# 第一章 绪 论

## 1.1 命名约定

在本章开始时，需要了解一下贯穿全书的命名约定。

1. 应用程序(Applications)

隶属于一个给定的应用程序的所有表、程序、屏幕、菜单以及报表都以相同的两个字符开头。这些字符就是这个应用程序的“调用字母”。

例如，你会发现本书示例的应用程序中的所有表都是以“ex”开头的(由于“example”的缘故)。

“调用字母”的作用是双重的。首先，由于所有的应用程序部件的开头字符(“EX”)都是一样的，因而，可以很容易地使用简单的“ex*.*”备份整个应用程序。其次，我们可以很快地指出谁属于这个应用程序，而不必再查找其他任何目录项。

2. 表(Tables)

在给一个表命名时，头两个字母是为应用程序调用字母保留的，剩下的六个字符用来标识内容。

例如，存有客户数据的表的名字可以是“excust”。

3. 字段(Fields)

一个表中的字段的命名既是为了标识该表，也是为了标识该字段的内容。

正如应用程序调用字母一样，每个表都有一个两字符的唯一标识符。标识符后面紧跟的是一个下划线，它的作用相当于一个空格，因为中间嵌入空格是不允许的。剩下的七个字符用于表示这个字段的内容。

例如，客户表中的所有字段都以“cu_”开头。字段“cu_custid”表明这个字段的内容是来自于客户表的客户标识码。

这些字段级的命名约定的好处如下：

首先，我们可以立刻知道这个字段所在的表，而不必去利用数据字典或列出所有的表以及它们的结构。

其次，如果我们对剩下的有效字符利用得好的话，就可以知道这个字段的内容，而不必再在整个文件或数据字典中查寻。

第三，我们可以利用这个命名约定去表明来自不同的表的那个字段是相关(或外部)键字段。

例如，通过将定单表(exordh)中的客户标识码命名为“oh_custid”，就可以很明显地看出

字段 cu_custid 和 oh_custid 包含的信息是相同的以及它们分别来自的表。

第四，由于使用这个字段命名约定保证了明确的字段名，因此我们就可以在写一个 SQL_SELECT 时，直接写这个字段，而不必加表的别名前缀。

例如，我们能简单地通过写"oh_custid=cu_custid"将客户表和定单表连接起来。如果我们仅仅使用"id"作为字段名，那我们将不得不写成"excust. id=exordh. id"。

最后，我们在使用 SCATTER 和 GATHER MEMVAR 时不会出问题。由于这些命令都是基于这样一个前提，那就是变量与字段是匹配的，因而如果我们打算 SCATTER 多个表时，我们必须使字段名唯一。如果字段名不唯一的话，那么当把这些字段信息 GATHER 回记录时，将不能保证正确的字段值放到了正确的表中。

假定我们为客户文件中的客户 id 和定单文件中的发票 id 使用了字段名"id"。如果我们将两个表都 SCATTER 了，那么最后一个 SCATTER 的将是 m. id 的值。然而，如果我们现在 GATHER 到第一个表中，那么"id"就会被一个错误的值所代替。

另一方面的益处就是在决定程序引用的"id"是客户的，还是定单的时，不必再查阅大量的代码。

总而言之，合适的字段命名可以提高的应用程序的维护速度和新应用程序的开发速度。

4. 标识(Tags)

在简单的一个字段表达式上创建的标识(Tags)的名字就可以是这个字段的名字(例如，如果主要基于 cu_custid 字段的话，那么标识的名字就是 cu_custid)。这就使识别标识的根据更容易了。

当使用复杂表达式(表达式基于多个字段，字段的部分或任何一个有效的 FoxPro 表达式)时，要尽可能给出一个名字。

5. 内存变量(Memory Variables)

FoxPro 环境中有两种类型的内存变量，PUBLIC 和 PRIVATE。知道变量的类型是很重要的。

6. Public(公共)

Public 变量在应用程序中的任何一个地方都是有效的，直到它们被明显地释放掉。由于这个缘故，因而就可以很容易地通过给这个名字赋一个新值而偶然地改变这个变量的值。

一个公用的变量名可以是"action"。如果使用这个变量去决定运行哪个程序(最终用户选择的系统"action")的话，我们必须去维持这个值，直到又有一个新选择出现。因而我们可以定义一个 PUBLIC 变量，并且现在可在任何一个程序中访问当前系统选项。

但是，我们也许还想利用变量"action"去保存程序中的当前活动的值(例如增加)。一个程序中变量"action"值的改变同样会引起当前系统选项值的改变。这样，系统就会混乱起来。

为了避免这种变量值的冲突，MEI 在所有的 PUBLIC 变量前都加了一个"K"。这样，系统变量"action"就变成了"Kaction"。

7. Private(私有)

Private 变量意味着它仅在一个给定的程序或过程内才存在。一旦这个程序或过程终止了,该变量就不存在了。

由于在 FoxPro 中,缺省情况下,变量被定义成 PRIVATE,因而我们就能在一个应用内反复使用变量名,并能保证在每个程序或过程内我们都拥有当前程序的值,且不会与任何其他同名变量发生冲突。

但现在我们必须要解决一下 PRIVATE 变量的偶然覆盖写的问题。

我们将通过使用 PRIVATE 命令完成它,在 PRIVATE 命令后面紧跟的是对该局部过程/程序加言完全私有的变量表(例如 PRIVATE ans)。

现在,如果我们在一个过程中使用了相同的变量名(PRIVATE ans),FoxPro 将保持有两个不同的"ans"变量。当退出这个过程后,只有在这个例程中创建的"ans"变量才会被释放掉,另一个仍然存在,且仍保持原有的值。

这种类型的变量有时也称作"局部"(local)变量。为了方便在一个 PRIVATE 语句中命名局部变量,MEI 使用下面的约定:

所有的局部变量都以 L_开头。PRIVATE 语句变成:PRIVATE ALL LIKE L_*。

我们现在可以在程序或过程的开头使用这个声明了,并且自信不会危害到调用树中较高层的同名变量的值。

最后,如果我们想让一个变量被多个程序使用,但又不想在应用结束时显式地释放它,我们就可以给它任一个命名,只要它不以一个"K"或"_"开头。

# 第二章　事件驱动程序设计入门

本章要点：

- 应用程序模型
- 事件驱动组件

本章目的：

完成本章后，你将会：

- 应用程序设计模型间的差异
- 事件循环(Event Loop)工具
- 事件(Event)
- FoxPro 触发器(Trigger)
- FoxPro 对话(Session)
- FoxPro 标志(Flag)

## 2.1　应用程序模型

本小节目的：

当你完成本小节的学习后，你会理解下列程序模型间的差异：

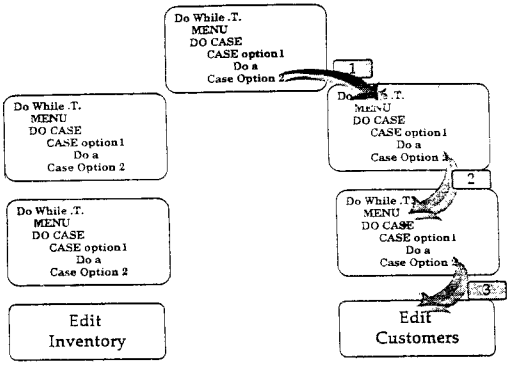
- 形式化应用程序(Modal Applications)
- 半形式化应用程序(Semi-modal Applications)
- 无模型应用程序(Modeless Applications)

在本节开始前，有必要先对事件(event)和对话(session)作一下定义。

一个"对话"就是一个独立的程序，比如编辑客户信息(如 excust)。

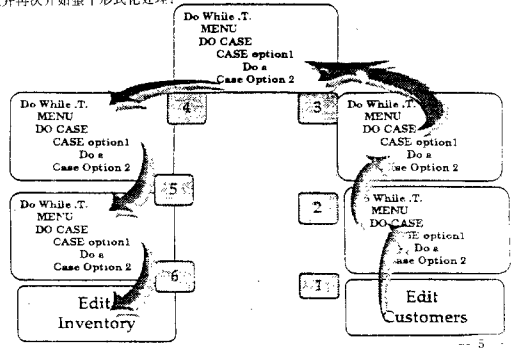
一个"事件"就是系统接收到的一个信号，它表明用户请求在当前对话中作一个更新，当"事件"被处理时，用户转向一个新的对话，此时不必关闭任何一个先前打开的对话，也不会从任何一个先前打开的对话中丢失信息。

**应用程序模型之一：形式化程序设计**




最早的程序设计模型是形式化模型。当你编写一个形式化应用程序时，用户必须通过一系列的菜单选择才能开始一个请求的对话(如编辑客户信息)。

为了选择另一个事件，比如像编辑库存，用户必须正确地从当前的对话中退出，然后从一系列的菜单中返回，这时程序才仅仅到达主菜单。在那点上，用户才可以选择另一个对话选项并再次开始整个形式化处理：



由于都是函数，因而形式化程序设计经常会很费时并且无效。毕竟用户真正想做的就是在尽可能少的使用菜单的情况下从一个对话(客户编辑)转向另一个对话(库存编辑)。

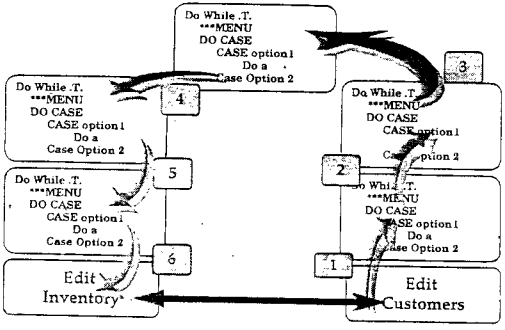
然后，为了给用户提供能从任何一个其他对话直接访问任何一个对话的功能，将面临许多问题。

首先，DO WHILE/ENDDO 循环的嵌套层数是有限制的(最多 16 层)。为了保持打开数个不同的菜单层和程序，这个极限很容易被突破。

其次，应用程序也只能使用 5 层嵌套的 READ 语句。这将不能满足在同一时间使所有用户的对话都可用的需要。

最后，FoxPro 的 READ 对话是独立的。也就是说，当用户转向另外一个独立的对话时，前一个对话的变量就会丢失。这样，当用户再次进入任何一个先前的对话时，他们将不得不再次进行整个更新和增加的过程。

**应用程序模型之二：半形式化程序设计**




半形式化程序设计可使用户自由地从菜单上直接选择一个新对话，而不必关闭当前的对话。

这样，用户就可以直接在菜单选项间转换，而根本不必管正确地从当前对话中退出。这是通过开发人员维护一个事件环实现的，事件环既控制当前对话的关闭，也控制使用户转向他们最新的菜单选择。

程序设计的半形式化模型虽然要比形式化模型好些，但它仍然不能为用户提供离开先前对话的灵活性，保证用户所有信息的完整性，以及当用户请求转向另一个对话时，打开这个对话。

**应用程序模型之三：事件驱动程序设计**

到目前为止你可能已经发现形式化、半形式和事件驱动程序设计的不同主要在于用户在不同对话间转换的自由程度。

实际上，这种自由程度在这些不同的应用程序模型的名字上已有所反映。例如，“形式化”程序设计就意味着在你完成和释放当前对话之前，你不能选择另一个行为(action)。

“半形式化”程序设计则表明用户没完成当前对话的情况下，可以选择另外的对话，尽管这个对话本身不能再保持打开的状态。

最后，事件驱动，或称“无模型化”程序设计，则为用户提供了一种直接在对话间转换并保证数据不受损害的能力。

你可能也已经发觉，如果没有大量的关于所要组件的考虑，要建立一个真正的事件驱动应用程序也不是很容易的。

## 2.2 事件驱动模型

本小节目的：完成本小节后，你将会：

- 事件驱动对话
- 事件驱动“触发器”
- 事件驱动“标志”

事件驱动应用程序“组件”(“component)包括：

- 一个处理对话请求的事件管理器
- 允许用户在对话内请求事件的触发器
- 应用程序用于维护各种打开的对话并辅助事件请求处理的标志。

我们将在下一节详细讨论事件管理器，因此让我们快速地浏览一下 FoxPro“对话”以及我们为什么需要“触发器”和“标志”。

**对话**(Session)

基本 FoxPro 对话命令都使系统处于一种“等待状态”。也就是说，系统会暂停处理，直到用户终止或中断该对话。这些对话命令包括 BROWSE，READ CYCLE，MODIFY FILE/MEMO/REPORT，ACTIVATE MENU 以及 READ VALID . T. 。

BROWSE 命令初始化一个 BROWSE 对话，该对话用于维护或显示应用程序的信息，比如像客户的信息。

正如 BROWSE 对话一样，READ CYCLE 用于使用户进行一个读/写对话。但是，READ 和 BROWSE 中的每个都包含有它自己的中断，并且谁都意识不到对方。换句话说，尽管一个 BROWSE 和一个 READ 可以共存于同一个普通的对话中(比如厂商维护)，但谁都不能直接与对方通信。确实，开发人员必须把它们当中的每一个都当成是同一程序中的一个独立的对话，但是，我们将在后面更详细地谈论这个问题。

ACTIVATE MENU 和 READ VALID. T. 命令都是用于处理菜单的，尽管这些命令可以在产品的 MS-DOS 版本中交替使用，但在 Windows 中 READ VALID . T. 则是一个更好的方法，

ACTIVATE MENU 命令在这种环境功能要弱些。

一旦用户处于这些对话中的某一个，我们就必须依靠各种各样的中断这个途径给事件管理器传递事件请求。这些中断就是"触发器"。

**触发器**(Trigger)

无论何时当遇到一个"触发器"或对话中断时，就可以请求事件了。

一个"全程"("global")触发器是独立于特定的"等待状态"的，它用 ON KEY LABEL 来定义。由于 FoxPro 是在内循环解释时间(如在执行代码行之间)检查 ON KEY LABEL 的，因而用户可在任何时间"触发"一个事件请求。

ON KEY LABEL 仅在一个特定程序内还可以有效地使用。当使用这个方法时，ON KEY LABEL 用于执行程序的主控过程。一个"全程"ON KEY LABEL 要比一个程序的特定触发器更常用。

由于 OKL 都是在命令之间被激活的，因而有一个小问题(尤其是在最终用户那里)。

假定我们用一个 OKL 做增加新记录的处理，代码如下：

```
ON KEY LABEL F3 Do addproc.
```

当用户按下{F3}后，程序控制权就传给了过程 ADDPROC。但是，由于 FoxPro 仍需要做所有的内循环解释工作，因而执行也就没有开始。用户则认为缺乏响应是由于按键失败造成的，因而再次按下{F3}键。

程序控制权再次被传递了，同时也引起了程序栈内的一个递归调用。

最终，用户将反复按键以致于在递归调用中产生了一个 FoxPro 错误。

为了在 OKL 中避免这个问题的出现，可以使用下面的代码：

```
ON KEY LABEL {KEY} mvar=;
IIF(PROGRAM()=="MAIN. PRG",myproc(),"")
```

其中：

{Key}是一个 OKL 键(如 F2)

mvar 是任何一个内存变量的名字

"MAIN. PRG"是 PRG 的文件名

myproc()是要执行的过程的名字

执行情况如下：

- 用户按下 OKL 键。
- PROGRAM()返回主控 PRG 的名字，这样程序控制权就传给了过程 MYPROC()。程序执行尚未开始。
- 用户再次按下了 OKL 键。
- PROGRAM()返回当前处于控制中的程序(如 MYPROC())的名字，这与主程序的名字不一样，因而从 OKL 返回的将是一个空串("")。
- FoxPro 只是忽略这个返回的空串并且当下一个命令准备好处理时，将执行 MYPROC()中的第一行代码。

在一个菜单内，ON SELECTION POPUP 命令用于控制把请求事件分配给一个"标志"，这个标志然后就传给事件管理器。

当事件管理器处理来自菜单的请求时，两件事中的一件必然发生。要么请求对话必须打开，要么用户必须处于合适的对话窗口。

一旦用户处于一个对话中，该对话就典型地被一个 READ CYCLE 或一个 BROWSE"等待状态"所控制。这两个对话命令都有中断，以及用于执行一个能触发一个事件请求的用户定义函数(UDF)的子句。

通过使用 GET VALID 或 GET WHEN，READ CYCLE 可以在输入对象级被中断。READ CYCLE 本身就具有中断子句，像 WHEN，VALID 和 SHOW。最后，利用 READ DEACTIVATE(离开一个窗口)和 READ ACTIVATE(进入一个新窗口)，窗口间的移动就能作为一个事件触发器了。

最后一个对话命令是 BROWSE。通过使用 VALID 和 WHEN 子句，可以中断 BROWSE 对话。

仅有的真正难点就在于 BROWSE 和 READ 对话间缺乏通信。尽管很容易控制从一个 READ CYCLE 到另一个 READ CYCLE 对话的转换，但很难监视 BROWSE 和 READ CYCLE 对话间的移动。然后，正如将在第十一章讨论的一样，总有办法实现它的。

**标志(Flag)**

本节的最后一个主题就是利用"标志"监视事件和对话，这些标志包括公共内存变量(如 kaction)、数组(如 kw)以及系统表(如 exsyst)的使用。

正如你会看到的那样，变量用于给事件管理器传递事件请求。数个数组用于维护信息，这些信息包括使用中的窗口，当前打开的对话和打开的顺序，以及每个对话的当前窗口位置。最后一个系统表用于维护来自每个对话的变量。

随着我们建立事件驱动应用程序的不断提高，每个"标志"的使用都会清楚的。

## 复　习

- 应用程序模型
- 事件驱动组件

## 复习思考题

1. 哪种应用程序模型能使用户选择一个新的对话而不必显式地关闭当前对话？
2. 说明形式化和事件驱动应用程序间的不同点？
3. 一个"事件"和一个"对话"间有何不同？
4. 列出 FoxPro 触发器及其所属的"等待状态"对话。

# 第三章　事件循环

本章要点：

· 主菜单

· 事件循环

本章目的：完成本章的学习后，你将会：

· 建立一个维护对话请求的菜单系统

· 创建一个事件管理器

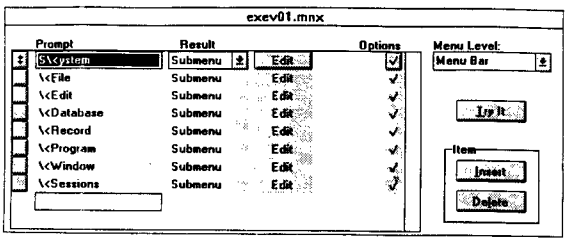
为了维护一个事件驱动应用程序，我们需要能处理来自用户的改变当前事件的请求。为了实现它，我们需要知道这个事件请求是什么，然后再满足这个请求。

这就需要我们弹出一个菜单，其中包括可能的对话名字。

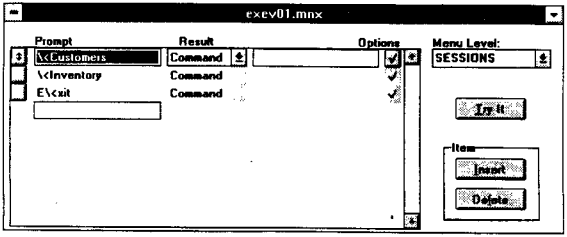
这是第一步，然后就是利用 Menu Builder（菜单构造器）去创建一个菜单项并与弹出式菜单建立联系。为了开始这个菜单对话，我们需要发出一个 MODIFY MENU 命令

```
MODIFY MENU exevmain
```

在建立了一个空的菜单结构后，我们将选择 QUICK SCREEN 把这个菜单置于 FoxPro 系统菜单中：

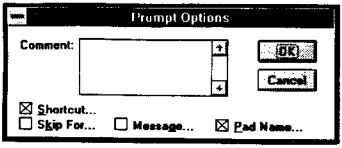



对于这个菜单，我们将增加我们自己的 SESSIONS 选项和一个包括了所有可能的对话的弹出式菜单（子菜单）：




由于 Windows 本来就依从于 CUA，因而我们就已经有了一个使用 ALT 加上一个设计好的热键(如\<Session)的简化键。但是，该产品的 MS-DOS 版本却不自动遵从 CUA，因此我们在设计菜单时就应把它考虑进去。

在同样的跨平台下为了菜单，我们必须使用 Options 核选框，这样才能为 MS-DOS 下增加一个简化键。对菜单项和子菜单选项都要做这样的工作。




由于 FoxPro 每次在菜单程序生成时都给菜单项分配一个它自己的唯一的很难识别的名字，因而我们需要给菜单项指定我们自己的名字。这个也是通过 Options 核选框对话实现的。一般地，我们都为菜单项选择一个有提示作用的名字(如对话)。

所有这些都完成后，我们用 PROGRAM 菜单项的 Generate 选项来生成这个菜单。一旦生成完毕，我们就可以使用下面的命令运行这个可行的菜单系统：

```
DO exevmain.MPR
```

## 3.1 事件管理器——Exevmain.PRG

仅仅"DO"菜单还是不够的，我还有三个基本问题。第一个就是尽管菜单可以被置于屏幕上，但它却不是激活的(注意一下光标仍然停留在 Command 窗口内)。

为了解决这个问题，我们需要创建一个程序（如 exevmain. prg），它用来使我们能在一个菜单对话内激活这个菜单。正如在前面提到的那样，可以使用 ACTIVATE MENU 或 READ VALID . T. 命令建立一个菜单对话。

假如这样的话，程序将包括下列代码：

```
SET TALK OFF
DO exevmain. MPR
READ VALID . T.
SET SYSMENU TO DEFAULT
```

当运行这个程序时，菜单系统就被激活了。但是在选择一个菜单选项时什么都不会发生。

请注意我们在这个程序的尾部加了一条 SET SYSMENU TO DEFAULT 命令。记住在你用自己设计的菜单系统替换掉了 FoxPro 系统菜单时，你需要在程序执行完后重新装入系统菜单。如果不这样的话，你会发现将不再能使用简化键，如用于帮助的{F1}和用于使 Command 窗口前行的{CTRL＋F2}。SET SYSMENU TO DEFAULT 命令会自动地重新装入 FoxPro 系统菜单。

还应注意在我们向事件驱动程序中增加代码时，新增加的代码将以无重音的斜体字出现在代码块中。

现在，再回到问题的解决上来。

第二个问题就是我们仍无法控制用户引起的行为。换句话说，我们无法确定用户选择了哪个菜单选项，并且即使我们能确定，我们仍无法按那个选择行事。

要解决这个问题需要好几个事件驱动工具。首先，我们需要一个包含了用户选择的事件请求的应用程序范围内的变量。其次，我们需要一个能给应用程序的“标志”赋合适的值的一个过程。最后，为了调用我们的事件更新过程，必须去激活其中一个“触发器”，或 FoxPro 对话中断。

我们每次只考虑其中一个问题。建立一个应用程序范围内的标志很容易。只需在 EXEVMAIN 程序中创建一个 PUBLIC 变量并将其值初始化成字符：

```
* * exevmain. prg
SET TALK OFF
PUBLIC kaction
kaction＝""
DO exevmain. MPR

READ VALID . T.
SET SYSMENU TO DEFAULT
```

既然变量已经定义好了，我们就需要一个能正确给它赋值（如用户的菜单选择）的过程。MEJ 在程序 L3KILLRD 中有这个例程。下面我们就看一下这个程序：

```
* l3killrd. prg
* Sets a control flag,and exits the current
* read cycle session
* p_var=variable to hold the flag value
* p_name=flag value to place in p_var
PARAMETERS p_var,p_name
DO vcontrol WITH "MENU"
RETURN
```

正如你所看到的，L3KILLRD 相当简单，你只需将这个标志变量的名字(如 Kaction)和想存储的值(如菜单选择)传给它。然后这个过程将把菜单的选择分配给 Kaction，并且我们都设好了(我们会马上回到 DO vcontrol WITH "MENU" 上)。

现在我们知道了用户发出的事件请求。但是，我们仍需要处理和满足这个请求。我们需要再一次增强事件管理器 exevmain：

```
* * exevmain. prg
SET TALK OFF
PUBLIC kaction
kaction=""
DO cxevmain. MPR
READ VALID . T.
DO CASE
    CASE UPPER (kaction)="CUST"
      WAIT WINDOW "Customer"
    CASE UPPER (kaction)="INVE"
      WAIT WINDOW "Inventory"
    CASE UPPER (kaction)="EXIT"
      WALT WINDOW "Exit"
ENDCASE
SET SYSMENU TO DEFAULT
PROCEDURE vcontrol
PARAMETERS dummy
RETURN
```

(注意由于 L3KILLRD 引用了一个过程(如 VCONTROL)，因此我们需要通过把这个过程放到事件管理器内来"尊敬"那个调用。但是，VCONTROL 过程什么都没做。)

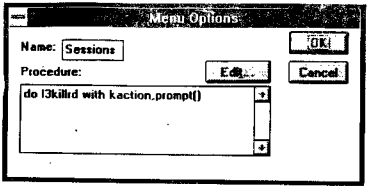
在我们能运行这个程序前，我们需要解决第三个问题：使菜单对话中断有足够长的时间以执行 L3KILLRD 程序。

菜单内仅有一个 FoxPro 触发器可用：ON SELECTION。既然我们想使这个触发器在从

SESSIONS 弹出式菜单产生一个选择的任何时刻都被"引发",我们就需要在一个生成的.MPR中创建一个 ON SELECTION POPUP sessions 命令。

我们通过在 Menu Bulider 中调用 exevmain 菜单,选择编辑 SESSIONS 子菜单,然后从 MENU 菜单项中选择 SESSIONS OPTIONS 来实现它。

我们在 Procedure 编辑区内输入对 L3KILLRD 程序的调用:




FoxPro 函数 PROMPT 包含了所选择的弹出式菜单提示(如客户)的值。假如 L3KILLRD 是这样工作的,无论何时当用户从 SESSIONS 弹出菜单做了一个选择时,那个选择的提示也将被放在变量 Kaction 中。

但是,你会发现在我们做了一个弹出式菜单的选择后,什么也没有发生。WAIT WINDOW 命令没有出现执行。其原因是菜单的 READ 对话仍然是激活的,它使菜单处于"热"态。只要菜单是激活的,别的行为就不可能发生。

因此我们必须"杀死"这个菜单对话,以使用户的请求能被处理。为此,我们将在 L3KILLRD 中使用附加的行:

```
DO vcontrol WITH "MENU"
```

由于 L3KILLRD 在将控制权返回给菜单前调用了一个过程,因而我们将利用它去发出一个 CLEAR READ,它将终止 READ VALID .T.。这样会依次结束菜单对话并允许用户的选择被处理:

```
* * exevmain.prg
SET TALK OFF
PUBLIC kaction
kaction=""
DO exevmain.MPR
READ VALID .T.
DO CASE
    CASE UPPER(kaction)="CUST"
        WAIT WINDOW "Customer"
    CASE UPPER(kaction)="INVE"
```

```
            WAIT WINDOW "Inventory"
        CASE UPPER(kaction)="EXIT"
            WAIT WINDOW "Exit"
    ENDCASE
    SET SYSMENU TO DEFAULT

    PROCEDURE vcontrol
    PARAMETERS dummy
    CLEAR READ
    RETURN
```

然而，当这个程序执行时，VCONTROL 过程在任何一个选择作出后就终止了 READ。实质上，用户只能作出一个选择，然后应用程序就结束了。这与我们在构造一个事件驱动对话时的想法完全相反！

我们需要的不是一个事件管理器，而是一个事件管理器循环。这可以通过把事件管理器(DO CASE/ENDCASE)放在一个 DO WHILE/ENDDO 中实现。当然，我们也需要在一个循环内移动菜单对话。事件管理器现在变成如下样子：

```
*** exevmain.prg
SET TALK OFF
PUBLIC kaction
kaction=""
DO exevmain.MPR
DO WHILE .T.
    DO CASE
        CASE UPPER(kaction)="CUST"
            WAIT WINDOW "Customer"
        CASE UPPER(kaction)="INVE"
            WAIT WINDOW "Inventory"
        CASE UPPER(kaction)="EXIT"
            WAIT WINDOW "Exit"
        OTHERWISE
            READ VALID .T.
    ENDCASE
ENDDO
SET SYSMENU TO DEFAULT

PROCEDURE vcontrol
PARAMETERS dummy
```

```
CLEAR READ
RETURN
```

现在，当用户第一次进行循环时，Kaction 的值将是空值并且 OTHERWISE 将被触发。依次一一激活了菜单。一旦用户选择了一个菜单选项，READ 就被清空并且循环再次执行。这次 Kaction 就会有一个合法的值并且它所匹配上的 CASE 语句中的包含的代码就会被执行（如 WAIT WINDOW "Customer"）。

但在 CASE 语句执行后会如何呢？由于 Kaction 的值没有发生变化，因而当遇到 ENDDO 时，循环再次执行。同一个 CASE 语句又被匹配上。现在用户陷入到一个死循环中，它将一直执行他们第一次做的菜单选择。

这可以通过在执行的那个 CASE 语句的代码前重新初始化 Kaction 的值避免：

```
* * exevmain. prg
SET TALK OFF
PUBLIC kaction
kaction =""
DO exevmain. MPR
DO WHILE . T.
    DO CASE
        CASE UPPER (kaction)="CUST"
          kaction=" "
          WAIT WINDOW "Customer"
        CASE UPPER (kaction)="INVE"
          kaction=" "
          WAIT WINDOW "Inventory"
        CASE UPPER (kaction)="EXIT"
          WAIT WINDOW "exit"
        OTHERWISE
          READ VALID . T.
    ENDCASE
ENDDO
SET SYSMENU TO DEFAULT

PROCEDURE vcontrol
PARAMETERS dummy
CLEAR READ
RETURN
```

现在无论何时当用户从与菜单选择相关的活动中返回时，Kaction 的值都不会与任何

个 CASE 匹配上，只有 OTHERWISE 被选上，并且用户也只留在菜单对话中。

随便说一句，在行为调用前安排 Kaction 的重新初始化是很有意图的。下面我们会看到。

最后，我们需要保证用户能从事件管理器循环中退出，然后终止这个应用。当然，我们只需要插一个 EXIT 命令于选项中：

```
* * exevmain.prg
SET TALK OFF
PUBLIC kaction
kaction =" "
DO exevmain.MPR
DO WHILE .T.
    DO CASE
        CASE UPPER (kaction)="CUST"
          kaction=" "
          WAIT WINDOW "Customer"
        CASE UPPER (kaction)="INVE"
          kaction=" "
          WAIT WINDOW "Inventory"
        CASE UPPER (kaction)="EXIT"
          EXIT
        OTHERWISE
          READ VALID .T.
  ENDCASE
ENDDO
SET SYSMENU TO DEFAULT

PROCEDURE vcontrol
PARAMETERS dummy
CLEAR READ
RETURN
```

现在，基本的事件管理器循环就完成了！

**练习：主事件循环**，Exevmain.MPR/Exevmain.PRG

- 创建一个菜单(exevmain)，它包括：
  - 一个名为 SESSIONS 的菜单项。一定要包括一个热键、简化键和菜单项的名字
  - SESSIONS 菜单项的子菜单，它包括下列选项：
    - 客户(Customer)
    - 库存(Inventory)