NAME

MLDBM − store multi−level Perl hash structure in single level tied hash

SYNOPSIS

```
use MLDBM;                          # this gets the default, SDBM
#use MLDBM qw(DB_File FreezeThaw);  # use FreezeThaw for serializing
#use MLDBM qw(DB_File Storable);    # use Storable for serializing

$dbm = tie %o, 'MLDBM' [..other DBM args..] or die $!;
```

DESCRIPTION

This module can serve as a transparent interface to any TIEHASH package that is required to store arbitrary perl data, including nested references. Thus, this module can be used for storing references and other arbitrary data within DBM databases.

It works by serializing the references in the hash into a single string. In the underlying TIEHASH package (usually a DBM database), it is this string that gets stored. When the value is fetched again, the string is deserialized to reconstruct the data structure into memory.

For historical and practical reasons, it requires the **Data::Dumper** package, available at any CPAN site. **Data::Dumper** gives you really nice-looking dumps of your data structures, in case you wish to look at them on the screen, and it was the only serializing engine before version 2.00. However, as of version 2.00, you can use any of **Data::Dumper**, **FreezeThaw** or **Storable** to perform the underlying serialization, as hinted at by the SYNOPSIS overview above. Using **Storable** is usually much faster than the other methods.

See the BUGS section for important limitations.

Changing the Defaults

MLDBM relies on an underlying TIEHASH implementation (usually a DBM package), and an underlying serialization package. The respective defaults are **SDBM_File** and **Data::Dumper**. Both of these defaults can be changed. Changing the **SDBM_File** default is strongly recommended. See WARNINGS below.

Three serialization wrappers are currently supported: **Data::Dumper**, **Storable**, and **FreezeThaw**. Additional serializers can be supported by writing a wrapper that implements the interface required by **MLDBM::Serializer**. See the supported wrappers and the **MLDBM::Serializer** source for details.

In the following, *$OBJ* stands for the tied object, as in:

```
$obj = tie %o, ....
$obj = tied %o;
```

`$MLDBM::UseDB` *or* *$OBJ*−>UseDB(*[TIEDOBJECT]*)

The global `$MLDBM::UseDB` can be set to default to something other than `SDBM_File`, in case you have a more efficient DBM, or if you want to use this with some other TIEHASH implementation. Alternatively, you can specify the name of the package at `use` time, as the first ‘‘parameter’’. Nested module names can be specified as ‘‘Foo::Bar’’.

The corresponding method call returns the underlying TIEHASH object when called without arguments. It can be called with any object that implements Perl’s TIEHASH interface, to set that value.

`$MLDBM::Serializer` *or* *$OBJ*−>Serializer(*[SZROBJECT]*)

The global `$MLDBM::Serializer` can be set to the name of the serializing package to be used. Currently can be set to one of `Data::Dumper`, `Storable`, or `FreezeThaw`. Defaults to `Data::Dumper`. Alternatively, you can specify the name of the serializer package at `use` time, as the second ‘‘parameter’’.

The corresponding method call returns the underlying MLDBM serializer object when called without arguments. It can be called with an object that implements the MLDBM serializer interface, to set that value.

Controlling Serializer Properties

These methods are meant to supply an interface to the properties of the underlying serializer used. Do **not** call or set them without understanding the consequences in full. The defaults are usually sensible.

Not all of these necessarily apply to all the supplied serializers, so we specify when to apply them. Failure to respect this will usually lead to an exception.

`$MLDBM::DumpMeth` *or* *`$OBJ`*`->`DumpMeth(*[METHNAME]*)

If the serializer provides alternative serialization methods, this can be used to set them.

With **Data::Dumper** (which offers a pure Perl and an XS verion of its serializing routine), this is set to `Dumpxs` by default if that is supported in your installation. Otherwise, defaults to the slower `Dump` method.

With **Storable**, a value of `portable` requests that serialization be architecture neutral, i.e. the deserialization can later occur on another platform. Of course, this only makes sense if your database files are themselves architecture neutral. By default, native format is used for greater serializing speed in **Storable**. Both **Data::Dumper** and **FreezeThaw** are always architecture neutral.

FreezeThaw does not honor this attribute.

`$MLDBM::Key` *or* *`$OBJ`*`->`Key(*[KEYSTRING]*)

If the serializer only deals with part of the data (perhaps because the TIEHASH object can natively store some types of data), it may need a unique key string to recognize the data it handles. This can be used to set that string. Best left alone.

Defaults to the magic string used to recognize MLDBM data. It is a six character wide, unique string. This is best left alone, unless you know what you are doing.

Storable and **FreezeThaw** do not honor this attribute.

`$MLDBM::RemoveTaint` *or* *`$OBJ`*`->`RemoveTaint(*[BOOL]*)

If the serializer can optionally untaint any retrieved data subject to taint checks in Perl, this can be used to request that feature. Data that comes from external sources (like disk-files) must always be viewed with caution, so use this only when you are sure that that is not an issue.

Data::Dumper uses `eval()` to deserialize and is therefore subject to taint checks. Can be set to a true value to make the **Data::Dumper** serializer untaint the data retrieved. It is not enabled by default. Use with care.

Storable and **FreezeThaw** do not honor this attribute.

EXAMPLES

Here is a simple example. Note that does not depend upon the underlying serializing package — most real life examples should not, usually.

```
use MLDBM;                          # this gets SDBM and Data::Dumper
#use MLDBM qw(SDBM_File Storable);  # SDBM and Storable
use Fcntl;                          # to get 'em constants

$dbm = tie %o, 'MLDBM', 'testmldbm', O_CREAT|O_RDWR, 0640 or die $!;

$c = [\ 'c'];
$b = {};
$a = [1, $b, $c];
$b->{a} = $a;
$b->{b} = $a->[1];
$b->{c} = $a->[2];
@o{qw(a b c)} = ($a, $b, $c);

#
# to see what was stored
#
use Data::Dumper;
print Data::Dumper->Dump([@o{qw(a b c)}], [qw(a b c)]);

#
# to modify data in a substructure
```

```
#
$tmp = $o{a};
$tmp->[0] = 'foo';
$o{a} = $tmp;

#
# can access the underlying DBM methods transparently
#
#print $dbm->fd, "\n";              # DB_File method
```

Here is another small example using Storable, in a portable format:

```
use MLDBM qw(DB_File Storable);     # DB_File and Storable

tie %o, 'MLDBM', 'testmldbm', O_CREAT|O_RDWR, 0640 or die $!;

(tied %o)->DumpMeth('portable');    # Ask for portable binary
$o{'ENV'} = \%ENV;                  # Stores the whole environment
```

BUGS

1. Adding or altering substructures to a hash value is not entirely transparent in current perl. If you want to store a reference or modify an existing reference value in the DBM, it must first be retrieved and stored in a temporary variable for further modifications. In particular, something like this will NOT work properly:

   ```
   $mldb{key}{subkey}[3] = 'stuff';        # won't work
   ```

   Instead, that must be written as:

   ```
   $tmp = $mldb{key};                      # retrieve value
   $tmp->{subkey}[3] = 'stuff';
   $mldb{key} = $tmp;                      # store value
   ```

   This limitation exists because the perl TIEHASH interface currently has no support for multidimensional ties.

2. The **Data::Dumper** serializer uses *eval()*. A lot. Try the **Storable** serializer, which is generally the most efficient.

WARNINGS

1. Many DBM implementations have arbitrary limits on the size of records that can be stored. For example, SDBM and many ODBM or NDBM implementations have a default limit of 1024 bytes for the size of a record. MLDBM can easily exceed these limits when storing large data structures, leading to mysterious failures. Although SDBM_File is used by MLDBM by default, it is not a good choice if you're storing large data structures. Berkeley DB and GDBM both do not have these limits, so I recommend using either of those instead.

2. MLDBM does well with data structures that are not too deep and not too wide. You also need to be careful about how many `FETCH`es your code actually ends up doing. Meaning, you should get the most mileage out of a `FETCH` by holding on to the highest level value for as long as you need it. Remember that every toplevel access of the tied hash, for example `$mldb{foo}`, translates to a MLDBM `FETCH()` call.

   Too often, people end up writing something like this:

   ```
   tie %h, 'MLDBM', ...;
   for my $k (keys %{$h{something}}) {
       print $h{something}{$k}[0]{foo}{bar};  # FETCH _every_ time!
   }
   ```

   when it should be written this for efficiency:

```
tie %h, 'MLDBM', ...;
my $root = $h{something};                    # FETCH _once_
for my $k (keys %$root) {
    print $k->[0]{foo}{bar};
}
```

AUTHORS

Gurusamy Sarathy <*gsar AT umich DOT edu*>.

Support for multiple serializing packages by Raphael Manfredi <*Raphael_Manfredi AT grenoble DOT hp DOT com*>.

Test suite fixes for perl 5.8.0 done by Josh Chamas.

VERSION

Version 2.05

SEE ALSO

perl (1), *perltie* (1), *perlfunc* (1), Data::Dumper, FreezeThaw, Storable, DBM::Deep, MLDBM::Serializer::JSON.