什么是现实理想主义者

曾经有人看了我的文章，以为我是一个“理想主义者”，来找我聊天。他说：“你知道吗，我跟你一样喜欢简单优雅的代码。上次我在某公司工作，看到他们的代码乱得不成样子，二话没说给他们重写了，结果有几个小地方跟原来的代码不大一样，后来系统因此当掉了。老板对我说，明天你不用再来上班了！你说我是不是好心没好报啊？”

虽然我同情他丢了工作，然而我并不认同这种不经同意就推翻重写别人代码的作法。实际上我曾经跟一个老喜欢重写别人代码的人合作，后来整个团队（包括我）都差点被他给弄疯了。所以我对他说：“你不可以这样改别人的代码的！如果我是你老板，可能不会开掉你，却也会给你一个严重警告的。”

从我们的对话你也许已经发现了，我并不是一个通常人所谓的“理想主义者”。虽然我有很多新颖而美好的想法，然而它们全都深深植根于现实中。我反对不以现实为基础的“理想”，实际上那不叫理想，而只能叫做“空想”。我的直觉和理性会很快的告诉我，哪些事情是可能的，哪些是不大可能的。我往往在早期就能察觉和避免那些最终会失败的“理想主义作法”。

从我对各种“新语言”，“新理论”和“新技术”的看法，你也许已经发现了我的这个特点，我不再是十年前那个“热爱新奇事物”的王垠。不理解的人甚至会觉得我“守旧”，然而我只是通过理性分析，预见了某些“新技术”的失败。在我的心里，事物和技术并没有新旧之分，只有合理与不合理的差别。

如何对待别人的代码

那么我是如何对待别人的“垃圾代码”的呢？你也许会很惊讶我的做法：我尽量不动它们！

虽然我喜欢简单优雅的代码，然而对于别人写的代码，就算它再丑再乱，我也不会乱动它。我就像一个外科专家，多次对已有代码进行“换心手术”。这种手术成功的要诀，是制造尽量小的“切口”，刚好可以换掉心脏，而不动其他部位。就算那些地方血管乱绕，堆满各种垃圾，也不要去动它们。

这是为什呢？因为代码首要的目标应该是“解决问题”（包括“没有 bug”），其次的目标才是“简单优雅”。如果不能解决问题，再优雅又有什么用呢，只不过是玩具而已。对于已经可以解决问题的代码，就算它再乱再复杂，我也是高度尊重的，绝对不敢像这个朋友一样，不假思索就删掉重写。这就像你给别人做换心手术，看到大腿上有些血管是乱的，就把大腿也切开倒腾，你的病人不死才怪呢。

我自己写代码的时候，“解决问题”和“简单优雅”往往是紧密结合，交织在一起的。如果我写不出简单优雅的代码，我就不能又快又正确的解决问题。所以我的代码往往从一开头就是简单优雅，模块化的。我从很小的函数开始写起，每个函数只解决很小的问题，最终我把它们组合在一起，解决掉整个问题。

对于别人的代码，情况就很不一样了。很多人写的代码很乱，很复杂，不易理解，看得我头痛，但由于他们在上面花了很多的时间，而且这些代码经过了很长时间的使用，大量现实情况的考验，所以它们已经算是解决了问题。对于这样的代码，我的经验是这样：如果把它删掉完全重写，是很难不犯原作者已经犯过的错误的。就算你自认为水平世界一流，写的代码极其简单和优雅，也不能避免犯错。

这不是一个智力的问题，而是一个智慧的问题。喜欢删掉别人代码重写的人，也许有很高的智力，却缺乏智慧。代码是用来解决现实问题的，而现实有许许多多的细节，代码需要覆盖现实世界各种不完美的地方。这些不完美也许来自库代码，也许来自操作系统，也许来自网络协议，也许来自用户习惯，也许来自自然界。我们必须承认，很多这些东西我们是没有能力，没有时间，也没有必要去改变的。

别人已经写好，用了几年的代码，很有可能已经遇到各种现实问题，各种边角情况，原来的作者虽然不像你一样思路清晰，却也为此付出了时间和精力。这些复杂混乱的代码逻辑里面，已经针对现实世界的不完美，做出了基本可行的解决方案。一个有智慧的人，必须能利用这些前人留下来的混乱代码，因为它包含了时间积累下来的财富。

那么我一般是如何利用别人遗留下来的代码的呢？我的策略包含好几个要点。

首先，我尽量保持别人的代码原封不动。因为别人的代码解决的问题，很可能不是我当前需要解决的问题。因为看不顺眼而去改别人的代码，不但分散自己的精力，而且有可能制造新的 bug，导致新老代码中同时多处出现 bug，难以追踪和修复。为了保持别人的代码原封不动，却又让自己写的新代码简单优雅，我必须理解原有代码的接口（interface），以及它原有的各种特征，我力求保持它们不变。这就像外科大夫做换心手术，他必须保证已有的血管都连接到正确的地方。

我喜欢把自己的代码做成一个可替换的，模块化的元件，可以随时在系统里插入或者移除。一旦发现出了问题，我可以随时切换到原来的代码，重新测试，这样我就可以知道问题出在原来的代码，还是出在我的新代码里面。另外，我还会注意避免对已有函数进行换名，这样我可以把自己的修改局限在一个或者少数几个文件里面，避免 Git 的历史里面出现不必要的，让人分心的修改。就算要换名也应该单独作为 commit，而不应该跟逻辑的修改混在一起。

如果经过多次试验，我发现别人的代码的确需要改，不然我没法继续写新的代码，那么我只好对它进行修改。由于已有的代码复杂混乱，我一般会极其小心的对待它。我不会删掉大片的代码，从头开始写，那几乎注定是要失败的。通常我会先“隔离”出很小的一块代码，对它进行重写。随之立即进行大量的测试和试验，找原作者来帮我

检查是否有问题，如此反复……

那么这块改掉的代码需要小到什么程度呢？我也许就只改写一个 for 循环，把几行代码提出去做成帮助函数，简化一个表达式，把一个类成员变成一个局部变量，改几个局部变量的名字之类的。你可以参考我在《编程的智慧》里提到的各种改进代码的方式。每一个这样的小改动都有可能出错，所以在此之后必须进行严格的验证，确保修改后的代码和原来的代码语义相同。这样反反复复很多次之后，你才能正确的替换掉原来的代码。

从我对待别人代码的方式，你也许已经发现了，我不是一个通常意义上的理想主义者。我不会为了自己简单优雅的理想，而完全推翻重写别人的代码，因为我知道现实世界的复杂性，我知道这样做注定是要失败的。我对待别人代码的态度，是深深地植根于现实的。通过极其严密的措施，我确保改进后的代码跟原来的代码语义完全相同，尽最大可能避免重复前人的错误，避免制造新的 bug。

由于我的理想植根于现实，我把自己称为“现实理想主义者”（practical idealist），而不是“理想主义者”（idealist）。我曾经跟纯粹的理想主义者共事，这种人总是嫌别人的代码丑，不经商量就大幅度的删除重写大量代码，结果给团队的开发带来灾难性的后果。我在将来会避免跟这样的人共事。

通过这个例子，你可能已经发现为什么“现实理想主义”是优于“理想主义”的。下面我来讲一下，为什么“现实理想主义”也超越了完全的“现实主义”。

超越现实主义

既然我不是一个完全的理想主义者，那么是不是说，我就是一个完全的“现实主义者”呢？在我的职业生涯中，我已经多次证明了，我不是一个完全的现实主义者，我能做到现实主义者做不到的事情。我心中的“理想”成分，让我能够看到现实主义者看不到的可能性，而我的“现实”成分，又帮助我为这种可能性找到切实可行的路线。理想和现实的结合，指引我达到现实主义者认为是不可能的目标。

说到这一点，第一个跳进我脑海里的例子，是我当年在 Google 完成的项目。Google 需要一个可以像 IDE 一样索引 Python 代码的工具，可以支持准确的“跳转到定义”功能。作为现实主义者的团队领导（Steve）对我说，你去拿一个开源的 Python 工具，比如 PyDev，修改之后插入到我们的构架里就可以了。

当我调研了十多个开源 Python 工具和 IDE 之后，发现它们都不能准确地实现“跳转到定义”。它们的实现方式基本都是字符串匹配而已，所以找出来的“定义”完全不着边际，甚至把字符串里出现的名字都给加亮了。这时候，我的理想成分告诉我，准确的定义查找应该是可能的，只不过现有的工具都不知道怎么实现它而已。为了给 Python 这样的动态语言实现精确的定义索引，就必须实现类型推导，而这是我很在行的事情。于是我决定做一个新的 Python 类型推导器，这样就可以利用它实现精确的跳转功能。

我把这个想法告诉了 Steve 和其它团队成员，结果作为现实主义者的他们，非常的担心这个项目无法在三个月的实习期内完成。Steve 说：“你知道吗，光是写一个 Python 的 parser 就够写三个月了。我很担心你不能完成任务！”这时候，我的现实成分开始起作用。我说：“你知道吗，我并不觉得写 Python 的 parser 是一件很难的事情，但我也不觉得它是一件很有意义的事情，所以我会拿一个开源的 parser 来，利用它生成的语法树，然后在上面完成我们需要的功能。”

结果，我拿了 Jython 里面的 Python parser，然后在上面实现了 PySonar。整个对付 parser 的过程只花了我两天时间，剩下的时间我都在研究和实现最关键，最有趣的部分。我拿了别人已经做好的，自己不想做的东西来，然后加上自己的核心思想，达到了最终的目的。最后，我不但在三个月的时间里完成了 PySonar，而且把它集成到了 Grok 项目里面。

在这个例子里，现实理想主义者帮助了现实主义者，完成了他们以为不可能的事情。本来 Grok 项目在 Google 处于濒临灭亡的境地，由于 PySonar 的成功实现增大了项目的影响力，团队在 Google 存活了下来，并且开始受到公司的重视，相关人员也获得了提拔。今天 PySonar 仍然在为 Google 的 Python 程序员提供高质量的索引服务，它生成的数据在背后默默支持着 CodeSearch 等内部代码搜索服务。

个人兴趣与企业兴趣

最后，我想再讲一个跟这个话题相关的故事，它说明现实理想主义者不但是一种个人技术财富，而且是企业的财富。他不但与“企业的兴趣”一点矛盾都没有，反而在很多时候可以帮助甚至拯救公司和团队。这个故事很有趣，但中间部分技术性有点强，看不懂的人可以跳过。

我曾经在职的某公司，邀请了某位“大牛”来做 VP。经过一段时间的接触，我发现这个人不懂很多东西，尽在瞎指挥。很明显，他并没有把公司的利益放在心上。在多次的瞎指挥之后，有一天他又提出一个“新想法”。他说，我们团队的代码应该实现“模块化管理”。如何实现模块化管理呢？我们把代码按目录结构切分开，分成 30 个“模块”。把每个模块做成一个 Git 代码库（repository），代码库之间通过 Maven 的版本号依赖关系进行连接。每个人负责一两个模块，使用“语义版本号”（semver）标注模块的版本。如果修改了代码，就更新对应的版本号，这样依赖于这个模块的代码库就必须做出相应的修改，才能连接到新的模块代码，不然它们就可以继续使用旧的模块代码……

这个新想法没有经过团队的集体讨论研究，就被 VP 的一个亲信动手实现了。一夜醒来，我们发现代码库被他分成了 30 多个，制定了一系列规章条款，要我们遵守。接下来的事情，我发现自己没法工作了。一天当中有超过半

天的时间，我发现自己在为那些 semver 伤脑经。你刚刚更新了所有的代码，才工作了个把小时，正要提交的时候，却发现另外几个模块的版本号更新了！你得手动去看是哪些代码库发生了改变，更新自己 maven 文件里的依赖关系，然后才能进行测试，提交自己的代码。有时候当你提交之前，忽然又有其它的模块版本号发生了改变，所以你前功尽弃，又得去查到底是谁改了他的模块版本号。有很多次，有人没有把版本号完全搞对就提交了代码，结果导致项目 build 失败。

后来我发现，这种所谓的“模块化”，根本就不是真正的模块化，而 semver 版本号，在这里也并不比 Git 的 hash 更好。模块不应该是按目录结构划分的，而应该是按代码的逻辑结构，而且模块之间不应该有“循环依赖关系”，否则这些模块就不应该被分成模块，而应该合并在一起。另外，semver 根本不是用来干这个事情的，它根本不应该被用于连接同一个项目里的多个模块，它只能被用来引用库代码。每一个 Git commit 的 hash，本身就是一个“全宇宙唯一”的版本号，它包含了代码所处的独一无二的状态。所以 Git 其实自然而然的解决了这种“模块”间版本依赖的问题。所以把代码拆分成 30 多个 Git 代码库，使用 semvar 连接它们，完全是多此一举，而且严重的损害了开发效率。

观察到这个问题之后，我向团队群发了邮件，告诉他们我觉得这样的做法已经造成了我工作效率严重打折，并且指出了问题的要害。一个来自法国的资深工程师深有同感，也开始抱怨，说自己花了超过一半的时间来折腾这些版本号。然而 VP 听了这些意见，却坚持认为自己的“创新”是有价值的，对我们说：“任何一项伟大的创新，都会受到不理解它的旧势力的阻碍。同志们，困难是暂时的，适应是必须的！” 为了这个问题，我们在 email 里面吵了两个星期之久。任凭我们据理力争，拿出具体的证据证明这种做法不可行，严重的伤害了团队的开发效率，VP 凭着自己的名气和地位，毫不退缩。

最后无赖之下，我决定采取实际的行动。我写了一个 Python 脚本，它调用 Git 的一些罕见命令，可以自动把多个 Git 代码库合并成一个，并且保留所有的历史 commit 信息。有了这个脚本之后，我可以随时制造出一个合并的代码库。我把这个脚本分享给了团队，告诉他们我随时可以把代码库合并在一起，而且给了他们一个合并后的代码库，作为试验用。我告诉他们，可以试用这个代码库，看它是否解决了 30 个代码库带来的问题。最后法国同事和其它几个人采用了我的代码库，发现不再有之前的头痛问题。

我们用理论和切实的证据证明了所谓的“模块化代码管理”的不可行。通过对其它公司代码的观察，我们发现 Google 的 Chrome 项目有三千多万行代码，却全都存放在同一个 Git 代码库里。这说明一个 Git 代码库足以支持管理 Chrome 那么大的项目。我们的团队总共才 20 多人，代码不超过十万行，却被强行切分成 30 多个代码库，这是非常荒唐滑稽的。

最后在工程师们的一致同意下，再加上团队 director 委婉的支持，我用脚本将 30 个代码库合并在了一起，结束了大家的痛苦…… 在此之后，VP 的亲信们还不死心，在合并后的代码库里又做了一些手脚，故意加大工作的复杂性，让我们依赖于他们的“工具”，这些我就不细说了。总之你看到了，这位 VP 的瞎指挥，导致团队浪费很多的时间和精力。如果这种情况不受控制继续下去，整个团队甚至整个公司，都有可能因此走向灭亡。

我发现很多所谓管理人物，他们到一个新的公司出任要职，其实并没把公司的利益放在心上。他们不是为了公司的发展和成功做出决定，而是为了自己的“仕途”。这些管理者明白，公司就像一艘船，自己表面上在为公司服务，而其实是在利用公司的资源达成自己的目标。由于自己挥霍公司的资源，而不作出实质的贡献，甚至瞎指挥帮倒忙，这艘船在将来很可能会沉没。但作为管理者，自己总是可以在沉船之前跳到另外一艘船上，靠着自己的关系网，不断找到高薪的职位……

像这样的例子我还有很多。为了团队，为了公司能够达成自己的目标，我多次顶着压力，帮助团队和公司避免不必要的浪费，甚至悬崖勒马。当然很多时候团队在错误的道路上走得太远，看清真相的我却受到压制，没有话语权，所以也爱莫能助，只能听之任之。注意我在这里谈“企业利益”，并不是说我喜欢为资本家卖命。这里的“公司”和“企业”，只是代表一个集体，它包括了公司里所有的员工和股东。

从这样一个例子，你也可以看到我作为一个“现实理想主义者”的特征。这个 VP 可算是“理想主义”了，他一拍脑袋提出了“新颖”的，其它公司都没想到的工作方式，结果却给大家带来了灾难。我从现实和理性的角度，分析得知这种做法的荒谬，论证了“传统做法”的和理性，与他据理力争，维护公司和团队的利益，再加上团结大多数有职业素养的工程师，最终我们合力战胜了 VP 的瞎指挥，逆转了他给团队和公司带来的伤害，避免了灾难性的后果。

当我离开这个公司的时候，我收到了这样一封来自团队成员的感谢 email：

他说：“谢谢你帮助我们保持了常理和理智，把事业推向前进。我们会怀念你的！”

这样的现实理想主义者，不管是作为员工，作为团队的领导，还是作为公司的统帅，都会身体力行，给他们带来帮助，避免不必要的浪费和弯路，引导企业走上正轨，走向兴旺繁荣。我希望广大 IT 工作者能理解我这里说的东西，把自己的“伟大理想”植根于现实，避免因为自己的轻狂而走向歧途。

如果你觉得这篇文章对你有帮助，可以自愿付费购买，建议零售价：￥30。