咕泡学院 JavaVIP 高级课程教案

深入分析 Spring 源码（第三阶段）

关于本文档

主题	咕泡学院 Java VIP 高级课程教案--深入分析 Spring 源码（第三阶段）
主讲	Tom 老师
适用对象	咕泡学院 Java 高级 VIP 学员及 VIP 授课老师
源码版本	v3.2.6.RELEASE

5.6、Spring AOP 设计原理及具体实践

5.6.1、SpringAOP 应用示例

AOP 是 OOP 的延续，是 Aspect Oriented Programming 的缩写，意思是面向切面编程。可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP 设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP 可以说也是这种目标的一种实现。

我们现在做的一些非业务，如：日志、事务、安全等都会写在业务代码中(也即是说，这些非业务类横切于业务类)，但这些代码往往是重复，复制——粘贴式的代码会给程序的维护带来不便，AOP 就实现了把这些业务需求与系统需求分开来做。这种解决的方式也称代理机制。

先来了解一下 AOP 的相关概念

- **切面（Aspect）**：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。“切面”在 ApplicationContext 中<aop:aspect>来配置。
- **连接点（Joinpoint）**：程序执行过程中的某一行为，例如，MemberService .get 的调用或者 MemberService .delete 抛出异常等行为。
- **通知（Advice）**：“切面”对于某个“连接点”所产生的动作。其中，一个“切面”可以包含多个“Advice”。
- **切入点（Pointcut）**：匹配连接点的断言，在 AOP 中通知和一个切入点表达式关联。切面中的所有通知所关注的连接点，都由切入点表达式来决定。
- **目标对象（Target Object）**：被一个或者多个切面所通知的对象。例如，AServcieImpl 和 BServiceImpl，当然在实际运行时，Spring AOP 采用代理实现，实际 AOP 操作的是 TargetObject 的代理对象。
- **AOP 代理（AOP Proxy）**：在 Spring AOP 中有两种代理方式，JDK 动态代理和 CGLIB 代理。默认情况下，TargetObject 实现了接口时，则采用 JDK 动态代理，例如，AServiceImpl；反之，采用 CGLIB 代理，例如，BServiceImpl。强制使用 CGLIB 代理需要将 <aop:config>的 proxy-target-class 属性设为 true。

通知（Advice）类型：

- **前置通知（Before advice）**：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext 中在<aop:aspect>里面使用<aop:before>元素进行声明。例如，TestAspect 中的 doBefore 方法。
- **后置通知（After advice）**：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext 中在<aop:aspect>里面使用<aop:after>元素进行声明。例如，ServiceAspect 中的 returnAfter 方法，所以 Teser 中调用 UserService.delete 抛出异常时，returnAfter 方法仍然执行。
- **返回后通知（After return advice）**：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext 中在<aop:aspect>里面使用<after-returning>元素进行声明。
- **环绕通知（Around advice）**：包围一个连接点的通知，类似 Web 中 Servlet 规范中的 Filter 的 doFilter 方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在<aop:aspect>里面使用<aop:around>元素进行声明。例如，ServiceAspect 中的 around 方法。
- **抛出异常后通知（After throwing advice）**：在方法抛出异常退出时执行的通知。ApplicationContext 中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。例如，ServiceAspect 中的 returnThrow 方法。

注：可以将多个通知应用到一个目标对象上，即可以将多个切面织入到同一目标对象。

使用 Spring AOP 可以基于两种方式，一种是比较方便和强大的注解方式，另一种则是中规中矩的 xml 配置方式。

先说注解，使用注解配置 Spring AOP 总体分为两步，第一步是在 xml 文件中声明激活自动扫描组件功能，同时激活自动代理功能（来测试 AOP 的注解功能）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.gupaoedu"/>
    <context:annotation-config />
</beans>
```

第二步是为 Aspect 切面类添加注解：

```
//声明这是一个组件
@Component
//声明这是一个切面Bean
@Aspect
public class AnnotaionAspect {

    private final static Logger log = Logger.getLogger(AnnotaionAspect.class);

    //配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点
    @Pointcut("execution(* com.gupaoedu.aop.service..*(..))")
    public void aspect(){ }

    /*
     * 配置前置通知,使用在方法aspect()上注册的切入点
     * 同时接受JoinPoint切入点对象,可以没有该参数
     */
    @Before("aspect()")
    public void before(JoinPoint joinPoint){
        log.info("before " + joinPoint);
    }

    //配置后置通知,使用在方法aspect()上注册的切入点
    @After("aspect()")
```

```
    public void after(JoinPoint joinPoint){
        log.info("after " + joinPoint);
    }

    //配置环绕通知,使用在方法aspect()上注册的切入点
    @Around("aspect()")
    public void around(JoinPoint joinPoint){
        long start = System.currentTimeMillis();
        try {
            ((ProceedingJoinPoint) joinPoint).proceed();
            long end = System.currentTimeMillis();
            log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms!");
        } catch (Throwable e) {
            long end = System.currentTimeMillis();
            log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms with exception : " +
e.getMessage());
        }
    }

    //配置后置返回通知,使用在方法aspect()上注册的切入点
    @AfterReturning("aspect()")
    public void afterReturn(JoinPoint joinPoint){
        log.info("afterReturn " + joinPoint);
    }

    //配置抛出异常后通知,使用在方法aspect()上注册的切入点
    @AfterThrowing(pointcut="aspect()", throwing="ex")
    public void afterThrow(JoinPoint joinPoint, Exception ex){
        log.info("afterThrow " + joinPoint + "\t" + ex.getMessage());
    }

}
```

测试代码

```
@ContextConfiguration(locations = {"classpath*:application-context.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class AnnotationTester {
    @Autowired MemberService annotationService;
    @Autowired ApplicationContext app;


    @Test
//  @Ignore
    public void test(){
        System.out.println("=====这是一条华丽的分割线======");
```

```
        AnnotaionAspect aspect = app.getBean(AnnotaionAspect.class);
        System.out.println(aspect);
        annotationService.save(new Member());

        System.out.println("=====这是一条华丽的分割线======");
        try {
            annotationService.delete(1L);
        } catch (Exception e) {
            //e.printStackTrace();
        }
    }

}
```

控制台输出如下：

```
=====这是一条华丽的分割线======
com.gupaoedu.aop.aspect.AnnotaionAspect@6ef714a0
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - before execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.ArgsAspect - beforeArgUser execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - save member method . . .
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - around execution(void
com.gupaoedu.aop.service.MemberService.save(Member))       Use time : 38 ms!
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - after execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - afterReturn execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
=====这是一条华丽的分割线======
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - before execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.ArgsAspect - beforeArgId execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))    ID:1
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - delete method . . .
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - around execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))    Use time : 3 ms with exception : spring aop
ThrowAdvice演示
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - after execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))

[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - afterReturn execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))
```

可以看到，正如我们预期的那样，虽然我们并没有对 MemberService 类包括其调用方式做任何改变，但是 Spring 仍然拦截到了其中方法的调用，或许这正是 AOP 的魔力所在。

再简单说一下 xml 配置方式，其实也一样简单：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                        http://www.springframework.org/schema/tx
                        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
    <aop:aspectj-autoproxy proxy-target-class="true"/>

    <bean id="xmlAspect" class="com.gupaoedu.aop.aspect.XmlAspect"></bean>
    <!-- AOP配置 -->
    <aop:config>
        <!-- 声明一个切面,并注入切面Bean,相当于@Aspect -->
        <aop:aspect ref="xmlAspect">
            <!-- 配置一个切入点,相当于@Pointcut -->
            <aop:pointcut expression="execution(* com.gupaoedu.aop.service..*(..))" id="simplePointcut"/>
            <!-- 配置通知,相当于@Before、@After、@AfterReturn、@Around、@AfterThrowing -->
            <aop:before pointcut-ref="simplePointcut" method="before"/>
            <aop:after pointcut-ref="simplePointcut" method="after"/>
            <aop:after-returning pointcut-ref="simplePointcut" method="afterReturn"/>
            <aop:after-throwing pointcut-ref="simplePointcut" method="afterThrow" throwing="ex"/>
        </aop:aspect>
    </aop:config>

</beans>
```

个人觉得不如注解灵活和强大，你可以不同意这个观点，但是不知道如下的代码会不会让你的想法有所改善：

```
//配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点
    @Pointcut("execution(* com.gupaoedu.aop.service..*(..))")
    public void aspect(){ }

    //配置前置通知,拦截返回值为cn.ysh.studio.spring.mvc.bean.User的方法
    @Before("execution(com.gupaoedu.model.Member com.gupaoedu.aop.service..*(..))")
    public void beforeReturnUser(JoinPoint joinPoint){
        log.info("beforeReturnUser " + joinPoint);
    }
```

```
    //配置前置通知,拦截参数为cn.ysh.studio.spring.mvc.bean.User的方法
    @Before("execution(* com.gupaoedu.aop.service..*(com.gupaoedu.model.Member))")
    public void beforeArgUser(JoinPoint joinPoint){
        log.info("beforeArgUser " + joinPoint);
    }

    //配置前置通知,拦截含有long类型参数的方法,并将参数值注入到当前方法的形参id中
    @Before("aspect()&&args(id)")
    public void beforeArgId(JoinPoint joinPoint, long id){
        log.info("beforeArgId " + joinPoint + "\tID:" + id);
    }
```

以下是 MemberService 的代码：

```
@Service
public class MemberService {

    private final static Logger log = Logger.getLogger(AnnotaionAspect.class);

    public Member get(long id){
        log.info("getMemberById method . . .");
        return new Member();
    }


    public Member get(){
        log.info("getMember method . . .");
        return new Member();
    }

    public void save(Member member){
        log.info("save member method . . .");
    }

    public boolean delete(long id) throws Exception{
        log.info("delete method . . .");
        throw new Exception("spring aop ThrowAdvice演示");
    }

}
```

应该说学习 Spring AOP 有两个难点，第一点在于理解 AOP 的理念和相关概念，第二点在于灵活掌握和使用切入点表达式。概念的理解通常不在一朝一夕，慢慢浸泡的时间长了，自然就明白了，下面我们简单地介绍一下切入点表达式的配置规则吧。

通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
throws-pattern?
```

- **modifiers-pattern**：方法的操作权限
- **ret-type-pattern**：返回值
- **declaring-type-pattern**：方法所在的包
- **name-pattern**：方法名
- **parm-pattern**：参数名
- **throws-pattern**：异常

其中，除 ret-type-pattern 和 name-pattern 之外，其他都是可选的。上例中，execution(* com.spring.service.*.*(..))表示 com.spring.service 包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

最后说一下通知参数

可以通过 args 来绑定参数，这样就可以在通知（Advice）中访问具体参数了。例如，<aop:aspect>配置如下：

```
<aop:config>
        <aop:aspect ref="xmlAspect">
         <aop:pointcut id="simplePointcut"
          expression="execution(* com.gupaoedu.aop.service..*(..)) and args(msg,..)" />
          <aop:after pointcut-ref="simplePointcut" method="after"/>
        </aop:aspect>
</aop:config>
```

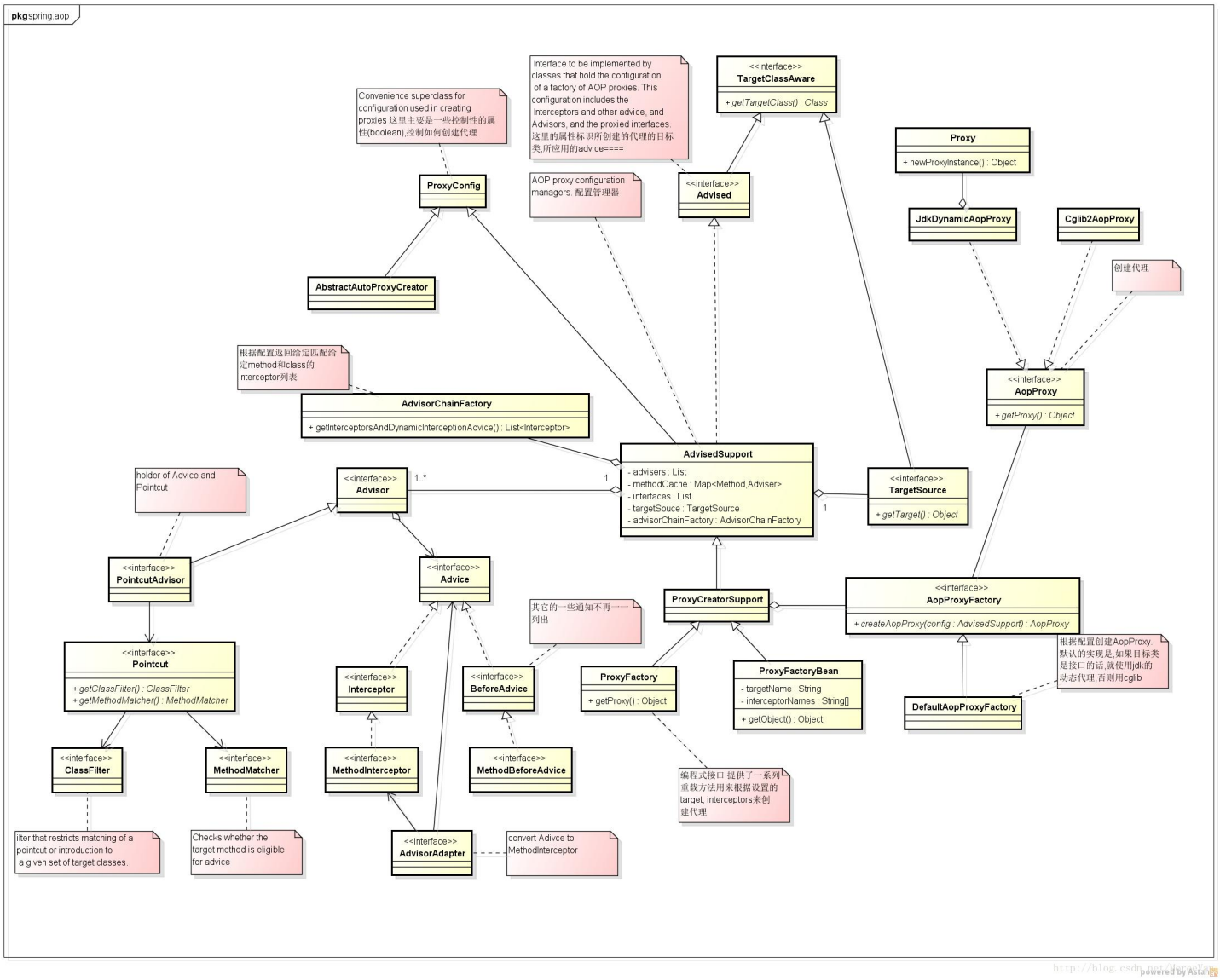
上面的代码 args(msg,..)是指将切入点方法上的第一个 String 类型参数添加到参数名为 msg 的通知的入参上，这样就可以直接使用该参数啦。

访问当前的连接点

在上面的 Aspect 切面 Bean 中已经看到了，每个通知方法第一个参数都是 JoinPoint。其实，在 Spring 中，任何通知（Advice）方法都可以将第一个参数定义为 **org.aspectj.lang.JoinPoint** 类型用以接受当前连接点对象。JoinPoint 接口提供了一系列有用的方法， 比如 getArgs()（返回方法参数）、getThis()（返回代理对象）、getTarget()（返回目标）、getSignature()（返回正在被通知的方法相关信息）和 toString()（打印出正在被通知的方法的有用信息）。

5.6.2、SpringAOP 设计原理及源码分析

开始之前先上图，看看 Spring 中主要的 AOP 组件



Spring 提供了两种方式来生成代理对象: JDKProxy 和 Cglib，具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。下面我们来研究一下 Spring 如何使用 JDK 来生成代理对象，具体的生成代码放在 JdkDynamicAopProxy 这个类中，直接上相关代码：

```
/**
    * <ol>
    * <li>获取代理类要实现的接口,除了 Advised 对象中配置的,还会加上 SpringProxy, Advised(opaque=false)
    * <li>检查上面得到的接口中有没有定义 equals 或者 hashcode 的接口
    * <li>调用 Proxy.newProxyInstance 创建代理对象
    * </ol>
    */
    public Object getProxy(ClassLoader classLoader) {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating JDK dynamic proxy: target source is " +
```

```
this.advised.getTargetSource());
        }
        Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }
```

那这个其实很明了，注释上我也已经写清楚了，不再赘述。

下面的问题是，代理对象生成了，那切面是如何织入的？

我们知道 InvocationHandler 是 JDK 动态代理的核心，生成的代理对象的方法调用都会委托到 InvocationHandler.invoke()方法。而通过 JdkDynamicAopProxy 的签名我们可以看到这个类其实也实现了 InvocationHandler，下面我们就通过分析这个类中实现的 invoke()方法来具体看下 Spring AOP 是如何织入切面的。

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        MethodInvocation invocation;
        Object oldProxy = null;
        boolean setProxyContext = false;

        TargetSource targetSource = this.advised.targetSource;
        Class targetClass = null;
        Object target = null;

        try {
            //eqauls()方法，具目标对象未实现此方法
            if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
                // The target does not implement the equals(Object) method itself.
                return equals(args[0]);
            }
            //hashCode()方法，具目标对象未实现此方法
            if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
                // The target does not implement the hashCode() method itself.
                return hashCode();
            }
            //Advised 接口或者其父接口中定义的方法,直接反射调用,不应用通知
            if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&
                    method.getDeclaringClass().isAssignableFrom(Advised.class)) {
                // Service invocations on ProxyConfig with the proxy config...
                return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
            }

            Object retVal;

            if (this.advised.exposeProxy) {
```

```
                // Make invocation available if necessary.
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }

            // May be null. Get as late as possible to minimize the time we "own" the target,
            // in case it comes from a pool.
            //获得目标对象的类
            target = targetSource.getTarget();
            if (target != null) {
                targetClass = target.getClass();
            }

            // Get the interception chain for this method.
            //获取可以应用到此方法上的 Interceptor 列表
            List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
targetClass);

            // Check whether we have any advice. If we don't, we can fallback on direct
            // reflective invocation of the target, and avoid creating a MethodInvocation.
            //如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 method.invoke(target, args)
            if (chain.isEmpty()) {
                // We can skip creating a MethodInvocation: just invoke the target directly
                // Note that the final invoker must be an InvokerInterceptor so we know it does
                // nothing but a reflective operation on the target, and no hot swapping or fancy
proxying.
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
            }
            else {
                // We need to create a method invocation...
                //创建 MethodInvocation
                invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass,
chain);
                // Proceed to the joinpoint through the interceptor chain.
                retVal = invocation.proceed();
            }

            // Massage return value if necessary.
            Class<?> returnType = method.getReturnType();
            if (retVal != null && retVal == target && returnType.isInstance(proxy) &&
                    !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                // Special case: it returned "this" and the return type of the method
                // is type-compatible. Note that we can't help if the target sets
                // a reference to itself in another returned object.
```

```
                retVal = proxy;
            } else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
                throw new AopInvocationException("Null return value from advice does not match
primitive return type for: " + method);
            }
            return retVal;
        }
        finally {
            if (target != null && !targetSource.isStatic()) {
                // Must have come from TargetSource.
                targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
                // Restore old proxy.
                AopContext.setCurrentProxy(oldProxy);
            }
        }
    }
```

主流程可以简述为：获取可以应用到此方法上的通知链（Interceptor Chain），如果有，则应用通知，并执行 joinpoint；如果没有，则直接反射执行 joinpoint。而这里的关键是通知链是如何获取的以及它又是如何执行的，下面逐一分析下。

首先，从上面的代码可以看到，通知链是通过 Advised.getInterceptorsAndDynamicInterceptionAdvice()这个方法来获取的，我们来看下这个方法的实现：

```
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, Class targetClass)
{
        MethodCacheKey cacheKey = new MethodCacheKey(method);
        List<Object> cached = this.methodCache.get(cacheKey);
        if (cached == null) {
            cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
                    this, method, targetClass);
            this.methodCache.put(cacheKey, cached);
        }
        return cached;
    }
```

可以看到实际的获取工作其实是由 AdvisorChainFactory. getInterceptorsAndDynamicInterceptionAdvice()这个方法来完成的，获取到的结果会被缓存。

下面来分析下这个方法的实现：

```
/**
    * 从提供的配置实例 config 中获取 advisor 列表，遍历处理这些 advisor.如果是 IntroductionAdvisor，
    * 则判断此 Advisor 能否应用到目标类 targetClass 上.如果是 PointcutAdvisor，则判断
```

```
    * 此 Advisor 能否应用到目标方法 method 上.将满足条件的 Advisor 通过 AdvisorAdaptor 转化成 Interceptor
列表返回.
    */
    public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
            Advised config, Method method, Class targetClass) {

        // This is somewhat tricky... we have to process introductions first,
        // but we need to preserve order in the ultimate list.
        List<Object> interceptorList = new ArrayList<Object>(config.getAdvisors().length);
        //查看是否包含 IntroductionAdvisor
        boolean hasIntroductions = hasMatchingIntroductions(config, targetClass);
        //这里实际上注册一系列 AdvisorAdapter,用于将 Advisor 转化成 MethodInterceptor
        AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
        for (Advisor advisor : config.getAdvisors()) {
            if (advisor instanceof PointcutAdvisor) {
                // Add it conditionally.
                PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
                if (config.isPreFiltered() ||
pointcutAdvisor.getPointcut().getClassFilter().matches(targetClass)) {
                    //这个地方这两个方法的位置可以互换下
                   //将 Advisor 转化成 Interceptor
                    MethodInterceptor[] interceptors = registry.getInterceptors(advisor);

                    //检查当前 advisor 的 pointcut 是否可以匹配当前方法
                    MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
                    if (MethodMatchers.matches(mm, method, targetClass, hasIntroductions)) {
                        if (mm.isRuntime()) {
                            // Creating a new object instance in the getInterceptors() method
                            // isn't a problem as we normally cache created chains.
                            for (MethodInterceptor interceptor : interceptors) {
                                interceptorList.add(new
InterceptorAndDynamicMethodMatcher(interceptor, mm));
                            }
                        }
                        else {
                            interceptorList.addAll(Arrays.asList(interceptors));
                        }
                    }
                }
            }
            else if (advisor instanceof IntroductionAdvisor) {
                IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
                if (config.isPreFiltered() || ia.getClassFilter().matches(targetClass)) {
                    Interceptor[] interceptors = registry.getInterceptors(advisor);
```

```
                    interceptorList.addAll(Arrays.asList(interceptors));
                }
            }
            else {
                Interceptor[] interceptors = registry.getInterceptors(advisor);
                interceptorList.addAll(Arrays.asList(interceptors));
            }
        }
        return interceptorList;
    }
```

这个方法执行完成后，Advised 中配置能够应用到连接点或者目标类的 Advisor 全部被转化成了 MethodInterceptor.

接下来我们再看下得到的拦截器链是怎么起作用的。

```
......

// Check whether we have any advice. If we don't, we can fallback on direct
// reflective invocation of the target, and avoid creating a MethodInvocation.
//如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 method.invoke(target, args)
if (chain.isEmpty()) {
     // We can skip creating a MethodInvocation: just invoke the target directly
     // Note that the final invoker must be an InvokerInterceptor so we know it does
     // nothing but a reflective operation on the target, and no hot swapping or fancy proxying.
     retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
}
else {
     // We need to create a method invocation...
     //创建 MethodInvocation
     invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
     // Proceed to the joinpoint through the interceptor chain.
     retVal = invocation.proceed();
}

......
```

从这段代码可以看出，如果得到的拦截器链为空，则直接反射调用目标方法，否则创建 MethodInvocation，调用其 proceed 方法，触发拦截器链的执行，来看下具体代码

```
public Object proceed() throws Throwable {
        //  We start with an index of -1 and increment early.
        //如果 Interceptor 执行完了，则执行 joinPoint
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
            return invokeJoinpoint();
        }
```

```
        Object interceptorOrInterceptionAdvice =
                this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);

        //如果要动态匹配 joinPoint
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
            // Evaluate dynamic method matcher here: static part will already have
            // been evaluated and found to match.
            InterceptorAndDynamicMethodMatcher dm =
                    (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
            //动态匹配：运行时参数是否满足匹配条件
            if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
                //执行当前 Intercetpor
                return dm.interceptor.invoke(this);
            }
            else {
                // Dynamic matching failed.
                // Skip this interceptor and invoke the next in the chain.
                //动态匹配失败时,略过当前 Intercetpor,调用下一个 Interceptor
                return proceed();
            }
        }
        else {
            // It's an interceptor, so we just invoke it: The pointcut will have
            // been evaluated statically before this object was constructed.
            //执行当前 Intercetpor
            return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
    }
```

5.7、Spring JDBC 设计原理及二次开发

使用 Spring 进行基本的 JDBC 访问数据库有多种选择。Spring 至少提供了三种不同的工作模式：JdbcTemplate，一个在 Spring2.5 中新提供的 SimpleJdbc 类能够更好的处理数据库元数据；还有一种称之为 RDBMS Object 的风格的面向对象封装方式，有点类似于 JDO 的查询设计。我们在这里简要列举你采取某一种工作方式的主要理由. 不过请注意，即使你选择了其中的一种工作模式，你依然可以在你的代码中混用其他任何一种模式以获取其带来的好处和优势。所有的工作模式都必须要求 JDBC 2.0 以上的数据库驱动的支持，其中一些高级的功能可能需要 JDBC 3.0 以上的数据库驱动支持。

JdbcTemplate - 这是经典的也是最常用的 Spring 对于 JDBC 访问的方案。这也是最低级别的封装，其他的工作模式事实上在底层使用了 JdbcTemplate 作为其底层的实现基础。JdbcTemplate 在 JDK 1.4 以上的环境上工作得很好。

NamedParameterJdbcTemplate - 对 JdbcTemplate 做了封装，提供了更加便捷的基于命名参数的使用方式而不是传统的 JDBC 所使用的“?”作为参数的占位符。这种方式在你需要为某个 SQL 指定许多个参数时，显得更加直观而易用。该特性必须工作在 JDK 1.4 以上。

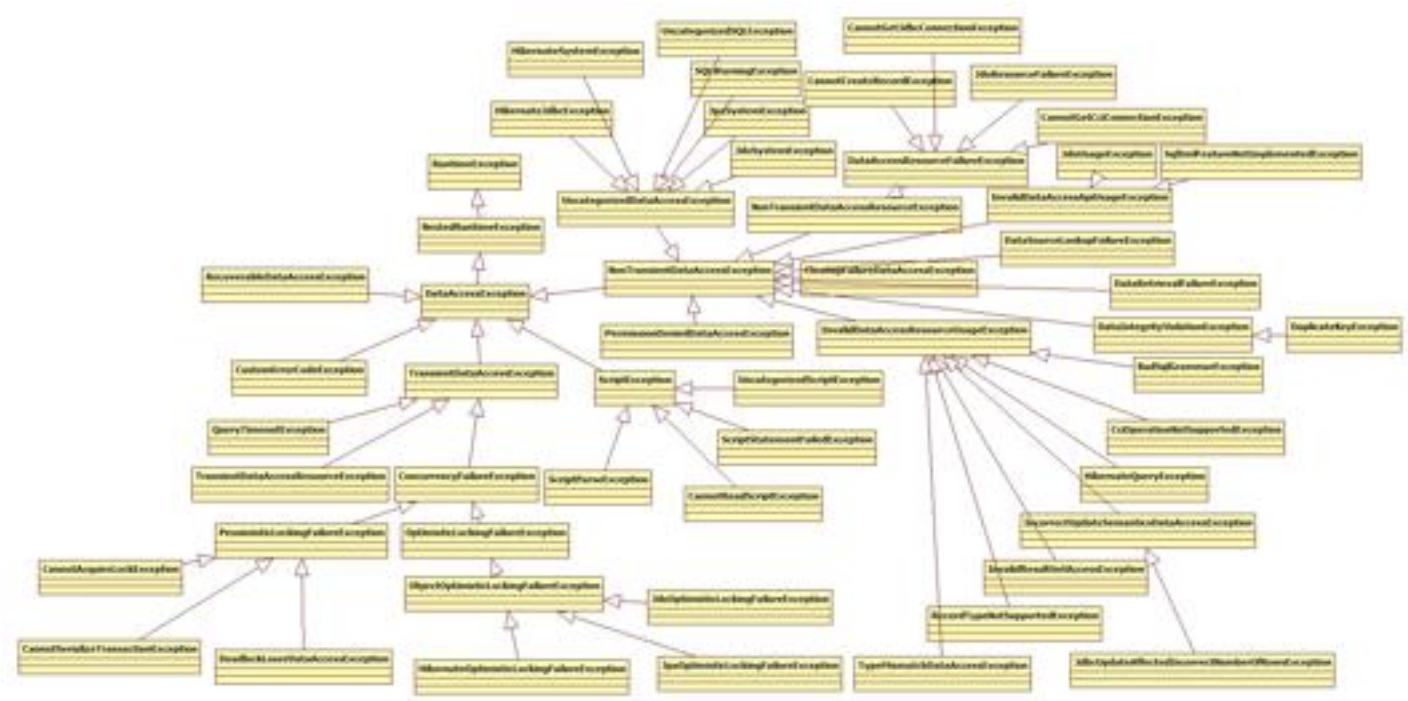
SimpleJdbcTemplate - 这个类结合了 JdbcTemplate 和 NamedParameterJdbcTemplate 的最常用的功能，同时它也利用了一些 Java 5 的特性所带来的优势，例如泛型、varargs 和 autoboxing 等，从而提供了更加简便的 API 访问方式。需要工作在 Java 5 以上的环境中。

SimpleJdbcInsert 和 SimpleJdbcCall - 这两个类可以充分利用数据库元数据的特性来简化配置。通过使用这两个类进行编程，你可以仅仅提供数据库表名或者存储过程的名称以及一个 Map 作为参数。其中 Map 的 key 需要与数据库表中的字段保持一致。这两个类通常和 SimpleJdbcTemplate 配合使用。这两个类需要工作在 JDK 5 以上，同时数据库需要提供足够的元数据信息。

RDBMS 对象包括 MappingSqlQuery, SqlUpdate and StoredProcedure - 这种方式允许你在初始化你的数据访问层时创建可重用并且线程安全的对象。该对象在你定义了你的查询语句，声明查询参数并编译相应的 Query 之后被模型化。一旦模型化完成，任何执行函数就可以传入不同的参数对之进行多次调用。这种方式需要工作在 JDK 1.4 以上。

1. 异常处理

异常结构如下：



SQLExceptionTranslator 是一个接口，如果你需要在 SQLException 和 org.springframework.dao.DataAccessException 之间作转换，那么必须实现该接口。 转换器类的实现可以采用一般通用的做法(比如使用 JDBC 的 SQLState code)，如果为了使转换更准确，也可以进行定制（比如使用 Oracle 的 error code）。

SQLErrorCodeSQLExceptionTranslator 是 SQLExceptionTranslator 的默认实现。 该实现使用指定数据库厂商的 error code，比采用 SQLState 更精确。转换过程基于一个 JavaBean（类型为 SQLErrorCodes）中的 error code。 这个 JavaBean 由 SQLErrorCodesFactory 工厂类创建，其中的内容来自于 “sql-error-codes.xml”配置文件。该文件中的数据库厂商代码基于 Database MetaData 信息中的 DatabaseProductName，从而配合当前数据库的使用。

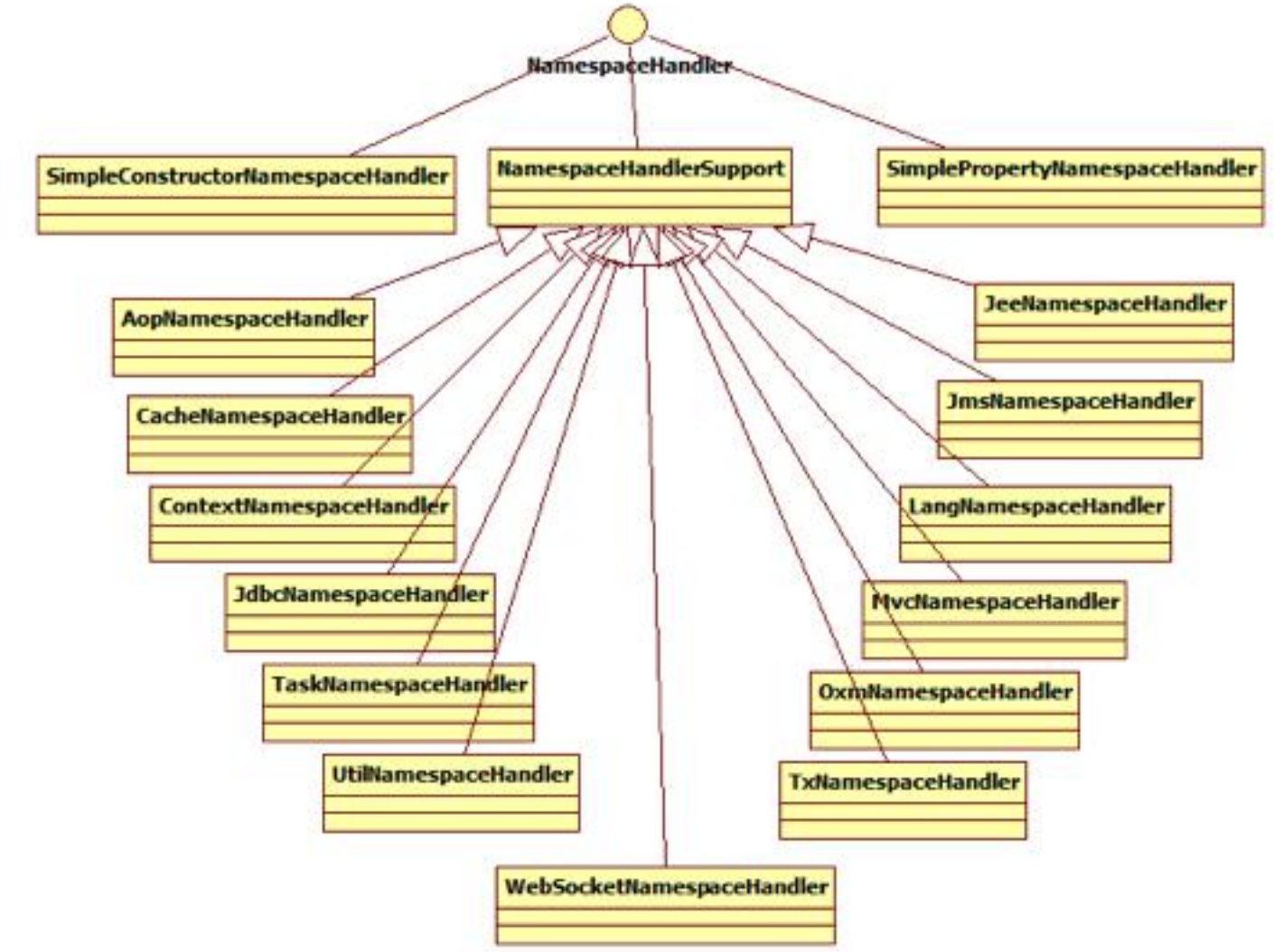
SQLErrorCodeSQLExceptionTranslator 使用以下的匹配规则：

首先检查是否存在完成定制转换的子类实现。通常 SQLErrorCodeSQLExceptionTranslator 这个类可以作为一个具体类使用，不需要进行定制，那么这个规则将不适用。

接着将 SQLException 的 error code 与错误代码集中的 error code 进行匹配。默认情况下错误代码集将从 SQLErrorCodesFactory 取得。 错误代码集来自 classpath 下的 sql-error-codes.xml 文件，它们将与数据库 metadata 信息中的 database name 进行映射。使用 fallback 翻译器。SQLStateSQLExceptionTranslator 类是缺省的 fallback 翻译器。

2. config 模块

NamespaceHandler 接口，DefaultBeanDefinitionDocumentReader 使用该接口来处理在 spring xml 配置文件中自定义的命名空间。



在 jdbc 模块，我们使用 JdbcNamespaceHandler 来处理 jdbc 配置的命名空间，其代码如下：

```
public class JdbcNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionParser("embedded-database", new EmbeddedDatabaseBeanDefinitionParser());
        registerBeanDefinitionParser("initialize-database", new InitializeDatabaseBeanDefinitionParser());
    }
```

其中，EmbeddedDatabaseBeanDefinitionParser 继承了 AbstractBeanDefinitionParser，解析<embedded-database>元素，并使用 EmbeddedDatabaseFactoryBean 创建一个 BeanDefinition。顺便介绍一下用到的软件包 org.w3c.dom。

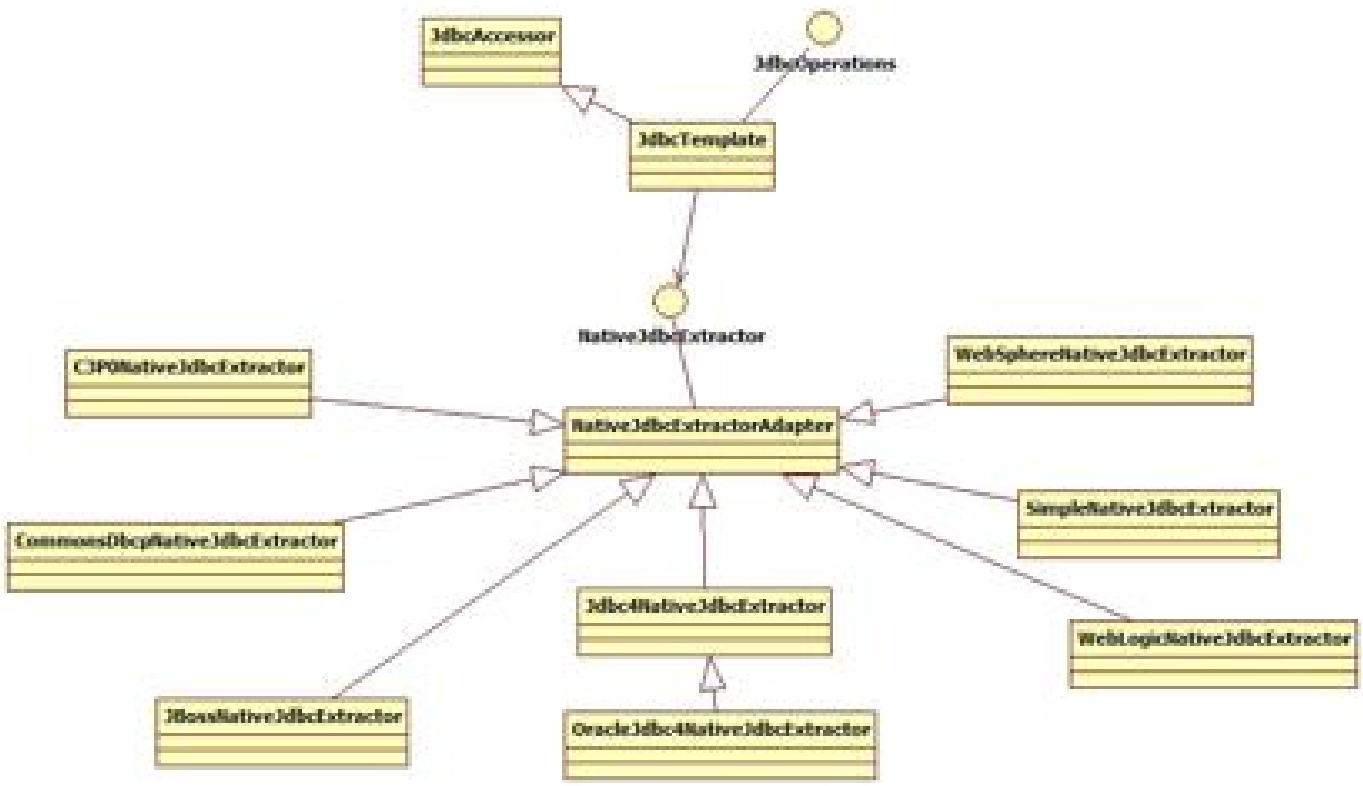
软件包 org.w3c.dom:为文档对象模型 (DOM) 提供接口，该模型是 Java API for XML Processing 的组件 API。该 Document Object Model Level 2 Core API 允许程序动态访问和更新文档的内容和结构。

Attr：Attr 接口表示 Element 对象中的属性。
CDATASection： CDATA 节用于转义文本块，该文本块包含的字符如果不转义则会被视为标记。
CharacterData： CharacterData 接口使用属性集合和用于访问 DOM 中字符数据的方法扩展节点。
Comment： 此接口继承自 CharacterData 表示注释的内容，即起始 '<!--' 和结束 '-->' 之间的所有字符。
Document： Document 接口表示整个 HTML 或 XML 文档。
DocumentFragment： DocumentFragment 是“轻量级”或“最小”Document 对象。
DocumentType： 每个 Document 都有 doctype 属性，该属性的值可以为 null，也可以为 DocumentType 对象。
DOMConfiguration： 该 DOMConfiguration 接口表示文档的配置，并维护一个可识别的参数表。
DOMError： DOMError 是一个描述错误的接口。
DOMErrorHandler： DOMErrorHandler 是在报告处理 XML 数据时发生的错误或在进行某些其他处理（如验证文档）时 DOM 实现可以调用的回调接口。
DOMImplementation： DOMImplementation 接口为执行独立于文档对象模型的任何特定实例的操作提供了许多方法。
DOMImplementationList： DOMImplementationList 接口提供对 DOM 实现的有序集合的抽象，没有定义或约束如何实现此集合。
DOMImplementationSource： 此接口允许 DOM 实现程序根据请求的功能和版本提供一个或多个实现，如下所述。
DOMLocator： DOMLocator 是一个描述位置（如发生错误的位置）的接口。
DOMStringList： DOMStringList 接口提供对 DOMString 值的有序集合的抽象，没有定义或约束此集合是如何实现的。
Element： Element 接口表示 HTML 或 XML 文档中的一个元素。
Entity： 此接口表示在 XML 文档中解析和未解析的已知实体。
EntityReference： EntityReference 节点可以用来在树中表示实体引用。
NamedNodeMap： 实现 NamedNodeMap 接口的对象用于表示可以通过名称访问的节点的集合。
NameList NameList 接口提供对并行的名称和名称空间值对（可以为 null 值）的有序集合的抽象，无需定义或约束如何实现此集合。
Node： 该 Node 接口是整个文档对象模型的主要数据类型。
NodeList： NodeList 接口提供对节点的有序集合的抽象，没有定义或约束如何实现此集合。
Notation： 此接口表示在 DTD 中声明的表示法。
ProcessingInstruction： ProcessingInstruction 接口表示“处理指令”，该指令作为一种在文档的文本中保持特定于处理器的信息的方法在 XML 中使用。

Text： 该 Text 接口继承自 CharacterData，并且表示 Element 或 Attr 的文本内容（在 XML 中称为 字符数据）。
TypeInfo： TypeInfo 接口表示从 Element 或 Attr 节点引用的类型，用与文档相关的模式指定。
UserDataHandler： 当使用 Node.setUserData() 将一个对象与节点上的键相关联时，当克隆、导入或重命名该对象关联的节点时应用程序可以提供调用的处理程序。

3. core 模块

3.1 NativeJdbcExtractor 从线程池中的封装的对象中提取出本地的 jdbc 对象，其结构如下：



其实现原理如下(以 c3po 为例)：

```
/**
     * Retrieve the Connection via C3PO's {@code rawConnectionOperation} API,
     * using the {@code getRawConnection} as callback to get access to the
     * raw Connection (which is otherwise not directly supported by C3PO).
     * @see #getRawConnection
     */
    @Override
    protected Connection doGetNativeConnection(Connection con) throws SQLException {
        if (con instanceof C3POProxyConnection) {
            C3POProxyConnection cpCon = (C3POProxyConnection) con;
            try {
                return (Connection) cpCon.rawConnectionOperation(
```

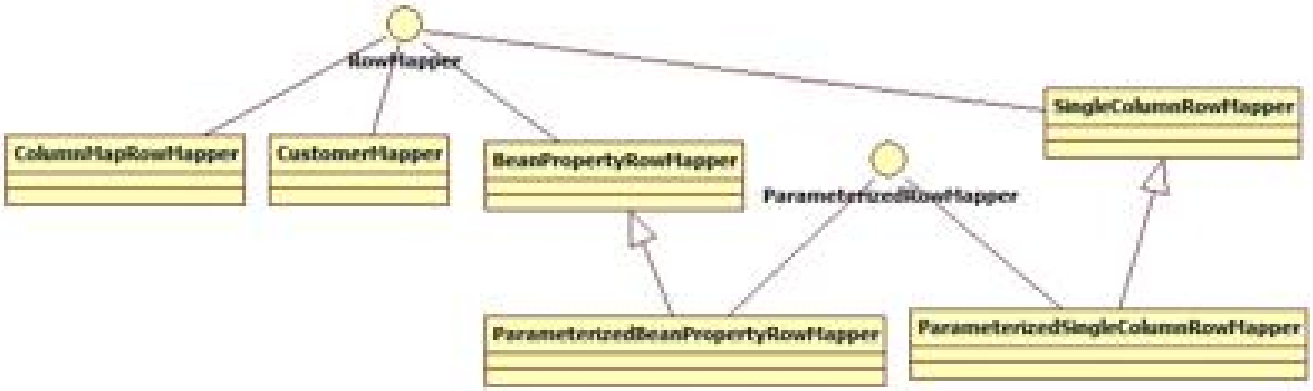
```
                        this.getRawConnectionMethod, null, new Object[]
{C3POProxyConnection.RAW_CONNECTION});
                }
                catch (SQLException ex) {
                    throw ex;
                }
                catch (Exception ex) {
                    ReflectionUtils.handleReflectionException(ex);
                }
            }
            return con;

        }
```

上述代码通过调用 C3P0 的 rawConnectionOperation api 来获取 Connection，使用 getRawConnection 的回调方法来获取原生的 Connection(C3P0 不直接支持原生的 Connection)。NativeJdbcExtractorAdapter 是 NativeJdbcExtractor 的一个简单实现，它的 getNativeConnection()方法检查 ConnectionProxy 链，并且代理 doGetNativeConnection 方法。Spring 的 TransactionAwareDataSourceProxy 和 LazyConnectionDataSourceProxy 使用 ConnectionProxy。目标 Connection 置于本地连接池中，被子类实现的 doGetNativeConnection 的方法去掉封装获取到原生的 Connection。其实现如下：

```
@Override
    public Connection getNativeConnection(Connection con) throws SQLException {
        if (con == null) {
            return null;
        }
        Connection targetCon = DataSourceUtils.getTargetConnection(con);
        Connection nativeCon = doGetNativeConnection(targetCon);
        if (nativeCon == targetCon) {
            // We haven't received a different Connection, so we'll assume that there's
            // some additional proxying going on. Let's check whether we get something
            // different back from the DatabaseMetaData.getConnection() call.
            DatabaseMetaData metaData = targetCon.getMetaData();
            // The following check is only really there for mock Connections
            // which might not carry a DatabaseMetaData instance.
            if (metaData != null) {
                Connection metaCon = metaData.getConnection();
                if (metaCon != null && metaCon != targetCon) {
                    // We've received a different Connection there:
                    // Let's retry the native extraction process with it.
                    nativeCon = doGetNativeConnection(metaCon);
                }
            }
        }
        return nativeCon;
```

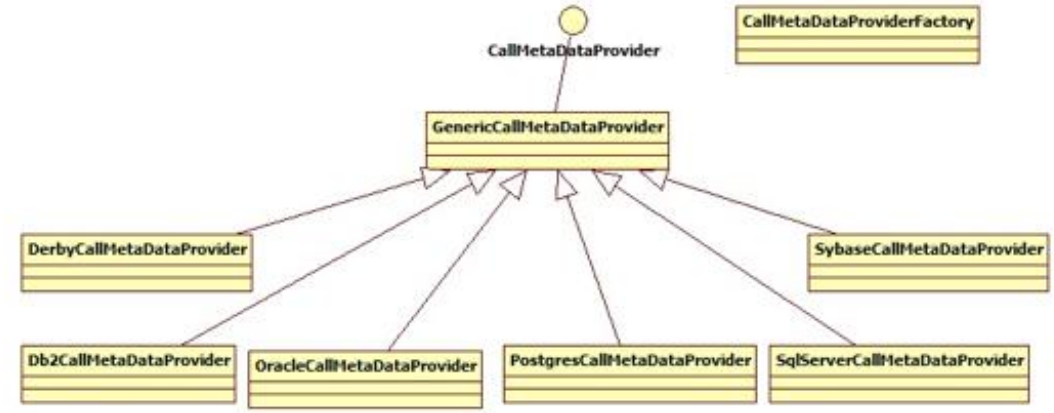
```
}
```

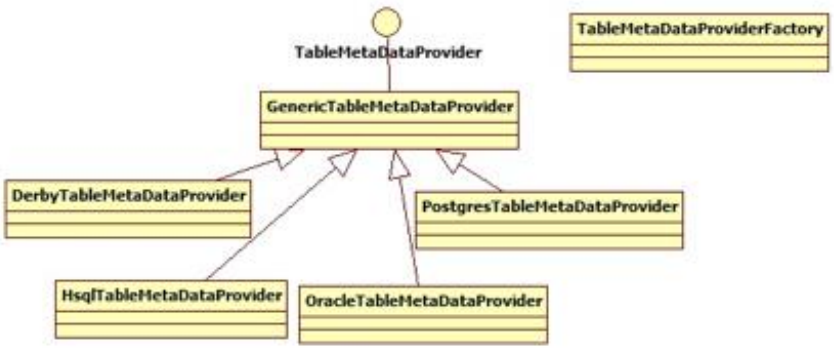
3.2 RowMapper



3.3 元数据 metaData 模块

本节中 spring 应用到工厂模式，结合代码可以更具体了解。





CallMetaDataProviderFactory 创建 CallMetaDataProvider 的工厂类，其代码如下：

```
    /** List of supported database products for procedure calls */
    public static final List<String> supportedDatabaseProductsForProcedures = Arrays.asList(
            "Apache Derby",
            "DB2",
            "MySQL",
            "Microsoft SQL Server",
            "Oracle",
            "PostgreSQL",
            "Sybase"
        );
    /** List of supported database products for function calls */
    public static final List<String> supportedDatabaseProductsForFunctions = Arrays.asList(
            "MySQL",
            "Microsoft SQL Server",
            "Oracle",
            "PostgreSQL"
        );

    /**
     * Create a CallMetaDataProvider based on the database metadata
     * @param dataSource used to retrieve metadata
     * @param context the class that holds configuration and metadata
     * @return instance of the CallMetaDataProvider implementation to be used
     */
    static public CallMetaDataProvider createMetaDataProvider(DataSource dataSource, final
CallMetaDataContext context) {
        try {
            return (CallMetaDataProvider) JdbcUtils.extractDatabaseMetaData(dataSource, new
DatabaseMetaDataCallback() {
                @Override
                public Object processMetaData(DatabaseMetaData databaseMetaData) throws SQLException,
MetaDataAccessException {
                    String databaseProductName =
JdbcUtils.commonDatabaseName(databaseMetaData.getDatabaseProductName());
                    boolean accessProcedureColumnMetaData = context.isAccessCallParameterMetaData();
                    if (context.isFunction()) {
                        if (!supportedDatabaseProductsForFunctions.contains(databaseProductName)) {
                            if (logger.isWarnEnabled()) {
                                logger.warn(databaseProductName + " is not one of the databases fully
supported for function calls " +
                                        "-- supported are: " + supportedDatabaseProductsForFunctions);
                            }
                            if (accessProcedureColumnMetaData) {
                                logger.warn("Metadata processing disabled - you must specify all parameters
```

```
explicitly");
                                    accessProcedureColumnMetaData = false;
                                }
                            }
                        }
                        else {
                            if (!supportedDatabaseProductsForProcedures.contains(databaseProductName)) {
                                if (logger.isWarnEnabled()) {
                                    logger.warn(databaseProductName + " is not one of the databases fully
supported for procedure calls " +
                                            "-- supported are: " + supportedDatabaseProductsForProcedures);
                                }
                                if (accessProcedureColumnMetaData) {
                                    logger.warn("Metadata processing disabled - you must specify all parameters
explicitly");
                                    accessProcedureColumnMetaData = false;
                                }
                            }
                        }

                        CallMetaDataProvider provider;
                        if ("Oracle".equals(databaseProductName)) {
                            provider = new OracleCallMetaDataProvider(databaseMetaData);
                        }
                        else if ("DB2".equals(databaseProductName)) {
                            provider = new Db2CallMetaDataProvider((databaseMetaData));
                        }
                        else if ("Apache Derby".equals(databaseProductName)) {
                            provider = new DerbyCallMetaDataProvider((databaseMetaData));
                        }
                        else if ("PostgreSQL".equals(databaseProductName)) {
                            provider = new PostgresCallMetaDataProvider((databaseMetaData));
                        }
                        else if ("Sybase".equals(databaseProductName)) {
                            provider = new SybaseCallMetaDataProvider((databaseMetaData));
                        }
                        else if ("Microsoft SQL Server".equals(databaseProductName)) {
                            provider = new SqlServerCallMetaDataProvider((databaseMetaData));
                        }
                        else {
                            provider = new GenericCallMetaDataProvider(databaseMetaData);
                        }
                        if (logger.isDebugEnabled()) {
                            logger.debug("Using " + provider.getClass().getName());
```

```
                        }
                        provider.initializeWithMetaData(databaseMetaData);
                        if (accessProcedureColumnMetaData) {
                            provider.initializeWithProcedureColumnMetaData(
                                    databaseMetaData, context.getCatalogName(), context.getSchemaName(),
context.getProcedureName());
                        }
                        return provider;
                    }
                });
        }
        catch (MetaDataAccessException ex) {
            throw new DataAccessResourceFailureException("Error retreiving database metadata", ex);
        }


    }
```

TableMetaDataProviderFactory 创建 TableMetaDataProvider 工厂类，其创建过程如下：

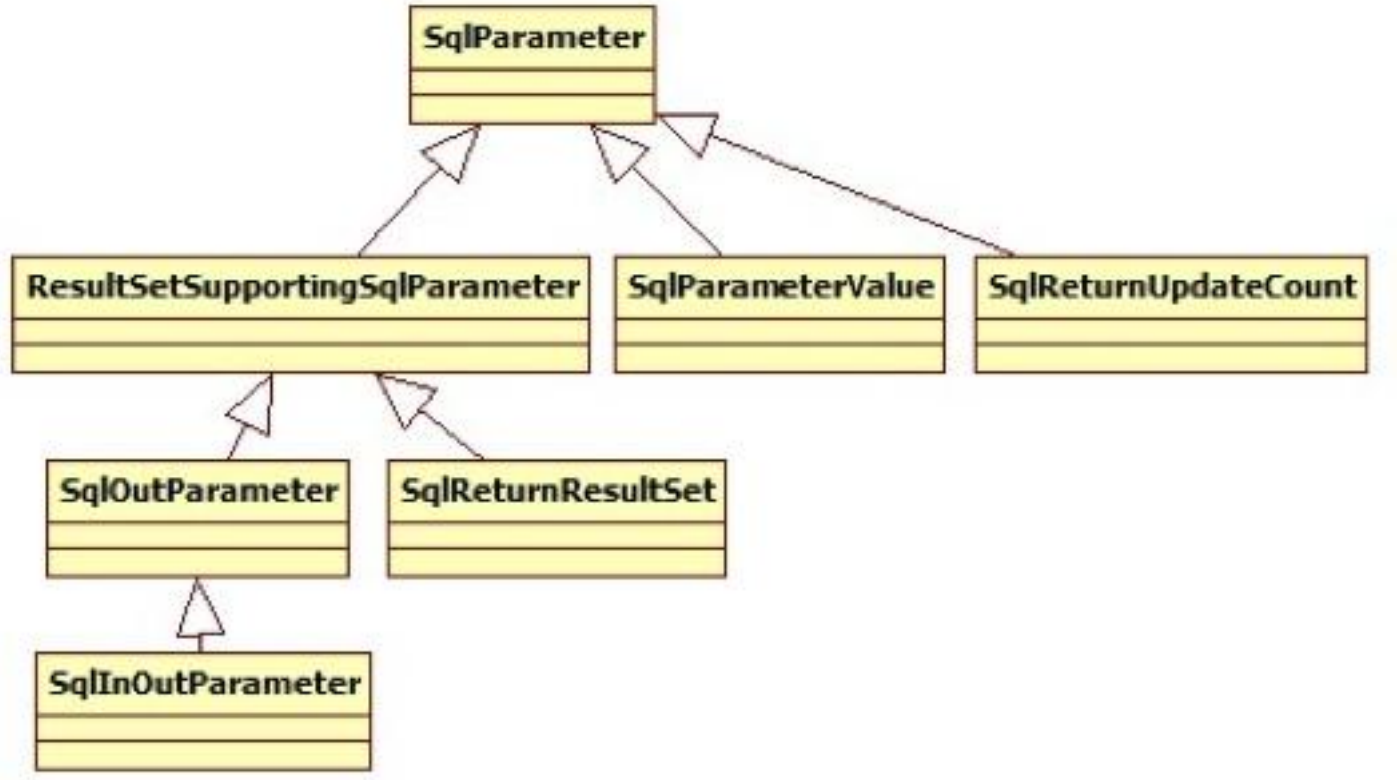
```
/**
     * Create a TableMetaDataProvider based on the database metedata
     * @param dataSource used to retrieve metedata
     * @param context the class that holds configuration and metedata
     * @param nativeJdbcExtractor the NativeJdbcExtractor to be used
     * @return instance of the TableMetaDataProvider implementation to be used
     */
    public static TableMetaDataProvider createMetaDataProvider(DataSource dataSource,
                final TableMetaDataContext context, final NativeJdbcExtractor nativeJdbcExtractor) {
        try {
            return (TableMetaDataProvider) JdbcUtils.extractDatabaseMetaData(dataSource,
                    new DatabaseMetaDataCallback() {
                        @Override
                        public Object processMetaData(DatabaseMetaData databaseMetaData) throws
SQLException {
                            String databaseProductName =

JdbcUtils.commonDatabaseName(databaseMetaData.getDatabaseProductName());
                            boolean accessTableColumnMetaData = context.isAccessTableColumnMetaData();
                            TableMetaDataProvider provider;
                            if ("Oracle".equals(databaseProductName)) {
                                provider = new OracleTableMetaDataProvider(databaseMetaData,
                                        context.isOverrideIncludeSynonymsDefault());
                            }
```

```
                        else if ("HSQL Database Engine".equals(databaseProductName)) {
                            provider = new HsqlTableMetaDataProvider(databaseMetaData);
                        }
                        else if ("PostgreSQL".equals(databaseProductName)) {
                            provider = new PostgresTableMetaDataProvider(databaseMetaData);
                        }
                        else if ("Apache Derby".equals(databaseProductName)) {
                            provider = new DerbyTableMetaDataProvider(databaseMetaData);
                        }
                        else {
                            provider = new GenericTableMetaDataProvider(databaseMetaData);
                        }
                        if (nativeJdbcExtractor != null) {
                            provider.setNativeJdbcExtractor(nativeJdbcExtractor);
                        }
                        if (logger.isDebugEnabled()) {
                            logger.debug("Using " + provider.getClass().getSimpleName());
                        }
                        provider.initializeWithMetaData(databaseMetaData);
                        if (accessTableColumnMetaData) {
                            provider.initializeWithTableColumnMetaData(databaseMetaData,
context.getCatalogName(),
                                    context.getSchemaName(), context.getTableName());
                        }
                        return provider;
                    }
                });
        }
        catch (MetaDataAccessException ex) {
            throw new DataAccessResourceFailureException("Error retrieving database metadata", ex);
        }

    }
```

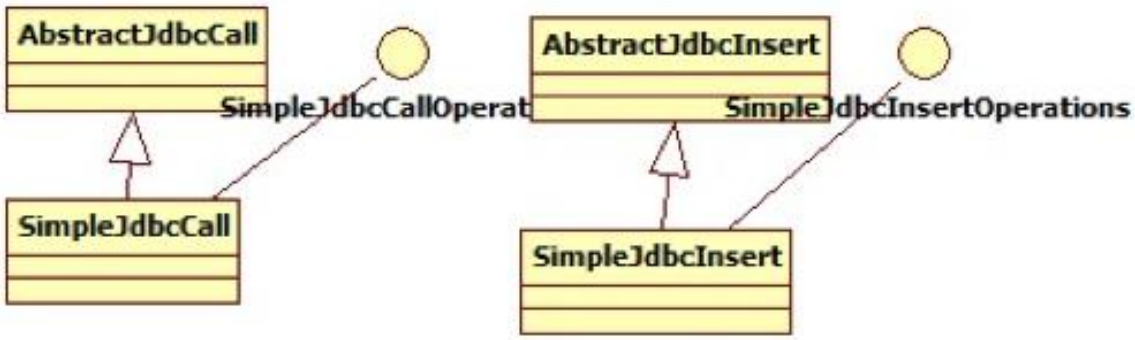
3.4 使用 SqlParameterSource 提供参数值

使用 Map 来指定参数值有时候工作得非常好，但是这并不是最简单的使用方式。Spring 提供了一些其他的 SqlParameterSource 实现类来指定参数值。 我们首先可以看看 BeanPropertySqlParameterSource 类，这是一个非常简便的指定参数的实现类，只要你有一个符合 JavaBean 规范的类就行了。它将使用其中的 getter 方法来获取参数值。

SqlParameter 封装了定义 sql 参数的对象。CallableStateMentCallback，PrePareStateMentCallback，StateMentCallback，ConnectionCallback 回调类分别对应 JdbcTemplate 中的不同处理方法。



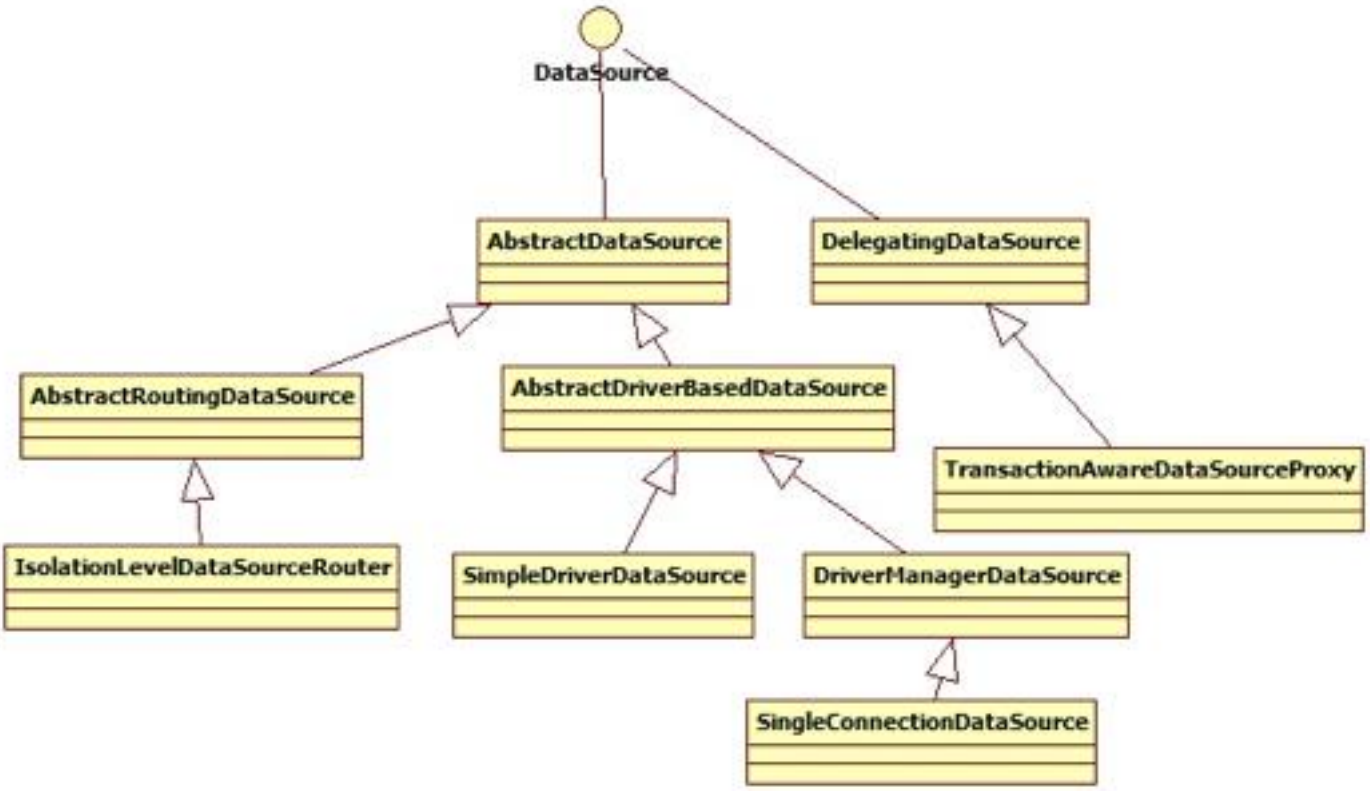
3.5 simple 实现



4. DataSource

spring 通过 DataSource 获取数据库的连接。Datasource 是 jdbc 规范的一部分，它通过 ConnectionFactory 获取。一个容器和框架可以在应用代码层中隐藏连接池和事务管理。

当使用 spring 的 jdbc 层，你可以通过 JNDI 来获取 DataSource，也可以通过你自己配置的第三方连接池实现来获取。流行的第三方实现由 apache Jakarta Commons dbcp 和 c3p0.

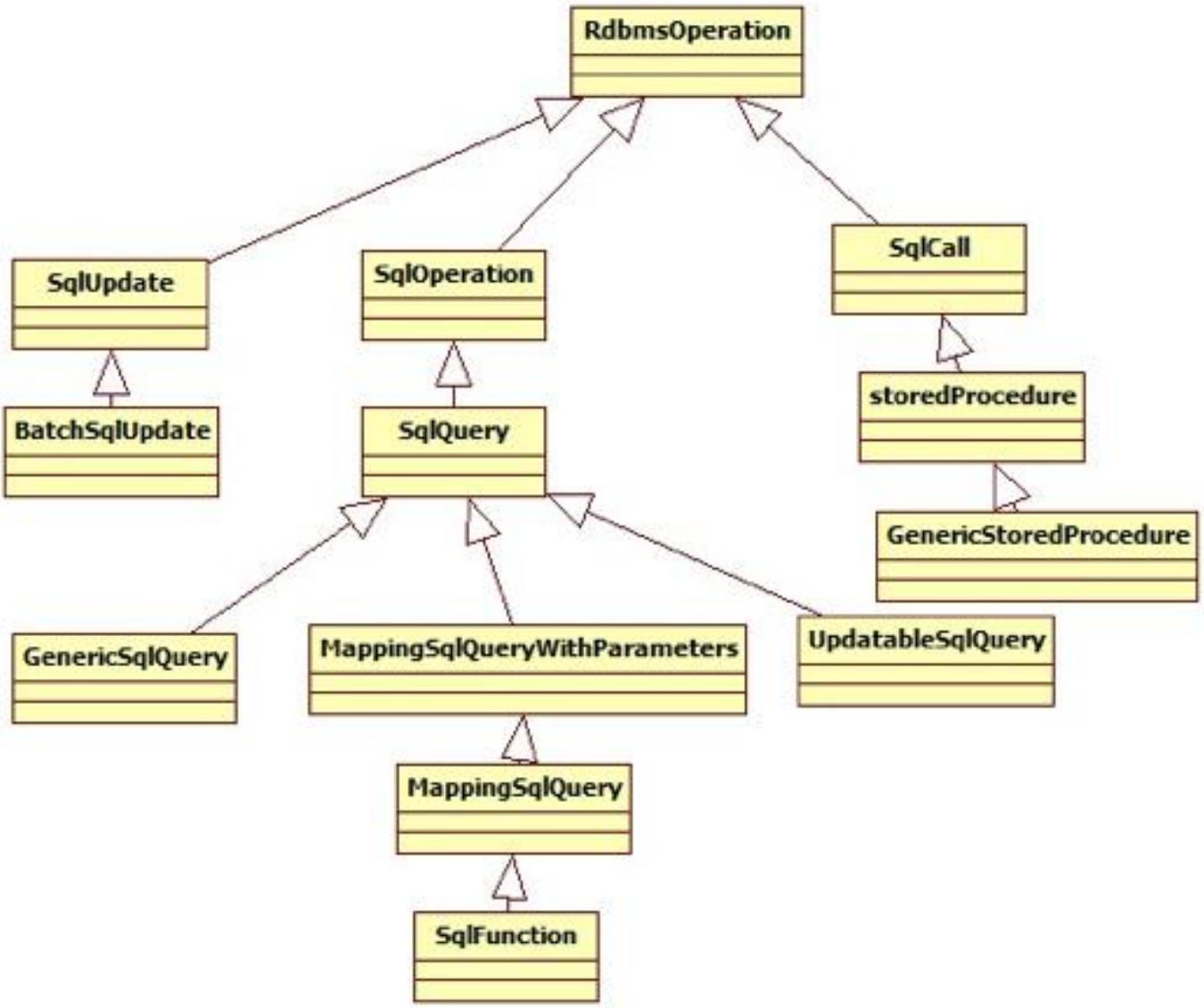


TransactionAwareDataSourceProxy 作为目标 DataSource 的一个代理， 在对目标 DataSource 包装的同时，还增加了 Spring 的事务管理能力， 在这一点上，这个类的功能非常像 J2EE 服务器所提供的事务化的 JNDI DataSource。

Note
该类几乎很少被用到，除非现有代码在被调用的时候需要一个标准的 JDBC DataSource 接口实现作为参数。 这种情况下，这个类可以使现有代码参与 Spring 的事务管理。通常最好的做法是使用更高层的抽象 来对数据源进行管理，比如 JdbcTemplate 和 DataSourceUtils 等等。

注意：DriverManagerDataSource 仅限于测试使用，因为它没有提供池的功能，这会导致在多个请求获取连接时性能很差。

5. object 模块



6. JdbcTemplate 是 core 包的核心类。

它替我们完成了资源的创建以及释放工作，从而简化了我们对 JDBC 的使用。 它还可以帮助我们避免一些常见的错误，比如忘记关闭数据库连接。 JdbcTemplate 将完成 JDBC 核心处理流程，比如 SQL 语句的创建、执行，而把 SQL 语句的生成以及查询结果的提取工作留给我们的应用代码。 它可以完成 SQL 查询、更新以及调用存储过程，可以对 ResultSet 进行遍历并加以提取。 它还可以捕获 JDBC 异常并将其转换成 org.springframework.dao 包中定义的，通用的，信息更丰富的异常。

使用 JdbcTemplate 进行编码只需要根据明确定义的一组契约来实现回调接口。 PreparedStatementCreator 回调接口通过给定的 Connection 创建一个 PreparedStatement，包含 SQL 和任何相关的参数。 CallableStatementCreateor 实现同样的处理，只不过它创建的是 CallableStatement。 RowCallbackHandler 接口则从数据集的每一行中提取值。

我们可以在 DAO 实现类中通过传递一个 DataSource 引用来完成 JdbcTemplate 的实例化，也可以在 Spring 的 IoC 容器中配置

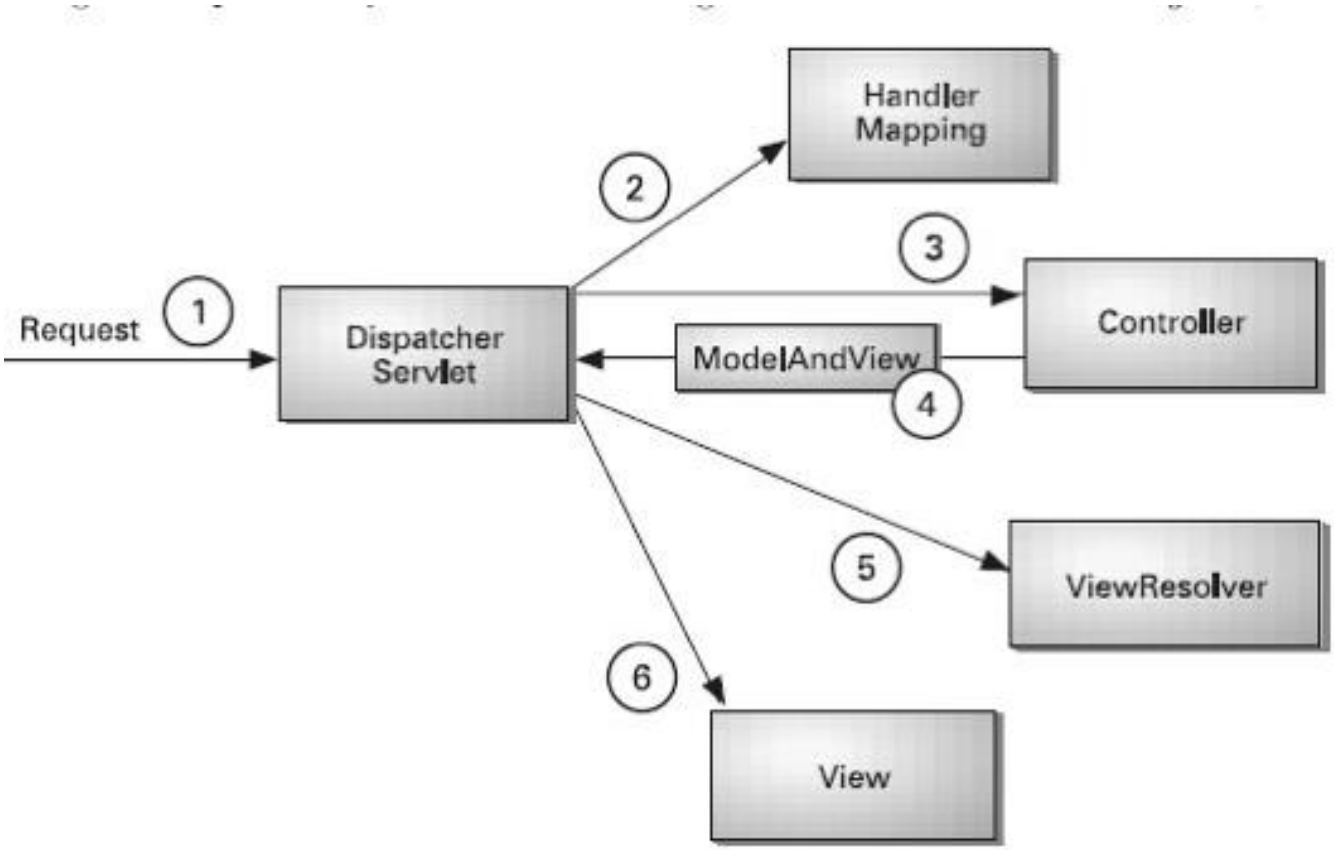
一个 JdbcTemplate 的 bean 并赋予 DAO 实现类作为一个实例。 需要注意的是 DataSource 在 Spring 的 IoC 容器中总是配制成一个 bean，第一种情况下，DataSource bean 将传递给 service，第二种情况下 DataSource bean 传递给 JdbcTemplate bean。

7. NamedParameterJdbcTemplate 类为 JDBC 操作增加了命名参数的特性支持，而不是传统的使用（'?'）作为参数的占位符。NamedParameterJdbcTemplate 类对 JdbcTemplate 类进行了封装， 在底层，JdbcTemplate 完成了多数的工作。

5.8、 Spring MVC 框架设计原理及手写实现

5.8.1 Spring MVC 请求处理流程

引用 spring in action 上的一张图来说明了 springmvc 的核心组件和请求处理流程:



①：DispatcherServlet 是 springmvc 中的前端控制器(front controller),负责接收 request 并将 request 转发给对应的处理组件.

②：HanlerMapping 是 springmvc 中完成 url 到 controller 映射的组件.DispatcherServlet 接收 request,然后从 HandlerMapping 查找处理 request 的 controller.

③：Cntroller 处理 request,并返回 ModelAndView 对象,Controller 是 springmvc 中负责处理 request 的组件(类似于 struts2 中的 Action),ModelAndView 是封装结果视图的组件.

④ ⑤ ⑥：视图解析器解析 ModelAndView 对象并返回对应的视图给客户端.

5.8.2 Spring MVC 的工作机制

在容器初始化时会建立所有 url 和 controller 的对应关系,保存到 Map<url,controller>中.tomcat 启动时会通知 spring 初始化容器(加载 bean 的定义信息和初始化所有单例 bean),然后 springmvc 会遍历容器中的 bean,获取每一个 controller 中的所有方法访问的 url,然后将 url 和 controller 保存到一个 Map 中;

这样就可以根据 request 快速定位到 controller,因为最终处理 request 的是 controller 中的方法,Map 中只保留了 url 和 controller 中的对应关系,所以要根据 request 的 url 进一步确认 controller 中的 method,这一步工作的原理就是拼接 controller 的 url(controller 上@RequestMapping 的值)和方法的 url(method 上@RequestMapping 的值),与 request 的 url 进行匹配,找到匹配的那个方法;

确定处理请求的 method 后,接下来的任务就是参数绑定,把 request 中参数绑定到方法的形式参数上,这一步是整个请求处理过程中最复杂的一个步骤。springmvc 提供了两种 request 参数与方法形参的绑定方法:

① 通过注解进行绑定,@RequestParam

② 通过参数名称进行绑定.

使用注解进行绑定,我们只要在方法参数前面声明@RequestParam("a"),就可以将 request 中参数 a 的值绑定到方法的该参数上.使用参数名称进行绑定的前提是必须要获取方法中参数的名称,Java 反射只提供了获取方法的参数的类型,并没有提供获取参数名称的方法.springmvc 解决这个问题的方法是用 asm 框架读取字节码文件,来获取方法的参数名称.asm 框架是一个字节码操作框架,关于 asm 更多介绍可以参考它的官网.个人建议,使用注解来完成参数绑定,这样就可以省去 asm 框架的读取字节码的操作.

5.8.3 Spring MVC 源码分析

我们根据工作机制中三部分来分析 springmvc 的源代码.

其一,ApplicationContext 初始化时建立所有 url 和 controller 类的对应关系(用 Map 保存);

其二,根据请求 url 找到对应的 controller,并从 controller 中找到处理请求的方法;

其三,request 参数绑定到方法的形参,执行方法处理请求,并返回结果视图.

第一步、建立 Map<urls,controller>的关系

我们首先看第一个步骤,也就是建立 Map<url,controller>关系的部分.第一部分的入口类为 ApplicationObjectSupport 的 setApplicationContext 方法.setApplicationContext 方法中核心部分就是初始化容器 initApplicationContext(context),子类 AbstractDetectingUrlHandlerMapping 实现了该方法,所以我们直接看子类中的初始化容器方法.

```
public void initApplicationContext() throws ApplicationContextException {
```

```
        super.initApplicationContext();
        detectHandlers();
    }
    /**
     * 建立当前ApplicationContext中的所有controller和url的对应关系
     */
    protected void detectHandlers() throws BeansException {
        if (logger.isDebugEnabled()) {
            logger.debug("Looking for URL mappings in application context: " + getApplicationContext());
        }
        // 获取ApplicationContext容器中所有bean的Name
        String[] beanNames = (this.detectHandlersInAncestorContexts ?
                BeanFactoryUtils.beanNamesForTypeIncludingAncestors(getApplicationContext(),
Object.class) :
                getApplicationContext().getBeanNamesForType(Object.class));

        // 遍历beanNames,并找到这些bean对应的url
        for (String beanName : beanNames) {
            // 找bean上的所有url(controller上的url+方法上的url),该方法由对应的子类实现
            String[] urls = determineUrlsForHandler(beanName);
            if (!ObjectUtils.isEmpty(urls)) {
                // 保存urls和beanName的对应关系,put it to Map<urls,beanName>,该方法在父类
AbstractUrlHandlerMapping中实现
                registerHandler(urls, beanName);
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Rejected bean name '" + beanName + "': no URL paths identified");
                }
            }
        }
    }
    /** 获取controller中所有方法的url,由子类实现,典型的模板模式 **/

protected abstract String[] determineUrlsForHandler(String beanName);
```

determineUrlsForHandler(String beanName)方法的作用是获取每个 controller 中的 url,不同的子类有不同的实现,这是一个典型的模板设计模式.因为开发中我们用的最多的就是用注解来配置 controller 中的 url,DefaultAnnotationHandlerMapping 是 AbstractDetectingUrlHandlerMapping 的子类,处理注解形式的 url 映射.所以我们这里以 DefaultAnnotationHandlerMapping 来进行分析.我们看 DefaultAnnotationHandlerMapping 是如何查 beanName 上所有映射的 url.

```
    /**
     * 获取controller中所有的url
```

```
     */
    protected String[] determineUrlsForHandler(String beanName) {
        // 获取ApplicationContext容器
        ApplicationContext context = getApplicationContext();
        //从容器中获取controller
         Class<?> handlerType = context.getType(beanName);
         // 获取controller上的@RequestMapping注解
        RequestMapping mapping = context.findAnnotationOnBean(beanName, RequestMapping.class);
        if (mapping != null) { // controller上有注解
            this.cachedMappings.put(handlerType, mapping);
              // 返回结果集
            Set<String> urls = new LinkedHashSet<String>();
              // controller的映射url
            String[] typeLevelPatterns = mapping.value();
            if (typeLevelPatterns.length > 0) { // url>0
                // 获取controller中所有方法及方法的映射url
                String[] methodLevelPatterns = determineUrlsForHandlerMethods(handlerType, true);
                for (String typeLevelPattern : typeLevelPatterns) {
                    if (!typeLevelPattern.startsWith("/")) {
                        typeLevelPattern = "/" + typeLevelPattern;
                    }
                    boolean hasEmptyMethodLevelMappings = false;
                    for (String methodLevelPattern : methodLevelPatterns) {
                        if (methodLevelPattern == null) {
                            hasEmptyMethodLevelMappings = true;
                        }
                        else {
                                // controller的映射url+方法映射的url
                            String combinedPattern = getPathMatcher().combine(typeLevelPattern,
methodLevelPattern);
                                // 保存到set集合中
                                 addUrlsForPath(urls, combinedPattern);
                        }
                    }
                    if (hasEmptyMethodLevelMappings ||

org.springframework.web.servlet.mvc.Controller.class.isAssignableFrom(handlerType)) {
                        addUrlsForPath(urls, typeLevelPattern);
                    }
                }
                  // 以数组形式返回controller上的所有url
                return StringUtils.toStringArray(urls);
            }
            else {
```

```
                // controller上的@RequestMapping映射url为空串,直接找方法的映射url
                return determineUrlsForHandlerMethods(handlerType, false);
            }
        } // controller上没@RequestMapping注解
        else if (AnnotationUtils.findAnnotation(handlerType, Controller.class) != null) {
            //  获取controller中方法上的映射url
            return determineUrlsForHandlerMethods(handlerType, false);
        }
        else {
            return null;
        }

    }
```

到这里 HandlerMapping 组件就已经建立所有 url 和 controller 的对应关系。

第二步、根据访问 url 找到对应的 controller 中处理请求的方法

下面我们开始分析第二个步骤,第二个步骤是由请求触发的,所以入口为 DispatcherServlet 的核心方法为 doService(),doService()中的核心逻辑由 doDispatch()实现,我们查看 doDispatch()的源代码.

```
/**  中央控制器,控制请求的转发  **/
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        int interceptorIndex = -1;

        try {
            ModelAndView mv;
            boolean errorView = false;
            try {
                  // 1.检查是否是文件上传的请求
            processedRequest = checkMultipart(request);

            // 2.取得处理当前请求的controller,这里也称为hanlder,处理器,第一个步骤的意义就在这里体现了.这里并不是直接返回controller,而是返回的HandlerExecutionChain请求处理器链对象,该对象封装了handler和interceptors.
                mappedHandler = getHandler(processedRequest, false);
                  //  如果handler为空,则返回404
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }
            //3.  获取处理request的处理器适配器handler adapter
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
            //  处理  last-modified  请求头
```

```
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                    if (logger.isDebugEnabled()) {
                        String requestUri = urlPathHelper.getRequestUri(request);
                        logger.debug("Last-Modified value for [" + requestUri + "] is: " + lastModified);
                    }
                    if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }

                // 4.拦截器的预处理方法
                HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();
                if (interceptors != null) {
                    for (int i = 0; i < interceptors.length; i++) {
                        HandlerInterceptor interceptor = interceptors[i];
                        if (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {
                            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response,
null);

                            return;
                        }
                        interceptorIndex = i;
                    }
                }

                // 5.实际的处理器处理请求,返回结果视图对象
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                //  结果视图对象的处理
                if (mv != null && !mv.hasView()) {
                    mv.setViewName(getDefaultViewName(request));
                }

                // 6.拦截器的后处理方法
                if (interceptors != null) {
                    for (int i = interceptors.length - 1; i >= 0; i--) {
                        HandlerInterceptor interceptor = interceptors[i];
                        interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);
                    }
                }
            }
            catch (ModelAndViewDefiningException ex) {
```

```
                logger.debug("ModelAndViewDefiningException encountered", ex);
                mv = ex.getModelAndView();
            }
            catch (Exception ex) {
                Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
                mv = processHandlerException(processedRequest, response, handler, ex);
                errorView = (mv != null);
            }


            if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);
                if (errorView) {
                    WebUtils.clearErrorRequestAttributes(request);
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Null ModelAndView returned to DispatcherServlet with name '" +
getServletName() +
                            "': assuming HandlerAdapter completed request handling");
                }
            }

            // 请求成功响应之后的方法
            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null);

        }
```

第 2 步:getHandler(processedRequest)方法实际上就是从 HandlerMapping 中找到 url 和 controller 的对应关系.这也就是第一个步骤:建立 Map<url,Controller>的意义.我们知道,最终处理 request 的是 controller 中的方法,我们现在只是知道了 controller,还要进一步确认 controller 中处理 request 的方法.由于下面的步骤和第三个步骤关系更加紧密,直接转到第三个步骤.

第三步、反射调用处理请求的方法，返回结果视图

上面的方法中,第 2 步其实就是从第一个步骤中的 Map<urls,beanName>中取得 controller,然后经过拦截器的预处理方法,到最核心的部分--第 5 步调用 controller 的方法处理请求.在第 2 步中我们可以知道处理 request 的 controller,第 5 步就是要根据 url 确定 controller 中处理请求的方法,然后通过反射获取该方法上的注解和参数,解析方法和参数上的注解,最后反射调用方法获取 ModelAndView 结果视图。因为上面采用注解 url 形式说明的,所以我们这里继续以注解处理器适配器来说明.第 5 步调用的就是 AnnotationMethodHandlerAdapter 的 handle().handle()中的核心逻辑由 invokeHandlerMethod(request, response, handler)实现。

```
/** 获取处理请求的方法,执行并返回结果视图 **/
```

```
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response,
Object handler)
            throws Exception {
        // 1.获取方法解析器
        ServletHandlerMethodResolver methodResolver = getMethodResolver(handler);
        // 2.解析request中的url,获取处理request的方法
        Method handlerMethod = methodResolver.resolveHandlerMethod(request);
        // 3.方法调用器
        ServletHandlerMethodInvoker methodInvoker = new ServletHandlerMethodInvoker(methodResolver);
        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        ExtendedModelMap implicitModel = new BindingAwareModelMap();
        // 4.执行方法
        Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler, webRequest,
implicitModel);
        // 5.封装结果视图
        ModelAndView mav =
                methodInvoker.getModelAndView(handlerMethod, handler.getClass(), result, implicitModel,
webRequest);
        methodInvoker.updateModelAttributes(handler, (mav != null ? mav.getModel() : null), implicitModel,
webRequest);
        return mav;

    }
```

这一部分的核心就在 2 和 4 了.先看第 2 步,通过 request 找 controller 的处理方法.实际上就是拼接 controller 的 url 和方法的 url,与 request 的 url 进行匹配,找到匹配的方法.

```
/** 根据url获取处理请求的方法 **/
    public Method resolveHandlerMethod(HttpServletRequest request) throws ServletException {
            // 如果请求url为,localhost:8080/springmvc/helloWorldController/say.action, 则
lookupPath=helloWorldController/say.action
            String lookupPath = urlPathHelper.getLookupPathForRequest(request);
            Comparator<String> pathComparator = pathMatcher.getPatternComparator(lookupPath);
            Map<RequestSpecificMappingInfo, Method> targetHandlerMethods = new
LinkedHashMap<RequestSpecificMappingInfo, Method>();
            Set<String> allowedMethods = new LinkedHashSet<String>(7);
            String resolvedMethodName = null;
            // 遍历controller上的所有方法,获取url匹配的方法
            for (Method handlerMethod : getHandlerMethods()) {
                RequestSpecificMappingInfo mappingInfo = new
RequestSpecificMappingInfo(this.mappings.get(handlerMethod));
                boolean match = false;
                if (mappingInfo.hasPatterns()) {// 获取方法上的url
                    for (String pattern : mappingInfo.getPatterns()) { // 方法上可能有多个url,springmvc支持方
```

```
法映射多个url
                        if (!hasTypeLevelMapping() && !pattern.startsWith("/")) {
                            pattern = "/" + pattern;
                        }
                            // 获取controller上的映射和url和方法上的url,拼凑起来与lookupPath是否匹配
                        String combinedPattern = getCombinedPattern(pattern, lookupPath, request);
                        if (combinedPattern != null) {
                            if (mappingInfo.matches(request)) {
                                match = true;
                                mappingInfo.addMatchedPattern(combinedPattern);
                            }
                            else {
                                if (!mappingInfo.matchesRequestMethod(request)) {
                                    allowedMethods.addAll(mappingInfo.methodNames());
                                }
                                break;
                            }
                        }
                    }
                    mappingInfo.sortMatchedPatterns(pathComparator);
                }
                else if (useTypeLevelMapping(request)) {
                // other

        }
```

通过上面的代码,已经可以找到处理 request 的 controller 中的方法了,现在看如何解析该方法上的参数,并调用该方法。也就是执行方法这一步.执行方法这一步最重要的就是获取方法的参数,然后我们就可以反射调用方法了.

```
public final Object invokeHandlerMethod(Method handlerMethod, Object handler,
            NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {

         Method handlerMethodToInvoke = BridgeMethodResolver.findBridgedMethod(handlerMethod);
        try {
            boolean debug = logger.isDebugEnabled();
              // 处理方法上的其他注解
            for (String attrName : this.methodResolver.getActualSessionAttributeNames()) {
                Object attrValue = this.sessionAttributeStore.retrieveAttribute(webRequest, attrName);
                if (attrValue != null) {
                    implicitModel.addAttribute(attrName, attrValue);
                }
            }
            for (Method attributeMethod : this.methodResolver.getModelAttributeMethods()) {
```

```
                    Method attributeMethodToInvoke =
BridgeMethodResolver.findBridgedMethod(attributeMethod);
                    Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler, webRequest,
implicitModel);
                    if (debug) {
                        logger.debug("Invoking model attribute method: " + attributeMethodToInvoke);
                    }
                    String attrName = AnnotationUtils.findAnnotation(attributeMethod,
ModelAttribute.class).value();
                    if (!"".equals(attrName) && implicitModel.containsAttribute(attrName)) {
                        continue;
                    }
                    ReflectionUtils.makeAccessible(attributeMethodToInvoke);
                    Object attrValue = attributeMethodToInvoke.invoke(handler, args);
                    if ("".equals(attrName)) {
                        Class resolvedType = GenericTypeResolver.resolveReturnType(attributeMethodToInvoke,
handler.getClass());
                        attrName = Conventions.getVariableNameForReturnType(attributeMethodToInvoke,
resolvedType, attrValue);
                    }
                    if (!implicitModel.containsAttribute(attrName)) {
                        implicitModel.addAttribute(attrName, attrValue);
                    }
                }
                  // 核心代码,获取方法上的参数值
                Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler, webRequest,
implicitModel);
                if (debug) {
                    logger.debug("Invoking request handler method: " + handlerMethodToInvoke);
                }
                ReflectionUtils.makeAccessible(handlerMethodToInvoke);
                return handlerMethodToInvoke.invoke(handler, args);

            }
```

resolveHandlerArguments 方法实现代码比较长,它最终要实现的目的就是:完成 request 中的参数和方法参数上数据的绑定.

springmvc 中提供两种 request 参数到方法中参数的绑定方式:

① 通过注解进行绑定,@RequestParam

② 通过参数名称进行绑定.

使用注解进行绑定,我们只要在方法参数前面声明@RequestParam("a"),就可以将 request 中参数 a 的值绑定到方法的该参数上.使用参数名称进行绑定的前提是必须要获取方法中参数的名称,Java 反射只提

供了获取方法的参数的类型,并没有提供获取参数名称的方法.springmvc 解决这个问题的方法是用 asm 框架读取字节码文件,来获取方法的参数名称.asm 框架是一个字节码操作框架,关于 asm 更多介绍可以参考它的官网.个人建议,使用注解来完成参数绑定,这样就可以省去 asm 框架的读取字节码的操作.

```
private Object[] resolveHandlerArguments(Method handlerMethod, Object handler,
        NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {
    // 1.获取方法参数类型的数组
    Class[] paramTypes = handlerMethod.getParameterTypes();
    // 声明数组,存参数的值
    Object[] args = new Object[paramTypes.length];
    //2.遍历参数数组,获取每个参数的值
    for (int i = 0; i < args.length; i++) {
        MethodParameter methodParam = new MethodParameter(handlerMethod, i);
        methodParam.initParameterNameDiscovery(this.parameterNameDiscoverer);
        GenericTypeResolver.resolveParameterType(methodParam, handler.getClass());
        String paramName = null;
        String headerName = null;
        boolean requestBodyFound = false;
        String cookieName = null;
        String pathVarName = null;
        String attrName = null;
        boolean required = false;
        String defaultValue = null;
        boolean validate = false;
        int annotationsFound = 0;
        Annotation[] paramAnns = methodParam.getParameterAnnotations();
        // 处理参数上的注解
        for (Annotation paramAnn : paramAnns) {
            if (RequestParam.class.isInstance(paramAnn)) {
                RequestParam requestParam = (RequestParam) paramAnn;
                paramName = requestParam.value();
                required = requestParam.required();
                defaultValue = parseDefaultValueAttribute(requestParam.defaultValue());
                annotationsFound++;
            }
            else if (RequestHeader.class.isInstance(paramAnn)) {
                RequestHeader requestHeader = (RequestHeader) paramAnn;
                headerName = requestHeader.value();
                required = requestHeader.required();
                defaultValue = parseDefaultValueAttribute(requestHeader.defaultValue());
                annotationsFound++;
            }
            else if (RequestBody.class.isInstance(paramAnn)) {
                requestBodyFound = true;
```

```
                annotationsFound++;
            }
            else if (CookieValue.class.isInstance(paramAnn)) {
                CookieValue cookieValue = (CookieValue) paramAnn;
                cookieName = cookieValue.value();
                required = cookieValue.required();
                defaultValue = parseDefaultValueAttribute(cookieValue.defaultValue());
                annotationsFound++;
            }
            else if (PathVariable.class.isInstance(paramAnn)) {
                PathVariable pathVar = (PathVariable) paramAnn;
                pathVarName = pathVar.value();
                annotationsFound++;
            }
            else if (ModelAttribute.class.isInstance(paramAnn)) {
                ModelAttribute attr = (ModelAttribute) paramAnn;
                attrName = attr.value();
                annotationsFound++;
            }
            else if (Value.class.isInstance(paramAnn)) {
                defaultValue = ((Value) paramAnn).value();
            }
            else if ("Valid".equals(paramAnn.annotationType().getSimpleName())) {
                validate = true;
            }
        }

        if (annotationsFound > 1) {
            throw new IllegalStateException("Handler parameter annotations are exclusive choices - " +
                    "do not specify more than one such annotation on the same parameter: " +
handlerMethod);
        }

        if (annotationsFound == 0) {// 如果没有注解
            Object argValue = resolveCommonArgument(methodParam, webRequest);
            if (argValue != WebArgumentResolver.UNRESOLVED) {
                args[i] = argValue;
            }
            else if (defaultValue != null) {
                args[i] = resolveDefaultValue(defaultValue);
            }
            else {
                Class paramType = methodParam.getParameterType();
                  // 将方法声明中的Map和Model参数,放到request中,用于将数据放到request中带回页面
```

```
                        if (Model.class.isAssignableFrom(paramType) || Map.class.isAssignableFrom(paramType))
{
                            args[i] = implicitModel;
                        }
                        else if (SessionStatus.class.isAssignableFrom(paramType)) {
                            args[i] = this.sessionStatus;
                        }
                        else if (HttpEntity.class.isAssignableFrom(paramType)) {
                            args[i] = resolveHttpEntityRequest(methodParam, webRequest);
                        }
                        else if (Errors.class.isAssignableFrom(paramType)) {
                            throw new IllegalStateException("Errors/BindingResult argument declared " +
                                    "without preceding model attribute. Check your handler method signature!");
                        }
                        else if (BeanUtils.isSimpleProperty(paramType)) {
                            paramName = "";
                        }
                        else {
                            attrName = "";
                        }
                    }
                }
                  // 从request中取值,并进行赋值操作
                if (paramName != null) {
                      // 根据paramName从request中取值,如果没有通过RequestParam注解指定paramName,则使用
asm读取class文件来获取paramName
                    args[i] = resolveRequestParam(paramName, required, defaultValue, methodParam, webRequest,
handler);
                }
                else if (headerName != null) {
                    args[i] = resolveRequestHeader(headerName, required, defaultValue, methodParam,
webRequest, handler);
                }
                else if (requestBodyFound) {
                    args[i] = resolveRequestBody(methodParam, webRequest, handler);
                }
                else if (cookieName != null) {
                    args[i] = resolveCookieValue(cookieName, required, defaultValue, methodParam, webRequest,
handler);
                }
                else if (pathVarName != null) {
                    args[i] = resolvePathVariable(pathVarName, methodParam, webRequest, handler);
                }
                else if (attrName != null) {
```

```
                    WebDataBinder binder =
                            resolveModelAttribute(attrName, methodParam, implicitModel, webRequest, handler);
                    boolean assignBindingResult = (args.length > i + 1 &&
Errors.class.isAssignableFrom(paramTypes[i + 1]));
                    if (binder.getTarget() != null) {
                        doBind(binder, webRequest, validate, !assignBindingResult);
                    }
                    args[i] = binder.getTarget();
                    if (assignBindingResult) {
                        args[i + 1] = binder.getBindingResult();
                        i++;
                    }
                    implicitModel.putAll(binder.getBindingResult().getModel());
                }
            }
             // 返回参数值数组
            return args;

        }
```

关于 asm 框架获取方法参数的部分,这里就不再进行分析了.感兴趣的话自己去就能看到这个过程.

到这里,方法的参数值列表也获取到了,就可以直接进行方法的调用了.整个请求过程中最复杂的一步就是在这里了.ok,到这里整个请求处理过程的关键步骤都分析完了.理解了 springmvc 中的请求处理流程,整个代码还是比较清晰的.

5.8．4 谈谈 Spring MVC 的优化

上面我们已经对 springmvc 的工作原理和源码进行了分析,在这个过程发现了几个优化点:

1.controller 如果能保持单例,尽量使用单例,这样可以减少创建对象和回收对象的开销.也就是说,如果 controller 的类变量和实例变量可以以方法形参声明的尽量以方法的形参声明,不要以类变量和实例变量声明,这样可以避免线程安全问题.

2.处理 request 的方法中的形参务必加上@RequestParam 注解,这样可以避免 springmvc 使用 asm 框架读取 class 文件获取方法参数名的过程.即便 springmvc 对读取出的方法参数名进行了缓存,如果不要读取 class 文件当然是更加好.

3.阅读源码的过程中,发现 springmvc 并没有对处理 url 的方法进行缓存,也就是说每次都要根据请求 url 去匹配 controller 中的方法 url,如果把 url 和 method 的关系缓存起来,会不会带来性能上的提升呢?有点恶心的是,负责解析 url 和 method 对应关系的 ServletHandlerMethodResolver 是一个 private 的内部类,不能直接继承该类增强代码,必须要该代码后重新编译.当然,如果缓存起来,必须要考虑缓存的线程安全问题.