一、MyBatis

1、MyBatis简介

1.1、MyBatis历史

MyBatis最初是Apache的一个开源项目**iBatis**, 2010年6月这个项目由Apache Software Foundation迁移到了Google Code。随着开发团队转投Google Code旗下， iBatis3.x正式更名为MyBatis。代码于2013年11月迁移到Github。

iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。 iBatis提供的持久层框架包括SQL Maps和Data Access Objects（DAO）。

1.2、MyBatis特性

1） MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架

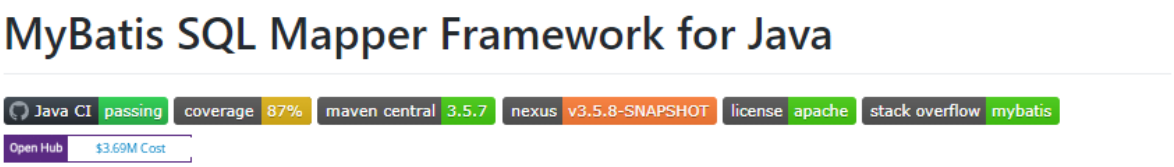
2） MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集

3） MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录

4） MyBatis 是一个 半自动的ORM（Object Relation Mapping）框架

1.3、MyBatis下载

MyBatis下载地址：https://github.com/mybatis/mybatis-3



The MyBatis SQL mapper framework makes it easier to use a relational database with object-oriented applications. MyBatis couples objects with stored procedures or SQL statements using a XML descriptor or annotations. Simplicity is the biggest advantage of the MyBatis data mapper over object relational mapping tools.

Essentials

- See the docs
- Download Latest 点这里下载最新版
- Download Snapshot

1.4、和其它持久化层技术对比

- JDBC
  - SQL 夹杂在Java代码中耦合度高，导致硬编码内伤
  - 维护不易且实际开发需求中 SQL 有变化，频繁修改的情况多见
  - 代码冗长，开发效率低
- Hibernate 和 JPA
  - 操作简便，开发效率高
  - 程序中的长难复杂 SQL 需要绕过框架

    - 内部自动生产的 SQL，不容易做特殊优化
    - 基于全映射的全自动框架，大量字段的 POJO 进行部分映射时比较困难。
    - 反射操作太多，导致数据库性能下降
- MyBatis
    - 轻量级，性能出色
    - SQL 和 Java 编码分开，功能边界清晰。Java代码专注业务、SQL语句专注数据
    - 开发效率稍逊于HIbernate，但是完全能够接受

2、搭建MyBatis

2.1、开发环境

IDE：idea 2019.2

构建工具：maven 3.5.4

MySQL版本：MySQL 8

MyBatis版本：MyBatis 3.5.7

> MySQL不同版本的注意事项
>
> 1、驱动类driver-class-name
>
> MySQL 5版本使用jdbc5驱动，驱动类使用：com.mysql.jdbc.Driver
>
> MySQL 8版本使用jdbc8驱动，驱动类使用：com.mysql.cj.jdbc.Driver
>
> 2、连接地址url
>
> MySQL 5版本的url：
>
> jdbc:mysql://localhost:3306/ssm
>
> MySQL 8版本的url：
>
> jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
>
> 否则运行测试用例报告如下错误：
>
> java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more

2.2、创建maven工程

①打包方式：jar

②引入依赖

```
<dependencies>
    <!-- Mybatis核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
```

```
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
</dependencies>
```

2.3、创建MyBatis的核心配置文件

> 习惯上命名为mybatis-config.xml，这个文件名仅仅只是建议，并非强制要求。将来整合Spring之后，这个配置文件可以省略，所以大家操作时可以直接复制、粘贴。
>
> 核心配置文件主要用于配置连接数据库的环境以及MyBatis的全局配置信息
>
> 核心配置文件存放的位置是src/main/resources目录下

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--设置连接数据库的环境-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/ssm?
serverTimezone=UTC"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <!--引入映射文件-->
    <mappers>
        <package name="mappers/UserMapper.xml"/>
    </mappers>
</configuration>
```

2.4、创建mapper接口

> MyBatis中的mapper接口相当于以前的dao。但是区别在于，mapper仅仅是接口，我们不需要提供实现类。

```
public interface UserMapper {

    /**
     * 添加用户信息
     */
    int insertUser();

}
```

2.5、创建MyBatis的映射文件

相关概念：**ORM**（**O**bject **R**elationship **M**apping）对象关系映射。

- 对象：Java的实体类对象
- 关系：关系型数据库
- 映射：二者之间的对应关系

Java概念	数据库概念
类	表
属性	字段/列
对象	记录/行

> 1、映射文件的命名规则：
>
> 表所对应的实体类的类名+Mapper.xml
>
> 例如：表t_user，映射的实体类为User，所对应的映射文件为UserMapper.xml
>
> 因此一个映射文件对应一个实体类，对应一张表的操作
>
> MyBatis映射文件用于编写SQL，访问以及操作表中的数据
>
> MyBatis映射文件存放的位置是src/main/resources/mappers目录下
>
> 2、 MyBatis中可以面向接口操作数据，要保证两个一致：
>
> a>mapper接口的全类名和映射文件的命名空间（namespace）保持一致
>
> b>mapper接口中方法的方法名和映射文件中编写SQL的标签的id属性保持一致

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.atguigu.mybatis.mapper.UserMapper">

    <!--int insertUser();-->
    <insert id="insertUser">
        insert into t_user values(null,'admin','123456',23,'男','12345@qq.com')
    </insert>

</mapper>
```

2.6、通过junit测试功能

```
//读取MyBatis的核心配置文件
InputStream is = Resources.getResourceAsStream("mybatis-config.xml");
//创建SqlSessionFactoryBuilder对象
SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
//通过核心配置文件所对应的字节输入流创建工厂类SqlSessionFactory，生产SqlSession对象
SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);
//创建SqlSession对象，此时通过SqlSession对象所操作的sql都必须手动提交或回滚事务
//SqlSession sqlSession = sqlSessionFactory.openSession();
//创建SqlSession对象，此时通过SqlSession对象所操作的sql都会自动提交
SqlSession sqlSession = sqlSessionFactory.openSession(true);
//通过代理模式创建UserMapper接口的代理实现类对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
//调用UserMapper接口中的方法，就可以根据UserMapper的全类名匹配元素文件，通过调用的方法名匹配
映射文件中的SQL标签，并执行标签中的SQL语句
int result = userMapper.insertUser();
//sqlSession.commit();
System.out.println("结果："+result);
```

> - SqlSession：代表Java程序和**数据库**之间的**会话**。（HttpSession是Java程序和浏览器之间的会话）
> - SqlSessionFactory：是“生产”SqlSession的“工厂”。
> - 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象。

2.7、加入log4j日志功能

①加入依赖

```
<!-- log4j日志 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

②加入log4j的配置文件

> log4j的配置文件名为log4j.xml，存放的位置是src/main/resources目录下

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd HH:mm:ss,SSS}
%m  (%F:%L) \n" />
        </layout>
    </appender>
    <logger name="java.sql">
        <level value="debug" />
    </logger>
    <logger name="org.apache.ibatis">
```

```
        <level value="info" />
    </logger>
    <root>
        <level value="debug" />
        <appender-ref ref="STDOUT" />
    </root>
</log4j:configuration>
```

> **日志的级别**
>
> FATAL(致命)>ERROR(错误)>WARN(警告)>INFO(信息)>DEBUG(调试)
>
> 从左到右打印的内容越来越详细

3、核心配置文件详解

> 核心配置文件中的标签必须按照固定的顺序：
>
> properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,environments?,databaseIdProvider?,mappers?

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--
        MyBatis核心配置文件中，标签的顺序：
        properties?,settings?,typeAliases?,typeHandlers?,
        objectFactory?,objectWrapperFactory?,reflectorFactory?,
        plugins?,environments?,databaseIdProvider?,mappers?
    -->

    <!--引入properties文件-->
    <properties resource="jdbc.properties" />

    <!--设置类型别名-->
    <typeAliases>
        <!--
            typeAlias：设置某个类型的别名
            属性：
                type：设置需要设置别名的类型
                alias：设置某个类型的别名，若不设置该属性，那么该类型拥有默认的别名，即类名且不区分大小写
        -->
        <!--<typeAlias type="com.atguigu.mybatis.pojo.User"></typeAlias>-->
        <!--以包为单位，将包下所有的类型设置默认的类型别名，即类名且不区分大小写-->
        <package name="com.atguigu.mybatis.pojo"/>
    </typeAliases>

    <!--
        environments：配置多个连接数据库的环境
        属性：
            default：设置默认使用的环境的id
    -->
    <environments default="development">
```

```
        <!--
            environment：配置某个具体的环境
            属性：
                id：表示连接数据库的环境的唯一标识，不能重复
        -->
        <environment id="development">
            <!--
                transactionManager：设置事务管理方式
                属性：
                    type="JDBC|MANAGED"
                    JDBC：表示当前环境中，执行SQL时，使用的是JDBC中原生的事务管理方式，事务的提交或回滚需要手动处理
                    MANAGED：被管理，例如Spring
            -->
            <transactionManager type="JDBC"/>
            <!--
                dataSource：配置数据源
                属性：
                    type：设置数据源的类型
                    type="POOLED|UNPOOLED|JNDI"
                    POOLED：表示使用数据库连接池缓存数据库连接
                    UNPOOLED：表示不使用数据库连接池
                    JNDI：表示使用上下文中的数据源
            -->
            <dataSource type="POOLED">
                <!--设置连接数据库的驱动-->
                <property name="driver" value="${jdbc.driver}"/>
                <!--设置连接数据库的连接地址-->
                <property name="url" value="${jdbc.url}"/>
                <!--设置连接数据库的用户名-->
                <property name="username" value="${jdbc.username}"/>
                <!--设置连接数据库的密码-->
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>

        <environment id="test">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/ssmserverTimezone=UTC"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <!--引入映射文件-->
    <mappers>
        <!--<mapper resource="mappers/UserMapper.xml"/>-->
        <!--
            以包为单位引入映射文件
            要求：
            1、mapper接口所在的包要和映射文件所在的包一致
            2、mapper接口要和映射文件的名字一致
        -->
        <package name="com.atguigu.mybatis.mapper"/>
    </mappers>
```

```
</configuration>
```

4、MyBatis的增删改查

4.1、新增

```
<!--int insertUser();-->
<insert id="insertUser">
    insert into t_user values(null,'admin','123456',23,'男')
</insert>
```

4.2、删除

```
<!--int deleteUser();-->
<delete id="deleteUser">
    delete from t_user where id = 7
</delete>
```

4.3、修改

```
<!--int updateUser();-->
<update id="updateUser">
    update t_user set username='ybc',password='123' where id = 6
</update>
```

4.4、查询一个实体类对象

```
<!--User getUserById();-->
<select id="getUserById" resultType="com.atguigu.mybatis.bean.User">
    select * from t_user where id = 2
</select>
```

4.5、查询list集合

```
<!--List<User> getUserList();-->
<select id="getUserList" resultType="com.atguigu.mybatis.bean.User">
    select * from t_user
</select>
```

> 注意：
>
> 1、查询的标签select必须设置属性resultType或resultMap，用于设置实体类和数据库表的映射关系
>
> resultType：自动映射，用于属性名和表中字段名一致的情况
>
> resultMap：自定义映射，用于一对多或多对一或字段名和属性名不一致的情况

5、MyBatis获取参数值的两种方式

> MyBatis获取参数值的两种方式：**${}**和**#{}**

> ${}的本质就是字符串拼接，#{}的本质就是占位符赋值
>
> ${}使用字符串拼接的方式拼接sql，若为字符串类型或日期类型的字段进行赋值时，需要手动加单引
>
> 号；但是#{}使用占位符赋值的方式拼接sql，此时为字符串类型或日期类型的字段进行赋值时，可以自
>
> 动添加单引号

5.1、单个字面量类型的参数

> 若mapper接口中的方法参数为单个的字面量类型
>
> 此时可以使用${}和#{}以任意的名称获取参数的值，注意${}需要手动加单引号

5.2、多个字面量类型的参数

> 若mapper接口中的方法参数为多个时
>
> 此时MyBatis会自动将这些参数放在一个map集合中，以arg0,arg1...为键，以参数为值；以
>
> param1,param2...为键，以参数为值；因此只需要通过${}和#{}访问map集合的键就可以获取相对应的
>
> 值，注意${}需要手动加单引号

5.3、map集合类型的参数

> 若mapper接口中的方法需要的参数为多个时，此时可以手动创建map集合，将这些数据放在map中
>
> 只需要通过${}和#{}访问map集合的键就可以获取相对应的值，注意${}需要手动加单引号

5.4、实体类类型的参数

> 若mapper接口中的方法参数为实体类对象时
>
> 此时可以使用${}和#{}，通过访问实体类对象中的属性名获取属性值，注意${}需要手动加单引号

5.5、使用@Param标识参数

> 可以通过@Param注解标识mapper接口中的方法参数
>
> 此时，会将这些参数放在map集合中，以@Param注解的value属性值为键，以参数为值；以
>
> param1,param2...为键，以参数为值；只需要通过${}和#{}访问map集合的键就可以获取相对应的值，
>
> 注意${}需要手动加单引号

6、MyBatis的各种查询功能

6.1、查询一个实体类对象

```java
/**
 * 根据用户id查询用户信息
 * @param id
 * @return
 */
User getUserById(@Param("id") int id);
```

```xml
<!--User getUserById(@Param("id") int id);-->
<select id="getUserById" resultType="User">
    select * from t_user where id = #{id}
</select>
```

6.2、查询一个list集合

```java
/**
 * 查询所有用户信息
 * @return
 */
List<User> getUserList();
```

```xml
<!--List<User> getUserList();-->
<select id="getUserList" resultType="User">
    select * from t_user
</select>
```

> 当查询的数据为多条时，不能使用实体类作为返回值，否则会抛出异常TooManyResultsException；但是若查询的数据只有一条，可以使用实体类或集合作为返回值

6.3、查询单个数据

```java
/**
 * 查询用户的总记录数
 * @return
 * 在MyBatis中，对于Java中常用的类型都设置了类型别名
 * 例如： java.lang.Integer-->int|integer
 * 例如： int-->_int|_integer
 * 例如： Map-->map,List-->list
 */
int getCount();
```

```xml
<!--int getCount();-->
<select id="getCount" resultType="_integer">
    select count(id) from t_user
</select>
```

6.4、查询一条数据为map集合

```
/**
 * 根据用户id查询用户信息为map集合
 * @param id
 * @return
 */
Map<String, Object> getUserToMap(@Param("id") int id);
```

```
<!--Map<String, Object> getUserToMap(@Param("id") int id);-->
<!--结果： {password=123456, sex=男 , id=1, age=23, username=admin}-->
<select id="getUserToMap" resultType="map">
    select * from t_user where id = #{id}
</select>
```

6.5、查询多条数据为map集合

①方式一

```
/**
 * 查询所有用户信息为map集合

 * @return
 * 将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，此时可以将这些map放在一个list集合中获取
 */
List<Map<String, Object>> getAllUserToMap();
```

```
<!--Map<String, Object> getAllUserToMap();-->
<select id="getAllUserToMap" resultType="map">
    select * from t_user
</select>
```

②方式二

```
/**
 * 查询所有用户信息为map集合

 * @return
 * 将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，并且最终要以一个map的方式返回数据，此时需要通过@MapKey注解设置map集合的键，值是每条数据所对应的map集合
 */
@MapKey("id")
Map<String, Object> getAllUserToMap();
```

```
<!--Map<String, Object> getAllUserToMap();-->
<!--
{
    1={password=123456, sex=男, id=1, age=23, username=admin},
    2={password=123456, sex=男, id=2, age=23, username=张三},
    3={password=123456, sex=男, id=3, age=23, username=张三}
}
-->
<select id="getAllUserToMap" resultType="map">
    select * from t_user
</select>
```

7、特殊SQL的执行

7.1、模糊查询

```
/**
 * 测试模糊查询
 * @param mohu
 * @return
 */
List<User> testMohu(@Param("mohu") String mohu);
```

```
<!--List<User> testMohu(@Param("mohu") String mohu);-->
<select id="testMohu" resultType="User">
    <!--select * from t_user where username like '%${mohu}%'-->
    <!--select * from t_user where username like concat('%',#{mohu},'%')-->
    select * from t_user where username like "%"#{mohu}"%"
</select>
```

7.2、批量删除

```
/**
 * 批量删除
 * @param ids
 * @return
 */
int deleteMore(@Param("ids") String ids);
```

```
<!--int deleteMore(@Param("ids") String ids);-->
<delete id="deleteMore">
    delete from t_user where id in (${ids})
</delete>
```

7.3、动态设置表名

```
/**
 * 动态设置表名，查询所有的用户信息
 * @param tableName
 * @return
 */
List<User> getAllUser(@Param("tableName") String tableName);
```

```
<!--List<User> getAllUser(@Param("tableName") String tableName);-->
<select id="getAllUser" resultType="User">
    select * from ${tableName}
</select>
```

7.4、添加功能获取自增的主键

> 场景模拟：
>
> t_clazz(clazz_id,clazz_name)
>
> t_student(student_id,student_name,clazz_id)
>
> 1、添加班级信息
>
> 2、获取新添加的班级的id
>
> 3、为班级分配学生，即将某学的班级id修改为新添加的班级的id

```
/**
 * 添加用户信息
 * @param user
 * @return
 * useGeneratedKeys：设置使用自增的主键
 * keyProperty：因为增删改有统一的返回值是受影响的行数，因此只能将获取的自增的主键放在传输的参数user对象的某个属性中
 */
int insertUser(User user);
```

```
<!--int insertUser(User user);-->
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user values(null,#{username},#{password},#{age},#{sex})
</insert>
```

8、自定义映射resultMap

8.1、resultMap处理字段和属性的映射关系

> 若字段名和实体类中的属性名不一致，则可以通过resultMap设置自定义映射

```
<!--
    resultMap：设置自定义映射
    属性：
    id：表示自定义映射的唯一标识
    type：查询的数据要映射的实体类的类型
    子标签：
    id：设置主键的映射关系
    result：设置普通字段的映射关系
```

```
    association：设置多对一的映射关系
    collection：设置一对多的映射关系
    属性：
    property：设置映射关系中实体类中的属性名
    column：设置映射关系中表中的字段名
-->
<resultMap id="userMap" type="User">
    <id property="id" column="id"></id>
    <result property="userName" column="user_name"></result>
    <result property="password" column="password"></result>
    <result property="age" column="age"></result>
    <result property="sex" column="sex"></result>
</resultMap>
<!--List<User> testMohu(@Param("mohu") String mohu);-->
<select id="testMohu" resultMap="userMap">
    <!--select * from t_user where username like '%${mohu}%'-->
    select id,user_name,password,age,sex from t_user where user_name like
concat('%',#{mohu},'%')
</select>
```

> 若字段名和实体类中的属性名不一致，但是字段名符合数据库的规则（使用_），实体类中的属性名符合Java的规则（使用驼峰）
>
> 此时也可通过以下两种方式处理字段名和实体类中的属性的映射关系
>
> a>可以通过为字段起别名的方式，保证和实体类中的属性名保持一致
>
> b>可以在MyBatis的核心配置文件中设置一个全局配置信息mapUnderscoreToCamelCase，可以在查询表中数据时，自动将_类型的字段名转换为驼峰
>
> 例如：字段名user_name，设置了mapUnderscoreToCamelCase，此时字段名就会转换为
>
> userName

8.2、多对一映射处理

> 场景模拟：
>
> 查询员工信息以及员工所对应的部门信息

8.2.1、级联方式处理映射关系

```
<resultMap id="empDeptMap" type="Emp">
    <id column="eid" property="eid"></id>
    <result column="ename" property="ename"></result>
    <result column="age" property="age"></result>
    <result column="sex" property="sex"></result>
    <result column="did" property="dept.did"></result>
    <result column="dname" property="dept.dname"></result>
</resultMap>
<!--Emp getEmpAndDeptByEid(@Param("eid") int eid);-->
<select id="getEmpAndDeptByEid" resultMap="empDeptMap">
    select emp.*,dept.* from t_emp emp left join t_dept dept on emp.did =
dept.did where emp.eid = #{eid}
</select>
```

8.2.2、使用association处理映射关系

```
<resultMap id="empDeptMap" type="Emp">
    <id column="eid" property="eid"></id>
    <result column="ename" property="ename"></result>
    <result column="age" property="age"></result>
    <result column="sex" property="sex"></result>
    <association property="dept" javaType="Dept">
        <id column="did" property="did"></id>
        <result column="dname" property="dname"></result>
    </association>
</resultMap>
<!--Emp getEmpAndDeptByEid(@Param("eid") int eid);-->
<select id="getEmpAndDeptByEid" resultMap="empDeptMap">
    select emp.*,dept.* from t_emp emp left join t_dept dept on emp.did =
dept.did where emp.eid = #{eid}
</select>
```

8.2.3、分步查询

①查询员工信息

```
/**
 * 通过分步查询查询员工信息
 * @param eid
 * @return
 */
Emp getEmpByStep(@Param("eid") int eid);
```

```
<resultMap id="empDeptStepMap" type="Emp">
    <id column="eid" property="eid"></id>
    <result column="ename" property="ename"></result>
    <result column="age" property="age"></result>
    <result column="sex" property="sex"></result>
    <!--
        select：设置分步查询，查询某个属性的值的sql的标识（namespace.sqlId）
        column：将sql以及查询结果中的某个字段设置为分步查询的条件
    -->
    <association property="dept"
select="com.atguigu.MyBatis.mapper.DeptMapper.getEmpDeptByStep" column="did">
</association>
</resultMap>
<!--Emp getEmpByStep(@Param("eid") int eid);-->
<select id="getEmpByStep" resultMap="empDeptStepMap">
    select * from t_emp where eid = #{eid}
</select>
```

②根据员工所对应的部门id查询部门信息

```
/**
 * 分步查询的第二步： 根据员工所对应的did查询部门信息
 * @param did
 * @return
 */
Dept getEmpDeptByStep(@Param("did") int did);
```

```
<!--Dept getEmpDeptByStep(@Param("did") int did);-->
<select id="getEmpDeptByStep" resultType="Dept">
    select * from t_dept where did = #{did}
</select>
```

8.3、一对多映射处理

8.3.1、collection

```
/**
 * 根据部门id查新部门以及部门中的员工信息
 * @param did
 * @return
 */
Dept getDeptEmpByDid(@Param("did") int did);
```

```
<resultMap id="deptEmpMap" type="Dept">
    <id property="did" column="did"></id>
    <result property="dname" column="dname"></result>
    <!--
        ofType：设置collection标签所处理的集合属性中存储数据的类型
    -->
    <collection property="emps" ofType="Emp">
        <id property="eid" column="eid"></id>
        <result property="ename" column="ename"></result>
        <result property="age" column="age"></result>
        <result property="sex" column="sex"></result>
    </collection>
</resultMap>
<!--Dept getDeptEmpByDid(@Param("did") int did);-->
<select id="getDeptEmpByDid" resultMap="deptEmpMap">
    select dept.*,emp.* from t_dept dept left join t_emp emp on dept.did =
emp.did where dept.did = #{did}
</select>
```

8.3.2、分步查询

①查询部门信息

```
/**
 * 分步查询部门和部门中的员工
 * @param did
 * @return
 */
Dept getDeptByStep(@Param("did") int did);
```

```
<resultMap id="deptEmpStep" type="Dept">
    <id property="did" column="did"></id>
    <result property="dname" column="dname"></result>
    <collection property="emps" fetchType="eager"
select="com.atguigu.MyBatis.mapper.EmpMapper.getEmpListByDid" column="did">
    </collection>
</resultMap>
<!--Dept getDeptByStep(@Param("did") int did);-->
<select id="getDeptByStep" resultMap="deptEmpStep">
    select * from t_dept where did = #{did}
</select>
```

②根据部门id查询部门中的所有员工

```
/**
 * 根据部门id查询员工信息
 * @param did
 * @return
 */
List<Emp> getEmpListByDid(@Param("did") int did);
```

```
<!--List<Emp> getEmpListByDid(@Param("did") int did);-->
<select id="getEmpListByDid" resultType="Emp">
    select * from t_emp where did = #{did}
</select>
```

> 分步查询的优点：可以实现延迟加载
>
> 但是必须在核心配置文件中设置全局配置信息：
>
> lazyLoadingEnabled：延迟加载的全局开关。当开启时，所有关联对象都会延迟加载
>
> aggressiveLazyLoading：当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载
>
> 此时就可以实现按需加载，获取的数据是什么，就只会执行相应的sql。此时可通过association和collection中的fetchType属性设置当前的分步查询是否使用延迟加载， fetchType="lazy(延迟加载)|eager(立即加载)"

9、动态SQL

> Mybatis框架的动态SQL技术是一种根据特定条件动态拼装SQL语句的功能，它存在的意义是为了解决 拼接SQL语句字符串时的痛点问题。

9.1、if

> if标签可通过test属性的表达式进行判断，若表达式的结果为true，则标签中的内容会执行；反之标签中的内容不会执行

```
<!--List<Emp> getEmpListByCondition(Emp emp);-->
<select id="getEmpListByMoreTJ" resultType="Emp">
    select * from t_emp where 1=1
    <if test="ename != '' and ename != null">
        and ename = #{ename}
    </if>
    <if test="age != '' and age != null">
        and age = #{age}
    </if>
    <if test="sex != '' and sex != null">
        and sex = #{sex}
    </if>
</select>
```

9.2、where

> where和if一般结合使用：
>
> a>若where标签中的if条件都不满足，则where标签没有任何功能，即不会添加where关键字
>
> b>若where标签中的if条件满足，则where标签会自动添加where关键字，并将条件最前方多余的and去掉
>
> 注意：where标签不能去掉条件最后多余的and

```
<select id="getEmpListByMoreTJ2" resultType="Emp">
    select * from t_emp
    <where>
        <if test="ename != '' and ename != null">
            ename = #{ename}
        </if>
        <if test="age != '' and age != null">
            and age = #{age}
        </if>
        <if test="sex != '' and sex != null">
            and sex = #{sex}
        </if>
    </where>
</select>
```

9.3、trim

> trim用于去掉或添加标签中的内容
>
> 常用属性：
>
> prefix：在trim标签中的内容的前面添加某些内容
>
> prefixOverrides：在trim标签中的内容的前面去掉某些内容
>
> suffix：在trim标签中的内容的后面添加某些内容
>
> suffixOverrides：在trim标签中的内容的后面去掉某些内容

```
<select id="getEmpListByMoreTJ" resultType="Emp">
    select * from t_emp
    <trim prefix="where" suffixOverrides="and">
        <if test="ename != '' and ename != null">
```

```
            ename = #{ename} and
        </if>
        <if test="age != '' and age != null">
            age = #{age} and
        </if>
        <if test="sex != '' and sex != null">
            sex = #{sex}
        </if>
    </trim>
</select>
```

9.4、choose、when、otherwise

> choose、when、 otherwise相当于if...else if..else

```
<!--List<Emp> getEmpListByChoose(Emp emp);-->
<select id="getEmpListByChoose" resultType="Emp">
    select <include refid="empColumns"></include> from t_emp
    <where>
        <choose>
            <when test="ename != '' and ename != null">
                ename = #{ename}
            </when>
            <when test="age != '' and age != null">
                age = #{age}
            </when>
            <when test="sex != '' and sex != null">
                sex = #{sex}
            </when>
            <when test="email != '' and email != null">
                email = #{email}
            </when>
        </choose>
    </where>
</select>
```

9.5、foreach

```
<!--int insertMoreEmp(List<Emp> emps);-->
<insert id="insertMoreEmp">
    insert into t_emp values
    <foreach collection="emps" item="emp" separator=",">
        (null,#{emp.ename},#{emp.age},#{emp.sex},#{emp.email},null)
    </foreach>
</insert>
<!--int deleteMoreByArray(int[] eids);-->
<delete id="deleteMoreByArray">
    delete from t_emp where
    <foreach collection="eids" item="eid" separator="or">
        eid = #{eid}
    </foreach>
</delete>
<!--int deleteMoreByArray(int[] eids);-->
<delete id="deleteMoreByArray">
    delete from t_emp where eid in
    <foreach collection="eids" item="eid" separator="," open="(" close=")">
```

```
        #{eid}
    </foreach>
</delete>
```

9.6、SQL片段

> sql片段，可以记录一段公共sql片段，在使用的地方通过include标签进行引入

```
<sql id="empColumns">
    eid,ename,age,sex,did
</sql>
select <include refid="empColumns"></include> from t_emp
```

10、MyBatis的缓存

10.1、MyBatis的一级缓存

一级缓存是SqlSession级别的，通过同一个SqlSession查询的数据会被缓存，下次查询相同的数据，就会从缓存中直接获取，不会从数据库重新访问

使一级缓存失效的四种情况：

1) 不同的SqlSession对应不同的一级缓存

2) 同一个SqlSession但是查询条件不同

3) 同一个SqlSession两次查询期间执行了任何一次增删改操作

4) 同一个SqlSession两次查询期间手动清空了缓存

10.2、MyBatis的二级缓存

二级缓存是SqlSessionFactory级别，通过同一个SqlSessionFactory创建的SqlSession查询的结果会被缓存；此后若再次执行相同的查询语句，结果就会从缓存中获取

二级缓存开启的条件：

a>在核心配置文件中，设置全局配置属性cacheEnabled="true"，默认为true，不需要设置

b>在映射文件中设置标签<cache/>

c>二级缓存必须在SqlSession关闭或提交之后有效

d>查询的数据所转换的实体类类型必须实现序列化的接口

使二级缓存失效的情况：

两次查询之间执行了任意的增删改，会使一级和二级缓存同时失效

10.3、二级缓存的相关配置

在mapper配置文件中添加的cache标签可以设置一些属性：

①eviction属性：缓存回收策略，默认的是 LRU。

LRU（Least Recently Used） – 最近最少使用的：移除最长时间不被使用的对象。

FIFO（First in First out） – 先进先出：按对象进入缓存的顺序来移除它们。

SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。

WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

②flushInterval属性：刷新间隔，单位毫秒

默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新

③size属性：引用数目，正整数

代表缓存最多可以存储多少个对象，太大容易导致内存溢出

④readOnly属性：只读， true/false

true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了 很重要的性能优势。

false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

10.4、MyBatis缓存查询的顺序

先查询二级缓存，因为二级缓存中可能会有其他程序已经查出来的数据，可以拿来直接使用。

如果二级缓存没有命中，再查询一级缓存

如果一级缓存也没有命中，则查询数据库

SqlSession关闭之后，一级缓存中的数据会写入二级缓存

10.5、整合第三方缓存EHCache

10.5.1、添加依赖

```
<!-- Mybatis EHCache整合包 -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
<!-- slf4j日志门面的一个具体实现 -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

10.5.2、各jar包功能

jar包名称	作用
mybatis-ehcache	Mybatis和EHCache的整合包
ehcache	EHCache核心包
slf4j-api	SLF4J日志门面包
logback-classic	支持SLF4J门面接口的一个具体实现

10.5.3、创建EHCache的配置文件ehcache.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- 磁盘保存路径 -->
    <diskStore path="D:\atguigu\ehcache"/>
    <defaultCache
        maxElementsInMemory="1000"
        maxElementsOnDisk="10000000"
        eternal="false"
        overflowToDisk="true"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

10.5.4、设置二级缓存的类型

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

10.5.5、加入logback日志

> 存在SLF4J时，作为简易日志的log4j将失效，此时我们需要借助SLF4J的具体实现logback来打印日志。 创建logback的配置文件logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <!-- 指定日志输出的位置 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 日志输出的格式 -->
            <!-- 按照顺序分别是： 时间、日志级别、线程名称、打印日志的类、日志主体内容、换行
-->
            <pattern>[%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger]
[%msg]%n</pattern>
        </encoder>
    </appender>

    <!-- 设置全局日志级别。日志级别按顺序分别是： DEBUG、INFO、WARN、ERROR -->
    <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志。 -->
    <root level="DEBUG">
        <!-- 指定打印日志的appender，这里通过“STDOUT”引用了前面配置的appender -->
        <appender-ref ref="STDOUT" />
    </root>

    <!-- 根据特殊需求指定局部日志级别 -->
    <logger name="com.atguigu.crowd.mapper" level="DEBUG"/>
</configuration>
```

10.5.6、EHCache配置文件说明

属性名	是否必须	作用
maxElementsInMemory	是	在内存中缓存的element的最大数目
maxElementsOnDisk	是	在磁盘上缓存的element的最大数目，若是0表示无穷大
eternal	是	设定缓存的elements是否永远不过期。 如果为true，则缓存的数据始终有效， 如果为false那么还要根据timeToIdleSeconds、timeToLiveSeconds判断
overflowToDisk	是	设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上
timeToIdleSeconds	否	当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时， 这些数据便会删除，默认值是0,也就是可闲置时间无穷大
timeToLiveSeconds	否	缓存element的有效生命期，默认是0.,也就是element存活时间无穷大
diskSpoolBufferSizeMB	否	DiskStore(磁盘缓存)的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区
diskPersistent	否	在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。
diskExpiryThreadIntervalSeconds	否	磁盘缓存的清理线程运行间隔，默认是120秒。每个120s， 相应的线程会进行一次EhCache中数据的清理工作
memoryStoreEvictionPolicy	否	当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。 默认是LRU （最近最少使用），可选的有LFU （最不常使用）和FIFO （先进先出）

11、MyBatis的逆向工程

> 正向工程：先创建Java实体类，由框架负责根据实体类生成数据库表。 Hibernate是支持正向工程的。
>
> 逆向工程：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：
>
> - Java实体类
> - Mapper接口
> - Mapper映射文件

11.1、创建逆向工程的步骤

①添加依赖和插件

```
<!-- 依赖MyBatis核心包 -->
```

```
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- log4j日志 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
</dependencies>

<!-- 控制Maven在构建过程中相关配置 -->
<build>

    <!-- 构建过程中用到的插件 -->
    <plugins>

        <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.0</version>

            <!-- 插件的依赖 -->
            <dependencies>

                <!-- 逆向工程的核心依赖 -->
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifactId>
                    <version>1.3.2</version>
                </dependency>

                <!-- MySQL驱动 -->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>8.0.16</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
```

```
</build>
```

②创建MyBatis的核心配置文件

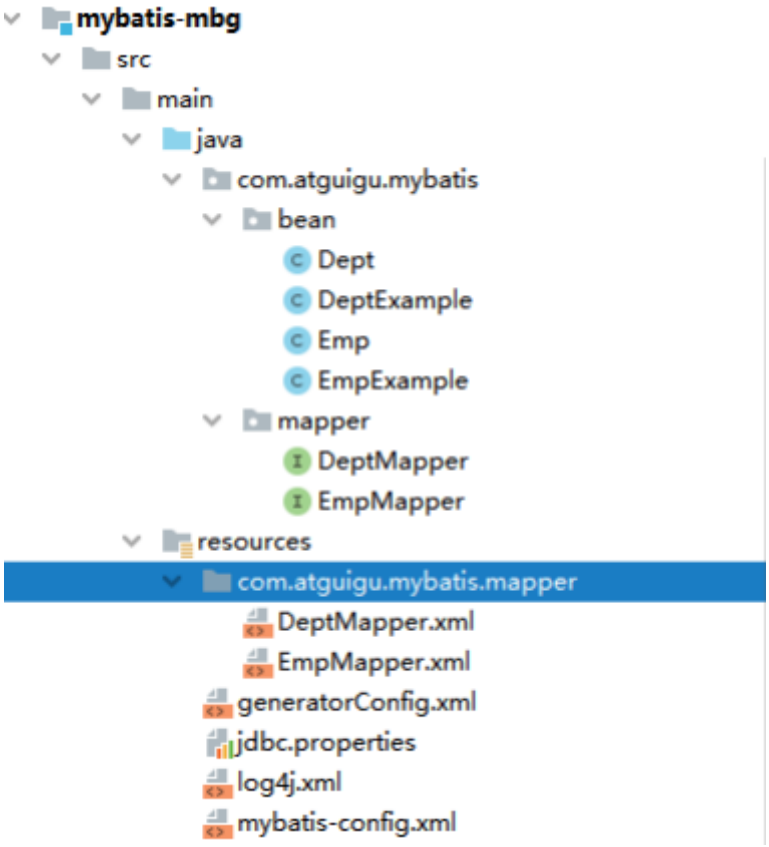
③创建逆向工程的配置文件

> 文件名必须是：generatorConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--
            targetRuntime: 执行生成的逆向工程的版本
                    MyBatis3Simple: 生成基本的CRUD（清新简洁版）
                    MyBatis3: 生成带条件的CRUD（奢华尊享版）
     -->
    <context id="DB2Tables" targetRuntime="MyBatis3">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:3306/mybatis?
serverTimezone=UTC"
                        userId="root"
                        password="123456">
        </jdbcConnection>
        <!-- javaBean的生成策略-->
        <javaModelGenerator targetPackage="com.atguigu.mybatis.pojo"
targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
            <property name="trimStrings" value="true" />
        </javaModelGenerator>
        <!-- SQL映射文件的生成策略 -->
        <sqlMapGenerator targetPackage="com.atguigu.mybatis.mapper"
 targetProject=".\src\main\resources">
            <property name="enableSubPackages" value="true" />
        </sqlMapGenerator>
        <!-- Mapper接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER"
targetPackage="com.atguigu.mybatis.mapper"  targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true" />
        </javaClientGenerator>
        <!-- 逆向分析的表 -->
        <!-- tableName设置为*号，可以对应所有表，此时不写domainObjectName -->
        <!-- domainObjectName属性指定生成出来的实体类的类名 -->
        <table tableName="t_emp" domainObjectName="Emp"/>
        <table tableName="t_dept" domainObjectName="Dept"/>
    </context>
</generatorConfiguration>
```

④执行MBG插件的generate目标

⑤效果



11.2、QBC查询

```
@Test
public void testMBG(){
    try {
        InputStream is = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(is);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        //查询所有数据
        /*List<Emp> list = mapper.selectByExample(null);
        list.forEach(emp -> System.out.println(emp));*/
        //根据条件查询
        /*EmpExample example = new EmpExample();
```

```
        example.createCriteria().andEmpNameEqualTo("张
三").andAgeGreaterThanOrEqualTo(20);
        example.or().andDidIsNotNull();
        List<Emp> list = mapper.selectByExample(example);
        list.forEach(emp -> System.out.println(emp));*/
        mapper.updateByPrimaryKeySelective(new
Emp(1,"admin",22,null,"456@qq.com",3));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

12、分页插件

limit index,pageSize

pageSize：每页显示的条数

pageNum：当前页的页码

index：当前页的起始索引，index=(pageNum-1)*pageSize

count：总记录数

totalPage：总页数

totalPage = count / pageSize;

if(count % pageSize != 0){

totalPage += 1;

}

pageSize=4，pageNum=1，index=0 limit 0,4

pageSize=4，pageNum=3，index=8 limit 8,4

pageSize=4，pageNum=6，index=20 limit 8,4

首页 上一页 2 3 4 5 6 下一页 末页

12.1、分页插件的使用步骤

①添加依赖

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.2.0</version>
</dependency>
```

②配置分页插件

在MyBatis的核心配置文件中配置插件

```
<plugins>
    <!--设置分页插件-->
    <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
</plugins>
```

12.2、分页插件的使用

a>在查询功能之前使用PageHelper.startPage(int pageNum, int pageSize)开启分页功能

> pageNum：当前页的页码
>
> pageSize：每页显示的条数

b>在查询获取list集合之后，使用PageInfo<T> pageInfo = new PageInfo<>(List<T> list, int navigatePages)获取分页相关数据

> list：分页之后的数据
>
> navigatePages：导航分页的页码数

c>分页相关数据

> PageInfo{
>
> pageNum=8, pageSize=4, size=2, startRow=29, endRow=30, total=30, pages=8,
>
> list=Page{count=true, pageNum=8, pageSize=4, startRow=28, endRow=32, total=30,
>
> pages=8, reasonable=false, pageSizeZero=false},
>
> prePage=7, nextPage=0, isFirstPage=false, isLastPage=true, hasPreviousPage=true,
>
> hasNextPage=false, navigatePages=5, navigateFirstPage4, navigateLastPage8,
>
> navigatepageNums=[4, 5, 6, 7, 8]
>
> }
>
> pageNum：当前页的页码
>
> pageSize：每页显示的条数
>
> size：当前页显示的真实条数
>
> total：总记录数
>
> pages：总页数
>
> prePage：上一页的页码
>
> nextPage：下一页的页码
>
> isFirstPage/isLastPage：是否为第一页/最后一页
>
> hasPreviousPage/hasNextPage：是否存在上一页/下一页
>
> navigatePages：导航分页的页码数
>
> navigatepageNums：导航分页的页码，[1,2,3,4,5]

二、Spring

1、Spring简介

1.1、Spring概述

官网地址：https://spring.io/

> Spring 是最受欢迎的企业级 Java 应用程序开发框架，数以百万的来自世界各地的开发人员使用 Spring 框架来创建性能好、易于测试、可重用的代码。
>
> Spring 框架是一个开源的 Java 平台，它最初是由 Rod Johnson 编写的，并且于 2003 年 6 月首次在 Apache 2.0 许可下发布。
>
> Spring 是轻量级的框架，其基础版本只有 2 MB 左右的大小。
>
> Spring 框架的核心特性是可以用于开发任何 Java 应用程序，但是在 Java EE 平台上构建 web 应用程序是需要扩展的。 Spring 框架的目标是使 J2EE 开发变得更容易使用，通过启用基于 POJO 编程模型来促进良好的编程实践。

1.2、Spring家族

项目列表：https://spring.io/projects

1.3、Spring Framework

Spring 基础框架，可以视为 Spring 基础设施，基本上任何其他 Spring 项目都是以 Spring Framework 为基础的。

1.3.1、Spring Framework特性

- 非侵入式：使用 Spring Framework 开发应用程序时，Spring 对应用程序本身的结构影响非常小。对领域模型可以做到零污染；对功能性组件也只需要使用几个简单的注解进行标记，完全不会破坏原有结构，反而能将组件结构进一步简化。这就使得基于 Spring Framework 开发应用程序时结构清晰、简洁优雅。
- 控制反转：IOC——Inversion of Control，翻转资源获取方向。把自己创建资源、向环境索取资源变成环境将资源准备好，我们享受资源注入。
- 面向切面编程：AOP——Aspect Oriented Programming，在不修改源代码的基础上增强代码功能。
- 容器：Spring IOC 是一个容器，因为它包含并且管理组件对象的生命周期。组件享受到了容器化的管理，替程序员屏蔽了组件创建过程中的大量细节，极大的降低了使用门槛，大幅度提高了开发效率。
- 组件化：Spring 实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用 XML 和 Java 注解组合这些对象。这使得我们可以基于一个个功能明确、边界清晰的组件有条不紊的搭建超大型复杂应用系统。
- 声明式：很多以前需要编写代码才能实现的功能，现在只需要声明需求即可由框架代为实现。
- 一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库。而且 Spring 旗下的项目已经覆盖了广泛领域，很多方面的功能性需求可以在 Spring Framework 的基础上全部使用 Spring 来实现。

1.3.2、Spring Framework五大功能模块

功能模块	功能介绍
Core Container	核心容器，在 Spring 环境下使用任何功能都必须基于 IOC 容器。
AOP&Aspects	面向切面编程
Testing	提供了对 junit 或 TestNG 测试框架的整合。
Data Access/Integration	提供了对数据访问/集成的功能。
Spring MVC	提供了面向Web应用程序的集成功能。

2、IOC

2.1、IOC容器

2.1.1、IOC思想

IOC：Inversion of Control，翻译过来是**反转控制。**

①获取资源的传统方式

自己做饭：买菜、洗菜、择菜、改刀、炒菜，全过程参与，费时费力，必须清楚了解资源创建整个过程中的全部细节且熟练掌握。

在应用程序中的组件需要获取资源时，传统的方式是组件**主动**的从容器中获取所需要的资源，在这样的模式下开发人员往往需要知道在具体容器中特定资源的获取方式，增加了学习成本，同时降低了开发效率。

②反转控制方式获取资源

点外卖：下单、等、吃，省时省力，不必关心资源创建过程的所有细节。

反转控制的思想完全颠覆了应用程序组件获取资源的传统方式：反转了资源的获取方向——改由容器主动的将资源推送给需要的组件，开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率。这种行为也称为查找的**被动**形式。

③DI

DI：Dependency Injection，翻译过来是**依赖注入。**

DI 是 IOC 的另一种表述方式：即组件以一些预先定义好的方式（例如：setter 方法）接受来自于容器的资源注入。相对于IOC而言，这种表述更直接。

所以结论是：IOC 就是一种反转控制的思想， 而 DI 是对 IOC 的一种具体实现。

2.1.2、IOC容器在Spring中的实现

Spring 的 IOC 容器就是 IOC 思想的一个落地的产品实现。IOC 容器中管理的组件也叫做 bean。在创建 bean 之前，首先需要创建 IOC 容器。Spring 提供了 IOC 容器的两种实现方式：

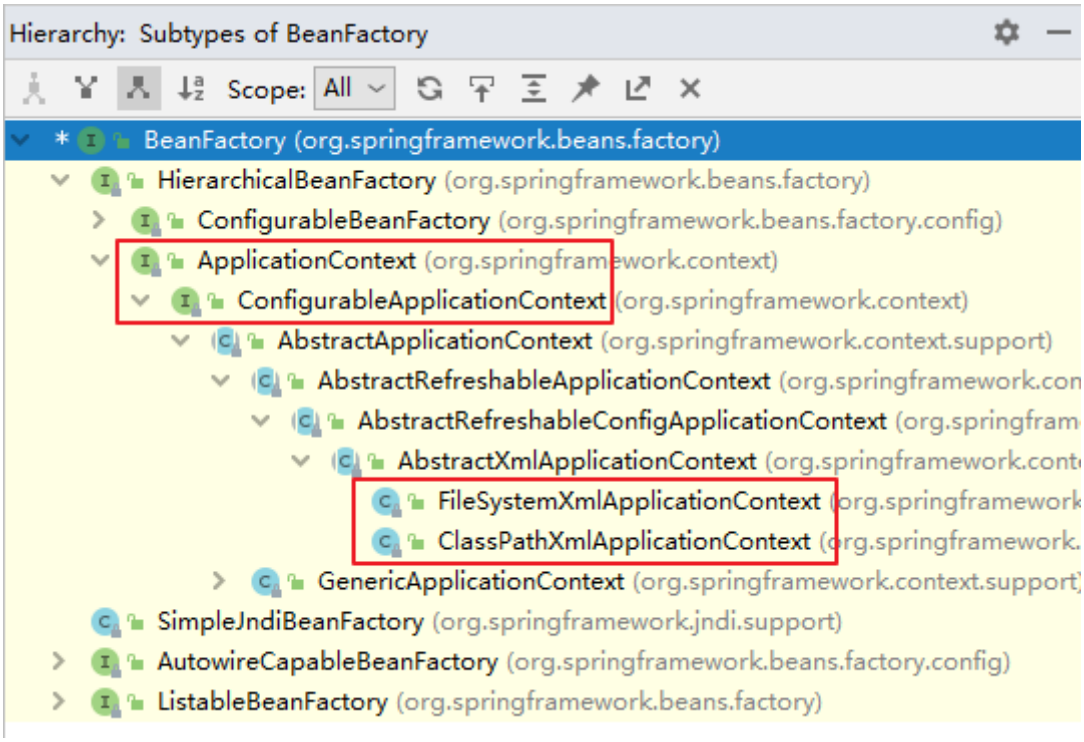
①BeanFactory

这是 IOC 容器的基本实现，是 Spring 内部使用的接口。面向 Spring 本身，不提供给开发人员使用。

②ApplicationContext

BeanFactory 的子接口，提供了更多高级特性。面向 Spring 的使用者，几乎所有场合都使用 ApplicationContext 而不是底层的 BeanFactory。

③ApplicationContext的主要实现类



类型名	简介
ClassPathXmlApplicationContext	通过读取类路径下的 XML 格式的配置文件创建 IOC 容器对象
FileSystemXmlApplicationContext	通过文件系统路径读取 XML 格式的配置文件创建 IOC 容器对象
ConfigurableApplicationContext	ApplicationContext 的子接口，包含一些扩展方法 refresh() 和 close() ，让 ApplicationContext 具有启动、关闭和刷新上下文的能力。
WebApplicationContext	专门为 Web 应用准备，基于 Web 环境创建 IOC 容器对象，并将对象引入存入 ServletContext 域中。

2.2、基于XML管理bean

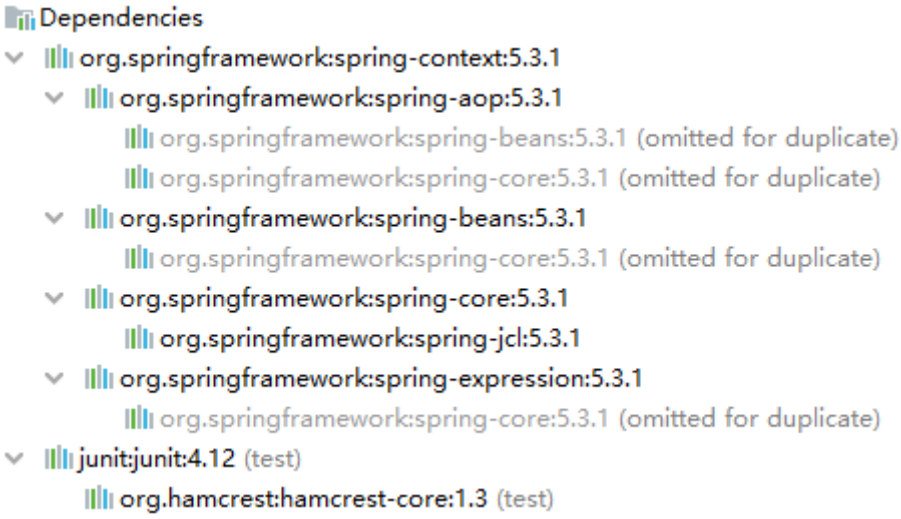
2.2.1、实验一：入门案例

①创建Maven Module

②引入依赖

```
<dependencies>
    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
```

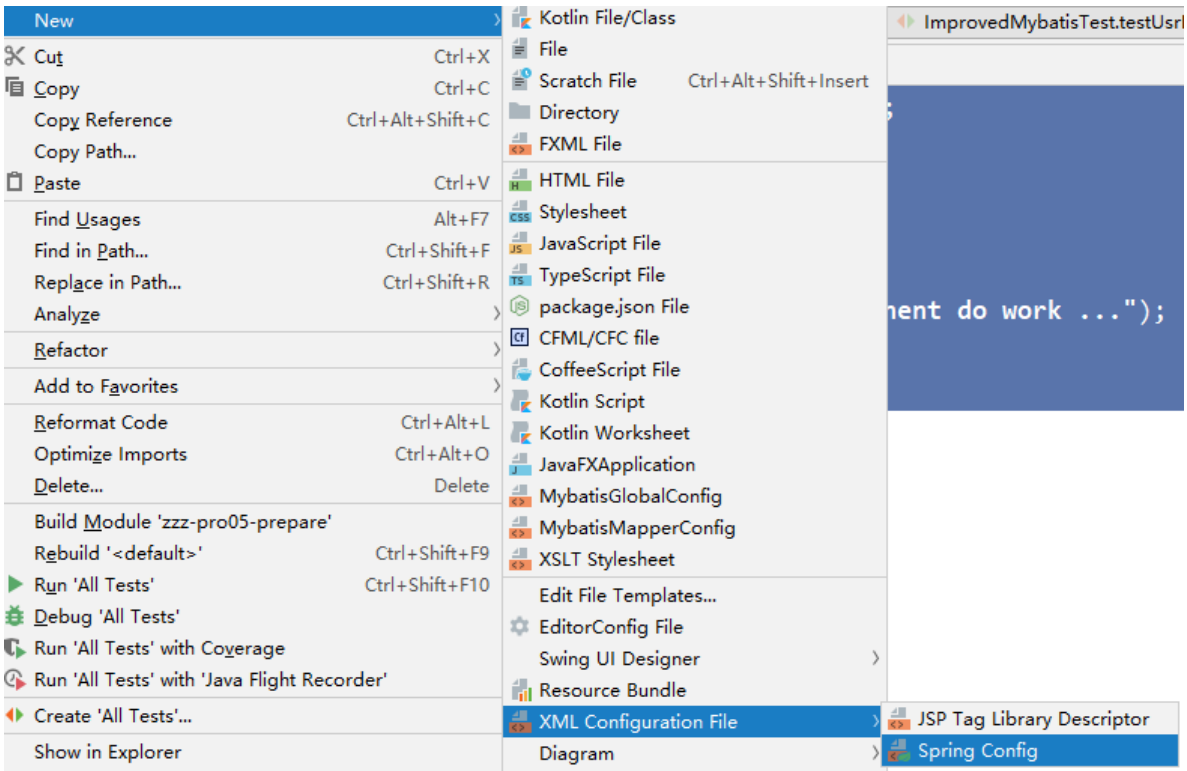
```
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

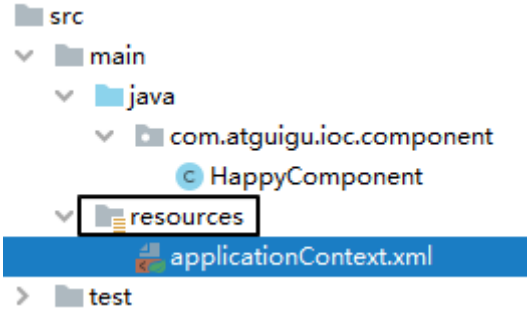


③创建类HelloWorld

```
public class HelloWorld {

    public void sayHello(){
        System.out.println("helloworld");
    }

}
```

④创建Spring的配置文件





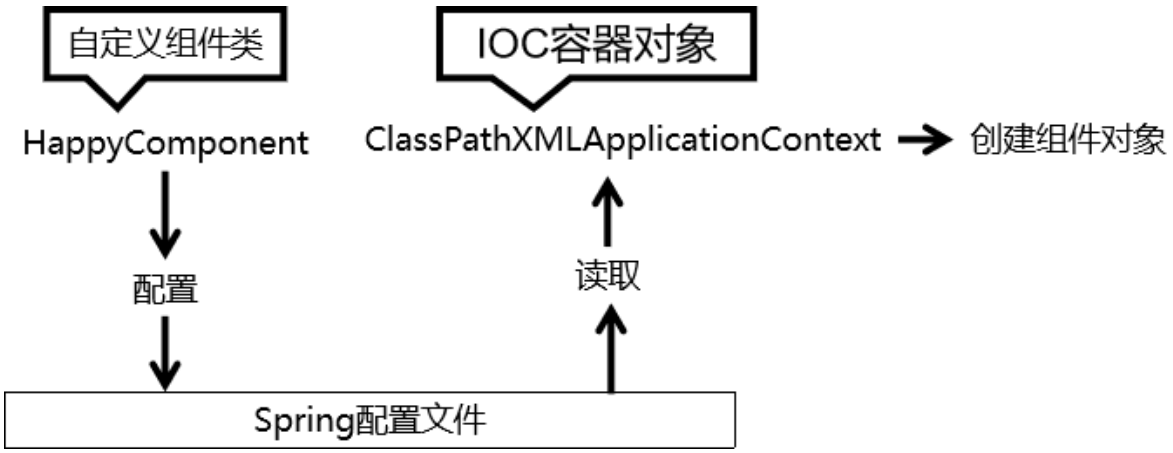
⑤在Spring的配置文件中配置bean

```
<!--
    配置HelloWorld所对应的bean，即将HelloWorld的对象交给Spring的IOC容器管理
    通过bean标签配置IOC容器所管理的bean
    属性：
        id：设置bean的唯一标识
        class：设置bean所对应类型的全类名
-->
<bean id="helloworld" class="com.atguigu.spring.bean.HelloWorld"></bean>
```

⑥创建测试类测试

```
@Test
public void testHelloWorld(){
    ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
    HelloWorld helloworld = (HelloWorld) ac.getBean("helloworld");
    helloworld.sayHello();
}
```

⑦思路



⑧注意

Spring 底层默认通过反射技术调用组件类的无参构造器来创建组件对象，这一点需要注意。如果在需要无参构造器时，没有无参构造器，则会抛出下面的异常：

> org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'helloworld' defined in class path resource [applicationContext.xml]: Instantiation of bean failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.atguigu.spring.bean.HelloWorld]: No default constructor found; nested exception is java.lang.NoSuchMethodException: com.atguigu.spring.bean.HelloWorld.<init>()

2.2.2、实验二：获取bean

①方式一：根据id获取

由于 id 属性指定了 bean 的唯一标识，所以根据 bean 标签的 id 属性可以精确获取到一个组件对象。上个实验中我们使用的就是这种方式。

②方式二：根据类型获取

```
@Test
public void testHelloWorld(){
    ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
    HelloWorld bean = ac.getBean(HelloWorld.class);
    bean.sayHello();
}
```

③方式三：根据id和类型

```
@Test
public void testHelloWorld(){
    ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
    HelloWorld bean = ac.getBean("helloworld", HelloWorld.class);
    bean.sayHello();
}
```

④注意

当根据类型获取bean时，要求IOC容器中指定类型的bean有且只能有一个

当IOC容器中一共配置了两个：

```
<bean id="helloworldOne" class="com.atguigu.spring.bean.HelloWorld"></bean>
<bean id="helloworldTwo" class="com.atguigu.spring.bean.HelloWorld"></bean>
```

根据类型获取时会抛出异常：

> org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.atguigu.spring.bean.HelloWorld' available: expected single matching bean but found 2: helloworldOne,helloworldTwo

⑤扩展

如果组件类实现了接口，根据接口类型可以获取 bean 吗?

> 可以，前提是bean唯一

如果一个接口有多个实现类，这些实现类都配置了 bean，根据接口类型可以获取 bean 吗?

> 不行，因为bean不唯一

⑥结论

根据类型来获取bean时，在满足bean唯一性的前提下，其实只是看：『对象 **instanceof** 指定的类型』的返回结果，只要返回的是true就可以认定为和类型匹配，能够获取到。

2.2.3、实验三：依赖注入之setter注入

①创建学生类Student

```
public class Student {

    private Integer id;

    private String name;

    private Integer age;

    private String sex;

    public Student() {
    }

    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;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ", sex='" + sex + '\'' +
                '}';
    }
```

```
}
```

②配置bean时为属性赋值

```
<bean id="studentOne" class="com.atguigu.spring.bean.Student">
    <!-- property标签：通过组件类的setXxx()方法给组件对象设置属性 -->
    <!-- name属性：指定属性名（这个属性名是getXxx()、setXxx()方法定义的，和成员变量无关）
-->
    <!-- value属性：指定属性值 -->
    <property name="id" value="1001"></property>
    <property name="name" value="张三"></property>
    <property name="age" value="23"></property>
    <property name="sex" value="男"></property>
</bean>
```

③测试

```
@Test
public void testDIBySet(){
    ApplicationContext ac = new ClassPathXmlApplicationContext("spring-di.xml");
    Student studentOne = ac.getBean("studentOne", Student.class);
    System.out.println(studentOne);
}
```

2.2.4、实验四：依赖注入之构造器注入

①在Student类中添加有参构造

```
public Student(Integer id, String name, Integer age, String sex) {
    this.id = id;
    this.name = name;
    this.age = age;
    this.sex = sex;
}
```

②配置bean

```
<bean id="studentTwo" class="com.atguigu.spring.bean.Student">
    <constructor-arg value="1002"></constructor-arg>
    <constructor-arg value="李四"></constructor-arg>
    <constructor-arg value="33"></constructor-arg>
    <constructor-arg value="女"></constructor-arg>
</bean>
```

> 注意：
>
> constructor-arg标签还有两个属性可以进一步描述构造器参数：
>
> - index属性：指定参数所在位置的索引（从0开始）
> - name属性：指定参数名

③测试

```
@Test
public void testDIBySet(){
    ApplicationContext ac = new ClassPathXmlApplicationContext("spring-di.xml");
    Student studentOne = ac.getBean("studentTwo", Student.class);
    System.out.println(studentOne);
}
```

2.2.5、实验五：特殊值处理

①字面量赋值

> 什么是字面量?
>
> int a = 10;
>
> 声明一个变量a，初始化为10，此时a就不代表字母a了，而是作为一个变量的名字。当我们引用a的时候，我们实际上拿到的值是10。
>
> 而如果a是带引号的：'a'，那么它现在不是一个变量，它就是代表a这个字母本身，这就是字面量。所以字面量没有引申含义，就是我们看到的这个数据本身。

```
<!-- 使用value属性给bean的属性赋值时，Spring会把value属性的值看做字面量 -->
<property name="name" value="张三"/>
```

②null值

```
<property name="name">
    <null />
</property>
```

> 注意：
>
> ```
> <property name="name" value="null"></property>
> ```
>
> 以上写法，为name所赋的值是字符串null

③xml实体

```
<!-- 小于号在XML文档中用来定义标签的开始，不能随便使用 -->
<!-- 解决方案一：使用XML实体来代替 -->
<property name="expression" value="a &lt; b"/>
```

④CDATA节

```
<property name="expression">
    <!-- 解决方案二：使用CDATA节 -->
    <!-- CDATA中的C代表Character，是文本、字符的含义，CDATA就表示纯文本数据 -->
    <!-- XML解析器看到CDATA节就知道这里是纯文本，就不会当作XML标签或属性来解析 -->
    <!-- 所以CDATA节中写什么符号都随意 -->
    <value><![CDATA[a < b]]></value>
</property>
```

2.2.6、实验六：为类类型属性赋值

①创建班级类Clazz

```
public class Clazz {

    private Integer clazzId;

    private String clazzName;

    public Integer getClazzId() {
        return clazzId;
    }

    public void setClazzId(Integer clazzId) {
        this.clazzId = clazzId;
    }

    public String getClazzName() {
        return clazzName;
    }

    public void setClazzName(String clazzName) {
        this.clazzName = clazzName;
    }

    @Override
    public String toString() {
        return "Clazz{" +
                "clazzId=" + clazzId +
                ", clazzName='" + clazzName + '\'' +
                '}';
    }

    public Clazz() {
    }

    public Clazz(Integer clazzId, String clazzName) {
        this.clazzId = clazzId;
        this.clazzName = clazzName;
    }
}
```

②修改Student类

在Student类中添加以下代码：

```
private Clazz clazz;

public Clazz getClazz() {
    return clazz;
}

public void setClazz(Clazz clazz) {
    this.clazz = clazz;
}
```

③方式一：引用外部已声明的bean

配置Clazz类型的bean：

```
<bean id="clazzOne" class="com.atguigu.spring.bean.Clazz">
    <property name="clazzId" value="1111"></property>
    <property name="clazzName" value="财源滚滚班"></property>
</bean>
```

为Student中的clazz属性赋值：

```
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
    <property name="clazz" ref="clazzOne"></property>
</bean>
```

错误演示：

```
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <property name="clazz" value="clazzOne"></property>
</bean>
```

> 如果错把ref属性写成了value属性，会抛出异常： Caused by: java.lang.IllegalStateException: Cannot convert value of type 'java.lang.String' to required type 'com.atguigu.spring.bean.Clazz' for property 'clazz': no matching editors or conversion strategy found
>
> 意思是不能把String类型转换成我们要的Clazz类型，说明我们使用value属性时，Spring只把这个属性看做一个普通的字符串，不会认为这是一个bean的id，更不会根据它去找到bean来赋值

④方式二：内部bean

```
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <property name="clazz">
        <!-- 在一个bean中再声明一个bean就是内部bean -->
        <!-- 内部bean只能用于给属性赋值，不能在外部通过IOC容器获取，因此可以省略id属性 -->
        <bean id="clazzInner" class="com.atguigu.spring.bean.Clazz">
            <property name="clazzId" value="2222"></property>
            <property name="clazzName" value="远大前程班"></property>
        </bean>
    </property>
</bean>
```

③方式三：级联属性赋值

```
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <!-- 一定先引用某个bean为属性赋值，才可以使用级联方式更新属性 -->
    <property name="clazz" ref="clazzOne"></property>
    <property name="clazz.clazzId" value="3333"></property>
    <property name="clazz.clazzName" value="最强王者班"></property>
</bean>
```

2.2.7、实验七：为数组类型属性赋值

①修改Student类

在Student类中添加以下代码：

```
private String[] hobbies;

public String[] getHobbies() {
    return hobbies;
}

public void setHobbies(String[] hobbies) {
    this.hobbies = hobbies;
}
```

②配置bean

```
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
    <property name="clazz" ref="clazzOne"></property>
    <property name="hobbies">
        <array>
            <value>抽烟</value>
            <value>喝酒</value>
            <value>烫头</value>
        </array>
    </property>
</bean>
```

2.2.8、实验八：为集合类型属性赋值

①为List集合类型属性赋值

在Clazz类中添加以下代码：

```
private List<Student> students;

public List<Student> getStudents() {
    return students;
}

public void setStudents(List<Student> students) {
    this.students = students;
}
```

配置bean：

```
<bean id="clazzTwo" class="com.atguigu.spring.bean.Clazz">
    <property name="clazzId" value="4444"></property>
    <property name="clazzName" value="JavaeeO222"></property>
    <property name="students">
        <list>
            <ref bean="studentOne"></ref>
            <ref bean="studentTwo"></ref>
            <ref bean="studentThree"></ref>
        </list>
    </property>
</bean>
```

> 若为Set集合类型属性赋值，只需要将其中的list标签改为set标签即可

②为Map集合类型属性赋值

创建教师类Teacher：

```
public class Teacher {

    private Integer teacherId;

    private String teacherName;

    public Integer getTeacherId() {
        return teacherId;
    }

    public void setTeacherId(Integer teacherId) {
        this.teacherId = teacherId;
    }

    public String getTeacherName() {
        return teacherName;
    }

    public void setTeacherName(String teacherName) {
        this.teacherName = teacherName;
    }

    public Teacher(Integer teacherId, String teacherName) {
        this.teacherId = teacherId;
        this.teacherName = teacherName;
    }
```

```
    public Teacher() {

    }

    @Override
    public String toString() {
        return "Teacher{" +
                "teacherId=" + teacherId +
                ", teacherName='" + teacherName + '\'' +
                '}';
    }
}
```

在Student类中添加以下代码：

```
private Map<String, Teacher> teacherMap;

public Map<String, Teacher> getTeacherMap() {
    return teacherMap;
}

public void setTeacherMap(Map<String, Teacher> teacherMap) {
    this.teacherMap = teacherMap;
}
```

配置bean：

```
<bean id="teacherOne" class="com.atguigu.spring.bean.Teacher">
    <property name="teacherId" value="10010"></property>
    <property name="teacherName" value="大宝"></property>
</bean>

<bean id="teacherTwo" class="com.atguigu.spring.bean.Teacher">
    <property name="teacherId" value="10086"></property>
    <property name="teacherName" value="二宝"></property>
</bean>

<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
    <property name="clazz" ref="clazzOne"></property>
    <property name="hobbies">
        <array>
            <value>抽烟</value>
            <value>喝酒</value>
            <value>烫头</value>
        </array>
    </property>
    <property name="teacherMap">
        <map>
            <entry>
                <key>
```

```
                    <value>10010</value>
                </key>
                <ref bean="teacherOne"></ref>
            </entry>
            <entry>
                <key>
                    <value>10086</value>
                </key>
                <ref bean="teacherTwo"></ref>
            </entry>
        </map>
    </property>
</bean>
```

③引用集合类型的bean

```
<!--list集合类型的bean-->
<util:list id="students">
    <ref bean="studentOne"></ref>
    <ref bean="studentTwo"></ref>
    <ref bean="studentThree"></ref>
</util:list>
<!--map集合类型的bean-->
<util:map id="teacherMap">
    <entry>
        <key>
            <value>10010</value>
        </key>
        <ref bean="teacherOne"></ref>
    </entry>
    <entry>
        <key>
            <value>10086</value>
        </key>
        <ref bean="teacherTwo"></ref>
    </entry>
</util:map>
<bean id="clazzTwo" class="com.atguigu.spring.bean.Clazz">
    <property name="clazzId" value="4444"></property>
    <property name="clazzName" value="Javaee0222"></property>
    <property name="students" ref="students"></property>
</bean>
<bean id="studentFour" class="com.atguigu.spring.bean.Student">
    <property name="id" value="1004"></property>
    <property name="name" value="赵六"></property>
    <property name="age" value="26"></property>
    <property name="sex" value="女"></property>
    <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
    <property name="clazz" ref="clazzOne"></property>
    <property name="hobbies">
        <array>
            <value>抽烟</value>
            <value>喝酒</value>
            <value>烫头</value>
        </array>
    </property>
    <property name="teacherMap" ref="teacherMap"></property>
```

```
</bean>
```

> 使用util:list、util:map标签必须引入相应的命名空间，可以通过idea的提示功能选择

2.2.9、实验九：p命名空间

引入p命名空间后，可以通过以下方式为bean的各个属性赋值

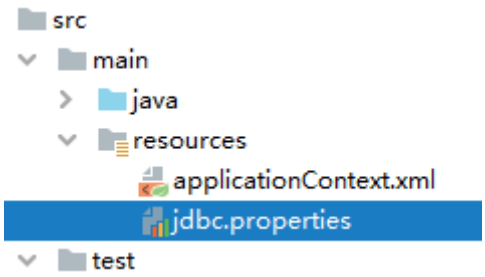
```
<bean id="studentSix" class="com.atguigu.spring.bean.Student"
    p:id="1006" p:name="小明" p:clazz-ref="clazzOne" p:teacherMap-
ref="teacherMap"></bean>
```

2.2.10、实验十：引入外部属性文件

①加入依赖

```
 <!-- MySQL驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.16</version>
</dependency>
<!-- 数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.31</version>
</dependency>
```

②创建外部属性文件




```
jdbc.user=root
jdbc.password=atguigu
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.driver=com.mysql.cj.jdbc.Driver
```

③引入属性文件

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

④配置bean

```
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="username" value="${jdbc.user}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

⑤测试

```
@Test
public void testDataSource() throws SQLException {
    ApplicationContext ac = new ClassPathXmlApplicationContext("spring-
datasource.xml");
    DataSource dataSource = ac.getBean(DataSource.class);
    Connection connection = dataSource.getConnection();
    System.out.println(connection);
}
```

2.2.11、实验十一：bean的作用域

①概念

在Spring中可以通过配置bean标签的scope属性来指定bean的作用域范围，各取值含义参加下表：

取值	含义	创建对象的时机
singleton（默认）	在IOC容器中，这个bean的对象始终为单实例	IOC容器初始化时
prototype	这个bean在IOC容器中有多个实例	获取bean时

如果是在WebApplicationContext环境下还会有另外两个作用域（但不常用）：

取值	含义
request	在一个请求范围内有效
session	在一个会话范围内有效

②创建类User

```
public class User {

    private Integer id;

    private String username;

    private String password;

    private Integer age;

    public User() {
    }

    public User(Integer id, String username, String password, Integer age) {
        this.id = id;
        this.username = username;
```

```
        this.password = password;
        this.age = age;
    }

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

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

    public String getPassword() {
        return password;
    }

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

    public Integer getAge() {
        return age;
    }

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

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

③配置bean

```
<!-- scope属性：取值singleton（默认值），bean在IOC容器中只有一个实例，IOC容器初始化时创建对象 -->
<!-- scope属性：取值prototype，bean在IOC容器中可以有多个实例，getBean()时创建对象 -->
<bean class="com.atguigu.bean.User" scope="prototype"></bean>
```

④测试

```
@Test
public void testBeanScope(){
    ApplicationContext ac = new ClassPathXmlApplicationContext("spring-
scope.xml");
    User user1 = ac.getBean(User.class);
    User user2 = ac.getBean(User.class);
    System.out.println(user1==user2);
}
```

2.2.12、实验十二：bean的生命周期

①具体的生命周期过程

- bean对象创建（调用无参构造器）
- 给bean对象设置属性
- bean对象初始化之前操作（由bean的后置处理器负责）
- bean对象初始化（需在配置bean时指定初始化方法）
- bean对象初始化之后操作（由bean的后置处理器负责）
- bean对象就绪可以使用
- bean对象销毁（需在配置bean时指定销毁方法）
- IOC容器关闭

②修改类User

```
public class User {

    private Integer id;

    private String username;

    private String password;

    private Integer age;

    public User() {
        System.out.println("生命周期：1、创建对象");
    }

    public User(Integer id, String username, String password, Integer age) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        System.out.println("生命周期：2、依赖注入");
        this.id = id;
    }

    public String getUsername() {
        return username;
```

```
    }

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

    public String getPassword() {
        return password;
    }

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

    public Integer getAge() {
        return age;
    }

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

    public void initMethod(){
        System.out.println("生命周期：3、初始化");
    }

    public void destroyMethod(){
        System.out.println("生命周期：5、销毁");
    }

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

> 注意其中的initMethod()和destroyMethod()，可以通过配置bean指定为初始化和销毁的方法

③配置bean

```
<!-- 使用init-method属性指定初始化方法 -->
<!-- 使用destroy-method属性指定销毁方法 -->
<bean class="com.atguigu.bean.User" scope="prototype" init-method="initMethod"
destroy-method="destroyMethod">
    <property name="id" value="1001"></property>
    <property name="username" value="admin"></property>
    <property name="password" value="123456"></property>
    <property name="age" value="23"></property>
</bean>
```

④测试

```
@Test
public void testLife(){
    ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("spring-lifecycle.xml");
    User bean = ac.getBean(User.class);
    System.out.println("生命周期：4、通过IOC容器获取bean并使用");
    ac.close();
}
```

⑤bean的后置处理器

bean的后置处理器会在生命周期的初始化前后添加额外的操作，需要实现BeanPostProcessor接口，且配置到IOC容器中，需要注意的是，bean后置处理器不是单独针对某一个bean生效，而是针对IOC容器中所有bean都会执行

创建bean的后置处理器：

```
package com.atguigu.spring.process;

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

public class MyBeanProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("☆☆☆" + beanName + " = " + bean);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("★★★" + beanName + " = " + bean);
        return bean;
    }
}
```

在IOC容器中配置后置处理器：

```
<!-- bean的后置处理器要放入IOC容器才能生效 -->
<bean id="myBeanProcessor" class="com.atguigu.spring.process.MyBeanProcessor"/>
```

2.2.13、实验十三：FactoryBean

①简介

FactoryBean是Spring提供的一种整合第三方框架的常用机制。和普通的bean不同，配置一个FactoryBean类型的bean，在获取bean的时候得到的并不是class属性中配置的这个类的对象，而是getObject()方法的返回值。通过这种机制，Spring可以帮我们把复杂组件创建的详细过程和繁琐细节都屏蔽起来，只把最简洁的使用界面展示给我们。

将来我们整合Mybatis时，Spring就是通过FactoryBean机制来帮我们创建SqlSessionFactory对象的。

```
/*
```

```
 * Copyright 2002-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.springframework.beans.factory;

import org.springframework.lang.Nullable;

/**
 * Interface to be implemented by objects used within a {@link BeanFactory}
which
 * are themselves factories for individual objects. If a bean implements this
 * interface, it is used as a factory for an object to expose, not directly as a
 * bean instance that will be exposed itself.
 *
 * <p><b>NB: A bean that implements this interface cannot be used as a normal
bean.</b>
 * A FactoryBean is defined in a bean style, but the object exposed for bean
 * references ({@link #getObject()}) is always the object that it creates.
 *
 * <p>FactoryBeans can support singletons and prototypes, and can either create
 * objects lazily on demand or eagerly on startup. The {@link SmartFactoryBean}
 * interface allows for exposing more fine-grained behavioral metadata.
 *
 * <p>This interface is heavily used within the framework itself, for example
for
 * the AOP {@link org.springframework.aop.framework.ProxyFactoryBean} or the
 * {@link org.springframework.jndi.JndiObjectFactoryBean}. It can be used for
 * custom components as well; however, this is only common for infrastructure
code.
 *
 * <p><b>{@code FactoryBean} is a programmatic contract. Implementations are not
 * supposed to rely on annotation-driven injection or other reflective
facilities.</b>
 * {@link #getObjectType()} {@link #getObject()} invocations may arrive early in
the
 * bootstrap process, even ahead of any post-processor setup. If you need access
to
 * other beans, implement {@link BeanFactoryAware} and obtain them
programmatically.
 *
 * <p><b>The container is only responsible for managing the lifecycle of the
FactoryBean
 * instance, not the lifecycle of the objects created by the FactoryBean.</b>
Therefore,
 * a destroy method on an exposed bean object (such as {@link
java.io.Closeable#close()}
```

```
 * will <i>not</i> be called automatically. Instead, a FactoryBean should
implement
 * {@link DisposableBean} and delegate any such close call to the underlying
object.
 *
 * <p>Finally, FactoryBean objects participate in the containing BeanFactory's
 * synchronization of bean creation. There is usually no need for internal
 * synchronization other than for purposes of lazy initialization within the
 * FactoryBean itself (or the like).
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 08.03.2003
 * @param <T> the bean type
 * @see org.springframework.beans.factory.BeanFactory
 * @see org.springframework.aop.framework.ProxyFactoryBean
 * @see org.springframework.jndi.JndiObjectFactoryBean
 */
public interface FactoryBean<T> {

    /**
     * The name of an attribute that can be
     * {@link org.springframework.core.AttributeAccessor#setAttribute set} on a
     * {@link org.springframework.beans.factory.config.BeanDefinition} so that
     * factory beans can signal their object type when it can't be deduced from
     * the factory bean class.
     * @since 5.2
     */
    String OBJECT_TYPE_ATTRIBUTE = "factoryBeanObjectType";

    /**
     * Return an instance (possibly shared or independent) of the object
     * managed by this factory.
     * <p>As with a {@link BeanFactory}, this allows support for both the
     * Singleton and Prototype design pattern.
     * <p>If this FactoryBean is not fully initialized yet at the time of
     * the call (for example because it is involved in a circular reference),
     * throw a corresponding {@link FactoryBeanNotInitializedException}.
     * <p>As of Spring 2.0, FactoryBeans are allowed to return {@code null}
     * objects. The factory will consider this as normal value to be used; it
     * will not throw a FactoryBeanNotInitializedException in this case anymore.
     * FactoryBean implementations are encouraged to throw
     * FactoryBeanNotInitializedException themselves now, as appropriate.
     * @return an instance of the bean (can be {@code null})
     * @throws Exception in case of creation errors
     * @see FactoryBeanNotInitializedException
     */
    @Nullable
    T getObject() throws Exception;

    /**
     * Return the type of object that this FactoryBean creates,
     * or {@code null} if not known in advance.
     * <p>This allows one to check for specific types of beans without
     * instantiating objects, for example on autowiring.
     * <p>In the case of implementations that are creating a singleton object,
     * this method should try to avoid singleton creation as far as possible;
     * it should rather estimate the type in advance.
```

```
     * For prototypes, returning a meaningful type here is advisable too.
     * <p>This method can be called <i>before</i> this FactoryBean has
     * been fully initialized. It must not rely on state created during
     * initialization; of course, it can still use such state if available.
     * <p><b>NOTE:</b> Autowiring will simply ignore FactoryBeans that return
     * {@code null} here. Therefore it is highly recommended to implement
     * this method properly, using the current state of the FactoryBean.
     * @return the type of object that this FactoryBean creates,
     * or {@code null} if not known at the time of the call
     * @see ListableBeanFactory#getBeansOfType
     */
    @Nullable
    Class<?> getObjectType();

    /**
     * Is the object managed by this factory a singleton? That is,
     * will {@link #getObject()} always return the same object
     * (a reference that can be cached)?
     * <p><b>NOTE:</b> If a FactoryBean indicates to hold a singleton object,
     * the object returned from {@code getObject()} might get cached
     * by the owning BeanFactory. Hence, do not return {@code true}
     * unless the FactoryBean always exposes the same reference.
     * <p>The singleton status of the FactoryBean itself will generally
     * be provided by the owning BeanFactory; usually, it has to be
     * defined as singleton there.
     * <p><b>NOTE:</b> This method returning {@code false} does not
     * necessarily indicate that returned objects are independent instances.
     * An implementation of the extended {@link SmartFactoryBean} interface
     * may explicitly indicate independent instances through its
     * {@link SmartFactoryBean#isPrototype()} method. Plain {@link FactoryBean}
     * implementations which do not implement this extended interface are
     * simply assumed to always return independent instances if the
     * {@code isSingleton()} implementation returns {@code false}.
     * <p>The default implementation returns {@code true}, since a
     * {@code FactoryBean} typically manages a singleton instance.
     * @return whether the exposed object is a singleton
     * @see #getObject()
     * @see SmartFactoryBean#isPrototype()
     */
    default boolean isSingleton() {
        return true;
    }
}
```

②创建类UserFactoryBean

```
public class UserFactoryBean implements FactoryBean<User> {
    @Override
    public User getObject() throws Exception {
        return new User();
    }

    @Override
    public Class<?> getObjectType() {
        return User.class;
    }
}
```

③配置bean

```
<bean id="user" class="com.atguigu.bean.UserFactoryBean"></bean>
```

④测试

```
@Test
public void testUserFactoryBean(){
    //获取IOC容器
    ApplicationContext ac = new ClassPathXmlApplicationContext("spring-
factorybean.xml");
    User user = (User) ac.getBean("user");
    System.out.println(user);
}
```

2.2.14、实验十四：基于xml的自动装配

> 自动装配：
>
> 根据指定的策略，在IOC容器中匹配某一个bean，自动为指定的bean中所依赖的类类型或接口类型属性赋值

①场景模拟

创建类UserController

```
public class UserController {

    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public void saveUser(){
        userService.saveUser();
    }

}
```

创建接口UserService

```
public interface UserService {

    void saveUser();

}
```

创建类UserServiceImpl实现接口UserService

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;

    public void setUserDao(UserDao userDao) {
```

```
        this.userDao = userDao;
    }

    @Override
    public void saveUser() {
        userDao.saveUser();
    }

}
```

创建接口UserDao

```
public interface UserDao {

    void saveUser();

}
```

创建类UserDaoImpl实现接口UserDao

```
public class UserDaoImpl implements UserDao {

    @Override
    public void saveUser() {
        System.out.println("保存成功");
    }

}
```

②配置bean

> 使用bean标签的autowire属性设置自动装配效果
>
> 自动装配方式：byType
>
> byType：根据类型匹配IOC容器中的某个兼容类型的bean，为属性自动赋值
>
> 若在IOC中，没有任何一个兼容类型的bean能够为属性赋值，则该属性不装配，即值为默认值null
>
> 若在IOC中，有多个兼容类型的bean能够为属性赋值，则抛出异常NoUniqueBeanDefinitionException

```
<bean id="userController"
class="com.atguigu.autowire.xml.controller.UserController" autowire="byType">
</bean>

<bean id="userService"
class="com.atguigu.autowire.xml.service.impl.UserServiceImpl" autowire="byType">
</bean>

<bean id="userDao" class="com.atguigu.autowire.xml.dao.impl.UserDaoImpl"></bean>
```

> 自动装配方式：byName
>
> byName：将自动装配的属性的属性名，作为bean的id在IOC容器中匹配相对应的bean进行赋值

```
<bean id="userController"
class="com.atguigu.autowire.xml.controller.UserController" autowire="byName">
</bean>

<bean id="userService"
class="com.atguigu.autowire.xml.service.impl.UserServiceImpl" autowire="byName">
</bean>
<bean id="userServiceImpl"
class="com.atguigu.autowire.xml.service.impl.UserServiceImpl" autowire="byName">
</bean>

<bean id="userDao" class="com.atguigu.autowire.xml.dao.impl.UserDaoImpl"></bean>
<bean id="userDaoImpl" class="com.atguigu.autowire.xml.dao.impl.UserDaoImpl">
</bean>
```

③测试

```
@Test
public void testAutoWireByXML(){
    ApplicationContext ac = new ClassPathXmlApplicationContext("autowire-
xml.xml");
    UserController userController = ac.getBean(UserController.class);
    userController.saveUser();
}
```

2.3、基于注解管理bean

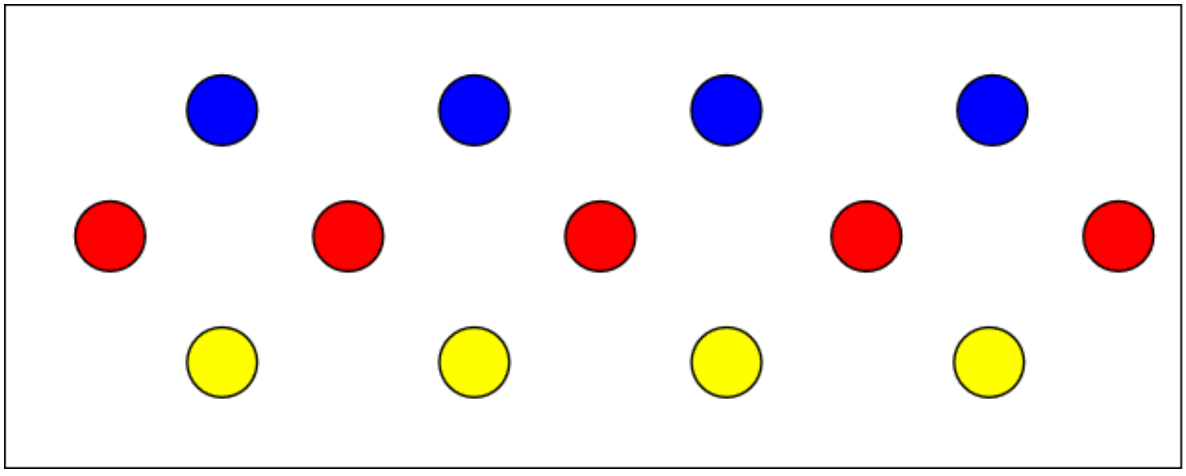
2.3.1、实验一：标记与扫描

①注解

和 XML 配置文件一样，注解本身并不能执行，注解本身仅仅只是做一个标记，具体的功能是框架检测到注解标记的位置，然后针对这个位置按照注解标记的功能来执行具体操作。

本质上：所有一切的操作都是Java代码来完成的，XML和注解只是告诉框架中的Java代码如何执行。

举例：元旦联欢会要布置教室，蓝色的地方贴上元旦快乐四个字，红色的地方贴上拉花，黄色的地方贴上气球。



班长做了所有标记，同学们来完成具体工作。墙上的标记相当于我们在代码中使用的注解，后面同学们做的工作，相当于框架的具体操作。

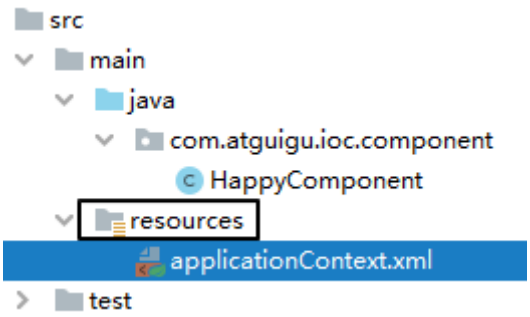
②扫描

Spring 为了知道程序员在哪些地方标记了什么注解，就需要通过扫描的方式，来进行检测。然后根据注解进行后续操作。

③新建Maven Module

```
<dependencies>

    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

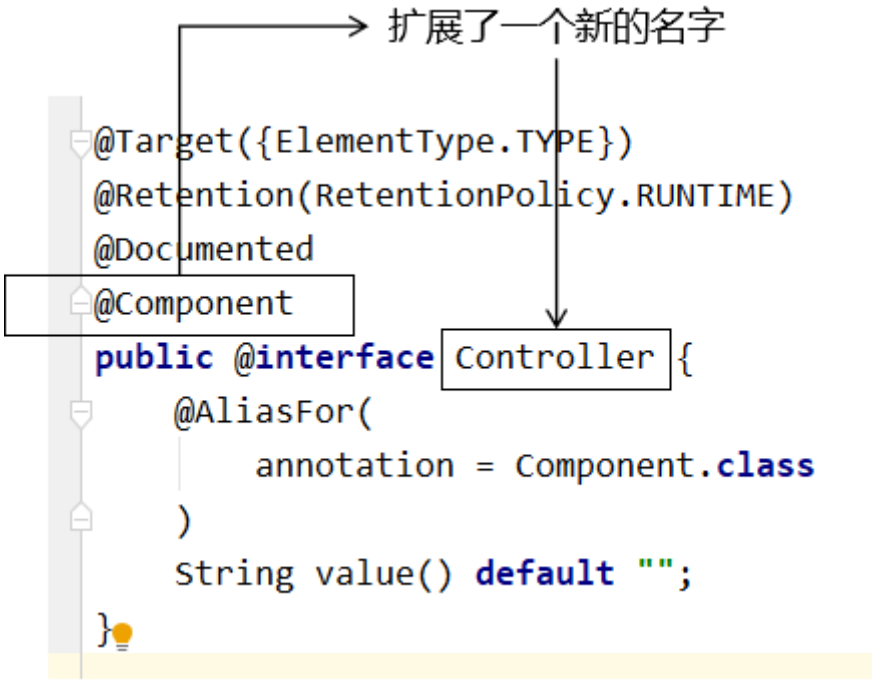
④创建Spring配置文件



⑤标识组件的常用注解

> @Component：将类标识为普通组件 @Controller：将类标识为控制层组件 @Service：将类标识为业务层组件 @Repository：将类标识为持久层组件

问：以上四个注解有什么关系和区别？



通过查看源码我们得知，@Controller、@Service、@Repository这三个注解只是在@Component注解的基础上起了三个新的名字。

对于Spring使用IOC容器管理这些组件来说没有区别。所以@Controller、@Service、@Repository这三个注解只是给开发人员看的，让我们能够便于分辨组件的作用。

注意：虽然它们本质上一样，但是为了代码的可读性，为了程序结构严谨我们肯定不能随便胡乱标记。

⑥创建组件

创建控制层组件

```
@Controller
public class UserController {

}
```

创建接口UserService

```
public interface UserService {

}
```

创建业务层组件UserServiceImpl

```
@Service
public class UserServiceImpl implements UserService {

}
```

创建接口UserDao

```
public interface UserDao {

}
```

创建持久层组件UserDaoImpl

```
@Repository
public class UserDaoImpl implements UserDao {

}
```

⑦扫描组件

情况一：最基本的扫描方式

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

情况二：指定要排除的组件

```
<context:component-scan base-package="com.atguigu">
    <!-- context:exclude-filter标签：指定排除规则 -->
    <!--
        type：设置排除或包含的依据
        type="annotation"，根据注解排除，expression中设置要排除的注解的全类名
        type="assignable"，根据类型排除，expression中设置要排除的类型的全类名
    -->
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
        <!--<context:exclude-filter type="assignable"
expression="com.atguigu.controller.UserController"/>-->
</context:component-scan>
```

情况三：仅扫描指定组件

```
<context:component-scan base-package="com.atguigu" use-default-filters="false">
    <!-- context:include-filter标签：指定在原有扫描规则的基础上追加的规则 -->
    <!-- use-default-filters属性：取值false表示关闭默认扫描规则 -->
    <!-- 此时必须设置use-default-filters="false"，因为默认规则即扫描指定包下所有类 -->
    <!--
        type：设置排除或包含的依据
        type="annotation"，根据注解排除，expression中设置要排除的注解的全类名
        type="assignable"，根据类型排除，expression中设置要排除的类型的全类名
    -->
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    <!--<context:include-filter type="assignable"
expression="com.atguigu.controller.UserController"/>-->
</context:component-scan>
```

⑧测试

```
@Test
public void testAutowireByAnnotation(){
    ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
    UserController userController = ac.getBean(UserController.class);
    System.out.println(userController);
    UserService userService = ac.getBean(UserService.class);
    System.out.println(userService);
    UserDao userDao = ac.getBean(UserDao.class);
    System.out.println(userDao);
}
```

⑨组件所对应的bean的id

在我们使用XML方式管理bean的时候，每个bean都有一个唯一标识，便于在其他地方引用。现在使用注解后，每个组件仍然应该有一个唯一标识。

> 默认情况
>
> 类名首字母小写就是bean的id。例如：UserController类对应的bean的id就是userController。
>
> 自定义bean的id
>
> 可通过标识组件的注解的value属性设置自定义的bean的id
>
> @Service("userService")//默认为userServiceImpl public class UserServiceImpl implements UserService {}

2.3.2、实验二：基于注解的自动装配

①场景模拟

> 参考基于xml的自动装配
>
> 在UserController中声明UserService对象
>
> 在UserServiceImpl中声明UserDao对象

②@Autowired注解

在成员变量上直接标记@Autowired注解即可完成自动装配，不需要提供setXxx()方法。以后我们在项目中的正式用法就是这样。

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    public void saveUser(){
        userService.saveUser();
    }

}
```

```
public interface UserService {
    void saveUser();
}
```

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    @Override
    public void saveUser() {
        userDao.saveUser();
    }
}
```

```
public interface UserDao {
    void saveUser();
}
```

```
@Repository
public class UserDaoImpl implements UserDao {
    @Override
    public void saveUser() {
        System.out.println("保存成功");
    }
}
```

③@Autowired注解其他细节

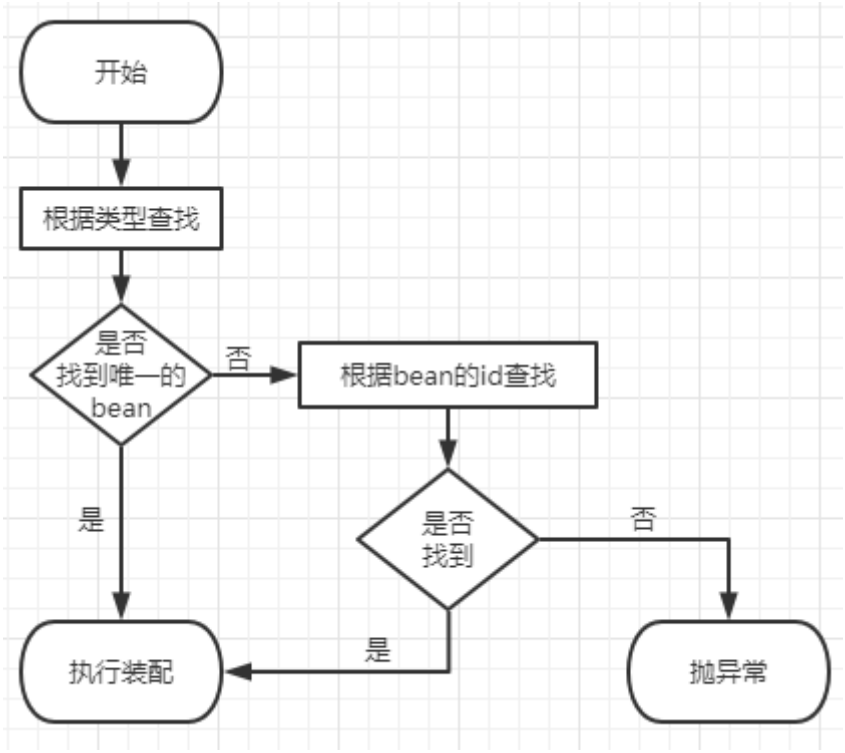
> @Autowired注解可以标记在构造器和set方法上

```
@Controller
public class UserController {

    private UserService userService;

    @Autowired
    public UserController(UserService userService){
        this.userService = userService;
    }

    public void saveUser(){
        userService.saveUser();
    }

}
```

```
@Controller
public class UserController {

    private UserService userService;

    @Autowired
    public void setUserService(UserService userService){
        this.userService = userService;
    }

    public void saveUser(){
```

```
        userService.saveUser();
    }

}
```

④@Autowired工作流程



- 首先根据所需要的组件类型到IOC容器中查找
  - 能够找到唯一的bean：直接执行装配
  - 如果完全找不到匹配这个类型的bean：装配失败
  - 和所需类型匹配的bean不止一个
    - 没有@Qualifier注解：根据@Autowired标记位置成员变量的变量名作为bean的id进行匹配
      - 能够找到：执行装配
      - 找不到：装配失败
    - 使用@Qualifier注解：根据@Qualifier注解中指定的名称作为bean的id进行匹配
      - 能够找到：执行装配
      - 找不到：装配失败

```
@Controller
public class UserController {

    @Autowired
    @Qualifier("userServiceImpl")
    private UserService userService;

    public void saveUser(){
        userService.saveUser();
    }

}
```

> @Autowired中有属性required，默认值为true，因此在自动装配无法找到相应的bean时，会装配失败
>
> 可以将属性required的值设置为true，则表示能装就装，装不上就不装，此时自动装配的属性为默认值
>
> 但是实际开发时，基本上所有需要装配组件的地方都是必须装配的，用不上这个属性。

3、AOP

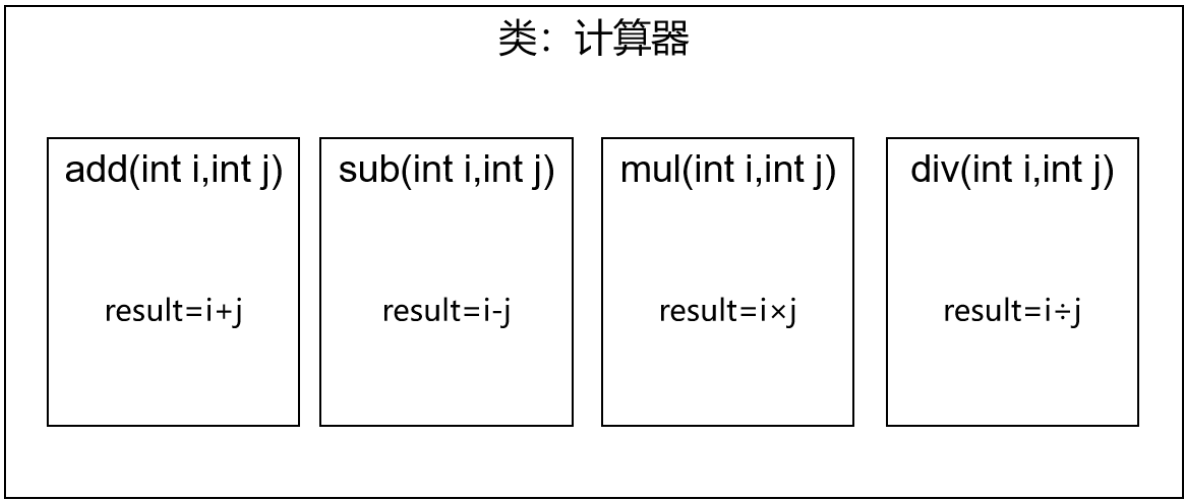
3.1、场景模拟

3.1.1、声明接口

声明计算器接口Calculator，包含加减乘除的抽象方法

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);

}
```

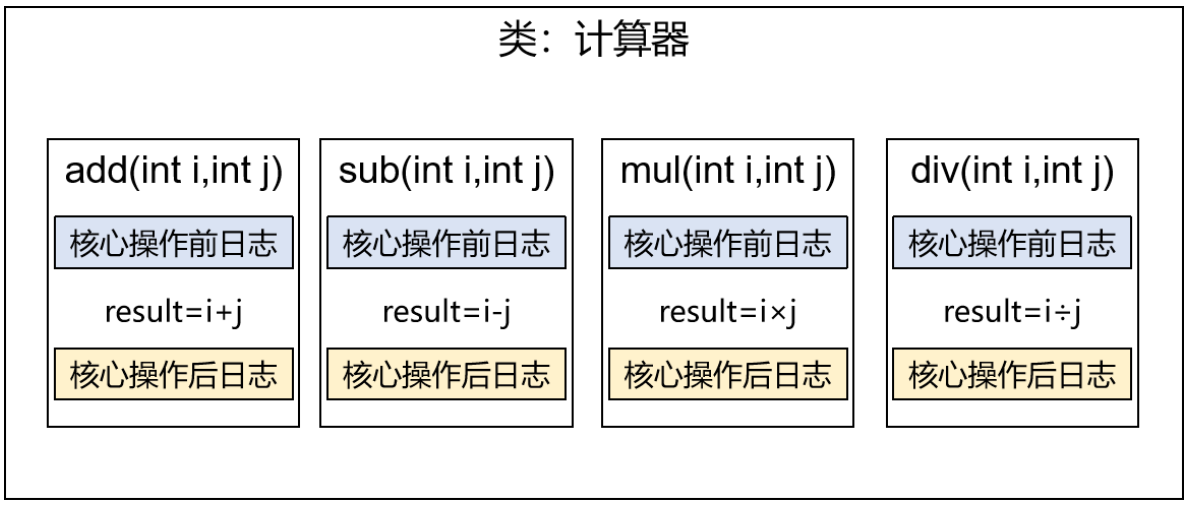
3.1.2、创建实现类




```
public class CalculatorPureImpl implements Calculator {

    @Override
    public int add(int i, int j) {

        int result = i + j;

        System.out.println("方法内部 result = " + result);

        return result;
    }
```

```
    @Override
    public int sub(int i, int j) {

        int result = i - j;

        System.out.println("方法内部 result = " + result);

        return result;
    }

    @Override
    public int mul(int i, int j) {

        int result = i * j;

        System.out.println("方法内部 result = " + result);

        return result;
    }

    @Override
    public int div(int i, int j) {

        int result = i / j;

        System.out.println("方法内部 result = " + result);

        return result;
    }
}
```

3.1.3、创建带日志功能的实现类




```
public class CalculatorLogImpl implements Calculator {

    @Override
    public int add(int i, int j) {

        System.out.println("[日志] add 方法开始了，参数是：" + i + "," + j);

        int result = i + j;
```

```
        System.out.println("方法内部 result = " + result);

        System.out.println("[日志] add 方法结束了，结果是：" + result);

        return result;
    }

    @Override
    public int sub(int i, int j) {

        System.out.println("[日志] sub 方法开始了，参数是：" + i + "," + j);

        int result = i - j;

        System.out.println("方法内部 result = " + result);

        System.out.println("[日志] sub 方法结束了，结果是：" + result);

        return result;
    }

    @Override
    public int mul(int i, int j) {

        System.out.println("[日志] mul 方法开始了，参数是：" + i + "," + j);

        int result = i * j;

        System.out.println("方法内部 result = " + result);

        System.out.println("[日志] mul 方法结束了，结果是：" + result);

        return result;
    }

    @Override
    public int div(int i, int j) {

        System.out.println("[日志] div 方法开始了，参数是：" + i + "," + j);

        int result = i / j;

        System.out.println("方法内部 result = " + result);

        System.out.println("[日志] div 方法结束了，结果是：" + result);

        return result;
    }
}
```

3.1.4、提出问题

①现有代码缺陷

针对带日志功能的实现类，我们发现有如下缺陷：

- 对核心业务功能有干扰，导致程序员在开发核心业务功能时分散了精力
- 附加功能分散在各个业务功能方法中，不利于统一维护

②解决思路

解决这两个问题，核心就是：解耦。我们需要把附加功能从业务功能代码中抽取出来。

③困难

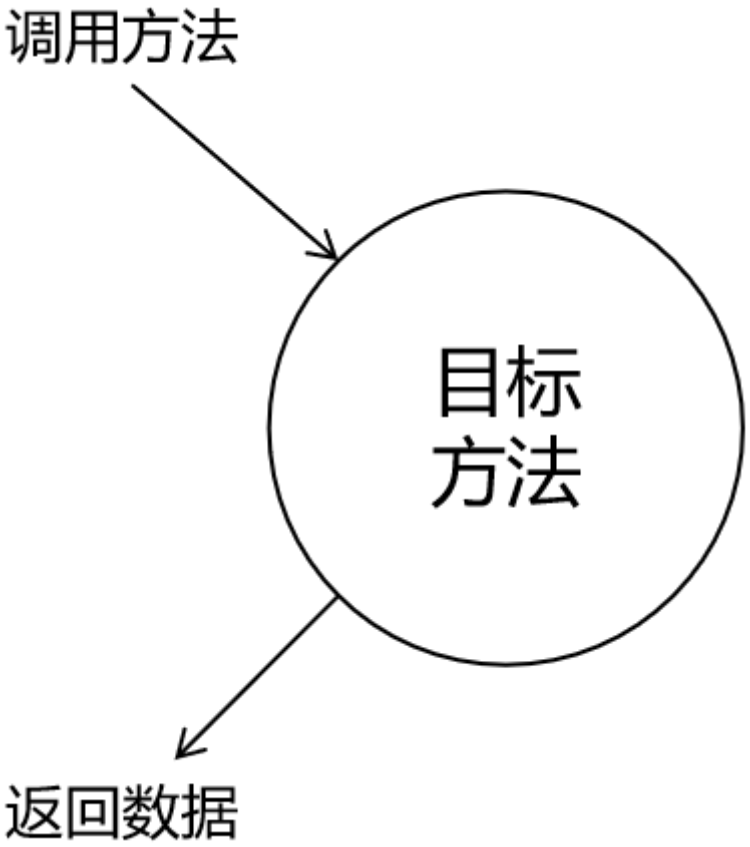
解决问题的困难：要抽取的代码在方法内部，靠以前把子类中的重复代码抽取到父类的方式没法解决。所以需要引入新的技术。

3.2、代理模式

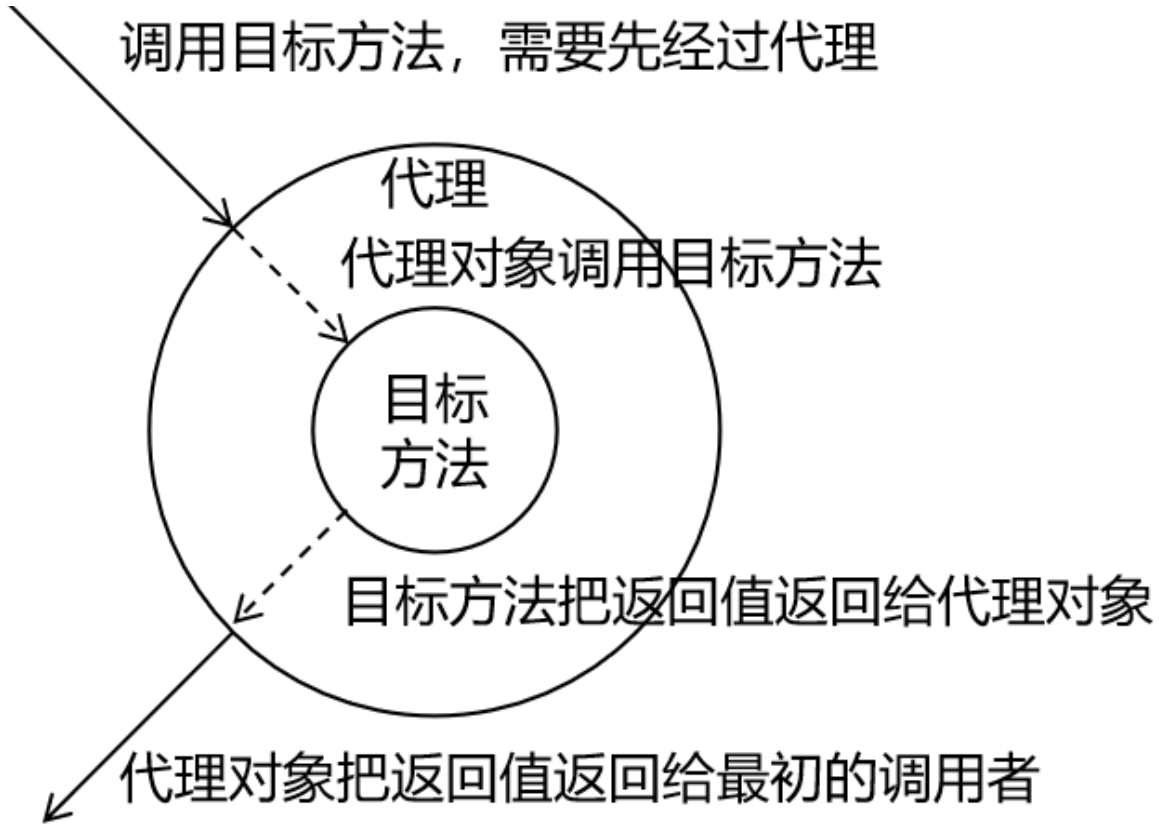
3.2.1、概念

①介绍

二十三种设计模式中的一种，属于结构型模式。它的作用就是通过提供一个代理类，让我们在调用目标方法的时候，不再是直接对目标方法进行调用，而是通过代理类**间接**调用。让不属于目标方法核心逻辑的代码从目标方法中剥离出来——**解耦**。调用目标方法时先调用代理对象的方法，减少对目标方法的调用和打扰，同时让附加功能能够集中在一起也有利于统一维护。



使用代理后：



②生活中的代理

- 广告商找大明星拍广告需要经过经纪人
- 合作伙伴找大老板谈合作要约见面时间需要经过秘书
- 房产中介是买卖双方的代理

③相关术语

- 代理：将非核心逻辑剥离出来以后，封装这些非核心逻辑的类、对象、方法。
- 目标：被代理“套用”了非核心逻辑代码的类、对象、方法。

3.2.2、静态代理

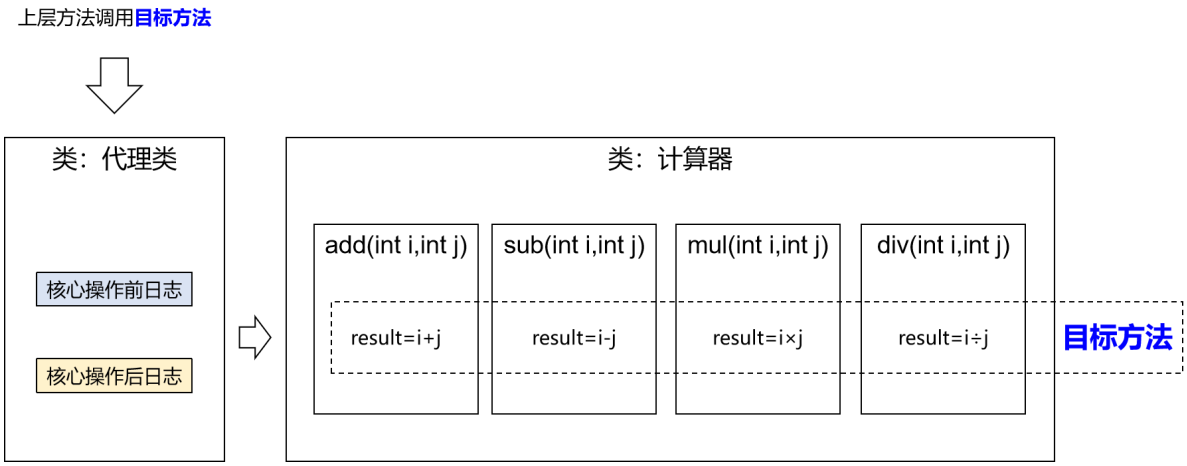
创建静态代理类：

```
public class CalculatorStaticProxy implements Calculator {

    // 将被代理的目标对象声明为成员变量
    private Calculator target;

    public CalculatorStaticProxy(Calculator target) {
        this.target = target;
    }

    @Override
    public int add(int i, int j) {

        // 附加功能由代理类中的代理方法来实现
        System.out.println("[日志] add 方法开始了，参数是：" + i + "," + j);

        // 通过目标对象来实现核心业务逻辑
        int addResult = target.add(i, j);

        System.out.println("[日志] add 方法结束了，结果是：" + addResult);
```

```
        return addResult;
    }
}
```

> 静态代理确实实现了解耦，但是由于代码都写死了，完全不具备任何的灵活性。就拿日志功能来说，将来其他地方也需要附加日志，那还得再声明更多个静态代理类，那就产生了大量重复的代码，日志功能还是分散的，没有统一管理。
>
> 提出进一步的需求：将日志功能集中到一个代理类中，将来有任何日志需求，都通过这一个代理类来实现。这就需要使用动态代理技术了。

3.2.3、动态代理



生产代理对象的工厂类：

```
public class ProxyFactory {

    private Object target;

    public ProxyFactory(Object target) {
        this.target = target;
    }

    public Object getProxy(){

        /**
         * newProxyInstance()：创建一个代理实例
         * 其中有三个参数：
         * 1、classLoader：加载动态生成的代理类的类加载器
         * 2、interfaces：目标对象实现的所有接口的class对象所组成的数组
         * 3、invocationHandler：设置代理对象实现目标对象方法的过程，即代理类中如何重写接口中的抽象方法
         */
        ClassLoader classLoader = target.getClass().getClassLoader();
        Class<?>[] interfaces = target.getClass().getInterfaces();
        InvocationHandler invocationHandler = new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                /**
                 * proxy：代理对象
                 * method：代理对象需要实现的方法，即其中需要重写的方法
                 * args：method所对应方法的参数
```

```
                 */
                Object result = null;
                try {
                    System.out.println("[动态代理][日志] "+method.getName()+"，参
数："+ Arrays.toString(args));
                    result = method.invoke(target, args);
                    System.out.println("[动态代理][日志] "+method.getName()+"，结
果："+ result);
                } catch (Exception e) {
                    e.printStackTrace();
                    System.out.println("[动态代理][日志] "+method.getName()+"，异
常："+e.getMessage());
                } finally {
                    System.out.println("[动态代理][日志] "+method.getName()+"，方法
执行完毕");
                }
                return result;
            }
        };

        return Proxy.newProxyInstance(classLoader, interfaces,
invocationHandler);
    }
}
```

3.2.4、测试

```
@Test
public void testDynamicProxy(){
    ProxyFactory factory = new ProxyFactory(new CalculatorLogImpl());
    Calculator proxy = (Calculator) factory.getProxy();
    proxy.div(1,0);
    //proxy.div(1,1);
}
```

3.3、AOP概念及相关术语

3.3.1、概述

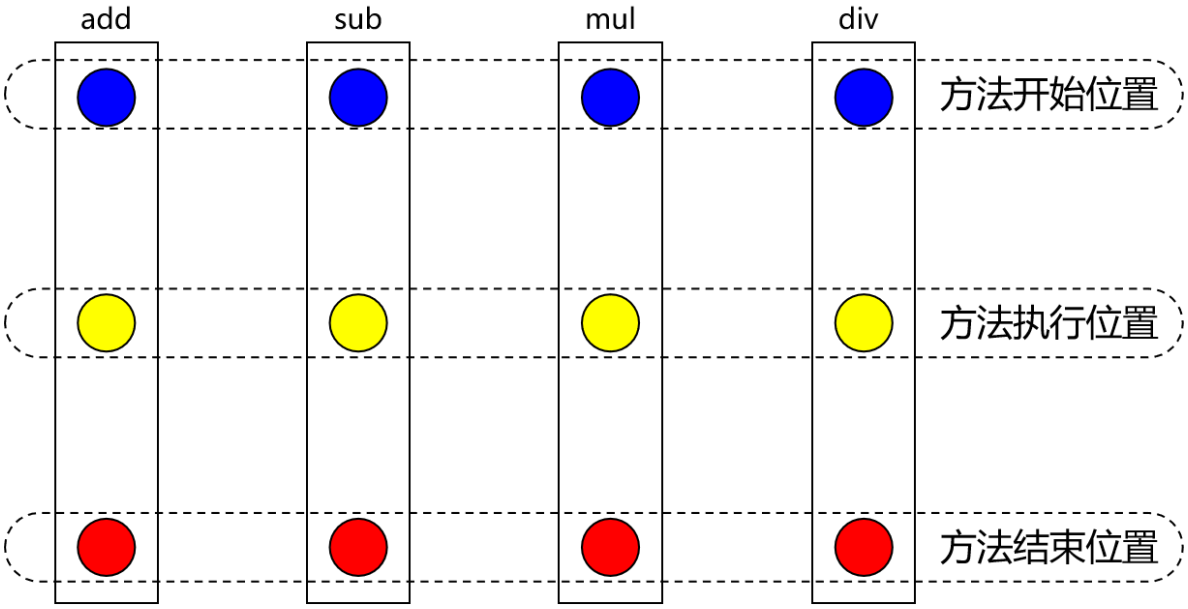
AOP（Aspect Oriented Programming）是一种设计思想，是软件设计领域中的面向切面编程，它是面向对象编程的一种补充和完善，它以通过预编译方式和运行期动态代理方式实现在不修改源代码的情况下给程序动态统一添加额外功能的一种技术。

3.3.2、相关术语

①横切关注点

从每个方法中抽取出来的同一类非核心业务。在同一个项目中，我们可以使用多个横切关注点对相关方法进行多个不同方面的增强。

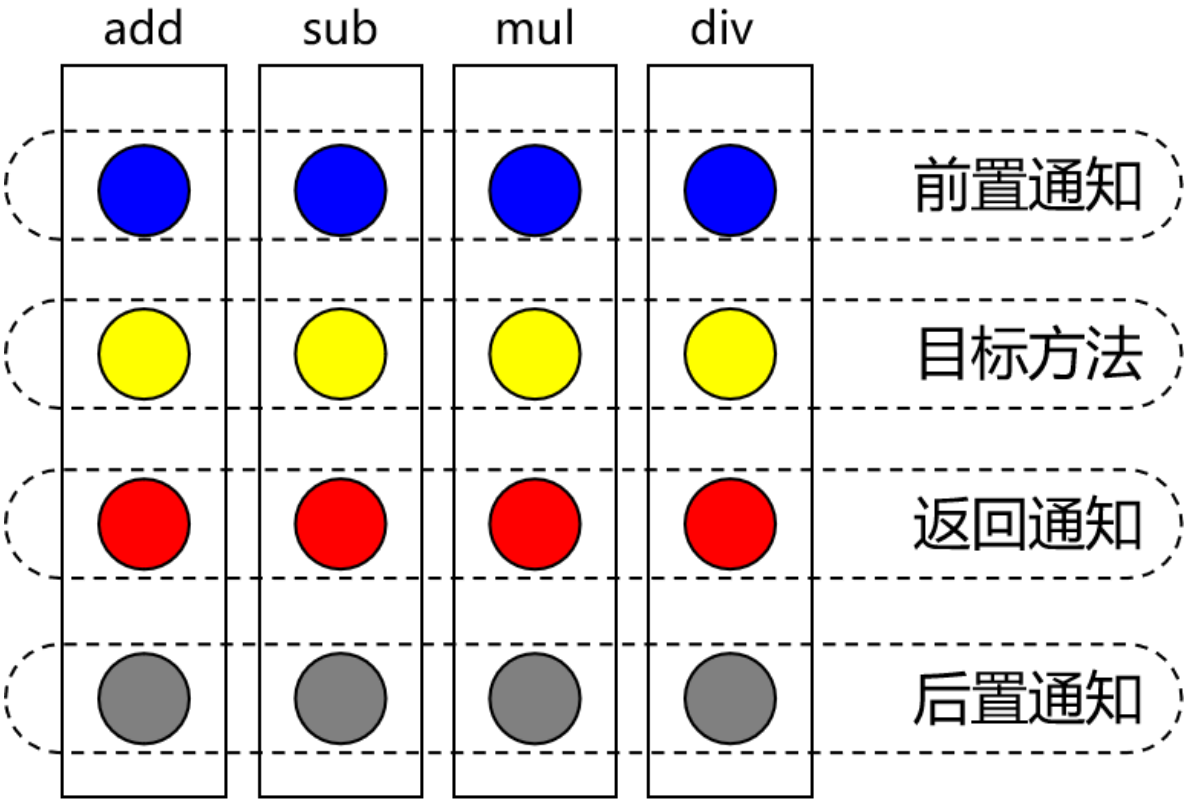
这个概念不是语法层面天然存在的，而是根据附加功能的逻辑上的需要：有十个附加功能，就有十个横切关注点。



②通知

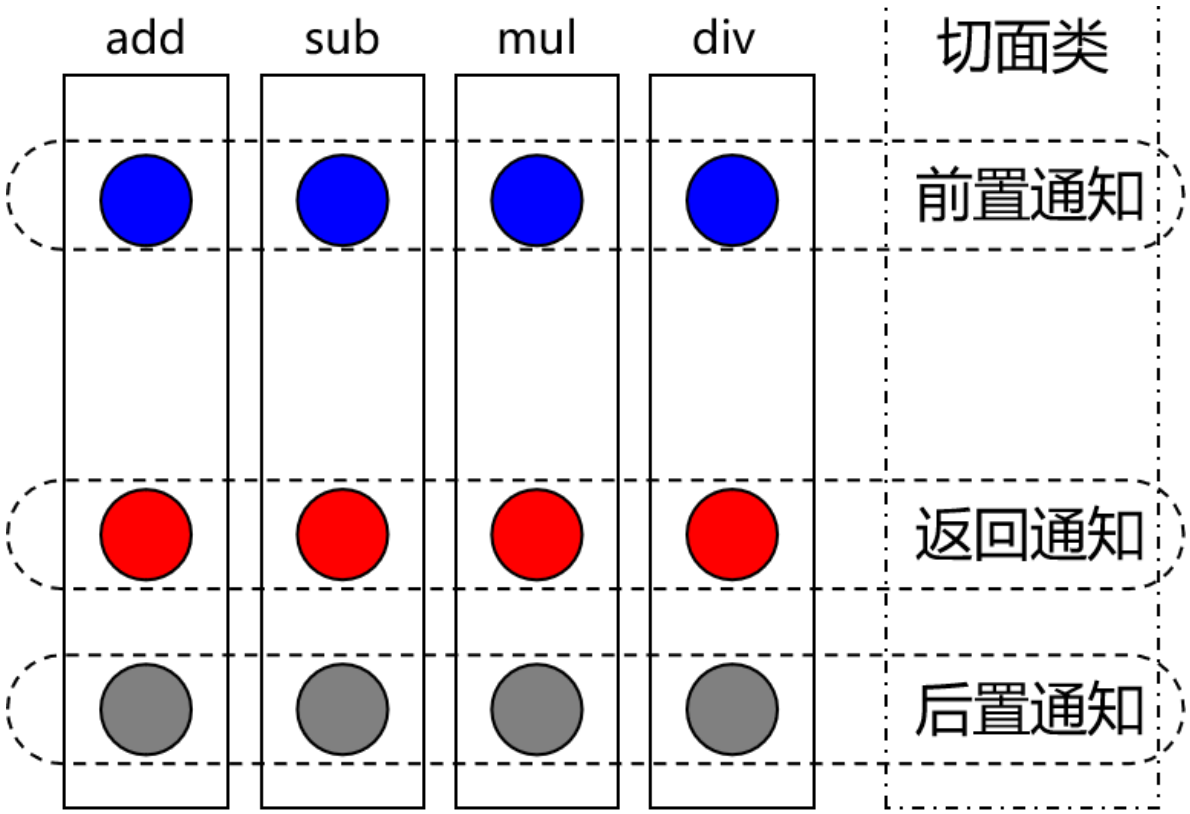
每一个横切关注点上要做的事情都需要写一个方法来实现，这样的方法就叫通知方法。

- 前置通知：在被代理的目标方法**前**执行
- 返回通知：在被代理的目标方法**成功结束**后执行（**寿终正寝**）
- 异常通知：在被代理的目标方法**异常结束**后执行（**死于非命**）
- 后置通知：在被代理的目标方法**最终结束**后执行（**盖棺定论**）
- 环绕通知：使用try...catch...finally结构围绕**整个**被代理的目标方法，包括上面四种通知对应的所有位置



③切面

封装通知方法的类。



④目标

被代理的目标对象。

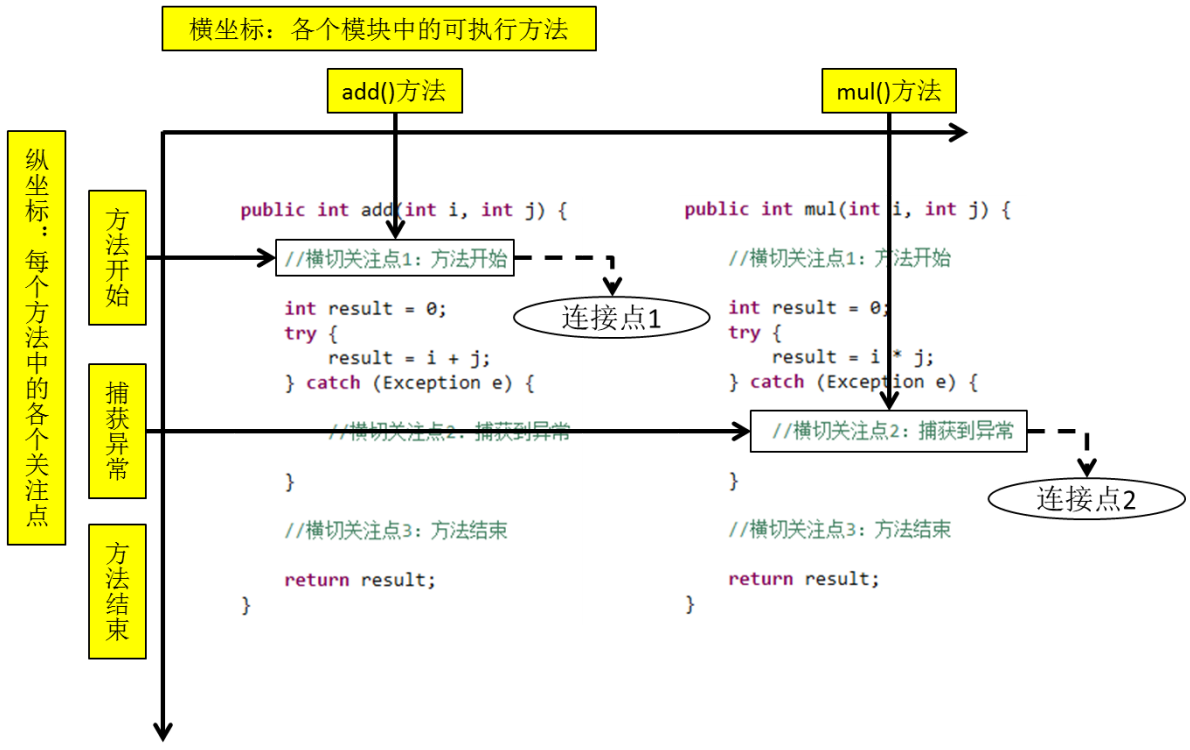
⑤代理

向目标对象应用通知之后创建的代理对象。

⑥连接点

这也是一个纯逻辑概念，不是语法定义的。

把方法排成一排，每一个横切位置看成x轴方向，把方法从上到下执行的顺序看成y轴，x轴和y轴的交叉点就是连接点。



⑦切入点

定位连接点的方式。

每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物（从逻辑上来说）。

如果把连接点看作数据库中的记录，那么切入点就是查询记录的 SQL 语句。

Spring 的 AOP 技术可以通过切入点定位到特定的连接点。

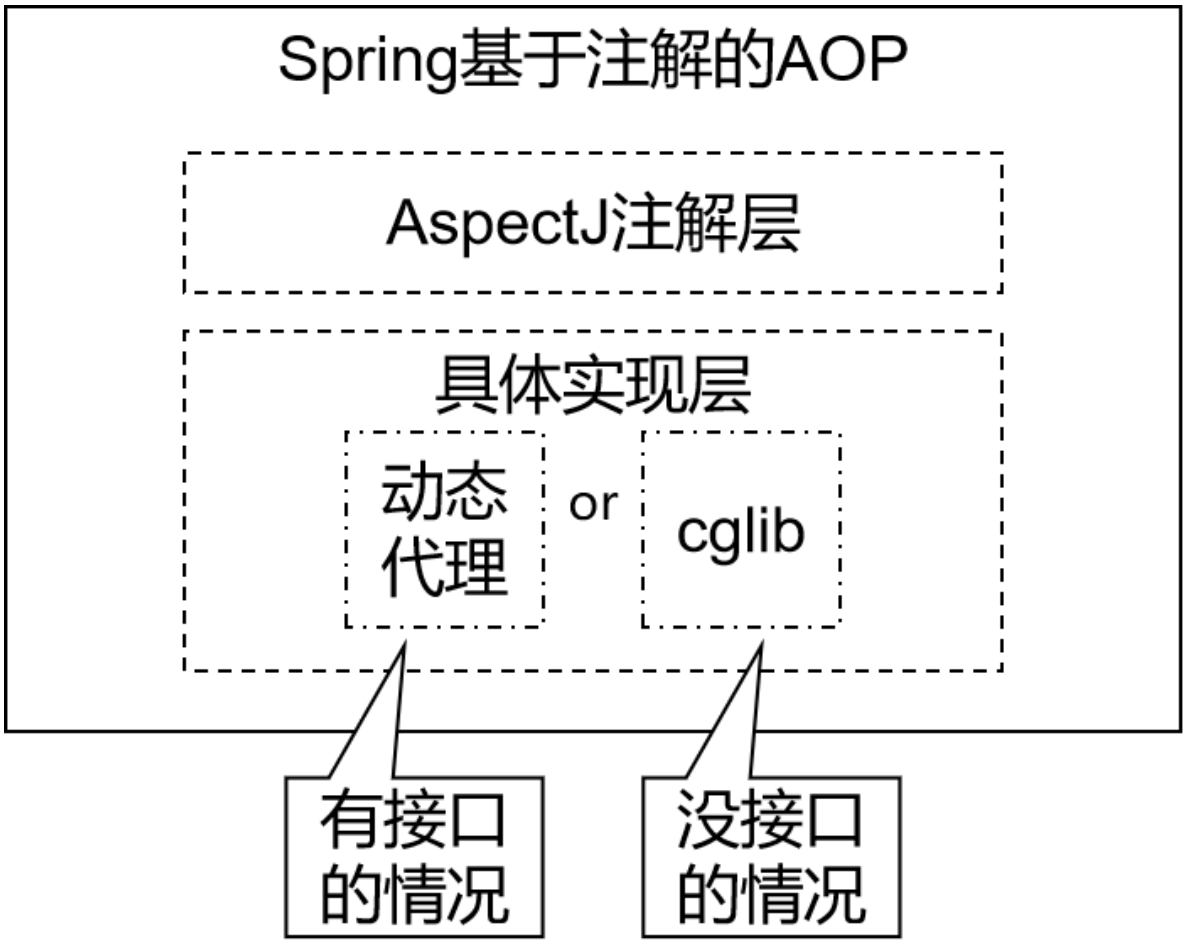
切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

3.3.3、作用

- 简化代码：把方法中固定位置的重复的代码**抽取**出来，让被抽取的方法更专注于自己的核心功能，提高内聚性。
- 代码增强：把特定的功能封装到切面类中，看哪里有需要，就往上套，被**套用**了切面逻辑的方法就被切面给增强了。

3.4、基于注解的AOP

3.4.1、技术说明



- 动态代理（InvocationHandler）：JDK原生的实现方式，需要被代理的目标类必须实现接口。因为这个技术要求**代理对象和目标对象实现同样的接口**（兄弟两个拜把子模式）。
- cglib：通过**继承被代理的目标类**（认干爹模式）实现代理，所以不需要目标类实现接口。
- AspectJ：本质上是静态代理，**将代理逻辑"织入"被代理的目标类编译得到的字节码文件**，所以最终效果是动态的。weaver就是织入器。Spring只是借用了AspectJ中的注解。

3.4.2、准备工作

①添加依赖

在IOC所需依赖基础上再加入下面依赖即可：

```
<!-- spring-aspects会帮我们传递过来aspectjweaver -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.3.1</version>
</dependency>
```

②准备被代理的目标资源

接口：

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);

}
```

实现类：

```
@Component
public class CalculatorPureImpl implements Calculator {

    @Override
    public int add(int i, int j) {

        int result = i + j;

        System.out.println("方法内部 result = " + result);

        return result;
    }

    @Override
    public int sub(int i, int j) {

        int result = i - j;

        System.out.println("方法内部 result = " + result);

        return result;
    }

    @Override
    public int mul(int i, int j) {

        int result = i * j;
```

```
        System.out.println("方法内部 result = " + result);

        return result;
    }

    @Override
    public int div(int i, int j) {

        int result = i / j;

        System.out.println("方法内部 result = " + result);

        return result;
    }
}
```

3.4.3、创建切面类并配置

```
// @Aspect表示这个类是一个切面类
@Aspect
// @Component注解保证这个切面类能够放入IOC容器
@Component
public class LogAspect {

    @Before("execution(public int com.atguigu.aop.annotation.CalculatorImpl.*
(..))")
    public void beforeMethod(JoinPoint joinPoint){
        String methodName = joinPoint.getSignature().getName();
        String args = Arrays.toString(joinPoint.getArgs());
        System.out.println("Logger-->前置通知，方法名："+methodName+"，参
数："+args);
    }

    @After("execution(* com.atguigu.aop.annotation.CalculatorImpl.*(..))")
    public void afterMethod(JoinPoint joinPoint){
        String methodName = joinPoint.getSignature().getName();
        System.out.println("Logger-->后置通知，方法名："+methodName);
    }

    @AfterReturning(value = "execution(*
com.atguigu.aop.annotation.CalculatorImpl.*(..))", returning = "result")
    public void afterReturningMethod(JoinPoint joinPoint, Object result){
        String methodName = joinPoint.getSignature().getName();
        System.out.println("Logger-->返回通知，方法名："+methodName+"，结
果："+result);
    }

    @AfterThrowing(value = "execution(*
com.atguigu.aop.annotation.CalculatorImpl.*(..))", throwing = "ex")
    public void afterThrowingMethod(JoinPoint joinPoint, Throwable ex){
        String methodName = joinPoint.getSignature().getName();
        System.out.println("Logger-->异常通知，方法名："+methodName+"，异常："+ex);
    }

    @Around("execution(* com.atguigu.aop.annotation.CalculatorImpl.*(..))")
    public Object aroundMethod(ProceedingJoinPoint joinPoint){
```

```
        String methodName = joinPoint.getSignature().getName();
        String args = Arrays.toString(joinPoint.getArgs());
        Object result = null;
        try {
            System.out.println("环绕通知-->目标对象方法执行之前");
            //目标对象（连接点）方法的执行
            result = joinPoint.proceed();
            System.out.println("环绕通知-->目标对象方法返回值之后");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
            System.out.println("环绕通知-->目标对象方法出现异常时");
        } finally {
            System.out.println("环绕通知-->目标对象方法执行完毕");
        }
        return result;
    }

}
```

在Spring的配置文件中配置：

```
    <!--
        基于注解的AOP的实现：
        1、将目标对象和切面交给IOC容器管理（注解+扫描）
        2、开启AspectJ的自动代理，为目标对象自动生成代理
        3、将切面类通过注解@Aspect标识
    -->

    <context:component-scan base-package="com.atguigu.aop.annotation">
</context:component-scan>

    <aop:aspectj-autoproxy />
```

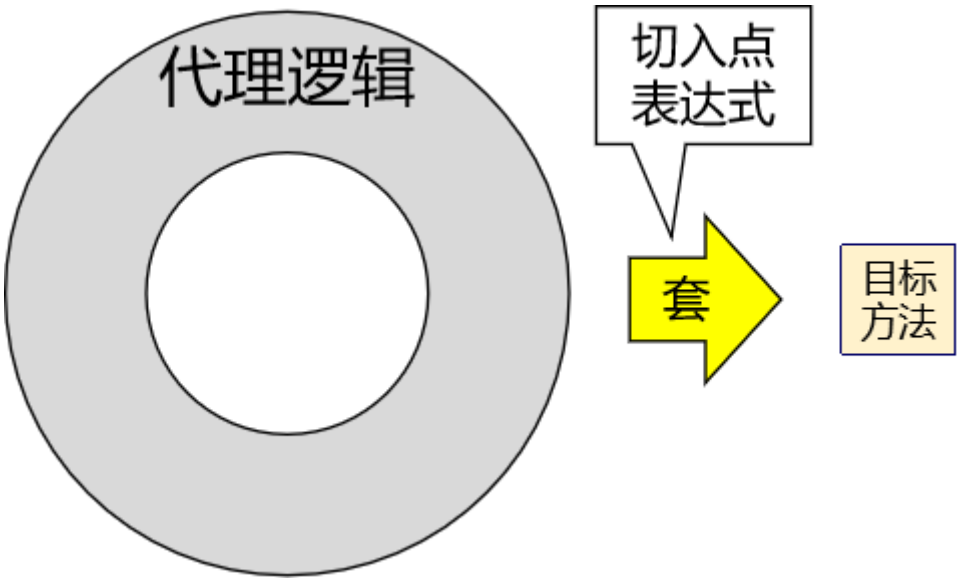
3.4.4、各种通知

- 前置通知：使用@Before注解标识，在被代理的目标方法**前**执行
- 返回通知：使用@AfterReturning注解标识，在被代理的目标方法**成功结束**后执行（**寿终正寝**）
- 异常通知：使用@AfterThrowing注解标识，在被代理的目标方法**异常结束**后执行（**死于非命**）
- 后置通知：使用@After注解标识，在被代理的目标方法**最终结束**后执行（**盖棺定论**）
- 环绕通知：使用@Around注解标识，使用try...catch...finally结构围绕**整个**被代理的目标方法，包括上面四种通知对应的所有位置

> 各种通知的执行顺序：
>
> - Spring版本5.3.x以前：
>   - 前置通知
>   - 目标操作
>   - 后置通知
>   - 返回通知或异常通知
> - Spring版本5.3.x以后：
>   - 前置通知
>   - 目标操作
>   - 返回通知或异常通知
>   - 后置通知

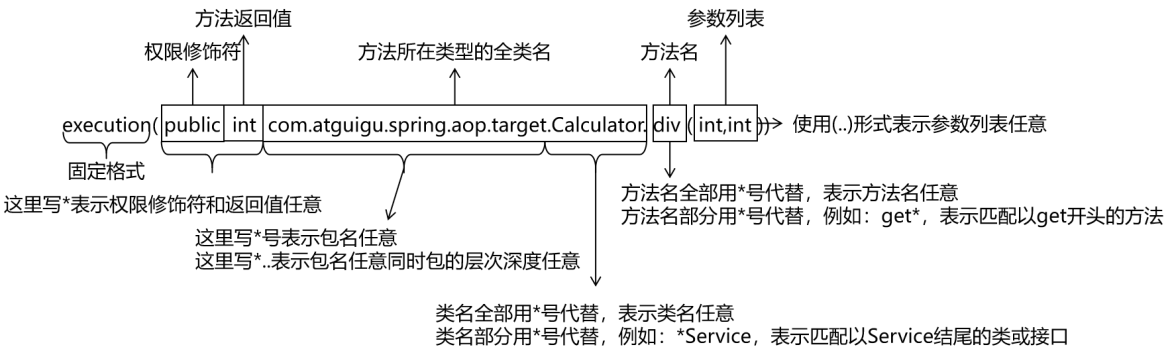
3.4.5、切入点表达式语法

①作用



②语法细节

- 用*号代替“权限修饰符”和“返回值”部分表示“权限修饰符”和“返回值”不限
- 在包名的部分，一个“*”号只能代表包的层次结构中的一层，表示这一层是任意的。
  - 例如：*.Hello匹配com.Hello，不匹配com.atguigu.Hello
- 在包名的部分，使用“*..”表示包名任意、包的层次深度任意
- 在类名的部分，类名部分整体用*号代替，表示类名任意
- 在类名的部分，可以使用*号代替类名的一部分
  - 例如：*Service匹配所有名称以Service结尾的类或接口
- 在方法名部分，可以使用*号表示方法名任意
- 在方法名部分，可以使用*号代替方法名的一部分
  - 例如：*Operation匹配所有方法名以Operation结尾的方法
- 在方法参数列表部分，使用(..)表示参数列表任意
- 在方法参数列表部分，使用(int,..)表示参数列表以一个int类型的参数开头
- 在方法参数列表部分，基本数据类型和对应的包装类型是不一样的
  - 切入点表达式中使用 int 和实际方法中 Integer 是不匹配的
- 在方法返回值部分，如果想要明确指定一个返回值类型，那么必须同时写明权限修饰符
  - 例如：execution(public int ..Service.*(.., int)) 正确
  - 例如：execution(* int ..Service.*(.., int)) 错误



3.4.6、重用切入点表达式

①声明

```
@Pointcut("execution(* com.atguigu.aop.annotation.*.*(..))")
public void pointCut(){}
```

②在同一个切面中使用

```
@Before("pointCut()")
public void beforeMethod(JoinPoint joinPoint){
    String methodName = joinPoint.getSignature().getName();
    String args = Arrays.toString(joinPoint.getArgs());
    System.out.println("Logger-->前置通知，方法名："+methodName+"，参数："+args);
}
```

③在不同切面中使用

```
@Before("com.atguigu.aop.CommonPointCut.pointCut()")
public void beforeMethod(JoinPoint joinPoint){
    String methodName = joinPoint.getSignature().getName();
    String args = Arrays.toString(joinPoint.getArgs());
    System.out.println("Logger-->前置通知，方法名："+methodName+"，参数："+args);
}
```

3.4.7、获取通知的相关信息

①获取连接点信息

获取连接点信息可以在通知方法的参数位置设置JoinPoint类型的形参

```
@Before("execution(public int com.atguigu.aop.annotation.CalculatorImpl.*(..))")
public void beforeMethod(JoinPoint joinPoint){
    //获取连接点的签名信息
    String methodName = joinPoint.getSignature().getName();
    //获取目标方法到的实参信息
    String args = Arrays.toString(joinPoint.getArgs());
    System.out.println("Logger-->前置通知，方法名："+methodName+"，参数："+args);
}
```

②获取目标方法的返回值

@AfterReturning中的属性returning，用来将通知方法的某个形参，接收目标方法的返回值

```
@AfterReturning(value = "execution(* com.atguigu.aop.annotation.CalculatorImpl.*
(..))", returning = "result")
public void afterReturningMethod(JoinPoint joinPoint, Object result){
    String methodName = joinPoint.getSignature().getName();
    System.out.println("Logger-->返回通知，方法名："+methodName+"，结果："+result);
}
```

③获取目标方法的异常

@AfterThrowing中的属性throwing，用来将通知方法的某个形参，接收目标方法的异常

```
@AfterThrowing(value = "execution(* com.atguigu.aop.annotation.CalculatorImpl.*(..))", throwing = "ex")
public void afterThrowingMethod(JoinPoint joinPoint, Throwable ex){
    String methodName = joinPoint.getSignature().getName();
    System.out.println("Logger-->异常通知，方法名："+methodName+"，异常："+ex);
}
```

3.4.8、环绕通知

```
@Around("execution(* com.atguigu.aop.annotation.CalculatorImpl.*(..))")
public Object aroundMethod(ProceedingJoinPoint joinPoint){
    String methodName = joinPoint.getSignature().getName();
    String args = Arrays.toString(joinPoint.getArgs());
    Object result = null;
    try {
        System.out.println("环绕通知-->目标对象方法执行之前");
        //目标方法的执行，目标方法的返回值一定要返回给外界调用者
        result = joinPoint.proceed();
        System.out.println("环绕通知-->目标对象方法返回值之后");
    } catch (Throwable throwable) {
        throwable.printStackTrace();
        System.out.println("环绕通知-->目标对象方法出现异常时");
    } finally {
        System.out.println("环绕通知-->目标对象方法执行完毕");
    }
    return result;
}
```

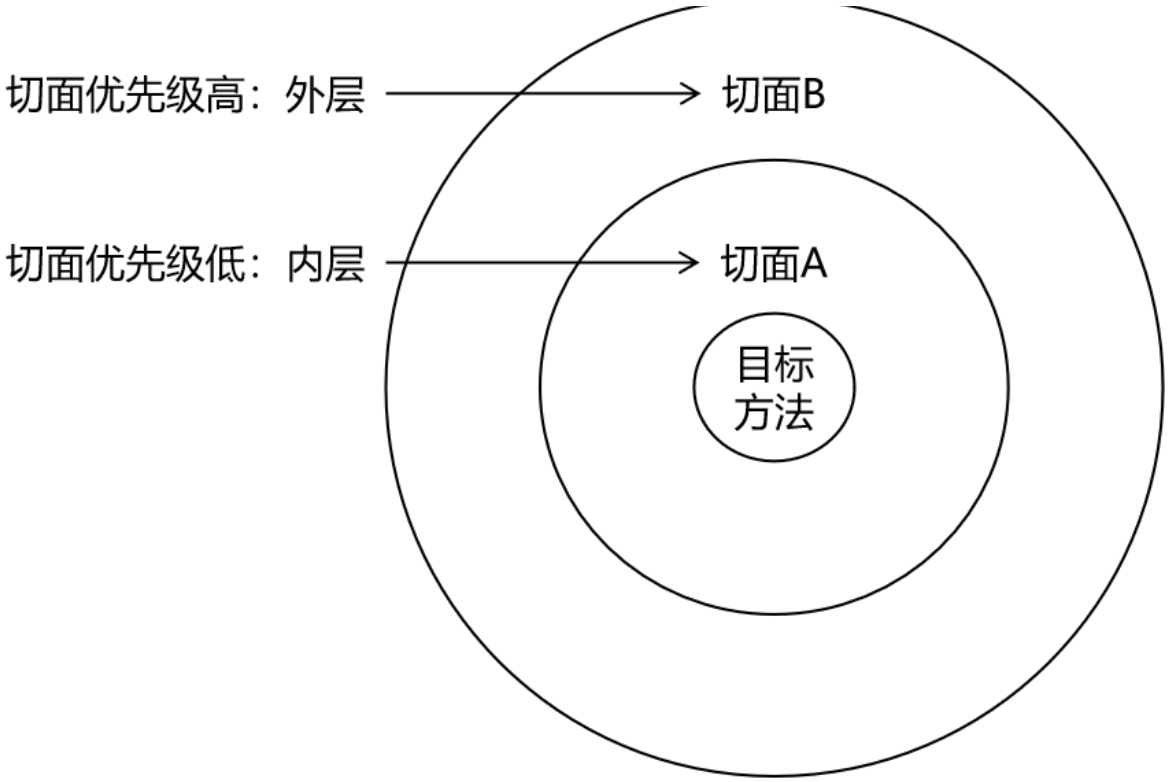
3.4.9、切面的优先级

相同目标方法上同时存在多个切面时，切面的优先级控制切面的**内外嵌套**顺序。

- 优先级高的切面：外面
- 优先级低的切面：里面

使用@Order注解可以控制切面的优先级：

- @Order(较小的数)：优先级高
- @Order(较大的数)：优先级低



3.5、基于XML的AOP（了解）

3.5.1、准备工作

参考基于注解的AOP环境

3.5.2、实现

```
<context:component-scan base-package="com.atguigu.aop.xml"></context:component-
scan>

<aop:config>
    <!--配置切面类-->
    <aop:aspect ref="loggerAspect">
        <aop:pointcut id="pointCut" expression="execution(*
com.atguigu.aop.xml.CalculatorImpl.*(..))"/>
        <aop:before method="beforeMethod" pointcut-ref="pointCut"></aop:before>
        <aop:after method="afterMethod" pointcut-ref="pointCut"></aop:after>
        <aop:after-returning method="afterReturningMethod" returning="result"
pointcut-ref="pointCut"></aop:after-returning>
        <aop:after-throwing method="afterThrowingMethod" throwing="ex" pointcut-
ref="pointCut"></aop:after-throwing>
        <aop:around method="aroundMethod" pointcut-ref="pointCut"></aop:around>
    </aop:aspect>
    <aop:aspect ref="validateAspect" order="1">
        <aop:before method="validateBeforeMethod" pointcut-ref="pointCut">
</aop:before>
    </aop:aspect>
</aop:config>
```

4、声明式事务

4.1、JdbcTemplate

4.1.1、简介

Spring 框架对 JDBC 进行封装，使用 JdbcTemplate 方便实现对数据库操作

4.1.2、准备工作

①加入依赖

```
<dependencies>

    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- Spring 持久化层支持jar包 -->
    <!-- Spring 在执行持久化层操作、与持久化层技术进行整合过程中，需要使用orm、jdbc、tx三个
jar包 -->
    <!-- 导入 orm 包就可以通过 Maven 的依赖传递性把其他两个也导入 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- Spring 测试相关 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
    <!-- 数据源 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.0.31</version>
    </dependency>

</dependencies>
```

②创建jdbc.properties

```
jdbc.user=root
jdbc.password=atguigu
jdbc.url=jdbc:mysql://localhost:3306/ssm
jdbc.driver=com.mysql.cj.jdbc.Driver
```

③配置Spring的配置文件

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

<!-- 配置数据源 -->
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${atguigu.url}"/>
    <property name="driverClassName" value="${atguigu.driver}"/>
    <property name="username" value="${atguigu.username}"/>
    <property name="password" value="${atguigu.password}"/>
</bean>

<!-- 配置 JdbcTemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 装配数据源 -->
    <property name="dataSource" ref="druidDataSource"/>
</bean>
```

4.1.3、测试

①在测试类装配 JdbcTemplate

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:spring-jdbc.xml")
public class JDBCTemplateTest {

    @Autowired
    private JdbcTemplate jdbcTemplate;

}
```

②测试增删改功能

```
@Test
//测试增删改功能
public void testUpdate(){
    String sql = "insert into t_emp values(null,?,?,?)";
    int result = jdbcTemplate.update(sql, "张三", 23, "男");
    System.out.println(result);
}
```

③查询一条数据为实体类对象

```
@Test
//查询一条数据为一个实体类对象
public void testSelectEmpById(){
    String sql = "select * from t_emp where id = ?";
    Emp emp = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>
(Emp.class), 1);
    System.out.println(emp);
}
```

④查询多条数据为一个list集合

```
@Test
//查询多条数据为一个list集合
public void testSelectList(){
    String sql = "select * from t_emp";
    List<Emp> list = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(Emp.class));
    list.forEach(emp -> System.out.println(emp));
}
```

⑤查询单行单列的值

```
@Test
//查询单行单列的值
public void selectCount(){
    String sql = "select count(id) from t_emp";
    Integer count = jdbcTemplate.queryForObject(sql, Integer.class);
    System.out.println(count);
}
```

4.2、声明式事务概念

4.2.1、编程式事务

事务功能的相关操作全部通过自己编写代码来实现：

```
Connection conn = ...;

try {

    // 开启事务：关闭事务的自动提交
    conn.setAutoCommit(false);

    // 核心操作

    // 提交事务
    conn.commit();

}catch(Exception e){

    // 回滚事务
    conn.rollBack();

}finally{
```

```
        // 释放数据库连接
        conn.close();

    }
```

编程式的实现方式存在缺陷：

- 细节没有被屏蔽：具体操作过程中，所有细节都需要程序员自己来完成，比较繁琐。
- 代码复用性不高：如果没有有效抽取出来，每次实现功能都需要自己编写代码，代码就没有得到复用。

4.2.2、声明式事务

既然事务控制的代码有规律可循，代码的结构基本是确定的，所以框架就可以将固定模式的代码抽取出来，进行相关的封装。

封装起来后，我们只需要在配置文件中进行简单的配置即可完成操作。

- 好处1：提高开发效率
- 好处2：消除了冗余的代码
- 好处3：框架会综合考虑相关领域中在实际开发环境下有可能遇到的各种问题，进行了健壮性、性能等各个方面的优化

所以，我们可以总结下面两个概念：

- **编程式**：**自己写代码**实现功能
- **声明式**：通过**配置**让**框架**实现功能

4.3、基于注解的声明式事务

4.3.1、准备工作

①加入依赖

```
<dependencies>

    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- Spring 持久化层支持jar包 -->
    <!-- Spring 在执行持久化层操作、与持久化层技术进行整合过程中，需要使用orm、jdbc、tx三个
jar包 -->
    <!-- 导入 orm 包就可以通过 Maven 的依赖传递性把其他两个也导入 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- Spring 测试相关 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.3.1</version>
```

```
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
    <!-- 数据源 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.0.31</version>
    </dependency>

</dependencies>
```

②创建jdbc.properties

```
jdbc.user=root
jdbc.password=atguigu
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.driver=com.mysql.cj.jdbc.Driver
```

③配置Spring的配置文件

```
<!--扫描组件-->
<context:component-scan base-package="com.atguigu.spring.tx.annotation">
</context:component-scan>

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

<!-- 配置数据源 -->
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<!-- 配置 JdbcTemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 装配数据源 -->
    <property name="dataSource" ref="druidDataSource"/>
</bean>
```

④创建表

```
CREATE TABLE `t_book` (
  `book_id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `book_name` varchar(20) DEFAULT NULL COMMENT '图书名称',
  `price` int(11) DEFAULT NULL COMMENT '价格',
  `stock` int(10) unsigned DEFAULT NULL COMMENT '库存（无符号）',
  PRIMARY KEY (`book_id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
insert  into `t_book`(`book_id`,`book_name`,`price`,`stock`) values (1,'斗破苍
穹',80,100),(2,'斗罗大陆',50,100);
CREATE TABLE `t_user` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `username` varchar(20) DEFAULT NULL COMMENT '用户名',
  `balance` int(10) unsigned DEFAULT NULL COMMENT '余额（无符号）',
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
insert  into `t_user`(`user_id`,`username`,`balance`) values (1,'admin',50);
```

⑤创建组件

创建BookController：

```
@Controller
public class BookController {

    @Autowired
    private BookService bookService;

    public void buyBook(Integer bookId, Integer userId){
        bookService.buyBook(bookId, userId);
    }
}
```

创建接口BookService：

```
public interface BookService {
    void buyBook(Integer bookId, Integer userId);
}
```

创建实现类BookServiceImpl：

```
@Service
public class BookServiceImpl implements BookService {

    @Autowired
    private BookDao bookDao;

    @Override
    public void buyBook(Integer bookId, Integer userId) {
        //查询图书的价格
        Integer price = bookDao.getPriceByBookId(bookId);
        //更新图书的库存
        bookDao.updateStock(bookId);
        //更新用户的余额
        bookDao.updateBalance(userId, price);
    }
```

```
}
```

创建接口BookDao：

```
public interface BookDao {
    Integer getPriceByBookId(Integer bookId);

    void updateStock(Integer bookId);

    void updateBalance(Integer userId, Integer price);
}
```

创建实现类BookDaoImpl：

```
@Repository
public class BookDaoImpl implements BookDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public Integer getPriceByBookId(Integer bookId) {
        String sql = "select price from t_book where book_id = ?";
        return jdbcTemplate.queryForObject(sql, Integer.class, bookId);
    }

    @Override
    public void updateStock(Integer bookId) {
        String sql = "update t_book set stock = stock - 1 where book_id = ?";
        jdbcTemplate.update(sql, bookId);
    }

    @Override
    public void updateBalance(Integer userId, Integer price) {
        String sql = "update t_user set balance = balance - ? where user_id =
?";
        jdbcTemplate.update(sql, price, userId);
    }
}
```

4.3.2、测试无事务情况

①创建测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:tx-annotation.xml")
public class TxByAnnotationTest {

    @Autowired
    private BookController bookController;

    @Test
    public void testBuyBook(){
        bookController.buyBook(1, 1);
    }

}
```

②模拟场景

用户购买图书，先查询图书的价格，再更新图书的库存和用户的余额

假设用户id为1的用户，购买id为1的图书

用户余额为50，而图书价格为80

购买图书之后，用户的余额为-30，数据库中余额字段设置了无符号，因此无法将-30插入到余额字段

此时执行sql语句会抛出SQLException

③观察结果

因为没有添加事务，图书的库存更新了，但是用户的余额没有更新

显然这样的结果是错误的，购买图书是一个完整的功能，更新库存和更新余额要么都成功要么都失败

4.3.3、加入事务

①添加事务配置

在Spring的配置文件中添加配置：

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

<!--
    开启事务的注解驱动
    通过注解@Transactional所标识的方法或标识的类中所有的方法，都会被事务管理器管理事务
-->
<!-- transaction-manager属性的默认值是transactionManager，如果事务管理器bean的id正好就是这个默认值，则可以省略这个属性 -->
<tx:annotation-driven transaction-manager="transactionManager" />
```

注意：导入的名称空间需要 **tx 结尾**的那个。

②添加事务注解

因为service层表示业务逻辑层，一个方法表示一个完成的功能，因此处理事务一般在service层处理

在BookServiceImpl的buybook()添加注解@Transactional

③观察结果

由于使用了Spring的声明式事务，更新库存和更新余额都没有执行

4.3.4、@Transactional注解标识的位置

@Transactional标识在方法上，咋只会影响该方法

@Transactional标识的类上，咋会影响类中所有的方法

4.3.5、事务属性：只读

①介绍

对一个查询操作来说，如果我们把它设置成只读，就能够明确告诉数据库，这个操作不涉及写操作。这样数据库就能够针对查询操作来进行优化。

②使用方式

```
@Transactional(readOnly = true)
public void buyBook(Integer bookId, Integer userId) {
    //查询图书的价格
    Integer price = bookDao.getPriceByBookId(bookId);
    //更新图书的库存
    bookDao.updateStock(bookId);
    //更新用户的余额
    bookDao.updateBalance(userId, price);
    //System.out.println(1/0);
}
```

③注意

对增删改操作设置只读会抛出下面异常：

Caused by: java.sql.SQLException: Connection is read-only. Queries leading to data modification are not allowed

4.3.6、事务属性：超时

①介绍

事务在执行过程中，有可能因为遇到某些问题，导致程序卡住，从而长时间占用数据库资源。而长时间占用资源，大概率是因为程序运行出现了问题（可能是Java程序或MySQL数据库或网络连接等等）。

此时这个很可能出问题的程序应该被回滚，撤销它已做的操作，事务结束，把资源让出来，让其他正常程序可以执行。

概括来说就是一句话：超时回滚，释放资源。

②使用方式

```
@Transactional(timeout = 3)
public void buyBook(Integer bookId, Integer userId) {
    try {
        TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    //查询图书的价格
    Integer price = bookDao.getPriceByBookId(bookId);
    //更新图书的库存
    bookDao.updateStock(bookId);
    //更新用户的余额
    bookDao.updateBalance(userId, price);
    //System.out.println(1/0);
}
```

③观察结果

执行过程中抛出异常：

org.springframework.transaction.**TransactionTimedOutException**: Transaction timed out: deadline was Fri Jun 04 16:25:39 CST 2022

4.3.7、事务属性：回滚策略

①介绍

声明式事务默认只针对运行时异常回滚，编译时异常不回滚。

可以通过@Transactional中相关属性设置回滚策略

- rollbackFor属性：需要设置一个Class类型的对象
- rollbackForClassName属性：需要设置一个字符串类型的全类名
- noRollbackFor属性：需要设置一个Class类型的对象
- rollbackFor属性：需要设置一个字符串类型的全类名

②使用方式

```
@Transactional(noRollbackFor = ArithmeticException.class)
//@Transactional(noRollbackForClassName = "java.lang.ArithmeticException")
public void buyBook(Integer bookId, Integer userId) {
    //查询图书的价格
    Integer price = bookDao.getPriceByBookId(bookId);
    //更新图书的库存
    bookDao.updateStock(bookId);
    //更新用户的余额
    bookDao.updateBalance(userId, price);
    System.out.println(1/0);
}
```

③观察结果

虽然购买图书功能中出现了数学运算异常（ArithmeticException），但是我们设置的回滚策略是，当出现ArithmeticException不发生回滚，因此购买图书的操作正常执行

4.3.8、事务属性：事务隔离级别

①介绍

数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。一个事务与其他事务隔离的程度称为隔离级别。SQL标准中规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

隔离级别一共有四种：

- 读未提交：READ UNCOMMITTED

  允许Transaction01读取Transaction02未提交的修改。

- 读已提交：READ COMMITTED、

  要求Transaction01只能读取Transaction02已提交的修改。

- 可重复读：REPEATABLE READ

  确保Transaction01可以多次从一个字段中读取到相同的值，即Transaction01执行期间禁止其它事务对这个字段进行更新。

- 串行化：SERIALIZABLE

  确保Transaction01可以多次从一个表中读取到相同的行，在Transaction01执行期间，禁止其它事务对这个表进行添加、更新、删除操作。可以避免任何并发问题，但性能十分低下。

各个隔离级别解决并发问题的能力见下表：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	有	有	有
READ COMMITTED	无	有	有
REPEATABLE READ	无	无	有
SERIALIZABLE	无	无	无

各种数据库产品对事务隔离级别的支持程度：

隔离级别	Oracle	MySQL
READ UNCOMMITTED	×	√
READ COMMITTED	√(默认)	√
REPEATABLE READ	×	√(默认)
SERIALIZABLE	√	√

②使用方式

```
@Transactional(isolation = Isolation.DEFAULT)//使用数据库默认的隔离级别
@Transactional(isolation = Isolation.READ_UNCOMMITTED)//读未提交
@Transactional(isolation = Isolation.READ_COMMITTED)//读已提交
@Transactional(isolation = Isolation.REPEATABLE_READ)//可重复读
@Transactional(isolation = Isolation.SERIALIZABLE)//串行化
```

4.3.9、事务属性：事务传播行为

①介绍

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

②测试

创建接口CheckoutService：

```
public interface CheckoutService {
    void checkout(Integer[] bookIds, Integer userId);
}
```

创建实现类CheckoutServiceImpl：

```
@Service
public class CheckoutServiceImpl implements CheckoutService {

    @Autowired
    private BookService bookService;

    @Override
    @Transactional
    //一次购买多本图书
    public void checkout(Integer[] bookIds, Integer userId) {
        for (Integer bookId : bookIds) {
            bookService.buyBook(bookId, userId);
        }
    }
}
```

在BookController中添加方法：

```
@Autowired
private CheckoutService checkoutService;

public void checkout(Integer[] bookIds, Integer userId){
    checkoutService.checkout(bookIds, userId);
}
```

在数据库中将用户的余额修改为100元

③观察结果

可以通过@Transactional中的propagation属性设置事务传播行为

修改BookServiceImpl中buyBook()上，注解@Transactional的propagation属性

@Transactional(propagation = Propagation.REQUIRED)，默认情况，表示如果当前线程上有已经开启的事务可用，那么就在这个事务中运行。经过观察，购买图书的方法buyBook()在checkout()中被调用，checkout()上有事务注解，因此在此事务中执行。所购买的两本图书的价格为80和50，而用户的余额为100，因此在购买第二本图书时余额不足失败，导致整个checkout()回滚，即只要有一本书买不了，就都买不了

@Transactional(propagation = Propagation.REQUIRES_NEW)，表示不管当前线程上是否有已经开启的事务，都要开启新事务。同样的场景，每次购买图书都是在buyBook()的事务中执行，因此第一本图书购买成功，事务结束，第二本图书购买失败，只在第二次的buyBook()中回滚，购买第一本图书不受影响，即能买几本就买几本

4.4、基于XML的声明式事务

4.3.1、场景模拟

参考基于注解的声明式事务

4.3.2、修改Spring配置文件

将Spring配置文件中去掉tx:annotation-driven 标签，并添加配置：

```
<aop:config>
    <!-- 配置事务通知和切入点表达式 -->
    <aop:advisor advice-ref="txAdvice" pointcut="execution(*
com.atguigu.spring.tx.xml.service.impl.*.*(..))"></aop:advisor>
</aop:config>
<!-- tx:advice标签：配置事务通知 -->
<!-- id属性：给事务通知标签设置唯一标识，便于引用 -->
<!-- transaction-manager属性：关联事务管理器 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!-- tx:method标签：配置具体的事务方法 -->
        <!-- name属性：指定方法名，可以使用星号代表多个字符 -->
        <tx:method name="get*" read-only="true"/>
        <tx:method name="query*" read-only="true"/>
        <tx:method name="find*" read-only="true"/>

        <!-- read-only属性：设置只读属性 -->
        <!-- rollback-for属性：设置回滚的异常 -->
        <!-- no-rollback-for属性：设置不回滚的异常 -->
        <!-- isolation属性：设置事务的隔离级别 -->
        <!-- timeout属性：设置事务的超时属性 -->
        <!-- propagation属性：设置事务的传播行为 -->
        <tx:method name="save*" read-only="false" rollback-
for="java.lang.Exception" propagation="REQUIRES_NEW"/>
        <tx:method name="update*" read-only="false" rollback-
for="java.lang.Exception" propagation="REQUIRES_NEW"/>
        <tx:method name="delete*" read-only="false" rollback-
for="java.lang.Exception" propagation="REQUIRES_NEW"/>
    </tx:attributes>
</tx:advice>
```

> 注意：基于xml实现的声明式事务，必须引入aspectJ的依赖
>
> ```
> <dependency>
>  <groupId>org.springframework</groupId>
>  <artifactId>spring-aspects</artifactId>
>  <version>5.3.1</version>
> </dependency>
> ```

三、SpringMVC

1、SpringMVC简介

1.1、什么是MVC

MVC是一种软件架构的思想，将软件按照模型、视图、控制器来划分

M：Model，模型层，指工程中的JavaBean，作用是处理数据

JavaBean分为两类：

- 一类称为实体类Bean：专门存储业务数据的，如 Student、User 等
- 一类称为业务处理 Bean：指 Service 或 Dao 对象，专门用于处理业务逻辑和数据访问。

V：View，视图层，指工程中的html或jsp等页面，作用是与用户进行交互，展示数据

C：Controller，控制层，指工程中的servlet，作用是接收请求和响应浏览器

MVC的工作流程： 用户通过视图层发送请求到服务器，在服务器中请求被Controller接收，Controller调用相应的Model层处理请求，处理完毕将结果返回到Controller，Controller再根据请求处理的结果找到相应的View视图，渲染数据后最终响应给浏览器

1.2、什么是SpringMVC

SpringMVC是Spring的一个后续产品，是Spring的一个子项目

SpringMVC 是 Spring 为表述层开发提供的一整套完备的解决方案。在表述层框架历经 Strust、WebWork、Strust2 等诸多产品的历代更迭之后，目前业界普遍选择了 SpringMVC 作为 Java EE 项目表述层开发的**首选方案**。

> 注：三层架构分为表述层（或表示层）、业务逻辑层、数据访问层，表述层表示前台页面和后台servlet

1.3、SpringMVC的特点

- **Spring 家族原生产品**，与 IOC 容器等基础设施无缝对接
- **基于原生的Servlet**，通过了功能强大的**前端控制器DispatcherServlet**，对请求和响应进行统一处理
- 表述层各细分领域需要解决的问题**全方位覆盖**，提供**全面解决方案**
- **代码清新简洁**，大幅度提升开发效率
- 内部组件化程度高，可插拔式组件**即插即用**，想要什么功能配置相应组件即可
- **性能卓著**，尤其适合现代大型、超大型互联网项目要求

2、入门案例

2.1、开发环境

IDE：idea 2019.2

构建工具：maven3.5.4

服务器：tomcat8.5

Spring版本：5.3.1

2.2、创建maven工程

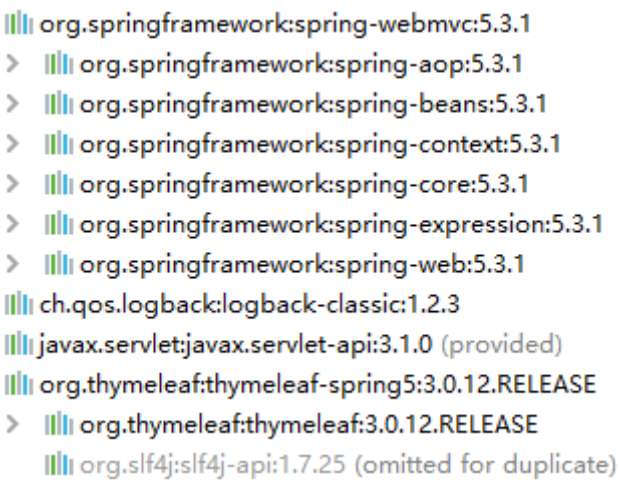
①添加web模块

②打包方式：war

③引入依赖

```
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- 日志 -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>

    <!-- ServletAPI -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>

    <!-- Spring5和Thymeleaf整合包 -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.0.12.RELEASE</version>
    </dependency>
</dependencies>
```

注：由于 Maven 的传递性，我们不必将所有需要的包全部配置依赖，而是配置最顶端的依赖，其他靠传递性导入。



2.3、配置web.xml

注册SpringMVC的前端控制器DispatcherServlet

①默认配置方式

此配置作用下，SpringMVC的配置文件默认位于WEB-INF下，默认名称为<servlet-name>-servlet.xml，例如，以下配置所对应SpringMVC的配置文件位于WEB-INF下，文件名为springMVC-servlet.xml

```
<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <!--
        设置springMVC的核心控制器所能处理的请求的请求路径
        /所匹配的请求可以是/login或.html或.js或.css方式的请求路径
        但是/不能匹配.jsp请求路径的请求
    -->
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

②扩展配置方式

可通过init-param标签设置SpringMVC配置文件的位置和名称，通过load-on-startup标签设置SpringMVC前端控制器DispatcherServlet的初始化时间

```
<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 通过初始化参数指定SpringMVC配置文件的位置和名称 -->
    <init-param>
        <!-- contextConfigLocation为固定值 -->
        <param-name>contextConfigLocation</param-name>
        <!-- 使用classpath:表示从类路径查找配置文件，例如maven工程中的src/main/resources -->
        <param-value>classpath:springMVC.xml</param-value>
    </init-param>
    <!--
        作为框架的核心组件，在启动过程中有大量的初始化操作要做
        而这些操作放在第一次请求时才执行会严重影响访问速度
        因此需要通过此标签将启动控制DispatcherServlet的初始化时间提前到服务器启动时
    -->
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <!--
        设置springMVC的核心控制器所能处理的请求的请求路径
        /所匹配的请求可以是/login或.html或.js或.css方式的请求路径
        但是/不能匹配.jsp请求路径的请求
    -->
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

> 注：

> <url-pattern>标签中使用/和/*的区别：
>
> /所匹配的请求可以是/login或.html或.js或.css方式的请求路径，但是/不能匹配.jsp请求路径的请求
>
> 因此就可以避免在访问jsp页面时，该请求被DispatcherServlet处理，从而找不到相应的页面
>
> /*则能够匹配所有请求，例如在使用过滤器时，若需要对所有请求进行过滤，就需要使用/*的写法

2.4、创建请求控制器

由于前端控制器对浏览器发送的请求进行了统一的处理，但是具体的请求有不同的处理过程，因此需要创建处理具体请求的类，即请求控制器

请求控制器中每一个处理请求的方法成为控制器方法

因为SpringMVC的控制器由一个POJO（普通的Java类）担任，因此需要通过@Controller注解将其标识为一个控制层组件，交给Spring的IoC容器管理，此时SpringMVC才能够识别控制器的存在

```
@Controller
public class HelloController {

}
```

2.5、创建SpringMVC的配置文件

```
<!-- 自动扫描包 -->
<context:component-scan base-package="com.atguigu.mvc.controller"/>

<!-- 配置Thymeleaf视图解析器 -->
<bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
    <property name="order" value="1"/>
    <property name="characterEncoding" value="UTF-8"/>
    <property name="templateEngine">
        <bean class="org.thymeleaf.spring5.SpringTemplateEngine">
            <property name="templateResolver">
                <bean
class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">

                    <!-- 视图前缀 -->
                    <property name="prefix" value="/WEB-INF/templates/"/>

                    <!-- 视图后缀 -->
                    <property name="suffix" value=".html"/>
                    <property name="templateMode" value="HTML5"/>
                    <property name="characterEncoding" value="UTF-8" />
                </bean>
            </property>
        </bean>
    </property>
</bean>

<!--
   处理静态资源，例如html、js、css、jpg
  若只设置该标签，则只能访问静态资源，其他请求则无法访问
```

```
  此时必须设置<mvc:annotation-driven/>解决问题
 -->
<mvc:default-servlet-handler/>

<!-- 开启mvc注解驱动 -->
<mvc:annotation-driven>
    <mvc:message-converters>
        <!-- 处理响应中文内容乱码 -->
        <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
            <property name="defaultCharset" value="UTF-8" />
            <property name="supportedMediaTypes">
                <list>
                    <value>text/html</value>
                    <value>application/json</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

2.6、测试HelloWorld

①实现对首页的访问

在请求控制器中创建处理请求的方法

```
// @RequestMapping注解：处理请求和控制器方法之间的映射关系
// @RequestMapping注解的value属性可以通过请求地址匹配请求，/表示的当前工程的上下文路径
// localhost:8080/springMVC/
@RequestMapping("/")
public String index() {
    //设置视图名称
    return "index";
}
```

②通过超链接跳转到指定页面

在主页index.html中设置超链接

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
    <h1>首页</h1>
    <a th:href="@{/hello}">HelloWorld</a><br/>
</body>
</html>
```

在请求控制器中创建处理请求的方法

```
@RequestMapping("/hello")
public String HelloWorld() {
    return "target";
}
```

2.7、总结

浏览器发送请求，若请求地址符合前端控制器的url-pattern，该请求就会被前端控制器DispatcherServlet处理。前端控制器会读取SpringMVC的核心配置文件，通过扫描组件找到控制器，将请求地址和控制器中@RequestMapping注解的value属性值进行匹配，若匹配成功，该注解所标识的控制器方法就是处理请求的方法。处理请求的方法需要返回一个字符串类型的视图名称，该视图名称会被视图解析器解析，加上前缀和后缀组成视图的路径，通过Thymeleaf对视图进行渲染，最终转发到视图所对应页面

3、@RequestMapping注解

3.1、@RequestMapping注解的功能

从注解名称上我们可以看到，@RequestMapping注解的作用就是将请求和处理请求的控制器方法关联起来，建立映射关系。

SpringMVC 接收到指定的请求，就会来找到在映射关系中对应的控制器方法来处理这个请求。

3.2、@RequestMapping注解的位置

@RequestMapping标识一个类：设置映射请求的请求路径的初始信息

@RequestMapping标识一个方法：设置映射请求请求路径的具体信息

```
@Controller
@RequestMapping("/test")
public class RequestMappingController {

    //此时请求映射所映射的请求的请求路径为：/test/testRequestMapping
    @RequestMapping("/testRequestMapping")
    public String testRequestMapping(){
        return "success";
    }

}
```

3.3、@RequestMapping注解的value属性

@RequestMapping注解的value属性通过请求的请求地址匹配请求映射

@RequestMapping注解的value属性是一个字符串类型的数组，表示该请求映射能够匹配多个请求地址所对应的请求

@RequestMapping注解的value属性必须设置，至少通过请求地址匹配请求映射

```
<a th:href="@{/testRequestMapping}">测试@RequestMapping的value属性--
>/testRequestMapping</a><br>
<a th:href="@{/test}">测试@RequestMapping的value属性-->/test</a><br>
```

```
@RequestMapping(
        value = {"/testRequestMapping", "/test"}
)
public String testRequestMapping(){
    return "success";
}
```

3.4、@RequestMapping注解的method属性

@RequestMapping注解的method属性通过请求的请求方式（get或post）匹配请求映射

@RequestMapping注解的method属性是一个RequestMethod类型的数组，表示该请求映射能够匹配多种请求方式的请求

若当前请求的请求地址满足请求映射的value属性，但是请求方式不满足method属性，则浏览器报错405：Request method 'POST' not supported

```
<a th:href="@{/test}">测试@RequestMapping的value属性-->/test</a><br>
<form th:action="@{/test}" method="post">
    <input type="submit">
</form>
```

```
@RequestMapping(
        value = {"/testRequestMapping", "/test"},
        method = {RequestMethod.GET, RequestMethod.POST}
)
public String testRequestMapping(){
    return "success";
}
```

> 注：
>
> 1、对于处理指定请求方式的控制器方法，SpringMVC中提供了@RequestMapping的派生注解
>
> 处理get请求的映射-->@GetMapping
>
> 处理post请求的映射-->@PostMapping
>
> 处理put请求的映射-->@PutMapping
>
> 处理delete请求的映射-->@DeleteMapping
>
> 2、常用的请求方式有get，post，put，delete
>
> 但是目前浏览器只支持get和post，若在form表单提交时，为method设置了其他请求方式的字符串（put或delete），则按照默认的请求方式get处理
>
> 若要发送put和delete请求，则需要通过spring提供的过滤器HiddenHttpMethodFilter，在RESTful部分会讲到

3.5、@RequestMapping注解的params属性（了解）

@RequestMapping注解的params属性通过请求的请求参数匹配请求映射

@RequestMapping注解的params属性是一个字符串类型的数组，可以通过四种表达式设置请求参数和请求映射的匹配关系

"param"：要求请求映射所匹配的请求必须携带param请求参数

"!param"：要求请求映射所匹配的请求必须不能携带param请求参数

"param=value"：要求请求映射所匹配的请求必须携带param请求参数且param=value

"param!=value"：要求请求映射所匹配的请求必须携带param请求参数但是param!=value

```
<a th:href="@{/test(username='admin',password=123456)">测试@RequestMapping的
params属性-->/test</a><br>
```

```
@RequestMapping(
        value = {"/testRequestMapping", "/test"}
        ,method = {RequestMethod.GET, RequestMethod.POST}
        ,params = {"username","password!=123456"}
)
public String testRequestMapping(){
    return "success";
}
```

> 注：
>
> 若当前请求满足@RequestMapping注解的value和method属性，但是不满足params属性，此时页面回报错400：Parameter conditions "username, password!=123456" not met for actual request parameters: username={admin}, password={123456}

3.6、@RequestMapping注解的headers属性（了解）

@RequestMapping注解的headers属性通过请求的请求头信息匹配请求映射

@RequestMapping注解的headers属性是一个字符串类型的数组，可以通过四种表达式设置请求头信息和请求映射的匹配关系

"header"：要求请求映射所匹配的请求必须携带header请求头信息

"!header"：要求请求映射所匹配的请求必须不能携带header请求头信息

"header=value"：要求请求映射所匹配的请求必须携带header请求头信息且header=value

"header!=value"：要求请求映射所匹配的请求必须携带header请求头信息且header!=value

若当前请求满足@RequestMapping注解的value和method属性，但是不满足headers属性，此时页面显示404错误，即资源未找到

3.7、SpringMVC支持ant风格的路径

？：表示任意的单个字符

*：表示任意的0个或多个字符

**：表示任意层数的任意目录

注意：在使用**时，只能使用/**/xxx的方式

3.8、SpringMVC支持路径中的占位符（重点）

原始方式：/deleteUser?id=1

rest方式：/user/delete/1

SpringMVC路径中的占位符常用于RESTful风格中，当请求路径中将某些数据通过路径的方式传输到服务器中，就可以在相应的@RequestMapping注解的value属性中通过占位符{xxx}表示传输的数据，在通过@PathVariable注解，将占位符所表示的数据赋值给控制器方法的形参

```
<a th:href="@{/testRest/1/admin}">测试路径中的占位符-->/testRest</a><br>
```

```
@RequestMapping("/testRest/{id}/{username}")
public String testRest(@PathVariable("id") String id, @PathVariable("username")
String username){
    System.out.println("id:"+id+",username:"+username);
    return "success";
}
//最终输出的内容为-->id:1,username:admin
```

4、SpringMVC获取请求参数

4.1、通过ServletAPI获取

将HttpServletRequest作为控制器方法的形参，此时HttpServletRequest类型的参数表示封装了当前请求的请求报文的对象

```
@RequestMapping("/testParam")
public String testParam(HttpServletRequest request){
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    System.out.println("username:"+username+",password:"+password);
    return "success";
}
```

4.2、通过控制器方法的形参获取请求参数

在控制器方法的形参位置，设置和请求参数同名的形参，当浏览器发送请求，匹配到请求映射时，在DispatcherServlet中就会将请求参数赋值给相应的形参

```
<a th:href="@{/testParam(username='admin',password=123456)}">测试获取请求参数--
>/testParam</a><br>
```

```
@RequestMapping("/testParam")
public String testParam(String username, String password){
    System.out.println("username:"+username+",password:"+password);
    return "success";
}
```

> 注：
>
> 若请求所传输的请求参数中有多个同名的请求参数，此时可以在控制器方法的形参中设置字符串数组或者字符串类型的形参接收此请求参数
>
> 若使用字符串数组类型的形参，此参数的数组中包含了每一个数据
>
> 若使用字符串类型的形参，此参数的值为每个数据中间使用逗号拼接的结果

4.3、@RequestParam

@RequestParam是将请求参数和控制器方法的形参创建映射关系

@RequestParam注解一共有三个属性：

value：指定为形参赋值的请求参数的参数名

required：设置是否必须传输此请求参数，默认值为true

若设置为true时，则当前请求必须传输value所指定的请求参数，若没有传输该请求参数，且没有设置defaultValue属性，则页面报错400：Required String parameter 'xxx' is not present；若设置为false，则当前请求不是必须传输value所指定的请求参数，若没有传输，则注解所标识的形参的值为null

defaultValue：不管required属性值为true或false，当value所指定的请求参数没有传输或传输的值为""时，则使用默认值为形参赋值

4.4、@RequestHeader

@RequestHeader是将请求头信息和控制器方法的形参创建映射关系

@RequestHeader注解一共有三个属性：value、required、defaultValue，用法同@RequestParam

4.5、@CookieValue

@CookieValue是将cookie数据和控制器方法的形参创建映射关系

@CookieValue注解一共有三个属性：value、required、defaultValue，用法同@RequestParam

4.6、通过POJO获取请求参数

可以在控制器方法的形参位置设置一个实体类类型的形参，此时若浏览器传输的请求参数的参数名和实体类中的属性名一致，那么请求参数就会为此属性赋值

```
<form th:action="@{/testpojo}" method="post">
    用户名：<input type="text" name="username"><br>
    密码：<input type="password" name="password"><br>
    性别：<input type="radio" name="sex" value="男">男<input type="radio"
name="sex" value="女">女<br>
    年龄：<input type="text" name="age"><br>
    邮箱：<input type="text" name="email"><br>
    <input type="submit">
</form>
```

```
@RequestMapping("/testpojo")
public String testPOJO(User user){
    System.out.println(user);
    return "success";
}
//最终结果-->User{id=null, username='张三', password='123', age=23, sex='男',
email='123@qq.com'}
```

4.7、解决获取请求参数的乱码问题

解决获取请求参数的乱码问题，可以使用SpringMVC提供的编码过滤器CharacterEncodingFilter，但是必须在web.xml中进行注册

```
<!--配置springMVC的编码过滤器-->
```

```
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

> 注：
>
> SpringMVC中处理编码的过滤器一定要配置到其他过滤器之前，否则无效

5、域对象共享数据

5.1、使用ServletAPI向request域对象共享数据

```
@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request){
    request.setAttribute("testScope", "hello,servletAPI");
    return "success";
}
```

5.2、使用ModelAndView向request域对象共享数据

```
@RequestMapping("/testModelAndView")
public ModelAndView testModelAndView(){
    /**
     * ModelAndView有Model和View的功能
     * Model主要用于向请求域共享数据
     * View主要用于设置视图，实现页面跳转
     */
    ModelAndView mav = new ModelAndView();
    //向请求域共享数据
    mav.addObject("testScope", "hello,ModelAndView");
    //设置视图，实现页面跳转
    mav.setViewName("success");
    return mav;
}
```

5.3、使用Model向request域对象共享数据

```
@RequestMapping("/testModel")
public String testModel(Model model){
    model.addAttribute("testScope", "hello,Model");
    return "success";
}
```

5.4、使用map向request域对象共享数据

```
@RequestMapping("/testMap")
public String testMap(Map<String, Object> map){
    map.put("testScope", "hello,Map");
    return "success";
}
```

5.5、使用ModelMap向request域对象共享数据

```
@RequestMapping("/testModelMap")
public String testModelMap(ModelMap modelMap){
    modelMap.addAttribute("testScope", "hello,ModelMap");
    return "success";
}
```

5.6、Model、ModelMap、Map的关系

Model、ModelMap、Map类型的参数其实本质上都是 BindingAwareModelMap 类型的

```
public interface Model{}
public class ModelMap extends LinkedHashMap<String, Object> {}
public class ExtendedModelMap extends ModelMap implements Model {}
public class BindingAwareModelMap extends ExtendedModelMap {}
```

5.7、向session域共享数据

```
@RequestMapping("/testSession")
public String testSession(HttpSession session){
    session.setAttribute("testSessionScope", "hello,session");
    return "success";
}
```

5.8、向application域共享数据

```
@RequestMapping("/testApplication")
public String testApplication(HttpSession session){
    ServletContext application = session.getServletContext();
    application.setAttribute("testApplicationScope", "hello,application");
    return "success";
}
```

6、SpringMVC的视图

SpringMVC中的视图是View接口，视图的作用渲染数据，将模型Model中的数据展示给用户

SpringMVC视图的种类很多，默认有转发视图和重定向视图

当工程引入jstl的依赖，转发视图会自动转换为JstlView

若使用的视图技术为Thymeleaf，在SpringMVC的配置文件中配置了Thymeleaf的视图解析器，由此视图解析器解析之后所得到的是ThymeleafView

6.1、ThymeleafView

当控制器方法中所设置的视图名称没有任何前缀时，此时的视图名称会被SpringMVC配置文件中所配置的视图解析器解析，视图名称拼接视图前缀和视图

后缀所得到的最终路径，会通过转发的方式实现跳转

```
@RequestMapping("/testHello")
public String testHello(){
    return "hello";
}
```

```
View view;  view: ThymeleafView@5342
String viewName = mv.getViewName();  viewName: "hello"
if (viewName != null) {
    // We need to resolve the view name.
    view = resolveViewName(viewName, mv.getModelInternal(), locale, request);  viewName: "hello"
    if (view == null) {  view: ThymeleafView@5342
        throw new ServletException("Could not resolve view with name '" + mv.getViewName() +
                "' in servlet with name '" + getServletName() + "'");
    }
}
```

+ {ThymeleafView@5342}

6.2、转发视图

SpringMVC中默认的转发视图是InternalResourceView

SpringMVC中创建转发视图的情况：

当控制器方法中所设置的视图名称以"forward:"为前缀时，创建InternalResourceView视图，此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀"forward:"去掉，剩余部分作为最终路径通过转发的方式实现跳转

例如"forward:/"，"forward:/employee"

```
@RequestMapping("/testForward")
public String testForward(){
    return "forward:/testHello";
}
```

```
1367        View view;  view: "org.springframework.web.servlet.view.InternalResourceView: [InternalResourceView];
1368        String viewName = mv.getViewName();  viewName: "forward:/testHello"
1369        if (viewName != null) {
1370            // We need to resolve the view name.
1371            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);  viewName: "forward:/te
                .view.InternalResourceView: [InternalR
                name '" + mv.getViewName() +
1374                    "' in servlet with name '" + getServletName() + "'");
1375            }
1376        }
```

+ {InternalResourceView@5557} "org.springframework.web.servlet.view.InternalResourceView: [InternalResourceView]; URL [/testHello]"

6.3、重定向视图

SpringMVC中默认的重定向视图是RedirectView

当控制器方法中所设置的视图名称以"redirect:"为前缀时，创建RedirectView视图，此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀"redirect:"去掉，剩余部分作为最终路径通过重定向的方式实现跳转

例如"redirect:/"，"redirect:/employee"

```
@RequestMapping("/testRedirect")
public String testRedirect(){
    return "redirect:/testHello";
}
```

> 注：
>
> 重定向视图在解析时，会先将redirect:前缀去掉，然后会判断剩余部分是否以/开头，若是则会自动拼接上下文路径

6.4、视图控制器view-controller

当控制器方法中，仅仅用来实现页面跳转，即只需要设置视图名称时，可以将处理器方法使用view-controller标签进行表示

```
<!--
    path：设置处理的请求地址
    view-name：设置请求地址所对应的视图名称
-->
<mvc:view-controller path="/testView" view-name="success"></mvc:view-controller>
```

> 注：
>
> 当SpringMVC中设置任何一个view-controller时，其他控制器中的请求映射将全部失效，此时需要在SpringMVC的核心配置文件中设置开启mvc注解驱动的标签：
>
> <mvc:annotation-driven />

7、RESTful

7.1、RESTful简介

REST：**Re**presentational **S**tate **T**ransfer，表现层资源状态转移。

①资源

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是名词。一个资源可以由一个或多个URI来标识。URI既是资源的名称，也是资源在Web上的地址。对某个资源感兴趣的客户端应用，可以通过资源的URI与其进行交互。

②资源的表述

资源的表述是一段对于资源在某个特定时刻的状态的描述。可以在客户端-服务器端之间转移（交换）。资源的表述可以有多种格式，例如HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

③状态转移

状态转移说的是：在客户端和服务器端之间转移（transfer）代表资源状态的表述。通过转移和操作资源的表述，来间接实现操作资源的目的。

7.2、RESTful的实现

具体说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。

它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。

REST 风格提倡 URL 地址使用统一的风格设计，从前到后各个单词使用斜杠分开，不使用问号键值对方式携带请求参数，而是将要发送给服务器的数据作为 URL 地址的一部分，以保证整体风格的一致性。

操作	传统方式	REST风格
查询操作	getUserById?id=1	user/1-->get请求方式
保存操作	saveUser	user-->post请求方式
删除操作	deleteUser?id=1	user/1-->delete请求方式
更新操作	updateUser	user-->put请求方式

7.3、HiddenHttpMethodFilter

由于浏览器只支持发送get和post方式的请求，那么该如何发送put和delete请求呢?

SpringMVC 提供了 **HiddenHttpMethodFilter** 帮助我们**将 POST 请求转换为 DELETE 或 PUT 请求**

HiddenHttpMethodFilter 处理put和delete请求的条件：

a>当前请求的请求方式必须为post

b>当前请求必须传输请求参数_method

满足以上条件，**HiddenHttpMethodFilter** 过滤器就会将当前请求的请求方式转换为请求参数_method的值，因此请求参数_method的值才是最终的请求方式

在web.xml中注册**HiddenHttpMethodFilter**

```
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

> 注：

> 目前为止，SpringMVC中提供了两个过滤器：CharacterEncodingFilter和HiddenHttpMethodFilter
>
> 在web.xml中注册时，必须先注册CharacterEncodingFilter，再注册HiddenHttpMethodFilter
>
> 原因：
>
> - 在 CharacterEncodingFilter 中通过 request.setCharacterEncoding(encoding) 方法设置字符集的
> - request.setCharacterEncoding(encoding) 方法要求前面不能有任何获取请求参数的操作
> - 而 HiddenHttpMethodFilter 恰恰有一个获取请求方式的操作：
> - `String paramValue = request.getParameter(this.methodParam);`

8、RESTful案例

8.1、准备工作

和传统 CRUD 一样，实现对员工信息的增删改查。

- 搭建环境
- 准备实体类

```
package com.atguigu.mvc.bean;

public class Employee {

    private Integer id;
    private String lastName;

    private String email;
    //1 male, 0 female
    private Integer gender;

    public Integer getId() {
        return id;
    }

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

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
```

```
    }

    public Integer getGender() {
        return gender;
    }

    public void setGender(Integer gender) {
        this.gender = gender;
    }

    public Employee(Integer id, String lastName, String email, Integer
gender) {
        super();
        this.id = id;
        this.lastName = lastName;
        this.email = email;
        this.gender = gender;
    }

    public Employee() {
    }
}
```

- 准备dao模拟数据

```
package com.atguigu.mvc.dao;

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

import com.atguigu.mvc.bean.Employee;
import org.springframework.stereotype.Repository;


@Repository
public class EmployeeDao {

    private static Map<Integer, Employee> employees = null;

    static{
        employees = new HashMap<Integer, Employee>();

        employees.put(1001, new Employee(1001, "E-AA", "aa@163.com", 1));
        employees.put(1002, new Employee(1002, "E-BB", "bb@163.com", 1));
        employees.put(1003, new Employee(1003, "E-CC", "cc@163.com", 0));
        employees.put(1004, new Employee(1004, "E-DD", "dd@163.com", 0));
        employees.put(1005, new Employee(1005, "E-EE", "ee@163.com", 1));
    }

    private static Integer initId = 1006;

    public void save(Employee employee){
        if(employee.getId() == null){
            employee.setId(initId++);
        }
        employees.put(employee.getId(), employee);
```

```
    }

    public Collection<Employee> getAll(){
        return employees.values();
    }

    public Employee get(Integer id){
        return employees.get(id);
    }

    public void delete(Integer id){
        employees.remove(id);
    }
}
```

8.2、功能清单

功能	URL 地址	请求方式
访问首页√	/	GET
查询全部数据√	/employee	GET
删除√	/employee/2	DELETE
跳转到添加数据页面√	/toAdd	GET
执行保存√	/employee	POST
跳转到更新数据页面√	/employee/2	GET
执行更新√	/employee	PUT

8.3、具体功能：访问首页

①配置view-controller

```
<mvc:view-controller path="/" view-name="index"/>
```

②创建页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" >
    <title>Title</title>
</head>
<body>
<h1>首页</h1>
<a th:href="@{/employee}">访问员工信息</a>
</body>
</html>
```

8.4、具体功能：查询所有员工数据

①控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.GET)
public String getEmployeeList(Model model){
    Collection<Employee> employeeList = employeeDao.getAll();
    model.addAttribute("employeeList", employeeList);
    return "employee_list";
}
```

②创建employee_list.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Employee Info</title>
    <script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
</head>
<body>

    <table border="1" cellpadding="0" cellspacing="0" style="text-align:
center;" id="dataTable">
        <tr>
            <th colspan="5">Employee Info</th>
        </tr>
        <tr>
            <th>id</th>
            <th>lastName</th>
            <th>email</th>
            <th>gender</th>
            <th>options(<a th:href="@{/toAdd}">add</a>)</th>
        </tr>
        <tr th:each="employee : ${employeeList}">
            <td th:text="${employee.id}"></td>
            <td th:text="${employee.lastName}"></td>
            <td th:text="${employee.email}"></td>
            <td th:text="${employee.gender}"></td>
            <td>
                <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
                <a th:href="@{'/employee/'+${employee.id}}">update</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

8.5、具体功能：删除

①创建处理delete请求方式的表单

```
<!-- 作用：通过超链接控制表单的提交，将post请求转换为delete请求 -->
<form id="delete_form" method="post">
    <!-- HiddenHttpMethodFilter要求：必须传输_method请求参数，并且值为最终的请求方式 -->
    <input type="hidden" name="_method" value="delete"/>
</form>
```

②删除超链接绑定点击事件

引入vue.js

```
<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
```

删除超链接

```
<a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
```

通过vue处理点击事件

```
<script type="text/javascript">
    var vue = new Vue({
        el:"#dataTable",
        methods:{
            //event表示当前事件
            deleteEmployee:function (event) {
                //通过id获取表单标签
                var delete_form = document.getElementById("delete_form");
                //将触发事件的超链接的href属性为表单的action属性赋值
                delete_form.action = event.target.href;
                //提交表单
                delete_form.submit();
                //阻止超链接的默认跳转行为
                event.preventDefault();
            }
        }
    });
</script>
```

③控制器方法

```
@RequestMapping(value = "/employee/{id}", method = RequestMethod.DELETE)
public String deleteEmployee(@PathVariable("id") Integer id){
    employeeDao.delete(id);
    return "redirect:/employee";
}
```

8.6、具体功能：跳转到添加数据页面

①配置view-controller

```
<mvc:view-controller path="/toAdd" view-name="employee_add"></mvc:view-
controller>
```

②创建employee_add.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Add Employee</title>
</head>
<body>
```

```
<form th:action="@{/employee}" method="post">
    lastName:<input type="text" name="lastName"><br>
    email:<input type="text" name="email"><br>
    gender:<input type="radio" name="gender" value="1">male
    <input type="radio" name="gender" value="0">female<br>
    <input type="submit" value="add"><br>
</form>

</body>
</html>
```

8.7、具体功能：执行保存

①控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.POST)
public String addEmployee(Employee employee){
    employeeDao.save(employee);
    return "redirect:/employee";
}
```

8.8、具体功能：跳转到更新数据页面

①修改超链接

```
<a th:href="@{'/employee/'+${employee.id}}">update</a>
```

②控制器方法

```
@RequestMapping(value = "/employee/{id}", method = RequestMethod.GET)
public String getEmployeeById(@PathVariable("id") Integer id, Model model){
    Employee employee = employeeDao.get(id);
    model.addAttribute("employee", employee);
    return "employee_update";
}
```

③创建employee_update.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Update Employee</title>
</head>
<body>

<form th:action="@{/employee}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="hidden" name="id" th:value="${employee.id}">
    lastName:<input type="text" name="lastName" th:value="${employee.lastName}">
<br>
    email:<input type="text" name="email" th:value="${employee.email}"><br>
    <!--
        th:field="${employee.gender}"可用于单选框或复选框的回显
```

```
        若单选框的value和employee.gender的值一致，则添加checked="checked"属性
    -->
    gender:<input type="radio" name="gender" value="1"
th:field="${employee.gender}">male
    <input type="radio" name="gender" value="0"
th:field="${employee.gender}">female<br>
    <input type="submit" value="update"><br>
</form>

</body>
</html>
```

8.9、具体功能：执行更新

①控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.PUT)
public String updateEmployee(Employee employee){
    employeeDao.save(employee);
    return "redirect:/employee";
}
```

9、SpringMVC处理ajax请求

9.1、@RequestBody

@RequestBody可以获取请求体信息，使用@RequestBody注解标识控制器方法的形参，当前请求的请求体就会为当前注解所标识的形参赋值

```
<!--此时必须使用post请求方式，因为get请求没有请求体-->
<form th:action="@{/test/RequestBody}" method="post">
    用户名：<input type="text" name="username"><br>
    密码：<input type="password" name="password"><br>
    <input type="submit">
</form>
```

```
@RequestMapping("/test/RequestBody")
public String testRequestBody(@RequestBody String requestBody){
    System.out.println("requestBody:"+requestBody);
    return "success";
}
```

输出结果：

requestBody:username=admin&password=123456

9.2、@RequestBody获取json格式的请求参数

> 在使用了axios发送ajax请求之后，浏览器发送到服务器的请求参数有两种格式：
>
> 1、name=value&name=value...，此时的请求参数可以通过request.getParameter()获取，对应SpringMVC中，可以直接通过控制器方法的形参获取此类请求参数

> 2、{key:value,key:value,...}，此时无法通过request.getParameter()获取，之前我们使用操作json的相关jar包gson或jackson处理此类请求参数，可以将其转换为指定的实体类对象或map集合。在SpringMVC中，直接使用@RequestBody注解标识控制器方法的形参即可将此类请求参数转换为java对象

使用@RequestBody获取json格式的请求参数的条件：

1、导入jackson的依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.1</version>
</dependency>
```

2、SpringMVC的配置文件中设置开启mvc的注解驱动

```
<!--开启mvc的注解驱动-->
<mvc:annotation-driven />
```

3、在控制器方法的形参位置，设置json格式的请求参数要转换成的java类型（实体类或map）的参数，并使用@RequestBody注解标识

```
<input type="button" value="测试@RequestBody获取json格式的请求参数"
@click="testRequestBody()"><br>

<script type="text/javascript" th:src="@{/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testRequestBody(){
                axios.post(
                    "/SpringMVC/test/RequestBody/json",
                    {username:"admin",password:"123456"}
                ).then(response=>{
                    console.log(response.data);
                });
            }
        }
    });
</script>
```

```
//将json格式的数据转换为map集合
@RequestMapping("/test/RequestBody/json")
public void testRequestBody(@RequestBody Map<String, Object> map,
HttpServletResponse response) throws IOException {
    System.out.println(map);
    //{username=admin, password=123456}
    response.getWriter().print("hello,axios");
}
//将json格式的数据转换为实体类对象
@RequestMapping("/test/RequestBody/json")
```

```
public void testRequestBody(@RequestBody User user, HttpServletResponse
response) throws IOException {
    System.out.println(user);
    //User{id=null, username='admin', password='123456', age=null,
gender='null'}
    response.getWriter().print("hello,axios");
}
```

9.3、@ResponseBody

@ResponseBody用于标识一个控制器方法，可以将该方法的返回值直接作为响应报文的响应体响应到浏览器

```
@RequestMapping("/testResponseBody")
public String testResponseBody(){
    //此时会跳转到逻辑视图success所对应的页面
    return "success";
}

@RequestMapping("/testResponseBody")
@ResponseBody
public String testResponseBody(){
    //此时响应浏览器数据success
    return "success";
}
```

9.4、@ResponseBody响应浏览器json数据

服务器处理ajax请求之后，大多数情况都需要向浏览器响应一个java对象，此时必须将java对象转换为json字符串才可以响应到浏览器，之前我们使用操作json数据的jar包gson或jackson将java对象转换为json字符串。在SpringMVC中，我们可以直接使用@ResponseBody注解实现此功能

@ResponseBody响应浏览器json数据的条件：

1、导入jackson的依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.1</version>
</dependency>
```

2、SpringMVC的配置文件中设置开启mvc的注解驱动

```
<!--开启mvc的注解驱动-->
<mvc:annotation-driven />
```

3、使用@ResponseBody注解标识控制器方法，在方法中，将需要转换为json字符串并响应到浏览器的java对象作为控制器方法的返回值，此时SpringMVC就可以将此对象直接转换为json字符串并响应到浏览器

```
<input type="button" value="测试@ResponseBody响应浏览器json格式的数据"
@click="testResponseBody()"><br>

<script type="text/javascript" th:src="@{/js/vue.js}"></script>
```

```
<script type="text/javascript" th:src="@{/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testResponseBody(){
                axios.post("/SpringMVC/test/ResponseBody/json").then(response=>{
                    console.log(response.data);
                });
            }
        }
    });
</script>
```

```
//响应浏览器list集合
@RequestMapping("/test/ResponseBody/json")
@ResponseBody
public List<User> testResponseBody(){
    User user1 = new User(1001,"admin1","123456",23,"男");
    User user2 = new User(1002,"admin2","123456",23,"男");
    User user3 = new User(1003,"admin3","123456",23,"男");
    List<User> list = Arrays.asList(user1, user2, user3);
    return list;
}

//响应浏览器map集合
@RequestMapping("/test/ResponseBody/json")
@ResponseBody
public Map<String, Object> testResponseBody(){
    User user1 = new User(1001,"admin1","123456",23,"男");
    User user2 = new User(1002,"admin2","123456",23,"男");
    User user3 = new User(1003,"admin3","123456",23,"男");
    Map<String, Object> map = new HashMap<>();
    map.put("1001", user1);
    map.put("1002", user2);
    map.put("1003", user3);
    return map;
}

//响应浏览器实体类对象
@RequestMapping("/test/ResponseBody/json")
@ResponseBody
public User testResponseBody(){
    return user;
}
```

9.5、@RestController注解

@RestController注解是springMVC提供的一个复合注解，标识在控制器的类上，就相当于为类添加了@Controller注解，并且为其中的每个方法添加了@ResponseBody注解

10、文件上传和下载

10.1、文件下载

ResponseEntity用于控制器方法的返回值类型，该控制器方法的返回值就是响应到浏览器的响应报文

使用ResponseEntity实现下载文件的功能

```
@RequestMapping("/testDown")
public ResponseEntity<byte[]> testResponseEntity(HttpSession session) throws
IOException {
    //获取ServletContext对象
    ServletContext servletContext = session.getServletContext();
    //获取服务器中文件的真实路径
    String realPath = servletContext.getRealPath("/static/img/1.jpg");
    //创建输入流
    InputStream is = new FileInputStream(realPath);
    //创建字节数组
    byte[] bytes = new byte[is.available()];
    //将流读到字节数组中
    is.read(bytes);
    //创建HttpHeaders对象设置响应头信息
    MultiValueMap<String, String> headers = new HttpHeaders();
    //设置要下载方式以及下载文件的名字
    headers.add("Content-Disposition", "attachment;filename=1.jpg");
    //设置响应状态码
    HttpStatus statusCode = HttpStatus.OK;
    //创建ResponseEntity对象
    ResponseEntity<byte[]> responseEntity = new ResponseEntity<>(bytes, headers,
statusCode);
    //关闭输入流
    is.close();
    return responseEntity;
}
```

10.2、文件上传

文件上传要求form表单的请求方式必须为post，并且添加属性enctype="multipart/form-data"

SpringMVC中将上传的文件封装到MultipartFile对象中，通过此对象可以获取文件相关信息

上传步骤：

①添加依赖：

```
<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload --
>
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```

②在SpringMVC的配置文件中添加配置：

```
<!--必须通过文件解析器的解析才能将文件转换为MultipartFile对象-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
</bean>
```

③控制器方法：

```
@RequestMapping("/testUp")
public String testUp(MultipartFile photo, HttpSession session) throws
IOException {
    //获取上传的文件的文件名
    String fileName = photo.getOriginalFilename();
    //处理文件重名问题
    String hzName = fileName.substring(fileName.lastIndexOf("."));
    fileName = UUID.randomUUID().toString() + hzName;
    //获取服务器中photo目录的路径
    ServletContext servletContext = session.getServletContext();
    String photoPath = servletContext.getRealPath("photo");
    File file = new File(photoPath);
    if(!file.exists()){
        file.mkdir();
    }
    String finalPath = photoPath + File.separator + fileName;
    //实现上传功能
    photo.transferTo(new File(finalPath));
    return "success";
}
```

11、拦截器

11.1、拦截器的配置

SpringMVC中的拦截器用于拦截控制器方法的执行

SpringMVC中的拦截器需要实现HandlerInterceptor

SpringMVC的拦截器必须在SpringMVC的配置文件中进行配置：

```
<bean class="com.atguigu.interceptor.FirstInterceptor"></bean>
<ref bean="firstInterceptor"></ref>
<!-- 以上两种配置方式都是对DispatcherServlet所处理的所有的请求进行拦截 -->
<mvc:interceptor>
    <mvc:mapping path="/**"/>
    <mvc:exclude-mapping path="/testRequestEntity"/>
    <ref bean="firstInterceptor"></ref>
</mvc:interceptor>
<!--
    以上配置方式可以通过ref或bean标签设置拦截器，通过mvc:mapping设置需要拦截的请求，通过
mvc:exclude-mapping设置需要排除的请求，即不需要拦截的请求
-->
```

11.2、拦截器的三个抽象方法

SpringMVC中的拦截器有三个抽象方法：

preHandle：控制器方法执行之前执行preHandle()，其boolean类型的返回值表示是否拦截或放行，返回true为放行，即调用控制器方法；返回false表示拦截，即不调用控制器方法

postHandle：控制器方法执行之后执行postHandle()

afterCompletion：处理完视图和模型数据，渲染视图完毕之后执行afterCompletion()

11.3、多个拦截器的执行顺序

①若每个拦截器的preHandle()都返回true

此时多个拦截器的执行顺序和拦截器在SpringMVC的配置文件的配置顺序有关：

preHandle()会按照配置的顺序执行，而postHandle()和afterCompletion()会按照配置的反序执行

②若某个拦截器的preHandle()返回了false

preHandle()返回false和它之前的拦截器的preHandle()都会执行，postHandle()都不执行，返回false的拦截器之前的拦截器的afterCompletion()会执行

12、异常处理器

12.1、基于配置的异常处理

SpringMVC提供了一个处理控制器方法执行过程中所出现的异常的接口：HandlerExceptionResolver

HandlerExceptionResolver接口的实现类有：DefaultHandlerExceptionResolver和SimpleMappingExceptionResolver

SpringMVC提供了自定义的异常处理器SimpleMappingExceptionResolver，使用方式：

```
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <!--
                properties的键表示处理器方法执行过程中出现的异常
                properties的值表示若出现指定异常时，设置一个新的视图名称，跳转到指定页面
            -->
            <prop key="java.lang.ArithmeticException">error</prop>
        </props>
    </property>
    <!--
        exceptionAttribute属性设置一个属性名，将出现的异常信息在请求域中进行共享
    -->
    <property name="exceptionAttribute" value="ex"></property>
</bean>
```

12.2、基于注解的异常处理

```
//@ControllerAdvice将当前类标识为异常处理的组件
@ControllerAdvice
public class ExceptionController {

    //@ExceptionHandler用于设置所标识方法处理的异常
    @ExceptionHandler(ArithmeticException.class)
    //ex表示当前请求处理中出现的异常对象
    public String handleArithmeticException(Exception ex, Model model){
        model.addAttribute("ex", ex);
        return "error";
    }

}
```

13、注解配置SpringMVC

使用配置类和注解代替web.xml和SpringMVC配置文件的功能

13.1、创建初始化类，代替web.xml

在Servlet3.0环境中，容器会在类路径中查找实现javax.servlet.ServletContainerInitializer接口的类，如果找到的话就用它来配置Servlet容器。 Spring提供了这个接口的实现，名为SpringServletContainerInitializer，这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成。Spring3.2引入了一个便利的WebApplicationInitializer基础实现，名为AbstractAnnotationConfigDispatcherServletInitializer，当我们的类扩展了AbstractAnnotationConfigDispatcherServletInitializer并将其部署到Servlet3.0容器的时候，容器会自动发现它，并用它来配置Servlet上下文。

```
public class WebInit extends
AbstractAnnotationConfigDispatcherServletInitializer {

    /**
     * 指定spring的配置类
     * @return
     */
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }

    /**
     * 指定SpringMVC的配置类
     * @return
     */
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{WebConfig.class};
    }

    /**
     * 指定DispatcherServlet的映射规则，即url-pattern
     * @return
     */
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    /**
     * 添加过滤器
     * @return
     */
    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();
        encodingFilter.setEncoding("UTF-8");
        encodingFilter.setForceRequestEncoding(true);
        HiddenHttpMethodFilter hiddenHttpMethodFilter = new
HiddenHttpMethodFilter();
        return new Filter[]{encodingFilter, hiddenHttpMethodFilter};
    }
}
```

13.2、创建SpringConfig配置类，代替spring的配置文件

```
@Configuration
public class SpringConfig {
    //ssm整合之后，spring的配置信息写在此类中
}
```

13.3、创建WebConfig配置类，代替SpringMVC的配置文件

```
@Configuration
//扫描组件
@ComponentScan("com.atguigu.mvc.controller")
//开启MVC注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    //使用默认的servlet处理静态资源
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

    //配置文件上传解析器
    @Bean
    public CommonsMultipartResolver multipartResolver(){
        return new CommonsMultipartResolver();
    }

    //配置拦截器
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        FirstInterceptor firstInterceptor = new FirstInterceptor();
        registry.addInterceptor(firstInterceptor).addPathPatterns("/**");
    }

    //配置视图控制

    /*@Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }*/

    //配置异常映射
    /*@Override
    public void
configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        SimpleMappingExceptionResolver exceptionResolver = new
SimpleMappingExceptionResolver();
        Properties prop = new Properties();
        prop.setProperty("java.lang.ArithmeticException", "error");
        //设置异常映射
        exceptionResolver.setExceptionMappings(prop);
        //设置共享异常信息的键
        exceptionResolver.setExceptionAttribute("ex");
        resolvers.add(exceptionResolver);
```

```
    }*/

    //配置生成模板解析器
    @Bean
    public ITemplateResolver templateResolver() {
        WebApplicationContext webApplicationContext =
ContextLoader.getCurrentWebApplicationContext();
        // ServletContextTemplateResolver需要一个ServletContext作为构造参数，可通过
WebApplicationContext 的方法获得
        ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver(
                webApplicationContext.getServletContext());
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setCharacterEncoding("UTF-8");
        templateResolver.setTemplateMode(TemplateMode.HTML);
        return templateResolver;
    }

    //生成模板引擎并为模板引擎注入模板解析器
    @Bean
    public SpringTemplateEngine templateEngine(ITemplateResolver
templateResolver) {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);
        return templateEngine;
    }

    //生成视图解析器并未解析器注入模板引擎
    @Bean
    public ViewResolver viewResolver(SpringTemplateEngine templateEngine) {
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setCharacterEncoding("UTF-8");
        viewResolver.setTemplateEngine(templateEngine);
        return viewResolver;
    }


}
```

13.4、测试功能

```
@RequestMapping("/")
public String index(){
    return "index";
}
```

14、SpringMVC执行流程

14.1、SpringMVC常用组件

- DispatcherServlet：**前端控制器**，不需要工程师开发，由框架提供

作用：统一处理请求和响应，整个流程控制的中心，由它调用其它组件处理用户的请求

- HandlerMapping：**处理器映射器**，不需要工程师开发，由框架提供

作用：根据请求的url、method等信息查找Handler，即控制器方法

- Handler：**处理器**，需要工程师开发

作用：在DispatcherServlet的控制下Handler对具体的用户请求进行处理

- HandlerAdapter：**处理器适配器**，不需要工程师开发，由框架提供

作用：通过HandlerAdapter对处理器（控制器方法）进行执行

- ViewResolver：**视图解析器**，不需要工程师开发，由框架提供

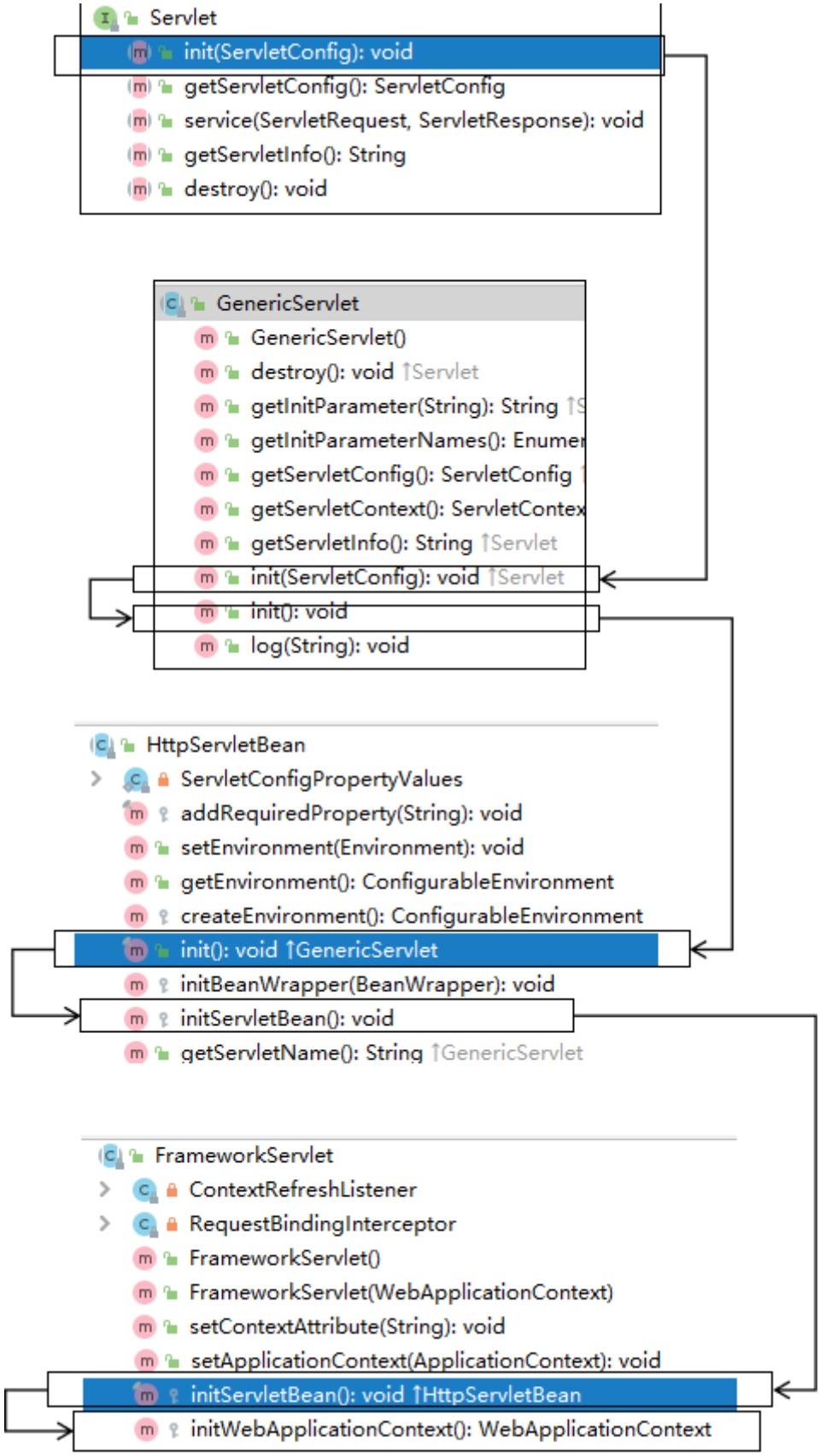
作用：进行视图解析，得到相应的视图，例如：ThymeleafView、InternalResourceView、RedirectView

- View：**视图**

作用：将模型数据通过页面展示给用户

14.2、DispatcherServlet初始化过程

DispatcherServlet 本质上是一个 Servlet，所以天然的遵循 Servlet 的生命周期。所以宏观上是 Servlet 生命周期来进行调度。



①初始化WebApplicationContext

所在类：org.springframework.web.servlet.FrameworkServlet

```
protected WebApplicationContext initWebApplicationContext() {
    WebApplicationContext rootContext =

 WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;
```

```
    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        wac = this.webApplicationContext;
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac =
(ConfigurableWebApplicationContext) wac;
            if (!cwac.isActive()) {
                // The context has not yet been refreshed -> provide services
such as
                // setting the parent context, setting the application context
id, etc
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit
parent -> set
                    // the root application context (if any; may be null) as the
parent
                    cwac.setParent(rootContext);
                }
                configureAndRefreshWebApplicationContext(cwac);
            }
        }
    }
    if (wac == null) {
        // No context instance was injected at construction time -> see if one
        // has been registered in the servlet context. If one exists, it is
assumed
        // that the parent context (if any) has already been set and that the
        // user has performed any initialization such as setting the context id
        wac = findWebApplicationContext();
    }
    if (wac == null) {
        // No context instance is defined for this servlet -> create a local one
        // 创建WebApplicationContext
        wac = createWebApplicationContext(rootContext);
    }

    if (!this.refreshEventReceived) {
        // Either the context is not a ConfigurableApplicationContext with
refresh
        // support or the context injected at construction time had already been
        // refreshed -> trigger initial onRefresh manually here.
        synchronized (this.onRefreshMonitor) {
            // 刷新WebApplicationContext
            onRefresh(wac);
        }
    }

    if (this.publishContext) {
        // Publish the context as a servlet context attribute.
        // 将IOC容器在应用域共享
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
    }

    return wac;
}
```

②创建WebApplicationContext

所在类：org.springframework.web.servlet.FrameworkServlet

```
protected WebApplicationContext createWebApplicationContext(@Nullable
ApplicationContext parent) {
    Class<?> contextClass = getContextClass();
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass))
{
        throw new ApplicationContextException(
            "Fatal initialization error in servlet with name '" +
getServletName() +
            "': custom WebApplicationContext class [" + contextClass.getName() +
            "] is not of type ConfigurableWebApplicationContext");
    }
    // 通过反射创建 IOC 容器对象
    ConfigurableWebApplicationContext wac =
        (ConfigurableWebApplicationContext)
BeanUtils.instantiateClass(contextClass);

    wac.setEnvironment(getEnvironment());
    // 设置父容器
    wac.setParent(parent);
    String configLocation = getContextConfigLocation();
    if (configLocation != null) {
        wac.setConfigLocation(configLocation);
    }
    configureAndRefreshWebApplicationContext(wac);

    return wac;
}
```

③DispatcherServlet初始化策略

FrameworkServlet创建WebApplicationContext后，刷新容器，调用onRefresh(wac)，此方法在DispatcherServlet中进行了重写，调用了initStrategies(context)方法，初始化策略，即初始化DispatcherServlet的各个组件

所在类：org.springframework.web.servlet.DispatcherServlet

```
protected void initStrategies(ApplicationContext context) {
   initMultipartResolver(context);
   initLocaleResolver(context);
   initThemeResolver(context);
   initHandlerMappings(context);
   initHandlerAdapters(context);
   initHandlerExceptionResolvers(context);
   initRequestToViewNameTranslator(context);
   initViewResolvers(context);
   initFlashMapManager(context);
}
```

14.3、DispatcherServlet调用组件处理请求

①processRequest()

FrameworkServlet重写HttpServlet中的service()和doXxx()，这些方法中调用了processRequest(request, response)

所在类：org.springframework.web.servlet.FrameworkServlet

```
protected final void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {

    long startTime = System.currentTimeMillis();
    Throwable failureCause = null;

    LocaleContext previousLocaleContext =
LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

    RequestAttributes previousAttributes =
RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request,
response, previousAttributes);

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(),
new RequestBindingInterceptor());

    initContextHolders(request, localeContext, requestAttributes);

    try {
        // 执行服务，doService()是一个抽象方法，在DispatcherServlet中进行了重写
        doService(request, response);
    }
    catch (ServletException | IOException ex) {
        failureCause = ex;
        throw ex;
    }
    catch (Throwable ex) {
        failureCause = ex;
        throw new NestedServletException("Request processing failed", ex);
    }

    finally {
        resetContextHolders(request, previousLocaleContext, previousAttributes);
        if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }
        logResult(request, response, failureCause, asyncManager);
        publishRequestHandledEvent(request, response, startTime, failureCause);
    }
}
```

②doService()

所在类：org.springframework.web.servlet.DispatcherServlet

```
@Override
protected void doService(HttpServletRequest request, HttpServletResponse
response) throws Exception {
```

```
        logRequest(request);

        // Keep a snapshot of the request attributes in case of an include,
        // to be able to restore the original attributes after the include.
        Map<String, Object> attributesSnapshot = null;
        if (WebUtils.isIncludeRequest(request)) {
            attributesSnapshot = new HashMap<>();
            Enumeration<?> attrNames = request.getAttributeNames();
            while (attrNames.hasMoreElements()) {
                String attrName = (String) attrNames.nextElement();
                if (this.cleanupAfterInclude ||
attrName.startsWith(DEFAULT_STRATEGIES_PREFIX)) {
                    attributesSnapshot.put(attrName,
request.getAttribute(attrName));
                }
            }
        }

        // Make framework objects available to handlers and view objects.
        request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
getWebApplicationContext());
        request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
        request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
        request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

        if (this.flashMapManager != null) {
            FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request,
response);
            if (inputFlashMap != null) {
                request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE,
Collections.unmodifiableMap(inputFlashMap));
            }
            request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
            request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE, this.flashMapManager);
        }

        RequestPath requestPath = null;
        if (this.parseRequestPath &&
!ServletRequestPathUtils.hasParsedRequestPath(request)) {
            requestPath = ServletRequestPathUtils.parseAndCache(request);
        }

        try {
            // 处理请求和响应
            doDispatch(request, response);
        }
        finally {
            if
(!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
                // Restore the original attribute snapshot, in case of an include.
                if (attributesSnapshot != null) {
                    restoreAttributesAfterInclude(request, attributesSnapshot);
                }
            }
            if (requestPath != null) {
                ServletRequestPathUtils.clearParsedRequestPath(request);
            }
        }
```

```
}
```

③doDispatch()

所在类：org.springframework.web.servlet.DispatcherServlet

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // Determine handler for the current request.
            /*
                mappedHandler：调用链
                包含handler、interceptorList、interceptorIndex
                handler：浏览器发送的请求所匹配的控制器方法
                interceptorList：处理控制器方法的所有拦截器集合
                interceptorIndex：拦截器索引，控制拦截器afterCompletion()的执行
            */
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // Determine handler adapter for the current request.
            // 通过控制器方法创建相应的处理器适配器，调用所对应的控制器方法
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = "GET".equals(method);
            if (isGet || "HEAD".equals(method)) {
                long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                if (new ServletWebRequest(request,
response).checkNotModified(lastModified) && isGet) {
                    return;
                }
            }

            // 调用拦截器的preHandle()
            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }
```

```
            // Actually invoke the handler.
            // 由处理器适配器调用具体的控制器方法，最终获得ModelAndView对象
            mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }

            applyDefaultViewName(processedRequest, mv);
            // 调用拦截器的postHandle()
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            dispatchException = ex;
        }
        catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler methods as
well,
            // making them available for @ExceptionHandler methods and other
scenarios.
            dispatchException = new NestedServletException("Handler dispatch
failed", err);
        }
        // 后续处理：处理模型数据和渲染视图
        processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
    }
    catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
                               new NestedServletException("Handler processing
failed", err));
    }
    finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            // Instead of postHandle and afterCompletion
            if (mappedHandler != null) {

 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        }
        else {
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
}
```

④processDispatchResult()

```
private void processDispatchResult(HttpServletRequest request,
HttpServletResponse response,
```

```
                                    @Nullable HandlerExecutionChain
mappedHandler, @Nullable ModelAndView mv,
                                    @Nullable Exception exception) throws
Exception {

    boolean errorView = false;

    if (exception != null) {
        if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered",
exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler()
: null);
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }

    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        // 处理模型数据和渲染视图
        render(mv, request, response);
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isTraceEnabled()) {
            logger.trace("No view rendering, null ModelAndView returned.");
        }
    }

    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }

    if (mappedHandler != null) {
        // Exception (if any) is already handled..
        // 调用拦截器的afterCompletion()
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
```

14.4、SpringMVC的执行流程

1) 用户向服务器发送请求，请求被SpringMVC 前端控制器 DispatcherServlet捕获。

2) DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI），判断请求URI对应的映射：

a) 不存在

i. 再判断是否配置了mvc:default-servlet-handler

ii. 如果没配置，则控制台报映射查找不到，客户端展示404错误

```
DEBUG org.springframework.web.servlet.DispatcherServlet - GET /springMVC/testHaha , parameters={}
WARN org.springframework.web.servlet.PageNotFound - No mapping for GET /springMVC/testHaha
DEBUG org.springframework.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND
```

HTTP Status 404 -

type Status report

message

description The requested resource is not available.

Apache Tomcat/7.0.79

iii. 如果有配置，则访问目标资源（一般为静态资源，如：JS,CSS,HTML），找不到客户端也会展示404错误

```
DispatcherServlet - GET "/springMVC/testHaha", parameters={}
handler.SimpleUrlHandlerMapping - Mapped to org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler
DispatcherServlet - Completed 404 NOT_FOUND
```

HTTP Status 404 - /springMVC/testHaha

type Status report

message /springMVC/testHaha

description The requested resource is not available.

Apache Tomcat/7.0.79

b) 存在则执行下面的流程

3) 根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain执行链对象的形式返回。

4) DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。

5) 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(...)方法【正向】

6) 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)方法，处理请求。在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

a) HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

b) 数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

c) 数据格式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

d) 数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

7) Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象。

8) 此时将开始执行拦截器的postHandle(...)方法【逆向】。

9) 根据返回的ModelAndView（此时会判断是否存在异常：如果存在异常，则执行HandlerExceptionResolver进行异常处理）选择一个适合的ViewResolver进行视图解析，根据Model和View，来渲染视图。

10) 渲染视图完毕执行拦截器的afterCompletion(...)方法【逆向】。

11) 将渲染结果返回给客户端。

四、SSM整合

4.1、ContextLoaderListener

Spring提供了监听器ContextLoaderListener，实现ServletContextListener接口，可监听ServletContext的状态，在web服务器的启动，读取Spring的配置文件，创建Spring的IOC容器。web应用中必须在web.xml中配置

```
<listener>
    <!--
        配置Spring的监听器，在服务器启动时加载Spring的配置文件
        Spring配置文件默认位置和名称：/WEB-INF/applicationContext.xml
        可通过上下文参数自定义Spring配置文件的位置和名称
    -->
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!--自定义Spring配置文件的位置和名称-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring.xml</param-value>
</context-param>
```

4.2、准备工作

①创建Maven Module

②导入依赖

```
<packaging>war</packaging>

<properties>
    <spring.version>5.3.1</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!--springmvc-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- Mybatis核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>
    <!--mybatis和spring的整合包-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.6</version>
    </dependency>
    <!-- 连接池 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.0.9</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- MySQL驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
    <!-- log4j日志 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/com.github.pagehelper/pagehelper -->
```

```
    <dependency>
        <groupId>com.github.pagehelper</groupId>
        <artifactId>pagehelper</artifactId>
        <version>5.2.0</version>
    </dependency>
    <!-- 日志 -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
    <!-- ServletAPI -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.1</version>
    </dependency>
    <dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
    </dependency>
    <!-- Spring5和Thymeleaf整合包 -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.0.12.RELEASE</version>
    </dependency>
</dependencies>
```

③创建表

```
CREATE TABLE `t_emp` (
  `emp_id` int(11) NOT NULL AUTO_INCREMENT,
  `emp_name` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `sex` char(1) DEFAULT NULL,
  `email` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`emp_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

4.3、配置web.xml

```
<!-- 配置Spring的编码过滤器 -->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
```

```
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- 配置处理请求方式PUT和DELETE的过滤器 -->
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- 配置SpringMVC的前端控制器 -->
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!-- 设置SpringMVC的配置文件的位置和名称 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:SpringMVC.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- 设置Spring的配置文件的位置和名称 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:Spring.xml</param-value>
</context-param>

<!-- 配置Spring的监听器 -->
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

4.4、创建SpringMVC的配置文件并配置

```
<!--扫描组件-->
```

```
<context:component-scan base-package="com.atguigu.ssm.controller">
</context:component-scan>

<!--配置视图解析器-->
<bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
    <property name="order" value="1"/>
    <property name="characterEncoding" value="UTF-8"/>
    <property name="templateEngine">
        <bean class="org.thymeleaf.spring5.SpringTemplateEngine">
            <property name="templateResolver">
                <bean
class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">
                    <!-- 视图前缀 -->
                    <property name="prefix" value="/WEB-INF/templates/"/>
                    <!-- 视图后缀 -->
                    <property name="suffix" value=".html"/>
                    <property name="templateMode" value="HTML5"/>
                    <property name="characterEncoding" value="UTF-8" />
                </bean>
            </property>
        </bean>
    </property>
</bean>

<!-- 配置访问首页的视图控制 -->
<mvc:view-controller path="/" view-name="index"></mvc:view-controller>

<!-- 配置默认的servlet处理静态资源 -->
<mvc:default-servlet-handler />

<!-- 开启MVC的注解驱动 -->
<mvc:annotation-driven />
```

4.5、搭建MyBatis环境

①创建属性文件jdbc.properties

```
jdbc.user=root
jdbc.password=atguigu
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.driver=com.mysql.cj.jdbc.Driver
```

②创建MyBatis的核心配置文件mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

    <settings>
        <!--将下划线映射为驼峰-->
        <setting name="mapUnderscoreToCamelCase" value="true"/>
```

```
    </settings>

    <plugins>
        <!--配置分页插件-->
        <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
    </plugins>

</configuration>
```

③创建Mapper接口和映射文件

```
public interface EmployeeMapper {
    List<Employee> getEmployeeList();
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.atguigu.ssm.mapper.EmployeeMapper">

    <select id="getEmployeeList" resultType="Employee">
        select * from t_emp
    </select>

</mapper>
```

④创建日志文件log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd HH:mm:ss,SSS}
%m  (%F:%L) \n" />
        </layout>
    </appender>
    <logger name="java.sql">
        <level value="debug" />
    </logger>
    <logger name="org.apache.ibatis">
        <level value="info" />
    </logger>
    <root>
        <level value="debug" />
        <appender-ref ref="STDOUT" />
    </root>
</log4j:configuration>
```

4.6、创建Spring的配置文件并配置

```
<?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/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <!--扫描组件-->
    <context:component-scan base-package="com.atguigu.ssm">
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 引入jdbc.properties -->
    <context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>

    <!-- 配置Druid数据源 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>

    <!-- 配置用于创建SqlSessionFactory的工厂bean -->
    <bean class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 设置MyBatis配置文件的路径（可以不设置） -->
        <property name="configLocation" value="classpath:mybatis-config.xml">
</property>
        <!-- 设置数据源 -->
        <property name="dataSource" ref="dataSource"></property>
        <!-- 设置类型别名所对应的包 -->
        <property name="typeAliasesPackage" value="com.atguigu.ssm.pojo">
</property>
        <!--
            设置映射文件的路径
            若映射文件所在路径和mapper接口所在路径一致，则不需要设置
        -->
        <!--
            <property name="mapperLocations" value="classpath:mapper/*.xml">
</property>
        -->
    </bean>

    <!--
        配置mapper接口的扫描配置
        由mybatis-spring提供，可以将指定包下所有的mapper接口创建动态代理
        并将这些动态代理作为IOC容器的bean管理
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.atguigu.ssm.mapper"></property>
    </bean>

</beans>
```

4.7、测试功能

①创建组件

实体类Employee

```
public class Employee {

    private Integer empId;

    private String empName;

    private Integer age;

    private String sex;

    private String email;

    public Employee() {
    }

    public Employee(Integer empId, String empName, Integer age, String sex,
String email) {
        this.empId = empId;
        this.empName = empName;
        this.age = age;
        this.sex = sex;
        this.email = email;
    }

    public Integer getEmpId() {
        return empId;
    }

    public void setEmpId(Integer empId) {
        this.empId = empId;
    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public Integer getAge() {
        return age;
    }

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

    public String getSex() {
        return sex;
```

```
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

创建控制层组件EmployeeController

```
@Controller
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @RequestMapping(value = "/employee/page/{pageNum}", method =
RequestMethod.GET)
    public String getEmployeeList(Model model, @PathVariable("pageNum") Integer
pageNum){
        PageInfo<Employee> page = employeeService.getEmployeeList(pageNum);
        model.addAttribute("page", page);
        return "employee_list";
    }

}
```

创建接口EmployeeService

```
public interface EmployeeService {
    PageInfo<Employee> getEmployeeList(Integer pageNum);
}
```

创建实现类EmployeeServiceImpl

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeMapper employeeMapper;

    @Override
    public PageInfo<Employee> getEmployeeList(Integer pageNum) {
        PageHelper.startPage(pageNum, 4);
        List<Employee> list = employeeMapper.getEmployeeList();
        PageInfo<Employee> page = new PageInfo<>(list, 5);
        return page;
    }
}
```

②创建页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Employee Info</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}">
</head>
<body>

    <table>
        <tr>
            <th colspan="6">Employee Info</th>
        </tr>
        <tr>
            <th>emp_id</th>
            <th>emp_name</th>
            <th>age</th>
            <th>sex</th>
            <th>email</th>
            <th>options</th>
        </tr>
        <tr th:each="employee : ${page.list}">
            <td th:text="${employee.empId}"></td>
            <td th:text="${employee.empName}"></td>
            <td th:text="${employee.age}"></td>
            <td th:text="${employee.sex}"></td>
            <td th:text="${employee.email}"></td>
            <td>
                <a href="">delete</a>
                <a href="">update</a>
            </td>
        </tr>
        <tr>
            <td colspan="6">
                <span th:if="${page.hasPreviousPage}">
                    <a th:href="@{/employee/page/1}">首页</a>
                    <a th:href="@{'/employee/page/'+${page.prePage}}">上一页</a>
                </span>

                <span th:each="num : ${page.navigatepageNums}">
                    <a th:if="${page.pageNum==num}"
th:href="@{'/employee/page/'+${num}}" th:text="'['+${num}+']'" style="color:
red;"></a>
                    <a th:if="${page.pageNum!=num}"
th:href="@{'/employee/page/'+${num}}" th:text="${num} "></a>
                </span>

                <span th:if="${page.hasNextPage}">
                    <a th:href="@{'/employee/page/'+${page.nextPage}}">下一页</a>
                    <a th:href="@{'/employee/page/'+${page.pages}}">末页</a>
                </span>
            </td>
        </tr>
    </table>
```

```
</body>
</html>
```

③访问测试分页功能

localhost:8080/employee/page/1