spring

主讲：石小俊

1.介绍spring

业务逻辑组件

1-1 两大原则

- 高低原则
  - 高内聚
  - 低耦合
- 开闭原则
  - 对扩展开
  - 对修改闭

1-2 主要内容

- IOC
  - 使得组件松散
- AOP
  - 使得应用易于扩展
- 通用支持
  - 整合其他框架

2.IOC介绍

2-1 IOC与DI

- IOC
  - Inversion of Control
  - 反转控制
  - 凡是自己所需要使用的对象不是由自己所准备的
  - 而是交给别人来控制器的
  - 由别人控制其所需要的对象
  - 我们所进行的操作是受其他人控制的
- DI
  - 依赖注入
  - 凡是对象的属性的值不是自己所准备的

    - 而是别人注入给他的
    - 这种情况称之为DI

2-2 加载spring容器

- BeanFactory
    - spring核心
    - 用于读取spring容器中的数据
    - 解析xml
- ApplicationContext
    - BeanFactory的子接口
    - ClassPathXmlApplicationContext
        - 根据类加载路径来找
        - 参数可以是字符串
        - 还可以是字符串数组
        - 支持通配符的方式
    - FileSystemXmlApplicationContext
        - 根据文件系统来找

3.IOC使用

3-1 装配

3-1-1简单值装配

基本数据类型、String、包装类型、Class、Resource

```
<!--<bean id="someBean" class="ioc01.SomeBean">-->
    <!--<property name="id">-->
        <!--<value>1</value>-->
    <!--</property>-->
    <!--<property name="name">-->
        <!--<value>admin</value>-->
    <!--</property>-->
    <!--<property name="c">-->
        <!--<value>java.lang.String</value>-->
    <!--</property>-->
<!--</bean>-->


<bean id="someBean" class="ioc01.SomeBean">
    <property name="id" value="2"></property>
    <property name="name" value="alice"></property>
    <property name="c" value="java.lang.Exception"></property>
```

```
</bean>
```

3-1-2 其他bean的引用

```
<!-- 其他bean的引用 -->
<bean id="otherBean" class="ioc02.OtherBean">
    <property name="id" value="1"></property>
    <property name="name" value="other"></property>
</bean>


<bean id="someBean" class="ioc02.SomeBean">
    <property name="age" value="25"></property>
    <!--<property name="otherBean" ref="otherBean"></property>-->
    <property name="otherBean">
        <ref bean="otherBean"></ref>
    </property>
</bean>
```

3-1-3 集合类型装配

List、Set、Map、array、properties

```
<!-- 集合类型装配 -->
<bean id="someBean" class="ioc03.SomeBean">
    <property name="list">
        <list>
            <value>a</value>
            <value>b</value>
            <value>c</value>
        </list>
    </property>
    <property name="arr">
        <array>
            <value>aa</value>
            <value>bb</value>
            <value>cc</value>
        </array>
    </property>
    <property name="set">
        <set>
            <value>aaa</value>
            <value>bbb</value>
            <value>ccc</value>
        </set>
    </property>
    <property name="map">
        <map>
```

```
            <entry key="a">
                <value>1</value>
            </entry>
            <entry key="b">
                <value>2</value>
            </entry>
            <entry key="c">
                <value>3</value>
            </entry>
        </map>
    </property>
    <property name="properties">
        <props>
            <prop key="aa">11</prop>
            <prop key="bb">22</prop>
            <prop key="cc">33</prop>
        </props>
    </property>
</bean>



<bean id="someBean2" class="ioc03.SomeBean">
    <property name="set">
        <list>
            <value>s1</value>
            <value>s2</value>
            <value>s3</value>
        </list>
    </property>
    <property name="list">
        <list>
            <value>l1</value>
            <value>l2</value>
            <value>l3</value>
        </list>
    </property>
    <property name="arr">
        <list>
            <value>a1</value>
            <value>a2</value>
            <value>a3</value>
        </list>
    </property>

</bean>
```

3-2 实例化

实例化-->DI-->初始化-->就绪-->使用-->销毁

3-2-1 初始化销毁方法

```
<bean id="someBean" class="ioc04.SomeBean" init-method="a" destroy-method="b">
    <property name="name" value="admin"></property>
</bean>
```

```
ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("ioc04/applicationContext.xml");

SomeBean someBean = (SomeBean) ac.getBean("someBean");

//主动销毁，当前容器销毁的时候回自动执行对应的销毁方法
//        ac.destroy();
//当虚拟机关闭的时候，销毁当前容器中所有的bean
ac.registerShutdownHook();

//        while(true){
//
//        }
```

3-2-2 实例化途径

- 无参构造函数创建对象

```
<!-- 无参构造函数创建对象 -->
<!--<bean id="someBean" class="ioc05.SomeBean"></bean>-->
```

- 有参构造函数创建对象

```
<!-- 有参构造函数创建对象  -->
<bean id="someBean" class="ioc05.SomeBean">
    <constructor-arg index="1" type="int">
        <value>1</value>
    </constructor-arg>
    <constructor-arg index="2" type="java.lang.String">
        <value>2</value>
    </constructor-arg>
    <constructor-arg index="0" type="java.lang.String">
        <value>3</value>
    </constructor-arg>
</bean>
```

- 静态工厂创建对象

```
<!-- 静态工厂创建对象 -->
<bean id="someService" class="ioc06.SomeServiceFactory" factory-method="getObject"></bean>


<bean id="env" class="java.lang.System" factory-method="getenv"></bean>

<bean id="javaHome" class="java.lang.System" factory-method="getenv">
    <constructor-arg>
        <value>JAVA_HOME</value>
    </constructor-arg>
</bean>
```

- 实例工厂创建对象

```
<!-- 实例工厂创建对象 -->
<bean id="factory" class="ioc07.SomeServiceFactory"></bean>

<!--
    通过工厂获取到对应的SomeService
    factory-bean:表示当前的bean是通过哪一个工厂的bean来获取的
-->
<bean id="someService" factory-bean="factory" factory-method="getObject"></bean>


<!-- 练习  -->
<!-- 首先通过静态工厂获取对象Calender -->
<bean id="calender" class="java.util.Calendar" factory-method="getInstance"></bean>
<!-- 通过实例工厂获取到对应的Date对象 -->
<bean id="date" factory-bean="calender" factory-method="getTime"></bean>
```

3-2-3 实例化时机

- BeanFactory
  - 默认使用到对应的bean的时候才会进行实例化
  - 延迟实例化
- ApplicationContext
  - 默认在解析ioc容器的时候会对当前容器中存在的所有的bean进行实例化
  - 预先实例化
  - 可以更改其实例化的时机
  - lazy-init：是否延迟实例化
    - default:使用默认的
    - true:延迟实例化
    - false:预先实例化

```
<!-- 实例化时机 -->
<bean id="someBean" class="ioc08.SomeBean" lazy-init="true"></bean>
```

3-2-4 组件作用域

默认情况下，组件是单例的

可以通过scope属性改变其组件性质

scope:

- prototype:每次创建都会生成一个新的对象
- singleton:单例

3-3 继承配置

```
<!-- 继承 -->

<!-- 第一种情况:多个bean使用相同的属性，对这些属性注入相同的值 -->
<!-- 首先定义一个虚拟的父类，为该类注入对应的属性值 -->
<bean id="fatherBean" abstract="true">
    <property name="username" value="admin"></property>
</bean>

<!-- 将具体的类继承该虚拟的父类 -->
<bean id="someBean" class="ioc10.SomeBean" parent="fatherBean">
    <property name="username" value="alice"></property>
    <!--<property name="password" value="123456"></property>-->
</bean>

<bean id="otherBean" class="ioc10.OtherBean" parent="fatherBean"></bean>


<!-- 第二种情况:同一个bean，但是拥有多个对象，这些对象中username属性相同，password不同 -->
<bean id="fatherSomeBean" class="ioc10.SomeBean" abstract="true">
    <property name="username" value="admin"></property>
</bean>
<bean id="someBean1" parent="fatherSomeBean">
    <property name="password" value="123"></property>
</bean>
<bean id="someBean2" parent="fatherSomeBean">
    <property name="password" value="321"></property>
</bean>
<bean id="someBean3" parent="fatherSomeBean">
    <property name="username" value="alice"></property>
    <property name="password" value="123456"></property>
</bean>
```

3-4 自动装配

autowire:

- byType:根据bean的类型自动进行注入操作
  - 通过属性的类型在ioc容器中查找与其类型一致的bean
  - 将该bean的值注入到对应的属性中
- byName:根据bean的id来查找
  - 通过属性名去ioc容器中查找与其名字一致的bean的id
  - 将该bean的值注入到对应的属性中

```
<bean id="otherBean" class="ioc11.OtherBean">
    <property name="name" value="admin"></property>
</bean>

<bean id="ob" class="ioc11.OtherBean">
    <property name="name" value="alice"></property>
</bean>

<!--<bean id="someBean" class="ioc11.SomeBean">-->
    <!--<property name="otherBean" ref="otherBean"></property>-->
<!--</bean>-->
<!--<bean id="someBean" class="ioc11.SomeBean" autowire="byType"></bean>-->
<bean id="someBean" class="ioc11.SomeBean" autowire="byName"></bean>
```

3-5 Resource

```
<bean id="someBean" class="ioc12.SomeBean">
    <property name="resource" value="classpath:ioc12/dataSource.properties"></property>
</bean>
```

```
package ioc12;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import java.io.*;
```

```
/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/15 14:44
 * Description:
 * version:1.0
 */
public class Test {

    public static void main(String[] args) throws IOException {
//        File file = new File("ioc12/a.txt");

//        Resource resource = new ClassPathResource("ioc12/a.txt");
//        System.out.println(resource.exists());
//        System.out.println(resource.getFilename());
//
//        InputStream in = resource.getInputStream();
//        BufferedReader br = new BufferedReader(new InputStreamReader(in,"utf-8"));
//        System.out.println(br.readLine());


        ApplicationContext ac = new
ClassPathXmlApplicationContext("ioc12/applicationContext.xml");
//        Resource resource = ac.getResource("ioc12/a.txt");
//        Resource resource = ac.getResource("classpath:ioc12/a.txt");
//        InputStream in = resource.getInputStream();
//        BufferedReader br = new BufferedReader(new InputStreamReader(in,"utf-8"));
//        System.out.println(br.readLine());


        SomeBean someBean = (SomeBean) ac.getBean("someBean");
        Resource resource = someBean.getResource();
        InputStream in = resource.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(in,"utf-8"));
        System.out.println(br.readLine());


    }

}
```

3-6 在bean中取出当前容器

实现ApplicationContextAware接口

重写setApplicationContext方法

实现了注入的操作，注入的就是我们所使用的容器

```
package ioc13;
```

```
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/15 14:57
 * Description:
 * version:1.0
 */
public class SomeBean implements ApplicationContextAware {

    private ApplicationContext ac;


    public void doSome(){
//        ApplicationContext ac = new
ClassPathXmlApplicationContext("ioc13/applicationContext.xml");
        System.out.println("doSome ac:"+ac);
        //此时发现取出来的容器与使用的容器并不是同一个
        //此处通过new出来的是一个新的容器，并不是原来的
        //那怎样才能取出与使用的容器一致的对象
    }

    public void setApplicationContext(ApplicationContext ac) throws BeansException {
        this.ac=ac;
    }
}
```

3-7 FactoryBean

该bean本质上是一个工厂

它并不是用来创建自己的

而是用于创建另外一个bean

在某些时候我们获取某个对象的过程比较复杂

而中途所涉及的所有的对象我们并不关心

我们所关心只是最终生成的对象

当遇到这种情况我们可以选择使用FactoryBean

开发步骤

- 创建一个java类，用于生产某个的bean的过程

    - 实现FactoryBean接口
    - 重写其中的三个方法
        - getObject
        - getObjectType
        - isSingleton
- 配置ioc
    - 配置的bean是最终所需要的bean
    - 其class指向的是对应的FactoryBean

```
<bean id="parser" class="ioc14.SAXParserFactoryBean"></bean>

<bean id="date" class="ioc14.DateFactoryBean">
    <property name="year" value="2018"></property>
    <property name="month" value="01"></property>
    <property name="day" value="01"></property>
</bean>
```

SAXParserFactoryBean

```
package ioc14;

import org.springframework.beans.factory.FactoryBean;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/15 15:53
 * Description:
 * version:1.0
 */
public class SAXParserFactoryBean implements FactoryBean {
    /**
     * 创建某个对象的过程
     * @return
     * @throws Exception
     */
    public Object getObject() throws Exception {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        return parser;
    }

    /**
     * 创建的对象的类型
     * @return
     */
```

```
    public Class getObjectType() {
        return SAXParser.class;
    }

    /**
     * 该对象是否是单例的
     * @return
     */
    public boolean isSingleton() {
        return false;
    }
}
```

DateFactoryBean

```
package ioc14;

import org.springframework.beans.factory.FactoryBean;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/15 16:40
 * Description:
 * version:1.0
 */
public class DateFactoryBean implements FactoryBean {

    private String year;
    private String month;
    private String day;

    public String getYear() {
        return year;
    }

    public void setYear(String year) {
        this.year = year;
    }

    public String getMonth() {
        return month;
    }

    public void setMonth(String month) {
        this.month = month;
    }
```

```
    public String getDay() {
        return day;
    }

    public void setDay(String day) {
        this.day = day;
    }

    public Object getObject() throws Exception {
        Date date = new SimpleDateFormat("yyyyMMdd").parse(year+month+day);
        return date;
    }

    public Class<?> getObjectType() {
        return Date.class;
    }

    public boolean isSingleton() {
        return true;
    }
}
```

3-8 后处理bean

对当前容器中的所有bean做一个后处理

后处理bean并不是针对某一个bean

而是针对的是整个容器

实现BeanPostProcessor接口

实例化-->DI-->-->postProcessBeforeInitialization-->初始化-->postProcessAfterInitialization-->就绪-->使用-->销毁

```
package ioc15;

import com.sun.org.apache.bcel.internal.generic.RET;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 9:13
 * Description:
 * version:1.0
 */
public class SomeBeanPostProcessor implements BeanPostProcessor {
```

```
    //实例化-->DI-->-->postProcessBeforeInitialization-->初始化-->postProcessAfterInitialization-
->就绪-->使用-->销毁
    //在初始化之前做后处理
    public Object postProcessBeforeInitialization(Object bean, String id) throws BeansException
{
        return bean;
    }

    /**
     * 初始化之后做后处理
     * @param bean  正在做后处理的bena是谁
     * @param id    正在做后处理的bean的id是多少
     * @return
     * @throws BeansException
     */
    public Object postProcessAfterInitialization(Object bean, String id) throws BeansException {
        //由于后处理bean是对于整个容器中所有的bean做后处理
        //而此时我们只希望对SomeBean做后处理
        //怎么才能知道当前正在处理的是SomeBean
        //方式一
//        if("someBean".equals(id)){
//
//        }
        //方式二
        if(bean instanceof SomeBean){
            SomeBean someBean = (SomeBean) bean;
            someBean.setName(someBean.getName().toUpperCase());
        }
        if("otherBean".equals(id)){
            OtherBean otherBean = (OtherBean) bean;
            otherBean.setName(otherBean.getName().toLowerCase());
        }
        return bean;
//        return "admin";
    }
}
```

```
<bean id="someBean" class="ioc15.SomeBean">
    <property name="name" value="adMIn"></property>
</bean>

<bean id="otherBean" class="ioc15.OtherBean">
    <property name="name" value="Assaq"></property>
</bean>

<!-- 配置后处理bean -->
<bean class="ioc15.SomeBeanPostProcessor"></bean>
```

3-9 访问properties文件

```
<!-- 配置读取properties文件 -->
    <!-- 方式一:使用后处理bean来实现 -->
    <!--<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">-->
        <!--<property name="location" value="classpath:ioc16/bean.properties"></property>-->
    <!--</bean>-->
    <!-- 方式二:使用context命名空间的方法-->
    <context:property-placeholder location="classpath:ioc16/bean.properties"></context:property-
placeholder>


    <bean id="someBean" class="ioc16.SomeBean">
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${password}"></property>
    </bean>
```

3-10 属性编辑

将简单值与对象进行互相转换

编写转换规则

转换规则

- 编写一个java类
  - 继承PropertyEditorSupport
  - 重写setAsText/getAsText
    - setAsText:将字符串转换为对象
    - getAsText:将对象转换为字符串
  - 通过setValue/getValue获取对象
    - setValue():设置转换后的对象
    - getValue():获取需要转换的对象
- 注册转换规则
  - 通过spring所提供一个后处理bean进行注册
  - org.springframework.beans.factory.config.CustomEditorConfigurer

```
<bean id="address" class="ioc17.Address">
    <property name="province" value="江苏省"></property>
    <property name="city" value="南京市"></property>
</bean>

<bean id="user" class="ioc17.User">
    <property name="id" value="1"></property>
    <property name="username" value="admin"></property>
    <property name="password" value="123456"></property>
    <property name="phone" value="13812345678"></property>
```

```
    <property name="address" value="江苏省-南京市"></property>
    <property name="birthday" value="1990-01-01"></property>
    <!--<property name="address" ref="address"></property>-->
</bean>



<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <!--
                key:表示的是遇到什么类型的时候需要来找该转换规则
                value:转换规则是谁
             -->
            <!-- Spring4以下版本用法 -->
            <!--<entry key="ioc17.Address">-->
                <!--<bean class="ioc17.AddressEditor"></bean>-->
            <!--</entry>-->
            <!-- spring4版本用法 -->
            <entry key="ioc17.Address" value="ioc17.AddressEditor"></entry>
            <entry key="java.util.Date" value="ioc17.DateEditor"></entry>
        </map>
    </property>
</bean>
```

```
package ioc17;

import java.beans.PropertyEditorSupport;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 10:33
 * Description:
 * version:1.0
 */
public class AddressEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        //想要将一个字符串转换为指定的对象
        //首先该字符串需要遵循某种规则
        //假设此时字符串格式为xxx-yyy
        //当将其转换为Address的时候
        //xxx表示省份
        //yyy表示城市
        String[] arr = text.split("-");
        Address address = new Address();
        address.setProvince(arr[0]);
        address.setCity(arr[1]);
        setValue(address);
```

```
    }

}
```

```
package ioc17;

import java.beans.PropertyEditorSupport;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 11:50
 * Description:
 * version:1.0
 */
public class DateEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        //yyyy-MM-dd
        Date date = null;
        try {
            date = new SimpleDateFormat("yyyy-MM-dd").parse(text);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        setValue(date);
    }
}
```

4.AOP

Aspect Oriented Programming

面向切面编程

将应用中所需要使用的交叉业务逻辑提取出来封装成切面

由AOP容器在适当的时机

将这些切面动态的织入到具体的业务逻辑中

4-1 目的

- 将具体的核心业务逻辑与交叉业务逻辑相分离
    - 解耦和
- 切面的复用
    - 重复使用切面(某种交叉业务逻辑)
- 独立模块化
- 在不改变原有代码的基础上，增加新的功能

4-2 原理

AOP采用的就是动态代码的模式

动态代理就是根据目标类的类加载器、代理的接口、交叉业务逻辑代码

生成对应的代理类

LogInvocationHandler

```
package aop02;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 13:51
 * Description:
 * version:1.0
 */
public class LogInvocationHandler implements InvocationHandler {

    private Object target;

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

    /**
     *
     * @param proxy 代理对象,没用
     * @param method 目标方法，当前正在调用的方法
     * @param args 方法的参数列表
     * @return 目标方法的返回值
     * @throws Throwable
     */
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println(method.getName()+"方法执行时间:"+new SimpleDateFormat("yyyy年MM月dd日
HH:mm:sss").format(new Date()));
```

```
        return method.invoke(target,args);
    }
}
```

Test

```
package aop02;

import java.lang.reflect.Proxy;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 13:49
 * Description:
 * version:1.0
 */
public class Test {

    public static void main(String[] args) {
        SomeService someService = (SomeService) Proxy.newProxyInstance(
                SomeServiceImpl.class.getClassLoader(),
                SomeServiceImpl.class.getInterfaces(),
                new LogInvocationHandler(new SomeServiceImpl())
        );
        someService.doSome();

        someService.doOther();


        OtherService otherService = (OtherService) Proxy.newProxyInstance(
                OtherServiceImpl.class.getClassLoader(),
                OtherServiceImpl.class.getInterfaces(),
                new LogInvocationHandler(new OtherServiceImpl())
        );
        otherService.a();
        otherService.b();

    }

}
```

4-3 AOP1.X

通知类型	描述	接口
前置通知	在执行业务方法之前执行	MethodBeforeAdvice
后置通知	在业务方法正常执行结束之后执行	AfterReturningAdvice
异常通知	在业务方法执行出错时执行	ThtowsAdvice
环绕通知	包含以上三种	MethodInterceptor

4-3-1 前置通知

```
package aop03;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 14:57
 * Description:
 * version:1.0
 */
public class LogAdvice implements MethodBeforeAdvice {


    /**
     *
     * @param method  目标方法
     * @param objects  方法的参数列表
     * @param o 目标类
     * @throws Throwable
     */
    public void before(Method method, Object[] objects, Object o) throws Throwable {
        System.out.println(method.getName()+"方法执行时间:"+new SimpleDateFormat("yyyy年MM月dd日
HH:mm:ss").format(new Date()));
    }
}
```

4-3-2 后置通知

```
package aop03;

import org.springframework.aop.AfterReturningAdvice;

import java.lang.reflect.Method;
```

```
/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 15:03
 * Description:
 * version:1.0
 */
public class WelcomeAdvice implements AfterReturningAdvice {

    /**
     * 后置通知
     * @param returnObject  返回值
     * @param method    目标方法
     * @param objects   方法参数列表
     * @param target    目标类
     * @throws Throwable
     */
    public void afterReturning(Object returnObject, Method method, Object[] objects, Object
target) throws Throwable {
        System.out.println("欢迎您,"+method.getName());
    }
}
```

4-3-3 异常通知

```
package aop03;

import org.springframework.aop.ThrowsAdvice;

import java.lang.reflect.Method;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 15:08
 * Description:
 * version:1.0
 */
public class SomeExceptionAdvice implements ThrowsAdvice {

    //当前实现ThrowsAdvice接口的时候，并没有提示我们要重写的方法
    //原因是该接口存在多个方法，而这些方法的方法名是一致的
    //即存在方法的重载
    //这些方法我们只需要实现其中任意一个就行
    //所以并没有提示我们重写哪一个，需要我们自己去选择

    /**
     * 异常通知
     * @param method    目标方法
     * @param args      方法的参数列表
     * @param target    目标类
     * @param ex        异常类型
     */
```

```
    public void afterThrowing(Method method, Object[] args, Object target, Exception ex){
        System.out.println(target.getClass().getName()+"类"+method.getName()+"方法执行出错,异常原因:"+ex);
    }

}
```

4-3-4 环绕通知

```
package aop03;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

import java.lang.reflect.Method;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/16 16:00
 * Description:
 * version:1.0
 */
public class AroundAdvice implements MethodInterceptor {

    /**
     * 环绕通知
     * @param mi    用于获取与目标方法相关的信息
     * @return      返回的是目标方法具体的返回值
     * @throws Throwable
     */
    public Object invoke(MethodInvocation mi) throws Throwable {

        //获取目标类
        Object target = mi.getThis();
        //获取目标方法
        Method method = mi.getMethod();
        //获取参数列表
        Object[] args = mi.getArguments();

        System.out.println("环绕通知之前置");
        Object result = null;
        long begin = System.currentTimeMillis();
        try {
            //执行目标方法
            result = mi.proceed();

            long end = System.currentTimeMillis();
            System.out.println("环绕通知之后置");

            System.out.println("执行方法共花费了:"+(end-begin)+"毫秒");
```

```
        } catch (Throwable throwable) {
            System.out.println("环绕通知之异常通知");
            long end = System.currentTimeMillis();
            System.out.println("执行方法共花费了:"+(end-begin)+"毫秒");
        }


        return result;
    }
}
```

4-3-5 通知的配置

```
<bean id="someServiceTarget" class="aop03.SomeServiceImpl"></bean>

<!-- 前置通知 -->
<bean id="logAdvice" class="aop03.LogAdvice"></bean>

<!-- 后置通知 -->
<bean id="welcomeAdvice" class="aop03.WelcomeAdvice"></bean>

<!-- 异常通知 -->
<bean id="someExceptionAdvice" class="aop03.SomeExceptionAdvice"></bean>

<!-- 环绕通知 -->
<bean id="aroundAdvice" class="aop03.AroundAdvice"></bean>


<!-- 通过spring将通知注入到具体的目标类中 -->
<bean id="someService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <!-- 指定目标类 -->
    <property name="target" ref="someServiceTarget"></property>
    <!-- 代理的接口 -->
    <property name="proxyInterfaces" value="aop03.SomeService"></property>
    <!-- 交叉业务逻辑代码 -->
    <property name="interceptorNames">
        <list>
            <value>logAdvice</value>
            <value>welcomeAdvice</value>
            <value>someExceptionAdvice</value>
            <value>aroundAdvice</value>
        </list>
    </property>
</bean>
```

4-4 AOP2.X

4-4-1 特点

- 非倾入性
- 完全基于配置
- 引入了切点表达式
- 引入了新的命名空间

4-4-2 使用方式

- xml配置
- 注解

4-4-3 通知类型

- 前置通知
- 后置通知
  - 正常返回通知
  - 异常通知
  - 后置通知
- 环绕通知
  - 对于方法有要求
    - 方法必须有返回值，表示的是目标方法的返回值
    - 方法必须有参数，用于获取所需要的资源
    - 需要抛出Throwable

无参数的前置与后置通知

```
package aop04;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 10:28
 * Description:
 * version:1.0
 */
public class LogAdvice {
    public void a(){
        System.out.println("前置通知");
    }
    public void b(){
        System.out.println("正常返回的后置通知");
    }
    public void c(){
        System.out.println("异常通知");
    }
```

```
    public void d(){
        System.out.println("后置通知");
    }
}
```

环绕通知

```
package aop04;

import org.aspectj.lang.ProceedingJoinPoint;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 11:44
 * Description:
 * version:1.0
 */
public class AroundAdvice {

    public Object around(ProceedingJoinPoint jp) throws Throwable{
        //目标类
        Object target = jp.getThis();
        //目标方法名
        String methodName = jp.getSignature().getName();
        //目标的方法参数列表
        Object[] args = jp.getArgs();

        Object result = null;
        System.out.println("环绕通知之前置通知");
        long begin = System.currentTimeMillis();

        try {
            result = jp.proceed();
            long end = System.currentTimeMillis();

            System.out.println("环绕通知之正常返回通知");
            System.out.println("执行目标类"+target+"中的目标方法"+methodName+"共花费了"+(end-
begin)+"毫秒");
        } catch (Throwable throwable) {
            long end = System.currentTimeMillis();

            System.out.println("环绕通知之异常通知");
            System.out.println("执行目标类"+target+"中的目标方法"+methodName+"共花费了"+(end-
begin)+"毫秒");
        }

        return result;
    }

}
```

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


    <!-- AOP2.X -->

    <!-- 目标类 -->
    <bean id="someService" class="aop04.SomeServiceImpl"></bean>

    <bean id="otherService" class="aop04.OtherServiceImpl"></bean>

    <!-- 通知 -->
    <bean id="logAdvice" class="aop04.LogAdvice"></bean>

    <bean id="aroundAdvice" class="aop04.AroundAdvice"></bean>

    <!--
        引入新的命名空间
        通过切点表达式对符合规则的bean做处理
    -->
    <aop:config>
        <!--
            配置切点表达式
            用于配置切入点
            id属性:当前切点表达式的唯一性标识符
            expression属性:切点表达式的内容
        -->
        <!--<aop:pointcut id="pc1" expression="within(aop04.SomeServiceImpl)"></aop:pointcut>-->
        <!--<aop:pointcut id="pc2" expression="within(aop04.OtherServiceImpl)"></aop:pointcut>--
>


        <!--<aop:pointcut id="pc3" expression="execution(void aop04.SomeServiceImpl.doSome())">
</aop:pointcut>-->
        <!--<aop:pointcut id="pc3" expression="execution(* aop04.*Impl.*(..))"></aop:pointcut>--
>
        <!--<aop:pointcut id="pc3" expression="execution(* aop04.*Impl.doSome(..))">
</aop:pointcut>-->
        <!--<aop:pointcut id="pc3" expression="execution(* aop04.SomeServiceImpl.doSome(..))">
</aop:pointcut>-->
        <!--<aop:pointcut id="pc3" expression="execution(* aop04.SomeServiceImpl.doSome(..))">
</aop:pointcut>-->
        <!--<aop:pointcut id="pc3" expression="execution(* aop04.SomeServiceImpl.doSome(..)) or
execution(* aop04.OtherServiceImpl.doOther(..))"></aop:pointcut>-->
```

```
        <aop:pointcut id="pc3" expression="execution(* aop04.*Impl.*(..)) and not execution(*
aop04.SomeServiceImpl.doSome(..))"></aop:pointcut>

        <!--
            配置切面
            即配置通知
            ref属性:当前使用的通知是谁
            子标签类型:
                before:配置前置通知
                after:配置后置通知
                after-returning:配置正常返回通知
                after-throwing:配置异常通知
                    method属性:该通知下的哪一个方法作为指定类型的通知
                    pointcut-ref属性:该通知是为哪个切面服务
        -->
        <!--<aop:aspect ref="logAdvice">-->
            <!--<aop:before method="a" pointcut-ref="pc1"></aop:before>-->
            <!--<aop:after-returning method="b" pointcut-ref="pc1"></aop:after-returning>-->
            <!--<aop:after-throwing method="c" pointcut-ref="pc1"></aop:after-throwing>-->
            <!--<aop:after method="d" pointcut-ref="pc1"></aop:after>-->
        <!--</aop:aspect>-->
        <!--<aop:aspect ref="logAdvice">-->
            <!--<aop:before method="a" pointcut-ref="pc2"></aop:before>-->
            <!--<aop:after-returning method="b" pointcut-ref="pc2"></aop:after-returning>-->
            <!--<aop:after-throwing method="c" pointcut-ref="pc2"></aop:after-throwing>-->
            <!--<aop:after method="d" pointcut-ref="pc2"></aop:after>-->
        <!--</aop:aspect>-->


        <!--<aop:aspect ref="logAdvice">-->
            <!--<aop:before method="a" pointcut-ref="pc3"></aop:before>-->
            <!--<aop:after-returning method="b" pointcut-ref="pc3"></aop:after-returning>-->
            <!--<aop:after-throwing method="c" pointcut-ref="pc3"></aop:after-throwing>-->
            <!--<aop:after method="d" pointcut-ref="pc3"></aop:after>-->
        <!--</aop:aspect>-->

        <!--
            around:环绕通知
        -->
        <!--<aop:aspect ref="aroundAdvice">-->
            <!--<aop:around method="around" pointcut-ref="pc3"></aop:around>-->
        <!--</aop:aspect>-->
        <!--<aop:aspect ref="aroundAdvice">-->
            <!--<aop:around method="around" pointcut-ref="pc4"></aop:around>-->
        <!--</aop:aspect>-->


        <aop:aspect ref="aroundAdvice">
            <aop:around method="around" pointcut-ref="pc3"></aop:around>
        </aop:aspect>

    </aop:config>
```

```
</beans>
```

有参数的前置与后置通知

```
package aop05;


import org.aspectj.lang.JoinPoint;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 13:31
 * Description:
 * version:1.0
 */
public class LogAdvice {

    public void before(JoinPoint jp){
        Object[] args = jp.getArgs();
        System.out.println(jp.getSignature().getName()+"方法即将执行,方法的参数数量
为:"+args.length);
    }

    public void afterReturning(JoinPoint jp,Object result){
        System.out.println(jp.getSignature().getName()+"方法正常执行完毕，方法的返回值为:"+result);
    }

    public void afterThrowing(JoinPoint jp,Exception e){
        System.out.println(jp.getSignature().getName()+"方法执行出错,异常为:"+e);
    }

    public void after(JoinPoint jp){
        System.out.println(jp.getSignature().getName()+"执行完毕");
    }



}
```

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


    <bean id="someService" class="aop05.SomeServiceImpl"></bean>

    <bean id="logAdvice" class="aop05.LogAdvice"></bean>

    <bean id="aroundAdvice" class="aop05.AroundAdvice"></bean>

    <aop:config>
        <aop:pointcut id="pc1" expression="within(aop05.SomeServiceImpl)"></aop:pointcut>

        <aop:aspect ref="aroundAdvice" order="2">
            <aop:around method="around" pointcut-ref="pc1"></aop:around>
        </aop:aspect>

        <aop:aspect ref="logAdvice" order="1">
            <aop:before method="before" pointcut-ref="pc1"></aop:before>
            <aop:after-returning method="afterReturning" pointcut-ref="pc1" returning="result">
</aop:after-returning>
            <aop:after-throwing method="afterThrowing" pointcut-ref="pc1" throwing="e">
</aop:after-throwing>
            <aop:after method="after" pointcut-ref="pc1"></aop:after>
        </aop:aspect>



    </aop:config>


</beans>
```

4-4-4 切点表达式

- within
  - 匹配指定的类中的所有方法
  - within(包名.类名)
- execution
  - 匹配某些类中的某些方法
  - execution(返回值类型 包名.类名.方法名(参数类型列表))
  - 且支持通配符
    - *
      - 第一种用法:匹配任意一个参数

        - 第二种用法:匹配0或多个字符
    - ..
        - 表示匹配0或者多个参数
  - 支持连接条件
    - and:且的意思,多个条件同时满足才能执行
    - or:或的意思,多个条件只要满足任意一个就能执行
    - not:非的意思,只要不满足对应条件就可以执行
        - 要求:**not前必须有空格**

4-4-5 切面优先级

默认情况下,谁在前，谁的优先级高

可以通过order熟悉控制优先级

值为数字，值越小，优先级越高

5.annotation

spring注解是从spring2.5开始的

真正的spring开发中不会全部使用注解

而是通过配置文件+注解的形式

对于总的配置可以使用配置文件

对于细节配置可以使用注解

当然，目前已经逐步进入到无配置时代

5-1 IOC注解

- 普通bean
  - @Component
    - 无参数时，表示配置当前bean，bean的id即为当前类的类名，首字母小写
    - 有参数时，表示配置当前bean，bean的id即为参数中的数据
        - 例如:@Component("someBean")
        - 相当于
- 持久层bean
  - @Repository
    - 用法同Component
- 业务层bean
  - @Service

        - 用法同Component
- 控制器bean
    - @Controller
        - 详见springmvc
- 注入值
    - @Value
- 自动装配
    - @Autowired
        - 根据类型或者bean的id
        - 只要有一个符合就行
    - @Autowired@Qualifier("bean的id")
        - 只根据bean的id来找

```
package bean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 14:18
 * Description:
 * version:1.0
 */
@Component
//@Component("aa")
public class SomeBean {

    @Autowired
//    @Qualifier("ob")
    private OtherBean ob;

    public OtherBean getOb() {
        return ob;
    }

    public void setOb(OtherBean ob) {
        this.ob = ob;
    }
}
```

```
package bean;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
```

```java
/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 14:18
 * Description:
 * version:1.0
 */
@Component
public class OtherBean {

//    @Value("1")
//    private int id;
//    @Value("admin")
//    private String name;


//    private int id = 2;
//    private String name = "alice";


    @Value("${id}")
    private int id;
    @Value("${name}")
    private String name;


    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```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">
```

```
    <!--<bean id="otherBean" class="bean.OtherBean">-->
        <!--<property name="id" value="1"></property>-->
        <!--<property name="name" value="admin"></property>-->
    <!--</bean>-->
    <!--<bean id="someBean" class="bean.SomeBean">-->
        <!--<property name="otherBean" ref="otherBean"></property>-->
    <!--</bean>-->

    <!-- 配置文件+注解的形式简化开发 -->

    <!-- 扫包 -->
    <context:component-scan base-package="bean"></context:component-scan>

    <!-- 访问properties文件 -->
    <!--<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">-->
        <!--<property name="location" value="classpath:test.properties"></property>-->
    <!--</bean>-->

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


</beans>
```

5-2 AOP注解

- @Aspect
  - 将该类定义为切面
- @Pointcut(切点表达式)
  - 定义切点表达式
- Before
  - 定义前置同
- @After
  - 定义后置通知
- @AfterReturning
  - 定义正常返回通知
- @ArterThrowing
  - 定义异常通知
- @Around
  - 定义环绕通知

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

```java
package advice;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/19 15:10
 * Description:
 * version:1.0
 */
@Component
@Aspect
public class LogAdvice {

    //配置切点,切点对应也是一个方法
    @Pointcut(value = "execution(* bean.*.*(..))")
//    @Pointcut("within(bean.SomeService)")
    public void pointcutName(){}

    @Before(value = "pointcutName()")
    public void before(JoinPoint jp){
        System.out.println(jp.getSignature().getName()+"方法即将执行");
    }

    @After("pointcutName()")
    public void after(JoinPoint jp){
        System.out.println(jp.getSignature().getName()+"方法执行完毕");
    }

    @AfterReturning(value="pointcutName()",returning = "result")
    public void afterReturning(JoinPoint jp,Object result){
        System.out.println("方法正常返回,返回值为:"+result);
    }

    @AfterThrowing(value = "pointcutName()",throwing = "e")
    public void afterThrowing(JoinPoint jp, Exception e){
        System.out.println("方法执行出错,异常为:"+e);
    }


    @Around("pointcutName()")
    public Object around(ProceedingJoinPoint jp) throws Throwable{
        //目标类
        Object target = jp.getThis();
        //目标方法名
        String methodName = jp.getSignature().getName();
        //目标的方法参数列表
        Object[] args = jp.getArgs();

        Object result = null;
```

```
        System.out.println("环绕通知之前置通知");
        long begin = System.currentTimeMillis();

        try {
            result = jp.proceed();
            long end = System.currentTimeMillis();

            System.out.println("环绕通知之正常返回通知");
            System.out.println("执行目标类"+target+"中的目标方法"+methodName+"共花费了"+(end-
begin)+"毫秒");
        } catch (Throwable throwable) {
            long end = System.currentTimeMillis();

            System.out.println("环绕通知之异常通知");
            System.out.println("执行目标类"+target+"中的目标方法"+methodName+"共花费了"+(end-
begin)+"毫秒");
        }

        return result;
    }


}
```

6.通用支持

6-1 整合jdbc

配置数据源

- 使用spring提供的DriverManagerDataSource

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?
useUnicode=true&amp;characterEncoding=utf-8"></property>
    <property name="username" value="root"></property>
    <property name="password" value=""></property>
</bean>
```

- 使用第三方连接池BasicDataSource

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <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>
    <!-- 除了以上通用配置之外，我们还有一个可能会用到的额外配置 -->
```

```
    <!-- 最大连接数 -->
    <property name="maxActive" value="1"></property>
    <!-- 初始化连接数 -->
    <property name="initialSize" value="1"></property>
    <!-- 最长等待时间 -->
    <property name="maxWait" value="3000"></property>

</bean>
```

实现JDBC支持

- 方式一
  - 使用的是Spring所提供的模板类JdbcTemplate
  - dataSource-->JdbcTemplate-->dao

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

<!-- UserDao -->
<bean id="userDao" class="dao.impl.UserDaoImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
```

- 方式二
  - 使用JdbcDaoSupport
  - dataSource-->dao
  - 通过getJdbcTemplate()获取JdbcTemplae对象

```
<bean id="userDao2" class="dao.impl.UserDaoImpl2">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

6-2 整个mybatis

```
<?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">


    <!-- 配置数据源 -->
    <context:property-placeholder location="classpath:dataSource.properties"></context:property-
placeholder>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <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mybatis -->
    <!--
        配置SqlSessionFactory
        SqlSessionFactoryBean
     -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 此处配置一切与mybatis相关的操作 -->
        <!-- 注入数据源 -->
        <property name="dataSource" ref="dataSource"></property>
        <!-- 配置别名 -->
        <!--<property name="typeAliasesPackage" value="entity"></property>-->
        <!-- 注册mapper文件 -->
        <property name="mapperLocations">
            <list>
                <!--<value>classpath:mapper/UserMapper.xml</value>-->
                <!-- 支持通配符的方式 -->
                <value>classpath:mapper/*.xml</value>
            </list>
        </property>
        <!-- 引入mybatis-config.xml文件 -->
        <!--<property name="configLocation" value="classpath:mybatis-config.xml"></property>-->
    </bean>

    <!-- 配置UserMapper -->
    <!-- 方式一:使用FactoryBean -->
    <!--<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">-->
        <!--<property name="sqlSessionFactory" ref="sqlSessionFactory"></property>-->
        <!--<property name="mapperInterface" value="dao.UserMapper"></property>-->
    <!--</bean>-->

    <!--
        方式二:执行别名操作
        通过spring提供的后处理bean对接口所在的包做扫包操作
        最终将该包下的所有的类名作为当前bean的id
        首字母小写
    -->
```

```
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="dao"></property>
    </bean>



</beans>
```

6-3 事务支持

传统数据库事务的特性

ACID

- A：原子性
- C：一致性
- I：独立性(隔离性)
- D：持久性

Spring使用的事务类型

- jdbc/mybatis
  - DataSourceTransactionManager
- Hibernate
  - HibernateTransactionManager

事务属性

- 传播行为
  - 传播规则
    - REQUIRED
      - 表示当前方法必须运行在一个有事务所管理的环境中
      - 如果当前方法有一个事务正在执行
        - 那么该方法会在当前事务环境中执行
      - 如果当前方法没有一个事务正在执行
        - 则会开启一个新的事务
      - 简单来讲
        - 有事务，加入
        - 没有事务，新建
        - 一般用于保存操作
    - SUPPORTS
      - 表示当前方法不一定运行在一个有事务所管理的环境中
      - 如果当前方法有一个事务正在执行

        - 阿么该方法会加入到当前的环境中
    - 如果当前方法没有一个事务正在执行
        - 那么就拉倒
    - 简单来讲
        - 有事务，加入
        - 没有事务，拉倒
        - 一般用于查询操作
- 回滚条件
    - 默认情况下遇到RunTimeException以及其子类，会自动回滚
    - rollbackFor="异常类型"，表示遇到指定异常会回滚
    - noRollbackFor="异常类型"，表示遇到指定的异常不会回滚
- 只读优化
    - 当你确定你的操作中有且仅有查询时使用
    - readOnly=true
- 隔离级别

```
static int TRANSACTION_NONE
          指示事务不受支持的常量。
static int TRANSACTION_READ_UNCOMMITTED
          指示可以发生脏读 (dirty read)、不可重复读和虚读 (phantom read) 的常量。
static int TRANSACTION_READ_COMMITTED
          指示不可以发生脏读的常量；不可重复读和虚读可以发生。
static int TRANSACTION_REPEATABLE_READ
          指示不可以发生脏读和不可重复读的常量；虚读可以发生。
static int TRANSACTION_SERIALIZABLE
          指示不可以发生脏读、不可重复读和虚读的常量。
```

- 超时

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


    <!-- 配置扫包 -->
    <context:component-scan base-package="service.impl"></context:component-scan>
```

```
    <!-- 配置数据源 -->
    <context:property-placeholder location="classpath:dataSource.properties"></context:property-
placeholder>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <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mybatis -->
    <!--
        配置SqlSessionFactory
        SqlSessionFactoryBean
     -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 此处配置一切与mybatis相关的操作 -->
        <!-- 注入数据源 -->
        <property name="dataSource" ref="dataSource"></property>
        <!-- 配置别名 -->
        <!--<property name="typeAliasesPackage" value="entity"></property>-->
        <!-- 注册mapper文件 -->
        <property name="mapperLocations">
            <list>
                <!--<value>classpath:mapper/UserMapper.xml</value>-->
                <!-- 支持通配符的方式 -->
                <value>classpath:mapper/*.xml</value>
            </list>
        </property>
        <!-- 引入mybatis-config.xml文件 -->
        <!--<property name="configLocation" value="classpath:mybatis-config.xml"></property>-->
    </bean>

    <!-- 配置UserMapper -->
    <!-- 方式一:使用FactoryBean -->
    <!--<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">-->
        <!--<property name="sqlSessionFactory" ref="sqlSessionFactory"></property>-->
        <!--<property name="mapperInterface" value="dao.UserMapper"></property>-->
    <!--</bean>-->

    <!--
        方式二:执行别名操作
        通过spring提供的后处理bean对接口所在的包做扫包操作
        最终将该包下的所有的类名作为当前bean的id
        首字母小写
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="dao"></property>
    </bean>
```

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

    <!-- AOP注解配置 -->
    <tex:annotation-driven transaction-manager="transactionManager"></tex:annotation-driven>

</beans>
```

```
package service.impl;

import dao.UserMapper;
import entity.User;
import entity.UserExample;
import exception.UserNotExistsException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import service.UserService;

import java.util.List;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/20 15:03
 * Description:
 * version:1.0
 */
//在类上配置事务，表示当前事务对当前类中的所有方法生效
@Service
@Transactional(propagation = Propagation.REQUIRED,rollbackFor = Exception.class)
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    public void regist(User user) {
        userMapper.insertSelective(user);
    }

    //在方法上配置事务，表示只对当前方法生效
    //方法上的事务配置优先级高于类上的
    @Transactional(propagation = Propagation.SUPPORTS,readOnly = true)
    public User login(String username, String password) throws UserNotExistsException {
        UserExample example = new UserExample();
        example.or()
                .andUsernameEqualTo(username)
```

```
                .andPasswordEqualTo(password);
        List<User> users = userMapper.selectByExample(example);
        if(users.isEmpty()){
            throw new UserNotExistsException("用户名或密码错误");
        }
        return users.get(0);
    }


}
```

6-4 web支持

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <!-- 指定spring容器所在位置 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <!-- 此处支持通配符 -->
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>


    <!-- 配置ContextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

</web-app>
```

```
package servlet;

import entity.User;
import exception.UserNotExistException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;
import service.UserService;
```

```java
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Author:shixiaojun@itany.com
 * Date:2018/11/20 16:13
 * Description:
 * version:1.0
 */
@WebServlet("/login")
public class LoginServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
//        ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
//
//        UserService userService = (UserService) ac.getBean("userService");

        ApplicationContext ac =
WebApplicationContextUtils.getWebApplicationContext(getServletContext());
        UserService userService = (UserService) ac.getBean("userService");


        String username = request.getParameter("username");
        String password = request.getParameter("password");

        try {
            User user = userService.login(username,password);
            request.getSession().setAttribute("user",user);
            response.sendRedirect(request.getContextPath()+"/success.jsp");
        } catch (UserNotExistException e) {
            request.setAttribute("loginMsg",e.getMessage());
            request.getRequestDispatcher("login.jsp").forward(request,response);
        }


    }
}
```