

# 我为什么选择了Erlang?

许式伟

2007-10-10

# 面向的受众

- 我们不解释**Erlang**基础文法。假设读者已经对**Erlang**进行过了解。如果你期望入门资料，那么它并不适合你。
- 本文谨献给那么已经初步了解**Erlang**，仍然赶到迷茫与困惑的人。

# Erlang推荐资料

- Programming Erlang
- Course.pdf
- Joes thesis 2003
  - 面对软件错误构建可靠的分布式系统(段先德译)

# 纲要

- Erlang为什么广受关注？
  - 业界趋势
  - Erlang的问题域
- Erlang的哲学
- Erlang的困惑
- Erlang流行的困难之处
  - 如何应对
- Erlang的不足（非设计缺陷）

# Erlang为什么广受关注？

- 2007 has been a good year for Erlang
  - Amazon
    - Joe's book (Programming Erlang) is ranked #86 on the amazon.com bestseller list (category: computers & internet)
  - Google Trends
    - <http://www.google.com/trends?q=smalltalk%2C+erlang>
    - <http://www.google.com/trends?q=erlang%2C+ocaml>

# 并行、分布式的趋势

- 多核（并行）
  - 让我们假设**Intel**是正确的：让我们假设 **Keifer** 项目会获得成功。如果是这样，那么 **32**个核心的处理器在**2009/2010**年就将会出现在市场上。
  - 这毫不奇怪，**Sun**已经制造出了**Niagara**，它拥有**8**个核，每个核运行**4**个超线程（这相当于**32**个核）。
  - 《What's all this fuss about Erlang?》
    - by Joe Armstrong （Erlang创始人）

# 软件业发展趋势

- 总体趋势
  - 从卖工具软件转卖服务（软件免费）。
  - 从纯客户端到**B/S**或**C/S**。存储与计算向服务端转移。
  - 从**PC**单机到服务器+客户端多元化（手机、**PC**、**PDA**、电视机顶盒、车载终端等）。
- 结果
  - 客户可以更加方便地享受到服务。
  - 信息越来越膨胀（高速增长）。
  - 服务器承受越来越高的负荷（用户数+数据量）。
  - 竞争更加激烈、客户更加挑剔。
  - 服务质量差异显著化。

# Erlang: 并行/分布

- Erlang: 并行/分布式语言。
  - 从根本上解放了程序员的生产力，让程序员可以全力关注于自身的逻辑，而不需要被太多的机器硬件所困扰。这就像GC语言把程序员从内存管理中解放出来一样，现在Erlang解放的是程序员在并行处理上的困扰，而这比内存管理的复杂度更大，也更为迫切。

# Erlang的问题域

- 高并发：应对超大量的并发活动。
- 分布式：系统可以跨计算机分布运行。
- 持续服务：系统应该能不间断运行许多年。
- 热升级：软件维护/升级应该能在不停止系统的情况下进行。
- 可靠：满足苛刻的质量和可靠性需求。包括提供容错功能，在硬件失灵和软件错误时继续提供正常的服务。

# Erlang: 专注

- Erlang Focus的问题域恰恰是现代服务器的普适需求
  - 高负荷、高可靠、持续服务、热升级。
- Erlang是一门专注的语言
  - 有自己Focus的问题域。
  - 不是用来解决所有问题的语言。
  - Erlang是轻量级的语言。

# Erlang的哲学

- 面向并发编程（COP）
  - 进程隔离（isolation）
  - 消息传递
- 错误处理哲学
  - 速错（fail-fast）

# 进程隔离（Isolation）

- 无共享
  - 进程具有“不共享任何资源”的语意。
  - 进程被认为是运行在物理上独立的计算机上。
  - 分布式计算所需的任何数据都必须通过拷贝。
- 消息传递
  - 消息传递是进程间传递数据的唯一方式。
  - 消息传递必须是异步的。如果采用同步方式，那么当消息的接收者偶然发生一个软件错误时，就会永久阻塞住消息的发送者，破坏了隔离的特性。
  - 消息传递被认为是不可靠的。要知道消息是否被正确送达的唯一方法就是发送一个确认消息回来。
- 位置透明
  - 你不知道你的联系人（进程）位于什么地方。只要有它的标识（PID）你就可以与之通讯。

# 消息传递

- 消息传递当是原子化的（**atomic**），意思是一个消息要么整个儿被传递，要么根本就不传递。
- 一对进程之间的消息传递是有序的，意思是当在任何一对进程之间进行消息序列的收发时，消息被接收的顺序与对方发送的顺序相同。
- 消息不能包含指向进程中的数据结构的指针——它们只能够包含常量和（或）**Pid**。

# 错误处理哲学

- 故障即停（Halt on failure）
  - 速错（fail-fast）。
  - 不成功，便成仁。
    - If you can't do what you want to do, die.
  - 任它崩溃。杜绝防御式编程。
- 让其它进程来监视并修复错误
  - 无论软件故障、硬件故障，均可监视&修复。
- 持久存储
  - 存储器应当分为持久存储器（stable storage，进程崩掉时依然存在，用于故障恢复）和临时存储器（volatile storage，崩掉就没了）。
  - 由“事务机制”（transaction mechanism）来提供数据和消息的完整性。

# Erlang困惑

- Erlang为什么没有数组？
  - 为什么我不能以 $O(1)$ 的速度访问列表成员？
  - 为什么不能进行二分查找（binary search）？
- 为什么Erlang不是OOP语言？
  - 为什么没有类？
  - 为什么没有虚拟（多态）？

# 我的解释

- Erlang为什么没有数组？
  - Erlang的Binary是一个无数据类型的数组（可以认为是C/C++中的void\*）。
    - 例：Buffer从Offset开始取得N字节，赋给Fetch。  
`<<_:Offset/binary, Fetch:N/binary, _/binary>> = Buffer.`
  - 由于数组的修改代价很大，因此Erlang中的数组意义不大（相当于传统语言）。但不可修改的常数组有存在的意义。
    - 例：读取文件内容到FileBuf中，从而在内存中进行随机访问。  
`{ ok, FileBuf } = file:open(FilePath, read).`

# 我的解释

- 为什么 **Erlang** 不是 **OOP** 语言？
  - **OOP** 的核心——类在于封装变化；**Erlang** 的变量是一次赋值，不可变化。故 **Erlang** 与 **OOP** 是天生冲突的。
  - 对比
    - **Java**

```
stack stk = new stack();
stk.push(10);
val = stk.top();
```
    - **Erlang**

```
Stk1 = stack:new(),
Stk2 = stack:push(10, Stk1),
Val = stack:top(Stk2).
```

# Erlang的困难之处

- 对软件开发人员来说，养成面对问题就用多线程的方式（并行）来进行思考本身就是一个比较大的跨越。
  - 养成多线程的代码编写习惯是未来程序员的基本技能之一。
  - 从我的角度来看，PC平台上的电脑迟早是要淘汰的。未来的终端可能是PDA或者是其它便携的设备，因为将来的用户并不需要一个这样很大的机器，而是由一个大型服务器作为数据和计算的中心来完成计算和存储等工作。
  - 要么就不写程序，要么就写服务器端的程序，当然，你也可以去撰写移动终端设备上的代码，还是上面的观点，在PC平台上做开发的空間很小。我个人还是非常鼓励开发人员做服务器端程序的开发，同时也希望它们在撰写代码的时候能够养成多线程的习惯来并行处理他们的应用。
  - 《程序员》杂志2006年第9期，Sun公司的工程技术总监柯泰博士

# Erlang的困难之处

- 尽管Erlang程序可以很方便地并行/分布式，但是这并不意味着并行思维模式不需要被理解和掌握。

# Erlang is FP language

	函数式	命令式
语言	Lisp, Erlang, Haskell	C/C++, Java
理论基础	Lambda演算	图灵机
特点	程序为函数求值； 函数可作为另一个函数的参数以及返回值。	程序为一条条指令；函数执行具有副作用。

# Erlang的困难之处

- 缺乏深入人心的FP编程理论
  - FP语言确实有其很大不同之处。其理论支撑是我们所不熟悉的。
  - 没有FP“数据结构”学。大学的数据结构课程，都是基于命令式语言的。

# Erlang的困难之处

- 需要更高的培训成本
  - 由于FP没有深入人心，Erlang亦仍然属于小众语言，故此培训成本较高。

# 应对：信念

- Erlang, the next Java?
  - 我坚信这一点。
  - Erlang精简、健壮、容易掌握（相对于其他FP语言而言，它的难处是FP共同的难处）。

# 应对：普及FP理论

- **FP**只是有点不太一样，但实际上更它简洁、更容易理解。

# Erlang的不足（非设计缺陷）

- 性能仍然有一点点不足
  - 指的是“单进程内执行指令”的速度。
  - 可持续改善
- 代码管理的工程强度不足
  - **Java**的成功在于它的代码管理足够简单、强大、优雅。命名空间、模块、包等等。
  - **Erlang**的一级命名空间显得单薄。

Q & A

谢谢！