

Rustler - Safe Elixir and Erlang NIFs in Rust

Feb 5, 2017

Natively Implemented Functions, more commonly known as NIFs, are not a new thing in Erlang. They have been around for several years, and are commonly used for speeding up simple tasks like JSON parsing. The reason why they are not more commonly used for general computation is the massive disadvantages they carry with them. While Erlang is generally built for reliability and fault tolerance, all bets are off when writing NIFs written in a language like C. Any programmer error in the NIF code can very easily take the entire Erlang VM down with it, which includes all processes, supervision trees and data contained within.

[The Rust programming language](#) has been gaining more momentum and popularity lately. Its main selling points include guaranteed memory safety, usage of threads without fear of data races, and a large focus on zero cost abstractions. This means you can write and run totally safe code in Rust, no worrying about segfaults. These qualities would be excellent to have when writing NIFs for Erlang, given that a single bug could easily crash the entire VM, or even cause silent corruption.

However, writing safe code that interfaces with C code requires bindings to be written. A binding will generally wrap a unsafe C API to Rusts semantics. This is where [rustler](#) comes in.

The rustler project

The goal of the [rustler](#) project is to provide fully safe bindings for the Erlang NIF API, along with supporting utilities. This means that the user is able to write lightning fast NIFs in Rust, which can run just as fast as their C counterparts, all without risk of bringing the entire VM down when something goes wrong.

A basic example

The majority of the Rustler API deals with getting values in and out of Erlang terms. In Rust-land, Erlang terms are represented by a single type, namely `NifTerm`.

The most basic API for working with `NifTerm`s involves encoding and decoding. Say we want to decode a number, add one to it, and reencode it as a new `NifTerm`:

```
// An input term. We probably got this passed as an argument to the NIF
```

```
// An input term, we probably got this passed as an argument to the NIF.
let in_term: NifTerm = args[0];

// The `NifTerm::decode` method takes care of decoding a term into a value.
// We indicate that we want the result to be an `i64` (64 bit signed integer).
// Since a term can be any value, we want to fail if the term does not actually
// contain a number. We do this by using the `?` operator, which will return
// early if the result is an error.
let number: i64 = in_term.decode()?;

// We add one to the number, and encode the result back into a `NifTerm`.
// The `env` argument to the `encode` method is an internal data structure which
// contains some information on what Erlang process we want to encode the term
// into. We get an `env` as an argument to a nif function call.
let out_term: NifTerm = (number + 1).encode(env);
```

Okey, cool, but not very useful. Let's look at a more useful and complete example.

Mutable binary buffer NIF

Buffers are very commonly used in programming. If you wanted to store a reasonably sized image, you would probably use a binary buffer. Immutable binary buffers are no problem in Elixir, you could simply use a binary. Things get slightly more tricky if you want to be able to arbitrarily modify the data in the buffer (say to arbitrarily set pixels in an image).

If you want to have a mutable binary buffer in Elixir, you have a couple of obvious choices:

1. Use a binary. This is really easy to do, and will probably work fine for smaller buffers. The disadvantage is that this will perform a copy every time you mutate the buffer, even if you only change a single byte.
2. The `:array` module. This is an implementation of arrays that tries to be as efficient as possible while using only immutable data structures (nested tuples) to minimize the cost of random mutation. While this may work well for many use-cases, if you are doing lots of random mutations on a large buffer, this quickly gets costly.
3. Use ETS. This will enable you to use actual mutation, but it is still not completely optimal. The ETS table types are not very well suited for what we are trying to do, and we have to call out of the current process every time we want to access the data.

A better approach would be to use NIFs with resource objects. Resource objects allow you to associate a piece of native data with an opaque Erlang term. You can create a Rust struct containing a binary buffer (or any thing else), put that in a resource object, and use that as an argument to subsequent NIF calls.

This approach has many advantages, which includes very fast mutation and random access to the buffer, with no need to leave the current process.

Let's start off by defining a new Rust struct that contains our buffer of bytes, `Vec<u8>`. We also use a `RwLock` to ensure that the same buffer doesn't get accessed from two places at the same time. Don't worry about forgetting this, Rust will only let your code compile if it's safe.

```
struct Buffer {
    data: RwLock<Vec<u8>>,
}
```

We then need to make this struct into a resource object type. You can think of this as an analog to classes in object-oriented programming. We define a type (class) now, and create instances of it later. This will guarantee that our NIF never gets the wrong type of resource object.

Because we need to initialize a new resource object type, we need a way to run some code when our NIF first gets loaded by the VM. For this we can define a `on_load` function.

TODO: Change name of `resource_struct_init` to `resource_type_init` or something

```
// The ``a` in this function definition is something called a lifetime.
// This will inform the Rust compiler of how long different things are
// allowed to live. Don't worry too much about this, as this will be the
// exact same for most function definitions.
fn on_init<'a>(env: NifEnv<'a>, _load_info: NifTerm<'a>) -> bool {
    // This macro will take care of defining and initializing a new resource
    // object type.
    resource_struct_init!(Buffer, env);
    true
}
```

Now that we have our resource object type, we need to make a new instance of it. To do this we will define a new NIF function that we export to Erlang.

```
fn buffer_new<'a>(env: NifEnv<'a>, args: &[NifTerm<'a>]) -> NifResult<NifTerm<'a>> {
    // The NIF should have a single argument provided, namely
    // the size of the buffer we want to create.
    let buffer_size: usize = args[0].decode()?;

    // Create the actual buffer and initialize it with zeroes.
    let mut buffer = Vec::with_capacity(buffer_size);
    for i in 0..buffer_size { buffer.push(0); }

    // Make the actual struct
    let buffer_struct = Buffer {
        data: RwLock::new(buffer),
    };
}
```

```

    // Return it!
    Ok(ResourceArc::new(buffer_struct).encode(env))
}

```

All we need now is some way to actually modify the buffer. We define a NIF function for getting and setting a byte.

```

fn buffer_get<'a>(env: NifEnv<'a>, args: &[NifTerm<'a>]) -> NifResult<NifTerm<'a>> {
    let buffer: ResourceArc<Buffer> = args[0].decode()?;
    let offset: usize = args[1].decode()?;

    let byte = buffer.data.read()[offset];
    Ok(byte.encode(env))
}

fn buffer_set<'a>(env: NifEnv<'a>, args: &[NifTerm<'a>]) -> NifResult<NifTerm<'a>> {
    let buffer: ResourceArc<Buffer> = args[0].decode()?;
    let offset: usize = args[1].decode()?;
    let byte = args[2].decode()?;

    buffer.data.write()[offset] = byte;
    Ok(byte.encode(env))
}

```

We then export all the functions we have defined so they can be called by the VM.

```

rustler_export_nifs!(
    // The name of the module we want the NIF to be in.
    "Elixir.Buffer.Native",

    // The functions we want to export.
    // They consist of a tuple containing the function name,
    // function arity, and the Rust function itself.
    [
        ("new", 1, buffer_new),
        ("get", 2, buffer_get),
        ("set", 3, buffer_set),
    ],
    // Our on_load function. Will get called on load.
    Some(on_load)
);

```

To actually compile the NIF, we use the [rustler](#) package on [hex.pm](#). We need to add the NIF crate to the `mix.exs` of your package.

```
def project do
  [
    app: :buffer,
    # Some lines omitted for brevity.
    compilers: [:rustler] ++ Mix.compilers(),
    rustler_crates: rustler_crates()
  ]
end

def rustler_crates do
  [
    html5ever_nif: [
      path: "native/buffer_nif",
      features: [],
    ]
  ]
end
```

Lastly, we need to actually load the compiled NIF. To do this, we need to define the module we specified in the `rustler_export_nifs` macro.

```
defmodule NifNotLoadedError do
  defexception message: "nif not loaded"
end

defmodule Buffer.Native do
  use Rustler, otp_app: :buffer, crate: "buffer_nif"

  def new(size), do: err()
  def get(buffer, idx), do: err()
  def set(buffer, idx, value), do: err()

  defp err() do
    throw NifNotLoadedError
  end
end
```

Caveats

So far we have looked at the advantages of writing NIFs in rustler, and how it can be done. Although the NIFs you write should never crash the BEAM, there are still a couple of bad things you can do.

How the Erlang VM runs code in Erlang processes is a bit strange. You have a small amount of OS threads which act as Erlang schedulers (by default there is one thread per CPU core).

These scheduler threads are what actually runs your Erlang code. The special part about how Erlang does scheduling, is that it doesn't allow a single process to run for more than a small amount of time before it reschedules it and starts executing another process. What this means for the user is that a single process which takes up a lot of CPU time, will not degrade the latency of any of the other (millions?) of processes running within the VM. If you are interested in knowing more about Erlang's schedulers, [this](#) is a good article that goes into further details.

Since NIFs run within the scheduler threads by default, bad things will happen to the latency and performance of processes within the VM if a NIF runs for more than around a millisecond. This should be avoided, and care therefore needs to be taken to avoid blocking the scheduler thread for too long.

Luckily, a few techniques can be used to avoid this.

1. Do the computation in smaller chunks, making sure to return from the NIF before too much time is taken.
2. Put the computation on a separate thread, potentially in a thread pool. When the computation is finished, send the result back to the correct Erlang process.

We should add a safe API for rescheduling in a future version. Right now there are some invariants we can't enforce at compile-time.

Wrapping up

The project currently works very well for writing many types of NIFs. It is not feature-complete, and you should therefore expect breaking API changes in the future.

Despite this, I would say the library is reasonably ready for production use. There are no known ways to crash the BEAM as long as you are only using safe code (if there is a way, that's a bug). If you need a NIF, you are generally a lot better off with choosing Rust over C when it comes to safety.

Resources

- [Github repository](#)
- [Rust crate documentation](#)
- [Mix compiler documentation](#)

These examples are actively maintained and kept up to date with rustler.

- [Test suite](#)
- [html5ever](#) - html5ever binding

If you want to talk to us, we are easily reachable in multiple places. We don't usually bite, so if you have any questions or concerns, feel free to reach out :)

- [#rustler](#) on [freenode](#)
- [#rustler](#) on [the elixir-lang slack](#)
- [Issue tracker](#) - Only for issues, not for general help.

2 Comments Hans' blog

[Login](#) [Recommend](#) 2 [Share](#)[Sort by Best](#) 

LOG IN WITH

OR SIGN UP WITH DISQUS **Faris El alaoui** • a year ago

Great perspective.

I'm learning Elixir right now and I was heartbroken when I learned that the BEAM was "bad" for number crunching.

With this I have a clear path for exploring data analytics in Rusty Elixir way!

Keep up the good work!

^ | v • Reply • Share ›

jbcdu87 jbcn1 • a year ago



That was pretty interesting! Thanks!

^ | v • Reply • Share ›

[Subscribe](#) [Add Disqus to your site](#) [Add Disqus](#) [Add Disqus](#)[Disqus Privacy Policy](#) [Disqus Privacy Policy](#) [Disqus Privacy Policy](#)

The blog of Hans

The blog of Hans
me@hansihe.com

 [hansihe](#)
 [hansihe](#)