CL-STORE: CL Serialization Package

Table of Contents

1 Introduction

CL-STORE is a portable serialization package for Common Lisp which allows the reading and writing of most objects found in Common Lisp resolving any circularities which it detects. It is intended to serve the same purpose as Java's ObjectOutput and ObjectInputStream, although it's somewhat more extensible.

The CL-STORE Home Page is at http://common-lisp.net/project/cl-store where one can find details about mailing lists, cvs repositories and various releases.

This documentation is for CL-STORE version 0.6 .

Enjoy Sean.

1.1 Example

```
(defclass myclass () ((a :accessor a :initarg :a)))
(cl-store:store (make-instance 'myclass :a 3) "/tmp/test.out")

(a (cl-store:restore "/tmp/test.out"))
```

1.2 Supported Objects

- Numbers (floats, integers, complex, NaN floats, rationals)
- Strings (Supports Unicode Strings)
- Characters
- Symbols
- Packages
- HashTables
- Lists
- Vectors And Arrays
- Instances of CLOS Classes
- CLOS Classes
- Structure Instances
- Structure Definitions (CMUCL and SBCL only)
- Functions (stores the function name)
- Generic Functions (stores generic-function-name)

1.3 Supported Implementations

- SBCL
- CMUCL
- CLISP
- Lispworks
- Allegro CL
- OpenMCL
- ECL

2 Getting Started

CL-STORE uses asdf as it's system definition tool and is required whenever you load the package. You will need to download it, or if you have sbcl `(require 'asdf)`

2.1 Downloading

- ASDF-INSTALL CL-STORE is available through asdf-install. If you are new to Common Lisp this is the suggested download method. With asdf-install loaded run `(asdf-install:install :cl-store)` This will download and install the package for you. Asdf-install will try to verify that the package signature is correct and that you trust the author. If the key is not found or the trust level is not sufficient a continuable error will be signalled. You can choose to ignore the error and continue to install the package. See the documentation of asdf-install for more details.
- DOWNLOAD

  The latest cl-store release will always be available from cl.net. Download and untar in an appropriate directory then symlink '`cl-store.asd`' to a directory on `asdf:*central-registry*` (see the documentation for asdf for details about setting up asdf).
- CVS

  If you feel the need to be on the bleeding edge you can use anonymous CVS access, see the Home Page for more details for accessing the archive. Once downloaded follow the symlink instructions above.

2.2 Installing

Once downloaded and symlinked you can load CL-STORE at anytime using `(asdf:oos 'asdf:load-op :cl-store)` This will compile CL-STORE the first time it is loaded.

2.3 Testing

Once installed you can run the regression tests for it. The tests depend on the Regression Tests asdf package which is asdf-installable. The tests can be run be executing `(asdf:oos 'asdf:test-op :cl-store)`

If any tests fail please send a message to one of the Mailing Lists.

3 API

3.1 Variables

`*nuke-existing-classes*` *Default NIL* [Variable]
Determines wether or not to override existing classes when restoring a CLOS Class. If `*nuke-existing-classes*` is not NIL the current definition will be overridden.

`*store-class-superclasses*` *Default NIL* [Variable]
If `*store-class-superclasses*` is not NIL when storing a CLOS Class all superclasses will be stored.

`*store-class-slots*` *Default T* [Variable]
If `*store-class-slots*` is NIL slots which are class allocated will not be serialized when storing objects.

`*nuke-existing-packages*` *Default NIL* [Variable]
If `*nuke-existing-packages*` is non-nil then packages which already exist will be deleted when restoring packages.

`*store-used-packages*` *Default NIL* [Variable]
The variable determines how packages on a package use list will be serialized. If non-nil the the package will be fully serialized, otherwise only the name will be stored.

`*store-hash-size*` *Default 50* [Variable]
The default size of the hash-table created to keep track of objects which have already been stored. By binding the variable to a suitable value you can avoid the consing involved by rehashing hash-tables.

`*restore-hash-size*` *Default 50* [Variable]
The default size of the hash-table created to keep track of objects which have already been restored. By binding the variable to a suitable value you can avoid the consing involved by rehashing hash-tables.

`*check-for-circs*` *Default t* [Variable]
Binding this variable to nil when storing or restoring an object inhibits all checks for circularities which gives a severe boost to performance. The downside of this is that no restored objects will be eq and attempting to store circular objects will hang. The speed improvements are definitely worth it if you know that there will be no circularities or shared references in your data (eg spam-filter hash-tables).

`*default-backend*` [Variable]
The backend that will be used by default.

3.2 Functions

`store` *object place* **&optional** (*backend* ***default-backend***) [Generic]
Stores *object* into *place* using *backend*. *Place* must be either a `stream` or a `pathname-designator`. All conditions signalled from store can be handled by catching `store-error`. If the `store-error` is not handled the causing error will be signalled.

`restore` *place* **&optional** (*backend* ***default-backend***) [Generic]
Restores an object serialized using `store` from *place* using *backend*. *Place* must be either a `stream` or a `pathname-designator`. Restore is setffable eg.

```
(store 0 "/tmp/counter")
(incf (restore "/tmp/counter"))
```

All conditions signalled from restore can be handled by catching `restore-error`. If the `restore-error` is not handled the causing error will be signalled.

`find-backend` *name* **&optional** (*errorp* **nil**) [Function]
Return backup called *name*. If there is no such backend NIL is returned if *errorp* is false, otherwise an error is signalled.

`caused-by` *cl-store-error* [Function]
Returns the `condition` which caused `cl-store-error` to be signalled.

3.3 Macros

`with-backend` *backend* **&body** *body* [Macro]
Execute *body* with `*default-backend*` bound to the backend designated by *backend*.

3.4 Conditions

`cl-store-error` [Condition]
Class Precedence: `condition`

Root CL-STORE Condition all errors occuring while storing or restoring can be handled by catching `cl-store-error`

`store-error` [Condition]
Class Precedence: `cl-store-error`

A `store-error` will be signalled when an error occurs within `store` or `multiple-value-store`. The causing error can be obtained using `(caused-by condition)`

`restore-error` [Condition]
Class Precedence: `cl-store-error`

A `restore-error` will be signalled when an error occurs within `restore`. The causing error can be obtained using `(caused-by condition)`

4 Customizing

4.1 About Customizing

Each backend in CL-STORE can be customized to store various values in a custom manner. By using the `defstore-<backend-name>` and `defrestore-<backend-name>` macros you can define your own methods for storing various objects. This may require a marginal understanding of the backend you wish to extend.

eg.

```
(in-package :cl-user)

(use-package :cl-store)

(setf *default-backend* (find-backend 'cl-store))

;; Create the custom class
(defclass random-obj () ((a :accessor a :initarg :a)))

;; Register random object. This is specific to the
;; cl-store-backend.
(defvar *random-obj-code* (register-code 110 'random-obj))

;; Create a custom storing method for random-obj
;; outputting the code previously registered.
(defstore-cl-store (obj random-obj stream)
  (output-type-code *random-obj-code* stream)
  (store-object (a obj) stream))

;; Define a restoring method.
(defrestore-cl-store (random-obj stream)
  (random (restore-object stream)))

;; Test it out.
(store (make-instance 'random-obj :a 10) "/tmp/random")

(restore "/tmp/random")
=> ; some number from 0 to 9
```

If you need to get fancier take a look at the macroexpansion of the customizing macros.

4.2 Customizing API

This API is primarily concerned with the cl-store-backend although other backends will be similar in structure.

4.2.1 Functions

`register-code` *code name* **&optional** (*errorp* **t**) [Function]
Registers *name* under the code *code* into the cl-store-backend. The backend will use this mapping when restoring values. Will signal an error if code is already registered and *errorp* is not NIL. Currently codes 1 through 35 are in use.

`output-type-code` *type-code stream* [Function]
Writes *type-code* into *stream*. This must be done when writing out objects so that the type of the object can be identified on deserialization.

`store-32-bit` *integer stream* [Function]
Outputs the the low 32 bits from *integer* into *stream*.

`read-32-bit` *stream* [Function]
Reads a 32-bit integer from *stream*.

`store-object` *object place* [Generic]
Stores *object* into *place*. This should be used inside `defstore-cl-store` to output parts of objects. `store` should not be used.

`restore-object` *place* [Generic]
Restore an object, written out using `store-object` from *place*.

`get-slot-details` *slot-definition* [Generic]
Generic function which returns a list of slots details which can be used as an argument to `ensure-class`. Currently it is only specialized on slot-definition

`serializable-slots` *object* [Generic]
Method which returns a list of slot-definition objects which will be serialized for *object*. The default is to call `serializable-slots-using-class`.

`serializable-slots-using-class` *object class* [Generic]
Returns a list of slot-definition objects which will be serialized for object and class. Example. When serializing cl-sql objects to disk or to another lisp session the view-database slot should not be serialized. Instead of specializing serializable-slots for each view-class created you can do this.

```
(defmethod serializable-slots-using-class
    ((object t) (class clsql-sys::standard-db-class))
  (delete 'clsql-sys::view-database (call-next-method)
          :key 'slot-definition-name))
```

4.2.2 Macros

`defstore-cl-store` (*var type stream* **&key** *qualifier*) **&body** *body* [Macro]

Create a custom storing mechanism for *type* which must be a legal Class Name. *Body* will be called when an object of class *type* is stored using `store-object` with *var* bound to the object to be stored and *stream* bound to the stream to output to. If *qualifier* is given it must be a legal qualifier to `defmethod`. Example.

```
(defstore-cl-store (obj ratio stream)
   (output-type-code +ratio-code+ stream)
   (store-object (numerator obj) stream)
   (store-object (denominator obj) stream))
```

`defrestore-cl-store` (*type stream*) **&body** *body* [Macro]

Create a custom restoring mechanism for the *type* registered using `register-code`.*Body* will be executed with *stream* being the input stream to restore an object from.

Example.

```
(defrestore-cl-store (ratio stream)
  (/ (restore-object stream)
     (restore-object stream)))
```

`resolving-object` (*var create*) **&body** *body* [Macro]

Executes *body* resolving circularities detected in *object*. Resolving-object works by creating a closure, containing code to set a particular place in *object*, which is then pushed onto a list. Once the object has been fully restored all functions on this list are called and the circularities are resolved. Example.

```
(defrestore-cl-store (cons stream)
  (resolving-object (object (cons nil nil))
    (setting (car object) (restore-object stream))
    (setting (cdr object) (restore-object stream))))
```

`setting` *place get* [Macro]

This macro can only be used inside `resolving-object`. It sets the value designated by *place* to *get* for the object that is being resolved.

Example.

```
(defrestore-cl-store (simple-vector stream)
  (let* ((size (restore-object stream))
         (res (make-array size)))
    (resolving-object (object res)
     (loop repeat size for i from 0 do
       ;; we need to copy the index so that
       ;; it's value is preserved for after the loop.
       (let ((x i))
         (setting (aref object x) (restore-object stream)))))
    res))
```

`setting-hash` *getting-key getting-value* [Macro]

`setting-hash` works identically to setting although it is used exclusively on hash-tables due to the fact that both the key and the value being restored could be a circular reference. Example.

```
(defrestore-cl-store (hash-table stream)
  (let ((rehash-size (restore-object stream))
        (rehash-threshold (restore-object stream))
        (size (restore-object stream))
        (test (restore-object stream))
        (count (restore-object stream)))
    (let ((hash (make-hash-table :test (symbol-function test)
                                 :rehash-size rehash-size
                                 :rehash-threshold rehash-threshold
                                 :size size)))
      (resolving-object (obj hash)
        (loop repeat count do
              (setting-hash (restore-object stream)
                            (restore-object stream))))
      hash)))
```

5 New Backends

5.1 About

You can define your own backends in cl-store to do custom object I/O. Theoretically one can add a backend that can do socket based communication with any language provided you know the correct format to output objects in. If the framework is not sufficient to add your own backend just drop me a line and we will see what we can do about it.

5.2 The Process

5.2.1 Add the backend

Use `defbackend` to define the new backend choosing the output format, an optional magic number, extra fields for the backend and a backend to extend which defaults to the base backend. eg. (from the cl-store-backend)

```
(defbackend cl-store :magic-number 1347643724
                :stream-type '(unsigned-byte 8)
                :old-magic-numbers (1912923 1886611788 1347635532)
                :extends resolving-backend
                :fields ((restorers :accessor restorers :initform (make-hash-table))))
```

5.2.2 Recognizing Objects.

Decide how to recognize objects on restoration. When restoring objects the backend has a responsibility to return a symbol identifying the `defrestore` method to call by overriding the `get-next-reader` method. In the cl-store backend this is done by keeping a mapping of type codes to symbols. When storing an object the type code is written down the stream first and then the restoring details for that particular object. The `get-next-reader` method is then specialized to read the type code and look up the symbol in a hash-table kept on the backend.

eg. (from the cl-store-backend)

```
(defvar *cl-store-backend* (find-backend 'cl-store))
;; This is a util method to register the code with a symbol
(defun register-code (code name &optional (errorp t))
  (aif (and (gethash code (restorers *cl-store-backend*)) errorp)
       (error "Code ~A is already defined for ~A." code name)
       (setf (gethash code (restorers *cl-store-backend*))
             name))
  code)
;; An example of registering the code 7 with ratio
(defconstant +ratio-code+ (register-code 7 'ratio))

;; Extending the get-next-reader method
(defmethod get-next-reader ((backend cl-store) (stream stream))
  (let ((type-code (read-type-code stream)))
    (or (gethash type-code (restorers backend))
```

```
        (values nil (format nil "Type ~A" type-code)))))

(defstore-cl-store (obj ratio stream)
  (output-type-code +ratio-code+ stream) ;; output the type code
  (store-object (numerator obj) stream)
  (store-object (denominator obj) stream))
```

5.2.3 Extending the Resolving backend

If you are extending the `resolving-backend` you have a couple of extra responsibilities to ensure that circular references are resolved correctly. `Store-referrer` must be extended for your backend to output the referrer code. This must be done as if it were a `defstore` for a referrer. A `defrestore-<backend-name>` must also be defined for the referrer which must return a referrer created with `make-referrer`. Once that is done you can use `resolving-object` and `setting` to resolve circularities in objects.

eg (from the cl-store backend)

```
(defconstant +referrer-code+ (register-code 1 'referrer nil))
(defmethod store-referrer (ref stream (backend cl-store))
  (output-type-code +referrer-code+ stream)
  (store-32-bit ref stream))

(defrestore-cl-store (referrer stream)
  (make-referrer :val (read-32-bit stream nil)))
```

5.3 Example: Simple Pickle Format

As a short example we will define a backend that can handle simple objects using the python pickle format.

5.3.1 Define the backend

```
(in-package :cl-user)
(use-package :cl-store)

(defbackend pickle :stream-type 'character)
```

5.3.2 Recognize Objects

This is just a simple example to be able to handle single strings stored with Python's pickle module.

```
(defvar *pickle-mapping*
  '((#\S . string)))

(defmethod get-next-reader ((backend pickle) (stream stream))
  (let ((type-code (read-char stream)))
    (or (cdr (assoc type-code *pickle-mapping*))
        (values nil (format nil "Type ~A" type-code)))))

(defrestore-pickle (noop stream))

(defstore-pickle (obj string stream)
  (format stream "S'~A'~%p0~%." obj))

(defrestore-pickle (string stream)
  (let ((val (read-line stream)))
    (read-line stream) ;; remove the PUSH op
    (read-line stream) ;; remove the END op
    (subseq val 1 (1- (length val)))))
```

5.3.3 Test the new Backend.

This can be tested with the code

```
Python
>>> import pickle
>>> pickle.dump('Foobar', open('/tmp/foo.p', 'w'))

Lisp
* (cl-store:restore "/tmp/foo.p" 'pickle)
=> "Foobar"
And

Lisp
* (cl-store:store "BarFoo" "/tmp/foo.p" 'pickle)

Python
>>> pickle.load(open('/tmp/foo.p'))
'BarFoo'
```

5.4 API

5.4.1 Functions

`backend-restore` *backend place* [Generic]
Restore the object found in stream *place* using backend *backend*. Checks the magic-number and invokes `backend-restore-object`. Called by `restore`, override for custom restoring.

`backend-restore` *backend place* [Generic]
Find the next function to call to restore the next object with *backend* and invoke it with *place*. Called by `restore-object`, override this method to do custom restoring (see '`circularities.lisp`' for an example).

`backend-store` *backend place obj* [Generic]
Stores the backend code and calls `store-object`. This is called by `store`. Override for custom storing.

`backend-store-object` *backend obj place* [Generic]
Called by `store-object`, override this to do custom storing (see '`circularities.lisp`' for an example).

`get-next-reader` *backend place* [Generic]
Method which must be specialized for *backend* to return the next symbol designating a `defrestore` instance to restore an object from *place*. If no reader is found return a second value which will be included in the error.

5.4.2 Macros

`defbackend` *name* **&key** (*stream-type* **(required-arg "stream-type")**) [Macro]
magic-number fields (*extends* **'backend**) *old-magic-numbers*
eg. `(defbackend pickle :stream-type 'character)` This creates a new backend called *name*, *stream-type* describes the type of stream that the backend will serialize to which must be suitable as an argument to open. *Magic-number*, when present, must be of type (unsigned-byte 32) which will be written as a verifier for the backend. *Fields* are extra fields to be added to the new class which will be created. By default the *extends* keyword is *backend*,the root backend, but this can be any legal backend. *Old-magic-numbers* holds previous magic-numbers that have been used by the backend to identify incompatible versions of objects stored.

6 Notes

6.1 Backend Designators

The *backend* argument to store, restore and with-backend is a backend designator which can be one of.

- A backend returned by `(find-backend name)`
- A symbol designating a backend (the first argument to defbackend).

6.2 Known Issues

- CLISP, OpenMCL, Allegro CL cannot store structure instances.
- Structure definitions are only supported in SBCL and CMUCL.
- Due to the fact that function's aren't fully supported CLOS Classes initfunction slot cannot be serialized.

6.3 Regarding String Serialization

Users are required to be extremely careful when serializing strings from one lisp implementation to another since the array-element-type will be tracked for strings and the Hyperspec does not specify an upper limit for base-chars. This can be a problem if you serialize a simple-base-string containing wide characters, in an implementation which specifies no limit on base-char, to an implementation with a limit. If you have a solution I would be happy to hear it.

7 Credits

Thanks To

- Common-Lisp.net: For project hosting.
- Alain Picard : Structure Storing and support for Infinite Floats for Lispworks.
- Robert Sedgewick: Package Imports for OpenMCL and suggesting Multiple Backends.
- Thomas Stenhaug: Comprehensive package storing and miscellaneous improvements.
- Killian Sprotte: Type specification fixups.

8 Index

8.1 Function Index

8.2 Variable Index