The Hitchhiker's Guide to The Actor Model in Rust

Ryan J. Kung

ryankung@ieee.org

January 25, 2019

Overview

- Introduction to The Actor Model
  - History of Actor Model
  - Hierarchies in Actor Model
  - Actor Model and Process Calculus
- Introduction to Actix & Actix-web
  - Actor & Arbiter
  - Message and Handler
- More trait Types in Actix
  - SyncArbiter & Supervised
- Ghost in the Shell
  - Tokio & Futures
- Issues & Solutions

History of Actor Model

A Universal Modular ACTOR Formalism for Artificial Intelligence.

Carl Hewitt, Peter Bishop, Richard Steiger
in
1973

Actor induction and meta-evaluation.

Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Maston, Rechard Steiger
in
1973

History of Actor Model

"Our formalism shows how all of the modes of behavior can be defined in terms of one kind of behavior: Sending message to actors, An actor is always invoked uniformly in exactly the same way regardless of whether it behaves: As a recursive function, data structure, or process"[1][2]

History of Actor Model

"It's van to multiply Entities beyond need. – William of Occam"
"Monotheism is the Answer"[1]

Hierarchies in Actor Model

- **Scheduling:**
  :- Every actor has a scheduler which determines when the actor actually acts after it is sent a message.
- **Intentions:**
  :- Every actor has an intention which makes certain that the prerequisites and context of the actor being sent the message are satisfied.
- **Monitoring:**
  :- Every actor can have monitors which look over each message sent to the actor.
- **binding:**
  :- Every actor can have a procedure for looking up the values of names that occur within it.
- **Resource Management:**
  :- Every actor has a banker which monitors the use of space and time.

Actor Model and Process Calculus

There are numerous process calculi[3, 4], each of them are using difference abstract method and formalize Symbol. And they (the *Processes* or *Agent*) usually sharing a *Channel*. Actually, we can also model the Actor Model with Algebra method, there existing some related work about it[5].

CSP	CCS	LOTOS
events $(e, e_1, \ldots)$	actions $(a, a_1, \ldots)$	actions or gates $(g, g_1, \ldots)$
processes $(P, P_1, \ldots)$	agents $(A, A_1, \ldots)$	behaviour expressions $(B, B_1, \ldots)$
[Hoare 1985, chs.1-3]	basic calculus	basic LOTOS
[Hoare 1985, chs.4-6]	value-passing calculus	full LOTOS
alphabet (αP)	sort $(\mathcal{L}(A))$	"events" or "actions"
channels $(c, c_1, \ldots)$	ports $(p, p_1, \ldots)$	gates $(g, g_1, \ldots)$
symbols	labels	"gates"
"domains"	value sets	sorts $(t, t_1, \ldots)$

Table 1: Terminology.

Introduction to Actix & Actix-web

Actix is a Actor Model implementation for Rust[6], which is Blazingfy Fast (about 28x faster than flask[7]), Type Safe, and easy to use.

Responses per second at 20 queries per request, Dell R440 Xeon Gold + 10 GbE (337 tests)

Rnk	Framework	Performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	actix-pg	45,719	100.0%	0	Mcr	Rus	Non	act	Lin	Pg	Lin	Raw	Rea
2	actix-raw	45,170	98.8%	0	Plt	Rus	Non	act	Lin	Pg	Lin	Raw	Rea
3	vertx-postgres	41,920	91.7%	0	Plt	Jav	ver	Non	Lin	Pg	Lin	Raw	Rea
4	vertx-web-postgres	38,617	84.5%	0	Mcr	Jav	vtx	Non	Lin	Pg	Lin	Raw	Rea
5	http4k	28,062	61.4%	0	Mcr	Kot	Svt	Non	Lin	Pg	Lin	Raw	Rea

Introduction to Actix & Actix-web

```rust
extern crate actix_web;
use actix_web::{server, App, HttpRequest, Responder};

fn greet(req: &HttpRequest) -> impl Responder {
    let to = req.match_info().get("name").unwrap_or("
        World");
    format!("Hello {}!", to)
}
fn main() {
    server::new(|| {
        App::new()
            .resource("/", |r| r.f(greet))
            .resource("/{name}", |r| r.f(greet))
    })
    .bind("127.0.0.1:8000")
    .expect("Can not bind to port 8000")
    .run();
}
```

Aribiter & Actor

Actor is the most basic trait of *actix*, which encapsulate state and behavior and defined as:

```rust
pub trait Actor: Sized + 'static {
    type Context: ActorContext;
    fn started(&mut self, ctx: &mut Self::Context)
    fn stopping(&mut self, ctx: &mut Self::Context)
    fn stopped(&mut self, ctx: &mut Self::Context)
    fn start(self) -> Addr<Self>
    fn create<F>(f: F) -> Addr<Self>
}
```

Aribiter & Actor

Arbiter is Event loop controller, which is also a Actor implementation, Arbiter controls event loop in its thread. Each arbiter runs in separate thread. Arbiter provides several api for event loop access.

```
impl Arbiter {
  pub fn current() -> Addr<Arbiter>;
  pub fn spawn<F>(future: F);
  pub fn start<A, F>(f: F) -> Addr<A>
}
```

And has follow *impl*s for controlling itself:

- *Actor*
- $Handler < StopArbiter >$
- $Handler < Execute < I, E >>$

Where $< StopArbbiter >$ and $< Execute < I, E >$ are impls of *Message* trait.

Message and Handler

The *Message* and *Handler* trait was defined as:

```
pub trait Message {
  type Result: 'static;
}

pub trait Handler<M>
where Self: Actor,
      M: Message
{
  type Result: MessageResponse<Self, M>;
  fn handle(&mut self, msg: M, ctx: &mut Self::
      Context) -> Self::Result;
}
```

Message and Handler

```
struct MyActor { count: usize };
struct Ping(usize);

impl Actor for MyActor {
    type Context = Context<Self>;
}

impl Message for Ping {
    type Result = usize;
  }

impl Handler<Ping> for MyActor {
  type Result = usize;
  fn handle(&mut self, msg: Ping, ctx: &mut Context<Self>) -> Self::Result {
    self.count += msg.0;
    self.count
  }
}
```

Message and Handler

```rust
fn main() {
    let system = System::new("test");

    // start new actor
    let addr = MyActor{count: 10}.start();

    // send message and get future for result
    let res = addr.send(Ping(10));

    Arbiter::spawn(
        res.map(|res| {
            println!("RESULT: {}", res == 20);
        })
        .map_err(|_| ()));

    system.run();
}
```

SyncArbiter

Sync actors could be used for cpu bound load.
Only one sync actor runs within arbiter's thread. Sync actor process one message at a time. Sync arbiter can start multiple threads with separate instance of actor in each.

SyncArbiter

By modified the *ping* example:

```
struct MyActor { count: usize };
struct Ping(usize);

impl Actor for MyActor {
    type Context = SyncContext<Self>;
}

impl Message for Ping {
    type Result = usize;
  }

impl Handler<Ping> for MyActor {
  type Result = usize;
  fn handle(&mut self, msg: Ping, ctx: &mut Context<
       Self>) -> Self::Result {
    self.count += msg.0;
    self.count
  }
}
```

SyncArbiter

And start 2 worker as Actor.

```
fn main() {
    System::run(|| {
        let addr = SyncArbiter::start(2, || MyArbiter);
    });
}
```

Supervised

Supervisor is a event manager which handler *restart* event for an *Actor*, and it's quiet useful for implementation an SystemRegistry(wont discuss here):

```
impl actix::Supervised for MyActor {
    fn restarting(&mut self, ctx: &mut Context<MyActor
        >) {
        println!("restarting");
    }
}
```

Tokio & Futures

A common usage of Actor Message calling is like this:

```
fn index(state: State<AppState>) -> FutureResponse<
    HttpResponse> {
  state
      .db
      .send(&some_msg)
      .from_err()
      .and_then(|res| mk_json_response(res))
      .responder()
  }
```

fn and_then is a method function of *ActorFuture* trait, which is very similar to a regular Future. and also requied a *poll* method:

```
fn poll(
    &mut self,
    srv: &mut Self::Actor,
    ctx: &mut <Self::Actor as Actor>::Context
) -> Poll<Self::Item, Self::Error>
```

Tokio & Futures

After calling *and_then*, it will return an *AndThen* which was created by the *Future* :: *and_then*.

```
pub struct AndThen<A, B, F> where
  A: Future,
  B: IntoFuture,  { /* fields omitted */ }
```

Which is meaning that the event loop of *Arbiter* is managing by Tokio and Futures.

Tokio & Futures

Let's go check what *actix* actually did when spawn an *Arbiter*

```
use tokio::executor::current_thread::spawn;
use tokio::runtime::current_thread::Builder as RuntimeBuilder;
impl ArbiterBuilder {
    fn new() -> Self {
        ArbiterBuilder {
            name: None,
            stop_system_on_panic: false,
            runtime: RuntimeBuilder::new(),
        }
}
```

No panic!

Addressed to Actix is using *tokio_threadpool* :: *park*, which is guarantee that the calling function *should* not panic. So panic in *Handler* may block the thread, and there is no default strategy for monitor and auto-restarting.
So which is means that you should wrap any returning value in Result or Option type, and do not panic!

Towel Needed

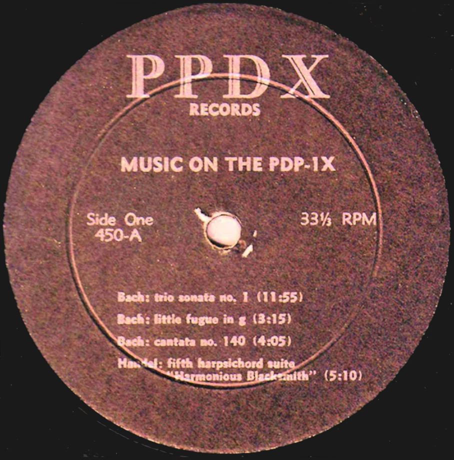
```
pub enum HandlerFn<Q, R> {
    Unary(fn(&ArbiterConnections) -> R),
    Binary(fn(&ArbiterConnections, &Q) -> R),
}

pub struct MixedQueryMessage<'a, Q, R> {
    pub query: Option<Q>,
    pub handler: &'a HandlerFn<Q, R>,
}

impl<'a, Q, R: 'static> Message for MixedQueryMessage<'a, Q, R>
where
    Q: Sync,
    R: Send,
{
    type Result = R;
}
```

Thanks

Programs should not only work, but they should appear to work as well.
– by PDP-1X Dogma



Reference I

- Richard Steiger Carl Hewitt, Peter Bishop.
  A universal modular actor formalism for artificial intelligence, 1973.
- Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger.
  Actor induction and meta-evaluation.
  pages 153–168, 10 1973.
- Colin Fidge.
  A comparative introduction to csp, ccs and lotos.
- J.C.M. Baeten.
  A brief history of process algebra.
- Gianluigi Zvattaro Mauro Gaspari.
  An algebra of actor.
- Nikolay Kim.
  Actix.rs.

Reference II

techempower.
Web framework benchmarks.