第2天: Spring

一、学习目标

1. 能够使用Jdbc的模板
2. 能够配置Spring的连接池
3. 能够使用JdbcTemplate完成增删查改操作
4. 使用注解改造CRUD工程
5. 基于纯注解改造CRUD工程
6. 能够实现Spring框架整合Junit
7. 理解代理设计模式
8. 掌握动态代理的两种实现方式
9. 完成动态代理案例

二、Spring整合JDBC

2.1 JdbcTeamplate概诉

> JdbcTemplate是Spring提供的一个模板类，它是对jdbc的封装。用于支持持久层的操作。它的特点是：简单、方便。
>
> 它简化了JDBC的使用并有助于避免常见错误。它执行核心的JDBC工作流程，留下应用程序代码以提供SQL并提取结果。此类执行SQL查询或更新，在ResultSets上启动迭代并捕获JDBC异常，并将其转换为`org.springframework.dao`程序包中定义的通用异常，信息量更大的异常层次结构 。

2.2 入门案例

2.2.1 准备数据

- 数据库: db.properties

```
db.driver=com.mysql.jdbc.Driver
# db.url=jdbc:mysql:///mybatisdb?serverTimezone=UTC&amp;useSSL=false&amp;allowPublicKeyRetrieval=true&amp;characterEncoding=UTF-8
db.url=jdbc:mysql:///mybatisdb?serverTimezone=UTC&useSSL=false&allowPublicKeyRetrieval=true&characterEncoding=UTF-8
db.username=root
db.password=root
```

- 数据表: `mybatis`.`account`

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for account
-- ----------------------------
```

```
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `uid` int(11) DEFAULT '1' COMMENT '用户编号',
  `money` decimal(10,2) DEFAULT '0.00' COMMENT '余额',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=114 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of account
-- ----------------------------
INSERT INTO `account` VALUES ('1', '1', '10.00');
INSERT INTO `account` VALUES ('2', '10', '9.90');
INSERT INTO `account` VALUES ('3', '24', '99.00');
```

2.2.2 案例演示

- 工程名称: spring-day02-jdbc
- 添加依赖: pom.xml

```
<!-- 添加依赖 -->
<dependencies>
    <!-- 1. Spring IOC 依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.7.RELEASE</version>
    </dependency>
    <!-- 2. Spring Jdbc 依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.1.7.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.17</version>
    </dependency>
    <!-- 3. Junit 单元测试 依赖 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

- 创建实体: com.itheima.jdbc.Account

```
package com.itheima.jdbc;

/**
 * 账户实体.
 *
 * @author : Jason.lee
 * @version : 1.0
```

```
10
11      private Integer id;
12      private Integer uid;
13      private Double money;
14
15      @Override
16      public String toString() {
17          return "Account{" +
18                  "id=" + id +
19                  ", uid=" + uid +
20                  ", money=" + money +
21                  '}';
22      }
23  }
```

- 案例代码: JdbcTests

```
1   import com.itheima.jdbc.Account;
2   import org.junit.Test;
3   import org.springframework.jdbc.core.BeanPropertyRowMapper;
4   import org.springframework.jdbc.core.JdbcTemplate;
5   import org.springframework.jdbc.datasource.DriverManagerDataSource;
6
7   import javax.sql.DataSource;
8   import java.util.List;
9
10  /**
11   * JdbcTemplate操作案例.
12   *
13   * @author : Jason.lee
14   * @version : 1.0
15   */
16  public class JdbcTests {
17
18      /**
19       * 演示: 查询所有账户
20       */
21      @Test
22      public void testJdbcTemplate (){
23          DataSource ds = createDataSource();
24
25          // 1. 创建操作模板对象
26          JdbcTemplate jd = new JdbcTemplate(ds);
27          // 2. 定义Sql语句
28          String sql = "select * from account";
29          // 3. 处理结果
30          List<Account> all = jd.query(sql, new
    BeanPropertyRowMapper<Account>(Account.class));
31          all.forEach(x -> System.out.println(x));
32      }
33
34      /**
35       * 创建数据源
36       * @return 数据源对象
```

```
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql:///mybatisdb?
serverTimezone=UTC&useSSL=false&allowPublicKeyRetrieval=true&characterE
ncoding=UTF-8");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}
```

2.3 使用IOC管理模板

2.3.1 添加配置

- 配置文件: applicationContext.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">

    <!-- 1. 定义数据源 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///mybatisdb?
serverTimezone=UTC&amp;useSSL=false&amp;allowPublicKeyRetrieval=true&am
p;characterEncoding=UTF-8"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
    <!-- 2. 创建操作模板 -->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

2.3.2 使用容器

- 案例代码: JdbcTests

```
/**
     * 演示: 查询所有账户
     */
ClassPathXmlApplicationContext ioc = new
ClassPathXmlApplicationContext("applicationContext.xml");
// 1. 获取操作模板对象
JdbcTemplate jd = ioc.getBean(JdbcTemplate.class);
@Test
public void testIocJdbc (){
```

```
    // 3. 处理结果
    List<Account> all = jd.query(sql, new BeanPropertyRowMapper<Account>(Account.class));
    all.forEach(x -> System.out.println(x));
}
```

2.4 结果集映射

2.4.1 多记录映射器

- 查询多个账户

```
/**
  * 演示: 多记录查询 (BeanProperty映射器)
  */
@Test
public void testMany (){
    // 2. 定义Sql语句
    String sql = "select * from account where id in(?,?)";
    // 3. 处理结果
    List<Account> all = jd.query(sql, new BeanPropertyRowMapper<Account>(Account.class), new Integer[]{1, 3});
    all.forEach(x -> System.out.println(x));
}
```

2.4.2 自定义映射器

- 查询多个账户
- 自定义映射器: com.itheima.jdbc.AccountRowMapper

```
package com.itheima.jdbc;

import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * 自定义账户属性与表字段映射器.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AccountRowMapper implements RowMapper<Account> {
    /**
     * 单行记录处理方法
     * @param rs 结果集
     * @param i 行号 (0开始)
     * @return 行记录的封装对象
     * @throws SQLException 异常信息
     */
    @Override
```

```
        account.setId(rs.getInt("id"));
        account.setUid(rs.getInt("uid"));
        account.setMoney(rs.getDouble("money"));
        return account;
    }
}
```

- 单元测试: JdbcTests

```
/**
  * 演示: 多记录查询 (自定义映射器)
  */
@Test
public void testCustom (){
    // 2. 定义Sql语句
    String sql = "select * from account where id in(?,?)";
    // 3. 处理结果
    List<Account> all = jd.query(sql, new AccountRowMapper(), new
Integer[]{1, 3});
    all.forEach(x -> System.out.println(x));
}
```

2.4.3 单记录映射

- 根据编号查找账户

```
/**
  * 演示: 单记查询 (自定义映射器)
  */
@Test
public void testOne () {
    // 2. 定义Sql语句
    String sql = "select * from account where id in(?)";
    // 3. 处理结果
    Account account = jd.queryForObject(sql, new RowMapper<Account>() {
        /**
             * 每行记录调用的封装方法
             *      好处: 可以自由映射表字段与对象属性的关系
             * @param resultSet 结果集
             * @param i 行号 (0开始)
             * @return 封装的对象
             */
        @Override
        public Account mapRow(ResultSet resultSet, int i) throws
SQLException {
            System.out.println("=========================="+ i);
            Account account = new Account();
            account.setId(resultSet.getInt("id"));
            account.setUid(resultSet.getInt("uid"));
            account.setMoney(resultSet.getDouble("money"));
            return account;
        }
    }, 1);
    System.out.println(account);
```

- 添加依赖: pom.xml

```xml
<!-- 4. Log4j 依赖 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.26</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.7</version>
</dependency>

<!-- C3P0数据源 -->
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.4</version>
</dependency>
<!-- Druid数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.12</version>
</dependency>
```

2.5.1 内置数据源

- applicationContext.xml

```xml
<!-- 内部数据源 -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///mybatisdb?
serverTimezone=UTC&amp;useSSL=false&amp;allowPublicKeyRetrieval=true&amp
;characterEncoding=UTF-8"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

2.5.2 C3P0数据源

- applicationContext.xml

```
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql:///mybatisdb?
serverTimezone=UTC&amp;useSSL=false&amp;allowPublicKeyRetrieval=true&amp
;characterEncoding=UTF-8"/>
    <property name="user" value="root"/>
    <property name="password" value="root"/>
</bean>
```

2.5.3 Druid数据源

- applicationContext.xml

```
<!-- druid数据源 -->
<bean id="druid" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///mybatisdb?
serverTimezone=UTC&amp;useSSL=false&amp;allowPublicKeyRetrieval=true&amp
;characterEncoding=UTF-8"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

三、实现IOC的CRUD

- 需求: 使用Spring管理对象
- 技术: 使用jdbcTemplate实现crud

2.1 构建工程

- 工程名称: spring-day02-crud
- pom.xm

```
<?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">
    <parent>
        <artifactId>spring2</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-day02-crud</artifactId>

    <dependencies>
        <!-- Spring IOC依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
```

```
        </dependency>
        <!-- Spring JDBC依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
        <!-- ali 连接池依赖 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.12</version>
        </dependency>
        <!-- Mysql 数据库驱动依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.15</version>
        </dependency>
        <!-- 单元测试 依赖 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

</project>
```

- Account.java

```
package com.itheima.crud.entity;

/**
 * 账户类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;

    public Integer getId() {
        return id;
    }

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

    public Integer getUid() {
        return uid;
```

```
        this.uid = uid;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                '}';
    }
}
```

- AccountDaoImpl.java

```
package com.itheima.crud.dao;

import com.itheima.crud.entity.Account;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import java.util.List;

/**
 * 账户持久层操作类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AccountDaoImpl implements AccountDao {

    JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public List<Account> findAll() {
        String sql = "select * from account";
        return jdbcTemplate.query(
                sql, new BeanPropertyRowMapper(Account.class)
        );
    }

    @Override
```

```
        jdbcTemplate.update(sql, account.getId(), account.getUid(),
account.getMoney());
    }

    @Override
    public void update(Account account) {
        String sql = "update account set uid=?, money=? where id=?";
        jdbcTemplate.update(sql, account.getUid(), account.getMoney(),
account.getId());
    }

    @Override
    public void del(Account account) {
        String sql = "delete from account where id=?";
        jdbcTemplate.update(sql, account.getId());
    }
}
```

- AccountServiceImpl.java

```
package com.itheima.crud.service;

import com.itheima.crud.dao.AccountDao;
import com.itheima.crud.entity.Account;

import java.util.List;

/**
 * 账户业务类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AccountServiceImpl implements AccountService {

    AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public List<Account> findAll() {
        return accountDao.findAll();
    }

    @Override
    public void add(Account account) {
        accountDao.add(account);
    }

    @Override
    public void update(Account account) {
```

```
    @Override
    public void del(Account account) {
        accountDao.del(account);
    }
}
```

2.2 文件配置

- db.properties

```
db.driver=com.mysql.cj.jdbc.Driver
# 针对Mysql 8.x数据库添加的参数
#   serverTimezone: 指定时区(UTC)
#   useSSL: 指定是否使用加密安全连接(false)
#   allowPublicKeyRetrieval: 是否允许检索公钥(true)
db.url=jdbc:mysql:///mybatisdb?
serverTimezone=UTC&useSSL=false&allowPublicKeyRetrieval=true
db.username=root
db.password=root
```

- applicationContext.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">

    <!-- 加载配置文件 -->
    <context:property-placeholder location="classpath:db.properties"/>

    <!-- 定义数据源对象 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${db.driver}"/>
        <property name="url" value="${db.url}"/>
        <property name="username" value="${db.username}"/>
        <property name="password" value="${db.password}"/>
    </bean>

    <!-- 定义JDBC操作模板对象 -->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 定义账户操作类对象 -->
    <bean id="accountDao" class="com.itheima.crud.dao.AccountDaoImpl">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>
```

```
class="com.itheima.crud.service.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"/>
    </bean>

</beans>
```

2.3 单元测试

- 测试类

```
import com.itheima.crud.entity.Account;
import com.itheima.crud.service.AccountService;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * IOC容器的CRUD单元测试.
 */
public class CrudTests {

    // 创建IOC容器
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取对象
    AccountService service = (AccountService)
context.getBean("accountService");

    @Test
    public void testFind (){
        List<Account> all = service.findAll();
        all.stream().forEach(x-> System.out.println(x));
    }
    @Test
    public void testAdd (){
        Account account = new Account();
        account.setId(11);
        account.setUid(1);
        account.setMoney(999.0);
        service.add(account);
    }
    @Test
    public void testUpdate (){
        Account account = new Account();
        account.setId(11);
        account.setUid(1);
        account.setMoney(0.0);
        service.update(account);
    }
    @Test
    public void testDel (){
        Account account = new Account();
```

```
    }
}
```

四、使用注解改造案例

- 配置+注解是企业开发的常用方式

4.1 工程改造

- 工程名称: spring-day02-ax

4.2 注解改造

- AccountDaoImpl.java

```java
package com.itheima.crud.dao;

import com.itheima.crud.entity.Account;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

/**
 * 使用@Repository代替
 * <bean id="accountDaoImpl" class="com.itheima.crud.dao.AccountDaoImpl">.
 */
@Repository
public class AccountDaoImpl implements AccountDao {

    // 使用@Autowired代替
    // <property name="jdbcTemplate" ref="jdbcTemplate"/>
    @Autowired
    JdbcTemplate jdbcTemplate;

//    使用注解后无需提供set方法
//    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
//        this.jdbcTemplate = jdbcTemplate;
//    }

    @Override
    public List<Account> findAll() {
        String sql = "select * from account";
        return jdbcTemplate.query(
                sql, new BeanPropertyRowMapper(Account.class)
        );
    }

    @Override
```

```
        jdbcTemplate.update(sql, account.getId(), account.getUid(),
account.getMoney());
    }

    @Override
    public void update(Account account) {
        String sql = "update account set uid=?, money=? where id=?";
        jdbcTemplate.update(sql, account.getUid(), account.getMoney(),
account.getId());
    }

    @Override
    public void del(Account account) {
        String sql = "delete from account where id=?";
        jdbcTemplate.update(sql, account.getId());
    }
}

```

- AccountServiceImpl.java

```
package com.itheima.crud.service;

import com.itheima.crud.dao.AccountDao;
import com.itheima.crud.entity.Account;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * 使用@Service代替
 * <bean id="accountServiceImpl"
class="com.itheima.crud.dao.AccountDaoImpl">.
 */
@Service
public class AccountServiceImpl implements AccountService {

    /**
     * 使用@Autowired代替
     * <bean id="accountService"
class="com.itheima.crud.service.AccountServiceImpl">
     */
    @Autowired
    AccountDao accountDao;

//    使用注解不需要提供set方法
//    public void setAccountDao(AccountDao accountDao) {
//        this.accountDao = accountDao;
//    }

    @Override
    public List<Account> findAll() {
        return accountDao.findAll();
```

```
    public void add(Account account) {
        accountDao.add(account);
    }

    @Override
    public void update(Account account) {
        accountDao.update(account);
    }

    @Override
    public void del(Account account) {
        accountDao.del(account);
    }
}
```

4.3 配置改造

- applicationContext.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">

    <!-- 【重点】 -->
    <!-- 使用注解需要开启注解扫面 -->
    <context:component-scan base-package="com.itheima.crud"/>


    <!-- 加载配置文件 -->
    <context:property-placeholder location="classpath:db.properties"/>

    <!-- 定义数据源对象 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${db.driver}"/>
        <property name="url" value="${db.url}"/>
        <property name="username" value="${db.username}"/>
        <property name="password" value="${db.password}"/>
    </bean>

    <!-- 定义JDBC操作模板对象 -->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 定义账户操作类对象 -->
    <!-- 使用@Repository代替了 -->
    <!--<bean id="accountDao" class="com.itheima.crud.dao.AccountDaoImpl">
```

```
36
37      <!-- 定义账户业务类对象 -->
38      <!-- 使用@Service代替了 -->
39      <!--<bean id="accountService"
class="com.itheima.crud.service.AccountServiceImpl">
40          <property name="accountDao" ref="accountDao"/>
41      </bean>-->
42
43  </beans>
```

4.4 单元测试

```
1   import com.itheima.crud.entity.Account;
2   import com.itheima.crud.service.AccountService;
3   import org.junit.Test;
4   import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6   import java.util.List;
7
8   /**
9    * 改造内容(根据类型获取service对象).
10  *  创建对象注解创建的对象默认以首字母小写的类名为对象名 (accountServiceImpl)
11  *
12  * @author : Jason.lee
13  * @version : 1.0
14  */
15  public class CrudTests {
16
17      // 创建IOC容器
18      ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
19      // 获取对象
20      AccountService service =
context.getBean(/*accountService*/AccountService.class);
21
22      @Test
23      public void testFind (){
24          List<Account> all = service.findAll();
25          all.stream().forEach(x-> System.out.println(x));
26      }
27      @Test
28      public void testAdd (){
29          Account account = new Account();
30          account.setId(11);
31          account.setUid(1);
32          account.setMoney(999.0);
33          service.add(account);
34      }
35      @Test
36      public void testUpdate (){
37          Account account = new Account();
38          account.setId(11);
39          account.setUid(1);
```

```
    }
    @Test
    public void testDel (){
        Account account = new Account();
        account.setId(11);
        service.del(account);
    }
}
```

五、Spring纯注解开发

- 纯注解也可以理解为 **零配置**
- 改造目标: 去除所有配置文件
- 核心思想: 找到相应的注解 **代替** 配置文件已经配置文件中的配置项

5.1 替代注解

- 先睹为快 (使用示例)

```
package com.itheima.crud;

import com.alibaba.druid.pool.DruidDataSource;
import com.itheima.crud.dao.AccountDao;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;

/**
 * Spring 配置类.
 * @Configuration: 代替Spring配置文件 (applicationContext.xml)
 * @ComponentScan: 代替<context:component-scan..> value=base-package
 * @PropertySource 代替<context:property-placeholder..> value=location
 * @Import: 代替<import resource=".."/> value=配置类字节码
 */
@Configuration
@ComponentScan("com.itheima.crud")
@PropertySource("classpath:db.properties")
@Import(OtherConfig.class)
public class Config {

    /**
     * @Value: 用于注入配置文件中的参数值
     *  value: 属性值|基本数据类型
     *      ${}: 注入配置文件中参数值
     *      #{}: 注入SpEl表达式结果
     */
    @Value("${db.driver}")
    String driver;
    @Value("${db.url}")
```

```
    String username;
    @Value("${db.password}")
    String password;

    /**
     * @Bean: 用于将方法返回值对象添加到IOC容器
     *  value: 对象名称
     */
    @Bean
    public DataSource druidDataSource(){
        DruidDataSource source = new DruidDataSource();
        source.setDriverClassName(driver);
        source.setUrl(url);
        source.setUsername(username);
        source.setPassword(password);
        return source;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource){
        return new JdbcTemplate(dataSource);
    }
}
```

@Configuration

- 位置: 类
- 作用: 修饰类识别为配置类
- 意义: 代替Spring配置文件 (applicationContext.xml)

@ComponentScan

- 位置: 类
- 作用: 配置注解扫描包
- 意义: 代替<context:component-scan..
- value: 与base-package属性相同

@PropertySource

- 位置: 类
- 作用: 加载配置文件
- 意义: 代替<context:property-placeholder..
- value: 与location属性相同

@Bean

- 位置: 方法
- 作用: 将返回的对象添加到IOC容器
- 意义: 代替<bean标签弥补第三方无法注解缺陷
- value: 对象名称

@Import

- 位置: 类

- value: 其他配置类字节码

5.2 创建容器

- 单元测试

```
import com.itheima.crud.Config;
import com.itheima.crud.entity.Account;
import com.itheima.crud.service.AccountService;
import org.junit.Test;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * 改造内容(根据类型获取service对象).
 *  创建对象注解创建的对象默认以首字母小写的类名为对象名 (accountServiceImpl)
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AnnoTests {

    // 创建IOC容器
    AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(Config.class);
    // 获取对象
    AccountService service =
context.getBean(/*accountService*/AccountService.class);

    @Test
    public void testFind (){
        List<Account> all = service.findAll();
        all.stream().forEach(x-> System.out.println(x));
    }
    @Test
    public void testAdd (){
        Account account = new Account();
        account.setId(11);
        account.setUid(1);
        account.setMoney(999.0);
        service.add(account);
    }
    @Test
    public void testUpdate (){
        Account account = new Account();
        account.setId(11);
        account.setUid(1);
        account.setMoney(0.0);
        service.update(account);
    }
```

```
        Account account = new Account();
        account.setId(11);
        service.del(account);
    }
}
```

六、Spring测试框架

6.1 单元测试的问题

- 每次都需要手动创建IOC容器

6.2 Spring测试框架

- Spring-test整合了Junit框架
- 仅需在测试类上添加两个注解即可自动创建容器
- pom.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">
    <parent>
        <artifactId>spring2</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-day02-ax</artifactId>

    <dependencies>
        <!-- Spring IOC依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
        <!-- Spring JDBC依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
        <!-- ali 连接池依赖 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.12</version>
        </dependency>
        <!-- Mysql 数据库驱动依赖 -->
        <dependency>
```

```
            <version>8.0.15</version>
        </dependency>
        <!-- 单元测试 依赖 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <!-- 添加Spring测试框架 依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
    </dependencies>

</project>
```

@RunWith

- 位置: 类
- 作用: 指定测试框架启动类
- 意义: 代替ApplicationContext实现类

@ContextConfiguration

- 位置: 类
- 作用: 指定配置文件或配置类
- 意义: 配合@RunWith使用代理ApplicationContext实现类

5.3 单元测试示例

```
import com.itheima.crud.Config;
import com.itheima.crud.entity.Account;
import com.itheima.crud.service.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

/**
 * @RunWith: 指定启动器
 * @ContextConfiguration: 指定启动配置
 *  value: 配置文件路径
 *  classes: 配置类路径
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {Config.class})
```

```
     * 使用Spring-test框架
     * 可以直接使用注入注解给属性赋值
     */
    @Autowired
    AccountService service;

    @Test
    public void testFind (){
        List<Account> all = service.findAll();
        all.stream().forEach(x-> System.out.println(x));
    }
}
```

七、代理模式

- 为目标对象提供一个代理对象以便控制外部对目标对象的访问

2.1 静态代理

- 代理类是程序运行前准备好的代理方式被成为 静态代理
- interface: Star.java

```
package com.itheima.proxy;

/**
 * 明星接口.
 *
 * @author : Jason.lee
 */
public interface Star {
    /**
     * 唱歌.
     *
     * @param money the money
     */
    void sing(Integer money);

    /**
     * 拍戏.
     *
     * @param money the money
     */
    void act(Integer money);
}
```

- target: LiuStar.java

```
package com.itheima.proxy;
```

```
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LiuStar implements Star {
    @Override
    public void sing(Integer money) {
        System.out.println("刘德华唱歌真好听，收费: "+money);
    }

    @Override
    public void act(Integer money) {
        System.out.println("刘德华拍戏棒棒哒，收费: "+money);
    }
}
```

- proxy: LiuStarProxy.java

```
package com.itheima.proxy;

/**
 * 刘德华代理人(经纪人).
 *  静态代理:
 *      特点:
 *          1. 需要实现与目标对象相同的接口
 *          2. 需要在内部维护目标对象
 *      缺点:
 *          1. 接口方法较多时，代理实现较麻烦 (可能只需要代理个别方法)
 *          2. 接口扩展方法后，目标对象与代理对象都需要更改
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LiuStarProxy implements Star {

    Star star = new LiuStar();

    @Override
    public void sing(Integer money) {
        if (money > 10000) {
            star.sing(money);
        } else {
            System.out.println("档期忙!");
        }
    }

    @Override
    public void act(Integer money) {
        if (money > 20000) {
            star.act(money);
        } else {
            System.out.println("档期忙!");
        }
    }
}
```

```
package com.itheima.proxy;

import org.junit.Test;

public class LiuStarProxyTest {

    @Test
    public void sing() {
        new LiuStarProxy().sing(10001);
        new LiuStarProxy().act(10001);
    }
}
```

- 静态代理的缺点
  - 接口方法较多时, 代理实现较麻烦 (可能只需要代理个别方法)
  - 接口扩展方法后, 目标对象与代理对象都需要更改

2.2 动态代理

- 代理类在程序运行时创建的代理方式被称为 动态代理

2.2.1 Proxy动态代理

- JDK提供的动态代理API

```
@Test
public void testProxy() {
    // 目标对象
    LiuStar ls = new LiuStar();

    /**
      * 动态创建代理对象:
      *  Proxy: JDK动态代理的Api也是生成的代理对象的超类
      *      loader: 类加载器
      *      interfaces: 代理对象需要实现的接口组
      *      h:  代理对象方法调用的处理程序
      */
    Star star = (Star) Proxy.newProxyInstance(
        // 默认类加载器
        this.getClass().getClassLoader(),
        // 实现的接口组
        new Class[]{Star.class},
        // 调用处理程序
        new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                // 获取方法名
                String name = method.getName();
                // 获取参数值
                Integer arg = (Integer) args[0];
                // 拦截方法
```

```
                    return method.invoke(ls, args);
                }
                return null;
            }
        }
    );

    // 代理类继承了Proxy
    System.out.println(star instanceof Proxy);
    // 代理类实现了Star
    System.out.println(star instanceof Star);
    // 查看类字节码
    System.out.println(star.getClass()); // class com.sun.proxy.$Proxy4

    // 调用代理方法
    star.sing(10001);
    star.act(10001);

}
```

- JDK动态代理也称之为 **接口代理**：代理类实现了与目标对象相同的接口 ($Proxy4 implement Star)

2.2.2 Cglib动态代理

- 第三方框架实现的动态代理

```
@Test
public void testCglib() {

    // 目标对象
    LiuStar ls = new LiuStar();

    /**
      * 动态创建代理对象:
      *  Enhancer: Cglib动态代理的Api
      *      superclass: 超类字节码
      *      interfaces: 代理对象需要实现的接口组
      *      callback:  代理对象方法调用的拦截器
      */
    Star star = (Star) Enhancer.create(
        LiuStar.class,
        new Class[]{ Star.class },
        new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method method, Object[]
objects, MethodProxy methodProxy) throws Throwable {
                // 获取方法名
                String name = method.getName();
                // 获取参数值
                Integer arg = (Integer) objects[0];
                // 拦截方法
                if ((name.equals("sing") && arg > 10000)
                    || (name.equals("act") && arg > 20000)) {
```

```
                return null;
            }
        }
    );

    // 代理类继承了LiuStar
    System.out.println(star instanceof LiuStar);
    // 代理类实现了Star
    System.out.println(star instanceof Star);
    // 查看类字节码
    System.out.println(star.getClass()); // class
com.itheima.proxy.LiuStar$$EnhancerByCGLIB$$13a93339
    // 调用代理方法
    star.sing(10001);
    star.act(10001);
}
```

- Cglib动态代理也称之为 **子类代理**：代理类继承了目标对象 ($$EnhancerByCGLIB extends LiuStar)

2.2.3 动态代理对比

	JDK	CGLIB
实现原理	利用 **实现** 接口方法来拦截目标方法	利用 **继承** 覆写方法来拦截目标方法
实现前提	**需要目标对象实现接口**	需要目标方法未使用final/static修饰
实现效率	JDK1.6以前较Cglib慢; 1.6以及1.7大量调用时较慢	1.8时被JDK代理超越