当函数式编程遇到并行计算 - Elixir

接触elixir实在是一个机缘巧合，当时在处理并行计算但一直由于没有找到合适的语言或框架而苦恼。在现如今的多种机器语言中，大多支持多线程并发执行，但是只有elixir是真正意义上的并行语言。当提到elixir的时候大家可能会想到更多的是函数式编程语法，没错它确实本质上是一个纯函数式的编程语言，但是同时具备了并行运行能力。本文带你进入elixir世界，看看其并行处理能力和应用场景。

# 并行（Parallel）和并发（Concurrent）的区别

很多开发者其实并没有搞清楚并行和并发的区别，相当一部分人认为他们是同一个概念，特别是在NodeJS盛行的时代，单线程、NIO成为构架的首选结构，这使得程序员一度忽略并发并且不愿意触碰并发这个地带。在大部分后台应用场景，例如：大数据、深度计算、神经网络等等，都需要借助并行执行的优势来提升运行效率。当然，相比NIO那种单线程构架并行计算会带来很多类似死锁、共享资源等问题，但是为了达到执行效果，这些都是我们应当可以克服的苦难。

## 并发

大部分程序或多或少都使用了并发，特别是在多线程环境下。当有多个线程在操作时，这些现成不一定是真正的在系统上并行运行，如果系统只有一个CPU，则它根本不可能真正同时进行一个以上的线程。它只能把CPU运行时间划分成若干个时间段，再将时间段分配给各个线程执行，在一个时间段的线程运行时，其它线程处于挂起状态。这种方式我们称之为并发(Concurrent)。

## 并行

同样是多线程的例子，当系统有一个以上CPU时或者一个CPU支持多核，则线程的操作有可能非并发。当一个CPU Core执行一个线程时，另一个CPU Core可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。

与大部分语言支持的多线程模型不同的是elixir支持的并行运行是在多进程基础上，处理多进程间的关系不是一件容易的事情，通常有下面几种方式：

* 共享内存  
  可以说这是最常用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。
* 信号  
  它是在软件层次上对中断机制的一种模拟，是一种比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致得。在使用Linux操作系统时经常需要进行kill操作，kill命令就是给进程发送一个中断信号。
* 管道  
  管道可用于具有亲缘关系进程间的通信。
* Socket  
  这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，可以采用tcp或udp的通讯方式。

不论采用什么方式，多进程之间的关系都是一个不容易维护复杂任务。但是elixir对多进程管理方面提供了更为方便的解决方案，对于程序员来讲这样的语言实现了操作多进程就如同多线程一样容易。

# Elixir的诞生

在系统对实时性要求越来越高的今天，并行计算和并行开发是应用程序的基本需求，一些编程语言也对其进行了相应的支持。Elixir就是在这样的环境下诞生的一款并行函数式语言，他的语法很类似Ruby，是建立在Erlang之上并且具备了Erlang的运行环境所具备的所有优势，同时支持Erlang的各种第三方库函数。

# Elixir的多进程处理

为了解释并行带来的性能提升，我给大家写几段简单的代码了解一下他的执行过程。代码采用elixir语言，对于不熟悉elixir的同学不要着急，有过编程经验的程序员都可以看懂下面的代码。

request\_data = fn(id) ->

:timer.sleep(1000)

"{data: {id: #{id}}}"

end

上面的代码定义了一个请求数据的函数，为了模拟请求数据的后台我们假定请求时间是1秒，函数中通过sleep一秒来实现这个停顿效果。然后返回数据的json格式。很简单也很容易理解。在运行的时候，程序会等待一秒然后返回json结果，如下所示：

iex(9)> request\_data.(100)

"{data: {id: 100}}"

如果我们要请求10条记录会怎样：

iex(9)> Enum.map(1..10, &request\_data.(&1))

["{data: {id: 1}}", "{data: {id: 2}}", "{data: {id: 3}}", "{data: {id: 4}}",

"{data: {id: 5}}", "{data: {id: 6}}", "{data: {id: 7}}", "{data: {id: 8}}",

"{data: {id: 9}}", "{data: {id: 10}}"]

程序会等待10秒，然后输出结果。为了使产品运行的相对流畅，我们需要解决这个10秒的停顿，这时就需要并行运行程序。我们先来定义一个并行函数来调用刚才的数据请求：

async\_get\_request = fn(id) ->  
 spawn(fn -> IO.puts(get\_request.(id)) end)  
end

iex(12)> async\_get\_request.(100)  
#PID<0.119.0> // 等待1秒  
{data: {id: 100}}

运行后程序会立刻返回然后等待1秒输出json数据结果。async\_get\_request/1 是一个异步函数，在运行时它调用 spawn/1 创建一个子进程用来执行get\_request请求并立刻返回并不等待子进程运行结束。spawn出来的子进程会在自己的context下运行，这个子进程是一个非常轻量的进程，他可以通过消息与父进程通信，所以我们会先看到#PID<0.119.0>的输出，之后等待1秒出现json数据。

在同样请求10个数据的时候，我们得到下面的输出：

iex(13)> Enum.map(1..10, &async\_get\_request.(&1))  
[#PID<0.122.0>, #PID<0.123.0>, #PID<0.124.0>, #PID<0.125.0>, #PID<0.126.0>,  
 #PID<0.127.0>, #PID<0.128.0>, #PID<0.129.0>, #PID<0.130.0>, #PID<0.131.0>]  
{data: {id: 1}}  
{data: {id: 2}}  
{data: {id: 3}}  
{data: {id: 4}}  
{data: {id: 5}}  
{data: {id: 6}}  
{data: {id: 7}}  
{data: {id: 8}}  
{data: {id: 9}}  
{data: {id: 10}}

程序首先输出10个进程ID，之后等待1秒输出每个进程的返回结果。在上面的第一个例子中我们等待了10秒的时间才看到全部输出。这是因为CPU在并行运行10个进程并没有按顺序一个一个的运行，所以总程序的运行时间缩短到了1秒。

关于spawn你可能会有很多疑问，下面我给出几个常见问题的解答：

* spawn一个子进程是否阻塞父进程的运行  
  不会的，父进程会等待子进程被创建出来但并不等待子进程的返回。
* spawn出的子进程id是随机产生的吗？  
  并不全是，但是你不能控制这个id的生成。
* 它可以接收参数吗？  
  spawn/1不可以，但是spawn/3是可以接收参数的。
* 子进程何时结束  
  当子进程运行完以后就会退出。
* 如何得到子进程的运行结果  
  可以通过send/2收集子进程的运行结果，在下面的例子中我会给大家介绍如何使用。

上面的例子只是把结果打印到了命令行，这并不能体现elixir在处理多进程时的优势，通常情况下我们需要获得结果并继续后面的逻辑操作。我把代码更新为下面的样子：

async\_get\_request = fn(id) ->  
 caller = self  
 spawn(fn ->  
 send(caller, {:result, request\_data.(id)})  
 end)  
end

修改后的async\_get\_request/1 函数内部定义了调用子进程时的源进程caller，并通过send/2 函数把子进程的结果发给父进程。看一卡下面的代码：

iex(28)> Enum.map(1..10, &async\_get\_request.(&1))   
[#PID<0.297.0>, #PID<0.298.0>, #PID<0.299.0>, #PID<0.300.0>, #PID<0.301.0>,  
 #PID<0.302.0>, #PID<0.303.0>, #PID<0.304.0>, #PID<0.305.0>, #PID<0.306.0>]  
iex(29)> flush  
{:result, "{data: {id: 1}}"}  
{:result, "{data: {id: 2}}"}  
{:result, "{data: {id: 3}}"}  
{:result, "{data: {id: 4}}"}  
{:result, "{data: {id: 5}}"}  
{:result, "{data: {id: 6}}"}  
{:result, "{data: {id: 7}}"}  
{:result, "{data: {id: 8}}"}  
{:result, "{data: {id: 9}}"}  
{:result, "{data: {id: 10}}"}   
:ok

刚才提到过spawn/1函数创建的子进程通过消息与父进程通讯，在调用完10次异步请求以后通过flush将消息队列打印出来，可以看到所有的结果都已经收集到父进程的消息队列中。

下面我们把函数式编程风格和并行计算合并到一起，为了获得结构先定义一个get\_result/0函数：

get\_result = fn ->  
 receive do  
 {:result, result} -> result  
 end  
End

之后定义函数式如下：

1..10  
|> Enum.map(&async\_get\_request.(&1))  
|> Enum.map(fn(\_) -> get\_result.() end)

["{data: {id: 1}}", "{data: {id: 2}}", "{data: {id: 3}}", "{data: {id: 4}}",  
 "{data: {id: 5}}", "{data: {id: 6}}", "{data: {id: 7}}", "{data: {id: 8}}",  
 "{data: {id: 9}}", "{data: {id: 10}}"]

上面的代码先定义一个从1到10的数值序列，然后传到异步调用async\_get\_request/1中，map便利其返回结果并传到get\_result/0里并打印。由于共有10个数值所以会遍历10次。

# 深入分析Elixir多进程

## 从Sender/Receiver开始

上面的例子只是简单介绍了一下Elixir如何创建进程，如何得到进程的运行结果。下面我们更深入的分析一下Elixir在使用多进程时的具体细节。为此我从一个sender／receiver的场景为例来为大家说明一下他的用法。

先定义一个Receiver：

defmodule Receiver do  
 def receiver do  
 receive do  
 :receiver -> IO.puts('received ping')  
 end  
 end  
  
 def start do  
 spawn(\_\_MODULE\_\_, :receiver, [])  
 end  
end

该Receiver在start的时候创建一个子进程，子进程中接收输入的消息。下面可以运行该程序并打印输出：

iex(2)> receiver = Receiver.start

#PID<0.101.0>

iex(3)> Process.alive? receiver # 检查进程是否正在运行

true

iex(7)> send receiver, :receiver # 向Receiver发送一个消息

received ping # Receiver收到消息并打印

:receiver

iex(8)> Process.alive? receiver # 当Receiver收到消息以后，进程运行结束，所以返回false

false

在上面的例子中，我们的Receiver在收到第一个消息后就结束运行并退出，这是因为当收到第一个消息以后，程序会继续运行，而IO.puts()下面的代码是end，所以进程会退出执行。为了让进程持续保持接收消息状态我们修改代码如下：

def receiver do  
 receive do # 阻塞程序直到收到一条消息  
 :ping -> IO.puts('received ping')  
 end  
 receiver # 处理消息以后递归调用自己  
end

在end之后加上一个调用自己的方法从而实现继续监听消息。运行以后你会发现你可以持续发送消息给Receiver，

iex(12)> send receiver, :receiver

received ping

:receiver

iex(13)> send receiver, :receiver

received ping

:receiver

iex(14)> send receiver, :receiver

received ping

:receiver

iex(15)> Process.alive? receiver

true

这里有人可能会有顾虑，在新添加的那行递归调用自身的代码会不会导致方法调用栈益处，在其他语言中这确实是一个问题，但是Elixir在编译时对其进行了特别处理，把递归调用编译成一个循环，所以这里不会出现栈益处异常。Erlang提供的observer可以观察到进程运行的消耗。

iex(16)> :observer.start

:ok

iex(17)> p = Receiver.start

#PID<0.1165.0>

iex(18)> 1..100000000 |> Stream.map(fn \_ -> send p, {self(), :receiver} end) |> Enum.count

上面先调用Observer窗口，然后运行一亿个进程，在Observer中的Load Charts可以看到内存和IO使用情况，

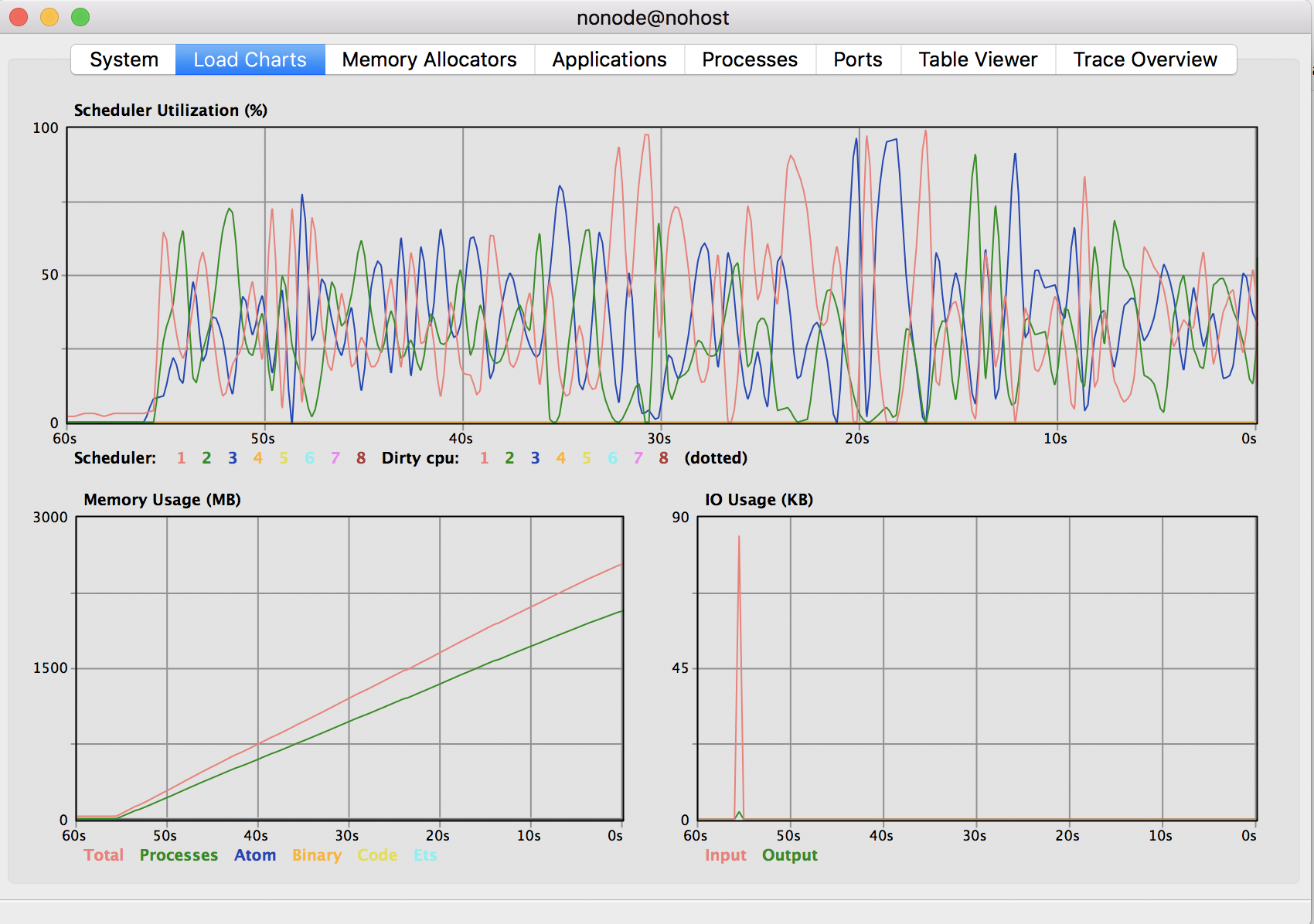


图1: Observer Load Charts

在下图中可以观察进程的运行情况：

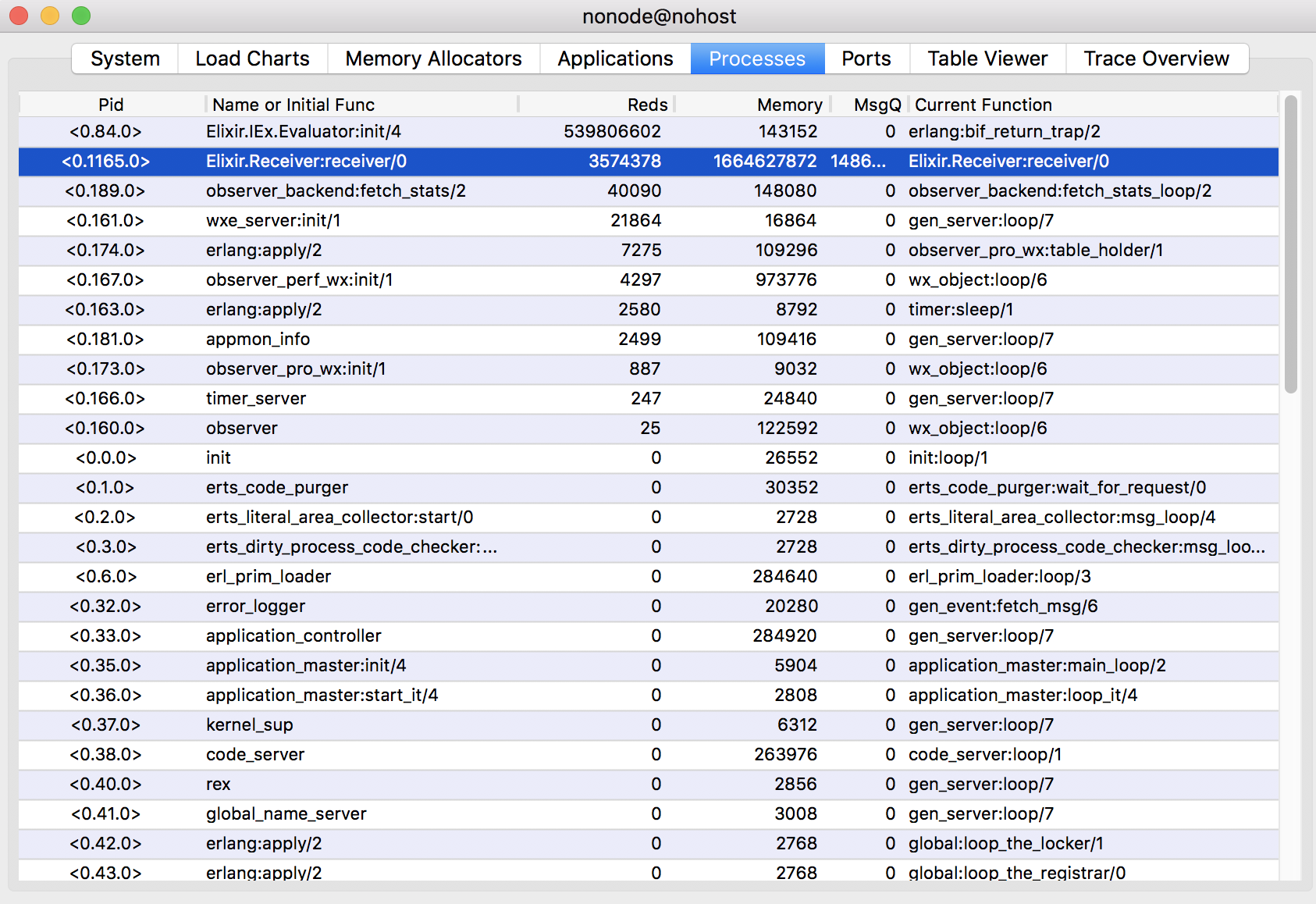


图2: Observer Processes

## 理解Self

在Elixir中，self指的是“我是谁”。为了理解它，我给大家写了下面的代码：

iex(5)> :observer.start()

iex(6)> self()

#PID<0.84.0>

iex(7)> send self(), "HELLO"

"HELLO"

iex(8)> send self(), "HELLO"

"HELLO"

iex(9)> send self(), "HELLO"

"HELLO"

iex(10)> send self(), "HELLO"

"HELLO"

它往self进程发送了4条消息，现在让我们打开Observer的Processes，它显示了进程的消息个数。

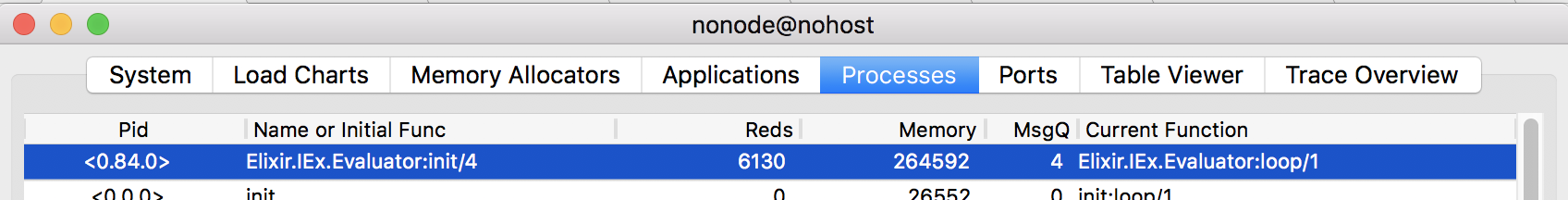


图3: Observer Processes

下图是这个进程的详细信息：

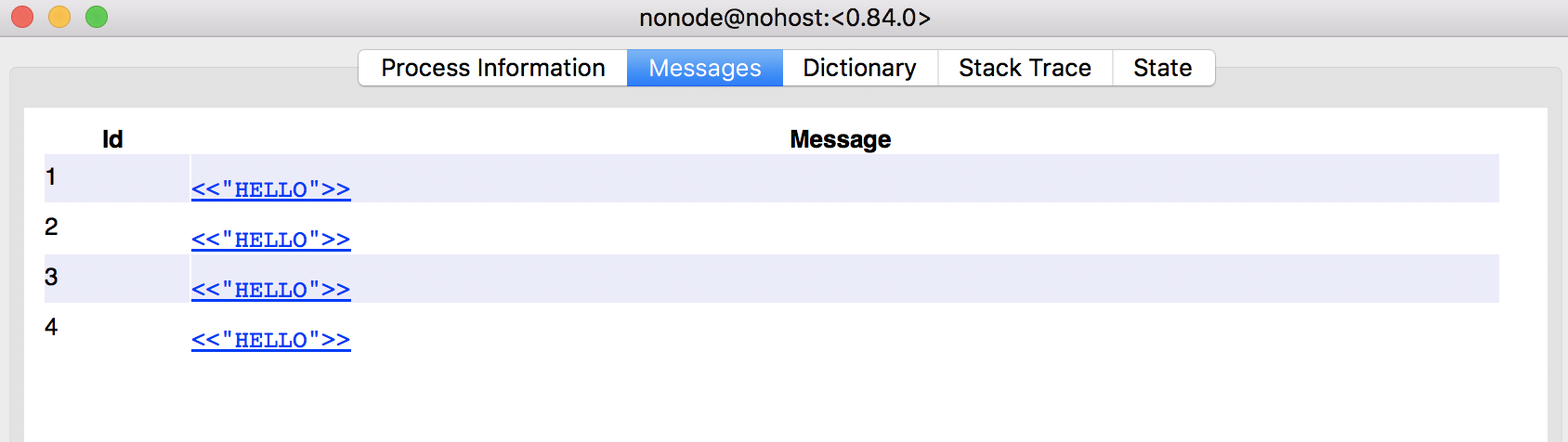


图4: Messages Queue

我们可以看到他的消息队列，现在有4条消息在消息队列中，如何得到这些消息呢？flush可以获取这些消息并打印到console中。有了这个概念以后相信大家对Elixir多进程中消息发送和接收的机制多了一份了解。在上面的例子中我们的Sender实际上就是命令接口，接收方是一个Elixir函数，那么如果我们把发送方也写成一个函数会是什么样子呢？我相信通过本文的解释作为开发人员的你应该能够自己写出Sender的代码，这里我就不占用篇幅来罗列这些代码了。

# Elixir生态环境

对于每一项技术、框架、语言来说，都会有一套与之相关的生态环境，例如：与Hadoop相关的大数据构架可以运行在Hortonworks云平台，同时Hive、Pig又支持以SQL或者脚本语言的方式检索HDFS上的存储，可以说整个从平台到语言到框架，Hadoop构造了一个属于它自己的生态环境。正式由于存在了这么多生态相关的产品，该技术才能得到更多的支持从而可以应用到不同的场景中。同样的道理对于Elixir也有自己的生态，下面我就给大家介绍一些与Elixir相关的技术和框架。

* Mix  
  Mix是Elixir应用的构建工具，Java里的Maven和Gradle，js里的webpack。它不仅仅可以用作编译项目源代码，而且还可以用来创建工程、启动项目、publish你的项目到Hex上。
* Hex  
  Hex是Elixir应用的包管理器，相当于NodeJs中的npm，Ruby里的Gems。
* Phoenix  
  这是一款基于Elixir的Web容器，它可以把Elixir作为Web后台的Restful服务。相当于RubyOnRails里面的Rails。
* Plug  
  与Phoenix类似，Plug是一款轻量级的Elixir Web容器。他没有额外的依赖是完全基于Elixir语言开发的。
* Erlang  
  前面已经提到，Elixir是构建在Erlang之上的语言，所有支持Erlang的框架都可以应用到Elixr上来。

正是由于具备了完善的生态，使得Elixir成为产品技术选型上的一个主要候选成员。

# 小结

函数式编程和并行计算都是很流行的概念，Elixir很巧妙的把两者结合起来，学习一门新的语言通常是阻碍开发者进入的障碍，幸运的是Elixir很想Ruby并且借用了Erlang VM的运行环境，这使得一些开发者觉得相对亲切和熟悉。多进程开发总是一件令人不愉快的事情，很多时候大家都避免多进程之间的场景，Elixir使得进程的维护非常容易简单，Elixir创建的进程非常轻量，轻量到相当于一个子线程那样简单，并且可以利用CPU多核资源。并且Elixir是开源免费的语言，支持它的框架和构架工具也同样开源，所以大家可以放心使用。

# 作者简介

赵翼，毕业于北京理工大学，目前就职于SouthbankSoftware，从事NoSQL，MongoDB方面的开发工作。曾在GE，ThoughtWorks，元气兔，Big Crunch担任项目开发，技术总监等职位，接触过的项目种类繁多，有Web，Mobile，医疗器械，社交网络，大数据存储等。