课程目标

1. 开发环境搭建

1.1 整合思路

- 数据库配置信息由交给 `Spring` 读取和加载
- 数据源信息 `DataSource` 交给 `Spring` 管理
- `SqlSessionFactory` 交给 `Spring` 进行**单例** 管理
- `SqlSession` 交给 `Spring` 进行**多例**管理
- 由 `Spring` 来管理 `Mapper` 接口的实现类(`Mapper` 接口需要程序员实现)
- 由 `Spring` 通过动态代理自动生成 `Mapper` 接口的代理类实现类

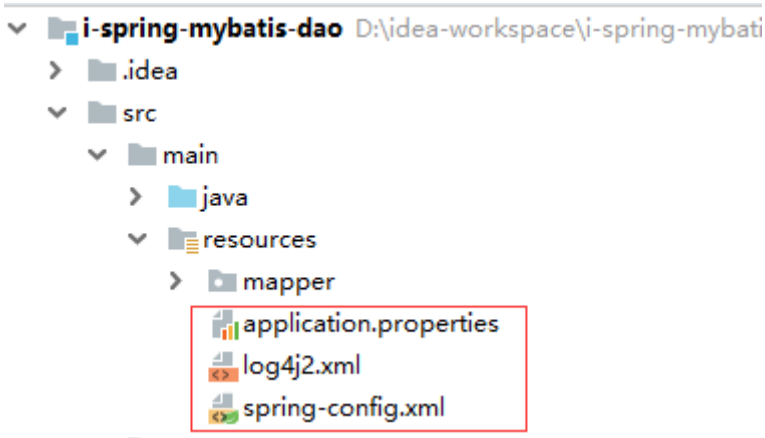
1.2 项目依赖

```
<dependencies>
    <!-- junit start -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.10.RELEASE</version>
        <scope>test</scope>
    </dependency>
    <!-- junit end -->

    <!-- 日志类库 log4j2 -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.12.1</version>
    </dependency>

    <!-- spring start -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>

    <!--spring 和 aspectj 框架整合的模块-->
    <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>

    <!--spring 支持jdbc编程模块-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>
    <!-- spring end -->

    <!-- mysql驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.20</version>
        <scope>runtime</scope>
    </dependency>

    <!--数据库连接池-->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.20</version>
    </dependency>

    <!-- mybatis -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>

    <!-- mybatis 和 spring 整合依赖包 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.2</version>
    </dependency>
</dependencies>
```

1.3 配置文件

- 创建名称为 `i-spring-mybaits-dao` 的 `maven` 项目。



application.properties

```
jdbc.username=root
jdbc.password=123456
jdbc.url=jdbc:mysql://127.0.0.1:3306/i-soufang?useSSL=false&serverTimezone=Asia/Shanghai
jdbc.driverClassName=com.mysql.jdbc.Driver
```

log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--先定义所有的appender-->
    <appenders>
        <!--输出控制台的配置-->
        <console name="Console" target="SYSTEM_OUT">
            <!--输出日志的格式-->
            <!--<patternlayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%p] %c %m%n"/>-->
            <patternlayout pattern="[%p] %m %n"/>
        </console>
    </appenders>

    <!-- 然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <!-- 日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->
    <loggers>
        <root level="DEBUG">
            <!--输出到控制台-->
            <appender-ref ref="Console"/>
        </root>
        <!--org.springframework
        <logger name="org.springframework" level="INFO"/>-->
    </loggers>
</configuration>
```

spring-config.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">

    <!--启用注解扫描，扫描含有注解的类-->
    <context:component-scan base-package="com.imcode.soufang"/>

    <!--加载属性配置文件的内容-->
    <context:property-placeholder location="classpath:application.properties"/>

    <!--数据源交给spring容器管理-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
    </bean>

    <!--SqlSessionFactory 会话工厂交给spring容器管理-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--注入数据源-->
        <property name="dataSource" ref="dataSource"/>
        <!--配置Mapper映射文件的位置-->
        <property name="mapperLocations" value="classpath:mapper/*Mapper.xml"/>
    </bean>

    <!--SqlSession 会话交给spring容器管理-->
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate"
scope="prototype">
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>
</beans>
```

1.4 实战案例

- 根据ID查询房源信息
- 根据ID删除房源信息

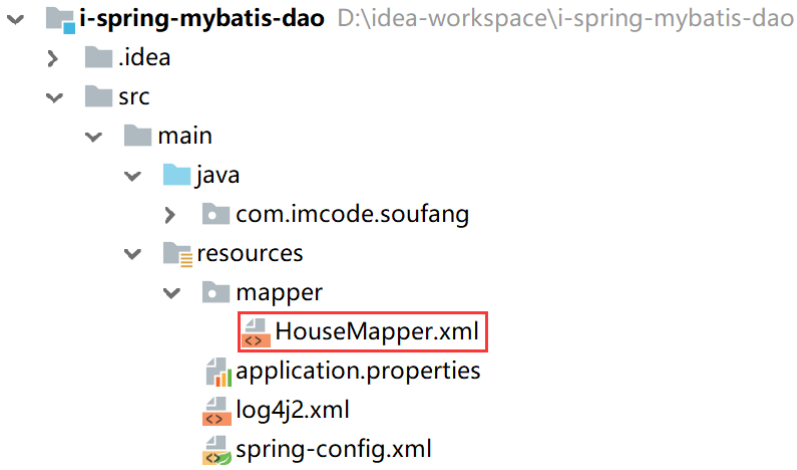
Model

```
public class House {
    private Integer id;
    private String title;
    private String logo;
    private Double price;
    private Double area;
    private String city;
    private String district;
    private Date createTime;
    private Date updateTime;
    ...
}
```

Mapper 接口

```java
public interface HouseMapper {
    House getById(Integer id);
    void deleteById(Integer id);
}
```

Mapper 映射文件




```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">
<!--namespace命名空间，作用就是对sql进行分类化管理，理解sql隔离-->
<mapper namespace="com.imcode.soufang.mapper.HouseMapper">
    <select id="getById" parameterType="java.lang.Integer"
            resultType="com.imcode.soufang.model.House">
        SELECT
           id,
           title,
           logo,
           price,
           area,
           city,
           district,
           create_time createTime,
           update_time updateTime
        FROM house WHERE id = #{houseId}
    </select>
</mapper>
```

2. 手动实现Mapper接口

2.1 Mapper 接口实现类

```java
@Repository
public class HouseMapperImpl implements HouseMapper {

//    @Autowired
//    private SqlSessionFactory sqlSessionFactory;
```

```
6
7       @Autowired
8       private SqlSession sqlSession;
9
10      @Override
11      public House getById(Integer id) {
12  //        SqlSession sqlSession = sqlSessionFactory.openSession();
13          House house
14              =
    sqlSession.selectOne("com.imcode.soufang.mapper.HouseMapper.getById",1);
15          return house;
16      }
17
18      @Override
19      public void deleteById(Integer id) {
20
    sqlSession.delete("com.imcode.soufang.mapper.HouseMapper.deleteById",id);
21          // 注意：spring 管理 会话工厂和会话后，事务的提交、会话的打开和关闭都由spring容器控
    制
22          // 我们无法控制事务的提交和回滚 也无法自己关闭sql会话 后续知识点解决
23          //sqlSession.rollback();
24          //sqlSession.commit();
25      }
26  }
```

2.2 单元测试

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @ContextConfiguration(locations = "classpath:spring-config.xml")
3   public class HouseMapperImplTest {
4
5       @Autowired
6       private HouseMapper houseMapper;
7
8       @Test
9       public void getById() {
10          House house = houseMapper.getById(1);
11          System.out.println(house);
12      }
13
14      @Test
15      public void deleteById() {
16          houseMapper.deleteById(8);
17      }
18  }
```

3. 动态代理实现Mapper接口

- 复制 i-spring-mybatis-dao 项目，重命名为 i-spring-mybatis-mapper

3.1 spring-config.xml

- 删除 第二步中自己动手实现的 `Housemapper` 接口的实现类 `HouseMapperImpl`
- 在 `spring-config.xml` 文件中增加 `spring` 扫描 `Mapper` 接口的配置
  - 该配置需要注入会话工厂
  - 该配置需要指定mapper接口所在的包

```
<!--扫描Mapper接口的配置-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!--注入sqlSessionFactory会话工厂-->
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    <!--配置mapper接口所在的位置-->
    <property name="basePackage" value="com.imcode.soufang.mapper"/>
</bean>
```

3.2 单元测试

- 复用 手动实现Mapper 接口的单元测试就可以，无需修改

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-config.xml")
public class HouseMapperImplTest {

    @Autowired
    private HouseMapper houseMapper;

    @Test
    public void getById() {
        House house = houseMapper.getById(1);
        System.out.println(house);
    }

    @Test
    public void deleteById() {
        houseMapper.deleteById(8);
    }
}
```

3.3 总结

- 使用动态代理实现 `Mapper` 接口的开发模式，程序员不需要编写接口实现类， `spring` 框架在扫描到mapper接口后，会使用 `jdk` 或 `cglib` 动态代理技术在运行期间帮我们生成 `mapper` 接口的代理实现类，并初始化代理实现类的对象到 `spring` 容器中。

4. Spring 事务管理

4.1 配置事务管理器

- 在 `spring-config.xml` 文件增加事务管理器的配置
    - `mybatis` 使用 `jdbc` 的事务管理器

```
<!--配置spring的 jdbc 的事务管理器-->
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源-->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

4.2 增加服务层

```
/**
 * 酒店服务类
 */
@Service
public class HouseService {

    @Autowired
    private HouseMapper houseMapper;

    /**
     * 根据酒店id删除酒店信息
     * @param id
     */
    public void deleteById(Integer id){
        houseMapper.deleteById(id);
        int i = 100/0;
    }

    public House getById(Integer id) {
        return houseMapper.getById(id);
    }
}
```

4.3 基于XML

- 在spring-config.xml 文件增加事务配置，测试事务是否起作用

```
<!-- 事务增强/通知-->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="*"/>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="find*" read-only="true"/>
    </tx:attributes>
</tx:advice>

<!--事务切面-->
<aop:config>
    <aop:advisor advice-ref="txAdvice"
```

```
                    pointcut="execution(* com.imcode.soufang.service.*Service.*(..))"/>
    </aop:config>
```

4.4 基于注解

- 注释掉基于xml的注解配置，开启基于注解的事务控制

```
    <!-- aop注解生效 -->
    <aop:aspectj-autoproxy/>
    <!--启用基于注解的事务管理-->
    <tx:annotation-driven transaction-manager="txManager"/>

    <!-- 事务增强/通知
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="*"/>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="find*" read-only="true"/>
        </tx:attributes>
    </tx:advice>-->

    <!--事务切面
    <aop:config>
        <aop:advisor advice-ref="txAdvice"
                     pointcut="execution(* com.imcode.soufang.service.*Service.*(..))"/>
    </aop:config>-->
```

- 在需要事务控制的方法或类上增加 `@Transactional` 注解

  可以用在方法和类上

```
    @Service
    @Transactional
    public class HouseService {

        @Autowired
        private HouseMapper houseMapper;

        // @Transactional
        public void deleteById(Integer id){
            houseMapper.deleteById(id);
            int i = 100/0;
        }

        @Transactional(readOnly = true)
        public House getById(Integer id) {
            return houseMapper.getById(id);
        }
    }
```

> @Transactional 注解使用到类上，类中所有方法都受事务控制
>
> @Transactional 注解使用到方法上，只对该方法进行事务控制

> 类和方法上都有 @Transactional ， 方法上的注解配置覆盖类上的注解配置

4.5 事务管理细节

事务管理器

`PlatformTransactionManager` 用于执行具体的事务操作。事务管理器

接口定义：

```
1   public interface PlatformTransactionManager{
2      //获取事务状态
3      TransactionStatus getTransaction(TransactionDefinition definition)
4                              throws TransactionException;
5      //提交事务
6      void commit(TransactionStatus status)throws TransactionException;
7
8      //回滚事务
9      void rollback(TransactionStatus status)throws TransactionException;
10  }
```

根据底层所使用的不同的持久化 API 或框架，

PlatformTransactionManager 的主要实现类：

- `DataSourceTransactionManager`
  - 适用于使用JDBC和MyBatis进行数据持久化操作的情况。
- `HibernateTransactionManager`
  - 适用于使用Hibernate进行数据持久化操作的情况。
- `JpaTransactionManager`
  - 适用于使用JPA进行数据持久化操作的情况。

- `DataSourceTransactionManager` 在定义时需要提供底层的数据源作为其属性，也就是 `DataSource`
- 与 `HibernateTransactionManager` 对应的是 `SessionFactory`
- 与 `JpaTransactionManager` 对应的是 `EntityManagerFactory`

隔离级别

隔离级别是指若干个并发的事务之间的隔离程度。

`TransactionDefinition` 接口中定义了五个表示隔离级别的常量：

```java
package org.springframework.transaction;
import org.springframework.lang.Nullable;

public interface TransactionDefinition {
    ...
    int ISOLATION_DEFAULT = -1; // 使用数据库默认的隔离级别
    int ISOLATION_READ_UNCOMMITTED = 1; // 未提交读  脏读 幻读 不可重复度
    int ISOLATION_READ_COMMITTED = 2;   // 已提交读  幻读 不可重复度  避免脏读
    int ISOLATION_REPEATABLE_READ = 4;  // 可重复读  幻读           避免不可重复度、避免脏读
    int ISOLATION_SERIALIZABLE = 8;     // 串行化    脏读 幻读 不可重复度都不会发生
    ....
}
```

```java
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {
    @AliasFor("transactionManager")
    String value() default "";

    @AliasFor("value")
    String transactionManager() default "";

    Propagation propagation() default Propagation.REQUIRED;

    Isolation isolation() default Isolation.DEFAULT;

    int timeout() default -1;

    boolean readOnly() default false;

    Class<? extends Throwable>[] rollbackFor() default {};

    String[] rollbackForClassName() default {};

    Class<? extends Throwable>[] noRollbackFor() default {};

    String[] noRollbackForClassName() default {};
}
```

```java
public enum Isolation {
    DEFAULT(-1),
    READ_UNCOMMITTED(1),
    READ_COMMITTED(2),
    REPEATABLE_READ(4),
    SERIALIZABLE(8);

    private final int value;

    private Isolation(int value) {
        this.value = value;
    }

    public int value() {
        return this.value;
```

```
    }
}
```

- TransactionDefinition.ISOLATION_DEFAULT
  - 这是默认值，表示使用底层数据库的默认隔离级别
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED
  - 该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。
  - 该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。
- TransactionDefinition.ISOLATION_READ_COMMITTED
  - 该隔离级别表示一个事务只能读取另一个事务已经提交的数据。
  - 该级别可以防止脏读，这也是大多数情况下的推荐值。
- TransactionDefinition.ISOLATION_REPEATABLE_READ
  - 该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。
  - 即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。
  - 该级别可以防止脏读和不可重复读。
- TransactionDefinition.ISOLATION_SERIALIZABLE
  - 所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰
  - 该级别可以防止脏读、不可重复读以及幻读
  - 但是这将严重影响程序的性能。通常情况下也不会用到该级别

传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。

案例准备

```
HouseDetailMapper
    deleteByHouseId(id);//通过房源id删除房源详情

HouseDetailService {
   HouseDetailMapper houseDetailMapper;
   @Transactional
   deleteByHouseId(id) {
      // 开启事务
      houseDetailMapper.deleteByHouseId(id);
      // 提交事务
   }
}

HouseMapper
   deleteById(id);

HouseService {
   HouseMapper houselMapper;
   HouseDetailService houseDetailService;

   // 根据id删除房源，同时删除该房源的详情信息
   //@Transactional
   deleteById(Integer id){
       //开启事务
```

```
        // 遇到了另外一个也受事务控制的方法
        houseDetailService.deleteByHouseId(id); //加入事务
        deleteById(id);
        //提交事务
    }
}
```

```
package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface TransactionDefinition {

    // ------ 事务的传播行为
    //如果当前存在事务则加入该事务；如果当前没有事务，则创建一个新的事务
    int PROPAGATION_REQUIRED = 0;

    //如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行
    int PROPAGATION_SUPPORTS = 1;

    int PROPAGATION_MANDATORY = 2;
    int PROPAGATION_REQUIRES_NEW = 3;
    int PROPAGATION_NOT_SUPPORTED = 4;
    int PROPAGATION_NEVER = 5;
    int PROPAGATION_NESTED = 6;
    ....
}
```

常用传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。

在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

- `TransactionDefinition.PROPAGATION_REQUIRED` 增加、修改、删除
  - 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务
- **TransactionDefinition.PROPAGATION_REQUIRES_NEW**
  - 创建一个新的事务，如果当前存在事务，则把当前事务挂起
  - **REQUIRES_NEW的事务执行完毕后，继续执行挂起的事务**
- `TransactionDefinition.PROPAGATION_SUPPORTS` 查询 只读事务
  - 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED**
  - 以非事务方式运行，如果当前存在事务，则把当前事务挂起
- **TransactionDefinition.PROPAGATION_NEVER**
  - 以非事务方式运行，如果当前存在事务，则抛出异常。
- **TransactionDefinition.PROPAGATION_MANDATORY**
  - 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
- **TransactionDefinition.PROPAGATION_NESTED 嵌套事务**
  - 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；
  - 如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION_REQUIRED。

事务超时

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

只读事务

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。所谓事务性资源就是指那些被事务管理的资源，比如数据源、 JMS 资源，以及自定义的事务性资源等等。如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。在 TransactionDefinition 中以 boolean 类型来表示该事务是否只读。

回滚事务

通常情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常），则默认将回滚事务。如果没有抛出任何异常，或者抛出了已检查异常，则仍然提交事务。这通常也是大多数开发者希望的处理方式。但是，我们可以根据需要人为控制事务在抛出某些未检查异常时任然提交事务，或者在抛出某些已检查异常时回滚事务。