Using C++ From Elixir With Nifs

17 December 2017

previous next

In the previous post, we explored the possibility of running python code from Elixir by using the erlport library that is based on the ports facility available in Erlang(and Elixir).

Using ports requires launching an OS process that will host the Python code, we also saw how the erlport library makes it easy to send messages from and to the Python process.

In this post, we are going to see how to implement functions in C++ and how to use them from Elixir without launching any OS process by using the Nifs facility available in Erlang.

Full code sample containing the examples in this post is available on Github.

Native Implemented Functions

Native implemented functions is a feature available in Erlang that allows to call code implemented in C/C++ from Elixir, the native code (code in C/C++) is compiled into a shared library and is usually loaded in the Erlang VM when it starts up.

Functions defined in C++ cannot be called directly and some wrapping needs to take place; the erl_nif.h C library is used to define Elixir to C/C++ function mapping as well as to convert Elixir data structures to their corresponding representations in C/C++.

Nifs can be called without having to use OS processes **but beware, a crash in the C/C++ code will cause the entire Erlang VM to crash; effectively bypassing any error handling mechanism.**

In fact, C/C++ is notoriously unsafe especially when manipulating pointers. Unfortunately, to mitigate this *unsafety* you will have to code very defensively in your C/C++ code which is in some sense against the tao of *let it crash*.

Minimal Example

Let's take a look at a very basic example, we wish to implement an `add` function in C++ and use it from Elixir.

C++ Side

First, we have the simple `add` C++ function:

```
int add(int a, int b)
{
    return a + b;
}
```

Then, we need to use the `<erl_nif.h>` library to convert the integers passed from elixir to plain C++ `int`:

```
#include <erl_nif.h>

int add (int a, int b)
{
    return a + b;
}

ERL_NIF_TERM add_nif(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    int a = 0;
    int b = 0;

    if (!enif_get_int(env, argv[0], &a)) {
        return enif_make_badarg(env);
    }
    if (!enif_get_int(env, argv[1], &b)) {
        return enif_make_badarg(env);
    }

    int result = add(a, b);
    return enif_make_int(env, result);
}

ErlNifFunc nif_funcs[] =
{
    {"add", 2, add_nif},
};

ERL_NIF_INIT(Elixir.Nativly, nif_funcs, nullptr,
    nullptr, nullptr, nullptr);
```

Let's analyze this bottom-up, the `ERL_NIF_INIT` initialization function accepts the name of the Erlang module that contains the nifs as well as the array containing nifs mapping; Elixir modules are just basically Erlang modules prefixed with `Elixir.` The `ERL_NIF_INIT` function simply wires things up.

The `nif_funcs` two dimensional array specifies the Elixir function to C++ function mapping in the following form: `{<elixir_fn_name>, <arity>, <cpp_fn_name>}`

The `add_nif` function has 3 main responsibilities:

1. Converting Elixir data to C++ data, notice how we used the `enif_get_int` function defined in `<erl_nif.h>` to convert `ERL_NIF_TERM` to an `int` *with error checking* since any Elixir datatype can be passed
2. Executing the native add function
3. Converting the result from C++ data to Elixir data and returning it

Finally, we need to compile the previous C++ file to a shared library which is going to be basically a single binary file.

We compile the C++ code by using the following command on a linux machine:

```
g++ -O3 -fpic -shared -o nativly.so nativly.cpp
```

Let's mash a bit on the previous, here we used the g++ compiler to produce the `nativly.so` shared library from the `nativly.cpp` file.

The `-o` flag allows us to specify the name of the shared library and the `-shared` flag allows us to instruct g++ to output a shared library.

The `-fpic` allows us to generate position independent code which is always recommended for shared libraries.

Last but not least, the `-O3` flag instructs the compiler **to perform very aggressive performance optimizations** on the generated machine code. The idea behind using C++ is to have very fast and machine optimized CPU bound functions.

Elixir Side

Before proceeding to the Elixir side, it is essential to make sure that the shared library has the same name as the Elixir module containing the nifs, in our example the name is `nativly`.

On the Elixir side, we have to define the façade functions that are going to act as gateways to the native implemented functions and we need to load the shared library as well.

Consider the following:

```
defmodule Nativly do
  @on_load :load_nifs

  def load_nifs do
    :erlang.load_nif('./nativly', 0)
  end

  def add(_a, _b) do
    raise "NIF add/2 not implemented"
```

```
  end
end
```

The `Nativly.load_nifs` function is scheduled to be executed when the `Natively` module loads with `@on_load` and is responsible for loading the C++ shared library.

The `Nativly.add` function will be mapped to the C++ function when the shared library is correctly loaded and from then on calls to `add` will be calls to the C++ code.

When no mapping is found on the C++ side, calling `add` will `raise` an exception indicating that it is not implemented. This can occur when the mapping is simply not available or when the shared library binary is incompatible with the current architecture.

Hooking The C++ Compilation To Mix

Manually compiling the C++ code on each change is tedious but fortunately we can hook the native code compilation to mix which runs it as part of the Elixir compilation.

Consider this snippet from `mix.exs`:

```
defmodule Mix.Tasks.Compile.Nativly do
  def run(_args) do
    {result, _errcode} = System.cmd("g++",
      ["--std=c++11",
        "-O3",
        "-fpic",
        "-shared",
        "-o", "nativly.so",
        "native_lib/nativly.cpp"
      ], stderr_to_stdout: true)
    IO.puts(result)
  end
end

defmodule Nativly.Mixfile do
  use Mix.Project

  def project do
    [
      app: :nativly,
      ...
      compilers: [:nativly] ++ Mix.compilers,
    ]
  end
  ...
end
```

The `Mix.Tasks.Compile.Nativly` module defines in its `run` function how to compile the C++ code, then under the returned list from the `project` function we add the `compilers` entry which contains our defined compiler: `:nativly` as well as the default compilers `Mix.compilers`.

Running `mix compile` or `iex -S mix` will cause the C++ code to compile alongside the Elixir code.

Dot Product Example

Let's now take a look at a slightly more interesting example, the dot product is a mathematical operation that takes two vectors and calculates the sum of the element-wise products.

Assuming that both vectors have the same size for the sake of simplicity, `dot` can be implemented in Elixir as follows:

```
def dot(a, b) do
    Enum.zip(a, b)
    |> Enum.map(fn {ea, eb} -> ea * eb end)
    |> Enum.reduce(&Kernel.+/2)
end
```

And in C++ in an imperative and procedural style as follows:

```
double dot(const vector<double> &a, const vector<double> &b)
{
    double result = 0.0d;
    for (int i = 0; i < a.size(); ++i)
    {
        result += a[i] * b[i];
    }
    return result;
}
```

The previous C++ function uses the `std::vector` class which represents a dynamic list, note also how we used the const reference syntax to define the arguments of `dot` to avoid using pointers as well as passing the vectors by value.

Now, we are going to see how to convert an Elixir list to a C++ `std::vector`, consider the following:

```
bool enif_fill_vector(ErlNifEnv* env, ERL_NIF_TERM listTerm,
    vector<double> &result)
{
    unsigned int length = 0;

    if (!enif_get_list_length(env, listTerm, &length))
    {
        return false;
    }

    double actualHead;
    ERL_NIF_TERM head;
    ERL_NIF_TERM tail;
    ERL_NIF_TERM currentList = listTerm;
    for (unsigned int i = 0; i < length; ++i)
    {
        if (!enif_get_list_cell(env, currentList, &head, &tail))
        {
            return false;
        }
        currentList = tail;
```

```
        if (!enif_get_double(env, head, &actualHead))
        {
            return false;
        }
        result.push_back(actualHead);
    }

    return true;
}

ERL_NIF_TERM dot_nif(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    vector<double> a;
    vector<double> b;

    if (!enif_fill_vector(env, argv[0], a))
    {
        return enif_make_badarg(env);
    }
    if (!enif_fill_vector(env, argv[1], b))
    {
        return enif_make_badarg(env);
    }

    double result = dot(a, b);
    return enif_make_double(env, result);
}
```

As in the `add` example we defined a `dot_nif` function that is going to be mapped to its corresponding function in Elixir, here all of the complexity of list conversion has been pushed to the `enif_fill_vector` function.

`enif_fill_vector` uses the `enif_get_list_length` in order to get the size of the list and uses the `enif_get_list_cell` that allows to extract the `head` and `tail` of the list. Elements of the list are extracted iteratively from the beginning of the list and appended in the resulting `vector`.

The list conversion has an N complexity which can be a considerable overhead, nonetheless the C++ version still performs an order of magnitude faster for `N = 1'000'000`:

```
iex(1)> Nativly.benchmark
For N = 1000000:
=============
Elixir took 0.285704s
Native took 0.05946s
```

Closing Thoughts

Using Nifs involves a lot of boilerplate, but despite the overhead that the boilerplate implies native code performs way better for CPU bound problems than Erlang/Elixir.

C/C++ is a language that needs to be deeply understood in order to leverage its benefits *safely*. I heard that Rust can be safer alternative and thanks to the Rustler library it is possible to write Nifs in Rust.

Finally, I would like to acknowledge that the Using C from Elixir with NIFs post has been very helpful in understanding nifs usage from Elixir. Definitely check it out.