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:p=*"http://www.springframework.org/schema/p"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.1.xsd"*>

<context:component-scan base-package=*"com.habuma.spitter.domain"*/>

<aop:aspectj-autoproxy />

<context:annotation-config />//when component was exist it will be unnecessary

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"/*>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/spitter"/*>

<property name=*"username"* value=*"root"/*>

<property name=*"password"* value=*"mysqladmin"/*>

</bean>

<bean id=*"sessionFactory"*

class=*"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*/>

<property name=*"packagesToScan"* value=*"com.habuma.spitter.domain"*/>

<property name=*"hibernateProperties"*>

<props>

<prop key=*"dialect"*>org.hibernate.dialect.MySQLDialect</prop>

</props>

</property>

</bean>

<bean id=*"txManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"*/>

</bean>

<tx:annotation-driven transaction-manager=*"txManager"*/>

装配Bean

scope="prototype" (原型,范例)用于在装配bean时每次都产生新的实例singleton(default) request session

init-method="method" destroy-method="method" 定义初始化时调用和销毁时调用方法

也可在http后定义全局default-init-method="method" default-destroy-method="method"

<constructor-arg value=""><constructor-arg ref="">构造器注入

<propertry name="" value="">注入属性,一定要有setter方法

<property name="">

<bean class=""> //内部bean的定义，构造方法同样支持，但是结构会非常乱，不可复用

</property>

p:paramname="" p:paramname-ref="" //简单注入参数法

<property name="">

<list>

<ref bean=""> //用value的话就是表示一般类型

<ref bean="">

<ref bean="">

</list>

</property>

<property name="">

<map>

<ref key="" value-ref=""> //用value的话就是表示一般类型

<ref key="" value-ref="">

<ref key="" value-ref="">

</map>

</property>

如果是properties 则用<props>跟map的区别是key-value都只能用String

<props>

<prop key="">string</prop>

<prop key="">string</prop>

<prop key="">string</prop>

</props>

<property name=""><null/></property>转配空值的方法

Spring Expression Language(SpEL)表达式

<property name=”” value=”#{5}”>

<property name=”” value=”this value is #{5}”>

<property name=”” value=”#{89.7}”>

<property name=”” value=”#{1e4}”>

<property name=”” value=”#{‘string’}”>

<property name=”” value=”#{false}”>

<property name=”” ref=”#{bean}”>

<property name=”” value=”#{bean.param}”> 反射为getParam()

<property name=”” value=”#{bean.method()}”> 还可以累加toUpperCase()将结果大写

<property name=”” value=”#{bean.method()?.toUpperCase()}”>防止方法返回空的方法调用toUpperCase()会出错的问题

<property name=”” value=”#{T(java.lang.Math).random()}”>

<property name=”” value=”#{T(java.lang.Math).PI}”>

<property name=”” value=”#{各种数学逻辑运算符}”>

==(ea) <(lt) >(gt) <=(le) >=(ge) &&(and) ||(or) !(not)

<property name=”” value=”#{shape.kind==’cricle’ and shape.perimeter gt 10000}”>

<property name=”” value=”#{true?select1:select2}”>

<property name=”” value=”#{param==null?select1:select2}”>可以省略为#{param?:select?}

<property name=”” valaue=”#{str1 matches ‘str2’}”>正则表达式

<util:list id=”cities”>

<bean />

<bean/>

</util:list>

<property name=”” value=”#{cities[1]}”>支持集合

T(java.lang.Math).random() \* cities.size() //随机选择集合元素

还有查询，投影功能

<property name=”” value=”#{cities.?[population gt 1000]}”>选出人口大于1000的城市

cities.^[]查询第一个 cities.$[]查询最后一个

<property name=”” value=”#{cities.![name]}”>将name元素集合投影

Bean的自动装配与检测

autowiring(自动装配):用于消除property跟constructor-arg （可以在头文件中输入default-autowire=”autodetect”）映射所有bean

byName用于通过属性名字装配bean

byType用于装配与bean中属性所需bean相同类型的bean,多个符合抛出异常

constructor用于构造器中装配bean

autodetect先尝试用constructor，然后byType

bean中标记primary=”false”可以不被首选(测试无效)

bean中标记autowire-candidate=”false”在自动装配中被忽略

<constructor>是不能又自动又手动的，byType可以

注解

<context:annotation-config /> //用注解时要在xml里配置

@Autowired注解(setter默认byType)测试用于constructor是可行的，属性也是可以的，前提是用Component

@Autowired(required=false)当找不到时设置为空，只有一个构造器能设为true,多个构造器时选择入参最多的一个

@Qualifier(合格者)(“guitar”)用于指定属性bean,不能用于构造器，也可以在bean中为起命新的专用名<qualifier value=””>

@Inject(JSR-330)注解

常用方法：注解属性，方法，构造器

@Inject

public KnifeJuggler(Provider<Knife> KnifeProvider){

knives = new HashSet<Knife>();

for(int i=0;i<5;i++){

knives.add(KnifeProvider.get());

}

}

@Value(“”)注解,支持直接传入基本数据类型，和支持表达式语言

获得5个knife，但是Knife 的bean必须标识为prototype才会获得不同的bean

@Named(“guitar”) 跟@Quailfier一个道理

@Resource(JSR-250)注解

autodiscovery(自动检测):用于直接消除bean

xmlns:context=”<http://www.springframework.org/schema/context>”

http://www.springframework.org/schema/context

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

<context:component-scan base-package=*"com.springinaction.springidol"* />//需要配置的文件

@Component:标识为Spring组件

跟@Autowired混合使用可以达到超级好的效果！！！

@Controller:标识为Spring MVC controller

@Repository:标识为数据仓库

@Service：标识为服务

如果不能看源码的时候，可以这样配置(用过滤器)：(当然也有exclude)

<context:component-scan base-package=”com.springinaction.springidol”>

<context:include-filter type=”assignable” expression=”com.springinaction.springidol.Instrument”/>

</context:component-scan>

5种过滤器：

annotation扫描expression指定注解的类

assignable扫描expression指定类型的类

aspectj扫描sxpression指定AspectJ表达式匹配的类

custom自定义，由expression指定

regex扫描类的名称与expression所指定正则表达式所匹配的类

<context:component-scan base-package=”com.springinaction.springidol”>

<context:exclude-filter type=”annotation” expression=”com.springinaction.springidol.SkipIt”/>

</context:component-scan>

就是说,用了@SkipIt注解的类不自动装配

AOP

Advice(通知):切面的工作被称为通知

⊙Before在方法调用之前通知

⊙After在方法调用之后，无论方法执行是否成功

⊙After-returning 方法成功执行之后

⊙After-throwing 在方法抛出异常之后

⊙Around通知包裹了被通知的方法，自动执行自定义行为

Joinpoint(连接点):通知的时机

Pointcut(切点):定义何处执行

Aspect(切面):切面是连接点和切点的结合

Introduction(引入):允许我们向现有的类添加新方法或属性

Weaving(织入):织入是将切面应用到目标对象来创建新的代理对象的过程(在运行期织入)

AOP基本理念：创建切点来定义切面织入的连接点是AOP框架的基本功能

Spring AOP是基于代理的，所以只支持对方法的拦截，不能对属性和构造器拦截

AspectJ是支持基于注解的切面，能获得更强大和细粒度的控制

Spring AOP所支持的AspectJ的切点表达式

arg() 在execution中用and args()可以把参数传入指定param中，然后before中用arg-name=”param”得到此参数，用于方法中

@args()

execution:用于匹配是连接点的执行方法

this()

target()

@target()

within():可以限定哪个包中的类execution()&&within(package.\*)，可以把within替换成and bean()来限制是哪个bean

@within()

@annotation

Spring AOP标签

<aop:advisor>

<aop:after> <aop:after-retuning> <aop:after-throwing>

<aop:around>关键点：ProceedingJoinPoint为入参,joinpoint.proceed();为执行入参方法

<aop:aspect> <aop:aspect-autoproxy>

<aop:before>

<aop:config>

<aop:declare-parents>引入方法 //不明白是怎么回事，或有什么具体作用

<aop:aspect>

<aop:declare-parents types-mathing=” ..Perform+” implement-interface=”..Contestant” default-impl=”bean” or delegate-ref=”bean”>

</aop:aspect>

<aop:pointcut>

Spring AOP注解

xmlns:aop=”<http://www.springframework.org/schema/aop>”

http://www.springframework.org/schema/aop

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

不能注解构造器跟参数，只能注解方法

<aop:aspectj-autoproxy /> <!—自动装配成AOP -->

@Aspect

public class {

@Pointcut(“execution()”)

public void fun();

}

@Before(“fun()”) @After(“fun()”) @AfterThrowing(“fun()”) @AfterReturning(“fun()”) @Around(“fun()”) @DelcareParents

传入参数的设置

@Pointcut("execution(\* com.springinaction.aspectj.Thinker.thinkOfSomething(String))&&args(thoughts)")

**public** **void** thinking(String thoughts){}

@Before("thinking(thoughts)")

DataBase

DataAccessException不需要捕获的异常

需要添加spring-jdbc library和mysql-driver,但在xml是不需要添加任何http的

使用基于JDBC驱动的数据源

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"org.gjt.mm.mysql.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/spitter"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"mysqladmin"*/>

</bean>

使用数据源连接池

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"/*>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/spitter"/*>

<property name=*"username"* value=*"root"/*>

<property name=*"password"* value=*"mysqladmin"/*>

<property name=*"initialSize"* value=*"5"/*>

<property name=*"maxActive"* value=*"30"/*>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<constructor-arg ref=*"dataSource"*/>

</bean>

然后在实际操作类中添加一个JdbcTemplate即可，方法调用jdbcTemplate.method()

Spring集成Hibernate

从SessionFactory(管理session打开,关闭)获取session接口,session接口是实现一切的关键

延迟加载：只抓取需要的数据

预先抓取：获得完整的关联对象

级联：当一个对象改变时，所有关联对象都会被影响

Hibernate功能:缓存,延迟加载,预先抓取,分布式缓存

Hibernate自身已经能保证每个事务使用同一session

//mappingResources装配Hibernate映射文件，定义应用程序持久化策略

//hibernateProperties配置Hibernate操作细节

@Transient 表示非一个字段的映射Hibernate应该讲其忽略

事务

原子性：由一个或多个活动所组成 的一个工作单元

一致性：系统确保所建模的业务处于一致的状态

隔离性：事务允许多个用户对相同的数据进行操作，互不影响

持久性：一旦事务完成，事务的结果应该持久化，存储到数据库当中

Spring并不直接管理事务，而是提供多种事务管理器:

jdbc.datasource.DataSourceTransactionManger

orm.hibernate3.HibernateTransactionManager

orm.jpa.JpaTransactionManager

事务成功时会调用Transaction对象的的commit()方法，类似的，事务失败则调用Transaction对象的rollback()方法

Spring声明事务的3种方法：

1. Spring AOP
2. TransactionProxyFactoryBean的代理bean来实现
3. 注解@Transaction

**事务属性**:

**传播行为**：定义客户端与被调用方法之间的事务边界（规则）

PROPAGATION\_MANDATORY(强制传播)：强制方法在事务中运行，事务不存在则抛出异常

PROPAGATION\_NEVER(不要传播)：表示这个方法不能在事务容器中总运行，有事务存在则抛出异常

PROPAGATION\_NESTED(嵌套传播)：嵌套进当前事务运行，但不受当前事务影响，如果当前没事务跟REQUIRED一样

PROPAGATION\_NOT\_SUPPORT(不支持传播)：表示该方法不应该在事务容器中运行，如果要运行则当前事务被挂起

PRAOPAGATION\_SUPPORTS(随便)：不需要事务容器支持，但是如果有事务容器则会到容器中运行

PROPAGATION\_REQUIRED(要求传播)：表示必须在事务容器中运行，没事务容器就创建一个

PROPAGATION\_REQUIRED\_NEW(新的传播)：表示一定要在自己创建的新事务容器中运行，如果当前有事务则被挂起

其中要考虑边界问题，REQUIRED\_NEW完全用自己的边界，REQUIRED则要看当前是否存在容器

**隔离级别**：定义一个事务可能受其他并发事务影响的程度，也就是事务对数据的自私程序

赃读：一个事务读了一另个事务还未提交的数据，读完后另一个事务回滚了，简单说就是被坑了，没想到它这么靠不住

不可重复读：一个事务执行多次查询操作都获得不同的数据，通常是因为有个熊事务一直在改数据

幻读：一个事务读取了几行记录，但是随后另一个事务又插入几行数据，等第一个事务回去查询时发现这么多了几条数据，出现幻觉了？

ISOLATION\_DEFAULT(默认隔离)：使用后端数据库默认隔离级别

ISOLATION\_READ\_UNCOMMITTED：允许读未提交的数据，会出现脏读，幻读，或不可重复读

ISOLATION\_READ\_COMMITED：允许读取已经提交的事务，可以阻止赃读，但不可重复读和幻读还是存在

ISOLATION\_REPEATABLE\_READ：允许重复读，对同一字段读取一定一致，但不可以避免幻读

ISOLATION\_SERIALIAZABLE(可序列化的)：完成服从ACID(原子,一致, 隔离,持久)，不存在赃读，不可重复读和幻读，效率最低

**只读，超时**设置只有在那些具备启动一个新事务的传播行为(也不一定能产生,除非是propagation\_required\_new)required,required\_new,nested三者设置那两个从有意义

**回滚：**遇到运行时异常才会回滚，检查性异常不会回滚，跟EJB一致

注解参数设置

isolation :指定隔离级别 propagation定义传播规则 read-only：是否只读 timeout:定义超时时间

rollback-for:指定那些检查型异常应当回滚不提交 no-rollbace-for:指定哪些异常不回滚

**注解事务**

**@Transactional(propagation=Propagation.SUPPORTS,readOnly=”true”)**

Spring MVC

Spring MVC的核心是DispatcherServlet(前端控制器)

@Controller注解表明这个类是一个控制器类，跟@Component差不多

@Inject(javax.inject.Inject) 暂时不知道跟@Autowired有什么区别

@RequestMapping 表明方法是一个请求处理方法，要处理 /和/home中的路径请求

视图解析器

InternalResolverViewResolver:在Web应用程序的WAR文件中查找视图模块。视图模块的路径根据加完前缀和后缀的逻辑视图名称来确定

Apache Tiles是一个模板框架，它将页面分成片段并在运行时组装成完整的页面。