从Erlang开始了解Actor模型

2020-03-31

Actor Model是一个宽泛的概念，早在上个世纪就被提出来，它将Actor视作一个整体，可以是原子变量，也可以是一个实体，也可以代表一个线程，Actor之间相互通信，每个Actor都有自己的状态，在接收到其他Actor的消息后可以改变自己的状态，或者做一些其他事情。一般提到Actor，会用Erlang、Elixir或Akka来举例，它们都在一定程度上实现了Actor模型。

前端的MVVM框架React、Vue等都有各自的数据流管理框架，比如Redux和Vuex，这些数据流管理框架中有几个类似的概念，Action、Reducer、State之类，这些概念有时候会让人感到迷惑。现在前端变得越来越复杂，其中有一些东西可能是借鉴后端的，像TypeScript的类型系统。我好奇这些前端框架里的Action和后端的Actor模型在概念上是否有相似的地方。

其实Action的本质是简单的，甚至代码的原理也是简单的，reducer里面用switch判断不同的操作类型，去调不同的方法。最简化的形式就是一个方法Action改变了全局变量state的值。Redux文档里说它的设计来自Flux架构，Flux架构的来源暂时不得而知，但也不太可能说是受到了Actor模型的启发。

```
let state = null
function action(val) {
  state = val
}
```

Erlang是一门古老的编程语言，也是一门典型的受Actor Model启发的编程语言。单纯去理解概念是空泛的，从具体的、特定的语言入手也许能帮助我们探索这些理论。就像学习FP，选择Haskell要好过Java很多倍。Elixir是基于Erlang虚拟机的一门语言，与Erlang的关系类似Scala和Java的关系，也因此Erlang的语法相对简单和干净一点。

Erlang

Erlang的代码块以.结尾，代码块可能只有一行，也可以有多行，.的作用类似于`}`，只是Erlang里没有`{`。代码块内的语句以,结尾，意味一个语句的结束，相当于一些语言的`;`。

Erlang将一个程序文件定义为一个模块，在命令行中使用`c(test).`可以加载模块。模块名称必须和文件名称一致：

```
-module(test).
```

文件头部需要定义程序export的函数，这是模块的出口：

```
-export([start/0, ping/3, pong/0]).
```

这里导出了3个函数，方括号和其他语言一样表示数组，函数名称后面的`/0`、`/3`指函数参数的个数。start函数将作为程序的主入口，负责启动整个程序，ping负责发送消息，pong负责接收消息并做出响应。

Erlang里面有个`process`的概念，它不是线程，也不是指计算机层面的进程，它就是`process`，或者也能把它当做线程，但是要明白它和线程不一样。我们将启动两个process，一个负责ping，一个负责pong，模拟消息的传输和交互。可以类比启动了两个线程，一个负责生产，一个负责消费。

```
ping(0, Pong_PID, StartTime) ->
    Pong_PID ! {finished, StartTime};
```

这是ping函数的第一部分，是ping函数的一个分支，接收3个参数，如果第一个参数是0，就会执行这个函数中的语句。第二个参数`Pong_PID`指包含pong的process，第三个参数指程序启动的时间，用于记录程序的运行时长。函数体内只有一个语句，`!`是发送消息的意思，意为将数据`{finished, StartTime}`发送到id为`Pong_PID`的process中，其中finished是一个`Atom`，作为标识

发送到pong那里。Atom是Erlang的数据类型之一，相当于……不需要声明的常量。

```
ping(N, Pong_PID, StartTime) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("~w~n", [N])
    end,
    ping(N - 1, Pong_PID, StartTime).
```

这是ping函数的第二部分，如果函数接收到的第一个参数不等于0，就会执行这个函数内的语句。这一部分函数在接收到请求后，首先会做和分支一同样的事情，就是把数据`{ping, self()}`发送给pong，区别在于这里的标识为`ping`而不是`finished`，pong那里会根据这个标识做不同的操作，至于第二个参数，`self()`会返回当前process的id，也就是把ping的id传给了pong，用以pong回复消息。pong会选择性的使用第二个参数。

把数据发送到pong之后，有一个`receive ... end`的代码段，这个代码段会阻塞当前程序的执行，直到当前process接收到数据。代码段里是一个简单的模式匹配，`pong`是一个Atom类型的变量，如果接收到pong这样的标识，就会执行`->`后面的语句。`io:format`是一个简单的格式化输出，把N的值打印到屏幕上。

receive结束之后，马上又调了一下ping自己，递归……直到N为0，也就是说ping和pong的交互会持续N次，`io:format`那里会把交互次数打印出来。这是ping函数的两个分支，pong函数和ping函数的程序类似：

```
pong() ->
    receive
        {finished, StartTime} ->
            io:format("The End");
            io:format("~w~n", [erlang:timestamp()]);
            io:format("~w~n", [StartTime]);
        {ping, Ping_PID} ->
            Ping_PID ! pong,
            pong()
    end.
```

pong函数在入参层面没有分支，但是receive里有两种匹配，如果接收到了结束标识finished，会把开始时间和结束时间都打印出来，然后程序结束。如果接收到的标识是ping而不是finished，首先给Ping_PID也就是ping的process一个pong的响应，然后调了一遍自己，相当于先发了一个消息出去，接着自己等待消息的回复，如果没有收到回复，它就一直等着。

```
start() ->
    Pong_PID = spawn(test, pong, []),
    spawn(test, ping, [10, Pong_PID, erlang:timestamp()]).
```

最后是start函数，程序的入口函数，spawn了两个process，这两个process分别单独地运行。当传入ping的第一个参数为10，ping和pong的交互将持续10次。

交互速率

以前听到过一个所谓的“大牛”讲，我们现在想要提高计算机的速率，瓶颈是什么呢，我们应该往哪个方向努力呢，应该是CPU的利用率，Actor是很快的，为什么快呢，因为一个Actor就是一个整体，一个Actor只在一个内核中运行，连CPU内核之间的交互都省了……这种说法的正确性可能有待验证，不过Actor是否真的快呢，我有点好奇，也因此萌生了测试一下Actor速度的想法。

必须要说明的是，我也相当清楚，这种测试方法很不靠谱。

在Erlang程序里启动两个process，两个process之间相互通信，测试不同数量级的通信次数，记录下程序执行所花费的时间。与Erlang作为对比，在Java里启动两个线程，用线程的睡眠和唤醒实现线程间的通信。同样的，在Go语言里用两个协程通信。至于Akka……其实也是Actor的代表。下表是测试之后的结果，次数从1到1亿，时间单位为毫秒。

次数	Erlang	Java	Go	Akka
1	0	0	0	3
10	0	1	0	7
100	3	4	1	17
1,000	26	30	4	83
10,000	610	168	42	225
100,000	2783	1295	404	674
1,000,000	27,085	11,300	4489	3515
10,000,000	273,912	107,673	40,335	29,368
100,000,000	2,851,680	1,092,879	482,196	300,228

本来尝试用Echarts之类渲染一下这些数据，方便对比，后来发现这些数据绘制出来的折线图并不友好。

总的来看，Erlang的速度是最慢的，这可能和Erlang历史悠久有关，也许是因为没有得到足够的优化，相信Elixir的速度会好一些。相较之下，Java的速度胜过Erlang，Go语言的速度胜过Java，这似乎是意料之中的事情。Java的耗时是Erlang的1/3，Go语言的耗时是Java的1/2。

最让人惊讶的在于，Akka的Actor速度竟然比Go语言的协程还要快。在交互1000次之前，Akka的速度比Erlang还要慢，在10K数量级的时候，它的速度超过了Erlang，在100K数量级的时候，速度超过了Java，直到1M数量级的时候，Akka超过了Go语言，并且一直保持领先。这是一个令人难以置信的结果，同样是运行在JVM上，Akka的耗时是Java的1/3，可能Java线程间的交互确实带来了很大的开销。

没有用Elixir做测试是一个遗憾。关于Akka为什快，和Actor模型有没有关系，有多大的关系，还需要进一步探索。

(The End)

Akka

用来做测试的Akka程序是Akka官方的Hello Wrold程序，能看到明显的Actor模型的影子，尤其是`!`运算符和`receive`方法。

```
import akka.actor.typed.ActorRef
import akka.actor.typed.ActorSystem
import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import GreeterMain.SayHello
```

这是导入部分，如果使用VS Code之类的编辑器，这段代码还是很重要的。和Erlang的程序类似，有一个发消息的Greeter和一个接收并回复消息的GreeterBot，另外还有一个主方法。

```
object Greeter {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String, from: ActorRef[Greet])

  def apply(): Behavior[Greet] =
    Behaviors.receive { (context, message) =>
      message.replyTo ! Greeted(message.whom, context.self)
      Behaviors.same
    }
}
```

这是发消息的Greeter，当Greeter作为函数被调用，会自动执行apply中的代码。apply方法是一个receive，和Erlang的receive一样会阻塞程序直到Actor接收到消息。replyTo是GreeterBot的”pid”，Greeter接收到消息后会回复消息给GreeterBot。

```
object GreeterBot {
  var startTime = System.currentTimeMillis()

  def apply(max: Int) = {
    bot(0, max)
  }

  private def bot(greetingCounter: Int, max: Int): Behavior[Greeter.Greeted] =
    Behaviors.receive { (context, message) =>
      val n = greetingCounter + 1
      context.log.info("{}", n)
      if (n >= max) {
        context.log.info("The End | {}", System.currentTimeMillis() - startTime)
        Behaviors.stopped
      } else {
        message.from ! Greeter.Greet(message.whom, context.self)
        bot(n, max)
      }
    }
}
```

这是GreeterBot，和Erlang简洁的代码比起来，Scala冗长的类型声明可能显得有些……烦杂。GreeterBot接收到来自Greeter的消息后，判断n是否为max，如果已经执行够次数了，就停止，否则调用自己进行递归。

```
object GreeterMain {

  final case class SayHello(name: String)

  def apply(): Behavior[SayHello] =
    Behaviors.setup { context =>
      val greeter = context.spawn(Greeter(), "greeter")

      Behaviors.receiveMessage { message =>
        val replyTo = context.spawn(GreeterBot(max = 10), message.name)
        greeter ! Greeter.Greet(message.name, replyTo)
        Behaviors.same
      }
    }
}

object AkkaQuickstart extends App {
  val greeterMain = ActorSystem(GreeterMain(), "AkkaQuickStart")
  greeterMain ! SayHello("Charles")
}
```

最后是主方法，看着可能也有点……长。继承于App的类是能够运行的主类，向Actor系统中注册了GreetMain，同时GreetMain的apply方法被执行了一次。GreetMain里spawn了两个process，和Erlang的程序行为是类似的。

Go

Go语言的程序真的要简洁很多，这是程序头部：

```
package main

import(
  "fmt"
  "time"
)

var maxCount = 100000000
var startTime = time.Now().UnixNano() / 1e6
```

定义了两个变量，一个是程序执行次数，一个是程序开始时间。

```
func main() {
  ch := make(chan bool)
  exit := make(chan bool)

  go func() {
    for i := 0; i < maxCount; i++ {
      fmt.Println(i)
      <- ch
      ch <- true
    }
  }()

  go func() {
    defer func() {
      timeUsed := time.Now().UnixNano() / 1e6 - startTime
      fmt.Println("The End | ", timeUsed)
      close(ch)
      close(exit)
    }()
    for i := 0; i < maxCount; i++ {
      ch <- true
      <- ch
    }
  }()

  <- exit
}
```

两个协程，从channel中取数据和向channel中写数据交替。Go语言的程序看着清爽太多了，Scala扎眼睛。

Java

Java的冗长程度不比Scala轻。

```
public class Test{
    public static void main(String[] args) {
        Object lock = new Object();
        Thread sender = new Sender(lock);
        Thread receiver = new Receiver(lock);
        sender.start();
        receiver.start();
    }
}
```

主方法里启动了两个线程，锁是共享资源。

```
class Message {
    static long MAX_COUNT = 100000000;
    static String status = new String("init");
    static long count = 0;
    static long startTime = 0;
    public static void send() {
        System.out.println(count);
        status = "sent";
        count++;
        if (count == 1) {
            startTime = System.currentTimeMillis();
        }
        if (count >= MAX_COUNT) {
            status = "stop";
            long time = System.currentTimeMillis() - startTime;
            System.out.println("The End | " + time);
        }
    }
    public static void receive() {
        status = "received";
```

```
    }
    public static String getStatus() {
        return status;
    }
}
```

Message是临界资源，储存消息的内容。消息内容变更时做了一点其他的事情，把需要的日志打印到屏幕上。

```
class Sender extends Thread {
    Object lock = null;
    public Sender(Object lock) {
        this.lock = lock;
    }
    @Override
    public void run() {
        while (!Message.getStatus().equals("stop")) {
            synchronized (lock) {
                if (Message.getStatus().equals("init")
                  || Message.getStatus().equals("received")) {
                    Message.send();
                    lock.notify();
                    try {
                        lock.wait();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

class Receiver extends Thread {
    Object lock = null;
    public Receiver(Object lock) {
        this.lock = lock;
    }
    @Override
    public void run() {
        while (!Message.getStatus().equals("stop")) {
            synchronized (lock) {
                if (Message.getStatus().equals("sent")) {
                    Message.receive();
                    lock.notify();
                    try {
                        lock.wait();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

Sender和Receiver的程序类似，Sender先发送消息，然后wait，等着接收Receiver的消息，Receiver用while不停地判断有没有收到消息，如果有则回复消息，并且唤醒Sender，通知它该处理消息了，叫醒Sender后自己wait，等着Sender的反馈。