NAME

Storable - persistence for Perl data structures

SYNOPSIS

```
use Storable;
store \%table, 'file';
$hashref = retrieve('file');

use Storable qw(nstore store_fd nstore_fd freeze thaw dclone);

# Network order
nstore \%table, 'file';
$hashref = retrieve('file'); # There is NO nretrieve()

# Storing to and retrieving from an already opened file
store_fd \@array, \*STDOUT;
nstore_fd \%table, \*STDOUT;
$aryref = fd_retrieve(\*SOCKET);
$hashref = fd_retrieve(\*SOCKET);

# Serializing to memory
$serialized = freeze \%table;
%table_clone = %{ thaw($serialized) };

# Deep (recursive) cloning
$cloneref = dclone($ref);

# Advisory locking
use Storable qw(lock_store lock_nstore lock_retrieve)
lock_store \%table, 'file';
lock_nstore \%table, 'file';
$hashref = lock_retrieve('file');
```

DESCRIPTION

The Storable package brings persistence to your Perl data structures containing SCALAR, ARRAY, HASH or REF objects, i.e. anything that can be conveniently stored to disk and retrieved at a later time.

It can be used in the regular procedural way by calling `store` with a reference to the object to be stored, along with the file name where the image should be written.

The routine returns `undef` for I/O problems or other internal error, a true value otherwise. Serious errors are propagated as a `die` exception.

To retrieve data stored to disk, use `retrieve` with a file name. The objects stored into that file are recreated into memory for you, and a *reference* to the root object is returned. In case an I/O error occurs while reading, `undef` is returned instead. Other serious errors are propagated via `die`.

Since storage is performed recursively, you might want to stuff references to objects that share a lot of common data into a single array or hash table, and then store that object. That way, when you retrieve back the whole thing, the objects will continue to share what they originally shared.

At the cost of a slight header overhead, you may store to an already opened file descriptor using the `store_fd` routine, and retrieve from a file via `fd_retrieve`. Those names aren't imported by default, so you will have to do that explicitly if you need those routines. The file descriptor you supply must be already opened, for read if you're going to retrieve and for write if you wish to store.

```
store_fd(\%table, *STDOUT) || die "can't store to stdout\n";
$hashref = fd_retrieve(*STDIN);
```

You can also store data in network order to allow easy sharing across multiple platforms, or when storing

on a socket known to be remotely connected. The routines to call have an initial `n` prefix for *network*, as in `nstore` and `nstore_fd`. At retrieval time, your data will be correctly restored so you don't have to know whether you're restoring from native or network ordered data. Double values are stored stringified to ensure portability as well, at the slight risk of loosing some precision in the last decimals.

When using `fd_retrieve`, objects are retrieved in sequence, one object (i.e. one recursive tree) per associated `store_fd`.

If you're more from the object-oriented camp, you can inherit from Storable and directly store your objects by invoking `store` as a method. The fact that the root of the to-be-stored tree is a blessed reference (i.e. an object) is special-cased so that the retrieve does not provide a reference to that object but rather the blessed object reference itself. (Otherwise, you'd get a reference to that blessed object).

MEMORY STORE

The Storable engine can also store data into a Perl scalar instead, to later retrieve them. This is mainly used to freeze a complex structure in some safe compact memory place (where it can possibly be sent to another process via some IPC, since freezing the structure also serializes it in effect). Later on, and maybe somewhere else, you can thaw the Perl scalar out and recreate the original complex structure in memory.

Surprisingly, the routines to be called are named `freeze` and `thaw`. If you wish to send out the frozen scalar to another machine, use `nfreeze` instead to get a portable image.

Note that freezing an object structure and immediately thawing it actually achieves a deep cloning of that structure:

```
dclone(.) = thaw(freeze(.))
```

Storable provides you with a `dclone` interface which does not create that intermediary scalar but instead freezes the structure in some internal memory space and then immediately thaws it out.

ADVISORY LOCKING

The `lock_store` and `lock_nstore` routine are equivalent to `store` and `nstore`, except that they get an exclusive lock on the file before writing. Likewise, `lock_retrieve` does the same as `retrieve`, but also gets a shared lock on the file before reading.

As with any advisory locking scheme, the protection only works if you systematically use `lock_store` and `lock_retrieve`. If one side of your application uses `store` whilst the other uses `lock_retrieve`, you will get no protection at all.

The internal advisory locking is implemented using Perl's *flock()* routine. If your system does not support any form of *flock()*, or if you share your files across NFS, you might wish to use other forms of locking by using modules such as LockFile::Simple which lock a file using a filesystem entry, instead of locking the file descriptor.

SPEED

The heart of Storable is written in C for decent speed. Extra low-level optimizations have been made when manipulating perl internals, to sacrifice encapsulation for the benefit of greater speed.

CANONICAL REPRESENTATION

Normally, Storable stores elements of hashes in the order they are stored internally by Perl, i.e. pseudo-randomly. If you set `$Storable::canonical` to some `TRUE` value, Storable will store hashes with the elements sorted by their key. This allows you to compare data structures by comparing their frozen representations (or even the compressed frozen representations), which can be useful for creating lookup tables for complicated queries.

Canonical order does not imply network order; those are two orthogonal settings.

CODE REFERENCES

Since Storable version 2.05, CODE references may be serialized with the help of B::Deparse. To enable this feature, set `$Storable::Deparse` to a true value. To enable deserialization, `$Storable::Eval` should be set to a true value. Be aware that deserialization is done through `eval`, which is dangerous if the Storable file contains malicious data. You can set `$Storable::Eval` to a subroutine reference which would be used instead of `eval`. See below for an example using a Safe compartment for deserialization of

CODE references.

If `$Storable::Deparse` and/or `$Storable::Eval` are set to false values, then the value of `$Storable::forgive_me` (see below) is respected while serializing and deserializing.

FORWARD COMPATIBILITY

This release of Storable can be used on a newer version of Perl to serialize data which is not supported by earlier Perls. By default, Storable will attempt to do the right thing, by `croak()`ing if it encounters data that it cannot deserialize. However, the defaults can be changed as follows:

utf8 data

Perl 5.6 added support for Unicode characters with code points > 255, and Perl 5.8 has full support for Unicode characters in hash keys. Perl internally encodes strings with these characters using utf8, and Storable serializes them as utf8. By default, if an older version of Perl encounters a utf8 value it cannot represent, it will `croak()`. To change this behaviour so that Storable deserializes utf8 encoded values as the string of bytes (effectively dropping the *is_utf8* flag) set `$Storable::drop_utf8` to some `TRUE` value. This is a form of data loss, because with `$drop_utf8` true, it becomes impossible to tell whether the original data was the Unicode string, or a series of bytes that happen to be valid utf8.

restricted hashes

Perl 5.8 adds support for restricted hashes, which have keys restricted to a given set, and can have values locked to be read only. By default, when Storable encounters a restricted hash on a perl that doesn't support them, it will deserialize it as a normal hash, silently discarding any placeholder keys and leaving the keys and all values unlocked. To make Storable `croak()` instead, set `$Storable::downgrade_restricted` to a `FALSE` value. To restore the default set it back to some `TRUE` value.

files from future versions of Storable

Earlier versions of Storable would immediately croak if they encountered a file with a higher internal version number than the reading Storable knew about. Internal version numbers are increased each time new data types (such as restricted hashes) are added to the vocabulary of the file format. This meant that a newer Storable module had no way of writing a file readable by an older Storable, even if the writer didn't store newer data types.

This version of Storable will defer croaking until it encounters a data type in the file that it does not recognize. This means that it will continue to read files generated by newer Storable modules which are careful in what they write out, making it easier to upgrade Storable modules in a mixed environment.

The old behaviour of immediate croaking can be re-instated by setting `$Storable::accept_future_minor` to some `FALSE` value.

All these variables have no effect on a newer Perl which supports the relevant feature.

ERROR REPORTING

Storable uses the ''exception'' paradigm, in that it does not try to workaround failures: if something bad happens, an exception is generated from the caller's perspective (see Carp and `croak()`). Use eval {} to trap those exceptions.

When Storable croaks, it tries to report the error via the `logcroak()` routine from the `Log::Agent` package, if it is available.

Normal errors are reported by having *store()* or *retrieve()* return `undef`. Such errors are usually I/O errors (or truncated stream errors at retrieval).

WIZARDS ONLY

Hooks

Any class may define hooks that will be called during the serialization and deserialization process on objects that are instances of that class. Those hooks can redefine the way serialization is performed (and therefore, how the symmetrical deserialization should be conducted).

Since we said earlier:

```
dclone(.) = thaw(freeze(.))
```

everything we say about hooks should also hold for deep cloning. However, hooks get to know whether the operation is a mere serialization, or a cloning.

Therefore, when serializing hooks are involved,

```
dclone(.) <> thaw(freeze(.))
```

Well, you could keep them in sync, but there's no guarantee it will always hold on classes somebody else wrote. Besides, there is little to gain in doing so: a serializing hook could keep only one attribute of an object, which is probably not what should happen during a deep cloning of that same object.

Here is the hooking interface:

`STORABLE_freeze` *obj*, *cloning*

The serializing hook, called on the object during serialization. It can be inherited, or defined in the class itself, like any other method.

Arguments: *obj* is the object to serialize, *cloning* is a flag indicating whether we're in a *dclone()* or a regular serialization via *store()* or *freeze()*.

Returned value: A LIST `($serialized, $ref1, $ref2, ...)` where `$serialized` is the serialized form to be used, and the optional`$ref1`, `$ref2`, etc... are e xtra references that you wish to let the Storable engine serialize.

At deserialization time, you will be given back the same LIST, but all the extra references will be pointing into the deserialized structure.

The **first time** the hook is hit in a serialization flow, you may have it return an empty list. That will signal the Storable engine to further discard that hook for this class and to therefore revert to the default serialization of the underlying Perl data. The hook will again be normally processed in the next serialization.

Unless you know better, serializing hook should always say:

```
sub STORABLE_freeze {
my ($self, $cloning) = @_;
return if $cloning; # Regular default serialization
....
}
```

in order to keep reasonable *dclone()* semantics.

`STORABLE_thaw` *obj*, *cloning*, *serialized*, ...

The deserializing hook called on the object during deserialization. But wait: if we're deserializing, there's no object yet... right?

Wrong: the Storable engine creates an empty one for you. If you know Eiffel, you can view `STORABLE_thaw` as an alternate creation routine.

This means the hook can be inherited like any other method, and that *obj* is your blessed reference for this particular instance.

The other arguments should look familiar if you know `STORABLE_freeze`: *cloning* is true when we're part of a deep clone operation, *serialized* is the serialized string you returned to the engine in `STORABLE_freeze`, and there may be an optional list of references, in the same order you gave them at serialization time, pointing to the deserialized objects (which have been processed courtesy of the Storable engine).

When the Storable engine does not find any `STORABLE_thaw` hook routine, it tries to load the class by requiring the package dynamically (using the blessed package name), and then re-attempts the lookup. If at that time the hook cannot be located, the engine croaks. Note that this mechanism will

fail if you define several classes in the same file, but perlmod(1) warned you.

It is up to you to use this information to populate *obj* the way you want.

Returned value: none.

`STORABLE_attach` *class*, *cloning*, *serialized*

While `STORABLE_freeze` and `STORABLE_thaw` are useful for classes where each instance is independent, this mechanism has difficulty (or is incompatible) with objects that exist as common process-level or system-level resources, such as singleton objects, database pools, caches or memoized objects.

The alternative `STORABLE_attach` method provides a solution for these shared objects. Instead of `STORABLE_freeze` --> `STORABLE_thaw`, you implement `STORABLE_freeze` --> `STORABLE_attach` instead.

Arguments: *class* is the class we are attaching to, *cloning* is a flag indicating whether we're in a *dclone()* or a regular de-serialization via *thaw()*, and *serialized* is the stored string for the resource object.

Because these resource objects are considered to be owned by the entire process/system, and not the ''property'' of whatever is being serialized, no references underneath the object should be included in the serialized string. Thus, in any class that implements `STORABLE_attach`, the `STORABLE_freeze` method cannot return any references, and `Storable` will throw an error if `STORABLE_freeze` tries to return references.

All information required to ''attach'' back to the shared resource object **must** be contained **only** in the `STORABLE_freeze` return string. Otherwise, `STORABLE_freeze` behaves as normal for `STORABLE_attach` classes.

Because `STORABLE_attach` is passed the class (rather than an object), it also returns the object directly, rather than modifying the passed object.

Returned value: object of type `class`

Predicates

Predicates are not exportable. They must be called by explicitly prefixing them with the Storable package name.

`Storable::last_op_in_netorder`

The `Storable::last_op_in_netorder()` predicate will tell you whether network order was used in the last store or retrieve operation. If you don't know how to use this, just forget about it.

`Storable::is_storing`

Returns true if within a store operation (via STORABLE_freeze hook).

`Storable::is_retrieving`

Returns true if within a retrieve operation (via STORABLE_thaw hook).

Recursion

With hooks comes the ability to recurse back to the Storable engine. Indeed, hooks are regular Perl code, and Storable is convenient when it comes to serializing and deserializing things, so why not use it to handle the serialization string?

There are a few things you need to know, however:

- You can create endless loops if the things you serialize via *freeze()* (for instance) point back to the object we're trying to serialize in the hook.
- Shared references among objects will not stay shared: if we're serializing the list of object [A, C] where both object A and C refer to the SAME object B, and if there is a serializing hook in A that says freeze(B), then when deserializing, we'll get [A', C'] where A' refers to B', but C' refers to D, a deep clone of B'. The topology was not preserved.

That's why `STORABLE_freeze` lets you provide a list of references to serialize. The engine guarantees that those will be serialized in the same context as the other objects, and therefore that shared objects will stay shared.

In the above [A, C] example, the `STORABLE_freeze` hook could return:

```
("something", $self->{B})
```

and the B part would be serialized by the engine. In `STORABLE_thaw`, you would get back the reference to the B' object, deserialized for you.

Therefore, recursion should normally be avoided, but is nonetheless supported.

Deep Cloning

There is a Clone module available on CPAN which implements deep cloning natively, i.e. without freezing to memory and thawing the result. It is aimed to replace Storable's *dclone()* some day. However, it does not currently support Storable hooks to redefine the way deep cloning is performed.

Storable magic

Yes, there's a lot of that :-) But more precisely, in UNIX systems there's a utility called `file`, which recognizes data files based on their contents (usually their first few bytes). For this to work, a certain file called *magic* needs to taught about the *signature* of the data. Where that configuration file lives depends on the UNIX flavour; often it's something like */usr/share/misc/magic* or */etc/magic*. Your system administrator needs to do the updating of the *magic* file. The necessary signature information is output to STDOUT by invoking *Storable::show_file_magic()*. Note that the GNU implementation of the `file` utility, version 3.38 or later, is expected to contain support for recognising Storable files out-of-the-box, in addition to other kinds of Perl files.

You can also use the following functions to extract the file header information from Storable images:

`$info` = Storable::file_magic(`$filename`)

If the given file is a Storable image return a hash describing it. If the file is readable, but not a Storable image return `undef`. If the file does not exist or is unreadable then croak.

The hash returned has the following elements:

`version`

This returns the file format version. It is a string like ''2.7''.

Note that this version number is not the same as the version number of the Storable module itself. For instance Storable v0.7 create files in format v2.0 and Storable v2.15 create files in format v2.7. The file format version number only increment when additional features that would confuse older versions of the module are added.

Files older than v2.0 will have the one of the version numbers ''-1'', ''0'' or ''1''. No minor number was used at that time.

`version_nv`

This returns the file format version as number. It is a string like ''2.007''. This value is suitable for numeric comparisons.

The constant function `Storable::BIN_VERSION_NV` returns a comparable number that represents the highest file version number that this version of Storable fully supports (but see discussion of `$Storable::accept_future_minor` above). The constant `Storable::BIN_WRITE_VERSION_NV` function returns what file version is written and might be less than `Storable::BIN_VERSION_NV` in some configurations.

`major`, `minor`

This also returns the file format version. If the version is ''2.7'' then major would be 2 and minor would be 7. The minor element is missing for when major is less than 2.

`hdrsize`

The is the number of bytes that the Storable header occupies.

`netorder`
: This is TRUE if the image store data in network order. This means that it was created with *nstore()* or similar.

`byteorder`
: This is only present when `netorder` is FALSE. It is the`$Config`{byteorder} string of the perl that created this image. It is a string like ‘‘1234’’ (32 bit little endian) or ‘‘87654321’’ (64 bit big endian). This must match the current perl for the image to be readable by Storable.

`intsize`, `longsize`, `ptrsize`, `nvsize`
: These are only present when `netorder` is FALSE. These are the sizes of various C datatypes of the perl that created this image. These must match the current perl for the image to be readable by Storable.

  The `nvsize` element is only present for file format v2.2 and higher.

`file`
: The name of the file.

`$info` = Storable::read_magic(`$buffer`)
`$info` = Storable::read_magic(`$buffer`, `$must_be_file`)
: The `$buffer` should be a Storable image or the first few bytes of it. If `$buffer` starts with a Storable header, then a hash describing the image is returned, otherwise `undef` is returned.

  The hash has the same structure as the one returned by *Storable::file_magic()*. The `file` element is true if the image is a file image.

  If the `$must_be_file` argument is provided and is TRUE, then return `undef` unless the image looks like it belongs to a file dump.

  The maximum size of a Storable header is currently 21 bytes. If the provided `$buffer` is only the first part of a Storable image it should at least be this long to ensure that *read_magic()* will recognize it as such.

EXAMPLES

Here are some code samples showing a possible usage of Storable:

```
use Storable qw(store retrieve freeze thaw dclone);

%color = ('Blue' => 0.1, 'Red' => 0.8, 'Black' => 0, 'White' => 1);

store(\%color, 'mycolors') or die "Can't store %a in mycolors!\n";

$colref = retrieve('mycolors');
die "Unable to retrieve from mycolors!\n" unless defined $colref;
printf "Blue is still %lf\n", $colref->{'Blue'};

$colref2 = dclone(\%color);

$str = freeze(\%color);
printf "Serialization of %%color is %d bytes long.\n", length($str);
$colref3 = thaw($str);
```

which prints (on my machine):

```
Blue is still 0.100000
Serialization of %color is 102 bytes long.
```

Serialization of CODE references and deserialization in a safe compartment:

```
use Storable qw(freeze thaw);
use Safe;
use strict;
my $safe = new Safe;
# because of opcodes used in "use strict":
$safe->permit(qw(:default require));
local $Storable::Deparse = 1;
local $Storable::Eval = sub { $safe->reval($_[0]) };
my $serialized = freeze(sub { 42 });
my $code = thaw($serialized);
$code->() == 42;
```

SECURITY WARNING

Do not accept Storable documents from untrusted sources!

Some features of Storable can lead to security vulnerabilities if you accept Storable documents from untrusted sources. Most obviously, the optional (off by default) CODE reference serialization feature allows transfer of code to the deserializing process. Furthermore, any serialized object will cause Storable to helpfully load the module corresponding to the class of the object in the deserializing module. For manipulated module names, this can load almost arbitrary code. Finally, the deserialized object's destructors will be invoked when the objects get destroyed in the deserializing process. Maliciously crafted Storable documents may put such objects in the value of a hash key that is overridden by another key/value pair in the same hash, thus causing immediate destructor execution.

In a future version of Storable, we intend to provide options to disable loading modules for classes and to disable deserializing objects altogether. *Nonetheless, Storable deserializing documents from untrusted sources is expected to have other, yet undiscovered, security concerns such as allowing an attacker to cause the deserializer to crash hard.*

Therefore, let me repeat: Do not accept Storable documents from untrusted sources!

If your application requires accepting data from untrusted sources, you are best off with a less powerful and more-likely safe serialization format and implementation. If your data is sufficiently simple, JSON is a good choice and offers maximum interoperability.

WARNING

If you're using references as keys within your hash tables, you're bound to be disappointed when retrieving your data. Indeed, Perl stringifies references used as hash table keys. If you later wish to access the items via another reference stringification (i.e. using the same reference that was used for the key originally to record the value into the hash table), it will work because both references stringify to the same string.

It won't work across a sequence of `store` and `retrieve` operations, however, because the addresses in the retrieved objects, which are part of the stringified references, will probably differ from the original addresses. The topology of your structure is preserved, but not hidden semantics like those.

On platforms where it matters, be sure to call `binmode()` on the descriptors that you pass to Storable functions.

Storing data canonically that contains large hashes can be significantly slower than storing the same data normally, as temporary arrays to hold the keys for each hash have to be allocated, populated, sorted and freed. Some tests have shown a halving of the speed of storing — the exact penalty will depend on the complexity of your data. There is no slowdown on retrieval.

BUGS

You can't store GLOB, FORMLINE, REGEXP, etc.... If you can define semantics for those operations, feel free to enhance Storable so that it can deal with them.

The store functions will `croak` if they run into such references unless you set `$Storable::forgive_me` to some `TRUE` value. In that case, the fatal message is converted to a warning and some meaningless string is stored instead.

Setting `$Storable::canonical` may not yield frozen strings that compare equal due to possible

stringification of numbers. When the string version of a scalar exists, it is the form stored; therefore, if you happen to use your numbers as strings between two freezing operations on the same data structures, you will get different results.

When storing doubles in network order, their value is stored as text. However, you should also not expect non-numeric floating-point values such as infinity and ''not a number'' to pass successfully through a *nstore()*/*retrieve()* pair.

As Storable neither knows nor cares about character sets (although it does know that characters may be more than eight bits wide), any difference in the interpretation of character codes between a host and a target system is your problem. In particular, if host and target use different code points to represent the characters used in the text representation of floating-point numbers, you will not be able be able to exchange floating-point data, even with *nstore()*.

`Storable::drop_utf8` is a blunt tool. There is no facility either to return **all** strings as utf8 sequences, or to attempt to convert utf8 data back to 8 bit and `croak()` if the conversion fails.

Prior to Storable 2.01, no distinction was made between signed and unsigned integers on storing. By default Storable prefers to store a scalars string representation (if it has one) so this would only cause problems when storing large unsigned integers that had never been converted to string or floating point. In other words values that had been generated by integer operations such as logic ops and then not used in any string or arithmetic context before storing.

64 bit data in perl 5.6.0 and 5.6.1

This section only applies to you if you have existing data written out by Storable 2.02 or earlier on perl 5.6.0 or 5.6.1 on Unix or Linux which has been configured with 64 bit integer support (not the default) If you got a precompiled perl, rather than running Configure to build your own perl from source, then it almost certainly does not affect you, and you can stop reading now (unless you're curious). If you're using perl on Windows it does not affect you.

Storable writes a file header which contains the sizes of various C language types for the C compiler that built Storable (when not writing in network order), and will refuse to load files written by a Storable not on the same (or compatible) architecture. This check and a check on machine byteorder is needed because the size of various fields in the file are given by the sizes of the C language types, and so files written on different architectures are incompatible. This is done for increased speed. (When writing in network order, all fields are written out as standard lengths, which allows full interworking, but takes longer to read and write)

Perl 5.6.x introduced the ability to optional configure the perl interpreter to use C's `long long` type to allow scalars to store 64 bit integers on 32 bit systems. However, due to the way the Perl configuration system generated the C configuration files on non-Windows platforms, and the way Storable generates its header, nothing in the Storable file header reflected whether the perl writing was using 32 or 64 bit integers, despite the fact that Storable was storing some data differently in the file. Hence Storable running on perl with 64 bit integers will read the header from a file written by a 32 bit perl, not realise that the data is actually in a subtly incompatible format, and then go horribly wrong (possibly crashing) if it encountered a stored integer. This is a design failure.

Storable has now been changed to write out and read in a file header with information about the size of integers. It's impossible to detect whether an old file being read in was written with 32 or 64 bit integers (they have the same header) so it's impossible to automatically switch to a correct backwards compatibility mode. Hence this Storable defaults to the new, correct behaviour.

What this means is that if you have data written by Storable 1.x running on perl 5.6.0 or 5.6.1 configured with 64 bit integers on Unix or Linux then by default this Storable will refuse to read it, giving the error *Byte order is not compatible*. If you have such data then you should set `$Storable::interwork_56_64bit` to a true value to make this Storable read and write files with the old header. You should also migrate your data, or any older perl you are communicating with, to this current version of Storable.

If you don't have data written with specific configuration of perl described above, then you do not and

should not do anything. Don't set the flag - not only will Storable on an identically configured perl refuse to load them, but Storable a differently configured perl will load them believing them to be correct for it, and then may well fail or crash part way through reading them.

CREDITS

Thank you to (in chronological order):

```
Jarkko Hietaniemi <jhi@iki.fi>
Ulrich Pfeifer <pfeifer@charly.informatik.uni-dortmund.de>
Benjamin A. Holzman <bholzman@earthlink.net>
Andrew Ford <A.Ford@ford-mason.co.uk>
Gisle Aas <gisle@aas.no>
Jeff Gresham <gresham_jeffrey@jpmorgan.com>
Murray Nesbitt <murray@activestate.com>
Marc Lehmann <pcg@opengroup.org>
Justin Banks <justinb@wamnet.com>
Jarkko Hietaniemi <jhi@iki.fi> (AGAIN, as perl 5.7.0 Pumpkin!)
Salvador Ortiz Garcia <sog@msg.com.mx>
Dominic Dunlop <domo@computer.org>
Erik Haugan <erik@solbors.no>
Benjamin A. Holzman <ben.holzman@grantstreet.com>
Reini Urban <rurban@cpanel.net>
```

for their bug reports, suggestions and contributions.

Benjamin Holzman contributed the tied variable support, Andrew Ford contributed the canonical order for hashes, and Gisle Aas fixed a few misunderstandings of mine regarding the perl internals, and optimized the emission of ''tags'' in the output streams by simply counting the objects instead of tagging them (leading to a binary incompatibility for the Storable image starting at version 0.6--older images are, of course, still properly understood). Murray Nesbitt made Storable thread-safe. Marc Lehmann added overloading and references to tied items support. Benjamin Holzman added a performance improvement for overloaded classes; thanks to Grant Street Group for footing the bill.

AUTHOR

Storable was written by Raphael Manfredi <*Raphael_Manfredi@pobox.com*> Maintenance is now done by the perl5-porters <*perl5-porters@perl.org*>

Please e-mail us with problems, bug fixes, comments and complaints, although if you have compliments you should send them to Raphael. Please don't e-mail Raphael with problems, as he no longer works on Storable, and your message will be delayed while he forwards it to us.

SEE ALSO

Clone.