NAME

Sereal::Decoder – Fast, compact, powerful binary deserialization

SYNOPSIS

```
use Sereal::Decoder
  qw(decode_sereal sereal_decode_with_object scalar_looks_like_sereal);

my $decoder = Sereal::Decoder->new({...options...});

my $structure;
$decoder->decode($blob, $structure); # deserializes into $structure

# or if you don't have references to the top level structure, this works, too:
$structure = $decoder->decode($blob);

# alternatively functional interface: (See Sereal::Performance)
sereal_decode_with_object($decoder, $blob, $structure);
$structure = sereal_decode_with_object($decoder, $blob);

# much slower functional interface with no persistent objects:
decode_sereal($blob, {... options ...}, $structure);
$structure = decode_sereal($blob, {... options ...});

# Not a full validation, but just a quick check for a reasonable header:
my $is_likely_sereal = scalar_looks_like_sereal($some_string);
# or:
$is_likely_sereal = $decoder->looks_like_sereal($some_string);
```

DESCRIPTION

This library implements a deserializer for an efficient, compact-output, and feature-rich binary protocol called *Sereal*. Its sister module Sereal::Encoder implements an encoder for this format. The two are released separately to allow for independent and safer upgrading.

The Sereal protocol versions that are compatible with this decoder implementation are currently protocol versions 1, 2, 3 and 4. As it stands, it will refuse to attempt to decode future versions of the protocol, but if necessary there is likely going to be an option to decode the parts of the input that are compatible with version 4 of the protocol. The protocol was designed to allow for this.

The protocol specification and many other bits of documentation can be found in the github repository. Right now, the specification is at <https://github.com/Sereal/Sereal/blob/master/sereal_spec.pod>, there is a discussion of the design objectives in <https://github.com/Sereal/Sereal/blob/master/README.pod>, and the output of our benchmarks can be seen at <https://github.com/Sereal/Sereal/wiki/Sereal-Comparison-Graphs>.

CLASS METHODS

new

Constructor. Optionally takes a hash reference as first parameter. This hash reference may contain any number of options that influence the behaviour of the encoder.

Currently, the following options are recognized, none of them are on by default.

refuse_snappy

If set, the decoder will refuse Snappy-compressed input data. This can be desirable for robustness. See the section `ROBUSTNESS` below.

refuse_objects

If set, the decoder will refuse deserializing any objects in the input stream and instead throw an exception. Defaults to off. See the section `ROBUSTNESS` below.

no_bless_objects

If set, the decoder will deserialize any objects in the input stream but without blessing them. Defaults to off. See the section ROBUSTNESS below.

validate_utf8

If set, the decoder will refuse invalid UTF-8 byte sequences. This is off by default, but it's strongly encouraged to be turned on if you're dealing with any data that has been encoded by an external source (e.g. http cookies).

max_recursion_depth

`Sereal::Decoder` is recursive. If you pass it a Sereal document that is deeply nested, it will eventually exhaust the C stack. Therefore, there is a limit on the depth of recursion that is accepted. It defaults to 10000 nested calls. You may choose to override this value with the `max_recursion_depth` option. Beware that setting it too high can cause hard crashes.

Do note that the setting is somewhat approximate. Setting it to 10000 may break at somewhere between 9997 and 10003 nested structures depending on their types.

max_num_hash_entries

If set to a non-zero value (default: 0), then `Sereal::Decoder` will refuse to deserialize any hash/dictionary (or hash-based object) with more than that number of entries. This is to be able to respond quickly to any future hash-collision attacks on Perl's hash function. Chances are, you don't want or need this. For a gentle introduction to the topic from the cryptographic point of view, see <http://en.wikipedia.org/wiki/Collision_attack>.

incremental

If set to a non-zero value (default: 0), then `Sereal::Decoder` will destructively parse Sereal documents out of a variable. Every time a Sereal document is successfully parsed it is removed from the front of the string it is parsed from.

This means you can do this:

```
while (length $buffer) {
    my $data= decode_sereal($buffer,{incremental=>1});
}
```

alias_smallint

If set to a true value then `Sereal::Decoder` will share integers from −16 to 15 (encoded as either SRL_HDR_NEG and SRL_HDR_POS) as read-only aliases to a common SV.

The result of this may be significant space savings in data structures with many integers in the specified range. The cost is more memory used by the decoder and a very modest speed penalty when deserializing.

Note this option changes the structure of the dumped data. Use with caution.

See also the ‘‘alias_varint_under’’ option.

alias_varint_under

If set to a true positive integer smaller than 16 then this option is similar to setting ‘‘alias_smallint’’ and causes all integers from −16 to 15 to be shared as read-only aliases to the same SV, except that this treatment ALSO applies to SRL_HDR_VARINT. If set to a value larger than 16 then this applies to all varints varints under the value set. (In general SRL_HDR_VARINT is used only for integers larger than 15, and SRL_HDR_NEG and SRL_HDR_POS are used for −16 to −1 and 0 to 15 respectively.)

In simple terms if you want to share values larger than 16 then you should use this option, if you want to share only values in the −16 to 15 range then you should use the ‘‘alias_smallint’’ option instead.

The result of this may be significant space savings in data structures with many integers in the desire range. The cost is more memory used by the decoder and a very modest speed penalty when deserializing.

Note this option changes the structure of the dumped data. Use with caution.

use_undef

If set to a true value then this any undef value to be deserialized as PL_sv_undef. This may change the structure of the data structure being dumped, do not enable this unless you know what you are doing.

set_readonly

If set to a true value then the output will be completely readonly (deeply).

set_readonly_scalars

If set to a true value then scalars in the output will be readonly (deeply). References won't be readonly.

INSTANCE METHODS

decode

Given a byte string of Sereal data, the `decode` call deserializes that data structure. The result can be obtained in one of two ways: `decode` accepts a second parameter, which is a scalar to write the result to, AND `decode` will return the resulting data structure.

The two are subtly different in case of data structures that contain references to the root element. In that case, the return value will be a (non-recursive) copy of the reference. The pass-in style is more correct. In other words,

```
$decoder->decode($sereal_string, my $out);
# is almost the same but safer than:
my $out = $decoder->decode($sereal_string);
```

This is an unfortunate side-effect of perls standard copy semantics of assignment. Possibly one day we will have an alternative to this.

decode_with_header

Given a byte string of Sereal data, the `decode_with_header` call deserializes that data structure as `decode` would do, however it also decodes the optional user data structure that can be embedded into a Sereal document, inside the header (see Sereal::Encoder::encode).

It accepts an optional second parameter, which is a scalar to write the body to, and an optional third parameter, which is a scalar to write the header to.

Regardless of the number of parameters received, `decode_with_header` returns an ArrayRef containing the deserialized header, and the deserialized body, in this order.

See `decode` for the subtle difference between the one, two and three parameters versions.

If there is no header in a Sereal document, corresponding variable or return value will be set to undef.

decode_only_header

Given a byte string of Sereal data, the `decode_only_header` deserializes only the optional user data structure that can be embedded into a Sereal document, inside the header (see Sereal::Encoder::encode).

It accepts an optional second parameter, which is a scalar to write the header to.

Regardless of the number of parameters received, `decode_only_header` returns the resulting data structure.

See `decode` for the subtle difference between the one and two parameters versions.

If there is no header in a Sereal document, corresponding variable or return value will be set to undef.

decode_with_offset

Same as the `decode` method, except as second parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional ‘‘pass-in’’ style scalar (see `decode` above) is relegated to being the third parameter.

decode_only_header_with_offset

Same as the `decode_only_header` method, except as second parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional ‘‘pass-in’’ style scalar (see `decode_only_header` above) is relegated to being the third parameter.

decode_with_header_and_offset

Same as the `decode_with_header` method, except as second parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional ‘‘pass-in’’ style scalars (see `decode_with_header` above) are relegated to being the third and fourth parameters.

bytes_consumed

After using the various `decode` methods documented previously, `bytes_consumed` can return the number of bytes **from the body** of the input string that were actually consumed by the decoder. That is, if you append random garbage to a valid Sereal document, `decode` will happily decode the data and ignore the garbage. If that is an error in your use case, you can use `bytes_consumed` to catch it.

```
my $out = $decoder->decode($sereal_string);
if (length($sereal_string) != $decoder->bytes_consumed) {
  die "Not all input data was consumed!";
}
```

Chances are that if you do this, you’re violating UNIX philosophy in ‘‘be strict in what you emit but lenient in what you accept’’.

You can also use this to deserialize a list of Sereal documents that is concatenated into the same string (code not very robust...):

```
my @out;
my $pos = 0;
eval {
  while (1) {
    push @out, $decoder->decode_with_offset($sereal_string, $pos);
    $pos += $decoder->bytes_consumed;
    last if $pos >= length($sereal_string)
         or not $decoder->bytes_consumed;
  }
};
```

As mentioned, only the bytes consumed from the body are considered. So the following example is correct, as only the header is deserialized:

```
my $header = $decoder->decode_only_header($sereal_string);
my $count = $decoder->bytes_consumed;
# $count is 0
```

decode_from_file

```
Sereal::Decoder->decode_from_file($file);
$decoder->decode_from_file($file);
```

Read and decode the file specified. If called in list context and incremental mode is enabled then decodes all packets contained in the file and returns a list, otherwise decodes the first (or only) packet in the file. Accepts an optional ‘‘target’’ variable as a second argument.

looks_like_sereal

Performs some rudimentary check to determine if the argument appears to be a valid Sereal packet or not. These tests are not comprehensive and a true result does not mean that the document is valid, merely that it appears to be valid. On the other hand a false result is always reliable.

The return of this method may be treated as a simple boolean but is in fact a more complex return. When the argument does not look anything like a Sereal document then the return is perl’s FALSE, which has the property of being string equivalent to "‘‘ and numerically equivalent to 0. However when the argument appears to be a UTF–8 encoded protocol 3 Sereal document (by noticing that the \xF3 in the magic string has been replaced by \xC3\xB3) then it returns 0 (the number, which is string equivalent to ’’0"), and otherwise returns the protocol version of the document. This means you can write something like this:

```
$type= Sereal::Decoder->looks_like_sereal($thing);
if ($type eq '') {
    say "Not a Sereal document";
} elsif ($type eq '0') {
    say "Possibly utf8 encoded Sereal document";
} else {
    say "Sereal document version $type";
}
```

For reference, Sereal's magic value is a four byte string which is either `=srl` for protocol version 1 and 2 or `=\xF3rl` for protocol version 3 and later. This function checks that the magic string corresponds with the reported version number, as well as other checks, which may be enhanced in the future.

Note that **looks_like_sereal()** may be called as a class or object method, and may also be called as a single argument function. See the related **scalar_looks_like_sereal()** for a version which may ONLY be called as a function, not as a method (and which is typically much faster).

EXPORTABLE FUNCTIONS

sereal_decode_with_object

The functional interface that is equivalent to using `decode`. Takes a decoder object reference as first parameter, followed by a byte string to deserialize. Optionally takes a third parameter, which is the output scalar to write to. See the documentation for `decode` above for details.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

sereal_decode_with_header_with_object

The functional interface that is equivalent to using `decode_with_header`. Takes a decoder object reference as first parameter, followed by a byte string to deserialize. Optionally takes third and fourth parameters, which are the output scalars to write to. See the documentation for `decode_with_header` above for details.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

sereal_decode_only_header_with_object

The functional interface that is equivalent to using `decode_only_header`. Takes a decoder object reference as first parameter, followed by a byte string to deserialize. Optionally takes a third parameters, which outputs scalars to write to. See the documentation for `decode_with_header` above for details.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

sereal_decode_only_header_with_offset_with_object

The functional interface that is equivalent to using `decode_only_header_with_offset`. Same as the `sereal_decode_only_header_with_object` function, except as the third parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional ''pass-in'' style scalar (see `sereal_decode_only_header_with_object` above) is relegated to being the fourth parameter.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

sereal_decode_with_header_and_offset_with_object

The functional interface that is equivalent to using `decode_with_header_and_offset`. Same as the `sereal_decode_with_header_with_object` function, except as the third parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional ''pass-in''

style scalars (see `sereal_decode_with_header_with_object` above) are relegated to being the fourth and fifth parameters.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

sereal_decode_with_offset_with_object

The functional interface that is equivalent to using `decode_with_offset`. Same as the `sereal_decode_with_object` function, except as the third parameter, you must pass an integer offset into the input string, at which the decoding is to start. The optional "pass-in" style scalar (see `sereal_decode_with_object` above) is relegated to being the third parameter.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead. See Sereal::Performance for a discussion on how to tune Sereal for maximum performance if you need to.

decode_sereal

The functional interface that is equivalent to using `new` and `decode`. Expects a byte string to deserialize as first argument, optionally followed by a hash reference of options (see documentation for `new()`). Finally, `decode_sereal` supports a third parameter, which is the output scalar to write to. See the documentation for `decode` above for details.

This functional interface is significantly slower than the OO interface since it cannot reuse the decoder object.

decode_sereal_with_header_data

The functional interface that is equivalent to using `new` and `decode_with_header`. Expects a byte string to deserialize as first argument, optionally followed by a hash reference of options (see documentation for `new()`). Finally, `decode_sereal` supports third and fourth parameters, which are the output scalars to write to. See the documentation for `decode_with_header` above for details.

This functional interface is significantly slower than the OO interface since it cannot reuse the decoder object.

scalar_looks_like_sereal

The functional interface that is equivalent to using `looks_like_sereal`.

Note that this version cannot be called as a method. It is normally executed as a custom opcode, as such errors about its usage may be caught at compile time, and it should be much faster than looks_like_sereal.

ROBUSTNESS

This implementation of a Sereal decoder tries to be as robust to invalid input data as reasonably possible. This means that it should never (though read on) segfault. It may, however, cause a large malloc to fail. Generally speaking, invalid data should cause a Perl-trappable exception. The one exception is that for Snappy-compressed Sereal documents, the Snappy library may cause segmentation faults (invalid reads or writes). This should only be a problem if you do not checksum your data (internal checksum support is a To-Do) or if you accept data from potentially malicious sources.

It requires a lot of run-time boundary checks to prevent decoder segmentation faults on invalid data. We implemented them in the lightest way possible. Adding robustness against running out of memory would cause an very significant run-time overhead. In most cases of random garbage (with valid header no less) when a **malloc()** fails due to invalid data, the problem was caused by a very large array or string length. This kind of very large malloc can then fail, being trappable from Perl. Only when packet causes many repeated allocations do you risk causing a hard OOM error from the kernel that cannot be trapped because Perl may require some small allocations to succeed before the now-invalid memory is released. It is at least not entirely trivial to craft a Sereal document that causes this behaviour.

Finally, deserializing proper objects is potentially a problem because classes can define a destructor. Thus, the data fed to the decoder can cause the (deferred) execution of any destructor in your application. That's why the `refuse_objects` option exists and what the `no_bless_objects` can be used for as well. Later on, we may or may not provide a facility to whitelist classes. Furthermore, if the encoder emitted any

objects using `FREEZE` callbacks, the `THAW` class method may be invoked on the respective classes. If you can't trust the source of your Sereal documents, you may want to use the `refuse_objects` option. For more details on the `FREEZE/THAW` mechanism, please refer to Sereal::Encoder.

PERFORMANCE

Please refer to the Sereal::Performance document that has more detailed information about Sereal performance and tuning thereof.

THREAD-SAFETY

`Sereal::Decoder` is thread-safe on Perl's 5.8.7 and higher. This means ''thread-safe'' in the sense that if you create a new thread, all `Sereal::Decoder` objects will become a reference to undef in the new thread. This might change in a future release to become a full clone of the decoder object.

BUGS, CONTACT AND SUPPORT

For reporting bugs, please use the github bug tracker at <http://github.com/Sereal/Sereal/issues>.

For support and discussion of Sereal, there are two Google Groups:

Announcements around Sereal (extremely low volume): <https://groups.google.com/forum/?fromgroups#!forum/sereal−announce>

Sereal development list: <https://groups.google.com/forum/?fromgroups#!forum/sereal−dev>

AUTHORS AND CONTRIBUTORS

Yves Orton <demerphq@gmail.com>

Damian Gryski

Steffen Mueller <smueller@cpan.org>

Rafaël Garcia-Suarez

Ævar Arnfjörð Bjarmason <avar@cpan.org>

Tim Bunce

Daniel Dragan <bulkdd@cpan.org> (Windows support and bugfixes)

Zefram

Borislav Nikolov

Ivan Kruglov <ivan.kruglov@yahoo.com>

Eric Herman <eric@freesa.org>

Some inspiration and code was taken from Marc Lehmann's excellent JSON::XS module due to obvious overlap in problem domain.

ACKNOWLEDGMENT

This module was originally developed for Booking.com. With approval from Booking.com, this module was generalized and published on CPAN, for which the authors would like to express their gratitude.

COPYRIGHT AND LICENSE