二

39 Linux 架构优秀在哪里

我们在面试的时候经常会和面试官聊架构，多数同学可能会认为架构是一个玄学问题，讨论的是“玄而又玄”的知识——如同道德经般的开头“玄之又玄、众妙之门”。**其实架构领域也有通用的语言，有自己独有的词汇。**虽然架构师经常为了系统架构争得面红耳赤，但是即使发生争吵，大家也会遵守架构思想准则。

这些优秀的架构思想和准则，很大一部分来自早期的黑客们对程序语言编译器实现的探索、对操作系统实现方案的探索，以及对计算机网络应用发展的思考，并且一直沿用至今。比如现在的面向对象编程、函数式编程、子系统的拆分和组织，以及分层架构设计，依然沿用了早期的架构思路。

其中有一部分非常重要的思想，被著名的计算机科学家、Unix 代码贡献者 Douglas McIlroy 誉为 Unix 哲学，也是 Linux 遵循的设计思想。今天我就和你一起讨论下，这部分前人留下的思想精华，希望可以帮助到你日后的架构工作。

组合性设计（Composability）

Unix 系设计的哲学，都在和单体设计（Monolithic Design）和中心化唱反调。作为社区产品，开发者来自全世界不同的地方，这就构成了一个巨大的开发团队，自然会反对中心化。

而一个巨大的开发团队的管理，一定不能是 Mono 的。举个例子，如果代码仓库是Mono 的，这意味着所有的代码都存放在一个仓库里。如果要上线项目中的一个功能，那所有项目中的代码都要一起上线，只要一个小地方出了问题，就会影响到全局。在我们设计这个系统的时候，应该允许不同的程序模块通过不同的代码仓库发布。

再比如说，整体的系统架构应该是可以组合的。比如文件系统的设计，每个目录可以有不同的文件系统，我们可以随时替换文件系统、接入新的文件系统。比如接入一个网络的磁盘，或者接入一个内存文件系统。

与其所有的程序工具模块都由自己维护，不如将这项权利分发给需要的人，让更多的人参与进来。让更多的小团队去贡献代码，这样才可以把更多的工具体验做到极致。

这个思想在面向对象以及函数式编程的设计中，同样存在。比如在面向对象中，我们会尽量使用组合去替代继承。**因为继承是一种 Mono 的设计，一旦发生继承关系，就意味着父类和子类之间的强耦合。而组合是一种更轻量级的复用。**对于函数式编程，我们有 Monad 设计（单子），本质上是让事物（对象）和处理事物（计算）的函数之间可以进行组合，这样就可以最小粒度的复用函数。

同理，Unix 系操作系统用管道组合进程，也是在最小粒度的复用程序。

管道设计（Pipeline）

提到最小粒度的复用程序，就必然要提到管道（Pipeline）。Douglas McIlroy 在 Unix 的哲学中提到：**一个应用的输出，应该是另一个应用的输入。这句话，其实道出了计算的本质。**

计算其实就是将一个计算过程的输出给另一个计算过程作为输入。在构造流计算、管道运算、Monad 类型、泛型容器体系时——很大程度上，我们希望计算过程间拥有一定的相似性，比如泛型类型的统一。这样才可以把一个过程的输出给到另一个过程的输入。

重构和丢弃

在 Unix 设计当中有一个非常有趣的哲学。**就是希望每个应用都只做一件事情，并且把这件事情做到极致。如果当一个应用变得过于复杂的时候，就去重构这个应用，或者重新写一个应用。而不是在原有的应用上增加功能。**

上述逻辑和商业策略是否有相悖的地方？

关于这个问题，我觉得需要你自己进行思考，我不能给你答案，但欢迎把你的想法和答案写在留言区，我们一起交流。

设想一下，我们把微信的聊天工具、朋友圈、短视频、游戏都做成不同的应用，是不是会更好一些？

这是一个见仁见智的问题。但是目前来看，如果把短视频做成一个单独的应用，比如抖音，它在全球已经拥有 10 几亿的用户了；如果把游戏做成一个单独的应用，比如王者荣耀和 LoL，它们深受程序员们和广大上班族的喜爱。

还有，以我多年从事大型系统开发的经验来看，我宁愿重新做一些微服务，也不愿意去重构巨大的、复杂的系统。换句话说，我更乐意将新功能做到新系统里面，而不是在一个巨大的系统上不断地迭代和改进。这样不仅节省开发成本，还可以把事情做得更好。从这个角度看，我们进入微服务时代，是一个不可逆的过程。

另外多说一句，如果一定要在原有系统上增加功能，也应该多重构。**重构和重写原有的系统有很多的好处**，希望你不要有**畏难情绪**。优秀的团队，总是处在一个代码不断迭代的过程。一方面是因为业务在高速发展，旧代码往往承接不了新需求；另一方面，是因为程序员本身也在不断地追求更好的架构思路。

而重构旧代码，还经常可以看到业务逻辑中出问题的地方，看到潜在的隐患和风险，同时让程序员更加熟悉系统和业务逻辑。而且程序的复杂度，并不是随着需求量线性增长的。**当需求量超过一定的临界值，复杂度增长会变快，类似一条指数曲线。因此，控制复杂度也是软件工程的一个核心问题。**

写复杂的程序就是写错了

我们经常听到优秀的架构师说，**程序写复杂了，就是写错了。**在 Unix 哲学中，也提出过这样的说法：**写一个程序的时候，先用几周时间去构造一个简单的版本，如果发现复杂了，就重写它。**

确实实际情景也是如此。我们在写程序的时候，如果一开始没有用对工具、没有分对层、没有选对算法和数据结构、没有用对设计模式，那么写程序的时候，就很容易陷入大量的调试，还会出现很多 Bug。**优秀的程序往往是思考的过程很长，调试的时间很短，能够迅速地在短时间内完成测试和上线。**

所以当你发现一段代码，或者一段业务逻辑很消耗时间的时候，可能是你的思维方式出错了。想一想是不是少了必要的工具的封装，或者遗漏了什么中间环节。当然，也有可能是你的架构设计有问题，这就需要重新做架构了。

优先使用工具而不是“熟练”

关于优先使用工具这个哲学，我深有体会。

很多程序员在工作当中都忽略了去积累工具。比如说：

- 你经常要重新配置自己的开发环境，也不肯做一个 Docker 的镜像；
- 你经常要重新部署自己的测试环境，而且有时候还会出现使用者太多而不够用的情况。即使这样的情况屡屡发生，也不肯做一下容器化的管理；
- Git 的代码提交之后，不会触发自动化测试，需要人工去点鼠标，甚至需要由资深的测试手动去测。

很多程序员都认为自己对某项技术足够熟练了。因此，宁愿长年累月投入更多的时间，也不愿意主动跳脱出固化思维。宁愿不断使用某一项技术，而不愿意将重复劳动转化成工具。比

如写一个小型的 ORM 框架、缓存引擎、业务容器……总之，养成良好的习惯，可以让开发效率越来越高。

在 Unix 哲学当中，有这样一条规则：**有些人使用“熟练”而不是使用工具来减轻工作，即便是临时需要去构造一个工具，你也应该尽可能去尝试实现。**

我们现在每天都用的 Git 版本控制工具，就是基于这样的哲学被构建出来的。当时刚好是 Linux 内核研发团队的商业代码管理工具到期了，Linux 的缔造者们基于这个经验教训，就自主研发了 Git 这款工具，不仅顺利地推进了后续的研发工作，还做成了一个巨大的程序员交友生态。

再给你讲一个我身边的故事：我刚刚工作的时候，我的老板自己写了一个小程序，去判断 HR 发过来简历是否符合他的用人条件。所以他每天可以看完几百份简历，并筛选出面试人选。而那些没有利用工具的技术 Leader，每天都在埋怨简历太多看不过来。

这些故事告诉我们，**作为程序员，不仅仅需要完成工作，还要重视中间过程的工具缔造。**

其他优秀的原则

我在学习 Unix 哲学的过程中，还看到很多有趣的规则，这里我摘选了一些和你分享。

比如：**不要试图猜测程序可能的瓶颈在哪里，而是试图证明这个瓶颈，因为瓶颈会出现在出乎意料的地方。**这句话告诉我们，要多写性能测试程序并且构造压力测试的场景。只有这样，才能让你的程序更健壮，承载更大的压力。

再比如：**花哨的算法在业务规模小的时候通常运行得很慢，因此业务规模小的时候不要用花哨的算法。简单的算法，往往性能更高。如果你的业务规模很大，可以尝试去测试并证明需要用怎样的算法。**

这也是我们在架构程序的时候经常会出错的地方。我们习惯性地选择用脑海中记忆的时间复杂度最低的算法，但是却忽略了**时间复杂度只是一种增长关系，一个算法在某个场景中到底可不可行，是要以实际执行时收集数据为准的。**

再比如：**数据主导规则。当你的数据结构设计得足够好，那么你的计算方法就会深刻地反映出你系统的逻辑。这也叫作自证明代码。编程的核心是构造好的数据结构，而不是算法。**

尽管我们在学习的时候，算法和数据结构是一起学的。但是在大牛们看来，**数据结构的抽象可以深刻反映系统的本质。**比如抽象出文件描述符反应文件、抽象出页表反应内存、抽象出 Socket 反应连接——这些数据结构才是设计系统最核心的东西。

总结

最后，再和你分享一句 Unix 的设计者Ken Thompson 的经典语录：**搞不定就用蛮力。**这是打破所有规则的规则。**在我们开发的过程当中，首先要把事情搞定！只有把事情搞定，才有我们上面谈到的这一大堆哲学产生价值的可能性。**事情没有搞定，一切都尘归尘土归土，毫无意义。

今天所讲的这些哲学，可以作为你平时和架构师们沟通的语言。架构有自己领域的语言，比如设计模式、编程范式、数据结构，等等。还有许多像 Unix 哲学这样——经过历史积淀，充满着人文气息的行业标准和规范。

如果你想仔细看看当时 Unix 的设计者都总结了哪些哲学，可以阅读这篇文档。

上一页 下一页