

On Top of Erlang VM

KrzyCube Yu

2008-12-08

krzycube@gmail.com

内容

- 基于Erlang VM的语言
- 与Erlang VM结合的层面
 - Core Erlang简介
- Reia的长处
- Single & Multiple Assignment(Reia 的MA话题)
 - Static Single Assignment(SSA)
 - Single Assignment有多重要

这里讨论的内容，涉及到基于Erlang VM实现的语言的简单介绍。

开发这样的语言，有什么用，说好听点，有什么意义。

语言在实现上与Erlang的结合。

然后，我觉得最想说的，是由Reia引出的关于单次赋值和多次赋值的话题。

不同的方式

- 以往与Erlang交互的方式
 - port
 - c node
 - jinterface
- 基于Erlang VM实现的语言
 - Reia
 - LFE
 - ErlyJS
 -

不同语言之间的交互：

1.translate

2.基于A的B语言解释器（或编译器）

3.外部接口

4.RPC

Reia是介绍的大头，其次是LFE，会提一下ErlyJS。



说实话，我头一次看到在其它文档都还不齐全的时候，直接在文档里的正式给出了关于标识的说明。

在Reia的作者看来，让语言的标识明确，对于搜寻与它相关的信息，比如文档，是很有现实意义的。

而为了少一点炫耀，多追求一点极简主义，Reia的标识非常简单。

Reia的动机

- 利用Erlang VM现有的优势
 - Concurrent
 - Fault tolerant
 - bla,bla,bla.....
- 并增强
 - 字符串处理
 - 正则表达式
 - 与外部工具库的连接
 - 以及通常在Erlang之外考虑的东西
 - 例如OO
- 填补转换到并发编程的跳跃空间

跳跃空间：

From imperative to declarative.

From procedural to functional.

From synchronous to asynchronous.

Reia Sample Code

```
>> module Foo
.. def plustwo(n)
..   n + 2
.. def plusthree(n)
..   plustwo(n) + 1
..
=> ~ok
>> Foo.plusthree(3)
=> 6
```

```
>> 1..10
=> [1,2,3,4,5,6,7,8,9,10]
>>
>>
>> [n | n in 1..10, n % 2 == 0]
=> [2,4,6,8,10]
>>
>>
>> [n * 5 | n in 1..10, n % 2 == 0]
=> [10,20,30,40,50]
```


明显，Reia拥有彪悍的List Comprehension能力，Ruby没有这个能力，Python和Erlang有。

Ruby和Python没有Pattern Matching及receive语句，Erlang有。

Ruby有first class 的正则表达式，Python和Erlang没有。

Lisp Flavoured Erlang

- LFE是一个（特殊的）基于Erlang VM的Lisp
- 不是Scheme，也不是Common Lisp
 - 最初版本的LFE是lisp-1 like
 - 后来版本的LFE是lisp-2 like
- 也不能实现完整的Scheme或Common Lisp的功能
 - 全局数据 & 破坏性操作，



```
(define (foo x y) ...)
(define (bar x y)
  (let ((foo (lambda (a)
    ...)))
    (foo x y)
    ...))
```

lisp-1 是指数据变量的名字空间和函数的名字空间是不分离的，lisp-2则是分离的。

scheme是lisp-1 like. lisp-2 是指变量和函数在不同的名字空间里，common lisp是lisp-2 like.

ETOS，一个Erlang到Scheme的编译器。早在1997年，应该是第一个做Lisp和Erlang交互的项目

LFE，反过来，将Lisp的理念引入到Erlang中

LFE的动机

- Something that would have the benefit of both worlds
 - Lisp World & Erlang World
- And Fun

Common Lisp

成熟、良好的文档、理解透彻、实用

然而：

跟不上新的系统设计

对并行的支持不好

没有丰富的内建数据结构

基础架构发展乏力

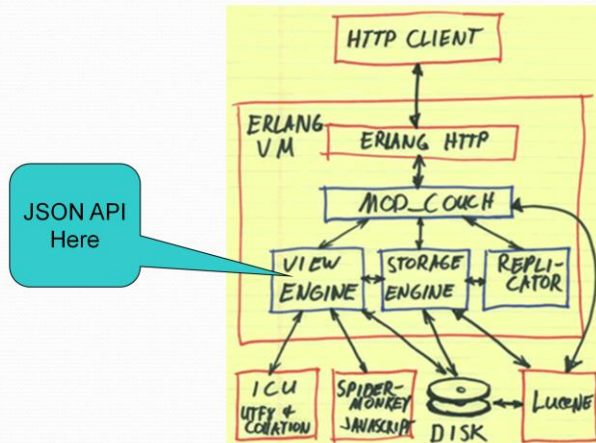
ErlyJS

- 纯Erlang实现的JavaScript Compiler
- 面向Erlang VM

ErlyJS的动机

- 高可扩展且快如闪电的服务端JavaScript
- 简化Ajax及Comet Web App的开发
 - 客户端和服务端都好用的语言
 - mod_js(server side javascript under apache)
 - <http://jerakeen.org/blog/2007/07/server-side-javascript-under-apache/>
 - 以及，从CouchDB的角度
 - CouchDB的JSON API

CouchDB Sketch



语法层面的结合

- Reia:

```
>> module ShowDate
.. def show
..   erlang::date()
..
=> ~ok
>> ShowDate.show()
=> (2008,12,8)
```

- LFE:

```
> (call 'erlang' date)
#(2008 12 8)
>
```

```
> (date)
#(2008 12 8)
>
```

- ErlyJS:

- 没有代码可看

反向的调用(e2r/arity)，还没那么好玩

LFE对erlang data types的支持

{ok,X} -> (tuple 'ok x)

error -> 'error

{yes,[X|Xs]} -> (tuple 'yes (x . xs))

<<34,F/float>> -> (binary 34 (f float))

[P|Ps]=All -> (= (p . ps) all)

实现层面

- Reia
 - Reia Forms -> Erlang Forms -> {Erlang Compiler handle this, finally BEAM}
- LFE
 - Lisp Forms -> Core Erlang -> BEAM
- ErlyJS
 - 正在做从Reia方法到LFE方法的改造

LFE的作者Robert Virding是Erlang的最初设计者之一，自然了解Core Erlang。
LFE其实是展示了在Core Erlang层面实现语言途经

- 面向Erlang Forms可以
 - 有现成的工具用于处理Forms的转换
 - 避免深入到BEAM内部机制
 - Core Erlang貌似是个挺晦涩的东西
- 面向Core Erlang的方式
 - 能让代码更快
 - Core Erlang的Forms需要先编译，再Load执行
 - 但与上面那个更快不冲突

Core Erlang

- 是Erlang的一个中间表示
 - 基于Erlang Forms和中间码（例如JAM，BEAM）之间的表示层
- 是一个Erlang的子集
 - 精密
- 高度抽象
 - 函数式（高阶函数）
 - 动态类型
 - 模式匹配
 - 并发

Core是很早就设计出来的东西（1999开始设计，2000年有第一个版本，现在是1.0.3）

Erlang的演化过程中，语法变得越来越复杂。使得开发处理Erlang代码层面的工具变得日益困难。（Profilers、debuggers）

JAM 和BEAM实际上都是Imperative的，很多在Functional层面的优化工作，到了这里就做不了。

Core Erlang的设计要求

- Core Erlang必须是规整的
 - 简化Code-Walking 工具的开发简洁的语义
- 在不同的Erlang实现中，都可以实现与不同的中间码格式的直接转换
 - JAM,WAM, BEAM
- 语法清晰
- 高度可读性

相对于Erlang来说，Core Erlang是一个更加严格，精确，稳定的形式，作为Compile Target更有好处

Core Erlang Sample

- Erlang:

```
MyCons = fun cons/2,  
MyCons(A,B)
```

- Core Erlang:

```
let MyCons = 'cons' /2 in apply MyCons(A, B)
```

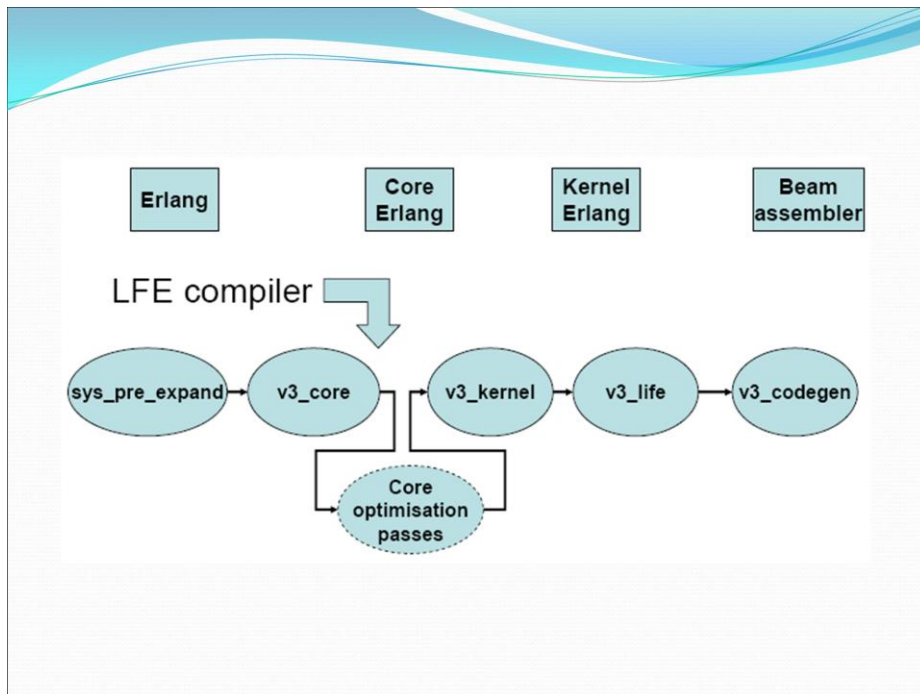
Core Erlang Sample cont

```
f(x) ->  
  case X of  
    {foo, A} -> B = g(A);  
    {bar, A} -> B = h(A);  
  end,  
  {A, B}.
```



```
'f'/1 = fun(X) ->  
  let <X1, X2> =  
    case X of  
      {foo, A} when 'true' ->  
        let B = apply 'g'/1(A)  
        in <A, B>  
      {bar, A} when 'true' ->  
        let B = apply 'h'/1(A)  
        in <A, B>  
    end,  
  in <X1, X2>
```

事实上是，Erlang的各种特性还是使得Core Erlang有些诡异了



`sys_pre_expand`

- Expand records, packages, annotate funs

- `v3_core`

- List comprehensions, add lexical scoping, return exported variables, sequentialise code, expand =, add explicit fail clauses

- `v3_kernel`

- Compile pattern matching, lambda lift local functions and funs, flatten nested calls

回到Reia

- Reia的对象
- Reia的长处
- Reia的Multiple Assignment

Reia的对象

- 对象之间只能通过Message传递
 - 基于Erlang VM的异步消息传递
 - Object相当与Erlang中的独立进程
 - Objects对应于gen_servers
 - transactional state holders which respond to OO-like method invocation.
 - So, Objects是事务化的
 - 如果对一个Object的调用（发消息）半途崩了，没啥影响

Reia Method

```
class Foo  
  def ->ShowMsg(msg)  
    puts(msg)
```



```
foo = Foo.new  
foo<-ShowMsg("Hello Reia")
```


靠，又多搞一个语言

- 为什么要添加一个扩展语言
 - 利用既有平台的优势
 - 通过简单的途经扩充既有平台的能力
 - 引入不同的范式
- 语法？No！
 - Erlang的语法设计是为了支持Erlang的语义和特性
 - 而且，Erlang不是一个OO的语言

- 1.多搞一个语言是为了什么
- 2.这个新来的语言能做什么
- 3.这个新来的语言能做到什么程度

Reia的长处

- A. Erlang测试
 - 并发程序的测试非常痛苦
 - 现有的Erlang测试方案
 - EUnit, ErUnit, ErlUnit
 - 然后有个笑话：
 - Ericsson公司的主要Erlang工程中，平均每3行代码都跟着2行test case
 - 哟，我现在想改几行代码。
- Reia能做的
 - 动态语言的测试方式
 - Mocking
 - stubbing
 - 声明式的测试框架
 - Ruby's RSpec
 - 与Erlang的无缝结合

- 
- B. 云计算
 - 快速开发一个 “distributed, fault-tolerant” 的程序
 - 云要求的特性
 - 例如CouchDB中的JSON
 - C. WebApp和Server端都好用的语言
 - D. 来自Erlang VM的优势

Reia允许Multiple Assignment

- Oh, My! Gag me with a spoon!
- Erlang已经展示了Single Assignment有多么重要，多么强大，干嘛又基于Erlang搞了个Multiple Assignment出来
- $X = X + 1$ 就是个错误

Reia的Multiple Assignment缘由

- 基于历史原因它更好使
- Reia的多次赋值其实是Fake的
 - Reia Forms → Erlang Forms

↑
Static Single Assignment Transform

Static Single Assignment

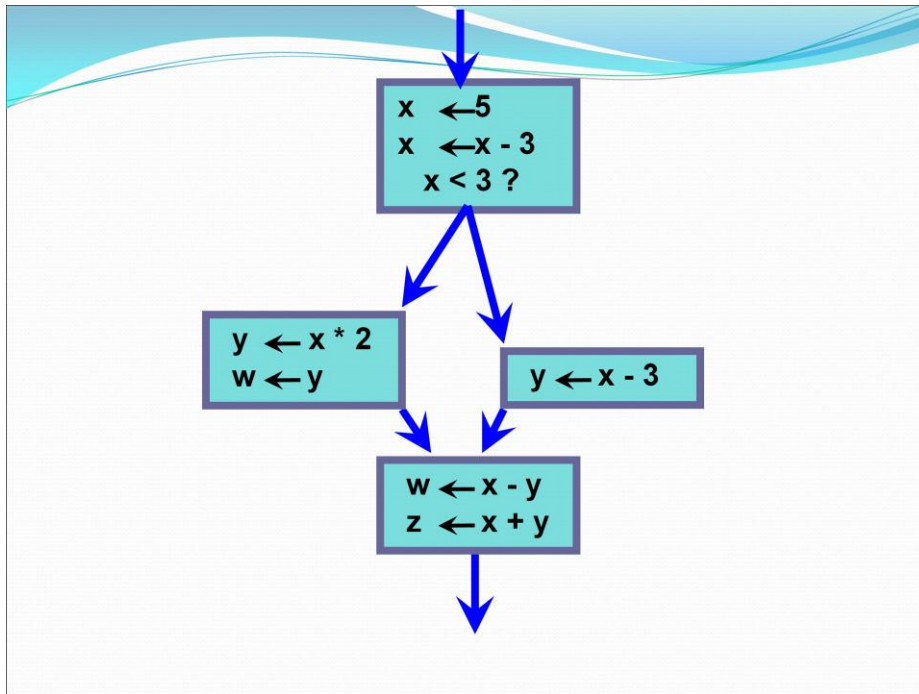
- 编译器设计中的一种变量的中间表示形式
 - 用于做编译优化
- 将原有的变量 Var 的多次出现切分为以 $\text{Var}_1, \text{Var}_2, \dots, \text{Var}_n$ 形式表现的独立变量
- 即每个变量只能被赋值一次

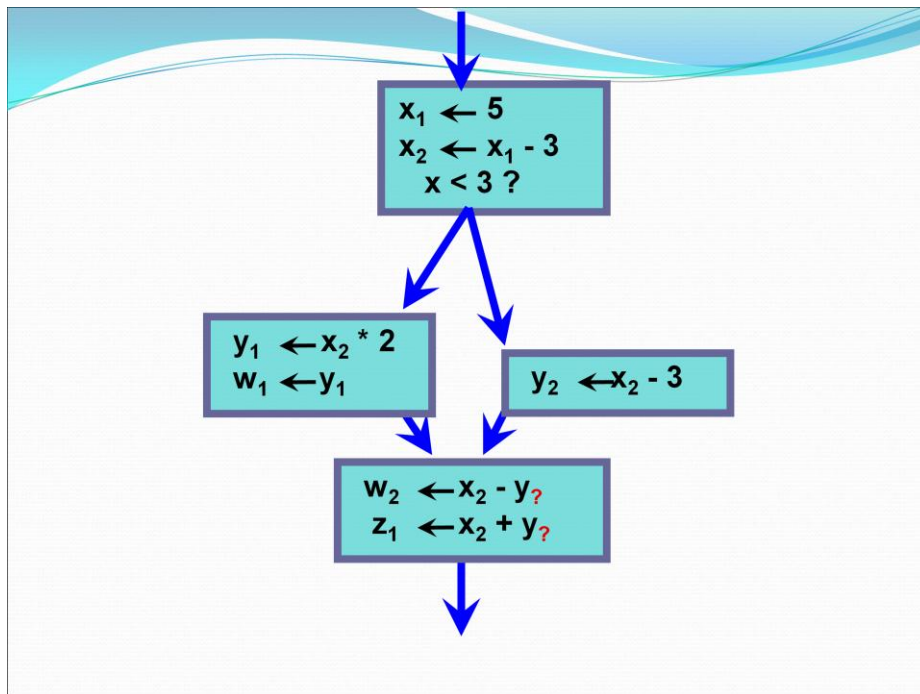
```
x = 1;  
x = 2;  
y = x;
```

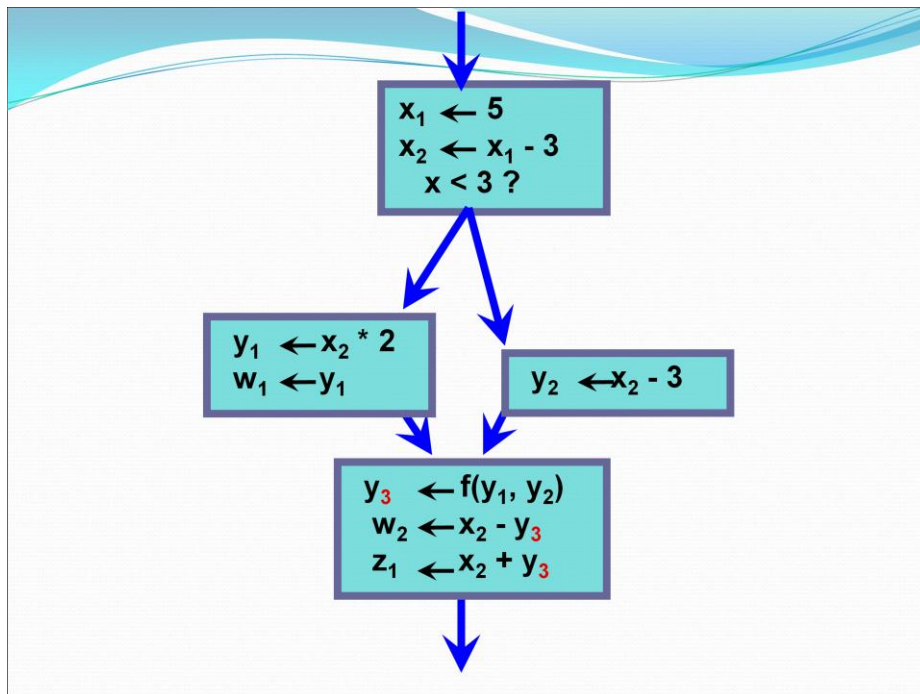


```
x1 = 1;  
x2 = 2;  
y1 = x2;
```

可达性判断







Single Assignment

- 真的有那么重要吗
- 它是Erlang的核心基石吗?
 - 当然
 - Erlang的并发
 - Erlang的容错
 - bla, bla.....

关于State

- Erlang有状态
 - mutability才是那个关键的词
- State是关于绑定吗？
 - 不是，State是关于Data
 - 绑定本身不属于State的一部分
- 绑定关系变了
 - 并不会直接影响Data

Single Assignment

- 是关于可以如何修改函数的绑定关系的约束
- 一个绑定是变量和它所代表的值的映射
 - 并且绑定是受函数的作用域限制的
 - (1 名字) -> (2 存储位置) -> (3 值)
- 在SA的情况下，绑定关系仍然是可变的
 - Unbound -> Bounded
- MA只是以不同的方式改变绑定

所以在Erlang中，变量它还是叫变量

Single assignment 是Erlang 高容错性的基础？

- 不是吧！
- Single Assignment干嘛的呢
 - 规避了一种特定的人为错误（Programmer Error）
 - 即，经常因为重新绑定而，无意识地覆盖了变量的值，其实原来的Data没变
- Erlang容错的关键是：
 - "fail early, fail often"策略
 - linked processes
 - supervision trees,
 - distribution and failover
- 其实呢
 - 有了SA，还是会有同样的错误的，比如返回了错误版本的变量

这些能力并不是全都烙印在OTP 里头的，相反，OTP 通过提供这些组件，让你可以将

这些能力添加到你的应用中。它们之中，没有哪个是依附于Single assignment 实现的。即便

是事务语义（Transactional semantics）这个Joe Armstang 在他的书里大肆褒奖的东西，也并

不是必须依赖于Single assignment 的，很显然Multiple assignment 也可以轻松实现它。

Single assignment 是Erlang 出色的 并发能力的基础？

- Erlang并发的关键因素
 - 是不可改变的State而不是绑定
- Erlang中的并发单元是Process
- 一个Process是一个Function
- Single Assignment发生在Function中

然后

- 所有的Erlang进程通过Message通信
- Message发送的内容是数据的拷贝

所以，Erlang的基石是

- 1. Actor Model
 - 基于异步消息传递的分布式处理模型
 - 独立的并发进程
 - Message ID
 - 独立的执行线程
 - 消息队列
 - 基本原语
 - send
 - new actor
 - ready
- 2. Linked Porcess
- 3. Supervision trees
- 4. "fail-fast,....." policy

Then Why Single Assignment

- 更清晰的语义
 - 当某个变量出错了，知道最早定义的地方就行了
- 对可靠性的极致追求
- Hot Swap

结论

- Reia看起来不错
 - 易于接受的语法
 - 拥有Erlang（OTP）的能力
 - 扩展了Erlang（OTP）的能力
 - 引入了多范式
- **Practical**

结论 cont

- 不是要找出一个Universal的关于变量，值，绑定的分类系统。这其实没有多大的意义。
 - “变量”其实是“跟某个语言相关的”这个层面的概念，否则它就是未定义的

结论 cont

- Erlang VM提供了很好的在其上扩展一个语言的能力
 - yecc & leex
 - Erlang的Yacc 和 Lex
 - core erlang

结论 cont

- 不要误解了Single Assignment的作用
- Erlang（OTP）的许多优秀的设计决策一起构筑了这个优秀的平台
 - 这些东西还不是僵化的
 - 开发者可以任意取它们中的一个或几个用于自己的APP



Thank you !