Notes Collection of General Knowledge

General Knowledge

THEQIONG.COM

穷屌丝联盟

Beyond General Knowledge

穷屌丝联盟

2014 年 3 月 25 日

Contents

List of Figures

List of Tables

1

如何掌握程序语言

> 这篇文章曾经叫做《初学者程序语言的选择》，但是后来我发现，这里给出的看法其实不只是给初学者看的，甚至可能会让初学者看不懂。而就我在 Google 实习的时候的观察看来，很多写了几十年程序的资深程序员，可能也没有明白这里指出的道理。所以我把题目改了一下，并且加入了新的内容，希望对新手和老手都有所帮助。[1]

学习程序语言是每个程序员的必经之路。可是这个世界上有太多的程序语言，每一种都号称具有最新的“特性”。所以程序员的苦恼就在于总是需要学习各种稀奇古怪的语言，而且必须紧跟“潮流”，否则就怕被时代所淘汰。

作为一个程序语言的研究者，我深深的知道这种心理产生的根源。程序语言里面其实有着非常简单，永恒不变的原理。看到了它们，就可以一劳永逸的掌握所有的程序语言，而不是只见树木不见森林。我想写一本书，试图用最简单的方式来解释程序语言 (以至于计算机科学) 的根本性原理，从而让人可以在非常短的时间内掌握所有程序语言的精髓。但是在还没有完成之前，我想先提出一些建议和参考书。

1.1 对程序语言的各种误解

学习程序语言的人，经常会出现以下几种心理，以至于他们会觉得有学不完的东西，或者走上错误的道路。以下我把这些心理简要分析一下，希望可以消除一些疑惑。

1. 追求“新语言”。

基本的哲学告诉我们，新出现的事物并不一定是“新事物”，它们有可能是历史的倒退。事实证明，新出现的语言，很多还不如早就存在的。正视这个事实吧，现代语言的多少“新概念”不存在于最古老的一些语言里呢？程序语言就像商品，每一家其实都是

在打广告。而绝大多数的设计，包括某些最“艰深”最“理论”的语言里面的概念，都可能是肤浅而短命的。如果你看不透这些东西的设计，就会被它们蒙蔽住。过度的热情和过多的宣传，往往意味着肤浅。很多语言设计者其实并不真的懂得程序语言设计的原理，所以常常在设计中重复的犯前人犯过的错误。但是为了推销自己的语言和系统，他们必须夸夸其谈，进行宗教式的宣传。

2. “存在即是合理”。

记得某名人说过：“不能带来新的思维方式的语言，是没有必要存在的。”他说的是相当正确的。世界上有这么多的语言，有哪些带来了新的思维方式呢？其实少之又少。绝大部分的语言给世界带来的不过是混乱。有人可能反驳说：“你怎么能说 A 语言没必要存在？我要用的那个库 L，别的语言不支持，只能用 A。”但是注意，他说的是存在的“必要性”。

如果你把存在的“事实”作为存在的“必要性”，那就逻辑错乱了。就像如果二战时我们没能打败希特勒，现在都做了他的奴隶，然后你就说：“希特勒应该存在，因为他养活了我们。”显然这个逻辑有问题，因为如果历史走了另外一条路（即希特勒不存在），我们会过上自由幸福的生活，所以希特勒不应该存在。对比一个东西存在与不存在的两种可能的后果，然后做出判断，这才是正确的逻辑。按照这样的推理，如果设计糟糕的 A 语言不存在，那么设计更好的 B 语言很有可能就会得到更多的支持，从而实现甚至超越 L 库的功能。

3. 追求“新特性”。

程序语言的设计者总是喜欢“发明”新的名词，喜欢炒作。普通程序员往往看不到，大部分这些“新概念”其实徒有高深而时髦的外表，却没有实质的内涵。常常是刚学会一个语言 A，又来了另一个语言 B，说它有一个叫 XYZ 的新特性。于是你又开始学习 B，如此继续。在内行人看来，这些所谓的“新特性”，绝大部分都是新瓶装老酒。很多人写论文喜欢起这样的标题：《XYZ：A Novel Method for ...》。这造成了概念的爆炸，却没有实质的进步。可以说这是计算机科学最致命的缺点。

4. 追求“小窍门”。

很多编程书喜欢卖弄一些小窍门，让程序显得“短小”。比如它们会跟你讲“`(i++) - (++i)`”应该得到什么结果；或者追究运算符的优先级，说这样可以少打括号；要不就是告诉你“if 后面如果只有一行代码就可以不加花括号”，等等。殊不知这些小窍门，其实大部分都是程序语言设计的败笔或者历史遗留问题。它们带来的不是清晰的思路，而是是逻辑的混乱和认知的负担。比如 C 语言的 ++ 运算符，它的出现是因为 C 语言设计者们当初用的计算机内存小的可怜，而“`i++`”显然比“`i=i+1`”少 2 个字符，所以他们觉得可以节省一些空间。现在我们再也不缺那点内存，可是 ++ 运算符带来的混乱和迷惑，

却流传了下来。现在最新的一些语言，也喜欢耍这种语法上的小把戏。如果你追求这些小窍门，往往就抓不住精髓。

5. 针对“专门领域”。

很多语言没有新的东西，为了占据一方土地，就号称自己适合某种特定的任务，比如文本处理，数据库查询，Web 编程，游戏设计，并行计算，或者别的什么专门的领域。但是我们真的需要不同的语言来干这些事情吗？其实绝大部分这些事情都能用同一种通用语言来解决，或者在已有语言的基础上做很小的改动。只不过由于各种政治和商业原因，不同的语言被设计用来占领市场。就学习而言，它们其实是无关紧要的，而它们带来的“多语言协作”问题，其实差不多掩盖了它们带来的好处。其实从一些设计良好的通用语言，你可以学会所有这些“专用语言”的精髓。后面我会推荐一两个这样的语言。

我必须指出，以上这些心理不但对自己是有害的，而且对整个业界有很大的危害。受到这些思想教导的人进入了公司，就会开始把他们曾经惧怕的这些东西变成教条，用来筛选新人，从而导致恶性循环。

1.2 如何掌握所有的程序语言

可以老实的说，对几乎所有风格的程序语言，我都有专家级的见解。它们在我的头脑里如此简单，以至于我不再是任何语言（包括函数式语言）的“支持者”。但是我花费了太多的时间去摸索这条道路，我希望能够提取出一些“窍门”，可以帮助人们在短时间内达到这种通用的理解。具体的细节足够写成一本书，我现在只在这里提出一些初步的建议。

1. 专注于“精华”和“原理”。

就像所有的科学一样，程序语言最精华的原理其实只有很少数几个，它们却可以被用来构造出许许多多纷繁复杂的概念。但是人们往往忽视了简单原理的重要性，匆匆看过之后就去追求最新的，复杂的概念。他们却没有注意到，绝大部分最新的概念其实都可以用最简单的那些概念组合而成。而对基本概念的一知半解，导致了他们看不清那些复杂概念的实质。比如这些概念里面很重要的一个就是递归。国内很多学生对递归的理解只停留于汉诺塔这样的程序，而对递归的效率也有很大的误解，认为递归没有循环来得高效。而其实递归比循环表达能力强很多，而且效率几乎一样。有些程序比如解释器，不用递归的话基本没法完成。

2. 实现一个程序语言。

学习使用一个工具的最好的方式就是制造它，所以学习程序语言的最好方式就是实现一个程序语言。这并不需要一个完整的编译器，而只需要写一些简单的解释器，实现

最基本的功能。之后你就会发现，所有语言的新特性你都大概知道可以如何实现，而不只停留在使用者的水平。实现程序语言最迅速的方式就是使用一种像 Scheme 这样代码可以被作为数据的语言。它能让你很快的写出新的语言的解释器。我的 GitHub 里面有一些我写的解释器的例子（比如这个短小的代码实现了 Haskell 的 lazy 语义），有兴趣的话可以参考一下。

1.3 几种常见风格的语言

下面我简要的说一下几种常见风格的语言以及它们的问题。注意这里的分类不是严格的学术性质的分类，有些在概念上可能有所重叠。

1. 面向对象语言

事实说明，“面向对象”这整个概念基本是错误的。它的风靡是因为当初的“软件危机”(天知道是不是真的存在这危机)。设计的初衷是让“界面”和“实现”分离，从而使得下层实现的改动不影响上层的功能。可是大部分面向对象语言的设计都遵循一个根本错误的原则：“所有的东西都是对象（Everything is an object)。”以至于所有的函数都必须放在所谓的“对象”里面，从而不能直接被作为参数或者变量传递。这导致很多时候需要使用繁琐的设计模式 (design patterns) 来达到甚至对于 C 语言都直接了当的事情。而其实“界面”和“实现”的分离，并不需要把所有函数都放进对象里。另外的一些概念，比如继承、重载，其实带来的问题比它们解决的还要多。

“面向对象方法”的过度使用，已经开始引起对整个业界的负面作用。很多公司里的程序员喜欢生搬硬套一些不必要的设计模式，其实什么好事情也没干，只是使得程序冗长难懂。不得不指出，《Design Patterns》这本书，是这很大一部分复杂性的罪魁祸首。不幸的是，如此肤浅，毫无内容，偷换概念的书籍，居然被很多人捧为经典。

那么如何看待具备高阶函数的面向对象语言，比如 Python，JavaScript，Ruby，Scala？当然有了高阶函数，你可以直截了当的表示很多东西，而不需要使用设计模式。但是由于设计模式思想的流毒，一些程序员居然在这些不需要设计模式的语言里也采用繁琐的设计模式，让人哭笑不得。所以在学习的时候，最好不要用这些语言，以免受到不必要的干扰。到时候必要的时候再回来使用它们，就可以取其精华，去其糟粕。

2. 低级过程式语言

那么是否 C 这样的“低级语言”就会好一些呢？其实也不是。很多人推崇 C，因为它可以让人接近“底层”，也就是接近机器的表示，这样就意味着它速度快。这里其实有三个问题：

- 接近“底层”是否对于初学者是好事？

- “速度快的语言”是什么意思？
- 接近底层的语言是否一定速度快？

对于第一个问题，答案是否定的。其实编程最重要的思想是高层的语义 (semantics)。语义构成了人关心的问题以及解决它们的算法。而具体的实现 (implementation)，比如一个整数用几个字节表示，虽然还是重要，但却不是至关重要的。如果把实现作为学习的主要目标，就本末倒置了。因为实现是可以改变的，而它们所表达的本质却不会变。所以很多人发现自己学会的东西，过不了多久就“过时”了。那就是因为他们学习的不是本质，而只是具体的实现。

其次，谈语言的“速度”，其实是一句空话。语言只负责描述一个程序，而程序运行的速度，其实绝大部分不取决于语言。它主要取决于 1) 算法和 2) 编译器的质量。编译器和语言基本是两码事。同一个语言可以有很多不同的编译器实现，每个编译器生成的代码质量都可能不同，所以你没法说“A 语言比 B 语言快”。你只能说“A 语言的 X 编译器生成的代码，比 B 语言的 Y 编译器生成的代码高效”。这几乎等于什么也没说，因为 B 语言可能会有别的编译器，使得它生成更快的代码。

我举个例子吧。在历史上，Lisp 语言享有“龟速”的美名。有人说“Lisp 程序员知道每个东西的值，却不知道任何事情的代价”，讲的就是这个事情。但这已经是很久远的事情了，现代的 Lisp 系统能编译出非常高效的代码。比如商业的 Chez Scheme 编译器，能在 5 秒钟之内编译它自己，编译生成的目标代码非常高效。它的实现真的令人惊叹，因为它的作者 R. Kent Dybvig 几乎不依赖于任何已有的软件和设计。这个编译器从最初的 parser，到宏扩展，语义分析，寄存器分配，各种优化，……一直到汇编器，函数库，全都是他一个人写的。它可以直接把 Scheme 程序编译到多种处理器的机器指令，而不通过任何第三方软件。它内部的一些算法，其实比开源的 LLVM 之类的先进很多。但是由于是商业软件，这些算法一直被作为机密没有发表。

另外一些函数式语言也能生成高效的代码，比如 OCaml。在一次程序语言暑期班上，Cornell 的 Robert Constable 教授讲了一个故事，说是他们用 OCaml 重新实现了一个系统，结果发现 OCaml 的实现比原来的 C 语言实现快了 50 倍。经过 C 语言的那个小组对算法多次的优化，OCaml 的版本还是快好几倍。这里的原因其实在于两方面。第一是因为函数式语言把程序员从底层细节中解脱出来，让他们能够迅速的实现和修改自己的想法，所以他们能够迅速的找到更好的算法。第二是因为 OCaml 有高效的编译器实现，使得它能生成很好的代码。

从上面的例子，你也许已经可以看出，其实接近底层的语言不一定速度就快。因为编译器这种东西其实可以有很高级的“智能”，甚至可以超越任何人能做到的底层优化。但是编译器还没有发展到可以代替人来制造算法的地步。所以现在人需要做的，其实只

是设计和优化自己的高层算法。

3. 高级过程式语言

很早的时候，国内计算机系学生的第一门编程课都是 Pascal。Pascal 是很好的语言，可是很多人当时都没有意识到。大一的时候，我的 Pascal 老师对我们说：“我们学校的教学太落后了。别的学校都开始教 C 或者 C++ 了，我们还在教 Pascal。”现在真正理解了程序语言的设计原理以后我才真正的感觉到，原来 Pascal 是比 C 和 C++ 设计更好的语言。

它不但把人从底层细节里解脱出来，没有面向对象的思维枷锁，而且含有函数式语言的一些特征（比如可以嵌套函数定义）。可是由于类似的误解和误导，Pascal 这样的语言已经几乎没有人用了。这并不很可惜，因为它的精髓，其实已经存在于像 Scheme 这样的函数式语言里。Scheme 也有赋值语句，所以它实质上含有 Pascal 的所有功能。所以现在的含有赋值语句的函数式语言，可以被看作是是高级过程式语言的“改良版本”。

4. 函数式语言

函数式语言相对来说是当今最好的设计，因为它们不但让人专注于算法和对问题的解决，而且没有面向对象语言那些思维的限制。但是需要注意的是并不是每个函数式语言的特性都是好东西。它们的支持者们经常把缺点也说成是优点，结果它们其实还是被挂上一些不必要的枷锁。比如 OCaml 和 SML，因为它们的类型系统里面有很多不成熟的设计，导致你需要记住太多不必要的限制。

很多人推崇“纯函数式”语言（比如 Haskell，Clean），而极力反对其它的带有“赋值”语句的语言（比如 Scheme 和 ML）。这其中的依据其实是站不住脚的。如果你写过一个函数式语言的编译器，你就会了解如何把一个纯函数式语言翻译成机器指令。这些高级的编译器变换（比如 CPS 和 ANF），其实在本质上揭示了纯函数式语言的本质。它们其实与带有赋值语句的语言没有本质上的区别，但是由于没有赋值语句，一些事情必须拐弯抹角的实现。理智的使用局部变量或者数组的赋值，会使程序更加简单，容易理解，甚至更加高效。

5. 逻辑式语言

逻辑式语言（比如 Prolog）是一种超越函数式语言的新的思想，所以需要一些特殊的训练。逻辑式语言写的程序，是能“反向运行”的。普通程序语言写的程序，如果你给它一个输入，它会给你一个输出。但是逻辑式语言很特别，如果你给它一个输出，它可以反过来运行，给你所有可能的输入。其实通过很简单的方法，可以顺利的把程序从函数式转换成逻辑式的。但是逻辑式语言一般要在“pure”的情况下（也就是没有复杂的赋值操作）才能反向运行。所以学习逻辑式语言最好是从函数式语言开始，在理解了递归，模式匹配等基本的函数式编程技巧之后再来看 Prolog，就会发现逻辑式编程简单了很多。

1.4 从何开始

可是学习编程总要从某种语言开始。那么哪种语言呢？其实每种语言都有自己的问题，以至于在我未来的书里，会使用一种非常简单的语言，它含有所有语言的精髓，却不带有多余的东西。可是在我完成这本书之前，我想先推荐一两个现成的语言。

就我的观点，首先可以从 Scheme 入门，然后学习一些 Haskell (但不是全部)，之后其它的也就触类旁通了。你并不需要学习它们的所有细枝末节，而只需要学习最精华的部分。所有剩余的细节，会在实际使用中很容易的被填补上。我后面会提一下哪些是精华的，哪些是最开头没必要学的。

从 Scheme（而不是 Haskell）作为入门的第一步，是因为：

1. Scheme 没有像 Haskell 那样的静态类型系统 (static type system)。并不是说静态类型不好，但是我不得不说，Haskell 那样的静态类型系统，还远远没有发展到可以让人可以完全的写出符合事物本质的程序来。比如，一些重要的概念比如 Y combinator，没法用 Haskell 直接写出来。当然你可以在 Haskell 里面使用作用类似 Y combinator 的东西 (比如 fix，或者利用它的 laziness)，但是这些并不揭示递归的本质，你只是在依靠 Haskell 已经实现的递归来进行递归，而不能实际的体会到递归是如何产生的。而用 Scheme，你可以轻松的写出 Y combinator，并且实际的投入使用。
2. Scheme 不需要 monad。Haskell 是一个“纯函数式”(purely functional) 的语言，所有的“副作用”(side-effect)，比如打印字符到屏幕，都得用一种故作高深的概念叫 monad 实现。这种概念其实并不是本质的，它所有的功能都可以通过“状态传递”(state passing) 来实现。通过写状态传递程序，你可以清楚的看到 monad 的本质。可以说 monad 是 Haskell 的一个“设计模式”。过早的知道这个东西，并不有助于理解函数式程序设计的本质。

那么为什么又要学 Haskell？那是因为 Haskell 含有 Scheme 缺少的一些东西，并且没有 Scheme 设计上的一些问题。比如：

1. 模式匹配：Scheme 没有一个标准的，自然的模式匹配 (pattern matching) 系统，而 Haskell 的模式匹配是一个优美的实现。也有些 Scheme 的扩展实现（比如 Racket）具有相当好的模式匹配机制。
2. 类型：Scheme 把所有不是 `#f (false)` 的值都作为 true，这是不对的。Haskell 里面的 Boolean 就只有两个值：True 和 False。Scheme 程序员声称这样可以写出简洁的代码，因为 (or x y z) 可以返回一个具体的值，而不只是一个布尔变量。但是就为了在少数情况下可以写出短一点的代码，是否值得付出如此沉痛的代价？我看到这个设计带来了很多无需有的问题。
3. 宏系统：宏 (macro) 通常被认为是 Lisp 系列语言的一个重要优点。但是我要指出的

是，它们并不是必要的，至少对于初学者是这样。其实如果一个语言的语义设计好了，你会几乎不需要宏。因为宏的本质是让程序员可以自己修改语言的设计，添加新的构造。可是宏的主要缺点是，它把改变语言这种极其危险的“权力”给人滥用了。其实只有极少数的人具有改变一个语言所需的智慧和经验。如果让普通程序员都能使用宏，那么程序将变得非常难以理解。所以其实一般程序员都不需要学习宏的使用，也不必为略过这个东西而产生负罪感。等你进步到可以设计自己的程序语言，你自然会明白宏是什么东西。[1]

1.5 推荐的书籍

《The Little Schemer》：我觉得 Dan Friedman 的 The Little Schemer (TLS) 是目前最好，最精华的编程入门教材。它的前身叫《The Little Lisper》。很多资深的程序语言专家都是从这本书学会了 Lisp。虽然它叫”The Little Schemer”，但它并不使用 Scheme 所有的功能，而是忽略了上面提到的 Scheme 的毛病，直接进入最关键的主题：递归和它的基本原则。这本书不但很薄，很精辟，而且相对于其他编程书籍非常便宜（在美国才卖 $23）。

《SICP》：The Little Schemer 其实是比较难的读物，所以我建议把它作为下一步精通的读物。Structure and Interpretation of Computer Programs 比较适合作为第一本教材。但是我需要提醒的是，你最多只需要看完前三章。因为从第四章开始，作者开始实现一个 Scheme 解释器，但是作者的实现并不是最好的方式。你可以从别的地方更好的学到这些东西。具体在哪里学，我还没想好（也许我自己写个教学也说不定）。不过也许你可以看完 SICP 第一章之后就可以开始看 TLS。

《A Gentle Introduction to Haskell》: 对于 Haskell，我最开头看的是 A Gentle Introduction to Haskell，因为它特别短小。当时我已经会了 Scheme，所以不需要再学习基本的函数式语言的东西。我从这个文档学到的只不过是 Haskell 对于类型和模式匹配的概念。Real World Haskell 是一本流行的教材，但是它试图包罗万象，所以很多地方过于冗长。最根本的函数式编程概念，还是 TLS 讲的透彻。

1.6 过渡到面向对象语言

那么如果从函数式语言入门，如何过渡到面向对象语言呢？毕竟大部分的公司用的是面向对象语言。如果你真的学会了函数式语言，你真的会发现面向对象语言已经易如反掌。函数式语言的设计比面向对象语言简单和强大很多，而且几乎所有的函数式语言教材（比如 SICP）都会教你如何实现一个面向对象系统。

[1]作者注：注意，这些是我自己的观点，并不代表 Scheme 设计者们的观点。

你会深刻的看到面向对象的本质以及它存在的问题，所以你会很容易的搞清楚怎么写面向对象的程序，并且会发现一些窍门来避开它们的局限。你会发现，即使在实际的工作中必须使用面向对象语言，也可以避免面向对象的思维方式，因为面向对象的思想带来的大部分是混乱和冗余。

1.7 深入本质和底层

那么是不是完全不需要学习底层呢？当然不是。但是一开头就学习底层硬件，就会被纷繁复杂的硬件设计蒙蔽头脑，看不清楚本质上简单的原理。

在学会高层的语言之后，可以进行语义学和编译原理的学习。简言之，语义学 (semantics) 就是研究程序的符号表示如何对机器产生“意义”，通常语义学的学习包含 lambda calculus 和各种解释器的实现。编译原理 (compilation) 就是研究如何把高级语言翻译成低级的机器指令。

编译原理其实包含了计算机的组成原理，比如二进制的构造和算术，处理器的结构，内存寻址等等。但是结合了语义学和编译原理来学习这些东西，会事半功倍。因为你会直观的看到为什么现在的计算机系统会设计成这个样子：为什么处理器里面有寄存器 (register)，为什么需要堆栈 (stack)，为什么需要堆 (heap)，它们的本质是什么。

这些甚至是很多硬件设计者都不明白的问题，所以它们的硬件里经常含有一些没必要的东西。因为他们不理解语义，所以经常不明白他们的硬件到底需要哪些部件和指令。但是从高层语义来解释它们，就会揭示出它们的本质，从而可以让你明白如何设计出更加优雅和高效的硬件。

这就是为什么一些程序语言专家后来也开始设计硬件。比如 Haskell 的创始人之一 Lennart Augustsson，后来设计了 BlueSpec，一种高级的硬件描述语言，可以 100% 的合成 (synthesis) 为硬件电路。Scheme 也被广泛的使用在硬件设计中，比如 Motorola，Cisco 和曾经的 Transmeta，它们的芯片设计里面含有很多 Scheme 程序。

这基本上就是我对学习程序语言的初步建议。

来源

[1] 王垠. 如何掌握程序语言. URL http://kb.cnblogs.com/page/152132/.

2

从工具的奴隶到工具的主人

我们每个人都是工具的奴隶。随着我们的学习，我们不断的加深自己对工具的认识，从而从它们里面解脱出来。现在我就来说一下我作为各种工具的奴隶，以及逐渐摆脱它们的“思想控制”的历史吧。

当我高中毕业进入大学计算机系的时候，辅导员对我们说：“你们不要只学书本知识，也要多见识一下业界的动态，比如去电脑城看看人家怎么装机。”当然他说我们要多动手，多长见识，这是对的。不过如果成天就研究怎么“装机”，研究哪种主板配哪种 CPU 之类的东西，你恐怕以后就只有去电脑城卖电脑了。

本科的时候，我经常发现一些同学不来上数学课。后来却发现他们在宿舍自己写程序，对 MFC 之类的东西津津乐道，引以为豪。当然会用 MFC 没有什么不好，可是如果你完全沉迷于这些东西，恐怕就完全局限于 Windows 的一些表面现象了。

所以我在大学的时候就开始折腾 Linux，因为它貌似让我能够“深入”到计算机内部。那个时候，书店里只有一本 Linux 的书，封面非常简陋。这是一本非常古老的书，它教的是怎样得到 Slackware Linux，然后把它从二三十张软盘装到电脑上。总之，我就是这样开始使用 Linux 的。后来我就走火入魔了，有时候上课居然在看 GCC 的内部结构文档。后来我又开始折腾 TeX，把 TeXbook 都看了两遍，恁是用它写了我的本科毕业论文。

后来进了清华，因为不满意有人嘲笑我用 Linux 这种“像 DOS 的东西”，以及国内网站都对 Windows 和 IE 进行“优化”的情况，就写了个“完全用 Linux 工作”。确实，会 Linux 的人现在更容易找到工作，更容易被人当成高手。但是那些工具同样的奴役了我，经常以一些雕虫小技而自豪，让我看不到如何才能设计出新的，更好的东西。当它们的设计改变的时候，我就会像奴隶一样被牵着鼻子走。

这也许就是为什么我在清华的图书馆发现《SICP》的时候如此的欣喜。那本书是崭新的，后面的借书记录几乎是空白的。这些看似简单的东西教会我的，却比那些大部头

和各种 HOWTO 教会我的更多，因为它们教会我的是 WHY，而不只是 HOW。当时我就发现，虽然自认为是一个“资深”的研究生，学过那么多种程序语言，各种系统工具甚至内核实现，可是相对于 SICP 的认识深度，我其实几乎完全不会写程序！在第三章，SICP 教会了我如何实现一个面向对象系统。这是我第一次感觉到自己真正的在开始认识和控制自己所用的工具。

因为通常人们认为 Scheme 不是一个“实用”的语言，没有很多“库”可以用，效率也不高，而 Common Lisp 是“工业标准”，再加上 Paul Graham 文章的怂恿，所以我就开始了解 Common Lisp。在那段时间，我看了 Paul Graham 的《On Lisp》和 Peter Norvig 的《Paradigms of Artificial Intelligence Programming》。怎么说呢？当时我以为自己学到很多，可是现在看来，它们教会我的并没有《SICP》的东西那么精髓和深刻。开头以为一山还有一山高，最后回头望去，其实复杂的东西并不比简单的好。现在当我再看 Paul Graham 和 Peter Norvig 的文章，就觉得相当幼稚了，而且有很大的宗教成分。

进入 Cornell 之后，因为 Cornell 的程序语言课是用 SML 的，我才真正的开始学习“静态类型”的函数式语言。之前在清华的时候，有个同学建议我试试 ML 和 Haskell，可是因为我对 Lisp 的执着，把他的话当成了耳边风。当然现在用上 SML 就免不了发现 ML 的类型系统的一些挠人的问题，所以我就开始了解 Haskell，并且由于它看似优美的设计，我把“终极语言”的希望寄托于它。我开始着迷一些像 monads, type class, lazy evaluation 一类的东西，看 Simon Peyton Jones 的一些关于函数式语言编译器的书。以至于走火入魔，对其它一切“常规”语言都持鄙视态度，看到什么都说“那只不过是个 monad”。虽然有些语言被鄙视是合理的，有些却是被错怪了的。后来我也发现 monad, type class, lazy evaluation 这些东西其实并不是什么包治百病的灵丹妙药。

但是我很不喜欢 Cornell 的压抑气氛，所以最后决定离开。在不知何去何从的时候，我发了一封 email 给曾经给过我 fellowship 的 IU 教授 Doug Hofstadter （《GEB》的作者）。我说我不知道该怎么办，后悔来了 Cornell，我现在对函数式语言感兴趣。他跟我说，IU 的 Dan Friedman 就是做函数式语言的啊，你跟他联系一下，就说是我介绍你来的。我开头看过一点 The Little Schemer，跟小人书似的，所以还以为 Friedman 是个年轻小伙。当我联系上 Friedman 的时候，他貌似早就认识我了一样。他说当年你的申请材料非常 impressive，可惜你最后没有选择我们。你要知道，世界上最重要的不是名气，而是找到赏识你，能够跟你融洽共事的人。你的材料都还在，我会请委员会重新考虑你的申请。IU 的名气实在不大，而 Friedman 实在是太谦虚了，所以连跟他打电话都没有明确表态想来 IU，只是说“我考虑一下……”这就是我怎么进入 IU 的。

Friedman 的教学真的有一手。虽然每个人对他看法不同，但是有几个最重要的地方他的指点是帮了我大忙的。有人可能想象不到，在 Scheme 这种动态类型语言的“老槽”，

其实有人对“静态类型系统”的理解如此深刻。也就是在 Friedman 的指点下，我发现类型推导系统不过是一种“抽象解释”，而各种所谓的“typing rule”，不过是抽象解释器里面的分支语句。我后来就通过这个“直觉”，再加上 Friedman 的逻辑语言 miniKanren 里面对逻辑变量和 unification 的实现，做出了一个 Hindley-Milner 类型推导系统（HM 系统），也就是 ML 和 Haskell 的类型系统。虽然我在 Cornell 的课程作业里实现过一个 HM 系统，但是直到 Friedman 的提点，我才明白了它“为什么”是那个样子，以至于达到更加优美的实现。后来经他一句话点拨，我又写出了一个 lazy evaluation 的解释器（也就是 Haskell 的语义），才发现原来 SPJ 的书里所谓的“graph reduction”，不过就是如此简单的思想。只不过在 SPJ 的书里，细节掩盖了本质。后来我在之前的 HM 系统之上做了一个非常小的改动，就实现了 type class 的功能，并且比 Haskell 的实现更加灵活。所以，就此我基本上掌握了 ML 和 Haskell 的理论精髓。

可是类型系统却貌似一个无止境的东西。在 ML 的系统之上，还有 System F，Fw, MLF，Martin Lof Type Theory，CIC，……怎么没完没了？我一直觉得这些东西过度复杂，有那个必要吗？直到 Amal Ahmed 来到 IU，我才相信了自己的感觉。然而，这却是以一种“反面”的方式达到的。

Amal 是著名的 Andrew Appel （“虎书”的作者）的学生，在类型系统和编译器的逻辑验证方面做过很多工作。可是她比较让人受不了，她总是显得好像自己是这里唯一懂得类型的人，而其他人都是类型白痴。她不时的提到跟 Bob Harper, Benjamin Pierce 等类型大牛一起合作的事情。如果你问她什么问题，她经常会回答你：“Bob Harper 说……”她提到一个术语的时候总是把它说得无比神奇，把它的提出者的名字叫得异常响亮。有一次她上课给我们讲 System F，我问她，为什么这个系统有两个“binder”，貌似太复杂了，为什么不能只用一个？她没有正面回答，而是嘲讽似的说：“不是你说可以就可以的。它就是这个样子的。”后来我却发现其实有另外一个系统，它只有一个 binder，而且设计得更加简洁。后来我又在课程的 mailing list 问了一个问题，质疑一个编译器验证方面的概念。本来是纯粹的学术讨论，却发现这封 email 根本没有发到全班同学信箱里，被 Amal 给 moderate 掉了！

看到这种种诡异的行为，我才意识到原来学术界存在各种“帮派”。即使一些人的理论完全被更简单的理论超越，他们也会为“自己人”的理论说话，让你搞不清到底什么好，什么不好。所以后来我对一些类型系统，以及 Hoare Logic 一类的“程序逻辑”产生了怀疑。我的课程 project 报告，就是指出 Hoare Logic 和 Separation Logic 所能完成的功能，其实用“符号执行”或者“model checking”就能完成。而这些程序逻辑所做的事情，不过是把程序翻译成了等价的逻辑表达式而已。到时候你要得知这些逻辑表达式的真伪，又必须经过一个类似程序分析的过程，所以这些逻辑只不过让你白走了一些弯路。

当 Amal 听完我的报告，勉强的笑着说：“你告诉了我们这个结论，可是你能用它来做什么呢？”我才发现原来透彻的看法，并不一定能带来认同。人们都太喜欢“发明”东西，却不喜欢“归并”和“简化”东西。

可是这类型系统的迷雾却始终没有散去，像一座大山压在我头上。我不满意 Haskell 和 ML 的类型系统，又觉得 System F 等过于复杂。可是由于它们的“理论性”和它们创造者的“权威”，我不敢断定自己的看法就不是偏颇的。对付疑惑和恐惧的办法就是面对它们，看透它们，消灭它们。于是，我利用一个 independent study 的时间，独立实现了一个类型系统。我试图让它极度的简单，却又“包罗万象”。经过一番努力，这个类型系统“涵盖”了 System F, MLF 以及另外一些类似系统的推导功能，却不直接“实现”他们。后来我就开始试图让它涵盖一种非常强大的类型系统，叫做 intersection types。这种类型系统的研究已经进行了 20 多年，它不需要程序员写任何类型标记，却可以给任何“停机”的程序以类型。著名的 Benjamin Pierce 当年的博士论文，就是有关 intersection types 的。没几天，我就对自己的系统稍作改动，让它涵盖了一种最强大的 intersection type 系统（System I）的所有功能。然而我却很快发现这个系统是不能实用的，因为它在进行类型推导的时候相当于是在运行这个程序，这样类型推导的计算复杂度就会跟这个程序一样。这肯定是完全不能接受的。后来我才发现，原来已经有人指出了 System I 的这个问题。但是由于我事先实现了这个系统，所以我直接的看到了这个结论，而不需要通过繁琐的证明。

所以，我对类型推导的探索就这样到达了一个终点。我的类型系统是如此的简单，以至于我看到了类型推导的本质，而不需要记住复杂的符号和推理规则。我的系统在去掉了 intersection type 之后，仍然比 System F 和 MLF 都要强大。我也看到了 Hindley-Milner 系统里面的一个严重问题，它导致了这几十年来很多对于相关类型系统的研究，其实是在解决一个根本不存在的问题。而自动定理证明的研究者们，却直接的“绕过”了这个问题。这也就是我为什么开始对自动定理证明开始感兴趣。

后来对自动定理证明，Partial Evaluation 和 supercompilation 的探索，让我看到那些看似高深的 Martin Lof Type Theory, Linear Logic 等概念，其实不过也就是用不同的说法来重复相同的话题。具体的内容我现在还不想谈，但是我清楚的看到在“形式化”的美丽外衣下，其实有很多等价的，重复的，无聊的东西。与其继续“钻研”它们，反复的叨咕差不多的内容，还不如用它们的“精髓”来做点有用的事情。

所以到现在，我已经基本上摆脱了几乎所有程序语言，编译器，类型系统，操作系统，逻辑推理系统给我设置的思维障碍。它们对我来说不再是什么神物，它们的设计者对我来说也不再是高不可攀的权威。我很开心，经过这段漫长的探索，让我自己的思想得到了解放，翻身成为了这些工具的主人。虽然我看到对这些理论工具的研究恐怕早就

已经到达路的尽头，然而它们里面隐含的美却是无价和永恒的。这种美让我对这个世界的许多其它方面有了焕然一新的看法。一个工具的价值不在于它自己，而在于你如何利用它创造出对人有益的东西，以及如何让更多的人掌握它。这就是我打算现在去做的。

3

UNIX 的缺陷

我想通过这篇文章解释一下我对 UNIX 哲学本质的理解。我虽然指出 UNIX 的一个设计问题，但目的并不是打击人们对 UNIX 的兴趣。虽然 UNIX 在基础概念上有一个挺严重的问题，但是经过多年的发展之后，这个问题恐怕已经被各种别的因素所弥补（比如大量的人力)。但是如果开始正视这个问题，我们也许就可以缓慢的改善系统的结构，从而使得它用起来更加高效，方便和安全，那又未尝不可。同时也希望这里对 UNIX 命令本质的阐述能帮助人迅速的掌握 UNIX，灵活的应用它的潜力，避免它的缺点。

通常所说的“UNIX 哲学”包括以下三条原则 [Mcllroy]：

1. 一个程序只做一件事情，并且把它做好。
2. 程序之间能够协同工作。
3. 程序处理文本流，因为它是一个通用的接口。

这三条原则当中，前两条其实早于 UNIX 就已经存在，它们描述的其实是程序设计最基本的原则 —模块化原则。任何一个具有函数和调用的程序语言都具有这两条原则。简言之，第一条针对函数，第二条针对调用。所谓“程序”，其实是一个叫“main”的函数 (详见下文)。

所以只有第三条（用文本流做接口）是 UNIX 所特有的。下文的“UNIX 哲学”如果不加修饰，就特指这第三条原则。但是许多的事实已经显示出，这第三条原则其实包含了实质性的错误。它不但一直在给我们制造无需有的问题，并且在很大程度上破坏前两条原则的实施。然而，这条原则却被很多人奉为神圣。许多程序员在他们自己的程序和协议里大量的使用文本流来表示数据，引发了各种头痛的问题，却对此视而不见。

Linux 有它优于 UNIX 的革新之处，但是我们必须看到，它其实还是继承了 UNIX 的这条哲学。Linux 系统的命令行，配置文件，各种工具之间都通过非标准化的文本流传递数据。这造成了信息格式的不一致和程序间协作的困难。然而，我这样说并不等于

Windows 或者 Mac 就做得好很多，虽然它们对此有所改进。实际上，几乎所有常见的操作系统都受到 UNIX 哲学潜移默化的影响，以至于它们身上或多或少都存在它的阴影。

UNIX 哲学的影响是多方面的。从命令行到程序语言，到数据库，Web……计算机和网络系统的方方面面无不显示出它的影子。在这里，我会把众多的问题与它们的根源—UNIX 哲学相关联。现在我就从最简单的命令行开始吧，希望你能从这些最简单例子里看到 UNIX 执行命令的过程，以及其中存在的问题。(文本流的实质就是字符串，所以在下文里这两个名词通用)

3.1 Linux 命令执行的基本过程

几乎每个 Linux 用户都为它的命令行困惑过。很多人（包括我在内）用了好几年 Linux 也没有完全的掌握命令行的用法。虽然看文档看书以为都看透了，到时候还是会出现莫名其妙的问题，有时甚至会耗费大半天的时间在上面。其实如果看透了命令行的本质，你就会发现很多问题其实不是用户的错。Linux 遗传了 UNIX 的“哲学”，用文本流来表示数据和参数，才导致了命令行难学难用。

我们首先来分析一下 Linux 命令行的工作原理。下图是一个很简单的 Linux 命令运行的过程。当然这不是全过程，但是更具体的细节跟我现在要说的主题无关。

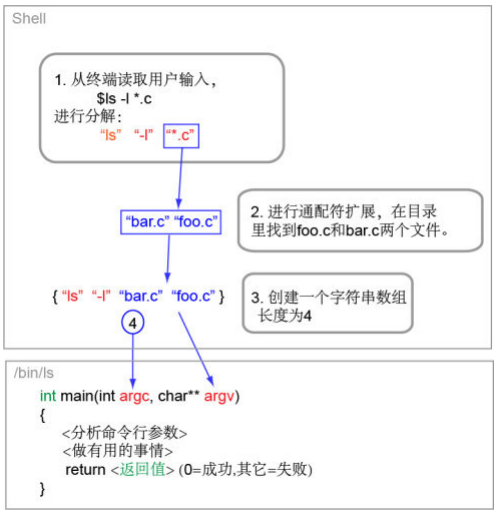

Figure 3.1: Linux 命令运行的过程

从上图我们可以看到，在 ls 命令运行的整个过程中，发生了如下的事情：

1. shell（在这个例子里是 bash）从终端得到输入的字符串`"ls -l *.c"`。然后 shell 以空白字符为界，切分这个字符串，得到 `"ls"`, `"-l"` 和 `"*.c"` 三个字符串。

2. shell 发现第二个字符串是通配符 `"*.c"`，于是在当前目录下寻找与这个通配符匹配的文件。它找到两个文件：foo.c 和 bar.c。
3. shell 把这两个文件的名字和其余的字符串一起做成一个字符串数组 `{"ls", "-l", "bar.c", "foo.c"}`，它的长度是 4。
4. shell 生成一个新的进程，在里面执行一个名叫`"ls"` 的程序，并且把字符串数组 `{"ls", "-l", "bar.c", "foo.c"}` 和它的长度 4，作为 ls 的 main 函数的参数。main 函数是 C 语言程序的“入口”，这个你可能已经知道。
5. ls 程序启动并且得到的这两个参数 (`argv`, `argc`) 后，对它们做一些分析，提取其中的有用信息。比如 ls 发现字符串数组 `argv` 的第二个元素`"-l"` 以`"-"` 开头，就知道那是一个选项 —用户想列出文件详细的信息，于是它设置一个布尔变量表示这个信息，以便以后决定输出文件信息的格式。
6. `ls` 列出 foo.c 和 bar.c 两个文件的“长格式”信息之后退出。以整数 0 作为返回值。
7. shell 得知 `ls` 已经退出，返回值是 0。在 shell 看来，0 表示成功，而其它值（不管正数负数）都表示失败。于是 shell 知道 `ls` 运行成功了。由于没有别的命令需要运行，shell 向屏幕打印出提示符，开始等待新的终端输入……

从上面的命令运行的过程中，我们可以看到文本流（字符串）在命令行中的普遍存在：

- 用户在终端输入是字符串。
- shell 从终端得到的是字符串，分解之后得到 3 个字符串，展开通配符后得到 4 个字符串。
- ls 程序从参数得到那 4 个字符串，看到字符串`"-l"` 的时候，就决定使用长格式进行输出。

接下来你会看到这样的做法引起的问题。

3.2 冰山一角

在《UNIX 痛恨者手册》(The UNIX-Hater's Handbook, 以下简称 UHH) 这本书开头，作者列举了 UNIX 命令行用户界面的一系列罪状，咋一看还以为是脾气不好的初学者在谩骂。可是仔细看看，你会发现虽然态度不好，他们某些人的话里面有非常深刻的道理。我们总是可以从骂我们的人身上学到一些东西，所以仔细看了一下，发现其实这些命令行问题的根源就是“UNIX 哲学”——用文本流（字符串）来表示参数和数据。很多人都没有意识到，文本流的过度使用，引发了太多问题。我会在后面列出这些问题，不过我现在先举一些最简单的例子来解释一下这个问题的本质，你现在就可以自己动手试一

下。

1. 在 Linux 终端里执行如下命令（依次输入：大于号，减号，小写字母 l），这会在目录下建立一个叫”-l” 的文件。

```
$ >-l
```

2. 执行命令 `ls *`（你的意图是以短格式列出目录下的所有文件）。

你看到什么了呢？你没有给 `ls` 任何选项，文件却出人意料的以“长格式”列了出来，而这个列表里面却没有你刚刚建立的那个名叫`"-l"` 的文件。比如我得到如下输出：

```
-rw-r--r-- 1 wy wy 0 2011-05-22 23:03 bar.c
-rw-r--r-- 1 wy wy 0 2011-05-22 23:03 foo.c
```

到底发生了什么呢？重温一下上面的示意图吧，特别注意第二步。原来 shell 在调用 ls 之前，把通配符 * 展开成了目录下的所有文件，那就是 “foo.c”, “bar.c”, 和一个名叫`"-l"` 的文件。它把这 3 个字符串加上 ls 自己的名字，放进一个字符串数组 `{"ls", "bar.c", "foo.c", "-l"}`，交给 ls。接下来发生的是，ls 拿到这个字符串数组，发现里面有个字符串是 `"-l"`，就以为那是一个选项：用户想用“长格式”输出文件信息。因为`"-l"` 被认为是选项，就没有被列出来。于是我就得到上面的结果：长格式，还少了一个文件！

这说明了什么问题呢？是用户的错吗？高手们也许会笑，怎么有人会这么傻，在目录里建立一个叫”-l” 的文件。但是就是这样的态度，导致了我们对错误视而不见，甚至让它发扬光大。其实撇除心里的优越感，从理性的观点看一看，我们就发现这一切都是系统设计的问题，而不是用户的错误。如果用户要上法庭状告 Linux，他可以这样写：

```
起诉状
原告：用户 luser
被告：Linux 操作系统
事由：合同纠纷
1.被告的文件系统给用户提供了机制建立这样一个叫 "-l" 的文件，这表示原告有权使用这个文件名。
2.既然 "-l" 是一个合法的文件名，而 "*" 通配符表示匹配“任何文件”，那么在原告使用 "ls *"
    命令的时候，被告就应该像原告所期望的那样，以正常的方式列出目录下所有的文件，包括 "-l"
    在内。
3.但是实际上原告没有达到他认为理所当然的结果。"-l" 被 ls 命令认为是一个命令行选项，
    而不是一个文件。
4.原告认为自己的合法权益受到侵犯。
```

我觉得为了免去责任，一个系统必须提供切实的保障措施，而不只是口头上的约定来要求用户“小心”。就像如果你在街上挖个大洞施工，必须放上路障和警示灯。你不能只插一面小旗子在那里，用一行小字写着：“前方施工，后果自负。”我想每一个正常人都会判定是施工者的错误。

可是UNIX对于它的用户却一直是像这样的施工者,它要求用户:“仔细看man page,否则后果自负。”其实不是用户想偷懒,而是这些条款太多,根本没有人能记得住。而且没被咬过之前,谁会去看那些偏僻的内容啊。但是一被咬,就后悔都来不及。完成一个简单的任务都需要知道这么多可能的陷阱,那更加复杂的任务可怎么办。其实UNIX的这些小问题累加起来,不知道让人耗费了多少宝贵的时间。

如果你想更加确信这个问题的危险性,可以试试如下的做法。在这之前,请新建一个测试用的目录,以免丢失你的文件!

1. 在新目录里,我们首先建立两个文件夹 `dir-a`、`dir-b` 和三个普通文件 `file1`、`file2` 和`"-rf"`。然后我们运行`"rm *"`,意图是删除所有普通文件,而不删掉目录。

```
$ mkdir dir-a dir-b
$ touch file1 file2
$ > -rf
$ rm *
```

2. 然后用ls查看目录。

你会发现最后只剩下一个文件:`"-rf"`。本来`"rm *"`只能删除普通文件,现在由于目录里存在一个叫`"-rf"`的文件。rm以为那是叫它进行强制递归删除的选项,所以它把目录里所有的文件连同目录全都删掉了(除了`"-rf"`)。

3.3 表面解决方案

难道这说明我们应该禁止任何以`"-"`开头的文件名的存在,因为这样会让程序分不清选项和文件名?可是不幸的是,由于UNIX给程序员的“灵活性”,并不是每个程序都认为以`"-"`开头的参数是选项。比如,Linux下的tar、ps等命令就是例外,所以这个方案不大可行。

从上面的例子我们可以看出,问题的来源似乎是因为ls根本不知道通配符 * 的存在,事实上是shell把通配符展开以后给ls。其实ls得到的是文件名和选项混合在一起的字符串数组。所以UHH的作者提出的一个看法:“shell根本不应该展开通配符。通配符应该直接被送给程序,由程序自己调用一个库函数来展开。”

这个方案确实可行:如果shell把通配符直接给ls,那么ls会只看到“*”一个参数。它会调用库函数在文件系统里去寻找当前目录下的所有文件,它会很清楚的知道`"-l"`“是一个文件,而不是一个选项,因为它根本没有从shell那里得到任何选项(它只得到一个参数:`"*"`),所以问题貌似就解决了。

但是这样每一个命令都自己检查通配符的存在,然后去调用库函数来解释它,大大增加了程序员的工作量和出错的概率。况且shell不但展开通配符,还有环境变量,花括

号展开，～展开，命令替换，算术运算展开……这些让每个程序都自己去做？这恰恰违反了第一条 UNIX 哲学 —模块化原则。而且这个方法并不是一劳永逸的，它只能解决这一个问题。我们还将遇到文本流引起的更多的问题，它们没法用这个方法解决。下面就是一个这样的例子。

3.4 冰山又一角

这些看似微不足道的问题里面其实包含了 UNIX 本质的问题。如果不能正确认识到它，我们跳出了一个问题，还会进入另一个。我讲一个自己的亲身经历吧，前年夏天在 Google 实习快结束的时候发生了这样一件事情……

由于我的项目对一个开源项目的依赖关系，我必须在 Google 的 Perforce 代码库中提交这个开源项目的所有文件。这个开源项目里面有 9000 多个文件，而 Perforce 是如此之慢，在提交进行到一个小时的时候，突然报错退出了，说有两个文件找不到。又试了两次（顺便出去喝了咖啡，打了台球），还是失败，这样一天就快过去了。于是我搜索了一下这两个文件，确实不存在。怎么会呢？我是用公司手册上的命令行把项目的文件导入到 Perforce 的呀，怎么会无中生有？这条命令是这样：

```
find -name *.java -print | xargs p4 add
```

它的工作原理是，find 命令在目录树下找到所有的以 ".java" 结尾的文件，把它们用空格符隔开做成一个字符串，然后交给 xargs。之后 xargs 以空格符把这个字符串拆开成多个字符串，放在 "p4 add" 后面，组合成一条命令，然后执行它。基本上你可以把 find 想象成 Lisp 里的 "filter"，而 xargs 就是 "map"。所以这条命令转换成 Lisp 样式的伪码就是：

```
(map (lambda (x) (p4 add x))
    (filter (lambda (x) (regexp-match? "*.java" x))
           (files-in-current-dir)))
```

问题出在哪里呢？经过一下午的困惑之后我终于发现，原来这个开源项目里某个目录下，有一个叫做 "App Launcher.java" 的文件。由于它的名字里面含有一个空格，被 xargs 拆开成了两个字符串："App" 和 "Launcher.java"。当然这两个文件都不存在了！所以 Perforce 在提交的时候抱怨找不到它们。我告诉组里的负责人这个发现后，他说："这些家伙，怎么能给 Java 程序起这样一个名字？也太菜了吧！"

但是我却不认为是这个开源项目的程序员的错误，这其实显示了 UNIX 的问题。这个问题的根源是因为 UNIX 的命令 (find, xargs) 把文件名以字符串的形式传递，它们默认的 "协议" 是 "以空格符隔开文件名"。而这个项目里恰恰有一个文件的名字里面有空格符，所以导致了歧义的产生。该怪谁呢？既然 Linux 允许文件名里面有空格，那么用

户就有权使用这个功能。到头来因此出了问题，用户却被叫做菜鸟，为什么自己不小心，不看 man page。

后来我仔细看了一下 find 和 xargs 的 man page，发现其实它们的设计者其实已经意识到这个问题。所以 find 和 xargs 各有一个选项："-print0" 和"-0"。它们可以让 find 和 xargs 不用空格符，而用"NULL"（ASCII 字符 0）作为文件名的分隔符，这样就可以避免文件名里有空格导致的问题。可是，似乎每次遇到这样的问题总是过后方知。难道用户真的需要知道这么多，小心翼翼，才能有效的使用 UNIX 吗？

3.5 文本流不是可靠的接口

这些例子其实从不同的侧面显示了同一个本质的问题：用文本流来传递数据有严重的问题。是的，文本流是一个“通用”的接口，但是它却不是一个“可靠”或者“方便”的接口。UNIX 命令的工作原理基本是这样：

- 从标准输入得到文本流，处理，向标准输出打印文本流。
- 程序之间用管道进行通信，让文本流可以在程序间传递。

这其中主要有两个过程：

1. 程序向标准输出“打印”的时候，数据被转换成文本。这是一个编码过程。
2. 文本通过管道（或者文件）进入另一个程序，这个程序需要从文本里面提取它需要的信息。这是一个解码过程。

编码的貌似很简单，你只需要随便设计一个“语法”，比如“用空格隔开”，就能输出了。可是编码的设计远远不是想象的那么容易。要是编码格式没有设计好，解码的人就麻烦了，轻则需要正则表达式才能提取出文本里的信息，遇到复杂一点的编码（比如程序文本），就得用 parser。最严重的问题是，由于鼓励使用文本流，很多程序员很随意的设计他们的编码方式而不经过严密思考。这就造成了 UNIX 的几乎每个程序都有各自不同的输出格式，使得解码成为非常头痛的问题，经常出现歧义和混淆。

上面 find/xargs 的问题就是因为 find 编码的分隔符（空格）和文件名里可能存在的空格相混淆——此空格非彼空格也。而之前的 ls 和 rm 的问题就是因为 shell 把文件名和选项都“编码”为“字符串”，所以 ls 程序无法通过解码来辨别它们的到底是文件名还是选项——此字符串非彼字符串也！

如果你使用过 Java 或者函数式语言（Haskell 或者 ML），你可能会了解一些类型理论 (type theory)。在类型理论里，数据的类型是多样的，Integer, String, Boolean, List, record ……程序之间传递的所谓“数据”，只不过就是这些类型的数据结构。然而按照 UNIX 的设计，所有的类型都得被转化成 String 之后在程序间传递。这样带来一个问题：由于无

结构的 String 没有足够的表达力来区分其它的数据类型，所以经常会出现歧义。相比之下，如果用 Haskell 来表示命令行参数，它应该是这样：

```
1 data Parameter = Option String | File String | ...
```

虽然两种东西的实质都是 String，但是 Haskell 会给它们加上“标签”以区分 Option 还是 File。这样当 ls 接收到参数列表的时候，它就从标签判断哪个是选项，哪个是参数，而不是通过字符串的内容来瞎猜。

3.6 文本流带来太多的问题

综上所述，文本流的问题在于，本来简单明了的信息，被编码成为文本流之后，就变得难以提取，甚至丢失。前面说的都是小问题，其实文本流的带来的严重问题很多，它甚至创造了整个的研究领域。文本流的思想影响了太多的设计。比如：

- 配置文件：几乎每一个都用不同的文本格式保存数据。想想吧：.bashrc, .Xdefaults, .screenrc, .fvwm, .emacs, .vimrc, /etc 目录下那系列！这样用户需要了解太多的格式，然而它们并没有什么本质区别。为了整理好这些文件，花费了大量的人力物力。
- 程序文本：程序被作为文本文件，所以我们才需要 parser。这导致了整个编译器领域花费大量人力物力研究 parsing。其实程序完全可以被作为 parse tree 直接存储，这样编译器可以直接读取 parse tree，不但节省编译时间，连 parser 都不用写。
- 数据库接口：程序与关系式数据库之间的交互使用含有 SQL 语句的字符串，由于字符串里的内容跟程序的类型之间并无关联，导致了这种程序非常难以调试。
- XML：设计的初衷就是解决数据编码的问题，然而不幸的是，它自己都难 parse。它跟 SQL 类似，与程序里的类型关联性很差。程序里的类型名字即使跟 XML 里面的定义有所偏差，编译器也不会报错。Android 程序经常出现的 “force close”，大部分时候是这个原因。与 XML 相关的一些东西，比如 XSLT, XQuery, XPath 等等，设计也非常糟糕。
- Web：JavaScript 经常被作为字符串插入到网页中。由于字符串可以被任意组合，这引起很多安全性问题。Web 安全研究，有些就是解决这类问题的。
- IDE 接口：很多编译器给编辑器和 IDE 提供的接口是基于文本的。编译器打印出出错的行号和信息，比如 “102:32 variable x undefined”，然后由编辑器和 IDE 从文本里面去提取这些信息，跳转到相应的位置。一旦编译器改变打印格式，这些编辑器和 IDE 就得修改。
- log 分析：有些公司调试程序的时候打印出文本 log 信息，然后专门请人写程序分析这种 log，从里面提取有用的信息，非常费时费力。

- 测试：很多人写 unit test 的时候，喜欢把数据结构通过 toString 等函数转化成字符串之后，与一个标准的字符串进行比较，导致这些测试在字符串格式改变之后失效而必须修改。

还有很多的例子，你只需要在你的身边去发现。

3.7 “人类可读”和“通用”接口

当我提到文本流做接口的各种弊端时，经常有人会指出，虽然文本流不可靠又麻烦，但是它比其它接口更通用，因为它是唯一人类可读 (human-readable) 的格式，任何编辑器都可以直接看到文本流的内容，而其它格式都不是这样的。对于这一点我想说的是：

- 什么叫做“人类可读”？文本流真的就是那么的可读吗？几年前，普通的文本编辑器遇到中文的时候经常乱码，要折腾好一阵子才能让它们支持中文。幸好经过全世界的合作，我们现在有了 Unicode。
- 现在要阅读 Unicode 的文件，你不但要有支持 Unicode 的编辑器/浏览器，你还得有能显示相应码段的字体。文本流达到“人类可读”真的不费力气？
- 除了文本流，其实还有很多人类可读的格式，比如 JPEG。它可比文本流“可读”和“通用”多了，连字体都用不着。

所以，文本流的根本就不是“人类可读”和“通用”的关键。真正的关键在于“标准化”。如果其它的数据类型被标准化，那么我们可以在任何编辑器，浏览器，终端里加入对它们的支持，完全达到人类和机器都可轻松读取，就像我们今天读取文本和 JPEG 一样。

3.8 解决方案

其实有一个简单的方式可以一劳永逸的解决所有这些问题：

1. 保留数据类型本来的结构。不用文本流来表示除文本以外的数据。
2. 用一个开放的，标准化的，可扩展的方式来表示所有数据类型。
3. 程序之间的数据传递和存储，就像程序内部的数据结构一样。

3.9 UNIX 命令行的本质

虽然文本流引起了这么多问题，但是 UNIX 还是不会消亡，因为毕竟有这么多的上层应用已经依赖于它，它几乎是整个 Internet 的顶梁柱。所以这篇文章对于当前状况的

一个实际意义，也许是可以帮助人们迅速的理解 UNIX 的命令行机制，并且鼓励程序员在新的应用中使用结构化的数据。

UNIX 命令虽然过于复杂而且功能冗余，但是如果你看透了它们的本质，就能轻而易举的学会它们的使用方法。简而言之，你可以用普通的编程思想来解释所有的 UNIX 命令：

1. 函数：每一个 UNIX 程序本质上是一个函数 (main)。
2. 参数：命令行参数就是这个函数的参数。所有的参数对于 C 语言来说都是字符串，但是经过 parse，它们可能有几种不同的类型：
   - 变量名：实际上文件名就是程序中的变量名，就像 x, y。而文件的本质就是程序里的一个对象。
   - 字符串：这是真正的程序中的字符串，就像`"hello world"`。
   - keyword argument：选项本质上就是`"keyword argument"`(kwarg)，类似 Python 或者 Common Lisp 里面那个对应的东西，短选项（看起来像`"-l"`, `"-c"` 等等），本质上就是 bool 类型的 kwarg。比如`"ls -l"` 以 Python 的语法就是 `ls(l=true)`。长选项本质就是 string 类型的 kwarg。比如`"ls --color=auto"` 以 Python 的语法就是 `ls(color=auto)`。
3. 返回值：由于 main 函数只能返回整数类型 (int)，我们只好把其它类型 (string, list, record, ...) 的返回值序列化为文本流，然后通过文件送给另一个程序。这里“文件”通指磁盘文件，管道等等。它们是文本流通过的信道。我已经提到过，文件的本质是程序里的一个对象。
4. 组合：所谓“管道”，不过是一种简单的函数组合 (composition)。比如`"A x | B"`，用函数来表示就是`"B(A(x))"`。但是注意，这里的计算过程，本质上是 lazy evaluation (类似 Haskell)。当 B “需要”数据的时候，A 才会读取更大部分的 x，并且计算出结果送给 B。并不是所有函数组合都可以用管道表示，比如，如何用管道表示`"C(B(x), A(y))"`? 所以函数组合是更加通用的机制。
5. 分支：如果需要把返回值送到两个不同的程序，你需要使用 tee。这相当于在程序里把结果存到一个临时变量，然后使用它两次。
6. 控制流：main 函数的返回值（int 型）被 shell 用来作为控制流。shell 可以根据 main 函数返回值来中断或者继续运行一个脚本。这就像 Java 的 exception。
7. shell: 各种 shell 语言的本质都是用来连接这些 main 函数的语言，而 shell 的本质其实是一个 REPL (read-eval-print-loop，类似 Lisp)。用程序语言的观点，shell 语言完全是多余的东西，我们其实可以在 REPL 里用跟应用程序一样的程序语言。Lisp 系统就是这样做的。

3.10 数据直接存储带来的可能性

由于存储的是结构化的数据，任何支持这种格式的工具都可以让用户直接操作这个数据结构。这会带来意想不到的好处。

1. 因为命令行操作的是结构化的参数，系统可以非常智能的按类型补全命令，让你完全不可能输入语法错误的命令。
2. 可以直接在命令行里插入显示图片之类的 “meta data”。
3. Drag&Drop 桌面上的对象到命令行里，然后执行。
4. 因为代码是以 parse tree 结构存储的，IDE 会很容易的扩展到支持所有的程序语言。
5. 你可以在看 email 的时候对其中的代码段进行 IDE 似的结构化编辑，甚至编译和执行。
6. 结构化的版本控制[1]和程序比较 (diff)。

还有很多很多，仅限于我们的想象力。

3.11 程序语言，操作系统，数据库三位一体

如果 main 函数可以接受多种类型的参数，并且可以有 keyword argument，它能返回一个或多个不同类型的对象作为返回值，而且如果这些对象可以被自动存储到一种特殊的“数据库”里，那么 shell，管道，命令行选项，甚至连文件系统都没有必要存在。我们甚至可以说，“操作系统”这个概念变得“透明”。因为这样一来，操作系统的本质不过是某种程序语言的“运行时系统”(runtime system)。这有点像 JVM 之于 Java。其实从本质上讲，UNIX 就是 C 语言的运行时系统。

如果我们再进一步，把与数据库的连接做成透明的，即用同一种程序语言来“隐性”(implicit) 的访问数据库，而不是像 SQL 之类的专用数据库语言，那么“数据库”这个概念也变得透明了。我们得到的会是一个非常简单，统一，方便，而且强大的系统。这个系统里面只有一种程序语言，程序员直接编写高级语言程序，用同样的语言从命令行执行它们，而且不用担心数据放在什么地方。这样可以大大的减小程序员工作的复杂度，让他们专注于问题本身，而不是系统的内部结构。

实际上，类似这样的系统在历史上早已存在过 (Lisp Machine, System/38, Oberon)，而且收到了不错的效果。但是由于某些原因（历史的，经济的，政治的，技术的），它们都消亡了。但是不得不说它们的这种方式比 UNIX 现有的方式优秀，所以何不学过来？我相信，随着程序语言和编译器技术发展，它们的这种简单而统一的设计理念，有一天会改变这个世界。

[1]参见：http://yinwang0.wordpress.com/2012/02/12/structural-version-control/