SSM框架

内容

- Spring
- SpringMVC
- Maven高级
- SpringBoot
- MybatisPlus

说明

- 重点：Spring、SpringBoot
- SpringMVC隶属于Spring
- Maven高级是学习SpringBoot的基础前置知识
- Mybatis升级版--------MybatisPlus

目标

- 基于SpringBoot实现基础SSM框架整合
- 掌握第三方技术与SpringBoot

一、Spring

1.1 Spring介绍

1.1.1 优势

- 简化开发
  - IoC
  - AOP
    - 事务处理
- 框架整合
  - MyBatis
  - MyBatis-plus
  - Struts
  - Struts2
  - Hibernate

- ……

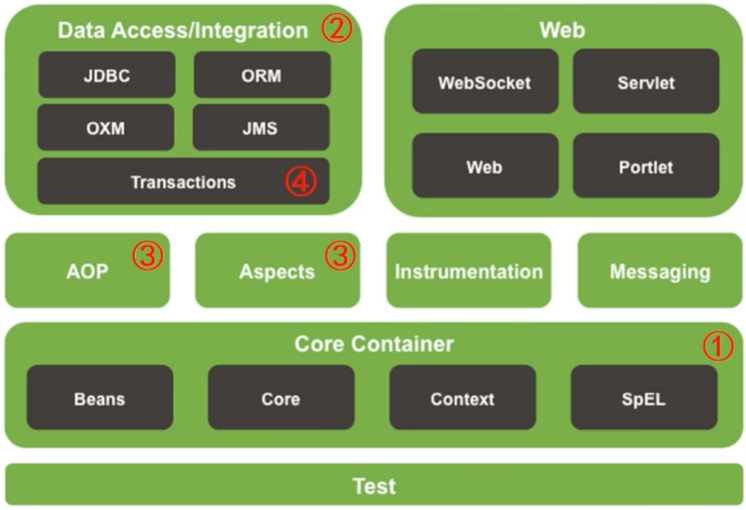
1.2.1 Spring发展

Spring发展到今天已经形成了一种开发的生态圈，Spring提供了若个个项目，每个项目用于完成特定的功能。

- Spring Framework：底层框架
- SpringBoot
- SpringClound

1.2 Spring Framework

1.2.1 系统架构



- Data Access：数据访问
- Data Integration：数据集成
- Web：Web开发
- AOP：面向切面编程
- Aspects：AOP思想实现
- Core Container：核心容器
- Test：单元测试与集成测试

1.2.2 核心容器

1. 一些概念解析

(1) IoC

IoC（Inversion of Control）控制反转：对象的创建控制权由程序转移到**外部**，这种思想称为控制反转

- IoC前:

  代码耦合度高

- IoC后:

耦合度高的主要原因是：传统的代码使用new创建对象，所以当改变类时需改动很多代码。

所以，使用对象时，在程序中不要主动使用new产生对象，转为由**外部**提供对象

(2) IoC容器

- Spring技术对IoC思想进行了实现，它提供了一个容器，称为IoC容器，用来充当IoC思想中的“外部”
- IoC容器负责对象的创建、初始化等一系列工作，被创建或被管理的对象在IoC中统称为**Bean**

(3) DI依赖注入

DI（Dependency Injection）依赖注入

- 在容器中建立bean与bean之间的依赖关系的整个过程，称为**依赖注入**
- 依赖，即一个对象引用/使用另一个对象

充分解耦：

- 使用IoC容器管理bean
- 在IoC容器内将有依赖关系的bean进行关系绑定

2. IoC容器

管理什么----------Service与Dao

如何将被管理的对象告知IoC容器--------配置

如何获取到IoC容器----------接口 ApplicationContext

如何从IoC容器中获取bean----------接口方法 getBean

使用Spring导入哪些坐标----------pom.xml

1.2.3 新建Spring项目

1. 步骤

(1)新建maven项目

使用maven创建一个maven项目

(2) 添加依赖

在pom.xml中添加spring和其他依赖

```
<!--   spring 依赖     -->
<dependency>
    <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-context</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>

<!--        junit 单元测试-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
</dependency>
```

(3) 创建Spring配置文件

在scr/main/resources包下右键new --> XML Configuration File --> Spring Config，新建**applicationContext.xml**

(4) 配置bean

在Spring配置文件applicationContext.xml中配置bean

```
<!--
    在<beans>标签中配置
    id属性：给bean起名字，同一上下文不能重复
    class属性：给bean定义类型（管理什么类型的对象）
    -->
    <bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl"/>
    <bean id="bookService" class="com.hardy.service.impl.BookServiceImpl">
        <!--  配置Server与Dao的关系（配置对象（bean）之间的依赖关系）       -->
        <!--  property标签表示配置当前bean的属性-->
        <!--  name属性表示配置哪一个具体的属性-->
        <!--  ref属性表示参照哪一个bean，其值为bean的id（依赖哪一个）-->
        <property name="bookDao" ref="bookDao"/>
    </bean>
```

bean配置的一些说明：

- 当两个类或者说对象有依赖关系时（假设A依赖B，即A中有一个属性指向B类一个的对象），则在A的bean中使用property标签来配置依赖关系，并且在A类中设置对应属性的set方法

(5) 编写业务代码

```
// 1. 获取IoC容器
ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");

// 2. 获取bean
BookDao bookDao = (BookDao) ctx.getBean("bookDao");
bookDao.save();
```

1.2.4 bean的配置

1. bean起别名

- 使用**name**属性，多个别名之间可用"," ";" 或者空格隔开

```
<bean id="bookService" name="service
service2"class="com.hardy.service.impl.BookServiceImpl">
        <property name="bookDao" ref="bookDao"/>
</bean>
```

2. bean的作用范围

- 即创建出来的bean是多个对象还是一个对象，即单例还是非单例。
- spring默认创建的是单例
- 使用**scope**属性指定作用范围
    - singleton：单例
    - prototype：非单例

```
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl" scope="prototype"/>
```

说明：

- 适合交给容器进行管理的bean（单例）（可以复用的对象）
    - 表现层对象
    - 业务层对象
    - 数据层对象
    - 工具对象
- 不适合交给容器进行管理的bean（非单例）（有状态的）
    - 封装实体的域对象

1.2.5 bean的实例化

1. bean是如何创建的

- bean本质上就是对象，创建bean**使用构造方法**完成
- bean使用反射的方法来获得构造方法创建bean，所以即使构造方法是private也可以调用
- 创建bean调用的是无参的构造方法，所以必须要有一个无参的构造方法，否则抛出异常BeanCreationException

2. 实例化bean的三种方式

(1) 构造方法实例化bean（常用）

(2) 静态工厂实例化bean（了解）

- 前提：必须要真实存在工厂类（bookDaoFactory），以及类中的静态方法（getOrderDao），在静态方法中返回一个需要的对象
- 该种方法主要是为了兼容早期遗留的系统使用

```
<bean id="bookDao" class="com.hardy.factory.bookDaoFactory" factory-
method="getOrderDao"/>
```

```
// 静态工厂
public class bookDaoFactory(){
    public static OrderDao(){
        return new OrderDaoIml();
    }
}
```

(3) 实例工厂实例化bean（了解）

```
<!-- 1.实例化工厂 -->
<bean id="userFactory" class="com.hardy.factory.userDaoFactory"/>

<!-- 2. 使用实例化工厂实例化bean -->
<bean id="userDao" factory-method="getUserDao" factory-bean="userFactory"/>
```

```
// 实例工厂
public class UserDaoFactory{
    public UserDao getUserDao(){
        return new UserDaoImpl();
    }
}
```

(4) 实例工厂实例化bean --- 改进（实用）

无需改动业务代码

```
<bean id="userDao" class="com.hardy.factory.UserDaoFactoryBean"/>
```

```
// 需实现FactoryBean接口，传递的泛型即想要实例化对象的类型
public class UserDaoFactoryBean implements FactoryBean<UserDao> {
    // 替代原始实例工厂中创建对象的方法
    public UserDao getObject() throw Exception{
        return new UserDaoImpl();
    }

    public Class<?> getObjectType(){
        return UserDao.class;
    }

    // 该方法决定使用这种方式创建的bean是单例还是非单例，通常是单例，所以该方法一般不屑
```

```
    public boolean isSingleton(){
        return false
    }
}
```

1.2.6 bean的生命周期

- bean的生命周期：从创建到销毁的整体过程】
- bean的生命周期控制：在bean创建后到销毁前做一些事情

1. bean生命周期控制

(1) 配置

- init-method
- destroy-method

```
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl" init-method="init"
destroy-method="destory"/>
```

```
public class BookDaoImpl implements BookDao {
    // 表示bean初始化对应的操作（根据业务来写，如读取文件、加载资源）
    public void init(){
        System.out.println("init..");
    }
    // 表示bean销毁前对应的操作
    public void destory(){
        System.out.println("destory..");
    }

}
```

- 容器关闭前触发bean销毁。一般不会执行。java程序在java虚拟机中运行，当程序运行完后虚拟机退出，没有给bean销毁的机会。
- 关闭容器方法

  前提：ctx必须是ApplicationContext的实现类

  - 手动关闭

    ```
    ctx.close(); // 必须在程序执行完成退出前使用，否则出现异常
    ```

  - 设置关闭钩子

    ```
    ctx.registerShutdownHook();  // 可在任何地方使用
    ```

(2) 接口

- InitializingBean
- DisposableBean

```
// 实现InitializingBean, DisposableBean接口，无需像上面一样配置bean
public class BookServiceImpl implements BookService, InitializingBean,
DisposableBean {

    public void save(){
        System.out.println("book service save...");
        bookDao.save();
    }

    // 实现两个接口的方法
    @Override
    public void destroy() throws Exception {
        System.out.println("bookService destroy");
    }

    // 该方法在属性设置完成后才执行
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("bookService init");
    }
}
```

2. bean生命周期

- 初始化容器
    1. 创建对象（内存分配）
    2. 执行构造方法
    3. 执行属性注入（set操作）
    4. 执行bean初始化方法

- 使用bean
    1. 执行业务操作

- 关闭/销毁bean
    1. 执行bean销毁方法

1.2.7 依赖注入的三种方式

依赖注入的类型：

- 引用类型
- 简单类型（基本数据类型与String）

依赖注入方式：

- setter注入
  - 简单类型
  - 引用类型
- 构造器注入
  - 简单类型
  - 引用类型
- 自动装配

1. setter注入引用类型

```
    <bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl"/>
    <bean id="userDao" class="com.hardy.dao.impl.UserDaoImpl"/>

    <bean id="bookService" class="com.hardy.service.impl.BookServiceImpl">
        <property name="bookDao" ref="bookDao"/>
        <property name="userDao" ref="userDao"/>
    </bean>
```

```
// BookServiceImpl.java

// 有两个引用类型属性
private BookDao bookDao;
private UserDao userDao;

// 提供对应的setter方法
public void setUserDao(UserDao userDao) {
    this.userDao = userDao;
}

public void setBookDao(BookDao bookDao) {
    this.bookDao = bookDao;
}
```

2. setter注入简单类型

```
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl">
    <property name="databaseName" value="mysql"/>
    <property name="conenctionNum" value="10"/>
</bean>
```

```java
// BookDaoImpl.java
private int conenctionNum;
private String databaseName;

public void setConenctionNum(int conenctionNum) {
    this.conenctionNum = conenctionNum;
}

public void setDatabaseName(String databaseName) {
    this.databaseName = databaseName;
}
```

3. 构造器注入引用类型

```xml
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl"/>
<bean id="userDao" class="com.hardy.dao.impl.UserDaoImpl"/>

<bean id="bookService" class="com.hardy.service.impl.BookServiceImpl">
    <constructor-arg name="bookDao" ref="bookDao"/>
    <constructor-arg name="userDao" ref="userDao"/>
</bean>
```

```java
// BookServiceImpl.java
private BookDao bookDao;
private UserDao userDao;

public BookServiceImpl(BookDao bookDao, UserDao userDao) {
    this.bookDao = bookDao;
    this.userDao = userDao;
}
```

4. 构造器注入简单类型

```xml
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl">
    <constructor-arg name="connectionNum" value="10"/>
    <constructor-arg name="databaseName" value="mysql"/>
</bean>
```

```java
// BookDaoImpl.java
private String databaseName;
private int connectionNum;

public BookDaoImpl(String databaseName, int connectionNum) {
    this.databaseName = databaseName;
    this.connectionNum = connectionNum;
}
```

5. 依赖注入方式选择

1. 强制依赖使用构造器进行，使用setter注入有概率不进行注入导致null对象出现
2. 可选依赖使用setter注入进行，灵活性强
3. Spring框架倡导使用构造器，第三方框架内部大多数采用构造器注入的形式进行数据初始化，相对严谨
4. 如果有必要可以两者同时使用，使用构造器注入完成强制依赖的注入，使用setter方法注入完成可选依赖的注入
5. 实际开发过程中还要根据实际情况分析，如果受控对象没有提供setter方法就必须使用构造器注入
6. 自己开发的模块推荐使用setter注入

1.2.8 自动装配

- IoC容器根据bean所依赖的资源**在容器中自动查找并注入到bean中**的过程称为自动装配
- 自动装配方式
  - 按类型（常用）
  - 按名称
  - 按构造方法
  - 不启用自动装配

1. 按类型（推荐）

- 使用该方法时，类型必须唯一。

```
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl"/>

<!-- 该bean与上一个类型相同，类型不唯一，装配失败，应删去其中一个 -->
<bean id="bookDao2" class="com.hardy.dao.impl.BookDaoImpl"/>

<bean id="bookService" class="com.hardy.service.impl.BookServiceImpl"
autowire="byType"/>
```

```
// BookDaoImpl.java
private BookDao bookDao;

public void setBookDao(BookDao bookDao) {
    this.bookDao = bookDao;
```

2. 按名称（不推荐）

要装配的变量名与id名相同

```
<bean id="bookDao" class="com.hardy.dao.impl.BookDaoImpl"/>

<bean id="bookService" class="com.hardy.service.impl.BookServiceImpl"
autowire="byName"/>
```

```java
// BookDaoImpl.java
private BookDao bookDao;

public void setBookDao(BookDao bookDao) {
    this.bookDao = bookDao;
```

3. 依赖自动装配特征

- 自动装配用于引用类型依赖注入，不能对简单类型进行操作
- 自动装配优先级低于setter注入与构造器注入，同时出现时自动装配配置失效

1.2.9 集合注入

1.2.10 使用properties文件

1. 开启context命名空间

```xml
<!--applicationContext.xml-->
<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"
       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-beans.xsd
        ">
```

2. 使用context空间加载properties文件

```xml
<!--applicationContext.xml-->
<context:property-placeholder location="jdbc.properties"/>
<bean class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

1.2.11 注解开发定义bean

1. 简单使用

1. 使用@Component定义bean

```
@Component("bookDao")
public class BookDaoImpl implements BookDao {
}
```

2. 核心配置文件中通过组件扫面加载bean

   需要增加context命名空间

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

```
<context:component-scan base-package="com.hardy.dao"/>
```

@Component的三个衍生注解

- @Controller：表现层
- @Repository：数据层
- @Service：业务层

2. 纯注解开发

使用Java类（SpringConfig.java，配置类）代替xml配置文件

- @Configuration注解用于设定当前类为配置类
- @ComponentScan注解用于设定扫描路径，此注解只能添加一次，多个数据用数据格式
- 可以有多个配置类
  - 在某个配置类中可以使用@Import注解导入其他配置类
  - @Import(jdbcConfig.class)

```java
// SpringConfig.java
@Configuration
@ComponentScan("com.hardy")
public class SpringConfig {
}


// App.java
ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
```

3. 纯注解--bean管理

作用范围与生命周期管理

- 作用范围（用在类上）
    - @Scope注解
- 生命周期管理（用在方法上）
    - @PostConstruct
    - @PreDestroy

```java
@Repository("bookDao")
@Scope("prototype")
public class BookDaoImpl implements BookDao {
    @PostConstruct
    public void init() {
        System.out.println("BookDao init");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("BookDao destroy");
    }
}
```

4. 纯注解--依赖注入

注入方式——自动装配

(1) 引用类型

- 使用@Autowire注解（写在需要注入的变量上）
- 不需要再写setter
- @Qualifier("bookDao") ---- 多个相同类型bean时使用该注解指定bean

```java
// BookServiceImpl.java
@Autowired
private BookDao bookDao;
```

(2) 简单类型

```
@Value("hardy")
private String name;
```

一般配合配置文件使用

1. 首先在配置类SpringConfig.java中，使用@PropertySource注解。多个properties文件使用数组。不允许使用通配符

```
// SpringConfig.java
@Configuration
@ComponentScan("com.hardy")
@PropertySource("jdbc.properties")
public class SpringConfig {
}
```

2.

```
// BookDaoImpl.java
@Value("${name}")
private String name;
```

3.

```
// jdbc.properties
name=haaaardys
```

1.2.12 第三方bean管理

1. @Bean

可以新建另一个配置类文件（不能动别人jar包中的代码，bean写在配置类中）

```
@Configuration
public class SpringConfig {
    // 1.定义一个方法获得要管理的对象
    // 2.添加@Bean，表示当前方法的返回值是一个bean
    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("1234");
        return ds;
    }
}
```

2. 第三方bean依赖注入

- 引用类型：方法形参

  出现在形参中，就会自动装配。前提是必须有这个bean。

```
@Bean
public DataSource dataSource(BookDao bookDao){
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
    ds.setUsername("root");
    ds.setPassword("1234");
    return ds;
}
```

- 简单类型：成员变量

```
@Configuration
public class SpringConfig {
    @Value("com.mysql.jdbc.Driver")
    private String driver;

    @Value("jdbc:mysql://localhost:3306/spring_db")
    private String url;

    @Value("root")
    private String userName;

    @Value("1234")
    private String password;

    @Bean
    public DataSource dataSource (){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(userName);
        ds.setPassword(password);
        return ds;
    }
}
```

1.3 Spring整合

1.3.1 Spring整合MyBatis

1. pom.xml

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.0</version>
</dependency>
```

1.3.2 Spring整合JUnit

1.4 AOP

1.4.1 简介

- **AOP**（Aspect-Oriented Programming）面向切面编程，一种编程范式，指导开发者如何组织程序结构
- 作用：在不惊动原始设计的基础上为其进行功能增强
- SpringAOP的核心本质是采用**代理模式**实现的

1.4.2 AOP中的核心概念

- 连接点（JoinPoint）：程序执行过程中的任意位置，粒度为执行方法、抛出异常、设置变量等
  - 在Spring AOP中，理解为方法的执行
- **切入点**（Pointcut）：匹配连接点的式子
  - 在Spring AOP中，一个切入点可以只描述一个具体方法，也可以匹配多个方法。例：
    - 一个具体方法：com.hardy.dao包下的BookDao接口中的无形参无返回值方法
    - 匹配多个方法：所有save方法，所有get开头的方法，所有以Dao结尾的接口中的任意方法，所有带有一个参数的方法

- **通知**（Advice）：在切入点处执行的操作，也就是共性功能
  - 在Spring AOP中，功能最终以方法的形式呈现
- 通知类：定义通知的类
- **切面**（Aspect）：描述通知与切入点的对应关系

1.4.3 AOP快速入门案例

使用注解开发

1.导入坐标

- aop和aspectweaver，其中aop在导入spring-context时已导入

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
```

2. 在通知类里写通知

```
// MyAdvice.java

@Component
@Aspect
public class MyAdvice {
    @Pointcut("execution(void com.hardy.dao.BookDao.update())")
    private void pt(){}

    @Before("pt()")
    public void method(){
        System.out.println(System.currentTimeMillis());
    }
}
```

其中

- @Component -------- 告诉spring放入IOC容器
- @Aspect ----------- 指示spring不要当成普通bean，而是当成AOP来管理
- method() 方法就是通知，无参、无返回值、方法名任意。方法体内写具体的通知
- @Pointcut 注解写在一个无参、无返回值、方法名任意的私有空方法上。其中的参数表示程序执行到com.hardy.dao.BookDao.update()这个返回值是void（无返回值）的方法时执行通知方法。
- @Before("pt()") 表示前置通知，在切入点之前执行通知

3. 在spring配置类中开始aop

```
// SpringConfig.class
@Configuration
@ComponentScan("com.hardy")
@EnableAspectJAutoProxy
public class SpringConfig {
}
```

- @EnableAspectJAutoProxy -------- 开启aop
- @Configuration -------- 表明这是一个spring配置类（注解开发方法）
- ComponentScan("com.hardy") -------- 扫描com.hardy这个包下所有配置了bean 类或方法（ 类：@Component/@Repository/@Controller/@Service　方法：@bean）

1.4.4 AOP切入点表达式

1. 语法格式

> 动作关键字（访问修饰符 返回值 包名.类/接口名.方法名（参数）异常名）
>
> execution (public User com.hardy.service.UserService.findById (int))

- 动作关键字：描述切入点的行为动作，例如execution表示执行到指定切入点
- 访问修饰符：public等，可以省略
- 异常名：方法定义中抛出指定异常，可以省略

2. 通配符

可以使用通配符描述切入点。

- " * " ---------- 单个独立的任意符号，可以独立出现，也可以作为前缀或者后缀的匹配符出现

```
execution ( public * com.hardy.*.UserService.find* (*) )
```

匹配com.hardy包下任意包中的UserService类或接口中所有find开头的带有一个参数的方法

- ".." ---------- 多个连续的任意符号，可以独立出现，常用于简化包名与参数的书写

```
execution ( public User com.hardy.service.UserService.findById (..) )
```

匹配com包下的任意包中的UserService类或接口中所有名称为fingById的方法

- "+" ---------- 专用于匹配子类类型

```
execution ( * *..*Service+.* (..) )
```

> 要使用通配符，一定要按照规范进行开发

1.4.5 AOP通知类型

- 前置通知 @Before
- 后置通知 @After
- 环绕通知（重点）@Around
- 返回后通知（了解）@AfterReturning
- 抛出异常后通知（了解）@AfterThrowing

1. 环绕通知

```
// MyAdvice.java

// 原方法无返回值时
@Around("pt()")
public void func(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice");
    pjp.proceed();  // 表示对原始操作的调用
    System.out.println("around after advice");
}

// 原方法有返回值时
@Around("pt()")
public Object func(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice");
    Object ret = pjp.proceed();  // 表示对原始操作的调用，ret是原方法的返回值
    System.out.println("around after advice");
    return ret;
}
```

- 环绕通知必须依赖形参ProceedingJoinPoint才能实现对原始方法的调用，进而实现原始方法调用前后同时添加通知
    - 可以实现对原始方法的隔离
- 对原始方法的调用可以不接受返回值，通知方法设置成void即可，如果接受返回值，必须设定为Object
- 对原始方法的返回值如果是void类型，通知方法的返回值类型可以设置成void，也可以设置成Object
- 由于无法预知原始方法运行后是否会抛出异常，因此环绕通知必须抛出Throwable对象

```
Signature signature = pjp.getSignature();
signature.getDeclaringType(); // 原始方法所在全类名
signature.getName(); // 原始方法名称
```

1.4.6 AOP通知获取数据

- 获取切入点方法的参数
  - JoinPoint ： 适用于前置、后置、返回后、抛出异常后通知
  - ProceedingJoinPoint：适用于环绕通知
  - JoinPoint 是一个接口，ProceedingJoinPoint是其实现类
- 获取切入点方法的返回值
  - 返回后通知
  - 环绕通知
- 获取切入点方法的运行异常信息
  - 抛出异常后通知
  - 环绕通知

1. 获取切入点方法的参数

```
@Before("pt()")
public void before(JoinPoint jp){
    Object[] args = jp.getArgs();
    ...
}
```

```
@Around("pt()")
public void func(ProceedingJoinPoint pjp) throws Throwable {
    Object[] args = pjp.getArgs();
    ...
}
```

- 可以在通知里对原始的参数进行修改，防止错误的参数，提高健壮性

```
@Around("pt()")
public void func(ProceedingJoinPoint pjp) throws Throwable {
    Object[] args = pjp.getArgs();
    args[0] = 100;
    Object ret = pjp.proceed(args);
    ...
}
```

2. 获取切入点方法的返回值

```
@AfterReturning(value="pt()", returning="ret")
public void afterReturning(Object ret){
    System.out.println(ret);
}
```

- returning的值必须与形参名相同
- 形参里如果有JoinPoint类型的参数，必须要在第一位

3. 获取切入点方法的运行异常信息

```
@AfterThrowing(value="pt()", throwing="t")
public void afterThrowing(Throwable t){
    ...
}
```

1.5 Spring事务

事务作用：在数据层保障一系列的数据库操作同成功同失败

Spring事务作用：在数据层或**业务层**保障一系列的数据库操作同成功同失败

Spring事务实现 ---------- 通过一个接口：**PlatformTransactionManage**r，其实现类 DataSourceTransantionManager

1.5.1 简单案例

1. 添加Spring事务管理

- 使用**@Transactional**
- 通常添加到接口中而不会添加到实现类中，降低耦合。
- 除添加到方法上表示当前方法开始事务，也可以添加到接口上表示当前接口**所有方法**开启事务

```
public interface AccountService {
    @Transactional
    public void transfer(String out, String in, Double money);
}
```

2. 设置事务管理器

```
// jdbcConfig.java
@Bean
public PlatformTransactionManager transactionManager(DataSource datasource){
    DataSourceTransactionManager ptm = new DataSourceTransactionManager();
    ptm.setDataSourceManager(dataSource);
    return ptm;
}
```

- 事务管理区要根据实现技术进行选择
- MyBatis框架使用的是JDBC事务

3. 开启注解式事务驱动

- @EnableTransactionManager

```
// SpringConfig.java
@EnableTransactionManager
public class SpringConfig{

}
```

1.5.2 Spring事务角色

- 事务管理员：发起事务方，在Spring中通常指代业务层开启事务的方法
- 事务协调员：加入事务方，在Spring中通常指代数据层方法，也可以是业务层方法。

1.5.3 Srping事务属性

即@Transactional 写在括号内的属性值

1. rollbackFor

- 设置事务回滚异常（class）
- Srping事务遇到某些异常默认是不回滚的，如运行时异常。需要改变这种默认值时，

```
@Transactional(rollbackFor = {IOException.class})
```

2. rollbackForClassName

- 设置事务回滚异常（String）

3. noRollbackFor

- 设置事务不回滚异常（class）

4. noRollbackClassName

- 设置事务不回滚异常（String）

5. readOnly

- 设置是否为只读事务
- readOnly = true

6. timeout

- 设置事务超时时间
- timeout = -1 （永不超时）

7. propagation

- 设置事务传播行为
- 事务传播行为：事务协调员对事务管理员所携带事务的处理态度（要不要加入你的事务）

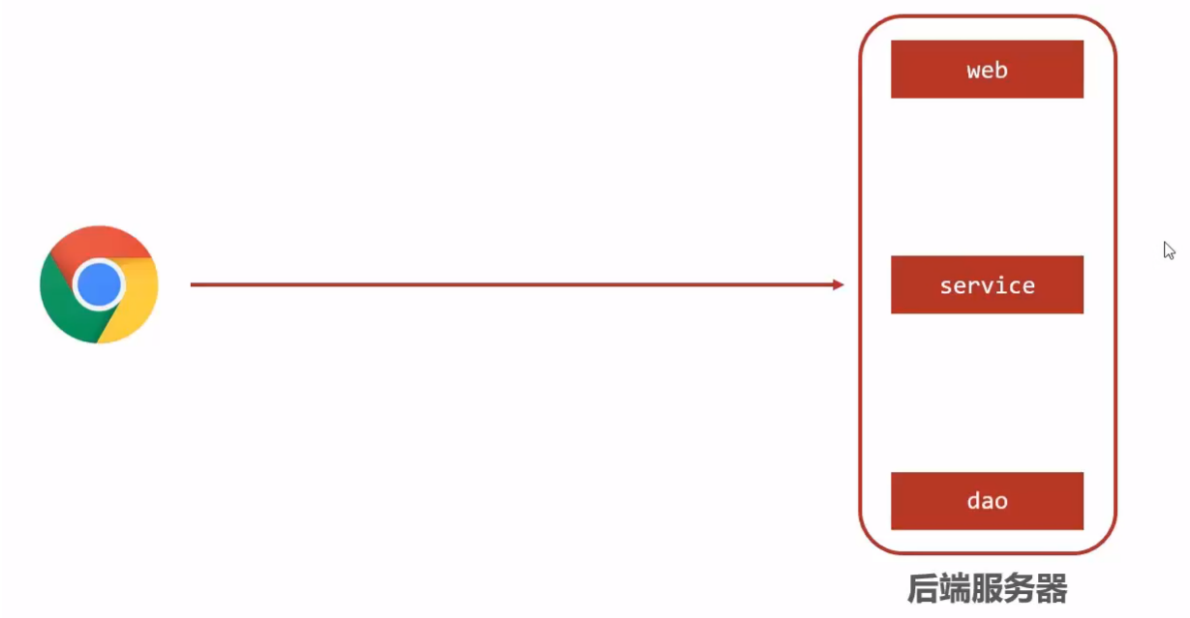
```
@Transactional(propagation = propagation.REQUIRES_NEW)    // 事务协调员单独开一个事务
```

二、SpringMVC

2.1 SpringMVC简介

2.1.1 请求响应模式演进过程

1. 三层架构

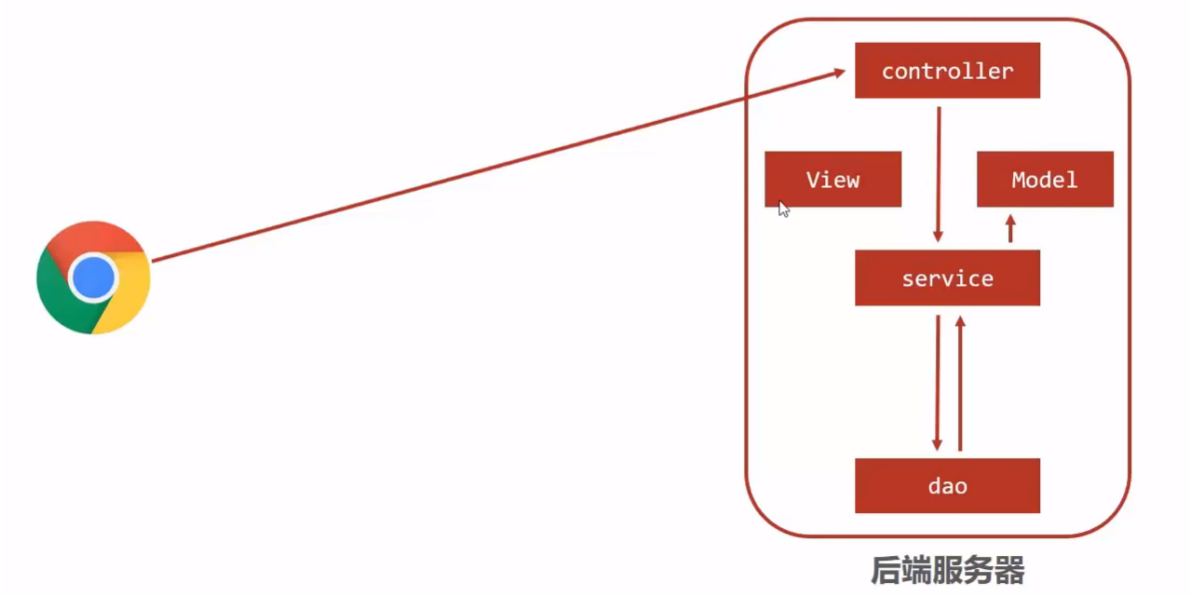


web：负责页面数据的收集以及产生页面

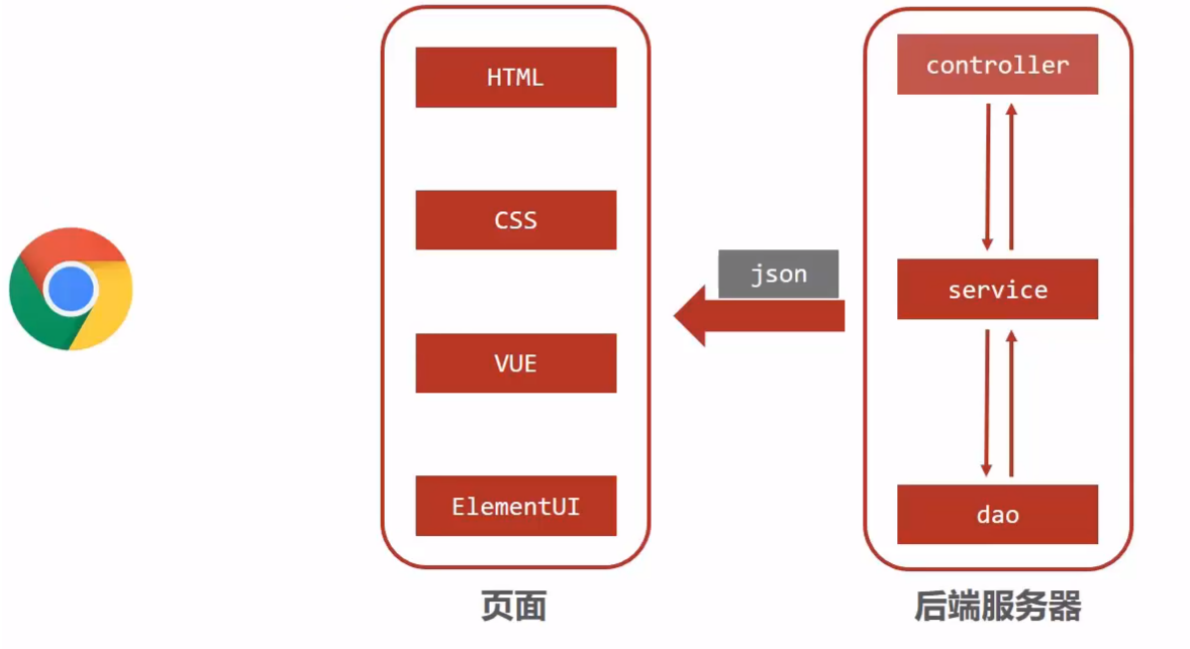
service：业务层，负责业务处理，如对数据进行一些处理

dao：数据的持久化，存、查数据等

2. MVC模式



3. 异步调用



2.1.2 SpringMVC概述

- SpringMVC是一种基于Java实现MVC模型的轻量级Web框架
- 优点
  - 使用简单，开发便捷（相较于Servlet）
  - 灵活性强

2.2 SpringMVC入门案例

- 使用Servlet技术开发web程序流程

1. 创建web工程（Maven结构）
2. 设置tomcat服务器，加载web工程（tomcat插件）
3. 导入坐标（Servlet）
4. 定义请求的功能类（UserServlet）
5. 设置请求映射（配置映射关系）

- 使用SpringMVC技术开发web程序流程

1. 创建web工程（Maven结构）
2. 设置tomcat服务器，加载web工程（tomcat插件）
3. 导入坐标（**SpringMVC**+Servlet）
4. 定义请求的功能类（**UserController**）
5. **设置请求映射（配置映射关系）**
6. 将SpringMVC设定加载到tomcat容器中