原文: https://medium.com/@adamhjk/rust-and-go-e18d511fbd95

翻译者： Scott Huang

翻译日期： August 22,2015 于厦门

Rust and Go

I've been spending a bit of my time playing around with new languages—in particular, Rust has captured my imagination. The bulk of the code we write at Chef is in Ruby, Erlang, and Javascript (lately Angular.) There are things I like about all those languages:

我已经花了不少时间在把玩一些新语言 – 特别要指出，Rust 抓住了我的想象力。我们在 Chef 中使用大量的 Ruby，Erlang 和 Javascript(后来是 Angular)代码。以下是我喜欢这些语言的事情：

- Ruby feels like it always hits the "whipuptitude" part of my brain. It's easy to simply sit down and start typing, with very little in the way. It also has the expressiveness that I always loved in Perl. The more you understand the language, the

more it feels like I can express myself in the same way I do with English.

- Ruby 的感觉就像永远在敲打我头脑中“whipuptitude”部分。它很容易，可以简单的坐下来开始编码，你基本不会碰到困难。它也有类似 Perl 的我很喜欢的表达力。

  你越熟练这个语言，你就会越觉得你好像可以用英语（母语）表达自己的想法。

- Erlang and OTP are glorious to operate. Things like pattern matching, actor concurrency, single assignment, and a lovely runtime make it a joy to run, manage, and debug production services. I think the syntax is awkward, but it too has a terse kind of beauty when you soak in it.
- Erlang 和 OTP 极好操作。那些模式匹配，Actor 并发，单一赋值，还有令人愉快的运行时导致执行、管理、调试产品服务是一种乐趣，我认为它语法是丑陋的，但是当你掌握它后，你会发现它的某种简洁美。
- Modern Javascript is becoming delightful in its own way. The ease with which you can grab community packages and frameworks, the sheer expressiveness of things like Angular, and the progressive slimming down of the often used parts of the language make the experience delightful again. It used to feel awful to me.

- 现代 Javascrip 我行我素的变得越来越讨人喜欢。你很容易就可以获取社区贡献的各种包和框架，Angular 的表现力非常好，还有不断进步的对语言常用部分的瘦身使得它再次获取了人们的欢心。过去我不喜欢它。

So—I decided to write a little Rust and, because everyone in my world seems swoony over it, Go.

所以 – 我绝对写一些 Rust，还有 Go，因为我周围的人看起来对它很着迷。

Rust

I started paying attention to Rust a couple of months ago. The language is designed to fill the same niche as C/C++—suitable for low level programming, with high safety guarantees, and a novel approach to memory management (borrowing). It also has a bunch of language features that I love from Erlang—pattern matching, actor style concurrency, immutable variables.

我几个月前开始注意到 Rust。这个语言被设计用来取代 C/C++，适合进行底层开发，带有高的安全性保障，和革新性的内存管理方式（borrowing 借用）。它还有我喜欢的和 Erlang 蛮类似的语言特性：模式匹配，Actor 样式并发，缺省不可变变量。

My history with C/C++ is limited—to be honest, my history with any strongly typed language is limited. As a systems administrator, there just was rarely a reason to do more than patch someone elses code (which I've done plenty of—but a patch isn't the same as writing something from scratch.) In the few experiences I've had, I found the compile->run->segfault loop to be deeply irritating. I recognize that this is my own ignorance on display: if I spent more time in the language, I would certainly come to understand the pitfalls that were so regularly striking me.

我使用 C/C++的历史并不多。诚实的讲，我的强类型语言的编程经验并不多。作为一个系统管理员，我们通常就是给某人的程序打补丁（我做了很多这种事，但打补丁不像从头开发一下东西）。在我拥有的有限经验中，我发现编译->运行->排错循环太令人恼火。我承认这是我个人的无知：如果我多花时间来理解语言，我当然可以发现那些经常打击我的陷阱，

So it was that I set out to re-write a command line utility from Ruby to Rust. I have no intention of shipping it, or even really sharing it—it was just an excuse to learn the language. Here is what I observed:

所以，我开始用 Rust 重写一个 Ruby 开发的命令行工具。我并没有想卖它，或者共享它，它仅仅是我学习语言的一个借口（动力）。以下是我观察到的：

Cargo is nice
Cargo货物（包）管理器非常好

Cargo as the front-end tool to Rust projects is coming along super nicely. Creating a new Rust project, adding in testing, dependencies, etc was easy and super clear.

Cargo 是负责 Rust 项目管理的一个非常好的前端工具。创建一个新的 Rust 项目，添加测试、依赖等等都非常的容易和简洁。

Expressive like a scripting language 像脚本语言一样富有表现力

Rust can be quite expressive in a way that feels like a low level scripting language. Check this out:

Rust 就像一种非常富有表达力的底层脚本语言。更多请看下面这个链接：

https://gist.github.com/adamhjk/e296257b32818ee8691d.js

While things like as_slice() and Vec<String> might take a little explaining, but it's remarkably straightforward. Here is the review function:

有些东西 as_slice()和 Vec<String>也许需要一点时间来解释，但它非常的直白。下面链接可以回顾这些函数：

https://gist.github.com/adamhjk/ead9de51d6f6d3aefeb2#file-review-rs

Again, I don't think this is any more difficult than it would have been in a scripting language. It takes a little getting used to the fact that, for example, I return an Option<Path> from have_dot_git(cwd), and use pattern matching to extract the value (or raise an error)—but once you realize whats going on under the hood, it's pretty great.

再说一次，我不认为它会比一种脚本语言更难。它需要一点时间来习惯那些事实，比如，我从 have_dot_git(cwd)返回一个选项

Option<Path>，并且用模式匹配来抽取值（或者抛出错误）-但当你认识到罩子下面发生什么事情后(理解后)，你就会觉得这很棒。

Strongly typed 强类型

In every other language that was strongly typed, I always felt like I was in a wrestling match with the compiler. I found it exceedingly difficult to trust that it was stopping me from doing something stupid, rather than getting in my way. (I recognize this is an emotional flaw of my own) Rust felt different for three reasons:

在任何一种强类型语言中，我总是觉得我和编译器在较劲。我发现很难信任编译器，与其说它可以阻止我做一些傻事，

还不如说它老是挡道。（我承认这是我自己的一种情绪缺陷）
Rust 从下面三方面让人觉得不同：

- Its error messages are exceedingly clear, pointing you directly to what is going wrong, and often showing you the precise solution. (“Did you mean…”)
- 它的编译错误信息非常的清楚，直接指出哪里错了，还经常给你提供精准的解决方案（“你是不是想…”）
- The compile loop was very fast, and when my code compiled successfully, it **always ran exactly like I wanted it to**.
- 编译循环非常快，当程序成功编译后，它总是按我的想法准确的运行。
- The idea of borrowing, and the idea of lifetimes, takes some getting used to. However, the compiler really is acting in your best interest all the time, and again the error messages and rules are clear. There is no traditional garbage collection in Rust—but it has very clear rules for how long things on the stack and heap will live. Those rules tie closely to scoping, in a way that feels really natural.
- 借用、生命周期等概念需要点时间来习惯。但是，编译器每次都吸引住你的兴趣，同时错误信息和规则是清晰的。Rust 没有传统

的垃圾收集器-但它有非常清晰的规则来指示在栈和堆上的东西该活多久。这些规则和区域结合的很紧，让你觉得很自然。

Most importantly, for me, my code never segfaulted or panicked. This was an experience that I've never had in my life when working with a low level systems programming language. If the compiler accepted my input, it ran—fast and correctly. Period.

对于我最重要的是，我的代码从来没有异常错误或者恐慌。这些都是我以前用底层系统编程语言中从来没有拥有过的体验。如果编译器

接受我的输入，它开始运行-快速并且准确。

Libraries don't really exist 有些库还不存在

Rust is still evolving, heading towards a 1.0 release. As a side effect, a lot of the higher level libraries just don't exist, or haven't been written. A good (trivial) example was a great command parsing library just doesn't exist yet. There is one included in the standard library, but the interface is clunky at best. A few promising libraries existed on Github, but the best one (from an interface/functionality point of view) failed to

compile on the latest version of Rust. (This was due to the fail!(..) macro being renamed to panic!(..)—not a big deal, but a sign of the kind of things that are still evolving.)

Rust 还在演进，快到 1.0 了（译者：现在 1.2 都出来了）。作为一个副作用，许多高阶库都还不存在，或者还没人写。一个好的（小）例子

是命令行语法分析器的库还不存在。在标准库里有一个，但是界面比较还不好。Github 中有一些有希望的库，

但是最好的那个（从接口、功能性等来看）没有办法用最新的 Rust 来编译通过。（这是由于 fail!(..)宏被改为 panic!(..)-不是一件大事，

但也是一件说明 Rust 还在演进的事实。）

Similarly, Cargo is great at fetching dependencies—but there isn't a collection of what is available, a-la NPM, Rubygems, or CPAN.

同样，Cargo 可以很好的获取依赖-但没有说明那些包可以获取，就像 NPM,Rubygems 或者 CPAN 那样。

Final Thoughts on Rust 关于 Rust 的最终想法

Rust is a powerful, deeply well thought out systems programming language with a pretty amazing and useful set of features. (Generics, Traits, Pattern Matching, Actor concurrency, Borrowing—the list goes on) There is a tax on learning the surface area and “Rust Way” of solving a problem—the language itself is simple, but has lots of different concepts to wrestle with.j

Rust 是强大的，高度深思熟虑的系统编程语言，带有一些很好的、很有用的特性。（泛型，特质，模式匹配，Actor 并发，借用-列表还很长）。那里需要一下税负来学习并用“Rust 方法”来解决问题-语言本身是简单的，但需要和一些不同的概念较劲。（译者：如果你了解函数式编程 Scala, Haskell，就会觉得学习 Rust 很快）

That said, its the first time in my life I feel like I could trust myself to write fast, efficient, low level systems code. The compiler was your buddy, and the errors it fed me were useful and clear. When the language hits 1.0, and the ecosystem of libraries starts to heat up—Rust is going to be a force to be reckoned with.

这么说吧，我第一次觉得我可以相信自己可以快速写出一个有效率的底层系统代码。编译器就是你的兄弟，它提供的错误提示非常

有用和清晰。当它演进到 1.0，并且库的生态系统开始升温时-Rust 估计会成为一种新势力。

Go

Lots of people have said lots of things about how amazing Go is. After spending even a few hours re-building something in the language, I see the appeal.

许多人说了许多关于 Go 有多好的好话。当我花了几个小时用 Go 重新编写一些东西后，我明白为什么这么说。

Go is small Go 比较小巧

You can learn the surface area of the language super quickly, assuming you have ever programmed in anything C derived. There are pretty much only the things you already know how to use in the language: if/then/else, switch, for. They use the syntax you already know. They provide short hand for things that you could type verbosely, but don't want to (:=, for example).

如果你有一些 C 衍生语言的编程经验，那么你可以很快的学会这门语言的一些表面领域。你简单的学会一下东西后，就可以开始使用

这门语言了：if/then/else,switch,for。它们的语法你已经知道了。它提供一些速记语法来简化一些繁琐的打字工作。（比如 :=）。

Go is opinionated—about lots of things
Go 是固执己见的 - 很多方面都是

When you read How to Write Go Code it starts with telling you how to lay out the top level source directory for *everything you will write in Go*. The Go tooling knows how to take that structure and do the right thing with it—compile binaries that go into your 'bin' directory, run your tests, grab dependencies.

当你读 "怎么编写 Go 代码"时，它开始告诉你怎样从顶层源代码目录开始的每件东西你该怎么用 Go 写。Go 的工具链

知道怎样取得结构和做正确的事情-编译二进制到你的'bin'目录，运行你的测试，捕获依赖。

The static output is nice—and the reality that, whatever you compiled locally will be what your users consume feels great.

静态输出是好的-现实是，任何你本地编译的都会是你用户消费的，感觉不错。

Go gets out of your way Go 避免你的方法

Where the Rust compiler made sure that what I expressed would result in working low level code, Go sits in the middle. The language is strongly typed, but in practice it seems to infer what you mean most of the time. For example:

Rust 编译器让你很确定你的想法会如何在底层代码中工作，Go 坐在中间。语言是强类型的，但实践是感觉推断出大部分你的想法。举例：

https://gist.github.com/adamhjk/5a475b8dd45971a4e814#file-dot_git_dir-go

The only type annotations in that function are the inputs and the outputs—everything else just works. It feels fast, easy, approachable and very low hassle.

唯一的类型注释功能是输入和输出-其他的每件事就是工作。感觉快速、容易、可以接近和非常少的烦恼。

Go failed at runtime Go 的运行时失败了

It was very, very hard to get Rust to fail at runtime. The compiler made sure I checked every code path, always made sure I covered every angle. Go was more than happy to do what I told it—while the compiler would stop me from being egregiously stupid, it wasn't going to stop me from doing something like this:

Rust 运行时非常非常难出错。编译器让我确信我检查了每一个代码路径，总是让我确信我覆盖了所有角落。Go 就像我说的那样，

工作的很爽 – 但是它的编译器会让我觉得很傻，它不会阻止我做如下一些事情：

https://gist.github.com/adamhjk/9c7eae1d053a1788f581#file-panic-go

Where line 9 there just blindly assumes the regex found a match, and causes quite the run-time error message. This was impossible

in Rust:

第九行盲目的假设 regex 匹配一个结果，然后导致运行时错误信息。这在 Rust 中绝不会发生。

https://gist.github.com/adamhjk/2eb74d326242a9093f9e#file-current_branch-rs

Because the return of my regex capture as an Option, the compiler knew I needed to deal with all the possible values. I

literally couldn't choose to ignore it. This happened when I wrote the code—in Go, I just wrote what I thought I wanted, then learned about my mistake at runtime (and it's a trivial fix.) In Rust, the compiler told me when I first tried the function that I hadn't dealt with the None result of my Option<Capture> return.

因为 regex 捕获的返回类型是 Option 选项，编译器知道我需要处理所有可能的值。我没有办法选择忽视它。这个在我写代码的时候

就发生了-在 Go 中，我想我这么写就是我要的，然后在运行时才知道哪里错了（这是个很小的 fix）在 Rust 中，编译器在我第一次尝试这个函数时告诉我，我还没有对 Option<Capture>返回值处理 None 结果。

Speed, accessibility, and libraries 速度，易接近，还有库

The trade-off is interesting. The Go code took far less time to write, the language was so accessible, and the ecosystem was strong. It let me work like I would have in a scripting language, while stopping the most egregious kinds of errors. Because the only language constructs are the most common constructs, it took no time at all to feel facile in the language.

代价是兴趣。Go 代码用更少的时间来写，语言容易接近，生态系统强大。它让我感觉在用脚本语言在工作，但同时避免了大多数

恶名昭著的错误。因为语言唯一的构造是非常同样的构造，根本没有时间体会到语言的灵活性。

Go had a huge library ecosystem, a clear way to find them, and an easy way to get them.

Go 有巨大的库生态系统，很容易找到和获取它们。

So.. Rust or Go?
所以。。Rust还是Go?

The pragmatic answer is ‘horses for courses’, I guess. Go could clearly fill the same niche that people write Ruby, Python, or Perl for. It’s an easy, approachable language with a strong and fabulous ecosystem. I see why people love it. Static binaries, great ecosystem. Kind of a layup. I wrote the Go code probably twice as fast as as the Rust code.

我猜实用主义的回答是‘物尽其善，那个实用用哪个’。Go 可以替代人们用 Ruby，Pythong 或者 Perl 所写的东西。它是一个简单，

可接近的语言，加上强类型和难以置信的生态系统。

我明白人们为什么喜欢它。静态二进制，很好的生态系统，配合的很好。我写 Go 代码大概比 Rust 快一倍。

The thing is, I liked Rust more. I **like** that it has pattern matching, generics, immutable data by default. I liked that if my code compiled at all, the odds that it did what I wanted were extraordinarily high. When I think about what an ecosystem built on those fundamentals would be like, I want to live in that world. It's better than the world I live in on languages I already have an affinity for for a wide range of use cases, and lets me play in a space I haven't really ever felt comfortable in. I'm pretty sure the advice that Rust is amazing where you would use C/C++, and Go is amazing where you would use Ruby or Python is true. However, I think once Rusts ecosystem takes off, it will start being a great choice for the Ruby/Python niche as well.

但是，我更喜欢 **Rust**。我喜欢它有模式匹配，泛型，缺省的不可变数据。我喜欢如果我的代码编译通过，程序按我设计的意愿运行的

可能性是极高的。当我想象一个生态系统将在这样的一个基础上建立出来时，我愿意活在那个世界里。这比我依赖大量语言用例，

并且让我没有真正感觉到舒适的空间的世界要好。我很确定如果你用 C/C++的话，Rust 将非常吸引你。而你用 Ruby 或 Python 的话

Go 对你更有吸引力。然而，我想一旦 Rusts 生态系统起飞，它将同时吸引那些使用 Ruby/Python 的人。

I was reminded of a lesson I learned about executive hiring—rule number one is that they have to be amazing at something. Someone who is just "good" at everything won't, in general, actually turn out to be great. Go felt that way to me—it was good at everything, but nothing grabbed me and made me feel excited in a way I wasn't already about something else in my ecosystem. Rust is absolutely amazing at ensuring the code you write is correct and safe at runtime—it's a revelation

我记得一次高层招聘规则获取的教训，第一条是他/她必须在某些方面让人惊讶.一个在很多方面"好"的人，通常来说不是极佳的。

Go 就是给我这种感觉-它在每一方面都好，但没有抓住我的心和让我觉得因为在我已有的生态系统里面没有这些东西而觉得非常兴奋，

Rust 非常迷人的地方在于确保你写的代码是正确的和运行时安全的-这是一种启示。

You should definitely give both a try.

你应该要两者同时试一下。