File Your Collections

Doug Hennig

While collections are extremely useful, array-based collections with a lot of members can suffer from poor performance at load and destroy time, and take up tons of memory. A file-based collection, on the other hand, takes no load or destroy time at all, since member objects are created on an "as needed" basis.

In my July 1998 column, "Collecting Objects", I wrote about collections in VFP. I mentioned in that article that I was starting to use collections quite frequently. Today, I'm even more enthusiastic about them; I doubt I've written anything in quite some time that *didn't* use a collection for something. Often, these collections just consist of a class that contains an array of object references. Sometimes, the objects in a collection are very simple: they don't have any methods, but are just holders of properties. Other times, they're more complex objects, such as forms.

Just to refresh your memory, here are a few of the reasons I have for using collections:

- Frequently, collections are replacements for arrays; I originally created my collection class to replace an array property with many columns. I found I was always forgetting which column certain information had to go into. It's a lot easier to understand and maintain code that simply sets or gets properties of objects in a collection than working with rows and columns.
- Since a collection is an object, it's easier to pass to a method than an array (after all, there's nothing harder to pass than an array, other than maybe a kidney stone). For example, with an array, you have to use @ to ensure it's passed by reference, but you can't do that with a array property, so you end up copying the array property to a local array, passing the local array to the method, then (if the method changed the array) copying the array back to the array property. Oh, but don't forget to redimension the array property first or you might get an error. With a collection, you can omit several lines of ugly code. Passing arrays is even more complex when the object you're passing it to is a non-VFP COM object, which may not deal with arrays the same way VFP does.
- It can be easier to search for something in a collection than in an array. ASCAN searches all columns, not just the one you're interested in, so it may find false matches. Also, if the array contains references to objects, you won't be able to use ASCAN at all. With a collection, you can code for the exact search behavior you want in your collection class, then simply call the appropriate method.
- You can more easily protect the contents of a collection than you can an exposed array property.

In my July 1998 article, I presented two classes: SFCollection and a subclass called SFCollectionOneClass. SFCollection was the parent for all collection classes, and allowed objects of different lineage in the collection. A collection managed by SFCollectionOneClass consisted of objects of a single class (a more common scenario). Since I wrote that article, I created a higher-level parent class called SFCollectionAbstract. This class, from which SFCollection is subclassed, has almost no code but really just defines the interface for collection classes.

Problems with Array-Based Collections

While an array-based collection works great for small and moderately sized collections, it doesn't do so well when there are a lot of members in the collection. For example, one form I created instantiated a field collection class and loaded the collection with information about the 11,000 fields that existed in an application. When I ran the form, I discovered to my horror that it took several minutes to come up. The reason is because it had to instantiate 11,000 objects and fill several properties for each one. Even closing the form took a long time, because all 11,000 objects had to be destroyed. In addition, an enormous amount of memory was used up by this collection.

I hated to give up the idea of using a collection, but clearly this wouldn't work. Then it occurred to me: what if the collection class was just an object-oriented interface to a table of collection members? There'd be no array of objects to load, keep around in memory, and destroy. Instead, when asked for a reference to a member object, the collection would find the matching record in the table, create the member object on the fly, and fill in the properties from the values in the table just before returning the object to the caller. Although the performance of requesting an object is a little slower (rather than simply finding an item in an array and returning the object stored in the appropriate array element, the code has to SEEK in a table, create an object, and copy field values to properties of the object), there's no load or destroy time, so the perceived performance is greatly improved and memory requirements plummet with the new scheme.

So, I created SFCollectionFile, a subclass of SFCollectionAbstract that provides a file-based collection. The collection table can have any structure but needs a field to hold the names of collection members and a tag on that field so SEEK can be used to locate a member. So, SFCollectionFile can be used to create collections of meta data, reports in an application, customers, invoices, etc.

SFCollectionFile

In order to work with a collection table, SFCollectionFile needs to know three things about the table: the path and name of the table (stored in the cCollectionTable property), the name of the field containing the names of the collection members (put this into cObjectNameField), and the tag to use for SEEKs on that field (the cObjectNameTag property). In addition, it needs to know the name and library of the class to use for member objects (the cItemClass and cItemLibrary properties) and the name of the property in that class where the member name will be stored (cObjectNameProperty).

The Init method of SFCollectionFile sets the protected cCollectionAlias property to SYS(2015) so it contains the unique alias we'll use for the collection table; that way, if another collection uses the same table, there won't be a collision in alias names. However, it doesn't open the collection table since cCollectionTable may not be set yet. The OpenCollectionTable method is responsible for opening the collection table; it ensures the various required properties contain valid values, and if the collection table isn't already open, opens it and sets the lOpened property to .T. The code for this relatively simple method is mostly error trapping:

```
local llReturn
with This

* Ensure the cCollectionTable property is filled in and
* the specified table exists.

  assert vartype(.cCollectionTable) = 'C' and ;
    not empty(.cCollectionTable) ;
    message 'SFCollectionFile: cCollectionTable ' + ;
    'not specified'
  assert file(forceext(.cCollectionTable, 'dbf')) ;
    message 'SFCollectionFile: ' + cCollectionTable + ;
    ' does not exist'

* If the table's already open, return .T. Otherwise,
* try to open it and return .T. if we succeeded.

  if used(.cCollectionAlias)
    llReturn = .T.
  else
    use (.cCollectionTable) again shared ;
      alias (.cCollectionAlias) in 0
    llReturn = used(.cCollectionAlias)
    .lOpened = .T.
  endif used(.cCollectionAlias)

* Ensure the cObjectNameField, cObjectNameTag, and
* cObjectNameProperty properties are filled in.

  assert vartype(.cObjectNameField) = 'C' and ;
    not empty(.cObjectNameField) and ;
    type(.cCollectionAlias + '.' + ;
    .cObjectNameField) = 'C' ;
    message 'SFCollectionFile: cObjectNameField ' + ;
    'not valid'
  assert vartype(.cObjectNameTag) = 'C' and ;
    not empty(.cObjectNameTag) and ;
    tagno(.cObjectNameTag, '', .cCollectionAlias) <> 0 ;
    message 'SFCollectionFile: cObjectNameTag not valid'
  assert vartype(.cObjectNameProperty) = 'C' and ;
    not empty(.cObjectNameProperty) ;
    message 'SFCollectionFile: cObjectNameProperty ' + ;
    'not specified'
endwith
return llReturn
```

The ReleaseMembers method (which is called from the Destroy method in SFCustom, the ultimate parent of this class) closes the collection table if it's open and lOpened is .T.

Because SFCollectionFile is a subclass of SFCollectionAbstract, it inherits a Count property that contains the current number of members in the collection, and a Count_Assign method that makes this property read-only to anything outside this class. However, Count_Access is overwritten; we have to count the number of records in the collection table, respecting any filter and deleted records. Here's the code:

```
local lnSelect, ;
  lnCount
with This
  if .Count = 0
    lnSelect = select()
    .OpenCollectionTable()
    select (.cCollectionAlias)
    count to lnCount
    .Count = lnCount
    select (lnSelect)
  else
    lnCount = .Count
  endif .Count = 0
endwith
return lnCount
```

Notice the count is only calculated if it's 0. It's way too inefficient to use COUNT TO every time this property is accessed, so we'll have to ensure that we keep it up-to-date as members are added and removed.

SFCollectionFile also inherits an Item array property and its access and assign methods. However, in this class, Item isn't actually used to store member objects; instead, we're just using its access and assign methods to do our work (Item_Assign actually does nothing). The purpose of Item_Access is to locate the specified member (either by name or by index number) in the collection and return an object for that member. As you can probably guess, we'll do a SEEK into the collection table to find the object by name or a SKIP from the start of the table if an index is specified (way less desirable for obvious reasons!). If we find the member, AddItem is called to create an object to hold the member's properties and GetObject is called to fill the new object's properties with the values from the current record in the collection table. In this class, GetObject is abstract because the implementation will vary with the structure of the collection table. Here's the code for Item_Access:

```
lparameters tuIndex
local loItem, ;
  lnSelect, ;
  lcItem
loItem   = .NULL.
lnSelect = select()
with This
  do case

* Ensure the collection table is open. If not, we
* can't continue.

    case not .OpenCollectionTable()

* If we have a character index, look for it in the table.

    case vartype(tuIndex) = 'C'
      lcItem = padr(tuIndex, fsize(.cObjectNameField, ;
        .cCollectionAlias))
      if seek(upper(lcItem), .cCollectionAlias, ;
        .cObjectNameTag)
        select (.cCollectionAlias)
        loItem = .AddItem()
        .GetObject(loItem)
        select (lnSelect)
      endif seek(upper(lcItem) ...

* If the index is numeric, go to that record in the
* table.

    case vartype(tuIndex) $ 'NFIBY' and tuIndex <= .Count
      select (.cCollectionAlias)
      locate
      skip tuIndex - 1
      loItem = .AddItem()
      .GetObject(loItem)
      select (lnSelect)
  endcase
endwith
return loItem
```

Unlike other collection classes, SFCollectionFile's AddItem method doesn't actually add anything to the collection, but simply creates and returns an empty member object. The reason it doesn't add anything to the collection is because most of the properties of the member object are empty, so there's no point in adding it to the collection table yet.

```
lparameters tcName
local loItem
with This
  loItem = MakeObject(.cItemClass, .cItemLibrary)
  store tcName to ('loItem.' + .cObjectNameProperty)
endwith
return loItem
```

To actually add a member to the collection, or to save changes made to an existing member's properties, call the SaveItem method, passing it the member object. Having to specifically call this method is the only downside to using a file-based collection, since it makes the behavior of this collection different from others. SaveItem checks if a member with the same name of the passed member already exists, and if not, adds a new record to the collection table and increments the Count property. Then it calls the SaveObject method to save the member object's properties to the appropriate fields in the table. As with GetObject, SaveObject is abstract in this class because the implementation will vary with the structure of the collection table.

```
lparameters toItem
local llReturn, ;
  lcName, ;
  lcField, ;
  lnCount
with This

* Ensure the collection table is open.

  llReturn = .OpenCollectionTable()
  if llReturn

* Add a record to the table if it doesn't exist.

    lcName  = evaluate('toItem.' + .cObjectNameProperty)
    lcField = .cObjectNameField
    if isnull(.Item[lcName])
      lnCount = .Count
      insert into (.cCollectionAlias) ;
          (&lcField) ;
        values ;
          (lcName)
      .Count = lnCount + 1
    endif isnull(.Item[lcName])
    .SaveObject(toItem)
  endif llReturn
endwith
```

The RemoveItem method removes the specified member from the collection. If the member can be found, it’s deleted from the table and Count is decremented.

```
lparameters tuIndex
local loItem, ;
  llReturn
with This
  loItem   = .Item[tuIndex]
  llReturn = vartype(loItem) = 'O'
  if llReturn
    delete in (.cCollectionAlias)
    .Count = .Count - 1
  endif llReturn
endwith
return llReturn
```

Example: Field Meta Data

As an example of a file-based collection, look at FieldCollection in SAMPLE.VCX. This subclass of SFCollectionFile provides meta data for fields in an application. cItemClass and cItemLibrary are set to Field and SAMPLE.VCX, respectively; the Field class is simply a subclass of SFCustom with the custom properties we’ll see in a moment. cCollectionTable is set to Fields, so FIELDS.DBF will be used for the collection table. cObjectNameField and cObjectNameTag are both set to FieldName, so a field with that name and a tag on that field with the same name must exist in the collection table. cObjectNameProperty contains cName, so that’s the property in the Field class that contains the name of a field member.

The Init method creates FIELDS.DBF if it doesn’t exist:

```
with This
  if not file(forceext(.cCollectionTable, 'dbf'))
    create table (.cCollectionTable) free ;
      (FIELDNAME C(30), ;
      TYPE C(1), ;
      SIZE N(3), ;
      DECIMALS N(3), ;
      CAPTION C(60))
    index on upper(FIELDNAME) tag FIELDNAME
    use
  endif not file(forceext(.cCollectionTable, 'dbf'))
endwith
dodefault()
```

The two methods we have to implement in a subclass of SFCollectionFile are GetObject and SaveObject. GetObject is called to fill the passed object’s properties with values from the current record in the collection table. As you can tell from the code below, the collection table fields named FIELDNAME, CAPTION, TYPE, DECIMALS, and SIZE map to the cName, cCaption, cType, nDecimals, and nSize properties of the Field class.

```
lparameters toField
with toField
  .cName     = trim(FIELDNAME)
  .cCaption  = trim(CAPTION)
  .cType     = TYPE
  .nDecimals = DECIMALS
  .nSize     = SIZE
endwith
```

SaveObject does just the opposite: it writes the properties of the passed object to the fields in the current record of the collection table.

```
lparameters toField
with toField
  replace FIELDNAME with .cName, ;
      CAPTION       with .cCaption, ;
      TYPE          with .cType, ;
      DECIMALS      with .nDecimals, ;
      SIZE          with .nSize ;
    in (This.cCollectionAlias)
endwith
```

FIELDS.PRG shows how FieldCollection can be used. It instantiates a FieldCollection object and if the collection is empty, opens the CUSTOMER table in the VFP sample TESTDATA database, spins through all the fields in the table, and adds a member to the collection for each one. It then simply prints the name and caption of each field. The cool part is if you run this a second time, the CUSTOMER table isn't opened. Thus, we have persistent meta data for fields that's independent of the data itself. This is especially useful when working with non-VFP data (such as Access or SQL Server) where you want to minimize access to the database server.

```
loCollection = newobject('FieldCollection', 'Sample.vcx')
if loCollection.Count = 0
  wait window 'Getting field information...' nowait
  open database (_samples + 'DATA\TESTDATA')
  use CUSTOMER
  lnFields = afields(laFields)
  for lnI = 1 to lnFields
    lcField           = lower(laFields[lnI, 1])
    lcAField          = 'customer.' + lcField
    lcCaption         = dbgetprop(lcAField, 'Field', ;
      'Caption')
    lcCaption         = iif(empty(lcCaption), ;
      proper(strtran(laFields[lnI, 1], '_', ' ')), ;
      lcCaption)
    loField           = loCollection.AddItem(lcAField)
    loField.cType     = laFields[lnI, 2]
    loField.nSize     = laFields[lnI, 3]
    loField.nDecimals = laFields[lnI, 4]
    loField.cCaption  = lcCaption
    loCollection.SaveItem(loField)
  next lnI
  use
  wait clear
endif loCollection.Count = 0

* Now display the fields.

clear
for lnI = 1 to loCollection.Count
  loField = loCollection.Item[lnI]
  ? 'Field: ' + loField.cName, 'Caption: ' + ;
    loField.cCaption
next lnI
```

Enhancements

Once a member has been located in the collection table, the Item method calls AddItem to create an object for the member and calls GetObject to fill properties of the member object with the values in the current record of the collection table. This has the advantage that the property names of the member object can be different than the field names in the table, and some fields in the table may not be presented as properties of the member object; the code in GetObject is responsible for mapping between the fields in the table and the properties of the object. However, if you don't mind having the property and field names match, having a property for every field, and having property values padded with spaces to the length of fields rather than being trimmed, performance can be improved by using SCATTER NAME. This command creates an object with one property for every field in the selected table and sets the property values to the field values in the current record. This has the additional advantage that you don't have to create a class just to hold member properties (such as the Field class in the example above).

Other useful enhancements might be the ability to filter the collection (for example, so you only see fields from a single table in the collection) and providing a method to create an array of member names for processing purposes or to be used as the row source for a combobox.

Other Thoughts

While a file-based collection works well in a VFP-only LAN-based application, other techniques might be better suited to different environments. You might wish to use ADO or XML as a collection tool in DCOM or Internet applications. Like SFCollectionFile, they are essentially object-oriented interfaces to data (in the case of XML, when using an XML Document Object Model, or DOM), but they can be marshaled over connections much more easily and efficiently than VFP objects or data can be. Because neither ADO nor XMLDOM have a collection-like interface of Item, AddItem, and RemoveItem methods, you may want to create wrapper classes to provide this type of interface to these objects.

Conclusion

In addition to better performance and lower memory requirements, file-based collections have the additional advantage of having persisted values. This may provide an additional performance boost over collections that have to build their content dynamically every time they're instantiated. I'm sure that, like me, you'll find a lot of uses for collections in your development efforts.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.