spring杂谈[原创]

作者: jinnianshilongnian
http://jinnianshilongnian.iteye.com

spring杂谈[原创]

目 录

1. spring杂谈[原创]

1.1 Spring事务处理时自我调用的解决方案及一些实现方式的风险 4

1.2 我对AOP的理解 22

1.3 Spring开闭原则的表现-BeanPostProcessor的扩展点-1 33

1.4 我对IoC/DI的理解 45

1.5 SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结 52

1.6 »Spring 之AOP AspectJ切入点语法详解（最全了，不需要再去其他地找了） 63

1.7 Spring开闭原则的表现-BeanPostProcessor扩展点-2 84

1.8 Spring3.1 对Bean Validation规范的新支持(方法级别验证) 100

1.9 Spring对事务管理的支持的发展历程（基础篇） 109

1.10 基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。 116

1.11 在spring中获取代理对象代理的目标对象工具类 130

1.12 如何为spring代理类设置属性值 133

1.13 我对SpringDAO层支持的总结 135

1.14 我对SpringDAO层支持的总结 141

1.15 我对SpringDAO层支持的总结 147

1.16 我对Spring 容器管理事务支持的总结 153

1.17 我对Spring 容器管理事务支持的总结 160

1.18 不重复配置——利用Spring通用化配置 167

1.19 @Value注入Properties 错误的使用案例 178
1.20 @Value注入Properties 使用错误的案例 182
1.21 扩展SpringMVC以支持更精准的数据绑定 186
1.22 扩展SpringMVC以支持更精准的数据绑定1 191
1.23 扩展SpringMVC以支持绑定JSON格式的请求参数 197
1.24 扩展SpringMVC以支持绑定JSON格式的请求参数 204
1.25 在应用层通过spring特性解决数据库读写分离 210
1.26 context:component-scan扫描使用上的容易忽略的use-default-filters 218
1.27 idea内嵌jetty运行springmvc项目报ConversionFailedException 222
1.28 springmvc 3.2 @MatrixVariable注解 224
1.29 spring3.2 带matrix变量的URL匹配问题 226
1.30 Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired失效 228
1.31 Spring事务不起作用 问题汇总 232
1.32 Spring3 Web MVC下的数据类型转换（第一篇）——《跟我学Spring3 Web MVC》抢先看 233
1.33 Spring 注入集合类型 249

1.1 Spring事务处理时自我调用的解决方案及一些实现方式的风险

发表时间: 2012-04-16 关键字: spring

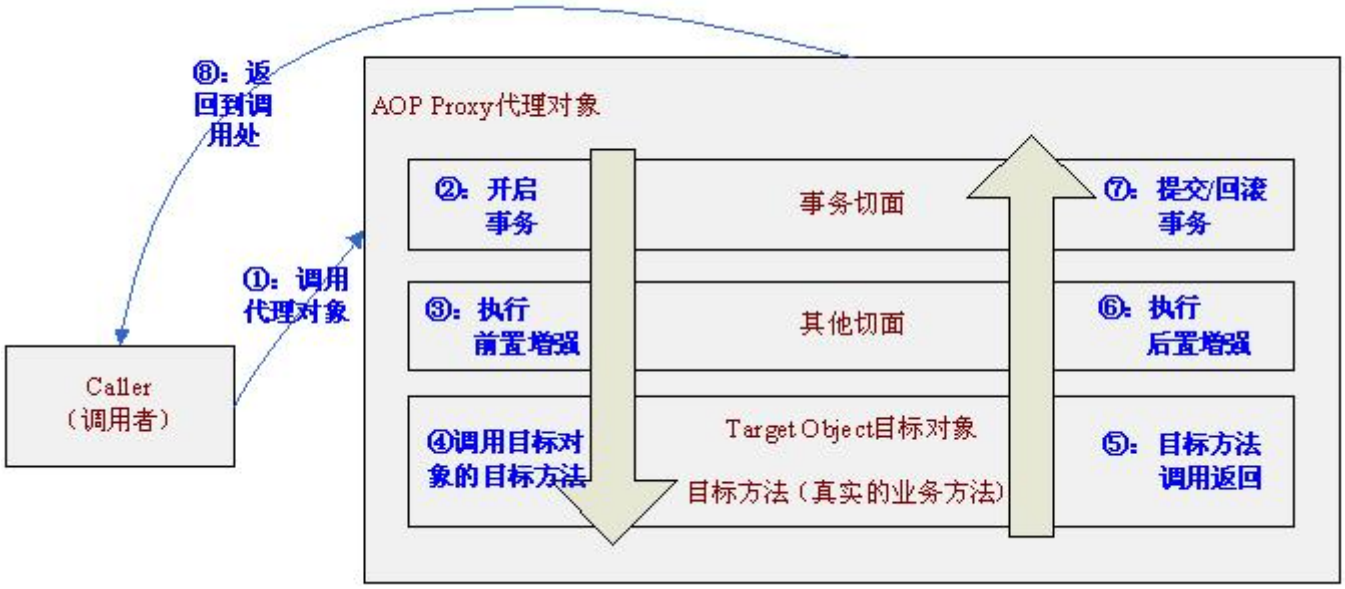
前些日子一朋友在需要在目标对象中进行自我调用，且需要实施相应的事务定义，且网上的一种通过BeanPostProcessor的解决方案是存在问题的。因此专门写此篇帖子分析why。

1、预备知识

aop概念请参考【http://www.iteye.com/topic/1122401】和【http://jinnianshilongnian.iteye.com/blog/1418596】

spring的事务管理，请参考【http://jinnianshilongnian.iteye.com/blog/1441271】

使用AOP 代理后的方法调用执行流程，如图所示



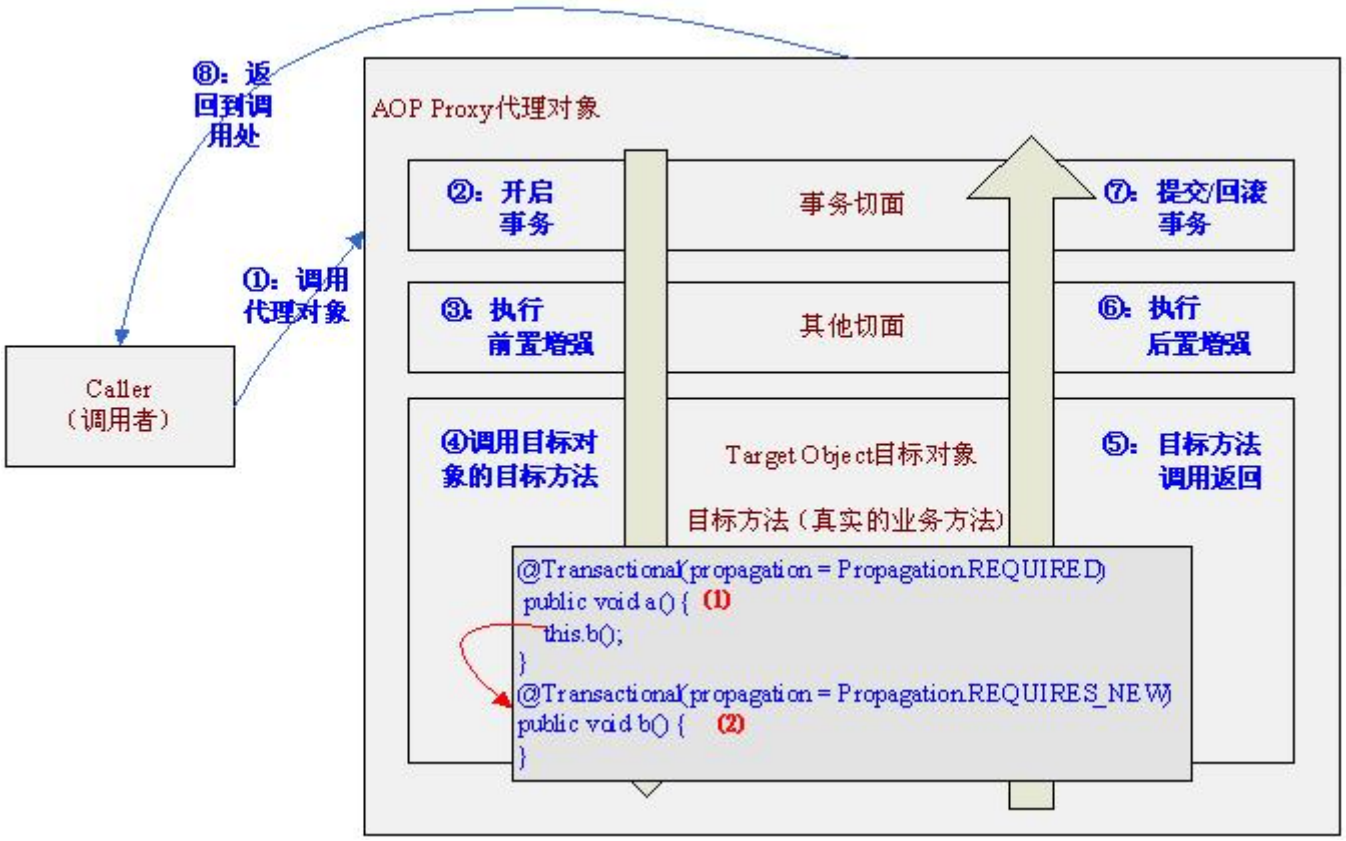
也就是说我们首先调用的是AOP代理对象而不是目标对象，首先执行事务切面，事务切面内部通过TransactionInterceptor环绕增强进行事务的增强，即进入目标方法之前开启事务，退出目标方法时提交/回滚事务。

2、测试代码准备

```
public interface AService {
    public void a();
    public void b();
}

@Service()
public class AServiceImpl1 implements AService{
    @Transactional(propagation = Propagation.REQUIRED)
    public void a() {
        this.b();
    }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void b() {
    }
}
```

3、问题

目标对象内部的自我调用将无法实施切面中的增强，如图所示



此处的this指向目标对象，因此调用this.b()将不会执行b事务切面，即不会执行事务增强，因此b方法的事务定义“@Transactional(propagation = Propagation.REQUIRES_NEW)”将不会实施，即结果是b和a方法的事务定义是一样的，可以从以下日志看出：

org.springframework.transaction.annotation.AnnotationTransactionAttributeSource Adding transactional method 'a' with attribute: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''

org.springframework.beans.factory.support.DefaultListableBeanFactory Returning cached instance of singleton bean 'txManager'

org.springframework.orm.hibernate4.HibernateTransactionManager Creating new transaction with name [com.sishuok.service.impl.AServiceImpl1.a]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; '' -----创建a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session …… for Hibernate transaction ---打开Session

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl1.a]

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl1.a] ----完成a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session .....---提交a方法事务

或

org.springframework.orm.hibernate4.HibernateTransactionManager Rolling back Hibernate transaction on Session .....---如果有异常将回滚a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

.....

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session ..... after transaction --关闭Session

我们可以看到事务切面只对a方法进行了事务增强，没有对b方法进行增强。

3、解决方案

此处a方法中调用b方法时，只要通过AOP代理调用b方法即可走事务切面，即可以进行事务增强，如下所示：

```
public void a() {
aopProxy.b();//即调用AOP代理对象的b方法即可执行事务切面进行事务增强
}
```

判断一个Bean是否是AOP代理对象可以使用如下三种方法：

AopUtils.isAopProxy(bean) ： 是否是代理对象；

AopUtils.isCglibProxy(bean) ： 是否是CGLIB方式的代理对象；

AopUtils.isJdkDynamicProxy(bean) ： 是否是JDK动态代理方式的代理对象；

3.1、通过ThreadLocal暴露Aop代理对象

1、开启暴露Aop代理到ThreadLocal支持（如下配置方式从spring3开始支持）

```
<aop:aspectj-autoproxy expose-proxy="true"/><!—注解风格支持-->
```

```
<aop:config expose-proxy="true"><!—xml风格支持-->
```

2、修改我们的业务实现类

this.b();----------修改为--------->((AService) AopContext.currentProxy()).b();

3、执行测试用例，日志如下

org.springframework.beans.factory.support.DefaultListableBeanFactory Returning cached instance of singleton bean 'txManager'

org.springframework.orm.hibernate4.HibernateTransactionManager Creating new transaction with name [com.sishuok.service.impl.AServiceImpl2.a]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; '' -----创建a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session ……for Hibernate transaction --打开a Session

org.springframework.orm.hibernate4.HibernateTransactionManager Preparing JDBC Connection of Hibernate Session ……

org.springframework.orm.hibernate4.HibernateTransactionManager Exposing Hibernate transaction as JDBC transaction ……

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl2.a]

org.springframework.transaction.annotation.AnnotationTransactionAttributeSource Adding transactional method 'b' with attribute: PROPAGATION_REQUIRES_NEW,ISOLATION_DEFAULT; ''

……

org.springframework.orm.hibernate4.HibernateTransactionManager Suspending current transaction, creating new transaction with name [com.sishuok.service.impl.AServiceImpl2.b] -----创建b方法事务（并暂停a方法事务）

……

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session for Hibernate transaction ---打开b Session

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl2.b]

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl2.b] ----完成b方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session ..... ---提交b方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

.....

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session ..... after transaction --关闭 b Session

-----到此b方法事务完毕

org.springframework.orm.hibernate4.HibernateTransactionManager Resuming suspended transaction after completion of inner transaction ---恢复a方法事务

.....

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl2.a] ----完成a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session .....---提交a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

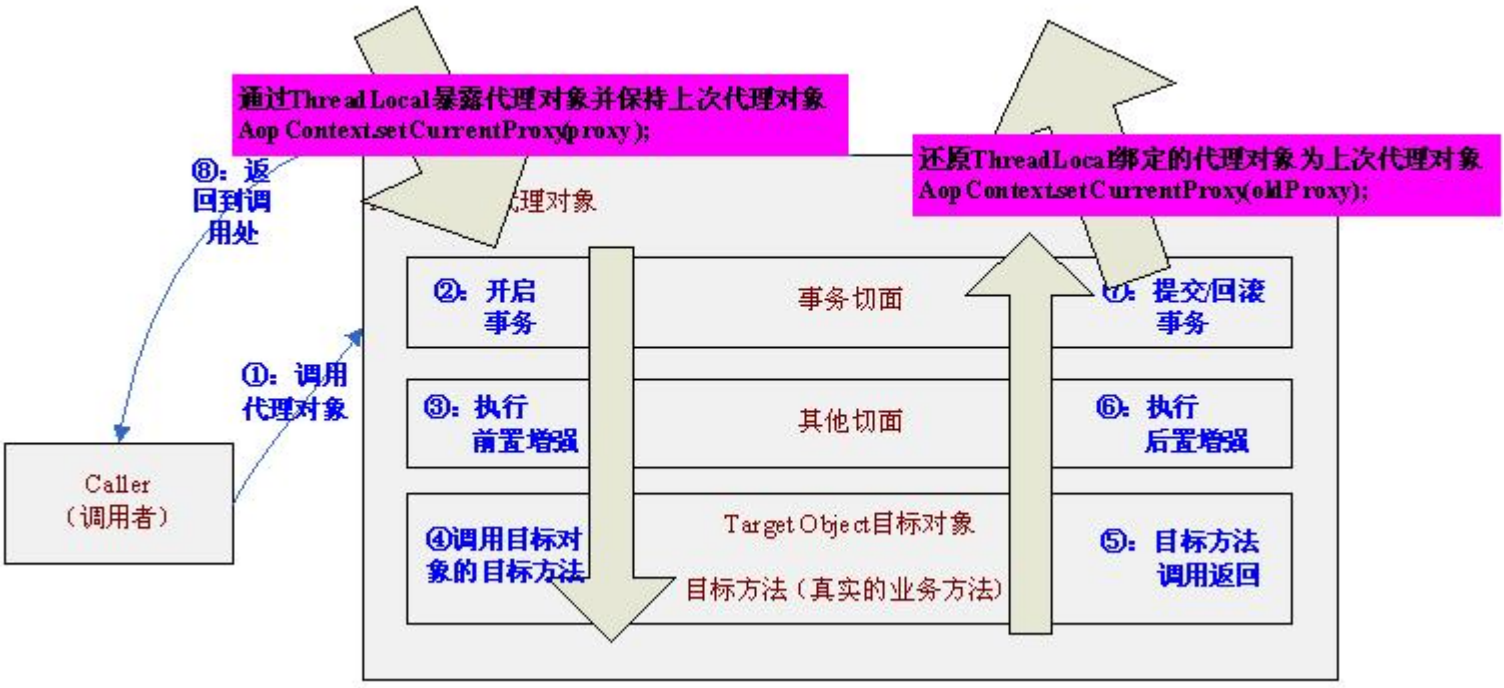
.....

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session ..... after transaction --关闭 a Session

此处我们可以看到b方法的事务起作用了。

以上方式是解决目标对象内部方法自我调用并实施事务的最简单的解决方案。

4、实现原理分析



4.1、在进入代理对象之后通过AopContext.serCurrentProxy(proxy)暴露当前代理对象到ThreadLocal，并保存上次ThreadLocal绑定的代理对象为oldProxy；

4.2、接下来我们可以通过 AopContext.currentProxy() 获取当前代理对象；

4.3、在退出代理对象之前要重新将ThreadLocal绑定的代理对象设置为上一次的代理对象，即AopContext.serCurrentProxy(oldProxy)。

有些人不喜欢这种方式，说通过ThreadLocal暴露有性能问题，其实这个不需要考虑，因为事务相关的（Session和Connection）内部也是通过SessionHolder和ConnectionHolder暴露到ThreadLocal实现的。

不过自我调用这种场景确实只有很少情况遇到，因此不用这种方式我们也可以通过如下方式实现。

3.2、通过初始化方法在目标对象中注入代理对象

```
@Service
public class AServiceImpl3 implements AService{
    @Autowired  //①  注入上下文
    private ApplicationContext context;

    private AService proxySelf; //②  表示代理对象，不是目标对象
```

```
    @PostConstruct      //③ 初始化方法
    private void setSelf() {
        //从上下文获取代理对象（如果通过proxtSelf=this是不对的，this是目标对象）
        //此种方法不适合于prototype Bean，因为每次getBean返回一个新的Bean
        proxySelf = context.getBean(AService.class);
    }
    @Transactional(propagation = Propagation.REQUIRED)
    public void a() {
       proxySelf.b(); //④ 调用代理对象的方法 这样可以执行事务切面
    }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void b() {
    }
}
```

此处日志就不分析，和3.1类似。此种方式不是很灵活，所有需要自我调用的实现类必须重复实现代码。

3.3、通过BeanPostProcessor 在目标对象中注入代理对象

此种解决方案可以参考http://fyting.iteye.com/blog/109236。

BeanPostProcessor 的介绍和使用敬请等待我的下一篇分析帖。

一、定义BeanPostProcessor 需要使用的标识接口

```
public interface BeanSelfAware {
    void setSelf(Object proxyBean);
}
```

即我们自定义的BeanPostProcessor （InjectBeanSelfProcessor）如果发现我们的Bean是实现了该标识接口就调用setSelf注入代理对象。

二、Bean实现

```
@Service
public class AServiceImpl4 implements AService, BeanSelfAware {//此处省略接口定义
    private AService proxySelf;
    public void setSelf(Object proxyBean) { //通过InjectBeanSelfProcessor注入自己（目标对象）的AO
        this.proxySelf = (AService) proxyBean;
    }
    @Transactional(propagation = Propagation.REQUIRED)
    public void a() {
        proxySelf.b();//调用代理对象的方法 这样可以执行事务切面
    }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void b() {
    }
}
```

实现BeanSelfAware标识接口的setSelf将代理对象注入，并且通过“proxySelf.b()”这样可以实施b方法的事务定义。

三、InjectBeanSelfProcessor实现

```
@Component
public class InjectBeanSelfProcessor implements BeanPostProcessor {
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansExc
        return bean;
    }
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansExce
        if(bean instanceof BeanSelfAware) {//如果Bean实现了BeanSelfAware标识接口，就将代理对象注入
            ((BeanSelfAware) bean).setSelf(bean); //即使是prototype Bean也可以使用此种方式
        }
        return bean;
    }
}
```

postProcessAfterInitialization根据目标对象是否实现BeanSelfAware标识接口，通过setSelf(bean)将代理对象（bean）注入到目标对象中，从而可以完成目标对象内部的自我调用。

关于BeanPostProcessor的执行流程等请一定参考我的这篇帖子，否则无法继续往下执行。

四、InjectBeanSelfProcessor的问题

（1、场景：通过InjectBeanSelfProcessor进行注入代理对象且循环依赖场景下会产生前者无法通过setSelf设置代理对象的问题。 循环依赖是应该避免的，但是实际工作中不可避免会有人使用这种注入，毕竟没有强制性。

（2、用例

（2.1、定义BeanPostProcessor 需要使用的标识接口

和3.1中一样此处不再重复。

（2.2、Bean实现

```
@Service
public class AServiceImpl implements AService, BeanSelfAware {//此处省略Aservice接口定义
    @Autowired
    private BService bService;   //①  通过@Autowired方式注入BService
    private AService self;       //②  注入自己的AOP代理对象
    public void setSelf(Object proxyBean) {
        this.self = (AService) proxyBean;  //③ 通过InjectBeanSelfProcessor注入自己（目标对象）的
        System.out.println("AService=="+ AopUtils.isAopProxy(this.self)); //如果输出true标识AOP代
    }
    @Transactional(propagation = Propagation.REQUIRED)
    public void a() {
        self.b();
    }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
```

```
    public void b() {
    }
}
```

```
@Service
public class BServiceImpl implements BService, BeanSelfAware {//此处省略Aservice接口定义
    @Autowired
    private AService aService;  //①  通过@Autowired方式注入AService
    private BService self;      //②  注入自己的AOP代理对象
    public void setSelf(Object proxyBean) {  //③ 通过InjectBeanSelfProcessor注入自己（目标对象）
        this.self = (BService) proxyBean;
        System.out.println("BService=" + AopUtils.isAopProxy(this.self)); //如果输出true标识AOP代
    }
    @Transactional(propagation = Propagation.REQUIRED)
    public void a() {
        self.b();
    }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void b() {
    }
}
```

此处A依赖B，B依赖A，即构成循环依赖，此处不探讨循环依赖的设计问题（实际工作应该避免循环依赖），只探讨为什么循环依赖会出现注入代理对象失败的问题。

循环依赖请参考我的博文【http://jinnianshilongnian.iteye.com/blog/1415278】。

依赖的初始化和销毁顺序请参考我的博文【http://jinnianshilongnian.iteye.com/blog/1415461】。

（2.3、InjectBeanSelfProcessor实现

和之前3.3中一样 此处不再重复。

（2.4、测试用例

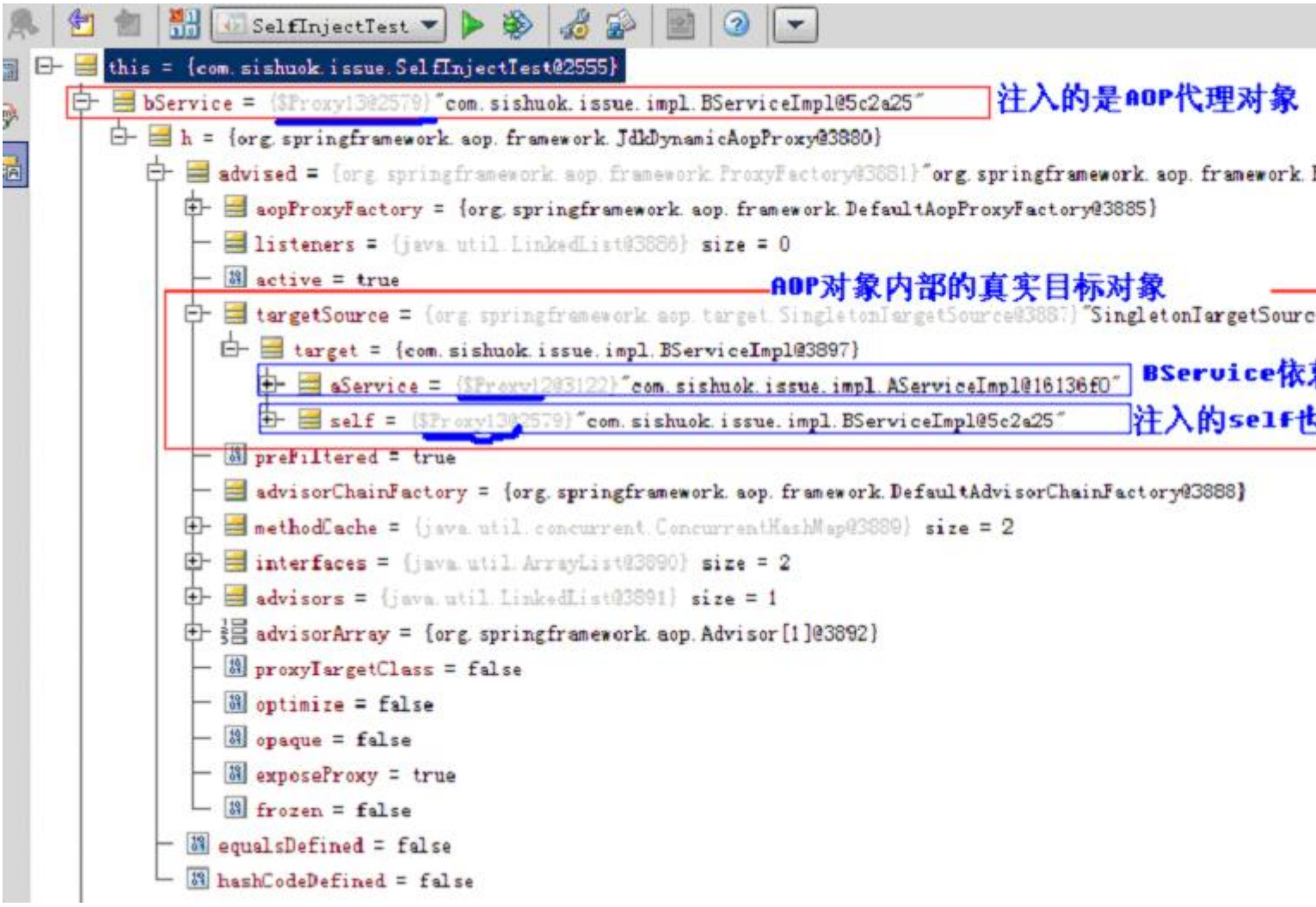
```
@RunWith(value = SpringJUnit4ClassRunner.class)
@ContextConfiguration(value = {"classpath:spring-config.xml"})
public class SelfInjectTest {
    @Autowired
    AService aService;
    @Autowired
    BService bService;
    @Test
    public void test() {
    }
}
```

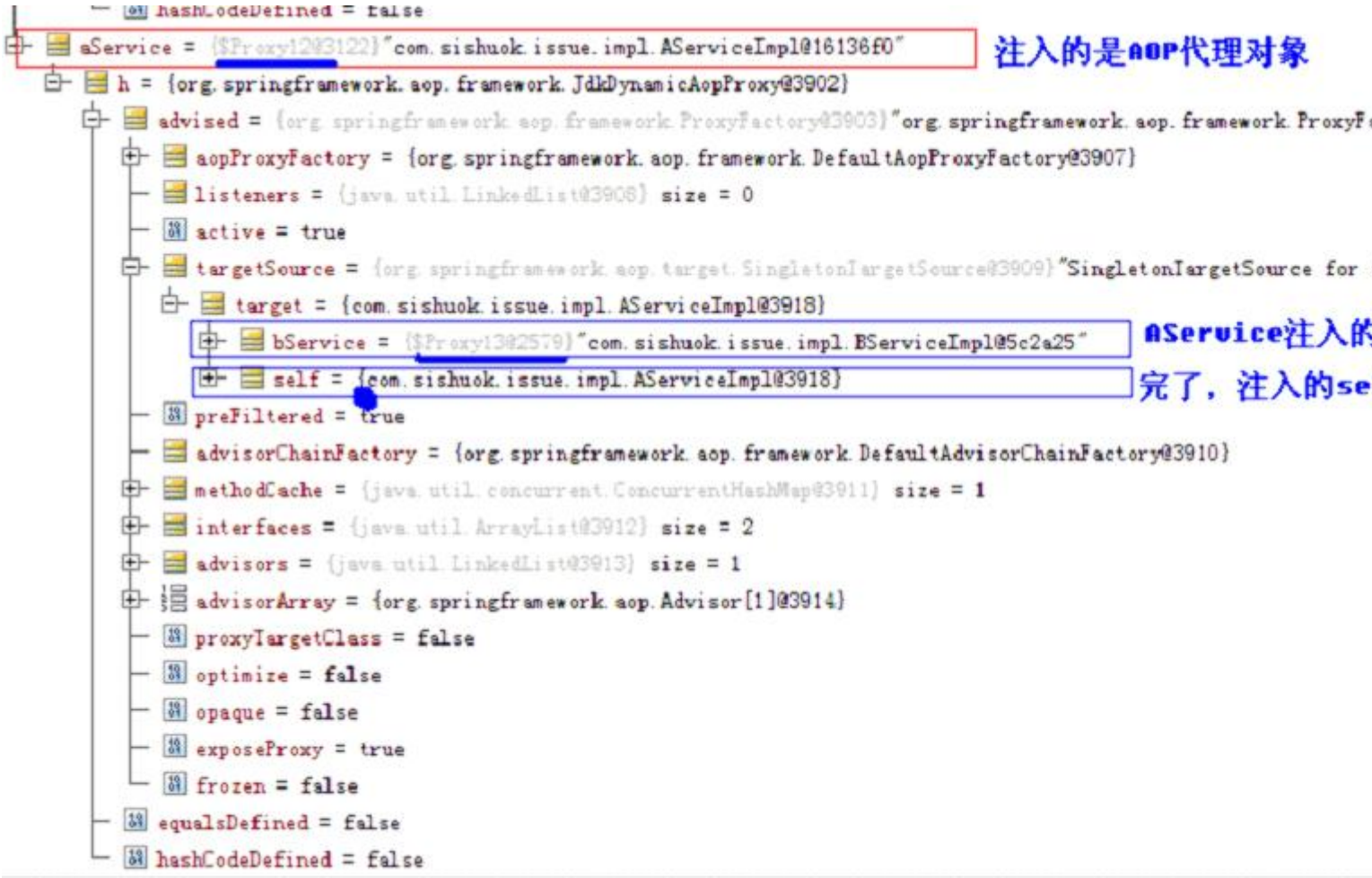
执行如上测试用例会输出：

BService=true

AService==false

即BService通过InjectBeanSelfProcessor注入代理对象成功，而AService却失败了（实际是注入了目标对象），如下是debug得到的信息：


SelfInjectTest
this = {com.sishuok.issue.SelfInjectTest@2555}
bService = {$Proxy13@2579} "com.sishuok.issue.impl.BServiceImpl@5c2a25"
注入的是AOP代理对象
h = {org.springframework.aop.framework.JdkDynamicAopProxy@3880}
advised = {org.springframework.aop.framework.ProxyFactory@3881} "org.springframework.aop.framework.
aopProxyFactory = {org.springframework.aop.framework.DefaultAopProxyFactory@3885}
listeners = {java.util.LinkedList@3886} size = 0
active = true
AOP对象内部的真实目标对象
targetSource = {org.springframework.aop.target.SingletonTargetSource@3887} "SingletonTargetSourc
target = {com.sishuok.issue.impl.BServiceImpl@3897}
aService = {$Proxy12@3122} "com.sishuok.issue.impl.AServiceImpl@16136f0"
BService依
self = {$Proxy13@2579} "com.sishuok.issue.impl.BServiceImpl@5c2a25"
注入的self也
preFiltered = true
advisorChainFactory = {org.springframework.aop.framework.DefaultAdvisorChainFactory@3888}
methodCache = {java.util.concurrent.ConcurrentHashMap@3889} size = 2
interfaces = {java.util.ArrayList@3890} size = 2
advisors = {java.util.LinkedList@3891} size = 1
advisorArray = {org.springframework.aop.Advisor[1]@3892}
proxyTargetClass = false
optimize = false
opaque = false
exposeProxy = true
frozen = false
equalsDefined = false
hashCodeDefined = false



（2. 5、这是为什么呢，怎么在循环依赖会出现这种情况？

敬请期待我的下一篇分析帖。

3.4、改进版的InjectBeanSelfProcessor的解决方案

```
@Component
public class InjectBeanSelfProcessor2 implements BeanPostProcessor, ApplicationContextAware {
    private ApplicationContext context;
    //① 注入ApplicationContext
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansExcept
        this.context = applicationContext;
    }
```

```
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansExce
        if(!(bean instanceof BeanSelfAware)) { //② 如果Bean没有实现BeanSelfAware标识接口 跳过
            return bean;
        }
        if(AopUtils.isAopProxy(bean)) { //③ 如果当前对象是AOP代理对象，直接注入
            ((BeanSelfAware) bean).setSelf(bean);
        } else {
            //④ 如果当前对象不是AOP代理，则通过context.getBean(beanName)获取代理对象并注入
            //此种方式不适合解决prototype Bean的代理对象注入
            ((BeanSelfAware)bean).setSelf(context.getBean(beanName));
        }
        return bean;
    }
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansExc
        return bean;
    }
}
```

5、总结

纵观其上：

【3.1 通过ThreadLocal暴露Aop代理对象】适合解决所有场景（不管是singleton Bean还是prototype Bean）的AOP代理获取问题（即能解决目标对象的自我调用问题）；

【3.2 通过初始化方法在目标对象中注入代理对象】 和【3.4 改进版的InjectBeanSelfProcessor的解决方案】能解决普通（无循环依赖）的AOP代理对象注入问题，而且也能解决【3.3】中提到的循环依赖（应该是singleton之间的循环依赖）造成的目标对象无法注入AOP代理对象问题，但该解决方案不适合解决循环依赖中包含prototype Bean的自我调用问题；

【3.3 通过BeanPostProcessor 在目标对象中注入代理对象】：只能解决 普通（无循环依赖）的 的Bean注入AOP代理，无法解决循环依赖的AOP代理对象注入问题，即无法解决目标对象的自我调用问题。

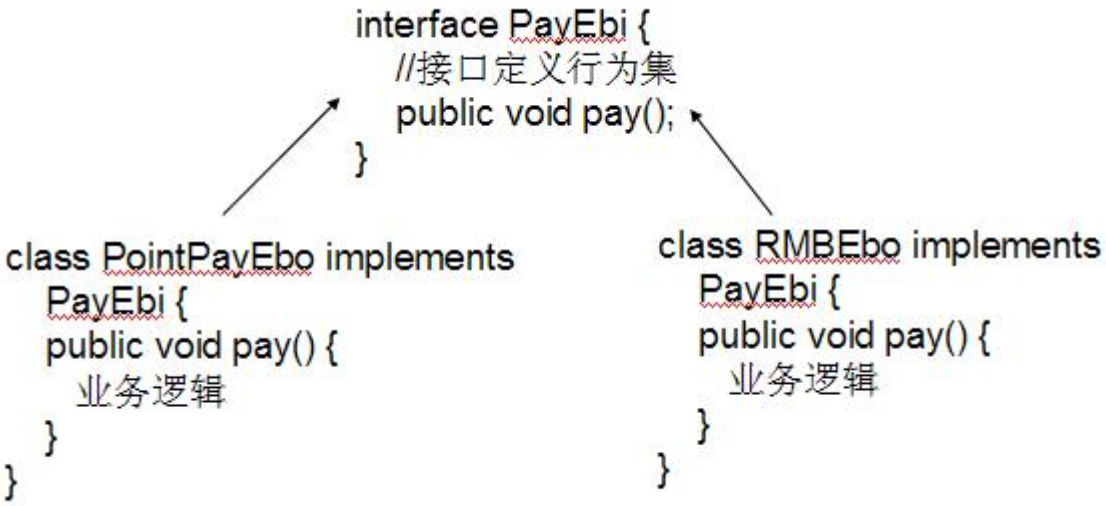
没有完美的解决方案，只有最适用的解决方案。

测试代码请参考附件，jar包与http://www.iteye.com/topic/1120924使用的是一样的

1.2 我对AOP的理解

发表时间: 2012-04-05

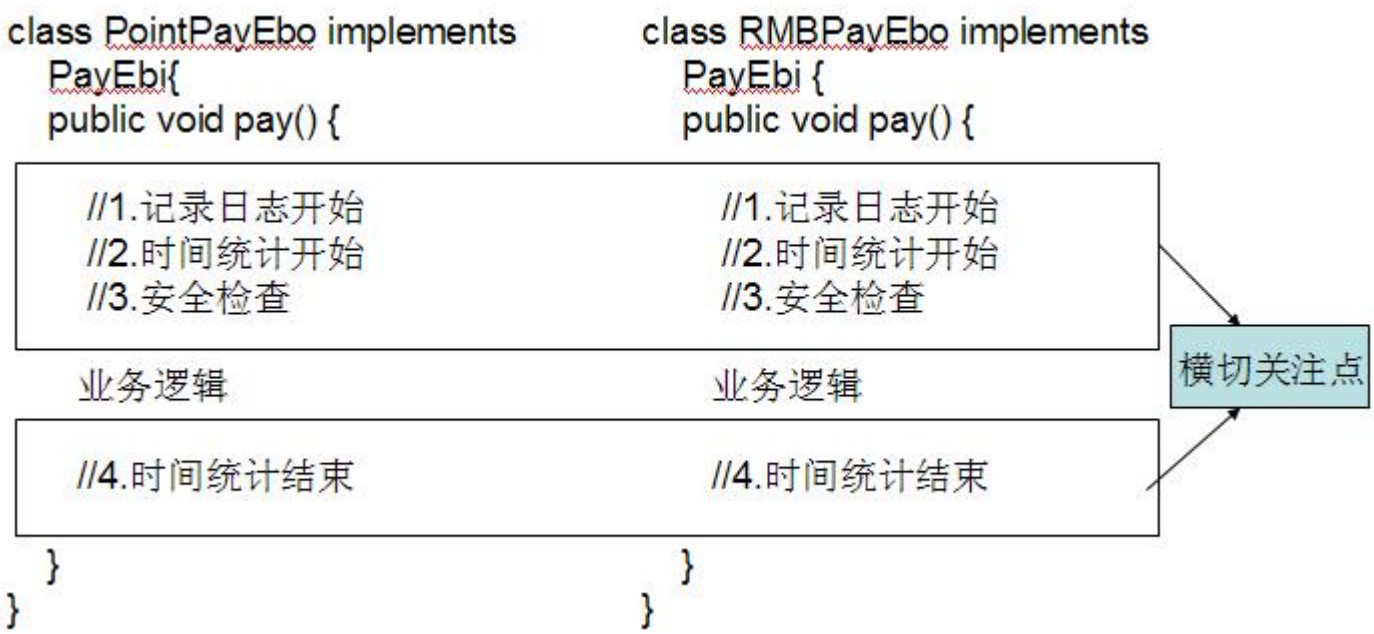
1、问题



问题：想要添加日志记录、性能监控、安全监测

2、最初解决方案

2.1、最初解决方案

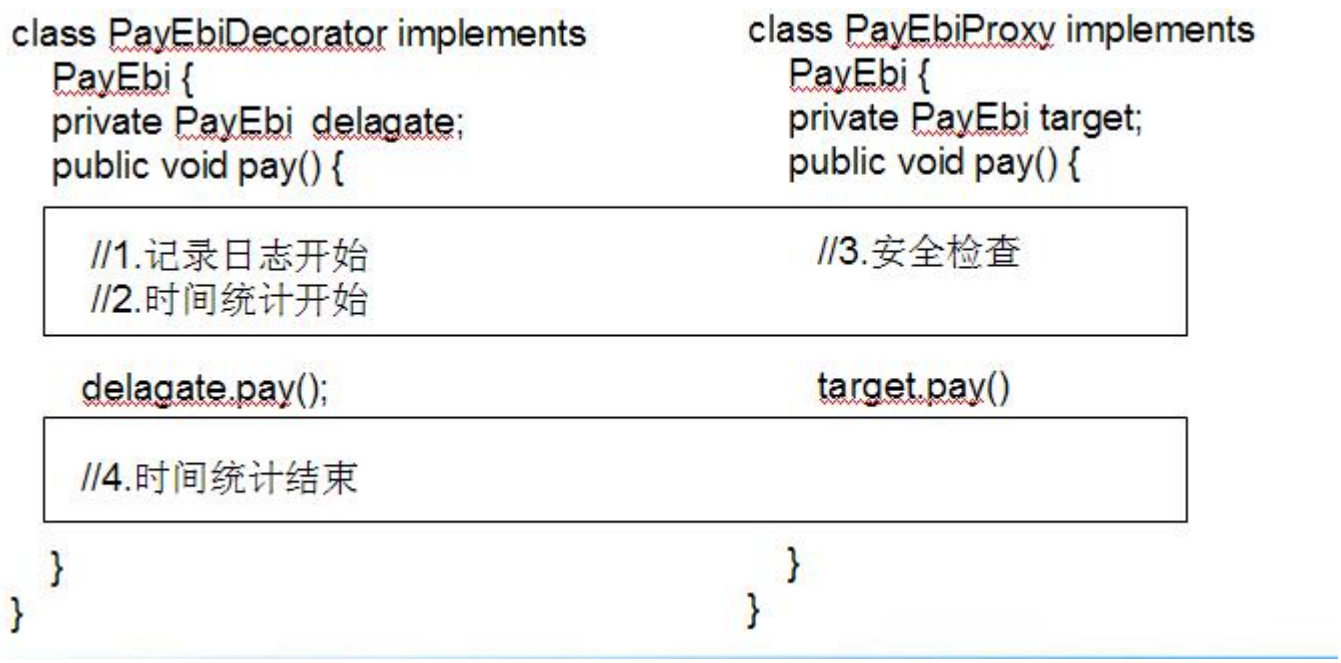


缺点：太多重复代码，且紧耦合

2.2、抽象类进行共性设计，子类进行个性设计，此处不讲解，缺点一荣俱荣，一损俱损

2.3、使用装饰器模式/代理模式改进的解决方案

装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说， 装饰器模式相比生成子类更为灵活。
代理模式：为其他对象提供一种代理以控制对这个对象的访问。



缺点：紧耦合，每个业务逻辑需要一个装饰器实现或代理

2.4、JDK动态代理解决方案（比较通用的解决方案）

```
public class MyInvocationHandler implements InvocationHandler {
    private Object target;
    public MyInvocationHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //1.记录日志    2.时间统计开始      3.安全检查
        Object retVal = method.invoke(target, args);
        //4.时间统计结束
        return retVal;
    }
    public static Object proxy(Object target) {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(), new MyInvocationHandler(target));
    }
}
```

编程模型

```
    //proxy      在其上调用方法的代理实例
    //method 拦截的方法
    //args       拦截的参数
    Override
     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
         Object retVal=null;
         //预处理
         //前置条件判断
         boolean ok = true;
         if(!ok) {//不满足条件
             throw new RuntimeException("你没有权限");
         }
         else {//反射调用目标对象的某个方法
             retVal = method.invoke(target, args);
         }
         //后处理
         return retVal;
     }
```

缺点：使用麻烦，不能代理类，只能代理接口

CGLIB动态代理解决方案（比较通用的解决方案）

```
public class MyInterceptor implements MethodInterceptor  {
    private Object target;
    public MyInterceptor(Object target) {
        this.target = target;
    }
    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
                                          MethodProxy invocation) throws Throwable {
        //1.记录日志 2.时间统计开始    3.安全检查
        Object retVal = invocation.invoke(target, args);
        //4.时间统计结束
        return retVal;
```

```
    }
    public static Object proxy(Object target) {
        return Enhancer.create(target.getClass(), new MyInterceptor(target));
    }
}
```

编程模型

```
 //proxy 在其上调用方法的代理实例     method拦截的方法     args   拦截的参数
 //invocation 用来去调用被代理对象方法的
@Override
public Object intercept(Object proxy, Method method, Object[] args,
                                        MethodProxy invocation) throws Throwable {
    //预处理
    //前置条件判断
    boolean ok = true;
    if(!ok) {//不满足条件
        throw new RuntimeException("出错了");
    }
    else {//调用目标对象的某个方法
        Object retVal = invocation.invoke(target, args);
    }
    //后处理
    return retVal;
}
```

优点：能代理接口和类

缺点：使用麻烦，不能代理final类

动态代理本质

本质：对目标对象增强

最终表现为类（动态创建子类），看手工生成（子类）还是自动生成（子类）

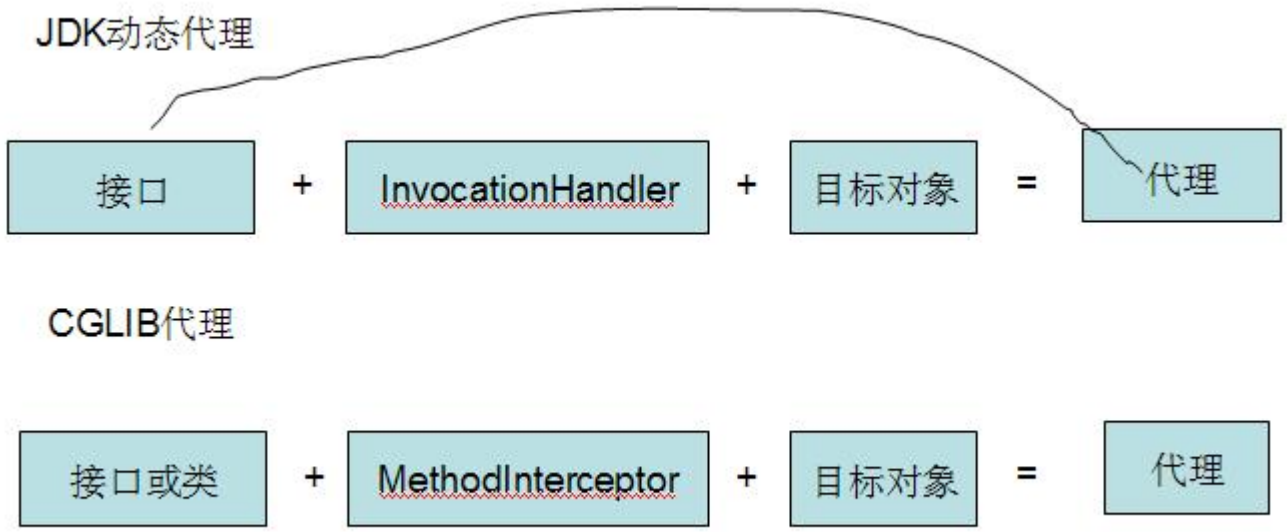
代理限制：

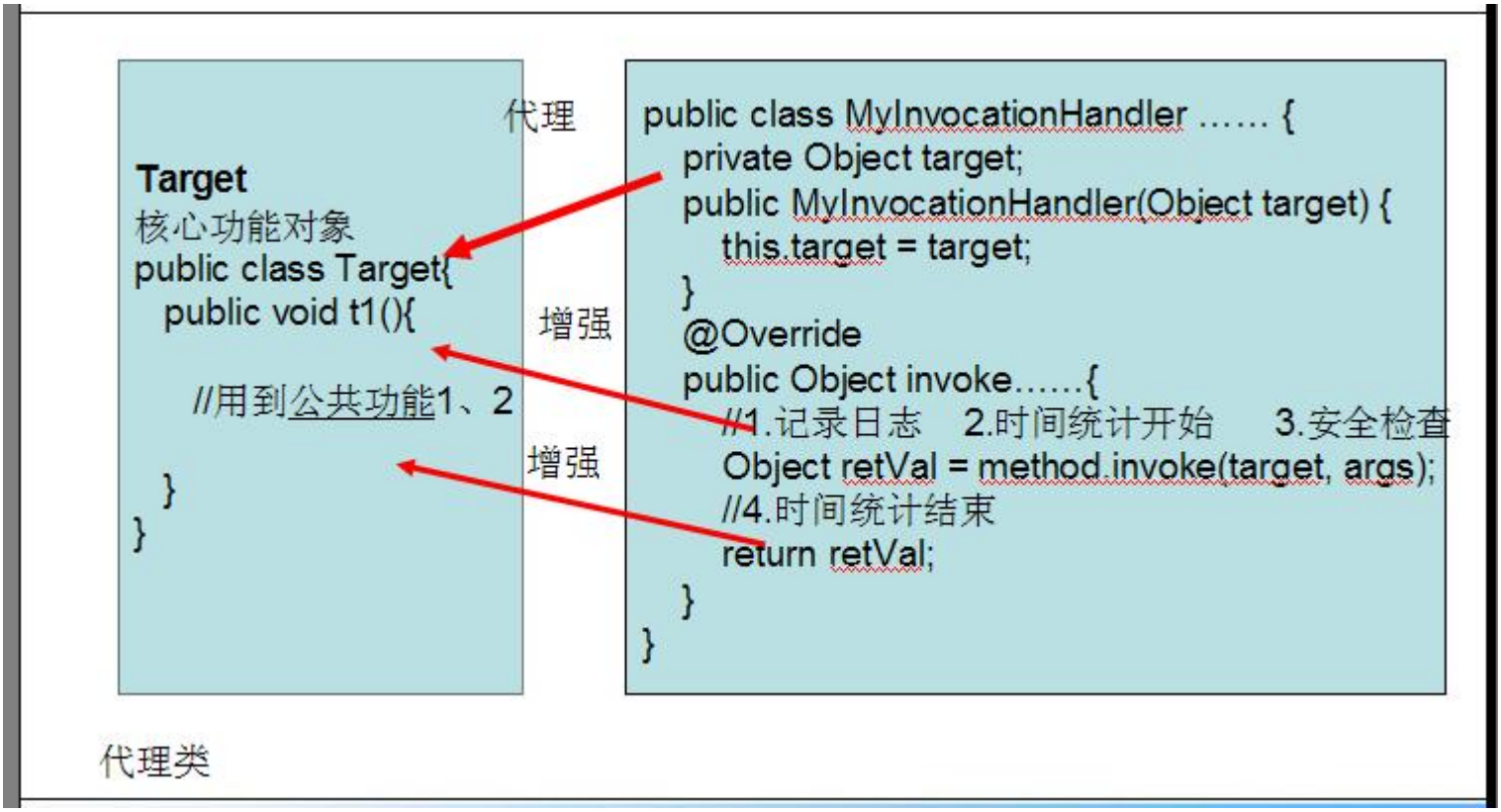
只能在父类方法被调用之前或之后进行增强（功能的修改），不能在中间进行修改，要想在方法调用中增强，需要ASM(java 字节码生成库)

其他动态代理框架

jboss：javassist（hibernate 3.3中默认为javassist）

（hibernate 3.3之前中默认为cglib）





2.5、AOP解决方案（通用且简单的解决方案）

```
@Aspect
public class PayEbiAspect {
```

```
    @Pointcut(value="execution(* pay(..))")
    public void pointcut() {}
    @Around(value="pointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        //1.记录日志
        //2.时间统计开始
        //3.安全检查
        Object retVal = pjp.proceed();//调用目标对象的真正方法
        //4.时间统计结束
        return retVal;
    }
}
```

编程模型

```
    //2 切入点
    @Pointcut(value="execution(* *(..))")
    public void pointcut() {}
    //3 拦截器的interceptor
    @Around(value="pointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        Object retVal=null;
        //预处理
        //前置条件判断
        boolean ok = true;
        if(!ok) {//不满足条件
           throw new RuntimeException("你没有权限");
        }
        else {//调用目标对象的某个方法
             retVal = pjp.proceed();
        }
        //后处理
        return retVal;
    }
```

缺点：依赖AOP框架

AOP入门

概念：

n**关注点：**可以认为是所关注的任何东西，比如上边的支付组件；

n**关注点分离：**将问题细化为单独部分，即可以理解为不可再分割的组件，如上边的日志组件和支付组件；

n**横切关注点：**会在多个模块中出现，使用现有的编程方法，横切关注点会横越多个模块，结果是使系统难以设计、理解、实现和演进，如日志组件横切于支付组件。

织入：横切关注点分离后，需要通过某种技术将横切关注点融合到系统中从而完成需要的功能，因此需要织入，织入可能在编译期、加载期、运行期等进行。

nAOP是什么(Aspect Oriented Programming)

AOP是一种编程范式，提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。

AOP为开发者提供了一种描述横切关注点的机制，并能够自动将横切关注点织入到面向对象的软件系统中，从而实现了横切关注点的模块化。

AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

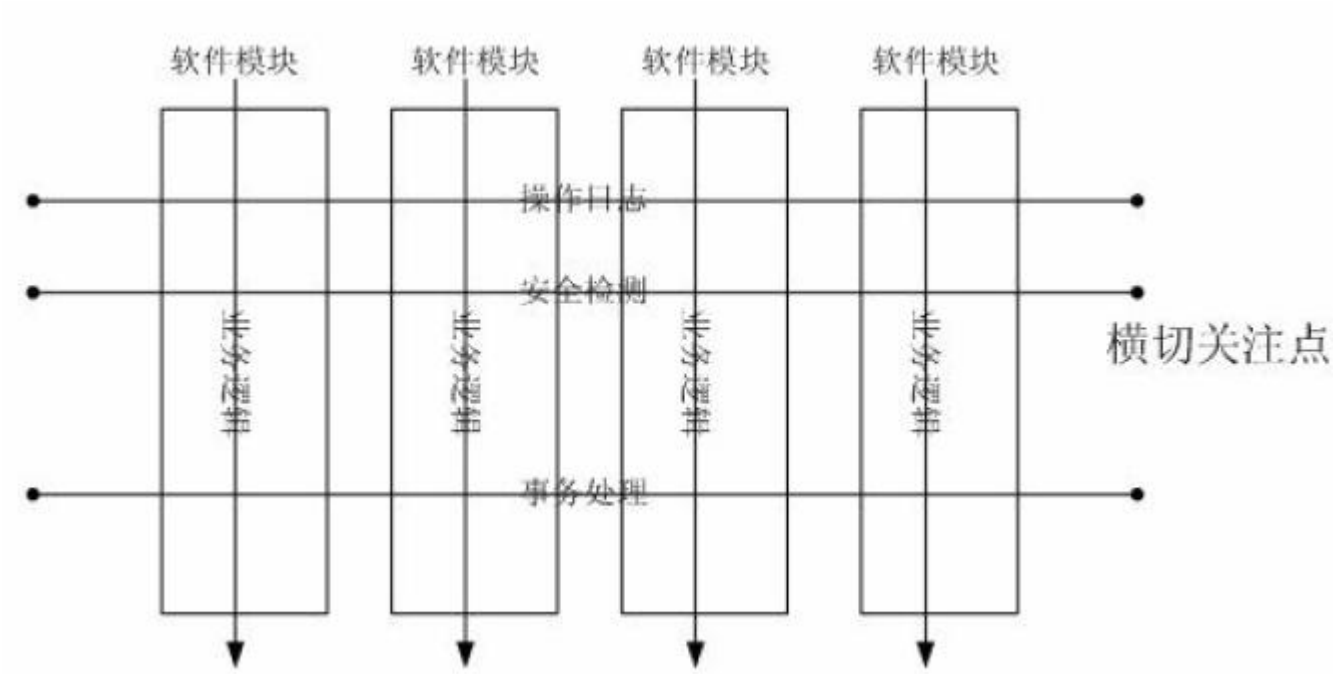
nAOP能干什么，也是AOP带来的好处

1：降低模块的耦合度

2：使系统容易扩展

3：设计决定的迟绑定：使用AOP,设计师可以推迟为将来的需求作决定，因为它
可以把这种需求作为独立的方面很容易的实现。

4：更好的代码复用性



AOP基本概念

连接点（Joinpoint）：

表示需要在程序中插入横切关注点的扩展点，连接点可能是类初始化、方法执行、方法调用、字段调用或处理异常等等，Spring只支持方法执行连接点，**在AOP中表示为“在哪里做”**；

切入点（Pointcut）：

选择一组相关连接点的模式，即可以认为连接点的集合，Spring支持perl5正则表达式和AspectJ切入点模式，Spring默认使用AspectJ语法，**在AOP中表示为“在哪里做的集合”**；

增强（Advice）：或称为增强

在连接点上执行的行为，增强提供了在AOP中需要在切入点所选择的连接点处进行扩展现有行为的手段；包括前置增强（before advice）、后置增强 (after advice)、环绕增强（around advice），在Spring中通过代理模式实现AOP，并通过拦截器模式以环绕连接点的拦截器链织入增强；**在AOP中表示为“做什么”；**

方面/切面（Aspect）：

横切关注点的模块化，比如上边提到的日志组件。可以认为是增强、引入和切入点的组合；在Spring中可以使用Schema和@AspectJ方式进行组织实现；**在AOP中表示为“在哪里做和做什么集合”；**

目标对象（Target Object）：

需要被织入横切关注点的对象，即该对象是切入点选择的对象，需要被增强的对象，从而也可称为“被增强对象”；由于Spring AOP 通过代理模式实现，从而这个对象永远是被代理对象，**在AOP中表示为“对谁做”**；

AOP代理（AOP Proxy）：

AOP框架使用代理模式创建的对象，从而实现在连接点处插入增强（即应用切面），就是**通过代理来对目标对象应用切面。**在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。

织入（Weaving）：

织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程，织入可以在编译期、类装载期、运行期进行。

引入（inter-type declaration）：

也称为内部类型声明，为已有的类添加额外新的字段或方法，Spring允许引入新的接口（必须对应一个实现）到所有被代理对象（目标对象），**在AOP中表示为“做什么（新增什么）”**；

AOP的Advice类型

前置增强（Before advice）：

在某连接点之前执行的增强，但这个增强不能阻止连接点前的执行（除非它抛出一个异常）。

后置返回增强（After returning advice）：

在某连接点正常完成后执行的增强：例如，一个方法没有抛出任何异常，正常返回。

后置异常增强（After throwing advice）：

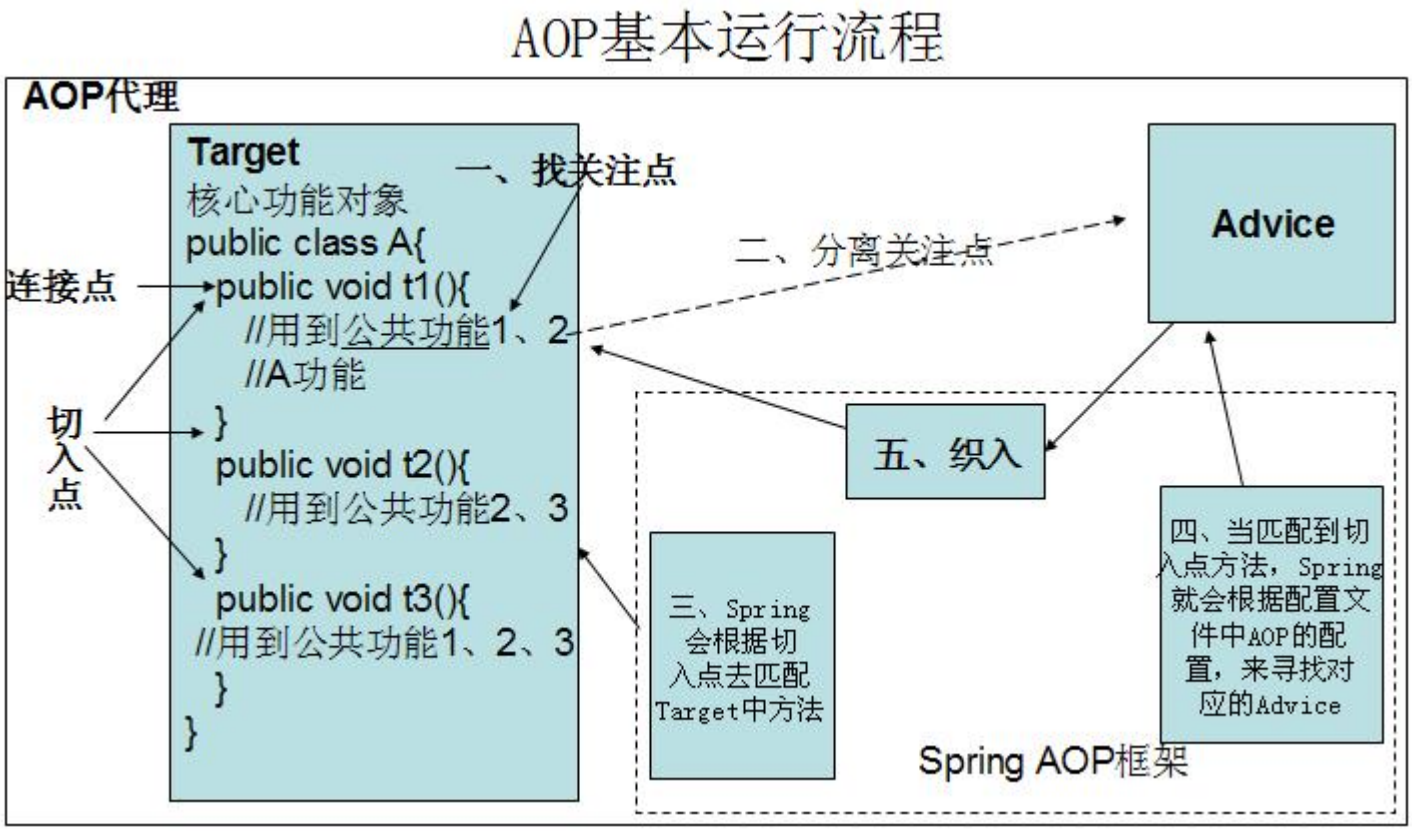
在方法抛出异常退出时执行的增强。

后置最终增强（After (finally) advice）：

当某连接点退出的时候执行的增强（不论是正常返回还是异常退出）。

环绕增强（Around Advice）：

包围一个连接点的增强，如方法调用。这是最强大的一种增强类型。 环绕增强可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。



AOP开发步骤

类似于IoC/DI容器开发步骤，需要描述哪个连接点需要哪个通用功能（增强）

参考：http://www.iteye.com/topic/1122310

横切关注点的表现有：

□**代码纠结/混乱**——当一个模块或代码段同时管理多个关注点时发生这种情况。如我既要实现业务、还要实现安全和事务。即有些关注点同时被多个不同的模块实现。实现了重复的功能。

□**代码分散**——当一个关注点分布在许多模块中并且未能很好地局部化和模块化时发生这种情况 。如许多模块调用用户是否登录验证代码。调用了重复的功能。

AOP包括三个清晰的开发步骤：

1：**功能横切**：找出横切关注点。

2：**实现分离**：各自独立的实现这些横切关注点所需要完成的功能。

3：**功能回贴**：在这一步里，方面集成器通过创建一个模块单元—— 方面来指定重组的规则。重组过程——也叫织入或结合—— 则使用这些信息来构建最终系统。

推荐阅读书籍：

AspectJ in Action

AOSD中文版--基于用例的面向方面软件开发

推荐阅读的帖子：

AOP的实现机制

文章主要是为了抛砖引玉，希望有更多牛人的指点。

1.3 Spring开闭原则的表现-BeanPostProcessor的扩展点-1

发表时间: 2012-04-18

上接Spring事务处理时自我调用的解决方案及一些实现方式的风险继续分析，在分析上篇的问题之前，我们需要了解下BeanPostProcessor概念和Spring容器创建Bean的流程。

一、BeanPostProcessor是什么

接口定义

```
package org.springframework.beans.factory.config;
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
    Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
}
```

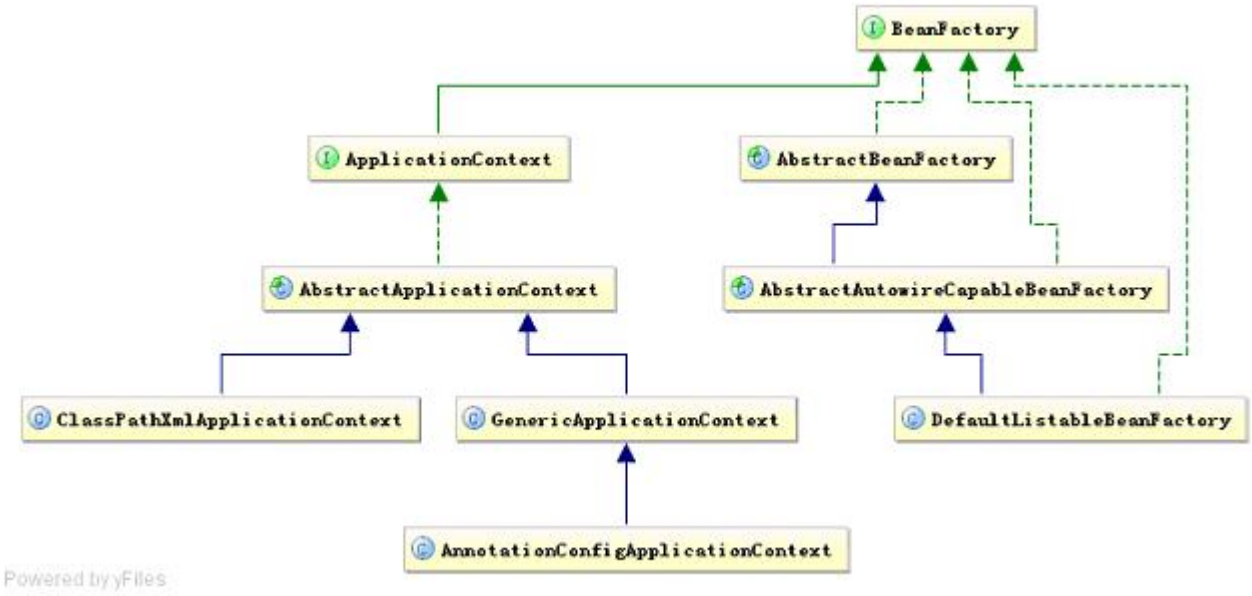
BeanPostProcessor是Spring容器的一个扩展点，可以进行自定义的实例化、初始化、依赖装配、依赖检查等流程，即可以覆盖默认的实例化，也可以增强初始化、依赖注入、依赖检查等流程，其javadoc有如下描述：

> e.g. checking for marker interfaces or wrapping them with proxies.
>
> 大体意思是可以检查相应的标识接口完成一些自定义功能实现，如包装目标对象到代理对象。

我们可以看到BeanPostProcessor一共有两个回调方法postProcessBeforeInitialization和postProcessAfterInitialization，那这两个方法会在什么Spring执行流程中的哪个步骤执行呢？还有目前Spring提供哪些相应的实现呢？

Spring还提供了BeanPostProcessor一些其他接口实现，来完成除实例化外的其他功能，后续详细介绍。

二、通过源代码看看创建一个Bean实例的具体执行流程：



AbstractApplicationContext内部使用DefaultListableBeanFactory，且DefaultListableBeanFactory继承AbstractAutowireCapableBeanFactory，因此我们此处分析AbstractAutowireCapableBeanFactory即可。

一、AbstractAutowireCapableBeanFactory的createBean方法代码如下：

```
protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[]
    resolveBeanClass(mbd, beanName); /1解析Bean的class
    mbd.prepareMethodOverrides(); //2 方法注入准备
    Object bean = resolveBeforeInstantiation(beanName, mbd); //3 第一个BeanPostProcessor扩展点
    if (bean != null) { //4 如果3处的扩展点返回的bean不为空，直接返回该bean，后续流程不需要执行
            return bean;
    }
    Object beanInstance = doCreateBean(beanName, mbd, args); //5 执行spring的创建bean实例的流程啦
    return beanInstance;
}
```

0.3 第一个BeanPostProcessor扩展点（只有InstantiationAwareBeanPostProcessor接口的实现才会被调用）

二、AbstractAutowireCapableBeanFactory的resolveBeforeInstantiation方法代码如下：

```
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
                Object bean = null;
                if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
                        // Make sure bean class is actually resolved at this point.
                        if (mbd.hasBeanClass() && !mbd.isSynthetic() && hasInstantiationAwareBe
                                //3.1、执行InstantiationAwareBeanPostProcessor的postProcessBefo
                                bean = applyBeanPostProcessorsBeforeInstantiation(mbd.getBeanCl
                                if (bean != null) {
                                        //3.2、执行InstantiationAwareBeanPostProcessor的postPro
                                        bean = applyBeanPostProcessorsAfterInitialization(bean,
                                }
                        }
                        mbd.beforeInstantiationResolved = (bean != null);
                }
                return bean;
}
```

通过如上代码可以进行实例化的预处理（自定义实例化bean，如创建相应的代理对象）和后处理（如进行自定义实例化的bean的依赖装配）。

三、AbstractAutowireCapableBeanFactory的doCreateBean方法代码如下：

```
        // 6、通过BeanWrapper实例化Bean
                BeanWrapper instanceWrapper = null;
                if (mbd.isSingleton()) {
                        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
                }
                if (instanceWrapper == null) {
                        instanceWrapper = createBeanInstance(beanName, mbd, args);
                }
                final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstar
```

```
Class beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() :

//7、执行MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition流程
synchronized (mbd.postProcessingLock) {
        if (!mbd.postProcessed) {
                applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName
                mbd.postProcessed = true;
        }
}
// 8、及早暴露单例Bean引用，从而允许setter注入方式的循环引用
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularRefere
                isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
        //省略log
        addSingletonFactory(beanName, new ObjectFactory() {
                public Object getObject() throws BeansException {
                        //8.1、调用SmartInstantiationAwareBeanPostProcessor的ge
                        return getEarlyBeanReference(beanName, mbd, bean);
                }
        });
}

Object exposedObject = bean;
try {
        populateBean(beanName, mbd, instanceWrapper); //9、组装-Bean依赖
        if (exposedObject != null) {
                exposedObject = initializeBean(beanName, exposedObject, mbd); /
        }
}
catch (Throwable ex) {
    //省略异常
}


//11如果是及早暴露单例bean，通过getSingleton触发3.1处的getEarlyBeanReference调用获
if (earlySingletonExposure) {
        Object earlySingletonReference = getSingleton(beanName, false);
```

```
                if (earlySingletonReference != null) {
                        if (exposedObject == bean) {
                                exposedObject = earlySingletonReference;
                        }
                        else if (!this.allowRawInjectionDespiteWrapping && hasDependent
                                String[] dependentBeans = getDependentBeans(beanName);
                                Set<String> actualDependentBeans = new LinkedHashSet<St
                                for (String dependentBean : dependentBeans) {
                                        if (!removeSingletonIfCreatedForTypeCheckOnly(d
                                                actualDependentBeans.add(dependentBean)
                                        }
                                }
                                if (!actualDependentBeans.isEmpty()) {
                                        throw new BeanCurrentlyInCreationException(bean
                                                        "Bean with name '" + beanName +
                                                        StringUtils.collectionToCommaDe
                                                        "] in its raw version as part o
                                                        "wrapped. This means that said
                                                        "bean. This is often the result
                                                        "'getBeanNamesOfType' with the
                                }
                        }
                }
        }
        //12、注册Bean的销毁回调
        try {
                registerDisposableBeanIfNecessary(beanName, bean, mbd);
        }
        catch (BeanDefinitionValidationException ex) {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        }
```

四、AbstractAutowireCapableBeanFactory的populateBean方法代码如下：

```
//9、组装-Bean
	protected void populateBean(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw
		PropertyValues pvs = mbd.getPropertyValues();
		//省略部分代码
		//9.1、通过InstantiationAwareBeanPostProcessor扩展点允许自定义装配流程（如@Autowir
		//执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation
		boolean continueWithPropertyPopulation = true;
		if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
			for (BeanPostProcessor bp : getBeanPostProcessors()) {
				if (bp instanceof InstantiationAwareBeanPostProcessor) {
					InstantiationAwareBeanPostProcessor ibp = (Instantiatio
					if (!ibp.postProcessAfterInstantiation(bw.getWrappedIns
						continueWithPropertyPopulation = false;
						break;
					}
				}
			}
		}
		if (!continueWithPropertyPopulation) {
			return;
		}
		if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
				mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY
			MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
			// 9. 2、自动装配（根据name/type）
			if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAM
				autowireByName(beanName, mbd, bw, newPvs);
			}
			if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYP
				autowireByType(beanName, mbd, bw, newPvs);
			}
			pvs = newPvs;
		}
		boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
		boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDE
```

```
        //9. 3、执行InstantiationAwareBeanPostProcessor的postProcessPropertyValues
        if (hasInstAwareBpps || needsDepCheck) {
                PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDepender
                if (hasInstAwareBpps) {
                        for (BeanPostProcessor bp : getBeanPostProcessors()) {
                                if (bp instanceof InstantiationAwareBeanPostProcessor)
                                        InstantiationAwareBeanPostProcessor ibp = (Inst
                                        pvs = ibp.postProcessPropertyValues(pvs, filter
                                        if (pvs == null) {
                                                return;
                                        }
                                }
                        }
                }
                //9. 4、执行依赖检查
                if (needsDepCheck) {
                        checkDependencies(beanName, mbd, filteredPds, pvs);
                }
        }
        //9. 5、应用依赖注入
        applyPropertyValues(beanName, mbd, bw, pvs);
 }
```

五、AbstractAutowireCapableBeanFactory的initializeBean方法代码如下：

```
    //10、实例化Bean
    protected Object initializeBean(final String beanName, final Object bean, RootBeanDefini
            //10.1、调用Aware接口注入（BeanNameAware、BeanClassLoaderAware、BeanFactoryAware
            invokeAwareMethods(beanName, bean);//此处省略部分代码
            //10.2、执行BeanPostProcessor扩展点的postProcessBeforeInitialization进行修改实例化
            Object wrappedBean = bean;
            if (mbd == null || !mbd.isSynthetic()) {
                    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
            }
            //10.3、执行初始化回调（1、调用InitializingBean的afterPropertiesSet  2、调用自定义
```

```
            try {
                    invokeInitMethods(beanName, wrappedBean, mbd);
            }
            catch (Throwable ex) {
                    //异常省略
            }
            //10.4、执行BeanPostProcessor扩展点的postProcessAfterInitialization进行修改实例化
            if (mbd == null || !mbd.isSynthetic()) {
                    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, b
            }
            return wrappedBean;
    }
```

三、创建一个Bean实例的执行流程简化：

protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[] args); 创建Bean

（1、resolveBeanClass(mbd, beanName); 解析Bean class，若class配置错误将抛出CannotLoadBeanClassException；

（2、mbd.prepareMethodOverrides(); 准备和验证配置的方法注入，若验证失败抛出BeanDefinitionValidationException

有关方法注入知识请参考【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3 3.3.5 方法注入；

（3、Object bean = resolveBeforeInstantiation(beanName, mbd); 第一个BeanPostProcessor扩展点，此处只执行InstantiationAwareBeanPostProcessor类型的BeanPostProcessor Bean；

（3.1、bean = applyBeanPostProcessorsBeforeInstantiation(mbd.getBeanClass(), beanName);执行InstantiationAwareBeanPostProcessor的实例化的预处理回调方法postProcessBeforeInstantiation（自定义的实例化，如创建代理）；

（3.2、bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);执行InstantiationAwareBeanPostProcessor的实例化的后处理回调方法postProcessAfterInitialization（如依赖注入），如果3.1处返回的Bean不为null才执行；

（4、如果3处的扩展点返回的bean不为空，直接返回该bean，后续流程不需要执行；

（5、Object beanInstance = doCreateBean(beanName, mbd, args); 执行spring的创建bean实例的流程；

（6、createBeanInstance(beanName, mbd, args); 实例化Bean

（6.1、instantiateUsingFactoryMethod 工厂方法实例化；请参考【http://jinnianshilongnian.iteye.com/blog/1413857】

（6.2、构造器实例化，请参考【http://jinnianshilongnian.iteye.com/blog/1413857】；

（6.2.1、如果之前已经解析过构造器

（6.2.1.1 autowireConstructor：有参调用autowireConstructor实例化

（6.2.1.2、instantiateBean：无参调用instantiateBean实例化；

（6.2.2、如果之前没有解析过构造器：

（6.2.2.1、通过SmartInstantiationAwareBeanPostProcessor的determineCandidateConstructors回调方法解析构造器，第二个BeanPostProcessor扩展点，返回第一个解析成功（返回值不为null）的构造器组，如AutowiredAnnotationBeanPostProcessor实现将自动扫描通过@Autowired/@Value注解的构造器从而可以完成构造器注入，请参考【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3 ；

（6.2.2.2、autowireConstructor：如果（6.2.2.1返回的不为null，且是有参构造器，调用autowireConstructor实例化；

（6.2.2.3、instantiateBean： 否则调用无参构造器实例化；

（7、applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);第三个BeanPostProcessor扩展点，执行Bean定义的合并；

（7.1、执行MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition回调方法，进行bean定义的合并；

（8、addSingletonFactory(beanName, new ObjectFactory() {

public Object getObject() throws BeansException {

return getEarlyBeanReference(beanName, mbd, bean);

}

}); 及早暴露单例Bean引用，从而允许setter注入方式的循环引用

（8.1、SmartInstantiationAwareBeanPostProcessor的getEarlyBeanReference；第四个BeanPostProcessor扩展点，当存在循环依赖时，通过该回调方法获取及早暴露的Bean实例；

（9、populateBean(beanName, mbd, instanceWrapper);装配Bean依赖

（9.1、InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation；第五个BeanPostProcessor扩展点，在实例化Bean之后，所有其他装配逻辑之前执行，如果false将阻止其他的InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation的执行和从（9.2到（9.5的执行，通常返回true；

（9.2、autowireByName、autowireByType：根据名字和类型进行自动装配，自动装配的知识请参考【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3 3.3.3 自动装配；

（9.3、InstantiationAwareBeanPostProcessor的postProcessPropertyValues：第六个BeanPostProcessor扩展点，完成其他定制的一些依赖注入，如AutowiredAnnotationBeanPostProcessor执行@Autowired注解注入，CommonAnnotationBeanPostProcessor执行@Resource等注解的注入，PersistenceAnnotationBeanPostProcessor执行@ PersistenceContext等JPA注解的注入，RequiredAnnotationBeanPostProcessor执行@ Required注解的检查等等，请参考【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3；

（9.4、checkDependencies：依赖检查，请参考【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3 3.3.4 依赖检查；

（9.5、applyPropertyValues：应用明确的setter属性注入，请参考【第三章】 DI 之 3.1 DI的配置使用 ——跟我学spring3 ；

（10、exposedObject = initializeBean(beanName, exposedObject, mbd); 执行初始化Bean流程；

（10.1、invokeAwareMethods（BeanNameAware、BeanClassLoaderAware、BeanFactoryAware）：调用一些Aware标识接口注入如BeanName、BeanFactory；

（10.2、BeanPostProcessor的postProcessBeforeInitialization：第七个扩展点，在调用初始化之前完成一些定制的初始化任务，如BeanValidationPostProcessor完成JSR-303 @Valid注解Bean验证，InitDestroyAnnotationBeanPostProcessor完成@PostConstruct注解的初始化方法调用，ApplicationContextAwareProcessor完成一些Aware接口的注入（如EnvironmentAware、ResourceLoaderAware、ApplicationContextAware），其返回值将替代原始的Bean对象；

（10.3、invokeInitMethods ： 调用初始化方法；

（10.3.1、InitializingBean的afterPropertiesSet ：调用InitializingBean的afterPropertiesSet回调方法；

（10.3.2、通过xml指定的自定义init-method ：调用通过xml配置的自定义init-method

（10.3.3、BeanPostProcessor的postProcessAfterInitialization ：第八个扩展点，AspectJAwareAdvisorAutoProxyCreator（完成xml风格的AOP配置(<aop:config>)的目标对象包装到AOP代理对象）、AnnotationAwareAspectJAutoProxyCreator（完成@Aspectj注解风格（<aop:aspectj-autoproxy> @Aspect）的AOP配置的目标对象包装到AOP代理对象），其返回值将替代原始的Bean对象；

（11、if (earlySingletonExposure) {

Object earlySingletonReference = getSingleton(beanName, false);

……

}：如果是earlySingleExposure，调用getSingle方法获取Bean实例；

earlySingleExposure =(mbd.isSingleton() && this.allowCircularReferences && isSingletonCurrentlyInCreation(beanName))

只要单例Bean且允许循环引用（默认true）且当前单例Bean正在创建中

（11.1、如果是earlySingletonExposure调用getSingleton将触发【8】处ObjectFactory.getObject()的调用，通过【8.1】处的getEarlyBeanReference获取相关Bean（如包装目标对象的代理Bean）；（在循环引用Bean时可能引起Spring事务处理时自我调用的解决方案及一些实现方式的风险）；

-

（12、registerDisposableBeanIfNecessary(beanName, bean, mbd) ： 注册Bean的销毁方法（只有非原型Bean可注册）；

（12.1、单例Bean的销毁流程

（12.1.1、DestructionAwareBeanPostProcessor的postProcessBeforeDestruction ： 第九个扩展点，如InitDestroyAnnotationBeanPostProcessor完成@PreDestroy注解的销毁方法注册和调用；

（12.1.2、DisposableBean的destroy：注册/调用DisposableBean的destroy销毁方法；

（12.1.3、通过xml指定的自定义destroy-method ： 注册/调用通过XML指定的destroy-method销毁方法；

（12.1.2、Scope的registerDestructionCallback：注册自定义的Scope的销毁回调方法，如RequestScope、SessionScope等；其流程和【12.1 单例Bean的销毁流程一样】，关于自定义Scope请参考【第三章】 DI 之 3.4 Bean的作用域 ——跟我学spring3

（13、到此Bean实例化、依赖注入、初始化完毕可以返回创建好的bean了。

从上面的流程我们可以看到BeanPostProcessor一个使用了九个扩展点，其实还一个扩展点（SmartInstantiationAwareBeanPostProcessor的predictBeanType在下一篇介绍），接下来我们看看BeanPostProcessor这些扩展点都主要完成什么功能及常见的BeanPostProcessor。

我将在下一帖子中使用例子来解析这八个扩展点的主要功能，及一些Spring默认提供的BeanPostProcessor主要作用。

上接Spring事务处理时自我调用的解决方案及一些实现方式的风险继续分析，在分析上篇的问题之前，我们需要了解下BeanPostProcessor概念和Spring容器创建Bean的流程。

1.4 我对IoC/DI的理解

发表时间: 2012-03-31 关键字: IoC, DI

IoC

IoC： Inversion of Control，控制反转， 控制权从应用程序转移到框架（如IoC容器），是框架共有特性

1、为什么需要IoC容器

1.1、应用程序主动控制对象的实例化及依赖装配

```
A a = new AImpl();
B b = new BImpl();
a.setB(b);
```

本质：创建对象，主动实例化，直接获取依赖，主动装配

缺点：更换实现需要重新编译源代码

很难更换实现、难于测试

耦合实例生产者和实例消费者

```
A a = AFactory.createA();
B b = BFactory.createB();
a.setB(b);
```

本质：创建对象，被动实例化，间接获取依赖，主动装配 （简单工厂）

缺点：更换实现需要重新编译源代码

很难更换实现、难于测试

```
A a = Factory.create( “a” );
B b = Factory.create( “b” );
a.setB(b);
```

<!—配置.properties-->

```
a=AImpl
b=BImpl
```

本质：创建对象，被动实例化，间接获取依赖， 主动装配
（工厂+反射+properties配置文件、
Service Locator、注册表）
缺点：冗余的依赖装配逻辑

我想直接：
//返回装配好的a

```
A a = Factory.create( “a” );
```

1.2、可配置通用工厂：**工厂主动控制**，应用程序被动接受，控制权从应用程序转移到工厂

```
//返回装配好的a
A a = Factory.create( “a” );
```

<!—配置文件-->

```
<bean id= “a”  class= “AImpl” >
    <property name= “b”  ref= “b” />
</bean>
<bean id= “b”  class= “BImpl” />
```

本质：创建对象和装配对象，
被动实例化，被动接受依赖，被动装配
（工厂+反射+xml配置文件）

缺点：不通用

步骤：

1、读取配置文件根据配置文件通过反射

创建AImpl

2、发现A需要一个类型为B的属性b

3、到工厂中找名为b的对象，发现没有，读取

配置文件通过反射创建BImpl

4、将b对象装配到a对象的b属性上

【组件的配置与使用分离开（解耦、更改实现无需修改源代码、易于更好实现）】

1.3、 IoC(控制反转)容器：**容器主动控制**

```
//返回装配好的a
A a = ApplicationContext.getBean( “a” );
```

<!—配置文件-->

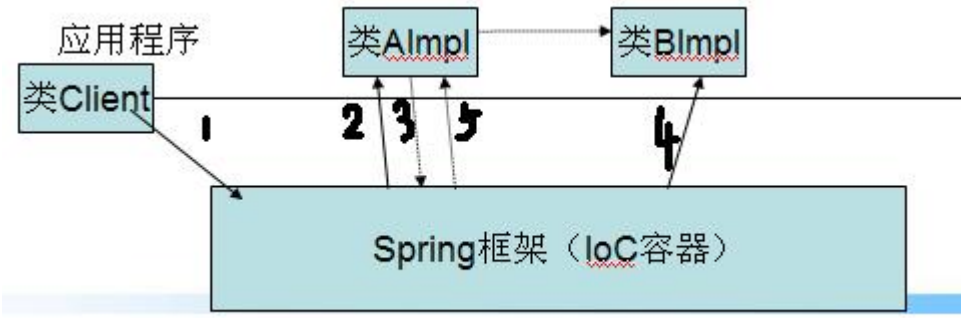
```
<bean id= “a”  class= “AImpl” >
    <property name= “b”  ref= “b” />
</bean>
<bean id= “b”  class= “BImpl” />
```

本质：创建对象和装配对象、管理对象生命周期

被动实例化，被动接受依赖，被动装配

（工厂+反射+xml配置文件）

通用



IoC容器：实现了IoC思想的容器就是IoC容器

2、IoC容器特点

【1】无需主动new对象；而是描述对象应该如何被创建即可

IoC容器帮你创建，即被动实例化；

【2】不需要主动装配对象之间的依赖关系，而是描述需要哪个服务（组件），

IoC容器会帮你装配（即负责将它们关联在一起），被动接受装配；

【3】主动变被动，好莱坞法则：别打电话给我们，我们会打给你；

【4】迪米特法则（最少知识原则）：不知道依赖的具体实现，只知道需要提供某类服务的对象（面向抽象编程），松散耦合，一个对象应当对其他对象有尽可能少的了解,不和陌生人（实现）说话

【5】IoC是一种让服务消费者不直接依赖于服务提供者的组件设计方式，是一种减少类与类之间依赖的设计原则。

3、理解IoC容器问题关键：控制的哪些方面被反转了？

1、谁控制谁？为什么叫反转？ ------ **IoC容器控制，而以前是应用程序控制，所以叫反转**

2、控制什么？ ------ **控制应用程序所需要的资源（对象、文件……）**

3、为什么控制？ ------ **解耦组件之间的关系**

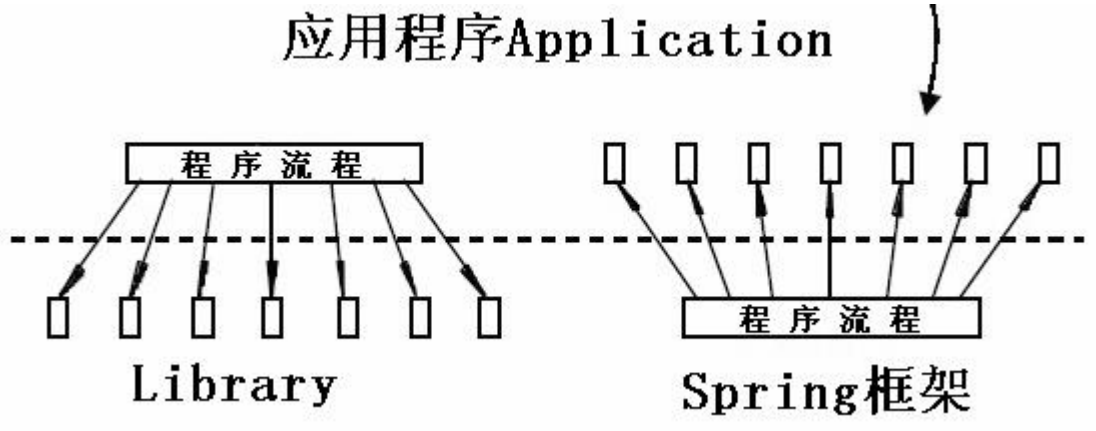
4、控制的哪些方面被反转了？ ------ **程序的控制权发生了反转：从应用程序转移到了IoC容器。**

思考：

1： IoC/DI等同于工厂吗？

2： IoC/DI跟以前的方式有什么不一样？

领会：**主从换位的思想**



4、实现了IoC思想的容器就是轻量级容器吗？

如果仅仅因为使用了控制反转就认为这些轻量级容器与众不同，就好象在说我的轿车与众不同因为它有四个轮子？

容器：提供组件运行环境，管理组件声明周期（不管组件如何创建的以及组件之间关系如何装配的）；

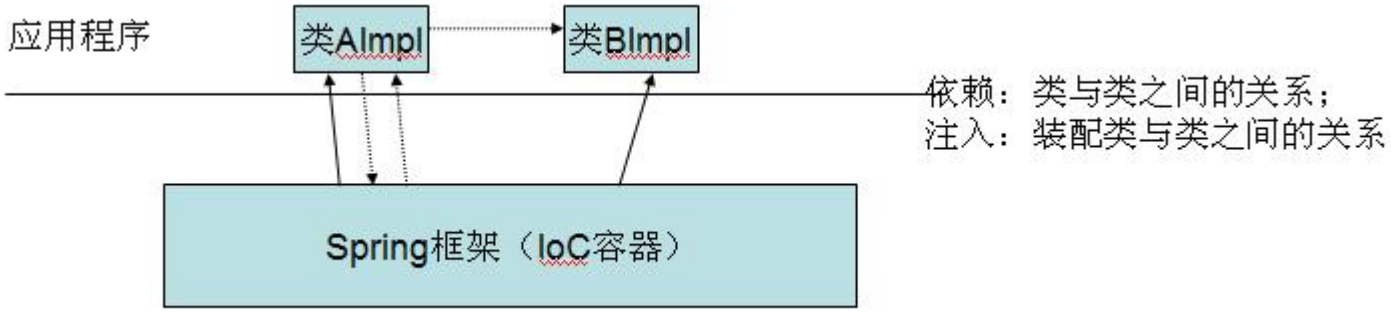
IoC容器不仅仅具有容器的功能，而且还具有一些其他特性---如依赖装配

控制反转概念太广泛，让人迷惑，后来Martin Fowler 提出依赖注入概念
Martin Fowler Inversion of Control Containers and the Dependency Injection pattern
http://martinfowler.com/articles/injection.html

DI

2、什么是DI

DI：依赖注入（Dependency Injection）：用一个单独的对象（装配器）来装配对象之间的依赖关系。



2、理解DI问题关键

谁依赖于谁？ ------- **应用程序依赖于IoC容器**
为什么需要依赖？ ------- **应用程序依赖于IoC容器装配类之间的关系**
依赖什么东西？ ------- **依赖了IoC容器的装配功能**
谁注入于谁？ ------- **IoC容器注入应用程序**
注入什么东西？ ------- **注入应用程序需要的资源（类之间的关系）**

更能描述容器其特点的名字——“依赖注入”（Dependency Injection）
IoC容器应该具有依赖注入功能，因此也可以叫DI容器

3、DI优点

【1】帮你看清组件之间的依赖关系，只需要观察依赖注入的机制（setter/构造器），就可以掌握整个依赖（类与类之间的关系）。

【2】组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

【3】依赖注入的目标并非为软件系统带来更多的功能，而是为了提升组件重用的概率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不用关心具体的资源来自何处、由谁实现。

使用DI限制：组件和装配器（IoC容器）之间不会有依赖关系，因此组件无法从装配器那里获得更多服务，只能获得配置信息中所提供的那些。

4、实现方式

1、构造器注入

2、setter注入

3、接口注入：在接口中定义需要注入的信息，并通过接口完成注入

```
@Autowired
public void prepare(MovieCatalog movieCatalog,
    CustomerPreferenceDao customerPreferenceDao) {
    this.movieCatalog = movieCatalog;
    this.customerPreferenceDao = customerPreferenceDao;
}
```

使用IoC/DI容器开发需要改变的思路

1、应用程序不主动创建对象，但要描述创建它们的方式。

2、在应用程序代码中不直接进行服务的装配，但要配置文件中描述哪一个组件需要哪一项服务。容器负责将这些装配在一起。

其原理是基于OO设计原则的The Hollywood Principle：Don‘t call us, we’ll call you（别找我，我会来找你的）。也就是说，所有的组件都是被动的（Passive），所有的组件初始化和装配都由容器负责。组件处在一个容器当中，由容器负责管理。

IoC容器功能：实例化、初始化组件、装配组件依赖关系、负责组件生命周期管理。

本质：

IoC：控制权的转移，由应用程序转移到框架；

IoC/DI容器：**由**应用程序主动实例化对象**变**被动等待对象（被动实例化）；

DI： 由专门的装配器装配组件之间的关系；

IoC/DI容器：**由**应用程序主动装配对象的依赖**变**应用程序被动接受依赖

关于IoC/DI与DIP之间的关系 详见 http://www.iteye.com/topic/1122310?page=5#2335746

IoC/DI与迪米特法则 详见http://www.iteye.com/topic/1122310?page=5#2335748

1.5 SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结

发表时间: 2012-02-26 关键字: spring, ssh, hibernate, 企业应用

下载地址

一 开发环境

1、动态web工程

2、部分依赖

java代码：

```
hibernate-release-4.1.0.Final.zip
hibernate-validator-4.2.0.Final.jar
spring-framework-3.1.1.RELEASE-with-docs.zip
proxool-0.9.1.jar
log4j 1.2.16
slf4j -1.6.1
mysql-connector-java-5.1.10.jar
hamcrest 1.3.0RC2
ehcache 2.4.3
```

3、为了方便学习，暂没有使用maven构建工程

二 工程主要包括内容

1、springMVC + spring3.1.1 + hibernate4.1.0集成

2、通用DAO层 和 Service层

3、二级缓存 Ehcache

4、REST风格的表现层

5、通用分页（两个版本）

5.1、首页 上一页,下一页 尾页 跳转

5.2、上一页 1 2 3 4 5 下一页

6、数据库连接池采用proxool

7、spring集成测试

8、表现层的 java validator框架验证（采用hibernate-validator-4.2.0实现）

9、视图采用JSP，并进行组件化分离

三 TODO LIST 将本项目做成脚手架方便以后新项目查询

1、Service层进行AOP缓存（缓存使用Memcached实现）

2、单元测试（把常见的桩测试、伪实现、模拟对象演示一遍 区别集成测试）

3、监控功能

后台查询hibernate二级缓存 hit/miss率功能

后台查询当前服务器状态功能（如 线程信息、服务器相关信息）

4、spring RPC功能

5、spring集成 quartz 进行任务调度

6、spring集成 java mail进行邮件发送

7、DAO层将各种常用框架集成进来（方便查询）

8、把工作中经常用的东西 融合进去，作为脚手架，方便以后查询

四 集成重点及常见问题

1、spring-config.xml 配置文件：

1.1、该配置文件只加载除表现层之外的所有bean，因此需要如下配置：

java代码：

```
    <context:component-scan base-package="cn.javass">
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype
    </context:component-scan>
```

通过exclude-filter 把所有 @Controller注解的表现层控制器组件排除

1.2、国际化消息文件配置

java代码：

```
<!-- 国际化的消息资源文件 -->
    <bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBu
        <property name="basenames">
            <list>
                <!-- 在web环境中一定要定位到classpath 否则默认到当前web应用下找  -->
                <value>classpath:messages</value>
            </list>
        </property>
        <property name="defaultEncoding" value="UTF-8"/>
        <property name="cacheSeconds" value="60"/>
    </bean>
```

此处basenames内一定是 classpath:messages ，如果你写出“messages” ，将会到你的web应用的根下找 即你的messages.properties一定在 web应用/messages.propertis。

1.3、hibernate的sessionFactory配置 需要使用 org.springframework.orm.hibernate4.LocalSessionFactoryBean，其他都是类似的，具体看源代码。

1．4、<aop:aspectj-autoproxy expose-proxy="true"/> 实现@AspectJ注解的，默认使用AnnotationAwareAspectJAutoProxyCreator进行AOP代理，它是BeanPostProcessor的子类，在容器启动时Bean初始化开始和结束时调用进行AOP代理的创建，因此只对当容器启动时有效，使用时注意此处。

1.5、声明式容器管理事务

建议使用声明式容器管理事务，而不建议使用注解容器管理事务（虽然简单），但太分布式了，采用声明式容器管理事务一般只对service层进行处理。

java代码：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="merge*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="put*" propagation="REQUIRED" />
        <tx:method name="use*" propagation="REQUIRED"/>
        <!--hibernate4必须配置为开启事务 否则 getCurrentSession()获取不到-->
        <tx:method name="get*" propagation="REQUIRED" read-only="true" />
        <tx:method name="count*" propagation="REQUIRED" read-only="true" />
        <tx:method name="find*" propagation="REQUIRED" read-only="true" />
        <tx:method name="list*" propagation="REQUIRED" read-only="true" />
        <tx:method name="*" read-only="true" />
```

```
        </tx:attributes>
    </tx:advice>
    <aop:config expose-proxy="true">
        <!-- 只对业务逻辑层实施事务 -->
        <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*.*(..))" 
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
    </aop:config>
```

此处一定注意 使用 hibernate4，在不使用OpenSessionInView模式时，在使用getCurrentSession()时会有如下问题：

当有一个方法list 传播行为为Supports，当在另一个方法getPage()（无事务）调用list方法时会抛出org.hibernate.HibernateException: No Session found for current thread 异常。

这是因为getCurrentSession()在没有session的情况下不会自动创建一个，不知道这是不是Spring3.1实现的bug，欢迎大家讨论下。

因此最好的解决方案是使用REQUIRED的传播行为。

二、spring-servlet.xml：

2.1、表现层配置文件，只应加装表现层Bean，否则可能引起问题。

java代码：

```
    <!-- 开启controller注解支持 -->
    <!-- 注：如果base-package=cn.javass 则注解事务不起作用-->
    <context:component-scan base-package="cn.javass.demo.web.controller">
```

```
        <context:include-filter type="annotation" expression="org.springframework.stereotype
    </context:component-scan>
```

此处只应该加载表现层组件，如果此处还加载dao层或service层的bean会将之前容器加载的替换掉，而且此处不会进行AOP织入，所以会造成AOP失效问题（如事务不起作用），再回头看我们的1.4讨论的。

2.2、<mvc:view-controller path="/" view-name="forward:/index"/> 表示当访问主页时自动转发到index控制器。

2.3、静态资源映射

java代码：

```
    <!-- 当在web.xml 中   DispatcherServlet使用     <url-pattern>/</url-pattern> 映射时，能映射
    <mvc:default-servlet-handler/>
    <!-- 静态资源映射 -->
    <mvc:resources mapping="/images/**" location="/WEB-INF/images/" />
    <mvc:resources mapping="/css/**" location="/WEB-INF/css/" />
    <mvc:resources mapping="/js/**" location="/WEB-INF/js/" />
```

以上是配置文件部分，接下来来看具体代码。

三、通用DAO层Hibernate4实现

为了减少各模块实现的代码量，实际工作时都会有通用DAO层实现，以下是部分核心代码：

java代码：

```
public abstract class BaseHibernateDao<M extends java.io.Serializable, PK extends java.io.Se

    protected static final Logger LOGGER = LoggerFactory.getLogger(BaseHibernateDao.class);

    private final Class<M> entityClass;
    private final String HQL_LIST_ALL;
    private final String HQL_COUNT_ALL;
    private final String HQL_OPTIMIZE_PRE_LIST_ALL;
    private final String HQL_OPTIMIZE_NEXT_LIST_ALL;
    private String pkName = null;

    @SuppressWarnings("unchecked")
    public BaseHibernateDao() {
        this.entityClass = (Class<M>) ((ParameterizedType) getClass().getGenericSuperclass()
        Field[] fields = this.entityClass.getDeclaredFields();
        for(Field f : fields) {
            if(f.isAnnotationPresent(Id.class)) {
                this.pkName = f.getName();
            }
        }

        Assert.notNull(pkName);
        //TODO @Entity name not null
        HQL_LIST_ALL = "from " + this.entityClass.getSimpleName() + " order by " + pkName +
        HQL_OPTIMIZE_PRE_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where " +
        HQL_OPTIMIZE_NEXT_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where "
        HQL_COUNT_ALL = " select count(*) from " + this.entityClass.getSimpleName();
    }

    @Autowired
    @Qualifier("sessionFactory")
    private SessionFactory sessionFactory;
```

```
    public Session getSession() {
        //事务必须是开启的，否则获取不到
        return sessionFactory.getCurrentSession();
    }
……
}
```

Spring3.1集成Hibernate4不再需要HibernateDaoSupport和HibernateTemplate了，直接使用原生API即可。

四、通用Service层代码 此处省略，看源代码，有了通用代码后CURD就不用再写了。

java代码：

```
@Service("UserService")
public class UserServiceImpl extends BaseService<UserModel, Integer> implements UserService

    private static final Logger LOGGER = LoggerFactory.getLogger(UserServiceImpl.class);

    private UserDao userDao;

    @Autowired
    @Qualifier("UserDao")
    @Override
    public void setBaseDao(IBaseDao<UserModel, Integer> userDao) {
        this.baseDao = userDao;
        this.userDao = (UserDao) userDao;
    }



    @Override
    public Page<UserModel> query(int pn, int pageSize, UserQueryModel command) {
```

```
        return PageUtil.getPage(userDao.countQuery(command) ,pn, userDao.query(pn, pageSize,
    }
}
```

五、表现层 Controller实现

采用SpringMVC支持的REST风格实现，具体看代码，此处我们使用了java Validator框架 来进行 表现层数据验证

在Model实现上加验证注解

java代码：

```
    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{username.illegal}") //java validator验
    private String username;

    @NotEmpty(message = "{email.illegal}")
    @Email(message = "{email.illegal}") //错误消息会自动到MessageSource中查找
    private String email;

    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{password.illegal}")
    private String password;

    @DateFormat( message="{register.date.error}")//自定义的验证器
    private Date registerDate;
```

在Controller中相应方法的需要验证的参数上加@Valid即可

java代码：

```
    @RequestMapping(value = "/user/add", method = {RequestMethod.POST})
    public String add(Model model, @ModelAttribute("command") @Valid UserModel command, Bind
```

六、Spring集成测试

使用Spring集成测试能很方便的进行Bean的测试，而且使用@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)能自动回滚事务，清理测试前后状态。

java代码：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:spring-config.xml"})
@Transactional
@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)
public class UserServiceTest {

    AtomicInteger counter = new AtomicInteger();

    @Autowired
    private UserService userService;
    ……
}
```

其他部分请直接看源码，欢迎大家讨论。

1.6 »Spring 之AOP AspectJ切入点语法详解（最全了，不需要再去其他地找了）

发表时间: 2012-02-21 关键字: spring, aop

6.5 AspectJ切入点语法详解

6.5.1 Spring AOP支持的AspectJ切入点指示符

切入点指示符用来指示切入点表达式目的，，在Spring AOP中目前只有执行方法这一个连接点，Spring AOP支持的AspectJ切入点指示符如下：

execution：用于匹配方法执行的连接点；

within：用于匹配指定类型内的方法执行；

this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也类型匹配；

target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；

args：用于匹配当前执行的方法传入的参数为指定类型的执行方法；

@within：用于匹配所以持有指定注解类型内的方法；

@target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；

@args：用于匹配当前执行的方法传入的参数持有指定注解的执行；

@annotation：用于匹配当前执行方法持有指定注解的方法；

bean：Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法；

reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持。

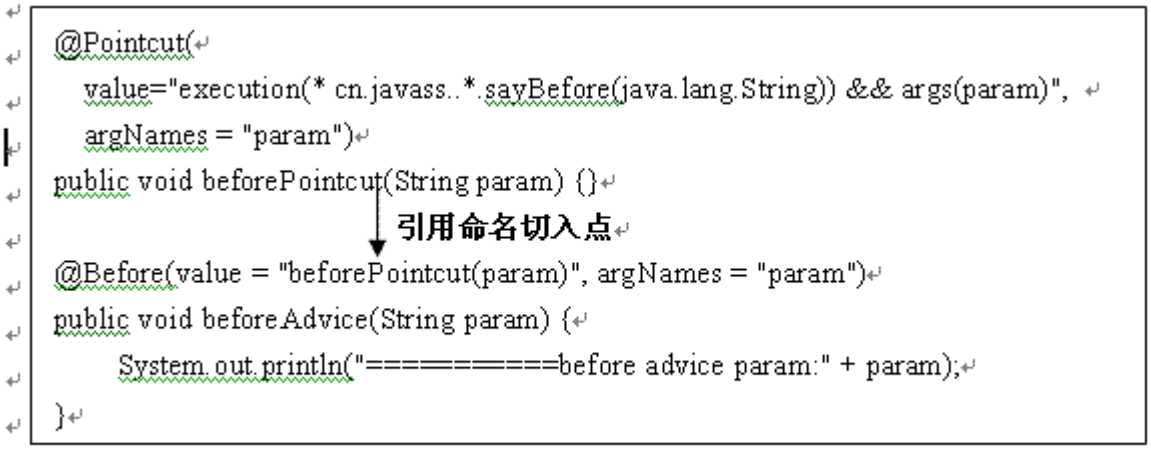
AspectJ切入点支持的切入点指示符还有： call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但Spring AOP目前不支持这些指示符，使用这些指示符将抛出IllegalArgumentException异常。这些指示符Spring AOP可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有@AspectJ支持命名切入点，而Schema风格不支持命名切入点。

如下所示，@AspectJ使用如下方式引用命名切入点：



6.5.2 ；类型匹配语法

首先让我们来了解下AspectJ类型匹配的通配符：

*：匹配任何数量字符；

..：匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

java代码：

查看复制到剪贴板打印

1. java.lang.String 匹配String类型；
2. java.*.String 匹配java包下的任何“一级子包”下的String类型；
3. 如匹配java.lang.String，但不匹配java.lang.ss.String
4. java..* 匹配java包及任何子包下的任何类型;
5. 如匹配java.lang.String、java.lang.annotation.Annotation
6. java.lang.*ing 匹配任何java.lang包下的以ing结尾的类型；
7. java.lang.Number+ 匹配java.lang包下的任何Number的自类型；
8. 如匹配java.lang.Integer，也匹配java.math.BigInteger

接下来再看一下具体的匹配表达式类型吧：

匹配类型：使用如下方式匹配

java代码：

查看复制到剪贴板打印

1. 注解？ 类的全限定名字

- **注解：**可选，类型上持有的注解，如@Deprecated；
- **类的全限定名：**必填，可以是任何类全限定名。

匹配方法执行：使用如下方式匹配：

java代码：

查看复制到剪贴板打印

1. 注解？ 修饰符? 返回值类型 类型声明?方法名(参数列表) 异常列表？

- **注解：**可选，方法上持有的注解，如@Deprecated；
- **修饰符：**可选，如public、protected；
- **返回值类型：**必填，可以是任何类型模式；“*”表示所有类型；
- **类型声明：**可选，可以是任何类型模式；
- **方法名：**必填，可以使用“*”进行模式匹配；
- **参数列表：**“()”表示方法没有任何参数；“(..)”表示匹配接受任意个参数的方法，“(..,java.lang.String)”表示匹配接受java.lang.String类型的参数结束，且其前边可以接受有任意个参数的方法；“(java.lang.String,..)” 表示匹配接受java.lang.String类型的参数开始，且其后边可以接受任意个参数的方法；“(*,java.lang.String)” 表示匹配接受java.lang.String类型的参数结束，且其前边接受有一个任意类型参数的方法；

- **异常列表：**可选，以“throws 异常全限定名列表”声明，异常全限定名列表如有多个以“，”分割，如throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException。

匹配Bean名称：可以使用Bean的id或name进行匹配，并且可使用通配符“*” ；

6.5.3 组合切入点表达式

AspectJ使用 且（&&）、或（||）、非（！）来组合切入点表达式。

在Schema风格下，由于在XML中使用“&&”需要使用转义字符“&”来代替之，所以很不方便，因此Spring ASP 提供了and、or、not来代替&&、||、！。

6.5.3 切入点使用示例

一、execution：使用“execution(方法表达式)”匹配方法执行；

模式	描述
public * *(..)	任何公共方法的执行
* cn.javass..IPointcutService.*()	cn.javass包及所有子包下IPointcutService接口中的任何无参方法
* cn.javass..*.*(..)	cn.javass包及所有子包下任何类的任何方法
* cn.javass..IPointcutService.*(*)	cn.javass包及所有子包下IPointcutService接口的任何只有一个参数方法
* (!cn.javass..IPointcutService+).*(..)	非“cn.javass包及所有子包下IPointcutService接口及子类型”的任何方法
* cn.javass..IPointcutService+.*()	cn.javass包及所有子包下IPointcutService接口及子类型的的任何无参方法

* cn.javass..IPointcut*.test*(java.util.Date)	cn.javass包及所有子包下IPointcut前缀类型的的以test开头的只有一个参数类型为java.util.Date的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的 如定义方法：public void test(Object obj);即使执行时传入java.util.Date，也不会匹配的；
* cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass包及所有子包下IPointcut前缀类型的的任何方法，且抛出IllegalArgumentException和ArrayIndexOutOfBoundsException异常
* (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)	任何实现了cn.javass包及所有子包下IPointcutService接口和java.io.Serializable接口的类型的任何方法
@java.lang.Deprecated * *(..)	任何持有@java.lang.Deprecated注解的方法
@java.lang.Deprecated @cn.javass..Secure * *(..)	任何持有@java.lang.Deprecated和@cn.javass..Secure注解的方法
@(java.lang.Deprecated || cn.javass..Secure) * *(..)	任何持有@java.lang.Deprecated或@ cn.javass..Secure注解的方法
(@cn.javass..Secure *) *(..)	任何返回值类型持有@cn.javass..Secure的方法
* (@cn.javass..Secure *).*(..)	任何定义方法的类型持有@cn.javass..Secure的方法
* *(@cn.javass..Secure (*), @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数都被@Secure标记了， 如public void test(@Secure String str1, @Secure String str1);

* *((@ cn.javass..Secure *))或 * *(@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有@cn.javass..Secure； 如public void test(Model model);且Model类上持有@Secure注解
* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被@cn.javass..Secure标记了；且这两个参数的类型上都持有@ cn.javass..Secure；
* *(java.util.Map<cn.javass..Model, cn.javass..Model> , ..)	任何带有一个java.util.Map参数的方法，且该参数类型是以< cn.javass..Model, cn.javass..Model >为泛型参数；注意只匹配第一个参数为java.util.Map,不包括子类型； 如public void test(HashMap<Model, Model> map, String str);将不匹配，必须使用“* *(java.util.HashMap<cn.javass..Model,cn.javass..Model> , ..)”进行匹配； 而public void test(Map map, int i);也将不匹配，因为泛型参数不匹配
* *(java.util.Collection<@cn.javass..Secure *>)	任何带有一个参数（类型为java.util.Collection）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解； 如public void test(Collection<Model> collection);Model类型上持有@cn.javass..Secure
* *(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型继承与HashMap；

Spring AOP目前测试不能正常工作

* *(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型是HashMap的基类型；如public voi test(Map map)； **Spring AOP目前测试不能正常工作**
* *(*<@cn.javass..Secure *>)	任何带有一个参数的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解； **Spring AOP目前测试不能正常工作**

二、within：使用“within(类型表达式)”匹配指定类型内的方法执行；

模式	**描述**
within(cn.javass..*)	cn.javass包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass包或所有子包下IPointcutService类型及子类型的任何方法
within(@cn.javass..Secure *)	持有cn.javass..Secure注解的任何类型的任何方法 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、this：使用“this(类型全限定名)”匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意this中使用的表达式必须是类型全限定名，不支持通配符；

模式	**描述**

this(cn.javass.spring.chapter6.service.IPointcutService)	当前AOP对象实现了 IPointcutService接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前AOP对象实现了 IIntroductionService接口的任何方法 也可能是引入接口

四、**target：**使用“target(类型全限定名)”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意target中使用的表达式必须是类型全限定名，不支持通配符；

模式	**描述**
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象（非AOP对象）实现了IPointcutService接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象（非AOP对象） 实现了IIntroductionService 接口的任何方法 不可能是引入接口

五、**args：**使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述
args (java.io.Serializable,..)	任何一个以接受“传入参数类型为java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的

六、@within：使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	描述
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

七、@target：使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述
@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、@args：使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	描述

@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure；动态切入点，类似于arg指示符；

九、@annotation：使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

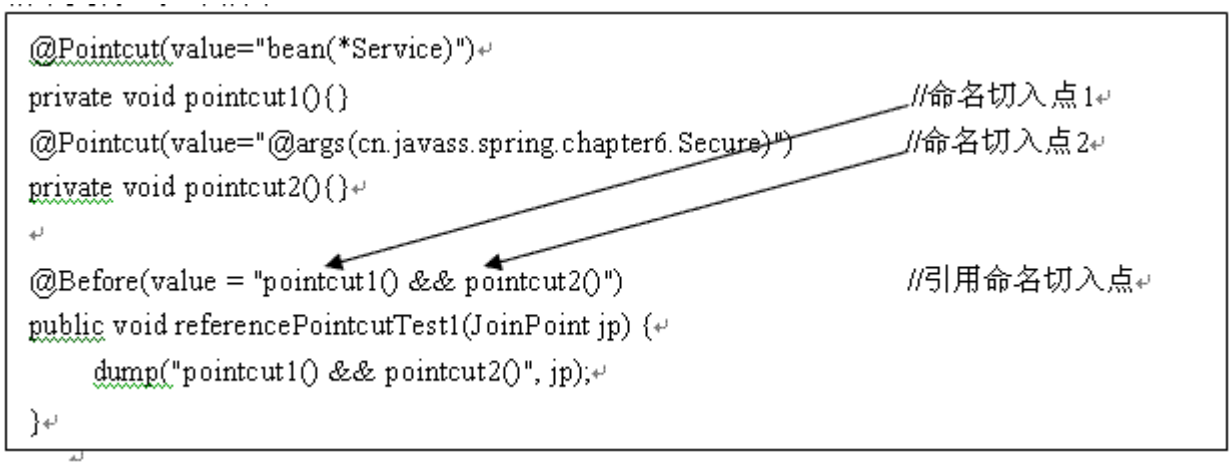
模式	**描述**
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解 cn.javass.spring.chapter6.Secure将被匹配

十、bean：使用“bean(Bean id或名字通配符)”匹配特定名称的Bean对象的执行方法；Spring ASP扩展的，在AspectJ中无相应概念；

模式	**描述**

bean(*Service)	匹配所有以Service命名（id或name）结尾的Bean

十一、reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持，如下所示：



比如我们定义如下切面：

java代码：

查看复制到剪贴板打印

```
1. package cn.javass.spring.chapter6.aop;
2. import org.aspectj.lang.annotation.Aspect;
3. import org.aspectj.lang.annotation.Pointcut;
4. @Aspect
5. public class ReferencePointcutAspect {
6.     @Pointcut(value="execution(* *())")
7.     public void pointcut() {}
8. }
```

可以通过如下方式引用：

java代码：

查看复制到剪贴板打印

```
1. @Before(value = "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.pointcut()")
2. public void referencePointcutTest2(JoinPoint jp) {}
```

除了可以在@AspectJ风格的切面内引用外，也可以在Schema风格的切面定义内引用，引用方式与@AspectJ完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况，但当然还有更复杂的语法等等，如果以上介绍的不能满足您的需要，请参考AspectJ文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在cn.javass.spring.chapter6. PointcutTest文件中，需要时请参考该文件。

6.6 通知参数

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- **使用JoinPoint获取**：Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

java代码：

查看复制到剪贴板打印

```
1. package org.aspectj.lang;
2. import org.aspectj.lang.reflect.SourceLocation;
3. public interface JoinPoint {
4.     String toString();         //连接点所在位置的相关信息
5.     String toShortString();     //连接点所在位置的简短相关信息
6.     String toLongString();     //连接点所在位置的全部相关信息
7.     Object getThis();          //返回AOP代理对象
8.     Object getTarget();        //返回目标对象
9.     Object[] getArgs();        //返回被通知方法参数列表
10.    Signature getSignature();  //返回当前连接点签名
11.    SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
12.    String getKind();          //连接点类型
13.    StaticPart getStaticPart(); //返回连接点静态部分
14. }
```

2) ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

java代码：

查看复制到剪贴板打印

```
1. public interface ProceedingJoinPoint extends JoinPoint {
2.     public Object proceed() throws Throwable;
3.     public Object proceed(Object[] args) throws Throwable;
4. }
```

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

java代码：

查看复制到剪贴板打印

```
1. public interface StaticPart {
2. Signature getSignature();    //返回当前连接点签名
3. String getKind();          //连接点类型
4.     int getId();             //唯一标识
5. String toString();         //连接点所在位置的相关信息
6.     String toShortString();    //连接点所在位置的简短相关信息
7.     String toLongString();    //连接点所在位置的全部相关信息
8. }
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用jp.getArgs()就能获取到被通知方法参数：

java代码：

查看复制到剪贴板打印

```
1. @Before(value="execution(* sayBefore(*))")
2. public void before(JoinPoint jp) {}
3.
4. @Before(value="execution(* sayBefore(*))")
5. public void before(JoinPoint.StaticPart jp) {}
```

- **自动获取：**通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

java代码：

查看复制到剪贴板打印

```
1. @Before(value="execution(* test(*)) && args(param)", argNames="param")
2. public void before1(String param) {
3.     System.out.println("===param:" + param);
4. }
```

切入点表达式execution(* test(*)) && args(param) ：

1）首先execution(* test(*))匹配任何方法名为test，且有一个任何类型的参数；

2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即java.lang.String；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了execution和bean指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2构造器注入】中的参数名注入限制：**在class文件中没生成变量调试信息是获取不到方法参数名字的。**

所以我们可以使用策略来确定参数名：

1. 如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

java代码：

查看复制到剪贴板打印

```
1. @Before(value=" args(param)", argNames="param") //明确指定了
2. public void before1(String param) {
3.     System.out.println("===param:" + param);
4. }
```

1. 如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

java代码：

查看复制到剪贴板打印

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

1. 如果“**class文件中含有变量调试信息**”将使用这些方法签名中的参数名来确定参数名；

java代码：

查看复制到剪贴板打印

```
@Before(value=" args(param)") //不需要argNames了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

1. 如果没有“**class文件中含有变量调试信息**”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

java代码：

查看复制到剪贴板打印

```
@Before(value=" args(param)")
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

1. 以上策略失败将抛出IllegalArgumentException。

接下来让我们示例一下组合情况吧：

java代码：

查看复制到剪贴板打印

```
1. @Before(args(param) && target(bean) && @annotation(secure)",
2.         argNames="jp,param,bean,secure")
3. public void before5(JoinPoint jp, String param,
4.  IPointcutService pointcutService, Secure secure) {
5. ……
6. }
```

该示例的执行步骤如图6-5所示。

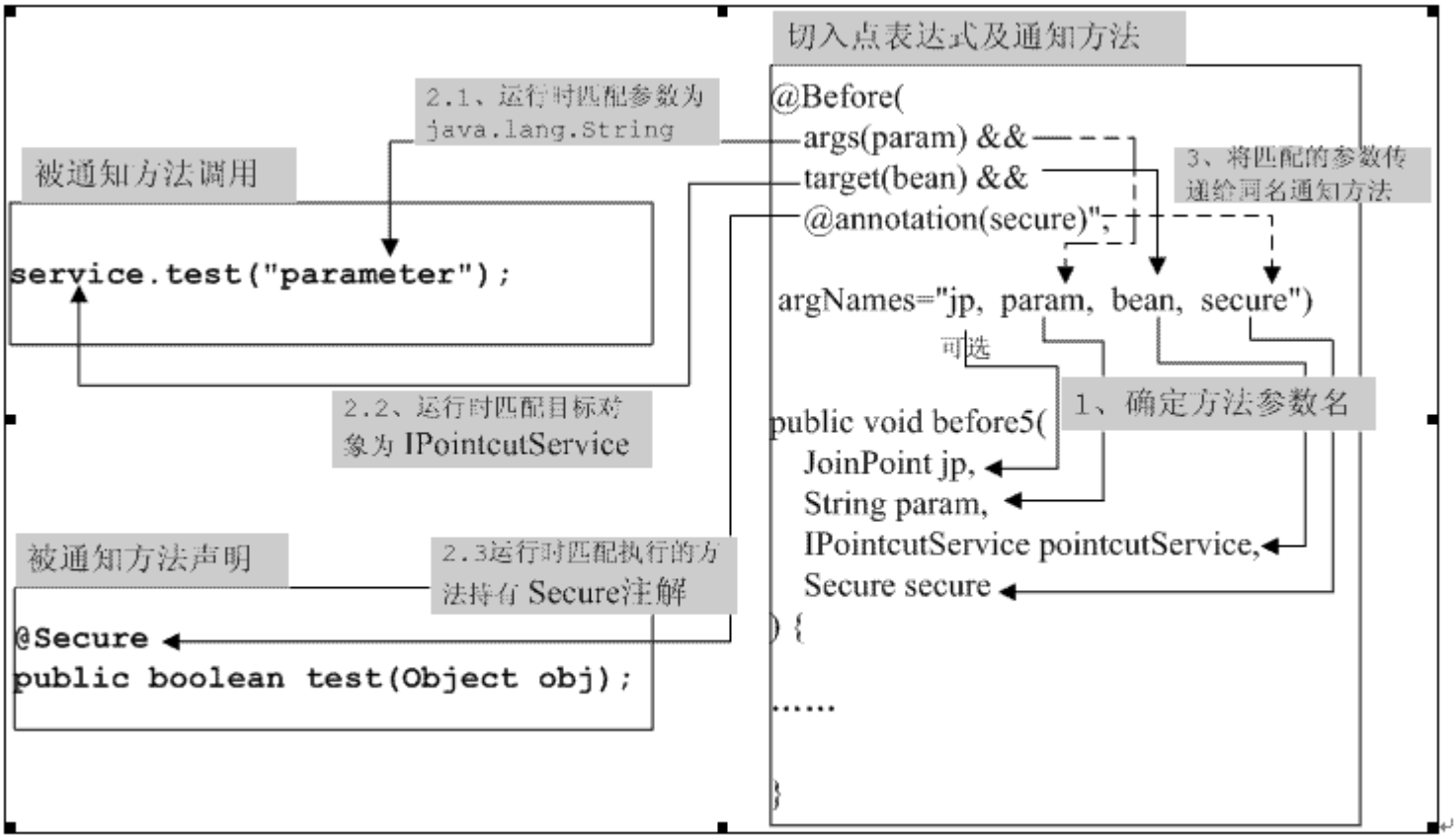


图 6-5 参数自动获取流程

图6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

java代码：

查看复制到剪贴板打印

```
1. @Pointcut(value="args(param)", argNames="param")
2. private void pointcut1(String param){}
3. @Pointcut(value="@annotation(secure)", argNames="secure")
4. private void pointcut2(Secure secure){}
5.
6. @Before(value = "pointcut1(param) && pointcut2(secure)",
7.  argNames="param, secure")
```

```
8. public void before6(JoinPoint jp, String param, Secure secure) {
9. ……
10. }
```

自此给通知传递参数已经介绍完了，示例代码在cn.javass.spring.chapter6.ParameterTest文件中。

在Spring配置文件中，所以AOP相关定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，配置顺序不可变。

- <aop:pointcut>：用来定义切入点，该切入点可以重用；
- <aop:advisor>：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

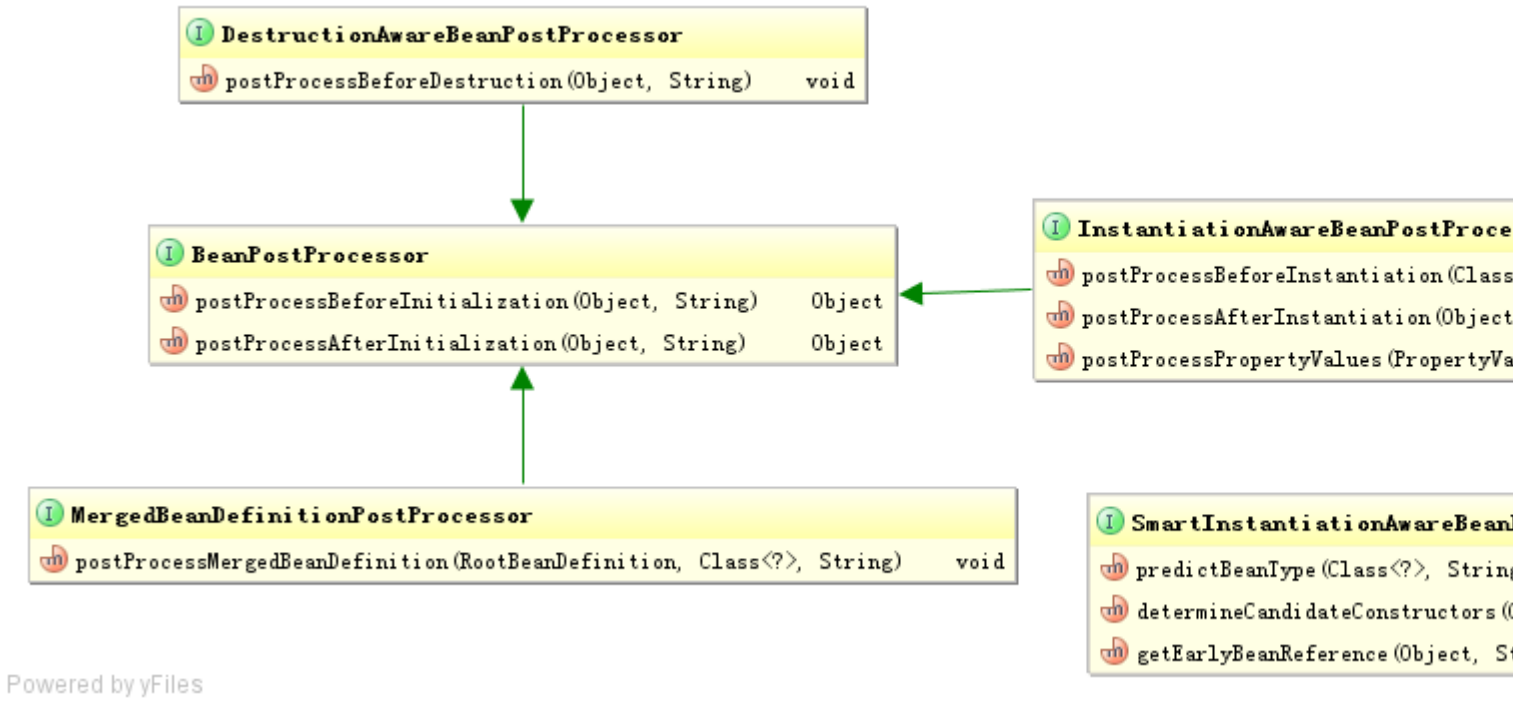
<aop:config>	AOP定义开始（有序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:advisor/>	Advisor定义（零个或多个）
<aop:aspect>	切面定义开始（零个或多个，无序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:before"/>	前置通知（零个或多个）
<aop:after-returning/>	后置返回通知（零个或多个）
<aop:after-throwing/>	后置异常通知（零个或多个）
<aop:after/>	后置最终通知（零个或多个）
<aop:around/>	环绕通知（零个或多个）
<aop:declare-parents/>	引入定义（零个或多个）
</aop:aspect>	切面定义开始（零个或多个）
</aop:config>	AOP定义结束

1.7 Spring开闭原则的表现-BeanPostProcessor扩展点-2

发表时间: 2012-04-20 关键字: spring, aop, bean, java, jpa

上接Spring提供的BeanPostProcessor的扩展点-1继续分析。

四、BeanPostProcessor接口及回调方法图



从图中我们可以看出一共五个接口，共十个回调方法，即十个扩展点，但我们之前的文章只分析了其中八个，另外两个稍候也会解析一下是干什么的。

==========

==========

五、五个接口十个扩展点解析

1、InstantiationAwareBeanPostProcessor：实例化Bean后置处理器（继承BeanPostProcessor）

postProcessBeforeInstantiation ：在实例化目标对象之前执行，可以自定义实例化逻辑，如返回一个代理对象等，（3.1处执行；如果此处返回的Bean不为null将中断后续Spring创建Bean的流程，且只执行postProcessAfterInitialization回调方法，如当AbstractAutoProxyCreator的实现者注册了TargetSourceCreator（创建自定义的TargetSource）将改变执行流程，不注册TargetSourceCreator我们默认使用的是SingletonTargetSource（即AOP代理直接保证目标对象），此处我们还可以使用如ThreadLocalTargetSource（线程绑定的Bean）、CommonsPoolTargetSource（实例池的Bean）等等，大家可以去spring官方文档了解TargetSource详情；

postProcessAfterInitialization ： Bean实例化完毕后执行的后处理操作，所有初始化逻辑、装配逻辑之前执行，如果返回false将阻止其他的InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation的执行，（3.2和（9.1处执行；在此处可以执行一些初始化逻辑或依赖装配逻辑；

postProcessPropertyValues ：完成其他定制的一些依赖注入和依赖检查等，如AutowiredAnnotationBeanPostProcessor执行@Autowired注解注入，CommonAnnotationBeanPostProcessor执行@Resource等注解的注入，PersistenceAnnotationBeanPostProcessor执行@ PersistenceContext等JPA注解的注入，RequiredAnnotationBeanPostProcessor执行@ Required注解的检查等等，（9.3处执行；

2、MergedBeanDefinitionPostProcessor：合并Bean定义后置处理器 （继承BeanPostProcessor）

postProcessMergedBeanDefinition：执行Bean定义的合并，在（7.1处执行，且在实例化完Bean之后执行；

3、SmartInstantiationAwareBeanPostProcessor：智能实例化Bean后置处理器（继承InstantiationAwareBeanPostProcessor）

predictBeanType：预测Bean的类型，返回第一个预测成功的Class类型，如果不能预测返回null；当你调用BeanFactory.getType(name)时当通过Bean定义无法得到Bean类型信息时就调用该回调方法来决定类型信息；BeanFactory.isTypeMatch(name, targetType)用于检测给定名字的Bean是否匹配目标类型（如在依赖注入时需要使用）；

determineCandidateConstructors：检测Bean的构造器，可以检测出多个候选构造器，再有相应的策略决定使用哪一个，如AutowiredAnnotationBeanPostProcessor实现将自动扫描通过@Autowired/@Value注解的

构造器从而可以完成构造器注入，请参考【第十二章】零配置 之12.2 注解实现Bean依赖注入 ——跟我学spring3 ，（6.2.2.1处执行；

getEarlyBeanReference：当正在创建A时，A依赖B，此时通过（8将A作为ObjectFactory放入单例工厂中进行early expose，此处B需要引用A，但A正在创建，从单例工厂拿到ObjectFactory（其通过getEarlyBeanReference获取及早暴露Bean），从而允许循环依赖，此时AspectJAwareAdvisorAutoProxyCreator（完成xml风格的AOP配置(<aop:config>)将目标对象（A）包装到AOP代理对象）或AnnotationAwareAspectJAutoProxyCreator（完成@Aspectj注解风格（<aop:aspectj-autoproxy> @Aspect）将目标对象（A）包装到AOP代理对象），其返回值将替代原始的Bean对象，即此时通过early reference能得到正确的代理对象，（8.1处实施；如果此处执行了，（10.3.3处的AspectJAwareAdvisorAutoProxyCreator或AnnotationAwareAspectJAutoProxyCreator的postProcessAfterInitialization将不执行，即这两个回调方法是二选一的；

4、BeanPostProcessor：Bean后置处理器

postProcessBeforeInitialization：实例化、依赖注入完毕，在调用显示的初始化之前完成一些定制的初始化任务，如BeanValidationPostProcessor完成JSR-303 @Valid注解Bean验证，InitDestroyAnnotationBeanPostProcessor完成@PostConstruct注解的初始化方法调用，ApplicationContextAwareProcessor完成一些Aware接口的注入（如EnvironmentAware、ResourceLoaderAware、ApplicationContextAware），其返回值将替代原始的Bean对象；（10.2处执行；

postProcessAfterInitialization：实例化、依赖注入、初始化完毕时执行，如AspectJAwareAdvisorAutoProxyCreator（完成xml风格的AOP配置(<aop:config>)的目标对象包装到AOP代理对象）、AnnotationAwareAspectJAutoProxyCreator（完成@Aspectj注解风格（<aop:aspectj-autoproxy> @Aspect）的AOP配置的目标对象包装到AOP代理对象），其返回值将替代原始的Bean对象；（10.3.3处执行；此处需要参考getEarlyBeanReference；

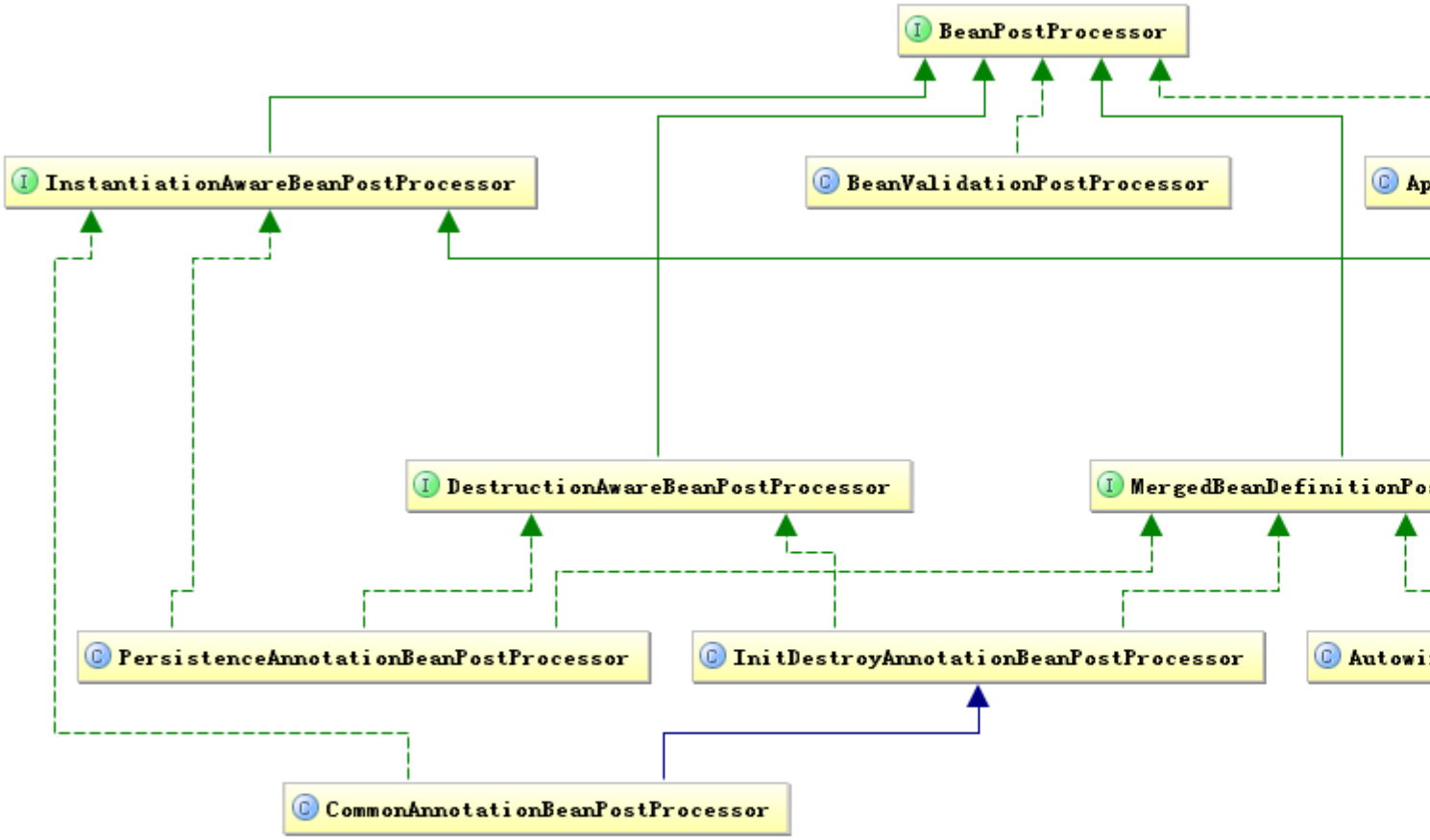
5、DestructionAwareBeanPostProcessor：销毁Bean后置处理器（继承BeanPostProcessor）

postProcessBeforeDestruction：销毁后处理回调方法，该回调只能应用到单例Bean，如InitDestroyAnnotationBeanPostProcessor完成@PreDestroy注解的销毁方法调用；（12.1.1处执行。

==========

==========

六、内置的一些BeanPostProcessor



部分内置的BeanPostProcessor

此图只有内置的一部分。

1、ApplicationContextAwareProcessor

容器启动时会自动注册。注入那些实现ApplicationContextAware、MessageSourceAware、ResourceLoaderAware、EnvironmentAware、

EmbeddedValueResolverAware、ApplicationEventPublisherAware标识接口的Bean需要的相应实例，在postProcessBeforeInitialization回调方法中进行实施，即（10.2处实施。

2、CommonAnnotationBeanPostProcessor

CommonAnnotationBeanPostProcessor继承InitDestroyAnnotationBeanPostProcessor，当在配置文件有<context:annotation-config>或<context:component-scan>会自动注册。

提供对JSR-250规范注解的支持@javax.annotation.Resource、@javax.annotation.PostConstruct和@javax.annotation.PreDestroy等的支持。

2.1、通过@Resource注解进行依赖注入：

postProcessPropertyValues：通过此回调进行@Resource注解的依赖注入；（9.3处实施；

2.2、用于执行@PostConstruct 和@PreDestroy 注解的初始化和销毁方法的扩展点：

postProcessBeforeInitialization()将会调用bean的@PostConstruct方法；（10.2处实施；

postProcessBeforeDestruction()将会调用单例 Bean的@PreDestroy方法（此回调方法会在容器销毁时调用），（12.1.1处实施。

详见【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3，JSR-250注解部分。

3、AutowiredAnnotationBeanPostProcessor

当在配置文件有<context:annotation-config>或<context:component-scan>会自动注册。

提供对JSR-330规范注解的支持和Spring自带注解的支持。

3.1、Spring自带注解的依赖注入支持，@Autowired和@Value：

determineCandidateConstructors ：决定候选构造器；详见【12.2中的构造器注入】；（6.2.2.1处实施；

postProcessPropertyValues ：进行依赖注入；详见【12.2中的字段注入和方法参数注入】；（9.3处实施；

3.2、对JSR-330规范注解的依赖注入支持，@Inject：

同2.1类似只是查找使用的注解不一样；

详见【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3，Spring自带依赖注入注解和JSR-330注解部分。

4、RequiredAnnotationBeanPostProcessor

当在配置文件有<context:annotation-config>或<context:component-scan>会自动注册。

4.1、提供对@ Required注解的方法进行依赖检查支持：

postProcessPropertyValues：如果检测到没有进行依赖注入时抛出BeanInitializationException异常；（9.3处实施；

详见【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3，**@Required：依赖检查。**

5、PersistenceAnnotationBeanPostProcessor

当在配置文件有<context:annotation-config>或<context:component-scan>会自动注册。

5.1、通过对JPA @ javax.persistence.PersistenceUnit和@ javax.persistence.PersistenceContext注解进行依赖注入的支持；

postProcessPropertyValues ： 根据@PersistenceUnit/@PersistenceContext进行EntityManagerFactory和EntityManager的支持；

6、AbstractAutoProxyCreator

AspectJAwareAdvisorAutoProxyCreator和AnnotationAwareAspectJAutoProxyCreator都是继承AbstractAutoProxyCreator，AspectJAwareAdvisorAutoProxyCreator提供对（<aop:config>）声明式AOP的支持，AnnotationAwareAspectJAutoProxyCreator提供对（<aop:aspectj-autoproxy>）注解式（@AspectJ）AOP的支持，因此只需要分析AbstractAutoProxyCreator即可。

当使用<aop:config>配置时自动注册AspectJAwareAdvisorAutoProxyCreator，而使用<aop:aspectj-autoproxy>时会自动注册AnnotationAwareAspectJAutoProxyCreator。

6.1、predictBeanType：预测Bean的类型，如果目标对象被AOP代理对象包装，此处将返回AOP代理对象的类型；

```
public Class<?> predictBeanType(Class<?> beanClass, String beanName) {
                Object cacheKey = getCacheKey(beanClass, beanName);
                return this.proxyTypes.get(cacheKey); //获取代理对象类型，可能返回null
}
```

6.2、postProcessBeforeInstantiation：

```
public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansE
        //1、得到一个缓存的唯一key（根据beanClass和beanName生成唯一key）
        Object cacheKey = getCacheKey(beanClass, beanName);
        //2、如果当前targetSourcedBeans（通过自定义TargetSourceCreator创建的TargetSource）不包含ca
        if (!this.targetSourcedBeans.contains(cacheKey)) {
                //2.1、advisedBeans（已经被增强的Bean，即AOP代理对象）中包含当前cacheKey或nonAdvise
                if (this.advisedBeans.contains(cacheKey) || this.nonAdvisedBeans.contains(cache
                        return null;
                }
                //2.2、如果是基础设施类（如Advisor、Advice、AopInfrastructureBean的实现）不进行处理
                //2.2、shouldSkip 默认false，可以生成子类覆盖，如AspectJAwareAdvisorAutoProxyCrea
                if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                        this.nonAdvisedBeans.add(cacheKey);//在不能增强的Bean列表缓存当前cacheKey
```

```
                        return null;
                }
        }

        //3、开始创建AOP代理对象
        //3.1、配置自定义的TargetSourceCreator进行TargetSource创建
        TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
        if (targetSource != null) {
                //3.2、如果targetSource不为null 添加到targetSourcedBeans缓存，并创建AOP代理对象
                this.targetSourcedBeans.add(beanName);
                // specificInterceptors即增强（包括前置增强、后置增强等等）
                Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanNam
                //3.3、创建代理对象
                Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSou
                //3.4、将代理类型放入proxyTypes从而允许后续的predictBeanType()调用获取
                this.proxyTypes.put(cacheKey, proxy.getClass());
                return proxy;
        }
        return null;
}
```

从如上代码可以看出，当我们配置TargetSourceCreator进行自定义TargetSource创建时，会创建代理对象并中断默认Spring创建流程。

6.3、getEarlyBeanReference

```
//获取early Bean引用（只有单例Bean才能回调该方法）
public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        //1、将cacheKey添加到earlyProxyReferences缓存，从而避免多次重复创建
        this.earlyProxyReferences.add(cacheKey);
        //2、包装目标对象到AOP代理对象（如果需要）
        return wrapIfNecessary(bean, beanName, cacheKey);
}
```

6.4、postProcessAfterInitialization

```
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansExceptic
        if (bean != null) {
                Object cacheKey = getCacheKey(bean.getClass(), beanName);
                //1、如果之前调用过getEarlyBeanReference获取包装目标对象到AOP代理对象（如果需要），则
                if (!this.earlyProxyReferences.contains(cacheKey)) {
                        //2、包装目标对象到AOP代理对象（如果需要）
                        return wrapIfNecessary(bean, beanName, cacheKey);
                }
        }
        return bean;
}
```

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
        if (this.targetSourcedBeans.contains(beanName)) {//通过TargetSourceCreator进行自定义Targ
                return bean;
        }
        if (this.nonAdvisedBeans.contains(cacheKey)) {//不应该被增强对象不需要包装
                return bean;
        }
        if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {/
                this.nonAdvisedBeans.add(cacheKey);
                return bean;
        }

        // 如果有增强就执行包装目标对象到代理对象
        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName,
        if (specificInterceptors != DO_NOT_PROXY) {
                this.advisedBeans.add(cacheKey);//将cacheKey添加到已经被增强列表，防止多次增强
                Object proxy = createProxy(bean.getClass(), beanName, specificInterceptors, new
                this.proxyTypes.put(cacheKey, proxy.getClass());//缓存代理类型
                return proxy;
        }
        this.nonAdvisedBeans.add(cacheKey);
```

```
        return bean;
}
```

从如上流程可以看出 getEarlyBeanReference和postProcessAfterInitialization是二者选一的，而且单例Bean目标对象只能被增强一次，而原型Bean目标对象可能被包装多次。

7、BeanValidationPostProcessor

默认不自动注册，Spring3.0开始支持。

提供对JSR-303验证规范支持。

根据afterInitialization是false/true决定调用postProcessBeforeInitialization或postProcessAfterInitialization来通过JSR-303规范验证Bean，默认false。

8、MethodValidationPostProcessor

Spring3.1开始支持，且只支持Hibernate Validator 4.2及更高版本，从Spring 3.2起可能将采取自动检测Bean Validation 1.1兼容的提供商且自动注册（Bean Validation 1.1 (JSR-349)正处于草案阶段，它将提供方法级别的验证，提供对方法级别的验证），目前默认不自动注册。

Bean Validation 1.1草案请参考http://jcp.org/en/jsr/detail?id=349 http://beanvalidation.org/。

提供对方法参数/方法返回值的进行验证（即前置条件/后置条件的支持），通过JSR-303注解验证，使用方式如：

public @NotNull Object myValidMethod(@NotNull String arg1, @Max(10) int arg2)

默认只对@org.springframework.validation.annotation.Validated注解的Bean进行验证，我们可以修改validatedAnnotationType为其他注解类型来支持其他注解验证。而且目前只支持Hibernate Validator实现，在未来版本可能支持其他实现。

有了这东西之后我们就不需要在进行如Assert.assertNotNull（ ）这种前置条件/后置条件的判断了。

9、ScheduledAnnotationBeanPostProcessor

当配置文件中有<task:annotation-driven>自动注册或@EnableScheduling自动注册。

提供对注解@Scheduled任务调度的支持。

postProcessAfterInitialization：通过查找Bean对象类上的@Scheduled注解来创建ScheduledMethodRunnable对象并注册任务调度方法（仅返回值为void且方法是无形式参数的才可以）。

可参考Spring官方文档的任务调度章节学习@Scheduled注解任务调度。

10、AsyncAnnotationBeanPostProcessor

当配置文件中有<task:annotation-driven>自动注册或@EnableAsync自动注册。

提供对@ Async和EJB3.1的@javax.ejb.Asynchronous注解的异步调用支持。

postProcessAfterInitialization：通过ProxyFactory创建目标对象的代理对象，默认使用AsyncAnnotationAdvisor（内部使用AsyncExecutionInterceptor 通过AsyncTaskExecutor（继承TaskExecutor）通过submit提交异步任务）。

可参考Spring官方文档的异步调用章节学习@Async注解异步调用。

11、ServletContextAwareProcessor

在使用Web容器时自动注册。

类似于ApplicationContextAwareProcessor，当你的Bean 实现了ServletContextAware/ ServletConfigAware会自动调用回调方法注入ServletContext/ ServletConfig。

==========

==========

七、BeanPostProcessor如何注册

1、如ApplicationContextAwareProcessor会在ApplicationContext容器启动时自动注册，而CommonAnnotationBeanPostProcessor和AutowiredAnnotationBeanPostProcessor会在当你使用<context:annotation-config>或<context:component-scan>配置时自动注册。

2、只要将BeanPostProcessor注册到容器中，Spring会在启动时自动获取并注册。

==========

==========

八、BeanPostProcessor的执行顺序

1、如果使用BeanFactory实现，非ApplicationContext实现，BeanPostProcessor执行顺序就是添加顺序。

2、如果使用的是AbstractApplicationContext（实现了ApplicationContext）的实现，则通过如下规则指定顺序。

2.1、PriorityOrdered（继承了Ordered），实现了该接口的BeanPostProcessor会在第一个顺序注册，标识高优先级顺序，即比实现Ordered的具有更高的优先级；

2.2、Ordered，实现了该接口的BeanPostProcessor会第二个顺序注册；

int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;//最高优先级

int LOWEST_PRECEDENCE = Integer.MAX_VALUE;//最低优先级

即数字越小优先级越高，数字越大优先级越低，如0（高优先级）——1000（低优先级）

2.3、无序的，没有实现Ordered/ PriorityOrdered的会在第三个顺序注册；

2.4、内部Bean后处理器，实现了MergedBeanDefinitionPostProcessor接口的是内部Bean PostProcessor，将在最后且无序注册。

3、接下来我们看看内置的BeanPostProcessor执行顺序

//1注册实现了PriorityOrdered接口的BeanPostProcessor

//2注册实现了Ordered接口的BeanPostProcessor

AbstractAutoProxyCreator 实现了Ordered，order = Ordered.LOWEST_PRECEDENCE

MethodValidationPostProcessor 实现了Ordered，LOWEST_PRECEDENCE

ScheduledAnnotationBeanPostProcessor 实现了Ordered，LOWEST_PRECEDENCE

AsyncAnnotationBeanPostProcessor 实现了Ordered，order = Ordered.LOWEST_PRECEDENCE

//3注册无实现任何接口的BeanPostProcessor

BeanValidationPostProcessor 无序

ApplicationContextAwareProcessor 无序

ServletContextAwareProcessor 无序

//3 注册实现了MergedBeanDefinitionPostProcessor接口的BeanPostProcessor，且按照实现了Ordered的顺序进行注册，没有实现Ordered的默认为Ordered.LOWEST_PRECEDENCE。

PersistenceAnnotationBeanPostProcessor 实现了PriorityOrdered，Ordered.LOWEST_PRECEDENCE - 4

AutowiredAnnotationBeanPostProcessor 实现了PriorityOrdered，order = Ordered.LOWEST_PRECEDENCE - 2

RequiredAnnotationBeanPostProcessor 实现了PriorityOrdered，order = Ordered.LOWEST_PRECEDENCE - 1

CommonAnnotationBeanPostProcessor 实现了PriorityOrdered，Ordered.LOWEST_PRECEDENCE

从上到下顺序执行，如果order相同则我们应该认为同序（谁先执行不确定，其执行顺序根据注册顺序决定）。

==========

==========

九、完成Spring事务处理时自我调用的解决方案及一些实现方式的分析分析

场景请先参考请参考Spring事务处理时自我调用的解决方案及一些实现方式的风险中的3.3、通过BeanPostProcessor 在目标对象中注入代理对象。

分析：

问题出现在5和9处：

5、使用步骤1处注册的SingletonFactory（ObjectFactory.getObject() 使用AnnotationAwareAspectJAutoProxyCreator的getEarlyBeanReference获取循环引用Bean），因此此处将返回A目标对象的代理对象；

9、此处调用AnnotationAwareAspectJAutoProxyCreator的postProcessAfterInitialization，但发现之前调用过AnnotationAwareAspectJAutoProxyCreator的getEarlyBeanReference获取代理对象，此处不再创建代理对象，而是直接返回目标对象，因此使用InjectBeanSelfProcessor不能注入代理对象；但此时的Spring容器中的A已经是代理对象了，因此我使用了从上下文重新获取A代理对象的方式注入（context.getBean(beanName)）。

此处的getEarlyBeanReference和postProcessAfterInitialization为什么是二者选一的请参考之前介绍的AbstractAutoProxyCreator。

到此问题我们分析完毕，实际项目中的循环依赖应该尽量避免，这违反了“无环依赖原则”。

下一篇我将介绍一些内置BeanPostProcessor的使用和自定义一些自己的BeanPostProcessor来更好的理解这些扩展点。

1.8 Spring3.1 对Bean Validation规范的新支持(方法级别验证)

发表时间: 2012-04-23 关键字: spring, beanvalidation

上接Spring提供的BeanPostProcessor的扩展点-1继续学习。

一、 Bean Validation框架简介

写道

> Bean Validation standardizes constraint definition, declaration and validation for the Java platform.

大体意思是：Bean Validation 标准化了Java平台的约束定义、描述、和验证。

详细了解请参考：http://beanvalidation.org/

Bean Validation现在一个有两个规范：

1、Bean Validation 1.0（JSR-303）

写道

> This JSR will define a meta-data model and API for JavaBeanTM validation based on annotations, with overrides and extended meta-data through the use of XML validation descriptors.

定义了基于注解方式的JavaBean验证元数据模型和API，也可以通过XML进行元数据定义，但注解将覆盖XML的元数据定义。

详细了解请参考：http://jcp.org/en/jsr/detail?id=303

JSR-303主要是对JavaBean进行验证，如方法级别（方法参数/返回值）、依赖注入等的验证是没有指定的。因此又有了JSR-349规范的产生。

2、Bean Validation 1.1（JSR-349）

写道

> Bean Validation standardizes constraint definition, declaration and validation for the Java platform.

Bean Validation 标准化了Java平台的约束定义、描述、和验证。

此规范目前处于草案状态，详细了解请参考：http://jcp.org/en/jsr/detail?id=349.

该草案现在主要内容：

方法级别验证支持（验证方法参数和和返回值）；

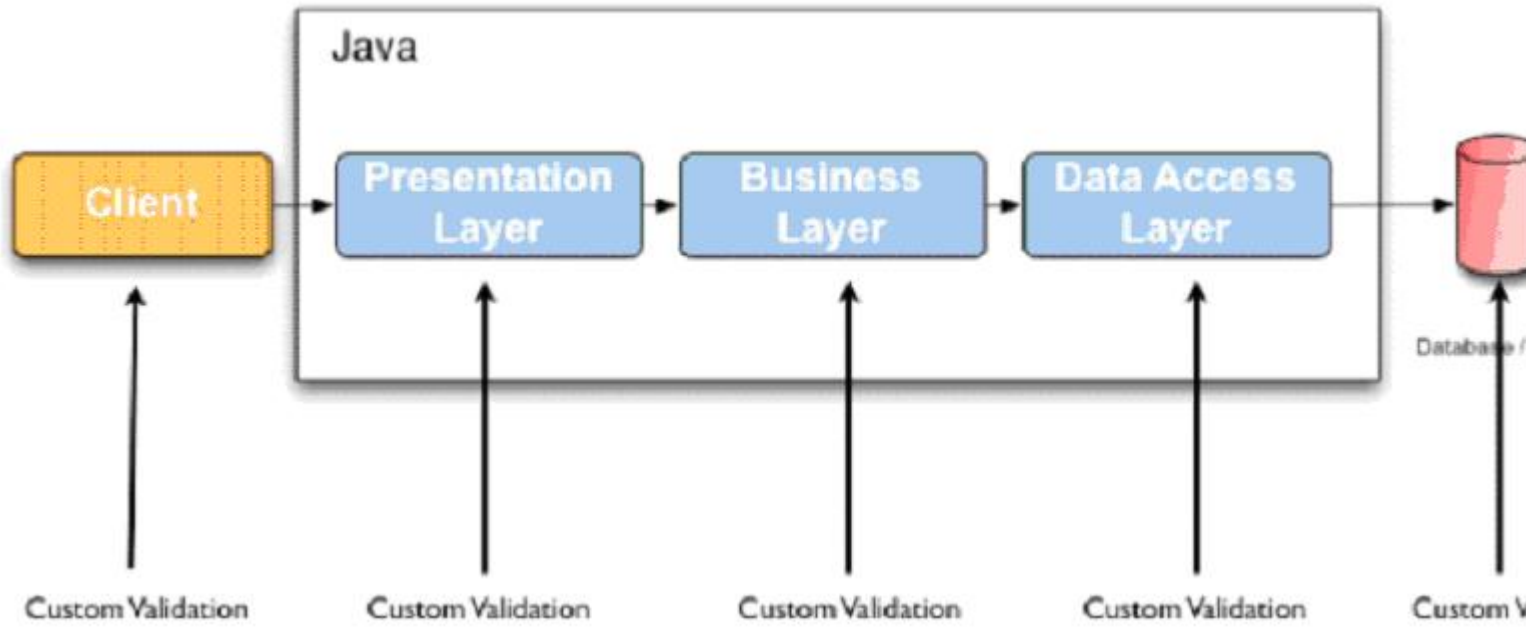
依赖注入验证的支持。

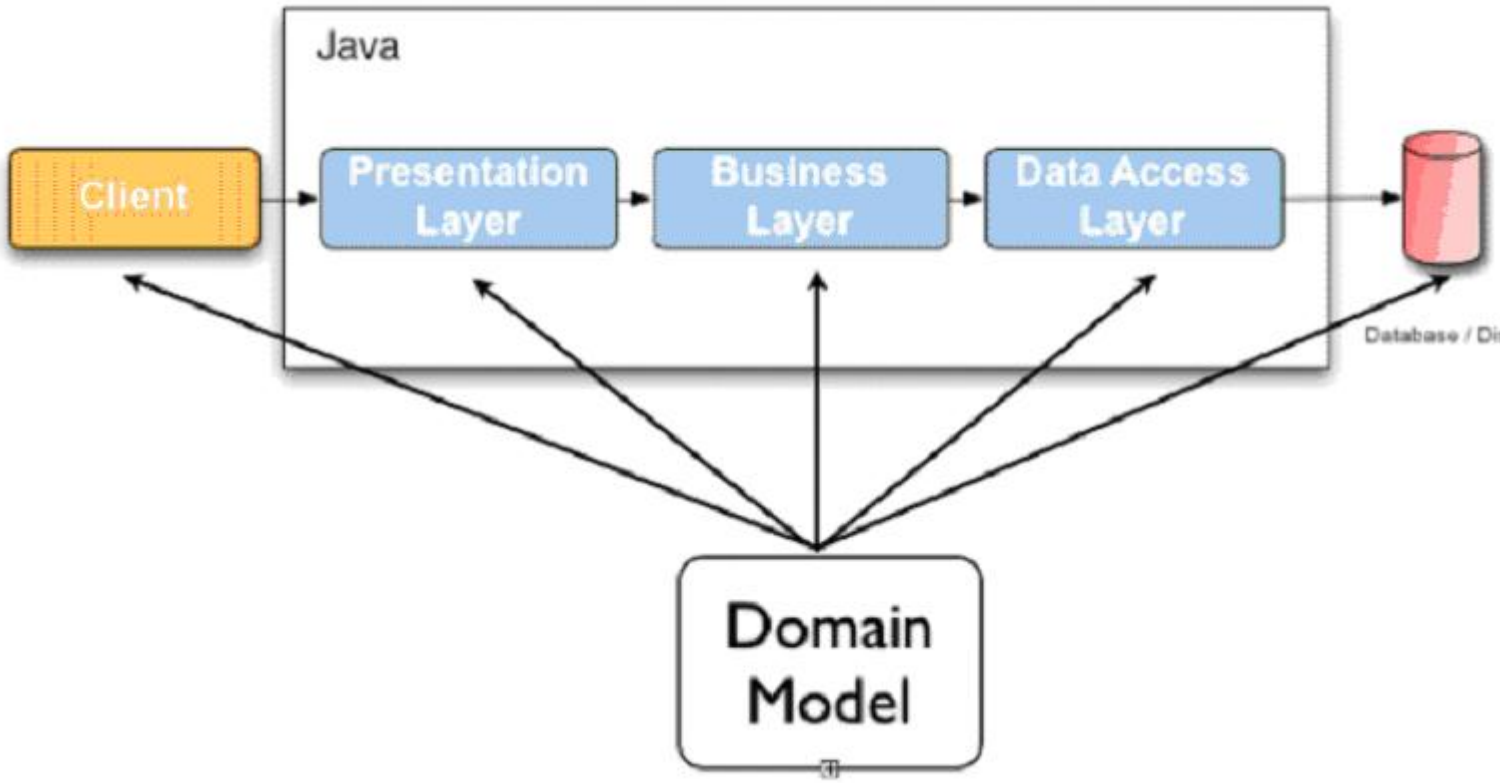
对Bean Validation的详细介绍可参考Bean Validation官网查看http://beanvalidation.org/。

Spring3.1目前已经完全支持依赖注入验证和方法级别验证的支持，只是不是原生的（规范还是草案）。

Bean Validation 1.0的参考实现有Hibernate Validator（下载地址：http://www.hibernate.org/subprojects/validator.html）；1.1还处于草案状态。

二、 Bean Validation在开发中的位置





上图摘自hibernate validator 参考文档，从图中可以看出，我们可以在任何位置实施验证。

1、表现层验证：SpringMVC提供对JSR-303的表现层验证；

2、业务逻辑层验证：Spring3.1提供对业务逻辑层的方法验证（当然方法验证可以出现在其他层，但笔者觉得方法验证应该验证业务逻辑）；

3、DAO层验证：Hibernate提供DAO层的模型数据的验证（可参考hibernate validator参考文档的7.3. ORM集成）。

4、数据库端的验证：通过数据库约束来进行；

5、客户端验证支持：JSR-303也提供编程式验证支持。

对于DAO层和客户端验证支持不在我们示例范围，忽略，感兴趣的同学可以参考《hibernate validator reference》（有中文）。

在测试支持大家需要准备好如下jar包：

validation-api-1.0.0.GA.jar

hibernate-validator-4.2.0.Final.jar

四、Spring3.0支持表现层验证

可以参考我的《最新SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结》或《SpringMVC 使用JSR-303进行校验 @Valid》。

此处不再阐述。

五、 Spring3.0支持依赖注入验证（ Bean Validation 1.1草案 ）

Spring3.0开始支持对依赖注入的依赖进行验证。Spring对依赖注入验证支持请参考《Spring开闭原则的表现-BeanPostProcessor扩展点-2》中的BeanValidationPostProcessor。

示例：

1、Bean组件类定义

```
public class UserModel {
    @NotNull(message = "user.username.null")
    @Pattern(regexp = "[a-zA-Z0-9_]{5,10}", message = "user.username.illegal")
    private String username;
    @Size(min = 5, max=10, message = "password.length.illegal")
    private String password;
    //省略setter/getter
}
```

2、开启依赖注入验证支持（spring-config-bean-validator.xml）

```
<!--注册Bean验证后处理器-->
<bean class="org.springframework.validation.beanvalidation.BeanValidationPostProcessor"/>
```

3、Bean的XML配置定义（spring-config-bean-validator.xml）

```
<bean id="user" class="com.sishuok.validator.UserModel">
    <property name="username" value="@"/>
    <property name="password" value="#"/>
</bean>
```

4、测试用例

```
@RunWith(value = SpringJUnit4ClassRunner.class)
@ContextConfiguration(value = {"classpath:spring-config-bean-validator.xml"})
public class BeanValidatorTest {
```

```
    @Autowired
    UserModel user;
    @Test
    public void test() {
    }
}
```

5、运行测试后，容器启动失败并将看到如下异常：

```
java.lang.IllegalStateException: Failed to load ApplicationContext
……
Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with na
……
Caused by: org.springframework.beans.factory.BeanInitializationException: Bean state is invalid
```

我们可以看出 用户名验证失败。

六、Spring3.1支持方法级别验证（Bean Validation 1.1草案）

Spring3.1开始支持方法级别的验证。Spring对方法级别的验证支持请参考《Spring开闭原则的表现-BeanPostProcessor扩展点-2》中的MethodValidationPostProcessor。

有了方法级别验证，我们就能够更加简单的在Java世界进行契约式设计了，关于契约式设计请参考《建造无错软件：契约式设计引论》。

没有MethodValidationPostProcessor之前我们可能这样验证：

```
public UserModel get(Integer uuid) {
    //前置条件
    Assert.notNull(uuid);
    Assert.isTrue(uuid > 0, "uuid must lt 0");

    //获取 User Model
    UserModel user = new UserModel(); //此处应该从数据库获取
```

```
    //后置条件
    Assert.notNull(user);
    return user;
}
```

前置条件和后置条件的书写是很烦人的工作。

有了MethodValidationPostProcessor之后我们可以这样验证：

```
public @NotNull UserModel get2(@NotNull @Size(min = 1) Integer uuid) {
    //获取 User Model
    UserModel user = new UserModel(); //此处应该从数据库获取
    return user;
}
```

前置条件的验证：在方法的参数上通过Bean Validation注解进行实施；

后置条件的验证：直接在返回值上通过Bean Validation注解进行实施。

非常好，非常好，自此我们可以在Java世界进行更完美的契约式编程了。

示例：

1、Service类定义

```
@Validated      //① 告诉MethodValidationPostProcessor此Bean需要开启方法级别验证支持
public class UserService {
    public @NotNull UserModel get2(@NotNull @Min(value = 1) Integer uuid) { //②声明前置条件/后置
        //获取 User Model
        UserModel user = new UserModel(); //此处应该从数据库获取
        if(uuid > 100) {//方便后置添加的判断（此处假设传入的uuid>100 则返回null）
            return null;
        }
```

```
        return user;
    }
}
```

2、开启Spring3.1对方法级别验证支持（spring-config-method-validator.xml）

```
<!--注册方法验证的后处理器-->
<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

3、Bean的XML配置定义（spring-config-method-validator.xml）

```
<bean id="userService" class="com.sishuok.validator.UserService"/>
```

4、测试用例

```
@RunWith(value = SpringJUnit4ClassRunner.class)
@ContextConfiguration(value = {"classpath:spring-config-method-validator.xml"})
public class MethodValidatorTest {
    @Autowired
    UserService userService;
    @Test
    public void testConditionSuccess() {//① 正常流程
        userService.get2(1);
    }
    @Test(expected = org.hibernate.validator.method.MethodConstraintViolationException.class)
    public void testPreCondtionFail() { //②错误的uuid(即前置条件不满足)
        userService.get2(0);
    }

    @Test(expected = org.hibernate.validator.method.MethodConstraintViolationException.class)
    public void testPostCondtionFail() { //③不满足后置条件的返回值
        userService.get2(10000);
    }
}
```

通过如上测试，我们可以看出Spring3.1已经非常好的支持契约式编程了。

注意，在使用方法级别验证时：

1、由于Bean Validation1.1正处于草案状态，Spring3.1无法支持原生的Bean Validation1.1，在未来的Bean Validation1.1发布时会直接使用原生的。

2、Spring3.1需要使用Hibernate Validator 4.2及更高版本。

让我们期待Bean Validation 1.1的发布吧。

1.9 Spring对事务管理的支持的发展历程（基础篇）

发表时间: 2012-04-24 关键字: spring

1、问题

```
Connection conn =
    DataSourceUtils.getConnection();
 //开启事务
conn.setAutoCommit(false);
try {
    Object retVal =
        callback.doInConnection(conn);
    conn.commit(); //提交事务
    return retVal;
}catch (Exception e) {
    conn.rollback();//回滚事务
    throw e;
}finally {
    conn.close();
}
```

```
Session session = null;
Transaction transaction = null;
try {
    session = factory.openSession();
    //开启事务
    transaction = session.beginTransaction();
    transation.begin();
    session.save(user);
    transaction.commit();//提交事务
} catch (Exception e) {
    e.printStackTrace();
    transaction.rollback();//回滚事务
    return false;
```

```
}finally{
    session.close();
}
```

缺点：不一致的事务管理，复杂

2、高层次解决方案(编程式实现事务)

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

```
        //1.获取事务管理器
        PlatformTransactionManager txManager = (PlatformTransactionManager)
             ctx.getBean("txManager");
        //2.定义事务属性
        DefaultTransactionDefinition td = new DefaultTransactionDefinition();
        td.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
        //3开启事务,得到事务状态
        TransactionStatus status = txManager.getTransaction(td);
        try {
            //4.执行数据库操作
            System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
            //5、提交事务
            txManager.commit(status);

        }catch (Exception e) {
            //6、回滚事务
            txManager.rollback(status);
        }
```

3、高层次解决方案（模板解决方案）

```
//1.获取事务管理器
PlatformTransactionManager txManager = (PlatformTransactionManager)
       ctx.getBean("txManager");
//2、定义事务管理的模板
TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
//3.定义事务属性
transactionTemplate.
    setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
//4.回调，执行真正的数据库操作，如果需要返回值需要在回调里返回
transactionTemplate.execute(new TransactionCallback() {
    @Override
    public Object doInTransaction(TransactionStatus status) {
    //5.执行数据库操作
    System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
    return null;
    }
});
```

4、AOP解决方案

nSpring框架提供了一致的事务管理抽象，这带来了以下好处：

1：为复杂的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO

2：支持声明式事务管理

3：提供比复杂的事务API（诸如JTA）更简单的、更易于使用的编程式事务管理API

4：非常好地整合Spring的各种数据访问抽象

实施事务的步骤

1、定义(资源)DataSource/SessionFactory……

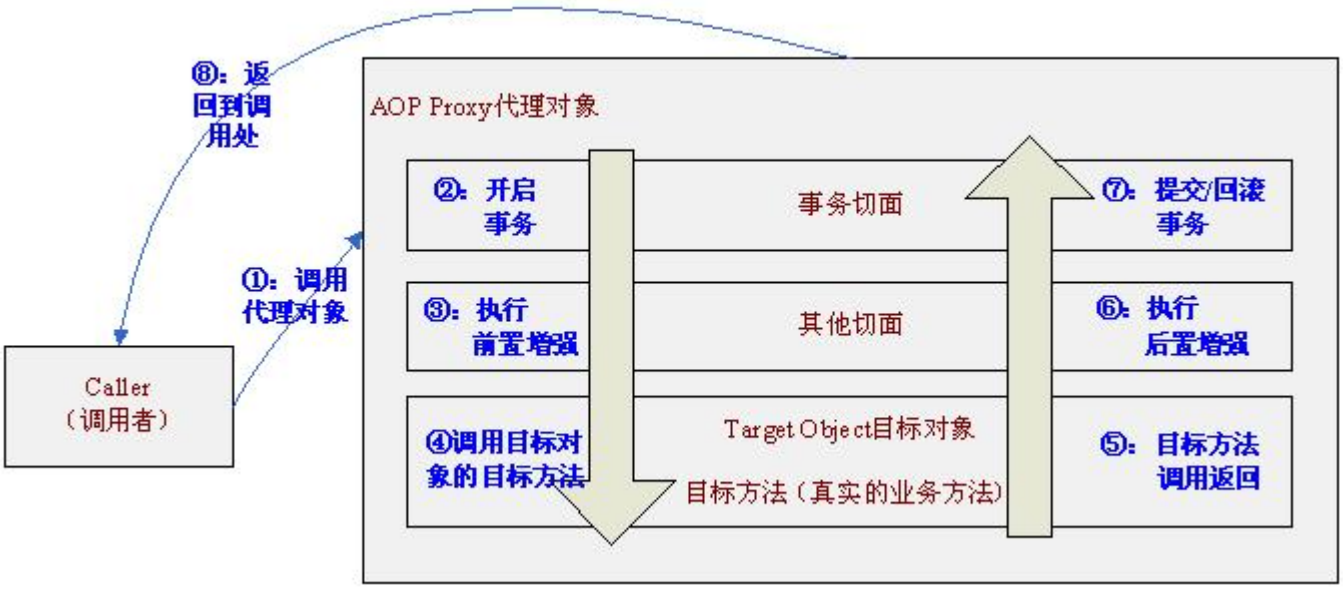
2、定义事务管理器（管理资源的事务）

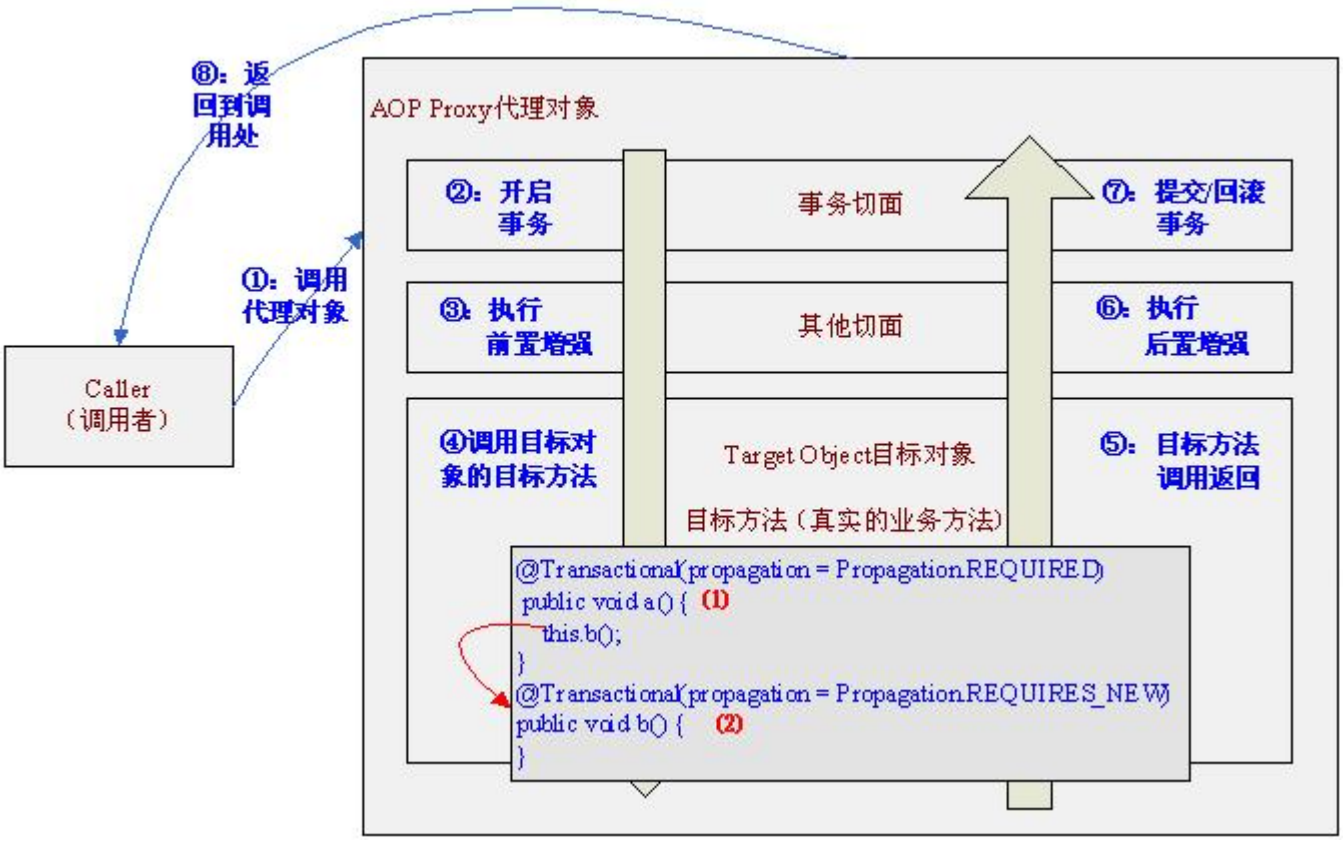
3、定义事务通知：定义了如何实施事务（实施事务的方法名和对应的事务属性），需要使用事务管理器管理事务，定义了如何选择目标对象的方法及实施的事务属性

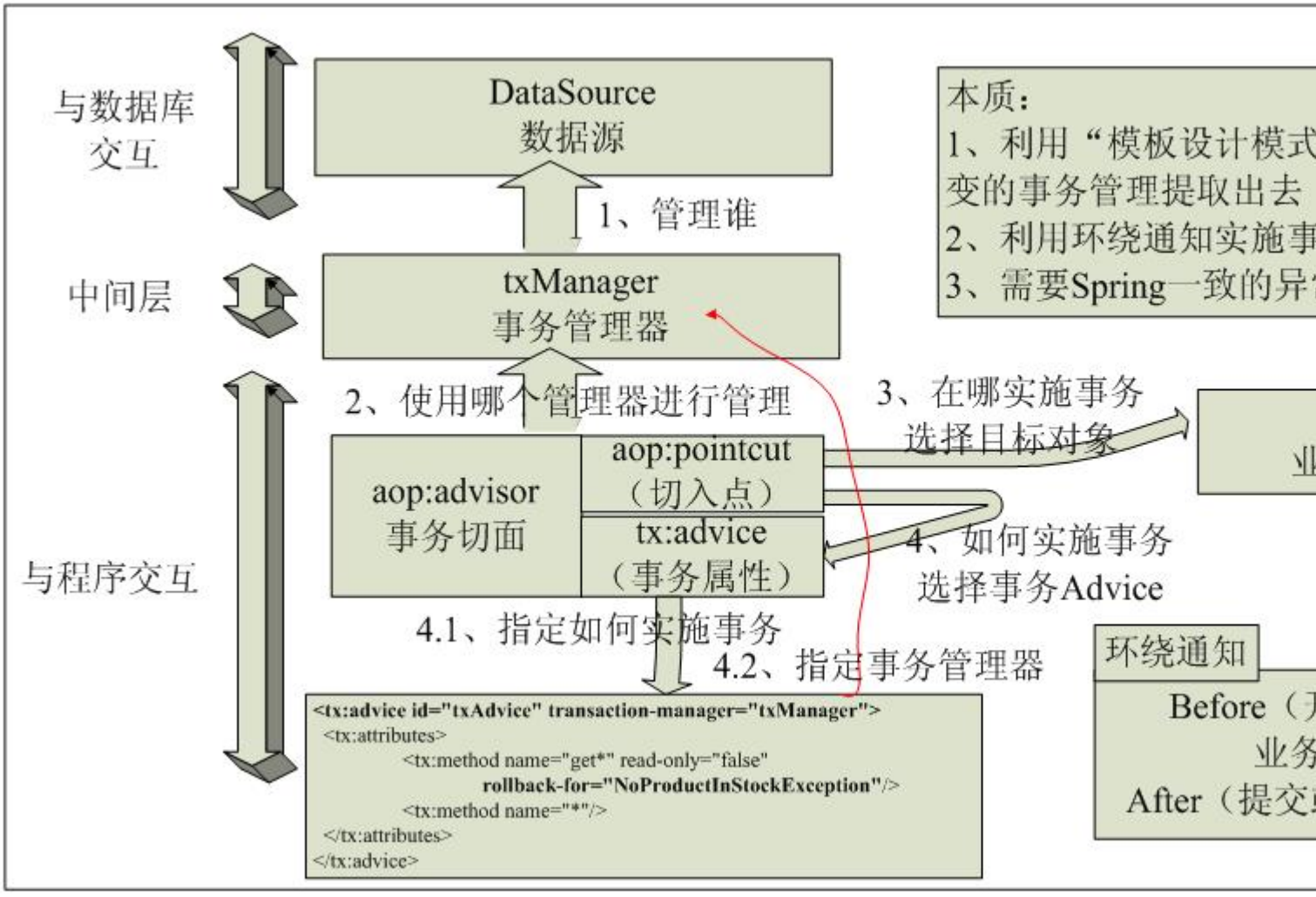
4、定义advisor（切入点和事务通知）：切入点选择需要实施事务的目标对象

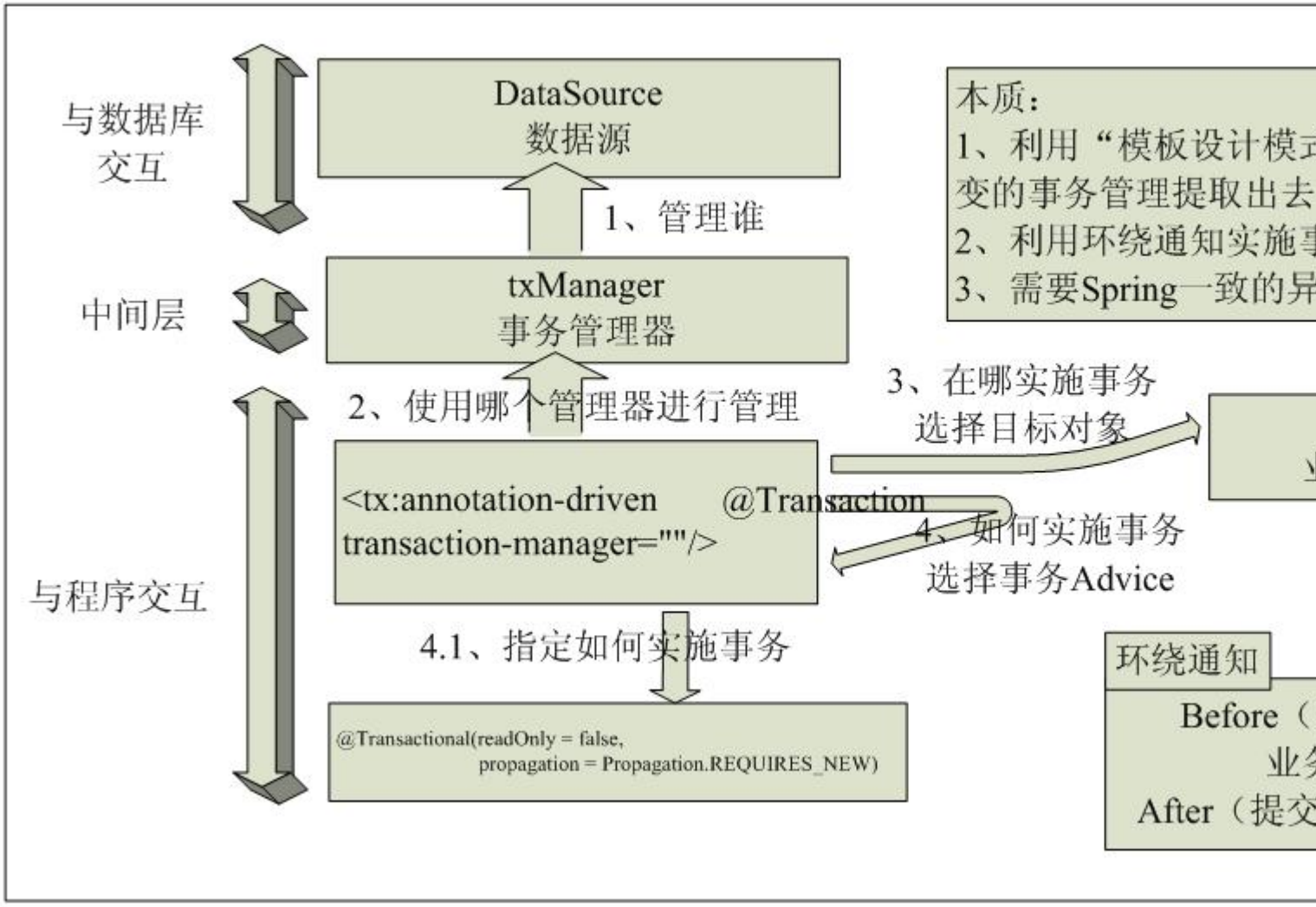
5、Spring织入事务通知到目标对象（AOP代理）

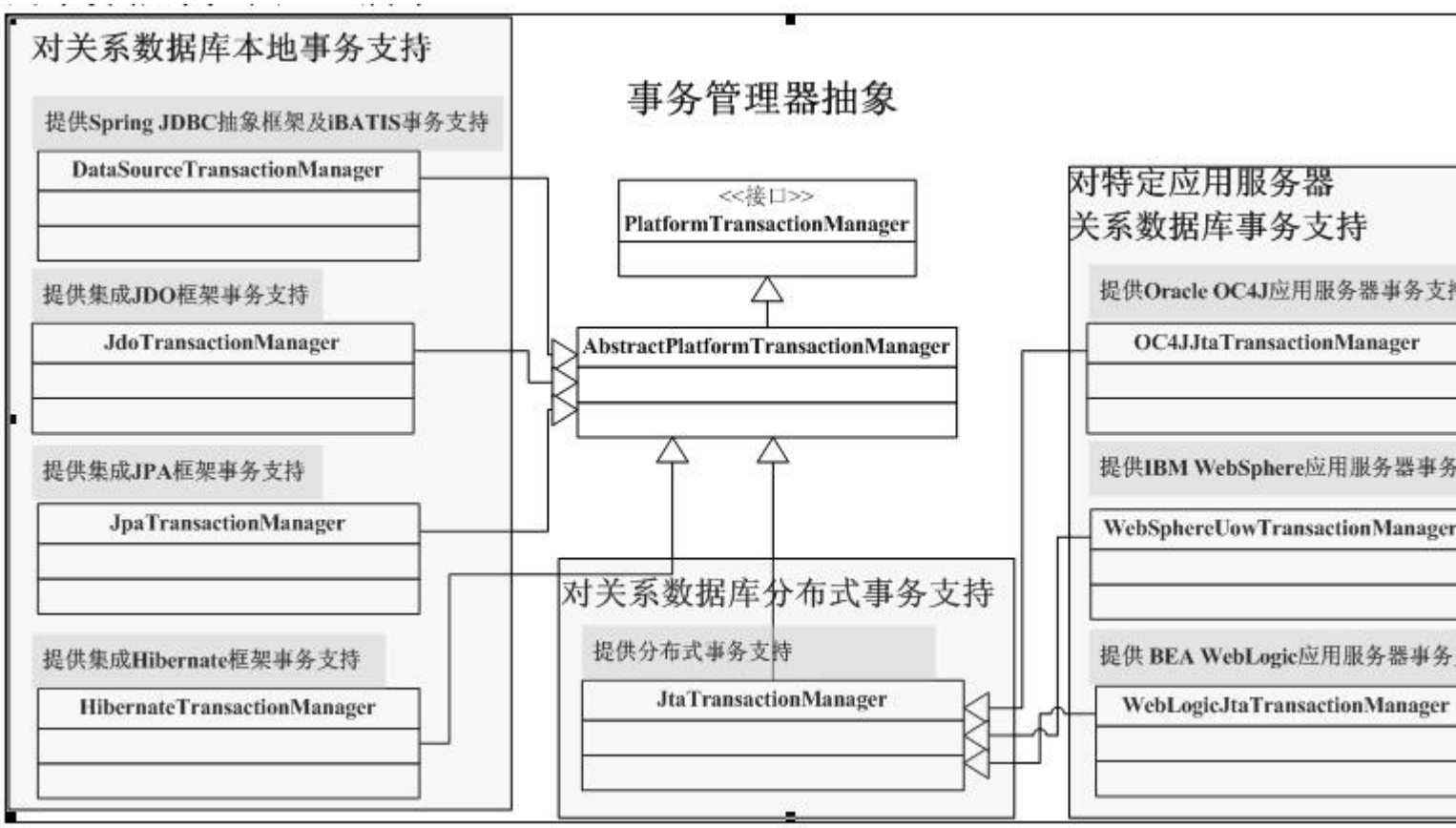
实施流程：


⑧：返回到调用处
AOP Proxy代理对象
②：开启事务
事务切面
⑦：提交/回滚事务
①：调用代理对象
③：执行前置增强
其他切面
⑥：执行后置增强
Caller（调用者）
④调用目标对象的目标方法
Target Object目标对象
目标方法（真实的业务方法）
⑤：目标方法调用返回


⑧：返回到调用处
AOP Proxy代理对象
②：开启事务
事务切面
⑦：提交/回滚事务
①：调用代理对象
③：执行前置增强
其他切面
⑥：执行后置增强
Caller（调用者）
④调用目标对象的目标方法
Target Object目标对象
目标方法（真实的业务方法）
⑤：目标方法调用返回
@Transactional(propagation = Propagation.REQUIRED)
public void a() { (1)
this.b();
}
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void b() { (2)
}


与数据库交互
DataSource
数据源
1、管理谁
本质：
1、利用“模板设计模式
变的事务管理提取出去
2、利用环绕通知实施事
3、需要Spring一致的异
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
3、在哪实施事务
选择目标对象
aop:advisor
事务切面
aop:pointcut
（切入点）
tx:advice
（事务属性）
4、如何实施事务
选择事务Advice
与程序交互
4.1、指定如何实施事务
4.2、指定事务管理器
环绕通知
Before（
业务
After（提交
<tx:advice id="txAdvice" transaction-manager="txManager">
<tx:attributes>
<tx:method name="get*" read-only="false"
rollback-for="NoProductInStockException"/>
<tx:method name="*"/>
</tx:attributes>
</tx:advice>


与数据库交互
DataSource
数据源
1、管理谁
本质：
1、利用“模板设计模式
变的事务管理提取出去
2、利用环绕通知实施事
3、需要Spring一致的异
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
3、在哪实施事务
选择目标对象
<tx:annotation-driven
transaction-manager=""/>
@Transaction
4、如何实施事务
选择事务Advice
与程序交互
4.1、指定如何实施事务
环绕通知
Before（
业务
After（提交
@Transactional(readOnly = false,
propagation = Propagation.REQUIRES_NEW)



更多相关知识请参考：

Spring事务处理时自我调用的解决方案及一些实现方式的风险

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3

【第九章】 Spring的事务 之 9.3 编程式事务 ——跟我学spring3

【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3

1.10 基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

发表时间: 2012-05-02

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

我还是喜欢基于Schema风格的Spring事务管理，但也有很多人在用基于@Trasactional注解的事务管理，但在通过基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务是有区别的，我们接下来看看到底有哪些区别。

一、基础工作

首先修改我们上一次做的 SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结，如下所示：

将xml声明式事务删除

java代码：

```
<aop:config expose-proxy="true">
        <!-- 只对业务逻辑层实施事务 -->
        <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*.*(..))" 
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
    </aop:config>
```

并添加注解式事务支持：

java代码：

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

在我们的BaseService接口上添加 @Transactional 使该方法开启事务

java代码：

```
package cn.javass.common.service;
public interface IBaseService<M extends java.io.Serializable, PK extends java.io.Serializabl
@Transactional   //开启默认事务
    public int countAll();
}
```

在我们的log4j.properties中添加如下配置，表示输出spring的所有debug信息

java代码：

```
log4j.logger.org.springframework=INFO,CONSOLE
```

在我们的resources.properties里将hibernate.show_sql=true 改为true，为了看到hibernate的sql。

单元测试类：

java代码：

```
package cn.javass.ssonline.spider.service.impl;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.transaction.TransactionConfiguration;

import cn.javass.demo.service.UserService;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:spring-config.xml"})
public class UserServiceTest2 {

    @Autowired
    private UserService userService;
    @Test
    public void testCreate() {
       userService.countAll();
    }
}
```

基础工作做好，接下来我们详细看看 Spring基于 JDK动态代理 和 CGLIB类级别代理到底有什么区别。

二、基于JDK动态代理：

java代码：

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

该配置方式默认就是JDK动态代理方式

运行单元测试，核心日志如下：

java代码：

```
2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionMana
2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionMana

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchron
2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchron
2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.interceptor.TransactionInte
2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchron
2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionMana

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchron
Hibernate:
    select
        count(*) as col_0_0_
    from
        tbl_user usermodel0_

2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionMana

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchron
```

到此我们可以看到事务起作用了，也就是说即使把@Transactional放到接口上 基于JDK动态代理也是可以工作的。

三、基于CGLIB类代理：

java代码：

```
<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>
```

该配置方式是基于CGLIB类代理

启动测试会报错，No Session found for current thread，说明事务没有起作用

java代码：

```
org.hibernate.HibernateException: No Session found for current thread
    at org.springframework.orm.hibernate4.SpringSessionContext.currentSession(SpringSessionC
    at org.hibernate.internal.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:1
    at cn.javass.common.dao.hibernate4.BaseHibernateDao.getSession(BaseHibernateDao.java:63)
    at cn.javass.common.dao.hibernate4.BaseHibernateDao.aggregate(BaseHibernateDao.java:238)
    at cn.javass.common.dao.hibernate4.BaseHibernateDao.countAll(BaseHibernateDao.java:114)
    at cn.javass.common.service.impl.BaseService.countAll(BaseService.java:60)
    at cn.javass.common.service.impl.BaseService$$FastClassByCGLIB$$5b04dd69.invoke(<generat
    at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:149)
    at org.springframework.aop.framework.Cglib2AopProxy$DynamicAdvisedInterceptor.intercept(
    at cn.javass.demo.service.impl.UserServiceImpl$$EnhancerByCGLIB$$7d46c567.countAll(<gene
    at cn.javass.ssonline.spider.service.impl.UserServiceTest2.testCreate(UserServiceTest2.j
```

如果将注解放在具体类上或具体类的实现方法上才会起作用。

java代码：

```
package cn.javass.common.service.impl;
public abstract class BaseService<M extends java.io.Serializable, PK extends java.io.Seriali

   @Transactional()   //放在抽象类上
    @Override
    public int countAll() {
        return baseDao.countAll();
    }
}
```

运行测试类，将发现成功了，因为我们的UserService继承该方法，但如果UserService覆盖该方法，如下所示，也将无法织入事务(报错)：

java代码：

```
package cn.javass.demo.service.impl;
public class UserServiceImpl extends BaseService<UserModel, Integer> implements UserService
    //没有@Transactional
    @Override
    public int countAll() {
        return baseDao.countAll();
    }
}
```

四、基于aspectj的

java代码：

```
<tx:annotation-driven transaction-manager="txManager" mode="aspectj" proxy-target-class="tru
```

在此就不演示了，我们主要分析基于JDK动态代理和CGLIB类代理两种的区别。

五、结论：

基于JDK动态代理 ，可以将@Transactional放置在接口和具体类上。

基于CGLIB类代理，只能将@Transactional放置在具体类上。

因此 在实际开发时全部将@Transactional放到具体类上，而不是接口上。

六、分析

1、 JDK动态代理

1.1、Spring使用JdkDynamicAopProxy实现代理：

java代码：

```
package org.springframework.aop.framework;
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {
    //注意此处的method 一定是接口上的method（因此放置在接口上的@Transactional是可以发现的）
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    }
}
```

注意此处的method 一定是接口上的method（因此放置在接口上的@Transactional是可以发现的）

1.2、如果<tx:annotation-driven 中 proxy-target-class="true" ，Spring将使用CGLIB动态代理，而内部通过Cglib2AopProxy实现代理，而内部通过DynamicAdvisedInterceptor进行拦截：

java代码：

```
package org.springframework.aop.framework;
final class Cglib2AopProxy implements AopProxy, Serializable {
        private static class DynamicAdvisedInterceptor implements MethodInterceptor, Seriali
            //注意此处的method 一定是具体类上的method（因此只用放置在具体类上的@Transactional是可
            public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
            }
       }
}
```

1.3、Spring使用AnnotationTransactionAttributeSource通过查找一个类或方法是否有@Transactional注解事务来返回TransactionAttribute（表示开启事务）：

java代码：

```
package org.springframework.transaction.annotation;
public class AnnotationTransactionAttributeSource extends AbstractFallbackTransactionAttribu
         protected TransactionAttribute determineTransactionAttribute(AnnotatedElement ae) {
                for (TransactionAnnotationParser annotationParser : this.annotationParsers)
                        TransactionAttribute attr = annotationParser.parseTransactionAnnotat
                        if (attr != null) {
                                return attr;
                        }
                }
                return null;
        }
}
```

而AnnotationTransactionAttributeSource又使用SpringTransactionAnnotationParser来解析是否有@Transactional注解：

java代码：

```
package org.springframework.transaction.annotation;

public class SpringTransactionAnnotationParser implements TransactionAnnotationParser, Seria

        public TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae) {
                Transactional ann = AnnotationUtils.getAnnotation(ae, Transactional.class);
                if (ann != null) {
                        return parseTransactionAnnotation(ann);
                }
                else {
                        return null;
                }
        }

        public TransactionAttribute parseTransactionAnnotation(Transactional ann) {

    }

}
```

此处使用AnnotationUtils.getAnnotation(ae, Transactional.class)； 这个方法只能发现当前方法/类上的注解，不能发现父类的注解。 Spring还提供了一个 AnnotationUtils.findAnnotation()方法 可以发现父类/父接口中的注解（但spring没有使用该接口）。

如果Spring此处换成AnnotationUtils.findAnnotation()，将可以发现父类/父接口中的注解。

这里还一个问题，描述如下：

在接口中删除@Transactional //开启默认事务

java代码：

```
package cn.javass.common.service;
public interface IBaseService<M extends java.io.Serializable, PK extends java.io.Serializabl
    public int countAll();
}
```

在具体类中添加@Transactional

java代码：

```
package cn.javass.common.service.impl;
public abstract class BaseService<M extends java.io.Serializable, PK extends java.io.Seriali

    @Transactional()   //开启默认事务
    @Override
    public int countAll() {
        return baseDao.countAll();
    }
}
```

问题：

我们之前说过，基于JDK动态代理时， method 一定是接口上的method（因此放置在接口上的@Transactional是可以发现的），但现在我们放在具体类上，那么Spring是如何发现的呢？？

还记得发现TransactionAttribute是通过AnnotationTransactionAttributeSource吗？具体看步骤1.3：

而AnnotationTransactionAttributeSource 继承AbstractFallbackTransactionAttributeSource

java代码：

```
package org.springframework.transaction.interceptor;
public abstract class AbstractFallbackTransactionAttributeSource implements TransactionAttri

        public TransactionAttribute getTransactionAttribute(Method method, Class<?> targetCl
                //第一次 会委托给computeTransactionAttribute
}

    //计算TransactionAttribute的
        private TransactionAttribute computeTransactionAttribute(Method method, Class<?> tar

                //省略

                // Ignore CGLIB subclasses - introspect the actual user class.
                Class<?> userClass = ClassUtils.getUserClass(targetClass);
                // The method may be on an interface, but we need attributes from the target
                // If the target class is null, the method will be unchanged.
                //①此处将查找当前类覆盖的方法
                Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);
```

```
                // If we are dealing with method with generic parameters, find the original m
                specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);

                // First try is the method in the target class.
                TransactionAttribute txAtt = findTransactionAttribute(specificMethod);
                if (txAtt != null) {
                        return txAtt;
                }

                //找类上边的注解
                // Second try is the transaction attribute on the target class.
                txAtt = findTransactionAttribute(specificMethod.getDeclaringClass());
                if (txAtt != null) {
                        return txAtt;
                }
                //②如果子类覆盖的方法没有 再直接找当前传过来的
                if (specificMethod != method) {
                        // Fallback is to look at the original method.
                        txAtt = findTransactionAttribute(method);
                        if (txAtt != null) {
                                return txAtt;
                        }
                        // Last fallback is the class of the original method.
                        return findTransactionAttribute(method.getDeclaringClass());
                }
                return null;
        }
}
```

//①此处将查找子类覆盖的方法

Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);

```
    // ClassUtils.getMostSpecificMethod
    public static Method getMostSpecificMethod(Method method, Class<?> targetClass) {
    Method specificMethod = null;
    if (method != null && isOverridable(method, targetClass) &&
        targetClass != null && !targetClass.equals(method.getDeclaringClass())) {
      try {
        specificMethod = ReflectionUtils.findMethod(targetClass, method.getName(),
method.getParameterTypes());
      } catch (AccessControlException ex) {
        // security settings are disallowing reflective access; leave
        // 'specificMethod' null and fall back to 'method' below
      }
    }
    return (specificMethod != null ? specificMethod : method);
  }
```

可以看出将找到当前类的那个方法。因此我们放置在BaseService countAll方法上的@Transactional起作用了。

```
  //②如果子类覆盖的方法没有 再直接找当前传过来的
    if (specificMethod != method) {
      // Fallback is to look at the original method.
      txAtt = findTransactionAttribute(method);
      if (txAtt != null) {
```

```
            return txAtt;
        }
        // Last fallback is the class of the original method.
        return findTransactionAttribute(method.getDeclaringClass());
    }
```

查找子类失败时直接使用传过来的方法。

因此，建议大家使用基于Schema风格的事务（不用考虑这么多问题，也不用考虑是类还是方法）。而@Transactional建议放置到具体类上，不要放置到接口。

作者原创【http://sishuok.com/forum/blogPost/list/0/3845.html#9317】

1.11 在spring中获取代理对象代理的目标对象工具类

发表时间: 2012-07-31

昨天晚上一哥们需要获取代理对象的目标对象，查找了文档发现没有相应的工具类，因此自己写了一个分享给大家。能获取JDK动态代理/CGLIB代理对象代理的目标对象。

问题描述：：

我现在遇到个棘手的问题,要通过spring托管的service类保存对象,这个类是通过反射拿到的,经过实验发现这个类只能反射取得sservice实现了接口的方法,而extends类的方法一律不出现,debug后发现这个servie实例被spring替换成jdkdynmicproxy类,而不是原始对象了,,它里面只有service继承的接口方法,而没有extends 过的super class方法,怎么调用原生对象的方法!!!!!

用托管的spring service类调用getClass().getName()方法,发现输出都是$proxy43这类东西!!

通过此种方式获取目标对象是不可靠的，或者说任何获取目标对象的方式都是不可靠的，因为TargetSource，TargetSource中存放了目标对象，但TargetSource有很多种实现，默认我们使用的是SingletonTargetSource ，但还有其他的比如ThreadLocalTargetSource、CommonsPoolTargetSource 等等。

这也是为什么spring没有提供获取目标对象的API。

```
import java.lang.reflect.Field;
```

```
import org.springframework.aop.framework.AdvisedSupport;
import org.springframework.aop.framework.AopProxy;
import org.springframework.aop.support.AopUtils;

public class AopTargetUtils {



	/**
	 * 获取 目标对象
	 * @param proxy 代理对象
	 * @return
	 * @throws Exception
	 */
	public static Object getTarget(Object proxy) throws Exception {

		if(!AopUtils.isAopProxy(proxy)) {
			return proxy;//不是代理对象
		}

		if(AopUtils.isJdkDynamicProxy(proxy)) {
			return getJdkDynamicProxyTargetObject(proxy);
		} else { //cglib
			return getCglibProxyTargetObject(proxy);
		}




	}


	private static Object getCglibProxyTargetObject(Object proxy) throws Exception {
		Field h = proxy.getClass().getDeclaredField("CGLIB$CALLBACK_0");
	h.setAccessible(true);
	Object dynamicAdvisedInterceptor = h.get(proxy);

	Field advised = dynamicAdvisedInterceptor.getClass().getDeclaredField("advised");
	advised.setAccessible(true);
```

```
        Object target = ((AdvisedSupport)advised.get(dynamicAdvisedInterceptor)).getTargetSourc

        return target;
        }


        private static Object getJdkDynamicProxyTargetObject(Object proxy) throws Exception {
                Field h = proxy.getClass().getSuperclass().getDeclaredField("h");
        h.setAccessible(true);
        AopProxy aopProxy = (AopProxy) h.get(proxy);

        Field advised = aopProxy.getClass().getDeclaredField("advised");
        advised.setAccessible(true);

        Object target = ((AdvisedSupport)advised.get(aopProxy)).getTargetSource().getTarget();

        return target;
        }

}
```

附件下载:

- AopTargetUtils.rar (664 Bytes)
- dl.iteye.com/topics/download/9b8d5cfa-2b6b-3466-bf38-d47769b2f0b8

1.12 如何为spring代理类设置属性值

发表时间: 2012-09-14

在问答频道 有朋友问《如何为spring代理类设置属性值》 就写了个小工具 供使用。思想就不讲了。

现在有一个bean包含了私有属性，如下：

Java代码

```
1.  @Component
2.  public class Bean {
3.      String name;
4.
5.      public String getName() {
6.          return name;
7.      }
8.
9.      public void setName(String name) {
10.         this.name = name;
11.     }
12.
13. }
```

它被AOP配置过代理，代理配置为：

Java代码

```
1.  <aop:pointcut expression="execution(* com..*Bean.*(..))"
2.              id="txBean" />
```

现在对它进行测试：

Java代码

```
1.  public class BeanTest extends SpringContextTestCase{
2.      @Autowired
3.      private Bean bean;
4.      @Test
5.      public void testBean(){
6.          bean.setName("dylan");
7.          System.out.println(bean.name);
8.          System.out.println(bean.getName());
9.      }
10. }
```

这里的测试结果中，第一个输出为null,第二个输出为dylan,

由于项目中需要直接通过bean.name的方式来获取属性值，却一直都只能得到null，请问如何才能获取到我所期望的值"dylan"呢

默认是没有办法的。我帮你写了个AOP切面 帮你完成设置属性。

Java代码

```
1.  import java.beans.PropertyDescriptor;
2.  import java.lang.reflect.Field;
3.  import java.lang.reflect.Method;
4.
5.  import org.aspectj.lang.JoinPoint;
6.  import org.aspectj.lang.annotation.After;
7.  import org.aspectj.lang.annotation.Aspect;
8.  import org.springframework.aop.support.AopUtils;
9.  import org.springframework.beans.BeanUtils;
10. import org.springframework.core.annotation.Order;
11.
12. @Aspect
13. @Order(Integer.MIN_VALUE)
14. public class SetterAspect {
15.
16.     @After(value="execution(* *.set*(*)) && args(value)", argNames="value")
17.     public void after(JoinPoint jp, Object value) {
18.         Object proxy = jp.getThis();
19.         Object target = jp.getTarget();
20.
21.         if(AopUtils.isAopProxy(proxy)) {//只有代理对象才需要处理
22.
23.             try {
24.                 Class<?> proxyClass = proxy.getClass();
25.                 Class<?> targetClass = target.getClass();
26.                 String methodName = jp.getSignature().getName();
27.
28.                 Method m = BeanUtils.findDeclaredMethod(proxyClass, methodName, new Class[]{value.getClass()});
29.                 PropertyDescriptor descriptor = BeanUtils.findPropertyForMethod(m);
30.                 String propName = descriptor.getName();
31.
32.                 Field f = targetClass.getClass().getDeclaredField(propName);
33.                 if(f != null) {
34.                     f.setAccessible(true);
35.                     f.set(proxy, value);
36.                 }
37.             } catch (Exception e) {
38.                 e.printStackTrace();//记录好异常进行处理
39.             }
40.         }
41.     }
42.
43. }
```

1.13 我对SpringDAO层支持的总结

发表时间: 2012-09-26

之前发过两篇关于Spring的总结帖子 反响还不错，再把剩下的几篇发上来。共享给大家。

我对IoC/DI的理解

我对AOP的理解

1、问题

1、JDBC/ORM框架（如Hibernate）开发中编程模型有哪些缺点？ 如JDBC

2、解决方案（模板设计模式，本质：将可变的和不可变的分离）

模板方法模式：定义操作的步骤（固定的），将可变的步骤交给子类处理。

```
public interface JdbcOperations {
    //接口定义行为集
    public Object execute() throws SQLException ;
}
```

```
public abstract class AbstractJdbcOperations implements JdbcOperations {
    @Override
    public final Object execute() throws SQLException {
       Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = doInConnection(conn);
            conn.commit();
            return retVal;
        }catch (Exception e) {   conn.rollback(); throw e;}
        finally {                          conn.close(); }
    }
    public abstract Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class DataSourceUtils {
    public static Connection getConnection() {
        //返回数据库连接
        return null;
    }
}
```

```
  JdbcOperations select = new AbstractJdbcOperations() {
          @Override
           public Object doInConnection(Connection conn) throws SQLException {
               PreparedStatement pstmt =
                   conn.prepareStatement("select * from tbl_user");
               ResultSet rs = pstmt.executeQuery();
               List result = null;
               //处理结果集
               return result;
           }
        };
        select.execute();
```

缺点：不够灵活

3、解决方案（模板设计模式+回调，本质：将可变的和不可变的分离，可变通过回调）

回调（命令）：由组件定义，但不由组件调用，而是由系统调用

一般用于：可变不可变分离，未知功能。

```
public interface JdbcOprations2 {
    //接口定义行为集
    public Object execute(ConnectionCallback callback) throws Exception ;

}
```

```
public interface ConnectionCallback {
    public Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class JdbcTemplate implements JdbcOprations2 {
    @Override
    public Object execute(ConnectionCallback callback) throws Exception {
        Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = callback.doInConnection(conn);
            conn.commit();
            return retVal;
        }
        catch (Exception e) {
            conn.rollback();
            throw e;
        }
        finally {
            conn.close();
        }
    }
}
```

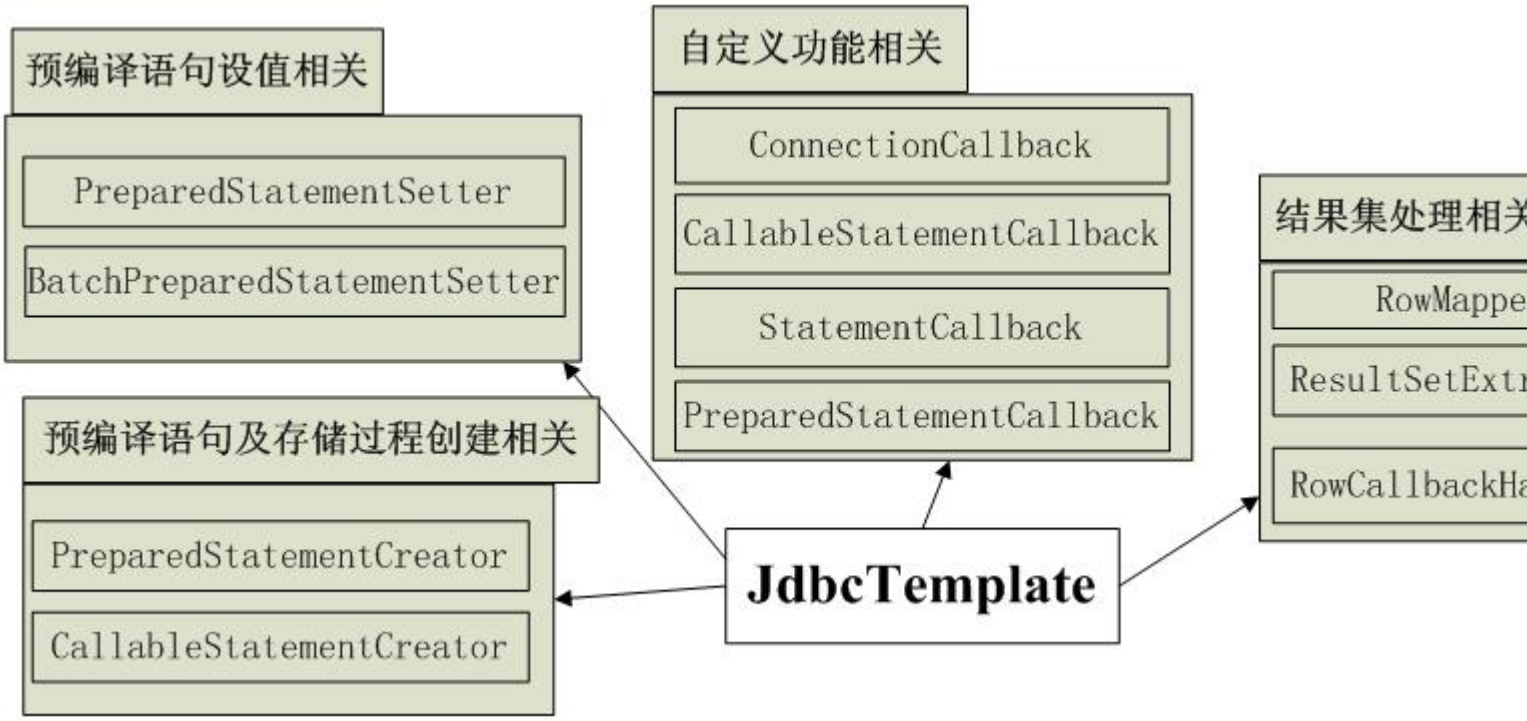
```
jdbcTemplate.execute(new ConnectionCallback() {

            @Override
```

```
            public Object doInConnection(Connection conn) throws SQLException {
                //可变操作
                return null;
            }
        });
```

```
public interface PreparedStatementCallback {
    public Object doInPreparedStatement(PreparedStatement pstmt)
        throws  SQLException;
}
```

缺点：灵活但不通用

4、解决方案（Spring JDBC框架）

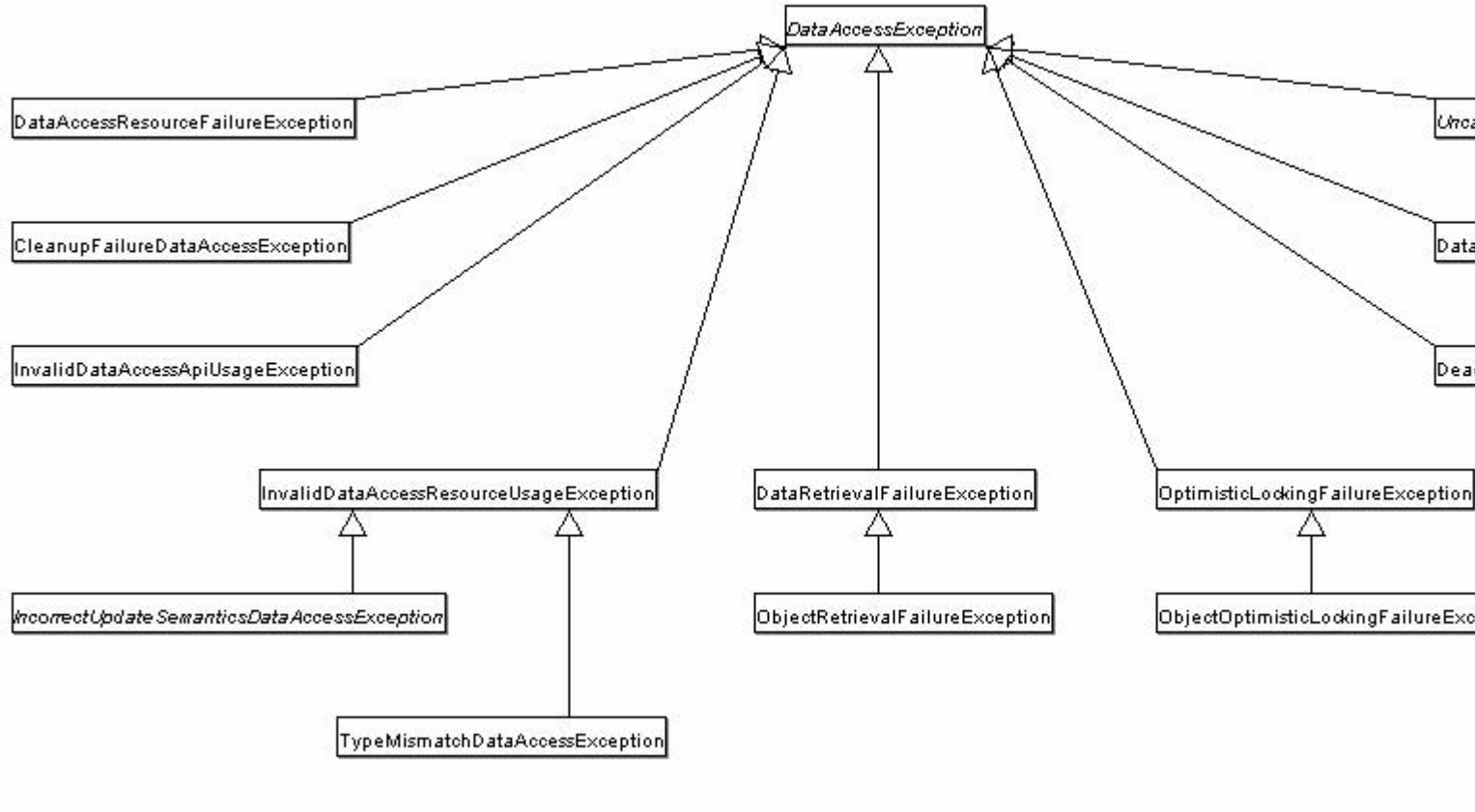


JdbcTemplate jdbcTemplate = **new** JdbcTemplate(ds);

5、异常问题

JDBC/ORM框架（如Hibernate）开发中异常处理有哪些缺点？ 不一致的异常类，如JDBC是SQLException 而hibernate是HibernateException，等。

缺点：不一致的异常体系，需要了解每个框架的异常含义 ，Spring的一致的异常体系（DataAccessException）



6、DAO访问问题

访问JDBC和访问Hibernate使用完全不同且根本不类似的API？

为了便于以一种一致的方式使用各种数据访问技术，如JDBC、JDO和Hibernate，

Spring提供了一套抽象DAO类供你扩展。

JdbcDaoSupport - JDBC数据访问对象的基类。

需要一个DataSource，同时为子类提供 JdbcTemplate。

HibernateDaoSupport - Hibernate数据访问对象的基类。

需要一个SessionFactory，同时为子类提供 HibernateTemplate。

JdoDaoSupport - JDO数据访问对象的基类。

需要设置一个PersistenceManagerFactory， 同时为子类提供JdoTemplate。

JpaDaoSupport - JPA数据访问对象的基类。

需要一个EntityManagerFactory，同时 为子类提供JpaTemplate。

一致的DAO抽象

Callback：定义可变行为，即不可预知行为(命令设计模式)

Template：提供模板支持，简化开发

Support：提供对Template的支持，提供一组简便方法，并提供获取Template的方法

相关资料

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3

【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3

【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3

【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3

1.14 我对SpringDAO层支持的总结

发表时间: 2012-09-26

之前发过两篇关于Spring的总结帖子 反响还不错，再把剩下的几篇发上来。共享给大家。

我对IoC/DI的理解

我对AOP的理解

1、问题

1、JDBC/ORM框架（如Hibernate）开发中编程模型有哪些缺点？ 如JDBC

2、解决方案（模板设计模式，本质：将可变的和不可变的分离）

模板方法模式：定义操作的步骤（固定的），将可变的步骤交给子类处理。

```
public interface JdbcOperations {
    //接口定义行为集
    public Object execute() throws SQLException ;
}
```

```
public abstract class AbstractJdbcOperations implements JdbcOperations {
    @Override
    public final Object execute() throws SQLException {
       Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = doInConnection(conn);
            conn.commit();
            return retVal;
        }catch (Exception e) {   conn.rollback(); throw e;}
        finally {                          conn.close(); }
    }
    public abstract Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class DataSourceUtils {
    public static Connection getConnection() {
        //返回数据库连接
        return null;
    }
}
```

```
  JdbcOperations select = new AbstractJdbcOperations() {
          @Override
           public Object doInConnection(Connection conn) throws SQLException {
               PreparedStatement pstmt =
                   conn.prepareStatement("select * from tbl_user");
               ResultSet rs = pstmt.executeQuery();
               List result = null;
               //处理结果集
               return result;
           }
        };
        select.execute();
```

缺点：不够灵活

3、解决方案（模板设计模式+回调，本质：将可变的和不可变的分离，可变通过回调）

回调（命令）：由组件定义，但不由组件调用，而是由系统调用

一般用于：可变不可变分离，未知功能。

```
public interface JdbcOprations2 {
    //接口定义行为集
    public Object execute(ConnectionCallback callback) throws Exception ;

}
```

```
public interface ConnectionCallback {
    public Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class JdbcTemplate implements JdbcOprations2 {
    @Override
    public Object execute(ConnectionCallback callback) throws Exception {
        Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = callback.doInConnection(conn);
            conn.commit();
            return retVal;
        }
        catch (Exception e) {
            conn.rollback();
            throw e;
        }
        finally {
            conn.close();
        }
    }
}
```

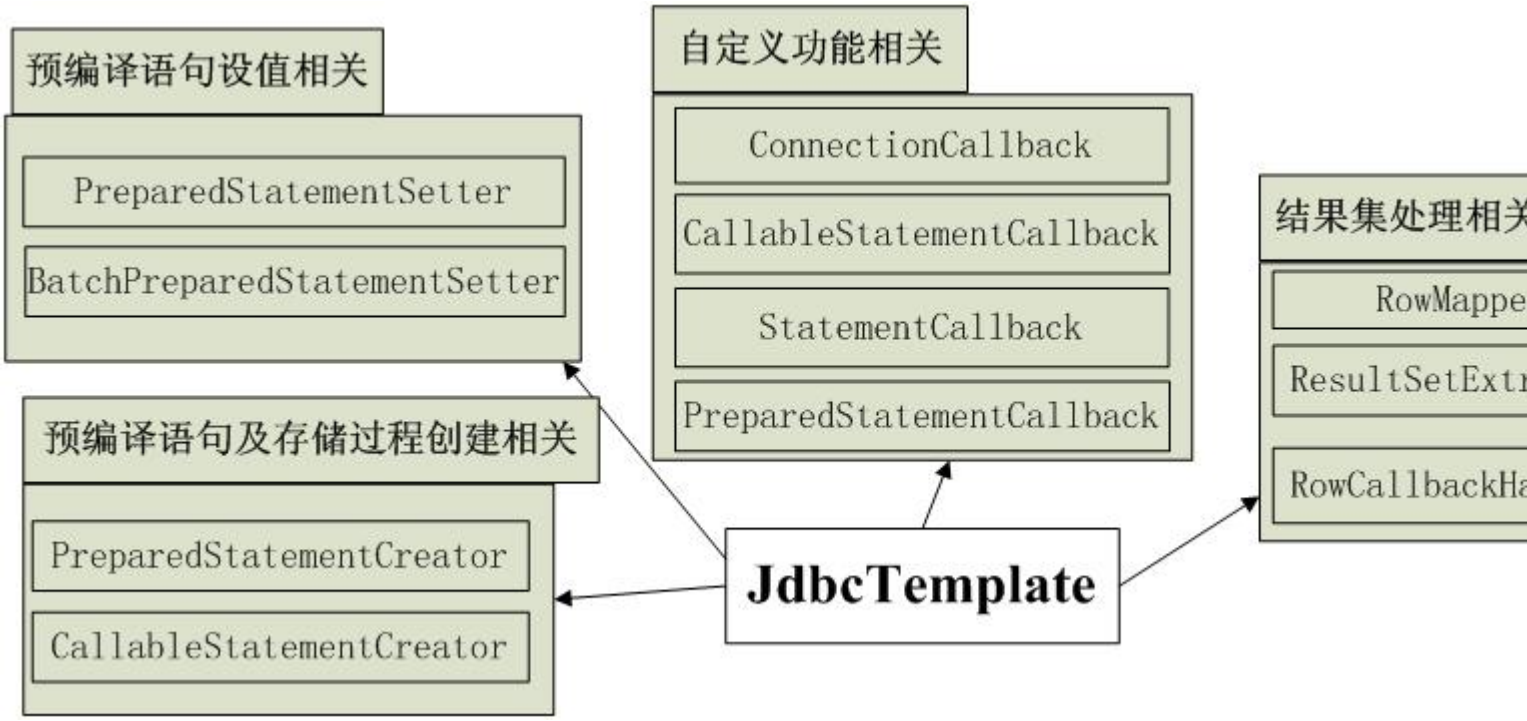
```
jdbcTemplate.execute(new ConnectionCallback() {

            @Override
            public Object doInConnection(Connection conn) throws SQLException {
```

```
                //可变操作
                return null;
            }
        });
```

```
public interface PreparedStatementCallback {
    public Object doInPreparedStatement(PreparedStatement pstmt)
        throws  SQLException;
}
```

缺点：灵活但不通用

4、解决方案（Spring JDBC框架）

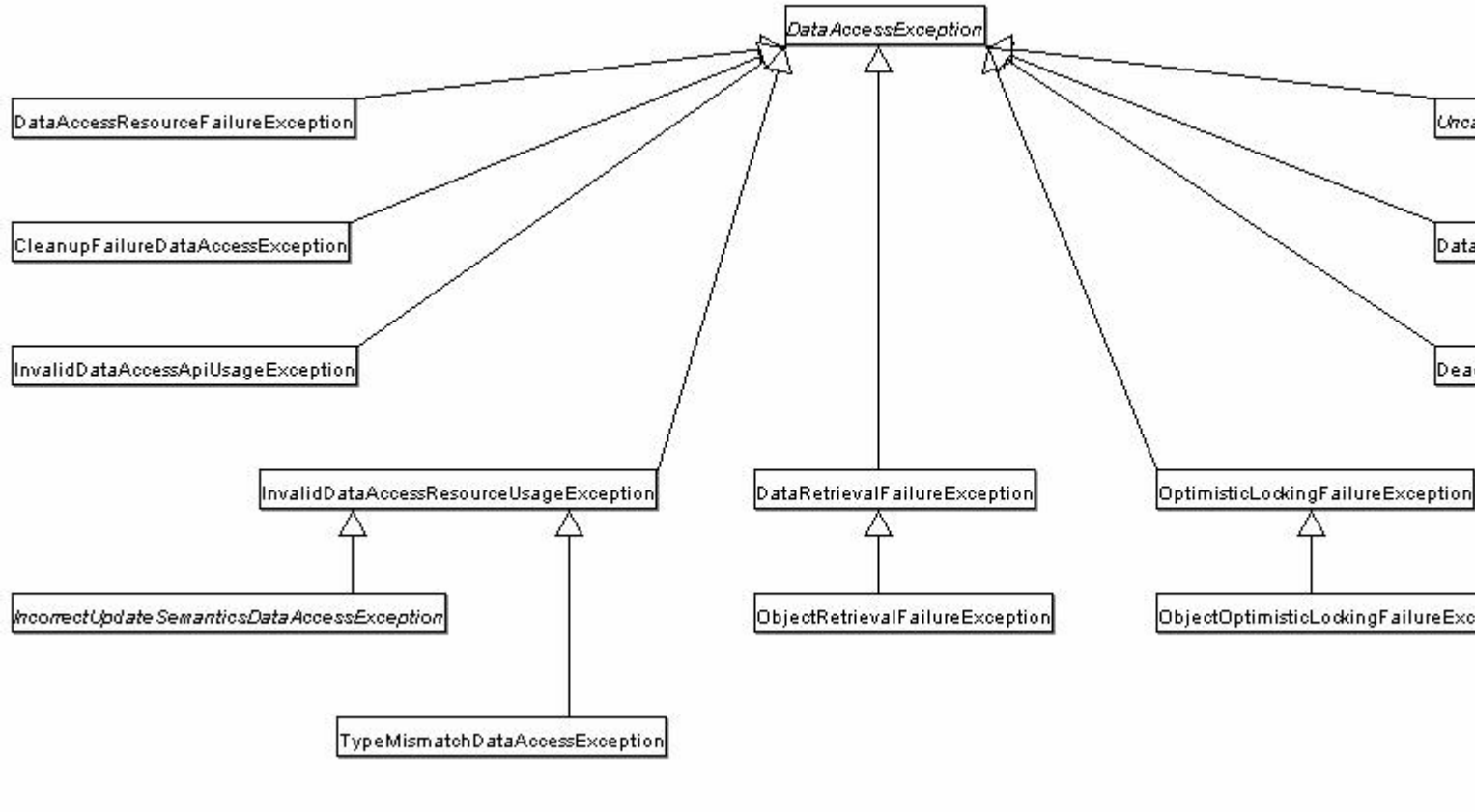


JdbcTemplate jdbcTemplate = **new** JdbcTemplate(ds);

5、异常问题

JDBC/ORM框架（如Hibernate）开发中异常处理有哪些缺点？ 不一致的异常类，如JDBC是SQLException 而hibernate是HibernateException，等。

缺点：不一致的异常体系，需要了解每个框架的异常含义 ，Spring的一致的异常体系（DataAccessException）



6、DAO访问问题

访问JDBC和访问Hibernate使用完全不同且根本不类似的API？

为了便于以一种一致的方式使用各种数据访问技术，如JDBC、JDO和Hibernate，

Spring提供了一套抽象DAO类供你扩展。

JdbcDaoSupport - JDBC数据访问对象的基类。

需要一个DataSource，同时为子类提供 JdbcTemplate。

HibernateDaoSupport - Hibernate数据访问对象的基类。

需要一个SessionFactory，同时为子类提供 HibernateTemplate。

JdoDaoSupport - JDO数据访问对象的基类。

需要设置一个PersistenceManagerFactory， 同时为子类提供JdoTemplate。

JpaDaoSupport - JPA数据访问对象的基类。

需要一个EntityManagerFactory，同时 为子类提供JpaTemplate。

一致的DAO抽象

Callback：定义可变行为，即不可预知行为(命令设计模式)

Template：提供模板支持，简化开发Support：提供对Template的支持，提供一组简便方法，并提供获取Template的方法

相关资料

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3

【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3

【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3

【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3

1.15 我对SpringDAO层支持的总结

发表时间: 2012-09-26

之前发过两篇关于Spring的总结帖子 反响还不错，再把剩下的几篇发上来。共享给大家。

我对IoC/DI的理解

我对AOP的理解

1、问题

1、JDBC/ORM框架（如Hibernate）开发中编程模型有哪些缺点？ 如JDBC

传统JDBC	Spring JDBC
1.获取JDBC连接 2.声明SQL 3.预编译SQL 4.执行SQL 5.处理结果集 6.释放结果集 7.释放Statement 8.提交事务 9.处理异常并回滚事务 10.释放JDBC连接	~~1.获取JDBC连接~~ 2.声明SQL （√） ~~3.预编译SQL~~ ~~4.执行SQL~~ 5.处理结果集（√） ~~6.释放结果集~~ ~~7.释放Statement~~ ~~8.提交事务~~ ~~9.处理异常并回滚事务~~ ~~10.释放JDBC连接~~
缺点： 1.冗长、重复 2.显示事务控制 3.每个步骤不可或缺 4.显示处理受检查异常	优点： 1.简单、简洁 2.Spring事务管理 3.只做需要做的 4.一致的非检查异常体系

2、解决方案（模板设计模式，本质：将可变的和不可变的分离）

模板方法模式：定义操作的步骤（固定的），将可变的步骤交给子类处理。

```
public interface JdbcOperations {
    //接口定义行为集
```

```
    public Object execute() throws SQLException ;
}
```

```
public abstract class AbstractJdbcOperations implements JdbcOperations {
    @Override
    public final Object execute() throws SQLException {
       Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = doInConnection(conn);
            conn.commit();
            return retVal;
        }catch (Exception e) {   conn.rollback(); throw e;}
        finally {                          conn.close(); }
    }
    public abstract Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class DataSourceUtils {
    public static Connection getConnection() {
        //返回数据库连接
        return null;
    }
}
```

```
  JdbcOperations select = new AbstractJdbcOperations() {
           @Override
            public Object doInConnection(Connection conn) throws SQLException {
                PreparedStatement pstmt =
                    conn.prepareStatement("select * from tbl_user");
                ResultSet rs = pstmt.executeQuery();
                List result = null;
                //处理结果集
                return result;
            }
```

```
        };
        select.execute();
```

缺点：不够灵活

3、解决方案（模板设计模式+回调，本质：将可变的和不可变的分离，可变通过回调）

回调（命令）：由组件定义，但不由组件调用，而是由系统调用

一般用于：可变不可变分离，未知功能。

```
public interface JdbcOprations2 {
    //接口定义行为集
    public Object execute(ConnectionCallback callback) throws Exception ;

}
```

```
public interface ConnectionCallback {
    public Object doInConnection(Connection conn) throws SQLException;
}
```

```
public class JdbcTemplate implements JdbcOprations2 {
    @Override
    public Object execute(ConnectionCallback callback) throws Exception {
        Connection conn = DataSourceUtils.getConnection();
        try {
            Object retVal = callback.doInConnection(conn);
            conn.commit();
            return retVal;
        }
        catch (Exception e) {
            conn.rollback();
            throw e;
        }
        finally {
            conn.close();
        }
```

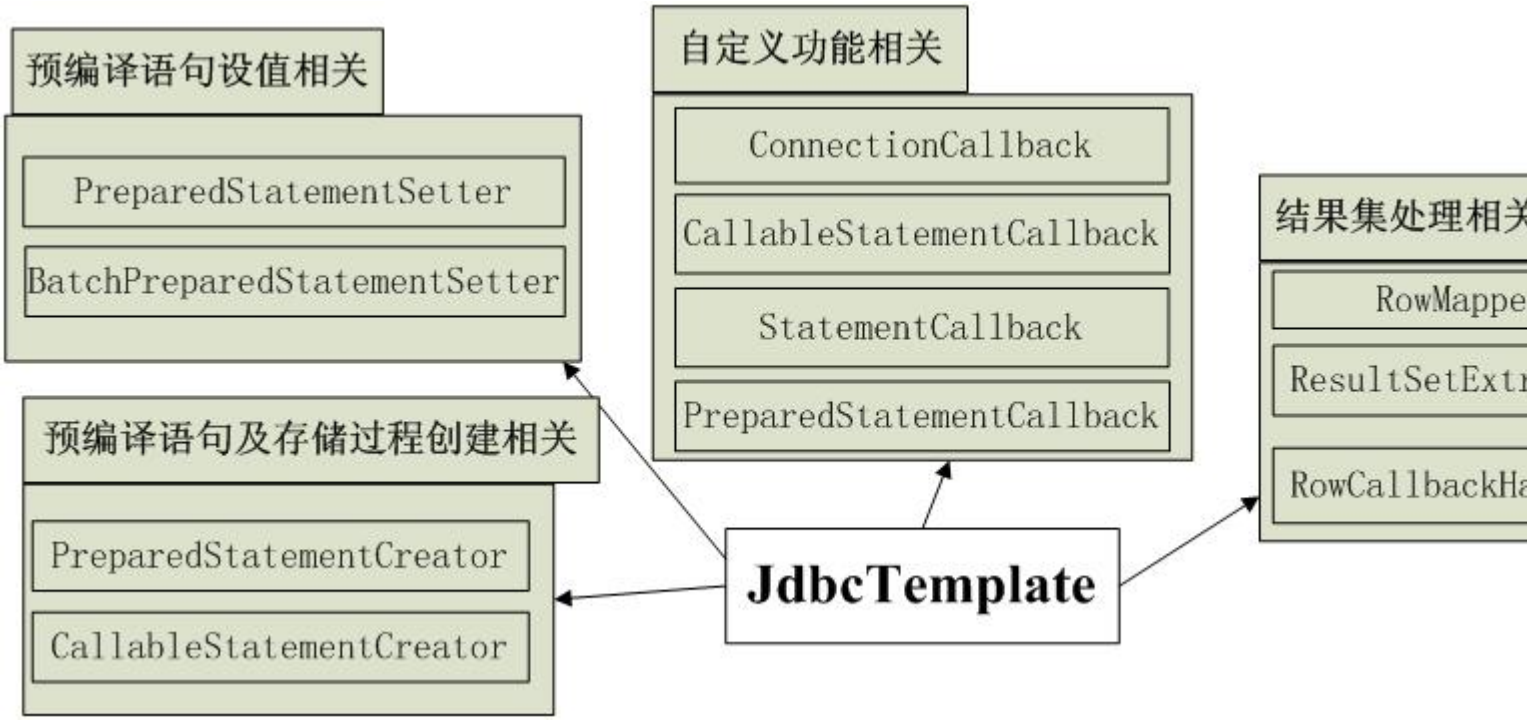
```
    }
}
```

```
jdbcTemplate.execute(new ConnectionCallback() {

            @Override
            public Object doInConnection(Connection conn) throws SQLException {
                //可变操作
                return null;
            }
        });
```

```
public interface PreparedStatementCallback {
    public Object doInPreparedStatement(PreparedStatement pstmt)
        throws  SQLException;
}
```

缺点：灵活但不通用

4、解决方案（Spring JDBC框架）

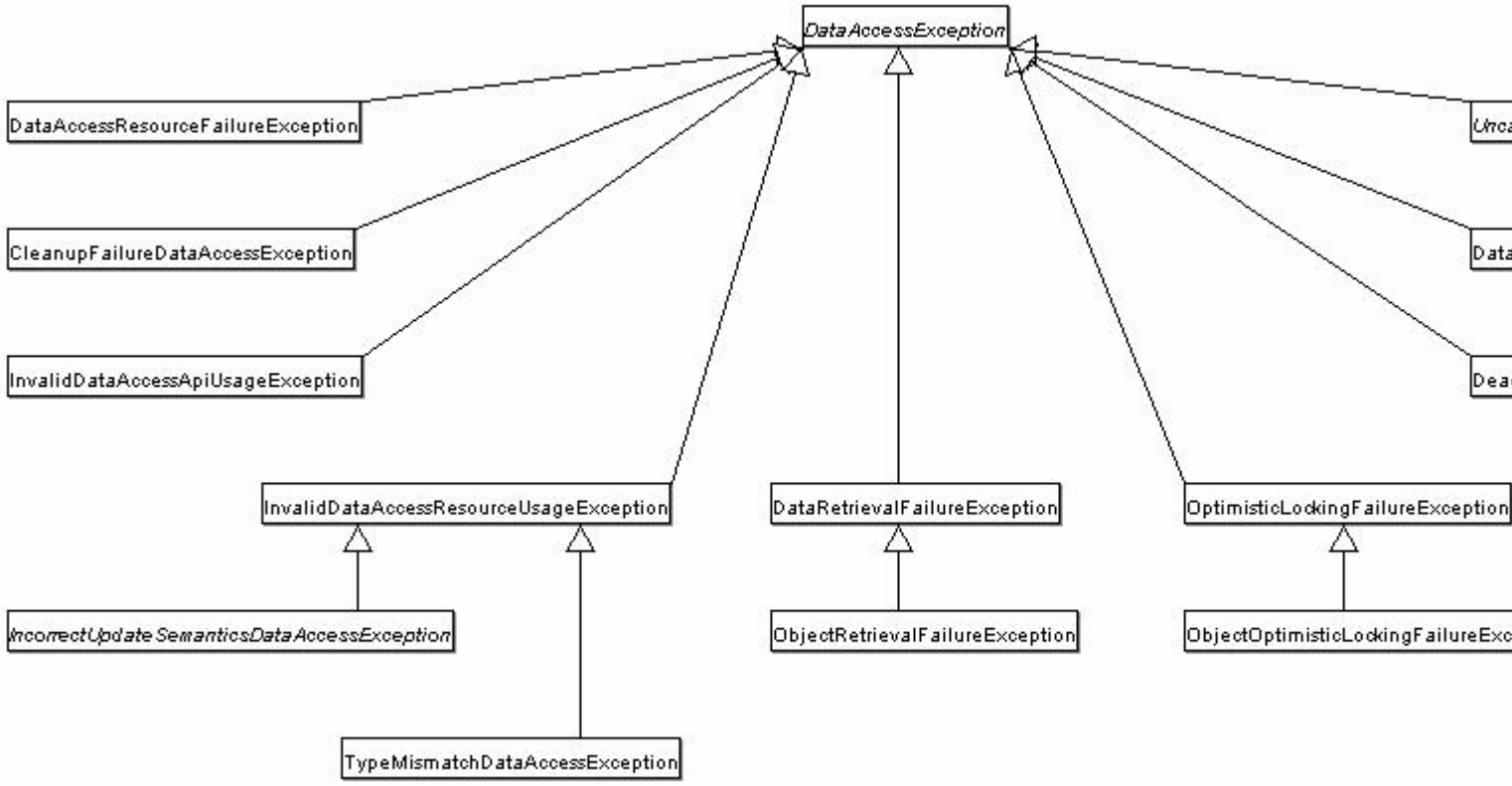


JdbcTemplate jdbcTemplate = **new** JdbcTemplate(ds);

5、异常问题

JDBC/ORM框架（如Hibernate）开发中异常处理有哪些缺点？ 不一致的异常类，如JDBC是SQLException 而hibernate是HibernateException，等。

缺点：不一致的异常体系，需要了解每个框架的异常含义 ，Spring的一致的异常体系（DataAccessException）



6、DAO访问问题

访问JDBC和访问Hibernate使用完全不同且根本不类似的API？

为了便于以一种一致的方式使用各种数据访问技术，如JDBC、JDO和Hibernate，

Spring提供了一套抽象DAO类供你扩展。

JdbcDaoSupport - JDBC数据访问对象的基类。

需要一个DataSource，同时为子类提供 JdbcTemplate。

HibernateDaoSupport - Hibernate数据访问对象的基类。

需要一个SessionFactory，同时为子类提供 HibernateTemplate。

JdoDaoSupport - JDO数据访问对象的基类。

需要设置一个PersistenceManagerFactory， 同时为子类提供JdoTemplate。

JpaDaoSupport - JPA数据访问对象的基类。

需要一个EntityManagerFactory，同时 为子类提供JpaTemplate。

一致的DAO抽象

Callback：定义可变行为，即不可预知行为(命令设计模式)

Template：提供模板支持，简化开发Support：提供对Template的支持，提供一组简便方法，并提供获取Template的方法

相关资料

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3

【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3

【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3

【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3

1.16 我对Spring 容器管理事务支持的总结

发表时间: 2012-09-27

之前发过几篇关于Spring的总结帖子 反响还不错，再把剩下的几篇发上来。共享给大家。

我对IoC/DI的理解

我对AOP的理解

我对SpringDAO层支持的总结

1、问题

```
Connection conn =
    DataSourceUtils.getConnection();
 //开启事务
conn.setAutoCommit(false);
try {
    Object retVal =
        callback.doInConnection(conn);
    conn.commit(); //提交事务
    return retVal;
}catch (Exception e) {
    conn.rollback();//回滚事务
    throw e;
}finally {
    conn.close();
}
```

```
Session session = null;
Transaction transaction = null;
try {
    session = factory.openSession();
```

```
    //开启事务
    transaction = session.beginTransaction();
    transation.begin();
    session.save(user);
    transaction.commit();//提交事务
} catch (Exception e) {
    e.printStackTrace();
    transaction.rollback();//回滚事务
    return false;
}finally{
    session.close();
}
```

缺点：不一致的事务管理，复杂 尤其当多个方法调用需要在同一个事务内时；

2、高层次解决方案

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

```
        //1.获取事务管理器
        PlatformTransactionManager txManager = (PlatformTransactionManager)
             ctx.getBean("txManager");
        //2.定义事务属性
        DefaultTransactionDefinition td = new DefaultTransactionDefinition();
        td.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
        //3开启事务,得到事务状态
        TransactionStatus status = txManager.getTransaction(td);
        try {
            //4.执行数据库操作
            System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
```

```
            //5、提交事务
            txManager.commit(status);

        }catch (Exception e) {
            //6、回滚事务
            txManager.rollback(status);
        }
```

重复代码太多，而且必须手工开启/释放（提交/回滚）事务。

3、高层次模板解决方案

```
//1.获取事务管理器
PlatformTransactionManager txManager = (PlatformTransactionManager)
       ctx.getBean("txManager");
//2、定义事务管理的模板
TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
//3.定义事务属性
transactionTemplate.
    setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
//4.回调，执行真正的数据库操作，如果需要返回值需要在回调里返回
transactionTemplate.execute(new TransactionCallback() {
    @Override
    public Object doInTransaction(TransactionStatus status) {
    //5.执行数据库操作
    System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
    return null;
    }
});
```

需要写模板代码，我们知道事务其实是一个切面，因此我们通过AOP来解决

4、AOP解决方案——一种声明时事务方案

nSpring框架提供了一致的事务管理抽象，这带来了以下好处：

1：为复杂的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO

2：支持声明式事务管理

3：提供比复杂的事务API（诸如JTA）更简单的、更易于使用的编程式事务管理API

4：非常好地整合Spring的各种数据访问抽象

实施事务的步骤

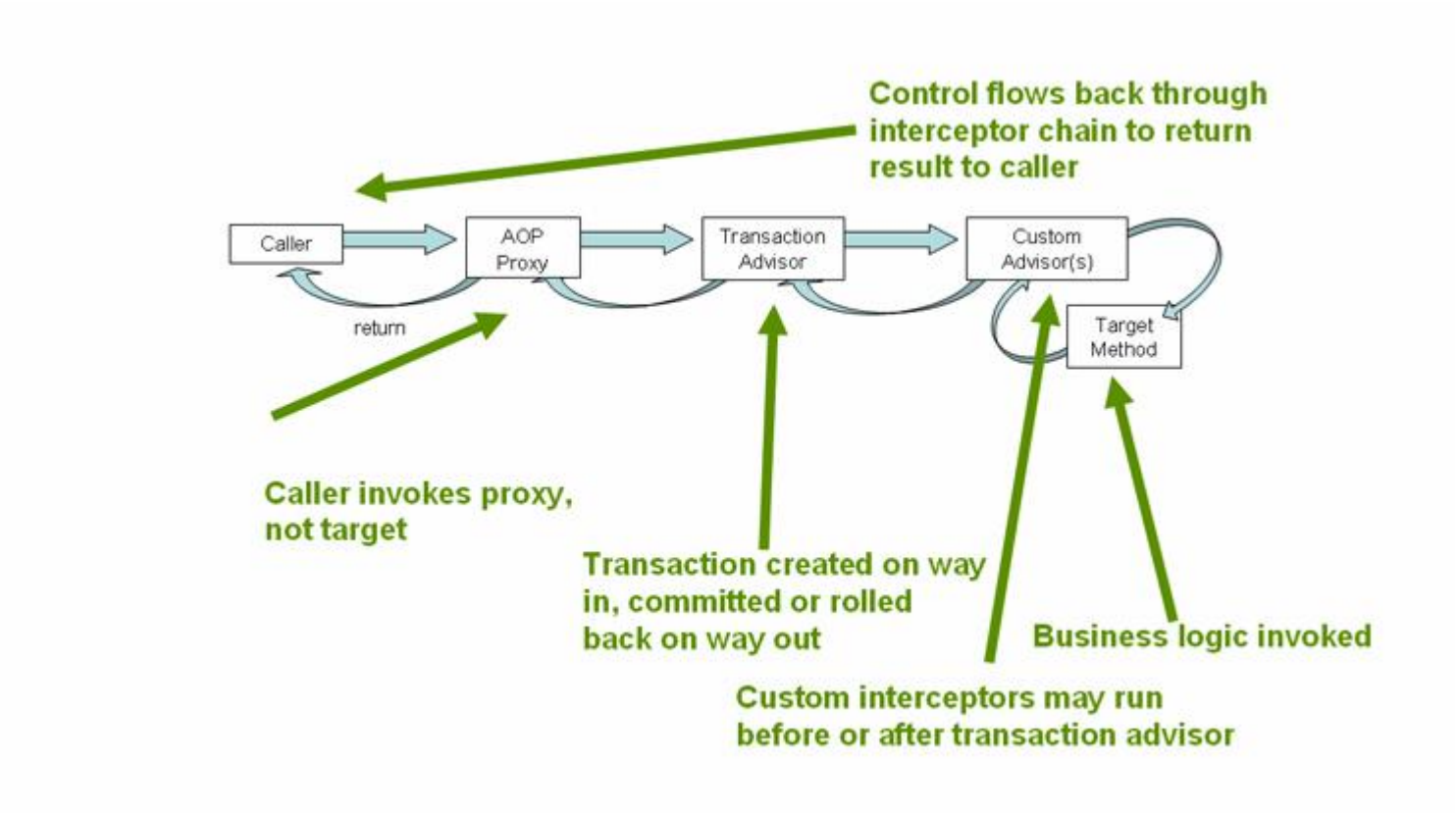
1、定义(资源)DataSource/SessionFactory……

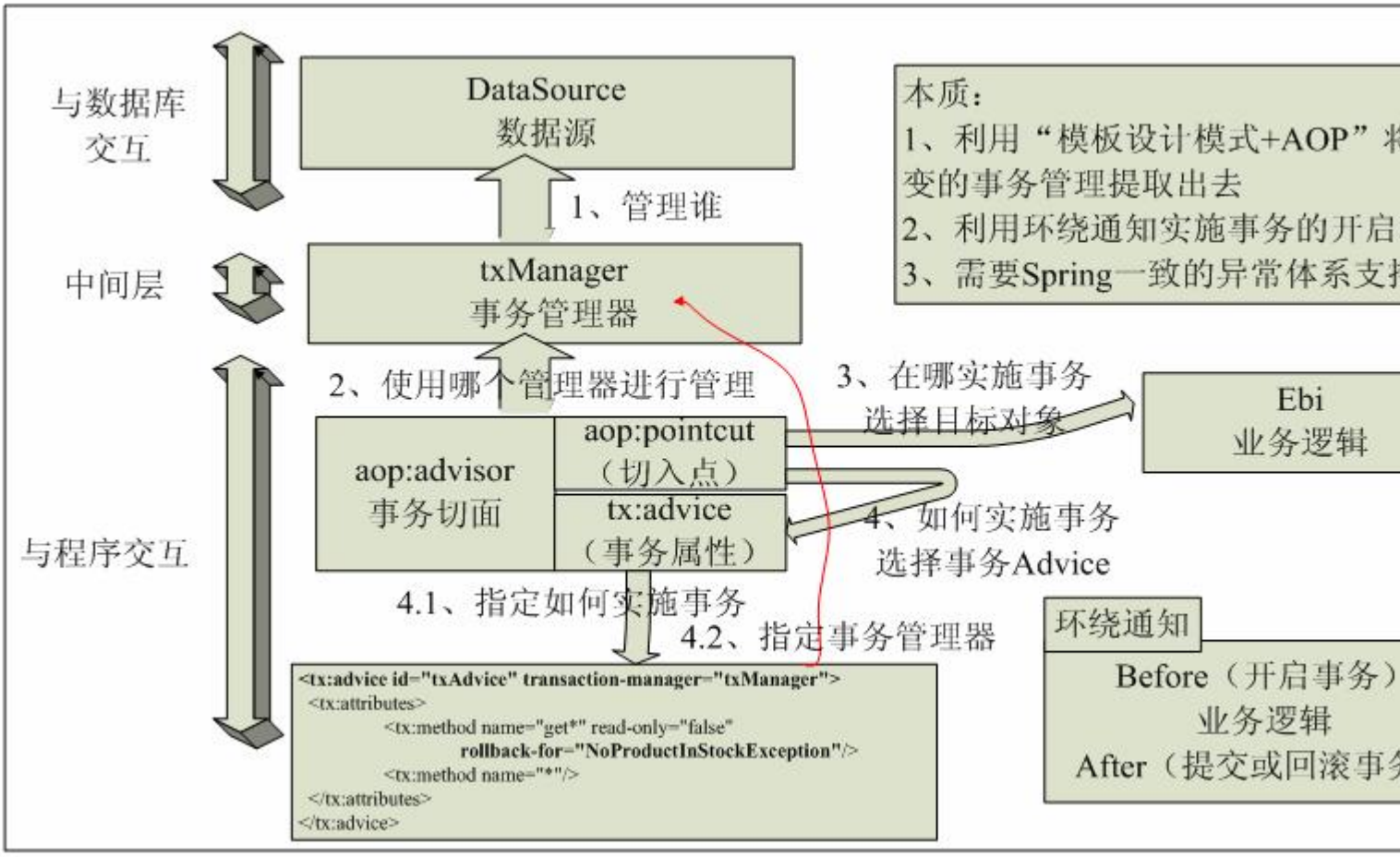
2、定义事务管理器（管理资源的事务）

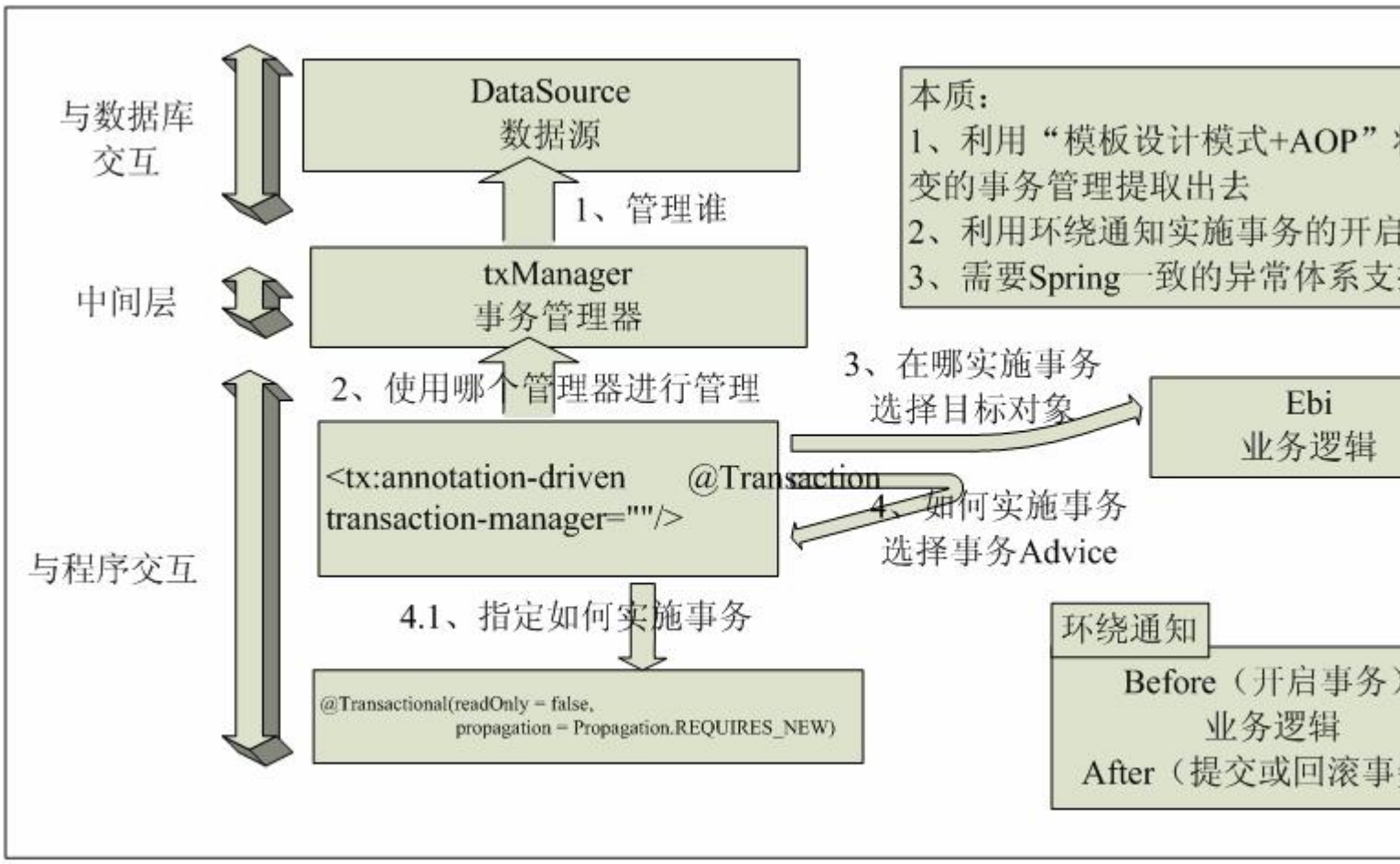
3、定义事务通知：定义了如何实施事务（实施事务的方法名和对应的事务属性），需要使用事务管理器管理事务，定义了如何选择目标对象的方法及实施的事务属性

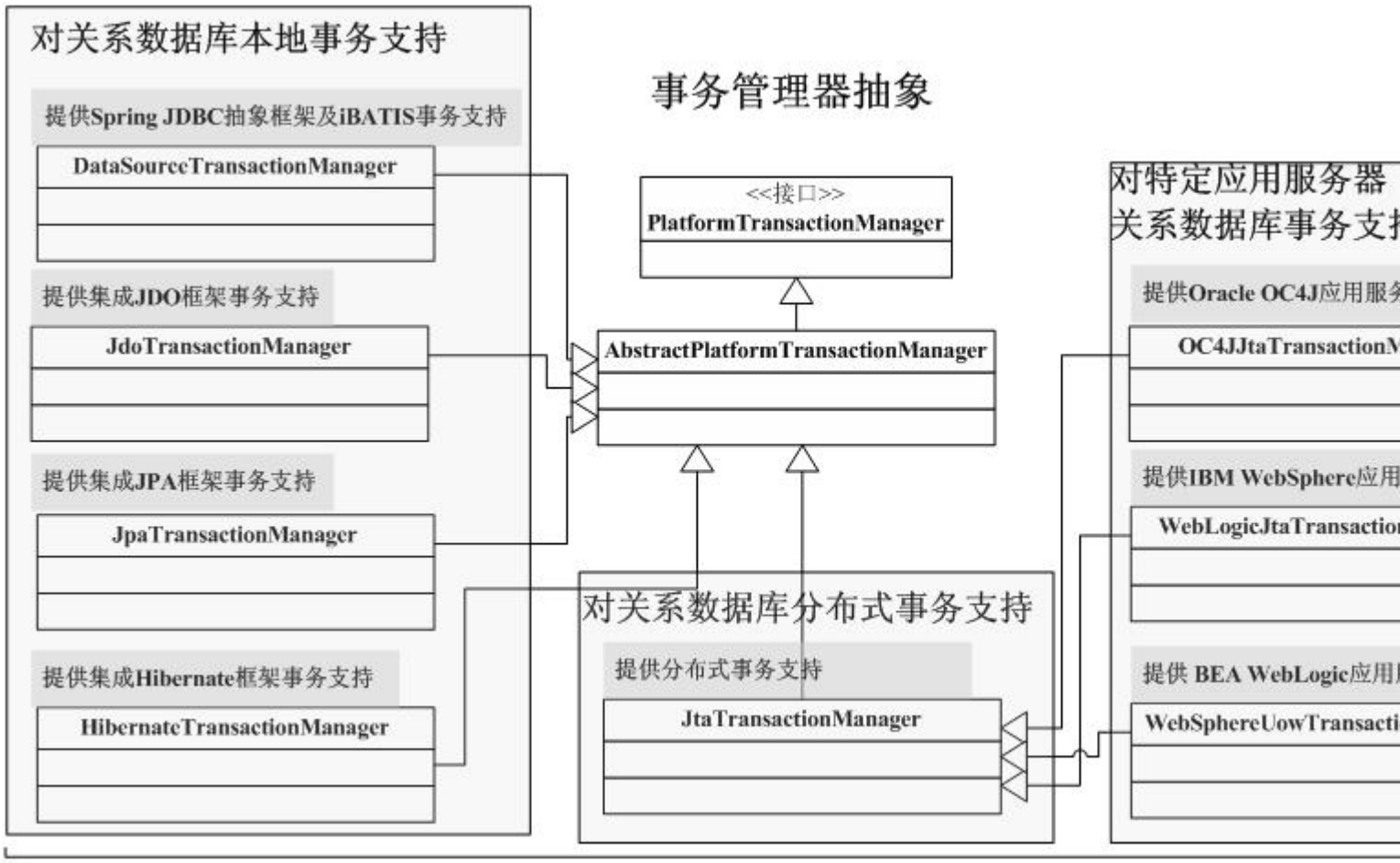
4、定义advisor（切入点和事务通知）：切入点选择需要实施事务的目标对象（一定是业务逻辑层）

5、Spring织入事务通知到目标对象（AOP代理）




与数据库交互
DataSource
数据源
1、管理谁
本质：
1、利用“模板设计模式+AOP”将
变的事务管理提取出去
2、利用环绕通知实施事务的开启
3、需要Spring一致的异常体系支
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
3、在哪实施事务
选择目标对象
Ebi
业务逻辑
aop:advisor
事务切面
aop:pointcut
（切入点）
tx:advice
（事务属性）
4、如何实施事务
选择事务Advice
与程序交互
4.1、指定如何实施事务
4.2、指定事务管理器
环绕通知
Before（开启事务）
业务逻辑
After（提交或回滚事
<tx:advice id="txAdvice" transaction-manager="txManager">
<tx:attributes>
<tx:method name="get*" read-only="false"
rollback-for="NoProductInStockException"/>
<tx:method name="*"/>
</tx:attributes>
</tx:advice>


与数据库交互
DataSource
数据源
1、管理谁
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
与程序交互
<tx:annotation-driven
transaction-manager=""/>
@Transaction
4.1、指定如何实施事务
@Transactional(readOnly = false,
propagation = Propagation.REQUIRES_NEW)
3、在哪实施事务
选择目标对象
Ebi
业务逻辑
4、如何实施事务
选择事务Advice
本质：
1、利用“模板设计模式+AOP”
变的事务管理提取出去
2、利用环绕通知实施事务的开启
3、需要Spring一致的异常体系支
环绕通知
Before（开启事务）
业务逻辑
After（提交或回滚事



更多学习资料：

【第九章】 Spring的事务 之 9.1 数据库事务概述 ——跟我学spring3

【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3

【第九章】 Spring的事务 之 9.3 编程式事务 ——跟我学spring3

【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

Spring事务处理时自我调用的解决方案及一些实现方式的风险

1.17 我对Spring 容器管理事务支持的总结

发表时间: 2012-09-27

之前发过几篇关于Spring的总结帖子 反响还不错，再把剩下的几篇发上来。共享给大家。

我对IoC/DI的理解

我对AOP的理解

我对SpringDAO层支持的总结

1、问题

```
Connection conn =
    DataSourceUtils.getConnection();
 //开启事务
conn.setAutoCommit(false);
try {
    Object retVal =
        callback.doInConnection(conn);
    conn.commit(); //提交事务
    return retVal;
}catch (Exception e) {
    conn.rollback();//回滚事务
    throw e;
}finally {
    conn.close();
}
```

```
Session session = null;
Transaction transaction = null;
try {
```

```
    session = factory.openSession();
    //开启事务
    transaction = session.beginTransaction();
    transation.begin();
    session.save(user);
    transaction.commit();//提交事务
} catch (Exception e) {
    e.printStackTrace();
    transaction.rollback();//回滚事务
    return false;
}finally{
    session.close();
}
```

缺点：不一致的事务管理，复杂 尤其当多个方法调用需要在同一个事务内时；

2、高层次解决方案

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

```
        //1.获取事务管理器
        PlatformTransactionManager txManager = (PlatformTransactionManager)
             ctx.getBean("txManager");
        //2.定义事务属性
        DefaultTransactionDefinition td = new DefaultTransactionDefinition();
        td.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
        //3开启事务,得到事务状态
        TransactionStatus status = txManager.getTransaction(td);
        try {
            //4.执行数据库操作
```

```
            System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
            //5、提交事务
            txManager.commit(status);

        }catch (Exception e) {
            //6、回滚事务
            txManager.rollback(status);
        }
```

重复代码太多，而且必须手工开启/释放（提交/回滚）事务。

3、高层次模板解决方案

```
//1.获取事务管理器
PlatformTransactionManager txManager = (PlatformTransactionManager)
       ctx.getBean("txManager");
//2、定义事务管理的模板
TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
//3.定义事务属性
transactionTemplate.
    setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
//4.回调，执行真正的数据库操作，如果需要返回值需要在回调里返回
transactionTemplate.execute(new TransactionCallback() {
    @Override
    public Object doInTransaction(TransactionStatus status) {
    //5.执行数据库操作
    System.out.println(jdbcTempate.queryForInt("select count(*) from tbl_doc"));
    return null;
    }
});
```

需要写模板代码，我们知道事务其实是一个切面，因此我们通过AOP来解决

4、AOP解决方案——一种声明时事务方案

nSpring框架提供了一致的事务管理抽象，这带来了以下好处：

1：为复杂的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO

2：支持声明式事务管理

3：提供比复杂的事务API（诸如JTA）更简单的、更易于使用的编程式事务管理API

4：非常好地整合Spring的各种数据访问抽象

实施事务的步骤

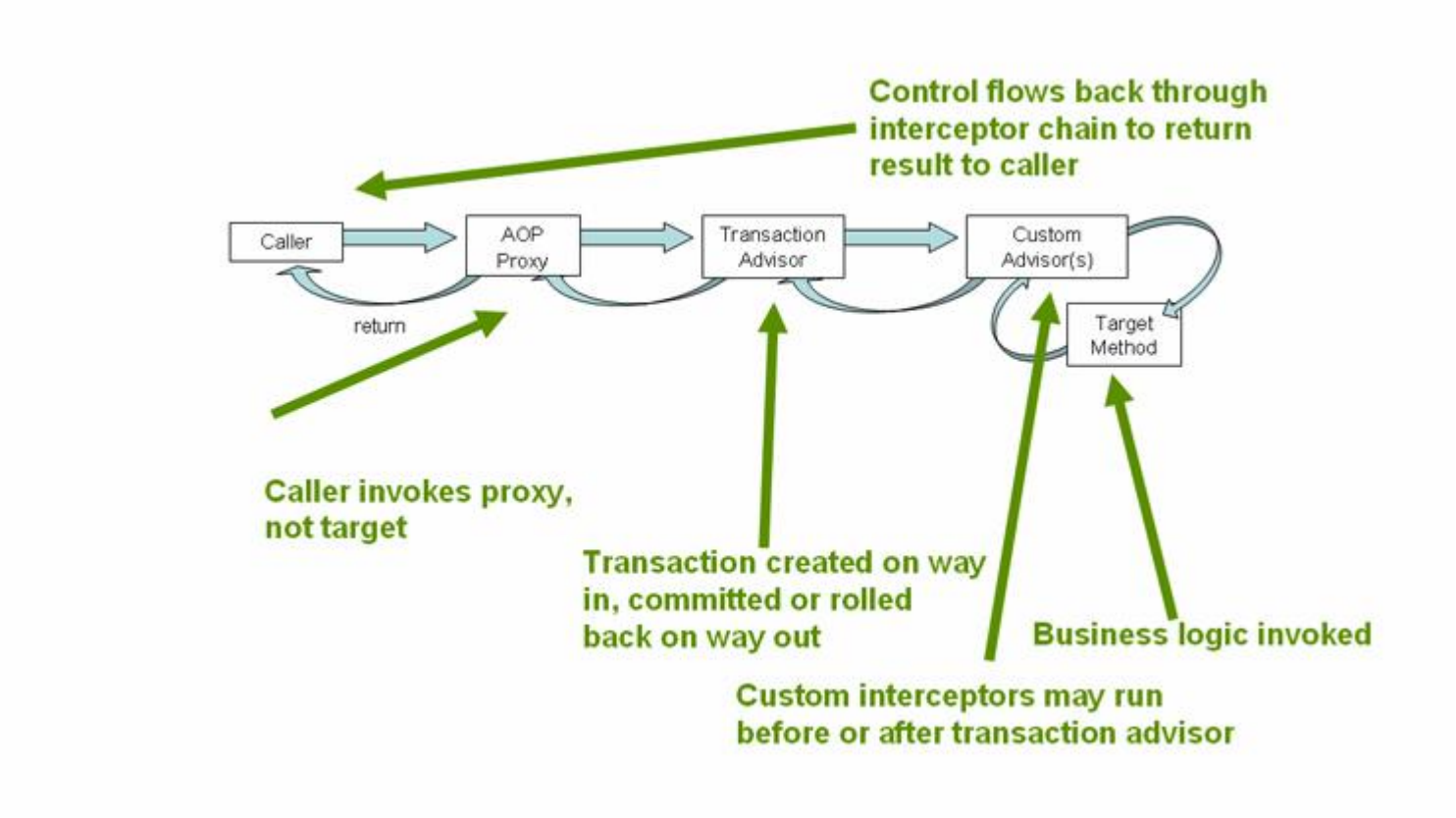
1、定义(资源)DataSource/SessionFactory……

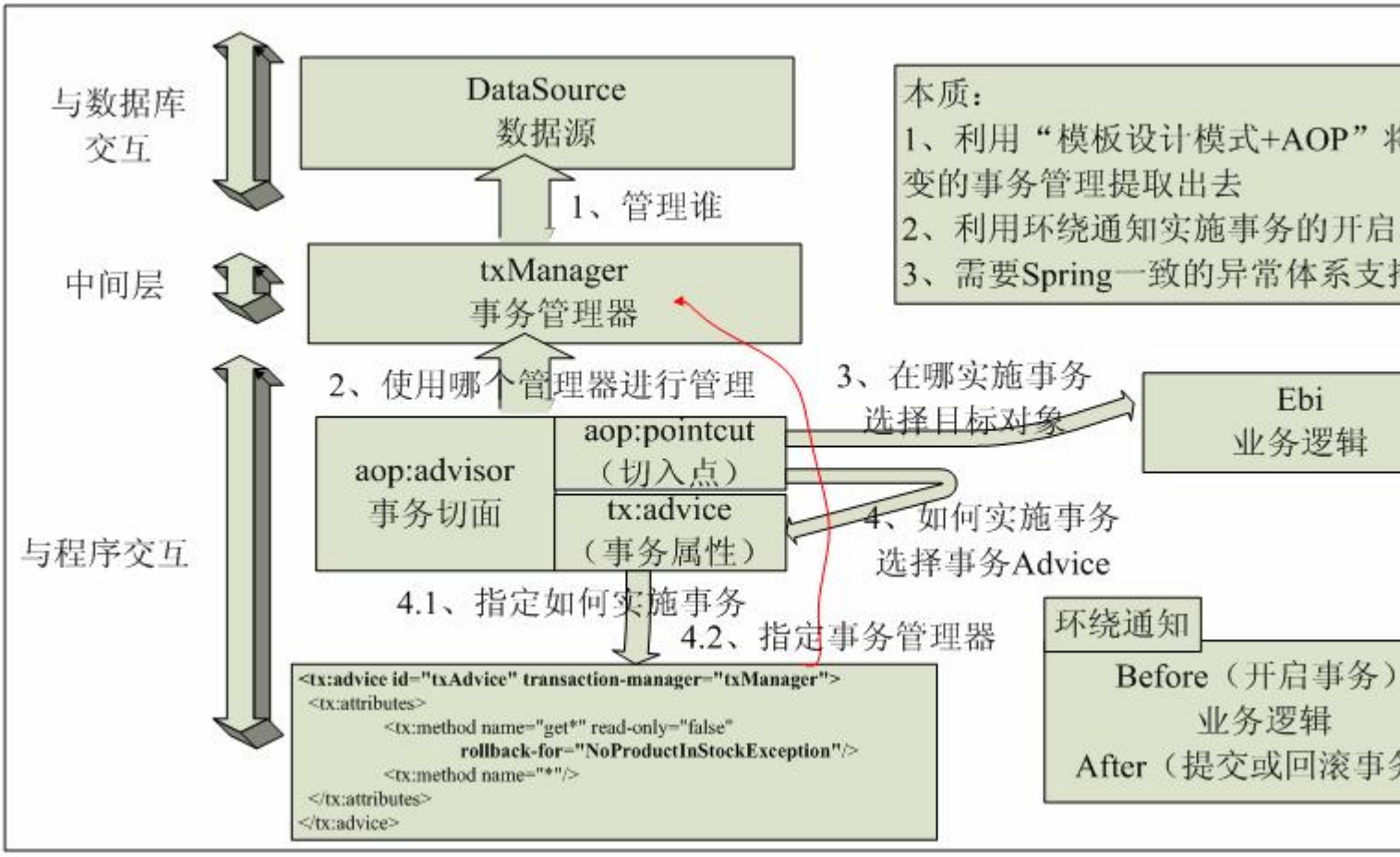
2、定义事务管理器（管理资源的事务）

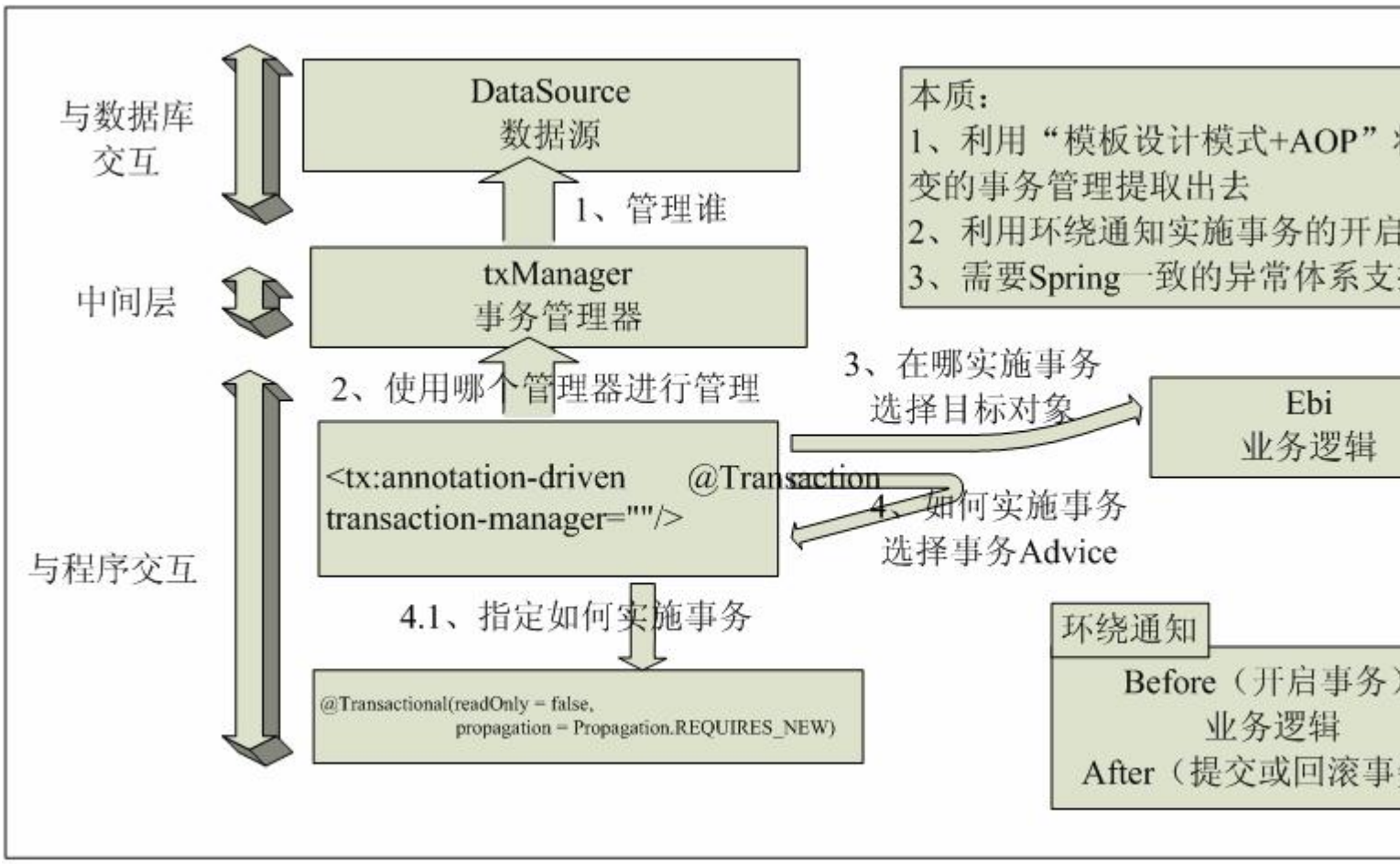
3、定义事务通知：定义了如何实施事务（实施事务的方法名和对应的事务属性），需要使用事务管理器管理事务，定义了如何选择目标对象的方法及实施的事务属性

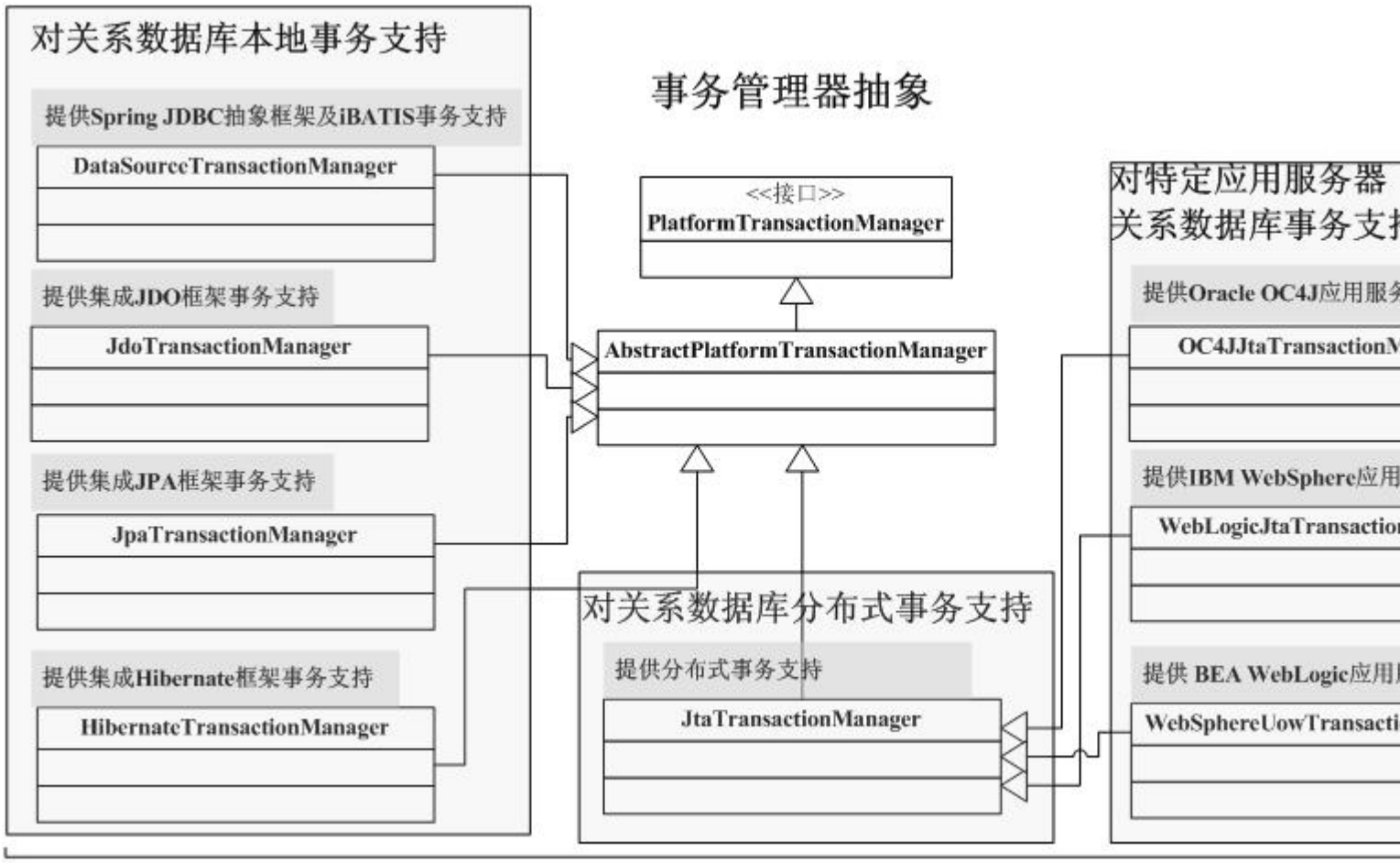
4、定义advisor（切入点和事务通知）：切入点选择需要实施事务的目标对象（一定是业务逻辑层）

5、Spring织入事务通知到目标对象（AOP代理）




与数据库交互
DataSource
数据源
1、管理谁
本质：
1、利用“模板设计模式+AOP”将
变的事务管理提取出去
2、利用环绕通知实施事务的开启
3、需要Spring一致的异常体系支
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
3、在哪实施事务
选择目标对象
Ebi
业务逻辑
aop:advisor
事务切面
aop:pointcut
（切入点）
tx:advice
（事务属性）
4、如何实施事务
选择事务Advice
与程序交互
4.1、指定如何实施事务
4.2、指定事务管理器
环绕通知
Before（开启事务）
业务逻辑
After（提交或回滚事
<tx:advice id="txAdvice" transaction-manager="txManager">
<tx:attributes>
<tx:method name="get*" read-only="false"
rollback-for="NoProductInStockException"/>
<tx:method name="*"/>
</tx:attributes>
</tx:advice>


与数据库交互
DataSource
数据源
1、管理谁
中间层
txManager
事务管理器
2、使用哪个管理器进行管理
与程序交互
<tx:annotation-driven
transaction-manager=""/>
@Transaction
3、在哪实施事务
选择目标对象
Ebi
业务逻辑
4、如何实施事务
选择事务Advice
4.1、指定如何实施事务
@Transactional(readOnly = false,
propagation = Propagation.REQUIRES_NEW)
环绕通知
Before（开启事务）
业务逻辑
After（提交或回滚事
本质：
1、利用“模板设计模式+AOP”
变的事务管理提取出去
2、利用环绕通知实施事务的开启
3、需要Spring一致的异常体系支



更多学习资料：

【第九章】 Spring的事务 之 9.1 数据库事务概述 ——跟我学spring3

【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3

【第九章】 Spring的事务 之 9.3 编程式事务 ——跟我学spring3

【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

Spring事务处理时自我调用的解决方案及一些实现方式的风险

1.18 不重复配置——利用Spring通用化配置

发表时间: 2012-10-16

还记得 如下这种配置吗：

1、struts2作用域：每一个Action我们必须设置scope为prototype 每次都做重复的配置，而且有时候忘记配置还会出现bug，想不想删掉它？

```
<bean id="**Action" class="***Action" scope="prototype">
```

2、在使用spring集成hibernate时，每次都必须注入sessionFactory，虽然可以用父子bean解决 但还是要写parent="abstractHibernateDao"之类的。

```
<bean id="***Dao" class="***DaoImpl">
    <property name="sessionFactory" ref="sessionFactory">
</bean>
```

受够了这种配置，得想法解决这个重复配置，怎么解决呢？

补充：

首先感谢downpour大哥的批评：

downpour 写道

> 在beans level上可以设置autowired的方式，sessionFactory这段配置从来就是可以省略的。
> prototype属性不能也不该省略，配置是给人看的，要是人看不懂就是垃圾。
> 综上所述，此方案纯粹脱裤子放屁多此一举。

1、sessionFactory注入问题：

如果是注解这个可以写个通用的基类就很容易搞定；

如果是XML 也可以通过在beans标签上使用 default-autowire="byName" default-autowire-candidates="*Dao" 也能解决问题，当我们通过类似于如下方式时，必须在每个相关的配置文件中都写上。

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>
      classpath:spring-common-config.xml,
      classpath:spring-budget-config.xml
   </param-value>
</context-param>
```

2、struts2 Action scope问题

如果使用StrutsPrepareAndExecuteFilter可以通过：

```
            <init-param>
                        <param-name>actionPackages</param-name>
                        <param-value>Action所在包前缀</param-value>
                </init-param>
```

scope会自动是prototype

使用我说的这种设置方式：我觉得因为只要会Struts2+Spring集成都知道struts2的Action是prototype，可以用；『prototype属性不能也不该省略，配置是给人看的，要是人看不懂就是垃圾。』这个是这么回事，需要仔细考虑下；当然我可以考虑在配置文件中加上注释 说明一下 告诉其他人是怎么回事。

另外这个功能我想可以改建为检查配置是否正确 类似于spring的依赖检查。欢迎大家拍砖。

思路：

在BeanFactory创建Bean之前查找所有我们需要通用化配置的Bean 然后修改BeanDefinition注入我们的通用数据就可以解决我们这个问题。

Spring提供了BeanFactoryPostProcessor扩展点，用于提供给我们修改BeanDefinition数据的。

还记得org.springframework.beans.factory.config.PropertyPlaceholderConfigurer吗？ 替换占位符数据，它就是一个BeanFactoryPostProcessor的实现。

好了思路有了，接下来我们实现一下吧：

1、XML配置方式

```
*
* 使用方法：<br/>
* <pre>
*  <bean class="cn.javass.common.spring.CommonConfigureProcessor">
      <property name="config">
          <map>
            <!--  aspectj表达式 选择所有Action结尾的Bean 注入scope数据 为 prototype  -->
            <entry key="cn.javass..*Action">
               <props>
                  <prop key="scope">prototype</prop>
```

```
            </props>
        </entry>
        <!-- aspectj表达式 选择所有的HibernateDaoSupport实现Bean 注入sessionFactory -->
        <entry key="org.springframework.orm.hibernate3.support.HibernateDaoSupport+">
          <props>
            <prop key="property-ref">sessionFactory=sessionFactory</prop>
          </props>
        </entry>
      </map>
    </property>
  </bean>
* </pre>
*
* 目前支持三种配置：
*  scope:注入作用域
*  property-ref:注入Bean引用 如之上的sessionFactory
*     propertyName=beanName
*  property-value:注入常量值
*     propertyName=常量
```

2、CommonConfigureProcessor源码

```
package cn.javass.common.spring;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Properties;

import org.aspectj.bridge.IMessageHandler;
import org.aspectj.weaver.ResolvedType;
import org.aspectj.weaver.World;
import org.aspectj.weaver.bcel.BcelWorld;
import org.aspectj.weaver.patterns.Bindings;
import org.aspectj.weaver.patterns.FormalBinding;
import org.aspectj.weaver.patterns.IScope;
import org.aspectj.weaver.patterns.PatternParser;
import org.aspectj.weaver.patterns.SimpleScope;
import org.aspectj.weaver.patterns.TypePattern;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.config.RuntimeBeanNameReference;
import org.springframework.beans.factory.config.RuntimeBeanReference;
import org.springframework.util.StringUtils;
/**
 *
 * 设置通用配置<br/>
 *
 * 使用方法：<br/>
 * <pre>
 *  <bean class="cn.javass.common.spring.CommonConfigureProcessor">
       <property name="config">
            <map>
```

```
            <!--  aspectj表达式 选择所有Action结尾的Bean 注入scope数据 为 prototype  -->
            <entry key="cn.javass..*Action">
                <props>
                    <prop key="scope">prototype</prop>
                </props>
            </entry>
            <!-- aspectj表达式 选择所有的HibernateDaoSupport实现Bean 注入sessionFactory -->
            <entry key="org.springframework.orm.hibernate3.support.HibernateDaoSupport+">
               <props>
                  <prop key="property-ref">sessionFactory=sessionFactory</prop>
               </props>
            </entry>
         </map>
      </property>
   </bean>
 * </pre>
 *
 * 目前支持三种配置：
 *  scope:注入作用域
 *  property-ref:注入Bean引用 如之上的sessionFactory
 *     propertyName=beanName
 *  property-value:注入常量值
 *     propertyName=常量
 *
 * @author Zhangkaitao
 * @version 1.0
 *
 */
public class CommonConfigureProcessor implements BeanFactoryPostProcessor {

    private Logger log = LoggerFactory.getLogger(CommonConfigureProcessor.class);

    private Map<String, Properties> config = new HashMap<String, Properties>();
    public void setConfig(Map<String, Properties> config) {
        this.config = config;
    }
```

```
@Override
public void postProcessBeanFactory(ConfigurableListableBeanFactory factory) throws BeansExc
    log.debug("apply common config start");
    for(Entry<String, Properties> entry : config.entrySet()) {
        String aspectjPattern = entry.getKey();
        Properties props = entry.getValue();

        List<BeanDefinition> bdList = findBeanDefinition(aspectjPattern, factory);

        apply(bdList, props);
    }
    log.debug("apply common config end");
}


private void apply(List<BeanDefinition> bdList, Properties props) {
    for(Entry<Object, Object> entry : props.entrySet()) {
        String key = (String) entry.getKey();
        String value = (String) entry.getValue();

        switch(SupportedConfig.keyToEnum(key)) {
            case scope :
                applyScope(bdList, value);
                break;
            case propertyRef:
                applyPropertyRef(bdList, value);
                break;
            case propertyValue:
                applyPropertyValue(bdList, value);
                break;
            default:
                throw new IllegalArgumentException(String.format("错误的配置：[%s]", key));
        }


    }
}
```

```
private void applyPropertyValue(List<BeanDefinition> bdList, String value) {
    for(BeanDefinition bd : bdList) {

        String propertyName = value.split("=")[0];
        String propertyValue = value.substring(propertyName.length()+1);
        bd.getPropertyValues().add(propertyName, propertyValue);

        log.debug("apply property value {} to {}", value, bd.getBeanClassName());
    }
}

private void applyPropertyRef(List<BeanDefinition> bdList, String value) {
    for(BeanDefinition bd : bdList) {

        String propertyName = value.split("=")[0];
        String propertyValue = value.substring(propertyName.length()+1);
        bd.getPropertyValues().addPropertyValue(propertyName, new RuntimeBeanReference(prop

        log.debug("apply property ref {} to {}", value, bd.getBeanClassName());
    }
}

private void applyScope(List<BeanDefinition> bdList, String value) {
    for(BeanDefinition bd : bdList) {
        bd.setScope(value);
        log.debug("apply scope {} to {}", value, bd.getBeanClassName());
    }
}

private List<BeanDefinition> findBeanDefinition(String aspectjPattern, ConfigurableListable
    List<BeanDefinition> bdList = new ArrayList<BeanDefinition>();

    for(String beanName : factory.getBeanDefinitionNames()) {
        BeanDefinition bd = factory.getBeanDefinition(beanName);
```

```
            if(matches(aspectjPattern, bd.getBeanClassName())) {
                bdList.add(bd);
            }

        }

        return bdList;
    }


    private boolean matches(String aspectjPattern, String beanClassName) {
        if(!StringUtils.hasLength(beanClassName)) {
            return false;
        }
        return new AspectJTypeMatcher(aspectjPattern).matches(beanClassName);
    }

    //支持的操作
    private static enum SupportedConfig {
        scope("scope"),
        propertyRef("property-ref"),
        propertyValue("property-value"),

        error("error"); //出错的情况

        private final String key;
        private SupportedConfig(String key) {
            this.key = key;
        }

        public static SupportedConfig keyToEnum(String key) {
            if(key == null) {
                return error;
            }
            for(SupportedConfig config : SupportedConfig.values()) {
                if(config.key.equals(key.trim())) {
                    return config;
```

```
            }
        }
        return error;
    }

}


public static interface TypeMatcher {
    public boolean matches(String className);
}

static class AspectJTypeMatcher implements TypeMatcher {
    private final World world;
    private final TypePattern typePattern;

    public AspectJTypeMatcher(String pattern) {

        this.world = new BcelWorld(Thread.currentThread().getContextClassLoader(), IMessage
        this.world.setBehaveInJava5Way(true);
        PatternParser patternParser = new PatternParser(pattern);
        TypePattern typePattern = patternParser.parseTypePattern();
        typePattern.resolve(this.world);
        IScope scope = new SimpleScope(this.world, new FormalBinding[0]);
        this.typePattern = typePattern.resolveBindings(scope, Bindings.NONE, false, false);
    }
    @Override
    public boolean matches(String className) {
        ResolvedType resolvedType = this.world.resolve(className);
        return this.typePattern.matchesStatically(resolvedType);
    }
}

public static void main(String[] args) {
    //System.out.println(new AspectJTypeMatcher("cn.javass..*Action").matches("cn.javass.te
    //System.out.println(new AspectJTypeMatcher("com.opensymphony.xwork2.ActionSupport+").m
```

```
    }
}
```

此类只实现基本的通用配置，欢迎大家提供想法并完善这个工具类。

1.19 @Value注入Properties 错误的使用案例

发表时间: 2012-10-23

场景：

需要注入Properties的value数据到Bean或方法参数。

准备数据：

```
    <bean id="props" class="java.util.Properties">
        <constructor-arg index="0">
           <props>
              <prop key="a">123</prop>
           </props>
        </constructor-arg>
    </bean>

    <bean id="map" class="java.util.HashMap">
        <constructor-arg index="0">
           <map>
              <entry key="a" value="234"/>
           </map>
        </constructor-arg>
    </bean>
```

失败的做法：

```
@Value("#{props['a']}")
    private String propsA;
```

此时无法获取props的a这个键对应的值。

正确的做法：

```
    @Value("#{props.getProperty('a')}")
    private String propsA;

    @Value("#{map['a']}")
    private String mapA;

    @RequestMapping(value="/value2")
    public String test2(@Value("#{props.getProperty('a')}") String propsA, @Value("#{map['a']}"
```

为什么？最后探讨。

最好的正确做法：

```
    <util:properties id="props2">
       <prop key="a">123</prop>
    </util:properties>
```

//该标签内部使用org.springframework.beans.factory.config.PropertiesFactoryBean FactoryBean实现；

```
@Value("#{props2['a']}") String props2A,
```

注入数据直接使用props[key]即可。

why ?

为什么如下方式不行：

```
<bean id="props" class="java.util.Properties">
    <constructor-arg index="0">
       <props>
          <prop key="a">123</prop>
       </props>
    </constructor-arg>
</bean>
```

而如下这种方式行呢？

```
<util:properties id="props2">
   <prop key="a">123</prop>
</util:properties>
```

原因很简单：API不熟造成的：

1、首先我们来看下Properties构造器定义：

javadoc 写道

public Properties(Properties defaults)创建一个带有指定默认值的空属性列表。

参数：
defaults - 默认值。

1）、构造器第一个参数是默认值；

2）、当我们使用getProperty(key) 时，首先查自己的prop，如果有直接返回，否则查defaults中的key。

javadoc 写道

> public String getProperty(String key)
> 用指定的键在此属性列表中搜索属性。如果在此属性列表中未找到该键，则接着递归检查默认属性列表及其默认值。如果未找到属性，则此方法返回 null。

3）、原因到此很明显了，此时我们使用#{props.getProperty('a')} 其实是查的defaults

4）、Properties本身继承了Hashtable，其实是一种错误的用法，造成了我们现在的我们现在的问题，get(key)只查自己，而getProperty会先查自己 再查defaults。

2、而使用<util:properties id="props2"> 本身是一个FactoryBean，帮我们创建没有我们真实需要的properties。有兴趣可以看下源代码。

总结：

1、API不熟造成了我们错误的使用；

2、Properties继承Hashtable本身就是错误的继承，造成了两套不同的API。

1.20 @Value注入Properties 使用错误的案例

发表时间: 2012-10-23

最近有朋友问@Value注入Properties数据注入不进去，接下来我就分析一下为什么。

场景：

需要注入Properties的value数据到Bean或方法参数。

准备数据：

```
    <bean id="props" class="java.util.Properties">
        <constructor-arg index="0">
            <props>
                <prop key="a">123</prop>
            </props>
        </constructor-arg>
    </bean>

    <bean id="map" class="java.util.HashMap">
        <constructor-arg index="0">
            <map>
                <entry key="a" value="234"/>
            </map>
        </constructor-arg>
    </bean>
```

失败的做法：

```
@Value("#{props['a']}")
    private String propsA;
```

此时无法获取props的a这个键对应的值。

正确的做法：

```
    @Value("#{props.getProperty('a')}")
    private String propsA;

    @Value("#{map['a']}")
    private String mapA;

    @RequestMapping(value="/value2")
    public String test2(@Value("#{props.getProperty('a')}") String propsA, @Value("#{map['a']}'
```

为什么？最后探讨。

最好的正确做法：

```
    <util:properties id="props2">
       <prop key="a">123</prop>
    </util:properties>
```

//该标签内部使用org.springframework.beans.factory.config.PropertiesFactoryBean FactoryBean实现；

```
@Value("#{props2['a']}") String props2A,
```

注入数据直接使用props[key]即可。

why？

为什么如下方式不行：

```
    <bean id="props" class="java.util.Properties">
        <constructor-arg index="0">
```

```
        <props>
           <prop key="a">123</prop>
        </props>
     </constructor-arg>
  </bean>
```

而如下这种方式行呢？

```
  <util:properties id="props2">
     <prop key="a">123</prop>
  </util:properties>
```

原因很简单：API不熟造成的：

1、首先我们来看下Properties构造器定义：

javadoc 写道

public Properties(Properties defaults)创建一个带有指定默认值的空属性列表。

参数：
defaults - 默认值。

1）、构造器第一个参数是默认值；

2）、当我们使用getProperty(key) 时，首先查自己的prop，如果有直接返回，否则查defaults中的key。

javadoc 写道

public String getProperty(String key)
用指定的键在此属性列表中搜索属性。如果在此属性列表中未找到该键，则接着递归检查默认属性列表及其默认值。如果未找到属性，则此方法返回 null。

3）、原因到此很明显了，此时我们使用#{props.getProperty('a')} 其实是查的defaults

4）、Properties本身继承了Hashtable，其实是一种错误的用法，造成了我们现在的我们现在的问题，get(key)只查自己，而getProperty会先查自己 再查defaults。

2、而使用<util:properties id="props2"> 本身是一个FactoryBean，帮我们创建没有我们真实需要的properties。有兴趣可以看下源代码。

总结：

1、API不熟造成了错误理解（以为类似于如集合框架是做拷贝）；

2、Properties继承Hashtable本身就是一种错误的继承，形成两套不同的API。

1.21 扩展SpringMVC以支持更精准的数据绑定

发表时间: 2012-11-06

问题描述：

springMVC 数据绑定 多个对象 如何准确绑定?

```
<form>

    <input name="student.name" value="Kate" />

    <input name="student.type" value="自费" />

    <input name="teacher.name" value="Gavin" />

    <input name="teacher.level" value="2" />

</form>
```

```
@RequestMapping("/school.do")

public String school(Student student, Teacher teacher) {

     return "school";

}
```

如果还是想刚才的jsp那些写表单，是不能封装参数的，必须把“student.”和“teacher.”去掉，但是这样封装就不能准确封装了。

我们都知道struts2默认就是这种方案，这是因为struts2采用了OGNL，并通过栈（根对象）进行操作的，而且栈中默认有action实例，所以很自然的没有这种问题。

springmvc不同，没有根对象的概念，而且本身很难来解决这个问题，因此大家在使用时最好避免这种方式或者使用类似于struts1的FormBean组合对象来解决。

解决方案：

扩展spring的HandlerMethodArgumentResolver以支持自定义的数据绑定方式。

1、请下载附件的代码，放到工程中；

2、在RequestMappingHandlerAdapter添加自定义HandlerMethodArgumentResolver Bean；

```
    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAd
       <!--线程安全的访问session-->
        <property name="synchronizeOnSession" value="true"/>
        <property name="customArgumentResolvers">
           <list>
              <bean class="cn.javass.spring.mvc.method.annotation.RequestJsonParamMethodArgume
              <bean class="cn.javass.spring.mvc.method.annotation.FormModelMethodArgumentResol
           </list>
        </property>
    </bean>
```

//customArgumentResolvers用于注入自定义的参数解析器，此处我们注了FormModelMethodArgumentResolver；FormModelMethodArgumentResolver我直接修改的org.springframework.web.servlet.mvc.method.annotation.ServletModelAttributeMethodProcessor；

3、使用方式

```
public String user(@FormModel("student") Student student, @FormModel("teacher") Teacher teache
```

4、测试控制器

```
package cn.javass.chapter6.web.controller.formmodel;

import java.util.Arrays;
import java.util.List;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import cn.javass.chapter6.model.UserModel;
import cn.javass.spring.mvc.bind.annotation.FormModel;
import cn.javass.spring.mvc.util.MapWapper;


@Controller
@RequestMapping("/formmodel")
public class FormModelController {

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/user?user.username=zhang&user.pas
    @RequestMapping("/user/{user.realname}")
    public String user(@FormModel("user") UserModel user) {
        System.out.println(user);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/array1?array[0]=zhang&array[1]=li
    @RequestMapping("/array1")
    public String array1(@FormModel("array") String[] array) {
        System.out.println(Arrays.toString(array));
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/array2?array[0].username=zhang&ar
    @RequestMapping("/array2")
    public String array2(@FormModel("array") UserModel[] array) {
        System.out.println(Arrays.toString(array));
        return "redirect:/success";
    }


    //ok   http://localhost:9080/springmvc-chapter6/formmodel/list1?list[0]=123&list[1]=234
    @RequestMapping("/list1")
    public String list1(@FormModel("list") List<Integer> list) {
```

```
        System.out.println(list);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/list2?list[0].username=zhang&list
    @RequestMapping("/list2")
    public String list2(@FormModel("list") List<UserModel> list) {
        System.out.println(list);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/map1?map['0']=123&map["1"]=234
    @RequestMapping("/map1")
    public String map1(@FormModel("map") MapWapper<String, Integer> map) {
        System.out.println(map);
        return "redirect:/success";
    }
  //ok   http://localhost:9080/springmvc-chapter6/formmodel/map2?map['0'].password=123&map['0']
    @RequestMapping("/map2")
    public String map2(@FormModel("map") MapWapper<Integer, UserModel> map) {
        System.out.println(map);
        return "redirect:/success";
    }
}
```

支持的spring版本：

spring 3.1.x，暂不支持3.0。

支持绑定的数据：

模型、集合、数组、MapWapper（Map的一个包装器，通过getInnerMap获取真实Map）

缺点：

spring自定义的参数解析器会放在默认解析器之后，不能指定order，因此如果我们@FormModel("map") Map map，此map会变成Model（请参考http://jinnianshilongnian.iteye.com/blog/1698916 第六部分、Model Map ModelMap），希望未来的版本支持自定义顺序来解决这个问题；此处我们使用MapWapper解决，可以通过MapWapper.getInnerMap()拿到我们需要的Map

其他方案：

[SpringMVC]修改源码使之能够更加智能的自动装配request请求参数.（不建议修改源代码解决）

@**rainsoft** 也给出了类似的方案， http://www.iteye.com/topic/1124433#2357830

欢迎大家反馈问题，我会及时修正。

下一个扩展： 绑定请求参数（JSON字符串，如 deptIds=[{"deptId":4,"isPrimary":true}] ）到 模型对象。

1.22 扩展SpringMVC以支持更精准的数据绑定1

发表时间: 2012-11-06

最新版请点击查看FormModelMethodArgumentResolver.java

问题描述：

springMVC 数据绑定 多个对象 如何准确绑定?

```
<form>

    <input name="student.name" value="Kate" />

    <input name="student.type" value="自费" />

    <input name="teacher.name" value="Gavin" />

    <input name="teacher.level" value="2" />

</form>
```

```
@RequestMapping("/school.do")

public String school(Student student, Teacher teacher) {

     return "school";

}
```

如果还是想刚才的jsp那些写表单，是不能封装参数的，必须把“student.”和“teacher.”去掉，但是这样封装就不能准确封装了。

这个问题最近老是有人问，所以写一个扩展很容易解决这个问题，springmvc和spring一样，预留的扩展点足够多。

我们都知道struts2默认就是这种方案，这是因为struts2采用了OGNL，并通过栈（根对象）进行操作的，而且栈中默认有action实例，所以很自然的没有这种问题。

springmvc不同，没有根对象的概念，而且本身很难来解决这个问题，因此大家在使用时最好避免这种方式或者使用类似于struts1的FormBean组合对象来解决。

解决方案：

扩展spring的HandlerMethodArgumentResolver以支持自定义的数据绑定方式。

1、请下载附件的代码，放到工程中；

2、在RequestMappingHandlerAdapter添加自定义HandlerMethodArgumentResolver Bean；

```
    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAd
       <!--线程安全的访问session-->
        <property name="synchronizeOnSession" value="true"/>
        <property name="customArgumentResolvers">
           <list>
              <bean class="cn.javass.spring.mvc.method.annotation.RequestJsonParamMethodArgume
              <bean class="cn.javass.spring.mvc.method.annotation.FormModelMethodArgumentResol
           </list>
        </property>
    </bean>
```

//customArgumentResolvers用于注入自定义的参数解析器，此处我们注了FormModelMethodArgumentResolver；FormModelMethodArgumentResolver我直接修改的org.springframework.web.servlet.mvc.method.annotation.ServletModelAttributeMethodProcessor；

3、使用方式

```
public String user(@FormModel("student") Student student, @FormModel("teacher") Teacher teache
```

4、测试控制器

```
package cn.javass.chapter6.web.controller.formmodel;

import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import cn.javass.chapter6.model.UserModel;
import cn.javass.spring.mvc.bind.annotation.FormModel;
import cn.javass.spring.mvc.util.MapWapper;


@Controller
@RequestMapping("/formmodel")
public class FormModelController {

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/user?user.username=zhang&user.pas
    @RequestMapping("/user/{user.realname}")
    public String user(@FormModel("user") UserModel user) {
        System.out.println(user);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/array1?array[0]=zhang&array[1]=li
    @RequestMapping("/array1")
    public String array1(@FormModel("array") String[] array) {
        System.out.println(Arrays.toString(array));
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/array2?array[0].username=zhang&ar
    @RequestMapping("/array2")
    public String array2(@FormModel("array") UserModel[] array) {
        System.out.println(Arrays.toString(array));
        return "redirect:/success";
    }
```

```
    //ok   http://localhost:9080/springmvc-chapter6/formmodel/list1?list[0]=123&list[1]=234
    @RequestMapping("/list1")
    public String list1(@FormModel("list") List<Integer> list) {
        System.out.println(list);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/list2?list[0].username=zhang&list
    @RequestMapping("/list2")
    public String list2(@FormModel("list") List<UserModel> list) {
        System.out.println(list);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/formmodel/map1?map['0']=123&map["1"]=234
    @RequestMapping("/map1")
    public String map1(@FormModel("map") MapWapper<String, Integer> map) {
        System.out.println(map);
        return "redirect:/success";
    }
  //ok   http://localhost:9080/springmvc-chapter6/formmodel/map2?map['0'].password=123&map['0']
    @RequestMapping("/map2")
    public String map2(@FormModel("map") MapWapper<Integer, UserModel> map) {
        System.out.println(map);
        return "redirect:/success";
    }
}
```

具体使用可以下载之前springmvc第六章源代码http://jinnianshilongnian.iteye.com/blog/1683388

将附件中的FormModel.rar解压放到src下进行测试。

支持的spring版本：

springmvc 3.1.x，暂不支持3.0。为什么不支持呢？springmvc 3.1 和 3.0 从架构上发生了变化，而且springmvc3.1更容易扩展。

支持绑定的数据：

模型、集合、数组、MapWapper（Map的一个包装器，通过getInnerMap获取真实Map）

缺点：

spring自定义的参数解析器会放在默认解析器之后，不能指定order，因此如果我们@FormModel("map") Map map，此map会变成Model（请参考http://jinnianshilongnian.iteye.com/blog/1698916 第六部分、Model Map ModelMap），希望未来的版本支持自定义顺序来解决这个问题；此处我们使用MapWapper解决，可以通过MapWapper.getInnerMap()拿到我们需要的Map

其他方案：

[SpringMVC]修改源码使之能够更加智能的自动装配request请求参数.（不建议修改源代码解决）

@**rainsoft** 也给出了类似的方案， http://www.iteye.com/topic/1124433#2357830

如果你使用的是mvc:annotation-driven，请这样配置

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
            <bean class="com.sishuok.es.common.web.bind.method.annotation.FormModelMethodArgume
       </mvc:argument-resolvers>
</mvc:annotation-driven>
```

欢迎大家反馈问题，我会及时修正。

下一个扩展： 绑定请求参数（JSON字符串，如 deptIds=[{"deptId":4,"isPrimary":true}] ）到 模型对象。

附件下载:

- FormModel.rar (10.7 KB)
- dl.iteye.com/topics/download/8059687f-5d0e-306d-acc3-d3f1aa774036

1.23 扩展SpringMVC以支持绑定JSON格式的请求参数

发表时间: 2012-11-07

上一篇：《扩展SpringMVC以支持更精准的数据绑定》

问题描述：

```
你好，对于如下的json数据，springmvc的数据绑定该如何做？
accessionDate   2012-11-21
deptIds [{"deptId":4,"isPrimary":true}]
email   ewer@dsfd.com
fax     3423432
gender  true
其实就是我前台一表单提交的数据，extjs form提交的，关键在于deptIds的映射，数组不知怎么解决，还有就是springmvc的数据绑定如
```

如请求参数是deptIds={"deptId":4,"isPrimary":true}] 是一个json数组，此时需要绑定为一个对象，该如何实现呢？

解决方案：

类似于之前写的《扩展SpringMVC以支持更精准的数据绑定》，扩展spring的HandlerMethodArgumentResolver以支持自定义的数据绑定方式。

1、请下载附件的代码，放到工程中；

2、在RequestMappingHandlerAdapter添加自定义HandlerMethodArgumentResolver Bean；

```
    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
       <!--线程安全的访问session-->
        <property name="synchronizeOnSession" value="true"/>
        <property name="customArgumentResolvers">
           <list>
```

```
            <bean class="cn.javass.spring.mvc.method.annotation.RequestJsonParamMethodArgumentResolver"/>
            <bean class="cn.javass.spring.mvc.method.annotation.FormModelMethodArgumentResolver"/>
        </list>
    </property>
</bean>
```

//customArgumentResolvers用于注入自定义的参数解析器，此处我们注入了RequestJsonParamMethodArgumentResolver?

3、使用方式

```
@RequestMapping("/list")
public String list(@RequestJsonParam("list") List<Integer> list)
```

4、测试控制器

```
package cn.javass.chapter6.web.controller.jsonparam;

import java.util.Arrays;
import java.util.List;
import java.util.Set;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import cn.javass.chapter6.model.DataBinderTestModel;
import cn.javass.chapter6.model.UserModel;
import cn.javass.spring.mvc.bind.annotation.RequestJsonParam;
import cn.javass.spring.mvc.util.MapWapper;



@Controller
@RequestMapping("/jsonparam")
public class JsonParamController {

```

```
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/list?list=[1,2,34]
//fail http://localhost:9080/springmvc-chapter6/jsonparam/list?list=[1,2,a]
@RequestMapping("/list")
public String list(@RequestJsonParam("list") List<Integer> list) {
    System.out.println(list);
    return "redirect:/success";
}

//ok   http://localhost:9080/springmvc-chapter6/jsonparam/set?set=[1,2,34]
//fail http://localhost:9080/springmvc-chapter6/jsonparam/set?set=[1,2,a]
@RequestMapping("/set")
public String set(@RequestJsonParam("set") Set<Integer> set) {
    System.out.println(set);
    return "redirect:/success";
}


//ok   http://localhost:9080/springmvc-chapter6/jsonparam/array?array=[1,2,3]
//fail http://localhost:9080/springmvc-chapter6/jsonparam/array?array=[1,2,a]
@RequestMapping("/array")
public String list(@RequestJsonParam("array") int[] array) {
    System.out.println(Arrays.toString(array));
    return "redirect:/success";
}


//ok   http://localhost:9080/springmvc-chapter6/jsonparam/map?map={"a":1, "b":2}
//fail http://localhost:9080/springmvc-chapter6/jsonparam/map?map={"a":1, "b":a}
@RequestMapping("/map")
public String map(@RequestJsonParam(value = "map", required=false) MapWapper<String, Integer> map) {
    System.out.println(map);
    return "redirect:/success";
}


//UserModel[]
```

```
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/array2?array=[{"username":"123"},{"username":"234
@RequestMapping("/array2")
public String array2(@RequestJsonParam(value = "array") UserModel[] array) {
    System.out.println(Arrays.toString(array));
    return "redirect:/success";
}

//List<UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/list2?list=[{"username":"123"},{"username":"234"
@RequestMapping("/list2")
public String list2(@RequestJsonParam(value = "list") List<UserModel> list) {
    System.out.println(list);
    return "redirect:/success";
}

//Set<UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/set2?set=[{"username":"123"},{"username":"234"}]
@RequestMapping("/set2")
public String set2(@RequestJsonParam(value = "set") Set<UserModel> set) {
    System.out.println(set);
    return "redirect:/success";
}

//Map<String, UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/map2?map={"a":{"username":"123"},"b":{"username"
//暂不支持 Map<UserModel, UserModel>
@RequestMapping("/map2")
public String map2(@RequestJsonParam(value = "map") MapWapper<String, UserModel> map) {
    System.out.println(map);
    return "redirect:/success";
}



//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model1?model={"username":123,"password":234,"rea
//没有realname1
//fail http://localhost:9080/springmvc-chapter6/jsonparam/model1?model={"username":123,"password":234,"rea
```

```
    @RequestMapping("/model1")
    public String model1(@RequestJsonParam(value = "model", required=true) UserModel user) {
        System.out.println(user);
        return "redirect:/success";
    }

    //ENUM
    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"state":"normal"}
    //List<基本类型>
    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"hobbyList":["film", "music"]}
    //Map<基本类型，基本类型>
    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"map":{"key":"value", "a":"b"}}
    @RequestMapping("/model2")
    public String model2(@RequestJsonParam(value = "model", required=true) DataBinderTestModel model) {
        System.out.println(model);
        return "redirect:/success";
    }

    //List<UserModel>
    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/model3?model={"userList":[{"username":"1"},{"user
    //Map<String , UserModel>
    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/model3?model={"userMap":{"1":{"username":"1"},"2"

    //暂不支持 类似于 Map<UserModel, UserModel> 形式
    @RequestMapping("/model3")
    public String model3(@RequestJsonParam(value = "model") DataBinderTestModel model) {
        System.out.println(model);
        return "redirect:/success";
    }

}
```

支持的spring版本：

springmvc 3.0 和 3.1.x。

支持绑定的数据：

模型、集合、数组、MapWapper（Map的一个包装器，通过getInnerMap获取真实Map）

暂时不支持JSR-303数据校验.

缺点：

和《扩展SpringMVC以支持更精准的数据绑定》一样的缺点。

spring自定义的参数解析器会放在默认解析器之后，不能指定order，因此如果我们@FormModel("map") Map map，此map会变成Model（请参考http://jinnianshilongnian.iteye.com/blog/1698916 第六部分、Model Map ModelMap），希望未来的版本支持自定义顺序来解决这个问题；此处我们使用MapWapper解决，可以通过MapWapper.getInnerMap()拿到我们需要的Map

欢迎大家反馈问题，我会及时修正。

支持一下博主：----------

http://blog.51cto.com/contest2012/2058573

附件下载:

- jsonparam.rar (12 KB)
- dl.iteye.com/topics/download/1e903c1e-4a21-3512-9c25-d30e23fdf792

1.24 扩展SpringMVC以支持绑定JSON格式的请求参数

发表时间: 2012-11-08

上一篇：《扩展SpringMVC以支持更精准的数据绑定》

此方案是把请求参数（JSON字符串）绑定到java对象，，@RequestBody是绑定内容体到java对象的。

问题描述：

```
你好，对于如下的json数据，springmvc的数据绑定该如何做？
accessionDate   2012-11-21
deptIds [{"deptId":4,"isPrimary":true}]
email   ewer@dsfd.com
fax     3423432
gender  true
其实就是我前台一表单提交的数据，extjs form提交的，关键在于deptIds的映射，数组不知怎么解决，还有就是springmvc的数据绑定
```

如请求参数是deptIds={"deptId":4,"isPrimary":true}] 是一个json数组，此时需要绑定为一个对象，该如何实现呢？

解决方案：

类似于之前写的《扩展SpringMVC以支持更精准的数据绑定》，扩展spring的HandlerMethodArgumentResolver以支持自定义的数据绑定方式。

1、请下载附件的代码，放到工程中；

2、在RequestMappingHandlerAdapter添加自定义HandlerMethodArgumentResolver Bean；

```
    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
       <!--线程安全的访问session-->
        <property name="synchronizeOnSession" value="true"/>
        <property name="customArgumentResolvers">
           <list>
              <bean class="cn.javass.spring.mvc.method.annotation.RequestJsonParamMethodArgumentResolver"/>
              <bean class="cn.javass.spring.mvc.method.annotation.FormModelMethodArgumentResolver"/>
           </list>
        </property>
    </bean>
```

//customArgumentResolvers用于注入自定义的参数解析器，此处我们注入了RequestJsonParamMethodArgumentResolver?

3、使用方式

```
@RequestMapping("/list")
public String list(@RequestJsonParam("list") List<Integer> list)
```

4、测试控制器

```
package cn.javass.chapter6.web.controller.jsonparam;

import java.util.Arrays;
import java.util.List;
import java.util.Set;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import cn.javass.chapter6.model.DataBinderTestModel;
import cn.javass.chapter6.model.UserModel;
import cn.javass.spring.mvc.bind.annotation.RequestJsonParam;
import cn.javass.spring.mvc.util.MapWapper;
```

```
@Controller
@RequestMapping("/jsonparam")
public class JsonParamController {

    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/list?list=[1,2,34]
    //fail http://localhost:9080/springmvc-chapter6/jsonparam/list?list=[1,2,a]
    @RequestMapping("/list")
    public String list(@RequestJsonParam("list") List<Integer> list) {
        System.out.println(list);
        return "redirect:/success";
    }

    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/set?set=[1,2,34]
    //fail http://localhost:9080/springmvc-chapter6/jsonparam/set?set=[1,2,a]
    @RequestMapping("/set")
    public String set(@RequestJsonParam("set") Set<Integer> set) {
        System.out.println(set);
        return "redirect:/success";
    }


    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/array?array=[1,2,3]
    //fail http://localhost:9080/springmvc-chapter6/jsonparam/array?array=[1,2,a]
    @RequestMapping("/array")
    public String list(@RequestJsonParam("array") int[] array) {
        System.out.println(Arrays.toString(array));
        return "redirect:/success";
    }


    //ok   http://localhost:9080/springmvc-chapter6/jsonparam/map?map={"a":1, "b":2}
    //fail http://localhost:9080/springmvc-chapter6/jsonparam/map?map={"a":1, "b":a}
    @RequestMapping("/map")
    public String map(@RequestJsonParam(value = "map", required=false) MapWapper<String, Integer> map) {
        System.out.println(map);
```

```
    return "redirect:/success";
}



//UserModel[]
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/array2?array=[{"username":"123"},{"username":"234
@RequestMapping("/array2")
public String array2(@RequestJsonParam(value = "array") UserModel[] array) {
    System.out.println(Arrays.toString(array));
    return "redirect:/success";
}

//List<UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/list2?list=[{"username":"123"},{"username":"234"}
@RequestMapping("/list2")
public String list2(@RequestJsonParam(value = "list") List<UserModel> list) {
    System.out.println(list);
    return "redirect:/success";
}

//Set<UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/set2?set=[{"username":"123"},{"username":"234"}]
@RequestMapping("/set2")
public String set2(@RequestJsonParam(value = "set") Set<UserModel> set) {
    System.out.println(set);
    return "redirect:/success";
}

//Map<String, UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/map2?map={"a":{"username":"123"},"b":{"username"
//暂不支持 Map<UserModel, UserModel>
@RequestMapping("/map2")
public String map2(@RequestJsonParam(value = "map") MapWapper<String, UserModel> map) {
    System.out.println(map);
    return "redirect:/success";
}
```

```
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model1?model={"username":123,"password":234,"rea
//没有realname1
//fail http://localhost:9080/springmvc-chapter6/jsonparam/model1?model={"username":123,"password":234,"rea
@RequestMapping("/model1")
public String model1(@RequestJsonParam(value = "model", required=true) UserModel user) {
    System.out.println(user);
    return "redirect:/success";
}

//ENUM
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"state":"normal"}
//List<基本类型>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"hobbyList":["film", "music"]}
//Map<基本类型，基本类型>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model2?model={"map":{"key":"value", "a":"b"}}
@RequestMapping("/model2")
public String model2(@RequestJsonParam(value = "model", required=true) DataBinderTestModel model) {
    System.out.println(model);
    return "redirect:/success";
}

//List<UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model3?model={"userList":[{"username":"1"},{"use
//Map<String , UserModel>
//ok   http://localhost:9080/springmvc-chapter6/jsonparam/model3?model={"userMap":{"1":{"username":"1"},"2
//暂不支持 类似于 Map<UserModel, UserModel> 形式
@RequestMapping("/model3")
public String model3(@RequestJsonParam(value = "model") DataBinderTestModel model) {
    System.out.println(model);
    return "redirect:/success";
}
```

```
}
```

支持的spring版本：

springmvc 3.0 和 3.1.x。

支持绑定的数据：

模型、集合、数组、MapWapper（Map的一个包装器，通过getInnerMap获取真实Map）

暂时不支持JSR-303数据校验.

缺点：

1、和《扩展SpringMVC以支持更精准的数据绑定》一样的缺点。

spring自定义的参数解析器会放在默认解析器之后，不能指定order，因此如果我们@FormModel("map") Map map，此map会变成Model（请参考http://jinnianshilongnian.iteye.com/blog/1698916 第六部分、Model Map ModelMap），希望未来的版本支持自定义顺序来解决这个问题；此处我们使用MapWapper解决，可以通过MapWapper.getInnerMap()拿到我们需要的Map。

2、支持的jackson版本是1.4.2，版本比较老，spring3.2里程碑版已升级到2.x。

3、暂时没有提供JSR303数据校验。

欢迎大家反馈问题，我会及时修正。

附件下载:

- JsonParam.rar (12 KB)
- dl.iteye.com/topics/download/b1086a93-e45c-38ce-8bc9-427966b309ca

1.25 在应用层通过spring特性解决数据库读写分离

发表时间: 2012-11-08

如何配置mysql数据库的主从？

单机配置mysql主从：http://my.oschina.net/god/blog/496

常见的解决数据库读写分离有两种方案

1、应用层

http://neoremind.net/2011/06/spring实现数据库读写分离

目前的一些解决方案需要在程序中手动指定数据源，比较麻烦，后边我会通过AOP思想来解决这个问题。

2、中间件

mysql-proxy：http://hi.baidu.com/geshuai2008/item/0ded5389c685645f850fab07

Amoeba for MySQL：http://www.iteye.com/topic/188598和http://www.iteye.com/topic/1113437

此处我们介绍一种在应用层的解决方案，通过spring动态数据源和AOP来解决数据库的读写分离。

该方案目前已经在一个互联网项目中使用了，而且可以很好的工作。

该方案目前支持

一读多写；当写时默认读操作到写库、当写时强制读操作到读库。

考虑未来支持

读库负载均衡、读库故障转移等。

使用场景

不想引入中间件，想在应用层解决读写分离，可以考虑这个方案；

建议数据访问层使用jdbc、ibatis，不建议hibernate。

优势

应用层解决，不引入额外中间件；

在应用层支持『当写时默认读操作到写库』，这样如果我们采用这种方案，在写操作后读数据直接从写库拿，不会产生数据复制的延迟问题；

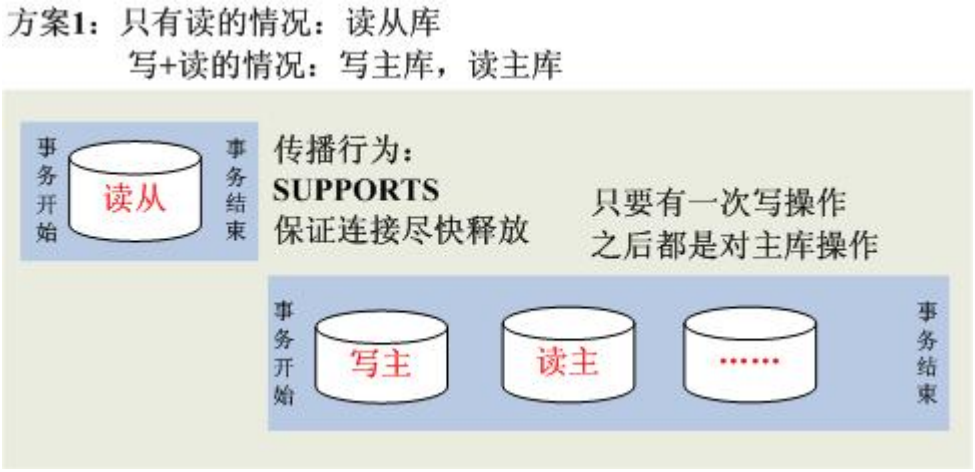
应用层解决读写分离，理论支持任意数据库。

缺点

1、不支持@Transactional注解事务，此方案要求所有读方法必须是read-only=true，因此如果是@Transactional，这样就要求在每一个读方法头上加@Transactional 且readOnly属性=true，相当麻烦。:oops:

2、必须按照配置约定进行配置，不够灵活。

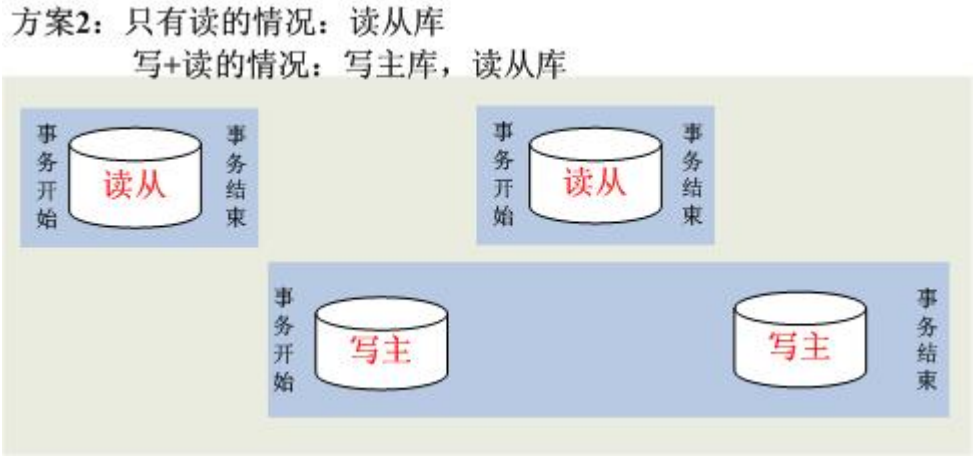
两种方案



方案1：当只有读操作的时候，直接操作读库（从库）；

当在写事务（即写主库）中读时，也是读主库（即参与到主库操作），这样的优势是可以防止写完后可能读不到刚才写的数据；

此方案其实是使用事务传播行为为：SUPPORTS解决的。



方案2：当只有读操作的时候，直接操作读库（从库）；

当在写事务（即写主库）中读时，强制走从库，即先暂停写事务，开启读（读从库），然后恢复写事务。

此方案其实是使用事务传播行为为：NOT_SUPPORTS解决的。

核心组件

cn.javass.common.datasource.ReadWriteDataSource：读写分离的动态数据源，类似于AbstractRoutingDataSource，具体参考javadoc；

cn.javass.common.datasource.ReadWriteDataSourceDecision：读写库选择的决策者，具体参考javadoc；

cn.javass.common.datasource.ReadWriteDataSourceProcessor：此类实现了两个职责（为了减少类的数量将两个功能合并到一起了）：读/写动态数据库选择处理器、通过AOP切面实现读/写选择，具体参考javadoc。

具体配置

1、数据源配置

1.1、写库配置

```
        <bean id="writeDataSource" class="org.logicalcobwebs.proxool.ProxoolDataSource">
              <property name="alias" value="writeDataSource"/>
              <property name="driver" value="${write.connection.driver_class}" />
              <property name="driverUrl" value="${write.connection.url}" />
              <property name="user" value="${write.connection.username}" />
              <property name="password" value="${write.connection.password}" />
              <property name="maximumConnectionCount" value="${write.proxool.maximum.connecti
              <property name="minimumConnectionCount" value="${write.proxool.minimum.connecti
              <property name="statistics" value="${write.proxool.statistics}" />
              <property name="simultaneousBuildThrottle" value="${write.proxool.simultaneous.
      </bean>
```

1.2、读库配置

```
    <bean id="readDataSource1" class="org.logicalcobwebs.proxool.ProxoolDataSource">
        <property name="alias" value="readDataSource"/>
        <property name="driver" value="${read.connection.driver_class}" />
        <property name="driverUrl" value="${read.connection.url}" />
```

```
        <property name="user" value="${read.connection.username}" />
        <property name="password" value="${read.connection.password}" />
        <property name="maximumConnectionCount" value="${read.proxool.maximum.connection.count}
        <property name="minimumConnectionCount" value="${read.proxool.minimum.connection.count}
        <property name="statistics" value="${read.proxool.statistics}" />
        <property name="simultaneousBuildThrottle" value="${read.proxool.simultaneous.build.thr
    </bean>
```

1.3、读写动态库配置

通过writeDataSource指定写库，通过`readDataSourceMap`指定从库列表，从库列表默认通过顺序轮询来使用读库，具体参考`javadoc`；

```
    <bean id="readWriteDataSource" class="cn.javass.common.datasource.ReadWriteDataSource">
        <property name="writeDataSource" ref="writeDataSource"/>
        <property name="readDataSourceMap">
            <map>
                <entry key="readDataSource1" value-ref="readDataSource1"/>
                <entry key="readDataSource2" value-ref="readDataSource1"/>
                <entry key="readDataSource3" value-ref="readDataSource1"/>
                <entry key="readDataSource4" value-ref="readDataSource1"/>
            </map>
        </property>
    </bean>
```

2、XML事务属性配置

所以读方法必须是read-only（必须，以此来判断是否是读方法）。

```
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="save*" propagation="REQUIRED" />
            <tx:method name="add*" propagation="REQUIRED" />
            <tx:method name="create*" propagation="REQUIRED" />
            <tx:method name="insert*" propagation="REQUIRED" />
            <tx:method name="update*" propagation="REQUIRED" />
```

```
            <tx:method name="merge*" propagation="REQUIRED" />
            <tx:method name="del*" propagation="REQUIRED" />
            <tx:method name="remove*" propagation="REQUIRED" />

            <tx:method name="put*" read-only="true"/>
            <tx:method name="query*" read-only="true"/>
            <tx:method name="use*" read-only="true"/>
            <tx:method name="get*" read-only="true" />
            <tx:method name="count*" read-only="true" />
            <tx:method name="find*" read-only="true" />
            <tx:method name="list*" read-only="true" />

            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
```

3、事务管理器

事务管理器管理的是readWriteDataSource

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="readWriteDataSource"/>
    </bean>
```

4、读/写动态数据库选择处理器

根据之前的txAdvice配置的事务属性决定是读/写，具体参考javadoc；

forceChoiceReadWhenWrite：用于确定在如果目前是写（即开启了事务），下一步如果是读，是直接参与到写库进行读，还是强制从读库读，具体参考javadoc；

```
    <bean id="readWriteDataSourceTransactionProcessor" class="cn.javass.common.datasource.Readw
       <property name="forceChoiceReadWhenWrite" value="false"/>
    </bean>
```

5、事务切面和读/写库选择切面

```
<aop:config expose-proxy="true">
    <!-- 只对业务逻辑层实施事务 -->
    <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>

    <!-- 通过AOP切面实现读/写库选择 -->
    <aop:aspect order="-2147483648" ref="readWriteDataSourceTransactionProcessor">
       <aop:around pointcut-ref="txPointcut" method="determineReadOrWriteDB"/>
    </aop:aspect>
</aop:config>
```

1、事务切面一般横切业务逻辑层；

2、此处我们使用readWriteDataSourceTransactionProcessor的通过AOP切面实现读/写库选择功能，order=Integer.MIN_VALUE(即最高的优先级)，从而保证在操作事务之前已经决定了使用读/写库。

6、测试用例

只要配置好事务属性（通过read-only=true指定读方法）即可，其他选择读/写库的操作都交给readWriteDataSourceTransactionProcessor完成。

可以参考附件的：

cn.javass.readwrite.ReadWriteDBTestWithForceChoiceReadOnWriteFalse

cn.javass.readwrite.ReadWriteDBTestWithNoForceChoiceReadOnWriteTrue

可以下载附件的代码进行测试，具体选择主/从可以参考日志输出。

暂不想支持@Transactional注解式事务。

PS：欢迎拍砖指正。

附件下载:

- master-slaver.rar (6.7 MB)
- dl.iteye.com/topics/download/cd0a906a-11ba-320d-9503-6252c3fe58c5

1.26 context:component-scan扫描使用上的容易忽略的use-default-filters

发表时间: 2013-01-05

问题

如下方式可以成功扫描到@Controller注解的Bean，不会扫描@Service/@Repository的Bean。正确

```
 <context:component-scan base-package="org.bdp.system.test.controller">
     <context:include-filter type="annotation" expression="org.springframework.stereotype.Contr
</context:component-scan>
```

但是如下方式，不仅仅扫描@Controller，还扫描@Service/@Repository?Bean?????????

```
 <context:component-scan base-package="org.bdp">
     <context:include-filter type="annotation" expression="org.springframework.stereotype.Contr
</context:component-scan>
```

这个尤其在springmvc+spring+hibernate等集成时最容易出问题的地，最典型的错误就是：

事务不起作用

这是什么问题呢？

分析

1、<context:component-scan>会交给org.springframework.context.config.ContextNamespaceHandler处理；

```
registerBeanDefinitionParser("component-scan", new ComponentScanBeanDefinitionParser());
```

2、ComponentScanBeanDefinitionParser会读取配置文件信息并组装成org.springframework.context.annotation.ClassPathBeanDefinitionScanner进行处理；

3、如果没有配置<context:component-scan>的use-default-filters属性，则默认为true，在创建ClassPathBeanDefinitionScanner时会根据use-default-filters是否为true来调用如下代码：

```
    protected void registerDefaultFilters() {
                this.includeFilters.add(new AnnotationTypeFilter(Component.class));
                ClassLoader cl = ClassPathScanningCandidateComponentProvider.class.getClassLoad
                try {
                        this.includeFilters.add(new AnnotationTypeFilter(
                                        ((Class<? extends Annotation>) cl.loadClass("javax.annc
                        logger.info("JSR-250 'javax.annotation.ManagedBean' found and supported
                }
                catch (ClassNotFoundException ex) {
                        // JSR-250 1.1 API (as included in Java EE 6) not available - simply sk
                }
                try {
                        this.includeFilters.add(new AnnotationTypeFilter(
                                        ((Class<? extends Annotation>) cl.loadClass("javax.inje
                        logger.info("JSR-330 'javax.inject.Named' annotation found and supporte
                }
                catch (ClassNotFoundException ex) {
                        // JSR-330 API not available - simply skip.
                }
        }
```

可以看到默认ClassPathBeanDefinitionScanner会自动注册对@Component、@ManagedBean、@Named注解的Bean进行扫描。如果细心，到此我们就找到问题根源了。

4、在进行扫描时会通过include-filter/exclude-filter来判断你的Bean类是否是合法的：

```
protected boolean isCandidateComponent(MetadataReader metadataReader) throws IOException {
            for (TypeFilter tf : this.excludeFilters) {
                    if (tf.match(metadataReader, this.metadataReaderFactory)) {
                            return false;
                    }
            }
            for (TypeFilter tf : this.includeFilters) {
                    if (tf.match(metadataReader, this.metadataReaderFactory)) {
                            AnnotationMetadata metadata = metadataReader.getAnnotationMetac
                            if (!metadata.isAnnotated(Profile.class.getName())) {
                                    return true;
                            }
                            AnnotationAttributes profile = MetadataUtils.attributesFor(meta
                            return this.environment.acceptsProfiles(profile.getStringArray(
                    }
            }
            return false;
    }
```

即

首先通过exclude-filter 进行黑名单过滤；

然后通过include-filter 进行白名单过滤；

否则默认排除。

结论

```
<context:component-scan base-package="org.bdp">
     <context:include-filter type="annotation" expression="org.springframework.stereotype.Contr
</context:component-scan>
```

为什么这段代码不仅仅扫描@Controller注解的Bean，而且还扫描了@Component的子注解@Service、@Reposity。因为use-default-filters默认为true。所以如果不需要默认的，则use-default-filters= “false”禁用掉。

请参考

《SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结》

《第三章 DispatcherServlet详解 ——跟开涛学SpringMVC》中的ContextLoaderListener初始化的上下文和DispatcherServlet初始化的上下文关系。

如果在springmvc配置文件，不使用cn.javass.demo.web.controller前缀，而是使用cn.javass.demo，则service、dao层的bean可能也重新加载了，但事务的AOP代理没有配置在springmvc配置文件中，从而造成新加载的bean覆盖了老的bean，造成事务失效。只要使用use-default-filters= “false”禁用掉默认的行为就可以了。

问题不难，spring使用上的问题。总结一下方便再遇到类似问题的朋友参考。

1.27 idea内嵌jetty运行springmvc项目报ConversionFailedException

发表时间: 2013-03-16

22:35:50.609 [Scanner-0] DEBUG o.s.beans.TypeConverterDelegate - Original ConversionService attempt failed - ignored since PropertyEditor based conversion eventually succeeded

org.springframework.core.convert.ConversionFailedException: Failed to convert from type java.util.ArrayList<?> to type java.util.List<org.springframework.core.io.Resource> for value '[/WEB-INF/static/]'; nested exception is org.springframework.core.convert.ConverterNotFoundException: No converter found capable of converting from type java.lang.String to type org.springframework.core.io.Resource

at org.springframework.core.convert.support.ConversionUtils.invokeConverter(ConversionUtils.java:41) ~[spring-core-3.2.1.RELEASE.jar:3.2.1.RELEASE]

at org.springframework.core.convert.support.GenericConversionService.convert(GenericConversionService.java:169) ~[spring-core-3.2.1.RELEASE.jar:3.2.1.RELEASE]

at org.springframework.beans.TypeConverterDelegate.convertIfNecessary(TypeConverterDelegate.java:161) ~[spring-beans-3.2.1.RELEASE.jar:3.2.1.RELEASE]

at org.springframework.beans.BeanWrapperImpl.convertIfNecessary(BeanWrapperImpl.java:448) [spring-beans-3.2.1.RELEASE.jar:3.2.1.RELEASE]

at org.springframework.beans.BeanWrapperImpl.convertForProperty(BeanWrapperImpl.java:494) [spring-beans-3.2.1.RELEASE.jar:3.2.1.RELEASE]

.....

此问题之前也有朋友站内信我。该错误是因为：

1、内嵌jetty在运行时会锁定静态资源；因此在运行过程中不能删除静态资源；

2、问题的根源是启动了多个jetty实例，但是实际没有报端口冲突，可以检查任务管理器看是否有多个java/javaw进程在运行，如果是杀掉进程然后重试。

1.28 springmvc 3.2 @MatrixVariable注解

发表时间: 2013-03-16

示例

1、url 格式 /path;name=value;name=value，如

/showcase/product/category/select/single;domId=categoryId;domName=categoryName

2、控制器处理方法

```
 @RequestMapping(value = {"select/{selectType}", "select"}, method = RequestMethod.GET)
@PageableDefaults(sort = "weight=desc")
public String select(
        Searchable searchable, Model model,
        @PathVariable(value = "selectType") String selectType,
        @MatrixVariable(value = "domId", pathVar = "selectType") String domId,
        @MatrixVariable(value = "domName", pathVar = "selectType", required = false) String domName) {
    model.addAttribute("selectType", selectType);
```

通过@MatrixVariable可以获取到相关的参数值。

这个一般在传递path相关的参数时比较有用； 比如此处我是做数据参照（如树，可能在不同的模块参照，具有相同的参照值，但具有不同的名字），利用MatrixVariable 可以动态传递参照的元素id。

具体示例可参考

https://github.com/zhangkaitao/es 中的

com.sishuok.es.web.showcase.product.web.controller.CategoryController

1.29 spring3.2 带matrix变量的URL匹配问题

发表时间: 2013-03-18

spring3.2.3已经修复该bug

问题描述：

1、url可能是

http://localhost:9080/es-web/login

或一个以;开头的matrix变量

http://localhost:9080/es-web/login;JSESSIONID=a3595636-e414-4cff-bd37-a42edf53193d

2、控制器处理方法@RequestMapping匹配pattern写法

spring 3.1前，以下写法是没有问题的

```
@RequestMapping(value = {"/login"}, method = RequestMethod.GET)
    public String loginForm() {
        return "front/login";
    }
```

自spring3.2引入@MatrixVariable来匹配如/category;domId=a这样模式，使用如上pattern匹配『/login;JSESSIONID=a3595636-e414-4cff-bd37-a42edf53193d』时就会报如下错误

写道

```
java.lang.IllegalStateException: Pattern "/login" is not a match for "/login;JSESSIONID=a3595636-e414-4cff-bd37-a42edf53193d"
at org.springframework.util.Assert.state(Assert.java:385)
at org.springframework.util.AntPathMatcher.extractUriTemplateVariables(AntPathMatcher.java:287)
at org.springframework.web.servlet.mvc.method.RequestMappingInfoHandlerMapping.handleMatch(RequestMappi
```

3、解决方案

我们可以使用正则表达式解决

```
@RequestMapping(value = {"/{login:login;?.*}"}, method = RequestMethod.GET)
public String loginForm() {
    return "front/login";
}
```

有朋友可能觉得如下也是可以的

1、@RequestMapping(value = {"/login", "/login;JSESSIONID=*"})

实际是不行的，因为再选择最佳匹配模式时，使用如下代码：

```
Set<String> patterns = info.getPatternsCondition().getPatterns();
String bestPattern = patterns.isEmpty() ? lookupPath : patterns.iterator().next();
```

即第一个。所以此时即使配置了两个，也永远选择"/login"（不管顺序如何）；如果此处采用fallback也能搞定的。

2、修改spring源码，匹配时把『;.....』，即分号及后边的去掉（spring3.1就是这么实现的）。

综上，在不修改源码的情况下，最简单的就是使用正则表达式模式/{login:login;?.*}

@MatrixVariable使用请参考

http://jinnianshilongnian.iteye.com/blog/1830409

1.30 Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired失效

发表时间: 2013-04-19

今天一iteye网页在问答频道提问【Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired失效】，记录一下。

问题：

```
@ParentPackage("all")
@Namespace("/project")
public class ProjectAction extends BaseAction {
public final static Logger logger = LoggerFactory
.getLogger(ProjectAction.class);

@Autowired(required=true)
private ProjectService projectService;
```

如上代码@Autowired注入不了

分析：

1、首先从如上代码可以看出 走的是struts2注解，而且使用了struts2 convention 插件，这个插件会扫描如下配置的actionPackages 寻找action

```
<filter>
        <filter-name>struts2</filter-name>
        <filter-classfilter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteF
        <init-param>
<param-name>actionPackages</param-name>
```

```
<param-value>cn.javass</param-value>
</init-param>
```

2、但此时并没有把action交给spring，

3、接下来，因为集成了spring（有struts2-spring-plugin），所以要使用StrutsSpringObjectFactory创建bean，代码分析

```
@Override
    public Object buildBean(String beanName, Map<String, Object> extraContext, boolean injectIr
        Object o;

        if (appContext.containsBean(beanName)) {
            o = appContext.getBean(beanName); //拿不到bean
        } else {
            Class beanClazz = getClassInstance(beanName);
            o = buildBean(beanClazz, extraContext); //所以创建了一个
        }
        if (injectInternal) {
            injectInternalBeans(o);
        }
        return o;
    }
```

```
  /**
     * @param clazz
     * @param extraContext
     * @throws Exception
     */
    @Override
    public Object buildBean(Class clazz, Map<String, Object> extraContext) throws Exception {
```

```
        Object bean;

        try {
            // Decide to follow autowire strategy or use the legacy approach which mixes inject
            if (alwaysRespectAutowireStrategy) {//默认false
                // Leave the creation up to Spring
                bean = autoWiringFactory.createBean(clazz, autowireStrategy, false);
                injectApplicationContext(bean);
                return injectInternalBeans(bean);
            } else {
                bean = autoWiringFactory.autowire(clazz, AutowireCapableBeanFactory.AUTOWIRE_CO
                bean = autoWiringFactory.applyBeanPostProcessorsBeforeInitialization(bean, bean
                // We don't need to call the init-method since one won't be registered.
                bean = autoWiringFactory.applyBeanPostProcessorsAfterInitialization(bean, bean.
                return autoWireBean(bean, autoWiringFactory);
            }
        } catch (UnsatisfiedDependencyException e) {
            if (LOG.isErrorEnabled())
                LOG.error("Error building bean", e);
            // Fall back
            return autoWireBean(super.buildBean(clazz, extraContext), autoWiringFactory);
        }
    }
```

我们在shiro里使用如下代码 去代理shiro的代理：

```
<bean id="proxyCreator" class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoPr
          depends-on="lifecycleBeanPostProcessor">
        <property name="proxyTargetClass" value="true"/>
    </bean>

    <bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvis
        <property name="securityManager" ref="securityManager"/>
    </bean>
```

//StrutsSpringObjectFactory的如下代码将执行处理器的预处理

bean = autoWiringFactory.applyBeanPostProcessorsBeforeInitialization(bean, bean.getClass().getName());

//DefaultAdvisorAutoProxyCreator的postProcessBeforeInstantiation：将去完成代理bean 因此此时将返回代理Bean

//接着StrutsSpringObjectFactory的autoWireBean(bean, autoWiringFactory); 进行注入 所以此时注入到的是代理对象，因此如果字段注入 将注入不了。

解决方案

1、不使用actionPackages 而是 在类上加 @Controller @Scope 完全走spring

2、使用setter注入 而不是字段 如

```
private ProjectService projectService;

@Autowired(required=true)
public void setProjectService() {
}
```

1.31 Spring事务不起作用 问题汇总

发表时间: 2013-04-19

总有很多朋友询问spring事务不起作用怎么回事，这里我汇总下，欢迎补充：

1、首先使用如下代码 确认你的bean 是代理对象吗？

AopUtils.isAopProxy()

AopUtils.isCglibProxy() //cglib

AopUtils.isJdkDynamicProxy() //jdk动态代理

如果不是 那么就是切入点配置出错了 或者如果你使用了springmvc，可能是context:component-scan重复扫描引起的：

http://jinnianshilongnian.iteye.com/blog/1423971

http://jinnianshilongnian.iteye.com/blog/1762632

http://jinnianshilongnian.iteye.com/blog/1857189

2、如果是aop代理，那么说明代理成功，那么可能是如使用mysql且引擎是MyISAM造成的（因为不支持事务），改成InnoDB即可。

3、 如果你是基于类的代理，而非接口，如果想代理父类里的，可以用

execution(* com.sishuok.es..service..*+.*(..))

//+表示子类的也扫描（参考http://jinnianshilongnian.iteye.com/blog/1420691）

欢迎补充。

1.32 Spring3 Web MVC下的数据类型转换（第一篇）——《跟我学Spring3 Web MVC》抢先看

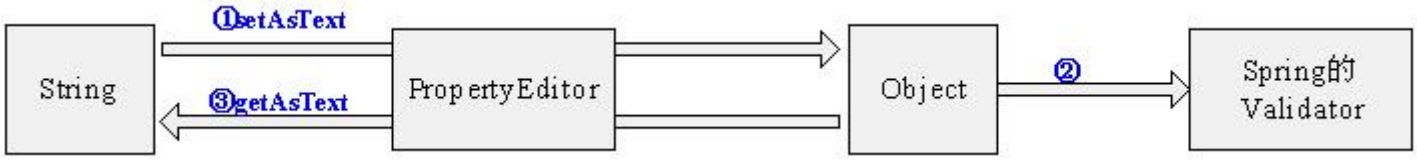
发表时间: 2012-05-03 关键字: spring, springMVC

基于spring-framework-3.1.1.RELEASE

7.1、简介

在编写可视化界面项目时，我们通常需要对数据进行类型转换、验证及格式化。

一、在Spring3之前，我们使用如下架构进行类型转换、验证及格式化：



流程：

①：类型转换：首先调用PropertyEditor的setAsText（String），内部根据需要调用setValue(Object)方法进行设置转换后的值；

②：数据验证：需要显示调用Spring的Validator接口实现进行数据验证；

③：格式化显示：需要调用PropertyEditor的getText进行格式化显示。

使用如上架构的缺点是：

（1、PropertyEditor被设计为只能String<——>Object之间转换，不能任意对象类型<——>任意类型，如我们常见的Long时间戳到Date类型的转换是办不到的；

（2、PropertyEditor是线程不安全的，也就是有状态的，因此每次使用时都需要创建一个，不可重用；

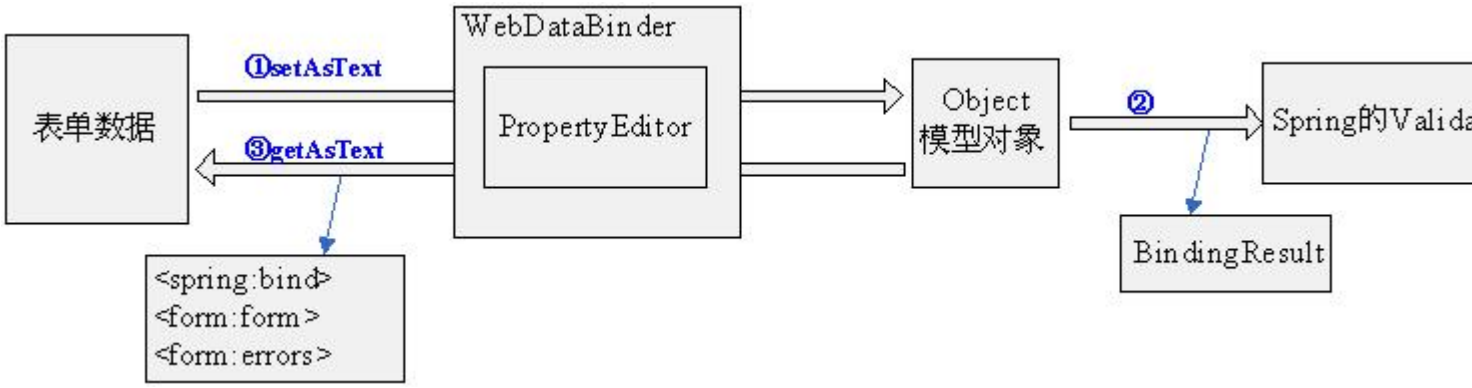
（3、PropertyEditor不是强类型的，setValue（Object）可以接受任意类型，因此需要我们自己判断类型是否兼容；

（4、需要自己编程实现验证，Spring3支持更棒的注解验证支持；

（5、在使用SpEL表达式语言或DataBinder时，只能进行String<--->Object之间的类型转换；

（6、不支持细粒度的类型转换/格式化，如UserModel的registerDate需要转换/格式化类似“2012-05-01”的数据，而OrderModel的orderDate需要转换/格式化类似“2012-05-01 15：11：13”的数据，因为大家都为java.util.Date类型，因此不太容易进行细粒度转换/格式化。

在Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：



流程：

①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实现；

②：数据验证：在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中；

③：格式化显示：在表单页面可以通过如下方式展示通过PropertyEditor格式化的数据和错误信息：

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

首先需要通过如上taglib指令引入spring的两个标签库。

```
//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
<spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>
```

```
//2、<spring:eval>标签，自动调用ConversionService并选择相应的Converter SPI进行格式化展示
<spring:eval expression="dataBinderTest.phoneNumber"></spring:eval>
```

如上代码能工作的前提是在RequestMappingHandlerMapping配置了`ConversionServiceExposingInterceptor`，它的作用是暴露`conversionService`到请求中以便如`<spring:eval>`标签使用。

```
//3、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
<form:form commandName="dataBinderTest">
    <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
</form:form>
```

```
//4、显示验证失败后的错误信息
<form:errors></form:errors>
```

接下来我们就详细学习一下这些知识吧。

7.2、数据类型转换

7.2.1、Spring3之前的PropertyEditor

PropertyEditor介绍请参考【4.16.1、数据类型转换】。

一、测试之前我们需要准备好测试环境：

（1、模型对象，和【4.16.1、数据类型转换】使用的一样，需要将DataBinderTestModel模型类及相关类拷贝过来放入cn.javass.chapter7.model包中。

（2、控制器定义：

```
package cn.javass.chapter7.web.controller;
//省略import
```

```
@Controller
public class DataBinderTestController {
    @RequestMapping(value = "/dataBind")
    public String test(DataBinderTestModel command) {
                //输出command对象看看是否绑定正确
                System.out.println(command);
                model.addAttribute("dataBinderTest", command);
                return "bind/success";
        }
}
```

(3、Spring配置文件定义，请参考chapter7-servlet.xml，并注册DataBinderTestController：

```
<bean class="cn.javass.chapter7.web.controller.DataBinderTestController"/>
```

（4、测试的URL：

http://localhost:9080/springmvc-chapter7/
dataBind?username=zhang&bool=yes&schooInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=mu
16:48:48&state=blocked

二、注解式控制器注册PropertyEditor：

1、使用WebDataBinder进行控制器级别注册PropertyEditor（控制器独享）

```
@InitBinder
//此处的参数也可以是ServletRequestDataBinder类型
public void initBinder(WebDataBinder binder) throws Exception {
        //注册自定义的属性编辑器
        //1、日期
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

```
        CustomDateEditor dateEditor = new CustomDateEditor(df, true);
        //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
        binder.registerCustomEditor(Date.class, dateEditor);
        //自定义的电话号码编辑器(和【4.16.1、数据类型转换】一样)
        binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());
}
```

和【4.16.1、数据类型转换】一节类似，只是此处需要通过@InitBinder来注册自定义的PropertyEditor。

2、使用WebBindingInitializer批量注册PropertyEditor

和【4.16.1、数据类型转换】不太一样，因为我们的注解式控制器是POJO，没有实现任何东西，因此无法注入WebBindingInitializer，此时我们需要把WebBindingInitializer注入到我们的RequestMappingHandlerAdapter或AnnotationMethodHandlerAdapter，这样对于所有的注解式控制器都是共享的。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter
    <property name="webBindingInitializer">
        <bean class="cn.javass.chapter7.web.controller.support.initializer.MyWebBindingInitiali
    </property>
</bean>
```

此时我们注释掉控制器级别通过@InitBinder注册PropertyEditor的方法。

3、全局级别注册PropertyEditor（全局共享）

和【4.16.1、数据类型转换】一节一样，此处不再重复。请参考【4.16.1、数据类型转换】的【全局级别注册PropertyEditor（全局共享）】。

接下来我们看一下Spring3提供的更强大的类型转换支持。

7.2.2、Spring3开始的类型转换系统

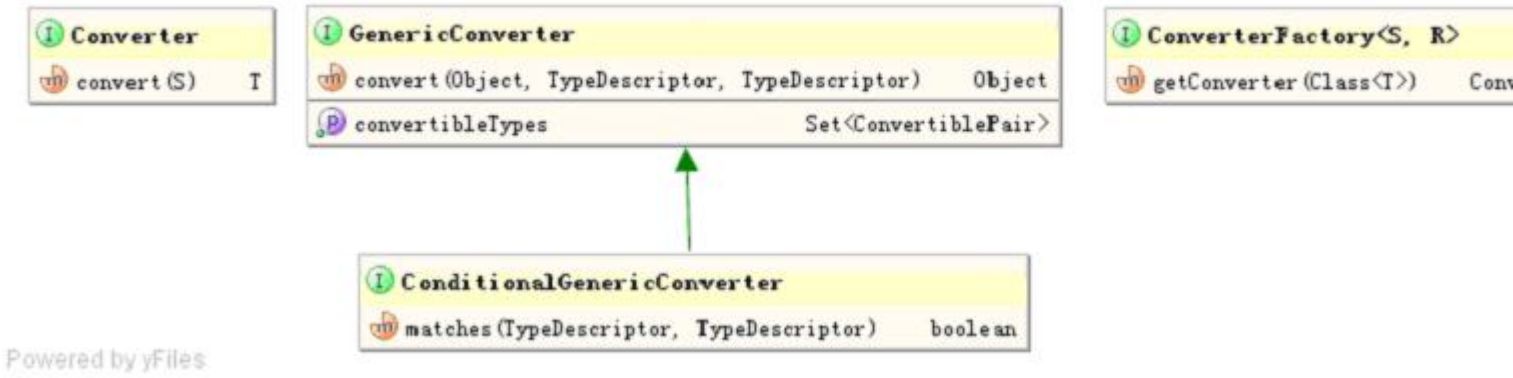
Spring3引入了更加通用的类型转换系统，其定义了SPI接口（Converter等）和相应的运行时执行类型转换的API（ConversionService等），在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来转换外部Bean属性的值到Bean属性需要的类型。

该类型转换系统是Spring通用的，其定义在org.springframework.core.convert包中，不仅仅在Spring Web MVC场景下。目标是完全替换PropertyEditor，提供无状态、强类型且可以在任意类型之间转换的类型转换系统，可以用于任何需要的地方，如SpEL、数据绑定。

Converter SPI完成通用的类型转换逻辑，如java.util.Date<---->java.lang.Long或java.lang.String---->PhoneNumberModel等。

7.2.2.1、架构

1、类型转换器：提供类型转换的实现支持。



一个有如下三种接口：

（1、Converter：类型转换器，用于转换S类型到T类型，此接口的实现必须是线程安全的且可以被共享。

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> { //① S是源类型 T是目标类型
        T convert(S source); //② 转换S类型的source到T目标类型的转换方法
}
```

示例：请参考cn.javass.chapter7.converter.support.StringToPhoneNumberConverter转换器，用于将String--->PhoneNumberModel。

此处我们可以看到Converter接口实现只能转换一种类型到另一种类型，不能进行多类型转换，如将一个数组转换成集合，如（`String[] ----> List<String>`、String[]----->List<PhoneNumberModel>等）。

（2、GenericConverter和ConditionalGenericConverter：GenericConverter接口实现能在多种类型之间进行转换，ConditionalGenericConverter是有条件的在多种类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
        Set<ConvertiblePair> getConvertibleTypes();
        Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

getConvertibleTypes:指定了可以转换的目标类型对；

convert：在sourceType和targetType类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface ConditionalGenericConverter extends GenericConverter {
        boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

matches：用于判断sourceType和targetType类型之间能否进行类型转换。

示例：如org.springframework.core.convert.support.ArrayToCollectionConverter和CollectionToArrayConverter用于在数组和集合间进行转换的ConditionalGenericConverter实现，如在String[]<---->List<String>、String[]<---->`List<PhoneNumberModel>`等之间进行类型转换。

对于我们大部分用户来说一般不需要自定义GenericConverter, 如果需要可以参考内置的GenericConverter来实现自己的。

（3、ConverterFactory：工厂模式的实现，用于选择将一种S源类型转换为R类型的子类型T的转换器的工厂接口。

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
        <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

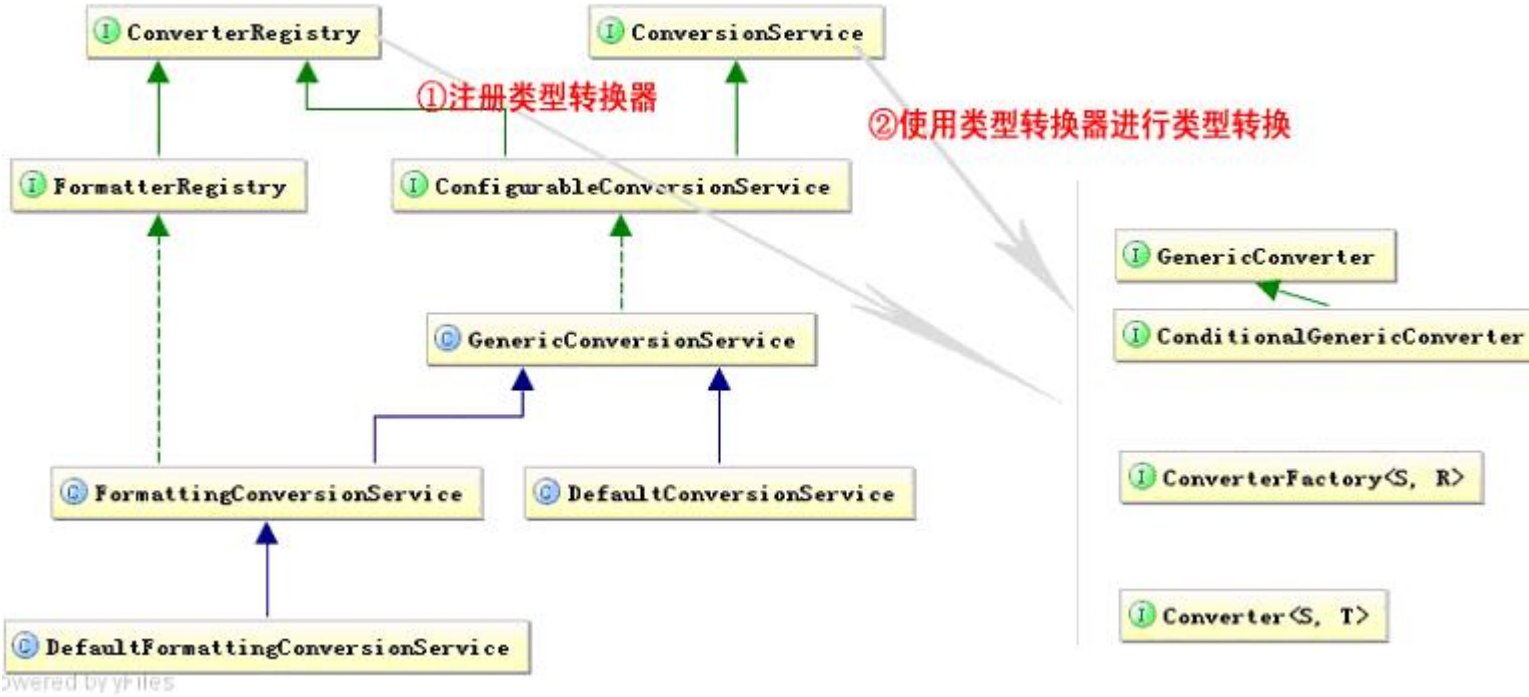
S：源类型；R目标类型的父类型；T：目标类型，且是R类型的子类型；

getConverter：得到目标类型的对应的转换器。

示例：如org.springframework.core.convert.support.NumberToNumberConverterFactory用于在Number类型子类型之间进行转换，如Integer--->Double， Byte---->Integer， Float--->Double等。

对于我们大部分用户来说一般不需要自定义ConverterFactory，如果需要可以参考内置的ConverterFactory来实现自己的。

2、类型转换器注册器、类型转换服务：提供类型转换器注册支持，运行时类型转换API支持。



一共有如下两种接口：

（1、ConverterRegistry：类型转换器注册支持，可以注册/删除相应的类型转换器。

```
package org.springframework.core.convert.converter;
public interface ConverterRegistry {
        void addConverter(Converter<?, ?> converter);
        void addConverter(Class<?> sourceType, Class<?> targetType, Converter<?, ?> converter);
        void addConverter(GenericConverter converter);
        void addConverterFactory(ConverterFactory<?, ?> converterFactory);
        void removeConvertible(Class<?> sourceType, Class<?> targetType);
}
```

可以注册：Converter实现，GenericConverter实现，ConverterFactory实现。

（2、ConversionService：运行时类型转换服务接口，提供运行期类型转换的支持。

```
package org.springframework.core.convert;
public interface ConversionService {
        boolean canConvert(Class<?> sourceType, Class<?> targetType);
        boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
        <T> T convert(Object source, Class<T> targetType);
```

```
        Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

convert：将源对象转换为目标类型的目标对象。

Spring提供了两个默认实现（其都实现了ConverterRegistry、ConversionService接口）：

DefaultConversionService:默认的类型转换服务实现；

DefaultFormattingConversionService：带数据格式化支持的类型转换服务实现，一般使用该服务实现即可。

7.2.2.2、Spring内建的类型转换器如下所示：

类名	说明
第一组：标量转换器	
StringToBooleanConverter	String----->Boolean true:true/on/yes/1； false:false/off/no/0
ObjectToStringConverter	Object----->String 调用toString方法转换
StringToNumberConverterFactory	String----->Number（如Integer、Long等）
NumberToNumberConverterFactory	Number子类型(Integer、Long、Double等)<——> Number子类型(Integer、Long、Double等)
StringToCharacterConverter	String----->java.lang.Character

	取字符串第一个字符
NumberToCharacterConverter	Number子类型(Integer、Long、Double等)——> java.lang.Character
CharacterToNumberFactory	java.lang.Character ——>Number子类型(Integer、Long、Double等)
StringToEnumConverterFactory	String----->enum类型 通过Enum.valueOf将字符串转换为需要的enum类型
EnumToStringConverter	enum类型----->String 返回enum对象的name()值
StringToLocaleConverter	String----->java.util.Local
PropertiesToStringConverter	java.util.Properties----->String 默认通过ISO-8859-1解码
StringToPropertiesConverter	String----->java.util.Properties 默认使用ISO-8859-1编码

第二组：集合、数组相关转换器

ArrayToCollectionConverter	任意S数组---->任意T集合（List、Set）
CollectionToArrayConverter	任意T集合（List、Set）---->任意S数组
ArrayToArrayConverter	任意S数组<---->任意T数组

CollectionToCollectionConverter	任意T集合（List、Set）<---->任意T集合（List、Set） 即集合之间的类型转换
MapToMapConverter	Map<---->Map之间的转换
ArrayToStringConverter	任意S数组---->String类型
StringToArrayConverter	String----->数组 默认通过“,”分割，且去除字符串的两边空格(trim)
ArrayToObjectConverter	任意S数组---->任意Object的转换 (如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToArrayConverter	Object----->单元素数组
CollectionToStringConverter	任意T集合（List、Set）---->String类型
StringToCollectionConverter	String----->集合（List、Set） 默认通过“,”分割，且去除字符串的两边空格(trim)
CollectionToObjectConverter	任意T集合---->任意Object的转换 (如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToCollectionConverter	Object----->单元素集合

第三组：默认（fallback）转换器：之前的转换器不能转换时调用

ObjectToObjectConverter	Object（S）----->Object（T） 首先尝试valueOf进行转换、没有则尝试new构造器(S)
IdToEntityConverter	Id(S)----->Entity(T) 查找并调用public static T find[EntityName](S)获取目标对象，EntityName是T类型的简单类型
FallbackObjectToStringConverter	Object----->String ConversionService作为恢复使用，即其他转换器不能转换时调用（执行对象的toString()方法）

S：代表源类型，T：代表目标类型

如上的转换器在使用转换服务实现DefaultConversionService和DefaultFormattingConversionService时会自动注册。

7.2.2.3、示例

（1、自定义String----->PhoneNumberModel的转换器

```
package cn.javass.chapter7.web.controller.support.converter;
//省略import
public class StringToPhoneNumberConverter implements Converter<String, PhoneNumberModel> {
        Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
        @Override
        public PhoneNumberModel convert(String source) {
                if(!StringUtils.hasLength(source)) {
                        //①如果source为空 返回null
```

```
                return null;
            }
            Matcher matcher = pattern.matcher(source);
            if(matcher.matches()) {
                //②如果匹配 进行转换
                PhoneNumberModel phoneNumber = new PhoneNumberModel();
                phoneNumber.setAreaCode(matcher.group(1));
                phoneNumber.setPhoneNumber(matcher.group(2));
                return phoneNumber;
            } else {
                //③如果不匹配 转换失败
                throw new IllegalArgumentException(String.format("类型转换失败，需要格式|
            }
        }
}
```

String转换为Date的类型转换器，请参考
cn.javass.chapter7.web.controller.support.converter.StringToDateConverter。

(2、测试用例(cn.javass.chapter7.web.controller.support.converter.ConverterTest)

```
@Test
public void testStringToPhoneNumberConvert() {
        DefaultConversionService conversionService = new DefaultConversionService();
        conversionService.addConverter(new StringToPhoneNumberConverter());

        String phoneNumberStr = "010-12345678";
        PhoneNumberModel phoneNumber = conversionService.convert(phoneNumberStr, PhoneNumberMod

        Assert.assertEquals("010", phoneNumber.getAreaCode());
}
```

类似于PhoneNumberEditor将字符串“010-12345678”转换为PhoneNumberModel。

```
@Test
public void testOtherConvert() {
        DefaultConversionService conversionService = new DefaultConversionService();

        //"1"--->true（字符串“1”可以转换为布尔值true）
        Assert.assertEquals(Boolean.valueOf(true), conversionService.convert("1", Boolean.class

        //"1,2,3,4"--->List（转换完毕的集合大小为4）
        Assert.assertEquals(4, conversionService.convert("1,2,3,4", List.class).size());
}
```

其他类型转换器使用也是类似的，此处不再重复。

7.2.2.4、集成到Spring Web MVC环境

（1、注册ConversionService实现和自定义的类型转换器

```
<!-- ①注册ConversionService -->
<bean id="conversionService" class="org.springframework.format.support.
                                                                        Format
    <property name="converters">
       <list>
                <bean class="cn.javass.chapter7.web.controller.support.
                                                        converter.StringToPhoneNumberConverter
                <bean class="cn.javass.chapter7.web.controller.support.
                                                        converter.StringToDateConverter">
                    <constructor-arg value="yyyy-MM-dd"/>
                </bean>
        </list>
    </property>
</bean>
```

FormattingConversionServiceFactoryBean：是FactoryBean实现，默认使用DefaultFormattingConversionService转换器服务实现；

converters：注册我们自定义的类型转换器，此处注册了String--->PhoneNumberModel和String--->Date的类型转换器。

（2、通过ConfigurableWebBindingInitializer注册ConversionService

```
<!-- ②使用ConfigurableWebBindingInitializer注册conversionService -->
<bean id="webBindingInitializer" class="org.springframework.web.bind.support.


        <property name="conversionService" ref="conversionService"/>
</bean>
```

此处我们通过ConfigurableWebBindingInitializer绑定初始化器进行ConversionService的注册；

3、注册ConfigurableWebBindingInitializer到RequestMappingHandlerAdapter

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.
                                                        RequestMappingHandlerAdapter">
<property name="webBindingInitializer" ref="webBindingInitializer"/>
</bean>
```

通过如上配置，我们就完成了Spring3.0的类型转换系统与Spring Web MVC的集成。此时可以启动服务器输入之前的URL测试了。

此时可能有人会问，如果我同时使用PropertyEditor和ConversionService，执行顺序是什么呢？内部首先查找PropertyEditor进行类型转换，如果没有找到相应的PropertyEditor再通过ConversionService进行转换。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

1.33 Spring 注入集合类型

发表时间: 2013-04-28

最近有朋友问我如下问题：

我定义了一个类：

```
@Service
public class StringTest implements CachedRowSet,SortedSet<String>,Cloneable
```

```
@Controller
public class HomeController {
@Autowired
CachedRowSet message;

@Autowired
CachedRowSet message1;
}
```

这里CachedRowSet ， 等其他接口都是可以注入的，包括StringTest 也行。

但是使用：

```
@Autowired
SortedSet<String> message
```

就不行了。启动报错。

源码分析：

org.springframework.beans.factory.support.DefaultListableBeanFactory

```
  protected Object doResolveDependency(DependencyDescriptor descriptor, Class<?> type, String b
                        Set<String> autowiredBeanNames, TypeConverter typeConverter) throws Bea

                Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor);
                if (value != null) {
                        if (value instanceof String) {
```

```
                        String strVal = resolveEmbeddedValue((String) value);
                        BeanDefinition bd = (beanName != null && containsBean(beanName)
                        value = evaluateBeanDefinitionString(strVal, bd);
                }
                TypeConverter converter = (typeConverter != null ? typeConverter : getT
                return (descriptor.getField() != null ?
                                converter.convertIfNecessary(value, type, descriptor.ge
                                converter.convertIfNecessary(value, type, descriptor.ge
        }

        if (type.isArray()) {
                Class<?> componentType = type.getComponentType();
                Map<String, Object> matchingBeans = findAutowireCandidates(beanName, co
                if (matchingBeans.isEmpty()) {
                        if (descriptor.isRequired()) {
                                raiseNoSuchBeanDefinitionException(componentType, "arra
                        }
                        return null;
                }
                if (autowiredBeanNames != null) {
                        autowiredBeanNames.addAll(matchingBeans.keySet());
                }
                TypeConverter converter = (typeConverter != null ? typeConverter : getT
                return converter.convertIfNecessary(matchingBeans.values(), type);
        }
        else if (Collection.class.isAssignableFrom(type) && type.isInterface()) {
                Class<?> elementType = descriptor.getCollectionType();
                if (elementType == null) {
                        if (descriptor.isRequired()) {
                                throw new FatalBeanException("No element type declared
                        }
                        return null;
                }
                Map<String, Object> matchingBeans = findAutowireCandidates(beanName, el
                if (matchingBeans.isEmpty()) {
                        if (descriptor.isRequired()) {
                                raiseNoSuchBeanDefinitionException(elementType, "collec
```

```
                        }
                        return null;
                }
                if (autowiredBeanNames != null) {
                        autowiredBeanNames.addAll(matchingBeans.keySet());
                }
                TypeConverter converter = (typeConverter != null ? typeConverter : getT
                return converter.convertIfNecessary(matchingBeans.values(), type);
        }
        else if (Map.class.isAssignableFrom(type) && type.isInterface()) {
                Class<?> keyType = descriptor.getMapKeyType();
                if (keyType == null || !String.class.isAssignableFrom(keyType)) {
                        if (descriptor.isRequired()) {
                                throw new FatalBeanException("Key type [" + keyType + '
                                                "] must be assignable to [java.lang.Str
                        }
                        return null;
                }
                Class<?> valueType = descriptor.getMapValueType();
                if (valueType == null) {
                        if (descriptor.isRequired()) {
                                throw new FatalBeanException("No value type declared fo
                        }
                        return null;
                }
                Map<String, Object> matchingBeans = findAutowireCandidates(beanName, va
                if (matchingBeans.isEmpty()) {
                        if (descriptor.isRequired()) {
                                raiseNoSuchBeanDefinitionException(valueType, "map with
                        }
                        return null;
                }
                if (autowiredBeanNames != null) {
                        autowiredBeanNames.addAll(matchingBeans.keySet());
                }
                return matchingBeans;
        }
```

```
            else {
                    Map<String, Object> matchingBeans = findAutowireCandidates(beanName, ty
                    if (matchingBeans.isEmpty()) {
                            if (descriptor.isRequired()) {
                                    raiseNoSuchBeanDefinitionException(type, "", descriptor
                            }
                            return null;
                    }
                    if (matchingBeans.size() > 1) {
                            String primaryBeanName = determinePrimaryCandidate(matchingBear
                            if (primaryBeanName == null) {
                                    throw new NoUniqueBeanDefinitionException(type, matchir
                            }
                            if (autowiredBeanNames != null) {
                                    autowiredBeanNames.add(primaryBeanName);
                            }
                            return matchingBeans.get(primaryBeanName);
                    }
                    // We have exactly one match.
                    Map.Entry<String, Object> entry = matchingBeans.entrySet().iterator().r
                    if (autowiredBeanNames != null) {
                            autowiredBeanNames.add(entry.getKey());
                    }
                    return entry.getValue();
            }
    }
```

从上边的源码大家可以看出：

1、首先判断注入的类型，如果是数组、Collection、Map，则注入的是元素数据，即查找与元素类型相同的Bean的注入到集合，而不是找跟集合类型相同的

2、对于Map，key只能是String类型，而且默认是Bean的名字

结论：

1、对于数组、集合、Map，注入的元素类型，如SortedSet<String> 其实是找所有String类型的Bean注入到集合

2、Map，key只能是String类型，而且默认是Bean的名字