Actix

Actix是一个强大的Rust的actor系统, 在它之上是actix-web框架。这是你在工作中大多使用的东西。Actix-web给你提供了一个有趣且快速的Web开发框架。

我们称actix-web为小而务实的框架。对于所有的意图和目的来说，这是一个有少许曲折的微框架。如果你已经是一个Rust程序员，你可能会很快熟悉它，但即使你是来自另一种编程语言，你应该会发现actix-web很容易上手。

使用actix-web开发的应用程序将在本机可执行文件中包含HTTP服务器。你可以把它放在另一个像nginx这样的HTTP服务器上。即使完全不存在另一个HTTP服务器的情况下，actix-web也足以提供HTTP 1和HTTP 2支持以及SSL/TLS。这对于构建微服务分发非常有用。

最重要的是：actix-web可以稳定发布。

Awesome-Actix

快速入门

在开始编写`actix`应用程序之前，您需要安装Rust版本。我们建议您使用`rustup`来安装或配置此类版本。

安装Rust

在开始之前，我们需要使用rustup安装Rust：

```
curl https://sh.rustup.rs -sSf | sh
```

如果已经安装了`rustup`，请运行此命令以确保您拥有最新版本的Rust：

```
rustup update
```

actix框架需要Rust版本1.21及更高版本。

运行示例

开始试验actix的最快方法是克隆actix仓库并运行examples 目录中包含的示例。以下命令集运行`ping`示例：

```
git clone https://github.com/actix/actix
cargo run --example ping
```

查看examples/目录以获取更多示例.

开始

让我们创建并运行我们的第一个actix应用程序。我们将创建一个新的Cargo项目，该项目依赖于actix然后运行应用程序。

在上一节中，我们已经安装了所需的Rust版本 现在让我们创建新的Cargo项目。

Ping actor

让我们来写第一个actix应用程序吧！首先创建一个新的基于二进制的Cargo项目并切换到新目录：

```
cargo new actor-ping --bin
cd actor-ping
```

现在，通过确保您的Cargo.toml包含以下内容，将actix添加为项目的依赖项：

```
[dependencies]
actix = "0.7"
```

让我们创建一个接受Ping消息的actor，并使用处理的ping数进行响应。

actor是实现Actor trait的类型：

```
extern crate actix;
use actix::prelude::*;
struct MyActor {
    count: usize,
}
impl Actor for MyActor {
    type Context = Context<Self>;
}
fn main() {}
```

每个actor都有一个Context, 对于MyActor我们将使用Context<A>。有关actor上下文的更多信息，请参阅下一节。

现在我们需要定义actor需要接受的Message。消息可以是实现Messagetrait的任何类型。

```
extern crate actix;
use actix::prelude::*;
struct Ping(usize);
impl Message for Ping {
    type Result = usize;
```

```
}
fn main() {}
```

Messagetrait的主要目的是定义结果类型。 Ping消息定义usize，表示任何可以接受Ping消息的actor都需要返回usize值。

最后，我们需要声明我们的actor MyActor可以接受Ping并处理它。为此，actor需要实现Handler <Ping> trait。

```
extern crate actix;
use actix::prelude::*;
struct MyActor {
   count: usize,
}
impl Actor for MyActor {
    type Context = Context<Self>;
}
struct Ping(usize);
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
    }
}
fn main() {}
```

现在我们只需要启动我们的actor并向其发送消息。启动过程取决于actor的上下文实现。在我们的情况下可以使用Context <A>其基于tokio / future。我们可以用Actor :: start () 开始它或者Actor :: create () 。第一个是在可以立即创建actor实例时使用的。第二种方法用于我们在创建之前需要访问上下文对象的情况actor实例。对于MyActor actor，我们可以使用start () 。

与actor的所有通信都通过一个address。你可以do_send一条消息无需等待响应，或send给具有特定消息的actor。start () 和create () 都返回一个address对象。

在下面的示例中，我们将创建一个MyActor actor并发送一条消息。

```
extern crate actix;
extern crate futures;
use futures::Future;
use actix::prelude::*;
struct MyActor {
   count: usize,
}
impl Actor for MyActor {
```

```
    type Context = Context<Self>;
}
struct Ping(usize);
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
    }
}
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| {
            # System::current().stop();
            println!("RESULT: {}", res == 20);
        })
        .map_err(|_| ()));
    system.run();
}
```

Ping示例位于示例中。

Actor

Actix是一个Rust库，为开发并发应用程序提供了框架。

Actix建立在Actor Model上。允许将应用程序编写为一组独立执行但合作的应用程序 通过消息进行通信的"Actor"。 Actor是封装的对象状态和行为，并在actix库提供的**Actor System**中运行。

Actor在特定的执行上下文`Context`中运行,上下文对象仅在执行期间可用。每个Actor都有一个单独的执行上下文。执行上下文还控制actor的生命周期。

Actor通过交换消息进行通信。分派Actor可以可选择等待响应。Actor不是直接引用，而是通过引用地址

任何Rust类型都可以是一个actor，它只需要实现`Actor` trait。

为了能够处理特定消息`actor`必须提供的此消息的`Handler`实现。所有消息是静态类型的。消息可以以异步方式处理。Actor可以生成其他actor或将future/stream添加到执行上下文。`Actor` trait提供了几种允许控制`actor`生命周期的方法。

Actor生命周期

Started

`actor`总是以`Started`状态开始。在这种状态下，`actor`的`started ()` 方法被调用。 `Actor` trait为此方法提供了默认实现。在此状态期间可以使用actor上下文，并且actor可以启动更多actor或注册异步流或执行任何其他所需的配置。

Running

调用Actor的`started ()` 方法后，actor转换为`Running`状态。`actor`可以一直处于`running`状态。

Stopping

在以下情况下，Actor的执行状态将更改为`stopping`状态：

- `Context :: stop`由actor本身调用
- `actor`的所有地址都被消毁。即没有其他Actor引用它。
- 在上下文中没有注册事件对象。

一个actor可以通过创建一个新的地址或添加事件对象,并返回`Running :: Continue`，从而使`stopped`状态恢复到`running`状态，。

如果一个actor因为调用了`Context :: stop ()` 状态转换为`stop`，则上下文立即停止处理传入的消息并调用`Actor::stopping()`。如果`actor`没有恢复到`running`状态，那么全部未处理的消息被删除。

默认情况下，此方法返回`Running :: Stop`，确认停止操作。

Stopped

如果actor在停止状态期间没有修改执行上下文，则actor状态会转换到`Stopped`。这种状态被认为是最终状态，此时`actor`被消毁

Message

Actor通过发送消息与其他actor通信。在actix中所有消息是typed。消息可以是实现Message trait的任何rust类型。`Message :: Result`定义返回类型。让我们定义一个简单的`Ping`消息 - 接受此消息的actor需要返回`io::Result<bool>`。

```
extern crate actix;
use std::io;
use actix::prelude::*;
struct Ping;
impl Message for Ping {
    type Result = Result<bool, io::Error>;
}
fn main() {}
```

生成actor

如何开始一个actor取决于它的上下文（context）。产生一个新的异步`actor`是通过实现Actortrait 的`start`和`create`方法。它提供了几种不同的方式创造Actor;有关详细信息，请查看文档。

完整的例子

```
use std::io;
use actix::prelude::*;
use futures::Future;
/// Define message
struct Ping;
impl Message for Ping {
    type Result = Result<bool, io::Error>;
}
// Define actor
struct MyActor;
// Provide Actor implementation for our actor
impl Actor for MyActor {
    type Context = Context<Self>;
    fn started(&mut self, ctx: &mut Context<Self>) {
       println!("Actor is alive");
    }
    fn stopped(&mut self, ctx: &mut Context<Self>) {
       println!("Actor is stopped");
    }
}
/// Define handler for `Ping` message
impl Handler<Ping> for MyActor {
    type Result = Result<bool, io::Error>;
    fn handle(&mut self, msg: Ping, ctx: &mut Context<Self>) ->
Self::Result {
        println!("Ping received");
        Ok(true)
```

```
    }
}
fn main() {
    let sys = System::new("example");
    // Start MyActor in current thread
    let addr = MyActor.start();
    // Send Ping message.
    // send() message returns Future object, that resolves to message
result
    let result = addr.send(Ping);
    // spawn future to reactor
    Arbiter::spawn(
        result.map(|res| {
            match res {
                Ok(result) => println!("Got result: {}", result),
                Err(err) => println!("Got error: {}", err),
            }
        })
        .map_err(|e| {
            println!("Actor is probably died: {}", e);
        }));
    sys.run();
}
```

使用MessageResponse进行响应

让我们看看上面例子中为`impl Handler`定义的`Result`类型。看看我们如何返回`Result <bool, io :: Error>`？我们能够用这种类型响应我们的actor的传入消息，因为它具有为该类型实现的`MessageResponse` trait。这是该 trait的定义：

```
pub trait MessageResponse <A: Actor, M: Message> {
    fn handle <R: ResponseChannel <M >> (self, ctx: &mut A :: Context, tx:
Option <R>) ;
}
```

有时，使用没有为其实现此 trait的类型响应传入消息是有意义的。当发生这种情况时，我们可以自己实现这一trait。这是一个例子，我们用`GotPing`回复`Ping`消息，并用`GotPong`回复`Pong`消息。

```
use actix::dev::{MessageResponse, ResponseChannel};
use actix::prelude::*;
use futures::Future;
enum Messages {
    Ping,
    Pong,
}
enum Responses {
    GotPing,
    GotPong,
}
```

```
impl<A, M> MessageResponse<A, M> for Responses
where
    A: Actor,
    M: Message<Result = Responses>,
{
    fn handle<R: ResponseChannel<M>>(self, _: &mut A::Context, tx:
Option<R>) {
        if let Some(tx) = tx {
            tx.send(self);
        }
    }
}
impl Message for Messages {
    type Result = Responses;
}
// Define actor
struct MyActor;
// Provide Actor implementation for our actor
impl Actor for MyActor {
    type Context = Context<Self>;
    fn started(&mut self, ctx: &mut Context<Self>) {
        println!("Actor is alive");
    }
    fn stopped(&mut self, ctx: &mut Context<Self>) {
        println!("Actor is stopped");
    }
}
/// Define handler for `Messages` enum
impl Handler<Messages> for MyActor {
    type Result = Responses;
    fn handle(&mut self, msg: Messages, ctx: &mut Context<Self>) ->
Self::Result {
        match msg {
            Messages::Ping => Responses::GotPing,
            Messages::Pong => Responses::GotPong,
        }
    }
}
fn main() {
    let sys = System::new("example");
    // Start MyActor in current thread
    let addr = MyActor.start();
    // Send Ping message.
    // send() message returns Future object, that resolves to message
result
    let ping_future = addr.send(Messages::Ping);
    let pong_future = addr.send(Messages::Pong);
    // Spawn pong_future onto event loop
    Arbiter::spawn(
        pong_future
            .map(|res| {
                match res {
                    Responses::GotPing => println!("Ping received"),
                    Responses::GotPong => println!("Pong received"),
```

```
                }
            })
            .map_err(|e| {
                println!("Actor is probably died: {}", e);
            }),
    );
    // Spawn ping_future onto event loop
    Arbiter::spawn(
        ping_future
            .map(|res| {
                match res {
                    Responses::GotPing => println!("Ping received"),
                    Responses::GotPong => println!("Pong received"),
                }
            })
            .map_err(|e| {
                println!("Actor is probably died: {}", e);
            }),
    );
    sys.run();
}
```

Address

Actor通过交换消息进行通信。发送者可以选择等待回应。`actor`不能直接引用，只能通过他们的`Address`引用。

有几种方法可以获得`Actor`的`Address`。 `Actor` trait提供启动`actor`的两个辅助方法。两者都返回已启动`actor`的`Address`。

这是一个`Actors::start()`方法用法的例子。在这个例子中，`MyActor` `actor`是异步的，并且在与调用者相同的线程中启动。

```
extern crate actix;
use actix::prelude::*;
struct MyActor;
impl Actor for MyActor {
    type Context = Context<Self>;
}
fn main() {
System::new("test");
let addr = MyActor.start();
}
```

异步actor可以从`Context`对象获取其地址。Context需要实现`AsyncContext` trait。`AsyncContext::address()`提供了actor的`addres`。

```
extern crate actix;
use actix::prelude::*;
struct MyActor;
impl Actor for MyActor {
    type Context = Context<Self>;
    fn started(&mut self, ctx: &mut Context<Self>) {
       let addr = ctx.address();
    }
}
fn main() {}
```

Mailbox

所有消息首先转到actor的`mailbox`，然后是`actor`的执行上下文调用特定的消息处理。`mailbox`通常是`bounded`的。容量是特定于上下文实现。对于`Context`类型，容量设置为默认情况下有16条消息，可以增加Context:: set_mailbox_capacity()。

Message

为了能够处理特定消息, `actor`必须提供此消息的`Handler<M>`实现。所有消息都是静态类型的。消息可以以异步方式处理。actor可以产生其他`actor`或添加`future`或 `streams`到执行上下文。 `actor`trait提供了几种方

法允许控制actor的生命周期。

要向actor发送消息，需要使用Addr对象。 Addr提供了几个发送消息的方式。

- addr::do_send(M) - 这个方法忽略了actor的mailbox容量无条件地发送消息到邮箱。此方法不返回消息处理结果，如果actor不在，则无声地失败。
- Addr :: try_send(M) - 此方法尝试立即发送消息。如果mailbox已满或关闭（actor已死），此方法返回一个SendError。
- Addr :: send(M) - 此消息返回一个解析为future对象的结果,如果消息处理过程返回的Future对象被消毁，那么消息被取消。

Recipient

Recipient是address的特别版本，仅支持一种类型的message。它可以用于需要将消息发送给不同类型的actor的情况。可以使用Addr :: recipient () 从address创建recipient对象。

例如，recipient可以用于订阅系统。在以下示例中ProcessSignals actor向所有订阅者发送Signal消息。订户可以是任何实现Handler <Signal> trait 的actor。

```
#[macro_use] extern crate actix;
use actix::prelude::*;
#[derive(Message)]
struct Signal(usize);
/// Subscribe to process signals.
#[derive(Message)]
struct Subscribe(pub Recipient<Signal>);
/// Actor that provides signal subscriptions
struct ProcessSignals {
    subscribers: Vec<Recipient<Signal>>,
}
impl Actor for ProcessSignals {
    type Context = Context<Self>;
}
impl ProcessSignals {
    /// Send signal to all subscribers
    fn send_signal(&mut self, sig: usize) {
        for subscr in &self.subscribers {
           subscr.do_send(Signal(sig));
        }
    }
}
/// Subscribe to signals
impl Handler<Subscribe> for ProcessSignals {
    type Result = ();
    fn handle(&mut self, msg: Subscribe, _: &mut Self::Context) {
        self.subscribers.push(msg.0);
    }
}
fn main() {}
```

Context

Actors都维护内部执行上下文或状态。 这允许actor确定它自己的地址，更改邮箱限制或停止执行。

邮箱(Mailbox)

所有消息首先进入actor的邮箱，然后actor的执行上下文调用特定的消息处理程序。邮箱通常是有界的。容量特定于上下文实现。对于`Context`类型，默认情况下容量设置为16条消息，可以使用`Context :: set_mailbox_capacity ()`增加 容量。

```
extern crate actix;
use actix::prelude::*;
struct MyActor;
impl Actor for MyActor {
    type Context = Context<Self>;

    fn started(&mut self, ctx: &mut Self::Context) {
        ctx.set_mailbox_capacity(1);
    }
}

fn main() {
System::new("test");
let addr = MyActor.start();
}
```

请记住，这不适用于绕过邮箱队列限制的`Addr :: do_send (M)` ，或者完全绕过邮箱的`AsyncContext :: notify (M)`和`AsyncContext :: notify_later (M, Duration)` 。

获取`Actor`地址

Actor可以从它的上下文中查看它自己的地址。 也许您想要稍后重新排队事件，或者您想要转换消息类型。 也许你想用你的地址回复一条消息。 如果您希望让`actor`向他自己发送消息，请查看`AsyncContext :: notify (M)` 。

要从上下文中获取地址，请调用`Context :: address ()` 。 一个例子是：

```
struct MyActor;

struct WhoAmI;

impl Message for WhoAmI {
    type Result = Result<actix::Addr<MyActor>, ()>;
}

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

```
}

impl Handler<WhoAmI> for MyActor {
    type Result = Result<actix::Addr<MyActor>, ()>;

    fn handle(&mut self, msg: WhoAmI, ctx: &mut Context<Self>) ->
Self::Result {
        Ok(ctx.address())
    }
}

let who_addr = addr.do_send(WhoAmI {} );
```

停止Actor

从actor执行上下文中，您可以选择阻止actor处理任何未来的邮箱消息。 这可能是对错误情况的响应，也可能是程序关闭的一部分。 为此，您可以调用`Context :: stop ()` .

这是一个调整后的`Ping`示例，在收到4个ping后停止。

```
extern crate actix;
extern crate futures;
use futures::Future;
use actix::prelude::*;
struct MyActor {
   count: usize,
}
impl Actor for MyActor {
    type Context = Context<Self>;
}

struct Ping(usize);

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;

        if self.count > 5 {
            println!("Shutting down ping receiver.");
            ctx.stop()
        }

        self.count
    }
}
```

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

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

    // send message and get future for result
    let addr_2 = addr.clone();
    let res = addr.send(Ping(6));

    Arbiter::spawn(
        res.map(move |res| {
            // Now, the ping actor should have stopped, so a second
message will fail
            // With a SendError::Closed
            assert!(addr_2.try_send(Ping(6)).is_err());

            // Shutdown gracefully now.
            System::current().stop();
        })
        .map_err(|_| ()));

    system.run();
}
```

Arbiter

仲裁者(Arbiters)为`actor`提供异步执行上下文。 如果Actor包含定义其Actor特定执行状态的`Context`，则`Arbiters`将托管`actor`运行的环境。

因此`Arbiters`执行许多功能。 最值得注意的是，它们能够生成新的OS线程，运行事件循环，在该事件循环上异步生成任务，并充当异步任务的帮助程序。

系统和仲裁者

在我们之前的所有代码示例中，函数`System :: new`为您的`actor`创建了一个`Arbiter`。 当你在`actor`上调用`start ()` 时，它会在`System Arbiter`的线程内部运行。 在许多情况下，这是使用`Actix`的程序所需的全部内容。

使用Arbiter解析异步事件

如果您不是生锈`futures`的专家，`Arbiter`可以是一个有用且简单的包装器，可以按顺序解析异步事件。 考虑我们有两个`actor`，A和B，我们想在A的完成后取得结果时就立即在B上运行一个事件。 我们可以使用`Arbiter :: spawn`来协助完成这项任务。

```
extern crate actix;
extern crate futures;
use futures::Future;
use actix::prelude::*;

struct SumActor {}

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

struct Value(usize, usize);

impl Message for Value {
   type Result = usize;
}

impl Handler<Value> for SumActor {
    type Result = usize;

    fn handle(&mut self, msg: Value, ctx: &mut Context<Self>) ->
Self::Result {
        msg.0 + msg.1
    }
}

struct DisplayActor {}

impl Actor for DisplayActor {
```

```
    type Context = Context<Self>;
}

struct Display(usize);

impl Message for Display {
   type Result = ();
}

impl Handler<Display> for DisplayActor {
    type Result = ();

    fn handle(&mut self, msg: Display, ctx: &mut Context<Self>) ->
Self::Result {
        println!("Got {:?}", msg.0);
    }
}


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

    // start new actor
    let sum_addr = SumActor{}.start();
    let dis_addr = DisplayActor{}.start();

    let res = sum_addr.send(Value(6, 7));

    Arbiter::spawn(
        res.map(move |res| {

            let dis_res = dis_addr.send(Display(res));

            Arbiter::spawn(
                dis_res.map(move |_| {
                    // Shutdown gracefully now.
                    System::current().stop();
                })
                .map_err(|_| ())
            );

        }) // end res.map
        .map_err(|_| ())
    );

    system.run();
}
```

SyncArbiter

当您正常运行Actors时，系统的Arbiter线程上会运行多个Actors，使用它的事件循环。 但是，对于CPU绑定工作负载或高度并发工作负载，您可能希望让一个Actor并行运行多个实例。

这就是SyncArbiter提供的功能 - 能够在OS线程池上启动Actor的多个实例。

重要的是要注意SyncArbiter只能托管单一类型的Actor。 这意味着您需要为要以此方式运行的每种类型的Actor创建SyncArbiter。

创建同步Actor

当实现要在SyncArbiter上运行的Actor时，它需要将Actor的Context从Context更改为SyncContext。

```
extern crate actix;
use actix::prelude::*;

struct MySyncActor;

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

fn main() {
System::new("test");
}
```

启动同步仲裁器

现在我们已经定义了一个Sync Actor，我们可以在一个由我们的SyncArbiter创建的线程池上运行它。 我们只能在SyncArbiter创建时控制线程数 - 我们以后不能添加/删除线程。

```
extern crate actix;
use actix::prelude::*;

struct MySyncActor;

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

fn main() {
System::new("test");
let addr = SyncArbiter::start(2, || MySyncActor);
}
```

我们可以使用与我们之前开始的`Actors`相同的方式与`addr`进行通信。 我们可以发送消息，接收`futures`和结果等。

同步Actor邮箱

`Sync Actors`没有邮箱限制，但您仍应使用`do_send`，`try_send`、`send` 正常发送以解决其他可能的错误或同步与异步行为。