Rotor Documentation

Release 0.6.3

Paul Colomiets

November 24, 2016

Contents

This is work in progress book about writing applications in “rotor”. There are also Api Docs.

Contents:

CHAPTER 1

Loop Initialization

1.1 Overview

Loop initialization has two stages. First is created by:

```
let loop_creator = try!(rotor::Loop::new());
```

And the second is created by:

```
let loop_instance = loop_creator.instantiate(context)
```

Then you can run the loop:

```
try!(loop_instance.run());
```

As you can see the `loop_creator.instantiate(..)` takes a context for the instantiation. This *is* the **key difference** between two stages.

There is a shortcut if you want to skip second stage of initialization:

```
let loop_creator = try!(rotor::Loop::new());
try!(loop_creator.run(context));
```

1.2 Adding State Machines

To have something useful of main loop you need to add a state machine to it. State machine initialization is done via `add_machine_with` method:

```
try!(loop_creator.add_machine_with(|scope| {
    Ok(Tcp::new(addr, scope))
}));
```

And in loop instance there is similar method:

```
try!(loop_instance.add_machine_with(|scope| {
    Ok(Tcp::new(addr, scope))
}));
```

The difference is in the signature of the function:

```
impl Loop {
    fn add_machine_with<F>(&mut self, fun: F)
        -> Result<(), SpawnError<()>>
        where F: FnOnce(&mut EarlyScope) -> Result<M, Box<Error>>;
}
impl LoopInstance {
    fn add_machine_with<F>(&mut self, fun: F)
        -> Result<(), SpawnError<()>>
        where F: FnOnce(&mut Scope<C>) -> Result<M, Box<Error>>;
}
```

As you can see the only difference is that loop creator gets `EarlyScope` as an argument and latter gets `Scope<Context>` as an argument:

1. Both have `GenericScope` implementation, so you can have constructors generic over the scope type
2. `Scope` dereferences to the context while `EarlyScope` does not

Thats it. But in reality it's important. For example, rotor-dns creates a pair: a state machine and a resolver object. State machine is just added to a loop, but you may want to put resolver object to a context. For example:

```
extern crate rotor_dns;

let resolver_opt = None;
try!(loop_creator.add_machine_with(|scope| {
    let (res, fsm) = try!(rotor_dns::create_resolver(scope, cfg));
    resolver_opt = Some(res);
    Ok(fsm)
}));
let resolver = resolver_opt.unwrap();
let mut loop_instance = loop_creator.instantiate(Context {
    dns: resolver,
});
loop_instance.add_machine_with(..)
```

With rotor-tools the code is simplified to:

```
extern crate rotor_dns;
extern crate rotor_tools;
use rotor_tools::LoopExt;  // The trait with helper functions

let resolver = try!(loop_creator.add_and_fetch(|scope| {
    rotor_dns::create_resolver(scope, cfg)
}));
let mut loop_instance = loop_creator.instantiate(Context {
    dns: resolver,
});
loop_instance.add_machine_with(..)
```

CHAPTER 2

Implementing State Machines

This guide is for **authors** of the protocols, *not* for **users** of the protocols. Read it if you want to write an **new** protocol implementation on top of raw `rotor::Machine`. Otherwise consult on your protocol documenation (probably good links are in *Ecosystem*)

2.1 Boilerplate

This is just a blanket stub implementation I usually start with, filling in methods one by one:

```
extern crate rotor;

use rotor::{Machine, EventSet, Scope, Response};
use rotor::void::{unreachable, Void};

impl<C> Machine for Fsm<C> {
    type Context = C;
    type Seed = Void;
    fn create(seed: Self::Seed, _scope: &mut Scope<C>)
        -> Response<Self, Void>
    {
        unreachable(seed)
    }
    fn ready(self, _events: EventSet, _scope: &mut Scope<C>)
        -> Response<Self, Self::Seed>
    {
        unimplemented!();
    }
    fn spawned(self, _scope: &mut Scope<C>) -> Response<Self, Self::Seed>
    {
        unimplemented!();
    }
    fn timeout(self, _scope: &mut Scope<C>) -> Response<Self, Self::Seed>
    {
        unimplemented!();
    }
    fn wakeup(self, _scope: &mut Scope<C>) -> Response<Self, Self::Seed>
    {
        unimplemented!();
    }
}
```

There are two intricate things here:

1. We use `void` crate and `void::Void` type to denote that seed can't be created so `create` method is never called

   Keep the type `void` unless your machine spawns new state machines. And in the latter case it's advised to use some abstraction for state machine spawning. There is an `rotor_stream::Accept` for accepting sockets, more to come.

2. Implementation should almost always use generic context (`impl<C>`) as only end application should know the exact layout of a context.

   You may limit the generic with some traits (`impl<C: HttpContext>`).

   Often, your state machine doesn't rely on context at all. Currently, this requires adding a `PhantomData<*const C>` marker to state machine. The **marker_** is zero-sized, so it just a little bit of boring code.

CHAPTER 3

Ecosystem

3.1 Libraries

- rotor-tools – a collection of small convenience utilities
- rotor-test – a collection of utilities for writing unit tests
- rotor-stream – an abstraction for writing protocols which use TCP or Unix stream sockets
- rotor-carbon – implementation of the carbon protocol (more known as graphite)
- rotor-dns – DNS support for rotor
- rotor-http – HTTP server and client implementation
- rotor-redis – redis client implementation
- hyper – the implementation of HTTP protocol added to hyper itself
- rotor-capnp – implementation of Cap'n'Proto protocol

3.2 Applications

- Kinglet – a HTTP server
- basic-http-server – also a HTTP server

3.3 Other

- stator – a wrapper around foreign function interface (FFI) for various rotor libraries that allows dispatching them from scripting languages; thus offloading asynchronous and protocol parsing work to rotor that is put in separate thread; so rust code is running in parallel to the scripting language interpreter.

CHAPTER 4

Glossary

state machine You probably know the theory. In this docs when we refer to state machine we refer to a type (most of the time the enum) that implements some trait designed according to the rules below. There is some introductory article about why state machines are designed that way.

State machine implements at least abstract `rotor::Machine` trait. But there are also more state machine traits that are more concrete.

Rules: TBD

child state machine Often one state machine calls an action from another state machine. The one that calls actions is a **parent**. The one that receives actions is called **child**. The parent state machine usually also owns the child state machine (means that when parent is shut down, the all the children too).

There might be multiple child state machines when the protocol allows multiple underlying requests/substreams/whatever to be mixed and used simultaneously

parent state machine See *child state machine*

Indices and tables

- genindex
- search