

Spring框架

Spring框架

- 课程介绍

 - 课程内容

 - 框架概述

 - Spring入门

- IOC容器

 - 概念和原理

 - 什么是IOC

 - IOC底层原理

 - IOC (BeanFactory接口)

 - IOC操作Bean管理

 - 概念

 - 基于xml

 - FactoryBean

 - Bean作用域

 - Bean生命周期

 - xml自动装配

 - 外部属性文件

 - 基于注解

- AOP

 - 概念

 - 底层原理

 - JDK动态代理 (Spring底层)

 - 术语

 - AOP操作

 - 准备工作

 - AspectJ 注解

 - AspectJ配置文件

- JdbcTemplate

 - 概念和准备

 - 什么是JdbcTemplate

 - 准备工作

 - 操作数据库

 - 添加 update(String sql,Object... args)

 - 修改和删除 update(String sql,Object... args)

 - 查询返回某个值 queryForObject(String sql,Class requiredType)

 - 查询返回对象 queryForObject(String sql,RowMapper rowMapper,Object ... args)

 - 查询返回集合 query(String sql,RowMapper rowMapper,Object... args)

 - 批量操作 batchUpdate(String sql,List<Object[]> batchArgs)

- 事务操作

 - 事务概念

 - 搭建事务操作环境

 - Spring事务管理介绍

 - 管理介绍

注解声明式事务管理
声明式事务管理参数配置
XML声明式事务管理
完全注解声明式事务管理

Spring5新特性

日志封装
Spring5 框架核心容器支持@Nullable 注解
Spring5 核心容器支持函数式风格 GenericApplicationContext
Spring5 支持整合 JUnit5
Spring5 框架新功能 (Webflux)
 SpringWebflux 介绍
 响应式编程 (Java 实现)
 响应式编程 (Reactor 实现)
 SpringWebflux 执行流程和核心 API
 SpringWebflux (基于注解编程模型)
 SpringWebflux (基于函数式编程模型)

课程介绍

课程内容

1. Spring概念
2. IOC容器
 1. IOC底层原理
 2. IOC接口 (BeanFactory)
 3. IOC操作Bean管理 (基于xml)
 4. IOC操作Bean管理 (基于注解)
3. Aop
4. JdbcTemplate
5. 事务管理
6. Spring5新特性

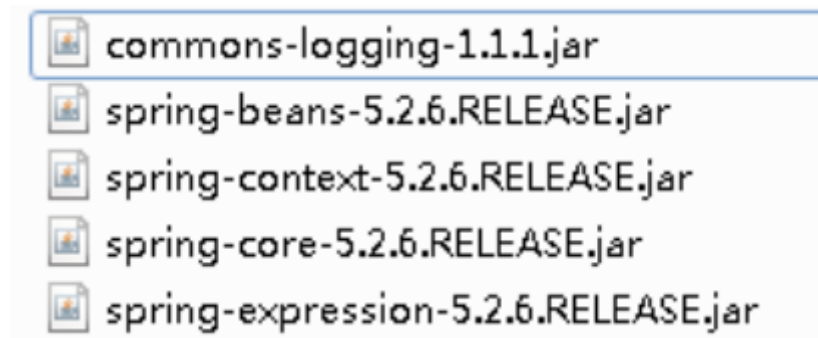
框架概述

- Spring是轻量级的开源的JavaEE框架
- Spring可以解决企业应用开发的复杂性
- Spring有两个核心部分：IOC和Aop
 - IOC：控制反转，把创建对象过程交给Spring进行管理
 - Aop：面向切面，不修改源代码进行功能增强
- Spring特点
 - 方便解耦，简化开发
 - Aop编程支持

- 方便程序测试
- 方便进行事务操作
- 降低API开发难度

Spring入门

- 下载Spring5
 - <https://repo.spring.io/ui/native/release/org/springframework/spring/>
 - 选择版本5.X
- 打开idea工具，创建普通Java工程
- 导入Spring相关jar包 (commons-logging、spring-beans、spring-context、spring-core、spring-expression)



- 创建普通类，在这个类中创建普通方法
- 创建Spring配置文件，在配置文件中配置创建的对象，创建xml文件
- 测试
 - 加载Spring配置文件 (ApplicationContext)
 - 获取配置创建的对象 (context.getBean)
- 默认配置文件头部信息

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:util="http://www.springframework.org/schema/util"

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

IOC容器

概念和原理

什么是IOC

1. 控制反转，把对象创建和对象之间的调用过程，交给Spring进行管理
2. 使用IOC目的，为了耦合度降低
3. 做入门案例就是IOC的实现

IOC底层原理

- xml解析，工厂模式，反射
- 图解
 - 第一步 xml配置文件，配置创建的对象

```
<bean id='dao' class='com.atguigu.UserDao'></bean>
```

- 第二步 有service类和dao类，创建工厂类

```
class UserFactory{
    public static UserDao getDao(){
        //xml解析
        String classValue = class属性值;
        //通过反射创建对象
        Class clazz = Class.forName(classValue);
        return (UserDao)clazz.newInstance();
    }
}
```

IOC (BeanFactory接口)

1. IOC思想基于IOC容器完成，IOC容器底层就是对象工厂
2. Spring提供IOC容器实现两种方式：（两个接口）
 1. *BeanFactory*: IOC容器基本实现，是Spring内部的使用接口，不提供给开发人员进行使用，**加载配置文件时不会创建对象，在获取对象时采取创建对象**
 2. *ApplicationContext*: *BeanFactory*接口的子接口，提供更多更强大的功能，一般由开发人员进行使用，**加载配置文件时候就会把配置文件中的对象创建好**
3. *ApplicationContext*接口有实现类
 1. *FileSystemXmlApplicationContext*: 配置文件路径写为在系统盘符中所在的位置，相当于文件系统

2. ClassPathXmlApplicationContext: 配置文件路径写为src下的路径，即基于src书写路径

IOC操作Bean管理

概念

- 什么是Bean管理
 - Bean管理指的是两个操作
 - Spring创建对象
 - Spring注入属性
- Bean管理的两种方式
 - 基于xml配置文件方式实现
 - 基于注解方式实现

基于xml

1. 基于xml方式创建对象

```
<bean id='user' class='com.atguigu.spring.User'></bean>
```

1. 在Spring配置文件中，使用bean标签，标签里面添加对应的属性，就可以实现对象创建
2. 在bean标签里面有很多属性
 1. id属性：唯一标识
 2. class属性：类全路径（包类路径）
 3. name属性：和id属性作用一致，但是可以用符号开头
3. 创建对象时候，默认也是执行**无参数构造器**完成对象创建（bean类需要提供无参构造器）

2. 基于xml方式注入属性

1. DI：依赖注入，就是注入属性
3. **第一种注入方式**：使用set方法进行注入
 1. 创建类，定义属性和对应的set方法

```
public class Book{  
    private String bname;  
    public void setBname(String bname){  
        this.bname = bname;  
    }  
}
```

2. 在Spring配置文件中配置对象创建，配置属性注入 (property)

```
<bean id='book' class='com.atguigu.Book'>
    <property name='bname' value='《javaweb》'>
</property>
    <!--可以写多个属性-->
</bean>
```

4. 第二种注入方式：使用有参数构造器进行注入

1. 创建类，定义属性，创建属性对应参数构造器

```
public class Orders{
    private String oname;
    public Orders(String oname){
        this.oname = oname;
    }
}
```

2. 在Spring配置文件中进行配置 (constructor-arg)

```
<bean id="orders" class="com.atguigu.spring5.Orders">
    <constructor-arg name='oname' value="电脑">
</constructor-arg>
    <!--可以写多个属性-->
</bean>
```

5. p名称空间注入

1. 使用p名称空间注入，可以简化基于xml配置方式
2. 第一步 添加p名称空间在配置文件中

```
<beans
xmlns:xsi="http://www.springframework.org/schema/p">
</beans>
```

3. 第二步 进行属性注入，在bean标签里面进行操作

```
<!--可以写多个 p:属性名-->
<bean id='book' class='com.atguigu.spring.Book'
p:bname="《javaweb》" ></bean>
```

6. xml注入其他类型属性

1. 字面量

1. null值

```
<property name='address'>
    <null/>
</property>
```

2. 属性值包含特殊符号

```
<!--属性值包含特殊符号（两种方式解决）
    1.把< >进行转义 &lt; &gt;
    2.把带特殊符号内容写到 CDATA 格式 <![CDATA[]]>
-->
<property name='address'>
    <value><![CDATA[<<南京>>]]></value>
</property>
```

7. 注入属性——外部bean（非bean类中调用bean）

1. 创建两个类 service类和dao类
2. 在service调用dao里面的方法
3. 在Spring配置文件中进行配置

```
public class UserService {
    //创建 UserDao 类型属性，生成 set 方法
    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    public void add() {
        System.out.println("service
add.....");
        userDao.update();
    }
}
```

```
<!--1 service 和 dao 对象创建-->
<bean id="userService"
    class="com.atguigu.spring5.service.UserService">
    <!--注入 userDao 对象
    name 属性：类里面属性名称
    ref 属性：创建 userDao 对象 bean 标签 id 值
-->
    <property name="userDao" ref="userDaoImpl">
</property>
</bean>
<bean id="userDaoImpl"
    class="com.atguigu.spring5.dao.UserDaoImpl">
</bean>
```

8. 注入属性——内部bean (**bean类中调用bean**)

1. 一对多关系：部门和员工
2. 在实体类之间表示一对多关系，员工表示所属部门，使用对象类型属性进行表示

```
//部门类
public class Dept {
    private String dname;
    public void setDname(String dname) {
        this.dname = dname;
    }
}

//员工类
public class Emp {
    private String ename;
    private String gender;
    //员工属于某一个部门，使用对象形式表示
    private Dept dept;
    public void setDept(Dept dept) {
        this.dept = dept;
    }
    public void setName(String ename) {
        this.ename = ename;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
}
```

3. 在Spring配置文件中进行配置

```
<!-- 内部 bean -->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!-- 设置两个普通属性 -->
    <property name="ename" value="lucy"></property>
    <property name="gender" value="女"></property>

    <!-- 设置对象类型属性 -->
    <property name="dept">
        <bean id="dept"
            class="com.atguigu.spring5.bean.Dept">
            <property name="dname" value="安保部">
            </property>
        </bean>
    </property>
</bean>
```


9. 注入属性——级联赋值

1. 第一种写法

```
<!--级联赋值-->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!--设置两个普通属性-->
    <property name="ename" value="lucy"></property>
    <property name="gender" value="女"></property>
    <!--级联赋值-->
    <property name="dept" ref="dept"></property>
</bean>
<bean id="dept" class="com.atguigu.spring5.bean.Dept">
    <property name="dname" value="财务部"></property>
</bean>
```

2. 第二种写法

```
//员工属于某一个部门，使用对象形式表示
private Dept dept;
public Dept getDept(){
    return dept;
}
public void setDept(Dept dept){
    this.dept = dept;
}
```

```
<!--级联赋值-->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!--设置两个普通属性-->
    <property name="ename" value="lucy"></property>
    <property name="gender" value="女"></property>
    <!--级联赋值-->
    <property name="dept" ref="dept"></property>
    <property name="dept.dname" value="技术部">
</property>
</bean>
<bean id="dept" class="com.atguigu.spring5.bean.Dept">
    <property name="dname" value="财务部"></property>
</bean>
```

10. 注入属性——集合属性

1. 注入数组类型属性、注入List集合类型属性、注入Map集合类型属性

1. 创建类，定义数组、list、map、set类型属性，生成对应的set方法

2. 在spring配置文件中配置

```
<!--1 集合类型属性注入-->
<bean id='stu' class='con.atguigu.spring.Stu'>
    <!--数组类型属性注入-->
    <property name='course'>
        <array>
            <value>java课程</value>
            <value>数据库课程</value>
        </array>
    </property>

    <!--List类型属性注入-->
    <property name='list'>
        <list>
            <value>张三</value>
            <value>李四</value>
        </list>
    </property>

    <!--map 类型属性注入-->
    <property name='maps'>
        <map>
            <entry key='JAVA' value='java'>
        </entry>
        </map>
    </property>

    <!--set 类型属性注入-->
    <property name='sets'>
        <set>
            <value>MySQL</value>
            <value>Redis</value>
        </set>
    </property>
</bean>
```

2. 在集合里面设置对象类型值

```
<!--创建多个 course 对象-->
<bean id="course1"

    class="com.atguigu.spring5.collectiontype.Course">
    <property name="cname" value="Spring5 框架">
</property>
</bean>
<bean id="course2"

    class="com.atguigu.spring5.collectiontype.Course">
```

```

        <property name="cname" value="MyBatis 框架">
    </property>
</bean>
<!--注入 list 集合类型，值是对象-->
<property name="courseList">
    <list>
        <ref bean="course1"></ref>
        <ref bean="course2"></ref>
    </list>
</property>

```

3. 把集合注入部分提取出来作为公共部分

1. 在spring配置文件中引入名称空间util

```

<?xml version="1.0" encoding="utf-8"?>
<beans
    xmlns:util="http://www.springframework.org/schem
a/util"
    xsi:schemaLocation="http://www.springframework.o
rg/schema/util
http://www.springframework.org/schema/util/sprin
g-util.xsd"></beans>

```

2. 使用util标签完成list集合注入提取

```

<util:list id="bookList">
    <value>Mysql</value>
    <value>JAVA</value>
</util:list>
<bean id="book" class="com.atguigu.spring.Book">
    <property name='list' ref='bookList'>
</property>
</bean>

```

FactoryBean

1. Spring有两种类型，一种是普通的bean，另一种是工厂bean (FactoryBean)
2. 普通bean：在配置文件中定义bean类型就是**返回类型**
3. 工厂bean：在配置文件中定义bean类型**可以和返回类型不一样**
 1. 第一步 创建类，让这个类作为工厂bean，实现**接口FactoryBean**
 2. 第二步 实现接口里面的方法，在实现的方法中定义返回的bean类型

```

public class MyBean implements FactoryBean<Course> {
    //定义返回 bean
    //定义返回类型
    @Override

```

```

    public Course getObject() throws Exception {
        Course course = new Course();
        course.setName("abc");
        //返回Course类型
        return course;
    }
    @Override
    public Class<?> getObjectType() {
        return null;
    }

    //是否为单例
    @Override
    public boolean isSingleton() {
        return false;
    }
}

```

```

<!--获取对象为Course类型-->
<bean id="myBean" class="com.atguigu.spring.MyBean">
</bean>

```

Bean作用域

1. 在Spring里面，设置创建bean实例是单实例还是多实例
2. 在Spring里面，默认情况下，bean是单实例对象

```

@Test
public void test(){
    ApplicationContext = context = new
    ClassPathXmlApplicationContext("bean2.xml");
    Book book1 = context.getBean("book",Book.class);
    Book book2 = context.getBean("book",Book.class);
    //输出两个对象的地址相同
    System.out.println("book1");
    System.out.println("book2");
}

```

3. 如何设置单实例还是多实例

1. 在Spring配置文件bean标签里面有属性 **scope** 用于设置单实例还是多实例

2. scope属性值

1. 第一个值 默认值， **singleton**表示单实例 （不同对象地址相同）
2. 第二个值 **prototype** 表示多实例 （不同对象地址不同）

- 3. request, session设置创建的对象放在哪个数据域
- 3. singleton 和 prototype 区别
 - 1. **singleton单实例, prototype多实例**
 - 2. 设置 scope 值是 **singleton** 时候, 加载 spring 配置文件时候就会创建单实例对象设置 scope 值是 prototype 时候, 不是在加载 spring 配置文件时候创建对象, 在调用getBean方法时候创建多实例对象

Bean生命周期

1. bean生命周期

- 1. 通过构造器创建bean实例 (无参构造器)
- 2. 为bean的属性设置值和对其他bean引用 (调用set方法)
- 3. 调用bean的初始化的方法 (需要进行配置初始化的方法)
 - 1. 创建初始化的方法initMethod()
 - 2. **配置文件bean标签 init-method='initMethod'**
- 4. bean可以使用了 (对象获取到了)
- 5. 当容器关闭的时候, 调用bean的销毁的方法 (需要进行配置销毁的方法 context.close())
 - 1. 创建销毁方法destoryMethod()
 - 2. **配置文件bean标签 destroy-method="destroyMethod"**

2. bean的后置处理器, bean生命周期有七步

- 1. 通过构造器创建bean实例 (无参构造器)
- 2. 为bean的属性设置值和对其他bean引用 (调用set方法)
- 3. **把bean实例传递bean后置处理器的方法**
postProcessBeforeInitialization
- 4. 调用bean的初始化的方法 (需要进行配置初始化的方法)
- 5. **把bean实例传递bean后置处理器的方法**
postProcessAfterInitialization
- 6. bean可以使用了 (对象获取到了)
- 7. 当容器关闭的时候, 调用bean的销毁的方法 (需要进行配置销毁的方法 context.close())

xml自动装配

1. 什么是自动装配

- 1. 根据指定转配规则 (属性名称或属性类型), Spring自动将匹配的属性值进行注入

2. 演示自动装配过程

- 1. 根据属性名称自动注入

```

<!--实现自动装配
    bean标签属性autowire，配置自动装配
    zutowire属性常用两个值：
        byName根据属性名称注入，注入值bean的id值和类属性名称
        一样
        byType根据属性类型注入
-->
<bean id='emp' class='com.atguigu.spring.Emp'
autowire='byName'>
    <!--<property name='dept' ref='dept'></property-->
</bean>
<bean id='dept' class='com.atguigu.spring.Dept'></bean>

```

2. 根据属性类型自动注入

```

<bean id="emp" class="com.atguigu.spring5.autowire.Emp"
autowire="byType">
    <!--<property name="dept" ref="dept"></property-->
</bean>
<bean id="dept"
class="com.atguigu.spring5.autowire.Dept">
</bean>

```

外部属性文件

1. 直接配置数据库信息

1. 配置德鲁伊连接池
2. 引入德鲁伊连接池依赖jar包 (druid-1.1.9.jar)

```

<!--直接配置连接池-->
<bean id='dataSource'
class='com.atguigu.spring,DruidDataSource'>
    <property name='driverClassName'
value='com.mysql.jdbc.Driver' ></property>
    <property name='url'
value='jdbc:mysql://localhost:3306/userDb'></property>
    <property name='username' value='root'></property>
    <property name='password' value='root'></property>
</bean>

```

2. 引入外部属性文件配置数据库连接池信息

1. 创建外部属性文件，**properties格式文件**，写数据库信息

```
prop.driverClass=com.mysql.jdbc.Driver
prop.url=jdbc:mysql://localhost:3306/userDb
prop.userName=root
prop.password=root
```

2. 把外部properties属性文件引入到spring配置文件中

1. 引入context名称空间

```
<beans
xmlns:context="http://www.springframework.org/sc
hema/context"
xsi:schemaLocation="http://www.springframework.o
rg/schema/context
http://www.springframework.org/schema/context/sp
ring-context.xsd"></beans>
```

2. 在spring配置文件只用标签引入外部属性文件，①使用 **context:property-placeholder** 的location引入，格式 classpath:文件名；②使用 **\${键名}** 引入属性值

```
<!--引入外部属性文件-->
<context:property-placeholder
location="classpath:jdbc.properties">
</context:property-placeholder>

<!--配置连接池-->
<bean id="dataSource"

class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName"
value="${prop.driverClass}">
    </property>
    <property name="url" value="${prop.url}">
    </property>
    <property name="username"
value="${prop.userName}">
    </property>
    <property name="password"
value="${prop.password}">
    </property>
</bean>
```

基于注解

1. 什么是注解

1. 注解是代码特殊标记，**格式：@注解名称(属性名=属性值, ……)**
2. 使用注解，注解作用在**类上面，方法上面，属性上面**
3. 使用注解目的：**简化xml配置**

2. Spring针对Bean管理中创建对象提供注解

1. @Component 普通 的注解，基本
2. @Service 业务逻辑层
3. @Controller web层
4. @Repository dao层，持久层
5. 注：以上四个注解**功能是一样的**，都可以用来创建Bean实例

3. 基于注解方式实现对象创建

1. 第一步：引入依赖 **spring-aop-5.2.6.RELEASE.jar**
2. 第二步：开启组件扫描

```
<!--开启组件扫描（扫描包中的类）
    1 引入名称空间 context
    2 如果扫描过个包，多个包使用 逗号 隔开
    3 扫描包上层目录
-->
<context:component-scan base-package='com.atguigu'>
</context:component-scan>
```

3. 第三步 创建类，在类上面添加创建对象注解

```
//在注解里面 value 属性值可以省略不写，直接@Component，其他注解同理
//如果不写value值，默认值是类名称，首字母小写
//比如 UserService -- userService

@Component(value='userService')    //相当于<bean
id='userService' class=''>
public class UserService{
    public void add(){
        System.out.println("service add……")
    }
}
```

4. 开启组件扫描细节配置


```

<!--示例 1
    use-default-filters="false" 表示现在不使用默认 filter（默认扫描包中所有类），自己配置 filter
    context:include-filter ， 设置扫描哪些内容
-->
<!--只扫描包中注解为Controller的类-->
<context:component-scan base-package="com.atguigu" use-defaultfilters="false">
    <context:include-filter type="annotation"

    expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

```

<!--示例 2
    下面配置扫描包所有内容
    context:exclude-filter: 设置哪些内容不进行扫描
-->
<!--不扫描包中注解为Controller的类-->
<context:component-scan base-package="com.atguigu">
    <context:exclude-filter type="annotation"

    expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

5. 基于注解方式实现属性注入

1. @Autowired：根据属性类型进行自动装配

1. 第一步 把service和dao对象创建，在service和dao（**如果为接口添加到实现类**）类添加创建对象注解
2. 第二步 在service注入dao对象，在service类添加dao类型属性，在属性上面使用注解

```

@Service
public class UserDao{

}
@Service
public class UserService{
    //定义dao类型属性
    //不需要添加set方法
    //添加注入属性注解
    @Autowired
    private UserDao userDao;
    public void add(){
        System.out.println("service add.....");
        userDao.add();
    }
}

```

```
}
```

2. @Qualifier: 根据名称自动装配

1. 使用方式与@Autowired一起使用

```
//定义 dao 类型属性  
//不需要添加 set 方法//添加注入属性注解  
@Autowired //根据类型进行注入  
@Qualifier(value = "userDaoImpl1") //根据名称进行  
注入  
private UserDao userDao;
```

3. @Resource: 可以根据类型注入, 可以根据名称注入 (不是spring内部的jar包, 属于javax的)

```
@Resource(name='userDaoImpl1')  
private UserDao userDao;
```

4. @Value: 注入普通类型属性

```
@Value(value='abc')  
private String name;
```

6. 完全注解开发

1. 创建配置类, 代替xml配置文件 @Configuration, @ComponentScan 扫描文件

```
@Configuration //作为配置类, 替代 xml 配置文件  
@ComponentScan(basePackages = {"com.atguigu"})  
public class SpringConfig {  
}
```

2. 编写测试类, 加载配置文件用AnnotationConfigApplicationContext 配置类.calss

```

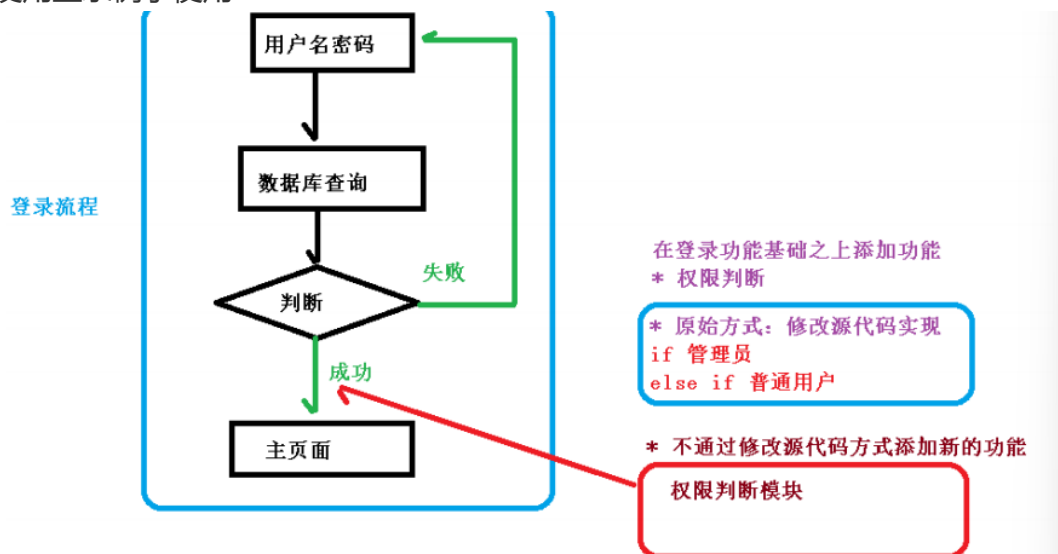
@Test
public void testService2() {
    //加载配置类
    ApplicationContext context = new
    AnnotationConfigApplicationContext(SpringConfig.class);
    UserService userService =
    context.getBean("userService", UserService.class);
    System.out.println(userService);
    userService.add();
}

```

AOP

概念

1. 面向切面编程（方面），利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可复用性，同时提高开发效率。
2. 通俗描述：**不通过修改源代码的方式，在主干功能猫添加新的功能。**
3. 使用登录例子使用AOP



底层原理

1. AOP底层使用动态代理
 1. 有两种情况使用动态代理
 2. 第一种 有接口的情况，使用JDK动态代理
 1. 创建接口实现类代理对象，增强类的方法

```
//代理对象
interface UserDao{
    public void login();
}
//实现类
class UserDaoImpl implements UserDao{
    public void login(){

    }
}
```

3. 第二种 没有接口的情况，使用CGLIB动态代理

1. 创建子类的代理对象，增强类的方法

```
//代理对象
class User{
    public void add(){

    }
}
//子类
class Person extends User{
    public void add(){
        super.add();
    }
}
```

JDK动态代理（Spring底层）

1. 使用JDK动态代理，使用Proxy类里面的方法创建代理对象

1. 调用newProxyInstance方法

2. 方法中有三个参数：

1. 第一参数：类加载器
2. 第二参数：增强方法所在的类，这个类实现的接口，支持多个接口
3. 第三参数：实现这个接口 InvocationHandler ，创建代理对象，写增强方法

2. 编写JDK动态代理代码

1. 创建接口，定义方法

```
public interface UserDao{
    public int add(int a,int b);
    public String update(String id);
}
```

2. 创建接口实现类，实现方法

```
public class UserDaoImpl implements UserDao{
    @Override
    public int add(int a,int b){
        return a+b;
    }
    @Override
    public String update(String id){
        return id;
    }
}
```

3. 使用Proxy类创建接口代理对象

```
public class JDKProxy {
    public static void main(String[] args) {
        //创建接口实现类代理对象
        Class[] interfaces = {UserDao.class};

        //Proxy.newProxyInstance(JDKProxy.class.getClassLoader
        (),interfaces,
            new InvocationHandler() {
                // @Override
                // public Object invoke(Object proxy,
                Method method, Object[] args) throws Throwable {
                    // return null;
                    // }
                    // });
                UserDaoImpl userDao = new UserDaoImpl();
                UserDao dao =

                (UserDao)Proxy.newProxyInstance(JDKProxy.class.getClass
                Loader(),

                interfaces,

                new
                UserDaoProxy(userDao));
                int result = dao.add(1, 2);
                System.out.println("result:"+result);
            }
    }
}
```

```

//创建代理对象代码
class UserDaoProxy implements InvocationHandler {
    //1 把创建的是谁的代理对象，把谁传递过来
    //有参数构造传递
    private Object obj;
    public UserDaoProxy(Object obj) {
        this.obj = obj;
    }
    //增强的逻辑
    @Override
    public Object invoke(Object proxy, Method method,
        Object[] args) throws
        Throwable {
        //方法之前
        System.out.println("方法之前执
行...."+method.getName()+" :传递的参数...."+
        Arrays.toString(args));
        //被增强的方法执行
        Object res = method.invoke(obj, args);
        //方法之后
        System.out.println("方法之后执行...."+obj);
        return res;
    }
}

```

术语

1. 连接点

1. 类里面哪些方法可以被增强，这些方法称为连接点

2. 切入点

1. 实际被真正增强的方法，称为切入点

3. 通知（增强）

1. 实际增强的逻辑部分称为通知（增强）

2. 通知有多种类型












1. 前置通知 @Before
2. 后置通知 @AfterReturning
3. 环绕通知 @After
4. 异常通知 @AfterThrowing
5. 最终通知 @Around

4. 切面

1. 是动作，把通知应用到切入点过程

AOP操作

准备工作

1. Spring框架一般都是基于**AspectJ**实现AOP操作
 1. AspectJ 不是 Spring 组成部分，独立AOP框架，一般把 AspectJ 和 Spring 框架一起使用，进行AOP操作
2. 基于AspectJ实现AOP操作
 1. 基于xml配置文件实现
 2. 基于注解方式实现（使用）
3. 引入AOP相关jar包【spring-aop、spring-aspects、com.springsource.net.sf.cglib、com.springsource.org.aopalliance、com.springsource.org.aspectj.weaver】
 - >  com.springsource.net.sf.cglib-2.2.0.jar
 - >  com.springsource.org.aopalliance-1.0.0.jar
 - >  com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
 - >  commons-logging-1.1.1.jar
 - >  druid-1.1.9.jar
 - >  spring-aop-5.2.6.RELEASE.jar
 - >  spring-aspects-5.2.6.RELEASE.jar
 - >  spring-beans-5.2.6.RELEASE.jar
 - >  spring-context-5.2.6.RELEASE.jar
 - >  spring-core-5.2.6.RELEASE.jar
 - >  spring-expression-5.2.6.RELEASE.jar
4. 切入点表达式
 1. 切入点表达式作用：知道对哪个类里面的哪个方法进行增强
 2. 语法结构：execution([权限修饰符] [返回类型] [类全路径] [方法名称] [参数列表])
 1. 对 com.atguigu.dao.BookDao类里面的add方法进行增强
execution(* com.atguigu.dao.BookDao.add (...))
 2. 对 com.atguigu.dao.BookDao类里面的所有方法进行增强
execution(* com.atguigu.dao.BookDao.* (...))
 3. 对 com.atguigu.dao 包里面所有类，类里面所有方法进行增强
execution(* com.atguigu.dao.. (...))

AspectJ 注解

1. 创建类，在类里面定义方法

```
public class User{
    public void add(){
        System.out.println("add.....");
    }
}
```

2. 创建增强类（编写增强逻辑）

1. 在增强类里面创建方法，让不同方法代表不同通知类型

```
//增强的类
public class UserProxy{
    public void before(){
        System.out.println("before.....");
    }
}
```

3. 进行通知的配置

1. 在spring配置文件中，开启注解扫描

```
<?xml version="1.0" encoding="UTF-8"?>

<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

    xmlns:context="http://www.springframework.org/schema/co
ntext"

    xmlns:aop="http://www.springframework.org/schema/aop"

    xsi:schemaLocation="http://www.springframework.org/sche
ma/beans

http://www.springframework.org/schema/beans/spring-
beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-
context.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-
aop.xsd">
    <!-- 开启注解扫描 -->
```



```

<context:component-scan
basepackage="com.atguigu.spring5.aopanno">
</context:component-scan>
</beans>

```

2. 使用注解创建User和UserProxy对象

```

//被增强的类
@Component
public class User{

}

//增强的类
@Component
public class UserProxy{

}

```

3. 在增强类上面添加注解@Aspect

```

//增强的类
@Component
@Aspect //生成代理对象
public class UserProxy{

}

```

4. 在spring配置文件中开启生成代理对象

```

<!-- 开启 Aspect 生成代理对象-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

```

4. 配置不同类型的通知

1. 在增强类的里面，再作为通知方法上面添加通知类注解，使用切入点表达式配置

```

//增强的类
@Component
@Aspect
public class UserProxy{

    //前置通知
    //@Before注解表示作为前置通知
    @Before(value="execution(*
com.atguigu.spring.aopanno.User.add(..))")
    public void before(){
        system.out.println("before.....");
    }

}

```

```

//后置通知（返回通知）
@AfterReturning(value="execution(*
com.atguigu.spring.anpanno.User.add(..))")
public void afterReturning(){
    System.out.println("afterReturning.....");
}

//最终通知
@After(value="execution(*
com.atguigu.spring.anpanno.User.add(..))")
public void after(){
    System.out.println("after.....");
}

//异常通知
@AfterThrowing(value="execution(*
com.atguigu.spring.User.add(..))")
public void afterThrowing(){
    System.out.println("afterThrowing.....");
}

//环绕通知
@Around(value="execution(*
com.atguigu.spring.User.add(..))")
public void around(ProceedingJoinPoint
proceedingJoinPoint) throws Throwable{
    System.out.println("环绕之前.....");
    //被增强的方法执行
    proceedingJoinPoint.proceed();
    System.out.println("环绕之后.....");
}
}

```

5. 相同的切入点抽取

```

//相同切入点抽取
@Pointcut(value="execution(*
com.atguigu.spring.User.add(..))")
public void pointdemo(){
    //前置通知
    //@Before注解表示作为前置通知
    @Before(value="pointdemo()")
    public void before(){
        System.out.println("before.....");
    }
}
}

```

6. 有多个增强类对同一个方法进行增强，设置增强类**优先级**

1. 在增强类上面添加注解**@Order(数字)**，数字类型值**越小优先级越高**

```
@Component
@Aspect
@Order(1)
public class PersonProxy{}
```

7. 完全注解开发

1. 创建配置类，不需要创建xml配置文件

```
@Configuration
@ComponentScan(basePackages={"com.atguigu"})
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class ConfigAop{}
```

AspctJ配置文件

1. 创建两个类，增强类和被增强类，创建相应的方法
2. 在spring配置文件中创建两个类对象

```
<!--创建对象-->
<bean id='book' class='com.atguigu.spring.Book'></bean>
<bean id='bookProxy' class='com.atguigu.spring.BookProxy'>
</bean>
```

3. 在spring配置文件中配置切入点

```
<!--配置aop增强-->
<aop:config>
    <!--切入点-->
    <aop:pointcut id='p' expression="execution(*
com.atguigu.spring.Book.buy(..))"/>
    <!--配置切面，ref指向哪个增强类-->
    <aop:aspect ref="bookProxy">
        <!--增强作用在具体方法上-->
        <aop:before method="before" pointcut-ref="p" />
    </aop:aspect>
</aop:config>
```

JdbcTemplate

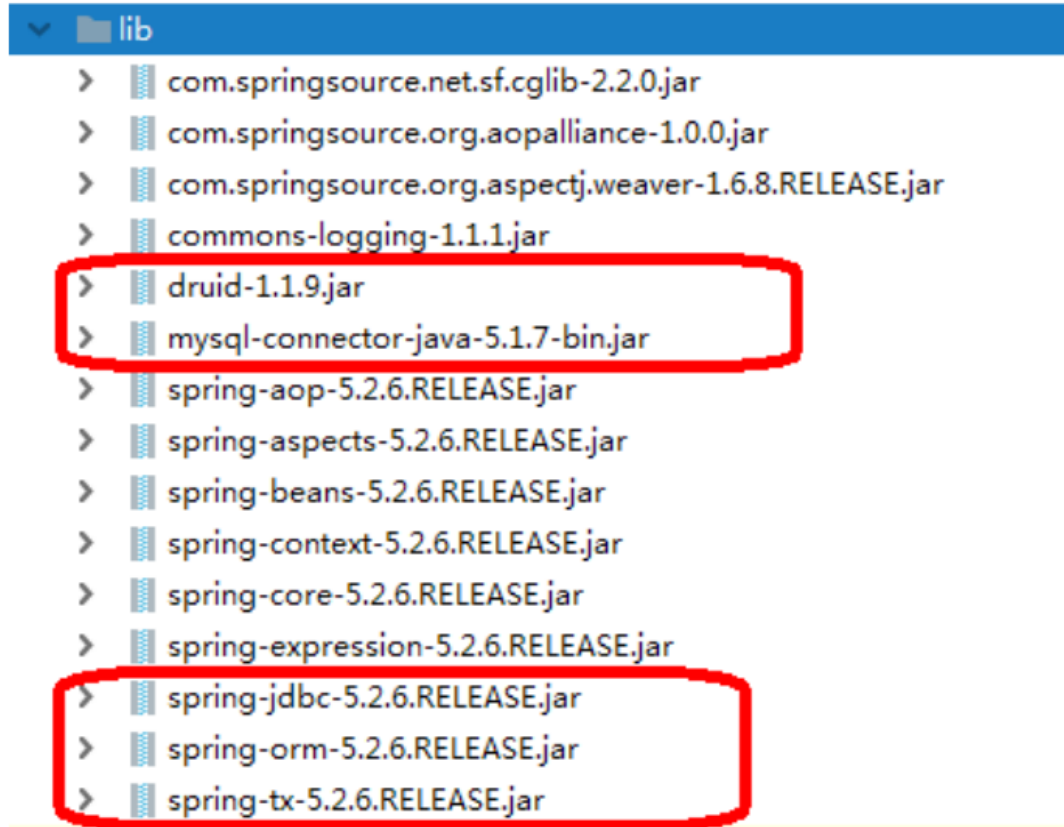
概念和准备

什么是JdbcTemplate

1. Spring框架对jdbc进行封装，使用JdbcTemplate方便实现**对数据库操作**

准备工作

1. 引入相关jar包 **【druid.jar、mysql-connector-java.jar、spring-jdbc、spring-orm、spring-tx】**



2. 在Spring配置文件中**配置数据库连接池**

```
<!-- 数据库连接池 -->
<bean id="dataSource"
      class="com.alibaba.druid.pool.DruidDataSource"
      destroy-method="close">
    <property name="url" value="jdbc:mysql:///user_db" />
    <property name="username" value="root" />
    <property name="password" value="root" />
    <property name="driverClassName"
      value="com.mysql.jdbc.Driver" />
</bean>
```

3. 配置JdbcTemplate对象，注入DataSource

```

<!--JdbcTemplate对象-->
<bean id='jdbcTemplate'
class='org.springframework.jdbc.core.JdbcTemplate'>
    <property name='dataSource' ref='dataSource'></property>
</bean>

```

4. 创建service类, 创建dao类, 在dao注入JdbcTemplate对象

```

<!--组件扫描-->
<context:component-scan base-package='com.atguigu'>
</context:component-scan>

```

```

//Service
@Service
public class BookService{
    //注入dao
    @Autowired
    private BookDao bookDao;
}

//Dao
@Repository
public class BookDaoImpl implements BookDao{
    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;
}

```

操作数据库

添加 update(String sql,Object... args)

1. 对应数据库创建**实体类**

```

public class User{
    private String userID;
    private String userName;
    private String ustatus;
    //对应个get和set方法以及无参有参构造器
}

```

2. 编写service和dao

1. 在dao进行数据库添加操作
2. 调用jdbcTemplate对象里面的update方法添加操作

1. 第一个参数 sql语句
2. 第二个参数 可变参数, 占位符的值

```
@Repository
public class BookDaoImpl implements BookDao{
    @Autowired
    private JdbcTemplate jdbcTemplate;
    //添加的方法
    @Override
    public void add(Book book){
        //创建Sql语句
        String sql = "insert into t_book values(?,?,?)";
        //调用方法实现
        Object[] args=
        {book.getUserId(),book.getUserName(),book.getUstatus()};
        int update = jdbcTemplate.update(sql,args);
    }
}
```

3. 测试类

```
@Test
public void testJdbcTemplateAdd(){
    ApplicationContext context = new
    ClassPathApplicationContext("bean1.xml");
    BookService bookService =
    context.getBean("bookService",BookService.class);
    Book book = new Book();
    book.setUserID("1");
    book.setUserName("java");
    book.setUstatus("a");
    bookService.addBook(book);
}
```

修改和删除 update(String sql,Object... args)

1. 修改

```

@Override
public void updateBook(Book book) {
    String sql = "update t_book set username=?,ustatus=? where user_id=?";
    Object[] args = {book.getUsername(),
book.getUstatus(),book.getUserId()};
    int update = jdbcTemplate.update(sql, args);
    System.out.println(update);
}

```

2. 删除

```

@Override
public void delete(String id) {
    String sql = "delete from t_book where user_id=?";
    int update = jdbcTemplate.update(sql, id);
    System.out.println(update);
}

```

查询返回某个值 queryForObject(String sql,Class requiredType)

1. 查询表里面有多少条记录，返回是某个值
2. 使用JdbcTemplate实现查询返回某个值代码
 1. queryForObject(String sql,Class requiredType)
 2. 两个参数，第一个SQL语句，第二个**返回类型**

```

//查询表记录数
@Override
public int selectCount(){
    String sql = "select count(*) from t_book";
    Integer count =
jdbcTemplate.queryForObject(sql,Integer.class);
    return count;
}

```

查询返回对象 queryForObject(String sql,RowMapper rowMapper,Object ... args)

1. 查询图书详情
2. JdbcTemplate 实现查询返回对象
 1. queryForObject(String sql,RowMapper rowMapper,Object ... args)
 2. 三个参数

1. 第一个为sql语句
2. 第二个为RowMapper接口，针对返回不同类型数据，使用这个接口里面的实现类完成数据封装，如
BeanPropertyRowMapper(Book.class)
3. 第三个参数占位符值

```
//查询返回对象
@Override
public Book findBookInfo(String id){
    String sql = "select * from t_book where user_id = ?";
    Book book = jdbcTemplate.queryForObject(sql,new
    BeanPropertyRowMapper<Book>(Book.class),id);
}
```

查询返回集合 query(String sql,RowMapper rowMapper,Object... args)

1. 场景：查询图书列表分页
2. 调用jdbcTemplate方法实现查询返回集合
 1. query(String sql,RowMapper rowMapper,Object... args)
 2. 三个参数
 1. 第一个参数为sql语句
 2. 第二个参数为RowMapper接口，针对返回不同类型数据，使用这个接口里面的实现类完成数据封装
 3. 第三个参数，占位符的值

```
//查询返回集合
@Override
public List<Book> findAllBook(){
    String sql = "select * from t_book";
    List<Book> bookList = jdbcTemplate.query(sql,new
    BeanPropertyRowMapper<Book>(Book.class));
    return bookList;
}
```

批量操作 batchUpdate(String sql,List<Object[]> batchArgs)

1. 操作表里面多条记录
2. JdbcTemplate实现**批量添加操作**
3. batchUpdate(String sql,List<Object[]> batchArgs)

4. 两个参数

1. 第一个 sql语句
2. 第二个 List集合，添加多条记录数据

```
//批量添加
@Override
public void batchAddBook(List<Object[]> batchArgs){
    String sql = "insert into t_book values(?,?,?)";
    int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);
}

//批量添加测试
List<Object[]> batchArgs = new ArrayList<>();
Object[] o1 = {"3", "java", "a"};
Object[] o2 = {"4", "C++", "b"};
Object[] o3 = {"5", "MySQL", "c"};
batchArgs.add(o1);
batchArgs.add(o2);
batchArgs.add(o3);
bookService.batchAdd(batchArgs);
```

5. JdbcTemplate实现批量修改操作

```
//批量修改
@Override
public void batchUpdateBook(List<Object[]> batchArgs) {
    String sql = "update t_book set username=?,ustatus=? where user_id=?";
    int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);
    System.out.println(Arrays.toString(ints));
}

//批量修改
List<Object[]> batchArgs = new ArrayList<>();
Object[] o1 = {"java0909", "a3", "3"};
Object[] o2 = {"c++1010", "b4", "4"};
Object[] o3 = {"MySQL1111", "c5", "5"};
batchArgs.add(o1);
batchArgs.add(o2);
batchArgs.add(o3);
//调用方法实现批量修改
bookService.batchUpdate(batchArgs);
```

6. JdbcTemplate实现批量删除操作

```
//批量删除
@Override
public void batchDeleteBook(List<Object[]> batchArgs) {
    String sql = "delete from t_book where user_id=?";
```

```
int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);
System.out.println(Arrays.toString(ints));
}
//批量删除
List<Object[]> batchArgs = new ArrayList<>();Object[] o1 =
{"3"};
Object[] o2 = {"4"};
batchArgs.add(o1);
batchArgs.add(o2);
//调用方法实现批量删除
bookService.batchDelete(batchArgs);
```

事务操作

事务概念

1. 事务是数据库操作最基本单元，逻辑上一组操作，**要么都成功，要么有一个失败所有操作都失败**
2. 典型场景：银行转账
3. 事务的四个特性（ACID）
 1. 原子性——过程不可分割
 2. 一致性——操作前后总量不变
 3. 隔离性——多事务操作之前不会相互影响
 4. 持久性——提交后表中数据发生变化

搭建事务操作环境

1. Service 业务操作、Dao 数据库操作，不写业务
2. 创建数据库表，添加记录
3. 创建Service，搭建Dao，完成对象创建和注入关系
 1. 在service注入dao，在dao注入JdbcTemplate，在JdbcTemplate注入DataSource

```

@Service
public class UserService{
    @Autowired
    private UserDao userDao;
}

@Repository
public class UserDaoImpl implements UserDao{
    @Autowired
    private JdbcTemplate jdbcTemplate;
}

```

4. 在dao创建两个方法，多钱和少钱的方法，在service创建方法，转账的方法

```

@Repository
public class UserDaoImpl implements UserDao{
    @Autowired
    private JdbcTemplate jdbcTemplate;

    //少钱
    @Override
    public void reduceMoney(){
        String sql = "update t_count set money=money-? where username=?";
        jdbcTemplate.update(sql,100,"lucy");
    }

    //多钱
    public void addMoney(){
        String sql = "update t_count set money=money+? where username=?";
        jdbcTemplate.update(sql,100,"lucy");
    }
}

```

```

@Service
public class UserService{
    @Autowired
    private UserDao userDao;

    //转账的方法
    public void accountMoney(){
        userDao.reduceMoney();
        userDao.addMoney();
    }
}

```

5. 上面代码，如果代码执行过程中出现异常，那么会出现问题，使用事务进行解决

6. 事务操作过程

```
public void accountMoney(){
    try{
        //第一步 开启事务
        //第二步 进行业务操作
        userDao.reduceMoney();
        //模拟异常
        int i = 10/0;
        userDao.addMoney();
        //第三步 没有发生异常，提交事务
    }catch(Exception e){
        //第四步 出现异常，事务回滚
    }
}
```

Spring事务管理介绍

管理介绍

1. 事务添加到JavaEE三层架构里面Service层（业务逻辑层）
2. 在Spring进行事务管理操作
 1. 两种方式：编程式事务管理和声明式事务管理（使用）
3. 声明式事务管理
 1. 基于注解方式（使用）
 2. 基于xml配置文件方式
4. 在Spring进行声明式事务管理，底层使用AOP原理
5. Spring事务管理API
 1. 提供一个接口，代表事务管理器，这个接口针对不同的框架提供不同的实现类
 2. 接口——PlatformTransactionManager
 3. 实现类——DataSourceTransactionManager

注解声明式事务管理

1. 在Spring配置文件配置事务管理器

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

2. 在Spring配置文件，开启事务注解

1. 在Spring配置文件引入名称空间tx
2. 开始事务注解

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"

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

```
<!--开启事务注解-->
<tx:annotation-driven transaction-
manager='transactionManager'></tx:annotation-driven>
```

3. 在Service类上面（或者Service类里面的方法上面）添加事务注解

1. @Transactional，这个注解添加到**类上面，也可以添加方法上面**
2. 如果把这个注解添加到类上面，这个类里面**所有方法都添加事务**
3. 如果把这个注解添加到方法上面，这个方法添加事务

```
@Service
@Transactional
public class UserService{}
```

声明式事务管理参数配置

1. 在Service类上面添加注解@**Transaction**，这个注解里面可以配置事务相关参数

2. propagation: 事务传播行为

1. 多事务方法直接进行调用，这个过程中事务是如何进行管理的

2. 事务方法：对数据库表数据进行变化的操作

3. Spring框架事务**传播行为有7种**

1. REQUIRED：如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动新的事务，并在自己的事务内进行
2. REQUIRED_NEW：当前的方法必须启动新事务，并在它自己的事务内进行，如果有事务正在运行，必须将它挂起
3. SUPPORTS：如果有事务在运行，当前的方法就在这个事务内运行，否则它可以不运行在事务中
4. NOT_SUPPORTED：当前的方法不应该运行在事务中，如果有运行的事务，将它挂起
5. MANDATORY：当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
6. NEVER：当前的方法不应该运行在事务中，如果有运行的事务，就抛出异常
7. NESTED：如果有事务正在运行，当前的方法就应该在这个事务的嵌套事务内运行，否则，就启动一个新的事务，并在它自己的事务内运行

```
@Service
@Transactional(propagation = Propagation.REQUIRED)
public class UserService{}
```

3. isolation: 事务隔离级别

1. 事务有特性成为隔离性，多事务操作之间不会产生影响，不考虑隔离性产生很多问题

2. 有三个读问题：脏读、不可重复读、虚（幻）读

1. **脏读**：一个未提交事务读取到另一个未提交事务的数据（不能发生）
2. **不可重复读**：一个未提交事务读取到另一个提交事务**修改数据**
3. **虚（幻）读**：一个未提交事务读取到另一个提交事务**添加数据**

3. **解决：通过设置事务隔离级别，解决读问题**

	脏读	不可重复读	幻读
READ_UNCOMMITTED (读未提交)	有	有	有
READ_COMMITTED (读已提交)	无	有	有
REPEATABLE_READ (可重复读)	无	无	有
SERIALIZABLE (串行化)	无	无	无

```
@Service
@Transactional(propagation=
Propagation.REQUIRED, isolation=
Isolation.REPEATABLE_READ)
```

4. timeout: 超时时间

1. 事务需要在一定时间内进行提交，如果不提交进行回滚
2. 默认值为-1，设置时间以**秒为单位**进行计算

5. readOnly: 是否只读

1. 读：查询操作，写：添加修改删除操作
2. readOnly默认值为false，表示可以查询，可以添加删除修改

6. rollbackFor: 回滚

1. 设置出现哪些异常进行事务回滚

7. noRollbackFor: 不回滚

1. 设置出现哪些异常进行事务不回滚

XMI声明式事务管理

1. 第一步 配置事务管理器
2. 第二步 配置通知
3. 第三步 配置切入点和切面

```
<!--1 创建事务管理器-->
<bean id='transactionManager'
class='org.springframework.jdbc.datasource.DataSourceTransacti
onManager'>
    <!--注入数据源-->
    <property name='dataSource' ref='dataSource'></property>
</bean>

<!--2 配置通知-->
<tx:advice id='txadvice'>
    <!--配置事务参数-->
```

```

        <tx:attributes>
            <!--指定哪种规则的方法上面添加事务-->
            <tx:method name='accountMoney' propagation='REQUIRED' />
        </tx:attributes>
    </tx:advice>

    <!--3 配置切入点 and 切面-->
    <aop:config>
        <!--切入点-->
        <aop:pointcut id='pt' expression='execution(*
com.atguigu.spring.UserService.*(..))' />
        <!--切面-->
        <aop:advisor advice-ref='txadvice' pointcut-ref='pt' />
    </aop:config>

```

完全注解声明式事务管理

1. 创建配置类，使用配置类代替xml配置文件

```

@Configuration //配置类
@ComponentScan(basePackages='com.atguigu') //组件扫描
@EnableTransactionManagement //开启事务
public class TxConfig{
    //创建数据库连接池
    @Bean
    public DuridDataSource getDuridDataSource(){
        DuridDataSource dataSource = new DuridDataSource();

        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql:///user_db");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        return dataSource;
    }

    //创建JdbcTemplate
    @Bean
    public JdbcTemplate getJdbcTemplate(DataSource dataSource)
    {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    //创建事务管理器
    @Bean

```



```

public DataSourceTransactionManager
getDataSourceTransactionManager(DataSource dataSource){
    DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();
    transactionManager.setDataSource(dataSource);
    return transactionManager;
}
}

```

Spring5新特性

1. 整个Spring5框架的代码基于**Java8**，运行兼容Java9，许多不建议使用的类和方法在代码库中删除

日志封装

1. Spring5.0 框架自带了通用的**日志封装**
 1. Spring5已经移除Log4jConfigListener，官方建议使用Log4j2
 2. Spring5 框架整合Log4j2
2. 第一步 引入jar包【log4j-api-2.11.2.jar、log4j-core、log4j-slf4j-impl、slf4j-api】
3. 第二步 创建log4j2.xml配置文件

```

<?xml version="1.0" encoding="UTF-8" ?>
<configuration status='INFO'>
    <!--先定义所有的appender-->
    <appenders>
        <!--输出日志信息到控制台-->
        <console name='Console' target='SYSTEM_OUT'>
            <!--控制台日志输出的格式-->
            <PatternLayout pattern="%d{yyyy-MM-dd
HH:mm:ss.SSS} [%t]%-5level %logger{36} -%msg%n" />
        </console>
    </appenders>
    <!--然后定义 logger，只有定义 logger 并引入的 appender，
appender 才会生效-->
    <!--root: 用于指定项目的根日志，如果没有单独指定 Logger，则会使用
root 作为默认的日志输出-->
    <loggers>
        <root level='info'>
            <appender-ref ref='Console' />
        </root>
    </loggers>
</configuration>

```

Spring5 框架核心容器支持@Nullable 注解

1. @Nullable 注解可以使用在方法上面，属性上面，参数上面，表示方法返回可以为空，属性值可以为空，参数值可以为空
2. 注解用在方法上面，方法返回值可以为空

```
@Nullable  
String getId();
```

3. 注解使用在方法参数上面，方法参数可以为空

```
public<T> void registerBean(@Nullable String  
benName, this.reader, registerBean(beanClass, beanName, suppli)){}
```

4. 注解使用在属性上面，属性值可以为空

```
@Nullable  
private String bookName;
```

Spring5 核心容器支持函数式风格 GenericApplicationContext

```
//函数式风格创建对象，交给spring进行管理  
@Test  
public void testGenericApplicationContext(){  
    //1 创建GenericApplicationContext对象  
    GenericApplicationContext context = new  
GenericApplicationContext();  
    //2 调用context的方法对象注册  
    context.refresh();  
    context.registerBean("user1",User.class,()->new User());  
    //3 获取在spring注册的对象  
    User user = (User)context.getBean("user1");  
}
```

Spring5 支持整合 JUnit5

1. 整合JUnit4
 1. 第一步 引入Spring相关对测试依赖 【spring-test-5.2.6.RELEASE.jar】
 2. 第二步 创建测试类，使用注解方式完成

```
//单元测试框架
@RunWith(SpringJUnit4ClassRunner.class)
//加载配置文件
@ContextConfiguration("classpath:bean1.xml")
public class JTest4{
    @Autowired
    private UserService userService;
    @Test
    public void test1(){
        userService.arrcountMoney();
    }
}
```

2. Spring整合JUnit5

1. 第一步 引入JUnit5的jar包 【JUnit5.3.jar】
2. 第二步 创建测试类，使用注解完成

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:bean1.xml")
public class JTest5{
    @Autowired
    private UserService userService;
    @Test
    public void test1{
        userService.accountMoney();
    }
}
```

3. 使用一个复合注解替代上面两个完全注解完成整合

```
@SpringJUnitConfig(locations="classpath:bean1.xml")
public class JTest5{
    @Autowired
    private UserService userService;
    @Test
    public void test1(){
        userService.accountMoney();
    }
}
```

Spring5 框架新功能 (Webflux)

SpringWebflux 介绍

1. 是Spring5添加新的模块，**用于web开发**，功能和SpringMVC 类似的，Webflux使用当前一种比较流程响应式编程出现的框架
2. 使用传统web框架，比如SpringMVC，这些基于Servlet容器，Webflux是一种**异步非阻塞**的框架，异步非阻塞的框架在Servlet3.1以后才支持，核心是基于Reactor的相关API实现的。

3. 什么是异步非阻塞

1. 异步和同步

1. **异步和同步针对调用者**，调用者发送请求，如果等着对方回应之后才去做其他事情就是同步，如果发送请求之后不等则对方回应就去做其他的事情就是异步

2. 非阻塞和阻塞

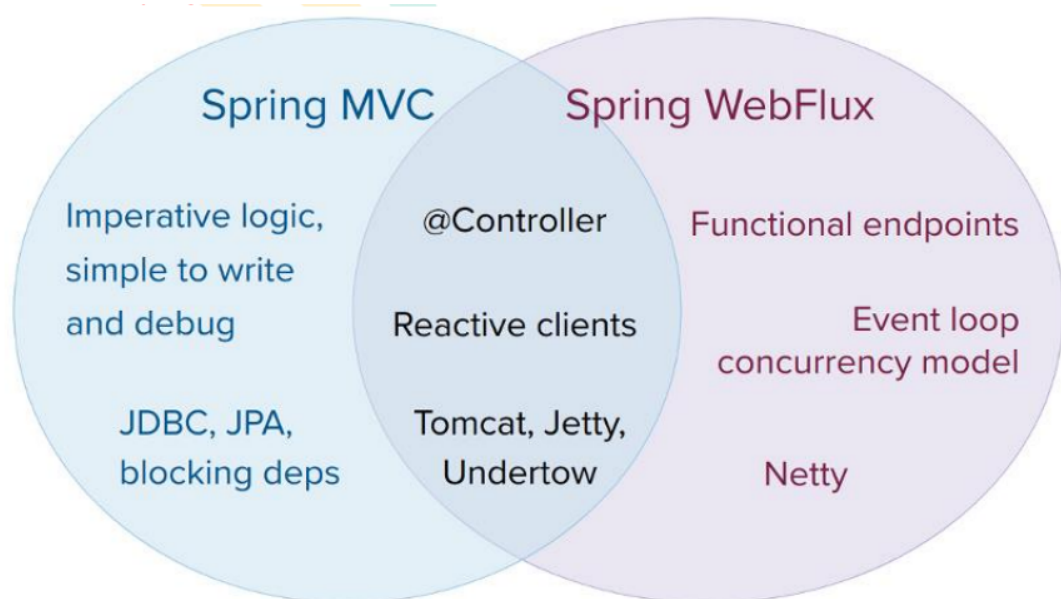
1. **阻塞和非阻塞针对被调用者**，被调用者收到请求之后，做完请求任务之后才给出反馈就是阻塞，收到请求之后马上给出反馈然后再去做事情就是非阻塞

3. 上面都是针对对象不一样

4. Webflux特点

1. 第一 非阻塞式：在有限资源下，提高系统吞吐量和伸缩性，以Reactor为基础实现响应式编程
2. 第二 函数式编程：在Spring5框架基于java8，Webflux使用java8函数式编程方式实现路由请求

5. 比较SpringMVC



1. 第一 两个框架都可以使用注解方式，都运行在Tomcat等容器中
2. 第二 SpringMVC采用命令式编程，Webflux使用异步响应式编程

响应式编程 (Java 实现)

1. 什么是响应式编程

1. 响应式编程是一种面向数据流和变化传播的编程范式。这意味着可以在编程语言中很方便地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。电子表格程序就是响应式编程的一个例子。单元格可以包含字面值或类似" $B1+C1$ "的公式，而包含公式的单元格的值会依据其他单元格的值的变化而变化。

2. Java8及其之前版本

1. 提供的观察者模式两个类Observer和Observable

```
public class ObserverDemo extends Observable{
    public static void main(String[] args){
        ObserverDemo observer = new ObserverDemo();
        //添加观察者
        observer.addObserver((o,arg)->{
            System.out.println("发生变化");
        });
        observer.addObserver((o,arg)->{
            System.out.println("手动呗观察者通知，准备改变");
        });
        //数据变化
        observer.setChanged();
        //通知
        observer.notifyObservers();
    }
}
```

响应式编程 (Reactor 实现)

1. 响应式编程操作中，Reactor是满足Reactive规范框架
2. Reactor有两个核心类，Mono和Flux，这两个类实现接口Publisher，提供丰富操作符。Flux对象思想发布者，返回N个元素；Mono实现发布者，返回0或者1个元素
3. Flux和Mono都是数据流的发布者，使用Flux和Mono都可以发出三种数据信号：**元素值，错误信号，完成信号**，错误信号和完成信号都代表终止信号，终止信号用于告诉订阅者数据流结束了，错误信号终止数据流同时把错误信息传递给订阅者
4. 演示Flux和Mono

```
<!--第一步 引入依赖-->
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.1.5.RELEASE</version>
</dependency>
```

```
//第二步 编程代码
public static void main(String[] args){
    //just方法直接声明
    Flux.just(1,2,3,4);
    Mono.just(1);
    //其他方法
    Integer[] array = {1,2,3,4};
    Flux.fromArray(array);

    List<Integer> list = Arrays.asList(array);
    Flux.fromIterable(list);

    Stream<Integer> stream = list.stream();
    Flux.fromStream(stream);
}
```

5. 三种信号特点

1. 错误信号和完成信号都是终止信号，不能共存的
 2. 如果没有发送任何元素值，而是直接发送错误或者完成信号，表示空数据流
 3. 如果没有错误信号，没有完成信号，表示是无限数据流
6. 调用just或者其他方法只是声明数据流，数据流并没有发出，只有进行订阅之后才会触发数据流，不订阅什么都不会发生的

```
//just方法直接声明
Flux.just(1,2,3,4).subscribe(System.out::print);
Mono.just(1).subscribe(System.out::print);
```

7. 操作符

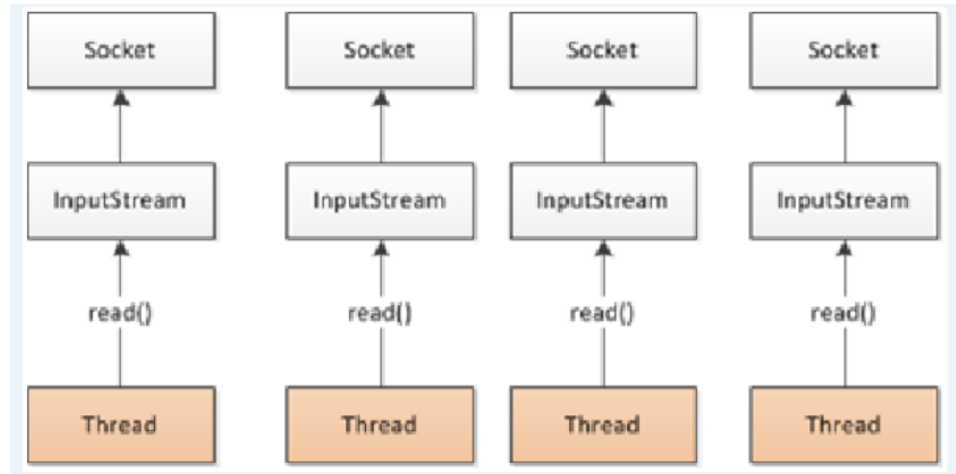
1. 对数据流进行一道道操作，成为操作符，比如工厂流水线
2. 第一 map 元素映射为新元素
3. 第二 flatMap 元素映射为流
 1. 把每个元素转换流，把转换之后多个流合并大的流

SpringWebflux 执行流程和核心 API

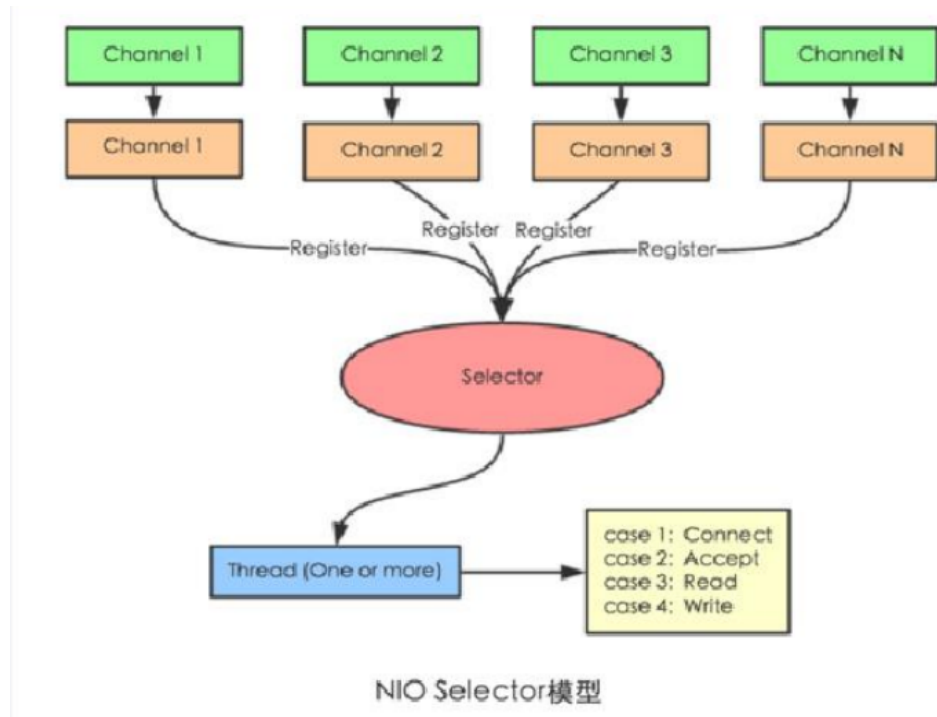
1. SpringWebflux 基于 Reactor，默认使用容器是Netty，Netty是高性能的NIO框架，**异步非阻塞**的框架

2. Netty

1. BIO



2. NIO



3. SpringWebflux 执行过程和 SpringMVC相似的

1. SpringWebflux核心控制器 DispatcherHandler，实现接口 WebHandler

2. 接口 WebHandler 有一个方法

```
public interface webHandler{  
    Mono<Void> handle(ServerWebExchange var1);  
}
```

4. SpringWebflux 里面 DispatcherHandler，负责请求的处理

1. HandlerMapping：请求查询到处理的方法

2. HandlerAdapter: 真正负责请求处理
3. HandlerResultHandler: 响应结果处理
5. SpringWebflux 实现函数式编程, 两个接口: RouterFunction (路由处理) 和 HandlerFunction (处理函数)

SpringWebflux (基于注解编程模型)

1. SpringWebflux实现方式有两种: 注解编程模型和函数式编程模型
2. 使用注解编程模型, 和之前SpringMVC使用相似的, 只需要把相关依赖配置到项目中, SpringBoot自动配置相关运行容器, **默认情况下使用Netty服务器**
3. 第一步 创建SpringBoot工程, 引入Webflux依赖

```
<!--第一步 创建SpringBoot工程, 引入webflux依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

4. 第二步 配置启动端口号

```
server.port=8081
```

5. 第三步 创建包和相关类

```
//实体类
public class User{
    private String name;
    private String gender;
    private Integer age;

    //含参、无参构造器
    //get、set方法
}
```

```
//创建接口定义操作的方法
//用户操作接口
public interface UserService{
    //根据id查询用户
    Mono<User> getUserById(int id);
    //查询所有用户
    Flux<User> getAllUser();
    //添加用户
    Mono<Void> saveUserInfo(Mono<User> user);
}
```

```
//接口实现类
```



```

public class UserServiceImpl implements UserService{
    private final Map<Integer,User> users = new HashMap<>();
    public UserServiceImpl(){
        this.users.put(1,new User("lucy","man",20));
        this.users.put(2,new User("lucy","man",20));
        this.users.put(3,new User("lucy","man",20));
        this.users.put(4,new User("lucy","man",20));
    }

    //根据id查询
    @Override
    public Mono<User> getUserById(int id){
        return Mono.justOrEmpty(this.users.get(id));
    }
    //查询多个用户
    @Override
    public Flux<User> getAllUser(){
        Flux.fromIterable(this.users.values());
    }
    //添加用户
    @Override
    public Mono<Void> saveUserInfo(Mono<User> userMono){
        return userMono.doOnNext(person ->{
            int id = users.size()-1;
            users.put(id,person);
        }).thenReturn(Mono.empty());
    }
}

```

6. 创建controller

```

@RestController
public class UserController{
    @Autowired
    private UserService userService;

    //id 查询
    @GetMapping("/user/{id}")
    public Mono<User> geetUserId(@PathVariable int id){
        return userService.getUserById(id);
    }
    //查询所有
    @GetMapping("/user")
    public Flux<User> getUsers() {
        return userService.getAllUser();
    }
    //添加
    @PostMapping("/saveuser")

```

```

    public Mono<Void> saveUser(@RequestBody User user) {
        Mono<User> userMono = Mono.just(user);
        return userService.saveUserInfo(userMono);
    }
}

```

7. 说明

1. SpringMVC方式实现，同步阻塞方式，基于 SpringMVC+Servlet+Tomcat
2. SpringWebflux方式实现，异步非阻塞方式，基于 SpringWebflux+Reactor+netty

SpringWebflux（基于函数式编程模型）

1. 在使用函数式编程模型操作时候，需要自己初始化服务器
2. 基于函数式编程模型时候，有两个核心接口：RouterFunction（实现路由功能，请求转发给对应的 handler）和 HandlerFunction（处理请求生成响应的函数）。核心任务定义两个函数式接口的实现并且启动需要的服务器。
3. SpringWebflux 请求和响应不再是 ServletRequest 和 ServletResponse，而是 ServerRequest 和 ServerResponse
4. 第一步 把注解编程模型工程复制一份，保留 entity 和 service 内容
5. 第二步 创建 Handler（具体实现方法）

```

public class UserHandler {

    private final UserService userService;

    public UserHandler(UserService userService) {
        this.userService = userService;
    }

    //根据 id 查询

    public Mono<ServerResponse> getUserById(ServerRequest request) {

        //获取 id 值

        int userId =
Integer.valueOf(request.pathVariable("id"));

        //空值处理
        Mono<ServerResponse> notFound =
ServerResponse.notFound().build();
        //调用 service 方法得到数据
    }
}

```

```

        Mono<User> userMono =
this.userService.getUserById(userId);
        //把 userMono 进行转换返回
        //使用 Reactor 操作符 flatMap
        return userMono.flatMap(person ->
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bo
dy(fromObject(person))).switchIfEmpty(notFound);
    }

    //查询所有
    public Mono<ServerResponse> getAllUsers() {
        //调用 service 得到结果
        Flux<User> users = this.userService.getAllUser();
        return
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).b
ody(users,User.class);
    }

    //添加
    public Mono<ServerResponse> saveUser(ServerRequest
request) {
        //得到 user 对象
        Mono<User> userMono = request.bodyToMono(User.class);
        return
ServerResponse.ok().build(this.userService.saveUserInfo(userMo
no));
    }
}

```

6. 第三步 初始化服务器，编写 Router

```

//1 创建 Router 路由
public RouterFunction<ServerResponse> routingFunction() {
    //创建 hanler 对象
    UserService userService = new UserServiceImpl();
    UserHandler handler = new UserHandler(userService);
    //设置路由
    return
RouterFunctions.route(GET("/users/{id}").and(accept(APPLICATIO
N_JSON)),handler::getUserById).andRoute(GET("/users").and(acce
pt(APPLICATION_JSON)),handler::get
AllUsers);
}

//创建服务器完成适配
//2 创建服务器完成适配
public void createReactorServer() {
    //路由和 handler 适配
}

```

```

RouterFunction<ServerResponse> route = routingFunction();
HttpHandler httpHandler = toHttpHandler(route);
ReactorHttpHandlerAdapter adapter = new
ReactorHttpHandlerAdapter(httpHandler);
    //创建服务器
    HttpServer httpServer = HttpServer.create();
    httpServer.handle(adapter).bindNow();
}

//最终调用
public static void main(String[] args) throws Exception{
    Server server = new Server();
    server.createReactorServer();
    System.out.println("enter to exit");
    System.in.read();
}

```

7. 使用 WebClient 调用

```

public class Client {
    public static void main(String[] args) {
        //调用服务器地址
        WebClient webClient =
        WebClient.create("http://127.0.0.1:5794");
        //根据 id 查询
        String id = "1";
        User userresult = webClient.get().uri("/users/{id}",
        id).accept(MediaType.APPLICATION_JSON).retrieve().bodyToMono(U
        ser.class).block();
        System.out.println(userresult.getName());
        //查询所有
        Flux<User> results =
        webClient.get().uri("/users").accept(MediaType.APPLICATION_JSO
        N).retrieve().bodyToFlux(User.class);
        results.map(stu ->
        stu.getName()).buffer().doOnNext(System.out::println).blockFir
        st();
    }
}

```