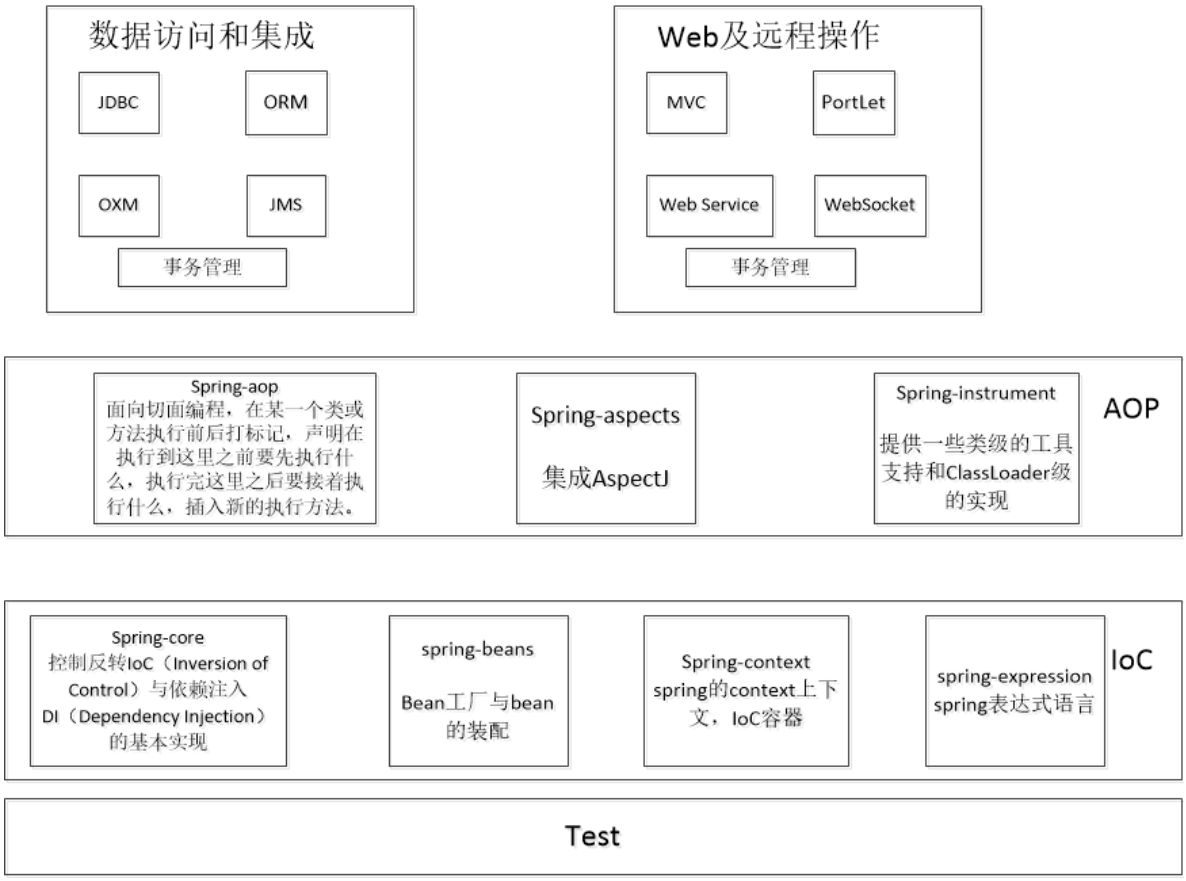
Spring

1 springt体系结构



2 简单案例

2.1 配置web.xml文件

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:smart-context.xml</param-value>
</context-param>
    <!-- ContextLoaderListener实现了ServletContextListener接口，只负责监听WEB容器启动和关闭-->
<listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

2.2 配置spring配置文件

```
<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-3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
       <property name="message" value="Hello World!"/>
   </bean>
</beans>
```

2.3 编写程序

```
public class HelloWorld {
   private String message;
   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

```
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
             new ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

2.4 执行程序

```
Your Message : Hello World!
```

3 IoC(控制反转)

3.1 依赖注入

三种类型：构造函数注入、属性注入和接口注入。

构造函数注入

```
public class MoAttack{
    private Geli geli;
    public MoAttack(Geli geli){
        this.geli=geli;
    }
}
```

属性注入

```
public class MoAttack{
    private Geli geli;
    public void setGeli(Geli geli){
        this.geli=geli;
    }
}
```

接口注入

```
public class MoAttack implements ActorArrangable{
    private Geli geli;
    //实现接口方法
    public void injectGeli(Geli geli){
        this.geli=geli;
    }
}
```

3.2 ApplicationContext配置

基于注解的配置信息提供类Bean

```
@Configuration
public class Beans{
    @Bean(name="car")
    public Car buildCar(){
    }
}
```

spring基于xml配置

```
<beans
        //默认命名空间、用于springBean的定义
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        //使用p命名空间，简化配置
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:context="http://www.springframework.org/schema/context"
        //aop的配置定义
        xmlns:aop="http://www.springframework.org/schema/aop"
        //声明式事物配置定义
        xmlns:tx="http://www.springframework.org/schema/tx"
```

基于注解的配置

@Component注解，细分为各层注解：

@Repository:用于对DAO实现类进行标注

@Service:用于对Service实现类进行标注

@Controller:用于对Controller实现类进行标注

扫描注解定义的Bean:

```
<context:component-scan base-package="com.study.service"/>
```

自动装配Bean

```
@Service
public class UserService {
    //使用@Qualifier指定注入Bean的名称    @Qualifier("userDao")
    private UserDao userDao;
    private LoginLogDao loginLogDao;
   @Autowired
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
```

4 spring AOP

4.1 关键词介绍

连接点（joinpoint）:在方法调用前后、抛出异常时及方法调用前后程序执行点织入增强

切点（pointcut）：正则表达语句

增强（Advice）：织入目标类连接点上的一段代码

目标对象（Target）:增强逻辑的织入目标类

切面（Aspect）:切点和增强组成。

Aspectj是语言级的AOP实现

4.2 使用Aspectj的例子

```
public class NaiveWaiter implements Waiter{
    public void greetTo(String clientName){
        System.out.println("NaiveWaiter:greet to" + clientName + "....");
    }
    public void serveTo(String clientName){
        System.out.println("NaiveWaiter:serving to" + clientName + "....");
    }
}
```

使用Aspectj注解定义一个切面

```
@Aspect   //通过该注解将PreGreetingAspect标识为一个切面
public class PreGreetingAspect{
    @Before("execution(* greeTo(..))")     //定义切点(括号中表达式)和增强类型（Before）
    public void beforeGreeting(){           //增强的横切逻辑
        System.out.println("how are you");
    }
}
```

通过AspectJProxyFactory为NaiveWaiter生成织入PreGreetingAspect切面的代理

```
public class AspectJProxyTest{
    @Test
    public void proxy(){
        Waiter target = new NaiveWaiter();
        AspectJProxyFactory factory = new AspectJProxyFactory();
        factory.setTarget(target);  //设置目标对象
```

```
        factory.addAspect(PreGreetingAspect.class); //添加切面类
        Waiter proxy = factory.getProxy();        //生成织入切面的代理对象
        proxy.greetTo("John");
        proxy.serveTo("John")
    }
}

how are you    //在greetTo方法前织入逻辑成功
greet to John...
serving John...
```

通过配置使用@AspectJ

```
<!--目标bean -->
    <bean id="waiter" class="com.smart.NaiveWaiter"/>
    <!-- 使用@AspectJ注解的切面类-->
    <bean class="com.smart.aspectj.example.PreGeetingAspect" />
    <!--自动代理创建器，自动将@AspectJ注解切面类织入目标Bean中 -->
    <!-- 使用aop命名空间后
    只需要：<aop:aspectj-autoproxy/>
        -->
    <bean
class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/>
```

4.3 @AspectJ语法基础

方法切点函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如@annotation(com.smart.anno.NeedTest)表示任何标注了@NeedTest 注解的目标类方法
方法入参切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter)表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如@args(com.smart.Monitorable)表示任何这样的一个目标方法：它有一个入参且入参对象的类标注@Monitorable 注解
目标类切点函数	within()	类名匹配串	表示特定域下的所有连接点。如 within(com.smart.service.*)表示 com.smart.service 包中的所有连接点，即包中所有类的所有方法；而 within(com.smart.service.*Service)表示在 com.smart.service 包中所有以 Service 结尾的类的所有连接点
	target()	类名	假如目标类按类型匹配于指定类，则目标类的所有连接点匹配这个切点。如通过 target(com.smart.Waiter)定义的切点、Waiter 及 Waiter 实现类 NaiveWaiter 中的所有连接点都匹配该切点
	@within()	类型注解类名	假如目标类按类型匹配于某个类 A，且类 A 标注了特定注解，则目标类的所有连接点匹配这个切点。如@within(com.smart.Monitorable)定义的切点，假如 Waiter 类标注了@Monitorable 注解，则 Waiter 及 Waiter 实现类 NaiveWaiter 的所有连接点都匹配这个切点
	@target()	类型注解类名	假如目标类标注了特定注解，则目标类的所有连接点都匹配该切点。如@target (com.smart.Monitorable)，假如 NaiveWaiter 标注了@Monitorable，则 NaiveWaiter 的所有连接点都匹配这个切点

通配符：

*：匹配任意字符，只能匹配上下文中的一个元素

..：匹配任意字符，可以匹配上下文中多个参数，在表示类时，必须和*联合使用，在表示入参时单独使用

+：表示按类型匹配指定类的所有类，必须跟在类名后面，如com.smart.Car+。继承或扩展指定类的所有类。

4.4 增强类型

1、@Before 前置增强，相当于BeforeAdvice

2、@AfterReturning 后置增强，相当于AfterReturningAdvice.

value:定义切点

pointcut:表示切点信息。

returning:将目标对象方法的返回值绑定给增强的方法。

3、@Around 环绕增强

4、@AfterThrowing 抛出增强

5、@After final增强，不管是抛出异常还是正常退出，该增强都会得到执行。

6、@DeclareParents 引介增强

切点函数详解

@annotation 标注某个注解的所有方法。

```
@Aspect
public class TestAspect{
    public void needTestFun(){
        @AfterReturning("@annotation(com.smart.anno.NeedTest)")  //后置增强切面
        System.out.println("needTestFun() executed!");
    }
}
```

```
public class NaughtyWaiter implements Waiter{
    @NeedTest    //标注注解
    public void greetTo(String clientName){
        System.out.println("NaughtyWaiter:greet to" + clientName + "...");
    }
}
```

```
public class NaiveWaiter implements Waiter{
    public void greetTo(String clientName){   //未标注
        System.out.println("NaiveWaiter:greet to" + clientName + "...");
    }
}
```

配置文件

```
<aop:aspectj-autoproxy>
<bean id="naiveWaiter" class="com.smart.NaiveWaiter"/>
<bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
<bean class="com.smart.aspectj.fun.TestAspect"/>
```

测试：

```
public class PointcutFunTest{
    @Test
    public void pointcut{
        String configPath = "com/smart/aspectj/fun/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configpath);
        Waiter naiveWaiter = (Waiter)ctx.getBean("naiveWaiter");
        Waiter naughtyWaiter = (Waiter)ctx.getBean("naughtyWaiter");
        naiveWaiter.greetTo("John");
        naughtyWaiter.greetTo("Tom");
    }
}
```

测试结果：

```
 naiveWaiter:greet to John...
 naughtyWaiter:greet to Tom...  //标注@NeedTest的方法执行后置增强
 needTestFun() executed!
```

execution

语法：execution(<修饰符模式>？<返回类型模式><方法名模式>(<参数模式>) <异常模式>？)

within

语法：within(<类匹配模式>)　within(com.smart.NaiveWaiter)

within(com.smart.*):匹配com.smart包中的所有类

within(com.smart..*):匹配com.smart包及子孙包中的类

target

语法：target(M)

target(com.smart.Waiter):NaiveWaiter、naughtyWaiter及CuteNaiveWaiter的所有方法都匹配切点

4.4.1 后置增强

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
       <aop:after-returning method="afterReturning"
           pointcut="target(com.smart.SmartSeller)" returning = "retVal"/>
    </aop:aspect>
</aop:config>              //returning属性必须和增强方法的入参名一致
```

4.4.2 环绕增强

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
       <aop:around method="aroundMethod"
           pointcut="execution(* serveTo(..) and within(com.smart.Waiter))"/>
    </aop:aspect>
</aop:config>
```

4.4.3 抛出异常增强

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
       <aop:after-throwing method="afterThrowingMethod"
           pointcut="target(com.smart.SmartSeller) and execution(*
checkBill(..))"
           throwing="iae"/>    //通过iae查找增强方法对应名字的入参
    </aop:aspect>
</aop:config>
```

4.4.4 Final增强

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
       <aop:after method="afterMethod"
           pointcut="execution(* com..*.Waiter.greetTo(..))"/>
    </aop:aspect>
</aop:config>
```

4.4.5 引介增强

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
        <aop:declare-parents
            implements-interface="com.smart.Seller"
            default-impl="com.smart.SmartSeller"
            types-matching="com.smart.Waiter+" />
    </aop:aspect>
 </aop:config>
```

5 整合ORM框架

5.1 使用jdbcTemplate

定义DataSource——定义JdbcTemplate——声明一个抽象的——配置具体的DAO

```
<!-- 定义一个使用DBCP实现的数据源-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:driverClassName="com.mysql.jdbc.Driver"
          p:url="jdbc:mysql://localhost:3306/sampledb"
          p:username="root"
          p:password="root" />

 <!-- 定义JDBC模板Bean-->
 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
          p:dataSource-ref="dataSource" />
```

声明一个具体的DAO

```
public class UserDao {
  private JdbcTemplate jdbcTemplate;
  private final static String QUERY_BY_USERNAME= "SELECT
user_id,user_name,credits "+
          "FROM t_user WHERE user_name=? ";
  @Autowired
  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }
  public User findUserByUserName(final String userName){
    final User user=new User();
   /**
   * 使用回调接口从结果集中获取数据，仅有的方法是：void processRow
   * @param userName
   * @return
   */
    jdbcTemplate.query(QUERY_BY_USERNAME, new Object[]{userName},
            new RowCallbackHandler() {
              public void processRow(ResultSet rs) throws SQLException {
                user.setUserId(rs.getInt("user_id"));
                user.setUserName(userName);
                user.setCredits(rs.getInt("credits"));
              }
            });
    return user;
  }
```

5.2 在Spring中使用Hibernate

移除了hibernate文件配置

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <bean id="sessionFactory"
         class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
```

```
        <!-- 指定Hibernate实体类映射文件 -->
        <property name="mappingLocations">
            <list>
                <value>classpth*:/com/smart/domain/hbm/Board.hbm.xml</value>
                <value>classpth*:/com/smart/domain/hbm/LoginLog.hbm.xml</value>
                <value>classpth*:/com/smart/domain/hbm/Post.hbm.xml</value>
                <value>classpth*:/com/smart/domain/hbm/User.hbm.xml</value>
                <value>classpth*:/com/smart/domain/hbm/Topic.hbm.xml</value>
            </list>
        </property>
        <!-- 指定Hibernate配置属性 -->
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect
                </prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>
    <!-- 注册hibernateTemplate -->
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate4.HibernateTemplate"
         p:sessionFactory-ref="sessionFactory" />
```

在配置映射文件属性时可以更方便的配置：

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        p:dataSource-ref="dataSource"
        p:mappingLocation="classpath:/com/smart/domain/hbm/**/*.hbm.xml"/>
```

使用注解配置：

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <!-- 指定Hibernate实体类映射文件 -->
        <property name="packagesToScan">
            <list>
                <value>com.smart.domain</value>
            </list>
        </property>
```

使用JPA注解的实体类

```
@Entity
@Table(name = "t_user")
public class User extends BaseDomain{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int userId;

    @Column(name = "user_name")
    private String userName;
```

```
    @Column(name = "user_type")
    private int userType = NORMAL_USER;

    @Column(name = "last_ip")
    private String lastIp;

    @Column(name = "last_visit")
    private Date lastVisit;
```

使用注解配置自动扫描

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <!-- 开启包扫描 -->
        <property name="packagesToScan" value="com.smart.domain" />
```

6 Spring的事务管理

6.1 配置事务管理器：

SpringJDBC和Mybatis都是基于数据源的Connection访问数据库，所以使用DataSourceTransactionManager事务管理器，配置如下：

```
//配置一个数据源
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />
//基于数据源的事务管理器
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
          p:dataSource-ref="dataSource" />    //引用数据源
```

配置Hibernate事务管理器

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />  //引用一个数据源
</bean>
<bean id="transactionManager"

 class="org.springframework.orm.hibernate4.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />    //注入会话工厂
</bean>
```

Hibernate使用org.hibernate.Session封装Connection,所以需要能创建session的SessionFactory.

6.2 基于aop/tx命名空间的配置

```
<aop:config proxy-target-class="true">
```

```
    <!-- 通过aop定义事务增强页面，切点表达式指com.smart.service包下所有
    用@Transactional标注的方法 -->
     <aop:pointcut id="serviceMethod"
                   expression="(execution(* com.smart.service..*(..))) and
  (@annotation(org.springframework.transaction.annotation.Transactional))" />
     <!-- 引用事务增强 -->
     <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
</aop:config>
<!-- 事务增强 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```