Java软件编程规范说明书

[一起种水果系统]

团 队：3+1团队

指导老师：代祖华

目录

[1 引言 3](#_Toc10706542)

[2 编码规范 3](#_Toc10706543)

[2 5](#_Toc10706544)

[2 6](#_Toc10706545)

[2 8](#_Toc10706546)

[2 9](#_Toc10706547)

[2 9](#_Toc10706548)

[2 14](#_Toc10706549)

[2 14](#_Toc10706550)

[2 15](#_Toc10706551)

[2 15](#_Toc10706552)

[2.2 Jsp 编码规范 16](#_Toc10706553)

[2 18](#_Toc10706554)

1 引言

代码规范提高软件代码的可读性，使得开发人员快速和彻底的理解新代码。好的代码风格不仅会提高可读性，而且会使代码更健壮，更为重要的是在修改时不容易出错。在现代软件开发中，维护工作会占用 80% 的时间，而且开发者和维护者通常不是同一个程序员。这意味着你经常要阅读和修改别人开发的程序，别人也同样可能需要阅读和修改你开发的程序。既然如此，为什么不把这利人利己的事情作好呢？一些习惯自由程序人员可能对这些规则很不适应，但是在多个开发人员共同写作的情况下，这些规则是必需的。

2 编码规范  
2.1 Java 编码规范  
2.1.1 命名规范  
 定义这个规范的目的是让项目中所有的文档都看起来像一个人写的，增加可读性，减少项目组中因为换人而带来的损失。（这些规范并不是一定要绝对遵守，但是一定要让程序有良好的可读性）  
  
使用匈牙利表示法  
**Package** 的命名  
Package 的名字应该都是由一个小写单词组成。  
package com。neu。util

**Class** 的命名  
Class 的名字必须由大写字母开头而其他字母都小写的单词组成 ,对于所有标识符，其中包  
含的所有单词都应紧靠在一起，而且大写中间单词的首字母。  
public class ThisAClassName{}

**Class** 变量的命名  
变量的名字必须用一个小写字母开头。后面的单词用大写字母开头  
userName，thisAClassMethod

**Static Final** 变量的命名  
static Final 变量的名字应该都大写，并且指出完整含义。  
/\*\*  
\*DBConfig PATH  
\*\*/  
public static final String  
DB\_CONFIG\_FILE\_PATH ="com。neu。etrain。dbconfig";

参数的命名  
参数的名字必须和变量的命名规范一致。

数组的命名  
数组应该总是用下面的方式来命名：  
byte[] buffer;  
而不是： byte buffer[];  
  
方法的参数  
使用有意义的参数命名，如果可能的话，使用和要赋值的字段一样的名字：  
SetCounter(int size){  
this。size = size;}

2.1.2 变量定义规范  
1。 去掉没必要的公共变量。  
2。 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共  
变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。  
3。 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。  
4。 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。  
5。 当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生  
6。 防止局部变量与公共变量同名。  
7。 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减  
少引起误用现象。  
8。 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保  
留余地（如预留一些空间等） 。  
9。 留心具体语言及编译器处理不同数据类型的原则及有关细节。  
10。 严禁使用未经初始化的变量。声明变量的同时对变量进行初始化。  
11。 编程时，要注意数据类型的强制转换。  
2.1.3 代码编写格式  
{} 对  
{} 中的语句应该单独作为一行。例如, 下面的第 1 行是错误的，第 2 行是正确的 :  
if (i>0) { i ++ }; // 错误，{ 和 } 在同一行  
if (i>0) {  
i ++  
}; // 正确，{ 单独作为一行 a  
} 语句永远单独作为一行 。  
如果 } 语句应该缩进到与其相对应的 { 那一行相对齐的位置。  
  
括 号  
左括号和后一个字符之间不应该出现空格，同样，右括号和前一个字符之间也不应该  
出现空格。下面的例子说明括号和空格的错误及正确使用 :  
CallProc( AParameter ); // 错误  
CallProc(AParameter); // 正确  
不要在语句中使用无意义的括号。括号只应该为达到某种目的而出现在源代码中。下  
面的例子说明错误和正确的用法 :  
if ((I) = 42) { // 错误 - 括号毫无意义  
if (I == 42) or (J == 42) then // 正确 - 的确需要括号

2.1.4 注释规范  
定义这个规范的目的是让项目中所有的文档都看起来像一个人写的，增加可读性，减少  
项目组中因为换人而带来的损失。 （这些规范并不是一定要绝对遵守，但是一定要让程序有良  
好的可读性） 。  
Java的语法与 C++ 及为相似，那么，你知道 Java 的注释有几种吗？是两种？  
注释一行  
注释若干行  
不完全对，除了以上两种之外，还有第三种，文档注释：  
注释若干行，并写入 javadoc 文档  
注释要简单明了。  
String userName = null; // 用户  
边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。  
在必要的地方注释，注释量要适中。注释的内容要清楚、明了，含义准确，防止注释二义  
性。保持注释与其描述的代码相邻，即注释的就近原则。  
对代码的注释应放在其上方相邻位置， 不可放在下面。 对数据结构的注释应放在其上方相  
邻位置，不可放在下面； 对结构中的每个域的注释应放在此域的右方；同一结构中不同域  
的注释要对齐。  
变量、常量的注释应放在其上方相邻位置或右方。全局变量要有较详细的注释，包括对其  
功能、取值范围、 哪些函数或过程存取它以及存取时注意事项等的说明。在每个源文件的  
头部要有必要的注释信息，包括：文件名；版本号；作者；生成日期；模块功能描述（如  
功能、主要算法、内部各部分之间的关系、该文件与其它文件关系等） ；主要函数或过程  
清单及本文件历史修改记录等。  
/\*\*  
  
\* Copy Right Information : Neusoft IIT  
\* Project : eTrain  
\* JDK version used : jdk1。3。1  
\* Comments : config path  
\* Version : 1。01  
\* Modification history :2003。5。1  
\* Sr Date Modified By Why & What is modified  
\* 1。 2003。5。2 Kevin Gao new  
\*\*/  
在每个函数或过程的前面要有必要的注释信息， 包括： 函数或过程名称； 功能描述； 输入、  
输出及返回值说明；调用关系及被调用关系说明等  
/\*\*  
\* Description :checkout 提款  
\* @param Hashtable cart info  
\* @param OrderBean order info  
\* @return String  
\*/  
public String checkout(Hashtable htCart,  
OrderBean orderBean)  
throws Exception{  
}

**javadoc**注释标签语法  
@author 对类的说明 标明开发该类模块的作者  
@version 对类的说明 标明该类模块的版本  
@see 对类、属性、方法的说明 参考转向，也就是相关主题  
@param 对方法的说明 对方法中某参数的说明  
@return 对方法的说明 对方法返回值的说明  
@exception 对方法的说明 对方法可能抛出的异常进行说

2.1.5 函数、过程  
1。 函数的规模尽量限制在 200 行以内。  
2。 一个函数最好仅完成一件功能。  
3。 为简单功能编写函数。  
4。 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。  
5。 尽量不要编写依赖于其他函数内部实现的函数。  
6。 避免设计多参数函数，不使用的参数从接口中去掉。  
7。 用注释详细说明每个参数的作用、取值范围及参数间的关系。  
8。 检查函数所有参数输入的有效性。  
9。 检查函数所有非参数输入的有效性，如数据文件、公共变量等。  
10。 函数名应准确描述函数的功能。  
11。 避免使用无意义或含义不清的动词为函数命名  
12。 函数的返回值要清楚、明了，让使用者不容易忽视错误情况。  
13/ 明确函数功能，精确（而不是近似）地实现函数设计。  
14。 减少函数本身或函数间的递归调用。  
15。 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、 V 操作）等手段对其加以保护。

2.1.6 编程技巧  
**byte** 数组转换到 **characters**  
为了将 byte 数组转换到 characters，你可以这么做：  
"Hello world!"。getBytes();  
**Utility** 类  
Utility 类（仅仅提供方法的类）应该被申明为抽象的来防止被继承或被初始化。  
11  
初始化  
下面的代码是一种很好的初始化数组的方法：  
objectArguments = new Object[] { arguments };  
枚举类型  
JAVA 对枚举的支持不好，但是下面的代码是一种很有用的模板：  
class Colour {  
public static final Colour BLACK = new Colour(0, 0, 0);  
public static final Colour RED = new Colour(0xFF, 0, 0);  
public static final Colour GREEN = new Colour(0, 0xFF, 0);  
public static final Colour BLUE = new Colour(0, 0, 0xFF);  
public static final Colour WHITE = new Colour(0xFF, 0xFF, 0xFF);  
}  
这种技术实现了 RED, GREEN, BLUE 等可以象其他语言的枚举类型一样使用的常量。 他们  
可以用 '==' 操作符来比较。  
但是这样使用有一个缺陷：如果一个用户用这样的方法来创建颜色 BLACK  
new Colour(0,0,0)  
那么这就是另外一个对象， '=='操作符就会产生错误。她的 equal() 方法仍然有效。由于  
这个原因，这个技术的缺陷最好注明在文档中，或者只在自己的包中使用。

2.1.7 程序编写规范  
**exit()**  
exit 除了在 main 中可以被调用外，其他的地方不应该调用。因为这样做不给任何代码  
代码机会来截获退出。 一个类似后台服务地程序不应该因为某一个库模块决定了要退出就  
退出。

异常  
申明的错误应该抛出一个 RuntimeException或者派生的异常。  
顶层的 main()函数应该截获所有的异常，并且打印（或者记录在日志中）在屏幕上。

垃圾收集  
JAVA 使用成熟的后台垃圾收集技术来代替引用计数。但是这样会导致一个问题：你必须  
在使用完对象的实例以后进行清场工作。比如一个 prel 的程序员可能这么写：  
。。。  
{  
FileOutputStream fos = new FileOutputStream(projectFile);  
project。save(fos, "IDE Project File");  
}  
。。。  
除非输出流一出作用域就关闭，非引用计数的程序语言，比如 JAVA，是不能自动完成变  
量的清场工作的。必须象下面一样写：  
FileOutputStream fos = new FileOutputStream(projectFile);  
project。save(fos, "IDE Project File");  
fos。close();

**Clone**  
下面是一种有用的方法：  
implements Cloneable  
public  
Object clone()  
{  
try {  
ThisClass obj = (ThisClass)super。clone();  
obj。field1 = (int[])field1。clone();  
obj。field2 = field2;  
return obj;  
} catch(CloneNotSupportedException e) {  
throw new InternalError("Unexpected CloneNotSUpportedException: " + e。getMessage());  
}  
}  
  
**final** 类  
绝对不要因为性能的原因将类定义为 final 的（除非程序的框架要求）  
如果一个类还没有准备好被继承， 最好在类文档中注明， 而不要将她定义为 final 的。  
这是因为没有人可以保证会不会由于什么原因需要继承她。  
访问类的成员变量  
大部分的类成员变量应该定义为 protected 的来防止继承类使用他们。  
注意，要用 "int[] packets" ，而不是 "int packets[]"，后一种永远也不要用。  
public void setPackets(int[] packets) { this。packets = packets; }  
CounterSet(int size)  
{  
this。size = size;  
}  
2.1.8 排版规范  
关键词和操作符之间加适当的空格。  
相对独立的程序块与块之间加空行  
较长的语句、表达式等要分成多行书写。  
划分出的新行要进行适应的缩进，使排版整齐，语句可读。  
长表达式要在低优先级操作符处划分新行，操作符放在新行之首。  
循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分。  
若函数或过程中的参数较长，则要进行适当的划分。  
不允许把多个短语句写在一行中，即一行只写一条语句。  
函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格。  
C/C++语言是用大括号‘{ ’和‘ } ’界定一段程序块的，编写程序块时‘ {’和‘ } ’应各  
独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、  
类的定义、结构的定义、枚举的定义以及 if 、 for、 do、 while、 switch、 case语句中的程序  
都要采用如上的缩进方式。  
  
2.1.9 Java 文件格式  
所有的 Java(\*。java) 文件都必须遵守如下的样式规则：  
版权信息  
版权信息必须在 java 文件的开头，比如：  
/\*  
\* Copyright ? 2000 Shanghai XXX Co。 Ltd。  
\* All right reserved。  
\*/  
其他不需要出现在 javadoc 的信息也可以包含在这里。

**Package/Imports**  
package 行要在 import 行之前， import 中标准的包名要在本地的包名之前，而且按照字母顺  
序排列。如果 import 行中包含了同一个包中的不同子目录，则应该用 \* 来处理。  
package hotlava。net。stats;  
import java。io。\*;  
import java。util。Observable;  
import hotlava。util。Application;  
这里 java。io。\* 使用来代替 InputStream and OutputStream的。

**Class**  
接下来的是类的注释，一般是用来解释类的。  
/\*\*  
\* A class representing a set of packet and byte counters  
\* It is observable to allow it to be watched, but only  
\* reports changes when the current set is complete  
\*/  
接下来是类定义，包含了在不同的行的 extends 和 implements  
15  
public class CounterSet  
extends Observable  
implements Cloneable

**Class Fields**  
接下来是类的成员变量：  
/\*\*  
\* Packet counters  
\*/  
protected int[] packets;  
public 的成员变量必须生成文档（ JavaDoc）。 proceted、 private 和 package 定义的成员变量如  
果名字含义明确的话，可以没有注释。  
存取方法  
接下来是类变量的存取的方法。它只是简单的用来将类的变量赋值获取值的话，可以简单的  
写在一行上。  
/\*\*  
\* Get the counters  
\* @reQturn an array containing the statistical data。 This array has been  
\* freshly allocated and can be modified by the caller。  
\*/  
public int[] getPackets() { return copyArray(packets, offset); }  
public int[] getBytes() { return copyArray(bytes, offset); }  
public int[] getPackets() { return packets; }  
public void setPackets(int[] packets) { this。packets = packets; }  
其它的方法不要写在一行上  
  
构造函数  
接下来是构造函数，它应该用递增的方式写（比如：参数多的写在后面） 。  
访问类型 ("public", "private" 等。) 和 任何 "static", "final" 或 "synchronized" 应该在一行中，  
并且方法和参数另写一行，这样可以使方法和参数更易读。  
public  
CounterSet(int size){  
this。size = size;  
}

类方法  
下面开始写类的方法：  
/\*\*  
\* Set the packet counters  
\* (such as when restoring from a database)  
\*/  
protected final  
void setArray(int[] r1, int[] r2, int[] r3, int[] r4)  
throws IllegalArgumentException  
{  
//  
// Ensure the arrays are of equal size  
//  
if (r1。length != r2。length || r1。length != r3。length || r1。length != r4。length)  
throw new IllegalArgumentException("Arrays must be of the same size");  
System。arraycopy(r1, 0, r3, 0, r1。length);  
System。arraycopy(r2, 0, r4, 0, r1。length);  
}

**toString** 方法  
无论如何，每一个类都应该定义 toString 方法：  
public  
String toString() {  
String retval = "CounterSet: ";  
for (int i = 0; i < data。length(); i++) {  
retval += data。bytes。toString();  
retval += data。packets。toString();  
}  
return retval;  
}  
}  
  
**main** 方法  
如果 main(String[]) 方法已经定义了，那么它应该写在类的底部 。

2.1.10 可读性  
避免使用不易理解的数字，用有意义的标识来替代。  
不要使用难懂的技巧性很高的语句。  
源程序中关系较为紧密的代码应尽可能相邻。

2.1.11 性能  
在写代码的时候， 从头至尾都应该考虑性能问题。 这不是说时间都应该浪费在优化代码上，  
而是我们时刻应该提醒自己要注意代码的效率。比如：如果没有时间来实现一个高效的算法，  
那么我们应该在文档中记录下来，以便在以后有空的时候再来实现她。  
不是所有的人都同意在写代码的时候应该优化性能这个观点的， 他们认为性能优  
化的问题应该在项目的后期再去考虑，也就是在程序的轮廓已经实现了以后。  
不必要的对象构造  
不要在循环中构造和释放对象

使用 **StringBuffer** 对象  
在处理 String 的时候要尽量使用 StringBuffer 类， StringBuffer 类是构成 String 类的基  
础。 String 类将 StringBuffer 类封装了起来， （以花费更多时间为代价）为开发人员提供了一  
个安全的接口。当我们在构造字符串的时候，我们应该用 StringBuffer 来实现大部分的工作，  
当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。比如：如果有一个字符串必  
须不断地在其后添加许多字符来完成构造，那么我们应该使用 StringBuffer 对象和她的  
append() 方法。如果我们用 String 对象代替 StringBuffer 对象的话，会花费许多不必要的创  
建和释放对象的 CPU 时间。

避免太多的使用 **synchronized** 关键字  
避免不必要的使用关键字 synchronized，应该在必要的时候再使用她，这是一个避免死锁的好方法。

2.1.12 可移植性  
换 行  
如果需要换行的话，尽量用 println 来代替在字符串中使用 "\n"。  
你不要这样：  
System。out。print("Hello,world!\n");  
要这样：  
System。out。println("Hello,world!");  
或者你构造一个带换行符的字符串，至少要象这样：  
String newline = System。getProperty("line。separator");  
System。out。println("Hello world" + newline);

**PrintStream**  
PrintStream 已经被不赞成（deprecated ）使用，用 PrintWrite 来代替她。

2.1.13 代码测试、维护  
1。单元测试要求至少达到语句覆盖。  
2。单元测试开始要跟踪每一条语句，并观察数据流及变量的变化。  
3。清理、整理或优化后的代码要经过审查及测试。  
4。代码版本升级要经过严格测试。  
  
2.1.14 质量保证  
1。在软件设计过程中构筑软件质量。  
代码质量保证优先原则  
（ 1）正确性，指程序要实现设计要求的功能。  
（ 2）稳定性、安全性，指程序稳定、可靠、安全。  
（ 3）可测试性，指程序要具有良好的可测试性。  
（ 4）规范 /可读性，指程序书写风格、命名规则等要符合规范。  
（ 5）全局效率，指软件系统的整体效率。  
（ 6）局部效率，指某个模块 /子模块 /函数的本身效率。  
（ 7）个人表达方式 /个人方便性，指个人编程习惯。  
2。只引用属于自己的存贮空间。  
3。防止引用已经释放的内存空间。  
4。过程 /函数中分配的内存，在过程 /函数退出之前要释放。  
5。过程 /函数中申请的（为打开文件而使用的）文件句柄，在过程 /函数退出前要关闭。  
6。防止内存操作越界。  
7。时刻注意表达式是否会上溢、下溢。  
8。认真处理程序所能遇到的各种出错情况。  
9。系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。  
10。系统运行之初，要对加载到系统中的数据进行一致性检查。  
11。严禁随意更改其它模块或系统的有关设置和配置。  
12。不能随意改变与其它模块的接口。  
13。充分了解系统的接口之后，再使用系统提供的功能。  
14。要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符。  
15。不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句。  
16。建议：使用第三方提供的软件开发工具包或控件时，要注意以下几点：  
（ 1）充分了解应用接口、使用环境及使用时注意事项。  
（ 2）不能过分相信其正确性。  
（ 3）除非必要，不要使用不熟悉的第三方工具包与控件。  
  
2.1.15 代码编译  
1。编写代码时要注意随时保存，并定期备份， 防止由于断电、硬盘损坏等原因造成代码丢  
失。  
2。同一项目组内，最好使用相同的编辑器，并使用相同的设置选项。  
3。合理地设计软件系统目录，方便开发人员使用。  
4。打开编译器的所有告警开关对程序进行编译。  
5。在同一项目组或产品组中，要统一编译开关选项。

2.2 Jsp 编码规范  
。整个 jsp/jsp bean表示层应当尽可能的瘦和简单化。  
。牢记大多数的 JSP都应当是只读的视图，而由页面 bean来提供模型。  
。应当一起设计 JSP和 JSP bean  
。在尽可能合理的情况下，把业务逻辑从 JSP中移走。具体于 HTTP 的逻辑（如，对 Cookie  
的处理）属于 bean或支持类中，而不是 JSP中。  
。尽量把条件逻辑放在控制器中而不是放在视图中。  
。为 JSP、包含的文件、 JSP Bean和实现扩展标记的类使用遵循标准的命名惯例如：  
jsp 控制器 xxxxController。jsp  
被包含的 : jsp \_descriptiveNameOfFragment。jsp  
jsp 页面模型 bean: <pagename>Bean如 loginBena。java  
jsp 会话 bena: xxxxSessionBean  
标记类 : xxxxTag,xxxxTagExtraInfo  
。 应当在 JSP中避免使用页面引入指令。 import 指令会促使类的实列化而不是 jsp bean的实例化：  
不用： <%@ page import = "com。java。util。\*" %>  
而用： <% java。util。List l = new java。util。LinkedList(); %>  
。 jsp 不应该直接去访问请求参数。 bean应当执行这样的处理过程并且输出所处理的模型数据。

jsp 不应当访问属性文件，或者使用 JNDI。 bean可以访问属性。  
。如果 jsp bean具有的所有的属性不能够从页面请求中被映射到，就要尽力在 <jsp:useBean>  
标记中设置属性。  
。应当避免设计既显示表单又处理结果的页面。  
。在 jsp 中避免代码重复。把要重复的功能放在一个包含的 jsp、 bean或标记扩展中，使得它  
能够被重用。  
。 jsp bean应当永远不要去产生 HTML  
。在 jsp 中应该避免使用 out。pringln()发放来产生页面内容。  
。 jsp 层不应该直接访问数据，这包括 JDBC 数据库访问和 EJB 访问。  
。在长度上， scriptlests 的代码最好不要超过 5 行。  
。除了 jsp bean之外， jsp 不应当去实例化复杂的可读写的对象。如果这样的话，就有可能在  
jsp 中去执行不适当的业务逻辑。  
。 jsp bean中不应当包含大量的数据。  
。如果使用了 <jsp:forward>和<jsp:include 标记 >，并且必须使用简单类型的值来与外部页面进  
行通讯的话，就应当使用一个或多个 <jsp:param>元素  
。定制标记应当用在适当把逻辑从 jsp 中移走的地方。  
。应当谨慎地使用 <jsp:forward>标记，在 jsp 中它是一个等价的 goto。  
。应当使用隐藏的注释来阻止输出的 HTML 过大。  
。在 jsp 中避免进行异常处理  
。每个 jsp 文件中都应当使用一个错误页面来处理不能够从中恢复的异常。  
。在 jsp 错误页面中，使用 HTML 注释来显示传递到该页面中的异常跟踪信息。  
。只有在能够获得性能上的好处时，才使用 jspInin()方法和 jspDestroy()方法。获取和放弃资  
源是 jsp beans和标记处理器的事，而不是由 jsp 来负责的。  
。如果没有充分的理由，就不要在 jsp 中定义方法。  
2.3 Struts 编码规范  
2.3.1 Action,Form,Bean 命名规则  
**Action**：命名以 **Action**结尾 **,**  
例：  
添加新闻的 Actoin,命名为 AddNewsAction  
修改新闻的 Action,命名为 UpNewsAction  
删除新闻的 Action,命名为 DelNewsAction  
查看新闻的 Action,命名为 ViewNewsAction  
  
**Form**：命名以 **Form** 结尾 **,**  
例：  
操作新闻的 Form，命名为 NewsForm’

**Bean:** 命名以 **Bean** 结尾，  
例：  
操作新闻的 Bean，命名为 NewsBean

2.3.2 struts-config.xml 的定义规则  
**form** 属性定义规则  
<form-bean name="testimgForm" type="cn。。TestimgForm" />

**action**属性定义规则

<action  
attribute="testimgForm"  
name="viewForm"  
path="/admin/example/viewTableAction"  
scope="request"  
type="cn。struts。action。example。ViewTableAction" />