CHAPTER 13

Persistence with AllegroCache

Introduction

Persistence is the first Lisp feature we've encountered which is not part of Common Lisp. Its goal is to make storage unobtrusive for the programmer by—in some sense—remembering "forever" all instances of certain CLOS classes. A typical side-effect of persistence is data sharing between different Lisp images. When people list features beyond the ANSI standard which they'd like to see standardized and more widely implemented, persistence is often high on their list.

There's a choice of persistence libraries and each has its pros and cons. We're going to look at one in some detail. I've picked AllegroCache because it's a good example of a proprietary library which ships with a Lisp implementation, in this case Allegro Common Lisp. We can expect it to integrate well with the underlying Lisp system but must realise in advance that any code we write using it won't port to other Lisps.

We can also expect it to integrate well with other Allegro add-ons. For example, *Allegro Prolog* can reason directly over AllegroCache data.

There's a database at the back of AllegroCache and many of the operations you can perform with persistent objects have equivalents in the relational database world. That leads us quite rightly to asking why we shouldn't use a standard SQL database instead. Persistence has the following advantages over SQL:

- Creation of the objects which your program uses and telling the database that these objects have been created become a single operation. Ditto for modifications to existing objects.

  You don't have to keep track of which objects need writing to the database. You don't have to stuff your code with hideous `format` statements expressing these changes as SQL.

- Objects retrieved from the database come back ready for use as CLOS instances. You don't have to build your objects by hand from sequences of slot values.
- You don't have to decide in advance how long your strings are going to be. In fact, as with CLOS objects in general, you don't have to decide *ever* what types your slot values are going to take, nor do these have to be the same from one instance to the next.
- If you want one object to refer to another, just do it as you would in any other CLOS application. There's no equivalent to tables whose sole purpose is to make `join`s work.
- You don't have to escape user-supplied strings. (SQL users should now consult *http://bobby-tables.com/*.)

In short, integrating persistence with the underlying objects makes you, the programmer, more productive by allowing you to get on with thinking about your application instead of the demands of communicating with the database upon which it depends.

Here's a simple example in which I'll create a database, serve it on my network, and consider using it as an authoring tool. We'll cover the following operations in detail later.

```
;; Load AllegroCache and use its exports in the current package.

(require :acache "acache-2.1.11.fasl")
(use-package :db.ac)

;; Define a persistent class.

(defclass sentence ()
  ((text :initarg :text :accessor sentence-text)
   (next :initform nil :accessor sentence-next))
  (:metaclass persistent-class))

;; Create a database and connect to it.

(start-server "~/play/acache" 8707 :if-does-not-exist :create)
(open-network-database "localhost" 8707)

;; Start writing; create some persistent objects...

(let ((first-sentence
       (make-instance 'sentence
                      :text (format nil "Persistence is the first Lisp~@
                                         feature we've encountered which~@
                                         is not part of Common Lisp.")))

      (second-sentence
       (make-instance 'sentence
                      :text "Its goal is to make storage unobtrusive.")))

  (setf (sentence-next first-sentence) second-sentence))
```

```
;; ... and save them to the database.

(commit)
```

And here's one way in which someone elsewhere (my editor, perhaps?) might check on my progress:

```
;; Load AllegroCache as above and connect to my server.

(require :acache "acache-2.1.11.fasl")
(use-package :db.ac)
(open-network-database "gannet.ravenbrook.com" 8707)

;; View all sentences, in some undetermined order.

(let ((sentences nil))
  (doclass (sentence 'sentence)
    (push (sentence-text sentence) sentences))
  sentences)
=>
("Persistence is the first Lisp
feature we've encountered which
is not part of Common Lisp."
 "Its goal is to make storage unobtrusive.")

;; Hmm, OK so far but not quite done.
```

There isn't space here to document every aspect of AllegroCache in full. You can download the AllegroCache reference manual along with Flash presentations, a tutorial, administrator's guide, and more by visiting:

http://franz.com/products/allegrocache/

This chapter and the next will show you how to create and update persistent objects with AllegroCache, work with its transaction model, build complex queries, and administer the underlying database. You'll need to be comfortable with the Common Lisp Object System (Chapter 7) to get the most from them; the material here will be useful for following examples in Chapters 15 and 16 but is not necessary for understanding the rest of the book.

AllegroCache brings us into contact with the libraries that deal with memory management and with multi-threading. Like AllegroCache they're non-standard but unlike it they're ubiquitous. They'll be the subjects of the Chapters 15 and 16.

Loading the Library

While writing this I worked with version 2.1.11 of AllegroCache running on version 8.1 of Allegro CL's free Express Edition on Windows XP. You can expect identical behavior with different editions of Allegro CL and with different OS platforms. Some features of AllegroCache itself may change incompatibly in future versions and the library's documentation makes it very clear which these are; we'll steer away from these here.

AllegroCache ships with the Lisp it runs on, Allegro CL. In the first instance, once you've installed ACL you don't have to download or install anything else in order to use the AllegroCache library.

AllegroCache and Allegro CL have different release cycles. So it's possible that the AllegroCache which you installed along with Allegro CL isn't the most recent version. That shouldn't be a problem for the purposes of this chapter but if you do want to get your hands on the latest and greatest then you'll need to follow these steps.

1. Load patches, by calling `(system:update-allegro)` and following any instructions which appear when it finishes running. You'll have to quit and restart all running Allegro CL images.
2. Call `(system.update:install-allegrocache)` to download the latest version of AllegroCache.

Whether or not you've done this download, the next step is to load AllegroCache into your Lisp session, thus:

```
(require :acache)
```

`Require` is a Common Lisp utility for ensuring that a library has been loaded into the current session. Its first argument names the library you're after. An optional second argument specifies which file or files to load; if this isn't given then it's totally up to the implementation to figure out where to find them. This makes `require` a natural way for implementations to control the loading of their own precompiled libraries but it's not much use for anybody else. See Chapter 19 for more about system definition.

Don't fret if the `require` form above gives you an immediate error ("`To prevent problems with loading incompatible databases, you must load a specific version of AllegroCache.`"). The error message will offer a choice of more specific forms for you to use instead. For experimentation purposes just pick the form with the highest version number; if you're running with AllegroCache "in production" you'll want to pick one version and stick with it.

```
CL-USER(1): (require :acache "acache-2.1.11.fasl")
; Fast loading C:\Program Files\acl81-express\code\acache-2.1.11.fasl
AllegroCache version 2.1.11
;   Fast loading C:\Program Files\acl81-express\code\SAX.001
;;; Installing sax patch, version 1.
;     Fast loading from bundle code\EF-E-ANYNL.fasl.
T
CL-USER(2):
```

The output you'll see depends on precise version and platform and so might not be exactly the same as this. What matters is that it doesn't say `error`.

Packages

All AllegroCache utilities are exported from the DB.ALLEGROCACHE package, nickname DB.AC. We'll assume for this chapter and the next that you're using this package.

```
CL-USER(2): (use-package :db.ac)
T
CL-USER(3):
```

In a real application it's often a matter of taste whether you "use" packages or explicitly qualify the symbols you need.

Connecting to the Database

AllegroCache stores your data in a (proprietary) database and our next task is to connect to that. There are two modes of use:

Standalone
: The database lives on your local filesystem and you can only have one connection open to it at a time; this mode is the more efficient of the two; "commit" always succeeds. An example of use would be backing store for a web server.

Client/server
: The database serves objects over your network and supports multiple connections; objects can be shared between processes on different machines; there's the possibility of a "commit" failing. We'll discuss network servers in "AllegroCache in Client/server Mode" on page 11 later in this chapter.

To connect to a standalone database, call open-file-database and pass it the location of a directory (this need not exist yet) in which your database files are going to live. This function takes a number of keyword arguments, among which in particular is if-does-not-exist, similar to the corresponding argument to cl:open. Pass the value :create to create a new database if one does not already exist; otherwise you'll get an error. For example:

```
CL-USER(3): (open-file-database "~/play/acache")
Error: database "c:\\home\\nick\\play\\acache" does not exist
[condition type: SIMPLE-ERROR]
Restart actions (select using :continue):
 0: Create the database
 1: Return to Top Level (an "abort" restart).
 2: Unwind to the top-level event-handling loop.
 3: Exit this IDE listener (Listener 1).
 4: Abort entirely from this (lisp) process.

[1] CL-USER(4): :pop
CL-USER(5): :pop(probe-file "~/play/acache")
NIL
CL-USER(6): (open-file-database "~/play/acache" :if-does-not-exist :create)
; Autoloading for DELETE-DIRECTORY-AND-FILES:
; Fast loading from bundle code\fileutil.fasl.
```

```
#<AllegroCache db "c:\\home\\nick\\play\\acache" @ #x212bcd72>
CL-USER(7): (directory "~/play/acache/")
(#P"c:\\home\\nick\\play\\acache\\ac000000.dat"
 #P"c:\\home\\nick\\play\\acache\\acache-params.cl" ...)
CL-USER(8): *allegrocache*
#<AllegroCache db "c:\\home\\nick\\play\\acache" @ #x212bcd72>
CL-USER(9):
```

Note (lines 5 and 7) that the second call to `open-file-database` has created and populated the directory *~/play/acache* which didn't exist previously.

The database is represented by a Lisp object which is returned from the call to `open-file-database` (line 6) and also stored into the value of the variable `*allegrocache*` (line 8). Several AllegroCache utilities accept a `:db` keyword argument for specifying which database they should refer to, and this parameter's value defaults to the current value of `*allegrocache*`. If you ever need to connect to more than one database at a time you'll have to either rebind `*allegrocache*` or pass around explicit values of `:db` as needed. Otherwise you can simply trust that the last database you opened will be the one that's used.

Note this pattern: we have a function which sets the object it's about to return into a global variable; utility functions accept a keyword argument which defaults to the current value of that variable. It's a useful approach and one you'll meet elsewhere.

As an example, to close the default database connection:

```
CL-USER(9): (close-database)
#<AllegroCache db "c:\\home\\nick\\play\\acache" -closed- @ #x214f10e2>
CL-USER(10):
```

—note how the database's printed representation has now changed—and to close any other connection either

```
(close-database :db *my-database*)
```

or

```
(let ((*allegrocache* *my-database*))
  (close-database))
```

As another example, here's a utility we might write for use on days when writer's block gets the better of us. In SQL we'd use the "delete" command...

```
(defun flush-all-sentences (&key (connection *allegrocache*))
  (doclass (sentence 'sentence :db connection)
     (delete-instance sentence))
  (commit :db connection))
```

Later on we'll find the following macro handy. It takes a list of variables, a form which should return a database connection, and a body. Each of the variables will be bound to its own connection and then body will be run. The connections are guaranteed to be closed on exit.

With a standalone database only one connection is possible at a time and in this case you should only supply one `connection-var`. But the macro will be more flexible if its syntax is designed to accept more than one connection, and we'll be able to use it as-is when we're experimenting with network servers later in the chapter.

```
(defmacro with-connections (connection-vars opening-form &body body)
  `(let ,(loop for connection in connection-vars collect
               `(,connection ,opening-form))
     (unwind-protect
         (progn ,@body)
      (dolist (connection (list ,@connection-vars))
        (close-database :db connection)))))
```

Exercise

Use `with-connections` to open and close some database connection. Macroexpand your call and check that you're happy with how the macro works. If the body were empty, how would you know whether the "open" succeeded?

Exercise

Write a related macro (called `with-connections*` perhaps?) in which each connection has its own `opening-form`. Macroexpand some test cases to check you're happy with it.

Exercise

Do you need to be running ACL to work on the last two exercises or will any Common Lisp implementation do? Why / why not?

Working with Persistent Objects

Here's what you have to do to make your data persistent: connect to the database, define a persistent class, make instances of that class and populate their slots, and `commit` your changes. Let's go through these steps in a little detail.

1. You've already connected to the database (see above).
2. To make your classes persistent, use the metaclass `persistent-class`:

```
   (defclass sentence ()
     ((text :initarg :text :accessor sentence-text)
      (next :initform nil :accessor sentence-next))
     (:metaclass persistent-class))
```

You can define your persistent classes either before or after opening a connection. In a real application it's likely you'll define the classes first (when the application is loaded) and then connect on startup.

Defining a persistent class corresponds to SQL's `create table`.

3. You can now start making persistent instances:

```
(make-instance 'sentence
               :text "You can now start making persistent instances:")
```

   This is the equivalent to SQL's `insert` statement.

4. Whenever you create a persistent object, the new object and its slot values are queued for writing to the database which is the current value of `*allegrocache*`. To make the writes happen, call `commit`:

```
CL-USER(20): (let ((text "To make the writes happen, call commit:"))
               (make-instance 'sentence :text text))
#<SENTENCE oid: 1011, ver 5, trans: NIL,  modified @ #x21235002>
CL-USER(21): (commit)
T
CL-USER(22):
```

AllegroCache can only store certain types of Lisp object in the database, so you're limited (but not much) as to what values you can set into a persistent object's slots:

Objects
: You can store references to other persistent objects in the same database. (You can also store non-persistent objects but you have to tell AllegroCache how to do this first: see the next chapter for details.)

Simple types
: Symbols, integers, floats, characters.

Sequences
: Strings. Lists (including dotted lists) and vectors [*] where the sequence's members are of types listed here. Also allowed are simple vectors of `(unsigned-byte 8)`.

Tables
: You can't store hash tables. AllegroCache provides similar behavior with "maps"; we'll discuss these in the next chapter.

Making changes to a persistent object's slots corresponds to SQL's `update` statement.

```
;; Modifying a persistent object's slot
(setf (sentence-next no-hash-tables) similar-behavior-with-maps)
```

If you change the values themselves then the new values will be queued automatically for writing to the database and that's all there is to it. But if a slot contains a value which

[*] To be precise: simple vectors of type `t`.

can be modified (a cons, list, vector, string or non-persistent object) then it doesn't matter what changes you like to the internals of that value: AllegroCache won't know that the database needs updating. You'll have to handhold it by calling `mark-instance-modified`.

Handy debugging aid: in the following example see how the object's printed representation changes (a) when AllegroCache believes that it needs writing—equivalently the object appears to have been modified since it was last written—and (b) to reflect the *transaction number* at which the object was last written: if this number hasn't gone up then the database hasn't seen the change.

```
;; Any new persistent object is automatically queued for writing.

(setf goal
      (make-instance 'sentence
                     :text "It's goal is to make storage unobtrusive?"))
=>
 #<SENTENCE oid: 1013, ver 5, trans: NIL,  modified @ #x212b3d72>

(progn (commit) goal)
=>
#<SENTENCE oid: 1013, ver 5, trans: 13,  not modified @ #x212b3d72>

;; Whoa! Apostrophe catastrophe! We replace the text slot with a different
;; string and our change will be queued automatically.

(setf (sentence-text goal) "Its goal is to make storage unobtrusive?")
(progn (commit) goal)
=>
#<SENTENCE oid: 1013, ver 5, trans: 14,  not modified @ #x212b3d72>

;; Fix punctuation. This time we modify the text slot (so it's still the
;; same Lisp object) and the change is not queued: the commit does nothing.

(let* ((text (sentence-text goal))
       (length (length text)))
  (setf (schar text (1- length)) #\.))
(progn (commit) goal)
=>
#<SENTENCE oid: 1013, ver 5, trans: 14,  not modified @ #x212b3d72>

;; Manual request for updating. This time (commit) will write our latest
;; change.

(mark-instance-modified goal)
=>
#<SENTENCE oid: 1013, ver 5, trans: 14,  modified @ #x212b3d72>

(progn (commit) goal)
=>
#<SENTENCE oid: 1013, ver 5, trans: 16,  not modified @ #x212b3d72>
```

Simple queries

We'll deal with this subject properly in the next chapter. For now here's one way of getting our data back. It's the macro doclass which has the following syntax:

```
(doclass (var class &key db) &body body)
```

The macro executes its body with var bound to successive instances of class (in an undetermined order); it's like using an unconstrained select in SQL to retrieve every instance of a class, except that SQL can never build your objects for you.

```
(let ((sentences nil))
  (doclass (sentence 'sentence)
    (push sentence sentences))
  sentences)
=>
(#<SENTENCE oid: 19011, ver 6, trans: 96,  not modified @ #x21256b5a>
 #<SENTENCE oid: 19010, ver 6, trans: 96,  not modified @ #x21256b42>)
```

Doclass returns nil; you can use (return ...) to terminate the loop early and return more interesting values.

The related macro doclass* also descends through subclasses.

Transactions

AllegroCache has a transaction model which relational database users should recognize. As already noted, none of your changes will be written to the database until you call commit. Alternatively, you can call rollback and all your changes (since the last commit or rollback) will be abandoned; your objects will now reflect the current state of the database. Any persistent objects freshly created since the last commit or rollback will continue to exist but they have been invalidated: if you try to read or write their slots you'll get an error.

```
(let* ((original-text "AllegroCache has a transaction model.")
       (sentence (make-instance 'sentence :text original-text)))
  (commit)
  (setf (sentence-text sentence) "Database users should recognize it.")
  (rollback)
  (sentence-text sentence))
=>
"AllegroCache has a transaction model."

(let* ((original-text "If you read or write their slots...")
       (sentence (make-instance 'sentence :text original-text)))
  (rollback)
  (setf (sentence-text sentence) "... you'll get an error."))
=>
Error: attempt to access a deleted object: #<SENTENCE oid: 1015, ver 5,
trans: NIL, deleted @ #x21463cda>
```

The predicate `deleted-object-p` will return true for an object which has been deleted, either explicitly by `delete-instance` or implicitly by a `rollback` before it was ever committed.

If the database is networked, use `rollback` to grab other clients' changes and update your local object cache to the current state of the database. Think of the `svn up` command in Subversion.

AllegroCache in Client/server Mode

Setting up AllegroCache to run as a server over your network is straightforward enough. The function to do it is `start-server`; this takes as arguments a location (as for `open-file-database`), a port on which to serve requests, and a number of keywords among which again is `if-does-not-exist`. For example:

```
(start-server "~/play/acache" 8707 :if-does-not-exist :create)
```

This function returns an object representing the server. To close the server, call `stop-server` and pass it that object.

Once the server is running, any number of clients can connect to it by calling `open-network-database`:

```
(open-network-database "gannet.ravenbrook.com" 8707)
=>
#<AllegroCache db "port 2345 to gannet.ravenbrook.com:8707" @ #x214d9e6a>
```

If you're running the *Express Edition* of ACL, don't forget that there's a limit of three open connections. If you're experimenting and opening lots of connections, you should close them as you go.

To close a connection, as with standalone connections, call `close-database`. To check whether a connection of either type is open, call `database-open-p`.

Calling `close-database` will drop any outstanding changes which you hadn't yet written to the database. You might want to call `commit` first.

Atomicity and Isolation

An AllegroCache transaction is *atomic*. That is, either it succeeds and all its changes are written to the database, or it fails and none of the changes are written. A typical reason for a write failing is that some other client has modified one of your objects behind your back. We'll take a proper look at this in the next section.

An AllegroCache connection is *isolated* from other connections. That is, you don't see other clients' changes until you're ready to do so and they don't see yours until you want them to.

- When you open any database connection, your local cache will be populated with the current state of the database.
- To update your cache so it reflects the current state of the database, call `rollback`. Any uncommitted changes will be dropped and the objects concerned rewound to their state at the last `rollback` or `commit`.
- To update the database so it reflects your local cache, call `commit`. As well as performing the write, this operation updates your cache with any changes committed by other connections since your last `rollback` or `commit`.

The following example employs the `with-connections` macro which we defined earlier in the chapter. It allows us to use a single Lisp session to simulate multiple sessions—possibly running on different machines—each with their own database connection.

```
(with-connections (connection-1 connection-2)
    (open-network-database "localhost" 8707)

  ;; Use the utility we defined earlier to flush all sentences out of the
  ;; database (and commit the change).

  (flush-all-sentences :connection connection-1)

  ;; Make sure the other connection is in the same state.

  (rollback :db connection-2)

  ;; First connection inserts a sentence and commits the change.

  (let ((*allegrocache* connection-1))
    (make-instance 'sentence
                   :text "An AllegroCache transaction is atomic."))
  (commit :db connection-1)

  ;; Second connection inserts and commits its own sentence.

  (let ((*allegrocache* connection-2))
    (make-instance 'sentence
                   :text "An AllegroCache connection is isolated."))
  (commit :db connection-2)

  ;; Second connection now sees both sentences (because commit updates your
  ;; cache).

  (let ((sentences nil))
    (doclass (sentence 'sentence :db connection-2)
      (push (sentence-text sentence) sentences))
    sentences))
=>
```

```
("An AllegroCache connection is isolated."
 "An AllegroCache transaction is atomic.")
```

Failure to Commit

Two users can't modify the same object at the same time. More precisely: if you `commit` a change to something then no other client can `commit` changes to the same object until they've picked up your change:

```
(with-connections (connection-1 connection-2)
    (open-network-database "localhost" 8707)
  (flush-all-sentences :connection connection-1)

  ;; Start with both connections seeing one (and the same) sentence:

  (let* ((*allegrocache* connection-1)
         (sentence-1 (make-instance 'sentence
                                    :text "If you commit a change...")))

    (commit :db connection-1)
    (rollback :db connection-2)

    ;; The first connection now updates the sentence and commits
    ;; its change.

    (setf (sentence-text sentence-1) "no other client can do likewise...")
    (commit :db connection-1)

    ;; The second connection has a own private copy of the sentence.
    ;; This copy does not reflect the update made by connection-1. (Why?)

    (doclass (sentence 'sentence :db connection-2)

      ;; The following change hasn't been committed and is therefore
      ;; local to connection-2.

      (setf (sentence-text sentence)
            "until they've picked up your change."))

    ;; This attempt to commit must fail.
    ;; The unwind-protect in with-connections ensures that the
    ;; connections are closed cleanly.

    (commit :db connection-2)))
=>
Error: Cannot commit because object #<SENTENCE oid: 4010, ver 5,
trans: 27, modified> was updated in another transaction
```

Exercise

Modify this example so that it succeeds.

Exercise

Trace `commit` to verify which call signaled the error. Where do the two "extra" calls come from? How can you check this? Why are these calls made?