01

MyBatis

封装JDBC，负责访问数据库，完成持久化操作。

02

Spring

使用其核心思想IOC管理组件，使用AOP思想实现功能增强

03

SpringMVC

接收浏览器发送的请求并响应浏览器数据

04

SSM整合

结合案例整合SSM，进一步了解各个框架的功能

目　录

MyBatis

一 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 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。

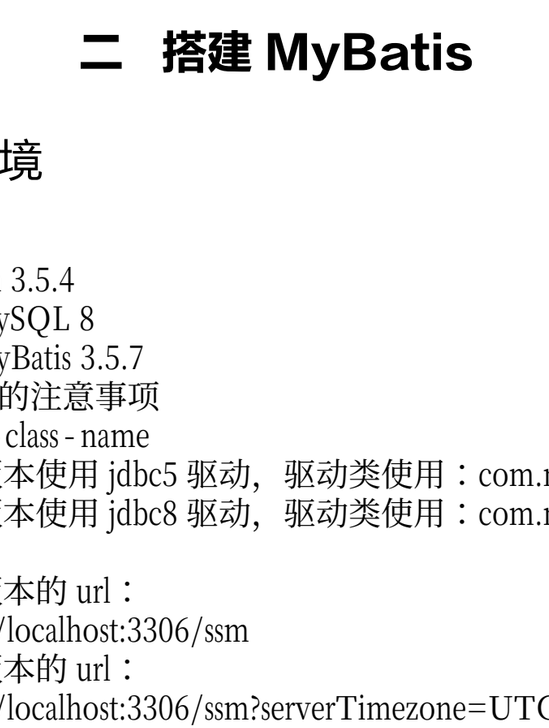
· MyBatis
  · 轻量级，性能出色；
  · SQL 和 Java 编码分开，功能边界清晰。Java 代码专注业务、SQL 语句专注数据；
  · 开发效率稍逊于 HIbernate，但是完全能够接受。

二 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 orrepresents more
```



2.2 创建 maven 工程

①打包方式：jar
②引入依赖

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

    <groupId>MyBatis</groupId>
    <artifactId>mybatis-helloworld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</
groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>16</source>
                    <target>16</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <packaging>jar</packaging>

    <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>

<!-- log4j 日志 -->
<dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>RELEASE</version>
        <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```


2.3 User 基本信息

```
package mybatis1.pojo;

public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
    private String email;

    public User() {
    }

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

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;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

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


2.4 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>
    <!--
    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，此后就可以在当前文件中使用 ${key} 的方式访问 value -->
    <properties resource="jdbc.properties" />
    <!--
        typeAliases：设置类型别名，即为某个具体的类型设置一个别名
        在 MyBatis 的范围中，就可以使用别名表示一个具体的类型
    -->
    <typeAliases>
        <!--
            type：设置需要设置别名的类型
            alias：设置某个类型的别名
        -->
        <!--<typeAlias type="com.atguigu.mybatis.pojo.User"></typeAlias>-->
        <!-- 若不设置 alias，当前的类型拥有默认的别名，即类名且不区分大小写 -->
        <!--<typeAlias type="mybatis.pojo.User" alias="abc"></typeAlias>-->
        <!-- 通过包设置类型别名，指定包下所有的类型将全部拥有默认的别名，即类名且不区分大小写 -->
        <package name="mybatis.pojo" />
    </typeAliases>
```

```
<!--
      environments：配置连接数据库的环境
      属性：default：设置默认使用的环境 id
-->
<environments default="development">
      <!--
            environment：设置一个具体的连接数据库环境
            属性：id：设置环境的唯一标识，不能重复
      -->
      <environment id="development">
            <!--
            transactionManager: 设置事务管理器
            属性：
                  type：设置事务管理的方式
                  type="JDBC|MANAGED"
                  JDBC：表示使用 JDBC 中原生的事务管理方式
                  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">
                        <property name="driver" value="${jdbc.drive}"
/>
                        <property name="url" value="${jdbc.url}" />
                        <property name="username" value="${jdbc.
username}" />
                        <property name="password" value="${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/ssm?serverTimezone=UTC" />
                        <property name="username" value="root" />
                        <property name="password" value="abc123" />
                  </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="mybatis.mapper" />
      </mappers>
</configuration>
```

2.5 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();

    // 修改用户信息
    void updateUser();

    // 删除用户信息
    void deleteUser();

    // 根据 ID 查询用户信息
    User getUserById();

    // 查询所有的用户信息
    List<User> getAllUser();
}
```


2.6 创建 MyBatis 的映射文件

相关概念：ORM（Object Relationship Mapping）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="java.mybatis.mapper.UserMapper">

    <!--
    mapper 接口和映射文件要保证两个一致:
        1.mapper 接口的全类名和映射文件的 namespace 一致
        2.mapper 接口中的方法的方法名要和映射文件中的 sql 的 id 保持一到
    -->
    <!-- int insertUser(); -->
    <insert id="insertUser">
        insert into t_user
        values (null, 'admin', '123456', 23, ' 男 ', '123456@qq.com')
    </insert>
```

```
<!-- void updateUser(); -->
<update id="updateUser">
        update t_user
        set username='root',
                password='123',
                where id = 3
</update>

<!-- void deleteUser; -->
<delete id="deleteUser">
        delete
        from t_user
        where id = 3
</delete>

<!-- User getUserById; -->
<!--
        resultType：设置结果类型，即查询的数据要转换为 java 类型
        resultMap：自定义映射，处理多对一或一对多的映射关系
        -->
        <select id="getUserById" resultType="java.mybatis.pojo.User">
                select * from t_user where id = 1
        </select>

        <!-- List<User> getAllUser() -->
        <select id="getAllUser" resultType="User">
                select * from t_user
        </select>
</mapper>
```


2.7 获取 SqlSession 工具类

```
public class SqlSessionUtils {
        public static SqlSession getSqlSession() {
                SqlSession sqlSession = null;
                try {
                        // 获取核心配置文件的输入流
                        InputStream is = Resources.
getResourceAsStream("mybatis-config.xml");
                        // 获取 SqlSessionFactoryBuilder
                        SqlSessionFactoryBuilder sqlSessionFactoryBuilder =
new SqlSessionFactoryBuilder();
                        // 获取 SqlSessionFactory
                        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
                        // 获取 SqlSession 对象
                        sqlSession = sqlSessionFactory.openSession(true);
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return sqlSession;
        }
}
```

2.8 加入 log4j 日志功能

①加入依赖
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

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。

```
<?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>
```


2.9 MyBatis 的增删改查

```
public class MybatisTest {
    @Test
    public void testInsert() throws IOException {
        // 获取核心配置文件的输入流
        InputStream is = Resources.
getResourceAsStream("mybatis-config.xml");
        // 获取 SqlSessionFactoryBuilder 对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        // 获取 SqlSessionFactory 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);
        // 获取 UserMapper 的代理实现类对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        // 调用 mapper 接口中的方法，实现添加用户信息的功能
        int result = mapper.insertUser();
        System.out.println(" 结果 " + result);
        // 提交事务
        // sqlSession.commit();
        // 关闭 Sqlsession
        sqlSession.close();
    }
    @Test
    public void testUpdata() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        mapper.updateUser();
        sqlSession.close();
    }
```

```
    @Test
    public void testDelete() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        mapper.deleteUser();
        sqlSession.close();
    }

    @Test
    public void testGetUserById() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        User user = mapper.getUserById();
        System.out.println(user);
    }

    @Test
    public void testGetAllUser() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        List<User> list = mapper.getAllUser();
        list.forEach(System.out::println);
    }

}
```


三　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。

1、若 mapper 接口方法的参数为单个的字面量类型；

此时可以通过 #{} 和 ${} 以任意的内容获取参数值，一定要注意 ${} 的单引号问题；

2、若 mapper 接口方法的参数为多个的字面量类型；

此时 NyBatis 会将参数放在 map 集合中，以两种方式存储数据；

a> 以 arg0,arg1... 为键，以参数为值；

b> 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，一定要注意 ${} 的单引号问题；

3、若 mapper 接口方法的参数为 map 集合类型的参数；

只需要通过 #{} 和 ${} 访问 map 集合的键，就可以获取相对应的值，一定要注意 ${} 的单引号问题；

4、若 mapper 接口方法的参数为实体类类型的参数；

只需要通过 #{} 和 ${} 访问实体类中的属性勺就可以获取相对应的属性值，一定要注 ${} 的单引号问题；

5、可以在 mapper 接口方法的参数上设置 @param 注解；

此时 NyBatis 会将这些参数放在 map 中，以两种方式进行存储；

a> 以 @param 注解的 value 属性值为键，以参数为值；

b> 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，一定要注意 ${} 的单引号问题。

UserMapper 接口

```
public interface UserMapper {
    // 根据用户名查询用户信息
    User getUserByUsername(String username);

    // 验证登录
    User checkLogin(String username, String password);

    // 验证登录 ( 以 map 集合作为参数 )
    User checkLoginByMap(Map<String, Object> map);

    // 添加用户信息
    void insertUser(User user);

    // 验证登录 ( 使用 @param 注解 )
    User checkLoginByParam(@Param("username") String username,
@Param("password") String password);
}
```

UserMapper.XML

```
<?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="mybatis/mapper/UserMapper">
    <!-- User getUserByUsername(String username); -->
    <select id="getUserByUsername" resultType="User">
        select * from t_user where username = #{username}
    </select>

    <!-- User checkLogin(String username, String password);-->
    <select id="checkLogin" resultType="User">
        select * from t_user where username = #{param1} and
password = #{param2}
    </select>

    <!-- User checkLoginByMap(Map<String, Object> map);-->
    <select id="checkLoginByMap" resultType="User">
        select * from t_user where  username = #{username} and
password = #{password}
    </select>

    <!-- void insertUser(User user);-->
    <insert id="insertUser">
        insert into t_user values(null, #{username}, #{password},
#{age}, #{gender}, #{email})
    </insert>

    <!-- User checkLoginByParam(@Param("username") String
username, @Param("password") String password);-->
    <select id="checkLoginByParam" resultType="User">
        select * from t_user where username = #{username} and
password = #{password}
    </select>
</mapper>
```

parameterTest

```
public class parameterTest {
    @Test
    public void testGetUserByUsername() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        User user = mapper.getUserByUsername("admin");
        System.out.println(user);
        sqlSession.close();
    }

    @Test
    public void testCheckLogin() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        User user = mapper.checkLogin("admin", "123456");
        System.out.println(user);
        sqlSession.close();
    }

    @Test
    public void testCheckLoginByMap() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        Map<String, Object> map = new HashMap<>();
        map.put("username", "admin");
        map.put("password", "abc123");
        User user = mapper.checkLoginByMap(map);
        System.out.println(user);
        sqlSession.close();
    }

    @Test
    public void testInsertUser() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        User user = new User(null, "root", "abc123", 33, " 男 ",
"123456@qq.com");
        mapper.insertUser(user);
    }

    @Test
    public void testCheckLoginByParam() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.
class);
        User user = mapper.checkLoginByParam("admin",
"123456");
        System.out.println(user);
        sqlSession.close();
    }
}
```

四　MyBatis 的各种查询功能

查询所有的用户信息为 map 集合

若查询的数据有多条时，并且要将每条数据转换为 map 集合

此时有两种解决方案：

1、将 mapper 接口方法的返回值设置为泛型是 map 的 list 集合

List<Map<String, Object>> getAllUserToMap();

结果：{password=123456,gender= 男 ,id=1,age=23, email=12345@qq.com,username=admin}, {password=123456,gender= 男 ,id=1,age=23,email=12345@qq.com,username=admin}

2、可以将每条数据转换的 map 集合放在一个大的 map 中，但是必须要通过 @MapKey 注解

将查询的某个字段的值作为大的 map 的键

@MapKey("id")

Map<String, Object> getAllUserToMap();

SelectMapper 接口

```
public interface SelectMapper {
    // 根据 id 查询用户信息
    User getUserById(@Param("id") Integer id);

    // 查询所有的用户信息
    List<User> getAllUser();

    // 查询用户的总数量
    Integer getCount();

    // 根据 id 查询用户信息为 map 集合
    Map<String, Object> getUserByIdToMap(@Param("id") Integer id);

    //List<Map<String, Object>> getAllUserToMap();
    @MapKey("id")
    Map<String, Object> getAllUserToMap();
}
```

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。

SelectMapper.XML

```
<?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="mybatis.mapper.SelectMapper">
    <!-- User getUserById(@Param("id") Integer id); -->
    <select id="getUserById" resultType="User">
        select *
        from t_user
        where id = #{id}
    </select>

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

    <!-- Integer getCount();-->
    <select id="getCount" resultType="java.lang.Integer">
        select count(*) from t_user
    </select>

    <!-- Map<String, Object> getUserByIdToMap(@Param("id") Integer id);-->
    <select id="getUserByIdToMap" resultType="map">
        select  * from t_user where id = #{id}
    </select>

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

SelectMapperTest

```
public class SelectMapperTest {
	@Test
	public void testGetUserById() {
		SqlSession sqlSession = SqlSessionUtils.getSqlSession();
		SelectMapper mapper = sqlSession.getMapper
(SelectMapper.class);
		User user = mapper.getUserById(1);
		System.out.println(user);
	}

	@Test
	public void testGetAllUser() {
		SqlSession sqlSession = SqlSessionUtils.getSqlSession();
		SelectMapper mapper = sqlSession.getMapper
(SelectMapper.class);
		List<User> list = mapper.getAllUser();
		list.forEach(System.out::println);
	}

	@Test
	public void testGetCount() {
		SqlSession sqlSession = SqlSessionUtils.getSqlSession();
		SelectMapper mapper = sqlSession.getMapper
(SelectMapper.class);
		Integer count = mapper.getCount();
		System.out.println(count);
	}

	@Test
	public void testGetUserByIdToMap() {
		SqlSession sqlSession = SqlSessionUtils.getSqlSession();
		SelectMapper mapper = sqlSession.getMapper
(SelectMapper.class);
		Map<String, Object> map = mapper.getUserByIdToMap(1);
		System.out.println(map);
	}

	@Test
	public void testGetAllUserToMap() {
		SqlSession sqlSession = SqlSessionUtils.getSqlSession();
		SelectMapper mapper = sqlSession.getMapper
(SelectMapper.class);
//		List<Map<String, Object>> list = mapper.getAllUserToMap();
//		System.out.println(list);
		Map<String, Object> map = mapper.getAllUserToMap();
		System.out.println(map);
	}
}
```

五　特殊 SQL 的执行

SpecialSQLMapper

```
public interface SpecialSQLMapper {
    /**
     * 通过用户名模糊查询用户信息
     * @param mohu
     * @return
     */
    List<User> getUserByLike(@Param("") String mohu);

    /**
     * 批量删除
     * @param ids
     */
    void deleteMoreUser(@Param("ids") String ids);

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

    /**
     * 添加用户信息，并获取自增的主键
     * @param user
     */
    void insertUser(User user);
}
```


SpecialSQLMapper.XML

```
<?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="">
    <!-- List<User> getUserByLike(@Param("") String mohu);-->
    <select id="getUserByLike" resultType="User">
        <!--select * from t_user where username like '%#{mohu}%'-
->
        <!--select * from t_user where username like count('%',
#{mohu}, '%')-->
        select * from t_user where username like "%"#{mohu}"%"
    </select>
    <!-- void deleteMoreUser(@Param("ids") String ids);-->
    <delete id="deleteMoreUser">
        delete
        from t_user
        where id in (${ids})
    </delete>
    <!-- List<User> getUserList(@Param("tableName") String
tableName);-->
    <select id="getUserList" resultType="User">
        select *
        from ${tableName}
    </select>
    <!--       void insertUser(User user);-->
    <!--
    useGeneratedKeys 表示当前添加功能使用自增的主键
    keyProperty 将添加数据的自增主键为实体类类型参数的属性赋值
    -->
    <insert id="insertUser" useGeneratedKeys="true"
keyProperty="id">
        insert into t_user
        values (null, #{username}, #{password}, #{age}, #{gender},
#{email})
    </insert>
</mapper>
```

SpecialSQLMapperTest

```
public class SpecialSQLMapperTest {
    @Test
    public void testGetUserByLike() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SpecialSQLMapper mapper = sqlSession.getMapper
(SpecialSQLMapper.class);
        List<User> list = mapper.getUserByLike("a");
        list.forEach(System.out::println);
    }

    @Test
    public void testDeleteMoreUser() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SpecialSQLMapper mapper = sqlSession.getMapper
(SpecialSQLMapper.class);
        mapper.deleteMoreUser("9, 10");
    }

    @Test
    public void testGetUserList() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SpecialSQLMapper mapper = sqlSession.getMapper
(SpecialSQLMapper.class);
        List<User> list = mapper.getUserList("t_user");
        list.forEach(System.out::println);
    }

    @Test
    public void testInsertUser() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SpecialSQLMapper mapper = sqlSession.getMapper
(SpecialSQLMapper.class);
        User user = new User(null, "xiaoming", "123456", 23, " 男 ",
"123@qq.com");
        mapper.insertUser(user);
        System.out.println(user);
    }
```



```
    public void testUser() {
        try {
            Class.forName("");
            Connection connection = DriverManager.
getConnection("", "", "");
            // String sql = "select * from t_user where username
like '%?%'";
            // PreparedStatement ps = connection.
prepareStatement("");
            // ps.setString(1, "a");
            String sql = "insert into t_user values()";
            PreparedStatement ps = connection.
prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
            ResultSet resultSet = ps.getGeneratedKeys();
            resultSet.next();
            int id = resultSet.getInt(1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

六　自定义映射 resultMap

字段名和属性名不一致的情况，如何处理映射关系？

1、为查询的字段设置别名，和属性名保持一致

2、当字段符合 MySQL 的要求使用 _，而属性符合 java 的要求使用驼峰，此时可以在 MyBatis 的核心配置文件中设置一个全局配置，可以自动将下划线映射为驼峰

emp_id : empId, emp_name : empName

3、使用 resultMap 自定义映射处理

处理多对一的映射关系：

1、级联方式处理

2、association

3、分步查询

resultMap：设置自定义的映射关系

id：唯一标识；

type：处理映射关系的实体类的类型；

常用的标签：

id：处理主键和实体类中属性的映射关系；

result：处理普通字段和实体类中属性的映射关系；

association: 处理多对一的映射关系（处理实体类类型的属性）；

column：设置映射关系中的字段名，必须是 sql 查询出的某个字段；

collection：处理一对多的映射关系（处理集合类型的属性）；

property：设置需要处理映射关系的属性的属性名，必须是处理的实体类类型中的属性名。

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>
    <properties resource="jdbc.properties" />
    <settings>
        <!-- 将下划线映射为驼峰 -->
        <setting name="mapUnderscoreToCamelCase"
value="true" />
        <!-- 开启延迟加载 -->
        <setting name="lazyLoadingEnabled" value="true" />
        <!-- 按需加载 -->
        <setting name="aggressiveLazyLoading" value="false" />
    </settings>
    <typeAliases>
        <package name="mybatis.pojo" />
    </typeAliases>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.drive}"
/>
                <property name="url" value="${jdbc.url}" />
                <property name="username" value="${jdbc.
username}" />
                <property name="password" value="${jdbc.
password}" />
            </dataSource>
        </environment>
    </environments>

<!-- 引入映射文件 -->
    <mappers>
        <package name="mybatis.mapper" />
    </mappers>
</configuration>
```

EmpMapper.XML

```
<?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="mybatis.mapper.EmpMapper">
    <!-- Emp getEmpByEmpId(@Param("empId") Integer empId);-->
        <resultMap id="empResultMap" type="">
        <id column="emp_id" property="empId"></id>
        <result column="emp_name" property="empName"></
result>
        <result column="emp_age" property="empAge"></
result>
        <result column="emp_gender"
property="empGender"></result>
    </resultMap>
    <select id="getEmpByEmpId" resultMap="empResultMap">
        select *
        from t_emp
        where emp_id = #{empId}
    </select>

    <select id="getEmpByEmpIdOld" resultType="Emp">
        select *
        from t_emp
        where emp_id = #{empId}
    </select>

    <resultMap id="empAndDeptResultMapOne" type="Emp">
        <id column="emp_id" property="emp_id"></id>
        <result column="emp_name" property="empName"></
result>
        <result column="age" property="age"></result>
        <result column="gender" property="gender"></result>
        <result column="dept_id" property="dept.deptId"></
result>
        <result column="dept_name" property="dept.
deptName"></result>
    </resultMap>
```



```
    <resultMap id="empAndDeptResultMap" type="Emp">
        <id column="emp_id" property="emp_id"></id>
        <result column="emp_name" property="empName">
</
result>
        <result column="age" property="age"></result>
        <result column="gender" property="gender"></result>
        <!--
        association：处理多对一的映射关系（处理实体类类型的属性）
        property：设置需要处理映射关系的属性的属性名
        javaType：设置要处理的属性的类型
        -->
        <association property="dept" javaType="Dept">
            <id column="dept_id" property="deprId"></id>
            <result column="dept_name" property="deptName">
</result>
        </association>
    </resultMap>

    <!--  Emp getEmpAndDeptByEmpId(@Param("empId") Integer
empId);-->
    <select id="getEmpAndDeptByEmpId" resultType="">
        select t_emp.*,
        t_dept.*
        from t_emp
        left join t_dept
        on t_emp.dept_id = t_dept.dept_id
        where t_emp.emp_id = #{empId}
    </select>

    <resultMap id="empAndDeptByStepResultMap" type="Emp">
        <id column="emp_id" property="emp_id"></id>
        <result column="emp_name" property="empName">
</
result>
        <result column="age" property="age"></result>
        <result column="gender" property="gender"></result>
        <!--
            property：设置需要处理映射关系的属性的属性名
            select：设置分步查询的 sql 的唯一标识
            column：将查询出的某个字段作为分步查询的 sql 的条件
```

```
                fetchType：在开启了延迟加载的环境中，通过该属性设置当前的分步查询是否使用延迟加载
                fetchType="eager( 立即加载 ) | lazy( 延迟加载 )”
            -->
            <association property="dept" select="mybatis.mapper.DeptMapper.getEmpAndDeptByStepTow" column="dept_id"></association>
        </resultMap>
        <!-- Emp getEmpAndDeptByStepOne(@Param("empId") Integer empId);-->
        <select id="getEmpAndDeptByStepOne" resultMap="empAndDeptByStepResultMap">
            select *
            from t_emp
            where emp_id #{empId}
        </select>

        <!-- List<Emp> getDeptAndEmpByStepTow(@Param("deptId") Integer deptId);-->
        <select id="getDeptAndEmpByStepTow" resultType="Emp">
            select *from t_emp where dept_id = #{deptId}
        </select>
    </mapper>
```


Emp 类

```
package mybatis.pojo;

public class Emp {
    private Integer empId;
    private String empName;
    private Integer age;
    private String gender;
    private Dept dept;

    public Emp() {
    }

    public Emp(Integer empId, String empName, Integer age, String gender) {
        this.empId = empId;
        this.empName = empName;
        this.age = age;
        this.gender = gender;
    }

    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 getGender() {
        return gender;
}

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

public Emp(Dept dept) {
        this.dept = dept;
}

        @Override
        public String toString() {
                return "Emp{" + "empId=" + empId + ", empName='" +
empName + '\'' + ", age=" + age + ", gender='" + gender
                                + '\'' + ", dept=" + dept + '}';
        }
}
```


EmpMapper 接口

```
public interface EmpMapper {
        /**
         * 根据 ID 查询员工信息
         *
         * @param empId
         * @return
         */
        Emp getEmpByEmpId(@Param("empId") Integer empId);

        /**
         * 获取员工以及所对应的部门信息
         *
         * @param empId
         * @return
         */
        Emp getEmpAndDeptByEmpId(@Param("empId") Integer
empId);

        /**
         * 通过分布查询查询员工以及所对应的部门信息的第一步
         *
         * @param empId
         * @return
         */
        Emp getEmpAndDeptByStepOne(@Param("empId") Integer
empId);

}
```

DeptMapper 接口

```
public interface DeptMapper {
    /**
     * 通过分步查询查询员工以及所对应的部门信息的第二步
     *
     * @return
     */
    Dept getEmpAndDeptByStepTow(@Param("deptId") Integer
deptId);

    /**
     * 查询部门以及部门中的员工信息
     *
     * @param deptId
     * @return
     */
    Dept getDeptAndEmpByDeptId(@Param("deptId") Integer
deptId);

    /**
     * 通过分步查询查询部门以及部门中的员工信息的第一步
     *
     * @param deptId
     * @return
     */
    Dept getDeptAndEmpByStepOne(@Param("deptId") Integer
deptId);

    /**
     * 通过分步查询查询部门以及部门中的员工信息的第二步
     *
     * @param deptId
     * @return
     */
    List<Emp> getDeptAndEmpByStepTow(@Param("deptId")
Integer deptId);
}
```


DeptMapper.XML

```
<?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="mybatis.mapper.DeptMapper">

    <!-- Dept getEmpAndDeptByStepTow(@Param("deptId") Integer
deptId);-->
    <select id="getEmpAndDeptByStepTow" resultType="Dept">
        select * from t_dept where dept_Id = #{deptId}
    </select>

    <resultMap id="deptAndEmpResultMap" type="Dept">
        <id column="dept_id" property="deptId"></id>
        <result column="dept_name" property="deprName">></
result>
        <!-- ofType：设置集合类型的属性中存储的数据的类型 -->
        <collection property="emps" ofType="Emp">
            <id column="emp_id" property="empId"></id>
            <result column="emp_name"
property="empName"></result>
            <result column="age" property="age"></result>
            <result column="gender" property="gender"></
result>
        </collection>
    </resultMap>
    <!-- Dept getDeptAndEmpByDeptId(@Param("deptId") Integer
deptId);-->
    <select id="getDeptAndEmpByDeptId" resultMap="deptAndEm
pResultMap">
        select * from t_dept
        left join t_emp
        on t_dept.dept_id = t_emp.dept_id
        where t_dept.dept_id = #{deptId}
    </select>

    <resultMap id="deptAndEmpResultByStep" type="Dept">
        <id column="dept_id" property="deptId"></id>
```

```
        <result column="dept_name" property="deprName"></
result>
        <collection property="emps" select="mybatis.mapper.
DeptMapper.getDeptAndEmpByStepTow" column="dept_id"></
collection>
    </resultMap>
    <!-- Dept getDeptAndEmpByStepOne(@Param("deptId")
Integer deptId);-->
    <select id="getDeptAndEmpByStepOne" resultMap="deptAndE
mpResultByStep">
        select * from t_dept where dept_id = #{deptId}
    </select>

</mapper>
```

DeptMapper.XML

```
public class ResultMapTest {
    @Test
    public void testGetEmpByEmpId() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.
class);
        Emp emp = mapper.getEmpByEmpId(1);
        System.out.println(emp);
    }

    @Test
    public void testGetEmpAndDeptByEmpId() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.
class);
        Emp emp = mapper.getEmpAndDeptByEmpId(1);
        System.out.println(emp);
    }
```



```
    @Test
    public void testGetEmpAndDeptByStep() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.
class);
        Emp emp = mapper.getEmpAndDeptByStepOne(2);
        System.out.println(emp.getEmpName());
    }

    @Test
    public void testGetDeptAndEmpByDeptId() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DeptMapper mapper = sqlSession.getMapper
(DeptMapper.class);
        Dept dept = mapper.getDeptAndEmpByDeptId(1);
        System.out.println(dept);
    }

    @Test
    public void testGetDeptAndEmpByStep() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DeptMapper mapper = sqlSession.getMapper
(DeptMapper.class);
        Dept dept = mapper.getDeptAndEmpByStepOne(1);
        System.out.println(dept);
    }
}
```

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)"。

七　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。

1、if，通过 test 属性中的表达式判断标签中的内容是否有效（是否会拼接到 sql 中）；

2、where

a. 若 where 标签中有条件成立，会自动生成 where 关键字

b. 会自动将 where 标签中内容前多余的 and 去掉，但是其中内容后多余的 and 无法去掉；

c. 若 where 标签中没有任何一个条件成立, 则 where 没有任何功能。

3、trim

prefix、suffix：在标签中内容前面或后面添加指定内容；

prefixOverrides、suffixOverrides: 在标签中内容前面或后面去掉指定内容；

4、choose, when, otherwise

相当于 java 中的 if...else if...else；

when 至少设置一个，otherwise 最多设置一个；

5、foreach

collection：设置要循环的数组或集合；

item：用一个字符串表示数组或集合中的每一个数据；

separator：设置每次循环的数据之间的分隔符；

open：循环的所有内容以什么开始；

close：循环的所有内容以什么结束。

6、sql 片段

可以记录一段 sql，在需要用的地方使用 include 标签进行引用。

```
<sql id="empColumns">
        emp_id, emp_name, age, gender, dept_id
</sql>
<include refid="empColumns"></include>
```


emp 类

```
package mybatis.pojo;

public class Emp {
    private Integer empId;
    private String empName;
    private Integer age;
    private String gender;

    public Emp() {
    }

    public Emp(Integer empId, String empName, Integer age, String
gender) {
        this.empId = empId;
        this.empName = empName;
        this.age = age;
        this.gender = gender;
    }

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 getGender() {
        return gender;
    }

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

    @Override
    public String toString() {
        return "Emo{" + "empId=" + empId + ", empName='" +
empName + '\'' + ", age=" + age + ", gender='" + gender
                + '\'' + '}';
    }
}
```


DynamicSQLMapper 接口

```
public interface DynamicSQLMapper {
    /**
     * 根据条件查询员工信息
     *
     * @param emp
     * @return
     */
    List<Emp> getEmpByCondition(Emp emp);

    /**
     * 使用 choose 查询员工信息
     *
     * @param emp
     * @return
     */
    List<Emp> getEmpByChoose(Emp emp);

    /**
     * 批量添加员工信息
     *
     * @param emps
     */
    void insertMoreEmp(@Param("emps") List<Emp> emps);

    /**
     * 批量删除
     *
     * @param empIds
     */
    void deleteMoreEmp(@Param("empIds") Integer[] empIds);

}
```

DynamicSQLMapper.XML

```
<?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="mybatis.mapper.DynamicSQLMapper">

    <sql id="empColumns">
        emp_id, emp_name, age, gender, dept_id
    </sql>
    <!-- List<Emp> getEmpByCondition(Emp emp);-->
    <select id="getEmpByCondition" resultType="Emp">
        select<include refid="empColumns"></include>
            from t_emp
        <trim prefix="where" suffixOverrides="and">
            <if test="empName != null and empName != ' '">
                emp_name = #{empName} and
            </if>
            <if test="age != null and age != ' '">
                age = #{age} and
            </if>
            <if test="gender != null and gender != ''">
                gender = #{gender}
            </if>
        </trim>
    </select>
    <select id="getEmpByConditionTow" resultType="Emp">
        select * from t_emp
        <where>
            <if test="empName != null and empName != ' '">
                emp_name = #{empName}
            </if>
            <if test="age != null and age != ' '">
                and age = #{age}
            </if>
            <if test="gender != null and gender != ''">
                and gender = #{gender}
            </if>
        </where>
```



```
    </select>
    <select id="getEmpByConditionOne" resultType="Emp">
        select * from t_emp where 1=1
        <if test="empName != null and empName != ' '">
            and emp_name = #{empName}
        </if>
        <if test="age != null and age != ' '">
            and age = #{age}
        </if>
        <if test="gender != null and gender != ''">
            and gender = #{gender}
        </if>
    </select>

    <!--    List<Emp> getEmpByChoose(Emp emp);-->
    <select id="getEmpByChoose" resultType="emp">
        select * from t_emp
        <where>
            <choose>
                <when test="empName != null and empName
!= ''">
                    emp_name = #{empName}
                </when>
                <when test="age != null and age != ''">
                    age = #{age}
                </when>
                <when test="gender != null and gender != ''">
                    gender = #{gender}
                </when>
            </choose>
        </where>
    </select>

    <!--    void insertMoreEmp(@Param("emps") List<Emp>
emps);-->
    <select id="insertMoreEmp">
        insert into t_emp values
        <foreach collection="list" item="emp" separator=",">
            (null, #{emp.empName} #{emp.age} #{emp.gender},
null)
```

```
            </foreach>
        </select>

        <!--        void deleteMoreEmp(@Param("empIds") Integer[]
empIds);-->
        <delete id="deleteMoreEmp">
            <!-- delete from t_emp where emp_id in
                 <foreach collection="empIds" item="empId"
separator="," open="(" close=")">
                    #{empId}
                    </foreach>-->
            delete from t_emp where
            <foreach collection="empIds" item="empId"
separator="or">
                emp_id = #{empId}
            </foreach>
        </delete>
</mapper>
```

DynamicMapperTest

```
public class DynamicMapperTest {
    @Test
    public void testGetEmpByCondition() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DynamicSQLMapper mapper = sqlSession.getMapper
(DynamicSQLMapper.class);
        Emp emp = new Emp(null, " 张三 ", 20, " 男 ");
        List<Emp> list = mapper.getEmpByCondition(emp);
        list.forEach(System.out::println);
    }
```



```
    @Test
    public void testGetEmpByChoose() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DynamicSQLMapper mapper = sqlSession.getMapper
(DynamicSQLMapper.class);
        Emp emp = new Emp(null, " 张三 ", 20, "");
        List<Emp> list = mapper.getEmpByChoose(emp);
        list.forEach(System.out::println);
    }

    @Test
    public void testInsertMoreEmp() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DynamicSQLMapper mapper = sqlSession.getMapper
(DynamicSQLMapper.class);
        Emp emp1 = new Emp(null, " 小明 1", 20, " 男 ");
        Emp emp2 = new Emp(null, " 小明 2", 20, " 男 ");
        Emp emp3 = new Emp(null, " 小明 3", 20, " 男 ");
        List<Emp> list = Arrays.asList(emp1, emp2, emp3);
        mapper.insertMoreEmp(list);
    }

    @Test
    public void testDeleteMoreEmp() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        DynamicSQLMapper mapper = sqlSession.
getMapper(DynamicSQLMapper.class);
        Integer[] empIds = new Integer[] { 6, 7 };
        mapper.deleteMoreEmp(empIds);
    }
}
```

八 MyBatis 的缓存

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。

MyBatis 的二级缓存：

MyBatis 的二级缓存是 sqlSessionFactory 级别的，即通过同一个 sqlSessionFactory 所获取的 sqlSession 对象查询的数据会被缓存，在通过同一个 SqlSessionFactory 所获取的 sqlSession 查询相同的数据会从缓存中获取。

二级缓存开启的条件：

1) 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；
2) 在映射文件中设置标签 <cache/>；
3) 二级缓存必须在 SqlSession 关闭或提交之后有效；
4) 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。

DynamicSQLMapper 接口

```
public interface CacheMapper {
    /**
     * 根据员工 ID 查询员工信息
     * @param empId
     * @return
     */
    Emp getEmpById(@Param("empId") Integer empId);

    /**
     * 添加员工信息
     * @param emp
     */
    void insertEmp(Emp emp);
}
```

CacheMapperTest

```
public class CacheMapperTest {
    @Test
    public void testCache() throws IOException {
        InputStream is = Resources.
getResourceAsStream("mybatis-config.xml");
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(is);
        SqlSession sqlSession1 = sqlSessionFactory.
openSession(true);
        CacheMapper mapper1 = sqlSession1.
getMapper(CacheMapper.class);
        Emp emp1 = mapper1.getEmpById(1);
        System.out.println(emp1);
        SqlSession sqlSession2 = sqlSessionFactory.
openSession(true);
        CacheMapper mapper2 = sqlSession2.
getMapper(CacheMapper.class);
        Emp emp2 = mapper2.getEmpById(1);
        System.out.println(emp2);
        sqlSession2.close();
    }
```

```
    @Test
    public void testGetEmpById() {
        SqlSession sqlSession1 = SqlSessionUtils.getSqlSession();
        CacheMapper mapper1 = sqlSession1.
getMapper(CacheMapper.class);
        Emp emp1 = mapper1.getEmpById(1);
        System.out.println(emp1);
        sqlSession1.clearCache();
        // mapper1.insertEmp(new Emp(null, " 小红 ", 23, " 男 "));
        Emp emp2 = mapper1.getEmpById(1);
        System.out.println(emp2);
        /*
         * SqlSession sqlSession2 = SqlSessionUtils.getSqlSession();
CacheMapper mapper2
         * = sqlSession2.getMapper(CacheMapper.class); Emp
emp3 = mapper2.getEmpById(1);
         * System.out.println(emp3);
         */
    }
}
```

CacheMapper.XML

```
<?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="mybatis.mapper.CacheMapper">
    <cache type="org.mybatis.caches.ehcache.EhcacheCache" />
    <!-- Emp getEmpById(@Param("empId") Integer empId);-->
    <select id="getEmpById" resultType="Emp">
        select * from t_emp where emp_id = #{empId}
    </select>
    <!-- void insertEmp(Emp emp);-->
    <insert id="insertEm">
        insert into t_emp values(null, #{empName}, #{age},
#{gender}, null)
    </insert>
</mapper>
```


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：

① eviction 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。

② flushInterval 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。

③ size 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。

④ readOnly 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。

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。

整合第三方缓存 EHCache

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="mybatis.mapper" level="DEBUG" />
</configuration>
```

设置二级缓存的类型

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


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>
```

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="H:\JAVA000"/>
	<defaultCache
		maxElementsInMemory="1000"
		maxElementsOnDisk="10000000"
		eternal="false"
		overflowToDisk="true"
		timeToIdleSeconds="120"
		timeToLiveSeconds="120"
		diskExpiryThreadIntervalSeconds="120"
		memoryStoreEvictionPolicy="LRU">
	</defaultCache>
</ehcache>
```

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

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）

九 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

创建逆向工程的步骤

①添加依赖和插件

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

    <groupId>mybatis.mbg</groupId>
    <artifactId>mybatis-mbg</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
    </properties>

    <packaging>jar</packaging>
    <!-- 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>
    <dependency>
        <groupId>com.github.pagehelper</groupId>
        <artifactId>pagehelper</artifactId>
        <version>5.2.0</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>
        <plugin>
            <groupId>org.apache.maven.plugins</
groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>16</source>
                <target>16</target>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

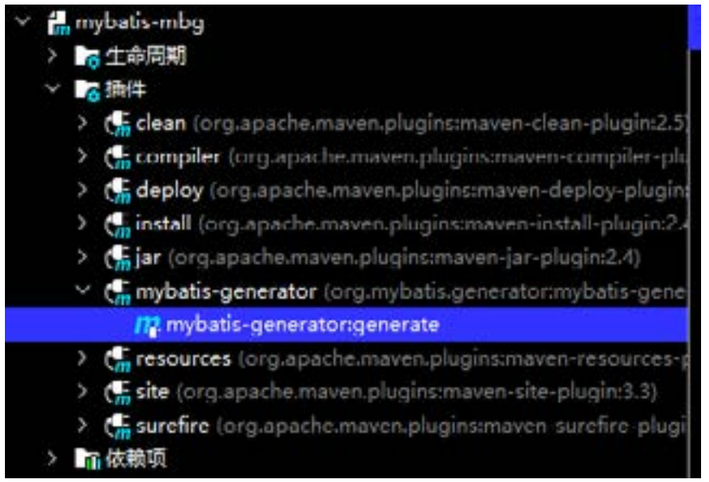
②创建 MyBatis 的核心配置文件

③创建逆向工程的配置文件

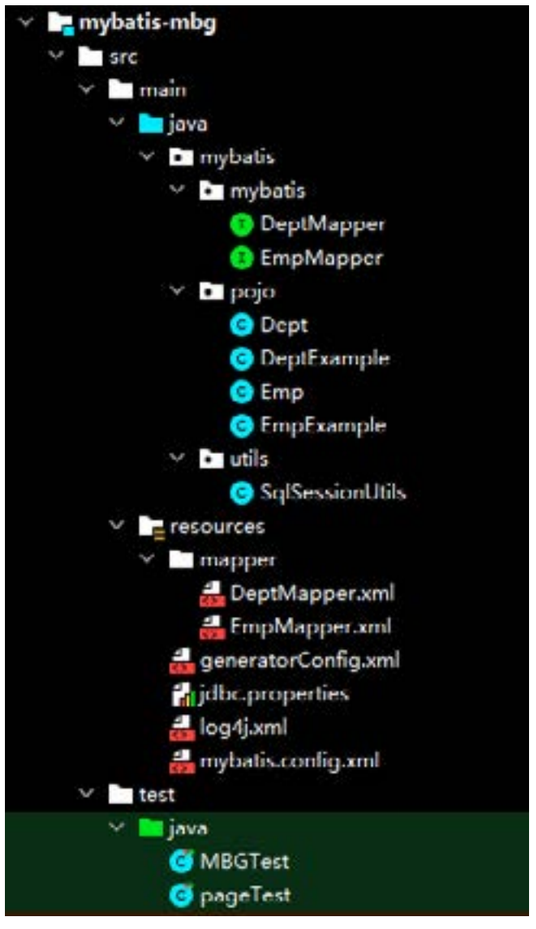
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD
MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/
mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--
    targetRuntime: 执行生成的逆向工程的版本
    MyBatis3Simple: 生成基本的 CRUD（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/
ssm?serverTimezone=UTC" userId="root" password="abc123"></
jdbcConnection>
            <!-- javaBean 的生成策略 -->
            <javaModelGenerator targetPackage="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="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



QBC 查询

```
public class MBGTest {
    @Test
    public void testMBG() {
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.
class);
        // 根据 ID 查询数据
        /*
         * Emp emp = mapper.selectByPrimaryKey(1); System.out.
println(emp);
         */
        // 查询所有数据
        /*
         * List<Emp> list = mapper.selectByExample(null);
         * list.forEach(System.out::println);
         */
        // 根据条件查询
        /*
         * EmpExample example = new EmpExample();
         * example.createCriteria().andEmpNameEqualTo(" 张三 ").
andAgeGreaterThanOrEqualTo(20);
         * example.or().andGenderEqualTo(" 男 ");
         * List<Emp> list =mapper.selectByExample(example);
         * list.forEach(System.out::println);
         */

        Emp emp = new Emp(1, " 小黑 ", null, " 女 ");
        // 测试普通修改功能
        // mapper.updateByPrimaryKey(emp);
        // 测试选择性修改功能
        mapper.updateByPrimaryKeySelective(emp);
    }
}
```


十　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

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

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

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]

pageTest

```
public class pageTest {
    @Test
    public void testPage(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        // 查询功能之前开启分页功能
        Page<Object> page = PageHelper.startPage(1, 4);
        List<Emp> list = mapper.selectByExample(null);
        list.forEach(System.out::println);
        System.out.println(page);
    }
}
```

Spring

一 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 JavaEE 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 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。

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。

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。

二 IOC

2.1 IOC 容器

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。

③ DI

· 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。

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：

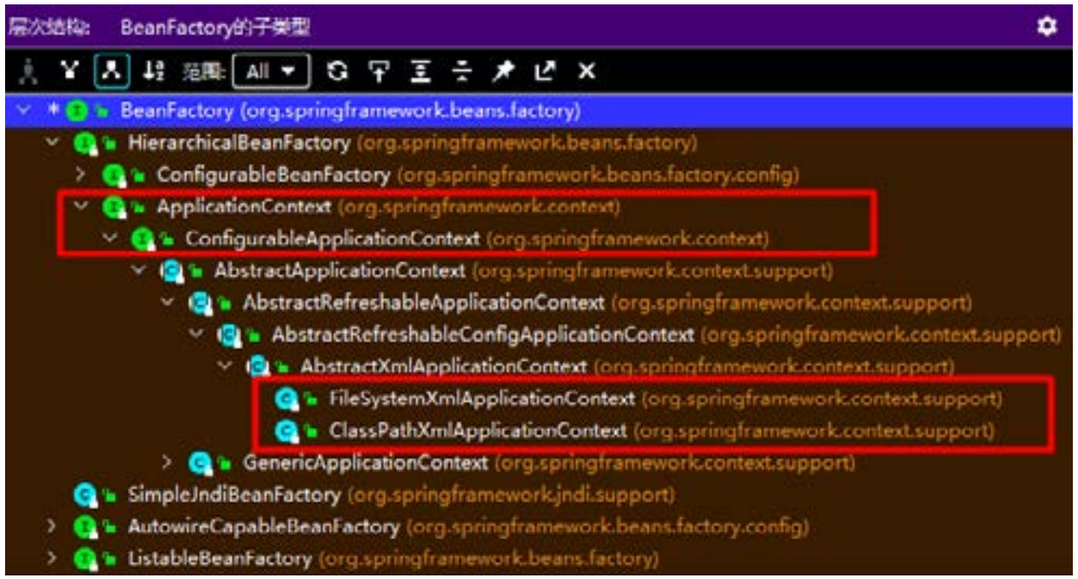
① BeanFactory

· 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。

② ApplicationContext

· 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。

③ ApplicationContext 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

实验一：入门案例

①创建 Maven Module
②引入依赖

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

    <groupId>spring</groupId>
    <artifactId>spring-helloworld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</
groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>16</source>
                    <target>16</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <properties>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
    </properties>
    <packaging>jar</packaging>
    <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>

</project>
```

③创建类 HelloWorld

```
public class HelloWorld {
    public void sayHello(){
        System.out.println("hello, spring");
    }
}
```

④创建 Spring 的配置文件

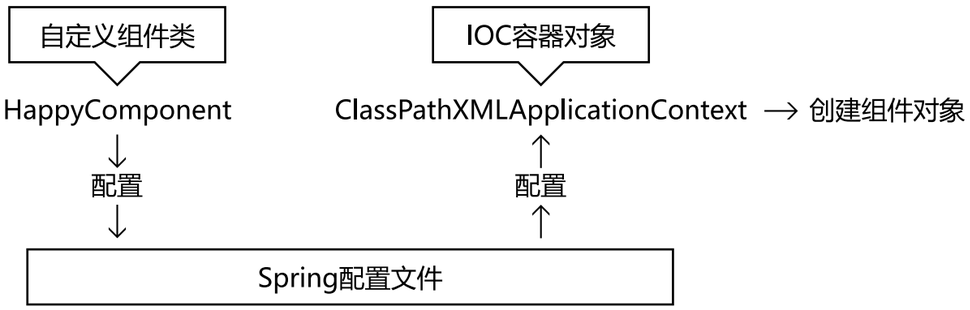
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd">

    <!--
        bean：配置一个 bean 对象，将对象交给 IOc 容器管理
        属性：
        id：bean 的唯一标识，不能重复
        cLass：设置 bean 对象所对应的类型
    -->
    <bean id="helloworld" class="spring.pojo.HelloWorld"></
bean>
</beans>
```

⑥创建测试类测试

```
public class HelloWorldTest {
    @Test
    public void test() {
        // 获取 IOC 容器
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("applicationContext.xml");
        // 获取 IOC 容器中的 bean
        HelloWorld helloworld = (HelloWorld) ioc.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> ()
```


①创建学生类 Student

```
public class Student implements Person {
    private Integer sid;
    private String sname;
    private Integer age;
    private String gender;
    private Double score;
    private Clazz clazz;
    private String[] hobby;
    private Map<String, Teacher> teacherMap;

    public Student() {
    }

    public Student(Integer sid, String sname, String gender, Integer
age) {
        this.sid = sid;
        this.sname = sname;
        this.gender = gender;
        this.age = age;
    }

    public Student(Integer sid, String sname, String gender, Double
score) {
        this.sid = sid;
        this.sname = sname;
        this.gender = gender;
        this.score = score;
    }

    public Integer getSid() {
        return sid;
    }

    public void setSid(Integer sid) {
        this.sid = sid;
    }
```

```
public String getSname() {
        return sname;
}

public void setSname(String sname) {
        this.sname = sname;
}

public Integer getAge() {
        return age;
}

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

public String getGender() {
        return gender;
}

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

public Double getScore() {
        return score;
}

public void setScore(Double score) {
        this.score = score;
}

public Clazz getClazz() {
        return clazz;
}

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



```
public String[] getHobby() {
        return hobby;
}

public void setHobby(String[] hobby) {
        this.hobby = hobby;
}

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

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

@Override
public String toString() {
        return "Student{" +
                "sid=" + sid +
                ", sname='" + sname + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\''+
                ", score=" + score +
                ", clazz=" + clazz +
                ", hobby=" + Arrays.toString(hobby) +
                ", teacherMap="
                        + teacherMap + '}';
        }
}
```

pring-ioc.xml

②配置 bean 时为属性赋值

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

    <bean id="studentOne" class="spring.pojo.Student"></bean>
    <bean id="studentTow" class="spring.pojo.Student">
        <!--
            property：通过成员变量的 set 方法进行赋值
            name：设置需要赋值的属性名（和 set 方法有关）
            value：设置为属性所赋的值
        -->
        <property name="sid" value="1001"></property>
        <property name="sname" value=" 张三 "></property>
        <property name="age" value="23"></property>
        <property name="gender" value=" 男 "></property>
    </bean>

    <bean id="studentThree" class="spring.pojo.Student">
        <constructor-arg value="1002"></constructor-arg>
        <constructor-arg value=" 李四 "></constructor-arg>
        <constructor-arg value=" 女 "></constructor-arg>
        <constructor-arg value="24" name="age"></constructor-arg>
    </bean>

    <bean id="studentFour" class="spring.pojo.Student">
        <property name="sid" value="1003"></property>
        <!--
            <: &lt;
            >: &gt;
            CDATA 节其中的内容会原样解析 <![CDATA[<。。。>]]>
            CDATA 节是 xml 中一个特殊的标签，因此不能写在一个属性中
        -->
        <!--<property name="sname" value="&lt; 王 五 &gt;"></property>-->
        <property name="sname">
            <value>
                <![CDATA[< 王五 >]]>
            </value>
        </property>
        <property name="gender">
            <null></null>
        </property>
    </bean>

    <bean id="studentFive" class="spring.pojo.Student">
        <property name="sid" value="1004"></property>
        <property name="sname" value=" 赵六 "></property>
        <property name="age" value="26"></property>
        <property name="gender" value=" 男 "></property>
        <!-- ref：引用 IOC 容器中的某个 bean 的 id -->
        <!-- <property name="clazz" ref="clazzOne"></property>-->
        <!-- 级联的方式，要保证提前为 clazz 属性赋值或者实例化 -->

        <!--<property name="clazz.cid" value="2222"></property>-->
        <!--<property name="clazz.cname" value=" 远 大 前 程 班 "></property>-->
        <property name="clazz">
            <!-- 内部 bean，只能在当前 bean 的内部使用，不能直接通过 IOc 容器获取 -->
            <bean id="clazzInner" class="spring.pojo.Clazz">
                <property name="cid" value="2222"></property>
                <property name="cname" value=" 远 大 前 程 班 "></property>
            </bean>
        </property>
        <property name="hobby">
```

```
            <array>
                <value> 抽烟 </value>
                <value> 喝酒 </value>
                <value> 烫头 </value>
            </array>
        </property>
        <property name="teacherMap" ref="teacherMap"></
property>
        <!--<property name="teacherMap">
            <map>
                <entry key="10086" value-ref="teacherOne"></
entry>
                <entry key="10010" value-ref="teacherTow"></
entry>
            </map>
        </property>-->
    </bean>

    <bean id="clazzOne" class="spring.pojo.Clazz">
        <property name="cid" value="1111"></property>
        <property name="cname" value=" 最 强 王 者 班 "></
property>
        <property name="students" ref="studentList"></
property>
        <!--<property name="students">
            <list>
                <ref bean="studentOne"></ref>
                <ref bean="studentTow"></ref>
                <ref bean="studentThree"></ref>
            </list>
        </property>-->
    </bean>

    <bean id="teacherOne" class="spring.pojo.Teacher">
        <property name="tid" value="10086"></property>
        <property name="tname" value=" 大宝 "></property>
    </bean>

    <bean id="teacherTow" class="spring.pojo.Teacher">
        <property name="tid" value="10010"></property>
        <property name="tname" value=" 小宝 "></property>
    </bean>
    <!-- 配置一个集合类型的 bean，需要使用 util 的约束 -->
    <util:list id="studentList">
        <ref bean="studentOne"></ref>
        <ref bean="studentTow"></ref>
        <ref bean="studentThree"></ref>
    </util:list>

    <util:map id="teacherMap">
        <entry key="10086" value-ref="teacherOne"></entry>
        <entry key="10010" value-ref="teacherTow"></entry>
    </util:map>

    <bean id="studentSix" class="spring.pojo.Student" p:sid="1005"
p:sname=" 小明 " p:teacherMap-ref="teacherMap"></bean>
</beans>
```

IOCByXMLTest

获取 bean 的三种方式：
1、根据 bean 的 id 获取
2、根据 bean 的类型获取
　　注意：根据类型获取 bean 时，要求 IOc 容器中有且只有一个类型匹配的 bean，没有任何一个类型匹配的 bean，此时抛出异常：NosuchBeanDefinitionException。若有多个类型匹配的 bean，此时抛出异常：NouniqueBeanDefinitionException。
3、根据 bean 的 id 和类型获取
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。
　　即通过 bean 的类型、bean 所继承的类的类型、bean 所实现的接口的类型都可以获取 bean。

```
public class IOCByXMLTest {

    @Test
    public void testIOC() {
        // 获取 IOC 容器
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-ioc.xml");
        // 获取 bean
//      Student studentOne = (Student) ioc.getBean("studentOne");
//      Student student = ioc.getBean(Student.class);
//      Student student = ioc.getBean("studentOne", Student.class);
        Person person = ioc.getBean(Person.class);
        System.out.println(person);
    }

    @Test
    public void testDI() {
        // 获取 IOC 容器
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-ioc.xml");
        // 获取 bean
        Student student = ioc.getBean("studentSix", Student.class);
        System.out.println(student);
//          Clazz clazz = ioc.getBean("clazzInner", Clazz.class);
//          System.out.println(clazz);
//          Clazz clazz = ioc.getBean("clazzOne", Clazz.class);
//          System.out.println(clazz);
    }
}
```

Person 接口

```
public interface Person {
}
```


Clazz

```
public class Clazz {
    private Integer cid;
    private String cname;
    private List<Student> students;

    public Clazz() {
    }
    public Clazz(Integer cid, String cname) {
        this.cid = cid;
        this.cname = cname;
    }
    public Integer getCid() {
        return cid;
    }
    public void setCid(Integer cid) {
        this.cid = cid;
    }
    public String getCname() {
        return cname;
    }
    public void setCname(String cname) {
        this.cname = cname;
    }
    public List<Student> getStudents() {
        return students;
    }
    public void setStudents(List<Student> students) {
        this.students = students;
    }

    @Override
    public String toString() {
        return "Clazz{" + "cid=" + cid + ", cname='" + cname + '\''
 + ", students=" + students + '}';
    }
}
```

Teacher

```
public class Teacher {
    private Integer tid;
    private String tname;

    public Teacher() {
    }

    public Teacher(Integer tid, String tname) {
        this.tid = tid;
        this.tname = tname;
    }

    public Integer getTid() {
        return tid;
    }

    public void setTid(Integer tid) {
        this.tid = tid;
    }

    public String getTname() {
        return tname;
    }

    public void setTname(String tname) {
        this.tname = tname;
    }

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


datasuorce.xml

```
<?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">

    <!-- 引入 jdbc.properties 之后可以通过 ￥{key} 的方式访问 value-->
    <context:property-placeholder location="jdbc.properties"></
context:property-placeholder>
    <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>
</beans>
```

DataSourcesTest

```
public class DataSourcesTest {
    @Test
    public void testDataSource() throws SQLException {
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-datasources.xml");
        DruidDataSource dataSource = ioc.getBean
(DruidDataSource.class);
        System.out.println(dataSource.getConnection());
    }
}
```

jdbc.properties

```
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.Driver=com.mysql.cj.jdbc.Driver
jdbc.username=root
jdbc.password=abc123
```

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() {
        System.out.println(" 生命周期 1：实例化 ");
    }

    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;
    }

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

    public void initMethod() {
        System.out.println(" 声明周期 3：初始化 ");
    }

    public void destroyMethod() {
        System.out.println(" 声明周期 4：销毁 ");
    }
}
```


③配置 bean

spring-scope.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd">
    <!--
        scope：设置 bean 的作用域
        scope="singleton|prototype"
        singleton（单例）：表示获取该 bean 所对应的对象都是同一个
        prototype（多例）：表示获取该 bean 所对应的对象都不是同
一个
    -->
    <bean id="student" class="spring.pojo.Student"
scope="singleton">
        <property name="sid" value="1001"></property>
        <property name="sname" value=" 张三 "></property>
    </bean>
</beans>
```

④测试

scopeTest

```
public class ScopeTest {
    @Test
    public void testScope() {
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-scope.xml");
        Student student1 = ioc.getBean(Student.class);
        Student student2 = ioc.getBean(Student.class);
        System.out.println(student1 == student2);
    }
}
```

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

spring-lifecycle.xml

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

    <bean id="user" class="spring.pojo.User" init-method=
"initMethod" destroy-method="destroyMethod">
        <property name="id" value="1"></property>
        <property name="username" value="admin"></property>
        <property name="password" value="abc123"></
property>
        <property name="age" value="23"></property>
    </bean>

    <bean id="myBeanProcessor" class="spring.process.
MyBeanProcessor"></bean>
</beans>
```


1、实例化
2、依赖注入
3、后置处理器的 postprocessBeforeInitialization
4、初始化，需要通过 bean 的 init-method 属性指定初始化的方法
5、后置处理器的 postprocessAfterInitialization
6、Ioc 容器关闭时销毁，需要通过 bean 的 destroy-method 属性指定销毁的方法

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。

注意：

若 bean 的作用域为单例时，生命周期的前三个步骤会在获取 IOC 容器时执行；

若 bean 的作用域为多例时，生命周期的前三个步骤会在获取 bean 时执行。

LifeCycleTest

```
public class LifeCycleTest {
    @Test
    public void test() {
        // ConfigurableApplicationContext 是 ApplicationContext
的子接口，其中扩展了刷新和关闭容器的方法
        ConfigurableApplicationContext ioc = new ClassPathXmlA
pplicationContext("spring-lifecycle.xml");
        User user = ioc.getBean(User.class);
        System.out.println(user);
        ioc.close();
    }
}
```

bean 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。

```
public class MyBeanProcessor implements BeanPostProcessor {
    @Override
    // 此方法在 bean 的生命周期初始化之前执行
    public Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
        System.out.println("MyBeanProcessor--> 后置处理器 postPr
ocessBeforeInitialization");
        return bean;
    }

    @Override
    // 此方法在 bean 的生命周期初始化之后执行
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("MyBeanProcessor--> 后置处理器 postPr
ocessAfterInitialization");
        return bean;
    }
}
```


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。

②创建类 UserFactoryBean

FactoryBean 是一个接口，需要创建一个类实现该接口

其中有三个方法：

getobject（）：通过一个对象交给 rOc 容器管理

getobjectType（)：设置所提供对象的类型

issingleton（）：所提供的对象是否单例

当把 FactoryBean 的实现类配置为 bean 时，会将当前类中 getObject（）所返回的对象交给 Ioc 容器管理。

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

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

pring-factory.xml

```
<bean class="spring.Factory.UserFactoryBean"></bean>
```

基于 xml 的自动装配

自动装配：

根据指定的策略，在 IOc 容器中匹配某个 bean，自动为 bean 中的类类型的属性或接口类型的属性赋值。

可以通过 bean 标签中的 autowire 属性设置自动装配的策略。

自动装配的策略：

1、no，default：表示不装配，即 oean 中的属性不会自动匹配某个 bean 为属性赋值，此时属性使用默认值

2、byType：根据要赋值的属性的类型，在 IOC 容器中匹配某个 bean，为属性赋值

注意：

a> 若通过类型没有找到任何一个类型匹配的 bean，此时不装配，属性使用默认值；

b> 若通过类型找到了多个类型匹配的 bean，此时会抛出异常：NoUniqueBeanDefinitionException；

总结：当使用 byType 实现自动装配时，IOC 容器中有且只有一个类型匹配的 bean 能够为属性赋值。

3、byName：将要赋值的属性的属性名作为 bean 的 id 在 IOC 容器中匹配某个 bean，为属性赋值。

总结：当类型匹配的 bean 有多个时，此时可以使用 byName 实现自动装配。

①场景模拟

UserController

```
public class UserController {
    private UserService userService;

    public UserService getUserService() {
        return userService;
    }
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    public void saveUser() {
        userService.saveUser();
    }
}
```


UserService 接口

```
public interface UserService {
    /**
     * 保存用户信息
     */

    void saveUser();
}
```

UserServiceImpl

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;

    public UserDao getUserDao() {
        return userDao;
    }

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

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

    }
}
```

UserDao 接口

```
public interface UserDao {
    /**
     * 保存用户信息
     */
    void saveUser();
}
```

UserServiceImpl

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

spring-autowire-xml.xml

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

    <bean id="userController" class="spring.Controller.
UserController" autowire="byName">
        <!-- <property name="userService" ref="userService"></
property>-->
    </bean>

    <bean id="userService" class="spring.Service.impl.
UserServiceImpl" autowire="byName">
        <!--<property name="userDao" ref="userDao"></
property>-->
    </bean>

    <bean id="service" class="spring.Service.impl.UserServiceImpl"
autowire="byName">
        <!--<property name="userDao" ref="userDao"></
property>-->
    </bean>

    <bean id="userDao" class="spring.dao.impl.UserDaoImpl"></
bean>
    <bean id="Dao" class="spring.dao.impl.UserDaoImpl"></
bean>
</beans>
```


AutoWireByXMLTest

```
public class AutoWireByXMLTest {

    @Test
    public void testAutowire() {
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-autowire-xml.xml");
        UserController userController = ioc.getBean(UserController.
class);
        userController.saveUser();
    }
}
```

2.3 基于注解管理 bean

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

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

    <groupId>com.atguigu.spring</groupId>
    <artifactId>spring_ioc_annotation</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <packaging>jar</packaging>

    <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>
</project>
```

④创建 Spring 配置文件

context:exclude-filter：排除扫描；
type：设置排除扫描的方式；
type="annotation|assignable"；
annotation：根据注解的类型进行排除，expression 需要设置排除的注解的全类名。
assignable：根据类的类型进行排除，expression 需要设置排除的类的全类名。

context:include-filter: 包含扫描；
注意：需要在 context:component-scan 标签中设置 use-default-filters="false"。
use-default-filters=“true”（默认），所设置的包下所有的类都需要扫描，此时可以使用排除扫描。
use-default-filters=“false“，所设置的包下所有的类都不需要扫描，此时可以使用包含扫描。

```xml
<?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="spring">
</context:component-scan>
       <!--<context:exclude-filter type="annotation" expression="org.
springframework.stereotype.Controller"/>
              <context:exclude-filter type="assignable" expression=
"spring.Controller.UserController"/>
              <context:include-filter type="annotation" expression="org.
springframework.stereotype.Controller"/>
              </context:component-scan>-->
       <bean id="service" class="spring.Service.Impl.
UserServiceImpl"></bean>
       <bean id="dao" class="spring.Dao.Impl.UserDaoImpl"></
bean>
</beans>
```


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

通过注解 + 扫描所配置的 bean 的 id，默认值为类的小驼峰，即类名的首字母为小写的结果可以通过标识组件的注解的 value 属性值设置 beon 的自定义的 id

@Autowired：实现自动装配功能的注解

1、@Autowired 注解能够标识的位置

a> 标识在成员变量上，此时不需要设置成员变量的 set 方法
b> 标识在 set 方法上
c> 标识在为当前成员变量赋值的有参构造上

2、@Autowired 注解的原理

a> 默认通过 byType 的方式，在 IOc 容器中通过类型匹配某个 bean 为属性赋值
b> 若有多个类型匹配的 bean，此时会自动转换为 byName 的方式实现自动装配的效果。即将要赋值的属性的属性名作为 bean 的 id 匹配某个 bean 为属性赋值。
c> 若 byType 和 byName 的方式都无妨实现自动装配，即 IOC 容器中有多个类型匹配的 bean。 且这些 bean 的 id 和要赋值的属性的属性名都不一致，此时抛异常：NoUniqueBeanDefinitionException。
d> 此时可以在要赋值的属性上，添加一个注解 @Qualifier。通过该注解的 value 属性值，指定某个 bean 的 id，将这个 bean 为属性赋值。

·注意：若 IOC 容器中没有任何一个类型匹配的 bean，此时抛出异常：NoSuchBeanDefinitionException。在 @Autowired 注解中有个属性 required，默认值为 true，要求必须完成自动装配，可以将 required 设置为 false，此时能装配则装配，无法装配则使用属性的默认值

@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

UserController

```
@Controller("controller")
public class UserController {
    @Autowired
    @Qualifier("userServiceImpl")
    private UserService userService;
    /*
    public UserController(UserService userService) {
        this.userService = userService; }
     */

//      @Autowired
    /*
    public void setUserService(UserService userService) {
        this.userService = userService; }
     */
    public void saveUser() {
        userService.saveUser();
    }
}
```

UserService 接口

```
public interface UserService {
    // 保存用户信息
    void saveUser();
}
```

UserServiceImpl

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    @Qualifier("userDaoImpl")
    private UserDao userDao;
    @Override
    public void saveUser() {
        userDao.saveUser();
    }
}
```


UserDao 接口

```
public interface UserDao {
    // 保存用户信息
    void saveUser();
}
```

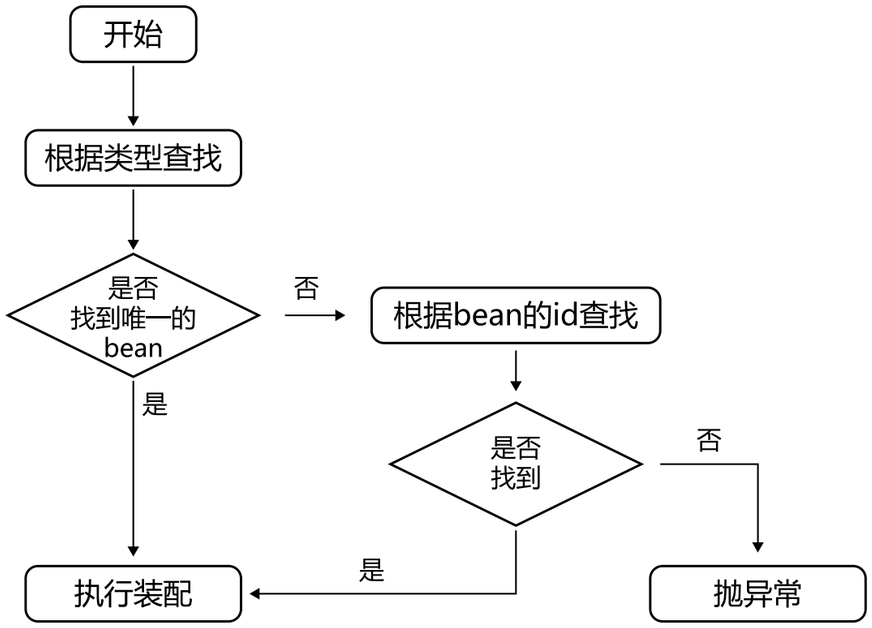
UserDaoImpl

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

IOCByAnnotationTest

```
public class IOCByAnnotationTest {
    @Test
    public void test() {
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("spring-ioc-annotation.xml");
        UserController userController = ioc.getBean("controller",
UserController.class);
        /*System.out.println(userController);
        UserService userService = ioc.getBean("userServiceImpl",
UserService.class);
        System.out.println(userService);
        UserDao userDao = ioc.getBean("userDaoImpl", UserDao.
class);
        System.out.println(userDao);*/
        userController.saveUser();
    }
}
```

@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。

三 APO

3.1 场景模拟

声明接口

```
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);
}
```

创建实现类

```
public class CalculatorImpl 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;
    }
}
```

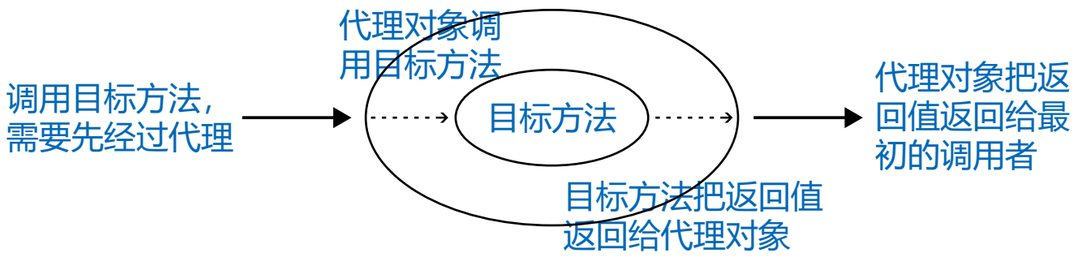
3.2 代理模式

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。

静态代理

创建静态代理类：

```
public class CalculatorStaticProxy implements Calculator {
    private CalculatorImpl target;
    public CalculatorStaticProxy(CalculatorImpl target) {
        this.target = target;
    }

    @Override
    public int add(int i, int j) {
        int result = 0;
        try {
            System.out.println(" 日志，方法：add，参数：" + i + "," + j);
            result = target.add(i, j);
            System.out.println(" 日 志， 方 法：add， 结 果：" + result);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
        }
        return result;
    }

    @Override
    public int sub(int i, int j) {
        System.out.println(" 日志，方法：sub，参数：" + i + "," + j);
        int result = target.add(i, j);
        System.out.println(" 日志，方法：sub，结果：" + result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        System.out.println(" 日志，方法：mul，参数：" + i + "," + j);
        int result = target.add(i, j);
        System.out.println(" 日志，方法：mul，结果：" + result);
        return result;
    }
```

```
    @Override
    public int div(int i, int j) {
        System.out.println(" 日志，方法：div，参数：" + i + "," + j);
        int result = target.add(i, j);
        System.out.println(" 日志，方法：div，结果：" + result);
        return result;
    }
}
```

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。

动态代理

生产代理对象的工厂类：

newProxyInstance()：创建一个代理实例
其中有三个参数：
ClassLoader loader : 指定加载动态生成的代理类的类加载器
Class[] interfaces : 获取目标对象实现的所有接口的 class 对象的数组
InvocationHandler h : 设置代理类中的抽象方法如何重写

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

```
public class ProxyFactory {
    private Object target;

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

    public Object getProxy() {
        ClassLoader classLoader = this.getClass().getClassLoader();
        Class<?>[] interfaces = target.getClass().getInterfaces();
        InvocationHandler h = new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                Object result = null;
                try {
                    System.out.println(" 日志，方法：" +
method.getName() + "，参数：" + Arrays.toString(args));
                    // proxy 表示代理对象，method 表示要执行的
方法，args 表示要执行的方法到的参数列表
                    result = method.invoke(target, args);
                    System.out.println(" 日志，方法：" +
method.getName() + "，结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                    System.out.println(" 日志，方法：" +
method.getName() + "，异常：" + e);
```

```
                } finally {
                    System.out.println(" 日志，方法：" +
method.getName() + "，方法执行完毕 ");
                }
                return result;
            }
        };
        return Proxy.newProxyInstance(classLoader, interfaces, h);
    }
}
```

测试

动态代理有两种：

1、jdk 动态代理，要求必须有接口，最终生成的代理类和目标类实现相同的接口在 com.sun.proxy 包下 , 类名为 $proxy2。

2、cglib 动态代理，最终生成的代理类会继承目标类，并且和目标类在相同的包下。

```
public class ProxyTest {
    @Test
    public void testProxy() {
        /*
        CalculatorStaticProxy proxy = new CalculatorStaticProxy
(new CalculatorImpl()); proxy.add(1, 2);
         */
        ProxyFactory proxyFactory = new ProxyFactory(new
CalculatorImpl());
        Calculator proxy = (Calculator) proxyFactory.getProxy();
        proxy.add(1, 2);

    }
}
```


3.3 AOP 概念及相关术语

概述

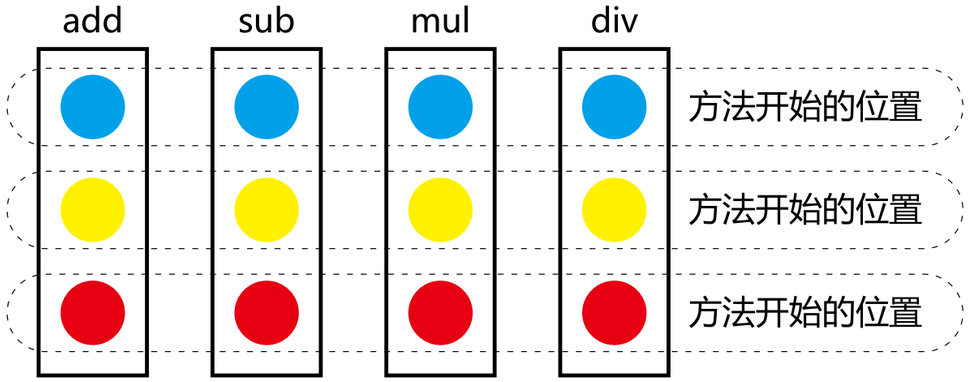
AOP（Aspect Oriented Programming）是一种设计思想，是软件设计领域中的面向切面编程，它是面向对象编程的一种补充和完善，它以通过预编译方式和运行期动态代理方式实现在不修改源代码的情况下给程序动态统一添加额外功能的一种技术。

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。

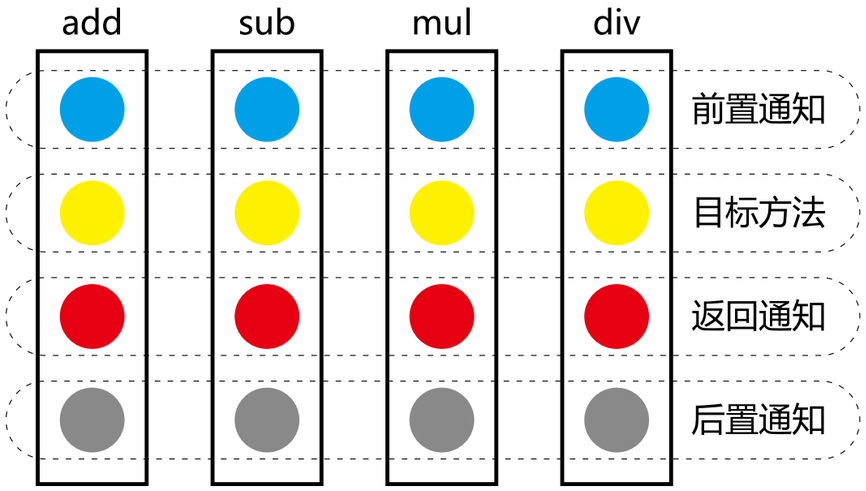
这个概念不是语法层面天然存在的，而是根据附加功能的逻辑上的需要：有十个附加功能，就有十个横切关注点。



②通知

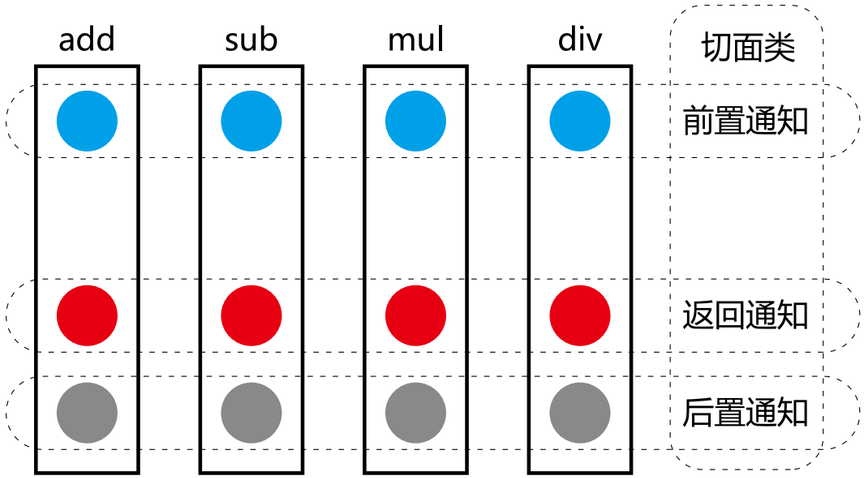
每一个横切关注点上要做的事情都需要写一个方法来实现，这样的方法就叫通知方法。

- 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。

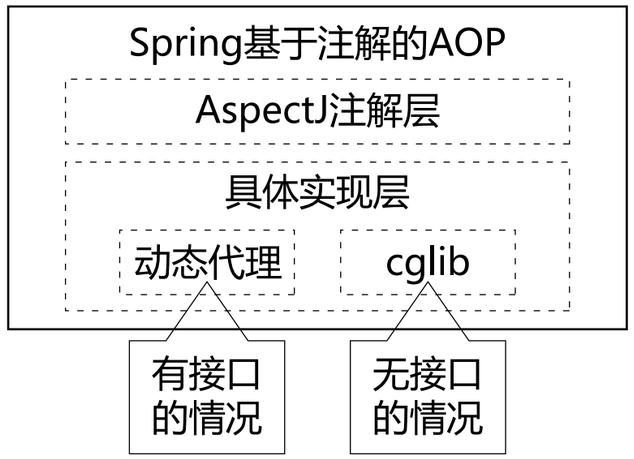
作用

· 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

技术说明



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：

```
<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>
    <!-- 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>
</dependencies>
```


②准备被代理的目标资源

Calculator 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);
}
```

CalculatorPureImpl

```
public class CalculatorImpl 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;
    }
}
```

创建切面类并配置

1、在切面中，需要通过指定的注解将方法标识为通知方法
@Before：前置通知，在目标对象方法执行之前执行
@After：后置通知，在目标对象方法的 finally 字句中执行
@AfterReturning：返回通知，在目标对象方法返回值之后执行
@AfterThrowing：异常通知，在目标对象方法的 catch 字句中执行

2、切入点表达式：设置在标识通知的注解的 value 属性中
@Before("execution(public int spring.aop.annotation.CalculatorImpl.add(int, int))")
@Before("execution(* spring.aop.annotation.*.*(..))")
第一个 * 表示任意的访问修饰符和返回值类型
第二个 * 表示类中任意的方法
.. 表示任意的参数列表

3、重用切入点表达式
//@pointcut 声明一个公共的切入点表达式
@pointcut("execution(* spring.aop.annotation.CalculatorImpl.(..))")
public void pointcut(){}
使用方式：@Before(“ pointcut() “)

4、获取连接点的信息
在通知方法的参数位置，设置 Joinpoint 类型的参数，就可以获取连接点所对应方法的信息
// 获取连接点所对应方法的签名信息
Signature signature = joinPoint.getSignature();
// 获取连接点所对应方法的参数
object[] args = joinPoint.getArgs();

5、切面的优先级
可以通过 @Order 注解的 value 属性设置优先级，默认值 Integer 的最大值
@Order 注解的 value 属性值越小，优先级越高


```
@Component
@Aspect // 将当前组件标识为切面
public class LoggerAspect {

    @Pointcut("execution(* spring.aop.annotation.CalculatorImpl.*(..))")
    public void pointCut() {
    }

    // @Before("execution(public int spring.aop.annotation.CalculatorImpl.add(int,
    // int))")
    // @Before("execution(* spring.aop.annotation.CalculatorImpl.*(..))")
    @Before("pointCut()")
    public void beforeAdviceMethod(JoinPoint joinPoint) {
        // 获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
        // 获取连接点所对应方法的参数
        Object[] args = joinPoint.getArgs();
        System.out.println("LoggerAspect, 方法：" + signature.getName() + "，参数：" + Arrays.toString(args));
    }

    @After("pointCut()")
    public void afterAdviceMethod(JoinPoint joinPoint) {
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect,  方  法：" + signature.getName() + "，执行完毕 ");
    }

    /*
     * 在返回通知中若要获取目标对象方法的返回值 只需要通过 @AfterReturning 注解的 returning 属性，就可以将通知方法的某个参数指定为接收目标对象方法的返回值的参数
     */
    @AfterReturning(value = "pointCut()", returning = "result")
    public void afterReturningAdviceMethod(JoinPoint joinPoint, Object result) {
        Signature signature = joinPoint.getSignature();
```

```
        System.out.println("LoggerAspect, 方法：" + signature.
getName() + "，结果：" + result);
    }

    /*
     * 在异常通知中若要获取目标对象方法的异常 只需要通过
AfterThrowing 注解的 throwing 属性，就可以将通知方法的某个参数指定
为接收目标对象方法出现的异常的参数
     */
    @AfterThrowing(value = "pointCut()", throwing = "ex")
    public void afterThrowingAdviceMethod(JoinPoint joinPoint,
Throwable ex) {
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect,  方  法：" + signature.
getName() + "，异常：" + ex);
    }

    // 环绕通知的方法的返回值一定要和目标对象方法的返回值一致
    @Around("pointCut()")
    public Object aroundAdviceMethod(ProceedingJoinPoint
joinPoint) {
        Object result = null;
        try {
            System.out.println(" 环绕通知 --> 前置通知 ");
            // 表示目标对象方法的执行
            result = joinPoint.proceed();
            System.out.println(" 环绕通知 --> 返回通知 ");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
            System.out.println(" 环绕通知 --> 异常通知 ");
        } finally {
            System.out.println(" 环绕通知 --> 后置通知 ");
        }
        return result;
    }

}
```


aop-annotation.xml

```
<?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"
    xmlns:aop="http://www.springframework.org/schema/aop"
    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 http://www.springframework.org/schema/aop https://www.
springframework.org/schema/aop/spring-aop.xsd">

    <!--
        AOP 的注意事项：
        切面类和目标类都需要交给 IOC 容器管理
        切面类必须通过 @Aspect 注解标识为一个切面
        在 spring 的配置文件中设置 <aop:aspectj-autoproxy /> 开启
基于注解的 Aop
    -->
    <context:component-scan base-package="spring.aop.
annotation"></context:component-scan>
    <!-- 开启基于注解的 AOP-->
    <aop:aspectj-autoproxy />
</beans>
```

AOPByAnnotationTest

```
public class AOPByAnnotationTest {
    @Test
    public void testAOPByAnnotation() {
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("aop-annotation.xml");
        Calculator calculator = ioc.getBean(Calculator.class);
        calculator.div(10, 1);
    }
}
```

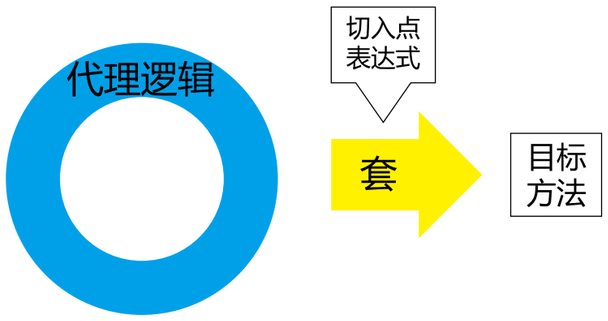
各种通知

· 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

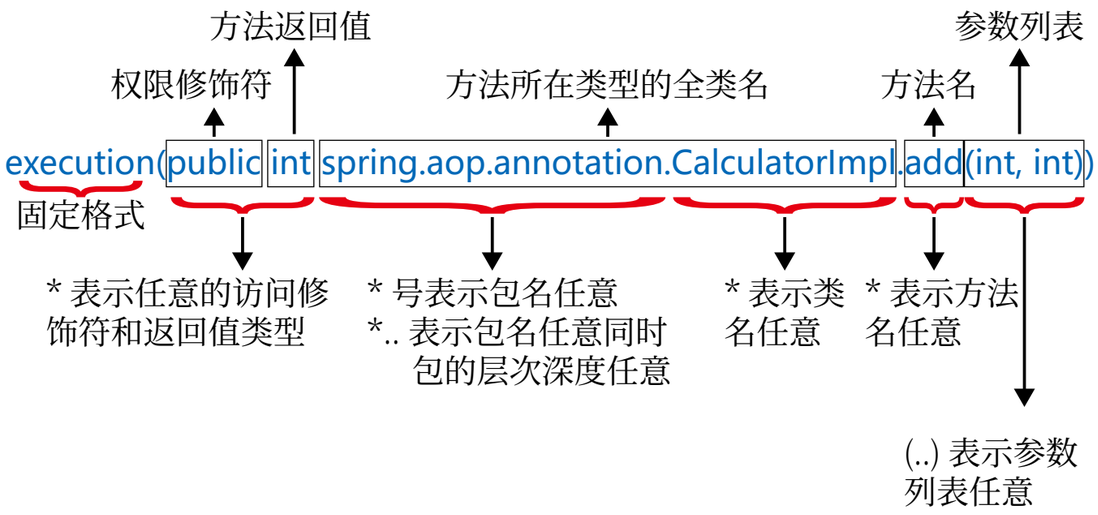
各种通知的执行顺序：
· Spring 版本 5.3.x 以前：
· 前置通知
· 目标操作
· 后置通知
· 返回通知或异常通知
Spring 版本 5.3.x 以后：
· 前置通知
· 目标操作
· 返回通知或异常通知
· 后置通知

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。



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）

准备工作

参考基于注解的 AOP 环境

aop-xml.xml

```
<?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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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 http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">
    <!-- 扫描组件 -->
    <context:component-scan base-package="spring.aop.xml"></context:component-scan>
    <aop:config>
        <!-- 设置一个公共的切入点表达式 -->
        <aop:pointcut id="pointCut" expression="execution(* spring.aop.xml.CalculatorImpl.*(..))" />
        <!-- 将 IOC 容器中的某个 bean 设置为切面 -->
        <aop:aspect ref="loggerAspect">
            <aop:before method="beforeAdviceMethod" pointcut-ref="pointCut"></aop:before>
            <aop:after method="afterAdviceMethod" pointcut-ref="pointCut"></aop:after>
            <aop:after-returning method="afterReturningAdviceMethod" returning="result" pointcut-ref="pointCut"></aop:after-returning>
            <aop:after-throwing method="afterThrowingAdviceMethod" throwing="ex" pointcut-ref="pointCut"></aop:after-throwing>
            <aop:around method="aroundAdviceMethod" pointcut-ref="pointCut"></aop:around>
```

```
            </aop:aspect>

            <aop:aspect ref="validateAspect" order="1">
                <aop:before method="beforeMethod" pointcut-
ref="pointCut"></aop:before>

            </aop:aspect>
        </aop:config>
</beans>
```

AOPXmlTest

```
public class AOPXmlTest {
    @Test
    public void testAOPByXml(){
        ApplicationContext ioc = new ClassPathXmlApplicationCon
text("aop-xml.xml");
        Calculator calculator = ioc.getBean(Calculator.class);
        calculator.add(1,1);
    }
}
```


四　声明式事务

4.1 JdbcTemplate

简介

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

准备工作

①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</versio n>
    </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.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.Driver=com.mysql.cj.jdbc.Driver
jdbc.username=root
jdbc.password=abc123
```

③配置 Spring 的配置文件

```
<?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">

    <!-- 引入 jdbc.properties-->
    <context:property-placeholder location="jdbc.properties"></
context:property-placeholder>
    <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>
    <bean class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></
property>
    </bean>
</beans>
```

测试

JdbcTemplateTest

```
// 指定当前测试类在 Spring 的测试环境中执行，此时就可以通过注入的方
式直接获取 IOc 容器中 bean
@RunWith(SpringJUnit4ClassRunner.class)
// 设置 spring 测试环境的配置文件
@ContextConfiguration("classpath:spring-jdbc.xml")
public class JdbcTemplateTest {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Test
    public void testInsert() {
        String sql = "insert into t_user values(null, ?, ?, ?, ?, ?)";
        jdbcTemplate.update(sql, "root", "123", 23, "女", "123@
qq.com");
    }

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

4.2 声明式事务概念

编程式事务

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。

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。

tx-xml.xml

```
<?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"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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 http://
www.alibaba.com/schema/stat http://www.alibaba.com/schema/
stat.xsd http://www.springframework.org/schema/aop https://www.
springframework.org/schema/aop/spring-aop.xsd">
```



```
       <!-- 扫描组件 -->
       <context:component-scan base-package="spring"></
context:component-scan>
       <!-- 引入 jdbc.properties-->
       <context:property-placeholder location="jdbc.properties"></
context:property-placeholder>
       <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>
       <bean class="org.springframework.jdbc.core.JdbcTemplate">
              <property name="dataSource" ref="dataSource">
property>
       </bean>
       <!-- 配置事务管理器 -->
       <bean id="transactionManager" class="org.springframework.
jdbc.datasource.DataSourceTransactionManager">
              <property name="dataSource" ref="dataSource">
property>
       </bean>
       <!-- 配置事物通知 -->
       <tx:advice id="tx" transaction-
manager="transactionManager">
              <tx:attributes>
                     <tx:method name="buyBook" />
                     <tx:method name="*" />
              </tx:attributes>
       </tx:advice>
       <aop:config>
              <aop:advisor advice-ref="tx" pointcut="execution(* spring.
Service.impl.*.*(..))"></aop:advisor>
       </aop:config>
</beans>
```

CheckoutService 接口

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

CheckoutService 接口

```
public class CheckoutServiceImpl implements CheckoutService {
	@Autowired
	private BookService bookService;

	@Override
	@Transactional
	public void checkout(Integer userId, Integer[] bookIds) {
		for (Integer bookId : bookIds) {
			bookService.buyBook(userId, bookId);
		}
	}
}
```

TxByAnnotationTest

声明式事务的配置步骤：

1、在 spring 的配置文件中配置事务管理器

2、开启事务的注解驱动

在需要被事务管理的方法上，添加 @Transactional 注解，该方法就会被事务管理

@Transactional 注解标识的位置：

1、标识在方法上

2、标识在类上，则类中所有的方法都会被事务营理


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

	@Autowired
	private BookController bookController;

	@Test
	public void testBuyBook() {
//		bookController.buyBook(1, 1);
		bookController.Checkout(1, new Integer[] { 1, 2 });

	}
}
```

TxByAnnotationTest

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

	@Autowired
	private BookController bookController;

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

	}
}
```

SpringMVC

一 SpringMVC 简介

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

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。

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。

二　入门案例

创建 maven 工程

①添加 web 模块
②打包方式：war
③引入依赖

```
<packaging>war</packaging>
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.18</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。

配置 web.xml

注册 SpringMVC 的前端控制器 DispatcherServlet

配置 springMVC 的前端控制器 DispatcherServLet
SpringMVC 的配置文件默认的位置和名称：
位置：WEB-INF 下
名称：<servlet-name>-servlet.xml, 当前配置下的配置文件名为 SpringMVC-servlet.xml
url-pattern 中 / 和 /* 的区别：
/：匹配浏览器向服务器发送的所有请求（不包括 .jsp）
/*：匹配浏览器向服务器发送的所有请求（包括 .jsp）

```
<servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.
DispatcherServlet</servlet-class>
    <!-- 设置 SpringMVc 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>
    <!-- 将 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>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

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。

HelloController

```
@Controller
public class HelloController {
    @RequestMapping("/")
    public String protal() {
        // 将逻辑视图返回
        return "index";
    }

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


创建 SpringMVC 的配置文件

```
<!-- 扫描控制层组件 -->
<context:component-scan base-package="controller"></
context:component-scan>
<!-- 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>
```

测试 HelloWorld

通过超链接跳转到指定页面
在主页 index.html 中设置超链接

index.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title> 首页 </title>
    </head>
    <body>
        <h1>index.html</h1>
        <a th:href="@{/hello}"> 测试 SpringMVC</a>
        <a href="/hello"> 测试绝对路径 </a>
    </body>
</html>
```

success.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title>Title</title>
    </head>
    <body>
        <h1>success.html</h1>
    </body>
</html>
```

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。

三 @RequestMapping 注解

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。

2、@RequestMapping 注解标识的位置

ORequestMapping 标识一个类：设置映射请求的请求路径的初始信息；
RequestMaping 标识一个方法：设置映射请求请求路径的具体信息。

3、@RequestMapping 注解 value 属性

作用：通过请求的请求路径匹配请求；

value 属性是数组类型，即当前浏览器所发送请求的请求路径匹配 value 属性中的任何一个值，则当前请求就会被注解所标识的方法进行处理。

4、@RequestMapping 注解的 method 属性

作用：通过请求的请求方式匹配请求；

method 属性是 RequestMethod 类型的数组，即当前浏览器所发送请求的请求方式匹配 method 属性中的任何一种请求方式，则当前请求就会被注解所标识的方法进行处理；

若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求方式不匹配，此时页面报错：405 - Request method 'xxx' not supported，在 @RequestMapping 的基础上，结合请求方式的一些派生注解：
@GetMapping,@PostMapping,@DeleteMapping,@PutMapping。

5、@RequestMapping 注解的 params 属性

作用：通过请求的请求参数匹配请求，即浏览器发送的请求的请求参数必须满足 params 属性到的设置 params 可以使用四种表达式：

"param"：表示当前所匹配请求的请求参数中必须携带 param 参数；

"!param"：表示当前所匹配请求的请求参数中一定不能携带 param 参数；

"param=value"：表示当前所匹配请求的请求参数中必须携带 param 参数且值必须为 value；

"param!=value"：表示当前所匹配请求的请求参数中可以不携带 param，若携带值一定不能是 value；

若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求参数不匹配。此时页面报错：400 - parameter conditions “ username" not met for actual request parameters:

6、@RequestMapping 注解的 headers 属性

作用：通过请求的请求头信息匹配请求，即浏览器发送的请求的请求头信息必须满足 headers 属性的设置，若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求头信息不匹配，

此时页面报错：400。

7、SpringMVC 支持 ant 风格的路径

在 @RequestMapping 注解的 value 属性值中设置一些特殊字符；

？：任意的单个字符（不包括?）；

*：任意个数的任意字符（不包括？和 /）；

**：任意层数的任意目录，注意使用方式只能 ** 写在双斜线中，前后不能有任何的其他字符。

8、@RequestMapping 注解使用路径中的占位符

传统：/deleteUser?id=1

rest: /user/delete/1

需要在 @RequestMapping 注解的 value 属性中所设置的路径中，使用 {xxx} 的方式表示路径中的数据；

在通过 @Pathvariable 注解，将占位符所标识的值和控制器方法的形参进行绑定。

ProatCollection

```
@Controller
public class ProatCollection {
    @RequestMapping("/")
    public String protal() {
        return "index";
    }
}
```


index.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title> 首页 </title>
    </head>
    <body>
        <h1>index.html</h1>
        <a th:href="@{/hello}"> 测试 @ReqbestMapping 注解所标识的位置 </a><br>
        <a th:href="@{/abc}"> 测试 @ReqbestMapping 注解 value 属性 </a>
        <form th:action="@{/hello}" method="post">
            <input type="submit" value="测试@ReqbestMapping 注解的 method 属性 ">
        </form>
        <a th:href="@{/hello?username=admin}"> 测试 @ReqbestMapping 注解 params 属性 </a><br>
        <a th:href="@{/hello(username='admin')}"> 测试 @ReqbestMapping 注解 params 属性 </a><br>
        <a th:href="@{/aaa/test/ant}"> 测试 @ReqbestMapping 注解支持 ant 风格的路径 </a><br>
        <a th:href="@{/test/rest/admin/1}"> 测试 @ReqbestMapping 注解支持 value 属性中的占位符 </a><br>
    </body>
</html>
```

success.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title> 首页 </title>
    </head>
    <body>
        <h1>success.html</h1>
    </body>
</html>
```

index.xml

```
@Controller
//@RequestMapping("/test")
public class TestRequestMappingController {
	// 此时控制器方法所匹配的请求的请求路径为 /test/hello
	@RequestMapping(
		value = { "/hello", "/abc" },
		method = { RequestMethod.POST, RequestMethod.GET },
//	params = {"username", "!password", "age=20", "gender!= 女 "}
		headers = { "referer" })

	public String hello() {
		return "success";
	}

	@RequestMapping("a?a/test/ant")
	public String testAne() {
		return "success";
	}

	@RequestMapping("/test/rest/{username}/{id}")
	public String testRest(
		@PathVariable("id") Integer id,
		@PathVariable("username") String username)
	{
		System.out.println("id" + id + "username" + username);
		return "success";
	}
}
```


四　SpringMVC 获取请求参数

获取请求参数的方式：

1、通过 servletAPI 获取

只需要在控制器方法的形参位置设置 HttpservletRequest 类型的形参；就可以在控制器方法中使用 request 对象获取请求参数。

2、通过控制器方法的形参获取

只需要在控制器方法的形参位置，设置一个形参，形参的名字和请求参数的名字一致即可。

3、@RequestParam：将请求参数和控制器方法的形参绑定

@RequestParam 注解的三个属性：value,required,defaultValue；

value：设置和形参绑定的请求参数的名字；

required: 设置是否必须传输 val ue 所对应的请求参数默认值为 true，表示 value 所对应的请求参数必须传输，否则页面报错：

400 - Required String parameter 'xxx' is not present；

若设置为 false，则表示 value 所对应的请求参数不是必须传输，若为传输，则形参值为 null。

defaultValue：设置当没有传输 value 所对应的请求参数时，为形参设置的默认值，此时和 required 属性值无关。

4、@RequestHeader：将请求头信息和控制器方法的形参绑定。

5、@CookieValue：将 cookie 数据和控制器方法的形参绑定。

6、通过控制器方法的实体类类型的形参获取请求参数

需要在控制方法的形参位置设置实体类类型的形参，要保证实体类中的属性的属性名和请求参数的名字一致；

可以通过实体类类型的形参获取请求参数。

7、解决获取请求此参数的乱码问题

在 web.xml 中配置 spring 的编码过滤器 characterEncodingFilter。

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。

TestParamController

```
@Controller
public class TestParamController {
	@RequestMapping("/param/servletAPI")
	public String getParamByServletAPI(HttpServletRequest request)
{
		HttpSession session = request.getSession();
		String username = request.getParameter("username");
		String password = request.getParameter("password");
		System.out.println("username" + username + ",password"
+ password);
		return "success";
	}

	@RequestMapping("/param")
	public String getParam(
		@RequestParam(
			value = "username",
			required = false,
			defaultValue = "hello"
	) String username, String password,
		@RequestHeader("referer") String referer,
		@CookieValue("JSESSIONID") String jsessionId)
	{
		System.out.println("jsessionId" + jsessionId);
		System.out.println("referer" + referer);
		System.out.println("username" + username + ",password"
+ password);
		return "success";
	}

	@RequestMapping("/param/pojo")
	public String getParamByPojo(User user) {
		return "success";
	}
}
```


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>
```

User

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

	public User() {
	}

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

	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;
    }

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


五　域对象共享数据

向域对象共享数据：
1. 通过 ModeLAndView 向请求域共享数据
　　使用 ModeLAndView 时，可以使用其 ModeL 功能向请求域共享数据
　　使用 View 功能设置逻辑视图，但是控制器方法一定要将
ModeLAndView 作为方法的返回值。
2、使用 Model 向请求域共享数据。
3、使用 ModeLMap 向请求域共享数据。
4、使用 map 向请求域共享数据。
5,Model 和 ModelMap 和 map 的关系
　　其实在底层中，这些类型的形参最终都是通过 BindingAwareModeLMap
创建；
　　public class BindingAwareModelMap extends ExtendedModelMap；
　　public class ExtendedModelMap extends ModelMap implements Model {}；
　　public class ModelMap extends LinkedHashMap<String, Object> {}。

ModelAndView 包含 Model 和 View 的功能
　　Model：向请求域中共享数据
　　VieN：设置逻辑视图实现页面跳转

index.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 首页 </title>
</head>
<body>
<h1>index.html</h1>
<a th:href="@{/hello}"> 测试 @ReqbestMapping 注解所标识的位置 </
a><br>
<a th:href="@{/abc}"> 测试 @ReqbestMapping 注解 value 属性 </a>
<form th:action="@{/hello}" method="post">
    <input type="submit" value=" 测试 @ReqbestMapping 注解的
method 属性 ">
</form>
<a th:href="@{/hello?username=admin}"> 测试 @ReqbestMapping
注解 params 属性 </a><br>
```

```
<a th:href="@{/hello(username='admin')}"> 测试 @ReqbestMapping 注解 params 属性 </a><br>
<a th:href="@{/aaa/test/ant}"> 测试 @ReqbestMapping 注解支持 ant 风格的路径 </a><br>
<a th:href="@{/test/rest/admin/1}"> 测试 @ReqbestMapping 注解支持 value 属性中的占位符 </a><br>
<hr>
<form th:href="@{/param/pojo}" method="post">
    用户名：<input type="text" name="username"><br>
    密码：<input type="password" name="password"><br>
    <input type="submit" value=" 登录 "><br>
</form>
<hr>
<a th:href="@{/test/mav}"> 测试通过 ModelAndView 向请求域共享数据 </a><br>
<a th:href="@{/test/model}"> 测试通过 Model 向请求域共享数据 </a><br>
<a th:href="@{/test/modelMap}"> 测试通过 ModelMap 向请求域共享数据 </a><br>
<a th:href="@{/test/map}"> 测试通过 Map 向请求域共享数据 </a><br>
<a th:href="@{/test/session}"> 测试向会话域共享数据 </a><br>
<a th:href="@{/test/application}"> 测试向应用域共享数据 </a><br>
</body>
</html>
```

success.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 首页 </title>
</head>
<body>
<h1>success.html</h1>
<p th:text="${testRequestScope}"></p>
<p th:text="${session.testSessionScope}"></p>
<p th:text="${application.testApplicationScope}"></p>
</body>
</html>
```


index.xml

```
@Controller
public class TestScopeController {
    @RequestMapping("/test/mav")
    public ModelAndView testMAV() {
        ModelAndView mav = new ModelAndView();
        // 向请求域中共享数据
        mav.addObject("testRequestScope", "hello, ModelAndView");
        // 设置逻辑视图
        mav.setViewName("success");
        return mav;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model) {
        System.out.println(model.getClass().getName());
        model.addAttribute("testModelScope", "hello, Model");
        return "success";
    }

    @RequestMapping("/test/modelMap")
    public String testModelMap(ModelMap modelMap) {
        System.out.println(modelMap.getClass().getName());
        modelMap.addAttribute("testModelMapScope", "hello, ModelMap");
        return "success";
    }

    @RequestMapping("/test/moap")
    public String testMap(Map<String, Object> map) {
        System.out.println(map.getClass().getName());
        map.put("testMapScope", "hello, map");
        return "success";
    }

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

```
        return "success";
    }

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


六 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。

TestViewController

```
@Controller
public class TestViewController {
    @RequestMapping("/test/view/thymeleaf")
    public String testThymeleafView() {
        return "success";
    }

    @RequestMapping("/test/view/forward")
    public String testInternalResourceView() {
        return "forward:/test/model";
    }

    @RequestMapping("/test/view/redirect")
    public String testRedirectView() {
        return "redirect:/test/model";
    }
}
```

index.xml

```
<hr>
<a th:href="@{/test/view/thymeleaf}"> 测试 SpringMVC 的视图
ThymeleafView</a><br>
<a th:href="@{/test/view/forward}"> 测试 SpringMVC 的视图
InternalResourceView</a><br>
<a th:href="@{/test/view/redirect}"> 测试 SpringMVC 的视图
InternalResourceView</a><br>
```

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。

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"。

转发视图

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"。

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。

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 viewcontroller 标签进行表示。

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 />

七 RESTful

RESTful 简介

REST：Representational State Transfer，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。

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。

查询所有的用户信息 - - >/user - - - >get
根据 id 查询用户信息 - - >/user/1 - - >get
添加用户信息 - - >/user - - >post
修改用户信息 - - >/user - - >put
删除用户信息 - - >/user/1 - - >delete

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。

web.xml

```
<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>
```


index.xml

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 首页 </title>
</head>
<body>
<h1>index.html</h1>
<a th:href="@{/user}"> 查询所有用户信息 </a>
<a th:href="@{/user/1}"> 根据 id 查询用户信息 </a>
<form th:action="@{/user}" method="post">
    <input type="submit" value=" 添加用户信息 ">
</form>
<form th:action="@{/user}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="submit" value=" 修改用户信息 ">
</form>
<form th:action="@{/user/1}" method="post">
    <input type="hidden" name="_method" value="delete">
    <input type="submit" value=" 删除用户信息 ">
</form>
</body>
</html>
```

注意：浏览器目前只能发送 get 和 post 请求

若要发送 put 和 delete 请求，需要在 web.xml 中配置一个过滤器 HiddenHttpMethodFilter。

配置了过滤器之后，发送的请求要满足两个条件，才能将请求方式转换为 put 或 delete：

1、当前请求的请求方式必须为 post；
2、当前请求必须传输请求参数 _method,_method 的值才是最终的请求方式。

TestController

```
@Controller
public class TestController {
	@RequestMapping(value = "/user", method = RequestMethod.
GET)
	public String getAllUser() {
		System.out.println(" 查询所有的用户信息 -->/user--->get");
		return "success";
	}

	@RequestMapping(value = "/user/1", method =
RequestMethod.GET)
	public String getUserById(@PathVariable("id") Integer id) {
		System.out.println(" 根据 id 查询用户信息 -->/user/" + id +
"-->get");
		return "success";
	}

	@RequestMapping(value = "/user", method = RequestMethod.
POST)
	public String insertUser() {
		System.out.println(" 添加用户信息 -->/user-->post");
		return "success";
	}

	@RequestMapping(value = "/user", method = RequestMethod.
PUT)
	public String updateUser() {
		System.out.println(" 修改用户信息 -->/user-->put");
		return "success";
	}

	@RequestMapping(value = "/user/{1}", method =
RequestMethod.DELETE)
	public String deleteUser(@PathVariable("id") Integer id) {
		System.out.println(" 删除用户信息 -->/user/" + id + "--
>post");
		return "success";
	}
}
```


八 RESTful 案例

Employee

```
public class Employee {

	private Integer id;
	private String lastName;
	private String email;
	private Integer gender;

	public Employee() {
	}

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

@Override
public String toString() {
        return "Employee{" + "id=" + id + ", lastName='" +
lastName + '\'' + ", email='" + email + '\'' + ", gender="
                        + gender + '}';
        }
}
```


EmployeeDao 准备 dao 模拟数据

```
@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);
        }
}
```

配置默认的 servlet 处理静态资源

· 当前工程的 web.xml 配置的前端控制器 DispatcherServlet 的 url-pattern 是 /。

· tomcat 的 web.xml 配置的 DefaultServlet 的 url-pattern 也是 /。

· 此时，浏览器发送的请求会优先被 DispatcherServlet 进行处理，但是 DispatcherServlet 无法处理静态资源。

· 若配置了 <mvc:default-servlet-handler />，此时浏览器发送的所有请求都会被 DefaultServlet 处理。

· 若配置了 <mvc:default-servlet-handler/> 和 <mvc:annotation-driven />。

· 浏览器发送的请求会先被 DispatcherServlet 处理，无法处理在交给 DefaultServlet 处理。

```
    <mvc:default-servlet-handler />

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

    <!-- 配置试图控制器 -->
    <mvc:view-controller path="/" view-name="index"></
mvc:view-controller>
    <mvc:view-controller path="/to/add" view-name="employee-
add"></mvc:view-controller>
```


Employee-list.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title>employee list</title>
            <link rel="stylesheet" th:href="@{/static/css/index_
work.css}">
    </head>
    <body>

        <div id="app">
            <table>
                <tr>
                    <th colspan="5">employee list</th>
                </tr>
                <tr>
                    <th>id</th>
                    <th>lastName</th>
                    <th>email</th>
                    <th>gender</th>
                    <th>
                        options(
                        <a th:href="@{/to/add}">add</a>
                        )
                    </th>
                </tr>
                <tr th:each="employee : ${allEmployee}">
                    <td th:text="${employee.id}"></td>
                    <td th:text="${employee.lastName}"></td>
                    <td th:text="${employee.email}"></td>
                    <td th:text="${employee.gender}"></td>
                    <td>
                        <a @click="deleteEmployee()"
th:href="@{'/employee/' + ${employee.id}}">delete</a>
                        <a th:href="@{'/employee/' +
${employee.id}}">update</a>
                    </td>
                </tr>
```

```
                </table>
            </div>
            <form method="post">
                <input type="hidden" name="_method"
value="delete">
            </form>

            <script type="text/javascript" th:src="@{/static/js/
vue,js}"></script>
            <script type="text/javascript">
            var vue = new Vue({
                el:"#app",
                methods:{
                    deleteEmployee(){
                        // 获取 form 表单
                        var form = document.
getElementsByTagName("form")[0]
                // 将超链接的 href 属性赋值给 form 表单的 action 属性
                //event.target 表示当前出发事件的标签
                        form.action = event.target.href;
                        // 表单提交
                        form.submit();
                        // 阻止超链接的默认行为
                        event.preventDefault();
                    }
                }
            });
    </script>
        </body>
    </html>
```


Employee-add.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title>add employee</title>
            <link rel="stylesheet" th:href="@{/static/css/index_
work.css}">
    </head>
    <body>
        <form th:action="@{/employee}" method="post">
            <table>
                <tr>
                    <th colspan="2">add employee</th>
                </tr>
                <tr>
                    <td>lastName</td>
                    <td>
                        <input type="text"
name="lastName">
                    </td>
                </tr>
                <tr>
                    <td>email</td>
                    <td>
                        <input type="text" name="email">
                    </td>
                </tr>
                <tr>
                    <td>gender</td>
                    <td>
                        <input type="redio" name="gender"
value="1">male
                            <input type="redio"
name="gender" value="0">female

                    </td>
                </tr>
                <tr>
```

```
                    <td colspan="2">
                        <input type="submit" value="add">
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Employee-update.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title>update employee</title>
            <link rel="stylesheet" th:href="@{/static/css/index_
work.css}">
    </head>
    <body>
        <form th:action="@{/employee}" method="post">
            <input type="hidden" name="_method"
value="put">
                <input type="hidden" name="id"
th:value="${employee.id}">
                    <table>
                        <tr>
                            <th colspan="2">add
employee</th>
                        </tr>
                        <tr>
                            <td>lastName</td>
                            <td>
                                <input type="text"
name="lastName" th:value="${employee.lastName}">
                            </td>
                        </tr>
                        <tr>
                            <td>email</td>
```



```
                            <td>
                                <input type="text"
name="email" th:value="${employee.email}">
                            </td>
                        </tr>
                        <tr>
                            <td>gender</td>
                            <td>
                                <input type="text"
name="gender" value="1" th:field="${employee.genderl}">male
                                <input type="text"
name="gender" value="0" th:field="${employee.genderl}">female

                            </td>
                        </tr>
                        <tr>
                            <td colspan="2">
                                <input type="submit"
name="update">
                            </td>
                        </tr>
                    </table>
        </form>
    </body>
</html>
```

Employee-add.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title>add employee</title>
            <link rel="stylesheet" th:href="@{/static/css/index_
work.css}">
    </head>
    <body>
        <form th:action="@{/employee}" method="post">
            <table>
                <tr>
                    <th colspan="2">add employee</th>
                </tr>
                <tr>
                    <td>lastName</td>
                    <td>
                        <input type="text"
name="lastName">
                    </td>
                </tr>
                <tr>
                    <td>email</td>
                    <td>
                        <input type="text" name="email">
                    </td>
                </tr>
                <tr>
                    <td>gender</td>
                    <td>
                        <input type="redio" name="gender"
value="1">male
                            <input type="redio"
name="gender" value="0">female

                    </td>
                </tr>
                <tr>
```


EmployeeController

查询所有的员工信息 - - >/employee - - >get
跳转到添加页面 - - >/to/add - - >get
新增员工信息 - - >/employee - - >post
跳转到修改页面 - - >/employee/1 - - >get
修改员工信息 - - >/employee - - >put
删除员工信息 - - >/employee/1 - - >delete

```
@Repository
@Controller
public class EmployeeController {
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method =
RequestMethod.GET)
    public String getAllEmployee(Model model) {
        // 获取所有的员工信息
        Collection<Employee> allEmployee = employeeDao.
getAll();
        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);
        // 跳转到列表页面
        return "employee-list";
    }

    @RequestMapping(value = "/employee", method =
RequestMethod.POST)
    public String addEmployee(Employee employee) {
        // 保存员工信息
        employeeDao.save(employee);
        // 重定向到列表功能：/employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "/employee/{id}", method =
RequestMethod.GET)
    public String toUpdate(@PathVariable("id") Integer id, Model
model) {
```

```
        // 根据 id 查询员工信息
        Employee employee = employeeDao.get(id);
        // 将员工信息共享到请求域中
        model.addAttribute("employee", employee);
        // 跳转到 employee-update.html
        return "employee-update";
    }

    @RequestMapping(value = "/employee", method =
RequestMethod.PUT)
    public String updateEmployee(Employee employee) {
        // 修改员工信息
        employeeDao.save(employee);
        // 重定向列表功能：/employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "/employee/{id}", method =
RequestMethod.DELETE)
    public String deleteEmployee(@PathVariable("id") Integer id) {
        // 删除员工信息
        employeeDao.delete(id);
        // 重定向到列表功能
        return "redirect:/employee";
    }
}
```


九 SpringMVC 处理 ajax 请求

1、@RequestBody：将请求体中的内容和控制器方法的形参进行绑定。
2、使用 @RequestBody 注解将 json 格式的请求参数转换为 java 对象
　　a> 导入 jackson 的依赖；
　　b> 在 springMvc 的配置文件中设置 <mvc:annotation-driven />；
　　c> 在处理请求的控制器方法的形参位置，直接设置 json 格式的请求参数要转换的 java 类型的形参使用 @RequestBody 注解标识即可。
3、@ResponseBody：将所标识的控制器方法的返回值作为响应报文的响应体响应到浏览器。
4、使用 @ResponseBody 注解响应浏览器 json 格式的数据
　　a> 导入 jackson 的依赖；
　　b> 在 springMVc 的配置文件中设置 <mvc:annotation-driven />；
　　c> 将需要转换为 json 字符串的 java 对象直接作为控制器方法的返回值，使用 @ResponseBody 注解标识控制器方法就可以将 java 对象直接转换为 json 字符串，并响应到浏览器。

　　常用的 Java 对象转换为 json 的结果：
　　实体类 - - >json 对象
　　map - - >json 对象
　　List - - >json 数组

@RestController == @Controller + @ResponseBody

```
axios({
    url:"", // 请求路径
    method: "", // 请求方式
    /* 以 name=value&name=value 的方式发送的请求参数，不管使用的请求方式是 get 或 post，请求参数都会被拼接到请求地址后，此种方式的请求参数可以通过 request.getParameter() 获取 */
    params:{},
    /* 以 json 格式发送的请求参数，请求参数会被保存到请求报文的请求体传输到服务器，此种方式的请求参数不可以通过 request.getParameter() 获取 */
    data:{}
    }).then(response=>{
        console.log(response.data)
    });
```

index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 首页 </title>
</head>
<body>
<div id="app">
    <h1>index.html</h1>
    <input type="button" value=" 测试 SpringMVC 处理 ajax" @
click="testAjax()"><br>
    <input type="button" value=" 使用 @RequestBody 注解处理 json
格式的请求参数 " @click="testRequestBody()"><br>
    <a th:href="@{/test/ResponseBody}"> 测试 @ResponseBody 注
解响应浏览器数据 </a><br>
    <input type="button" value=" 使用 @ResponseBody 注解处理
json 格式的数据 " @click="testResponseBody()"><br>
</div>

<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",
        method:{
            testAjax(){
                axios.post(
                    "/SpringMVC/test/ajax?id=1001",
                    {username:"admin", password:"abc123"}
                ).then(response=>{
                    console.log(response.data)
                });
            },

            textRequestBody(){
                axios.post(
                    "/SpringMVC/test/RequestBody.json",
                    {username:"admin", password:"abc123",
age:23, gender:" 男 "}
                ).then(response=>{
                    console.log(response.data);
                });
            },

            textResponseBody(){
                axios.post("/SpringMVC/test/ResponseBody/
json"
                ).then(response=>{
                    console.log(response.data)
                });
            }
        }
    });
</script>
</body>
</html>
```


pom.xml

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

User

```
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
```

TestAjaxController

```
@Controller
public class TestAjaxController {
	@RequestMapping("/test/ajax")
	public void testAjax(Integer id, @RequestBody String
requestBody, HttpServletResponse response) throws IOException {
		System.out.println("requestBody" + requestBody);
		System.out.println("id" + id);
		response.getWriter().write("hello, axios");
	}

	@RequestMapping("/test/RequestBody.json")
	public void testRequestBody(@RequestBody Map<String,
Object> map, HttpServletResponse response) throws IOException {
		System.out.println(map);
		response.getWriter().write("hello, RequestBody");
	}

	public void testRequestBody(@RequestBody User user,
HttpServletResponse response) throws IOException {
		System.out.println(user);
		response.getWriter().write("hello, RequestBody");
	}

	@RequestMapping("/test/RequestBody")
	@ResponseBody
	public String testResponseBody() {
		return "success";
	}

	/*@RequestMapping("/test/ResponseBody/json")
	@ResponseBody
	public Map<String, Object> testResponseBodyJson() {
		User user1 = new User(1001, "admin1", "abc123", 20, "男");
		User user2 = new User(1001, "admin2", "abc123", 20, "男");
		User user3 = new User(1001, "admin3", "abc123", 20, "男");
		Map<String, Object> map = new HashMap<>();
		map.put("1001", user1);
		map.put("1002", user2);
		map.put("1003", user3);
		return map;
	}*/

	@RequestMapping("/test/ResponseBody/json")
	@ResponseBody
	public List<User> testResponseBodyJson() {
		User user1 = new User(1001, "admin1", "abc123", 20, "男");
		User user2 = new User(1001, "admin2", "abc123", 20, "男");
		User user3 = new User(1001, "admin3", "abc123", 20, "男");
		List<User> list = Arrays.asList(user1, user2, user3);
		return list;
	}

}
```

十　文件上传和下载

pom.xml

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

springmvc.xml

```
<!-- 配置文件上传解析器 -->
    <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    </bean>
```

index.html

```
<a th:href="@{/test/down}"> 下载图片 </a>
<form th:action="@{/test/up}" method="post" enctype="multipart/form-data">
    头像：<input type="file" name="photo"><br>
    <input type="submit" value=" 上传 ">
</form>
```

ResponseEntity：可以作为控制器方法的返回值，表示响应到浏览器的完整的响应报文；

1、form 表单的请求方式必须为 post；

2、form 表单必须设置属性 enctype="multipart/form-data"。

FileUpAndDownController

```
@Controller
public class FileUpAndDownController {

    @RequestMapping("/test/up")
    public String testUp(MultipartFile photo, HttpSession session) throws IOException {
        // 获取上传文件的文件名
        String filename = photo.getOriginalFilename();
        // 获取上传文件的后缀名
        String hzName = filename.substring(filename.lastIndexOf("."));
        // 获取 UUID
        String uuid = UUID.randomUUID().toString();
        // 拼接一个新的文件名
        filename = uuid + hzName;
        // 获取 ServletContext 对象
        ServletContext servletContext = session.getServletContext();
        // 获取当前工程的真是路径
        String photoPath = servletContext.getRealPath("photo");
        // 创建 photoPath 所对应的 File 对象
        File file = new File(photoPath);
        // 判断 file 所对应目录是否存在
        if (!file.exists()) {
            file.mkdir();
        }
        String finalPath = photoPath + File.separator + filename;
        // 上传文件
        photo.transferTo(new File(finalPath));
        return "success";
    }

    @RequestMapping("/test/down")
    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("img");
            realPath = realPath + File.separator + "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;
        }
    }
```


十一　拦截器

springmvc.xml

```
<mvc:interceptors>
        <ref bean="firstInterceptor" />
        <ref bean="secondInterceptor" />
        <!--bean 和 ref 标签所配置的拦截器默认对 DispatcherServlet 处理
的所有的请求进行拦截 -->
        <!--<bean class="interceptor.FirstInterceptor"/>-->
        <!--<ref bean="firstInterceptor" />-->
        <!--<mvc:interceptor>
                &lt;!&ndash; 配置需要拦截的请求的请求路径，/** 表示所有请
求 &ndash;&gt;
                <mvc:mapping path="/**" />
                &lt;!&ndash; 配置需要排除拦截的请求的请求路径
&ndash;&gt;
                <mvc:exclude-mapping path="/abc" />
                &lt;!&ndash; 配置拦截器 &ndash;&gt;
                <ref bean="firstInterceptor"></ref>
        </mvc:interceptor>-->
</mvc:interceptors>
```

TestController

```
public class TestController {
        @RequestMapping("/test/hello")
        public String testhello() {
                return "success";
        }
}
```

拦截器的三个方法：

· preHandle()：在控制器方法执行之前执行，其返回值表示对控制器方法的拦截（false) 或放行 (true)

· postHandle()：在控制器方法执行之后执行

· afterCompletion()：在控制器方法执行之后，且渲染视图完毕之后执行

多个拦截器的执行顺序和在 SpringMvc 的配置文件中配置的顺序有关 preHandle() 按照配置的顺序执行，而 postHandle() 和 afterCompletion（) 按照配置的反序执行。

若拦截器中有某个拦戴器的 preHandle() 返回了 false，拦截器的 preHandle() 返回 false 和它之前的拦截器的 preHandle() 都会执行；

所有的拦截器的 postHandle() 都不执行，拦戴器的 preHandle() 返回 false 之前的拦戴器的 afterCompletion() 会执行。

FirstInterceptor

```
@Component
public class FirstInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("FirstInterceptor-->preHandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView
modelAndView) throws Exception {
        System.out.println("FirstInterceptor-->postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("FirstInterceptor-->afterCompletion");
    }
}
```


SecondInterceptor

```
@Component
public class SecondInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("SecondInterceptor-->preHandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView
modelAndView) throws Exception {
        System.out.println("SecondInterceptor-->postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("SecondInterceptor->afterCompletion");
    }
}
```

十二　异常处理器

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：

springmvc.xml

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <!--key 设置要处理的异常，value 设置出现该异常时要跳转的页面所对应的逻辑视图 -->
            <prop  key="java.lang.ArithmeticException">error</prop>
        </props>
    </property>
    <!-- 设置共享在请求域中的异常信息的属性名 -->
    <property name="exceptionAttribute" value="ex"></property>
</bean>
```

基于注解的异常处理

ExceptionController

```
// 将当前类标识为异常处理的组件
@ControllerAdvice
public class ExceptionController {
    // 设置要处理的异常信息
    @ExceptionHandler(ArithmeticException.class)
    public String handleException(Throwable ex, Model model) {
        // ex 表示控制器方法所出现的异常
        model.addAttribute("ex", ex);
        return "error";
    }
}
```


error.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title> 错误 </title>
    </head>
    <body>
        <h1>error.html</h1>
        <p th:text="${ex}"></p>
    </body>
</html>
```

TestController

```
public class TestController {
    @RequestMapping("/test/hello")
    public String testhello() {
        System.out.println(1 / 0);
        return "success";
    }
}
```

十三　注解配置 SpringMVC

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。

WebInit

```
// 代替 web.xml
public class WebInit extends AbstractAnnotationConfigDispatcherSer
vletInitializer {
	// 设置一个配置类代替 Spring 的配置文件
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class[] { SpringConfig.class };
	}

	// 设置一个配置类代替 SpringMVC 的配置文件
	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	// 设置 SpringMVC 的前端控制器 DispatcherServlet 的 url-pattern
	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}

	@Override
	// 设置当前的过滤器
	protected Filter[] getServletFilters() {
		// 创建编码过滤器
		CharacterEncodingFilter characterEncodingFilter = new
CharacterEncodingFilter();
		characterEncodingFilter.setEncoding("UTF-8");
		characterEncodingFilter.setForceEncoding(true);
		// 创建处理请求方式的过滤器
		HiddenHttpMethodFilter hiddenHttpMethodFilter = new
HiddenHttpMethodFilter();
		return new Filter[] { characterEncodingFilter,
hiddenHttpMethodFilter };
	}
}
```

TestController

```
@Controller
public class TestController {
}
```

SpringConfig

```
// 代替 Spring.xml 的配置文件
// 将类标识为配置类
@Configuration
public class SpringConfig {
}
```

WebConfig

```
/* 代替 SpringMVC 的配置文件
扫描组件、视图解析器、默认的 servlet、mvc 的注解驱动
视图控制器、文件上传解析器、拦截器、异常解析器
 */
// 将类标识为配置类
@Configuration
// 扫描组件
@ComponentScan("controller")
// 开启 mvc 的注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
    @Override
    // 默认的 servlet 处理静态资源
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

    @Override
    // 配置视图解析器
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }

    @Bean
    // @Bean 注解可以将标识的方法的返回值作为 bean 进行管理，bean 的 id 为方法的方法名
    public CommonsMultipartResolver multipartResolver() {
        return new CommonsMultipartResolver();
    }

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



```
    // 配置异常解析器
    @Override
    public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
        Properties properties = new Properties();
        properties.setProperty("java.lang.ArithmeticException", "error");
        exceptionResolver.setExceptionMappings(properties);
        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;
        }
}
```

index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
            <title> 首页 </title>
    </head>
    <body>
        <h1>index.html</h1>
    </body>
</html>
```

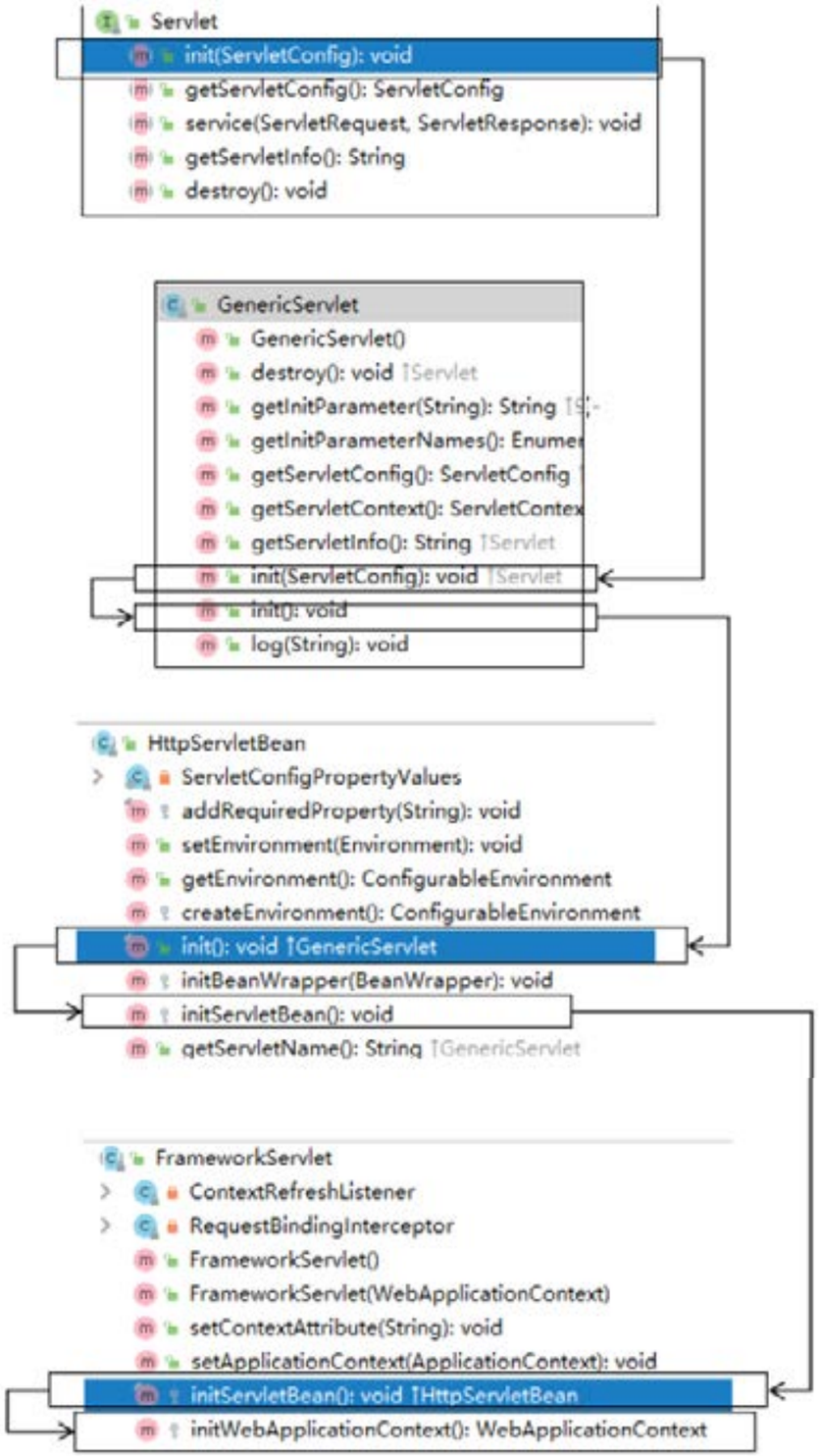

十四　SpringMVC 执行流程

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

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(cw
ac);
                        }
                }
        }
        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;
}
```

③ DispatcherServlet 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);
}
```

DispatcherServlet 调用组件处理请求

① processRequest()

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 =buildRequestAttrib
utes(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);
	}
}
```

② doService()
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.retrieve
AndUpdate(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);
        }
}
```

③ doDispatch()
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(processedReque
st, 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.applyAfterConcurrentHandling
Started(processedRequest, response);
                        }
                }
                else {
                // Clean up any resources used by a multipart request.
                        if (multipartRequestParsed) {
                                cleanupMultipart(processedRequest);
                        }
                }
        }
}
```

④ processDispatchResult()

```
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);
        }
}
```

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

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。

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。

SSM 整合

一 测试 ContextLoaderListener

web.xml

```
<servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.
DispatcherServlet</servlet-class>
        <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>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>
<listener>
        <!-- 在服务器启动时，加载 Spring 的配置文件 -->
        <listener-class>org.springframework.web.context.
ContextLoaderListener</listener-class>
</listener>
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring.xml</param-value>
</context-param>
```

spring.xml

```
<!-- 扫描组件 -->
<context:component-scan base-package="ssm.service.Impl">
</context:component-scan>
```

HelloService 接口

```
public interface HelloService {
}
```

HelloServiceImpl

```
@Service
public class HelloServiceImpl implements HelloService {
}
```


springmvc.xml

```
<!-- 扫描控制层组件 -->
<context:component-scan base-package="ssm.controller"></
context:component-scan>
<!-- 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>
<mvc:annotation-driven /><mvc:view-controller path="/" view-
name="index"></mvc:view-controller>
```

HelloController

```
@Controller
public class HelloController {
        @Autowired
        private HelloService helloService;
}
```

二　准备工作

①创建 Maven Module
②导入依赖

```
<properties>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
        <spring.version>5.3.1</spring.version>
</properties>

<packaging>war</packaging>

<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>2.0.7</version>
        </dependency>

        <!-- 连接池 -->
        <dependency>
                <groupId>com.alibaba</groupId>
                <artifactId>druid</artifactId>
                <version>1.0.8</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
```

三　配置 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">

    <!-- 设置 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 DispatcherServlet-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.
DispatcherServlet</servlet-class>
        <!-- 设置 SpringMVC 配置文件自定义的位置和名称 -->
        <init-param>
            <param-name>contextConfigLocation</param-
name>
            <param-value>classpath:springmvc.xml</param-
value>
        </init-param>
<!-- 将 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>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

<!-- 配置 Spring 的监听器，在服务器启动时加载 Spring 的配置文件 -->
    <listener>
        <listener-class>org.springframework.web.context.
ContextLoaderListener</listener-class>
    </listener>

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

四　创建 SpringMVC 的配置文件

```
<?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"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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 http://www.springframework.org/schema/mvc https://www.
springframework.org/schema/mvc/spring-mvc.xsd">

       <!-- 扫描控制层组件 -->
       <context:component-scan base-package="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 />

       <!-- 配置视图控制器 -->
       <mvc:view-controller path="/" view-name="index"></
mvc:view-controller>

       <!-- 配置文件上传解析器 -->
       <bean id="multipartResolver" class="org.springframework.web.
multipart.commons.CommonsMultipartResolver"></bean>
</beans>
```

五　创建 Spring 的配置文件

```
<?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" xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd http://www.springframework.org/schema/context https://www.
springframework.org/schema/context/spring-context.xsd http://
www.springframework.org/schema/tx http://www.springframework.
org/schema/tx/spring-tx.xsd">
       <!-- 扫描组件 ( 除控制层 )-->
       <context:component-scan base-package="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>

       <!-- 配置事务管理器 -->
       <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
将使用注解 oTransactional 标识的方法或类中所有的方法进行事务管理
       -->
       <tx:annotation-driven transaction-
manager="transactionManager" />

       <!-- 配置 SqlSessionFactoryBean，可以直接在 Spring 的 IOC 中获
取 SqlSessionFactory-->
       <bean class="org.mybatis.spring.SqlSessionFactoryBean">
              <!-- 设置 MyBatis 的核心配置文件的路径 -->
              <property name="configLocation"
value="classpath:mybatis-config.xml"></property>
              <!-- 设置数据源 -->
              <property name="dataSource" ref="dataSource"></
property>
              <!-- 设置类型别名对应的包 -->
              <property name="typeAliasesPackage" value="ssm.
pojo"></property>
              <!-- 设置映射文件的路径，只有映射文件的包和 mapper 接口
的包不一致时需要设置 -->
              <!--<property name="mapperLocations"
value="classpath:mappers/*.xml"></property>-->
              <!--<property name="plugins">
                     <array>
                            <bean class="com.github.pagehelper.
PageInterceptor"></bean>
                     </array>
                            </property>-->
       </bean>

       <!-- 配置 mapper 接口的扫描，可以将指定包下所有的 mapper 接口，
通过 SqlSession 创建代理实现类对象，并将这些对象交给 IOC 容器管理 -->
       <bean class="org.mybatis.spring.mapper.
MapperScannerConfigurer">
              <property name="basePackage" value="ssm.mapper"></
property>
       </bean>
</beans>
```

六　搭建 MyBatis 环境

①创建属性文件 jdbc.properties

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username=root
jdbc.password=abc123
```

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> getAllEmployee();
}

<mapper namespace="ssm.mapper.EmployeeMapper">
        <!--List<Employee> getAllEmployee();-->
        <select id="getAllEmployee" 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>
```

七　测试功能

①创建组件

Employee

```
public class Employee {
	private Integer empId;
	private String empName;
	private Integer age;
	private String gender;
	private String email;

	public Employee() {
	}

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

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

	public String getEmail() {
		return email;
	}

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

	@Override
	public String toString() {
		return "Employee{" +
				"empId=" + empId +
				", empName='" + empName + '\'' +
				", age=" + age +
				", gender='" + gender + '\'' +
				", email='" + email + '\'' +
		'}';
	}
}
```

EmployeeController

查询所有的员工信息 - - >/employee - - >get
查询员工的分页信息 - - >/employee/page/1 - - >get
根据 id 查询员工信息 - - >/employee/1 - - >get
跳转到添加页面 - - >/to/add - - >get
添加员工信息 - - >/employee - - >post
修改员工信息 - - >/employee - - >put
删除员工信息 - - >/employee/1 - - >delete

```
@Controller
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;

    @RequestMapping(value = "/employee/page/{pageNum}",
method = RequestMethod.GET)
    public String getEmployeePage(@PathVariable("pageNum")
Integer pageNum, Model model) {
        // 获取员工的分页信息
        PageInfo<Employee> page = employeeService.
getEmployeePage(pageNum);
        // 将分页数据共享到请求域中
        model.addAttribute("page", page);
        // 跳转到 employee_list.html
        return "employee_list";
    }

    @RequestMapping(value = "/employee", method =
RequestMethod.GET)
    private String getAllEmployee(Model model) {
        // 查询所有的员工信息
        List<Employee> list = employeeService.getAllEmployee();
        // 将员工信息在请求域中共享
        model.addAttribute("list", list);
        // 跳转到 employee_list.html
        return "employee_list";
    }

}
```


EmployeeController 接口

```
public interface EmployeeService {
    // 查询所有的员工信息
    List<Employee> getAllEmployee();

    // 获取员工的分页信息
    PageInfo<Employee> getEmployeePage(Integer pageNum);
}
```

EmployeeServiceImpl

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService {
    @Autowired
    private EmployeeMapper employeeMapper;

    @Override
    public List<Employee> getAllEmployee() {
        return employeeMapper.getAllEmployee();
    }

    @Override
    public PageInfo<Employee> getEmployeePage(Integer
pageNum) {
        // 开启分页功能
        PageHelper.startPage(pageNum, 4);
        // 查询所有的员工信息
        List<Employee> list = employeeMapper.getAllEmployee();
        // 获取分页相关数据
        PageInfo<Employee> page = new PageInfo<>(list, 5);
        return page;
    }
}
```

②创建页面

index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 首页 </title>
</head>
<body>
<h1>index.html</h1>
<a th:href="@{/employee/page/1}"> 查询员工的分页信息 </a>
</body>
</html>
```

EmployeeServiceImpl

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title> 员工列表 </title>
    <link rel="stylesheet" th:href="static/css/index_work.css">
</head>
<body>
<table>
    <tr>
        <th> 员工列表 </th>
    </tr>
    <tr>
        <th> 流水号 </th>
        <th> 员工姓名 </th>
        <th> 年龄 </th>
        <th> 性别 </th>
        <th> 邮箱 </th>
        <th> 操作 </th>
    </tr>
    <tr th:each="employee, status : ${page.list}">
        <td th:text="${status.count}"></td>
        <td th:text="${employee.empName}"></td>
        <td th:text="${employee.age}"></td>
        <td th:text="${employee.gender}"></td>
        <td th:text="${employee.email}"></td>
        <td>
            <a th:href=""> 删除 </a>
            <a th:href=""> 修改 </a>
        </td>
    </tr>

</table>

<div style="text-align: center;">
    <a th:if="${page.hasPreviousPage}" th:href="@{/employee/page/1}"> 首页 </a>
    <a th:if="${page.hasPreviousPage}" th:href="@{'/employee/page/' + ${page.prePage}}"> 上一页 </a>
    <span th:each="num : ${page.navigatpamageNums}">
        <a th:if="${page.pageNum == num}" style="color : red" th:href="@{'/employee/page/' + ${num}}" th:text="'[' + num + ']'"></a>
        <a th:if="${page.pageNum != num}" th:href="@{'/employee/page/' + ${num}}" th:text="num"></a>
    </span>
    <a th:if="${page.hasNextPage}" th:href="@{'/employee/page/' + ${page.nextPage}}"> 下一页 </a>
    <a th:if="${page.hasNextPage}" th:href="@{'/employee/page/' + ${page.pages}}"> 末页 </a>
</div>
</body>
</html>
```