建立一套商业规则，并做一个商业规则服务器

作者：Stephen Settimi
译者：fbilo

根据商业模型及其相关的商业对象复杂性的不同，建立一套商业规则可能是一个令人生畏的经历。如果把模型和它的各个商业对象清楚的定义好了，那么开发过程就会轻松的多。这里就是这么一种途径。斯蒂芬首先向你演示了怎样构想出一个商业对象（它是一些小对象的一个集合），清楚的辨别和定义好了这些对象以后（这篇文章不涉及后面的部分），就是为这些可能需要规则的对象进行设置的时候了，定义这些规则、收集它们、然后通过一个商业规则服务器来运行它们，并把它们当作是离散的部件来维护。

正文：

一个商业规则也许就像校验一个字段的内容那么简单：字段的值是否不在一些指定值的范围内？或者，也可能比较复杂，比如象将一个字段的值与大量的其它内容进行比较：：A 的值大于 B 吗？当 D 等于 A 但小于 B 的时候，与使用 E 相比，使用 D 是否更有利？...(省略其它联想 nK 字。^_^)再复杂一点，你也许会发现你需要去对每一个结果与来自于其它数据源的另一套数据进行组合和查询；也许要检查一个引用和权限、可能需要进行递归分析、运行一些辅助性的校验。如你所见，这个过程会变得越来越庞大，大得几乎足以淹死你。

准确的给每个商业对象进行分类、并控制它们的粒度（译者注：我理解粒度这个词的意思，不过要给它下一个中文的定义解释给大家听却有点难度。不过别担心，看到后面你会了解“粒度”这个词是什么意思的。）将会有助于弄清所有的商业模型以及每个单独的部件。模型的各个部分被分清楚以后，定义从属于各个对象的规则就变得非常轻松了。然后，对每个部件与其它部件的属性进行交叉比较，就得出了商业规则。要注意的一点是，当商业规则执行完毕的时候应该返回一个逻辑值。

对商业对象们进行准确的分类，并控制它们的粒度

定义商业规则的第一步，是分解出专用的商业对象。这里是一个典型的商业对象，不过它的粒度还没有控制：

- 一个销售订单

如果对粒度进行细化的话，这个对象可以被分解成一些内部对象：

- 一个销售订单
  1、订单项
  2、客户

再细化一下粒度，你会看到更多的对象：

- 一个销售订单
  1、定单项
    a、价格
    b、折扣
    c、细节
  2、客户
    a、姓名
    b、住址
  3、销售员
    a、姓名

我们可以走得更远一些：姓名分成 First、Last、Middle 和头衔；住址分成街道号码、街道名称、城市、州、邮编。如你所见，任何一个对象都可以轻松的被分成多个子对象，不过，当你象这样不断细分到某种程度以后这样做的好处就会开始降低。你到底需要写多少代码？你到底要在设计和开发每个部件并训练那些新程序员们掌握它们上分配多少管理资源？把电话号码拆分成区号、前缀、和号码部件真的值得吗？在某些情况下值得这么做，而另一些情况下则不然。商业规则部分的意思就是到底需要何种程度的粒度。这要由你自己决定。

这里的另一种观察角度是：一个订单（商业对象）其实是一个“能够接受客户希望给它的任何指令”的抽象。在 RAD 幕后的概念，是要知道定义常常会改变。因此，第一个任务就是要能改变这个部件而又不会弄乱商业流程。

更重要的是，某些情况下，定义需要与其它对象相交互。这是不可避免的。从某种角度上来说，你现在正处于关键对象之类定义的外围部分，并且开始处理它与其它对象之间的问题。这种交互作用也许是通过引用、也许是通过“没有定义在当前类中、但仍然是当前类的成员”的插件来添加更多的维度以抽象自己。注意这些插件对象有它们自己的定义和约定规则。

在前面的例子中，销售员是销售订单对象（oSalesOrder）中的一个对象。它没有定义在 oSalesOrder 中，而是用 Addobject()来作为一个插件被引用的。同样，客户和订单项是在 oSalesOrder 对象中的其它对象引用。看看下面这个狐狸的 Logo，它看起来就像是个迷宫，里面有不少相互连接的部分，每一个部分都是一个对象——组合起来的整体就是一个 Logo（见图 1）。跟图 2 比较一下，在图 2 中展示的是一个销售订单是怎样由它的各个部分组合起来的。

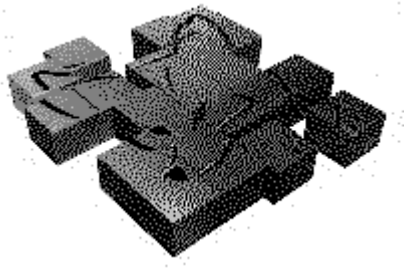

图 1

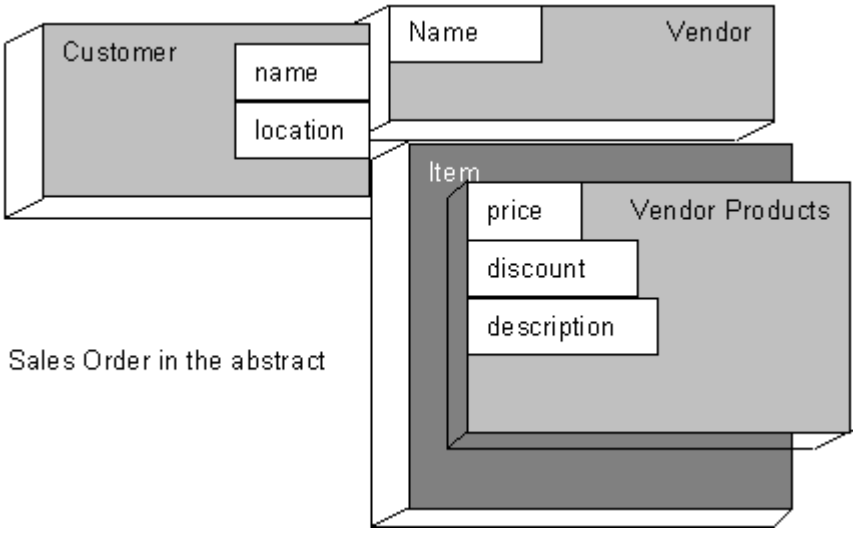

图 2

由多个对象所表现出来的粒度其实是由一些独立部件组合起来形成了一个大的商业对象。oCustomer(客户对象) + oItem（订单项对象） + oVendor（销售商对象） = oSalesOrder（订单对象）。这些部件中的每一个，都可以被定义为一个类，都可以拥有自己的一套商业规则。但是组合起来，就是一个商业对象，附带着一套商业规则。每个子对象及其商业规则都可以被单独的对待，但是它们之间的相似之处远超过不同，而将它们组合起来一起处理会更有效率。这篇文章讲述的，就是怎样分别建立各套商业规则。

VFP 的 DBC 浏览

在 VFP 的 DBC 中，存储过程是作为一个默认的函数库来服务的，在存储过程里可以放入需要被全局访问的任何函数或者过程。参照完整性过程放在这里，有时候某个主关键字的函数也放在这里。我们通常会把一系列的商业规则都放在这里。但是，存储过程文件并非就是放置商业规则的唯一地方，甚至也许根本不应该将商业规则放在存储过程文件里。尤其是当你的商业模型经常会变动时更应该如此。要记住的是，即使在 DBC 被打开时也可以对

过程文件进行编辑，不过，象其它文件一样，在使用时它是不能被编译的。

必须承认，把一些商业规则代码当作存储过程直接绑到某个表上的做法相对来说要容易些。一般是这么做的：执行 MODIFY TABLE，然后在字段的有效性规则、默认值、记录有效性规则、表触发器中添加一些函数调用，而这些函数的代码则是放在 DBC 的过程文件中的（就是放在存储过程里）。每当这些事件中的任何一个发生的时候，在过程文件中的代码就会被触发。现在，假定你想改动一下代码或者调用的方式，如果系统正在运行的话，你就有麻烦了：

1、打开 DBC；
2、找到表对象；
3、找到需要被改动的字段；
4、修改代码；
5、找到下一个需要被改动的字段；
6、修改代码...

即使在可视化的 VFP IDE 界面中，这么做都是费时费力的，更不用说如果这是一个被共享的 DBC 的话，那就是一个 Impossible mission 了。这里的关键在于，你必须要象黑客一样偷偷摸摸在别人不干活的空挡里把一切迅速搞定，这种情况是应该避免的。

与一个商业对象相关的所有象商业规则这样的东西，都应该被收集起来集中储存在一个位置（类或者文件中），并被当作只用于那个商业对象的一套商业规则来对待。只有当其它商业对象作为当前商业对象的插件出现时，其它商业对象的规则才可以调用当前对象的这些规则。从效果上来说，商业规则就是商业对象的影子。

类似的错误是：把商业规则放在了表单上，比如放在一个数据绑定型控件的 Valid 或者 LostFocus 中、或者放在一个表单的自定义方法中。记住，如果数据绑定型控件的值有了什么变化，DBC 的过程文件永远是最先被调用的——早于任何表单上或者 Valid 方法中的代码。只要记住了这一条，我们就应该好好的利用 DBC 过程文件提供的优点。不要把真正的商业规则直接存储在这里，而是在这里去引用一些外部的商业规则服务器。

保持商业规则与 DBC 或者其它接口之间的分离状态是我们的主要目标。保持收集好了的各套商业规则定义清晰、自包含，也是需要的，如果需要做某些策略上的改动，那么对小对象进行改动显然要比对一个庞然大物进行斧削要更有效率。为了实现这些目标，我将引用外部的商业规则。在做出决定时唯一要考虑的是，把这些规则放在哪里才更容易访问、实现起来的开销更少。是否任何时候我都可以访问这些规则？规则是否能以几乎可以忽略不计的开销被无缝执行？

商业规则

商业规则将在 DBC 之外进行维护，但又要利用 DBC 的自动化机制，以直接调用与商业对象相关的商业规则集。在这种模式下，DBC 的过程文件起的作用就好像是一个路由器。

基于这篇文章的目的，并且为了说得更清楚些起见，我将把商业规则部件称作是一个外部文件。毕竟，我们通常总是会听到大量混合的术语：说“对象”的时候实际上是在说“类”，说到“过程”的时候实际上说的是一个“函数”。正是大量这样的小问题影响了我们的交流。稍后，如果合适的话，我可以将与某个特定的规则相同的代码作为一个方法封装给一个 Custom 类。

为了把一个外部文件作为一个商业规则来使用，可以简单的把字段级别的校验代码放到一个 PRG 里面，并把这个文件与用于其它商业规则对象的文件一起保存在一个只用于商业规则的专门目录里。

在 DBC 中使用一个存储过程作为路由器

如果在表结构中注册了一个对某个函数或者过程的调用、而且该函数或过程存在于 DBC 的存储过程中，那么，只要注册了调用该函数或过程的字段、记录或者表发生了任何改动，该函数或者过程就会自动被调用。而且，由于 DBC 的存储过程起着类似于一个打开了的函数库的作用，因此，你可以从任何地方调用存储过程。在这里，我在存储过程中建立了一个命名为 BizRules 的函数，它接受的参数有：表名和字段名、一个值为"V"（校验）或"D"（默认值）或者"VD"（行校验）的标志。保存在存储过程中的这个函数其实并非是商业规则。它只是把请求传递（Route，也可以译作路由）给商业规则服务器的一个指令。我在这里说的“服务器”指的是任何执行商业规则的部件（程序、COM、FLL 或者任何你想用来处理商业规则的对象）。稍后，我将更详细的用几个具体的部件的形式来演示怎么维护这些代码。

假设我们现在要在 salesorder.dbf 表的 item 字段中调用 BizRules ,那么语法如下：BizRules("Salesorder.item", "V")。就是这个表达式，被放在表或者视图的 Item 字段的字段级校验中。为同一个字段获得任何默认值的语法与此类似，只是有个标志改动了一下：BizRules("SalesOrder.Item", "D")，要行校验的话也一样：BizRules("SalesOrder","RV")，区别是：这里不需要有字段的名称。（见图 3）

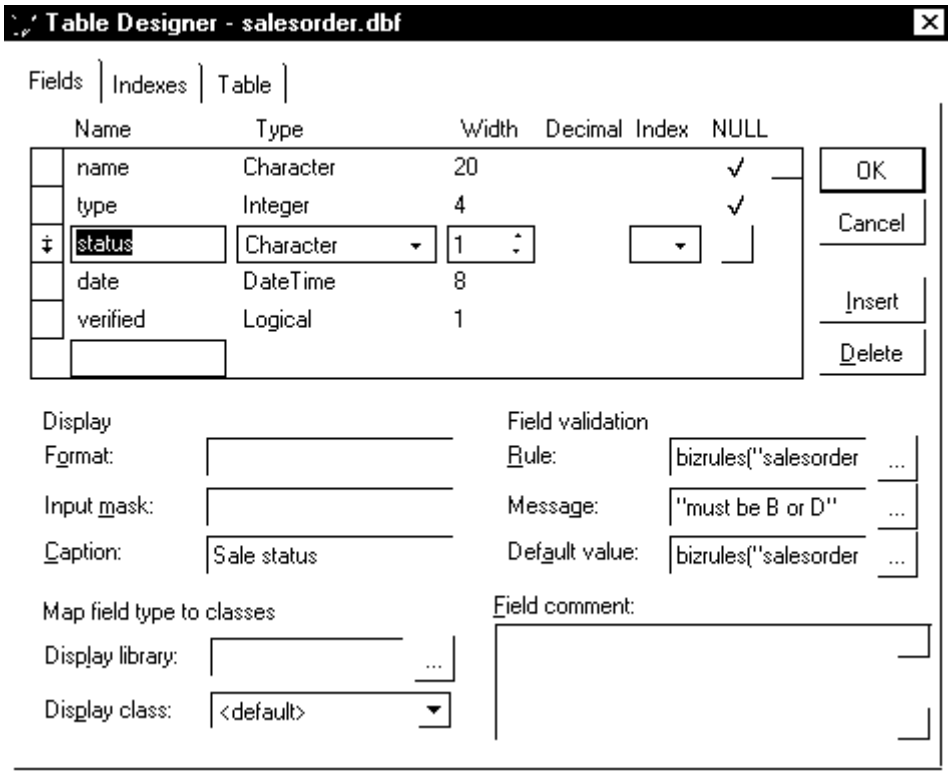

图 3

为了演示，在这里我只使用了一个精练的订单表（SalesOrder.dbf），该表只带了少量的字段（顺便说一句，你不会在这个表里看到什么价格字段，我猜这些东西是免费的:-)...）在每个字段的有效性规则和默认值里面，我只放入了一个简单的函数代码。对行校验也会那么做，虽然现在看不到这个校验，它在页框的第二页上，见图 3。注意，这里我还添加了一个字段标题（Caption）和一个文本片断，如果校验规则失败则会显示这段文本。这里没有连接的类——但即使仅仅给表结构添加了这么一点装饰酒可以给开发的速度带来戏剧性的增长，尤其是在表单的设计方面。

BizRules()会对调用请求进行分析，然后传递给相应的外部过程。我根据这样的命名规则给包含规则的外部文件命名：给文件名加上一个 Biz 前缀，后面跟着的是 DBF 文件名。比如说，销售订单商业规则对象的规则文件名将是 BizSalesOrder。

用于所有销售订单的所有商业规则对象都会在这个文件里找到。任何带有一个附属规则的字段的任何默认值也都在这个文件里。如果有行校验的话，它也会在这个文件里。在这个文件的内部是一些独立的过程，用于各种类型的调用。在这个示例里有三个：一个执行校验，一个获得默认值，一个检查一条记录的完整性。记住：调用那个规则文件，是由存储过程中的 BizRules 来决定的。不过，如果你不愿意使用 DBC 的触发器机制的话，那么查些文件也可以在存储过程之外的地方被调用。

建立和运行一个简单的商业规则

BizRules 会分析调用请求、取出字段和来源并传递它们。在这里，它还可以把任何数据绑定型控件作为一个对象参数来接受，它会取出该控件的 Control Source，然后以同样的方式对待请求。如果数据没有通过字段校验，你可以决定怎样处理事件：TableRevert()、默认值替代、CurVal 或 OldVal 替代、或其它你想要的操作。在这个示例里，我只是给用户一个简单的消息，但是我用一个更精练的 RulesMessageHandler 来代替标准的 messagebox 以根据模式来提供更多的选项。在这个示例里，我会把任何出现的脏数据替换成一个可以接受的默认值。这种途径对于新记录来说是可行的，但是它不适用于修改记录的情况。

在存储过程中工作的 BizRules 会在退出存储过程之前就处理好商业规则失败的事情：或者换上一个可接受的默认值、或者恢复到原始数据。不管哪种情况，从存储过程返回给 DBC 表的肯定是一个逻辑值 .T.，以避免默认的 VFP 警告用户有效性规则被违反。VFP 自己的这种途径并非那么糟糕，糟糕的是到那时候一切都晚了。

下面是存储过程中 BizRules 的代码，其中的注释很好的解释了整个过程：

```
PROCEDURE BizRules
LPARAMETER cType,cMethod

SET DATABASE TO demo
SET LIBRARY TO foxtools ADDITIVE
PRIVATE cAlias
*!* RETURN .T.

IF TYPE('cType')="C"
      * 判断是字段还是行规则
      cAlias=iif('.'$cType, ;
            left(cType,at('.',cType)-1),cType)
ELSE
      cAlias=left(cType.ControlSource,   ;
            at('.',cType.ControlSource)-1)
ENDIF

*!* 判断这是一个数据源调用还是对象调用
*! 处理数据源调用 (默认)
IF TYPE('cType')="C" && 来自 DBF 的调用
      IF '.'$cType
            IF   ISNULL(&cType) .and. cMethod="V"
                  RETURN .t.
            ENDIF
```

```
        ENDIF
        cControlSource=cType

        IF '.'$cType
            cField=right(cControlSource, len(cControlSource) ;
                - rat('.',cControlSource))
            cClass="biz"+(LEFT(cControlSource, ;
                RAT('.',cControlSource)-1))
        ELSE
            cClass="biz"+cControlSource
        ENDIF
ENDIF

*!* 处理对象引用形式的调用
* 如果是来自一个对象请求的调用
IF TYPE('cType')="O"
        IF ISNULL(cType.Value) .and. cMethod="V"
            RETURN.T.
        ENDIF

        * 将预先设定给该对象的默认特性再次分配给该对象
        oObject = cType
        cField = right(oObject.ControlSource, ;
            len(oObject.ControlSource)-         ;
                at('.',oObject.ControlSource))
            cClass = "biz"+(LEFT(oObject.ControlSource, ;
                RAT('.',oObject.ControlSource)-1))
ENDIF

*!* 对 BusinessRule 服务器的调用
*!* 如果要使用放在 COM 中的外部过程的话
*!* 需要将 CREATEOBJECT 那一行注释掉
IF TYPE('BizRuleServer')!="O"
        BizRuleServer=createobject('&cClass..&cClass')
ENDIF

IF TYPE('BizRuleServer')="O"
        DO CASE
            CASE UPPER(cMethod)="V"
                *!* 只有在使用 COM 的情况下才需要这段代码
                *!* 否则会出现 OLE 错误 "别名没有找到"
                xValue=IIF(TYPE('oObject')="O", ;
                    oObject.Value,Eval(cType))
                ReturnValue = BizRuleServer.DoValid(cField,xValue)
```

```
            CASE UPPER(cMethod) = "D"
                ReturnValue = BizRuleServer.DoDefault(cField)
            CASE UPPER(cMethod) = "RV"
                SCATTER to aRow
                ReturnValue = BizRuleServer.DoRowValid(@aRow)
                IF Type('ReturnValue') != "L"
                    cFieldNo = ReturnValue
                    ReturnValue = ''

                    FOR i = 1 to words(cFieldNo,' ')
                        *!*  对于逻辑型字段不做提示
                        *!*  只用字段规则来校验
                        if TYPE(field(val(wordnum(cFieldNo,i,    ;
                            ' '))))="L"
                            loop
                        endif
                        ReturnValue=ReturnValue+ ;
                            field(val(wordnum(cFieldNo,i,' ')))+','
                    ENDFOR
                    ReturnValue=lower(LEFT(ReturnValue,    ;
                        len(ReturnValue)-1))
                ENDIF
        ENDCASE
        Release BizRuleServer
ELSE
        SET PROCEDURE to (cClass)
        IF '.'$cType
            ReturnValue=&cClass(cField,cMethod)
        Else
            ReturnValue=&cClass(cAlias,cMethod)
        Endif
ENDIF

*!*  测试和处理返回值
*!*    为字段有效性规则或默认值处理返回值
IF '.'$cType
        IF TYPE('ReturnValue')="L" .and. ;
            !ISNULL(ReturnValue).and. cMethod!="D"
            *!*  默认值是一个逻辑值  .T.  的情况下
            IF ReturnValue
                RETURN .T.
            ELSE
                IF TYPE('oObject')="O"
                    cRuleText=DBGETPROP(oObject.ControlSource, ;
```

```
                    "FIELD","RuleText")
                cCaption=DBGETPROP(oObject.ControlSource, ;
                    "FIELD","Caption")
            ELSE
                cRuleText=DBGETPROP(cAlias+"."+cField, ;
                    "FIELD","RuleText")
                cCaption=DBGETPROP(cAlias+"."+cField,;
                    "FIELD","Caption")
            ENDIF
        ENDIF
    ELSE && 默认值
        RETURN ReturnValue   && 默认表达式
    ENDIF
ELSE
    IF TYPE('ReturnValue')!="L"
        cCaption=Proper(cAlias)
        cRuleText=DBGETPROP(cAlias,"TABLE","RuleText")
        cRuleText=cRuleText+' '+ReturnValue
    ENDIF
ENDIF

*!* 如果规则校验失败，则向用户显示一条消息
*!* 选择一条文本来显示
IF TYPE('cRuleText')!="U"
  *!* 如果一个错误没有相应预定义的错误消息
  *!* 则现在制造一条
  cRuleText = iif(Empty(cRuleText), ;
    'Developer: missing rule text',cRuleText)
  cMessage = cCaption + ' '+STRTRAN(cRuleText,"","")
  cMessageCaption = 'Please review...'

  *!* 用自定义对话框来根据模式向用户提供各种选择
  *!* 基于演示的缘故，这里只是用默认值来代替
  MESSAGEBOX(cMessage,cMessageCaption)
  IF '.'$cType
    ControlSource = cAlias+'.'+cField
    REPL &ControlSource with ;
        IIF(TYPE('BizRuleServer')="O",BizRuleServer. ;
        DoDefault(cField) ,&cClass(cField,"D"))
  ENDIF
  RETURN .T.
ENDIF
ENDPROC
```

BizSalesOrder 规则

下面是一套用于销售订单(Sales Order)商业对象的一套商业规则。实际上——并且还要根据数据规范化的程度、当前视图或表单的不规范化程度来决定——我们将透过相对应的 biz 文件来看看几个 DBF，不过所有的 biz 文件其实都在这个商业对象的商业规则里。

如前所述，在访问商业规则前的开头两步其实都是由 BizRules 来完成的——判定数据源、字段名，然后把相应的商业规则指定给那个数据源，还有判定这是一次校验(V)、行校验(RV)还是要取默认值(D)。然后，就会发出一个对相应的商业规则文件或者一个 COM 商业规则服务器的调用。用于各种类型调用的真正规则如下：

```
****************************************
*!* BIZRULES for SalesOrder: bizsalesorder
****************************************
LPARAMETERS cfield,cRuleLevel
LOCAL lReturn

DO CASE
    CASE cRuleLevel="V"
        LOCAL lAuthorized
        DO CASE
            CASE cfield='name'
                IF   !'-'$salesorder.name   .and. ;
                     !ISNULL(allt(salesorder.name))
                     lReturn=.t.
                ENDIF
            CASE cfield='type'
                *!*   因为只是演示，所以注释掉了下面的代码
                *!*   oTransType=CreateObject('TransType')
                *!*   oTransType.Type=salesorder.type
                *!*   lAuthorized=oTransType.ValidCode
                *!*   oTransType.Release()
                IF   BETWEEN(salesorder.type,1,5) .and. ;
                     !ISNULL(salesorder.type) &&.and. lAuthorized
                     lReturn=.t.
                ENDIF
            CASE cfield="status"
                IF UPPER(salesorder.status)$"BD"
                     lReturn =.t.
                ENDIF
            CASE cfield="date"
                IF BETWEEN(salesorder.date,gomonth(date(),-12), ;
                     gomonth(date(),12))
```

```
                    lReturn =.t.
                ENDIF
        ENDCASE
        RETURN lReturn
*****************
*!* 默认值
CASE cRuleLevel="D"
        LOCAL DefaultVal
        DO CASE
            CASE cfield='name'
                DefaultVal=.Null.
            CASE cfield='type'
                DefaultVal= .NULL.
            CASE cfield='status'
                DefaultVal="B"
            CASE cfield='date'
                DefaultVal=date()
            CASE cfield='verified'
                DefaultVal=.F.
        ENDCASE
        RETURN DefaultVal
*****************
*!* 行校验
*!* 这里只是检查一下各个字段中是否都已经输入了数据
CASE cRuleLevel="RV"
        * 在 COM 中，这一行将用 LPARAMETER aRow 来代替
        SCATTER to aRow
        LOCAL lReturn,cFields
        cFields=''
        i=0
        FOR each one in aRow
            i=i+1
            IF !EMPTY(one)
                lReturn=.T.
            ELSE
                lReturn=.f.
                *!* 对于逻辑型字段不作报告,
                *!* 只使用字段规则校验
                if TYPE('one')="L"
                    loop
                endif
                cFields=cFields+Field(i)+' '
                *!* 这一行只用于 COM 的情况下
                *!* cFields=cFields+rtrim(str(i))+' '
```

```
            ENDIF
        ENDFOR
        RETURN iif(lReturn,lReturn,cFields)
ENDCASE
```

注意，在这些规则中可能也会包含对另一个商业对象的调用，而那个商业对象自己也有一套相关的规则，这些规则可能是安全性检查、用户权限、或者其它用来判定是否批准的间接信息。象这样额外的调用可能也是合适的。例如，也许只有“A”先生可以完成一个销售事务。其它人可以输入数据，但只有“A”可以最后完成整个事务。或者，某些事件也可能会调用在 BizSalesOrder 对象中的 DoRowValid(cAlias) 方法而不是一个商业对象来检查当前用户的状态。商业规则的执行也会参照前面同样的概念，不过不是校验在 SalesOrder.dbf 中的字段，而是检查在 user.dbf 中的字段或 oUser (一个商业对象或者随便什么东西，你可以自己决定怎么去实现它)的一个属性。

所有这些规则（不管它们是放在外部过程文件中还是在某些对象的方法中）的组合会组成商业对象的整套商业规则。对于任何正统的 OOP 程序员来说，下面的一步当然就是把所有这些商业规则作为方法收集在一个专用的商业规则类 BizSalesOrder 中，并废除外部程序文件了。不过我们还是先别跳的那么快。

在建立一个商业规则服务器中的多种选择

这里是一些在建立商业规则服务器时的参考方案：

1、就用我们上面已经讲述过的方案。本质上，就是用一个 DBF 中的存储过程来调用一个外部过程——Okay，这我已经做过了，不过这种方案并非真正的实现了一个服务器，并且这当然不是一个 OOP 的方案。

2、使用单个商业规则服务器，这个服务器将成为一个类，而把所有 DBC 过程文件中的 BizRules 以及外部过程集分别作为这个类的一个方法保存起来——通常只有当你已经对整体表现的 OOP 代码已经失望了时才会做这样的选择。

3、一个多部件商业规则服务器类，它将存储过程放在一边，自己作为 OLE Public Custom 类来管理所有独立的规则——这里我们可以实现多个独立的服务器，但是这些服务器其实都寄生在一个 DLL 里面。嗯，这也是一种可能性。

4、还是使用单个商业规则服务器，不过是把 BizRules 打包到一个类里，而将所有的各套商业规则放在一个独立的 COM 部件里。

5、继续使用将存储过程作为一个对 COM 的路由器的那种方案，与方案 1 不同的只是用一个 COM 来代替外部过程调用。

6、不使用上面任何一种方案。

方案 1 和 5 我们已经有了：使用 COM 还是外部过程文件的区别只是在 BizRules 中注释掉了一行。

方案 2 我们不讨论。

方案 3 和 4 是关于在 BizRules 之外建立一个类，然后，或者在存储过程的外面（方案 4）或内部（方案 3）用调用来建立这个类的实例。以后的工作就可以象下面这样的代码在存储过程里来完成：

```
PROCEDURE BizRules
LPARAMETER cType,cMethod
    IF TYPE("BizRuleServer")!="O"
        SET CLASSLIB TO servers
        BizRuleServer=createobject('bizruleserver')
    ENDIF
    BizRuleServer.BizRules(cType,cMethod)
    return .t.
ENDPROC
```

我喜欢 DBC 过程文件的原因是它的通用性。你可以从任何地方调用放在存储过程里的函数和过程。

使用 COM

在已经将商业规则写做外部过程文件以后，我看着它们，对我自己说："从结构上来看，它们非常象一个类；所以我想我也许可以建立一些 COM 部件"。我可以调用这些 COM 部件中的任何一个把它自己当作一个服务器。但是我并不喜欢将 BizRules 从 DBC 的过程文件中拿出来做成一个类的想法，所以我将不去动它。这样，它就可以调用任何一个 COM 对象或者一个外部过程。我怀疑外部过程运行起来将比相应的 COM 对象要快得多。稍后，我将使用 VFP 中的代码范围分析器来验证这个假设。

首先，建立一个 Custom 基类 _BizRuleServer，并将它保存在一个服务器类库里。本质上，它是一个带着三种类型规则调用的外壳（Shell）：DoDefault()、DoValid()和 DoRowValid()。至于表更新、插入和删除，我把它们留给参照完整性生成器——为什么要重建这个功能？也许稍后我会扩展它。

我唯一会放在 _BizRuleServer 的类定义中代码只有象在外部过程文件中那样的 Parameter 语句（如果你正在使用 6.0，这个语句将会在子类的层面上被继承；如果不是的话，请在建立每个新的子类的时候自己重建这个 Parameter 语句）。现在，是时候去建立新

的项目以及为我试图要建立的商业规则集建立子类了。我将分别为销售订单、购买订单、用户安全性建立一个类，每个类都是 BizRulesServer.vcx 中的 _BizRuleServer 类的一个子类。这些新的类、它们的类库、以及项目都将与它们相应的商业规则对象一样有着同样的假名。将每个类标记为 OLE Public 类。在分别给这三个项目中的每个新子类的 DoDefault()、DoValid() 和 DoRowValid() 方法添加了代码之后（除了行校验以外，几乎都是与相应的外部过程文件中同样的代码），从项目管理器中编译成 COM（如果你用的是 VFP5，那么还需要做一个 main.prg 来编译这个部件）。你会注意到添加到项目里的只有你定义为 OLE Public 的商业规则类和 _BizRuleServer 子类，后者在项目被编译的时候会自动被添加到项目里。

```
BUILD PROJECT bizsalesorder FROM bizsalesorder.vcx
BUILD DLL bizsalesorder FROM bizsalesorder recompile
```

不要被 Build 命令或者 CreateObject() 函数中多余的命名方式所困扰。这个被命名的对象就是 BizRules 中会引用的那个。如果你仔细查看在 BizRules 中的代码的话，这种命名的特点可以给你带来好处。

我将象我建议的那样继续使用 DBC 过程文件来保存 BizRules 过程，如果想简化拦截字段和表改动的方式的话，那么 DBC 过程文件在这方面做得很好。

建立 BizRulesServer 的实例

在 BizRules 中建立单个的商业规则服务器的实例。必须要说的是，请记住建立对象实例的开销主要是在它的初始化过程里。在预计会重复出现同样类型事务的地方，不要释放这个对象，只管继续引用它，直到真正的商业对象（就是说，一个销售事务）被释放为止。当然，这个技术也有它自己的缺点。如果 COM 当前正在被使用，那么你就不能更新它。外部过程文件的情况也是如此。当你选择建立单个类服务器（方案 1 或 2）、或者一个多部件服务器（方案 3）的时候，这种情况尤其严重。没问题，前一种方法还是过程性代码而不是 OOP，但使用两种方式中的任何一种其优点都是一样的，只有一个例外：运行过程性代码速度要快不少。使用代码范围分析器后记录的时间如表 1，测试使用的是一台 K2 233 32M RAM 的机器。

表 1.过程性代码和 OOP 之间的响应时间比较

使用 COM			
Cold start			
2 Hits	1st 5.854	Avg 2.927	BizRuleServer=createobject('&cClass..&cClass')
1	1st	Avg	ReturnValue=BizRuleServer.DoValid(cField,xValue)

Hit	0.329	0.329	
1 Hit	1st 0.207	Avg 0.207	ReturnValue=BizRuleServer.DoRowValid(@aRow)
Cold start 和重复命中			
10 Hits	1st 5.865	Avg 1.130	BizRuleServer=createobject('&cClass..&cClass')
5 Hits	1st 0.356	Avg 0.087	ReturnValue=BizRuleServer.DoValid(cField,xValue)
5 Hits	1st 0.196	Avg 0.047	ReturnValue=BizRuleServer.DoRowValid(@aRow)
使用 PRG (FXP)			
Cold start			
5 Hits	1st 0.035	Avg 0.012	ReturnValue=&cClass(cField,cMethod) && 字段校验
5 Hits	1st 0.027	Avg 0.027	ReturnValue=&cClass(cAlias,cMethod) &&行校验

这些数字看起来几乎可以忽略不计。但是，当你真正运行这些代码的时候，区别就是很明显的。

注意从对象代码的“Cold Start”之后开始，时间发生了戏剧性的下降。不过请不要将孩子跟洗澡水一起泼掉，至少不要完全泼掉。同时使用 COM 和外部过程文件也有一个优点。我在考虑版本升级，或者...好吧，有些东西需要想一下。

先把区别放在一边，由于 COM 服务器的体积很小，并且使用范围严密，当 DLL 需要被更换的时候，该运行库被锁定的机会就被极大的减少了。不过，PRG 也有同样的优点。

最后的事项

下面是一些最后要注意的事项。

DBC 没有打开...还是可以使用商业规则

当商业规则校验(COM 或 PRGs)和用于路由调用请求的 DBC 过程文件中的 BizRules() 一切就位以后，不管是直接在一个表中还是从一个浏览窗口中进行工作、通过过程性代码还是命令行、或者与任何数据绑定型控件一起、不管 DBC 是否已经打开，这个校验都会起到应有的作用（事实上，一个数据库中的表被打开的时候，这个数据库的 DBC 文件也就自动

被打开了，只是还没有把这个数据库设置成当前数据库而已。在 BizRules 顶部的一点代码就是干这个的。)。了解了这一点以后，你也许就会发现在 BizRules 顶部没有注释掉的 Return .T. 这一行代码在调试规则或者成批加载数据的时候是非常有用的。

对象和 BizRules

有时，你也许会需要操作某个控件的值而又不想导致 BizRules 被触发，或者想让 BizRules 只在某些事件中才会被触发。为了达到这个目的，你必须要临时解开控件与数据源的绑定而稍后再重新绑定。你可以在控件的 INIT() 方法中记下控件的初始数据源。然后，再去操作它的值。从效果上来看，在这里你就像是在真空中工作一样。如果要运行商业规则，则记下新的值，重新将控件与它的初始数据源绑定，再给控件赋以那个新值，那么 BizRules 就会被触发。此外，你也可以通过编程将控件作为一个对象引用直接发送给 BizRules: = BizRules(this_or_that, "V")，或者象这样来取默认值： = BizRules(this_or_that,"D")。

表驱动的字段校验、默认值、行校验和触发器设置

不使用 VFP 自带功能的结果往往是必须重建这种功能。使用 ADBOBJECT()、DBGETPROP 和一对 For...ENDFOR 再结合使用 AFIELDS() 和 DBSETPROP()——所有这些都是由一个同样用这些函数调用建立的表来驱动的——你就可以在一个地方集中设置你所有的规则、规则文本、触发器、行校验等等东西。这要比通过 DBC 本身更容易来驾驭。如果你想要从系统范围来改变规则或者基于这些规则重新生成一个 DBC、或者使改动对商业对象产生影响的话，这种办法是非常方便的。这个小工具的代码可以从 http://www.home.sprintmail.com/~settimi 找到。

并不可笑的警告

COM 除了难以调试以外，对别名来说它看起来简直就是个瞎子＋哑巴。例如，象 Select 1 这样的语句是有效的，但是 Select SalesOrder 则无效。 IF xValue...ENDIF 可能有效，但是 IF salesorder.name...ENDIF 则无效。为了调试的方便，按照我在这里设计的进程来做是有益的：开始一个过程，测试；定义一个类、并使用已经测试过的过程、测试这个类、生成 COM、测试。

VFP：胜利、地位、或者荣耀...只管在它身上下注吧！

使用 VFP 来建立一个商业规则服务器、商业对象、或者一个后台数据服务器，能够让产品在这些日子以来被那么热烈追捧的三层应用中的任何一层中找到位置。一个商业规则服务器可以用于三层应用策略中的中间层。用 VFP 做的表单和 ActiveDocument 可以用在三

层应用中的第一层。VFP 的数据库可以用于后台。这就是完整的画面。它足以说明，VFP 和它的 COM 功能可以适用于今天的任何商业应用中的所有层上，在席位更加有限和昂贵的今天，这个好产品可以为任何商业应用和新软件开发节省大量的开销。