SSM 框架

SSM 框架

一、 Spring IoC/AOP 底层原理

1 IoC

引用 *Spring* 官方原文：*This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [1] principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.*

“控制反转（*IoC*）”也称为“依赖注入（*DI*）”，是一个定义对象依赖的过程，对象只和构造参数，工厂方法参数，对象实例属性或工厂方法返回相关。容器在创建这些 *bean* 的时候注入这些依赖。这个过程是一个反向的过程，所以命名为依赖反转，对象实例的创建由其提供的构造方法或服务定位机制来实现。

IoC 最大的好处就是“***解耦***”。

1.1 容器初始化流程

new ClasspathXmlApplicationContext();

ContextLoaderListener / DispatcherServlet -> WebApplicationContext

ApplicationContext 容器的初始化流程主要由 *AbstractApplicationContext* 类中的 ***refresh*** 方法实现。大致过程为：为 *BeanFactory* 对象执行后续处理（如：*context:propertyPlaceholder* 等）->在上下文（*Context*）中注册 *bean*->为 *bean* 注册拦截处理器（*AOP* 相关）->初始化上下文消息（初始化 *id* 为 *messgeSource* 的国际化 *bean* 对象）->初始化事件多播（处理事件监听，如 *ApplicationEvent* 等）->初始化主题资源（*SpringUI* 主题 *ThemeSource*）->注册自定义监听器->实例化所有非 *lazy-init* 的 *singleton* 实例->发布相应事件（*Lifecycle* 接口相关实现类的生命周期事件发布）

在 *spring* 中，构建容器的过程都是同步的。同步操作是为了保证容器构建的过程中，不出现多线程资源冲突问题。

```
516     public void refresh() throws BeansException, IllegalStateException {
517         synchronized (this.startupShutdownMonitor) {
518             // Prepare this context for refreshing.
```

BeanFactory 的构建。 *BeanFactory* 是 *ApplicationContext* 的父接口。是 *spring* 框架中的顶级容器工厂对象。*BeanFactory* 只能管理 *bean* 对象。没有其他功能。如：*aop* 切面管理，*propertyplaceholder* 的加载等。 ***构建 BeanFactory 的功能就是管理 bean 对象***。

创建 *BeanFactory* 中管理的 *bean* 对象。

postProcessBeanFactory - 加载配置中 *BeanFactory* 无法处理的内容。如：

propertyplacehodler 的加载。

invokeBeanFactoryPostProcessors - 将上一步加载的内容，作为一个容器可以管理的 *bean* 对象注册到 *ApplicationContext* 中。***底层实质是在将 postProcessBeanFactory 中加载的内容包装成一个容器 ApplicationContext 可以管理的 bean 对象。***

registerBeanPostProcessors - 继续完成上一步的注册操作。配置文件中配置的 *bean* 对象都创建并注册完成。

initMessageSource - *i18n*,国际化。初始化国际化消息源。

initApplicationEventMulticaster - 注册事件多播监听。如 *ApplicationEvent* 事件。是 *spring* 框架中的观察者模式实现机制。

onRefresh - 初始化主题资源（*ThemeSource*）。*spring* 框架提供的视图主题信息。

registerListeners - 创建监听器，并注册。

finishBeanFactoryInitialization - 初始化配置中出现的所有的 *lazy-init=false* 的 *bean* 对象。且 *bean* 对象必须是 *singleton* 的。

finishRefresh - 最后一步。 发布最终事件。生命周期监听事件。 *spring* 容器定义了生命周期接口。可以实现容器启动调用初始化，容器销毁之前调用回收资源。*Lifecycle* 接口。

1.2 多容器/父子容器概念

Spring 框架允许在一个应用中创建多个上下文容器。但是建议容器之间有父子关系。可以通过 *ConfigurableApplicationContext* 接口中定义的 *setParent* 方法设置父容器。一旦设置父子关系，则可以通过子容器获取父容器中除 *PropertyPlaceHolder* 以外的所有资源，父容器不能获取子容器中的任意资源（类似 *Java* 中的类型继承）。

典型的父子容器： *spring* 和 *springmvc* 同时使用的时候。*ContextLoaderListener* 创建的容器是父容器，*DispatcherServlet* 创建的容器是子容器。

保证一个 *JVM* 中，只有一个树状结构的容器树。可以通过子容器访问父容器资源。

1.3 p 域/c 域

Spring2.0 之后引入了 *p(property* 标签*)*域、*Spring3.1* 之后引入了 *c(constractor-arg* 标签*)*域。可以简化配置文件中对 *property* 和 *constructor-arg* 的配置。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="oneBean" class="com.sxt.OneBean"
        p:a="10"
        p:o-ref="otherBean"
        c:a="20"
        c:o-ref="otherBean"/>

    <bean id="otherBean" class="com.sxt.OtherBean" />
```

```
</beans>
class OneBean{
    int a;
    Object o;
    public OneBean(int a, Object o){ this.a = a; this.o = o;}
    // getters and setters for fields.
}
```

1.4 lookup-method

lookup-method 一旦应用，*Spring* 框架会自动使用 *CGLIB* 技术为指定类型创建一个动态子类型，并自动实现抽象方法。可以动态的实现依赖注入的数据准备。

在效率上，比直接自定义子类型慢。相对来说更加通用。可以只提供 *lookup-method* 方法的返回值对象即可实现动态的对象返回。

在工厂方法难以定制的时候使用。

也是模板的一种应用。工厂方法的扩展。

如：工厂方法返回对象类型为接口类型。且不同版本应用返回的对象未必相同时使用。可以避免多次开发工厂类。

```
package com.sxt.lookupmethod;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestLookupMethod {
    public static void main(String[] args) {
        ApplicationContext context =
                new
ClassPathXmlApplicationContext("classpath:lookupmethod/applicationContext.xml");

        CommandManager manager = context.getBean("manager",
CommandManager.class);

        System.out.println(manager.getClass().getName());
        manager.process();
    }
}

abstract class CommandManager{
    public void process() {
        MyCommand command = createCommand();
        // do something ...
        System.out.println(command);
```

```
        }

        protected abstract MyCommand createCommand();
    }

    class MyCommand{
        public MyCommand(){
            System.out.println("MyCommand instanced");
        }
    }
```

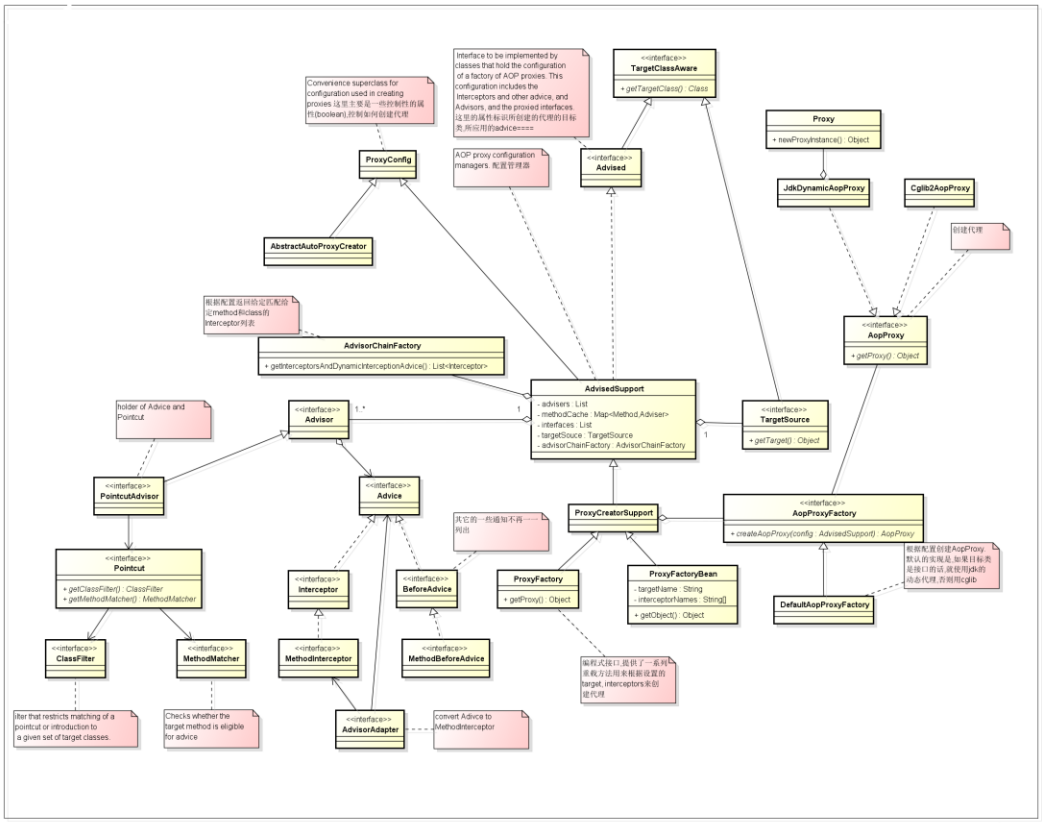
```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="manager" class="com.sxt.lookupmethod.CommandManager">
        <lookup-method bean="command" name="createCommand"/>
    </bean>

    <bean id="command" class="com.sxt.lookupmethod.MyCommand"></bean>

</beans>
```

2 AOP



面向切面编程，其底层原理就是动态代理实现。如果切面策略目标有接口实现，使用 *JDK* 的动态代理技术；无接口实现则使用 *CGLIB* 技术生成动态代理。

在商业环境中，接口使用度是非常高的，在这主要分析 *Spring* 如何使用 *JDK* 的动态代理技术生成动态代理对象。主要代码在 *JdkDynamicAopProxy*、*AdvisedSupport*、*DefaultAdvisorChainFactory*、*ReflectiveMethodInvocation* 类中。

2.1 JdkDynamicAopProxy.invoke();

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    MethodInvocation invocation;
    Object oldProxy = null;
    boolean setProxyContext = false;
    // 获取代理对象中的目标源对象。 相当于 Service 实现类。
    TargetSource targetSource = this.advised.targetSource;
    Object target = null;

    try {
        // 判断逻辑目的是避免代理对象执行出现 RuntimeException
        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
            // The target does not implement the equals(Object) method itself.
```

```
                return equals(args[0]);
            }
            else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
                // The target does not implement the hashCode() method itself.
                return hashCode();
            }
            else if (method.getDeclaringClass() == DecoratingProxy.class) {
                // There is only getDecoratedClass() declared -> dispatch to proxy config.
                return AopProxyUtils.ultimateTargetClass(this.advised);
            }
            else 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; // return value, 定义返回结果数据的引用。

            if (this.advised.exposeProxy) {
                // Make invocation available if necessary.
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }

            // Get as late as possible to minimize the time we "own" the target,
            // in case it comes from a pool. 目标对象获取。目标对象的类对象
            target = targetSource.getTarget();
            Class<?> targetClass = (target != null ? target.getClass() : null);

            // Get the interception chain for this method. 获取代理需要在目标方法执
行前后，切入的拦截器链。 关注方法
            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.
            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.
```

```
                    // 如果代理对象没有需要切入的拦截器，执行目标对象中的方法。
                    Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                    retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
                }
                else {
                    // We need to create a method invocation...
                    // 创建一个执行器，加入拦截信息，并按照顺序执行拦截代码和目标对象中的方法。
                    invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
                    // Proceed to the joinpoint through the interceptor chain.
                    // 方法执行。按照顺序执行拦截代码和目标对象中的方法。关注方法
                    retVal = invocation.proceed();
                }

                // Massage return value if necessary. 获取目标对象中方法的返回结果类型。
                Class<?> returnType = method.getReturnType();
                if (retVal != null && retVal == target &&
                        returnType != Object.class && 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);
                }
            }
        }
```

2.2 AdvisedSupport.

getInterceptorsAndDynamicInterceptionAdvice

```
        public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method,
@Nullable Class<?> targetClass) {
            // 方法匹配信息， 获取 spring 容器中的缓存。
            MethodCacheKey cacheKey = new MethodCacheKey(method);
            // 从已知的缓存中获取方法缓存匹配信息。
            List<Object> cached = this.methodCache.get(cacheKey);
            if (cached == null) {
                // 查询代理对象需要执行的拦截信息。关注方法。
                cached                                                    =
this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
                        this, method, targetClass);
                // 保存缓存数据，为后续其他代码提供缓存内容。
                this.methodCache.put(cacheKey, cached);
            }
            return cached;
        }
```

2.3 DefaultAdvisorChainFactory.

getInterceptorsAndDynamicInterceptionAdvice

```
        public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
                Advised config, Method method, @Nullable Class<?> targetClass) {

            // This is somewhat tricky... We have to process sintroduction first,
            // but we need to preserve order in the ultimate list.
            List<Object> interceptorList = new ArrayList<>(config.getAdvisors().length);
            Class<?>    actualClass    =    (targetClass    !=    null    ?    targetClass    :
method.getDeclaringClass());
            boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
            // 通知注册器。spring 容器会将配置好的所有通知使用注册器管理。
            AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
```

```
            // 从配置信息中获取通知对象。
            for (Advisor advisor : config.getAdvisors()) {
                if (advisor instanceof PointcutAdvisor) {
                    // Add it conditionally.
                    PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
                    if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {
                        MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
                        MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
                        if (MethodMatchers.matches(mm, method, actualClass, 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(actualClass)) {
                        Interceptor[] interceptors = registry.getInterceptors(advisor);
                        interceptorList.addAll(Arrays.asList(interceptors));
                    }
                }
                else {
                    Interceptor[] interceptors = registry.getInterceptors(advisor);
                    interceptorList.addAll(Arrays.asList(interceptors));
                }
            }

            return interceptorList;
        }
```

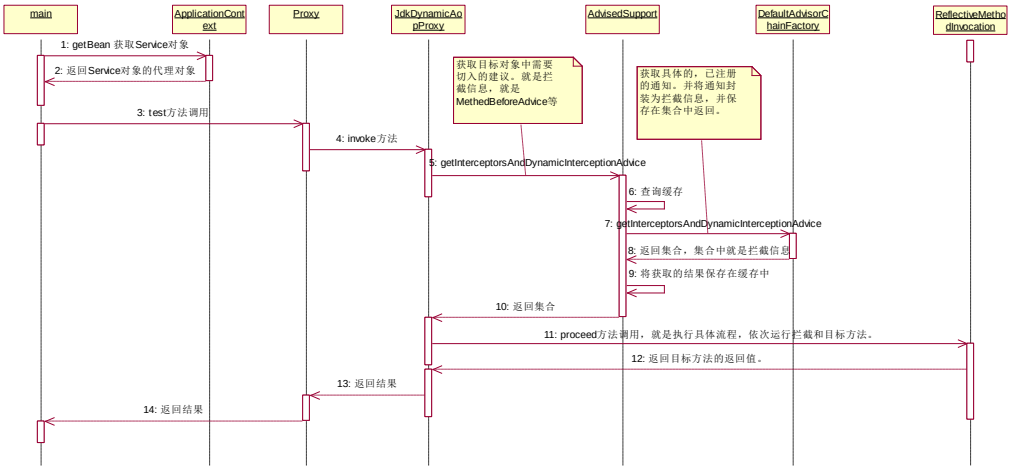
2.4 ReflectiveMethodInvocation. proceed

```
public Object proceed() throws Throwable {
    // 开始执行代理方法。包含通知方法和目标对象中的真实方法。
    // 判断当前代理是否还有需要执行通知。如果没有通知，执行目标代码。
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
        return invokeJoinpoint();
    }

    Object interceptorOrInterceptionAdvice =

        this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
    if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
        // Evaluate dynamic method matcher here: static part will already have
        // been evaluated and found to match.
        InterceptorAndDynamicMethodMatcher dm =
                (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
        if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
            return dm.interceptor.invoke(this);
        }
        else {
            // Dynamic matching failed.
            // Skip this interceptor and invoke the next in the chain.
            return proceed();
        }
    }
    else {
        // It's an interceptor, so we just invoke it: The pointcut will have
        // been evaluated statically before this object was constructed.
        return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
    }
}
```

2.5 AOP 源码流程图



3 AOP 中常用的 Pointcut-expression

AOP 开发中，有非常重要的几个概念，其中有一个概念叫“切点”。代表通知切入到代码执行流程的那个位置点。

切点一般通过表达式定义。*spring* 框架会通过 *SpringEL* 来解析表达式。表达式有多种定义方式。分别对应不同的解析结果。

3.1 execution 表达式

语法格式： *execution(*返回类型 包名.类名.方法名*(*参数表*))*

如：

execution(java.lang.String com.xxx.service.AService.test(java.lang.Integer))

在类型 *com.xxx.service.AService* 中有方法 *test*，且参数为 *Integer*，返回类型为 *String* 时增加切面。

execution(com.xxx.AService.*(..))*

com.xxx.AService 类型中的任意方法，任意类型返回结果，参数表不限定，都增加切面。

应用：***最常用***。也是相对最通用。根据方法执行的标准，定义切点。如：事务处理，日志处理。

3.2 target 表达式

以目标对象作为切点的表达式定义方式。

语法： *target(*包名.接口名*)*

如： *target(com.xxx.IA)* - 所有实现了 *IA* 接口的实现类，作为代理的目标对象，会自动增加通知的织入，实现切面。

应用：为某一个具体的接口实现提供的配置。如：登录。登录的时候需要执行的附属逻辑是比较多的。在不同的业务流程中，附属逻辑也不同。如：电商中，可能在登录的时候，需要去执行购物车合并。

3.3 this 表达式

实现了某接口的代理对象，会作为切点。 和 *target* 很类似。

语法： *this(*包名.接口名*)*

如： *this(com.xxx.IA)* - 代理对象 *Proxy* 如果实现了 *IA* 接口，则作为连接点。

应用：针对某个具体的代理提供的配置。比 *target* 切点粒度细致。因为目标对象可以多实现。代理对象可以针对目标对象实现的多个接口的某一个接口，提供特定的切点。如：银行中的登录，银行中的帐户种类非常多。且有交叉。如：借记卡，贷记卡，借记还贷卡，贷记还贷卡等。可以针对还贷接口提供一个切点，做还贷信息的记录等。

3.4 within 表达式

以包作为目标，定义切点。

语法： *within(*包名.*)* - 代表在包中的任意接口或类型都作为切点。

应用： 针对某一个包提供的切点，粒度比 *target* 粗糙。如：某包中的所有接口都需要执行某附属逻辑。如：电商平台中的下订单。下订单服务中可能需要特定的逻辑（时间戳校验，库存检查等），这些逻辑，是其他业务线中不需要提供切面的。

3.5 args 表达式

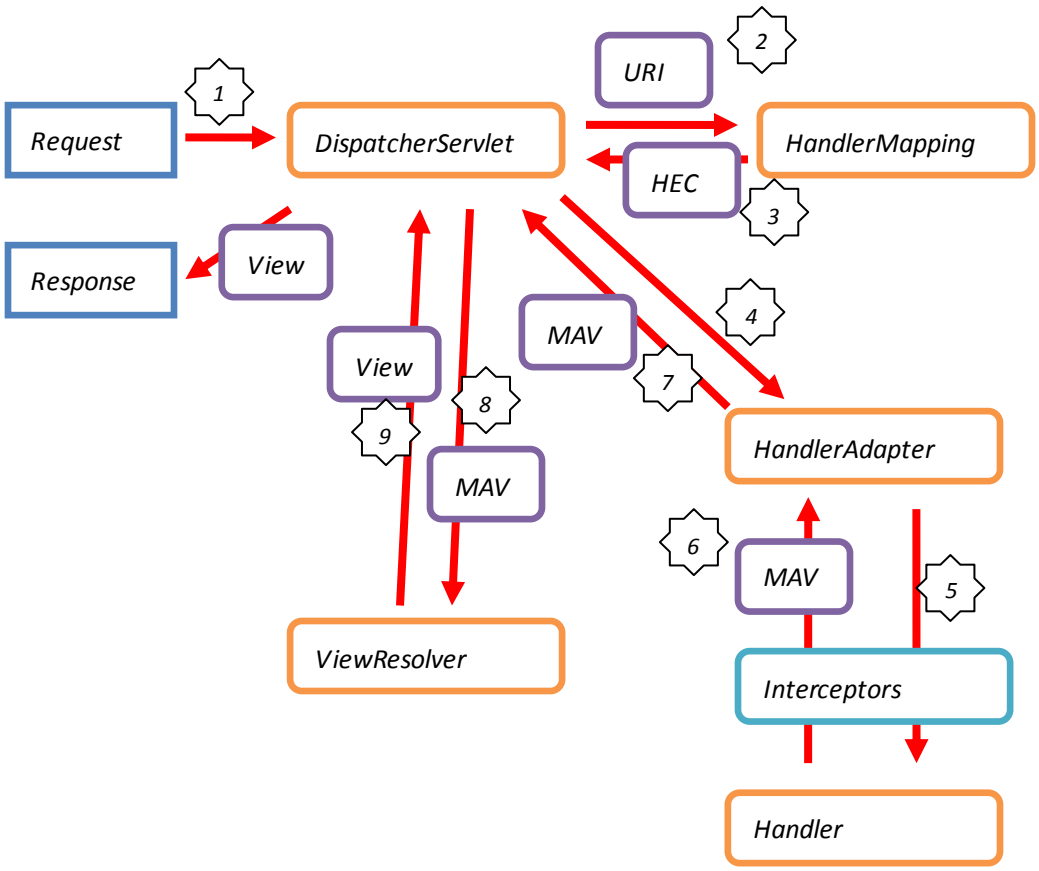
以参数标准作为目标，定义切点。

语法： *args(*类型,类型*)* - 代表方法的参数表符合要求的时候，作为切点。参数表是有顺序的。

应用：主要应用在参数校验中。如：登录的时候必须传递两个字符串参数（登录名和密码）。可以使用 *args* 来限定。 配合这 *execution* 实现。 如： *execution(* xxxx.*.login(..)) args(string,string)*。 ***是使用频率最低的表达式***。

二、 SpringMVC 组件实现原理

1 执行逻辑图



2 组件介绍

2.1 DispatcherServlet

DispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，*DispatcherServlet* 的存在降低了组件之间的耦合性。

MVC 模式: 传统定义,一个 *WEB* 应用中,只有唯一的一个控制器和客户端交互. 所有的客户端请求和服务器单点接触. 这个控制器称为核心控制器*(*前端控制器*)*。 传统定义中，核心控制器的实现使用 *Servlet* 实现。如：*SpringMVC*，*Struts1*。

MVC 优势： 单点接触，可以有效的解耦。可以实现功能的重用。

M - model

V - view

C - controller

2.2 HandlerMapping

处理映射器。

HandlerMapping 负责根据用户请求找到 *Handler* 即处理器（如：用户自定义的 *Controller*），*springmvc* 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

映射器相当于配置信息或注解描述。 映射器内部封装了一个类似 *map* 的数据结构。使用 *URL* 作为 *key*，*HandlerExecutionChain* 作为 *value*。核心控制器，可以通过请求对象（请求对象中包含请求的 *URL*）在 *handlerMapping* 中查询 *HandlerExecutionChain* 对象。

是 *SpringMVC* 核心组件之一。是必不可少的组件。无论是否配置，*SpringMVC* 会有默认提供。

如果有 *<mvc:annotation-driven/>* 标签配置，默认的映射器：*RequestMappingHandlerMapping*

如果没有*<mvc:annotation-driven/>*标签配置，且使用注解开发 *SpringMVC* 代码，默认的映射器是：*RequestMappingHandlerMapping*。（老版本中有其他的映射器，但是已经过时。）

2.3 HandlerAdapter

通过 *HandlerAdapter* 对处理器（*Handler*）进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

典型的适配器：*SimpleControllerHandlerAdapter*。最基础的。处理自定义控制器（*Handler*）和 *SpringMVC* 控制器顶级接口 *Controller* 之间关联的。

如果定义了 *<mvc:annotation-driven/>* 标签配置，使用适配器对象为：*HttpRequestHandlerAdapter*。

适配器也是 *SpringMVC* 中的核心组件之一。必须存在。*SpringMVC* 框架有默认值。

2.4 Handler

处理器。

Handler 是继 *DispatcherServlet* 前端控制器的后端控制器（自定义控制器），在 *DispatcherServlet* 的控制下 *Handler* 对具体的用户请求进行处理。由于 *Handler* 涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发 *Handler*。

在 *SpringMVC* 中对 *Handler* 没有强制的类型要求。在 *SpringMVC* 框架中，对 *Handler* 的引用定义类型为 *Object*。

处理器理论上说不是必要的核心组件。

SpringMVC 框架是一个线程不安全的，轻量级的框架。一个 handler 对象，处理所有的请求。开发过程中，注意线程安全问题。

2.5 ViewResolver

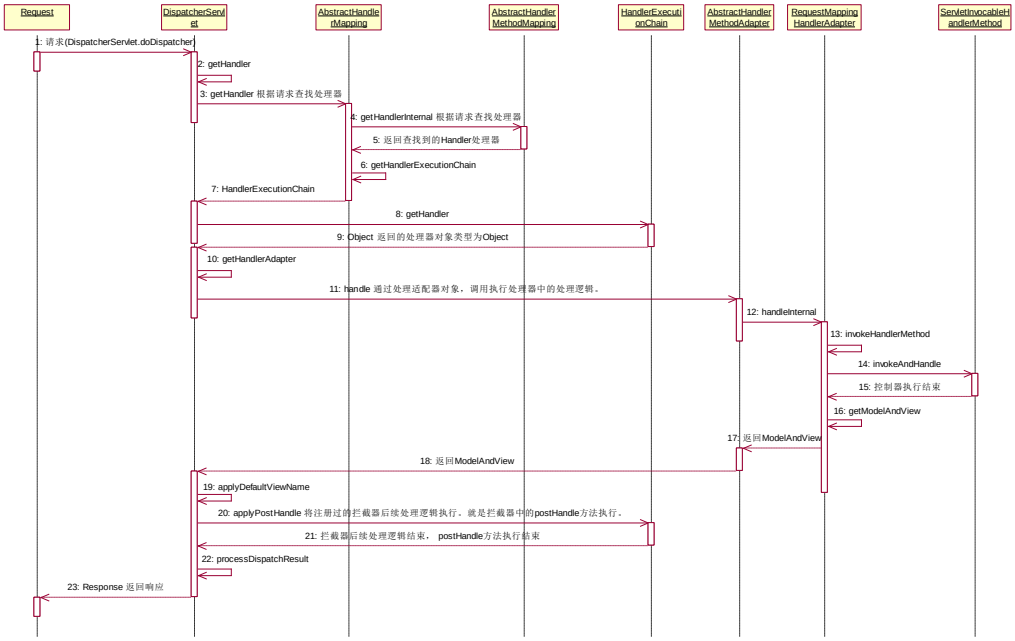
ViewResolver 负责将处理结果生成 *View* 视图，*ViewResolver* 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 *View* 视图对象，最后对 *View* 进行渲染将处理结果通过页面展示给用户。

是 *SpringMVC* 中必要的组件之一。*SpringMVC* 提供默认视图解析器。

InternalResourceViewResolver。内部资源视图解析器。

视图解析器是用于处理动态视图逻辑的。静态视图逻辑，不通过 *SpringMVC* 流程。直接通过 *WEB* 中间件（*Tomcat*）就可以访问静态资源。

3 源码解读



三、 MyBatis 自动化生成&关联查询

1 mybatis-generator-gui

代码自动生成插件。缺陷:没有多表操作.自动生成是针对互联网开发提出的.互联网开发中多表操作很少.

类似 *MyEclipse* 中的一个逆向工程组件.根据数据库表格设计,自动生成实体类型,*Mapper/DAO* 接口，有必要生成对应的实现类，相关的 *Mapper/DAO* 的配置文件。

MyBatis-generator ： 是一个开发完善的 *JavaSE* 工程。主要实现方式，就是使用 *JDBC/MyBatis* 连接数据库，查询表格信息，自动化生成对应的 *java* 代码。

版本：

gui 版本：提供可视化界面，操作简单。弊端是一次只能生成一张表格的对应代码。

console 版本：无可视化界面，操作相对麻烦。优势是可以一次性生成若干表格的对应代码。企业使用。

使用方式：

gui 版本：运行 *com.zzg.mybatis.generator.MainUI*。

2 mybatis-generator-console

插件工具。 是用于自动生成代码的。 可以生成的代码包括：实体类型，*Mapper* 接口，*Mapper* 接口对应的 *SQL* 映射文件。

2.1 代码生成方式

关注配置文件：*generatorConfig.xml*
配置文件中配置了需要逆向生成代码的表格有哪些。还配置了数据库的链接相关信息。

2.2 生成后的 Mapper 使用方式

详见代码。

2.3 自动生成代码的优缺点

2.3.1 优点

方便。*SQL* 规范。快捷。维护成本低

2.3.2 缺点

只能单表操作。（互联网应用中，多表联合查询相对较少。）
如果表格中有 *Text*，*BLOB* 字段。查询、更新的时候，需要特殊注意。默认的查询方法不查询 *Text* 和 *BLOB* 字段，默认的更新方法不更新 *Text* 和 *BLOB* 字段。必须调用 *selectByExampleWithBLOBs*，*updateByExamaleWithBLOBs*，*updateByPrimaryKeyWithBLOBs*

3 Interceptor

是 *MyBatis* 提供的一个插件（*plugin* 扩展）。代表拦截器。可以拦截代码中的数据库访问操作。就是 *Statement* 操作。
拦截后，可以去修改正在执行的 *SQL* 语句，可以额外访问数据库，可以实现若干数据计算和处理。
使用场景不多。针对某类型的 *SQL* 实现拦截的工具。粒度太粗糙。
市场常用的 *Interceptor* 插件只有 *PageHelper*。
Interceptor 影响执行效率。

4 关联查询

在 *MyBatis* 中，关联查询分为一次查询和 *N+1* 次查询。
一次查询：是使用多表联合查询 *SQL* 语法实现。
一次查询 *SQL* 语法相对复杂，效率比较高。如果查询的数据量大，不推荐使用。
N+1 次查询：是使用多个单表查询 *SQL* 语法实现。

N+1 次查询效率低，多次访问数据库，网络操作为多次。可以使用 *lazy* 实现懒加载。在 *MyBatis* 中懒加载并不是非常好用。

在一对一关系查询的时候，可以使用 *AutoMapping* 的方式实现查询。具体语法为：
select column as `关联属性名.关联对象内部属性名` *from ....*
AutoMapping 不推荐使用，语义不明确，维护成本高。

4.1 一对一

```
<resultMap type="具体类型" id="唯一命名">
    <id column="字段名" property="属性名"/>
    <result column="字段名" property="属性名"/>
    <!-- 一次访问数据库 -->
    <association property="属性名" javaType="属性的类型">
        <id column="字段名" property="属性名"/>
        <result column="字段名" property="属性名"/>
    </association>
    <!-- n+1 次访问数据库 -->
    <association property="属性名" javaType="属性的类型"
        select="命名空间.标签 ID 就是要调用的 SQL 语法"
        column="传递的参数, {参数名=当前行的字段名}"/>
</resultMap>
```

4.2 一对多&多对多

```
<resultMap type="具体类型" id="唯一命名">
    <id column="字段名" property="属性名"/>
    <result column="字段名" property="属性名"/>
    <!-- 一次访问数据库 -->
    <collection property="属性名" javaType="属性的类型" ofType="集合中的泛型">
        <id column="字段名" property="属性名"/>
        <result column="字段名" property="属性名"/>
    </ collection >
    <!-- n+1 次访问数据库 -->
    < collection property="属性名" javaType="属性的类型"
        ofType="集合中的泛型"
        select="命名空间.标签 ID 就是要调用的 SQL 语法"
        column="传递的参数, {参数名=当前行的字段名}"/>
</resultMap>
```

4.3 深层嵌套 不推荐应用

```
<resultMap type="具体类型" id="唯一命名">
    <id column="字段名" property="属性名"/>
    <result column="字段名" property="属性名"/>
    <!-- 一次访问数据库 -->
    <collection property="属性名" javaType="属性的类型" ofType="集合中的泛型">
        <id column="字段名" property="属性名"/>
        <result column="字段名" property="属性名"/>
        <association property="" type="">
            <id column="" property=""/>
            <collection />
        </association>
    </ collection >
    <!-- n+1 次访问数据库 -->
    < collection property="属性名" javaType="属性的类型"
        ofType="集合中的泛型"
        select="命名空间.标签 ID 就是要调用的 SQL 语法"
        column="传递的参数, {参数名=当前行的字段名}"/>
</resultMap>
```

5 Provider

@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider

MyBatis 中提供的一个注解开发模式下的动态 *SQL* 构建方式。相对来说，*SQL* 语法结构只满足标准 *SQL*。包含： *insert/update/select(select.from.where.group by.having.order by.)/delete*

Provider 编写的 *SQL* 语句。难以实现 *SQL* 优化。通过代码逻辑定义 *SQL* 语句。优化的代价就是重新编译代码。

不建议使用。如果需要动态 *SQL*，推荐使用配置文件中的对应标签。

如： *<if> <choose> <foreach>*等。

6 lazy

在 *mybatis* 核心配置文件中增加下述配置.

```
<!-- 配置环境参数 -->
<settings>
    <!-- 开启延迟加载 -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 关闭侵入性延迟加载
        侵入性延迟加载代表,如果访问了主数据对象，关联数据自动加载。
    -->
```

```
        <setting name="aggressiveLazyLoading" value="false"/>
    </settings>
```

四、 Maven+SSM

使用 *Maven* 做 *SSM* 框架整合。

看 *demo*。

看代码、配置文件。