class MyCalculatorImpl {
    @Action
    public int add(int a, int b) {
        return a + b;
    }

    public void min(int a, int b) {
        System.out.println(a + "-" + b + "=" + (a - b));
    }
}
```

接下来，定义增强（通知、Advice）：

```
@Component
@Aspect//表示这是一个切面
public class LogAspect {

    /**
     * @param joinPoint 包含了目标方法的关键信息
     * @Before 注解表示这是一个前置通知，即在目标方法执行之前执行，注解
     中，需要填入切点
```

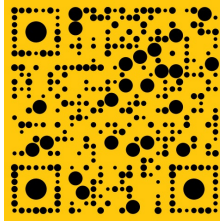


```
*/
@Before(value = "@annotation(Action)")
public void before(JoinPoint joinPoint) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法开始执行了...");
}

/**
 * 后置通知
 * @param joinPoint 包含了目标方法的所有关键信息
 * @After 表示这是一个后置通知，即在目标方法执行之后执行
 */
@After("@annotation(Action)")
public void after(JoinPoint joinPoint) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法执行结束了...");
}

/**
 * @param joinPoint
 * @AfterReturning 表示这是一个返回通知，即有目标方法有返回值的时候
才会触发，该注解中的 returning 属性表示目标方法返回值的变量名，这个需要和
参数一一对应吗，注意：目标方法的返回值类型要和这里方法返回值参数的类型一
致，否则拦截不到，如果想拦截所有（包括返回值为 void），则方法返回值参数可以
为 Object
 */
@AfterReturning(value = "@annotation(Action)", returning = "r")
public void returning(JoinPoint joinPoint, Integer r) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法返回: "+r);
}

/**
 * 异常通知
 * @param joinPoint
 * @param e 目标方法所抛出的异常，注意，这个参数必须是目标方法所抛出的
异常或者所抛出的异常的父类，只有这样，才会捕获。如果想拦截所有，参数类型声
明为 Exception
 */
@AfterThrowing(value = "@annotation(Action)", throwing = "e")
public void afterThrowing(JoinPoint joinPoint, Exception e) {
```

```
Signature signature = joinPoint.getSignature();
String name = signature.getName();
System.out.println(name + "方法抛异常了: "+e.getMessage());
}

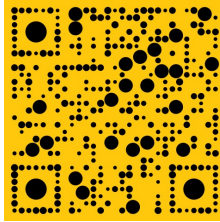
/**
 * 环绕通知
 *
 * 环绕通知是集大成者，可以用环绕通知实现上面的四个通知，这个方法的核心
有点类似于在这里通过反射执行方法
 * @param pjp
 * @return 注意这里的返回值类型最好是 Object，和拦截到的方法相匹配
 */
@Around("@annotation(Action)")
public Object around(ProceedingJoinPoint pjp) {
    Object proceed = null;
    try {
        //这个相当于 method.invoke 方法，我们可以在这个方法的前后分别
添加日志，就相当于前置/后置通知
        proceed = pjp.proceed();
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
    return proceed;
}
}
```

通知定义完成后，接下来在配置类中，开启包扫描和自动代理：

```
@Configuration
@ComponentScan
@EnableAspectJAutoProxy//开启自动代理
public class JavaConfig {
}
```

然后，在 Main 方法中，开启调用：

```
public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new AnnotationConfig
ApplicationContext(JavaConfig.class);
        MyCalculatorImpl myCalculator = ctx.getBean(MyCalculatorImpl.
class);
    }
}
```



```
        myCalculator.add(3, 4);
        myCalculator.min(3, 4);
    }
}
```

再来回顾 LogAspect 切面，我们发现，切点的定义不够灵活，之前的切点是直接写在注解里边的，这样，如果要修改切点，每个方法上都要修改，因此，我们可以将切点统一定义，然后统一调用。

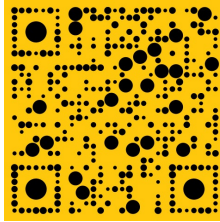
```
@Component
@Aspect//表示这是一个切面
public class LogAspect {

    /**
     * 可以统一定义切点
     */
    @Pointcut("@annotation(Action)")
    public void pointcut() {

    }

    /**
     * @param joinPoint 包含了目标方法的关键信息
     * @Before 注解表示这是一个前置通知，即在目标方法执行之前执行，注解
    中，需要填入切点
     */
    @Before(value = "pointcut()")
    public void before(JoinPoint joinPoint) {
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println(name + "方法开始执行了...");
    }

    /**
     * 后置通知
     * @param joinPoint 包含了目标方法的所有关键信息
     * @After 表示这是一个后置通知，即在目标方法执行之后执行
     */
    @After("pointcut()")
    public void after(JoinPoint joinPoint) {
        Signature signature = joinPoint.getSignature();
    }
}
```



```
String name = signature.getName();
System.out.println(name + "方法执行结束了...");
}
```

```
/**
 * @param joinPoint
 * @AfterReturning 表示这是一个返回通知，即有目标方法有返回值的时候
才会触发，该注解中的 returning 属性表示目标方法返回值的变量名，这个需要和
参数一一对应吗，注意：目标方法的返回值类型要和这里方法返回值参数的类型一
致，否则拦截不到，如果想拦截所有（包括返回值为 void），则方法返回值参数可以
为 Object
 */
```

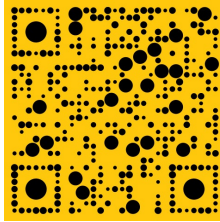
```
@AfterReturning(value = "pointcut()", returning = "r")
public void returning(JoinPoint joinPoint, Integer r) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法返回: "+r);
}
```

```
/**
 * 异常通知
 * @param joinPoint
 * @param e 目标方法所抛出的异常，注意，这个参数必须是目标方法所抛出的
异常或者所抛出的异常的父类，只有这样，才会捕获。如果想拦截所有，参数类型声
明为 Exception
 */
```

```
@AfterThrowing(value = "pointcut()", throwing = "e")
public void afterThrowing(JoinPoint joinPoint, Exception e) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法抛异常了: "+e.getMessage());
}
```

```
/**
 * 环绕通知
 *
 * 环绕通知是集大成者，可以用环绕通知实现上面的四个通知，这个方法的核心
有点类似于在这里通过反射执行方法
 */
```

```
* @param pjp
 * @return 注意这里的返回值类型最好是 Object，和拦截到的方法相匹配
 */
@Around("pointcut()")
public Object around(ProceedingJoinPoint pjp) {
    Object proceed = null;
```



```
try {
    //这个相当于 method.invoke 方法，我们可以在这个方法的前后分别
    添加日志，就相当于是前置/后置通知
    proceed = pjp.proceed();
} catch (Throwable throwable) {
    throwable.printStackTrace();
}
return proceed;
}
```

但是，大家也注意到，使用注解是侵入式的，我们还可以继续优化，改为非侵入式的。重新定义切点，新切点的定义就不需要 `@Action` 注解了，要拦截的目标方法上也不用添加 `@Action` 注解。下面这种方式是更为通用的拦截方式：

```
@Component
@Aspect//表示这是一个切面
public class LogAspect {

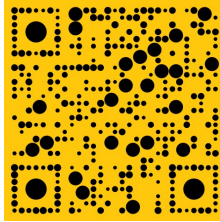
    /**
     * 可以统一定义切点
     */
    @Pointcut("@annotation(Action)")
    public void pointcut2() {

    }

    /**
     * 可以统一定义切点
     * 第一个 * 表示要拦截的目标方法返回值任意（也可以明确指定返回值类型
     * 第二个 * 表示包中的任意类（也可以明确指定类
     * 第三个 * 表示类中的任意方法
     * 最后面的两个点表示方法参数任意，个数任意，类型任意
     */
    @Pointcut("execution(* org.javaboy.aop.commons.*.*(..))")
    public void pointcut() {

    }

    /**
```



* @param joinPoint 包含了目标方法的关键信息
* @Before 注解表示这是一个前置通知，即在目标方法执行之前执行，注解中，需要填入切点

*/

@Before(value = "pointcut()")

```
public void before(JoinPoint joinPoint) {  
    Signature signature = joinPoint.getSignature();  
    String name = signature.getName();  
    System.out.println(name + "方法开始执行了...");  
}
```

/**

* 后置通知

* @param joinPoint 包含了目标方法的所有关键信息

* @After 表示这是一个后置通知，即在目标方法执行之后执行

*/

@After("pointcut()")

```
public void after(JoinPoint joinPoint) {  
    Signature signature = joinPoint.getSignature();  
    String name = signature.getName();  
    System.out.println(name + "方法执行结束了...");  
}
```

/**

* @param joinPoint

* @AfterReturning 表示这是一个返回通知，即有目标方法有返回值的时候才会触发，该注解中的 `returning` 属性表示目标方法返回值的变量名，这个需要和参数一一对应吗，注意：目标方法的返回值类型要和这里方法返回值参数的类型一致，否则拦截不到，如果想拦截所有（包括返回值为 `void`），则方法返回值参数可以为 `Object`

*/

@AfterReturning(value = "pointcut()", returning = "r")

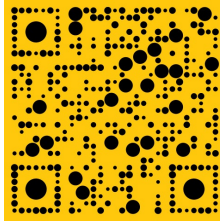
```
public void returning(JoinPoint joinPoint, Integer r) {  
    Signature signature = joinPoint.getSignature();  
    String name = signature.getName();  
    System.out.println(name + "方法返回: "+r);  
}
```

/**

* 异常通知

* @param joinPoint

* @param e 目标方法所抛出的异常，注意，这个参数必须是目标方法所抛出的异常或者所抛出的异常的父类，只有这样，才会捕获。如果想拦截所有，参数类型声



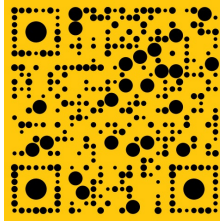
明为 Exception

```
*/
@AfterThrowing(value = "pointcut()",throwing = "e")
public void afterThrowing(JoinPoint joinPoint,Exception e) {
    Signature signature = joinPoint.getSignature();
    String name = signature.getName();
    System.out.println(name + "方法抛异常了: "+e.getMessage());
}

/**
 * 环绕通知
 *
 * 环绕通知是集大成者，可以用环绕通知实现上面的四个通知，这个方法的核心
有点类似于在这里通过反射执行方法
 * @param pjp
 * @return 注意这里的返回值类型最好是 Object ， 和拦截到的方法相匹配
 */
@Around("pointcut()")
public Object around(ProceedingJoinPoint pjp) {
    Object proceed = null;
    try {
        //这个相当于 method.invoke 方法，我们可以在这个方法的前后分别
添加日志，就相当于是前置/后置通知
        proceed = pjp.proceed();
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
    return proceed;
}
}
```

5.4 XML 配置 Aop

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>
<dependency>
```



```
<groupId>org.aspectj</groupId>
<artifactId>aspectjrt</artifactId>
<version>1.9.5</version>
</dependency>
```

接下来，定义通知/增强，但是单纯定义自己的行为即可，不再需要注解：

```
public class LogAspect {

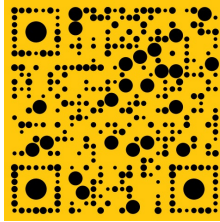
    public void before(JoinPoint joinPoint) {
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println(name + "方法开始执行了...");
    }

    public void after(JoinPoint joinPoint) {
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println(name + "方法执行结束了...");
    }

    public void returning(JoinPoint joinPoint, Integer r) {
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println(name + "方法返回: "+r);
    }

    public void afterThrowing(JoinPoint joinPoint, Exception e) {
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println(name + "方法抛异常了: "+e.getMessage());
    }

    public Object around(ProceedingJoinPoint pjp) {
        Object proceed = null;
        try {
            //这个相当于 method.invoke 方法，我们可以在这个方法的前后分别
            添加日志，就相当于前置/后置通知
            proceed = pjp.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        return proceed;
    }
}
```



```
}  
}
```

接下来在 spring 中配置 Aop:

```
<bean class="org.javaboy.aop.LogAspect" id="logAspect"/>  
<aop:config>  
  <aop:pointcut id="pc1" expression="execution(* org.javaboy.aop.co  
mmons.*.*(..))"/>  
  <aop:aspect ref="logAspect">  
    <aop:before method="before" pointcut-ref="pc1"/>  
    <aop:after method="after" pointcut-ref="pc1"/>  
    <aop:after-returning method="returing" pointcut-ref="pc1" ret  
urning="r"/>  
    <aop:after-throwing method="afterThrowing" pointcut-ref="pc1"  
throwing="e"/>  
    <aop:around method="around" pointcut-ref="pc1"/>  
  </aop:aspect>  
</aop:config>
```

最后, 在 Main 方法中加载配置文件:

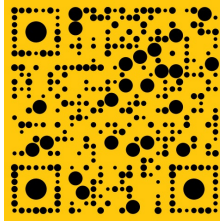
```
public class Main {  
  public static void main(String[] args) {  
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicat  
ionContext("applicationContext.xml");  
    MyCalculatorImpl myCalculator = ctx.getBean(MyCalculatorImpl.  
class);  
    myCalculator.add(3, 4);  
    myCalculator.min(5, 6);  
  }  
}
```

6. JdbcTemplate

JdbcTemplate 是 Spring 利用 Aop 思想封装的 JDBC 操作工具。

6.1 准备

创建一个新项目, 添加如下依赖:



```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.17</version>
  </dependency>
</dependencies>
```

准备数据库:

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/ `test01` /*!40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;
```

```
USE `test01`;
```

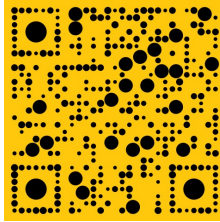
```
/*Table structure for table `user` */
```

```
DROP TABLE IF EXISTS `user`;
```

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `address` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

准备一个实体类:

```
public class User {
  private Integer id;
  private String username;
  private String address;
```



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

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;
}

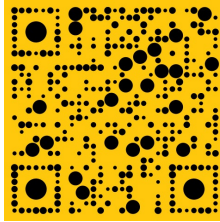
public String getAddress() {
    return address;
}

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

6.2 Java 配置

提供一个配置类，在配置类中配置 JdbcTemplate：

```
@Configuration
public class JdbcConfig {
    @Bean
    DataSource dataSource() {
```



```
DriverManagerDataSource dataSource = new DriverManagerDataSou
rce();
dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
dataSource.setUsername("root");
dataSource.setPassword("123");
dataSource.setUrl("jdbc:mysql:///test01");
return dataSource;
}
@Bean
JdbcTemplate jdbcTemplate() {
    return new JdbcTemplate(dataSource());
}
}
```

这里，提供两个 Bean，一个是 DataSource 的 Bean，另一个是

JdbcTemplate 的 Bean，JdbcTemplate 的配置非常容易，只需要 new 一个 Bean 出来，然后配置一下 DataSource 就可以。

```
public class Main {

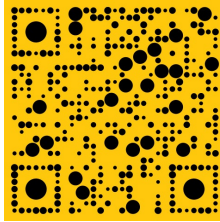
    private JdbcTemplate jdbcTemplate;

    @Before
    public void before() {
        AnnotationConfigApplicationContext ctx = new AnnotationConfig
        ApplicationContext(JdbcConfig.class);
        jdbcTemplate = ctx.getBean(JdbcTemplate.class);
    }

    @Test
    public void insert() {
        jdbcTemplate.update("insert into user (username,address) valu
        es (?,?);", "javaboy", "www.javaboy.org");
    }

    @Test
    public void update() {
        jdbcTemplate.update("update user set username=? where id=?",
        "javaboy123", 1);
    }

    @Test
```



```
public void delete() {
    jdbcTemplate.update("delete from user where id=?", 2);
}

@Test
public void select() {
    User user = jdbcTemplate.queryForObject("select * from user where id=?", new BeanPropertyRowMapper<User>(User.class), 1);
    System.out.println(user);
}
}
```

在查询时，如果使用了 BeanPropertyRowMapper，要求查出来的字段必须和 Bean 的属性名一一对应。如果不一样，则不要使用

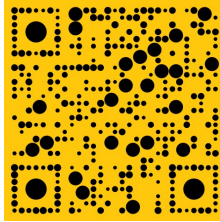
BeanPropertyRowMapper，此时需要自定义 RowMapper 或者给查询的字段取别名。

1. 给查询出来的列取别名：

```
@Test
public void select2() {
    User user = jdbcTemplate.queryForObject("select id,username as name,address from user where id=?", new BeanPropertyRowMapper<User>(User.class), 1);
    System.out.println(user);
}
```

2.自定义 RowMapper

```
@Test
public void select3() {
    User user = jdbcTemplate.queryForObject("select * from user where id=?", new RowMapper<User>() {
        public User mapRow(ResultSet resultSet, int i) throws SQLException {
            int id = resultSet.getInt("id");
            String username = resultSet.getString("username");
            String address = resultSet.getString("address");
            User u = new User();
            u.setId(id);
        }
    });
}
```



```
        u.setName(username);
        u.setAddress(address);
        return u;
    }
}, 1);
System.out.println(user);
}
```

查询多条记录，方式如下：

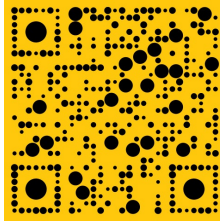
```
@Test
public void select4() {
    List<User> list = jdbcTemplate.query("select * from user", new BeanPropertyRowMapper<>(User.class));
    System.out.println(list);
}
```

6.3 XML 配置

以上配置，也可以通过 XML 文件来实现。通过 XML 文件实现只是提供 JdbcTemplate 实例，剩下的代码还是 Java 代码，就是 JdbcConfig 被 XML 文件代替而已。

```
<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource" id="dataSource">
    <property name="username" value="root"/>
    <property name="password" value="123"/>
    <property name="url" value="jdbc:mysql:///test01?serverTimezone=Asia/Shanghai"/>
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
</bean>
<bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

配置完成后，加载该配置文件，并启动：



```
public class Main {

    private JdbcTemplate jdbcTemplate;

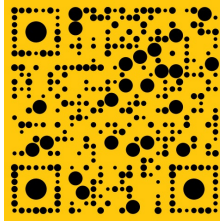
    @Before
    public void before() {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication
ionContext("applicationContext.xml");
        jdbcTemplate = ctx.getBean(JdbcTemplate.class);
    }

    @Test
    public void insert() {
        jdbcTemplate.update("insert into user (username,address) valu
es (?,?);", "javaboy", "www.javaboy.org");
    }
    @Test
    public void update() {
        jdbcTemplate.update("update user set username=? where id=?",
"javaboy123", 1);

    }
    @Test
    public void delete() {
        jdbcTemplate.update("delete from user where id=?", 2);
    }

    @Test
    public void select() {
        User user = jdbcTemplate.queryForObject("select * from user wh
ere id=?", new BeanPropertyRowMapper<User>(User.class), 1);
        System.out.println(user);
    }
    @Test
    public void select4() {
        List<User> list = jdbcTemplate.query("select * from user", new
BeanPropertyRowMapper<>(User.class));
        System.out.println(list);
    }

    @Test
    public void select2() {
        User user = jdbcTemplate.queryForObject("select id,username a
s name,address from user where id=?", new BeanPropertyRowMapper<User>
```



```
(User.class), 1);
    System.out.println(user);
}

@Test
public void select3() {
    User user = jdbcTemplate.queryForObject("select * from user where id=?", new RowMapper<User>() {
        public User mapRow(ResultSet resultSet, int i) throws SQLException {
            int id = resultSet.getInt("id");
            String username = resultSet.getString("username");
            String address = resultSet.getString("address");
            User u = new User();
            u.setId(id);
            u.setName(username);
            u.setAddress(address);
            return u;
        }
    }, 1);
    System.out.println(user);
}
}
```

7. 事务

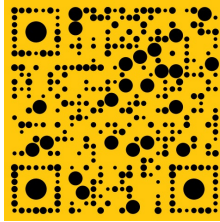
Spring 中的事务主要是利用 Aop 思想，简化事务的配置，可以通过 Java 配置也可以通过 XML 配置。

准备工作：

我们通过一个转账操作来看下 Spring 中的事务配置。

首先准备 SQL：

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/`test01` /*!40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;
```



```
USE `test01`;
```

```
/*Table structure for table `account` */
```

```
DROP TABLE IF EXISTS `account`;
```

```
CREATE TABLE `account` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,  
  `money` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

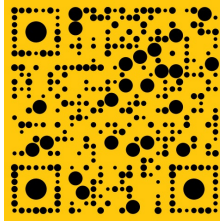
```
/*Data for the table `account` */
```

```
insert into `account`(`id`,`username`,`money`) values (1,'zhangsan',  
1000),(2,'lisi',1000);
```

然后配置 JdbcTemplate ， JdbcTemplate 的配置和第 6 小节一致。

然后，提供转账操作的方法：

```
@Repository  
public class UserDao {  
    @Autowired  
    JdbcTemplate jdbcTemplate;  
  
    public void addMoney(String username, Integer money) {  
        jdbcTemplate.update("update account set money=money+? where u  
sername=?", money, username);  
    }  
  
    public void minMoney(String username, Integer money) {  
        jdbcTemplate.update("update account set money=money-? where u  
sername=?", money, username);  
    }  
}  
  
@Service  
public class UserService {  
    @Autowired  
    UserDao userDao;  
    public void updateMoney() {
```

```
        userDao.addMoney("zhangsan", 200);
        int i = 1 / 0;
        userDao.minMoney("lisi", 200);
    }
}
```

最后，在 XML 文件中，开启自动化扫描：

```
<context:component-scan base-package="org.javaboy"/>
<bean class="org.springframework.jdbc.datasource.DriverManagerDataSo
urce" id="dataSource">
    <property name="username" value="root"/>
    <property name="password" value="123"/>
    <property name="url" value="jdbc:mysql:///test01?serverTimezone=A
sia/Shanghai"/>
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"
/>
</bean>
<bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTem
plate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

7.1 XML 配置

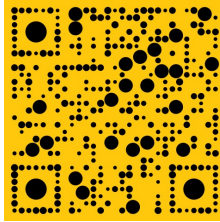
XML 中配置事务一共分为三个步骤：

1.配置 TransactionManager

```
<bean class="org.springframework.jdbc.datasource.DataSourceTransacti
onManager" id="transactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

2.配置事务要处理的方法

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="update*" />
        <tx:method name="insert*" />
        <tx:method name="add*" />
        <tx:method name="delete*" />
    </tx:attributes>
</tx:advice>
```



```
</tx:attributes>
</tx:advice>
```

注意，一旦配置了方法名称规则之后，service 中的方法一定要按照这里的名称规则来，否则事务配置不会生效

3.配置 Aop

```
<aop:config>
  <aop:pointcut id="pc1" expression="execution(* org.javaboy.servic
e.*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="pc1"/>
</aop:config>
```

4.测试

```
@Before
public void before() {
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationC
ontext("applicationContext.xml");
    jdbcTemplate = ctx.getBean(JdbcTemplate.class);
    userService = ctx.getBean(UserService.class);
}
@Test
public void test1() {
    userService.updateMoney();
}
```

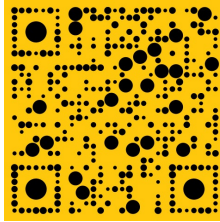
7.2 Java 配置

如果要开启 Java 注解配置，在 XML 配置中添加如下配置：

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

这行配置，可以代替下面两个配置：

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="update*" />
    <tx:method name="insert*" />
    <tx:method name="add*" />
```



```
<tx:method name="delete*" />
</tx:attributes>
</tx:advice>
<aop:config>
  <aop:pointcut id="pc1" expression="execution(* org.javaboy.servic
e.*.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="pc1" />
</aop:config>
```

然后，在需要添加事务的方法上，添加 `@Transactional` 注解，表示该方法开启事务，当然，这个注解也可以放在类上，表示这个类中的所有方法都开启事务。

```
@Service
public class UserService {
    @Autowired
    UserDao userDao;
    @Transactional
    public void updateMoney() {
        userDao.addMoney("zhangsan", 200);
        int i = 1 / 0;
        userDao.minMoney("lisi", 200);
    }
}
```