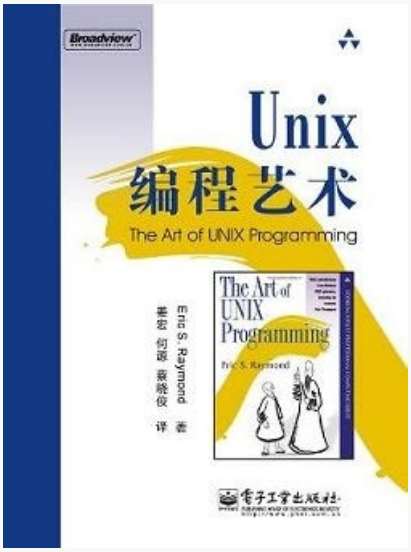
读《Unix编程艺术》



非常好的一本书，绝对必读。是从老廖那里蹭来零碎时间看的，没有做读书笔记。看完以后印象最深的一个词语就是：明文！

以后有机会一定要重读，并做笔记。

闲言碎语：

写游戏客户端也算是三年了，但客户端我会的也就是整几个界面之类的，实在不算一个称职的客户端程序。对框架和设计之类的通用技术倒是研究了不少，歪门邪道。

整天在Win下用C++整东西，还老是嘴硬说自己在C#程序员。到现在C#人家也要4.0了，咱还停留在“当年”这个浪漫的自我陶醉之中。

好好的Win不做，来看什么Unix xxx的，莫非想当叛徒？Take it easy，其实我只是想把自己的空虚用漫无目的的busy填充起来罢了。

别心急，但也别太缓了。没有目标导向就会迷失，过度的目标导向就会急功近利。

多读书，读好书！

以前读过一次，受益匪浅。项目买了后，再把关键的部分读了一遍，顺便做了书摘，以供反思咀嚼。

不喜欢的地方：

- 过度的推崇Unix的同时超级的鄙视其他操作系统，尤其是Windows貌似颇有偏见
- 鄙视C++而推崇C和脚本粘合。我想语言可能是个问题，但还是要冷静
- 推崇Emacs而鄙视其他编辑器，甚至vi。我想这个不至于，也没必要

喜欢的地方：

- 敢“大逆不道”的站出来讲什么是正确的

有缘拿到此书者，均值得一读。

译序

- 所见即所得的编辑器不如手写标记的纯文本更直接——90%的人会想：这怎么可能！

序

- 知识和专能差异巨大，凭借知识可以推断出该做什么，而专能让你甚至在无意之间，条件反射似的把事情做好
- 少一些技术，多一些共享文化； 显见好隐微的，直观和潜流的
- 不至于方法，更重乎理念

第1章 哲学

- 不懂Unix的人注定最终还要重复发明一个蹩脚的Unix
- 每过18个月，就有一半的知识会过时
- 机制，而不是策略。策略相对短寿，而机制才会长存
- 行为的最终逻辑被尽可能推后到使用端
- 最终用户永远比软件设计人员更清楚他们需要什么
- 用错误的方法解决正确的问题总比要用正确的方法解决错误的问题好
- 趣味性是一个峰值效率的标志。充满痛苦的开发环境只会浪费劳动力和创造力，这样的环境会在无形之中耗费大量时间、资金，还有机会
- 那些毫无动力、松松垮垮而且薪水微薄的程序员们，能在短短期限内，如同神灵附体般创造出稳定而新颖的软件——这只不过是经理人永远的梦呓罢了
- 让每个程序就做好一件事。如果有新任务，就重新开始，不要往原程序中加入新功能而搞的复杂
- 假定每个程序的输出都会成为另一个程序的输入，哪怕那个程序还是未知的。输出中不要有无关的信息干扰。避免使用严格的分栏格式和二进制格式输入。不要坚持使用交互式输入
- 尽可能早的将设计和编译的软件投入使用。对拙劣的代码别犹豫，扔掉重写
- 优先使用工具而不是拙劣的帮助来减轻编程的负担。工欲善其事，必先利其器
- 一个程序只做一件事，并做好。程序要能协作。程序要能处理文本流，因为这是最通用的接口
- 你无法断定程序会在什么地方耗费运行时间。瓶颈经常出现在想不到的地方，所以别急于胡乱找个地方改代码，除非你已经证实那儿就是瓶颈所在
- 在你没对代码进行剖析，特别是没找到最耗费时间的那部分之前，别去优化速度

- 花哨的算法在n很小时通常很慢，而n通常很小。花哨算法的常数复杂度很大。除非你确定n总是很大，否则不要用花哨算法
- 花哨的算法比简单的算法更容易出bug、更难实现。尽量使用简单的算法配合简单的数据结构
- 数据压倒一切。如果已经选择了正确的数据结构并且把一切都组织得井井有条，正确的算法也就不言自明。编程的核心是数据结构，而不是算法。
- 拿不准就穷举
- 模块原则：使用简洁的接口拼合简单的部件
- 清晰原则：清晰胜于技巧
- 组合原则：设计时考虑拼接组合
- 分离原则：策略同机制分离，接口同引擎分离
- 简洁原则：设计要简洁，复杂度能低则低
- 吝啬原则：除非却无他法，不要编写庞大的程序
- 透明性原则：设计要可见，以便审查和调试
- 健壮原则：健壮源于透明与简洁
- 表示原则：把知识叠入数据以求逻辑质朴而健壮
- 通俗原则：接口设计避免标新立异
- 缄默原则：如果一个程序没什么好说的，就沉默
- 补救原则：出现异常时，马上退出并给出足够错误信息
- 经济原则：宁花机器一分钟，不花程序员一秒
- 生成原则：避免手工hack，尽量编写程序去生成程序
- 优化原则：雕琢前先要有原型，跑之前先学会走
- 多样原则：绝不相信所谓“不二法门”的断言
- 扩展原则：设计着眼未来，未来总比预想来的快
- 计算机编程的本质就是控制复杂度
- 要编制复杂软件而又不至于一败涂地的方法就是降低其复杂度——用清晰的接口把若干简单的模块组合成一个复杂软件
- 把策略和机制揉成一个团有两个负面影响：会使策略变得死板，难以适应用户需求的改变；意味着任何策略的改变都有可能动摇机制
- 程序员的设计能力一般都大大超出他们的实现和排错能力，结果便是代价高昂的废品
- 恶心循环开始了：比别人花哨的方法就是使自己更变得更花哨，很快，庞大臃肿变成了业界标准
- 以简洁为美，人人对庞大复杂的东西群起而攻之
- 调试通常会占用四分之三甚至更多的开发时间，所以一开始就多做点工作以减少日后调试的工作量会很划算
- 如果不能理解一个程序的逻辑，就不能确信其是否正确，也就不能在出错的时候修复它
- 保证软件健壮性的一个相当重要的策略就是避免在代码中出现特例
- 在设计中，你应该主动将代码的复杂度转移到数据之中去
- 最好让不同的事物有明显区别，而不要看起来几乎一模一样
- 要么响亮的倒塌，要么为工作链下一环的程序输出一个严谨干净正确的数据
- 教会机器如何做更多层次的变成工作
- 人类很不善于干辛苦的细节工作，程序中的任何手工hacking都是滋生错误和延误的温床
- 还不知道瓶颈所在就匆忙进行优化，这可能是唯一一个比乱加功能更损害设计的错误
- 先求运行，再求正确，最后求快
- 设计一个僵化、封闭、不愿与外界沟通的软件，简直就是病态的傲慢
- 设计协议或是文件格式时，应使其具有充分的自描述性一遍可以扩展
- 只要可行，一切都因该做成与来源和目标无关的过滤器
- 数据流应尽可能文本化（这样可以使用标准工具来查看和过滤）
- 数据库部署和应用协议应尽可能文本化（让人可以阅读和编辑）
- 复杂的前端（用户界面）和后端应该泾渭分明
- 如何可能，在用C编写前，先用解释性语言搭建模型
- 当且仅当只用一门语言会提高程序复杂度时，混合语言编程才比单一语言编程来得好
- 宽收严发（对接收的东西要包容，对输出的东西要严格）
- 过滤时，不需要丢弃的信息绝不丢
- 小就是美，在确保完成任务的基础上，程序功能尽可能少
- 如果不能确定什么是对的，那么就只做最少量的工作，确保任务完成就行，至少知道明白什么是对的
- 你就应该不断追求卓越
- 软件设计是一门技艺，值得你付出所有的智慧、创造力和激情
- 你应该珍视你的时间绝不浪费
- 永远不要蛮干
- 善用工具，尽可能一切都自动化
- 软件设计是一门充满快乐的艺术，一种高水平的游戏

第2章 历史——双流记

- 忘记过去的人，注定要重蹈覆辙。前事不忘后事之师。
- 第二版效应——由于迫切希望把所有首次开发时遗漏的功能都添加进去，往往导致设计十分庞大、过于复杂
- 第三版效应——在第二版系统不堪自身重负而崩溃之后，有可能返璞归真，走上正道
- 好东西太多了，没有人希望把这些创新占为己有
- 外界的压力和纯粹出于对技艺的冲?荣誉感，促使人们在有了更多的初步思路后，去重写或抛开已有的大量代码
- 性能的局限不仅成就了经济性，而且鼓励了设计的简约

- 距开源越近就越繁荣
- 过度依赖任何一种技术或者商业模式都是错误的，保持软件及其设计的灵活性才是长存之道
- 如果我们日后不思悔改，可能还得大吃苦头
- 别和低价而灵活的方案较劲。低档的硬件只要数量足够，就能爬上性能曲线而最终获胜。

第3章 Unix哲学同其他哲学的比较

- 操作系统的设计，造就了该系统下软件开发的大大小小、方方面面的风格
- 二进制数据文件只有通过专用工具才能访问。开发者的思维会以工具而非以数据为中心，这样不同版本的文件格式很难兼容
- 客户端操作系统更关注用户体验，而不是7*24小时的连续正常运行
- 要打的字越多，愿意用的人就越少

第4章 模块性：保持清晰，保持简洁

- 软件设计有两种方式：一种是设计得极为简洁，没有看得到的缺陷；另一种是设计的极为复杂，有缺陷也看不出来
- 模块化：要编写复杂软件而又不至于一败涂地的唯一方法，就是用定义清晰的接口把若干简单模块组合起来。如此一来，多数问题只会出现在局部，那么还有希望对局部进行改进或优化，而不至于牵动全身。
- 封装良好的模块不会过多向外部披露自身的细节，不会直接调用其他模块的实现码，也不会胡乱共享全局数据。模块之间通过良好定义的API来通信。
- API在模块间扮演双重角色。在实现层面，作为模块之间的check point，阻止各自的内部细节被相邻模块知晓；在设计层面，正是API真正定义了整个体系
- 设计良好的API都能用纯人类语言描述清楚
- 软件系统应设计成层次分明的嵌套模块组成，而且每个层面上的模块粒度应降至最低
- 模块过小时，几乎所有复杂度都在于接口，想要理解任何一部分代码之前必须理解全部代码，因此阅读代码非常困难
- 最佳代码逻辑行数建议为200~400，物理行数建议为400~800
- 紧凑性就是一个设计是否能装进人脑中的特性。有经验的用户通常不需要操作手册。
- C++的设计者承认，他根本不指望有哪个程序员能够完全理解C++
- 合理对紧凑性，设计中要尽量考虑，绝不随意抛弃
- 在纯粹的正交设计中，任何操作均无副作用
- 不要重复自身：任何一个知识点在系统内都应有一个唯一、明确、权威的表述。真理的单点性。
- 解决一个定义明确的问题
- 限制不仅提倡了经济型，而且某种程度上提倡了设计的优雅
- 自顶向下的设计者通常先考虑程序的主事件循环，以后才插入具体的事件；自底向上的设计者通常先考虑封装具体的任务，以后再按照某种相关次序把这些东西粘合在一起
- 胶合层必须尽可能的薄。胶合层用来将东西粘在一起，但不应该用来隐藏各层的裂痕和不平整
- 完美之道，不在无可增加，而在无可删减
- 所有OO语言都显示出某种使程序陷入过度分层陷阱的倾向
- 如果你知道自己在做什么，三层就够了，如果你不知道自己在做什么，十七层也没用

第5章 文本化：好协议产生好实践

- 文本流的限制帮助了强化封装：因为文本流不鼓励丰富内容、编码结构密集的复杂表达式，也不提倡程序互相干涉内部状态
- 当你很想设计一个复杂的二进制文件格式，或一个复杂的二进制应用程序协议时，通常，明智的作法是躺下来等这种感觉过去。
- 文本格式的位密度未必一定比二进制格式低多少
- 当认为找到一种极端情况，有足够理由使用二进制文件格式或协议时，需仔细考虑扩展性，并在设计中为以后发展留出余地
- 选择XML可以简化问题，但也可能使问题复杂化。谨慎选择，牢记KISS原则
- 就做好一件事

第6章 透明性：来点儿光

- 美在计算科学中的地位，要比在其他任何技术中的地位都重要，因为软件是太复杂了。美是抵御复杂的终极武器
- 优雅是力量与简洁的结合。优雅的代码事半功倍；优雅的代码不仅正确，而且显然正确；优雅的代码不仅将算法传达给计算机，同时也把见解和信心传递给阅读代码的人。通过追求代码的优雅，我们可以编写更好的代码。
- 不要让调试工具仅仅成为一种事后追加或者用过之后就束之高阁的东西。它们是通往代码的窗口：不要只在墙上凿出粗糙的洞，要修正这些洞并装上窗。如果打算代码移植可被维护，就始终必须让光照进去
- 让UI沉默只做对了一半。真正的聪明是找到一个方法，可以访问具体细节，但又不让它们太显眼
- 优雅不是一种奢侈
- 要追求代码的透明，最有效的方法很简单，就是不要在具体操作的代码上叠放太多的抽象层
- 不要垒高台，要用设计简单而透明的算法和数据结构紧贴基面
- 宁愿抛弃、重建代码也不愿意修补那些蹩脚的代码

第7章 多道程序设计：分离进程为独立的功能

- 在开发出可以把全局复杂度降至最低程度的干净体系之前，关注性能问题便是过早优化——万恶之源
- 线程不是降低而是提高了全局复杂度，除非万不得已，尽量避免使用线程
- 这是一个普遍原则——人们总想挪用你写的任何工具，所以必须把程序设计成要么根本无法挪用，要么总是可以干净的挪用。这是你仅有的选择。
- 支持RPC的常见理由是它比文本流方法允许“更丰富”的接口——也就是说，接口可以具有更复杂、更专用的数据类型本体。但是想想简洁原则吧！RPC增加了，而不是降低了程序的全局复杂度
- 根本不要指望线程程序可移植

第8章 微型语言：寻找歌唱的乐符

- 程序员每百行代码出错率和所使用的语言在很大程度上无关。更高级的语言可以用更少的行数完成更多的任务，也意味着更少的bug
- 现代微型语言，要么就非常通用而不紧凑，要么就非常不通用而紧凑，不通用也不紧凑的语言完全没有竞争力
- C的宏处理器是有意设计的非常轻巧和简单，因为更强大的处理器可能会带来更糟糕的麻烦
- 宏产生的问题比宏解决的问题多

第9章 生成：提升规格说明的层次

- 仔细思考数据才是最好的行动，表达是编程的精髓
- 尽可能少干活；让数据塑造代码；依靠工具；把机制从策略中分离
- 建设性的惰性是大师级程序员的基本美德之一

第10章 配置：迈出正确的第一步

- 提高适应能力，除非这样做会产生超过0.7秒的延时。
- 人们几乎觉察不到少于0.7秒的启动延时；没来得及转移注意力，它就消失了
- 用户不应该看到优化开关，让程序经济运行是设计者的任务，不是用户的任务。
- 能用脚本包装器或简单管道实现的任务，就不要用配置开关实现
- 能简单利用其他程序来完成的任务，就不要增加本程序的复杂度

第11章 接口：Unix环境下的用户接口设计模式

- 新颖其实是进入门槛，给用户加上学习负担，所以，能少则少
- 新颖性提高了用户与接口最初几次的交互成本，但糟糕的设计永远使得接口令人痛苦而多余
- 如果可能，尽可能允许用户将接口功能委派给熟悉的程序来完成
- GUI所有的交互都需要人来驱动，不方便脚本化
- GUI在处理小数量物体简单行为的情况下，工作的很好，但是当行为或是物体的数目增加时，直接操作很快就变成了机械重复的苦差。用户降格成了装配工
- 对于复杂的计算，不仅仅需求步骤是正确的，而且其正确性要可见
- 无论你归属哪一方，学会倾听他人的声音都是明智的，而理解对立观点的前提无疑同样明智
- 过滤器模式的设计原则：宽进严出；在过滤时，不需要的信息也绝不丢弃；在过滤时，绝不增加无用数据
- MVC在实践中，V和C部分结合通常比两者同M部分的结合更为紧密。而往往只在应用程序需要模型的多重视图时才把他们分开
- 要促进脚本化和管线能力，最好就是尽可能的选择最简单的接口设计模式——牵扯环境因素最少、交互最少的模式
- 浏览器解开前端和后端耦合的方式具有重大意义
- GUI设计师的工作是方便用户，而不是在用户面前碍眼

第12章 优化

- 过早优化乃是万恶之源
- 最关键的是知道何时不去优化
- 最有效的优化往往是优化之外的其他事情，如：清晰干净的设计
- 程序员工具箱中最强大的优化技术就是不做优化
- 考虑到硬件和程序员时间成本比率，总是有更值得打发时间的事，别去优化一个工作中的系统
- 如果仅仅只是为了减少资源使用的一个常数部分而优化，那是很不值得的。线性性能增益往往很快就会被摩尔定律覆盖了
- 要通过剖析明确瓶颈所在，直觉实在是个糟糕的向导，即使特别熟悉可以代码的人也不例外
- 宏和内联函数耗费的时间都算在了调用函数的头上
- 保持代码短小简单，不要将核心数据机构和时间关键循环抛出缓存
- 通常指令加载要比执行花费的时间更长

- 许多优化方法都是暂时的而且常常随着成本比例而变化，唯一可去了解的方法就是衡量后再看
- 尽可能采用低时延的设计，减少协议的往返握手次数，忽略带宽成本，除非profile明确指出。带宽问题可以在后期采用一些技巧来解决，但要在已有设计中去除高延时则要困难得多（往往根本就不可能）
- 处理器周期是廉价的，除非在做计算密集的应用程序。更好的选择是短暂的启动和快速的响应
- 减少时延的策略：对可以共享启动开销的事务进行批处理；允许事务重叠；缓存
- 缓存的不一致性会经常引发bug。认为迫切需要缓存的时候，明智的做法是能够从更深层次来考虑，并稳稳为什么缓存是必须的。二进制缓存文件是一项不稳定的技法，应尽量避免

第13章 复杂度：尽可能简单

- 事情要尽可能简单，但别简单过了头
- 简单即美即雅即善，而复杂即丑即怪即恶
- 复杂就是成本。复杂所带来的成本，开发时便汹涌而来，部署后更变本加厉。复杂是bug滋生的温床，在整个软件按是生存期，世界都将不得安宁
- 复杂度——程序员为了能够试图理解一个程序，从而建立其思维模型并调试该程序的困难程度
- 不能为简单性而牺牲掉功能
- 处理各种复杂度，必然更依赖于见识而非方法
- 通过发现更简单的方法，可以去除偶然复杂度
- 依赖上下文环境判断哪些功能值得去做，可以去除选择复杂度
- 而要去除本质复杂度，只能通过对现实真谛的洞察和顿悟，从根本上重新定义所要解决的问题
- 计算资源以及人类的思考，同财富一样，不是靠储藏而是靠消费来证明其价值的
- 同其他美学一样，我们需要注意何时设计上的简约已经不再是有价值的自律形式，而开始成为一件伪装的苦行者外衣——一种实际上把美德作为借口来敷衍工作的纵容方式
- 添加新功能很容易根本考虑不到对整体设计的复杂度影响
- 一个十分不明显的背景，就像鱼没有注意到它游着的水一样，这就是框架的存在
- 选择需要管理的上下文环境，并且按照边界所允许的最小化方式构建程序
- 只有实证了其他方法行不通时才写庞大程序
- 框架是机制，尽可能少的包含策略
- 行程才是目的，顿悟在每日的实践中

第14章 语言：C还是非C

- 我语言的极限便是我世界的极限
- 急剧下降的硬件成本从根本上改变了编程的经济含义
- 过度强调节约机器资源已不再有任何意义，经济方面的最优选择已经变成尽可能减少调试时间，尽可能延长人类对代码的长期可维护性
- 至少30%~40%的开发时间都用于存储管理
- 混合语言是一种知识密集型（而不是编码密集型）的编程。要让它能够工作，我们不仅应该具备相当数量的多种语言应用知识，并且还必须具备能够判断这些语言在什么地方最合适、以及怎样把他们组合在一起的潜经验
- C语言是既要尽量接近裸机又要仍然保持稳定的最佳选择
- C语言最佳之处是资源效率和接近机器语言，而最糟糕的地方是其编程简直就是资源管理的炼狱
- C++试图满足所有人的需求，但代价是C++比任何一个程序员所能处理的复杂度都高
- C++：狗被钉上软肢而变成的章鱼
- C++的最佳之处是编译效率以及面向对象和泛型编程的结合，最糟之处是它非常怪异复杂，往往鼓励过分复杂的设计

第15章 工具：开发的战术

- 攀爬学习曲线的一次性付出，得到的是更有效编写程序的能力；精力也可以更多的放在设计层面而不是低层次的细节操作
- Lex生成的符号分析器在识别输入流中的低级模式相当快速，但是lex能够了解的正则表达式微型语言却不擅长统计事务，或是识别递归嵌套结构。为了能够解析这些，就需要yacc。
- 良好设计的程序代码生成器，应该永远不需要用户手动的改变甚至查看到自动生成的部分。把这些做好，是代码生成器的本分。
- 将时间花费在设计质量上，而不是底层的细节上，尽可能的自动化一切

第16章 重用：决不要重新发明轮子

- 不愿做不必要的工作是程序员的一大美德
- 每个新项目都从刀耕火种干起简直就是极端的浪费
- 应该耗费在解决新问题，而不是对那些已存在确切解决方案的问题老调重弹
- 养成良好的习惯，尝试通过最少的新发明，组合现有组件以形成原型，而非匆忙的编写独立的、只能使用一次的代码
- 源码可以用续，目标码则不行
- 阅读代码是为未来而投资，可以从中学到甚多——新技术、分解问题的新方法、不同的风格和手段。使用代码和学些代码都能得到有价值的回报。即使并不使用所研究代码中的方法，学习他人解决方案中的改良问题定义，也许能够帮助自己发现一个更好的方法
- 写之前先读：培养阅读代码的习惯

- 很少有什么彻底全新的问题，所以几乎总是能够发现非常接近的代码，成为自己需要的一个良好开端

第17章 可移植性：软件可移植性

- 以任何方式超出C的抽象机器模型而依赖于硬件的代码，都是不良形式
- 成为标准的最好办法就是发布一个高质量的开源实现
- 基于开放源码编程，便不会束手无策

第19章 开放源码：在Unix新社区中编程

- 处于准备期的发布应该同主干开发线分开，这样就不至于阻塞主线开发过程
- 精工细作，等一切都完美了才发布的想法是要不得的
- 抛弃秘密开发的习惯，转而支持过程的透明化和同行复审是炼金术成为化学学科最关键的一步
- 在移植层#ifdef和#if的使用是允许的（如果控制得当）。在此之外，应该尽量限制使用，只用于有条件的触发#include

第20章 未来：危机与机遇

- 预测未来最好的方法就是去创造未来
- 寡言的程序不会分散或浪费用户的注意力
- 主要的重心依然是模块化和清晰的代码——为严肃、高度可靠的计算和通信提供正确的基础设施
- 在那些巨大程序成为障碍之前，必须瘦身、重构、分解成块
- 更优秀解决方案的最危险敌人，就是一个现存的、足够优秀的代码库
- 我们能赢——只要我们想赢

附录D 无根的根：无名师的Unix心传

- 一行shell脚本胜过万行C程序
- 做一件事并做好，软件应当具有简单一致的行为，人和其他程序便都很容易想象其心理模型
- 现在得到的90%比等不来的100%更有价值
- 优化程序时不对热点进行反复衡量，就像渔夫把网撒入空湖中
- 尊者之智，就是了解自身之愚
- 以自然法则前进，在程序员手中，吸纳各种优良设计
- 家猫也能欺负老虎，但猫叫永远比不上虎吼

来源：〈http://dearymz.blog.163.com/blog/static/205657420108297232847 1/〉