Data Storage: Entry Caching part 2 of 3

by Bob Ebert, Newton Developer Advocate, ebert@newton.apple.com

Introduction

This series of articles focuses on information useful to those who have mastered the basics of the Newton data storage APIs. They assume the reader is already familiar with NewtonScript and the Newton data storage concepts in the Newton Programmer's Guide (NPG) and has some experience writing Newton applications.

In part 1 of this series we talked about how soups index their entries, and how to create efficient searches. There we asserted that reading in entries was slow. This article gives the details behind what happens when entries are read and written. Knowing how this works can help you write the most efficient applications.

Correction

In part 1 of the series, the first paragraph in the first section, "Under the Hood," showed incorrect calls to `soup:AddXmit` and `soup:AddToDefaultStoreXmit`. Both calls were missing a second argument, which is the change symbol for the notification.

Caveat Coder

Warning: This article contains undocumented, unsupported details regarding how the current release of the Newton OS caches data storage objects. The information is presented because I believe that understanding what's going on helps. Knowledge like this makes it easier to design efficient code, easier to debug problem code, and may give you ideas for your own designs. This information is not presented so that you can write code that relies on the current design. Do not do this.

At least one developer made this mistake, creating a 1.x application that used an undocumented slot in a cursor. That application subsequently failed because of changes made for the 2.0 release of the OS. What was worse was that the operation in question could have easily been done in a supported way.

> *There is almost always a supported way to do what you want. Look for it!*

Entries

As explained in part 1, entry data isn't kept in the user store in a form that is directly accessible to NewtonScript. There isn't anything like a frame on the user store. Instead, strings are written one place,

the rest of the frame is serialized and written to another place, and tagged and indexed slots are stored still elsewhere for efficient searching.

When you call the `Entry` method of a cursor, you get a frame built up from the data on the user store. We generally blur the distinction between the data on the store and the frame that temporarily exists in memory, calling both things an **entry**. When a distinction needs to be made, the frame in the NewtonScript heap is called the **cached entry frame** or simply **cached entry**.

At some point the OS must read data from the store and create the cached entry frame. Obviously, this must be done before any data in the entry can be accessed. However, it's often the case that an entry needs to be referenced but no data from that entry is needed. When you move a cursor using a method like `Next`, `Prev`, `Move`, or `Reset` the cursor "points at" an entry, but at this point nothing may need data from the entry. It would be a waste of time and memory to do the work of creating the cached entry frame as soon as a cursor pointed to an entry.

Entries are Fault Blocks

A soup entry is really a special object called a **fault block**, which is a special class of object composed of two parts. One part is a simple NewtonScript frame often called the cached frame. The other part is a handler which knows how to create and save the cached frame.

Most of the OS treats a fault block as if it were a simple frame. When a slot is accessed the OS checks to see if the cached frame exists, and if so the slot is simply looked up in that frame. When a slot is added or changed it works the same way, if the cached frame exists the slot is set in that frame.

When the cached entry doesn't exist, a fault occurs and a message is sent to the handler, which creates or *faults in* the cached frame. Once this is complete and the cached frame exists, slot access continues as described above.

In the case of soup entries, the cached frame is the cached entry frame. It's what the rest of your code reads and modifies when you work with soup entries. When a cursor accesses a new entry, a fault block gets created with a handler that knows how to retrieve the data, but the cached frame does not yet exist. It's not until the first time some code looks at the slots in the entry that the cached entry frame gets created. This explains why validTests are potentially slow. The `validTest` function typically looks at slots in the soup

entry, which reads in the entry data from the user store (deserializing the entry's elements) and creates the cached frame (allocating space from the NewtonScript heap.)

> *The overhead of reading in the soup entry is incurred the very first time a slot in the entry is touched.*

There are functions which work with a entry that do not cause the entry to be faulted in. Most of the global functions that work with entries, such as `EntryUniqueID`, `EntryModTime`, or `EntryRemoveFromSoupXmit` don't cause the cached entry frame to be created.

You can read more about entry caching in the Newton Programmer's Guide section on data storage. Fault blocks (and how to create your own versions) are covered in the section on Mock Entries.

It's now easy to understand how some of the other entry management functions work. It is the cached entry frame that holds all modifications made to a soup entry. These modifications don't become permanent until `EntryChangeXmit` is called. `EntryChangeXmit` causes the fault block's handler to write the data in the cached frame to the user store. `IsSoupEntry` checks to see if the object passed is a fault block for soups, rather than a regular frame.

`EntryUndoChanges` works by throwing away the cached entry frame. The next time someone needs to access data in the entry, the entry handler faults again, creating a new cached frame from the (unmodified) data on the user store.

`AddXmit` and the other entry adding methods are an unusual case. When you call an add method, you pass a regular NewtonScript frame that will be turned into a soup entry. The add method does the necessary work to write the data in the frame to the user store, but it then has to somehow turn that frame into a fault block, so that any references to that frame now refer to the soup entry (the fault block.)

These add methods are very unusual functions which effectively modify one of their arguments. Note the distinction: lots of functions modify the contents of a passed frame, array, string, or other binary object, but all function calls in NewtonScript are call-by-value, so actually modifying an argument isn't otherwise possible. `ReplaceObject` is what actually accomplishes this trick. The add methods uses `ReplaceObject` to change any and all references to the passed frame into references to the fault block.

Entry Management

With the 2.0 release of the Newton OS, some new functions were added to allow you to more closely manage the entry's cached frame.

`AddFlushedXmit` does exactly the same thing that `AddXmit` does with respect to creating the fault block, writing the data to the store, updating indexes, etc. The difference is the fault block that's created will not have its cached frame set. There are also flushed versions of the add methods for union soups.

As part of adding a soup entry with `AddXmit`, the OS does two important things. It walks the frame being added, writing the data to the store, and it uses `EnsureInternal` to create the cached frame, so that the requirement that data in soup entries is safe from card ejection is met. This `EnsureInternal` step can be expensive. Since `AddFlushedXmit` doesn't create the cached frame, it can skip the `EnsureInternal` step. The process of reading in an entry from the user store (*faulting it in*) guarantees the cached frame will be safe from card ejection.

`AddFlushedXmit` saves both time and memory, and can be a real win if you know that nothing is going to cause the entry to be faulted in right away. On the other hand, if things are set up so that an entry is used or modified right after it's created, `AddFlushedXmit` doesn't help, since the first access will cause the cached frame to be created. You should experiment with `AddXmit` and `AddFlushedXmit` when creating entries. If you're building up a soup from static data, `AddFlushedXmit` may be a lot faster. If you're creating entries one at a time as the user enters data, the difference may not be as noticeable.

`EntryFlushXmit` and `EntryChangeXmit` are similar in the same way. `EntryChangeXmit` does an `EnsureInternal` on the cached frame, then writes the data to the soup. `EntryFlushXmit` skips the `EnsureInternal` step, writes the data to the soup, then discards the cached frame so that the entry must be read in again next time it's needed.

Use `EntryFlushXmit` if you need to keep a reference to the entry around for some reason, but have no plans to touch data in the entry for a while. The big win comes from avoiding `EnsureInternal`. Keep in mind that `EntryFlushXmit` doesn't actually reclaim the NewtonScript heap space used by the cached entry frame, it just removes the only reference to it. The garbage collector still has to do cleanup, the same cleanup that it would normally do if you no longer referenced the entry itself (that is, the fault block.) Making sure you

don't keep references to unneeded entries or cursors may pay off more than trying to be tricky with `EntryFlushXmit`.

Other Caches

You may have noticed that each time you call `GetStores`, you get an array containing the same objects, one per store. That is, the OS doesn't create new store objects each time you make this call, but rather appears to return an existing object. This is hardly surprising; there are a lot of objects that exist even when your application isn't using them, like global variables, other applications, or the root view.

However, you may not have noticed that each time you call `store:GetSoup` or `GetUnionSoup` to get a particular soup, you also get the same object. Once again, the OS appears to return an existing object rather than create one each time you call the function. This also isn't very surprising, because clearly there is only one instance of a soup on a given store, and naturally you expect to get that one each time.

When you perform a `Query`, you get a cursor object. In this case, if you call the `Query` method a second time with the same arguments, you don't get the same object, but rather a new different cursor. Again, this makes perfect sense, each cursor needs to have its own "pointer" into the soup, and if you always got the same cursor for the same query, there would be no way to have them reference different entries.

When you call `cursor:Entry`, you get a soup entry. If you navigate some other cursor to the same place in the soup, and call `cursor2:Entry`, you get the identical entry object—the same fault block. This makes sense too, since clearly there's only one copy of the entry on the store. The OS gives the illusion that there is some entry object in memory, just waiting for someone to ask for it, and which will be given to anyone who asks.

As an aside, this is occasionally a problem for programming. If one application modifies an entry's cached frame, any other application that happens to be using that entry is suddenly working with a modified object, even though `EntryChangeXmit` or `EntryFlushXmit` wasn't called and no notification has yet been sent. The OS designers had to make a tradeoff between living with this behavior and the alternative, which would be to give each application a separate version of the entry and add a more complex database-like locking scheme to prevent multiple applications from modifying the same entry at the same time.

In a handheld single user device, it's unlikely that two applications will need to be modifying the same data at the same time, and so you can adopt a programming technique to avoid the problem. The technique is to make sure your application calls `EntryChangeXmit` or `EntryFlushXmit` relatively soon after modifying an entry. This is a good idea anyway, since a reset between when you change an entry and when you save it back to the soup would cause data loss. Built-in applications and applications based on the NewtApp framework typically save changes within a few seconds of modifying an entry's cached frame and when closing an editor.

Back to the mystery of identical objects. Clearly there isn't enough memory in the NewtonScript heap for the OS to really keep all the stores, soups, cursors, and entries around just waiting for someone to need them. Something special is going on behind the curtain to maintain this illusion.

The OS maintains independent lists of stores, union soups, soups, cursors, and entries that are in use. When someone asks for one of these objects, the OS first looks in its list to see if the needed object is present, and if it is, it returns that object. If the needed object isn't there, a new object of the proper type is created, tucked away in the list, and returned. These lists, or caches, are NewtonScript arrays.

The caches are more than just standard arrays, however. If they were normal arrays, then the references to the objects in the array would be enough to keep the objects themselves from being garbage collected. The OS would have to know when no other application needed the soup, store, cursor, or whatever and explicitly remove the reference from the cache.

The caches are implemented using a special NewtonScript object called a **weak array**. Weak arrays are just like normal arrays in most respects, with one important distinction. During garbage collection, if the only references to an object are in weak arrays, then that object is disposed of and the corresponding elements of the weak arrays are set to `NIL`. Weak arrays are documented in the Newton Programmer's Reference, and can be created using the global function `NewWeakArray`.

Here's a quick demonstration, from the NTK inspector. Note the contents of the array weenie change after garbage collection. The otherwise unreferenced string `"Atlas"` disappears, but the location string remains.

```
weenie := NewWeakArray(2);
weenie[0] := "Atlas";
```

```
weenie[1] := GetUserConfig('location).name;
weenie
#4418B25  [_weakarray: "Atlas", "Cupertino"]

GC();
weenie
#44146D1  [_weakarray: NIL, "Cupertino"]
```

Where the OS Caches Objects

Stores are not cached in a weak array, there is a real array of store objects maintained by the OS. The GetStores function simply returns this array. When a memory card is inserted a new store object is created and stored in the array. When the card is removed the stores in unmounted and the object removed from the array.

Inside each store object is a slot called 'soups which contains a weak array of soups in use on that store. When the OS needs to use a soup on the store, a soup object is created and stored in this weak array in an empty position, or in a newly created element if there are no empty positions. Since it's a weak array, there is no worrying about when the soup is no longer needed, garbage collection takes care of the cleanup. The store methods GetSoupNames and GetSoup should be used to access soups in a supported way.

The OS maintains a separate weak array of union soup objects. There is no direct access available to this weak array from scripting. Again, garbage collection takes care of the cleanup.

Each union soup maintains a list of member soups in a slot called 'soupList. This is not a weak array, since the member soups are a finite set and will be needed for as long as the union soup is needed. Note that this list may not contain a soup for each store, since member soups are not created until needed. The union soup method GetSoupList should be used to access this array in a supported way.

Each soup or unionSoup maintains a weak array of cursors that use that soup in a slot called 'cursors. This cache isn't necessary for implementing the Query method, but it's part of how the OS manages to keep cursors up to date when the soup contents change. Again, since the cursors are kept in a weak array, garbage collection takes care of the cleanup. The soup method Query should be used to get a cursor in a supported way.

Soups maintain a reference to the store which contains them, in a slot called 'storeObj, for use in various soup methods such as RemoveFromStore. The supported way to get at this is the soup method GetStore.

Each soup (but not union soup) also maintains a weak array of entry frames from that soup that are currently in use, in a slot called 'cache. This cache is used to ensure that different cursors or different applications all read and write to the same cached entry frame. The only supported way to get a soup entry is via a cursor.

It's harder to tell what's in a cursor, since the implementation is all done in a C++ class and NewtonScript slots aren't used, but each cursor holds a reference to the soup or union soup which it's searching.

It's also harder to tell what's in an entry fault block, since that's also implemented in a C++ class and doesn't use NewtonScript structures,. However, each fault block also maintains a reference to the soup object that the entry is contained in. That soup together with indexing information about the entry is sufficient to allow the cursor to locate the real data when an entry needs to be faulted in. That same data allows the EntrySoup and EntryStore functions to be easily implemented.

> *Don't write production code that uses undocumented slots in stores, soup, union soup, or cursor objects.*

As you can see, there is a lot of cross-referencing going on with the data storage model. All this cross referencing means that lots of data can be kept in memory by just a single reference. By forgetting to clean up a single reference to a single entry or cursor, you can force the soup and unionSoup objects to remain in memory.

If you've ever used TrueSize to try to get the space used by a given entry's frame, you'd have been surprised. It typically returns results that are much larger than expected. A quick test I did showed my card in the "Names" soup takes over 110K! Clearly that's not right. All this cross-referencing explains why.

```
e := GetUnionSoup("Names"):Query(
      {indexpath: 'sorton,
      startKey: "Ebert Bob"}
   ):Entry();
TrueSize(e, nil);
objects             431           121254
...
```

TrueSize follows references and knows how to look inside some kinds of C++ objects, like entry fault blocks and cursors, for contained references. This means that calling TrueSize on a soup entry actually counts the size of everything soup or store related that's currently in memory! The links are followed from the entry fault

block to the soup, to the store to other soups on the store and from there to cursors and entries for those otherwise unrelated soups. If you want to know how big a cached entry frame is for a particular entry, just clone it before passing it to TrueSize. Note that the size of the cached frame in the NewtonScript heap is different from the size of the entry on the store. The global function EntrySize will tell you how much store space an entry requires.

```
TrueSize(Clone(e), nil);
objects             26              1002
...

EntrySize(e);
#6B4      429
```

You can make use of the various cross-reference lists to help track down unneeded references. By forcing a garbage collection then looking in the various lists you can easily tell if something in your application is referencing a soup, cursor, or entry that it shouldn't, because the item will appear in a list where you don't expect it to.

To easily track down where an unneeded reference exists, you can use the other feature of TrueSize, which is searching (nearly) everywhere for an object. If I wanted to find out what was hanging on to my names soup entry, for example:

```
TrueSize(nil, e);
...
  person                      undo[0][0].receiver._proto
                              .realData.faultSoup.storeObj
                              .soups[4].cache[2]
  person                      vars.e
```

This tells me that there's two places holding a reference to this entry, one is in a global variable e, which I expect since I created it. The other is in some weird place that I've never heard of before, but appears to be in one of the caches (in this case a weak array) in a soup that's reference by a store that happens to be referenced from some other soup needed for some undo action. (See, those cross-references are pervasive!)

My first thought on seeing this was that I should have forced a garbage collection first to get rid of the reference in the cache, which shows that even experienced programmers can have wrong thoughts. Forcing a GC wouldn't clear out the reference in the cache weak array, because the global variable has a "strong" reference to the entry. To really get rid of it, I'd clear out the global variable first, then force a garbage collect. Once that's done, there's no way to verify that the entry is really gone using only TrueSize, since there's

no longer anything to pass to that function! To verify that nothing else is holding a reference I'd have to go poke around in those undocumented lists in the stores and soups, or carefully check free memory with `GC` and `Stats`.

Conclusion

Entries in NewtonScript are special objects and there is significant overhead involved in both reading and writing them from the user store. Careful thought while designing your applications will pay off by minimizing the NewtonScript heap space used and the time needed to access your data. You have control over when an entry's cached frame is or is not faulted in, and you can take advantage of this to improve performance.

Knowing how stores, soups, union soups, cursors, and entries relate to each other helps when creating efficient applications, and helps even more when tracking down performance or space problems. If you're ever unsure about how to optimize your application, experiment with different alternatives, measure the space and speed differences, and choose accordingly.