NAME

Memoize::Expire - Plug-in module for automatic expiration of memoized values

SYNOPSIS

```
use Memoize;
use Memoize::Expire;
tie my %cache => 'Memoize::Expire',
LIFETIME => $lifetime, # In seconds
NUM_USES => $n_uses;

memoize 'function', SCALAR_CACHE => [HASH => \%cache ];
```

DESCRIPTION

Memoize::Expire is a plug-in module for Memoize. It allows the cached values for memoized functions to expire automatically. This manual assumes you are already familiar with the Memoize module. If not, you should study that manual carefully first, paying particular attention to the HASH feature.

Memoize::Expire is a layer of software that you can insert in between Memoize itself and whatever underlying package implements the cache. The layer presents a hash variable whose values expire whenever they get too old, have been used too often, or both. You tell `Memoize` to use this forgetful hash as its cache instead of the default, which is an ordinary hash.

To specify a real-time timeout, supply the `LIFETIME` option with a numeric value. Cached data will expire after this many seconds, and will be looked up afresh when it expires. When a data item is looked up afresh, its lifetime is reset.

If you specify `NUM_USES` with an argument of *n*, then each cached data item will be discarded and looked up afresh after the *n*th time you access it. When a data item is looked up afresh, its number of uses is reset.

If you specify both arguments, data will be discarded from the cache when either expiration condition holds.

Memoize::Expire uses a real hash internally to store the cached data. You can use the `HASH` option to Memoize::Expire to supply a tied hash in place of the ordinary hash that Memoize::Expire will normally use. You can use this feature to add Memoize::Expire as a layer in between a persistent disk hash and Memoize. If you do this, you get a persistent disk cache whose entries expire automatically. For example:

```
# Memoize
# |
# Memoize::Expire enforces data expiration policy
# |
# DB_File implements persistence of data in a disk file
# |
# Disk file

use Memoize;
use Memoize::Expire;
use DB_File;

# Set up persistence
tie my %disk_cache => 'DB_File', $filename, O_CREAT|O_RDWR, 0666];

# Set up expiration policy, supplying persistent hash as a target
tie my %cache => 'Memoize::Expire',
LIFETIME => $lifetime, # In seconds
NUM_USES => $n_uses,
```

```
HASH => \%disk_cache;

# Set up memoization, supplying expiring persistent hash for cache
memoize 'function', SCALAR_CACHE => [ HASH => \%cache ];
```

INTERFACE

There is nothing special about Memoize::Expire. It is just an example. If you don't like the policy that it implements, you are free to write your own expiration policy module that implements whatever policy you desire. Here is how to do that. Let us suppose that your module will be named MyExpirePolicy.

Short summary: You need to create a package that defines four methods:

TIEHASH
: Construct and return cache object.

EXISTS
: Given a function argument, is the corresponding function value in the cache, and if so, is it fresh enough to use?

FETCH
: Given a function argument, look up the corresponding function value in the cache and return it.

STORE
: Given a function argument and the corresponding function value, store them into the cache.

CLEAR
: (Optional.) Flush the cache completely.

The user who wants the memoization cache to be expired according to your policy will say so by writing

```
tie my %cache => 'MyExpirePolicy', args...;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

This will invoke `MyExpirePolicy->TIEHASH(args)`. MyExpirePolicy::TIEHASH should do whatever is appropriate to set up the cache, and it should return the cache object to the caller.

For example, MyExpirePolicy::TIEHASH might create an object that contains a regular Perl hash (which it will to store the cached values) and some extra information about the arguments and how old the data is and things like that. Let us call this object 'C'.

When Memoize needs to check to see if an entry is in the cache already, it will invoke `C->EXISTS(key)`. `key` is the normalized function argument. MyExpirePolicy::EXISTS should return 0 if the key is not in the cache, or if it has expired, and 1 if an unexpired value is in the cache. It should *not* return `undef`, because there is a bug in some versions of Perl that will cause a spurious FETCH if the EXISTS method returns `undef`.

If your EXISTS function returns true, Memoize will try to fetch the cached value by invoking `C->FETCH(key)`. MyExpirePolicy::FETCH should return the cached value. Otherwise, Memoize will call the memoized function to compute the appropriate value, and will store it into the cache by calling `C->STORE(key, value)`.

Here is a very brief example of a policy module that expires each cache item after ten seconds.

```
package Memoize::TenSecondExpire;

sub TIEHASH {
my ($package, %args) = @_;
my $cache = $args{HASH} || {};
bless $cache => $package;
}
```

```
sub EXISTS {
my ($cache, $key) = @_;
if (exists $cache->{$key} &&
$cache->{$key}{EXPIRE_TIME} > time) {
return 1
} else {
return 0; # Do NOT return `undef' here.
}
}

sub FETCH {
my ($cache, $key) = @_;
return $cache->{$key}{VALUE};
}

sub STORE {
my ($cache, $key, $newvalue) = @_;
$cache->{$key}{VALUE} = $newvalue;
$cache->{$key}{EXPIRE_TIME} = time + 10;
}
```

To use this expiration policy, the user would say

```
use Memoize;
tie my %cache10sec => 'Memoize::TenSecondExpire';
memoize 'function', SCALAR_CACHE => [HASH => \%cache10sec];
```

Memoize would then call `function` whenever a cached value was entirely absent or was older than ten seconds.

You should always support a `HASH` argument to `TIEHASH` that ties the underlying cache so that the user can specify that the cache is also persistent or that it has some other interesting semantics. The example above demonstrates how to do this, as does `Memoize::Expire`

Another sample module, Memoize::Saves, is available in a separate distribution on CPAN. It implements a policy that allows you to specify that certain function values would always be looked up afresh. See the documentation for details.

ALTERNATIVES

Brent Powers has a `Memoize::ExpireLRU` module that was designed to work with Memoize and provides expiration of least-recently-used data. The cache is held at a fixed number of entries, and when new data comes in, the least-recently used data is expired. See <http://search.cpan.org/search?mode=module&query=ExpireLRU>.

Joshua Chamas's Tie::Cache module may be useful as an expiration manager. (If you try this, let me know how it works out.)

If you develop any useful expiration managers that you think should be distributed with Memoize, please let me know.

CAVEATS

This module is experimental, and may contain bugs. Please report bugs to the address below.

Number-of-uses is stored as a 16-bit unsigned integer, so can't exceed 65535.

Because of clock granularity, expiration times may occur up to one second sooner than you expect. For example, suppose you store a value with a lifetime of ten seconds, and you store it at 12:00:00.998 on a certain day. Memoize will look at the clock and see 12:00:00. Then 9.01 seconds later, at 12:00:10.008 you try to read it back. Memoize will look at the clock and see 12:00:10 and conclude that the value has expired. This will probably not occur if you have `Time::HiRes`

installed.

AUTHOR

Mark-Jason Dominus (mjd-perl-memoize+@plover.com)

Mike Cariaso provided valuable insight into the best way to solve this problem.

SEE ALSO

perl(1)

The Memoize man page.

http://www.plover.com/~mjd/perl/Memoize/
(for news and updates)

I maintain a mailing list on which I occasionally announce new versions of Memoize. The list is for announcements only, not discussion. To join, send an empty message to mjd-perl-memoize-request@Plover.com.