

78 | 开源实战二（上）：从Unix开源开发学习应对大型复杂项目开发

王争 · 设计模式之美



软件开发的难度无外乎两点，一是技术难，意思是说，代码量不一定多，但要解决的问题比较难，需要用到一些比较深的技术解决方案或者算法，不是靠“堆人”就能搞定的，比如自动驾驶、图像识别、高性能消息队列等；二是复杂度，意思是说，技术不难，但项目很庞大，业务复杂，代码量多，参与开发的人多，比如物流系统、财务系统等。第一点涉及细分专业的领域知识，跟我们专栏要讲的设计、编码无关，所以我们重点来讲第二点，如何应对软件开发的复杂度。

简单的“hello world”程序，谁都能写得出来。几千行的代码谁都能维护得了。但是，当代码超过几万行、十几万，甚至几十万行、上百万行的时候，软件的复杂度就会呈指数级增长。这种情况下，我们不仅仅要求程序运行得了，运行得正确，还要求代码看得懂、维护得了。实际上，复杂度不仅仅体现在代码本身，还体现在协作研发上，如何管理庞大的团队，来进行有条不紊地协作开发，也是一个很复杂的难题。

如何应对复杂软件开发？Unix 开源项目就是一个值得学习的例子。

Unix 从 1969 年诞生，一直演进至今，代码量有几百万行，如此庞大的项目开发，能够如此完美的协作开发，并且长期维护，保持足够的代码质量，这里面有很多成功的经验可以借鉴。所以，接下来，我们就以 Unix 开源项目的开发为引子，分三节课的时间，通过下面三个话题，详细地讲讲应对复杂软件开发的方法论。希望这些经验能为你所用，在今后面对复杂项目开发的时候，能让你有条不紊、有章可循地从容应对。

从设计原则和思想的角度来看，如何应对庞大而复杂的项目开发？

从研发管理和开发技巧的角度来看，如何应对庞大而复杂的项目开发？

聚焦在 Code Review 上来看，如何通过 Code Review 保持项目的代码质量？

话不多说，让我们正式开始今天的学习吧！

封装与抽象

在 Unix、Linux 系统中，有一句经典的话，“Everything is a file”，翻译成中文就是“一切皆文件”。这句话的意思就是，在 Unix、Linux 系统中，很多东西都被抽象成“文件”这样一个概念，比如 Socket、驱动、硬盘、系统信息等。它们使用文件系统的路径作为统一的命名空间（namespace），使用统一的 read、write 标准函数来访问。

比如，我们要查看 CPU 的信息，在 Linux 系统中，我们只需要使用 Vim、Gedit 等编辑器或者 cat 命令，像打开其他文件一样，打开 /proc/cpuinfo，就能查看到相应的信息。除此之外，我们还可以通过查看 /proc/uptime 文件，了解系统运行了多久，查看 /proc/version 了解系统的内核版本等。

实际上，“一切皆文件”就体现了封装和抽象的设计思想。

封装了不同类型设备的访问细节，抽象为统一的文件访问方式，更高层的代码就能基于统一的访问方式，来访问底层不同类型的设备。这样做的好处是，隔离底层设备访问的复杂性。统一的访问方式能够简化上层代码的编写，并且代码更容易复用。

除此之外，抽象和封装还能有效控制代码复杂性的蔓延，将复杂性封装在局部代码中，隔离实现的易变性，提供简单、统一的访问接口，让其他模块来使用，其他模块基于抽象的接口而非

具体的实现编程，代码会更加稳定。

分层与模块化

前面我们也提到，模块化是构建复杂系统的常用手段。

对于像 Unix 这样的复杂系统，没有人能掌控所有的细节。之所以我们能开发出如此复杂的系统，并且能维护得了，最主要的原因就是将系统划分成各个独立的模块，比如进程调度、进程通信、内存管理、虚拟文件系统、网络接口等模块。不同的模块之间通过接口来进行通信，模块之间耦合很小，每个小的团队聚焦于一个独立的高内聚模块来开发，最终像搭积木一样，将各个模块组装起来，构建成一个超级复杂的系统。

除此之外，Unix、Linux 等大型系统之所以能做到几百、上千人有条不紊地协作开发，也归功于模块化做得好。不同的团队负责不同的模块开发，这样即便在不了解全部细节的情况下，管理者也能协调各个模块，让整个系统有效运转。

实际上，除了模块化之外，分层也是我们常用来架构复杂系统的方法。

我们常说，计算机领域的任何问题都可以通过增加一个间接的中间层来解决，这本身就体现了分层的重要性。比如，Unix 系统也是基于分层开发的，它可以大致上分为三层，分别是内核、系统调用、应用层。每一层都对上层封装实现细节，暴露抽象的接口来调用。而且，任意一层都可以被重新实现，不会影响到其他层的代码。

面对复杂系统的开发，我们要善于应用分层技术，把容易复用、跟具体业务关系不大的代码，尽量下沉到下层，把容易变动、跟具体业务强相关的代码，尽量上移到上层。

基于接口通信

刚刚我们讲了分层、模块化，那不同的层之间、不同的模块之间，是如何通信的呢？一般来讲都是通过接口调用。在设计模块（module）或者层（layer）要暴露的接口的时候，我们要学会隐藏实现，接口从命名到定义都要抽象一些，尽量少涉及具体的实现细节。

比如，Unix 系统提供的 `open()` 文件操作函数，底层实现非常复杂，涉及权限控制、并发控制、物理存储，但我们用起来却非常简单。除此之外，因为 `open()` 函数基于抽象而非具体的实现来定义，所以我们在改动 `open()` 函数的底层实现的时候，并不需要改动依赖它的上层代码。

高内聚、松耦合

高内聚、松耦合是一个比较通用的设计思想，内聚性好、耦合少的代码，能让我们在修改或者阅读代码的时候，聚集到在一个小范围的模块或者类中，不需要了解太多其他模块或类的代码，让我们的焦点不至于太发散，也就降低了阅读和修改代码的难度。而且，因为依赖关系简单，耦合小，修改代码不会牵一发而动全身，代码改动比较集中，引入 bug 的风险也就减少了很多。

实际上，刚刚讲到的很多方法，比如封装、抽象、分层、模块化、基于接口通信，都能有效地实现代码的高内聚、松耦合。反过来，代码的高内聚、松耦合，也就意味着，抽象、封装做到比较到位、代码结构清晰、分层和模块化合理、依赖关系简单，那代码整体的质量就不会太差。即便某个具体的类或者模块设计得不怎么合理，代码质量不怎么高，影响的范围也是非常有限的。我们可以聚焦于这个模块或者类做相应的小型重构。而相对于代码结构的调整，这种改动范围比较集中的小型重构的难度就小多了。

为扩展而设计

越是复杂项目，越要在前期设计上多花点时间。提前思考项目中未来可能会有哪些功能需要扩展，提前预留好扩展点，以便在未来需求变更的时候，在不改动代码整体结构的情况下，轻松地添加新功能。

做到代码可扩展，需要代码满足开闭原则。特别是像 Unix 这样的开源项目，有 n 多人参与开发，任何人都可以提交代码到代码库中。代码满足开闭原则，基于扩展而非修改来添加新功能，最小化、集中化代码改动，避免新代码影响到老代码，降低引入 bug 的风险。

除了满足开闭原则，做到代码可扩展，我们前面也提到很多方法，比如封装和抽象，基于接口编程等。识别出代码可变部分和不可变部分，将可变部分封装起来，隔离变化，提供抽象化的

不可变接口，供上层系统使用。当具体的实现发生变化的时候，我们只需要基于相同的抽象接口，扩展一个新的实现，替换掉老的实现即可，上游系统的代码几乎不需要修改。

KISS 首要原则

简单清晰、可读性好，是任何大型软件开发要遵循的首要原则。只要可读性好，即便扩展性不好，顶多就是多花点时间、多改动几行代码的事情。但是，如果可读性不好，连看都看不懂，那就不是多花时间可以解决得了的了。如果你对现有代码的逻辑似懂非懂，抱着尝试的心态去修改代码，引入 bug 的可能性就会很大。

不管是自己还是团队，在参与大型项目开发的时候，要尽量避免过度设计、过早优化，在扩展性和可读性有冲突的时候，或者在两者之间权衡，模棱两可的时候，应该选择遵循 KISS 原则，首选可读性。

最小惊奇原则

《Unix 编程艺术》一书中提到一个 Unix 的经典设计原则，叫“最小惊奇原则”，英文是“The Least Surprise Principle”。实际上，这个原则等同于“遵守开发规范”，意思是，在做设计或者编码的时候要遵守统一的开发规范，避免反直觉的设计。实际上，关于这一点，我们在前面的编码规范部分也讲到过。

遵从统一的编码规范，所有的代码都像一个人写出来的，能有效地减少阅读干扰。在大型软件开发中，参与开发的人员很多，如果每个人都按照自己的编码习惯来写代码，那整个项目的代码风格就会千奇百怪，这个类是这种编码风格，另一个类又是另外一种风格。在阅读的时候，我们要不停地切换去适应不同的编码风格，可读性就变差了。所以，对于大型项目的开发来说，我们要特别重视遵守统一的开发规范。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天，我们主要从设计原则和思想的角度，也可以说是从设计开发的角度，来学习如何应对复杂软件开发。我总计了 7 点我认为比较重要的。这 7 点前面我们都详细讲过，如果你对哪块理解得不够清楚，可以回过头去再看下。这 7 点分别是：

封装与抽象

分层与模块化

基于接口通信

高内聚、松耦合

为扩展而设计

KISS 首要原则

最小惊奇原则

当然，这 7 点之间并不是相互独立的，有几点是互相支持的，比如“高内聚、松耦合”与抽象封装、分层模块化、基于接口通信。有几点是互相冲突的，比如 KISS 原则与为扩展而设计，这都需要我们根据实际情况去权衡。

课堂讨论

从设计原则和思想的角度来看，你觉得哪些原则或思想在大型软件开发中最能发挥作用，最有效地应对代码的复杂性？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

Unix开源项目是一个值得学习的例子，它从1969年诞生至今，代码量庞大，却能完美协作开发并长期维护，保持足够的代码质量。本文从设计原则和思想、研发管理和开发技巧、以及Code Review三个角度详细讲解了应对复杂软件开发的方法论。文章首先强调了封装与抽象的重要性，Unix系统中的“一切皆文件”设计思想体现了封装和抽象，有效隔离底层设备访问的复杂性，简化了上层代码的编写。其次，文章提到了分层与模块化的重要性，Unix系统将系统划分成各个独立的模块，不同的模块之间通过接口进行通信，实现了高内聚、松耦合的代码结构。最后，基于接口通信和设计高内聚、松耦合的代码也是文章强调的重点。这些方法不仅能有效应对复杂软件开发，还能提高代码质量和可维护性。

文章还提到了一些重要的设计原则和思想，如为扩展而设计、KISS首要原则和最小惊奇原则。为扩展而设计强调在前期设计上多花点时间，提前思考项目中未来可能会有哪些功能需要扩展，以便在未来需求变更的时候，在不改动代码整体结构的情况下，轻松地添加新功能。KISS首要原则强调简单清晰、可读性好是任何大型软件开发要遵循的首要原则。最小惊奇原则则强调遵守统一的开发规范，避免反直觉的设计。这些原则和思想在大型软件开发中能有效地应对代码的复杂性，提高代码质量和可维护性。

总的来说，本文通过详细讲解Unix开源项目的设计原则和思想，以及一些重要的设计原则和思想，为读者提供了应对复杂软件开发的方法论，帮助读者更好地理解和应用这些方法，提高软件开发的效率和质量。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (37)

最新 精选



Jxin

2020-05-03

1.不用哪些，只要一个，就是合理的分层。

2.大型软件的持续开发，。人多，代码量大，时间长会有这三个问题。

3.人多：人一多什么鸟都有，在快节奏下，你很难去保证所有人的所有代码质量。即便你有code review，但质量是要对业务做让步的，而这是合理的。那么这时候去要求每个人的，编码规范，抽象封装能力，就非常的难。所以这些能力对软件质量很重要，但你抓不了也是白搭。反观分层，它其实是限定了一块业务逻辑，实现代码的基本拆分和归类，定义了一个基本的规范。任何人都可以顺着这个规范去阅读他人的代码。实现了最基本的复杂性隔离。可执行可落地，试用期基本就可以灌输成功。

4.代码量大：对于代码量大的项目，要找到目标功能，是很痛苦的。而分层在这时候就具备类似索引的功能。哪怕项目没注释，只要它按着分层写，你就可以顺着 业务 功能 细节这样的线去摸到目标功能，无需从入口开始读代码。

5.时间长：软件在时间线上是动态的，当下的业务边界很可能因时间的推移而被变革，需要重组模块的数据范围和业务边界。好的分层可以让你更快的重组模块，解决当前模块划分不合理的问题。具体可以看下ddd的分层。它可以让你在重组模块时，只需花一两个小时，剪切粘帖整块聚合的业务代码，并调整一些基础功能的实现，便能实现模块重组。而不需长达数个月的风险评估，代码调整，测试覆盖。

共 2 条评论 >

👍 70



javaadu

2020-05-05

年前做的一个项目，是一个能力编排引擎，这是我实际参与的第一个具备良好设计的软件项目，满足了：抽象和封装、模块化和分层结构、基于接口而非实现编程等设计原则，在这个项目中我才真正获得了对这些设计原则的理解。这个经历说明—我应该尽量去高水平高素质的团队，才有机会遇到高水平的项目和代码

共 2 条评论 >

👍 41



下雨天

2020-05-05

分层和模块化，基于接口通讯，这两点最重要！

这个相当于整个架构搭起来了，每个模块怎么划分怎么交流定好了，其他扩展行，可读性，抽象都可以细化到模块中实施。



👍 14



jaryoung

2020-05-02

个人觉得是：高内聚、松耦合，高内聚说明合适的人都在一起了，松耦合说明不合适的人都隔离起来。

共 1 条评论 >

👍 9



Frank

2020-05-01

我觉得在大型项目开发中，单一职责和最小知识原则也是发挥很大的作用，从编码角度来看，类，模块都遵守单一和最小知识原则，这样的话内聚性高，耦合少，每个类和模块可能都不会太复杂，可读性，可测试性也就不会太差。从一个系统的生命周期来看，单一职责体现为产品，开发，测试，运维。各个角色的人各司其职，耦合不会太多，能有效的提高效率。会想起以前在某家传统公司，需求沟通、设计、编码、测试、维护几乎要自己一个人干，有时候觉得太累。



👍 8



辣么大

2020-05-01

五一快乐！

我觉得抽象封装和分层模块化最能发挥作用。最近在看ROS机器人操作系统，是开源一个中间件系统，思想是通过封装，抽象，使得不懂硬件的程序员可以对机器人进行编程。里面所有的可执行程序，都可以叫做一个node（节点），机器人可以组装的（移动底盘，机器臂等）这个是模块化，机械臂控制使用moveit运动学控制规划模块，底座导航使用导航功能模块，

这个算是模块化。机器的各个部分，都使用命名空间的方式访问，和争哥将的linux系统结构的方式差不多。

共 3 条评论 >

👍 5



落尘kira

2020-05-13

我觉得是最小惊奇原则，可读性一定要是第一位的（不管代码写的有多惊奇，起码得让后面的同学看懂，多写一行注释也好）



👍 4



xk_

2020-05-09

单一职责原则和KISS原则，其他原则太复杂，就记得这两个。



👍 4



,

2020-05-07

如何定义复杂度: 对软件做一些修改,所需的人力物力较少,那么我们就可以说他复杂度低,反之则认定它复杂度较高

分层和模块化应对的是架构层面的复杂度,影响的是整个软件的质量,他应该是软件开发中最重要的部分,一旦发生改动,需要多个模块进行修改,将近于重新做架构设计,成本非常大

实例: TCP/IP网络模型,应用层,传输层,网络传输层,网络层的划分是水平方向的划分,应用服务器,负载均衡服务器,DNS服务器,个人电脑,体现的是垂直方向的划分,这种划分方式能够很好的应对复杂度

基于接口通讯是应对的是模块通讯时的复杂度,一次改动,通常会影响到多个模块,需要多个模块的开发者协作才能完成改动,但毕竟只需要修改模块之间的通讯,成本相对较小,所以较为次要

KISS,为扩展设计,最小惊奇应对的是代码层面的复杂度,属于实现细节,改动通常不会被其他模块感知,改动需要的人力物力较上两者更少,所以他最为次要



👍 2



J

2020-05-05

封装与抽象、分层与模块化、基于接口通信，我觉得是最重要的三个设计原则。

封装与抽象是从使用者的角度来考虑系统该如何设计。

分层与模块化则是在系统建设者之间划分好界限和职责。
基于接口通信构建了内外之间最合适的交互方式。



2