Java实践:用动态代理进行修饰Java认证考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/644/2021_2022_Java_E5_AE_9E_E8_B7_B5_c104_644679.htm 动态代理工具 是 java.lang.reflect 包的一部分，在 JDK 1.3 版本中添加到 JDK，它允许程序创建代理对象，代理对象能实现一个或多个已知接口，并用反射代替内置的虚方法分派，编程地分派对接口方法的调用。这个过程允许实现“ 截取 ”方法调用，重新路由它们或者动态地添加功能。 动态代理为实现许多常见设计模式（包括 Facade、Bridge、Interceptor、Decorator、Proxy（包括远程和虚拟代理）和 Adapter 模式）提供了替代的动态机制。虽然这些模式不使用动态代理，只用普通的类就能够实现，但是在许多情况下，动态代理方式更方便、更紧凑，可以清除许多手写或生成的类。 Proxy模式 Proxy 模式中要创建“ stub ”或“ surrogate ”对象，它们的目的是接受请求并把请求转发到实际执行工作的其他对象。远程方法调用（RMI）利用 Proxy 模式，使得在其他 JVM 中执行的对象就像本地对象一样；企业 JavaBeans （EJB）利用 Proxy 模式添加远程调用、安全性和事务分界；而 JAX-RPC Web 服务则用 Proxy 模式让远程服务表现得像本地对象一样。在每一种情况中，潜在的远程对象的行为是由接口定义的，而接口本质上接受多种实现。调用者（在大多数情况下）不能区分出它们只是持有一个对 stub 而不是实际对象的引用，因为二者实现了相同的接口；stub 的工作是查找实际的对象、封送参数、把参数发送给实际对象、解除封送返回值、把返回值返回给调用者。代理可以用来提供远程控制（就像在 RMI、EJB 和 JAX-RPC 中那样），用

安全性策略包装对象（EJB）、为昂贵的对象（EJB 实体 Bean）提供惰性装入，或者添加检测工具（例如日志记录）。 在 5.0 以前的 JDK 中，RMI stub（以及它对等的 skeleton）是在编译时由 RMI 编译器（rmic）生成的类，RMI 编译器是 JDK 工具集的一部分。对于每个远程接口，都会生成一个 stub（代理）类，它代表远程对象，还生成一个 skeleton 对象，它在远程 JVM 中做与 stub 相反的工作 解除封送参数并调用实际的对象。类似地，用于 Web 服务的 JAX-RPC 工具也为远程 Web 服务生成代理类，从而使远程 Web 服务看起来就像本地对象一样。 不管 stub 类是以源代码还是以字节码生成的，代码生成仍然会向编译过程添加一些额外步骤，而且因为命名相似的类的泛滥，会带来意义模糊的可能性。另一方面，动态代理机制支持在编译时没有生成 stub 类的情况下，在运行时创建代理对象。在 JDK 5.0 及以后版本中，RMI 工具使用动态代理代替了生成的 stub，结果 RMI 变得更容易使用。许多 J2EE 容器也使用动态代理来实现 EJB。EJB 技术严重地依靠使用拦截（interception）来实现安全性和事务分界；动态代理为接口上调用的所有方法提供了集中的控制流程路径。 动态代理机制 动态代理机制的核心是 InvocationHandler 接口，如清单 1 所示。调用句柄的工作是代表动态代理实际执行所请求的方法调用。传递给调用句柄一个 Method 对象（从 java.lang.reflect 包），参数列表则传递给方法；在最简单的情况下，可能仅仅是调用反射性的方法 Method.invoke() 并返回结果。 清单 1. InvocationHandler 接口 public interface InvocationHandler { Object invoke(Object proxy, Method method, Object[] args) throws Throwable. } 每个代理都有一个与之关联

的调用句柄，只要代理的方法被调用时就会调用该句柄。根据通用的设计原则：接口定义类型、类定义实现，代理对象可以实现一个或多个接口，但是不能实现类。因为代理类没有可以访问的名称，它们不能有构造函数，所以它们必须由工厂创建。清单 2 显示了动态代理的最简单的可能实现，它实现 Set 接口并把所有 Set 方法（以及所有 Object 方法）分派给封装的 Set 实例。 清单 2. 包装 Set 的简单的动态代理 public class SetProxyFactory { public static Set getSetProxy(final Set s) { return (Set) Proxy.newProxyInstance (s.getClass().getClassLoader(), new Class[] { Set.class }, new InvocationHandler() { public Object invoke(Object proxy, Method method, Object[] args) throws Throwable { return method.invoke(s, args). } }). } } SetProxyFactory 类包含一个静态工厂方法 getSetProxy()，它返回一个实现了 Set 的动态代理。代理对象实际实现 Set 调用者无法区分（除非通过反射）返回的对象是动态代理。SetProxyFactory 返回的代理只做一件事，把方法分派给传递给工厂方法的 Set 实例。虽然反射代码通常比较难读，但是这里的内容很少，跟上控制流程并不难 只要某个方法在 Set 代理上被调用，它就被分派给调用句柄，调用句柄只是反射地调用底层包装的对象上的目标方法。当然，绝对什么都不做的代理可能有点傻，是不是呢？ 