任务三：Spring JDBCTemplate & 声明式事务

课程任务主要内容：

```
* Spring的JdbcTemplate
* Spring的事务
* Spring集成web环境
```

一 Spring的JdbcTemplate

1.1 JdbcTemplate是什么?

JdbcTemplate是spring框架中提供的一个模板对象，是对原始繁琐的Jdbc API对象的简单封装。

核心对象

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(DataSource dataSource);
```

核心方法

```
int update(); 执行增、删、改语句

List<T>  query(); 查询多个
T queryForObject(); 查询一个
     new BeanPropertyRowMapper<>(); 实现ORM映射封装
```

举个栗子

查询数据库所有账户信息到Account实体中

```
public class JdbcTemplateTest {

    @Test
    public void testFindAll() throws Exception {
        // 创建核心对象
        JdbcTemplate jdbcTemplate = new JdbcTemplate(JdbcUtils.getDataSource());
        // 编写sql
        String sql = "select * from account";
        // 执行sql
```

```
        List<Account> list = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(Account.class));
    }

}
```

1.2 Spring整合JdbcTemplate

需求

基于Spring的xml配置实现账户的CRUD案例

步骤分析

1. 创建java项目，导入坐标
2. 编写Account实体类
3. 编写AccountDao接口和实现类
4. 编写AccountService接口和实现类
5. 编写spring核心配置文件
6. 编写测试代码

1）创建java项目，导入坐标

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.15</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.13</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
</dependencies>
```

2）编写Account实体类

```
public class Account {

    private Integer id;
    private String name;
    private Double money;
}
```

3）编写AccountDao接口和实现类

```
public interface AccountDao {

    public List<Account> findAll();

    public Account findById(Integer id);

    public void save(Account account);

    public void update(Account account);

    public void delete(Integer id);
}
```

```
@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public List<Account> findAll() {
```

```
        // 编写sql
        String sql = "select * from account";
        // 执行sql
        return jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(Account.class));
    }

    @Override
    public Account findById(Integer id) {
        // 编写sql
        String sql = "select * from account where id = ?";
        // 执行sql
        return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>
(Account.class),id);

    }

    @Override
    public void save(Account account) {
        // 编写sql
        String sql = "insert into account values(null,?,?)";
        // 执行sql
        jdbcTemplate.update(sql, account.getName(), account.getMoney());
    }

    @Override
    public void update(Account account) {
        // 编写sql
        String sql = "update account set name = ?,money = ? where id = ?";
        // 执行sql
        jdbcTemplate.update(sql, account.getName(),
account.getMoney(),account.getId());
    }

    @Override
    public void delete(Integer id) {
        // 编写sql
        String sql = "delete from account where id = ?";
        // 执行sql
        jdbcTemplate.update(sql, id);
    }
}
```

4）编写AccountService接口和实现类

```java
public interface AccountService {

    public List<Account> findAll();

    public Account findById(Integer id);

    public void save(Account account);

    public void update(Account account);

    public void delete(Integer id);
}
```

```java
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

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

    @Override
    public Account findById(Integer id) {
        return accountDao.findById(id);
    }

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

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

    @Override
    public void delete(Integer id) {
        accountDao.delete(id);
    }
}
```

5）编写spring核心配置文件

```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
        http://www.springframework.org/schema/context/spring-context.xsd">


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


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


    <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 id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
      <constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
  </bean>
</beans>
```

6）编写测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    //测试保存
    @Test
    public void testSave() {
        Account account = new Account();
        account.setName("lucy");
        account.setMoney(100d);
        accountService.save(account);
    }

    //测试查询
    @Test
    public void testFindById() {
        Account account = accountService.findById(3);
        System.out.println(account);
    }

    //测试查询所有
    @Test
    public void testFindAll() {
        List<Account> accountList = accountService.findAll();
        for (Account account : accountList) {
```

```
            System.out.println(account);
        }
    }

    //测试修改
    @Test
    public void testUpdate() {
        Account account = new Account();
        account.setId(3);
        account.setName("rose");
        account.setMoney(2000d);
        accountService.update(account);
    }

    //测试删除
    @Test
    public void testDelete() {
        accountService.delete(3);
    }
}
```

1.4 实现转账案例

步骤分析

1. 创建java项目，导入坐标
2. 编写Account实体类
3. 编写AccountDao接口和实现类
4. 编写AccountService接口和实现类
5. 编写spring核心配置文件
6. 编写测试代码

1）创建java项目，导入坐标

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.15</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.13</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
</dependencies>
```

2）编写Account实体类

```
public class Account {

    private Integer id;
    private String name;
    private Double money;
    // setter getter....
}
```

3）编写AccountDao接口和实现类

```
public interface AccountDao {

    public void out(String outUser, Double money);

    public void in(String inUser, Double money);

}
```

```
@Repository
public class AccountDaoImpl implements AccountDao {
```

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void out(String outUser, Double money) {
        jdbcTemplate.update("update account set money = money - ? where name =
?", money, outUser);
    }

    @Override
    public void in(String inUser, Double money) {
        jdbcTemplate.update("update account set money = money + ? where name =
?", money, inUser);
    }
}
```

4）编写AccountService接口和实现类

```
public interface AccountService {

    public void transfer(String outUser, String inUser, Double money);
}
```

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Override
    public void transfer(String outUser, String inUser, Double money) {
        accountDao.out(outUser, money);
        accountDao.in(inUser, money);
    }
}
```

5）编写spring核心配置文件

```
<?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
        http://www.springframework.org/schema/context/spring-context.xsd">
```

```
    <!--IOC注解扫描-->
    <context:component-scan base-package="com.lagou"/>

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

    <!--把数据库连接池交给IOC容器-->
    <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>

    <!--把JdbcTemplate交给IOC容器-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
    </bean>


</beans>
```

6）编写测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    @Test
    public void testTransfer() throws Exception {
        accountService.transfer("tom", "jerry", 100d);
    }

}
```

二 Spring的事务

2.1 Spring中的事务控制方式

Spring的事务控制可以分为编程式事务控制和声明式事务控制。

编程式

开发者直接把事务的代码和业务代码耦合到一起，在实际开发中不用。

声明式

开发者采用配置的方式来实现的事务控制，业务代码与事务代码实现解耦合，使用的AOP思想。

2.2 编程式事务控制相关对象【了解】

2.2.1 PlatformTransactionManager

PlatformTransactionManager接口，是spring的事务管理器，里面提供了我们常用的操作事务的方法。

方法	说明
TransactionStatus getTransaction(TransactionDefinition definition);	获取事务的状态信息
void commit(TransactionStatus status);	提交事务
void rollback(TransactionStatus status);	回滚事务

注意：

```
* PlatformTransactionManager 是接口类型，不同的 Dao 层技术则有不同的实现类。

* Dao层技术是jdbcTemplate或mybatis时：
        DataSourceTransactionManager

* Dao层技术是hibernate时：
        HibernateTransactionManager

* Dao层技术是JPA时：
        JpaTransactionManager
```

2.2.2 TransactionDefinition

TransactionDefinition接口提供事务的定义信息（事务隔离级别、事务传播行为等等）

方法	说明
int getIsolationLevel()	获得事务的隔离级别
int getPropogationBehavior()	获得事务的传播行为
int getTimeout()	获得超时时间
boolean isReadOnly()	是否只读

1）事务隔离级别

设置隔离级别，可以解决事务并发产生的问题，如脏读、不可重复读和虚读（幻读）。

```
* ISOLATION_DEFAULT              使用数据库默认级别

* ISOLATION_READ_UNCOMMITTED    读未提交

* ISOLATION_READ_COMMITTED      读已提交

* ISOLATION_REPEATABLE_READ     可重复读

* ISOLATION_SERIALIZABLE        串行化
```

2）事务传播行为

事务传播行为指的就是当一个业务方法【被】另一个业务方法调用时，应该如何进行事务控制。

参数	说明
REQUIRED	如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）
SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）
MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常
REQUERS_NEW	新建事务，如果当前在事务中，把当前事务挂起
NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
NEVER	以非事务方式运行，如果当前存在事务，抛出异常
NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行 REQUIRED 类似的操作

```
* read-only（是否只读）：建议查询时设置为只读

* timeout（超时时间）：默认值是-1，没有超时限制。如果有，以秒为单位进行设置
```

2.2.3 TransactionStatus

TransactionStatus 接口提供的是事务具体的运行状态。

方法	说明
boolean isNewTransaction()	是否是新事务
boolean hasSavepoint()	是否是回滚点
boolean isRollbackOnly()	事务是否回滚
boolean isCompleted()	事务是否完成

可以简单的理解三者的关系：**事务管理器**通过读取**事务定义参数**进行事务管理，然后会产生一系列的**事务状态。**

2.2.4 实现代码

1）配置文件

```
<!--事务管理器交给IOC-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

2）业务层代码

```
@Service
public class AccountServiceImpl implements AccountService {
    @Autowired
    private AccountDao accountDao;

    @Autowired
    private PlatformTransactionManager transactionManager;

    @Override
    public void transfer(String outUser, String inUser, Double money) {
        // 创建事务定义对象
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        // 设置是否只读，false支持事务
        def.setReadOnly(false);
        // 设置事务隔离级别，可重复读mysql默认级别
        def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);
        // 设置事务传播行为，必须有事务
        def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        // 配置事务管理器
        TransactionStatus status = transactionManager.getTransaction(def);

        try {
            // 转账
            accountDao.out(outUser, money);
            accountDao.in(inUser, money);

            // 提交事务
            transactionManager.commit(status);
        } catch (Exception e) {
            e.printStackTrace();
            // 回滚事务
            transactionManager.rollback(status);
        }
    }
}
```

2.2.5 知识小结

Spring中的事务控制主要就是通过这三个API实现的

```
* PlatformTransactionManager 负责事务的管理，它是个接口，其子类负责具体工作

* TransactionDefinition 定义了事务的一些相关参数

* TransactionStatus 代表事务运行的一个实时状态
```

理解三者的关系：**事务管理器**通过读取**事务定义参数**进行事务管理，然后会产生一系列的**事务状态。**

2.3 基于XML的声明式事务控制【重点】

在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。底层采用AOP思想来实现的。

声明式事务控制明确事项：

- 核心业务代码(目标对象) （切入点是谁？）
- 事务增强代码(Spring已提供事务管理器)) （通知是谁？）
- 切面配置（切面如何配置？）

2.3.1 快速入门

需求

使用spring声明式事务控制转账业务。

步骤分析

```
1. 引入tx命名空间

2. 事务管理器通知配置

3. 事务管理器AOP配置

4. 测试事务控制转账业务代码
```

1）引入tx命名空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w2.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/s chema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">
```

```
</beans>
```

2）事务管理器通知配置

```
<!--事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!--通知增强-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--定义事务的属性-->
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

3）事务管理器AOP配置

```
<!--aop配置-->
<aop:config>
    <!--切面配置-->
    <aop:advisor advice-ref="txAdvice"
                 pointcut="execution(* com.lagou.serivce..*.*(..))">
    </aop:advisor>
</aop:config>
```

4）测试事务控制转账业务代码

```
@Override
public void transfer(String outUser, String inUser, Double money) {
    accountDao.out(outUser, money);
    //  制造异常
    int i = 1 / 0;
    accountDao.in(inUser, money);
}
```

2.3.2 事务参数的配置详解

```
<tx:method name="transfer" isolation="REPEATABLE_READ" propagation="REQUIRED"
timeout="-1" read-only="false"/>

* name：切点方法名称

* isolation:事务的隔离级别

* propogation：事务的传播行为

* timeout：超时时间

* read-only：是否只读
```

CRUD常用配置

```
<tx:attributes>
    <tx:method name="save*" propagation="REQUIRED"/>
    <tx:method name="delete*" propagation="REQUIRED"/>
    <tx:method name="update*" propagation="REQUIRED"/>
    <tx:method name="find*" read-only="true"/>
    <tx:method name="*"/>
</tx:attributes>
```

2.3.3 知识小结

```
* 平台事务管理器配置

* 事务通知的配置

* 事务aop织入的配置
```

2.4 基于注解的声明式事务控制【重点】

2.4.1 常用注解

步骤分析

```
1. 修改service层，增加事务注解
2. 修改spring核心配置文件，开启事务注解支持
```

1）修改service层，增加事务注解

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;
```

```
    @Transactional(propagation = Propagation.REQUIRED, isolation =
Isolation.REPEATABLE_READ, timeout = -1, readOnly = false)
    @Override
    public void transfer(String outUser, String inUser, Double money) {
        accountDao.out(outUser, money);
        int i = 1 / 0;
        accountDao.in(inUser, money);
    }
}
```

2）修改spring核心配置文件，开启事务注解支持

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w2.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!--省略之前datsSource、jdbcTemplate、组件扫描配置-->

    <!--事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--事务的注解支持-->
    <tx:annotation-driven/>
</beans>
```

2.4.2 纯注解

核心配置类

```
@Configuration  // 声明为spring配置类
@ComponentScan("com.lagou") // 扫描包
@Import(DataSourceConfig.class) // 导入其他配置类
@EnableTransactionManagement // 事务的注解驱动
public class SpringConfig {

    @Bean
    public JdbcTemplate getJdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
```

```
    }

    @Bean("transactionManager")
    public PlatformTransactionManager getPlatformTransactionManager(@Autowired
DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

}
```

数据源配置类

```
@PropertySource("classpath:jdbc.properties")
public class DataSourceConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;

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

2.4.3 知识小结

```
* 平台事务管理器配置（xml、注解方式）

* 事务通知的配置（@Transactional注解配置）

* 事务注解驱动的配置 <tx:annotation-driven/>、@EnableTransactionManagement
```

三 Spring集成web环境

3.1 ApplicationContext应用上下文获取方式

应用上下文对象是通过 `new ClasspathXmlApplicationContext(spring配置文件)` 方式获取的，但是每次从容器中获得Bean时都要编写 `new ClasspathXmlApplicationContext(spring配置文件)` ，这样的弊端是配置文件加载多次，应用上下文对象创建多次。

解决思路分析：

在Web项目中，可以使用**ServletContextListener**监听Web应用的启动，我们可以在Web应用启动时，就加载Spring的配置文件，创建应用上下文对象**ApplicationContext**，在将其存储到最大的域**servletContext**域中，这样就可以在任意位置从域中获得应用上下文**ApplicationContext**对象了。

3.2 Spring提供获取应用上下文的工具

上面的分析不用手动实现，Spring提供了一个监听器**ContextLoaderListener**就是对上述功能的封装，该监听器内部加载Spring配置文件，创建应用上下文对象，并存储到**ServletContext**域中，提供了一个客户端工具**WebApplicationContextUtils**供使用者获得应用上下文对象。

所以我们需要做的只有两件事：

1. 在web.xml中配置ContextLoaderListener监听器（导入spring-web坐标）
2. 使用WebApplicationContextUtils获得应用上下文对象ApplicationContext

3.3 实现

1）导入Spring集成web的坐标

```
<dependency>
     <dependency>
      <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>5.1.5.RELEASE</version>
    </dependency>
      <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
```

2）配置ContextLoaderListener监听器

```
<!--全局参数-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<!--Spring的监听器-->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

3）通过工具获得应用上下文对象

```
ApplicationContext applicationContext =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
    Object obj = applicationContext.getBean("id");
```