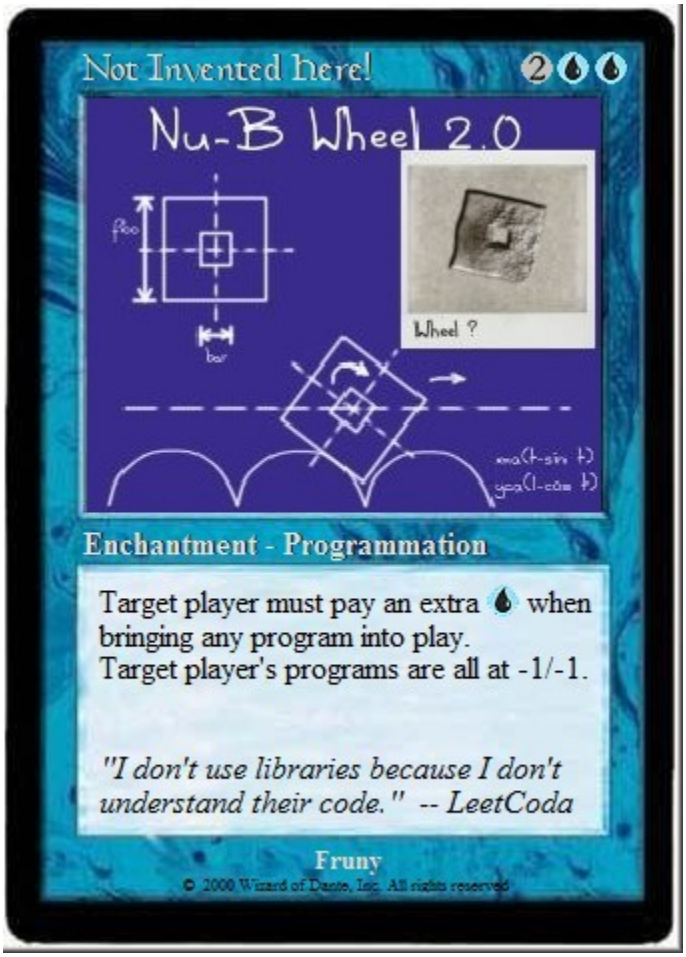
“你为什么喜欢 Erlang？”

作者：Tony Arcieri(tony@clickcaster.com)
原文地址：http://www.clickcaster.com/items/why-do-you-like-erlang



自从大约两个月前我变成了一个 Erlang 铁杆以来，人们经常问我这个问题。我认为最好的解释方式就是把它作为 Greenspun 第十定律的一个推论：

> *任何足够复杂的并发的 C 程序都内含了一个临时拼凑起来、没有正式规范、充满错误并且低效的半吊子 Erlang 实现。*

我一直着迷于那些要处理大量并发连接的网络服务器的开发。多年以来，我尝试着用 C 来写，考察了事件监测的很多不同方法，也研究了诸如《Unix 网络编程》这些书中所描述的不同的实现网络服务器的方式。遗憾的是，我发现这些模型和方法中的不少都已经过时了。自从 glibc 支持了 Linux，线程模型就变成了事实上的标准。

几年后，我决心用每个请求一个线程的模型来实现一个大规模并发的网络服务器。此前，我已经建立起一个属于我个人的服务器框架，伴随着该框架有很多操作系统的基础设施的抽象接口，并且我已经写了一个有超过 2 万行代码的网络服务器。

正是在这个时候，我开始在线程方面碰壁。本来，我已经对 Win32 和 pthreads 之间进行了跨平台的线程模型抽象，正在很多平台上测试。我开始注意到，在某些平台上我会遇到死锁，但是其余的则不会。随着数据结构高级程度的增加，我开始发现更多的死锁。当那些重要的共享数据需要在所有线程之间进行同步的时候，锁变得更复杂，更难调试，程序性能也开始下降。

我开始更加了解互斥量的开销，这种开销对于一个关心性能的 C 程序员来说是非常令人抓狂的。例如，在 Win32 平台上锁住一个互斥量，需要使用超过 600 条的 CPU 指令！我很快意识到最好的的方法确实是一个 CPU 核上只跑一或两个线程，并且，能避免线程同步问题的更轻量级的并发机制是处理大量并发连接的更好方法。

首先映入我眼帘的是《计算机程序设计艺术》中描述的 Knuth 协程，它们提供了一种轻量级的协作调度方法，不需要系统调用即可在它们之间进行切换。GNU Pth 库的实现和其非常类似。我找到了很多相似的用户空间线程的方法，其中包括像被 Win32 纤程和*IX 的 ucontexts 实现的“微线程”思想。

所有这些方法的最终目标都是从把行为和功能分离开。每个原生线程（假如你有超过 1 个）都把它的行为隔离进一个内嵌的事件循环，功能则可以被实现为一个微线程。

假设已经到了这一步，当你发现你需要一种在微线程之间传递消息的途径的时候（该途径要在某种意义上抽离出网络 I/O 事件的处理，并且如果你使用了线程，那么也允许你在它们之间发送消息），问题便接踵而至。我就这个问题实验了很久，弄得在多事件源和线程模型之间的抽象相当的复杂。

结果是，为了建立一个并发网络编程框架，我付出了多年的努力，却从来没有达到一个让我满意的程度。我从来就没有感觉到我已经真正把我要去解决的那些问题抽离出来了。

终于，我认识到不共享任何东西才是可行之道。我开始对 Dan Bernstein 的 qmail 迷恋起来。Qmail 使用了很多不共享任何东西的操作系统进程，它们通过管道进行交互以处理网络通信。厌倦了线程，厌倦了微线程，也厌倦了企图去同步共享的信息，这就是我得到的最终结果。Unix 哲学的智慧“做一件事情并且把它做好”最终在我的眼里彰显了它自己。

最后，我发现了 Erlang。随着我对它了解的越多，我越认识到它正是那个我期待了很久的模型的一个绝对卓越的实现（我自己曾经企图去实现过）。此外，它还把每 CPU 内核上多个原生线程轮流运行多个“微线程”的模型与一个消息通信系统整合在了一起，远比我自己当初的整合要优雅得多。

对于那些还不了解的读者，这是从最近出版的《Programming Erlang》一书中摘录出来的关于于 Erlang 消息处理的一个描述：

1. *当进入一条 **receive** 语句的时候，我们启动一个计时器（但是仅当表达式中存在一个 **after** 段的时候）*

2. *取出收件箱里面的第一条消息并尝试和模式 **1**，模式 **2** 等等进行匹配。如果匹配成功，则把该消息从收件箱中删除，这一模式之后的表达式将被求值。*

3. *如果 **receive** 语句中没有任何消息匹配，那么第一条消息被从收件箱中删除并放于一个“保存队列”中。然后，尝试收件箱中第二条消息。这一步骤被重复直到找到一个匹配的消息或者收件箱中的所有消息都被检查完毕为止。*

4. *如果收件箱中没有消息被匹配，那么进程被挂起并且会在下次一条新的消息放置于收件箱中后被重新调度执行。注意，当一条新的消息到达的时候，保存队列中的消息不会被重新匹配，只有新的消息才会被匹配。*

5. *一旦一条消息已经被匹配，那么所有已经被放进保存队列的消息将按照他们抵达进程的顺序重新进入收件箱中。如果设置了计时器，那么该计时器会被清掉。*

6. *如果在我们等待消息的时候计时器已经耗完，则计算 ExpressionsTimeout 的那些表达式，并且把保存的信息按照它们抵达进程的顺序重新放回收件箱。*

读完这个以后，我认识到这就是我一直在尝试发明的东西。我把它给 Zed Shaw 看了（他可能以 Mongrel web 服务器的创建者被你所知）。他表示这和他在 Myriad 框架中实现的消息处理方式非常类似。我不打算为 Zed 就 Myriad 是一个 Erlang 的重新实现这一点去辩护什么，我只是要指出当人们处理此类问题足够久之后，人们得出解决这类问题的结论是相同的。

总之，我对 Erlang 处理事情的方式感到自然的舒服。一直以来，我就是这样写程序的。终于找到了一门抽离了所有处理并发网络复杂行为的语言，这种感觉非常爽。更进一步，就一旦所有底层问题都被抽离，你能构建出什么这方面而言，分布式 Erlang 和 OTP 框架更是超过了我最疯狂的梦想。

现在我终于可以不用试图构建一个临时拼凑起来、没有正式规范、充满错误的框架了，我开始集中精力于我实际上一直所关心的：功能。