在 Spring 的 Aop 中，适配器模式应用的非常广泛。Spring 使用 Advice（通知）来增强被代理类的功能，

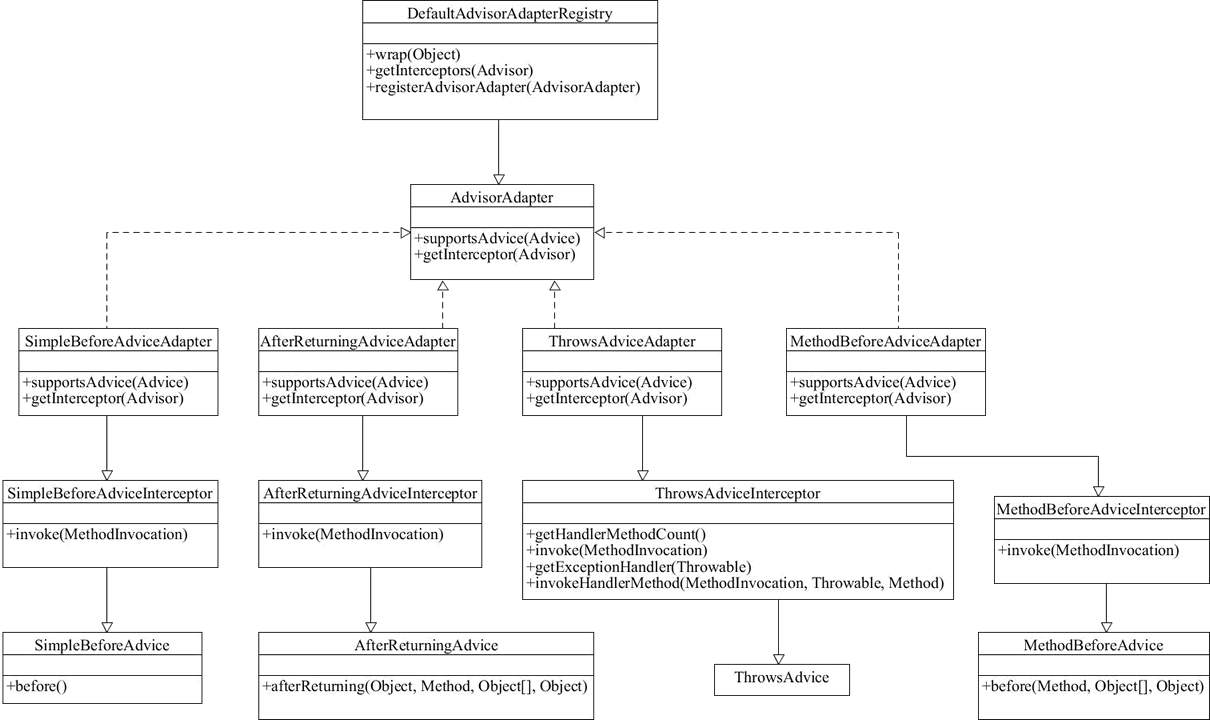
Advice 的类型（ 这些就是要被适配转换的对象)主要有 BeforeAdvice、AfterReturningAdvice、ThrowsAdvice。

每种 Advice 都有对应的拦截器，即 MethodBeforeAdviceInterceptor、AfterReturningAdviceInterceptor、ThrowsAdviceInterceptor。

Spring需要将每个 Advice 都封装成对应的拦截器类型，返回给容器，所以需要使用适配器模式对 Advice 进行转换。

AdvisorAdapter就是目标接口Target, 有两个方法，一个判断 Advice 类型是否匹配supportAdvice()，一个是工厂方法getInterceptor()，创建对应类型的 Advice 对应的拦截器.

AdvisorAdapter有几个实现类: SimpleBeforeAdviceAdapter, AfterReturningAdviceAdapter, ThrowsAdviceAdapter,MethodBeforeAdapter.  
  
各种不同类型的 Interceptor，通过适配器统一对外提供接口，如下类图所示：client ---> target ---> adapter ---> interceptor ---> advice。最终调用不同的 advice来实现被代理类的增强。



先来看顶层接口 AdvisorAdapter,它是目标接口Target。

1. **public** **interface** AdvisorAdapter {
2. boolean supportsAdvice(Advice advice);
3. MethodInterceptor getInterceptor(Advisor advisor);
4. }

再来看 MethodBeforeAdviceAdapter 适配器类。

1. **class** MethodBeforeAdviceAdapter **implements** AdvisorAdapter, Serializable {
2. @Override
3. **public** boolean supportsAdvice(Advice advice) {
4. **return** (advice **instanceof** MethodBeforeAdvice);
5. }
6. @Override //在这里产生不同的拦截器.
7. **public** MethodInterceptor getInterceptor(Advisor advisor) {
8. MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
9. **return** **new** MethodBeforeAdviceInterceptor(advice);
10. }
11. }

Spring 会根据不同的 AOP 配置来确定使用对应的 Advice，与策略模式不同的是，一个方法可以同时拥有多个 Advice。

再看它的客户端类:

public class DefaultAdvisorAdapterRegistry implements AdvisorAdapterRegistry, Serializable {

private final List<AdvisorAdapter> adapters = new ArrayList(3);

public DefaultAdvisorAdapterRegistry() {

// 构造时就在这里注册了适配器

this.registerAdvisorAdapter(new MethodBeforeAdviceAdapter());

this.registerAdvisorAdapter(new AfterReturningAdviceAdapter());

this.registerAdvisorAdapter(new ThrowsAdviceAdapter());

}

public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException {

List<MethodInterceptor> interceptors = new ArrayList(3);

Advice advice = advisor.getAdvice();

if (advice instanceof MethodInterceptor) {

interceptors.add((MethodInterceptor)advice);

}

Iterator var4 = this.adapters.iterator();

//逐个取出注册的适配器，调用 supportsAdvice() 方法来判断 Advice 对应的类型，然后调用 getInterceptor() 创建对应类型的拦截器

while(var4.hasNext()) {

AdvisorAdapter adapter = (AdvisorAdapter)var4.next();

if (adapter.supportsAdvice(advice)) { // 判断是否支持此

interceptors.add(adapter.getInterceptor(advisor)); // 这里调用适配器方法

}

}

if (interceptors.isEmpty()) {

throw new UnknownAdviceTypeException(advisor.getAdvice());

} else {

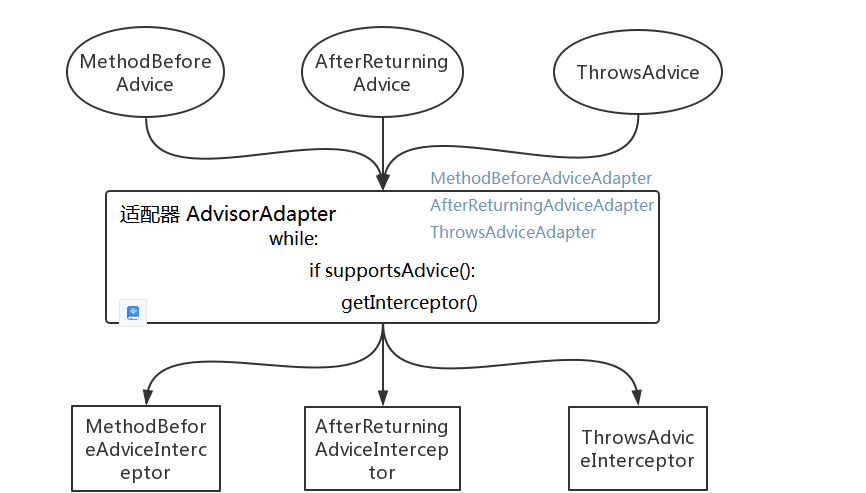
return (MethodInterceptor[])interceptors.toArray(new MethodInterceptor[0]);

}

}

// ...省略...

}



## SpringMVC中的适配器模式

Spring MVC中的适配器模式主要用于执行目标 Controller 中的请求处理方法。

在Spring MVC中，DispatcherServlet 作为用户，HandlerAdapter 作为期望接口，具体的适配器实现类用于对目标类进行适配，Controller 作为需要适配的类。

为什么要在 Spring MVC 中使用适配器模式？Spring MVC 中的 Controller 种类众多，不同类型的 Controller 通过不同的方法来对请求进行处理。如果不利用适配器模式的话，DispatcherServlet 直接获取对应类型的 Controller，需要的自行来判断，像下面这段代码一样：

if( mappedHandler.getHandler() instanceof MultiActionController){

((MultiActionController)mappedHandler.getHandler()).xxx

}else if(mappedHandler.getHandler() instanceof XXX){

...

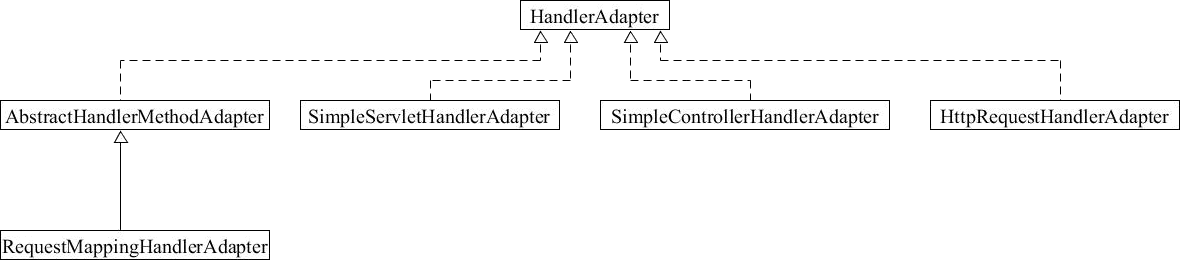
}else if(...){

...

}

这样假设如果我们增加一个 HardController,就要在代码中加入一行 if(mappedHandler.getHandler() instanceof HardController)，这种形式就使得程序难以维护，也违反了[设计模式](https://so.csdn.net/so/search?q=%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_37774171/article/details/_blank)中的开闭原则 – 对扩展开放，对修改关闭。

适配器模式在 SpringMVC 中的经典使用体现在它的核心方法 doDispatch 方法中，再来看一个 Spring MVC 中的 HandlerAdapter 类，它也有多个子类，类图如下。



我们来看看源码，首先是适配器接口 HandlerAdapter:

public interface HandlerAdapter {

boolean supports(Object var1);

ModelAndView handle(HttpServletRequest var1, HttpServletResponse var2, Object var3) throws Exception;

long getLastModified(HttpServletRequest var1, Object var2);

}

现该接口的适配器每一个 Controller 都有一个适配器与之对应，这样的话，每自定义一个 Controller 需要定义一个实现 HandlerAdapter 的适配器。

以HttpRequestHandlerAdapter 这个适配器代码如下：

public class HttpRequestHandlerAdapter implements HandlerAdapter {

public HttpRequestHandlerAdapter() {

}

public boolean supports(Object handler) {

return handler instanceof HttpRequestHandler;

}

public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

((HttpRequestHandler)handler).handleRequest(request, response);

return null;

}

public long getLastModified(HttpServletRequest request, Object handler) {

return handler instanceof LastModified ? ((LastModified)handler).getLastModified(request) : -1L;

}

}

当Spring容器启动后，会将所有定义好的适配器对象存放在一个List集合中，当一个请求来临时，DispatcherServlet 会通过 handler 的类型找到对应适配器，并将该适配器对象返回给用户，然后就可以统一通过适配器的 hanle() 方法来调用 Controller 中的用于处理请求的方法。

public class DispatcherServlet extends FrameworkServlet {

private List<HandlerAdapter> handlerAdapters;

//初始化handlerAdapters

private void initHandlerAdapters(ApplicationContext context) {

//..省略...

}

// 遍历所有的 HandlerAdapters，通过 supports 判断找到匹配的适配器

protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {

for (HandlerAdapter ha : this.handlerAdapters) {

if (logger.isTraceEnabled()) {

logger.trace("Testing handler adapter [" + ha + "]");

}

if (ha.supports(handler)) {

return ha;

}

}

}

// 分发请求，请求需要找到匹配的适配器来处理

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = null;

// Determine handler for the current request.

mappedHandler = getHandler(processedRequest);

// 确定当前请求的匹配的适配器.

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

ha.getLastModified(request, mappedHandler.getHandler());

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

}

// ...省略...

}

通过适配器模式我们将所有的 controller 统一交给 HandlerAdapter 处理，免去了写大量的 if-else 语句对 Controller 进行判断，也更利于扩展新的 Controller 类型。