NAME

Class::XSAccessor::Array – Generate fast XS accessors without runtime compilation

SYNOPSIS

```
package MyClassUsingArraysAsInternalStorage;
use Class::XSAccessor::Array
  constructor => 'new',
  getters => {
    get_foo => 0, # 0 is the array index to access
    get_bar => 1,
  },
  setters => {
    set_foo => 0,
    set_bar => 1,
  },
  accessors => { # a mutator
    buz => 2,
  },
  predicates => { # test for definedness
    has_buz => 2,
  },
  lvalue_accessors => { # see below
    baz => 3,
  },
  true => [ 'is_token', 'is_whitespace' ],
  false => [ 'significant' ];

# The imported methods are implemented in fast XS.

# normal class code here.
```

As of version 1.05, some alternative syntax forms are available:

```
package MyClass;

# Options can be passed as a HASH reference if you prefer it,
# which can also help PerlTidy to flow the statement correctly.
use Class::XSAccessor {
  getters => {
    get_foo => 0,
    get_bar => 1,
  },
};
```

DESCRIPTION

The module implements fast XS accessors both for getting at and setting an object attribute. Additionally, the module supports mutators and simple predicates (`has_foo()` like tests for definedness of an attributes). The module works only with objects that are implemented as **arrays**. Using it on hash-based objects is bound to make your life miserable. Refer to Class::XSAccessor for an implementation that works with hash-based objects.

A simple benchmark showed a significant performance advantage over writing accessors in Perl.

Since version 0.10, the module can also generate simple constructors (implemented in XS) for you. Simply supply the `constructor => 'constructor_name'` option or the `constructors => ['new', 'create', 'spawn']` option. These constructors do the equivalent of the following Perl code:

```
sub new {
  my $class = shift;
  return bless [], ref($class)||$class;
}
```

That means they can be called on objects and classes but will not clone objects entirely. Note that any parameters to **new()** will be discarded! If there is a better idiom for array-based objects, let me know.

While generally more obscure than hash-based objects, objects using blessed arrays as internal representation are a bit faster as its somewhat faster to access arrays than hashes. Accordingly, this module is slightly faster (~10–15%) than Class::XSAccessor, which works on hash-based objects.

The method names may be fully qualified. In the example of the synopsis, you could have written `MyClass::get_foo` instead of `get_foo`. This way, you can install methods in classes other than the current class. See also: The `class` option below.

Since version 1.01, you can generate extremely simple methods which just return true or false (and always do so). If that seems like a really superfluous thing to you, then think of a large class hierarchy with interfaces such as PPI. This is implemented as the `true` and `false` options, see synopsis.

OPTIONS

In addition to specifying the types and names of accessors, you can add options which modify behaviour. The options are specified as key/value pairs just as the accessor declaration. Example:

```
use Class::XSAccessor::Array
  getters => {
    get_foo => 0,
  },
  replace => 1;
```

The list of available options is:

replace

Set this to a true value to prevent `Class::XSAccessor::Array` from complaining about replacing existing subroutines.

chained

Set this to a true value to change the return value of setters and mutators (when called with an argument). If `chained` is enabled, the setters and accessors/mutators will return the object. Mutators called without an argument still return the value of the associated attribute.

As with the other options, `chained` affects all methods generated in the same `use Class::XSAccessor::Array ...` statement.

class

By default, the accessors are generated in the calling class. Using the `class` option, you can explicitly specify where the methods are to be generated.

LVALUES

Support for lvalue accessors via the keyword `lvalue_accessors` was added in version 1.08. At this point, **THEY ARE CONSIDERED HIGHLY EXPERIMENTAL**. Furthermore, their performance hasn't been benchmarked yet.

The following example demonstrates an lvalue accessor:

```
package Address;
use Class::XSAccessor
  constructor => 'new',
  lvalue_accessors => { zip_code => 0 };

package main;
my $address = Address->new(2);
print $address->zip_code, "\n"; # prints 2
```

```
$address->zip_code = 76135; # <--- This is it!
print $address->zip_code, "\n"; # prints 76135
```

CAVEATS

Probably wouldn't work if your objects are *tied*. But that's a strange thing to do anyway.

Scary code exploiting strange XS features.

If you think writing an accessor in XS should be a laughably simple exercise, then please contemplate how you could instantiate a new XS accessor for a new hash key or array index that's only known at run-time. Note that compiling C code at run-time a la Inline::C is a no go.

Threading. With version 1.00, a memory leak has been **fixed** that would leak a small amount of memory if you loaded `Class::XSAccessor`–based classes in a subthread that hadn't been loaded in the "main" thread before. If the subthread then terminated, a hash key and an int per associated method used to be lost. Note that this mattered only if classes were **only** loaded in a sort of throw-away thread.

In the new implementation as of 1.00, the memory will not be released again either in the above situation. But it will be recycled when the same class or a similar class is loaded again in **any** thread.

SEE ALSO

Class::XSAccessor

AutoXS

AUTHOR

Steffen Mueller <smueller@cpan.org>

chocolateboy <chocolate@cpan.org>

COPYRIGHT AND LICENSE