NAME

Types::Serialiser − simple data types for common serialisation formats

SYNOPSIS

DESCRIPTION

This module provides some extra datatypes that are used by common serialisation formats such as JSON or CBOR. The idea is to have a repository of simple/small constants and containers that can be shared by different implementations so they become interoperable between each other.

SIMPLE SCALAR CONSTANTS

Simple scalar constants are values that are overloaded to act like simple Perl values, but have (class) type to differentiate them from normal Perl scalars. This is necessary because these have different representations in the serialisation formats.

BOOLEANS (Types::Serialiser::Boolean class)

This type has only two instances, true and false. A natural representation for these in Perl is `1` and `0`, but serialisation formats need to be able to differentiate between them and mere numbers.

`$Types::Serialiser::true`, Types::Serialiser::true
: This value represents the "true" value. In most contexts is acts like the number `1`. It is up to you whether you use the variable form (`$Types::Serialiser::true`) or the constant form (`Types::Serialiser::true`).

  The constant is represented as a reference to a scalar containing `1` − implementations are allowed to directly test for this.

`$Types::Serialiser::false`, Types::Serialiser::false
: This value represents the "false" value. In most contexts is acts like the number `0`. It is up to you whether you use the variable form (`$Types::Serialiser::false`) or the constant form (`Types::Serialiser::false`).

  The constant is represented as a reference to a scalar containing `0` − implementations are allowed to directly test for this.

`$is_bool` = Types::Serialiser::is_bool `$value`
: Returns true iff the `$value` is either `$Types::Serialiser::true` or `$Types::Serialiser::false`.

  For example, you could differentiate between a perl true value and a `Types::Serialiser::true` by using this:

```
$value && Types::Serialiser::is_bool $value
```

`$is_true` = Types::Serialiser::is_true `$value`
: Returns true iff `$value` is `$Types::Serialiser::true`.

`$is_false` = Types::Serialiser::is_false `$value`
: Returns false iff `$value` is `$Types::Serialiser::false`.

ERROR (Types::Serialiser::Error class)

This class has only a single instance, `error`. It is used to signal an encoding or decoding error. In CBOR for example, and object that couldn't be encoded will be represented by a CBOR undefined value, which is represented by the error value in Perl.

`$Types::Serialiser::error`, Types::Serialiser::error
: This value represents the "error" value. Accessing values of this type will throw an exception.

  The constant is represented as a reference to a scalar containing `undef` − implementations are allowed to directly test for this.

`$is_error` = Types::Serialiser::is_error `$value`
: Returns false iff `$value` is `$Types::Serialiser::error`.

NOTES FOR XS USERS

The recommended way to detect whether a scalar is one of these objects is to check whether the stash is the `Types::Serialiser::Boolean` or `Types::Serialiser::Error` stash, and then follow the scalar reference to see if it's `1` (true), `0` (false) or `undef` (error).

While it is possible to use an isa test, directly comparing stash pointers is faster and guaranteed to work.

For historical reasons, the `Types::Serialiser::Boolean` stash is just an alias for `JSON::PP::Boolean`. When printed, the classname with usually be `JSON::PP::Boolean`, but isa tests and stash pointer comparison will normally work correctly (i.e. Types::Serialiser::true ISA JSON::PP::Boolean, but also ISA Types::Serialiser::Boolean).

A GENERIC OBJECT SERIALIATION PROTOCOL

This section explains the object serialisation protocol used by CBOR::XS. It is meant to be generic enough to support any kind of generic object serialiser.

This protocol is called ''the Types::Serialiser object serialisation protocol''.

ENCODING

When the encoder encounters an object that it cannot otherwise encode (for example, CBOR::XS can encode a few special types itself, and will first attempt to use the special `TO_CBOR` serialisation protocol), it will look up the `FREEZE` method on the object.

Note that the `FREEZE` method will normally be called *during* encoding, and *MUST NOT* change the data structure that is being encoded in any way, or it might cause memory corruption or worse.

If it exists, it will call it with two arguments: the object to serialise, and a constant string that indicates the name of the data model. For example CBOR::XS uses `CBOR`, and the JSON and JSON::XS modules (or any other JSON serialiser), would use `JSON` as second argument.

The `FREEZE` method can then return zero or more values to identify the object instance. The serialiser is then supposed to encode the class name and all of these return values (which must be encodable in the format) using the relevant form for Perl objects. In CBOR for example, there is a registered tag number for encoded perl objects.

The values that `FREEZE` returns must be serialisable with the serialiser that calls it. Therefore, it is recommended to use simple types such as strings and numbers, and maybe array references and hashes (basically, the JSON data model). You can always use a more complex format for a specific data model by checking the second argument, the data model.

The ''data model'' is not the same as the ''data format'' – the data model indicates what types and kinds of return values can be returned from `FREEZE`. For example, in `CBOR` it is permissible to return tagged CBOR values, while JSON does not support these at all, so `JSON` would be a valid (but too limited) data model name for `CBOR::XS`. similarly, a serialising format that supports more or less the same data model as JSON could use `JSON` as data model without losing anything.

DECODING

When the decoder then encounters such an encoded perl object, it should look up the `THAW` method on the stored classname, and invoke it with the classname, the constant string to identify the data model/data format, and all the return values returned by `FREEZE`.

EXAMPLES

See the `OBJECT SERIALISATION` section in the CBOR::XS manpage for more details, an example implementation, and code examples.

Here is an example `FREEZE`/`THAW` method pair:

```
sub My::Object::FREEZE {
   my ($self, $model) = @_;

   ($self->{type}, $self->{id}, $self->{variant})
}
```

```
sub My::Object::THAW {
   my ($class, $model, $type, $id, $variant) = @_;

   $class->new (type => $type, id => $id, variant => $variant)
}
```

BUGS

The use of overload makes this module much heavier than it should be (on my system, this module: 4kB RSS, overload: 260kB RSS).

SEE ALSO

Currently, JSON::XS and CBOR::XS use these types.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://home.schmorp.de/