perltie

Table des matières

1 NAME/NOM 1

2 SYNOPSIS 1

3 DESCRIPTION 1
- 3.1 Les scalaires liés (par tie()) . . . 2
- 3.2 Les tableaux liés (par tie()) . . . 4
- 3.3 Les tables de hachage liées (par tie()) . . . 7
- 3.4 Les descripteurs de fichiers (filehandles) liés (par tie()) . . . 11
- 3.5 Les pièges du `untie` . . . 13

4 VOIR AUSSI 15

5 BUGS 15

6 AUTEURS 15

7 TRADUCTION 16
- 7.1 Version . . . 16
- 7.2 Traducteur . . . 16
- 7.3 Relecture . . . 16

8 À propos de ce document 16

1 NAME/NOM

perltie - Comment cacher un objet d'une classe derrière une simple variable

2 SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST

$object = tied VARIABLE

untie VARIABLE
```

3 DESCRIPTION

Avant la version 5.0 de Perl, un programmeur pouvait utiliser la fonction dbmopen() pour connecter magiquement une table de hachage %HASH de son programme à une base de données sur disque au format standard Unix dbm(3x). En revanche, son Perl était construit avec l'une ou l'autre des implémentations de la bibliothèque dbm mais pas les deux et il n'était pas possible d'étendre ce mécanisme à d'autres packages ou à d'autres types de variables.

C'est maintenant possible.

La fonction tie() relie une variable à une classe (ou un package) qui fournit l'implémentation des méthodes permettant d'accéder à cette variable. Une fois ce lien magique établit, l'accès à cette variable déclenche automatiquement l'appel aux méthodes de la classe choisie. La complexité de la classe est cachée derrière des appels magiques de méthodes. Les noms de ces méthodes sont entièrement en MAJUSCULES. C'est une convention pour signifier que Perl peut appeler ces méthodes implicitement plutôt qu'explicitement – exactement comme les fonctions BEGIN() et END().

Dans l'appel de tie(), VARIABLE est le nom de la variable à relier. CLASSNAME est le nom de la classe qui implémente les objets du bon type. Tous les autres arguments dans LIST sont passés au constructeur approprié pour cette classe – à savoir TIESCALAR(), TIEARRAY(), TIEHASH(), ou TIEHANDLE(). (Ce sont par exemple les arguments à passer à la fonction C dbminit().) L'objet retourné par la méthode "new" est aussi retourné par la fonction tie() ce qui est pratique pour accéder à d'autres méthodes de la classe CLASSNAME. (Vous n'avez pas besoin de retourner une référence vers un vrai "type" (e.g. HASH ou CLASSNAME) tant que c'est un objet correctement béni – par la fonction bless().) Vous pouvez aussi retrouver une référence vers l'objet sous-jacent en utilisant la fonction tied().

À la différence de dbmopen(), la fonction tie() ne fera pas pour vous un use ou un require d'un module – vous devez le faire vous-même explicitement.

3.1 Les scalaires liés (par tie())

Une classe qui implémente un scalaire lié devrait définir les méthodes suivantes : TIESCALAR, FETCH, STORE et éventuellement UNTIE et DESTROY.

Jetons un oeil à chacune d'elle en utilisant comme exemple une classe liée (par tie()) qui permet à l'utilisateur de faire :

```
tie $his_speed, 'Nice', getppid();
tie $my_speed,  'Nice', $$;
```

À partir de maintenant, à chaque accès à l'une de ces variables, la priorité courante du système est retrouvée et retournée. Si ces variables sont modifiées alors la priorité du process change.

Nous utilisons la classe BSD::Resource de Jarkko Hietaniemi <*jhi@iki.fi*> (non fournie) pour accéder aux constantes systèmes PRIO_PROCESS, PRIO_MIN, et PRIO_MAX ainsi qu'aux appels systèmes getpriority() et setpriority(). Voici le préambule de la classe :

```
package Nice;
use Carp;
use BSD::Resource;
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

TIESCALAR classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourné une référence bénie (par bless()) vers un nouveau scalaire (probablement anonyme) qu'il a créé. Par exemple :

```
sub TIESCALAR {
    my $class = shift;
    my $pid = shift || $$; # 0 means me

    if ($pid !~ /^\d+$/) {
        carp "Nice::Tie::Scalar got non-numeric pid $pid" if $^W;
        return undef;
    }

    unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::Tie::Scalar got bad pid $pid: $!" if $^W;
        return undef;
    }

    return bless \$pid, $class;
}
```

Cette classe liée a choisi de retourner une erreur plutôt que de lancer une exception si son constructeur échoue. Bien que ce soit la manière dont dbmopen() fonctionne, d'autres classes peuvent très bien être moins tolérantes. La classe regarde tout de même la variable $^W pour savoir si elle doit émettre un peu de bruit ou non.

FETCH this

Cette méthode se déclenche chaque fois qu'on accède (en lecture) à une variable liée. Elle ne prend aucun argument mis à part une référence qui est l'objet qui représente le scalaire à utiliser. Puisque dans notre cas nous utilisons une simple référence SCALAIRE comme objet du scalaire lié, un appel à $$self permet à la méthode d'obtenir la valeur réelle stockée. Dans notre exemple ci-dessous, cette valeur réelle est l'ID du process que nous avons lié à notre variable.

```
sub FETCH {
    my $self = shift;
    confess "wrong type" unless ref $self;
    croak "usage error" if @_;
    my $nicety;
    local($!) = 0;
    $nicety = getpriority(PRIO_PROCESS, $$self);
    if ($!) { croak "getpriority failed: $!" }
    return $nicety;
}
```

Cette fois, nous avons décidé d'exploser (en générant une exception) si l'obtention de la priorité échoue – il n'y a aucun autre moyen pour retourner une erreur et c'est probablement la meilleure chose à faire.

STORE this, value

Cette méthode est appelée à chaque fois que la variable liée est modifiée (affectée). Derrière sa propre référence, elle attend un (et un seul) argument : la nouvelle valeur que l'utilisateur essaye d'affecter. Ne vous préoccupez pas de la valeur retournée par STORE : le fait qu'une affectation retourne la valeur qui vient d'être affectée est implémenté en passant par FETCH.

```
sub STORE {
    my $self = shift;
    confess "wrong type" unless ref $self;
    my $new_nicety = shift;
    croak "usage error" if @_;

    if ($new_nicety < PRIO_MIN) {
        carp sprintf
          "WARNING: priority %d less than minimum system priority %d",
              $new_nicety, PRIO_MIN if $^W;
        $new_nicety = PRIO_MIN;
    }

    if ($new_nicety > PRIO_MAX) {
        carp sprintf
          "WARNING: priority %d greater than maximum system priority %d",
              $new_nicety, PRIO_MAX if $^W;
        $new_nicety = PRIO_MAX;
    }

    unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
        confess "setpriority failed: $!";
    }
}
```

UNTIE this

Cette méthode sera appelée lorsqu'un `untie` aura lieu. Cela peut être pratique pour la classe a besoin de savoir qu'on ne l'appellera plus (sauf via DESTROY évidemment). Voir Les pièges du `untie` ci-dessous pour plus de détails.

DESTROY this

Cette méthode est appelée lorsque la variable liée doit être détruite. Comme pour les autres classes d'objets, une telle méthode est rarement nécessaire parce que Perl désalloue automatiquement pour vous la mémoire associé à des objets moribonds – ce n'est pas le cas de C++. Nous utilisons donc une méthode DESTROY ici uniquement pour débugguer.

```
sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    carp "[ Nice::DESTROY pid $$self ]" if $Nice::DEBUG;
}
```

C'est tout ce qu'il y a à voir. C'est même un peu plus parce que nous avons fait des jolies petites choses dans un souci de complétude, robustesse et, plus généralement, d'esthétique. Des classes plus simples liant un scalaire sont certainement faisables.

3.2 Les tableaux liés (par tie())

Une classe qui implémente un tableau lié ordinaire devrait définir les méthodes suivantes : TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE et éventuellement UNTIE et DESTROY.

Les méthodes FETCHSIZE et STORESIZE sont nécessaires pour pouvoir fournir `$#array` et autres fonctionnalités équivalentes comme `scalar(@array)`.

Les méthodes POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE et EXISTS sont indispensables si l'opérateur Perl ayant le même nom (mais en minuscules) doit être appliqué aux tableaux liés. La classe **Tie::Array** peut être utilisée comme classe de base pour implémenter ces méthodes à partir des cinq méthodes initiales de base. Les implémentations par défaut de DELETE et de EXISTS dans **Tie::Array** appellent simplement `croak`.

De plus, la méthode EXTEND sera appelée quand perl devra pré-étendre l'allocation du tableau.

Pour illustrer cela, nous allons implémenter un tableau dont la longueur des éléments est constante et fixée lors de la création du tableau. Si on tente de créer un élément plus grand que cette limite, cela produira une exception. Par exemple :

```
use FixedElem_Array;
tie @array, 'FixedElem_Array', 3;
$array[0] = 'cat';  # ok.
$array[1] = 'dogs'; # exception, length('dogs') > 3.
```

Le code en préambule de cette classe est le suivant :

```
package FixedElem_Array;
use Carp;
use strict;
```

TIEARRAY classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par bless()) à travers laquelle on pourra accéder au nouveau tableau (probablement une référence à un TABLEAU anonyme)

Dans notre exemple, juste pour montrer qu'il n'est pas *VRAIMENT* nécessaire de retourner une référence vers un TABLEAU, nous avons choisi une référence à une HASHTABLE pour représenter notre objet. Cette HASHTABLE s'élabore exactement comme un type d'enregistrement générique : le champ `{ELEMSIZE}` stockera la taille maximale autorisée et le champ `{ARRAY}` contiendra la référence vers le vrai TABLEAU. Si quelqu'un en dehors de la classe tente de déréférencer l'objet retourné (croyant sans doute avoir à faire à une référence vers un TABLEAU), cela ne fonctionnera pas. Tout cela pour vous montrer que vous devriez respecter le vie privée des objets.

```
sub TIEARRAY {
  my $class = shift;
  my $elemsize = shift;
  if ( @_ || $elemsize =~ /\D/ ) {
    croak "usage: tie ARRAY, '" . __PACKAGE__ . "', elem_size";
  }
  return bless {
    ELEMSIZE => $elemsize,
    ARRAY    => [],
  }, $class;
}
```

FETCH this, indice

Cette méthode est appelée à chaque fois qu'on accède (en lecture) à un élément individuel d'un tableau lié. Elle prend un argument en plus de sa propre référence : l'indice de la valeur à laquelle on veut accéder.

```
sub FETCH {
  my $self  = shift;
  my $indice = shift;
  return $self->{ARRAY}->[$indice];
}
```

Si un indice négatif est utilisé par lire dans un tableau, l'indice est converti de manière interne en une valeur positive en appelant d'abord FETCHSIZE avant de passer l'indice résultant à FETCH. Vous pouvez désactiver cette fonctionnalité en affectant une valeur vrai à la variable `$NEGATIVE_INDICES` de la classe du tableau lié.

Comme vous avez pu le remarquer le nom de la méthode FETCH (et al.) est le même pour tous les accès bien que les constructeurs ont des noms différents (TIESCALAR vs TIEARRAY). En théorie, une même classe peut servir pour différents types d'objets liés. En pratique, cela devient rapidement encombrant et il est beaucoup plus simple de n'avoir qu'un seul type d'objets liés par classe.

STORE this, indice, value

Cette méthode est appelée à chaque fois qu'un élément d'un tableau lié est modifié (affecté). Elle prend deux arguments en plus de sa propre référence : l'indice de l'élément où nous voulons stocker quelque chose et la valeur que nous voulons stocker.

Dans notre exemple, `undef` est en fait un nombre d'espaces égale à `$self->{ELEMSIZE}`. Nous avons donc un peu de travail à faire ici...

```
sub STORE {
  my $self = shift;
  my( $indice, $value ) = @_;
  if ( length $value > $self->{ELEMSIZE} ) {
    croak "la longueur de $value est superieure a $self->{ELEMSIZE}";
  }
  # remplir les trous
  $self->EXTEND( $indice ) if $indice > $self->FETCHSIZE();
  # aligner à droite pour les éléments les plus petits
  $self->{ARRAY}->[$indice] = sprintf "%$self->{ELEMSIZE}s", $value;
}
```

Les indice négatifs sont traités de la même manière qu'avec FETCH.

FETCHSIZE this `FETCHSIZE`

Retourne le nombre total d'éléments présents dans le tableau associé à l'objet *this*. (Similaire à `scalar(@array)`.) Par exemple :

```
sub FETCHSIZE {
  my $self = shift;
  return scalar @{$self->{ARRAY}};
}
```

STORESIZE this, count

Fixe à *count* le nombre total d'éléments stockés dans le tableau associé à l'objet *this*. Si cela aggrandit le tableau, la valeur `undef` proposée par la classe devrait être retournée pour les nouveaux emplacements créés. Si le tableau diminue alors les éléments en trop sont supprimés.

Dans notre exemple, la valeur 'undef' est en fait un suite de `$self->{ELEMSIZE}` espaces. Ce qui donne :

```
sub STORESIZE {
  my $self  = shift;
  my $count = shift;
  if ( $count > $self->FETCHSIZE() ) {
    foreach ( $count - $self->FETCHSIZE() .. $count ) {
      $self->STORE( $_, '' );
    }
  } elsif ( $count < $self->FETCHSIZE() ) {
    foreach ( 0 .. $self->FETCHSIZE() - $count - 2 ) {
      $self->POP();
    }
  }
}
```

EXTEND this, count

C'est un appel informatif permettant d'indiquer que le nombre d'éléments du tableau risque d'atteindre *count*. Ça peut servir pour optimiser l'utilisation de la mémoire. Cette méthode n'est pas obligée de faire quelque chose.

Dans notre exemple, nous voulons être sûr qu'il n'y a pas d'éléments vide (ou `undef`). Donc `EXTEND` fait appel à `STORESIZE` pour remplir les éléments comme il faut :

```
sub EXTEND {
  my $self  = shift;
  my $count = shift;
  $self->STORESIZE( $count );
}
```

EXISTS this, key

Permet de vérifier si l'élément dont l'indice est *key* existe dans le tableau lié à *this*.

Dans notre existe, nous décidons qu'un élément constitué uniquement de `$self->{ELEMSIZE}` espcaes n'existe pas :

```
sub EXISTS {
  my $self  = shift;
  my $index = shift;
  return 0 if ! defined $self->{ARRAY}->[$index] ||
              $self->{ARRAY}->[$index] eq ' ' x $self->{ELEMSIZE};
  return 1;
}
```

DELETE this, key

Efface l'élément du tableau lié à *this* dont l'indice est *key*.

Dans notre exemple, un élément effacé contient `$self->{ELEMSIZE}` espaces :

```
sub DELETE {
  my $self  = shift;
  my $index = shift;
  return $self->STORE( $index, '' );
}
```

CLEAR this

Efface (supprime, détruit, ...) tous les éléments stockés dans le tableau lié à l'objet *this*. Par exemple :

```
sub CLEAR {
  my $self = shift;
  return $self->{ARRAY} = [];
}
```

PUSH this, LIST

Ajoute tous les éléments de *LIST* au tableau. Par exemple :

```
sub PUSH {
  my $self = shift;
  my @list = @_;
  my $last = $self->FETCHSIZE();
  $self->STORE( $last + $_, $list[$_] ) foreach 0 .. $#list;
  return $self->FETCHSIZE();
}
```

POP this

Supprime et retourne le dernier élément du tableau. Par exemple :

```
sub POP {
  my $self = shift;
  return pop @{$self->{ARRAY}};
}
```

SHIFT this

Supprime et retourne le premier élément du tableau (en décalant les éléments suivants). Par exemple :

```
sub SHIFT {
  my $self = shift;
  return shift @{$self->{ARRAY}};
}
```

UNSHIFT this, LIST

Insère tous les éléments de *LIST* au début du tableau en décalant les éléments existants pour faire de la place. Par exemple :

```
sub UNSHIFT {
  my $self = shift;
  my @list = @_;
  my $size = scalar( @list );
  # on fait de la place pour la liste
  @{$self->{ARRAY}}[ $size .. $#{$self->{ARRAY}} + $size ]
   = @{$self->{ARRAY}};
  $self->STORE( $_, $list[$_] ) foreach 0 .. $#list;
}
```

SPLICE this, offset, length, LIST

Réalise l'équivalent d'un `splice` sur le tableau.

offset est optionnel et vaut zéro par défaut. Les valeurs négatives sont comptés à partir de la fin du tableau.

length est optionnel et, par défaut, va jusqu'à la fin du tableau.

LIST peut être vide.

Retourne la liste des éléments originaux débutant à *offset* et de longueur *length*.

Dans notre exemple, on utilise un petit raccourci si il y a une *LIST* :

```
sub SPLICE {
  my $self   = shift;
  my $offset = shift || 0;
  my $length = shift || $self->FETCHSIZE() - $offset;
  my @list   = ();
  if ( @_ ) {
    tie @list, __PACKAGE__, $self->{ELEMSIZE};
    @list   = @_;
  }
  return splice @{$self->{ARRAY}}, $offset, $length, @list;
}
```

UNTIE this

Sera appelée lorsqu'un `untie` se produira. (Voir Les pièges du `untie` ci-dessous.)

DESTROY this

Cette méthode est appelée lorsque qu'une variable liée doit être détruite. Comme dans le cas des scalaires, cela est rarement nécessaire avec un langage qui a son propre ramasse-miettes. Donc, cette fois nous la laisserons de côté.

3.3 Les tables de hachage liées (par tie())

Les tables de hachage ont été le premier type de données en Perl à pouvoir être lié (voir dbmopen()). Une classe qui implémente une table de hachage liée devrait définir les méthodes suivantes : TIEHASH est le constructeur ; FETCH et STORE accèdent aux paires clés/valeurs. EXISTS indique si une clé est présente dans la table et DELETE en supprime une. CLEAR vide la table en supprimant toutes les paires clé/valeur. FIRSTKEY et NEXTKEY implémentent les fonctions keys() et each() pour parcourir l'ensemble des clés. SCALAR est appelé lorsque la table de hachage liée est évaluée dans un contexte scalaire. UNTIE est appelé lorsqu'un `untie` a lieu, et DESTROY est appelée lorsque la variable liée est détruite.

Si cela vous semble beaucoup, vous pouvez hériter du module standard Tie::StdHash pour la plupart de vos méthodes et ne redéfinir que celles qui vous intéressent. Voir *Tie::Hash* pour plus de détails.

Souvenez-vous que Perl distingue le cas où une clé n'existe pas dans la table de celui où la clé existe mais correspond à une valeur `undef`. Les deux possibilités peuvent être testées via les fonctions `exists()` et `defined()`.

Voici l'exemple relativement intéressant d'une classe liant une table de hachage : cela vous fournit une table de hachage représentant tous les fichiers .* d'un utilisateur. Vous donnez comme index dans la table le nom du fichier (le point en moins) et vous récupérez le contenu de ce fichier. Par exemple :

```
use DotFiles;
tie %dot, 'DotFiles';
if ( $dot{profile} =~ /MANPATH/ ||
     $dot{login}   =~ /MANPATH/ ||
     $dot{cshrc}   =~ /MANPATH/    )
{
    print "you seem to set your MANPATH\n";
}
```

Ou une autre utilisation possible de notre classe liée :

```
tie %him, 'DotFiles', 'daemon';
foreach $f ( keys %him ) {
    printf "daemon dot file %s is size %d\n",
        $f, length $him{$f};
}
```

Dans notre exemple de table de hachage liée DotFiles, nous utilisons une table de hachage normale pour stocker dans l'objet plusieurs champs importants dont le champ `{LIST}` qui apparaît à l'utilisateur comme le contenu réel de la table.

USER
: l'utilisateur pour lequel l'objet représente les fichiers .*

HOME
: l'endroit où se trouve ces fichiers

CLOBBER
: si nous pouvons essayer de modifier ou de supprimer ces fichiers.

LIST
: la table de hachage des noms des fichiers .* et de leur contenu

Voici le début de *Dotfiles.pm* :

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . '()' }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

Dans notre exemple, nous voulons avoir la possibilité d'émettre des informations de debug pour aider au développement. Nous fournissons aussi une fonction pratique pour aider à l'affichage des messages d'avertissements ; whowasi() retourne le nom de la fonction qui l'a appelé.

Voici les méthodes pour la table de hachage DotFiles.

TIEHASH classname, LIST
: C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par bless()) au travers de laquelle on peut accéder au nouvel objet (probablement mais pas nécessairement une table de hachage anonyme).
Voici le constructeur :

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || '';
    croak "usage: @{[&whowasi]} [USER [DOTDIR]]" if @_;
    $user = getpwuid($user) if $user =~ /^\d+$/;
    my $dir = (getpwnam($user))[7]
            || croak "@{[&whowasi]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER    => $user,
        HOME    => $dir,
        LIST    => {},
        CLOBBER => 0,
    };

    opendir(DIR, $dir)
            || croak "@{[&whowasi]}: can't opendir $dir: $!";
    foreach $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\.//;
        $node->{LIST}{$dot} = undef;
    }
    closedir DIR;
    return bless $node, $self;
}
```

Il n'est pas inutile de préciser que, si vous voulez tester un fichier dont le nom est produit par readdir, vous devez la précéder du nom du répertoire en question. Sinon, puisque nous ne faisons pas de chdir() ici, il ne testera sans doute pas le bon fichier.

FETCH this, key

Cette méthode est appelée à chaque fois qu'on accède (en lecture) à un élément d'une table de hachage liée. Elle prend un argument en plus de sa propre référence : la clé d'accès à la valeur que l'on cherche à lire.

Voici le code FETCH pour notre exemple DotFiles.

```
sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{LIST}->{$dot} || -f $file) {
        carp "@{[&whowasi]}: no $dot file" if $DEBUG;
        return undef;
    }

    if (defined $self->{LIST}->{$dot}) {
        return $self->{LIST}->{$dot};
    } else {
        return $self->{LIST}->{$dot} = `cat $dir/.$dot`;
    }
}
```

C'est facile à écrire en lui faisant appeler la commande Unix cat(1) mais ce serait probablement plus portable d'ouvrir le fichier manuellement (et peut-être plus efficace). Bien sûr, puisque les fichiers .* sont un concept Unix, nous ne sommes pas concernés.

STORE this, key, value

Cette méthode est appelée à chaque accès (en écriture) à un élément d'une table de hachage liée. Elle prend deux arguments en plus de sa propre référence : la clé qui nous servira pour stocker quelque chose et la valeur que vous lui associez.

Dans notre exemple DotFiles, nous n'autorisons la réécriture du fichier que dans le cas où la méthode clobber() a été appelée à partir de la référence objet retournée par tie().

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;
    my $file = $self->{HOME} . "/.$dot";
    my $user = $self->{USER};

    croak "@{[&whowasi]}: $file not clobberable"
        unless $self->{CLOBBER};

    open(F, "> $file") || croak "can't open $file: $!";
    print F $value;
    close(F);
}
```

Si quelqu'un veut écraser quelque chose, il doit dire :

```
$ob = tie %daemon_dots, 'daemon';
$ob->clobber(1);
$daemon_dots{signature} = "A true daemon\n";
```

Un autre moyen de mettre la main sur une référence à l'objet sous-jacent est d'utiliser la fonction tied(). Il serait donc aussi possible de dire :

```
tie %daemon_dots, 'daemon';
tied(%daemon_dots)->clobber(1);
```

La méthode clobber est très simple :

```
sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}
```

DELETE this, key

Cette méthode est appelée lorsqu'on supprime un élément d'une table de hachage en utilisant par exemple la fonction delete(). Encore une fois, nous vérifions que nous voulons réellement écraser les fichiers.

```
sub DELETE   {
    carp &whowasi if $DEBUG;

    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}: won't remove file $file"
        unless $self->{CLOBBER};
    delete $self->{LIST}->{$dot};
    my $success = unlink($file);
    carp "@{[&whowasi]}: can't unlink $file: $!" unless $success;
    $success;
}
```

La valeur retournée par DELETE deviendra la valeur retournée par l'appel à delete(). Si vous voulez simuler le comportement normal de delete(), vous devriez retourner ce qu'aurait retourné FETCH pour cette clé. Dans notre exemple, nous avons préféré retourner une valeur qui indique à l'appelant si le fichier a été correctement effacé.

CLEAR this

Cette méthode est appelée lorsque l'ensemble de la table de hachage est effacée, habituellement en lui affectant la liste vide.

Dans notre exemple, cela devrait supprimer tous les fichiers .* de l'utilisateur ! C'est une chose tellement dangereuse que nous exigeons un positionnement de CLOBBER à une valeur supérieure à 1 pour l'autoriser.

```
sub CLEAR    {
    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: won't remove all dot files for $self->{USER}"
        unless $self->{CLOBBER} > 1;
    my $dot;
    foreach $dot ( keys %{$self->{LIST}}) {
        $self->DELETE($dot);
    }
}
```

EXISTS this, key

Cette méthode est appelée lorsque l'utilisateur appelle la fonction exists() sur une table pour une clé particulière. Dans notre exemple, nous cherchons la clé dans la table de hachage `{LIST}` :

```
sub EXISTS   {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    return exists $self->{LIST}->{$dot};
}
```

FIRSTKEY this

Cette méthode est appelée lorsque l'utilisateur commence une itération à travers la table de hachage via un appel à keys() ou each().

```
sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $a = keys %{$self->{LIST}};          # reset each() iterator
    each %{$self->{LIST}}
}
```

NEXTKEY this, lastkey

Cette méthode est appelée lors d'une itération via keys() ou each(). Son second argument est la dernière clé à laquelle on a accédé. C'est pratique si vous voulez faire attention à l'ordre, si vous appelez l'itérateur pour plusieurs séquences à la fois ou si vous ne stockez pas vos données dans une table de hachage réelle.

Dans notre exemple, nous utilisons une vraie table de hachage. Nous pouvons donc faire simplement en accédant tout de même au champ LIST de manière indirecte.

```
sub NEXTKEY  {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return each %{ $self->{LIST} }
}
```

SCALAR this

Cette méthode est appelée lorsqu'on évalue la table de hachage dans un contexte scalaire. Afin d'adopter un comportement similaire à celui des tables de hachage normales, cette méthode devrait retourner une valeur fausse lorsque la table de hachage liée peut être considérée comme étant vide. Si cette méthode n'existe pas, perl tentera d'adopter un comportement compatible et retournera vrai si la table de hachage est en cours d'itération. Si ce n'est pas le cas, FIRSTKEY sera appelé et le résultat sera une valeur fausse si FIRSTKEY retourne une liste vide, et une valeur vraie sinon.

Par contre, **ne** vous reposez **pas** aveuglément sur ce mécanisme en espérant que perl s'en sortira toujours. Par exemple, perl continuera à retourner une valeur vraie si vous videz la table de hachage par une suite d'appels à DELETE. Vous devez donc fournir vous-même une bonne méthode SCALAR sai vous souhaitez une comportement correct dans un contexte scalaire.

Dans notre exemple, il nous suffit d'appeler `scalar` en l'appliquant à la table de hachage sous-jacente référencée par `$self->{LIST}` :

```
sub SCALAR {
  carp &whowasi if $DEBUG;
  my $self = shift;
  return scalar %{ $self->{LIST} }
}
```

UNTIE this

Sera appelée lorsqu'un `untie` se produira. (Voir Les pièges du `untie` ci-dessous.)

DESTROY this

Cette méthode est appelée lorsque une table de hachage liée va disparaître. Vous n'en avez pas réellement besoin sauf pour débugguer ou pour nettoyer quelques donnés auxiliaires. Voici une fonction très simple :

```
sub DESTROY  {
    carp &whowasi if $DEBUG;
}
```

Notez que des fonctions telles que keys() et values() peuvent retourner des listes énormes lorsqu'elles sont utilisées sur de gros objets comme des fichiers DBM. Vous devriez plutôt utiliser la fonction each() pour les parcourir. Exemple :

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

3.4 Les descripteurs de fichiers (filehandles) liés (par tie())

Pour l'instant, ce n'est que partiellement implémenté.

Une classe qui implémente un filehandle lié devrait définir les méthodes suivantes : TIEHANDLE, au moins une méthode parmi PRINT, PRINTF, WRITE, READLINE, GETC, READ, et éventuellement CLOSE, UNTIE et DESTROY. La classe peut aussi fournir : BINMODE, OPEN, EOF, FILENO, SEEK et TELL (si les opérateurs Perl correspondants sont utilisés sur le descripteur).

Lorsque STDERR est lié, c'est sa méthode PRINT qui sera appelée à chaque émission d'un message d'avertissement ou d'erreur. Cette fonctionnalité est temporairement désactivée durant cet appel ce qui signifie que vous pouvez faire appel à `warn()` à l'intérieur de PRINT sans risquer une série d'appels récursifs. Notez aussi que, comme les gestionnaires de signaux `__WARN__` et `__DIE__`, la méthode PRINT de STDERR peut être appelée pour indiquer des erreurs de compilations. Par conséquent, toutes les remarques faites dans %SIG in *perlvar* s'appliquent.

Tout cela est particulièrement pratique lorsque perl est utilisé à l'intérieur d'un autre programme dans lequel STDOUT et STDERR doivent être redirigés d'une manière un peu spéciale. Voir nvi et le module Apache par exemple.

Dans l'exemple suivant, nous essayons de créer un filehandle hurleur (*shouting* en anglais).

```
package Shout;
```

TIEHANDLE classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner un référence bénie (par bless()). Cette référence peut être utilisée pour stocker des informations internes.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

WRITE this, LIST

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `syswrite`.

```
sub WRITE {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

PRINT this, LIST

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `print()`. En plus de sa propre référence, elle attend une liste à passer à la fonction print.

```
sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)),$\ }
```

PRINTF this, LIST

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `printf()`. En plus de sa propre référence, elle attend un format et une liste à passer à la fonction printf.

```
sub PRINTF {
    shift;
    my $fmt = shift;
    print sprintf($fmt, @_);
}
```

READ this, LIST

Cette méthode est appelée lorsque le filehandle est lu via les fonctions `read()` ou `sysread()`.

```
sub READ {
    my $self = shift;
    my $bufref = \$_[0];
    my(undef,$len,$offset) = @_;
    print "READ called, \$buf=$bufref, \$len=$len, \$offset=$offset";
    # add to $$bufref, set $len to number of characters read
    $len;
}
```

READLINE this

Cette méthode est appelée lorsque le filehandle est lu via <HANDLE>. Elle devrait retourner undef lorsque il n'y a plus de données.

```
sub READLINE { $r = shift; "PRINT called $$r times\n"; }
```

GETC this

Cette méthode est appelée lorsque la fonction `getc` est appelée.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

CLOSE this
Cette méthode est appelée lorsque le filehandle est fermé via la fonction `close`.

```
sub CLOSE { print "CLOSE called.\n" }
```

UNTIE this
Comme pour les autres types de liaisons, cette méthode sera appelée lorsqu'un appel à `untie` aura lieu. Il peut être intéressant de faire un "auto CLOSE" à ce moment-là. Voir Les pièges de `untie` ci-dessous.

DESTROY this
Comme pour les autres types de variables liées, cette méthode sera appelée lorsque le filehandle lié va être détruit. C'est pratique pour débugguer et éventuellement nettoyer.

```
sub DESTROY { print "</shout>\n" }
```

Voici comment utiliser notre petit exemple :

```
tie(*FOO,'Shout');
print FOO "hello\n";
$a = 4; $b = 6;
print FOO $a, " plus ", $b, " equals ", $a + $b, "\n";
print <FOO>;
```

3.5 Les pièges du `untie`

Si vous projetez d'utiliser l'objet retourné soit par tie() soit par tied() et si la classe liée définit un destructeur, il y a un piège subtile que vous *devez* connaître.

Comme base, considérons cette exemple d'utilisation de tie; la seule chose qu'il fait est de garder une trace de toutes les valeurs affectées à un scalaire.

```
package Remember;

use strict;
use IO::File;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    my $handle = IO::File->new("> $filename")
                  or die "Cannot open $filename: $!\n";

    print $handle "The Start\n";
    bless {FH => $handle, Value => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{Value};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
    print $handle "$value\n";
    $self->{Value} = $value;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{FH};
    print $handle "The End\n";
    close $handle;
}
```

```
1;
```

Voici un exemple d'utilisation :

```
use strict;
use Remember;

my $fred;
tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

Et voici le résultat de l'exécution :

```
The Start
1
4
5
The End
```

Jusqu'à maintenant, tout va bien. Pour ceux qui ne l'auraient pas remarqué, l'objet lié n'a pas encore été utilisé. Ajoutons donc, à la classe Remember, une méthode supplémentaire permettant de mettre des commentaires dans le fichier – disons quelque chose comme :

```
sub comment {
    my $self = shift;
    my $text = shift;
    my $handle = $self->{FH};
    print $handle $text, "\n";
}
```

Voici maintenant l'exemple précédent modifié pour utiliser la méthode `comment` (qui nécessite l'objet lié) :

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

Lorsque ce code s'exécute, il ne produit rien. Voici pourquoi...

Lorsqu'une variable est liée, elle est associée avec l'objet qui est retourné par la fonction TIESCALAR, TIEARRAY, ou TIEHASH. Cette objet n'a normalement qu'une seule référence, à savoir, la référence implicite prise par la variable liée. Lorsque untie() est appelé, cette référence est supprimée, Et donc, comme dans notre premier exemple ci-dessus, le destructeur (DESTROY) de l'objet est appelé ce qui est normal pour un objet qui n'a plus de référence valide; et donc le fichier est fermé.

Dans notre second exemple, par contre, nous avons stocké dans $x une autre référence à l'objet lié. Si bien que lorsque untie() est appelé, il reste encore une référence valide sur l'objet, donc le destructeur n'est pas appelé à ce moment et donc le fichier n'est pas fermé. Le buffer du fichier n'étant pas vidé, ceci explique qu'il n'y ait pas de sortie.

Bon, maintenant que nous connaissons le problème, que pouvons-nous faire pour l'éviter ? Avant l'introduction de la méthode optionnelle UNTIE, il n'y avait que la bonne vieille option `-w`. Celle-ci vous signale tout appel à untie() pour lequel l'objet lié possède encore des références valides. Si le second script ci-dessus est utilisé avec l'option `-w`, Perl affiche le message d'avertissement suivant :

```
untie attempted while 1 inner references still exist
```

Pour faire fonctionner correctement ce script et sans message, assurez-vous qu'il n'existe plus de références valides vers l'objet lié *avant* d'appeler untie() :

```
undef $x;
untie $fred;
```

Maintenant que UNTIE existe, le concepteur de la classe peut décider quelle partie des fonctionnalités doit être associée à un `untie` et celle qui doit être associée à la destruction de l'objet. Le choix de ce qui doit être conservé pour une classe donnée dépend des appels à aux méthodes ne relevant pas de la liaison et qui doivent pouvoir fonctionner sur l'objet lui-même. Dans la plupart des cas, il est maintenant recommandé de placer dans la méthode UNTIE tout ce qui était précédemment dans DESTROY.

Si la méthode UNTIE existe, l'avertissement ci-dessus ne peut pas avoir lieu. À la place, la méthode UNTIE reçoit comme argument le compte des références existantes et peut émettre son propre avertissement si nécessaire. Pour simuler le comportement correspondant au cas sans méthode UNTIE, la méthode suivante peut être utilisée :

```
sub UNTIE
{
  my ($obj,$count) = @_;
  carp "untie attempted while $count inner references still exist" if $count;
}
```

4 VOIR AUSSI

Voir DB_File ou *Config* pour quelques exemples d'utilisations intéressantes de tie(). Une bonne base de départ pour la plupart des utilisations de tie() passe par l'un des modules *Tie::Scalar*, *Tie::Array*, *Tie::Hash* ou *Tie::Handle*.

5 BUGS

L'information d'utilisation de "buckets" fournie par `scalar(%hash)` n'est pas disponible. Cela signifie que son utilisation dans un contexte booléen est impossible (acteullement, cela retourne toujours faux que la table de hachage soit vide ou contienne des éléments).

La local-isation de tableaux ou de tables de hachage liés ne fonctionne pas. Au moment où on quitte la portée de l'appel à local, le tableau ou la table ne retrouve pas sa valeur initiale.

Compter le nombre d'entrées d'une table de hachage via `scalar(keys(%hash))` orou `scalar(values(%hash))` est très inefficace puisque cela nécessite un parcours complet de la table via FIRSTKEY/NEXTKEY.

La gestion des tranches (slices) d'une table ou d'un tableau lié fait appel à de nombreux appels FETCH/STORE puisqu'il n'y pas de méthodes liées pour les opérations sur les tranches.

Vous ne pouvez pas lier facilement une structure multi-niveaux (comme une table de tables de hachage) à un fichier dbm. Le premier problème est la limite de taille d'une valeur (sauf pour GDBM et Berkeley DB). Mais, au-delà, vous aurez aussi le problème du stockage de références sur disque. L'un des modules expérimentaux qui tente de répondre à ce besoin est DBM::Deep. Allez voir le code source sur un site CPAN (comme décrit par *perlmodlib*). Notez que, contrairement à ce que pourrait laisser croire son nom, DBM::Deep n'utilise pas dbm. Un autre module à voir est MLDBM, qui est aussi sur CPAN, mais il a de nombreuses limitations.

Les descripteurs de fichier liés sont encore incomplets. Les appels à sysopen(), truncate(), flock(), fcntl(), stat() et -X ne peuvent pas encore être interceptés.

6 AUTEURS

Tom Christiansen

TIEHANDLE par Sven Verdoolaege <*skimo@dns.ufsia.ac.be*> et Doug MacEachern <*dougm@osf.org*>

UNTIE par Nick Ing-Simmons <*nick@ing-simmons.net*>

SCALAR par Tassilo von Parseval <*tassilo.von.parseval@rwth-aachen.de*>

Les tableaux liés par Casey West <*casey@geeknest.com*>

7 TRADUCTION

7.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

7.2 Traducteur

Traduction initiale et mise à jour v5.10.0 : Paul Gaborit <Paul.Gaborit at enstimac.fr>

7.3 Relecture

Régis Julié <Regis.Julie@cetelem.fr>

8 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.