怎样构建更好的企业应用程序异常处理框架 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/141/2021_2022__E6_80_8E_E6_A0_B7_E6_9E_84_E5_c29_141431.htm 企业应用程序在构建时常常对异常处理关注甚少，这会造成对低级异常（如 java.rmi.RemoteException 和 javax.naming.NamingException）的过度依赖。我们向客户机提供诸如 ApplicationException 和 InvalidDataException 之类的异常，而没有让 Web 层处理象 java.rmi.RemoteException 或 javax.naming.NamingException 这样的异常。 远程和命名异常是系统级异常，而应用程序和非法数据异常是业务级异常，因为它们提交更适用的业务信息。当决定抛出何种类型的异常时，您应该总是首先考虑将要处理所报告异常的层。 Web 层通常是由执行业务任务的最终用户驱动的，所以最好用它处理业务级异常。但是，在 EJB 层，您正在执行系统级任务，如使用 JNDI 或数据库。尽管这些任务最终将被合并到业务逻辑中，但是最好用诸如 RemoteException 之类的系统级异常来表示它们。 来源：www.examda.com 理论上，您可以让所有 Web 层方法预期处理和响应单个应用程序异常，正如我们在先前的一些示例中所做的一样。但这种方法不适用于长时间运行。让您的委派方法抛出更具体的异常，这是一个好得多的异常处理方案，从根本上讲，这对接收客户机更有用。在这篇技巧文章中，我们将讨论两种技术，它们将有助于您创建信息更丰富、更具体的异常，而不会生成大量不必要的代码。 嵌套的异常在设计可靠的异常处理方案时，要考虑的第一件事情就是对所谓的低级或系统级异常进行抽象化。这些核心 Java 异常通

常会报告网络流量中的错误、JNDI 或 RMI 问题，或者是应用程序中的其它技术问题。RemoteException、EJBException 和 NamingException 是企业 Java 编程中低级异常的常见例子。 这些异常完全没有任何意义，由 Web 层的客户机接收时尤其容易混淆。如果客户机调用 purchase() 并接收到 NamingException，那么它在解决这个异常时会一筹莫展。同时，应用程序代码可能需要访问这些异常中的信息，因此不能轻易地抛弃或忽略它们。 答案是提供一类更有用的异常，它还包含低级异常。清单 1 演示了一个专为这一点设计的简单 ApplicationException： 清单 1. 嵌套的异常 package com.ibm. import java.io.PrintStream. import java.io.PrintWriter. public class ApplicationException extends Exception { /** A wrapped Throwable */ protected Throwable cause. public ApplicationException() { super("Error occurred in application."). } public ApplicationException(String message) { super(message). } public ApplicationException(String message, Throwable cause) { super(message). this.cause = cause. } // Created to match the JDK 1.4 Throwable method. public Throwable initCause(Throwable cause) { this.cause = cause. return cause. } public String getMessage() { // Get this exceptions message. String msg = super.getMessage(). Throwable parent = this. Throwable child. // Look for nested exceptions. while((child = getNestedException(parent)) != null) { // Get the childs message. String msg2 = child.getMessage(). // If we found a message for the child exception, // we append it. if (msg2 != null) { if (msg != null) { msg = ": " msg2. } else { msg = msg2. } } // Any nested ApplicationException will append its own // children, so

we need to break out of here. if (child instanceof ApplicationException) { break. } parent = child. } // Return the completed message. return msg. } public void printStackTrace() { // Print the stack trace for this exception. super.printStackTrace(). Throwable parent = this. Throwable child. // Print the stack trace for each nested exception. while((child = getNestedException(parent)) != null) { if (child != null) { System.err.print("Caused by: "). child.printStackTrace(). if (child instanceof ApplicationException) { break. } parent = child. } } } public void printStackTrace(PrintStream s) { // Print the stack trace for this exception. super.printStackTrace(s). Throwable parent = this. Throwable child. // Print the stack trace for each nested exception. while((child = getNestedException(parent)) != null) { if (child != null) { s.print("Caused by: "). child.printStackTrace(s). if (child instanceof ApplicationException) { break. } parent = child. } } } public void printStackTrace(PrintWriter w) { // Print the stack trace for this exception. super.printStackTrace(w). Throwable parent = this. Throwable child. // Print the stack trace for each nested exception. while((child = getNestedException(parent)) != null) { if (child != null) { w.print("Caused by: "). child.printStackTrace(w). if (child instanceof ApplicationException) { break. } parent = child. } } } public Throwable getCause() { return cause. } } 清单 1 中的代码很简单；我们已经简单地将多个异常“ 串 ”在一起，以创建单个、嵌套的异常。但是，真正的好处在于将这种技术作为出发点，以创建特定于应用程序的异常层次结构。异常层次结构将允许 EJB 客户机既接收特定于业务的异常也接收特定于

系统的信息，而不需要编写大量额外代码。 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com