2005 年第 12 期

未找到。”在本文中，**Uwe Habermann** 将会解释如何导致这个错误的，并提出几个解决办法。

Andy Kramek & Marcia Akins

作者：**Uwe Habermann**　译者：**fbilo**

无论你的数据是多么的完整，在很多情况下还是用图形化来表示信息的办法更简单。一种常见的特殊需求是根据时间进行数据分析。察看图形格式的数据要比死盯着表格中大量的行和列容易的多。然而，**VFP** 自己并没有内建的工具让我们可以由数据来建立图形，所以我们必须求助于别的方法。这个月，**Andy Kramek** 和 **Marcia Akins** 要解决用 **VFP** 代码和数据在 **Excel** 中建立一个图形的任务。

Log4Fox：一个为 VFP 开发的日志 API

作者：Lisa Slater Nicholls

译者：fbilo

日志在任何开发环境中都是一个标准的需求。在这篇文章中，Lisa Slater Nicholls 采用了一个基础的日志库 Log4j，并将之应用到 VFP 里。

要真正理解在你的应用程序运行时发生了什么事情，你必须用武器装备它，以使它提供在任何代码领域的跟踪数据。为了实现这个目标，Microsoft 的 .NET FrameWork 有 System.Diagnostics.TraceListener 类，而 Sun 的 Java 平台则包含 java.util.logging 包。

尽管 Visual FoxPro 的 SET COVERAGE、SET DEBUGOUT、和 SET EVENTTRACKING 已经在各个版本中被逐步的增强了，但 VFP 还是没有象其它开发环境那样尽可能完整的提供了对跟踪数据的层次和类型最佳配置的内部支持。没关系，VFP 的 OOP 和输出处理资源使得我们可以轻松的提供用于处理所有这些需求任务的类库。

当你研究在不同环境中的设备（instrumentation）API 的时候，你会发现其用法都是大同小异的。为 VFP 使用经典的日志模式也是有意义的。当你听到我所写的那个基于 VFP 的日志类库时没有遵循微软的或者 Sun 的引导时，你可以会惊讶。然而，在开源开发世界里有一个著名的成功者：log4j，一个 Java 工具包，看上去它的应用范围要比销售商的解决方案广泛的多。这个 log4j 工具包的功能明显比 java.util.logging 的完整得多。你还会发现许多 .NET 的开发人员们也选择了模仿 log4j 的 API 和功能，在可能的时候甚至直接封装了它。

当我在 VFP 中工作的时候，因为觉得缺少了 log4j 的功能，所以决定在本文附带的源代码中的 PRG 类库 Log4Fox 来模仿它。

一个日志 API 是如何工作的

我缺少的那些属于 log4j 的功能可以简单的用大多数别的语言对日志 API 的实现的一个描述来说明。它们通常都有以下类类型：

- 一个输出观察员（observers，一种设计模式的名称）的集合，其中每一个观察员都明白如何输出到一个特定的目标设备。用设计模式的术语来说，一个应用程序的运行——比如一个 REPORT FORM 命令的运行——是一个可观察的事件，而

一个输出观察员，例如 VFP 自己的 ReportListener 类，在该事件的进程中采取独立的操作。例如，一个对象在给管理员发送 Email 的同时，另一个对象正在写系统事件日志。许多观察员类型被称为监听者（listeners），因为它们监视着应用程序的活动，就好像是从听到的交响乐中寻找它们想要的音符和旋律一样。log4j 工具包把这些对象叫做添加者（appenders），因为它们的功能就是，发送消息给一个存储信息的目标（就好像往一个仓库里搬东西一样）。

- 一个观察员集合的拥有者（owner）对象，你的应用程序指定它作为输出目标。这个对象有着一些你可以用来开始和停止记录日志的方法、以及令一个用于发送你的事件消息的方法。除非你想要定制观察员对象，否则，在你的应用程序运行时，你就只需要跟这个拥有者对象交互就可以了，根本不用碰那个观察员对象的集合或者该集合的对象引用。
- 一个格式化或者布局接口，它通常作为对每个输出对象的一个 helper 对象来提供。这样可以提供一种标准的途径以指定如何对消息进行修饰、同时还需要每一个观察员为它们的内容使用不同类型的修饰方式。例如，一个 Email 观察员也许会给消息添加 HTML 格式，而事件日志观察员则也许会对消息的长度有限制。

这三种类的类型通常是“紧密耦合”的，这表示它们是被专门设计为一起协同工作的。在许多情况下，它们如果不能在正确的环境中找到自己就会拒绝被实例化。你可以从图 1 中看到这些类之间的关系。

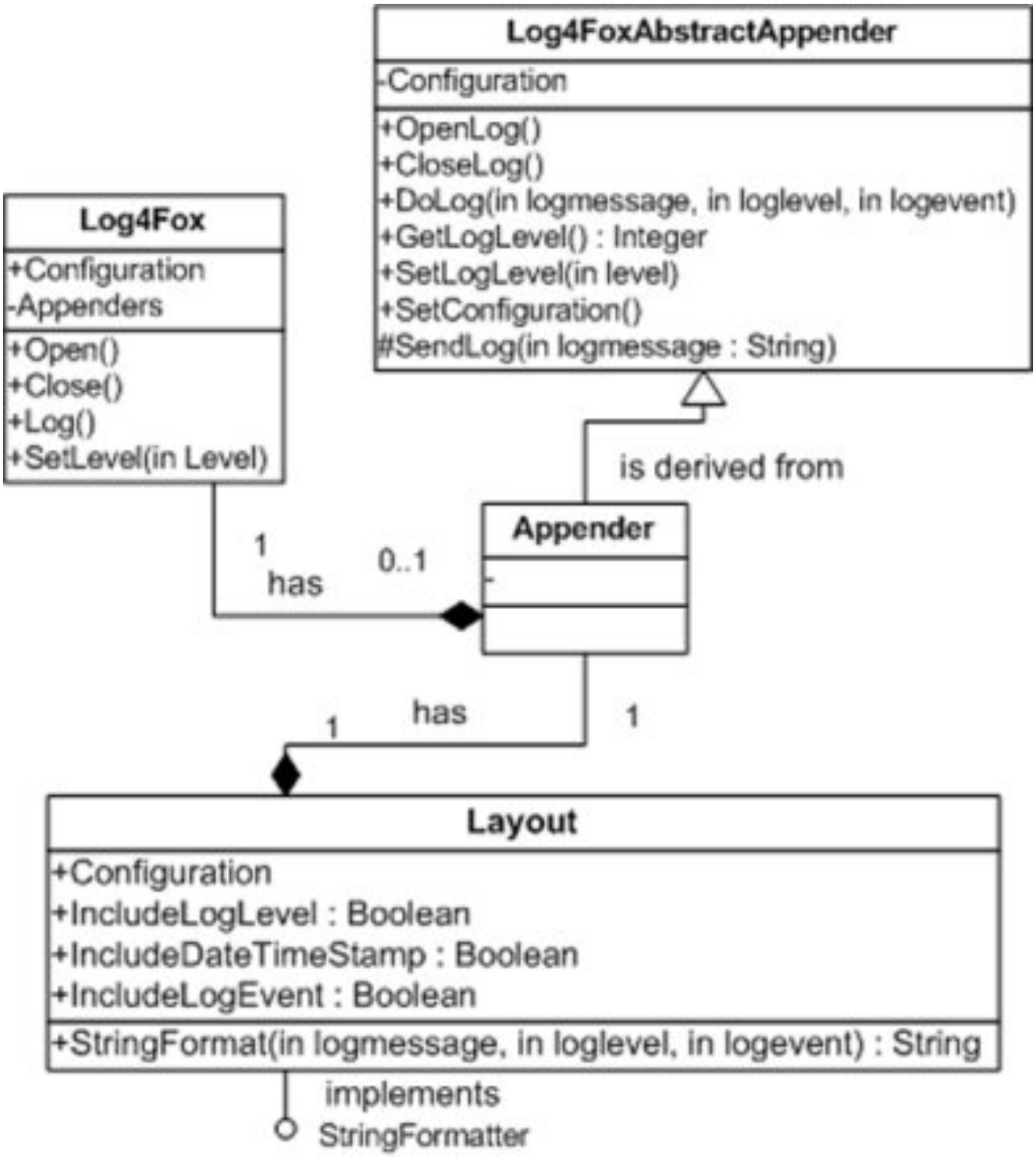

图 1、Log4Fox 类图

除了这些标准的类类型（**Class types**）和接口以外，每个 **API** 还包含着一些消息级别或者安全性级别的想法。例如，**System.Diagnostics.TraceLevel** 支持 **Off**、**Error**、**Warning**、**Info**、和 **Verbose(**详情**)**五种级别，而 **java.utility.logging** 有着从 **Finest** 到 **SERVER** 的七种级别。你可以分别为每个观察员设置一个消息必须匹配或者完成的安全性级别、它的输出类型等等。

当你发送一个事件消息的时候，你也可以执行你正在发送的消息的级别，观察员会将这个消息的级别与“观察员自己被设定的记录日志的级别”进行比较。消息的级别要能触发输出，就必须高于或者等于观察员的级别。

下面的第一个消息事件示例将触发一个来自于被设置为 **Info** 或者 **Debug** 级别的一个观察员的输出，但它不会触发来自于被设置为 **Warning** 级别的观察员的输出。第二行

则对两个观察员的输出都会触发。

```
MyApplication.MyLoggerImplementation.Log(;
    "User chose to run this report: " + ;
    lcReport, ;
    LOGLEVEL_INFO)

MyApplication.MyLoggerImplementation.Log(;
    "Cannot run the following program: " + ;
    lcApp, ;
    LOGLEVEL_ERROR)
```

Log4Fox(象 **log4j** 一样)支持 **FATAL(**致命的**)**、**ERROR**、**WARN**、**INFO** 和 **DEBUG** 几种以安全性的降序排列、或者（换一种思考方法）以详细程度的升序排列的级别。我给 **Log4Fox** 在 **DEBUG** 后面添加 **ALL** 作为一个新的级别，这个 **ALL** 级别意味着“不管我是否正在调试（**Debug**），都要把每一条消息都记录到日志里。”我不想改变 **log4j** 的核心枚举值，但是我发现 **DEBUG** 这个标题对于其目的 **VERBOSE** （详情）来说很容易混淆。

这个“结构化层次的消息精确级别”的概念，对“你的管理和调整你的程序跟踪和应用程序消息策略”来说是很关键的。例如，你可以给一个 **Email** 观察员设置为一个 **LOGLEVEL_ERROR** 级别，那么在某些重要事情发生的时候，管理员得到的只有 **Email**。对于同一个应用程序，你可以为事件日志观察员设置 **LOGLEVEL_INFO** 或者更高级别的消息。

当然，每个观察员还可以根据消息级别选择不同类型的操作。在这个示例中的事件日志观察员可以为 **LOGLEVEL_WARNING** 和 **LOGLEVEL_ERROR** 事件写一种类型的消息，也可以为 **LOGLEVEL_INFO** 事件写另一种不同类型的消息。

为一个 logging API 设置选项

在 **loggin API** 中的另一个常用功能，是它们的“将记录日志过程设置为使用某些类型设置对象或者文件”的能力。无论它是如何实现的，可设置资源都允许记录日志的 **Owner** 对象去收集它自己的观察员的集合，并设置它们的选项而不需要额外的代码。

Log4j 中也是这样，它使用一个 **XML** 文件作为全局的和额外的设置。**Log4Fox** 在这里偏离了 **log4j** 的轨道，它提供了一种抽象的设置能力，可以使用任何类型来实现这种能力。

我已经提供了一个实现的示例 **Log4FoxConfigAlias**，它使用一个游标格式来保持对

Log4Fox 的 **VFP** 继承。不过你当然可以使用 **log4j** 标准的 **XML** 设置文件格式、或者任何你喜欢的预定目标的设置对象以及其它 **Log4Fox** 派生的类。

在 **Log4Fox** 中，我已经包含了一个 **appender (Log4FoxSQLAppender)**，它跟 **Log4FoxConfigAlias** 一样使用一个游标以记录一些设置选项，而类库中的其它 **appenders** 则不会去读该游标。这么做了以后，**Log4FoxSQLAppender** 还是可以与除了 **Log4FoxConfigAlias** 以外的其它 **Log4Fox** 类一起工作，而 **Log4FoxConfigAlias** 也仍然可以建立类库中其它 **appender** 类的实例、并按照它的全局规则设置它们。总之，这里的差异仅仅是为了描述的目的罢了；通常你建立的是“能够跟它们的 **owner** 识别同一个设置设备”的观察员。记住：它们是紧密耦合的类类型。在使用你的日志记录类系列的不同成员的时候，你应该能够指望使用统一的设置。

Log4Fox 有一个标准的格式化类 **Log4FoxBaseFormatter**，在这个类库中的所有 **appender** 都使用这个类。如果你没有设置一个带有“对你希望采用何种格式的更明确说明”的 **appender** 类，抽象的 **appender** 父类 **Log4FoxAbstractAppender** 将提供这个（对 **Log4FoxBaseFormatter** 的）引用。当 **Appender** 们在 **Init** 事件过程中接收到设置信息的时候，会建立 **formater** 的实例，此后，任何时候你改动了它们的设置信息，它们都可以选择换一个 **formatters**。在 **Log4FoxAbstractAppender** 的源代码中，你可以看到用于这个目的的 **GetFormatter**、**HasValidFormatter** 、和 **GetDefaultFormatter** 方法，以及这个类中没有包含在图 **1** 里面的其它一些受保护的方法。

Log4FoxBaseFormatter 包括一些典型的日志记录选项设置属性，从图类图 **1** 中可以看到它们，同时它还有它自己的设置成员，因此它具有根据你选择的不管什么设置方案来设置这些选项的能力。不过，这些选项都不是 **Log4Fox API** 所要求的。

为了向你演示一个布局类可以是多么的简单，我已经包含了一个没有被该类库中任何一个类用到的派生类：**log4FoxStripHTMLFormatter**。这个类使用了一个正则表达式对象来在将消息发送给它的目标前从日志消息中去掉任何 **HTML** 标记。格式化对象不但不给接收到的消息增加修饰符、反而会从消息中去掉一些——这一点可能会令人觉得反常。我有一个应用程序，它以发送 **HTML** 格式的消息作为其主要活动之一。**HTML** 消息的主体部分是在一个很长的进程中的各个时间点上慢慢积累而成的。为了将这些消息作为一个经过审核的跟踪日志存档，我们决定增加一个日志事件，当应用程序准备好 **Email** 之后，这个事件将每一个消息主体完整的发送给一个 **SQL** 数据库。在数据库中，我们只存储这个消息的内容，不包括它的格式。

Log4Fox 的类层次

你可以从类浏览器中看到（如图 2），构成 **Log4Fox** 的所有三个参与者都有一个公共的父类：**VFPHelperObject**。前面我没有提到它是因为：这个类中的技术与跟踪日志没有关系。

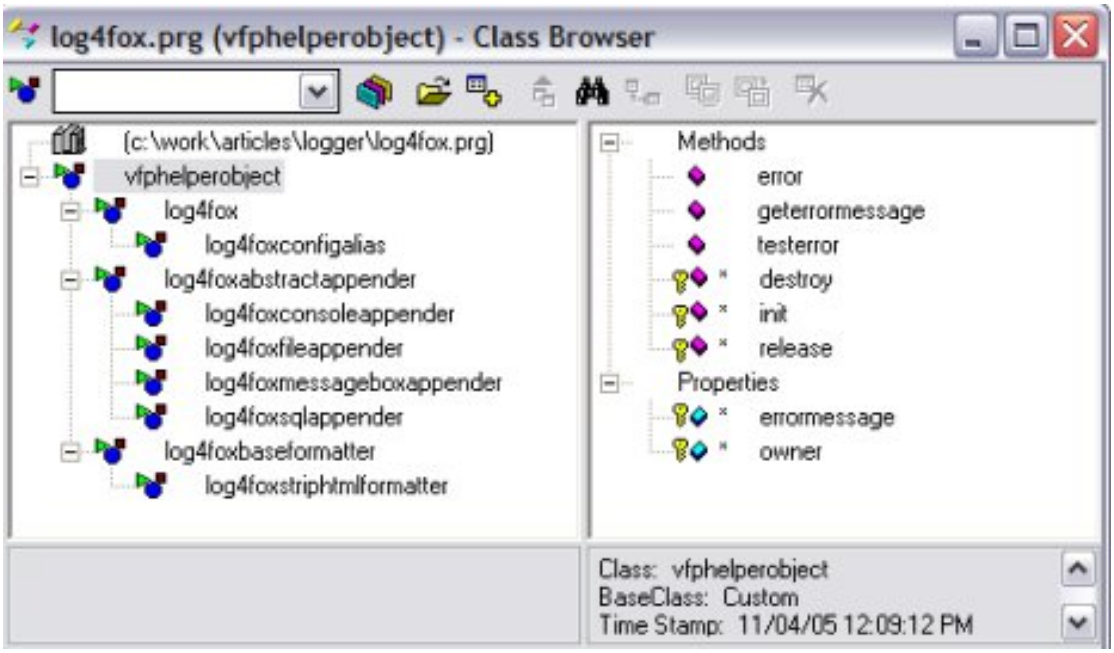

图 2、在 VFP 9 类浏览器中的 Log4Fox 类层次

从 **VFPHelperObject** 派生的类期望有一个对一个 **owner** 类的引用，这是在它们的 **Init** 方法中作为参数接收到的、并会对接收到的这个参数进行校验。如果它们对 **owner** 的类型不满意、或者没有接收到任何对 **owner** 的引用，就会拒绝（自己）被实例化。在它们的生命周期内，它们通常把错误处理授权给“这个 **owner** 对象”而不是“在应用程序中它们自己的错误处理策略”来处理。

VFPHelperObject 允许在 **Init** 中接收一个“表示你想要忽略对 **owner** 对象的需求”的标记，在源代码中的 **Test_Log4Fox.PRG** 程序就为它实例化的主 **Log4Fox** 对象使用了这个功能。为了方便，你也可以在调试 **appender** 和 布局对象的时候使用这个标记。不过对于产品来说，你的主 **Log4Fox** 对象的引用将由一个应用程序对象来掌握。你也许能理解为什么这种办法对于一个象 **Log4Fox** 系列这样的紧密协同的类组来说是非常有用的，而且不会对你的应用程序对象构成任何的限制。如果你不能理解，可以试一下把那三个最初的类改成由 **Custom** 而不是 **VFPHelperObject** 派生出来。

从图 **2** 中你可以看到，**Log4Fox** 提供了四个 **appender** 的具体实现，因为这个类类型正是这个系统的真正核心。每一个 **appender** 各管理一种不同目标设备的输出：

- Log4FoxConsoleAppender 使用 DEBUGOUT;

- Log4FoxMessageBoxAppender 使用 MESSAGEBOX 并带有一个 Timeout 选项；
- Log4FoxFileAppender 使用 TEXTMERGE 来输出到一个文件；
- Log4FoxSQLAppender 利用一个你提供的句柄来使用 SQL passthrough 去写入到一个指定的表格式（你可以在源代码中找到这个表的结构）。

每个类根据它的目标类型有着自己相应的考虑和选项。例如，文件和 **SQL** 实现有着“添加/覆盖”的能力，但这个功能对于 **Debug** 窗口或者 **MESSAGEBOX** 命令来说是没有意义的。每个 **appender** 还必须考虑它如何、或者是否要实现 **OpenLog** 和 **CloseLog** 方法。

唯一一个每个 **appender** 都必须实现的方法——如果该 **appender** 想要有任何输出的话——是它的 **protected** 方法 **SendLog**。当这个 **appender** 的公共的 **DoLog** 方法已经检查了日志级别的值，并判定应该发生输出的时候，它就把它的布局成员应用给你发送过来的无论什么信息。**SendLog** 会接收这个经过了格式化的字符串，并将之以无论什么适当的方式写入到该 **appender** 的目标设备。

每个 **SendLog** 方法都非常的简单；那个 **SQL** 实现会生成一个字符串，这个字符串会在日志被关闭时被发送到数据库，文件 **appender** 会发出一个文本合并行，如此等等。每一个 **appender** 都会做它被设置好要做的任何事情；在图 **3** 中，你可以看到同时运行的控制台（**console**）和对话框（**Messagebox**）**appender**，以及它们的布局对象的 **IncludeLogEvent** 标记设置的不同。

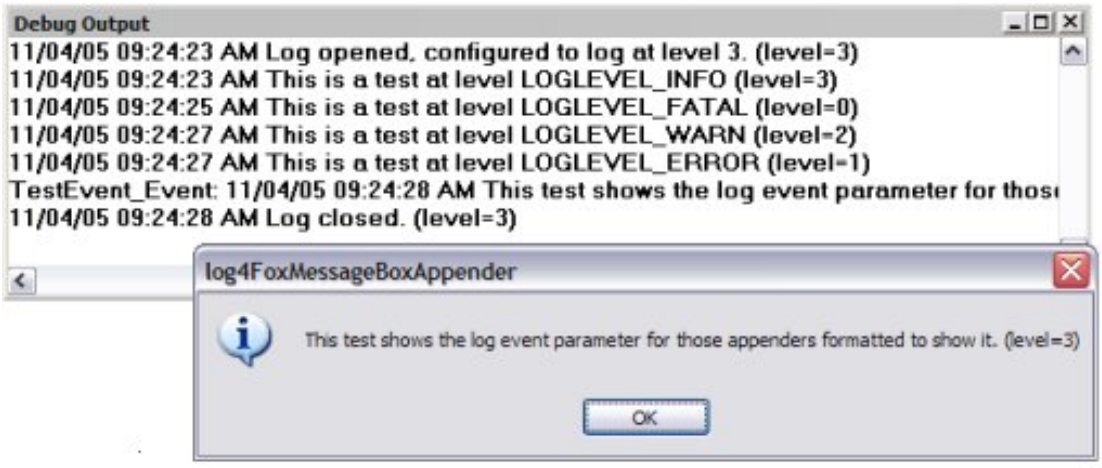

图 3、每个控制台和对话框 **appender** 各自都带有一个不同的样式和设置来处理同样的应用程序事件

你如何使用 Log4Fox

图 **3** 中的输出来自 **Test_Log4Fox.PRG**，这个程序向你演示 **Log4Fox** 的用法。在

这个案例里的主对象是 **Log4FoxConfigAlias**。这个测试程序首先用一些对 **Log4FoxConfigAlias** 应该为它的集合添加的 **appender** 的指令动态建立一个游标，然后建立 **Log4Fox owner** 对象的实例，并把游标的别名（**"x"**） 传递给该对象的 **Init**：

```
* 这个测试不需要一个 owner 对象
loLogger = CREATEOBJECT("Log4FoxConfigAlias", NULL,.T.,"x")

* 本来这个对象是又一个 owner 的:
* loLogger = CREATEOBJECT("Log4FoxConfigAlias", THIS.App,.F.,"x")
```

如果你给了测试程序一个句柄，它就会添加一个 **SQL appender**（测试程序会检查你的数据库中是否存在着一个适用的表）。由于 **SQL appender** 要求有一个有效句柄，所以你需要显式的建立这个 **appender** 或者在它的设置数据里面提供一些关于句柄的信息，后者我在这个示例里就忽略了：

```
IF NOT EMPTY(iHandle)
    CREATEOBJECT("Log4FoxSQLAppender", loLogger,.F.,iHandle)
    ? "# appenders after adding SQL appender: ", ;
        TRANSFORM(loLogger.Appenders.COUNT)
ENDIF
```

注意你不需要使用 **Log4Fox** 的 **.Appenders** 集合的 **Add** 方法。在校验了它们的 **owner** 对象引用之后，这些 **appenders** 会自动把它们自己添加到集合中去，并向集合提供一个唯一的键。它们这么做是为了确保能够正确的 **Null** 掉自己（在集合中）、以及对 **Owner** 的引用，不管它们在集合中是如何、或者按什么方式排序的，这些成对的对象都会被释放掉。

一旦 **appender** 们就位了以后，用一个简单的命令就可以开始记录日志，并可以在任何需要的地方指引你的代码进行记录日志调用：

```
loLogger.Open()
loLogger.Log(;
    "This is a test at level LOGLEVEL_ERROR", ;
    LOGLEVEL_ERROR)
loLogger.Log(;
    "This test shows the log event param for " + ;
    "those appenders formatted to show it.",;
    LOGLEVEL_INFO,"TestEvent")
loLogger.Close()
```

你可以在一个应用程序运行中的任何时候打开和关闭日志。所有的 **appender** 都能够察觉到日志记录当前是打开还是关闭的，因此即使日志当前不可用的时候，你调用记录日志方法也不会有什么不良的后果。

这个测试程序为用到的不同 **appender** 设置了各种日志记录级别，所以你会看到集合中不同的设置成员会产生不同数量的日志输出行（见图 **4**）。

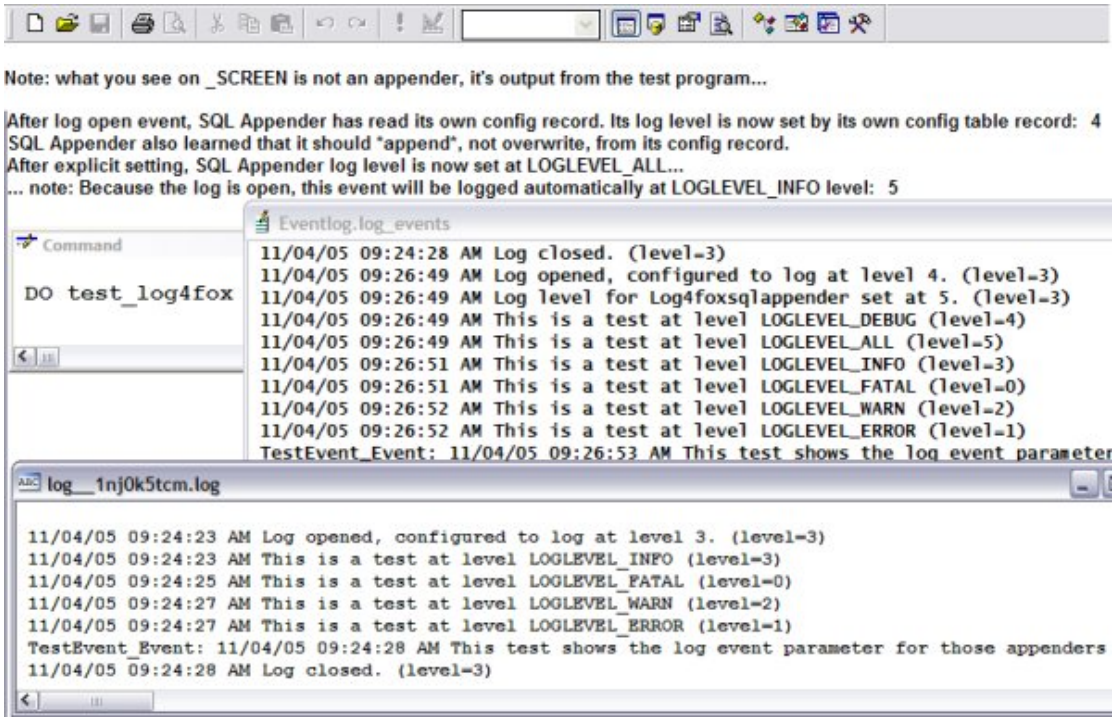

图 **4**、每个 **SQL** 和文件 **appender** 都有着各自不同的日记记录级别，不会处理所有同样的事件

你将使用 Log4Fox 吗？

即使在这个简单的版本中，**Log4Fox** 还是有一些我没有谈到的开关和功能。**Log4j** 有着更多我甚至无法想像的选项和设置的方法。但是有些人仍然想要更多的选择。**log4j** 是一个 **Apache** 软件基金（**http://apache.org**）的项目，而 **Apache** 提供的不只是 **log4j** 而已，而是一个日志记录服务 **API**，它就像是 **log4j** 穿上的一个马甲。

那是一个伟大、巨大、精彩的世界。我不能确信 **Log4Fox** 已经是世上最好的用 **Visual FoxPro** 仿造的 **log4j** 了。我确信的是：你、以及所有需要精密的跟踪工具包的开发人员们、还有这个实现所揭示的东西，将会给你们一些自己去深入的想法。

下载：**512NICHOLLS.ZIP**

来自德国的好消息

作者：Rainer Becker

译者：fbilo

让我来自我介绍：我的名字是 **Rainer Becker**，是从 **2005** 年 **12** 期开始的 **FoxTalk 2.0** 的新编辑。在进入主体之前，先让我向 **David Stevenson** 为他在过去一年半中作为编辑的伟大工作、为他在十一月刊上在线发布的精彩社论、以及在交接中的协作表示感谢。

还要感谢的是所有已经写了大量精彩文章的 **FoxTalk 2.0** 作者们。有这么一个久经考验的上佳团队，我的工作已经没多少挑战性了！在本期中，**Doug Hennig** 写下了他的第 **100** 篇文章，**11** 月份 **Andy Kramek** 刚刚发表了他的第 **90** 篇文章，而 **Marcia Akins** 则将在 **1** 月份发表她的第 **50** 篇 **"Kit Box"**专栏！有了这样一个团队，我们将延续 **FoxTalk** 的提供最高水平技术文章的长期传统。

因此为了保持这个高标准，我将竭尽全力——但不会自己写太多的文章。我最大的问题是我的英语不够好；这对我是一个真正的挑战，尤其是作为一个编辑来说。但幸运的是我是德语 **FoxPro** 用户组织的领导者，该组织将为我提供翻译和校对的资源。

此外，我是一个微软 **MVP**（最有价值专家）、德国 **FoxPro** 季刊的发行者、德国 **Visual FoxPro** 会议的组织者。因此，我非常了解所有最好的 **VFP** 作者和演讲者，同时我还有一个广泛的欧洲作者网络，他们将为“你已经习惯于从 **Foxtalk 2.0** 中读到其文章”的当前作者队伍提供补充。我们还制作了 **VFP 8.0** 和 **9.0** 的德文、法文、捷克本地化包。此外，我们庞大的 **FoxPro** 文档证明了我和我的队伍在过去的世纪里已经堆积和管理了怎么一个资料库。我说这么多，不是为了用 **VFP** 在德语国家生气勃勃的现状来吸引你，而是为了让你更清楚的了解我的背景。

你也许会奇怪现在你碰到的是怎么样的一个编辑——这家伙竟然既不写文章、也不做校对！让我来用一套设计模式把我的工作定义解释一下：我就是一个“为特定的工作选择最适合做它的人”的工厂、分配人员并为他们建立一座通讯的桥梁、定义团队中的责任链条、观察所有任务的状态、并在所有相关的人员之间作为一个中介者而存在。让我们把这个结构就叫做一个“**FoxTalk** 编辑”吧！之前我作为 **FoxPro** 用户组德语杂志的领导者和编辑已经成功的运用了这种结构，该组织的运作流程基本上跟 **FoxTalk 2.0** 是一样的。

我还有一些好消息要报告一下：在此之前数个月内，德国用户组已经出售了比我们出售的 **Visual FoxPro 8.0** 的总数还要多的 **Visual FoxPro 9.0** 的拷贝。**Ken Levy** 最近

在 11 月份的德国 DEVCON 的讲话上已经证实了这一点：目前，德国历史上第一次占据了 Visual FoxPro 销售量第二的位置。此外，Doug Hennig 报告说 Stonefield Query 的销售在此次 Devcon 会议上相当的活跃。我还要很高兴的向大家报告我们的 Visual Extend framework 的销量在过去的一年中增长了百分之二百五十八，而且我们百分之六十的收入是来自于新客户的。此外，虽然这一年 VFP 没有新版本的发布，而且其它的会议也苦于业绩平平，但本年度德国 DevCon 的参加者还是保持了去年的水平。这是个好消息！尤其是在考虑到德国经济持续疲软的情况下。

如你所见，我这个人对 VFP 9 的技术功能相当的热心，但同时也对销售的成功、以及来自微软自己的公开支持声明充满热情。VFP 开发组做了一件很了不起的工作，感谢他们！

我将继续为 FoxPro 社区工作，我期望当前版本的良好销售势头也能继续下去、并且也乐于看到将来代号 Sedna 的 Visual FoxPro 9.0 新版本也能重复这种彻底的成功。你将能从 Andrew MacNeill 的 “The FoxShow”在线广播的一个部分中听到更多关于转变方面的以及其它好消息。Andrew 对我的采访可以从 www.thefoxshow.com 在线访问到，而抄本稍后将作为 FoxTalk 2.0 的一份仅在线阅读文章提供。

在今天的的 FoxPro 世界里有着丰富的有趣话题。我将尽我所能在每一期 FoxTalk 2.0 的杂志中提高读者的技术水平。

基于角色的安全性（一）

作者：Doug Hennig

译者：CY

基于角色的安全可以让你指定那个用户对特殊的安全实体有权限，是基于组，或角色，级别等。本月，Doug Hennig 开始讨论如何在应用程序里实现基于角色的安全。

作为我的第 **100** 篇文章（是啊，就是这么久了！），我打算开始一个多章节的关于最近我想加入到我的应用程序的系列：角色安全。

在 **2004** 年 **9** 月到 **11** 月出版的 **FoxTalk 2.0**，**Andy Kramek** 和 **Marcia Akins** 详细讨论过这个主题。因此为什么要重提？虽然他们的设计是很伟大的，并且我也从中学了到许多，我需要一个不同的方法。详细的说：

我需要用户可以是多个角色。程序管理员并不需要是高级用户，但是可以访问表单的人的权限却可以是不同的角色。虽然这是不太可能的，但可能的是部门的部长却可能是程序管理员。然而这并不意味她有权限查看工资表的薪资数据。因此，部长也可能是个“管理人员”角色，允许访问程序的一部分而拒绝访问其他部分。

Andy 和 **Marcia** 是通过排除法来达到安全的，在里面一个用户是假定有足够权限来访问的，除非其他指定的。因为多种原因，我需要的相反：一个用户没有权限来访问，除非指定分配权限给它。

他们的设计实现了字段级的安全，这正是我所需要的，却是用不同的方法。对比于在表单上隐藏或禁止绑定于字段的控件安全，我的应用程序需要知道用户可以在报表里看到哪些字段（如果用户不能看到报表里的任何字段，那么也将看不到整个报表）。

最后，我想展示所有维护安全的方法，比如增加用户，更改用户角色，等等，作为一个安全管理类的方法，这些任务可以是程序化的，比如通过一个 **COM** 对象，如果有必要。

虽然我宁可用已经有的代码而不愿意重新写，我所需要的却是与 **Andy** 和 **Marcia** 的设计大大不同，因此我不得不开始草拟（当然，正如我先前提到的，我从他们的设计里获得了许多灵感）。

我想要改变你对角色的看法。**Andy** 和 **Marcia** 有一个非常清楚的定义：角色就是一种可以被应用于一人或多人的应用程序的指定访问模式。它并不需要与指定的作业描述或用

户相关。因此，带着这个想法，让我们来看看我的实现方法。

数据模型

正如你在图 1 所见，这里有五个表与我的应用程序实现角色安全有关：

- USERS—包含系统里每个用户的信息，比如他们的用户名和密码（当然是加密过的）。
- ROLES—包含系统里定义的角色信息。
- USERROLES—提供一个用户和角色的连接表，以解决他们的多对多关系。
- ELEMENTS—包含安全对象的信息。我把安全对象命名为“元素”，因为有许多不同的安全对象：表单、字段、报表、菜单功能，等等。这个表可以被用于管理员对话框里，指定权限于特定元素以分配到指定角色。
- SECURITY—定义每个角色对每个指定元素的权限。

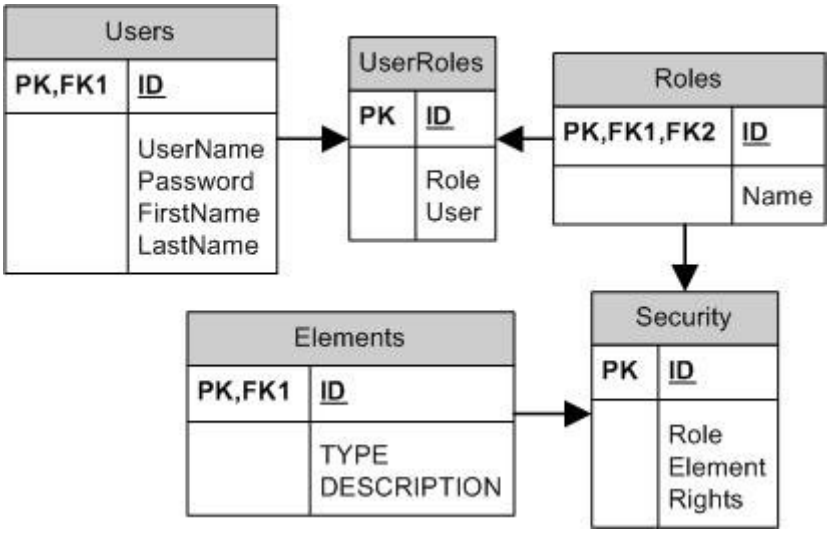

图 1：角色安全的数据模型

用户和角色集合

虽然我们可以通过在表内的记录访问用户和角色，它被作为集合的成员来代替，特别是我们想要在 COM 对象里使用它时。**SFUserCollection** 类是一个用户对象的集合，包含每个指定用户的属性。类似的，**SFRoleCollection** 是一个角色对象的集合，包含角色的属性。这两个类都包含在 **SFSecurity.VCX** 里。

在我们查看这些类前，让我们来看看它们的基类，**SFCollectionTable**。**SFCollectionTable**，定义于 **SFCtrls.VCX**，是 **SFCollectionOneClass** 的子类，也是 **SFCollection** 的子类，也是 **Collection** 的子类。**SFCollectionTable** 是特别设计作为表内记录的接口。集合的成员是对象，在表内的一个记录，其属性映射于表内的列。

FillCollection 方法是用于从它所分配的表里装载集合。**FillCollection** 调用

OpenTable（我们没有看到）以打开在 **cTable** 属性所指定的表，并把它的别名放入 **cAlias**。然后 **FillCollection** 遍历表，对每个记录调用 **CreateEntityObject** 方法，以创建一个对象，其属性映射于记录的字段。**CreateEntityObject** 在这个类里是抽象的，但是可以很简单的如 **RETURN SCATTER loObject**，或是复杂点就如实例化一个对象并设置它的属性为当前记录的字段值。然后 **FillCollection** 调用 **GetEntityName**，它在这个类里也是抽象的，以取得我们想要加入到集合的对象所使用的名，并且最后调用 **Add** 方法以存储这个对象。

```
local lnSelect, ;
    loEntity, ;
    lcName, ;
    llReturn

if This.OpenTable()
    lnSelect = select()
    select (This.cAlias)
    scan
        loEntity = This.CreateEntityObject()
        lcName = This.GetEntityName(loEntity)
        This.Add(loEntity, lcName)
    endscan
    select (lnSelect)
    llReturn = .T.
endif This.OpenTable()
return llReturn
```

Add 方法是在 **SFCollectionTable** 里重载的。**Add** 方法可以以两种方式来调用：通过传递一个要存储的对象和它作为关键值的名字，也就是在 **FillCollection** 里被调用的，或者通过传递这个名，也就是它通常在客户端代码里所被调用的。在后一种情况里，我们只是加入一个对象到集合里，只是增加一个新记录到表里。

```
lparameters tuItem, ;
    tcKey
local loEntity

do case
    * If we were passed a name, add a record to the table
    * and use the default behavior to add it to the
    * collection. Note that we re-read loEntity from the
    * collection in case we were specified a duplicate
    * name and it isn't added to the collection.
    case vartype(tuItem) = 'C'
        loEntity = This.AddRecord(tuItem)
        loEntity = dodefault(loEntity, tuItem)
    * We were passed an object, so add it to the
```

```
        * collection. As above, we re-read the object from
        * the collection in case it's a duplicate.
        case vartype(tuItem) = 'O'
            loEntity = tuItem
            loEntity = dodefault(loEntity, tcKey)
        * Invalid parameters.
        otherwise
            loEntity = .NULL.
            throw 'Function argument value, type, or ' + ;
                'count is invalid.'
    endcase
    nodefault
    return loEntity
```

其他方法也是在 **SFCollectionTable** 里被重载或增加的，包括 **Remove** 和 **RemoveRecord**，那就是我们所看到的。

SFUserCollection 是 **SFCollectionTable** 的子类。它的 **cTable** 属性默认设置为 **USER.DBF**，但是当然也可以被改变为指向不同的用户表。

因为我不需要用户对象有任何行为，这个类的 **CreateEntityRecord** 方法使用 **SCATTER NAME** 来创建一个其属性匹配于字段的对象。注意到我没有对密码解密，它是加密存储在表里。那样做的问题是密码在内存里就可能是明文，它就有可能被黑客找出密码。我将在下月讨论密码解密。

```
    local lnSelect, ;
    loUser

    lnSelect = select()
    select (This.cAlias)
    scatter name loUser

    with loUser
        .UserName = trim(.UserName)
        .Password = trim(.Password)
        .FirstName = trim(.FirstName)
        .LastName = trim(.LastName)
    Endwith

    select (lnSelect)
    return loUser
```

AddRecord 的方法，是从父类的 **Add** 方法里调用以增加一个新记录到表，检查是否已经存在指定的名，如果没有，就增加它。然后它调用 **CreateEntityObject** 来从记录创建一个用户对象。

```
lparameters tcName
local loUser

if not seek(upper(padr(tcName, ;
    len(__USERS.USERNAME))), '__USERS', 'USERNAME')
    insert into __USERS (USERNAME) values (tcName)
endif not seek(upper(padr(tcName ...

loUser = This.CreateEntityObject()
return loUser
```

SFUserCollection 有一个新的方法，**GetUserByID**。因为在集合里用户对象的关键值是用户名，通过 **ID** 来查找用户是很困难的。为使得它更容易，**GetUserByID** 仅简单地在表里 **SEEK** 查找 **ID**，如果找到，就调用 **CreateEntityObject** 为用户记录创建一个用户对象。

```
lparameters tiUserID
local loUser

if seek(tiUserID, '__USERS', 'ID')
    loUser = This.CreateEntityObject()
else
    loUser = .NULL.
endif seek(tiUserID, '__USERS', 'ID')
return loUser
```

我们已经明白新记录是如何加入到用户表的，但是用户属性的改变是如何保存的，比如当用户密码改变时？**SFCollection** 有一个 **SFCollection** 方法，当集合被释放时它被调用，当然它也可以被人工调用以根据需求来保存集合。它遍历整个集合并为每个成员调用 **SaveItem**。**SaveItem** 在 **SFCollection** 里是抽象的，但是 **SFUserCollection** 的实现是利用 **GATHER NAME** 来更新用户表中相应的记录为当前对象的属性值。

我们并没有看 **SFRoleCollection**，是因为它非常类似于 **SFUserCollection**。就象 **SFUserCollection**，它有一个 **GetRoleByID** 方法以返回指定 **ID** 的角色对象。

安全管理器

现在该是来看看安全管理器类。我并没有篇幅在文章里涵盖整个类，我将在下月继续讨论。

SFSecurity，是在 **SFSecurity.VCX** 里，是 **SFCustom** 的子类，后者是定义在

SFCtrls.VCX 里的 **Custom** 的基类。它的 **Init** 方法实例化了一个用户和角色集合到 **oUsers** 和 **oRoles** 属性里。注意到并不是直接的指定 **SFUserCollection** 和 **SFRoleCollection**，这个代码从属性里获取了类的名和类库。这是我灵活的设计，因为我可以子类化 **SFSecurity**，并指定不同的类，如果我有需要。**Init** 方法也打开了 **VFPEncryption.FLL**，一个由 **VFP** 社区的新 **MVP Craig Boyd** 所提供的加密类库，并且可以从他的网站下载（**www.sweetpotatosoftware.com/SPSBlog**）。我将下月更详细的讨论 **VFPEncryption.FLL**。

```
with This
    * Instantiate user and role collection objects.
    .oUsers = newobject(.cUserCollectionClass,    .cUserCollectionLibrary)
    .oRoles = newobject(.cRoleCollectionClass,    .cRoleCollectionLibrary)

    * Open the VFP encryption library.
    set library to VFPEncryption.FLL additive
endwith
```

Setup 方法打开了安全相关表，指令用户和角色集合从相应的记录来填充自己，并且设置 **lSetup** 属性以指明设置已经完成。它也实例化了一个本地化对象。我不在本文里对这个对象展开更多，我把它留在将来的文章里。然而，这个对象是用于本地化目的，它的主要方法，**GetLocalizedString**，在资源表里查找一个字符串，并返回指定语言的等效字符串。这允许你在代码里避免使用指定语言的字符串，因此你可以很容易把它本地化为其他语言。注意到 **Setup** 方法可以在实例化对象后被人工调用，但是它也可以从许多方法里被自动调用。

```
with This
    llReturn = .OpenTables()
    llReturn = llReturn and .oUsers.FillCollection()
    llReturn = llReturn and .oRoles.FillCollection()

    if llReturn and vartype(.oLocalizer) <> 'O'
        .oLocalizer = newobject('SFLocalize',    'SFLocalize.VCX')
    endif llReturn ...
    .lSetup = llReturn
Endwith

return llReturn
```

如果指定的用户是属于指定的角色，**IsUserInRole** 方法将返回为**.T.**。注意到用户和角色都是通过 **ID** 来指定而不是其名字，在类里的大多数方法都是这样的。同样也注意到如果用户 **ID** 没有指定，就会使用当前登录的用户 **ID**。我们将在下月看到用户是如何登录的，但是现在，一旦用户登录，用户对象的指针就被存储在 **oCurrentUser** 属性里。

```
lparameters tiUser, ;
    tiRole
    local liUser, ;
    llReturn, ;
    lcMessage

with This
    * If the user ID wasn't specified, use the one for
    * the logged-in user.
    if (vartype(tiUser) <> 'N' or tiUser = 0) and ;
          vartype(.oCurrentUser) = 'O'
          liUser = .oCurrentUser.ID
    else
          liUser = tiUser
    endif (vartype(tiUser) <> 'N' ...

    do case
          * Ensure we've been set up properly.
          case not .lSetup and not .Setup()
                llReturn = .F.
                lcMessage = .GetLocalizedString(.cErrorMessage)
          * Ensure the parameters are valid
          case not .CheckUser(liUser)
                llReturn = .F.
                lcMessage = ;
                      .GetLocalizedString('ERR_INVALID_USER_ID')
          case not .CheckRole(tiRole)
                llReturn = .F.
                lcMessage = ;
                      .GetLocalizedString('ERR_INVALID_ROLE_ID')
          * See if the specified user is in the specified role.
          otherwise
                llReturn = seek(str(tiRole) + str(liUser), ;
                      '__USERROLES', 'ROLEUSER')
          endcase
endwith

* Raise an error if we had a problem (do this outside
* the WITH structure to avoid problems with unclosed
* WITHs).
if not empty(lcMessage)
    throw lcMessage
endif not empty(lcMessage)
```

```
return llReturn
```

AddUserToRole 和 **RemoveUserFromRole** 非常简单：它们调用 **IsUserInRole** 来判断指定的用户是否属于指定的角色，并且相应的更新 **USERROLES** 表。这里是 **AddUserToRole** 的代码：

```
lparameters tiUser, ;
    tiRole

if not This.IsUserInRole(tiUser, tiRole)
    insert into __USERROLES (ROLE, USER) values (tiRole, tiUser)
endif not This.IsUserInRole(tiUser, tiRole)
```

GetRolesForUser 返回用户所属的角色集合。注意到你可以指明是否角色 **ID** 被用作为集合里项目的关键值（通常是使用名字），通过传递**.T.**作为第二个参数。

```
lparameters tiUser, ;
    tlIDAsKey

local loRoles, ;
    liUser, ;
    lcMessage, ;
    lnSelect, ;
    loRole

with This
    * Create a collection for the roles.
    loRoles = newobject('SFCollection', 'SFCtrls.VCX')

    * If the user ID wasn't specified, use the one for
    * the logged-in user.
    if (vartype(tiUser) <> 'N' or tiUser = 0) and ;
        vartype(.oCurrentUser) = 'O'
        liUser = .oCurrentUser.ID
    else
        liUser = tiUser
    endif (vartype(tiUser) <> 'N' ...

    do case
        * Ensure we've been set up properly.
        case not .lSetup and not .Setup()
            lcMessage = .GetLocalizedString(.cErrorMessage)
        * Ensure the parameters are valid.
        case not .CheckUser(liUser)
            lcMessage = ;
```

```
                    .GetLocalizedString('ERR_INVALID_USER_ID')
            * Get all records from the user roles table that have
            * the specified user, get the appropriate role
            * objects from the roles collection, and add them to
            * the collection we'll return.
            otherwise
                lnSelect = select()
                select __USERROLES
                scan for USER = liUser
                    loRole = .oRoles.Item(ROLE)
                    do case
                        case vartype(loRole) <> 'O'
                        case tlIDAsKey
                            loRoles.Add(loRole, transform(ROLE))
                        otherwise
                            loRoles.Add(loRole, loRole.Name)
                    endcase
                endscan for USER = liUser

                select (lnSelect)
            endcase
        endwith

        * Raise an error if we had a problem (do this
        * outside the WITH structure to avoid problems
        * with unclosed WITHs).
        if not empty(lcMessage)
            throw lcMessage
        endif not empty(lcMessage)

        return loRoles
```

检验

让我们来检验下我们所涵盖的代码。下列来自于 **TestSecurity.PRG**，创建了一对角色和用户，并增加那些用户到相应的角色。

```
loSecurity = newobject('SFSecurity', 'SFSecurity.vcx')

if not loSecurity.Setup()
    messagebox(loSecurity.cErrorMessage)
    return
endif not loSecurity.Setup()
```

```
with loSecurity
    * Create the roles we'll need.
    .oRoles.Add('Administrators')
    .oRoles.Add('Everyone')

    * Create the ADMIN user.
    loUser = .oUsers.Add('ADMIN')
    loUser.Password = 'Testing123'
    loUser.FirstName = 'Administrative'
    loUser.LastName = 'User'

    * Create the DHENNIG user.
    loUser = .oUsers.Add('DHENNIG')
    loUser.Password = 'DumbPassword'
    loUser.FirstName = 'Doug'
    loUser.LastName = 'Hennig'

    * Add the ADMIN user to the various roles.
    loUser = .oUsers.Item('ADMIN')
    loRole = .oRoles.Item('Administrators')
    .AddUserToRole(loUser.ID, loRole.ID)
    loRole = .oRoles.Item('Everyone')
    .AddUserToRole(loUser.ID, loRole.ID)

    * Add the DHENNIG user to the appropriate roles.
    loUser = .oUsers.Item('DHENNIG')
    loRole = .oRoles.Item('Everyone')
    .AddUserToRole(loUser.ID, loRole.ID)

    * See which roles are available for DHENNIG, then add
    * and remove the Administrators role.
    loUser = .oUsers.Item('DHENNIG')
    loAdminRole = .oRoles.Item('Administrators')
    messagebox('DHENNIG is ' + ;
        iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
        '', 'not ') + 'in the Administrators role.')
    .AddUserToRole(loUser.ID, loAdminRole.ID)

    messagebox('DHENNIG is ' + ;
        iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
        '', 'not ') + 'in the Administrators role.')
    loRoles = .GetRolesForUser(loUser.ID)
    lcRoles = ''
```

```
        for each loRole in loRoles
            lcRoles = lcRoles + ;
                iif(empty(lcRoles), '', ', ') + loRole.Name
        next loRole

        messagebox('DHENNIG is in the following roles:' + ;
            chr(13) + chr(13) + lcRoles)
        .RemoveUserFromRole(loUser.ID, loAdminRole.ID)
        messagebox('DHENNIG is ' + ;
            iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
            '', 'not ') + 'in the Administrators role.')
    endwith
```

总结

在这个多章节的角色安全文章的第一部分，我们看到了用户和角色集合，和处理用户和角色管理的安全管理器类的一部分。下月，我们将继续我们关于 **SFSecurity** 的讨论，看看管理一个有不同元素角色的方法，用户登录和登出，以及加密和解密密码。

下载：**512HENNIG.ZIP**

VFP9 的 REPORTBEHAVIOR 90

——在你客户的机器上应用新的报表功能

作者：Uwe Habermann

译者：CY

在开发机上，SET REPORTBEHAVIOR 90 设置后，一切都运行良好。但是在客户的机器上，执行 REPORT FORM 命令会产生一个运行时错误 12："变量 _REPORTOUTPUT 未找到。"在本文中，Uwe Habermann 将会解释如何导致这个错误的，并提出几个解决办法。

VFP9 最博大的新特性是扩展的报表设计器。在本文中，我不讨论 **VFP9** 的这些新特性和报表引擎的多种可用性以及报表设计器。取而代之的是，我将仅讲述把新特性集成到最终用户的应用程序里。

你可能已经注意到你将不得不在命令窗口设置 **SET REPORTBEHAVIOR 90** 才能切换到报表设计器的新特性。默认值是 **SET REPORTBEHAVIOR 80**，是禁止新特性的。为什么这样？在我们开始查找答案前，让我们尝试和测试新的特性。

你可以在开发环境里加入命令 **SET REPORTBEHAVIOR 90**，那么报表设计器和报表引擎就都会有新特性。真的很好。如果你在应用程序的启动程序里加入 **SET REPORTBEHAVIOR 90** 命令，你将会看到启动程序执行后，新的报表特性将会在你新的应用程序里运行起来。然后你可以创建一个执行文件并从 **Windows** 资源管理器里启动这个 **EXE** 文件。一切如你所愿。

现在你以通常的方法把你的程序安装到一个客户的 **PC**，它没有安装 **VFP9** 开发环境。就会出现奇怪的东西了。新的报表特性将不能工作。如果命令 **REPORT FORM** 被执行到，就会产生一个运行时错误（见图 **1**）。客户的机器与开发机器出现了不同的东西，然而软件其实是做同样的事情。实际上不同的是在客户的机器上缺少报表程序。

图 1：为什么客户机器上会出现这个错误？

报表程序

对于报表程序，我认为是下列文件：

- ReportOutput.app
- ReportPreview.app
- ReportBuilder.app

这些报表文件是新报表引擎和报表设计器的组件，相应类似于向导，它们是用 VFP 编写的。一旦设置了 SET REPORTBEHAVIOR 90，或者 REPORT 命令使用了 OBJECT TYPE 子句，你就需要报表程序。

在你的开发机器上，你可以在两个目录下找到报表程序。第一，这些文件是放在 VFP9 安装目录下。第二个，你可以在包含运行时文件的相同目录下找到，通常是在 C:\Program Files\CommonFiles\Microsoft Shared\VFP。

那个报表程序是对应那个命令？

根据应用程序，你不需要把所有的报表程序都发布给客户。

REPORT FORM (or LABEL) ... TO PRINTER

为了在 SET REPORTBEHAVIOR 90 下执行 REPORT FORM 命令，你通常需要报表程序 ReportOutput.app。正如先前所说明的，如果缺少报表程序，就会产生运行时错误。

REPORT FORM (or LABEL) ... PREVIEW

这个命令的执行需要在客户 PC 上有报表程序 ReportOutput.app 和 ReportPreview.app。没有 ReportOutput.app 文件，将会象其他用法的 REPORT FORM 命令一样产生运行时错误。

如果缺少 ReportPreview.app 文件，不会产生运行时错误，但是报表页不会显示。程序继续执行下一行。一个意料外的行为，你需要在你的支持 FAQ 里作提示。

MODIFY REPORT

开发者想要让他们的客户有机会在应用程序里执行报表设计器来根据他们自己的需要重新修改报表，通过 **MODIFY REPORT ?** 命令。这种情况在所有版本的 **VFP** 里都有。

某些问题必须要考虑，如果你想要在 **SET REPORTBEHAVIOR 90** 下运行。当报表设计器的事件发生时，程序将启动存储在**_reportbuilder** 系统变量里的文件名和路径。

执行 **MODIFY REPORT** 命令需要程序 **ReportBuilder.app** 在客户机器上。如果没有，报表将会如同设置了 **SET REPORT BEHAVIOR 80** 一样运行。如果程序 **ReportBuilder.app** 不存在，不会产生运行时错误。

系统变量

文件 **ReportOutput.app, ReportPreview.app** 和 **ReportBuilder.app** 的完整路径名被赋值在系统变量里：

```
_reportoutput = <path> + "ReportOutput.app"
_reportpreview = <path> + "ReportPreview.app"
_reportbuilder = <path> + "ReportBuilder.app"
```

如果报表程序没有安装在客户机器上，这些系统变量被赋值为空字符串。

现在来看看运行时错误。**REPORT FORM** 命令需要 **ReportOutput.app** 来执行，因为设置了 **SET REPORTBEHAVIOR 90**。**VFP** 根据系统变量**_reportoutput** 来查找这个程序。如果没有找到这个程序，将会出现错误信息。不可否认信息“变量**_REPORTOUTPUT** 没有找到”是有些糊涂。依靠调试器你可以很容易的确定这个系统变量是存在的，它的类型为字符，并且包含一个空字符串。

当然，你可以在运行时给这些系统变量赋值，因此可以使用存储于不同路径的报表程序。给这些系统变量赋值不同的特定程序是切实可行的。

微软对此做了什么？

因此微软选择了默认值为 **SET REPORT BEHAVIOR 80**，因为为了使用这些新特性，要求的附加程序文件默认情况下是没有的。

不需要简单复制报表程序到你的客户机器上的 **EXE** 文件目录。但是它却可以复制报表程序到 **VFP9** 的运行文件目录。通常它是 **C:\Program Files\Common Files\Microsoft Shared\VFP** 目录。你的应用程序将会查找报表程序。报表程序的完整路径名会自动赋值给系统变量**_reportbuilder, _reportoutput** 和**_reportpreview**。

你可以为此在你的启动程序里执行一个很容易的任务，并包含那个合并模块 **VFP9RptApps.msm**。这个合并模块存在于 **Microsoft Visual FoxPro 9 Report Applications** 名下的 **InstallShield Express Visual FoxPro Limited Edition**。如果你包含了这个合并模块，**ReportOutput.app, ReportPreview.app** 和 **ReportBuilder.app** 文件将会安装在客户机器的 **C:\Program Files\Common Files\Microsoft Shared\VFP** 目录下。于是它们可以是自己的程序。

原则上所有都是固定的，但是你可以再次返回到这个问题，当你在先前描述的过程后在客户机器上看到了第一页时。

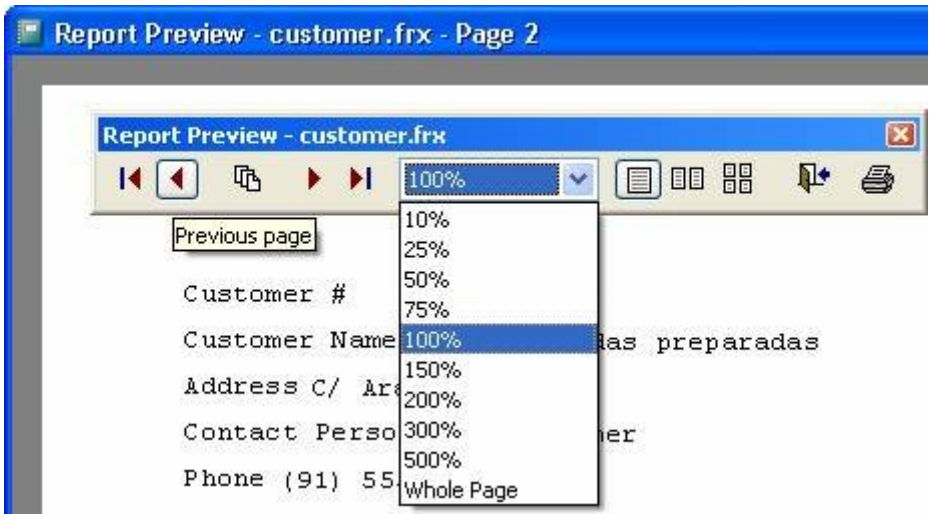

图 2：原始的微软 **ReportPreview.app** 的页面。

随同 **VFP** 发布的报表程序的安装的不利之处是，它仅只有英语的。为了向客户提供本地化文本，开发者将不得不走不同的路。

在你的应用程序里包含报表程序

区别于前面所讲的把报表程序与你自己的程序配置在一起，你可以选择把报表程序的源代码包含入你自己的项目。

VFP9 安装的 **Tools\xsource\xsource.zip** 文件包含有报表程序的源代码。你也可以在 **VFP9** 安装的 **FFC** 目录的源代码部分找到它。把文件从 **FFC** 目录再分发的限制，可见于文件 **redist.txt**。因此，从 **xsource.zip** 压缩包里使用这个文件。你可以从 **xsource.zip** 里集成所有的文件到你自己的项目，作修改，并分发给你的客户。

你需要下列目录下的项目：

- VFPSource\ReportOutput
- VFPSource\ReportPreview
- VFPSource\ReportBuilder

包含报表程序于定制项目的一个好处是，你可以翻译文本为德文或其它语言。

ReportOutput

为包含一个本地化或定制版本的 **ReportOutput** 到你的应用程序，你将不得不从目录 **VFPSource\ReportOutput** 下复制源代码到你自己的项目目录。同样，**frxoutput.prg** 文件也必须加入到你的项目里。

在启动程序的开始处或者 **config.fpw** 文件里，把现在集成的 **ReportOutput** 程序赋值给系统变量**_reportoutput**。

```
_REPORTOUTPUT = "frxoutput.prg"
```

现在当你复制了项目的所有文件，所有另外需要的文件被加入到项目里。**EXE** 文件将会因为增加了源代码增长了将近 **500KB**。

ReportPreview

在 **REPORTBEHAVIOR 90** 下的页面需要 **VFPSource\ReportPreview** 目录下的 **ReportPreview** 程序。**frxpreview.prg** 文件必须被加入到你的项目。为使得 **VFP** 可以使用现在集成的 **ReportPreview app**，这个 **PRG** 文件必须在启动程序的开始处被赋值给系统变量**_reportpreview**。

```
_REPORTPREVIEW = "frxpreview.prg"
```

你也可以在配置文件 **config.fpw** 里配置这个路径。

集成 **ReportPreview** 使得你也要集成 **ReportOutput** 到你自己的项目里。加入 **ReportOutput** 和 **ReportPreview** 将导致你产生的 **EXE** 文件将会增长大约 **650KB**。

ReportBuilder

包含一个本地化或定制版本的 **ReportBuilder** 在你自己的应用程序里并不会运行，除非你从 **VFPSource\ReportBuilder** 目录下复制了源代码文件到你自己的项目目录。并且 **frxbuilder.prg** 文件必须被加入到你的项目里。编译所有的项目文件将会自动加入所有附加的必需文件到项目里。注意到自由表 **frxbuilder.dbf**，它是以排除方式加入到项目里的。这才是正确的方法。**ReportBuilder** 对这个表的只读访问将会导致出错。

现在你必须确保包含在项目里的 **ReportBuilder.app** 实际上是要被用到的。你将不得不指派现在新集成的 **ReportBuilder.app** 的名给系统变量**_reportbuilder**，最好是在你的启动程序的开始处来执行或是在 **config.fpw** 文件里。

```
_REPORTBUILDER = "frxbuilder.prg"
```

现在你可以从你的项目里产生执行文件，并且在客户的机器上运行它。命令 **MODIFY REPORT ?**将会在 **SET REPORTBEHAVIOR 90** 设置下正确运行。注意到 **ReportBuilder app** 的源代码大小大约是 **1MB**，并且将因此增大了 **EXE** 文件。如果 **ReportBuilder.app**

文件是配置于客户，这将需要在硬盘上 **722KB** 的存储容量。

公共文件

报表程序 **ReportBuilder.app** 和 **ReportPreview.app** 一起使用下列文件：

- FrxControls.vcx/vct
- FrxCommon.prg
- Grabber.gif
- Wwrite.ico
- FoxPro_Reporting.h

每个文件在项目里只包含一份。报表程序 **ReportOutput.app** 和 **ReportPreview.app** 使用同一个包含文件，命名为 **FoxPro_Reporting.h**。

帮助

ReportBuilder.app 和它所属的项目源代码包含有 **HelpContextIDs**，它指向 **VFP** 帮助文件 **dv_FoxHelp.chm**。依照 **redist.txt**，**VFP** 帮助文件不需要分发给客户。

猜想你可能与你自己的应用程序一起分发了帮助文件。如果用到的 **HelpContextIDs** 不存在，你的帮助文件的起始页就会被显示出来。

为了显示智能的帮助文件，你可以在你的 **Help** 项目使用被 **ReportBuilder** 所使用的 **HelpContextIDs**，或者改变在 **ReportBuilder** 里的 **HelpContextIDs**。当然，你将不得不在这两种情况下编写你自己相应的帮助文本，而不是部分的 **VFP** 帮助文件被分发。你不需要为 **ReportBuilder** 编写帮助标题。然后你可以采取一个简单的技巧。帮助按钮可以从 **ReportBuilder** 里被移出。最后，在包含文件 **frxbuilder.h** 里设置 **SHOW_HELP_BUTTON_ON_HANDLER_FORMS** 常量为**.F.**，并编译项目的所有文件。

这也是把源代码包含到你自己项目里，或者生成你自己定制的报表程序的原因之一。

本地化

在报表程序里用到的所有文本是包含在包含文件里的。对于本地化，在下列文件里的文本必须被翻译：

- **ReportOutput:** **reportoutput_locs.h**
  reportlisteners_locs.h
- **ReportPreview:** **frxpreview_loc.h**

- **ReportBuilder:** **_frxcursor.h**

  frxbuilder_loc.h

在这样的情况下，在 **frxbuilder_loc.h** 和 **frxpreview_loc.h** 文件里设置 **USE_LOC_STRINS_IN_UI** 常量为**.T.**是很重要的。

```
#define USE_LOC_STRINGS_IN_UI .T.
```

只有当这个值被设置了，本地化的文本才会被使用在报表程序里。这个努力的结果是很象样的，并且你的客户将会对它很感激的（见图 **3**）。

现在你可以看到这个策略所带来的好处。为这个页面产生本地化版本的文本现在已经变得很容易了（见图 **4**）。对于更早版本的 **VFP**，这个文本由运行时文件所预先决定的。

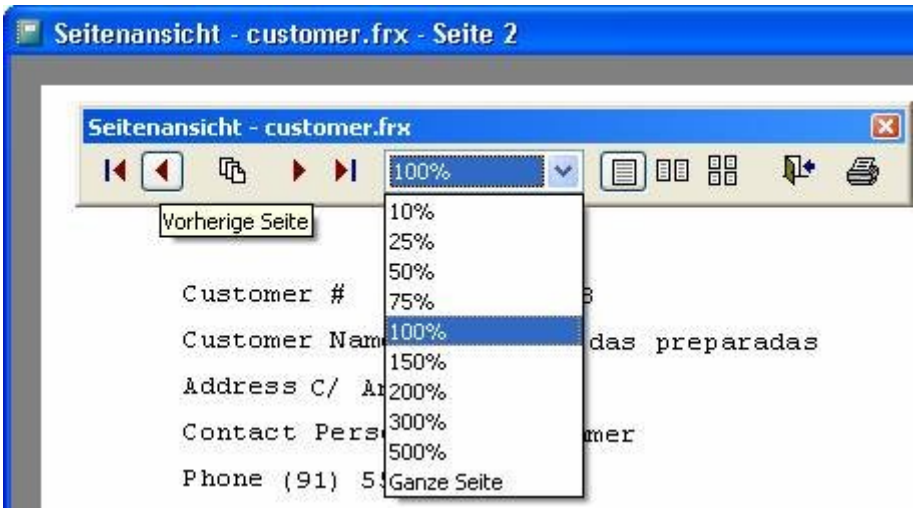

图 **3**：德语的本地化页面从 **ReportPreview.app** 里被集成到源代码里。

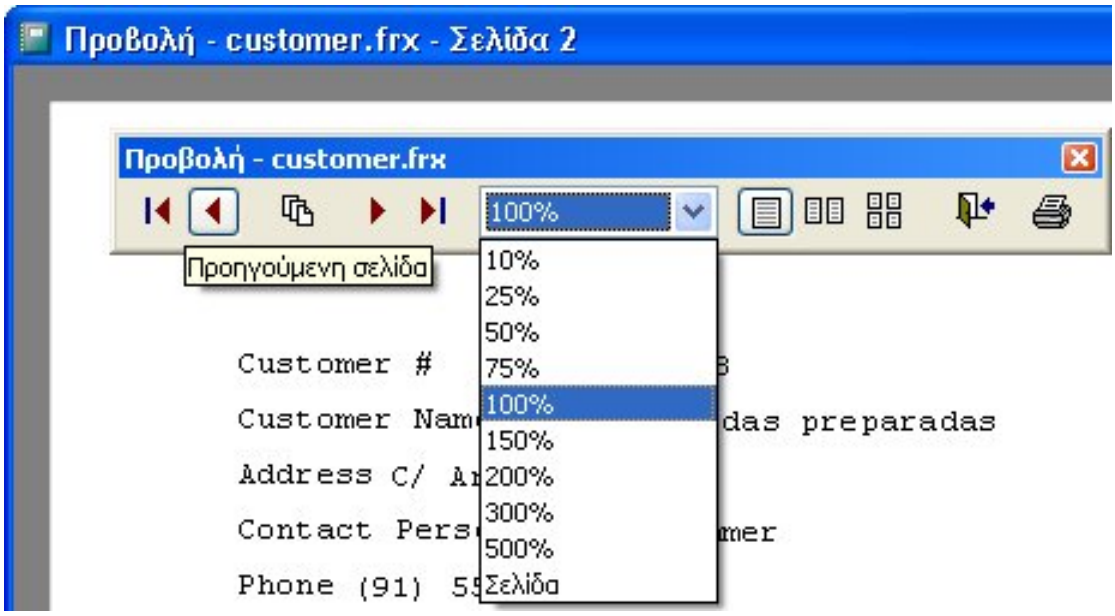

图 **4**：希腊文的本地化版本的页面。

更多

因此，对所有由新报表引擎所提供的功能调用，是否可以用于客户环境？现在还不完全，我感到遗憾。这篇文章的目的首先是让在 **SET REPORTBEHAVIOR 90** 设置下的应用程序可以在客户环境下运行。那些渴望使用更多深远功能的，如报表输出到 **XML** 或 **HTML** 格式，将仍然需要将附加的**_reportlistener.vcx/vct** 加入到他们的项目里。

总结

我的看法是，高质量应用程序的配置是完全需要所有组件的本地化。因此，你需要查看报表程序的源代码是必然的，并且你自己要仔细的本地化（或者，至少对于德语本地化，使用由德语 **FoxPro** 用户组 **dFPUG** 所提供的包含文件）。

你最后可以包含源代码到你自己的项目里，并为此生成独立的程序，并把它分发给你的客户。这个决定提升了每个开发者的体验。

下载：**512HABERMANN.ZIP**

一张图片顶一千句话

作者：**Andy Kramek & Marcia Akins**

译者：**fbilo**

无论你的数据是多么的完整，在很多情况下还是用图形化来表示信息的办法更简单。一种常见的特殊需求是根据时间进行数据分析。察看图形格式的数据要比死盯着表格中大量的行和列容易的多。然而，VFP 自己并没有内建的工具让我们可以由数据来建立图形，所以我们必须求助于别的方法。这个月，Andy Kramek 和 Marcia Akins 要解决用 VFP 代码和数据在 Excel 中建立一个图形的任务。

安　迪：你是这里的自动化行家了，这次我就全靠你了。我的客户想要看到在屏幕窗口和打印报表上以图形表达的销售趋势。要实现这个目标什么办法最好？

玛西亚：哦，既然你有打印图形的需求，那么最好排除使用 **MS Chart** 了，因为这么做惟一可以选择的输出格式只有剪贴板的图元文件了，而且没有简单的办法可以把它包含在一个报表中。

安　迪：好吧，那么就告诉我能做什么吧！现在我能做什么？

玛西亚：嗯，他们已经有了什么软件？

安　迪：没什么特别的。我确定所有的工作站都安装了 **Microsoft Office**，不过我不能确定他们安装了的 **office** 版本是否一样。

玛西亚：这样的话，你有两种明显的选择。你可以使用 **MS Graph**（它是 **Microsoft Office** 自带的）或者 **Excel**。

安　迪：我觉得 **MS Graph** 不过就是 **Excel** 图形引擎而已。难道它们不是同一件东西吗？

玛西亚：多少有一点一样，但它们保存和更新图形的途径是不同的。使用 **MS Graph**，你必须自己格式化数据并把数据直接交给这个图形工具。而在使用 **Excel** 自动化的时候，你只要把数据交给 **Excel**，然后告诉它去用数据来建立图形就够了。

安　迪：听起来好像直接用 **Excel** 会比较好，因为这么做的话至少我们还可以把工作表连带着图形和数据都保存起来，以供日后需要时修改。那么我需要干什么呢？

玛西亚：第一件事情是建立一个封装的类以处理导出数据并把它交给 **Excel** 的事情。

安　迪：在 **FoxPro** 基础类库中难道不是有一些类（**autograph.vcx**）是专门干这个的吗？

玛西亚：是这样没错，可它使用的是 **MS Graph**，而你刚刚还说过不打算这么做呢！

安　迪：是的，我不想用 **MS Graph** 了。目前我看不到不使用 **Excel** 的任何好处。现在让我们继续吧。

玛西亚：我们将把这个类建立为基于 **VFP** 的 **Custom** 基类。

安　迪：打住，为什么不使用 **Session** 类呢？**Session** 是专门为处理数据而设计的，并且它的 **PEMS** 也比 **Custom** 要少。

玛西亚：是的，但我想你将会从一个表单、或者一个报表上开始运行这个功能，那么你就已经有了一个带有所有重要数据的的活动数据工作期。你不会真的想要一个独立的、私有的数据工作期吧，是吗？

安　迪：没错，但如果你把一个 **Session** 对象作为一个子对象添加给表单（就是说，使用表单的 **AddObject()**方法），那么它就会使用该表单已有的数据工作期而不是自己新建一个。

玛西亚：但如果你使用 **CREATEOBJECT()** 去建立一个 **Session** 类的实例，然后把它赋值给一个表单的属性，那么你就一定会得到一个额外的私有数据工作期，有趣吧？我还是认为在我们想要从一个基于 **Session** 类的对象内建立图形类的实例的时候应该使用一个 **Custom** 类。那么，你到底需要建立多少个图形呢？

安　迪：多少个图形？你说的到底什么意思？我没有想过这个事情；象用户需要的那么多吧，我想。

玛西亚：哦，这样的话，我们应该把这个类做出是数据驱动的，这样你就可以在元数据中指定不同类型的图形，并让一个类用同样的方式来处理所有不同类型的图形类。

安　迪：哦，我明白了，你的意思是指要有多少种图形的类型。好吧，现在已经知道的是需要一个按地区的销售图，不过我敢打赌，一旦把这个图搞定以后马上就会被要求再做按客户的销售、按代理的销售、按时间段的销售等等新图。

玛西亚：**Okay**，我明白了。我们需要的第一件事情，是能够以一个指定的格式生成数据。**Excel** 希望用于一个图形的数据应该有特定的格式：第一列里放的是标志每一行的一个标签（**label**），而第一行里放的是标志每一列的标签，如表 **1** 所示：

表 1、Excel 要求的图形格式

	Year1993	Year1995	Year1996
Feb			835.43
Apr			1372.27
Sep		1006.86	

Oct	857.58	1188.66	

安　迪：可它看起来像是交叉表的结果，而且没有办法可以直接用一个查询就从源数据来生成它。

玛西亚：是不能，但请记住我们的目标是要建立一个将会处理所有的图形的类。所以第一个步骤是以标准的格式生成某些中间数据，如表 **2** 所示：

表 2、前期数据的标准格式

cMonth	nTotal	nYear	nMonth
Feb	835.4300	1996	2
Apr	1372.2700	1996	4
Sep	1006.8600	1995	9
Oct	857.5800	1993	10
Oct	1188.6600	1995	10

安　迪：如果我不想让数据按时间分组的话会发生什么事情？如果我想让数据按客户 **ID** 分组会怎么样？如果标准格式是这样的话，下一步处理不会很吃力吗？

玛西亚：不会，因为下一个任务是接收生成的无论什么中间数据，并将之转换成图形要求的格式。这也是为什么我们需要用元数据来驱动这个过程。我们必须既能够指定用于生成中间数据的查询、又能指定用于将结果集转换为最终格式的处理过程。我使用一个名为 **queries.dbf** 的表来定义这些事情，并将它们与一个特殊的用户友好的名称和图形类型相关联（见表 **3**）。

表 3、元数据的结构

字段名称	数据类型	说明
cqueryname	C (20,0)	用于找到该记录的候选键
cquerydesc	C (50,0)	该图形的英文说明(用于为用户生成一个可用图形的列表)
cpopupform	C (80,0)	一个用于收集需求参数的表单的名称(当需要的时候)
ngraphtype	I (4,0)	图形类型(条形图、饼图、等等)。真正的值是一个 Excel 常量
mquery	M (4,0)	要被执行以生成中间数据的 SQL 查询
cmethod	C (80,0)	将中间数据转换成最终格式的方法的名称
mtitlex	M (4,0)	给 Excel 中图形的各种属性分配的值
mtitley	M (4,0)	在本例中，我们只提供了这三个
mtitlez	M (4,0)	但你可以自己定义更多你需要的属性的值

安　迪：明白了。那么我们要如何才能由表 **2** 中的数据去生成表 **1** 中的数据呢？

玛西亚：这是我的标准处理方法中的一个——在这个案例中，这个方法的名称叫做 **MakeYearColumns()**。该方法将基于时间的数据按年分组。它的代码从中间数据中取出所有 **distinct** 的年和月，并建立一个输出游标，游标中有一个用于每个

月的行和一个用于每个年的列。

安　迪：聪明！那么现在我们有数据了，怎么把它们弄进 **Excel** 里去呢？我假设我们已经建立了一个 **Excel** 的实例，并打开了一个新的工作表（这里代码的细节请看下载文件中的 **Automate()** 方法）。

玛西亚：完成这个任务最快的办法，是以 **tab** 分隔的格式把数据放入 **Windows** 的剪贴板，然后使用工作表的 **Paste()** 方法来生成电子表格。

安　迪：这简单——我们可以使用 **_VFP.DataToClip()** 来把游标的内容拷贝到剪贴板里去。

玛西亚：只有我们正在数据工作期 **1**（就是俗称的“默认”数据工作期）上工作的时候、或者我们只会在表单上建立图形的时候，才可以用这个办法。如果我们从一个基于 **Session** 类的对象上使用这个类，**_VFP.DataToClip()**就无效了。

安　迪：你说的“无效”是什么意思？

玛西亚：它不会发生一个错误，但它也不会象你希望的那样工作。我想我应该说的是，在这个案例中，**_VFP.DataToClip()** 只对 **1** 号数据工作期才工作。如果你是在一个私有数据工作期中而且不是在一个表单上，那么如果你使用这个方法得到的将是 **1** 号数据工作期中当前选中表的内容（如果有这么一个表的话），或者如果没有任何表被打开，你就会得到最近一次放在剪贴板中的任何东西。所以我们将使用下面这样的代码来代替：

```
SELECT csrResults

*** 首先拷贝到一个临时文件
lcFileName = SYS( 2015 ) + '.txt'
COPY TO ( lcFileName ) TYPE DELIMITED WITH TAB

** 然后把结果放到一个字符串变量中
lcFields = FILETOSTR( lcFileName )
ERASE ( lcFileName )

*** 现在添加用来放字段名的那一行
lnFields = AFIELDS( laFields, [csrResults] )
lcFieldNames = []
FOR lnRow = 1 TO lnFields
    lcFieldNames = lcFieldNames + PROPER( ;
        laFields[ lnRow, 1 ] ) + CHR( 9 )
ENDFOR
lcFieldNames = lcFieldNames + CHR( 13 ) + CHR( 10 )
```

```
*** 现在把这些字段名和字段放到剪贴板里
_CLIPTEXT = lcFieldNames + lcFields
```

安　迪：但我们怎么定义这个图呢？我明白我们可以顺利的把这些数据粘贴到 **Excel** 里面，但我们把它们放在哪？

玛西亚：我们要为它定义一个 **range** （范围）。麻烦的是：这个 **Range** 的大小必须正确以能放置这些数据。

安　迪：一个 **Range** 难道不是通过指定其左上角和右下角的单元格来定义的吗？

玛西亚：是这样的，所以我们需要统计出列的数量，并指定相应的列标志符（**(1 = “A”，26 = “Z”**)。

安　迪：如果数据中的列超过了 **26** 个会发生什么事情？

玛西亚：你会得到一个没法读的图形！你最好用下面两种办法扩展这个类：或者增加额外的序列字母来生成**“AA...IV”**这样的标记、或者拆分数据来生成多个图形。

安　迪：足够了。当然，行号就简单了，只要用结果集的 **Reccount()** 就是了。

玛西亚：当然不对了。你需要在发送给 **Excel** 的数据中插入一个包含字段名称的行作为第一行。这里是定义 **Range** 并把我们拷贝到剪贴板中的数据粘贴给 **Range** 的代码：

```
*** 取得每条记录中字段的数量
lnFldCount = FCOUNT( 'csrResults' )

*** 取得行的数量并加 1，
*** 因为我们需要包含那个带有字段名称的行
lnRecCnt = RECCOUNT( 'csrResults' ) + 1

*** 取得工作表中数据的 Range
lcCell = 'A1:' + CHR( 64 + lnFldCount ) + ;
   TRANSFORM( lnRecCnt )

*** 把数据粘贴上去
loWB.ActiveSheet.Paste( .Range( lcCell ) )
```

安　迪：最后关头啦！我们已经有了带数据的 **Range**，但只是把数据放到 **Range** 里面并不会自动生成一个图形，是吗？大概现在我们需要调用在 **Excel** 里的一些其它方法吧！

玛西亚：首先我们必须要做一件小事情。我们必须确保在我们刚刚生成的 **Range** 的左上角单元格内是空的。

安　迪：为什么？

玛西亚：因为如果不是这样的话，**Excel** 就无法正确的生成这个图形。

安　迪：听你的吧。那么现在我们可以生成我们的图形了吗？

玛西亚：是的。我们需要做的是给工作表（**Workbook**）的 **Charts** （图表）集合添加一个新的 **Chart** （图表）对象，然后告诉图形使用哪个 **Range** 来生成它自己以及是否要使用在所有的行或者列中的数据来绘制这个图形（这是作为参数 **tnPlotBy** 传递的），象这样：

```
.SetSourceData(loWB.Sheets("ChartData").Range(lcCell), ;
    tnPlotBy )
```

然后设置它的图表类型。

安　迪：难道我们用来自行或者列中的数据去生成的图形会有什么不同吗？毕竟用的是一样的数据。

玛西亚：要不要打赌？如果我们使用来自表 **1** 中所有列的数据，我们会得到图 **1** 中的图形。而如果我们使用在所有行中的数据来生成同一个图表，那么我们会得到图 **2** 中的图形。

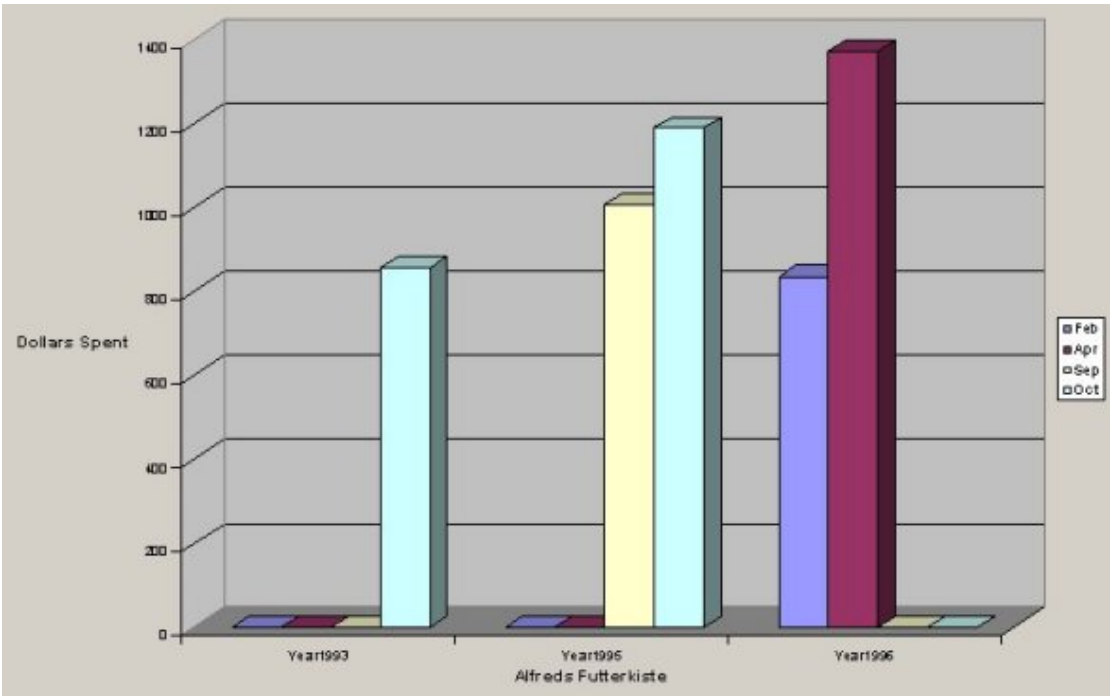

图 1、通过以列中数据绘制成的图表

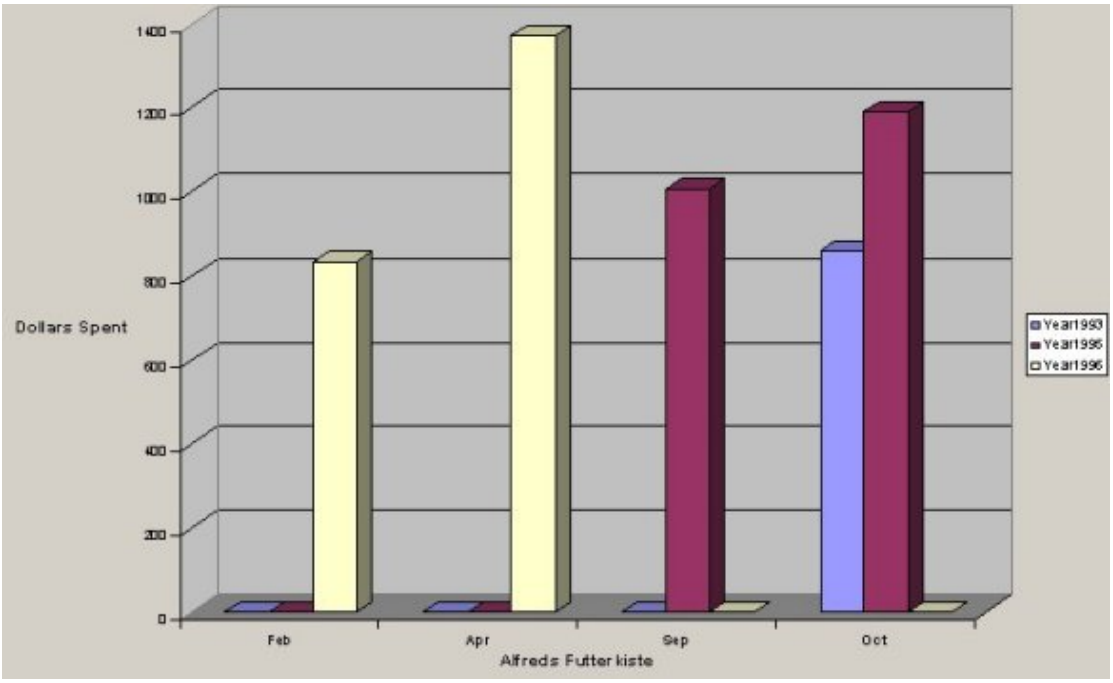

图 2、通过以行中数据绘制成的图表

安　迪：那么现在我们终于有了一个图表啦！接下来干什么？

玛西亚：把它保存为一个临时文件，然后把文件名返回给那个最先调用 **Excel** 自动化类的无论什么东西。

安　迪：但是我们需要显示这个图表，并能够在一个报表中打印它呀！我们现在所拥有的就是一个 **Excel** 文件，那么怎么把这个图表取出来呢？

玛西亚：最简单的办法是把它导出到一个带有一个 **General** 字段的游标中去。我们的示例表单就是这么做的，并在图表被取出后使用一个 **OleBoundControl** 来显示它。而取出图表所需的不过是：用相应的值取代下面代码中的两个名称后运行这行代码：

```
APPEND GENERAL <field_name> ;
    FROM ( <Excel_FileName> ) ;
    CLASS "Excel.Chart"
```

安　迪：真的？我不清楚你可以象这样取出一个图表。

玛西亚：取出图表的命令就这样了，不过在使用 **General** 字段作为一个 **OleBoundControl** 的 **ControlSource** 的时候还需要再多做一点事情。为了让这个控件能够正确的刷新，你必须首先取消绑定、然后取得新图表、最后再重新绑定控件。活不难，重要的是要按照这个顺序来做。

安　迪：我这么做了以后还能操作这个图表吗？比如设置标题、图中的说明（**legend**）、线条的样式等等。

玛西亚：当然可以。在这个时候，它是你表单上的一个活的、会呼吸的对象。这正是我为什么要在元数据表中加上那些额外的字段——这样我就可以通过读表中的这些值来设置图表的属性了。

安　迪：明白了！最后一个问题……

玛西亚：我知道，要把它放到一个报表里去，是吗？

安　迪：是的，请告诉我简单点，谢谢。

玛西亚：**Okay**，它就很简单！如果你是从“生成带 **General** 字段的游标”的那个表单来运行报表的话，它真的很简单。你只要建立一个带有一个 **OleBoundControl** 的报表，并把游标的 **General** 字段设置为 **ControlSource**。没法再简单了。

安　迪：**Cool**。而且我想如果我要不从那个表单上开始运行这个报表，我总是可以把游标保存到一个表里，然后使用表中的字段来作为 **ControlSource**。

玛西亚：是的，**General** 字段会变得相当大，不过我想如果你必须要实现这个任务的话应该能够忍受这一点的。老规矩，类和示例表单的完整代码包含在这个专栏的下载文件中。

下载：**512KITBOX.ZIP**