InfoQ 软件开发丛书 Software Development Series

完美软件开发：方法与逻辑

◎作者：李智勇　丁静

InfoQ

完美软件开发：
方法与逻辑

李智勇 丁静

横尽虚空，山河大地，一无可恃，而可恃唯我；
竖尽久劫，前古后今，一无可据，而可据唯目前。
---杨昌济

目录

前言：对软件开发的一次另类思考.....4
第一章 完美软件开发之解构.....6
§1.1 完美软件开发的定义.....6
§1.2 完美软件开发的构成.....8
§1.3 完美软件开发的前提.....11
§1.4 完美软件开发的用途.....14
第二章 完美项目管理之解构.....17
§2.1 项目管理的存在意义.....17
2.1.1 价值根源.....17
2.1.2 定性分析.....19
§2.2 完美项目管理的要素.....20
2.2.1 逻辑链 1：意愿之价值.....24
2.2.2 逻辑链 2：物理环境.....26
2.2.3 逻辑链 3：文化环境之“意识形态”.....27
2.2.4 逻辑链 4：文化环境之“观点整合”.....32
2.2.5 逻辑链 5：制度环境之“势”.....35
2.2.6 逻辑链 6：制度环境之“量化管理”.....39
2.2.7 逻辑链 7：内耗之终结.....41
2.2.8 逻辑链 8：沟通之成本.....44
2.2.9 逻辑链 9： 组织行为之优化.....45
§2.3 完美项目管理.....48
2.3.1 完美项目管理的形象.....48
2.3.2 完美项目管理的关联要素.....50
第七章 完美设计和编码之解构.....52
§.1 设计、编码和文档间的关系.....53
.1.1 【设计 = 编码】 VS 【设计 ≠ 编码】.....53
.1.2 文档的角色.....56
.1.3 设计知识归类法.....57
§.2 设计和编码的存在意义.....61
.2.1 价值根源.....62
.2.2 定性分析.....64
§.3 完美设计和编码的要素.....66
.3.1 逻辑链 1：正交的分解.....68
.3.2 逻辑链 2：层次的控制.....78
.3.3 逻辑链 3：时序下的数据流.....88
.3.4 逻辑链 4：信息的隐藏.....91
.3.5 逻辑链 5：“名”与“实”的契合.....95
.3.6 逻辑链 6：设计的终结.....99
§.4 完美设计和编码.....101
.4.1 完美设计和编码的形象.....101
.4.2 完美设计和编码的关联要素.....105

内容简介

这本书剖析了软件开发中主要环节（管理、流程、估算、开发模型、估算、需求开发和设计编码）的运作规律。

在剖析过程中，主要使用演绎法进行推导，同时使用现实中累积的经验对推导出来的结论进行验证。在这一过程中，借鉴并吸取了PMBOK、CMMI、敏捷、功能点方法、面向对象分析与设计等思想或方法的精华内容。

从读者的角度看，本书更适合有3年以上开发经验，希望在职业路径上走向下一步的人。也适合不只满足于完成手里的工作，还喜欢透过现象思考本质的人。毕业生可以用这本书来开阔视野，谋划自己的发展路径，但有些地方会可能会感到不容易理解。

对本书的进一步交流沟通，可以移步作者微博：@李智勇SZ

前言：对软件开发的一次另类思考

在武侠小说中，常会把绝世武功分为两个部分：招式和心法。招式得其形，而心法传其神。从这个角度看，这本书是即讲招式也讲心法的书。招式繁杂，暂且不提；心法却可以概括。

如果非要用三句话来概括本书中所提心法的全部，那么他们是（顺序不可颠倒，有因果关系）：

> 在尺度中潜在的已经包含本质 ;尺度的发展过程只在于将它所包含的潜在的东西实现出来。 --黑格尔，《小逻辑》
>
> 人心惟危，道心惟微，惟精惟一，允执厥中。
>
> --《尚书》
>
> 横尽虚空，山河大地，一无可恃，而可恃唯我；竖尽久劫，前古后今，一无可据，而可据唯目前。---杨昌济

但这三句话都过于精妙，一般来讲很难把他们和管理、流程、估算、开发模型、需求开发、设计编码联合起来。本书做的正是这样一种尝试：在把软件作为一个整体进行考察的同时，把精妙抽象的东西和具体的东西结合起来。

把软件作为一个整体考察是因为管理、流程、设计编码等都是影响最终成效的砝码，单纯某一个维度上（比如流程）效能最佳不等于整体效能最佳。

把精妙抽象和具体相结合则是因为虽然众象纷繁，但总有些规则凌驾于现象之上，不把握这些规则必然陷入杂多之中，进而明于微而昧于巨。反之，精妙的东西又只有通过具体的手段才能实现自身，并无法单独而存在。恰如下棋时每一步都机关算尽，但总是脱不开既定规则，只有有限的结局（或输、或赢、或和），而既定规则之力量又只能实现于每一步之中。

作为结果，我们可以讲：

> 这书只信逻辑和事实。虽参照诸多素材（敏捷、CMMI、OO、设计模式等）但主要依赖于独立思考才最终塑成体态，纵然错漏难免，但书中所言皆是自主反思所得，虽常有反主流之观点，用心想来却不一定是无稽之谈。

这书直指本质。虽然有的地方略显艰深晦涩，但实是因为无法简化，绝非毫无价值，更非故弄玄虚。同时，为避免偏颇，本书主要使用演绎法，基于以下四个预设前提推导各种结论，并用事实进行佐证。

- 软件是一种固化的思维
- 意识指导行动
- 项目所能耗费的资源是有限的
- 重复做同样的工作会降低效率

这书是培养帅才的书。如果想成为一方悍将（比如：C++高手，Android高手），那这书是不太适合的；但如果想鸟瞰全局，运筹帷幄，带领团队攻城略地，那这书是很有参考价值的。

这书一定程度上可以终止某些争议。软件开发这个行业之中过度相信经验主义，但事实证明效果并不好。几十年下来依然异论相搅，纷争不断，比如：是做架构设计，还是测试驱动；是敏捷开发，还是CMMI等。本书一定程度上可以包容这些矛盾，恰如黑格尔的辩证法可以包容康德的二律背反。

这书是一个开始而非结束。限于作者的眼界、能力、时间等，这书无法终结所提及的所有问题。希望能有志同道合者一同来继续这个题目，也希望能收到各种批评的建议来不断自我提高。

第一章 完美软件开发之解构

某位大哲学家曾经说过：现实世界不过是理想世界的一个苍白摹本。好比说完美的圆只在概念中存在，现实中的圆只能是对完美的圆的无限回归。这也就意味着所谓完美大多只处于虚无之中。

即使如此完美境界仍然具有永恒的价值，这与其可实现与否并无关联，而在于其可以给不懈努力者以终极的指引。

真的完美境界更多的会体现为一种铁则，一种规律，一种必然性，它并不以个人的喜好而变动半分，当你背离它时，它则以惨痛让你重新认识到它的存在。而所谓成功项目或者失败项目所昭示的则是对这种规则性的顺应程度。

当我们处于蒙昧之中时，往往并不知道当为不当为之准绳，其结局必然是在沉重中惶恐，在惶恐中希冀，在希冀中重新陷入迷惘。当此情境，唯有理想真知才能让人破妄返真，重拾方向与希望。所以觉醒之后，虽知一去难返，却终究要在虚无中寻找永恒，以完美境界为现实之归宿。

对完美之追寻，实是以短暂之身，叩问永恒之道，窥天心而解惑。软件亦莫能外---仅以此拉开完美软件开发的序幕。

§1.1 完美软件开发的定义

不识庐山真面目，只缘身在此山中。

任何软件开发，其输入是需求，工具(编译工具等)和人，这三者在特定的时空背景下受主观的影响而改变的可能性小。接下来，基于上述三者，通过选定的管理方法、流程、开发模型、需求开发方法、估算方法、设计编码方法等等对软件进行构建，期望最终达成多重质量目标。最终输出的是软件产品。对软件产品的度量至少有两个维度：一是用户层面的，如功能正确，性能优异；一是结构层面上的，如容易维护等。

从输入到输出这一过程受三个维度的因素影响，一是商业因素，二是技术因素，三是“政治”因素。商业因素决定收益，技术因素决定成本和质量，而“政治”因素添加变数。在这里，“政治”是指人与人之间非理性的复杂关系，是“项目政治”，而非“国家政治”。

如果只考虑技术因素也就是说只考虑软件开发自身的内在合理性，那么最终推导出的就是完美方法，掺杂了商业因素和政治因素后的真实的方法，将是对完美方法的折中和回归。

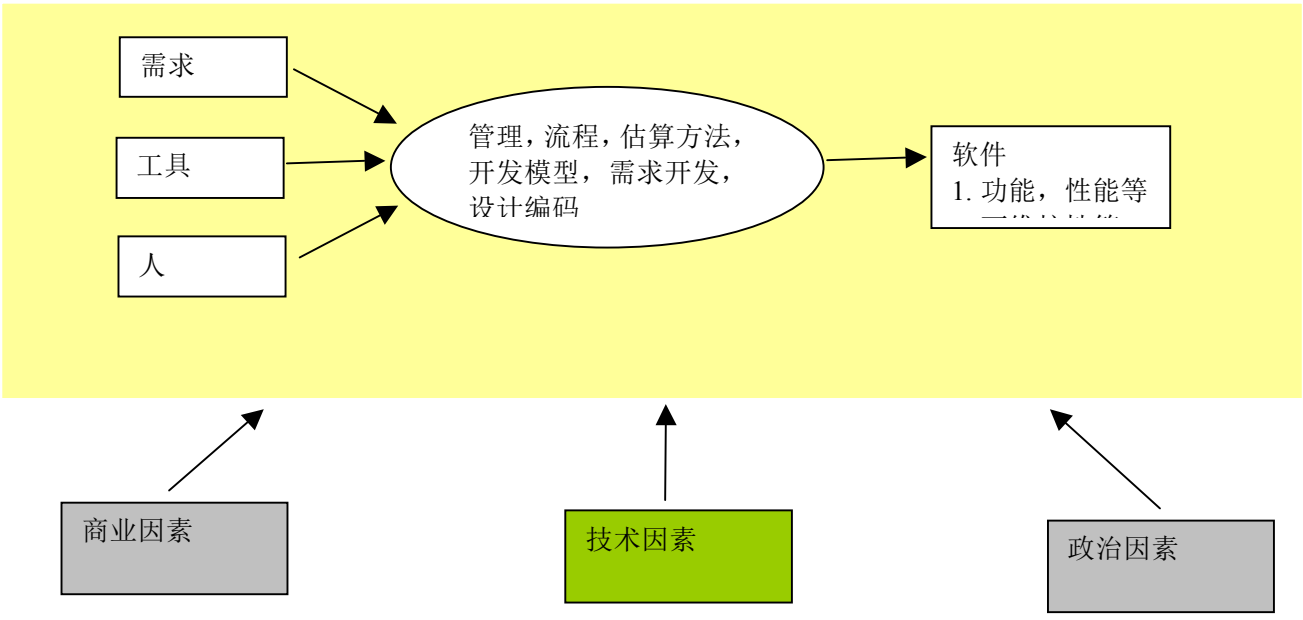


图 1-1：软件开发的基本组成

完美软件开发追求这样一种状态：在只考虑技术因素的时候，在限定的要求、工具、指定的人员状况下（输入），在既定的质量水平下（输出），生产效能最高。在这一状态下，任何对管理、流程、估算方法等在尺度上的修正，如果不以质量降低为代价，都将导致生产效能的降低。

建模

设定一个范围，把范围内的东西抽象出对应的概念，再标明这些概念间的关系这样一个过程就是建模。为免晦涩，这里的定义可能和教科书里略有不同，但建模的概念大致如此，并不高深，几乎人人可做。

§1.2 完美软件开发的构成

自其异者视之，肝胆楚越也；自其同者视之，万物皆一也
---庄子，《德充符》

软件是一种固化的思维，这一点决定了许多的事情。从特质上来看，既然软件是固化的思维，那就必然同时具备**思维**以及**思维所承载之物**之特质。

- 思维的特质是指：思维的澄清通常是渐进的，思维自身是不可度量的，思维的主体一定是人，思维通常由概念和逻辑组成，思维的无边界化（灵活易变）这样的特质。这部分特质是共通部分，同时属于所有软件。
- 思维承载之物之特质是指：当思维的对象是数学的时候，思维本身也就具备了数学的特质；当思维的对象是商业逻辑的时候，思维自身也就具备了商业逻辑的特质。

既然思维自身的特质是复合的，那么作为固化思维的软件，其特质必然也是复合的：

既有属于所有软件的共同特质，也有特属于某类软件，甚至同其他类软件完全相反的独有特质。

这也就意味着在软件这一大的范畴里，两种矛盾的说法同时成立，并不是什么值得惊讶的事情。

既然软件的世界如此之多元，那么进行完美软件开发的讨论时，就必须忽略一些细节，才可能在限定的篇幅下，取得有价值的结论。因此，在后续讨论中，将更多的基于思维的特质，而非思维承载之物的特质，来探讨软件。这样得出的结论才有更多的普适价值。

如果说软件是一种固化的思维，那么无疑的软件开发即是思维固化的过程。在思维固化的过程中事实上有两个层面的问题同时需要解决：

- 思维的主体必然是人，当多个人在一起协作的时候，彼此间的关系如何处理？在这背后隐含的两个领域是**管理**和**流程**。
- 在软件开发过程中，从本质来看，事实上只有两个根本步骤：一是弄清楚要做什么（**需求开发**），二是对思维进行固化（**设计**，**编码**）。对这个两个步骤的时序进行各种安排，则产生各种**开发模型**。为支持这两个步骤能够平滑进行，那么需要预先进行**估算**。

上述分解总结起来就是下图：

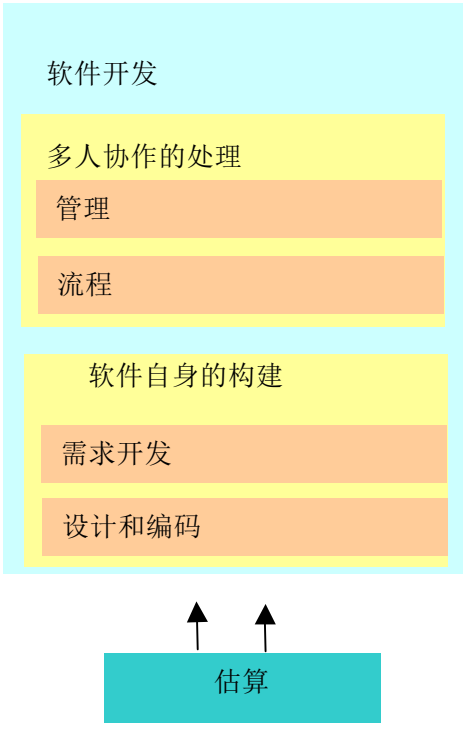


图 1-2：软件开发中各项活动的归类

在做出上述分解后，我们可以进一步推断完美软件开发必然包含着两个根本命题：

一是上述各个分解自身的最优化；一是上述各个步骤彼此间搭配的最优化。

也就是说现存的大多方法论（CMMI、敏捷等），由于其过度强调某一单一维度，同时漠视方法与软件本质间的关联，一定程度上讲其本质是苍白的。

下图用于说明这种关联的复杂性：

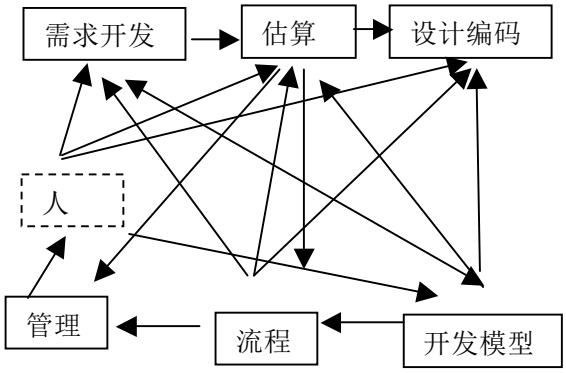


图 1-3：软件开发中各项活动的关联关系

本质与细节

这世上同时存在着两种对立的声音：本质决定成败和细节决定成败。偏好本质的人喜欢说本质论。偏好细节的人则喜欢说精细化管理。但如果在较长的时间轴上考量这两种观点，就会发现他们之间并不真的对立。

本质决定大尺度时间上的走势和必然性，而细节则决定差异（包括短期成败）。比如说：人的本质特征是能思考，有一个头，会衰老，寿命有限等，但区别不同人却不是这些，而是性格，肤色，发色等细节。

具体来看：软件本质上是只有人才能处理的东西，因此公司中程序员群体的衰落

一定会导致软件自身的衰落，只有优秀的程序员群体，才能保证软件的持久成功，这是必然性。但优秀的程序员却不一定确保当前项目成功，任何人在细节上的小疏忽，都可能导致软件在市场上崩溃，死锁，进而导致灾难性后果，这就是细节决定成败。

成败自身虽然万众瞩目，对个体而言却只是一种偶然和机巧。当事人可以很努力的平衡本质上的追求（长期视点）和细节上的追求（短期视点），但变更的始终是一种成败可能性。

§1.3 完美软件开发的前提

十世古今，始终不离于当念；无边刹境，自他不隔于毫端

---李通玄

当我们试图对完美的软件开发进行阐述时，事实上总是有两类方法可以帮助我们达成目的：

- 归纳法。使用归纳法时需要基于项目经验，总结出一定的规律，再推而广之。这样的话在没有反例出现之前，那么总结出的规律一直成立。
- 演绎法。使用演绎法时需要基于预设的前提，依照逻辑，推导各种结论。这个时候，只要前提没被推翻，逻辑又没有错误，那么得出的结论就具有必然性，虽然这种必然性何时体现在现实之中比较难以测度。

当前软件行业的大多方法论和结论是基于归纳法的。我们前面曾经论及，软件作为一种固化的思维，必然同时具备思维以及思维承载之物的双重特质，而思维承载之物的特质有时候甚至可能产生矛盾，因此基于归纳法得出的结论，容易偏狭并引发争议。

显然的，从数学类软件上得出的结论与业务流类软件上得出的结论并非是共通的；适用于操作系统内核的经验，也并不一定适合于视频播放软件。与此同时，人的精力是有限的，这就导致一个人即不太可能经历所有的软件开发领域，也不可能穷尽同一类软件下的所有变数。比如说：A所经历的MIS系统和B所经历的

MIS系统很可能天差地别。

这很可能是软件行业中纷争不断的一个主要原因---每个人都以为自己看到的就是全部，而当另一种与自己不同的观点出现时，这种观点会因与己不同而被判处死刑，但事实上两者可能同时正确，只是正确的边界不同。

为避免这类争议，在这本书里，我们将主要使用演绎法。为得到各种结论，我们将主要基于以下四个前提，使用逻辑进行推导：

- 软件是一种固化的思维
- 意识指导行动
- 项目所能耗费的资源是有限的
- 重复做同样的工作会降低效率

这四个前提被假设为公理，将不额外进行说明。而为使结论不显得晦涩和突兀，我们将尽可能使用现实里的例子对逻辑链和结论进行佐证和补充。

为了从这四个基本前提，推导出属于各个部分（管理、流程等）相关结论，我们还需要对影响团队创造价值的因素做进一步分解，否则事情会保持在混沌状态，进而无法以正确的尺度判断某一行为所能产生的正负两方面影响。

如果我们假设一个人的工程素养为E，一个人的工作意愿为W，组织所能提供的力量为O，内耗系数为M，那么对于一个拥有n个人的团队，其在单位时间内最终可能贡献值可以表示为：

$$[(E_1*W_1 + O) + (E_2*W_2 + O) + \ldots\ldots + (E_nW_n + O)] * M$$

其中M的取值可以为0到1。0表示完全内耗掉，整个团队完全没有贡献。

*注：很多人会对为什么可以用加法累积不同人的贡献这一点产生疑问，比如：架构设计师的贡献和测试人员的贡献为什么可以叠加等，这点将在**附录1**中统一进行说明。*

这也就意味着：

单位时间团队总贡献=[（A的工程素养*A的工作意愿+组织力量） + （B的工程素养*B的工作意愿+组织力量）+...]*内耗系数

其中：

- 工程素养是指人员的工程能力，比如：需求分析能力，架构设计能力，编码能力等。
-
- 工作意愿是指一个人愿意不愿意工作。工程素养和工作意愿乘起来才是一个人的可能贡献。这并不难理解，一个能力不好的人，再怎么努力，贡献也不会好；而一个能力好的人，每天蒙事，其贡献也不会好。
-
- 组织提供的力量是指常说的组织力。比如说：组织内有比较好用的重用代码库，那么不管谁都可以从中受益。探讨组织力，会使范畴发散，因此在本书中大多时候我们会忽视这个维度。
-
- 内耗系数是指人员彼此间贡献的耗散程度。影响因素会比较复杂。比如说：A开发的一个模块和B开发的模块有80%重叠，这无疑会产生内耗。再比如说：A和B坐在一起工作就会吵架，这也会产生内耗。

上面这段分析可以总结为下图：

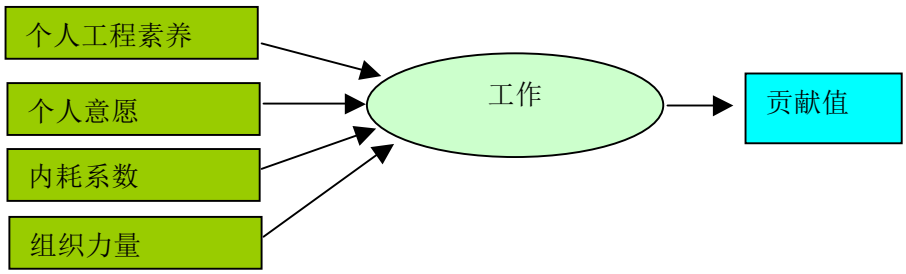


图 1-4： 贡献值的分解

那么在指定的时间段，这一团体的平均生产率为：

$$\frac{[(E_1 * W_1 + O) + (E_2 * W_2 + O) + \ldots\ldots + (E_n W_n + O)] * M}{n}$$

上面的公式可以用来定性分析贡献值的大小，但并不能用来分析贡献值的效果。比如说 :某个团队可能在上述四方面都没有问题 ,最终也开发出了比较好的软件，但由于云计算的兴起，公司开发的这类富客户端软件已经没有市场。这时候贡献值再好，也体现不出效果。

这意味着要想获得更好的结果，一是要保证贡献值尽可能的大，二要保证贡献值使用在正确的方向上。控制方向性虽然重要，但与商业因素等有较大关联，本书中大多时候不会对其进行考量。

演绎法与空谈

经验主义者是不喜欢演绎法的，极端的甚至认为演绎法即是空谈。但我们不应该忘记《资本论》与《相对论》这两个与演绎法有深刻关联的东西对世界所产生的巨大且深刻的影响。

在软件工程或者项目管理领域中，对演绎法的排斥达到了极致。人们似乎很忌讳谈及不与具体实践相关联的东西，对经验的推崇则是达到了无以复加的地步。但事实却证明这并不很有效，几十年下来，软件这个行业中一如既往的纷争不断。

比如说：Linux之父Linus Torvalds大概每间隔一段时间就会站出来贬斥一下C++（有时还连带上面向对象），并经常用垃圾，很差这样的形容词来描述C++。与此同时世界上很多关键系统是用C++成功开发出来的，拥有着许多属于自己的铁杆粉丝。

这似乎很矛盾，也很难判定出是非，但当我们用演绎法来分析这类事情时，就会发现这些貌似对立的事情并非不可调和。Linus主要专注的领域是内核，所以它的观点在内核领域里一定是不能漠视的，但这并不意味着C++和面向对象就不适合游戏，富客户端，数据库程序等等。

这类事实的存在意味着，我们应该切换一下视角，用演绎法来重新审视一下软件这个行业，而不能只是用有限领域中得来的经验去臆测无限范围中事物---这是必将陷入矛盾方法。

§1.4 完美软件开发的用途

> 夫英雄者，胸怀大志，腹有良谋，有包藏宇宙之机，吞吐天地之志者也。
> ---罗贯中，《三国演义》

完美境界是这世界上的终极力量，真的完美绝不苍白无力。其背后所隐藏的是规则的力量。

牛顿第一定律说：任何一个物体在不受任何外力或受到的力平衡时，总保持匀速直线运动或静止状态，直到有作用在它上面的外力迫使它改变这种状态为止。这无疑的是在描述一种完美状态，任何物体当然是受外力的，但谁敢说这一完美状态是苍白无力的。

完美的软件开发状态，其存在意义与上述相同。具体而言，其真实作用是帮我们俯视全局，对种种问题洞若观火，进而把持解决问题的方向和尺度。

为获得最终的软件产品，中间必然面临种种艰难。而在特定场景下，这种艰难背后所隐含的主要矛盾往往不同，所需的处理方法也因此而不同。比如：如果项目的主要问题是缺人或人员不负责任，那么加大技术培训，增强编码能力，无疑是缘木求鱼。

为了解决种种问题，事实上需要有人有一个软件开发的全体视图，再从中找到最关键的矛盾，接下来采取具体措施进行解决，这就是小说里常说的包藏宇宙之机，且腹有良谋。

而上述所有关键步骤都需要完美状态作为参照，并把握现实和完美状态间的距离---现实世界中难的并非是没有方法，而是待选方法太多，不好把握其间尺度。这就要求首先要有完美状态下的整体视图，同时不能让采取的措施与完美状态下的规则相背离，否则就会陷入迷途。

智慧与珍珠

佛家有“智珠”之说，“智珠在握”则用来形容有高深的智慧可以应对任何事情。非常巧合的是，达成“智珠在握”状态的过程与珍珠的形成过程极度类似。

一旦有异物侵入蚌的外套膜， 蚌就会不停的分泌珍珠质，最终得到的就是珍珠。这与智慧的形成过程相类。

一个人对软件开发的全体视图可以浅陋，单薄，但一定要有。这就是形成珍珠的那粒杂质，它可以是沙粒，甚至可以是鸟屎，这都没有关系，关键是要有。在此基础上，可以读书，可以从实践中反省，最终就会生成璀璨夺目的“珠”。但如果没有这粒杂质，很多东西就无所依凭，到头来学到的东西就会彼此冲突，知识将反成为一种障碍。

这正是本书的目标之一---通过解构和逻辑链来帮助每个人形成自己的那粒杂质。

第二章 完美项目管理之解构

项目管理自身并不复杂，却往往被看做一门复杂的学问。究其根本，实是因为为术则日繁，为道则日远。这就好比伐树的时候却从剪枝叶做起，其结果必然是只见其繁杂，而不见其根本。

在这一章里，我们将对管理的根本命题进行解构。如果你曾经对下面这些问题困惑过，那么在这里，你将找到一份逻辑上说的通的答案。

为什么同样人数，天分又差不多的团队，爆发出来的战斗力却有天壤之别？

为什么看上去很好的量化管理，一旦导入却会天怒人怨？

为什么团队的成员会从朝气蓬勃，变得得过且过？

为什么开源项目没有项目经理，往往也运作的很好？管理真的有价值么？

为什么工厂式的开发不适合软件？

项目经理究竟要不要懂技术？

… …

§2.1 项目管理的存在意义

君子务本，本立则道生。

---孔子，《论语》

2.1.1 价值根源

价值是一个可以引起无数纷争的词语。在商业社会中，价值的界定就更为艰难。即使是完全相同的两个程序，广为人知的可能价值亿万，没于静室的却可能一文不值。所以当我们考察项目管理的价值时，我们必须剥离一些东西，使价值的范畴更加清晰。

在后续各个章节中，对价值进行讨论时，我们将要剥离的是软件的商业价值。也就是说，在完美的世界中，我们关注的是做最好的软件，但并不关注最好的软件是否有市场。后者显然是有意义的，但并不在我们的考察范围之内。

在剔除商业价值之后，我们来看一下软件的一个根本特质。由于软件是一种固化的思维，而思维固化的媒介必然是代码，因此我们可以讲：

软件与代码相等价，是同一事物的一体两面。也即说代码可以表征软件的一切价值。

如果项目管理确实有意义，那么项目管理必须直接或间接的对代码施加影响。而显然的，项目管理很难直接对代码产生影响，这也就意味着项目管理自身并不直接创生价值，必须以他人为媒介才可能最终产生自己的价值。

项目管理者对他人所可能产生的影响可以分为两类：一是对个人的影响；一是对组织的影响。

- 对个人的影响可以体现在对消极、排斥异己、懒惰、好高骛远等负面情绪的遏制上，也可以体现在肯定成绩、表达信任这类激励方法上。
-
- 对组织的影响则可以体现在对谋而无断、职责不清、流程不清这类负面组织行为的遏制上，也可以体现在创建共识、对组织意识形态进行引导这类避免纷争的手段上。

上述两点正是项目管理的价值根源。

在下一节里，我们将对上述的价值根源做一些定性的分析，但在进行定性分析前，有必要对项目管理这一常用词语的范畴做一点补充说明。因为我们推导出来的价值根源与大多数人的缺省认识可能已经有了较大的偏差。

我们来看一下PMBOK对项目管理的定义：

> 项目管理就是把各种知识，技能，手段和技术应用于项目活动之中，以达到项目的要求。项目管理是通过应用和综合诸如启动，规划，实施，监控和收尾等项目管理过程来进行的。
>
> ---PMBOK

这个定义中的关键词是【各种】，在PMBOK中，【各种】被定义为诸如：

法律、金融、标准、规章制度等，甚至也包含文化和社会环境、政治环境等。

PMBOK和我上述的推导结果间的一个显然差异是 :PMBOK强调的项目管理的职责更多的是向外看的，而我们的推导结果是：项目管理的价值根源在于对内部的人或组织施加影响。

造成这种差异的一个原因是：PMBOK考虑的是现实的世界，并没有如我们一般剥离商业价值这一维度。而如何在两者之间寻找平衡，则是从完美世界回归现实时需要考虑的问题，需要具体情况具体分析，我们这里就不做进一步讨论了。

2.1.2 定性分析

让我们回到第三章中提到的公式：一个人的工程素养为E，一个人的工作意愿为W，组织所能提供的力量为O，内耗系数为M，那么对于一个拥有n个人的团队，其在单位时间内最终可能贡献值可以表示为：

$$[(E_1*W_1 + O) + (E_2*W_2 + O) + \ldots\ldots + (E_nW_n + O)] * M$$

其中M的取值可以为0到1。0表示完全内耗掉，整个团队完全没有贡献。

没有管理者时，在指定的时间段里，这一团体的生产率为：

$$\frac{[(E_1*W_1 + O) + (E_2*W_2 + O) + \ldots\ldots + (E_nW_n + O)] * M}{n}$$

假设加入了一个管理人员，由于管理人员不直接创生价值，生产率的公式变为：

$$\frac{[(E_1*W_1 + O) + (E_2*W_2 + O) + \ldots\ldots + (E_nW_n + O)] * M}{(n+1)}$$

任何管理手段几乎不可能对工程素养E产生影响，短期内也很难影响组织力O，因此如果工作意愿W或者内耗系数M没有变化，生产率必会降低。

反过来讲，项目管理者必须对工作意愿W或者内耗系数M施加正面影响才能促进生产率的正向增长，进而阻止生产效能的降低。

为做进一步分析，我们引入经济学中边际价值和边际价值递减的概念：

边际价值是指在其他条件不变的前提下，增加一单位要素投入所增加的产品的价值。边际价值递减是指超过某一水平之后，边际投入的边际产出下降。

边际价值递减可以帮我们推导出下面的结论：

当一个人处于消极状态时，其工作意愿可挖掘空间比较大，管理的潜在价值也比较大（边际价值大）。但当一个人本来就是为兴趣而工作，其工作意愿必然是处在高端上，这时候事实上你即使做很大投入，也很难进一步提高其工作意愿（边际价值递减）。

在后一种情形下，管理者的价值根源就变成了降低内耗系数。惩罚，奖励等对个人施加影响的手段变得无效，管理者的职能也就进一步弱化，变为coordinator。

这能很好的解释，为什么开源项目中并不存在真正意义的管理者，但很多项目依然运作的很好。

Eric Raymond在《大教堂与市集》这篇文章里专门提到了管理，在列举了一些管理的基本任务（如：确立目标、监督、激励、组织、资源监护）后，Eric Steven Raymond 说：

> 显然所有这些目标都是有价值的。但是在开源模式以及其社会语境中，这些目标会变得出奇的不靠谱。

上面的定性分析可以成为这一现实的最合理解释。

到这里，我们可以得出几个很有意思的结论：人工作意愿越差、内耗越高的时候，管理价值越大，此时边际价值极高。也就是说，组织越成熟，管理的价值越小。组织越不成熟，管理的价值越大。这也意味着，管理的晋级方向是消灭管理，而不是让管理更有价值。

§2.2 完美项目管理的要素

> 万物负阴而抱阳，冲气以为和。
>
> ---老子，《道德经》

根据2.1一节里的结论，项目管理为了体现价值，必须对个人意愿或者内耗系数施加影响。接下来，我们将对如何施加影响进行一些分析。

任何人都有一个自己的精神世界，外部种种往往成为对这一精神世界的输入。意愿则事实上是个人精神世界对种种输入的一种反应。

一个人的精神世界往往有其深厚之历史根源（家庭、经历、文化背景等），其改变往往需要剧烈刺激（家破人亡、生命受到威胁等），但大多时候，这种剧烈刺激并不存在，也就是说，大多时候影响个人意愿的主要手段是控制种种“输入”，而非直接使其改变。

“输入”本身大致可以分为如下三类：物理环境、文化环境（或者说非制度环境）及制度环境。

- 物理环境是指空间、桌椅、电脑等。
- 文化环境是指未曾明确定义，但又相对清晰的价值取向和行为规范。
- 制度环境是指明确定义了的行为规范。

上面的表述可以概括成下图：

图 2-1： 影响个人工作意愿的要素

假使说个人意愿上的所有问题都得到了解决，每个人都可以很理智的看待问题，做到彻底的公心公论，那么内耗产生的根本原因可以进一步分解为：观点的分歧和不恰当的组织行为。

- “观点的分歧”是指盲人摸象式的争执等纷争，这时任何人的观点都有属于自己的合理支撑，但不一致，进而造成对立。

- “不恰当的组织行为”是指权责不清或者工作时序安排不妥这类问题。比如：

  例1：项目的关联要素往往形成一定的因果关系。如果这种因果关系发生混乱，那么会导致人员空闲或者返工等现象出现，进而降低效能。假设说为开发某个程序需要购入某款设备，而设备的购入周期为一个月。如果预先购入设备，那么项目无疑能够平滑进行，但如果在使用前才意识到需要购入这款设备，那么很可能所有团队成员都需要空等一个月。

  例2：项目中人员的能力，兴趣和擅长领域是不同的，如果在项目执行到一定程度的时候，才发现，一个人在做根本不适合他做的工作，那么无疑的团队的效能会被降低。任何项目的问题都具有一定时空特征，而在大多时候，早发现问题并进行应对，代价总是要比晚发现，在问题已经变严重后再进行应对成本要低。

上面的观点可以概括成下图：

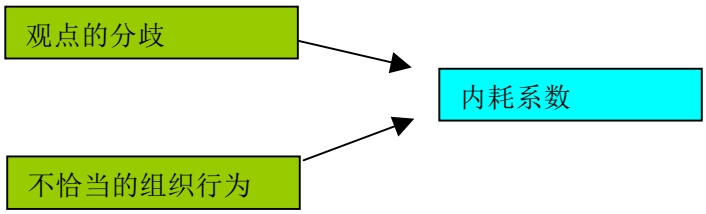


图 2-2： 影响内耗系数的要素

“不恰当的组织行为”这一维度是当前流行的“项目管理学”的主战场，是PMP这类认证的关键，但实际上这只应该是项目管理中的一个分支，只关注它，会导致以偏概全。

上面的分解告诉我们，用逻辑对管理方法进行推导的时候，我们需要从下面几个方面入手：

- 个人意愿
  - 物理环境
  - 文化环境
  - 制度环境
- 内耗系数

■ 观点的分歧
■ 不恰当的组织行为

针对这几个方面我们提出一些逻辑链来分析其最关键的控制点。

<table>
<tr><th colspan="3">管理的分解</th><th>逻辑链</th></tr>
<tr><td rowspan="7">管理</td><td rowspan="6">个人意愿</td><td colspan="2">软件是一种固化的思维 → 只有人才是思维的主体 → 思维依赖于个人 → 个人意愿对软件生产的影响巨大</td></tr>
<tr><td>物理环境</td><td>软件是一种固化的思维 → 思维的基本组成是概念和逻辑 → 概念和逻辑的推演和确立是需要集中精力，连续实施的 → 物理环境上要尽可能保证思维着的人有独立思考的空间和时间而不被外物所打断</td></tr>
<tr><td rowspan="2">文化环境</td><td>意识指导行动 → 个人意识受环境中现存共识影响比较大 → 真正的共识下，个人行为即组织行为，组织行为也是个人行为 →如果所有组织行为，都是共识下的行为，那永远不会有个人意愿的损耗，也不会有纷争内耗</td></tr>
<tr><td>软件是一种固化的思维 → 思考是思维形成必经之路 → 积极的思考和被动的思考差异巨大 → 组织里需要让人积极思考的氛围 → 形成积极思考氛围的关键要素是：积极思考是被鼓励的 → 积极思考的成果没有被轻易漠视，对个人有逻辑的观点的强制否决要尽可能的少</td></tr>
<tr><td rowspan="2">制度环境</td><td>意识指导行动 → 总体来看，个人意识是倾向于扩张而非收敛（收入，荣誉，地位，自己意志的实行程度）→ 扩张的目标和现实的差距是每个人前进的动力 → 管理的一重使命是确保这种差距存在。</td></tr>
<tr><td>软件是一种固化的思维 →思维的本质是概念和逻辑 → 概念和逻辑无法直接度量和精确度量 → 度量过程中需要很多的主观判断 → 以目标为导向的，以个人为中心的量化管理（相关的激励和惩罚）将崩溃 → 参照无歧义数据（函数复杂度等）的判断将成为程序员评价中的辅助手段</td></tr>
<tr><td>内耗</td><td>观点的分歧</td><td>软件是一种固化的思维 → 就思维上大多的问题而言，基本上是一题多解 → 组织结构要确保各种情境下都能做出尽可能正确的决断，但又不能伤害个人意愿 → 层级过多，会导致做决定的人离现场越远，对真实的情况把握越不清楚（信息在传递时会损失），同时误判增加，对个人意愿伤害增加→ 组织应该尽可能扁平 → 为避免谋而无断，即使扁平，也要有能做出决定的人</td></tr>
<tr><td colspan="2"></td><td>软件是一种固化的思维 → 思维自身具有迭代特质（否定之</td></tr>
</table>

			否定）→ 思维的主体必然是人，思维的对接也是只有人才能完成的工作 → 项目越大，需要的迭代也就越多，必须的人和人间的沟通量也就越多 → 参与沟通的人越多，需要协调的不同个性的人也就越多，效率也就越差 →所以软件生产往往表现为规模不经济。同一个软件，一个人开发，效率最高，人越多效率越差 → 尽可能控制团队规模 （控制沟通成本）
		不恰当的组织行为	重复是降低工作效率的 → 工作分解必须是正交的，并且时序合理，符合因果关系 → 项目的资源是有限的，必须合适的人做合适的事情 → 为达成这一目的，项目管理者必须既理解所做的事，也理解相关的人员特性 → 为避免遗忘，使经验转化为价值，普适于多个项目的，特定的因果关系和时序需要被提取出来并强制实行，最终形成流程

表 2-1： 项目管理相关的逻辑链

其中第一条逻辑链不针对具体子项，而是直属于“工作意愿”的。主要是用来说明，为什么软件这一行业中，“工作意愿”需要被特别强调。

下面我们将对这9条逻辑链分别进行说明，为使逻辑链不至于过度晦涩，在进行说明的同时，我们会对每条逻辑链进行命名，在后续章节中将一直使用类似的方法。

2.2.1 逻辑链 1：意愿之价值

软件是一种固化的思维 → 只有人才是思维的主体 → 思维依赖于个人 → 个人意愿对思维的确立影响巨大。

在不同的组织里，人的价值往往不同，意愿的价值也因此而不同。通常来讲，当一项工作越倾向于思维，个人的价值越大；反之则个人价值越小。当个人价值大的时候，工作意愿所隐含的价值也随之增大。

这与我们日常的所见所闻相符，我们来看两个极端的情形：

在生产性企业（工厂等）里，人员的职能相对比较单一，可替换性也就比较强，个人的价值也就比较小；但在理论研究这样的领域里，人几乎就是一切。

在生产性企业里，即使工作人员对工作比较反感，只要能保证工作比较机械性的一面，则仍然可以持续创造价值。但搞理论研究的人，如果心不在焉，那出成绩的可能性几乎是零。

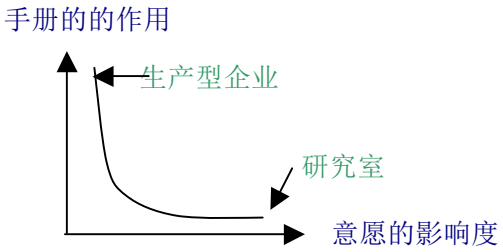


图 2-3：工作意愿与企业类型的示意图

如果我们认为生产性企业和研究性组织是两个端点的话，那么软件则在这两个端点之间占据很长的跨度，但总的来看比较倾向于后者。这是由软件自身的特质所决定的。

如前文所述，软件同时具有思维的特质和思维承载之物的特质。这就使软件的复杂程度可以具有很高的浮动空间。有的软件开发门槛比较低，甚至可能达到只要是正常智商的人就可以进行的地步。

这并不让人惊讶。

思维是上天赐予人类的礼物，只要是正常的人，无分高低贵贱，都具有使用思维的能力。从这个角度上看，只要是人就具备进行程序开发的能力 --- 只要软件承载的东西足够的少。但也正如语言人人可以懂，人人可以说，却远不是每个人都是文学家一样。当软件自身的内在复杂度逐步提升，质量要求也逐步提升的时候，软件的个人依赖特质，就会表现的越发明显。

我们常见的小规模的信息管理系统更接近于前者，而操作系统内核，大规模的系统则更接近于后者。但不管是那类的软件开发，只要规模，质量要求，性能的要求达到了一定的程度，都会使软件开发更倾向于研究性组织而非生产性企业，在这种时候意愿对软件生产的影响就变的巨大起来。

量化意愿很难，但定性分级却并不难，大致可以有这样三个层次：

- 当一天和尚撞一天钟，有事就应付一下的层次。
- 能对自己工作负得起责任的层次。
- 全身心投入，主动工作的层次。

“当一天和尚撞一天钟”状态下，工作意愿可以无限趋近于0。

意愿状态的判断

在日本管理人员中流传着这样一种共识，当你不太弄得清一个公司的状况时，你可以去看看它的卫生间。卫生间脏乱差的公司大致上是混乱的公司。这方法听着偏门，但确实在一定范围内是有效的，因为完全不在意卫生间环境的团队，往往就是工作意愿较差，抱着当一天和尚撞一天钟想法的团队。这同时也提示我们检查一个团队的工作意愿并不难，并不需要繁琐且复杂的方法，见微知著已是足够：

工作没完，但如果没人督促，所有人都到点下班。这表示这是一个“当一天和尚撞一天钟”的团队。

说好的事，大家都是能拖就拖，能不做就不做。这也表示这是一个“当一天和尚撞一天钟”的团队。

顺手就可以做好的事，却从来没人关心，比如：水龙头漏水了，却从来没人和负责人进行联络。这也表示这是一个“当一天和尚撞一天钟”的团队。

……

2.2.2 逻辑链 2：物理环境

软件是一种思维 → 思维的基本组成是概念和逻辑 → 概念和逻辑的推演和确立是需要集中精力，连续实施的 → 物理环境上要尽可能保证思维着的人有独立思考的空间和时间而不被外物所打断

无论是软件中各个概念的确定，还是逻辑的推演，事实上都是一个连续思考的过程，任何对这一过程的打断，都需要在脑海中重新恢复现场，重新进入状态，这无疑是影响效率的。

一如《人件》所强调的，只要当事人从事的软件开发还不是不用思考，随随便便就可以完成那一类时，无疑的要确保当事人思考的空间和时间。

以当前国内的社会条件而论，达成《人件》中所描述场景（如：每人一间办公室）估计还需要较长的时间。这首先是经济问题。如果软件每年所带来的收益只等价于100间办公室的租金，并且需要1000个人来开发，那么上述想法就是不可能的。

如果能提供一组数据来描述【人/每平方米】和【思维效率】间的对应关系，无疑是有帮助的，可惜当前还找不到这样的数据。

作为折中，也可以尝试培养“上午少互相打扰，下午多交流”这类的工作习惯来做些弥补。

2.2.3 逻辑链 3：文化环境之“意识形态”

意识指导行动 → 个人意识受环境中现存共识影响比较大 → 真正的共识下，个人行为即组织行为，组织行为也是个人行为 →如果所有组织行为，都是共识下的行为，那永远不会有个人意愿的损耗，也不会有纷争内耗

组织中的共识，勉强可以称为“意识形态”。意识形态的力量可以无限大，再怎么高估也不为过。

经济学家凯恩斯曾经讲道：

经济学家与政治学家的思想，其力量之大，往往出于常人意料。实际上，统治世界的不过就是这些思想。

这虽然未必完全正确，但实是对意识所蕴含的力量的极佳诠释。意识形态这个题目摊开来，可以无限广，在这里我们把范围收束在团队内部。

科幻小说作家刘慈欣在《白垩纪往事》中描写了这样一种场景：

蚂蚁和恐龙分别创建了自己的帝国，而有一天两个帝国要开战了，其原因是：蚂蚁认为上帝的样子应该是蚂蚁，而恐龙则认为上帝的样子应该是恐龙。

与蚂蚁和恐龙的战争相类似的事情，在项目团队内部每天上演，但其频度和程度

则受限于共识的多少。

一些不是项目层面可以控制的事情可以被忽略，但在项目层面至少要对一些经常发生的事情形成共识，不然个人意愿会降低，内耗会增大。

下面列举一些经常发生的，需要达成共识的例子，以供参考：

- 相信谁的共识。项目中碰到问题了，大家调查了一下答案：麻省理工可能说 A 对，CMMI 可能说 B 对，团队中有两个人坚持认为自己是对的。这时候怎么处理？这类事情上需要培养的是一种：不唯书，不唯上，只为实的共识---即使最终自己仍然可能是错的。

- 如何判断意见有没有被尊重的共识。团队可以尽可能尊重每个人的意见，但最终各种事情的选择往往只能有一个。这意味着很多人的意见最终会被抛弃。这个时候团队中如果没有“意见被考虑过，不被采纳也是正常”这样的共识，那么个人意愿会非常容易受到干扰。

- 忙与不忙的共识。比如说：A 说我很忙，无法做当前工作之外的事情。B 则认为你从来都不加班，那里就忙了，明显是不愿意多承担工作。于是两人之间就可能产生不必要的恩怨，进而成为内耗的种子。

- 小利益分配上的共识。比如说：有新人加入时，一般会分配新的电脑。这时有的公司把分配的权利下放给团队自身。而有的团队则会把新电脑分配给老员工，新员工用换下来的旧电脑。这时候如果没有共识（可以表现为定义好的分配规则），会导致意想不到的麻烦。
  - 新员工会以为，明显是我的，怎么拿给别人，很气愤。
  - 分不到的老员工会以为，怎么给他不给我，很气愤。
  - 分到的老员工会以为，明显我工作需要，你们有什么好不满的，也很气愤。

- 【组织】与【个人】间利益均衡上的共识

  即使在最为公正的环境中，组织利益和个人利益还是会有所冲突。这种冲突，不涉及是非标准，而只和价值取向有关系。比如：有的员工可能特别喜欢自由随意的风格，但为了项目安全，具体的工作却可能要求员工严格遵守某些确定的规范。这个时候很难讲对错，所能有的只是一种选择。

  这类情形下，需要培养的共识是关于那种程度的冲突是正常的应该接受，那种程度的冲突是过分的，应该坚决反对的。要么接受，要么拒绝都是不错的选择，勉强接受，而又心里抱怨就不好了。

  没有这类共识，长时间下来，每个人心里剩下的可能就只是抱怨。

- 无心之失上的共识

考虑下面这两类情形：A 努力做事，承担的工作也比较多，且艰难；B 比较消极，承担的工作也就比较少，且容易。这时候 A 因为某个疏忽导致了一个错误，影响了工作；B 的工作却做的比较顺风顺水。

这时候应该如何看待这 A 和 B？这并非是一个是非问题，而是一个选择问题。

假如说，一个组织以责任感和主动性为评价一个人的唯一准绳，那么A就并不值得苛责，反倒是B会逐渐出局。

中国古代的奖惩观

在《尚书· 大禹谟》中皋陶在评价帝舜时说：

“..宥过无大，刑故无小；罪疑惟轻，功疑惟重；与其杀不辜，宁失不经...” 。

大意是说：宽恕臣民无心的大过，但重罚那种故意犯下的罪行，即使那只是小罪。对有疑之罪处罚从轻，对有疑之功赏则从重。为了避免枉杀无辜，宁可放弃那种不能肯定的罪人。---何新《大政宪典· 尚书新解》

细心想来，这数千年前的智慧足以击穿时空，今时今日仍然闪烁着耀眼的光芒。

- 保持现状与勇往直前上的共识

如果我们认可外部环境始终在发生着变化，那么不断的改变自身通常就是必要的。与这种必要性相对的是----很多时候人们讨厌变化。

社会变革中，这种反对力量其根源也许更加复杂，团队中这种反对力量大多来源于一种安全的假象以及对思考和风险的厌恶。

潜意识中很多人认为，对于一种行之已久的方法，既然他没导致任何问题，那对项目而言这种方法是安全的，毕竟它不会导致较大的失败。与此同时，任何改变在可能获得收益之前，首先要面对的则是风险。这可能是很多人所不愿承担的。

从【不唯上，不唯书，只唯实】的视角来看，任何改变，其起点必然是某种已经暴露出来的不合理性，而要想提出合适的改变方法，对这种不合理性进行深度思考则是必须的，这将成为一份额外的工作，这种思考可能是很多人所厌恶的。

但是，一旦在一个团队中产生因循守旧的氛围，那么像主动性这类最必须的东西

就会在不知不觉中消失殆尽，团队前进的动力也就会逐步消失。这是极为危险的，所以在这类问题上的共识至关重要。

假设说我们从A点出发进行改善，途经B,C,D但最终却回到了A。这个时候如果我们能保证A→B，B→C，C→D，D→A的变化是有逻辑的，是合理的。那么这种否定之否定的过程就远比始终停留在A要更有价值。

数十年前，鲁迅先生在《无声的中国》里讲：

譬如你说，这屋子太暗，须在这里开一个窗。大家一定不允许的。但如果你主张拆掉房顶，他们都会来调和，愿意开窗了。没有更激烈的主张，他们总连平和的改革也不肯行。

鲁迅先生写这段文字的时候，心里想必是无奈的。这种无奈至少不应该在任何一个团队成员的心里产生。

- 暧昧与直来直去上的共识

  权利无论大小，往往令人迷恋。直接与上级论及其自身的是非，往往会引致上司的不快。而上司往往拥有对工资、奖金等的影响力，如果上司不能公心公论，那么无疑的后果是严重的。

  上面讲的是最简单的人都会想到的事情，于是很多事情趋于暧昧，琢磨上司的心理成为一种必须功课。这种氛围一旦形成，组织就会逐渐往“一言堂”的方向靠拢。

  所以我们有理由呼唤一种直来直去的风格，软件团队是做开发的，是非面前不需要太多的含蓄。

  这对管理者的要求是，一定要确保没有人会因为直接发表观点受到任何形式的批评。对于管理者而言，破坏一种直来直去的风格，实在是再简单不过的事情。

- 具体与似是而非上的共识

  陶渊明曾在《五柳先生传》中写到：*好读书，不求甚解，每有会意，便欣然忘食*。以诗文而论，首要之事是悟其神韵，所以“不求甚解”是比“咬文嚼字”要高明的治学态度。但对项目管理则恰恰相反，更多的时候是需要咬文嚼字的澄清各个概念的确切含义，而不能不求甚解。

  当我们设立一个目标或分解一个任务，目标或任务的接受者对这个目标的理解越没偏差，团队的行进方向越不会出现偏差。这个时候最可怕的事情是容忍【差不多就是这个样子】这类的想法，这样会导致最终每个人不是在同一个方向上出力。

上面只是列举了一些常见的例子，实际中针对具体工作或团队特色，需要达成共识的地方可能不同 ，但像小利益分配这样的事情确实很多很杂 ，影响又比较微妙，必须达成共识。

而共识本身其实并无对错之分，只要存在并是真的共识。

达成共识的关键手法是以逻辑和事实为基础进行公开讨论 ，而尽可能客观公正的关键点则在于剥离利害关系。

比如说：项目组获得了一笔额外的奖金，明天就要发放了，那么如果今天才讨论奖金分配的原则，就几乎不可能有结论。而如果在没有奖金的时候，事先讨论就会顺利很多。

“深”文化与“浅”文化

团队的文化是有深浅的。如果说“岂曰无衣？与子同袍 。”是一种“深”文化；那工作之余，喝酒打牌无疑的是一种“浅”文化。

以公司这种组织形态而论，如果文化本身不以员工的核心利益诉求为根基，那文化建设往往是“虚情假意”的，得到的也只能是“浅”文化。

只有直接面对员工的人 ，关心各个员工的核心利益诉求 ，才可能形成真正的“深”文化。比如：对年青的员工要尽可能清楚的告诉员工他三年后可以成长为什么样子。

无论在那里，公司与个人之间始终会存在着矛盾。而面对矛盾，要么调和而共同发展，要么激化而斗争，舍此无他，后者则是场灾难。

事实上只有“深文化”才有可能调和矛盾，避免灾难发生。

如果把个人和公司抽象为两极，那么只有在两极上的砝码等重，才可能达到均衡状态。　在公司这种组织形态下，天生的公司一极砝码较重，比如说：运作项目时 ，天生的是项目利益压倒其它很多个人考量 。这个时候如果没有其他“势力”与之对冲，那么最终结局必然是天平不断向公司一端倾斜。这时候，往往就需要

“深”文化来对此进行平衡，否则倾斜到一定尺度，必然矛盾激化导致斗争（离职也可以看做是一种斗争形式）。

2.2.4 逻辑链 4：文化环境之“观点整合”

软件是一种固化的思维 → 思考是思维形成必经之路 → 积极的思考和被动的思考差异巨大 → 组织里需要让人积极思考的氛围 → 形成积极思考氛围的关键要素是：积极思考是被鼓励的 → 积极思考的成果没有被轻易漠视，对个人有逻辑的观点的强制否决要尽可能的少

当一个人认为他的观点被毫无理由的忽略，并被强行拉到了另一条路上时，通常这个人会表现出愤怒，消极等情绪。

如果这是偶然事件，那么时间可以帮助淡化其负面影响，但如果这是一种组织的文化特征，那么无疑的这类情形会对所有人的工作意愿造成致命伤害。

从原因来看，观点分歧大致有两类：

- 一类是有的人认知还不够清楚，比较片面，是盲人摸象式的。
- 一类是认知已经足够清楚了，但视角不同，选择不同，是横看成岭侧成峰式的。

对于盲人摸象式分歧，为避免对工作意愿形成致命伤害，那么做决定的人无疑的要通过逻辑分析把各种认识引导到同一观点上来。这时候事实上要求做决定的人必须对问题的本质有着更深层次的把握。

对于横看成岭侧成峰式分歧，由于并不单纯的是你对我错，而是多种观点同时具有合理性，只不过视角不同。那么，当不选择某些观点时，至少要让有观点的人知道，他的观点没被接受的原因是什么。

上面的简单逻辑坚持的好，最终就会形成真的“民主集中制”：有问题畅所欲言，有决定后集中实施的文化环境；坚持的不好，就会成为“一言堂”：有问题没人说话，做起事情牢骚满腹的文化环境。

在处理观点整合时，比较差的情形是走两个极端：要么是“一言堂”，要么是“绝对自由主义”。从危害来看，这两者一样的差，并无区别。所不同的是人们对“一言堂”模式深恶痛绝，无比警觉，而对“绝对自由主义”却很包容，认为是民主风范，反倒向往。我们来一起分析一下这两种情形：

- “一言堂”与“绝对自由主义”

在“一言堂”的模式下，项目管理者具有凌驾于其他成员之上权威，甚至可以以个人喜好来代替是非标准。

在“绝对自由主义”的模式下，没人能把自身的意志强加在别人身上，体现为议而不决，谋而不断。

如果我们认为，项目管理是一种存在着某种内在的规律的活动，并且只有在人的行为符合了这种内在规律之后，才能把项目管理做好的话，那么无疑的上述两种极端情形都是不好的，他们都使顺应这种规律更加困难。

“一言堂”的状态下，通常很难集思广益，不利于认知的深入，同时会扼杀团队成员的责任感和积极性。一旦责任感丧失，信任感也会逐渐失去，团队的效能也就会一路下滑。毕竟测试驱动开发这类方法论，不是管理人员一个人决定后就可以自然的获得成功的东西，更需要的是群体的合作和努力。

“一言堂”状态非常容易识别，通常有几个典型特征：

一是所有的上司来的要求体现为一种命令，执行者甚至不关注这么做背后的理由。

一是沟通往往是单向的，上司很难收到下属来的反馈。

假设说有一辆车，我们希望它走的快，那么无论推或拉都不是最佳手段（即使推或拉的人的力量非常强悍），只有这辆车能够自行前进了，那么未来才是光明的，否则如何面对 500 辆车的情形。

“绝对自由主义”的状态下，团队成员各行其事，方向混乱，内耗严重，执行力低下，团队效能低下。数千年前的大哲亚里斯多德对这种状态极其反感，他说：多头是有害的，让一个人去统治吧！

这种状态的典型特征是会议比较多，但每次的问题议而不决，决了也没人做。

以项目管理而论，上述两种极端情形都需要被避免，需要向理性进行回归：

- 当一个人是讲逻辑的，那么结合特定情境（时限，人力资源，技术

难度等），事情如何处理大多会有定论，也就是说可以辩明是非，这类情形下无疑不需要强权。而如果一个人是不讲道理和逻辑的，那么更应该把这个人剔除出去，而不是在日常工作中以强权进行压制。

- 当面临各有利弊的选择时，根据权利与责任当为常数这一原则，无疑的职位高的人要做出选择。职位所赋予的权利只应使用在这类场景下。

- 在解决观点冲突时，强权是绝对必要的，但也是最应该避免的。这与软件的特质与组织结构的特质有关。与此同时项目组作为一种相对比较松散的组织结构，并不足以支撑强权的大幅使用。

比如说：在清朝时，主子可以要求奴才既要绝对服从也要主动。这是因为那时候奴才并没有什么选择的权利。而项目组这种结构显然不具备这类“强约束力”，人员至少可以选择退出（离职)或消极（得过且过）。如前文所说，消极本身对强调思考的软件开发而言是致命的。

反过来讲，即使用强权也要求主动也不是完全不可以，但你要有办法缩小人员选择的权利。比如：给超过平均水平2倍的工资。但这似乎很难，也很难持久，并且长线来看，始终是会伤害工作意愿。

权利的来源与类型

管理人员需要对能否提高团队的效能负责，所以他的权利不是来源于他的老板，而是来自他手里的事业。其实这是对事不对人的本源，也应该成为项目管理中使用权利的准绳。

从执行的角度看，事业自身通常也可以表示为一组责任，因此责任和权利之间存在着因果关系。如果确实两者都可以量化，那么两者相除其商应该是个常数。

在专门对权利进行研究的书籍中会对权利的类型进行进一步的划分 :比如把权利分为授予型（granted)和挣得型(earned)权利。挣得型(earned)权利也即是我们通常所说的影响力。在这一节里面我们提到的权利专指授予型权利，即在一个组织中某个人被分配到的权利。形式上讲，大多时候影响力毕竟要通过授予型权利起作用，如果在同一个组织中正式地，同时存在着两个权利中心，那么在大多

时候并不是好事。

2.2.5 逻辑链 5：制度环境之“势”

意识指导行动 → 总体来看，个人意识是倾向于扩张而非收敛（收入，荣誉，地位，自己意志的实行程度）→ 扩张的目标和现实的差距是每个人前进的动力 → 管理的一重使命是确保这种差距存在。

知名学者何新先生讲过一段很有意思的话，他说：

> 人的本性类似于热力学第二定律，即趋向于“熵”的增大方向发展。也就是说生物（包括人），他们的本性就是追求自由，优越感，出人头地等。

熵增意味着个人的意识倾向于扩展而非收敛，这事实上对管理提出了一个课题：每个人所处的现实和他的可见未来之间必须存在着可见的足够大差距，这样“势能”才可能足够大，也才可能推进着一个人积极前行。

热力学第二定律

热力学第二定律可以表述为 :不可能把热从低温物体传到高温物体而不产生其他影响 ;不可能从单一热源取热使之完全转换为有用的功而不产生其他影响 ;不可逆热力过程中熵的微增量总是大于零。

也可以被表述为熵增原理 :在孤立系统中，一切不可逆过程必然朝着熵的不断增加的方向进行，这就是熵增加原理。

但热力学第二定律之所以有名却不是因为其对热力学的影响 ,而是因为其对形而上学的影响。比如说：我们可以把宇宙抽象为孤立系统，这样一来，宇宙最终的结局就是热寂。这不一定对，但对思考宇宙人生的人冲击确实非常强烈。

显然的“势”是一个相对概念 .对于每天吃窝窝头的人 ,吃白面馒头就可以成“势”。而对于每天吃白面馒头的人，多两个少两个则没什么价值。

这无疑的并不只是项目层次要解决的问题，但项目层次也并非什么都不能做。

马斯洛的需求层次理论把人的需求分为五个层次：从低到高分别为

- 【生理上需要，比如：对水，食物的需要】
- 【安全上的需要，比如：人身安全，财产所有性】
- 【情感和归属的需要，比如：爱情，友情】
- 【尊重的需要，比如成就感】
- 【自我实现的需要，比如：对理想的实现】

这里面无论那一项都可以转换为具体的势能，而显然的1,3,4,5和项目中的活动（评价，工作安排，交流）等直接相关。

具体来讲，在物质层面，“势”至少要具体体现为两类东西：一是职位以及收入的提升/下降；一是技能的提升。在精神层面，“势”，则可以具体体现为成就感，荣誉等。

从应用角度看，要结合每个人的现状，性格，背景，公司所能提供的环境来综合考虑，这样才能很好的判断个人的“势”究竟可以体现在那里。

比如说 :对于新毕业生 ,那就要很现实地共同考虑职业路径的下一步究竟在那里，走到那里之后未来会更光明一点。这个时候一定要真诚且具体，否则就是“画饼欺人”，最终只会适得其反。

这里的关键词是具体，但具体并不是很难达到，只要一起考虑一定可以找到合适的答案。在此之前，通常需要对软件所牵涉的各种知识做一个分类，下面来举一个还算具体的例子，来对如何规划职业路径进行说明：

构建软件的直接知识可以被分为三大类：

①用于打造概念边界并且优化概念间逻辑关系的知识。比如，面向对象分析和设计。
②用于实现概念和逻辑的通用领域知识。比如编程语言。
③用于解决指定领域的专业的领域知识。比如图形算法。

在很多场合③并不是必须的，但①和②则是不可分割的，并不存在孰轻孰重的问题。没有概念无法形成逻辑，没有逻辑，概念的彻底定义更是无从谈起。而没有载体（比如编程语言或UML）概念和逻辑根本就无法表达。事实上编程语言的演化很大程度上是想把这种表达变的更容易。

而与软件有关联的间接知识则有4大类：

①需求开发和描述。比如规格说明的编写。
②估算。比如功能点方法等。
③测试。比如验收测试等。
④软件工程和方法论。比如 CMMI 和敏捷。

根据，上面的分类，我们可以制作一份分类的表单。

关于软件的直接知识：

- 通用的领域知识
  - ✧ 编程语言（C/C++，Java，C#，Python，Perl 等）
  - ✧ 框架和类库(MFC，Boost，Struts，Hibernate 等)
  - ✧ 平台(Windows API，POSIX，.Net Framework※1，Java API，C/C++ Runtime Library 等)。恰如 Jeffry Richter 所说，大多时候可以从内存机制、线程机制、错误处理、异常处理、组件构建、组件组合等方面来进一步考察一个平台。
  - ✧ 计算机体系结构(CPU 指令，虚拟存储等)
  - ✧ 实用技巧(调试方法，代码生成器等)
  - ✧ … …

  ※1 有的时候子类别间的界限并不是很容易界定，其中一个主要原因就是存在着像.Net Framework 这样涵盖了过多内容的概念。

- 概念和逻辑创建和优化
  - ✧ 面向对象分析和设计/结构化分析和设计
  - ✧ 设计模式
  - ✧ 重构
  - ✧ 契约式编程
  - ✧ UML ※2
  - ✧ … …

  ※2 从形式上来看 UML 更近似于一种编程语言，但从其目的上来看也许归在这里是更合适的一种选择。

- 专业领域知识
  - ✧ 图形图像算法
  - ✧ 网络协议
  - ✧ 人工智能

- ✧ 数值/非数值类算法
- ✧ … …

关于软件的间接知识：

- 需求开发和描述
  - ✧ … …
- 估算
  - ✧ 估算法。比如，COCOMO, FP 等。
  - ✧ 估算术。比如，使用计数等原始办法。
- 测试
  - ✧ … …
- 软件工程和方法论
  - ✧ 轻量型方法论。比如敏捷。
  - ✧ 大方法论。比如 CMMI
  - ✧ 综合分析。比如，《人月神话》，《人件》所做的工作。

在这份表单里面，与管理相关的内容并未被列入。但基于这种分类已经可以设计一条很直观的，3~5年内可用的职业路径，如：

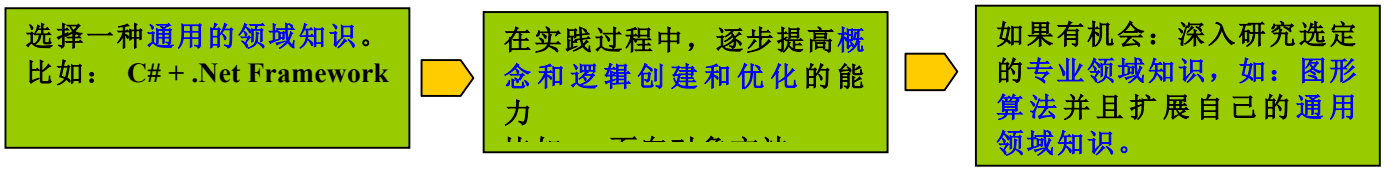


图 2-4：职业路径的例子

这时候要考虑当前的工作是否能支持既定的职业路径，比如：如果一个人当前的主要工作是使用C++做底层驱动，那么选择C#就不适合。

人是很复杂的复合体，这就导致在不同情景下精神诉求会有很大的不同，能成势的点也很不同。完全漠视这类个体的特殊性，而简单的把每个人都等价为相同的人，并只是信于流程这样的手段，事实上是管理上的惰性。

可以讲，“势”在很大的程度上决定了团队的活力和向前走的动力。同时“势”也是双赢的一个关键基础。以职场而论，公司的诉求往往单一而清晰，大多时候

是以平衡短期及长期利益为根本诉求；但个人的诉求却往往复杂而含糊，每个人的诉求即有共通之处，也有差异。如果在项目管理这一层次上漠视这种差异性，那个人诉求很可能就永远停留在不可见的区域里，这还谈什么双赢。

欲望的无边界特质

欲望的起源有很多，但最终大多以思维为媒介表达出来。在这一过程中，思维的的无边界特质直接导致了欲望的无边界化。比如说：一个人可能期望拥有整个宇宙。这最终导致到达一定程度之后，物质力量将不足以成势，精神的事情还是要回到精神层面来解决。这个时候人们很可能更需要成就感，需要自我实现的空间。但对刚毕业的人而言，这些则相对遥远。

2.2.6 逻辑链 6：制度环境之“量化管理”

软件是一种固化的思维 →思维的本质是概念和逻辑 → 概念和逻辑无法直接度量和精确度量 → 度量过程中需要很多的主观判断 → 以目标为导向的，以个人为中心的量化管理（相关的激励和惩罚）将崩溃 → 参照无歧义数据（函数复杂度等）的判断将成为程序员评价中的辅助手段

公平公正是管理的基石，为达成这一目的很多人会想到量化管理，但在现实中量化管理的基石却往往被忽略。

对人进行量化管理的基石是：量化后的数字主要受个人表现这一个因素的影响，否则将产生巨大的不公正，并对个人工作意愿产生不良影响，是真正的事与愿违。

好比说，不同的工人在同等条件下生产杯子，一个人一小时生产5个，一个人1小时生产6个，那显然后者要好于前者。这时，5和6可以用来比较的前提是两个人的生产条件相同，比如生产工艺等。在这种情况下，量化后的数字为个人表现的函数，因此量化管理基本上是公正的。

这时可以进一步来考虑下面的情形：两个人同时生产杯子，厂方安排一个人用工艺a，另一个人用工艺b，这个时候前者一小时生产5个，后者1小时生产6个。这

个时候单纯比较5和6事实上是不公平的，因为这1个杯子的差距可能是工艺造成的。

大多时候，软件的情况比后一个情形还要糟糕一些。在软件开发中，往往既没有办法清楚的界定一个人的输入，也没办法清楚的界定一个人的输出。

软件开发的输入是需求，对需求自身的复杂程度眼下来看还只能依赖判断，而不能精确度量，现实中并没有一种有效方法用以度量需求。

软件开发的输出是代码，而代码自身属于固化后的思维。在度量思维时，多少、大小、长短、厚薄这类惯常的度量方向，并不具有多大意义。就好比说，不能讲一个人代码写的越多贡献就越大一样。

诚然思维的表现形式是可以度量的，我们可以通过页数来度量一本书的厚薄，通过分钟来度量一部电影的长短，通过代码行来度量软件，但这种度量所反映的内涵是有限度的，精度也是有限度的。最终结果很可能是人员之间的差距是由误差或其他非主观因素造成的，而不是由个人工作好坏所造成的。

总结来看，在软件开发中，数字含义的模糊性会导致使用数字进行评价包含非常多的不公正，这种不公正会对工作意愿构成致命伤害。所以个人层面的量化管理在软件开发面前，必然崩溃。

但也不是所有数据都不能用于评价，无歧义数据是可以的。比如：函数的圈复杂度、对既定编码风格的遵守等都可以根据静态代码工具获取，这些数据是可以用来辅助评价的。但这些数据由于不能完整的表现一个人工作价值，所以只能成为判断的辅助手段。

为避免矫枉过正，最后需要强调的是：并不是说项目管理中不需要收集数据，只是说在软件这个行业中，各种数据的精度天生是有限的，因此必须用在允许有限精度的工作上（估算、任务安排等），而不能用在对人进行评价、对项目进行评价这样需要高精度数据的工作上。数据所对应的精度与其所适应的场景之间有着必然的关联。

换句话说，任何数据其天然模糊性所导致的偏差和其它输入（个人效能等）导入的偏差如果很容易区分，那么量化管理可以使用这些数字，因为数字可以直接映射到问题；否则数字不是唯一标准，因为分不清数字的真实含义。

软件项目中的评价

很不幸，基于这一节里的结论，软件项目中的评价只能基于判断。我们可以对待判断的工作进行重重分解，比如：把工作分拆为日常工作和具体项目工作，项目工作又可以进一步分拆为需求、设计、编码等，但归根到底还是判断。

剔除主观因素后，正确判断的基石是对事实的理解，这一点反过来限制了可管理软件团队的规模。假设说，一个人可以比较清楚的了解九个人的工作，那么可管理的软件团队规模必然为十人左右。大的软件团队则需要拼接很多个这样十人左右的团队。

一旦规模过大，由于评价者对事实解的不够清楚，往往就会偏颇。比如：会因为某人智力、想象力、知识渊博等而印象较好，进而给出较高的评价，但事实上考评更应该以成果为唯一基准。

2.2.7 逻辑链 7：内耗之终结

软件是一种固化的思维 → 就思维上大多的问题而言，基本上是一题多解 → 组织结构要确保各种情境下都能做出尽可能正确的决断，但又不能伤害个人意愿 → 层级过多，会导致做决定的人离现场越远，对真实的情况把握越不清楚（信息在传递时会损失），同时误判增加，对个人意愿伤害增加→ 组织应该尽可能扁平 → 为避免谋而无断，即使扁平，也要有能做出决定的人

三国演义中，曹操对袁绍的评价是：色厉而胆薄，好谋而无断。这很有意思，因为这种小说中的个人表现在现实中不断重复出现。如逻辑链4所述，充分尊重个人观点是保持工作意愿的必要条件。当需要协调的人数比较多的时候，事实上会出各种各样的观点和意见，

这时候协调各种意见成为一项挑战性比较强的工作，如果不能迅快的协调各方意见，那么组织很容易就变成袁绍型组织：会议很多，工作没进展，进而产生巨大内耗。

为避免无休止的内耗，并尽可能做出更正确的决定，组织结构上来看，关键点有三个：层级的多少，权责的对等与专权。

- 组织中的层级越多，参与解决分歧的人数越多，效率越差。

层级过多的一个坏处还在于信息在传导过程中损失。即使只基于常识，大多数人也可以理解信号在传导过程中会逐渐衰减，如果不采取措施，那么原初的信号会最终消失，所以在网络中需要【中继器】存在。信息传递亦是如此。

信息在传导过程中自身就有失真的倾向。语言或文字作为一种记录信息的手段，所能记录的远不是待传递信息的全部。

表达有如冰山，总有一个水下部分存在，比如个人背景等。而传导路径越长，隐含信息也就越容易丢失，其失真的程度也就越大。更不幸的是组织之中往往并没有【中继器】，因此尽可能的缩减层级成为防止信息失真的一个有效手段。

在组织中，通常是离现场越远的人决策权越大，这本身并没有什么问题。关键是一旦现场来的信息被过度扭曲，决策自身将变成无根之木。

以项目管理而论，需传递的细节信息较多，而各种决策并非高层决策，不允许离这些细节太远，因此大多时候都要尽可能维持 2 或 2.5 级结构。《人月神话》中提到的外科手术式团队是典型的两级结构。但实际中有时候 1 个人没法肩负起管理和技术两重角色，这时候就需要让渡出一部分权限，进而形成 2.5 级结构。

- 与层级所并列的第二个影响因素是：权责的对等，即两者的比例值应当为常数。

举一个极端的例子来对这一点做点说明。

正常来讲，设计的主要负责人有权利决定最终的设计方案。这个时候作为整个项目负责人的项目经理，有权利推翻设计负责人的决定，并选定其他方案，但是一旦项目经理这么做，那么就意味着项目经理行使了决定设计方案的权利，那接下来的任何根于这一选择的不良后果的主要责任人也就变成了项目经理，而非设计负责人。

权责不统一的短期后果可能不严重，但却对形成团队的基石：责任感有致命的负面影响，而没有责任感的团队必然是没有战斗力的。

- 与层级并列的第三个影响因素是：组织并非是多头的。

我们看一个极端的例子。假设说一个团队中存在着两个经理，一个偏向于技术，一个偏向于管理，两人层级相同，互不统属。

这种情况下，坏处有两个，一个是项目信息将被割裂成两个部分。两个

经理必然要做责任划分，一旦划分之后，就不再有一个人对项目整体情况有完整的了解，但决策实际上同时需要两个维度上的信息，这必然会增加判断出错的几率。另一个坏处是两个经理对某些事情看法上可能会有分歧，这种分歧可能并不关键，从项目自身来看有可能都是可以接受的，但在不好的氛围下，让任何一方放弃自己的想法，都可能需要相对比较高的成本。

PMBOK中把组织分为如下一些种类：

组织结构 / 项目特征	职能式	矩阵式			项目式
		弱矩阵	平衡矩阵	强矩阵	
项目经理权限	很少或没有	有限	少到中等	中等到大	很高，甚至全权
可利用资源	很少或没有	有限	少到中等	中等到大	很多，甚至全部
控制项目预算者	职能经理	职能经理	职能经理与项目经理	项目经理	项目经理
项目经理角色	半职	半职	全职	全职	全职
项目管理行政人员	半职	半职	半职	全职	全职

表 2-2：组织的类型

其中【职能式】组织，对增强组织透明度和灵活支配人力资源应该是有帮助的，但无疑的会使事权分裂。这个时候如果诸多当事人间纷争不下，那么内耗有可能无法控制。

资历的力量

资历往往让人想起论资排辈，进而给人一种不好的印象。但不管我们对其印象如何，资历自身确实是一种不能轻易忽视的力量。在组织中，当其他因素无法明显区分彼此的差别时，人们往往潜意识的选择服从于资历。年纪大的领导年纪小的，入公司早的领导入公司晚的，天生的就会少些纷争和矛盾，有助于削减内耗。但一旦组织中出现孙悟空这类具备大闹天空能力的人，就会对资历所构成的序列形成冲击和挑战，反过来造成更大的内耗。这背后，本质的原因是资历序列和能力序列并不相同。如何平衡两者大多时候并非是项目管理这一层面需要处理的问题，只是说资历作为一种现实存在的力量，并不能被彻底的忽视。

2.2.8 逻辑链 8：沟通之成本

软件是一种固化的思维 → 思维自身具有迭代特质（否定之否定）→ 思维的主体必然是人，思维的对接也是只有人才能完成的工作 → 项目越大，需要的迭代也就越多，必须的人和人间的沟通量也就越多 → 参与沟通的人越多，需要协调的不同个性的人也就越多，效率也就越差 →所以软件生产往往表现为规模不经济。同一个软件，一个人开发，效率最高，人越多效率越差 → 尽可能控制团队规模 （控制沟通成本）

在充分尊重个人意愿的前提下，解决内耗、进行决策这一行为本身是需要成本的。这一成本主要受两方面的因素影响：一是观点分歧次数的多少，二是解决每次分歧所需要的平均成本。

如果软件的复杂度确定，人员的基本特征确定，那么观点分歧的次数事实上是软件规模的函数。也就是说软件规模越大，可能产生的观点分歧越多。如果我们认为软件是可进行分解的，那么这种数目的增加将是线性的。

解决每次分歧所需要的平均成本则和参与的人数（组织层级），人员对既有问题的熟悉程度及人员间共识的程度有关。一般来讲陌生人间，对陌生问题的协调需要更多的时间。从这个角度看，参与的人员越多，解决每次分歧的平均成本越高。

上述两条可能是软件规模不经济的根本原因。

我们考虑下面两种情形：

情形1： 软件的规模由原来的n膨胀为2n，为了确保进度，人员规模从原来m也膨胀成为2m。

情形2：软件的规模由原来的n膨胀为2n，但由于没有时间压力，人员规模保持不变。

依据上述逻辑，情形1会表现出规模不经济之特质，而情形2不会。针对这一点，眼下还找不到有效的支撑数据，所以还仅止于一种假设。

那么为了使决策成本最低，究竟什么样的团队规模才是合适的？单纯从效率的角度看，一个人来做效率最高，但毕竟这不安全，同时大多时候无法满足时限上的要求。

为回答上述问题，我们需要重新再做一点假设：

在一个项目中管理所占的成本：假设这个值的上限是10%。

在一个项目中决策人只有1个，否则需要多人协调，会降低效率。

基于上述假设，可以推断：最佳团队规模是10个人左右，其中1人为专职管理者。

真理往往掌握在少数人手里

人的思辨能力就像一座座山峰，高度越高，人数越少。比如说：今时今日，也未必很多人读懂黑格尔。

如果我们认为掌握真理，并不是只靠运气，而是主要靠思维的能力，那无疑思辨能力强的人更容易掌握真理。因此甚至可以说：真理一定掌握在少数人手里。

2.2.9 逻辑链 9： 组织行为之优化

重复是降低工作效率的 → 工作分解必须是正交的，并且时序合理，符合因果关系 → 项目的资源是有限的，必须合适的人做合适的事情 → 为达成这一目的，项目管理者必须既理解所做的事，也理解相关的人员特性 → 为避免遗忘，使经验转化为价值，普适于多个项目的、特定的因果关系和时序需要被提取出来并强制实行，最终形成流程

事实上，很多人心中的项目管理所涉及的内容都和这条逻辑链相关，比如：

- 任务的分解和分配
- 变更的统计，跟踪和分析
- 缺陷的统计，跟踪和分析
- 时间偏差的统计和分析
- 投入人月的统计和偏差分析
- 代码质量的跟踪
- 配置管理规则的实施，跟踪和分析
- ...

在去除人的因素之后，所有这些任务的主要目的都是为了保证任务的分解和分配是正交的，同时在时序上符合因果关系。正交是指相同的工作没有以任何不同的形式做两次。比如：统计人月分布时，不需要以周为单位统计一次，再以月为单位统计一次。符合因果关系是指互成因果的两项工作没有错漏倒置，自相矛盾等。

常见的自相矛盾的行为有：

- 期望一个人工作效率高，但却让人同时处理多份工作。每个人每天有八个小时，如果他切换八个工作场景，那等于至少被打断八次，能用心做某一份工作的时间不足一小时，所以这类分配方法几乎必然是降低效率的。
- 期望推进进度，但工作负荷在时间轴上却相对不平均，忙的时候极忙，而闲得时候极闲。这类矛盾会导致先是很赶，写了许多质量不高的代码，接下来又在清理不良代码上花费更多的时间，最终真的成为欲速则不达。比如：10个人月可以完成的项目，却花了15个人月。
- 期望保证每个人工作的时候有明显的焦点，但却无法放弃可做可不做的事情。项目中的工作潜在的有变繁复的趋势，不管是输出的文档、会议次数和议题，还是流程的步骤。需要增加的东西必然是有一定合理性的，但工作变繁复的同时，必然也就意味着集中在某一项工作上的时间需要被摊薄。最终在不知不觉中反倒对项目自身造成伤害。这个时候，一个关键的事情是挑出那些不做会死的事情，把他们排到关键路径上，确保他们的资源，给予他们最大的关注度，同时尽可能放弃那些可做可不做的事情。
- 忽视历史数据。比如：历史数据显示编码的速度（不是生产率）大概在100SLOC/Man*Day。估计为10000SLOC的程序，却只保留了20Man*Day的工作量。
- 知道潜在有风险，却不在早期安排分析和应对。比如：C++开发的程序却不安排内存泄露和写超界的检查。

违背上述两条基本要求的组织行为即是错误的组织行为，会浪费人力和时间。

总结来看，避免错误的组织行为事实上有3个根本措施，那就是：参照历史，分配合适的人去做合适的事情，构建信息回路。

- 参照历史是指要对支撑规划的数据进行定义，并持续收集，持续使用。

  对大多软件项目而言，必须收集的数据大致有：规模、生产率、编码速度、各阶段的工作量比率、各阶段的时间的比率、缺陷率等。

  这些数据不能成为孤立的数据，其对应的项目特征，数据自身的计算方法都需要被明确定义。然后把这些数据应用到接下来的规划中。

- 分配合适的人去做合适的事情是指人员特征与工作相吻合。

如果对人的个性进行细分 ,就会发现有的人擅长沟通但不擅长技术；有的人擅长技术但性格比较软弱，容易人云亦云；有的人不聪明但足够执着；有的人能力不强但足够细致；有的人脑子灵活但很粗心等。

运作项目时需要避免两种危险的假设：第一是不能假设团队成员都很完美；第二不能假设团队成员可以改变。这两个假设，前者幼稚，后者癫狂，一旦在这样的前提下运作项目，那么项目几乎是一定失败。这个时候为避免内耗，首要因素则是知人善用，用人所长，避其所短。

而达成人员分配合理这一目的，并无捷径，它要求作为分配任务的人，不仅需要了解人员的特质，也需要了解工作的特质。从这个视角看，管理者是一定要懂技术的。

- 构建信息回路是指做某事的人（一个或多个），要能收到关于某事的反馈信息。在这一原则下，任何事情可以被分解为五个基本步骤：计划(P)，实施(D)，检查(C)，分析(A)，改进(I)。其中 PDCA 四步就是著名的戴明环。

这看似简单，但这是“流水不腐，户枢不蠹”的真意，而回路断裂，则是很多问题的根源（没执行力等）。

比如：修一条公路时，有的单位规划和招标，有的单位施工，开车的体验。结果路坏了，没人真正分析，只是接下来继续修，结果必然是路总也修不好。

通过回路存在与否可以大致判断组织的成熟程度。

比如：有的组织中，项目一个接着一个从来不做总结，那么这个项目组所在的组织一定是不成熟的。

再比如：一个人，总是不停的看书，写程序，却从不停下来思考，那么这种学习方法是不成熟的。

当某些组织行为具有共性时，通常它们需要被固化下来，来避免任务的不正交和因果上的混乱，最后这会形成流程。在下一章中会对流程进行专门的解构，这里不再展开。

冲虚道长与令狐冲

在金庸先生的《笑傲江湖》中，令狐冲为了去少林寺搭救盈盈，必须与武当的冲虚道长一战。

冲虚道长的太极剑法高妙无比，每一剑出，就是一个弧形，战到最后，所幻光圈越来越多，冲虚道长自身则隐在光圈之内。这让横绝天下的独孤九剑几无用武之地。虽然后来令狐冲行险取胜，但对太极剑法的精妙亦是拜服。

上文提到的计划(P)，实施(D)，检查(C)，分析(A)，改进(I)环，很像太极剑法里的圈和弧。独孤九剑是用来培养绝世高手的，可遇而不可求；而太极剑法则是兴堂堂之师，击煌煌之阵，是奇正相合中的正。

这个 PDCAI 圆事实上可以用在项目中的各个地方。用在项目管理中，那可以表现为计划、实施、检查、总结和 Action Item。用在程序开发中，则可以表

现为：设计，编码，编译，调试，修正。在《大规模 C++程序设计》一书中作者提到了典型的环连接的不好的情形：大规模项目编译速度可能变得很慢，这导致项目推进困难。而编译慢实质上断开了实施(D)和检查(C）间的连接。

但这似乎太简单了，以至于很多人都以为自己了解了，并毫不重视。老子曾讲：上士闻道，勤而行之；中士闻道，若存若亡；下士闻道，大笑之。不笑不足以为道。这反倒契合很多情境。

§2.3 完美项目管理

管理是一种实践。

--Peter Drucker

2.3.1 完美项目管理的形象

形象来讲任何一个团队都像一辆行走中的车。对这辆车而言，为了快速达到目的，三件事情最重要：

- 要走对方向
- 要跑得足够快
- 不能散架或翻车

为达成这三个方向上的目标，管理有两类核心工作要做：一为管人，一为管事，而管事则一定程度上是为管人服务。

当组织中的人员达到自为（共识较多）的状态时，管理的大部分意义和功用将会消亡，因此：

- 从方向性上来看，管理的完美状态一定要体现为管理自身的工作量可以持续降低。从反面来讲就是，如果组织中需要管理的事情越来越多，那并非证明管理越来越有效，而是佐证了管理正逐步走向失败。管理越来越少等价于说行走中的那辆车，既不依赖于推力也不依赖于拉力（激励、批评等），而是靠着自身的动力在行走。

- 从个人的角度看，组织中的个人要越来越倾向于自我管理。在意识层面上能够对自己所做的事情负起责任，同时积极思考应对各种变化（工作意愿走高）。在实施层面上能够对自己的工作进行恰当的分解，跟踪和总

结。

- 从管人的角度看，要保证开放轻松的氛围，尊重所有有观点的人的意见。要保证投入产出的公平和公正。这里的公平公正是指：
- 对于任何一个成员也可以用一个简单的坐标系的四个象限来进行划分，这个坐标系的横轴和纵轴分别是：能力是否能胜任既定的工作范畴；精神及价值取向是否与团队整体的价值取向相吻合。

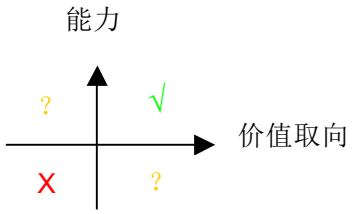


图 2-1：公平的一种定义

公平公正意味着，第一象限的人必须受到奖励，第四象限的人的方向性必须受到鼓励，第二象限的人，其方向性需要受到批评，第三象限的人应该受到惩罚，甚至淘汰。这是杰克.韦尔奇在《赢》中表达的观点，对项目管理也有很强的参照意义。这时一定不能拖泥带水，不能“善善而不能用，恶恶而不能去。”

【逻辑链 1：意愿之价值】和【逻辑链 5：制度环境之量化管理】则提醒我们在软件项目做到公平公正是何等的困难。一方面贡献值受意愿影响巨大，一方面却又没办法精确度量。

- 从管事的角度看，组织规模尽可能维持在 10 人左右，使组织尽可能扁平的同时确保有一个人处在塔尖上。要确保权利和责任之比值尽可能为常数。要确保恰当的人在做恰当的事情，与此同时在任务分解和分配上达到正交和没有因果上的矛盾。

我亦无他， 唯手熟尔

卖油翁把一个铜钱放在油葫芦嘴上后，通过铜钱中间的小孔向油葫芦里倒油时，油竟然一点也没沾到铜钱上。这很让人惊奇，但卖油翁对此的表示是：无他，唯手熟尔。

项目管理需要思考，但也需要当事人的手熟，要熟记软件开发的基本步骤和对应的数据（比如：多大规模的程序，大概产出多少页的需求文档，多少设计文档等），要清楚团队成员中每个人的个性特点，技术能力。这些东西并不蕴含太高的难度，

需要的是用心。

2.3.2 完美项目管理的关联要素

管理等于是为软件开发准备了人以及做事的平台，在此之上才能启动真正的软件开发。

在其他环节之中，与管理最为贴近的是流程。从一定程度上讲，流程甚至可以算作是管理的一种手段，在逻辑链8:组织行为中，我们曾提到了：

为使经验转化为价值，普适于多个项目的，特定的因果关系和时序需要被提取出来并强制实行，最终形成的就是流程。所以完美的管理方法需要完美的流程作为支撑。

对管理影响最大的是估算。有了估算的结果，才能有资源分配这类事情的发生。所以完美的管理方法事实上是需要完美的估算作为输入。如果实际需要100MM，估算结果却只有50MM，并按照50MM分配了人力，那管理难度必然倍增。

如果把管理抽象为一个处理器，那么这个处理器最关键的输入是人员特征和项目特征，这两点对把握管理中各个环节的尺度至关重要。

比如说： 对于比较简单，但规模比较大的软件和难度比较高，但规模小的软件，其管理方法可能会不同。虽然遵循相同的逻辑链，但在尺度上将有所差别。

从输出来看，管理的主要输出是人的状态，次要输出是各种计划。

如果团队中有下面这样的现象，那基本上是管理没做好：

- 团队整体处在得过且过的状态，没人会去主动考虑，那里可以做点改进，但抱怨很多。
- 出事后互相推诿。
- 随便做什么事都纷争不断，形不成统一认识。
- 一旦开始做事，大家各行其是，临时起意的事非常多。
- 总是犯些显而易见的错误。比如：事到临头才发现缺的软件没买。
- 日程表和实际根本不靠边，完全执行不了。比如：一个人常会被分配并行做多个任务，但每个任务都要求每天投入8小时。
- 多次犯同样的错误。
- 流程上定义好的事，一做就变样。
- … …

Outsourcing的影响

Outsourcing即是常说的外包。外包与其说是一种分工的手段，倒不如说是一种纯粹的商业行为。从商业的视角考察外包，是一个非常复杂的话题，很难详细展开。但从工程角度看，至少有一件事情是非常清楚的：外包事实上增加了管理的难度。这种难度主要来源于两个方面：

一是权利和责任的分化导致很难遏制代码质量下滑；一是地域分散所带来的各种内耗。

对于一份代码，其非功能性质量（如：可维护性等）事实上是一个有机体，需要持续维护。然而这一部分质量并不能很好的量化（逻辑链6），也就并不能很好的对应于人件费用，这会导致启用外包时会出现有心的无力，有力的无心这样一种局面：发包的有心，期望收获非功能性质量，但没力量去做，而接包的有力量去做，却没理由这样做。这必然会导致代码内在质量的下滑，但这是权利和责任分离的一种必然结果，暂时还看不到彻底的解决方法。

与此同时，团队成员地域分散必然会导致沟通效率下降。对此COCOMOII的考查结果是这会对工作效率产生56%的影响。也就是说同一地点的团队，其工作效率可能比分散在很多地域的团队高56%。

第七章 完美设计和编码之解构

如果我们把设计和编码中的各种问题看做种种矛盾，那么我们就会发现，在设计和编码环节，矛盾极多。而与此同时在限定的资源和时间约束下，解决所有的矛盾几乎是不可能的任务，这也就意味着在所有矛盾上平摊权重是极为平庸的处理方法。更合适的做法则是找到其中的主要矛盾，集中资源进行解决，这样一来由其衍生而来的各种矛盾就会自然消解，无法消解的也只能承受，也就是说需要带着明知的缺陷前行。

在这一部分里，我们的使命就是找出主要矛盾，以逻辑为武器，发现消解之道。

让我们带着下面的疑问，开始对设计和编码的解构。

为什么所有的设计原则都遵守了，最终代码的结构还是不好？

为什么面向对象流行这么久了，仍然有人对其不屑一顾？

为什么理解面向对象的代码时，反倒会觉得困难？

为什么有人认为编码也是设计，而有人认为不是？

为什么有的人坚持认为“Big Up Design Front”，有人却坚持测试驱动开发？设计究竟是要做，还是不要做？

§.1 设计、编码和文档间的关系

道术将为天下裂。

--庄子，《天下》

.1.1 【设计 = 编码】 VS 【设计 ≠ 编码】

在1992年，Jack W.Reeves发表了一篇名为：Code as Design的文章，这篇文章可以在《敏捷软件开发 原则、模式与实践》一书的附录中找到。

这篇文章的核心观点是：编码也是设计，而软件开发中与建筑行业中的施工所对等的工作，已经被编译器代理了。

这是几近20年前的文章，但时至今日，类似的争论仍未休止。解释这一问题并不复杂，但需要用到一点辩证法。我们可以讲：设计即是编码，也不是编码。

在前文我们曾经一再论及，软件是一种固化的思维。

从这一角度看，软件构建的核心步骤只有两个：一是明确固化什么，二是对思维进行固化。设计和编码确实都属于第二步，因此说设计即是编码也没什么不对，他们本质相同。

但分类的时候，有一个很有趣的现象就是：区别个体差异的往往并非该物种最本质的特征，而是某些微小差别。比如区分不同人的，并非心脏，神经系统，而是肤色，脸型等等。

当软件出现之后，人们定义设计，编码这样的名词时，所想到的估计并不是他们本质上一样不一样，而是他们那里不同。

设计和编码的相同点在于它们本质相同，不同点则是它们考虑的问题层次不同。也就是说考虑架构和考虑某个函数的实现时，本质并无差别，有差别的只是层次。从这个角度看，讲设计不是编码也没什么不对。

如果我们认为思维固化过程中确实需要层层分解，而这种层次是连续的，那确实很难讲清楚，从那个层次开始就不是设计而是编码了。

所以这种争议本身，起源于词汇自身的定义，并不是特别的有意义。在这本书里，我们强调的是思维的固化，因此并不会努力区分设计和编码的边界，而认为他们是同一工作的不同层次。

设计处的层次较高，但服务的对象却是更底层的编码，毕竟只有最终的代码才与软件等价，只有好的代码才代表好的软件。只是现实中这种依赖关系往往被倒置，变成了设计指挥编码。

关于软件架构设计

与设计相关的概念里最吸引眼球的应该是软件架构设计。但对什么是软件架构设计事实上并没有定论，反倒是从架构设计所要做的事情上更能看清什么是架构设计。

在《软件架构设计》一书中，温昱先生提到了5视图法，这可以让人比较清楚的一窥架构设计的概貌：

- 逻辑架构

  逻辑架构关注功能。不仅包括用户可见的功能，还包括为实现为实现用户功能而必须提供的“辅助功能模块”：它们可能是逻辑层、功能模块和类等。

- 开发架构

  开发架构关注程序包，不仅包括要编写的源程序，还包括直接使用的第三方 SDK 和现成框架、类库，以及开发的系统将运行于其上的系统软件或中间件。

- 运行架构

  运行架构关注进程、线程、对象等运行时概念，以及相关的并发、同步、通信等问题。

- 物理架构

  物理架构关注“目标程序及其依赖的运行库和系统软件”最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。

- 数据架构

  数据架构关注持久化数据的存储方案，不仅包括实体及实体关系的数据存储格式，还可能包括数据传递、数据复制和数据同步等策略。

--温昱，《软件架构设计》

这一分类的好处是让人比较容易的找到架构设计的切入点，坏处则是把架构设计这一工作无边界化。按照这种归类方法，几乎没什么不属于架构设计的范畴。

我们这里的归类与上述略有不同。

本书认为持续**打造概念的边界和定义概念间的逻辑关系是**设计和编码的核心任务，其它方面的内容对这一核心任务形成约束。也就是说，认为上面5视图法中的逻辑架构和运行架构 ,数据架构是设计的核心任务 ,而开发架构和物理架构(乃至其他）则只是约束的一种。与之类似的约束还有很多，要根据需求自身来逐一考虑。

这与Brooks在《人月神话》中所陈述的观点更为类似，Brooks认为：

所有软件活动包括根本任务----打造构成抽象软件实体的复杂概念结构，次要任务----使用编程语言表达这些抽象实体，在空间和时间限制内将它们映射成机器语言。

本书中认为开发架构和物理架构这类的约束诚然必要，有时甚至很关键，但更类似于Brooks所说的次要任务 ,其复杂度要逊于打造概念边界和定义概念间的逻辑关系。

比如说，当我们要开发一个网站的时候，是否选择Hadoop作为基础平台无疑是非常关键的，甚至可能决定产品未来的成败。但究竟选择那个平台却绝对不是软件开发中要解决的本质问题，否则就等价于认为软件开发是一种组装工作。平台自身并不是软件，只有平台上的开发东西才能决定软件的差异。

因此本书中并不会去探讨：如何考虑究竟是自己开发还是使用现成（开源的或者购买）的产品？依据什么样的原则来部署软件到不同电脑？如何组织代码的文件和目录结构？如何保证编译速度最佳？如何选择合适的数据库等？

但这并不意味着开发产品时判断应该使用什么框架 、使用那个数据库等并不重要或者意味着这种选择很简单，而只是说这是另外一个非常独立的话题，需要一些特别的考量，比如说：框架与需求的匹配程度，是否是开源的，是否可以得到良好的支持，license是否合适等，而本书中不会对此进行覆盖。

.1.2 文档的角色

文档主要是用于记录设计的结果，所以我们可以讲，文档与设计是同一个东西，也与代码是同一个东西。

当同一样东西用两类方式记述时，代价是比较大的。比如：维护两者间的一致性等都要消耗额外的人月。

文档所涉及的层次越低，涉及的细节越多，变化发生时，所引起的同步工作量也就越大，但文档并非没有价值。

代码之中的抽象是可以分为不同的层次的，有很多时候，我们需要在不同层次上审视软件的不同侧面。比如暂不关注类的细节，而关注所有类之间的静态关系。这个时候，直接使用代码将变得困难，因为代码自身并不区分层次，包含所有细节。

比如说 :打开数据库→ 查询→ 处理结果这样非常简单的核心逻辑往往会混杂在判断输入是否有效，创建SQL语句，逐行处理数据等诸多细节之中。

理解代码时，往往并不能一下子把握所有细节，而需要从大往小了看，这个时候文档是有作用的。

除此之外，在多人协作时，文档有助于统一认识，避免遗忘。

从这个角度看，与设计相关的文档远不是越细越好；而是应该关注于较高层次上的概念和逻辑间的关系。这里的“较高层次”似乎很难直接定义，但以现实为基础，换个视角后，却并非不可推断，比如说：

- 设计文档自身最终应该要支持一定的分工，一旦分工必然会涉及不同人负责的不同模块的交互。为支撑这种分工，那么不同人所负责的模块的接口应该被定义清楚，否则会导致过多的交流。
- 多线程或多进程结构需要预先理清数据流，因此需要预先讨论数据流和时序间的关系，为能够进行讨论，这部分内容需要文档化。
- … …
-

此外像重用的要求、可测的要求也都可以成为文档必要程度的一种度量尺度。

.1.3 设计知识归类法

软件行业中新名词已经多到了让人疯狂的地步，比如：

- 框架(Framework)，架构(Architecture)
- 面向对象分析和设计(Object Oriented Analysis and Design)
- 设计模式(Design Pattern)
- 契约式编程（Design by Contract)
- 测试驱动开发(Test Driven Development)
- 面向方面的编程（Aspect Oriented Programming)
- 模型驱动架构（Model Driven Architecture)
- 基于组件的开发（Component-Based Development）
- 敏捷软件开发(Agile Software Development)
- 元编程（Meta programming)
- 面向服务的体系结构（Service-oriented architecture）
- ...

可以想见这个列表在可见的未来，仍将无限的增长下去。

就像把狗毛和狗划分为并列的类别会引起思维的混乱一样，对种种新名词如果没有自己的归类体系，那思维很容易陷入混乱，而思维混乱的结局必然是当事人的无所适从---既不知道从那里开始学习，也不知道究竟应该怎么开始应用学到的东西---这时候往往是即不知道应该做什么，也不知道应该不做什么。

读书或学习的时候，如果自己对待学习的知识有一个大体的认知，那么学到的东西各归其类，学问和见识自然也就会逐渐积累，作为主体的人，其能力也就可以稳步提高，最终的结果就会是：**人驾驭知识**。

而与之相反，当我们对知识的体系没有认知，而零散的学习各个点的时候，就很容易茫然。每个点都是有道理的，但是点与点之间却可能是有冲突的。这个时候，很可能人的思维会陷入混乱，并最终导致：**人被知识所驾驭**的局面。

这并不是只属于软件的问题，在中国，宗师治学大多从目录学始。

> 季羡林先生在《从学习笔记本看陈寅恪的治学范围和途径》一文中说：“中国清代的朴学大师们以及近代的西方学者，研究学问都从书目开始，也以此来教学生。寅恪先生也不例外。他非常重视书目，在他的笔记本中，我发现了大量的书目。比如笔记本八第二本中有中亚书目一百七十种，西藏书目二百种。此外，在好多笔记本

中都抄有书目。从二十年代的水平来看，这些书目可以说非常完全了。就是到今天，它仍然有参考价值。”

---摘自豆瓣

但很不幸，在软件开发这一领域中，暂时还没有目录学这样一种学问，所以软件开发这一领域中仍然保持着混沌状态，纷争不断。

为使这种模糊状态减轻一点，在进一步对设计和编码进行展开前，我们需要对设计编码相关的知识大致做一个分类。

本书中预设的前提有一条是：软件是一种固化的思维，而设计和编码是思维固化的过程，在这一过程中开发者需要持续**打造概念的边界和定义概念间的逻辑关系。**

这也就决定了所谓的设计和编码方法只有三个基本选项：

- 以逻辑关系为中心---这就是我们常说的结构化分析和设计
- 以概念为中心---这就是我们常说的面向对象分析和设计
- 两者兼顾 ---这是一直被忽略的一个视角，本书会在后文论及这种混合模式

如果把视角再拔高一点就会发现，世界再怎么丰富多彩，但你描述它的时候，基本要素也只是：名词（主语）和动词（谓语）。

我们分解世界中问题的时候，要么以名词为中心，要么以动词为中心，要么混合着来。几千年累积的历史可为这一观点的明证。

在软件的世界中，以逻辑为中心大致等价于以动词为中心，以概念为中心大致等价于以名词为中心。

如果上述分析没错，那么各种新技术名词就只能是对结构化分析和设计（以动词为中心）以及面向对象分析设计（以名词为中心）的补充，而绝不是并列的概念。

在2.2.9节中，我们曾经提到了计划(P)，实施(D)，检查(C)，分析(A)，改进(I)这五个基本步骤是组织行为中的太极剑法。而各种对结构化分析设计与OO的补充则大致分散在这个五个步骤之中。毕竟，设计和编码也是一种组织行为，也需要构建上述的回路，这种回路越敏捷，效率也就越高。虽然很多技术的发起者可能并没有意识到这一点，但他们的工作却确实是在使这一回路在设计和编码中运

转的更加顺畅。

我们来看两个实例：

- 在设计和编码过程中：设计可以大致等价于计划(P)，编码可以大致等价于实施(D)，测试可以大致等价于检查(C)，调试和修正可以大致等价于分析(A)和改进(I)。

  契约式编程强调的是规范和检查。在《Design by Contract：原则和实践》一书中作者在前言里写道：

  契约式设计的本意很简单，就是在设计和编码阶段向面向对象程序中加入断言(assertion)。而所谓断言，实际就是必须为真的假设，只有这些假设为真，程序才可能做到正确无误。

  而对于为什么需要加入断言，进行契约式编程，作者并未强调，但不难理解这可以让错误尽早的被发现，使设计编码的人可以尽快的收到反馈，进行下一步处理，也就是说贡献于检查（C）。

  因此，从归类的角度看，契约式编程是对面向对象程序进行实现时的一种支持手段。

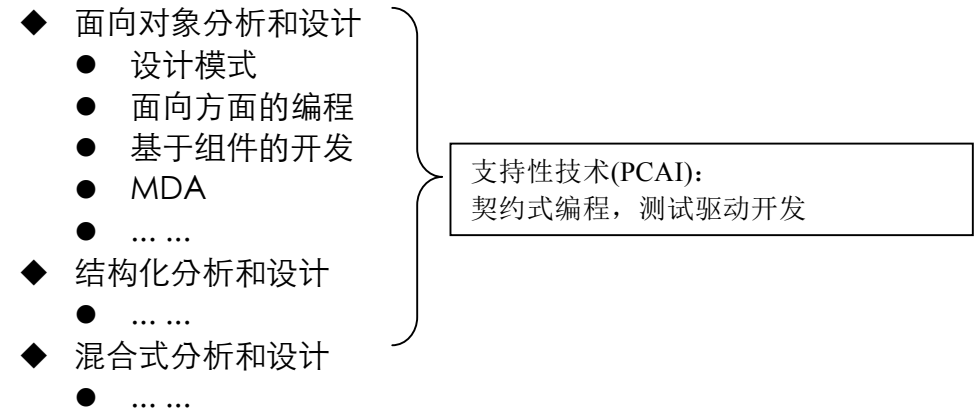
- 我们再来看一下面向方面的编程(AOP）。

  这一概念强调的是把业务逻辑和支撑业务逻辑的辅助功能（如：日志）分离开来。

  记录日志这一动作事实上对每一条业务逻辑都是必须的，这个时候可以选择把记录日志这一行为加到每一个与业务逻辑相关的方法中，也可以分离记录日志这一行为，加入一层代理，在代理中分别调用业务逻辑相关的方法和记录日志的方法。前者会导致记录日志这一行为分散在各个方法中，针对这一问题很多人提出了横切业务逻辑和日志的解决方法，这即是AOP的核心。

  从上述描述中我们可以看到，AOP更像是一种模式（范式），实质上是OO的一个补充，而非一个可以和OO并列的知识点，你不能讲AOP生成的对象就不是对象。

通过上述这样的分析，我们可以得到下面这样的归类：



而敏捷则是更大的概念，是利用了分析和设计技术的方法论。

编程的范式(paradigms)

编程的范式是另一个分类的重灾区。如果到Wiki上查，会发现类似下面这样的可怕列表：

… …

---图片取自Wiki

逐个分析每种范式是艰难的，但从分析和设计的角度看，我个人仍持前面所述的观点：要么以概念为中心，要么以逻辑为中心，要么混合。其他的东西是对这种大分类的补充。比如：很有名的泛型(Generic)以及基于此的元编程（Meta Programming)都是对上述三个大分类的补充。但这一观点确实没按照上述这样的列表逐一验证过，主要是这个列表太长，涉及的概念过于繁复。

§.2 设计和编码的存在意义

所有软件活动包括根本任务----打造构成抽象软件实体的复杂概念结构，次要任务----使用编程语言表达这些抽象实体，在空间和时间限制内将它们映射成机器语言。

---Brooks 《人月神话》

.2.1 价值根源

设计和编码的价值根源并不需要详细描述，因为它们是软件构建的根本。在软件开发中，可以没有管理、流程、估算、需求开发，但绝对不能没有设计和编码，否则就不会有软件的存在，一切都将无从谈起。

需要补充说明的是好的设计编码和坏的设计编码之间的价值差异。在软件开发中一个非常有意思的现象是一群人对着一堆有一定历史的代码，畏首畏尾，束手无策。

这就是垃圾代码的威力，牵涉到商业利益的垃圾代码就更有威力。就像城墙如果足够坚固，古代攻城时只能用人命来填一样。搞定牵涉商业利益的垃圾代码也只能用人月去填。

而事实上，这种惨烈的结局在同等投入下是可能避免的。

与生命不同，软件的生命周期同时取决于两个维度上的力量：

一是其商业价值，一是其内部衰败的程度。

在时间轴上，自然规律之下，动物身体的衰败无可避免；但软件内部的衰败却是人祸。

持续的良好的设计和编码，完全可以让软件内部的衰败得以避免。

如前文所提到的，软件是固化的思维这一特质，使思维的特质完全传导到软件之中。而一旦思维自身的规模被膨胀，清理凌乱思维的代价甚至会比重头创建还大。一种表现形式则是，正常情形下10个人月可以完成的项目，只准备了5个人月，结果由于赶工等因素，导致最终花了15个人月。

如果把视角扩宽一点，这种损失的表现形式就更加丰富，比如新产品时为了压低成本，牺牲了可维护性，但软件自身的生命周期却长达10年，这样开发新产品时的欠账，将在今后的10年里，数以倍计的返还。

减少这类成本，防止代码僵化并逐渐死去，是好的设计和编码的存在价值。这一点往往由于其不与短期商业利益直接发生关联而被忽略。

软件中的各种影响因素的权重

在现实中软件是商业的延续，因此大多时候在软件开发中商业因素比技术因素重要。

而商业也是人的商业，因此大多时候政治因素比商业因素重要。
所以最终现实是：【政治因素】 > 【商业因素】 > 【技术因素】。通俗一点讲就是：强权人物（表现为CEO或其他）的偏好可以压倒市场需要，市场需要可以压倒技术选择。

这很难看，所以很少有人这么说，但这更接近事实真相。

报复次序则正好反过来。

强人作出的决定如果顺应了商业规律，那么企业会顺风顺水。一旦强人的意志违背了商业规律，那么企业会倒闭。

商业考量自身如果顺应了技术发展趋势，那么会出商业神话。一旦商业考量背离了技术趋势，商业模式就会崩塌（生产力决定生产关系）。

技术因素就像一根不可能折断的弹性棒子，你可以弯曲它获得你想要的形状，但一旦超过一定限度，它就反弹回来，把一切都砸的稀烂。

李开复先生在《世界因你而不同》里面为此提供了一个经典的例子：

Vista项目的开发历史让我们可以清楚的看到【政治】因素是如何对一个项目产生影响。我们来看一下李开复先生讲述的这段历史的概要：

- Bill Gates 对 Vista 项目设定目标：

（1）支持新语言 C#，所有操作系统软件都改用 C#来写。因为 C#语言的运行较慢，但是开发速度很快，这样微软不会落后于多人参与的开源 Linux 操作系统的发展。

（2）开发 WinFS (Windows File System)，它是新一代档案系统，可以将每一个文件存成数据库。如果 WinFS 能够成功，慢慢的，全世界的数据就都存到微软的数据库，不但可以击败 Oracle、IBM 的数据库，也可以防止别的网络公司（例如 Google）掌控这些数据。

（3）开发 Avalon——新一代显示系统，让用户在浏览器里看到的网站或服务和传统的应用软件感觉一样。如果某网站的服务和用户端软件看起来一样，用户也更难理解网站服务的优点在什么地方。

- 了解到目标之后，很多总监对目标的难度心存疑虑。事实上这表征着，既定目标和项目自身所蕴含的合理性产生冲突。对这种疑虑，李开复先生的描述如下：

很多总监看到这个设想就倒吸了几口凉气：“技术难度太高了！C#这么慢，怎么能做操作系统啊？数据库不够快啊？怎么可能当做档案系统？”还有些研究芯片的专家常常看着 Intel 的芯片计划就开始担忧：“一定是微软习惯 Intel 芯片加速的速度，才这么乐观。但是每 18 个月芯片速度就快上一倍的日子已经过去了，别说 2004 年推出了的这些芯片，照这样，2007 年 Intel 的芯片都不够快啊。”

- 作为影响了一个时代的人，Bill Gates 具有无与伦比的影响力，对此李开复先生说：

  大部分的团队，就像我的团队一样，说服自己做了 leap of faith (信仰的飞跃)，相信在盖茨的督促之下，这三大目标都可以完成。

- 最终项目陷入困境。对此，李开复先生说：

  经过了三年的奋力拼搏，微软视窗团队的工程师们都已经疲惫不堪。但是，Windows Vista 的成功却似乎遥遥无期。其实灾难早就在酝酿，因为大家在一开始就知道，这个伟大的计划实现起来，其执行难度实在是太大了

WinFS团队虽然承诺了盖茨提出的三大目标，但是在实际的工作中感到了迷茫。WinFS团队认为他们的任务是“不可能的任务”，但是也不敢告诉盖茨。任何一个接触过Vista团队的人都知道，每次把测试版的Vista搭建出来以后，都发现庞大的系统根本无法运行。

这段历史之所以可以被看做是一个政治因素影响项目的例子，其根本原因不在于最初设定的项目目标过高，而在于认识到最初的目标不切实际竟然花了三年时间。

在这个例子中盖茨先生的影响力以及对商业因素的考量压倒了技术因素，但最终技术因素了报复了一切。

.2.2 定性分析

先回到我们一直在用的公式。

假设一个人的工程素养为E，一个人的工作意愿为W，组织所能提供的力量为O，

内耗系数为M，那么对于一个拥有n个人的团队，其在单位时间内最终可能贡献值可以表示为：

$$[(E_1*W_1 + O) + (E_2*W_2 + O) + \ldots\ldots + (E_nW_n + O)] * M$$

对设计和编码进行定性分析，所需要考察的维度和估算非常相似，都是内耗系数M和工作意愿W。

一是代码自身的结构越差，写同样代码行所需考虑的事情也就越多，这事实上增加内耗系数。最终导致的结果是从用户的角度来看，软件没有什么改观，但从开发的角度看，无数的人月却已经花进去了。简单来讲就是有投入没有产出，内耗极大。

根据5.1.2中的介绍，在COCOMO II中认为这种内耗可以使效能降低一半。

一是糟糕的代码为无所作为提供了冠冕堂皇的接口，团队的最终状态大多时候会变成和代码所处的状态一样。僵化、逻辑混乱的代码基本上会对应到僵化、迟缓、毫无斗志的团队。

我们前文曾经做过这样的假设：

如果我们认为积极向上的团队工作意愿为1，能对工作基本负责的团队工作意愿为0.5，负不起责任的团队工作意愿为0.5以下，极值为0。那么糟糕的代码很可能使工作效能降低一半。

所以说从长期来看，设计和编码的好坏至少可以导致4倍左右的效能差异。

这里需要特别提到的是对现有软件产品的选用，尤其是平台的选用。

当我们选择Windows或Linux，选择Java或.net，选择SQL或NoSQL等时，事实上也就选择了其背后所隐含的力量，这种力量最终将作用于组织力量O，决定基线的高度。假使说A技术代表的基线高度为20，B技术代表的基线高度为40，那么选择A的人或公司无疑的天生处于劣势。

本书并不以区分平台优劣为目标，因此并不会在各种现有技术的优劣上展开笔墨，但从架构设计的角度看，这些因素无疑的是不能忽略的。

COTS是什么？

在谈估算或开发模型类的书中，经常会提到COTS这个缩写。COTS是Commercial off-the-shelf的缩写。中文大多时候会被翻译成“商用现成产品"。按理说，软件开发的历史已经超过30年，商用现成产品应该很多，软件开发似乎应该沦为一种组装工作。但事实却远非如此，除了一些基础库外，在应用领域中大家还是在各干各的。这有法律，机制上的原因（你用非开源的产品，一旦出现问题可能完全解决不了），但更主要的原因则可能是需求变化太快，使更多的软件不像是螺丝，而是像是其他领域中完整的方案。

§.3 完美设计和编码的要素

> 简单的事情考虑得很复杂，可以发现新领域，把复杂的现象看得很简单，可以发现新规律。
>
> --牛顿

设计和编码是思维固化的过程，也是持续**打造概念的边界和定义概念间的逻辑关系**的过程。

在这一过程中，满足明确的需求之外，核心目标是保持简单性，而面向对象分析设计、结构化分析设计等，都是达成这一目标的手段。次要目标则是保证设计编码的高效，即不做无用功，不浪费时间。

之所以这么认为在于，软件所涉及的几乎所有其他质量特性，如：避免僵化性(Rigidity)、脆弱性(Fragility)、牢固性(Immobility)、增加可维护性等都是在使软件变的更复杂，如果没有一种反向的力量与之对冲，那么软件就一定只能变得越来越复杂，并且加速度过快。而可以与这种复杂化趋势相对立的力量也就必然是保持简单性---这是软件开发中的主要矛盾。

可以认为简单性大致等价于容易懂这样一种感受。而一件东西是否容易懂，则受制于人的一些基本特质：

- 人同一时间内，可以把握的概念和逻辑是有限的。如果理解一件事情要求人同时记住9个以上的状态，那无疑是不容易懂的。

  根据某些心理学家的研究，人一次能够掌握的信息块数大致为7，这个数字可以多至9也可少至5。

  ---Grady Booch，《面向对象分析与设计》

- 人对清晰的东西的理解要比模糊的东西的好。比如：如果你说这是圆的或者这是方的，都比较好理解，但你如果说这即是圆的也是方的，那就不太好理解了。
- 人对简单时序下的动作理解比较好。先做1，再做2，再做3,4这样的时序容易理解，但先做1，接下来做100，再做2，解下来做101，再做3,4这样的时序就困难了。
- 人是通过名字来认识事物的。(有名万物之始。)

总结来看，前三项强调的是概念和逻辑上不增加不必要的复杂度---完成既有要求的前提下尽可能简单，最后一项强调的则是名实相符。

针对如何确保简单性和高效率，我们分别提出以下的逻辑链：

设计编码的分解		逻辑链
确保简单性	不增加不必要的复杂度	软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，逻辑要尽可能的少→概念要尽可能正交
		软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，概念和逻辑的分散程度要尽可能的低 →概念和逻辑的层次要尽可能的少
		软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，时序自身要尽可能简单 → 如果多线程这样的手段使概念的实例处在很难理解的状态，逻辑关系将变得复杂→使时序复杂化时，要有关键理由
		软件是一种固化的思维 →思维的固化体现为概念和逻辑的固化 → 软件开发大多时候需要多人协作 →为保证简单性，但凡可封闭的信息要进行封闭 → 如果达不成这一目标，误用将增加不必要的逻辑关联
	名实相符	软件是一种固化的思维 → 理解已经固化的思维的唯一途径是阅读代码 → 第一个要接触的是名字 → 如果名字和他所要表达的概念和逻辑是吻合的，那么程序将变的简单 → 否则要加入一个从其他代码推导名字含义的过程，使代码变得不再简单

确保高效率	不浪费	项目所能耗费的资源是有限的 → 设计的东西如果用不上，是对资源的浪费，会减少其他环节的投入 → 如果设计看不到实效，那么必须终止

表 7-1：设计编码相关的逻辑链

实际设计时，一个疯狂的趋向是为可能发生也可能不发生的事情考虑的太多，这与我们这里强调的简单性原则冲突。我们这里强调的是，实现可见需求时，追求极致的简单性，而未来发生的问题，在发生之后再应对。

.3.1 逻辑链 1：正交的分解

软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，逻辑要尽可能的少 →概念要尽可能正交

1． 正交的基本定义

软件开发中最为核心的任务之一是分类，分类即是一种打造概念边界的过程。

打造概念边界的难点在于，概念本非有形之物，且概念之间又多有重叠、相关之处。这也就注定概念边界的澄清是一个渐进的过程，很多时候必须要进行迭代，进行反复，才能贴近最优的答案。

概念本身的边界究竟在那里在大多时候并不只有唯一答案，存在模糊性。比如：当我们描述一本书的时候，那么这本书是否被借出了这种信息，是既可以作为书的基本信息的一部分，也可以作为借阅人的基本信息的。

大多时候一旦切换视角，相同的概念又可以有多种划分方法。

好比说，我们可以很容易界定什么是人，什么是猿，但当我们试图把人猿归类的时候就依据我们的视角进行抉择。因为它似乎是人，似乎是猿，却又更是人猿。打造概念边界时正是类似人猿这类概念让我们犯难。

庄子对上述现象进行过非常形象的描述，他说：自其异者视之，肝胆楚越也；自其同者视之，万物皆一也 。

对上述问题，大多时候只有选择而没有答案，只不过选择背后所隐含的成本和收益往往不同。

打造概念边界时原则可以有许多，但主要手段只有一个，即抽象。从本质上讲，抽象是一种认清事物本质，并进行归类的过程。

抽象是设计工作的起点，而抽象的结果可以是一个具体的概念，也可以是一段逻辑。

正交性则是抽象时最为关键的一条原则，不正交的概念往往是含混的。

为了理解正交性，我们先来看一下这个词的几何解释：

图 7-1： 正交的几何含义

当两根直线互相垂直的时候，我们认为这两根直线是正交的，否则的话这两根直线就是不正交的。

这似乎和软件没什么关联。但如果我们假设相交的不是两根直线，而是两根圆柱的话，那么我们就可以看出来正交和非正交的差别所在。在正交的情况下，两根圆柱的最大接触面积始终会等于圆柱截面的面积，但在非正交的情形下，接触面积则要大于圆柱截面的面积，并且倾斜度越大，接触面积越大。

如果这两根圆柱是木材的话，那么接触面积越大，施工量越大，木材的可替换性也就越差。

概念或逻辑关系正交与否，其影响与上述类似。

假设说我们定义了两个类，类XMLReader负责具体读取XML文件中的节点，类XMLDataHandler负责加工从XML文件中读取出来的数据。这个时候如果在XMLDataHandler中出现了根据XPath读取XML内容的方法，那么这两个类无疑的会变成非正交的。

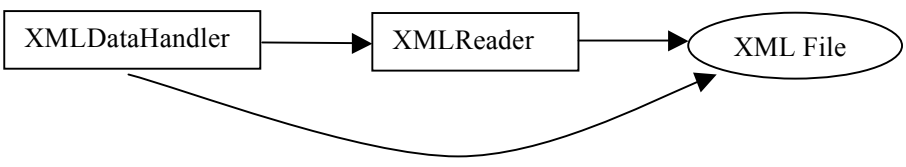


图 7-2： 程序中的非正交现象

因为读取这一功能既存在于XMLReader，也存在于XMLDataHandler。这种情形下，这两处地方都和XML的结构产生耦合，如果XML的结构发生变更，那么这两个地方都需要变更。同时也会导致程序中存在相似度比较高的代码，增加不必要的复杂度。

上述这类不正交的情况，有时候会被称为耦合，有时候会被称为不充分的抽象，但不管怎样，其根本问题在于概念或逻辑的非正交性。

不正交的情形有很多，但总结起来，这些情形大致可以分为两个类别，这两个类别与软件概念间可能的基本关系有关。

如果要把软件中的概念间的基本关系做个分类的话，那么大致可以分为两类：一为明确一种层次关系，不同的部分做的事情事实上是重叠的，但具体的程度不同，我们把这种关系称为**横向分割**，二为明确彼此关系，即你是什么，我是什么，我们把这种关系称为**纵向分割**。这种分割具有递归特质，概念或逻辑内部又可以进行新的一轮分割。

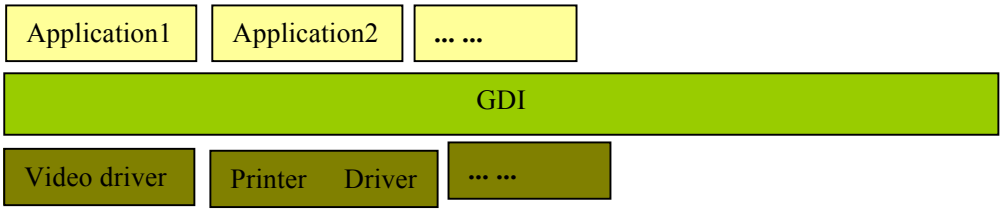
- 横向分割产生“层”的概念。

比较经典的例子有OSI的网络模型、Windows的GDI设计等。这里以Windows的GDI设计来做一些说明：

Windows一直强调一个所见即所得的概念（"WYSIWYG"），也就是说屏幕上用户看到的内容应该和打印机上打出来的内容保持一致。

如果应用程序（比如Word）与显示器的特性，乃至打印机的特性直接相关，那么几乎没可能达成这一目标。为解决这一问题，Windows中采用的办法是在具体设备和应用之间架起一个新的层次，这个新的层次即GDI(Graphics Device Interface)，其替代技术叫WPF（Windows Presentation Foundation）。

最终Windows中显示这部分的基本结构是：



注： 这里只画示意图，可能会给人一种错觉，感觉层次的切分是种容易的事情。如果有谁真的这么想，那就错的利害。写过显示器驱动和打印机驱动的人想必能了解定义层次间的接口是多么的困难。

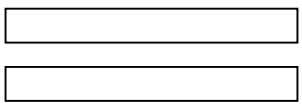
图7-3： Windows中的层次

这种情况下，GDI层和Driver层做的事情本质相同：即向指定页面描述指定内容。但具体描述方法不同，GDI较少关注设备特性（或者说只关注设备通用特性），而驱动程序则要关注设备的特有属性。

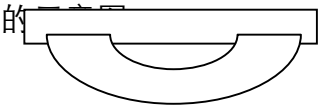
很多设计手法，其本质都是在软件的结构中加入更多的层次。像我们常说的Proxy，Facade模式等。

- 纵向分割则产生模块或对象，经典的例子是MVC等模式。Model，View和Controller其实是不同的概念，但他们彼此间有联系，所以这三个相对独立的概念要经过某种关系连接在一起。

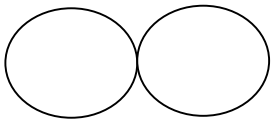
横向分割的时候，不正交体现为抽象层次上的不一致性，比如在Driver层面还做许多GDI层面应该做的事情；纵向分割的时候，不正交体现为重叠区域的存在，比如在View中直接对数据进行处理。请看下面的示意图



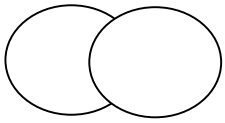
横向分割时的正交



横向分割时的不正交



纵向分割时的正交



纵向分割时的不正交

图 7-4： 正交于不正交的示意图

从上面的例子可以看出，正交性强调的是只让概念或逻辑在必须关联的点上产生关联。这里事实上包含了两个基本命题：一是不重叠，二是接触面要尽可能的小。

不重叠说的是两个概念间没有重复的部分，这很好理解。接触面最小则有点抽象，我们来结合一个例子进行说明。

2． 正交程度的优化

在《敏捷软件开发：原则，模式与实践》的第12章中，解释接口隔离原则(ISP)时提到了这样一个例子：

在安全系统中，有一些门，这些门可以被加载和解锁，并且知道自己是开着还是关着。

```
class Door
{
public:
    virtual void Lock() =0;
    virtual void Unlock() =0;
    virtual bool IsDoorOpen() =0;
};
```

接下来，一种更高级的门出现了，这种门如果开着的时间过长，就会发警报，这种门被称为TimedDoor。

因为要定时出发某些事件，所以TimedDoor要用到定时器，而定时器的基本创建机制是：

```
class TimerClient
{
public:
    virtual void TimeOut()=0;
};
class Timer
{
public:
    void Register(int timeout, TimerClient* client);
};
```

任何TimerClient都可以向Timer注册自己，而Timer则会按照指定的时间间隔来调用TimerClient的TimeOut()。

到现在为止，Timer、TimeClient、Door在概念上是正交的，没什么问题。

接下来，为了使TimedDoor具有定时发警报的功能，这三个概念要产生交互了。

Robert C. Martin在书中给出了三种可用的交互方式，我们这里简单引用其中的两个，并附加一个自己的解法，来演示正交程度的逐渐提高。

- 第一种方法Door从TimeClient继承，UML图如下：

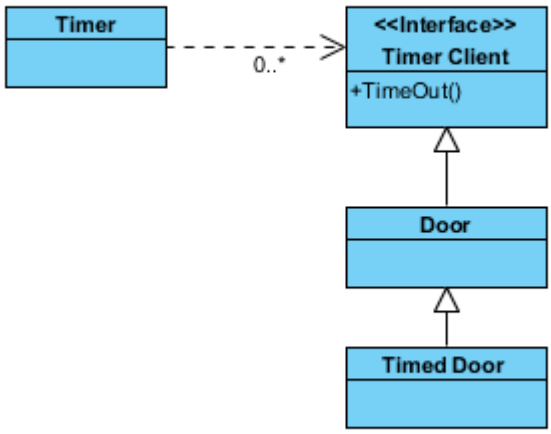


图 7-5： 定时门的 UML 图

这种方法的正交程度最不好，接触面过大，像Robert C.Martin在书里说的，很多Door根本和定时不定时没有关系，但一旦让Door从TimerClient继承，那就不管什么门都有了定时的特征，这种关联毫无道理。

- 第二种方法是让TimedDoor分别继承TimerClient和Door（多重继承），UML图如下：

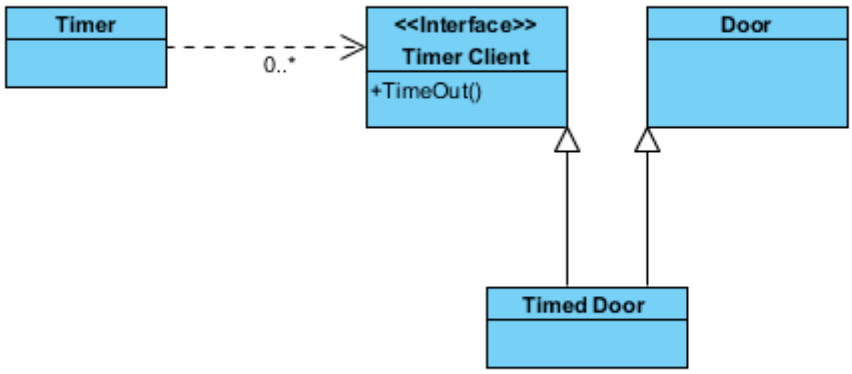


图 7-6： 定时门的 UML 图

Robert C.Martin认为自己会优选这个方式。

从正交的角度看，在这种方式下，TimedDoor只和与自己有关的部分产生关联，正交性已经非常好了。

- 如果仔细想想，就会发现第二种方法其实还是有问题。

门是否计时报警是一个功能，是否能录像也是一个功能，是否能自动发消息也是一个功能。

这些功能甚至可能是动态配置的，如果都用多重继承来解决，那会衍生出各种各样的对象，如：VideoDoor，TimeVideoDoor，MessageDoor，TimeVideo MessageDoor等等。这好像并不是什么好事。

这意味着正交的程度也许还是可以提升。

如果不让自己的思维局限于所有东西必须都是对象，那么就可以找到其他解法。

为了使比较更有焦点，我们先不考虑如何实现支持其他各种功能的门，仍只考虑如何为门添加定时功能。

现在我们认为定时报警功能是门的可组合部分，是使用关系而不是继承关系。也就是说，并不认为应该独立存在着TimerClient这样的类，而认为这是一种偶然

的组合。否则的话，XXClient这样的类会漫天飞。想象一下，门、马桶等都可以是TimerClient，也可以是Motor的Client。

修改后的Timer类像下面这个样子。

```
class Timer
{
public:
   void Register(int timeout, int timeOutID,void ( *start_address )( int timeOutID,void *arglist),void* arglist);
};
```

任何人使用Timer的时候，只需要提供一个触发间隔，一个回调函数，一个给回调函数用的参数。

当TimedDoor需要用到Timer时它需要干的事是实现一个回调函数，并调用Register()。

这时的TimedDoor像下面这个样子：

```
class TimedDoor:public Door
{
    static void DoorTimeOut(int timeOutID,void *arglist);
};
```

由于是静态函数，所以arglist需要用来传递当前实例的this指针。

这时因为去除了TimedDoor和TimeClient的继承关系，定时这一概念和门这一概念的正交性又有所提高。另一个收获是去除了TimeClient这样一个含混的概念。而之所以这么做的原因是TimedDoor和定时器的唯一交汇点就是“定时触发某个时间这一功能”，所有其它的关联不过是人为加上去的。

如果愿意再往下走一步，就会发现TimedDoor这个类事实上也不是必须的，任何一种门，如果它有定时功能，直接实现对应的函数即可，完全不必启用继承。在这种情形下，就只剩下两个概念：Door和Timer。

3． 在继承中确保正交

充分的抽象，其最终结果往往是正交的概念或逻辑，而正交的概念或逻辑大多时

候是应对变化、可测试、降低耦合度的基础。

这里面的一个关键点是如何看待继承，继承是OO的三大特征之一，但继承往往使正交程度减弱。

当前的主流观点是把继承等价于“IS　A”这样的关系。这是对的，但范围仍然太广，有点含混，只按照这一原则行事，容易导致类的杂多化。　在上述Door的例子里，这种杂多化体现为VideoDoor，TimeVideoDoor，MessageDoor，TimeVideoMessageDoor的出现等。这并不违反“IS　A”原则，所以可以用继承来实现他们，但无疑的违反正交原则或者我们将在下一节讲到的层次适度原则。

与继承相对的另一种视角是，不把Time、Video、Message视为种属上的必然差异而是视为一种偶然的可配置差异。这时候可以在Door中加入Classification这样的属性来标识不同的门。这种方法的正交性往往更好。

至于究竟在什么时候使用继承，什么时候使用属性来分类则和分类的根本原则有关。

我们可以根据肤色把人分为黄色人种，白色人种，黑色人种，但你不能根据人穿得衣服把人分为黄衣服的人，白衣服的人，黑衣服的人，后者一定会导致杂多的概念。分类的基础应该是尽可能本质的，恒常的特质，同时尽量避免基于偶然的，短期的特质进行分类。前者有必要使用继承，后者大多时候则可以体现为属性。

抽象与具体

在软件开发中抽象是一个经常会被用到的词，比如：有时候不好的程序会被指责为抽象不充分。

这里针对抽象和具体做一点说明。

从本质上讲，抽象是一种认清事物本质，并进行归类的过程。与抽象相对的是具体，但抽象的来源也是具体。

假使我们需要对【人】这一名词进行定义，那么必然是要从张三，李四等具体的人身上抽取共通特征，而后才能完成定义。最终结果是【人】这一概念来源于张三李四，但又不是张三李四。这样一个从具体的事物中抽取共性，再进行命名的过程就是抽象。

所以人是抽象的，真实的某个人是具体的。方法论是抽象的，按照方法论来运作项目是具体的。设计和软件是抽象的，软件的使用是具体的。

这听着有点绕，我们来看个具体的例子：

排序的时候，具体的算法和待排的东西是没关系的，待排的东西只要提供比较大小的函数就可以了。

这种情况下，如果我们把排序函数写成int sort(int*p, int len); 那么这里的抽象是不充分的，排序的方法和待排序的东西两者之间也是不正交的。为了进行充分的抽象，那么sort()要从具体的数据类型上解放出来：

```
void sort(
        void *base,
        size_t left,
        size_t right,
        size_t width,
        int (__cdecl *comp)(const void *, const void *)
) ;
```

随着抽象度的提高，适用范围确实提高了，但函数本身的可理解程度却降低了。这似乎是一对矛盾，抽象的东西灵活用途广泛，但可理解度会有所降低，具体的东西则利弊与此相反。

4． 小结

以正交性为分类的基本原则本身并没有太多值得争议之处，其关键在于认识到这是一个程度问题，而非可以完美解决的问题。在这一过程中很难追寻到彻底无瑕疵的答案，考虑到效能，正交自身则需要以合适为前提在恰当的时间点终止。

让我们以Grady Booch在《面向对象分析与设计》中的一段话来终结这一节：

> 那么分类为什么这么难呢？我们认为这里有两个重要原因。首先，尽管某些分类肯定比另一些分类好。但世上并不存在一个“完美”的分类。Coombs，Raffia和Thrall指出：将世界划分成对象体系的方法，其数量可能至少和完成这项任务的科学家的人数一样多。... ...其次，智能分类要求具有大量的创造性见解。Birtwistle，Dahl,Myhrhaug和Nygard提到：“有时答案是明显的，有时却只是一种感觉，而有时候恰当的构件是分析的关键点”。.....只有具有创造性思维的人才能找奥那些在别人眼中毫无联系的事物之间的相同点。
>
> ---Grady Booch，《面向对象分析与设计》

.3.2 逻辑链 2：层次的控制

软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，概念和逻辑的分散程度要尽可能的低 →概念和逻辑的层次要尽可能的少

1． 分层的利弊

对于软件而言，层次是让人又爱又恨的东西。很多问题是通过增加层次解决的，但另外一部分问题也是因为层次而导入的。我们来分别看几个例子。

例1：很多时候我们并不希望最终的应用绑定于某个指定平台，比如:Windows。为了达成这种跨平台的目的，就需要在OS和应用之间加入一个中间层，这个中间层负责屏蔽不同OS的差异。Java虚拟机等走的都是这样一条路线。

例2：当使用XML文件保存配置信息的时候，我们并不希望XML的结构在整个程序中随处可见。比如说：现在我们在Configuration/OutputFolder节点下保存了缺省保存目录，但将来很可能节点变成了Configuration/OutputFolder/Save。为了斩开与XML结构的关联，那么我们需要加入一个新的抽象层，来表征XML文件，再通过GetSaveFolder()这样的方法对缺省保存目录进行获取。

通过加入层次解决问题的同时，新的问题也随之发生。

在眼前蒙上一层薄纱可以防止眼睛被风沙所伤害，但如果蒙上十层，那更严重的后果将会出现---你看不到路了。

从可理解的角度看，只有某一功能所涉及的所有层次，所关联变量的各种可能性都被澄清之后，具体的代码才可能真的被理解。在排错的时候尤其如此。我们来看一个例子：

在用C++创建集合类的时候，我们可能希望对集合类的内存使用方法进行更多的定制。

有时候我们可能想预先保留一块内存，接下来在这块内存上进行二次分配来存放各种小的对象。有时候我们也可能想直接在磁盘上分配空间存放放入集合类的对象。

为了达成上面这些目的，层次又一次站出来发挥作用，我们可以通过allocator这样的类来建立一层抽象，创建集合类的时候，可以通过指定不同的allocator来控制内存使用的方法。

这应该是不错的设计方法，C++标准模板库里就是这么做的。

接下来我们来看一旦出了错的情形。

我们可能希望放入集合类的对象总是进行浅拷贝(swallow copy)，为此重载了类的拷贝构造函数和赋值函数，但最终发现当对象被放入集合类的时候，不知道为什么总是不成功。

这个时候，逻辑上程序没有任何问题，因此只靠脑子想是完全解决不了问题了。为了排错，我们只能启动调试器。

调试的过程中，我们通常并不能一下就确认问题和allocator究竟有没有关联，所以为了找出问题所在，我们也要对allocator这一层次做点分析。这种分析的开销事实上就成为添加allocator这一层次的代价---在这一场景下，所需要的只是分配内存，但却必须付出了解allocator机制的代价。

List容器的声明如下：

template < class Type, class Allocator=allocator<Type> > class list ;

通过添加allocator这个层次的方法解耦了“在那里分配”与“容器的实现”，但代价则是，一旦有问题，你要去挖穿各个层次，这有时候很困难。

通过上述的例子我们可以大致体会到层次这把双刃剑的威力。

通过层次我们可以让软件更灵活，抽象更充分，但层次也会把达成某一功能所必须的信息进行分割，增加复杂度。所以层次的多少往往并非是一个对与错的问题，而是一个程度问题，究竟什么样的层次才合适，是需要现场的人进行判断的。

2． 层次与信息分割相对冲的实例

为了更好的理解层次与抽象程度进行对冲这一事实，我们来看一个有名的例子，这个例子取自《重构：改善既有代码的设计》，代码则主要来自侯捷先生的网站，标为粗体的注释则是为了说明问题加上去的。

程序的目的是为影片出租店计算每一位顾客的消费金额并打印报表。

操作者会告诉程序：顾客租了那些影片，租期。程序则根据租赁时间和影片类型算出费用。影片分为三类：普通片，儿童片和新片。除了计算费用，程序还要计算每个人的积分，积分会根据影片种类而有所不同。

①最初版本。抽象不充分，但层次较少的代码。

表示电影和租赁的类都是数据类，因此略掉get/set相关的代码。

```
class Movie {
   public static final int CHILDRENS = 2;
   public static final int REGULAR = 0;
   public static final int NEW_RELEASE = 1;

   private String _title;              // 电影名
   private int _priceCode;             // 价格的代号

   ... ...
}

class Rental {                         //某一次租赁行为
   private Movie _movie;               // 租的电影
   private int _daysRented;            //    租的天数
   ... ...
}
```

下面是最主要的类 Customer:

```
class Customer {
   private String _name;                                    // 租电影的人的名字
   private Vector _rentals = new Vector();                  // 租借记录

   public Customer(String name) {
        _name = name;
   }

   public void addRental(Rental arg) {
        _rentals.addElement(arg);
   }

   ... ...

   public String statement() {     //这是原作者认为存在问题的地方，因此是重构的
重点
      double totalAmount = 0;                  // 总消费金额
      int frequentRenterPoints = 0;            // 积分
      Enumeration rentals = _rentals.elements();
      String result = "Rental Record for " + getName() + "\n";

      while(rentals.hasMoreElements()){
         double thisAmount = 0;
         Rental each = (Rental) rentals.nextElement();

         //determine amounts for each line
         switch(each.getMovie().getPriceCode()){// 按照价格的代码做分支，分别
计价
            case Movie.REGULAR:                   //普通片
               thisAmount += 2;
               if(each.getDaysRented()>2)
```

```
                thisAmount += (each.getDaysRented()-2)*1.5;
              break;

            case Movie.NEW_RELEASE:              //新片
              thisAmount += each.getDaysRented()*3;
              break;

            case Movie.CHILDRENS:                //儿童片
              thisAmount += 1.5;
              if(each.getDaysRented()>3)
                thisAmount += (each.getDaysRented()-3)*1.5;
              break;
          }

          // add frequent renter points          //按照既定规则，积分
          frequentRenterPoints ++;
          // add bonus for a two day new release rental
          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
              each.getDaysRented() > 1)
            frequentRenterPoints ++;

          // show figures for this rental
          result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
          totalAmount += thisAmount;
        }
        //... ... 打印输出部分略
      }
    }
```

原作者指出了这段程序的明显缺点：

- 报告部分和计算部分混杂在一起，一旦要添加新格式的报告，比如 html 格式的，那么需要编写全新的 htmlStatement()。这将导致同样的计费标准和影片分类规则在不同的代码段中存在，十分不利于面对变化。

事实上这就是不正交，可以独立的东西没有被独立。几乎所有人都会认同原作者的观点。但问题总有两面性，这么写程序也并不是一无是处：

- 层次很少，任何人都可以一眼看穿程序做了什么。新手也可以。

接下来我们看重构后的代码：

②重构后的代码。抽象较充分，但层次增加的代码。

```
public class Movie {     //主要变化是多了一个 Price 类
  public static final int CHILDRENS = 2;
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;

  private String _title;
  private Price _price;              //  用于处理计价和积分规则的新类
```

```
    public Movie(String title, int priceCode){
        _title = title;
        setPriceCode(priceCode);
    }
     // 转发调用，不在 Movie 类里直接包含价格代码，层次出现
     public int getPriceCode(){
        return _price.getPriceCode();     }

    public void setPriceCode(int arg) {     // 依据价格代号，创建不同对象，多态
        switch (arg) {
                case REGULAR:
                        _price = new RegularPrice();
                        break;
                case CHILDRENS:
                        _price = new ChildrensPrice();
                        break;
                case NEW_RELEASE:
                        _price = new NewReleasePrice();
                        break;
                default:
                    throw new IllegalArgumentException("Incorrect Price Code");
            }
    }
... ...
    double getCharge(int daysRented) {
        return _price.getCharge(daysRented); // 转发调用，层次出现
    }

    int getFrequentRenterPoints(int daysRented) {
        return _price.getFrequentRenterPoints(daysRented);// 转发调用，层次出
现
    }
}
```

重构后的 Rental 类：

```
class Rental {
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }

    ... ...

    double getCharge() {                    //新加部分,转发调用，层次出现
        return _movie.getCharge(_daysRented);
```

```
    }

    int getFrequentRenterPoints() {          //新加部分，转发调用，层次出现
        return _movie.getFrequentRenterPoints(_daysRented);
    }
}
```

重构后的 Customer 类：

```
class Customer {
    private String _name;
    private Vector _rentals = new Vector();
    ... ...
    //现在 statement 中只负责输出报告，其他功能通过调用方法完成
    public String statement() {          Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                        String.valueOf(each.getCharge()) + "\n";
        }

        //add footer lines
        result += "Amount owed is " +
                    String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
                    String.valueOf(getTotalFrequentRenterPoints()) +
                    " frequent renter points";
        return result;
    }

    //htmlStatement()  省略

    // query method
    private int getTotalFrequentRenterPoints(){
        int result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getFrequentRenterPoints();
        }
        return result;
    }

    // query method    计价
    private double getTotalCharge() {
        double result = 0;
```

```
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getCharge();
        }
        return result;
    }
}
```

新增加的 Price 类：

```
abstract class Price {    // 虚基类，多态
    abstract int getPriceCode();
    //真正计算价格的地方，要经过层次转发，才能到达这里
    abstract double getCharge(int daysRented);
    //真正计算积分的地方，要经过层次转接才能到达这里
    int getFrequentRenterPoints(int daysRented) {            return 1;
    }

}

class ChildrensPrice extends Price {    //子类
    int getPriceCode() {
        return Movie.CHILDRENS;
    }

    double getCharge(int daysRented) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }
}

class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }

    double getCharge(int daysRented) {
        return daysRented * 3;
    }

    int getFrequentRenterPoints(int daysRented) {
      return (daysRented > 1) ? 2: 1;
    }
}

class RegularPrice extends Price {
    int getPriceCode() {
```

```
            return Movie.REGULAR;
        }

        double getCharge(int daysRented) {
            double result = 2;
            if (daysRented > 2)
                result += (daysRented - 2) * 1.5;
            return result;
        }
    }
```

重构之后的程序的好处是：

- 报告和计算互不干扰，这使增加新的报告格式不会影响现有代码。
- 积分和计价规则独立在子类中，可以很容易的调整规则。

好处是非常明显的，几乎没有争议，如：正交性增加，灵活性增加等，但我们也失去了些东西。

- 层次增加了。getCharge()和getFrequentRenterPoints()被转发了2次。比如：Rental的getCharge() → Movie的getCharge() → Price的getCharge()再经过多态机制转到具体的对象。对于getCharge()而言，Movie这一层的主要作用是转发调用，并没有实际意义。
- 概念增加了。Price被衍生出来了。

对于小规模的程序，不论收益或者损失可能真的是无关紧要。Martin Fowler自身也强调对于这种规模的程序，重构是不值得的。所以希望读者把这一例子想象为大系统的一部分。

但事实上上面所提到的问题，在大系统中同样存在，收益和损失将同比放大。

这个例子提醒我们，设计中往往并非是追求单维度上的极值，而是要谋求一种平衡。7.3.1中所提到的正交是我们所要追求的，但这往往也带来层次。

平衡点究竟在那里是每一个设计人员必须关注的问题。就上述例子而言，我们无疑的可以发掘出层次数介于上述两者之间的方案，比如：单纯的分割计算和进行statement()的代码，把计算的结果作为statement()的参数。这样既不用生成新的类，又可以部分解决添加htmlStatement()所带来的困扰。

3． 一点总结

解耦这一工作本身达到一定水平后，其所带来的正效应会逐步降低，但其负效应却可能逐步增加。

单纯从理论上讲、可维护性、可扩展性、可重用性、松散耦合、性能、可移植性、精简性、时间开销等要求决定了一个最佳平衡点。超过这一平衡点后，任何一方面的增强都是以其他方面更大的损失为代价的。

从这个角度上，也可以讲，设计的过程是追求这个最佳平衡点的过程，而究竟平衡点在那里事实上需要一定度量手段进行支持，否则就必然因充斥过多的主观判断，而纷争无数。这一点我们会在第八章中进一步进行探讨。

曾经有人说过这样一句话，可供我们参考。他说：如果你知道自己在做什么三层足够；如果你不知道自己在做什么，那么十七层也没用。

如果我们认为9是一个人可以同时记住的最大的概念数，那么实现某一功能时层数无疑要小于这个值。

和层次相关的问题主要有两个：一个是层次的多少；另一个则是层次的一致性。如果说层次的多少是一个合适与否的问题，那么层次一致性则是一个是非问题。某一个层次上所体现出来的东西应该具有层次一致性。

这和前面在讨论的需求中的层次问题类似。比如说：如果有一个类叫Cat，那这个类的接口，可以有返回猫的颜色，猫的种类，这些属性是在一个抽象层面上的。但如果突然有一个接口是返回猫的个数，那么大多时候就会让人感到奇怪。

我们感觉到奇怪的本质原因是抽象的层次出现了不一致性。颜色和种类这种属性属于具体的某一只猫，而个数则属于猫的集合。

这种抽象层次的不一致实际上是增加耦合度的一个主要元凶。很不幸的是，这又是一个要依赖于个人技能的地方。眼下还看不到自动判断抽象层次是否合适的方法。

其实很多设计模式是通过增加层次来实现的。比如说：Factory，Proxy，Iterator,Command等。我们用《设计模式》一书中的Proxy来说明一下这种层次添加的过程。

假设说像Word这样的程序中有两个类：Application和Image。每当Word打开一个文件时，它逐个遍历Image的实例，装载所有的图像，并显示他们。这个时候Application直接调用Image的方法。但问题是这样一来，如果一个文档中有10000个图片，打开这个操作会很花时间。

所以需要用到Proxy模式。这个模式说的是，在Application和Image之间加入一个新的层次：ImageProxy。ImageProxy判断图像是否在显示区域内，如果在则调用Image的方法来显示图像，否则什么都不做。这样一来，装载速度的问题就解决了。

设计模式之所以有用往往在于其投入产出比较好，而不是因为其毫无代价--从保持简单性的角度看增加的层次即是代价。

为了驾驭设计模式，而不是为设计模式所驾驭，归根到底是要把握这里的设计原则，把握现场状况，把握具体得失。

.3.3 逻辑链 3：时序下的数据流

软件是一种固化的思维 → 思维的固化体现为概念和逻辑的固化 → 为保证简单性，时序自身要尽可能简单 → 如果多线程这样的手段使概念的实例处在很难理解的状态，逻辑关系将变得复杂→使时序复杂化时，要有关键理由

所谓时序就是动作的先后顺序。

时空的特征决定了这世界上大多的事情皆有因果，也决定了程序中凡事必有先后。当我们明确了先做什么，再做什么，最后做什么之后，我们才能对事情真正有所把握。不管采用什么设计方法，都要让这类路线尽可能明晰。惟其如此，程

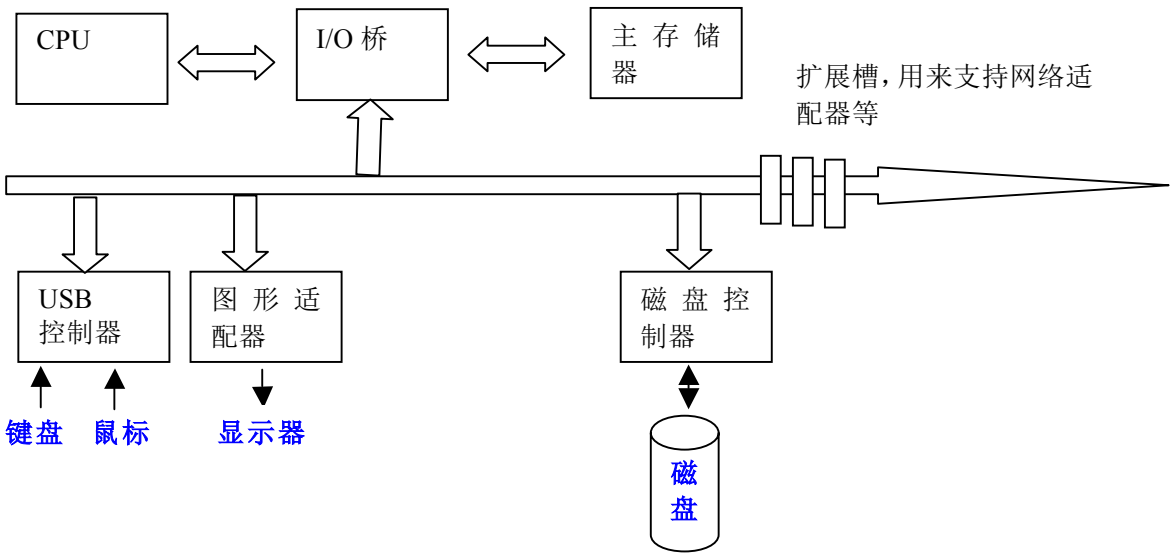
序才更容易懂，因为这是正常人最基本的认知世界的方式。

对时序影响很大的一个因素是并行，并行的一个常见实现方法是启用多线程（或多进程）。多线程类程序之所以难写，不在于本身的机制或同步多么难以掌握，而在于并发使整个程序中的逻辑变的更加复杂。

时序背后的东西是信息的流向。

关于这点可能很多人心存疑问，并会以为面向对象的世界里，这点已经无关紧要，但其实不是，明确信息流一如既往的重要。

正是在信息流动的过程中，概念才能完成衍化，归并，并最终再确定的过程。为理解这点，回顾一下计算机最基本的模型也许是有必要的。



改自《深入理解计算机系统》，在冯.诺依曼体系中，键盘和鼠标被称为输入设备，显示器被称为输出设备。

图 7-7： 计算机的基本体系结构

这个模型提醒我们一个最为基本的事实，不管你内部通过什么样的手段（OO或其他）做了多少细节处理，当人们使用计算机时，通常的过程是，通过输入一些东西（大多时候用键盘和鼠标），并希望获取一些输出（大多时候通过显示器），而输入、输出必不相同。

人们使用计算机的目的，也正是软件的根本使命。

软件对用户输入进行抽象，分割，而后按照指定的步骤进行相应处理，最终，把结果返回给用户。而处理步骤所体现的正是信息的流向。不管我们使用什么样的设计方法（面向对象或其他)，信息流向越复杂，信息的分散度越高，软件也就越难懂，软件的维护也就会越发的困难。

我们常说的顺序，分支，选择，优先顺位处理则都是时序的一种具体表现形式。=优先顺位处理关系是指抢占，一个典型的应用是操作系统中的线程调度。

大多情况下，并行对复杂度影响过大，并间接导致测试困难---多线程或多进程导致的问题往往是有时发生，有时不发生，一般的测试手段并不足以发现这类问题。

所以原则上应该尽可能不用，除非收益足够大。或则说在满足需求的前提下，线程数和进程数应该尽可能少。

以多线程和多进程而论，确定“什么时候适合使用这种技术”是比“怎么使用这项技术”困难的多的事情。

现实中人们往往对事件，信号灯等同步处理手段关注过多，而对究竟应不应该启用多线程/进程关注的太少。这里来简单做个总结，对于下面这些场景，一般来讲启用多线程/进程是合适的：

- 数据很容易分割，处理不同数据时彼此间没有什么交互。比如说：你有10万个文件需要处理，每个文件的处理彼此独立。这时候显然要做并行，这样最终数据处理速度将依赖于并行分布的程度。这好像很简单，但却触发了MapReduce这样著名的模型(参见《软件随想录》)。在待处理数据持续增长的情形下，如果不启用并行，那几乎没有办法保证速度不会下降，这并不是单纯靠优化算法可以解决的问题。
-
- 监控设备响应（端口监视也应该属于这个类别）。这时候我们会希望能及时响应设备的事件，而不希望设备上的数据采集和对数据的处理彼此干涉。比如：Windows中就单独建立了Raw Input Thread(RIT)来监控整个系统的键盘和鼠标消息，而后再分发给各个进程中的线程。(详见《Windows核心编程》)。
-
- 优化UI响应速度。比如：用文档编辑器打开大文件时，如果不分离UI响应和实际处理，UI会挂起，很典型的应用是进度条的处理。使用了MVC(Model-View-Controller)模式的应用中大多时候可以适用这条，因为很多时候都需要View和Controller总是保持响应。
-

- 待处理的请求天生就是并行的。比如：当浏览不同网站时，不同的网站的显示会在不同的窗口中处理，这类处理天生就是并行的，既可以用多线程也可以用多进程。客户-服务器体系也是这类情形。
-
- 计算量很大，并且算法可以分割。有些传说中的领域（大气模拟、基因工程等）事实上是并行计算的主战场，比如在展示MPI(Message Passing Interface)的使用时，经常使用的计算π的例子，就是如此。但这离通用软件开发的距离就有点远，知之不详，这里就不列了。
-

这个列表应该进一步加长，但限于精力，眼下却只能列这么多了。

对于多线程的误用，在MSDN上可以找到一份很好的总结，这里提取几个典型的：

- 锁争用和顺序执行。这时虽然启用了多线程，但实际执行起来这些线程是串行的。
- 过度订阅。如果系统的核数远少于线程数，同时启动很多个线程，时间往往会浪费在抢占上。
- I/O效率低。多线程被用来做频繁的I/O操作，这会导致很多时间开销在I/O操作上。

并行的种类

为并行分类并不是很容易的事，这里只介绍一种最简单的，依据内存架构对并行进行分类的方法。依据内存架构，并行可以分为三类：

- Shared Memory: 常说的多线程即是这一类，比较有名的实现是OpenMP。
- Distributed Memory：如果多台PC组合起来进行并行计算那就是这一类。比较有名的实现是MPI。
- Hybrid Distributed-Shared Memory：如果多台PC进行分布，每台PC上还启用多核，那就是这类。

有人似乎还想进一步区分并行和并发，这就有点微妙了，这里不做这类区分，请读者自行研究吧。

.3.4 逻辑链 4：信息的隐藏

软件是一种固化的思维 →思维的固化体现为概念和逻辑的固化 → 软件开发大多时候需要多人协作 →为保证简单性，但凡可封闭的信息要进行封闭 → 如

果达不成这一目标，误用将增加不必要的逻辑关联

信息隐藏的价值根源

假使说我们认同软件的构造是一个复杂的过程，那么管理这种复杂度必然需要一些技巧。而为了找出这些技巧，则需要先考察一下这种复杂度的基本构成。

软件的构造过程牵涉了两个最为基本的要素：一是软件自身，一是构造软件的人。

假设说存在着一个标准的人，这个人智力水平恒定，创新能力恒定，技能水平恒定。那么软件的复杂度只决定于其自身，比如软件所需要面对的业务规则、所需要的计算水平等。应对这类复杂度的有效手段是优化方法，好比说快速排序的效率大多时候就是会比冒泡排序好。

当我们开始考虑人的可变因素时，复杂度的来源则发生了变化。人是有着许多与生俱来的特质的，比如说：人是会犯错的、人同一时间可以处理的事情是有限度的等。

因为这些特性，人在应对软件自身所拥有的复杂度的同时，又带来了偶然的，与个人特质相关的复杂度。比如说：人无法同时记住过多的概念，否则容易记忆出错；局部变量的名称可能碰巧和一个全局变量相同，而我们又误把这个局部变量作为全局变量使用了。

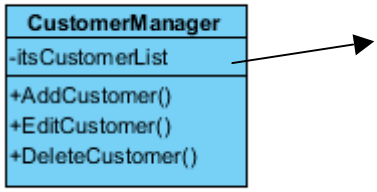
信息隐藏则是减少这类偶然性复杂度的一个有效手段。

对于方法或函数而言，输入参数，输出参数描述了它的边界。对于类而言，公有属性和公有方法描述了它的边界。

边界以内的即是被隐藏掉的信息。这个时候不管内部多么复杂，对外而言只是一个单一概念，这有助于使程序变的更加容易理解。也即是说信息隐藏可以减少在特定层面上需要暴露的信息块数

假设说我们用一个名为CustomerManager的类来管理客户的基本信息，但在开放出AddCustomer(), EditCustomer(),DeleteCustomer()这样接口的同时，也开放了内部存放客户信息的集合类。那么使用CustomerManager类的人，很可能直接通过操作集合类来获取客户信息。这个做法虽然可以被编译器所接受，但实际上一旦如此，任何一个人关注CustomerManager时就需要关注4

个信息块，而非是3个。为了避免这类误用，存放客户信息的集合类应该被隐藏起来，让用的人只看到接口。



一旦暴露了内部数据结构，那么必将有人直接使用，最终导致使用这一类别时需要把握的信息块增加。

图 7-8： 信息隐藏不好的例子

1． 信息隐藏的归纳

下面我们来对信息隐藏做一些归纳和总结：

- 不同手段下，信息隐藏的程度是强弱有别的。

如果外部只能看到接口和基本数据类型，无疑的，信息隐藏的最彻底。这个时候，如果必要通常可以在物理上把相关部分分离出来，如创建动态链接文件等。

使用类的各种关键字（如：private，public)来控制信息隐藏时，信息隐藏的程度已经弱了一级。因为这个时候类自身所处的状态已经成为一个必须了解的东西，而为了解这些东西，很多时候就必须了解内部的数据和私有函数。这不难理解，公有方法可以操作私有数据，那么理解公有方法就需要了解私有数据。

常说的全局变量信息隐藏程度最差，这几乎是不做信息隐藏。

- 信息隐藏是抽象数据类型(ADT)、类等之所以存在的一个比较主要的理由。但信息隐藏并不只在使用抽象数据类型和类的时候才有意义。定义接口的时候，甚至实现的时候信息隐藏这一原则同样可以发挥作用。
- 比如说：实现方法A的时候，实际上只需要类的两个属性，但却把整个类定义成了这个方法的参数，这也是违背信息隐藏原则的。

- 目的是信息隐藏，但实际上却违背了信息隐藏原则的情形。

比如说：类的方法中常常会使用到成员变量。有时候和类自身关联不是很紧密的属性也会被添加为成员变量。表征数据库的类（比如DataBase）大多时候并不需要和获取数据的类（比如传感器的类Sensor）直接关联，但有时候Sensor的实例却会被添加为DataBase的私有成员变量。

- 对外是信息隐藏，但对内却完全不隐藏的情形。大多时候类中的方法并不会把成员变量作为自身的参数，而是会直接使用，比如说：由于Sensor的实例是DataBase的私有成员变量，那么DataBase::GatherData()在用到Sensor实例的时候会直接使用，而不会把接口定义为.DataBase::GatherData(Sensor& sensor)。这种用法与成员变量过度膨胀的趋势撞在一起后，有意思的事情发生了。

从外部看，传感器这样的成员变量是私有的，达到了信息隐藏的目的。但对于方法而言，结果却完全相反。

每多使用一个成员变量，其自身的信息隐藏就被破坏一点 极端情形下，如果整个程序只有一个类，那事实上等于没有信息隐藏。

如果在同一方法中，用到五六个成员变量，那么这个方法会变得非常难懂。因为对于类中的方法而言，这些成员变量自身的变化是完全不被隐藏的，每个都有自己独立的脉络，比如说：DataBase::SetInterval()方法中可能会调整Sensor实例的状态，进而对Sensor::GatherData()产生影响。这会导致方法的边界不清晰。不清晰，也就不可能紧凑。不紧凑意味着阅读的时候，一个方法所需要的信息不能立刻获取。最终会导致程序难读，难改，难维护。

这类实例提醒我们 ,并非把很多操作和数据成员都放到类里面就达成了信息隐藏的目的。事实上这也对每个类的可能规模提出了限制。极端的情况下，如果一个类很大，有2000SLOC，那么所有这2000SLOC代码就等于通过成员变量粘结在一起，这和过度使用全局变量在一定程度上等价。

从信息隐藏的程度上来看，对方法而言，局部变量最佳，类的成员或属性稍差，全局变量最差。局部变量把信息隐藏在方法之内，成员或属性把信息隐藏在类之内，而全局变量等于不做信息隐藏。

2． 一点总结

信息隐藏和前面提到的层次问题并不能分割开来。从基本趋势来看，信息隐藏做的越彻底，层次也必然就会越多，而前面曾经提及过层级自身是有副作用的。

从最小化层次的角度来看，使用函数或方法来隐藏数据与使用类似抽象数据类型(ADT)的方法来把数据封装到类之中，代价非常微小，几乎总是对的。但在类A中包含B，类B中包含C，类C中包含D这类的方式则福祸难料。诚然B,C,D被隐藏了，但绝不是毫无代价。

抽象数据类型(ADT：Abstract Data Type)

谈及封装的话，很多人立刻就会想到面向对象，但事实上封装的历史要比面向对象还要长一些。从抽象数据类型（ADT）开始，人们已经开始使用封装这一思想。

抽象数据类型(ADT)更像是类的前身，本身并不是复杂的概念。当你有一组数据以及与其捆绑在一起的一组操作，你就有了一个抽象数据类型(ADT)。

《代码大全》中列了一些典型的抽象数据类型，如：

- 油罐
- 填充油罐
- 排空油罐
- 获取油罐容积
- 获取油罐状态
- 列表
- 初始化列表
- 向列表中插入条目
- 从列表中删除条目
- 读取列表中的下一个条目

.3.5 逻辑链 5：“名”与“实”的契合

软件是一种固化的思维 → 理解已经固化的思维的唯一途径是阅读代码 → 第一个要接触的是名字 → 如果名字和他所要表达的概念和逻辑是吻合的，那么程序将变的简单 → 否则要加入一个从其他代码推导名字含义的过程，使代码变得

不再简单

老子在《道德经》里说：无名天地之始；有名万物之母。 程序也是这样，当我们看到一份设计图或一份代码时，大多数人会【望文生义】。但使人【望文生义】却正是语言文字的根本使命，并不是什么不合理的事情。因此，如果一个函数被命名为Add()，但内部实际做的是减法，那么这份设计或者这份代码，一定是很难理解的。

于是一个非常现实的问题就摆在了我们的面前：我们究竟应该如何为类，为方法等等命名？

在命名时，有两类较大的错误：一个是名实不符，一个是词义混淆。

名实不符的常见情形又有两类。比如：

- 以偏概全。假设说一个方法被命名为OutputLineNumber()，但实际上这个方法会在分析源文件后，同时输出LineNumber和指定行号下的内容
- 大而无当。假使说一个类被命名为XMLHandler，那么看得人基本不能从这个名字上获得有效信息。这主要是由于Handler这个词的含义过于宽泛。

上文所说的命名为Add()但实际做减法操作是名实不符的极端情形，达到了名实相反的地步。

词义混淆则源自语言文字自身的限制。常表现为一词多义或多词一义。语言文字可以被看成是一种表意的符号系统，这个符号系统的典型特征是其模糊性。同一个意思可以有许多种表达方法。比如说：index和number是两个不同的词，但很多时候其意义互相重叠。而假如说一个变量名为fileNumber，那么这个变量既可以代表文件的总数，也可以代表某个具体文件的编号。

上述问题会因为英语不是我们的母语而变得更为麻烦。

名实不符与词义混淆这类陷阱的一个主要触发条件是软件自身的不停衍化。

前文曾经提到过，对于概念和逻辑的认知是一个逐层递进的过程。在这一过程中，包、类、方法等的内涵必然会发生变更。变更无疑的会使名实不符这类问题加剧。比如：一个类原本负责输出测试结果，这时候OutputTestResult这样的命名可能是合适的。可随着软件功能的增强，最终可能不只要输出结果，还要对结果进行

一定分析和统计。这样原来的命名就显的有些不合适了。

在对命名这一问题的根源进行分析之后，我们来看看可能的应对方法。

命名问题事实上并不能只在命名这一环节进行解决，首先要有容易命名的对象，接下来才有容易命名的事实。比如说：是先有猫和狗这类外在特征不同的动物，接下来我们才用猫和狗这样不同的名字来称呼他们，而非相反。

如果概念之间是正交的，概念的边界也是清晰的，那么命名无疑会容易很多。好的设计是改善命名的基础。也就是说，很多时候我们感到命名困难，真正原因可能并非是命名的技能不够，而是设计还不够优化。

在努力改善设计之后，才需要面对纯粹的命名问题。

从本质上来看，命名问题并不是一个编程的问题，而是一个表达的问题。命名最终对读程序的人负责。

有些表达上的基本原则对于解决命名问题会有些帮助，比如：

- 尊重既成事实

无疑的每个人都是有创造性的，但在命名的时候发挥创造性则更可能是有害的。比如说：在XML的世界中一般使用sibling这个词来表示兄弟节点，这种时候就不需要创造性的使用brother node这样的自制词汇了。

- 用完整词汇

对一般人而言要求事先记忆几十个缩写词，而后来读程序是不太可能的事情。所以如果一个程序中充满_Tx和CSCP这样的特制符号的话，那这样的程序几乎一定是不容易懂的。

一个典型的反例是P.J Plauger版的C++标准模板库。也许是出于隐藏实现细节的目的，这份标准模板库的实现里面几乎完全不用完整词汇。如果想体验一下不用完整词汇的后果，那么可以读一下这里面的代码。

- 要具体

具体是一个方向。比如说：可以用OutputLinenumber()的时候，就不要用Output()。而可以用OutputCppLinenumber()的时候就不要用OutputLinenumber()。

- 多人协作的时候，使用统一规范

这条规则最终会导致建立常说的Coding Rule。对具体如何定义Coding Rule，《代码大全》一书中给出了详细的指导，这里就不在说明了。

有一点需要补充的是，就和说话的时候记不住语法一样，设计或做编码的人往往也记不住编码规则。所以规则也不宜过多。如果必须详细，那么至少要主次分明。比如说 :统一使用主宾结构还是使用动宾结构这样的选择会影响程序的大部分内容，那么就应该优先统一。

现代的IDE中大多提供了对重命名的支持，因此，一旦对如何命名有所把握，修改已有的不恰当的命名反倒不是困难的事情。

Code Review的价值

Code Review在很多项目中是“鸡肋”环节---扔了可惜，做了难受。但事实上Code Review也是最容易被误用的环节，就好像锤子可以用来打钉，但不能用来拧螺丝一样，使用Code Review前也要先考察，它所适用的场景究竟什么。

这里做一点简单的总结，以供参考，Code Review对达成下面这些目标帮助较大：

检查代码非功能性质量（如：命名是否符合规范，分解是否正交等）

检查并行相关的错误。（如：多线程下，有的问题是有时候有，有时候没有的，这时候如果没有在Code Review中做一定检查，很可能在黑盒测试中无法发现。）

提高团队成员的编程能力。

有些目标本身就不适合用Code Review来达成：

检查软件的功能是否正确。

检查需求是否都已经被实现。

.3.6 逻辑链 6：设计的终结

项目所能耗费的资源是有限的 → 设计的东西如果用不上，是对资源的浪费，会减少其他环节的投入 → 如果设计看不到实效，那么必须终止

在探讨什么时候可以结束设计时，需要假设输入的需求是正确的。在此前提下，用不上的设计大致可以分为两类：

一是程度问题。设计如果关注某些细节，而这些细节自身却随着认知的深入而发生变化，那么与此相关的设计会变成无用的。

一是无意间扩散了需求的边界。比如：在看不到可移植性需求的情形下，为可移植性做了准备。这类设计的无用并不体现在技术上，而是体现在成本上。

对于程度问题，直接探讨终结尺度在那里，很难找到令人信服的答案，因此我们可以像在6.2.3需求开发的终结一节中所做的一样，把这种程度问题转化为经济问题，这样更便于我们思考。

假使说设计上考虑了一个因素，其所需要的开销是X，而不考虑这一要素，在后期发现并进行对应，其对应成本是Y，那么凡是会导致X<Y的东西就都是需要在设计中预先澄清，而不能等待在编码中逐步澄清的。

从这个角度看，我们就可以列出来设计大致的终结尺度：

- 影响全局的由需求所定义的功能性要求和非功能性要求是否都被考虑到了。如：性能目标是否能达成，国际化的需求是否被考虑到了，出错处理是否有统一的机制进行支持，是否需要复用既有代码，是否某部分要留做将来复用。对此，《代码大全》中提供了一份非常详细的Checklist，此外很多关于架构设计的书中也主要以此为着眼点进行展开，这里就不再进一步展开了。

- 购买（或使用开源代码）还是自主开发这一维度是否被考虑到了。软件这一行业存在的历史越长，重复发明轮子的可能性越大。因此，是否购买（或者使用开源）而不是自己开发会成为一个越来越重要的问题。评价是否购买（或者使用开源）则可以进一步分解为：功能上能不能满足目标，经济上划算不划算，能否及时获得支持等。与这种考量关联最紧的则是估算，没有比较精确的估算就没法判断是赔还是赚。
-
- 软件的可测性是否能够达成。软件是否可测本身更大程度上是由设计决

定的，而不是由测试所决定的。

-
- 既有的设计是否支持对当前人员进行分工。这个维度事实上经常被忽略，但其本身则是一个比较明晰的，判断设计是否可以终止的尺度。假设说你的团队有10个人，那么只和1个人工作有关的部分可以不在设计时进行考虑，而与2人或2人以上相关的部分则要在设计时定义清楚接口。
-
- 需求碎片化的应对。如果说是不停地往已有的代码中增加新特征，而每一个新特征都非常小，那这时候，设计会变的无限薄直至取消。

在对无意间扩散了需求边界这一问题进行具体分析之前，需要先看一对基本矛盾：软件设计自身最优化与项目本身资源有限、时间有限间的矛盾。

有些书籍会堂而皇之的谈到：设计时需要考虑并发问题，需要考虑可移植性问题。这无疑是对的，但把问题过于单纯化了，事实上设计最优化并不是总有意义。

假设说我们承认，开源之外，软件是商业的延续，那我们可以认为商业世界定义了软件的价值根源，而需求本身对这种价值根源的边界进行定义，设计本身并不能超出这种价值边界，否则就是种浪费。

这似乎有点抽象，我们还是用上面的例子来辅助说明。

假设说我们要开发一款简单的分布式计算框架（服务器端负责管理任务，把任务分发到客户端，而客户端负责执行具体任务并把结果汇报给服务器端），这个时候是否使这个框架跨平台将对设计产生很大影响。

从软件自身最优化的角度看，支持跨平台是毋庸置疑的问题。但如果在需求之中看不到跨平台的要求，甚至说作为需求分析的结果一旦选定平台后，未来5年内，几乎完全看不到跨平台的需要，那么在设计层面究竟应该如何处理这一“技术问题”？

如果项目人力资源有限，时间有限，支持了这个，很可能就不能支持那个？或者就要增加开销？这真的是设计层面应该考虑的问题么？

在这一点上，我个人倾向于认为这是需求问题。设计人员需要假设需求开发人员所做的需求开发是完善的，有疑问可以确认，但不能在设计层面随意扩张需求的边界。

毕竟在大多时候，如果资源限定，完美是良好的大敌。

钱的贴现值

本书抽去了商业这一维度，因此这一章里有的地方变得有点含糊。但事实上如果把这里提出问题转换为经济问题，就更容易理解一点。我们可以把支不支持跨平台这类问题同投资相类比。

如果我现在需要投入1000块钱，但可能在10年后才产生收益，那么如果使用5%的贴现率，只要10年后的投入不超过1628，就还是10年后再投入划算。设计也一样，如果晚做，没什么太大损失，那就没必要预先做了，这会浪费现时点的有限资源。

在经济学家眼里，现在的1元钱和1年后的1元钱是不等价的。如果贴现率是5%，那么现在的1元钱等于一年后的1.05元。这和支不支持跨平台，支不支持并行这类问题很类似，与其说这是技术问题，倒不如说这是个经济问题。

§.4 完美设计和编码

伟人之所以看起来伟大，是因为我们自己在跪着。站起来吧！

--列宁

.4.1 完美设计和编码的形象

当一辆车从苏州跑向北京时，为了缩短时间它必须不停的寻找最佳的路径。而为了寻找最佳路径，一是要知道需求，也就是说要知道究竟都要去那里，这样才能设计最佳的路线；二是要不停的确认路况，那条路通畅，那里封路了，这样才能具体选择究竟走那条路。

这一过程很类似于软件中的设计和编码，设计和编码基本上是在大致知道了要去

那里的前提下，寻找具体路径的过程。在这一过程中，通常并没法只按单一原则行事，路况好的可能路远，路近的则可能路况不好。这意味着追求完美设计即是追求多原则下最佳平衡点的过程。

总结来看，在寻找最佳解决问题方法时，完美的设计与编码有六个关键原则需要遵守：

- 正交的概念和逻辑
- 层次的最小化
- 时序清晰
- 隐藏不必要公开的信息
- 名实相符

这五条规则与常见的面向对象设计原则不同，这起源于我们只指向一个目的：简单化。我们认为软件设计或编码的根本评价标准是在**满足现有可见需求的基础上，使代码尽可能简单**。而之所以选择简单性这一维度作为设计和编码的核心考量，最关键的原因是所有其他质量属性（灵活性等）都潜在的使软件有复杂化的趋势。当我们面临有矛盾的诸多选择时（如：易用性和可测性），我们就必须定下君臣佐使，否则就会陷入到这段代码体现易用性，那段代码体现可测性的无秩序状态。

在此之后，我们可以从另一个侧面对完美的设计编码做点总结：

假使说既定要求最终复杂度可以为100~120，那么完美的软件开发则要求达成以下两点：

一是设计和编码要确保总复杂度不能超过100。

一是100的复杂度可以平均分摊到10个模块，那么每个模块的复杂度为10。这样的话，如果复杂度为10的理解难度不大，那整个程序是可以理解的，复杂度不高。但如果其中5个模块复杂度为15，另5个模块复杂度为5，那么虽然总值不变，呈现给后来程序员的复杂度却是不同。

简单来讲，达成要求以外，简单性压倒一切，接下来才是设计原则。如果遵守设计原则，代码变得更简单，那么遵守设计原则。如果相反，那么选择简单性原则。

在后续的案例1中，我们会探讨如何牺牲一点原则来获取简单性。

同时也需要认识到，因为需求而牺牲简单性是必然的，比如说：性能的要求可能要求把简单的代码变得复杂，这并不是例外，关键是当这么做的时候，要清楚的知道付出的代价。

在案例2中，我们会探讨一个因需求而牺牲简单性的例子。

最后一个需要阐明的问题是，如果切分设计和编码，那么设计应该停在什么样的程度上？这是个经济问题，如果设计需要的投入是X，而可以获得的收益是Y，那么X<Y时设计就应该做下去，直到X差不多等于Y。

直接计算X和Y有点困难，但并不是没有取巧的办法。

设计自身蕴含着一种分工方法，假设一个团队有10个人，其中1个人为架构设计师。

架构设计师最终的输出必然要支持对10人的一种分工方案。

- 从分工的角度看，凡是涉及两人以上的东西（接口，数据等）X<Y的可能性都极大。因为这部分工作一旦做了，并且是正确的，那节约的时间往往是要以倍计的。所以设计的终止尺度可以是每个人和别人间的关联得到澄清。
- 从支撑需求的角度看，那么则要检查是否可以支撑体现软件核心价值的需求。尤其是非功能性的需求。这类需求甚至有导致既有实现推倒重来的可能性，风险太高，一定要事先考虑。比如：是否跨平台等。
- 从人员状况来看，能力差的人越多，设计的程度应该越细，反之则可以使自主程度高一点。因为能力差的人自身所做的决策，错的可能性较高，即Y偏大。

面向对象设计原则和上述关键点的关系

在《敏捷软件开发中：原则，模式和实践》一书中，作者一共提及了11条面向对象的设计原则。他们分别是：

- SRP 单一职责原则：就一个类而言，应该仅有一个引起它变化的原因
- OCP 开放-封闭原则：软件实体(类，模块，函数等）应该是可以扩展的，但是不可修改
- LSP Liskov替换原则：子类型必须能够替换掉它们的基类型
- DIP 依赖倒置原则：抽象不依赖于细节。细节应该依赖于抽象。
- ISP 接口隔离原则：不应该强迫客户依赖于他们不用的方法，接口属于

客户，不属于它所在的类层次结构。

- REP 重用发布等价原则：重用的粒度就是发布的粒度。
- CCP 共同封闭原则： 包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对该包中的所有类产生影响，而对于其他的包不造成任何影响。
- CRP 共同重用原则：一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么就要重用包中所有的类。
- ADP 无环依赖原则： 在包得依赖关系图中不允许存在环。
- SDP 稳定依赖原则：朝着稳定的方向进行依赖。
- SAP 稳定抽象原则：包得抽象程度应该和其稳定程度一致。
-

老实讲，这些原则并不好记。在过去很多年里，我曾经一直试图记忆他们，但总是记不全。

后来在逐步实践过程中，我个人感觉也许对这些原则可以有更精练的表示，当以降低复杂度为前提对设计和编码进行推演时，似乎找到了一个答案。

事实上，支持单一职责原则，开放-封闭原则，Liskov替换原则，接口隔离原则，依赖倒置原则，共同封闭原则，共同重用原则的要求的是同样的东西，这就是我们在这里说的【概念正交分解】。【层次适度】和【信息隐藏】。也就是说，一旦概念是正交的，层次适度，信息隐藏适度，那基本上可以满足上述这些原则。

但反过来讲则不成立，满足了面向对象的设计原则，很多时候不能满足层次适度的要求。

而重用发布等价原则很特别，它是要求一种一般设计之外的东西，我们这里并没有覆盖。

无环依赖原则，稳定依赖原则，稳定抽象原则更类似于一种现象，但眼下收集的证据不够，无法对其原因是否就是违反了【概念正交分解】，【层次适度】和【信息隐藏】进行判断，也只能暂且搁置。

同时在谈及各种原则的时候，Robert C.Martin并没有探讨代价。这也许是因为作者认为遵守这些原则是稳赚不赔的。

这也许和对软件质量属性的认知有一定关系。在本书中，我们认为实现需求之外，

唯一衡量设计编码好坏的标准是代码的简单性（复杂度最低）。而灵活性这样的质量属性，是要考虑，但优先级略低。所以我们认为原则的导入必须同时考虑投入产出，即对复杂度的影响。

面向对象原则，更多的考虑的是灵活性(Robert C.Martin把僵化性排为不良设计的第一项)，即靠增加复杂度来增加灵活性，而我们这里则是考虑的简单性。

这确实是两个不同的方向，增加层次，诚然可以使软件灵活，但复杂度必然增加，很多时候这很可能是在扩散需求的边界。

也许正是因为这种视角上的不同，导致了在上述面向对象设计原则中看不到与时序相关的内容。

.4.2 完美设计和编码的关联要素

之前所论的各个环节：管理、流程、开发模型、估算、需求开发所产生的一切后果，无论好的或坏的都会在设计编码中体现出来。而设计编码对其他环节的影响则往往只能在项目结束之后。

这其中开发模型、需求开发和管理对设计和编码影响最大。

开发模型决定后，设计做到什么程度的问题将得到一定的澄清；需求则不但决定了设计编码中要做什么，有时也决定了设计编码的风格。比如：是否使用测试驱动开发，是否使用结对编程等。

设计和编码中有些问题是对错问题，有些则是选择问题。

对错问题的处理依赖于设计编码的能力，比如：增加了不必要的复杂度是一个对错问题，但能不能优化掉不必要的复杂度则依赖于当事人的能力。

选择问题的处理大多时候则只能到需求中找答案。比如说：

在特定的项目中，需要分别定义两组UI来完成对一组数据的添加和编辑工作。

这个时候的解决方案至少会有两种，一种方法是以数据操作为中心，在数据操作的类中提供显示不同UI的接口，如下：

调用Database.Add()的时候添加数据的UI将被显示出来。

调用Database.Edit()的时候编辑数据的UI将被显示出来。

在这种方法下，UI从属于数据操作。

另一种方法是把UI和数据操作进行更彻底的分离。

在对UI消息进行响应的时候，调用Database.Add()或Database.Edit()这类的方法进行内部数据操作。

第一种方法的优势在于UI是数据操作的一个部分，便于集中定义不同的事务(Transaction)。(在《敏捷软件开发：原则模式与实践》的12.5章中举了这样一个例子。)

第二种方法的优势在于把UI看成一个整体概念，并独立出来，便于对UI整体进行优化。

在这种情况下，单纯的判断那种设计方法更好是没有意义的。如果想判断那种设计更好需要的是回到这次设计的具体需求，之后才能进行判断。

如果项目中存在着这样一种变数，比如：在日后所有的UI都希望能够统一进行处理，有可能使用绘图工具来全部自画UI，并设定UI的迁移图。这个时候明显第二种方法是合适的，因为这时候，UI是作为一个独立的概念存在的，任何UI的变化都将被隔离，而不会影响到数据操作本身。

根据热力学第二定律，孤立系统总是趋向于熵增，最终达到熵的最大状态，也就是系统的最混乱无序状态。这个规则对应到软件，可以表述为：

如果我们什么都不干，只是向软件中追加或调整各种功能，那么软件必然会趋于混乱，也就是说不必要的复杂度会持续增加。

只要我们持续调整代码，那么与前一节的结论（概念正交分解）不相吻合的设计和代码必然会出现，并导致代码越来越不可维护，这其实是要求代码必须是“活”的。

也就是说，管理和流程这样的环节要保证鼓励寻找相对安全的手段（重构、自动回归测试等）对代码的修改，而非相反。否则在没有力量阻遏熵增的情形下，垃圾代码只能越来越多，修改一行代码的单价也只能越来越大。

这是一个非常有意思的两难课题。

假设说这是一个生命攸关的软件，由于担心触发不良后果，通常没人敢对代码进行优化。由此而导致的结局必然是程序越来越复杂，也就越来越容易出错，最终更容易导致不良后果。本质上看，这并非技术问题，需要在更高层面来接解决，但只是项目级很可能对此无能为力。

设计编码有问题的典型症状是：

- 程序员普遍焦虑。一旦改了点代码，很难准确知道会有什么后果。
- 没人敢改代码，也没人愿意改代码。
- 不太可能写出真的UT。
- 新人介入成本极大，往往需要了解整个程序。
- 功能无法拆装拼接。
- … …

重构与《扁鹊见蔡桓公》

重构的想法并不复杂。Martin Fowler自己这样描述重构：

所谓重构是这样一个过程：在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。

---《重构：改善既有代码的设计》

而与重构对冲的则是担心出问题，影响商业利益，所以很多时候重构启动不起来，大家只能对着垃圾代码抱怨。

这点并不难理解，再垃圾的代码只要存在，并完成了特定功能，保证了一定性能，其在用户的眼里和好的代码差别并不大。而任何对这种代码的修正则可能导致直接影响用户的不良后果。

这场景有时候会让人想起《韩非子· 喻老》中《扁鹊见蔡桓公》这篇寓言：

扁鹊第一次见蔡桓公的时候说：“你的皮肤纹理间有点病，不治会变严重地。”

蔡桓公根本不相信，说：“那有这回事，你不要胡扯。”等扁鹊走了后，蔡桓公对身边的人说：“医生就喜欢说没病的人有病，这样好显着自己有本事”。

过了十几天，扁鹊再见到蔡桓公，道：“你的病已经在肌肤里面了，不治会更严重的。”

结果蔡桓公没搭理他。

又过了十几天，扁鹊见到蔡桓公后，道：“你的病已经在肠胃里面了，不治后果会相当严重。”

结果蔡桓公相当生气。

再过了十几天，扁鹊再看到蔡桓公后，什么也不说，转身就跑了。这下蔡桓公很奇怪，忙派人去问他跑什么。扁鹊道：“你们老板的病，之前是治起来麻不麻烦的问题，所以我提醒下还有意义。现在是神仙来了都没办法，我不跑还能怎么地？”

再过了五天，蔡桓公病发了，再找扁鹊，扁鹊已经跑到秦国去了。于是蔡桓公只好去世了。

程序的最初版本往往很像文中得了小病的蔡桓公，大多时候是一定有点问题的。如果没法开启重构，代码一定会变成垃圾代码，而上规模的代码垃圾到一定程度后是完全没法改的，很像文中的病入膏肓的蔡桓公，你只能看着它慢慢死去---只是维护他的人要跟着葬送很多时光。而重构本身并不难，难的是启动重构，并持续做下去，这并非是技术问题。