NAME

Hash::Util::FieldHash - Support for Inside-Out Classes

SYNOPSIS

```
### Create fieldhashes
use Hash::Util qw(fieldhash fieldhashes);

# Create a single field hash
fieldhash my %foo;

# Create three at once...
fieldhashes \ my(%foo, %bar, %baz);
# ...or any number
fieldhashes @hashrefs;

### Create an idhash and register it for garbage collection
use Hash::Util::FieldHash qw(idhash register);
idhash my %name;
my $object = \ do { my $o };
# register the idhash for garbage collection with $object
register($object, \ %name);
# the following entry will be deleted when $object goes out of scope
$name{$object} = 'John Doe';

### Register an ordinary hash for garbage collection
use Hash::Util::FieldHash qw(id register);
my %name;
my $object = \ do { my $o };
# register the hash %name for garbage collection of $object's id
register $object, \ %name;
# the following entry will be deleted when $object goes out of scope
$name{id $object} = 'John Doe';
```

FUNCTIONS

Hash::Util::FieldHash offers a number of functions in support of "The Inside-out Technique" of class construction.

id

```
id($obj)
```

Returns the reference address of a reference `$obj`. If `$obj` is not a reference, returns `$obj`.

This function is a stand-in replacement for Scalar::Util::refaddr, that is, it returns the reference address of its argument as a numeric value. The only difference is that `refaddr()` returns `undef` when given a non-reference while `id()` returns its argument unchanged.

`id()` also uses a caching technique that makes it faster when the id of an object is requested often, but slower if it is needed only once or twice.

id_2obj

```
$obj = id_2obj($id)
```

If `$id` is the id of a registered object (see "register"), returns the object, otherwise an undefined value. For registered objects this is the inverse function of `id()`.

register

```
register($obj)
register($obj, @hashrefs)
```

In the first form, registers an object to work with for the function `id_2obj()`. In the second

form, it additionally marks the given hashrefs down for garbage collection. This means that when the object goes out of scope, any entries in the given hashes under the key of `id($obj)` will be deleted from the hashes.

It is a fatal error to register a non-reference `$obj`. Any non-hashrefs among the following arguments are silently ignored.

It is *not* an error to register the same object multiple times with varying sets of hashrefs. Any hashrefs that are not registered yet will be added, others ignored.

Registry also implies thread support. When a new thread is created, all references are replaced with new ones, including all objects. If a hash uses the reference address of an object as a key, that connection would be broken. With a registered object, its id will be updated in all hashes registered with it.

idhash

```
idhash my %hash
```

Makes an idhash from the argument, which must be a hash.

An *idhash* works like a normal hash, except that it stringifies a *reference used as a key* differently. A reference is stringified as if the `id()` function had been invoked on it, that is, its reference address in decimal is used as the key.

idhashes

```
idhashes \ my(%hash, %gnash, %trash)
idhashes \ @hashrefs
```

Creates many idhashes from its hashref arguments. Returns those arguments that could be converted or their number in scalar context.

fieldhash

```
fieldhash %hash;
```

Creates a single fieldhash. The argument must be a hash. Returns a reference to the given hash if successful, otherwise nothing.

A *fieldhash* is, in short, an idhash with auto-registry. When an object (or, indeed, any reference) is used as a fieldhash key, the fieldhash is automatically registered for garbage collection with the object, as if `register $obj, \ %fieldhash` had been called.

fieldhashes

```
fieldhashes @hashrefs;
```

Creates any number of field hashes. Arguments must be hash references. Returns the converted hashrefs in list context, their number in scalar context.

DESCRIPTION

A word on terminology: I shall use the term *field* for a scalar piece of data that a class associates with an object. Other terms that have been used for this concept are ‘‘object variable’’, ‘‘(object) property’’, ‘‘(object) attribute’’ and more. Especially ‘‘attribute’’ has some currency among Perl programmer, but that clashes with the `attributes` pragma. The term ‘‘field’’ also has some currency in this sense and doesn't seem to conflict with other Perl terminology.

In Perl, an object is a blessed reference. The standard way of associating data with an object is to store the data inside the object's body, that is, the piece of data pointed to by the reference.

In consequence, if two or more classes want to access an object they *must* agree on the type of reference and also on the organization of data within the object body. Failure to agree on the type results in immediate death when the wrong method tries to access an object. Failure to agree on data organization may lead to one class trampling over the data of another.

This object model leads to a tight coupling between subclasses. If one class wants to inherit from another (and both classes access object data), the classes must agree about implementation

details. Inheritance can only be used among classes that are maintained together, in a single source or not.

In particular, it is not possible to write general-purpose classes in this technique, classes that can advertise themselves as "Put me on your `@ISA` list and use my methods". If the other class has different ideas about how the object body is used, there is trouble.

For reference `Name_hash` in "Example 1" shows the standard implementation of a simple class `Name` in the well-known hash based way. It also demonstrates the predictable failure to construct a common subclass `NamedFile` of `Name` and the class `IO::File` (whose objects *must* be globrefs).

Thus, techniques are of interest that store object data *not* in the object body but some other place.

The Inside-out Technique

With *inside-out* classes, each class declares a (typically lexical) hash for each field it wants to use. The reference address of an object is used as the hash key. By definition, the reference address is unique to each object so this guarantees a place for each field that is private to the class and unique to each object. See `Name_id` in "Example 1" for a simple example.

In comparison to the standard implementation where the object is a hash and the fields correspond to hash keys, here the fields correspond to hashes, and the object determines the hash key. Thus the hashes appear to be turned *inside out*.

The body of an object is never examined by an inside-out class, only its reference address is used. This allows for the body of an actual object to be *anything at all* while the object methods of the class still work as designed. This is a key feature of inside-out classes.

Problems of Inside-out

Inside-out classes give us freedom of inheritance, but as usual there is a price.

Most obviously, there is the necessity of retrieving the reference address of an object for each data access. It's a minor inconvenience, but it does clutter the code.

More important (and less obvious) is the necessity of garbage collection. When a normal object dies, anything stored in the object body is garbage-collected by perl. With inside-out objects, Perl knows nothing about the data stored in field hashes by a class, but these must be deleted when the object goes out of scope. Thus the class must provide a `DESTROY` method to take care of that.

In the presence of multiple classes it can be non-trivial to make sure that every relevant destructor is called for every object. Perl calls the first one it finds on the inheritance tree (if any) and that's it.

A related issue is thread-safety. When a new thread is created, the Perl interpreter is cloned, which implies that all reference addresses in use will be replaced with new ones. Thus, if a class tries to access a field of a cloned object its (cloned) data will still be stored under the now invalid reference address of the original in the parent thread. A general `CLONE` method must be provided to re-establish the association.

Solutions

`Hash::Util::FieldHash` addresses these issues on several levels.

The `id()` function is provided in addition to the existing `Scalar::Util::refaddr()`. Besides its short name it can be a little faster under some circumstances (and a bit slower under others). Benchmark if it matters. The working of `id()` also allows the use of the class name as a *generic object* as described further down.

The `id()` function is incorporated in *id hashes* in the sense that it is called automatically on every key that is used with the hash. No explicit call is necessary.

The problems of garbage collection and thread safety are both addressed by the function `register()`. It registers an object together with any number of hashes. Registry means that when the object dies, an entry in any of the hashes under the reference address of this object will be

deleted. This guarantees garbage collection in these hashes. It also means that on thread cloning the object's entries in registered hashes will be replaced with updated entries whose key is the cloned object's reference address. Thus the object-data association becomes thread-safe.

Object registry is best done when the object is initialized for use with a class. That way, garbage collection and thread safety are established for every object and every field that is initialized.

Finally, *field hashes* incorporate all these functions in one package. Besides automatically calling the `id()` function on every object used as a key, the object is registered with the field hash on first use. Classes based on field hashes are fully garbage-collected and thread safe without further measures.

More Problems

Another problem that occurs with inside-out classes is serialization. Since the object data is not in its usual place, standard routines like `Storable::freeze()`, `Storable::thaw()` and `Data::Dumper::Dumper()` can't deal with it on their own. Both `Data::Dumper` and `Storable` provide the necessary hooks to make things work, but the functions or methods used by the hooks must be provided by each inside-out class.

A general solution to the serialization problem would require another level of registry, one that associates *classes* and fields. So far, the functions of `Hash::Util::FieldHash` are unaware of any classes, which I consider a feature. Therefore `Hash::Util::FieldHash` doesn't address the serialization problems.

The Generic Object

Classes based on the `id()` function (and hence classes based on `idhash()` and `fieldhash()`) show a peculiar behavior in that the class name can be used like an object. Specifically, methods that set or read data associated with an object continue to work as class methods, just as if the class name were an object, distinct from all other objects, with its own data. This object may be called the *generic object* of the class.

This works because field hashes respond to keys that are not references like a normal hash would and use the string offered as the hash key. Thus, if a method is called as a class method, the field hash is presented with the class name instead of an object and blithely uses it as a key. Since the keys of real objects are decimal numbers, there is no conflict and the slot in the field hash can be used like any other. The `id()` function behaves correspondingly with respect to non-reference arguments.

Two possible uses (besides ignoring the property) come to mind. A singleton class could be implemented this using the generic object. If necessary, an `init()` method could die or ignore calls with actual objects (references), so only the generic object will ever exist.

Another use of the generic object would be as a template. It is a convenient place to store class-specific defaults for various fields to be used in actual object initialization.

Usually, the feature can be entirely ignored. Calling *object methods* as *class methods* normally leads to an error and isn't used routinely anywhere. It may be a problem that this error isn't indicated by a class with a generic object.

How to use Field Hashes

Traditionally, the definition of an inside-out class contains a bare block inside which a number of lexical hashes are declared and the basic accessor methods defined, usually through `Scalar::Util::refaddr` Further methods may be defined outside this block. There has to be a DESTROY method and, for thread support, a CLONE method.

When field hashes are used, the basic structure remains the same. Each lexical hash will be made a field hash. The call to `refaddr` can be omitted from the accessor methods. DESTROY and CLONE methods are not necessary.

If you have an existing inside-out class, simply making all hashes field hashes with no other change should make no difference. Through the calls to `refaddr` or equivalent, the field hashes never get to see a reference and work like normal hashes. Your DESTROY (and CLONE) methods

are still needed.

To make the field hashes kick in, it is easiest to redefine `refaddr` as

```
sub refaddr { shift }
```

instead of importing it from `Scalar::Util` It should now be possible to disable DESTROY and CLONE. Note that while it isn't disabled, DESTROY will be called before the garbage collection of field hashes, so it will be invoked with a functional object and will continue to function.

It is not desirable to import the functions `fieldhash` and/or `fieldhashes` into every class that is going to use them. They are only used once to set up the class. When the class is up and running, these functions serve no more purpose.

If there are only a few field hashes to declare, it is simplest to

```
use Hash::Util::FieldHash;
```

early and call the functions qualified:

```
Hash::Util::FieldHash::fieldhash my %foo;
```

Otherwise, import the functions into a convenient package like `HUF` or, more general, `Aux`

```
{
package Aux;
use Hash::Util::FieldHash ':all';
}
```

and call

```
Aux::fieldhash my %foo;
```

as needed.

Garbage-Collected Hashes

Garbage collection in a field hash means that entries will ''spontaneously'' disappear when the object that created them disappears. That must be borne in mind, especially when looping over a field hash. If anything you do inside the loop could cause an object to go out of scope, a random key may be deleted from the hash you are looping over. That can throw the loop iterator, so it's best to cache a consistent snapshot of the keys and/or values and loop over that. You will still have to check that a cached entry still exists when you get to it.

Garbage collection can be confusing when keys are created in a field hash from normal scalars as well as references. Once a reference is *used* with a field hash, the entry will be collected, even if it was later overwritten with a plain scalar key (every positive integer is a candidate). This is true even if the original entry was deleted in the meantime. In fact, deletion from a field hash, and also a test for existence constitute *use* in this sense and create a liability to delete the entry when the reference goes out of scope. If you happen to create an entry with an identical key from a string or integer, that will be collected instead. Thus, mixed use of references and plain scalars as field hash keys is not entirely supported.

EXAMPLES

The examples show a very simple class that implements a *name*, consisting of a first and last name (no middle initial). The name class has four methods:

- `init()`

  An object method that initializes the first and last name to its two arguments. If called as a class method, `init()` creates an object in the given class and initializes that.

- `first()`

  Retrieve the first name

- `last()`

Retrieve the last name

- `name()`

  Retrieve the full name, the first and last name joined by a blank.

The examples show this class implemented with different levels of support by `Hash::Util::FieldHash` All supported combinations are shown. The difference between implementations is often quite small. The implementations are:

- `Name_hash`

  A conventional (not inside-out) implementation where an object is a hash that stores the field values, without support by `Hash::Util::FieldHash` This implementation doesn't allow arbitrary inheritance.

- `Name_id`

  Inside-out implementation based on the `id()` function. It needs a `DESTROY` method. For thread support a `CLONE` method (not shown) would also be needed. Instead of `Hash::Util::FieldHash::id()` the function `Scalar::Util::refaddr` could be used with very little functional difference. This is the basic pattern of an inside-out class.

- `Name_idhash`

  Idhash-based inside-out implementation. Like `Name_id` it needs a `DESTROY` method and would need `CLONE` for thread support.

- `Name_id_reg`

  Inside-out implementation based on the `id()` function with explicit object registry. No destructor is needed and objects are thread safe.

- `Name_idhash_reg`

  Idhash-based inside-out implementation with explicit object registry. No destructor is needed and objects are thread safe.

- `Name_fieldhash`

  FieldHash-based inside-out implementation. Object registry happens automatically. No destructor is needed and objects are thread safe.

These examples are realized in the code below, which could be copied to a file *Example.pm*.

Example 1

```
use strict; use warnings;

{
package Name_hash; # standard implementation: the
# object is a hash
sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless {}, $obj unless ref $obj;
$obj->{ first} = $first;
$obj->{ last} = $last;
$obj;
}

sub first { shift()->{ first} }
sub last { shift()->{ last} }
```

```
sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}

}

{
package Name_id;
use Hash::Util::FieldHash qw(id);

my (%first, %last);

sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless \ my $o, $obj unless ref $obj;
$first{ id $obj} = $first;
$last{ id $obj} = $last;
$obj;
}

sub first { $first{ id shift()} }
sub last { $last{ id shift()} }

sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}

sub DESTROY {
my $id = id shift;
delete $first{ $id};
delete $last{ $id};
}

}

{
package Name_idhash;
use Hash::Util::FieldHash;

Hash::Util::FieldHash::idhashes( \ my (%first, %last) );

sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless \ my $o, $obj unless ref $obj;
$first{ $obj} = $first;
$last{ $obj} = $last;
$obj;
}
```

```
sub first { $first{ shift()} }
sub last { $last{ shift()} }

sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}

sub DESTROY {
my $n = shift;
delete $first{ $n};
delete $last{ $n};
}

}

{
package Name_id_reg;
use Hash::Util::FieldHash qw(id register);

my (%first, %last);

sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless \ my $o, $obj unless ref $obj;
register( $obj, \ (%first, %last) );
$first{ id $obj} = $first;
$last{ id $obj} = $last;
$obj;
}

sub first { $first{ id shift()} }
sub last { $last{ id shift()} }

sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}
}

{
package Name_idhash_reg;
use Hash::Util::FieldHash qw(register);

Hash::Util::FieldHash::idhashes \ my (%first, %last);

sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless \ my $o, $obj unless ref $obj;
```

```
register( $obj, \ (%first, %last) );
$first{ $obj} = $first;
$last{ $obj} = $last;
$obj;
}

sub first { $first{ shift()} }
sub last { $last{ shift()} }

sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}
}

{
package Name_fieldhash;
use Hash::Util::FieldHash;

Hash::Util::FieldHash::fieldhashes \ my (%first, %last);

sub init {
my $obj = shift;
my ($first, $last) = @_;
# create an object if called as class method
$obj = bless \ my $o, $obj unless ref $obj;
$first{ $obj} = $first;
$last{ $obj} = $last;
$obj;
}

sub first { $first{ shift()} }
sub last { $last{ shift()} }

sub name {
my $n = shift;
join ' ' => $n->first, $n->last;
}
}

1;
```

To exercise the various implementations the script below can be used.

It sets up a class `Name` that is a mirror of one of the implementation classes `Name_hash`, `Name_id`, ..., `Name_fieldhash`. That determines which implementation is run.

The script first verifies the function of the `Name` class.

In the second step, the free inheritability of the implementation (or lack thereof) is demonstrated. For this purpose it constructs a class called `NamedFile` which is a common subclass of `Name` and the standard class `IO::File` This puts inheritability to the test because objects of `IO::File` *must* be globrefs. Objects of `NamedFile` should behave like a file opened for reading and also support the `name()` method. This class juncture works with exception of the `Name_hash` implementation, where object initialization fails because of the incompatibility of object bodies.

Example 2

```
use strict; use warnings; $| = 1;

use Example;

{
package Name;
use parent 'Name_id'; # define here which implementation to run
}


# Verify that the base package works
my $n = Name->init(qw(Albert Einstein));
print $n->name, "\n";
print "\n";

# Create a named file handle (See definition below)
my $nf = NamedFile->init(qw(/tmp/x Filomena File));
# use as a file handle...
for ( 1 .. 3 ) {
my $l = <$nf>;
print "line $_: $l";
}
# ...and as a Name object
print "...brought to you by ", $nf->name, "\n";
exit;


# Definition of NamedFile
package NamedFile;
use parent 'Name';
use parent 'IO::File';

sub init {
my $obj = shift;
my ($file, $first, $last) = @_;
$obj = $obj->IO::File::new() unless ref $obj;
$obj->open($file) or die "Can't read '$file': $!";
$obj->Name::init($first, $last);
}
__END__
```

GUTS

To make Hash::Util::FieldHash work, there were two changes to *perl* itself. PERL_MAGIC_uvar was made available for hashes, and weak references now call uvar `get` magic after a weakref has been cleared. The first feature is used to make field hashes intercept their keys upon access. The second one triggers garbage collection.

The PERL_MAGIC_uvar interface for hashes

PERL_MAGIC_uvar *get* magic is called from `hv_fetch_common` and `hv_delete_common` through the function `hv_magic_uvar_xkey`, which defines the interface. The call happens for hashes with "uvar" magic if the `ufuncs` structure has equal values in the `uf_val` and `uf_set` fields. Hashes are unaffected if (and as long as) these fields hold different values.

Upon the call, the `mg_obj` field will hold the hash key to be accessed. Upon return, the `SV*` value in `mg_obj` will be used in place of the original key in the hash access. The integer index value in

the first parameter will be the `action` value from `hv_fetch_common`, or -1 if the call is from `hv_delete_common`.

This is a template for a function suitable for the `uf_val` field in a `ufuncs` structure for this call. The `uf_set` and `uf_index` fields are irrelevant.

```
IV watch_key(pTHX_ IV action, SV* field) {
MAGIC* mg = mg_find(field, PERL_MAGIC_uvar);
SV* keysv = mg->mg_obj;
/* Do whatever you need to. If you decide to
supply a different key newkey, return it like this
*/
sv_2mortal(newkey);
mg->mg_obj = newkey;
return 0;
}
```

Weakrefs call uvar magic

When a weak reference is stored in an `SV` that has ''uvar'' magic, `set` magic is called after the reference has gone stale. This hook can be used to trigger further garbage-collection activities associated with the referenced object.

How field hashes work

The three features of key hashes, *key replacement*, *thread support*, and *garbage collection* are supported by a data structure called the *object registry*. This is a private hash where every object is stored. An ''object'' in this sense is any reference (blessed or unblessed) that has been used as a field hash key.

The object registry keeps track of references that have been used as field hash keys. The keys are generated from the reference address like in a field hash (though the registry isn't a field hash). Each value is a weak copy of the original reference, stored in an `SV` that is itself magical (`PERL_MAGIC_uvar` again). The magical structure holds a list (another hash, really) of field hashes that the reference has been used with. When the weakref becomes stale, the magic is activated and uses the list to delete the reference from all field hashes it has been used with. After that, the entry is removed from the object registry itself. Implicitly, that frees the magic structure and the storage it has been using.

Whenever a reference is used as a field hash key, the object registry is checked and a new entry is made if necessary. The field hash is then added to the list of fields this reference has used.

The object registry is also used to repair a field hash after thread cloning. Here, the entire object registry is processed. For every reference found there, the field hashes it has used are visited and the entry is updated.

Internal function Hash::Util::FieldHash::_fieldhash

```
# test if %hash is a field hash
my $result = _fieldhash \ %hash, 0;

# make %hash a field hash
my $result = _fieldhash \ %hash, 1;
```

`_fieldhash` is the internal function used to create field hashes. It takes two arguments, a hashref and a mode. If the mode is boolean false, the hash is not changed but tested if it is a field hash. If the hash isn't a field hash the return value is boolean false. If it is, the return value indicates the mode of field hash. When called with a boolean true mode, it turns the given hash into a field hash of this mode, returning the mode of the created field hash. `_fieldhash` does not erase the given hash.

Currently there is only one type of field hash, and only the boolean value of the mode makes a difference, but that may change.

AUTHOR

Anno Siegel (ANNO) wrote the xs code and the changes in perl proper Jerry Hedden (JDHEDDEN) made it faster

COPYRIGHT AND LICENSE